

# UNIT-I

## Introduction to Algorithm and Data Structures

**Definition:** - An algorithm is a Step By Step process to solve a problem, where each step indicates an intermediate task. Algorithm contains finite number of steps that leads to the solution of the problem.

Algorithm has the following basic properties

- **Input-Output:**- Algorithm takes '0' or more input and produces the required output. This is the basic characteristic of an algorithm.
  - **Finiteness:**- An algorithm must terminate in countable number of steps.
  - **Definiteness:** Each step of an algorithm must be stated clearly and unambiguously.
  - **Effectiveness:** Each and every step in an algorithm can be converted in to programming language statement.
  - **Generality:** Algorithm is generalized one. It works on all set of inputs and provides the required output. In other words it is not restricted to a single input value. Categories of Algorithm: Based on the different types of steps in an Algorithm, it can be divided into three categories, namely
    - Sequence
    - Selection and
    - Iteration
- Sequence:** The steps described in an algorithm are performed successively one by one without skipping any step. The sequence of steps defined in an algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm.

**Example:**

// adding two numbers

Step 1: start

Step 2: read a,b

Step 3: Sum=a+b

Step 4: write Sum

Step 5: stop

Selection: The sequence type of algorithms are not sufficient to solve the problems, which involves decision and conditions. In order to solve the problem which involve decision making or option selection, we go for Selection type of algorithm. The general format of Selection type of statement is as shown below:

```
if(condition)
    Statement-1;
else
    Statement-2;
```

The above syntax specifies that if the condition is true, statement-1 will be executed otherwise statement-2 will be executed. In case the operation is unsuccessful. Then sequence of algorithm should be changed/ corrected in such a way that the system will re execute until the operation is successful.

Example :

//biggest among two numbers

Step 1 : Start

Step 2: read a,b

step 3: if a>b then

step 4: write "a is greater than b"

step 5: else

step 6: write "b is greater than a"

step 7: stop

### **Performance Analysis an Algorithm:**

The Efficiency of an Algorithm can be measured by the following metrics. i. Time Complexity and ii. Space Complexity.

i. Time Complexity: The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum (reasonable) amount of time to complete its execution.

ii. Space Complexity: The amount of space occupied by an algorithm is known as Space Complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.

### 1. Write an algorithm for roots of a Quadratic Equation?

// Roots of a quadratic Equation

Step 1 : start

Step 2 : read a,b,c

Step 3 : if (a= 0) then step 4 else step 5

Step 4 : Write “ Given equation is a linear equation “

Step 5 :  $d = (b * b) - (4 * a * c)$

Step 6 : if ( d>0) then step 7 else step8

Step 7 : Write “ Roots are real and Distinct”

Step 8: if(d=0) then step 9 else step 10

Step 9: Write “Roots are real and equal”

Step 10: Write “ Roots are Imaginary”

Step 11: stop

### 2. Write an algorithm to find the largest among three different numbers entered by user

Step 1: Start

Step 2: Declare variables a,b and c.

Step 3: Read variables a,b and c.

Step 4: If a>b

    If a>c

        Display a is the largest number.

    Else

        Display c is the largest number.

    Else

    If b>c

        Display b is the largest number.

    Else

        Display c is the greatest number.

Step 5: Stop

## ASYMPTOTIC NOTATIONS

Asymptotic analysis of an algorithm refers to defining the mathematical bound/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm. Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as  $f(n)$  and may be for another operation it is computed as  $g(n^2)$ . This means the first operation running time will increase linearly with the increase in  $n$  and the running time of the second operation will increase exponentially when  $n$  increases. Similarly, the running time of both operations will be nearly the same if  $n$  is significantly small.

The time required by an algorithm falls under three types –

- Best Case – Minimum time required for program execution.
- Average Case – Average time required for program execution.
- Worst Case – Maximum time required for program execution.

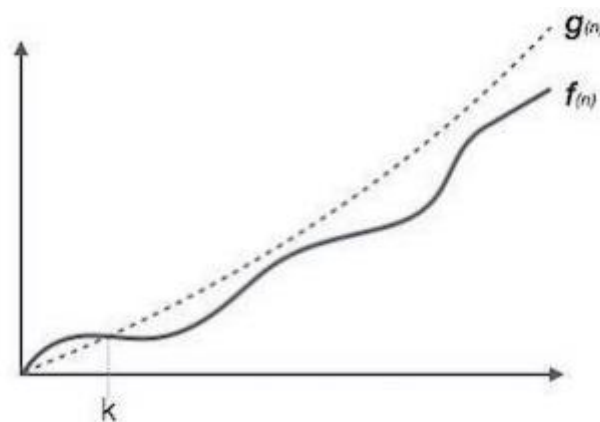
### Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- $\Omega$  Notation
- $\Theta$  Notation

### Big Oh Notation, O

The notation  $O(n)$  is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

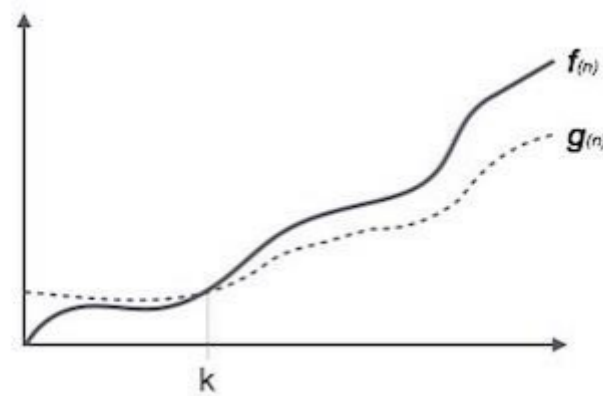


**For example,** for a function  $f(n)$

$$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n > n_0. \}$$

## Omega Notation, $\Omega$

The notation  $\Omega(n)$  is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

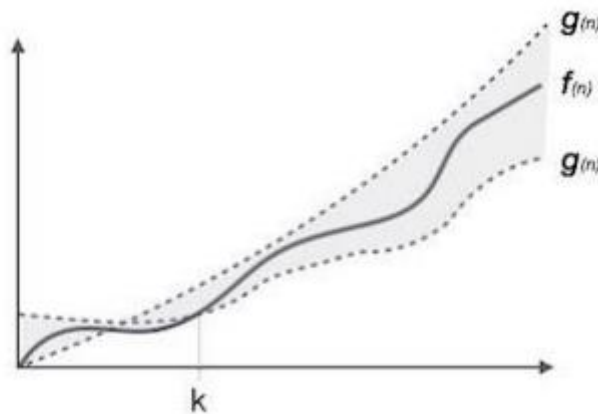


**For example,** for a function  $f(n)$

$$\Omega(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n > n_0. \}$$

## Theta Notation, $\theta$

The notation  $\theta(n)$  is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

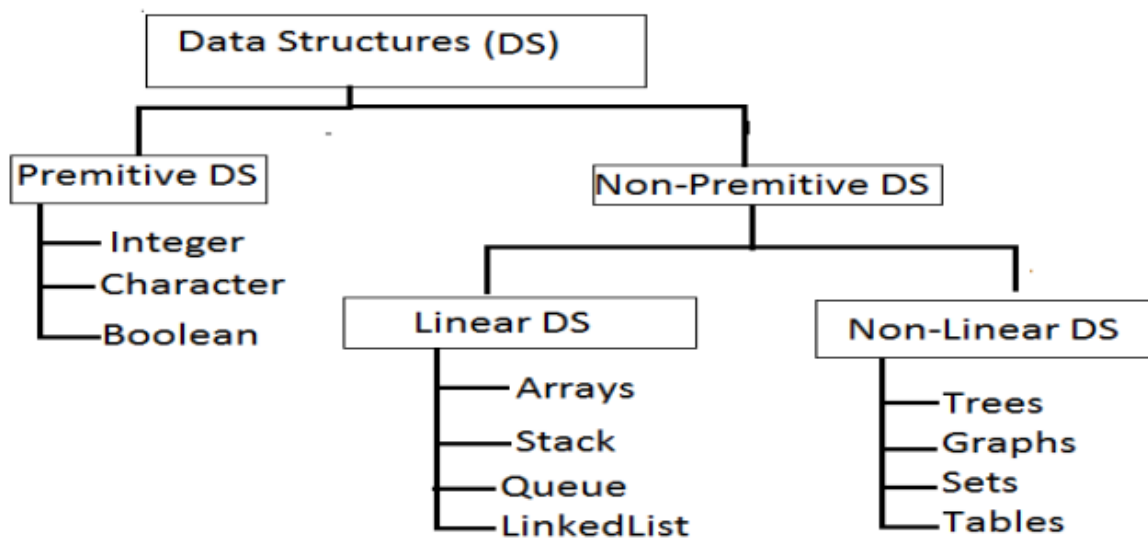


$$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$$

# DATA STRUCTURES

Data may be organized in many different ways logical or mathematical model of a program particularly organization of data. This organized data is called “Data Structure”.

Data Structure involves two complementary goals. The first goal is to identify and develop useful, mathematical entities and operations and to determine what class of problems can be solved by using these entities and operations. The second goal is to determine representation for those abstract entities to implement abstract operations on this concrete representation.



Primitive Data structures are directly supported by the language ie; any operation is directly performed in these data items.

Ex: integer, Character, Real numbers etc.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer.

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues.

In nonlinear data structures, data elements are not organized in a sequential fashion. Data structures like multidimensional arrays, trees, graphs, tables and sets are some examples of widely used nonlinear data structures.

## Operations on the Data Structures:

Following operations can be performed on the data structures:

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging

1. Traversing- It is used to access each data item exactly once so that it can be processed.

2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

3. Inserting- It is used to add a new data item in the given collection of data items.

4. Deleting- It is used to delete an existing data item from the given collection of data items.

5. Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

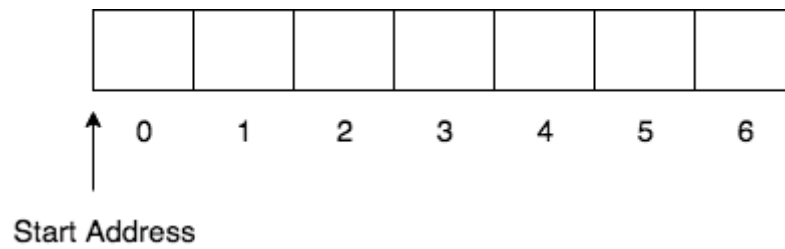
6. Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

## Arrays

Array is a linear data structure that groups elements of similar types and stores them at contiguous memory locations. The concept of arrays can be easily related with real-life scenarios. For example, percentage marks of 50 students, salary amounts of 100 employees and names of books present in a library are all examples of arrays.

Each element in an array is assigned an index number called array subscript. It is used to identify the location of element in the array. In C language, the index number of the base element (first element) of an array starts with 0. Therefore, the location of the last element of an array containing  $n$  elements is always  $n-1$ . An array is referred using the array name defined at the time of array declaration. Each individual array element is referred using array name and index number.

Let us consider an array A containing four elements. Figure 2.7 shows the logical representation of array A:



**Fig 1: Visual representation of an array**

Some of the typical operations performed on arrays include:

- 1. Insertion** Inserts an element into the array.
- 2. Deletion** Deletes an element from the array.
- 3. Traversal** Accesses each element of the array.
- 4. Sort** Rearranges the array elements in a specific order, such as ascending or descending.
- 5. Search** Searches a key value in the array.

C supports the following types of arrays:

- 1. Single-dimensional arrays** Represent elements of the array in a single column. The array A shown in Figure 2.7 is an instance of single-dimensional array.
- 2. Multi-dimensional arrays** Represent elements of an array in multiple columns and rows. A multi-dimensional array is also referred as array of arrays. Such arrays are commonly used to realize the mathematical concept of matrices.

Before an array can be used in a program, it needs to be first declared and initialized. You can initialize an array either at the time of its declaration or initialize it later in the program with the help of assignment operator.

The subsequent sections explain the declaration and initialization of single and multi-dimensional arrays along with related syntax and examples.

## Single-Dimensional Array

Array Declaration

Syntax

```
<data_type> <array_name>[size];
```



**Example:**

```
int A[10];
```

**Array initialization****Syntax**

```
<data-type> <array_name>[size]={element1,element2,.....,element};
```

```
<array_name>[index_number]=<element>;
```

**Example**

```
int A[3]={2,4,8};
```

```
A[0]=2;
```

**Array size (in bytes)**

The size of a single-dimensional array can be calculated by using the following formula:

array size = length of array \* size of data type "

**Example**

Consider the following array:

```
int A[4] = {10, 20, 30, 40};
```

Here, size of array A =  $4 * 2 = 8$  bytes.

**Array Representation**

Figure 2.8 shows the logical representation of single-dimensional array

A[i]	A[0]	A[1]	A[2]	A[3]
Memory address	10	20	30	40
	b	b+2	b+4	b+6

Logical representation of single-dimensional array

**Example :** Using single-dimensional array, write a program to find the average of 10 numbers.

**Program:** To find the average of 10 numbers

```
#include <stdio.h>

#include <conio.h>

void main()
{
    int a[10], sum, i;
    float ave;
    clrscr();
    printf("Enter ten integers: ");
    for(i=0;i<=9;i++)
        scanf("%d",&a[i]);
    sum=0;
    for(i=0;i<=9;i++)
        sum=sum+a[i];
    ave=1.0*sum/10;
    /*Displaying the results*/
    printf("\nAverage = %.2f",ave);
    getch();
}
```

***Output***

Enter ten integers:

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

Average = 5.50

## **Multi-dimensional Array**

Array Declaration

Syntax

<data-type> <array\_name> [row-subscript][column-subscript];

### **Example:**

```
int A[2][2];
```

### **Array Initialization**

#### **Syntax**

<array\_name> [row-index\_number][column-index\_number] = <element>;

<data-type> <array\_name> [row-subscript][column-subscript] = {element1, element2, ....., elementn};

#### **Example**

```
A[1][0]=3;
```

```
int A[2][2] = {1,2,3,4};
```

The above declaration statement initializes array A in the following manner:

```
A[0][0]=1
```

```
A[0][1]=2
```

```
A[1][0]=3
```

```
A[1][1]=4
```

### **Array size (in bytes)**

The size of a multi-dimensional array can be calculated by using the following formula:

array size = row-size \* column-size \* size of data type

**Example** Consider the following array:

```
int A[2][2] = {10, 20, 30, 40};
```

Here, size of array A =  $2 * 2 * 2 = 8$  bytes.

## Array Representation

Figure shows the logical representation of multi-dimensional array.

A[i][j]	A[0][0]	A[0][1]	A[1][0]	A[1][1]
Memory address	10	20	30	40
	b	b+2	b+4	b+6

Logical representation of multi-dimensional array

**Example:** Using multi-dimensional array, write a program to represent a 2 X 2 matrix.

```
#include <stdio.h>

#include <conio.h>

void main()
{
    int i,j,M[2][2];
    clrscr();
    printf("Enter the elements of the 2 X 2 matrix:\n");
    for(i=0; i<2;i++)
    for(j=0; j<2;j++)
    {
        printf("M[%d][%d] = ", i, j);
        scanf("%d",& M[i][j]);
    }
    printf("The matrix represented by the 2 X 2 2D array is:\n");
    for(i=0; i<2;i++)
    {
        printf("\n\t\t");
        for(j=0; j<2;j++)
            printf("%d",M[i][j]);
    }
}
```

Output:

Enter the elements of the 2 X 2 matrix:

$M[0][0] = 1$

$M[0][1] = 2$

$M[1][0] = 3$

$M[1][1] = 4$

The matrix represented by the 2 X 2 2D array is:

1    2

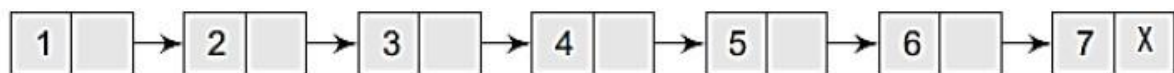
3    4

# Linked Lists, Stacks

## 2.1 LINKED LISTS-BASIC CONCEPT

A linked list is a collection of elements, but the elements are not stored in a consecutive location. A linked list is a very flexible, dynamic data structure in which elements (called nodes) form a sequential list. In a linked list, each node is allocated space as it is added to the list. Every node in the list points to the next node in the list. Therefore, in a linked list, every node contains the following two types of data:

1. The value of the node or any other data that corresponds to that node
2. A pointer or link to the next node in the list
  - The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list.
  - Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available.



**Fig.2.1** Representation of a linked list

In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

### 2.1.1 Linked List Declaration

The structure of a linked list can be defined as:

```
struct node
{
    int data;
    struct node *next;
}
```

In the above declaration, we have defined a structure named as a node consisting of two variables: an integer variable (data), and the other one is the pointer (next), which contains the address of the next node.

### 2.1.2 Advantages of Linked Lists

- Linked lists facilitate dynamic memory management by allowing elements to be added or deleted at any time during program execution.
- The use of linked lists ensure efficient utilization of memory space as only that much amount of memory space is reserved as is required for storing the list elements.
- It is easy to insert or delete elements in a linked list, unlike arrays , which require shuffling of other elements with each insert and delete operation.

### 2.1.3 Disadvantages of Linked Lists

- A linked list element requires more memory space in comparison to an array element because it has to also store the address of the next element in the list.
- Accessing an element is a little more difficult in linked list than arrays because unlike arrays, there is no index identifier associated with each list element. Thus, to access a linked list element, it is mandatory to traverse all the preceding elements.

## 2.2 LINKED LIST IMPLEMENTATION

The implementation of a linked list involves two tasks:

1. Declaring the list node
2. Defining the linked list operations

### 2.2.1 Linked List Node Declaration

Since a linked list node contains two parts, INFO and NEXT, a structure construct is best suited for its realization. The following structure declaration defines a linked list node:

```
struct node
{
    int INFO;
    struct node *NEXT;
};
typedef struct node NODE;
```

- The above structure declaration defines a new data type called NODE that represents a linked list node. The node structure contains two members, INFO for storing integer data values and NEXT for storing address of the next node.
- The statement, struct node \*NEXT, indicates that the pointer NEXT points at same structure type i.e. node. Such structures that contain pointer references to their own types are called as self-referential structures.

### 2.2.2 Linked List Operations

The typical operations performed on a linked list are:

1. Insert
2. Delete

3. Search
4. Print

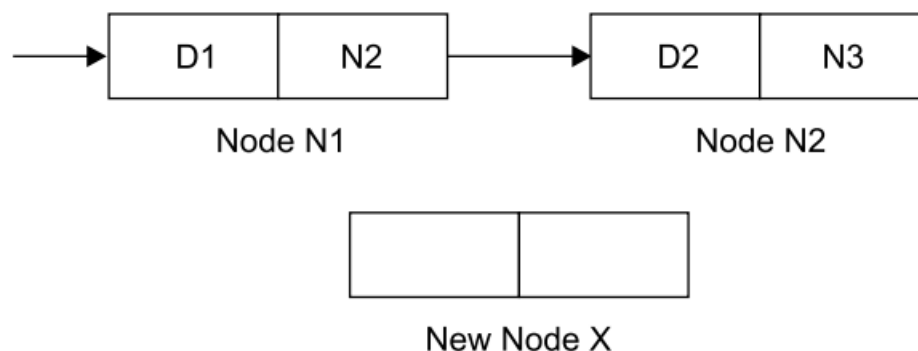
1. **Insert:** The insert operation adds a new element to the linked list. The following tasks are performed while adding the new element
  - a) Memory space is reserved for the new node.
  - b) The element is stored in the INFO part of the new node.
  - c) The new node is connected to the existing nodes in the list.

