# 1. DATA STRUCTURES

## UNIT - I

**Data Structures** - Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space complexity.
**Searching** - Linear search, Binary search, Fibonacci search.
**Sorting**- Insertion sort, Selection sort, Exchange (Bubble sort, quick sort), distribution (radix sort), merging (Merge sort) algorithms.

### (1) Definition:

- Data Structures is the concept of set of algorithms used to structure the information.
- These algorithms are implemented using C, C++, Java, etc
- Structure the information means store and process data in an efficient manner.
- To store and process data we may use the following operations
  1.create() 6.sorting()
  2.insert() 7.merging()
  3.delete() 8.splitting()
  4.display() 9.traversal()
  5.searching()
- So data structure may contain algorithms, use for different operations implement these algorithms by a programming language
- For example for stack data structure write algorithms for different operations
  1.push,
  2.pop and
  3.display.
- Implement these algorithms in a particular language say C.

### (2) Classification of Data Structures:

- Data structures are normally classified into two types.
- They are primitive data structures and non-primitive data structures.

#### (i)Primitive data structures:

- Primitive data structures are built in types in most programming languages. They are
- Integer: It is whole numbers. i.e. negative values,0,positive values
- Float: It is fractional numbers
- Character: It is character values
- Boolean: it represents true or false.

#### (ii)Non-primitive data structures:

- These are derived from primitive data structures.
- They are Array, Structure, Union, Files etc
- A Non-primitive data type is further divided into Linear and Non-Linear data structure.

**(a)Linear data structures:**

- Here the data elements are connected in a sequence manner.
- Examples are Arrays, Linked List, Stacks and Queues.

**Array:**

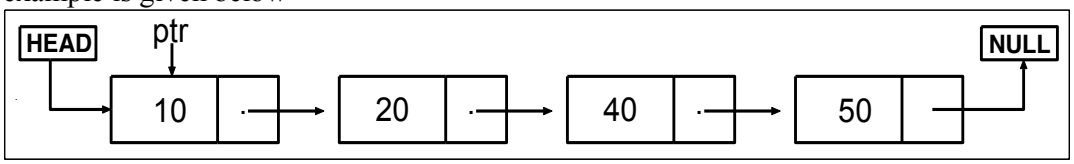
- It is collection of elements of the same type

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| a[5]= | 10 | 20 | 30 | 40 | 50 |

**Linked List:**

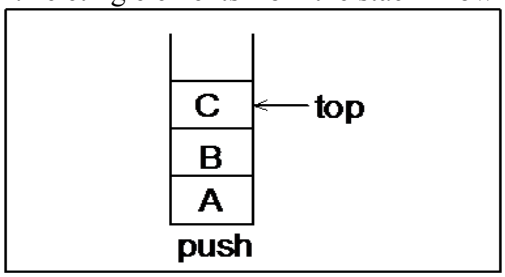
- *linked list* or single linked list is a sequence of elements in which every element has *link* to its next element in the sequence.
- Every element is called as a "node". Every "node" contains two fields, *data* and link. The data is a value or string and link is an address of next node.

- The first node is called HEAD which is an empty node contains an address of the first node so it link to the first node.
- The first node link to the second node and so on.
- The last node does not link to address but link to NULL. Let ptr be a pointer to the linked list. The example is given below



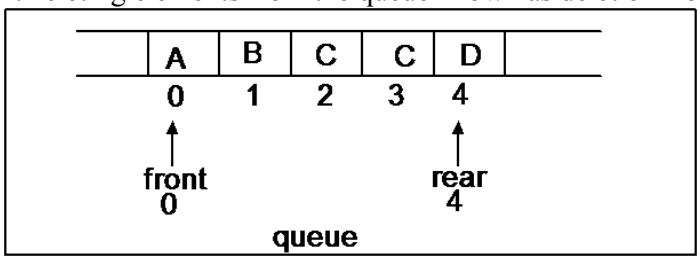

**Stack:**

- A stack is a data structure in which additions and deletions are made at the top of the stack. So we can perform two operations on stack.
  1. Adding elements into the stack known as push;
  2. Deleting elements from the stack known as pop




**Queue:**

- A queue is a data structure in which additions are made at one end and deletions are made at the other end. We can represent a queue in an array.
- Here we can perform two operations on queue.
  1. Adding elements into the queue known as insertion at rear
  2. Deleting elements from the queue known as deletion from front

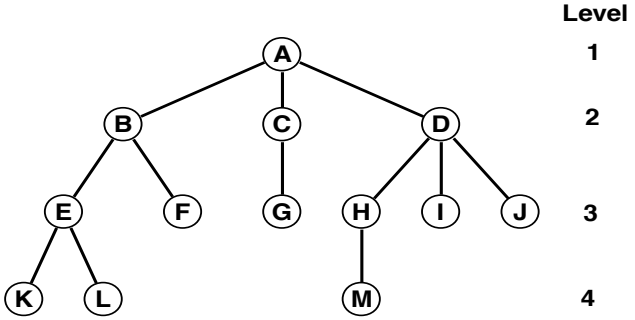



**(b)Non-linear data structures:**

- Here data elements are not connected in a sequence manner.
- Examples are: Trees and Graphs.

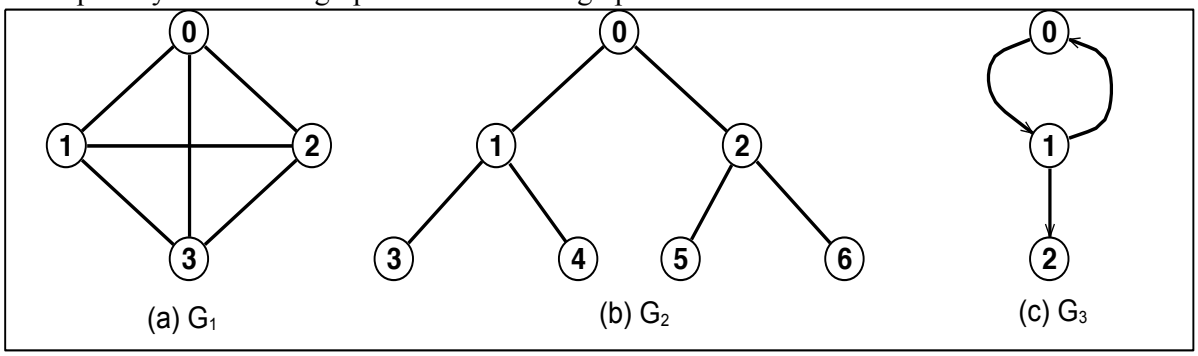
**Tree:**

- The tree is defined as a finite set of one or more nodes such that
  1.One node is called a root node and
  2.Remaining nodes partitioned into sub trees of the root.



**Graph:**

- A graph is a pictorial representation of a set of points or nodes termed as **vertices**, and the links that connect the vertices are called **edges**.
- A Graph(G) consists of two sets V and E where V is called vertices and E is called edges. We also write G = (V,E) to represent a graph.
- A Graph may be directed graph and undirected graph.




- The Fig(a),Fig(b) are called undirected graph & Fig(c) is called directed graph.

**Differences between Linear and Non Linear Data Structures:**

| Linear Data Structure | Non-Linear Data Structure |
|---|---|
| Every data element is connected to its previous & next one | Every data element is connected with many other data elements. |
| Data is arranged in a sequence manner | Data is not arranged in a sequence manner |
| Data can be traversed in a single run | Data cannot be traversed in a single run |
| Ex: Array, Stack, Queue, Linked List | Ex: Tree, Graph |
| Implementation is easy | Implementation is difficult |

## (3) Operations on Data Structures:

- The different operations used on data structure are:

**1.Create:**

- Here we reserve memory for program elements: This can be done by using malloc or calloc function. We can create a data structure with giving different elements.

**2.Insert:**

- Here we reserve memory for program element. This can be done by using malloc or calloc function. We can insert an element into a data structure.

**3.Delete:**

- It delete memory space allocated for specified data structure using free().

**4.Display:**

- It deals with accessing a particular data within a data structure.

**5.Searching:**

- It finds the data item in the list of data items.
- It also find the location of all elements.

**6.Sorting:**

- It is the process of arranging all data items in a data structure in a particular order say either in ascending order or in descending order.

**7.Merging:**

- It is the process of combining the data items of two different sorted list into a single sorted list.

**8.Splitting:**

- It is the process of partitioning single list to multiple list.

**9.Traversal:**

- It is the process of visiting each and every node of a list in systematic manner.

## (4) Abstract Data Type(ADT):

- An abstract data type (ADT) is a collection of values and a set of operations without specifying its implementation.
- For example in Array ADT set of values are index and item & set of operations are create(),retrieval() and store().
- The purpose of the ADT is to hide the implementation details of a data structure, thus improving software maintenance, reuse and portability.
- The developers of ADT will adapt changing requirements and save time.
- The users of ADT are concerned with the interface, but not the implementation.
- The different ADTs given by

  String ADT, List ADT, Stack (last-in, first-out) ADT,
  Queue (first-in, first-out) ADT
  Binary Search Tree ADT etc

**Example:**

- A List ADT contains operations known as add element, remove element, etc.
- A List ADT can be represented by an array-based implementation or a linked-list based implementation. In this the linked-list based implementation is so commonly used.
- Similarly, a Binary Search Tree ADT can be represented in different ways with the same operations known as insert, remove, display, etc.

**(i)The Array as an Abstract Data Type:**

- The Array ADT is a set of values (index, item) and a set of operations known as Array create(), Item Retrieve(), and Array Store().

The Array ADT algorithm is given by
Array ADT is
**objects:** A set of pairs <index,item> where for each index there is an item.
**functions:** for all A€Array, i€index, x€item
Array create() - It creates a new empty array
Item Retrieve(A,i) - It returns a value with a particular index, if the index is valid or an error if the index is invalid
Aray Store(A,i,x) - It stores an item
end Array

**(ii)The Stack as an Abstract Data Type:**

ADT stack is
**objects:** a finite ordered list with zero or more elements
**functions:** S € Stack, item € Element
Stack create() := create an empty stack
Stack push(S,item) := insert item into top of stack
Element pop(S) :=remove and return the item at the top of the stack
end Stack

**(iii)The Queue as an Abstract Data Type:**

ADT queue is
**objects:** a finite ordered list with zero or more elements
**functions:** Q € Queue, item € Element
Queue Create() := create an empty stack
Queue addq(Q,item) := insert item into queue
Element deleteq(Q) :=remove and return the item from the queue
end Stack

**(iv)The Binary Tree as an Abstract Data Type:**

ADT BinaryTree is
**objects:** a finite set of nodes
**functions:** for all bt, bt1, bt2 ϵ BinTree, item ϵElement
BinTree Create() := create an empty BinaryTree
Boolean IsEmpty := if(bt== empty binary tree) return TRUE else return FALSE

BinTree MakeBT(bt1,item,bt2) := return a binary tree whose left subtree is bt1, whose right subtree is bt2, and whose root node contains the data item.
BinTree Lchild(bt) := It return left subtree of bt
Element Data(bt) := It return the data in the root node of bt
BinTree Rchild(bt) := It return the right subtree of bt
End ADT BinaryTree

**(v)The Grapg as an Abstract Data Type:**

**ADT** Graph is
**objects:** a set of vertices & edges
**functions:** for all graph $\epsilon$ Graph, v, $v_1$ and $v_2$ $\epsilon$ Vertices
Graph Create() := create an empty Graph
Boolean IsEmpty() := if(graph== empty graph) return TRUE else return FALSE
Graph InsertVertex(graph,v) := return a graph with v inserted.
Graph InsertEdge(graph,$v_1$,$v_2$) := return a graph with a new edge ($v_1$, $v_2$) is inserted.
Graph DeleteVertex(graph,v) := return a graph in which vertex v is Removed
Graph DeleteEdge(graph,$v_1$,$v_2$) := return a graph in which the edge ($v_1$,$v_2$) is removed
End ADT Graph

## (5) Preliminaries of algorithms:

- Algorithm is step-by-step process or sequence of steps for solving a problem.

**Properties of an Algorithm**

- The properties of an algorithm is given by

**Input:**
The algorithm must have input values from a specified set
**Output:**
The algorithm must produce the output values from a specified set of input values. The output values are the solution to a problem.
F**initeness:**
For any input the algorithm must terminate after a finite number of steps.
**Definiteness:**
All the steps of the algorithm must be precisely defined.
**Effectiveness:**
It must be possible to perform each step of the algorithm correctly and in a finite amount of time.

- Each step should be well defined. It can be divided into 3 types.

**i. Sequence:**

- In an algorithm if all steps are shown then it is known as sequence.
- For example an algorithm for adding two values.

```
Ex: Step 1 : start
    Step 2 : read a,b
    Step 3 : r=a+b
    Step 4 : print r
    Step 5 : stop
```

**ii. Selection:**

- Here we use if condition and the condition is checked only one time.
- If a condition is satisfied then next statement is executed otherwise else statement is executed.
- For example an algorithm to check whether the given number is even or odd.

```
Ex: Step 1 : start
    Step 2 : read n
    Step 3 : if(n%2==0) goto step 4
             if not goto Step 6
    Step 4 : print "Even no" goto Step 7
    Step 5 : else
    Step 6 : print "Odd no"
    Step 7 : stop
```

**iii. Iteration:**

- ❖ Here we use while, do-while and for & the condition is checked number of times.
- ❖ i.e.The statements in an iteration block are executed no. of times based on some condition.
- ❖ For example an algorithm to print 1 to n numbers using while loop.

Ex: Step 1 : start
Step 2 : read n
Step 3 : initialize i=1
Step 4 :while(i<=10) goto Step-5
otherwise goto Step 7
Step 5 : print i
Step 6 : compute i++ goto Step 4
Step 7 : stop

- ➢ When an algorithm gets coded in a specified programming language such as C, C++, or Java, it becomes a program that can be executed on a computer.
- ➢ Multiple algorithms can exist to solve the same problem or complete the same task.
- ➢ The appropriate algorithm can be determined based on an number of factors:
  1. How long the algorithm takes to run
  2. What resources are required to execute the algorithm
  3. How much space or memory is required
  4. How exact is the solution provided by the algorithm

## (6) Time and Space complexity:

- ➢ Algorithm: Step by step process of solving a problem is called an algorithm.The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. There are two main complexity measures of the efficiency of an algorithm

### (i)Time complexity:

- ➢ Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- ➢ Time means the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed or some other natural unit related to the amount of real time the algorithm will take.

### (ii)Space complexity:

- ➢ Space complexity is a function describing the amount of memory an algorithm takes in terms of the amount of input to the algorithm. We often speak of extra memory needed, not counting the memory needed to store the input itself. We can use bytes, but it's easier to use say number of integers used, number of fixed-sized structures etc.
- ➢ In the end the function will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal.
- ➢ The complexity of an algorithm is studied with respect to the following 3 cases.

### (a)Worst Case Analysis:

- ➢ In the worst case analysis, we calculate upper bound on running time of an algorithm.
- ➢ We must know the case that causes maximum number of operations to be executed.
- ➢ For Linear Search the worst case happens when the element to be searched is not present in the array.

### (b)Average Case Analysis:

- ➢ In the average case analysis, we calculate average on running time of an algorithm. We must know the average number of operations to be executed.
- ➢ In the linear search problem, the average case occurs when x is present at average of its location.

### (c)Best Case Analysis:

- ➢ In the best case analysis, we calculate lower bound on running time of an algorithm.
- ➢ We must know the case that causes minimum number of operations to be executed
- ➢ In the linear search problem, the best case occurs when x is present at the first location.

# Searching

## (7) Searching:

- Searching is the process of finding a given value in a list of values.
- It is the algorithmic process of finding a particular item in a collection of items.

### (i) Linear Search:

**Definition:**

- It starts at the beginning of the list and checks every element of the list.
- i.e. It sequentially checks each element of the list until a match is found or the whole list has been searched. So it is also called sequential search.

**Example:**

- Let the elements are: 10,6,3,8,9,12,14
- The search element is : 12
- Now it compare 12 with each and every element.
- The 12 is available in $6^{th}$ place.
- So the searching process is success and element is found

**Algorithm:**

Step 1: Read elements in array
Step 2: Read the element to search
Step 3: Compare the element to sear and each element in array sequentially
Step 4: If match is found then the search success
Step 5: If match is not found upto the end then the search un success

**Program:**

```
#include <stdio.h>
int main()
{
        int a[100],n,i,s;
        printf("Enter Number of Elements in Array:\n");
        scanf("%d", &n);
        printf("Enter numbers:\n");
        for(i = 0; i < n; i++)
        scanf("%d",&a[i]);
        printf("Enter a number to search in Array:\n");
        scanf("%d", &s);
        for(i = 0; i < n; i++)
        {
                if(s==a[i])
                {
                        printf("Number found\n”);
                        break;
                }
        }
        if(i== n)
        printf("Number not found\n”);
        return 0;
}
```

### (ii) Binary Search:

**Definition:**

- Binary search is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that are used to solve problems.
- Binary search sorts the records either in ascending or descending order to gain much better performance than linear search.
- Now suppose we have an ascending order record. At the time of search it takes the middle record/element, if the searching element is greater than middle element then the element mush be located in the second part

else it is in the first half. In this way this search algorithm divides the records in the two parts in each iteration and thus called binary search.

**Example:**

- Let the elements in ascending order are
  2 4 6 8 10 12 15
- Let the element to search 12
- For searching it compare first middle element.
  2 4 6 8 10 12 15
- The middle element is 8 and is not equal to 12. Since 12 is greater than 8 search on right side part of 8. 12 is equal to right side part middle. So element is found.

**Algorithm:**

Step 1: Read sorted elements in array
Step 2: Read the element to search
Step 3: Compare the element to sear and middle element in array. If match is found the search success.
Step 4: If match is not found check the search element with middle element. If search element is greater than the middle element then search on right side of middle element otherwise search on left.
Step 5: This process is repeated for all elements in array. If no match is found upto the end then the search is not success.

**Program:**

```
#include <stdio.h>
int main()
{
    int i, first, last, middle, n, s, a[100];
    printf("Enter number of elements:\n");
    scanf("%d",&n);
    printf("Enter elements in ascending order:\n");
    for (i = 0; i < n; i++)
    scanf("%d",&a[i]);
    printf("Enter an element to search:\n");
    scanf("%d", &s);
    first = 0;
    last = n - 1;
    middle = (first+last)/2;
    while (first <= last)
    {
        if(s==a[middle])
        {
            printf("Element is found at index: %d",middle);
            break;
        }
        else if(s>a[middle])
        first = middle + 1;
        else if(s<a[middle])
        last = middle - 1;
        middle = (first + last)/2;
    }
    if (first > last)
    printf("Element is not found");
    return 0;
}
```

**(iii) Fibonacci Search:**

- Fibonacci Search uses Fibonacci numbers to search an element in a sorted array.
- Fibonacci numbers are: 0,1,1,2,3,5,8…
- Fibonacci series generates the subsequent number by adding two previous numbers

**Example:** Let the elements are given by

10, 22, 35, 40, 45, 50, 80, 82, 85, 90,100
arr[] = {10, 22, 35, 40, 45, 50, 80, 82, 85, 90,100}

- Fibonacci No's are: 0,1,1,2,3,5,8,13,21,… Target element x is 85. Length of array n = 11
- Find the smallest Fibonacci number greater than or equal to 11 is 13.
- As per our step, a= 5, b = 8, and c = a+b=13.
- Let offset=-1
- First Compute i= min(offset+a, n-1)
- If x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Else If x is less than the element, move the three Fibonacci variables two Fibonacci down. Else ( or when b=1) return i – This is the case Element Found
- If element not found return -1
- These steps are shown in the following table

| a | b | c | offset | i=min(offset +a, n-1) | arr[i] | Consequence |
|---|---|---|---|---|---|---|
| 5 | 8 | 13 | -1 | 4 | 45 | Move one down, reset offset |
| 3 | 5 | 8 | 4 | 7 | 82 | Move one down, reset offset |
| 2 | 3 | 5 | 7 | 9 | 90 | Move two down |
| 1 | 1 | 2 | 7 | 8 | 85 | Return i |

**Algorithm:**

Let arr[0..n-1] be the input array and element to be searched be x.
Step 1: Find the smallest Fibonacci Number greater than or equal to n. Let this number be c. Let the two Fibonacci numbers preceding it be a,b.
Step 2: While the array has elements to be inspected such as:
Step-2.1: Compute i= min(offset+a, n-1)
Step-2.2: If x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index.
Step-2.3: Else If x is less than the element, move the three Fibonacci variables two Fibonacci down
Step-2.4: Else ( or when b=1) return i – This is the case Element Found
Step-3: If element not found return -1

**Program:**

```
#include <stdio.h>
int min(int x, int y)
{
    return (x<=y)? x : y;
}
int search(int arr[], int x, int n)
{
    int a = 0;
    int b = 1;
    int c = a + b;
    while (c < n)
    {
        a = b;
        b = c;
        c  = a + b;
    }
    int offset = -1;
    while (c > 1)
    {
        int i = min(offset+a, n-1);
```

```
            if (x>arr[i])
            {
                    c  = b;
                    b = a;
                    a = c - b;
                    offset = i;
            }
            else if (x<arr[i])
            {
                    c  = a;
                    b = b - a;
                    a = c - b;
            }
            else return i;
        }
        return -1;
 }
 int main()
 {
       int arr[] = {10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100};
       int n = sizeof(arr)/sizeof(arr[0]);
       int x,s;
       printf("Enter an element to search:\n");
       scanf("%d",&x);
       s=search(arr, x, n);
      if(s==-1)
     printf("Element is not found");
     else
     printf("Element is Found at index: %d", s);
     return 0;
}
```

# Sorting

## (8) Sorting:

- Sorting is a process of placing a list of elements from the collection of data in some order.
- It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier

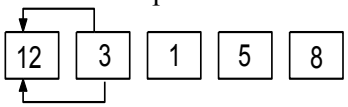
### (i) Insertion Sort:

- In this sorting technique first elements are stored in an array.
- The process of sorting starts with second element.
- First the second element is picked and is placed in specified order Next third element is picked and is placed in specified order. Similarly the fourth, fifth, …$n^{th}$ element .is placed in specified order.
- Finally we get the sorting elements.

