**Unit-1:**
*Algorithms:* Introduction, Algorithm Specifications, Recursive Algorithms, Performance Analysis of an algorithm- Time and Space Complexity, Asymptotic Notations.
*Arrays:* Arrays - ADT, Polynomials, Sparse matrices, Strings-ADT, Pattern Matching.
**Objective:**
To develop proficiency in the specification, representation, and implementation of abstract data types and data structures.
**Outcome:** Implementing the concepts of data structure using abstract data type and Evaluate an algorithm by using algorithmic performance and measures.

# Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important **types of algorithms**:

1. Sorting
2. Searching
3. Compression
4. Encoding
5. Fast Fourier transforms
6. Geometric
7. Pattern matching
8. Parsing

Algorithms are classified into 4 categories:

1. Iterative Algorithms
2. Divide and Conquer algorithms
3. Greedy algorithms
4. Back tracking algorithms

**Characteristics of an Algorithm:**

An algorithm should have the following characteristics

- Input − An algorithm should have 0 or more well-defined inputs.
- Output − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- Definiteness − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- Effectiveness –Algorithm should contain required statements only.
- Finiteness − Algorithms must terminate after a finite number of steps.

**Algorithm Specification:**

An algorithm contains 3 specifications.
1. Natural language like English

2. Graphical representation of algorithm called flow chart.
3. Pseudo code convention

Pseudo code convention: In this code convention algorithm will written in two parts.
a. Heading Part
b. Body part

**a. Heading Part:**
**Syntax:** algorithm Name(Parameters list)

**b. Body Part:** This body part statements must be in the following :

1. // Comments
2. {
3. Identifiers
4. <Variable>=<Value>
5. Operators
6. Looping statements
7. If condition Statements
8. Input and Output will write as read and write
9. }

**Performance analysis of an algorithm:**

The performance of a program is the amount of computer memory and time needed to run a program.
**Time Complexity:** The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion. The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

**Space Complexity**: The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

**Complexity of Algorithms:**

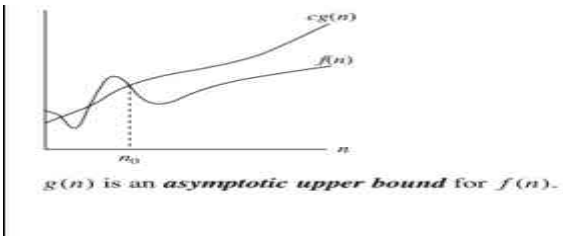
The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size „n“ of the input

data. Mostly, the storage space required by an algorithm is simply a multiple of the data size „n". Complexity shall refer to the running time of the algorithm.
The function f(n), gives the running time of an algorithm, depends not only on the size „n" of the input data but also on the particular data. The complexity function f(n) for certain cases are:
1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.
The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.
Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.

**Asymptotic Notations:**

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:
1. Big–OH (O)
2. Big–OMEGA (Ω ) and
3. THETA(θ)

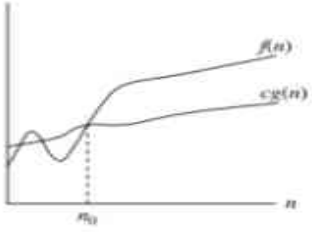
1. Big–OH O (Upper Bound) f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal that of g(n).



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

**2.** . Big–OMEGA (Ω ):

$\Omega(g(n)) = \{f(n)$ : there exist positive constants $c$ and $n_0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.

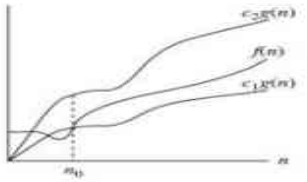


$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

**3.** THETA(θ) :

**Θ-notation**

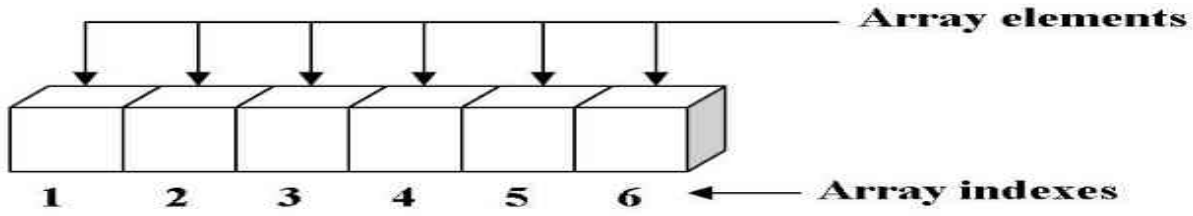
$\Theta(g(n)) = \{f(n)$ : there exist positive constants $c_1$, $c_2$, and $n_0$ such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$ .




$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

# Array ADT:

- The array is an abstract data type (ADT) that holds a collection of elements accessible by an index.




**One-dimensional array with six elements**

- The elements stored in an array can be anything from primitive types such as integers, characters, etc..
- An element is stored in each index and they can be retrieved later by specifying the same index.
- The Array ADT is usually implemented by an Array (Data Structure).

ADT — Operations

- Insertion- Adds an element at the given index.
- Deletion - Deletes an element at the given index
- Traverse- print all the array elements one by one
- Search- Searches an element using the given index or by the value
- Update- Updates an element at the given index

**Implementation of Array ADT:**

```
#include<stdio.h>
        int a[5],i,n,j,index,value;


        void insert()
        {
```

```
        printf("Enter 5 integers: ");
        for(int i = 0; i < 5; ++i)
        {
                scanf("%d", &a[i]);
        }
}
void delete()
{
        printf("\nEnter an element to delete\n");
scanf("%d",&n);
for(i=0;i<5;i++)
{
                if(a[i]==n)
                {
                for(j=i;j<=4;j++)
                        {
                if(j==4)
                {
                   a[j]=0;
                 }
                 else
                 {
                    a[j]=a[j+1];
                 }
                }
                break;
                }
        }
}
void display()
{
        for(i=0;i<5;i++)
{
                printf("%d ",a[i]);
        }
}

void search()
{
        printf("\nEnter an element to search\n");
scanf("%d",&n);
for(i=0;i<5;i++)
{
                if(a[i]==n)
                {
        printf("\nthe searching element %d found at %d index",n,i);
                }
        }
}
```

```
        void update()
        {
                printf("enter the index number and value which u want to update:");
                scanf("%d%d",&index,&value);
                a[index]=value;
        }
int main()
{
        int ch;
        do
        {
                printf("\n**** MENU ****");
      printf("\n1:insert()\n2:delete()\n3:display()\n4:search()\n5:update()\n");
                printf("\nEnter Your Choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: insert();
                                break;
                        case 2: delete();
                                break;
                        case 3: display();
                                break;
                        case 4: search();
                                break;
                        case 5: update();
                                break;
                        default: printf("choose correct option");

                }
        }while(ch!=0);
}
```

## Array ADT applications:

1. Polynomial Expression-Addition and Multiplication
2. Sparse Matrix

1. **Polynomial Expression:**

- A polynomial is an expression representing a mathematical sum of many terms.
- Each term has a number called the coefficient, a variable and a power of the variable called the exponent.

$$ax^n+ax^{n-1}+ax^{n-2}+\ldots\ldots+a$$

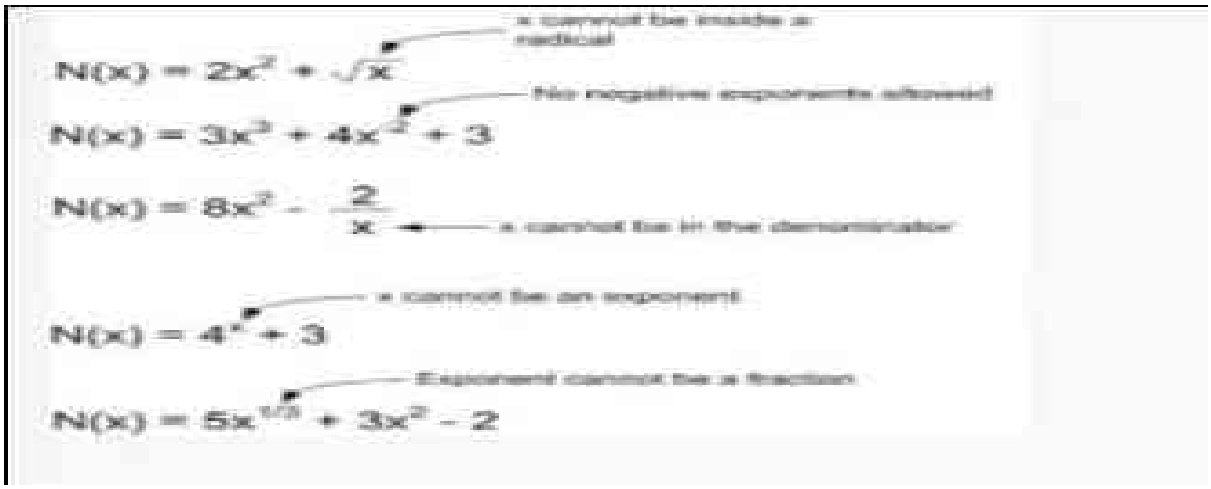
- The largest exponent is the polynomial's degree.

| Polynomial | Degree |
|---|---|
| $x^5 + 10x^2 + 7x - 15$ | 5 |
| $2x^2 + 3x + 5$ | 2 |
| $10x^5 + 5x^3 + 3x^2 + 20x$ | 5 |
| $2x^3 - 4x^2 + 6x - 10$ | 3 |

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself.
- Additional terms having equal exponent is possible one.
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent.

Not a polynomial:




- A Polynomial can be represented in two way:

1. Array Representation
2. Linked list Representation

1. Array Representation:

There may arise some situation where you need to evaluate many polynomial expressions and perform basic arithmetic operations like addition and subtraction with those numbers. For this, you will have to get a way to represent those polynomials. The simple way is to represent a polynomial with degree 'n' and store the coefficient of n+1 terms of the polynomial in the array. So every array element will consist of two values:

- Coefficient and
- Exponent

**Implementation of polynomial representation in arrays:**

```
#include<stdio.h>
#include<math.h>
struct poly
{
   float coeff;
   int exp;
```

```
};
//declaration of polynomials
struct poly a[50],b[50],c[50],d[50];
int main()
{
  int i;
 int deg1,deg2;        //stores degrees of the polynomial
  int k=0,l=0,m=0;
  printf("Enter the highest degree of polynomial1:");
  scanf("%d",&deg1);        //taking polynomial terms from the user
  for(i=0;i<=deg1;i++)
  {
     //entering values in coefficient of the polynomial terms
     printf("\nEnter the coeff of x^%d :",i);
     scanf("%f",&a[i].coeff);
     //entering values in exponent of the polynomial terms
     a[k++].exp = i;
  }
  //taking second polynomial from the user
  printf("\nEnter the highest degree of polynomial2:");
  scanf("%d",&deg2);
  for(i=0;i<=deg2;i++)
  {
     printf("\nEnter the coeff of x^%d :",i);
     scanf("%f",&b[i].coeff);
     b[l++].exp = i;
  }
  //printing first polynomial
  printf("\nExpression 1 = %.1f",a[0].coeff);
  for(i=1;i<=deg1;i++)
  {
     printf("+ %.1fx^%d",a[i].coeff,a[i].exp);
  }
  //printing second polynomial
  printf("\nExpression 2 = %.1f",b[0].coeff);
  for(i=1;i<=deg2;i++)
  {
       printf("+ %.1fx^%d",b[i].coeff,b[i].exp);
  }
  //Adding the polynomials
  if(deg1>deg2)
  {
     for(i=0;i<=deg2;i++)
     {
        c[m].coeff = a[i].coeff + b[i].coeff;
           c[m].exp = a[i].exp;
        m++;
     }
     for(i=deg2+1;i<=deg1;i++)
     {
```

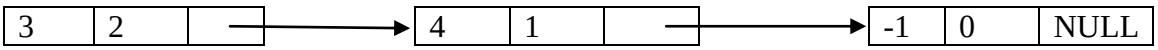
```
                c[m].coeff = a[i].coeff;
                c[m].exp = a[i].exp;
                m++;
            }
        }
        else
        {
            for(i=0;i<=deg1;i++)
            {
                c[m].coeff = a[i].coeff + b[i].coeff;
                c[m].exp = a[i].exp;
                m++;
            }
            for(i=deg1+1;i<=deg2;i++)
            {
                c[m].coeff = b[i].coeff;
                c[m].exp = b[i].exp;
                 m++;
            }
        }
        //printing the sum of the two polynomials
        printf("\nExpression after addition  = %.1f",c[0].coeff);
        for(i=1;i<m;i++)
        {
            printf("+ %.1fx^%d",c[i].coeff,c[i].exp);
        }
    return 0;
}
```

Polynomial representation in linked list:

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

- The exponent part
- The coefficient part

Ex: $3x^2+4x-1$




## 2. Sparse Matrix:

• A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m*n matrix.
• Sometimes it happens when a matrix has zero values is more than NON-ZERO values.

- The matrix which has a greater number of zero elements in comparison to the non-zero elements is known as a sparse matrix.

4X4 Matrix

| | | | |
|---|---|---|---|
| 0 | 0 | 3 | 0 |
| 0 | 0 | 0 | 8 |
| 1 | 0 | 3 | 0 |
| 0 | 0 | 7 | 0 |

Sparse Matrix

- So if we calculate the space
  - ➢ Integer value takes 2 bytes
  - ➢ Total space taken by 4 X 4 matrix is 4 X 4 X 2 = 32 bytes.

  where only 5 X 2 = 10 bytes are in use which has a non-zero elements and the rest 32 – 10 = 22 bytes are wastage of memory.
- When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix.
- We can save memory and computational time by the following representation
  - ➢ A sparse matrix can be represented in two way:

1. Array Representation(triplet representation)
2. Linked list Representation

1. **Array Representation:**

- In array representation of a sparse matrix, we consider only non-zero elements along with their row and column index numbers.
- The 2d array can be used to represent a sparse matrix in which there are three columns named as:

1. Row: It is an index of a row where a non-zero element is located.
2. Column: It is an index of the column where a non-zero element is located.
3. Value: The value of the non-zero element is located at the index (row, column).

- The sparse matrix is represented using triplets, i.e., row, column, and value because of that we called it is triplet representation.

| Row | Column | Value |
|---|---|---|
| 0 | 2 | 3 |
| 1 | 3 | 8 |
| 2 | 0 | 1 |
| 2 | 2 | 3 |
| 3 | 2 | 7 |

**Implementation of sparse matrix representation in arrays:**

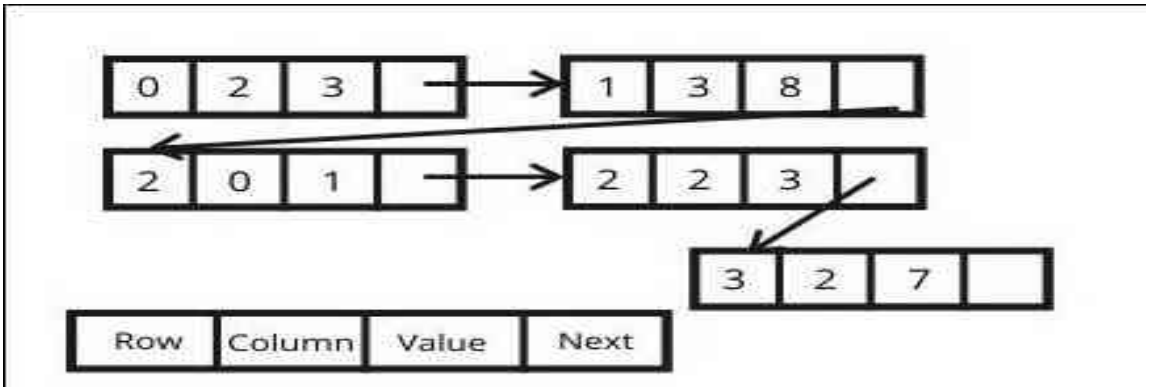
```
#include<stdio.h>
int main()
{
int a[50][50],r,c,i,j;
printf("Enter the no. of rows and columns of matrix:");
scanf("%d%d",&r,&c);

printf("Enter Sparse Matrix:\n ");
for(i=0;i<r;i++)
```

```
{
        for(j=0;j<c;j++)
        {
                printf("a[%d][%d]:",i,j);
                scanf("%d",&a[i][j]);
        }
}
 printf("\nPrinting the Sparse Matrix:");
   for(i=0;i<r;i++)
   {
      printf("\n");
      for (j=0;j<c;j++)
      {
         printf("%d\t",a[i][j]);
      }
   }
printf("\nThe 3 tuple represenation of the sparse matrix: ");
printf("\nR C V");
for(i=0;i<r;i++)
{
        for(j=0;j<c;j++)
        {
                if(a[i][j]!=0)
                {
                printf("\n%d %d %d",i,j,a[i][j]);
                }
        }
}
printf("\n");
return 0;
}
```

**2. Linked list Representation:**

- In linked list representation, we use a linked list to represent a sparse matrix. Each node of the linked list has four fields.
- These four fields are defined as:

1. Row: Row keeps row index of a non-zero element
2. Column: Column keeps column index of a non-zero element
3. Value: non zero element located at (row,column) index
4. Next node: Next node, stores the address of the next node




**Implementation of sparse matrix representation in linked list:**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int row;
int col;
int value;
struct node* next;
};
struct node *head=NULL,*last=NULL;
void display()
{
   struct node *temp=head;
   printf("\n");
   while(temp!=NULL)
   {
    printf("[%d,%d,%d,%p]",temp->row,temp->col,temp->value,temp->next);
         temp=temp->next;
         if(temp!=NULL)
         {
         printf("-->");
         }
   }
}
int main()
{

int a[50][50],r,c,i,j;
printf("Enter the no. of rows and columns of matrix:");
scanf("%d%d",&r,&c);

printf("Enter Sparse Matrix:\n ");
for(i=0;i<r;i++)
{
         for(j=0;j<c;j++)
         {
                  printf("a[%d][%d]:",i,j);
                  scanf("%d",&a[i][j]);
         }
```

```
}
 printf("\nPrinting the Sparse Matrix:");
   for(i=0;i<r;i++)
   {
      printf("\n");
      for (j=0;j<c;j++)
      {
         printf("%d\t",a[i][j]);
      }
   }

for(i=0;i<r;i++)
{
        for(j=0;j<c;j++)
        {
                if(a[i][j]!=0)
                {
                struct node *temp;
                temp=(struct node*)malloc(sizeof(struct node));
                temp->row=i;
                temp->col=j;
                temp->value=a[i][j];
                temp->next=NULL;
                if(head==NULL)
                {
                head=temp;
                last=head;
                }
                else
                {
                last->next=temp;
                last=temp;
                }
                }
        }
}
printf("\nThe linked list represenation of the sparse matrix: ");
display();
printf("\n");
return 0;
}
```

# String ADT:

- A string is a sequence of characters stored in an array(i.e., one dimensional array).
- Every string terminated with a special character called null character('\0').
- Each character in the array occupies one byte of memory, and the last character must always be 0.

- The termination character ('\0') is important in a string since it is the only way to identify where the string ends.
- There are two ways to declare a string in c language.

1. By char array
2. By string literal

1. **By char array:** we can declare a string by char array in C language by the following ways
   char ch[5]={'H', 'E', 'L', 'L', 'O', '\0'};

As we know, array index starts from 0, so it will be represented as in the figure given below.

- While declaring string, size is not mandatory. So we can write the above code as given below:

char ch[]={'H', 'E', 'L', 'L', 'O', '\0'};

2. **By string literal:**

char ch[]="Hello";
In such case, '\0' will be appended at the end of the string by the compiler.
Difference between char array and string literal:

1. We need to add the null character '\0' at the end of the array by ourself in char array whereas, it is appended internally by the compiler in the case of the string literal.
2. The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

- We can read a string from the keyboard in two ways

1. Using scanf() method - reads single word
2. Using gets() method - reads a line of text

- Using scanf() method we can read only one word of string. We use %s to represent string in scanf() and printf() methods.
- The puts() function is very much similar to printf() function.
- The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function.

Operations on string:

1. length
2. concatenation
3. copy
4. compare

**Implementation of String ADT:**

```
#include<stdio.h>
#include<string.h>
int main()
{
char a[30],b[30],c[30];
printf("Enter first string:");
scanf("%s",a);
printf("Enter second string:");
scanf("%s",b);
printf("Enter third string:");
scanf("%s",c);
```

```
printf("\nEntered string is %s",a);
printf("\nEntered string is %s",b);
printf("\nEntered string is %s",c);
int length=strlen(a);
printf("\nLength of the string is %d",length);

printf("\nafter concatenation, the string is %s",strcat(a,b));

printf("\nafter copy operation, the string is %s",strcpy(a,b));


printf("\nComparing the two strings are %d",strcmp(b,c));
}
```

## Pattern Matching:

- Pattern matching is the way of checking a series of pattern or a sequence of digits or string with some other pattern and find out if it matches or not, in pattern recognition, the match usually has to be exact.
- A pattern can be a series of digits, a string arranged in an order. The order is really important in case of pattern matching.

**Implementation of Pattern matching program:**

```
#include <stdio.h>
#include <string.h>
int match(char [], char []);

int main() {
  char a[100], b[100];
  int position;

  printf("Enter some text\n");
  gets(a);

  printf("Enter a string to find\n");
  gets(b);

  position = match(a, b);

  if (position != -1) {
    printf("Found at location: %d\n", position + 1);
  }
  else {
    printf("Not found.\n");
  }

  return 0;
}
```

```
int match(char text[], char pattern[]) {
  int c, d, e, text_length, pattern_length, position = -1;

  text_length    = strlen(text);
  pattern_length = strlen(pattern);

  if (pattern_length > text_length) {
    return -1;
  }

  for (c = 0; c <= text_length - pattern_length; c++) {
    position = e = c;

    for (d = 0; d < pattern_length; d++) {
      if (pattern[d] == text[e]) {
        e++;
      }
      else {
        break;
      }
    }
    if (d == pattern_length) {
      return position;
    }
  }
  return -1;
}
```

**Unit-2:**
*Stacks and Queues***:** Stacks, Stacks using Arrays, Stacks using dynamic arrays, Evaluation of Expressions – Evaluating Postfix Expression, Infix to Postfix.
*Queues:* Queues ADT, operations, Circular Queues, Applications
**Objective:**
To introduce various linear data Structures such as stacks, queues and their applications
**Outcome:** Implement linear data structures such as stacks, queues to solve various computing problems.

# Stack and Queue

Introduction to **Data Structures**:

A data structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used. The choice of the data structure begins from the choice of an abstract data type (ADT). A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

1. First, how the data will be stored, and

2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.

Classification of Data Structures: Data structures can be classified as

1. Linear Data structures
2. Non- Linear Data structures

1. Linear Data Structure:

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

Operations applied on linear data structure: The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

For example Stack, Queue, Tables, List, and Linked Lists.

2. Non-linear Data Structure:

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

Operations applied on non-linear data structures: The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element For example Tree, Decision tree, Graph and Forest

**Abstract Data Type:**

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called information hiding. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.

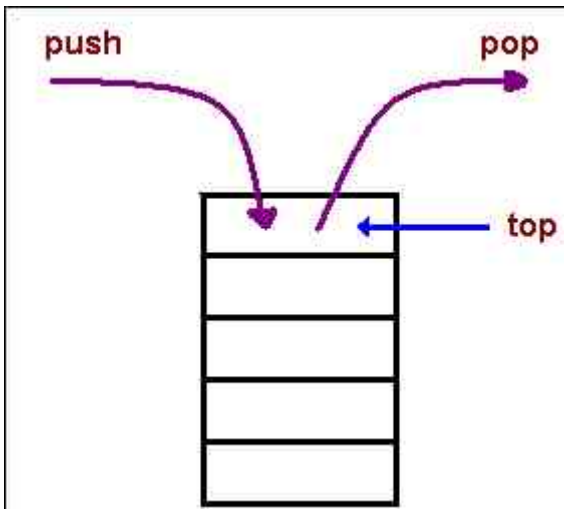
1. **Linear Data structures**: Linear data structures are Stack, Queue and Linked list.

**Stack:**

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. Push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think

of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.




Stack (ADT) Data Structure: Stack is an Abstract data structure (ADT) works on the principle Last In First Out (LIFO). The last element add to the stack is the first element to be delete. Insertion and deletion can be takes place at one end called TOP. It looks like one side closed tube.

- The add operation of the stack is called **push** operation.
- The delete operation is called as **pop** operation.
- Push operation on a full stack causes **stack overflow**.
- Pop operation on an empty stack causes **stack underflow**.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | | 4 | | 4 | | 4 | |
| 3 | | 3 | | 3 | | 3 | 40 |
| 2 | | 2 | | 2 | | 2 | 30 |
| 1 | | 1 | | 1 | 20 | 1 | 20 |
| 0 | | 0 | 10 | 0 | 10 | 0 | 10 |

Top=-1, stack is empty | top=0 | top=1 | top=3

Operations of stack: There are two operations applied on stack they are

1. push
2. pop.

While performing push & pop operations the following test must be conducted on the stack.

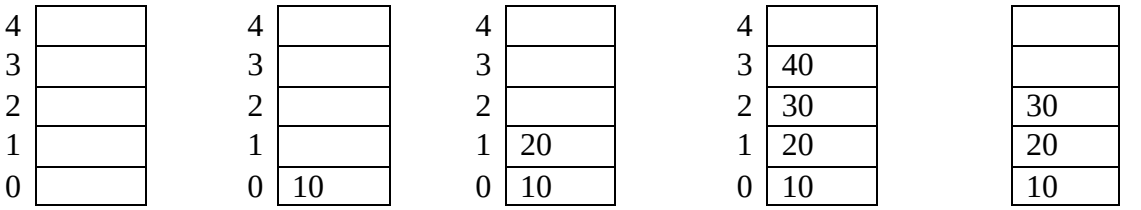
1) Stack is empty or not
2) Stack is full or not

**Push**: Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

**Pop**: Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

Representation of a Stack using Arrays:

Let us consider a stack with 5 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition. When a element is added to a stack, the operation is performed by push().




Top=-1 push(10) Top=0 push(20) top=1 push(30) push(40) Top=3 pop() top=2

1. PUSH: if (top==MAX), display Stack overflow else reading the data and making stack [top] =data and incrementing the top value by doing top++.
2. POP: if (top==-1), display Stack underflow else printing the element at the top of the stack and decrementing the top value by doing the top.

**Algorithms of stack operations:**

These algorithms are written in pseudo code convention.

- **Algorithm for Empty condition**

```
algorithm isEmpty()
{
top;
if(top==-1) then
write stack is empty;
else
```

```
write stack is not empty;
end if;
}
```

- **Algorithm for Full condition**

```
algorithm isFull()
{
top, maxsize;
if(top==maxsize-1) then
write stack is full;
else
write stack is not full;
end if;
}
```

- **Algorithm for Push Operation**

```
algorithm push(item)
{
top, a[maxsize];
if(top==maxsize-1) then
write stack overflow;
else
top=top+1;
a[top]=item;
}
```

- **Algorithm for Pop Operation**

```
algorithm pop()
{
top, item;
if(top==-1) then
write stack underflow;
else
write item=stack[top];
top=top-1;
return item;
}
```

- **Implementation of stack using array**

```
#include<stdio.h>
        int top=-1,a[5];
```

```
int isEmpty()
{
        if(top==-1)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
int isFull()
{
        if(top==4)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}
void push(int item)
{

        if(isFull())
        {
                printf("stack overflow");
        }
        else
        {
                top++;
                a[top]=item;
        }
}
void pop()
{
        if(isEmpty())
        {
                printf("stack underflow");
        }
        else
        {
                printf("The popped element is %d",a[top]);
                top--;
        }
}
void display()
{
        if(top>=0)
```

```
                {
                   printf("Elements in the stack are:");
                        for(int i=top;i>=0;i--)
                        {
                                printf("\n%d",a[i]);
                        }
                }
                else
                {
                        printf("stack is empty");

                }
        }
        void count()
        {
                printf("\nNumber of elements in the stack are: %d",top+1);
        }
        void peek()
        {
    if(top==-1)
                {
    printf("stack is empty");
                }
                else
                {
                  printf("The top element is %d",a[top]);
                }
}
        void search()
        {
                int index;
                if(top==-1)
                {
    printf("stack is empty");
                }
                else
                {
                printf("Enter index number u want to display:");
                scanf("%d",&index);
                printf("\nthe value in the %d index is %d",index,a[index]);
                }
        }
        void change()
        {
                int index,value;

                if(top==-1)
                {
    printf("stack is empty");
                }
```

```
                else
                {
                printf("enter the index number and value which u want to change:");
                scanf("%d%d",&index,&value);
                a[index]=value;
                }
        }
int main()
{
        int ch,item,index,value;
        do
        {
                printf("\n**** MENU ****");

printf("\n1:isEmpty()\n2:isFull()\n3:push()\n4:pop()\n5:count()\n6:display()\n7:peek()\n8:sea
rch()\n9:change()");
                printf("\nEnter Your Choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: if(isEmpty())
                                        printf("stack is empty\n");
                                else
                                        printf("stack is not empty\n");
                                break;
                        case 2: if(isFull())
                                        printf("stack is full\n");
                                else
                                        printf("stack is not full\n");
                                break;
                        case 3: printf("enter an item to push:");
                                scanf("%d",&item);
                                push(item);
                                break;
                        case 4: pop();
                                break;
                        case 5: count();
                                break;
                        case 6:     display();
                                break;
                        case 7:         peek();
                                break;
                        case 8:     search();
                                break;
                        case 9:  change();
                                break;
                        default: printf("choose correct option");

                }
        }while(ch!=0);
```

```
}
```

- **Implementation of stack using dynamic array**

```
#include<stdio.h>
#include<stdlib.h>
        int *a=NULL;
        int top=-1,size;
        void create_st()
        {
        printf("\nEnter size of the array");
        scanf("%d",&size);
        a=(int*)malloc(size*sizeof(int));
        }
        int isEmpty()
        {
                if(top==-1)
                {
                        return 1;
                }
                else
                {
                        return 0;
                }
        }
        int isFull()
        {
                if(top==size-1)
                {
                        return 1;
                }
                else
                {
                        return 0;
                }
        }
        void push(int item)
        {

                if(isFull())
                {
                        printf("stack overflow");
                }
                else
                {
                        top++;
                        *(a+top)=item;
                }
```

```
        }
        void pop()
        {
                if(isEmpty())
                {
                        printf("stack underflow");
                }
                else
                {
                        printf("The popped element is %d",*(a+top));
                        top--;
                }
        }
        void display()
        {
                if(top>=0)
                {
                   printf("Elements in the stack are:");
                        for(int i=0;i<=top;i++)
                        {
                                printf("\n%d",*(a+i));
                        }
                }
                else
                {
                        printf("stack is empty");

                }
        }
        /*void count()
        {
                printf("\nNumber of elements in the stack are: %d",top+1);
        }
        void peek()
        {
    if(top==-1)
                {
    printf("stack is empty");
                }
                else
                {
                  printf("The top element is %d",a[top]);
                }
}
        void search()
        {
                int index;
                if(top==-1)
                {
    printf("stack is empty");
```

```
                }
                else
                {
                printf("Enter index number u want to display:");
                scanf("%d",&index);
                printf("\nthe value in the %d index is %d",index,a[index]);
                }
        }
        void change()
        {
                int index,value;

                if(top==-1)
                {
      printf("stack is empty");
                }
                else
                {
                printf("enter the index number and value which u want to change:");
                scanf("%d%d",&index,&value);
                a[index]=value;
                }
        }*/
int main()
{
        int ch,item,index,value;
        do
        {
                printf("\n**** MENU ****");

printf("\n1:isEmpty()\n2:isFull()\n3:push()\n4:pop()\n6:count()\n5:display()\n7:peek()
\n8:search()\n9:change()\n10:create()");
                printf("\nEnter Your Choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: if(isEmpty())
                                        printf("stack is empty\n");
                                else
                                        printf("stack is not empty\n");
                                break;
                        case 2: if(isFull())
                                        printf("stack is full\n");
                                else
                                        printf("stack is not full\n");
                                break;
                        case 3: printf("enter an item to push:");
                                scanf("%d",&item);
                                push(item);
                                break;
```

```
                    case 4: pop();
                            break;
                    case 5:  display();
                            break;
                /*  case 6: count();
                            break;

                    case 7:  peek();
                            break;
                    case 8:  search();
                            break;
                    case 9:  change();
                            break;*/
                    case 10: create_st();
                          break;
                    default: printf("choose correct option");

                }
            }while(ch!=0);
        }
```

**Stack Applications:**

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to **evaluate a postfix expression.**
3. Stack is used to **convert an infix expression into postfix/prefix form.**

**Analysis of Stack Operations:**

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

Push Operation : O(1)
Pop Operation : O(1)
Top Operation : O(1)
Search Operation : O(n)

The time complexities for push() and pop() functions are O(1) because we always have to insert or remove the data from the top of the stack, which is a one step process.

## Infix- to Postfix expression Transformation:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

| Symbol | Postfix string | Stack |
|---|---|---|
| A | A | |
| + | A | + |
| B | A B | + |
| * | A B | + * |
| C | ABC | +* |
| - | ABC*+ | - |
| D | ABC*+D | - |
| / | ABC*+D | -/ |
| E | ABC*+DE | -/ |
| * | ABC*+DE/ | -* |
| H | ABC*+DE/H | -* |
| Empty | ABC*+DE/H*- | Popped all operators if expr is empty |

**Algorithm for conversion of infix to postfix expression:**

```
//Algorithm for conversion of infix to postfix expression

SET i to 0
GET infix expression from user into input_array
SET var with input_array[i]
CALL createStack
WHILE var != end of string
  IF var equals to '(' THEN
     CALL pushIntoStack (stack, var)
  ELSE IF var is a number THEN
      PRINT var
  ELSE IF var is an arithmetic operator THEN
      CALL pushIntoStack (stack, var)
      ELSE IF var equals to ')' THEN
    WHILE stackTop != '('
      IF stackTop is an arithmetic operator THEN
        PRINT stackTop
        popFromStack (stack)
      ENDIF
        ENDWHILE
   popFromStack (stack)
  ENDIF
```

```
  SET var with input_expression[INCREMENT i]
ENDWHILE
CALL freeStack (stack)
```

- **Implementation of conversion of infix to postfix expression**

```
#include<stdio.h>
char stk[20];
int top=-1;

int isEmpty()
{
if(top==-1)
{
return 1;
}
else
{
return 0;
}
}
int isFull()
{
if(top==19)
{
return 1;
}
else
{
return 0;
}
}
void push(int item)
{
if(isFull())
{
printf("stack overflow");
}
else
{
top++;
stk[top]=item;
}
}
```

```
char pop()
{
if(isEmpty())
{
printf("stack underflow");
}
else
{
return stk[top--];
}
}
char peep()
{
return stk[top];
}
int pri(char opr)
{
switch(opr)
{
case '(':return 0;
case ')':return 0;
case '+':return 1;
case '-':return 1;
case '*':return 2;
case '/':return 2;
case '%':return 2;
case '$':return 3;
}
}

int main()
{
int i,j=0;
char ch;
char infix[100],postfix[100];
printf("enter a valid infix expression:");
scanf("%s",infix);
i=0;
while(infix[i]!='\0')
{
                if(infix[i]=='(')
                {
                        push(infix[i]);
                }
```

