DATA Structures

Introduction :-

- Data may be organised in many different ways.
- A data structure is a particular way of organising data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexity of different task.

Ex- A list of items having same Data type can be stored using Array.

Classification of DS :-

Primitive Data Structure

→ int
→ float
→ char

Non-primitive Data Structure

- **Linear DS**
  - In Linear DS, the data items are arranged in a linear sequence
  - → Arrays, Linked list
  - → Stack, queue
- **Non-linear DS**
  - → Tree
  - → Graph
  - In Non-linear DS, data items are not arranged in sequence

DS Operations :-

(1) Create :- To create a DS
(2) Inserting :- Adding new records (details) to the structure.
(3) Deleting :- removing a record from the structure.
(4) Traversing :- Accessing each element for processing.
(5) Searching :- searching the record. (a particular element)
(6) Sorting :- Arranging records in some logical order.
(7) Merging :- (gets combined in a sorted manner)

Modifying
Copying
Concatinating
Splitting

 # Static DS

→ In Static DS, the size and structures associated memory locations are fixed at compile value.

Ex- Array

 # Dynamic DS

→ In Dynamic DS, it can expand or shrink as required during program execution and there associated memory locations can be chnged.

Ex- Linked List created using pointers

Scanned with CamScanner

DATA Structures

Introduction :-

- Data may be organised in many different ways.
- A data structure is a particular way of organising data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexity of different task.

Ex- A list of items having same Data type can be stored using Array.

Classification of DS :-

Primitive Data Structure

→ int
→ float
→ char

Non-primitive Data Structure

- **Linear DS**
  - → Arrays, Linked list
  - → Stack, queue

  In Linear DS, the data items are arranged in a linear sequence

- **Non-linear DS**
  - → Tree
  - → Graph

  In Non-linear DS, data items are not arranged in sequence

DS Operations :-

(1) Create :- To create a DS
(2) Inserting :- Adding new records (details) to the structure.
(3) Deleting :- removing a record from the structure.
(4) Traversing :- Accessing each element for processing.
(5) Searching :- searching the record. (a particular element)
(6) Sorting :- Arranging records in some logical order.
(7) Merging :- (gets combined in a sorted manner)

Modifying
Copying
Concatinating
Splitting

 # Static DS

→ In Static DS, the size and structures associated memory locations are fixed at compile value.

Ex- Array

 # Dynamic DS

→ In Dynamic DS, it can expand or shrink as required during program execution and there associated memory locations can be chnged.

Ex- Linked List created using pointers

Abstract Data Types (ADT) :-

→ An ADT is a mathematical model for data types where data is defined by its behaviour (possible values) and a set of possible operations on them.

→ An ADT refers to a set of data values and an associated operations on them.

Ex- int data type in C can be considered as an ADT which defines a set of numbers given by $\{-\infty, \ldots\ldots\ldots, 0, 1, \ldots\ldots\ldots \infty\}$ and also specifies the operations that can be performed on them ie $+, -, *, \%, /$.

Algorithms :-

→ An algorithm is a finite sequence of well defined instructions for solving a particular problem.

```
int add (int n1, int n2)
{ int s;
  s = n1 + n2;
  return s;
}
```

Algorithm analysis :-

→ Designing an efficient algorithm involves analysis of time and space complexities.

→ Time complexity :- amount of computer time the algorithm need to run (to) the complete the task.

→ Space complexity :- amount of memory the program needs to run.

Time complexity Measures (types) :-

(a) Best case :- Best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size n.

(b) Worst case :- WC time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size n.

(c) Average case :- AC time complexity is the time that a algorithm will require to execute a typical input data of size n i.e the value obtained by averaging the running time of an algorithm for all possible

inputs of size n.

Asymptotic Notations :-

→ Are the languages that allows to analyse algorithm's running time by identifying its behaviour as the input size increases.

→ This is also known as Algorithm Growth Rate.

1. Big Oh (O) Notation
2. Big Omega (Ω) "
3. Big Theta (Θ) " (Tight Bound)

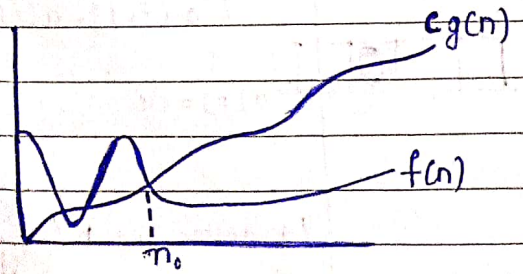
1. Big Oh (O) Notation :- Upper bound

→ defines an upper bound of an algorithm i.e a certain function will never exceed a specified time for any value of input n.

→ It represents the worst case of an algorithm.

For a non-negative function f(n), we can say that

$$O(g(n)) = \{ f(n) : \exists \text{ the constant } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le c\,g(n) \ \forall\, n \ge n_0 \}$$



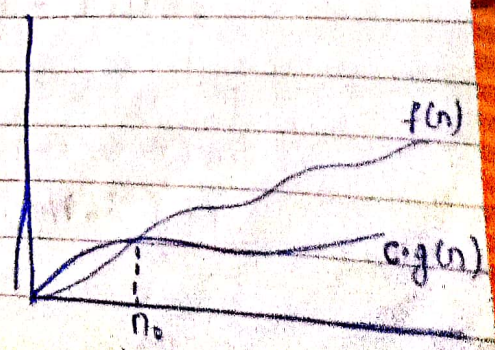
2. Big Omega (Ω) Notation :-

→ defines the lower bound of an algorithm i.e the minimum time required for any algorithm for all input values.

→ represents Best Case of an algorithm.

$$\boxed{0 \le c\,g(n) \le f(n)} \ \forall\, n \ge n_0$$

$$f(n) = \Omega(g(n))$$



Scanned with CamScanner

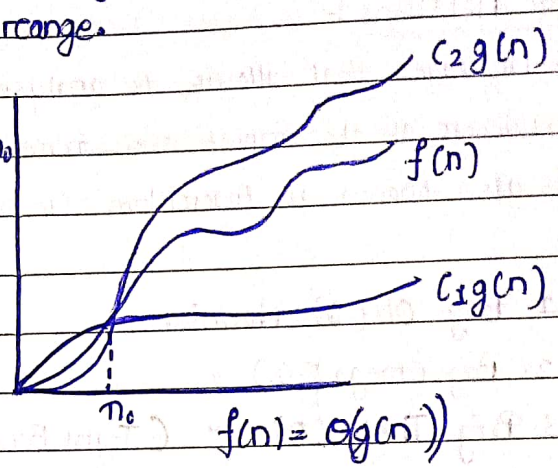
Homogenous - similar type of data (array) elements

Heterogenous/Non-Homogenous - different type of data elements (structure) (record) (list)

3. Big Theta (Θ) Notations :-

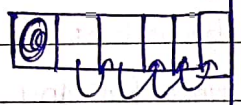
→ It defines the average value or range within which the execution time of an algorithm will range.

$$\boxed{c_1 g(n) \le f(n) \le c_2 g(n)} \quad \forall n \ge n_0$$

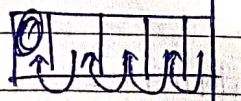


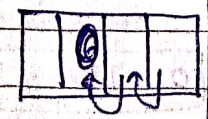
Array :-

```
/* searching an element */
for (i=0; i<n; i++)
  if (x == a[i])
    { pos = i;
      k++;
      break;
    }
if (k>0) pf("Element is present at %d", pos);
else pf("not present.
```



```
/ Sorting an element */
for (i=0; i<9; i++)
for (j=0; j<9-i; j++)
if (a[j] > a[j+1])
  { temp = a[j];
    a[j] = a[j+1];
    a[j+1] = temp;
  }
}
for (i=0; i<=9; i++)
  pf ("%d \n", a[i]);
```





```
/* insert beginning */
{ for (i=n; i>0; i--)
    { a[i+1] = a[i]
    } a[0] = ele;
}

/* insert specific */
{ for (i=n; i>=p; i--)
    { a[i+1] = a[i];
    }
  a[p] = ele;
}

/* delete beg */
for (i=0; i<n; i++)
  { b[i] = b[i+1];
  }
n = n-1

/* delete specific */
for (i=k; i<z; i++)
  { a[i] = a[i+1];
  }
z = z-1;
```

Scanned with CamScanner

Sparse Matrix :- Most of the elements are zero.

Ex- $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix} \longrightarrow$ Sparse Matrix

$$\begin{array}{c|cccc} & 0 & 1 & 2 \cdots\cdots & 99 \\ \hline 0 & 1 & 0 & 0 \cdots & 0 \\ 1 & 0 & 1 & 0 \cdots & 0 \\ 2 & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 59 & 0 & 0 & 1 \cdots & 0 \end{array}$$

$60 \times 4 = 240$ bytes

Huge Memory Loss

$60 \times 100 = 6000 \times 4$ bytes

$= 24{,}000$ bytes

To avoid wastage of memory, an alternative representation is used which stores only non-zero elements in a triplet format

Sparse Matrix Triplet Form Representation :-

[row col val]

no. of columns = 3

no. of rows = no. of non-zero elements + 1

	0	1	2
0	Total no. of rows in Sparse Matrix	Total no. of columns in Sparse Matrix	Total no. of non-zeros in Sparse Matrix
1			
2			
⋮			
no. of non-zeros			

Scanned with CamScanner

Ex:-

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 2 & 0 & 3 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 5 & 0 & 0 & 0 & 0 \end{array}_{5\times 6}$$

$\downarrow$ Triplet form

$$\begin{array}{ccc} 5 & 6 & 4 \\ \hline 0 & 0 & 1 \\ 2 & 2 & 2 \\ 2 & 4 & 3 \\ 4 & 1 & 5 \end{array}_{5\times 3}$$

Ex-

$$\begin{array}{c|ccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 5 & 0 & 3 & 0 & 0 \end{array}_{4\times 7}$$

$\downarrow$ Triplet form

$$\begin{array}{ccc} 4 & 7 & 4 \\ \hline 0 & 3 & 2 \\ 1 & 1 & 1 \\ 3 & 2 & 5 \\ 3 & 4 & 3 \end{array}_{5\times 3}$$

Scanned with CamScanner

TRIPLET PROGRAM

```
K=1;
for(i=0; i<row; i++)
  { for(j=0; j<col; j++)
    { if(a[i][j]!=0)
      { b[K][0]=i;
        b[K][1]=j;
        b[K][2]=a[i][j];
        K++;
      }
    }
  }

b[0][0] = row;
b[0][1] = col;
b[0][2] = K-1;
for(i=0; i<K; i++)
  { for(j=0; j<3; j++)
    { printf("%d\t", a[i][j]);
    }
    printf("\n");
  }
```

K is the no. of row of sparse Matrix of triplet form

$$\begin{bmatrix} 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10 & 0 \end{bmatrix}_{5\times6}$$

Triplet form

$$\begin{bmatrix} 5 & 6 & 5 \\ 0 & 2 & 5 \\ 1 & 3 & 1 \\ 2 & 0 & 7 \\ 3 & 0 & 8 \\ 4 & 4 & 10 \end{bmatrix}_{6\times3}$$

Scanned with CamScanner

Triplet to original matrix :-

$$\begin{bmatrix} 4 & 7 & 4 \\ 0 & 3 & 2 \\ 1 & 1 & 1 \\ 3 & 2 & 5 \\ 3 & 4 & 3 \end{bmatrix} \xrightarrow[\text{form}]{\text{original}} \begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 3 & 0 & 0 \end{bmatrix}_{4\times 7}$$

$$a = \begin{bmatrix} 3 & 3 & 2 \\ 0 & 0 & 5 \\ 1 & 2 & 11 \end{bmatrix} \xrightarrow[\text{form}]{\text{Original}} b = \begin{array}{c} \\ K \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{bmatrix} 5 & 0 & 0 \\ 0 & 0 & 11 \\ 0 & 0 & 0 \end{bmatrix}_{3\times 3} \end{array}$$

```
//*enter the triplet representation and convert it to original form*
    row = a[0][0];
    Col = a[0][1];
    K = 1;
  for(i=0; i< row; i++)
   {
     for(j=0; j< col; j++)
        {  if(a[K][0] == i && a[K][1] == j)
             {  b[i][j] = a[K][2];
                K++;
             }
          else
             b[i][j] = 0;
        }
   }
```

Transpose of Sparse Matrix :-

$$\begin{array}{c} \\ i \downarrow \\ \\ a \end{array}\begin{array}{c} \begin{array}{ccc} r & c & v \end{array} \\ \begin{bmatrix} 4 & 7 & 4 \\ 0 & 3 & 2 \\ 1 & 1 & 1 \\ 3 & 2 & 5 \\ 3 & 4 & 3 \end{bmatrix} \end{array} \xrightarrow{\text{Transpose}} \begin{array}{c} \\ k \downarrow \\ \\ b \end{array}\begin{array}{c} \begin{array}{ccc} c & r & v \end{array} \\ \begin{bmatrix} 7 & 4 & 4 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 2 & 3 & 5 \\ 3 & 0 & 2 \\ 4 & 3 & 3 \end{bmatrix} \end{array}$$

The elements in $[i][j]$ will move to $[j][i]$

```
r = a[0][0];
c = a[0][1];
v = a[0][2];
k = 1;
b[0][0] = c;
b[0][1] = r;
b[0][2] = v;
for (j=0; j<c; j++)
    { for (i= 1; i<=v; i++)
        { if (a[i][1] == j)
            { b[k][0] = a[i][1];
              b[k][1] = a[i][0];
              b[k][2] = a[i][2];
              k++;
            }
        }
    }
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 15 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 2 & 3 & 2 \\ 0 & 1 & 1 \\ 1 & 0 & 15 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 2 & 2 \\ 0 & 1 & 15 \\ 0 & 1 & \\ 1 & 0 & 1 \end{bmatrix}$$

Scanned with CamScanner

Addition of Sparse Matrix :-

$$a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}_{4\times4} \qquad b = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}_{4\times4}$$

Triplet Representation

Triplet representation

$$m_1 = \begin{bmatrix} 4 & 4 & 3 \\ 0 & 0 & 1 \\ 1 & 3 & 5 \\ 2 & 3 & 8 \end{bmatrix} \qquad m_2 = \begin{bmatrix} 4 & 4 & 1 \\ 1 & 3 & 2 \end{bmatrix}$$

$$m_1 + m_2 = \begin{bmatrix} 4 & 4 & 3 \\ 0 & 0 & 1 \\ 1 & 3 & 7 \\ 2 & 3 & 8 \end{bmatrix}$$

Ex

$$m_1 = \begin{bmatrix} 4 & 4 & 3 \\ 0 & 0 & 1 \\ 1 & 3 & 5 \\ 2 & 3 & 11 \end{bmatrix} \qquad m_2 = \begin{bmatrix} 4 & 4 & 2 \\ 1 & 3 & 2 \\ 3 & 3 & 5 \end{bmatrix}$$

5-1

$$\begin{bmatrix} 4 & 4 & 4 \\ 0 & 0 & 1 \\ 1 & 3 & 7 \\ 2 & 3 & 11 \\ 3 & 3 & 5 \end{bmatrix}$$

Scanned with CamScanner

Ex- $m_1 = \begin{bmatrix} 7 & 7 & 6 \\ 0 & 5 & 1 \\ 1 & 4 & 1 \\ 3 & 4 & 3 \\ 4 & 1 & 2 \\ 5 & 4 & 4 \\ 6 & 4 & 2 \end{bmatrix}$ $m_2 = \begin{bmatrix} 7 & 7 & 9 \\ 0 & 0 & 1 \\ 0 & 4 & 9 \\ 1 & 4 & 2 \\ 2 & 1 & 4 \\ 2 & 5 & 8 \\ 3 & 0 & 6 \\ 3 & 2 & 7 \\ 5 & 0 & 5 \\ 6 & 4 & 9 \end{bmatrix}$

$m_1 + m_2 = \begin{bmatrix} 7 & 7 & 13 \\ 0 & 0 & 1 \\ 0 & 4 & 9 \\ 0 & 5 & 1 \\ 1 & 4 & 3 \\ 2 & 1 & 4 \\ 2 & 5 & 8 \\ 3 & 0 & 6 \\ 3 & 2 & 7 \\ 3 & 4 & 3 \\ 4 & 1 & 2 \\ 5 & 0 & 5 \\ 5 & 4 & 4 \\ 6 & 4 & 11 \end{bmatrix}$

LS-2

Scanned with CamScanner

Stack:-

→ It is a linear DS in which items are inserted or deleted only at one end called the top of the stack.

→ It follows LIFO (Last in Fast Out) mechanism where the last element inserted in the stack will be the first one to be removed from the stack.

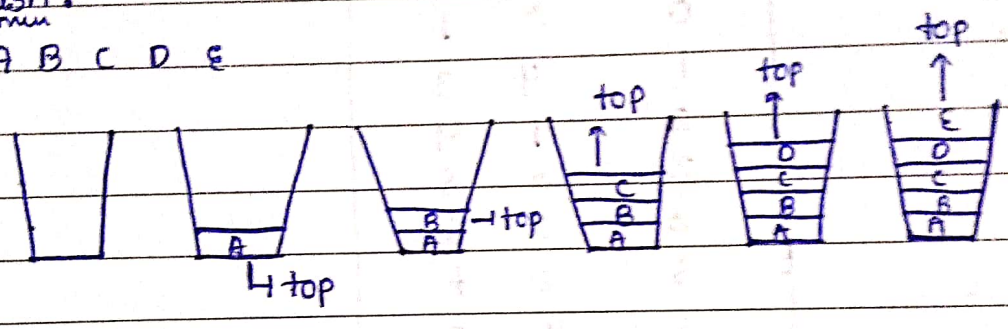
→ There are two basic operations associated with Stack:-

(1) Push - used to insert an element into the stack.

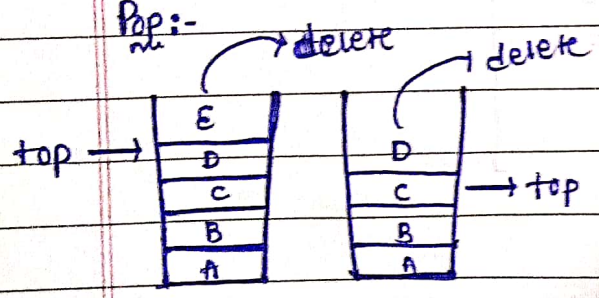
(2) Pop - used to delete an element.

Push:-

A B C D E

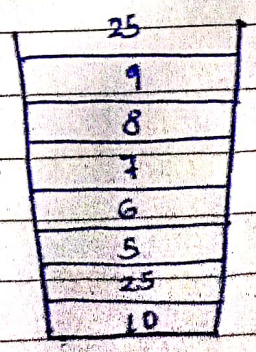


Pop:-



Ex:- For the given pattern find the contents of the stack after applying 4 insert, 3 delete operations.

10, 25, 5, 6, 7, 8, 9 (25, 60, 70, 80)



Scanned with CamScanner

Overflow:- trying to insert element into the stack after maximum size is reached.

Underflow :- trying to delete elements from an empty stack.

Q. For the following sequence of operations, what will be the sequence of popped out values?

Push (1), Push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop

Ans:- 2 2 1 1 2

Representation of Stack :-

Stack can be represented in 2 ways :-

(a) Array

(b) Linked List

Array representation of Stack :-