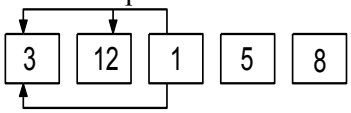
**Example:**

- Let us consider the elements: 12, 3, 1, 5, 8

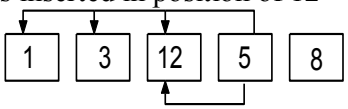
1. Checking second element of array with element before it and inserting it in proper position. In this case 3 is inserted in position of 12



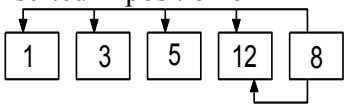

2. Checking third element of array with elements before it and inserting it in proper position. In this case 1 is inserted in position of 3




3. Checking fourth element of array with elements before it and inserting it in proper position. In this case 5 is inserted in position of 12




4. Checking fifth element of array with elements before it and inserting it in proper position. In this case 8 is inserted in position of 12




5. Sorted array in ascending order


**Algorithm:**

Step 1: Check second element of array with element before it and insert it in proper position.
Step 2: Checking third element of array with element before it and inserting it in proper position.
Step 3: Repeat this till all elements are checked.
Step 4: Stop

**Program:**

```
#include<stdio.h>
int main()
{
        int n,a[30],key,i,j;
        printf("Enter total elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        for(i=1;i<n;i++)
        {
                j=i;
```

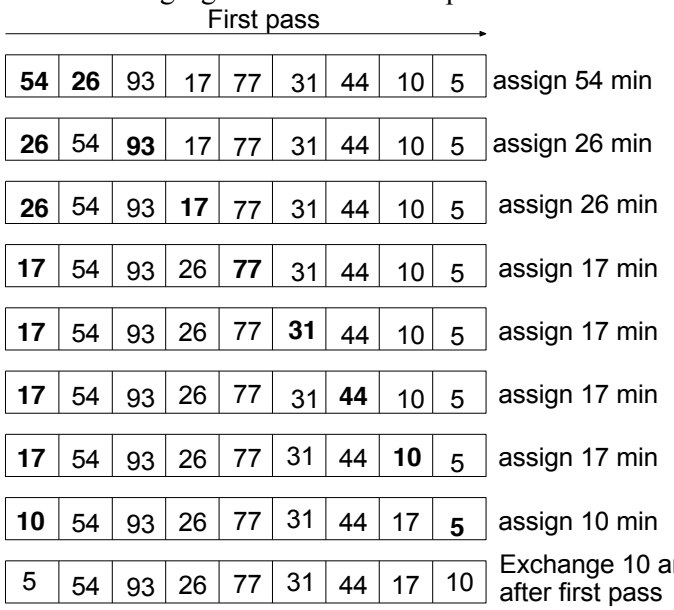
```
            while(j>0 && a[j]<a[j-1])
            {
                temp=a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
                j--;
            }
        }
        printf("After sorting is:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

**(ii) Selection Sort:**

- Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

**Example:**

- The following figure shows the first pass of a selection sort.




- In first pass the first element is compared with all remaining elements and exchange element if first one is greater than second so that the smallest value is in first place. Leave this element.
- In second pass compare second element to all elements and put the next smallest value, in second place. Leave this element. This process is repeated till all the elements are placed.
- Now we get the sorted elements.

**Algorithm:**

Step 1 − Set min to the first location.
Step 2 − Search the minimum element in the array.
Step 3 – swap the first location with the minimum value in the array.
Step 4 – assign the second element as min.
Step 5 − Repeat the process until we get a **sorted** array.

**Program:**

```
#include<stdio.h>
int main()
{
    int n,i,j,temp,a[20],min;
    printf("Enter total elements:\n");
    scanf("%d",&n);
```

```
    printf("Enter elements:\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);
    for(i=0;i<n;i++)
    {
        min=i;
        for(j=i+1;j<n;j++)
        {
            if(a[j] < a[min])
            min=j;
        }
        temp=a[i];
        a[i]=a[min];
        a[min]=temp;
    }
    printf("After sorting is:\n");
    for(i=0;i<n;i++)
    printf(" %d",a[i]);
    return 0;
}
```

**(iii) Exchange Sort:**

- The exchange sort is almost similar as the bubble sort. The exchange sort compares each element of an array and swap those elements that are not in their proper position,
  just like a bubble sort does. The only difference between the two sorting algorithms is the manner in which they compare the elements..

**(i) Bubble Sort:**

- Bubble Sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

**Example:**

- The following figure shows the first pass of a bubble sort. In first pass the first element is compared with second and exchange element if first one is greater than second.
- Similarly second element is compared with third and exchange element if second one is greater than third.
- Repeat this so that at the end of first pass the largest value is in last place. Leave this element.

First pass →

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **54** | **26** | 93 | 17 | 77 | 31 | 44 | 55 | 20 | Exchange |
| 26 | **54** | **93** | 17 | 77 | 31 | 44 | 55 | 20 | No Exchange |
| 26 | 54 | **93** | **17** | 77 | 31 | 44 | 55 | 20 | Exchange |
| 26 | 54 | 17 | **93** | **77** | 31 | 44 | 55 | 20 | Exchange |
| 26 | 54 | 17 | 77 | **93** | **31** | 44 | 55 | 20 | Exchange |
| 26 | 54 | 17 | 77 | 31 | **93** | **44** | 55 | 20 | Exchange |
| 26 | 54 | 17 | 77 | 31 | 44 | **93** | **55** | 20 | Exchange |
| 26 | 54 | 17 | 77 | 31 | 44 | 55 | **93** | **20** | Exchange |
| 26 | 54 | 17 | 77 | 31 | 44 | 55 | 20 | 93 | 93 in place after first pass |

- In second pass compare up to before last place value and put the next largest value, that before last place. Leave this element.

- This process is repeated till all the elements are placed. Now we get the sorted elements..

**Algorithm:**

Step 1: The first element is compared with second and exchange element if first one is greater than second
Step 2: Similarly second element is compared with third and exchange element if second one is greater than third
Step 3: Repeat this so that at the end the largest value is in last place
Step 4: Likewise sorting is repeated for all elements.

**Program:**

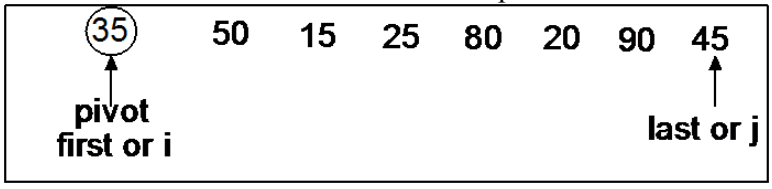
```
#include<stdio.h>
int main()
{
        int n,temp,i,j,a[20];
        printf("Enter total numbers of elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
              for(j=0;j<n-1;j++)
              {
                    if(a[j]>a[j+1])
                    {
                          temp=a[j];
                          a[j]=a[j+1];
                          a[j+1]=temp;
                    }
              }
        }
        printf("After sorting elements are:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

**(ii) Quick Sort:**

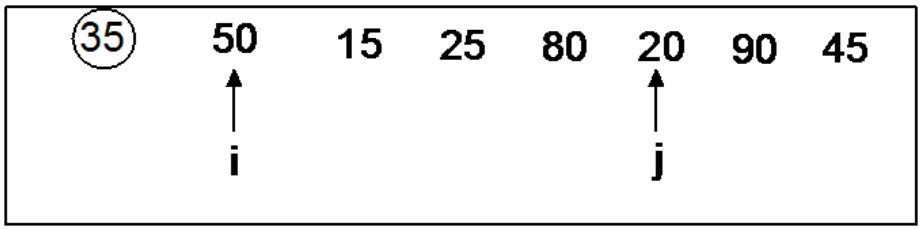
- Quick Sort is also one of the exchange sort.
- In a quick sort we take pivot element, then we place all the smaller elements are on one side of pivot, and greater elements are on other side of pivot.
- After partitioning we have pivot in the final position. After repeatedly partitioning, we get the sorted elements.

**Example:**

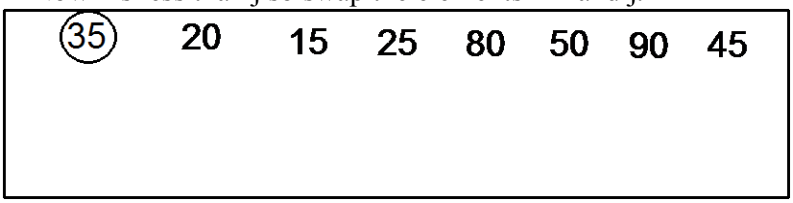
- Let us consider the elements:
  35,50,15,25,80,20,90,45
- Let us consider the first element 35 as pivot or i. The last element 45 as j



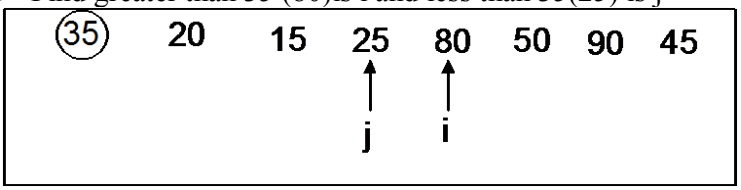

- 50 which is greater than pivot taken as i and the 20 smaller than pivot taken as j



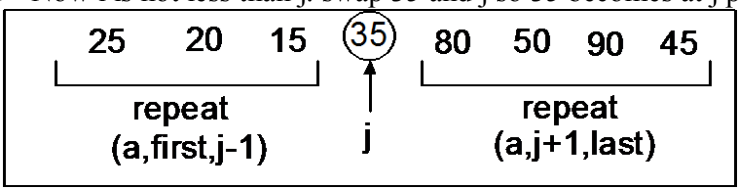

- Now i is less than j so swap the elements in i and j.



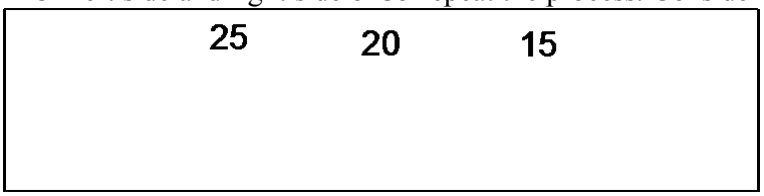

- Find greater than 35 (80)is i and less than 35(25) is j



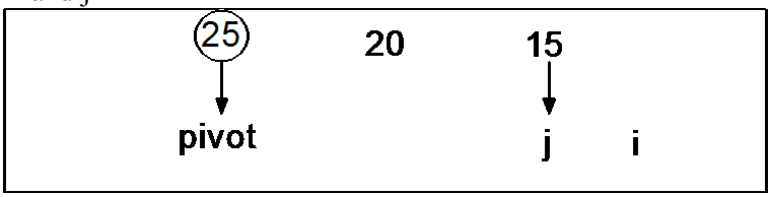

- Now i is not less than j. swap 35 and j so 35 becomes at j place.



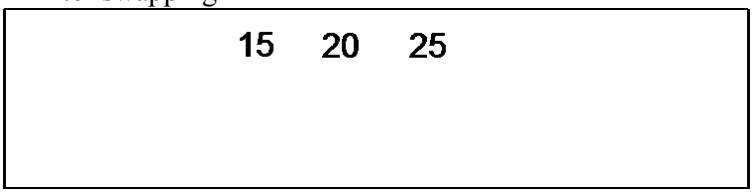

- Now 35 is in correct position.
- On left side and right side of 35 repeat the process. Consider on left side of 35



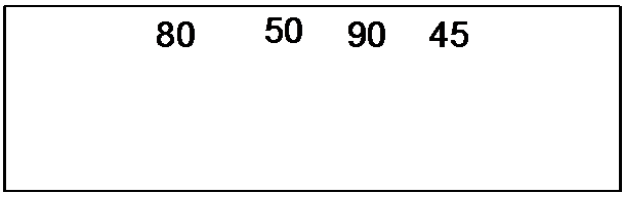

- Let 25 as pivot. The lesser of 25 that is 15 as j and there is no greater. So bring i after j i > j so swap pivot and j



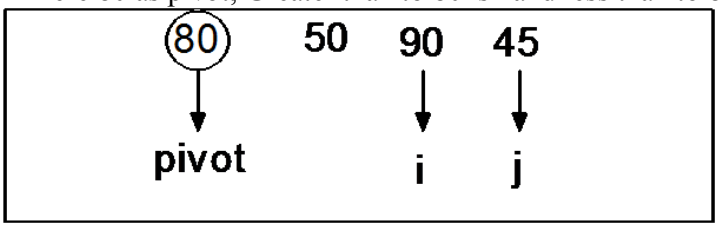

- After swapping



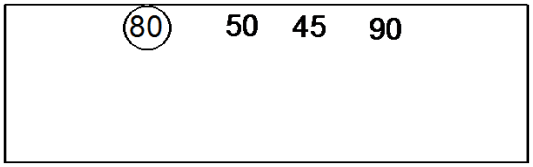

- Now the left part is sorted. Consider right part



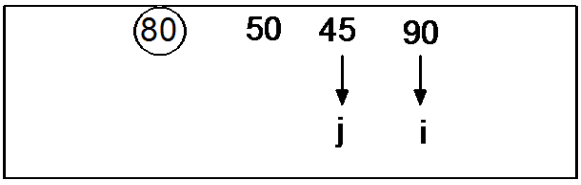

- Here 80 as pivot, Greater than to 80 is i and less than to 80 is j



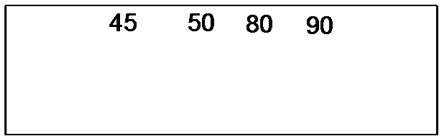

- Here i is less than j so swap i and j elements.



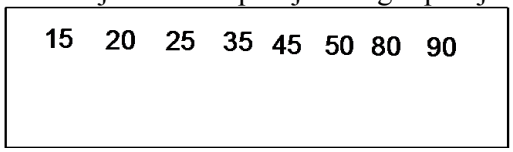

- first find greater to 80 is i and lesser to 80 is j. i > j so swap 80 and j.




- After swapping. The sorting elements are given by




- Now join all left part j and right part j to get the sorted elements




**Algorithm:**

**Step 1:** Let the first element taken as pivot
**Step 2:** Find lesser of pivot say i and greater of pivot say j.
**Step 3:** If i is less than j then i and j elements are swapped. Repeat step 2
**Step 4:** Repeat step 3 until i > j
Now swap j and pivot
**Step 5:** Now the pivot element is final position.
Repeat the above procedure for left and right side of pivot elements until all elements are sorted
**Step 6:** Stop

**Program:**

```
#include<stdio.h>
void quicksort(int a[25],int first,int last)
```

```
{
        int i, j, pivot, temp;
       if(first<last)
        {
                pivot=first;
                i=first;
                j=last;
                while(i<j)
                {
                        while(a[i]<a[pivot]&&i<=last)
                        i++;
                        while(a[j]>a[pivot])
                        j--;
                      if(i<j)
                        {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                        }
                }
                temp=a[pivot];
                a[pivot]=a[j];
                a[j]=temp;
                quicksort(a,first,j-1);
                quicksort(a,j+1,last);
        }
}
int main()
{
        int i, n, a[25];
        printf("Enter total a of elements:\n ");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        quicksort(a,0,n-1);
        printf("The Sorted elements are:\n ");
       for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

**(iv) Distribution Sort or Radix Sort:**

- Radix sort is one of the sorting algorithms used to sort a list of integer numbers in ascending or descending order.
- In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from least significant digit to the most significant digit
- Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers.
- For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.
- **Example:**

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

**Step 1 -** Define 10 queues each represents a bucket for digits from 0 to 9.


**Step 2 -** Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

**82, 901, 100, 12, 150, 77, 55 & 23**


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

**100, 150, 901, 82, 12, 23, 55 & 77**

**Step 3 -** Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

**100, 150, 901, 82, 12, 23, 55 & 77**


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

**100, 901, 12, 23, 150, 55, 77 & 82**

**Step 4 -** Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

**100, 901, 12, 23, 150, 55, 77 & 82**


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

**12, 23, 55, 77, 82, 100, 150, 901**

List got sorted in the incresing order.

**Algorithm**

Step 1 - Define 10 queues each representing a bucket for each digit from 0 to 9.
Step 2 - Consider the least significant digit of each number in the list which is to be sorted.
Step 3 - Insert each number into their respective queue based on the least significant digit.
Step 4 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
Step 5 - Repeat from step 3 based on the next least significant digit.
Step 6 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

**Program:**

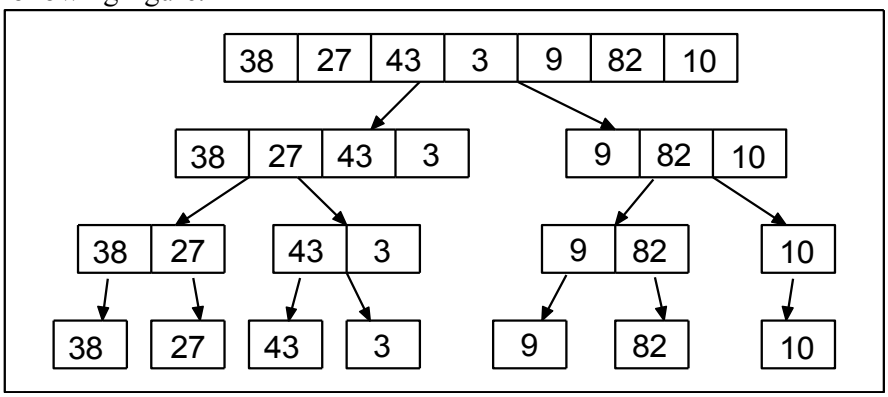
```
#include<stdio.h>
int getMax(int arr[], int n)
{
        int max = arr[0];
        int i;
        for (i = 1; i < n; i++)
        {
                if (arr[i] > max)
                max = arr[i];
        }
        return max;
}
void countSort(int arr[], int n, int exp)
{
        int output[n]; // output array
        int i, count[10] = { 0 };
        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
        for (i = 1; i < 10; i++)
        count[i] += count[i - 1];
        // Build the output array
        for (i = n - 1; i >= 0; i--)
        {
              output[count[(arr[i] / exp) % 10] - 1] = arr[i];
              count[(arr[i] / exp) % 10]--;
        }
        for (i = 0; i < n; i++)
        arr[i] = output[i];
}
// The main function to that sorts arr[] of size n using Radix Sort
void radixsort(int a[], int n)
{
       int max = getMax(a, n);
       int i;
       for (i = 1; max / i > 0; i *= 10)
       countSort(a, n, i);
}
int main()
{
       int a[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
       int i;
       int n = sizeof(a) / sizeof(a[0]);
       radixsort(a, n);
       for (i = 0; i < n; i++)
       printf("%d ", a[i]);
       return 0;    }
```

**(v) Merging or Merge Sort:**

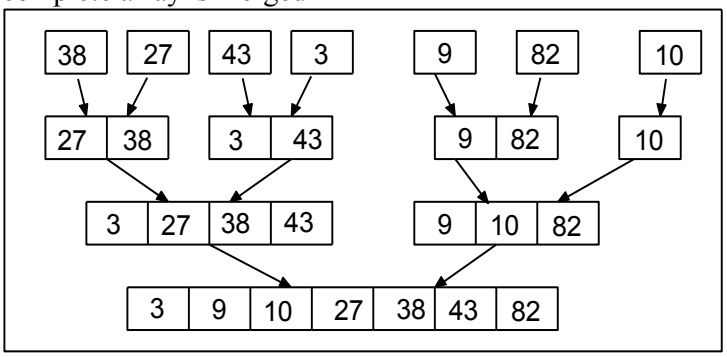
- It divides input array into two halves, calls itself for the two halves and then sorted and merged that two halves.

**Example:**

- For example consider the array of elements: 38, 27, 43, 3, 9, 82, 10
- Now the array is recursively divided into two halves till the size becomes one which is shown in the following figure.

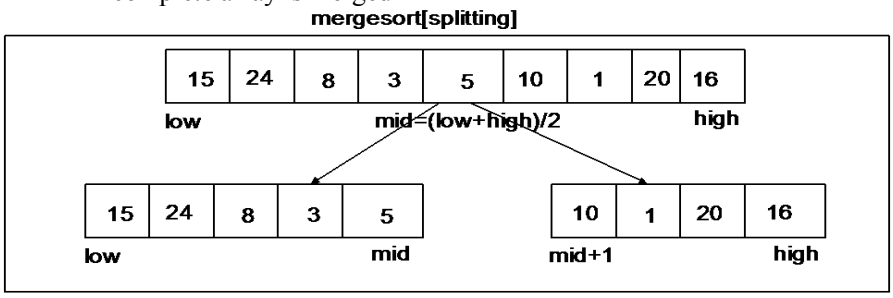


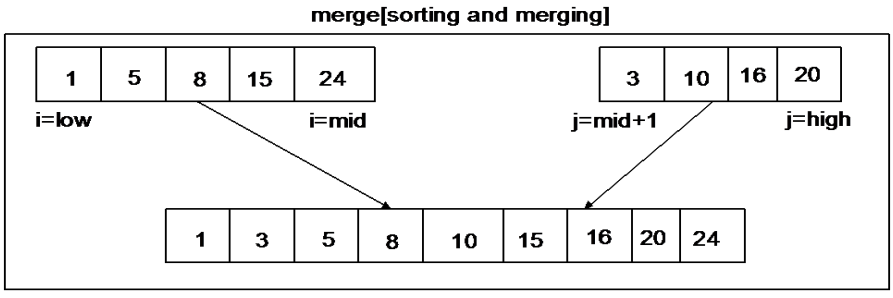

- Once the size becomes one, the merge process comes into action and starts merging with sorted array till the complete array is merged




**Algorithm:**

Step 1 − If it is only one element in the list then it is already sorted.
Step 2 − Divide the list recursively into two halves till the size becomes one.
Step 3 − Once the size becomes 1, the merge process comes into action and starts merging with sorted array till the complete array is merged



**Program:**

```
#include<stdio.h>
int n,a[30],i,j,k,temp[30];
void merge(int low,int mid,int high)
{
        i=low;
        j=mid+1;
        k=low;
        while((i<=mid) && (j<=high))
        {
                if(a[i]>=a[j])
                temp[k++]=a[j++];
                else
                temp[k++]=a[i++];
        }
        while(i<=mid)
        temp[k++]=a[i++];
        while(j<=high)
        temp[k++]=a[j++];
        for(i=low;i<=high;i++)
        a[i]=temp[i];
}
 void mergesort(int low,int high)
 {
        int mid;
        if(low!=high)
        {
                mid=((low+high)/2);
                mergesort(low,mid);
                mergesort(mid+1,high);
                merge(low,mid,high);
       }
}
int main()
{
        printf("Enter total elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        mergesort(0,n-1);
        printf("After sorting is:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

**Time Complexity:**

- **O(n^2)** means that for every insert, it takes n*n operations. i.e. 1 operation for 1 item, 4 operations for 2 items, 9 operations for 3 items.

**Comparison of Sorting Algorithms**

| Algorithm | Data Structure | Time Complexity | | |
|---|---|---|---|---|
| | | Best | Average | Worst |
| Quicksort | Array | O(n log(n)) | O(n log(n)) | O(n^2) |
| Mergesort | Array | O(n log(n)) | O(n log(n)) | O(n log(n)) |
| Bubble Sort | Array | O(n) | O(n^2) | O(n^2) |
| Insertion Sort | Array | O(n) | O(n^2) | O(n^2) |
| Select Sort | Array | O(n^2) | O(n^2) | O(n^2) |

**Complexity of Radix Sort:**

- Radix sort is a non-comparative algorithm, it has advantages over comparative sorting algorithms.
- For the radix sort that uses counting sort as an intermediate stable sort, the time complexity is O(d(n+k)).
- Here, d is the number cycle and O(n+k) is the time complexity of counting sort
- Thus, radix sort has linear time
- complexity which is better than
- O(nlog n) of comparative sorting algorithms.
- If we take very large digit numbers or the number of other bases like 32-bit and
- 64-bit numbers then it can perform in linear time however the intermediate sort takes large space.
- This makes radix sort space inefficient.
- This is the reason why this sort is not used in software libraries.

**Best case, Worst case and Average Case of Radix sort -**

- Radix sort complexity is O(kn) for n keys which are integers of word size k.
- For all there cases time i.e best , worst and average time complexity is O(kn).

# 2. LINKED LISTS

**UNIT II**

Linked List: Introduction, Single linked list, Representation of Linked list in memory, Operations on Single Linked list-Insertion, Deletion, Search and Traversal ,Reversing Single Linked list, Applications on Single Linked list- Polynomial Expression Representation ,Addition and Multiplication, Sparse Matrix Representation using Linked List, Advantages and Disadvantages of Single Linked list, Double Linked list-Insertion, Deletion, Circular Linked list-Insertion, Deletion.

## 2.1 Introduction

A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.

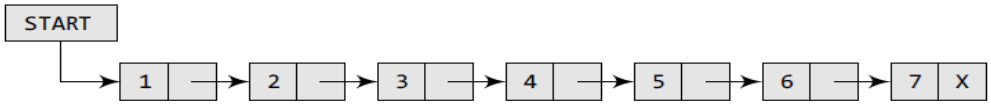
The elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes. Linked list is a data structure which in turn can be used to implement other data structures.

Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.

**Basic Terminology:**

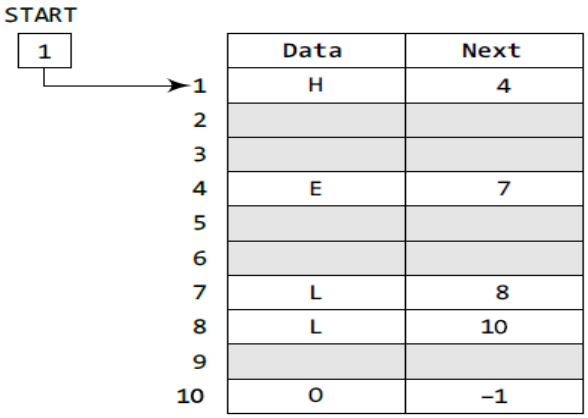
A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.




We can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The last node will have no next node connected to it, so it will store a special value called NULL.

Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.

Let us see how a linked list is maintained in the memory. When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.




**Linked Lists versus Arrays:**

Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations. Another point of difference between an array and a linked list is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner.

Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array.

**Memory Allocation and De-allocation for a Linked List:**

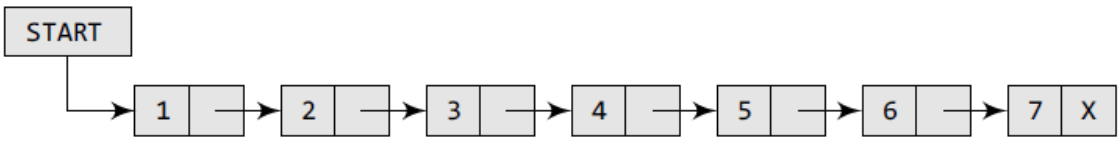
If we want to add a node to an already existing linked list in the memory, we first find free space in the memory and then use it to store the information.

Now, the question is which part of the memory is available and which part is occupied? When we delete a node from a linked list, then who changes the status of the memory occupied by it from occupied to available? The answer is the operating system.

The operating system scans through all the memory cells and marks those cells that are being used by some program. Then it collects all the cells which are not being used and adds their address to the free pool, so that these cells can be reused by other programs. This process is called garbage collection.

## 2.2 Single Linked Lists:

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

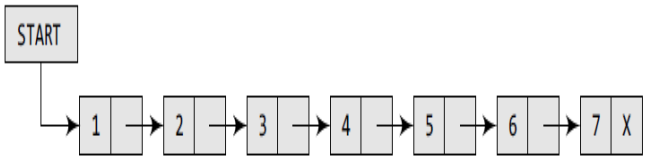



Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node.