```
        else if(infix[i]>='a' && infix[i]<='z')
        {
                postfix[j]=infix[i];
                j++;
        }
        else if(infix[i]>='A' && infix[i]<='Z')
        {
                postfix[j]=infix[i];
                j++;
        }
        else if(infix[i]=='+' || infix[i]=='-' || infix[i]=='*'
||infix[i]=='/'||infix[i]=='%'||infix[i]=='$')
        {

                if(top==-1)
                {
                        push(infix[i]);
                }
                else
                {

                        while(peep()=='+'||peep()=='-
'||peep()=='*'||peep()=='/'||peep()=='%'||peep()=='$'||peep()=='('||peep()==')')
                        {
                                if(pri(peep())>=pri(infix[i]))
                                {
                                postfix[j]=pop();
                                j++;
                                }
                                else
                                {
                                break;
                                }
                        }
                        push(infix[i]);
                }
        }
        else if(infix[i]==')')
        {
                while((ch=pop())!='(')
                {
                postfix[j]=ch;
                j++;
                }
```

```
                    }

        i++;
        }
        while(top>-1)
        {
        postfix[j]=pop();
        j++;
        }
        postfix[j]='\0';
        printf("%s",postfix);
        }
```

**Complexity:**

Time Complexity: O(n)
Space Complexity: O(n)

## Evaluation of Postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

| Symbol | Operand 2 | Operand 1 | Value | Stack | Remarks |
|---|---|---|---|---|---|
| 6 | | | | 6 | |
| 5 | | | | 6 5 | |
| 2 | | | | 6 5 2 | |
| 3 | | | | 6 5 2 3 | The first four symbols are placed on the stack. |
| + | 2 | 3 | 5 | 6 5 5 | Next a „+“ is read, so 3 and 2 are popped from the stack and their sum 5, is pushed |
| 8 | | | | 6 5 5 8 | Next 8 is pushed |
| * | 5 | 8 | 40 | 6 5 40 | Now a „*“ is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed |

| | | | | | |
|---|---|---|---|---|---|
| + | 40 | 5 | 45 | 6 45 | Next, a „+“ is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed |
| 3 | | | | 6 45 3 | 3 is pushed |
| + | 45 | 3 | 48 | 6 48 | Next, „+“ pops 3 and 45 and pushes 45 + 3 = 48 is pushed |
| * | 6 | 48 | 288 | 288 | Finally, a „*“ is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed |

**Algorithm of evaluation of postfix expression**

```
//Algorithm for Postfix expressin Evaluation

1. algorithm postfixeval(postfix[],i,top)
2.{
3. while(postfix[i]!='\0') do
4.if(postfix[i]>='0'&&postfix[i]<='9') then
        x=postfix[i]-48;
        push(x);
5.else  if(postfix[i]=='+'||postfix[i]=='-'||postfix[i]=='*'||postfix[i]=='/'||postfix[i]=='%')
then
        a=pop();
        b=pop();
6.switch(postfix[i])
        case '+':
                c=b+a;
                push(c);
                break;
        case '-':
                c=b-a;
                push(c);
                break;
        case '*':
                c=b*a;
                push(c);
                break;
        case '/':
                c=b/a;
                push(c);
```

```
                    break;
            case '%':
                    c=b%a;
                    push(c);
                    break;

        end if;
        i++;
        end while;
        write top;
        }
```

- **Implementation of evaluation of postfix expression**

```
#include<stdio.h>
int top=-1;
int stk[20];

int isEmpty()
{
if(top==-1)
{
return 1;
}
else
{
return 0;
}
}
int isFull()
{
if(top==19)
{
return 1;
}
else
{
return 0;
}
}
void push(int value)
{
if(isFull())
{
printf("stack overflow");
}
```

```
else
{
top++;
stk[top]=value;
printf("pushing element is %d\n",stk[top]);
}
}
int pop()
{
printf("the popped element is %d\n",stk[top]);
return stk[top--];
}
int peep()
{
return stk[top];
}
int main()
{
int i=0,x,a,b,c;
char postfix[100];
printf("enter a valid postfix Expression:");
scanf("%s",postfix);

while(postfix[i]!='\0')
{
if(postfix[i]>='0'&&postfix[i]<='9')
{
x=postfix[i]-48;
push(x);
}
else if(postfix[i]=='+'||postfix[i]=='-'||postfix[i]=='*'||postfix[i]=='/'||postfix[i]=='%')
{
a=pop();
b=pop();
switch(postfix[i])
{
case '+':
                    c=b+a;
                    push(c);
                    break;
case '-':
                    c=b-a;
                    push(c);
                    break;
```

```
case '*':
                c=b*a;
                push(c);
                break;
case '/':
                c=b/a;
                push(c);
                break;
case '%':
                c=b%a;
                push(c);
                break;
}
}
i++;
}
printf("The Answer is %d",peep());
}
```

**Complexity:**

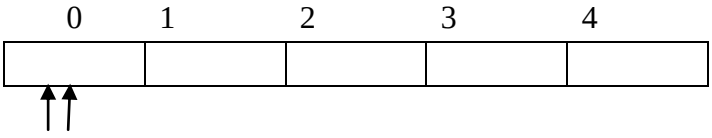
Time Complexity: O(n)
Space Complexity: O(n)

## Queue:

A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

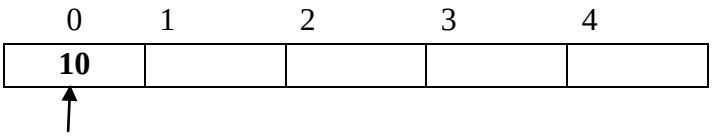
Representation of a Queue using Array:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



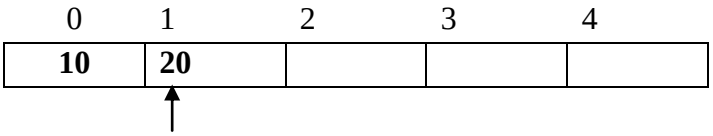

Q u e u e E mp t y F RO NT = REA R = -1

Now, insert 10 to the queue. Then queue status will be:



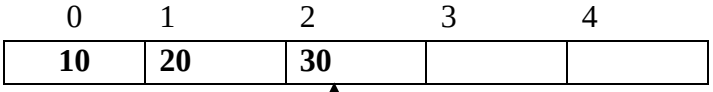
front=-1 rear rear=rear+1=-1+1=0

Now, insert 20 to the queue. Then queue status will be:



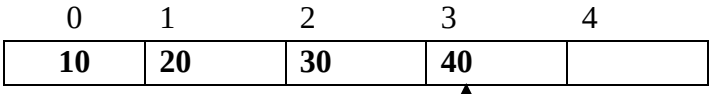

front=-1
rear=rear+1=-0+1=1 rear

Now, insert 30 to the queue. Then queue status will be:



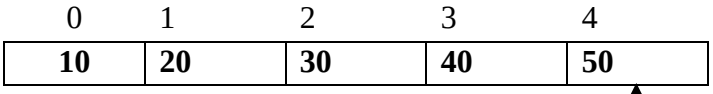

front=-1
rear=rear+1=-1+1=2 rear

Now, insert 40 to the queue. Then queue status will be:



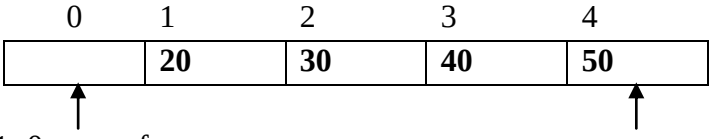

rear=rear+1=-2+1=3 rear

Now, insert 50 to the queue. Then queue status will be:



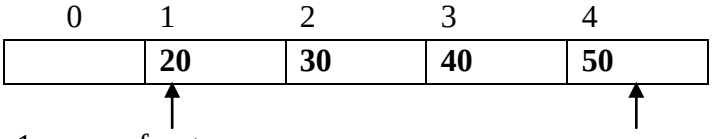

rear=rear+1=-3+1=4 rear

Now, delete an element. The element deleted is the element at the front of the queue, if we increment the front. So the status of the queue is:



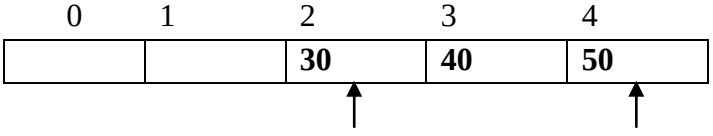

front=front+1=-1+1=0 front rear

Next another element deleted after incrementing the front




front=front+1=0+1=1 front rear

Next another element deleted after incrementing the front



front=front+1=1+1=2 front rear

**Queue operations:**

In queue ,inserting an element is called enqueue operation and removing element is called dequeue operation.

**Algorithms of Queue operations :**

- Algorithm for **empty condition**

```
algorithm isEmpty()
{
read front=-1,rear=-1;
if(front==-1 && rear==-1) then
write Queue is empty;
else
write Queue is not empty;
end if;
}
```

- Algorithm for **Full condition**

```
algorithm isFull(rear,maxsize)
{
if(rear==maxsize-1) then
write Queue is full;
else
write Queue is not full;
end if;
}
```

- Algorithm for **Enqueue operation**

```
algorithm Enqueue(value)
{
q[maxsize],rear,front;
if(rear=maxsize-1) then
write queue is full;
else if(front==-1 && rear==-1) then
read rear=0,front=0;
q[rear]=value;
else
rear++;
end if;
}
```

- Algorithm for **Dequeue operation**

```
algorithm Dequeue()
{
q[],front,rear,x;
```

```
if(front==-1 && rear==-1) then
        write queue is empty;
else if(front==rear) then
        read x=q[front];
        read front=-1;
        read rear=-1;
        return x;
else
        read x=q[front];
        front++;
        return x;
end if;
}
```

- **Implementation of Queue**

```
#include<stdio.h>
int front=-1,i;
int rear=-1;
int q[5];
int isEmpty()
{
if(front==-1 && rear==-1)
{
return 1;
}
else
{
return 0;
}
}
int isFull()
{
if(rear==4)
{
return 1;
}
else
{
return 0;
}
}
void enqueue(int value)
{
if(isFull())
{
```

```
printf("Queue is full");
return;
}
else if(isEmpty())
{
rear=0;
front=0;
}
else
{
rear++;
}
q[rear]=value;
}
void dequeue()
{
if(isEmpty())
{
printf("Queue is empty");
}
else if(front==rear)
{
printf("dequeued value: %d",q[front]);
front=-1;
rear=-1;
}
else
{
printf("dequeued value: %d",q[front]);
front++;
}
}
int count()
{
if(rear==-1&&front==-1)
{
return 0;
}
else
{
return (rear-front+1);
}
}
void display()
```

```
{
 if(isEmpty())
{
printf("Queue is empty");
}
else
{
printf("All valus in the queue are: ");
for(int i=front;i<=rear;i++)
{
printf("%d ",q[i]);
}
}
}
int main()
{
int option,value;
do
{
printf("\n**** MENU ****");
printf("\n1.Enqueue()");
printf("\n2.Dequeue()");
printf("\n3.isEmpty()");
printf("\n4.isFull()");
printf("\n5.Count()");
printf("\n6.Display()");
printf("\nselect an option:  if u want to exit press 0\n");
scanf("%d",&option);

switch(option)
{
case 0:
                    break;
case 1:
                    printf("Enter an item into the queue:");
                    scanf("%d",&value);
                    enqueue(value);
                    break;
case 2:     dequeue();
                            break;
case 3:
                    if(isEmpty())
                    {
                    printf("Queue is empty");
```

```
                }
                else
                {
                printf("Queue is not empty");
                }
                break;
case 4:
                if(isFull())
                {
                printf("Queue is Full");
                }
                else
                {
                printf("Queue is not full");
                }
                break;
case 5:
                printf("Number of items in the queue : %d",count());
                break;
case 6:
                display();
                break;
default:
                printf("Please select valid option");
                break;
}
}while(option!=0);
return 0;
}
```

**Applications of Queues:**

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

**Complexity Analysis of Queue Operations:**

Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

Enqueue: O(1)

Dequeue: O(1)
Size: O(1)

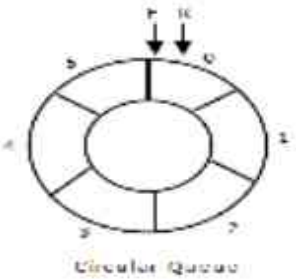
## Circular Queue:

Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

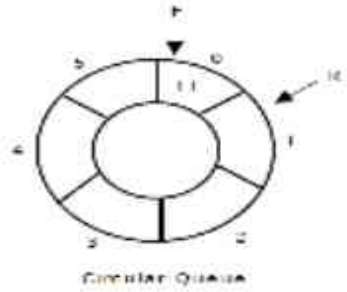
- Circular linked list fallow the First In First Out principle.
- Elements are added at the rear end and the elements are deleted at front end of the queue.

**Representation of Circular Queue:**

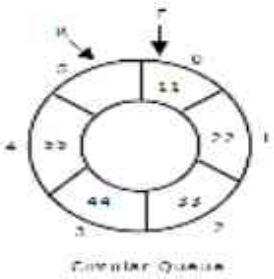
- Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



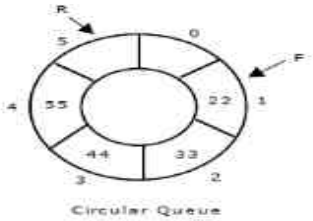
Now, insert 11 to the circular queue. Then circular queue status will be:



Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



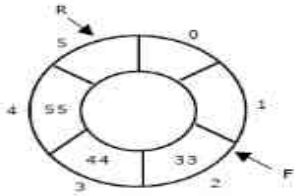
Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

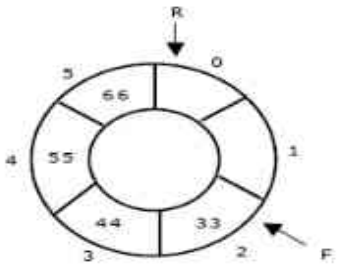
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

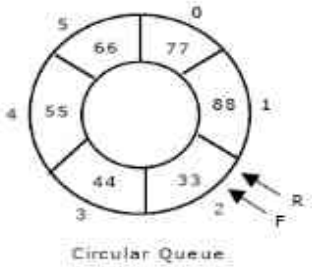
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

```
FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6
```

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

**Implementation of Circular Queue:**

```
#include<stdio.h>
int front=-1;
```

```
int rear=-1;
int q[5];
int itemcount=0;

int isEmpty()
{
if(front==-1 && rear==-1)
{
return 1;
}
else
{
return 0;
}
}
int isFull()
{
if((rear+1)%5==front)
{
return 1;
}
else
{
return 0;
}
}
void enqueue(int value)
{
if(isFull())
{
printf("Queue is full");
}
else if(isEmpty())
{
rear=0;
front=0;
q[rear]=value;
}
else
{
rear=((rear+1)%5);
q[rear]=value;
}

itemcount++;
}
void dequeue()
{
```

```
int x=0;
if(isEmpty())
{
printf("Queue is empty");
}
else if(front==rear)
{
x=q[front];
q[front]=0;
front=-1;
rear=-1;
itemcount--;
printf("dequeued value: %d",x);
}
else
{
x=q[front];
q[front]=0;
front=((front+1)%5);
itemcount--;
printf("dequeued value: %d",x);
}
}
int count()
{
if(rear==-1&&front==-1)
{
return 0;
}
else
{
return (itemcount);
}
}
void display()
{
printf("All valus in the queue are: ");
for(int i=0;i<5;i++)
{
printf("%d ",q[i]);
}
}

int main()
{
int option,value;
do
{
```

```
printf("\n**** MENU ****");
printf("\n1.Enqueue()");
printf("\n2.Dequeue()");
printf("\n3.isEmpty()");
printf("\n4.isFull()");
printf("\n5.Count()");
printf("\n6.Display()");
printf("\nselect an option:  if u want to exit press 0\n");
scanf("%d",&option);

switch(option)
{
case 0:
                 break;
case 1:
                printf("Enter an item into the queue:");
                scanf("%d",&value);
                enqueue(value);
                break;
case 2:  dequeue();
                break;
case 3:
                if(isEmpty())
                {
                printf("Queue is empty");
                }
                else
                {
                printf("Queue is not empty");
                }
                break;
case 4:
                if(isFull())
                {
                printf("Queue is Full");
                }
                else
                {
                printf("Queue is not full");
                }
                break;
case 5:
                printf("Number of items in the queue : %d",count());
                break;
case 6:
                display();
                break;
default:
```

```
                    printf("Please select valid option");
                    break;
        }
        }while(option!=0);
        return 0;
        }
```

**Complexity Analysis of Circular Queue Operations:**

Enqueue: O(1)
Dequeue: O(1)

**Unit-3:**

*Linked Lists:* Singly Linked Lists and Chains, Linked Stacks and Queues, Polynomials, Operations for Circularly linked lists, Equivalence Classes, Sparse matrices, Doubly Linked Lists.

*Hashing:* Static Hashing, Hash Tables, Hash Functions, Overflow Handling, Theoretical Evaluation of Overflow Techniques

**Objective:**

To introduce various non linear data Structures such as trees, graphs and their applications**.**

**Outcome:** Implement linked list data structures to solve various computing problems.

# Linked Lists

Linked lists and arrays are similar since they both store collections of data. Array is the most common data structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast.

**The disadvantages of arrays are:**

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Deleting an element from an array is not possible. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak.
- Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

**Linked List Concepts:**

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

**Advantages of linked lists:**

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not reallocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

**Disadvantages of linked lists:**

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

**Types of Linked Lists:**

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.

A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.

A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked by simply storing address of the very first node in the link field of the last node.

Comparison between array an linked list:

| ARRAY | LINKED LIST |
|---|---|
| Size of an array is fixed | Size of a list is not fixed |
| Memory is allocated from stack | Memory is allocated from heap |
| It is necessary to specify the number of elements during declaration (i.e., during compile time). | It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time). |
| It occupies less memory than a linked list for the same number of elements. | It occupies more memory. |
| Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room. | Inserting a new element at any position can be carried out easily. |
| Deleting an element from an array is not possible. | Deleting an element is possible. |

**Trade offs between linked lists and arrays:**

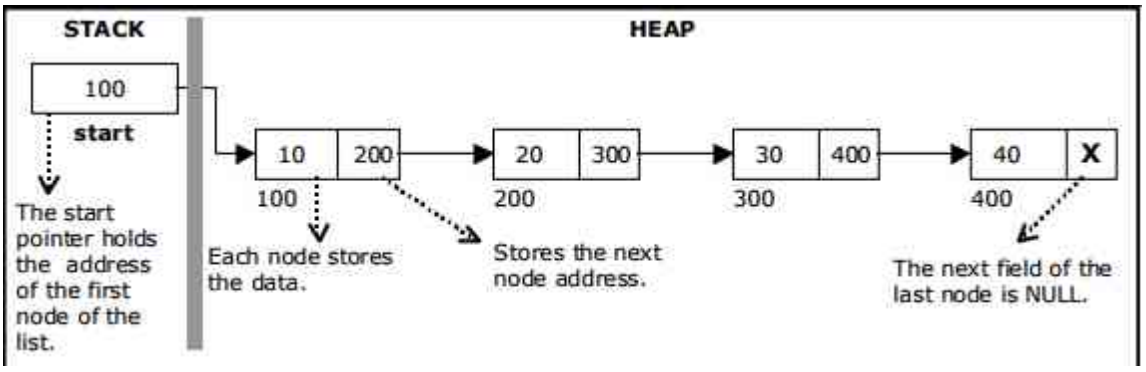
| FEATURE | ARRAYS | LINKED LISTS |
|---|---|---|
| Sequential access | efficient | efficient |
| Random access | efficient | inefficient |
| Resigning | inefficient | efficient |
| Element rearranging | inefficient | efficient |
| Overhead per elements | none | 1 or 2 links |

**Applications of linked list:**

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non zero coefficient and exponents. For example: P(x) = a0 Xn + a1 Xn-1 + …… + an-1 X + an
2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs. 4. Implement the symbol table in compiler construction.

## Single linked list:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node.




The beginning of the linked list is stored in a "start" pointer which points to the first node. The first node contains a pointer to the second node. The second node contains a pointer to the third node, ... and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the start and following the next pointers.

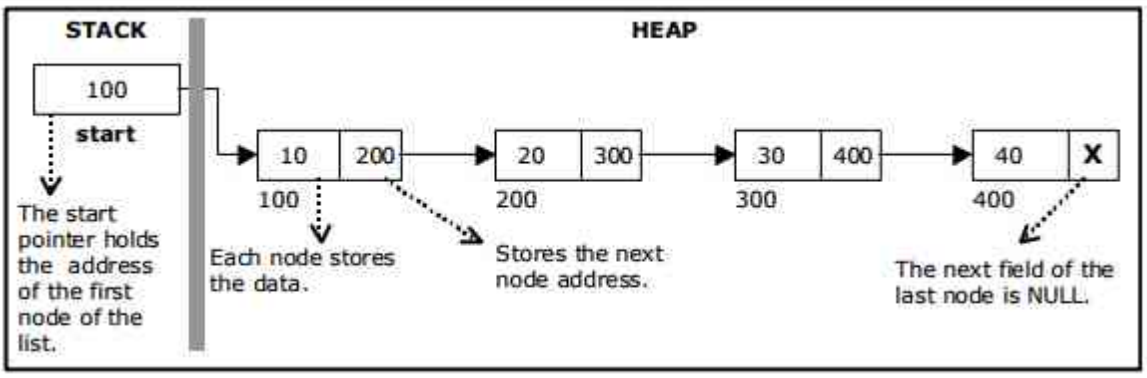
The start pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

**The basic operations in a single linked list are:**

- Creation.
- Insertion.
- Deletion.
- Traversing.

**Creating a node for Single Linked List:**

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory.

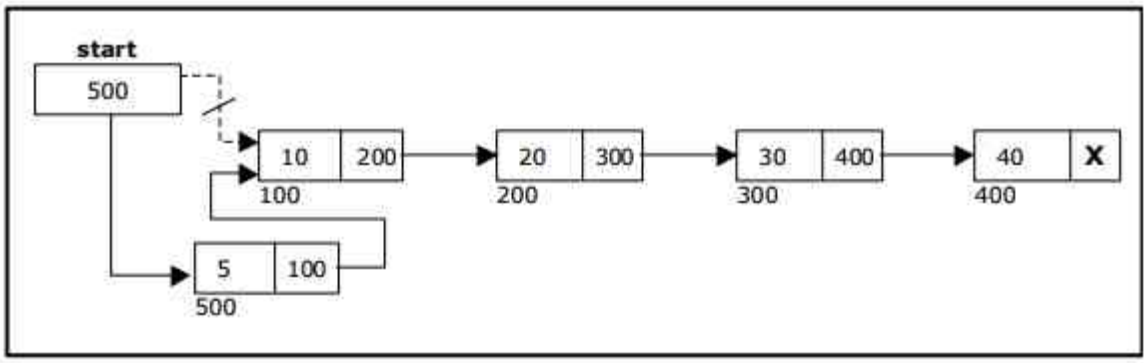



**Insertion of a Node:**

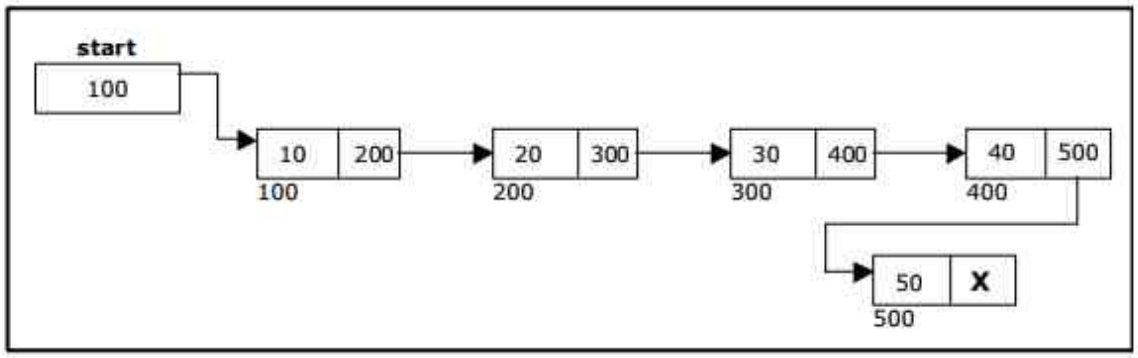
One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning.
- Inserting a node at the end.
- Inserting a node at intermediate position.

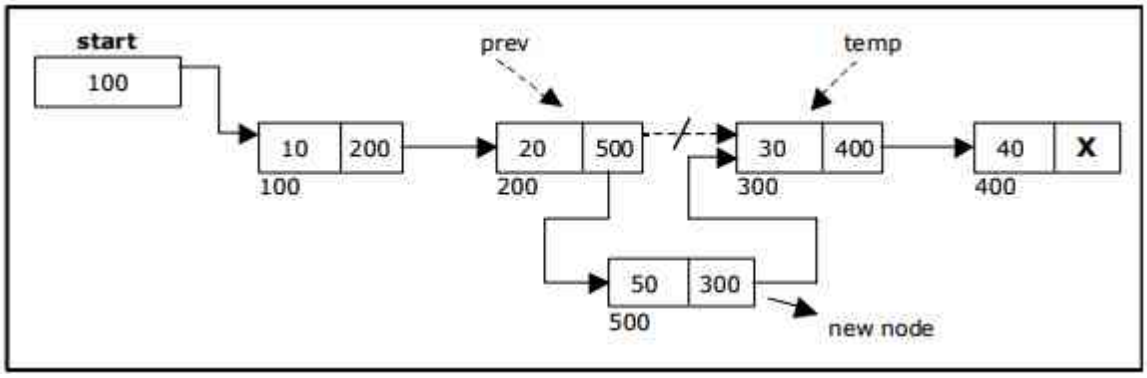
- **Inserting a node at the beginning.**




- **Inserting a node at the end:**




- **Inserting a node into the single linked list at a specified intermediate position other than beginning and end.**

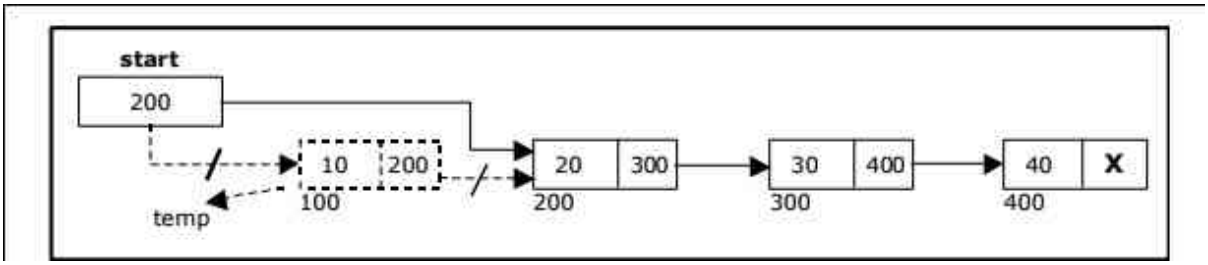



## Deletion of a node:

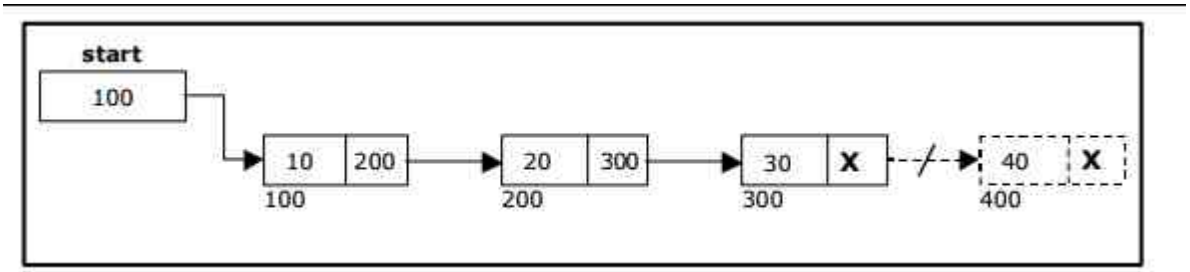
Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning.
- Deleting a node at the end.
- Deleting a node at intermediate position.

- **Deleting a node at the beginning:**

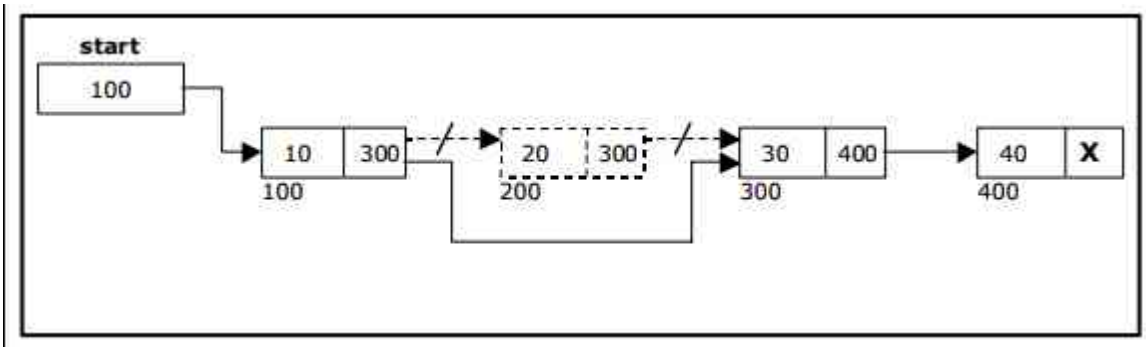



- **Deleting a node at the end:**




- **Deleting a node at Intermediate position**:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).




**Traversal and displaying a list (Left to Right):**

To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached.

Traversing a list involves the following steps:

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node. The function traverse () is used for traversing and displaying the information stored in the list from left to right.

**Algorithm for Traversing:**

```
1. algorithm traversing(head)
2. {
3. create  Node *temp=head;
4. read temp=head;
5.  if(temp==NULL) then
6. write "\nList is Empty";
7.  end if;
8. while(temp!=NULL) do
9. write temp->data;
10.  temp=temp->next;
11. end of while;
12. }
```

**Algorithm for Searching:**

```
1. algorithm searching(value)
2. {
3. read pos=0,flag=0;
4. if(head==NULL) then
5. write List is Empty;
6.  return;
7. endif;
8. create a Node *temp;
9. temp=head;
10. while(temp!=NULL) do
11. pos++;
12. if(temp->data==value) then
13. flag=1;
14. write element is found;
15. return;
16. endif ;
17.  temp=temp->next;
```

18. end of while;
19. if(!flag) then
20. write element is not found;
21. endif ;
22. }

**Algorithm for insertion into:**

```
1.  algorithm insertion(pos,ch,n)
2.  {
3.  Node *prev,*cur;
4.  head=Null;
5.  read prev=NULL,cur=head,count=1;
6.   create a Node *temp=new Node;
7.     temp->data=n;
8.  temp->next=NULL;
9.  write “INSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN
FIRST&LAST NODES";
10.  write “Enter Your Choice:";
11.  read ch;
12.  switch(ch)
{
case 1:
   temp->next=head;
   head=temp;
   break;
case 2:
   last->next=temp;
   last=temp;
   break;
case 3:
   write "\nEnter the Position to Insert:";
   read pos;
   while(count!=pos)
   {
      prev=cur;
      cur=cur->next;
      count++;
   }
13. if(count==pos) then
14. prev->next=temp;
15. temp->next=cur;
16. else
17. write “Not Able to Insert";
18. break;
19. }
```

**Algorithm for deletion from:**

```
1.  algorithm deletion(pos,ch)
2.  Node *prev=NULL,*cur=head,count=1;
3.     write “DELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
    NODES";
```

```
4.   write "\nEnter Your Choice:";
5.   read ch;
6.   switch(ch)
{
case 1:
   if(head!=NULL) then
      write "\nDeleted Element is " head->data;
      head=head->next;
   else
      write "\nNot Able to Delete";
       end if;
break;

case 2:
       if(head==NULL)
       cout<<"\nNot Able to Delete";
       else
   while(cur!=last)
      prev=cur;
      cur=cur->next;
       end while;
       end if;
   if(cur==last)
      write "\nDeleted Element is: "cur->data;
      prev->next=NULL;
      last=prev;
       endif;
   break;
case 3:
       wirte “nEnter the Position of Deletion:";
       read pos;
       if(head==NULL)
       write”/nNot Able to Delete";
       else
       while(count!=pos)
        prev=cur;
       cur=cur->next;
       count++;
       end while;
       end if;
       if(count==pos)
     write "\nDeleted Element is: " cur->data;
       prev->next=cur->next;
       end if;
   break;
7.  }
```

**Implementation of single linked list:**

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

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

   struct node *head=NULL,*last=NULL;
   void create();
   void insert();
   void delet();
   void display();
   void search();

void create()
{
   struct node *temp;
   temp=(struct node*)malloc(sizeof(struct node));
   int n;
   printf("\nEnter an Element:");
   scanf("%d",&n);
   temp->data=n;
   temp->next=NULL;
   if(head==NULL)
   {
      head=temp;
      last=head;
   }

   else
   {
      last->next=temp;
      last=temp;
   }
}
void insert()
{
   struct node *prev,*cur,*temp;
   prev=NULL;
   cur=head;
   int count=1,pos,ch,n;
   temp=(struct node*)malloc(sizeof(struct node));
  printf("\nEnter an Element:");
   scanf("%d",&n);
   temp->data=n;
   temp->next=NULL;
   printf("\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES");
   printf("\nEnter Your Choice:");
   scanf("%d",&ch);
   switch(ch)
   {
   case 1:
      temp->next=head;
      head=temp;
```

```
        break;
    case 2:
        last->next=temp;
        last=temp;
        break;
    case 3:
        printf("\nEnter the Position to Insert:");
        scanf("%d",&pos);
        printf("pos:%d,count=%d",pos,count);
        while(count!=pos)
        {
            prev=cur;
            cur=cur->next;
            count++;
        }
        if(count==pos)
        {
            prev->next=temp;
            temp->next=cur;
        }
        else
        {
            printf("\nNot Able to Insert");
        }
        break;

    }
}
void delet()
{
    struct node *prev=NULL,*cur=head;
    int count=1,pos,ch;
    printf("\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES");
    printf("\nEnter Your Choice:");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1:
        if(head!=NULL)
        {
            printf("Deleted Element is %d",head->data);
            head=head->next;
        }
        else
            printf("Not Able to Delete");
        break;
    case 2:
             if(head==NULL)
             {
             printf("Not Able to Delete");
             }
             else
             {
```

```
        while(cur!=last)
        {
           prev=cur;
           cur=cur->next;
        }
        if(cur==last)
        {
           printf("\nDeleted Element is:%d ",cur->data);
           prev->next=NULL;
           last=prev;
        }
        }
        break;
    case 3:
        printf("\nEnter the Position of Deletion:");
       scanf("%d",&pos);
            if(head==NULL)
            {
            printf("\nNot Able to Delete");
            }
            else
            {
            while(count!=pos)
            {
             prev=cur;
            cur=cur->next;
            count++;
                        }
            if(count==pos)
            {
                        printf("\nDeleted Element is:%d ",cur->data);
            prev->next=cur->next;
            }
        }
        break;
    }
}
void display()
{
    struct node *temp=head;
    if(temp==NULL)
    {
        printf("\nList is Empty");
    }
    while(temp!=NULL)
    {
        printf("%d",temp->data);
        temp=temp->next;
            if(temp!=NULL)
            {
            printf("-->");
            }
    }
}
void search()
```

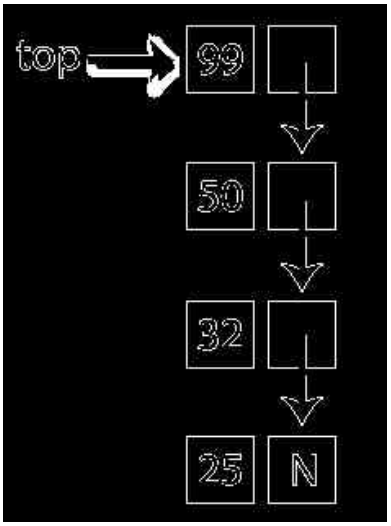
```
{
   int value,pos=0;
   int flag=0;
   if(head==NULL)
   {
      printf("List is Empty");
      return;
   }
   printf("Enter the Value to be Searched:");
   scanf("%d",&value);
   struct node *temp;
   temp=head;
   while(temp!=NULL)
   {
      pos++;
      if(temp->data==value)
      {
         flag=1;
         printf("Element %d is Found at %d Position",value,pos);
         return;
      }
      temp=temp->next;
   }
   if(!flag)
   {
      printf("Element %d not Found in the List",value);
   }
}
int main()
{
   int ch;
   while(1)
   {
      printf("\n**** MENU ****");
      printf("\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n");
      printf("\nEnter Your Choice:");
      scanf("%d",&ch);
      switch(ch)
      {
      case 1:
         create();
         break;
      case 2:
         insert();
         break;
      case 3:
         delet();
         break;
      case 4:
         search();
         break;
      case 5:
         display();
         break;
      case 6:
```

```
            return 0;
        default:
            printf("\n Invalid choice: Choose correct one");
            break;
        }
    }
    return 0;
}
```

**Linked representation of Stack:**

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.




