Data Structure & Algorithm Using C

Debottam Das

LIST OF CONTENTS

LIST OF CONTENTS

LIST OF CONTENTS

LIST OF CONTENTS

LIST OF CONTENTS

LIST OF CONTENTS

LIST OF CONTENTS

LIST OF CONTENTS

SYLLABUS

Module 1	**Introduction:** Basic Terminologies: Elementary Data Organizations, Data Structure Operations: insertion, deletion, traversal etc.; Analysis of an Algorithm, Asymptotic Notations, Time-Space trade off.
	Searching: Linear Search and Binary Search Techniques and their complexity analysis.
Module 2	**Stacks:** ADT Stack and its operations: Algorithms and their complexity analysis, Applications of Stacks: Expression Conversion and evaluation - corresponding algorithms and complexity analysis.
	Queues: ADT queue, Types of Queue: Simple Queue, Circular Queue, Priority Queue; Operations on each types of Queues: Algorithms and their analysis.
Module 3	**Linked Lists:** Singly linked lists: Representation in memory, Algorithms of several operations: Traversing, Searching, Insertion into, Deletion from linked list; Linked representation of Stack and Queue, Header nodes, Doubly linked list: operations on it and algorithmic analysis; Circular Linked Lists: all operations their algorithms and the complexity analysis.
	Trees: Basic Tree Terminologies, Different types of Trees: Binary Tree, Threaded Binary Tree, Binary Search Tree, AVL Tree; Tree operations on each of the trees and their algorithms with complexity analysis. Applications of Binary Trees. B Tree, B+ Tree: definitions, algorithms and analysis.
Module 4	**Sorting and Hashing:** Objective and properties of different sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort; Performance and Comparison among all the methods, Hashing.
	Graph: Basic Terminologies and Representations, Graph search and traversal algorithms and complexity analysis.

Chapter 1

Introduction

What is data ?

The quantities, characters or symbols on which operations are performed by a computer, which may be stored in the form of electrical signals and recorded on a magnetic, optical or mechanical recording media are called data.

Example: c = a + b. Here a and b both are data.

What is information ?

If data is arranged in a systematic way, then it gets structured and become meaningful. This meaningful processed data is called information.

Data: nuraB si eman yM.

Information: My name is Barun.

1.1 Data Types: Data type defines the type of data which a variable store in the memory. For example an integer variable 'a' is declared by the following statement.

int a;

The above statement implies that the variable 'a' can store integer value in the memory.

Two important points about data types:
1. It defines the certain domain of values.
2. It defines the operations allowed on those values.

In case of int type of variable it takes only integer values which satisfies the first criterion. More over it allows to perform the mathematical operations like addition, subtraction, multiplication, division, mod, bit-wise operation etc.
In case of float type of variable it takes only floating point values and allows the mathematical operations like addition, subtraction, multiplication, division etc. But it does not support bit-wise operations and mod (%) operations.

Classification of data types: There are two types of data types – 1) Primitive data type and 2) Non-primitive data type.

1) **Primitive data type –** Primitive data types or basic data types are the fundamental data types which are supported by a programming language. In case of C language the primitive data types are integer (declared by the keyword "int"), real (declared by the keyword "float"), character (declared by the keyword "char"), Boolean (declared by the keyword "bool") etc. In c the Boolean data type is used by importing the header file with "#include <stdbool.h>".

2) **Non-primitive data type –** Non-primitive data types are those data types which are created using primitive data types. Non-primitive data types are classified into two categories – a) User defined data type and b) abstract data type.

1. **User defined data type –** The operations and values of user defined data types are not specified in the language, but it is specified by the user.
   Example – structure, union and enumeration.
   By using structure we define our own data type by combining other primitive data types.
   In the following example we are creating our own data type i.e. Cartesian point with the help of other two integer types.

```
struct point
{
    int x;
    int y;
};
```

2. **Abstract data type (ADT) –** ADTs are user defined data types which defines operations on values by using user defined functions without specifying the inner details of the functions and how the operations are performed. Basically ADT focuses on "What to do?" hiding "How to do?" from the user. The word 'Abstract' in the context of ADT means considered apart from the detailed specifications or implementations. Therefore in case of ADT, the end-user is not concerned about the details of how the methods are implemented using the functions, they are only aware of how to use the method and get the result of that method.

   For example, stack and queue are the perfect example of ADT. In case of stack the user only knows the type of data that can be handled by the stack and the operations that can be performed using the stack. The following operations can be performed by the stack.

   - Push() – insert an element into the stack
   - Pop() – delete an element from the stack
   - isFull() – check whether the stack is full or not
   - isEmpty() – check whether the stack is empty or not

   In the above four operations the implementation details of the functions are hidden to the users, only the use of the functions are known to the user.

   There are multiple ways to implement an ADT. For example – an stack ADT can be implemented either using arrays or linked lists.

   Advantages: The program that uses the data structure is called client program. It has access to the ADTs i.e. interface. The program which implements the data structure is called the implementation program. Normally ADTs are used to separate the use of the data structure from the details of its implementation. In case of stack if someone wants to use it, he only needs to know the use of Push and Pop operations in the client program without knowing its implementation. Moreover if any modification of the data

structure is required, it is done in the implementation part only. There is no change in the ADT of the client program. For example – if the stack is implemented by using linked list instead of arrays, there will be modification or alteration in the functions of Push and Pop operations only. This reduces the complication of the use the data structure.

What is data structure ?

Data structure is the organization, management and storage of data in such a way that it can be accessed and modified efficiently. Here efficiency will be considered both in terms of time and space. That means in data structure the efficiency can be assessed by considering time efficiency as well as space efficiency. In other words, the data structure is used to implement an ADT. ADT tells us what is to be done and data structure tells us how to do it.

For example – array is a data structure. Suppose we are interested to store 100 integers. To store 100 integers we need 100 variables of integer data type. But it is not the efficient way to store 100 integers in this way, rather it is better to use an array of integer data type of size 100 to store all the values sequentially in memory. This will give us the better solution to handle 100 integers efficiently. Due to this reason the array is surely a data structure.

Now the question comes "which data structure should be used for a particular ADT ?". In reality, the different implementations of ADT are compared for time and space efficency. The one best suited according to the current requirement of the user will be selected. For example – stack ADT may be implemented using arrays or linked list. If array data structure provides better space efficiency and the user requires space efficiency, then stack ADTs are implemented by using arrays. On the contrary if time efficiency is preferred, the stack ADTs are implemented by using linked list.

Advantages of data structure:

1. Efficiency: Proper choice of the data structures makes the program efficient in terms of space and time.

2. Re-usability: One implementation can be used by multiple client program.

3. Abstraction: Data struc ture is specified by an ADT which provides a level of abstraction. The client programs does not have to worry about the implementation details.

Different types of data structure: There are two types of data structure. 1) Linear data structure and 2) Non-linear data structure.

1. Linear data structure – In a linear data structure, the elements are arranged in sequence i.e. one after other. That means every element of a linear data structure has one predecessor and one successor except the first and last element. In case of first element there is one successor, but no predecessor. In case of last element there is one predecessor, but no successor. Arrays, stacks, queues and linked lists are the examples of linear data structure.

a) Array – An array is a static data structure where the elements of same data type are stored in consecutive memory locations and each elements are accessed by an index. In C, arrays are declared using the following syntax.

data_type ArrayName[size];
For example: int marks[5];

The above statement declares an array named marks that contains 5 elements which are stored in 5 consecutive memory locations as shown in Fig.1.1.

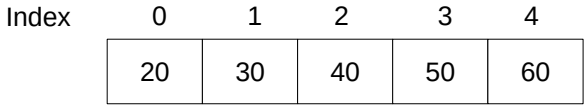


Fig.1.1: An array with five elements

In Fig.1.1 it is being observed that every element of the array has its predecessor and successor except 1st element and 5th element. That's why array is a linear data structure.

Limitation of Array:

1) Arrays are declared as fixed size. If it is required to handle more number of elements than the size of the array, it is not possible to do that. Due to this reason array is referred as static data structure.

2) Data elements of an array are stored in continuous memory locations which may not be available always.

3) Insertion and deletion of element in case of array becomes problematic because of shifting of elements from their positions.

b) Linked List – A linked list is a very flexible, dynamic data structure in which elements (called nodes) are connected to each other in a sequential manner. In this case each node of a linked list has two parts, one part contains the data and other part contains the address of the next node. Therefore the part of the node which contains the address of the next node, becomes the pointer of the next node. A pictorial view of a linked list is shown in Fig.1.2.

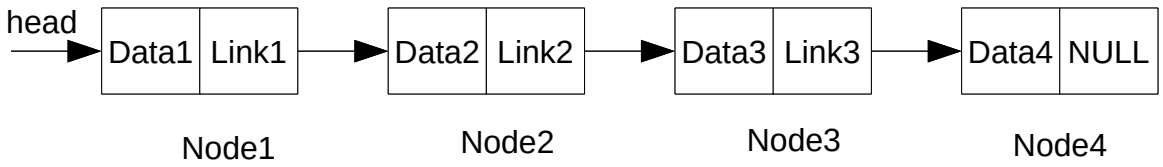


Fig.1.2: A linked list with four nodes whereas Node1 is pointed by a head pointer

In the above figure it is being seen that every node has its predecessor node and successor node except node1 and node4. Therefore linked list is a linear data structure. In case of a linked list the first node is pointed by a head pointer and the last node of the list contains NULL pointer to indicate that it is the end of the linked list.

Advantages of linked list:

1) All the nodes of a linked list are created dynamically during the execution of the program. This feature of the linked list gives the facility to the user to add as many nodes as he wants as per his requirement. Basically there is no limitation of nodes to be added in a linked list like array.

2) Insertion and deletion of a node from a linked list is easier than array data structure. Here shifting of the elements is not required like array which reduces the execution time of a program.

c) Stack – A stack is linear data structure in which insertion and deletion of elements are done at only one end which is known as top of the stack.Stack is called a last-in, first-out (LIFO) data structure because the last element which is added to the stack is the first element which is deleted from the stack. A pictorial view of a stack is shown in Fig.1.3.

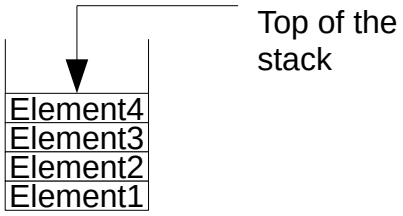


Fig.1.3: A stack with four elements

A stack can be implemented either by using arrays or by using linked lists. The details of the implementation will be discussed later. A stack supports three basic operations: push, pop and peep.

1) The push operation adds an element to the top of the stack. However, before inserting an element in the stack, we must check for overflow conditions. An overflow occurs when we try to insert an element into a stack that is already full.

2) The pop operation removes the element from the top of the stack. Before deleting an element from the stack, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a stack that is already empty.

3) The peep operation returns the value of the topmost element of the stack (without deleting it).

d) Queue – A queue is a linear data structure where the element inserted first will be taken out first. Due to this property of the queue it is called first-in, first-out (FIFO) data structure. The elements in a queue are added at one end called rear and removed from the other end called front. Like stack queue can also be implemented by arrays or linked lists. The pictorial view of a queue is shown in Fig.1.4.

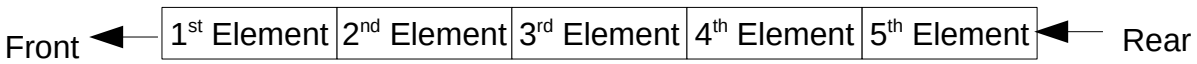


Fig.1.4: A queue with five elements

2. Non-linear data structure – Unlike linear data structure, the elements in a non-linear data structure are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements. Trees and graphs are the examples of non-linear data structure.

a) Tree – A tree is a data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

The simplest form of a tree is a binary tree. A binary tree consists of a root node and left and right sub-trees, where both sub-trees are also binary trees. Each node contains a data element, a left pointer which points to the left sub-tree, and a right pointer which points to the right sub-tree. The root element is the topmost node which is pointed by a 'root' pointer. If root = NULL then the tree is empty.

Figure 1.5 shows a binary tree, where R is the root node and T1 and T2 are the left and right sub-trees of R . If T1 is non-empty, then T1 is said to be the left successor of R . Likewise, if T2 is non-empty, then it is called the right successor of R .

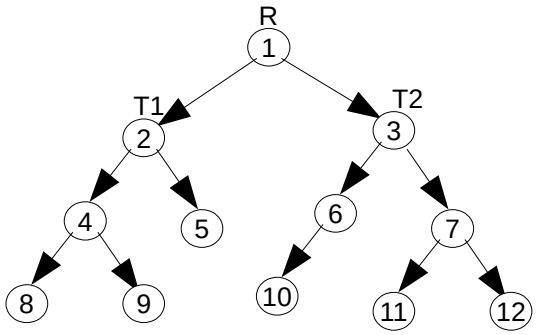


Fig.1.5: A binary tree with 12 nodes whereas node 1 is the root and T1 and T2 are the left and right sub-trees of the root node

In the above binary tree data structure some nodes have one predecessor and two successor which contradicts the arrangement of linear data structure. That's why binary tree is a non-linear data structure.

b) **Graph –** A graph is a non-linear data structure which is a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.

In a tree structure, nodes can have any number of children but only one parent, a graph on the other hand relaxes all such kinds of restrictions. Figure 1.6 shows a graph with five nodes.

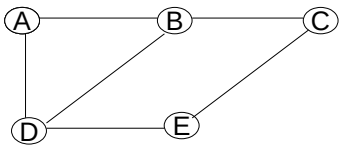


Fig.1.6: A graph with five vertices A, B, C, D and E

A node in the graph may represent a city and the edges connecting the nodes can represent roads. A graph can also be used to represent a computer network where the nodes are workstations and the edges are the network connections. Graphs have so many applications in computer science and mathematics that several algorithms have been written to perform the standard graph operations, such as searching the graph and finding the shortest path between the nodes of a graph.

Note that unlike trees, graphs do not have any root node. Rather, every node in the graph can be connected with every another node in the graph. When two nodes are connected via an edge, the two nodes are known as neighbours. For example, in Fig. 2.8, node A has two neighbours: B and D.

Chapter 2

Time and Space Complexity

Analyzing an algorithm means determining its efficiency i.e. determining the amount of resources (time and space) needed to execute it. Efficiency of an algorithm is measured in terms of time and space. Efficiency of an algorithm in terms of time is determined by time complexity. Similarly efficiency of an algorithm in terms of space is measured with the help of space complexity.

- **Space complexity –** The space complexity of an algorithm is the amount of memory space which is required during the program execution as a function of the input size. Generally space required by a program depends on the way it has been implemented. That means it depends on the type of data structure. If it is implemented by using instructions, constants, variables and static data structures like arrays, structures, it constitutes fixed space. If the program is implemented by using recursions and dynamic data structures like linked lists, it consumes variable memory space depending upon the input size. But minimum execution time of a program is more important now-a-days compared to the memory space. Due to this reason, calculation of time complexity is more crucial for our discussion.

- **Time complexity –** The time complexity of an algorithm is basically the running time of a program as a function of the input size. The running time of a program is proportional to the number of instructions it executes and the number of instructions which a program executes depends on the program's input size. Therefore the time complexity depends on the input size of a program. If f(n) represents the number of instructions executed for the input value n, then f(n) gives the time complexity of the program. Let's consider the following example to explain how to determine f(n).

```
for(i=0; i<n; i++)
    printf("Hello World\n");
```

Let's assume that the printf instruction takes 1 unit of time. In the above case the printf instruction will be executed for n no. of times. This will take n units of time to be executed. Therefore here f(n) = n.

We can compare two data structures for a particular operation by comparing their f(n) values. Depending on the f(n) values, one data structure is selected for that particular operation considering the time complexity. Suppose we have an array with 10 elements as shown in Fig.2.1.

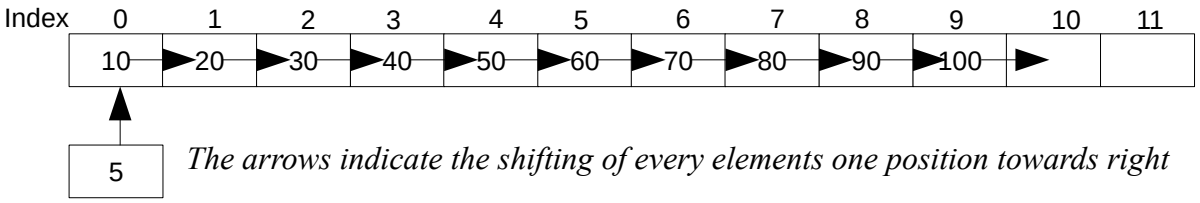


Fig.2.1: Insertion of an element 5 at the 1st position of an array with 10 elements

The element 5 is to be inserted at the 1st position of the array. Here we need to shift right the every elements. Therefore 10 no. of right shifts are required to accommodate the element 5 at the 1st position. If one shift takes 1 unit of time, then 10 shifts will take 10 units of time.

Hence f(n) = 10 + 1 (unit of time required to insert the new element). On the other hand, if we have a linked list with 10 nodes which consist of 10 elements, we do not require any right shift operation to place an element at the beginning of the linked list. Therefore it takes only 1 unit of time to place the new element at the beginning of the list which makes f(n) = 1. So obviously we should choose the linked list data structure as it has lower value of f(n) for the insertion operation.

An ideal data structure is one which takes the least possible time for all its operations and consumes the least memory space. But practically designing such ideal data structure is not an easy task. There can be more than one algorithm to solve a particular problem. One may require less memory space to store the program and other may require less CPU time to execute. Depending upon the criteria, where space is the main constraint, the program which takes less memory space is chosen. On the contrary where minimum CPU time is the main criterion, the program which takes less time to be executed is selected.

Calculation of f(n) – For determining the value of f(n) we are interested in growth rate of f(n) with respect to n because it might be possible that for smaller input size, one data structure may seem better than the other data structure, but for larger input size it may not. This concept is applicable in comparison between two algorithms as well. To determine the value of f(n) it is more important to find out which term in the expression of f(n) is contributing much in the total running time of an algorithm. The value of f(n) is approximated to the term which is dominant in the execution time of the algorithm. This approximate measure of time complexity by determining the approximate value of f(n) is called ***asymptotic complexity***. Let's take an example.

Suppose $f(n) = 5n^2 + 6n + 12$

For n = 1,

% of running time due to $5n^2$ = $\frac{5}{5+6+12} \times 100 = 21.74\%$

% of running time due to 6n = $\frac{6}{5+6+12} \times 100 = 26.09\%$

% of running time due to 12 = $\frac{12}{5+6+12} \times 100 = 52.17\%$

The percentage contributions of the terms $5n^2$, 6n and 12 in the total execution time are given in Table.2.1 for different values of n.

Table.2.1

n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

It is being observed in Table.2.1 that with the larger value of n the term $5n^2$ is contributing maximum in the running time and the effects of other two terms are negligible. Therefore we can approximate the value of f(n) to $5n^2$ neglecting the other two terms 6n and 12. As per the rules of asymptotic complexity $f(n) = 5n^2$.

Big O Notation – Big O notation is used to measure the performance of any algorithm by providing the order of growth of the function f(n) where f(n) gives the running time of the algorithm. In other words, it gives the *upper bound* on a function by which it can be ensured that the function will never grow faster than the upper bound. Asymptotic complexity gives the measure of approximate running time to determine the time complexity of a program. The Big O notation helps to achieve the approximate running time.

If f(n) and g(n) are the two functions, then f(n) = O(g(n)), where $0 \leq f(n) \leq c\ g(n)$ for $n \geq n_0$, if and only if the constants c and n_0 exist. If the condition $f(n) \leq c\ g(n)$ is satisfied for a finite values of c and n_0, then we can say that f(n) will not grow faster than c g(n) for any value of n which is greater or equal to n_0. As a result g(n) becomes the upper bound of f(n) beyond n_0. Therefore this concept gives us the worst case time complexity which says how worst a particular algorithm can perform. The pictorial representation of Big O notation is given in Fig.2.2 below.

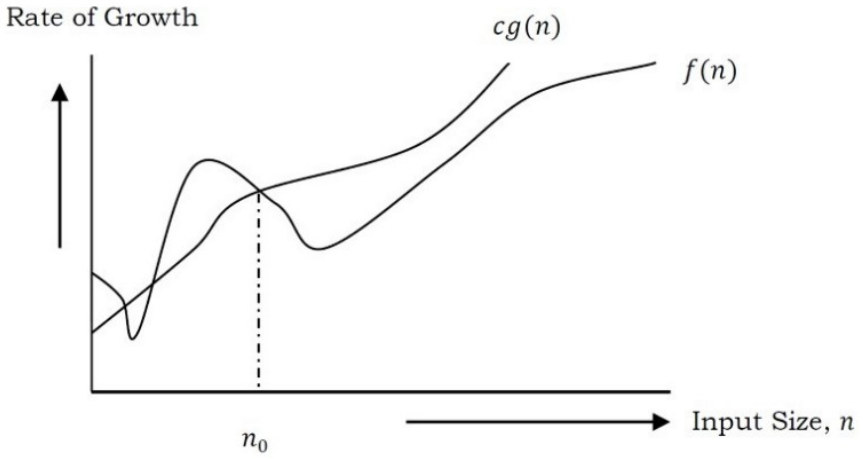


Fig.2.2: Pictorial representation of Big O notation

Calculation of Big O notation – For the calculation of Big O it is required to find out the expression of f(n) first for an algorithm. Now using the expression of f(n) and assuming the expression of g(n) we have to determine the values of the constants c and n_0 for which the condition $f(n) \leq c\ g(n)$ is satisfied. Once this condition is proved, then g(n) will give the Big O for that particular algorithm. To clarify this concept some examples are given below.

Example1: If f(n) = n and g(n) = 2n, then check f(n) = O(g(n) or not.

We know, $f(n) \leq c\ g(n)$

or, $n \leq c\ 2n$

or, $1 \leq 2c$ [$\because n \neq 0$]

or, $c \geq 0.5$

$\therefore c = 1$

Therefore $f(n) \leq c\ g(n)$ for $n \geq n_0 = 1$ and $c = 1$. According to the definition of Big O, we can write $f(n) = O(2n)$. Now we can ignore the coefficient 2 in the expression g(n) which does not affect the relation $f(n) \leq c\ g(n)$. Hence $f(n) = O(n)$ which implies that the growth rate of f(n) is linear.

Example2: If f(n) = 4n + 3 and g(n) = n, then check f(n) = O(g(n) or not.

We know, $f(n) \leq c\ g(n)$

or, $4n + 3 \leq c\ n$

Now to satisfy the above inequality c should be greater than 4. So, we have assumed c = 5. Putting c = 5 we get,

$4n + 3 \leq 5n$
or, $3 \leq 5n - 4n$
$\therefore \quad n \geq 3$

Therefore $f(n) \leq c\ g(n)$ for $n \geq n_0 = 3$ and $c = 5$. $\therefore f(n) = O(n)$ which implies that the growth rate of f(n) is linear. If we compare the growth rates of f(n) and g(n) by plotting f(n) and g(n) in Fig.2.3, it is being clearly observed that f(n) is always less than g(n) beyond n = 3. Therefore g(n) becomes upper bound on f(n) for $n \geq 3$.

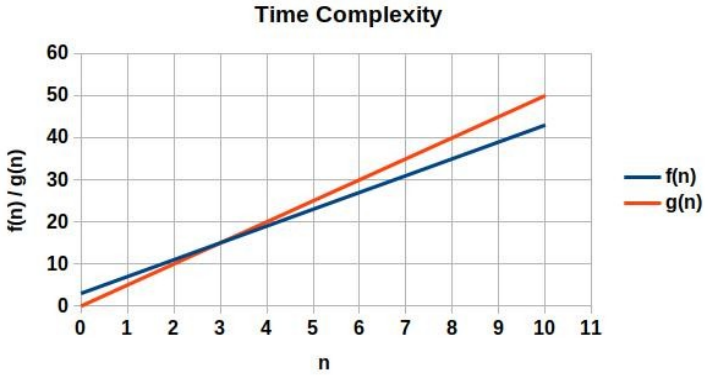


Fig.2.3: Plotting the growth rate of two functions f(n) and g(n)

In the above figure it is being seen that f(n) will never grow faster than c g(n) for $n \geq 3$. Again it is also being noticed that f(n) is larger than c g(n) for $n < 3$. Therefore the value of n_0 is important. Therefore it is a very tedious job to determine the value of n_0 without the help of Big O notation. Big O notation gives the easier way to determine n_0 which gives the worst case time complexity beyond a certain limit of n.

Example3: If $f(n) = 5n^2 + 4$ and $g(n) = n^2$, then check f(n) = O(g(n) or not.

We know, $f(n) \leq c\ g(n)$

or, $5n^2 + 4 \leq c\ n^2$

or, $5n^2 + 4 \leq 6n^2$ (Assuming c = 6)

or, $4 \leq n^2$

$\therefore n \geq 2$

Therefore $f(n) \leq c\ g(n)$ for $n \geq n_0 = 2$ and c = 6. $\therefore f(n) = O(n^2)$ which implies that the growth rate of f(n) is quadratic.

Let's consider the growth rates of some standard functions which are normally used for Big O notation in the following table.

Table.2.2

	g(n)						
n	1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	1	0	1	0	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4096	65536
32	1	5	32	160	1024	32768	429×10^7

In the above table it is clearly seen that g(n) = 1 provides best time complexity whereas $g(n) = 2^n$ gives the worst time complexity. The above functions are some standard functions used in Big O notation.

Example4: The following program calculates the sum of first n natural numbers.

```
int main()
{
    int i, n, sum = 0;
    scanf("%d", &n);
    for(i=1; i<=n; i++)
        sum = sum + i;
    printf("%d", sum);
    return 0;
}
```

Find the time complexity of the above program.

Program Statements	Time required
int main()	
{	
int i, n, sum = 0;	← 1 time
scanf("%d", &n);	← 1 time
for(i=1; i<=n; i++)	
sum = sum + i;	← n times
printf("%d", sum);	← 1 time
return 0;	← 1 time
}	

$\therefore f(n) = n + 4$

Let $g(n) = n$. Putting $f(n) = n + 4$ and $g(n) = n$ we get,

$f(n) \leq c\ g(n)$

or, $n + 4 \leq c\ n$

or, $n + 4 \leq 2n$ (Assuming $c = 2$)

$\therefore n \geq 4$

Therefore $f(n) \leq c\ g(n)$ for $n \geq n_0 = 4$ and $c = 2$. $\therefore f(n) = O(n)$ which implies the linear growth. But the above mentioned program's time complexity may be improved by modifying it. If we use the formula (sum = $\frac{n(n+1)}{2}$) to calculate the sum of first n natural numbers in the program, the time complexity will be reduced significantly. The modified program is given below.

Program Statements	Time required
int main()	
{	
int n, sum = 0;	← 1 time
scanf("%d", &n);	← 1 time
sum = (n*(n + 1)) / 2;	← 1 time
printf("%d", sum);	← 1 time
return 0;	← 1 time
}	

$\therefore f(n) = 5$

Let $g(n) = 1$. Putting $f(n) = 5$ and $g(n) = 1$ we get,

$$f(n) \leq c\ g(n)$$
$$\text{or, } 5 \leq c \cdot 1$$
$$\therefore c \geq 5$$

Therefore $f(n) \leq c\ g(n)$ for $n \geq n_0 = 1$ and $c = 6$. $\therefore f(n) = O(1)$ which implies the constant growth. In this case the execution time does not depend of the input size n. So, this algorithm becomes the fastest algorithm. Hence it is clear from the above program that the time complexity of the program has been improved from O(n) to O(1).

- **Big O for linear loop** – Let's consider a for loop as given below.

```
for(i=0; i<n; i++)
{
    //Statement
}
```

In this case the statement will be executed for n times. If the statement takes 1 unit of time to be executed, then the entire for loop will take n units of time. Therefore the time complexity will be O(n).

- **Big O for quadratic loop** – Let's consider the following nested for loops.

```
for(i=0; i<n; i++)
{
    for(j=0; j<n; j++)
    {
        //Statement
    }
}
```

In this case the inner for loop (with variable j) executes the statement for n times for a single execution of the outer for loop (with variable i). As the outer for loop executes for n time also, then the statement under the inner for loop will be executed for nxn i.e. n^2 times. Hence the time complexity will be $O(n^2)$.

- **Big O for consecutive statements** – Consider a part of program which consists of some instructions, one for loop and another nested for loops.

Program statements	Execution time

int x = 2;
int i;
x = x + 1;
	3 units of time

for(i=0; i<n; i++)
{
//Statement
}
	n units of time

for(i=0; i<n; i++)
{
for(j=0; j<n; j++)
{
//Statement
}
}
	n^2 units of time

Therefore the entire program will take a time $f(n) = n^2 + n + 3$. Let $g(n) = n^2$.

Now, $f(n) \leq c\ g(n)$

or, $n^2 + n + 3 \leq c\ n^2$

or, $n^2 + n + 3 \leq 2n^2$ (Assuming c = 2)

or, $n^2 - n - 3 \geq 0$

or, $(n - 2.3)(n + 1.3) \geq 0$

Now for $n > 0$ the factor $(n + 1.3) > 0$. Obviously $(n - 2.3) > 0$ i.e. $n > 2.3$.

Therefore $f(n) \leq c\ g(n)$ for $n \geq n_0 = 3$ and $c = 2$.
So, the time complexity will be $f(n) = O(n^2)$.

- **Big O for if then else statements** – Consider a part of program which consists of if-else statements.

Program statements	Execution time
scanf("%d", &n);	1 unit of time

if(n == 0)
{
//statement
}
	If (n==0) condition satisfied then this portion of codes will be executed. To check the condition 1 unit of time will be consumed and the statement under if portion will consume another unit of time. Therefore total time consumed for this if portion is 2 units of time.

else
{
for(i=1; i<=n; i++)
//statement
}
	If (n==0) condition is not satisfied, the else part will be executed. To check the condition 1 unit of time will be consumed and n units of time will be consumed for the execution of the for loop under else part. Therefore total time consumed for this else part is (n+1).

From the above discussion it is being observed that time consumed for the else part is larger than the time consumed for the if part. Hence to determine the worst case time complexity we have to calculate the total time taken by the program when it executes else part. Therefore in worst case the total time consumed by the program is (1 + n + 1) i.e. (n + 2). Thus the worst case time complexity of the program will be O(n).

- **Big O for logarithmic loop** – Consider the following piece of codes.

```
for(i=1; i<=n; i= i*2)
    //statement
```

In this case i is not incremented by 1 for every iteration rather i is doubled every time. That's why the statement under the for loop will not be executed for n times here. How many times the loop will be repeated can be evaluated as follows.

Iterations	Value of i
Iter 1	Initially i = 1 = 2^0
Iter 2	i = 2 = 2^1
Iter 3	i = 4 = 2^2
Iter 4	i = 8 = 2^3
:	:
Iter k	i = 2^{k-1} = n $\therefore$ k = $\log_2 n + 1$

Therefore time complexity will be $O(\log_2 n)$.

Consider the another case where i is halved for every iteration of the loop as shown below.

```
for(i=n; i>=1; i= i/2)
    //statement
```

Iterations	Value of i
Iter 1	Initially $i = n = n/2^0$
Iter 2	$i = n/2 = n/2^1$
Iter 3	$i = n/4 = n/2^2$
:	:
Iter k	$i = n/2^{k-1} = 1$ $\therefore k = \log_2 n + 1$

Therefore time complexity will be $O(\log_2 n)$.

- **Big O for linear logarithmic loop –** Consider the following part of program.

```
for(i=1; i<=n; i++)
    for(j=1; j<=n; j = j*2)
        //statement
```

In this case the inner loop executes for $(\log_2 n + 1)$ times whereas the outer loop executes for n times. Therefore the statement under the inner loop executes for $n(\log_2 n + 1)$ times which gives the time complexity as $O(n\log_2 n)$.

- **Big O for dependent quadratic loop –** Consider the following program.

```
for(i=1; i<=n; i++)
    for(j=1; j<=i; j++)
        //statement
```

In this case for i=1 of the outer loop the inner loop executes for 1 time, for i=2 the inner loop executes for 2 times, for i=3 the inner loop executes for 3 times and so on as given in the following table.

Outer loop's iterations	Inner loop's iterations	No. of executions of inner loop
For i =1	Executes from j = 1 to 1	Executes for 1 time
For i = 2	Executes from j = 1 to 2	Executes for 2 times
For i = 3	Executes from j = 1 to 3	Executes for 3 times
:	:	:
For i = n	Executes from j = 1 to n	Executes for n times

Therefore total no. of execution = 1 + 2 + 3 + 4 + ……. + n = $\frac{n(n+1)}{2}$ which gives the time complexity $O(n^2)$.

Problem1: *What is the time complexity of the following program and what will be the value of count after the execution of the program ?*

```
void fun(int n)
{
        int i, j, k, count = 0;
        for(i=n/2; i<=n; i++)
                for(j=1; j+n/2<=n; j++)
                        for(k=1; k<=n; k=k*2)
                                count++;
}
```

Ans: The first statement "int i, j, k, count = 0;" takes 1 unit of time. Therefore time complexity for this statement = O(1).

Now we can see there are three nested for loops with variables i, j and k. Under the inner-most for loop we have a statement "count++". We have to find out how many times the statement "count++" will be executed to determine the time complexity of this three nested loops.

Iterations	Value of i
Iter 1	i = n/2 + 0
Iter 2	i = n/2 + 1
Iter 3	i = n/2 + 2
Iter 4	i = n/2 + 3
:	:
Iter k	i = n/2 + (k – 1) = n ∴ k = n/2 + 1

So this for loop with variable i will be executed for (n/2 + 1) times which provides the time complexity of O(n).

Iterations	Value of j
Iter 1	j = 1
Iter 2	j = 2
Iter 3	j = 3
Iter 4	j = 4
:	:
Iter k	j = k ∴ k + n/2 = n ∴ k = n/2

As this for loop with variable j executes for n/2 times, the time complexity for this loop becomes O(n).

Iterations	Value of k
Iter 1	$k = 1 = 2^0$
Iter 2	$k = 2 = 2^1$
Iter 3	$k = 4 = 2^2$
Iter 4	$k = 8 = 2^3$
:	:
Iter p	$k = 2^{k-1} = n$ $\therefore k = \log_2 n + 1$

Therefore the time complexity for this inner-most loop is $O(\log_2 n)$.

Now the total time complexity of these three nested loops will be the product of each Big O i.e. $O(n^2 \log_2 n)$.

Again if the statement "count++" will be executed for x times, then we can write,

$$x = (n/2 + 1) \cdot n/2 \cdot (\log_2 n + 1) = \frac{1}{4} n(n+2)(\log_2 n + 1)$$

After 1 time execution the value of count will be 1. Therefore after the execution of the program count = $\frac{1}{4} n(n+2)(\log_2 n + 1)$.

Problem2: *What is the time complexity of the following program ?*

```
void fun(int n)
{
        int i, j;
        if(n <= 1)
                return;
        for(i=1; i<=n; i++)
                for(j=1; j<=n; j++)
                {
                        printf("Hello\n");
                        break;
                }
}
```

Ans: In the above program the first statement always runs for one time. At the second statement if the condition (n <= 1) is satisfied, then program returns to its calling function. Therefore if the condition is satisfied, the program takes 2 units of time. Thus the time complexity of the program becomes O(1) if the condition is true.

When the condition is false then it will execute the rest part of the program which consists a nested for loop – one inner loop with variable j and one outer loop with variable i. Now due to the presence of break statement under the inner loop, the inner loop always executes for one time. As a result for every value of i (during the execution of outer loop) from i=1 to n the inner loop executes for one time always. Therefore the entire nested loop prints "Hello" n no. of times. Hence the program executes for (n + 2) no. of instructions consuming (n + 2) units of time if the condition fails. In this way it will provide the time complexity of O(n).

Now we should consider the worst time complexity which is the largest one among O(1) and O(n). Ultimately the time complexity of the entire program becomes O(n).

Problem3: *What is the time complexity of the following program ?*

```
void fun(int n)
{
        int i, j;
        i = 1;
        while(i < n)
        {
                j = n;
                while(j >= 1)
                        j = j/2;
                i = 2 * i;
        }
}
```

Ans:

Iterations	Value of i
Iter 1	$i = 1 = 2^0$
Iter 2	$i = 2 = 2^1$
Iter 3	$i = 4 = 2^2$
Iter 4	$i = 8 = 2^3$
:	:
Iter k	$i = 2^{k-1} = n - 1$ $\therefore k = \log_2 (n-1) + 1$

Therefore the time complexity of the outer loop is $O(\log_2 n)$.

Iterations	Value of j
Iter 1	$j = n = n/2^0$
Iter 2	$j = n/2 = n/2^1$
Iter 3	$j = n/4 = n/2^2$
Iter 4	$j = n/8 = n/2^3$
:	:
Iter k	$j = n/2^{k-1} = 1$ $\therefore k = \log_2 n + 1$

The time complexity of the inner loop is $O(\log_2 n)$. The overall time complexity of the program will become $O(\log_2^2 n)$.

Problem4: *What is the time complexity of the following program ?*

```
void fun(int n)
{
        int i, j;

        for(i=1; i<=n/3; i++)
                for(j=1; j<=n; j = j+4)
                        printf("Hello\n");
}
```

Ans:

Iterations	Value of i
Iter 1	i = 1
Iter 2	i = 2
Iter 3	i = 3
Iter 4	i = 4
:	:
Iter k	i = k = n/3 $\therefore$ k = n/3

So the time complexity of the outer loop is O(n).

Iterations	Value of j
Iter 1	j = 1 = 1 + 0x4
Iter 2	j = 5 = 1 + 1x4
Iter 3	j = 9 = 1 + 2x4
Iter 4	j = 13 = 1 + 3x4
:	:
Iter k	j = 1 + (k-1)x4 = n $\therefore k = \frac{n-1}{4} + 1 = \frac{n+3}{4}$

Here the time complexity of the inner loop will be O(n). As a result the overall time complexity of the program will be $O(n^2)$.

Problem5: *What is the time complexity of the following program ?*

```
void fun(int n)
{
        int i=1, s=1;
        while(s <= n)
        {
                i++;
                s = s + i;
                printf("*");
        }
}
```

Ans:

Iterations	Value of s	Value of i
Iter 1	s = 1	i = 2
Iter 2	s = 1 + 2 = 3	i = 3
Iter 3	s = 1 + 2 + 3 = 6	i = 4
Iter 4	s = 1 + 2 + 3 + 4 = 10	i = 5
:		
Iter k	s = 1 + 2 + 3 + 4 + ... + k = n or, $s = \frac{k(k+1)}{2} = n$ or, $k^2 + k = 2n$ $\therefore k^2 + k - 2n = 0$	i = k + 1

Solving the quadratic equation we get, k = $\frac{-1 \pm \sqrt{1+8n}}{2}$

Now no. of iterations k is always positive.

$\therefore$ k = $\frac{\sqrt{8n+1}-1}{2}$

The time complexity will be O($\sqrt{n}$).

Problem6: *What is the time complexity of the following program ?*

```
void Read(int n)
{
        int k = 1;
        while(k < n)
                k = 3*k;
}
```

Ans:

Iterations	Value of k
Iter 1	k = 1 = 3^0
Iter 2	k = 3 = 3^1
Iter 3	k = 9 = 3^2
Iter 4	k = 27 = 3^3
:	:
Iter p	k = 3^{p-1} = n – 1 or, p – 1 = $\log_3$ (n-1) $\therefore$ p = $\log_3$ (n-1) + 1

The time complexity of the function will be O($\log_3$ n)

Problem7: *What is the time complexity of the following program ?*

```
void function(int n)
{
        int i,j;
        for(i=1; i<=n; i++)
                for(j=1; j<=n; j+=i)
                        printf("*");
}
```

Ans:

<table>
<tr><th>Iterations</th><th>Value of i</th><th>j loop execution</th><th>No. of times j loop executed</th></tr>
<tr><td>Iter 1</td><td>i = 1</td><td>
<table>
<tr><td>Iteration</td><td>Value of j</td></tr>
<tr><td>Iter 1</td><td>j = 1</td></tr>
<tr><td>Iter 2</td><td>j = 2</td></tr>
<tr><td>Iter 3</td><td>j = 3</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Iter p</td><td>j = p = n</td></tr>
</table>
</td><td>n</td></tr>
<tr><td>Iter 2</td><td>i = 2</td><td>
<table>
<tr><td>Iteration</td><td>Value of j</td></tr>
<tr><td>Iter 1</td><td>j = 1 + 0x2</td></tr>
<tr><td>Iter 2</td><td>j = 1 + 1x2</td></tr>
<tr><td>Iter 3</td><td>j = 1 + 2x2</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Iter p</td><td>j = 1 + 2(p-1) = n
$\therefore$ p = $\frac{n-1}{2}+1$</td></tr>
</table>
</td><td>$\frac{n-1}{2}+1$</td></tr>
<tr><td>Iter 3</td><td>i = 3</td><td>
<table>
<tr><td>Iteration</td><td>Value of j</td></tr>
<tr><td>Iter 1</td><td>j = 1 + 0x3</td></tr>
<tr><td>Iter 2</td><td>j = 1 + 1x3</td></tr>
<tr><td>Iter 3</td><td>j = 1 + 2x3</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Iter p</td><td>j = 1 + 3(p-1) = n
$\therefore$ p = $\frac{n-1}{3}+1$</td></tr>
</table>
</td><td>$\frac{n-1}{3}+1$</td></tr>
<tr><td>Iter 4</td><td>i = 4</td><td>
<table>
<tr><td>Iteration</td><td>Value of j</td></tr>
<tr><td>Iter 1</td><td>j = 1 + 0x4</td></tr>
<tr><td>Iter 2</td><td>j = 1 + 1x4</td></tr>
<tr><td>Iter 3</td><td>j = 1 + 2x4</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Iter p</td><td>j = 1 + 4(p-1) = n
$\therefore$ p = $\frac{n-1}{4}+1$</td></tr>
</table>
</td><td>$\frac{n-1}{4}+1$</td></tr>
<tr><td>:</td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><th>Iterations</th><th>Value of i</th><th>j loop execution</th><th>No. of times j loop executed</th></tr>
<tr><td>Iter n</td><td>i = n</td><td>
<table>
<tr><th>Iteration</th><th>Value of j</th></tr>
<tr><td>Iter 1</td><td>j = 1 + 0xn</td></tr>
<tr><td>Iter 2</td><td>j = 1 + 1xn</td></tr>
<tr><td>Iter 3</td><td>j = 1 + 2xn</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Iter p</td><td>j = 1 + n(p-1) = n
$\therefore p = \frac{n-1}{n} + 1$</td></tr>
</table>
</td><td>$\frac{n-1}{n} + 1$</td></tr>
</table>

Total no. of times

j loop executed = $n + \frac{n-1}{2} + 1 + \frac{n-1}{3} + 1 + \frac{n-1}{4} + 1 + \ldots\ldots + \frac{n-1}{n} + 1$

= (n/1 + n/2 + n/3 + n/4 + ……. + n/n) - (1/2 + 1/3 + 1/4 + ……. + 1/n) + (1 + 1 + 1 + …… upto n-1 no. of terms)

= n(1/1 + 1/2 + 1/3 + 1/4 + ……. + 1/n) – (1/1 + 1/2 + 1/3 + ……. + 1/n) + 1/1 + (n - 1)

= nlogn – logn + n

= (n-1)logn + n

≈ nlogn + n

≈ n(logn +1)

≈ nlogn (approximately)

Therefore time complexity of the program will be O(nlogn).

Chapter 3

Array

Suppose we have to store the marks of 100 students in an institution. If we want to do this job using variables in C language, we need to declare 100 separate variables with different names, which is a tedious and inefficient way. To get rid off this problem, C language offers a data structure known as array. Using the array we can store plenty of values in a well organized manner in the memory of the computer. Moreover any value stored inside an array may be accessed and modified very easily. In a word it can be said that an array is a collection of variables of same type. Therefore the concept of array plays a vital role in C language. The definition of an array is given as follows.

An array is a static linear data structure where the elements of same data type are stored in consecutive memory locations and each elements are accessed by an index.

Declaration of an array - In C, arrays are declared using the following syntax or more precisely it can be said that one dimensional array is declared using the following syntax.

data_type ArrayName[size];

In the above declaration *data_type* denotes the type of the values that can be stored in the array, *ArrayName* indicates the name of the array which is assigned by the programmer and the *size* enclosed with square brackets denotes the fixed or static size of the array which can not be changed throughout the program.

For example: int marks[100];

The above statement declares an array named marks that contains 100 elements which are stored in 100 consecutive memory locations as shown in Fig.3.1. Here the elements of the array contains the marks of different students.

1D Array 'marks'	marks[0] 75	marks[1] 86	marks[2] 91	……..	Marks[98] 85	Marks[99] 67
index→	0	1	2	……..	98	99
Starting Address→	1000	1004	1008	……..	1392	1396

Fig.3.1: Array stores the marks of 100 students

In Fig.3.1 it is being observed that every element of the array has its predecessor and successor except 1st element and 100th (last) element. That's why array is a linear data structure.

Understanding the definition of array – Here the significance of the important terms in the definition of an array will be explained in details.

I) *Data structure:* Data structure is a format of organizing and storing multiple data in an efficient way so that every data can be accessed easily. Array also stores plenty of data in some consecutive memory locations which gives the flexibility to read and modify those data very easily with the help of the indices of that particular array. That's why array is called data structure.

II) *Static data structure:* Why array is called static data structure is illustrated here. If the format of the declaration of an array is noticed, it will be seen that the size of the array enclosed by square brackets is made constant at the very beginning of any program in C. This size is basically the maximum size of the array. If an array is declared as "int a[50]", it can store maximum 50 integer values. Hence if we are going to store 51st value inside this array, we can not store it. Moreover this size can not be altered anywhere after this declaration in C. That's why the array data structure is called static. This creates a serious problem where the number of values to be stored in an array can not be known in advance. In this situation the maximum size of the array is taken very large in the declaration to protect the array to be exhausted. On the contrary due to the declaration of the array with large size plenty of array space may not be utilized.

III) *Linear data structure:* In case of an array every element has one predecessor and one successor element except the first and the last element of the array. In addition to this an array is organized in a sequential manner. Therefore an array is a linear data structure.

IV) *Elements of same data type:* All the elements or the values inside an array must have same data type either integer or floating point or character type. An array can not accommodate different types of data. For example it is not possible to store integer values and floating point values in the same array. Due to this reason the entire array is declared by a data type initially, which gives the information regarding the type of values to be stored inside the array.

V) *Consecutive memory locations:* The elements of an array are stored in successive memory locations. If an array 'a' is declared as 'int' in a program, each element will take 4 bytes of memory for 32-bit architecture and here 1st, 2nd, 3rd and so on elements will be stored with the starting addresses 1000, 1004, 1008 and so on respectively provided the starting address or the base address of the array is 1000. The memory organization of an array declared as 'int' is shown below in Fig.3.2 for clarification.

1D Array 'a' of int type	a[0]	a[1]	a[2]	a[3]	a[4]
index→	0	1	2	3	4
Starting Address→	1000	1004	1008	1012	1016

Fig.3.2: Memory representation of an integer type array with five elements

1) One dimensional array or 1D array – 1D array is an array where the elements are arranged sequentially in one direction. In this case only one row exists and all the elements are positioned one after another row-wise. The elements of a 1D array are accessed by one index only such as - 1st element of a 1D array is accessed by a[0], 2nd element by a[1], 3rd element by a[2] and so on where 0, 1, 2 enclosed by square brackets are the indices. In the above we have discussed one dimensional array mainly to demonstrate the definition of an array. A 1D array is shown in Fig.3.3 along with its memory representation.

1D Array 'a' of int type	a[0]	a[1]	a[2]	a[3]	a[4]
index→	0	1	2	3	4
Starting Address→	1000	1004	1008	1012	1016

Fig.3.3: 1D array of five elements along with index values and memory representation

Declaration of 1D array - A one dimensional array is declared as follows in C.

data_type ArrayName[size];

- data type—the kind of values it can store, for example, int , char , float , double .
- ArrayName—to identify the array.
- size—the maximum number of values that the array can hold.

Accessing the elements of 1D array – To access a particular element of a 1D array we have to write the array name followed by the index of the element enclosed within a square bracket like ArrayName[index]. In case of array in C language the index starts from 0 and ends to (size – 1). That means, the index of first element will be 0 and the index of the last element will be (size – 1).

If we want to access 1st element of an array named 'a', we should use a[0]. Similarly 2nd, 3rd, 4th, 5th element can be accessed by a[1], a[2], a[3], a[4] respectively.

Initialization of 1D array – There are four methods by which a 1D array can be initialized to some values.

Method 1: In this method the 1D array is initialized using the following syntax either during the declaration or after the declaration as given below.

```
int a[5] = {10, 20, 30, 40, 50};        [During array declaration]

            OR

int a[5];
a[5] = {10, 20, 30, 40, 50};            [After array declaration]
```

In the above statement, the 1st element is initialized to 10, 2nd element is initialized with 20 and so on. Therefore we can say a[0] = 10, a[1] = 20, a[2] = 30, a[3] = 40 and a[4] = 50.

Method 2: In this method the 1D array is initialized with some values during the declaration as follows.

```
int a[] = {10, 20, 30, 40, 50};          [During array declaration]
```

Like Method 1 the 1st element is initialized to 10, 2nd element is initialized with 20 and so on i.e. a[0] = 10, a[1] = 20, a[2] = 30, a[3] = 40 and a[4] = 50. But Method 2 is better than Method 1, because in Method 2 the size of the 1D array is not specified into the square brackets. Here the maximum size of the 1D array becomes equal to the number of elements initialized in the array declaration. For example – the size of the array is automatically fixed to 5 by the C compiler in the above statement. It gives the opportunity to the programmer to add as many elements as he wish to add into the 1D array.

Method 3: In this technique we can initialize the elements of the array individually with the help of the index as mentioned below.

```
int a[5];

a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
a[4] = 50;
```

But the above mentioned procedure is not efficient, because here every elements are being initialized separately using the assignment operator. Hence if 100 elements of an array are to be initialized using this Method 3, we have to use 100 assignments for 100 elements one by one which is a cumbersome task. Therefore this method is not accepted for array initialization.

Method 4: Here the initialization of the elements are done using a loop as given below.

```
int a[5], i;
int j = 10;

for(i=0; i<5; i++)
{
        a[i] = j;
        j = j + 10;
}
```

Among the above mentioned four methods Method 1 and Method 3 are not used due their limitations. Therefore Method 2 and Method 4 are generally used in various C programs.

- *What happens if the number of elements initialized in an array is less than the size of the array ?*

Solution: Suppose three elements of an array 'a' is initialized to some values where the size of the array is declared as 5 using the following statement.

```
int a[5] = {10, 20, 30};
```

Here it can be seen that only 3 elements (1st, 2nd and 3rd element) of the array are initialized with 10, 20 and 30 respectively instead of 5 elements. Now a question comes obviously that what will be the values of 4th and 5th element after this initialization. In this situation the remaining elements will be automatically filled by zero. Therefore 4th element (a[3]) and 5th element (a[4]) will become zero.

- *How are all the elements of a 1D array of size 100 initialized with zero ?*

Solution: To initialize a 1D array of size 100 with the value zero we have to follow the method given below.

```
int a[100] = {0};
```

When the above statement is complied the 1st element i.e. a[0] is made zero and the rest of the all 99 elements become zero automatically by the compiler of C. In this way it is possible to make all the elements of a 1D array to be zero very easily.

Note:
1. int a[10] = { }; is illegal, because we have to specify atleast one value inside the curly braces.
2. int a[5] = {10, 20, 30, 40, 50, 60}; is also illegal, because the number of elements initialized is larger than the size of the array.

Designated Initialization of 1D array – If we want to initialize some elements to be initialized with some non-zero values randomly (not sequentially) and other remaining elements to be zero, then it is better to use designated initialization of an array. For example 1st, 3rd and 7th elements are to be initialized with 45, 36 and 21 respectively and other elements like 2nd, 4th, 5th, 6th, 8th, 9th, 10th elements should be zero. In this case we have to use the following statement to accomplish this.

```
int a[10] = {[0] = 45, [2] = 36, [6] = 21};
```

In the above syntax the values inside the square brackets indicates the indices of 1st, 3rd and 7th element of the 1D array. This way of initialization is called designated initialization and each number in the square bracket is said to be a designator.

- *An array without any specified size is initialized using the statement "int a[ ] = {[0] = 45, [2] = 36, [6] = 21}". What will be the size of the array ?*

  Solution: It is being observed that the size of the array 'a' is not mentioned here and all the initialization are done designated initialization. Here the compiler will deduce the size of the 1D array from the largest designator in the list. Therefore the size of the array will be 7 in the above case, as the largest designator here is [6].

We can mix up the designated initialization and the normal initialization as follows.

```
int a[ ] = {1, 7, 5, [5] = 90, 6, [8] = 4};
```

The above statement is equivalent to the following initialization.

```
int a[ ] = (1, 7, 5, 0, 0, 90, 6, 0, 4};
```

The above two initialization implies that a[0] = 1, a[1] = 7, a[2] = 5, a[3] = 0, a[4] = 0, a[5] = 90, a[6] = 6, a[7] = 0, a[8] = 4.

- *Consider the initialization "int a[ ] = {1, 2, 3, [2] = 4, [6] = 45};" for a 1D array. What will be the value of third element of the array after this initialization ?*

Solution: Here the important point is that, there is a coincidence between normal initialization and designated initialization. Consider the above mentioned initialization where both the normal and designated initialization are used at a time.

```
int a[ ] = {1, 2, 3, [2] = 4, [6] = 45};
```

In the above case the third element (a[2]) is made 3 using normal initialization and at the same time it is assigned with a value of 4 using designated initialization. Naturally the question arises, which value will be initialized for the 3rd element. Here designated initialization will get the priority. That means, the value of the 3rd element will become 4 instead of 3. Therefore the equivalent statement for the above initialization will be given as:

```
int a[ ] = {1, 2, 4, 0, 0, 0, 45};
```

Advantages of designated initialization:

- No need to bother about the entries containing zero.

  ```
  int a[10] = {45, 0, 36, 0, 0, 0, 21, 0, 0, 0};int a[10] = {[0] = 45, [2] = 36, [6] = 21};
  ```

  The initialization can be performed in simpler way using designated initialization as follows.

  ```
  int a[10] = {[0] = 45, [2] = 36, [6] = 21};
  ```

- No need to bother about the order at all.

  ```
  int a[10] = {[0] = 45, [2] = 36, [6] = 21};
  int a[10] = {[2] = 36, [6] = 21, [0] = 45};
  ```

  The above two statements give the same result.

Memory representation of 1D array – We know that the elements are placed consecutively into the memory in case of a 1D array. If a 1D array is declared as int type, each element of the array will occupy 4 bytes in the memory space for a 32-bit architecture. If the starting address or the base address of the 1D array is 1000, the first element of the array will be stored from the address 1000 to 1003, the second element will be stored from 1004 to 1007, third element will occupy from the address 1008 to 1011 etc. Hence it is clear that the starting addresses of the consecutive elements in a 1D array will be incremented by 4 for a 1D array declared as int. Similarly the starting addresses of the successive elements will be incremented by 1 if the 1D array is declared as char. It implies that how much the addresses of the consecutive elements will be increased in a 1D array, that depends on the data type of the 1D array. A pictorial representation of the memory organization of an integer type 1D array as well as char type 1D array has been depicted in Fig.3.4(a) and Fig.3.4(b) respectively for better understanding of the above mentioned phenomenon.

1D Array a of int type	a[0]	a[1]	a[2]	a[3]	a[4]
index→	0	1	2	3	4
Starting Address→	1000	1004	1008	1012	1016

Fig.3.4(a): Memory organization of int type 1D array of size = 5 and base address = 1000

1D Array b of char type	b[0]	b[1]	b[2]	b[3]	b[4]
index→	0	1	2	3	4
Starting Address→	2000	2004	2008	2012	2016

Fig.3.4(b): Memory organization of char type 1D array of size = 5 and base address = 2000

Now we can determine the starting address of any element of a 1D array ‘a’ using the following formula and this formula can verified for the 1D array ‘a’ and ‘b’ in Fig.3.4.

Starting Address of $a[i] = B + (i - i_0) \times W$ Where B = Base address of 1D array
W = Storage size of each element in the array in bytes
i = Index of the element in the array
i_0 = Starting index of the 1D array

Now the storage size of every element in an array (W) depends on the date type of the array. Different values of W depending on the data type of the array are given below.

W = 4 bytes for int data type
W = 8 bytes for long int data type
W = 4 bytes for float data type
W = 8 bytes for double data type
W = 1 byte for char data type

In C language the index of a 1D array starts from 0 always. That’s why $i_0 = 0$ for the above mentioned formula in C.

2) Two dimensional array or 2D array – 2D array is an array where the elements are arranged sequentially in two directions – along the row and along the column. The elements of a 2D array is placed like a 2D matrix. The elements of a 2D array are accessed by two indices, one index gives the row-wise position and another index gives the column-wise position. Hence any element of a 2D array may be demoted by a[i][j] where i is the row index and j is the column index. The pictorial view of a 2D array with 5 rows and 6 columns is shown in Fig.3.5.

j ↓ i→	0	1	2	3	4	5
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]	a[3][5]
4	a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]	a[4][5]

Fig.3.5: 2D array with 5 rows and 6 columns

In Fig.3.5 we can see that every elements of the 2D array is represented by the array notation in C language. For example – a[0][0] is the element with 1st row and 1st column, a[4][2] is the element with 5th row and 3rd column. The 1st index inside the square bracket denotes the row of the array whereas 2nd index denotes the column of the array. Therefore in general a[i][j] denotes (i + 1)th row and (j + 1)th column of the 2D array. Therefore a 2D array with m no. of rows and n no. of columns can hold maximum (m × n) no. of elements.

Declaration of 2D array - A two dimensional array is declared as follows in C.

data_type ArrayName[SizeOfRow][SizeOfColumn];

- data type – the kind of values it can store, for example, int , char , float , double .
- ArrayName – to identify the array.
- SizeOfRow – the maximum number of rows that the array can hold.
- SizeOfColumn – the maximum number of columns that the array can hold.

For example, an integer type 2D array with 4 rows and 5 columns is declared as follows.

```
int a[4][5];
```

Accessing the elements of 2D array – To access a particular element of a 2D array we have to write the array name followed by the row index of the element enclosed within a square bracket and the column index of the element enclosed with another square bracket like ArrayName[i][j] where i and j are the row index and the column index of the particular element of the array. If we want to access 1st element of the 2D array named ‘a’, we should use a[0][0] where both zeros indicates the row index and the column index of the array.

Initialization of 2D array – There are two methods by which a 2D array can be initialized to some values.

Method 1: In this method a 2D array is initialized as follows.

```
int a[2][3] = {1, 2, 3, 4, 5, 6};
```

In the above case the number of rows is 2 and the number of columns is 3. Therefore total number of elements present in the 2D array is (2 × 3) = 6. Now the first three values will be assigned to the elements of the first row sequentially which results a[0][0] = 1, a[0][1] = 2, a[0][2] = 3. Similarly the values 4, 5 and 6 will be stored into the elements a[1][0], a[1][1] and a[1][2] respectively. The pictorial view of the 2D array is shown in Fig.3.6 after the initialization.

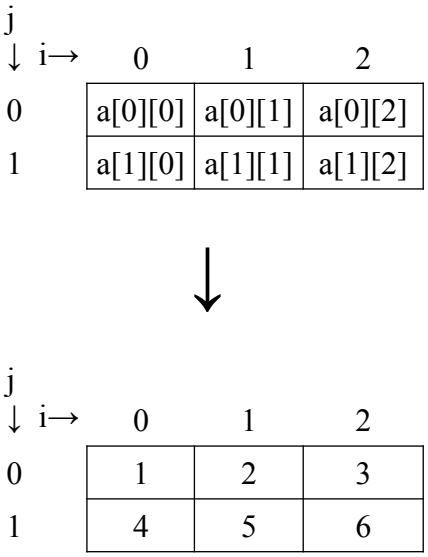


Fig.3.6: 2D array after initialization

Method 2: This method is better than Method 1 as the initialization of the elements can be visualized more clearly than Method 1. In this method the initialization is done by the following way.

```
int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

From the above statement it is clear that first curly braces denotes the first row and the second curly braces denotes the 2nd row of the 2D array. Here we shall get the same results after initialization as shown in Fig.3.6.

Memory representation of 2D array – There are basically two types of conventions by which a 2D array may be represented namely row major and column major.

1. Row major ordering - A 2D array can be considered as the combination of some 1D arrays, because each row of a 2D array may be considered as a 1D array. Therefore we can assume that there are m number of 1D arrays of size = n in a 2D array of m no. of rows and n no. of columns. If we consider a 2D array of size = 2 × 3, there will be two 1D arrays with 3 elements each as shown in Fig.3.7.

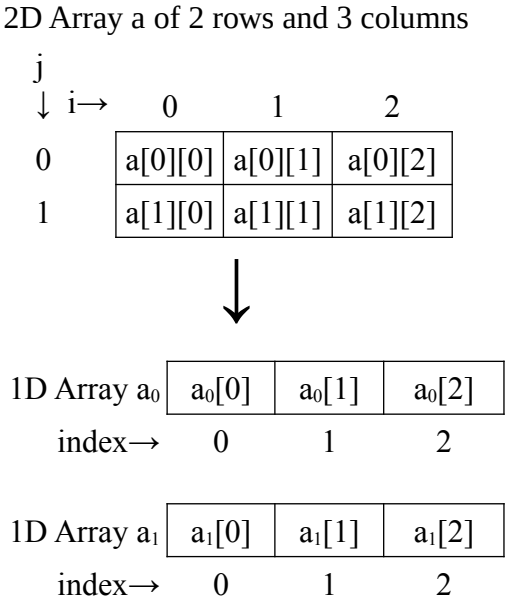


Fig.3.7: 2D array of size = 2 × 3 represented by two 1D arrays

We know that the elements are placed consecutively into the memory in case of a 1D array. If a 1D array is declared as int type, each element of the array will occupy 4 bytes in the memory space for a 32-bit architecture. According to the memory mapping of 1D array which has already been discussed previously the starting addresses of the consecutive elements in an integer type 1D array will be incremented by 4.

Memory mapping of a 2D array starts with the memory mapping of the first 1D array, then the second 1D array, after that third 1D array and so on sequentially. That means after the last element of 1st 1D array the 1st element of 2nd 1D array will be memory-mapped and after the last element of 2nd 1D array first element of 3rd 1D array comes. In this way the entire 2D array is memory-mapped row-wise as shown in Fig.3.8. This type of ordering of memory addresses in case of a 2D array is called as row major order. It is important to mention that C language follows row major form to represent the memory organization of a 2D array.

j ↓ i→	0	1	2	3	4	5
0	a[0][0] *1000*	a[0][1] *1004*	A[0][2] *1008*	A[0][3] *1012*	a[0][4] *1016*	a[0][5] *1020*
1	a[1][0] *1024*	a[1][1] *1028*	A[1][2] *1032*	A[1][3] *1036*	a[1][4] *1040*	a[1][5] *1044*
2	a[2][0] *1048*	a[2][1] *1052*	A[2][2] *1056*	A[2][3] *1060*	a[2][4] *1064*	a[2][5] *1068*

Nate: The number written in italics under each element of the 2D array in every cell are the starting address of every element.

Fig.3.8: Row major ordering for memory representation of a 2D array of size (3 × 6)

The following formula determines the address of any element in a 2D array of size (m × n) using row major ordering.

Address of a[i][j] = B + W × [(i – i_0) × n + (j – j_0)]

B = Base address of the 2D array
W = Storage size of each element in the 2D array in bytes
n = Maximum number of columns in the 2D array
i = Row index of the element a[i][j]
i_0 = Starting row index of the 2D array
j = Column index of the element a[i][j]
j_0 = Starting column index of the 2D array

Now we shall verify the above mentioned formula of row major ordering with the help of the 2D array shown in Fig.3.8. Suppose the address of the element a[2][3] is to be determined using this formula. In this case we have the values of the following parameters.

B = 1000, W = 4 bytes, n = 6, i = 2, i_0 = 0, j = 3, j_0 = 0

∴ Address of the element a[2][3] = B + W × [(i – i_0) × n + (j – j_0)]
= 1000 + 4 × [(2 – 0) × 6 + (3 – 0)]
= 1000 + 4 × [12 + 3]
= 1060

If the memory mapping of the 2D array shown in Fig.3.8 is compared with this result, it is exactly the same as before.

2. Column major ordering – In case of column major ordering the addresses of the successive elements are incremented by 4 column-wise. It says that the memory mapping is done for 1D array corresponding to 1st column, then for the 1D array corresponding to 2nd column and so on. After the address of the last element from the 1st column array the address of the first element of the 2nd column array comes. Column major ordering for a (3 × 6) 2D array is shown in Fig.3.9.

j ↓ i→	0	1	2	3	4	5
0	a[0][0] *1000*	a[0][1] *1012*	A[0][2] *1024*	A[0][3] *1036*	a[0][4] *1048*	a[0][5] *1060*
1	a[1][0] *1004*	a[1][1] *1016*	A[1][2] *1028*	A[1][3] *1040*	a[1][4] *1052*	a[1][5] *1064*
2	a[2][0] *1008*	a[2][1] *1020*	A[2][2] *1032*	A[2][3] *1044*	a[2][4] *1056*	a[2][5] *1068*

Nate: The number written in italics under each element of the 2D array in every cell are the starting address of every element.

Fig.3.9: Column major ordering for memory representation of a 2D array of size (3 × 6)

The following formula determines the address of any element in a 2D array of size (m × n) using column major ordering.

Address of a[i][j] = B + W × [(i – i_0) + (j – j_0) × m]

B = Base address of the 2D array
W = Storage size of each element in the 2D array in bytes
m = Maximum number of rows in the 2D array
i = Row index of the element a[i][j]
i_0 = Starting row index of the 2D array
j = Column index of the element a[i][j]
j_0 = Starting column index of the 2D array

Now we shall verify the above mentioned formula of column major ordering with the help of the 2D array shown in Fig.3.9. Suppose the address of the element a[2][3] is to be determined using this formula. In this case we have the values of the following parameters.

B = 1000, W = 4 bytes, m = 3, i = 2, i_0 = 0, j = 3, j_0 = 0

∴ Address of the element a[2][3] = B + W × [(i – i_0) + (j – j_0) × m]
= 1000 + 4 × [(2 – 0) + (3 – 0) × 3]
= 1000 + 4 × [2 + 9] = 1044 *(Verified from Fig.3.9)*

Here one point is important that the address of the last element of a 2D array will be same for both of the row major and the column major ordering.

Limitations of Array:

1) Arrays are declared as fixed size. If it is required to handle more number of elements than the size of the array, it is not possible to do that. Due to this reason array is referred as static data structure.
2) Data elements of an array are stored in continuous memory locations which may not be available always.
3) Insertion and deletion of element in case of array becomes problematic because of shifting of elements from their positions.

Different operation on a 1D array – In this section we shall discuss few operations which may be performed on a 1D array..

1. Insertion of an element into a 1D array – This operation inserts a new element at a specified position of the 1D array. If the position 'pos' is specified, then all the elements right of the position 'pos' and the element at position 'pos' will be shifted one position right to make a vacant space for the insertion of a new element.

Time complexity for the best case: When the new element is inserted at the last position, it is required to shift the last element only from (n – 1)th index to nth index of the array. Therefore only one shifting is required and this becomes the bast case. So the time complexity here will be O(1).

Time complexity for the worst case: When the new element is to be inserted at the 1st position, all the elements of the array (from 1st position to the last position) should be shifted right. Hence there will be n no. of shifting from the element a[0] to a[n-1], which becomes the worst case. Therefore time complexity for the worst case becomes O(n).

Algorithm to insert an element at a specific position of an array:

Step 1: Start

Step 2: Input the element which is to be inserted into the variable 'element'.

Step 3: Input the position at which the element is to be inserted into the variable 'pos'.

Step 4: Set i = n – 1 *[n is the size of the 1D array]*

Step 5: Repeat Step 6 to Step 7 while i >= pos – 1.

Step 6: Set a[i + 1] = a[i]

Step 7: Set i = i – 1

Step 8: Set a[pos – 1] = element *[To insert the element at specific position]*

Step 9: Set n = n + 1

Step 10: Stop

User defined function in C to insert an element at a specific position of the array.