Depending on the location where the new node is to be added, there are three scenarios possible, which are:

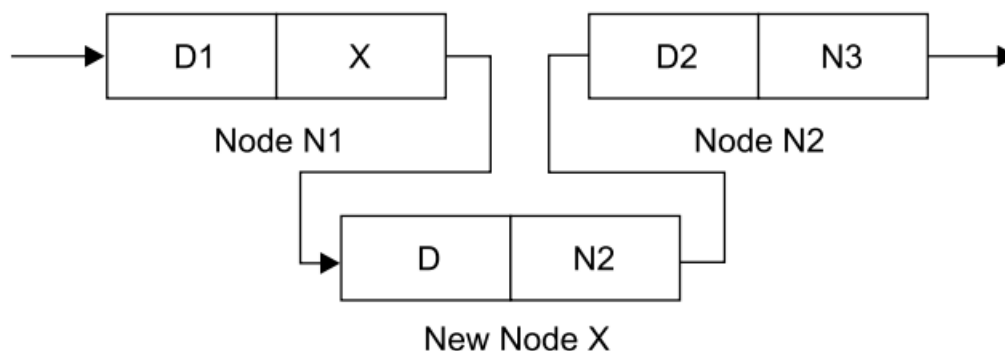
- i. Inserting the new element at the beginning of the list
- ii. Inserting the new element at the end of the list
- iii. Inserting the new element somewhere at the middle of the list

Inserting a new element at the beginning or end of the list is easy as it only requires resetting the respective NEXT fields. However, if the new element is to be added somewhere at the middle of the list then a search operation is required to be performed to identify the point of insertion.

Figures 2.2 (a) and (b) show the insertion of a new element between two existing elements of a linked list



**Fig 2.2(a)** Creating a new element



**Fig 2.2 (b)** Inserting the newly created element



**Example** Write an algorithm to insert an element at the end of a linked list.

insert (value)

Step 1: Start

Step 2: Set PTR = addressof (New Node) //Allocate a new node and assign its address to the pointer PTR

Step 3: Set PTR->INFO = value; //Store the element value to be inserted in the INFO part of the new node

Step 4: If FIRST = NULL, then go to Step 5 else go to Step 7 //Check whether the existing list is empty.

Step 5: Set FIRST=PTR and LAST=PTR //Update the FIRST and LAST pointers

Step 6: Set PTR->NEXT = NULL and goto Step 8

Step 7: Set LAST->NEXT=PTR, PTR->NEXT=NULL and LAST=PTR //Link the newly created node at the end of the list

Step 8: Stop

2. **Delete:** The delete operation removes an existing element from the linked list.

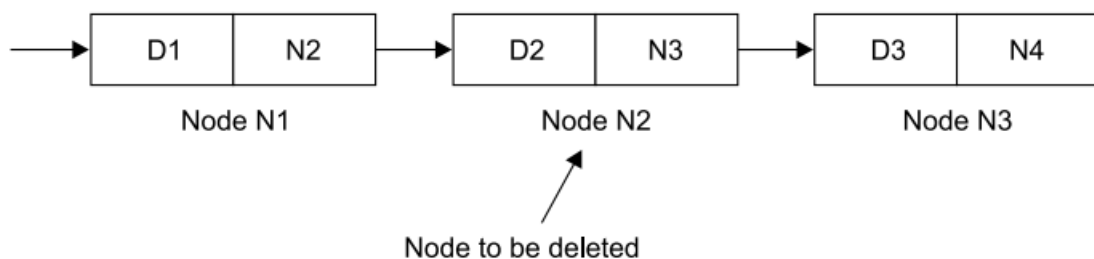
The following tasks are performed while deleting an existing element:

- The location of the element is identified.
- The element value is retrieved. In some cases, the element value is simply ignored.
- The link pointer of the preceding node is reset.

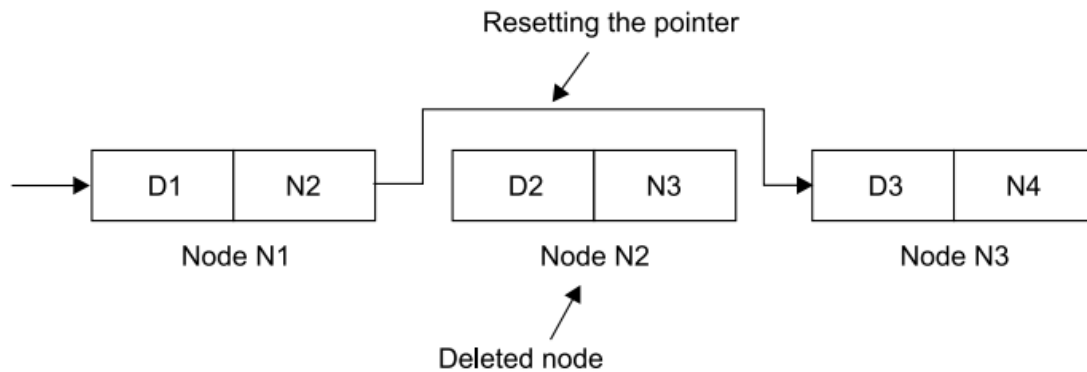
Depending on the location from where the element is to be deleted, there are three scenarios possible, which are:

- Deleting an element from the beginning of the list.
- Deleting an element from the end of the list.
- Deleting an element somewhere from the middle of the list.

Deleting an element from the beginning or end of the list is easy as it only requires resetting the first and last pointers. However, if an element is to be deleted from within the list then a search operation is required to be performed for locating that element. Figures 2.3 (a) and (b) show the deletion of an element that is present between two existing elements of a linked list.



**Fig 2.3 (a)** Identifying the node to be deleted



**Fig 2.3 (b)** Deleting the node

**Example** Write an algorithm to delete a specific element from a linked list.

delete (value)

Step 1: Start

Step 2: Set LOC = search (value) //Call the search module to search the location of the node to be deleted and assign it to LOC pointer

Step 3: If LOC=NULL goto Step 4 else goto Step 5

Step 4: Return ("Delete operation unsuccessful: Element not present") and Stop

Step 5: If LOC=FIRST goto Step 6 else goto Step 10//Check if the element to be deleted is the first element to be deleted

Step 6: If FIRST=LAST goto Step 7 else goto Step 8 //Check if there is only one element in the list

Step 7: Set FIRST=LAST=NULL and goto Step 9

Step 8: Set FIRST=FIRST->NEXT

Step 9: Return ("Delete operation successful") and Stop

Step 10: Set TEMP=LOC-1 //Assign the location of the node present before LOC to temporary pointer TEMP

Step 11: Set TEMP->NEXT=LOC->NEXT //Link the TEMP node with the node being currently pointed by LOC

Step 12: If LOC=LAST goto Step 13 else goto Step 14 //Check if the element to be deleted is currently the last element in the list

Step 13: Set LAST=TEMP

Step 14: Return ("Delete operation successful")

Step 15: Stop

3. **Search:** The search operation helps to find an element in the linked list.

The following tasks are performed while searching an element:

- Traverse the list sequentially starting from the first node.
- Return the location of the searched node as soon as a match is found.
- Return a search failure notification if the entire list is traversed without any match.

The NEXT pointers help in traversing the linked list from start till end.

**Example** Write an algorithm to search a specific element in the linked list.

search (value)

Step 1: Start

Step 2: If FIRST=NULL goto Step 3 else goto Step 4 //Check if the linked list is empty

Step 3: Return (“Search unsuccessful: Element not present”) and Stop

Step 4: Set PTR=FIRST

Step 5: Repeat Steps 6-8 until PTR!=LAST //Repeat Steps 6-8 until PTR is not equal to LAST

Step 6: If PTR->INFO=value goto Step 7 else goto Step 8

Step 7: Return (“Search successful”, PTR) and Stop

Step 8: Set PTR=PTR->NEXT

Step 9: If LAST->INFO=value goto Step 10 else goto Step 11 //Check if the element to be searched is the last element in the list

Step 10: Return (“Search successful”, LAST) and Stop

Step 11: Return (“Search unsuccessful: Element not present”)

Step 12: Stop

4. **Print** :The print operation prints or displays the linked list elements on the screen. To print the elements, the linked list is traversed from start till end using NEXT pointers.

**Example** Write an algorithm to print all the linked list elements.

print ()

Step 1: Start

Step 2: If FIRST=NULL goto Step 3 else goto Step 4 //Check if the linked list is empty

Step 3: Display (“Empty List”) and Stop

Step 4: If FIRST=LAST goto Step 5 else goto Step 6 //Check if the list has only one element

Step 5: Display (FIRST->INFO) and Stop

Step 6: Set PTR=FIRST

Step 7: Repeat Steps 8-9 until PTR!=LAST //Repeat Steps 8-9 until PTR is not equal to LAST

Step 8: Display (PTR->INFO) //Displaying list elements

Step 9: Set PTR=PTR->NEXT

Step 10: Display (LAST->INFO) //Displaying last element

Step 11: Stop

### 2.2.3 Linked List Implementation

Linked list implementation involves declaring its structure and defining its operations. The following example shows how a linked list is implemented using C language.

**Example** Write a program to implement a linked list and perform its common operations.

```

#include<stdio.h> #include<stdio.h>
/*Linked list declaration*/
struct node {
    int INFO;
    struct node *NEXT;
};
/*Declaring pointers to first and last node of the linked list*/
struct node *FIRST = NULL;
struct node *LAST = NULL;
/*Declaring function prototypes for linked list operations*/
void insert(int);
int delete(int);
void print(void);
struct node *search (int);
void main()
{
    int num1, num2, choice;
    struct node *location;
    /*Displaying a menu of choices for performing linked list operations*/
    while(1)
    {
        clrscr();
        printf("\n\nSelect an option\n");
        printf("\n1 - Insert");
        printf("\n2 - Delete");
        printf("\n3 - Search");
        printf("\n4 - Print");
        printf("\n5 - Exit");
        printf("\n\nEnter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        { case 1:
            {
                printf("\nEnter the element to be inserted into the linked list: ");
                scanf("%d",&num1);
                insert(num1); /*Calling the insert() function*/
                printf("\n%d successfully inserted into the linked list!",num1);
                getch();
                break;
            }

```

```

case 2:
{
printf("\nEnter the element to be deleted from the linked list: ");
scanf("%d",&num1);
num2=delete(num1); /*Calling the delete() function */
if(num2==-9999)
    printf("\n\t%d is not present in the linked list\n\t",num1);
else
    printf("\n\tElement %d successfuly deleted from the linked list\n\t",num2);
    getch();
    break;
}
case 3:
{
printf("\nEnter the element to be searched: ");
scanf("%d",&num1);
location=search(num1); /*Calling the search() function*/
if(location==NULL)
    printf("\n\t%d is not present in the linked list\n\t",num1);
else
{
    if(location==LAST)
        printf("\n\tElement %d is the last element in the list",num1);
    else
        printf("\n\tElement %d is present before element %d in the linked
list\n\t",num1,(location->NEXT)->INFO);
}
    getch();
    break;
}
case 4:
{
    print(); /*Printing the linked list elements*/
    getch();
    break;
}
case 5:
{
    exit(1);
    break;
}

```

```

}
default:
{
printf("\nIncorrect choice. Please try again.");
getch();
break;
}
}
}
}
/*Insert function*/
void insert(int value)
{
/*Creating a new node*/
struct node *PTR = (struct node*)malloc(sizeof(struct node));
/*Storing the element to be inserted in the new node*/
PTR->INFO = value;
/*Linking the new node to the linked list*/
if(FIRST==NULL)
{
FIRST = LAST = PTR;
PTR->NEXT=NULL;
} else
{
LAST->NEXT = PTR;
PTR->NEXT = NULL;
LAST = PTR;
}
}
/*Delete function*/
int delete(int value)
{
struct node *LOC,*TEMP;
int i;
i=value;
LOC=search(i); /*Calling the search() function*/
if(LOC==NULL) /*Element not found*/
return(-9999);
if(LOC==FIRST)
{

```

```

if(FIRST==LAST)
    FIRST=LAST=NULL;
else
    FIRST=FIRST->NEXT;
    return(value);
}
for(TEMP=FIRST;TEMP->NEXT!=LOC;TEMP=TEMP->NEXT)
; /* Here, a single semi-colon indicates that the for loop is not executing any
instructions; it is simply used to update the TEMP pointer through linked list
traversal*/
TEMP->NEXT=LOC->NEXT;
if(LOC==LAST)
LAST=TEMP;
return(LOC->INFO);
}
/*Search function*/
struct node *search (int value)
{
struct node *PTR; if(FIRST==NULL) /*Checking for empty list*/
return(NULL);
/*Searching the linked list*/
for(PTR=FIRST;PTR!=LAST;PTR=PTR->NEXT)
if(PTR->INFO==value) return(PTR); /*Returning the location of the searched
element*/
if(LAST->INFO==value)
return(LAST);
else
return(NULL); /*Returning NULL value indicating unsuccessful search*/
}
/*print function*/
void print()
{
struct node *PTR; if(FIRST==NULL) /*Checking whether the list is empty*/
{
printf("\n\tEmpty List!!");
return;
}
printf("\nLinked list elements:\n");
if(FIRST==LAST) /*Checking if there is only one element in the list*/
{

```

```

        printf("\t%d",FIRST->INFO);
        return;
    } /*Printing the list elements*/
    for(PTR=FIRST;PTR!=LAST;PTR=PTR->NEXT)
        printf("\t%d",PTR->INFO);
    printf(«\t%d»,LAST->INFO);
}

```

## Output

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 4

Empty List!!

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the element to be inserted into the linked list: 1

1 successfully inserted into the linked list!

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 1

Enter the element to be inserted into the linked list: 2

2 successfully inserted into the linked list!

Select an option

- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print



5 – Exit

Enter your choice: 1

Enter the element to be inserted into the linked list: 3

3 successfully inserted into the linked list!

Select an option

1 - Insert

2 - Delete

3 - Search

4 - Print

5 - Exit

Enter your choice: 3

Enter the element to be searched: 5

5 is not present in the linked list

Select an option

1 - Insert

2 - Delete

3 - Search

4 - Print

5 - Exit

Enter your choice: 3

Enter the element to be searched: 2

Element 2 is present before element 3 in the linked list

Select an option

1 - Insert

2 - Delete

3 – Search

4 - Print

5 - Exit

Enter your choice: 2

Enter the element to be deleted from the linked list: 2

Element 2 successfully deleted from the linked list

Select an option

1 - Insert

2 - Delete

3 - Search

4 - Print

5 - Exit

Enter your choice: 4

Linked list elements: 1 3

Select an option

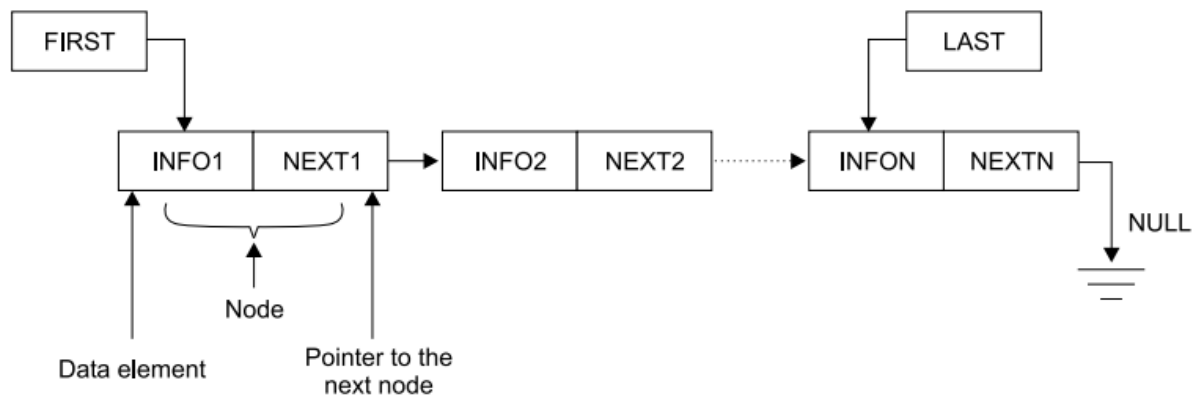
- 1 - Insert
- 2 - Delete
- 3 - Search
- 4 - Print
- 5 - Exit

Enter your choice: 5

## 2.3 TYPES IF LINKED LIST

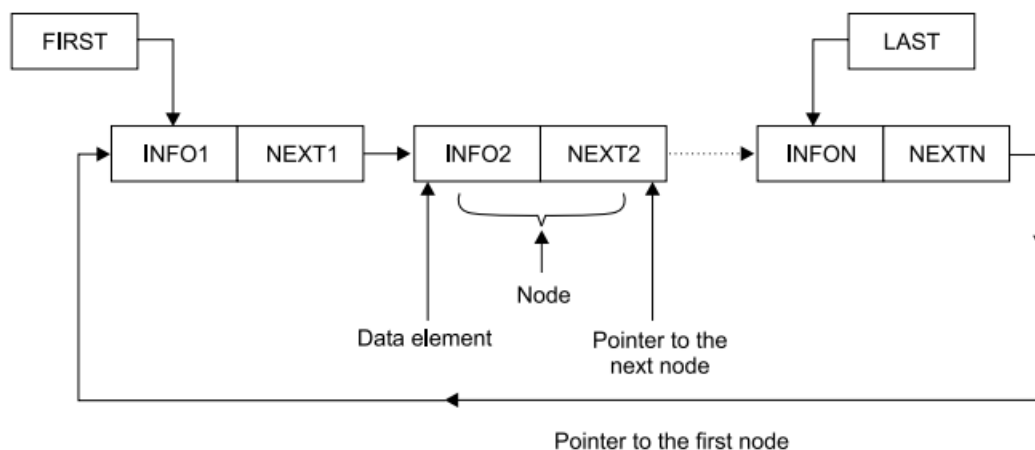
Depending on the manner in which nodes are interconnected with each other, linked lists are categorized into the following types:

1. **Singly linked list** In this type of linked list, each node points at the successive node. Thus, the list can only be traversed in the forward direction. The linked list implementation that we saw in the previous section is an example of singly linked list.



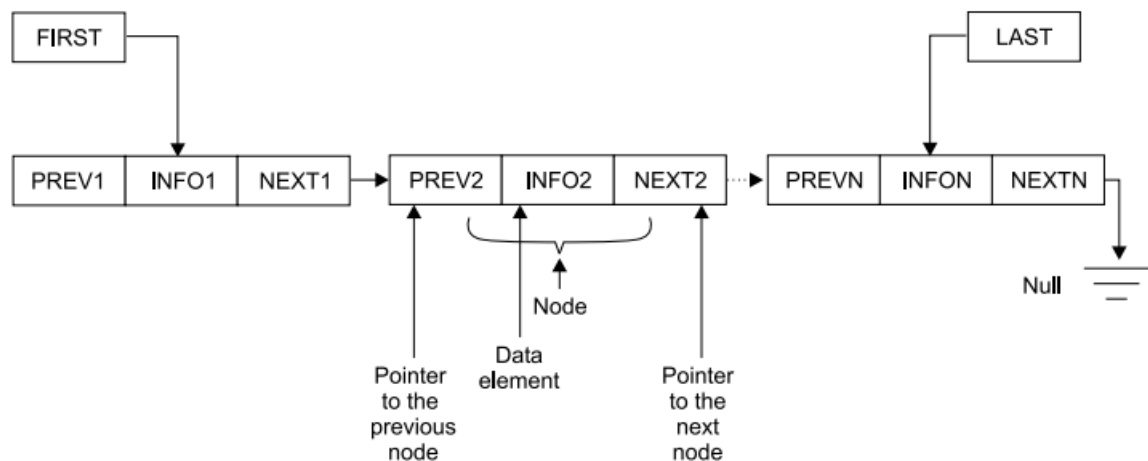
**Fig 2.4 (a)** Logical representation of a Singly linked list

2. **Circular list** In this type of the linked list, the first and the last node are logically connected with each other, thus giving the impression of a circular list formation. Actually, the NEXT part of the last node contains the address of the FIRST node, thus connecting the rear of the list to its front.



**Fig 2.4 (b)** Logical representation of a Circular linked list

3. **Doubly linked list** In this type of linked list, a node points at both its preceding as well as succeeding nodes. Thus, the list can be traversed in both forward as well as backward directions.