In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

**Implementation of stack using linked list:**

```
//implemnetation of stack using linked list
#include<stdio.h>
#include<stdlib.h>

//Structure of the Node
struct node
{
```

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

struct node *head=NULL,*last=NULL;
    void create();
    void insert();
    void delet();
    void display();
    void search();
// top pointer to keep track of the top of the stack
struct node *top = NULL;

//Function to check if stack is empty or not
int isEmpty()
        {
                if(top==NULL)
                {
                        return 1;
                }
                else
                {
                        return 0;
                }
        }

//Function to insert an element in stack
void push(int value)
{
  struct node *temp = (struct node*)malloc(sizeof(struct node));
  temp->data = value;
  temp->next = top;
  top = temp;
}

//Function to delete an element from the stack
void pop()
{
 if(isEmpty())
 {
  printf("Stack is Empty");
 }
 else
 {
  struct node *temp = top;
  top = top->next;
temp->next=NULL;
 }
}
// Function to show the element at the top of the stack
void showTop()
{
 if(isEmpty())
 {
  printf("Stack is Empty");
```

```
 }
 else
 {
  printf("Element at top is : %d", top->data);
 }
}

// Function to Display the stack
void displayStack()
{
 if(isEmpty())
 {
  printf("Stack is Empty");
 }
 else
 {
  struct node *temp=top;
  while(temp!=NULL)
  {
   printf("%d",temp->data);
   temp=temp->next;
            if(temp!=NULL)
   {
       printf("->");
   }
  }
  printf("\n");
 }
 }

// Main function
int main()
{
 int choice,flag=1,value;
 //Menu Driven Program using Switch
 while(flag)
 {
 printf("\n1.Push \n2.Pop \n3.showTop \n4.displayStack \n5.exit\n");
 scanf("%d",&choice);
 switch(choice)
 {
 case 1: printf("Enter Value:\n");
        scanf("%d",&value);
        push(value);
        break;
 case 2: pop();
        break;
 case 3: showTop();
        break;
 case 4: displayStack();
        break;
 case 5: flag=0;
        break;
 }
 }
```

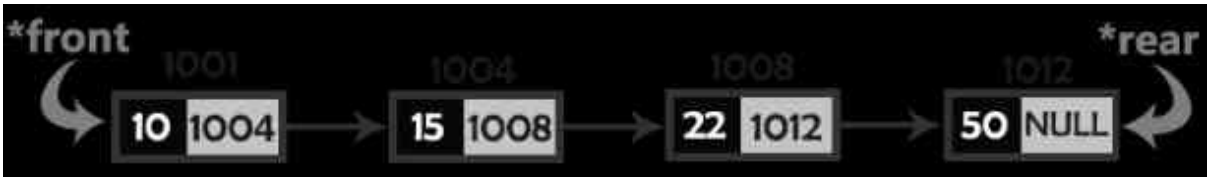
```
        return 0;
}
```

**Linked representation of Queue:**

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example:




In above example, the last inserted node is 50 and it is pointed by 'rear' and the first inserted node is 10 and it is pointed by 'front'. The order of elements inserted is 10, 15, 22 and 50.

**Implementation of queue using linked list:**

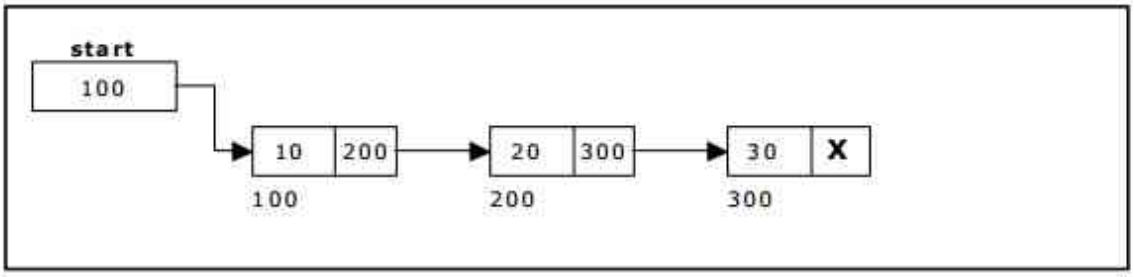
```
//implementation of queue using linked list
#include<stdio.h>
#include<stdlib.h>

// Structure of Node.
struct node
{
   int data;
   struct node *next;
};

struct node *front=NULL,*rear=NULL;
//Function to check if queue is empty or not

int isEmpty()
{
 if(front == NULL && rear == NULL)
 {
 return 1;
 }
 else
 {
```

```
 return 0;
 }
}

//function to enter elements in queue
void enqueue(int value)
{
 struct node *temp = (struct node*)malloc(sizeof(struct node));
 temp->data= value;
 temp->next = NULL;

 //If inserting the first element/node
 if(front == NULL)
 {
  front = temp;
  rear = temp;
 }
 else
 {
  rear ->next = temp;
  rear = temp;
 }
}

//function to delete/remove element from queue
void dequeue()
{
 if(isEmpty())
 {
 printf("Queue is empty\n");
 }
 else
 //only one element/node in queue.
 if( front == rear)
 {
  printf("deleted element is: %d",front->data);
  front = rear = NULL;
 }
 else
 {
  struct node *temp = front;
  printf("deleted element is: %d",front->data);
  front = front->next;
 }
}

//function to show the element at front
void showfront( )
{
 if( isEmpty())
printf("Queue is empty\n");
 else
 printf("element at front is: %d",front->data);
}
```

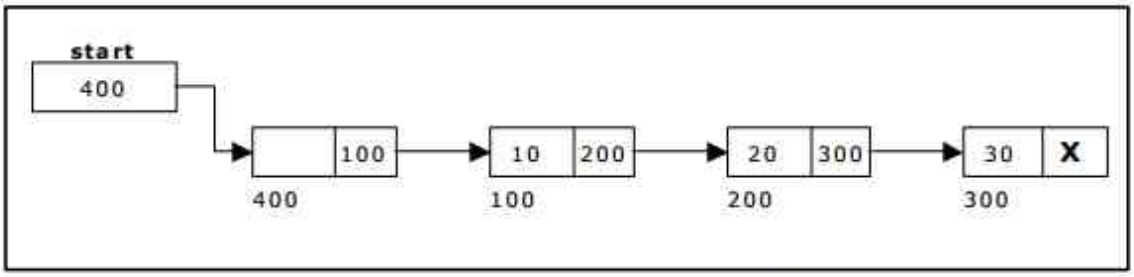
```
//function to display queue
void displayQueue()
{
 if(isEmpty())
 {
  printf("Queue is empty\n");
 }
 else
 {
  struct node *temp = front;
  while(temp!=NULL)
  {
   printf("%d",temp->data);
   temp= temp->next;
            if(temp!=NULL)
   {
      printf("->");
   }
  }
  printf("\n");
 }
}

//Main Function
int main()
{
 int choice, flag=1, value;
 while( flag == 1)
 {
  printf("\n1.enqueue 2.dequeue 3.showfront 4.displayQueue 5.exit\n");
  scanf("%d",&choice);
  switch (choice)
  {
  case 1: printf("Enter Value:\n");
         scanf("%d",&value);
         enqueue(value);
         break;
  case 2: dequeue();
         break;
  case 3: showfront();
         break;
  case 4: displayQueue();
         break;
  case 5: flag = 0;
         break;
  }
 }

 return 0;
}
```

**Using a header node:**

A header node is a special dummy node found at the front of the list. The use of header node is an alternative to remove the first node in a list. For example, the picture below shows how the list with data 10, 20 and 30 would be represented using a linked list without and with a header node:




Single Linked List without a header node




Single Linked List with header node

Note that if your linked lists do include a header node, there is no need for the special case code given above for the remove operation; node n can never be the first node in the list, so there is no need to check for that case. Similarly, having a header node can simplify the code that adds a node before a given node n.

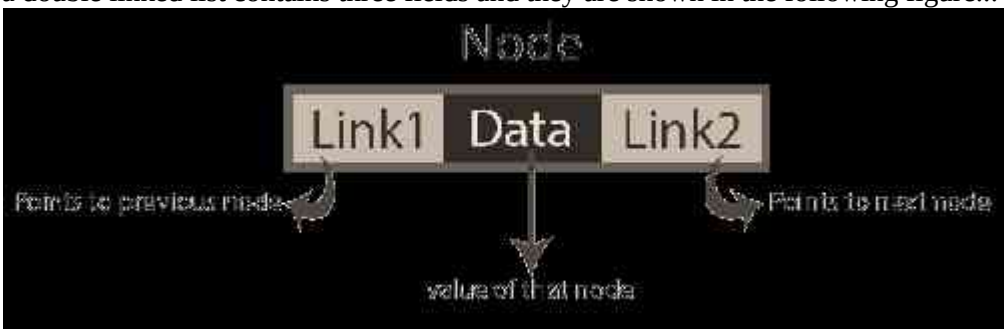
Note that if you do decide to use a header node, you must remember to initialize an empty list to contain one (dummy) node, you must remember not to include the header node in the count of "real" nodes in the list.

## Double Linked List:

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

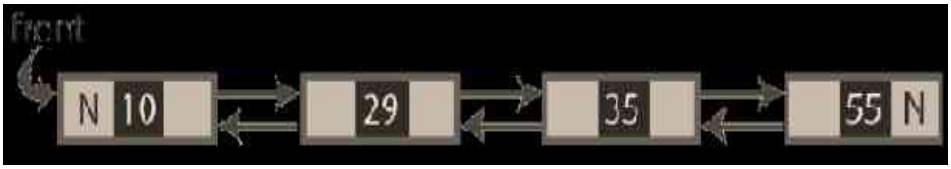
"Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence"

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...




Here, 'link1' field is used to store the address of the previous node in the sequence, 'link2' field is used to store the address of the next node in the sequence and 'data' field is used to store the actual value of that node.

**Example:**




- In double linked list, the first node must be always pointed by head.
- Always the previous field of the first node must be NULL.
- Always the next field of the last node must be NULL.
-

**Operations on Double Linked List:**

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

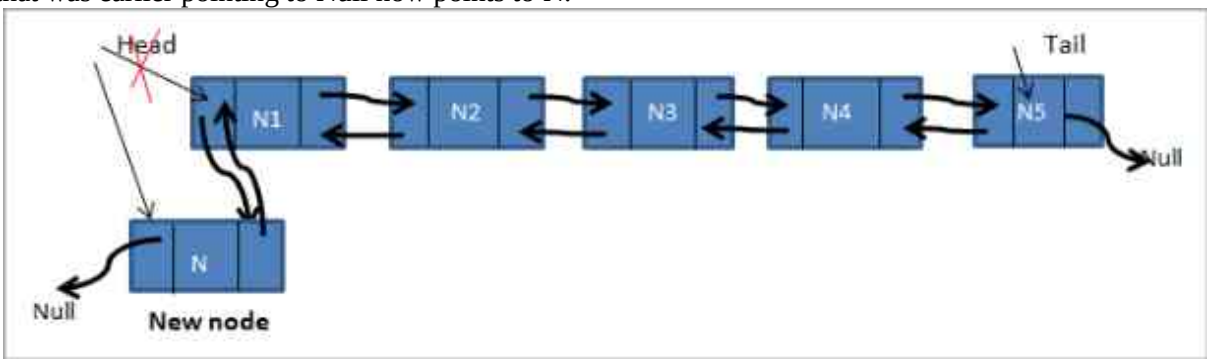
**1. Insertion:**

In a double linked list, the insertion operation can be performed in three ways as follows...

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

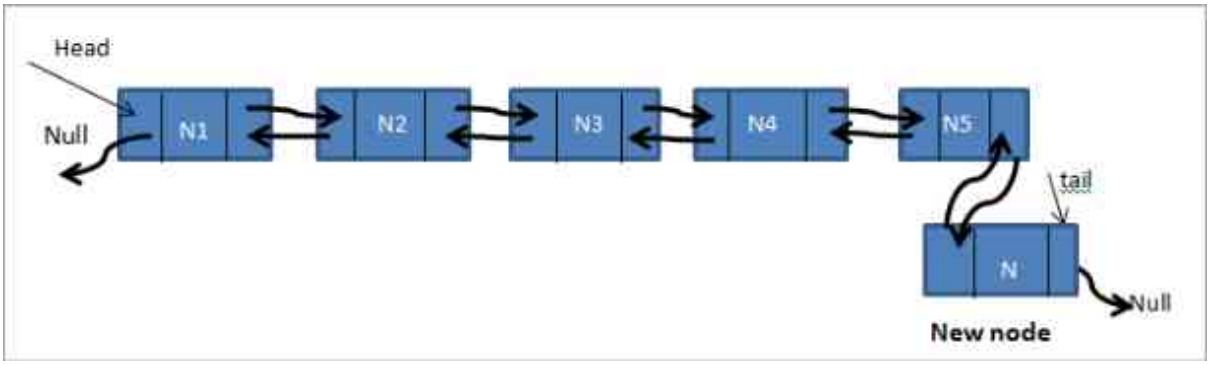
- **Inserting At Beginning of the list:**

Insertion of a new node at the front of the list is shown above. As seen, the previous new node N is set to null. Head points to the new node. The next pointer of N now points to N1 and previous of N1 that was earlier pointing to Null now points to N.



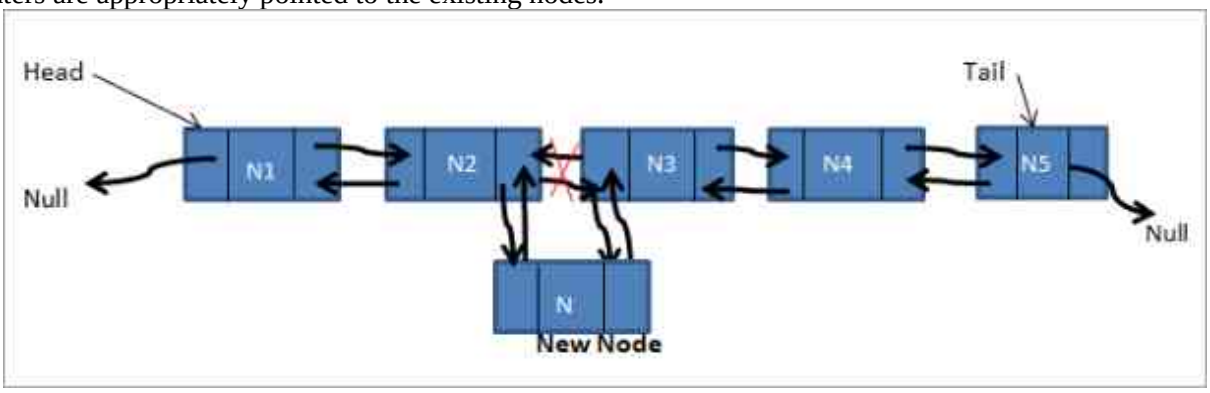

- **Inserting At End of the list:**

Inserting node at the end of the doubly linked list is achieved by pointing the next pointer of new node N to null. The previous pointer of N is pointed to N5. The 'Next' pointer of N5 is pointed to N.




- **Inserting At Specific location in the list:**

When we have to add a node before or after a particular node, we change the previous and next pointers of the before and after nodes so as to appropriately point to the new node. Also, the new node pointers are appropriately pointed to the existing nodes.




**2. Deletion:**

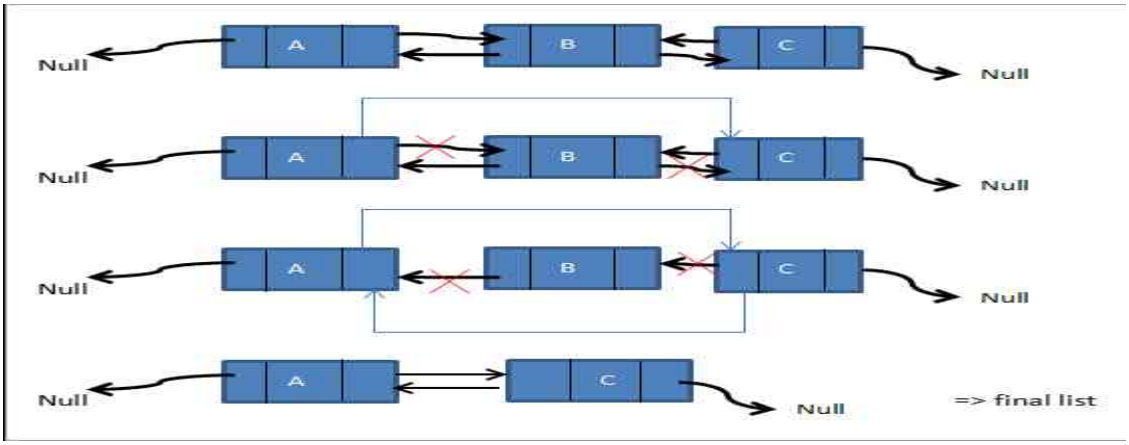
A node can be deleted from a doubly linked list from any position like from the front, end or any other given position or given data. When deleting a node from the doubly linked list, we first reposition the pointer pointing to that particular node so that the previous and after nodes do not have any connection to the node to be deleted. We can then easily delete the node.

In a double linked list, the deletion operation can be performed in three ways as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

The deletion of node B from the given linked list. The sequence of operation remains the same even if the node is first or last. The only care that should be taken is that if in case the first node is deleted, the second node's previous pointer will be set to null.

Similarly, when the last node is deleted, the next pointer of the previous node will be set to null. If in between nodes are deleted, then the sequence will be as above.




### Inserting At Beginning of the list:

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

### Inserting At End of the list:

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

### Inserting At Specific location in the list (After a Node):

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display '**Given node is not found in the list!!! Insertion not possible!!!**' and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

### Deleting from Beginning of the list:

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

### Deleting from End of the list:

We can use the following steps to delete a node from end of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp → previous → next** and delete **temp**.

### Deleting a Specific Node from the list:

We can use the following steps to delete a specific node from the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer '**temp**' and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5 -** If it is reached to the last node, then display '**Given node not found in the list! Deletion not possible!!!**' and terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp**)).

- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

### Displaying a Double Linked List:

We can use the following steps to display the elements of a double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display '**List is Empty!!!**' and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer '**temp**' and initialize with **head**.
- **Step 4 -** Display '**NULL** <--- '.
- **Step 5 -** Keep displaying **temp → data** with an arrow (<===>) until **temp** reaches to the last node
- **Step 6 -** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

### Implementation:

```
#include<stdio.h>

#include<stdlib.h>
struct node
{
   int data;
  struct node *prev,*next;
};

   struct node *head=NULL,*last=NULL;
   void create();
   void insert();
   void delet();
   void display();
   void search();

void create()
{
   struct node *temp;
   temp=(struct node*)malloc(sizeof(struct node));
   int n;
   printf("\nEnter an Element:");
   scanf("%d",&n);
   temp->data=n;
   temp->next=NULL;
   temp->prev=NULL;
   if(head==NULL)
   {
      head=temp;
      last=head;
   }
    else
   {
      last->next=temp;
      temp->prev=last;
```

```
        last=temp;
    }
}
void insert()
{
    struct node *old,*cur,*temp;
    old=NULL;
    cur=head;
    int count=1,pos,ch,n;
    temp=(struct node*)malloc(sizeof(struct node));
    printf("\nEnter an Element:");
    scanf("%d",&n);
    temp->data=n;
    temp->next=NULL;
    temp->prev=NULL;
    printf("\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES");
    printf("\nEnter Your Choice:");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1:
        temp->next=head;
        head->prev=temp;
        head=temp;
        break;
    case 2:
        last->next=temp;
        temp->prev=last;
        last=temp;
        break;
    case 3:
        printf("\nEnter the Position to Insert:");
        scanf("%d",&pos);
        while(count!=pos)
        {
            old=cur;
            cur=cur->next;
            count++;
        }
        if(count==pos)
        {
          temp->next=old->next;
          cur->prev=temp;
          old->next=temp;
          temp->prev=old;
        }
        else
            printf("\nNot Able to Insert");
        break;

    }
}
void delet()
{
```

```
struct node *old=NULL,*cur=head;
int count=1,pos,ch;
printf("\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST NODES");
printf("\nEnter Your Choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
   if(head==NULL)
   {
                printf("\nNot Able to Delete");
       }
       else
       {
       printf("\nDeleted Element is %d",head->data);
      if(head==last)
      {
      head=last=NULL;
      }
      else
      {
      struct node *temp;
      temp=head;
      head=head->next;
      head->prev=NULL;
      }
   }
         break;
case 2:
       if(head==NULL)
       {
       printf("\nNot Able to Delete");
       }
       else
       {

   while(cur!=last)
   {
      old=cur;
      cur=cur->next;
   }
   if(cur==last)
   {
      printf("\nDeleted Element is: %d",cur->data);
      if(old==NULL)
      {
      head=NULL;
      }
      else
      {
      old->next=NULL;
      last=old;
      }
    }
   }
```

```
        break;
    case 3:
        printf("\nEnter the Position of Deletion:");
        scanf("%d",&pos);
            if(head==NULL)
            {
            printf("\nNot Able to Delete");
            }
            else
            {
            while(count!=pos)
            {
                        old=cur;
            cur=cur->next;
            count++;
                        }
            if(count==pos)
            {
            printf("\nDeleted Element is:%d",cur->data);
            old->next=cur->next;
            (cur->next)->prev=old;
            }
        }
        break;
    }
}
void display()
{
    struct node *temp=head;
    if(temp==NULL)
    {
        printf("\nList is Empty");
    }
    while(temp!=NULL)
    {
        printf("%p[%p,%d,%p]",temp,temp->prev,temp->data,temp->next);
        temp=temp->next;
        if(temp!=NULL)
        {
        printf("-->");
        }
    }

}
void search()
{
    int value,pos=0;
    int flag=0;
    if(head==NULL)
    {
        printf("List is Empty");
        return;
    }
    printf("Enter the Value to be Searched:");
    scanf("%d",&value);
```

```
    struct node *temp;
    temp=head;
    while(temp!=NULL)
    {
       pos++;
       if(temp->data==value)
       {
          flag=1;
          printf("Element %d is Found at %d Position",value,pos);
          return;
       }
       temp=temp->next;
    }
    if(!flag)
    {
       printf("Element %d not Found in the List",value);
    }
}
int main()
{
    int ch;
    while(1)
    {
       printf("\n**** MENU ****");
       printf("\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n");
       printf("\nEnter Your Choice:");
       scanf("%d",&ch);
       switch(ch)
       {
       case 1:
          create();
          break;
       case 2:
          insert();
          break;
       case 3:
          delet();
          break;
       case 4:
          search();
          break;
       case 5:
          display();
          break;
       case 6:
          return 0;
       }
    }
    return 0;
}
```

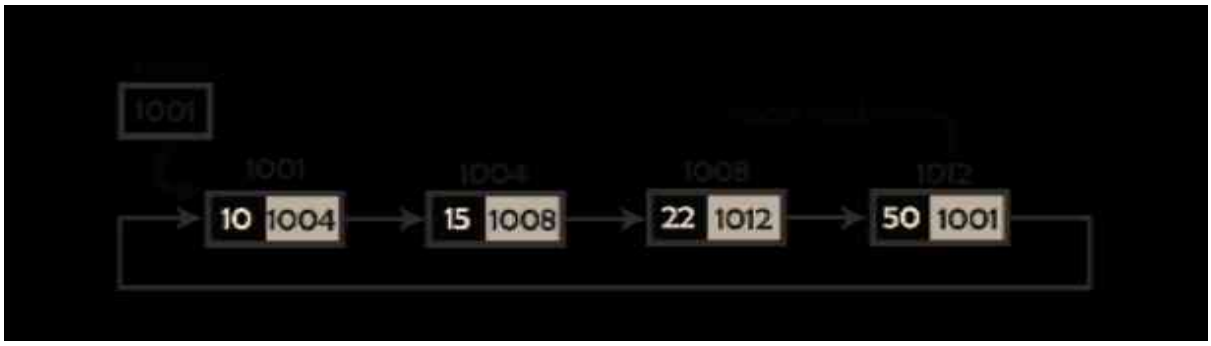
## Circular Linked List:

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to

the first node in the list. A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

Example:




**Operations:**
In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined** functions.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

## Insertion:

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

**Inserting At Beginning of the list:**
We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
- **Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

**Inserting At End of the list:**
We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6 -** Set **temp → next** = **newNode** and **newNode → next** = **head**.

**Inserting At Specific location in the list (After a Node):**
We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

## Deletion:

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

**Deleting from Beginning of the list:**
We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize both **'temp1'** and **'temp2'** with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head** )
- **Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

**Deleting from End of the list:**
We can use the following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
- **Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

**Deleting a Specific Node from the list:**
We can use the following steps to delete a specific node from the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display '**Given node not found in the list! Deletion not possible!!!**'. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

**Displaying a circular Linked List:**
We can use the following steps to display the elements of a circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display '**List is Empty!!!**' and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (--->) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **head → data**.

## Implementation:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head=NULL,*last=NULL;
   void create();
   void insert();
   void delet();
   void display();
   void search();
void create()
{
  struct node *temp;
   temp=(struct node*)malloc(sizeof(struct node));
   int n;
   printf("\nEnter an Element:");
   scanf("%d",&n);
   temp->data=n;
   temp->next=NULL;
   if(head==NULL)
   {
      head=temp;
      temp->next=head;
      last=head;
   }

   else
   {
     temp->next=last->next;
      last->next=temp;
      last=temp;
   }
}
void insert()
{
   struct node *prev,*cur,*temp;
   prev=NULL;
   cur=head;
   int count=1,pos,ch,n;
   temp=(struct node*)malloc(sizeof(struct node));
   printf("\nEnter an Element:");
   scanf("%d",&n);
   temp->data=n;
   temp->next=NULL;
   printf("\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES");
   printf("\nEnter Your Choice:");
   scanf("%d",&ch);
   switch(ch)
   {
   case 1:
```

```
            temp->next=head;
            head=temp;
            last->next=head;
            break;
         case 2:
            temp->next=last->next;
            last->next=temp;
            last=temp;
            break;
         case 3:
            printf("\nEnter the Position to Insert:");
            scanf("%d",&pos);
            while(count!=pos)
            {
               prev=cur;
               cur=cur->next;
               count++;
            }
            if(count==pos)
            {
               temp->next=prev->next;
               prev->next=temp;
            }
            else
               printf("\nNot Able to Insert");
            break;

         }
      }
      void delet()
      {
         struct node *prev=NULL,*cur=head;
         int count=1,pos,ch;
         printf("\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES");
         printf("\nEnter Your Choice:");
         scanf("%d",&ch);
         switch(ch)
         {
         case 1:
            if(head!=NULL)
            {
                     if(head==last)
                     {
                        printf("\nDeleted Element is %d",head->data);
                        head=NULL;
                     }
                          else
                          {
                            printf("\nDeleted Element is %d",head->data);
                        head=head->next;
                        last->next=head;
                     }
                    }
            else
```

```
            printf("\nNot Able to Delete");
          break;
      case 2:
        if(head==NULL)
        {
        printf("\nNot Able to Delete");
        }
        else if(last==head)
        {
        printf("\nDeleted Element is %d",head->data);
                        head=NULL;
                }
        else
        {

         while(cur!=last)
         {
            prev=cur;
            cur=cur->next;
         }
         if(cur==last)
         {
            printf("\nDeleted Element is: %d",cur->data);
            prev->next=head;
            last=prev;
         }
         }
         break;
      case 3:
         printf("\nEnter the Position of Deletion:");
         scanf("%d",&pos);
                if(head==NULL)
        {
        printf("\nNot Able to Delete");
        }
        else
        {
                while(count!=pos)
                {
                 prev=cur;
                cur=cur->next;
                count++;
                        }
                if(count==pos)
                {
                printf("\nDeleted Element is: %d",cur->data);
                prev->next=cur->next;
                }
         }
         break;
      }
}
void display()
{
   struct node *temp=head;
```

```
        if(temp==NULL)
        {
           printf("\nList is Empty");
        }
        else
        {
                    while(temp!=last)
                    {
                       printf("[data:%d,Present node address:%p,Next node address:%p]\n",temp->data,temp,temp->next);
                       temp=temp->next;
                    }
                    printf("[data:%d,Present node address:%p,Next node address:%p]",last->data,last,last->next);


          }
    }
    void search()
    {
       int value,pos=0;
       int flag=0;
       if(head==NULL)
       {
          printf("List is Empty");
          return;
       }
       printf("Enter the Value to be Searched:");
       scanf("%d",&value);
       struct node *temp;
       temp=head;
       do
       {
          pos++;
          if(temp->data==value)
          {
             flag=1;
             printf("Element %d is Found at %d Position",value,pos);
             return;
          }
          temp=temp->next;
       }while(temp!=head);
       if(!flag)
       {
          printf("Element %d not Found in the List",value);
       }
    }
    int main()
    {
       int ch;
       while(1)
       {
          printf("\n**** MENU ****");
          printf("\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n");
          printf("\nEnter Your Choice:");
```

```
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:
          create();
           break;
        case 2:
           insert();
           break;
        case 3:
           delet();
           break;
        case 4:
         search();
           break;
        case 5:
           display();
           break;
        case 6:
           return 0;
        }
    }
    return 0;
}
```

# Hashing:

In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements present in that data structure. In all these searching techniques, as the number of elements increases the time required to search an element also increases linearly.

Hashing is another approach in which time required to search an element doesn't depend on the total number of elements. Using hashing data structure, a given element is searched with constant time complexity. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

Hashing is defined as follows...

"Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key".

Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

In this data structure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a hash function.

That means every entry in the hash table is based on the hash key value generated using the hash function.

Hash Table is defined as follows...

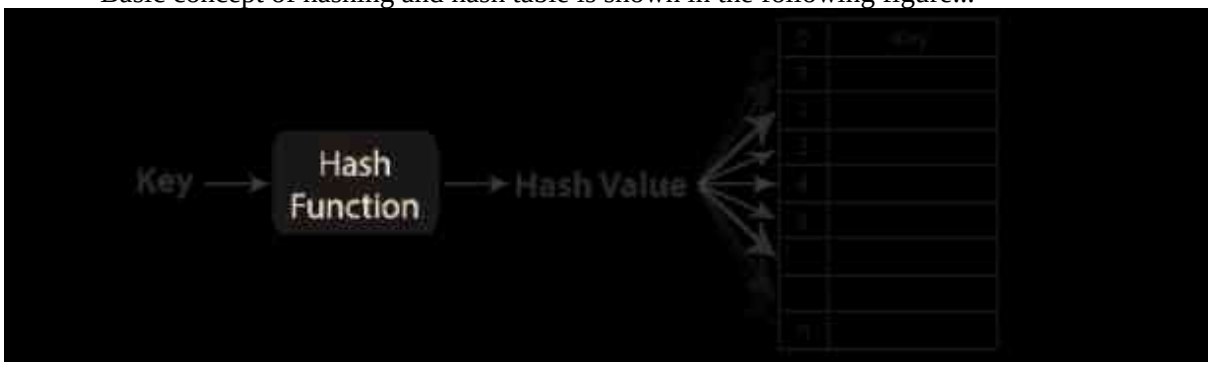
"Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1))".

Hash tables are used to perform insertion, deletion and search operations very quickly in a data structure. Using hash table concept, insertion, deletion, and search operations are accomplished in constant time complexity. Generally, every hash table makes use of a function called hash function to map the data into the hash table.

A hash function is defined as follows...

"Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table".

Basic concept of hashing and hash table is shown in the following figure...

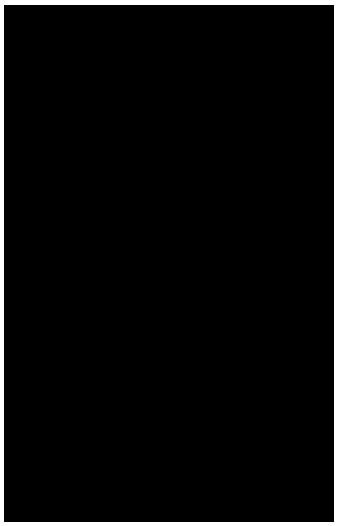



- To insert elements into the hash table we use hash function i.e., h(x)=x.
- Let us consider an example ,insert 5,7,10,12,15,100

Hash function is h(x)=x

Now, we will insert the values into the table

i.e., h(5)=5
h(7)=7
h(10)=10
h(12)=12
h(15)=15



Actually this function is a linear function, because of this we want to insert a large number then we require large amount of memory, so there may be a wastage of memory. To overcome this, we use another type of hash function called modulo division hash function.

**Modulo division hash function:**

- The hash function is

h(key)=key mod size of hash table (or)
h(key)=key % size of hash table

Example:

h(x)=x % n

where h(x) is the hash value obtained by dividing the key value x by size of hash table n using the remainder.

Let us take another example:
insert 162,196,185,93,97,692 and size of the hash table is 6.

i. h(162)=162 mod 6=162 % 6=0

here, remainder is So the key 162 is placed in $0^{th}$ index.

ii. h(196)=196 mod 6=196 % 6=4
iii. h(185)=185 mod 6=185 % 6=5
iv. h(93)=93 % 6=3
v. h(97)=97 % 6=1
vi. h(692)=692 % 6=2

| Index | Value |
|---|---|
| 0 | 162 |
| 1 | 97 |
| 2 | 692 |
| 3 | 93 |
| 4 | 196 |
| 5 | 185 |

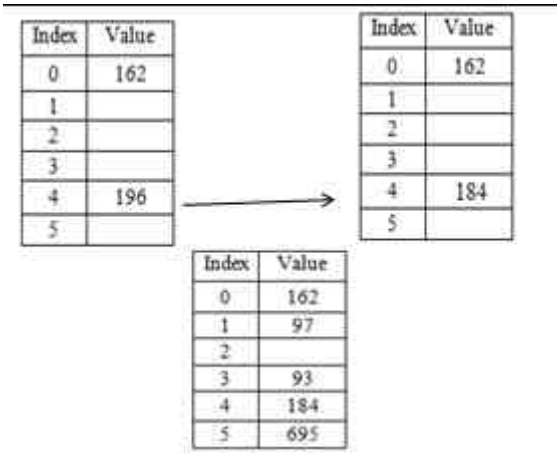
**Collision:**

If the hash function returns same hash key for more than one element, then that situation is called **Collision**.
In this collision, first element will overwrite with the second element.