## 2.2.1 Operations

**Traversing a Linked List:**

For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. Algorithm for traversing a linked list

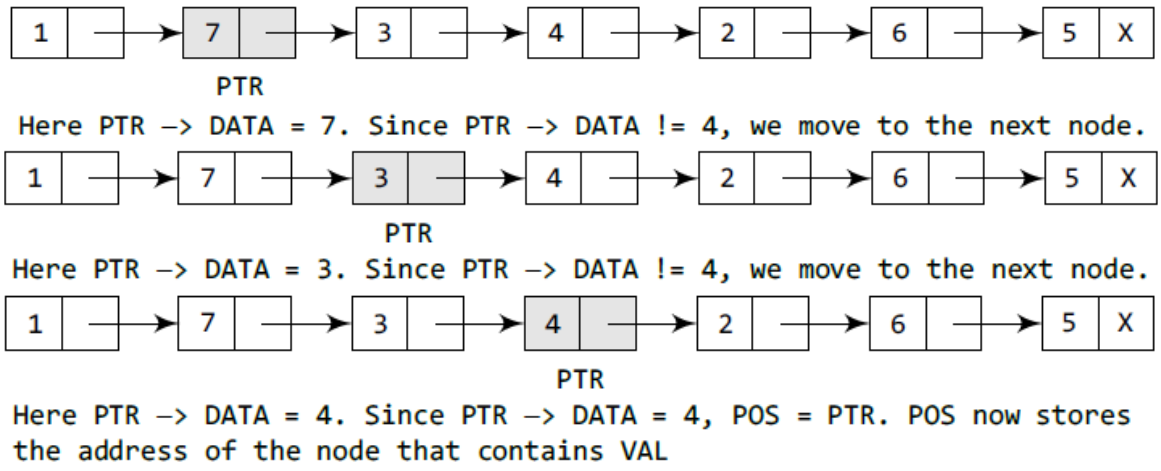



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:          Apply Process to PTR->DATA
Step 4:          SET PTR = PTR->NEXT
      [END OF LOOP]
Step 5: EXIT
```

**Searching for a Value in a Linked List:**

Searching a linked list means to find a particular element in the linked list. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Consider the linked list shown in below. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:     IF VAL = PTR -> DATA
                SET POS = PTR
                Go To Step 5
            ELSE
                SET PTR = PTR -> NEXT
            [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

**Inserting a New Node in a Linked List:**

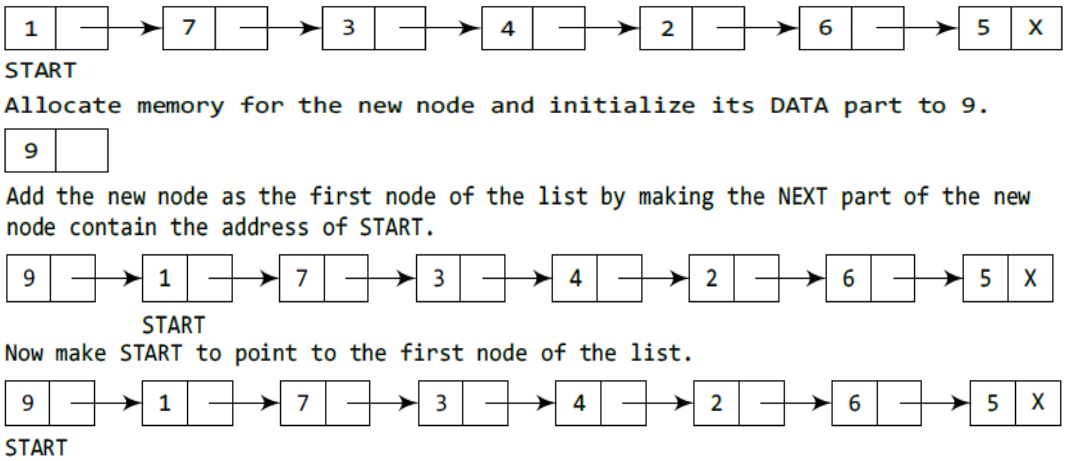
we will see how a new node is added into an already existing linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.

Let us first discuss an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

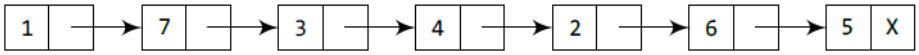
**Case 1: Inserting a Node at the Beginning of a Linked List**

Inserting a Node at the Beginning of a Linked List. Consider the linked list shown in below figure. Suppose we want to add a new node with data 9 and add it as the first node of the list.



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

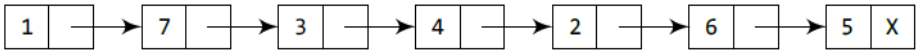
**Case 2: Inserting a Node at the End of a Linked List**




START

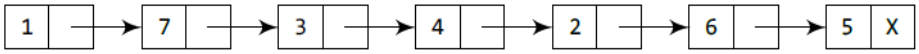
Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.


Take a pointer variable PTR which points to START.



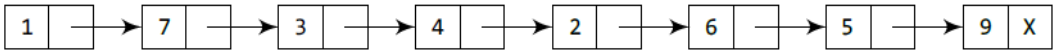

START, PTR

Move PTR so that it points to the last node of the list.




START PTR

Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

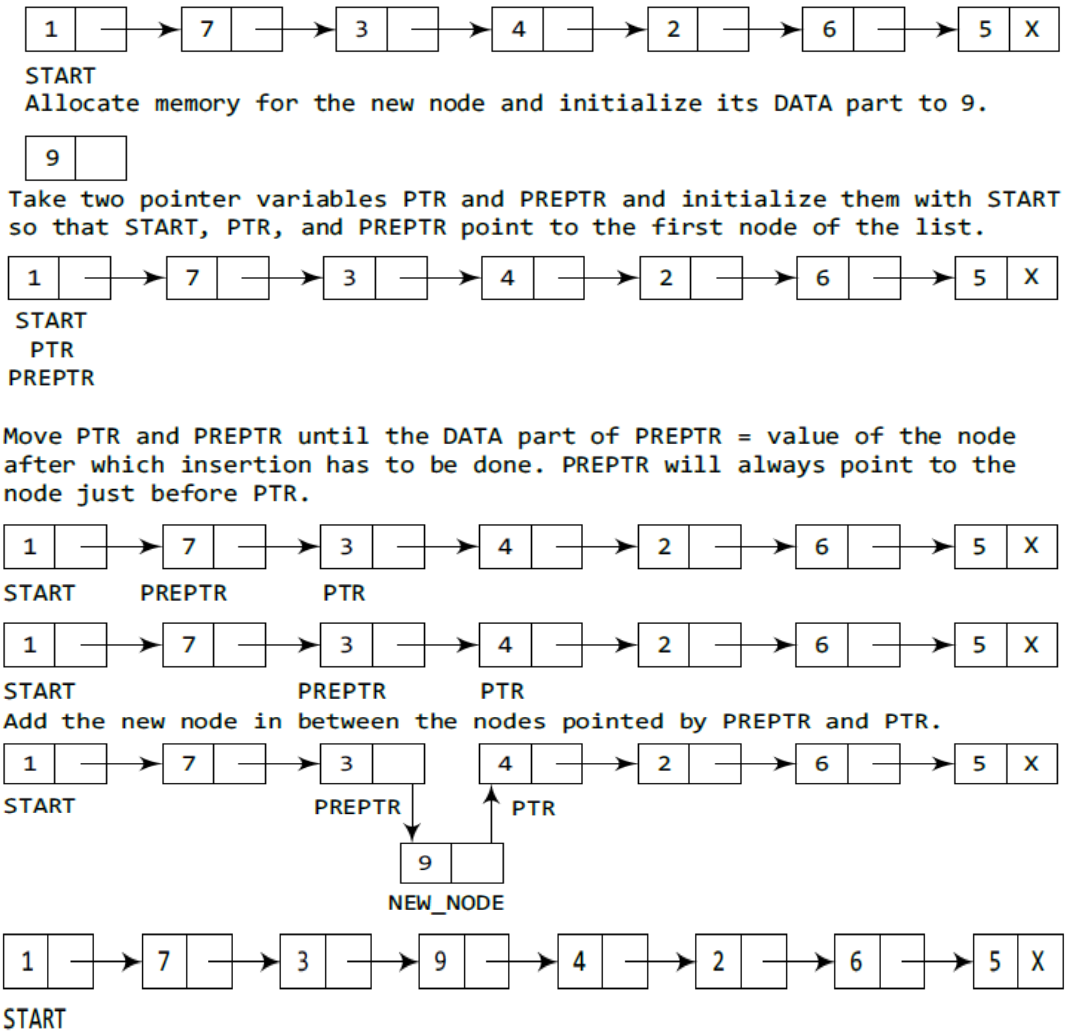



START PTR

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

**Case 3: Inserting a Node After a Given Node in a Linked List**

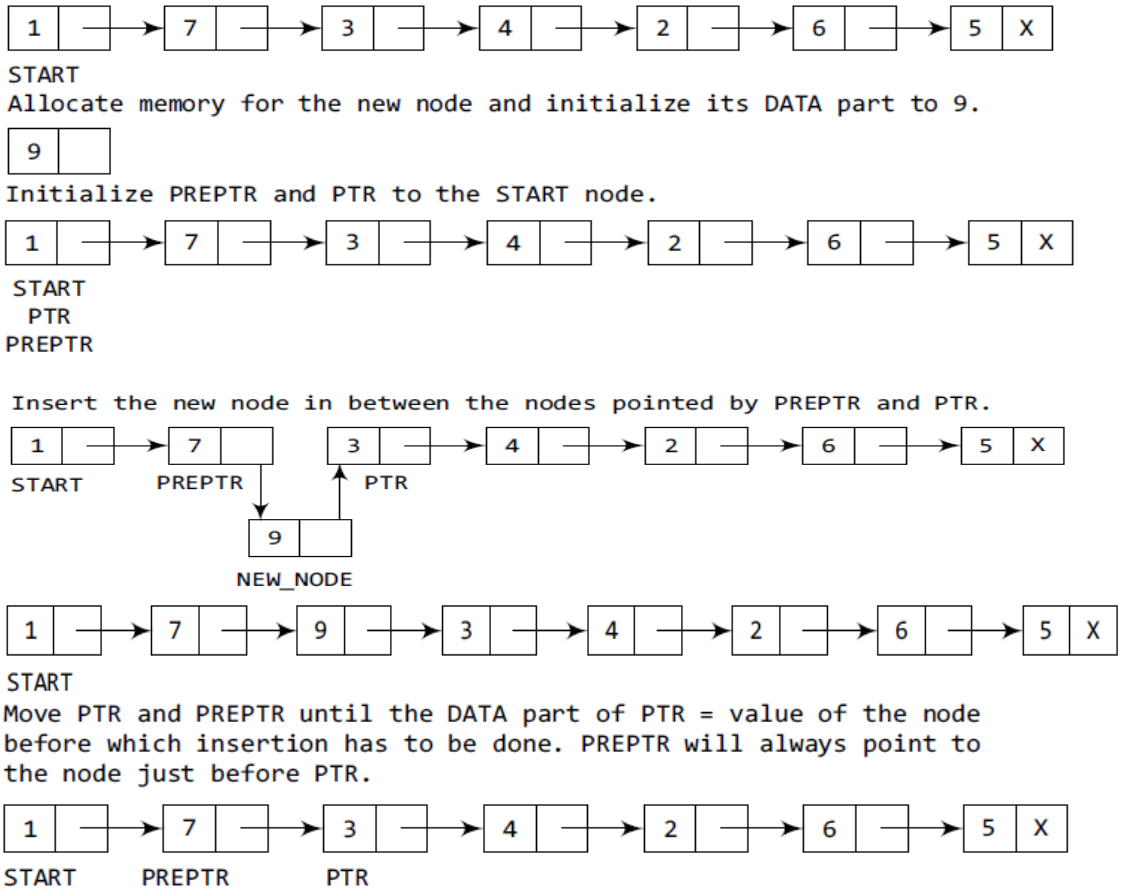
Consider the linked list shown in below figure. Suppose we want to add a new node with value 9 after the node containing 3.




```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

**Case 4: Inserting a Node Before a Given Node in a Linked List**

Consider the linked list shown in below figure. Suppose we want to add a new node with value 9 before the node containing 3.




```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR -> DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

**Deleting a Node from a Linked List:**

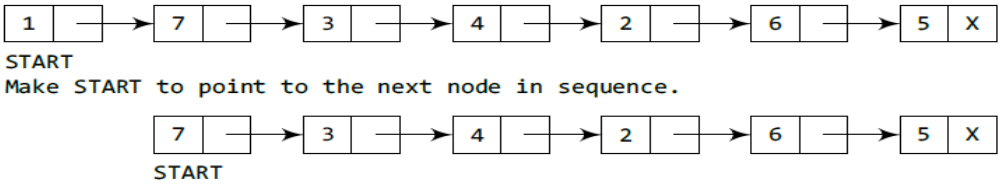
We will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

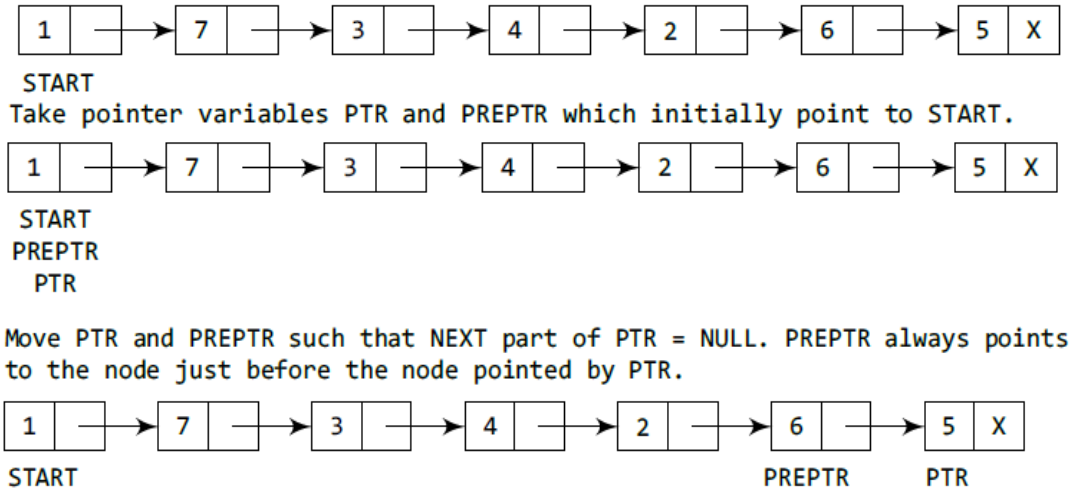
Case 3: The node after a given node is deleted.

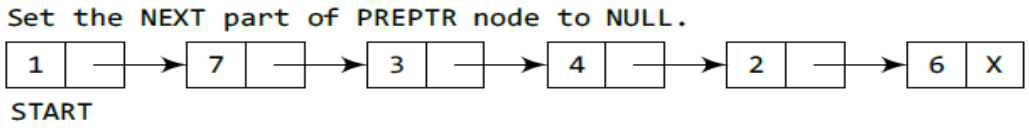
**Case 1: Deleting a First Node from a Linked List**




```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

**Case 2: Deleting the Last Node from a Linked List**

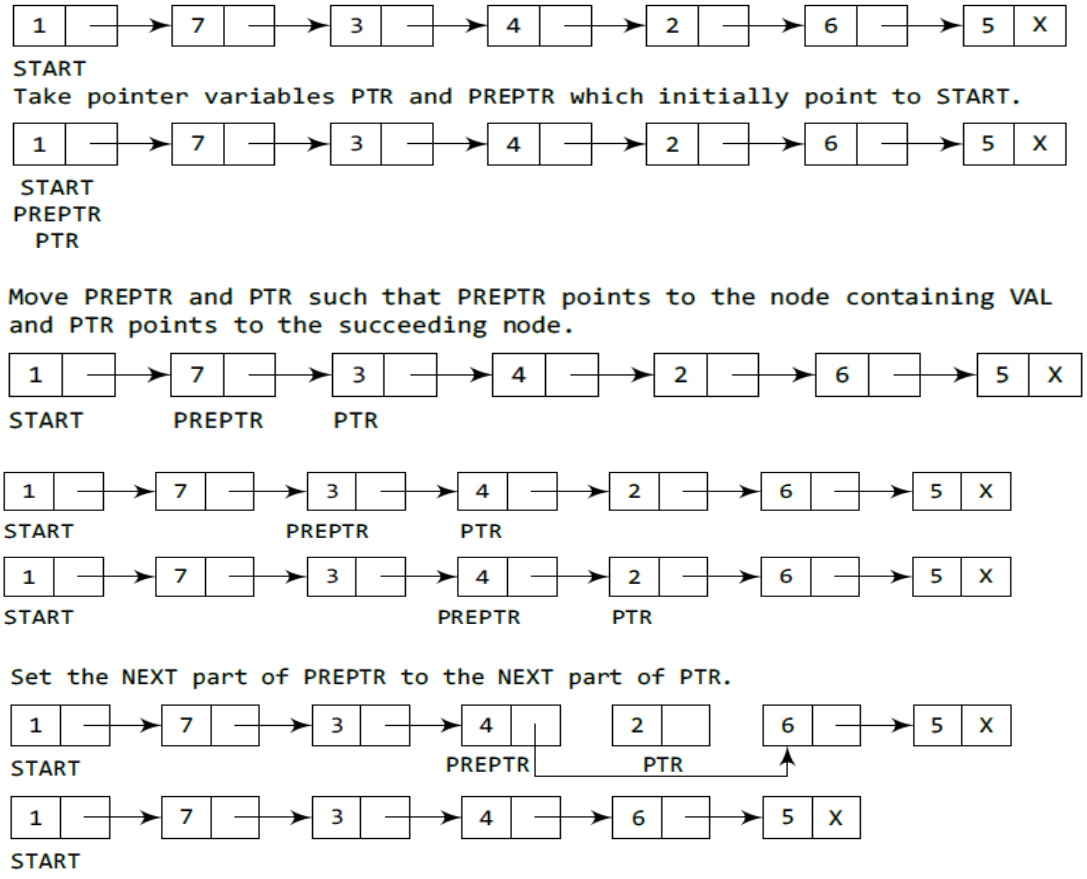





```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

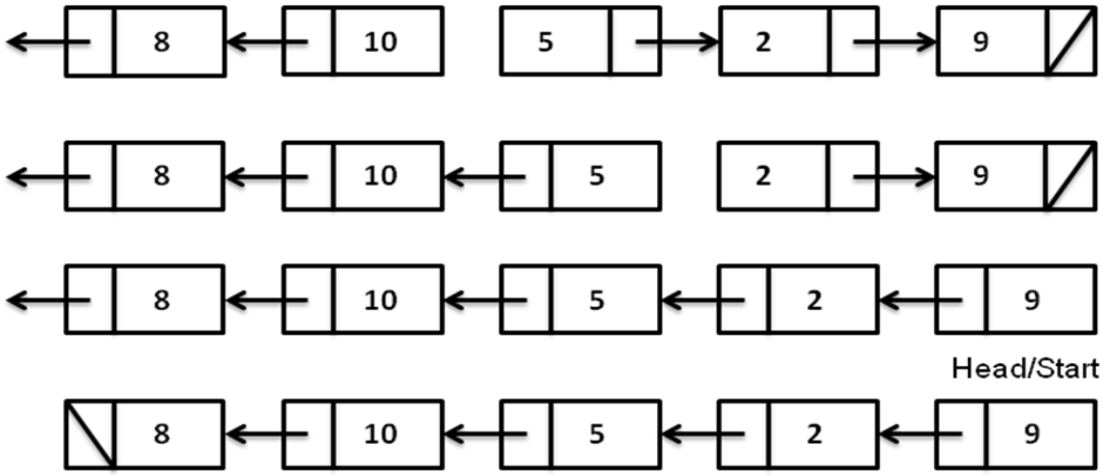
**Case 3: Deleting After a Given Node in a Linked List**

Consider the linked list shown in below figure. Suppose we want to delete the node that succeeds the node which contains data value 4.



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

## 2.3 Reversing Single Linked list:




| | |
|---|---|
| struct node *prevNode, *curNode;<br>if(head != NULL)<br>{ prevNode = head;<br>curNode = head->next;<br>head = head->next;<br>prevNode->next = NULL;<br>// Make first node as last node | while(head != NULL)<br>{ head = head->next;<br>curNode->next = prevNode;<br>prevNode = curNode;<br>curNode = head;<br>} head = prevNode;<br>// Make last node as head<br>} |

## 2. 4 Applications on Single Linked list

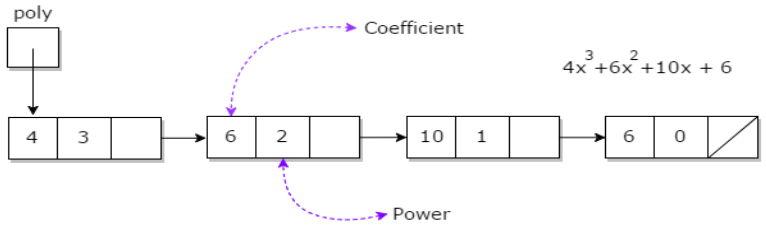
- ✓ Implementation of stacks and queues.
- ✓ Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- ✓ Dynamic memory allocation: We use linked list of free blocks.
- ✓ Maintaining directory of names. Performing arithmetic operations on long integers
- ✓ Manipulation of polynomials by storing constants in the node of linked list. Representing sparse matrices

**Polynomial Expression Representation:**

A polynomial is composed of different terms where each of them holds a coefficient and an exponent. An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts: one is the coefficient and other is the exponent

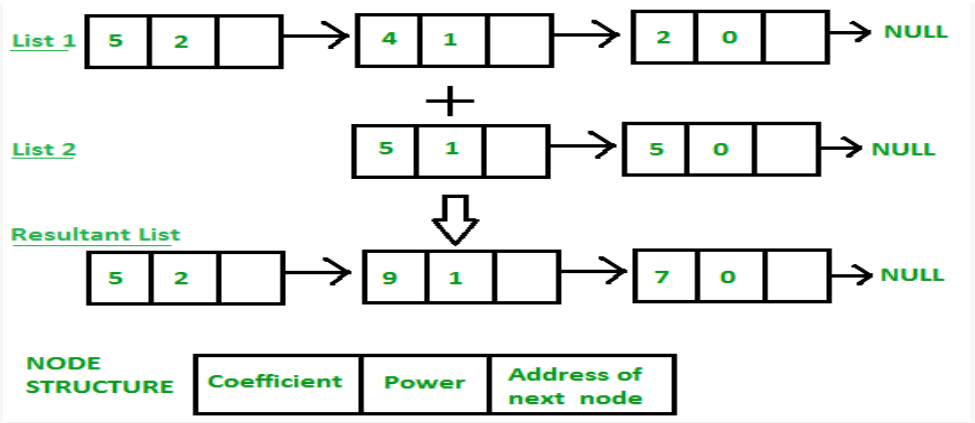
Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 are its exponential values.



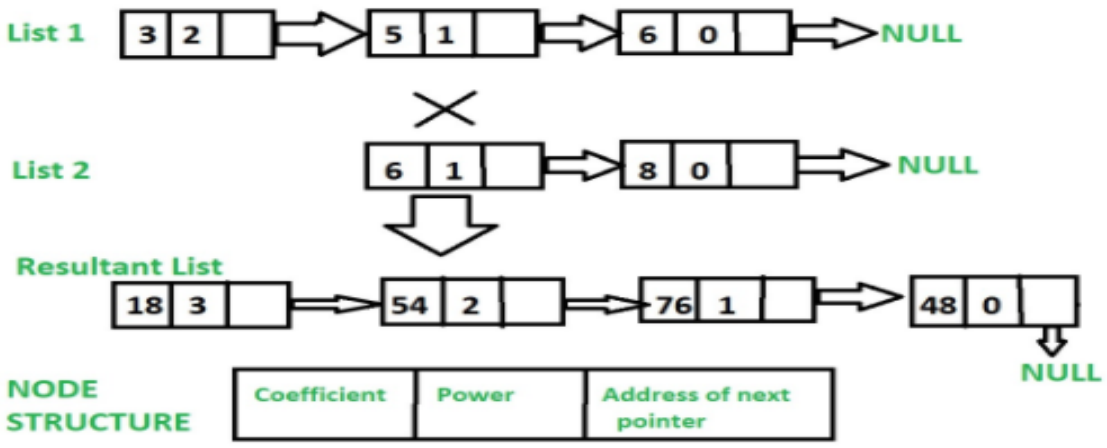

**Addition of two polynomials:**

```
Input:
      1st number = 5x^2 + 4x^1 + 2x^0
      2nd number = 5x^1 + 5x^0
Output:
          5x^2 + 9x^1 + 7x^0
Input:
      1st number = 5x^3 + 4x^2 + 2x^0
      2nd number = 5x^1 + 5x^0
Output:
          5x^3 + 4x^2 + 5x^1 + 7x^0
```



**Multiplication of two polynomials:**

```
Input: Poly1: 3x^2 + 5x^1 + 6, Poly2: 6x^1 + 8
Output: 18x^3 + 54x^2 + 76x^1 + 48
On multiplying each element of 1st polynomial with
elements of 2nd polynomial, we get
18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48
On adding values with same power of x,
18x^3 + 54x^2 + 76x^1 + 48