```
# include<stdio.h>
# include <stdlib.h>
void push ();
void pop();
void display ();
# define max 50
int stack[max];
int top=-1;
// int c=0;
main()
{ int ch;
  while(1) {
  printf ("1-push\n 2-pop\n 3-display\n 4- exit\n");
  scanf ("%d",&ch);
  Switch(ch)
      { case 1 :
                push();
                // c++; break;
```

```
Case 2:
        pop();
        break;
Case 3:
        display();
        break;
Case 4:
        exit(0);
default:
  printf("Invalid option\n Enter a valid option\n");
}
}
}

Void push()
{ int ele;
  if (top==max-1)
   printf("overflow");
  else {
        printf("Enter the element to be inserted\n");
        scanf("%d", &ele);
        top++;
        Stack[top] = ele;
        }
}
Void pop()
{ if (top==-1)
    printf("Underflow");
    else {
          int item = Stack[top];
           top--;
          // c--;
          printf("%d is poped out\n", item);
          }
    }
```

top++; Stack[top] = ele; } → can be written as stack[++top] = ele;

Scanned with CamScanner

```
void display () { int i;
    if (top==-1)
    printf ("stack is empty \n");
    else {
        pf ("Elements of stack are \n");
        //for (i=0; i<c; i++)      for (i=0; i<=top; i++)
        {
            printf ("%d \t", stack[i]);
        }
    }
}
```

* To print the stack elements from top to bottom :-

```
void display
{ int i;
  if (top==-1)
  pf ("Stack is empty \n");
  else
  { pf ("Elements of stack are \n");
    for (i= top; i>=0; i--)
      { pf ("%d \t", stack[i]);
      }
}
```

Scanned with CamScanner

```
Void display ()  { int i;
    if (top==-1)
    printf ("Stack is empty \n");
    else {
        pf ("Elements of stack are \n");
        // for (i=0; i<c; i++)      for (i=0; i<=top; i++)
        {
          printf ("%d \t", stack[i]);
        }
    }
}
```

* To print the stack elements from top to bottom :-

```
void display
{ int i;
  if (top==-1)
  pf ("Stack is empty \n");
  else
  { pf ("Elements of stack are \n");
    for (i= top; i>0; i--)
    { pf ("%d \t", stack[i]);
    }
}
```

Scanned with CamScanner

Algorithm for push operation:-

Step-1 :- If top = max size of the stack, then print overflow. and return

Step-2 :- Otherwise read the element to be inserted. Increase top by 1 i.e top = top+1.

Step-3 :- Store the element at stack[top] position and return.

Algorithm for pop operation:-

Step-1 :- If top = -1 (Stack is empty), then print underflow and return.

Step 2 :- Otherwise set item = stack[top]. Decriment top by 1 i.e top = top-1 and then return

...

```
main()
{ int x, z;
  while(1)
  { pf("....."); }
    sf("%d", &ch);  } Menu driven part
  switch(ch)
  { case 1:
      pf("Enter the element to be pushed");
      sf("%d", &x);
      push(x);
      break;
    Case 2:
      z= pop();
      pf("%d is popped out", z);
      break;
    ...
  }
}
```

Scanned with CamScanner

```
void push (int x)
{   // condn for overflow
    top++;
    stack[top] = x;
}

int pop()
{ // underflow condn
    int y;
    y = stack[top];
    top--;
    return y;
}
```

Application of Stack :-

(a) Reversing string

(b) Evaluation of Arithmatic expression

(c) Conversion of one form of arithmatic operation to another

(d) Recursion.

Reversing string using Stack :-

```
void push(char);
char pop();
# define max 50
char stack[max];
int top = -1;
main()
{   int i;
    char str[30];
    pf(" Enter the string");
    gets(str);
    for(i=0; i<strlen(str); i++)
```

Scanned with CamScanner

```
{   push(str[i]);
}
for(i=0; i< strlen(str); i++)
 {
        str[i] = pop();
 }  pf("Retrieved string is");
    puts(str);
}
Void push(char ch)
{   if (top==max)
      pf("Overflow");
    else {  top++;
            stack[top]=ch;
         }
}

char pop()
  { // underflow condn.   char c;
                          c = stack[top];
                          top--;
                          return c;
  }
```

Scanned with CamScanner

Date
Page

Decimal to binary conversion using stack :-

```
void push (int);
int pop ();
# define max 50
    int stack [max];
    int top = -1;
main()
{ int i, num, rem, c = 0
  pf("Enter a number");
  sf("%d", &num);
  while ( num != 0)
    { rem = num % 2;
      push( rem);
      num = num/2;
      c++;
    }
    i = 0;
  while (i < c)
    { int a;
        a = pop();


void push ( int n)
  {
      top++;
      stack [top] = n;
  }
int pop ()
  {
```

Scanned with CamScanner

Notations for arithmatic operations (expressions):-

There are 3 notations used to represent arithmatic expressions

(a) Infix :- operator is placed betn operands.

Ex- A + B → operands
↳ operator

(b) Postfix :- operator is placed after the operands.

Ex- AB+, CD-, GH*

(c) Prefix :- operator is placed before the operands.

Ex- +AB, -CD, *GH (Also called Polish Notation)

Ex- (A+B)*C → Infix
AB+C* → Postfix

Ex- (A+B)/(C-D) → Infix
AB+CD-/ → Postfix

Ex- A+B/C-D → Infix
ABC/+D- → postfix

Ex- (A+B)*(C+D) → AB+CD+*

Scanned with CamScanner

Add^n of Sparse Matrix

```
main()
{

  if(m1[0][0] != m2[0][0] || m1[0][1] != m2[0][1])
   {  pf("Add^n not possible");
      exit(0);
   }
  nzero1 = m1[0][2];
  nzero2 = m2[0][2];
  K1 = K2 = K3 = 1;
  int c = 0;
while( K1 < nzero1 + 1 && K2 < nzero2 + 1)
  {
    if(m1[K1][0] < m2[K2][0])
     {  m3[K3][0] = m1[K1][0];
        m3[K3][1] = m1[K1][1];
        m3[K3][2] = m1[K1][2];
        K3++;
        K1++;
     }
else if(m1[K1][0] > m2[K2][0])
     {  m3[K3][0] = m2[K2][0];
        m3[K3][1] = m2[K2][1];
        m3[K3][2] = m2[K2][2];
        K3++;
        K2++;
     }
else if((m1[K1][0] == m2[K2][0]) && (m1[K1][1] < m2[K2][1]))
     {  m3[K3][0] = m1[K1][0]
        m3[K3][1] = m1[K1][1]
        m3[K3][2] = m1[K1][2]
        K3++;
     }  K1++;
```

```
else if((m1[K1][0] == m2[K2][0]) && (m1[K1][1] > m2[K2][1]))
{   m1[K3][0] = m2[K2][0];
    m1[K3][1] = m2[K2][1];
    m1[K3][2] = m2[K3][2];
        K3++;
        K2++;
  }
else {
    m3[K3][0] = m2[K2][0];
    m3[K3][1] = m2[K2][1];
    m3[K3][2] = m2[K2][2] + m1[K1][2];
    c++;
    K3++;
    K2++;
    K1++;
  }
m3[0][0] = m1[0][0];
m3[0][1] = m1[0][1];
m3[0][2] = (nzero1 + nzero2) - c;
```

Scanned with CamScanner

(A+B) → parenthesis free

Arithmetic expressions :-

Ex- A + B * C - D/E * H ⟶ ABC*+DE/H*-

Input Symbol	Output position string	Stack
A	A	
+		+
B	AB	
*	AB	+ *
C	ABC	
-	ABC*+	-
D	ABC*+D	
/		- /
E	ABC*+DE	
*	ABC*+DE/	- *
H	ABC*+DE/H	
Input empty	ABC*+DE/H*-	

Algorithm for infix to postfix :-

Step 1:- Scan the infix expression from left to right

Step 2:- (a) If ~~stack~~ scanned symbol is a left parathesis, push it onto the stack

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

(c) If its a right parathesis, then pop all elements from the stack and place them in postfix expression till a matching left parenthesis is obtained.

(d) If its an operator, then (i) remove all operators from stack having higher precedence than the scanned one and then place it in postfix expression.

(ii) Otherwise, push the scanned operator onto the stack.

Step 3:- If the input is empty, pop out all stack elements and place in the output postfix expression.

Scanned with CamScanner

(A+B) → parathesis form

Arithmatic expressions :-

Ex- A+B*C - D/E*H ⟶ ABC*+DE/H*-

Input Symbol	Output position string	Stack
A	A	
+		+
B	AB	
*	AB	* +
C	ABC	* +
-	ABC*+	-
D	ABC*+D	
/		/ -
E	ABC*+DE	
*	ABC*+DE/	* -
H	ABC*+DE/H	
Input empty	ABC*+DE/H*-	

Algorithm for infix to postfix :-

Step 1:- Scan the infix expression from left to right

Step 2:- (a) If ~~stack~~ scanned symbol is a left parathesis, push it onto the stack

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

(c) If its a right parathesis, then pop all elements from the stack and place them in postfix expression till a matching left parenthesis is obtained.

(d) If its an operator, then (i) remove all operators from stack having higher precedence than the scanned one and then place it in postfix expression.

(ii) Otherwise, push the scanned operator onto the stack.

Step 3:- If the input is empty, pop out all stack elements and place in the output postfix expression.

Scanned with CamScanner

Infix to prefix conversion :-

Ex- $(A+B)*C \longrightarrow *+ABC$

Infix — prefix

Ex- $(A+B)/(C-D) \rightarrow /+AB-CD$

Ex - $A+B/C-D \rightarrow -+A/BCD$

Evaluation of Postfix expression :- (Algorithm)

Step 1- Scan the expression (input) from left to right

Step 2- If its an operand, push it onto the stack.

Step 3- If its an operator, pop out two elements and apply the operator and push the result on the stack.

Step 4- Write the result if the input is empty.

Ex - 6523+8*+3+*

Symbol	operands	operands	value	Stack
6				6
5				5, 6
2				2, 5, 6
3				3, 2, 5, 6
+	3	2	5	5, 5, 6
8				8, 5, 5, 6
*	5	8	40	40, 5, 6
+	40	5	45	45, 6
3				3, 45, 6
+	45	3	48	48, 6
*	48	6	288	288

Answer = 288

Scanned with CamScanner

Queue:- A queue is a linear list of elements in which insertion can take place only at one end called rear and deletion can take place only at the other end called front.

→ Queue follows FIFO Mechanism (First in First out) where the first element inserted into the queue will be the first element to be removed from the queue.

0	1	2	3	4

Queue

front = 0

rear = -1

1. Enqueue:- Add elements into the queue. // rear = rear + 1
2. Dequeue:- Remove elements from the queue. // front = front + 1

Ex:-

Assuming max. size of linear queue = 6. what is the value of front and rear after the following operations? (insert(10), insert(20), insert(30), delete(), delete(), insert(40), insert(60))

Ans:-

0	1	2	3	4
10	20	30	40	60

Front

rear

Ex- What is the content of the queue after applying operations (delete(), delete(), delete(), insert(66), delete(), insert(77))

0	1	2	3	4	5	6	7	8	9	10
11	22	33	44	55	66	77				

Ans:-

Ex- If initial configuration of the queue is a, b, c, d and 'a' is the front end, to get the configuration d, c, b, a in the queue, how many min. no. of delete and insert function are required.

a, b, c, d

d c b a

Ans:- 3 delete(), 3 insert

Algorithm (Implementation of linear queue using array):-

(a) For inserting:-

Step-1:- if queue is full (rear = size-1), print overflow and return.

Step-2:- Otherwise increase rear by 1 and store the element to be inserted at queue[rear] and return.

(b) For deletion:-

Step-1:- if queue is empty (rear = -1), print underflow and return.

Step 2:- Otherwise, remove the element from queue[front] and increase front by 1 (front = front + 1) and then return.