Let us consider an example

insert 162,196,184,93,97,695 and size is 6

i. h(162)=162% 6=0
ii. h(196)=196%6=4
iii. h(184)=184% 6=4
iv. h(93)=93% 6=3
v. h(97)=97% 6=1
vi. h(695)=695% 6=5



| Index | Value |
|---|---|
| 0 | 162 |
| 1 | |
| 2 | |
| 3 | |
| 4 | 196 |
| 5 | |

| Index | Value |
|---|---|
| 0 | 162 |
| 1 | |
| 2 | |
| 3 | |
| 4 | 184 |
| 5 | |

| Index | Value |
|---|---|
| 0 | 162 |
| 1 | 97 |
| 2 | |
| 3 | 93 |
| 4 | 184 |
| 5 | 695 |

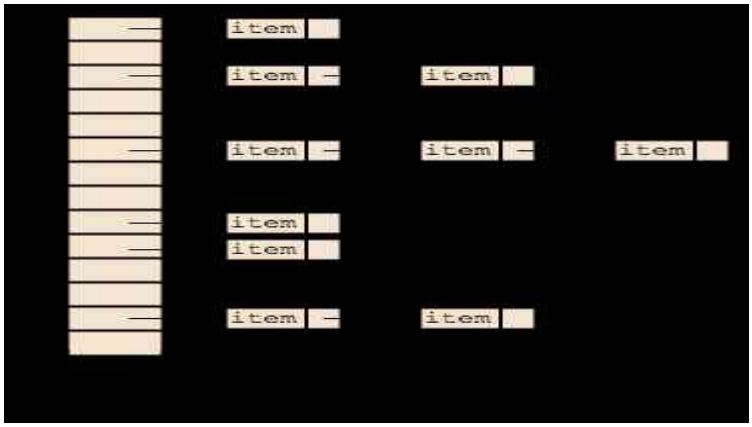
The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some **collision handling technique.**

There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing

***1. Separate Chaining:***

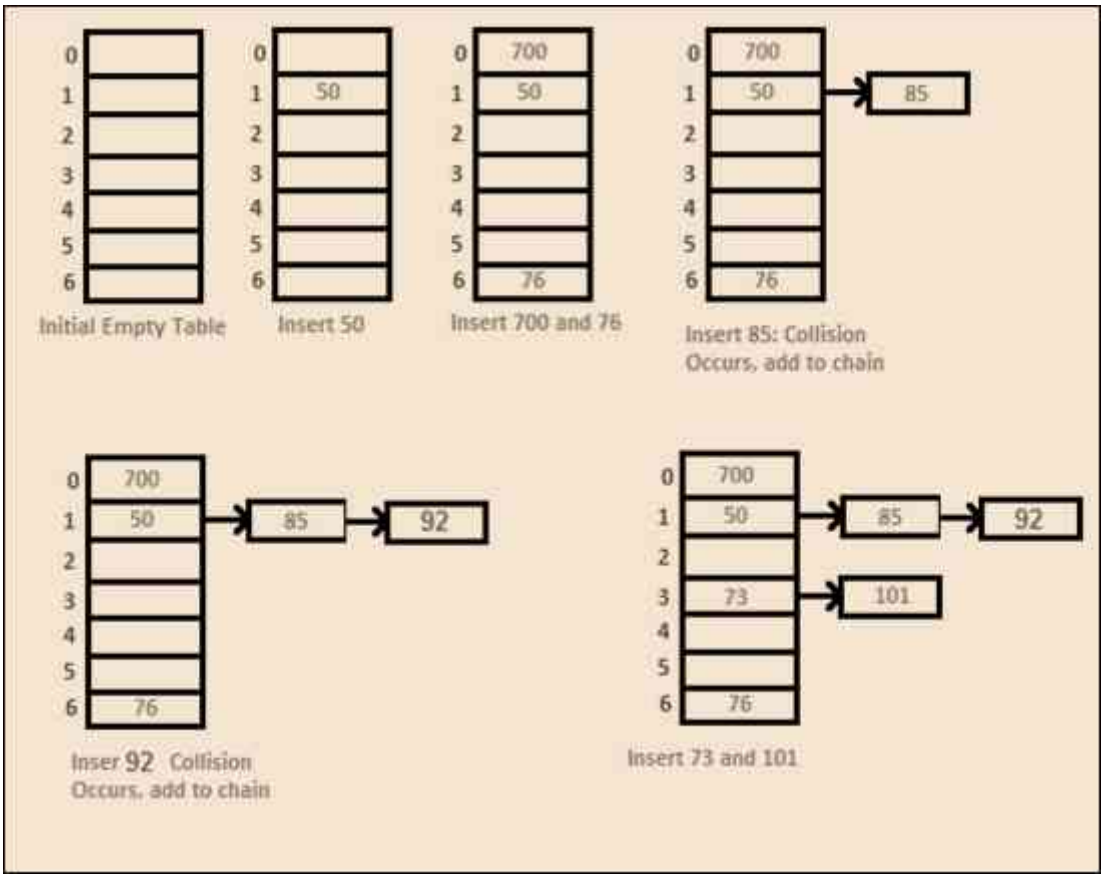
Separate Chaining is also called as closed addressing and open hashing. Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.




The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.




Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become O(n) in worst case.
- Uses extra space for links.

***2. Open Addressing:***

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. In open addressing, instead of linked lists, all entry elements are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

Open addressing is again three types:

*a. Linear probing:*

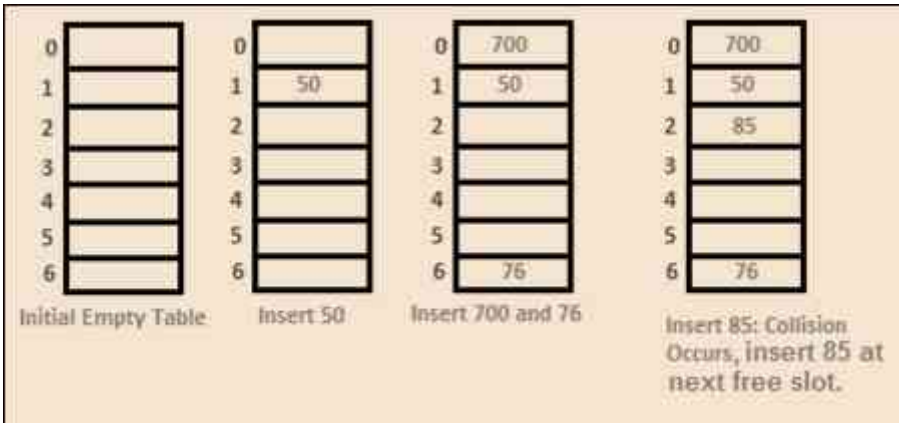
When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

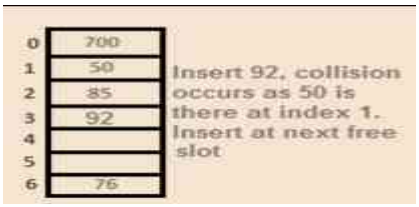
In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

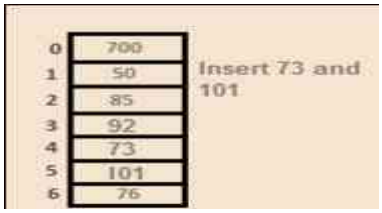
Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.




While inserting 85,there is a collision with slot 1,so we are using linear probing approach,
index = (index + 1) % hashTableSize
i.e., index=(85+1)%7
=86%7
=2 so now 85 will insert in slot 2 which is empty.

Next element 92, While inserting 92,there is a collision with slot 1so again we are using linear probing approach
index = (index + 1) % hashTableSize=(92+1)%7=1, there is a collision with slot 2
index = (index + 2) % hashTableSize=(92+2)%7=3 so now 92 will insert in slot 3 which is empty.



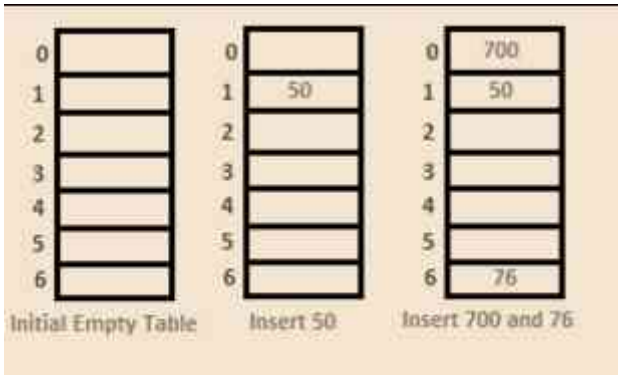

*b. Quadratic probing:*

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

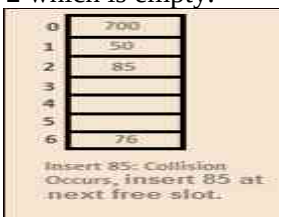
Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + $1^2$) % hashTableSize
index = (index + $2^2$) % hashTableSize
index = (index + $3^2$) % hashTableSize
and so on…

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.




While inserting 85,there is a collision with slot 1,so we are using quadratic probing approach,
index = (index + $1^2$) % hashTableSize
i.e., index=(85+$1^2$)%7=(85+1)%7
=86%7
=2 so now 85 will insert in slot 2 which is empty.




Next element 92, While inserting 92,there is a collision with slot 1so again we are using linear probing approach
index = (index + $1^2$) % hashTableSize=(92+$1^2$)%7=1, there is a collision with slot 2.
index = (index + $2^2$) % hashTableSize=(92+$2^2$)%7==(92+4)%7=5 so now 92 will insert in slot 5 which is empty.

| | |
|---|---|
| 0 | 700 |
| 1 | 50 |
| 2 | 85 |
| 3 | |
| 4 | |
| 5 | **92** |
| 6 | 76 |

Next element 73,
index = index % hashTableSize
index=73%7=3, so now 73 will insert in slot 3 which is empty.

| | |
|---|---|
| 0 | 700 |
| 1 | 50 |
| 2 | 85 |
| 3 | **73** |
| 4 | |
| 5 | 92 |

| | |
|---|---|
| 6 | 76 |

Next element 101,
index = index % hashTableSize=101%7=3, there is a collision with slot 3.
index = (index + $1^2$) % hashTableSize=(101+$1^2$)%7=4, so now 101 will insert in slot 4 which is empty.

| | |
|---|---|
| 0 | 700 |
| 1 | 50 |
| 2 | 85 |
| 3 | 73 |
| 4 | **101** |
| 5 | 92 |
| 6 | 76 |

*c. Double Hashing:*

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using :

**(hash1(index) + i * hash2(index)) % hashTableSize**

Here hash1() and hash2() are hash functions and hashTableSize is size of hash table.
(We repeat by increasing i when collision occurs)

First hash function is typically **hash1(index) = index % hashTableSize**
Second hash function is : **hash2(index) = PRIME – (index % PRIME)** where PRIME is a prime smaller than the hashTableSize.

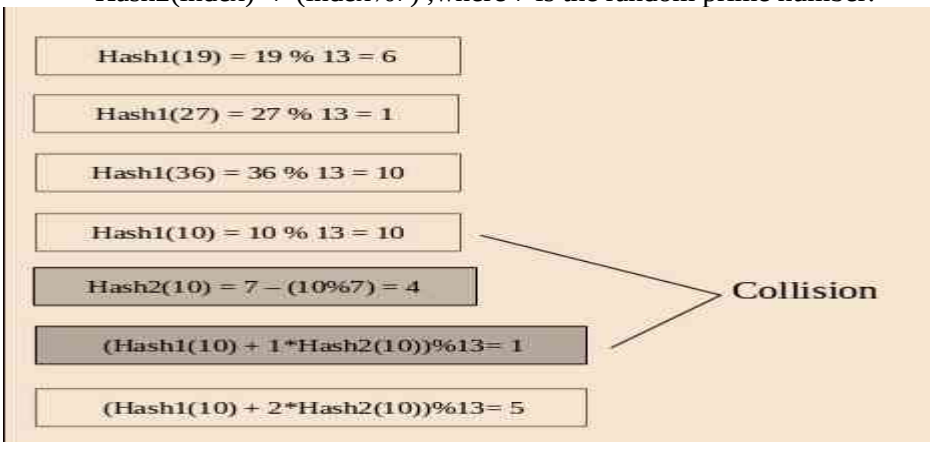
There are a couple of requirements for the second function:

- It must never evaluate to 0
- Must make sure that all cells can be probed

Let us consider a simple hash function as “key mod 13” and sequence of keys as 19,27,36,10. Assume size of the hash table is 13.

Let say Hash1(index)=index%13

Hash2(index)=7-(index%7) ,where 7 is the random prime number.




The hash table is

| | |
|---|---|
| 0 | |
| 1 | 27 |
| 2 | |
| 3 | |

| | |
|---|---|
| 4 | |
| 5 | 10 |
| 6 | 19 |
| 7 | |
| 8 | |
| 9 | |
| 10 | 36 |
| 11 | |
| 12 | |
| 13 | |

**Implementation of linear probing hash function:**

```
#include<stdio.h>

/*
This is code for linear probing in open addressing. If you want to do quadratic probing and double hashing which are also
open addressing methods in this code when I used hash function that (pos+1)%hFn in that place just replace with another function.
*/

void insert(int ary[], int size){
   int element,pos,n=0;

          printf("Enter key element to insert");
          scanf("%d",&element);

          pos = element%size;

          while(ary[pos]!= 0) {
                    if(ary[pos]== size)
         break;
                    pos = (pos+1)%size;
                    n++;
                    if(n==size)
                    break;      // If table is full we should break, if not check this, loop will go to infinite loop.
          }

          if(n==size)
      printf("Hash table was full of elements.No Place to insert this element\n");
          else
      ary[pos] = element;    //Inserting element
}

void delet(int ary[],int size){

          int element,n=0,pos;

          printf("Enter an element to delete");
          scanf("%d",&element);

          pos = element%size;
```

```
        while(n++ != size){
                if(ary[pos]==0){
                        printf("Element not found in hash table\n");
                        break;
                }
                else if(ary[pos]==element){
                        ary[pos]=0;
                        printf("Element deleted\n");
                        break;
                }
                else{
                        pos = (pos+1) % size;
                }
        }
        if(--n==size)
    printf("Element not found in hash table\n");
}

void search(int ary[],int size){
        int element,pos,n=0;

        printf("\nEnter an element you want to search");
        scanf("%d",&element);


        pos = element%size;

        while(n++ != size){
                if(ary[pos]==element){
                        printf("Element found at index %d",pos);
                        break;
                }
                else
         if(ary[pos]==size ||ary[pos]!=0)
           pos = (pos+1) %size;
         }
         if(--n==size)
           printf("Element not found in hash table\n");
}

void display(int ary[],int size){
        int i;

        printf("Index\t Value\n");

        for(i=0;i<size;i++)
    printf("%d\t %d\n",i,ary[i]);
}
int main(){
        int size,hFn,i,choice;

        printf("Enter size of hash table");
        scanf("%d",&size);
```

```
        int ary[size];


        for(i=0;i<size;i++)
    ary[i]=0;

        do{
                printf("Enter your choice\n");
                printf(" 1-> Insert\n 2-> Delete\n 3-> Display\n 4-> Searching\n 0-> Exit\n");

                scanf("%d",&choice);

                switch(choice){
                        case 1:
                                insert(ary,size);
                                break;
                        case 2:
                                delet(ary,size);
                                break;
                        case 3:
                                display(ary,size);
                                break;
                        case 4:
                                search(ary,size);
                                break;
                        default:
                                printf("Enter correct choice\n");
                                break;
                }
        }while(choice);

        return 0;
}
```

**Unit-IV:**
*Trees:* Introduction, Binary Trees, Binary Tree Traversals, Heaps, Binary Search trees (BST) : Definition, Searching an element, Insertion into a BST, Deletion from a BST.
*Efficient Binary Search Trees:* AVL Trees: Definition, Searching an element, Insertion into a AVL
**Objective:**
To discuss and implement data structure to solve real world problems.
**Outcome:** Decide a suitable tree data structure to solve a real world problem.

# Trees

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

"Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition".

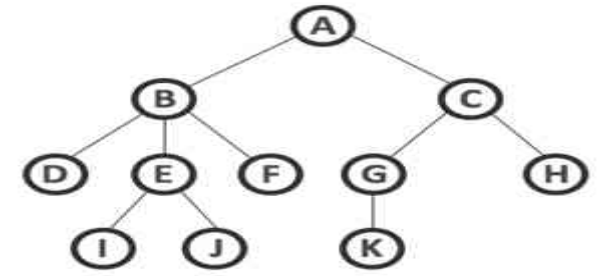
A tree data structure can also be defined as follows...

"Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively".

In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

## Example:



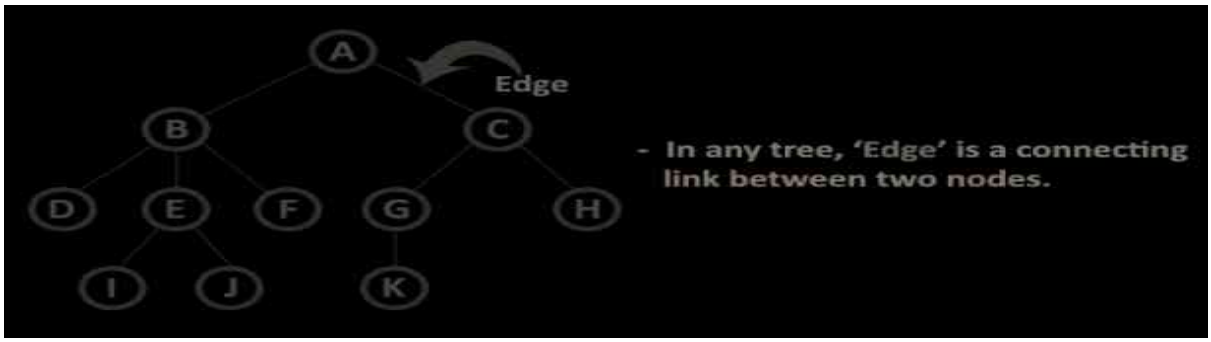

### Basic Tree Terminology:

**1. Root:**

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

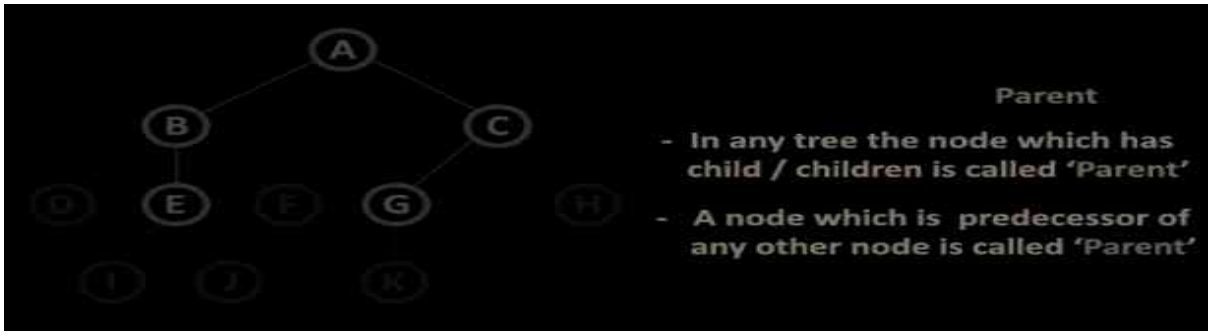

**2. Edge:**

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



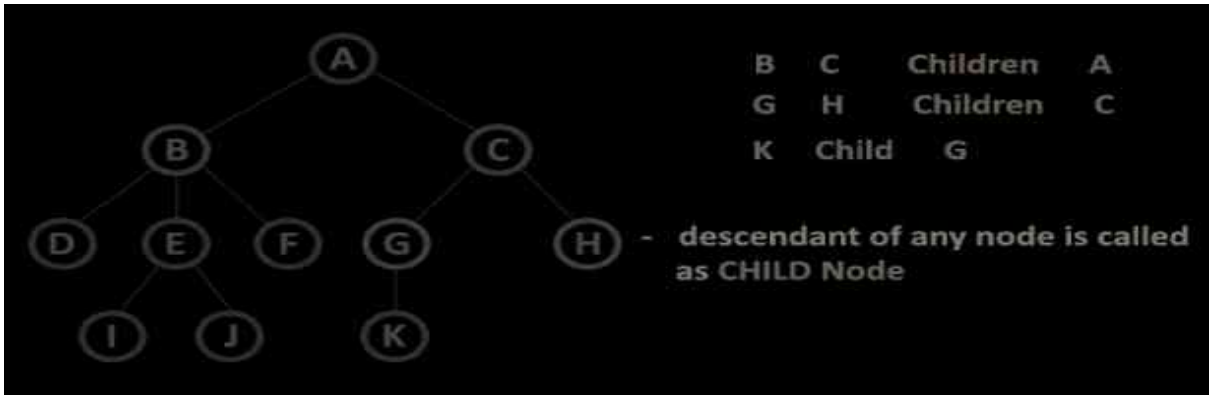

**3. Parent:**

In a tree data structure, the node which is a predecessor of any node is called as PARENT NODE. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "The node which has child / children".



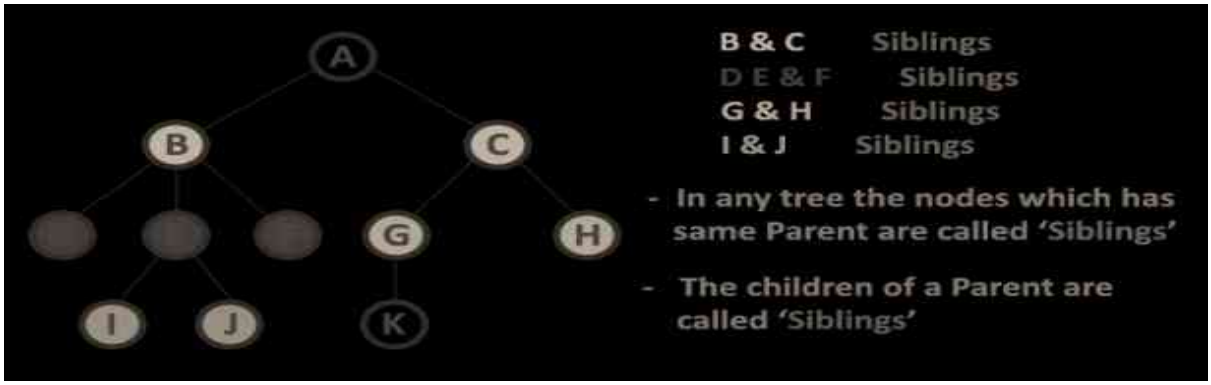

**4. Child:**

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.




**5. Siblings:**

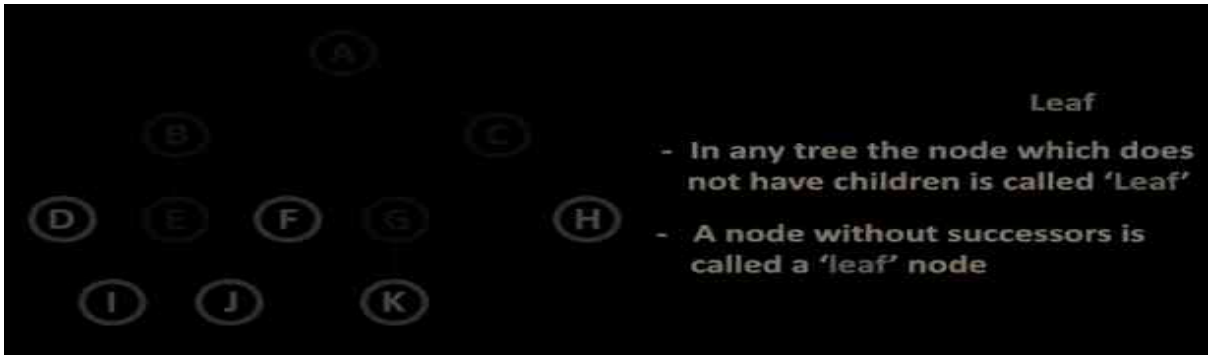
In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with the same parent are called Sibling nodes.




**6. Leaf:**

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.

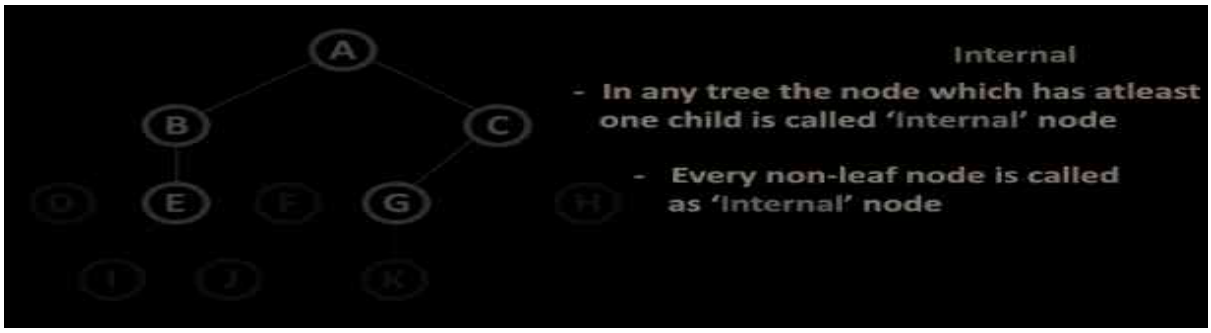
In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.




**7. Internal Nodes:**

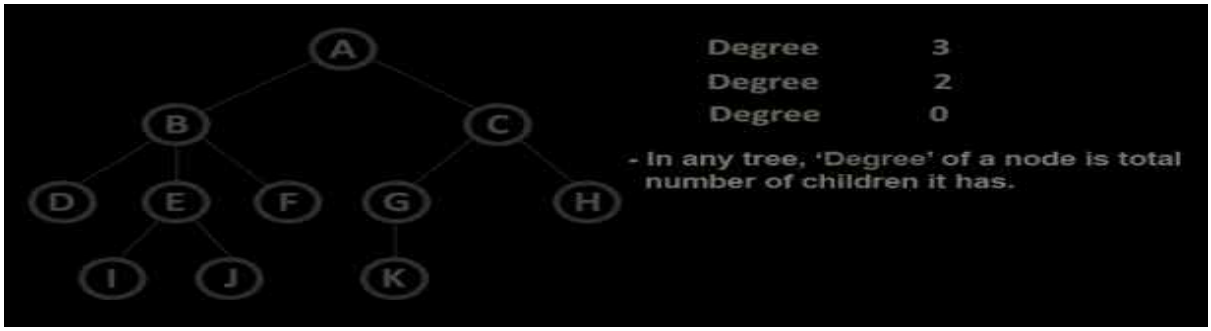
In a tree data structure, the node which has at least one child is called as INTERNAL Node. In simple words, an internal node is a node with at least one child.

In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



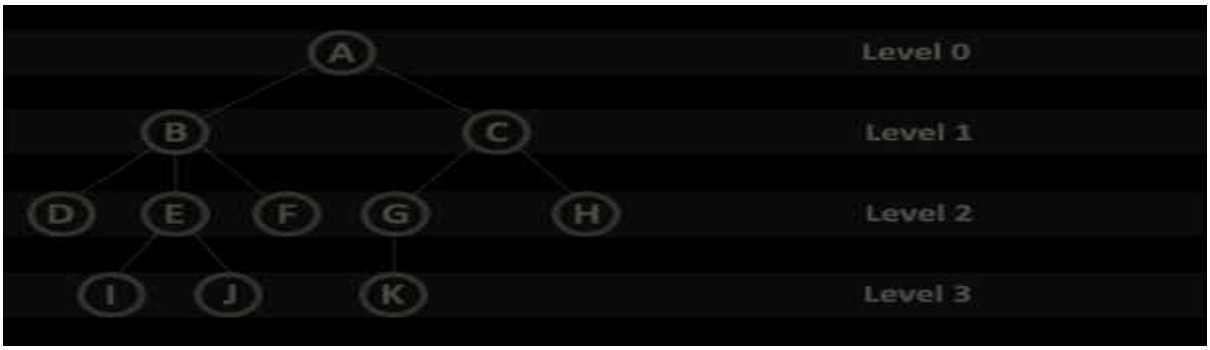

**8. Degree:**

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.



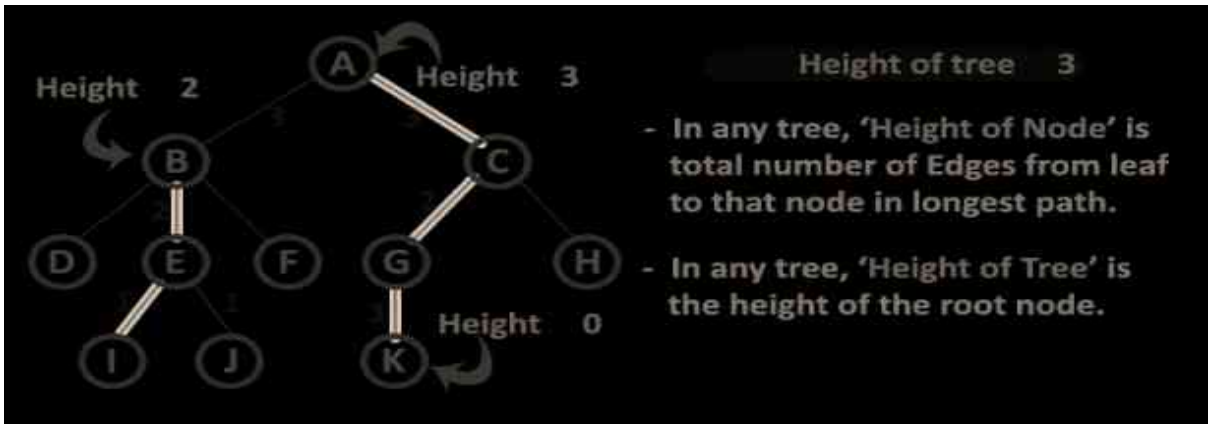

**9. Level:**

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



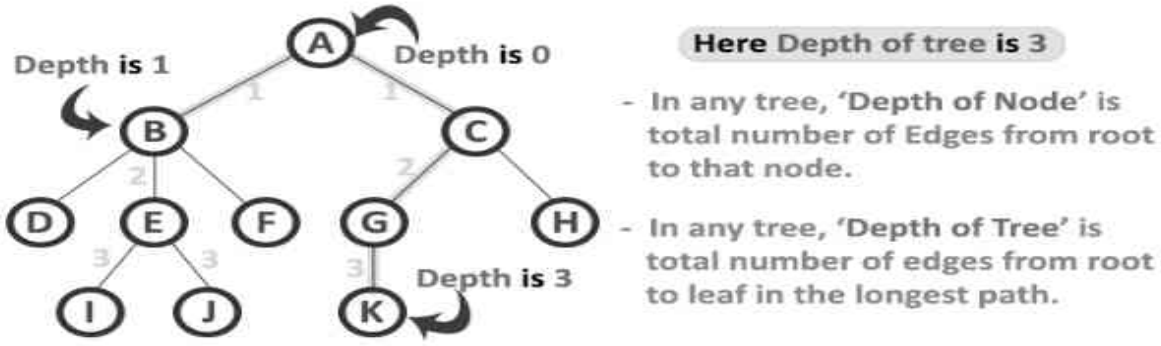

**10. Height:**

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



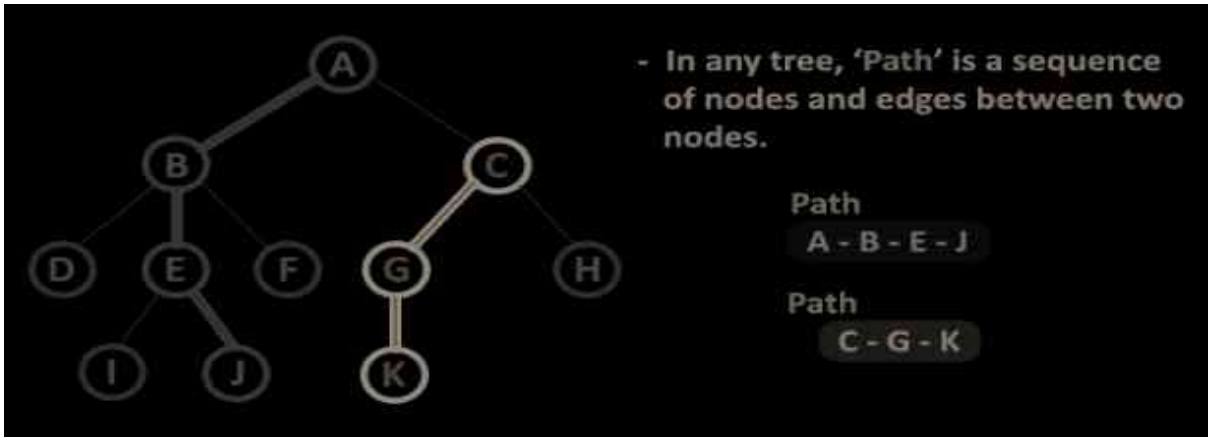

**11. Depth:**

In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.



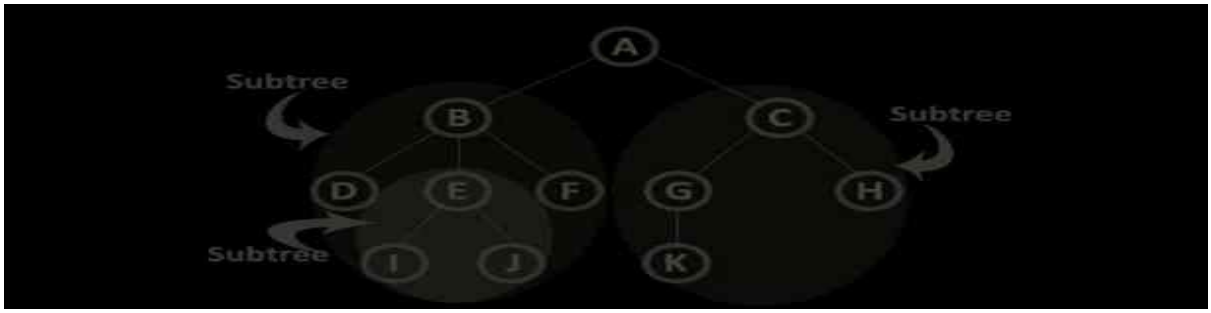

**12. Path:**

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of nodes in that path. In below example the path A - B - E - J has length 4.




**13. Sub Tree:**

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

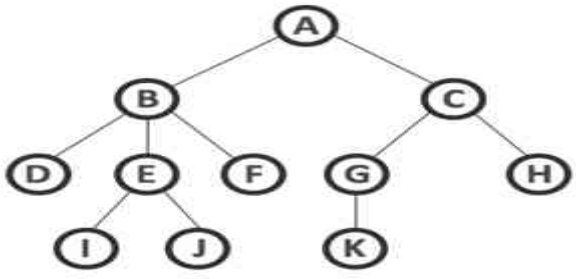



## Tree Representations:

A tree data structure can be represented in two methods. Those methods are as follows...