Input: Poly1: 3x^3 + 6x^1 + 9, Poly2: 9x^3 + 8x^2 + 7x^1 + 2
Output: 27x^6 + 24x^5 + 75x^4 + 135x^3 + 114x^2 + 75x^1 + 18
```




**Sparse Matrix Representation using Linked List:**

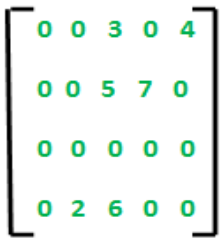
A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Example:
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

**Sparse Matrix Representation using arrays:**

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as
Row: Index of row, where non-zero element is located
Column: Index of column, where non-zero element is located
Value: Value of the non zero element located at index – (row, column)




| Row | 0 | 0 | 1 | 1 | 3 | 3 |
|---|---|---|---|---|---|---|
| Column | 2 | 4 | 2 | 3 | 1 | 2 |
| Value | 3 | 4 | 5 | 7 | 2 | 6 |

**Sparse Matrix Representation using Linked List:**

Method 2: Using Linked Lists

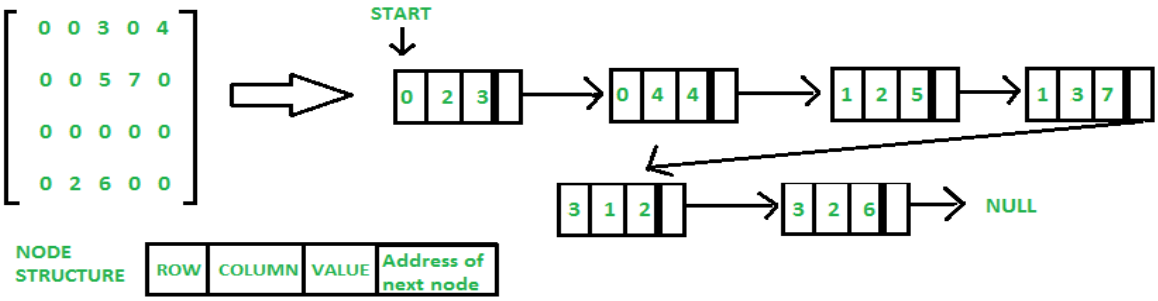
In linked list, each node has four fields. These four fields are defined as:

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row, column)

Next node: Address of the next node




**Advantages of Single Linked list:**

- Insertions and Deletions can be done easily.
- It does not need movement of elements for insertion and deletion.
- Space is not wasted as we can get space according to our requirements.
- Its size is not fixed. It can be extended or reduced according to requirements.
- Elements may or may not be stored in consecutive memory available, even then we can store the data in computer.
- It is less expensive.

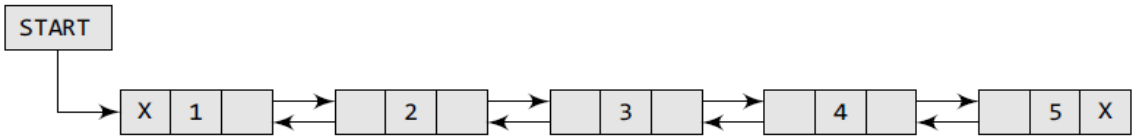
**Disadvantages of Single Linked list:**

- It requires more space as pointers are also stored with information.
- Different amount of time is required to access each element.
- If we have to go to a particular element then we have to go through all those elements that come before that element.
- We cannot traverse it from last & only from the beginning.
- It is not easy to sort the elements stored in the linear linked list.

## 2.5 Doubly linked list

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.

Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.




A doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).

The main advantage of using a doubly linked list is that it makes searching twice as efficient. Let us view how a doubly linked list is maintained in the memory.

START = 1

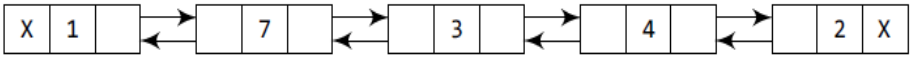
| | DATA | PREV | NEXT |
|---|---|---|---|
| 1 | H | −1 | 3 |
| 2 | | | |
| 3 | E | 1 | 6 |
| 4 | | | |
| 5 | | | |
| 6 | L | 3 | 7 |
| 7 | L | 6 | 9 |
| 8 | | | |
| 9 | O | 7 | −1 |

## Inserting a New Node in a Doubly Linked List:

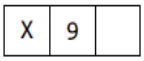
In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take four cases and then see how insertion is done in each case.

Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.

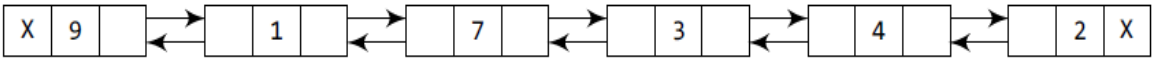
**Case 1: Inserting a Node at the Beginning of a Doubly Linked List**




START
Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.



Add the new node before the START node. Now the new node becomes the first node of the list.



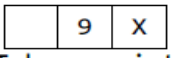

START

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

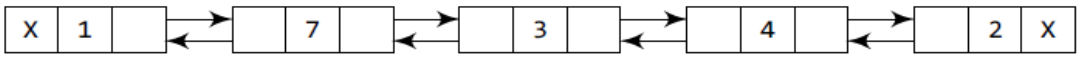
**Case 2: Inserting a Node at the end of a Doubly Linked List**


START

Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

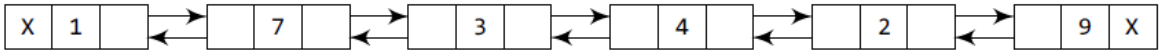



Take a pointer variable PTR and make it point to the first node of the list.




START,PTR

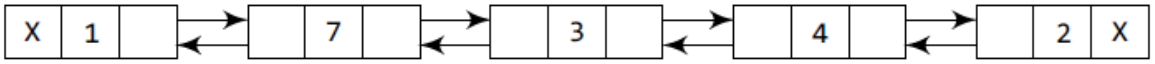
Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.




START PTR

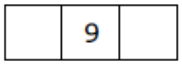
```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

**Case 3: Inserting a Node After a Given Node in a Doubly Linked List**

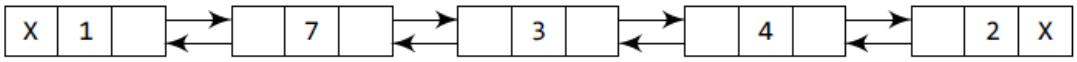



START

Allocate memory for the new node and initialize its DATA part to 9.

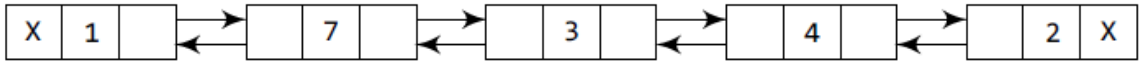



Take a pointer variable PTR and make it point to the first node of the list.

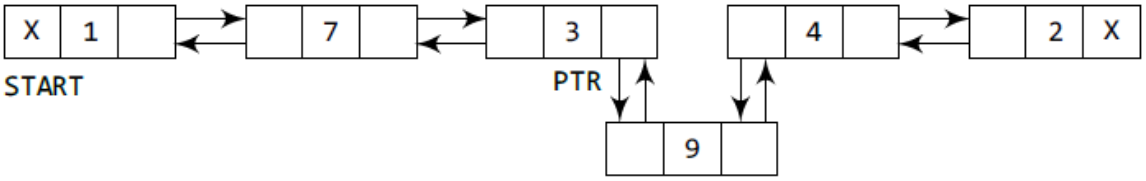


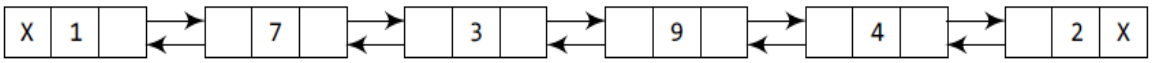

START,PTR

Move PTR further until the data part of PTR = value after which the node has to be inserted.




Insert the new node between PTR and the node succeeding it.

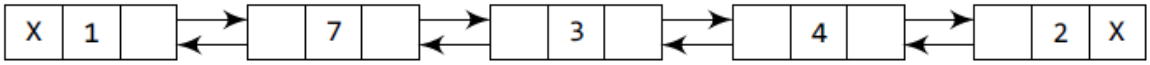



START

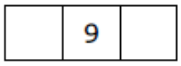
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

**Case 4: Inserting a Node Before a Given Node in a Doubly Linked List**




START

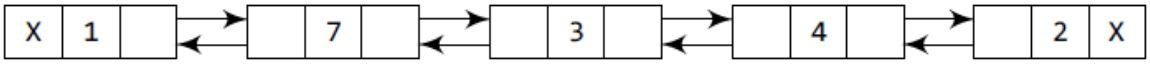
Allocate memory for the new node and initialize its DATA part to 9.




Take a pointer variable PTR and make it point to the first node of the list.

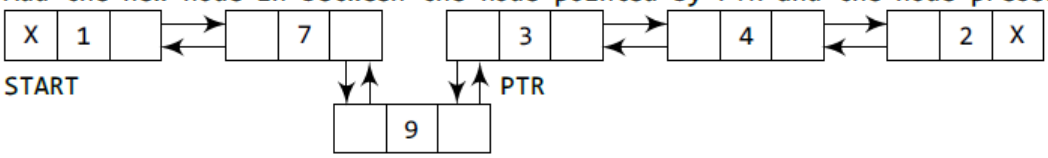

START, PTR

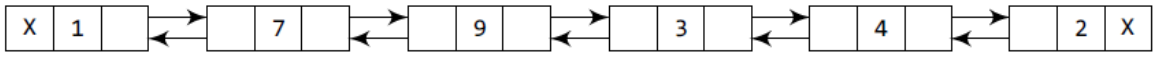
Move PTR further so that it now points to the node whose data is equal to the value before which the node has to be inserted.




START

Add the new node in between the node pointed by PTR and the node preceding it.




START

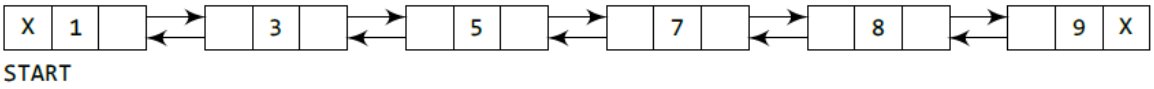
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR  -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

**Deleting a Node from a Doubly Linked List**

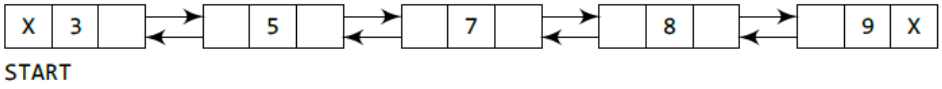
In this section, we will see how a node is deleted from an already existing doubly linked list. We will take four cases and then see how deletion is done in each case.

Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.
Case 4: The node before a given node is deleted.

**Case 1: Deleting the First Node from a Doubly Linked List**

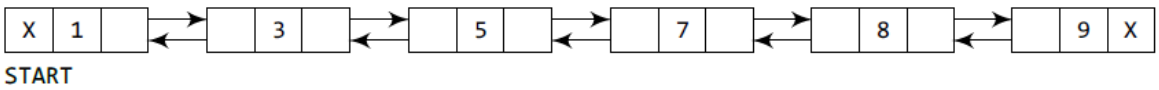



Free the memory occupied by the first node of the list and make the second node of the list as the START node.

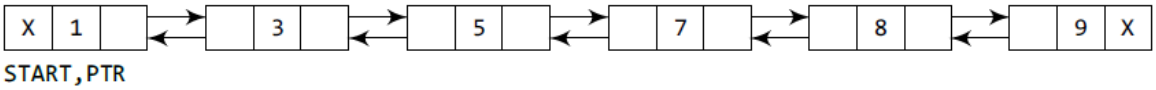



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

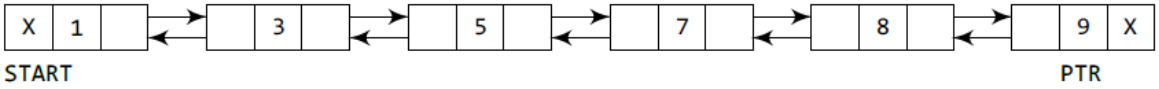
**Case 2: Deleting the Last Node from a Doubly Linked List**



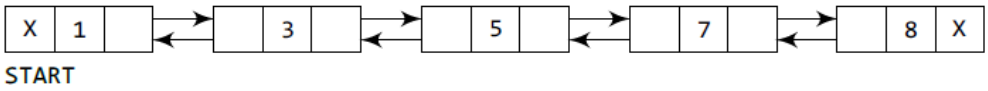

Take a pointer variable PTR that points to the first node of the list.




Move PTR so that it now points to the last node of the list.

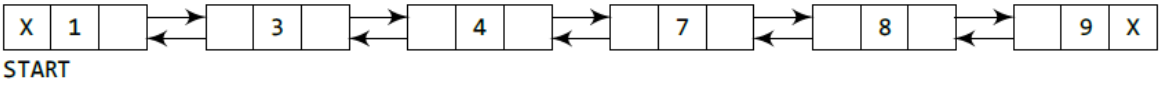


Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.

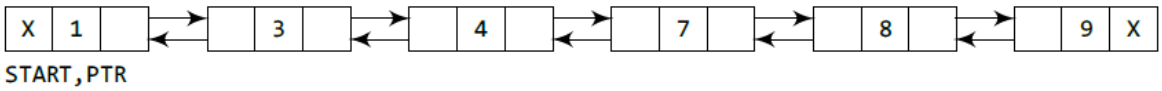



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

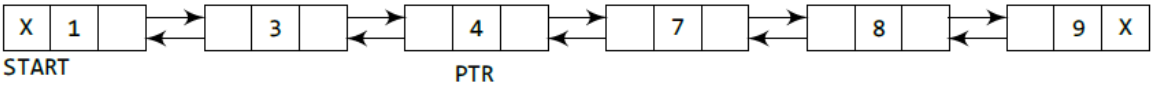
**Case 3: Deleting the Node After a Given Node in a Doubly Linked List**



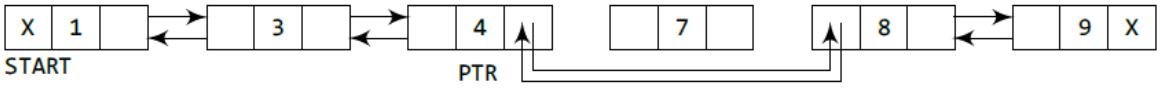

Take a pointer variable PTR and make it point to the first node of the list.

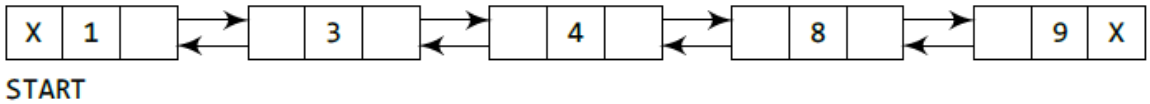



Move PTR further so that its data part is equal to the value after which the node has to be inserted.



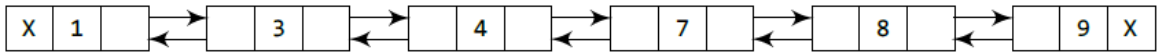

Delete the node succeeding PTR.



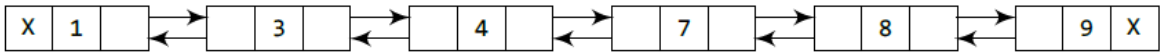
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

**Case 4: Deleting the Node Before a Given Node in a Doubly Linked List**



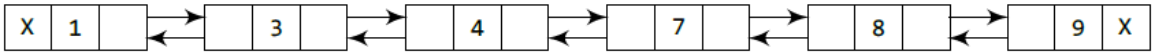

START

Take a pointer variable PTR that points to the first node of the list.



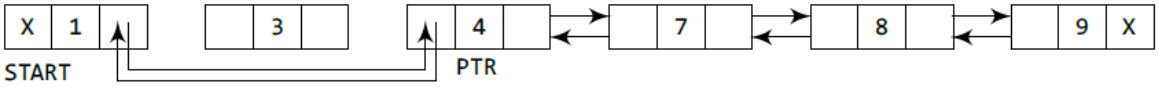

START,PTR

Move PTR further till its data part is equal to the value before which the node has to be deleted.

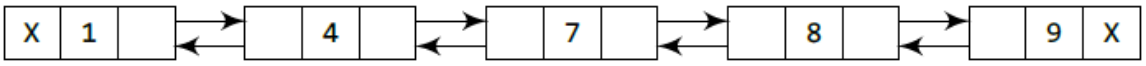



START PTR

Delete the node preceding PTR.




START PTR




START

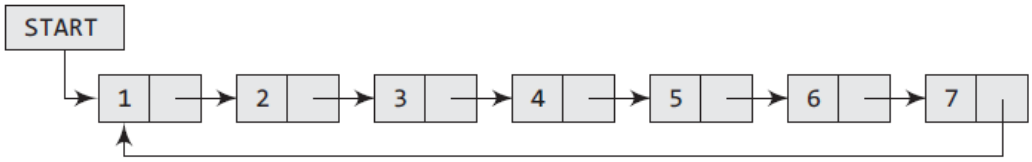
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

## 2.6 Circular Linked List

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.

Note that there are no NULL values in the NEXT part of any of the nodes of list.



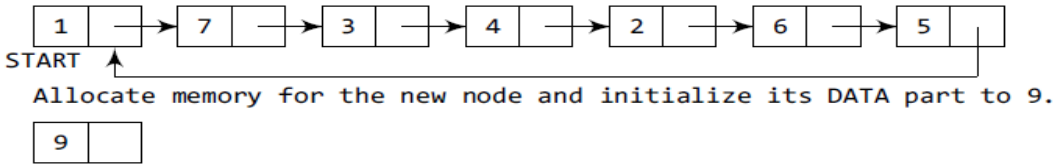

## Operation:

## Inserting a New Node in a Circular Linked List

In this section, we will see how a new node is added into an already existing linked list. We will take two cases and then see how insertion is done in each case.

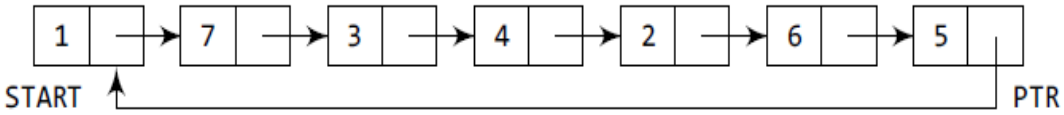
Case 1: The new node is inserted at the beginning of the circular linked list.
Case 2: The new node is inserted at the end of the circular linked list.

**Case 1: Inserting a Node at the Beginning of a Circular Linked List**

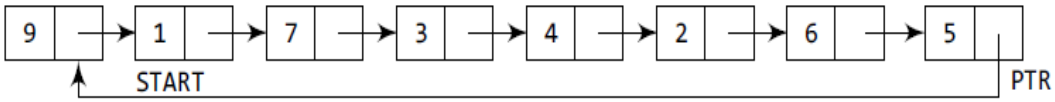


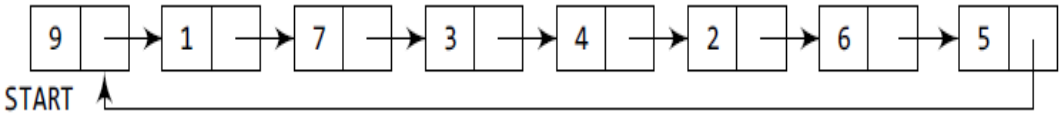
Take a pointer variable PTR that points to the START node of the list.


Move PTR so that it now points to the last node of the list.



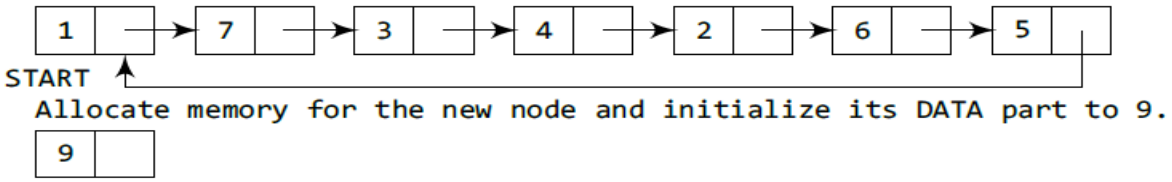

Add the new node in between PTR and START.



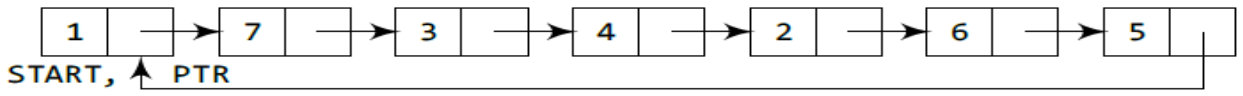




```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

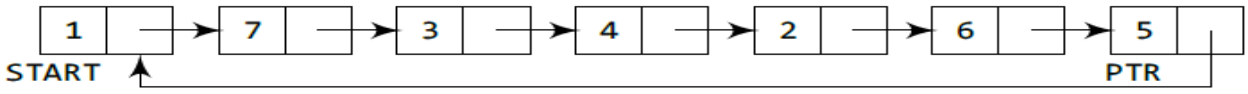
**Case 2: Inserting a Node at the End of a Circular Linked List**



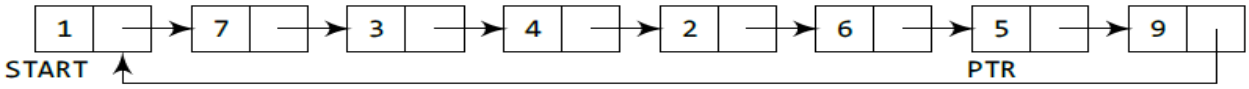
Take a pointer variable PTR which will initially point to START.




Move PTR so that it now points to the last node of the list.




Add the new node after the node pointed by PTR.



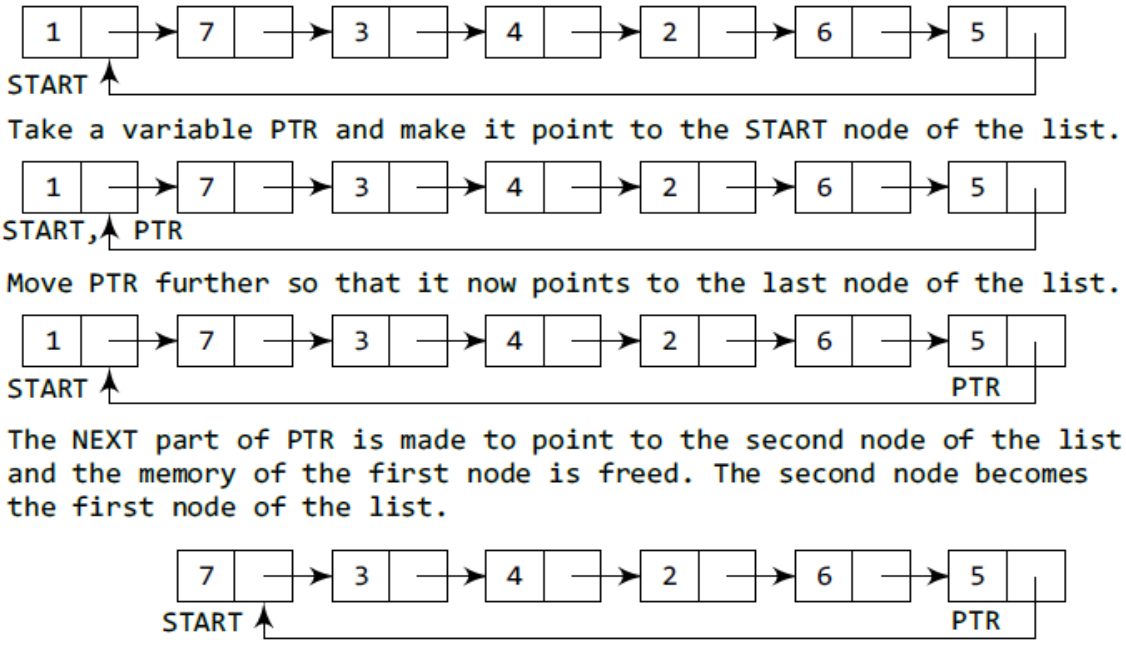

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

## Deleting a Node from a Circular Linked List

In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion are same as that given for singly linked lists.

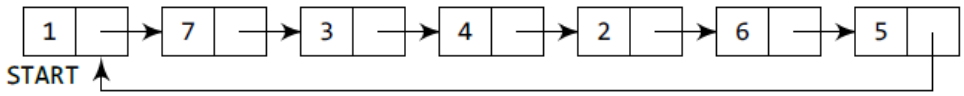
Case 1: The first node is deleted.
Case 2: The last node is deleted.

**Case 1: Deleting the First Node from a Circular Linked List**

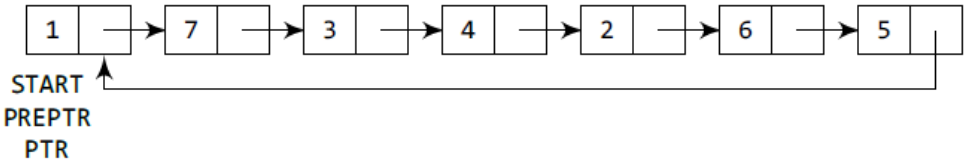



```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->NEXT != START
Step 4:       SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: SET PTR->NEXT = START->NEXT
Step 6: FREE START
Step 7: SET START = PTR->NEXT
Step 8: EXIT
```

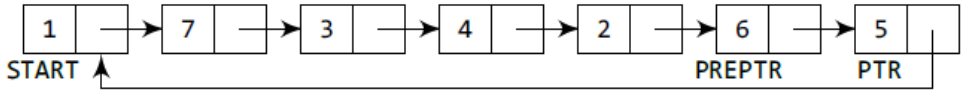
**Case 2: Deleting the Last Node from a Circular Linked List**



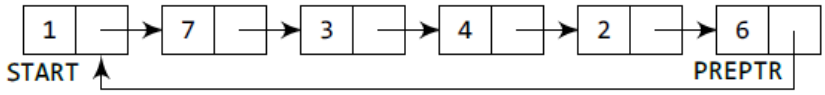

Take two pointers PREPTR and PTR which will initially point to START.




Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.




Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.




```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR->NEXT != START
Step 4:       SET PREPTR = PTR
Step 5:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 6: SET PREPTR->NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

# 3.QUEUES & STACKS

**UNIT III**

**Queues:** Introduction to Queues, Representation of Queues-using Arrays and using Linked list, Implementation of Queues-using Arrays and using Linked list, Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.
**Stacks:** Introduction to Stacks, Array Representation of Stacks, Operations on Stacks, Linked list Representation of Stacks, Operations on Linked Stack, Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

# QUEUES

## 3.1 INTRODUCTION TO QUEUES

Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure. A queue is a **FIFO (First-In, First-Out)** data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT. Queues can be implemented by using either arrays or linked lists. In this section, we will see how queues are implemented using each of these data structures.

## 3.2 ARRAY REPRESENTATION & IMPLEMENTATION OF QUEUES

Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.
The array representation of a queue is shown in Fig. 3.1.