```
# include <stdio.h>
# define max 50
int queue[max];
int front = 0;
int rear = -1;  void insert();  void delete();
main()
{

    Menu Driven Program

}
void insert ()
{ int [redacted] ele;
  [redacted]
  if (rear == max - 1)
  pf("overflow");
else { pf("Enter the element to be inserted");
        sf("%d"&ele)
       rear ++;
       queue[rear] = ele;
  }
}
```

```
void delete()
{   if (rear == -1)
      pf("Underflow");
    else {
          int item = queue[front];
          front ++;
          pf(" %d is poped/deleted out \n", item);
        }
}
```

Note:- If rear == front, it indicates that there is only one element in the queue.

Limitation of linear queue :-

→ When rear = max - 1, it indicates overflow condition i.e the queue is full.

→ Once the queue is full, if some elements are deleted from the front, then some free spaces are available in the queue.

→ But these free spaces cannot be reused for storing new elements as rear = max - 1 which indicates overflow situation.

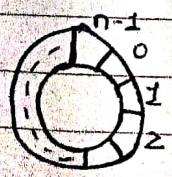
Solutions :-

→ Shifting all remaining elements in the queue towards front and updating rear position. However, it is not an effective method as shifting of all remaining elements each time a deletion is performed will be a time consuming process.

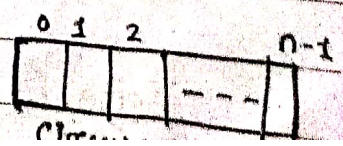
→ Convert linear queue into circular queue.

Circular queue :-

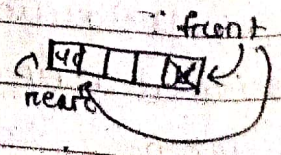
→ A circular queue is a queue in which the last and first positions are considered as adjacent position i.e the element next to the last element is the first element.



Circular queue (logical)



```
void insert () {  int ele;  // Enter the element
    if ((front == 0 && rear == max-1) || (front > 0 &&
    rear == front-1))
    pf("Overflow");
  else {
        if ( front > 0 && rear == max-1 )
          { rear = 0;
            queue[rear] = ele;
          }
      else
        { rear++;
          queue[rear] = ele;
        }
    }
  }

void delete()
{
  if ( front == 0 && rear == -1)
  pf("Underflow");

  if (front == rear)      // one element
    { item = queue[front];
      front = 0;
      rear = -1;
    }
  else {
        if (front == max-1)
          { item = queue[front];
            front = 0;
          }
  else { item = queue[front];
         front++;
```

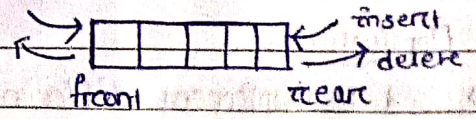


Scanned with CamScanner

```
pf("%d is deleted from the queue", item);
}
void display ()
{   if (front == 0 && rear == -1)
        pf("Queue is empty");
    if (front > rear)
    {   for (i=0; i<=rear; i++)
            pf("%d\t", queue[i]);
        for (j= front; j<= max-1; j++)
            pf("%d\t", queue[j]);
    }
    else {
        for (i= front; i<= rear; i++)
        {   pf("%d\t", queue[i]);
        }
    }
}
```

Deque:- (Double ended queue)

A deque is a linear list in which elements can be added or removed at either ends but not in the middle.

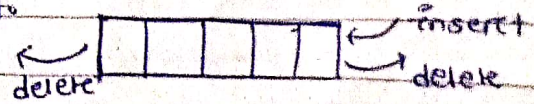


→ There are 2 variations of deque :-

(a) Input restricted deque

(b) Output restricted deque

(a) An Input restricted deque allows insertion at only one end of the list but allows deletion at both ends of the list.



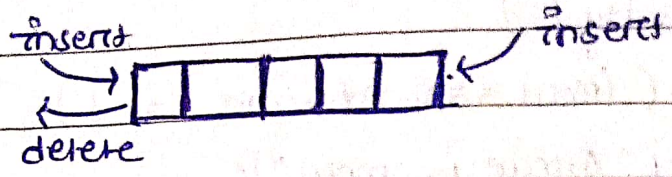
1- insert
2- delete front
3- delete rear
4- display
5- exit

} Menu

(b) An output restricted deque allows deletion at only one end of the list but allows insertion at both the ends.

1- insert front
2- insert rear
3- delete
4- display
5- exit

Menu



Priority Queue :-

A priority queue is a collection of elements where each element is associated with a priority value. The elements are deleted and processed using the following rules :-

→ An element with higher priority is processed before any element of lower priority.

	11	12	13	14
Priority	2	4	1	3

high ↓ low

4
3 — 12 14 11 13
2
1

→ Two elements with same priority are processed according to the order in which they are added to the queue

Application :-

→ Used to implement different CPU scheduling algorithms in multiprogramming and time sharing systems.

Scanned with CamScanner

LL

Limitations of array :-

- Size needs to be defined when program is written, therefore additional space cannot be provided if required.
- Insertion and deletion in array are expensive as each insertion and deletion requires shifting of remaining elements.

Sol^n :-

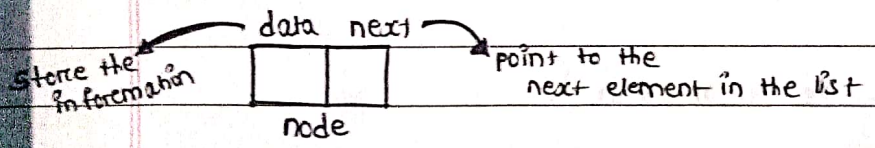
- Dynamic link representation of list

Linked List :-

→ It is a dynamic data structure where :-

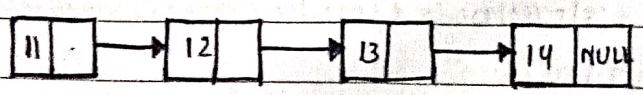
- amount of memory required can vary during use.
- adjacenty betn the elements is maintained by links/pointers i.e the address of the subsequent elements.

In a linked list, data and link, both are required to be maintained. An element in a linked list is called a 'node'. A node consist of two fields i.e 'data' and 'next'



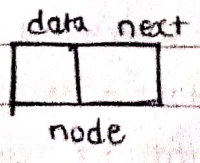
① Single linked list :- (one way list)

- Each node in SLL contains a data field and only one pointer field which will point to the adjacent node in the list



Structure of a node :-

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



Scanned with CamScanner

Allocating blocks of memory to a node :-

```
struct node * newnode;
newnode = (struct node *) malloc (sizeof (struct node));
```

Program to create and Traverse a linked list :-

```
#include <stdio.h>
# include <stdlib.h>
# include <malloc.h>
struct node {
        int data;
        struct node * next;
    } * start = NULL, *newnode, * current;
void create ();
void display();
void main()
{ create ();
  pf ("The elements of the list are : \n");
  display ();
}
void create ()
{ char ch;
  do { newnode = (structnode *) malloc (sizeof (structnode));
  pf("Enter the data \n");
  sf (" %d", &newnode → data);      or   { int x;
                                           sf("%d", &x)
                                           newnode → data = x;
  newnode → next = NULL;
  if( start == NULL)
  { start = newnode;
    current = newnode;
  }
  else { current → next = newnode;
         current = newnode;
  }
```

Scanned with CamScanner

```
pf(" Press n to stop\n");
ch=getchare();
}while(ch!='n');
}
Void display()
{    Struct node *ptr;
     ptr = start;
     while(ptr != NULL)
     {
          pf("%d\t", ptr->data);
          ptr = ptr->next;
     }
}
```

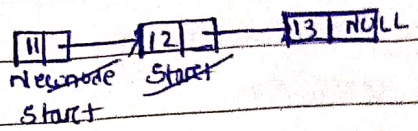
Menu driven program for insertion and deletion :-

```
#--------
Struct node
   { int data;
     Struct node *next;
   }*start, *current, *newnode;
main()
{  int x,ch;

   newnode=(structnode*)malloc(sizeof(structnode));
   pf("Enter the data for first node");
   sf("%d", &x);
   newnode->data=x;
   newnode->next=NULL;
   start=newnode;
   current = newnode;
   while(1) {
```

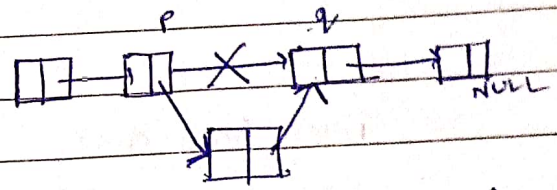
```
pf("Enter choice\n");
pf("1- insert end \n 2- insert beginning \n 3- display \n 4- exit \n");
sf("%d",&ch);
switch(ch)
{   case 1: insertend();
            break;
    case 2: insertbeg();
            break;
    case 3: display();
            break;
    case 4: exit();
            break;
    default: pf("Enter a valid choice");
            break;
 }
}

void insertend()
{ int x;
 newnode = (struct node *) malloc (sizeof(structnode));
 pf("Enter the data \n");
 sf("%d",&x);
 newnode -> data = x;
        newnode -> next = NULL;
        current -> next = newnode;
        current = newnode;
 }
void insertbeg()
 { int x;
  newnode = (struct node *) malloc (sizeof(structnode));
  pf("Enter the data \n");
  sf("%d",&x);
    newnode -> data = x;
```

Scanned with CamScanner

```
newnode -> next = start;
start = newnode;
}
```

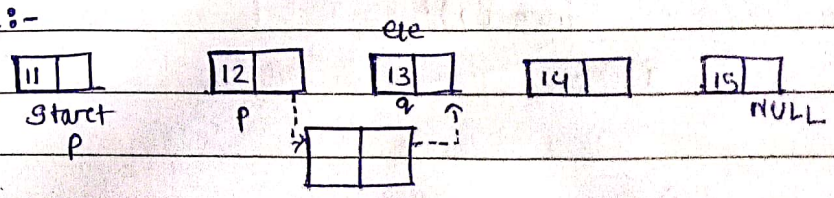


Insert after a specific value :-




```
void insertafter()
{   int ele;
    Struct node *p, *q;
    p = start;
    // * enter the data which you want to insert * //
    printf("Enter the pos")
    scf
    pf("Enter the element
    while (p -> data != ele && p -> next != NULL)
    {   p = p -> next;
    }
    q = p -> next;
newnode : (Structnode) * malloc (sizeof(Structnode));
newnode -> data = x;
p -> next = newnode;
newnode -> next = q;
```

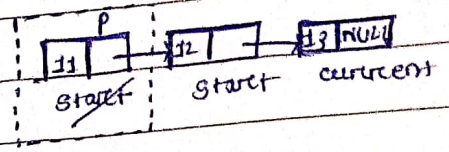
Insert before a specific value :-




```
while (p -> next -> data != ele)
{
    p = p -> next;
}
q = p -> next;
```

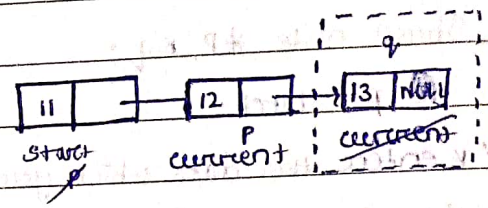
Delete from beginning :-

```
Void delete(beg)
{ struct node *p;
    p= start;
    start = start→next
    free(p);
}
```



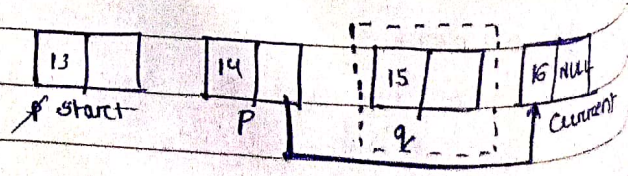
Delete from end :-

```
Void deleteend()
{
    struct node *p, *q;
    p= start;
    while (p→next != current)
    {
        p= p→next;
    }
    q= p→next;
    p→next = NULL;
    current = p;
    free (q);
}
```

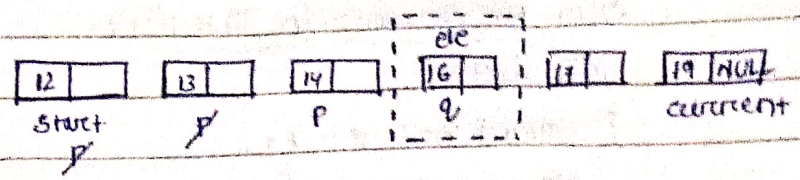


Delete after value :-

```
void deleteafter()
{ struct node *p, *q;
    p= start;
    while (p→data != ele)
    { p= p→next;
    }
    q= p→next;
    p→next = q→next;
    free(q);
```



Delete a specific value:-




```
void deletevalue()
{       struct node *p, *q;
        p = start;
    while(p→next→data != ele)
        {   p = p→next;
        }
        q = p→next;
        p→next = q→next;
        free(q);
}
```

Q. To count the no. of nodes in a linked list (function)? void nodes () {

```
    struct node *p;  p = start;
    int c = 1;
    while(p→next != NULL)
    { p = p→next;
      c++;
    }
    // print c
}
```

Searching a specific element:-

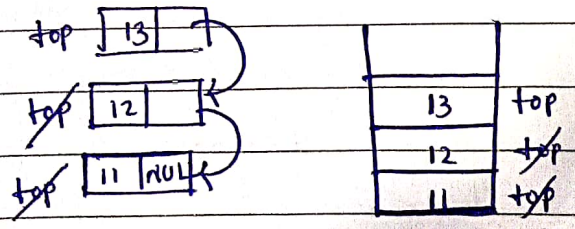
```
void search ()
{   struct node *p;  int ele;
        p = start;
    // enter the element to be searched
    while (p != NULL)
    { if(p→data == ele)
    { //element found
        //return
    }
    p = p→next;
    }
    pf("element not found");
}
```

Sort the elements in link list :-

```
void sort ()
{ struct node *p, *q;
  int temp;
  for (p= start; p→next != NULL; p= p→next)
  for (q=p→next; q != NULL; q= q→next)
    { if(p→data > q→data)
        { temp= p→data;
          p→data = q→data;
          q→data = temp;
        }
    }
}
```

* reversing
* Concatination / Merging

Link list representation of Stack :-



Push:-

```
struct node {
    int data;
    struct node *next;
    } *top= NULL, *newnode;
void push() int x;
{   newnode =(struct node *) malloc (sizeof (struct node));
    newnode → data =x;        // Enter the value of x
    newnode → next = top;
    top= newnode;
}
```

```
void pop()
{ struct node *P;
    p= top;        // pf("%d is deleted from stack", top -> data);
    top= top -> next;
    free(p);
}
```

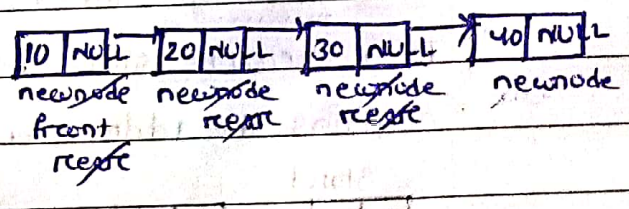
top [12] → top [11 | NULL]

```
void display()
{ struct node *P;
    p= top;
  while (P != NULL)
    {
        // print p->data
        p= p->next;
    }
}
```

```
if (top == NULL)
~~pf("stack is empty")~~
pf("Stack is empty");
```

Link list representation of Linear Queue :-

```
Struct node {
        int data;
        struct node *next;
    } *newnode, *front = NULL, *rear= NULL;
```



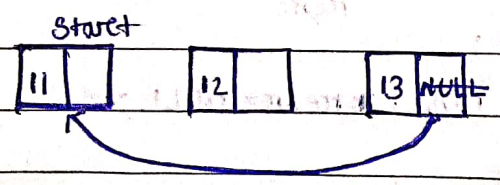

```
void insert()   int x;
{ newnode = (structnode *)malloc(sizeof(structnode));
  //Enter the value of x
  newnode -> data = x;
  newnode -> next = NULL;
  if (rear == NULL)
  { front = newnode;
    rear = newnode;
  }
  else {
```

Scanned with CamScanner

```
void delete ()
{ struct node *p;
    p= front;
    front = front -> next;
    item = p -> data;
    pf ("%d is deleted", item);
}
void display ()
{ struct node *p;
    p= front;
  while (p != NULL)
{ pf(" %d\t", p -> data)
    p= p -> next;
}
}
```

Circular Single Linked List :-

In a circular single linked list, the next field of last node contains the address of 1st node instead of NULL pointer.



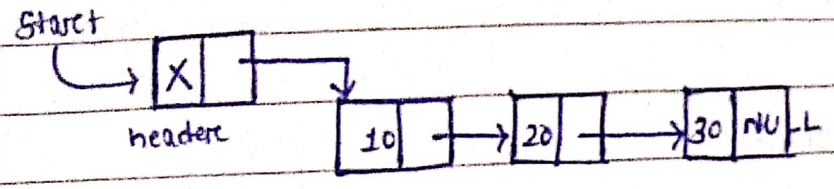

```
void insertend ()
{ - - - - - - -
  newnode -> data = x;
  newnode -> next = start;
  - - - - - -
}
```

```
void display()
{ struct node *p;
    p=start;
    do
    { pf("%d\t", p -> data);
      p= p -> next;
    } while (p != start);
}
```

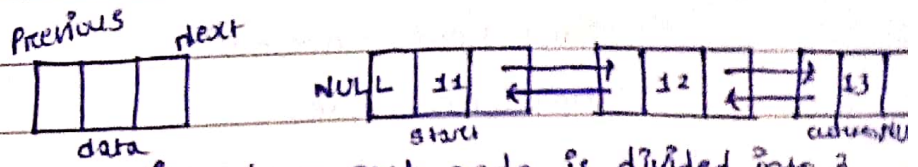
Scanned with CamScanner

Header Linked List :-

A header linked list is a list which always contains a special node called the header node at the beginning of the list.



Double Linked List :-



- A double linked list is a two way list where each node is divided into 3 parts :- 1. Data
  2. Next – points to the next node in the list
  3. Previous – points to the previous of the list
- The previous of 1st node and next of last node contains NULL which indicates the beginning and end of the list.
- The advantage of Double LL over Single LL is that, traversal is possible in both directions in case of Double LL.