```
int Insert(int a[],int n)
{
        int i, element, pos;

        printf("Enter the element to be inserted: ");
        scanf("%d", &element);

        printf("Enter the position for insertion: ");
        scanf("%d", &pos);

        for(i=n-1; i>=pos-1; i--)
                a[i+1] = a[i];

        a[pos - 1] = element;

        return n+1;
}
```

2. Deletion of an element from a 1D array – This operation will remove an existing element from the array. For this purpose the position 'pos' at which the element will be deleted is taken as input and all the elements beyond the index (pos – 1) will be shifted left by one position, which will replace the element at index (pos – 1) by the element at index pos. Thus the element at the specified position 'pos' will be removed.

Time complexity for the best case: The best case happens when the element at the last position is deleted. Here no left shifting is required to remove the last element, only the length of the array is decreases from n to (n – 1). Therefore the time complexity becomes O(1).

Time complexity for the worst case: The worst case occurs when 1st element is to be removed from the array. In this case all the elements starting from 2nd position to nth position are shifted left by one position. Thus we require (n – 1) no. of left shifting to accomplish this task. Therefore time complexity for the worst case becomes O(n).

Algorithm to delete an element at a specific position of an array:

Step 1: Start

Step 2: Input the position at which the element is to be deleted into the variable 'pos'.

Step 3: Set i = pos – 1.

Step 4: Repeat Step 5 to Step 6 while i <= n – 2 *[n is the size of the 1D array]*

Step 5: Set a[i] = a[i+1].

Step 6: Set i = i + 1

Step 7: Set n = n – 1

Step 8: Stop

User defined function in C to delete an element at a specific position of the array.

```
int Delete(int arr[],int n)
{
        int i, pos;

        printf("Enter the position for deletion: ");
        scanf("%d", &pos);

        for(i=pos-1; i<n-1; i++)
                arr[i] = arr[i+1];

        return n-1;
}
```

The entire C program to insert an element to a specified position of an array and also delete an element from a specified position of the same array is given below.

C program to insert an element to a specified position of an array and also delete an element from a specified position of the same array.

```
#include<stdio.h>
#include<stdlib.h>
#define Min 10
#define Max 100
#define ArrayLen 100

void Create(int [],int );
void Display(int [],int );
int Insert(int [],int );
int Delete(int [],int );

int main()
{
	int a[ArrayLen], n, option;
	int element, pos;

	printf("Enter the number of elements to create the array: ");
	scanf("%d", &n);

	Create(a, n);

	printf("The elements of the created array:\n");
	Display(a, n);

	while(1)
	{
		printf("*******************************\n");
		printf("Press 1 to insert an element.\n");
		printf("Press 2 to delete an element.\n");
		printf("Press 0 to exit the program.\n");
		printf("*******************************\n");
		printf("Enter your option: ");
		scanf("%d", &option);

		switch(option)
		{
			case 0: exit(1);

			case 1: n = Insert(a, n);
				printf("The elements of the array after insertion:\n");
				Display(a, n);
				break;
```

```
                        case 2: n = Delete(a, n);
                                printf("The elements of the array after deletion:\n");
                                Display(a, n);
                                break;

                        default: printf("Wrong option is selected.\n");
                                 break;
                }
        }

        return 0;
}

void Create(int arr[],int n)
{
        int i;

        for(i=0; i<n; i++)
                arr[i] = rand() % (Max - Min + 1) + Min;
}

void Display(int arr[],int n)
{
        int i;

        for(i=0; i<n; i++)
                printf("%d  ", arr[i]);

        printf("\n");
}

int Insert(int arr[],int n)
{
        int i, element, pos;

        printf("Enter the element to be inserted: ");
        scanf("%d", &element);
        printf("Enter the position for insertion: ");
        scanf("%d", &pos);

        for(i=n-1; i>=pos-1; i--)
                arr[i+1] = arr[i];

        arr[pos - 1] = element;

        return n+1;
}
```

```
int Delete(int arr[],int n)
{
        int i, pos;

        printf("Enter the position for deletion: ");
        scanf("%d", &pos);

        for(i=pos-1; i<n-1; i++)
                arr[i] = arr[i+1];

        return n-1;
}
```

Chapter 4

Stack

A stack is a linear data structure where insertions and deletions are allowed only at the one end which is called the top of the stack. Real life examples of the stack is – 1) stack of books 2) stack of coins. In case of a stack of books normally the top-most book can be picked up and if a new book is to be inserted, it can be placed on the top of the stack. In this case one may ask, any book except the top-most one may be taken out carefully, but if the entire stack of books is kept inside a jar, then it will be possible only to take out a book from the top.

Therefore it is clear that the top-most element of the stack is inserted at last and it will be taken out first. Due to this reason, the stack is called LIFO (Last-in First-out) data structure. Different operations like – push, pop, peek can be performed on a stack. Stack can be implemented either by using arrays or by using linked list. In this chapter the array implementation of the stack will be discussed first, then implementation using linked list will be covered. Before discussing different operations on stack it is important to explain how a stack can be initialized using array.

Initialization of Stack using array – In case of array implementation a stack is represented by a one dimensional array where each cell of the array is capable to hold the element inserted into the stack. That means the first element of the stack is placed in the cell with index 0, the second element is placed in the cell with index 1 and so on. It is clear from this discussion that the side of the array towards the index 0 is treated as bottom of the stack and the opposite side of the array is considered the open end or top of the stack where insertion or deletion may take place.

If stack is implemented by an array named ‘Stack’, then it will be declared by the following statement.

```
Date_type  Stack[10];
```

So it is similar to the declaration of a normal array. As Stack is a normal array, insertion and deletion operation can be performed at any place i.e. at any index of the array. But according to the property of a stack insertion and deletion may happen only at one end of the stack. Hence to implement a stack using array, we have to do something so that the array ‘Stack’ can behave like a stack.

As insertion and deletion are performed at one end called top of the stack, we need a variable ‘top’ which will be used to keep track of the last inserted element or topmost element of the array which is used as a stack. Basically here the variable ‘top’ will always hold the index value of the topmost element of the stack. Now initially the stack will be empty i.e. no element will be present inside the stack. Now the question arises “What will be the initial value of the variable ‘top’ ?”. We know that the valid index value of a stack starts from 0. This implies that any positive value along with 0 is considered as valid index of an array. Therefore initially for an empty stack the variable ‘top’ should be assigned to a negative value. Now the question comes “Which negative value ?”. If the first element is inserted into the stack, it occupies the cell with index 0 and becomes the topmost element of the stack. To track this topmost element the variable ‘top’ must hold the index value 0 after increment by one. Therefore the variable ‘top’ must be initialized with -1. Each time during push operation one element is inserted into the stack and the value of the ‘top’ variable is incremented by

one. On the contrary, each time during the pop operation, the topmost element is taken out of the stack and the variable 'top' is decremented by one. Thus the variable 'top' always holds the index value of the topmost element of the stack.

The declaration of the stack using array and initialization of the variable 'top' is given below.

```
#define Max 10
Data_type Stack[Max];
int top = -1;
```

Now the array 'Stack' and the variable 'top' will be utilized by the different user-defined functions used for the implementation of the various operations of the stack like push, pop, isEmpty, isFull etc. For this reason the array 'Stack' and the variable 'top' is declared globally outside the main function to avoid the passing the entire stack array and the 'top' variable every time during the use of these functions. The global declaration of the stack is shown below for clarification.

```
#include<stdio.h>
#define Max 10

Data_type Stack[Max];
int top = -1;

int main()
{
        .
        .
        .
        .
}
```

Operation on Stack – The operations on stack is categorized into two categories – 1) primary operation and 2) secondary operation.

1) Primary Operations – Under primary operations we have push and pop operations.

1. **Push –** This operation inserts data onto the stack. i.e. after insertions the newly inserted data becomes top-most element of the stack. Suppose we want to insert three elements (25, 10, 7) onto the stack one by one starting from left to right. That means, 25 will be pushed first, then 10 and so on. Pushing of these three numbers on the stack is shown pictorially in Fig.4.1.

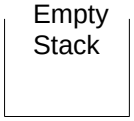
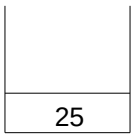
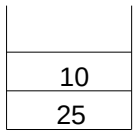
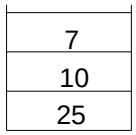
Element before insertion	Status of Stack after push operation
	
25	
10	
7	

Fig.4.1: Three numbers 25, 10, 7 are pushed into the stack one by one

We know that the variable 'top' is -1 when the stack is empty. When an element is pushed first on the stack, the 'top' will be incremented by one first to become 0 which is a valid index. Now the element is stored in Stack[top] of the array. One point is important to mention here that we can not push elements indefinitely on the stack in case of the array implementation. It is restricted by the maximum size of the array used for stack. If the maximum size of the array is 10, we can not insert 11 number of elements into the stack. If the maximum number of elements have been inserted into the stack, then the stack is said to be full. Therefore when the stack is full, then the value of the variable 'top' becomes (Max – 1) and in this situation no more element can be pushed. This status of the stack is also known as "STACK OVERFLOW". The algorithm of push operation is given below.

Algorithm of push operation:

Step 1: Start

Step 2: If the stack is Full, then
a) Print 'STACK OVERFLOW'.
b) Go to Step 5.
[End of If]

Step 3: Set top = top + 1

Step 4: Set Stack[top] = Value

Step 5: Stop

User defined function in C to push an element on the stack

```
void Push(int number)
{
        if(isFull())
        {
                printf("STACK OVERFLOW\n");
                return;
        }
        stack[++top] = number;
}
```

2. Pop – This operation removes the top-most element from a stack. Suppose we want to pop two elements from the stack of three elements as shown in Fig.4.1. How the two elements are popped from the stack is shown in the following figure with the corresponding status of the stack.

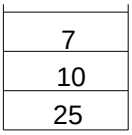
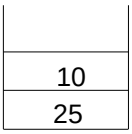
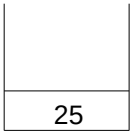
Status of Stack after pop operation	Element popped from the stack
	
	7
	10

Fig.4.2: Three elements are popped from the stack one by one

Algorithm of pop operation:

Step 1: Start

Step 2: If the stack is Empty, then
a) Print 'STACK UNDERFLOW'.
b) Go to Step 5.
[End of If]

Step 3: Set Value = Stack[top]

Step 4: Set top = top – 1

Step 5: Stop

User defined function in C to pop an element from the stack

```
int Pop()
{
        if(isEmpty())
        {
                printf("STACK UNDERFLOW\n");
                exit(1);
        }

        return stack[top--];
}
```

2) Secondary Operations – Secondary operations on stack include isFull, isEmpty and Peek.

1. **isFull –** In case of array implementation of stack, the stack is initialized as an array with its maximum size. During initialization of a stack the maximum size is defined by a macro 'Max' which is equal to 10. Therefore the stack can hold maximum 10 number of data. If 11^{th} data is attempted to pushed on the stack, this will not be allowed. This situation is known as "STACK OVERFLOW". The operation "isFull" checks that the stack is already full with data or not. If the top variable is equal to the maximum index value which is (Max – 1), the "isFull" function returns 1 to indicate that the stack is full. If top is less than (Max – 1), it returns 0 to show that the stack is not full. The algorithm for isFull operation is given below.

Algorithm of isFull operation:

Step 1: Start

Step 2: If top = Max – 1, then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isFull operation

```
int isFull()
{
        if(top == Max - 1)
                return 1;
        else
                return 0;
}
```

2. **isEmpty –** If there is no element in the stack, the stack is said to be empty. The isEmpty operation checks whether the stack is empty or not. In case of array implementation of the stack, the top variable is initialized to -1 which is an invalid index in case of any array. If the stack comprises one element, the top will hold index value 0. Therefore the top should hold -1 for empty stack so that it can be incremented to 0 when the first element of the stack is pushed. The isEmpty function checks the status of the variable top. If top is equal to -1, it implies that the stack is empty and the function isEmpty returns 1 in that case. If top is not equal to -1, the function isEmpty returns 0 for non-empty status of the stack.

Algorithm of isEmpty operation:

Step 1: Start

Step 2: If top = – 1, then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isEmpty operation

```
int isEmpty()
{
        if(top == -1)
                return 1;
        else
                return 0;
}
```

3. Peek – Peek operation gives the value of the topmost element of the stack without removing it from the stack. That means, the peek and pop both operations return the topmost element of the stack, but in case of pop the topmost element is removed from the stack whereas in case of peek operation the top element will not be deleted. Therefore the top variable will not be decremented by one for peek operation. In C the value of stack[top] gives the topmost element to implement the peek function whereas stack[top--] implement the pop operation.

Evaluation of Arithmetic Expression – Infix, prefix and postfix notations are the three different notations of writing algebraic expressions. Among these three notations we are mainly familiar with the infix notation to evaluate an algebraic expression.

1) Infix Notation – While writing an algebraic expression using infix notation, the operator is placed in between the operands. For example – A + B is an infix notation. Here the '+' operator is placed in between two operands A and B. In an infix expression the evaluation of the operands depends on the precedence and associativity of the operators. This implies that the operands with operator of highest precedence will be evaluated first. Let's take an example.

We have to determine the value of the infix expression "10 + 4 * 5". Here "4 * 5" will be determined before "10 + 4" as * operator has higher precedence over + operator. Now the result of "4 * 5" will be added to 10. Thus the expression will give the value of 30.

For simplicity of the algebraic expression we shall deal with only '^', '*', '/', '%', '+', '-' operators and parenthesis '( )' in an infix expression. The following table gives the precedence and associativity of the above operators along with the priority level.

Precedence and Associativity Table-4.1

Level	Category	Operators	Associativity
1	Parenthesis	( ) → parenthesis	Left to Right
2	Exponential	^ →Power	Left to Right
3	Multiplicative	* → Multiplication / → Division % → Modulo division	Left to Right
4	Additive	+ → Addition - → Subtraction	Left to Right

Determination of an infix expression is done using the precedence and associativity of the operators given in the above table. Let's take some examples to clarify this concept.

Example1: Evaluate the infix expression 8 + 4 * (6 / 3) / 8 – 2 ^ 3

Steps	Evaluation of Infix expression	Remarks
1	8 + 4 * (6 / 3) / 8 – 2 ^ 3 = 8 + 4 * 2 / 8 – 2 ^ 3	Parenthesis () has highest precedence. That's why the expression between the parenthesis will be evaluated first.
2	8 + 4 * 2 / 8 – 2 ^ 3 = 8 + 4 * 2 / 8 – 8	Now ^ operator will be evaluated due to its highest priority.
3	8 + 4 * 2 / 8 – 8 = 8 + 8 / 8 – 8 = 8 + 1 - 8	Now * and / both have the highest priority. So as per the rule of associativity left to right evaluation will be performed.
4	8 + 1 – 8 = 9 – 8 = 1	+ and - both have the highest priority. So as per the rule of associativity left to right evaluation will be performed.

Disadvantage: It is being seen from the above evaluation that the evaluation is not performed left to right always. It is done randomly depending upon the precedence of the operators. To determine the infix expression in a computer, it will be required to scan the expression multiple times from left to right which causes wastage of time.

To solve this problem postfix or prefix notation is used in a computer to evaluate an expression.

2) Postfix Notation - It was developed by Jan Łukasiewicz who was a Polish logician, mathematician and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN. In postfix notation, as the name suggests, the operator is placed after the operands.

For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation. A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation AB+C* . While evaluation, addition will be performed prior to multiplication. Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them.

Advantage: It requires only a single scan to evaluate an expression which makes the postfix expression preferable for computer.

Validity of an infix expression – Checking the validity of an infix expression is essential before evaluating an infix expression. Here validity of an infix expression involves the brackets used in the expression. An infix expression will be considered as valid if the following conditions are satisfied.

1. The total number of left brackets must be equal to the total number of right brackets used in the infix expression. Here left brackets means the opening brackets and right brackets indicates the closing brackets.

2. For every right bracket there should be a left bracket of the same type. This implies that there must be a closing or right parenthesis ')' immediately after the opening or left parenthesis '(' i.e. there should not be any right curly brace '}' or right square brace ']' after the left parenthesis '('.

Now an infix expression may be invalid due to the following three reasons.

1. Number of left brackets is greater than the number of right brackets. For example – [{(2 + 3) * 5}.

2. Number of right brackets is greater than the number of left brackets. For example – {(2 + 3) * 5}].

3. Mismatch occurs between left bracket and right bracket. For example – [{(2 + 3) * 5]}.

Algorithm to check the validity of an infix expression:

Step 1: Start

Step 2: Scan every character of the infix expression from left to right and repeat Step 3 to Step 4 until the end of the infix expression.

Step 3: If the character is a left bracket, then
- a) push the character on the stack.

[End of If (Step 3)]

Step 4: If the character is a right bracket, then
- a) If the stack is empty, then
  - i) Print "Invalid expression as no. of right brackets is greater than no. of left brackets".

  [End of If (a)]

- b) Pop the top-most bracket from the stack.

- c) If the popped left bracket does not match the right bracket, then
  - i) Print "Invalid expression due to mismatch in brackets".

  [End of If (c)]

Step 5: If the stack is empty after scanning the entire expression, then

a) Print “Valid expression”.

Otherwise,

b) Print “Invalid expression as no. of left brackets is greater than no. of right brackets”.

Step 6: Stop.

C Program to check the validity of an infix expression

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define Max 100
#define MaxLen 50

char stack[Max];
int top = -1;

void Push(char );
int isFull();
char Pop();
int isEmpty();
int ExpValidity(char []);

int main()
{
        char infix[MaxLen];

        printf("Enter the Expression: ");
        gets(infix);

        if(ExpValidity(infix))
                printf("The expression is valid.\n");
        else
                printf("The expression is not valid.\n");

        return 0;
}
```

```
int ExpValidity(char infix[])
{
        int i, StrLen;
        char symbol;

        StrLen = strlen(infix);

        for(i=0; i<StrLen; i++)
        {
                symbol = infix[i];

                switch(symbol)
                {

                        case '(':
                        case '{':
                        case '[': Push(symbol);
                                  break;
                        case ')': if(isEmpty())
                                  {
                                          printf("No. of right brackets is more than no. of left
                                          brackets.\n");
                                          return 0;
                                  }

                                  if(Pop() != '(')
                                  {
                                          printf("Bracket Mismatch.\n");
                                          return 0;
                                  }

                                  break;

                        case '}': if(isEmpty())
                                  {
                                          printf("No. of right brackets is more than no. of left
                                          brackets.\n");
                                          return 0;
                                  }

                                  if(Pop() != '{')
                                  {
                                          printf("Bracket Mismatch.\n");
                                          return 0;
                                  }

                                  break;
```

```
                        case ']': if(isEmpty())
                                {
                                        printf("No. of right brackets is more than no. of left
                                        brackets.\n");
                                        return 0;
                                }

                                if(Pop() != '[')
                                {
                                        printf("Bracket Mismatch.\n");
                                        return 0;
                                }

                                break;

                        default: break;
                }
        }

        if(isEmpty())
                return 1;
        else
        {
                printf("No. of left brackets is more than no. of right brackets.\n");
                return 0;
        }
}

void Push(char ch)
{
        if(isFull())
        {
                printf("STACK OVERFLOW\n");
                return;
        }

        stack[++top] = ch;
}

int isFull()
{
        if(top == Max - 1)
                return 1;
        else
                return 0;
}
```

```
char Pop()
{
        if(isEmpty())
        {
                printf("STACK UNDERFLOW\n");
                exit(1);
        }

        return stack[top--];
}

int isEmpty()
{
        if(top == -1)
                return 1;
        else
                return 0;
}
```

Conversion of an infix expression into a postfix expression – There are two procedures by which an infix expression can be converted into a postfix expression. One procedure is done manually and other is followed by a computer.

1. Infix to postfix conversion manually – In case of manual procedure the operator with highest priority is identified first, then the part of the infix expression formed by that operator and other two operands will be converted to its equivalent postfix expression. Now the postfix expression thus formed will be considered as a single operand and again the operator with highest precedence will be detected and converted to postfix notation. This process will continue until the entire infix expression will be transformed to its equivalent postfix expression. Here one point is essential to note that if two or more operators have the same highest priority in any time, then the associativity rule should be applied for the conversion. Let's take some examples to clarify the process of conversion.

Example2: Convert the following infix expression to its corresponding postfix expression.

7 + 5 * 3 / 5 ^ 1 % 2 + (3 – 2)

Step	Conversion of infix to postfix	Remarks
1	7 + 5 * 3 / 5 ^ 1 % 2 + (3 - 2)	Parenthesis () has highest precedence. That's why the expression within the parenthesis will be converted first.
2	= 7 + 5 * 3 / 5 ^ 1 % 2 + [32-]	The postfix notation 32- will be treated as a single operand enclosed with square bracket. Now the exponential operator ^ has the highest priority
3	= 7 + 5 * 3 / [51^] % 2 + [32-]	Now *, / and % have the same highest priority. So as per the rule of associativity left to right evaluation will be performed. Hence the infix expression 5 * 3 will be converted to its postfix equivalent first.
4	= 7 + [53*] / [51^] % 2 + [32-]	According to the rule of associativity / operator will be converted now.
5	= 7 + [53*51^/] % 2 + [32-]	Now % operator having the highest priority will be dealt with.
6	= 7 + [53*51^/2%] + [32-]	Both of the operators are + with same precedence. Therefore leftmost + operator will be considered first.
7	= [753*51^/2%+] + [32-]	
8	= 753*51^/2%+32-+	

Note: After the conversion from infix notation to postfix notation, the generated postfix notation is treated as a single operand. To maintain the clarity the converted postfix expression is enclosed with square brackets.

Example3: Convert the following infix expression to its corresponding postfix expression.
A – (B / C + (D % E * F) / G) * H

Step	Conversion of infix to postfix	Remarks
1	A – (B / C + (D % E * F) / G) * H	
2	= A – (B / C + ([DE%] * F) / G) * H	Self explanatory
3	= A – (B / C + [DE%F*] / G) * H	Self explanatory
4	= A – ([BC/] + [DE%F*] / G) * H	Self explanatory
5	= A – ([BC/] + [DE%F*G/]) * H	Self explanatory
6	= A – [BC/DE%F*G/+] * H	Self explanatory
7	= A – [BC/DE%F*G/+H*]	Self explanatory
8	= ABC/DE%F*G/+H*-	Self explanatory

2. Infix to postfix conversion using stack – In computer the conversion of an infix expression to its equivalent postfix expression is done by using stack. Here for simplicity of the operation, only exponential (^), multiplication (*), division (/), modulus (%), addition (+) and subtraction (-) operators are considered in the infix expression. The algorithm to convert the infix expression to the postfix expression is given below.

Algorithm of Infix to Postfix conversion:

Step 1: Start

Step 2: Scan every character of the infix expression from left to right and repeat Step 3 to Step 6 until the end of the infix expression.

Step 3: If the character is an operand, then
- a) add the character to the postfix expression.

[End of If]

Step 4: If the character is a left parenthesis '(', then
- a) push the character on the stack.

[End of If]

Step 5: If the character is a right parenthesis ')', then
- a) Repeatedly pop the operators from the stack and add it to the postfix expression until a left parenthesis '(' is found.
- b) Discard or remove the left parenthesis '(' from the stack and do not add it to the postfix expression.

[End of If]

Step 6: If the character is an operator (^, *, /, %, +, -), then
- a) Repeatedly pop the operators from the stack and add each operator (popped from the stack) to the postfix expression which has the same or higher precedence than the current operator.
- b) Push the current operator on the stack.

[End of If]

Step 7: If the stack is not empty after scanning all the characters of the infix expression, then
- a) Repeatedly pop from the stack and add it to the postfix expression until the stack is empty.

[End of If]

Step 8: Stop

Example 4 – Convert the following infix expression to its equivalent postfix expression using stack.

A – (B / C + (D % E * F) / G) * H

Infix character scanned	Stack	Postfix expression
A		A
-	-	A
(	- (	A
B	- (	AB
/	- (/	AB
C	- (/	ABC
+	- (+	ABC/
(	- (+ (	ABC/
D	- (+ (	ABC/D
%	- (+ (%	ABC/D
E	- (+ (%	ABC/DE
*	- (+ (*	ABC/DE%
F	- (+ (*	ABC/DE%F
)	- (+	ABC/DE%F*
/	- (+ /	ABC/DE%F*
G	- (+ /	ABC/DE%F*G
)	-	ABC/DE%F*G/+
*	- *	ABC/DE%F*G/+
H	- *	ABC/DE%F*G/+H
Scan complete		**ABC/DE%F*G/+H*-**

Note: Here left side is considered as the bottom of the stack and the right side is treated as the top of the stack.

Example 5 – Convert the following infix expression to its equivalent postfix expression using stack.

K + L - M * N + (O ^ P) * W / U / V * T + Q

Infix character scanned	Stack	Postfix expression
K		K
+	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	- *	KL+M
N	- *	KL+MN
+	+	KL+MN*-
(	+ (	KL+MN*-
O	+ (	KL+MN*-O
^	+ (^	KL+MN*-O
P	+ (^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+ *	KL+MN*-OP^
W	+ *	KL+MN*-OP^W
/	+ /	KL+MN*-OP^W*
U	+ /	KL+MN*-OP^W*U
/	+ /	KL+MN*-OP^W*U/
V	+ /	KL+MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*+Q
Scan Complete		**KL+MN*-OP^W*U/V/T*+Q+**

C Program of Infix to Postfix Conversion

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define Max 100
#define MaxLen 50

char stack[Max];
int top = -1;

void Push(char );
int isFull();
char Pop();
int isEmpty();
int ExpValidity(char []);
void InfixToPostfix(char [],char []);
int Precedence(char );

int main()
{
        char infix[MaxLen], postfix[MaxLen];

        printf("Enter the Expression: ");
        gets(infix);

        if(ExpValidity(infix))
        {
                InfixToPostfix(infix, postfix);
                printf("The postfix expression: ");
                printf("%s", postfix);
        }
        else
        {
                printf("The expression is not valid");
        }
        printf("\n");

        return 0;
}

int ExpValidity(char infix[])
{
        int i, StrLen;
        char symbol;

        StrLen = strlen(infix);
```

```
for(i=0; i<StrLen; i++)
{
        symbol = infix[i];

        switch(symbol)
        {

                case '(':
                case '{':
                case '[': Push(symbol);
                        break;
                case ')': if(isEmpty())
                        {
                                printf("No. of right brackets is more than no. of left
                                brackets.\n");
                                return 0;
                        }

                        if(Pop() != '(')
                        {
                                printf("Bracket Mismatch.\n");
                                return 0;
                        }

                        break;

                case '}': if(isEmpty())
                        {
                                printf("No. of right brackets is more than no. of left
                                brackets.\n");
                                return 0;
                        }

                        if(Pop() != '{')
                        {
                                printf("Bracket Mismatch.\n");
                                return 0;
                        }

                        break;

                case ']': if(isEmpty())
                        {
                                printf("No. of right brackets is more than no. of left
                                brackets.\n");
                                return 0;
                        }
```

```
                                if(Pop() != '[')
                                {
                                        printf("Bracket Mismatch.\n");
                                        return 0;
                                }

                                break;

                        default: break;
                }
        }

        if(isEmpty())
                return 1;
        else
        {
                printf("No. of left brackets is more than no. of right brackets.\n");
                return 0;
        }
}


int Precedence(char symb)
{
        switch(symb)
        {
                case '^': return 3;
                case '*':
                case '/':
                case '%': return 2;
                case '+':
                case '-': return 1;
                default: return 0;
        }
}
```

```
void InfixToPostfix(char infix[],char postfix[])
{
        int i, j = 0, StrLen;
        char symbol, ch;

        StrLen = strlen(infix);

        for(i=0; i<StrLen; i++)
        {
                symbol = infix[i];

                switch(symbol)
                {

                        case ' ': break;
                        case '(': Push(symbol);
                                  break;
                        case ')': while((ch = Pop()) != '(')
                                          postfix[j++] = ch;
                                  break;
                        case '+':
                        case '-':
                        case '*':
                        case '/':
                        case '%':
                        case '^': while(!isEmpty() &&
                                          Precedence(symbol) <= Precedence(stack[top]))
                                          postfix[j++] = Pop();

                                  Push(symbol);
                                  break;

                        default: postfix[j++] = symbol;
                                 break;


                }
        }

        while(!isEmpty())
                postfix[j++] = Pop();

        postfix[j] = '\0';

}
```

```
void Push(char ch)
{
	if(isFull())
	{
		printf("STACK OVERFLOW\n");
		return;
	}

	stack[++top] = ch;
}

int isFull()
{
	if(top == Max - 1)
		return 1;
	else
		return 0;
}

char Pop()
{
	if(isEmpty())
	{
		printf("STACK UNDERFLOW\n");
		exit(1);
	}

	return stack[top--];
}

int isEmpty()
{
	if(top == -1)
		return 1;
	else
		return 0;
}
```

Evaluation of a postfix expression – It has been already discussed that a postfix expression can be evaluated in a single scan by using stack. This ease of postfix evaluation gives the way to convert an infix expression to its equivalent postfix expression first and then the value of the expression is evaluated using that postfix expression.

Using stack a postfix expression can be evaluated very easily in a single scan. Every character of the postfix expression is scanned from left to right. If the character is an operand, it is pushed into the stack. If the character is found to be an operator, then the top two operands are popped from the stack and the operator is applied on these popped two operand. Thus the result achieved is again pushed onto the stack. The algorithm to evaluate a postfix expression is given below.

Algorithm of Postfix Evaluation:

Step 1: Start

Step 2: Scan every character of the postfix expression from left to right and repeat Step 3 and Step 4 until the end of the postfix expression.

Step 3: If the character is an operand, then
a) push the character on the stack.
[End of If]

Step 4: If the character is an operator, then
a) pop the top two operands out of the stack into the two variables A and B in such a way that A becomes the topmost operand and B becomes the operand below the topmost operand.
b) apply the operator on A and B to evaluate (B operator A).
c) push the result of the evaluation i.e. (B operator A) on the stack.
[End of If]

Step 5: After scanning all the characters of the postfix expression, the stack should have only one element which is the ultimate result of the postfix expression. Pop the element from the stack and print it on the screen.

Step 6: Stop.

Example 6: Evaluate the following postfix expression using stack.

753*51^/+32-+

Postfix character scanned from left to right	Operation performed	Stack
7		7
5		7 5
3		7 5 3
*	5 * 3 = 15	7 15
5		7 15 5
1		7 15 5 1
^	5 ^ 1 = 5	7 15 5
/	15 / 5 = 3	7 3
+	7 + 3 = 10	10
3		10 3
2		10 3 2
-	3 – 2 = 1	10 1
+	10 + 1 = 11	11
Scan Complete		**Result = 11**

C Program of Postfix Evaluation

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define Max 100
#define MaxLen 50

char stack[Max];
int top = -1;

void Push(char );
int isFull();
char Pop();
int isEmpty();
int PostfixEvaluation(char []);
```

```
int main()
{
	char postfix[MaxLen];
	int value;
	printf("Enter the postfix expression: ");
	gets(postfix);
	value = PostfixEvaluation(postfix);
	printf("The value of the expression: %d\n", value);
	return 0;
}

int PostfixEvaluation(char postfix[])
{
	int i, StrLen, a, b;
	char symbol;
	StrLen = strlen(postfix);
	for(i=0; i<StrLen; i++)
	{
		symbol = postfix[i];

		if(symbol >= '0' && symbol <= '9')
		{
			Push(symbol - '0');
		}
		else
		{
			a = Pop();
			b = Pop();
			switch(symbol)
			{
				case '+': Push(b + a);
					break;
				case '-': Push(b - a);
					break;
				case '*': Push(b * a);
					break;
				case '/': Push(b / a);
					break;
				case '%': Push(b % a);
					break;
				case '^': Push(pow(b, a));
					break;
				default: printf("Wrong postfix expression.\n");
					break;
			}
		}
	}
```

```
        return Pop();
}

void Push(char ch)
{
        if(isFull())
        {
                printf("STACK OVERFLOW\n");
                return;
        }

        stack[++top] = ch;
}

int isFull()
{
        if(top == Max - 1)
                return 1;
        else
                return 0;
}



char Pop()
{
        if(isEmpty())
        {
                printf("STACK UNDERFLOW\n");
                exit(1);
        }

        return stack[top--];
}

int isEmpty()
{
        if(top == -1)
                return 1;
        else
                return 0;
}
```

Now we have reached to a position where we have completed the program of checking the validity of an infix expression, the program of infix to postfix conversion and the program of postfix evaluation. We are familiar with the infix expression from very beginning. Therefore the ultimate goal is to implement a program where an infix expression will be given by the user and the program will evaluate the infix expression. The intermediate steps of infix to postfix conversion will remain hidden to the user. Now the sequence of steps to achieve this goal is given below.

Step 1: Check the validity of the infix expression.

Step 2: If the infix expression is valid, then
perform infix to postfix conversion
otherwise
Print “Invalid infix expression” and go to Step 4.

Step 3: Evaluate the converted postfix expression and display the value on the screen.

Step 4: Stop

According to the above mentioned sequence of steps the three already implemented programs (validity of infix expression, infix to postfix conversion, evaluation of postfix expression) is to be assembled to implement the program of infix evaluation which is given below.

C Program of Infix Evaluation

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define Max 100
#define MaxLen 50

char stack[Max];
int top = -1;

void Push(char );
int isFull();
char Pop();
int isEmpty();
int ExpValidity(char []);
void InfixToPostfix(char [],char []);
int Precedence(char );
int PostfixEvaluation(char []);
```

```
int main()
{
	char infix[MaxLen], postfix[MaxLen];
	int  value;
	printf("Enter the Expression: ");
	gets(infix);

	if(ExpValidity(infix))
	{
		InfixToPostfix(infix, postfix);
		printf("The postfix expression: ");
		printf("%s", postfix);
		value = PostfixEvaluation(postfix);
		printf("\nThe value of the expression: %d", value);

	}
	else
	{
		printf("The expression is not valid");
	}
	printf("\n");
	return 0;
}

int ExpValidity(char infix[])
{
	int i, StrLen;
	char symbol;

	StrLen = strlen(infix);

	for(i=0; i<StrLen; i++)
	{
		symbol = infix[i];

		switch(symbol)
		{

			case '(':
			case '{':
			case '[': Push(symbol);
				  break;
			case ')': if(isEmpty())
				  {
					printf("No. of right brackets is more than no. of left
					brackets.\n");
					return 0;
				  }
```

```
                                if(Pop() != '(')
                                {
                                        printf("Bracket Mismatch.\n");
                                        return 0;
                                }
                                break;

                        case '}': if(isEmpty())
                                {
                                        printf("No. of right brackets is more than no. of left
                                        brackets.\n");
                                        return 0;
                                }

                                if(Pop() != '{')
                                {
                                        printf("Bracket Mismatch.\n");
                                        return 0;
                                }
                                break;

                        case ']': if(isEmpty())
                                {
                                        printf("No. of right brackets is more than no. of left
                                        brackets.\n");
                                        return 0;
                                }

                                if(Pop() != '[')
                                {
                                        printf("Bracket Mismatch.\n");
                                        return 0;
                                }
                                break;

                        default: break;
                }
        }

        if(isEmpty())
                return 1;
        else
        {
                printf("No. of left brackets is more than no. of right brackets.\n");
                return 0;
        }
}
```

```
void InfixToPostfix(char infix[],char postfix[])
{
        int i, j = 0, StrLen;
        char symbol, ch;
        StrLen = strlen(infix);

        for(i=0; i<StrLen; i++)
        {
                symbol = infix[i];
                switch(symbol)
                {
                        case ' ': break;
                        case '(': Push(symbol);
                                  break;
                        case ')': while((ch = Pop()) != '(')
                                          postfix[j++] = ch;
                                  break;
                        case '+':
                        case '-':
                        case '*':
                        case '/':
                        case '%':
                        case '^': while(!isEmpty() && Precedence(symbol) <=
                                          Precedence(stack[top]))
                                          postfix[j++] = Pop();

                                  Push(symbol);
                                  break;

                        default: if(symbol >= '0' && symbol <= '9')
                                 {
                                          postfix[j++] = symbol;
                                          break;
                                 }
                                 else
                                 {
                                          printf("Operands of infix expression is not
                                          numbers.\n");
                                          exit(1);
                                 }
                }
        }

        while(!isEmpty())
                postfix[j++] = Pop();

        postfix[j] = '\0';
}
```

```
int Precedence(char symb)
{
	switch(symb)
	{
		case '^': return 3;
		case '*':
		case '/':
		case '%': return 2;
		case '+':
		case '-': return 1;
		default: return 0;
	}
}

int PostfixEvaluation(char postfix[])
{
	int i, StrLen, a, b;
	char symbol;
	StrLen = strlen(postfix);
	for(i=0; i<StrLen; i++)
	{
		symbol = postfix[i];
		if(symbol >= '0' && symbol <= '9')
		{
			Push(symbol - '0');
		}
		else
		{
			a = Pop();
			b = Pop();
			switch(symbol)
			{
				case '+': Push(b + a);
					break;
				case '-': Push(b - a);
					break;
				case '*': Push(b * a);
					break;
				case '/': Push(b / a);
					break;
				case '%': Push(b % a);
					break;
				case '^': Push(pow(b, a));
					break;
				default: printf("Wrong postfix expression.\n");
					break;
			}
		}
	}
	return Pop();
}
```

```
void Push(char ch)
{
	if(isFull())
	{
		printf("STACK OVERFLOW\n");
		return;
	}

	stack[++top] = ch;
}

int isFull()
{
	if(top == Max - 1)
		return 1;
	else
		return 0;
}

char Pop()
{
	if(isEmpty())
	{
		printf("STACK UNDERFLOW\n");
		exit(1);
	}

	return stack[top--];
}

int isEmpty()
{
	if(top == -1)
		return 1;
	else
		return 0;
}
```

3) Prefix Expression – In case of prefix expression, the operator is placed before the operands. For example – If the infix expression (A + B) is converted to prefix expression, then it becomes +AB.

Conversion of infix expression to prefix expression – Like the conversion of infix to postfix, the infix to prefix conversion can be done in two ways – either manually or using stack.

1. Infix to prefix conversion manually – In manual procedure the expression with highest precedence operator will be converted first to its equivalent prefix expression. If two or more operators have the same highest priority, then the leftmost operator will be considered first for infix to prefix conversion as per the rules of associativity. For better understanding let's take some examples.

Example 7: Convert the following infix expression to its corresponding prefix expression.

A – (B / C + (D % E * F) / G) * H

Step	Conversion of infix to prefix	Remarks
1	A – (B / C + (D % E * F) / G) * H	
2	= A – (B / C + ([%DE] * F) / G) * H	Self explanatory
3	= A – (B / C + [*%DEF] / G) * H	Self explanatory
4	= A – ([/BC] + [*%DEF] / G) * H	Self explanatory
5	= A – ([/BC] + [/*%DEFG]) * H	Self explanatory
6	= A – [+/BC/*%DEFG] * H	Self explanatory
7	= A – [*+/BC/*%DEFGH]	Self explanatory
8	= -A*+/BC/*%DEFGH	Self explanatory

2. Infix to prefix conversion using stack – In computer the infix to prefix conversion is done following the steps below.

Step 1: Reverse the infix expression from right to left.

Step 2: Exchange the left and right parenthesis i.e. left parenthesis '(' will be converted to right parenthesis ')' and right parenthesis ')' will be converted to left parenthesis '('.

Step 3: Convert the reversed infix expression to nearly postfix expression. Here the word "nearly" implies that the operators are popped from the stack until they have the higher precedence than the current operator.

Step 4: Again reverse the postfix expression to get the ultimate prefix expression.

Example 8: Convert the following infix expression to its corresponding prefix expression using stack.

A – (B / C + (D % E * F) / G) * H

Step 1: Reversed infix expression – H *) G /) F * E % D (+ C / B (- A

Step 2: Exchange of parenthesis – H * (G / (F * E % D) + C / B) - A

Step 3: Infix to nearly postfix conversion – The reversed infix expression is given below.

H * (G / (F * E % D) + C / B) - A

Infix character scanned	Stack	Nearly Postfix expression
H		H
*	*	H
(	* (	H
G	* (	HG
/	* (/	HG
(	* (/ (	HG
F	* (/ (	HGF
*	* (/ (*	HGF
E	* (/ (*	HGFE
%	* (/ (* %	HGFE
D	* (/ (* %	HGFED
)	* (/	HGFED%*
+	* (+	HGFED%*/
C	* (+	HGFED%*/C
/	* (+ /	HGFED%*/C
B	* (+ /	HGFED%*/CB
)	*	HGFED%*/CB/+
-	-	HGFED%*/CB/+*
A	-	HGFED%*/CB/+*A
Scan complete		HGFED%*/CB/+*A-

Step 4: Reverse the postfix expression to get the prefix expression.

Therefore the prefix expression: **- A*+/BC/*%DEFGH**

Example 9: Convert the following infix expression to its corresponding prefix expression using stack.

7 + 5 * 3 / 5 ^ 1 + (3 – 2)

Step 1: Reversed infix expression –) 2 – 3 (+ 1 ^ 5 / 3 * 5 + 7

Step 2: Exchange of parenthesis – (2 – 3) + 1 ^ 5 / 3 * 5 + 7

Step 3: Infix to nearly postfix conversion – The reversed infix expression is given below.

(2 – 3) + 1 ^ 5 / 3 * 5 + 7

Infix character scanned	**Stack**	**Nearly Postfix expression**
(	(	
2	(	2
-	(-	2
3	(-	23
)		23-
+	+	23-
1	+	23-1
^	+ ^	23-1
5	+ ^	23-15
/	+ /	23-15^
3	+ /	23-15^3
*	+ / *	23-15^3
5	+ / *	23-15^35
+	+ +	23-15^35*/
7	+ +	23-15^35*/7
Scan complete		23-15^35*/7++

Step 4: Reverse the nearly postfix expression to get the prefix expression.

Therefore the prefix expression: **++7/*53^51-32**

Evaluation of a prefix expression – The evaluation of a prefix expression is similar to the procedure of postfix evaluation. Only the difference is that every character of the prefix expression is scanned from right to left. If the character is an operand, it will be pushed on the stack and if the character is an operator, the operator is applied on the two top elements of the stack. Let's take an example to clarify this concept.

Example 10: Evaluate the following prefix expression using stack.

++7/*53^51-32

Prefix character scanned from right to left	Operation performed	Stack
2		2
3		2 3
-	3 – 2 = 1	1
1		1 1
5		1 1 5
^	5 ^ 1 = 5	1 5
3		1 5 3
5		1 5 3 5
*	5 * 3 = 15	1 5 15
/	15 / 5 = 3	1 3
7		1 3 7
+	7 + 3 = 10	1 10
+	10 + 1 = 11	11
Scan Complete		**Result = 11**

Chapter 5

Queue

1) Simple Queue or Queue – A simple queue or queue is a linear data structure in which the insertion is performed at one end called rear of the queue and deletion is performed at the other end called front of the queue. That means the insertion of an element into the queue happens only at the rear end and the removal of the element happens only from the beginning or front end of the queue. That means, the element which is inserted first at the rear end, will be deleted first from the front end of the queue. Therefore the queue is called FIFO (first in first out) data structure. Real life example of the queue is – the queue in front of the ticket counter in which the person joins the queue first at the rear end will get his ticket first and will leave the queue first from the front end of the queue. A pictorial view of a queue with 5 elements is shown in Fig.5.1.

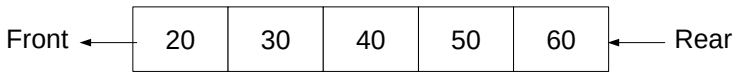


Fig.5.1: A queue of 5 elements with front and rear end

Various types of operations can be performed on a queue, such as – insert, delete, peek etc. The purpose of a insert operation is to add an element at the rear end of a queue. The delete operation removes one element from the front end of the queue. The function of peek is to retrieve the front/ first element of the queue and display it. Like stack, queue can be implemented either by using array or linked list. In this chapter the above mentioned basic operations of queue are implemented using array first, then these functions will be realized using linked list. Before discussing different operations on queue it is important to explain how a queue can be initialized using array.

Initialization of queue using array – In case of array implementation a queue is represented by a one dimensional array where each cell of the array is capable to hold one element. That means the first element of the queue is placed in the cell with index 0, the second element is placed in the cell with index 1 and so on. It is clear from this discussion that the side of the array towards the index 0 is treated as front end of the qeue and the opposite side of the array is considered the rear end of the queue.

If queue is implemented by an array named ‘queue’, then it will be declared globally outside the main function by the following statement.

```
Date_type  queue[10];
```

As insertion of an element is possible at the rear end of the queue, we need a variable ‘rear’ which will be used to keep track of the last inserted element of the queue. Similarly to keep track of the front element of the queue we require another variable ‘front’. Basically here the variable ‘rear’ will always hold the index value of the last element of the queue and the variable ‘front’ will always hold the index of the first element of the queue. Now initially the queue will be empty. The variables ‘front’ and ‘rear’ will be initialized globally with 0 and -1 respectively. When an element is inserted into the queue, the variable ‘rear’ will be incremented by one and the variable ‘front’ remains unchanged. When the element is removed from the front end of the queue, the variable ‘front’ will be incremented by one and the variable ‘rear’ is kept unchanged. That means, insertion operation of a queue is related to the variable ‘rear’ and the deletion of an element from the queue is related to the variable ‘front’.

The declaration of the queue using array along with the initialization of the variables 'front' and 'rear' are given below.

```
#define Max 5
Data_type queue[Max];
int front = 0;
int rear = -1;
```

Now the array 'queue' along with the variables 'front' and 'rear' will be utilized by the different user-defined functions like insert, delete, isEmpty, isFull, peek etc. For this reason the array 'queue' and the variables 'front' and 'rear' are declared globally outside the main function to avoid the passing the entire queue array and the 'front' and 'rear' variables every time during the use of these functions. The global declaration of the queue is shown below for clarification.

```
#include<stdio.h>
#define Max 5

Data_type queue[Max];
int front = 0;
int rear = -1;

int main()
{
        .
        .
        .
        .
}
```

Operation on Queue – Basically there are two operations which are performed on a queue - 1. Insert and 2. Delete. Except these two operations we have other two functions related to a queue - 3. isFull and 4. isEmpty. There are more two functions – 5. Peek which is used to get the front or first element of a queue and 4. Display which is used to print all the elements of a queue. All these functions are explained one by one below.

1. Insert – In this operation an element is inserted or added to the queue at the rear end. If the queue is implemented using array, we know that the variable rear is used to store the index value at the rear end. Initially when the queue is empty, the value of the rear is made -1. As one element is inserted into the queue the value of rear is incremented by 1 and it becomes 0. If we insert another element, the value of the rear is again incremented by 1. Thus the value of the variable 'rear' is incremented by 1 each time as soon as a new element is added at the rear end of the queue, but the other variable 'front' is made unchanged during the insertion operation. If the insertion of elements are being continued to the queue, a time will come when the value of the rear will become (Max – 1), where Max is the maximum size of the array used to represent a queue. In this situation we can not insert element further and the queue is said to be full. This condition is known as overflow condition of the queue. The insertion of elements into a queue is shown in Fig.5.2 using an array with maximum size 5.

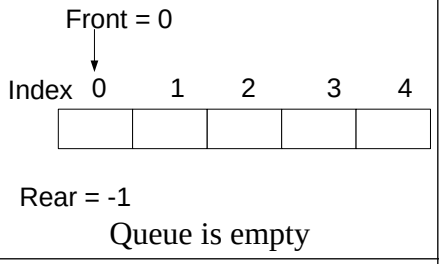
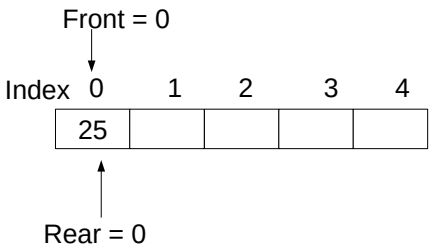
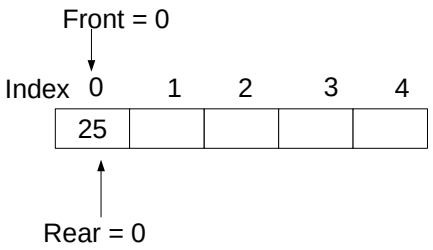
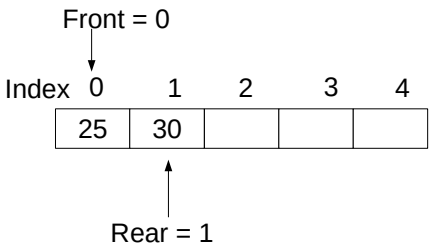
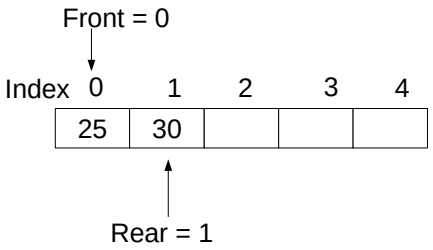
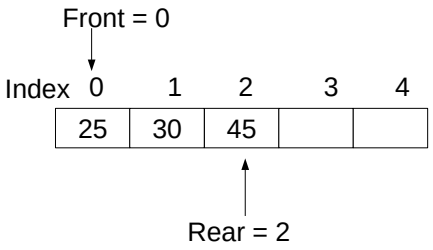
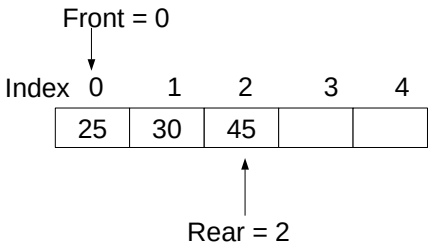
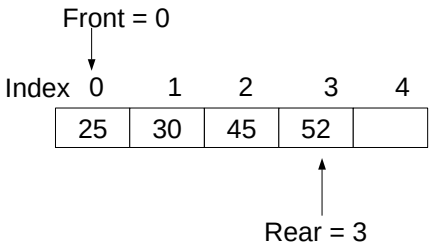
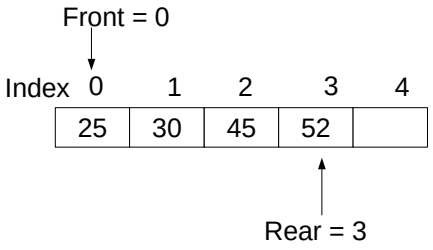
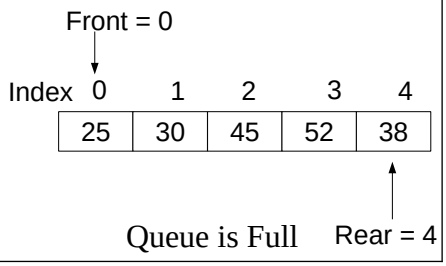
SL.	New element to be inserted	Before insertion	After insertion
1	25		
2	30		
3	45		
4	52		
5	38		

Fig.5.2: Insertion of elements into a queue

It is clear from the above discussion that element can not be inserted when the queue is full. Therefore before performing insertion operation it is mandatory to check that the queue is full or not. If it is full, no element will be allowed to insert into the queue and if the queue is not full, the variable rear will be incremented by 1 first, then the new element should be inserted or stored into the array of queue.

Algorithm of insert operation:

Step 1: Start

Step 2: If the queue is Full, then
a) Print 'QUEUE OVERFLOW'.
b) Go to Step 5.
[End of If]

Step 3: Set rear = rear + 1

Step 4: Set Queue[rear] = Value

Step 5: Stop

User defined function in C to insert an element into the queue

```
void insert(int element)
{
        if(isFull())
        {
                printf("QUEUE OVEFLOW\n\n");
                return;
        }

        queue[++rear] = element;
        printf("The element has been inserted successfully into the queue\n\n");
}
```

2. Delete – The delete operation removes the front or first element from the queue. As soon as the element is removed from the front end of the queue, the variable 'front' is incremented by one to the next index of the array. So initially the front was zero. After the deletion of the first element the value of front becomes 1 which is the index of the next element. If another element is deleted from the queue, the front becomes 2. In this way if the elements of the queue are removed one by one, a time will come when the value of the front and rear will become equal. When front and rear become equal, the queue will hold one element which is the last element of the queue. Now if this last element is deleted again, the queue will be empty and the value of the front will become (rear + 1). In this condition we can not remove element any more. This empty condition of the queue is known as underflow condition. Therefore it is essential to check the underflow condition of the queue before deleting an element. The deletion operation is shown pictorially in Fig.5.3 for a partially filled queue.

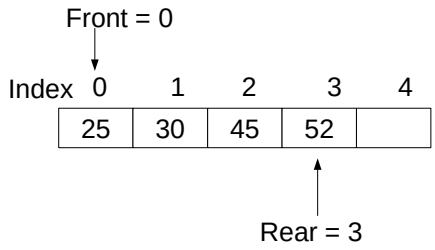
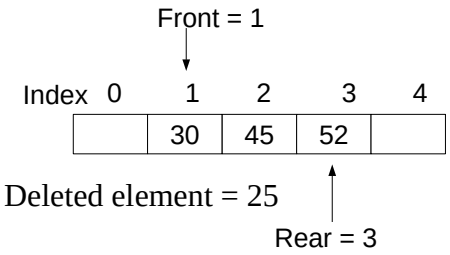
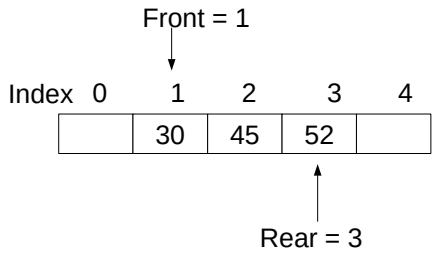
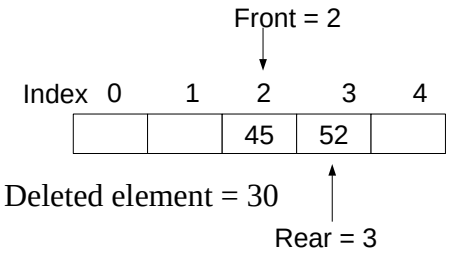
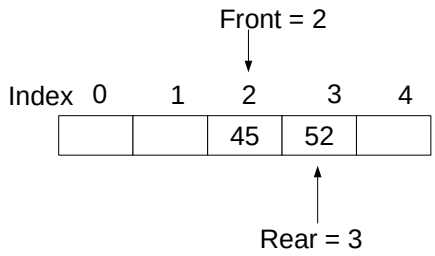
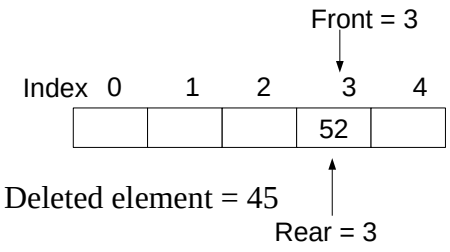
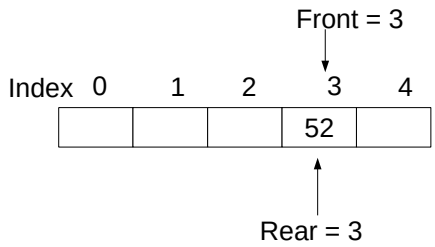
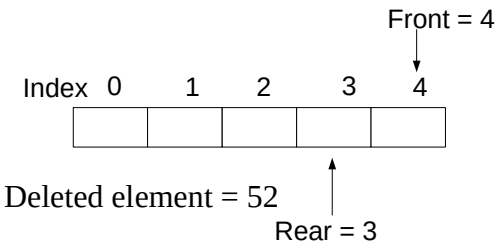
SL.	Before deletion	After deletion
1		
2		
3		
4		

Fig.5.3: Deletion of elements from a queue

Algorithm of delete operation:

Step 1: Start

Step 2: If the queue is Empty, then
a) Print 'QUEUE UNDERFLOW'.
b) Go to Step 5.
[End of If]

Step 3: Set Value = Queue[front]

Step 4: Set front = front + 1

Step 5: Stop

User defined function in C to delete an element from the queue

```
int delete()
{
        if(isEmpty())
        {
                printf("QUEUE UNDERFLOW\n\n");
                exit(1);
        }

        printf("The element has been deleted successfully from the queue\n\n");
        return queue[front++];
}
```

3. isFull – In case of array implementation of queue, the queue is initialized as an array with its maximum size. During initialization of a queue the maximum size is defined by a macro 'Max' which is equal to 5. Therefore the queue can hold maximum 5 number of data. If 6th data is attempted to insert into the queue, this will not be allowed. This situation is known as "QUEUE OVERFLOW". The operation "isFull" checks that the queue is already full or not. If the rear variable is equal to the maximum index value which is (Max – 1), the "isFull" function returns 1 to indicate that the queue is full. If rear is less than (Max – 1), it returns 0 to show that the queue is not full. The algorithm for isFull operation is given below.

Algorithm of isFull operation:

Step 1: Start

Step 2: If rear = Max – 1, then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isFull operation

```
int isFull()
{
        if(rear == Max - 1)
                return 1;
        else
                return 0;
}
```

4. isEmpty – If there is no element in the queue, the queue is said to be empty. The isEmpty operation checks whether the queue is empty or not. In case of array implementation of the queue, the front variable is initialized to 0. In Fig.5.3 it is being observed that when front exceeds the value of rear i.e. front = rear + 1, the queue becomes empty. The empty status of a queue is known as "QUEUE UNDERFLOW". If (front > rear) or (front = rear + 1), it implies that the queue is empty and the function isEmpty returns 1, otherwise the function returns 0.

Algorithm of isEmpty operation:

Step 1: Start

Step 2: If (front = rear + 1) or (front > rear), then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isEmpty operation

```
int isEmpty()
{
        if(front > rear)
                return 1;
        else
                return 0;
}
```

5. Peek – Peek operation gives the value of the first element of the queue without removing it from the queue. That means, the peek operation return the first element of the queue if the queue is not empty. If the queue is empty, it display an error message "QUEUE IS EMPTY". Therefore the front variable will not be changed for peek operation.

Algorithm of peek operation:

Step 1: Start

Step 2: If the queue is Empty, then
a) Print 'QUEUE UNDERFLOW'.
b) Go to Step 4.
[End of If]

Step 3: Return Queue[front]

Step 4: Stop

User defined function in C to implement peek operation for a queue

```
int peek()
{
        if(isEmpty())
        {
                printf("Queue is empty\n\n");
                exit(1);
        }
        return queue[front];
}
```

6. Display – Display function prints all the elements of a queue. Before printing the elements it checks the queue is empty or not. If the queue is empty, it prints an error message, otherwise it prints all the elements of the queue i.e. from first element (queue[front]) to the last element (queue[rear]). To do this a loop should be executed from i = front to i = rear. The algorithm of display function is given below.

Algorithm of display function:

Step 1: Start

Step 2: If the queue is Empty, then
a) Print 'QUEUE UNDERFLOW'.
b) Go to Step 7.
[End of If]

Step 3: Set i = front

Step 4: Repeat Step 5 to Step 6 while i <= rear

Step 5: Print Queue[i]

Step 6: Set i = i + 1

Step 7: Stop

User defined function in C to display all the elements of a queue

```
void display()
{
        int i;

        if(isEmpty())
        {
                printf("Queue is empty");
        }
```

```
        else
        {
                for(i=front; i<=rear; i++)
                        printf("%d  ", queue[i]);
        }
}
```

Now the entire C program to incorporate all the above mentioned functions like insert, delete, isFull, isEmpty, peek, display are given below.

C program to insert, delete, peek the first element and display all the elements of a queue

```
#include<stdio.h>
#include<stdlib.h>
#define Max 100
int queue[Max];
int front = 0;
int rear = -1;
int UnderflowFlag = 0;

void insert(int );
int isFull();
int delete();
int isEmpty();
int peek();
void display();

int main()
{
        int choice, element, value, firstelem;

        while(1)
        {


printf("*****************************************************************\n");
                printf("0: Terminate the program\n");
                printf("1: Insert an element into the queue\n");
                printf("2: Delete an element from the queue\n");
                printf("3: Display the first element of the queue\n");
                printf("4: Display all the elements of the queue\n");

printf("*****************************************************************\n\n");
```

```
                printf("Enter your option: ");
                scanf("%d",&choice);
                printf("\n");

                switch(choice)
                {
                        case 0: exit(1);

                        case 1: printf("Enter the element to be inserted into the queue: ");
                                scanf("%d", &element);
                                insert(element);
                                break;

                        case 2: value = delete();
                                if(!UnderflowFlag)
                                printf("The element %d has been deleted succeessfully.\n\n", value);
                                break;

                        case 3: firstelem = peek();
                                if(!UnderflowFlag)
                                printf("The first element of the queue is %d.\n\n", firstelem);
                                break;

                        case 4: printf("The elements of the queue:\n\n");
                                display();
                                break;

                        default: printf("Wrong option is selected\n\n");
                                 break;
                }
        }
        return 0;
}

void insert(int element)
{
        if(isFull())
        {
                printf("QUEUE OVEFLOW\n\n");
                return;
        }

        queue[++rear] = element;
        printf("The element has been inserted successfully into the queue\n\n");
}
```

```
int isFull()
{
	if(rear == Max -1)
		return 1;
	else
		return 0;
}

int delete()
{
	if(isEmpty())
	{
		printf("QUEUE UNDERFLOW\n\n");
		UnderflowFlag = 1;
		return 0;
	}

	UnderflowFlag = 0;
	return queue[front++];
}

int isEmpty()
{
	if(front > rear)
		return 1;
	else
		return 0;
}

int peek()
{
	if(isEmpty())
	{
		printf("Queue is empty\n\n");
		UnderflowFlag = 1;
		return 0;
	}