1. **List Representation**
2. **Left Child - Right Sibling Representation**

Consider the following tree...




**1. List Representation:**

In this representation, we use two types of nodes one for representing the node with data called 'data node' and another for representing only references called 'reference node'. We start with a

'data node' from the root node in the tree. Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly. This process repeats for all the nodes in the tree.

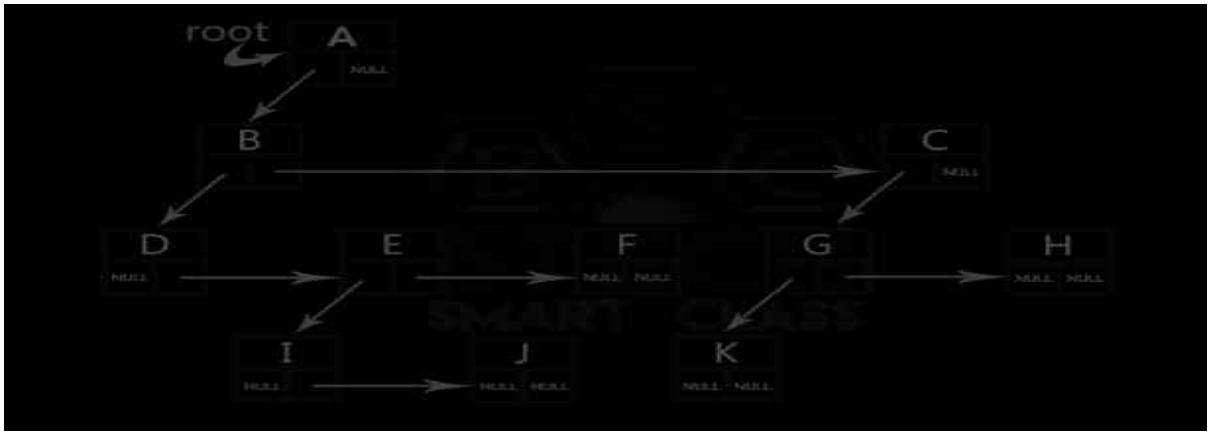
The above example tree can be represented using List representation as follows...


**2. Left Child - Right Sibling Representation:**

In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...


In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

The above example tree can be represented using Left Child - Right Sibling representation as follows...



**Different types of Trees:**

**1. Binary Tree:**

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as a left child and the other is known as right child.

"A tree in which every node can have a maximum of two children is called Binary Tree". In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example:

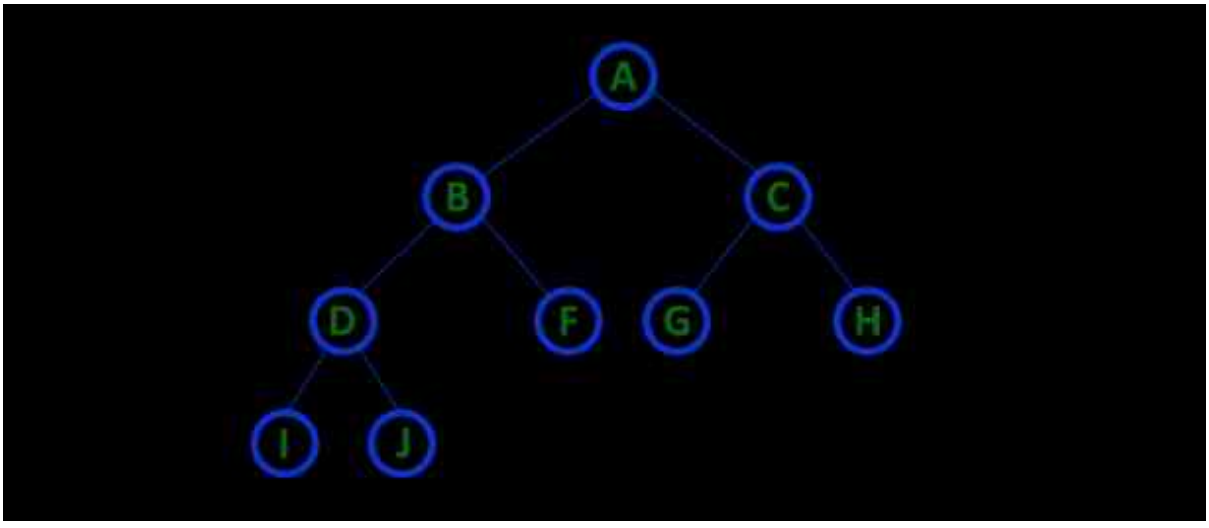
There are different types of binary trees and they are...

**1. Strictly Binary Tree:**

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

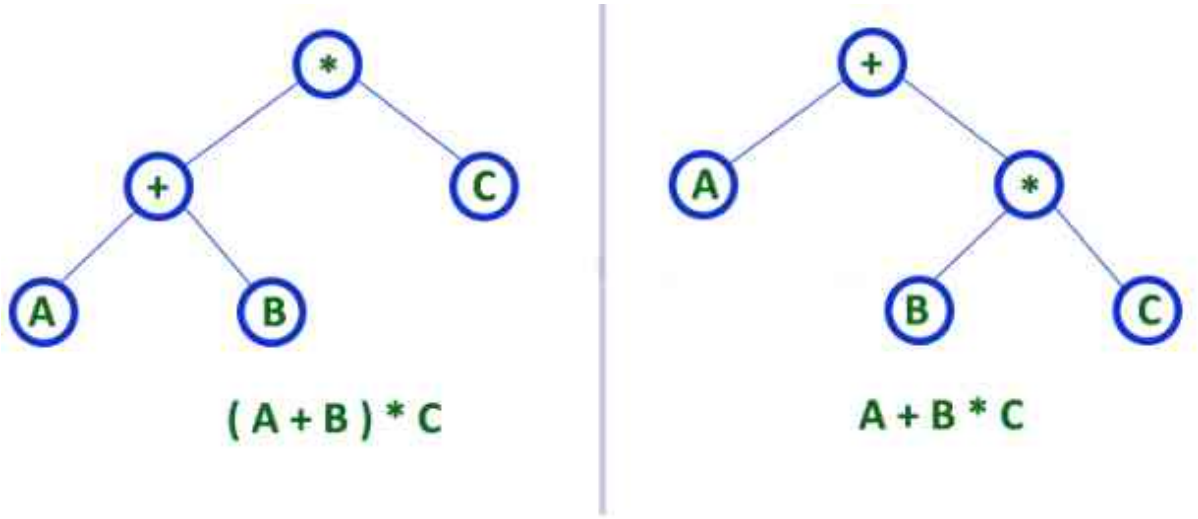
"A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree".

Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.




Strictly binary tree data structure is used to represent mathematical expressions.

**Example:**

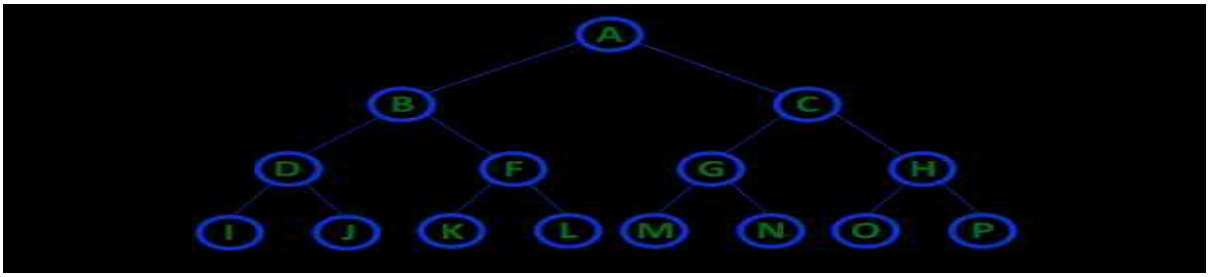



**2. Complete Binary Tree:**

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be $2^{level}$ number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

"A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree."

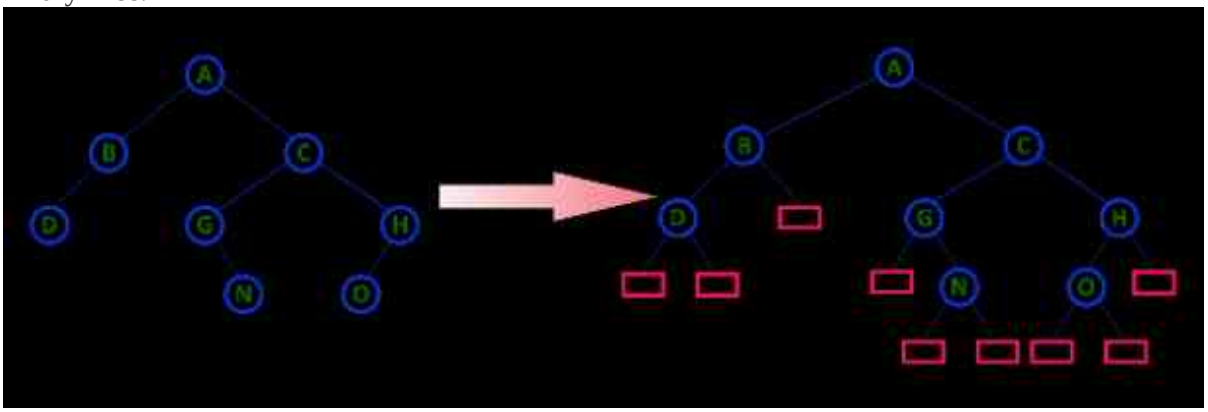
Complete binary tree is also called as **Perfect Binary Tree.**




**3. Extended Binary Tree:**

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



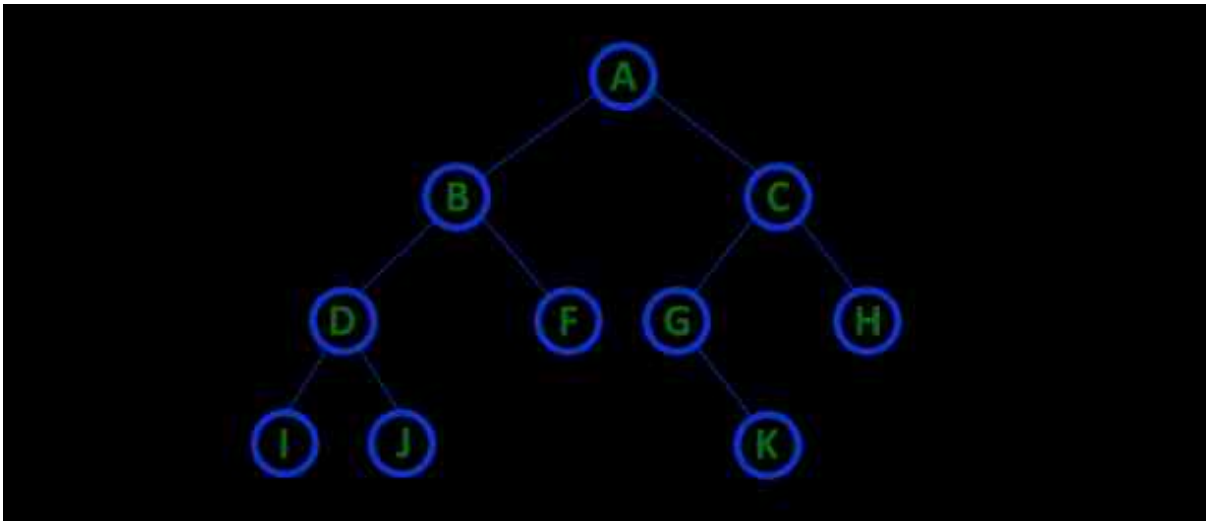
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

**Binary Tree Representations:**

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...




**1. Array Representation of Binary Tree**

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...

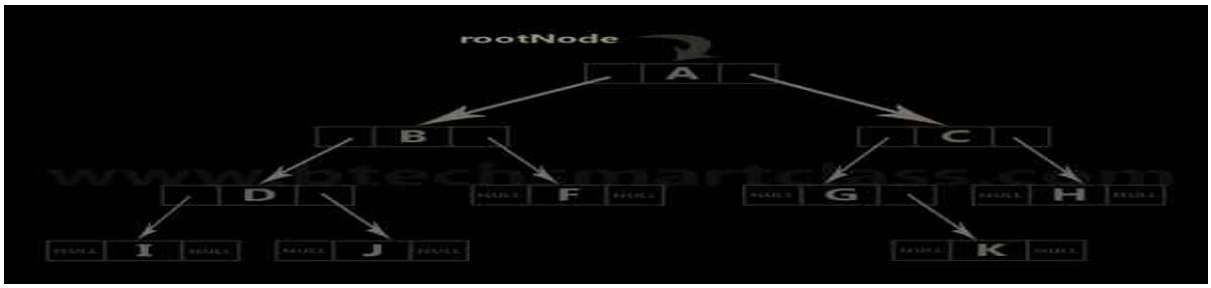

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1}$ -1.

**2. Linked List Representation of Binary Tree:**

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...




**Binary Tree Traversals:**

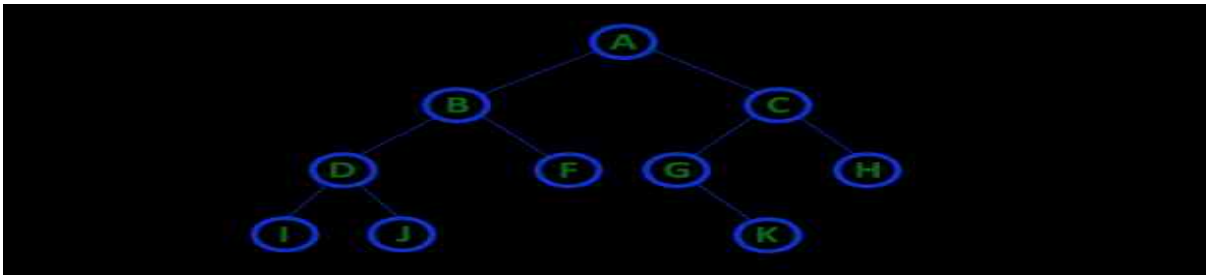
Traversal is a process to visit all the nodes of a tree and may print or display their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree, so we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

'Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal."

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree...




**1. In - Order Traversal ( leftChild - root - rightChild ):**

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left

part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

**2. Pre - Order Traversal ( root - leftChild - rightChild ):**

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K – H

**3. Post - Order Traversal ( leftChild - rightChild - root ):**

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C – A

**Algorithm for Pre-order traversal:**

Until all nodes are traversed −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

**Algorithm for In-order traversal:**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

**Algorithm for Post-order traversal:**

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

**1. Implementation of binary tree:**

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

struct node
{

    int data;
    struct node *left,*right;
};
struct node *root=NULL;
int level=-1;

void create()
{
        if(root==NULL)
        {
        struct node *temp = (struct node*)malloc(sizeof(struct node));
        int value;
        printf("Enter a value : ");
        scanf("%d",&value);
        temp->data = value;
        temp->left = NULL;
        temp->right = NULL;
        root = temp;
        level = 0;
        }
        else
        {
        printf("Root already exists");
        }
}

void Insert()
{
    if(root==NULL){
        printf("Root is NULL");
        printf("Create the tree to insert elements.");
        create();
```

```
    }
    else{
        struct node *temp = (struct node*)malloc(sizeof(struct node));
        int value;
        printf("Enter any value : ");
        scanf("%d",&value);
        temp->data = value;
        temp->left = NULL;
        temp->right = NULL;

        if(root->left == NULL || root->right == NULL)
        {
            if(root->left == NULL){
                root->left = temp;
            }
            else if(root->right == NULL){
                root->right = temp;
            }

            level = 1;
        }

        else if(level ==1 || level == 2)
        {
            if((root->left)->left == NULL){
                (root->left)->left = temp;
            }

            else if((root->left)->right == NULL){
                (root->left)->right = temp;
            }

            else if((root->right)->left == NULL){
                (root->right)->left = temp;
            }

            else if((root->right)->right == NULL){
                (root->right)->right = temp;
            }

            level = 2;
        }
    }
}

void preorder(struct node *temp)
{
    if(temp!=NULL)
```

```
    {
        printf("%d ",temp->data);

        if(temp->left)
            preorder(temp->left);

        if(temp->right)
            preorder(temp->right);
    }
    else{
        printf("Cannot display");
        return;
    }
}

void inorder(struct node *temp)
{
    if(temp!=NULL)
    {
        if(temp->left)
            inorder(temp->left);

        printf("%d ",temp->data);

        if(temp->right)
            inorder(temp->right);
    }
    else{
        printf("Cannot display");
        return;
    }
}

void postorder(struct node *temp)
{
    if(temp!=NULL)
    {
        if(temp->left)
            postorder(temp->left);

        if(temp->right)
            postorder(temp->right);

        printf("%d ",temp->data);
    }
    else{
        printf("Cannot display");
        return;
```

```
    }
}

int main()
{
    int ch,dis;
    while(1)
    {
        printf("\n1.Create\n2.Insert\n3.Display\n0.EXIT\n");
        printf("Enter your choice : ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: create(); break;
            case 2: Insert(); break;
            case 3: printf("1.Preorder\n2.Inorder\n3.Postorder\n");
                    printf("Enter your choice : ");
                    scanf("%d",&dis);
                    switch(dis)
                    {
                        case 1: preorder(root); break;
                        case 2: inorder(root); break;
                        case 3: postorder(root); break;
                        default : printf("Choose the correct option."); break;
                    }
                    break;
            case 0: return 0;
            default : printf("Choose the correct option."); break;
        }
    }
}
```

**2. Implementation of binary tree with different operations:**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *left,*right;
};
struct node *root=NULL;
int level=-1;

void create()
{
        if(root==NULL)
        {
```

```
        struct node *temp = (struct node*)malloc(sizeof(struct node));
        int value;
        printf("Enter a value : ");
        scanf("%d",&value);
        temp->data = value;
        temp->left = NULL;
        temp->right = NULL;
        root = temp;
        level = 0;
        }
        else
        {
        printf("Root already exists");
        }
}

void Insert()
{
    if(root==NULL){
        printf("Root is NULL");
        printf("Create the tree to insert elements.");
        create();
    }
    else{
        struct node *temp = (struct node*)malloc(sizeof(struct node));
        int value;
        printf("Enter any value : ");
        scanf("%d",&value);
        temp->data = value;
        temp->left = NULL;
        temp->right = NULL;

        if(root->left == NULL || root->right == NULL)
        {
            if(root->left == NULL){
                root->left = temp;
            }
            else if(root->right == NULL){
                root->right = temp;
            }

            level = 1;
        }

        else if(level ==1 || level == 2)
        {
            if((root->left)->left == NULL){
                (root->left)->left = temp;
```

```
                }

                else if((root->left)->right == NULL){
                        (root->left)->right = temp;
                }

                else if((root->right)->left == NULL){
                        (root->right)->left = temp;
                }

                else if((root->right)->right == NULL){
                        (root->right)->right = temp;
                }

                level = 2;
        }
    }
}
void preorder(struct node *temp)
{
    if(temp!=NULL)
    {
        printf("%d ",temp->data);

        if(temp->left)
                preorder(temp->left);

        if(temp->right)
                preorder(temp->right);
    }
    else{
        printf("tree doesnot exist");
        return;
    }
}

void inorder(struct node *temp)
{
    if(temp!=NULL)
    {
        if(temp->left)
                inorder(temp->left);

        printf("%d ",temp->data);

        if(temp->right)
                inorder(temp->right);
    }
```

```
    else{
        printf("tree doesnot exist");
        return;
    }
}

void postorder(struct node *temp)
{
    if(temp!=NULL)
    {
        if(temp->left)
            postorder(temp->left);

        if(temp->right)
            postorder(temp->right);

        printf("%d ",temp->data);
    }
    else{
        printf("tree doesnot exist");
        return;
    }
}
void deleteTree(struct node* temp)
{
   if (temp == NULL) return;
   /* first delete both subtrees */
   deleteTree(temp->left);
   deleteTree(temp->right);

   /* then delete the node */
   printf("\n Deleting node: %d", temp->data);
   free(temp);
}
void mirror(struct node* node)
{
  if (node==NULL)
    return;
  else
  {
    struct node* temp;

    /* do the subtrees */
    mirror(node->left);
    mirror(node->right);

    /* swap the pointers in this node */
    temp       = node->left;
```

```
    node->left  = node->right;
    node->right = temp;
   }
}
void printLevelOrder(struct node* root)
{
    int h = height(root);
    int i;
    for (i = 0; i <= h; i++)
        printCurrentLevel(root, i);
}
void printCurrentLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1) {
        printCurrentLevel(root->left, level-1);
        printCurrentLevel(root->right, level-1);
    }
}
int height(struct node* node)
{
    if (node == NULL)
        return 0;
    else {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return (lheight +1);
        else
            return (rheight +1);
    }
}

int main()
{
     int ch,dis;
     while(1)
     {
            printf("\n1.Create\n2.Insert\n3.Display\n4.DeleteTree\n5.mirror\n6.levelorder\n0.EXIT\n");
            printf("Enter your choice : ");
            scanf("%d",&ch);
            switch(ch)
```

```
        {
                case 1: create(); break;
                case 2: Insert(); break;
                case 3: printf("1.Preorder\n2.Inorder\n3.Postorder\n");
                                printf("Enter your choice : ");
                                scanf("%d",&dis);
                                switch(dis)
                                {
                                        case 1: preorder(root); break;
                                        case 2: inorder(root); break;
                                        case 3: postorder(root); break;
                                        default : printf("Choose the correct option."); break;
                                }
                                break;
                case 4: deleteTree(root);
                        root=NULL;
                        break;
                case 5: mirror(root);
                      break;
                case 6: printLevelOrder(root);
                      break;
                case 0: return 0;
                default : printf("Choose the correct option."); break;
        }
    }
}
```

## Threaded Binary Trees:

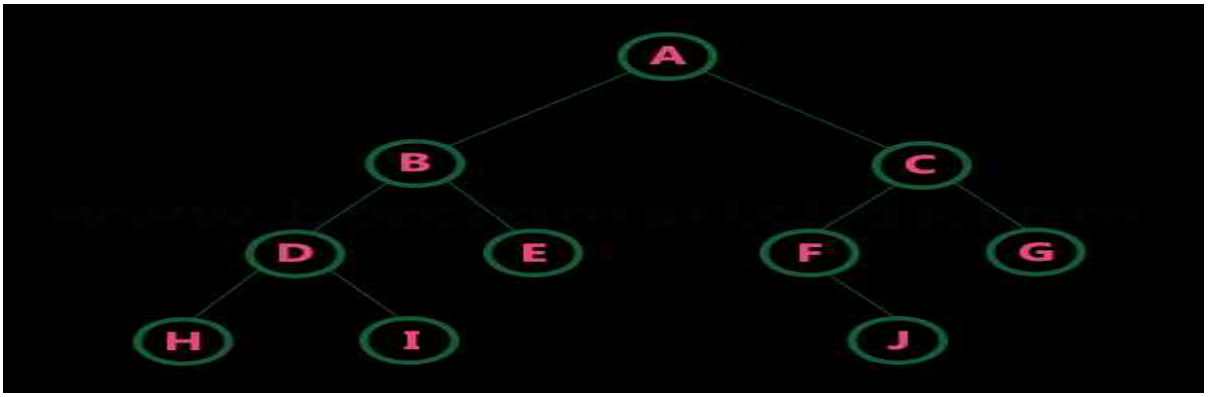
A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list representation, there is a number of NULL pointers are more than actual pointers i.e., if there are **2N** number of reference fields, then **N+1** number of reference fields are filled with NULL ( **N+1 are NULL out of 2N** ). This NULL pointer does not play any role except indicating that there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "***Threaded Binary Tree***", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as ***threads***.

"Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor."

If there is no in-order predecessor or in-order successor, then it points to the root node.

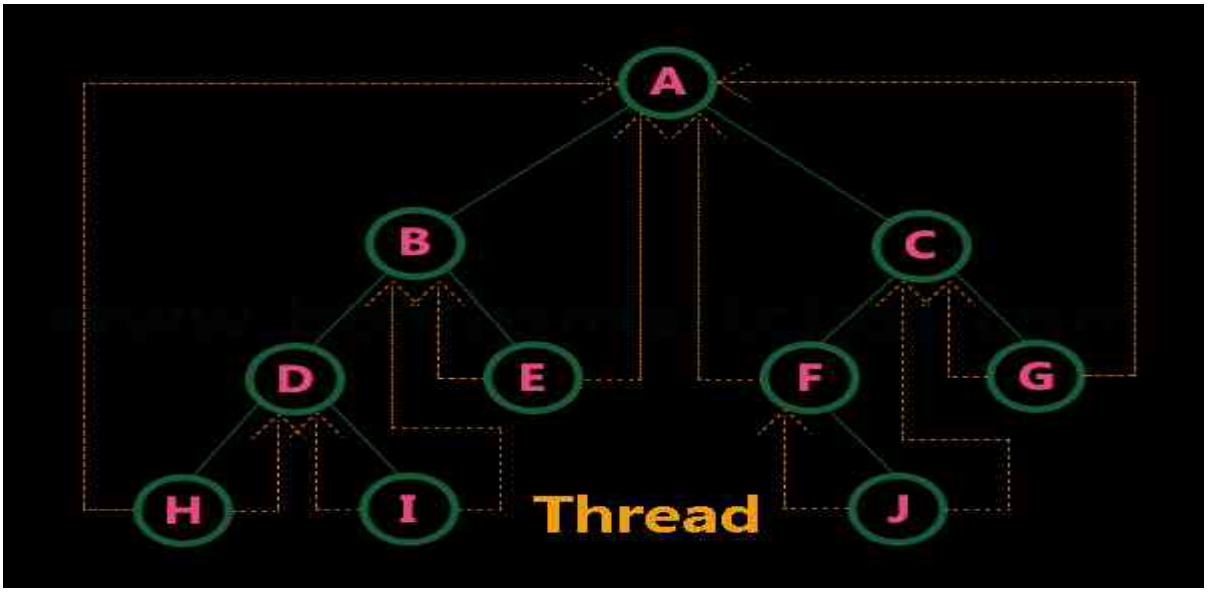
Consider the following binary tree...




To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...
In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C – G

When we represent the above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.
Above example binary tree is converted into threaded binary tree as follows.




In the above figure, threads are indicated with dotted links.

## Binary Search Tree:

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.
A binary tree has the following time complexities...

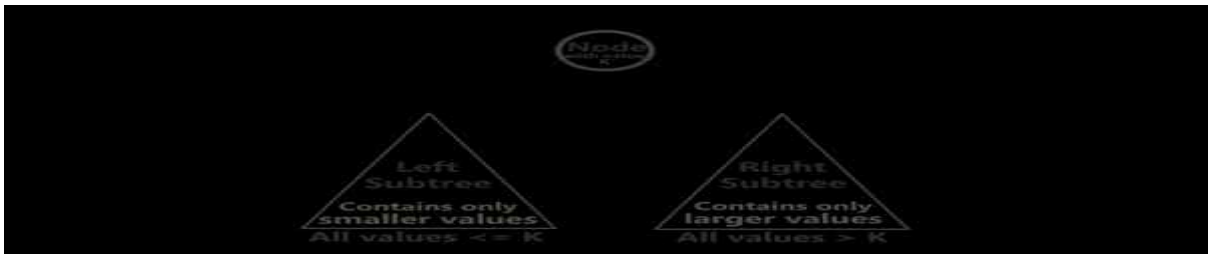
- Search Operation - O(n)
- Insertion Operation - O(1)

- Deletion Operation - O(n)

To enhance the performance of binary tree, we use a special type of binary tree known as Binary Search Tree. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

"Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree."

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...




**Example:**

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.


"Every binary search tree is a binary tree but every binary tree need not to be binary search tree."

**Operations on a Binary Search Tree:**

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

### 1. Search Operation in BST:

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.

Step 6- If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node
Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

### 2. Insert Operation in BST:

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Create a newNode with given value and set its left and right to NULL.
Step 2 - Check whether tree is Empty.
Step 3 - If the tree is Empty, then set root to newNode.
Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

### 3. Delete Operation in BST:

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

**Case 1: Deleting a leaf node**
We use the following steps to delete a leaf node from BST...
Step 1 - Find the node to be deleted using search operation
Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

**Case 2: Deleting a node with one child**
We use the following steps to delete a node with one child from BST...
Step 1 - Find the node to be deleted using search operation
Step 2 - If it has only one child then create a link between its parent node and child node.
Step 3 - Delete the node using free function and terminate the function.

**Case 3: Deleting a node with two children**
We use the following steps to delete a node with two children from BST...
Step 1 - Find the node to be deleted using search operation
Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
Step 3 - Swap both deleting node and node which is found in the above step.
Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.
Step 7 - Repeat the same process until the node is deleted from the tree.

**Implementation:**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *left,*right;
};
struct node *root=NULL;
void insert()
{
struct node *cur,*parent=NULL;
int n;
struct node *temp = (struct node*)malloc(sizeof(struct node));
printf("enter a number");
scanf("%d",&n);
temp->data=n;
temp->left=NULL;
temp->right=NULL;
if(root==NULL)
{
root=temp;
}
else
{
cur=root;
while(cur)
{
parent=cur;
if(temp->data>cur->data)
{
cur=cur->right;
}
else
{
cur=cur->left;
}
}
if(temp->data>parent->data)
{
parent->right=temp;
}
else
{
parent->left=temp;
}
}
}

void delet()
{
struct node *cur=root,*parent=NULL;
```

```
int n;
printf("enter a node data which u want to delete");
scanf("%d",&n);
if(root==NULL)
{
    printf("Tree is empty");
    return;
}
else
{
while(cur!=NULL)
{
    if(cur->data==n)
    {
          break;
    }
    else
    {
          parent=cur;
          if(n>cur->data)
                {
                cur=cur->right;
                }
          else
                {
                cur=cur->left;
                }
    }
}
if(cur==NULL)
{
    printf("Invalid data node.Try again");
    return;
}
}
//Leaf Node
if( cur->left == NULL && cur->right == NULL)
   {
      if(parent->left == cur)
      {
      parent->left = NULL;
      }
      else
      {
      parent->right = NULL;
          }
    return;
    }
//Node with single child

if((cur->left == NULL && cur->right != NULL)|| (cur->left != NULL&& cur->right == NULL))
 {
     if(cur->left == NULL && cur->right != NULL)
     {
        if(parent->left == cur)
```

```
            {
              parent->left = cur->right;
              }
            else
            {
              parent->right = cur->right;

            }
          }
          else // left child present, no right child
          {
            if(parent->left == cur)
            {
              parent->left = cur->left;

            }
            else
            {
              parent->right = cur->left;

            }
          }
        return;
    }

    //Nodes have 2 child nodes
    if (cur->left != NULL && cur->right != NULL)
    {
        struct node *t1,*t2;
        t1=cur->right;
        if(t1->left==NULL && t1->right==NULL)
        {
        cur->data=t1->data;
        cur->right=NULL;
         //delete t1;
        }
        else
        {
              if((cur->right)->left != NULL)
                 {
                    struct node *rcur;
                    struct node *rcurp;
                    rcurp = cur->right;
                    rcur = (cur->right)->left;
                    while(rcur->left != NULL)
                    {
                      rcurp = rcur;
                      rcur = rcur->left;
                    }
                        cur->data = rcur->data;
                    //delete rcur;
                    rcurp->left = NULL;
                 }
                 else
                 {
```

```
                struct node *tmp;
                tmp = cur->right;
                cur->data = tmp->data;
                cur->right = tmp->right;
                //delete tmp;
              }

     }
     return;
}
}

void search(struct node *root,int key)
{
if(root==NULL)
{
printf("Tree is empty/Element not found");
}
else if(root->data==key)
{
printf("element is found");
}
else if(root->data<key)
{
search(root->right,key);
}
else
{
search(root->left,key);
}
}
void preorder(struct node *t)
{
if(t != NULL)
   {
      printf("%d ",t->data);
      if(t->left) preorder(t->left);
      if(t->right) preorder(t->right);
   }
   else return;
}

void inorder(struct node *t)
{
if(t != NULL)
   {
      if(t->left) inorder(t->left);
      printf("%d ",t->data);
      if(t->right) inorder(t->right);
   }
   else return;
}
void postorder(struct node *t)
{
if(t != NULL)
```

```
    {
       if(t->left) postorder(t->left);
       if(t->right) postorder(t->right);
       printf("%d ",t->data);