```
Struct node {
        int data;
        Struct node *next, *prev;
    } *start, *current, *newnode;
main() { int x;
    newnode = (Struct node*) malloc (sizeof(Structnode));
    // Enter the value of x
    newnode → data = x;
    newnode → next = NULL;
    newnode → prev = NULL;
    Start = newnode;
    current = newnode;
  While(1) {
        Menu driven part (including switch)
  }
```


Scanned with CamScanner

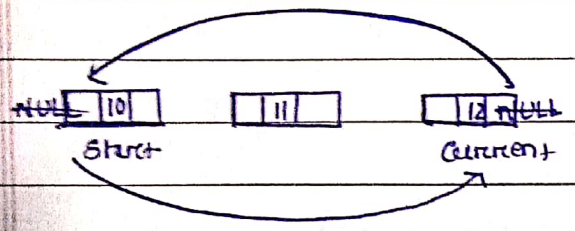
```
void insertBeg()
{   int x;
    newnode = (struct node *) malloc(sizeof(struct node));
    //Enter the value of x
    newnode -> data = x;
    newnode -> prev = NULL;
    newnode -> next = start;
    start -> prev = newnode;
    start = newnode;
}
void insertEnd()
{ int x;
  newnode = (struct node *) malloc (sizeof(struct node));
  //Enter the value of x
  newnode -> data = x;
  newnode -> next = NULL;
  newnode -> prev = current;
  current -> next = newnode;
  current = newnode;
}
void Traverseforeword ()
{ struct node *p;
    p = start;
  while (p != NULL)
  { pf ("%d\t", p -> data);
    p = p -> next;
  }
}
void Traverseback ()
{ struct node *p;
  p = current;
  while (p != NULL) { pf ("%d\t", p -> data)
```

Scanned with CamScanner

```
void deleteBeg()
{ struct node *p;
    p= start;
    start = start->next;
    free(p);
    start->prev = NULL;
}
```

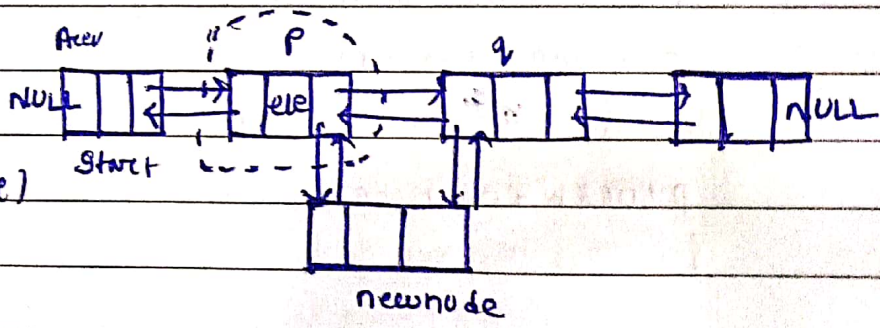
```
void deleteEnd()
{ struct node *p;
    p = current;
    current = current->prev;
    free(p);
    current->next = NULL;
}
```

Circular Double linked List :-




```
start->prev = current;
current->next = start;
```

Insert after specific value :-




```
void insertafter()
{ //Enter value after
  which you want to insert (ele)
    struct node *p, *q;
        p=start;
    while(p->data != ele)
      { p=p->next;
      }
    q = p->next;
    newnode = (struct node *) malloc(sizeof(struct node));
                newnode->data = x;
    p->next = newnode;
    newnode->prev = p;
    newnode -> next = q;
    q->prev = newnode;
}
```

Scanned with CamScanner

Q. Write a user defined function to insert after the ith node in a single linked list.

```
void insertafterpos ()
{ int x,i,pos;
  struct node *p;
  p= start;
  i=0;
  //Enter the node posn
  while(p->next != NULL && i < pos)
    {  p= p->next;
       i++;
    }
newnode =(struct node *) malloc (sizeof(struct node))
//Enter data for newnode
newnode->data =x;
newnode->next = p->next;
p->next = newnode;
}
```

to delete after the ith node (user defined function) in single linked list.

Largest element in single linked list (user defined function)

```
void largest ()
{ struct node *p;
  p= start;
  int max;
  max = p->data;
  while (p != NULL)
  {  if(p->data > max)
          max= p->data;
          p= p->next;
  }
```

// for (p= start; p != NULL; p=p

```
printf("Maximum = %d", max);
}
```

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 8 & 0 & 0 & 0 & 0 & 9 \\ 1 & 0 & 3 & 0 & 0 & 0 & 0 \\ 2 & 0 & 5 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 0 & 0 & 0 & 7 \\ 4 & 0 & 0 & 7 & 0 & 0 & 0 \end{array}$$

Sparse

$\longrightarrow$

$$\begin{bmatrix} 5 & 6 & 8 \\ 0 & 0 & 8 \\ 0 & 5 & 9 \\ 1 & 1 & 3 \\ 2 & 1 & 5 \\ 2 & 4 & 2 \\ 3 & 0 & 4 \\ 3 & 5 & 7 \\ 4 & 2 & 1 \end{bmatrix}$$

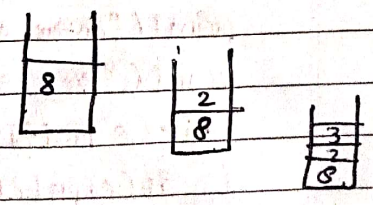
Triplet

$\downarrow$

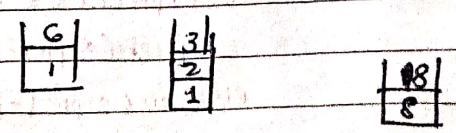
$$\begin{bmatrix} 6 & 5 & 8 \\ 0 & 0 & 8 \\ 0 & 3 & 4 \\ 1 & 1 & 3 \\ 1 & 2 & 5 \\ 2 & 4 & 7 \\ 4 & 2 & 2 \\ 5 & 0 & 9 \\ 5 & 3 & 1 \end{bmatrix}$$

Transpose

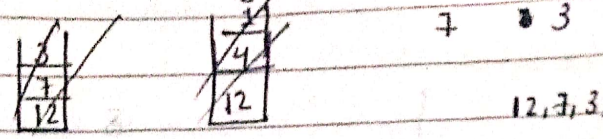
Q. Find the top two elements of the stack after first multiplication operation for the given postfix expression :- 823 ^/23* +51*-



Ans:-



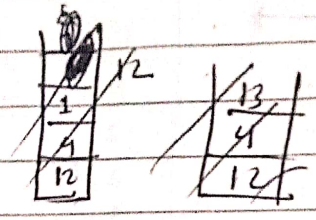
Scanned with CamScanner



7 3

12, 7, 3, −, /, 2, 1, 5, +, *, +

= 15



3
7
12

4
12

12 4

3

5
1
2
3

6

15

Infix to postfix :-

To check if the expression is balanced or not w.r.t parenthesis

```
:- char stack[max];
   int top=-1;
   void push(char);
   void pop();
Main()
{  int i=0;
   char expr[30];
   printf("Enter the expression\n");
   scanf("%s", expr);
   while(expr[i] != '\0')
   {  if(expr[i]==='c')
         push(expr[i]);
      else if (expr[i] ===')')
         pop();
      i++;
   }
```

((a+b)) → valid exp

(a+b) → invalid exp

Scanned with CamScanner

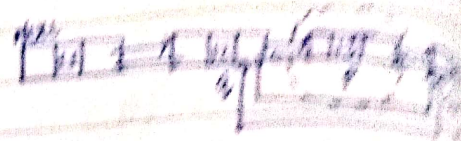
```
if (top==-1)
  pf("Valid expression");
else
  pf("Invalid expression");
}
void push(char ch)
{ top++;
  Stack[top] = ch;
}
void pop()
{
  top--;
}
```

( )
^
* /
+ -

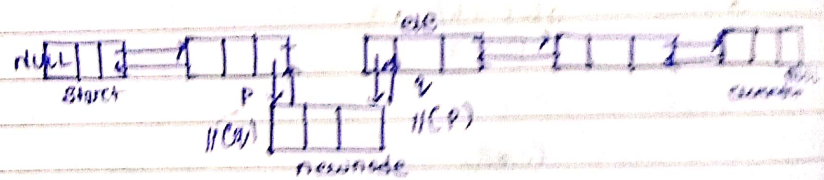
Q. Convert to postfix :- A-B+C*D^H/F+G-H

A	A	-
-		
B	AB	+
+		
C	ABC	* + -
*		
D	ABCD	^ * + -
^		
H	ABCDH	
/	ABCDH^*••	/ +
F	ABCDH^*+•F	
+	ABCDH^*••F/+	+
G	ABCDH^*+•F/G	- +
-		
H	ABCDH^*+•F/GH•+-	

```
void deletespecificvalue()
{ while ( ------ )
   { p=p->next;
   }
   q= p->next;
   rc= p->next;
   q->next = rc;
   rc->prev = q;
   free(p);
}
```



Q. Insert before a specific value

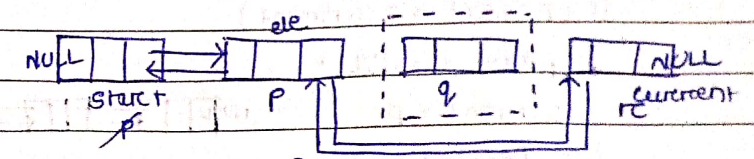



```
void insertbefore()
{ int ele,x;
  struct node *p,*q;  p=Start;
  // Enter the value before which you want to insert.
  // Enter the value you wants to insert.
  while( p->next->data != ele ) // while( p->data != ele )
     { p= p->next;
     }
     q= p->next;   // q= p->prev;
  newnode = (struct node *) malloc (sizeof (struct node));
  newnode->data = x;
   p->next = newnode;      // q->next = newnode;
   newnode->prev = p;      // newnode->prev = q;
   newnode->next = q;      // newnode->next = p;
   q->prev = newnode;      // p->prev = newnode;
}
```

Scanned with CamScanner

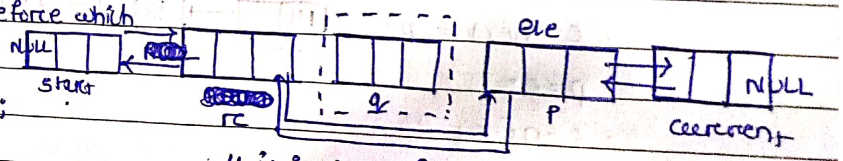
Delete after a specific value :-

```
void deleteafter()
{ struct node *p, *q;
  p = start;
  // enter the value after which you want to ~~insert~~ delete.
  while(p->data != ele)
   { p = p->next;
   }
  q = p->next;
  r = q->next;
  p->next = r;
  r->prev = p;
  free(q);
}
```

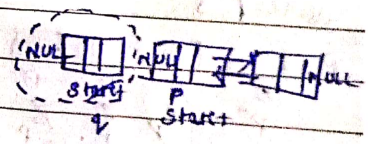


Delete before a specific value :-

```
void deletebefore()
{ // Enter the value before which
  you want to delete
  struct node *p, *q;
  p = start;
  while(p->data != ele)
   { p = p->next;
   }
  q = p->prev;
  r = q->prev;
  r->next = p;
  p->prev = r;
  free(q);
}
```



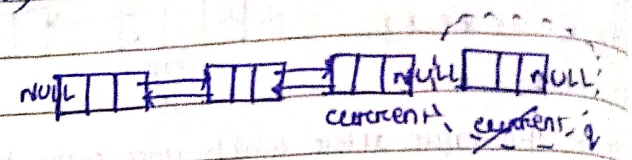

```
// if it is a first node
if (------)
{ q = p->prev;
  p->prev = NULL;
  free(q)
  start = p;
}
ele {
  delete before value program
}
```



Scanned with CamScanner

// if it is the last node q = current

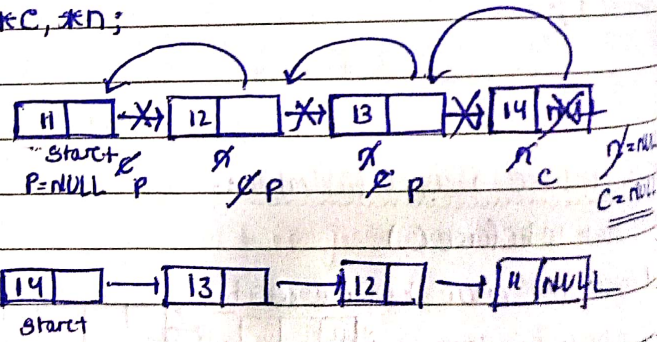
```
if (p→next == current)
{ p→next = NULL;
  current = p;
    free(q);
}
else
{ delete after value program
}
```

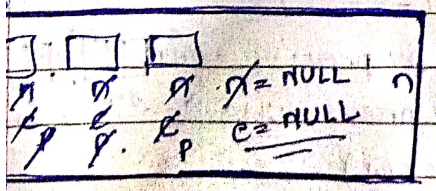


Reversing using Linked List :- (Single)

P = previous
c = current
n = next

```
void reverse ()
{ struct node *p, *c, *n;
    p = NULL;
    c = Start;
  while (c != NULL)
  { n = c→next;
    c→next = p;
    p = c;
    c = n;
  }
    Start = p;
```

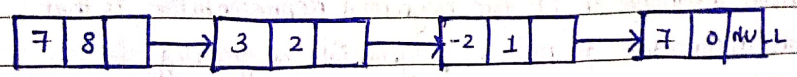




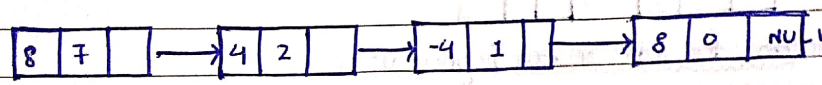
Scanned with CamScanner

Polynomials :-

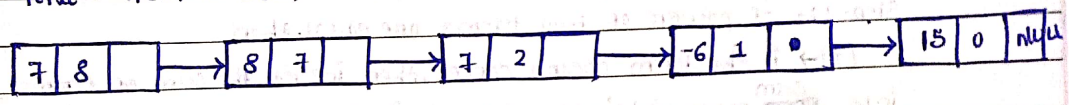
$P_1(X) = 7x^8 + 3x^2 - 2x + 7$



$P_2(X) = 8x^7 + 4x^2 - 4x + 8$



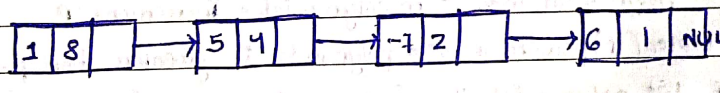
$P_{total} = P_1(x) + P_2(x) = 7x^8 + 8x^7 + 7x^2 - 6x + 15$



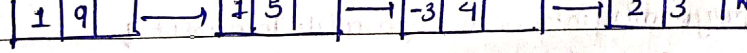
Q. $P_1(x) = x^8 + 5x^4 - 7x^2 + 6x$

$P_2(x) = x^9 + 7x^5 - 3x^4 + 2x^3$

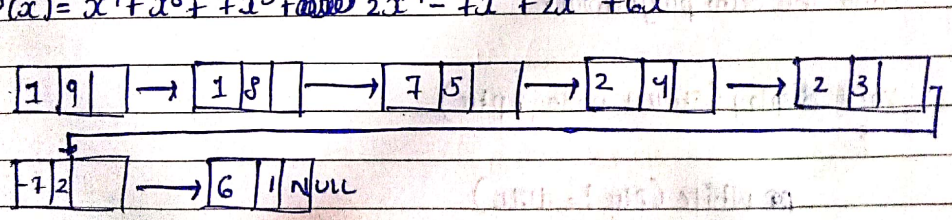
$P_1(x)$



$P_2(x)$



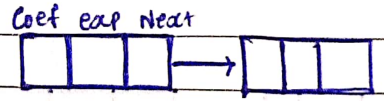
$P(x) = x^9 + x^8 + 7x^5 + 2x^4 - 7x^2 + 2x^3 + 6x$




```
struct node {
        int coef, exp;
        struct node *next;
    }
```

An important application of LL is to represent polynomials and manipulations.

The main advantage of LL for polynomial representation is that it can accomodate a no. of polynomials of growing size so that there combined size doesn't exceed the total memory size.



Algorithm for addn of two polynomials :-

Let p and q are two polynomials represented by LL.

Step-1:- While p and q are not NULL, repeat step 2.

Step-2:- If power of two terms are equal, then,

(i) → if the term doesn't cancel, then insert the sum of terms into sum polynomial (polynomial-3/rc). Increment p and q.

(ii) → else if power of first polynomial is greater than power of 2nd polynomial, then insert the term from 1st polynomial into sum polynomial. Increment p.

(iii) → else (if) power of 2nd polynomial is greater than power of 1st polynomial, then insert the terms from 2nd polynomial into sum polynomial. Increment q.

Step-3 :- Copy the remaining terms from the non-empty polynomial into the sum polynomial.