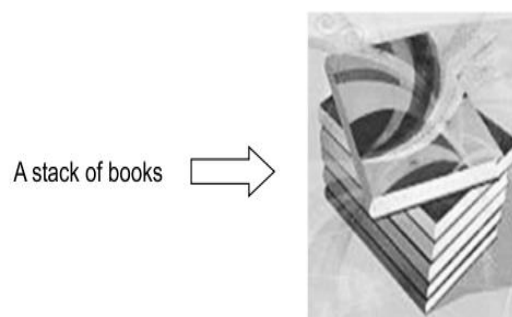
**Fig 2.4 (c)** Logical representation of a doubly linked list

## 2.4 STACKS

Previously we learnt how arrays are used for implementing linear data structures. Arrays provide the flexibility of adding or removing elements anywhere in the list. But there are certain linear data structures that permit the insertion and deletion operations only at the beginning or end of the list, but not in the middle. Such data structures have significant importance in systems processes such as compilation and program control.

- Stack is a linear data structure in which items are added or removed only at one end, called top of the stack.
- Thus, there is no way to add or delete elements anywhere else in the stack.
- A stack is based on Last-In-First-Out (LIFO) principle that means the data item that is inserted last into the stack is the first one to be removed from the stack.

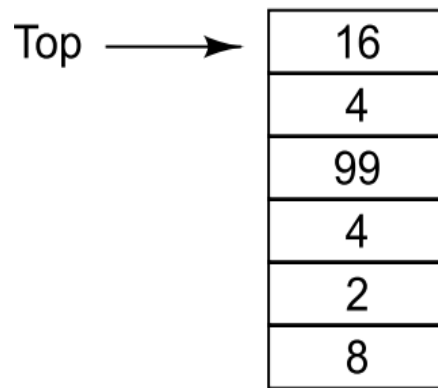
We can relate a stack to certain real-life objects and situations, as shown in Figs. 2.5.



**Fig 2.5** Stack of Books

### 2.4.1 Stack Representation in Memory

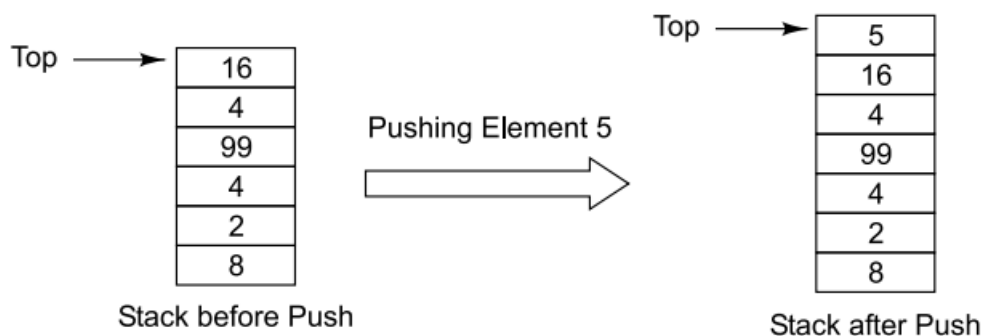
Just like their real world counterparts, stacks appear as a group of elements stored at contiguous locations in memory. Each successive insert or delete operation adds or removes an item from the group. The top location of the stack or the point of addition or deletion is maintained by a pointer called top. Figure 2.6 shows the logical representation of stacks in memory.



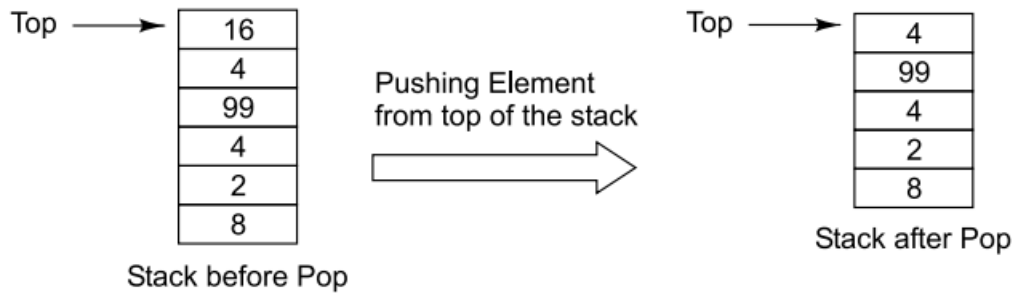
**Fig 2.6** Logical representation of stacks

### 2.5 STACK OPERATION

There are two key operations associated with the stack data structure: push and pop. Adding an element to the stack is referred as push operation while reading or deleting an element from the stack is referred as pop operation. Figures 2.7 (a) and (b) depict the push and pop operations on a stack.



**Fig. 2.7(a)** Push operation



**Fig. 2.7(b)** Pop operation

### 2.5.1 Push

As we can see in Fig. 2.7 (a), the push operation involves the following subtasks:

- Receiving the element to be inserted.
- Incrementing the stack pointer, top.
- Storing the received element at new location of top

### 2.5.2 Pop

As we can see in Fig. 2.7 (b), the pop operation involves the following subtasks:

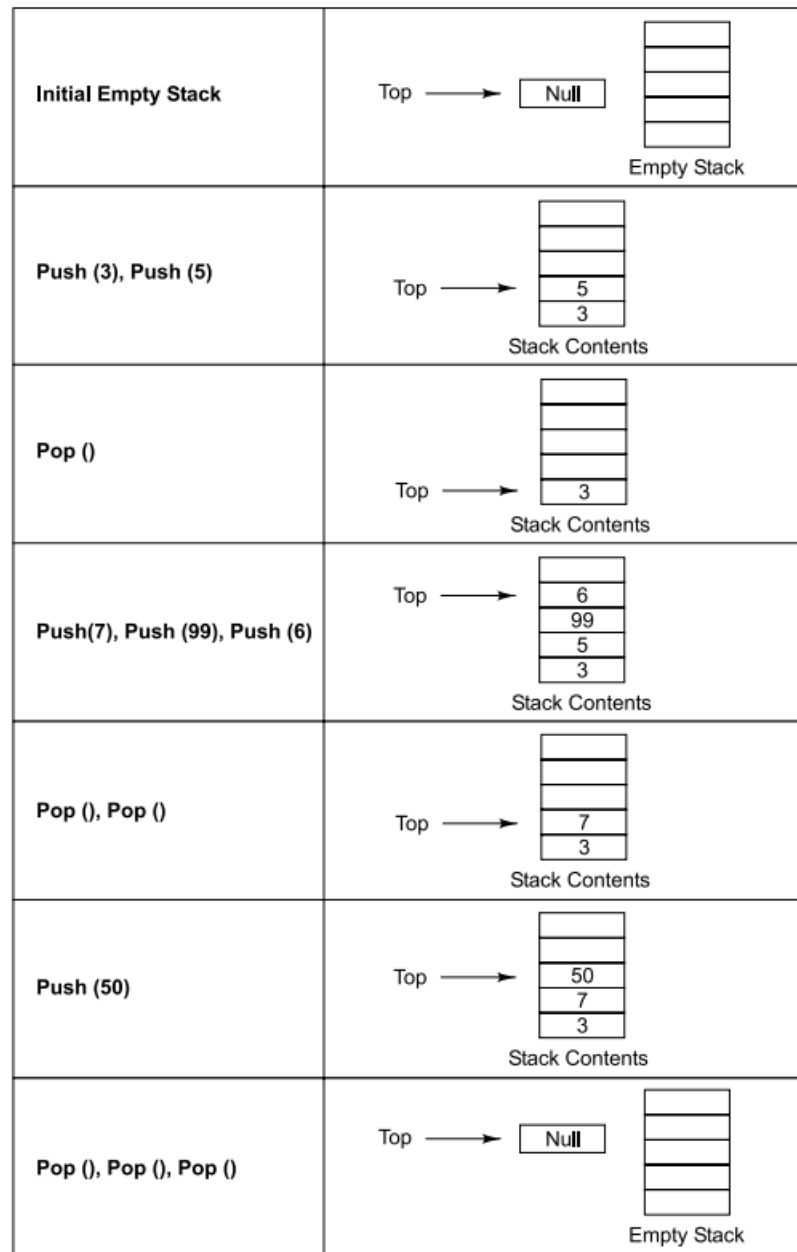
- Retrieving or removing the element at the top of the stack.
- Decrementing the stack pointer, top.

## 2.6 STACK IMPLEMENTATION

- Stack implementation involves choosing the data storage mechanism for storing stack elements and implementing methods for performing the two stack operations, push and pop.
- A typical implementation of the push operation checks if there is any room left in the stack, and if there is any, it increments the stack counter by one and inserts the received item at the top of the stack.
- Similarly, the implementation of the pop operation checks whether or not the stack is already empty, if it is not, it removes the top element of the stack and decrements the stack counter by one.
- We can implement stacks by using arrays or linked lists.
- The advantages or disadvantages of array or linked implementations of stacks are the same that are associated with such types of data structures. However, both implementation types have their own usage in specific situations.

### 2.6.1 Array Implementation of Stacks

The array implementation of stacks involves allocation of fixed size array in the memory. Both stack operations (push and pop) are made on this array with a constant check being made to ensure that the array does not go out of bounds.



**Fig 2.8** Stack operations

- **Push Operation** The push operation involves checking whether or not the stack pointer is pointing at the upper bound of the array.
- If it is not, the stack pointer is incremented by 1 and the new item is pushed (inserted) at the top of the stack.

**Example** Write an algorithm to implement the push operation under array representation of stacks.

push(stack[MAX],element)

Step 1: Start

Step 2: If  $\text{top} = \text{MAX}-1$  goto Step 3 else goto Step 4

Step 3: Display message "Stack Full" and exit

Step 4:  $\text{top} = \text{top} + 1$

Step 5:  $\text{stack}[\text{top}] = \text{element}$

Step 6: Stop

The above algorithm inserts an element at the top of a stack of size MAX.

- Pop Operation The pop operation involves checking whether or not the stack pointer is already pointing at NULL (empty stack).
- If it is not, the item that is being currently pointed is popped (removed) from the stack (array) and the stack pointer is decremented by 1.

**Example** Write an algorithm to implement the pop operation under array representation of stacks.

`pop(stack[MAX],element)`

Step 1: Start

Step 2: If  $\text{top} = -1$  goto Step 3 else goto Step 4

Step 3: Display message "Stack Empty" and exit

Step 4: Return  $\text{stack}[\text{top}]$  and set  $\text{top} = \text{top} - 1$

Step 5: Stop

The above algorithm removes the element at the top of the stack.

## 2.6.2 Linked Implementation of Stacks

- The linked implementation of stacks involves dynamically allocating memory space at run time while performing stack operations.
- Since, the allocation of memory space is dynamic, the stack consumes only that much amount of space as is required for holding its data elements. This is contrary to array implemented stacks which continue to occupy a fixed memory space even if there are no elements present.
- Thus, linked implementation of stacks based on dynamic memory allocation technique prevents wastage of memory space.

**Push Operation** The push operation under linked implementation of stacks involves the following tasks:

- Reserving memory space of the size of a stack element in memory.
- Storing the pushed (inserted) value at the new location.
- Linking the new element with existing stack .
- Updating the stack pointer

**Example** Write an algorithm to implement the push operation under linked representation of stacks.

push(structure stack, element, next, value)

Step 1: Start

Step 2: Set ptr=(struct stack\*)malloc(sizeof(struct stack)), to reserve a block of memory for the new stack node and assign its address to pointer ptr

Step 3: Set ptr->element=value, to copy the inserted value into the new node

Step 4: Set ptr->next=top, to link the new node to the current top node

Step 5: Set top = ptr to designate the new node as the top node

Step 6: Return

Step 7: Stop

The above algorithm inserts an element at the top of the stack.

**Pop Operation** The pop operation under linked implementation of stacks involves the following tasks:

- Checking whether the stack is empty .
- Retrieving the top element of the stack.
- Updating the stack pointer .
- Returning the retrieved (popped) value

**Example** Write an algorithm in C to implement the pop operation under linked representation of stacks.

pop(structure stack, element, next)

Step 1: Start

Step 2: If top = NULL goto Step 3 else goto Step 4

Step 3: Display message "Stack Empty" and exit

Step 4: Set temp=top->element, to retrieve the element at top node of the stack

Step 5: Set top=top->next, to designate the next stack node as the top node

Step 6: Return temp

Step 7: Stop

The above algorithm removes the element at the top of the stack.





# Queues

## 3.1 QUEUES – BASIC CONCEPT

- Queue is a linear data structure in which items are inserted at one end called 'Rear' and deleted from the other end called 'Front'.
- Queues are based on the First-In-First-Out (FIFO) principle that means the data item that is inserted first in the queue is also the first one to be removed from the queue.

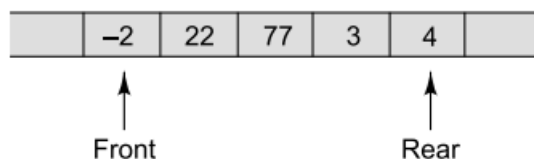


**Fig. 3.1** Queue of people

As we can see in Fig. 3.1, a person can join a queue of waiting people only at its tail end while the person who joined the queue first becomes the first one to leave the queue.

### 3.1.1 Logical Representation of Queues

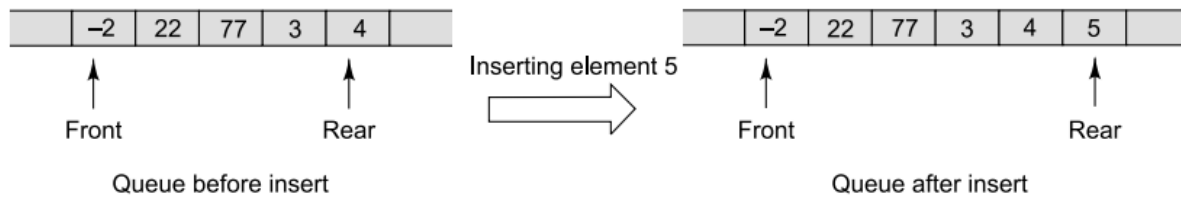
Just like their real world counterparts, queues appear as a group of elements stored at contiguous locations in memory. Each successive insert operation adds an element at the rear end of the queue while each delete operation removes an element from the front end of the queue. The location of the front and rear ends are marked by two distinct pointers called front and rear.



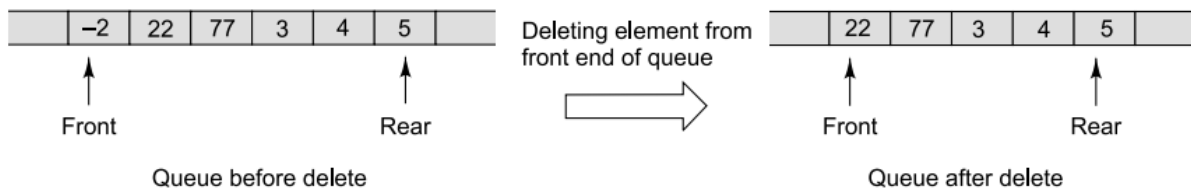
**Fig. 3.2** Logical representation of queues

## 3.2 QUEUE OPERATIONS

There are two key operations associated with the queue data structure: insert and delete. The insert operation adds an element at the rear end of the queue while the delete operation removes an element from the front end of the queue. Figures 3.3 (a) and (b) depict the insert and delete operations on a queue.



**Fig. 3.3(a)** Insert operation



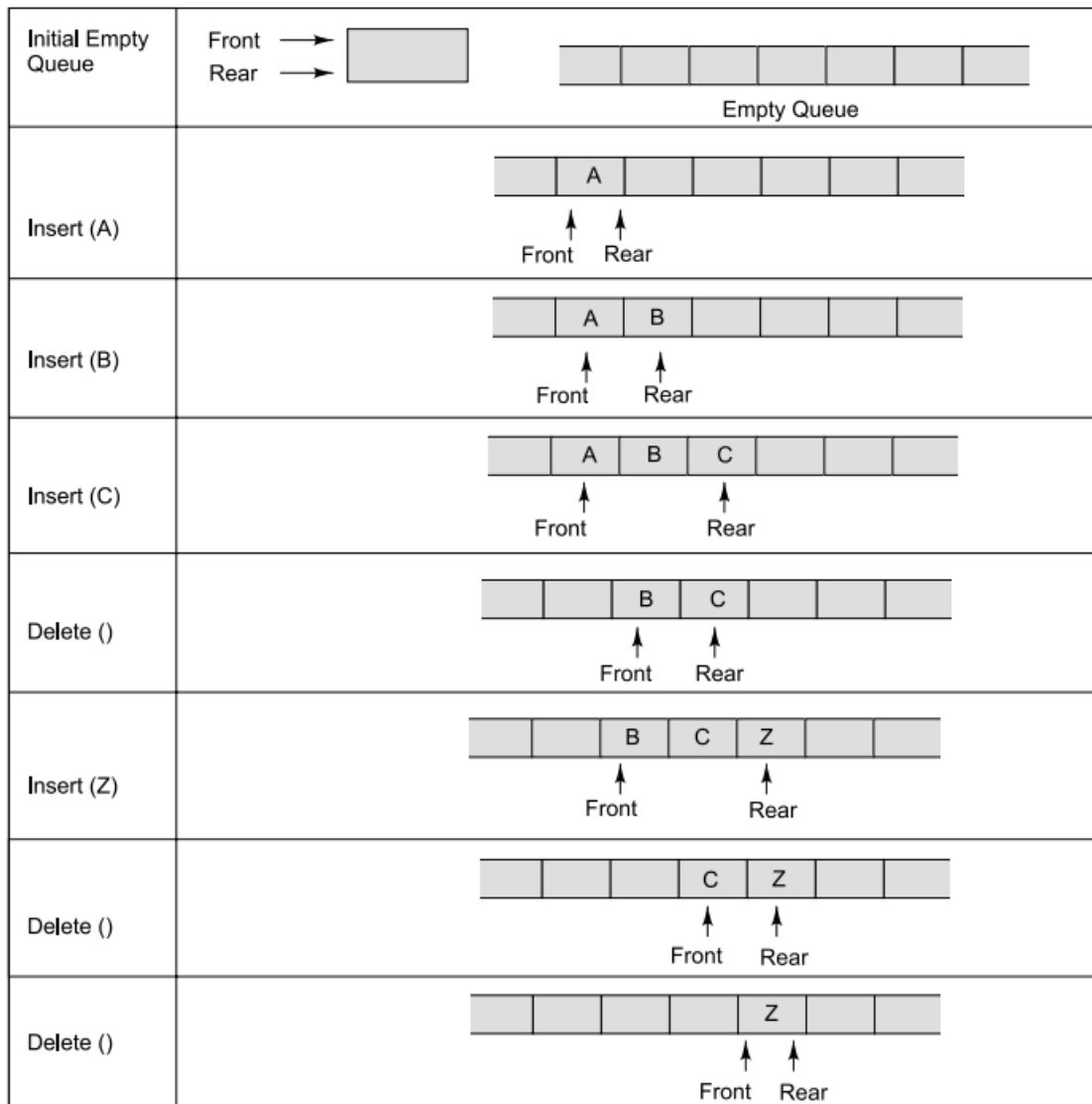
**Fig. 3.3(b)** Insert operation

**Insert** operation involves the following subtasks:

- Receiving the element to be inserted.
- Incrementing the queue pointer, rear.
- Storing the received element at new location of rear.

**Delete** operation involves the following subtasks:

- Retrieving or removing the element from the front end of the queue.
- Incrementing the queue pointer, front, to make it point to the next element in the queue.



**Fig. 3.4** Queue operations

### 3.3 QUEUE IMPLEMENTATION

Like stacks, we can implement queues by using arrays or linked lists. The advantages or disadvantages of array or linked implementations of queues are the same that are associated with such types of data structures. However, both types of implementation have their own usage in specific situations.

#### 3.3.1 Array Implementation of Queues

The array implementation of queues involves allocation of fixed size array in the memory. Both queue operations (insert and delete) are performed on this array with a constant check being made to ensure that the array does not go out of bounds.

**Insert** Operation involves checking whether or not the queue pointer rear is pointing at the upper bound of the array. If it is not, rear is incremented by 1 and the new item is added at the end of the queue.