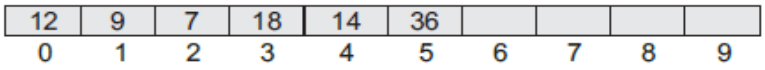



**Figure 3.1** Queue

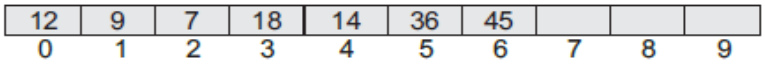



**Figure 3.2** Queue after insertion of a new element

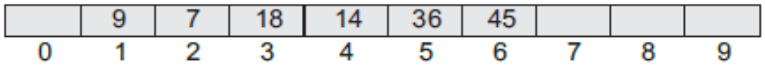



**Figure 3.2** Queue after deletion of an element

**Operations on Queues**

In Fig. 3.1, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.

The queue after addition would be as shown in Fig. 3.2. Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown in Fig. 3.3.
Here, FRONT = 1 and REAR = 6.

However, before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition. Note that we have written MAX – 1 because the index starts from 0. Similarly, before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue.

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to insert an element in a queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Algorithm to delete an element from a queue

NOTE: The process of inserting an element in the queue is called enqueue, and the process of deleting an element from the queue is called dequeue.

## 3.3 LINKED REPRESENTATION & IMPLEMENTATION OF QUEUES

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted.

And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.

In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).

In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.

All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty. The linked representation of a queue is shown in Fig. 3.4.

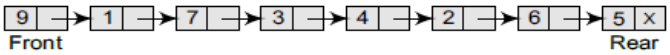



**Figure 3.4** Linked queue

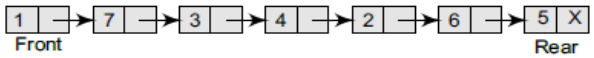



**Figure 3.5** Linked queue

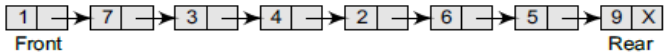



**Figure 3.6** Linked queue after inserting a new node

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Algorithm to insert an element in a linked queue

**Insert Operation**

The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 3.5.

To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear. However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear. Thus, the updated queue becomes as shown in Fig. 3.6.

The algorithm shows that inserting an element in a linked queue. In Step 1, the memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

**Delete Operation**

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown in Fig. 3.7.

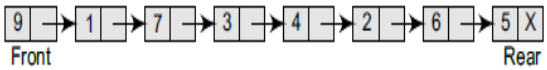



**Figure 3.7** Linked queue

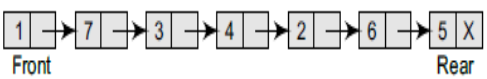



**Figure 3.8** Linked queue after deletion of an element

```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

## 3.4 APPLICATIONS OF QUEUES

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

## 3.5 TYPES OF QUEUES

A queue data structure can be classified into the following types:
1. Circular Queue 2. Deque 3. Priority Queue 4. Multiple Queue
We will discuss each of these queues in detail in the following sections.

### 3.5.1 Circular Queues

In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT. Look at the queue shown in Fig. 3.9.

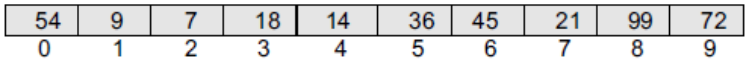



**Figure 3.9** Linear queue

Here, FRONT = 0 and REAR = 9.

Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made. The queue will then be given as shown in Fig. 3.10.

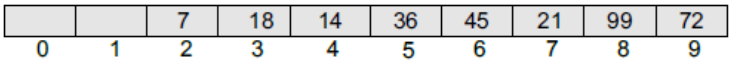



**Figure 3.10** Queue after two successive deletions

Here, front = 2 and REAR = 9.

Suppose we want to insert a new element in the queue shown in Fig. 3.10. Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true. This is a major drawback of a linear queue.

To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.

The second option is to use a circular queue. In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in Fig. 3.11.

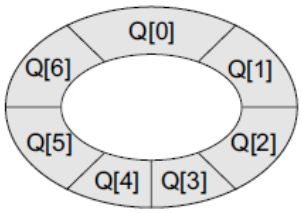



**Figure 3.11** Circular queue

The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations.

**For insertion,** we now have to check for the following three conditions:

- If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 3.12 which illustrates this point.
- If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in Fig. 3.13.
- If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown in Fig. 3.14.

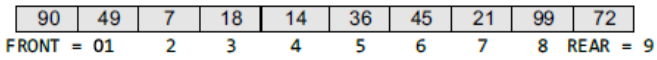



**Figure 3.12** Full queue

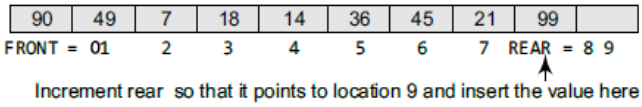



**Figure 3.13** Queue with vacant locations

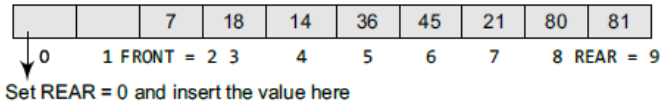



**Figure 3.14** Inserting an element in a circular queue

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

Let us look at the algorithm to insert an element in a circular queue. In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end. In Step 3, the value is stored in the queue at the location pointed by REAR.

Let us now discuss how **deletions** are performed in this case. To delete an element, again we check for three conditions.

- Look at Fig. 3.15. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is illustrated in Fig. 3.16.
- If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0. This is shown in Fig. 3.17.

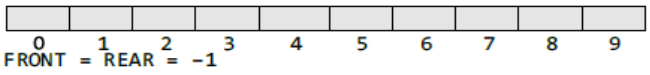



**Figure 3.15** Empty queue

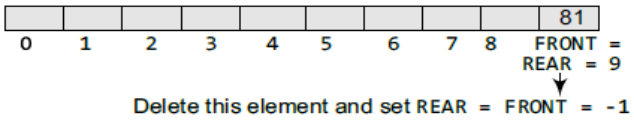



**Figure 3.16** Queue with a single element

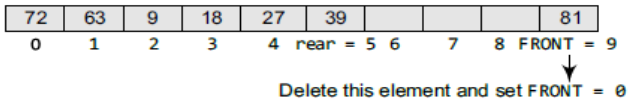



**Figure 3.17** Queue where FRONT = MAX–1 before deletion

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
         ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

Let us look at the algorithm to delete an element from a circular queue. In Step 1, we check for the underflow condition. In Step 2, the value of the queue at the location pointed by FRONT is stored in VAL. In Step 3, we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.

### 3.5.2 Deques

A deque (pronounced as 'deck' or 'dequeue') is a list in which the elements can be inserted or deleted at either end. It is also known as a head-tail linked list because elements can be added to or removed from either the front (head) or the back (tail) end.

However, no element can be added and deleted from the middle. In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list.

In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0]. Consider the deques shown in Fig. 3.18.

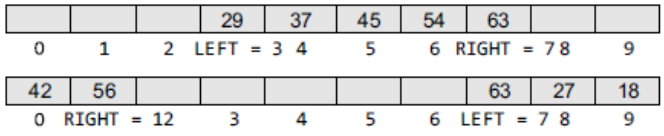



**Figure 3.18** Double-ended queues

There are two variants of a double-ended queue. They include

- *Input restricted deque* In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
- *Output restricted deque* In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

### 3.5.3 Priority Queues

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely.

**Implementation of a Priority Queue**

There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list.

Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed. While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element. On the contrary, an unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list.

Practically, both these techniques are inefficient and usually a blend of these two approaches is adopted that takes roughly O(log n) time or less.

### 3.5.4 Multiple Queues

When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered.

To deal with this problem, the code will have to be modified to reallocate more space for the array. In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory.

Thus, there lies a tradeoff between the frequency of overflows and the space allocated. So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size. Figure 3.19 illustrates this concept.


**Figure 3.19** Multiple queues

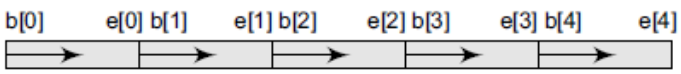



**Figure 3.20** Multiple queues

In the figure, an array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing—queue

A will grow from left to right, whereas queue B will grow from right to left at the same time. Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n], then each queue I will be allocated an equal amount of space bounded by indices b[i] and e[i]. This is shown in Fig. 3.20.

**Multiple-choice Questions**

**1**. A line in a grocery store represents a
(a) Stack (b) Queue
(c) Linked List (d) Array
2. In a queue, insertion is done at
(a) Rear (b) Front
(c) Back (d) Top
3. The function that deletes values from a queue is called
(a) enqueue (b) dequeue
(c) pop (d) peek
4. Typical time requirement for operations on queues is
(a) O(1) (b) O(n)
(c) O(log n) (d) O(n2)
5. The circular queue will be full only when
(a) FRONT = MAX –1 and REAR = Max –1
(b) FRONT = 0 and REAR = Max –1
(c) FRONT = MAX –1 and REAR = 0
(d) FRONT = 0 and REAR = 0

**Fill in the Blanks**

1. New nodes are added at _____ of the queue.

2. _____ allows insertion of elements at either ends but not in the middle.

3. The typical time requirement for operations in a linked queue is _____.

4. In _____, insertions can be done only at one end, while deletions can be done from both the ends.

5. Dequeue is implemented using _____.

6. _____ are appropriate data structures to process batch computer programs submitted to the computer centre.

7. _____ are appropriate data structures to process a list of employees having a contract for a seniority system for hiring and firing.

# STACKS

## 3.6 INTRODUCTION TO STACKS

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Fig. 3.21.

Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

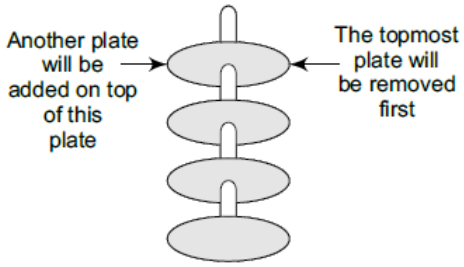



**Figure 3.21** Stack of plates

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP.

Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

## 3.7 ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.

There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full. (You must be wondering why we have written MAX–1. It is because array indices start from 0.) Look at Fig. 3.22.

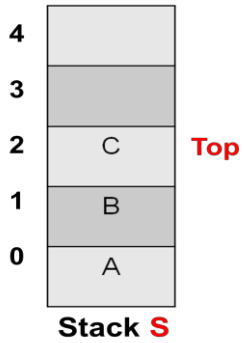



Fig. 3.22 Stack

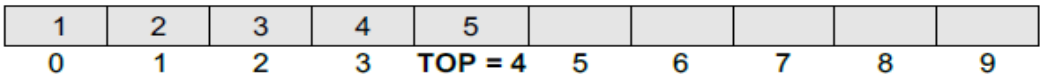
## 3.8 OPERATIONS ON A STACK

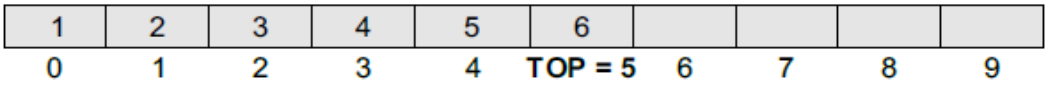
A stack supports three basic operations: push, pop, and peek.

The **push** operation adds an element to the top of the stack and the **pop** operation removes the element from the top of the stack. The **peek** operation returns the value of the topmost element of the stack.

### Push Operation

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- To insert an element with value 6, we first check if TOP=MAX–1.
- If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].

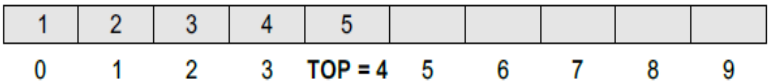


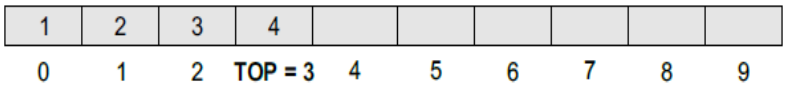




```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

### Pop Operation

- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done.
- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP.

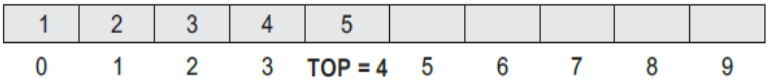



```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

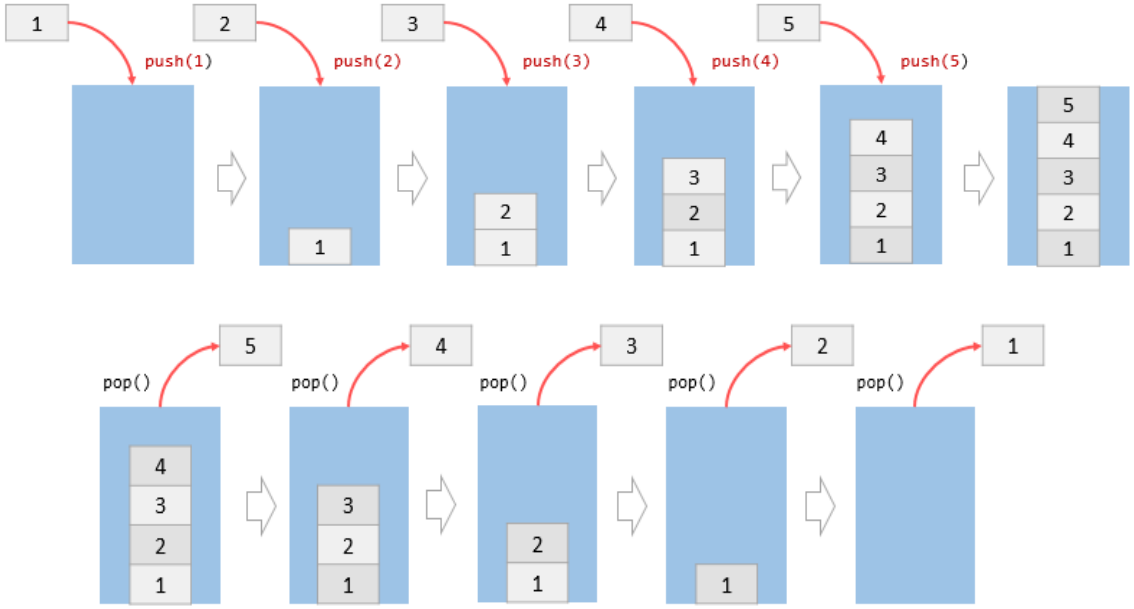
**Peek Operation**

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.
- Here, the Peek operation will return 5,as it is the value of the topmost element of the stack.




```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

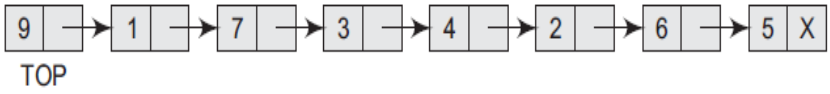
**Example:**




## 3.9 LINKED REPRESENTATION OF STACKS

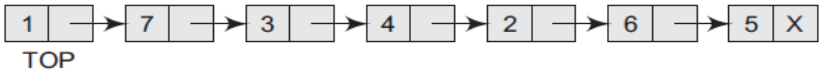
We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty. The linked representation of a stack is shown in below figure.



**Push Operation**

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in below figure.




To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.

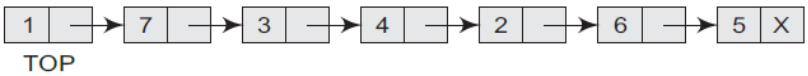

the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

**Pop Operation**

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in below figure.


In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in below figure.




```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

The algorithm to delete an element from a stack.

In Step 1, we first check for the UNDERFLOW condition.
In Step 2, we use a pointer PTR that points to TOP.
In Step 3, TOP is made to point to the next node in sequence.
In Step 4, the memory occupied by PTR is given back to the free pool.

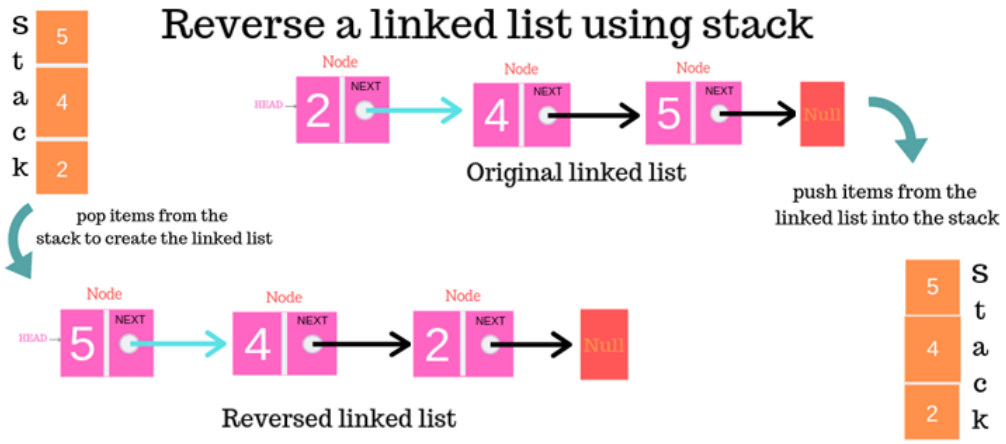
## 3.10 APPLICATIONS OF STACKS

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

### 3.10.1 Reversing list

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.




### 3.10.2 Evaluation of Arithmetic Expressions

**Polish Notations:**

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions.

While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; here, plus operator is placed between the two operands A and B. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and associativity rules, and brackets which override these rules.
So, computers work more efficiently with expressions written using prefix and postfix notations. Postfix notation was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

The expression (A + B) * C can be written as:
[AB+]*C
AB+C* in the postfix notation

A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.

For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication. Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, AB+C*, + is applied on A and B, then * is applied on the result of addition and C.


**Conversion of an Infix Expression into a Postfix Expression:**

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators.

The precedence of these operators can be given as follows:

- Higher priority *, /, %
- Lower priority +, –

No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

**Example:** Convert the following infix expressions into postfix expressions.
**Solution:**
**(a) (A–B) * (C+D)**
[AB–] * [CD+]
AB–CD+*

**(b) (A + B) / (C + D) – (D * E)**
[AB+] / [CD+] – [DE*]
[AB+CD+/] – [DE*]
AB+CD+/DE*–

The algorithm given below transforms an infix expression into postfix expression. The algorithm accepts an infix expression that may contain operators, operands, and parentheses.

For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

**Example:** Convert the following infix expression into postfix expression using the algorithm

A – (B / C + (D % E * F) / G)* H

A – (B / C + (D % E * F) / G)* H)

| Infix Character Scanned | Stack | Postfix Expression |
|---|---|---|
| | ( | |
| A | ( | A |
| – | ( – | A |
| ( | ( – ( | A |
| B | ( – ( | A B |
| / | ( – ( / | A B |
| C | ( – ( / | A B C |
| + | ( – ( + | A B C / |
| ( | ( – ( + ( | A B C / |
| D | ( – ( + ( | A B C / D |
| % | ( – ( + ( % | A B C / D |
| E | ( – ( + ( % | A B C / D E |
| * | ( – ( + ( % * | A B C / D E |
| F | ( – ( + ( % * | A B C / D E F |
| ) | ( – ( + | A B C / D E F * % |
| / | ( – ( + / | A B C / D E F * % |
| G | ( – ( + / | A B C / D E F * % G |
| ) | ( – | A B C / D E F * % G / + |
| * | ( – * | A B C / D E F * % G / + |
| H | ( – * | A B C / D E F * % G / + H |
| ) | | A B C / D E F * % G / + H * – |

**Evaluation of a Postfix Expression:**

The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.

Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.

Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

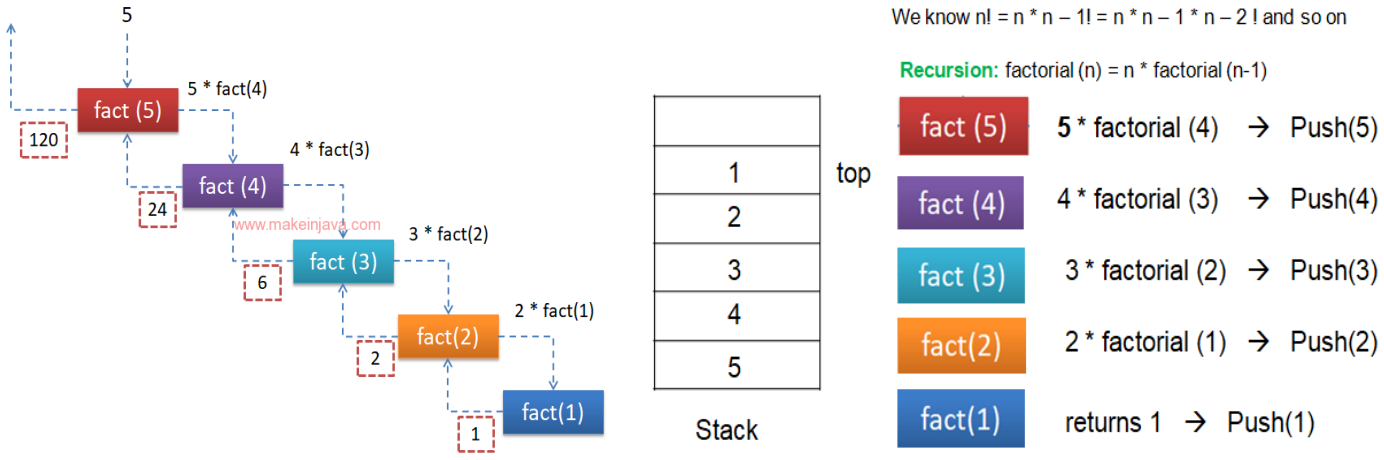
Algorithm to evaluate a postfix expression

| Character Scanned | Stack |
|---|---|
| 9 | 9 |
| 3 | 9, 3 |
| 4 | 9, 3, 4 |
| * | 9, 12 |
| 8 | 9, 12, 8 |
| + | 9, 20 |
| 4 | 9, 20, 4 |
| / | 9, 5 |
| - | 4 |

Evaluation of a postfix expression

Let us now take an example that makes use of this algorithm.
Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluate the expression.
The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation.

**Factorial Calculation:**



Stack

Pop(1) Factorial value = 1

Pop(2) Factorial value = 1 * 2 = 2

Pop(3) Factorial value = 2 * 3 = 6

Pop(4) Factorial value = 6 * 4 = 24

Pop(5) Factorial value = 24 * 5 = 120

**Factorial value = 120**

# 4.TREES

## UNIT IV

**Trees:** Basic Terminology in Trees, Binary Trees-Properties, Representation of Binary Trees using Arrays and Linked lists. Binary Search Trees- Basic Concepts, BST Operations: Insertion, Deletion, Tree Traversals, Applications-Expression Trees, Heap Sort, Balanced Binary Trees- AVL Trees, Insertion, Deletion and Rotations.

### 4.1 BASIC TERMINOLOGY IN TREES

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root. Figure 4.1 shows a tree where node A is the root node; nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

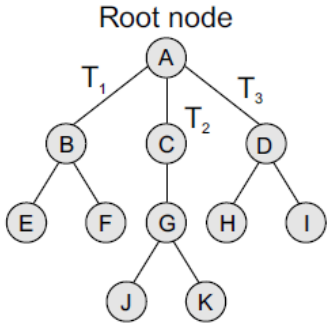



**Figure 4.1** Tree

**Root node** The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.

**Sub-trees** If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.

**Leaf node** A node that has no children is called the leaf node or the terminal node.

**Path** A sequence of consecutive edges is called a path. For example, in Fig. 9.1, the path from the root node A to node I is given as: A, D, and I.

**Ancestor node** An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. In the tree given in Fig. 9.1, nodes A, C, and G are the ancestors of node K.

**Descendant node** A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. In the tree given in Fig. 9.1, nodes C, G, J, and K are the descendants of node A.

**Level number** Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

**Degree** Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.

- ✓ **In-degree** In-degree of a node is the number of edges arriving at that node.
- ✓ **Out-degree** Out-degree of a node is the number of edges leaving that node.

## 4.2 BINARY TREES

A binary tree is a data structure that is defined as a collection of elements called nodes. In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.

A node that has zero children is called a leaf node or a terminal node. Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child. The root element is pointed by a 'root' pointer. If root = NULL, then it means the tree is empty.

Figure 4.2 shows a binary tree. In the figure, R is the root node and the two trees T1 and T2 are called the left and right sub-trees of R. T1 is said to be the left successor of R. Likewise, T2 is called the right successor of R.

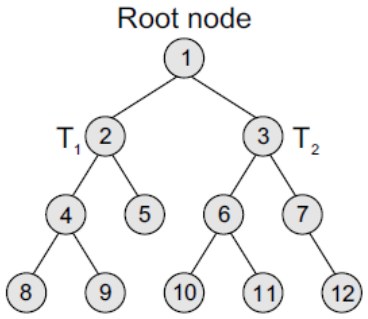



**Figure 4.2** Binary tree

Note that the left sub-tree of the root node consists of the nodes: 2, 4, 5, 8, and 9. Similarly, the right sub-tree of the root node consists of nodes: 3, 6, 7, 10, 11, and 12.

In the tree, root node 1 has two successors: 2 and 3. Node 2 has two successor nodes: 4 and 5. Node 4 has two successors: 8 and 9. Node 5 has no successor. Node 3 has two successor nodes: 6 and 7. Node 6 has two successors: 10 and 11. Finally, node 7 has only one successor: 12.

A binary tree is recursive by definition as every node in the tree contains a left sub-tree and a right sub-tree. Even the terminal nodes contain an empty left sub-tree and an empty right sub-tree.
Look at Fig. 4.2, nodes 5, 8, 9, 10, 11, and 12 have no successors and thus said to have empty sub-trees.

**Terminology:**

**Parent** If N is any node in T that has left successor $S_1$ and right successor $S_2$, then N is called the parent of $S_1$ and $S_2$. Correspondingly, $S_1$ and $S_2$ are called the left child and the right child of N. Every node other than the root node has a parent.

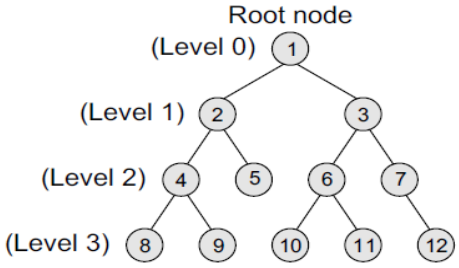



**Figure 4.3** Levels in binary tree

**Level number** Every node in the binary tree is assigned a level number (refer Fig. 4.3). The root node is defined to be at level 0. The left and the right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents. So all child nodes are defined to have level number as parent's level number + 1.

**Degree of a node** It is equal to the number of children that a node has. The degree of a leaf node is zero. For example, in the tree, degree of node 4 is 2, degree of node 5 is zero and degree of node 7 is 1.

**Sibling** All nodes that are at the same level and share the same parent are called siblings (brothers). For example, nodes 2 and 3; nodes 4 and 5; nodes 6 and 7; nodes 8 and 9; and nodes 10 and 11 are siblings.

**Leaf node** A node that has no children is called a leaf node or a terminal node. The leaf nodes in the tree are: 8, 9, 5, 10, 11, and 12.

**Similar binary trees** Two binary trees T and T' are said to be similar if both these trees have the same structure. Figure 4.4 shows two similar binary trees.

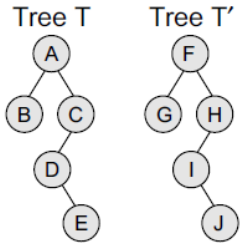



**Figure 4.4** Similar binary trees

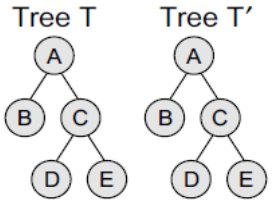



**Figure 4.5** T' is a copy of T

**Copies** Two binary trees T and T' are said to be copies if they have similar structure and if they have same content at the corresponding nodes. Figure 4.5 shows that T' is a copy of T.

**Edge** It is the line connecting a node N to any of its successors. A binary tree of n nodes has exactly n – 1 edges because every node except the root node is connected to its parent via an edge.

**Path** A sequence of consecutive edges. For example, in Fig. 4.3, the path from the root node to the node 8 is given as: 1, 2, 4, and 8.

**Depth** The depth of a node N is given as the length of the path from the root R to the node N. The depth of the root node is zero.

**Height of a tree** It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1.

A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. This is because every level will have at least one node and can have at most 2 nodes. So, if every level has two nodes then a tree with height h will have at the most $2^h - 1$ nodes as at level 0, there is only one element called the root. The height of a binary tree with n nodes is at least $\log_2(n+1)$ and at most n.

**In-degree/out-degree of a node** It is the number of edges arriving at a node. The root node is the only node that has an in-degree equal to zero. Similarly, out-degree of a node is the number of edges leaving that node. Binary trees are commonly used to implement binary search trees, expression trees, tournament trees, and binary heaps.

**Complete Binary Trees**

A complete binary tree is a binary tree that satisfies two properties. First, in a complete binary tree, every level, except possibly the last, is completely filled. Second, all nodes appear as far left as possible.
In a complete binary tree $T_n$, there are exactly n nodes and level r of T can have at most $2^r$ nodes. Figure 9.7 shows a complete binary tree.

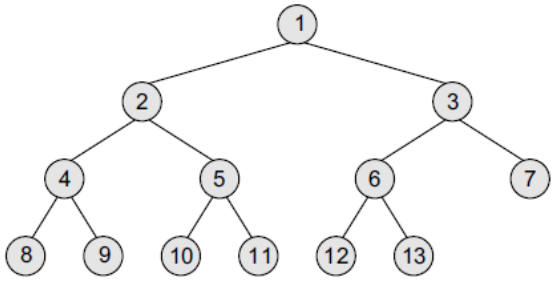



**Figure 4.6** Complete binary tree

Note that in Fig. 4.6, level 0 has $2^0 = 1$ node, level 1 has $2^1 = 2$ nodes, level 2 has $2^2 = 4$ nodes, level 3 has 6 nodes which is less than the maximum of $2^3 = 8$ nodes.

In Fig. 4.6, tree $T_{13}$ has exactly 13 nodes. They have been purposely labelled from 1 to 13, so that it is easy for the reader to find the parent node, the right child node, and the left child node of the given node.

The formula can be given as—if K is a parent node, then its left child can be calculated as 2 × K and its right child can be calculated as 2 × K + 1.
For example, the children of the node 4 are 8 (2 × 4) and 9 (2 × 4 + 1).
Similarly, the parent of the node K can be calculated as | K/2 |.
Given the node 4, its parent can be calculated as | 4/2 | = 2. The height of a tree $T_n$ having exactly n nodes is given as: $H_n = | \log_2 (n + 1) |$

NOTE: This means, if a tree T has 10,00,000 nodes, then its height is 21

**Extended Binary Trees**
A binary tree T is said to be an extended binary tree (or a 2-tree) if each node in the tree has either no child or exactly two children. Figure 4.7 shows how an ordinary binary tree is converted into an extended binary tree.
In an extended binary tree, nodes having two children are called internal nodes and nodes having no children are called external nodes. In Fig. 4.7, the internal nodes are represented using circles and the external nodes are represented using squares.
To convert a binary tree into an extended tree, every empty sub-tree is replaced by a new node. The original nodes in the tree are the internal nodes, and the new nodes added are called the external nodes.

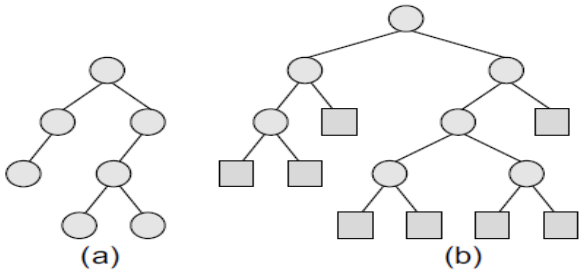



**Figure 4.7** (a) Binary tree and (b) extended binary tree

## 4. 3 REPRESENTATION OF BINARY TREES IN THE MEMORY

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.

**Linked representation of binary trees** In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.
So in C, the binary tree is built with a node type given below.