    }
   else return;
}

int main()
{
int ch,n;
while(1)
   {
      printf("\n");
      printf(" Binary Search Tree Operations\n ");
      printf(" ------------------------------ ");
      printf(" \n1. Insertion/Creation ");
      printf(" \n2. Pre-Order Traversal ");
      printf(" \n3. In-Order Traversal ");
      printf(" \n4. Post-Order Traversal ");
      printf(" \n5. Delete ");
      printf(" \n6. Search ");
      printf(" \n7. Exit ");
      printf(" \nEnter your choice : ");
      scanf("%d",&ch);
       switch(ch)
      {
         case 1 : insert();
                  break;
         case 2 : preorder(root);
                  break;
         case 3 : inorder(root);
                  break;
         case 4 : postorder(root);
                  break;
         case 5 : delet();
                  break;
         case 6: printf("enter an element to be search");
                           scanf("%d",&n);
                           search(root,n);
                           break;
         case 7: return 0;
         default: printf("Selct valid option");
                  break;
       }
      }
 }
```

## AVL Tree:

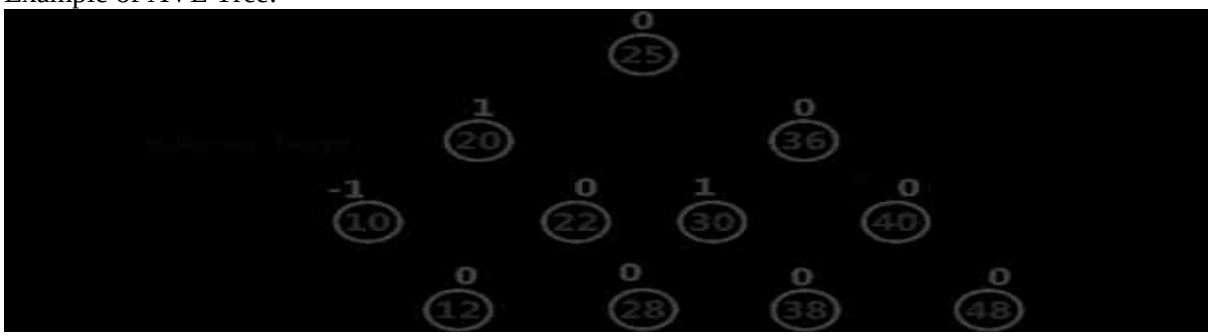
AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right subtree of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as balance factor. The AVL tree was introduced in the year 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...
"An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1."

Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we calculate as follows...

"Balance factor = heightOfLeftSubtree – heightOfRightSubtree"

Example of AVL Tree:

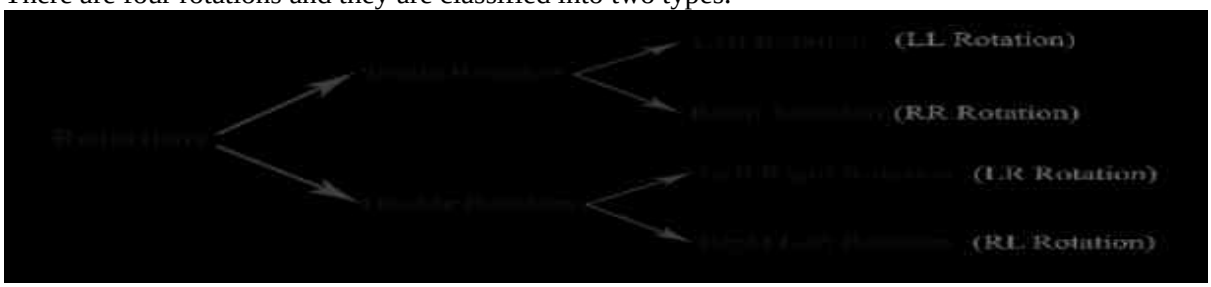



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.
"Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree."

**AVL Tree Rotations:**

In AVL tree, after performing operations like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. Whenever the tree becomes imbalanced due to any operation we use rotation operations to make the tree balanced.

Rotation operations are used to make the tree balanced.
"Rotation is the process of moving nodes either to left or to right to make the tree balanced."
There are four rotations and they are classified into two types.




**Single Left Rotation (LL Rotation):**

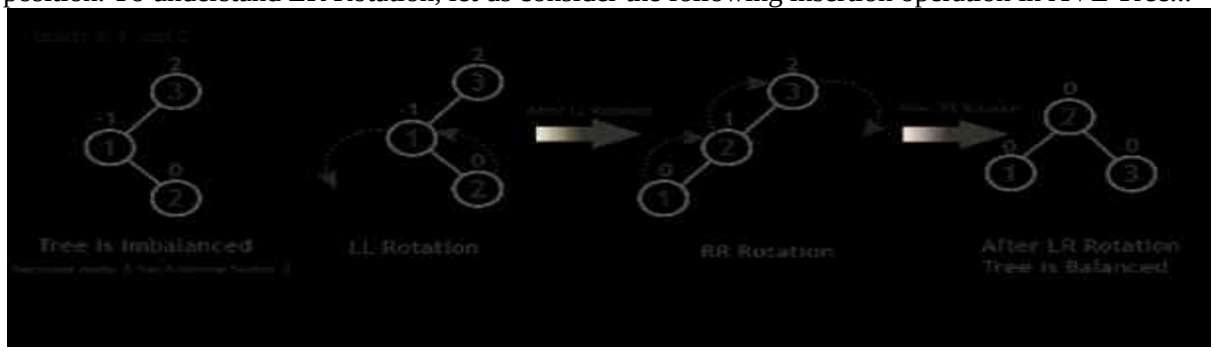
In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...


**Single Right Rotation (RR Rotation):**

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

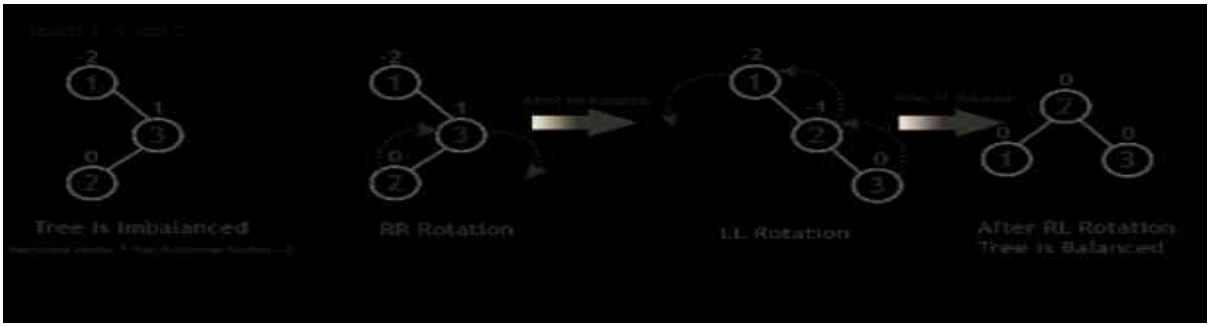

**Left Right Rotation (LR Rotation):**

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...




**Right Left Rotation (RL Rotation):**

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...




**Operations on an AVL Tree:**

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

**Search Operation in AVL Tree:**

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the value of root node in the tree.
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
Step 5 - If search element is smaller, then continue the search process in left subtree.
Step 6 - If search element is larger, then continue the search process in right subtree.
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
Step 8 - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

**Insertion Operation in AVL Tree:**

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
Step 2 - After insertion, check the Balance Factor of every node.
Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

**Deletion Operation in AVL Tree:**

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

**Implementation of AVL Tree with different operations:**

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

```
#include<stdlib.h>

// An AVL tree node
struct node
{
        int key;
        struct node *left,*right;
        int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct node *N)
{
   if (N == NULL)
      return 0;
   return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
   return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and NULL left and right
pointers. */
struct node* newNode(int key)
{
   struct node *temp = (struct node*)malloc(sizeof(struct node));
   temp->key = key;
   temp->left = NULL;
   temp->right = NULL;
   temp->height = 1; // new node is initially
                // added at leaf
   return(temp);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
struct node* rightRotate(struct node *y)
{
   struct node *x = y->left;
   struct node *T2 = x->right;

   // Perform rotation
   x->right = y;
   y->left = T2;

   // Update heights
   y->height = max(height(y->left),
            height(y->right)) + 1;
```

```
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
struct node* leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) -
           height(N->right);
}

struct node* insert(struct node *temp, int key)
{
    /* 1. Perform the normal BST rotation */
    if (temp == NULL)
        return(newNode(key));

    if (key < temp->key)
        temp->left = insert(temp->left, key);
    else if (key > temp->key)
        temp->right = insert(temp->right, key);
    else // Equal keys not allowed
        return temp;

    /* 2. Update height of this ancestor node */
    temp->height = 1 + max(height(temp->left),
                           height(temp->right));
```

```
    /* 3. Get the balance factor of this
       ancestor node to check whether
       this node became unbalanced */
    int balance = getBalance(temp);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < temp->left->key)
        return rightRotate(temp);

    // Right Right Case
    if (balance < -1 && key > temp->right->key)
        return leftRotate(temp);

    // Left Right Case
    if (balance > 1 && key > temp->left->key)
    {
        temp->left = leftRotate(temp->left);
        return rightRotate(temp);
    }

    // Right Left Case
    if (balance < -1 && key < temp->right->key)
    {
        temp->right = rightRotate(temp->right);
        return leftRotate(temp);
    }

    return temp;
}


struct node* minValueNode(struct node *temp)
{
    struct node *current = temp;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node
// with given key from subtree with
// given root. It returns root of the
// modified subtree.
struct node* deleteNode(struct node *root, int key)
{

    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == NULL)
        return root;
```

```
// If the key to be deleted is smaller
// than the root's key, then it lies
// in left subtree
if ( key < root->key )
   root->left = deleteNode(root->left, key);

// If the key to be deleted is greater
// than the root's key, then it lies
// in right subtree
else if( key > root->key )
   root->right = deleteNode(root->right, key);

// if key is same as root's key, then
// This is the node to be deleted
else
{
   // node with only one child or no child
   if( (root->left == NULL) ||
      (root->right == NULL) )
   {
      struct node *temp = root->left ?
                root->left :
                root->right;

      // No child case
      if (temp == NULL)
      {
         temp = root;
         root = NULL;
      }
      else // One child case
      *root = *temp; // Copy the contents of
                // the non-empty child
     // free(temp);
   }
   else
   {
      // node with two children: Get the inorder
      // successor (smallest in the right subtree)
      struct node *temp = minValueNode(root->right);

      // Copy the inorder successor's
      // data to this node
      root->key = temp->key;

      // Delete the inorder successor
      root->right = deleteNode(root->right,
                        temp->key);
   }
}

// If the tree had only one node
// then return
if (root == NULL)
```

```
    return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left),
                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF
    // THIS NODE (to check whether this
    // node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced,
    // then there are 4 cases

    // Left Left Case
    if (balance > 1 &&
        getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 &&
        getBalance(root->left) < 0)
    {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 &&
        getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 &&
        getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder
// traversal of the tree.
// The function also prints height
// of every node
void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ",root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
```

```
}

// Driver Code
int main()
{
struct node *root = NULL;

   /* Constructing tree given in
   the above figure */
   root = insert(root, 9);
   root = insert(root, 5);
   root = insert(root, 10);
   root = insert(root, 0);
   root = insert(root, 6);
   root = insert(root, 11);
   root = insert(root, -1);
   root = insert(root, 1);
   root = insert(root, 2);

   /* The constructed AVL Tree would be
         9
      / \
      1 10
      / \ \
   0 5 11
   / / \
   -1 2 6
   */

   printf("Preorder traversal of the constructed AVL tree is \n");
   preOrder(root);

   root = deleteNode(root, 10);

   /* The AVL Tree after deletion of 10
         1
      / \
      0 9
      / / \
   -1 5    11
      / \
      2 6
   */

   printf("\nPreorder traversal after deletion of 10 \n");
   preOrder(root);

   return 0;
}
```

# Heaps:

Heap data structure is a specialized binary tree-based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values. A heap data structure sometimes also called as Binary Heap.

There are two types of heap data structures and they are as follows...

1. Max Heap
2. Min Heap

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

**Max Heap:**

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

"Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes."

**Example:**


Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

**Operations on Max Heap:**

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

**Finding Maximum Value Operation in Max Heap:**

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

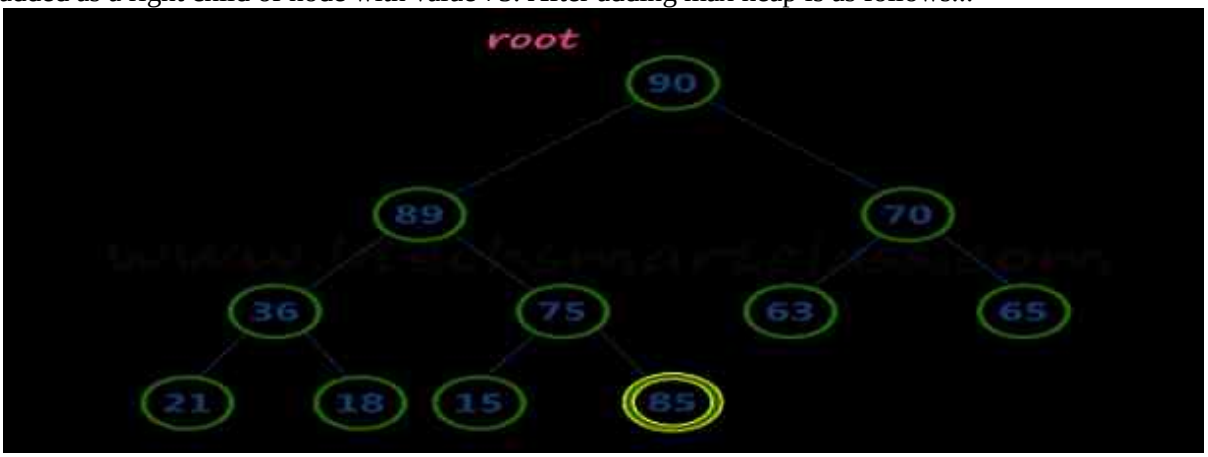
**Insertion Operation in Max Heap:**

Insertion Operation in max heap is performed as follows...

Step 1 - Insert the newNode as last leaf from left to right.
Step 2 - Compare newNode value with its Parent node.
Step 3 - If newNode value is greater than its parent, then swap both of them.
Step 4 - Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

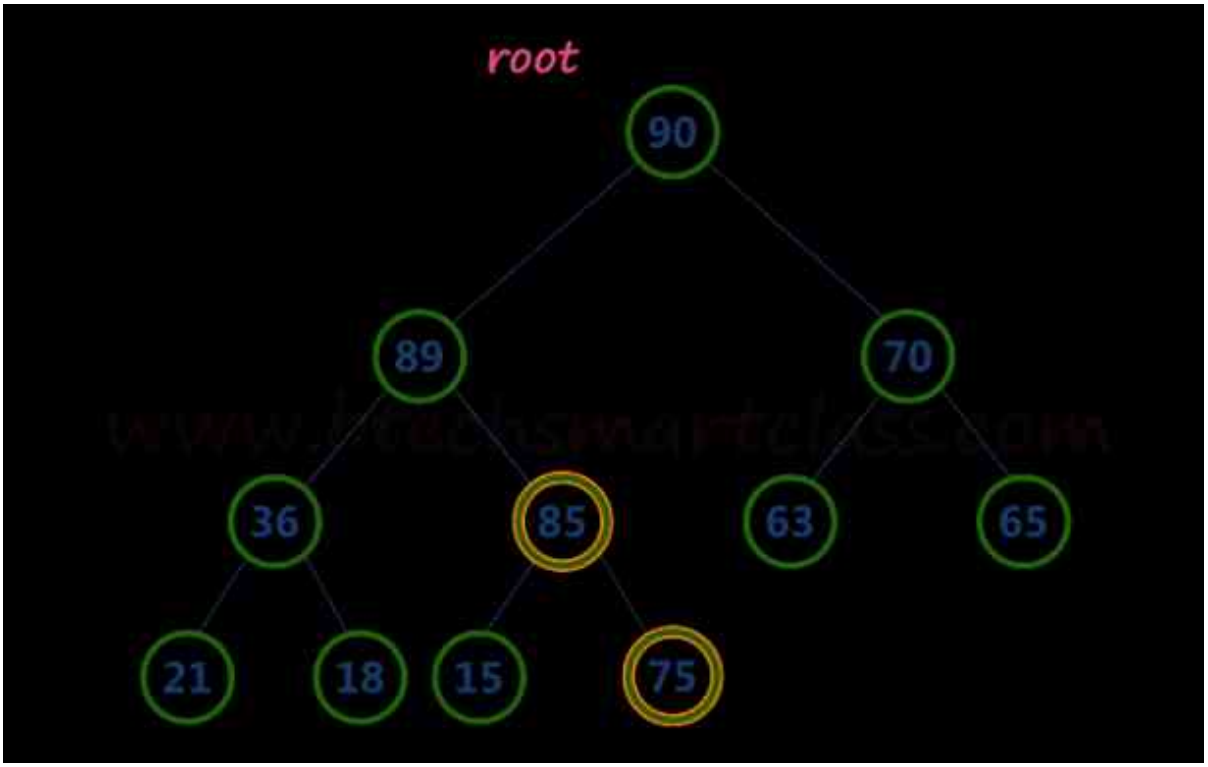
**Example:**
Consider the above max heap. Insert a new node with value 85.

Step 1 - Insert the newNode with value 85 as last leaf from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...

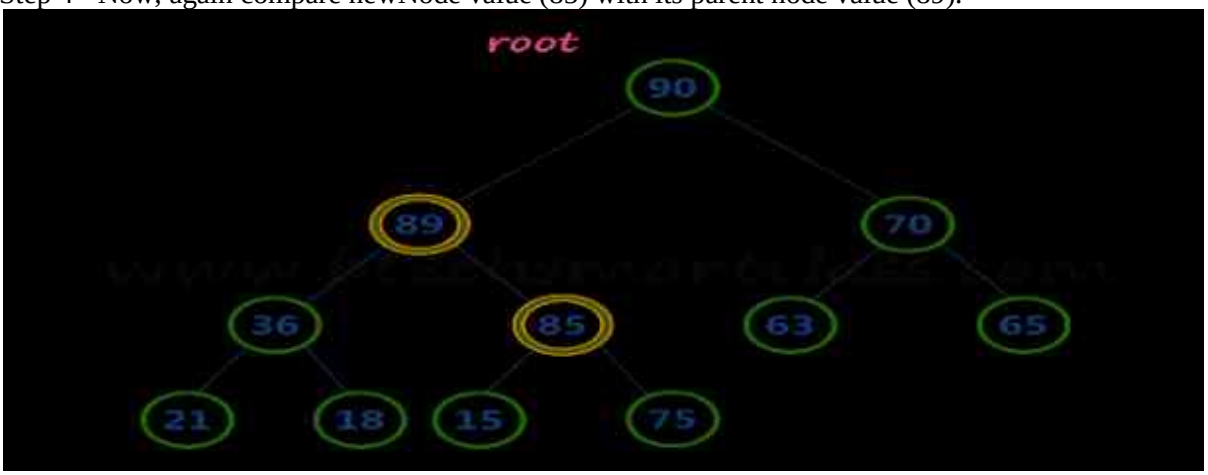



Step 2 - Compare newNode value (85) with its Parent node value (75). That means 85 > 75

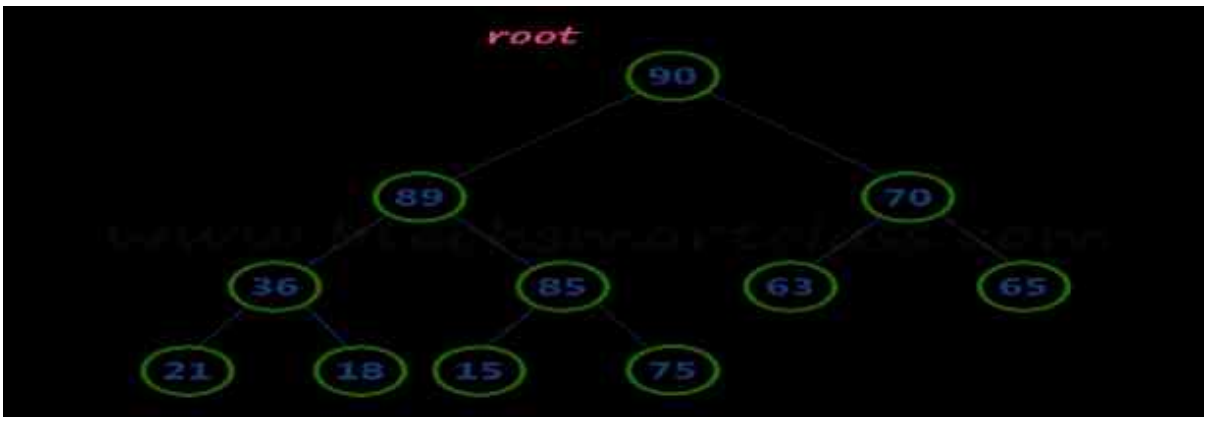

Step 3 - Here newNode value (85) is greater than its parent value (75), then swap both of them. After swapping, max heap is as follows...




Step 4 - Now, again compare newNode value (85) with its parent node value (89).




Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, max heap after insertion of a new node with value 85 is as follows...



**Deletion Operation in Max Heap:**

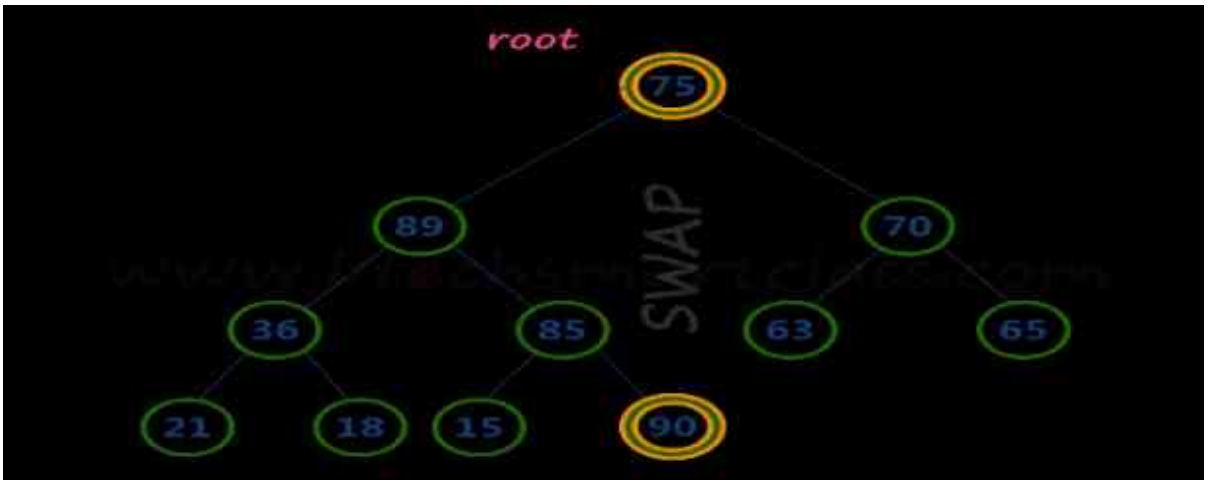
In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

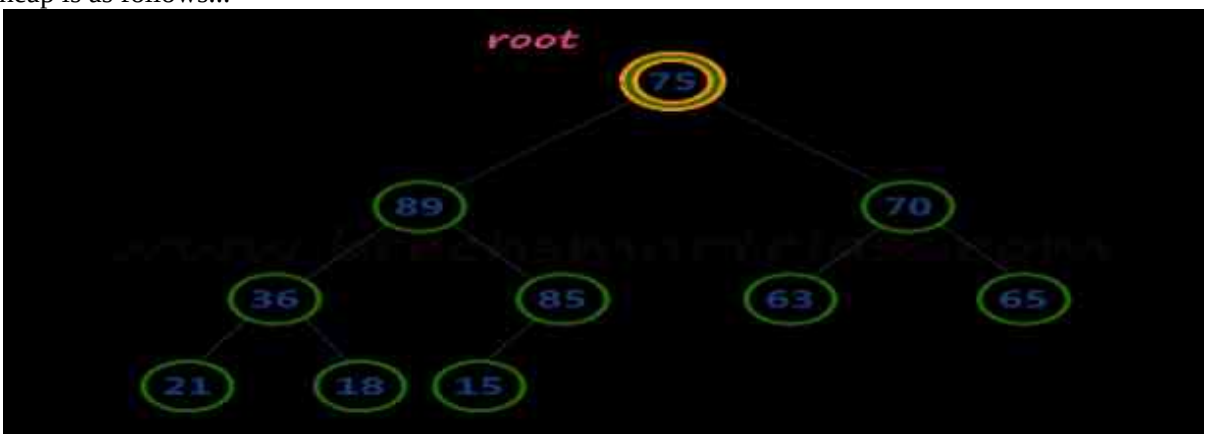
Step 1 - Swap the root node with last node in max heap
Step 2 - Delete last node.
Step 3 - Now, compare root value with its left child value.
Step 4 - If root value is smaller than its left child, then compare left child with its right sibling. Else goto Step 6
Step 5 - If left child value is larger than its right sibling, then swap root with left child otherwise swap root with its right child.
Step 6 - If root value is larger than its left child, then compare root value with its right child value.
Step 7 - If root value is smaller than its right child, then swap root with right child otherwise stop the process.
Step 8 - Repeat the same until root node fixes at its exact position.

**Example:**
Consider the above max heap. Delete root node (90) from the max heap.

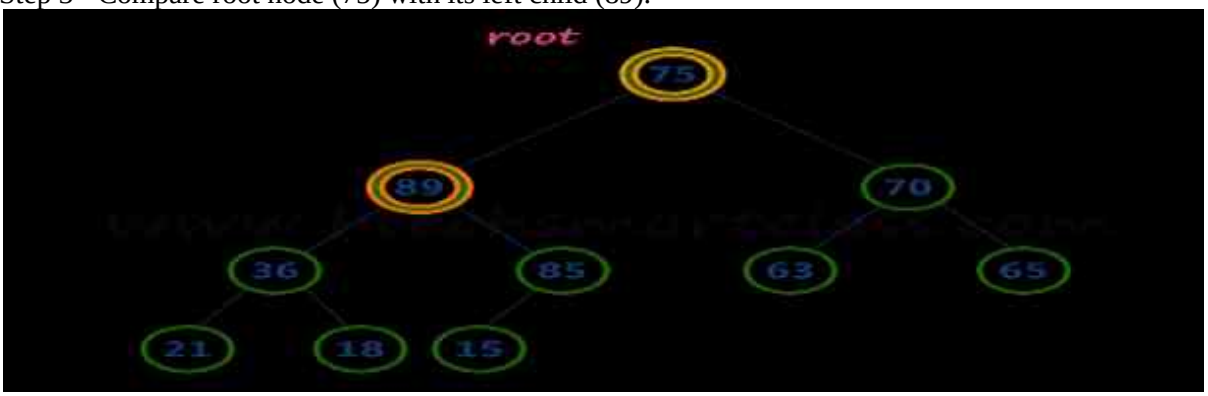
Step 1 - Swap the root node (90) with last node 75 in max heap. After swapping max heap is as follows...



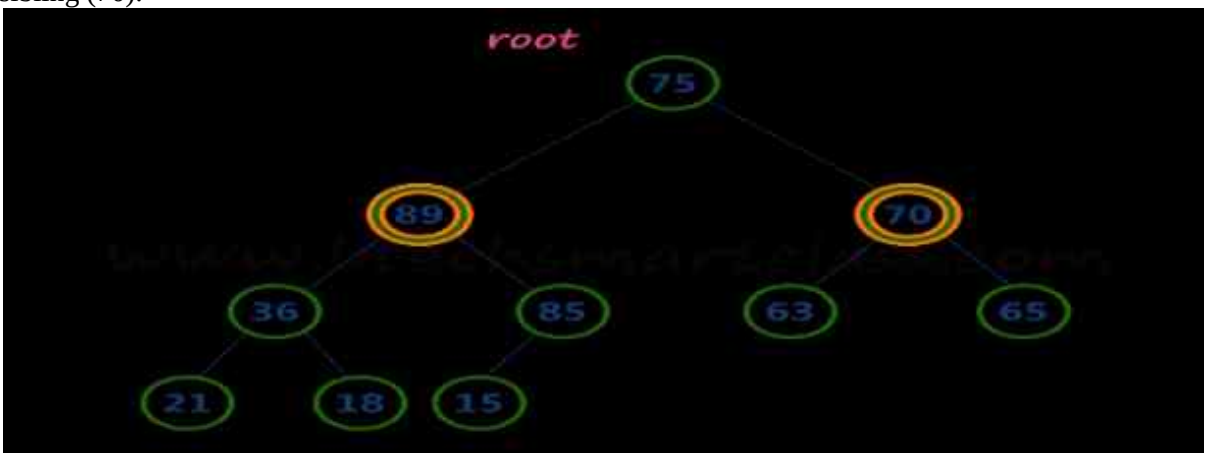

Step 2 - Delete last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



Step 3 - Compare root node (75) with its left child (89).

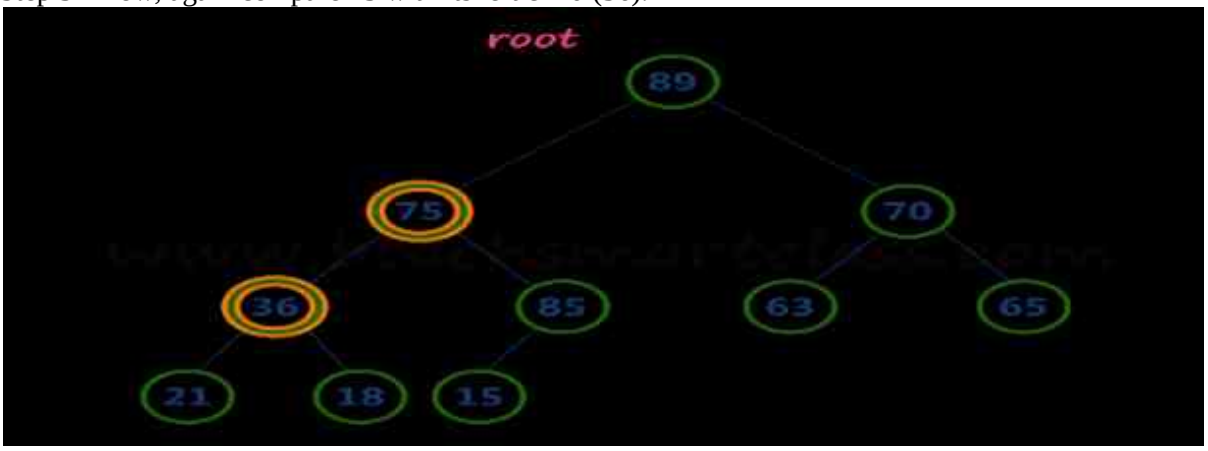



Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).

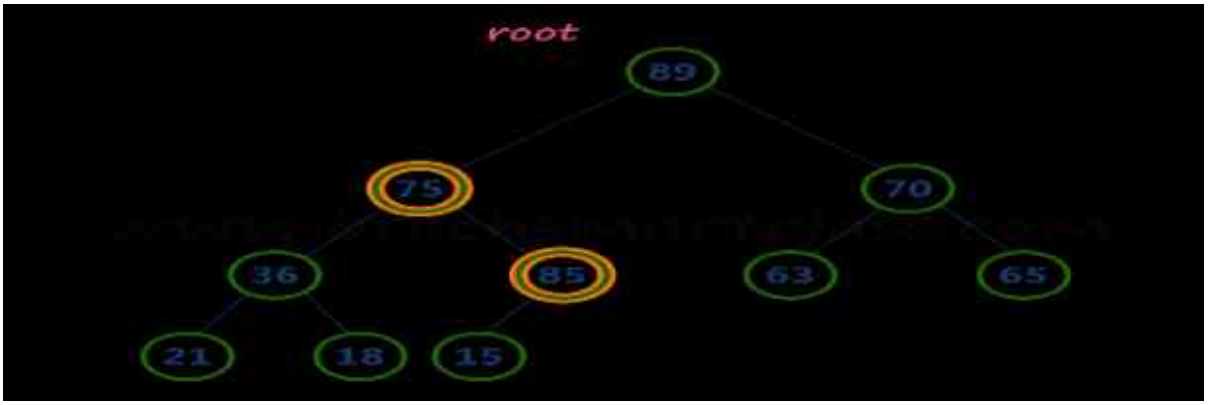



Step 4 - Here, left child value (89) is larger than its right sibling (70), So, swap root (75) with left child (89).

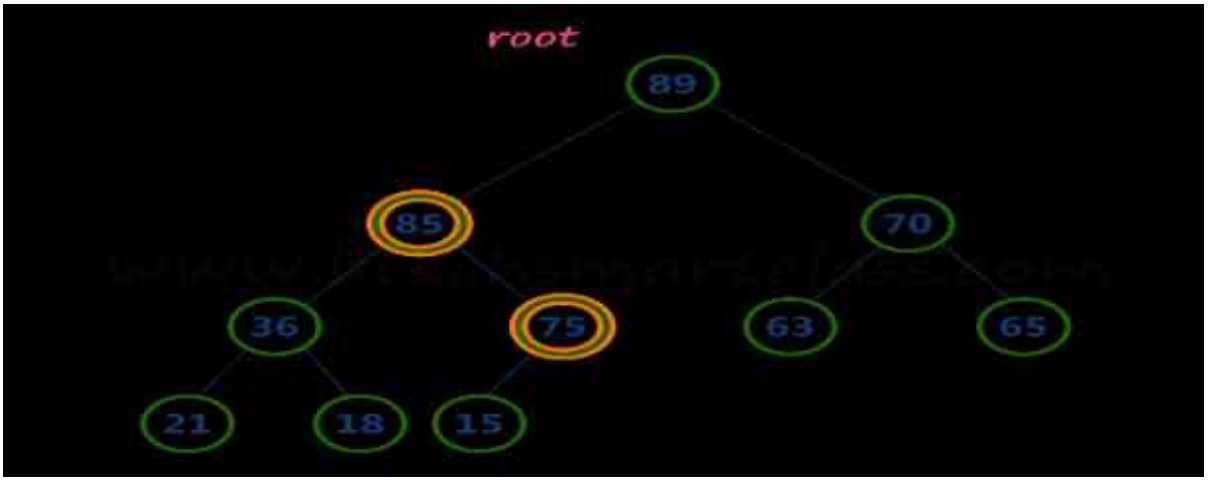

Step 5 - Now, again compare 75 with its left child (36).



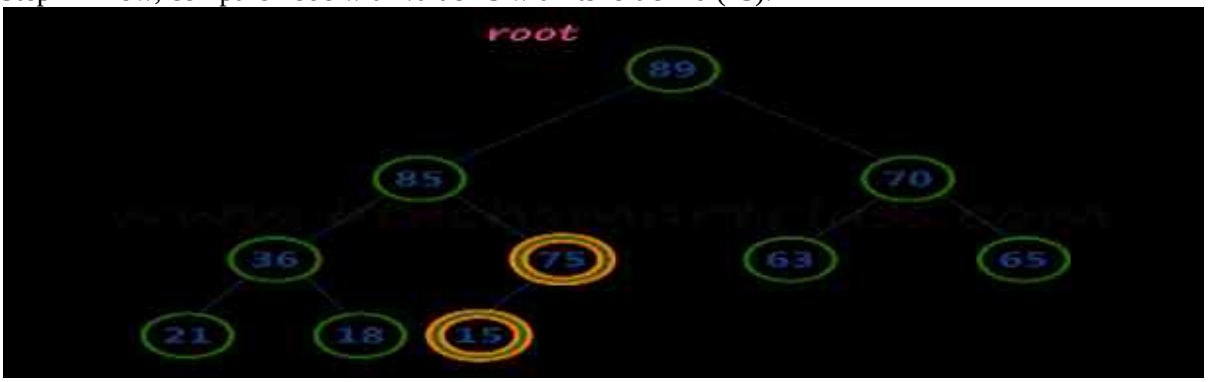

Here, node with value 75 is larger than its left child. So, we compare node 75 with its right child 85.




Step 6 - Here, node with value 75 is smaller than its right child (85). So, we swap both of them. After swapping max heap is as follows...

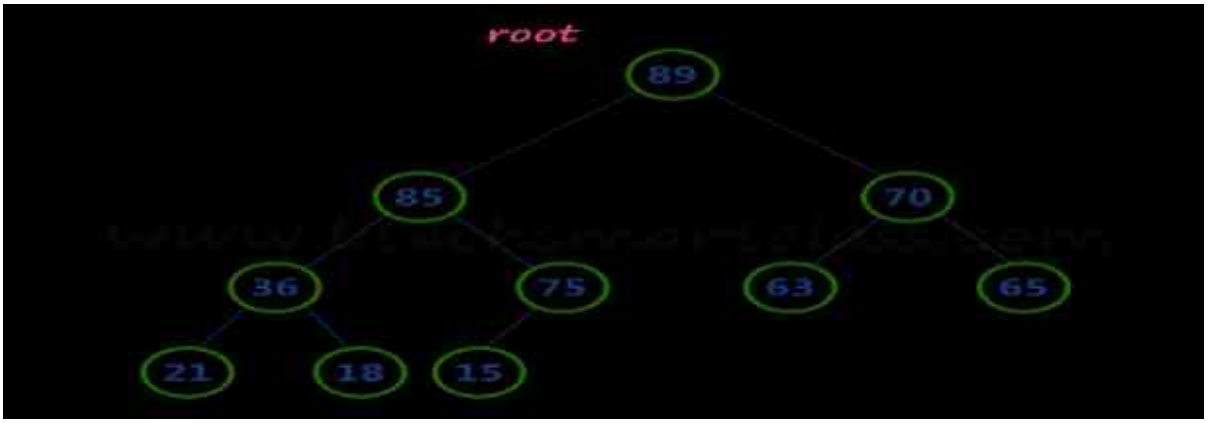



Step 7 - Now, compare node with value 75 with its left child (15).




Here, node with value 75 is larger than its left child (15) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (90) is as follows...


root
89
85
70
36
75
63
65
21
18
15

**Unit-V:**
*Graphs:* Graph Abstract Data Type, Elementary Graph operations (DFS and BFS), Minimum Cost Spanning Trees (Prim's and Kruskal's Algorithms).
*Sorting and Searching:* Insertion sort, Quick sort, Best computing time for Sorting, Merge sort, Heap sort, shell sort, Sorting on Several Keys, List and Table Sorts, Summary of Internal Sorting, Linear and Binary Search algorithms.
**Objective:**
To introduce various searching and sorting techniques.
**Outcome:** Interpret suitable searching and sorting techniques and implement non linear data structures using graphs to solve various computing problems.

# Sorting and Searching

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending). There are different types sorting techniques. They are

**1. Selection Sort:**

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

In selection sort, element at first location (0th location) is considered as least element, and it is compared with the other elements of the array. If any element is found to be minimum than the element in first location, then that location is taken as minimum and element in that location will be the minimum element.

After completing a set of comparisons, the minimum element is swapped with the element in first location (0th location).

Then again element second location is considered as minimum and it is compared with the other elements of array and the process continues till the array is sorted in ascending order.
Note: After first pass, smallest element in given list occupies the first position. After second pass, second largest element is placed at second position and so on..

***Step by Step Process:***

The selection sort algorithm is performed using the following steps...

Step 1 - Select the first element of the list (i.e., Element at first position in the list).
Step 2: Compare the selected element with all the other elements in the list.
Step 3: In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
Step 4: Repeat the same procedure with element in the next position in the list till the entire list is sorted.

***Implementation*:**

```
#include<stdio.h>
void main( )
{
int a[100], n, i, j, temp, min;
```

```
printf("Enter Number of elements in array: \n");
scanf("%d", &n);
printf("Enter Array Elements:\n");
for(i=0;i<n;i++)
scanf("%d", &a[i]);
for(i=0;i<n-1;i++)
{
min=i;
for(j=i+1;j<n;j++)
{
        if(a[min]>a[j])
        {
                min=j;
        }
}
if(min!=i)
{
temp=a[i];
a[i]=a[min];
a[min]=temp;
}
}
printf("Sorted order of given elements is:\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
printf("\n");
}
}
```

***Complexity of the Selection Sort Algorithm:***

To sort an unsorted list with 'n' number of elements, we need to make ((n-1)+(n-2)+(n-3)+.....+1) = (n (n-1))/2 number of comparisons in the worst case. If the list is already sorted then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$
Best Case : $\Omega(n^2)$
Average Case : $\Theta(n^2)$

**2. Bubble Sort:**

In bubble sort each element is compared with its adjacent element. If first element is larger than second one, then both elements are swapped. Otherwise, element are not swapped. Consider the following list of numbers.

***Algorithm:***

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for
```

```
  return list

end BubbleSort
```

***Implementation:***

```
#include<stdio.h>
int main()
{
int i, j, n, bubble[20], temp;
printf("enter range of elements");
scanf("%d",&n);
printf("enter elements");
for(i=0;i<n; i++)
{
scanf("%d",&bubble[i]);
}
for(i=0; i<n; i++)
{
for(j=0;j<n-i-1;j++)
{
if(bubble[j] > bubble[j+1])
{
temp=bubble[j];
bubble[j]=bubble[j+1];
bubble[j+1]=temp;
}
}
}
printf("after sorting");
for(i=0; i<n; i++)
{
printf("%d ",bubble[i]);
}
}
```

***Complexity of the Bubble Sort Algorithm:***

Worst Case : $O(n^2)$
Best Case : $O(n)$
Average Case : $O(n^2)$

**3. Insertion Sort:**

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

***Algorithm:***

Step 1 - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
Step 2: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
Step 3: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

***Implementation:***

```
//Insertion Sort
#include<stdio.h>
int main()
{
int i, j, key,n, a[20];
printf("enter range of elements: ");
scanf("%d", &n);
printf("enter elements\n");
for(i=0;i<n; i++)
{
scanf("%d", &a[i]);
}
for(i=1; i<n; i++)
{
key=a[i];
j=i-1;
while(j>=0&&a[j]>key)
{
a[j+1]=a[j];
j=j-1;
}
a[j+1]=key;
}
printf("after sorting ");
for(i=0; i<n; i++)
{
printf("%d ",a[i]);
}
}
```

***Complexity of the Insertion Sort Algorithm:***

To sort an unsorted list with 'n' number of elements, we need to make (1+2+3+.....+n-1) = (n (n-1))/2 number of comparisons in the worst case. If the list is already sorted then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$
Best Case : $\Omega(n)$
Average Case : $\Theta(n^2)$

**4. Quick Sort:**

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by C. A. R. Hoare.