	UnderflowFlag = 0;
	return queue[front];
}
```

```
void display()
{
        int i;

        if(isEmpty())
        {
                printf("Queue is empty");
        }
        else
        {
                for(i=front; i<=rear; i++)
                        printf("%d  ", queue[i]);
        }
        printf("\n\n");
}
```

Note: *In the above program a global variable 'underflowflag' is used to return the program sequence to the main function if "QUEUE UNDERFLOW" condition happens. For example in case of delete function underflow condition may happen any time. In this situation the remaining part of the delete function should not be executed and the program sequence should return back to the main function. As the returned data type of the delete function is int, we can not use 'return' to exit from the delete function. We have to use either 'exit(1)' or 'return 0'. But 'exit(1)' terminates the program completely which is not desired at all. On the other hand 'return 0' returns zero to the main, which will be considered as the deleted element, although no element has been deleted due to the underflow state of the queue. To refrain from this consequence, underflowflag is used in the main. If underflowflag = 1, it will show "Queue is empty" and if underflowflag = 0, then the deleted element will be stored into a variable 'value' and will be displayed.*

Drawback of queue – In array implementation of a queue if elements are inserted continuously, the queue will be full ultimately when the value of rear will reach to (Max – 1). In this situation if some elements are deleted from the queue, some empty cells will be generated starting from the front end of the queue. Consequently this queue does not remain empty, it has some spaces for storing elements once again. In spite of some existing vacant cells in the queue new element can not be inserted any more as the rear variable has reached to the maximum index of the array. Even if the queue is made completely empty by eliminating all the elements, it is not possible to add any element further. Therefore these spaces inside the queue remains unused. This situation is demonstrated in Fig.5.4 for better understanding.

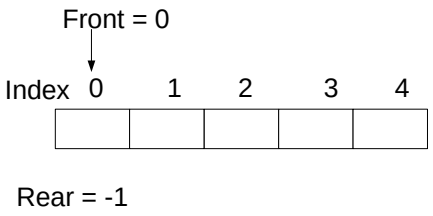
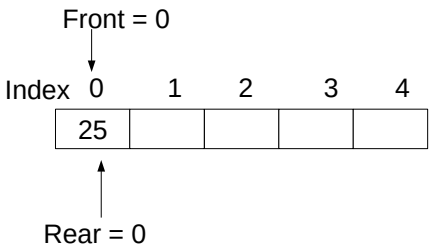
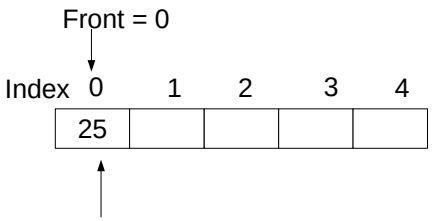
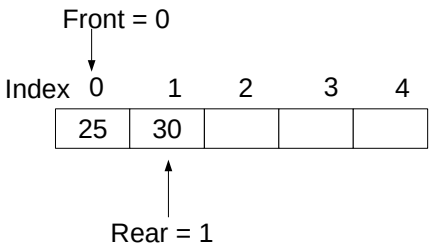
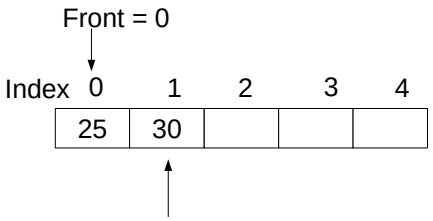
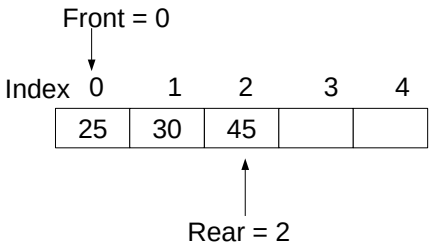
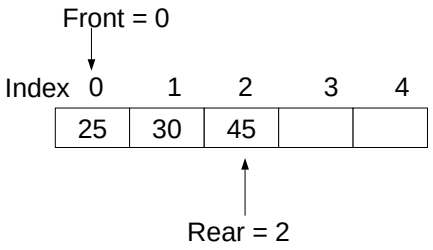
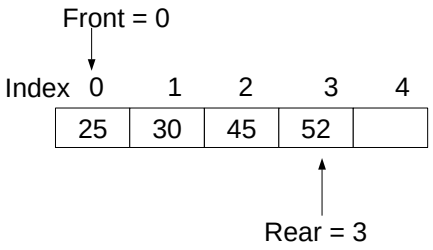
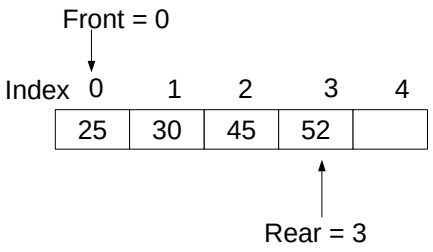
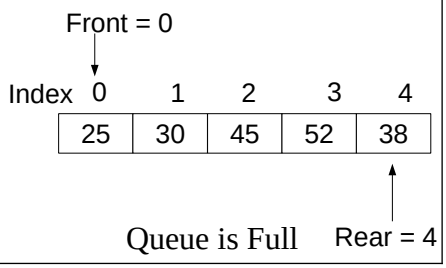
SL.	New element to be inserted	Before insertion	After insertion
1	25		
2	30		
3	45		
4	52		
5	38		

Fig.5.4: Insertion of elements into a queue

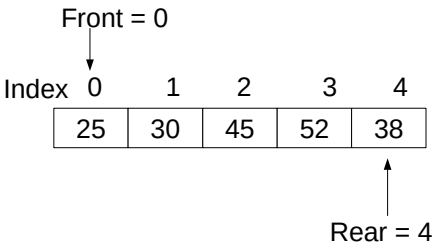
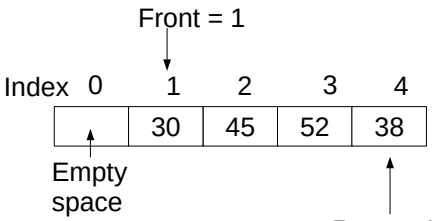
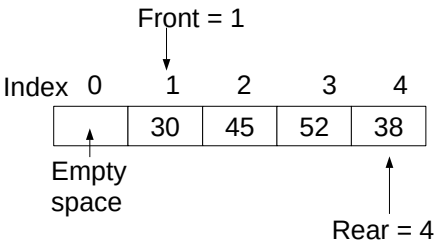
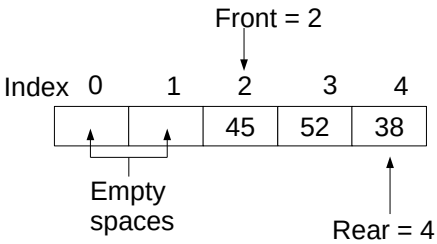
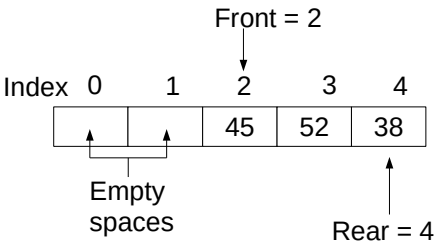
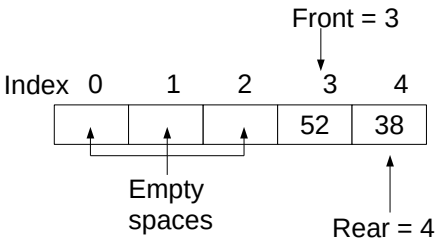
SL.	Before deletion	After deletion
1		 Deleted element = 25
2		 Deleted element = 30
3		 Deleted element = 45 In spite of these 3 empty cells, no element can be inserted further

Fig.5.4: Deletion of elements from the queue to create empty spaces

To resolve this limitation of queue, circular queue is introduced. In circular queue the unused empty cells are reused by the newly inserted elements. The detail discussion of circular queue is covered in the next section.

Other types of Queue – There are other various types of queue except simple queue as discussed above.

1) Circular Queue 2) Deque (Double ended queue) 3) Priority Queue

2) Circular Queue – A circular queue is a special type of queue where the first index comes just after the last index. This implies that if the array size is Max, then the next index of the index = Max – 1 is 0 i.e. if index = Max – 1, then (index + 1) = 0, not Max. This orientation forms a circular structure of a queue. Therefore a circular queue is visualized as shown in Fig.5.5.

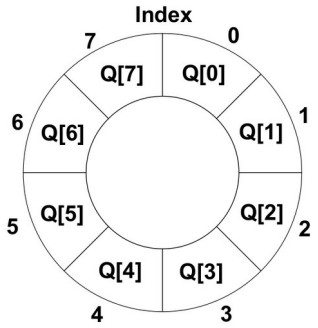


Fig.5.5: Array implementation of circular queue

In the above figure a circular queue has been realized using an array of 8 elements (Max = 8). If we move in clockwise direction, index 0 comes after index 7 (Max – 1), whereas index 0 is the first index and index 7 is the last index. If rear = 7 and the circular queue is not full, then rear will be changed to 0 to insert a new element into the queue. Similarly if front = 7 and the circular queue is not empty, the front will be modified to 0 after the deletion of an element. It is evident that to insert an element into the circular queue the queue is full or not should be checked first. Similarly "a circular queue is empty or not" – this should be verified before the deletion of an element. Therefore it is important to develop isFull/ isEmpty function before insertion/ deletion of an element.

isFull and isEmpty status of a circular queue – These two functions checks the current status of a circular queue i.e. isFull function checks the overflow condition of a queue and isEmpty function checks the underflow condition of a queue – isFull function returns true if the circular queue is full and isEmpty function returns true if the circular queue is empty. Now we have to develop two different conditions for checking these two status (either empty or full) of a circular queue.

isEmpty conditions for a circular queue: There are three conditions for which the circular queue will be considered as empty.

Condition 1 for empty: Initially when he circular queue is empty, the values of front and rear are 0 and -1 respectively. We know front and rear hold the index of the circular queue array. As -1 is an invalid index for an array in C, the rear = -1 is not a valid index initially. Therefore rear = -1 is one of the conditions for which the queue will be empty shown in Fig.5.6.

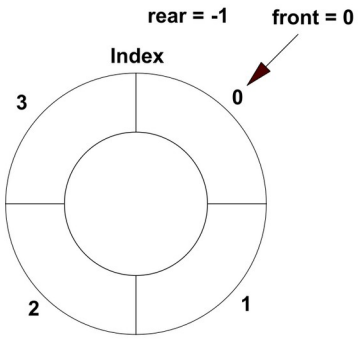
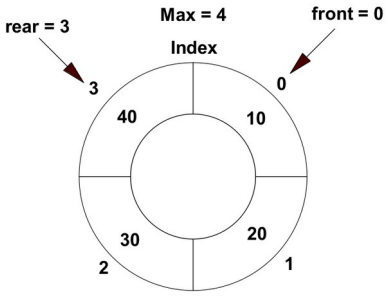
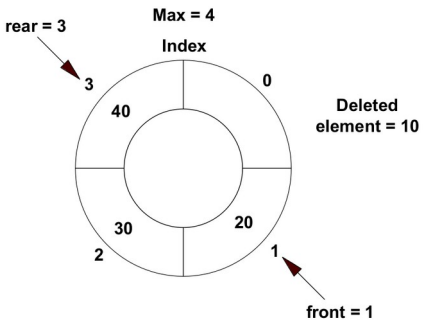
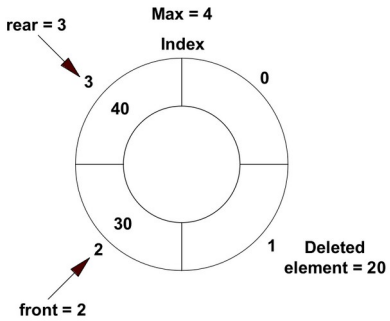


Fig.5.6: Empty circular queue with front = 0 and rear = -1

Condition 2 for empty: Suppose a circular queue is full completely with front = 0 and rear = Max – 1. If all the elements are started to remove by increasing the front by 1 and keeping the rear unchanged, a time will come when front will become (Max – 1) and rear will be also (Max – 1). This situation implies that the circular queue has only one element. If this last element is also deleted, the front will become 0 once again according to the protocol of a circular queue. Thus the second condition for a circular queue of being empty is front = 0 and rear = Max – 1. This situation is demonstrated in the following figure for clarification.

1.	
2.	
3.	

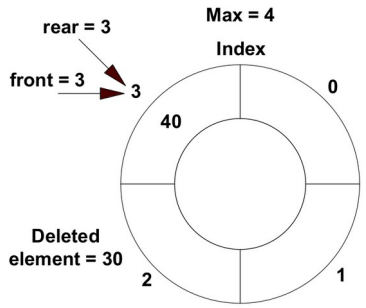
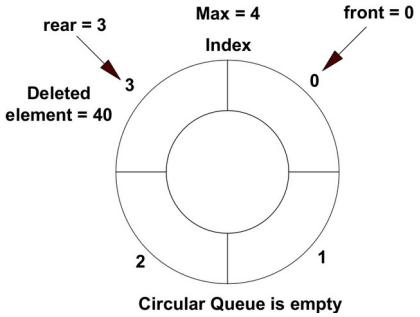
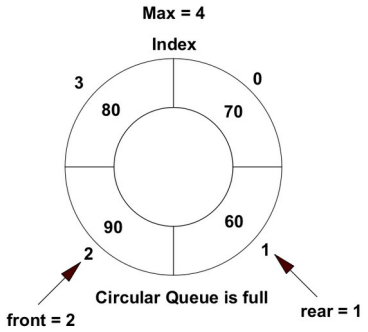
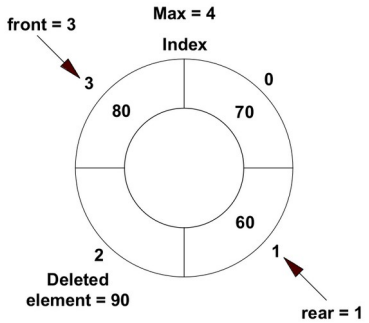
4.	
5.	

Fig.5.7: Sequence of deletion for condition2

Condition 3 for empty: Suppose a circular queue is full completely with front = rear + 1 assuming front ≠ 0 and rear ≠ Max − 1. In this situation if the queue is made empty by deleting all the elements one by one, the front will be (rear + 1) once again. Therefore the third condition for the circular queue of becoming empty is front = rear + 1. Condition 3 has been explained in Fig.5.8 below.

1.	
2.	

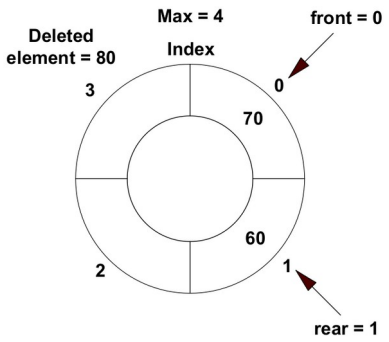
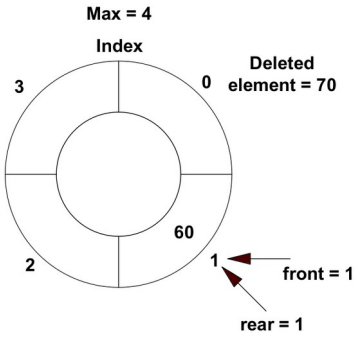
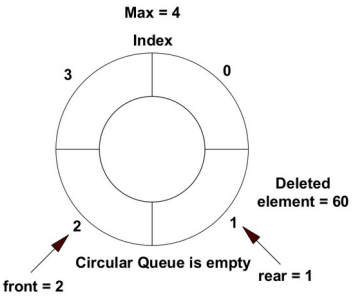
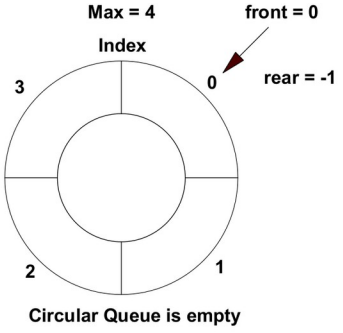
3.	
4.	
5.	

Fig.5.8: Sequence of deletion for condition3

isFull conditions for a circular queue: The two conditions for which a circular queue will be treated as full are given below.

Condition 1 for full: Initially the queue is empty with front = 0 and rear = -1. If the elements are added one by one by increasing the value of rear and keeping the value of front unchanged, the queue will be filled up by elements when the front and the rear will become 0 and (Max – 1) respectively. So the first condition of circular queue to be full is front = 0 and rear = Max – 1. This situation is shown in Fig.5.9 below.

1.	

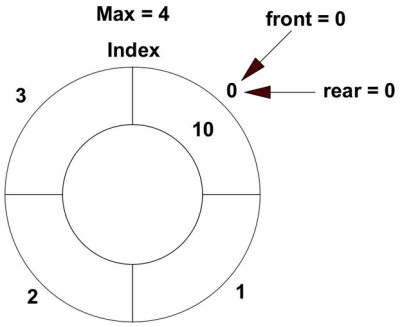
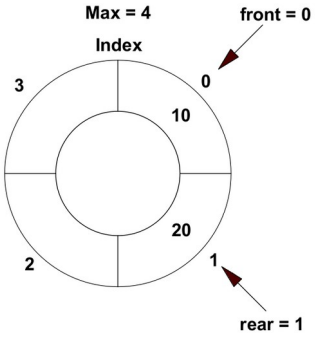
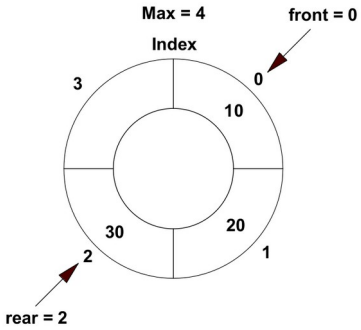
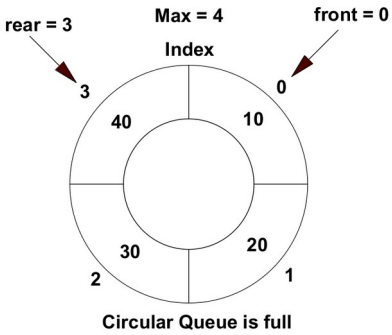
2.	
3.	
4.	
5.	

Fig.5.9: Sequence of insertion for condition 1

Condition 2 for full: Suppose a circular queue is holding a single element with front = rear assuming front = rear ≠ 0. In this situation if the queue is made full by inserting elements one by one, the rear will reach to the index just before the index of front i.e. front = rear + 1 which is the another condition for full circular queue. This situation is also demonstrated step by step in Fig.5.10 below.

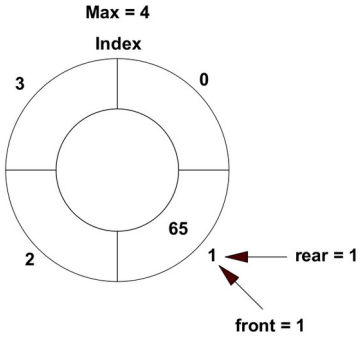
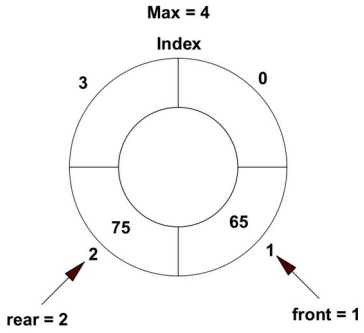
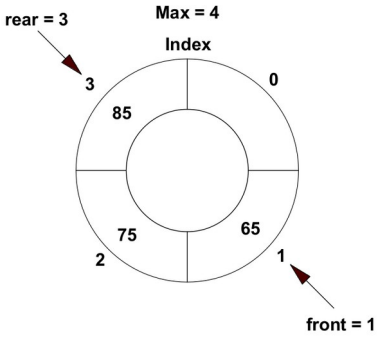
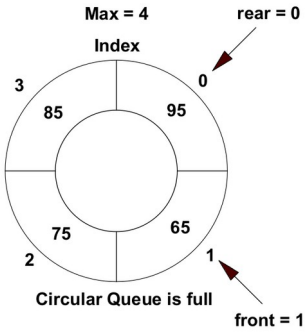
1.	
2.	
3.	
4.	

Fig.5.10: Sequence of insertion for condition 2

The above mentioned three conditions for empty circular queue and two conditions for full circular queue are summarized in the following table.

Conditions for empty circular queue	Conditions for full circular queue
Condition 1 → If rear = -1 Circular queue is empty. Condition 2 → If front = 0 and rear = Max - 1 Circular queue is empty. Condition 3 → If front = rear + 1 Circular queue is empty.	Condition 1 → If front = 0 and rear = Max - 1 Circular queue is full. Condition 2 → If front = rear + 1 Circular queue is full.

It is being seen from the above table that condition 2 for empty queue clashes with condition 1 for full queue, similarly condition 3 for empty queue is same with condition 2 for full queue. Therefore using these conditions we can not differentiate empty circular queue and full circular queue. So we have to do something so that we can get different conditions for empty queue and full queue. To resolve this problem the empty condition of a circular queue is made condition 1 only i.e. (rear = -1). But the question is "How is this done ? ".

Before making the circular queue empty totally, a situation must occur when the queue should have the last element to be deleted. In this position we have front = rear which implies that the queue is going to be empty next. Hence if the condition "front = rear" is satisfied, the values of front and rear are made 0 and -1 respectively to restore the circular queue in its initial empty state. The above mentioned condition checking is given in C code below.

```
If (front == rear)
{
        front = 0;
        rear = -1;
}
```

The above codes are incorporated inside the delete user-defined function of the circular queue. Therefore it is being observed that the conditions for empty circular queue have been reduced to a single condition. Now the condition "rear = -1" for empty queue also satisfies the condition of full circular queue "front = rear + 1" as we know when rear is -1, front will be 0 obviously. To resolve this issue the second condition of full circular queue is modified as follows.

Condition 2 for full circular queue: If rear ≠ -1 and front = rear + 1, then circular queue is full.

Finally all the conditions of a circular queue of being empty and full are written in the following table for clarification.

Final Conditions for empty circular queue	***Final Conditions for full circular queue***
Condition 1→If rear = -1 *Circular queue is empty.*	*Condition 1→If front = 0 and rear = Max - 1* *Circular queue is full.* *Condition 2→If rear ≠ -1 and front = rear + 1* *Circular queue is full.*

Algorithm of isFull operation:

Step 1: Start

Step 2: If (front = 0 and rear = Max – 1) or (rear ≠ -1 and front = rear + 1), then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isFull operation

```
int isFull()
{
        if((front == 0 && rear == Max -1) || (rear != -1 && front == rear + 1))
                return 1;
        else
                return 0;
}
```

Algorithm of isEmpty operation:

Step 1: Start

Step 2: If rear = -1, then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isEmpty operation

```
int isEmpty()
{
        if(rear == -1)
                return 1;
        else
                return 0;
}
```

1. Insert – In this operation an element is inserted or added to the circular queue at the rear end. Initially when the queue is empty, the value of the rear is made -1 and front is 0. As one element is inserted into the queue the value of rear is incremented by 1 and it becomes 0. If we insert another element, the value of the rear is again incremented by 1. Thus the value of the variable 'rear' is incremented by 1 each time as soon as a new element is added at the rear end of the circular queue, but the other variable 'front' is made unchanged during the insertion operation. That means, the insertion process in case of circular queue is almost similar to the insertion process of normal queue except a particular situation. This situation happens when rear reaches to the value (Max – 1). If another element is attempted to insert into the circular queue, the rear is made 0 after (Max – 1). The algorithm of the insertion operation into a circular queue is given below.

Algorithm of insert operation:

Step 1: Start

Step 2: If the circular queue is full, then
a) Print 'CIRCULAR QUEUE OVERFLOW'
b) Go to Step 5
[End of If]

Step 3: If rear = Max – 1, then
a) Set rear = 0
otherwise
b) Set rear = rear + 1
[End of If]

Step 4: Set Circular_Queue[rear] = Element to be inserted

Step 5: Stop

User defined function in C for insert operation

```
void insert(int element)
{
        if(isFull())
        {
                printf("CIRCULAR QUEUE OVEFLOW\n\n");
                return;
        }

        if(rear == Max - 1)
                rear = 0;
        else
                rear++;

        cqueue[rear] = element;
        printf("The element has been inserted successfully into the circular queue\n\n");
}
```

2. Delete – The delete operation removes the front or first element from the circular queue. As soon as the element is removed from the front end of the circular queue, the variable 'front' is changed in the following three ways.

1) When the circular queue has only one element left i.e. when front is equal to rear, set front to zero and rear to -1 for restoring the initial status of the circular queue. This is done to satisfy the condition of empty state of the circular queue as already mentioned in the previous section.

2) When front reaches to the last index (Max – 1), make front to be zero once again as it is a circular queue.

3) If the above mentioned two situations have not arisen, the front is incremented by one to point the next index of the circular queue.

Before modifying the variable front, the current front value of the circular queue is stored in a variable temporarily and this value is returned at the later stage.

Algorithm of delete operation:

Step 1: Start

Step 2: If the circular queue is empty, then
a) Print 'CIRCULAR QUEUE UNDERFLOW'
b) Go to Step 6
[End of If]

Step 3: Set Data = Circular_Queue[front]

Step 4: If front = rear, then
a) Set front = 0 and Set rear = -1
otherwise
b) If front = Max – 1, then
i) Set front = 0
otherwise
ii) Set front = front + 1
[End of If]

Step 5: Return Data

Step 6: Stop

User defined function in C for delete operation

```
int delete()
{
        int data;
        if(isEmpty())
        {
                printf("CIRCULAR QUEUE UNDERFLOW\n\n");
                exit(1);
        }

        data = cqueue[front];


        if(front == rear)
        {
                front = 0;
                rear = -1;
        }
        else if(front == Max - 1)
                front = 0;
        else
                front++;

        return data;
}
```

3. Peek – Peek operation gives the value of the first element of the circular queue without removing it from the queue. That means, the peek operation return the first element of the circular queue if the queue is not empty. If the queue is empty, it display an error message "QUEUE IS EMPTY". Therefore the front variable will not be changed for peek operation.

Algorithm of peek operation:

Step 1: Start

Step 2: If the circular queue is Empty, then
a) Print 'CIRCULAR QUEUE UNDERFLOW'.
b) Go to Step 4.
[End of If]

Step 3: Return Circular_Queue[front]

Step 4: Stop

User defined function in C to implement peek operation for a circular queue

```
int peek()
{
        if(isEmpty())
        {
                printf("Circular Queue is empty\n\n");
                exit(1);
        }

        return cqueue[front];
}
```

4. Display – Display function prints all the elements of a circular queue. Before printing the elements it checks the queue is empty or not. If the queue is empty, it prints an error message, otherwise it prints all the elements of the queue i.e. from first element (queue[front]) to the last element (queue[rear]). To do this a loop should be executed from i = front to i = rear. The algorithm of display function is given below.

Algorithm of display function:

Step 1: Start

Step 2: If the circular queue is Empty, then
a) Print 'CIRCULAR QUEUE UNDERFLOW'.
b) Go to Step 4.
[End of If]

Step 3: If front ≤ rear, then
a) Set i = front
i) Repeat ii) to iii) while i ≤ rear
ii) Print Circular_Queue[i]
iii) Set i = i + 1
[End of Loop]

otherwise
b) Set i = front
i) Repeat ii) to iii) while i < Max
ii) Print Circular_Queue[i]
iii) Set i = i + 1
[End of Loop]

c) Set i = 0
i) Repeat ii) to iii) while i ≤ rear
ii) Print Circular_Queue[i]
iii) Set i = i + 1
[End of Loop]
[End of If-Else at Step 3]

Step 4: Stop

User defined function in C to display all the elements of a circular queue

```
void display()
{
        int i;

        if(isEmpty())
        {
                printf("Circular Queue is empty");
        }
        else if(front <= rear)
        {
        for(i=front; i<=rear; i++)
                printf("%d  ", cqueue[i]);
        }
        else
        {
                for(i=front; i<Max; i++)
                        printf("%d  ", cqueue[i]);

                for(i=0; i<=rear; i++)
                        printf("%d  ", cqueue[i]);
        }
}
```

Now the entire C program to incorporate all the above mentioned functions like insert, delete, isFull, isEmpty, peek, display are given below.

C program to insert, delete, peek the first element and display all the elements of a circular queue.

```
#include<stdio.h>
#include<stdlib.h>
#define Max 10

int cqueue[Max];
int front = 0;
int rear = -1;
int UnderflowFlag = 0;

void insert(int );
int isFull();
int delete();
int isEmpty();
int peek();
void display();

int main()
{
        int choice, element, value, firstelem;


        while(1)
        {

printf("*****************************************************************\n");
                printf("0: Terminate the program\n");
                printf("1: Insert an element into the circular queue\n");
                printf("2: Delete an element from the circular queue\n");
                printf("3: Display the first element of the circular queue\n");
                printf("4: Display all the elements of the circular queue\n");


printf("*****************************************************************\n\n");
                printf("Enter your option: ");
                scanf("%d",&choice);
                printf("\n");

                switch(choice)
                {
                        case 0: exit(1);

                        case 1: printf("Enter the element to be inserted into the circular queue: ");
                                scanf("%d", &element);
                                insert(element);
                                break;
```

```
                        case 2: value = delete();
                                if(!UnderflowFlag)
                                printf("The element %d has been deleted succeessfully.\n\n", value);
                                break;

                        case 3: firstelem = peek();
                                if(!UnderflowFlag)
                                printf("The first element of the circular queue is %d.\n\n",
                                        firstelem);
                                break;

                        case 4: printf("The elements of the circular queue:\n\n");
                                display();
                                break;

                        default: printf("Wrong option is selected\n\n");
                                 break;
                }
        }
        return 0;
}

void insert(int element)
{
        if(isFull())
        {
                printf("CIRCULAR QUEUE OVEFLOW\n\n");
                return;
        }
        if(rear == Max - 1)
                rear = 0;
        else
                rear++;

        cqueue[rear] = element;
        printf("The element has been inserted successfully into the circular queue\n\n");
}

int isFull()
{
        if((front == 0 && rear == Max -1) || (rear != -1 && front == rear + 1))
                return 1;
        else
                return 0;
}
```

```
int delete()
{
	int data;
	if(isEmpty())
	{
		printf("CIRCULAR QUEUE UNDERFLOW\n\n");
		UnderflowFlag = 1;
		return 0;
	}
	UnderflowFlag = 0;
	data = cqueue[front];

	if(front == rear)
	{
		front = 0;
		rear = -1;
	}
	else if(front == Max - 1)
		front = 0;
	else
		front++;

	return data;
}

int isEmpty()
{
	if(rear == -1)
		return 1;
	else
		return 0;
}

int peek()
{
	if(isEmpty())
	{
		printf("Circular Queue is empty\n\n");
		UnderflowFlag = 1;
		return 0;
	}

	UnderflowFlag = 0;
	return cqueue[front];
}
```

```
void display()
{
        int i;

        if(isEmpty())
        {
                printf("Circular Queue is empty");
        }
        else if(front <= rear)
        {
        for(i=front; i<=rear; i++)
                printf("%d  ", cqueue[i]);
        }
        else
        {
                for(i=front; i<Max; i++)
                        printf("%d  ", cqueue[i]);

                for(i=0; i<=rear; i++)
                        printf("%d  ", cqueue[i]);
        }

        printf("\n\n");
}
```

3) Double Ended Queue or Deque – Double ended queue or Deque is a special type of queue in which insertion and deletion can be performed at either end of the queue. That means we can insert and delete an element at the front end as well as at the rear end of the queue. Therefore there are basically four operations that can be performed in deque – 1. insert front, 2. insert rear, 3. delete front and 4. delete rear which will be discussed in details in the next section. A deque is basically implemented using circular queue. Therefore it is essential to understand the circular queue before the discussion of deque.

1. Insert front – In this operation an element is inserted or added at the front of the deque. Initially when the queue is empty, the value of the rear is made -1 and front is 0. As the insertion is done at the front of the deque, the variable front will be changed only except during the insertion of the first element into the empty deque. When the first element is inserted at the front of the empty deque, the value of rear is incremented by 1 and it becomes 0 keeping front to be 0. If the queue is not empty, two situations may arise during the insertion at the front end of the queue.

In one situation when an element is going to be inserted at the front of the non-empty deque with front = 0 and rear ≠ Max - 1, the front will be made (Max – 1) keeping rear unchanged to utilize the free space at the end the queue. As a consequence the new element will be placed at the last index (Max – 1) of the queue.

In the other situation when an element is going to be inserted at the front of the non-empty deque with front ≠ 0, the front will be decremented by 1 keeping the variable rear unaltered. As a result the new element will be added at the index just before the index of the front. The algorithm of the insert operation into a double ended queue is given below.

Algorithm of insert front operation:

Step 1: Start

Step 2: If the deque is full, then
a) Print 'DEQUE OVERFLOW'
b) Go to Step 5
[End of If]

Step 3: If the deque is empty, then rear = Max – 1, then
a) Set rear = 0
otherwise
b) If front = 0, then
1. Set front = Max – 1
otherwise
2. Set front = front – 1
[End of If (b)]
[End of If (Step 3)]

Step 4: Set Deque[front] = Element to be inserted

Step 5: Stop

User defined function in C for insert operation at the front end

```
void insertfront(int element)
{
        if(isFull())
        {
                printf("DEQUE OVEFLOW\n\n");
                return;
        }
        else if(isEmpty())
                rear = 0;
        else if(front == 0)
                front = Max - 1;
        else
                front--;

        deque[front] = element;
        printf("The element has been inserted successfully at the front of the Deque\n\n");
}
```

2. Insert rear – In this operation an element is inserted or added at the rear end of the deque. Initially when the queue is empty, the value of the rear is made -1 and front is 0. As the insertion is done at the rear of the deque, the variable rear will be changed only except during the insertion of the first element into the empty deque. When the first element is inserted at the rear of the empty deque, the value of rear is incremented by 1 and it becomes 0 keeping front to be 0. If the queue is not empty, two situations may arise during the insertion at the rear end.

In one situation when an element is going to be inserted at the rear of the non-empty deque with front ≠ 0 and rear = Max - 1, the rear will be made 0 keeping the front unchanged to utilize the free space at the starting the queue. As a consequence the new element will be placed at the index 0 of the deque.

In the other situation when an element is going to be inserted at the rear of the non-empty deque with rear ≠ Max – 1, the rear will be incremented by 1 keeping the variable front unaltered. As a result the new element will be added at the index just after the index of the rear. The algorithm of the insert operation into a double ended queue is given below.

Algorithm of insert rear operation:

Step 1: Start

Step 2: If the deque is full, then
a) Print 'DEQUE OVERFLOW'
b) Go to Step 5
[End of If]

Step 3: If rear = Max – 1, then
a) Set rear = 0
otherwise
b) Set rear = rear + 1
[End of If (Step 3)]

Step 4: Set Deque[rear] = Element to be inserted

Step 5: Stop

User defined function in C for insert operation at the rear end

```
void insertrear(int element)
{
        if(isFull())
        {
                printf("DEQUE OVEFLOW\n\n");
                return;
        }
        else if(rear == Max - 1)
                rear = 0;
```

```
        else
                rear++;

        deque[rear] = element;
        printf("The element has been inserted successfully at the rear of the Deque\n\n");
}
```

3. Delete front – In this operation the element at the front end of a deque will be removed. Here three cases may occur during the deletion from the deque.

Case 1: If the queue has one element left i.e. front = rear before the deletion, it will be empty after the deletion of the last element. To maintain the condition of empty deque the front and the rear are assigned with 0 and -1 respectively.

Case 2: If the front points to the last index (Max – 1) of the array, the front will be made 0 keeping the value of rear unchanged.

Case 3: If the above mentioned two cases are not satisfied i.e. front ≠ rear and front ≠ Max – 1, the front will be incremented by one to point the next index.

The algorithm of the delete operation from a double ended queue is given below.

Algorithm of delete front operation:

Step 1: Start

Step 2: If the deque is empty, then
a) Print 'DEQUE UNDERFLOW'
b) Go to Step 6
[End of If]

Step 3: Set Data = Deque[front]

Step 4: If the deque has one element i.e. front = rear, then
a) Set front = 0 and Set rear = -1
otherwise
b) If front = Max – 1, then
1. Set front = 0
otherwise
2. Set front = front + 1
[End of If (b)]
[End of If (Step 4)]

Step 5: Return Data

Step 6: Stop

User defined function in C for delete operation at the front end

```
int deletefront()
{
        int data;
        if(isEmpty())
        {
                printf("DEQUE UNDERFLOW\n\n");
                UnderflowFlag = 1;
                return 0;
        }
        UnderflowFlag = 0;
        data = deque[front];

        if(front == rear)
        {
                front = 0;
                rear = -1;
        }
        else if(front == Max - 1)
                front = 0;
        else
                front++;
        return data;
}
```

4. Delete rear – In this case the element at the rear end of a deque will be removed. Here three cases may occur during the deletion from the deque like the deletion of element at the front end.

Case 1: If the queue has one element left i.e. front = rear before the deletion, it will be empty after the deletion of the that element. To maintain the condition of empty deque the front and the rear are assigned with 0 and -1 respectively.

Case 2: If the rear points to the first index of the array i.e. rear = 0, the rear will be made (Max - 1) keeping the value of front unchanged.

Case 3: If the above mentioned two cases are not fulfilled i.e. front ≠ rear and rear ≠ 0, the rear will be decremented by one to point the previous index.

The algorithm of the delete operation from a double ended queue is given below.

Algorithm of delete rear operation:

Step 1: Start

Step 2: If the deque is empty, then
a) Print 'DEQUE UNDERFLOW'
b) Go to Step 6
[End of If]

Step 3: Set Data = Deque[rear]

Step 4: If the deque has one element i.e. front = rear, then
a) Set front = 0 and Set rear = -1
otherwise
b) If rear = 0, then
1. Set rear = Max – 1
otherwise
2. Set rear = rear – 1
[End of If (b)]
[End of If (Step 4)]

Step 5: Return Data

Step 6: Stop

User defined function in C for delete operation at the rear end

```
int deleterear()
{
        int data;
        if(isEmpty())
        {
                printf("DEQUE UNDERFLOW\n\n");
                UnderflowFlag = 1;
                return 0;
        }

        UnderflowFlag = 0;
        data = deque[rear];

        if(front == rear)
        {
                front = 0;
                rear = -1;
        }
        else if(rear == 0)
                rear = Max - 1;
        else
                rear--;

        return data;
}
```

5. isFull and isEmpty – The protocols of these functions are identical to the isFull and isEmpty operation of a circular queue. That's why there is no need to explain them.

Algorithm of isFull operation:

Step 1: Start

Step 2: If (front = 0 and rear = Max – 1) or (rear ≠ -1 and front = rear + 1), then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isFull operation

```
int isFull()
{
        if((front == 0 && rear == Max -1) || (rear != -1 && front == rear + 1))
                return 1;
        else
                return 0;
}
```

Algorithm of isEmpty operation:

Step 1: Start

Step 2: If rear = -1, then
a) Return 1 (True)
otherwise
b) Return 0 (False)
[End of If]

Step 3: Stop

User defined function in C for isEmpty operation

```
int isEmpty()
{
        if(rear == -1)
                return 1;
        else
                return 0;
}
```

6. Peek – The algorithm of peek operation is same as the algorithm of peek function in case of a circular queue.

Algorithm of peek operation:

Step 1: Start

Step 2: If the deque is Empty, then
a) Print 'DEQUE UNDERFLOW'.
b) Go to Step 4.
[End of If]

Step 3: Return Deque[front]

Step 4: Stop

User defined function in C to implement peek operation for a deque

```
int peek()
{
        if(isEmpty())
        {
                printf("Deque is empty\n\n");
                UnderflowFlag = 1;
                return 0;
        }

        UnderflowFlag = 0;
        return deque[front];
}
```

7. Display – This user defined function prints all the elements in a deque. The algorithm of display operation is same as the algorithm of display operation in case of a circular queue. Therefore the algorithm and C code in this case are self-explanatory.

Algorithm of display function:

Step 1: Start

Step 2: If the Deque is Empty, then
a) Print 'DEQUE UNDERFLOW'.
b) Go to Step 4.
[End of If]

Step 3: If front ≤ rear, then
a) Set i = front
i) Repeat ii) to iii) while i ≤ rear
ii) Print Deque[i]

iii) Set i = i + 1
[End of Loop]
otherwise
b) Set i = front
i) Repeat ii) to iii) while i < Max
ii) Print Deque[i]
iii) Set i = i + 1
[End of Loop]

c) Set i = 0
i) Repeat ii) to iii) while i ≤ rear
ii) Print Deque[i]
iii) Set i = i + 1
[End of Loop]
[End of If-Else at Step 3]

Step 4: Stop

User defined function in C to display all the elements of a deque

```
void display()
{
	int i;

	if(isEmpty())
	{
		printf("Deque is empty");
	}
	else if(front <= rear)
	{
	for(i=front; i<=rear; i++)
		printf("%d  ", deque[i]);
	}
	else
	{
		for(i=front; i<Max; i++)
			printf("%d  ", deque[i]);

		for(i=0; i<=rear; i++)
			printf("%d  ", deque[i]);
	}

	printf("\n\n");
}
```

The entire C program of a deque with its various operations is given below.

C program to insert front, insert rear, delete front, delete rear, peek the first element and display all the elements of a double ended queue.

```
#include<stdio.h>
#include<stdlib.h>
#define Max 5

int deque[Max];
int front = 0;
int rear = -1;
int UnderflowFlag = 0;

void insertfront(int );
void insertrear(int );
int isFull();
int deletefront();
int deleterear();
int isEmpty();
int peek();
void display();

int main()
{
        int choice, element, value, firstelem;

        while(1)
        {
        printf("*****************************************************************\n");
                printf("0: Terminate the program\n");
                printf("1: Insert an element at the front of the Deque\n");
                printf("2: Insert an element at the rear of the Deque\n");
                printf("3: Delete an element from the front of the Deque\n");
                printf("4: Delete an element from the rear of the Deque\n");
                printf("5: Display the first element of the Deque\n");
                printf("6: Display all the elements of the Deque\n");
        printf("*****************************************************************\n\n");

                printf("Enter your option: ");
                scanf("%d",&choice);
                printf("\n");

                switch(choice)
                {
                        case 0: exit(1);

                        case 1: printf("Enter the element to be inserted at the front of the Deque: ");
                                scanf("%d", &element);
                                insertfront(element);
                                break;
```

```
                        case 2: printf("Enter the element to be inserted at the rear of the Deque: ");
                                scanf("%d", &element);
                                insertrear(element);
                                break;

                        case 3: value = deletefront();
                                if(!UnderflowFlag)
                                        printf("The element %d has been deleted succeessfully from
                                                the front of the Deque.\n\n", value);
                                break;

                        case 4: value = deleterear();
                                if(!UnderflowFlag)
                                        printf("The element %d has been deleted succeessfully from
                                                the rear of the Deque.\n\n", value);
                                break;

                        case 5: firstelem = peek();
                                if(!UnderflowFlag)
                                printf("The first element of the Deque is %d.\n\n", firstelem);
                                break;

                        case 6: printf("The elements of the Deque:\n\n");
                                display();
                                break;

                        default: printf("Wrong option is selected\n\n");
                                 break;
                }

        }

        return 0;
}

void insertfront(int element)
{
        if(isFull())
        {
                printf("DEQUE OVEFLOW\n\n");
                return;
        }
        else if(isEmpty())
                rear = 0;
        else if(front == 0)
                front = Max - 1;
        else
                front--;

        deque[front] = element;
        printf("The element has been inserted successfully at the front of the Deque\n\n");
}
```

```
void insertrear(int element)
{
	if(isFull())
	{
		printf("DEQUE OVEFLOW\n\n");
		return;
	}
	else if(rear == Max - 1)
		rear = 0;
	else
		rear++;

	deque[rear] = element;
	printf("The element has been inserted successfully at the rear of the Deque\n\n");
}

int isFull()
{
	if((front == 0 && rear == Max -1) || (rear != -1 && front == rear + 1))
		return 1;
	else
		return 0;
}

int deletefront()
{
	int data;
	if(isEmpty())
	{
		printf("DEQUE UNDERFLOW\n\n");
		UnderflowFlag = 1;
		return 0;
	}

	UnderflowFlag = 0;
	data = deque[front];

	if(front == rear)
	{
		front = 0;
		rear = -1;
	}
	else if(front == Max - 1)
		front = 0;
	else
		front++;

	return data;
}
```

```
int deleterear()
{
	int data;
	if(isEmpty())
	{
		printf("DEQUE UNDERFLOW\n\n");
		UnderflowFlag = 1;
		return 0;
	}

	UnderflowFlag = 0;
	data = deque[rear];

	if(front == rear)
	{
		front = 0;
		rear = -1;
	}
	else if(rear == 0)
		rear = Max - 1;
	else
		rear--;

	return data;
}


int isEmpty()
{
	if(rear == -1)
		return 1;
	else
		return 0;
}

int peek()
{
	if(isEmpty())
	{
		printf("Deque is empty\n\n");
		UnderflowFlag = 1;
		return 0;
	}

	UnderflowFlag = 0;
	return deque[front];
}
```

```
void display()
{
	int i;

	if(isEmpty())
	{
		printf("Deque is empty");
	}
	else if(front <= rear)
	{
	for(i=front; i<=rear; i++)
		printf("%d  ", deque[i]);
	}
	else
	{
		for(i=front; i<Max; i++)
			printf("%d  ", deque[i]);

		for(i=0; i<=rear; i++)
			printf("%d  ", deque[i]);
	}

	printf("\n\n");
}
```

Types of deque – There are two types of deque – 1. Input restricted deque and 2. Output restricted deque.

1. Input restricted deque – In this deque insertion can be done only at one of the ends whereas deletion can be done at both ends of the deque.

2. Output restricted deque – In this deque deletion can be done only at one of the ends whereas insertion can be done at both ends of the deque.

Chapter 6

Linked List

In Chapter 1 it has been discussed that an array is a linear data structure in which the elements are stored in consecutive memory locations. It is also mentioned that an array has some crucial limitations. An array is declared with fixed size which restricts the number of elements that an array can store. In some situations the number of elements to be dealt with can not be known in advance. In those cases the use of an array may fail incidentally. To overcome those situations a linked list is the solution where the elements are allocated dynamically in the memory. Due to this feature a linked list is capable to deal with enormous number of elements endlessly unless the memory space is exhausted.

A linked list is a very flexible, dynamic, linear data structure in which elements (called nodes) are connected to each other in a sequential manner. Every node of a linked list has two parts – 1) Data part and 2) Link part. The data part consists of one or more number of data with different types like int, float, char etc. The link part consists the address of another node which may be the predecessor node or the successor node. Hence the link part is basically a pointer which points to another node of the linked list. Thus the connection between the two nodes is established with the help of this link part and the consecutive nodes of a linked list are connected in a sequential manner. As the nodes of a linked list is allocated dynamically inside the memory, a linked list does not store its elements in consecutive memory locations. A pictorial view of a linked list is shown in Fig.6.1 for better understanding.

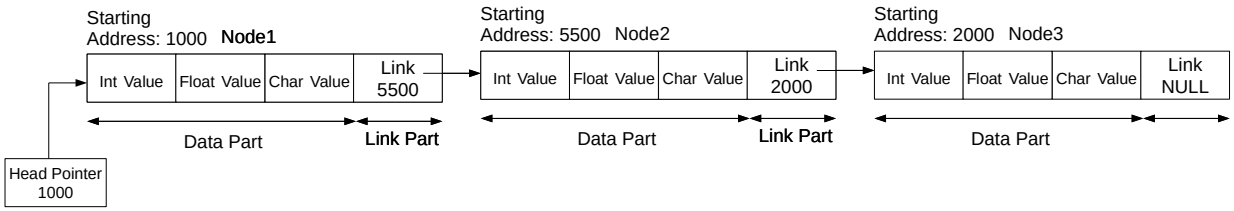


Fig.6.1: Pictorial view of a linked list

In the above figure a linked list is shown with three nodes with starting addresses 1000, 5500 and 2000 respectively. So it is evident that the nodes have occupied the memory locations randomly i.e. not consecutively. Moreover it is also being seen that the data part of each node is holding one integer value, one floating point value and one character type value. The link part of each node stores the starting address of next node to establish the connection. The link part of first node holds the starting address (5500) of the second node, the link of the second node holds the starting address (2000) of the third node and so on. The link part of the last node is not holding any address. That's why it is assigned to NULL to indicate the end of the linked list. The starting address of the first node is stored in a pointer called head pointer. The entire linked list can be accessed by only the starting pointer or head pointer.

A comparative study between array and linked list is given in the following table.

SL.	Array	Linked List
1.	Arrays are declared as fixed size. If it is required to handle more number of elements than the size of the array, it is not possible to do that. Due to this reason array is referred as static data structure.	All the nodes of a linked list are created dynamically during the execution of the program. This feature of the linked list gives the facility to the user to add as many nodes as he wants as per his requirement. Basically there is no limitation of nodes to be added in a linked list unless the memory space becomes exhausted. Due to the dynamic allocation of the nodes the linked list is referred as dynamic data structure.
2.	Data elements of an array are stored in consecutive memory locations.	Nodes of a linked list are stored randomly in the memory i.e. they are not allocated in consecutive memory locations.
3.	Insertion and deletion of an element in case of array becomes problematic because of shifting of elements from their positions. The program execution time increases due these shifting of the elements in case of an array.	Insertion and deletion of a node from a linked list is easier than array data structure. Here shifting of the elements is not required like array which reduces the execution time of a program.
4.	Array is a linear data structure.	Linked list is a linear data structure.
5.	Array allows random access to its elements using the indices. Due to this reason array is faster for accessing the elements.	Linked list does not provide random access to elements like arrays. To access an element in a linked list, we have to start at the beginning of the list and traverse the list until we find the desired element. Due to this reason linked list is slower for accessing elements.
6.	Arrays needs memory to store the elements themselves, which takes less memory space.	Linked list requires extra memory compared to array. Each element in a linked list requires a reference to the next element, which takes additional memory space.
7.	Implementation of an array is easier than a linked list because array does not require dynamic memory allocation.	Implementation of a linked list is more complex than an array because it uses dynamic memory allocation and pointers to hold the address of another node.

Types of Linked List – There are three types of linked list – 1) singly linked list, 2) doubly lined list and 3) circular linked list.

1) Singly Linked List – A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way i.e. from first node to second node, from second node to third node and so on. We can not traverse from third node to second node, from second node to first node. This kind of traversal is called forward traversal. Figure 6.2 shows a singly linked list.

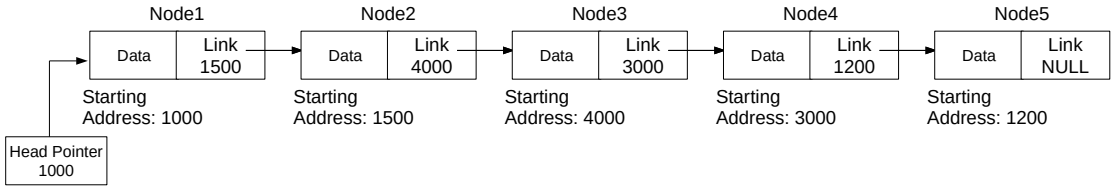


Fig.6.2: Pictorial view of a singly linked list with 5 nodes

The above singly linked list with five nodes is self-explanatory. The last node of the singly linked list has no next node connected it, so it will store NULL pointer in the link part of the last node. The NULL pointer of a singly linked list also indicates the last node. In case of a singly linked list the starting node is always pointed by a pointer called the head pointer. The head pointer is very important for a singly linked list, because without this head pointer the linked list becomes inaccessible. We can traverse the entire linked list using head pointer. If head pointer is equal to NULL, then it implies that the linked list is empty.

Declaring a singly linked list – It is being observed that every node of a singly linked list has two parts – one is data and other is link or pointer to the next node. This implies that every node of a linked list is formed with some mixed data types. Therefore node of a singly linked list may be constructed only by using structure in C language. As in a linked list, every node contains a pointer to another node of same type, it is also called a *self-referential data type*.

In C a node of a singly linked list can be implemented by the following code.

```
struct node
{
        int data;
        struct node *link;
};
```

1. Creating a singly linked list – Creation of a singly linked list means, generation of a specified number of nodes which will be connected to each other in a sequential manner. Here the number of nodes (n) will be provided by the user as per his requirement. After the execution of this operation a singly linked list with n number of nodes will be generated and the function will return the head pointer of the linked list. The algorithm to create a singly linked list with n number of nodes is given below.

Algorithm of singly linked list creation:

Step 1: Start

Step 2: Set Head_Pointer = NULL

Step 3: Input n for number of nodes to be created

Step 4: Set i = 0

Step 5: Repeat Step 6 to Step 8 while i < n

Step 6: Input Data to be stored into the node

Step 7: If i = 0 (for first node creation), then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = NULL
Set Head_Pointer = New_Node_Pointer
Set Ptr = Head_Pointer
Otherwise
b) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set ptr → link = New_Node_Pointer
Set ptr → link → link = NULL
Set ptr = ptr → link
[End of If-Else (Step 7)]

Step 8: Set i = i + 1

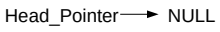
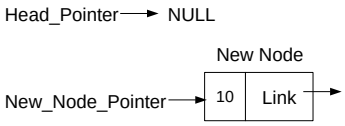
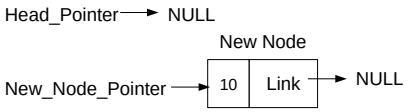
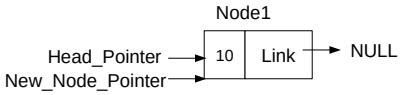
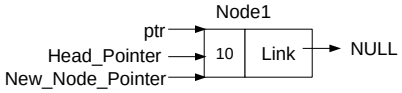
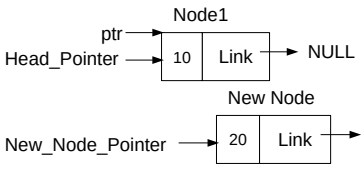
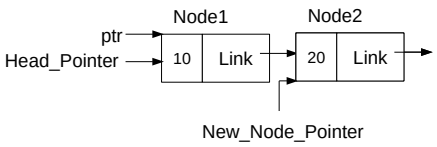
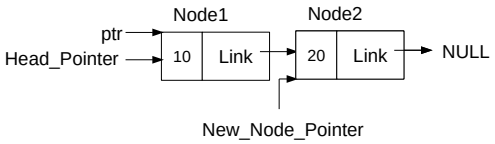
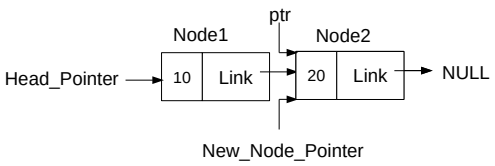
Step 9: Print "Singly Linked List has been created successfully".

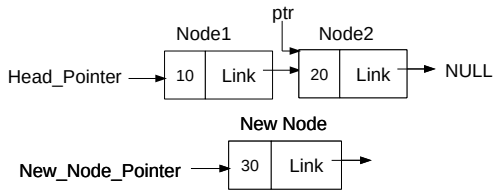
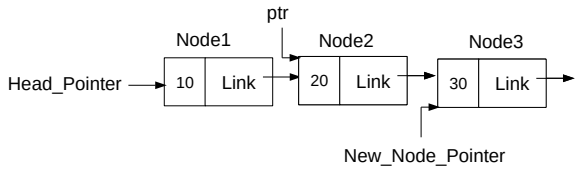
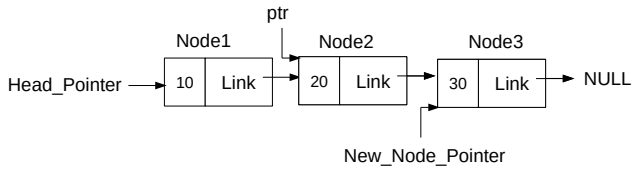
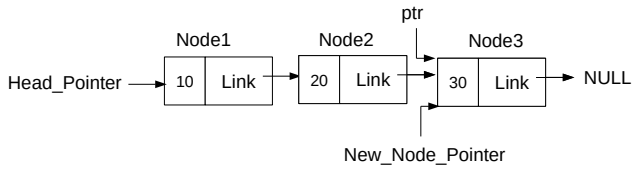
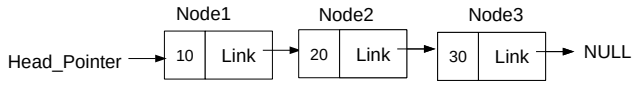
Step 10: Set ptr = NULL

Step 11: Set New_Node_Pointer = NULL

Step 12: Stop

Now the above mentioned algorithm for the creation of a singly linked list is explained with the help of the following pictorial representation. In this case we have assumed that the number of nodes (n) to be created is 3. Initially the Head_Pointer is NULL to indicate that the linked list is empty i.e. the linked list has no nodes.

Steps	Statements	Linked List after the execution of the statement
Initially	Head_Pointer = NULL	
i = 0 Data = 10	New_Node_Pointer → data = Data	
	New_Node_Pointer → link = NULL	
	Head_Pointer = New_Node_Pointer	
	Ptr = Head_Pointer	
i = 1 Data = 20	New_Node_Pointer → data = Data	
	ptr → link = New_Node_Pointer	
	ptr → link → link = NULL	
	ptr = ptr → link	

Steps	Statements	Linked List after the execution of the statement
i = 2 Data = 30	New_Node_Pointer → data = Data	
	ptr → link = New_Node_Pointer	
	ptr → link → link = NULL	
	ptr = ptr → link	
Finally	ptr = NULL New_Node_Pointer = NULL	

It is being observed from the above pictorial representation of singly linked list creation that three nodes with values 10, 20 and 30 have been created as per the algorithm. The pointers ptr and New_Node_Pointer are used temporarily. After the creation of the linked list there will be no use of these pointers. That's why they are made NULL pointers. Now we can easily understand how the algorithm of linked list creation is working step by step from the above pictorial explanation.

User defined function in C for singly linked list creation

```
struct node *CreateNode()
{
        struct node *headptr = NULL, *ptr, *newnodeptr;
        int i, n, Data;

        printf("Enter the no. of nodes to be created in the singly linked list: ");
        scanf("%d", &n);
```

```
        for(i=0; i<n; i++)
        {
                printf("Enter Data%d: ", i+1);
                scanf("%d", &Data);
                if(i == 0)
                {
                        newnodeptr = (struct node *)malloc(sizeof(struct node));
                        newnodeptr->data = Data;
                        newnodeptr->link = NULL;
                        headptr = newnodeptr;
                        ptr = headptr;
                }
                else
                {
                        newnodeptr = (struct node *)malloc(sizeof(struct node));
                        newnodeptr->data = Data;
                        ptr->link = newnodeptr;
                        ptr->link->link = NULL;
                        ptr = ptr->link;
                }
        }
        printf("Singly Linked list has been created successfully.\n\n");
        ptr = NULL;
        return headptr;
}
```

2. Displaying the data of all nodes in a singly linked list – This operation displays/ prints the data of all nodes in a singly linked list in forward direction i.e. from starting node to last node. In this case a temporary pointer is taken to point the starting node or first node initially. It is required to traverse all the nodes from first node to last node and during this traversal the data of every node is displayed on the screen. Here traversal means, pointing the nodes one after another in forward direction until the end of the linked list.

Algorithm of displaying the data of all nodes in a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Repeat Step 4 to Step 5 while ptr ≠ NULL

Step 4: Print ptr → data

Step 5: Set ptr = ptr → link

Step 6: Set ptr = NULL

Step 7: Stop

From this algorithm it is clear that the pointer ptr points to starting node of the singly linked list initially. The pointer ptr shifts its position from first node to second node, second node to third node, third node to fourth node and so on until it becomes NULL. During this traversal of all nodes the data value of every nodes are printed on the screen.

User defined function in C for displaying the data of all nodes in a singly linked list

```
void DisplayNode(struct node *headptr)
{
        struct node *ptr = headptr;

        printf("The singly linked list is given below:\n");

        while(ptr != NULL)
        {
                printf("%-5d", ptr->data);
                ptr = ptr->link;
        }

        printf("\n\n");

        ptr = NULL;
}
```

3. Counting the number of nodes in a singly linked list – This operation counts the number of nodes in a singly linked list. Here a variable 'count' is initialized to zero and incremented by one each time the pointer ptr traverses the successive nodes in the linked list.

Algorithm to count the number of nodes in a singly linked list:

Step 1: Start

Step 2: Set count = 0

Step 3: Set ptr = Head_Pointer

Step 4: Repeat Step 5 to Step 6 while ptr ≠ NULL

Step 5: Set count = count + 1

Step 6: Set ptr = ptr → link

Step 7: Print count to display the number of nodes

Step 8: Set ptr = NULL

Step 9: Stop

User defined function in C for counting the number of nodes in a singly linked list

```
void CountNodes(struct node *headptr)
{
        int count = 0;
        struct node *ptr = headptr;

        while(ptr != NULL)
        {
                count++;
                ptr = ptr->link;
        }
        printf("The no. of nodes in the linked list: %d\n", count);
        ptr = NULL;
}
```

4. Insert a node at the beginning of a singly linked list – Here a new node will be inserted at the beginning of a singly linked list. After the insertion the new node becomes the starting node or the first node of the singly linked list. Therefore the Head_Pointer must be shifted from the second node to the first node to point at the beginning of the linked list.

Algorithm to insert a node at the beginning of a singly linked list:

Step 1: Start

Step 2: Input Data to be stored into the new node

Step 3: Create a new node pointed by New_Node_Pointer

Step 4: Set New_Node_Pointer → data = Data

Step 5: Set New_Node_Pointer → link = Head_Pointer

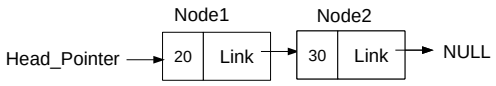
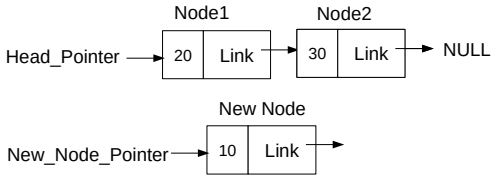
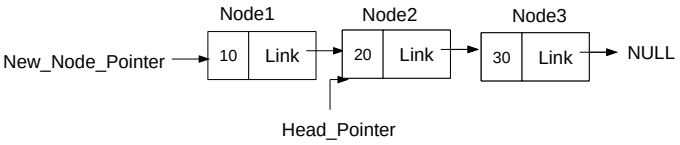
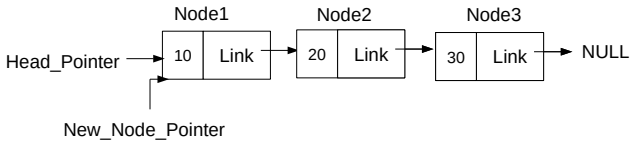
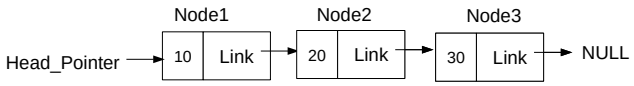
Step 6: Set Head_Pointer = New_Node_Pointer

Step 7: Print "A new node has been inserted at the starting successfully".

Step 8: New_Node_Pointer = NULL

Step 9: Stop

At first a value which is to be stored into the data part of the newly created node will be taken from the user and that value (called Data) will be stored into the new node. Now the new node will be added at the beginning of the existing linked list by the establishment of the link and Head_Pointer. The concept of the above mentioned algorithm is explained pictorially in the following table. In the following table it is assumed that the existing singly linked list has two nodes with values 20 and 30. Now the insertion of a new node with value 10 at the starting of the linked list is shown step by step.

Steps	Statements	Linked List after the execution of the statement
Initially		
Data = 10	New_Node_Pointer → data = Data	
	New_Node_Pointer → link = Head_Pointer	
	Head_Pointer = New_Node_Pointer	
Finally	New_Node_Pointer = NULL	

User defined function in C to insert a new node at the beginning of a singly linked list

```
struct node *InsertAtStart(struct node *headptr)
{
        struct node *newnodeptr;
        int Data;

        printf("Enter the data to be stored into the new node: ");
        scanf("%d", &Data);

        newnodeptr = (struct node *)malloc(sizeof(struct node));
        newnodeptr->data = Data;
        newnodeptr->link = headptr;
        headptr = newnodeptr;

        printf("A new node has been inserted at the starting successfully.\n\n");

        newnodeptr = NULL;

        return headptr;
}
```

5. Insert a node at the end of a singly linked list – This operation inserts a newly created node at the end of a singly linked list i.e. it add the new node after the last node of the existing linked list. After the insertion at the end the new node becomes the last node of the linked list and that's why the new node must be terminated with NULL link.

Algorithm to insert a node at the end of a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Data to be stored into the new node

Step 4: If ptr = NULL *(Condition for empty linked list)*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = NULL
Set Head_Pointer = New_Node_Pointer
Otherwise
b) Repeat (c) while ptr → link ≠ NULL
c) ptr = ptr → link

d) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set ptr → link = New_Node_Pointer
Set ptr → link → link = NULL
[End of If-Else (Step 4)]

Step 5: Print "A new node has been inserted at the end successfully".

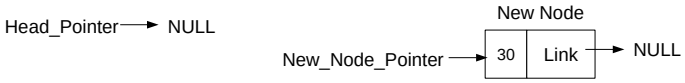
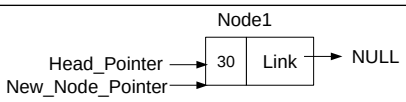
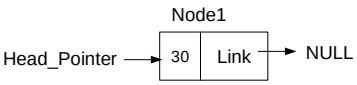
Step 6: Set ptr = NULL

Step 7: Set New_Node_Pointer = NULL

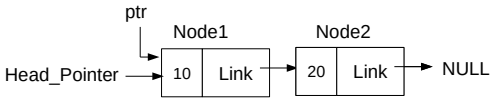
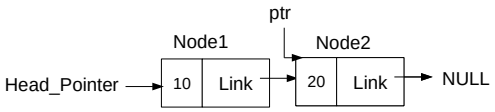
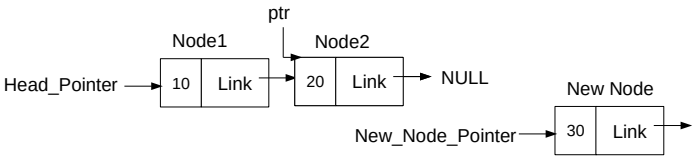
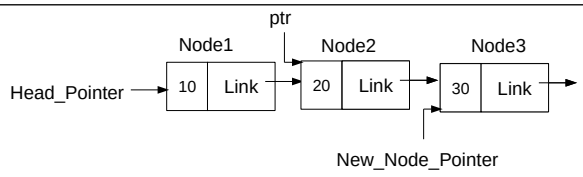
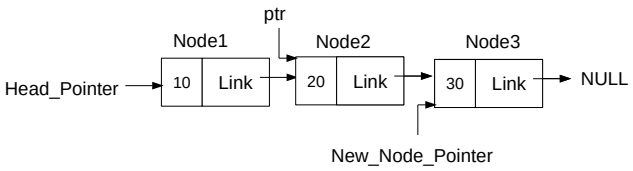
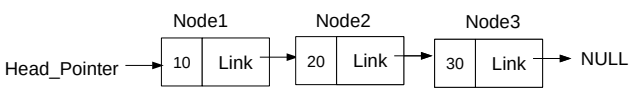
Step 8: Stop

In the above algorithm two situations during the insertion of a node at the end has been considered – one situation when the singly linked list is empty and other situation when the linked list is not empty. When the linked list is empty the Head_Pointer is shifted to point the new node after the insertion. When the linked list is not empty a temporary pointer ptr is used to point to the last node of the linked list and then the newly created node is inserted at the end of the linked list using the link part of the last node. In the following pictorial demonstration of inserting a node at the end, two situations have been considered, In the first pictorial representation a new node with value 30 is being added at the end of an empty linked list and in the second pictorial representation a new node with value 30 is being added at the end of a linked list of two nodes with values 10 and 20.

The following table gives the step by step demonstration when the linked list is empty initially.

Steps	Statements	Linked List after the execution of the statement
Initially	Head_Pointer = NULL ptr = Head_Pointer	
Data = 30	New_Node_Pointer → data = Data	
	New_Node_Pointer → link = NULL	
	Head_Pointer = New_Node_Pointer	
Finally	ptr = NULL New_Node_Pointer = NULL	

The following table gives the step by step demonstration when the linked list is not empty initially.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer	
Traversal of Nodes	ptr = ptr → link	
Data = 30	New_Node_Pointer → data = Data	
	ptr → link = New_Node_Pointer	
	ptr → link → link = NULL	
Finally	ptr = NULL New_Node_Pointer = NULL	

User defined function in C to insert a new node at the end of a singly linked list

```
struct node *InsertAtLast(struct node *headptr)
{
        struct node *ptr = headptr, *newnodeptr;
        int Data;

        printf("Enter the data to be stored into the new node: ");
        scanf("%d", &Data);

        if(ptr == NULL)
        {
                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->link = NULL;
                headptr = newnodeptr;
        }
        else
        {
                while(ptr->link != NULL)
                        ptr = ptr->link;

                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                ptr->link = newnodeptr;
                ptr->link->link = NULL;
        }

        printf("A new node has been inserted at the end successfully.\n\n");

        ptr = NULL;
        newnodeptr = NULL;

        return headptr;
}
```

6. Insert a node before a specific node in a singly linked list – This function will insert a new node just before another node which is specified by the user. Here we have to consider two situations.

In first situation the specified node is the first node in the linked list. So the insertion of a new node before the first node is basically the insertion of a new node at the beginning of the link list, which has been already discussed previously in this chapter. In second situation when the specified node is not first node, we have to insert the newly created node in between the specified node and the node just before the specified node. Except these two situations, there is another situation where the specified node is not found in the linked list. In that case an error message will be printed.

Algorithm of inserting a node before a specific node in a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node before which a new node will be inserted

Step 4: Input Data to be stored into the new node

Step 5: If ptr → data = Value *[Condition of finding Value in 1st node]*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Head_Pointer
Set Head_Pointer = New_Node_Pointer
Otherwise
b) Repeat (c) while ptr → link ≠ NULL and ptr → link → data ≠ Value
c) Set ptr = ptr → link

d) If ptr → link = NULL, then
i) Print "The node before which insertion will be done is not found".
Otherwise
ii) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = ptr → link
Set ptr → link = New_Node_Pointer
Print "A new node has been inserted before the specified node".
[End of If-Else (d)]
[End of If-Else (Step 5)]

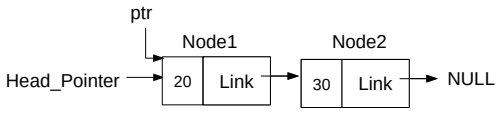
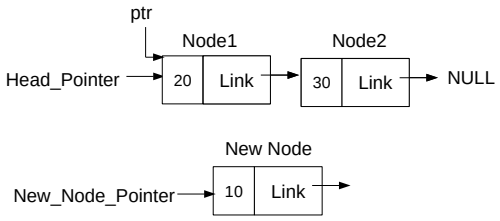
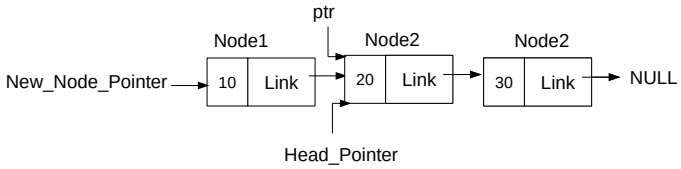
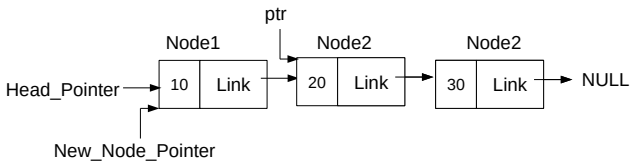
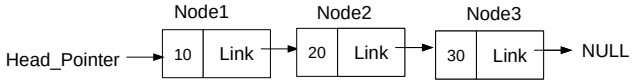
Step 6: Set ptr = NULL

Step 7: Set New_Node_Pointer = NULL

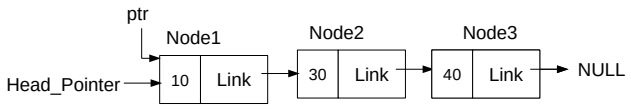
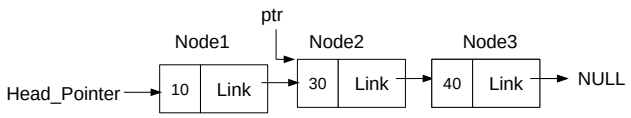
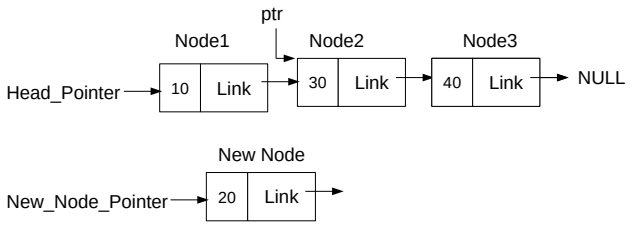
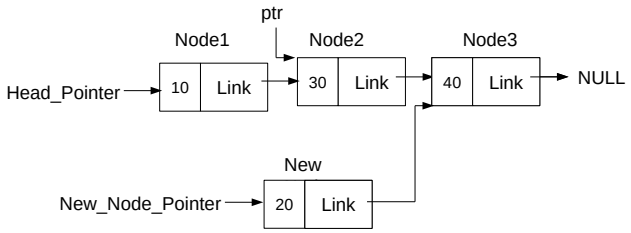
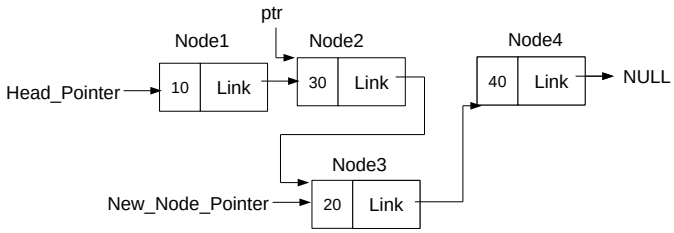
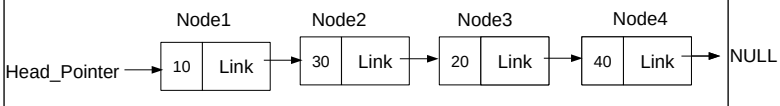
Step 8: Stop

In this algorithm we can see that there are two situations for inserting a node before another node. In one situation the specified node is the first node, which is given under the If condition in the algorithm. In this case, the algorithm of inserting a node at the beginning (explained previously) will be followed. In another situation when the specified node is not first node, the algorithm under the Otherwise/ Else condition will be executed. Under this Otherwise condition there are two cases – when the specified node is not found in the linked list and the specified node is found in any other position except the first position in the linked list. Now the two situations – 1) the specified node is the first node and 2) the specified node is any other node except the first node are illustrated in the following tabular pictorial representations.

The following table gives the step by step demonstration while the first node is the specified node before which a new node will be inserted.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer	
Value = 20 Data = 10	ptr -> data = 20 ptr → data equal to Value New_Node_Pointer → data = Data	
	New_Node_Pointer → link = Head_Pointer	
	Head_Pointer = New_Node_Pointer	
Finally	ptr = NULL New_Node_Pointer = NULL	

The following table gives the step by step demonstration while the the specified node before which a new node will be inserted is any other node except first node.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer	
Value = 40 Data = 20	ptr → link → data = 30 ptr → link → data not equal to Value ptr = ptr → link	
Value = 40 Data = 20	ptr → link → data = 40 ptr → link → data equal to Value New_Node_Pointer → data = Data	
	New_Node_Pointer → link = ptr → link	
	ptr -> link = New_Node_Pointer	
Finally	ptr = NULL New_Node_Pointer = NULL	

User defined function in C to insert a new node before a specific node in a singly linked list