```
struct node
{
        struct node *left;
        int data;
        struct node *right;
};
```

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty. Consider the binary tree given in Fig. 4.2. The schematic diagram of the linked representation of the binary tree is shown in Fig. 4.8.

In Fig. 4.8, the left position is used to point to the left child of the node or to store the address of the left child of the node. The middle position is used to store the data. Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node. Empty sub-trees are represented using X (meaning NULL).

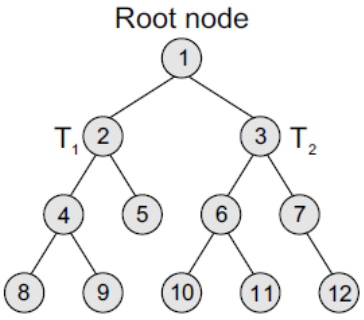



**Figure 4.2** Binary tree

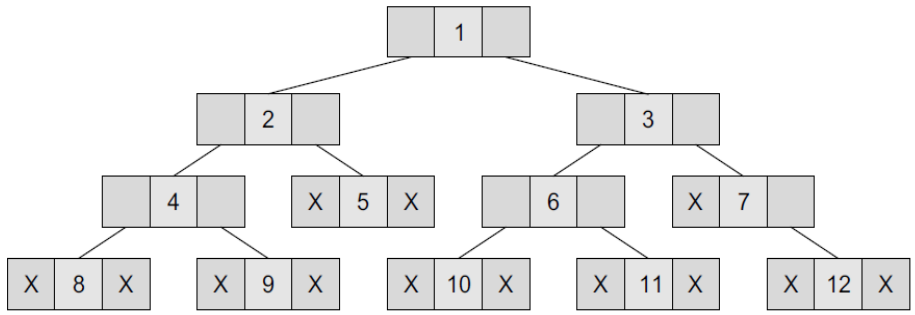



**Figure 4.8** Linked representation of a binary tree

Sequential representation of binary trees Sequential representation of trees is done using single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space.

A sequential binary tree follows the following rules:

- ✓ A one-dimensional array, called TREE, is used to store the elements of tree.
- ✓ The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
- ✓ The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1).
- ✓ The maximum size of the array TREE is given as (2h–1), where h is the height of the tree.
- ✓ An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.

Figure 4.9 shows a binary tree and its corresponding sequential representation. The tree has 11 nodes and its height is 4.

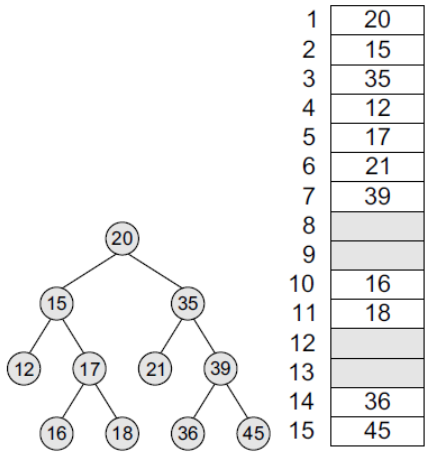



**Figure 4.9** Binary tree and its sequential representation

## 4.4 TRAVERSING A BINARY TREE

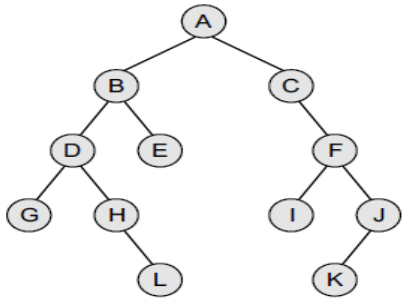
Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a nonlinear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited. In this section, we will discuss these algorithms.

### Pre-order Traversal

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:
1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

Pre-order traversal is also called as depth-first traversal. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right).

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:          Write TREE -> DATA
Step 3:          PREORDER(TREE -> LEFT)
Step 4:          PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```




TRAVERSAL ORDER:
A, B, D, G, H, L, E, C, F, I, J, and K

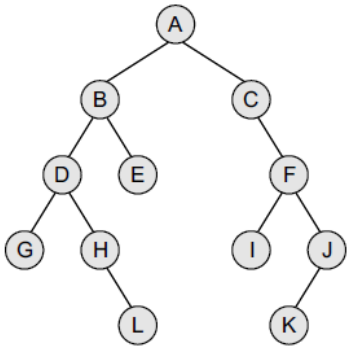
**In-order Traversal**

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:
1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           INORDER(TREE -> LEFT)
Step 3:           Write TREE -> DATA
Step 4:           INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

In-order traversal is also called as symmetric traversal. In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree.

The word 'in' in the in-order specifies that the root node is accessed in between the left and the right sub-trees. In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).



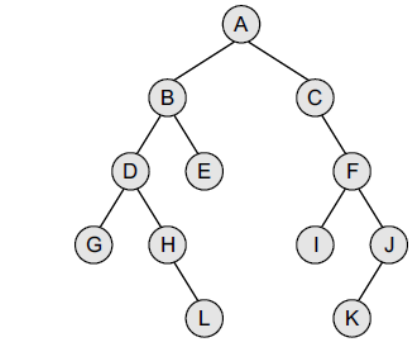

TRAVERSAL ORDER:
G, D, H, L, B, E, A, C, I, F, K, and J

**Post-order Traversal**

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:
1. Traversing the left sub-tree,
2. Traversing the right sub-tree, and finally
3. Visiting the root node.

In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word 'post' in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-Node).

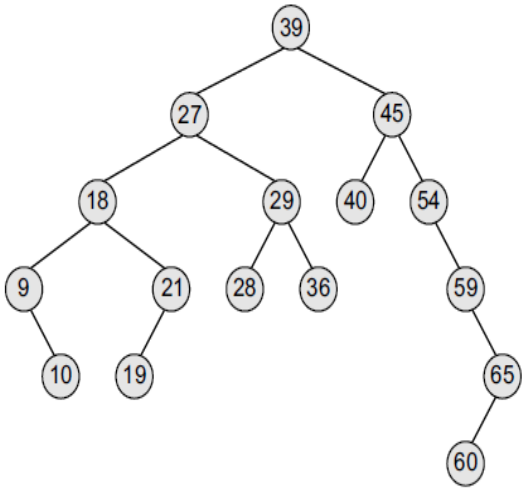
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           POSTORDER(TREE -> LEFT)
Step 3:           POSTORDER(TREE -> RIGHT)
Step 4:           Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```




TRAVERSAL ORDER:
G, L, H, D, E, B, I, K, J, F, C, and A

## 4.5 BINARY SEARCH TREES

A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order. In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree. (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)




The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36.

All these nodes have smaller values than the root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65.

Recursively, each of the sub-trees also obeys the binary search tree constraint.

For example, in the left sub-tree of the root node, 27 is the root and all elements in its left sub-tree (9, 10, 18, 19, 21) are smaller than 27, while all nodes in its right sub-tree (28, 29, and 36) are greater than the root node's value.

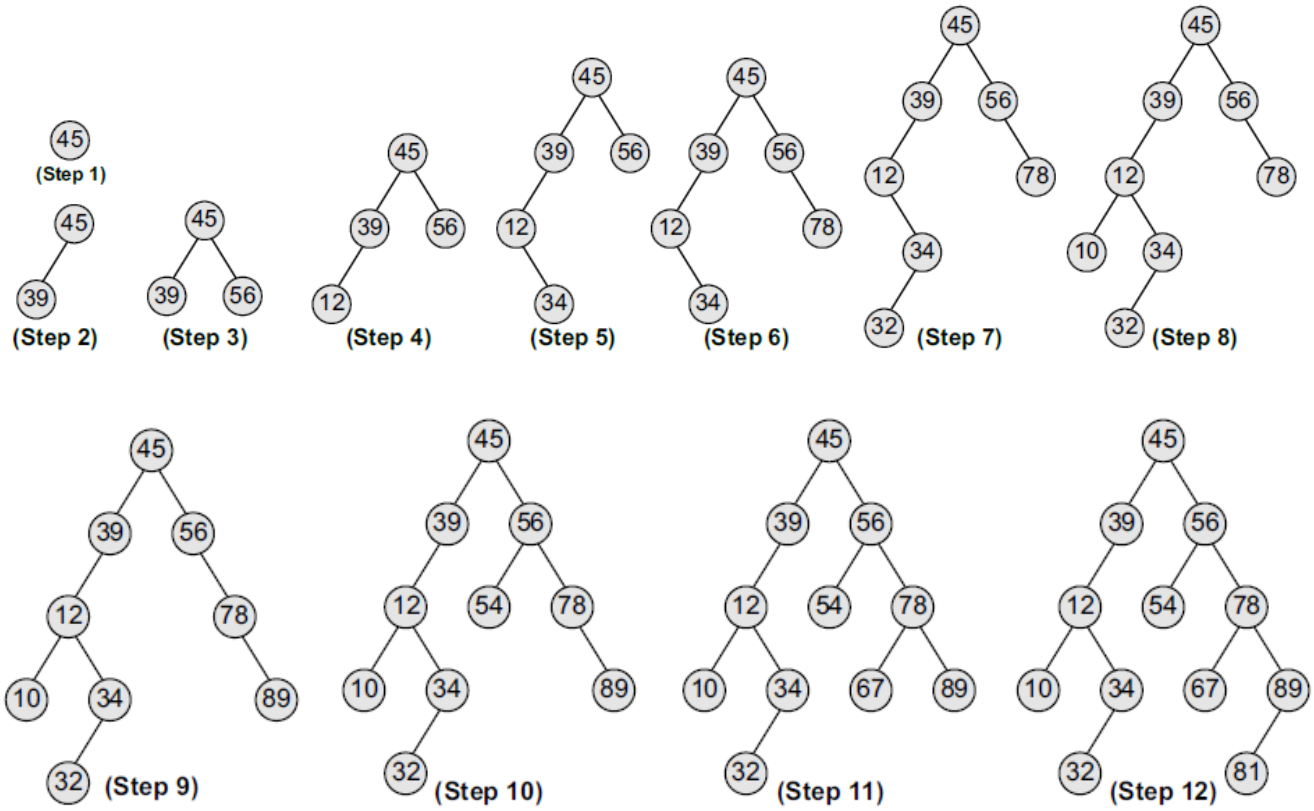
Binary search trees also speed up the insertion and deletion operations. The tree has a speed advantage when the data in the structure changes rapidly.
Binary search trees are considered to be efficient data structures especially when compared with sorted linear arrays and linked lists. In a sorted array, searching can be done in $O(\log_2 n)$ time, but insertions and deletions are quite expensive. In contrast, inserting and deleting elements in a linked list is easier, but searching for an element is done in O(n) time.
However, in the worst case, a binary search tree will take O(n) time to search for an element.

To summarize, a binary search tree is a binary tree with the following properties:

- ✓ The left sub-tree of a node N contains values that are less than N's value.
- ✓ The right sub-tree of a node N contains values that are greater than N's value.
- ✓ Both the left and the right binary trees also satisfy these properties and, thus, are binary search trees.

**Example:**

Create a binary search tree using the following data elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, and 81.




## 4.6 OPERATIONS ON BINARY SEARCH TREES

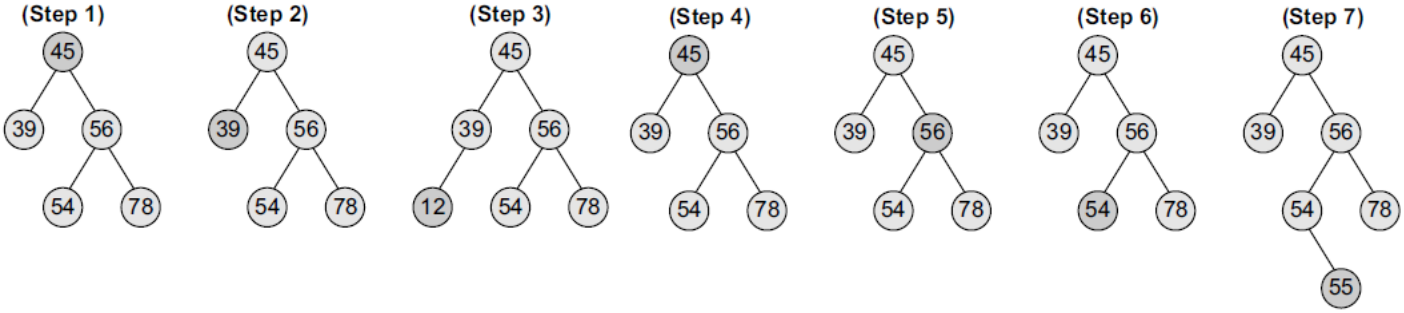
### Inserting a New Node in a Binary Search Tree

The insert function is used to add a new node with a given value at the correct position in the binary search tree. Adding the node at the correct position means that the new node should not violate the properties of the binary search tree. The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position. The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

The insert function requires time proportional to the height of the tree in the worst case. It takes O(log n) time to execute in the average case and O(n) time in the worst case.

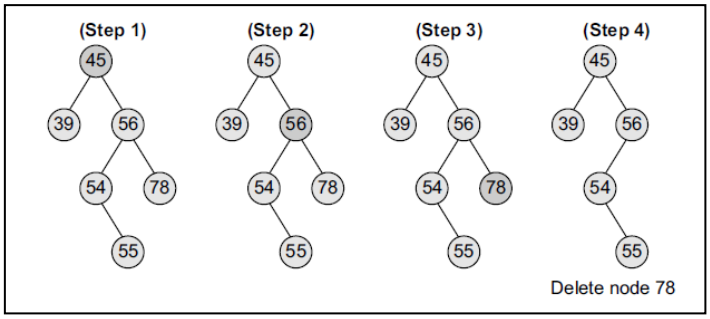
```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

**Example:** Inserting nodes with values 12 and 55 in the given binary search tree



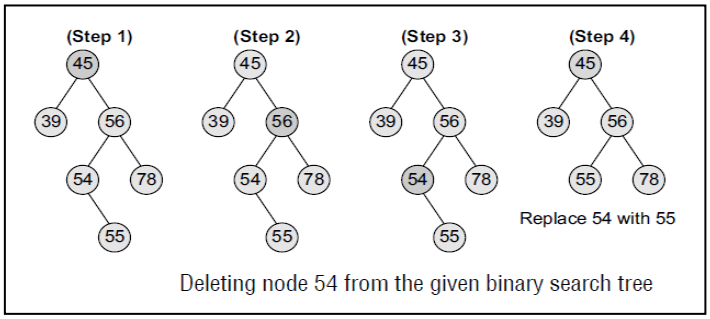

**Deleting a Node from a Binary Search Tree**

The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process.
Case 1: Deleting a Node that has No Children
Case 2: Deleting a Node with One Child
Case 3: Deleting a Node with Two Children




If we have to delete node 78, we can simply remove this node without any issue.
This is the simplest case of deletion.

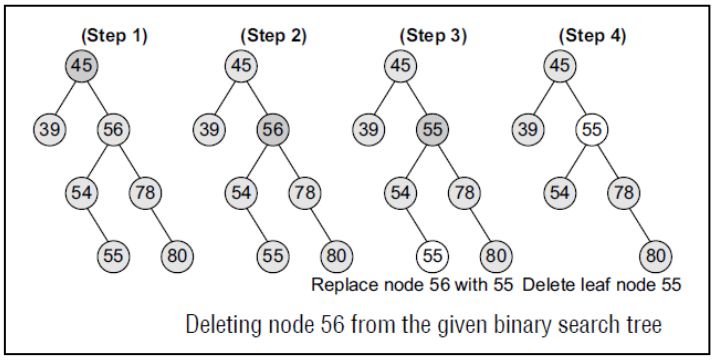
Case1: Deleting a Node that has No Children




To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.

Look at the binary search tree shown in figure and see how deletion of node 54 is handled.

Case 2: Deleting a Node with One Child




To handle this case, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
The in-order predecessor or the successor can then be deleted using any of the above cases.

Look at the binary search tree given in figure and see how deletion of node with value 56 is handled.

## 4.7 BALANCED BINARY TREES- AVL TREES

AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. The tree is named AVL in honour of its inventors. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the AVL tree is also known as a height-balanced tree. The key advantage of using an AVL tree is that it takes O(log n) time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to O(log n).

The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the Balance Factor. Thus, every node has a balance factor associated with it. The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of –1, 0, or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

*Balance factor = Height (left sub-tree) – Height (right sub-tree)*

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a left-heavy tree.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a right-heavy tree.

### 4.7.1 Operations on AVL Trees

**Searching for a Node in an AVL Tree**

Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree. Due to the height-balancing of the tree, the search operation takes O (log n) time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

**Inserting a New Node in an AVL Tree**

Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation. Rotation is done to restore the balance of the tree.

However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.

To perform rotation, our first task is to find the critical node. Critical node is the nearest ancestor node on the path from the inserted node to the root whose balance factor is neither –1, 0, nor 1.

The second task in rebalancing the tree is to determine which type of rotation has to be done. There are four types of rebalancing rotations and application of these rotations depends on the position of the inserted node with reference to the critical node.