**Example** Write an algorithm to realize the insert operation under array implementation of queues.

insert(queue[MAX], element, front, rear)

Step 1: Start

Step 2: If front = NULL goto Step 3 else goto Step 6

Step 3: front = rear = 0

Step 4: queue[front]=element

Step 5: Goto Step 10

Step 6: if rear = MAX-1 goto Step 7 else goto Step 8

Step 7: Display the message, “Queue is Full” and goto Step 10

Step 8: rear = rear +1

Step 9: queue[rear] = element

Step 10: Stop

**Delete** operation involves checking whether or not the queue pointer front is already pointing at NULL (empty queue). If it is not, the item that is being currently pointed is removed from the queue (array) and the front pointer is incremented by 1.

**Example** Write an algorithm to realize the delete operation under array implementation of queues.

delete(queue[MAX],front, rear)

Step 1: Start

Step 2: If front = NULL and rear = NULL goto Step 3 else goto Step 4

Step 3: Display the message, “Queue is Empty” and goto Step 10

Step 4: if front != NULL and front = rear goto Step 5 else goto Step 8

Step 5: Set i = queue[front]

Step 6: Set front = rear = -1

Step 7: Return the deleted element i and goto Step 10

Step 8: Set i = queue[front]

Step 9: Return the deleted element i

Step 10: Stop

### 3.3.2 Linked Implementation of Queues

The linked implementation of queues involves dynamically allocating memory space at run time while performing queue operations.

**Insert** Operation under linked implementation of queues involves the following tasks:

- Reserving memory space of the size of a queue element in memory .
- Storing the added (inserted) value at the new location .
- Linking the new element with existing queue .
- Updating the rear pointer

**Example** Write an algorithm to realize the insert operation under linked implementation of queues.

insert(structure queue, value, front, rear)

Step 1: Start

Step 2: Set ptr=(struct queue\*)malloc(sizeof(struct queue)), to reserve a block of memory for the new queue node and assign its address to pointer ptr

Step 3: Set ptr->element=value, to copy the inserted value into the new node

Step 4: if front = NULL goto Step 5 else goto Step 7

Step 5: Set front = rear = ptr

Step 6: Set ptr->next=NULL and goto Step 10

Step 7: Set rear->next=ptr

Step 8: Set ptr->next=NULL

Step 9: Set rear = ptr

Step 10: Stop

**Delete** Operation under linked implementation of queues involves the following tasks

- Checking whether the queue is empty.
- Retrieving the front most element of the queue.
- Updating the front pointer.
- Returning the retrieved (removed) value

**Example** Write an algorithm to realize the delete operation under linked implementation of queues.

delete(structure queue, front, rear)

Step 1: Start

Step 2: if front = NULL goto Step 3 else goto Step 4

Step 3: Display message, "Queue is Empty" and goto Step 7

Step 4: Set  $i = \text{front} \rightarrow \text{element}$

Step 5: Set  $\text{front} = \text{front} \rightarrow \text{next}$

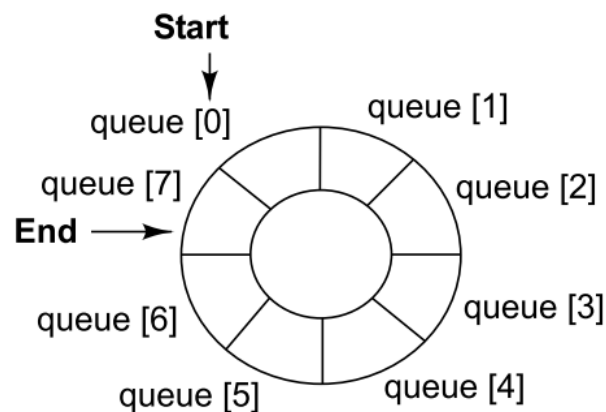
Step 6: Return the deleted element  $i$

Step 7: Stop

### 3.4 TYPES OF QUEUES

#### 3.4.1 Circular Queue

A circular queue is a more efficient version of a simple queue where the last position is connected back to the first position to make a circle. This avoids the issue of wasted space in a simple queue.



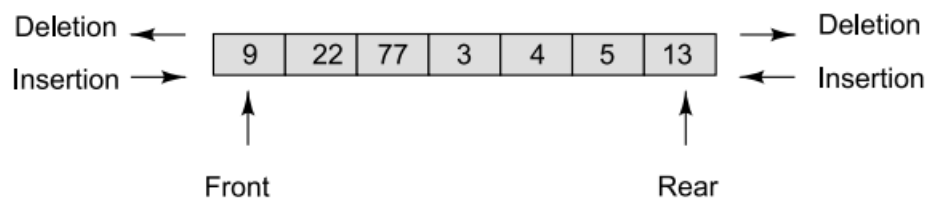
**Fig. 3.5** Circular Queue

#### 3.4.2 Priority Queue

A queue where elements are removed based on priority rather than their order in the queue. Higher priority elements are dequeued before lower priority ones.

#### 3.4.3 Double-Ended Queue

A queue where insertion and deletion can happen at both the front and the rear. It is a generalization of both stacks and queues.



**Fig. 3.6** Double-ended Queue

# MODULE -4 TREES

## Trees

### Introduction

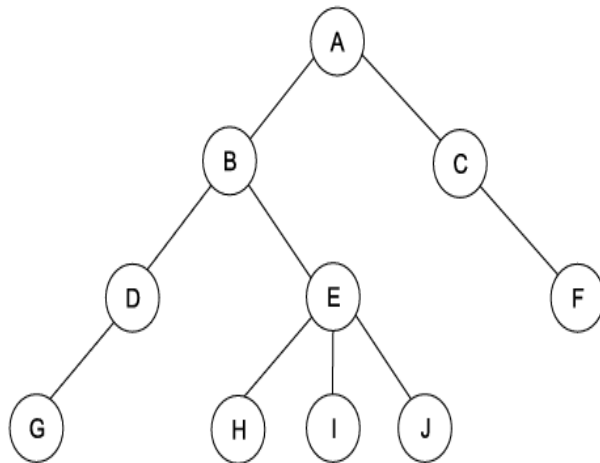
Tree is one such non-linear data structure which stores the data elements in a hierarchical manner. Each node of the tree stores a data value, and is linked to other nodes in a hierarchical fashion.

### Basic Concept

A tree is defined as a finite set of elements or nodes, such that

1. One of the nodes present at the top of the tree is marked as root node.
2. The remaining elements are partitioned across multiple subtrees present below the root node.

Figure 8.1 shows a sample tree T.



Here, T is a simple tree containing ten nodes with A being the root node. The node A contains two subtrees. The left subtree starts at node B while the right subtree starts at node C. Both the subtrees further contain subtrees below them, thus indicating recursive nature of the tree data structure. Each node in the tree has zero or more child nodes.

### Tree terminology

There are a number of key terms associated with trees.

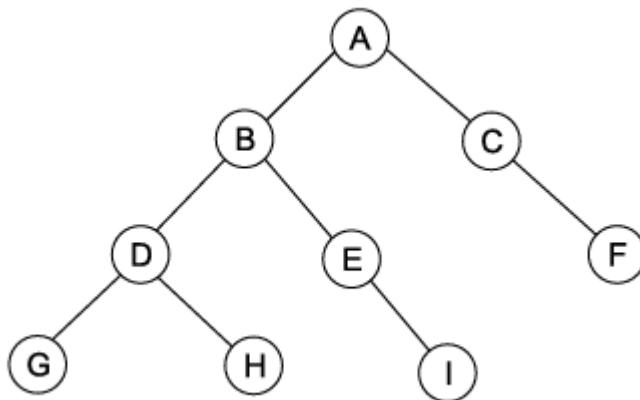
Key Term	Description	Example
Node	It is the data element of a tree. Apart from storing a value, it also specifies links to the other nodes.	A, B, C, D
Root	It is the top node in a tree.	A
Parent	A node that has one or more child nodes present below it is referred as parent node.	B is the parent node of D and E
Child	All nodes in a tree except the root node are child nodes of	H, I and J are child nodes of E



	their immediate predecessor nodes.	
Leaf	It is the terminal node that does not have any child nodes.	G, H, I, J and F are leaf nodes
Internal node	All nodes except root and leaf nodes are referred as internal nodes.	B, C, D and E are internal nodes
Sibling	All the child nodes of a parent node are referred as siblings.	D and E are siblings
Degree	The degree of a node is the number of subtrees coming out of the node.	Degree of A is 2 Degree of E is 3
Level	All the tree nodes are present at different levels. Root node is at level 0, its child nodes are at level 1, and so on.	A is at level 0 B and C are at level 1 G, H, I, J are at level 3
Depth or Height	It is the maximum level of a node in the tree.	Depth of tree T is 3
Path	It is the sequence of nodes from source node till destination node.	A–B–E–J

## Binary Tree

Binary tree is one of the most widely used non-linear data structures. It is a restricted form of a general tree. The restriction that it applies to a general tree is that its nodes can have a maximum degree of 2. That means, the nodes of a binary tree can have zero, one or two child nodes but not more than that.



As shown in the above binary tree, all nodes have a maximum degree of 2. The maximum number of nodes that can be present at level  $n$  is  $2^n$ .

## Binary tree concepts

Before we learn how binary trees are represented in memory, let us discuss some of the key concepts associated with binary trees. Table 8.2 lists these key concepts.

Concept	Description	Example
Strictly binary tree	A binary tree is called strictly binary if all its nodes barring the leaf nodes contain two child nodes.	
Complete binary tree	A binary tree of depth $d$ is called complete binary tree if all its levels from $0$ to $d-1$ contain maximum possible number of nodes and all the leaf nodes present at level $d$ are placed towards the left side.	
Perfect binary tree	A binary tree is called perfect binary tree if all its leaf nodes are at the lowest level and all the non-leaf nodes contain two child nodes.	
Balanced binary tree	A binary tree is called balanced binary tree if the depths of the subtrees of all its nodes do not differ by more than 1.	

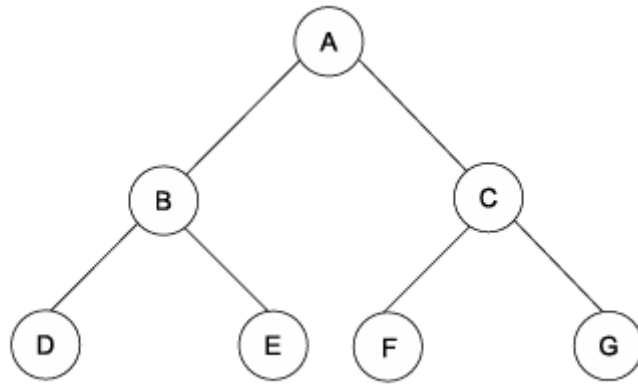
## Binary Tree Representation

The sequential representation of binary trees is done by using arrays while the linked representation is done by using linked lists.

### 1. Array Representation

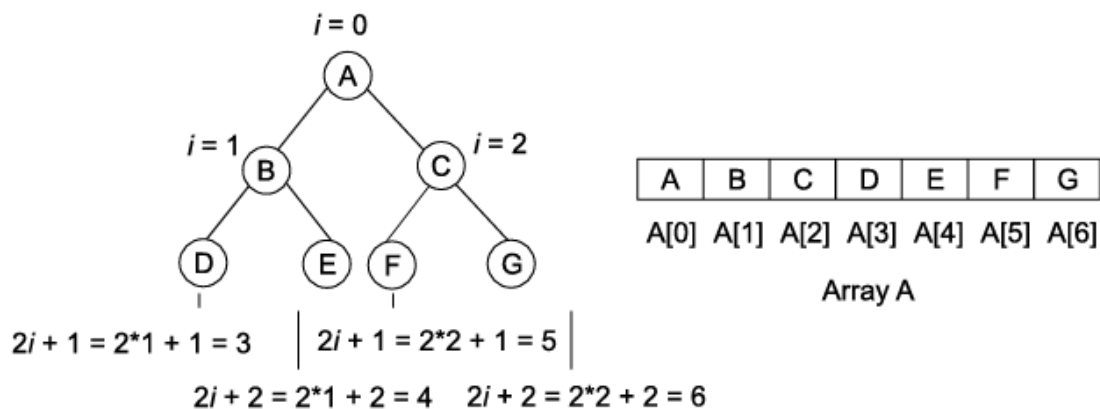
In the array representation of binary trees, one-dimensional array is used for storing the node elements. The following rules are applied while storing the node elements in the array:

1. The root node is stored at the first position in the array while its left and right child nodes are stored at the successive positions.
2. If a node is stored at index location  $i$  then its left child node will be stored at location  $2i+1$  while the right child node will be stored at location  $2i+2$ .  
Let us consider a binary tree T 1. As shown in the figure.



Binary tree T1

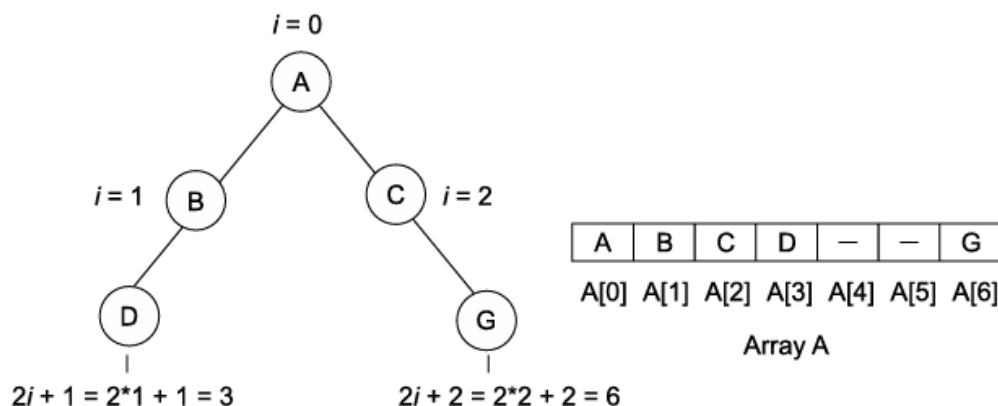
Here, T1 is a binary tree containing seven nodes with A being the root node. B and C are the left and right child nodes of A respectively. Let us apply the rules explained earlier to arrive at the array representation of binary tree T1. Figure 8.4 shows the array representation.



Array representation of binary tree T1

The above figure shows the array index values for each of the tree nodes. Array A is used for storing the node values.

Now, let us modify the binary tree T1 a little by deleting nodes E and F. The revised array representation of T1 is shown in the below figure.



Revised array representation of binary tree T1

As we can see in the above figure, even after removing two elements from the tree, it still requires the same number of memory locations for storing the node elements. This is the main

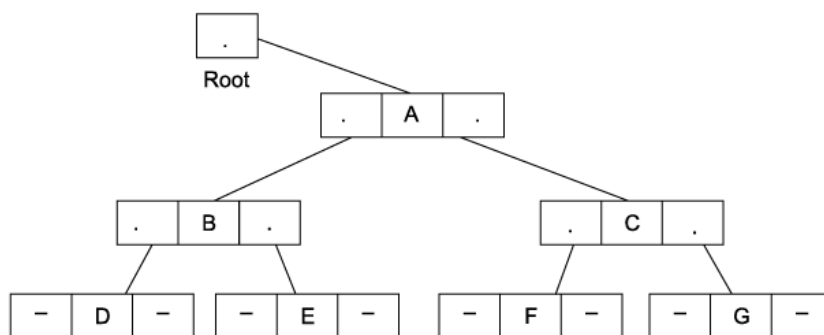
disadvantage of array representation of binary trees. It efficiently utilises the memory space only when the tree is a complete binary tree. Otherwise, there are always some memory locations lying vacant in the array.

## 2. Linked Representation

To avoid the disadvantages associated with array representation, linked representation is used for implementing binary trees. It uses a linked list for storing the node elements. Each tree node is represented with the help of the linked list node comprising of the following fields:

1. **INFO** Stores the value of the tree node.
2. **LEFT** Stores a pointer to the left child.
3. **RIGHT** Stores a pointer to the right child.

In addition, there is a special pointer that points at the root node. Figure 8.6 shows how linked list is used for representing a binary tree in memory.



**Linked representation of binary tree**

The linked representation of binary tree uses dynamic memory allocation technique for adding new nodes to the tree. It reserves only that much amount of memory space as is required for storing its node values. Thus, linked representation is more efficient as compared to array representation.

**program**

## Binary Tree Traversal

Traversal is the process of visiting the various elements of a data structure. Binary tree traversal can be performed using three methods:

1. Preorder
2. Inorder
3. Postorder

**Preorder:** The preorder traversal method performs the following operations:

- (a) Process the root node (N)
- (b) Traverse the left subtree of N (L).
- (c) Traverse the right subtree of N (R).

**Inorder:** The inorder traversal method performs the following operations:

- (a) Traverse the left subtree of N (L).
- (b) Process the root node (N).
- (c) Traverse the right subtree of N (R).

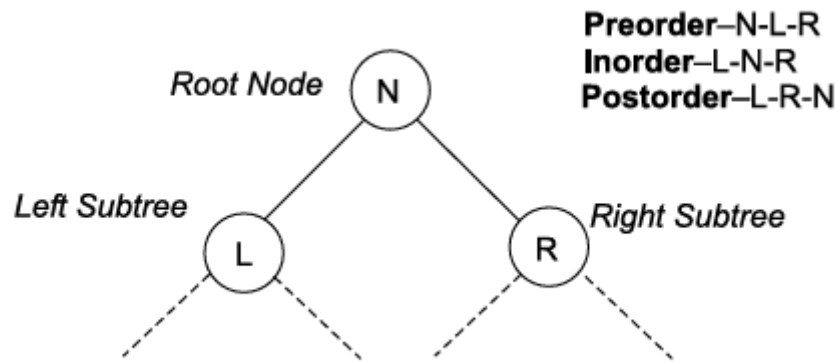
**Postorder:** The postorder traversal method performs the following operations:

- (a) Traverse the left subtree of N (L).
- (b) Traverse the right subtree of N (R).
- (c) Process the root node (N).

Below figure shows an illustration of the different binary tree traversal methods.

**Example**

Consider the following binary tree:

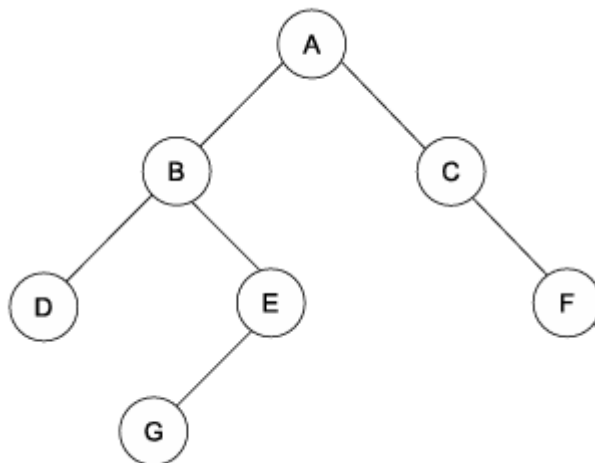


**Binary tree traversal**

For the above binary tree, deduce the following:

- (a) Preorder traversal sequence
- (b) Inorder traversal sequence
- (c) Postorder traversal sequence

**Solution**



- (a) Preorder traversal sequence  
A-B-D-E-G-C-F
- (b) Inorder traversal sequence  
D-B-G-E-A-C-F
- (c) Postorder traversal sequence  
D-G-E-B-F-C-A

**Example:** Write algorithms for the following:

- (a) Preorder traversal
- (b) Inorder traversal
- (c) Postorder traversal

### **Solution**

- (a) Preorder

**preorder**(root)

Step 1: Start

Step 2: Display root

Step 3: Function Call preorder(root->LEFT)

Step 4: Function Call preorder(root->RIGHT)

Step 5: Stop

- (b) Inorder

**inorder**(root)

Step 1: Start

Step 2: Function Call inorder(root->LEFT)

Step 3: Display root

Step 4: Function Call inorder(root->RIGHT)

Step 5: Stop

- (c) Postorder

**postorder**(root)

Step 1: Start

Step 2: Function Call postorder(root->LEFT)

Step 3: Function Call postorder(root->RIGHT)

Step 4: Display root

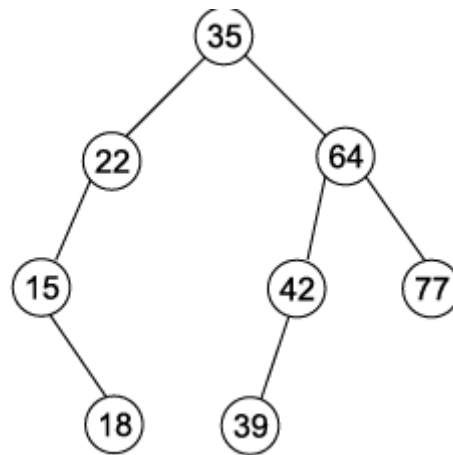
Step 5: Stop

### **Binary Search Tree**

A binary tree is referred as a binary search tree if for any node  $n$  in the tree:

1. the node elements in the left subtree of  $n$  are lesser in value than  $n$ .
2. the node elements in the right subtree of  $n$  are greater than or equal to  $n$ .