```
void display (struct node *ptrc)
{
    while (ptrc != NULL)
    { pf("%dx^%d", ptrc->coef, ptrc->exp);
      ptrc = ptrc->next;
                        if(ptrc != NULL)
                        { pf("+");
```

Q. $P_1(x) = 4x^3 + 5x^2 - 3x$ Multiplication

$P_2(x) = 2x^5 + 6x^4 + x^2 + 8$

$P(x) = 4x^3(2x^5 + 6x^4 + x^2 + 8) + 5x^2(2x^5 + 6x^4 + x^2 + 8) - 3x(2x^5 + 6x^4 + x^2 + 8)$

$= 8x^8 + 24x^7 + 4x^5 + 32x^3 + 10x^7 + 30x^6 + 5x^4 + 40x^2$

$- 6x^6 - 18x^5 - 3x^3 - 3x^2$

$= 8x^8 + 24x^7 + 10x^7 + 30x^6 - 6x^6 + 4x^5 - 18x^5 + 5x^4 + 32x^3 - 3x^3$

$- 3x^2$

$= 8x^8 + 34x^7 + 24x^6 - 14x^5 + 5x^4 + 29x^3 - 3x^2$

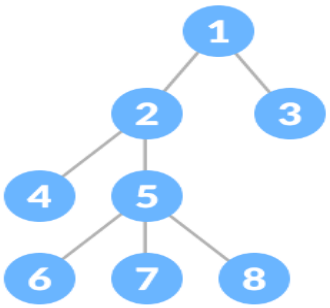
Scanned with CamScanner

Tree

A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.

Tree stores the information in hierarchical form that is represented by nodes connected by edges.

Example:



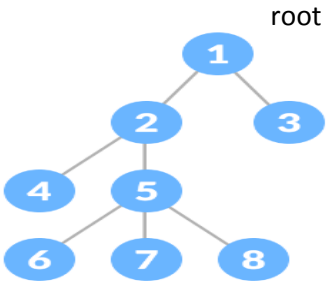
Applications:

- To implement file system for operating systems
- Evaluation of arithmetic expressions
- Efficient searching of elements, etc

A tree has following general properties:

- One node is distinguished as the root
- Every node (excluding the root) is connected by a directed edge from exactly one other node. A direction is from parent -> children

Example:



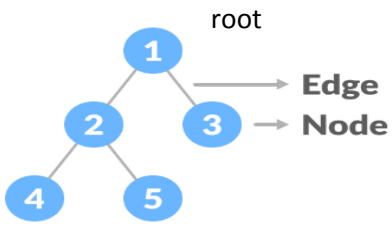
node 1 is the root node and parent of node 2 and node 3. Node 5 is the parent of node 6, 7, and 8

Tree Terminology:

Node: Each element of a tree is called a node. It specifies the information (data) and links (branches) to other data elements.

Example: There are 8 nodes in the example tree above.

Edge: It is the link between any two nodes.



Root: It is first node in the hierarchical arrangement of data items at the top of the tree.

Example: In the above tree node 1 is the root node

Parent of node n: The parent of node n is the unique node with an edge to node n. i.e. immediate predecessor of a node.

Example: Parent of node 4 in above figure is node 2.

Note: The root is the only node with no parent. Except the root, each node in the tree has one parent.

Child of node n: A child of node n is a node for which node n is the next node on the path. i.e. each immediate successor of a node.

Example: Node 4 and node 5 are child nodes of node 2.

Note: Each node may have 0 or more children.

Siblings: Nodes with the same parent are siblings.

Example: In the above figure node 4 and node 5 are siblings.

Leaf node: A leaf node is a node with no children.

Example: node 3, 4 and 5 are leaf nodes in above figure

Degree of node n: The degree of node n is the number of children node n has.

Example: degree of node 2 in above figure is 2.

Degree of a tree: The maximum degree of nodes in a given tree.

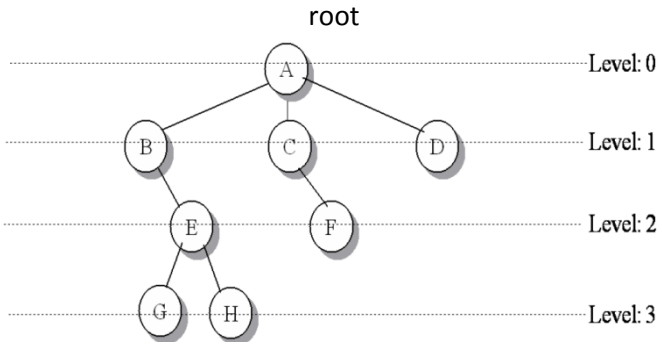
Example: degree of above tree is 2.

Depth of node n: The depth of node n is the length of the path from root to node n.

Example: root is at depth 0, depth of node 2 is 1.

Level: The tree is levelled in such a way that the root node is always at level 0. Its immediate children are at level 1, and their immediate children at level 2 and so on.

Example:



height of a node: Height of a node is the length of the longest path from the node to a leaf node.

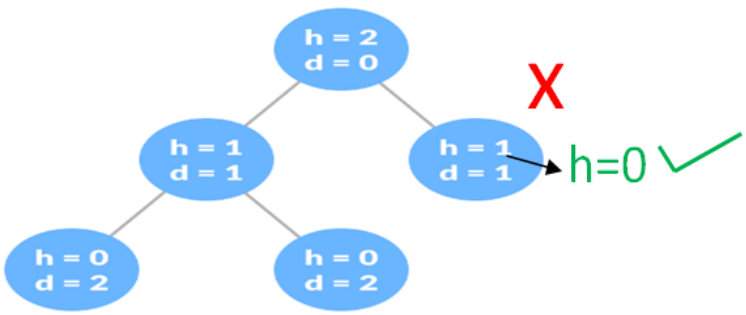
Example: height of node B in above figure is 2, height of node F is 0.

height of a tree: The height of tree is the largest depth of any node in the tree.

i.e. the height of the tree is equal to the height of the root node.

Example: height of the above tree is 3.

Example: height and depth of nodes in a tree



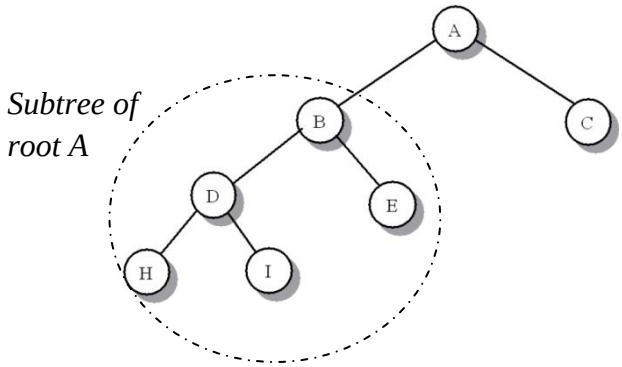
Tree Terminology cont... :

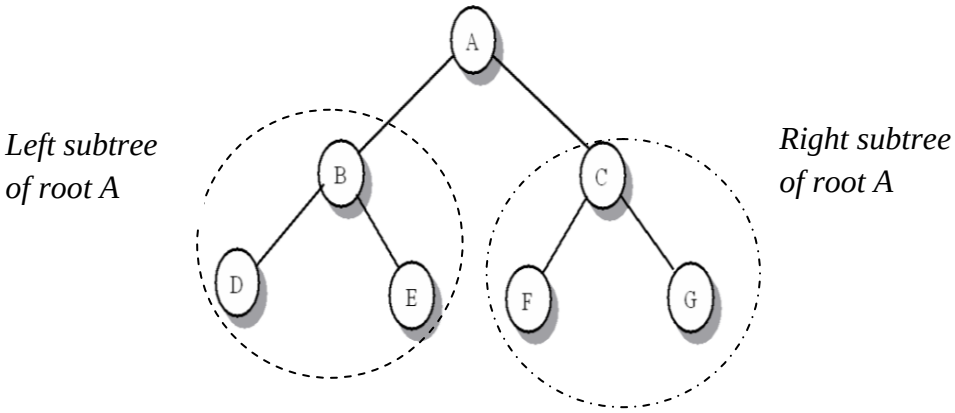
Path:

A *path* is a sequence of edges between nodes.

Subtrees of node n*:*

Subtrees of node *n* are the trees whose roots are the children of *n*.





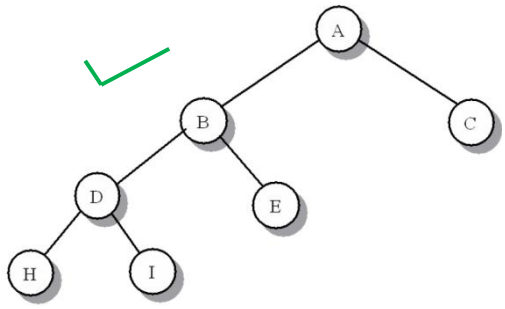
Binary Tree:

A *binary tree* is a tree in which each node has maximum two children, named as the *left child* and the *right child.*

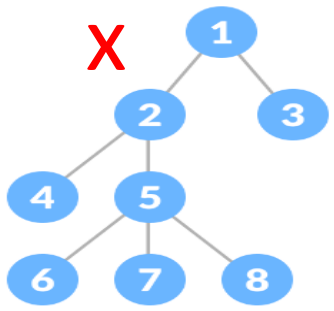
Any node in a binary tree has either 0, 1 or 2 child nodes.

Note: If a binary tree contains *m* nodes at level *l*, it contains at most *2m* nodes at level *l*+1

Example:



[A binary tree]

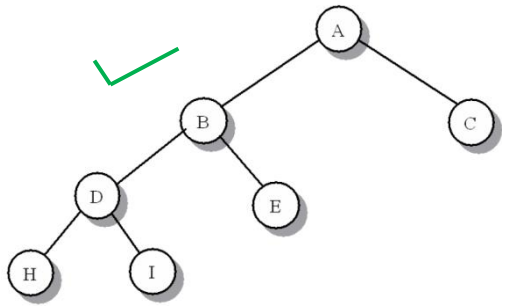


[Not a binary tree]

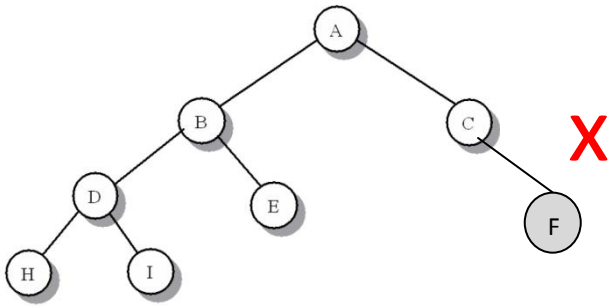
Strictly binary tree:

A strictly binary tree is a binary tree where each node has either 0 or 2 child nodes.

A strictly binary tree with *n* levels always contain *2n-1* nodes.



[A strictly binary tree]

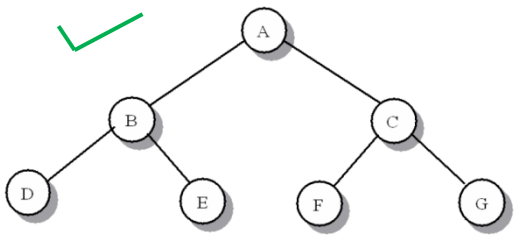


[A binary tree but not a strictly binary tree]

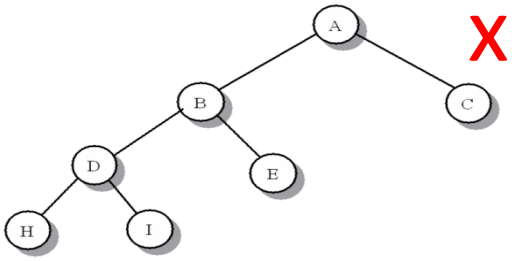
Complete binary tree:

A complete binary tree of depth *d* is the strictly binary tree all of whose leaf nodes are at level *d*.

A complete binary tree of depth *d* contains exactly 2^l nodes at each level.



[complete binary tree of depth 2]

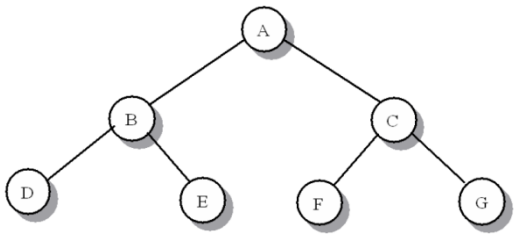


[not a complete binary tree, its a strict binary tree]

Some properties:

1. A complete binary tree of depth d contains exactly 2^l nodes at each level
2. In a complete binary tree, the number of nodes at depth d is $2^{d.}$
3. In a complete binary tree the height of the tree is $\log_2(n+1)-1$.

Explanations on properties:



[A complete binary tree of depth 2]

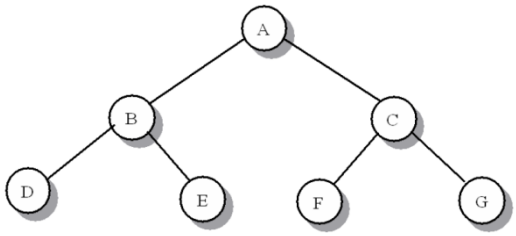
1. A complete binary tree of depth d contains exactly 2^l nodes at each level
   Example: depth of the tree=2
   At level 0, 2^0 nodes will be present. i.e. 1 nodes // node A
   At level 1, 2^1 nodes will be present. i.e. 2 nodes //node B, C
   At level 2, 2^2 nodes will be present. i.e. 4 nodes //node D, E, F, G

2. In a complete binary tree, the number of nodes at depth d is $2^{d.}$

   Example:
   Number of nodes at depth 0= 20 = 1. //node A
   Number of nodes at depth 1= 21 = 2. //node B, C
   Number of nodes at depth 2= 22 = 4. //Node D, E,F,G

3. In a complete binary tree the height of the tree is $\log_2(n+1)-1$, where n is the number of nodes.

   Example-1: If number of nodes in a tree=7
   So height of the tree= $\log_2(7+1)-1$
   $= \log_2(8)-1$
   $=3-1$
   $=2$
   height of the tree (with 7 nodes)=2



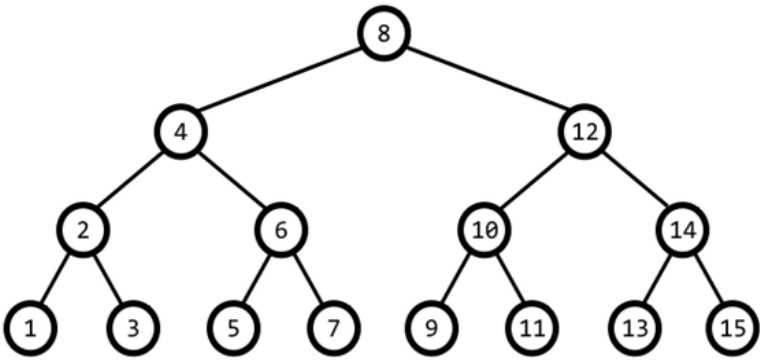
Example-2:

If number of nodes in tree = 15

So height of the tree= $\log_2(15+1)-1$

$$= \log_2(16)-1$$
$$=4-1$$
$$=3$$

Example: height of the tree (with 15 nodes)=3



Memory representation of Binary tree:

A binary tree data structure is represented using two methods:

1. Array Representation
2. Linked List Representation

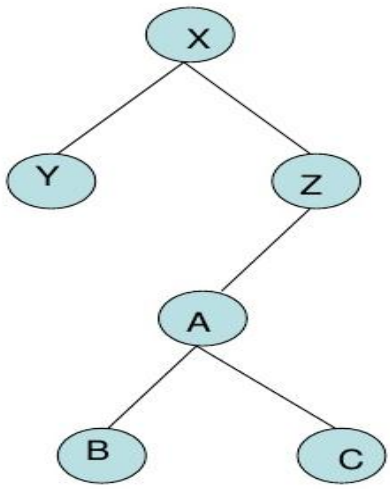
1. Array representation of Binary tree

This representation uses a single linear array ***tree*** following the below rules:

(i) The root of the tree is stored in ***tree[0].***

(ii) If a node occupies ***tree[i],*** then its left child is stored in ***tree[2*i+1],*** and its right child is stored in ***tree[2*i+2].***

Example:



	0	1	2	3	4	5	6	7	8	9	10	11	12
Tree:	X	Y	Z			A						B	C

Root is node X stored in tree [0], i.e. i=0
its left child will be stored in tree[2*i+1]=tree[2*0+1]=tree[1].
Therefore, the left child, i.e. node Y is stored in tree[1] location in the array.

The right child of node X will be stored in tree[2*i+2] =tree[2*0+2]=tree[2]
Therefore, the right child, i.e. node Z is stored in tree[2] location in the array.

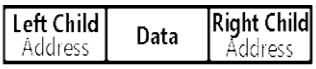
Left child of node Z at tree[2] location will be stored in tree[2*i+1]=tree[2*2+1]=tree[5] i.e. node A will be stored in tree [5]
And so on.

2. Linked list representation:

A node is divided into three fields:

- Data
- left pointer filed used to store address of left child,
- right pointer filed used to store address of right child

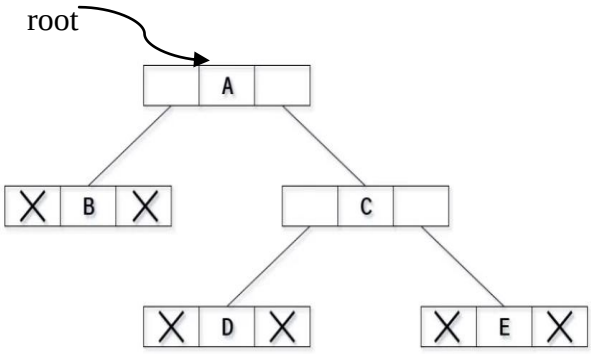
Example of a node:



Structure of a node:

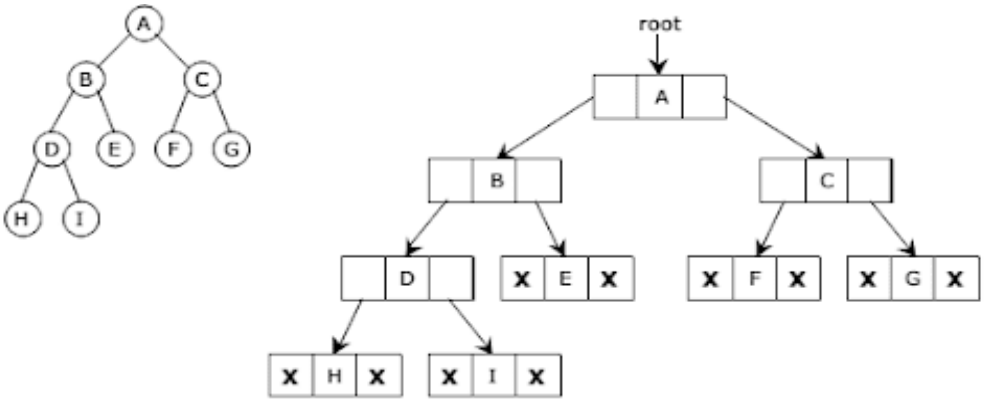
```
struct node
     {
     int data;
     struct node *left, *right;
     }*root=NULL;
```

Exmple-1:



If any of left or right child is empty then the pointers contain NULL value

Example-2 :



[Linked list representation of a binary tree]

// Cross symbol indicates NULL

Traversal in Binary Tree:

Traversal is the process to visit all the nodes of a tree at least once.

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

There are three ways used to traverse a tree –

A. Pre-order Traversal
B. In-order Traversal
C. Post-order Traversal

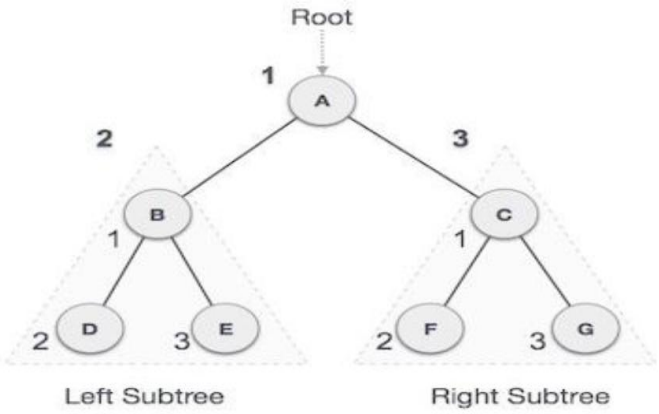
A. Pre-order Traversal:

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree using the pre-order traversal.

Steps:

1. Visit root
2. Traverse left sub tree in pre-order
3. Traverse right sub tree in pre-order

Example:



Output: The output of pre-order traversal for above tree will be −

A B D E C F G

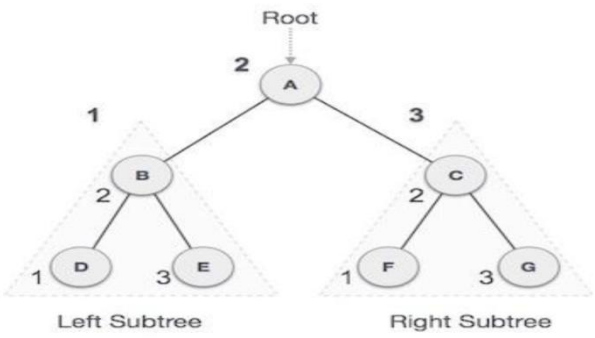
B. **In-order Traversal:**

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree is processed using the in-order traversal.

Steps:

1. Traverse left sub tree in in-order
2. Visit root
3. Traverse right sub tree in in-order

Example:



Output: The output of in-order traversal of the above tree will be −

D B E A F C G

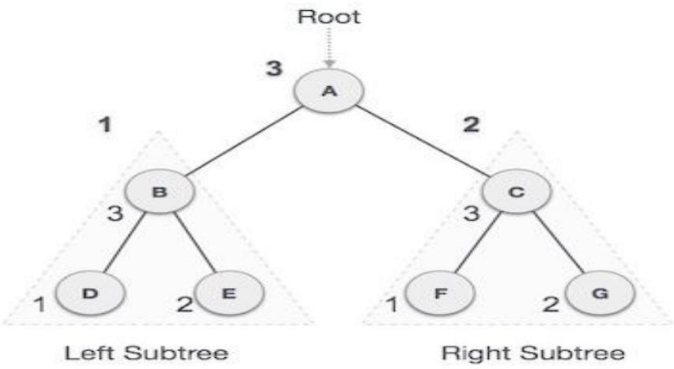
C. **Post-order traversal:**

In this traversal method, first the left subtree is traversed, then the right subtree and finally the root node using post order traversal.

Steps:

1. Traverse left sub tree in post-order
2. Traverse right sub tree in post-order
3. Visit root

Example:

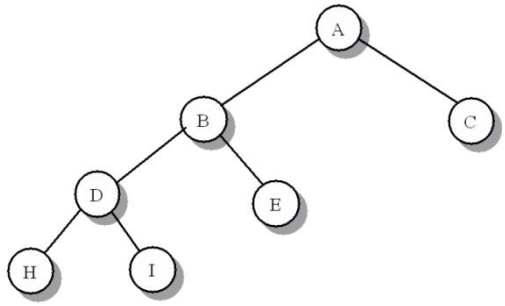


Output: The output of post order traversal of above tree will be –

D E B F G C A

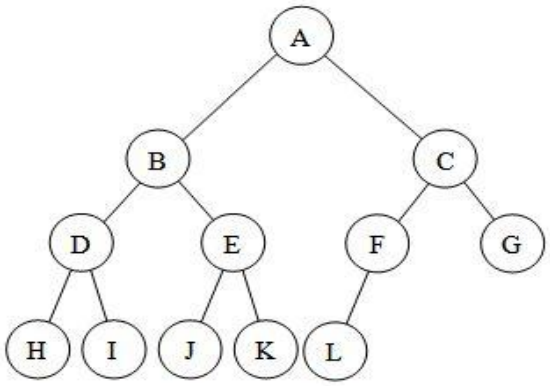
Examples:

Example-1



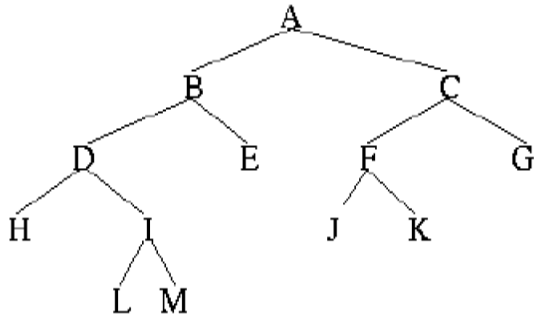
Pre-order: A, B, D, H, I, E, C
In-order: H, D, I, B, E, A, C
Post-order: H, I, D, E, B, C, A

Example-2



Pre-order: A, B, D, H, I, E, J, K, C, F, L, G
In-order: H, D, I, B, J, E, K, A, L, F, C, G
Post-order: H, I, D, J, K, E, B, L, F, G, C, A

Example-3:



Pre-order: A, B, D, H, I, L, M, E, C, F, J, K, G
In-order: H, D, L, I, M, B, E, A, J, F, K, C, G
Post-order: H, L, M, I, D, E, B, J, K, F, G, C, A

Traversal functions:

Pr-order traversal function: (RT, L, R)