The four categories of rotations are:

- **LL rotation:** The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
- **RR rotation:** The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
- **LR rotation:** The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
- **RL rotation:** The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

## LL Rotation

**Example** Consider the AVL tree given in Fig. and insert 18 into it.

***Solution***

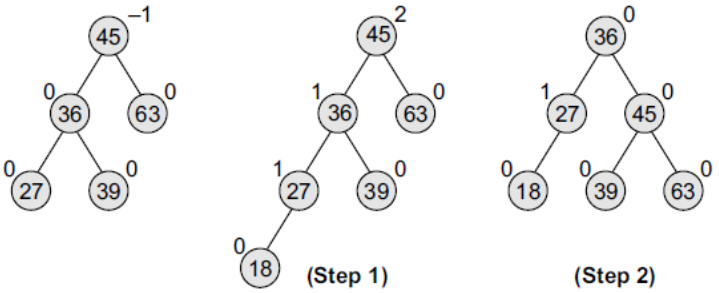
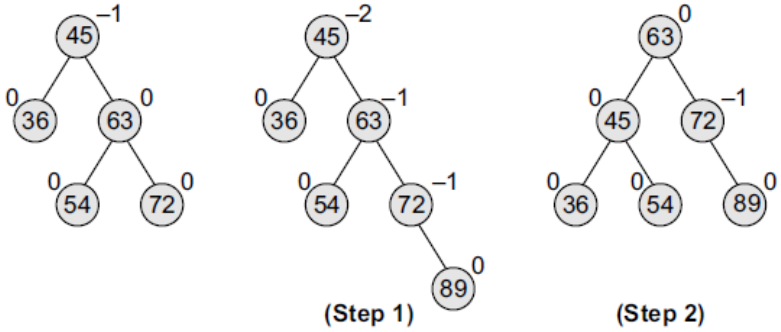



**Figure** AVL tree

## RR Rotation

**Example** Consider the AVL tree given in Fig. and insert 89 into it.

***Solution***




## LR and RL Rotations

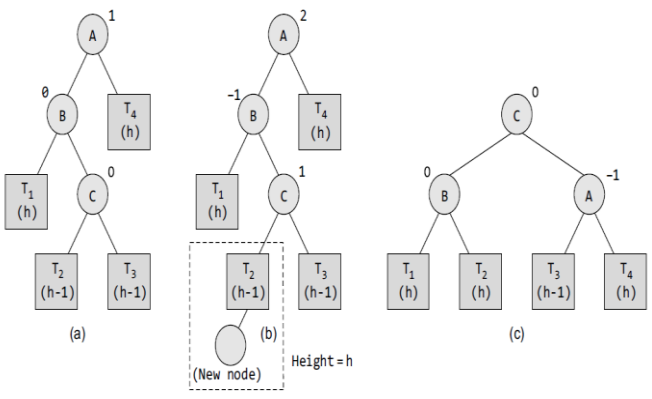



**Figure** LR rotation in an AVL tree

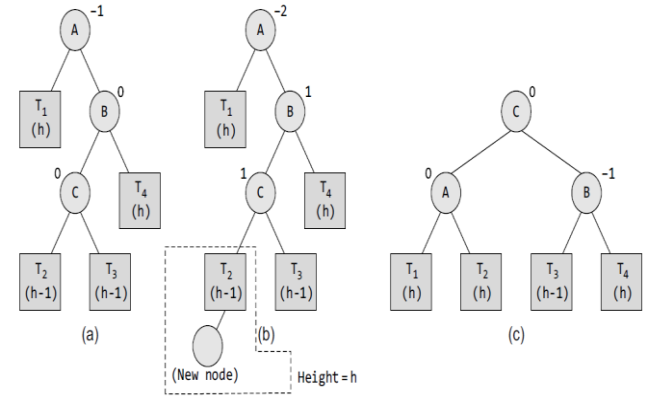
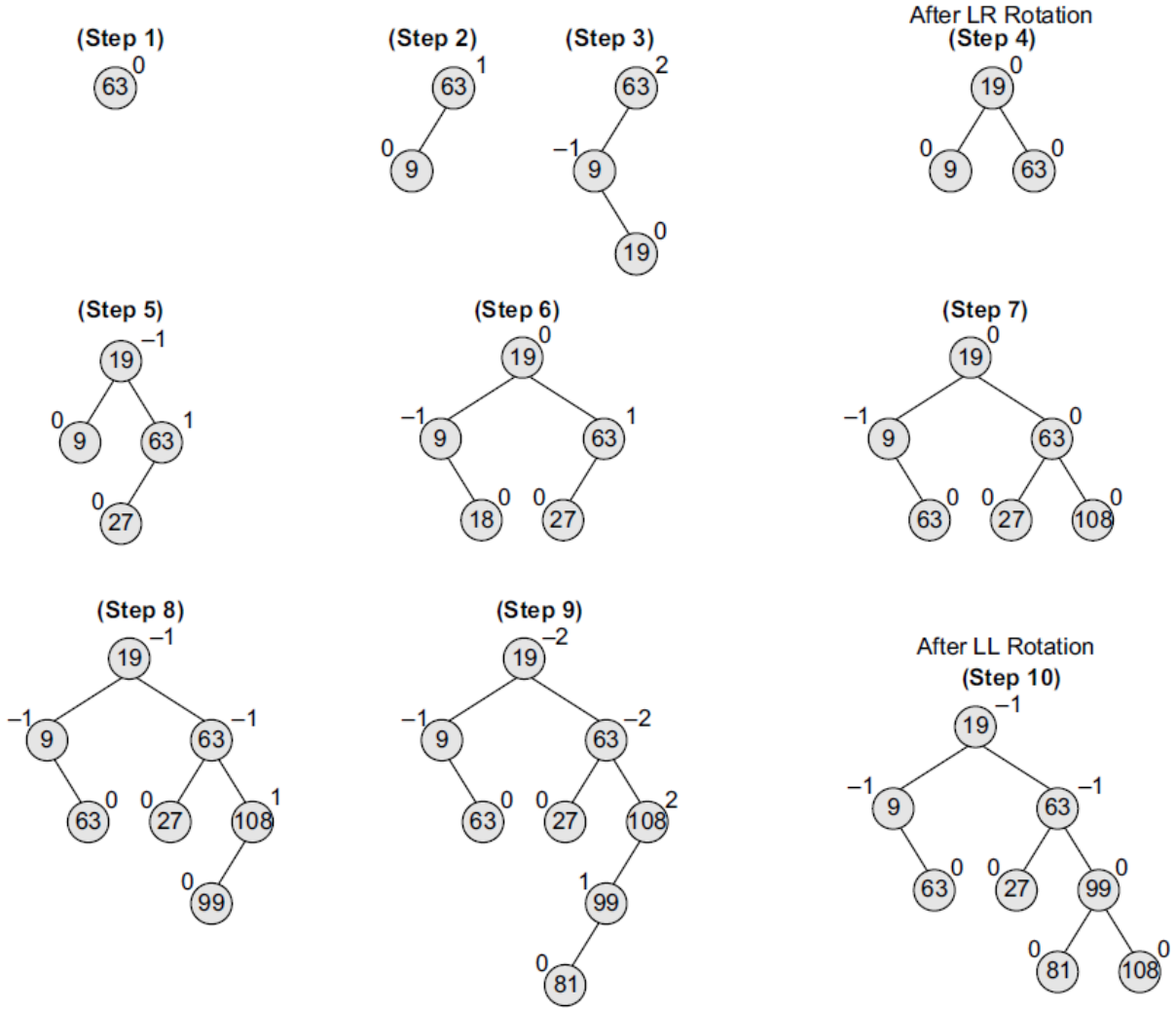



**Figure** RL rotation in an AVL tree

**Example:** Construct an AVL tree by inserting the following elements in the given order.
63, 9, 19, 27, 18, 108, 99, 81.



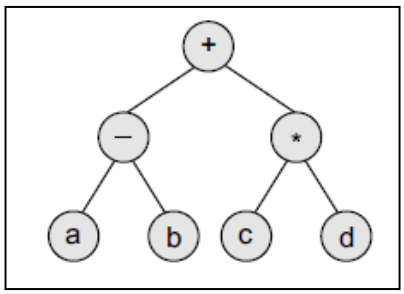

**Deleting a Node from an AVL Tree**

Deletion of a node in an AVL tree is similar to that of binary search trees. But it goes one step ahead. Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree, we need to perform rotations.

## 4.8 APPLICATIONS OF TREES

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system use.
- Another variation of tree, B-trees are prominently used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are an important data structure used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables.

### 4.8.1 Expression Trees

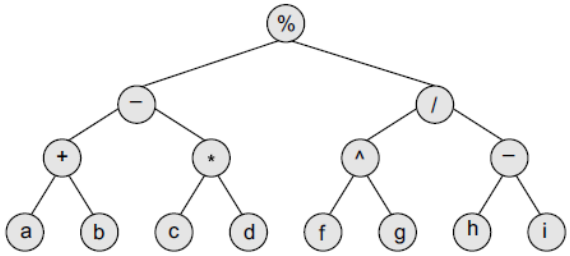



Binary trees are widely used to store algebraic expressions. For example, consider the algebraic expression given as:

Exp = (a – b) + (c * d)

This expression can be represented using a binary tree as shown in Figure.

Given an expression, Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h)), construct the corresponding binary tree.




Expression tree

### 4.8.2 Heap Sort

Heap: Recall that a heap is a complete binary tree such that the weight of every node is less than the weights of its children

A heap with n elements can be conveniently represented as the first n elements of an array. Furthermore, the children of a[i] can be found in a[2i] (left child) and a[2i + 1] (right child)

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
2. Start deleteMin operations, storing each deleted element at the end of the heap array.

After performing step 2, the order of the elements will be opposite to the order in the heap tree.
Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree in descending order - the greatest element will have the highest priority.

Note that we use only one array, treating its parts differently:

a. When building the heap tree, part of the array will be considered as the heap, and the rest part - the original array.
b. When sorting, part of the array will be the heap, and the rest part - the sorted array.

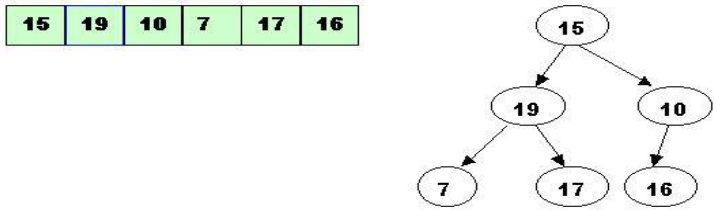
**Example:**

Given an array of 6 elements: 15, 19, 10, 7, 17, and 16 sort it in ascending order using heap sort

Here is the array: 15, 19, 10, 7, 17, and 6

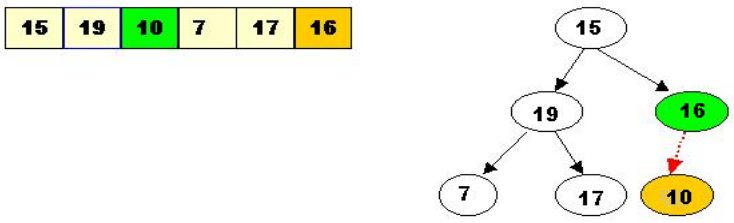
**Building the heap tree:**

The array represented as a tree, complete but not ordered:

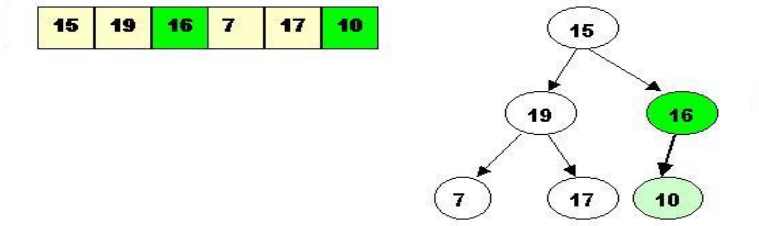



Start with the rightmost node at height 1, the node at position 3 = Size/2.

It has one greater child and has to be percolated down:



After processing array [3] the situation is:

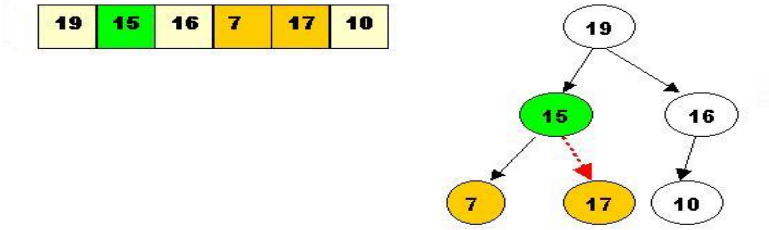



Next come array [2]. Its children are smaller, so no percolation is needed.

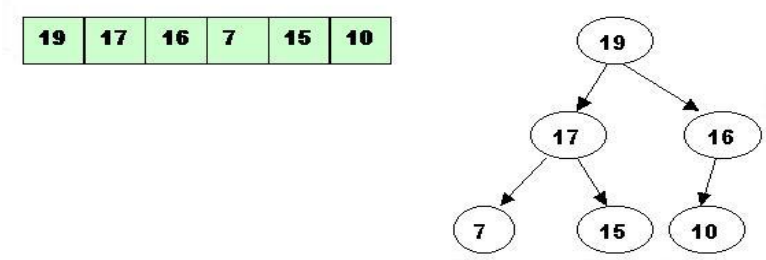
The last node to be processed is array [1]. Its left child is the greater of the children.

The item at array [1] has to be percolated down to the left, swapped with array [2].

As a result the situation is:




The children of array [2] are greater, and item 15 has to be moved down further, swapped with array [5].

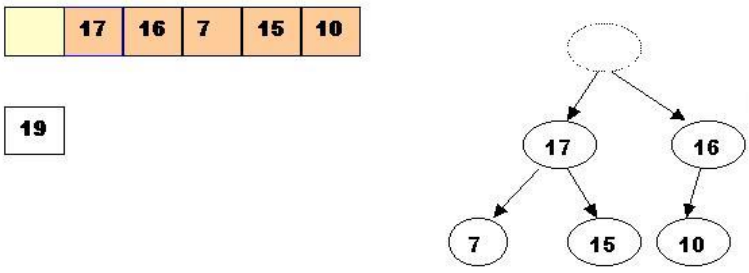



Now the tree is ordered, and the binary heap is built.

**Sorting - performing deleteMax operations:**

Delete the top element 19.

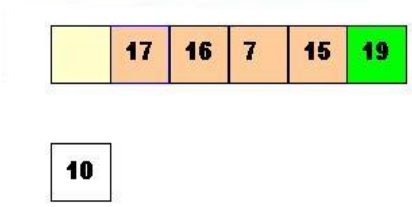
Store 19 in a temporary place, a hole is created at the top



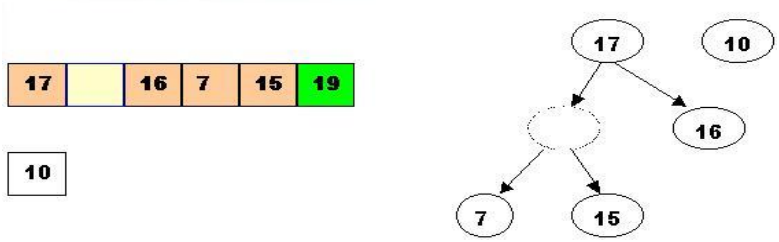
Swap 19 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.

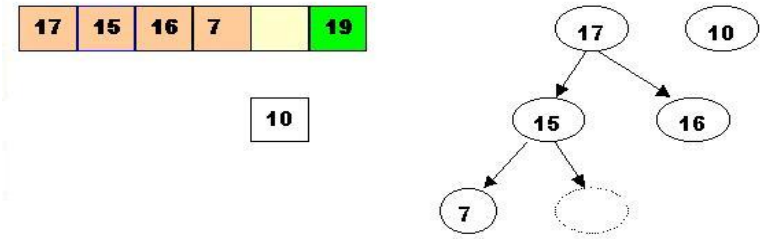
Instead it becomes a cell from the sorted array



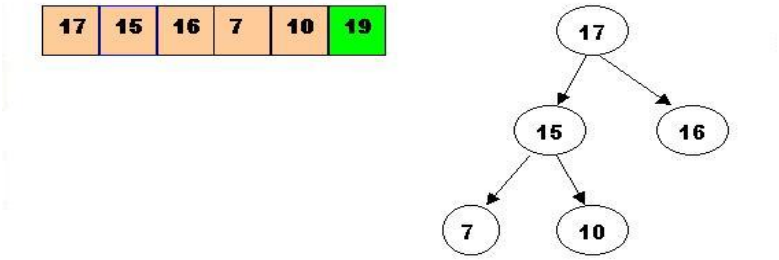

Percolate down the hole




Percolate once more (10 is less that 15, so it cannot be inserted in the previous hole)

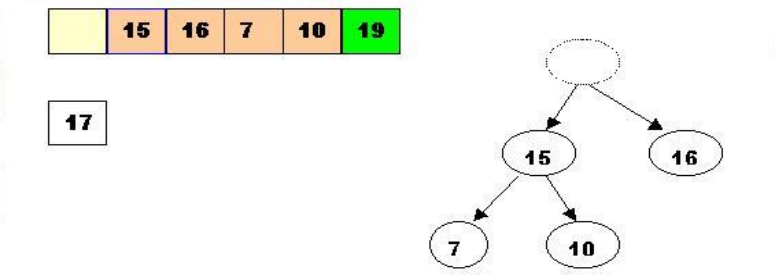



Now 10 can be inserted in the hole



**DeleteMax the top element 17**

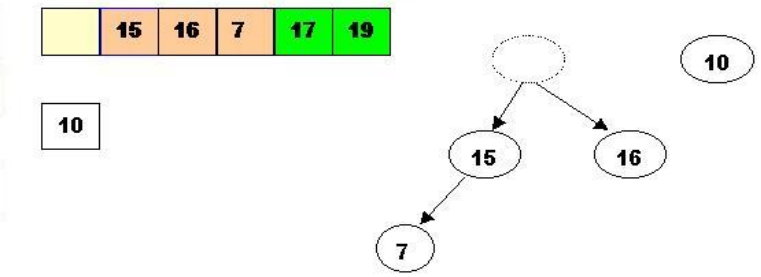
Store 17 in a temporary place, a hole is created at the top



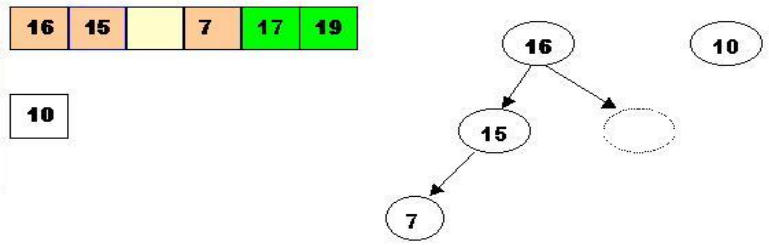

Swap 17 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.

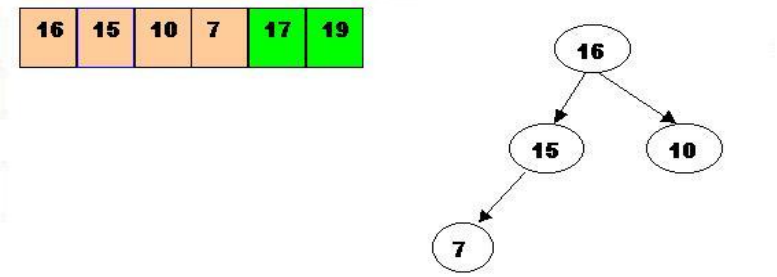
Instead it becomes a cell from the sorted array




The element 10 is less than the children of the hole, and we percolate the hole down:

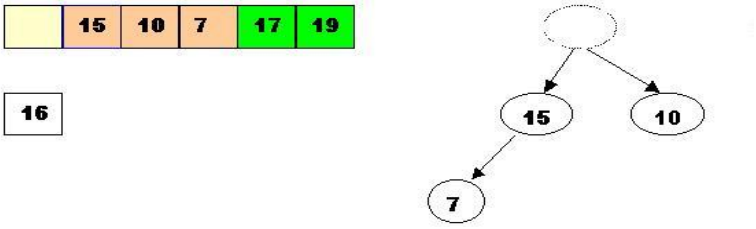



Insert 10 in the hole



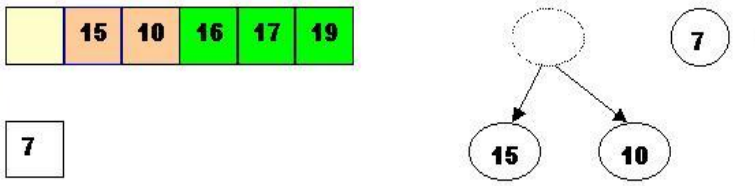
**DeleteMax 16**

Store 16 in a temporary place, a hole is created at the top

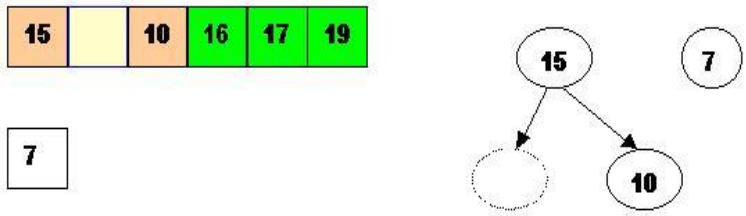



Swap 16 with the last element of the heap.

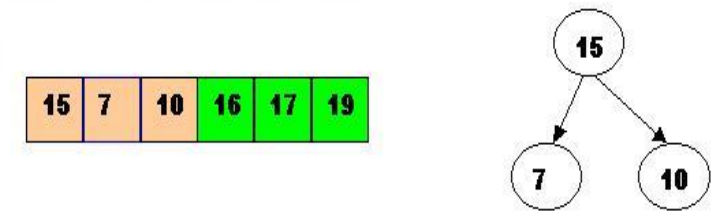
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array




Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)




Insert 7 in the hole

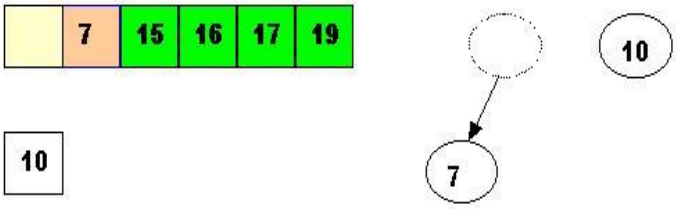



**DeleteMax the top element 15**

Store 15 in a temporary location, a hole is created.


Swap 15 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a position from the sorted array




Store 10 in the hole (10 is greater than the children of the hole)


**DeleteMax the top element 10**
Remove 10 from the heap and store it into a temporary location.


Swap 10 with the last element of the heap.
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array


Store 7 in the hole (as the only remaining element in the heap


7 is the last element from the heap, so now the array is sorted


The HEAPSORT procedure takes time O($n$ *log* $n$), since the call to BUILD_HEAP takes time $O(n)$ and each of the $n$ -1 calls to Heapify takes time O(log $n$).

# 5.GRAPHS

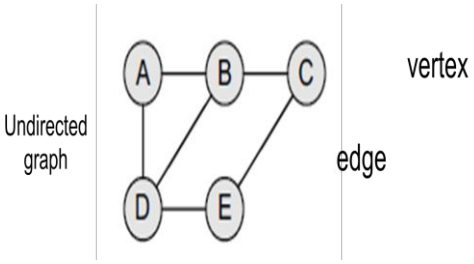
## UNIT V

**Graphs:** Basic Concepts, Representations of Graphs-Adjacency Matrix and using Linked list, Graph Traversals (BFT & DFT), Applications- Minimum Spanning Tree Using Prims & Kruskals Algorithm, Dijkstra's shortest path, Transitive closure, Warshall's Algorithm.

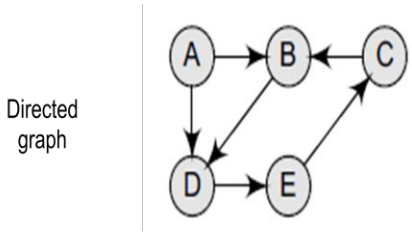
### 5.1 BASIC CONCEPTS

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices.

**Definition**
A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.




A graph can be directed or undirected. In an undirected graph, edges do not have any direction associated with them.
That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.
A graph G with V(G) = {A, B, C, D and E} and
E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}.




In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A.
The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).
A graph G with V(G) = {A, B, C, D and E} and
E(G) = {(A, B), (C, B), (A, D), (B, D), (D, E), (E, C)}.

#### 5.1.1 Graph Terminology

**Adjacent nodes** or **neighbours** For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbours.

**Degree of a node** Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

**Regular graph** It is a graph where each vertex has the same number of neighbours. That is, every node has the same degree. A regular graph with vertices of degree k is called a k–regular graph or a regular graph of degree k.

**Path** A path P written as P = {v0, v1, v2, ..., vn), of length n from a node u to v is defined as a sequence of (n+1) nodes. Here, u = v0, v = vn and vi–1 is adjacent to vi for i = 1, 2, 3,..., n.

**Closed path** A path P is known as a closed path if the edge has the same end-points. That is, if v0 = vn.

**Simple path** A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0 = vn, then the path is called a closed simple path.

**Cycle** A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

**Connected graph** A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree. Therefore, a tree is treated as a special graph.

**Complete graph** A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.

**Labelled graph** or **weighted graph** A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.

**Multiple edges** Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

**Loop** An edge that has identical end-points is called a loop. That is, e = (u, u).

**Multi-graph** A graph with multiple edges and/or loops is called a multi-graph.

**Size of a graph** The size of a graph is the total number of edges in it.

**BI-CONNECTED components**

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected.

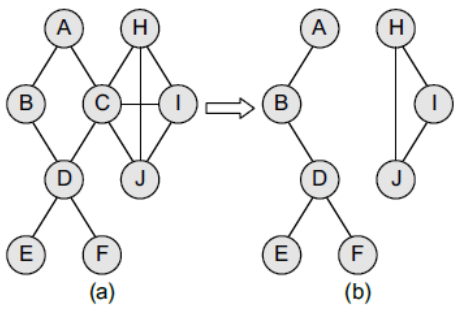



**Figure** Non bi-connected graph

A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.

In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

Note that the graph shown in Fig. (a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph (Fig. (b)).

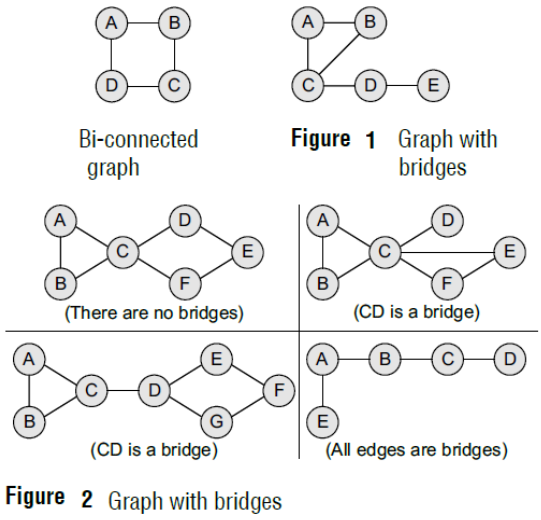



**Figure 2** Graph with bridges

As for vertices, there is a related concept for edges. An edge in a graph is called a bridge if removing that edge results in a disconnected graph.

Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge. Look at the graph shown in Fig.1.

In the graph, CD and DE are bridges. Consider some more examples shown in Fig. 2.

## 5.2 REPRESENTATION OF GRAPHS

There are three common ways of storing graphs in the computer's memory.
They are:

- Sequential representation by using an adjacency matrix.
- Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.
- Adjacency multi-list which is an extension of linked representation.

### 5.2.1 Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n × n.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry $a_{ij}$ in the adjacency matrix will contain 1, if vertices $v_i$ and $v_j$ are adjacent to each other. However, if the nodes are not adjacent, $a_{ij}$ will be set to zero. It is summarized in Figure.

Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

From the above examples, we can draw the following conclusions:

- ✓ For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.
- ✓ The adjacency matrix of an undirected graph is symmetric.
- ✓ The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- ✓ Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- ✓ The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

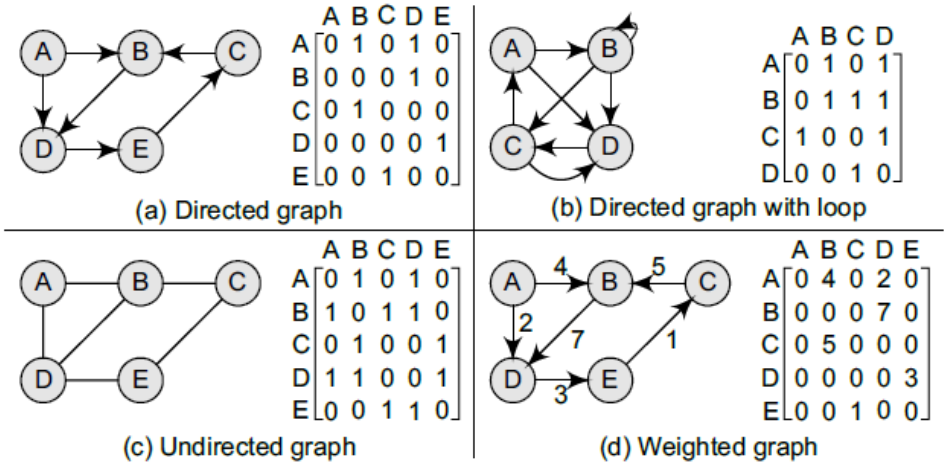



**Figure** Graphs and their corresponding adjacency matrices

### 5.2.2 Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory.
This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- ✓ It is easy to follow and clearly shows the adjacent nodes of a particular node.
- ✓ It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- ✓ Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

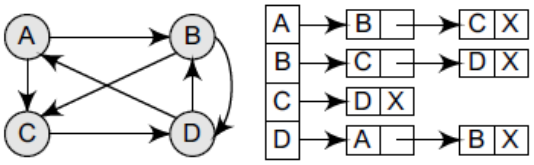



**Figure** Graph G and its adjacency list

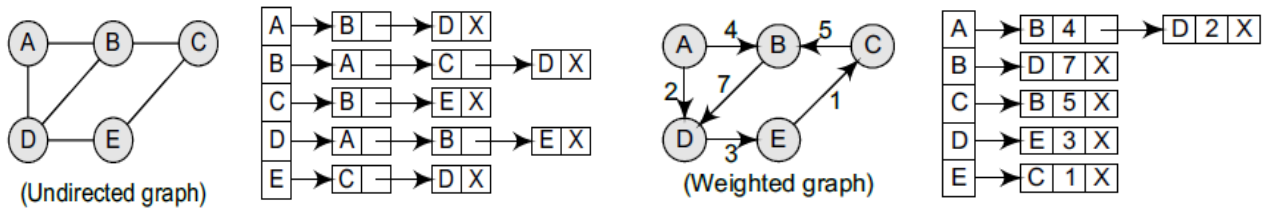



**Figure** Adjacency list for an undirected graph and a weighted graph

## 5.3 GRAPH TRAVERSAL ALGORITHMS

In this section, we will discuss how to traverse graphs. By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal which we will discuss in this section.
These two methods are:
1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state.

| Status | State of the node | Description |
|---|---|---|
| 1 | Ready | The initial state of the node N |
| 2 | Waiting | Node N is placed on the queue or stack and waiting to be processed |
| 3 | Processed | Node N has been completely processed |

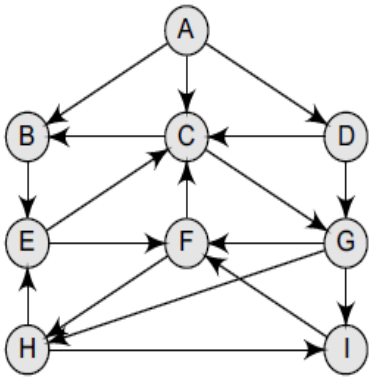
### 5.3.1 Breadth-first search (BFS):

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.

That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

Algorithm for breadth-first search




**Adjacency lists**
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

**Figure** Graph G and its adjacency list

**Solution:**

- The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered.
- During the execution of the algorithm, we use two arrays: QUEUE and ORIG.
- While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1.
- Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

| | |
|---|---|
| FRONT = 0 | QUEUE = A |
| REAR = 0 | ORIG = \0 |

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

| | | | | |
|---|---|---|---|---|
| FRONT = 1 | QUEUE = A | B | C | D |
| REAR = 3 | ORIG = \0 | A | A | A |

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

| | | | | | |
|---|---|---|---|---|---|
| FRONT = 2 | QUEUE = A | B | C | D | E |
| REAR = 4 | ORIG = \0 | A | A | A | B |

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

| | | | | | | |
|---|---|---|---|---|---|---|
| FRONT = 3 | QUEUE = A | B | C | D | E | G |
| REAR = 5 | ORIG = \0 | A | A | A | B | C |

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

| | | | | | | |
|---|---|---|---|---|---|---|
| FRONT = 4 | QUEUE = A | B | C | D | E | G |
| REAR = 5 | ORIG = \0 | A | A | A | B | C |

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| FRONT = 5 | QUEUE = A | B | C | D | E | G | F |
| REAR = 6 | ORIG = \0 | A | A | A | B | C | E |

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| FRONT = 6 | QUEUE = A | B | C | D | E | G | F | H | I |
| REAR = 9 | ORIG = \O | A | A | A | B | C | E | G | G |

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have Path P as A -> C -> G -> I.

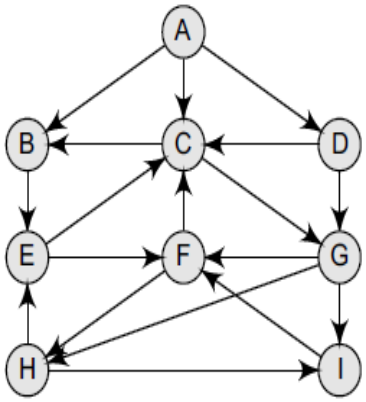
The time complexity can also be expressed as O( | E | + | V | )

### 5.3.2 Depth-first search (DFS)

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
In other words, depth-first search begins at a starting node A which becomes the current node.

Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```




**Adjacency lists**
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

**Figure** Graph G and its adjacency list

Consider the graph G given in Figure. The adjacency list of G is also given. Suppose we want to print all the nodes that can be reached from the node H (including H itself). One alternative is to use a depth-first search of G starting at node H. The procedure can be explained here.

(a) Push H onto the stack.

STACK: H

(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes

PRINT: H STACK: E, I

(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes

PRINT: I STACK: E, F

(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes

PRINT: F STACK: E, C

(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes

PRINT: C STACK: E, B, G

(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: G STACK: E, B

(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes

PRINT: B STACK: E

(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.

PRINT: E STACK:

Since the STACK is now empty, the depth-first search of G starting at node H is complete and the nodes which were printed are: H, I, F, C, G, B, E.

These are the nodes which are reachable from the node H.

The time complexity can be given as (O(|V| + |E|)).

## 5.4 SHORTEST PATH ALGORITHMS

In this section, we will discuss three different algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

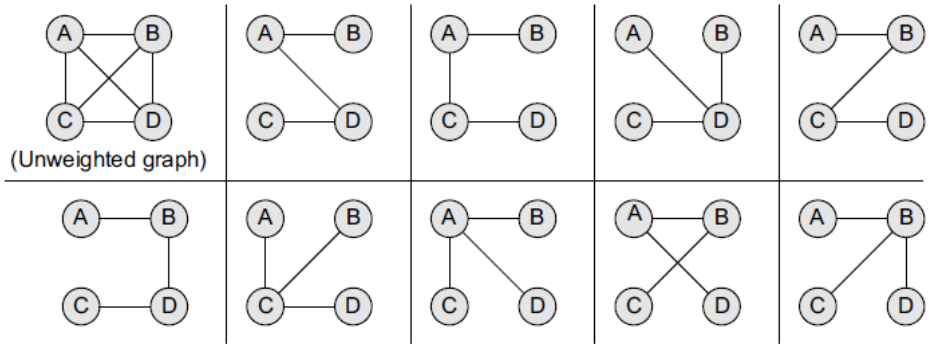
- Minimum spanning tree
- Dijkstra's algorithm
- Warshall's algorithm

While the first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

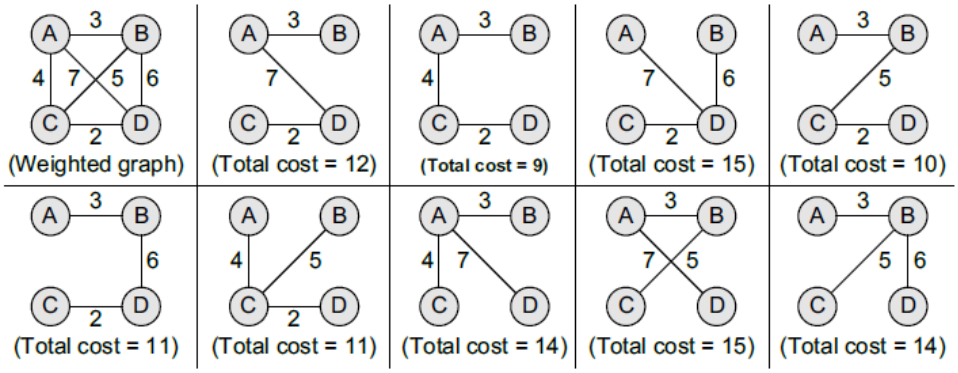
### 5.4.1 Minimum Spanning Trees

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees. We can assign weights to each edge (which is a number that represents how unfavourable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree. A *minimum spanning tree* (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Consider an unweighted graph G given below. From G, we can draw many distinct spanning trees. Eight of them are given here. For an unweighted graph, every spanning tree is a minimum spanning tree.




Consider a weighted graph G shown below. From G, we can draw three distinct spanning trees. But only a single minimum spanning tree can be obtained, that is, the one that has the minimum weight (cost) associated with it. Of all the spanning trees given in Figure, the one that is highlighted is called the minimum spanning tree, as it has the lowest cost associated with it.



### 5.4.1.1 Prims Algorithm

Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph. In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized.

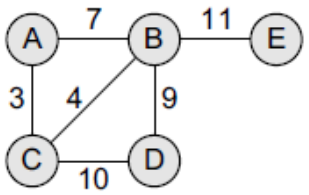
```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

Algorithm of Prims algorithm

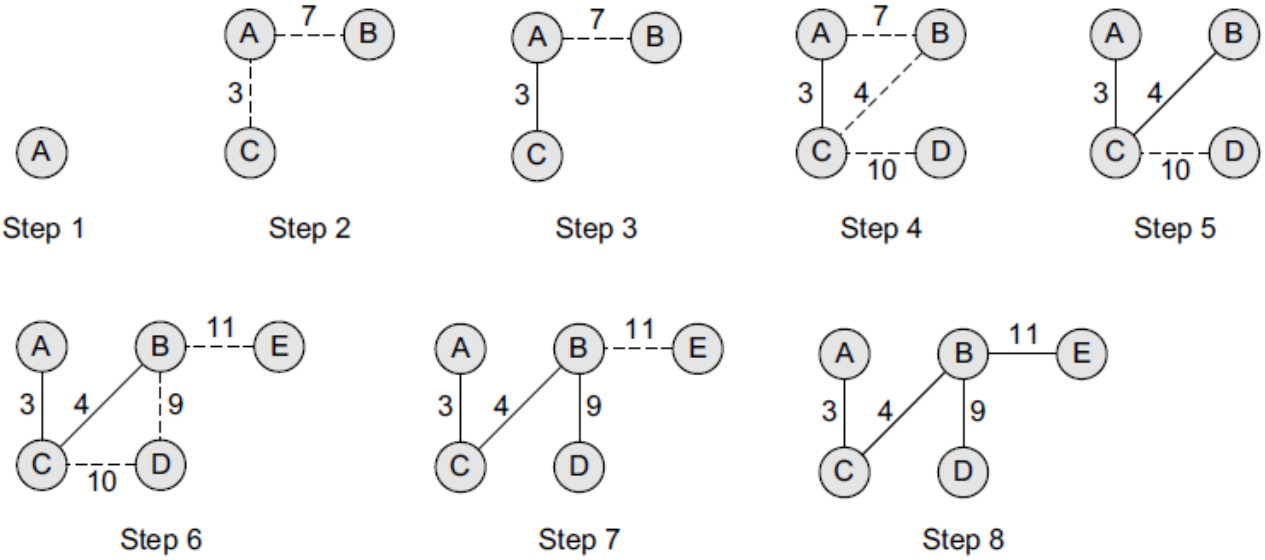
For this, the algorithm maintains three sets of vertices which can be given as below:

- Tree vertices Vertices that are a part of the minimum spanning tree T.
- Fringe vertices Vertices that are currently not a part of T, but are adjacent to some tree vertex.
- Unseen vertices Vertices that are neither tree vertices nor fringe vertices fall under this category.

Construct a minimum spanning tree of the graph




Step 1: Choose a starting vertex A.
Step 2: Add the fringe vertices (that are adjacent to A). The edges connecting the vertex and fringe vertices are shown with dotted lines.
Step 3: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting A and C has less weight, add C to the tree. Now C is not a fringe vertex but a tree vertex.
Step 4: Add the fringe vertices (that are adjacent to C).
Step 5: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting C and B has less weight, add B to the tree. Now B is not a fringe vertex but a tree vertex.
Step 6: Add the fringe vertices (that are adjacent to B).
Step 7: Select an edge connecting the tree vertex and the fringe vertex that has the minimum weight and add the selected edge and the vertex to the minimum spanning tree T. Since the edge connecting B and D has less weight, add D to the tree. Now D is not a fringe vertex but a tree vertex.
Step 8: Note, now node E is not connected, so we will add it in the tree because a minimum spanning tree is one in which all the n nodes are connected with n–1 edges that have minimum weight. So, the minimum spanning tree can now be given as,



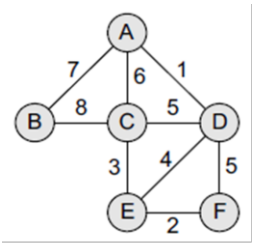

### 5.4.1.2 Kruskal's Algorithm

Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a minimum spanning forest. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees.

Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

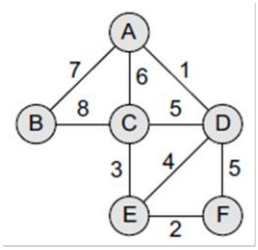
```
Step 1: Create a forest in such a way that each graph is a separate
        tree.
Step 2: Create a priority queue Q that contains all the edges of the
        graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4:      Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two different trees,
        then Add it to the forest (for combining two trees into one
        tree).
        ELSE
             Discard the edge
Step 6: END
```

Krushkals Algorithm




| Edge | A - B | A - C | A - D | B - C | C - D | C - E | D - E | D - F | E - F |
|---|---|---|---|---|---|---|---|---|---|
| Cost | 7 | 6 | 1 | 8 | 5 | 3 | 4 | 5 | 2 |

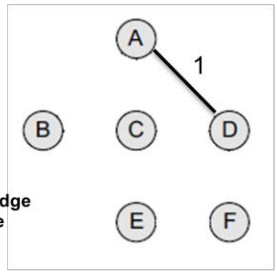
Arrange all edges in ascending order with respect to the cost of edge or weight



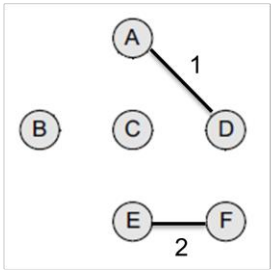

| Edge | A - D | E - F | C - E | D - E | C - D | D - F | A - C | A - B | B - C |
|---|---|---|---|---|---|---|---|---|---|
| Cost | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 |

Yes Yes Yes

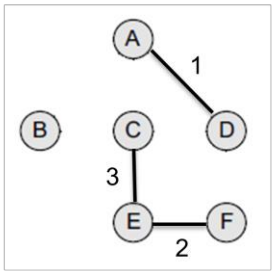
No Cycle with edge in spanning tree

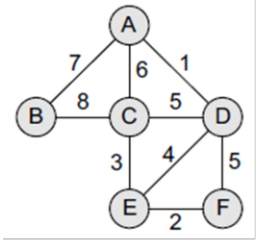



No Cycle with edge in spanning tree




No Cycle with edge in spanning tree

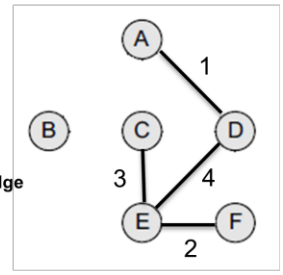



| Edge | A - D | E - F | C - E | D - E | C - D | D - F | A - C | A - B | B - C |
|---|---|---|---|---|---|---|---|---|---|
| Cost | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 |

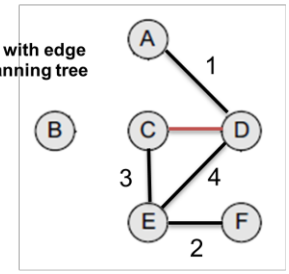
Yes Yes Yes Yes No No

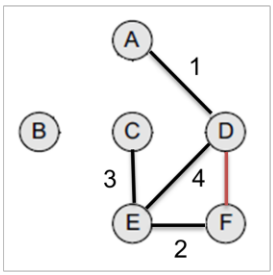
Cycle with edge in spanning tree

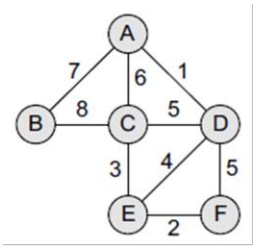



No Cycle with edge in spanning tree

Cycle with edge in spanning tree



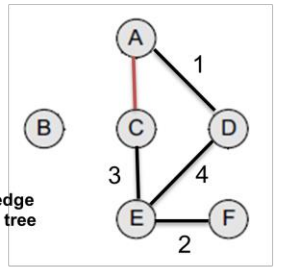



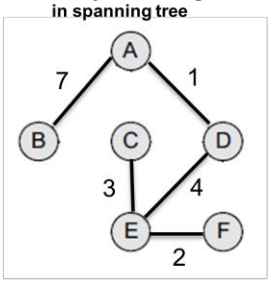

| Edge | A - D | E - F | C - E | D - E | C - D | D - F | A - C | A - B | B - C |
|---|---|---|---|---|---|---|---|---|---|
| Cost | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 |

Yes Yes Yes Yes No No No Yes No

No Cycle with edge in spanning tree




Cycle with edge in spanning tree




If all the vertices are connected then terminate the algorithm

This the minimum cost spanning tree for given graph

Mininum cost of the spanning tree is 1+2+3+4+7 = 17

- Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph.
- The algorithm aims to find a subset of the edges that forms a tree that includes every vertex.
- The total weight of all the edges in the tree is minimized.
- The running time of Kruskal's algorithm can be given as O(E log V), where E is the number of edges and V is the number of vertices in the graph.

### 5.4.2 Dijkstra's Algorithm

Dijkstra's algorithm, given by a Dutch scientist Edsger Dijkstra in 1959, is used to find the shortest path tree. This algorithm is widely used in network routing protocols, most notably IS-IS and OSPF (Open Shortest Path First).
Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities.

**Algorithm**

Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal can mean anything, shortest, cheapest, or fastest. If we start the algorithm with an initial node, then the distance of a node Y can be given as the distance from the initial node to that node. The below figure explains the Dijkstra's algorithm.

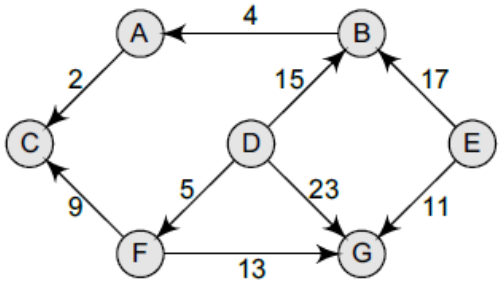
1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with 0, and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.
   (b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)
7. Pick a node not in N that has the smallest label assigned to it and add it to N.

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels: temporary and permanent. Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

1. If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.
2. If the destination node is not labelled, then there is no path from the source to the destination node.

Consider the graph G given in Fig. Taking D as the initial node, execute the Dijkstra's algorithm on it.



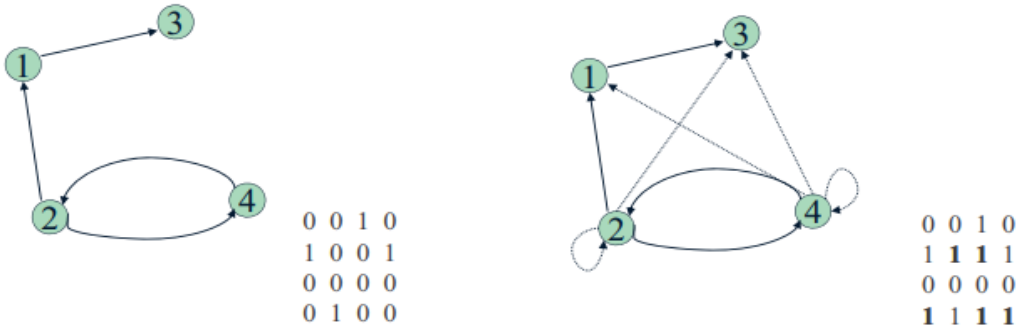

Step 1: Set the label of D = 0 and N = {D}.
Step 2: Label of D = 0, B = 15, G = 23, and F = 5. Therefore, N = {D, F}.
Step 3: Label of D = 0, B = 15, G has been re-labelled 18 because minimum (5 + 13, 23) = 18, C has been re-labelled 14 (5 + 9). Therefore, N = {D, F, C}.
Step 4: Label of D = 0, B = 15, G = 18. Therefore, N = {D, F, C, B}.
Step 5: Label of D = 0, B = 15, G = 18 and A = 19 (15 + 4). Therefore, N = {D, F, C, B, G}.
Step 6: Label of D = 0 and A = 19. Therefore, N = {D, F, C, B, G, A}
Note that we have no labels for node E; this means that E is not reachable from D. Only the nodes that are in N are reachable from B.

The running time of Dijkstra's algorithm can be given as $O(|V|^2+|E|)=O(|V|^2)$ where V is the set of vertices and E in the graph.

### 5.4.3 Warshall's Algorithm

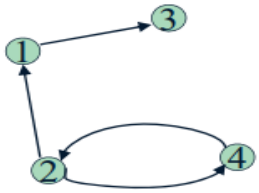
**Transitive closure**

- Transitive Closure it the reachability matrix to reach from vertex u to vertex v of a graph. One graph is given, we have to find a vertex v which is reachable from another vertex u, for all vertex pairs (u, v).
- The final matrix is the Boolean type. When there is a value 1 for vertex u to vertex v, it means that there is at least one path from u to v.



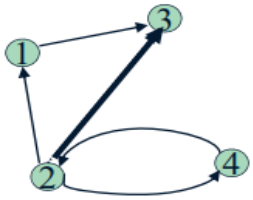

Algorithm:

- Main idea: a path exists between two vertices i, j, iff there is an edge from i to j; or
- there is a path from i to j going through vertex 1; or
- there is a path from i to j going through vertex 1 and/or 2; or
- there is a path from i to j going through vertex 1, 2, and/or 3; or ...
- there is a path from i to j going through any of the other vertices



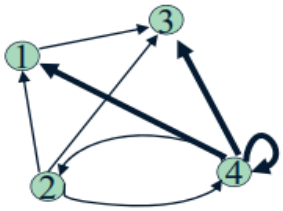
$R_0$

0 0 1 0
1 0 0 1
0 0 0 0
0 1 0 0



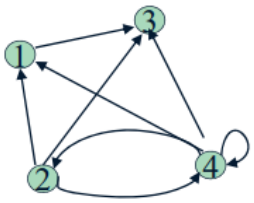
$R_1$

0 0 1 0
1 0 **1** 1
0 0 0 0
0 1 0 0



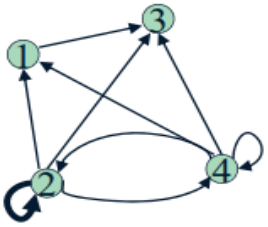
$R_2$

0 0 1 0
1 0 1 1
0 0 0 0
**1** 1 **1** **1**



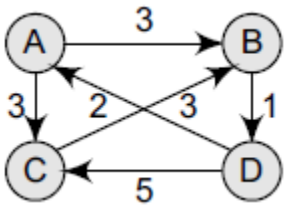
$R_3$

0 0 1 0
1 0 1 1
0 0 0 0
1 1 1 1



$R_4$

0 0 1 0
1 **1** 1 1
0 0 0 0
1 1 1 1




$$\begin{pmatrix} 0 & 3 & 3 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & 5 & 0 \end{pmatrix}$$

$$\mathbf{Q0} = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 9999 & 5 & 9999 \end{bmatrix}$$

$$\mathbf{Q1} = \begin{bmatrix} 9999 & 3 & 3 & 9999 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 9999 \\ 2 & 5 & 5 & 9999 \end{bmatrix}$$

$$\mathbf{Q2} = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$\mathbf{Q3} = \begin{bmatrix} 9999 & 3 & 3 & 4 \\ 9999 & 9999 & 9999 & 1 \\ 9999 & 3 & 9999 & 6 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$

$$\mathbf{Q4} = \mathbf{Q}\begin{bmatrix} 6 & 3 & 3 & 4 \\ 3 & 6 & 6 & 1 \\ 6 & 3 & 9 & 4 \\ 2 & 5 & 5 & 6 \end{bmatrix}$$