Thus, binary search tree arranges its node elements in a sorted manner. As the name suggests, the most important application of a binary search tree is searching. The average running time of searching an element in a binary search tree is  $O(\log n)$ , which is better than other data structures like array and linked lists.



**Binary search tree**

As we can see in the figure, all the nodes in the left subtree are less than the nodes in the right subtree.

The various operations performed on a binary search tree are:

1. Insert
2. Search
3. Delete

**1. Insert** The insert operation involves adding an element into the binary tree. The location of the new element is determined in such a manner that insertion does not disturb the sort order of the tree.

**Example** Write a C function for inserting an element into a binary search tree.

```

node *insert(node *r, int n)
{
    if(r==NULL)
    {
        r=(node*) malloc (sizeof(node));
        r->LEFT = r->RIGHT = NULL;
        r->INFO = n;
    }
    else if(n<r->INFO)
        r->LEFT = insert(r->LEFT, n);
    else if(n>r->INFO)
        r->RIGHT = insert(r->RIGHT, n);
    else if(n==r->INFO)
        printf("\nInsert Operation failed: Duplicate Entry!!");
    return(r);
}
  
```

*A series of recursive function calls are required to identify the precise location where the new node will be inserted.*

### 3. Search

The search operation involves traversing the various nodes of the binary tree to search each iteration, the number of nodes to be searched gets reduced. For example, if the value to be searched is less than the root value then the remainder of the search operation will only be performed in the left subtree while the right subtree will be completely ignored.

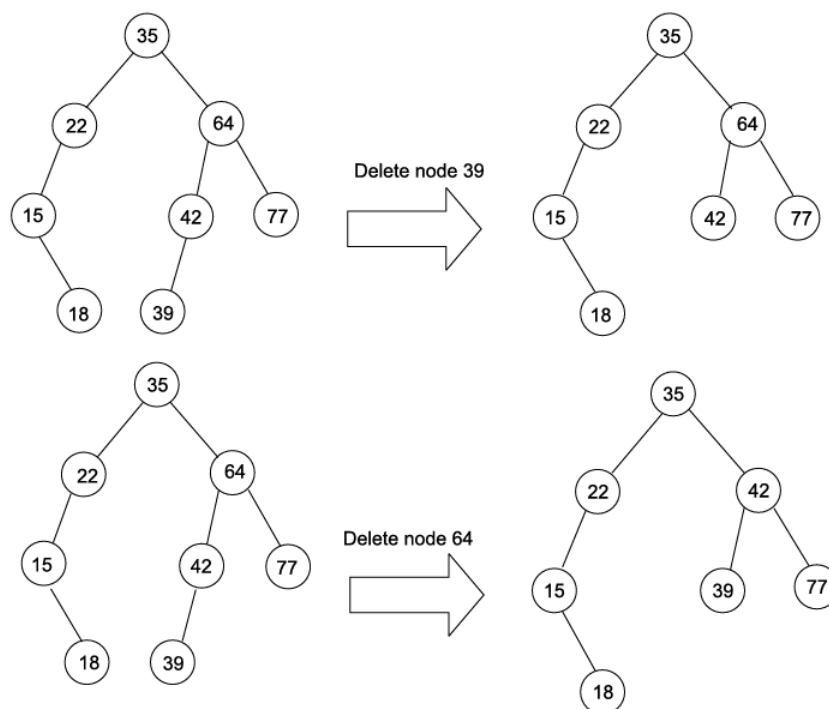
### Example

Write a C function for searching an element in a binary search tree.

```
void search(node *r,int n)
{
    if(r==NULL)
    {
        printf("\n%d not present in the tree!!",n);
        return;
    }
    else if(n==r->INFO)
        printf("\nElement %d is present in the tree!!",n);
    else if(n<r->INFO)
        search(r->LEFT,n);
    else
        search(r->RIGHT,n);
}
```

### 3.Delete

The delete operation involves removing an element from the binary search tree. It is in such a manner that the sort order of the tree is regained. The delete operation is depicted in Fig.



**Deleting an element from binary search tree**

As we can see in Fig, if the node to be deleted is a leaf node, then it is simply deleted without requiring any shuffling of other nodes. However, if the node to be deleted is an internal node then appropriate shuffling is required to ensure that the tree regains its sort order.



**Example :**

Write a C function for deleting an element from a binary search tree.

```
int del(node *r,int n)
{
    node *ptr;
    if (r==NULL)
    {
        return(0);
    }
    else if(n<r->INFO)
        return(del(r->LEFT,n));
    else if(n>r->INFO)
        return(del(r->RIGHT,n));
    else
    {
        if(r->LEFT==NULL)
        {
            ptr=r;
            r=r->RIGHT;
            free(ptr);
            return(1);
        }
        else if(r->RIGHT==NULL)
        {
            ptr=r;
            r=r->LEFT;
            free(ptr);
            return(1);
        }
        else
        {
            ptr=r->LEFT;
            while(ptr->RIGHT!=NULL)
                ptr=ptr->RIGHT;
            r->INFO=ptr->INFO;
            return(del(r->LEFT,ptr->INFO));
        }
    }
}
```

*A return value of 0 signifies unsuccessful search while a return value of 1 signifies successful search.*

**4.Implementation** The implementation of a binary search tree requires implementing the insert, search, and delete operations.

## Tree Variants

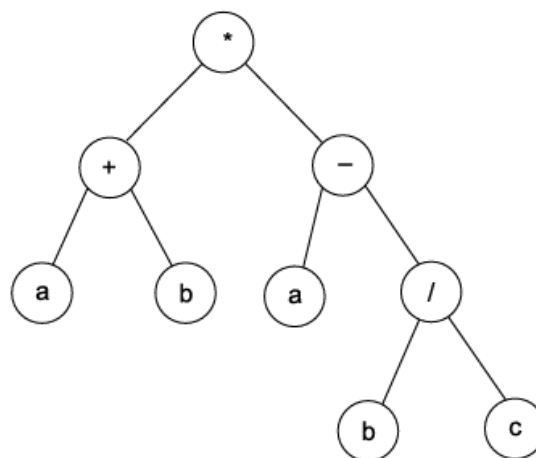
Based on the concept of trees, binary trees and binary search trees various tree variants have been deduced. Each of these variants possesses distinct characteristics and serves specific purposes. For example, balanced binary trees balance their nodes in such a way that the height of the tree is always kept to a minimum, thus ensuring better average case performance at the time of searching.

### i. Expression Trees

Expression tree is nothing but a binary tree containing mathematical expression. The internal nodes of the tree are used to store operators while the leaf or terminal nodes are used to store operands. Various compilers and parsers use expression trees for evaluating arithmetic and logical expressions.

Consider the following expression:  $(a+b)*(a-b/c)$

The expression tree for the above expression is shown in Fig.



**Expression tree**

As shown in the above tree, the internal nodes store the operators while the leaf nodes store the operands. While constructing a binary tree from a given expression, the following precedence rules are followed:

1. Parentheses are evaluated first
2. The exponential expressions are evaluated next.
3. Then, division and multiplication operations are evaluated.
4. Finally, addition and subtraction operations are evaluated.

Representing an expression using a binary tree has another key advantage. By applying the various traversal methods we can deduce the other representations of an expression. For example, the preorder traversal of an expression tree derives its prefix notation.

Table shows the various expression notations deduced after traversing the expression tree.

Expression Notation	Traversal Method	Example (Refer to Fig. 8.10)
Prefix	Preorder	*+ab-a/bc
Infix	Inorder	a+b*a-a/c
Postfix	Postorder	ab+abc/-*

### ii. Threaded Binary Trees

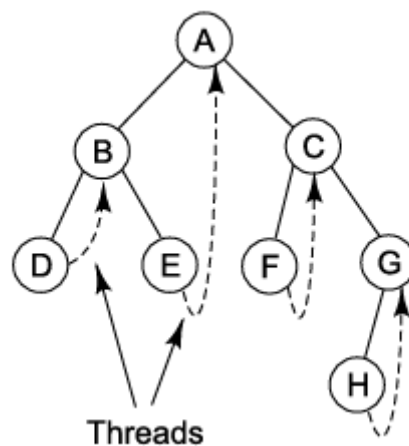
Let us recall the structure declaration of a tree node described during the linked implementation of a binary tree:

```
struct bin_tree
{
int INFO;
struct node *LEFT, *RIGHT;
};
```

As we can see in the above declaration, each node in a binary tree has two pointer nodes associated with it, i.e., LEFT and RIGHT. Now, in case of leaf nodes, these pointers contain NULL values. Considering the number of leaf nodes that are there in a typical binary tree, this leads to a lot of memory space getting wasted. Threaded binary trees offer an innovative alternate to avoid this memory wastage.

In a threaded binary tree, all nodes that do not have a right child contain a pointer or a thread to its inorder successor. The address of the inorder successor node is stored in the RIGHT pointer. But, how do we distinguish between a normal pointer and a thread pointer? This is done with the help of a Boolean variable, as shown in the below node declaration of a threaded binary tree:

```
struct t_tree
{
int INFO;
struct node *LEFT, *RIGHT;
boolean LThread, RThread;
};
```



**Threaded binary tree**

As shown in the figure, nodes D, E, F and H contain threads to point at their inorder successors. Now, what is the advantage of a threaded binary tree representation? Try to recall the algorithm for inorder traversal of a binary tree. The algorithm uses recursive function calls to determine the inorder traversal path. The execution of recursive function calls requires the use of stack and consumes both memory as well as time. The threaded tree traversal allows us to determine the inorder sequence using an iterative approach instead of a recursive approach.

### Example

Write the algorithm for traversal of a threaded binary tree to generate the inorder sequence.

### Solution

#### **inorder (node)**

Step 1: Start

Step 2: Set current = leftmost(node)

//current refers to the current node

//leftmost function returns the left most node value in a subtree

Step 3: while current != NULL repeat Steps 4-7

Step 4: Display current

Step 5: If current->RThread != NULL goto Step 6 else goto Step 7

Step 6: Set current = current->RIGHT

Step 7: Set current = leftmost(current->RIGHT)

Step 8: Stop

#### **leftmost (node)**

Step 1: Start

Step 2: Set ptr = node

Step 3: if ptr = NULL goto Step 4 else goto Step 5

Step 4: Return NULL and goto Step 8

Step 5: while ptr->LEFT != NULL repeat Step 6

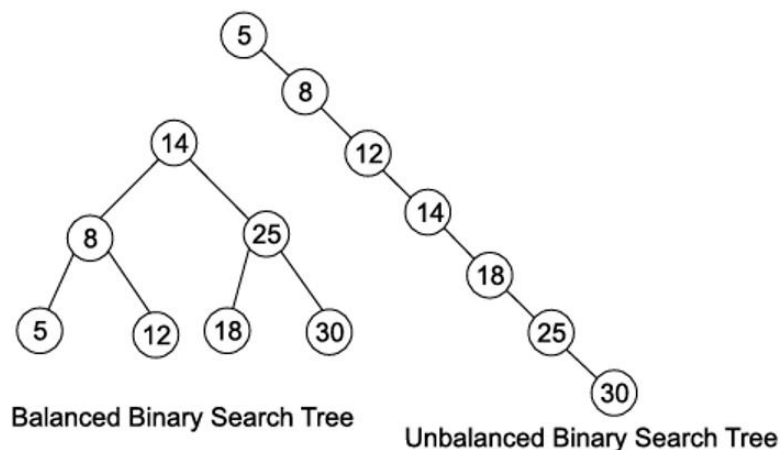
Step 6: Set ptr = ptr->LEFT

Step 7: Return ptr

Step 8: Stop

If we apply the above algorithm on the threaded binary tree shown in Fig. Then we will obtain the following inorder sequence: D-B-E-A-F-C-H-G

### iii. Balanced Trees



we saw how nodes are added to a binary search tree. With each addition of a node in a tree, there is a possibility that the height of the tree may also get changed. The height of a tree has a direct affect on its efficiency to perform the search operation.

For instance, consider the binary search trees shown in Fig.

Both the binary search trees shown in the above figure contain the same nodes however the height of the first tree is 2 while that of the second tree is 6. To search element 30 in the above trees, we need to dig a lot deeper in the second tree as compared to the first tree. Thus, while implementing binary trees, it is important to keep the height of the tree in check.

There are various binary search trees that keep the tree balanced whenever a new node is added by shuffling the tree nodes appropriately.

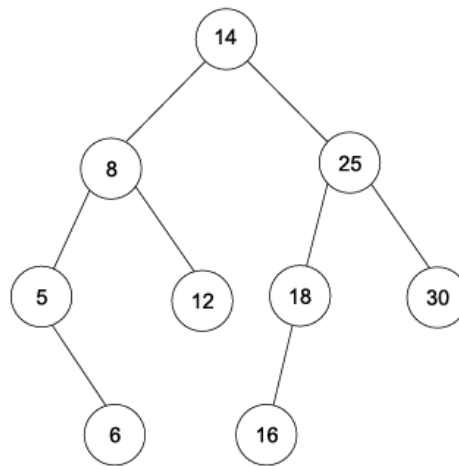
These are:

1. AVL tree
2. Red-Black tree

**1. AVL tree** AVL tree, also called height-balanced tree was defined by mathematicians Adelson, Velskii and Landis in the year 1962. The main characteristic of an AVL tree is that for all its nodes, the height of the left subtree and the height of the right subtree never differ by more than.

At any point of time, an AVL tree node is in any one of the following states:

- (a) *Balanced* The height of left subtree is equal to the height of right subtree.
- (b) *Left heavy* The height of left subtree is one more than the height of right subtree.
- (c) *Right heavy* The height of right subtree is one more than the height of left subtree.



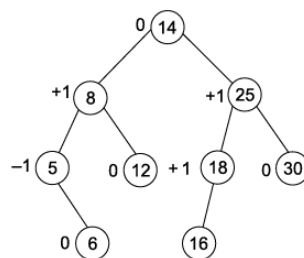
**AVL Tree**

The height of left and right subtrees of each node differs by not more than 1.

Now, how is an AVL tree created and maintained? This is done by associating a balance factor (BF) with each node that keeps a track of the height balance for that particular node. BF for a node is calculated by using the following formula:

$$\text{BF} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

Let us apply the above formula to calculate the balance factor for each node of the AVL tree shown in Fig.



**AVL Tree with Balance factor**

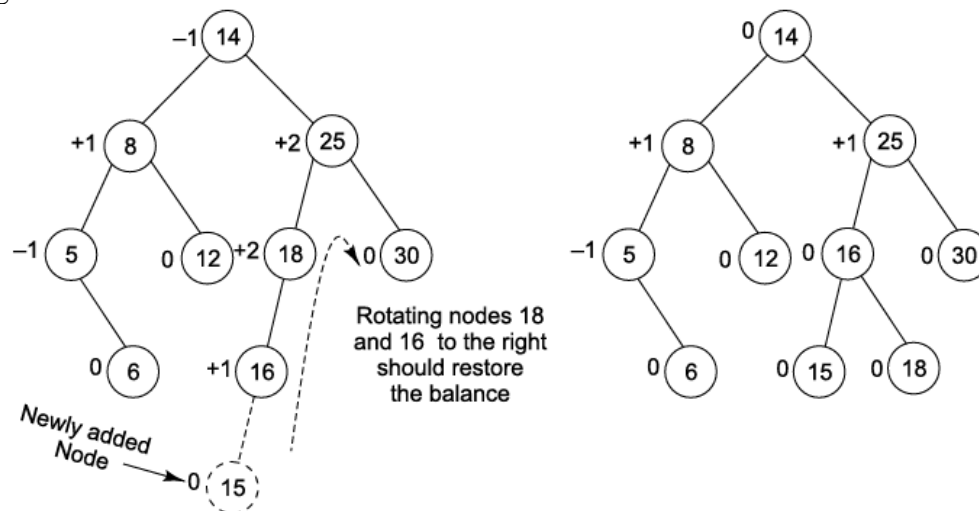
The balance factors of all the nodes are not more than 1, which is the key characteristic of an AVL tree.

The structure declaration of an AVL tree node contains an additional field for storing the balance factor, as shown below:

```
struct avl_node
{
    int INFO;
    struct node *LEFT, *RIGHT;
    int BF;
};
```

Whenever a new node is inserted in an AVL tree, a slight disbalance is created at the point of insertion which reflects in the balance factors of the nodes in its preceding path till the root node. To restore the balance of the tree, left and right rotations are carried out to move the nodes towards the right or left. This is repeated until the balance factors of all the nodes are reduced below 1.

Below figure shows the insertion of node value 15 into the AVL tree.



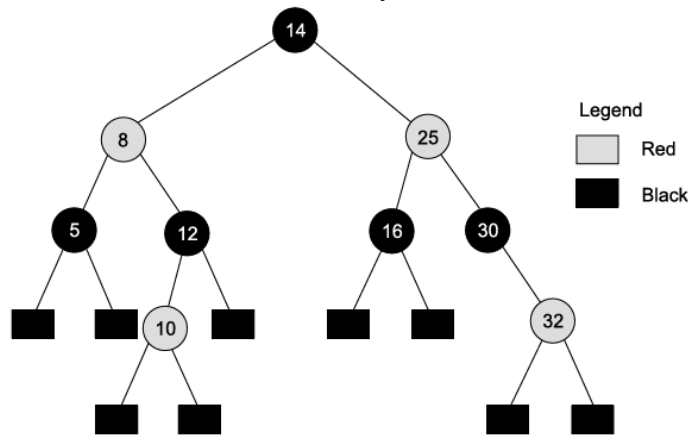
**Inserting an element in an AVL tree**

The delete operation follows a similar approach. A left or right rotation may need to be carried out if a node is deleted from an AVL tree.

**2. Red-Black tree** Red-Black tree is a self-balancing binary search tree that has an average running time of  $O(\log n)$  for insert, delete and search operations. As the name suggests, the red-black tree associates a color attribute with each node, which can possess only two values, red or black. That means each node in a red-black tree is either red or black colored. Apart from possessing the properties of a typical binary search tree, a red-black tree possesses the following properties:

- Each node is either red or black in color.
- The root node is black colored.
- The leaf nodes are black colored. It includes the NULL children.
- The child nodes of all red-colored nodes are black.
- Each path from a given node to any of its leaf nodes contains equal number of black nodes. The number of such black nodes is also referred as black-height (bh) of the node.

The above properties ensure that the length of the longest path from the root node to a leaf node is less than roughly twice of the shortest path. This ensures that the balance of the tree is always kept under check. The key advantage of a red-black tree is that its worst case running time is better than most of the other binary search trees.



Red-Black tree

The insert and delete operations on a red-black tree require small number of rotations as well as change of colors of some of the nodes so that the tree complies with all the properties of a red-black tree. However, the average running time of these operations is  $O(\log n)$ .

#### iv. Splay Trees

The concept of splay trees is based on the assumption that when a particular element is accessed from a binary search tree then there are high chances that the same element would be accessed again in future. Now, if the element is placed deep in the tree then all such repetitive accesses would be inefficient. To make the repetitive accesses of a node efficient, splay tree shifts the accessed node towards the root two levels at a time. This shifting is done through splay rotations. Table 8.4 shows the various types of splay rotations along with an illustration.