```
void preorder(struct node *ptr)
{
      if(ptr!=NULL)
      {
            printf("%d\t",ptr->data);   //print root
            preorder(ptr->left);  //process left subtree recursively
            preorder(ptr->right); //process right subtree recursively
      }
}
```

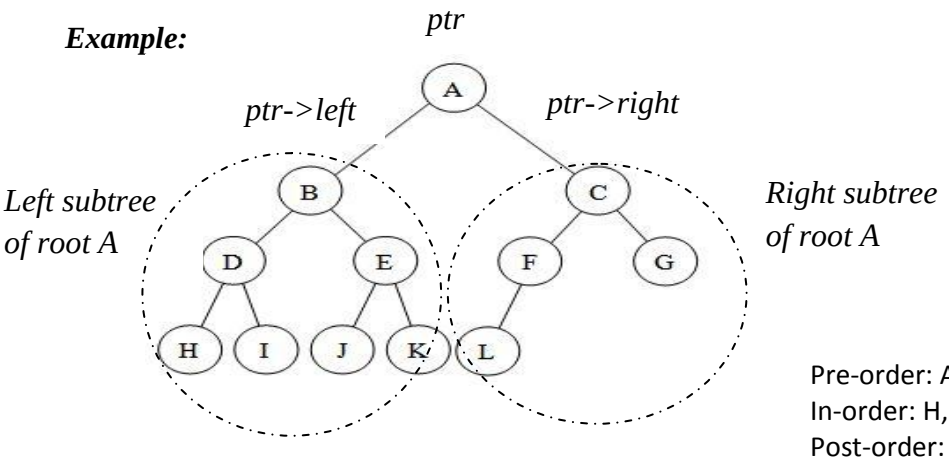
In-order traversal function: (L, RT, R)

```
void inorder(struct node *ptr)
{
      if(ptr!=NULL)
      {
            inorder(ptr->left);   //process left subtree recursively
            printf("%d\t",ptr->data);   //print root
            inorder(ptr->right);  //process right subtree recursively
      }
}
```

Post-order traversal function: (L, R, RT)

```
void postorder(struct node *ptr)
{
      if(ptr!=NULL)
      {

            postorder(ptr->left); //process left subtree recursively
            postorder(ptr->right);//process right subtree recursively
            printf("%d\t",ptr->data);   //print root
      }
}
```

Example:



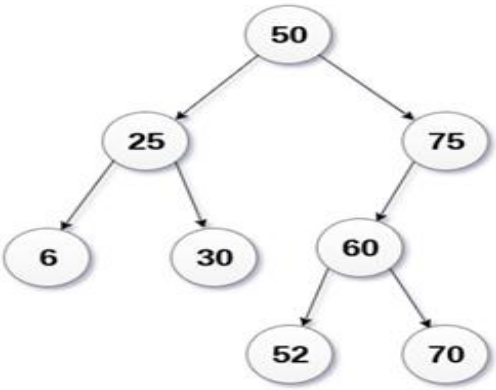
Pre-order: A, B, D, H, I, E, J, K, C, F, L, G
In-order: H, D, I, B, J, E, K, A, L, F, C, G
Post-order: H, I, D, J, K, E, B, L, F, G, C, A

Binary Search Tree (BST)

A Binary Search Tree (BST) is a binary tree in which all the nodes in the tree follow the below properties:

- All nodes of left sub-tree are less than the root node.
- All nodes of right sub-tree are greater than the root node.
- Both sub-trees of each node are also BSTs. i.e. they have the above two properties

Example:



Insertion in BST:

- A new node is always inserted at leaf node positions following the BST properties.
- In order to insert an element, the search starts from root of the tree.
- The new element is compared with the root node:
  - If it is less than the root node, then left sub-tree is recursively searched,
  - If it is greater than the root node, then right sub-tree is recursively searched.
- After reaching the appropriate leaf node position, the new node is inserted.

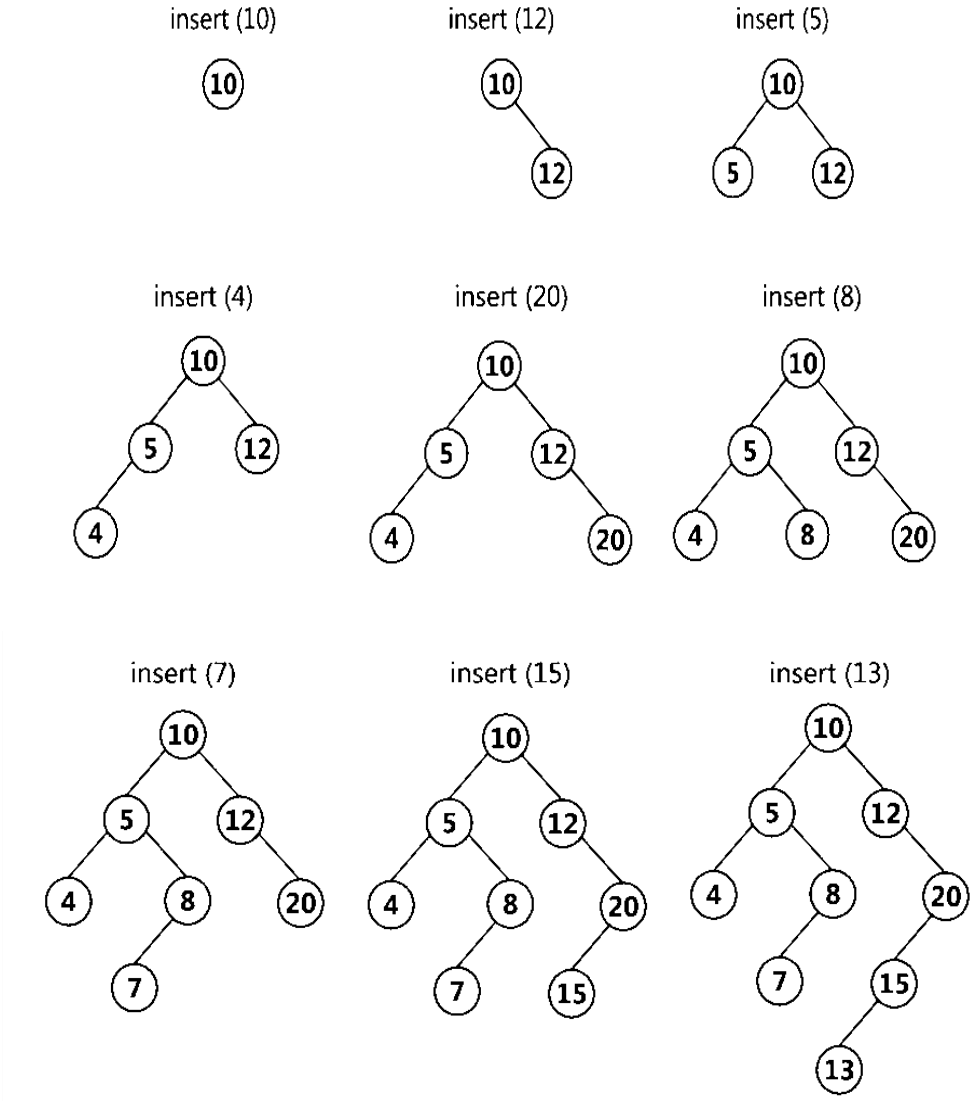
Example-1:

```
      100                                  100
     /   \        Insert 40               /    \
   20     500    --------->            20      500
  /  \                                 /  \
 10   30                              10   30
                                            \
                                             40
```

Example-2: Construct a BST by inserting the following sequence of numbers:

10, 12, 5, 4, 20, 8, 7, 15 and 13

Answer:

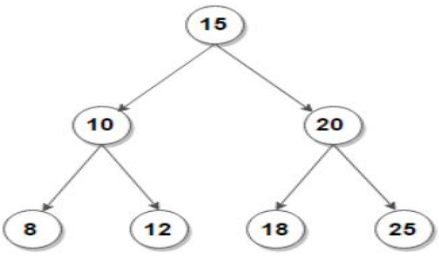


Example-3: Construct a BST by inserting the following sequence of numbers:

15, 20, 18, 10, 12, 25, 8

Answer: Derive all steps by your own

Final step: output of example-3

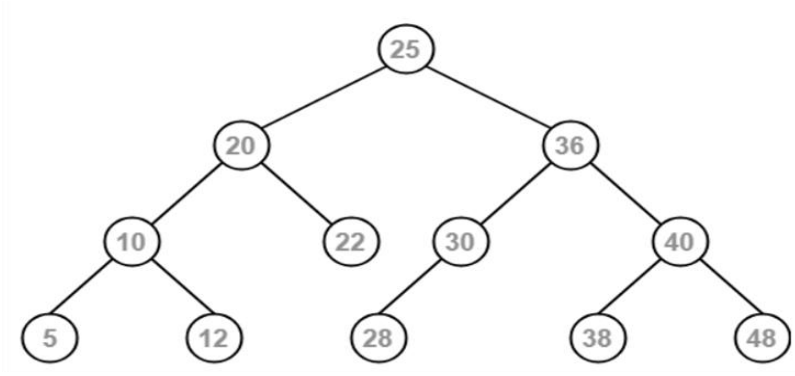


Example-4: Construct a BST step wise by inserting the following sequence of numbers:

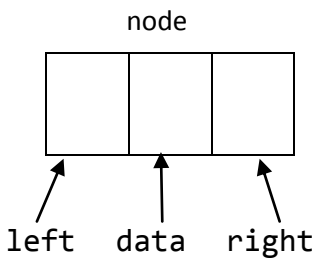
25, 36, 30, 40, 20, 28, 22, 10, 48, 12, 38, 5

Answer: Derive all steps by your own

Final step: output of example-4



Representation of a node in BST:




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

}*root=NULL;
```

Insert function:

```
struct node* insert(struct node *temp, int ele)
{
      if(temp==NULL)
            {
            temp =(struct node*)malloc(sizeof(struct node));
            temp ->data=ele;
            temp ->left=NULL;
            temp ->right=NULL;
            }
      else
            {
            if(ele < temp->data)
                  temp->left=insert(temp->left,ele);
            else
                  {
                  if(ele > temp->data)
                        temp->right=insert(temp->right,ele);
                  }
            }
      return temp;

}
```

Program:

// Menu driven program for Insert and pre-order traversal in BST:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
      int data;
      struct node *left,*right;
}*root=NULL;
struct node *insert(struct node *,int);
void preorder(struct node*);

main()
{
      int ch,x;
      while(1)
      {
            printf("\nMenu: \n1: insert\n2: pre-order traversal\n3:
exit\n");
            printf("\n Enter your choice");
            scanf("%d",&ch);
            switch(ch)
            {
                  case(1):
                        printf("enter the data to insert:");
                        scanf("%d",&x);
                        root=insert(root,x);
                        break;
                  case(2):
                        preorder(root);
                        break;
                  case(3):
                        exit(0);
                  default:
                        printf("Invalid option");
            }
      }
}

struct node *insert(struct node *temp,int ele)
      {
      ……
      ……
      //write insert function definition written above
      }


void preorder(struct node *ptr)
      {
      ……
      ……
      //write preorder traversal function definition written above
      }
```

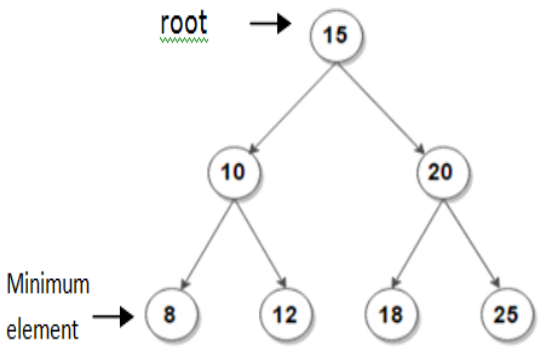
Searching in BST:

- In order to search a given value in BST, first it is compare with the root node-
  - if it matches the root value, then root is returned.
  - if it is smaller than the root value, the left sub-tree is searched recursively.
  - if it is greater than the root value, the right sub-tree is searched recursively.

```
//searching function
struct node* search(struct node* temp,int val)
{
	struct node *p;
	p=temp;
	while( p!=NULL && p->data!=val)
	{
		if(p->data > val)
			p=p->left;
		else
		{
			if(p->data < val)
				p=p->right;
		}
	}
	if(p==NULL)
		printf("element not found");
	else
		return p;

}
```

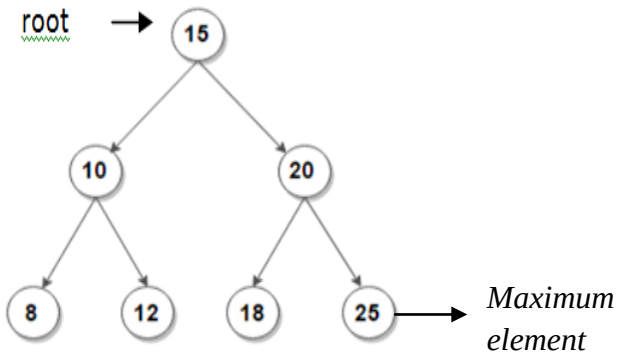
Find minimum element in BST

```
struct node* findmin(struct node* temp)
{
	if(temp==NULL)
		return NULL;
	else
	{
		if(temp->left==NULL)
			return temp;
		else
			return(findmin(temp->left));
	}
}
```



Find maximum element in BST:

```
struct node* findmax(struct node* temp)
      {
            ……….
            ……
            //write the function by your own
      }
```



Deletion in BST:

- Deleting a node from BST includes following three cases:

  1) Node to be deleted is a leaf node
  2) Node to be deleted has only one child
  3) Node to be deleted has two children

Case-1: ***Node to be deleted is a leaf node:***

- Simply remove the node from the tree.

Example:

```
        50                                  50
      /    \          delete(20)          /   \
     30     70        --------->        30     70
    /  \   /  \                          \    /  \
  20   40 60   80                        40  60   80
```

Case-2: ***Node to be deleted has only one child:***

- Copy the child to the node and delete the child node

Example:

```
     50                              50
  /     \          delete(30)      /   \
 30      70        --------->    40     70
   \    /  \                            /  \
   40  60   80                        60   80
```

Case-3: Node to be deleted has two children:

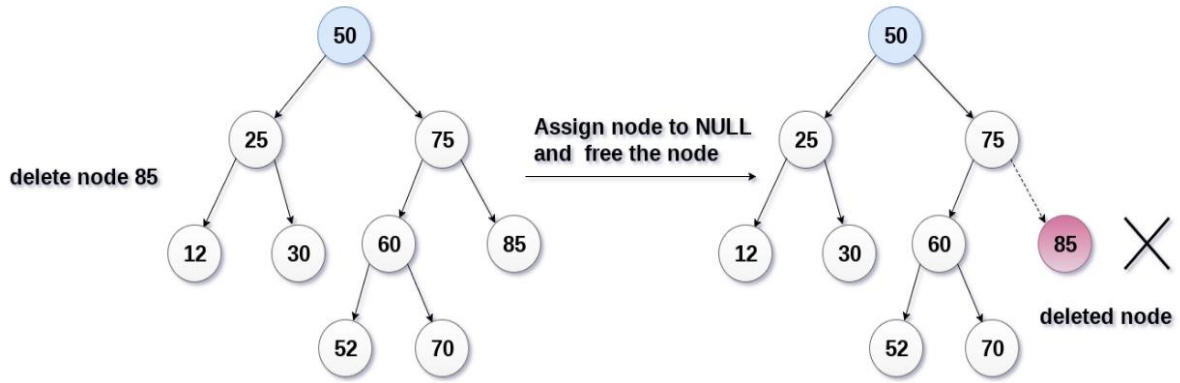
- Find in-order successor of the node.
- Copy contents of the in-order successor to the node
- Delete the in-order successor.

Example-1:

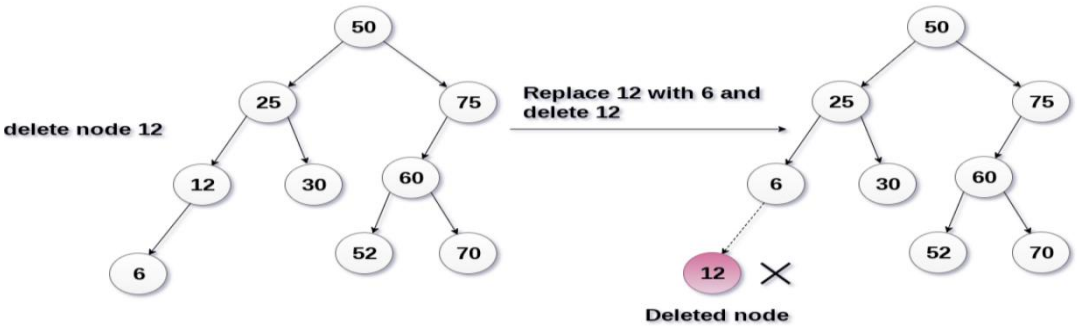
```
     50                              60
  /     \          delete(50)      /   \
 40      70        --------->    40    70
        /  \                            \
       60   80                           80
```

<u>Other examples on Deletion in BST:</u>

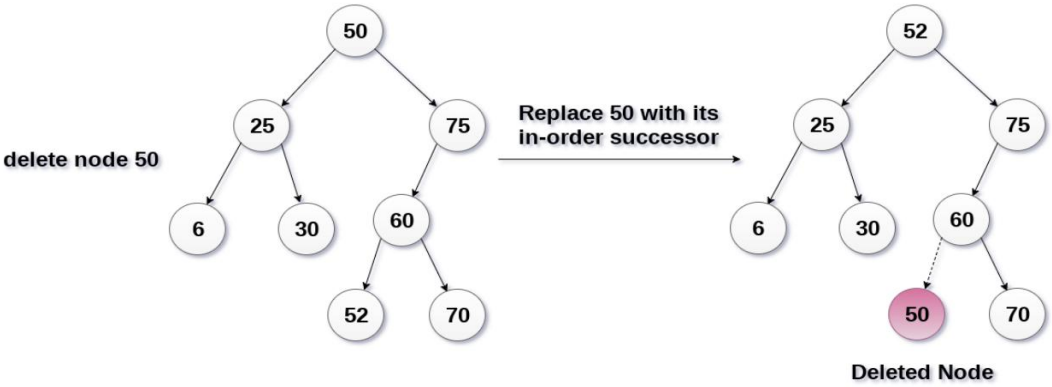
Example-1



Example-2



Example-3:



//Deletion function for BST

```
struct node* delete(struct node* temp, int val)
{

    if (temp == NULL)
      return temp;

    /* If the value to be deleted is smaller than the root's value,
     then it lies in left subtree */
    if (val < temp->data)
        temp->left = delete(temp->left, val);

    /* If the value to be deleted is greater than the root's value,
     then it lies in right subtree */
    else if (val > temp->data)
        temp->right = delete(temp->right, val);

    /* if the value is same as root's value, then this is the node
     to be deleted */
    else
    {
        // node with only one child or no child
        if (temp->left == NULL)
        {
            struct node *p = temp->right;
            free(temp);
```

```
            return p;
        }
        else if (temp->right == NULL)
        {
            struct node *p = temp->left;
            free(temp);
            return p;
        }

        /* node with two children: Get the inorder successor- smallest
         in the right subtree/*
        struct node* p = findmin(temp->right);