```
struct node *InsertBeforeNode(struct node *headptr)
{
        struct node *ptr = headptr, *newnodeptr;
        int Data, Value;

        printf("Enter the value of a node before which a new node will be inserted: ");
        scanf("%d", &Value);
        printf("Enter the data of the new node to be inserted before the specified node: ");
        scanf("%d", &Data);

        if(ptr->data == Value)
        {
                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->link = headptr;
                headptr = newnodeptr;
        }
        else
        {
                while(ptr->link != NULL && ptr->link->data != Value)
                        ptr = ptr->link;

                if(ptr->link == NULL)
                     printf("The node before which insertion will be done is not found.\n\n");
                else
                {
                        newnodeptr = (struct node *)malloc(sizeof(struct node));
                        newnodeptr->data = Data;
                        newnodeptr->link = ptr->link;
                        ptr->link = newnodeptr;
                        printf("A new node has been inserted before the specified node
                                successfully.\n\n");
                }
        }

        newnodeptr = NULL;
        ptr = NULL;

        return headptr;
}
```

7. Insert a node after a specific node in a singly linked list – This operation will insert a new node just after another node which is specified by the user. Here at first the node with the specified value is searched through the traversal of the linked list and if the specified node is found, a new node with entered data will be inserted after it, otherwise an error message will be displayed.

Algorithm of inserting a node after a specific node in a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node after which a new node will be inserted

Step 4: Input Data to be stored into the new node

Step 5: Repeat Step 6 while ptr ≠ NULL and ptr → data ≠ Value

Step 6: Set ptr = ptr → link

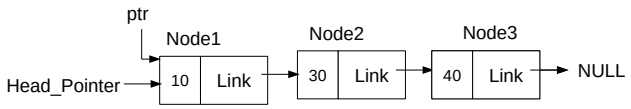
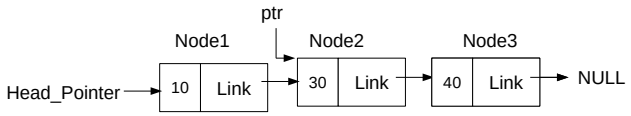
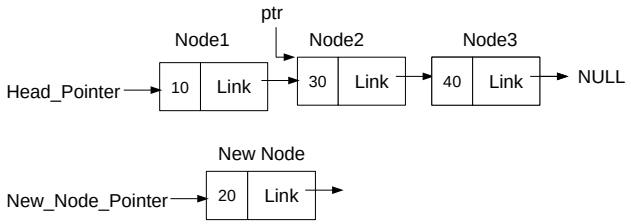
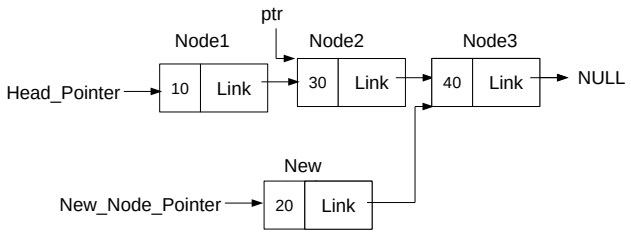
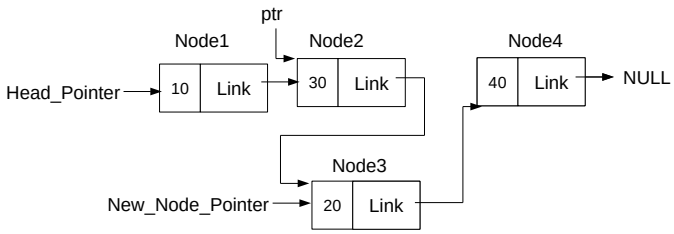
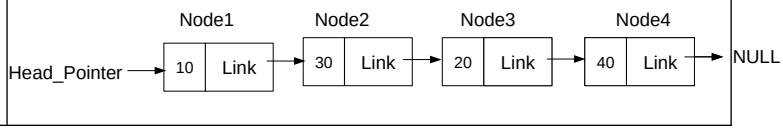
Step 7: If ptr = NULL, then
a) Print "The node after which insertion will be done is not found".
Otherwise
b) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = ptr → link
Set ptr → link = New_Node_Pointer
Print "A new node has been inserted after the specified node".
[End of If-Else (Step 7)]

Step 8: Set ptr = NULL

Step 9: Set New_Node_Pointer = NULL

Step 10: Stop

The following table gives the step by step demonstration while a new node will be inserted after the specified node at any position of the linked list.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer	
Value = 30 Data = 20	ptr → data = 10 ptr → data not equal to Value ptr = ptr → link	
Value = 30 Data = 20	ptr → data = 30 ptr → data equal to Value New_Node_Pointer → data = Data	
	New_Node_Pointer → link = ptr → link	
	ptr -> link = New_Node_Pointer	
Finally	ptr = NULL New_Node_Pointer = NULL	

User defined function in C to insert a new node after a specific node in a singly linked list

```
struct node *InsertAfterNode(struct node *headptr)
{
        struct node *ptr = headptr, *newnodeptr;
        int Data, Value;

        printf("Enter the value of a node after which a new node will be inserted: ");
        scanf("%d", &Value);
        printf("Enter the data of the new node to be inserted after the specified node: ");
        scanf("%d", &Data);

        while(ptr != NULL && ptr->data != Value)
                ptr = ptr->link;

        if(ptr == NULL)
                printf("The node after which insertion will be done is not found.\n\n");
        else
        {
                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->link = ptr->link;
                ptr->link = newnodeptr;
                printf("A new node has been inserted after the specified node.\n\n");
        }

        ptr = NULL;
        newnodeptr = NULL;

        return headptr;
}
```

8. Insert a node at a specific position of a singly linked list – This function gives the opportunity to insert a new node with an entered data at a specified position of a singly linked list. In this case the remaining part of the linked list starting from the specific position will be shifted right and the new node will be placed at the specified position. For example – if the specified position is 3rd in a singly linked list of three nodes, then the new node will be placed at 3rd position and the 3rd node of the existing linked list will be shifted right one position to become 4th node.

Algorithm of inserting a node at a specific position of a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Position where a new node will be inserted

Step 4: Input Data to be stored into the new node

Step 5: Set NoOfNodes = Total number of nodes in the linked list

Step 6: If Position < 1 or Position > NoOfNodes + 1, then
a) Print "Entered position is Invalid".
Otherwise
b) If Position = 1 *(Condition of 1st node selection)*, then
i) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Head_Pointer
Set Head_Pointer = New_Node_Pointer
Otherwise
i) Set i = 1
ii) Repeat iv) and v) while i < Position – 1
iii) Set ptr = ptr → link
iv) Set i = i + 1

v) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = ptr → link
Set ptr → link = New_Node_Pointer
Print "A new node has been inserted at a specific position".
[End of If-Else (b)]
[End of If-Else (Step 6)]

Step 7: Set ptr = NULL

Step 8: Set New_Node_Pointer = NULL

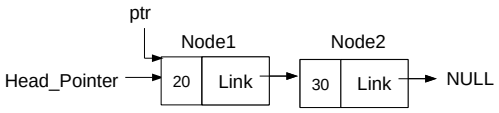
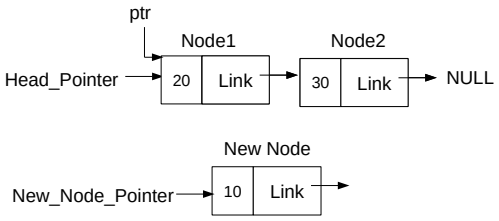
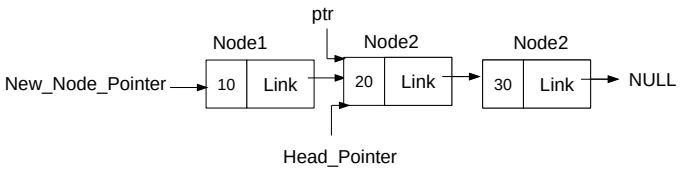
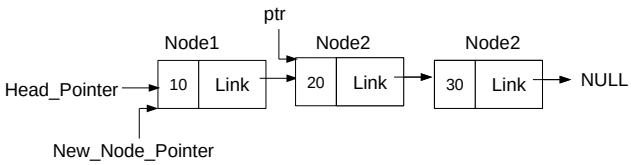
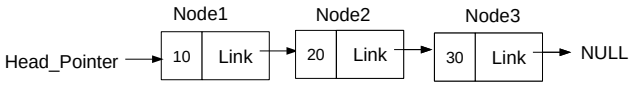
Step 9: Stop

In this algorithm we have assumed two cases.

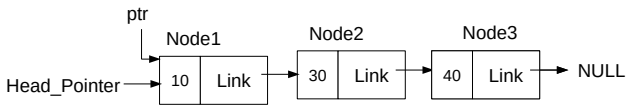
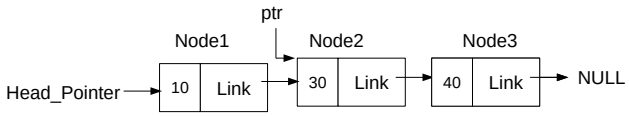
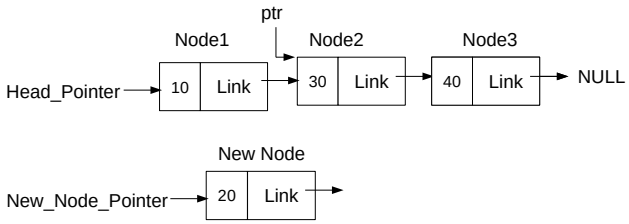
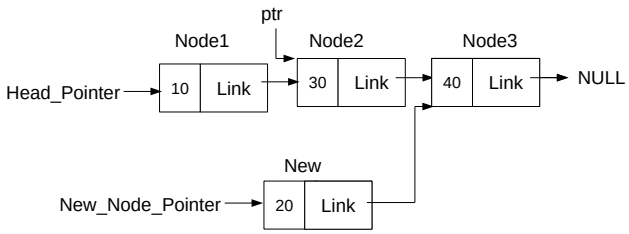
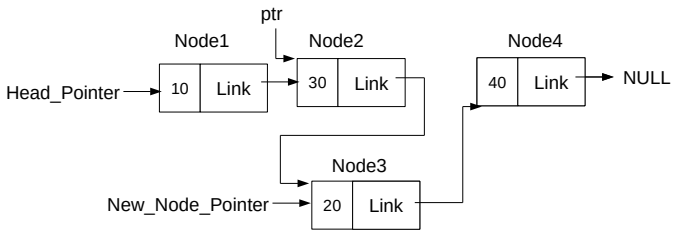
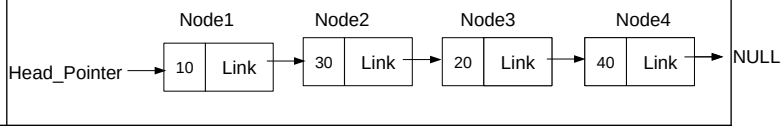
1) When 1st position is selected, the new node will be inserted at the beginning of the linked list. Therefore we have followed the algorithm of inserting a node at the starting of the linked list.

2) When any other position except 1st position is selected, then after traversal of nodes a temporary pointer ptr will point to the predecessor node of the node at selected position. Now a new node will be inserted in between the predecessor node and the node at the specific position. The two cases have been demonstrated pictorially in the following two tables.

The following table gives the step by step demonstration while a new node will be inserted at the first position of the linked list.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer	
Position = 1 Data = 10	New_Node_Pointer → data = Data	
	New_Node_Pointer → link = Head_Pointer	
	Head_Pointer = New_Node_Pointer	
Finally	ptr = NULL New_Node_Pointer = NULL	

The following table gives the step by step demonstration while a new node will be inserted at the specific position of the linked list.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer	
Position = 3 Data = 20	i = 1 i < Position – 1 ptr = ptr → link	
Position = 3 Data = 20	i = 2 i < Position – 1 not satisfied New_Node_Pointer → data = Data	
	New_Node_Pointer → link = ptr → link	
	ptr → link = New_Node_Pointer	
Finally	ptr = NULL New_Node_Pointer = NULL	

User defined function in C to insert a new node at a specific position of a singly linked list

```
struct node *InsertAtPos(struct node *headptr)
{
        struct node *ptr = headptr, *newnodeptr;
        int i, NoOfNodes;

        NoOfNodes = CountNodes(headptr);

        printf(“Enter the position where a new node will be inserted: “);
        scanf(“%d”, &Position);
        printf(“Enter the data of the new node to be inserted: “);
        scanf(“%d”, &Data);

        if(Position < 1 || Position > NoOfNodes + 1)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->link = headptr;
                headptr = newnodeptr;
        }
        else
        {
                for(i=1; i<Position - 1; i++)
                        ptr = ptr->link;

                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->link = ptr->link;
                ptr->link = newnodeptr;
                printf("A new node has been inserted at a specific position.\n\n");
        }

        newnodeptr = NULL;
        ptr = NULL;

        return headptr;
}
```

9. Delete a node at the beginning of a singly linked list – This operation deletes a node at the beginning of a singly linked list. Basically it removes the first node from the linked list. As a result of this, the second node becomes the first node of the list. Therefore the Head_Pointer should be shifted from the first node to the second node and then the first node is removed from the linked list.

Algorithm to delete a node at the beginning of a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty"
Otherwise
b) Set Head_Pointer = ptr → link
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
[End of If-Else (Step 3)]

Step 4: Set ptr = NULL

Step 5: Stop

The algorithm is very short. Therefore pictorial representation in this case is not required. How the algorithm is working is thus completely self-explanatory.

User defined function in C to delete a node at the beginning of a singly linked list

```
struct node *DeleteAtStart(struct node *headptr)
{
        struct node *ptr = headptr;

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else
        {
                headptr = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return headptr;
}
```

10. Delete a node at the end of a singly linked list – In this operation the node at the end of a singly linked list will be removed. Here two situations have been considered. In one situation, in case of a linked list with a single node, the one and only node will be pointed by ptr, the Head_Pointer is made NULL and finally the node is removed. In another situation, the ptr points to the last node and the prevvptr points to the node just before the last node, prevptr → link is made NULL and finally the last node pointed by the pointer ptr will be deleted from the linked list.

Algorithm to delete a node at the end of a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Set prevptr = Head_Pointer

Step 4: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty"
Otherwise
b) Repeat c) and d) while ptr → link ≠ NULL
c) Set prevptr = ptr
d) Set ptr = ptr → link

e) Set prevptr → link = NULL
f) If ptr = Head_Pointer *(Condition for single node in linked list)*, then
i) Set Head_Pointer = NULL
[End of If (f)]
g) Remove the node pointed by ptr
h) Print "The node has been deleted at the end of the linked list"
[End of If-Else (Step 4)]

Step 5: Set ptr = NULL

Step 6: Set prevptr = NULL

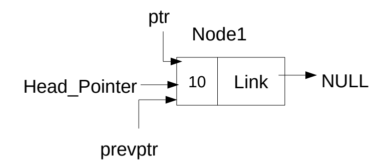
Step 7: Stop

In this algorithm two situations have been considered.

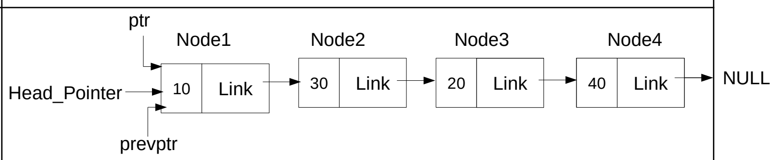
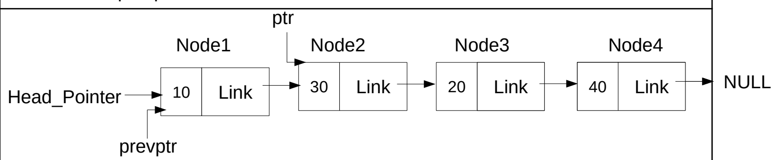
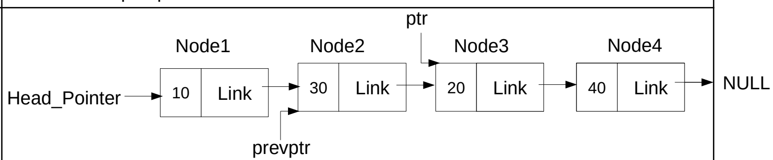
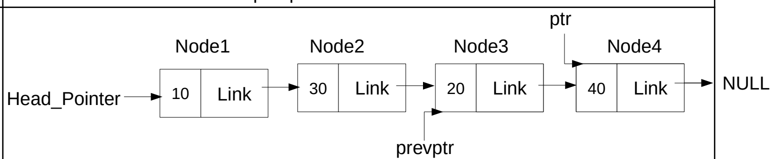
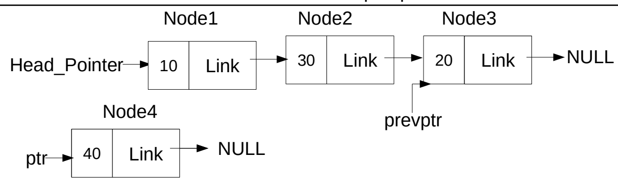
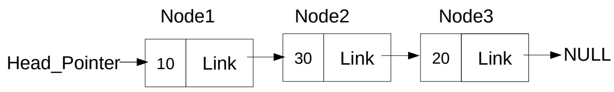
1) When the linked list contains a single node, the node will be pointed by ptr first, then the Head_Pointer is made NULL and finally the starting node will be removed.

2) When the linked list has multiple nodes, the last node will be pointed by a pointer ptr and the node before the last node will be pointed by another pointer prevptr after traversing the nodes one by one starting from 1st node. Finally prevptr → link is made NULL and the last node pointed by ptr is removed from the linked list.

The following table gives step by step demonstration to delete the last node when the linked list consists of only one node.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer prevptr = Head_Pointer	
	Head_Pointer = NULL	
Finally	Remove the node pointed by ptr [free(ptr)] ptr = NULL prevptr = NULL	

The following table gives step by step demonstration to delete the last node when the linked list consists of multiple number of nodes.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer prevptr = Head_Pointer	
Traversal of Nodes	prevptr = ptr ptr = ptr → link	
	prevptr = ptr ptr = ptr → link	
	prevptr = ptr ptr = ptr → link	
	prevptr → link = NULL	
Finally	Remove the node pointed by ptr [free(ptr)] ptr = NULL prevptr = NULL	

User defined function in C to delete a node at the end of a singly linked list

```
struct node *DeleteAtLast(struct node *headptr)
{
        struct node *ptr = headptr, *prevptr = headptr;

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else
        {
                while(ptr->link != NULL)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }

                prevptr->link = NULL;
                if(ptr == headptr)
                        headptr = NULL;
                free(ptr);
                printf("The node has been deleted at the end successfully.\n\n");

        }

        prevptr = NULL;
        ptr = NULL;
        return headptr;
}
```

11. Delete a node before a specific node in a singly linked list – Here three error cases and two cases of deletion will be considered.
The three error cases are:
1) The first error case occurs when the linked list is empty.
2) The second error case occurs when the first node is selected. Obviously there will be no node for deletion before the first node.
3) The third error case occurs when a value given by a user is not found in any node of the linked list.

The two cases to delete a node before a specified node are given below.
1) If the value of the second node is given by a user, then obviously the first node of the linked list will be deleted. Here the algorithm of deletion of a node at the beginning of the linked list will be followed.
2) If the value of a node except the second node is selected, then the node just before the specified node will be removed using another procedure.

Algorithm to delete a node before the specified node of a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Set prevptr = Head_Pointer

Step 4: Input Value of a node before which a node will be deleted

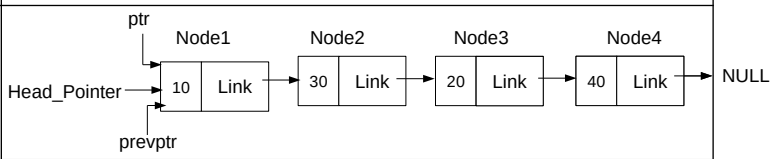
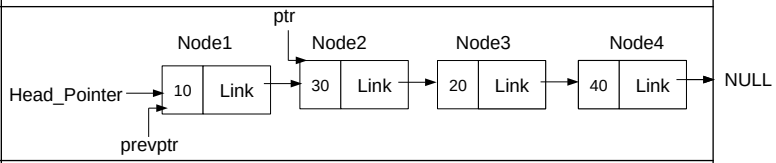
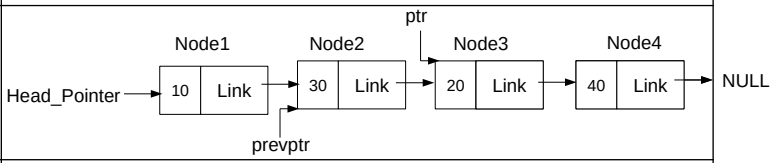
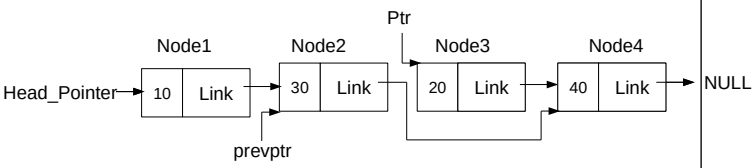
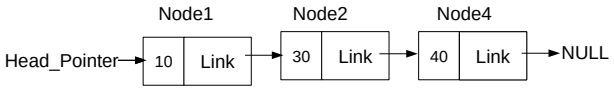
Step 5: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) If Head_Pointer → data = Data *(Condition of matching Data in 1st node)*, then
i) Print "The deletion of a node before the first node is not possible".
Otherwise
ii) If ptr → link ≠ NULL and ptr → link → data = Value *[for 2nd node selection]*, then
1. Delete the node at the beginning of the linked list
Otherwise
2. Repeat 3. and 4. while ptr → link ≠ NULL and ptr → link → data ≠ Value
[Loop for traversal of nodes]
3. Set prevptr = ptr
4. Set ptr = ptr → link
5. If ptr → link = NULL, then
A. Print " The node before which deletion will be done is not found".
Otherwise
B. Set prevptr → link = ptr → link
C. Remove the node pointed by ptr
D. Print " The node has been deleted before the specified node".
[End of If-Else (5.)]
[End of If-Else (ii)]
[End of If-Else (b)]
[End of If-Else (Step 5)]

Step 6: Set prevptr = NULL

Step 7: Set ptr = NULL

Step 8: Stop

In the above algorithm the pointer prevptr is used to point the predecessor node of the node pointed by the pointer ptr. Here two situations occur – in the first case the node at the beginning of the linked list is deleted when the second node is selected. The first case has already been explained previously. In the second case any other node except the first node will be removed. Only this situation is demonstrated pictorially in the following table.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer prevptr = Head_Pointer	
Traversal of Nodes Value = 40	prevptr = ptr ptr = ptr → link	
	prevptr = ptr ptr = ptr → link	
	prevptr → link = ptr → link	
Finally	Remove the node pointed by ptr [free(ptr)] ptr = NULL prevptr = NULL	

User defined function in C to delete a node before a specific node in a singly linked list

```
struct node *DeleteBeforeNode(struct node *headptr)
{
        struct node *ptr = headptr, *prevptr = headptr;

        printf("Enter the value of a node before which another node will be deleted: ");
        scanf("%d", &Value);

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else if(headptr->data == Value)
        {
                printf("The deletion of a node before the first node is not possible.\n\n");
        }
        else if(ptr->link != NULL && ptr->link->data == Value)
        {
                headptr = DeleteAtStart(headptr);
        }
```

```
        else
        {
                while(ptr->link != NULL && ptr->link->data != Value)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }


                if(ptr->link == NULL)
                        printf("The node before which deletion will be done is not found.\n\n");
                else
                {
                        prevptr->link = ptr->link;

                        free(ptr);
                        printf("The node has been deleted before the specified node\n\n");
                }

        }

        prevptr = NULL;
        ptr = NULL;
        return headptr;
}
```

12. Delete a node after a specific node in a singly linked list – Here three error cases will be considered.
The three error cases are:
1) The first error case occurs when the linked list is empty.
2) The second error case occurs when the last node is selected. Obviously there will be no node for deletion after the last node.
3) The third error case occurs when a value given by a user is not found in any node of the linked list.

In case of removal of a node after another node, the value of the node after with a node will be deleted is provided by a user. After successive traversal of the nodes, the node whose value is given is pointed by a pointer prevptr and the successor node is pointed by another pointer ptr. Now the node pointed by ptr will be removed from the linked list as per the requirement. Here prevptr always points to the predecessor node of the node pointed by ptr.

Algorithm to delete a node after the specified node of a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Set prevptr = Head_Pointer

Step 4: Input Value of a node after which a node will be deleted

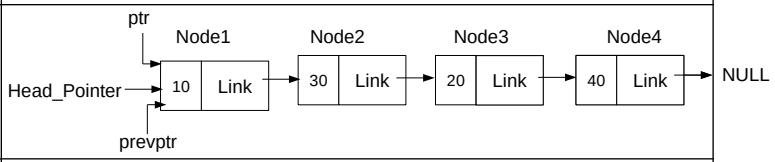
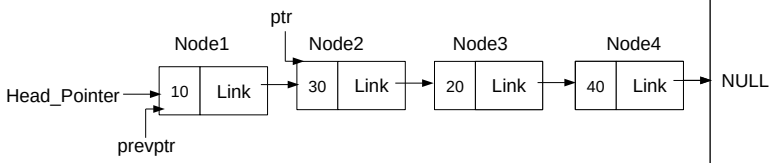
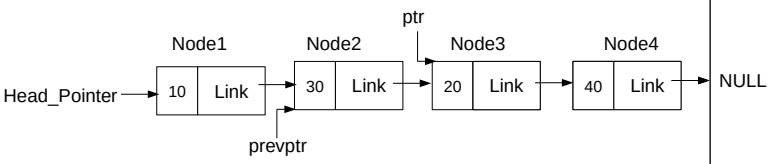
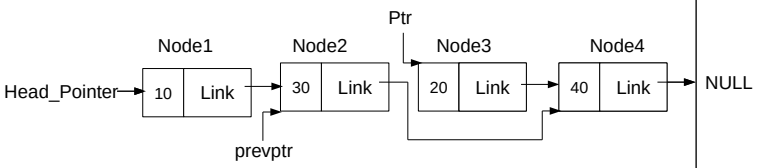
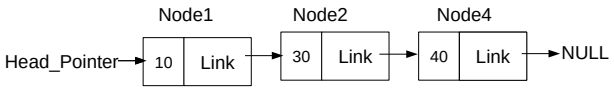
Step 5: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) Set ptr = ptr → link
c) Repeat d) and e) while ptr ≠ NULL and prevptr → data ≠ Value
Loop for traversal of nodes]
d) Set prevptr = ptr
e) Set ptr = ptr → link
f) If prevptr → data ≠ Value, then
i) Print "The node after which deletion will be done is not found".
Otherwise
ii) If prevptr → link = NULL, then
1. Print "The deletion of a node after the last node is not possible".
Otherwise
2. Set prevptr → link = ptr → link
3. Remove the node pointed by ptr
4. Print " The node has been deleted after the specified node".
[End of If-Else (ii)]
[End of If-Else (f)]
[End of If-Else (Step 5)]

Step 6: Set prevptr = NULL

Step 7: Set ptr = NULL

Step 8: Stop

In the above algorithm the specified value is searched among the nodes of the linked list. The data of each and every node is compared to the given value during the traversal of nodes from the 1st node. If the value is found in the data part of any node, the node is pointed by prevptr and the successor node is pointed by ptr. Ultimately the successor node will be removed from the linked list as per our requirement. The step-by-step demonstration of this algorithm is given below.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer prevptr = Head_Pointer	
	ptr = ptr → link	
Traversal of Nodes Value = 30	prevptr = ptr ptr = ptr → link	
	prevptr → link = ptr → link	
Finally	Remove the node pointed by ptr [free(ptr)] ptr = NULL prevptr = NULL	

User defined function in C to delete a node after a specific node in a singly linked list

```
struct node *DeleteAfterNode(struct node *headptr)
{
        struct node *ptr = headptr, *prevptr = headptr;

        printf("Enter the value of a node after which another node will be deleted: ");
        scanf("%d", &Value);

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else
        {
                ptr = ptr->link;

                while(ptr != NULL && prevptr->data != Value)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }
```

```
                if(prevptr->data != Value)
                        printf("The node after which deletion will be done is not found.\n\n");
                else if(prevptr->link == NULL)
                        printf("The deletion of a node after the last node is not possible.\n\n");
                else
                {
                    prevptr->link = ptr->link;
                    free(ptr);
                    printf("The node has been deleted after the specified node successfully.");
                }

        }

        prevptr = NULL;
        ptr = NULL;
        return headptr;
}
```

13. Delete a specific node of a singly linked list – In this case a value is specified by a user and that value is searched among the nodes of the linked list. If the value is found in the data part of any node, that particular node will be removed from the linked list. Here two error cases occurs.
1) When the linked list is completely empty, there will be no node for deletion.
2) When the specified value is not found in the data part of any node of the linked list.

Here two cases will happen for deleting a particular node.
1) When first node is selected for deletion, the algorithm of "deletion at the beginning of the linked list" is applied.
2) When any other node except the first node is chosen for removal, the other procedure is applied.

Algorithm to delete a specific node in a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Set prevptr = Head_Pointer

Step 4: Input Value of a node which will be deleted

Step 5: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) If Head_Pointer → data = Data *[Condition for matching Data in 1st node]*, then
i) Delete the node at the beginning of the linked list

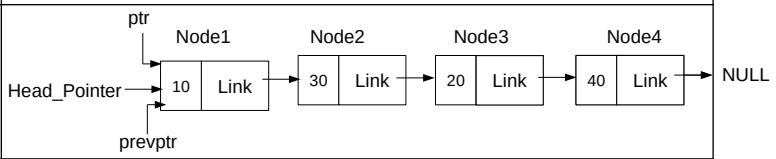
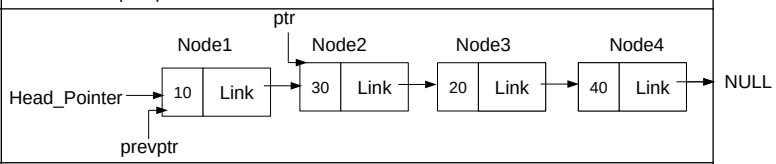
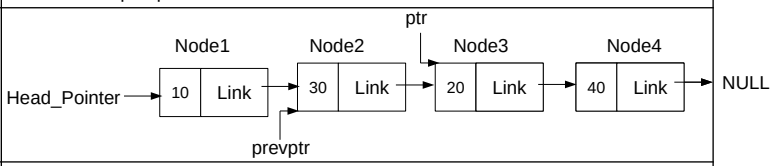
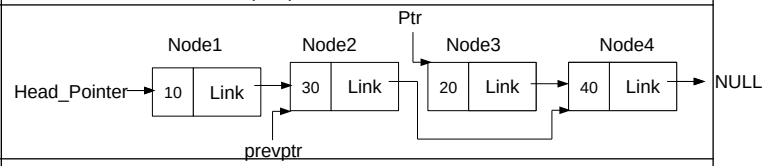
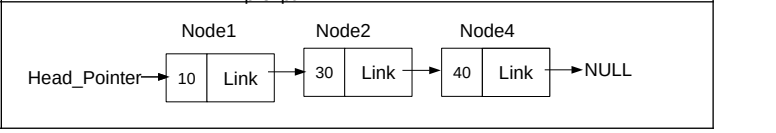
Otherwise
ii) Repeat iii) and iv) while ptr ≠ NULL and ptr → data ≠ Value
[Loop for traversal of nodes]
iii) Set prevptr = ptr
iv) Set ptr = ptr → link
v) If ptr = NULL, then
1. Print "The specified node which will be deleted is not found.".
Otherwise
2. Set prevptr → link = ptr → link
3. Remove the node pointed by ptr
4. Print "The specified node has been deleted successfully.".
[End of If-Else (v)]
[End of If-Else (b)]
[End of If-Else (Step 5)]

Step 6: Set prevptr = NULL

Step 7: Set ptr = NULL

Step 8: Stop

In this algorithm the same procedure is used to delete a node at the beginning of the linked list when the data value of the first node is selected for deletion. Therefore the same pictorial representation of deletion at the beginning of the linked list is not shown here. Only the step by step demonstration of the part where any other node except the first node is deleted is represented in the following table.

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer prevptr = Head_Pointer	
Traversal of Nodes Value = 20	prevptr = ptr ptr = ptr → link	
	prevptr = ptr ptr = ptr → link	
	prevptr → link = ptr → link	
Finally	Remove the node pointed by ptr [free(ptr)] ptr = NULL prevptr = NULL	

User defined function in C to delete a specific node in a singly linked list

```
struct node *DeleteNode(struct node *headptr)
{
        struct node *ptr = headptr, *prevptr = headptr;

        printf(“Enter the value of a node which will be deleted: “);
        scanf(“%d”, &Value);

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else if(headptr->data == Value)
        {
                headptr = DeleteAtStart(headptr);
        }
        else
        {
                while(ptr != NULL && ptr->data != Value)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }

                if(ptr == NULL)
                        printf("The specified node which will be deleted is not found.\n\n");
                else
                {
                        prevptr->link = ptr->link;
                        free(ptr);
                        printf("The specified node has been deleted successfully.\n\n");
                }

        }

        prevptr = NULL;
        ptr = NULL;
        return headptr;
}
```

14. Delete a node at a specific position of a singly linked list – Here a specific position like 1st, 2nd, 3rd etc is given by a user and the particular node at that given position will be deleted from the linked list.

Two error cases may happen in this case.
1) When the linked list is completely empty.

2) When the given position is not valid for a particular linked list. For example – if 0^{th} position or 5^{th} position is given for a linked list with 4 nodes, then no node is present at the 0^{th} position or at the 5^{th} position. Here these positions will be considered as invalid.

In addition to this, two situations for deletion of a node may occur.
1) If the 1^{st} position is given by a user, then the same algorithm to delete a node at the beginning of the linked list will be followed.
2) If any other position except 1^{st} position is given, then another procedure will be followed.

Algorithm to delete a node at a specific position in a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Set prevptr = Head_Pointer

Step 4: Input Position where a node will be deleted

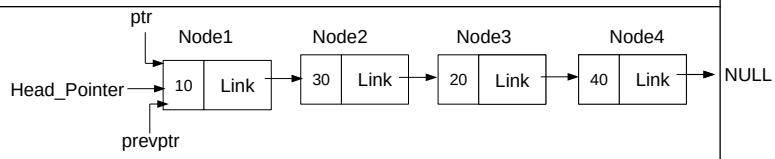
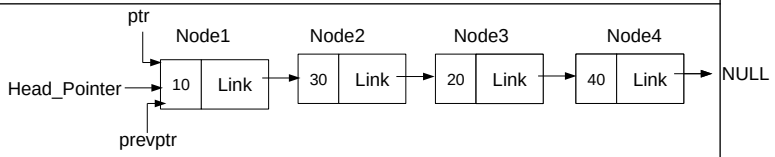
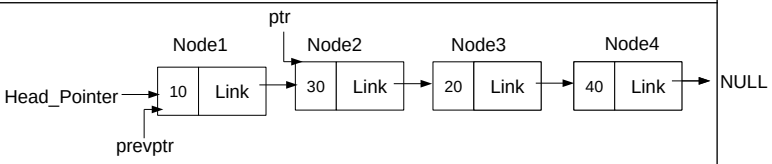
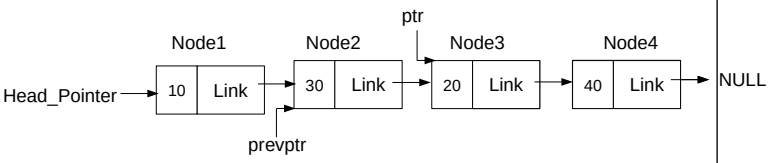
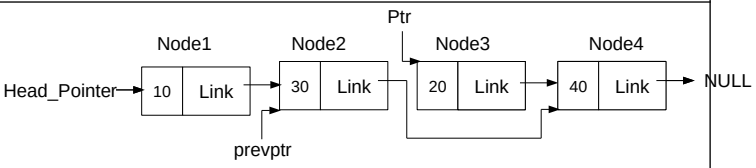
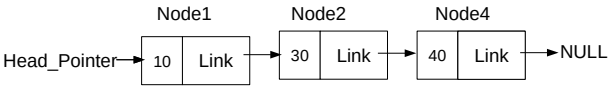
Step 5: Count the number of nodes and store that count into the variable NoOfNodes.

Step 6: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) If Position < 1 or Position > NoOfNodes, then
i) Print "The specified position is wrong".
Otherwise
ii) If Position = 1 *[Condition for 1^{st} node selection]*, then
1. Delete the node at the beginning of the linked list
Otherwise
2. Set i = 1
3. Repeat 4. and 5. while i < Position
[Loop for traversal of nodes]
4. Set prevptr = ptr
5. Set ptr = ptr → link
6. Set prevptr→link = ptr→link
7. Remove the node pointed by ptr
8. Print "The node at the specified position has been deleted successfully".
[End of If-Else (ii)]
[End of If-Else (b)]
[End of If-Else (Step 6)]

Step 7: Set prevptr = NULL

Step 8: Set ptr = NULL

Step 9: Stop

Steps	Statements	Linked List after the execution of the statement
Initially	ptr = Head_Pointer prevptr = Head_Pointer	
Position = 3	NoOfNodes = 4	
Traversal of Nodes Position = 3	i = 1 i < Position is satisfied prevptr = ptr ptr = ptr → link	
	i = 2 i < Position is satisfied prevptr = ptr ptr = ptr → link	
	prevptr → link = ptr → link	
Finally	Remove the node pointed by ptr [free(ptr)] ptr = NULL prevptr = NULL	

User defined function in C to delete a node at a specific position in a singly linked list

```
struct node *DeleteAtPos(struct node *headptr)
{
        struct node *ptr = headptr, *prevptr = headptr;
        int i, NoOfNodes;

        NoOfNodes = CountNodes(headptr);

        printf("Enter the position where a node will be deleted: ");
        scanf("%d", &Position);

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
```

```
        else if(Position<1 || Position > NoOfNodes)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                headptr = DeleteAtStart(headptr);
        }
        else
        {
                for(i=1; i<Position; i++)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }

                prevptr->link = ptr->link;
                free(ptr);
                printf("The node at the specified position has been deleted successfully.\n\n");
        }

        prevptr = NULL;
        ptr = NULL;
        return headptr;
}
```

In the above discussion fourteen different operations on singly linked list have been explained with the help of algorithms and the corresponding pictorial representation. All these operations are combined in a single C program and implemented using different user-defined functions which are already given in the previous sections. The fourteen different operations along with the user-defined functions are given below for recapitulation.

SL	Operations applied on singly linked list	User-defined functions
1	Creating a singly linked list	CreateNode( )
2	Displaying the data of all nodes in a singly linked list	DisplayNodes( )
3	Counting the number of nodes in a singly linked list	CountNodes( )
4	Insert a node at the beginning of a singly linked list	InsertAtStart( )
5	Insert a node at the end of a singly linked list	InsertAtLast( )
6	Insert a node before a specific node in a singly linked list	InsertBeforeNode( )
7	Insert a node after a specific node in a singly linked list	InsertAfterNode( )
8	Insert a node at a specific position of a singly linked list	InsertAtPos( )
9	Delete a node at the beginning of a singly linked list	DeleteAtStart( )
10	Delete a node at the end of a singly linked list	DeleteAtLast( )

SL	Operations applied on singly linked list	User-defined functions
11	Delete a node before a specific node in a singly linked list	DeleteBeforeNode( )
12	Delete a node after a specific node in a singly linked list	DeleteAfterNode( )
13	Delete a specific node of a singly linked list	DeleteNode( )
14	Delete a node at a specific position of a singly linked list	DeleteAtPos( )

C Program to create a singly linked list, implement different insertion operations and different deletion operations

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
        int data;
        struct node *link;
};

struct node *CreateNode();
struct node *InsertAtEmpty(int );
struct node *InsertAtEnd(struct node *,int );
void DisplayNodes(struct node *);
int CountNodes(struct node *);
struct node *FreeNode(struct node *);
struct node *InsertAtStart(struct node *,int );
struct node *InsertAtLast(struct node *,int );
struct node *InsertBeforeNode(struct node *,int ,int );
struct node *InsertAfterNode(struct node *,int ,int );
struct node *InsertAtPos(struct node *,int ,int );
struct node *DeleteAtStart(struct node *);
struct node *DeleteAtLast(struct node *);
struct node *DeleteBeforeNode(struct node *,int );
struct node *DeleteAfterNode(struct node *,int );
struct node *DeleteNode(struct node *,int );
struct node *DeleteAtPos(struct node *,int );

int main()
{
        struct node *head = NULL;
        int option, Value, Data, Position, NodeCount;
        system("clear");

        while(1)
        {
                printf("0: Exit\n");
                printf("1: Create singly linked list\n");
                printf("2: Display all nodes of the singly linked list\n");
                printf("3: Display no. of nodes in the singly linked list\n");
```

```
printf("4: Insert a node at the starting of the singly linked list\n");
printf("5: Insert a node at the end of the singly linked list\n");
printf("6: Insert a node before a specific node of the singly linked list\n");
printf("7: Insert a node after a specific node of the singly linked list\n");
printf("8: Insert a node at a specific position of the singly linked list\n");
printf("9: Delete a node at the starting of the singly linked list\n");
printf("10: Delete a node at the end of the singly linked list\n");
printf("11: Delete a node before a specific node of the singly linked list\n");
printf("12: Delete a node after a specific node of the singly linked list\n");
printf("13: Delete a specific node of the singly linked list\n");
printf("14: Delete a node at a specific position of the singly linked list\n");
printf("15: Delete all nodes of the singly linked list\n");
printf("\n");
printf("Enter your option: ");
scanf("%d", &option);

switch(option)
{
        case 0: head = FreeNode(head);
                exit(1);

        case 1: head = CreateNode();
                break;

        case 2: DisplayNodes(head);
                break;

        case 3: NodeCount = CountNodes(head);
                printf("No. of nodes in the linked list: %d\n\n", NodeCount);
                break;

        case 4: printf("Enter the data of a new node to be inserted at the starting: ");
                scanf("%d", &Data);
                head = InsertAtStart(head, Data);
                break;

        case 5: printf("Enter the data of a new node to be inserted at the last: ");
                scanf("%d", &Data);
                head = InsertAtLast(head, Data);
                break;

        case 6: printf("Enter the value of a node before which a new node will be
                        inserted: ");
                scanf("%d", &Value);
                printf("Enter the data of the new node to be inserted before the
                         specified node: ");
                scanf("%d", &Data);
                head = InsertBeforeNode(head, Value, Data);
                break;

        case 7: printf("Enter the value of a node after which a new node will be
                        inserted: ");
```

```
                    scanf("%d", &Value);
                    printf("Enter the data of the new node to be inserted after the
                        specified node: ");
                    scanf("%d", &Data);
                    head = InsertAfterNode(head, Value, Data);
                    break;

                case 8: printf("Enter the position where a node will be inserted: ");
                    scanf("%d", &Position);
                    printf("Enter the data of the node to be inserted at the specified
                        position: ");
                    scanf("%d", &Data);
                    head = InsertAtPos(head, Position, Data);
                    break;

                case 9: head = DeleteAtStart(head);
                    break;

                case 10: head = DeleteAtLast(head);
                    break;

                case 11: printf("Enter the value of a node before which a node will be
                        deleted: ");
                    scanf("%d", &Value);
                    head = DeleteBeforeNode(head, Value);
                    break;

                case 12: printf("Enter value of a node after which a node will be deleted: ");
                    scanf("%d", &Value);
                    head = DeleteAfterNode(head, Value);
                    break;

                case 13: printf("Enter the value of a node which will be deleted: ");
                    scanf("%d", &Value);
                    head = DeleteNode(head, Value);
                    break;

                case 14: printf("Enter the position where the node will be deleted: ");
                    scanf("%d", &Position);
                    head = DeleteAtPos(head, Position);
                    break;

                case 15: head = FreeNode(head);
                    printf("All nodes of the singly linked list have been deleted
                        successfully.\n\n");
                    break;

                default: printf("Wrong Option\n");
                    break;
            }
        }
```

```
	return 0;
}

struct node *CreateNode()
{
	struct node *headptr = NULL, *ptr;
	int i, n, Data;

	printf("Enter the no. of nodes to be created in the singly linked list: ");
	scanf("%d", &n);

	for(i=0; i<n; i++)
	{
		printf("Enter Data%d: ", i+1);
		scanf("%d", &Data);

		if(i == 0)
		{
			headptr = InsertAtEmpty(Data);
			ptr = headptr;
		}
		else
		{
			ptr = InsertAtEnd(ptr, Data);
		}
	}

	printf("Singly Linked list has been created successfully.\n\n");

	ptr = NULL;
	return headptr;
}

struct node *InsertAtEmpty(int Data)
{
	struct node *headptr;
	headptr = (struct node *)malloc(sizeof(struct node));

	headptr->data = Data;
	headptr->link = NULL;

	return headptr;
}

struct node *InsertAtEnd(struct node *ptr,int Data)
{
	ptr->link = (struct node *)malloc(sizeof(struct node));
	ptr->link->data = Data;
	ptr->link->link = NULL;

	return ptr->link;
}
```

```
void DisplayNodes(struct node *ptr)
{
	printf("The singly linked list is given below:\n");

	while(ptr != NULL)
	{
		printf("%-5d", ptr->data);
		ptr = ptr->link;
	}

	printf("\n\n");

	ptr = NULL;
}

int CountNodes(struct node *ptr)
{
	int count = 0;

	while(ptr != NULL)
	{
		count++;
		ptr = ptr->link;
	}

	ptr = NULL;

	return count;
}

struct node *InsertAtStart(struct node *headptr,int Data)
{
	struct node *ptr;

	ptr = (struct node *)malloc(sizeof(struct node));
	ptr->data = Data;
	ptr->link = headptr;
	headptr = ptr;

	printf("A new node has been inserted at the starting successfully.\n\n");

	ptr = NULL;

	return headptr;
}
```

```
struct node *InsertAtLast(struct node *headptr,int Data)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		headptr = InsertAtEmpty(Data);
	}
	else
	{
		while(ptr->link != NULL)
			ptr = ptr->link;

		ptr = InsertAtEnd(ptr, Data);
	}

	printf("A new node has been inserted at the end successfully.\n\n");

	ptr = NULL;
	return headptr;
}

struct node *InsertBeforeNode(struct node *headptr,int Value,int Data)
{
	struct node *ptr = headptr, *newnodeptr;

	if(ptr->data == Value)
	{
		headptr = InsertAtStart(headptr, Data);
	}
	else
	{
		while(ptr->link != NULL && ptr->link->data != Value)
			ptr = ptr->link;

		if(ptr->link == NULL)
			printf("The node before which insertion will be done is not found.\n\n");
		else
		{
			newnodeptr = (struct node *)malloc(sizeof(struct node));
			newnodeptr->data = Data;
			newnodeptr->link = ptr->link;
			ptr->link = newnodeptr;
			printf("A new node has been inserted before the specified node.\n\n");
		}
	}

	newnodeptr = NULL;
	ptr = NULL;

	return headptr;
}
```

```
struct node *InsertAfterNode(struct node *headptr,int Value,int Data)
{
	struct node *ptr = headptr, *newnodeptr;

	while(ptr != NULL && ptr->data != Value)
		ptr = ptr->link;

	if(ptr == NULL)
		printf("The node after which insertion will be done is not found.\n\n");
	else
	{
		newnodeptr = (struct node *)malloc(sizeof(struct node));
		newnodeptr->data = Data;
		newnodeptr->link = ptr->link;
		ptr->link = newnodeptr;
		printf("A new node has been inserted after the specified node successfully.\n\n");
	}

	newnodeptr = NULL;
	ptr = NULL;
	return headptr;
}

struct node *InsertAtPos(struct node *headptr,int Position,int Data)
{
	struct node *ptr = headptr, *newnodeptr;
	int i, NoOfNodes;
	NoOfNodes = CountNodes(headptr);

	if(Position < 1 || Position > NoOfNodes + 1)
	{
		printf("The specified position is wrong.\n\n");
	}
	else if(Position == 1)
	{
		headptr = InsertAtStart(headptr, Data);
	}
	else
	{
		for(i=1; i<Position - 1; i++)
			ptr = ptr->link;

		newnodeptr = (struct node *)malloc(sizeof(struct node));
		newnodeptr->data = Data;
		newnodeptr->link = ptr->link;
		ptr->link = newnodeptr;
		printf("A new node has been inserted at a specific position successfully.\n\n");
	}
	newnodeptr = NULL;
	ptr = NULL;
	return headptr;
}
```

```
struct node *DeleteAtStart(struct node *headptr)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else
	{
		headptr = ptr->link;
		free(ptr);
		printf("The node has been deleted at the starting successfully.\n\n");
	}
	ptr = NULL;
	return headptr;
}

struct node *DeleteAtLast(struct node *headptr)
{
	struct node *ptr = headptr, *prevptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else
	{

		while(ptr->link != NULL)
		{
			prevptr = ptr;
			ptr = ptr->link;
		}

		prevptr->link = NULL;
		if(ptr == headptr)
			headptr = NULL;
		free(ptr);
		printf("The node has been deleted at the end successfully.\n\n");

	}

	prevptr = NULL;
	ptr = NULL;
	return headptr;
}
```

```
struct node *DeleteBeforeNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr, *prevptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else if(headptr->data == Value)
	{
		printf("The deletion of a node before the first node is not possible.\n\n");
	}
	else if(ptr->link != NULL && ptr->link->data == Value)
	{
		headptr = DeleteAtStart(headptr);
	}
	else
	{
		while(ptr->link != NULL && ptr->link->data != Value)
		{
			prevptr = ptr;
			ptr = ptr->link;
		}


		if(ptr->link == NULL)
			printf("The node before which deletion will be done is not found.\n\n");
		else
		{
			prevptr->link = ptr->link;

			free(ptr);
			printf("The node has been deleted before the specified node.\n\n");
		}

	}

	prevptr = NULL;
	ptr = NULL;
	return headptr;
}
```

```
struct node *DeleteAfterNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr, *prevptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else
	{
		ptr = ptr->link;

		while(ptr != NULL && prevptr->data != Value)
		{
			prevptr = ptr;
			ptr = ptr->link;
		}

		if(prevptr->data != Value)
			printf("The node after which deletion will be done is not found.\n\n");
		else if(prevptr->link == NULL)
			printf("The deletion of a node after the last node is not possible.\n\n");
		else
		{
			prevptr->link = ptr->link;
			free(ptr);
			printf("The node has been deleted after the specified node.\n\n");
		}

	}

	prevptr = NULL;
	ptr = NULL;
	return headptr;
}

struct node *DeleteNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr, *prevptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else if(headptr->data == Value)
	{
		headptr = DeleteAtStart(headptr);
	}
	else
	{
```

```
                while(ptr != NULL && ptr->data != Value)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }
                if(ptr == NULL)
                        printf("The specified node which will be deleted is not found.\n\n");
                else
                {
                        prevptr->link = ptr->link;
                        free(ptr);
                        printf("The specified node has been deleted successfully.\n\n");
                }
        }
        prevptr = NULL;
        ptr = NULL;
        return headptr;
}

struct node *DeleteAtPos(struct node *headptr,int Position)
{
        struct node *ptr = headptr, *prevptr = headptr;
        int i, NoOfNodes;
        NoOfNodes = CountNodes(headptr);

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else if(Position<1 || Position > NoOfNodes)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                headptr = DeleteAtStart(headptr);
        }
        else
        {
                for(i=1; i<Position; i++)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }
                prevptr->link = ptr->link;
                free(ptr);
                printf("The node at the specified position has been deleted successfully.\n\n");
        }
        prevptr = NULL;
        ptr = NULL;
        return headptr;
}
```

```
struct node *FreeNode(struct node *headptr)
{
        struct node *ptr;

        while(headptr != NULL)
        {
                ptr = headptr->link;
                free(headptr);
                headptr = ptr;
        }

        headptr = NULL;
        ptr = NULL;
        return headptr;
}
```

2) Doubly Linked List – A doubly linked list is a special type of linked list where a node contains some data, one pointer to the next node and another pointer to the previous node. Here the pointer to the next node holds the address of the successor node and the pointer to the previous node holds the address of the predecessor node of the present node in the doubly linked list. Due to the presence of the previous pointer as well as next pointer it is possible to traverse in both directions through the doubly linked list. That means we can move from 1st node to 2nd node and 2nd node to 1st node as well. A doubly linked list is shown in Fig.6.3 for realization of the structure of a doubly linked list.

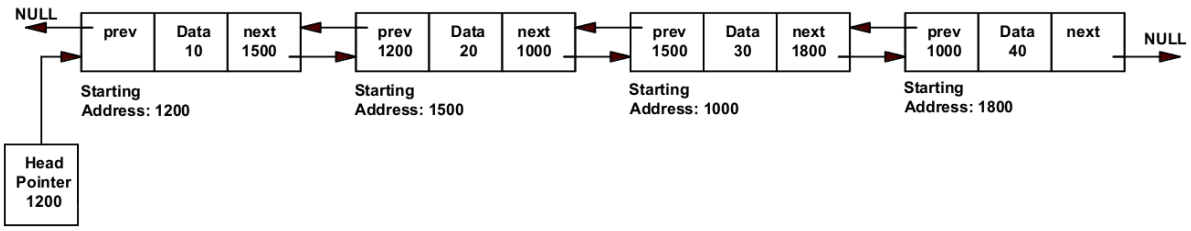


Fig.6.3: Pictorial view of a doubly linked list with 4 nodes

Like singly linked list the starting node of the doubly linked list is always pointed by a pointer called head pointer of the linked list. Here the pointer pointing to the next node is designated as 'next' and the pointer pointing to the previous node is denoted as 'prev' in the program. Hence the pointer next is made NULL for the last node and the pointer prev is made NULL for the first node in case of a doubly linked list. Moreover if the head pointer is NULL, then it implies that the doubly linked list is empty.

Declaring a doubly linked list – We know that every node of a doubly linked list has three parts – i) data part, ii) a pointer to point the next node and iii) a pointer to point the previous node. To implement the construction of a node in a doubly linked list a structure comprising of these members is declared in C language as given below.

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

1. Creating a doubly linked list – Using this function a user can create a doubly linked list with n no. of nodes according to his requirement where n is given by the user. After completion of this operation a doubly linked list will be generated with n no. of nodes. The algorithm to create n no. of nodes in a doubly linked list is given below.

Algorithm of doubly linked list creation:

Step 1: Start

Step 2: Set Head_Pointer = NULL

Step 3: Input n for number of nodes to be created

Step 4: Set i = 0

Step 5: Repeat Step 6 to Step 8 while i < n

Step 6: Input Data to be stored into the node

Step 7: If i = 0 (for first node creation), then
- a) Create a new node pointed by New_Node_Pointer
  - Set New_Node_Pointer → prev = NULL
  - Set New_Node_Pointer → data = Data
  - Set New_Node_Pointer → next = NULL
  - Set Head_Pointer = New_Node_Pointer
  - Set Ptr = Head_Pointer

Otherwise
- b) Create a new node pointed by New_Node_Pointer
  - Set New_Node_Pointer → data = Data
  - Set ptr → next = New_Node_Pointer
  - Set ptr → next → prev = ptr
  - Set ptr → next → next = NULL
  - Set ptr = ptr → next

[End of If-Else (Step 7)]

Step 8: Set i = i + 1

Step 9: Print “Doubly Linked List has been created successfully”.

Step 10: Set ptr = NULL

Step 11: Set New_Node_Pointer = NULL

Step 12: Stop

User defined function in C to create a doubly linked list with n no. of nodes

```
struct node *CreateNode()
{
	struct node *headptr = NULL, *ptr;
	int i, n, Data;

	printf("Enter the no. of nodes to be created in the doubly linked list: ");
	scanf("%d", &n);

	for(i=0; i<n; i++)
	{
		printf("Enter Data%d: ", i+1);
		scanf("%d", &Data);
		if(i == 0)
		{
			headptr = InsertAtEmpty(Data);
			ptr = headptr;
		}
		else
		{
			ptr = InsertAtEnd(ptr, Data);
		}
	}
	printf("Doubly Linked list has been created successfully.\n\n");
	ptr = NULL;
	return headptr;
}

struct node *InsertAtEmpty(int Data)
{
	struct node *headptr;
	headptr = (struct node *)malloc(sizeof(struct node));

	headptr->prev = NULL;
	headptr->data = Data;
	headptr->next = NULL;

	return headptr;
}
```

```
struct node *InsertAtEnd(struct node *ptr,int Data)
{
        ptr->next = (struct node *)malloc(sizeof(struct node));
        ptr->next->data = Data;
        ptr->next->prev = ptr;
        ptr->next->next = NULL;

        return ptr->next;
}
```

2. Displaying the data of all nodes in a doubly linked list – This operation displays/ prints the data of all nodes in a doubly linked list in forward direction i.e. from starting node to last node. In this case a temporary pointer is taken to point the starting node or first node initially and this temporary pointer prints the data of every node in the liked list while traversing from first node to the last node.

Algorithm of displaying the data of all nodes in doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Repeat Step 4 to Step 5 while ptr ≠ NULL

Step 4: Print ptr → data

Step 5: Set ptr = ptr → next

Step 6: Set ptr = NULL

Step 7: Stop

In the above algorithm all the nodes of a doubly linked list have been printed in forward direction from first node to the last node using the next pointer. If all the nodes are displayed properly, then it is clear that all the nodes of the doubly linked list are connected through the next link of each node. But it is not proved that the connections between the nodes are established or not with the help of prev link (previous pointer of a node). To check this the data of all nodes should be printed in reverse direction as well i.e. from the last node to the first node. Therefore these two operations - i) displaying all nodes in forward direction and ii) displaying all nodes in reverse direction have been incorporated in the same Display function in C language for a doubly linked list.

User defined function in C to display the data of all nodes in a doubly linked list

```
void DisplayNode(struct node *ptr)
{
        printf("The Doubly linked list is given below:\n");

        while(ptr != NULL)     //While loop for displaying all nodes in forward direction
        {
                printf("%-5d", ptr->data);
                if(ptr->next == NULL)
                        break;
                ptr = ptr->next;
        }

        printf("\n\n");

        printf("The Doubly linked list is given below in reverse order:\n");

        while(ptr != NULL)      //While loop for displaying all nodes in reverse direction
        {
                printf("%-5d", ptr->data);
                if(ptr->prev == NULL)
                        break;
                ptr = ptr->prev;
        }

        printf("\n\n");

        ptr = NULL;
}
```

3. Counting the number of nodes in a doubly linked list – This operation counts the number of nodes in a doubly linked list. Here a variable 'count' is initialized to zero and incremented by one each time the pointer ptr traverses the successive nodes in the linked list.

Algorithm of counting the number of nodes in doubly linked list:

Step 1: Start

Step 2: Set count = 0

Step 3: Set ptr = Head_Pointer

Step 4: Repeat Step 5 to Step 6 while ptr ≠ NULL

Step 5: Set count = count + 1

Step 6: Set ptr = ptr → next

Step 7: Print count to display the number of nodes in the doubly linked list

Step 8: Set ptr = NULL

Step 9: Stop

User defined function in C to count the number of nodes in a doubly linked list

```
int CountNodes(struct node *ptr)
{
        int count = 0;

        while(ptr != NULL)
        {
                count++;
                ptr = ptr->next;
        }

        return count;
}
```

4. Insert a node at the beginning of a doubly linked list – Here a new node will be inserted at the beginning of a doubly linked list. After the insertion the new node becomes the starting node or the first node of the linked list. Therefore the Head_Pointer must be shifted from the second node to the first node to point at the beginning of the linked list.

Algorithm to insert a node at the beginning of a doubly linked list:

Step 1: Start

Step 2: Input Data to be stored into the new node

Step 3: Create a new node pointed by a pointer 'ptr'

Step 4: Set ptr → data = Data

Step 5: Set ptr → prev = NULL

Step 6: Set ptr → next = Head_Pointer

Step 7: Set Head_Pointer → prev = ptr

Step 8: Set Head_Pointer = ptr

Step 9: Print "A new node has been inserted at the starting successfully".

Step 10: Set ptr = NULL

Step 11: Stop

User defined function in C to insert a node at the beginning of a doubly linked list

```
struct node *InsertAtStart(struct node *headptr,int Data)
{
        struct node *ptr;

        ptr = (struct node *)malloc(sizeof(struct node));
        ptr->prev = NULL;
        ptr->data = Data;
        ptr->next = headptr;
        headptr->prev = ptr;
        headptr = ptr;

        printf("A new node has been inserted at the starting successfully.\n\n");

        ptr = NULL;

        return headptr;
}
```

5. Insert a node at the end of a doubly linked list – This operation inserts a newly created node at the end of a doubly linked list i.e. it add the new node after the last node of the existing linked list. After the insertion at the end, the new node becomes the last node of the linked list and that's why the new node must be terminated with NULL using the next pointer.

Algorithm to insert a node at the end of a doubly linked list:

Step 1: Start

Step 2: Input Data to be stored into the new node

Step 3: Set ptr = Head_Pointer

Step 4: If ptr = NULL, then
- a) Create a new node pointed by Head_Pointer
  - Set Head_Pointer → data = Data
  - Set Head_Pointer → prev = NULL
  - Set Head_Pointer → next = NULL

Otherwise
- b) Repeat 1. while ptr → next ≠ NULL
  1. Set ptr = ptr → next

Create a new node pointed by ptr → next
Set ptr → next → data = Data
Set ptr → next → prev = ptr
Set ptr → next → next = NULL
Set ptr = ptr → next
[End If-Else Step 4]

Step 5: Print "A new node has been inserted at the end successfully".

Step 6: Set ptr = NULL

Step 7: Stop

User defined function in C to insert a node at the end of a doubly linked list

```
struct node *InsertAtLast(struct node *headptr,int Data)
{
	struct node *ptr = headptr;
	if(ptr == NULL)
	{
		headptr = InsertAtEmpty(Data);
		ptr = headptr;
	}
	else
	{
		while(ptr->next != NULL)
			ptr = ptr->next;
		ptr = InsertAtEnd(ptr, Data);
	}
	printf("A new node has been inserted at the end successfully.\n\n");

	ptr = NULL;
	return headptr;
}
struct node *InsertAtEmpty(int Data)
{
	struct node *headptr;
	headptr = (struct node *)malloc(sizeof(struct node));

	headptr->prev = NULL;
	headptr->data = Data;
	headptr->next = NULL;

	return headptr;
}
```

```
struct node *InsertAtEnd(struct node *ptr,int Data)
{
        ptr->next = (struct node *)malloc(sizeof(struct node));
        ptr->next->data = Data;
        ptr->next->prev = ptr;
        ptr->next->next = NULL;

        return ptr->next;
}
```

6. Insert a node before a specific node in a doubly linked list – This function will insert a new node just before another node which is specified by the user. Here we have to consider two situations.

In first situation the specified node is the first node in the doubly linked list. So the insertion of a new node before the first node is basically the insertion of a new node at the beginning of the link list, which has been already discussed. In second situation when the specified node is not first node, we have to insert the newly created node in between the specified node and the node just before the specified node. Except these two situations, there is another situation where the specified node is not found in the linked list. In that case an error message will be printed.

Algorithm of inserting a node before a specific node in a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node before which a new node will be inserted

Step 4: Input Data to be stored into the new node

Step 5: If ptr → data = Value *[Condition of finding Value in 1st node]*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → prev = NULL
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → next = Head_Pointer
Set Head_Pointer → prev = New_Node_Pointer
Set Head_Pointer = New_Node_Pointer
Otherwise
b) Repeat (c) while ptr → ≠ NULL and ptr → data ≠ Value
c) Set ptr = ptr → next

d) If ptr = NULL, then
i) Print “The node before which insertion will be done is not found”.
Otherwise
ii) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data

Set New_Node_Pointer → next = ptr
Set New_Node_Pointer → prev = ptr → prev
Set ptr → prev = New_Node_Pointer
Set New_Node_Pointer → prev → next = New_Node_Pointer
Print "A new node has been inserted before the specified node".
[End of If-Else (d)]
[End of If-Else (Step 5)]

Step 6: Set ptr = NULL

Step 7: Set New_Node_Pointer = NULL

Step 8: Stop

User defined function in C to insert a node before a specific node in a doubly linked list

```
struct node *InsertBeforeNode(struct node *headptr,int Value,int Data)
{
        struct node *ptr = headptr, *newnode;
        if(ptr->data == Value)
                headptr = InsertAtStart(headptr, Data);
        else
        {
                while(ptr != NULL && ptr->data != Value)
                        ptr = ptr->next;
                if(ptr == NULL)
                        printf("The node before which insertion will be done is not found\n");
                else
                {
                        newnode = (struct node *)malloc(sizeof(struct node));
                        newnode->data = Data;
                        newnode->next = ptr;
                        newnode->prev = ptr->prev;
                        ptr->prev = newnode;
                        newnode->prev->next = newnode;

                        printf("A new node has been inserted before the specified node
                        successfully.\n\n");
                }
        }
        newnode = NULL;
        ptr = NULL;
        return headptr;
}
```

```
struct node *InsertAtStart(struct node *headptr,int Data)
{
        struct node *ptr;

        ptr = (struct node *)malloc(sizeof(struct node));
        ptr->prev = NULL;
        ptr->data = Data;
        ptr->next = headptr;
        headptr->prev = ptr;
        headptr = ptr;

        printf("A new node has been inserted at the starting successfully.\n\n");

        ptr = NULL;

        return headptr;
}
```

7. Insert a node after a specific node in a doubly linked list – This operation will insert a new node just after another node which is specified by the user. Here at first the node with the specified value is searched through the traversal of the linked list and if the specified node is found, a new node with entered data will be inserted after it, otherwise an error message will be displayed.

Algorithm of inserting a node after a specific node in a singly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node after which a new node will be inserted

Step 4: Input Data to be stored into the new node

Step 5: Repeat Step 6 while ptr ≠ NULL and ptr → data ≠ Value

Step 6: Set ptr = ptr → next

Step 7: If ptr = NULL, then
a) Print "The node after which insertion will be done is not found".
Otherwise
b) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → prev = ptr
Set New_Node_Pointer → next = ptr → next
Set ptr → next = New_Node_Pointer

c) If New_Node_Pointer → next ≠ NULL *[Condition of not being last node]*, then
Set New_Node_Pointer → next → prev = New_Node_Pointer
[End of If (c)]

d) Print "A new node has been inserted after the specified node".
[End of If-Else (Step 7)]

Step 8: Set ptr = NULL

Step 9: Set New_Node_Pointer = NULL

Step 10: Stop

User defined function in C to insert a node after a specific node in a doubly linked list

```
struct node *InsertAfterNode(struct node *headptr,int Value,int Data)
{
        struct node *ptr = headptr, *newnode;

        while(ptr != NULL && ptr->data != Value)
                ptr = ptr->next;

        if(ptr == NULL)
                printf("The node after which insertion will be done is not found.\n\n");
        else
        {
                newnode = (struct node *)malloc(sizeof(struct node));
                newnode->data = Data;
                newnode->prev = ptr;
                newnode->next = ptr->next;
                ptr->next = newnode;
                if(newnode->next != NULL)
                        newnode->next->prev = newnode;
                printf("A node has been inserted after the specified node successfully\n");
        }
        newnode = NULL;
        ptr = NULL;

        return headptr;
}
```

8. Insert a node at a specific position of a doubly linked list – This function gives the opportunity to insert a new node with an entered data at a specified position of a doubly linked list. In this case the remaining part of the linked list starting from the specific position will be shifted right and the new node will be placed at the specified position. For example – if the specified position is 3rd in a doubly linked list of three nodes, then the new node will be placed at 3rd position and the 3rd node of the existing linked list will be shifted right one position to become 4th node.