Splay Rotation	Occurrence	Illustration
Zig	When root node P is the parent of the node N being accessed.	
Zigzag	When node N is the right child of parent P, which itself is the left child of grandparent G. Or, when node N is the left child of parent P, which itself is the right child of grandparent G.	
Zigzig	When both node N and parent P are left or right child of grandparent G.	

Splay rotations ensure that all future accesses of a node are efficient as compared to its first time access.

Let us now discuss what happens when typical tree-related operations are performed on a splay tree:

**1. Insert** New element is inserted at the root.

**2. Search** There are two possibilities:

(a) Successful search The searched node is moved to the root position.

(b) Unsuccessful search The last node accessed during the unsuccessful search operation is moved to the root position.

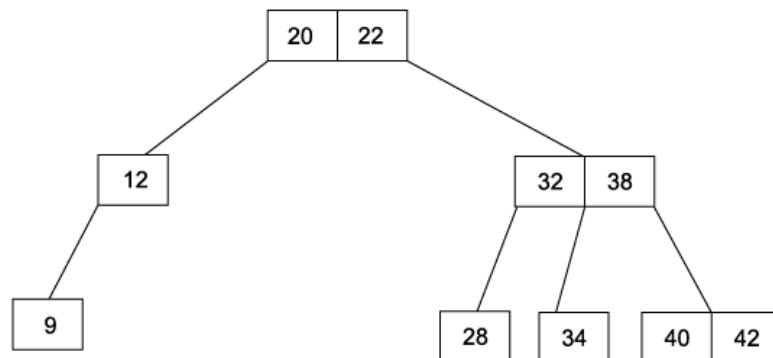
**3. Delete** the largest node in the left subtree is moved to the root position.

## v. m-way Trees

Binary search trees are more suitable for smaller data sets where the data is static. However, for large data sets which require dynamic access (example file storage); binary search trees are not exactly suitable. For such cases, the nodes of the tree are required to store large amounts of data. This is achieved with the help of m-way trees.

m-way search trees are an extension of binary search trees having the following properties:

1. Each node of the tree stores 1 to m-1 number of keys.
2. The keys are stored in a sorted manner inside the node.
3. A node containing k values can have a maximum of k+1 subtrees.
4. The subtree pointed by pointer T<sub>i</sub> has values less than the key value of k
5. All the subtrees are m-way trees.



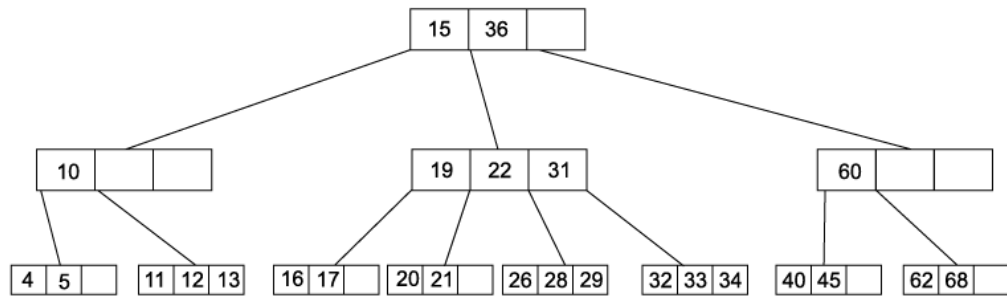
**Sample m-way tree**

(a) B tree To ensure efficiency while searching an m-way tree it is important to control its height.

This is achieved with the help of B tree. A B tree is nothing but a height balanced m-way search tree. A B tree of order m has the following properties:

- i. Root node is either a leaf node or it contains child nodes ranging from 2 to m.
- ii. All internal nodes contain a maximum of m-1 keys.
- iii. Number of children of internal nodes ranges from m/2 to m.
- iv. Number of keys stored in the leaf nodes ranges from (m-1)/2 to m-1. All the keys are stored in a sorted manner.
- v. All leaf nodes are at the same depth.





**Sample B tree**

An element is inserted in a B tree by first identifying the location where the new node should be inserted. If the existing node is not full, the new element is inserted within the existing node and an appropriate pointer is created linking it with the parent node. However, if the existing node is full then it is split into three parts. The middle part is accommodated with the parent node while the new element is inserted in one of the child nodes.

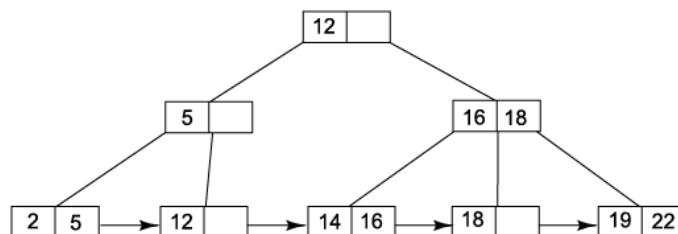
Similarly, deletion of an element from a B tree is done by first removing the element from a node and then carrying out appropriate redistributions to ensure that the tree stays true to its properties.

(b) B+ tree B + tree is a variant of B tree that is mainly used for implementing index sequential access of records. The main difference between B + in the leaf nodes. The internal nodes of a B + tree and B tree is that in B + tree data records are only stored tree are only used for storing the key values. The key values help in performing the search operation. If the target element is less than a key value then the search proceeds towards its left pointer. Similarly, if the target element is greater than a key value then the search proceeds towards its right pointer.

A B + tree of order m has the following properties:

- i. The internal nodes contain up to  $m-1$  keys.
- ii. The number of children of internal nodes lies between  $m/2$  and  $m$ .
- iii. The subtree between keys  $k_1$  and  $k_2$  contains value  $v$  such that  $k_1 \leq v < k_2$ .
- iv. All leaf nodes are at the same level.
- v. All the leaf nodes are sequentially connected through a linked list.

B+ tree is typically used for implementing index sequential file organization in a database. The internal nodes are used for representing index values through which data records in the sequence set are accessed.

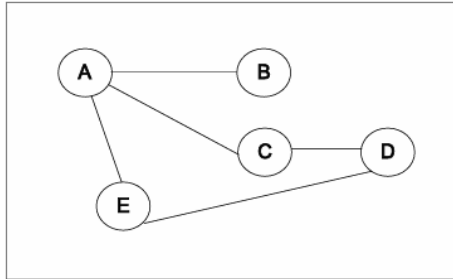


**Sample B + tree**

## MODULE-5 GRAPHS, SORTING AND SEARCHING

### Graphs

A graph is a set of nodes (also called as vertices) and a set of arcs (also called as edges).



**Graph**

### Introduction

Graph is similar to the mathematical graph structure, which comprises of a set of vertices connected with each other through edges. Some of the typical operations performed on a graph data structure include finding possible paths between two nodes and finding the shortest possible path.

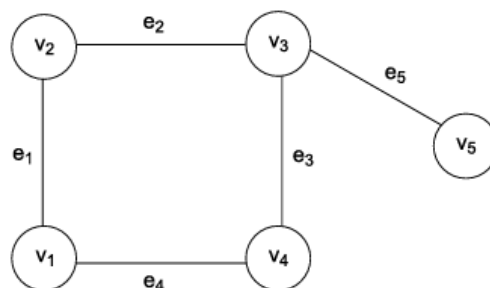
Graph data structure finds its application in varied domains, such as computer network analysis, travel application, chip designing, gaming and so on.

### Basic Concept

A graph  $G$  consists of the following elements:

- A set of vertices or nodes where  $V = \{v_1, v_2, v_3, \dots, v_n\}$
- A set of  $E$  edges also called arcs where  $E = \{e_1, e_2, e_3, \dots, e_n\}$

Here,  $G = (V, E)$

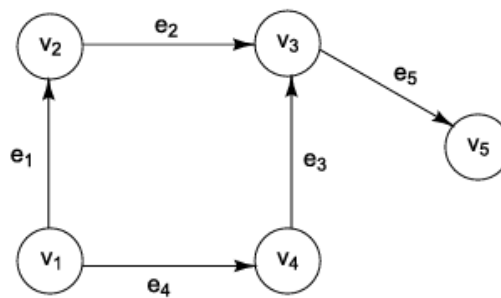


**Graph G**

In above Fig,  $e_1$  is an edge between  $v_1$  and  $v_2$  vertices while  $e_2$  is an edge between  $v_2$  and  $v_3$  vertices. Thus, we can generically represent an edge  $e=(u, v)$  where  $e$  connects both  $u$  and  $v$  vertices.

Now,  $e=(u,v)$  means the same things as  $e=(v,u)$ . This means that the ordering of the vertices has no significance here. Thus, we can call the graph  $G$  as undirected graph.

If we replace each edge of the Graph  $G$  with arrows, then it will become a directed graph or diagraph, as shown in Fig.



**Directed Graph**

In Fig graph, the set of vertices and edges are:

$V(G)=\{v_1, v_2, v_3, v_4, v_5\}$

$\{(v_1, v_2), (v_2, v_3), (v_1, v_4), (v_4, v_3), (v_3, v_5)\}$

## Graph Terminology

There are a number of key terms associated with the concept of graphs. Table explains some of these important key terms.

Key Terms	Description
Adjacent node	If $e(u, v)$ represents an edge between $u$ and $v$ vertices then both $u$ and $v$ are called adjacent to each other. That means, $u$ is adjacent to $v$ and $v$ is adjacent to $u$ .
Predecessor node	If $e(u, v)$ represents a directed edge from $u$ to $v$ then $u$ is a predecessor node of $v$ .
Successor node	If $e(u, v)$ represents a directed edge from $u$ to $v$ then $v$ is a successor node of $u$ .
Degree	Degree of a vertex is the number of edges connected to a vertex. For example, in the graph shown in Fig. 9.1, the degree of vertex $v_3$ is 3.
Indegree	In a directed graph, indegree of a vertex is the number of edges ending at the vertex.
Outdegree	In a directed graph, outdegree of a vertex is the number of edges beginning at the vertex.
Path	A path is a sequence of vertices each adjacent to the next. For example, in the graph shown in Fig. 9.2, the path between the vertices $v_1$ and $v_5$ is $v_1-v_2-v_3-v_5$ .
Cycle	It is a path that starts and ends at the same vertex.
Loop	It is an edge whose endpoints are same that is, $e = (u, u)$ .
Weight	It is a non-negative number assigned to an edge. It is also called length.
Order	Order of a graph is the number of the vertices contained in the graph.
Labeled Graph	It is a graph that has labeled edges.
Weighted Graph	It is a graph that has weights assigned to each of its edges.
Connected Graph	It is an undirected graph in which there is a path between each pair of nodes.
Strongly Connected Graph	It is a directed graph in which there is a route between each pair of nodes.
Complete Graph	It is an undirected graph in which there is a direct edge between each pair of nodes.
Tree	It is a connected graph with no cycles.

## Graph Implementation

Graphs are nothing but a collection of nodes and edges. Thus, while representing graphs in memory the only focus is on capturing details related to the different vertices and edges. Graphs can be implemented using the following methods:

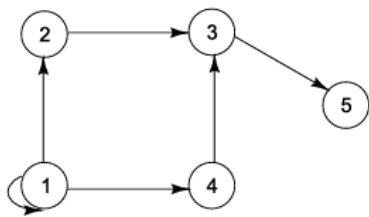
1. Adjacency matrix
2. Path matrix
3. Adjacency list

### 1. Implementing Graphs Using Adjacency Matrix

Consider a graph  $G=(V,E)$  having  $N$  nodes. The adjacency matrix of graph  $G$  is defined as an  $N \times N$  matrix  $A$ , where:

- i.  $A_{ij}=1$ , if there is an edge from vertex  $v_i$  to  $v_j$ .
- ii. And  $A_{ij}=0$ , if there is no edge from vertex  $v_i$  to  $v_j$ .

**Example :** Consider the graph shown in Figure.



The adjacency matrix of the above graph will be Here, 0s represent that there is no directed edge between the corresponding vertices while 1s represent the presence of a directed edge.

$A_{i,j}$	1	2	3	4	5
1	1	1	0	1	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	1	0	0
5	0	0	0	0	0

**Write a program in C to represent a graph using adjacency matrix.**

```
#include<stdio.h>
#include <conio.h>
void main()
{
    int A[5][5];
    int i,j;
    clrscr();
    for(i=0;i<5;i++)
    for(j=0;j<5;j++)
    A[i][j]=0; /*Initializing the array A*/
    /*Creating adjacency matrix*/
    A[0][0]=1;
    A[0][1]=1;
    A[0][3]=1;
    A[1][2]=1;
    A[2][4]=1;
    A[3][2]=1;
```

```

/*Printing Adjacency Matrix*/
printf("Adjacency Matrix:");
for(i=0; i<5;i++)
{
    printf("\n");
    for(j=0;j<5;j++)
    printf("%d ",A[i][j]);
}
getch();
}

```

## Output

```

Adjacency Matrix:
1 1 0 1 0
0 0 1 0 0
0 0 0 0 1
0 0 1 0 0
0 0 0 0 0

```

## 2. Implementing Graphs Using Path Matrix

Consider a diagraph  $G=(V,E)$  having  $N$  nodes .The path matrix of graph  $G$  is defined as an  $N \times N$  matrix  $P$ , where

1.  $P_{i,j} = 1$ , if there is a path fro vertex  $v_i$  to  $v_j$ , and
2.  $P_{i,j} = 0$ , if there is no path fro vertex  $v_i$  to  $v_j$ .

Now, the path matrix  $P$  can be deduced using the adjacency matrix of  $G$ , as depicted below:

$$P_N = A + A^2 + A^3 + \dots + A^N$$

Here,  $A^2$  (A square) is the square of the adjacency matrix  $A$ ,  $A^3$  (A cube) is the cube of  $A$ , and so on. All the non-zero entries resulting from the addition operation above are replaced by 1 to arrive at the path matrix.

This method of deriving the path matrix by computing powers of adjacency matrix is not very efficient, as it requires performing a number of matrix multiplication operations. Warshall has suggested a more simplified method of deriving the path matrix from the adjacency matrix. Warshall's method determines the presence of a path between  $v_i$  and  $v_j$  by

1. identifying a direct path from  $v_i$  to  $v_j$ , and
2. identifying an indirect path from  $v_i$ , and and  $v_j$  that is, a path from  $v_i$  to  $v_k$  and  $v_k$  to  $v_j$ .

That is,  $P[i,j] = P[i,j] \text{ OR } (P[i,k] \text{ AND } P[k,j])$

Here, OR represents the logical OR operation and AND represents the logical AND operation.

**Write the Warshall's algorithm for deriving the path matrix of a digraph G.**

**path\_matrix(Adjacency Matrix A[], N)**

Step 1: Start

Step 2: Set  $P[] = A[]$

Step 3: Set  $i = j = k = 1$

Step 4: Repeat Steps 5-10 while  $k \leq N$

Step 5: Repeat Steps 6-9 while  $i \leq N$

Step 6: Repeat Steps 7-8 while  $j \leq N$

Step 7:  $P[i,j] = P[i,j] \text{ OR } (P[i,k] \text{ AND } P[k,j])$

Step 8:  $j = j + 1$

Step 9:  $i = i + 1$

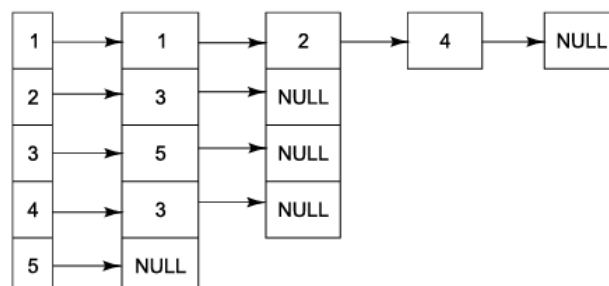
Step 10:  $k = k + 1$

Step 11: Display path matrix  $P[]$

Step 12: Stop

### 3. Implementing Graphs Using Adjacency List

Adjacency list is a linked representation of a graph. It consists of a list of graph nodes with each node itself consisting of a linked list of its neighboring nodes. Figure 9.4 shows the adjacency list of the directed graph shown in Fig.



**Adjacency list**

The adjacency list shown above contains five nodes with each node pointing towards its successor nodes.

**Write a program in C to represent a graph using adjacency list.**

```
#include
#include
#include

struct vertex
{
    struct vertex *edge[10];
    int id;
}node[10];

void display(int);

void main()
{
    int i,j,N;
    char ch;
    clrscr();
    i=j=N=0;
    printf("Enter number of graph vertices: ");
    scanf("%d",&N);
    for(i=0;i<N;i++)
    {
        node[i].id=i;
        fflush(stdin);
        for(j=0;j<N;j++)
        {
            fflush(stdin);
            printf("Edge from %d to %d? (y/n): ",i+1,j+1);
            scanf("%c",&ch);
            if(ch=='y') node[i].edge[j]=&node[j];
            else
                node[i].edge[j]=NULL;
        }
    }
}
```



```

display(N);
getch();
}

void display(int num)
{
    int i,j;
    printf("\n");
    for(i=0;i<num;i++)
    {
        printf("Edges of node[%d] are: ",i+1);
        for(j=0;j<num;j++)
        {
            if(node[i].edge[j]==NULL)
                continue;
            printf("(%d-%d) ",i+1,node[i].edge[j]->id+1);
        }
        printf("\n");
    }
}

```

## Output

```

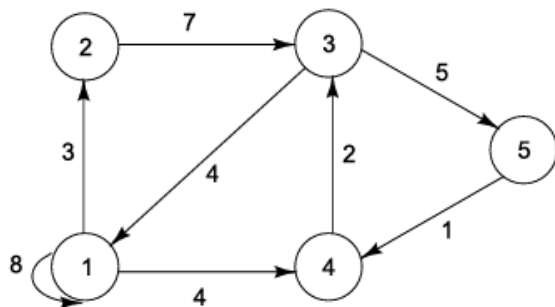
Enter number of graph vertices: 3
Edge from 1 to 1? (y/n): n
Edge from 1 to 2? (y/n): y
Edge from 1 to 3? (y/n): y
Edge from 2 to 1? (y/n): y
Edge from 2 to 2? (y/n): n
Edge from 2 to 3? (y/n): y
Edge from 3 to 1? (y/n): n
Edge from 3 to 2? (y/n): y
Edge from 3 to 3? (y/n): y
Edges of node[1] are: (1-2) (1-3)
Edges of node[2] are: (2-1) (2-3)
Edges of node[3] are: (3-2) (3-3)

```

## Shortest Path Algorithm

One of the most common problems associated with graphs is to find the shortest path from one node to the other. It finds its relevance in a number of real-life applications. For example, consider a scenario where a freight carrier departing from Delhi is required to drop consignments at Lucknow, Jaipur, Ahmedabad, Pune, and Hyderabad airports. In this situation, the route taken by the carrier is determined by assessing the distance between each of these cities. The final route undertaken should be the shortest path that covers all these cities. We can relate this scenario with a graph data structure where each city represents a graph node while the distance between the cities represents the weight of the edges.

Consider the weighted digraph shown in Figure.



The weight matrix of the above graph will be

$W_{i,j}$	1	2	3	4	5
1	8	3	0	4	0
2	0	0	7	0	0
3	4	0	0	0	5
4	0	0	2	0	0
5	0	0	0	1	0

Here,  $W_{i,j}$  represents the weight of the edge from node  $v_i$  to  $v_j$  + no direct edge between the corresponding nodes. Now, a modification of the Warshall's algorithm can be applied to the weight matrix to derive the shortest path matrix SP that represents the weight of the shortest possible path between any two nodes of a graph.

It begins with replacing all 0's in the weight matrix with  $\infty$ , as shown below.