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. Here pivot element is one of the elements in the list**.**

The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

***Algorithm:***

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).

Step 2 - Define two variables i and j. Set i and j to second and last elements of the list respectively.
Step 3 - Increment i until list[i] > pivot then stop.
Step 4 - Decrement j until list[j] < pivot then stop.
Step 5 - If i < j then exchange list[i] and list[j].
Step 6 - Repeat steps 3,4 & 5 until i > j.
Step 7 - Exchange the pivot element with list[j] element.

***Implementation:***

```
//Quick Sort
#include<stdio.h>
void QuickSort(int [],int,int);
int main()
{
int i, n, list[20];
printf("enter range of elements");
scanf("%d",&n);
printf("enter elements\n");
for(i=0;i<n; i++)
{
scanf("%d",&list[i]);
}
QuickSort(list,0,n-1);
printf("after sorting: ");
for(i=0; i<n; i++)
{
printf("%d ",list[i]);
}
}
void QuickSort(int list[],int first,int last)
{
   int pivot,i,j,temp;
    if(first < last)
    {
       pivot = first;
       i = first;
       j = last;

       while(i < j)
       {
          while(list[i] <= list[pivot] && i < last)
             i++;
          while(list[j] > list[pivot])
             j--;
          if(i < j)
          {
              temp = list[i];
              list[i] = list[j];
              list[j] = temp;
          }
       }

       temp = list[pivot];
       list[pivot] = list[j];
       list[j] = temp;
```

```
        QuickSort(list,first,j-1);
        QuickSort(list,j+1,last);
    }
}
```

***Complexity of the Quick Sort Algorithm:***

To sort an unsorted list with 'n' number of elements, we need to make ((n-1)+(n-2)+(n-3)+.....+1) = (n (n-1))/2 number of comparisons in the worst case. If the list is already sorted, then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$
Best Case : O (n log n)
Average Case : O (n log n)

***5. Merge Sort:***

Merge sort is a sorting technique based on **divide and conquer technique**. With worst-case time complexity being O(n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

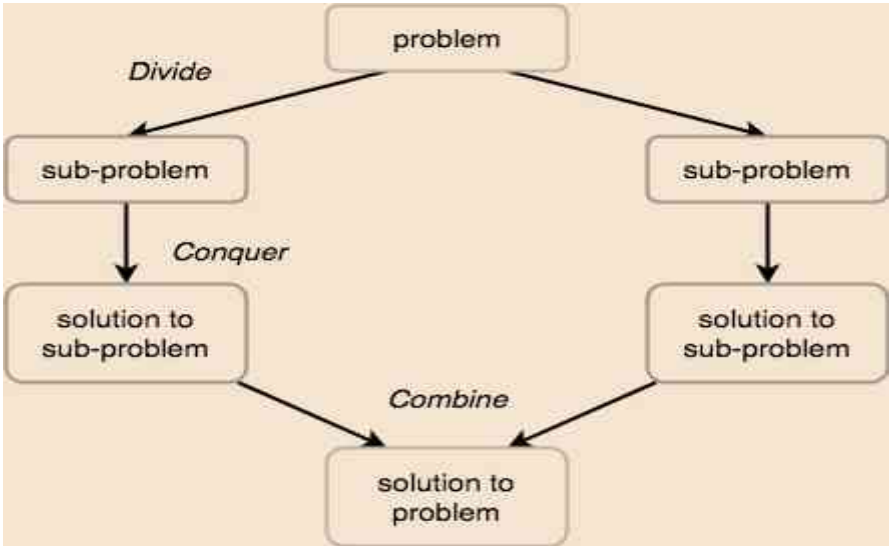
What is the rule of Divide and Conquer?

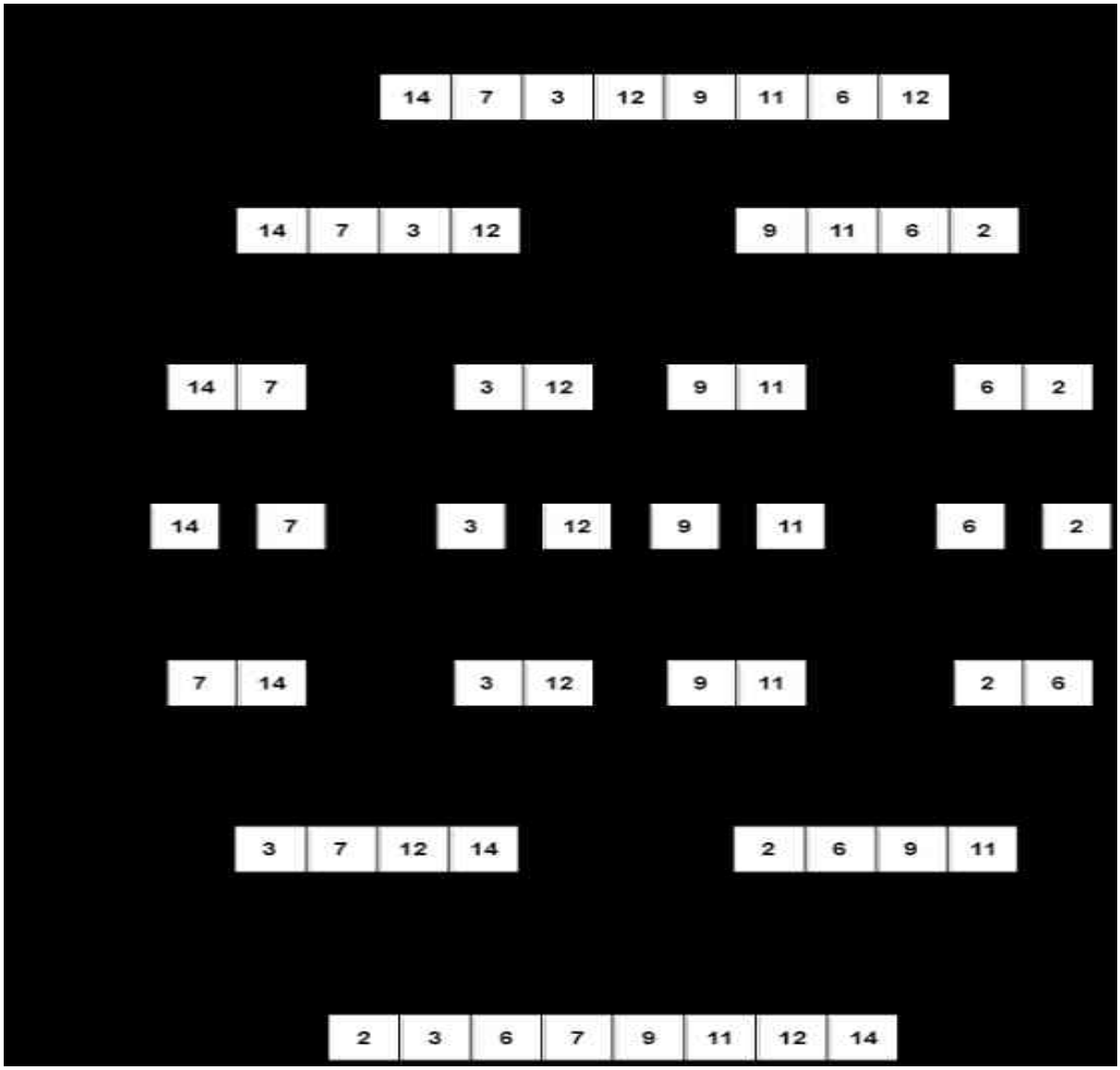
If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. Divide the problem into multiple small problems.
2. Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. Combine the solutions of the subproblems to find the solution of the actual problem.






***Algorithm:***

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

***Implementation:***

```
//Merge Sort
#include<stdio.h>
void mergesort(int a[],int lb,int mid, int ub)
{
int i=lb,j=mid+1,k=0,b[50];
        while(i<=mid&&j<=ub)
        {
                if(a[i]<a[j])
                {
                        b[k++]=a[i++];
                }
                else
                {
```

```
                        b[k++]=a[j++];
                }
        }
        while(i<=mid)
        {
        b[k++]=a[i++];
        }
        while(j<=ub)
        {
        b[k++]=a[j++];
        }
        for(k=0;k<=ub-lb;k++)
        {
        a[k+lb]=b[k];
        }
}
void merge(int a[],int low,int high)
{
int mid;
        if(low<high)
        {
        mid=(low+high)/2;
        merge(a,low,mid);
        merge(a,mid+1,high);
        mergesort(a,low,mid,high);
        }
}
int main()
{
int i, a[30],n;
printf("Enter the number of elements: ");
scanf("%d",&n);
printf("Enter elements:\n");
for(i=0;i<n;i++)
{
        scanf("%d",&a[i]);
}
printf("\n\nBefore sorting:");
for(i=0;i<n;i++)
{
        printf("%d ",a[i]);
}
merge(a,0,n-1);
printf("\n\nAfter sorting:");
for(i=0;i<n;i++)
{
        printf("%d ",a[i]);
}
return 0;
}
```

***Complexity of the Merge Sort Algorithm:***

Worst Case Time Complexity : O(n*log n)
Best Case Time Complexity : O(n*log n)
Average Time Complexity : O(n*log n)

**6. Heap Sort:**

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called Heap Tree. In this sorting algorithm, we use Max Heap to arrange list of elements in Descending order and Min Heap to arrange list of elements in Ascending order.

***Algorithm:***

Step 1 - Construct a Binary Tree with given list of Elements.
Step 2 - Transform the Binary Tree into Min Heap.
Step 3 - Delete the root element from Min Heap using Heapify method.
Step 4 - Put the deleted element into the Sorted list.
Step 5 - Repeat the same until Min Heap becomes empty.
Step 6 - Display the sorted list.

***Implementation:***

For example please go through the heap tree concept...

```
//Heap Sort
#include <stdio.h>
void heapify(int arr[], int n, int i)
{
   int smallest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
     if (l < n && arr[l] < arr[smallest])
       smallest = l;
   if (r < n && arr[r] < arr[smallest])
       smallest = r;
     if (smallest != i) {
       // swap arr[i] with a[smallest]
       int temp = arr[i];
       arr[i] = arr[smallest];
       arr[smallest] = temp;
       heapify(arr, n, smallest);
   }
}
void heapSort(int arr[], int n)
{
 for (int i = n / 2 - 1; i >= 0; i--)
       heapify(arr, n, i);
 for (int i = n - 1; i >= 0; i--) {
       // swap arr[0] with arr[i]
       int temp = arr[0];
       arr[0] = arr[i];
       arr[i] = temp;
        heapify(arr, i, 0);
   }
}

int main()
{
int a[50],n,i;
printf("Enter the size of the array:");
scanf("%d",&n);
```

```
printf("\nEnter the elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\nThe list before sorting:");
for(i=0;i<n;i++)
printf("%d ",a[i]);
heapSort(a,n);
printf("\nThe list after sorting:");
for(i=0;i<n;i++)
printf("%d ",a[i]);
}
```

***Complexity of the Heap Sort Algorithm:***

To sort an unsorted list with 'n' number of elements, following are the complexities...

Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

**7. Shell Sort:**

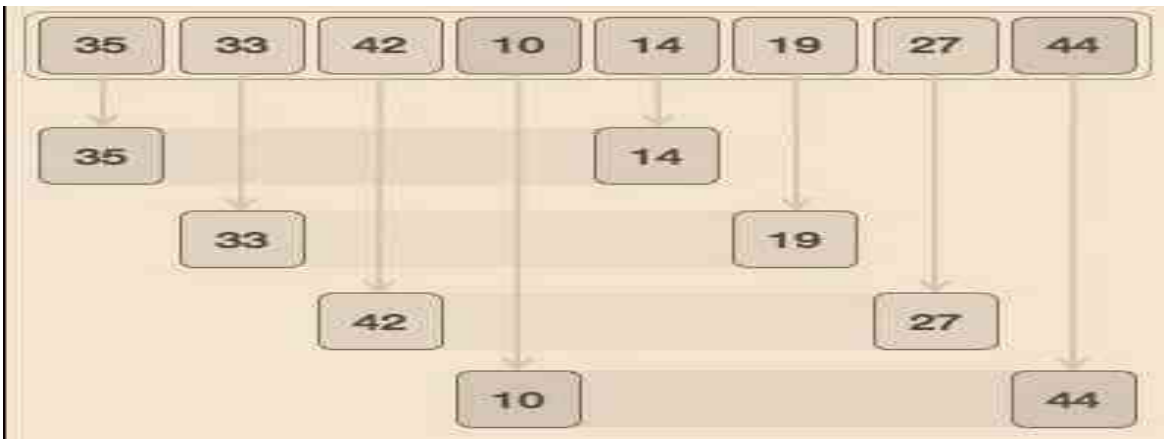
Shell sort is a special case of insertion sort. It was designed to overcome the drawbacks of insertion sort. Thus, it is more efficient than insertion sort. In shell sort, we can swap or exchange the far away items in addition to adjacent items.

In insertion sort, we could move the elements ahead only by one position at a time. If we wish to move an element to a faraway position in insertion sort, a lot of movements were involved thus increasing the execution time. Shell sort overcomes this problem of insertion sort by allowing movement and swap of far-away elements as well.

This sorting technique works by sorting elements in pairs, far away from each other and subsequently reduces their gap. The gap is known as the interval. We can calculate this gap/interval as total no.of elements/2(i.e.,n/2) and then n/4,n/8.....

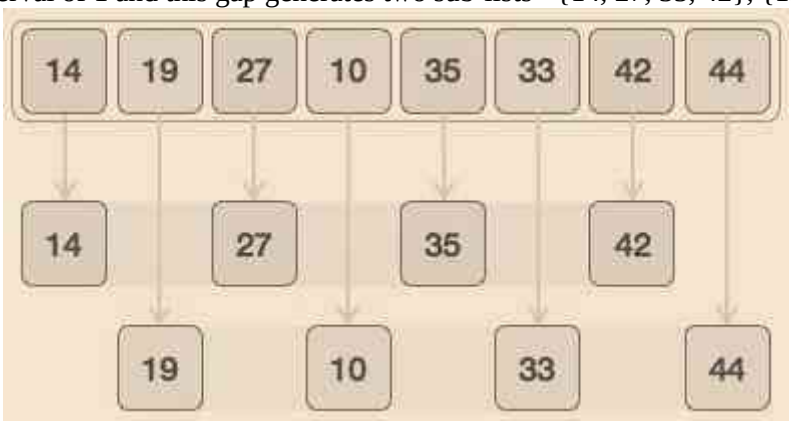
**How Shell Sort Works?**

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −


Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}




We compare and swap the values, if required, in the original array. After this step, the array should look like this −


Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −

**Algorithm:**

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using insertion sort
Step 3 − Repeat until complete list is sorted

**Implementation:**

```
#include <stdio.h>
void shellsort(int arr[], int num)
{
   int i, j, k, tmp;
   for (i = num / 2; i > 0; i = i / 2)
   {
      for (j = i; j < num; j++)
      {
         for(k = j - i; k >= 0; k = k - i)
         {
            if (arr[k+i] >= arr[k])
               break;
            else
            {
               tmp = arr[k];
               arr[k] = arr[k+i];
               arr[k+i] = tmp;
            }
         }
      }
   }
}
int main()
{
   int arr[30];
   int k,  num;
   printf("Enter total no. of elements : ");
   scanf("%d", &num);
   printf("\nEnter %d numbers: ", num);

   for (k =  0 ; k < num; k++)
   {
      scanf("%d", &arr[k]);
   }
   shellsort(arr, num);
   printf("\n Sorted array is: ");
   for (k = 0; k < num; k++)
      printf("%d ", arr[k]);
   return 0;
}
```

**Comparison among all the sorting techniques:**

| | Time | | | | | |
|---|---|---|---|---|---|---|
| Sort | Average | Best | Worst | Space | Stability | Remarks |
| Bubble sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | Constant | Stable | Always use a modified bubble sort |
| Selection Sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | Constant | Stable | Even a perfectly sorted input requires scanning the entire array |
| Insertion Sort | $O(n^2)$ | O(n) | $O(n^2)$ | Constant | Stable | In the best case (already sorted), every insert requires constant time |
| Heap Sort | O(n*log(n)) | O(n*log(n)) | O(n*log(n)) | Constant | Instable | By using input array as storage for the heap, it is possible to achieve constant space |
| Merge Sort | O(n*log(n)) | O(n*log(n)) | O(n*log(n)) | Depends | Stable | On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space |
| Quicksort | O(n*log(n)) | O(n*log(n)) | O(n^2) | Constant | Stable | Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array. |

## Searching:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.
Gathering any information (or) trying to find any data is said to be a Searching process.
Searching technique can be used more efficiently if the data is present in an ordered manner.
Most widely used Searching methods are:-
1) Linear Search (Sequential Search) 2) Binary Search

**1. Linear Search:**

Suppose an array is given, which contains “n” elements. If no other information is given and we are asked to search for an element in array, than we should compare that element, with all the elements present in the array. This method which is used to Search the element in the array is known as Liner Search. Since the key element/ the element which is to be searched in array, if found out by comparing with every element of array one-by-one, this method is also known as Sequential Search.

***Algorithm:***

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the first element in the list.
Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function
Step 4 - If both are not matched, then compare search element with the next element in the list.

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.
Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

***Implementation:***

```
#include<stdio.h>
void main()
{
int a[100], n, i, key,temp;
printf("Enter Number of elements in array: \n");
scanf("%d", &n);
printf("Enter Array Elements:\n");
for(i=0; i<n; i++)
scanf("%d", &a[i]);
printf("Enter Key value to search\n");
scanf("%d", &key);
for(i=0;i<n;i++)
if(a[i]==key)
{
temp=1;
printf("Number found at position %d\n",i+1);
break;
}
if(temp!=1)
{
printf("Number not found in given elements\n");
}
}
```

- The time complexity of Linear search is:

a. Best case = O(1)
b. Average case = n(n+1)/2n = O(n)
c. Worst case = O(n)

- The space complexity of Linear Search is O(1).

**2. Binary Search:**

Efficient search method for large arrays. Liner search, will required to compare the key element, with every element in array. If the array size is large, liner search requires more time for execution.
In such cases, binary search technique can be used.

To perform binary search:-
i) Elements should be entered into array in Ascending Order
ii) Middle element of the array must be found. This is done as follows

Find the lowest position & highest position of the array i.e., if an array contains „n" elements then:-
Low=0
High =n-1
Mid =(low+high)/2

Note: We are calculating mid position of the array not the middle element. The element present in the mid position is considered as middle element of array.

Now the search key element is compared with middle element of array. Three cases arises
Case 1 : If middle element is equal to key, then search is end.
Case 2 : If middle element is greater than key, then search is done on left sublist of the middle element

of array.
Case 3 : If middle element is less than key, then search is done on right sublist of the middle element of array.

This process is repeated till we get the key element (or) till the search comes to an end, since key element is not in the list.

***Algorithm:***

Step 1 - Read the search element from the user.
Step 2 - Find the middle element in the sorted list.
Step 3 - Compare the search element with the middle element in the sorted list.
Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

***Implementation:***

```
#include<stdio.h>
void main()
{
int a[100], n, i, high, low, mid, key, location;
printf("Enter Number of elements in array: \n");
scanf("%d", &n);
printf("Enter Array Elements in sorted order:\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("Enter Key value to search\n");
scanf("%d",&key);
low=0;
high=n-1;
mid=(low+high)/2;
if(a[mid]==key)
{
location=mid;
}
while((low<=high)&&(a[mid]!=key))
{
mid=(low+high)/2;
if(a[mid]<key)
low=mid+1;
else if(a[mid]>key)
high=mid-1;
else
location=mid;
}
if(a[location]==key)
```

```
printf("Number found at position %d\n",location+1);
else
printf ("Number not found in given elements\n");
}
```

- The time complexity of Binary search is:

a. Best case = O(1)
b. Average case = O(logn)
c. Worst case = O(logn)

- The space complexity of Binary Search is O(1).

# Graphs:

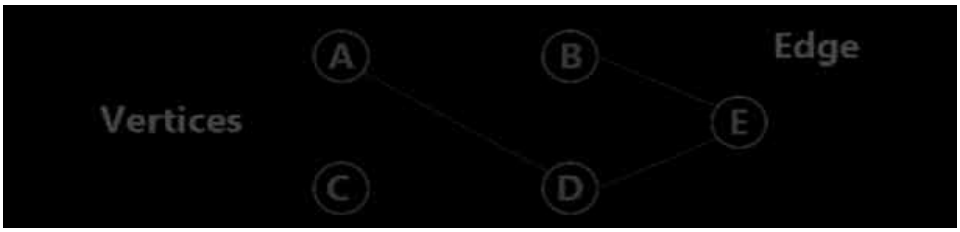
Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

"Graph is a collection of vertices and arcs in which vertices are connected with arcs" or "Graph is a collection of nodes and edges in which nodes are connected with edges".

Generally, a graph G is represented as G = ( V , E ), where V is set of vertices and E is set of edges.

**Example:**

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = ( V , E )
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.




**Basic Graph Terminologies:**

1. ***Vertex:***
   Individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.
2. ***Edge:***
   An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

   ➢ Edges are three types.

   - **Undirected Edge** - An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
   - **Directed Edge** - A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
   - **Weighted Edge** - A weighted egde is a edge with value (cost) on it.

3. ***Undirected Graph***: A graph with only undirected edges is said to be undirected graph.
4. ***Directed Graph:*** A graph with only directed edges is said to be directed graph.
5. ***Mixed Graph:*** A graph with both undirected and directed edges is said to be mixed graph.
6. ***End vertices or Endpoints:*** The two vertices joined by edge are called end vertices (or endpoints) of that edge.
7. ***Origin:*** If a edge is directed, its first endpoint is said to be the origin of it.
8. ***Destination:*** If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.
9. ***Adjacent:*** If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.
10. ***Incident:*** Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.
11. ***Outgoing Edge:*** A directed edge is said to be outgoing edge on its origin vertex.
12. ***Incoming Edge:*** A directed edge is said to be incoming edge on its destination vertex.
13. ***Degree:*** Total number of edges connected to a vertex is said to be degree of that vertex.
14. ***Indegree:*** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
15. ***Outdegree:*** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

16. ***Parallel edges or Multiple edges:*** If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

17. ***Self-loop:*** Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.
18. ***Simple Graph:*** A graph is said to be simple if there are no parallel and self-loop edges.
19. ***Path:*** A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

## Graph Representations:

Graph data structure is represented using following representations...

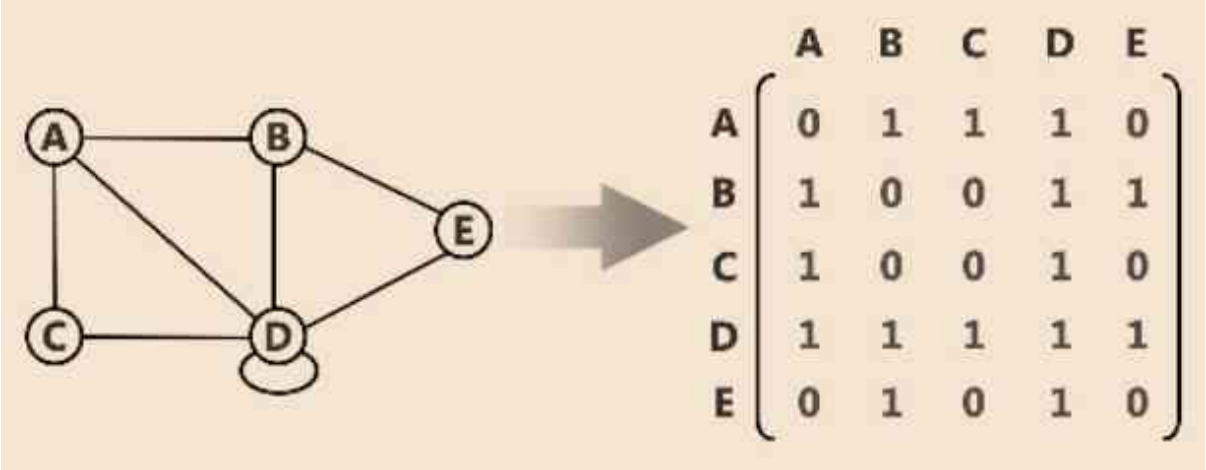
1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

### 1. *Adjacency Matrix:*

In this representation, the graph is represented using a matrix of size i.e., total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled

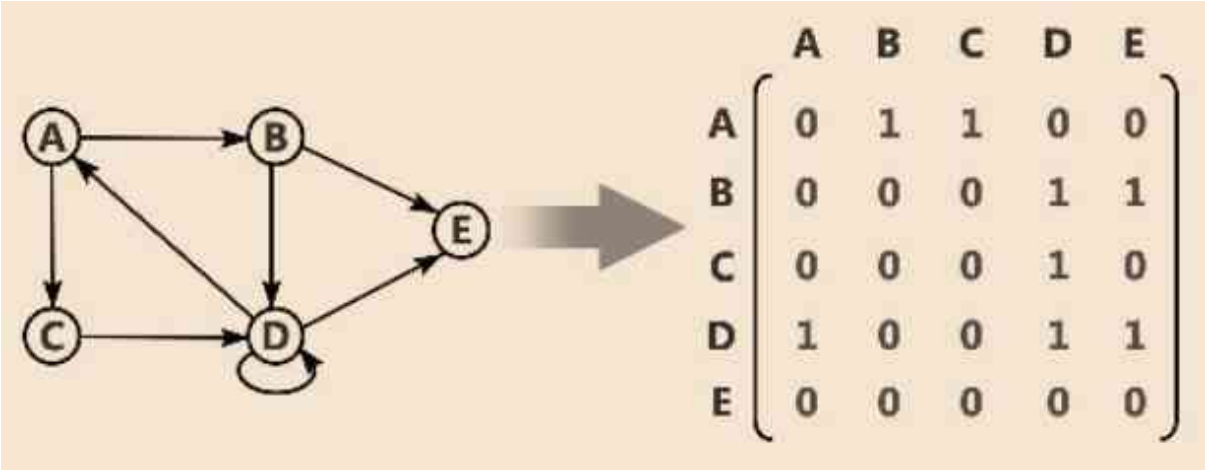
with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...




| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 0 |
| B | 1 | 0 | 0 | 1 | 1 |
| C | 1 | 0 | 0 | 1 | 0 |
| D | 1 | 1 | 1 | 1 | 1 |
| E | 0 | 1 | 0 | 1 | 0 |

Directed graph representation...

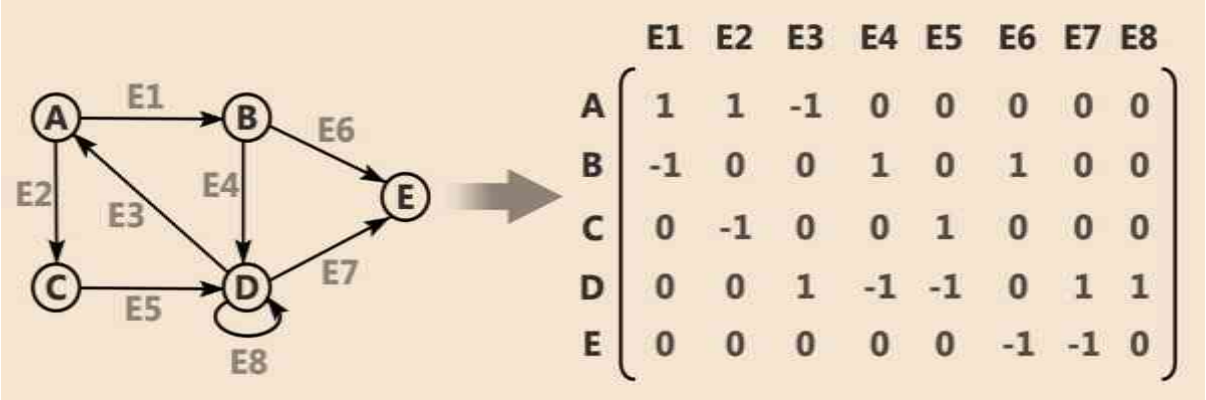



| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 |
| B | 0 | 0 | 0 | 1 | 1 |
| C | 0 | 0 | 0 | 1 | 0 |
| D | 1 | 0 | 0 | 1 | 1 |
| E | 0 | 0 | 0 | 0 | 0 |

***2. Incidence Matrix***

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represent edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

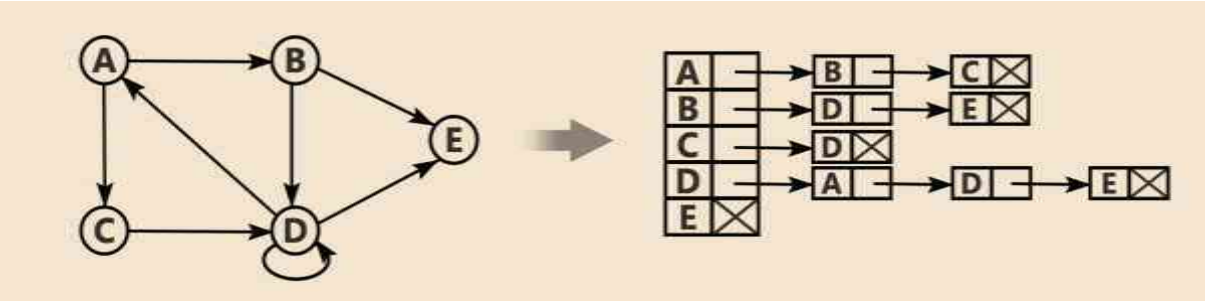



| | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 |
|---|---|---|---|---|---|---|---|---|
| A | 1 | 1 | -1 | 0 | 0 | 0 | 0 | 0 |
| B | -1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| C | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | -1 | -1 | 0 | 1 | 1 |
| E | 0 | 0 | 0 | 0 | 0 | -1 | -1 | 0 |

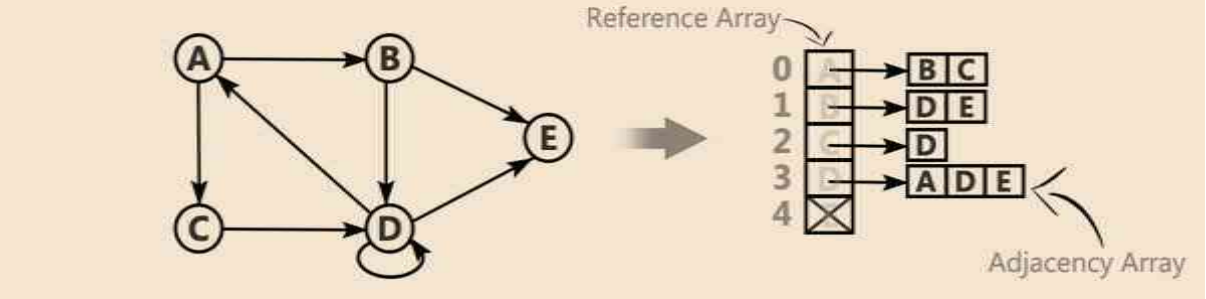
### 3. *Adjacency List:*

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...