Algorithm of inserting a node at a specific position of a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Position where a new node will be inserted

Step 4: Input Data to be stored into the new node

Step 5: Set NoOfNodes = Total number of nodes in the linked list

Step 6: If Position < 1 or Position > NoOfNodes + 1, then
a) Print "Entered position is Invalid".
Otherwise
b) If Position = 1, then
i) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → prev = NULL
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → next = Head_Pointer
Set Head_Pointer → prev = New_Node_Pointer
Set Head_Pointer = New_Node_Pointer
Otherwise
ii) Set i = 1
iii) Repeat iv) and v) while i < Position – 1
iv) Set ptr = ptr → next
v) Set i = i + 1
vi) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → prev = ptr
Set New_Node_Pointer → next = ptr → next
Set New_Node_Pointer → next = ptr → next
Set ptr → next = New_Node_Pointer
1. If New_Node_Pointer → next ≠ NULL *[Condition of not being last node]*,
Set New_Node_Pointer → next → prev = New_Node_Pointer
[End of If (1)]
vii) Print "A new node has been inserted at a specific position".
[End of If-Else (b)]
[End of If-Else (Step 6)]

Step 7: Set ptr = NULL

Step 8: Set New_Node_Pointer = NULL

Step 9: Stop

User defined function in C to insert a node at a specific position of a doubly linked list

```
struct node *InsertAtPos(struct node *headptr,int Position,int Data)
{
	struct node *ptr = headptr, *newnode;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(headptr);
	if(Position < 1 || Position > NoOfNodes + 1)
		printf("The specified position is wrong.\n\n");
	else if(Position == 1)
		headptr = InsertAtStart(headptr, Data);
	else
	{
		for(i=1; i<Position - 1; i++)
			ptr = ptr->next;

		newnode = (struct node *)malloc(sizeof(struct node));
		newnode->data = Data;
		newnode->prev = ptr;
		newnode->next = ptr->next;
		ptr->next = newnode;
		if(newnode->next != NULL)
			newnode->next->prev = newnode;

		printf("A new node has been inserted at a specific position successfully.\n");
	}
	newnode = NULL;
	ptr = NULL;
	return headptr;
}

int CountNodes(struct node *ptr)
{
	int count = 0;
	while(ptr != NULL)
	{
		count++;
		ptr = ptr->next;
	}
	return count;
}
```

```
struct node *InsertAtStart(struct node *headptr,int Data)
{
        struct node *ptr;

        ptr = (struct node *)malloc(sizeof(struct node));
        ptr->prev = NULL;
        ptr->data = Data;
        ptr->next = headptr;
        headptr->prev = ptr;
        headptr = ptr;
        printf("A new node has been inserted at the starting successfully.\n\n");

        ptr = NULL;
        return headptr;
}
```

9. Delete a node at the beginning of a doubly linked list – This operation deletes a node at the beginning of a doubly linked list. Basically it removes the first node from the linked list. As a result of this, the second node becomes the first node of the list. Therefore the Head_Pointer should be shifted from the first node to the second node and then the first node is removed from the linked list.

Algorithm to delete a node at the beginning of a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty"
Otherwise
b) Set Head_Pointer = ptr → next
c) If Head_Pointer ≠ NULL *(Condition for the existence of 2nd node)*, then
Set Head_Pointer → prev = NULL
[End of If (c)]
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
[End of If-Else (Step 3)]

Step 4: Set ptr = NULL

Step 5: Stop

User defined function in C to delete a node at the beginning of a doubly linked list

```
struct node *DeleteAtStart(struct node *headptr)
{
        struct node *ptr = headptr;

        if(ptr == NULL)
                printf("The linked list is empty.\n\n");
        else
        {
                headptr = ptr->next;
                if(headptr != NULL)
                        headptr->prev = NULL;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return headptr;
}
```

10. Delete a node at the end of a doubly linked list – In this operation the node at the end of a doubly linked list will be removed. Here two situations may happen. In one situation, in case of a linked list with a single node, the one and only node will be pointed by ptr, the Head_Pointer is made NULL and finally the node is removed. In another situation, the ptr points to the last node and ptr → prev → next is made NULL to make the previous node the last node after the deletion. Finally the node pointed by the pointer ptr will be deleted from the doubly linked list.

Algorithm to delete a node at the end of a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty"
Otherwise
b) Repeat c) while ptr → next ≠ NULL
c) Set ptr = ptr → next

d) If ptr = Head_Pointer *(Condition for single node in linked list)*, then
i) Set Head_Pointer = NULL
Otherwise
ii) Set ptr → prev → next = NULL
[End of If -Else (d)]
e) Remove the node pointed by ptr
f) Print "The node has been deleted at the end of the linked list"
[End of If-Else (Step 3)]

Step 4: Set ptr = NULL

Step 5: Stop

In this algorithm two situations have been considered.

1) When the linked list contains a single node, the node will be pointed by ptr first, then the Head_Pointer is made NULL and finally the starting node will be removed.

2) When the linked list has multiple nodes, the last node will be pointed by the pointer ptr and the node before the last node will be made last node after the deletion by setting ptr → prev → next = NULL. Finally the node pointed by ptr is removed from the doubly linked list.

User defined function in C to delete a node at the end of a doubly linked list

```
struct node *DeleteAtLast(struct node *headptr)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else
	{

		while(ptr->next != NULL)
		{
			ptr = ptr->next;
		}

		if(ptr == headptr)
			headptr = NULL;
		else
			ptr->prev->next = NULL;

		free(ptr);
		printf("The node has been deleted at the end successfully.\n\n");

	}

	ptr = NULL;
	return headptr;
}
```

11. Delete a node before a specific node in a doubly linked list – Here three error cases and two cases of deletion will be considered.
The three error cases are:
1) The first error case occurs when the linked list is empty.
2) The second error case occurs when the first node is selected. Obviously there will be no node for deletion before the first node.
3) The third error case occurs when a value given by a user is not found in any node of the linked list.

The two cases to delete a node before a specified node are given below.
1) If the value of the second node is given by a user, then obviously the first node of the linked list will be deleted. Here the algorithm of deletion of a node at the beginning of the linked list will be followed.
2) If the value of a node except the second node is selected, then the node just before the specified node will be removed using another procedure.

Algorithm to delete a node before the specified node of a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node before which a node will be deleted

Step 4: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) If ptr → next ≠ NULL and ptr → next → data = Value, then
(Condition of finding Value in 2nd node)
i) Set Head_Pointer = ptr → next
ii) Set Head_Pointer → prev = NULL
Otherwise
iii) Repeat iv) while ptr ≠ NULL and ptr → data ≠ Value
[Loop for traversal of nodes]
iv) Set ptr = ptr → next
v) If ptr = NULL *(Condition of Value not found)*, then
Print " The node before which deletion will be done is not found".
Otherwise
1. If ptr = Head_Pointer *(Condition of finding Value in 1st node)*, then
Print "The deletion of a node before the 1st node is not possible".
Otherwise
2. Set ptr = ptr → prev
Set ptr → prev → next = ptr → next
Set ptr → next → prev = ptr → prev
[End of If-Else (1)]
[End of If-Else (v)]
[End of If-Else (b)]
[End of If-Else (Step 4)]

Step 5: Remove the node pointed by ptr

Step 6: Print “The node has been deleted before the specified node successfully”.

Step 7: Set ptr = NULL

Step 8: Stop

User defined function in C to delete node before the specified node of a doubly linked list

```
struct node *DeleteBeforeNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
		printf("The linked list is empty.\n\n");
	else if(ptr->next != NULL && ptr->next->data == Value)
		headptr = DeleteAtStart(headptr);
	else
	{
		while(ptr != NULL && ptr->data != Value)
			ptr = ptr->next;

		if(ptr == NULL)
			printf("The node before which deletion will be done is not found.\n");
		else if(ptr == headptr)
			printf("The deletion of a node before the first node is not possible\n");

		else
		{
			ptr = ptr->prev;
			ptr->prev->next = ptr->next;
			ptr->next->prev = ptr->prev;

			free(ptr);
			printf("The node has been deleted before the specified node
				successfully.\n\n");
		}
	}

	ptr = NULL;
	return headptr;
}
```

```
struct node *DeleteAtStart(struct node *headptr)
{
        struct node *ptr = headptr;

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else
        {
                headptr = ptr->next;
                if(headptr != NULL)
                        headptr->prev = NULL;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return headptr;
}
```

12. Delete a node after a specific node in a doubly linked list – Here three error cases will be considered.
The three error cases are:
1) The first error case occurs when the linked list is empty.
2) The second error case occurs when the last node is selected. Obviously there will be no node for deletion after the last node.
3) The third error case occurs when a value given by a user is not found in any node of the linked list.

In case of removal of a node after another node, the value of the node after with a node will be deleted is provided by a user. After successive traversal of the nodes, the node whose value is given is pointed by a pointer ptr and the successor node is pointed by the same pointer ptr after one traversal of one node. Now the node pointed by ptr will be removed from the doubly linked list as per the requirement.

Algorithm to delete a node after the specified node of a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node after which a node will be deleted

Step 4: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) Repeat c) while ptr ≠ NULL and ptr → data ≠ Value
Loop for traversal of nodes]
c) Set ptr = ptr → next
d) If ptr = NULL *(Condition of Value not found)*, then
i) Print "The node after which deletion will be done is not found".
Otherwise
ii) If ptr → next = NULL *(Condition of finding Value in last node)*, then
1. Print "The deletion of a node after the last node is not possible".
Otherwise
2. Set ptr = ptr → next
3. Set ptr → prev → next = ptr → next
4. If ptr → next ≠ NULL *(Condition of not being last node)*, then
A. ptr → next → prev = ptr → prev
[End of If (4)]
5. Remove the node pointed by ptr
6. Print " The node has been deleted after the specified node".
[End of If-Else (ii)]
[End of If-Else (d)]
[End of If-Else (Step 4)]

Step 5: Set ptr = NULL

Step 6: Stop

User defined function in C to delete node after the specified node of a doubly linked list

```
struct node *DeleteAfterNode(struct node *headptr,int Value)
{
        struct node *ptr = headptr;

        if(ptr == NULL)
                printf("The linked list is empty.\n\n");
        else
        {
                while(ptr != NULL && ptr->data != Value)
                        ptr = ptr->next;

                if(ptr == NULL)
                        printf("The node after which deletion will be done is not found.\n\n");
                else if(ptr->next == NULL)
                        printf("The deletion of a node after the last node is not possible.\n\n");
```

```
                else
                {
                        ptr = ptr->next;
                        ptr->prev->next = ptr->next;
                        if(ptr->next !=NULL)
                                ptr->next->prev = ptr->prev;

                        free(ptr);
                        printf("The node has been deleted after the specified node
                                successfully.\n\n");
                }
        }
        ptr = NULL;
        return headptr;
}
```

13. Delete a specific node of a doubly linked list – In this case a value is specified by a user and that value is searched among the nodes of the linked list. If the value is found in the data part of any node, that particular node will be removed from the linked list. Here two error cases may occur.
1) When the linked list is completely empty, there will be no node for deletion.
2) When the specified value is not found in the linked list.

Here two cases will happen for deleting a particular node.
1) When first node is selected for deletion, the algorithm of "deletion at the beginning of the linked list" will be applied.
2) When any other node except the first node is chosen for removal, the other procedure is applied.

Algorithm to delete a specific node in a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Value of a node which will be deleted

Step 4: If ptr = NULL *(Condition for empty linked list)*, then
a) Print "The linked list is empty".
Otherwise
b) If ptr → data = Value *[Condition for matching Value in 1st node]*, then
[Procedure to delete a node at the beginning of the linked list will be followed]
i) Set Head_Pointer = ptr → next
ii) Set Head_Pointer → prev = NULL
Otherwise
iii) Repeat iv) while ptr ≠ NULL and ptr → data ≠ Value
[Loop for traversal of nodes]
iv) Set ptr = ptr → next

v) If ptr = NULL *(Condition of the node not found)*, then
1. Print "The specified node which will be deleted is not found.".
Otherwise
2. Set ptr → prev → next = ptr → next
3. If ptr → next ≠ NULL
A. Set ptr → next → prev = ptr → prev
[End of If (3)]
[End of If-Else (v)]
[End of If-Else (b)]
[End of If-Else (Step 4)]

Step 5: Remove the node pointed by ptr

Step 6: Print "The specified node has been deleted successfully.".

Step 7: Set ptr = NULL

Step 8: Stop

User defined function in C to delete a specific node in a doubly linked list

```
struct node *DeleteNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr;
	if(ptr == NULL)
		printf("The linked list is empty.\n\n");
	else if(ptr->data == Value)
		headptr = DeleteAtStart(headptr);
	else
	{
		while(ptr != NULL && ptr->data != Value)
			ptr = ptr->next;

		if(ptr == NULL)
			printf("The specified node which will be deleted is not found.\n\n");
		else
		{
			ptr->prev->next = ptr->next;
			if(ptr->next !=NULL)
				ptr->next->prev = ptr->prev;
			free(ptr);
			printf("The specified node has been deleted successfully.\n\n");
		}
	}
	ptr = NULL;
	return headptr;
}
```

```
struct node *DeleteAtStart(struct node *headptr)
{
        struct node *ptr = headptr;

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else
        {
                headptr = ptr->next;
                if(headptr != NULL)
                        headptr->prev = NULL;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return headptr;
}
```

14. Delete a node at a specific position of a doubly linked list – Here a specific position like 1st, 2nd, 3rd etc is given by a user and the particular node at that given position will be deleted from the linked list.

Two error cases may happen in this case.
1) When the linked list is completely empty.
2) When the given position is not valid for a particular linked list. For example – if 0th position or 5th position is given for a linked list with 4 nodes, then no node is present at the 0th position or at the 5th position. Here these positions will be considered as invalid.

In addition to this, two situations for deletion of a node may occur.
1) If the 1st position is given by a user, then the same algorithm to delete a node at the beginning of the linked list will be followed.
2) If any other position except 1st position is given, then another procedure will be followed.

Algorithm to delete a node at a specific position in a doubly linked list:

Step 1: Start

Step 2: Set ptr = Head_Pointer

Step 3: Input Position where a node will be deleted

Step 4: Count the number of nodes and store that count into the variable NoOfNodes.

Step 5: If ptr = NULL *(Condition for empty linked list)*, then
a) Print “The linked list is empty”.

Otherwise
b) If Position < 1 or Position > NoOfNodes, then
i) Print "The specified position is wrong".
Otherwise
ii) If Position = 1 *[Condition for 1st node selection]*, then
[Procedure to delete a node at the beginning of the linked list will be followed]
1. Set Head_Pointer = ptr → next
2. Set Head_Pointer → prev = NULL
Otherwise
3. Set i = 1
4. Repeat 5. while i < Position
[Loop for traversal of nodes]
5. Set ptr → prev → next = ptr → next
6. If ptr → next ≠ NULL, then
A. Set ptr → next → prev = ptr → prev
[End of If (6)]
[End of If-Else (ii)]
[End of If-Else (b)]
[End of If-Else (Step 5)]

Step 6: Remove the node pointed by ptr

Step 7: Print "The node at the specified position has been deleted successfully".

Step 8: Set ptr = NULL

Step 9: Stop

User defined function in C to delete a node at specific position in a doubly linked list

```
struct node *DeleteAtPos(struct node *headptr,int Position)
{
        struct node *ptr = headptr;
        int i, NoOfNodes;
        NoOfNodes = CountNodes(headptr);
        if(ptr == NULL)
                printf("The linked list is empty.\n\n");
        else if(Position<1 || Position > NoOfNodes)
                printf("The specified position is wrong.\n\n");
        else if(Position == 1)
                headptr = DeleteAtStart(headptr);
        else
        {
                for(i=1; i<Position; i++)
                        ptr = ptr->next;

                ptr->prev->next = ptr->next;
                if(ptr->next !=NULL)
                        ptr->next->prev = ptr->prev;
```

```
                free(ptr);
                printf("The node at the specified position has been deleted successfully.\n\n");
        }

        ptr = NULL;
        return headptr;
}

int CountNodes(struct node *ptr)
{
        int count = 0;

        while(ptr != NULL)
        {
                count++;
                ptr = ptr->next;
        }
        return count;
}

struct node *DeleteAtStart(struct node *headptr)
{
        struct node *ptr = headptr;

        if(ptr == NULL)
        {
                printf("The linked list is empty.\n\n");
        }
        else
        {
                headptr = ptr->next;
                if(headptr != NULL)
                        headptr->prev = NULL;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return headptr;
}
```

In the above discussion fourteen different operations on doubly linked list have been explained with the help of algorithms and the corresponding C functions. All these operations are combined in a single C program and implemented using different user-defined functions which are already given in the previous sections. The fourteen different operations along with the user-defined functions are given below for recapitulation.

SL	Operations applied on doubly linked list	User-defined functions
1	Creating a doubly linked list	CreateNode( )
2	Displaying the data of all nodes in a doubly linked list	DisplayNodes( )
3	Counting the number of nodes in a doubly linked list	CountNodes( )
4	Insert a node at the beginning of a doubly linked list	InsertAtStart( )
5	Insert a node at the end of a doubly linked list	InsertAtLast( )
6	Insert a node before a specific node in a doubly linked list	InsertBeforeNode( )
7	Insert a node after a specific node in a doubly linked list	InsertAfterNode( )
8	Insert a node at a specific position of a doubly linked list	InsertAtPos( )
9	Delete a node at the beginning of a doubly linked list	DeleteAtStart( )
10	Delete a node at the end of a doubly linked list	DeleteAtLast( )
11	Delete a node before a specific node in a doubly linked list	DeleteBeforeNode( )
12	Delete a node after a specific node in a doubly linked list	DeleteAfterNode( )
13	Delete a specific node of a doubly linked list	DeleteNode( )
14	Delete a node at a specific position of a doubly linked list	DeleteAtPos( )

C Program to create a doubly linked list, implement different insertion operations and different deletion operations

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
        struct node *prev;
        int data;
        struct node *next;
};

struct node *CreateNode();
struct node *InsertAtEmpty(int );
struct node *InsertAtEnd(struct node *,int );
void DisplayNode(struct node *);
int CountNodes(struct node *);
struct node *FreeNode(struct node *);
struct node *InsertAtStart(struct node *,int );
struct node *InsertAtLast(struct node *,int );
struct node *InsertBeforeNode(struct node *,int ,int );
struct node *InsertAfterNode(struct node *,int ,int );
struct node *InsertAtPos(struct node *,int ,int );
struct node *DeleteAtStart(struct node *);
struct node *DeleteAtLast(struct node *);
```

```
struct node *DeleteBeforeNode(struct node *,int );
struct node *DeleteAfterNode(struct node *,int );
struct node *DeleteNode(struct node *,int );
struct node *DeleteAtPos(struct node *,int );

int main()
{
	struct node *head = NULL;
	int option, Value, Data, Position, NodeCount;

	while(1)
	{
		printf("0: Exit\n");
		printf("1: Create doubly linked list\n");
		printf("2: Display all nodes of the doubly linked list\n");
		printf("3: Display no. of nodes in the doubly linked list\n");
		printf("4: Insert a node at the starting of the doubly linked list\n");
		printf("5: Insert a node at the end of the doubly linked list\n");
		printf("6: Insert a node before a specific node of the doubly linked list\n");
		printf("7: Insert a node after a specific node of the doubly linked list\n");
		printf("8: Insert a node at a specific position of the doubly linked list\n");
		printf("9: Delete a node at the starting of the doubly linked list\n");
		printf("10: Delete a node at the end of the doubly linked list\n");
		printf("11: Delete a node before a specific node of the doubly linked list\n");
		printf("12: Delete a node after a specific node of the doubly linked list\n");
		printf("13: Delete a specific node of the doubly linked list\n");
		printf("14: Delete a node at a specific position of the doubly linked list\n");
		printf("15: Delete all nodes of the doubly linked list\n");

		printf("\n");
		printf("Enter your option: ");
		scanf("%d", &option);

		switch(option)
		{
			case 0: head = FreeNode(head);
				exit(1);

			case 1: head = CreateNode();
				break;

			case 2: DisplayNode(head);
				break;

			case 3: NodeCount = CountNodes(head);
				printf("No. of nodes in the linked list: %d\n\n", NodeCount);
				break;

			case 4: printf("Enter the data of a new node to be inserted at the starting: ");
				scanf("%d", &Data);
				head = InsertAtStart(head, Data);
				break;
```

```
case 5: printf("Enter the data of a new node to be inserted at the last: ");
        scanf("%d", &Data);
        head = InsertAtLast(head, Data);
        break;

case 6: printf("Enter the value of a node before which a new node will be
                inserted: ");
        scanf("%d", &Value);
        printf("Enter the data of the new node to be inserted before the
                specified node: ");
        scanf("%d", &Data);
        head = InsertBeforeNode(head, Value, Data);
        break;

case 7: printf("Enter the value of a node after which a new node will be
                inserted: ");
        scanf("%d", &Value);
        printf("Enter the data of the new node to be inserted after the
                specified node: ");
        scanf("%d", &Data);
        head = InsertAfterNode(head, Value, Data);
        break;

case 8: printf("Enter the position where a node will be inserted: ");
        scanf("%d", &Position);
        printf("Enter the data of the node to be inserted at the specified
                position: ");
        scanf("%d", &Data);
        head = InsertAtPos(head, Position, Data);
        break;

case 9: head = DeleteAtStart(head);
        break;

case 10: head = DeleteAtLast(head);
        break;

case 11: printf("Enter the value of a node before which a node will be
                 deleted: ");
         scanf("%d", &Value);
         head = DeleteBeforeNode(head, Value);
         break;

case 12: printf("Enter the value of a node after which a node will be deleted ");
         scanf("%d", &Value);
         head = DeleteAfterNode(head, Value);
         break;

case 13: printf("Enter the value of a node which will be deleted: ");
         scanf("%d", &Value);
         head = DeleteNode(head, Value);
         break;
```

```
                        case 14: printf("Enter the position where the node will be deleted: ");
                                 scanf("%d", &Position);
                                 head = DeleteAtPos(head, Position);
                                 break;

                        case 15: head = FreeNode(head);
                                 printf("All nodes of the doubly linked list have been deleted
                                        successfully\n\n");
                                 break;

                        default: printf("Wrong Option\n");
                                 break;
                }
        }

        return 0;
}


struct node *CreateNode()
{
        struct node *headptr = NULL, *ptr;
        int i, n, Data;

        printf("Enter the no. of nodes to be created in the doubly linked list: ");
        scanf("%d", &n);

        for(i=0; i<n; i++)
        {
                printf("Enter Data%d: ", i+1);
                scanf("%d", &Data);

                if(i == 0)
                {
                        headptr = InsertAtEmpty(Data);
                        ptr = headptr;
                }
                else
                {
                        ptr = InsertAtEnd(ptr, Data);
                }
        }

        printf("Doubly Linked list has been created successfully.\n\n");

        ptr = NULL;

        return headptr;
}
```

```
struct node *InsertAtEmpty(int Data)
{
	struct node *headptr;
	headptr = (struct node *)malloc(sizeof(struct node));

	headptr->prev = NULL;
	headptr->data = Data;
	headptr->next = NULL;

	return headptr;
}

struct node *InsertAtEnd(struct node *ptr,int Data)
{
	ptr->next = (struct node *)malloc(sizeof(struct node));
	ptr->next->data = Data;
	ptr->next->prev = ptr;
	ptr->next->next = NULL;

	return ptr->next;
}

void DisplayNode(struct node *ptr)
{
	printf("The Doubly linked list is given below:\n");

	while(ptr != NULL)
	{
		printf("%-5d", ptr->data);
		if(ptr->next == NULL)
			break;
		ptr = ptr->next;
	}

	printf("\n\n");

	printf("The Doubly linked list is given below in reverse order:\n");

	while(ptr != NULL)
	{
		printf("%-5d", ptr->data);
		if(ptr->prev == NULL)
			break;
		ptr = ptr->prev;
	}

	printf("\n\n");

	ptr = NULL;
}
```

```
int CountNodes(struct node *ptr)
{
	int count = 0;

	while(ptr != NULL)
	{
		count++;
		ptr = ptr->next;
	}

	return count;
}

struct node *InsertAtStart(struct node *headptr,int Data)
{
	struct node *ptr;

	ptr = (struct node *)malloc(sizeof(struct node));
	ptr->prev = NULL;
	ptr->data = Data;
	ptr->next = headptr;
	headptr->prev = ptr;
	headptr = ptr;

	printf("A new node has been inserted at the starting successfully.\n\n");

	ptr = NULL;

	return headptr;
}

struct node *InsertAtLast(struct node *headptr,int Data)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		headptr = InsertAtEmpty(Data);
		ptr = headptr;
	}
	else
	{
		while(ptr->next != NULL)
			ptr = ptr->next;

		ptr = InsertAtEnd(ptr, Data);
	}
	printf("A new node has been inserted at the end successfully.\n\n");

	ptr = NULL;
	return headptr;
}
```

```
struct node *InsertBeforeNode(struct node *headptr,int Value,int Data)
{
	struct node *ptr = headptr, *newnode;

	if(ptr->data == Value)
	{
		headptr = InsertAtStart(headptr, Data);
	}
	else
	{
		while(ptr != NULL && ptr->data != Value)
			ptr = ptr->next;

		if(ptr == NULL)
			printf("The node before which insertion will be done is not found.\n\n");
		else
		{
			newnode = (struct node *)malloc(sizeof(struct node));
			newnode->data = Data;
			newnode->next = ptr;
			newnode->prev = ptr->prev;
			ptr->prev = newnode;
			newnode->prev->next = newnode;

			printf("A new node has been inserted before the specified node successfully);
		}
	}

	newnode = NULL;
	ptr = NULL;
	return headptr;
}

struct node *InsertAfterNode(struct node *headptr,int Value,int Data)
{
	struct node *ptr = headptr, *newnode;
	while(ptr != NULL && ptr->data != Value)
		ptr = ptr->next;
	if(ptr == NULL)
		printf("The node after which insertion will be done is not found.\n\n");
	else
	{
		newnode = (struct node *)malloc(sizeof(struct node));
		newnode->data = Data;
		newnode->prev = ptr;
		newnode->next = ptr->next;
		ptr->next = newnode;
		if(newnode->next != NULL)
			newnode->next->prev = newnode;

		printf("A new node has been inserted after the specified node successfully.\n\n");
	}
```

```
	newnode = NULL;
	ptr = NULL;

	return headptr;
}

struct node *InsertAtPos(struct node *headptr,int Position,int Data)
{
	struct node *ptr = headptr, *newnode;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(headptr);

	if(Position < 1 || Position > NoOfNodes + 1)
	{
		printf("The specified position is wrong.\n\n");
	}
	else if(Position == 1)
	{
		headptr = InsertAtStart(headptr, Data);
	}
	else
	{
		for(i=1; i<Position - 1; i++)
			ptr = ptr->next;

		newnode = (struct node *)malloc(sizeof(struct node));
		newnode->data = Data;
		newnode->prev = ptr;
		newnode->next = ptr->next;
		ptr->next = newnode;
		if(newnode->next != NULL)
			newnode->next->prev = newnode;

		printf("A new node has been inserted at a specific position successfully.\n\n");
	}

	newnode = NULL;
	ptr = NULL;

	return headptr;
}


struct node *DeleteAtStart(struct node *headptr)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
```

```
	else
	{
		headptr = ptr->next;
		if(headptr != NULL)
			headptr->prev = NULL;
		free(ptr);
		printf("The node has been deleted at the starting successfully.\n\n");
	}
	ptr = NULL;
	return headptr;
}

struct node *DeleteAtLast(struct node *headptr)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else
	{

		while(ptr->next != NULL)
		{
			ptr = ptr->next;
		}

		if(ptr == headptr)
			headptr = NULL;
		else
			ptr->prev->next = NULL;

		free(ptr);
		printf("The node has been deleted at the end successfully.\n\n");

	}

	ptr = NULL;
	return headptr;
}

struct node *DeleteBeforeNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
```

```
        else if(ptr->next != NULL && ptr->next->data == Value)
        {
                headptr = DeleteAtStart(headptr);
        }
        else
        {
                while(ptr != NULL && ptr->data != Value)
                        ptr = ptr->next;

                if(ptr == NULL)
                        printf("The node before which deletion will be done is not found.\n\n");
                else if(ptr == headptr)
                        printf("The deletion of a node before the first node is not possible.\n\n");

                else
                {
                        ptr = ptr->prev;
                        ptr->prev->next = ptr->next;
                        ptr->next->prev = ptr->prev;

                        free(ptr);
                        printf("The node has been deleted before the specified node successfully");
                }
        }
        ptr = NULL;
        return headptr;
}

struct node *DeleteAfterNode(struct node *headptr,int Value)
{
        struct node *ptr = headptr;
        if(ptr == NULL)
                printf("The linked list is empty.\n\n");
        else
        {
                while(ptr != NULL && ptr->data != Value)
                        ptr = ptr->next;
                if(ptr == NULL)
                        printf("The node after which deletion will be done is not found.\n\n");
                else if(ptr->next == NULL)
                        printf("The deletion of a node after the last node is not possible.\n\n");
                else
                {
                        ptr = ptr->next;
                        ptr->prev->next = ptr->next;
                        if(ptr->next !=NULL)
                                ptr->next->prev = ptr->prev;
                        free(ptr);
                        printf("The node has been deleted after the specified node successfullyn\n");
                }
        }
```

```
	ptr = NULL;
	return headptr;
}

struct node *DeleteNode(struct node *headptr,int Value)
{
	struct node *ptr = headptr;

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
	else if(ptr->data == Value)
	{
		headptr = DeleteAtStart(headptr);
	}
	else
	{
		while(ptr != NULL && ptr->data != Value)
			ptr = ptr->next;

		if(ptr == NULL)
			printf("The specified node which will be deleted is not found.\n\n");
		else
		{
			ptr->prev->next = ptr->next;
			if(ptr->next !=NULL)
				ptr->next->prev = ptr->prev;

			free(ptr);
			printf("The specified node has been deleted successfully.\n\n");
		}

	}

	ptr = NULL;
	return headptr;
}

struct node *DeleteAtPos(struct node *headptr,int Position)
{
	struct node *ptr = headptr;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(headptr);

	if(ptr == NULL)
	{
		printf("The linked list is empty.\n\n");
	}
```

```
        else if(Position<1 || Position > NoOfNodes)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                headptr = DeleteAtStart(headptr);
        }
        else
        {
                for(i=1; i<Position; i++)
                        ptr = ptr->next;

                ptr->prev->next = ptr->next;
                if(ptr->next !=NULL)
                        ptr->next->prev = ptr->prev;

                free(ptr);
                printf("The node at the specified position has been deleted successfully.\n\n");
        }

        ptr = NULL;
        return headptr;
}


struct node *FreeNode(struct node *headptr)
{
        struct node *ptr;

        while(headptr != NULL)
        {
                ptr = headptr->next;
                free(headptr);
                headptr = ptr;
        }

        headptr = NULL;
        ptr = NULL;
        return headptr;
}
```

3) Circular Linked List – A circular linked list is a special type of linked list where first and the last node are somehow connected to each other. There are two types of circular linked list – 1. Circular singly linked list and 2. Circular doubly linked list.

1. Circular Singly Linked List – It is a special type of singly linked list where the last node points to the first node of the linked list. Here every node of the linked list contains two parts which are data part and link part. The data part holds the data and the link part holds the address of the next or successor node. Thus the link of every node becomes the pointer to the next node. As the successor node of the last node is the first node in case of a circular singly linked list, the link of the last node points the first node by holding the address of the first node. The pictorial view of a circular singly linked list is shown in Fig.6.4.

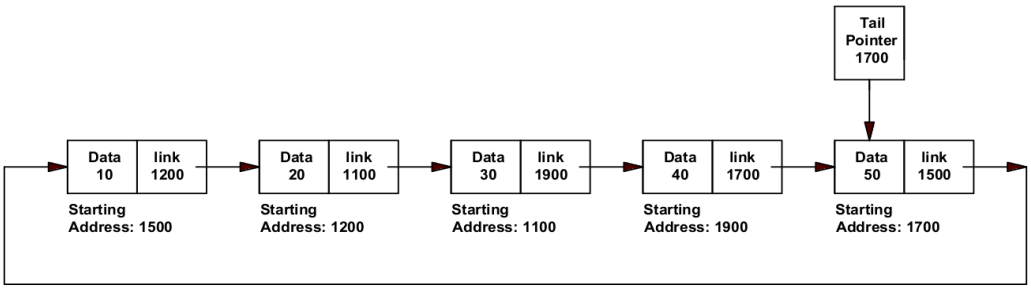


Fig.6.4: Pictorial view of a circular singly linked list

From the above figure it is being seen that the last node is connected to the first node with the help of the link of the last node. Therefore it is possible to traverse from the last node to the first node in a circular singly linked list.

Note: If the word "circular linked list" is specified in anywhere instead of the word "circular singly linked list", it should be considered as circular singly linked list. Therefore the words "circular linked list"and "circular singly linked list" will be used interchangeably in the rest part of this chapter.

Comparative study between singly linked list and circular singly linked list

Differences between singly linked list and circular singly linked list		
SL.	**Singly Linked List**	**Circular Singly linked List**
1.	Last node is not connected to the first node	Last node is connected to the first node
2.	Link part of the last node does not hold the address of the first node. The last node points to NULL to terminate the singly linked list.	Link part of the last node holds the address of the first node i.e. the last node points to the first node in case of a circular singly linked list.
3.	Traversal from last node to first node is not possible.	Traversal from last node to first node is possible.
4.	Singly linked list is accessed by a head pointer which points to the first node of the linked list.	4. Circular singly linked list is accessed by a tail pointer which points to the last node of the linked list.
5.	As the singly linked list is pointed by the head pointer at its first node, traversal of nodes is required to insert a new node at the end of the linked list. So the time complexity is O(n) here for a singly linked list with n no. of nodes.	As the circular singly linked list is pointed by the tail pointer at its last node, traversal of nodes is not required to insert a new node at the end of the linked list. So the time complexity is O(1) here for a circular doubly linked list with n no. of nodes

Similarities between singly linked list and circular singly linked list		
SL.	**Singly Linked List**	**Circular Singly linked List**
1.	It is a linear data structure, because every node has one predecessor and one successor except the first and the last node.	It is a linear data structure as every node has one predecessor and one successor.
2.	Every node has two parts – data part and link part.	Every node has two parts – data part and link part.
3.	Traversal of nodes is possible in one direction.	Traversal of nodes is possible in one direction.
4.	Insertion of a new node at the beginning of the singly linked list is done without any traversal. Therefore the time complexity becomes O(1).	Insertion of a new node at the beginning of the circular singly linked list is done without any traversal. Therefore the time complexity becomes O(1).

Declaring a circular singly linked list – It is being observed that every node of a circular singly linked list has two parts – one is data and other is link or pointer to the next node. This implies that every node of a circular singly linked list is formed with some mixed data types. Therefore node of a circular singly linked list may be constructed only by using structure in C language. In C a node of a circular singly linked list can be implemented by the following code.

```
struct node
{
        int data;
        struct node *link;
};
```

1. Creating a circular singly linked list – Creation of a circular singly linked list means, generation of a specified number of nodes which will be connected to each other in a sequential manner. After the execution of this operation a circular singly linked list with n number of nodes will be generated and the function will return the tail pointer of the linked list. Here the tail pointer is made null for empty circular singly linked list initially. After the creation of the first node the linked list contains only a single node. In this situation the first node is also the last node. That's why the link of this node points to itself as shown in Fig.6.5.

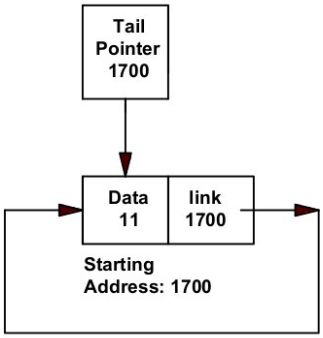


Fig.6.5: Creation of 1st node in a circular singly linked list

Therefore it implies that if the first node holds the address of itself, the circular singly linked list has only a single node in it whereas the singly linked list holds one node when the link of the first node becomes NULL. After the generation of second node the first node and the second node becomes two separate nodes and this time the link part of the last node i.e. the second node holds the address of the first node. Each time a new node will be added at the end of the circular singly linked list and this process will continue till the creation of the nth node. The algorithm to create n number of nodes in a circular singly linked list is given below.

Algorithm of circular singly linked list creation:

Step 1: Start

Step 2: Set Tail_Pointer = NULL

Step 3: Input n for number of nodes to be created

Step 4: Set i = 0

Step 5: Repeat Step 6 to Step 8 while i < n

Step 6: Input Data to be stored into the node

Step 7: If i = 0 *(for first node creation)*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = New_Node_Pointer
Set Tail_Pointer = New_Node_Pointer
Otherwise
b) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer
Set Tail_Pointer = Tail_Pointer → link
[End of If-Else (Step 7)]

Step 8: Set i = i + 1

Step 9: Print "Circular Singly Linked List has been created successfully".

Step 10: Set New_Node_Pointer = NULL

Step 11: Stop

Now the above mentioned algorithm for the creation of a circular singly linked list is explained with the help of the following pictorial representation in Fig.6.6.

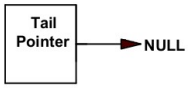
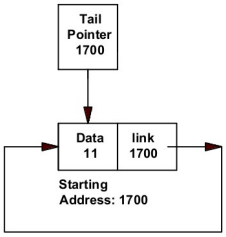
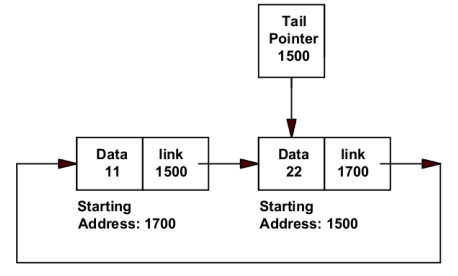
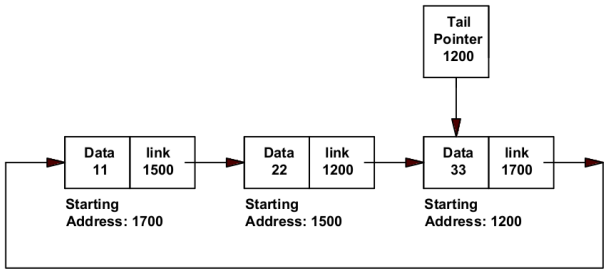
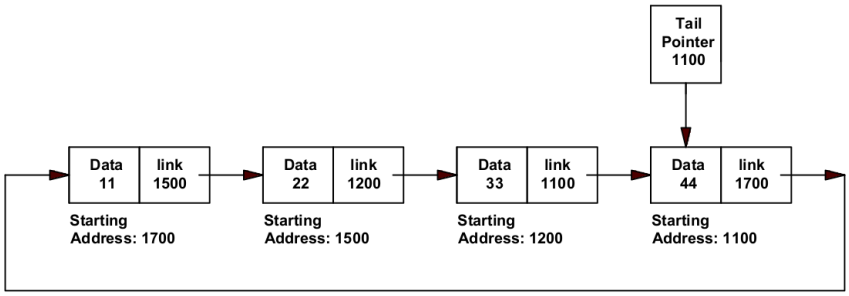
Steps	Status of Circular Singly Linked List
Initially	 Empty circular singly linked list
i = 0 Data = 11	 Circular singly linked list with 1 node
i = 1 Data = 22	 Circular singly linked list with 2 nodes
i = 2 Data = 33	 Circular singly linked list with 3 nodes
i = 3 Data = 44	 Circular singly linked list with 4 nodes

Fig.6.6: Pictorial representation of circular singly linked list creation

User defined function in C to create a circular singly linked list with n no. of nodes

```
struct node *CreateNode()
{
	struct node *tailptr = NULL;
	int i, n, Data;

	printf("Enter the no. of nodes to be created in the circular linked list: ");
	scanf("%d", &n);

	for(i=0; i<n; i++)
	{
		printf("Enter Data%d: ", i+1);
		scanf("%d", &Data);

		if(i == 0)
			tailptr = InsertAtEmpty(Data);
		else
			tailptr = InsertAtEnd(tailptr, Data);
	}
	printf("circular Linked list has been created successfully.\n\n");
	return tailptr;
}

struct node *InsertAtEmpty(int Data)
{
	struct node *tailptr;
	tailptr = (struct node *)malloc(sizeof(struct node));

	tailptr->data = Data;
	tailptr->link = tailptr;

	return tailptr;
}

struct node *InsertAtEnd(struct node *tailptr,int Data)
{
	struct node *ptr;
	ptr = (struct node *)malloc(sizeof(struct node));
	ptr->data = Data;
	ptr->link = tailptr->link;
	tailptr->link = ptr;

	ptr = NULL;
	return tailptr->link;
}
```

2. Displaying the data of all nodes in a circular singly linked list – This operation displays/ prints the data of all nodes in a circular singly linked list in forward direction i.e. from starting node to the last node. In this case a temporary pointer ptr is taken to point the starting node or first node initially. It is required to traverse all the nodes from first node to the last node and during this traversal the data of every node is displayed on the screen.

Hence the traversal of nodes should start from the first node and continue up to the last node of the circular singly linked list. Therefore the last node should be detected to stop the traversal process. As the last node is pointed by the tail pointer, any node in a circular singly linked list will be detected as the last node if its link becomes Tail_Pointer → link. But to accomplish this task we have to face a problem. When ptr is used to point the first node of a circular singly linked list, it will be equal to 'Tail_Pointer → link' as the first node is also pointed by the pointer 'Tail_Pointer → link' in case of a circular singly linked list. If 'while' loop is used with the condition "ptr ≠ Tail_Pointer → link" to traverse the nodes of a circular singly linked list, the loop will not be executed anymore as the condition "ptr = Tail_Pointer → link" will be satisfied for the first node in a circular linked list. To solve this problem a do-while loop is used instead of a while loop for the traversal of the nodes in a circular linked list. In case of a do-while loop the statement "ptr = ptr → link" inside the loop are executed first and then the condition " ptr ≠ Tail_Pointer → link" is checked. This time the loop will be iterated as usual, because the ptr will point to the second node instead of the first node during the condition checking of the do-while loop. The structure of the do-while loop for display function of a circular singly linked list is shown below.

```
do
{
        printf("%d ", ptr→data);
        ptr = ptr → link;
}while(ptr != tailptr->link);
```

Algorithm of displaying the data of all nodes in a circular singly linked list:

Step 1: Start

Step 2: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
Print "Circular Linked List is empty".
Go to Step 8
[End of If (Step 2)]

Step 3: Set ptr = Tail_Pointer → link

Step 4: Print ptr → data

Step 5: Set ptr = ptr → link

Step 6: Repeat Step 4 to Step 5 while ptr ≠ Tail_Pointer → link

Step 7: Set ptr = NULL

Step 8: Stop

User defined function in C to display the data of all nodes in a circular singly linked list

```
void DisplayNodes(struct node *tailptr)
{
        struct node *ptr;
        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
                return;
        }

        ptr = tailptr->link;
        printf("The circular linked list is given below:\n");

        do
        {
                printf("%-5d", ptr->data);
                ptr = ptr->link;
        }while(ptr != tailptr->link);

        printf("\n\n");

        ptr = NULL;
}
```

3. Counting the number of nodes in a circular singly linked list – This operation counts the number of nodes in a circular singly linked list. Here a variable 'count' is initialized to zero and incremented by one each time the pointer ptr traverses the successive nodes in the linked list.

Algorithm to count the number of nodes in a circular singly linked list:

Step 1: Start

Step 2: Set count = 0

Step 3: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
Print "Circular Linked List is empty".
Go to Step 10
[End of If (Step 3)]

Step 4: Set ptr = Tail_Pointer → link

Step 5: Set count = count + 1

Step 6: Set ptr = ptr → link

Step 7: Repeat Step 5 to Step 6 while ptr ≠ Tail_Pointer → link

Step 8: Print count to display the number of nodes

Step 9: Set ptr = NULL

Step 10: Stop

User defined function in C to count the number of nodes in a circular singly linked list

```
int CountNodes(struct node *tailptr)
{
        int count = 0;
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
                return count;
        }

        ptr = tailptr->link;

        do
        {
                count++;
                ptr = ptr->link;
        }while(ptr != tailptr->link);

        ptr = NULL;

        return count;
}
```

4. Insert a node at the beginning of a circular singly linked list – Here a new node will be inserted at the beginning of a circular singly linked list. After the insertion the new node becomes the starting node or the first node of the linked list.

Algorithm to insert a node at the beginning of a circular singly linked list:

Step 1: Start

Step 2: Input Data to be stored into the new node

Step 3: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = New_Node_Pointer
Set Tail_Pointer = New_Node_Pointer
Otherwise
b) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer
[End of If -Else (Step 3)]

Step 4: Print "A new node has been inserted at the starting successfully".

Step 5: New_Node_Pointer = NULL

Step 6: Stop

User defined function in C to insert a node at the beginning of a circular singly linked list

```
struct node *InsertAtStart(struct node *tailptr,int Data)
{
        struct node *ptr;

        if(tailptr == NULL)
                tailptr = InsertAtEmpty(Data);
        else
        {
                ptr = (struct node *)malloc(sizeof(struct node));
                ptr->data = Data;
                ptr->link = tailptr->link;
                tailptr->link = ptr;
        }
        printf("A new node has been inserted at the starting successfully.\n\n");
        ptr = NULL;
        return tailptr;
}
```

```
struct node *InsertAtEmpty(int Data)
{
        struct node *tailptr;
        tailptr = (struct node *)malloc(sizeof(struct node));

        tailptr->data = Data;
        tailptr->link = tailptr;

        return tailptr;
}
```

5. Insert a node at the end of a circular singly linked list – This operation inserts a newly created node at the end of a circular singly linked list i.e. it add the new node after the last node of the existing linked list. After the insertion at the end the new node becomes the last node of the linked list and that's why the new node must be pointed by Tail pointer.

Algorithm to insert a node at the end of a circular singly linked list:

Step 1: Start

Step 2: Input Data to be stored into the new node

Step 3: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = New_Node_Pointer
Set Tail_Pointer = New_Node_Pointer
Otherwise
b) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer;
Set Tail_Pointer = New_Node_Pointer
[End of If-Else (Step 3)]

Step 4: Print "A new node has been inserted at the end successfully".

Step 5: Set New_Node_Pointer = NULL

Step 6: Stop

In the above algorithm two situations during the insertion of a node at the end has been considered – one situation when the circular singly linked list is empty and other situation when the circular linked list is not empty. When the circular singly linked list is empty, the procedure same as the procedure to insert the first node into the empty linked list is followed. But the important point to observe here that traversal of nodes is not required during the insertion of a new node at the end of a non-empty circular singly linked list whereas traversal is mandatory for the insertion of a node at the end of a singly linked list.

User defined function in C to insert a node at the end of a circular singly linked list

```
struct node *InsertAtLast(struct node *tailptr,int Data)
{
        struct node *ptr;

        if(tailptr == NULL)
                tailptr = InsertAtEmpty(Data);
        else
        {
                ptr = (struct node *)malloc(sizeof(struct node));
                ptr->data = Data;
                ptr->link = tailptr->link;
                tailptr->link = ptr;
        }
        printf("A new node has been inserted at the end successfully.\n\n");

        ptr = NULL;
        return tailptr->link;
}
struct node *InsertAtEmpty(int Data)
{
        struct node *tailptr;
        tailptr = (struct node *)malloc(sizeof(struct node));

        tailptr->data = Data;
        tailptr->link = tailptr;

        return tailptr;
}
```

6. Insert a node before a specific node in a circular singly linked list – This function will insert a new node just before another node which is specified by the user. Here we have to consider two situations.

In first situation the specified node is the first node in the linked list. So the insertion of a new node before the first node is basically the insertion of a new node at the beginning of the link list, which has been already discussed previously in this chapter. In second situation when the specified node is not first node, we have to insert the newly created node in between the specified node and the node just before the specified node. Except these two situations, there are another two error situations where the circular linked list is empty and the specified node is not found in the linked list. In these cases an error message will be printed.

Algorithm of inserting a node before a specific node in a circular singly linked list:

Step 1: Start

Step 2: Input Value of a node before which a new node will be inserted

Step 3: Input Data to be stored into the new node

Step 4: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
Print "Circular Linked List is empty".
Go to Step 8
[End of If (Step 4)]

Step 5: If Tail_Pointer → link → data = Value *[Condition of finding Value in 1st node]*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer
Otherwise
b) Set ptr = Tail_Pointer → link
c) Repeat (d) while ptr → link → data ≠ Value and ptr ≠ Tail_Pointer
d) Set ptr = ptr → link

e) If ptr = Tail_Pointer, then
i) Print "The node before which insertion will be done is not found".
Otherwise
ii) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = ptr → link
Set ptr → link = New_Node_Pointer
Print "A new node has been inserted before the specified node".
[End of If-Else (e)]
[End of If-Else (Step 5)]

Step 6: Set ptr = NULL

Step 7: Set New_Node_Pointer = NULL

Step 8: Stop

User defined function in C to insert a node before a specific node in a circular singly linked list

```
struct node *InsertBeforeNode(struct node *tailptr,int Value,int Data)
{
	struct node *ptr, *newnodeptr;


	if(tailptr == NULL)
	{
		printf("Circular lined list is empty\n\n");
		return tailptr;
	}
	else if(tailptr->link->data == Value)
		tailptr = InsertAtStart(tailptr, Data);
	else
	{
		ptr = tailptr->link;
		while(ptr->link->data != Value && ptr != tailptr)
			ptr = ptr->link;

		if(ptr == tailptr)
			printf("The node before which insertion will be done is not found\n");
		else
		{
			newnodeptr = (struct node *)malloc(sizeof(struct node));
			newnodeptr->data = Data;
			newnodeptr->link = ptr->link;
			ptr->link = newnodeptr;
			printf("A new node has been inserted before the specified node
				successfully.\n\n");
		}
	}
	newnodeptr = NULL;
	ptr = NULL;
	return tailptr;
}

struct node *InsertAtStart(struct node *tailptr,int Data)
{
	struct node *ptr;
	if(tailptr == NULL)
		tailptr = InsertAtEmpty(Data);
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->link = tailptr->link;
```

```
                tailptr->link = ptr;
        }
        printf("A new node has been inserted at the starting successfully.\n\n");

        ptr = NULL;
        return tailptr;
}

struct node *InsertAtEmpty(int Data)
{
        struct node *tailptr;
        tailptr = (struct node *)malloc(sizeof(struct node));
        tailptr->data = Data;
        tailptr->link = tailptr;
        return tailptr;
}
```

7. Insert a node after a specific node in a circular singly linked list – This operation will insert a new node just after another node which is specified by the user. Here at first the node with the specified value is searched through the traversal of the linked list and if the specified node is found, a new node with entered data will be inserted after it, otherwise an error message will be displayed.

Algorithm of inserting a node after a specific node in a circular singly linked list:

Step 1: Start

Step 2: Input Value of a node after which a new node will be inserted

Step 3: Input Data to be stored into the new node

Step 4: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
Print "Circular Linked List is empty".
Go to Step 8
[End of If (Step 4)]

Step 5: If Tail_Pointer → data = Value *(Condition of finding Value in last node)*, then
a) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer;
Set Tail_Pointer = New_Node_Pointer
Otherwise
b) Set ptr = Tail_Pointer → link
c) Repeat (d) while ptr → data ≠ Value and ptr ≠ Tail_Pointer
d) Set ptr = ptr → link
e) If ptr = Tail_Pointer, then
i) Print "The node after which insertion will be done is not found".

Otherwise
ii) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = ptr → link
Set ptr → link = New_Node_Pointer
Print "A new node has been inserted after the specified node".
[End of If-Else (e)]
[End of If-Else (Step 5)]

Step 6: Set ptr = NULL

Step 7: Set New_Node_Pointer = NULL

Step 8: Stop

User defined function in C to insert a node after a specific node in a circular singly linked list

```
struct node *InsertAfterNode(struct node *tailptr,int Value,int Data)
{
        struct node *ptr, *newnodeptr;
        if(tailptr == NULL)
        {
                printf("Circular lined list is empty\n\n");
                return tailptr;
        }
        else if(tailptr->data == Value)
                tailptr = InsertAtLast(tailptr, Data);
        else
        {
                ptr = tailptr->link;
                while(ptr->data != Value && ptr != tailptr)
                        ptr = ptr->link;
                if(ptr == tailptr)
                        printf("The node after which insertion will be done is not found.\n\n");
                else
                {
                        newnodeptr = (struct node *)malloc(sizeof(struct node));
                        newnodeptr->data = Data;
                        newnodeptr->link = ptr->link;
                        ptr->link = newnodeptr;
                        printf("A new node has been inserted after the specified node.");
                }
        }
        newnodeptr = NULL;
        ptr = NULL;
        return tailptr;
}
```

```
struct node *InsertAtLast(struct node *tailptr,int Data)
{
        struct node *ptr;
        if(tailptr == NULL)
                tailptr = InsertAtEmpty(Data);
        else
        {
                ptr = (struct node *)malloc(sizeof(struct node));
                ptr->data = Data;
                ptr->link = tailptr->link;
                tailptr->link = ptr;
        }
        printf("A new node has been inserted at the end successfully.\n\n");

        ptr = NULL;
        return tailptr->link;
}

struct node *InsertAtEmpty(int Data)
{
        struct node *tailptr;
        tailptr = (struct node *)malloc(sizeof(struct node));

        tailptr->data = Data;
        tailptr->link = tailptr;

        return tailptr;
}
```

8. Insert a node at a specific position of a circular singly linked list – This function inserts a new node with an entered data at a specified position of a circular singly linked list. In this case the remaining part of the linked list starting from the specific position will be shifted right and the new node will be placed at the specified position. For example – if the specified position is 3rd in a linked list of three nodes, then the new node will be placed at 3rd position and the 3rd node of the existing linked list will be shifted right one position to become 4th node.

Algorithm of inserting a node at a specific position of a circular singly linked list:

Step 1: Start

Step 2: Input Position where a new node will be inserted

Step 3: Input Data to be stored into the new node

Step 4: Set NoOfNodes = Total number of nodes in the linked list

Step 5: If Position < 1 or Position > NoOfNodes + 1, then
a) Print "Entered position is Invalid".
Otherwise
b) If Position = 1 *(Condition of 1st node selection)*, then
i) Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer
Otherwise
ii) If Position = NoOfNodes + 1 *(Condition of last node selection)*, then
Create a new node pointed by New_Node_Pointer
Set New_Node_Pointer → data = Data
Set New_Node_Pointer → link = Tail_Pointer → link
Set Tail_Pointer → link = New_Node_Pointer
Set Tail_Pointer = New_Node_Pointer
Otherwise
iii) Set i = 1
iv) Set ptr = Tail_Pointer → link
v) Repeat vi) while i < Position – 1
vi) Set ptr = ptr → link

vii) Create a new node pointed by New_Node_Pointer
viii) Set New_Node_Pointer → data = Data
ix) Set New_Node_Pointer → link = ptr → link
x) Set ptr → link = New_Node_Pointer
xi) Print "A new node has been inserted at a specific position of the linked list".
[End of If-Else (ii)]
[End of If-Else (b)]
[End of If-Else (Step 5)]

Step 6: Set ptr = NULL

Step 7: Set New_Node_Pointer = NULL

Step 8: Stop

User defined function in C to insert a node at a specific position in a circular singly linked list

```
struct node *InsertAtPos(struct node *tailptr,int Position,int Data)
{
	struct node *ptr, *newnodeptr;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(tailptr);
	if(Position < 1 || Position > NoOfNodes + 1)
		printf("The specified position is wrong.\n\n");
	else if(Position == 1)
		tailptr = InsertAtStart(tailptr, Data);
	else if(Position == NoOfNodes + 1)
		tailptr = InsertAtLast(tailptr, Data);
	else
	{
		ptr = tailptr->link;
		for(i=1; i<Position - 1; i++)
			ptr = ptr->link;

		newnodeptr = (struct node *)malloc(sizeof(struct node));
		newnodeptr->data = Data;
		newnodeptr->link = ptr->link;
		ptr->link = newnodeptr;
		printf("A new node has been inserted at a specific position successfully.\n");
	}
	newnodeptr = NULL;
	ptr = NULL;
	return tailptr;
}

struct node *InsertAtStart(struct node *tailptr,int Data)
{
	struct node *ptr;
	if(tailptr == NULL)
		tailptr = InsertAtEmpty(Data);
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->link = tailptr->link;
		tailptr->link = ptr;
	}
	printf("A new node has been inserted at the starting successfully.\n\n");
	ptr = NULL;
	return tailptr;
}
```

```
struct node *InsertAtLast(struct node *tailptr,int Data)
{
	struct node *ptr;

	if(tailptr == NULL)
		tailptr = InsertAtEmpty(Data);
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->link = tailptr->link;
		tailptr->link = ptr;
	}
	printf("A new node has been inserted at the end successfully.\n\n");

	ptr = NULL;
	return tailptr->link;
}

struct node *InsertAtEmpty(int Data)
{
	struct node *tailptr;
	tailptr = (struct node *)malloc(sizeof(struct node));

	tailptr->data = Data;
	tailptr->link = tailptr;
	return tailptr;
}
```

9. Delete a node at the beginning of a circular singly linked list – This operation deletes a node at the beginning of a circular singly linked list. Here three situations have been considered.
1st situation: When the linked list is empty, an error message will be displayed.
2nd situation: When the linked list holds a single node, there will be no node left after the deletion and the Tail_Pointer is made NULL.
3rd situation: When the circular linked list consists of multiple node, the node at the beginning of the list will be removed.

Algorithm to delete a node at the beginning of a circular singly linked list:

Step 1: Start

Step 2: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Print "Circular Linked List is empty".
Otherwise
b) If Tail_Pointer = Tail_Pointer → link *(Condition for one node)*, then
Set ptr = Tail_Pointer
Remove the node pointed by ptr
Tail_Pointer = NULL
Print "The node has been deleted at the starting of the linked list"
Otherwise
c) Set ptr = Tail_Pointer → link
Set Tail_Pointer → link = ptr → link
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
[End of If-Else (b)]
[End of If-Else (Step 2)]

Step 3: Set ptr = NULL

Step 4: Stop

User defined function in C to delete a node at the beginning of a circular singly linked list

```
struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;
        if(tailptr == NULL)
                printf("Circular linked list is empty.\n\n");
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
```

```
        else
        {
                ptr = tailptr->link;
                tailptr->link = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

10. Delete a node at the end of a circular singly linked list – In this operation the node at the end of a circular singly linked list will be removed. Here three situations have been considered.
1st situation: When the linked list is empty, an error message will be displayed.
2nd situation: When the linked list holds a single node, there will be no node left after the deletion and the Tail_Pointer is made NULL.
3rd situation: When the circular linked list consists of multiple node, the node at the last of the list will be removed and Tail_Pointer is shifted to point the previous node.

Algorithm to delete a node at the end of a circular singly linked list:

Step 1: Start

Step 2: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Print "Circular Linked List is empty".
Otherwise
b) If Tail_Pointer = Tail_Pointer → link *(Condition for one node)*, then
Set ptr = Tail_Pointer
Remove the node pointed by ptr
Tail_Pointer = NULL
Print "The node has been deleted at the starting of the linked list"
Otherwise
c) Set ptr = Tail_Pointer → link
d) Repeat e) while ptr → link ≠ Tail_Pointer
e) Set ptr = ptr → link
f) Set ptr → link = Tail_Pointer → link
g) Remove the node pointed by Tail_Pointer
h) Set Tail_Poimter = ptr
i) Print "The node has been deleted at the end of the linked list"
[End of If-Else (b)]
[End of If-Else (Step 2)]

Step 3: Set ptr = NULL

Step 4: Stop

User defined function in C to delete a node at the end of a circular singly linked list

```
struct node *DeleteAtLast(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
                printf("Circular linked list is empty.\n\n");
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->link;
                while(ptr->link != tailptr)
                        ptr = ptr->link;

                ptr->link = tailptr->link;
                free(tailptr);
                tailptr = ptr;
                printf("The node has been deleted at the end successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

11. Delete a node before a specific node in a circular singly linked list – Here two error cases and two cases of deletion will be considered.
The two error cases are:
1) The first error case occurs when the linked list is empty.
2) The second error case occurs when a value given by a user is not found in the circular singly linked list.

The two cases to delete a node before a specified node are given below.
1) If the value of the second node is given by a user, then obviously the first node of the linked list will be deleted. Here the algorithm of deletion of a node at the beginning of the linked list will be followed.
2) If the value of a node except the second node is selected, then the node just before the specified node will be removed. Here the searching of the given value is started from 3rd node and continue up to 1st node of the circular singly linked list. If the value is found in the 1st node, the node before the 1st node i.e. the last node will be deleted in case of circular linked list whereas the deletion is not possible for such situation in case of a singly linked list.

Algorithm to delete a node before the specified node of a circular singly linked list:

Step 1: Start

Step 2: Input Value of a node before which a node will be deleted

Step 3: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Print "Circular Linked List is empty".
Otherwise
b) If Tail_Pointer → link → link → data = Value *(Condition for 2nd node selection)*, then
Set ptr = Tail_Pointer → link
Set Tail_Pointer → link = ptr → link
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
Otherwise
c) Set ptr = Tail_Pointer → link
d) Set prevptr = Tail_Pointer
e) Set prevptr = ptr
f) Set ptr = ptr → link
g) Repeat e) and f) while ptr → link → data ≠ Value and ptr ≠ Tail_Pointer → link
i) If ptr = Tail_Pointer → link, then
Printf "The node before which deletion will be done is not found".
Otherwise
ii) Set prevptr → link = ptr → link
1. If ptr = Tail_Pointer *(Condition of deletion of last node)*, then
Set Tail_Pointer = prevptr
[End of If (1)]
iii) Remove the node pointed by ptr
iv) Print "The node has been deleted before the specified node".
[End of If-Else (i)]
[End of If-Else (b)]
[End of If-Else (Step 3)]

Step 4: Set prevptr = NULL

Step 5: Set ptr = NULL

Step 6: Stop

In the above algorithm the pointer prevptr is used to point the predecessor node of the node pointed by the pointer ptr.

User defined function in C to delete a node before a specific node in a circular singly linked list

```
struct node *DeleteBeforeNode(struct node *tailptr,int Value)
{
	struct node *ptr, *prevptr;

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
	else if(tailptr->link->link->data == Value)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->link;
		prevptr = tailptr;

		do
		{
			prevptr = ptr;
			ptr = ptr->link;
		}while(ptr->link->data != Value && ptr != tailptr->link);

		if(ptr == tailptr->link)
			printf("The node before which deletion will be done is not found\n");

		else
		{
			prevptr->link = ptr->link;
			if(ptr == tailptr)
				tailptr = prevptr;
			free(ptr);
			printf("The node has been deleted before the specified node
				successfully.\n\n");
		}

	}

	prevptr = NULL;
	ptr = NULL;
	return tailptr;
}
```

```
struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->link;
                tailptr->link = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

12. Delete a node after a specific node in a circular singly linked list – Here two error cases will be considered.
The two error cases are:
1) The first error case occurs when the circular singly linked list is empty.
2) The second error case occurs when a value given by a user is not found in any node of the circular linked list.

The two cases to delete a node after a specified node are given below.
1) If the value of the last node is given by a user, then obviously the first node (the node after the last node for a circular linked list) of the linked list will be deleted. Here the algorithm of deletion of a node at the beginning of the linked list will be followed.
2) If the value of a node except the last node is selected, then the node just after the specified node will be removed. Here the searching of the given value is started from 1st node and continue up to the last but one node of the circular singly linked list. If the value is found in the last node, the node after the last node i.e. 1st node node will be deleted in case of circular linked list whereas the deletion is not possible for such situation in case of a singly linked list.

Algorithm to delete a node after the specified node of a circular singly linked list:

Step 1: Start

Step 2: Input Value of a node before which a node will be deleted

Step 3: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Print "Circular Linked List is empty".
Otherwise
b) If Tail_Pointer → data = Value *(Condition for last node selection)*, then
Set ptr = Tail_Pointer → link
Set Tail_Pointer → link = ptr → link
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
Otherwise
c) Set ptr = Tail_Pointer → link
d) Set prevptr = Tail_Pointer
e) Set prevptr = ptr
f) Set ptr = ptr → link
g) Repeat e) and f) while prevptr → data ≠ Value and ptr ≠ Tail_Pointer → link
i) If ptr = Tail_Pointer → link, then
Printf "The node after which deletion will be done is not found".
Otherwise
ii) Set prevptr → link = ptr → link
1. If ptr = Tail_Pointer *(Condition of deletion of last node)*, then
Set Tail_Pointer = prevptr
[End of If (1)]
iii) Remove the node pointed by ptr
iv) Print "The node has been deleted after the specified node".
[End of If-Else (i)]
[End of If-Else (b)]
[End of If-Else (Step 3)]

Step 4: Set prevptr = NULL

Step 5: Set ptr = NULL

Step 6: Stop

In the above algorithm the pointer prevptr is used to point the predecessor node of the node pointed by the pointer ptr.

User defined function in C to delete a node after a specific node in a circular singly linked list

```
struct node *DeleteAfterNode(struct node *tailptr,int Value)
{
	struct node *ptr, *prevptr;

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
	else if(tailptr->data == Value)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->link;
		prevptr = tailptr;

		do
		{
			prevptr = ptr;
			ptr = ptr->link;
		}while(prevptr->data != Value && ptr != tailptr->link);

		if(ptr == tailptr->link)
			printf("The node after which deletion will be done is not found.\n\n");

		else
		{
			prevptr->link = ptr->link;
			if(ptr == tailptr)
				tailptr = prevptr;
			free(ptr);
			printf("The node has been deleted after the specified node.\n\n");
		}
	}

	prevptr = NULL;
	ptr = NULL;
	return tailptr;
}
```

```
struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->link;
                tailptr->link = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

13. Delete a specific node of a circular singly linked list – In this case a value is specified by a user and that value is searched among the nodes of the linked list. If the value is found in the data part of any node, that particular node will be removed from the linked list. Here two error cases may occur.
1) When the circular linked list is completely empty, there will be no node for deletion.
2) When the specified value is not found in the data part of any node of the linked list.

Here two cases will happen for deleting a particular node.
1) When first node is selected for deletion, the algorithm of “deletion at the beginning of the linked list” is applied.
2) When any other node except the first node is chosen for removal, the other procedure is applied.

Algorithm to delete a specific node of a circular singly linked list:

Step 1: Start

Step 2: Input Value of a node before which a node will be deleted

Step 3: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Print "Circular Linked List is empty".
Otherwise
b) If Tail_Pointer → link → data = Value *(Condition for 1st node selection)*, then
Set ptr = Tail_Pointer → link
Set Tail_Pointer → link = ptr → link
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
Otherwise
c) Set ptr = Tail_Pointer → link
d) Set prevptr = Tail_Pointer
e) Set prevptr = ptr
f) Set ptr = ptr → link
g) Repeat e) and f) while ptr → data ≠ Value and ptr ≠ Tail_Pointer → link
i) If ptr = Tail_Pointer → link, then
Printf "The specified node which will be deleted is not found".
Otherwise
ii) Set prevptr → link = ptr → link
1. If ptr = Tail_Pointer *(Condition of deletion of last node)*, then
Set Tail_Pointer = prevptr
[End of If (1)]
iii) Remove the node pointed by ptr
iv) Print "The specified node has been deleted successfully".
[End of If-Else (i)]
[End of If-Else (b)]
[End of If-Else (Step 3)]

Step 4: Set prevptr = NULL

Step 5: Set ptr = NULL

Step 6: Stop

In the above algorithm the pointer prevptr is used to point the predecessor node of the node pointed by the pointer ptr.

User defined function in C to delete a specific node in a circular singly linked list

```
struct node *DeleteNode(struct node *tailptr,int Value)
{
	struct node *ptr, *prevptr;

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
	else if(tailptr->link->data == Value)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->link;
		prevptr = tailptr;

		do
		{
			prevptr = ptr;
			ptr = ptr->link;
		}while(ptr->data != Value && ptr != tailptr->link);

		if(ptr == tailptr->link)
			printf("The specified node which will be deleted is not found.\n\n");
		else
		{
			prevptr->link = ptr->link;
			if(ptr == tailptr)
				tailptr = prevptr;
			free(ptr);
			printf("The specified node has been deleted successfully.\n\n");
		}

	}

	prevptr = NULL;
	ptr = NULL;
	return tailptr;
}
```

```
struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->link;
                tailptr->link = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

14. Delete a node at a specific position of a circular singly linked list – Here a specific position like 1st, 2nd, 3rd etc is given by a user and the particular node at that given position will be deleted from the circular linked list.