$SP_{i,j}$	1	2	3	4	5
1	8	3	8	4	8
2	8	8	7	8	8
3	4	8	8	8	5
4	8	8	2	8	8
5	8	8	8	1	8

Now, the following relation is applied to arrive at the shortest path matrix:

$$SP_{i,j} = \text{Minimum of } (SP_{i,j}, SP_{i,k} + SP_{k,j})$$

The shortest path matrix obtained after applying the above relation for each graph node is

$SP_{i,j}$	1	2	3	4	5
1	8	3	6	4	11
2	11	14	7	13	12
3	4	7	8	6	5
4	6	9	2	8	7
5	7	10	3	1	8

**Write the modified Warshall's algorithm for deriving the shortest path matrix of a digraph G.**

**shortest\_path\_matrix(Path Matrix P[], N)**

Step 1: Start

Step 2: Set  $i = j = 1$

Step 3: Repeat Steps 4-9 while  $i \leq N$

Step 4: Repeat Steps 5-8 while  $j \leq N$

Step 5: if  $P[i,j] = 0$  goto Step 6 else goto Step 7

Step 6: Set  $SP[i,j] = 8$

Step 7: Set  $SP[i,j] = P[i,j]$

Step 8:  $j = j + 1$

Step 9:  $i = i + 1$

Step 10: Set  $i = j = k = 1$

Step 11: Repeat Steps 12-17 while  $k \leq N$

Step 12: Repeat Steps 13-16 while  $i \leq N$

Step 13: Repeat Steps 14-15 while  $j \leq N$

Step 14:  $SP[i,j] = \text{MINIMUM}(SP[i,j], SP[i,k] + SP[k,j])$

Step 15:  $j = j + 1$

Step 16:  $i = i + 1$

Step 17:  $k = k + 1$

Step 18: Display shortest path matrix SP[]

Step 19: Stop

**Write a program in C to deduce the shortest path matrix of a weighted digraph G uses modified Warshall's algorithm to derive the shortest path matrix of the weighted digraph.**

```

#include <stdio.h>

#include<conio.h>

int MIN(int, int); /*Function prototype for computing the minimum among two integers*/

void main()
{
int P[5][5], SP[5][5];
int i,j,k;
clrscr();
for(i=0;i<5;i++)
for(j=0;j<5;j++)
P[i][j]=0;
P[0][0]=8;
P[0][1]=3;
P[0][3]=4;
P[1][2]=7;
P[2][0]=4;
P[2][4]=5;
P[3][2]=2;
P[4][3]=1;
printf("Path Matrix: \n");
for(i=0;i<5;i++)
{
printf("\n");
for(j=0; j<5;j++)
printf("%d\t",P[i][j]);
}
for(i=0;i<5;i++)
for(j=0;j<5;j++)
if(P[i][j]==0)
SP[i][j]=999;

```

```

else
SP[i][j]=P[i][j];
for(k=0;k<5;j++)
for(i=0;i<5;i++)
for(j=0;j<5;j++)
SP[i][j]=MIN(SP[i][j],SP[i][k]+SP[k][j]);
printf("\n\nShortest Path Matrix: \n");
for(i=0; i<5;i++)
{
printf("\n");
for(j=0;j<5;j++)
printf("%d\t",SP[i][j]);
}
getch();
}
int MIN(int x, int y)
{
if(x<=y)
return(x);
else
return(y);
}

```

### Output

Path Matrix:

```

8 3 0 4 0
0 0 7 0 0
4 0 0 0 5
0 0 2 0 0
0 0 0 1 0

```

Shortest Path Matrix:

```

8 3 6 4 11
11 14 7 13 12
4 7 8 6 5
6 9 2 8 7

```

## Graph Traversal

One of the common tasks associated with graphs is to traverse or visit the graph nodes and edges in a systematic manner. There are two methods of traversing a graph:

1. Breadth First Search(BFS)
2. Depth First Search(DFS)

Both these methods consider the graph nodes to be in one of the following states at any given point of time:

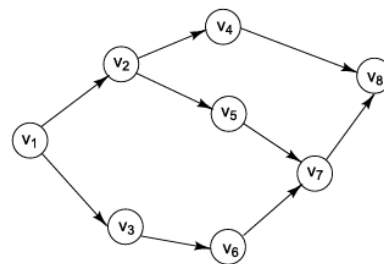
1. Ready state
2. Waiting state
3. Processed state

The state of a node keeps on changing as the graph traversal progresses. Once the state of a node becomes processed, it is considered as traversed or visited.

### 1. Breadth First Search

The BFS method begins with analyzing the starting node and then progresses by analysing its adjacent or neighbouring nodes. Once all the neighbouring nodes of the starting node are analyzed, the algorithm starts analyzing the neighboring nodes of each of the analyzed neighboring nodes. This method of graph traversal requires frequent backtracking to the already analyzed nodes. As a result, a data structure is required for storing information related to the neighboring nodes. The BFS method uses the queue data structure for storing the nodes data.

Consider the graph shown in Figure.



**Graph traversal**

The BFS traversal sequence for the above graph will be: v1, v2, v3, v4, v5, v6, v7, v8 .Another BFS traversal sequence can be v1, v3, v2, v6, v5, v4, v7, v8.

**Write an algorithm for the BFS graph traversal method.**

**BFS(adj[], status[], queue[], N)**

Step 1: Start

Step 2: Set status[] = 1

Step 3: Push(queue, v1)

Step 4: Set status[v1]=2

Step 5: Repeat Step 6–11 while queue[] is not empty

Step 6: V = Pop(queue)

Step 7: status[V]=3

Step 8: Repeat Step 9-11 while adj(V) is not empty

Step 9: If adj(V) = 1 goto step 10 else goto step 8

Step 10: Push(queue, adj(V))

Step 11: Set adj[v]=2

Step 12: Stop

## 2. Depth First Search

Unlike the BFS traversal method, which visits the graph nodes level by level, the DFS method visits the graph nodes along the different paths. It begins analyzing the nodes from the start to the end node and then proceeds along the next path from the start node. This process is repeated until all the graph nodes are visited.

The DFS method also requires frequent backtracking to the already analyzed nodes. It uses the stack data structure for storing information related to the previous nodes.

Let us again consider the graph shown in Fig. 9.6. The DFS traversal sequence for this graph will be:

v 1 , v 2 , v 4 , v 8 , v 5 , v 7 , v 3 , v 6 . Another DFS traversal sequence can be: v1,v3,v6,v7,v8,v2,v5,v4 .

**Write an algorithm for the DFS graph traversal method.**

**DFS(adj[], status[], stack[], N)**

Step 1: Start

Step 2: Set status[] = 1

Step 3: Push(stack, v1)

Step 4: Set status[v1]=2

Step 5: Repeat Step 6–11 while stack[] is not empty

Step 6: V = Pop(stack)

Step 7: status[V]=3

Step 8: Repeat Step 9-11 while adj(V) is not empty

Step 9: If  $\text{adj}(V) = 1$  goto step 10 else goto step 8

Step 10: Push(stack,  $\text{adj}(V)$ )

Step 11: Set  $\text{adj}[v]=2$

Step 12: Stop



# Sorting and Searching

## Introduction

Sorting and searching are two of the most common operations performed by computers.

The sorting operation arranges the numerical and alphabetical data present in a list, in a specific order or sequence. Searching, on the other hand, locates a specific element across a given list of elements. At times, a list may require sorting before the search operation can be performed on it.

There are a number of sorting techniques that can be employed to sort a given list of data elements. The suitability of a specific technique in a specific situation depends on a number of factors, such as

- Size of the data structure
- Algorithm efficiency, and
- Programmer's knowledge of the technique.

While all the sorting methods produce the same result, that is a list of sorted elements, it is one or more of the above factors that play an important role in choosing a specific sorting technique in a given situation.

## Sorting Techniques

Consider a list L containing n elements, as shown below.

$L_1, L_2, L_3, \dots, L_n$

Now, there are  $n!$  ways in which the elements can be arranged within the list. We can apply a sorting technique to the list L to arrange the elements in either ascending or descending order.

If we sort the list in ascending order, then

$$L_1 \leq L_2 \leq L_3 \dots \leq L_n$$

Alternatively, if we arrange the list in descending order, then

$$L_1 \geq L_2 \geq L_3 \dots \geq L_n$$

**Example:** Consider an array A containing five elements, as shown below

	22	77	3	-1	5	
	A[0]	A[1]	A[2]	A[3]	A[4]	

What would be the resultant array if it is sorted in  
ascending order  
descending order

**Solution :** Array A sorted in ascending order

	-1	3	5	22	77	
	A[0]	A[1]	A[2]	A[3]	A[4]	

Array A sorted in descending order.

	77	22	5	3	-1	
	A[0]	A[1]	A[2]	A[3]	A[4]	

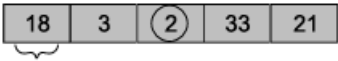
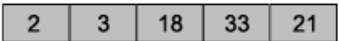
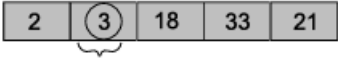



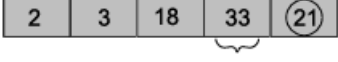
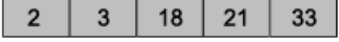


## 1) Selection Sort

Selection sort is one of the most basic sorting techniques. It works on the principle of identifying the smallest element in the list and moving it to the beginning of the list. This process is repeated until all the elements in the list are sorted. Let us consider an example where a list L contains five integers stored in a random fashion, as shown in figure.

18	3	2	33	21
List L				

### List of integers

Now, if the list L is sorted using selection sort technique then first of all the first element in the list, i.e., 18 will be selected and compared with all the remaining elements in the list. The element which is found to be the lowest amongst the remaining set of elements will be swapped with the first element. Then, the second element will be selected and compared with the remaining elements in the list. This process is repeated until all the elements are rearranged in a sorted manner. Below table illustrates the sorting of list L in ascending order using selection sort

Pass	Comparison	Resultant Array
1		
2		
3		
4		
<p>  → denotes the currently selected element   → denotes the smallest element identified in the current pass </p>		

A single iteration of the selection sorting technique that brings the smallest element at the beginning of the list is called a pass. As we can see in the above table, four passes were required to sort a list of five elements. Hence, we can say that selection sort requires  $n-1$  passes to sort an array of  $n$  elements.

**Write an algorithm to perform selection sort on a given array of integers.**

**selection**(arr[], size)

Step 1: Start

Step 2: Set  $i = 0$ ,  $loc = 0$  and  $temp = 0$

Step 3: Repeat Steps 4-6 while  $i < size$

Step 4: Set  $loc = \text{Min}(\text{arr}, i, \text{size})$

Step 5: Swap the elements stored at  $\text{arr}[i]$  and  $\text{a}[loc]$  by performing the following steps

I Set  $temp = \text{a}[loc]$

II Set  $\text{a}[loc] = \text{a}[i]$

III Set  $\text{a}[i] = temp$

Step 6: Set  $i = i + 1$

Step 8: Stop

**Min**(array[], LB, UB)

Step 1: Start

Step 2: Set  $m = LB$

Step 3: Repeat Steps 4-6 while  $LB < UB$

Step 4: if  $\text{array}[LB] < \text{array}[m]$  goto Step 5 else goto Step 6

Step 5: Set  $m = LB$

Step 6: Set  $LB = LB + 1$

Step 7: Return  $m$

Step 8: Stop

## 2) Insertion Sort

Insertion sort method sorts a list of elements by inserting each successive element in the previously sorted sublist. Such insertion of elements requires the other elements to be shuffled as required.

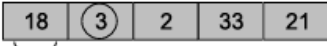
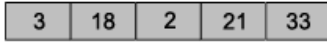


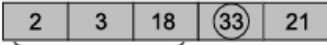
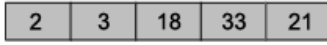


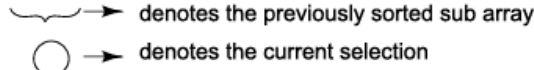
To understand the insertion sorting method, consider a scenario where an array A containing n elements needs to be sorted. Now, each pass of the insertion sorting method will insert the element A[i] into its appropriate position in the previously sorted subarray, i.e., A[1], A[2], ..., A[i-1]. The following list describes the tasks performed in each of the passes:

Pass 1 A[2] is compared with A[1] and inserted either before or after A[1]. This makes A[1], A[2] a sorted sub array.

Pass 2 A[3] is compared with both A[1] and A[2] and inserted at an appropriate place. This makes A[1], A[2], A[3] a sorted sub array.

Pass n-1 A[n] is compared with each element in the sub array A[1], A[2], A[3], ... A[n-1] and inserted at an appropriate position. This eventually makes the entire array A sorted.

Table illustrates the sorting of list L in ascending order using **insertion sort** technique.

Pass	Comparison	Resultant Array
1		
2		
3		
4		
		

The above illustration, four passes were required to sort a list of five elements. Hence, we can say that insertion sort requires n-1 passes to sort an array of n elements.

**Write an algorithm to perform insertion sort on a given array of integers.**

**insertion(arr[], size)**

Step 1: Start

Step 2: Set i = 1, j = 0 and temp = 0

Step 3: Repeat Steps 4-12 while i < size

Step 4: Set temp = arr[i]

Step 5: Set j = i-1

Step 6: Repeat Steps 7-10 while j >= 0

Step 7: if  $\text{arr}[j] > \text{temp}$  goto Step 8 else goto Step 9

Step 8: Set  $\text{arr}[j+1] = \text{arr}[j]$

Step 9: Branch out and go to Step 11

Step 10: Set  $j = j-1$

Step 11: Set  $\text{arr}[j+1] = \text{temp}$

Step 12: Set  $i = i + 1$

Step 13: Stop

**Write a C program to perform insertion sort on an array of N elements.**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void insertion(int [], int);
void main()
{
    int *arr;
    int i, N;
    clrscr();
    printf("Enter the number of elements in the array:\n");
    scanf("%d",&N);
    arr = (int*) malloc(sizeof(int));
    printf("Enter the %d elements to sort:\n",N);
    for (i=0;i<N;i++)
        scanf("%d",&arr[i]);
    insertion(arr,N);
    printf("\nThe sorted elements are:\n");
    for(i=0;i<N;i++)
        printf("%d\n",arr[i]);
    getch();
}
```

```

void insertion(int array[], int size)
{
    int i,j,temp;
    for(i=1;i<size;i++)
    {
        temp=array[i];
        for(j=i-1;j>=0;j--)
            if(array[j]>temp)
                array[j+1]=array[j];
        else
            break;
        array[j+1]=temp;
    }
}

```

### **Output**

Enter the number of elements in the array: 5

Enter the 5 elements to sort:

18

3

2

33

21

The sorted elements are: 2

3

18

21


33

### 3) Bubble Sort

Bubble sort is one of the oldest and simplest of sorting techniques. It focuses on bringing the largest element to the end of the list with each successive pass. Unlike selection sort, it does not perform a search to identify the largest element; instead it repeatedly compares two consecutive elements and moves the largest amongst them to the right. This process is repeated for all pairs of elements until the current iteration moves the largest element to the end of the list.

consider a scenario where an array A containing n elements needs to be sorted. In the first pass, elements A[1] and A[2] are compared and if A[1] is larger than A[2] then the two values are swapped. Next, A[2] and A[3] are compared. The last comparison of the first pass between A[n-1] and A[n] brings the largest element of the list to the end. The second pass repeats this process for the remaining n-1 elements. Finally, the last pass compares only the first two elements i.e., A[1] and A[2] to generate the sorted list.

Table illustrates the sorting of list L in ascending order using **bubble sort**:

Pass	Comparison	Resultant Array
1	<div>18 3 2 33 21</div> <div>3 18 2 33 21</div> <div>3 2 18 33 21</div> <div>3 2 18 33 21</div>	<div>3 2 18 21 33</div>
2	<div>3 2 18 21 33</div> <div>2 3 18 21 33</div> <div>2 3 18 21 33</div>	<div>2 3 18 21 33</div>
3	<div>2 3 18 21 33</div> <div>2 3 18 21 33</div>	<div>2 3 18 21 33</div>
4	<div>2 3 18 21 33</div>	<div>2 3 18 21 33</div>
 denotes the pair of consecutive elements being compared		

The above illustration, four passes were required to sort a list of five elements. Hence, we can say that bubble sort requires n-1 passes to sort an array of n elements.

**Write an algorithm to perform bubble sort on a given array of integers.**

**bubble(arr[], size)**

Step 1: Start

Step 2: Set  $i = \text{size}$ ,  $j = 0$  and  $\text{temp} = 0$

Step 3: Repeat Steps 4-9 while  $i > 1$

Step 4: Set  $j = 0$

Step 5: Repeat Steps 6-8 while  $j < i-1$

Step 6: if  $\text{arr}[j] > \text{arr}[j+1]$  goto Step 7 else goto Step 8

Step 7: Swap the elements stored at  $\text{arr}[j]$  and  $\text{arr}[j+1]$  by performing the following steps I Set  $\text{temp} = \text{arr}[j+1]$  II Set  $\text{arr}[j+1] = \text{arr}[j]$  III Set  $\text{arr}[j] = \text{temp}$

Step 8: Set  $j = j + 1$

Step 9: Set  $i = i - 1$

Step 10: Stop

**Write a C program to perform bubble sort on an array of N elements.**

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void bubble(int [], int);

void main()

{

    int *arr;

    int i, N;

    clrscr();

    printf("Enter the number of elements in the array:\n");

    scanf("%d",&N);

    arr = (int*) malloc(sizeof(int));

    printf("Enter the %d elements to sort:\n",N);

    for (i=0;i<N;i++)

        scanf("%d",&arr[i]);

    bubble(arr,N);

    printf("\nThe sorted elements are:\n");

    for(i=0;i<N;i++)
```



```

printf("%d\n",arr[i]);
getch();
}
void bubble(int array[], int size)
{
int i, j, temp;
for(i=size;i>1;i--)
for(j=0;j<i-1;j++)
if (array[j]>array[j+1])
{
temp = array[j+1];
array[j+1] = array[j];
array[j] = temp;
}
}

```

## Output

Enter the number of elements in the array: 5

Enter the 5 elements to sort: 18 3 2 33 21

The sorted elements are: 2 3 18 21 33

## 4) Quick Sort

Quick sort is one of the fastest sorting methods that is based on divide and conquer strategy. It divides the given list into a number of sub lists and then works on each of the sub lists to obtain the sorted output. It first chooses one of the list elements as a key value and then tries to place the key value at its final position in the list. Once, the key value is positioned correctly, the two sub lists to the left and right of the key value are processed in the similar fashion until the entire list becomes sorted.

Consider an array containing six elements : 34 99 5 2 57 40

Initially, the first list element i.e., 34 is chosen as the pivot element. Now, the list is scanned from right to left to identify the first element that is less than 34. This element is 2. So, both the elements are swapped and the list becomes:

2      99      5      34      57      40

Now, the list is scanned from left to right till the place where 34 is stored and the control stops at the first element that is greater than 34. This element is 99. So, both the elements are swapped and the list becomes:

2      34      5      99      57      40

Now, the list is again scanned from right to left starting with element 99 and ending at element 34. Element 5 is found to be lesser than 34, thus both the elements are swapped. Now, the list becomes:

2      5      34      99      57      40

Now, there are no elements present between 5 and 34, thus we can assume that 34 has attained its final position in the list. Now, the two sublists to the left and right of the pivot element are identified, as shown below:

2      5      (34)      99      57      40

Now, each of these lists is processed in the same fashion and eventually all the elements are placed at appropriate positions in the final sorted list.