This representation can also be implemented using an array as follows..




**Graph Search and traversal algorithms:**

Graph traversal is a technique used for a searching a vertex in graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

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

### *1. DFS (Depth First Search):*

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

***Algorithm:***

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

"Back tracking is coming back to the vertex from which we reached the current vertex".

***Implementation:***

```
#include<stdio.h>
  int cost[10][10],i,j,k,n,m,stack[10],top=-1,v;
  int visited[10];
void DFS()
{
   printf("enter the initial vertex");
   scanf("%d",&v);
   printf("visited vertices\n");
   stack[++top]=v;
   while(top!=-1)
     {
       v=stack[top--];
       if(visited[v]==0)
        {
          printf("%d",v);
          visited[v]=1;
        }
       else
         continue;
         for(j=n;j>=1;j--)
         if(!visited[j]&& cost[v][j]==1)
              stack[++top]=j;
    }
}
int main()
{
printf("enter the no of vertices");
    scanf("%d",&n);
     printf("enter the no of edges");
```

```
    scanf("%d",&m);
for ( int i=1;i<=10;i++)
        {
                visited[i]=0;
        }
       for (int i=0;i<10;i++)
        for(j=0;j<10;j++)
          cost[i][j]=0;
    printf("\nEnter EDGES");
    for(k=1;k<=m;k++)
      {
       scanf("%d%d",&i,&j);
       cost[i][j]=1;
       cost[j][i]=1;
      }
DFS();
}
```

***Complexity:***

Time complexity O(V+E), when implemented using an adjacency list.

2***. BFS (Breadth First Search):***

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

***Algorithm:***

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

***Implementation:***

```
#include<stdio.h>
  int cost[10][10],i,j,k,n,m,queue[10],front,rear,v;
  int visited[10];
void BFS()
{
   printf("enter the initial vertex");
   scanf("%d",&v);
   printf("visited vertices\n");

   visited[v]=1;
   queue[++rear]=v;
   while(front!=rear)
    {
      v=queue[++front];
      printf("%d",v);
```

```
        for(j=1;j<=n;j++)
          if(!visited[j]&& cost[v][j]==1)
            {
               queue[++rear]=j;
               visited[j]=1;
             }
      }
}

int main()
{
  printf("enter the no of vertices");
    scanf("%d",&n);
     printf("enter the no of edges");
     scanf("%d",&m);
           front=-1;rear=-1;
          for ( int i=1;i<=10;i++) visited[i]=0;
          for (int i=0;i<10;i++)
            for(j=0;j<10;j++)
               cost[i][j]=0;

     printf("\nEnter EDGES");
     for(k=1;k<=m;k++)
        {
          scanf("%d%d",&i,&j);
          cost[i][j]=1;
          cost[j][i]=1;
        }
BFS();
}
```

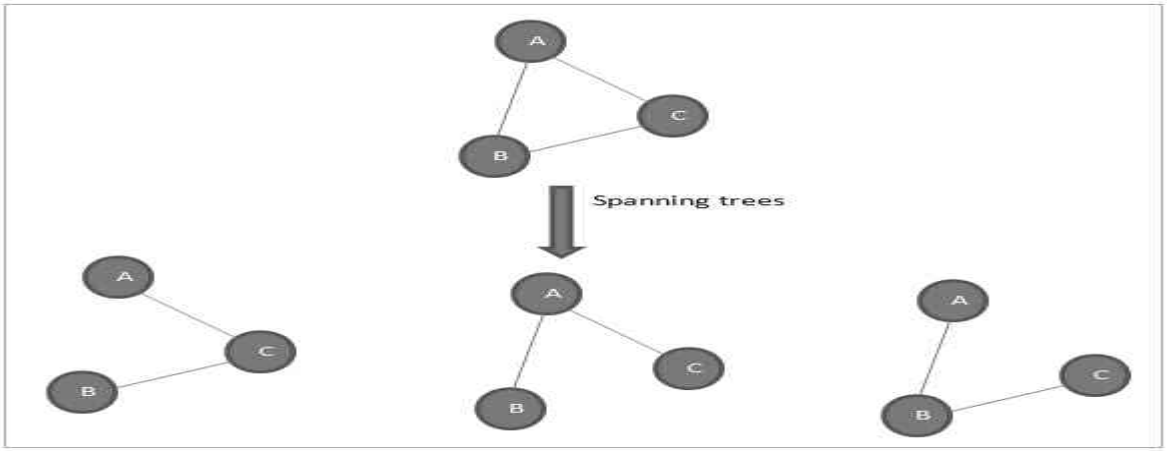
***Complexity:***

The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges.

## Minimum Cost Spanning Trees:

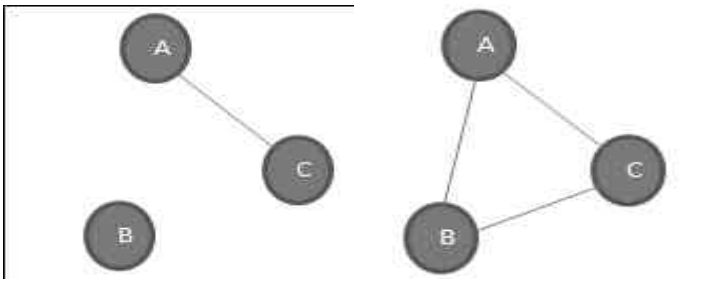
What is a Spanning Tree?

A Spanning tree can be defined as a subset of a graph, which consists of all the vertices covering minimum possible edges and does not have a cycle. Spanning tree can be connected graph.



**Some of the properties of the spanning tree :**

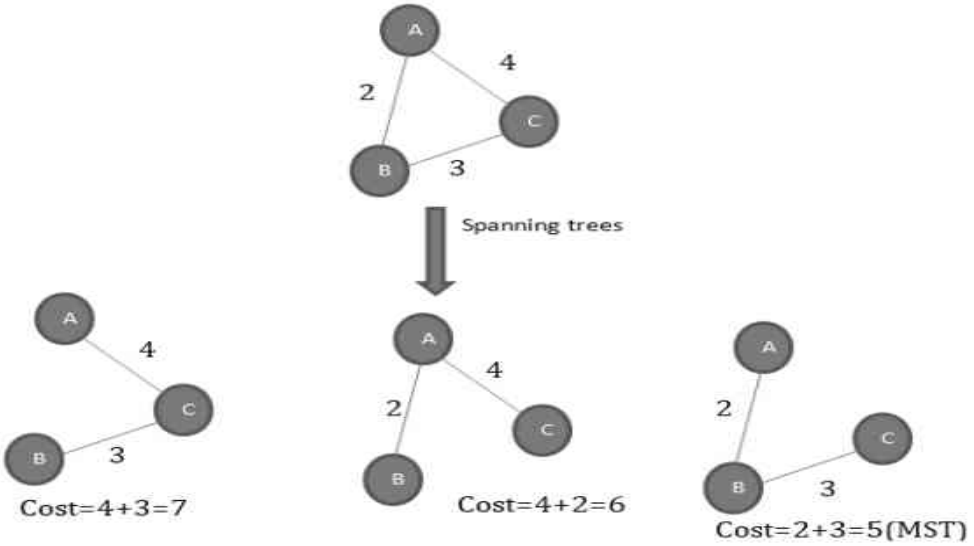
- A connected graph can have more than one spanning trees.
- All spanning trees in a graph have the same number of nodes and edges.
- If we remove one edge from the spanning tree, then it will become the disconnected graph.
- If we adding one edge to the spanning tree will create a loop.
- A spanning tree does not have a loop or a cycle.




Disconnected graph Connected graph

What is a Minimum Cost Spanning Tree (MST)?

- A minimum spanning tree is the one that contains the least weight among all the other spanning trees of a connected weighted graph.
- There can be more than one minimum spanning tree for a graph.




There are two most popular algorithms that are used to find the minimum spanning tree in a graph.

They are:

1. Prim's algorithm
2. Kruskal's algorithm

1. **Prim's algorithm:**

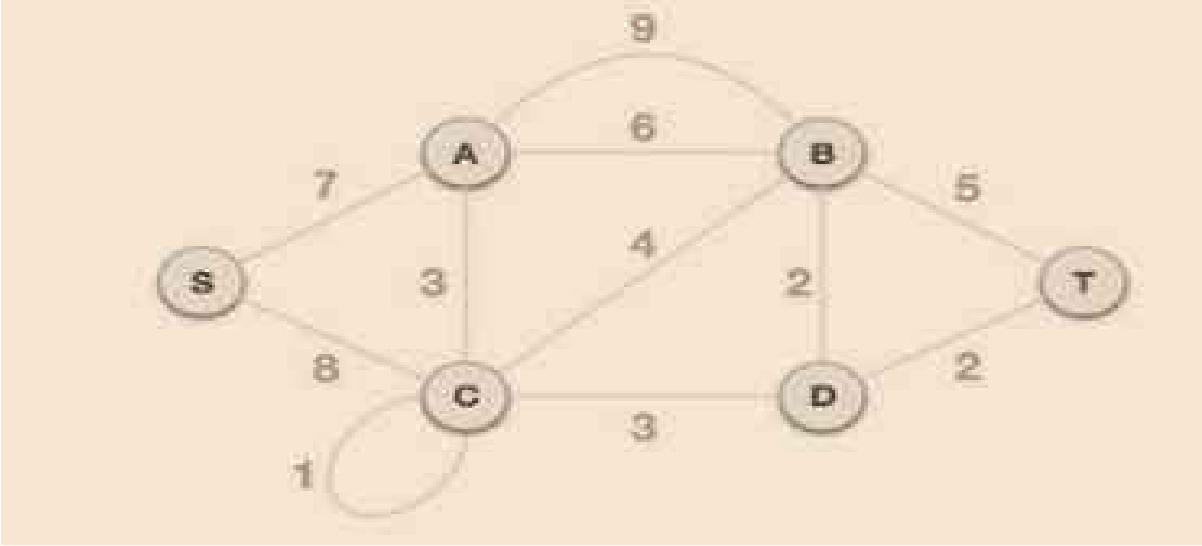
- Prim's algorithm is an algorithm to find the MST in a connected graph.
- The graph must be a simple graph.
- Prim's algorithm starts with a vertex.
- We start with one vertex and keep on adding edges with the least weight till all the vertices are covered.

**The sequence of steps for Prim's Algorithm is as follows:**

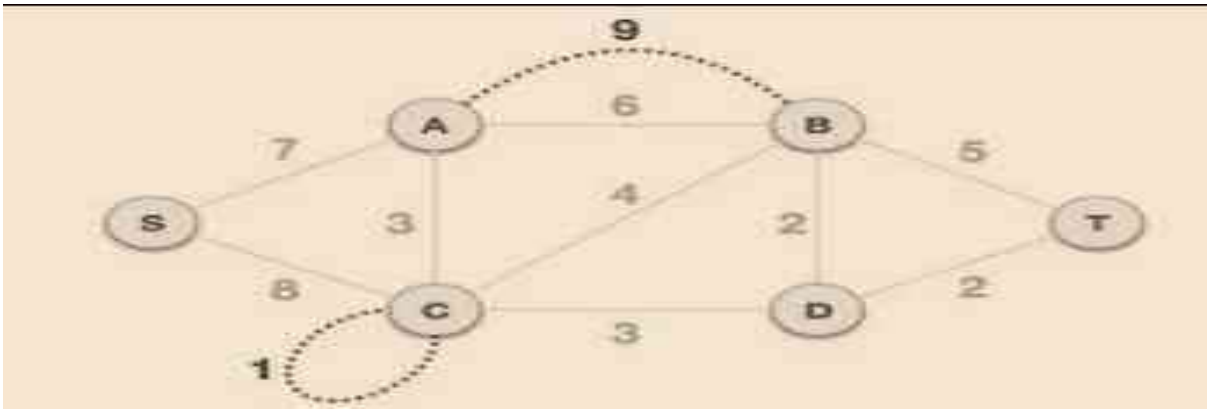
- Convert the graph to simple graph.

- Choose a random vertex as starting vertex and initialize a minimum spanning tree.
- Find the edges that connect to other vertices. Find the edge with minimum weight and add it to the spanning tree.
- Repeat step 2 until the spanning tree is obtained.

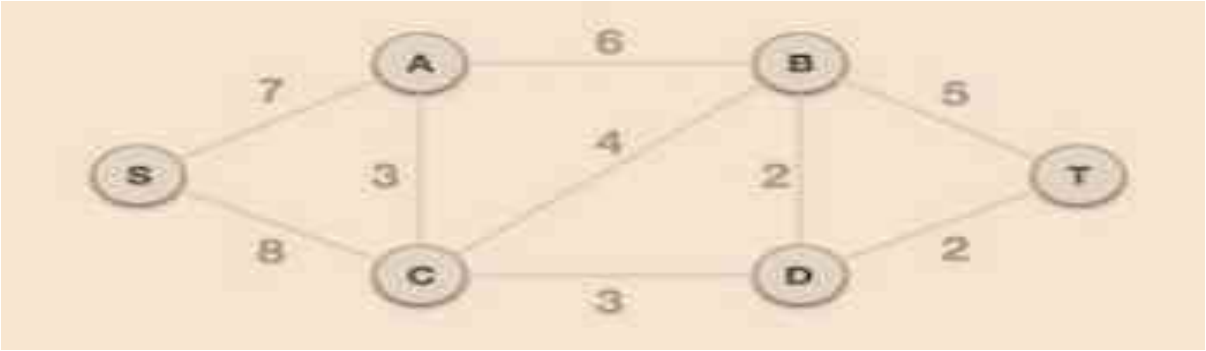
Consider the following example graph to illustrate of Prim's algorithm:



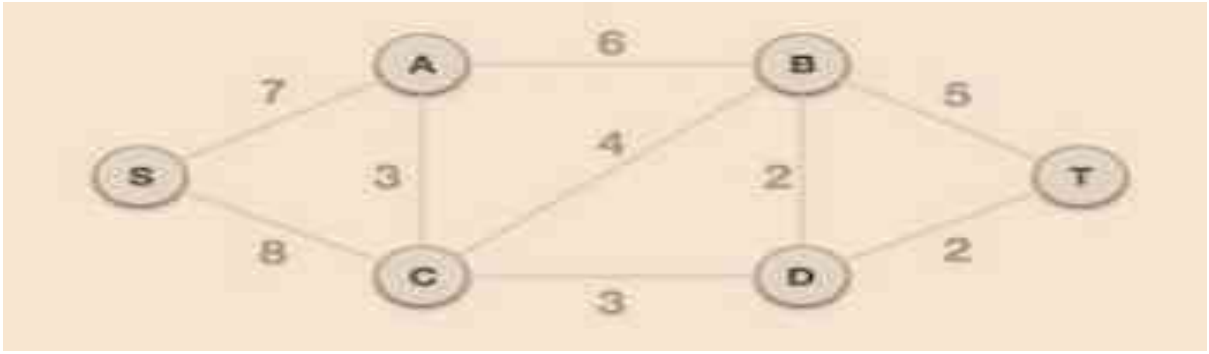

Step 1 - Remove all loops and parallel edges




- In case of parallel edges, keep the one which has the least cost associated and remove all others.



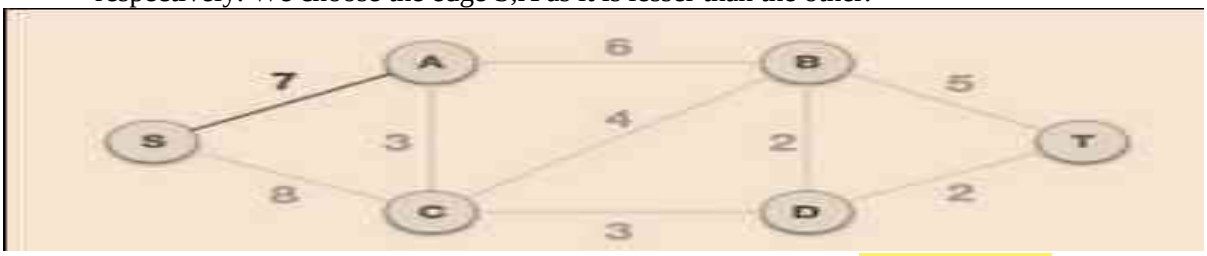

Step-2:Choose a random vertex as starting vertex and initialize a minimum spanning tree.



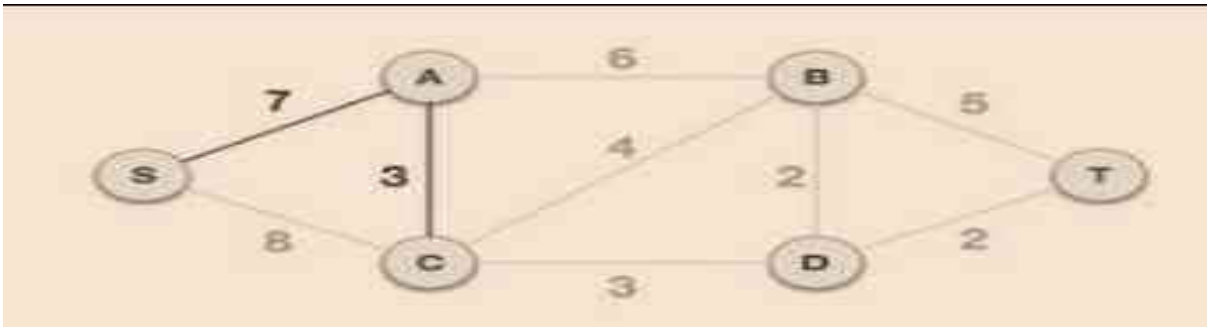

- we choose S node as the root node of Prim's spanning tree. This node is chosen, so any node can be the root node.

Step-3: Check outgoing edges and select the one with less cost

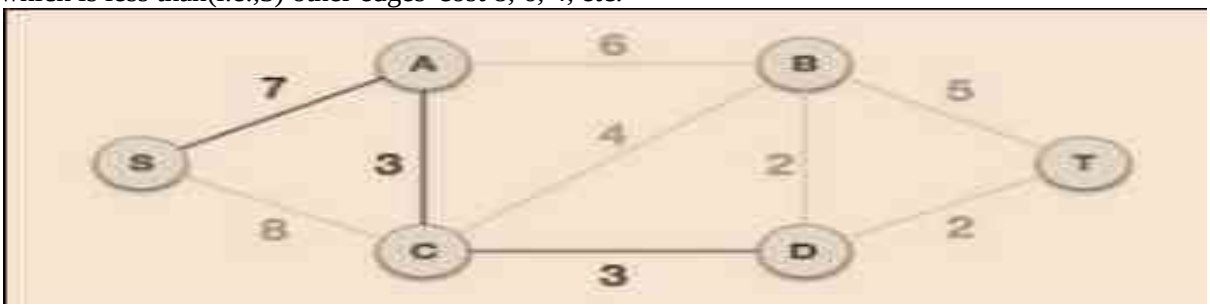
- After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



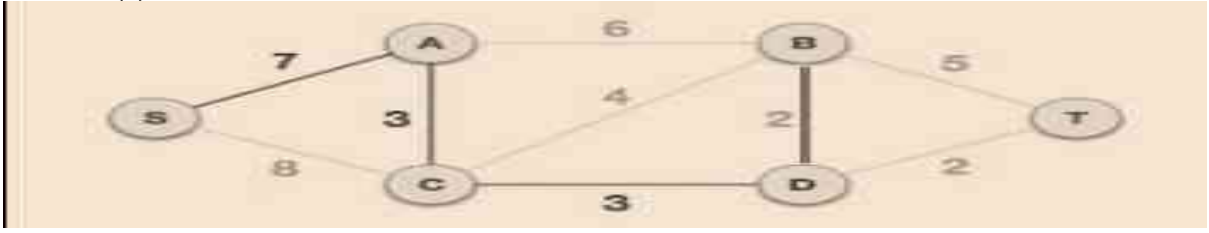

Step-4: Now, the tree S-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



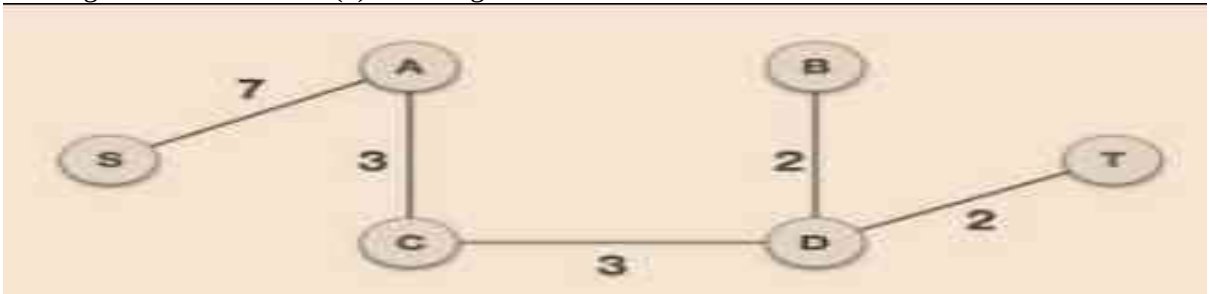

Step-5: After this step, S-A-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-D is the new edge, which is less than(i.e.,3) other edges' cost 8, 6, 4, etc.




Step-6: After adding node D to the spanning tree, we now have two edges going out of it having the same cost(2), i.e. D-T and D-B. Thus, we can add either one.



step-7: After adding node B to the spanning tree, the node will be S-A-C-D-B and we check for all edges going out from it. However, we will choose only the least cost edge. In this case, D-T is the new edge, which is less than(2) other edges' cost 8, 6, 5, 4, etc.




Implementation of Prim's algorithm:

```
#include<stdio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},min,mincost=0,cost[10][10];
void main()
{
        printf("\nEnter the number of nodes:");
        scanf("%d",&n);
        printf("\nEnter the adjacency matrix:\n");
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
                scanf("%d",&cost[i][j]);
                if(cost[i][j]==0)
                        cost[i][j]=999;
        }
        visited[1]=1;
        printf("\n");
        while(ne < n)

        {
                for(i=1,min=999;i<=n;i++)

                for(j=1;j<=n;j++)

                if(cost[i][j]< min)
                        if(visited[i]!=0)
                        {
                                min=cost[i][j];
                                b=v=j;
                        }
                        if(visited[u]==0 || visited[v]==0)
                        {
                                printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
                                mincost+=min;
                                visited[b]=1;
                        }
                cost[a][b]=cost[b][a]=999;
        }
        printf("\n Minimun cost=%d",mincost);
}
```

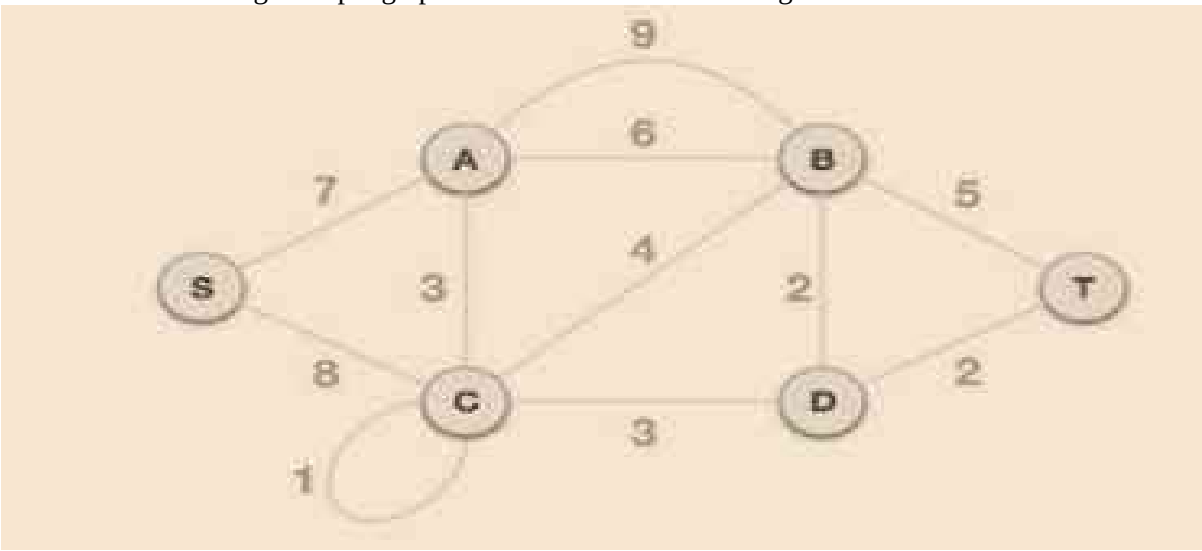
**2. Kruskal's algorithm:**

- Kruskal's algorithm is an algorithm to find the MST in a connected graph.
- The graph must be a simple graph.

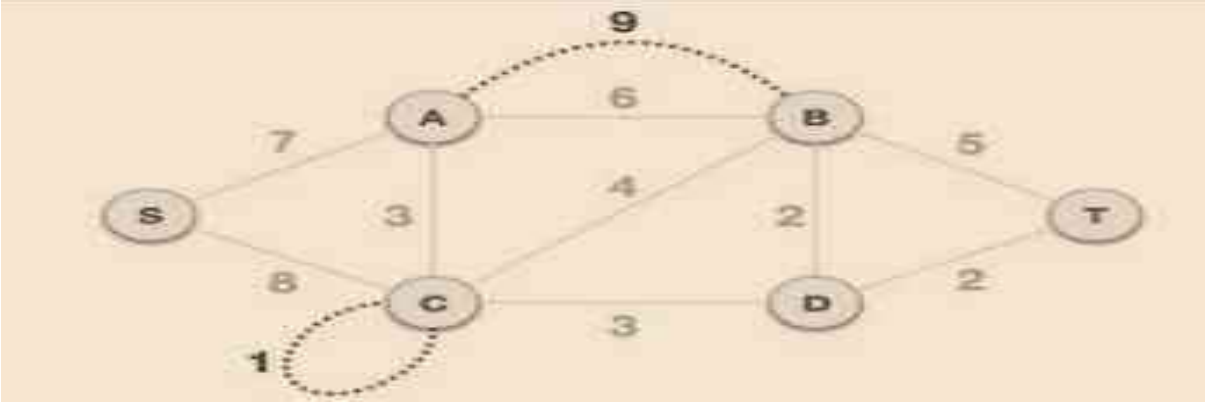
. **The sequence of steps for Kruskal's Algorithm is as follows:**

- Convert the graph to simple graph.
- sort all the edges from the lowest weight to highest.
- Take edge with the lowest weight and add it to the spanning tree. If the cycle is created, discard the edge.
- Keep adding edges like in step 2 until all the vertices are considered.

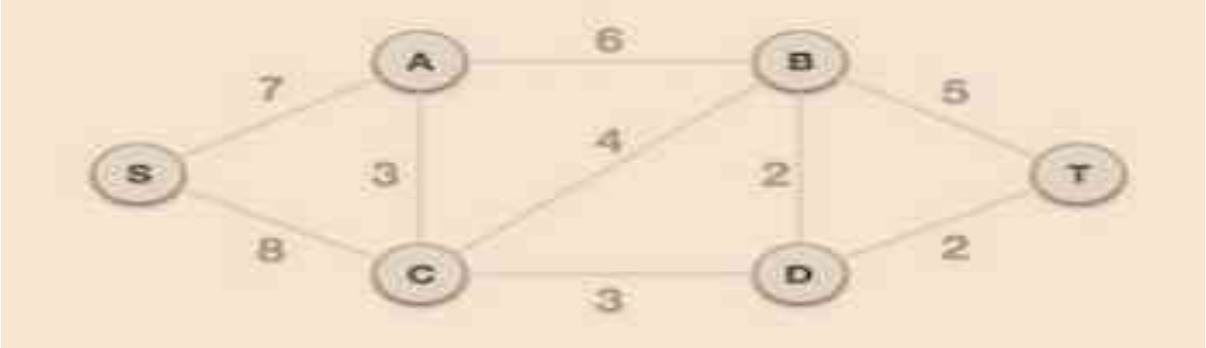
Consider the following example graph to illustrate of Kruskal's algorithm:




Step 1 - Remove all loops and parallel edges




- In case of parallel edges, keep the one which has the least cost associated and remove all others.



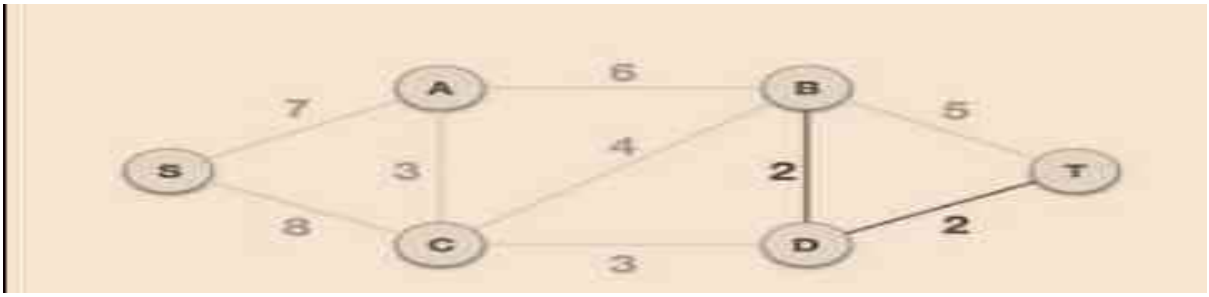
Step 2 - Arrange all edges in their increasing order of weight.

In this step, we will create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

| B, D | D, T | A, C | C, D | C, B | B, T | A, B | S, A | S, C |
|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 |

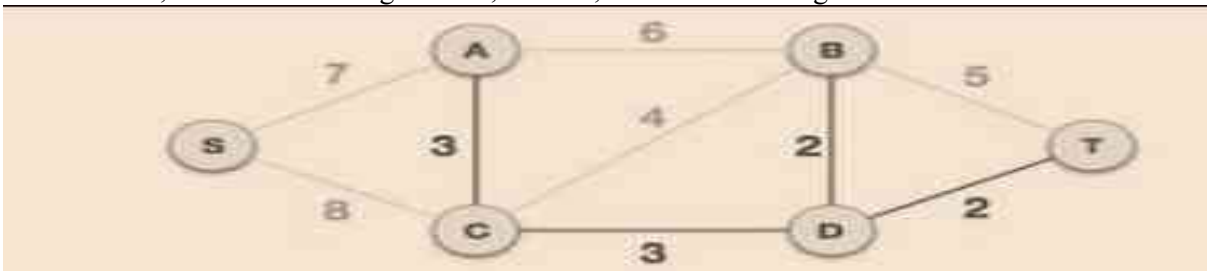
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

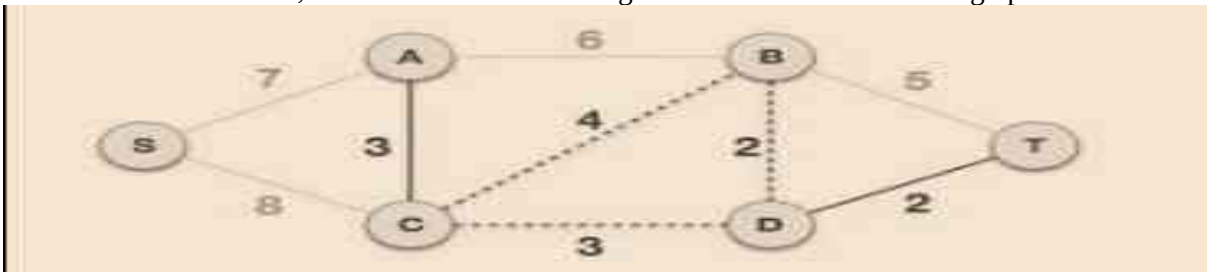



The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

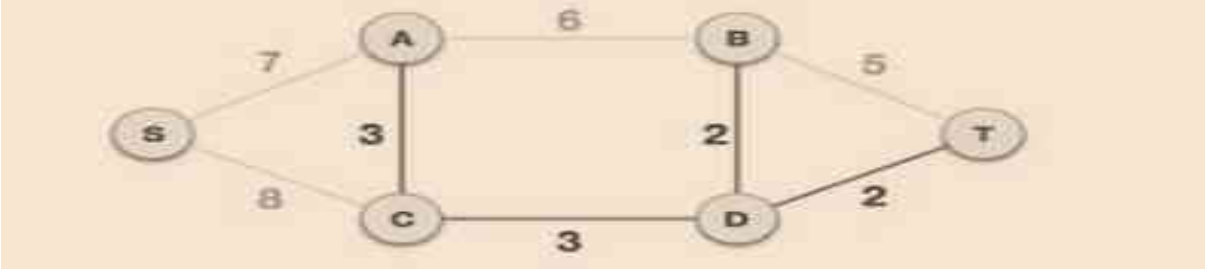
Next cost is 3, and associated edges are A,C and C,D. We add them again −



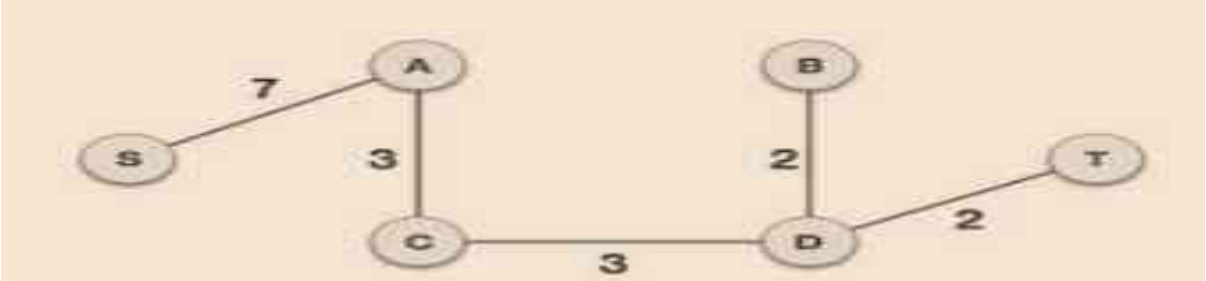

Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.




We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



- We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.
- Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.




- By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Implementation of Kruskal's algorithm:

```
#include<stdio.h>
#include<stdlib.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
        printf("\n\tImplementation of Kruskal's algorithm\n");
        printf("\nEnter the no. of vertices:");
        scanf("%d",&n);
        printf("\nEnter the cost adjacency matrix:\n");
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=n;j++)
                {
                        scanf("%d",&cost[i][j]);
                        if(cost[i][j]==0)
                                cost[i][j]=999;
                }
        }
        printf("The edges of Minimum Cost Spanning Tree are\n");
        while(ne < n)
        {
                for(i=1,min=999;i<=n;i++)
                {
                        for(j=1;j <= n;j++)
                        {
                                if(cost[i][j] < min)
                                {
                                        min=cost[i][j];
                                        a=u=i;
                                        b=v=j;
                                }
                        }
                }
                u=find(u);
                v=find(v);
```

```
                if(uni(u,v))
                {
                        printf("%d edge (%d,%d) =%d\n",ne++,a,b,min);
                        mincost +=min;
                }
                cost[a][b]=cost[b][a]=999;
        }
        printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i)
{
        while(parent[i])
        i=parent[i];
        return i;
}
int uni(int i,int j)
{
        if(i!=j)
        {
                parent[j]=i;
                return 1;
        }
        return 0;
}
```