Two error cases may happen in this case.
1) When the linked list is completely empty.
2) When the given position is not valid for a particular linked list. For example – if 0th position or 5th position is given for a linked list with 4 nodes, then no node is present at the 0th position or at the 5th position. Here these positions will be considered as invalid.

In addition to this, two situations for deletion of a node may occur.
1) If the 1st position is given by a user, then the same algorithm to delete a node at the beginning of the linked list will be followed.
2) If any other position except 1st position is given, then usual procedure will be followed.

Algorithm to delete a node at a specific position in a singly linked list:

Step 1: Start

Step 2: Input Position where a node will be deleted

Step 3: Count the number of nodes and store that count into the variable NoOfNodes.

Step 4: If Tail_Pointer = NULL *(Condition for empty linked list)*, then
a) Print "The circular linked list is empty".
Otherwise
b) If Position < 1 or Position > NoOfNodes, then
i) Print "The specified position is wrong".
Otherwise
ii) If Position = 1 *[Condition for 1st node selection]*, then
1. Set ptr = Tail_Pointer → link
Set Tail_Pointer → link = ptr → link
Remove the node pointed by ptr
Print "The node has been deleted at the starting of the linked list"
Otherwise
2. Set ptr = Tail_Pointer → link
3. Set prevptr = Tail_Pointer
4. Set i = 1
5. Repeat 6. and 7. while i < Position
6. Set prevptr = ptr
7. Set ptr = ptr → link
8. Set prevptr → link = ptr → link
A. If ptr = Tail_Pointer
Tail_Pointer = prevptr
[End of If (A)]
9. Remove the node pointed by ptr
10. Print "The node at the specified position has been deleted successfully".
[End of If-Else (ii)]
[End of If-Else (b)]
[End of If-Else (Step 4)]

Step 5: Set prevptr = NULL

Step 6: Set ptr = NULL

Step 7: Stop

User defined function in C to delete a node at a specific position in a circular singly linked list

```
struct node *DeleteAtPos(struct node *tailptr,int Position)
{
	struct node *ptr, *prevptr;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(tailptr);

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
	else if(Position<1 || Position > NoOfNodes)
	{
		printf("The specified position is wrong.\n\n");
	}
	else if(Position == 1)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->link;
		prevptr = tailptr;

		for(i=1; i<Position; i++)
		{
			prevptr = ptr;
			ptr = ptr->link;
		}

		prevptr->link = ptr->link;
		if(ptr == tailptr)
			tailptr = prevptr;
		free(ptr);
		printf("The node at the specified position has been deleted successfully.\n\n");
	}

	prevptr = NULL;
	ptr = NULL;
	return tailptr;
}
```

```
struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->link;
                tailptr->link = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

In the above discussion fourteen different operations on circular singly linked list have been explained with the help of algorithms. All these operations are combined in a single C program and implemented using different user-defined functions which are already given in the previous sections. The fourteen different operations along with the user-defined functions are given below for recapitulation.

SL	Operations applied on circular singly linked list	User-defined functions
1	Creating a circular singly linked list	CreateNode( )
2	Displaying the data of all nodes in a circular singly linked list	DisplayNodes( )
3	Counting the number of nodes in a circular singly linked list	CountNodes( )
4	Insert a node at the beginning of a circular singly linked list	InsertAtStart( )
5	Insert a node at the end of a circular singly linked list	InsertAtLast( )
6	Insert a node before a specific node in circular singly linked list	InsertBeforeNode( )
7	Insert a node after a specific node in a circular singly linked list	InsertAfterNode( )
8	Insert a node at a specific position of a circular singly linked list	InsertAtPos( )
9	Delete a node at the beginning of a circular singly linked list	DeleteAtStart( )
10	Delete a node at the end of a circular singly linked list	DeleteAtLast( )

SL	Operations applied on circular singly linked list	User-defined functions
11	Delete a node before a specific node in circular singly linked list	DeleteBeforeNode( )
12	Delete a node after a specific node in a circular singly linked list	DeleteAfterNode( )
13	Delete a specific node of a circular singly linked list	DeleteNode( )
14	Delete a node at a specific position of a circular singly linked list	DeleteAtPos( )

C Program to create a circular singly linked list, implement different insertion operations and different deletion operations

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
        int data;
        struct node *link;
};

struct node *CreateNode();
struct node *InsertAtEmpty(int );
struct node *InsertAtEnd(struct node *,int );
void DisplayNodes(struct node *);
int CountNodes(struct node *);
struct node *FreeNode(struct node *);
struct node *InsertAtStart(struct node *,int );
struct node *InsertAtLast(struct node *,int );
struct node *InsertBeforeNode(struct node *,int ,int );
struct node *InsertAfterNode(struct node *,int ,int );
struct node *InsertAtPos(struct node *,int ,int );
struct node *DeleteAtStart(struct node *);
struct node *DeleteAtLast(struct node *);
struct node *DeleteBeforeNode(struct node *,int );
struct node *DeleteAfterNode(struct node *,int );
struct node *DeleteNode(struct node *,int );
struct node *DeleteAtPos(struct node *,int );


int main()
{
        struct node *tail = NULL;
        int option, Value, Data, Position, NodeCount;

        system("clear");

        while(1)
        {
                printf("0: Exit\n");
                printf("1: Create circular linked list\n");
```

```
printf("2: Display all nodes of the circular linked list\n");
printf("3: Display no. of nodes in the circular linked list\n");
printf("4: Insert a node at the starting of the circular linked list\n");
printf("5: Insert a node at the end of the circular linked list\n");
printf("6: Insert a node before a specific node of the circular linked list\n");
printf("7: Insert a node after a specific node of the circular linked list\n");
printf("8: Insert a node at a specific position of the circular linked list\n");
printf("9: Delete a node at the starting of the circular linked list\n");
printf("10: Delete a node at the end of the circular linked list\n");
printf("11: Delete a node before a specific node of the circular linked list\n");
printf("12: Delete a node after a specific node of the circular linked list\n");
printf("13: Delete a specific node of the circular linked list\n");
printf("14: Delete a node at a specific position of the circular linked list\n");
printf("15: Delete all nodes of the circular linked list\n");

printf("\n");
printf("Enter your option: ");
scanf("%d", &option);

switch(option)
{
        case 0: tail = FreeNode(tail);
                exit(1);

        case 1: tail = CreateNode();
                break;

        case 2: DisplayNodes(tail);
                break;

        case 3: NodeCount = CountNodes(tail);
                printf("No. of nodes in the linked list: %d\n\n", NodeCount);
                break;

        case 4: printf("Enter the data of a new node to be inserted at the starting: ");
                scanf("%d", &Data);
                tail = InsertAtStart(tail, Data);
                break;

        case 5: printf("Enter the data of a new node to be inserted at the last: ");
                scanf("%d", &Data);
                tail = InsertAtLast(tail, Data);
                break;

        case 6: printf("Enter the value of a node before which a new node will be
                        inserted: ");
                scanf("%d", &Value);
                printf("Enter the data of the new node to be inserted before the
                        specified node: ");
                scanf("%d", &Data);
                tail = InsertBeforeNode(tail, Value, Data);
                break;
```

```
                case 7: printf("Enter the value of a node after which a new node will be
                                inserted: ");
                        scanf("%d", &Value);
                        printf("Enter the data of the new node to be inserted after the
                                specified node: ");
                        scanf("%d", &Data);
                        tail = InsertAfterNode(tail, Value, Data);
                        break;

                case 8: printf("Enter the position where a node will be inserted: ");
                        scanf("%d", &Position);
                        printf("Enter the data of the node to be inserted at the specified
                                position: ");
                        scanf("%d", &Data);
                        tail = InsertAtPos(tail, Position, Data);
                        break;

                case 9: tail = DeleteAtStart(tail);
                        break;

                case 10: tail = DeleteAtLast(tail);
                        break;

                case 11: printf("Enter the value of a node before which a node will be
                                deleted: ");
                        scanf("%d", &Value);
                        tail = DeleteBeforeNode(tail, Value);
                        break;

               case 12: printf("Enter the value of a node after which a node will be deleted: ");
                        scanf("%d", &Value);
                        tail = DeleteAfterNode(tail, Value);
                        break;

                case 13: printf("Enter the value of a node which will be deleted: ");
                        scanf("%d", &Value);
                        tail = DeleteNode(tail, Value);
                        break;

                case 14: printf("Enter the position where the node will be deleted: ");
                        scanf("%d", &Position);
                        tail = DeleteAtPos(tail, Position);
                        break;

                case 15: if(tail == NULL)
                                printf("Circular linked list is empty.\n\n");
                        else
                        {
                                tail = FreeNode(tail);
                           printf("All nodes of the circular linked list have been deleted\n");
                        }
                        break;
```

```
                        default: printf("Wrong Option\n");
                                 break;
                }
        }

        return 0;
}


struct node *CreateNode()
{
        struct node *tailptr = NULL;
        int i, n, Data;

        printf("Enter the no. of nodes to be created in the circular linked list: ");
        scanf("%d", &n);

        for(i=0; i<n; i++)
        {
                printf("Enter Data%d: ", i+1);
                scanf("%d", &Data);

                if(i == 0)
                {
                        tailptr = InsertAtEmpty(Data);
                }
                else
                {
                        tailptr = InsertAtEnd(tailptr, Data);
                }
        }

        printf("circular Linked list has been created successfully.\n\n");

        return tailptr;
}

struct node *InsertAtEmpty(int Data)
{
        struct node *tailptr;
        tailptr = (struct node *)malloc(sizeof(struct node));

        tailptr->data = Data;
        tailptr->link = tailptr;

        return tailptr;
}
```

```
struct node *InsertAtEnd(struct node *tailptr,int Data)
{
        struct node *ptr;
        ptr = (struct node *)malloc(sizeof(struct node));
        ptr->data = Data;
        ptr->link = tailptr->link;
        tailptr->link = ptr;
        ptr = NULL;
        return tailptr->link;
}

void DisplayNodes(struct node *tailptr)
{
        struct node *ptr;
        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
                return;  }

        ptr = tailptr->link;
        printf("The circular linked list is given below:\n");
        do
        {
                printf("%-5d", ptr->data);
                ptr = ptr->link;
        }while(ptr != tailptr->link);

        printf("\n\n");
        ptr = NULL;
}

int CountNodes(struct node *tailptr)
{
        int count = 0;
        struct node *ptr;
        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
                return count;
        }

        ptr = tailptr->link;
        do
        {
                count++;
                ptr = ptr->link;
        }while(ptr != tailptr->link);

        ptr = NULL;

        return count;
}
```

```
struct node *InsertAtStart(struct node *tailptr,int Data)
{
	struct node *ptr;

	if(tailptr == NULL)
	{
		tailptr = InsertAtEmpty(Data);
	}
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->link = tailptr->link;
		tailptr->link = ptr;
	}

	printf("A new node has been inserted at the starting successfully.\n\n");

	ptr = NULL;

	return tailptr;
}

struct node *InsertAtLast(struct node *tailptr,int Data)
{
	struct node *ptr;

	if(tailptr == NULL)
	{
		tailptr = InsertAtEmpty(Data);
	}
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->link = tailptr->link;
		tailptr->link = ptr;
	}

	printf("A new node has been inserted at the end successfully.\n\n");

	ptr = NULL;
	return tailptr->link;
}
```

```
struct node *InsertBeforeNode(struct node *tailptr,int Value,int Data)
{
	struct node *ptr, *newnodeptr;

	if(tailptr == NULL)
	{
		printf("Circular lined list is empty\n\n");
		return tailptr;
	}
	else if(tailptr->link->data == Value)
	{
		tailptr = InsertAtStart(tailptr, Data);
	}
	else
	{
		ptr = tailptr->link;

		while(ptr->link->data != Value && ptr != tailptr)
			ptr = ptr->link;

		if(ptr == tailptr)
			printf("The node before which insertion will be done is not found.\n\n");
		else
		{
			newnodeptr = (struct node *)malloc(sizeof(struct node));
			newnodeptr->data = Data;
			newnodeptr->link = ptr->link;
			ptr->link = newnodeptr;
			printf("A new node has been inserted before the specified node.\n\n");
		}
	}

	newnodeptr = NULL;
	ptr = NULL;

	return tailptr;
}

struct node *InsertAfterNode(struct node *tailptr,int Value,int Data)
{
	struct node *ptr, *newnodeptr;

	if(tailptr == NULL)
	{
		printf("Circular lined list is empty\n\n");
		return tailptr;
	}
	else if(tailptr->data == Value)
	{
		tailptr = InsertAtLast(tailptr, Data);
	}
```

```
        else
        {
                ptr = tailptr->link;

                while(ptr->data != Value && ptr != tailptr)
                        ptr = ptr->link;

                if(ptr == tailptr)
                        printf("The node after which insertion will be done is not found.\n\n");
                else
                {
                        newnodeptr = (struct node *)malloc(sizeof(struct node));
                        newnodeptr->data = Data;
                        newnodeptr->link = ptr->link;
                        ptr->link = newnodeptr;
                        printf("A new node has been inserted after the specified node successfully.");
                }
        }

        newnodeptr = NULL;
        ptr = NULL;
        return tailptr;
}

struct node *InsertAtPos(struct node *tailptr,int Position,int Data)
{
        struct node *ptr, *newnodeptr;
        int i, NoOfNodes;

        NoOfNodes = CountNodes(tailptr);

        if(Position < 1 || Position > NoOfNodes + 1)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                tailptr = InsertAtStart(tailptr, Data);
        }
        else if(Position == NoOfNodes + 1)
        {
                tailptr = InsertAtLast(tailptr, Data);
        }
        else
        {
                ptr = tailptr->link;
                for(i=1; i<Position - 1; i++)
                        ptr = ptr->link;

                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->link = ptr->link;
```

```
                ptr->link = newnodeptr;
                printf("A new node has been inserted at a specific position successfully.\n\n");
        }

        newnodeptr = NULL;
        ptr = NULL;
        return tailptr;
}

struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->link;
                tailptr->link = ptr->link;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}

struct node *DeleteAtLast(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->link)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
```

```
	else
	{
		ptr = tailptr->link;

		while(ptr->link != tailptr)
			ptr = ptr->link;

		ptr->link = tailptr->link;
		free(tailptr);
		tailptr = ptr;
		printf("The node has been deleted at the end successfully.\n\n");
	}

	ptr = NULL;
	return tailptr;
}

struct node *DeleteBeforeNode(struct node *tailptr,int Value)
{
	struct node *ptr, *prevptr;

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
	else if(tailptr->link->link->data == Value)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->link;
		prevptr = tailptr;

		do
		{
			prevptr = ptr;
			ptr = ptr->link;
		}while(ptr->link->data != Value && ptr != tailptr->link);

		if(ptr == tailptr->link)
			printf("The node before which deletion will be done is not found.\n\n");
		else
		{
			prevptr->link = ptr->link;
			if(ptr == tailptr)
				tailptr = prevptr;
			free(ptr);
			printf("The node has been deleted before the specified node successfully.");
		}
	}
```

```
        prevptr = NULL;
        ptr = NULL;
        return tailptr;
}

struct node *DeleteAfterNode(struct node *tailptr,int Value)
{
        struct node *ptr, *prevptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr->data == Value)
        {
                tailptr = DeleteAtStart(tailptr);
        }
        else
        {
                ptr = tailptr->link;
                prevptr = tailptr;

                do
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }while(prevptr->data != Value && ptr != tailptr->link);

                if(ptr == tailptr->link)
                        printf("The node after which deletion will be done is not found.\n\n");
                else
                {
                        prevptr->link = ptr->link;
                        if(ptr == tailptr)
                                tailptr = prevptr;
                        free(ptr);
                        printf("The node has been deleted after the specified node successfully.\n");
                }
        }

        prevptr = NULL;
        ptr = NULL;
        return tailptr;
}
```

```
struct node *DeleteNode(struct node *tailptr,int Value)
{
	struct node *ptr, *prevptr;

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
	else if(tailptr->link->data == Value)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->link;
		prevptr = tailptr;

		do
		{
			prevptr = ptr;
			ptr = ptr->link;
		}while(ptr->data != Value && ptr != tailptr->link);

		if(ptr == tailptr->link)
			printf("The specified node which will be deleted is not found.\n\n");
		else
		{
			prevptr->link = ptr->link;
			if(ptr == tailptr)
				tailptr = prevptr;
			free(ptr);
			printf("The specified node has been deleted successfully.\n\n");
		}

	}

	prevptr = NULL;
	ptr = NULL;
	return tailptr;
}

struct node *DeleteAtPos(struct node *tailptr,int Position)
{
	struct node *ptr, *prevptr;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(tailptr);

	if(tailptr == NULL)
	{
		printf("Circular linked list is empty.\n\n");
	}
```

```
        else if(Position<1 || Position > NoOfNodes)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                tailptr = DeleteAtStart(tailptr);
        }
        else
        {
                ptr = tailptr->link;
                prevptr = tailptr;

                for(i=1; i<Position; i++)
                {
                        prevptr = ptr;
                        ptr = ptr->link;
                }

                prevptr->link = ptr->link;
                if(ptr == tailptr)
                        tailptr = prevptr;
                free(ptr);
                printf("The node at the specified position has been deleted successfully.\n\n");
        }

        prevptr = NULL;
        ptr = NULL;
        return tailptr;
}

struct node *FreeNode(struct node *tailptr)
{
        struct node *ptr;
        while(tailptr != NULL)
        {
                if(tailptr == tailptr->link)
                {
                        free(tailptr);
                        tailptr = NULL;
                }
                else
                {
                        ptr = tailptr->link;
                        tailptr->link = ptr->link;
                        free(ptr);
                }
        }
        ptr = NULL;
        return tailptr;
}
```

2. Circular Doubly Linked List – It is a special type of doubly linked list where the last node points to the first node and the first node points to the last node. Here every node of the linked list contains three parts which are data part and two link parts – one link points to the predecessor node and other link points to the successor node. The data part holds the data, the predecessor link holds the address of the previous node and the successor link holds the address of the next node. The predecessor link is denoted by the pointer 'prev' and the successor link is denoted by the pointer 'next' in case of a circular doubly linked list. The pictorial view of a circular doubly linked list is shown in Fig.6.7.

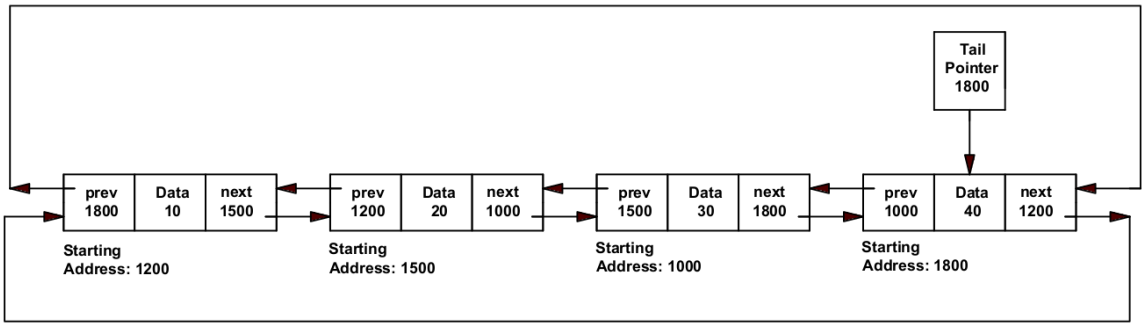


Fig.6.7: Pictorial view of a circular doubly linked list

Comparative study between doubly linked list and circular doubly linked list

Differences between doubly linked list and circular doubly linked list		
SL.	**Doubly Linked List**	**Circular Doubly Linked List**
1.	Last node is not connected to the first node as the prev pointer of the 1st node and the next pointer of the last node are made NULL.	First node and the last node are connected to each other with the help of the prev pointer of the 1st node and the next pointer of the last node.
2.	The prev part and the next part of the 1st and the last node point to NULL. That's why they do not hold any address.	The prev part and the next part of the 1st and the last node hold the addresses of last node and the 1st node respectively in case of a circular doubly linked list.
3.	Traversal from last node to first node or 1st node to last node is not possible.	Traversal from last node to 1st node as well as from 1st node to the last node, both are possible.
4.	Doubly linked list is accessed only by a head pointer which points to the first node of the linked list.	4. Circular singly linked list is accessed by a tail pointer which points to the last node of the linked list.
5.	As the doubly linked list is pointed by the head pointer at its first node, traversal of nodes towards the last node is required to insert a new node at the end of the linked list. Therefore the time complexity is O(n) here for a doubly linked list with n number of nodes	As the circular doubly linked list is pointed by the tail pointer at its last node, traversal of nodes is not required to insert a new node at the end of the linked list. Therefore the time complexity is O(1) here for a circular doubly linked list with n number of nodes

Similarities between doubly linked list and circular doubly linked list		
SL.	**Doubly Linked List**	**Circular Doubly linked List**
1.	It is a linear data structure, because every node has one predecessor and one successor except the first and the last node.	It is a linear data structure as every node has one predecessor and one successor.
2.	Every node has three parts – data part and two link parts.	Every node has three parts – data part and two link parts.
3.	Traversal of nodes is possible in both direction.	Traversal of nodes is possible in both direction.
4.	Insertion of a new node at the beginning of the doubly linked list is done without any traversal. Therefore the time complexity becomes O(1).	Insertion of a new node at the beginning of the circular doubly linked list is done without any traversal. Therefore the time complexity becomes O(1).

Declaring a circular doubly linked list – It is being observed that every node of a circular doubly linked list has three parts – one is data and other two are links or pointers. Among these two pointers one points to the predecessor node and denoted by 'prev' and other points to the successor node and denoted by 'next'. A circular doubly linked list is declared as follows in C language.

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

In this case fourteen different operations can be developed in the same way. The algorithms of these functions may be implemented in case of a circular doubly linked list also using the same convention like circular singly linked list. Therefore these algorithms are not revealed in this section. All these operations are combined in a single C program and implemented using different user-defined functions as given below.

SL	Operations applied on circular doubly linked list	User-defined functions
1	Creating a circular doubly linked list	CreateNode( )
2	Displaying the data of all nodes in a circular doubly linked list	DisplayNodes( )
3	Counting the number of nodes in a circular doubly linked list	CountNodes( )
4	Insert a node at the beginning of a circular doubly linked list	InsertAtStart( )
5	Insert a node at the end of a circular doubly linked list	InsertAtLast( )
6	Insert a node before a specific node in circular doubly linked list	InsertBeforeNode( )

SL	Operations applied on circular doubly linked list	User-defined functions
7	Insert a node after a specific node in a circular doubly linked list	InsertAfterNode( )
8	Insert a node at a specific position of a circular doubly linked list	InsertAtPos( )
9	Delete a node at the beginning of a circular doubly linked list	DeleteAtStart( )
10	Delete a node at the end of a circular doubly linked list	DeleteAtLast( )
11	Delete a node before a specific node in a circular doubly linked list	DeleteBeforeNode( )
12	Delete a node after a specific node in a circular doubly linked list	DeleteAfterNode( )
13	Delete a specific node of a circular doubly linked list	DeleteNode( )
14	Delete a node at a specific position of a circular doubly linked list	DeleteAtPos( )

C Program to create a circular doubly linked list, implement different insertion operations and different deletion operations

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
        struct node *prev;
        int data;
        struct node *next;
};

struct node *CreateNode();
struct node *InsertAtEmpty(int );
struct node *InsertAtEnd(struct node *,int );
void DisplayNodes(struct node *);
int CountNodes(struct node *);
struct node *FreeNode(struct node *);
struct node *InsertAtStart(struct node *,int );
struct node *InsertAtLast(struct node *,int );
struct node *InsertBeforeNode(struct node *,int ,int );
struct node *InsertAfterNode(struct node *,int ,int );
struct node *InsertAtPos(struct node *,int ,int );
struct node *DeleteAtStart(struct node *);
struct node *DeleteAtLast(struct node *);
struct node *DeleteBeforeNode(struct node *,int );
struct node *DeleteAfterNode(struct node *,int );
struct node *DeleteNode(struct node *,int );
struct node *DeleteAtPos(struct node *,int );
```

```
int main()
{
	struct node *tail = NULL;
	int option, Value, Data, Position, NodeCount;

	while(1)
	{
		printf("0: Exit\n");
		printf("1: Create circular doubly linked list\n");
		printf("2: Display all nodes of the circular doubly linked list\n");
		printf("3: Display no. of nodes in the circular doubly linked list\n");
		printf("4: Insert a node at the starting of the circular doubly linked list\n");
		printf("5: Insert a node at the end of the circular doubly linked list\n");
		printf("6: Insert a node before a specific node of the circular doubly linked list\n");
		printf("7: Insert a node after a specific node of the circular doubly linked list\n");
		printf("8: Insert a node at a specific position of the circular doubly linked list\n");
		printf("9: Delete a node at the starting of the circular doubly linked list\n");
		printf("10: Delete a node at the end of the circular doubly linked list\n");
		printf("11: Delete a node before a specific node of the circular doubly linked list\n");
		printf("12: Delete a node after a specific node of the circular doubly linked list\n");
		printf("13: Delete a specific node of the circular doubly linked list\n");
		printf("14: Delete a node at a specific position of the circular doubly linked list\n");
		printf("15: Delete all nodes of the circular doubly linked list\n");

		printf("\n");
		printf("Enter your option: ");
		scanf("%d", &option);

		switch(option)
		{
			case 0: tail = FreeNode(tail);
				exit(1);

			case 1: tail = CreateNode();
				break;

			case 2: DisplayNodes(tail);
				break;

			case 3: NodeCount = CountNodes(tail);
				printf("No. of nodes in the linked list: %d\n\n", NodeCount);
				break;

			case 4: printf("Enter the data of a new node to be inserted at the starting: ");
				scanf("%d", &Data);
				tail = InsertAtStart(tail, Data);
				break;

			case 5: printf("Enter the data of a new node to be inserted at the last: ");
				scanf("%d", &Data);
				tail = InsertAtLast(tail, Data);
				break;
```

```
                case 6: printf("Enter the value of a node before which a new node will be
                                inserted: ");
                        scanf("%d", &Value);
                        printf("Enter the data of the new node to be inserted before the
                                specified node: ");
                        scanf("%d", &Data);
                        tail = InsertBeforeNode(tail, Value, Data);
                        break;

                case 7: printf("Enter the value of a node after which a new node will be
                                inserted: ");
                        scanf("%d", &Value);
                        printf("Enter the data of the new node to be inserted after the
                                specified node: ");
                        scanf("%d", &Data);
                        tail = InsertAfterNode(tail, Value, Data);
                        break;

                case 8: printf("Enter the position where a node will be inserted: ");
                        scanf("%d", &Position);
                        printf("Enter the data of the node to be inserted at the specified
                                position: ");
                        scanf("%d", &Data);
                        tail = InsertAtPos(tail, Position, Data);
                        break;

                case 9: tail = DeleteAtStart(tail);
                        break;

                case 10: tail = DeleteAtLast(tail);
                        break;

                case 11: printf("Enter the value of a node before which a node will be
                                deleted: ");
                        scanf("%d", &Value);
                        tail = DeleteBeforeNode(tail, Value);
                        break;

               case 12: printf("Enter the value of a node after which a node will be deleted: ");
                        scanf("%d", &Value);
                        tail = DeleteAfterNode(tail, Value);
                        break;

                case 13: printf("Enter the value of a node which will be deleted: ");
                        scanf("%d", &Value);
                        tail = DeleteNode(tail, Value);
                        break;

                case 14: printf("Enter the position where the node will be deleted: ");
                        scanf("%d", &Position);
                        tail = DeleteAtPos(tail, Position);
                        break;
```

```
                        case 15: if(tail == NULL)
                                        printf("circular doubly linked list is empty.\n\n");
                                else
                                {
                                        tail = FreeNode(tail);
                                        printf("All nodes of the circular doubly linked list have been
                                                deleted successfully.\n\n");
                                }
                                break;

                        default: printf("Wrong Option\n");
                                break;
                }
        }
        return 0;
}

struct node *CreateNode()
{
        struct node *tailptr = NULL;
        int i, n, Data;

        printf("Enter the no. of nodes to be created in the circular doubly linked list: ");
        scanf("%d", &n);

        for(i=0; i<n; i++)
        {
                printf("Enter Data%d: ", i+1);
                scanf("%d", &Data);

                if(i == 0)
                {
                        tailptr = InsertAtEmpty(Data);
                }
                else
                {
                        tailptr = InsertAtEnd(tailptr, Data);
                }
        }

        printf("circular doubly Linked list has been created successfully.\n\n");

        return tailptr;
}
```

```
struct node *InsertAtEmpty(int Data)
{
	struct node *tailptr;
	tailptr = (struct node *)malloc(sizeof(struct node));

	tailptr->data = Data;
	tailptr->prev = tailptr;
	tailptr->next = tailptr;
	return tailptr;
}

struct node *InsertAtEnd(struct node *tailptr,int Data)
{
	struct node *ptr;
	ptr = (struct node *)malloc(sizeof(struct node));
	ptr->data = Data;
	ptr->prev = tailptr;
	ptr->next = tailptr->next;
	tailptr->next->prev = ptr;
	tailptr->next = ptr;
	ptr = NULL;
	return tailptr->next;
}

void DisplayNodes(struct node *tailptr)
{
	struct node *ptr;
	if(tailptr == NULL)
	{
		printf("circular doubly linked list is empty.\n\n");
		return;
	}
	ptr = tailptr->next;
	printf("The circular doubly linked list is given below:\n");
	do
	{
		printf("%-5d", ptr->data);
		ptr = ptr->next;
	}while(ptr != tailptr->next);

	ptr = ptr->prev;
	printf("\n\nThe circular doubly linked list is given below in reverse order:\n");
	do
	{
		printf("%-5d", ptr->data);
		ptr = ptr->prev;
	}while(ptr != tailptr);
	printf("\n\n");

	ptr = NULL;
}
```

```
int CountNodes(struct node *tailptr)
{
	int count = 0;
	struct node *ptr;

	if(tailptr == NULL)
	{
		printf("circular doubly linked list is empty.\n\n");
		return count;
	}

	ptr = tailptr->next;

	do
	{
		count++;
		ptr = ptr->next;
	}while(ptr != tailptr->next);

	ptr = NULL;

	return count;
}


struct node *InsertAtStart(struct node *tailptr,int Data)
{
	struct node *ptr;

	if(tailptr == NULL)
	{
		tailptr = InsertAtEmpty(Data);
	}
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->prev = tailptr;
		ptr->next = tailptr->next;
		tailptr->next->prev = ptr;
		tailptr->next = ptr;
	}

	printf("A new node has been inserted at the starting successfully.\n\n");

	ptr = NULL;

	return tailptr;
}
```

```
struct node *InsertAtLast(struct node *tailptr,int Data)
{
	struct node *ptr;

	if(tailptr == NULL)
		tailptr = InsertAtEmpty(Data);
	else
	{
		ptr = (struct node *)malloc(sizeof(struct node));
		ptr->data = Data;
		ptr->prev = tailptr;
		ptr->next = tailptr->next;
		tailptr->next->prev = ptr;
		tailptr->next = ptr;
	}
	printf("A new node has been inserted at the end successfully.\n\n");
	ptr = NULL;
	return tailptr->next;
}

struct node *InsertBeforeNode(struct node *tailptr,int Value,int Data)
{
	struct node *ptr, *newnodeptr;
	if(tailptr == NULL)
	{
		printf("circular doubly lined list is empty\n\n");
		return tailptr;
	}
	else if(tailptr->next->data == Value)
		tailptr = InsertAtStart(tailptr, Data);
	else
	{
		ptr = tailptr->next;
		while(ptr->next->data != Value && ptr != tailptr)
			ptr = ptr->next;
		if(ptr == tailptr)
			printf("The node before which insertion will be done is not found.\n\n");
		else
		{
			newnodeptr = (struct node *)malloc(sizeof(struct node));
			newnodeptr->data = Data;
			newnodeptr->prev = ptr;
			newnodeptr->next = ptr->next;
			ptr->next->prev = newnodeptr;
			ptr->next = newnodeptr;
		       printf("A new node has been inserted before the specified node successfully");
		}
	}
	newnodeptr = NULL;
	ptr = NULL;
	return tailptr;
}
```

```
struct node *InsertAfterNode(struct node *tailptr,int Value,int Data)
{
	struct node *ptr, *newnodeptr;

	if(tailptr == NULL)
	{
		printf("circular doubly lined list is empty\n\n");
		return tailptr;
	}
	else if(tailptr->data == Value)
	{
		tailptr = InsertAtLast(tailptr, Data);
	}
	else
	{
		ptr = tailptr->next;

		while(ptr->data != Value && ptr != tailptr)
			ptr = ptr->next;

		if(ptr == tailptr)
			printf("The node after which insertion will be done is not found.\n\n");
		else
		{
			newnodeptr = (struct node *)malloc(sizeof(struct node));
			newnodeptr->data = Data;
			newnodeptr->prev = ptr;
			newnodeptr->next = ptr->next;
			ptr->next->prev = newnodeptr;
			ptr->next = newnodeptr;
			printf("A new node has been inserted after the specified node successfully.");
		}
	}

	newnodeptr = NULL;
	ptr = NULL;
	return tailptr;
}

struct node *InsertAtPos(struct node *tailptr,int Position,int Data)
{
	struct node *ptr, *newnodeptr;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(tailptr);

	if(Position < 1 || Position > NoOfNodes + 1)
		printf("The specified position is wrong.\n\n");
	else if(Position == 1)
		tailptr = InsertAtStart(tailptr, Data);
```

```
        else if(Position == NoOfNodes + 1)
                tailptr = InsertAtLast(tailptr, Data);
        else
        {
                ptr = tailptr->next;
                for(i=1; i<Position - 1; i++)
                        ptr = ptr->next;

                newnodeptr = (struct node *)malloc(sizeof(struct node));
                newnodeptr->data = Data;
                newnodeptr->prev = ptr;
                newnodeptr->next = ptr->next;
                ptr->next->prev = newnodeptr;
                ptr->next = newnodeptr;
                printf("A new node has been inserted at a specific position successfully.\n\n");
        }
        newnodeptr = NULL;
        ptr = NULL;
        return tailptr;
}

struct node *DeleteAtStart(struct node *tailptr)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("circular doubly linked list is empty.\n\n");
        }
        else if(tailptr == tailptr->next)
        {
                ptr = tailptr;
                free(ptr);
                tailptr = NULL;
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        else
        {
                ptr = tailptr->next;
                tailptr->next = ptr->next;
                tailptr->next->prev = tailptr;
                free(ptr);
                printf("The node has been deleted at the starting successfully.\n\n");
        }
        ptr = NULL;
        return tailptr;
}
```

```
struct node *DeleteAtLast(struct node *tailptr)
{
	struct node *ptr;

	if(tailptr == NULL)
	{
		printf("circular doubly linked list is empty.\n\n");
	}
	else if(tailptr == tailptr->next)
	{
		ptr = tailptr;
		free(ptr);
		tailptr = NULL;
		printf("The node has been deleted at the starting successfully.\n\n");
	}
	else
	{
		ptr = tailptr->next;

		while(ptr->next != tailptr)
			ptr = ptr->next;

		ptr->next = tailptr->next;
		tailptr->next->prev = ptr;
		free(tailptr);
		tailptr = ptr;
		printf("The node has been deleted at the end successfully.\n\n");
	}

	ptr = NULL;
	return tailptr;
}

struct node *DeleteBeforeNode(struct node *tailptr,int Value)
{
	struct node *ptr;

	if(tailptr == NULL)
		printf("circular doubly linked list is empty.\n\n");
	else if(tailptr->next->next->data == Value)
		tailptr = DeleteAtStart(tailptr);
	else
	{
		ptr = tailptr->next;

		do
		{
			ptr = ptr->next;
		}while(ptr->next->data != Value && ptr != tailptr->next);

		if(ptr == tailptr->next)
			printf("The node before which deletion will be done is not found.\n\n");
```

```
                else
                {
                        ptr->prev->next = ptr->next;
                        ptr->next->prev = ptr->prev;
                        if(ptr == tailptr)
                                tailptr = ptr->prev;
                        free(ptr);
                        printf("The node has been deleted before the specified node successfully.");
                }
        }
        ptr = NULL;
        return tailptr;
}

struct node *DeleteAfterNode(struct node *tailptr,int Value)
{
        struct node *ptr;

        if(tailptr == NULL)
        {
                printf("Circular linked list is empty.\n\n");
        }
        else if(tailptr->data == Value)
        {
                tailptr = DeleteAtStart(tailptr);
        }
        else
        {
                ptr = tailptr->next;


                do
                {
                        ptr = ptr->next;
                }while(ptr->prev->data != Value && ptr != tailptr->next);

                if(ptr == tailptr->next)
                        printf("The node after which deletion will be done is not found.\n\n");
                else
                {
                        ptr->prev->next = ptr->next;
                        ptr->next->prev = ptr->prev;
                        if(ptr == tailptr)
                                tailptr = ptr->prev;
                        free(ptr);
                        printf("The node has been deleted after the specified node successfully.\n");
                }
        }

        ptr = NULL;
        return tailptr;
}
```

```
struct node *DeleteNode(struct node *tailptr,int Value)
{
	struct node *ptr;

	if(tailptr == NULL)
	{
		printf("circular doubly linked list is empty.\n\n");
	}
	else if(tailptr->next->data == Value)
	{
		tailptr = DeleteAtStart(tailptr);
	}
	else
	{
		ptr = tailptr->next;

		do
		{
			ptr = ptr->next;
		}while(ptr->data != Value && ptr != tailptr->next);

		if(ptr == tailptr->next)
			printf("The specified node which will be deleted is not found.\n\n");
		else
		{
			ptr->prev->next = ptr->next;
			ptr->next->prev = ptr->prev;
			if(ptr == tailptr)
				tailptr = ptr->prev;
			free(ptr);
			printf("The specified node has been deleted successfully.\n\n");
		}

	}

	ptr = NULL;
	return tailptr;
}


struct node *DeleteAtPos(struct node *tailptr,int Position)
{
	struct node *ptr, *prevptr;
	int i, NoOfNodes;

	NoOfNodes = CountNodes(tailptr);

	if(tailptr == NULL)
	{
		printf("circular doubly linked list is empty.\n\n");
	}
```

```
        else if(Position<1 || Position > NoOfNodes)
        {
                printf("The specified position is wrong.\n\n");
        }
        else if(Position == 1)
        {
                tailptr = DeleteAtStart(tailptr);
        }
        else
        {
                ptr = tailptr->next;

                for(i=1; i<Position; i++)                       {

                        ptr = ptr->next;
                }

                ptr->prev->next = ptr->next;
                ptr->next->prev = ptr->prev;
                if(ptr == tailptr)
                        tailptr = ptr->prev;
                free(ptr);
                printf("The node at the specified position has been deleted successfully.\n\n");
        }

        ptr = NULL;
        return tailptr;
}

struct node *FreeNode(struct node *tailptr)
{
        struct node *ptr;
        while(tailptr != NULL)
        {
                if(tailptr == tailptr->next)
                {
                        free(tailptr);
                        tailptr = NULL;
                }
                else
                {
                        ptr = tailptr->next;
                        tailptr->next = ptr->next;
                        tailptr->next->prev = tailptr;
                        free(ptr);
                }
        }

        ptr = NULL;
        return tailptr;
}
```

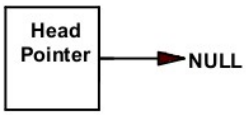
Polynomial Arithmetic – It is one of the applications of linked list. A polynomial is an algebraic expression with multiple terms where each term has two parts namely coefficient and exponent. For example – $5x^4 – 2.5x^2 + x – 3$ is a polynomial with four terms where 1st term having the coefficient +5 and the exponent +4, 2nd term having the coefficient -2.5 and the exponent +2 etc.

A polynomial can be represented using a singly linked list to perform various polynomial arithmetic like polynomial addition, polynomial subtraction, polynomial multiplication etc. Here each term of the polynomial is represented by the node of a singly linked list. As there are two parts (coefficient and exponent) in every term of a polynomial, every node of a singly list must have three parts in it. The first part holds the coefficient, the second part holds the exponent and the third part holds the link to point to the next node of the singly linked list. Therefore the singly linked list will be declared as follows here.

```
struct node
{
        float coef;
        int expo;
        struct node *link;
};
```

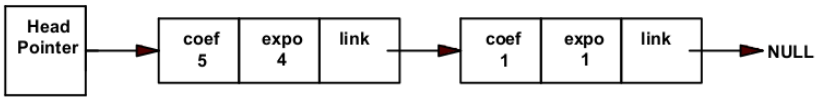
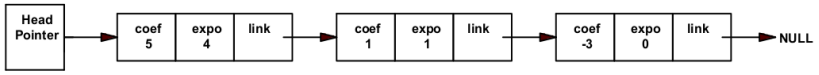
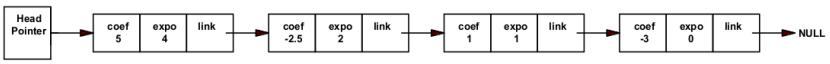
In the above declaration, the variable coef is used to store the coefficient and the variable expo is used to store the exponent of each term of the polynomial.

Now sometimes the polynomial may be given in the random order of the exponents of the terms. For example – the polynomial ($5x^4 – 2.5x^2 + x – 3$) which was given in the descending order of the exponents previously may be expressed in random order of the exponents like ($x + 5x^4 – 3 – 2.5x^2$) or ($– 2.5x^2 + x – 3 + 5x^4$). Here the polynomial should be represented using a singly linked list in such a way that the node with higher exponent comes first. In other words we can say that the singly linked list must be a sorted linked list in descending order of the exponents. Hence how the polynomial ($x + 5x^4 – 3 – 2.5x^2$) can be represented by a singly linked list is given below step by step.

<table>
<tr><th colspan="3">Polynomial: $x + 5x^4 – 3 – 2.5x^2$</th></tr>
<tr><th>Steps</th><th>Terms from polynomial</th><th>Status of the singly linked list</th></tr>
<tr><td>Initially</td><td></td><td> 

Empty linked list</td></tr>
<tr><td>Step 1</td><td>Add the term x with coefficient = 1 exponent = 1</td><td> 

After adding 1st term of polynomial</td></tr>
</table>

Steps	Terms from polynomial	Status of the singly linked list
Step 2	Add the term $5x^4$ with coefficient = 5 exponent = 4	 After adding 2nd term of polynomial
Step 3	Add the term - 3 with coefficient = -3 exponent = 0	 After adding 3rd term of polynomial
Step 4	Add the term $-2.5x^2$ with coefficient = -2.5 exponent = 2	 After adding 4th term of polynomial

From the above example it is clear that after receiving the exponents of successive terms of the polynomial, a new node is created with that exponent and the new node is inserted at the proper position into the singly linked list in such a way that the linked list remains sorted in descending of the exponents always. The algorithm of creating a singly linked list to represent a polynomial is given below.

Algorithm to create a singly linked list for polynomial representation:

Step 1: Start

Step 2: Set Head_Pointer of the linked list = NULL

Step 3: Input the number of terms of the polynomial into the variable ‘n’.

Step 4: Set i = 0

Step 5: Repeat Step 6 to Step 13 while i < n

Step 6: Input the coefficient of the term into the variable ‘coefficient’.

Step 7: Input the exponent of the term into the variable ‘exponent’.

Step 8: Create a new node pointed by New_Node_Pointer.

Step 9: Set New_Node_Pointer → coef = coefficient

Step 10: Set New_Node_Pointer → expo = exponent

Step 11: Set New_Node_Pointer → link = NULL

Step 12: If Head_Pointer = NULL *(Condition for empty linked list)* OR
Head_Pointer → expo < exponent of the term, then
a) Set New_Node_Pointer → link = Head_Pointer
b) Set Head_Pointer = New_Node_Pointer
Otherwise
c) Set ptr = Head_Pointer
d) Repeat e) while ptr → link ≠ NULL AND ptr → link → expo > exponential
e) Set ptr = ptr → link
[End of Loop (d)]
f) Set New_Node_Pointer → link = ptr → link
g) Set ptr → link = New_Node_Pointer
[End of If-Else (Step 12)]

Step 13: Set i = i + 1
[End of Loop (Step 5)]

Step 14: Set ptr = NULL

Step 15: Set New_Node_Pointer = NULL

Step 16: Stop

User defined function in C to create a singly linked list for polynomial representation

```
struct node *Create(struct node *headptr)
{
        int n, exponent, i;
        float coefficient;

        printf("Enter the no. of terms of the polynomial: ");
        scanf("%d", &n);

        for(i=0; i<n; i++)
        {
                printf("Enter the coefficient of term %d: ", i+1);
                scanf("%f", &coefficient);
                printf("Enter the exponent of term %d: ", i+1);
                scanf("%d", &exponent);
                headptr = Insert(headptr, coefficient, exponent);
        }
        return headptr;
}

struct node *Insert(struct node *headptr, float coeff, int expon)
{
        struct node *newnodeptr, *ptr;
        newnodeptr = (struct node *)malloc(sizeof(struct node));
        newnodeptr->coef = coeff;
```

```
        newnodeptr->expo = expon;
        newnodeptr->link = NULL;

        if(headptr == NULL || headptr->expo < expon)
        {
                newnodeptr->link = headptr;
                headptr = newnodeptr;
        }
        else
        {
                ptr = headptr;
                while(ptr->link != NULL && ptr->link->expo > expon)
                        ptr = ptr->link;

                newnodeptr->link = ptr->link;
                ptr->link = newnodeptr;
        }
        ptr = NULL;
        newnodeptr = NULL;

        return headptr;
}
```

Addition of Polynomials – Two polynomials can be added using singly linked lists. Here two polynomials are represented by two sorted singly linked lists first in a descending order of the exponents. During the addition of the polynomials the coefficients of likely nodes with the equal exponents are added together and stored into the node of the resultant linked list with same exponent and the other nodes with different exponents are directly copied into the node of the resultant linked list. After the completion of the addition the nodes of the resultant linked list are also arranged in descending order depending upon the exponents. Thus the sorted resultant linked list holds the addition of the two singly linked lists representing the two polynomials. The following example demonstrate the procedure of addition.

$$5x^4 - 2.5x^2 + x - 3$$

$$+\ 6x^6 + 1.5x^5 - 2x^4 - 2x + 7.5$$

$$6x^6 + 1.5x^5 + 3x^4 - 2.5x^2 - x + 4.5$$

Algorithm of polynomial addition:

Step 1: Start

Step 2: Set Head_Pointer3 of the resultant linked list = NULL

Step 3: Set ptr1 = Head_Pointer1 *(Head pointer of singly linked list for 1st polynomial)*

Step 4: Set ptr2 = Head_Pointer2 *(Head pointer of singly linked list for 2nd polynomial)*

Step 5: Repeat Step 6 while ptr1 ≠ NULL AND ptr2 ≠ NULL

Step 6: If ptr1 → expo = ptr2 → expo, then
- a) Add the coefficients and insert the newly created node into the sorted resultant linked list.
- b) Set ptr1 = ptr1 → link
- c) Set ptr2 = ptr2 → link

Otherwise
- d) If ptr1 → expo > ptr2 → expo, then
- e) Insert the node pointed by ptr1 into the sorted resultant linked list
- f) Set ptr1 = ptr1 → link

Otherwise
- g) Insert the node pointed by ptr2 into the sorted resultant linked list
- h) Set ptr2 = ptr2 → link

[End of If-Else (d)]

[End of If-Else (Step 6)]

[End of Loop (Step 5)]

Step 7: Repeat Step 8 and Step 9 while ptr1 ≠ NULL

Step 8: Insert the remaining nodes pointed by ptr1 into the resultant linked list

Step 9: Set ptr1 = ptr1 → link
[End of Loop (Step 7)]

Step 10: Repeat Step 11 and Step 12 while ptr2 ≠ NULL

Step 11: Insert the remaining nodes pointed by ptr2 into the resultant linked list

Step 12: Set ptr2 = ptr2 → link
[End of Loop (Step 10)]

Step 13: Stop

User defined function in C to add two polynomials represented by two singly linked lists

```
struct node *PolyAdd(struct node *headptr1, struct node *headptr2)
{
        struct node *headptr3 = NULL, *ptr1, *ptr2;

        ptr1 = headptr1;
        ptr2 = headptr2;

        while(ptr1 != NULL && ptr2 != NULL)
        {
                if(ptr1->expo == ptr2->expo)
                {
                        headptr3 = Insert(headptr3, ptr1->coef + ptr2->coef, ptr1->expo);
                        ptr1 = ptr1->link;
                        ptr2 = ptr2->link;
                }
                else if(ptr1->expo > ptr2->expo)
                {
                        headptr3 = Insert(headptr3, ptr1->coef, ptr1->expo);
                        ptr1 = ptr1->link;
                }
                else
                {
                        headptr3 = Insert(headptr3, ptr2->coef, ptr2->expo);
                        ptr2 = ptr2->link;
                }
        }

        while(ptr1 != NULL)
        {
                headptr3 = Insert(headptr3, ptr1->coef, ptr1->expo);
                ptr1 = ptr1->link;
        }

        while(ptr2 != NULL)
        {
                headptr3 = Insert(headptr3, ptr2->coef, ptr2->expo);
                ptr2 = ptr2->link;
        }

        return headptr3;
}
```

```
struct node *Insert(struct node *headptr, float coeff, int expon)
{
        struct node *newnodeptr, *ptr;
        newnodeptr = (struct node *)malloc(sizeof(struct node));
        newnodeptr->coef = coeff;
        newnodeptr->expo = expon;
        newnodeptr->link = NULL;

        if(headptr == NULL || headptr->expo < expon)
        {
                newnodeptr->link = headptr;
                headptr = newnodeptr;
        }
        else
        {
                ptr = headptr;
                while(ptr->link != NULL && ptr->link->expo > expon)
                        ptr = ptr->link;

                newnodeptr->link = ptr->link;
                ptr->link = newnodeptr;
        }
        ptr = NULL;
        newnodeptr = NULL;

        return headptr;
}
```

Now the entire C program to add two polynomials using singly linked list is given below for better understanding of the readers.

Entire C program to add two polynomials represented by two singly linked lists

```
#include<stdio.h>
#include<stdlib.h>

struct node *Create(struct node *);
struct node *Insert(struct node *,float ,int );
void Display(struct node *);
struct node *PolyAdd(struct node *,struct node *);

struct node
{
        float coef;
        int expo;
        struct node *link;
};
```

```
int main()
{
	struct node *head1 = NULL, *head2 = NULL, *head3 = NULL;

	printf("Enter the 1st polynomial:\n\n");
	head1 = Create(head1);

	printf("Enter the 2nd polynomial:\n\n");
	head2 = Create(head2);

	printf("The 1st polynomial:\n\n");
	Display(head1);

	printf("The 2nd polynomial:\n\n");
	Display(head2);

	head3 = PolyAdd(head1, head2);

	printf("The resultant polynomial:\n\n");
	Display(head3);
	return 0;
}

struct node *Create(struct node *headptr)
{
	int n, exponent, i;
	float coefficient;

	printf("Enter the no. of terms of the polynomial: ");
	scanf("%d", &n);

	for(i=0; i<n; i++)
	{
		printf("Enter the coefficient of term %d: ", i+1);
		scanf("%f", &coefficient);
		printf("Enter the exponent of term %d: ", i+1);
		scanf("%d", &exponent);

		headptr = Insert(headptr, coefficient, exponent);
	}

	return headptr;
}
```

```
struct node *Insert(struct node *headptr, float coeff, int expon)
{
        struct node *newnodeptr, *ptr;

        newnodeptr = (struct node *)malloc(sizeof(struct node));
        newnodeptr->coef = coeff;
        newnodeptr->expo = expon;
        newnodeptr->link = NULL;

        if(headptr == NULL || headptr->expo < expon)
        {
                newnodeptr->link = headptr;
                headptr = newnodeptr;
        }
        else
        {
                ptr = headptr;
                while(ptr->link != NULL && ptr->link->expo > expon)
                        ptr = ptr->link;

                newnodeptr->link = ptr->link;
                ptr->link = newnodeptr;
        }
        ptr = NULL;
        newnodeptr = NULL;
        return headptr;
}

struct node *PolyAdd(struct node *headptr1, struct node *headptr2)
{
        struct node *headptr3 = NULL, *ptr1, *ptr2;
        ptr1 = headptr1;
        ptr2 = headptr2;
        while(ptr1 != NULL && ptr2 != NULL)
        {
                if(ptr1->expo == ptr2->expo)
                {
                        headptr3 = Insert(headptr3, ptr1->coef + ptr2->coef, ptr1->expo);
                        ptr1 = ptr1->link;
                        ptr2 = ptr2->link;
                }
                else if(ptr1->expo > ptr2->expo)
                {
                        headptr3 = Insert(headptr3, ptr1->coef, ptr1->expo);
                        ptr1 = ptr1->link;
                }
                else
                {
                        headptr3 = Insert(headptr3, ptr2->coef, ptr2->expo);
                        ptr2 = ptr2->link;
                }
```

```
        }
        while(ptr1 != NULL)
        {
                headptr3 = Insert(headptr3, ptr1->coef, ptr1->expo);
                ptr1 = ptr1->link;
        }
        while(ptr2 != NULL)
        {
                headptr3 = Insert(headptr3, ptr2->coef, ptr2->expo);
                ptr2 = ptr2->link;
        }
        return headptr3;
}

void Display(struct node *headptr)
{
        struct node *ptr = headptr;
        while(ptr != NULL)
        {
                if(ptr->expo == 0)
                {
                        if(ptr == headptr && ptr->coef > 0)
                                printf("%.1f ",ptr->coef);
                        else if(ptr->coef > 0)
                                printf("+ %.1f ",ptr->coef);
                        else
                                printf("- %.1f ", (-1)*ptr->coef);
                }
                else
                {
                        if(ptr == headptr && ptr->coef > 0)
                                printf("%.1fx^%d ",ptr->coef, ptr->expo);
                        else if(ptr->coef > 0)
                                printf("+ %.1fx^%d ",ptr->coef, ptr->expo);
                        else
                                printf("- %.1fx^%d ", (-1)*ptr->coef, ptr->expo);
                }

                ptr = ptr->link;
        }

        printf("\n\n");

        ptr = NULL;
}
```

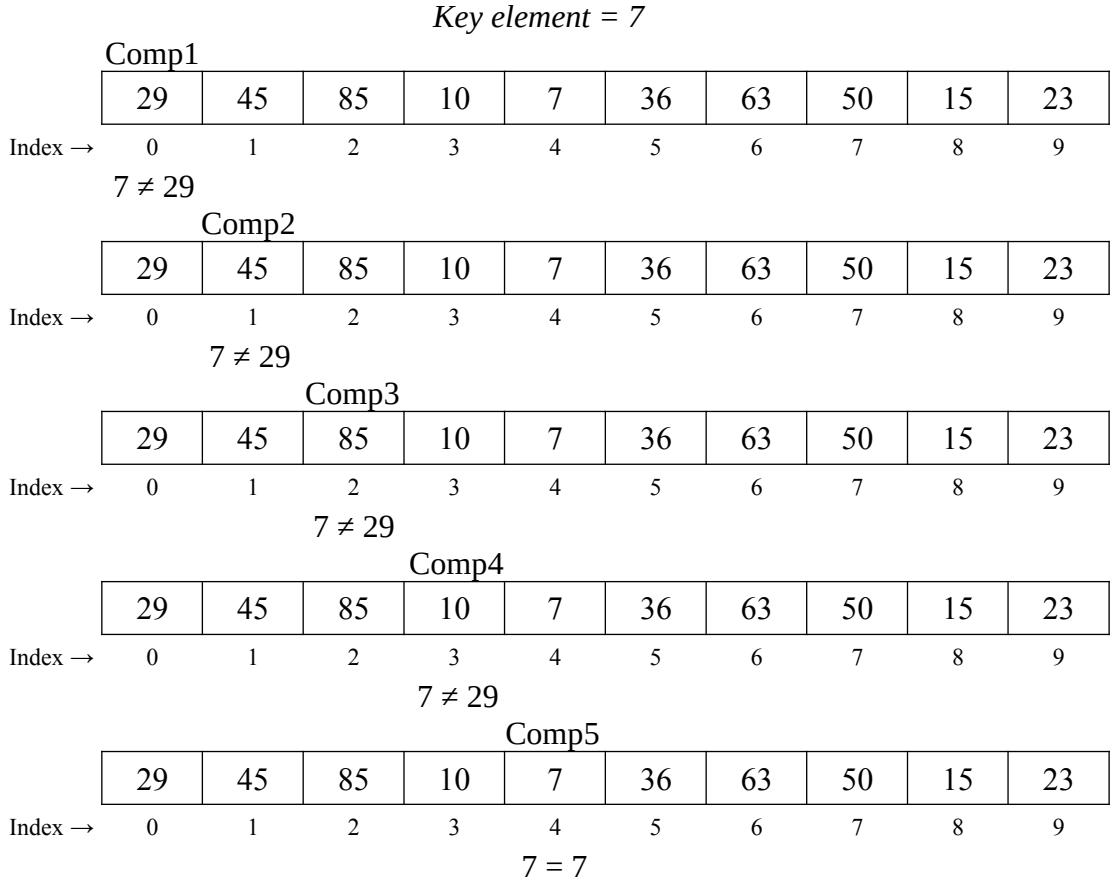
Chapter 7

Searching and Sorting

Searching – Searching is a method by which it is checked whether a particular value is present or not in a set of values. Normally the list of values are stored in an array and the value which is to be searched is taken into a variable. The value which is to be searched is known as key value. If the key value is found in the array, the index of the array or the position where the key value is found is printed and if it is not found, an error message "The value is not found" is displayed. Here we shall discuss the two searching techniques namely linear search and binary search.

1) Linear Search – In case of linear search the list of numbers may be *sorted or unsorted*. The key element which is to be searched in an unsorted array is compared with each and every element one by one. If any element in the array matches with the key element, the position or the index of that element will be displayed on the screen to indicate that the key element is found in the array. Once the key element is found, further comparison beyond the matched element is not required and the searching process should be stopped. If there is no matching with any of the elements of the array, a message "Element is not found" will be printed. For this purpose we have to go through the following example.

Suppose we have a list of elements like 29, 45, 85, 10, 7, 36, 63, 50, 15, 23 and the key element 36. The procedure to search the key element is given step by step in Fig.7.1.



Key element 7 is found at index = 4 or position = 5

Fig.7.1: Searching process of linear search step by step

Algorithm of linear search:

Step 1: Start

Step 2: Input the no. of elements in a variable ‘n’.

Step 3: Input the list of n no. of elements into an array ‘a’.

Step 4: Input the key element in a variable ‘key’.

Step 5: Set Index = - 1

Step 6: Set i = 0

Step 7: Repeat Step 8 to Step 9 while i < n

Step 8: If a[i] = key, then
a) Set Index = i
Print “The key element is found at position Index + 1 ”.
Go to Step 11
[End of If (Step 8)]

Step 9: Set i = i + 1
[End of Loop (Step 7)]

Step 10: If Index = - 1, then
a) Print “The key element is not found”.
[End of If (Step 10)]

Step 11: Stop

C program to search an element from a set of numbers using linear search.

```
#include<stdio.h>
int Lsearch(int [],int ,int );

int main()
{
        int a[50], i, n, key, result;

        printf("Enter the number of elements: ");
        scanf("%d", &n);
        printf("Enter the elements of the array:\n");
        for(i=0; i<n; i++)
        {
                printf("Enter Element%d: ", i+1);
                scanf("%d", &a[i]);
        }
```

```
        printf("Enter the value of the element to be searched: ");
        scanf("%d", &key);

        result = Lsearch(a, n, key);

        if(result == -1)
                printf("The element is not found.\n");
        else
                printf("The element is found at position %d\n", result + 1);

        return 0;
}

int Lsearch(int arr[],int len,int k)
{
        int i;

        for(i=0; i<len; i++)
        {
                if(arr[i] == k)
                        return i;
        }

        return -1;
}
```

In the above C program of linear search the index value of the matched element is returned if the key element is found in the list, otherwise -1 is returned by the user-defined function "Lsearch" to indicate that the key element is not found. The returned value is selected as -1, because -1 is an invalid index for an array in C and this does not coincide with a valid index which may be returned any time for a successful searching.

Time complexity of linear search – Time complexity of a linear search depends on the position where the key element is matched. There are two cases here to determine the time complexity, one is the time complexity for best case and another is the time complexity for worst case.

Best case: If the key element is found at the 1st position or the index 0 of the array, it consumes minimum time to complete the linear search procedure. That's why it becomes the best case. Now it requires only one comparison to find the key element in the list of numbers. Therefore the time complexity of linear search becomes O(1) for the best case.

Worst case: If the key element is found at the last position or (n – 1)th index of the array or the key element is not found in the list, it requires (n – 1) no. of comparisons to accomplish the linear search. Hence obviously it takes maximum time to complete the searching and time complexity becomes O(n) for the worst case.

Advantages of linear search:

1. It is very much easy to understand and implement.
2. Linear search can be applied on sorted as well as unsorted list.
3. A new element can be added at the last always in case of an unsorted array. This insertion does not require shifting of any element inside the array. Therefore insertion of a new element to an unsorted array does not affect the process of linear search.
4. Linear search can be implemented using both array and linked list.

Disadvantages of linear search:

1. As the time complexity of linear search is O(n) for worst case, it takes longer time to search an element placed at the end of the array. For example, if one comparison takes 1 unit of time, then linear search will take 1024 units of time to search an element placed at the last position of the unsorted array. Therefore linear search is inefficient for large datasets.

2) Binary Search – Binary search is a searching algorithm that is applicable only for sorted list i.e. a set of numbers either in ascending order or descending order. Consider a sorted list of numbers which are arranged in ascending order. Here the element at mid position of the list is compared with the key element. If the key element and the element at the mid position are equal, the position of that element in the list is displayed to indicate that the key element has been found in the list. If the key element is greater than the element, it can be said easily that the key element may be found only in the right sub-list after the mid-element as the right sub-list consists of the elements which are all larger than the mid-element. Again the mid-element from the right sub-list is compared with the key element. On the other hand if the key element becomes smaller than the mid-element, then it may be possible to find out the key element only in the left sub-list before the mid-element. Due to this reason the mid-element from the left sub-list is again compared with the key element. This process will continue until the key element becomes equal to the mid-element or the key element is not found in the sub-list of one element.

So, we can write in summary,

A set of n no. of elements are stored in a sorted array 'a' in ascending order.

MidIndex = (StartIndex + EndIndex) / 2 where StartIndex = 0 and EndIndex = n – 1

1. If key element = a[MidIndex], key element is found at position (MidIndex + 1)

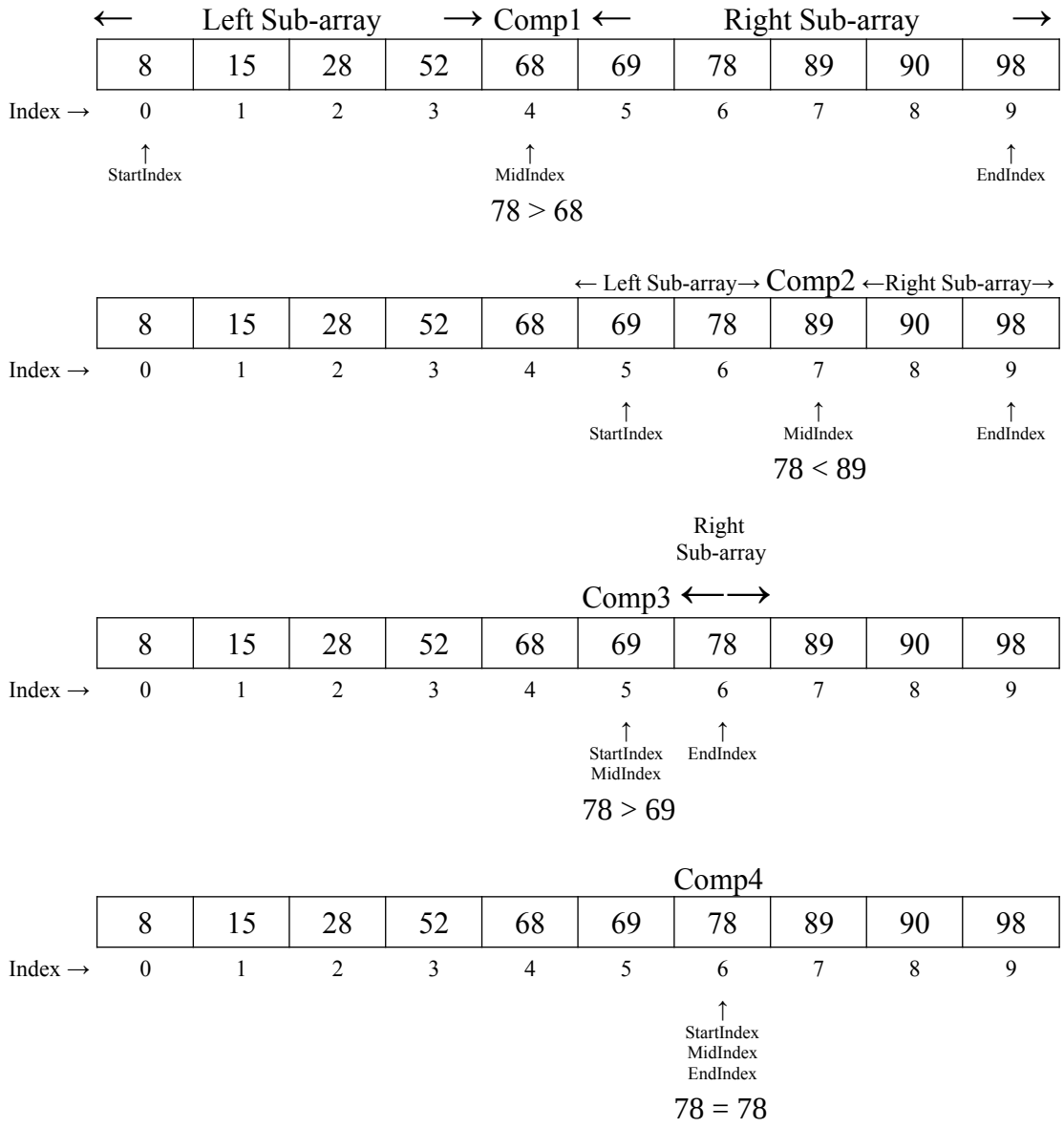
2. If key element > a[MidIndex], the key element is searched into the right sub-array in which StartIndex = MidIndex + 1 and EndIndex = n – 1.

3. If key element < a[MidIndex], the key element is searched into the left sub-array in which StartIndex = 0 and EndIndex = MidIndex - 1.

The procedure of binary search can be understood easily with the help of the following example.

Consider a list of ten elements 8, 15, 28, 52, 68, 69, 78, 89, 90, 98 are stored in a sorted array 'a' in ascending order and the key element which is to be searched is 78. Therefore the array 'a' starts from the index 0 and ends at the index 9. Here StartIndex = 0 and EndIndex = 9. So the MidIndex becomes (0 + 9)/2 i.e. 4. Now the procedure of binary search is shown step by step in Fig.7.2.

MidIndex = (StartIndex + EndIndex) / 2
Key element = 78



Key element 78 is found at index = 6 or position = 7

Fig.7.2: Searching process of binary search step by step

Algorithm of binary search:

Step 1: Start

Step 2: Input the no. of elements in a variable ‘n’.

Step 3: Input the list of n no. of elements into an array ‘a’.

Step 4: Input the key element in a variable ‘key’.

Step 5: Set Index = - 1

Step 6: Set StartIndex = 0

Step 7: Set Endndex = n – 1

Step 8: Repeat Step 9 to Step 11 while StartIndex ≤ EndIndex

Step 9: Set MidIndex = (StartIndex + EndIndex) / 2

Step 10: If a[MidIndex] = key, then
a) Set Index = MidIndex
Print “The key element is found at position Index + 1 ”.
Go to Step 11
[End of If (Step 10)]

Step 11: If a[MidIndex] > key, then
a) Set EndIndex = MidIndex – 1
Otherwise
b) Set StartIndex = MidIndex + 1
[End of If-Else (Step 11)]
[End of Loop (Step 8)]

Step 12: If Index = - 1, then
a) Print “The key element is not found”.
[End of If (Step 12)]

Step 13: Stop

Now the above mentioned algorithm of binary search can be implemented without using recursion or using recursion in C language. These two ways of implementations are given below.

C program to search an element from a set of numbers using binary search without using recursion.