**Write an algorithm to perform quick sort on a given array of integers.**

**quick(arr[], LB, UB)**

Step 1: Start

Step 2: Set pivot=0, next\_pvt=0, left=LB, right=UB

Step 3: Set pivot =arr[left] to select the first element as the pivot element

Step 4: Repeat Steps 5-14 while LB < UB    Step 5: Repeat Step 6 while arr[UB] >= pivot and LB < UB

Step 6: Set UB = UB - 1

Step 7: if LB is not equal to UB goto Step 8 else goto Step 10

Step 8: Set arr[LB]=arr[UB]

Step 9: Set LB = LB + 1

Step 10: Repeat Step 11 while arr[LB] <= pivot and LB < UB

Step 11: Set LB = LB + 1

Step 12: if LB is not equal to UB goto Step 13 else goto Step 15

Step 13: Set arr[UB]=arr[LB]

Step 14: Set  $UB = UB - 1$

Step 15: Set  $arr[LB] = \text{pivot}$

Step 16: Set  $\text{next\_pvt} = LB + 1$

Step 17: Set  $LB = \text{left}$  and  $UB = \text{right}$

Step 18: if  $LB < \text{next\_pvt}$  goto Step 19 else goto Step 20

Step 19: Apply quick sort in the left sub list by calling module  $\text{quick}(arr, LB, \text{next\_pvt}-1)$

Step 20: if  $UB > \text{next\_pvt}$  goto Step 21 else goto Step 22

Step 21: Apply quick sort in the right sub list by calling module  $\text{quick}(arr, \text{next\_pvt}+1, UB)$

Step 22: Stop

**Write a C program to perform quick sort on an array of N elements.**

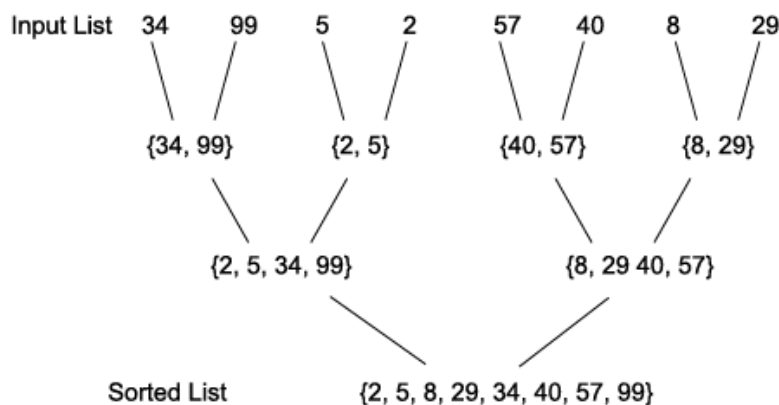
## 5) Merge Sort

Merge sort is another sorting technique that is based on divide-and-conquer approach. It divides a list into several sub lists of equal sizes and sorts them individually. It then merges the various sub lists in pairs to eventually form the original list, while ensuring that the sort order is not disturbed.

Consider a list L containing n elements on which merge sort is to be performed. Initially, the n elements of the list L are considered as n different sublists of one element each. Since, a list having one element is sorted in itself, thus there is no further action required on these sublists. Now, each of the sublists is merged in pairs to form  $n/2$  sublists having two elements each. While merging two lists the elements are compared and placed in a sorted fashion in the new sublist. This process is repeated until the original list is formed with elements arranged in a sorted fashion.

Consider an array containing six elements: 34 99 5 2 57 40 29

Following figure shows how merge sort is performed on the above list.



## Merge Sort

The above illustration, the sorted sublists are progressively merged in each pass to eventually generate the original list, sorted in ascending order.

**Write an algorithm to perform merge sort on a given array of integers.**

**mergesort**(arr[], size)

Step 1: Start

Step 2: Set mid = 0

Step 3: if size = 1 goto Step 4 else goto Step 5

Step 4: Stop and return back to the calling module

Step 5: Set mid = size / 2

Step 6: Call module mergesort(arr, mid)

Step 7: Call module mergesort(arr+mid, size-mid)

Step 8: Call module merge(arr, mid, arr+mid, size-mid)

Step 9: Stop

**merge**(a[], size1, b[], size2)

Step 1: Start

Step 2: Initialize a temporary array, temp\_array[size1+size2]

Step 3: Set i=0, j=0, k=0

Step 4: Repeat Step 5-9 while i < size1 and j < size2

Step 5: If a[i] < b[j] goto Step 6 else goto Step 8

Step 6: Set temp\_array[k] = a[i]

Step 7: Set k = k + 1 and i = i + 1

Step 8: Set temp\_array[k] = b[j]

Step 9: Set k = k + 1 and j = j + 1

Step 10: Repeat Step 11-12 while i < size1

Step 11: Set temp\_array[k] = a[i]

Step 12: Set k = k + 1 and i = i + 1

Step 13: Repeat Step 14-15 while j < size2

Step 14: Set temp\_array[k] = b[j]

Step 15: Set  $k = k + 1$  and  $j = j + 1$

Step 16: Set  $a[] = \text{temp\_array}[]$

Step 17: Stop

**Write a C program to perform merge sort on an array of N elements.**

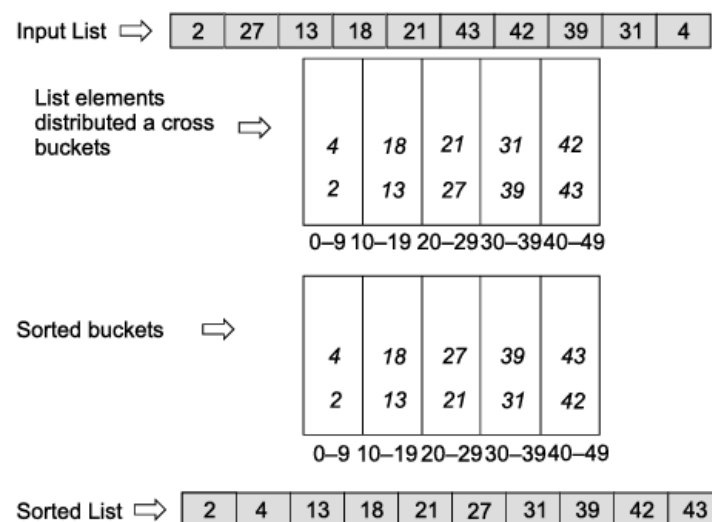
## 6) Bucket Sort

Bucket sort distributes the list of elements across different buckets in such a way that any bucket  $m$  contains elements greater than the elements of bucket  $m-1$  but less than the elements of bucket  $m+1$ . The elements within each bucket are sorted individually either by using some alternate sorting technique or by recursively applying bucket sort technique. In the end, elements of all the buckets are merged to generate the sorted list. This technique is particularly effective for smaller range of data series.

Consider a list containing ten integers stored in a random fashion.

2	27	13	18	21	43	42	39	31	4
---	----	----	----	----	----	----	----	----	---

In the above list, all elements are between the range of 0 to 50. So, let us create five buckets for storing ten elements each. Bucket sort figure (below fig) shows how these buckets are used for sorting the list.



### Bucket sort

**Write an algorithm to perform bucket sort on a given array of integers.**

**Assumption:** The input list elements are within the range of 0 to 49.

bucket(arr[], size)

Step 1: Start

Step 2: Set  $i = 0$ ,  $j = 0$  and  $k = 0$

Step 3: Initialize an array  $c[5]$  and set all its values to 0; it keeps track of the number of elements in each of the five buckets

Step 4: Create five buckets by initializing a 2-D array  $b[5][10]$  and set all its value to 0

Step 5: Now, distribute the input list elements across different buckets. To do this, repeat Steps 6-16 while  $i < \text{size}$

Step 6: if  $0 \leq \text{arr}[i] \leq 9$  then goto Step 7 else goto Step 8

Step 7: Set  $b[0][c[0]] = \text{arr}[i]$  and  $c[0] = c[0] + 1$

Step 8: if  $10 \leq \text{arr}[i] \leq 19$  then goto Step 9 else goto Step 10

Step 9: Set  $b[1][c[1]] = \text{arr}[i]$  and  $c[1] = c[1] + 1$

Step 10: if  $20 \leq \text{arr}[i] \leq 29$  then goto Step 11 else goto Step 12

Step 11: Set  $b[2][c[2]] = \text{arr}[i]$  and  $c[2] = c[2] + 1$

Step 12: if  $30 \leq \text{arr}[i] \leq 39$  then goto Step 13 else goto Step 14

Step 13: Set  $b[3][c[3]] = \text{arr}[i]$  and  $c[3] = c[3] + 1$

Step 14: if  $40 \leq \text{arr}[i] \leq 49$  then goto Step 15 else goto Step 16

Step 15: Set  $b[4][c[4]] = \text{arr}[i]$  and  $c[4] = c[4] + 1$

Step 16: Set  $i = i + 1$

Step 17: Sort each of the buckets  $b[i][j]$  by calling insertion sort module  $\text{insertion}(\&b[i][j], c[i])$

Step 18: Merge all the buckets together into the main array by setting  $\text{arr}[] = b[i][j]$

Step 19: Stop

## Searching Techniques

Searching refers to determining whether an element is present in a given list of elements or not. If the element is found to be present in the list then the search is considered as successful, otherwise it is considered as an unsuccessful search. The search operation returns the location or address of the element found.

There are various searching methods that can be employed to perform search on a data set. The choice of a particular searching method in a given situation depends on a number of factors, such as

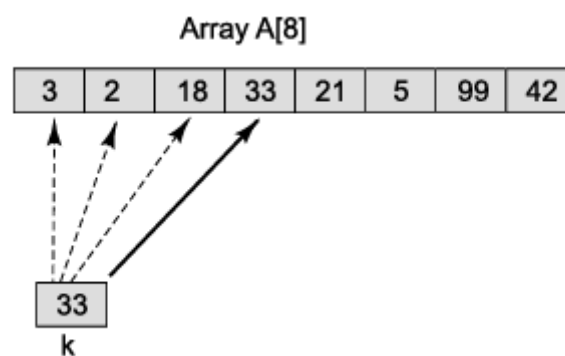
- order of elements in the list, i.e., random or sorted
- size of the list

## 1) Linear Search

It is one of the conventional searching techniques that sequentially searches for an element in the list. It typically starts with the first element in the list and moves towards the end in a step-by-step fashion. In each iteration, it compares the element to be searched with the list element, and if there is a match, the location of the list element is returned.

Consider an array of integers A containing n elements. Let k be the value that needs to be searched. The linear search technique will first compare A[0] with k to find out if they are same. If the two values are found to be same then the index value, i.e., 0 will be returned as the location containing k. However, if the two values are not same then k will be compared with A[1]. This process will be repeated until the element is not found. If the last comparison between k and A[n-1] is also negative then the search will be considered as unsuccessful.

Figure depicts the linear search technique performed on an array of integers.



### Linear search

As shown in above figure the value k is repeatedly compared with each element of the array A. As soon as the element is found, the corresponding index location is returned and the search operation is terminated.

**Write an algorithm to perform linear search on a given array of integers.**

**linear(arr[], size, k)**

Step 1: Start

Step 2: Set  $i = 0$

Step 3: Repeat Steps 4-6 while  $i < \text{size}$

Step 4: if  $k = \text{arr}[i]$  goto Step 5 else goto Step 6

Step 5: Return i and goto Step 9

Step 6: Set  $i = i + 1$

Step 7: If  $i = \text{size}$  goto Step 8 else goto Step 9

Step 8: Return NULL and goto Step 9

Step 9: Stop

**Write a C program to perform linear search on an array of N elements.**

```
#include

#include

#include

int linear(int [], int, int); /*Function prototype for performing linear search*/

void main()

{

    int *arr;

    int i, N, k, index;

    clrscr();

    printf("Enter the number of elements in the array arr:\n");

    scanf("%d",&N);

    arr = (int*) malloc(sizeof(int)); /*Dynamic allocation of memory for the array*/

    printf("\nEnter the %d elements of the array arr:\n",N);

    for (i=0;i<N;i++)

        scanf("%d",&arr[i]); /*Reading array elements*/

    printf("\nEnter the element to be searched:\n");

    scanf("%d",&k);

    index=linear(arr,N,k);

    if(index==-999)

        printf("\nElement %d is not present in array arr[%d]",k,N);

    else

        printf("\nElement %d is stored at index location %d in the array arr[%d]",k,index,N);

    getch();

}

int linear(int array[], int size, int num)

{

    int i;

    for(i=0;i<size;i++)

        if(num==array[i])
```



```

return(i);

if(i==size)

return(-999); /*Unsuccessful Search*/

}

```

## Output

Enter the number of elements in the array arr: 8

Enter the 8 elements of the array arr: 3 2 18 33 21 5 99 42

Enter the element to be searched: 33

Element 33 is stored at index location 3 in the array arr[8]

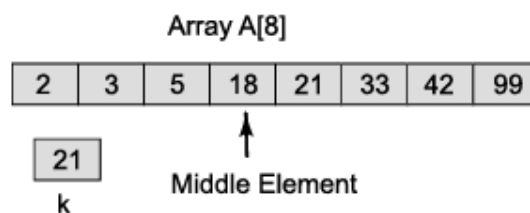
## 2) Binary Search

Binary search technique has a prerequisite – it requires the elements of a data structure (list) to be already arranged in a sorted manner before search can be performed in it. It begins by comparing the element that is present at the middle of the list. If there is a match then the search ends immediately and the location of the middle element is returned. However, if there is a mismatch then it focuses the search either in the left or the right sub list depending on whether the target element is lesser than or greater than middle element. The same methodology is repeatedly followed until the target element is found.

Binary search follows the same analogy as that of a telephone directory that we had discussed earlier. One needs to keep focusing on a smaller subset of directory pages every time there is a mismatch. However, such a search would not have been possible had the directory entries were not already sorted.

Consider an array of integers A containing eight elements, as shown in Fig.

Let  $k = 21$  be the value that needs to be searched.



### Binary search

we can see in Fig. 10.6, the array A on which binary search is to be performed is already sorted. The following steps describe how binary search is performed on array A to search for value  $k$ :

- First of all, the middle element in the array A is identified, which is 18.

- Now, k is compared with 18. Since k is greater than 18, the search is focused on the right sub list.
- The middle element in the right sub list is 33. Since k is less than 33, the search is focused on the left sub list, which is {21, 33}.
- Now, again k is compared with the middle element of {21, 33}, which is 21. Thus, it matches with k.
- The index value of 21, i.e., 4 is returned and the search is considered as successful.

**Write an algorithm to perform binary search on a given array of integers.**

binary(arr[], size, num)

Step 1: Start

Step 2: Set  $i = 0$ ,  $j = \text{size}$ ,  $k = 0$

Step 3: Repeat Steps 4-9 while  $i \leq j$

Step 4: Set  $k = (i + j)/2$

Step 5: If  $\text{arr}[k] = \text{num}$  goto Step 6 else goto Step 7

Step 6: return k and goto Step 11

Step 7: If  $\text{array}[k] < \text{num}$  goto Step 8 else goto Step 9

Step 8:  $i = k + 1$

Step 9:  $j = k - 1$

Step 10: Return NULL and goto Step 11

Step 11: Stop

**Write a C program to perform binary search on an array of N elements.**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

int binary(int [], int, int);

void main()
{
    int *arr;
    int i, N, k, index;

    clrscr();
```

```

printf("Enter the number of elements in the array arr:\n");
scanf("%d",&N);
arr = (int*) malloc(sizeof(int));
printf("\nEnter the %d elements of the array arr in sorted format:\n",N);
for (i=0;i<N;i++)
scanf("%d",&arr[i]);
printf("\nEnter the element to be searched:\n");
scanf("%d",&k);
index=binary(arr,N,k);
if(index==-999)
printf("\nElement %d is not present in array arr[%d]",k,N);
else
printf("\nElement %d is stored at index location %d in the array arr[%d]",k,index,N);
getch();
}
int binary(int array[], int size, int num)
{
int i=0,j=size, k;
while(i<=j)
{
k=(i+j)/2;
if(array[k]==num)
return(k); /*Successful search*/
else if(array[k] < num)
i=k+1;
else j=k-1;
}
return(-999); /*Unsuccessful search*/
}

```

## Output

Enter the number of elements in the array arr: 8

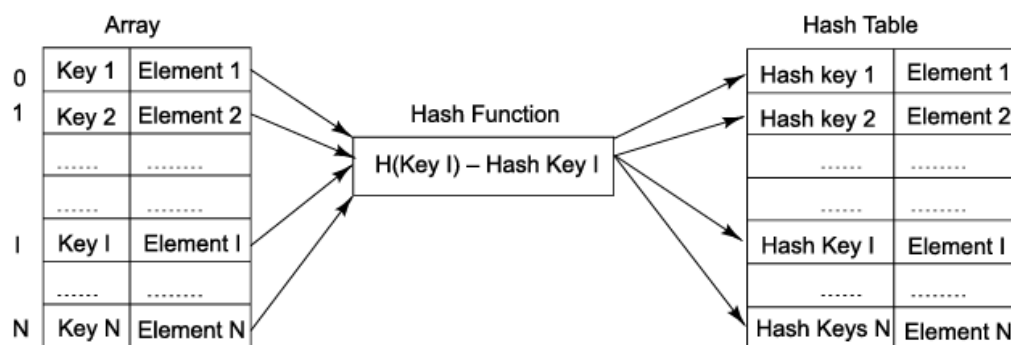
Enter the 8 elements of the array arr in sorted format: 2 3 5 18 21 33 42 99

Enter the element to be searched:21

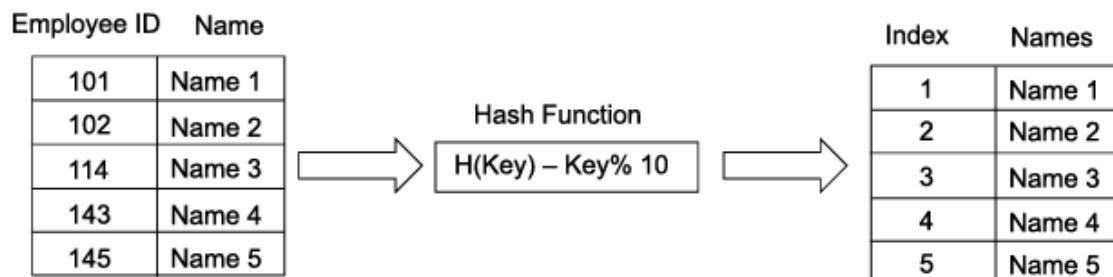
Element 21 is stored at index location 4 in the array arr[8]

### 3) Hashing

Hashing finds the location of an element in a data structure without making any comparisons. In contrast to the other comparison-based searching techniques, like linear and binary search, hashing uses 294 Data Structures Using C a mathematical function to determine the location of an element. This mathematical function called hash function accepts a value, known as key, as input and generates an output known as hash key. The hash function generates hash keys and stores elements corresponding to each hash key in the hash table. The keys that hash function accepts as input could be a digit associated with the input elements. In other words, we can say that a hash table is a data structure, which is implemented by a hash function and used for searching elements in quick time. In a hash table, hash keys act as the addresses of the elements.



Let us consider a simple example of a file containing information for five employees of a company. Each record in that file contains the name and a three digit numeric Employee ID of the employee. In this case, the hash function will implement a hash table of five slots using Employee IDs as the keys. That means, the hash function will take Employee IDs as input and generate the hash keys, as shown in figure.



Generating hash keys

In the hash table generated in the above example, the hash function is  $\text{Employee ID} \% 10$ . Thus, for Employee ID 101, hash key will be calculated as 1. Therefore, Name1 will be stored at position 1 in the hash table. For Employee ID 102, hash key will be 2, hence Name2 will be stored at position 2 in the hash table. Similarly, Name3, Name4, and Name5 will be stored at position 4, 3, and 5 respectively, as shown in linear search figure. Later, whenever an employee record is searched using the Employee ID, the hash function will indicate the exact position of that record in the hash table.

### **Problems:**

Consider the following array of integers: 35 18 7 12 5 23 16 3 1

Create a snapshot of the above array at each pass if the bubble sorting technique is applied on it.

### **Solution**

Initial array 35 18 7 12 5 23 16 3 1

Pass 1 1 35 18 12 7 23 16 5 3

Pass 2 1 3 35 18 12 23 16 7 5

Pass 3 1 3 5 35 18 23 16 12 7

Pass 4 1 3 5 7 35 23 18 16 12

Pass 5 1 3 5 7 35 23 18 16

Pass 6 1 3 5 7 12 16 35 23 18

Pass 7 1 3 5 7 12 16 18 35 23

Pass 8 1 3 5 7 12 16 18 23 35

**Problem 2:** Consider the following array of integers: 74 39 35 32 97 84

Initial array 74 39 35 32 97 84

Pass 1 32 39 35 74 97 84

Pass 2 32 35 39 74 97 84

Pass 3 32 35 39 74 97 84

Pass 4 32 35 39 74 97 84

Pass 5 32 35 39 74 84 97 (sorted array)