```
#include<stdio.h>
int Bsearch(int [],int ,int );

int main()
{
	int a[50], i, n, key, result;

	printf("Enter the number of elements: ");
	scanf("%d", &n);
	printf("Enter the elements of the array:\n");
	for(i=0; i<n; i++)
	{
		printf("Enter Element%d: ", i+1);
		scanf("%d", &a[i]);
	}
	printf("Enter the value of the element to be searched: ");
	scanf("%d", &key);

	result = Bsearch(a, n, key);

	if(result == -1)
		printf("The element is not found.\n");
	else
		printf("The element is found at position %d\n", result + 1);

	return 0;
}

int Bsearch(int arr[],int len,int k)
{
	int i = 0, StartIndex, EndIndex, MidIndex;
	StartIndex =0;
	EndIndex = len - 1;
	while(StartIndex <= EndIndex)
	{
		MidIndex = (StartIndex + EndIndex) / 2;
		if(arr[MidIndex] == k)
			return MidIndex;
		if(arr[MidIndex] > k)
			EndIndex = MidIndex - 1;
		else
			StartIndex = MidIndex + 1;
	}
	return -1;
}
```

C program to search an element from a set of numbers using binary search using recursion.

```
#include<stdio.h>
int Bsearch(int [],int ,int );

int main()
{
	int a[50], i, n, key, result;

	printf("Enter the number of elements: ");
	scanf("%d", &n);
	printf("Enter the elements of the array:\n");
	for(i=0; i<n; i++)
	{
		printf("Enter Element%d: ", i+1);
		scanf("%d", &a[i]);
	}
	printf("Enter the value of the element to be searched: ");
	scanf("%d", &key);

	result = Bsearch(a, n, key);

	if(result == -1)
		printf("The element is not found.\n");
	else
		printf("The element is found at position %d\n", result + 1);

	return 0;
}

int Bsearch(int arr[],int StartIndex,int EndIndex,int k)
{
	int i = 0, MidIndex;
	if(StartIndex <= EndIndex)
	{
		MidIndex = (StartIndex + EndIndex)/2;

		if(arr[MidIndex] == k)
			return MidIndex;

		if(arr[MidIndex] > k)
			return Bsearch(arr, StartIndex, MidIndex -1, k);

		return Bsearch(arr, MidIndex + 1, EndIndex, k);
	}
	return -1;
}
```

Time complexity of binary search – Here we have to consider the two situations for time complexity – 1. best case and 2. worst case.

Best case: When the same value of the key element is placed at the mid position of the array, it will match at the first chance. This takes the minimum time to complete the binary search and gives the bast case of searching. Therefore the time complexity of binary search becomes O(1) for best case.

Worst Case: The binary search algorithm takes maximum time to be executed when the key element is found at last i.e. when the sub-array consists of one element and that element becomes equal to the key element. In addition to this there is another situation to get the worst case when the key element is not found in the list. In this case the binary search algorithm has to go till the last sub-list of only one element.

We know every sub-array accomplish one comparison between the key element and the mid-element. Now we have to determine how many arrays to be searched to get ultimately an array of one element. Suppose we shall reach to an array of one element after k no. searching.

SL.	No. of elements in array/ sub-arrays	No. of comparisons
1	No. of elements in 1^{st} array = n	1 comparison in 1^{st} array
2	No. of elements in 2^{nd} array ≈ n / 2	1 comparison in 2^{nd} array
3	No. of elements in 3^{rd} array ≈ n / 4 = n / 2^2	1 comparison in 3^{rd} array
4	No. of elements in 4^{th} array ≈ n / 8 = n / 2^3	1 comparison in 4^{th} array
: :	: :	: :
k	No. of elements in k^{th} array ≈ n / 2^{k-1}	1 comparison in k^{th} array
		Total no. of comparisons = k

Now we know,

No. of elements in k^{th} array = 1 = n / 2^{k-1}

or, 2^{k-1} = n

or, k – 1 = $\log_2$ n

$\therefore$ k = $\log_2$ n + 1

Total no. of comparisons required for binary search in worst case = k = $\log_2$ n + 1

$\therefore$ Time complexity of binary search for worst case = O($\log_2$ n)

Advantages of binary search:

1. The time complexity of a binary search is O($\log_2$ n) which makes the binary search efficient for large set of data. If we want to search a data in a set of 1024 data using binary search, it requires only 10 comparisons in worst case where as linear search would perform 1024 comparisons in worst case. Therefore it is clear that binary search is more time efficient that linear search for large data-set.

Disadvantages of binary search:

1. Binary search only works on sorted arrays. If the array is not sorted, it must be sorted before the application of binary search. Therefore binary search is not suitable for unsorted list.
2. Binary search can be implemented by using array only, it can not be implemented with the help of linked list.
3. As the list must be sorted before the use of binary search, insertion of a new element into a sorted array requires shifting of elements. Therefore insertion of element affect the performance of binary search.

Comparative study between linear search and binary search:

SL.	Linear search	Binary search
1	Time complexity is O(n) for worst case and O(1) for best case.	Time complexity is O($\log_2$ n) for worst case and O(1) for best case.
2	It is applicable for both sorted and unsorted list.	It is only applicable for sorted list, but not applicable for unsorted list.
3	It can be implemented by using both array and linked list.	It can be implemented only by using array, but not for linked list.
4	It can not be applied for large set of data.	It can be applied for large set of data.
5	Insertion of element does not affect the performance of linear search.	Insertion of element affects the performance of binary search significantly.
6	The algorithm of linear search is very easy to understand and implement	The algorithm of binary search is cumbersome to implement compared to linear search.

Sorting – Sorting is a method by which a list of numerical values or alphanumerical values are arranged in ascending or descending order in an efficient way. After sorting the list of values is called a sorted list. Consider an array holding a set of ten values as given below.

	29	45	85	10	7	36	63	50	15	23
Index →	0	1	2	3	4	5	6	7	8	9

After sorting the above list in ascending order it will be a sorted list as given below.

	7	10	15	23	29	36	45	50	63	85
Index →	0	1	2	3	4	5	6	7	8	9

If this list is sorted in descending order the sorted list will be given as follows.

	85	63	50	45	36	29	23	15	10	7
Index →	0	1	2	3	4	5	6	7	8	9

Efficient sorting algorithms are used to optimize the use of other algorithms like search and merge algorithms which requires sorted list to work correctly. We know that the binary search can only be applied on a list if the list is sorted. Therefore the list must be arranged in ascending or descending order by using an efficient sorting algorithm in order to use the binary search algorithm on it. Depending upon the use of internal and external memory in computer there are two types of sorting, 1) Internal sorting and 2) External sorting.

1) Internal sorting – If the data sorting process takes place entirely into the internal memory or main memory (basically RAM) of a computer, it is called internal sorting. This is only possible if the size of a data to be sorted is small enough to fit into the main memory of the computer. Some examples of internal sorting are bubble sort, insertion sort, selection sort, quick sort, heap sort etc.

2) External sorting – If the data set to be sorted is massive in size, it is not possible to accommodate the entire data set into the internal memory or main memory of the computer and the rest of the data set is stored into the slower external memory (usually a hard drive) of the computer. This sorting process is called external sorting.

External sorting uses a hybrid sort-merge strategy which is basically composed of two phases, one is sort phase and another is merge phase. In sort phase chunks of data which are small enough to fit into the main memory are transferred to the main memory, then these small chunks of data are sorted in the main memory and finally written back to a temporary file into the external memory. In the merge phase these sorted sub-files are combined or merged into a sorted single larger file. One example of external sorting is merge sort.

Comparative study between internal sort and external sort:

SL.	Internal sort	External sort
1	All the data to be sorted is stored in main memory or internal memory of the computer at all times while sorting is in progress.	The entire data set in this case is sorted using internal as well as external memory of the computer. Sorting is done into the internal memory and merging is done into the external memory.
2	It is performed when the data to be sorted is small enough to fit into the main memory.	External sorting is usually applied when data is too large to fit into the main memory.
3	In this case the storage device used is only main memory (RAM) of the computer	In this case the storage devices are both external memory (hard drive) and the internal memory (RAM) of the computer.
4	Examples: bubble sort, insertion sort, selection sort, quick sort, heap sort etc.	Example: merge sort.

Depending upon how much space is used by a sorting algorithm there are two types of sorting. 1) In-place sorting and 2) Non-in-place sorting

1) In-place sorting – An in-place sorting algorithm uses constant memory space for producing the output i.e. the sorted data set. It sorts the list only by modifying the order of the elements within the list. Examples of in-place sorting algorithm are bubble sort, insertion sort, selection sort, quick sort etc as they do not use any additional space for sorting the whole list of data.

2) Non-in-place sorting – On the contrary in case of non-in-place sorting the sorting algorithm uses additional memory space which usually increases with the increase of input size i.e. with the increase of input data set. So the auxiliary space complexity of a non-in-place sorting becomes O(n) where n is the number of elements in the input data set. Examples of non-in-place sorting are merge sort and counting sort. In case of merge sort an additional array is used to merge all the data into a sorted one. Due to the use of an extra array space merge sort is a non-in-place sorting.

Depending upon the stability of the sorting algorithm there are two types namely 1) Stable sort and 2) Unstable sort.

1) Stable sort – After applying a sorting technique if two same elements appear in the same order as the original unsorted list without changing their positions, then the sorting algorithm is called stable sort. It implies that between the two same elements the element which placed first in the original list will be placed first in the sorted list and the element which comes second position will be placed just after the first element. Examples of some stable sorts are *bubble sort, insertion sort, merge sort* etc.

2) Unstable sort – After applying a sorting algorithm if two same elements interchange their positions in the sorted list i.e. first element is placed after the second element in the sorted list, then the sorting algorithm is known as unstable sort. *Selection sort* and *quick sort* are the examples of two unstable sorts.

Let's take an example to understand the stable and unstable sort clearly. Suppose we have an unsorted list of ten elements like 11, 25, 15, 19, 7, 48, 39, 15, 17, 20. Here we can observe the two same elements are 15. To clarify the positions of these two elements the first 15 is designated as 15_A and the second 15 as 15_B. Now the original unsorted list looks like 11, 25, 15_A, 19, 7, 48, 39, 15_B, 17, 20. After the application of a sorting algorithm SORTA if the sorted list will be 7, 11, 15_A, 15_B, 17, 19, 20, 25, 39, 48 where first 15 (designated as 15_A) and the second 15 (designated as 15_B) are placed first and second respectively in the sorted list maintaining the same order of original list, then SORTA will be a stable sort. If the same unsorted list is sorted with the help of another sorting technique SORTB which gives a sorted list 7, 11, 15_B, 15_A, 17, 19, 20, 25, 39, 48 where the two same elements 15_A and 15_B have interchanged their positions (15_B becomes first and 15_A becomes second), then SORTB will be an unstable sort.

Importance of the stability of a sorting algorithm – Now the common query may come in our mind "Why do we need to analyze the stability of a sorting algorithm ?". For this purpose we may consider the following practical example.

Suppose in a examination hall an invigilator is collecting the answer scripts of some students. A ranking list will be published after the examination depending upon their obtained marks. In the stack of the answer scripts collected by the invigilator the student who comes at the bottom of the stack has submitted his answer script first, then the student after that student in the stack has submitted the answer script second and so on. Now if two students score the same marks what will be the ranking of them. Here the student whose answer script is kept below the stack has submitted the answer script before the student who has obtained the same marks. Therefore the student below the stack will be ranked first. After evaluating the answer scripts the examiner will arrange the answer scripts in descending order using a sorting technique depending upon the obtained marks and the sorting technique used by the examiner must be a stable sort to publish the ranks correctly, otherwise those two students with same marks may be ranked wrong. Thus the sorted stack in descending order gives the 1^{st} rank for the topmost answer script, 2^{nd} rank for the answer script placed just below the topmost answer script and so on.

Before explaining the different sorting techniques some practical considerations of internal sorting should be focused to analyze the performance of the sorting algorithms.

Sorting on multiple keys – Many times when performing real-world applications, it is desired to sort the records of multiple fields such as the records of some students in a college with multiple fields like Student Name, Roll No., Department, Phone No. etc. If these records are to be sorted according to their departments and name, then after successful sorting the departments will be in ascending order and under each department the students will be sorted depending upon their names. In this case the records should be sorted using the field 'Department' first, then again all the records under each department will be sorted as per their names. Thus the first field 'Department' here is called the primary sort key and the second field 'Student Name' will be the secondary sort key. Consider the following example.

Unsorted records of five students

Student Name	Roll No.	Department	Phone No.
Ritam	ECE/23/01	ECE	8240566955
Bipasha	CSE/23/11	CSE	9432987654
Debatma	ECE/23/48	ECE	9836123456
Anusree	CSE/23/50	CSE	7094235128
Sreejan	ECE/23/16	ECE	9433554433

After sorting on primary key 'Department'

Student Name	Roll No.	Department	Phone No.
Bipasha	CSE/23/11	CSE	9432987654
Anusree	CSE/23/50	CSE	7094235128
Ritam	ECE/23/01	ECE	8240566955
Debatma	ECE/23/48	ECE	9836123456
Sreejan	ECE/23/16	ECE	9433554433

After sorting on secondary key 'Student Name'

Student Name	Roll No.	Department	Phone No.
Anusree	CSE/23/50	CSE	7094235128
Bipasha	CSE/23/11	CSE	9432987654
Debatma	ECE/23/48	ECE	9836123456
Ritam	ECE/23/01	ECE	8240566955
Sreejan	ECE/23/16	ECE	9433554433

Observe that the records are sorted based on department. However, within each department the records are sorted alphabetically based on the names of the students.

We know the elements to be sorted are stored in an array normally. But in case of sorting the records with multiple fields it is better to store the records in a linked list where each node of the linked list will hold these multiple fields.

Practical considerations of sorting – While analyzing the performance of different sorting algorithms, we should consider the following practical considerations.

1. Number of sort keys based on which sorting will be performed.

2. Number of times the records will be moved to get the final sorted list.

3. The sorting algorithm used is in-place sorting or not to determine that the extra space will be required or not.

4. Best case time complexity to analyze the performance of the sorting algorithm.

5. Worst case time complexity to analyze the performance of the sorting algorithm and this parameter is most important to measure the faster performance of the sorting technique for a large data set.

6. Average case time complexity to analyze the performance of the sorting algorithm.

7. Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done.

Now six different sorting techniques – bubble sort, selection sort, insertion sort, quick sort, merge sort and heap sort will be discussed one by one in the next section.

1) Bubble sort – Bubble sort is a very simple method that sorts the elements in ascending order by repeatedly moving the largest element to the highest index position of the array segment after each pass. In bubble sort, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are swapped. This process will continue till the list of unsorted elements exhausts. Note that at the end of the first pass, the largest element in the list will be placed at the end which is the proper position for the largest element in the sorted list. Therefore the largest element placed at the end of the list is not required to consider during the second pass and the second pass will be done with (n – 1) no. of elements. After completion of the second pass the second largest element will be placed at its proper position, just before the largest element. Similar case will occur during the third pass which will place the third largest element just before the second largest element. This process will continue until we get a pass where no swapping takes place between pair of consecutive elements or a pass with only two elements. The entire process of the bubble sort may be better understood with following example which is shown step by step in Fig.7.3.

Pass 1

Comp1

29	45	85	7	10	36
0	1	2	3	4	5

Index →

29 < 45 → No Swap

Comp2

29	45	85	7	10	36
0	1	2	3	4	5

Index →

45 < 85 → No Swap

Comp3

29	45	85	7	10	36
0	1	2	3	4	5

Index →

85 > 7 → Swap

Comp4

29	45	7	85	10	36
0	1	2	3	4	5

Index →

85 > 10 → Swap

Comp5

29	45	7	10	85	36
0	1	2	3	4	5

Index →

85 > 36 → Swap

29	45	7	10	36	85
0	1	2	3	4	5

Index →

Largest element 85 is placed at its proper position 6

Pass 2

Comp1

29	45	7	10	36	85
0	1	2	3	4	5

Index →

29 < 45 → No Swap

Comp2

29	45	7	10	36	85
0	1	2	3	4	5

Index →

45 > 7 → Swap

Pass 2

Comp3

	29	7	45	10	36	85
Index →	0	1	2	3	4	5

45 > 10 → Swap

Comp4

	29	7	10	45	36	85
Index →	0	1	2	3	4	5

45 > 36 → Swap

	29	7	10	36	45	85
Index →	0	1	2	3	4	5

Second largest element 45 is placed at its proper position 5

Pass 3

Comp1

	29	7	10	36	45	85
Index →	0	1	2	3	4	5

29 > 7 → Swap

Comp2

	7	29	10	36	45	85
Index →	0	1	2	3	4	5

29 > 10 → Swap

Comp3

	7	10	29	36	45	85
Index →	0	1	2	3	4	5

29 < 36 → No Swap

	7	10	29	36	45	85
Index →	0	1	2	3	4	5

Third largest element 36 is placed at its proper position 4

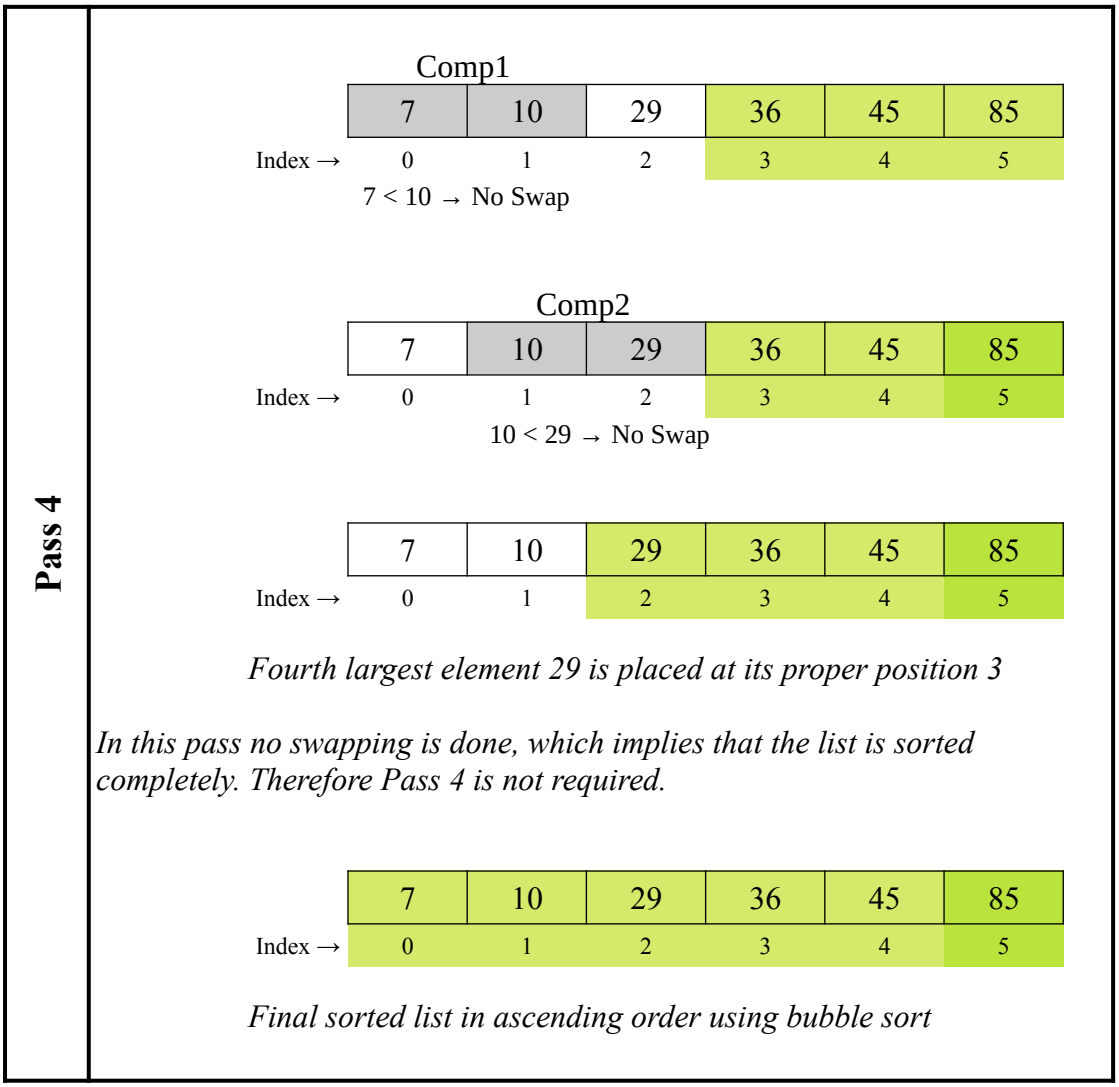


Fig.7.3: Step by step execution of bubble sort

From Fig.7.3 it is evident that if there would be at least one swapping in Pass 4, one more pass would be executed to complete the bubble sort. That means in normal case 5 passes should be executed for an unsorted list with 6 elements. Finally we can conclude the following points in case of a bubble sort.

- There will be (n – 1) no. of passes in a bubble sort for an unsorted list with n no. of elements.
- If no swapping happens in any pass during the execution of the bubble sort, it implies that the list is already sorted. Hence no pass is required to be executed further. In that case no. of passes will be less than (n – 1).
- Every comparison is done between a pair of adjacent elements. If the element at the lower index is larger than the element at the higher index, the two elements are swapped.

- There will be (n – 1) no. of comparisons during 1st pass, (n – 2) comparisons during 2nd pass, (n – 3) comparisons during 3rd pass and so on. Therefore 1 comparison will be conducted during the (n – 1)th pass in the bubble sort.

- After completion of every pass the length of the unsorted list is decremented by one and the largest element in the segment of the array during that pass will be placed at the end which is the proper position of that element in the final sorted list.

To implement bubble sort we have to use two nested loops, the outer loop for counting the number of passes and the inner loop for performing comparisons in a particular pass. In addition to this a swap_flag is used to track whether at least one swapping happens or not. The algorithm of bubble sort is given below.

Algorithm of bubble sort:

Step 1: Start

Step 2: Input the no. of elements in a variable ‘n’.

Step 3: Input the unsorted list of n no. of elements into an array ‘a’.

Step 4: Set i = 1

Step 5: Repeat Step 6 to Step 12 while i < n

Step 6: Set SwapFlag = 0

Step 7: Set j = 0

Step 8: Repeat Step 9 to Step 10 while j < n – 1

Step 9: If a[j] > a[j+1], then
a) Set temp = a[j]
b) Set a[j] = a[j+1]
c) Set a[j+1] = temp
[End of If (Step 9)]

Step 10: Set j = j + 1
[End of Loop (Step 8)]

Step 11: If SwapFlag = 0, then
a) Go to Step 13
[End of If (Step 11)]

Step 12: Set i = i + 1
[End of Loop (Step 5)]

Step 13: Stop

C program to arrange a set of numbers in ascending order using bubble sort.

```
#include<stdio.h>
void BubbleSort(int [],int );

int main()
{
	int a[50], i = 0, n;

	printf("Enter the number of elements: ");
	scanf("%d", &n);
	printf("Enter the elements of the array:\n");
	for(i=0; i<n; i++)
	{
		printf("Enter Element%d: ", i+1);
		scanf("%d", &a[i]);
	}
	BubbleSort(a, n);

	printf("Sorted List Using Bubble Sort: ");
	for(i=0; i<n; i++)
		printf("%d ", a[i]);

	printf("\n");
	return 0;
}

void BubbleSort(int arr[],int len)
{
	int i, j, temp, SwapFlag;

	for(i=1; i<len; i++)
	{
		SwapFlag = 0;
		for(j=0; j<len-i; j++)
		{
			if(arr[j] > arr[j+1])
			{
				temp = arr[j];
				arr[j] = arr[j+1];
				arr[j+1] = temp;
				SwapFlag = 1;
			}
		}
		if(SwapFlag == 0)
			break;
	}
}
```

Time complexity of bubble sort – To calculate the time complexity of bubble sort we have to consider the number of times comparisons happened to complete the bubble sort in best case and worst case.

Best case: In best case bubble sort takes minimum time to be executed and this situation happens when the list is already in sorted order. When this situation arises, only one pass (only 1st pass) will be executed to complete the bubble sort. We know (n – 1) no. of comparisons will be accomplished to complete the 1st pass in a bubble sort. Therefore time complexity of the bubble sort will be O(n) for best case.

Worst case: Worst case will happen when the unsorted list is in descending order and the list will have to be sorted in ascending order using bubble sort. In this situation there will be (n – 1) no. of passes and the swapping will happen between two adjacent elements for every comparison.

For 1st pass, no. of comparisons = n – 1

For 2nd pass, no. of comparisons = n – 2

For 3rd pass, no. of comparisons = n – 3

In this way no. of comparisons for $(n-1)^{th}$ pass = 1

$\therefore$ Total no. of comparisons = (n – 1) + (n – 2) + (n – 3) + …… + 1

$$= 1 + 2 + 3 + \ldots\ldots\ldots\ldots + (n-2) + (n-1)$$

$$= \frac{(n-1)(n-1+1)}{2}$$

$$= \frac{n(n-1)}{2} = \frac{(n^2-n)}{2}$$

Therefore the time complexity of a bubble sort becomes O(n^2) for worst case.

Average case: The time complexity of bubble sort in average case is O(n^2).

In-place sorting – Bubble sort does not require any extra space. That's why it is an in-place sorting.

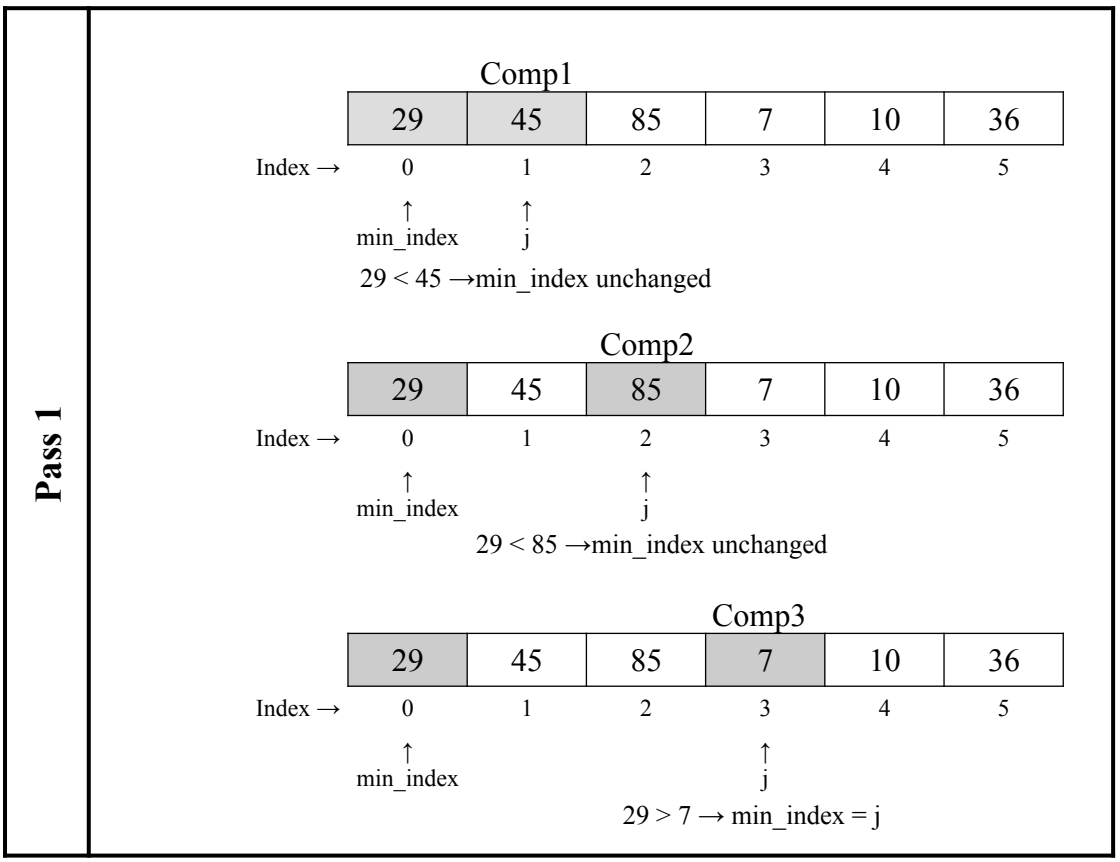
Stability of bubble sort – Bubble sort is a stable sorting technique.

Advantages of bubble sort:
1. It is easy to understand and implement.
2. It does not require extra memory space which makes the space complexity O(1).
3. It is a stable sorting algorithm.

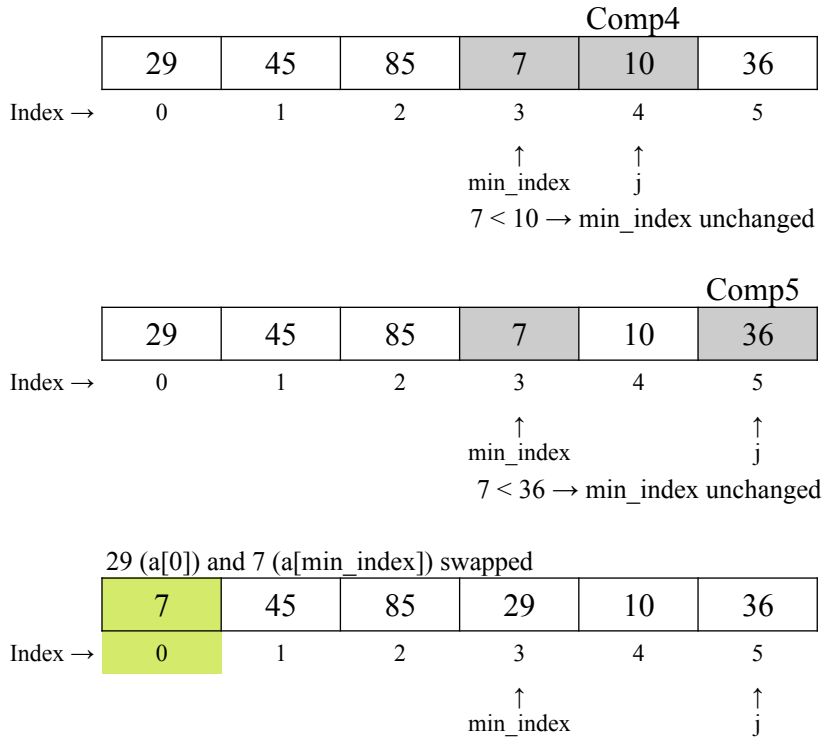
Disadvantages of bubble sort:

1. Selection sort has time complexity of O(n^2) in worst case and average case. This makes the bubble sort slower than other sorting techniques. Due to this reason bubble sort should not be used in large data set.
2. During the execution of bubble sort multiple number of swapping between the adjacent elements may happen. For every swapping memory write operation is performed which makes this sorting slower.

2) Selection sort – In this sorting after completion of every pass the minimum element from the list will be placed at the first position. As the first position is the correct position for the minimum element in a list, the selection sort puts the minimum element at its right position in this way. However the minimum element from the entire unsorted list will be placed at the first position (index = 0) of the array after the 1st pass. During the 2nd pass the first element is excluded and the minimum element from the remaining (n – 1) no. of elements of the array is selected and it will be placed at the 2nd position (index = 1) of the array which becomes the proper position of it. Thus the several passes are executed in selection sort to result the final sorted list in ascending order. If it is desired to sort the list in descending order using the selection sort, each time the minimum element will be placed at the end of the list. To determine the minimum element in every pass we have to find out the index of the minimum element by using a variable min_index. Then the first element and the element at min_index will be swapped to place the minimum element at the 1st position. The process of the selection sort has been demonstrated step by step in Fig.7.4 with the help of an example.

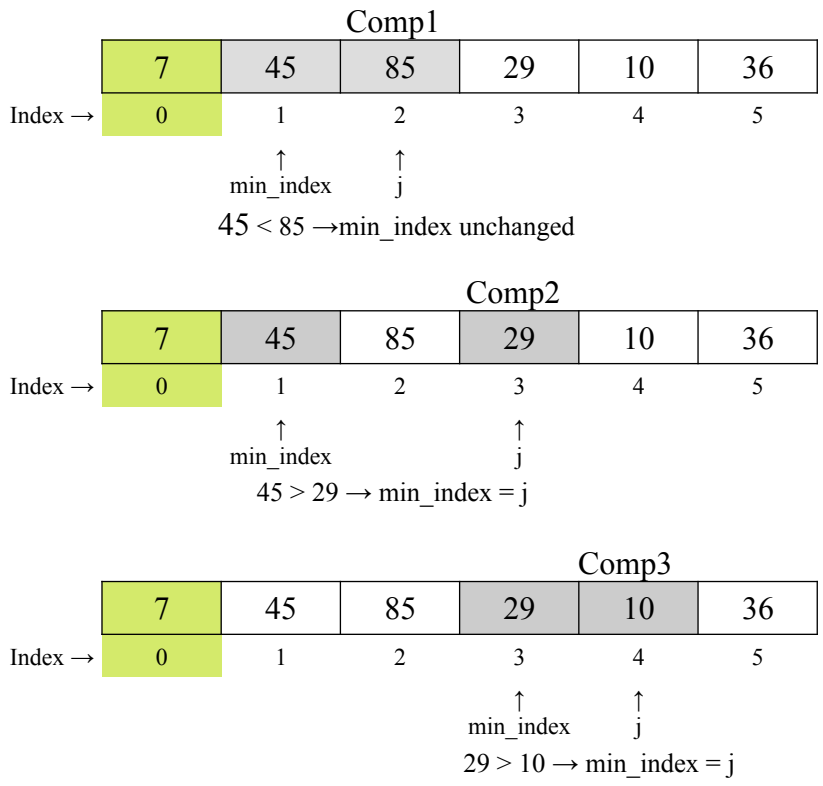


Pass 1

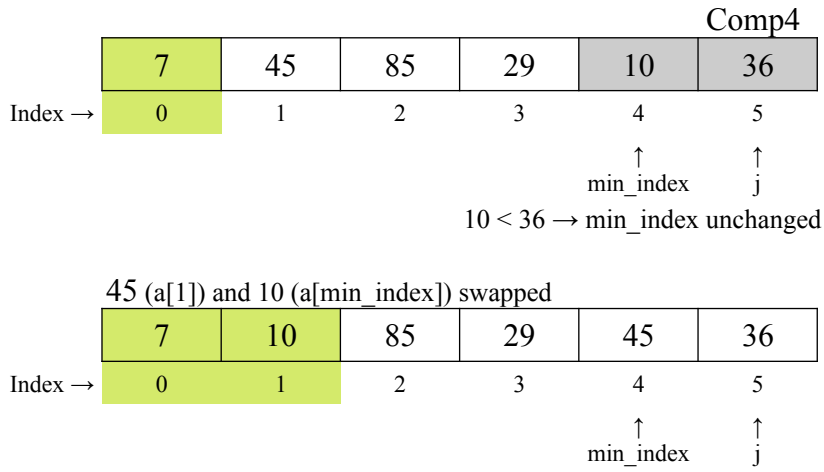


Smallest element 7 is placed at its proper position 1 (index = 0)

Pass 2

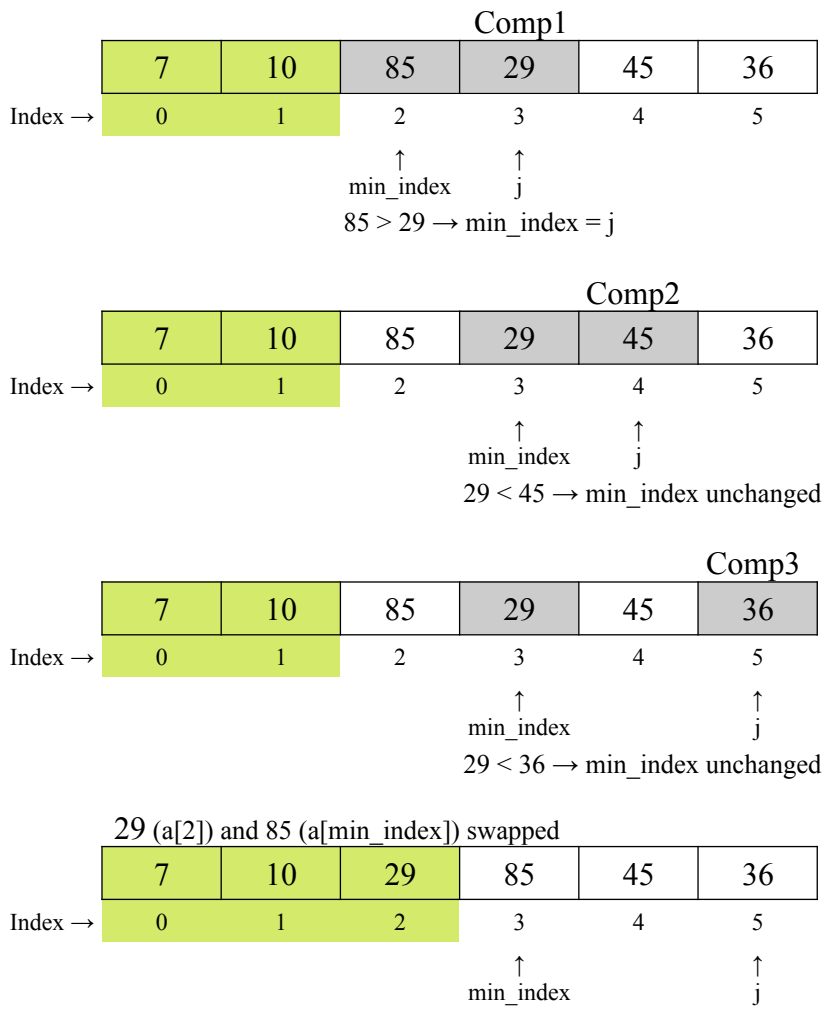


Pass 2



Second smallest element 10 is placed at its proper position 2 (index = 1)

Pass 3



Third smallest element 29 is placed at its proper position 3 (index = 2)

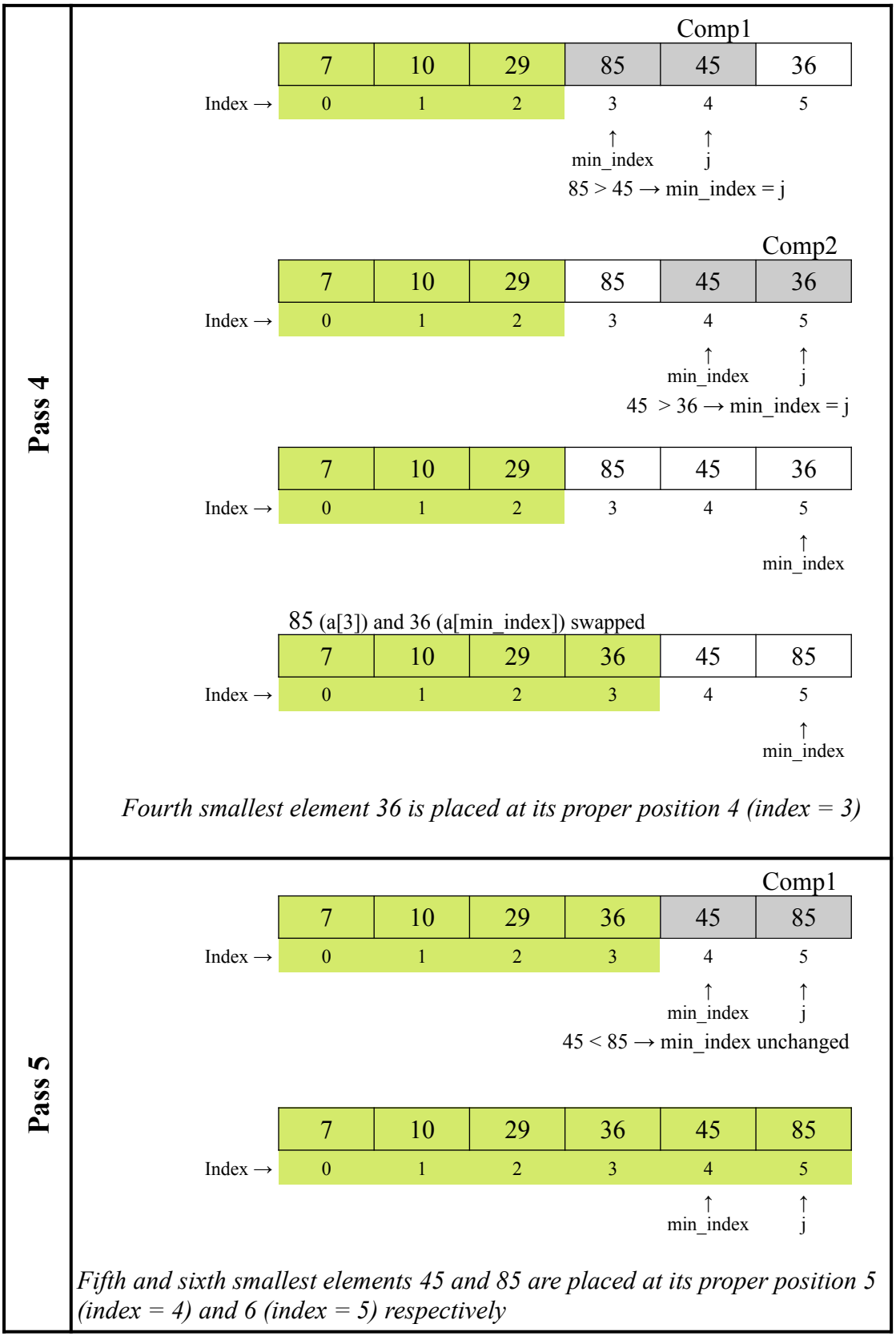


Fig.7.4: Step by step execution of selection sort

Observing the sequences of selection sort in Fig.7.4 we can conclude the following important points regarding the selection sort.

- There will be always (n – 1) no. of passes in a selection sort for an unsorted list with n no. of elements. Like bubble sort there is no chance of reduction of no. of passes.

- The variable min_index is taken for holding the minimum element in the list in case of selection sort.

- Every comparison is done between a pair of elements which may be adjacent or may not be adjacent. If the element at the lower index is larger than the element at the higher index, the min_index will be shifted to the index of smaller element.

- There will be (n – 1) no. of comparisons during 1^{st} pass, (n – 2) comparisons during 2^{nd} pass, (n – 3) comparisons during 3^{rd} pass and so on. Therefore 1 comparison will take place during the $(n – 1)^{th}$ pass in the selection sort.

- After completion of every pass the smallest element in the segment of the array during that pass will be placed at the 1^{st} position which is the proper position of that element in the final sorted list.

- Unlike bubble sort there may be only one swapping at the end of every pass. At the end of every pass the minimum element is pointed by min_index and a swapping may be done between that minimum element and the first element of that array segment in case of selection sort.

To implement selection sort we require two nested loops, the outer loop for counting the number of passes and the inner loop for performing comparisons in a particular pass. The algorithm of selection sort is given below.

Algorithm of selection sort:

Step 1: Start

Step 2: Input the no. of elements in a variable ‘n’.

Step 3: Input the unsorted list of n no. of elements into an array ‘a’.

Step 4: Set i = 0

Step 5: Repeat Step 6 to Step 12 while i < n – 1

Step 6: Set min_index = i

Step 7: Set j = i + 1

Step 8: Repeat Step 9 to Step 10 while j < n

Step 9: If a[min_index] > a[j], then
a) Set min_index = j
[End of If (Step 9)]

Step 10: Set j = j + 1
[End of Loop (Step 8)]

Step 11: If min_index ≠ i, then
a) Set temp = a[min_index]
b) Set a[min_index] = a[i]
c) Set a[i] = temp
[End of If (Step 11)]

Step 12: Set i = i + 1
[End of Loop (Step 5)]

Step 13: Stop

C program to arrange a set of numbers in ascending order using selection sort.

```
#include<stdio.h>
void SelectionSort(int [],int );

int main()
{
        int a[50], i = 0, n;

        printf("Enter the number of elements: ");
        scanf("%d", &n);

        printf("Enter the elements of the array:\n");
        for(i=0; i<n; i++)
        {
                printf("Enter Element%d: ", i+1);
                scanf("%d", &a[i]);
        }

        SelectionSort(a, n);

        printf("Sorted List Using Selection Sort: ");
        for(i=0; i<n; i++)
                printf("%d ", a[i]);

        printf("\n");
        return 0;
}
```

```
void SelectionSort(int arr[],int len)
{
        int i, j, min_index, temp;

        for(i=0; i<len-1; i++)
        {
                min_index = i;
                for(j=i+1; j<len; j++)
                        if(arr[min_index] > arr[j])
                                min_index = j;

                if(min_index != i)
                {
                        temp = arr[min_index];
                        arr[min_index] = arr[i];
                        arr[i] = temp;
                }
        }
}
```

Time complexity of selection sort – To calculate the time complexity of selection sort we should calculate the number of times comparisons happened to complete the selection sort in best case and worst case. In this case it is not possible to say in advance that the list is sorted in ascending order even for a sorted list. That's why selection sort always completes (n – 1) no. of passes for a list of n no. elements.

For 1^{st} pass, no. of comparisons = n – 1

For 2^{nd} pass, no. of comparisons = n – 2

For 3^{rd} pass, no. of comparisons = n – 3

In this way no. of comparisons for $(n-1)^{th}$ pass = 1

$\therefore$ Total no. of comparisons = (n – 1) + (n – 2) + (n – 3) + …… + 1

= 1 + 2 + 3 + …….. + (n – 2) + (n – 1)

$$= \frac{(n-1)(n-1+1)}{2}$$

$$= \frac{n(n-1)}{2} = \frac{(n^2-n)}{2}$$

Therefore the time complexity of a selection sort becomes $O(n^2)$ always. That means, the time complexity of selection sort is $O(n^2)$ for **best case**, **worst case** and **average case**.

In-place sorting – Selection sort does not require any extra space. Only it requires an extra variable (in this case 'temp') to swap two elements. That's why it is an in-place sorting.

Stability of selection sort – Selection sort is an unstable sorting because sometimes two equal elements interchange their positions in the sorted list. This situation can be best understood with the help of the following example.

	42	25_A	65	12	10	19	25_B	20
Index →	0	1	2	3	4	5	6	7

Unsorted Array with 8 elements

In the above list the two same numbers are 20_A and 20_B where suffix A and suffix B shows that the number 20_A is placed before 20_B.

	10	25_A	65	12	42	19	25_B	20
Index →	0	1	2	3	4	5	6	7

After completion of Pass 1

	10	12	65	25_A	42	19	25_B	20
Index →	0	1	2	3	4	5	6	7

After completion of Pass 2

	10	12	19	25_A	42	65	25_B	20
Index →	0	1	2	3	4	5	6	7

After completion of Pass 3

	10	12	19	20	42	65	25_B	25_A
Index →	0	1	2	3	4	5	6	7

After completion of Pass 4

	10	12	19	20	25_B	65	42	25_A
Index →	0	1	2	3	4	5	6	7

After completion of Pass 5

	10	12	19	20	25_B	25_A	42	65
Index →	0	1	2	3	4	5	6	7

After completion of Pass 6

	10	12	19	20	**25_B**	**25_A**	42	65
Index →	0	1	2	3	4	5	6	7

After completion of Pass 7 the final sorted list using selection sort

In the final sorted array it is being noticed that 20_B is placed before 20_A i.e. the positions of two equal elements have been interchanged after the execution of the selection sort. This clearly states that the selection sort is an unstable sorting technique.

Advantages of selection sort:
1. It is easy to understand and implement.
2. It requires constant memory space which makes the space complexity O(1).
3. It requires less number of swapping i.e. less number of memory writes. Therefore selection may the best choice when memory writes become costly.

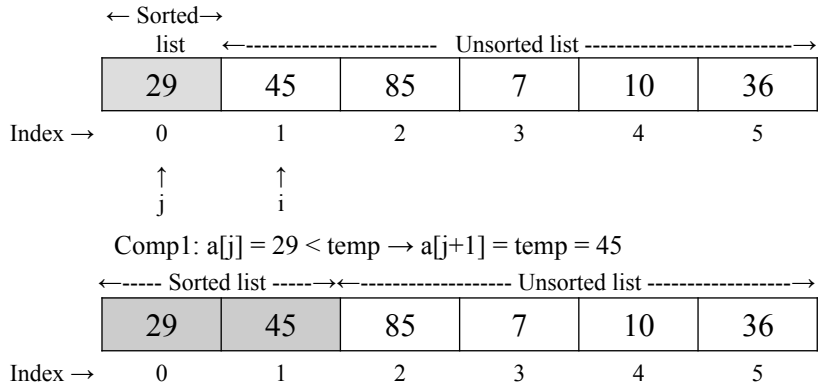
Disadvantages of selection sort:
1. Selection sort has time complexity of $O(n^2)$ always for best as well as worst case. This makes the selection sort slower compared to quick sort, merge sort etc.
2. This sorting technique is unstable in nature which may change the relative positions of the equal elements.

3) Insertion sort – Here initially a sorted list is formed with the first element and the remaining elements are considered to form an unsorted list. Thus the entire unsorted list is divided into one sorted list and one unsorted list. Now each and every element from the unsorted list is inserted into the sorted list in its proper position so that the sorted list remains sorted in ascending order. After completion of this method all the elements from the unsorted list occupy their proper position in the sorted list. As a result we get the entire sorted list by inserting elements from the unsorted list one by one. As the elements are inserted from the unsorted list into the sorted list, this process is called insertion sort.

During the first pass the 1^{st} element forms the sorted list and the remaining elements (2^{nd} element to n^{th} element) form the unsorted list. Now the 2^{nd} element from the unsorted list is compared with the 1^{st} element in the sorted list. If 1^{st} element is larger than the 2^{nd} element, then the 2^{nd} element will be placed before the 1^{st} element, otherwise the position of 2^{nd} element remains unchanged. After the first pass the sorted list will be expanded to include the 2^{nd} element into its proper position whereas the unsorted list will be reduced by one element i.e. 2^{nd} element. At the starting of second pass the third element will be compared with all the elements inside the sorted list and placed at its proper position. Therefore after the completion of the second pass the third element will be included into the sorted list and excluded from the unsorted list, which results the expansion of the sorted list and contraction of the unsorted list. This process will continue until the unsorted list be exhausted. The process of insertion sort can be understood clearly with the following example shown in Fig.7.5.

Pass 1

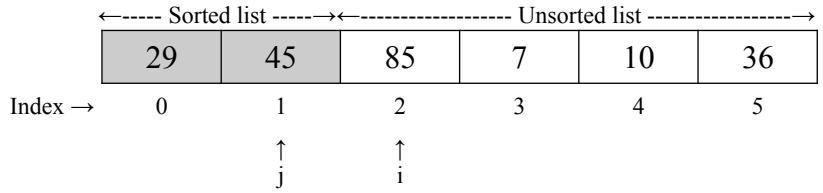
i = 1 and temp = a[i] = 45



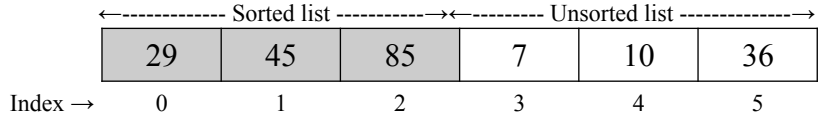
Comp1: a[j] = 29 < temp → a[j+1] = temp = 45

Pass 2

i = 2 and temp = a[i] = 85

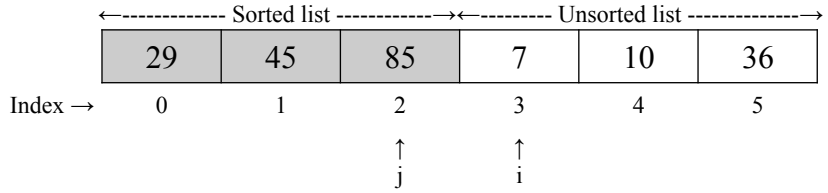


Comp1: a[j] = 45 < temp → a[j+1] = temp = 85

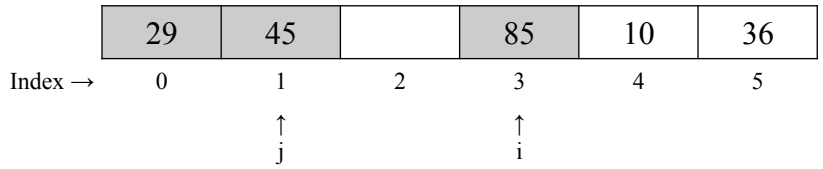


Pass 3

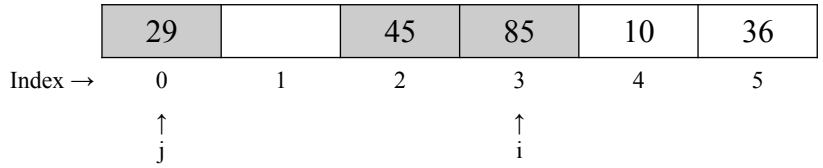
i = 3 and temp = a[i] = 7



Comp1: a[j] = 85 > temp → 85 will shift one position right

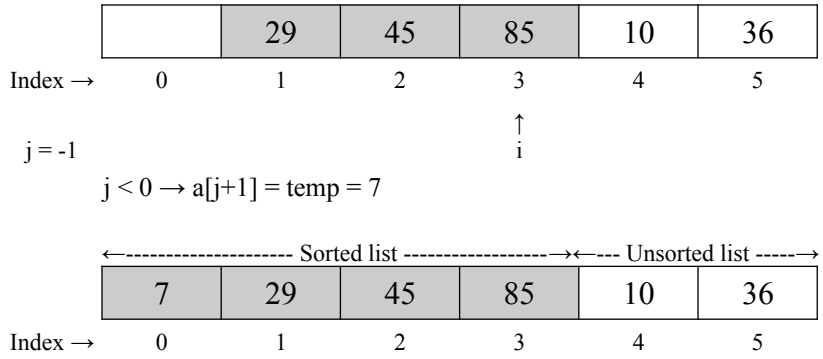


Comp2: a[j] = 45 > temp → 45 will shift one position right



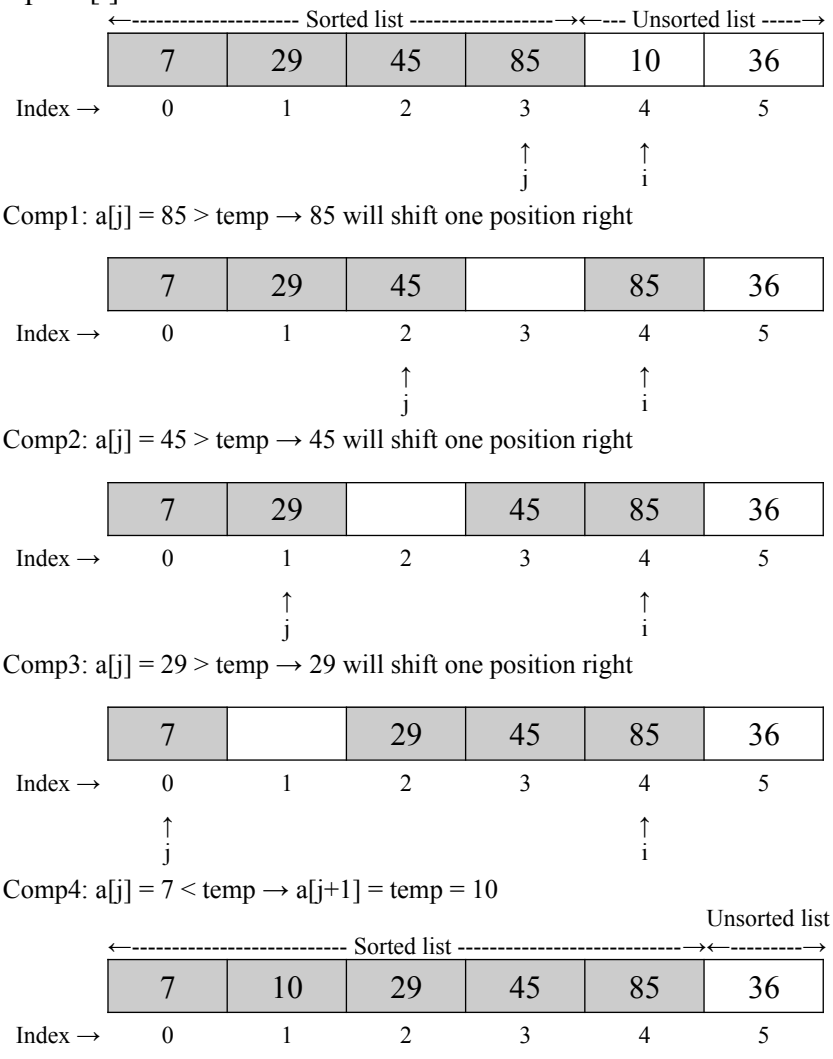
Comp3: a[j] = 29 > temp → 29 will shift one position right

Pass 3



Pass 4

i = 4 and temp = a[i] = 10



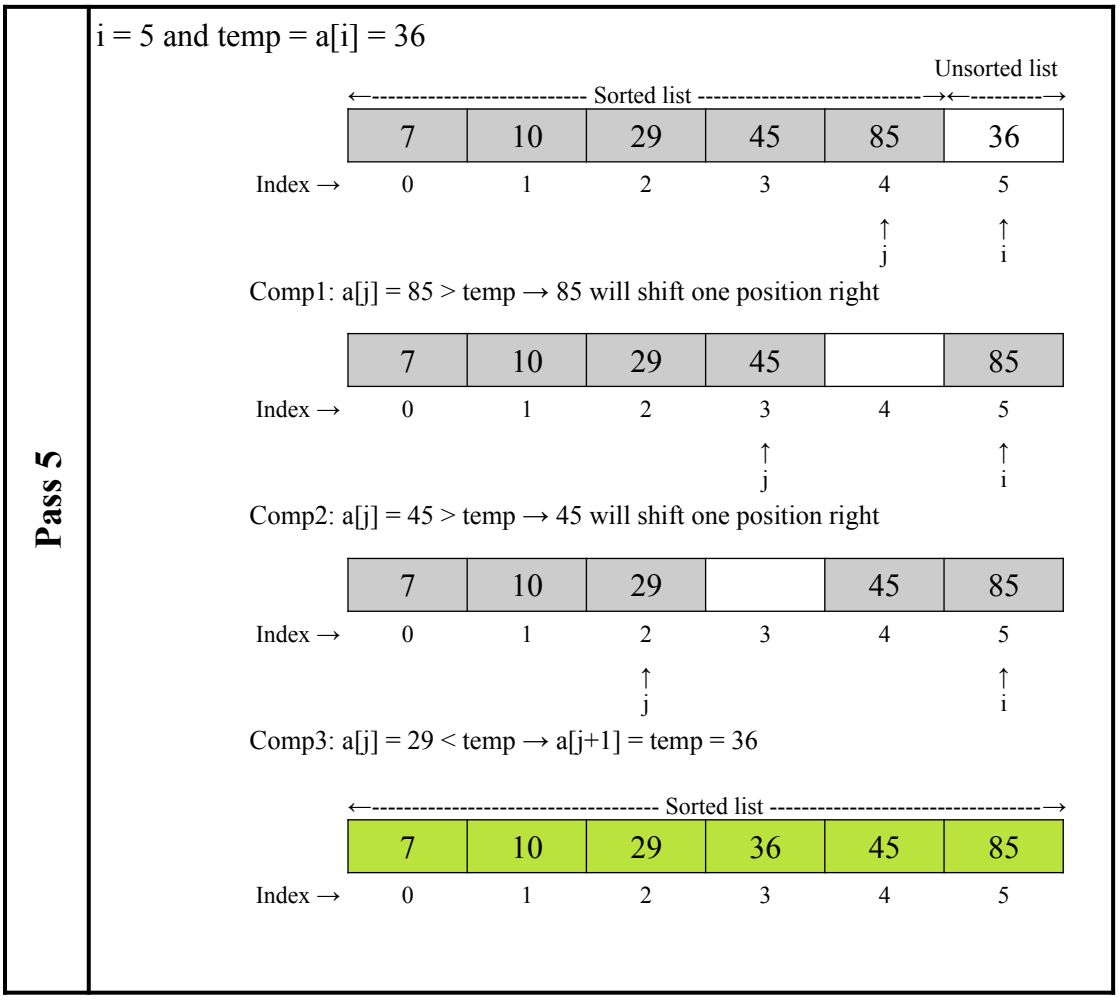


Fig.7.5: Step by step execution of insertion sort

Some observations regarding the insertion sort are given below.

- In insertion sort the entire list is divided into two lists, one sorted list and another unsorted list where sorted list starts with one element and the unsorted list starts with (n – 1) no. of elements. After completion of each pass the length of sorted list is incremented by one and the length of the unsorted list is decremented by one. Finally at the end of the insertion sort the length of the sorted list becomes n which gives the entire sorted list.

- There will be always (n – 1) no. of passes in insertion sort for an unsorted list with n no. of elements.

- During the execution of every pass first element from the unsorted list is inserted into the sorted list to a proper place. That's why this sorting is called insertion sort.

- There will be maximum 1 no. of comparison during 1st pass, maximum 2 no. of comparisons during 2nd pass, maximum 3 no. of comparisons during 3rd pass and so on. Therefore (n – 1) no. of comparisons will take place during the (n – 1)th pass in the insertion sort.

To implement insertion sort we require two nested loops, the outer loop for counting the number of passes and the inner loop for performing comparisons between the elements of the sorted list and the first or left-most element of the unsorted list. The algorithm of insertion sort is given below.

Algorithm of insertion sort:

Step 1: Start

Step 2: Input the no. of elements in a variable ‘n’.

Step 3: Input the unsorted list of n no. of elements into an array ‘a’.

Step 4: Set i = 1

Step 5: Repeat Step 6 to Step 12 while i < n

Step 6: Set temp = a[i]

Step 7: Set j = i – 1

Step 8: Repeat Step 9 to Step 10 while j ≥ 0 and a[j] > temp

Step 9: Set a[j + 1] = a[j]

Step 10: Set j = j – 1
[End of Loop (Step 8)]

Step 11: Set a[j + 1] = temp

Step 12: Set i = i + 1
[End of Loop (Step 5)]

Step 13: Stop

C program to arrange a set of numbers in ascending order using insertion sort.

```
#include<stdio.h>
void InsertionSort(int [],int );

int main()
{
	int a[50], i = 0, length;

	printf("Enter the number of elements: ");
	scanf("%d", &length);

	printf("Enter the elements of the array:\n");
	for(i=0; i<length; i++)
	{
		printf("Enter Element%d: ", i+1);
		scanf("%d", &a[i]);
	}

	InsertionSort(a, length);

	printf("Sorted List Using Insertion Sort: ");
	for(i=0; i<length; i++)
		printf("%d ", a[i]);

	printf("\n");
	return 0;
}

void InsertionSort(int arr[],int len)
{
	int i, j, temp;

	for(i=1; i<len; i++)
	{
		temp = arr[i];
		j = i - 1;

		while(j>=0 && arr[j] > temp)
		{
			arr[j+1] = arr[j];
			j--;
		}
		arr[j+1] = temp;
	}
}
```

Time complexity of insertion sort – To calculate the time complexity of insertion sort we have to consider the number of times comparisons happened to complete the insertion sort in best case and worst case.

Best case: When the list of elements are already sorted in ascending order, the insertion sort performs minimum number of comparisons to complete the sorting, which causes minimum execution time. In this condition there will be one comparison in every pass. As a result there will be (n – 1) no. of comparisons for (n – 1) no. of passes. Therefore the time complexity of the insertion sort becomes O(n) for best case.

Worst case: Worst case will happen when the unsorted list is in descending order and the list will have to be sorted in ascending order using insertion sort. In this situation we get the following results.

For 1^{st} pass, no. of comparisons = 1

For 2^{nd} pass, no. of comparisons = 2

For 3^{rd} pass, no. of comparisons = 3

In this way no. of comparisons for $(n-1)^{th}$ pass = n – 1

$\therefore$ Total no. of comparisons = 1 + 2 + 3 + …….. + (n – 2) + (n – 1)

$$= \frac{(n-1)(n-1+1)}{2}$$

$$= \frac{n(n-1)}{2} = \frac{(n^2-n)}{2}$$

Therefore the time complexity of insertion sort becomes $O(n^2)$ for worst case.

Average case: The time complexity of insertion sort in average case is $O(n^2)$.

In-place sorting – Insertion sort does not require any extra space. That's why it is an in-place sorting.

Stability of insertion sort – Insertion sort is a stable sorting technique.

Advantages of insertion sort:
1. It is easy to understand and implement.
2. It requires constant memory space which makes the space complexity O(1). That's why it is an in-place sorting.
3. It is a stable sorting algorithm.

Disadvantages of insertion sort:

1. It is not efficient for large data set.
2. It is not so efficient like merge sort, quick sort etc., because the time complexity of insertion sort for worst case is larger than that of the sorting techniques.

4) Quick sort – Quick sort algorithm was first developed by C. A. R. Hoare. It is a sorting algorithm based on Divide and Conquer strategy that picks up an element as a pivot or key element, place that pivot element in its proper position in the list and partitions the list into two parts with respect to the pivot element in such a way that all the elements in the left partition will be less or equal to the pivot element and all the elements in the right partition will be greater than the pivot element. Therefore after the completion of 1st pass we have two partitions (one left and other right partition) on the both sides of the pivot element. During the 2nd pass one element from the left partition will be selected as pivot element and other element from the right partition will be chosen as pivot element. Again the left partition as well as the right partition will be divided into two partitions with respect to the selected pivot elements when they are placed at their proper positions in the left and the right partition respectively. This process will continue until all partitions hold only a single element. Hence it can be observed that the list are being partitioned consecutively after the completion of every pass. Due to this reason quick sort is also known as partition exchange sort. The process of determining the proper place of the pivot element is given below in details.

1. Although any element may be selected as pivot element in case of quick sort, but here the 1st element in the list is selected as the pivot element.
2. Two variables 'start' and 'end' are taken initially to point the starting index and the last index of the array 'a'.
3. The pivot element is compared consecutively with other elements of the array one by one from left to right. If a[start] ≤ pivot, then the variable 'start' is incremented by one to point the next element, otherwise start is kept unchanged. Similarly the pivot element is also compared with the other elements in the array from right to left. If a[end] > pivot, then the variable 'end' is decremented by one each time to point the previous element, otherwise end remains unchanged.
4. When a[start] > pivot and a[end] ≤ pivot, the start and the end index remain unchanged somewhere in the list. Now the element pointed by start and the element pointed by end will be interchanged. Again the same procedure is continued until end becomes smaller than start.
5. When end becomes less than start, the process of comparisons is stopped and finally the proper position of the pivot element pointed by the end index is determined.
6. Therefore the pivot element and the element pointed by the variable end are swapped to achieve the correct position of the pivot element.
7. Now the list is partitioned into two parts with respect to the pivot element and the above mentioned same procedure is carried out to find out the proper place of another pivot element.

The above mentioned procedure of quick sort is demonstrated with the help of the following example in detail.

Pass 1

Array a

pivot = a[0] = 10 Lower Bound lb = 0 Upper Bound ub = 6

	10	15	1	2	9	10	11
Index →	0	1	2	3	4	5	6
	↑ start						↑ end

a[start] = pivot → start = start + 1

	10	15	1	2	9	10	11
Index →	0	1	2	3	4	5	6
		↑ start					↑ end

a[start] > pivot → start will be unchanged

	10	15	1	2	9	10	11
Index →	0	1	2	3	4	5	6
		↑ start					↑ end

a[end] > pivot → end = end – 1

	10	15	1	2	9	10	11
Index →	0	1	2	3	4	5	6
		↑ start				↑ end	

a[end] = pivot → end will be unchanged

	10	10	1	2	9	15	11
Index →	0	1	2	3	4	5	6
		↑ start				↑ end	

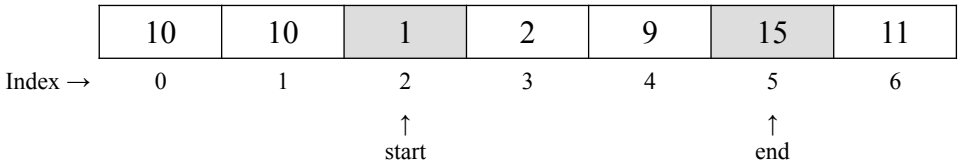
As start < end, a[start] and a[end] are swapped

	10	10	1	2	9	15	11
Index →	0	1	2	3	4	5	6
		↑ start				↑ end	

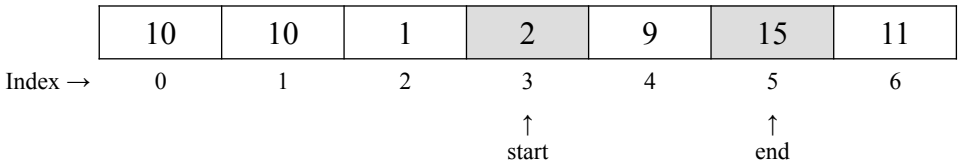
a[start] = pivot → start = start + 1

Pass 1

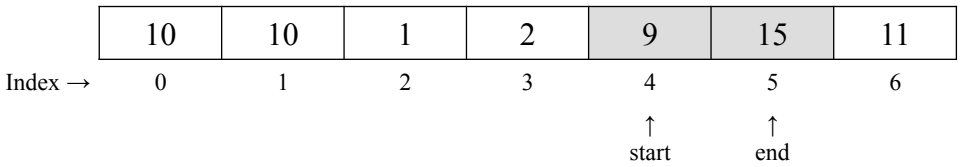
Array a



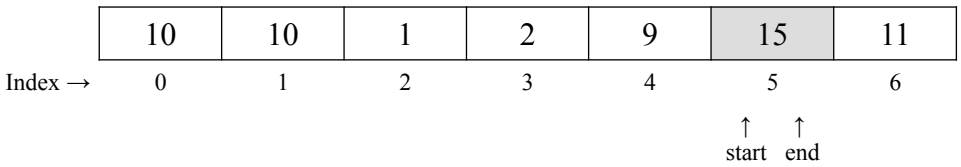
a[start] < pivot → start = start + 1



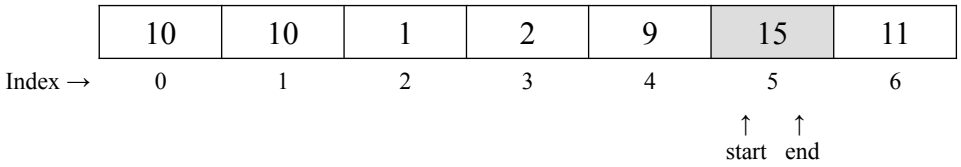
a[start] < pivot → start = start + 1



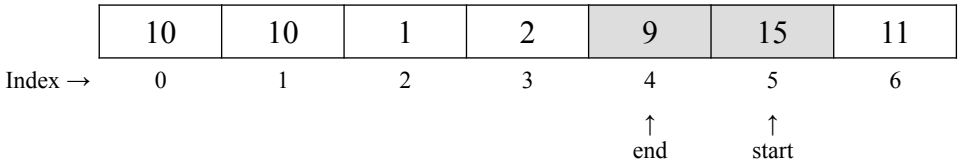
a[start] < pivot → start = start + 1



a[start] > pivot → start will be unchanged



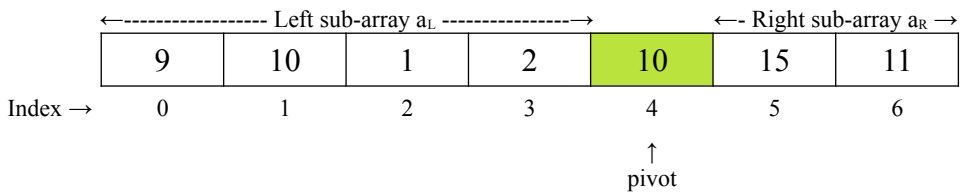
a[end] > pivot → end = end - 1



a[end] < pivot → end will be unchanged

As start ≮ end, a[start] and a[end] will not be swapped.
a[0] and a[end] will be swapped to place the pivot at its correct position.

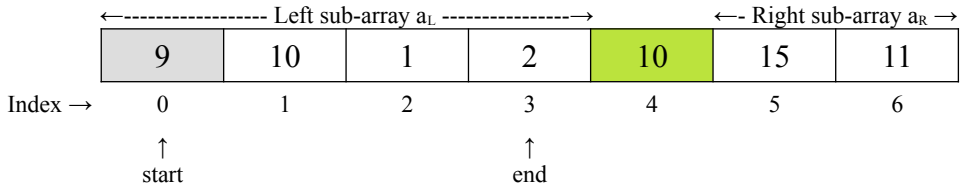
Pass 1 — **Array a**



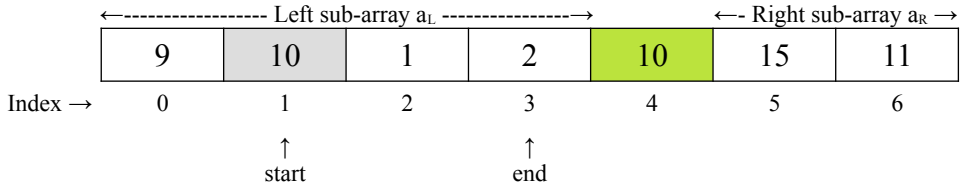
Pivot 10 has been placed at its correct index 4

Pass 2 — **Left sub-array a_L**

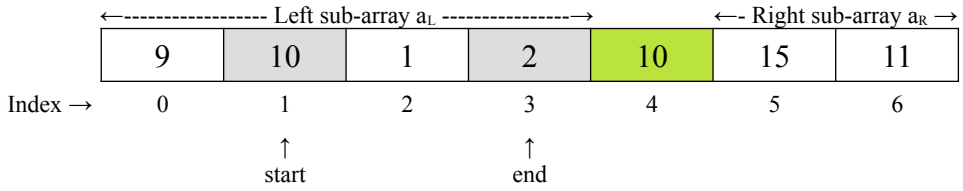
pivot = a[0] = 9 Lower Bound lb = 0 Upper Bound ub = 3



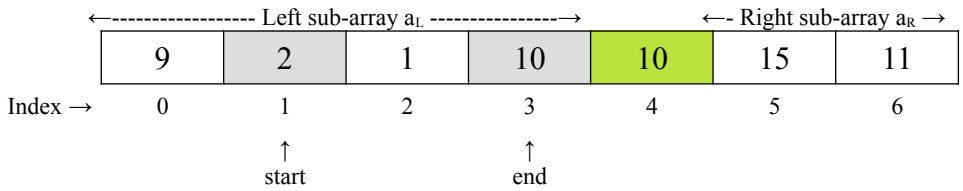
a[start] < pivot → start = start + 1



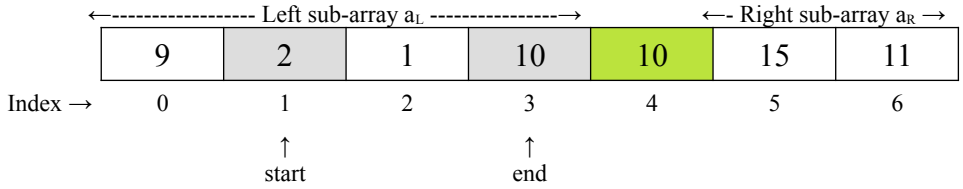
a[start] > pivot → start will be unchanged



a[end] < pivot → end will be unchanged



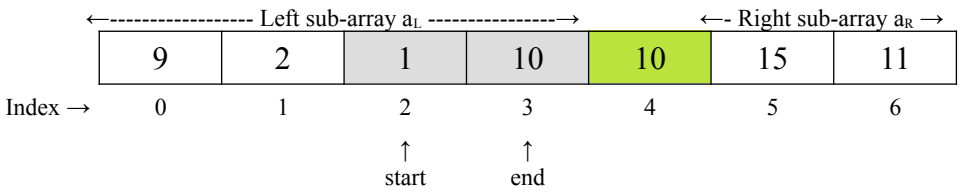
As start < end, a[start] and a[end] are swapped



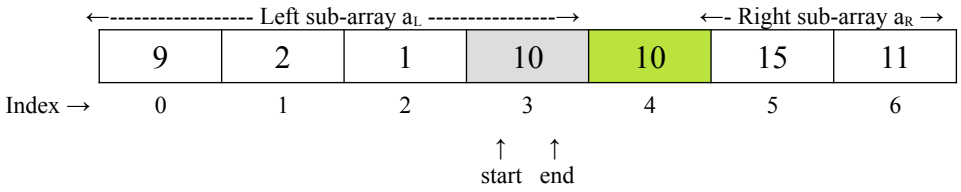
a[start] < pivot → start = start + 1

Pass 2

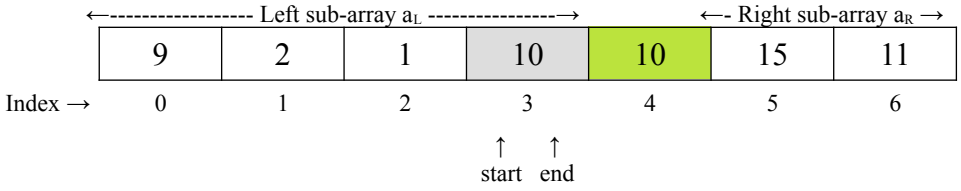
Left sub-array a_L



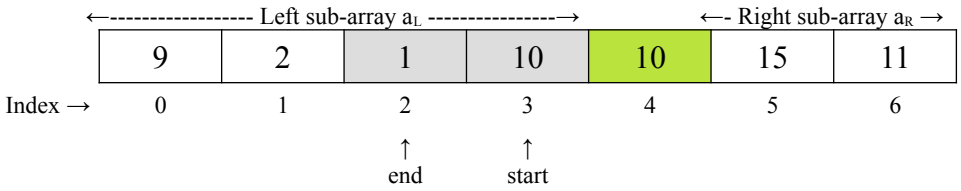
a[start] < pivot → start = start + 1



a[start] > pivot → start will be unchanged

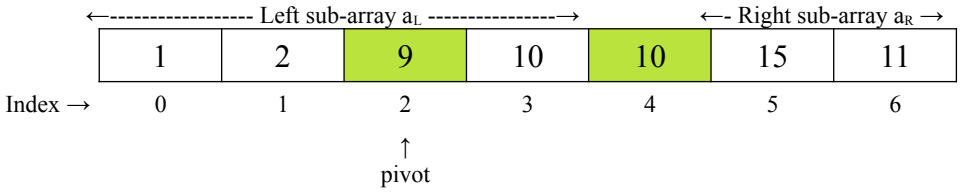


a[end] > pivot → end = end – 1



a[end] < pivot → end will be unchanged

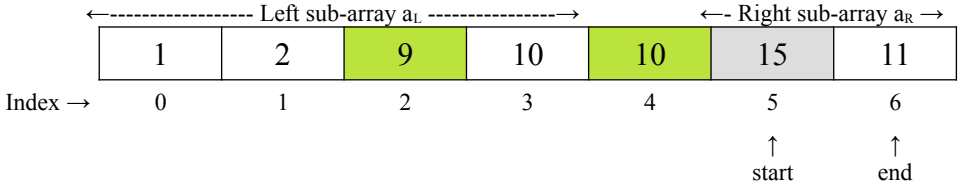
As start ≮ end, a[start] and a[end] will not be swapped.
a[0] and a[end] will be swapped to place the pivot at its correct position.



Pivot 9 has been placed at its correct index 2

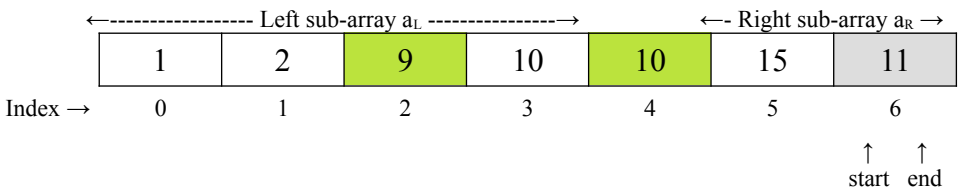
Right subarray a_R

pivot = a[5] = 15 Lower Bound lb = 5 Upper Bound ub = 6

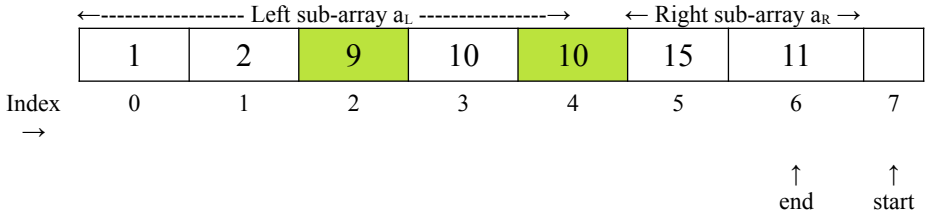


a[start] = pivot → start = start + 1

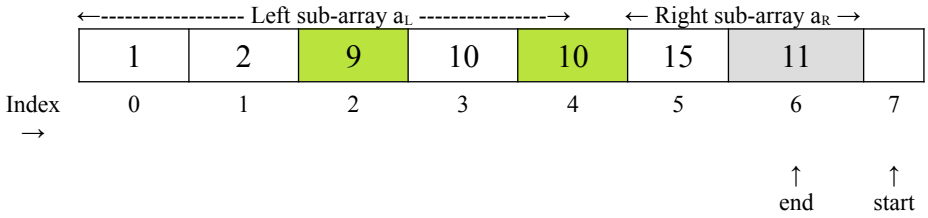
Pass 2 — **Right subarray a_R**



a[start] < pivot → start = start + 1

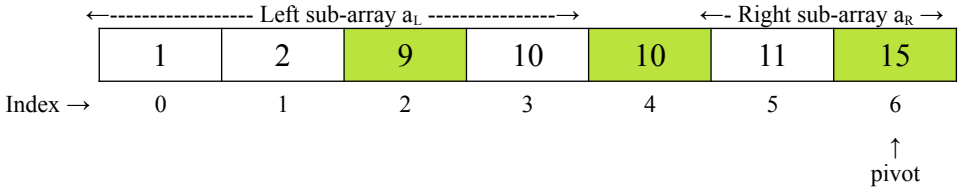


start > 6 → start = 7 (unchanged)



a[end] < pivot → end will be unchanged

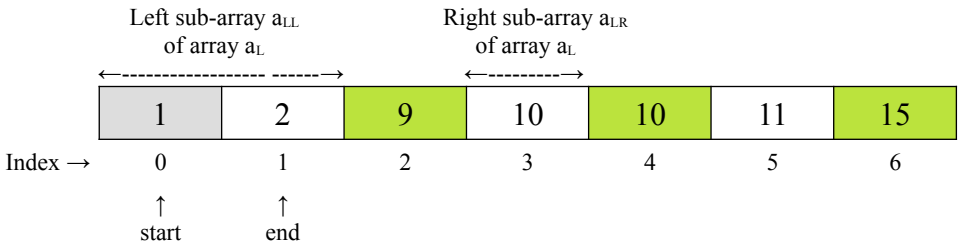
As start ≮ end, a[start] and a[end] will not be swapped.
a[5] and a[end] will be swapped to place the pivot at its correct position.



Pivot 15 has been placed at its correct index 6

Pass 3 — **Left sub-array a_{LL}**

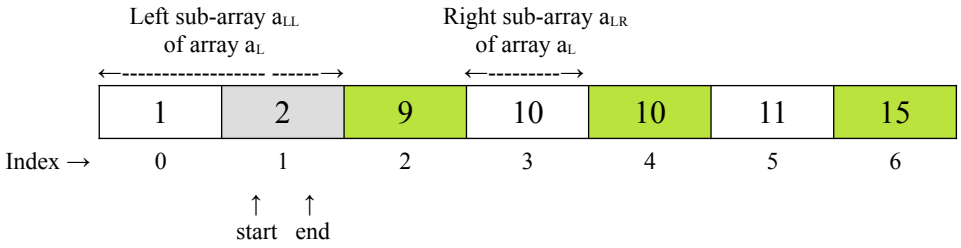
pivot = a[0] = 1 Lower Bound lb = 0 Upper Bound ub = 1



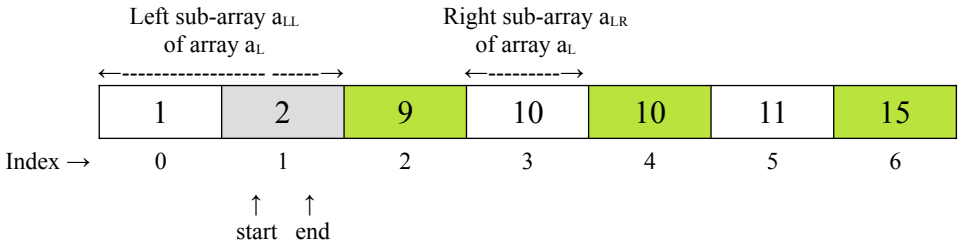
a[start] = pivot → start = start + 1

Pass 3

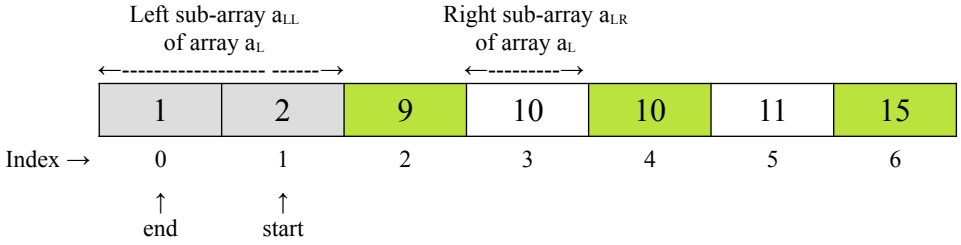
Left sub-array a_{LL}



a[start] > pivot → start will be unchanged

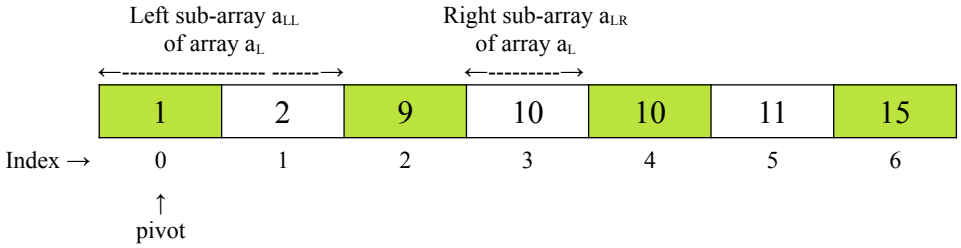


a[end] > pivot → end = end – 1



a[end] = pivot → end will be unchanged

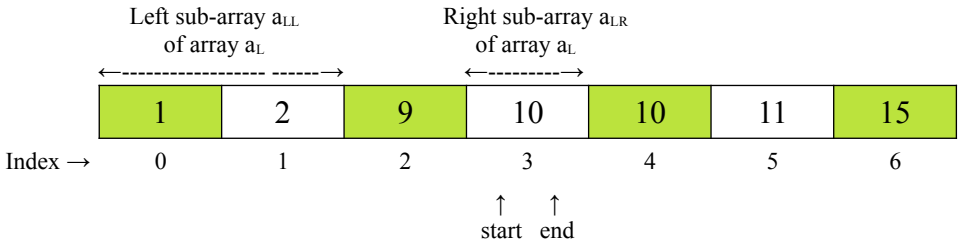
As start ≮ end, a[start] and a[end] will not be swapped.
a[0] and a[end] will be swapped to place the pivot at its correct position.



Pivot 1 has been placed at its correct index 0

Right sub-array a_{LR}

pivot = a[3] = 10 Lower Bound lb = 3 Upper Bound ub = 3



As lb ≮ ub, start and end will be unchanged

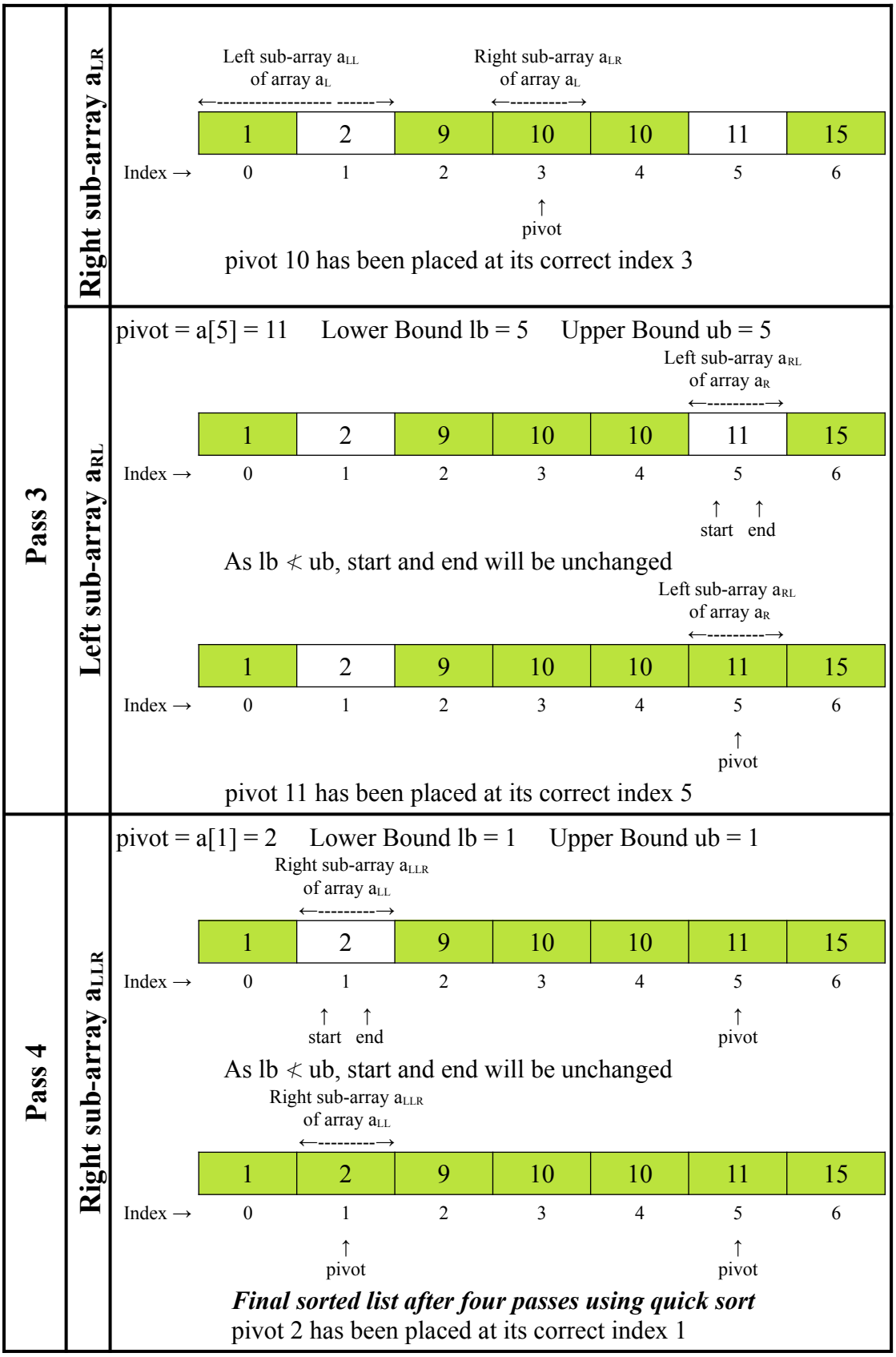


Fig.7.6: Step by step execution of quick sort

The above explained is shown again in brief only by giving the passes.

Initially:	10	15	1	2	9	10	11
Index →	0	1	2	3	4	5	6
After Pass 1:	9	10	1	2	10	15	11
Index →	0	1	2	3	4	5	6
After Pass 2:	1	2	9	10	10	11	15
Index →	0	1	2	3	4	5	6
After Pass 3:	1	2	9	10	10	11	15
Index →	0	1	2	3	4	5	6
After Pass 4:	1	2	9	10	10	11	15
Index →	0	1	2	3	4	5	6

Final sorted list after four passes using quick sort

Some observations regarding the quick sort are given below.

- In quick sort the first or last or any other element is selected as pivot element.
- After the completion of 1st pass the pivot element is placed at its correct position. The entire list is divided into two partitions with respect to the pivot element – one left partition and other right partition. During the 2nd pass the left and right partition are again sub-divided into four partitions after positioning the pivot elements into the proper places in the left and right partition. Hence it is evident that the pivot element is placed at its proper position after each pass.
- This partitioning is continued until all the partitions have single element. When this happens, the list becomes a sorted list.

Quick sort is a recursive process where a function is called to give the correct location of the pivot element after the completion of each pass. Now the quick sort function calls itself two times recursively – first function call for left partition and the second function call for right partition. The function call for left partition continues the same procedure of quick sort to determine the correct position of the pivot element from the left partition and divides it into another two partitions. Similarly the function call for the right partition does the same thing for right partition. The algorithm of quick sort is given below.

Algorithm of Quick Sort:
a → Array
lb → Lower bound
ub → Upper bound
loc → Correct location of the pivot element

QuickSort(a, lb, ub)
Step 1: Start

Step 2: If lb < ub, then
a) Set loc = Call Partition(a, lb, ub)
b) Call QuickSort(a, lb, loc – 1)
c) Call QuickSort(a, loc + 1, ub)
[End of If (Step 2)]

Step 3: Stop

Partition(a, lb, ub)
Step 1: Start

Step 2: Set start = lb

Step 3: Set end = ub

Step 4: Set pivot = a[lb]

Step 5: Repeat Step 6 to Step 11 while start ≤ end

Step 6: Repeat Step 7 while a[start] ≤ pivot and start <= ub

Step 7: Set start = start + 1
[End of Loop (Step 6)]

Step 8: Repeat Step 9 while a[end] > pivot and end ≥ lb

Step 9: Set end = end – 1
[End of Loop (Step 8)]

Step 10: If start < end, then

Step 11: Swap a[start] with a[end]
[End of If (Step 10)]
[End of Loop (Step 5)]

Step 12: Swap a[lb] with a[end]

Step 13: Return end

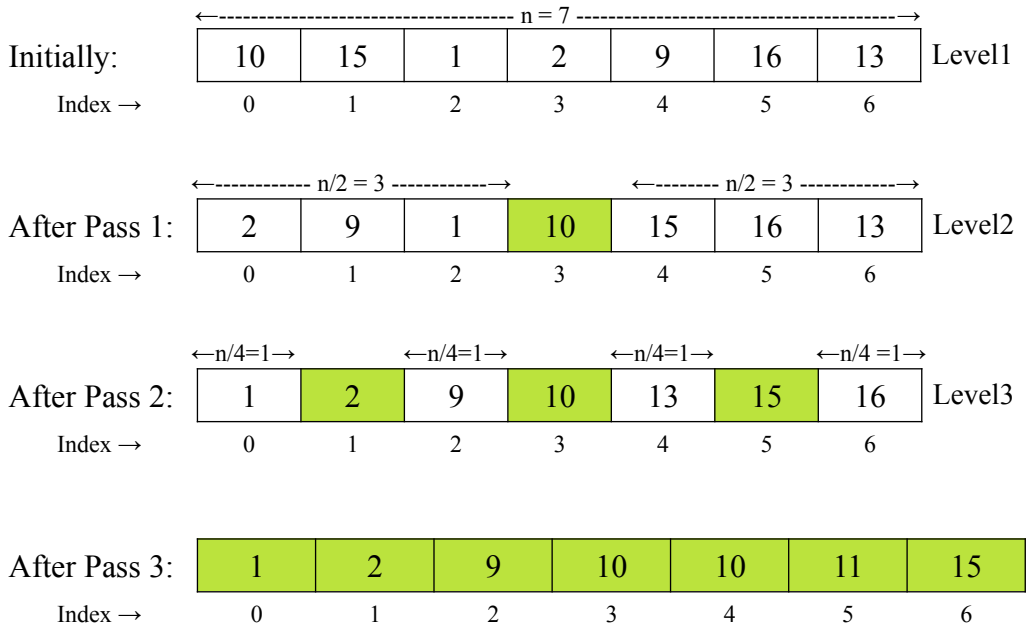
C program to arrange a set of numbers in ascending order using quick sort.

```
#include<stdio.h>

void QuickSort(int [],int ,int );
int Partition(int [],int ,int );
void Swap(int *,int *);

int main()
{
	int a[10], i = 0, length;

	printf("Enter the number of elements: ");
	scanf("%d", &length);

	printf("Enter the elements of the array:\n");
	for(i=0; i<length; i++)
	{
		printf("Enter Element%d: ", i+1);
		scanf("%d", &a[i]);
	}

	printf("\nUnsorted List: ");

	for(i=0; i<length; i++)
		printf("%d  ", a[i]);

	QuickSort(a, 0, length-1);

	printf("\n\nSorted List Using Quick Sort: ");

	for(i=0; i<length; i++)
		printf("%d  ", a[i]);

	printf("\n");
	return 0;
}

void QuickSort(int a[],int lb,int ub)
{
	int loc;
	if(lb < ub)
	{
		loc = Partition(a, lb, ub);
		QuickSort(a, lb, loc-1);
		QuickSort(a, loc+1, ub);
	}
}
```

```
int Partition(int a[], int lb,int ub)
{
        int start = lb, end = ub;
        int pivot = a[start];

        while(start <= end)
        {
                while(a[start] <= pivot && start <= ub)
                        start++;

                while(a[end] > pivot && end >= lb)
                        end--;

                if(start < end)
                        Swap(&a[start], &a[end]);
        }

        Swap(&a[lb], &a[end]);
        return end;
}

void Swap(int *p,int *q)
{
        int temp;

        temp = *p;
        *p = *q;
        *q = temp;
}
```

Time complexity of quick sort – To calculate the time complexity of quick sort we have to determine the number of times comparisons happened to complete the quick sort in best case and worst case.

Best Case: Best situation for quick sort happens when it take minimum time to execute i.e. when it accomplishes minimum number of comparisons. This will happen when the entire list/ partitions are sub-divided equally into two parts. It means that every time the pivot element will be placed at the middle of the list after completion of every pass and it will create two partitions with same number of elements with respect to the pivot element. The following example will clarify this situation with a list of 7 elements 10, 15, 1, 2, 9, 16, 13.



From the above example it is clear that at 2nd level the number of elements will be almost n/2 in every partition where n is the number of elements in the original array. Similarly there will be n/4 and n/8 elements approximately in each partition at 3rd and 4th level respectively. In this way the number of elements will be divided by 2 for successive levels until the number of elements reaches to one element.

Suppose it requires k number of levels to get one element in each partition using quick sort for a list of n number of elements.

At Level 1: No. of elements = n = $n / 2^0$

At Level 2: No. of elements = n/2 = $n / 2^1$

At Level 3: No. of elements = n/2 = $n / 2^2$

.

.

.

At Level k: No. of elements = $n / 2^{k-1} = 1$

or, $2^{k-1} = n$

$\therefore k = \log_2 n + 1$

Now it takes almost n comparisons for every level in quick sort. Therefore for k number of levels the quick sort requires n ($\log_2 n + 1$) comparisons. It constitutes the time complexity of quick sort to be O(n $\log_2 n$).

Recurrence Method: The best case time complexity of quick sort may be determined using recurrence relation also. To establish the recurrence relation of the quick sort we have to recall the function of quick sort which is given below for understanding.

```
void QuickSort(int a[],int lb,int ub) ---------  Time take by the QuickSort function for n
                                                  number of elements: T(n)
{
        int loc;
        if(lb < ub)
        {
                loc = Partition(a, lb, ub); -----Time taken by the Partition function to
                                                  complete n comparisons to place the
                                                  pivot element at proper position: n units
                QuickSort(a, lb, loc-1); ------ Time taken by the function for left
                                                  partition of n/2 elements: T(n/2)
                QuickSort(a, loc+1, ub); ----  Time taken by the function for right
                                                  partition of n/2 elements: T(n/2)
        }
}
```

If the time taken by the quick sort for n elements is T(n), the recurrence relation is given by

$$T(n) = n + T(n/2) + T(n/2)$$

or, $T(n) = 2\,T(n/2) + n$

or, $T(n) = 2\,[2\,T(n/4) + n/2] + n$ $\quad[\because T(n/2) = 2\,T(n/4) + n/2]$

or, $T(n) = 2^2\,T(n / 2^2) + 2n$

or, $T(n) = 2^2\,[2\,T(n/8) + n/4] + 2n$ $\quad[\because T(n/4) = 2\,T(n/8) + n/4]$

or, $T(n) = 2^3\,T(n / 2^3) + 3n$

or, $T(n) = 2^k\,T(n / 2^k) + kn$ ---------- (1)

Where k is the number of levels to get one element in each partition using quick sort

$\therefore n / 2^k = 1$

or, $2^k = n$

$\therefore k = \log_2 n$

Now putting $k = \log_2 n$ into equation (1) we get,

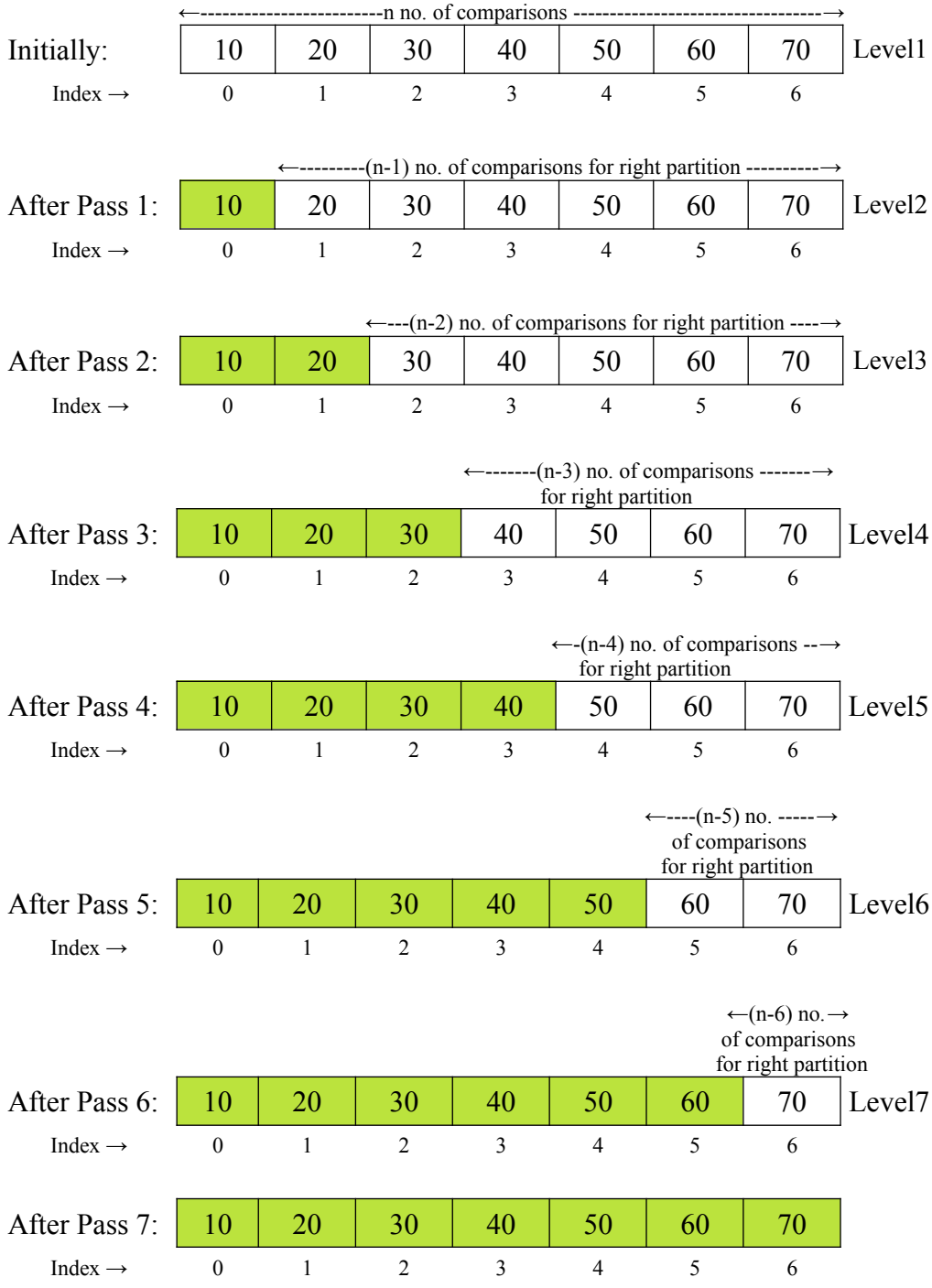
$$T(n) = n\,T(1) + n \log_2 n$$

For one element in the list there will be 1 comparison for quick sort which takes 1 unit of time i.e. T(1) = 1.

$$\therefore T(n) = n \cdot 1 + n \log_2 n = n (\log_2 n + 1)$$

Therefore the time complexity of quick sort in best case is $O(n \log_2 n)$.

Worst Case: Worst case for quick sort happens when the entire list of elements is already sorted i.e. the elements are either in ascending order or descending order. Let's take an example where all the elements are in ascending order.



From the above example we can observe that after completion of every pass only one partition (right partition) is generated because each time the selected pivot elements are already placed at their correct position. As there is no left partition no comparisons are done from the left partition and all the comparisons result from only right partitions. As a result we can write,

Total no. of comparisons = n + (n – 1) + (n – 2) + (n – 3) + …….. + 2 + 1

= 1 + 2 + 3 + ……. upto n no. of terms

$$= \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

Therefore the time complexity of quick sort in worst case is $O(n^2)$.

Recurrence Method: Like the recurrence method in the best case the recurrence relation is to be formulated here also. In this case we have already seen that no left partition is created with respect to the pivot element. Due to this reason the first function call for left partition is not executed, only the second function call for right partition is executed.

```
void QuickSort(int a[],int lb,int ub) ---------  Time take by the QuickSort function for n
                                                  number of elements: T(n)
{
        int loc;
        if(lb < ub)
        {
                loc = Partition(a, lb, ub); -----Time taken by the Partition function to
                                                  complete n comparisons to place the
                                                  pivot element at proper position: n units
                QuickSort(a, lb, loc-1); ------ Time taken by the function for left
                                                  partition: 0 since left partition is not
                                                  created
                QuickSort(a, loc+1, ub); ----  Time taken by the function for right
                                                  partition of (n - 1) elements: T(n – 1)
        }
}
```

If the time taken by the quick sort for n elements is T(n), the recurrence relation is given by

$$T(n) = n + 0 + T(n-1)$$

or, $T(n) = T(n-1) + n$

or, $T(n) = T(n-2) + (n-1) + n$ [$\because T(n-1) = T(n-2) + (n-1)$]

or, $T(n) = T(n-3) + (n-2) + (n-1) + n$ [$\because T(n-2) = T(n-3) + (n-2)$]

$$\text{or, } T(n) = T(n-k) + \{n-(k-1)\} + \{n-(k-2)\} + \ldots + (n-2) + (n-1) + n$$

Now putting $k = n - 1$ we get,

$$T(n) = T(n-n+1) + \{n-(n-1-1)\} + \{n-(n-1-2)\} + .. + (n-1) + n$$

$$\text{or } T(n) = T(1) + 2 + 3 + \ldots\ldots\ldots + (n-1) + n$$

For one element in the list there will be 1 comparison for quick sort which makes. $T(1) = 1$.

$$\therefore T(n) = 1 + 2 + 3 + \ldots\ldots\ldots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

Therefore the time complexity of quick sort in worst case is $O(n^2)$.

Average case: The time complexity of quick sort in average case is $O(n \log_2 n)$.

In-place sorting – Quick sort does not take any extra space. That's why it is an in-place sorting.

Stability of quick sort – Quick sort is an unstable sorting technique because it may interchange the positions of duplicate elements after sorting. That means, the element which comes before the duplicate element in original list may be placed after the duplicate element in the sorted list. This situation may be described with the help of the following example.

Initially:	**10_A**	15	1	2	9	**10_B**	11
Index →	0	1	2	3	4	5	6
After Pass 1:	9	**10_B**	1	2	**10_A**	15	11
Index →	0	1	2	3	4	5	6
After Pass 2:	1	2	9	**10_B**	**10_A**	11	15
Index →	0	1	2	3	4	5	6
After Pass 3:	1	2	9	**10_B**	**10_A**	11	15
Index →	0	1	2	3	4	5	6
After Pass 4:	1	2	9	**10_B**	**10_A**	11	15
Index →	0	1	2	3	4	5	6

Final sorted list after four passes using quick sort

It can be clearly observed in the above example that 10_A which is placed before 10_B in the original unsorted list is positioned after 10_B in the sorted list. This consequence shows that quick sort is unstable.

Advantages of quick sort:
1. Quick sort is faster than other algorithms like bubble sort, selection sort and insertion sort.
2. It is efficient for large data sets, but can be used for small or medium data set also.
3. It requires constant memory space which makes the space complexity O(1). That's why it is an in-place sorting.

Disadvantages of quick sort:
1. The process of quick sort is complex and massively recursive in nature.
2. It has the worst time complexity of O(n^2) when the list is already sorted.
3. It is not a stable sort.

5) Merge sort – Merge sort is a sorting algorithm which uses divide, conquer and combine strategy. It is first invented by John Von Neumann in the year 1945.

Divide: In this stage the aim is to divide the entire list into some sub-lists in such a way that every sub-list becomes sorted. Any sub-list can be said to be sorted if and only if the sub-list will hold one element. Due to this reason merge sort divides the list successively into sub-lists of one element. At first merge sort follows the divide strategy where the entire list of n elements is divided into two sub-lists of n/2 elements. Again each sub-list of n/2 elements will be divided into another two sub-lists of n/4 elements. This process will continue until all the sub-lists hold one element. To divide the array into two sub-array the minimum index and the maximum index of the array is pointed by lb (lower bound) and ub (upper bound) respectively and the middle index is determined by averaging lb and ub i.e. (lb + ub) / 2. Now the left partition is constructed using the sub-array starting from lb to mid and the right partition is formed by another sub-array starting from (mid + 1) to ub.

Conquer: Each sub-array is sorted individually using the merge sort algorithm.

Combine/ Merge: At this last stage the two sorted sub-lists are combined/ merged in such a way that the resultant merged list becomes sorted once again. As this technique requires two sorted sub-arrays, this process must be started using two sub-lists of one element. Because we know any list with single element can be considered as sorted always. After the completion of this merge process we shall get the entire list of n number of elements in sorted form. To achieve this sorted array another array of size n will be utilized. The use of an extra array makes the merge sort to be non-in-place sorting. Two elements – one from left sub-array and other from right sub-array are compared to each other. The element which is smaller will be placed into the extra array. Thus the extra array is filled with the elements from left partition and right partition in a sorted manner. Finally the sorted elements in the extra array are restored into the original array.

The step by step procedure of quick sort is shown below in Fig.7.7.

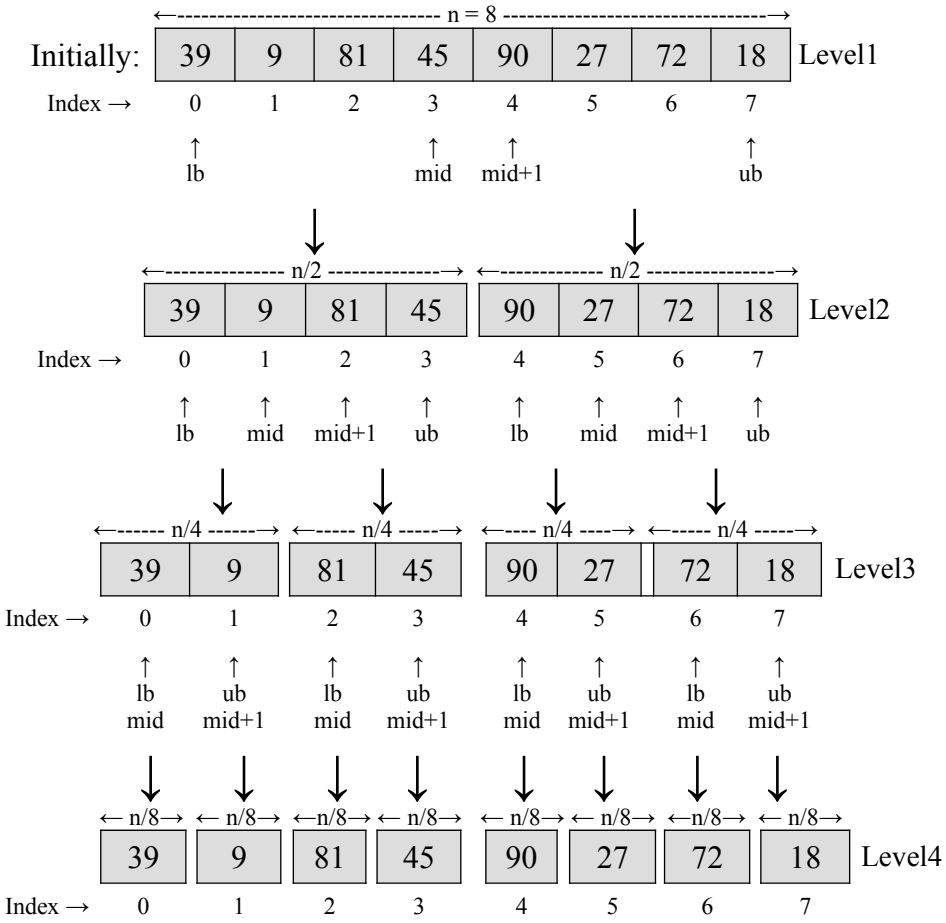


Fig.7.7(a): Divide and conquer of the array using merge sort

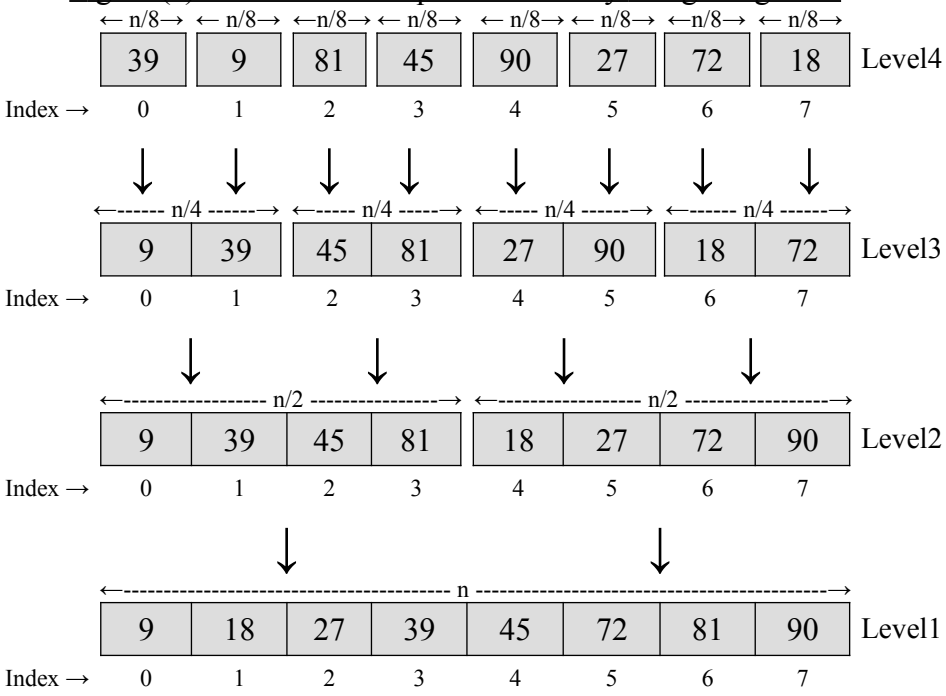


Fig.7.7(b): Merging of elements using merge sort to form sorted array

The summarized levels of merging the elements are shown in Fig.7.7(b), but the detail steps to merge two sorted sub-arrays are not shown for the ease of understanding. Now the detail sequences of merging two sub-arrays – (9, 39, 45, 81) and (18, 27, 72, 90) at Level 2 shown in Fig.7.7(b) are given in the following Fig.7.8.

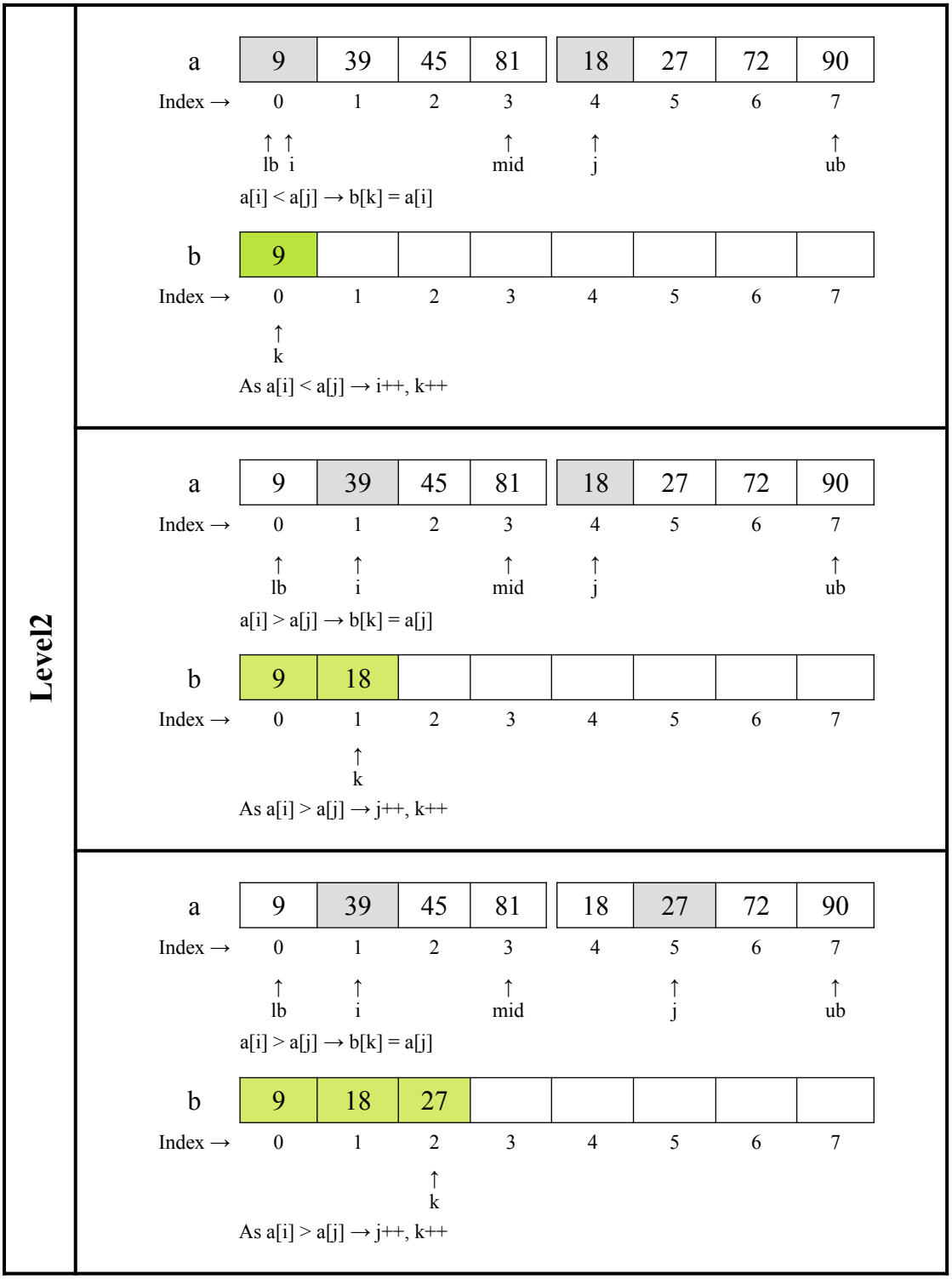


Fig.7.8: Step by step merging process

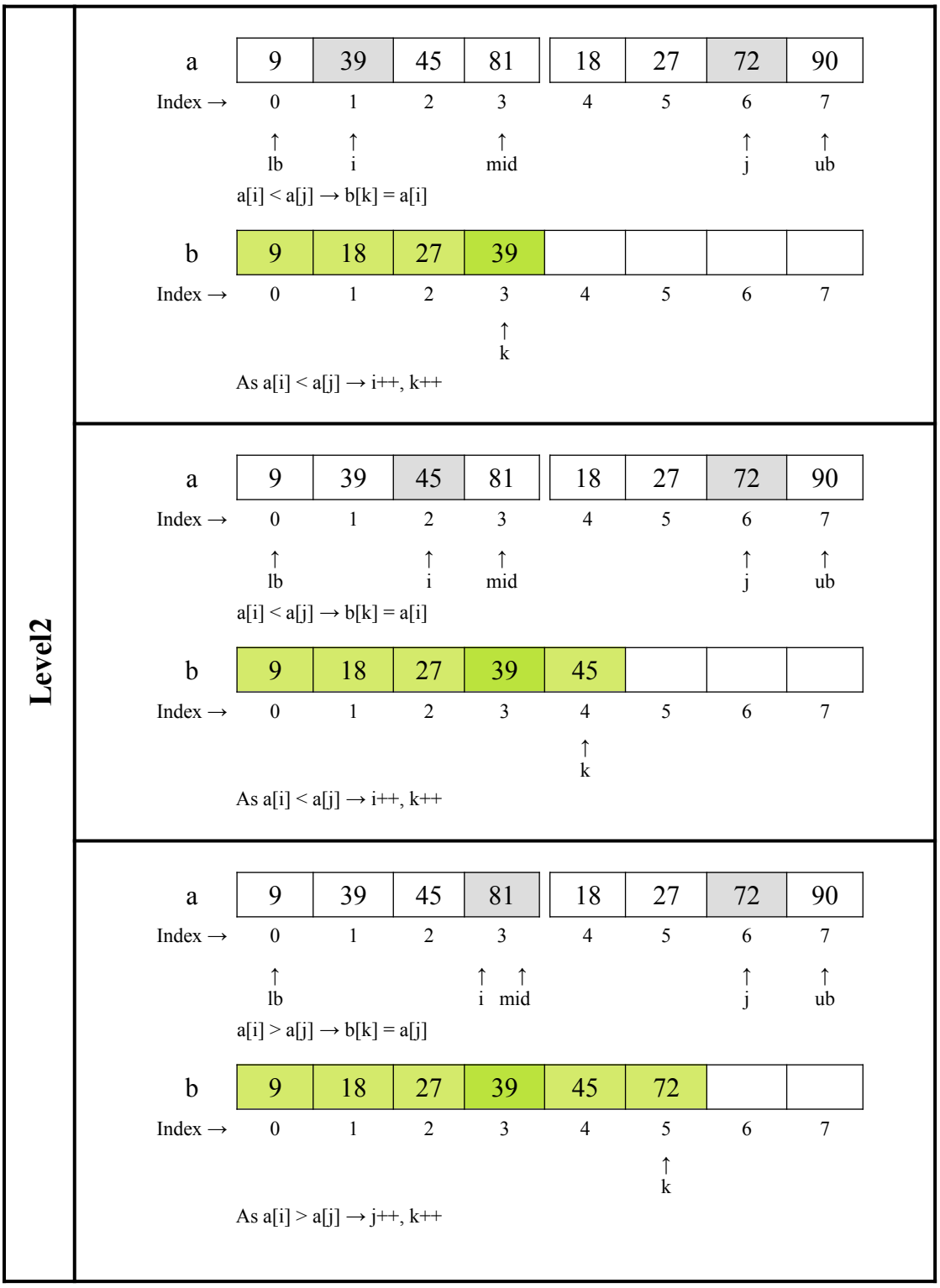


Fig.7.8: Step by step merging process

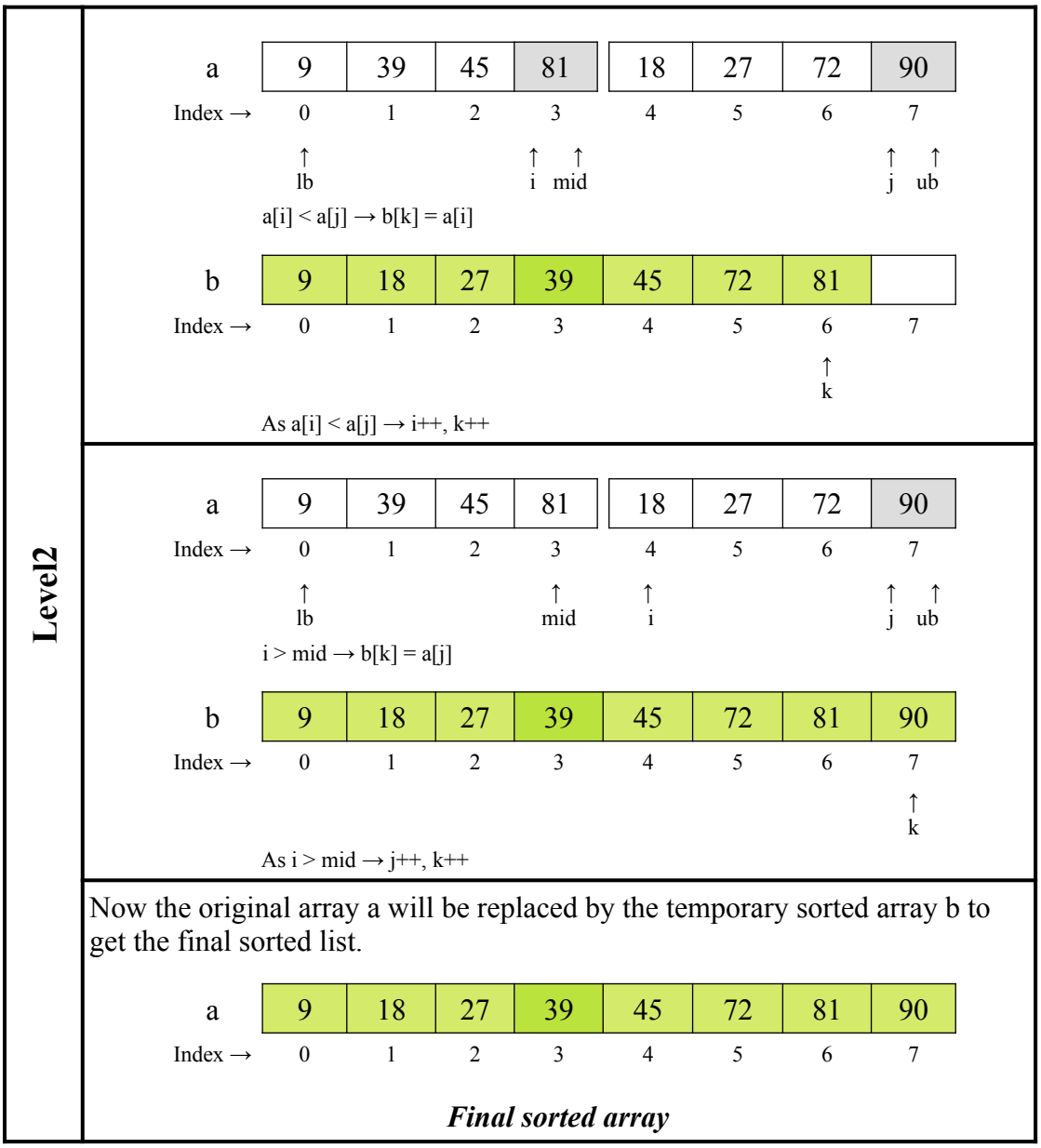


Fig.7.8: Step by step merging process

Some observations for the merge sort are given below.

- In merge sort the entire list will be sub-divided continuously into two halves i.e. two sub-lists with half of the elements of its parent list. This implies that the entire array of n elements will be sub-divided into two sub-arrays of n/2 elements, the sub-arrays of n/2 elements will be divided into sub-arrays of n/4 elements and so on. This division procedure will continue until all the sub-arrays hold one element.

- In merge sort all the sub-arrays should be sorted. To fulfill this condition all the sub-arrays will hold one element finally, because a sub-array with single element can be surely said sorted.

- The merging process will start using the sub-arrays with one elements and after merging at every stage the new resultant array becomes sorted once again. This process will be continued until the entire array becomes sorted.

Merge sort is a recursive process where the function MergeSort calls itself two times – one for the left half of the array and other for the right half of the array. Due these two recursive calls the array of n elements will be divided into one element finally. After these two recursive calls another function Merge is used to merge the sub-arrays to result a sorted array of n elements. The algorithm of merge sort is given below.

Algorithm of Merge Sort:
a → Array
lb → Lower bound
ub → Upper bound

MergeSort(a, lb, ub)
Step 1: Start

Step 2: If lb < ub, then
a) Set mid = (lb + ub) / 2;
b) Call MergeSort(a, lb, mid)
c) Call MergeSort(a, mid + 1, ub)
d) Call Merge(a, lb, mid, ub)
[End of If (Step 2)]

Step 3: Stop

Merge(a, lb, mid, ub)
Step 1: Start

Step 2: Set i = lb

Step 3: Set j = mid + 1

Step 4: Set k = lb

Step 5: Repeat Step 6 to Step 7 while i ≤ mid and j ≤ ub

Step 6: If a[i] ≤ a[j], then
a) Set b[k] = a[i]
b) Set i = i + 1
Otherwise
c) Set b[k] = a[j]
d) Set j = j + 1
[End of If-Else (Step 6)]

Step 7: Set k = k + 1
[End of Loop (Step 5)]

Step 8: If i > mid, then
a) Repeat b) to d) while j ≤ ub
b) Set b[k] = a[j]
c) Set j = j + 1
d) Set k = k + 1
[End of Loop (a)]
Otherwise
e) Repeat f) to h) while i ≤ mid
f) Set b[k] = a[i]
g) Set i = i + 1
h) Set k = k + 1
[End of Loop (e)]
[End of If-Else (Step 8)]

Step 9: Set i = lb

Step 10: Repeat Step 11 to Step 12 while i < k

Step 11: Set a[i] = b[i]

Step 12: Set i = i + 1
[End of Loop (Step 10)]

Step 13: Return

C program to arrange a set of numbers in ascending order using merge sort.

```
#include<stdio.h>

void MergeSort(int [],int ,int );
void Merge(int [],int ,int ,int );
int main()
{
        int a[50], i, length;
        printf("Enter the number of elements: ");
        scanf("%d", &length);
        printf("Enter the elements of the array:\n");
        for(i=0; i<length; i++)
        {
                printf("Enter Element%d: ", i+1);
                scanf("%d", &a[i]);
        }

        MergeSort(a, 0, length-1);
```

```
        printf("Sorted List Using Merge Sort: ");

        for(i=0; i<length; i++)
                printf("%d ", a[i]);

        printf("\n");
        return 0;
}

void MergeSort(int a[],int lb,int ub)
{
        int mid;

        if(lb < ub)
        {
                mid = (lb + ub)/2;
                MergeSort(a, lb, mid);
                MergeSort(a, mid+1, ub);
                Merge(a, lb, mid, ub);
        }
}

void Merge(int a[],int lb,int mid,int ub)
{
        int i, j, k;
        int b[50];

        i = lb;
        j = mid + 1;
        k = lb;

        while(i<=mid && j<=ub)
        {
                if(a[i] <= a[j])
                {
                        b[k] = a[i];
                        i++;
                }
                else
                {
                        b[k] = a[j];
                        j++;
                }
                k++;
        }
```

```
        if(i > mid)
        {
                while(j <= ub)
                {
                        b[k] = a[j];
                        j++;
                        k++;
                }
        }
        else
        {
                while(i <= mid)
                {
                        b[k] = a[i];
                        i++;
                        k++;
                }
        }

        for(i=lb; i<k; i++)
                a[i] = b[i];
}
```

Time complexity of merge sort – To calculate the time complexity of merge sort we have to determine the number of times operation happened to complete the merge sort in best case and worst case. In case of merge sort the time complexity remains same for best, worst and average case.

In Fig.7.7 we can observe that it divides the entire array of n elements into sub-arrays of n/2 elements (approximately) at Level2. Similarly we have n/4 elements for each sub-array at Level3, n/8 elements for every sub-array at Level4 and so on. Suppose at k^{th} Level each sub-array will hold one element and this k^{th} level is the last level in the merge sort.

At Level 1 number of elements in the array = $n = n / 2^0$

At Level 2 number of elements in each sub-array = $n / 2 = n / 2^1$

At Level 3 number of elements in each sub-array = $n / 4 = n / 2^2$

At Level 4 number of elements in each sub-array = $n / 8 = n / 2^3$

$\therefore$ At k^{th} Level number of elements in each sub-array = $n / 2^{k-1} = 1$

or, $2^{k-1} = n$

$\therefore k = \log_2 n + 1$

Now for every level we can see that there will be some comparisons and transferring of elements from the original array a to the auxiliary array b to store the elements in sorted manner. Finally the resultant sorted elements stored in the auxiliary array b are restored into original array a during the merge operation. Obviously it takes a time proportional to the number of elements in the entire list (n). As a result the time taken by each level becomes C n where C is a constant.

Therefore the total time taken due to k number of levels = k C n = C n ($\log_2$ n + 1). The time complexity of merge sort becomes O(n $\log_2$ n).

Recurrence Method: The time complexity of merge sort may be determined using recurrence relation also. To establish the recurrence relation of the merge sort we have to keep in mind the function of merge sort which is given below for clarification.

```
void MergeSort(int a[],int lb,int ub) ---------  Time take by the QuickSort function for n
                                                 number of elements: T(n)
{
        int mid;
        if(lb < ub)
        {
                mid = (lb + ub) / 2;
                MergeSort(a, lb, mid); ------  Time taken by the function for left
                                               sub-array of n/2 elements: T(n/2)
                MergeSort(a, mid+1, ub); ---  Time taken by the function for right
                                               sub-array of n/2 elements: T(n/2)
                Merge(a, lb, mid, ub); ------  Time taken by the Merge function: C n
                                               Where C is a constant and n is the
        }                                      number of elements in the entire array
}
```

If the time taken by the quick sort for n elements is T(n), the recurrence relation is given by

$$T(n) = T(n/2) + T(n/2) + C\,n$$

or, $T(n) = 2\,T(n/2) + C\,n$

or, $T(n) = 2\,[2\,T(n/4) + C\,n/2] + C\,n$ [$\because T(n/2) = 2\,T(n/4) + C\,n/2$]

or, $T(n) = 2^2\,T(n / 2^2) + 2C\,n$

or, $T(n) = 2^2\,[2\,T(n/8) + C\,n/4] + 2n$ [$\because T(n/4) = 2\,T(n/8) + C\,n/4$]

or, $T(n) = 2^3\,T(n / 2^3) + 3C\,n$

or, $T(n) = 2^k\,T(n / 2^k) + kC\,n$ ---------- (1)

Where k is the number of levels to get one element in each partition using merge sort

$\therefore n / 2^k = 1$

or, $2^k = n$

$\therefore k = \log_2 n$

Now putting $k = \log_2 n$ into equation (1) we get,

$$T(n) = n\,T(1) + C\,n \log_2 n$$

For one element in the list it takes 1 unit of time i.e. $T(1) = 1$.

$$\therefore T(n) = n \cdot 1 + C\,n \log_2 n = n\,(C \log_2 n + 1)$$

The time complexity of merge sort in best / worst / average case is $O(n \log_2 n)$.

Non-in-place sorting – Merge sort takes an extra auxiliary array to combine two sorted sub-arrays, which consumes extra memory space proportional to the number of elements. That's why it is a non-in-place sorting.

Stability of merge sort – Merge sort does not interchange the positions of duplicates elements in the original list. This makes the merge sort to be a stable sort.

Advantages of merge sort:

1. Merge has the time complexity in the order of $O(n \log_2 n)$ for best or average or worst case which makes it superior than other sorting algorithms like bubble sort, insertion sort, selection sort etc.
2. It is efficient for large data sets, but can be used for small or medium data set also.
3. It is a stable sort.

Disadvantages of merge sort:

1. The process of merge sort is quite complex and massively recursive in nature.
2. It takes extra space to sort the elements which makes it a non-in-place sorting algorithm.