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

        // Delete the inorder successor
        temp->right = delete(temp->right, p->data);
    }
    return temp;
}
```

Threaded Binary Tree

- When a binary tree is represented using linked list representation, the reference part (left and right) of the node which doesn't have a child is filled with a NULL pointer.

- There are a number of NULL pointers than actual pointers pointing to left or right child.

- If there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL.

- This NULL pointer does not play any role except indicating that there is no link (no child).

- Threaded Binary Tree is a binary tree which makes use of NULL pointers ***to improve its traversal process.*** The NULL pointers are replaced by references of other nodes in the tree. These extra references are called as threads.

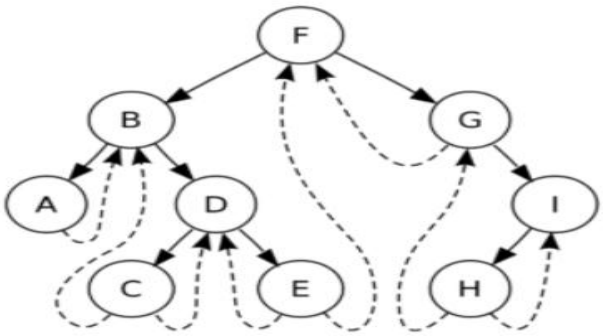
What is a Threaded Binary Tree ? :

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***.

In-order predecessor:
- The node coming before the current node in the in-order traversal of a binary tree.

In-order successor:
- The node coming next to the current node in the in-order traversal of a binary tree.

Example:



In-order traversal: A B C D E F G H I

Leaf nodes: A C E H
Node with one child (left child): I
Node with one child (right child): G

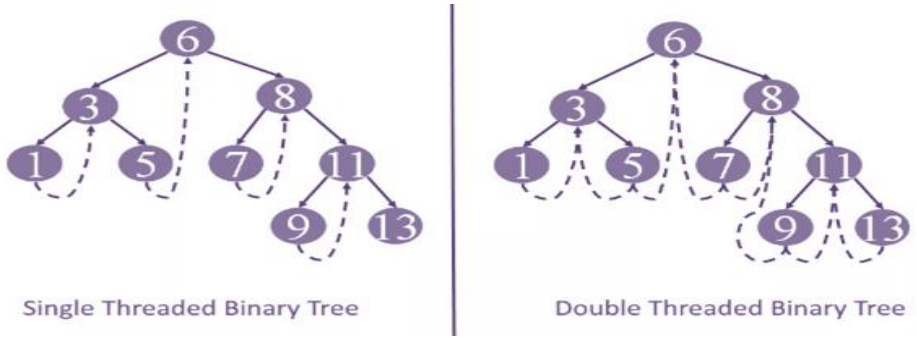
Nodes with both child nodes empty (leaf nodes):
Node A:
left: no predecessor available (NULL),
right: will point to B
Node C:
Left: will point to B,
Right: will point to D
And so on for E , H

Nodes with one child empty:
Node I:
right: no successor available (NULL)
Node G:
Left: will point to F

Based on the pointers threaded in a node there are two types of threaded binary trees:

(1) Single Threaded Trees:
For every node only one of the pointers is threaded i.e. either the left node is made to point the in-order predecessor **OR** the right node is made to point to the in-order successor of the tree.

(2) Double Threaded Trees:
For every node both of the pointers are threaded i.e. the left node is made to point the in-order predecessor **AND** the right node is made to point to the in-order successor of the tree.

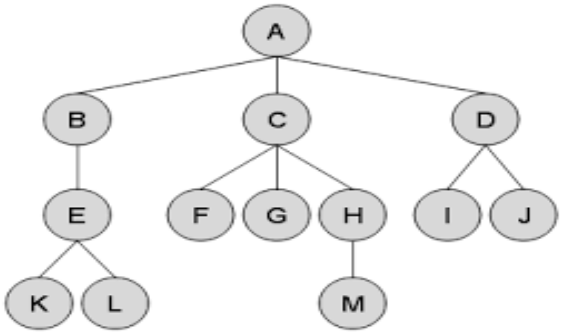


General Tree:

- General tree is a tree in which each node can have either zero or many child nodes. There is no limitation on the degree of a node. The topmost node of a general tree is called the root node.

General trees are used to model applications such as file systems.

Example:



Note: - A binary tree is a specialized case of a general tree where a node can have 0, 1, or 2 child nodes.

AVL Tree

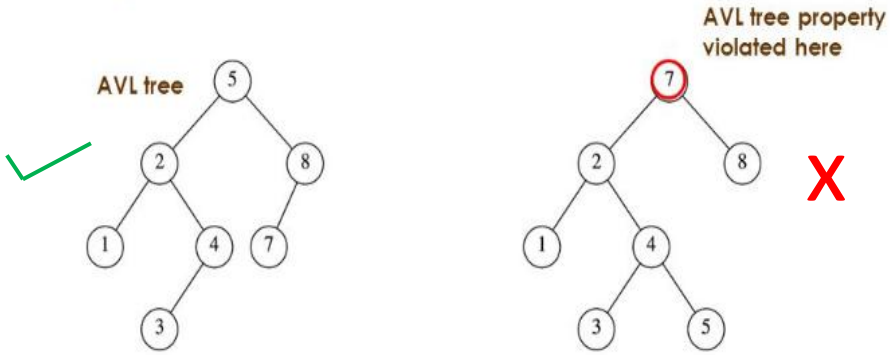
AVL tree is a binary search tree in which,

- the difference of heights of left and right sub-trees of any node is ***at most 1*** and
- the left and right sub trees are also AVL trees.

The technique of balancing the height of binary trees was developed by **A**delson, **V**elskii, and **L**andi and hence given the short form as **AVL** tree or Balanced Binary Tree.

Since AVL trees are height balance trees, operations like insertion and deletion have low time complexity.

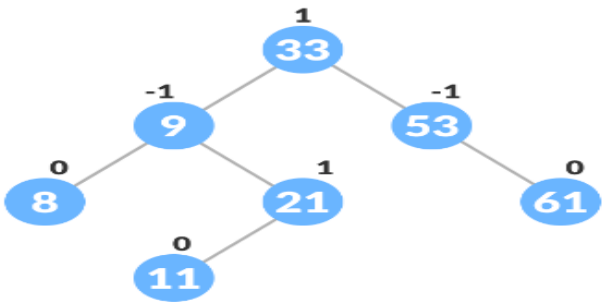
Example:



BalanceFactor = `height(left-sutree) - height(right-sutree)`

Height of a node= maximum number of edges to a leaf node
Height of an empty sub tree=-1
Height of one node=0

Example-1:



Balance factor of node 33= height of its left sub tree- height of its right sub tree= 2-1=1

Balance factor of node 9=0-1=-1

Balance factor of node 8=0, and so on

If balance factor of any node is:

1: the left sub-tree is one level higher than the right sub-tree.

0: the left sub-tree and right sub-tree contain equal height.

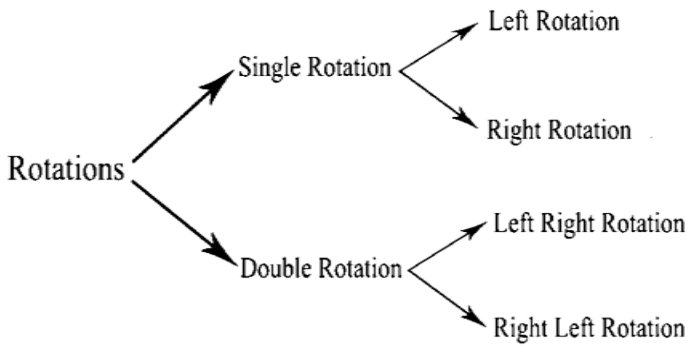
-1: the left sub-tree is one level lower than the right sub-tree.

AVL Tree Rotations:

In AVL tree, after performing operations like ***insertion and deletion*** the ***balance factors*** of every node in the tree are needed to be checked.

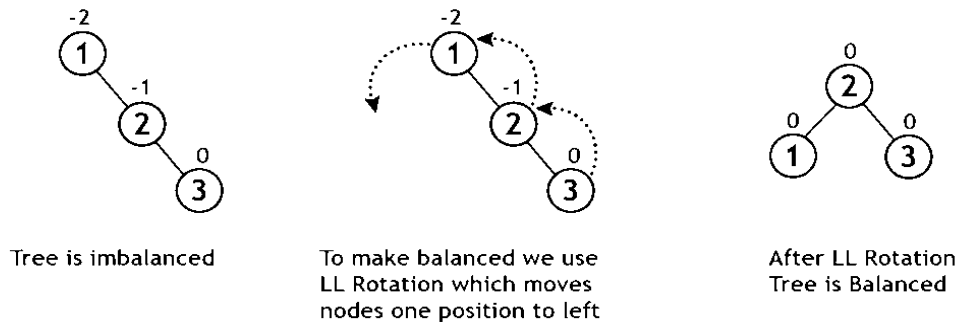
If every node satisfies the balance factor condition then the operation is concluded.

Otherwise whenever the tree becomes imbalanced due to any operation, rotation operations are applied to make the tree balanced.



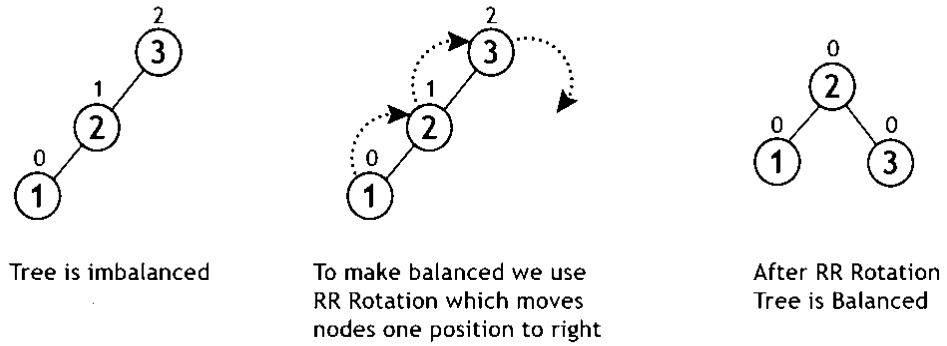
1. **LL Rotation:** In LL Rotation, every node moves one position to left from the current position.

   Example: insert 1, 2, 3



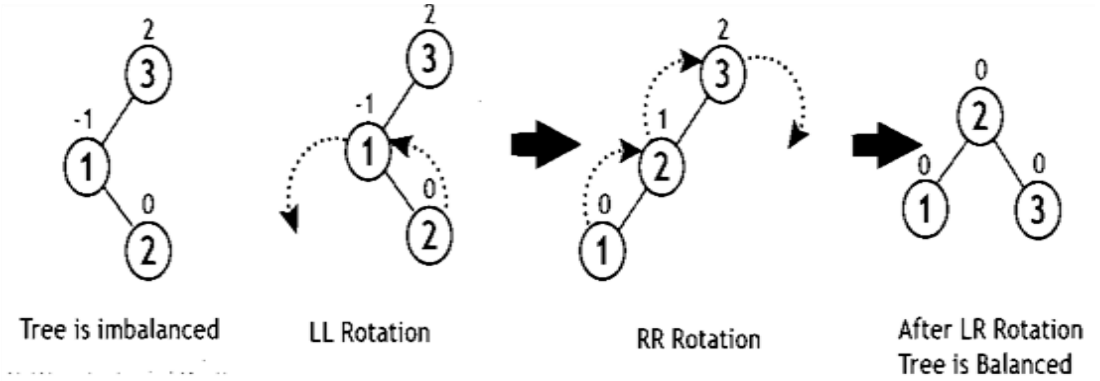
2. **RR Rotation:** In RR Rotation, every node moves one position to right from the current position.

   Example: insert 3, 2, 1

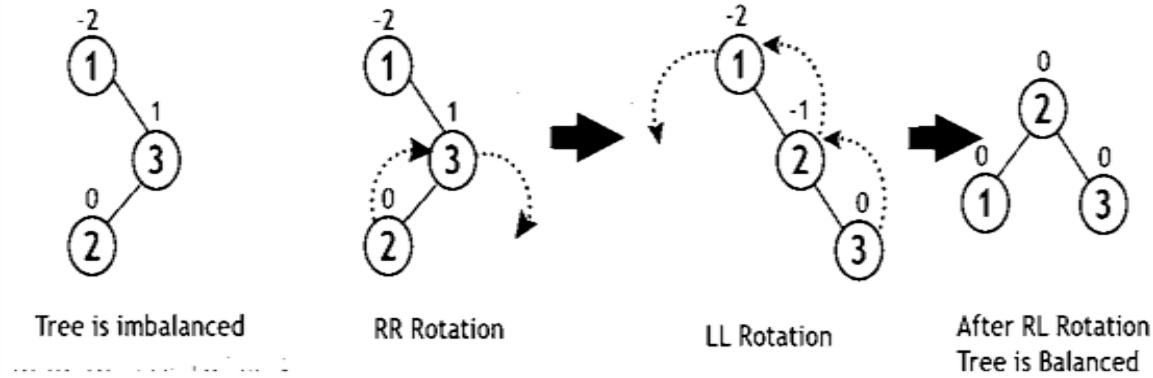


3. **LR Rotation:**
   The LR Rotation is a sequence of single left rotation followed by a single right rotation.
   In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.

   Example**:** insert 3, 1, 2



4. **RL Rotations:** 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.
   Example: insert 1, 3, 2



<u>Insertion in AVL tree:</u>

Steps:

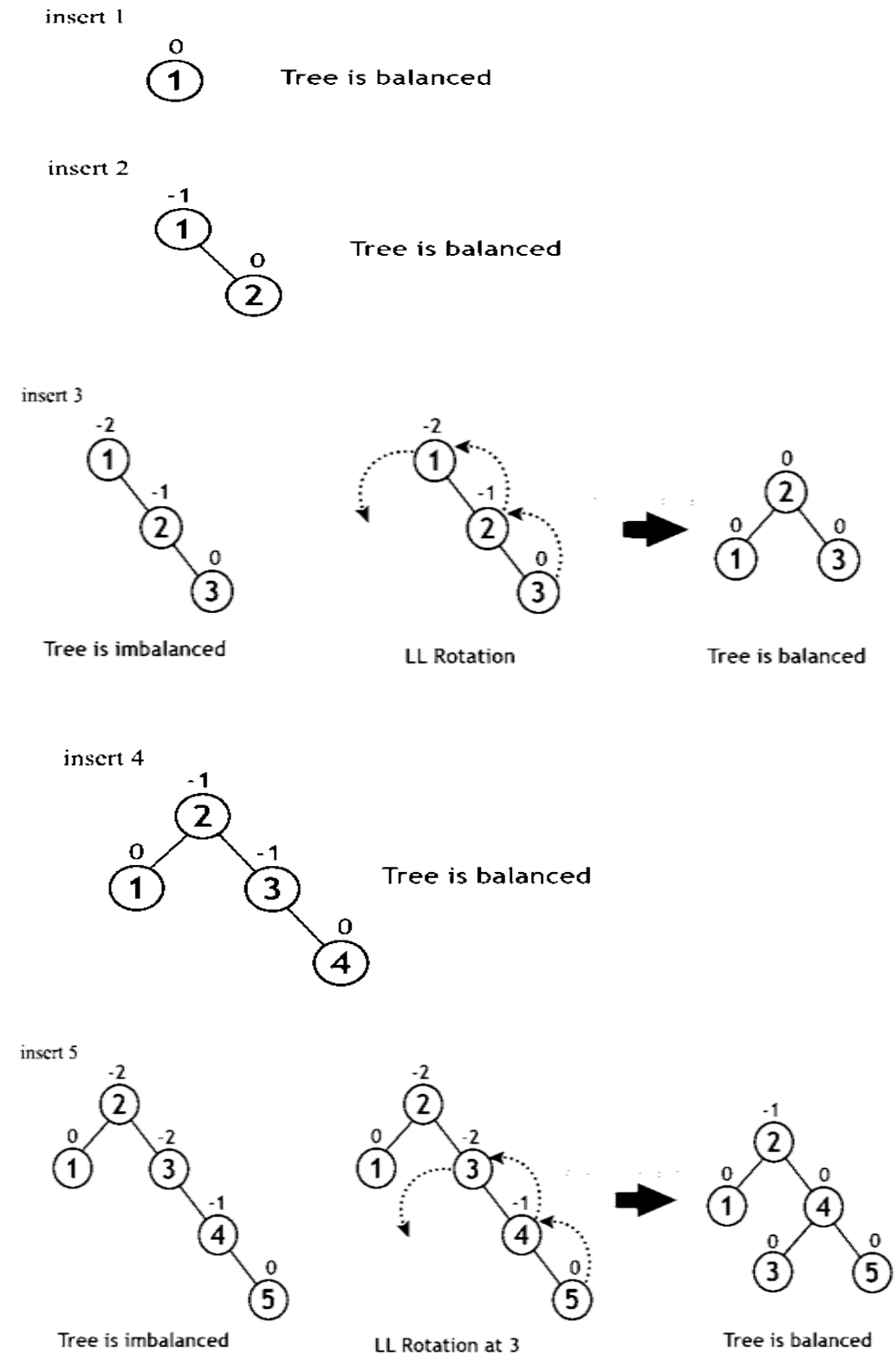
Step-1: Insert the new element into the tree using BST insertion concept.

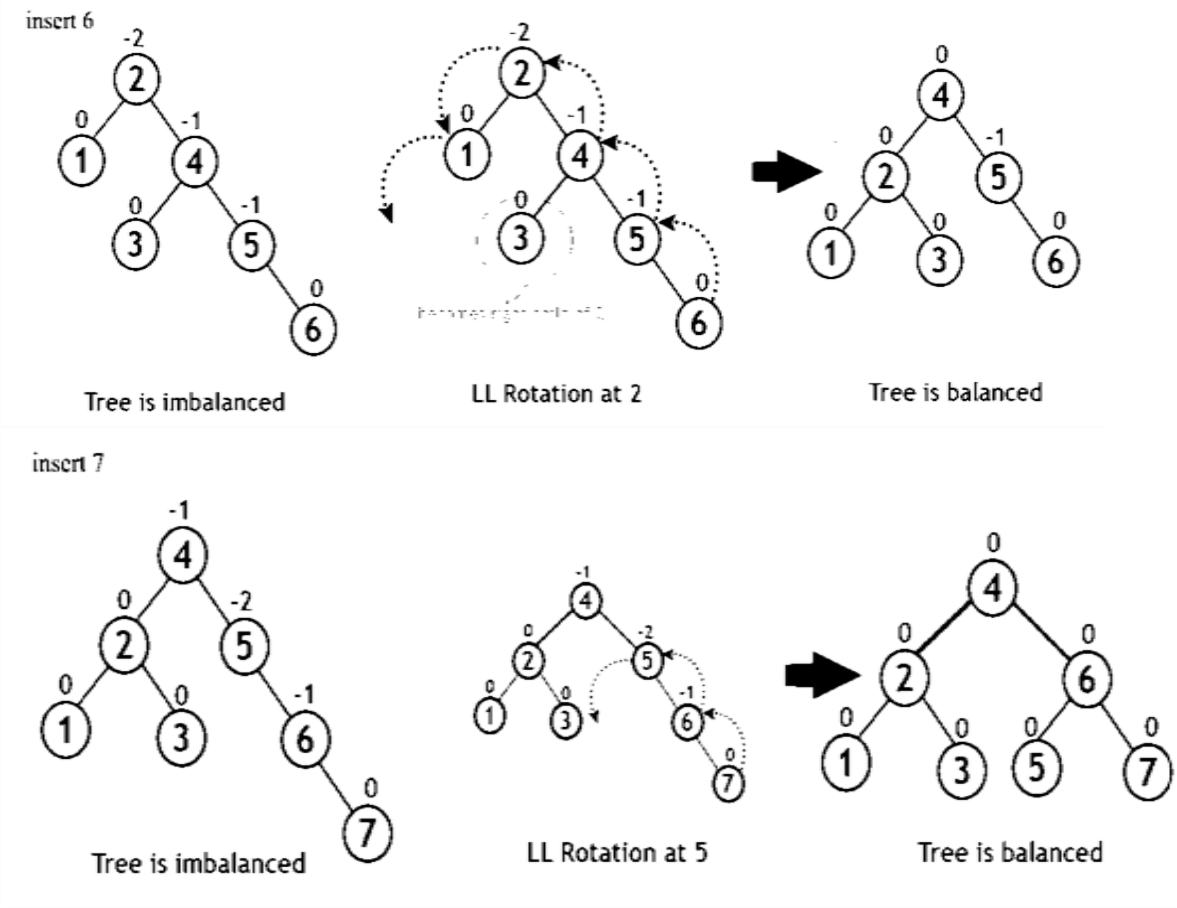
Step-2: After insertion, check the Balance Factor of every node.

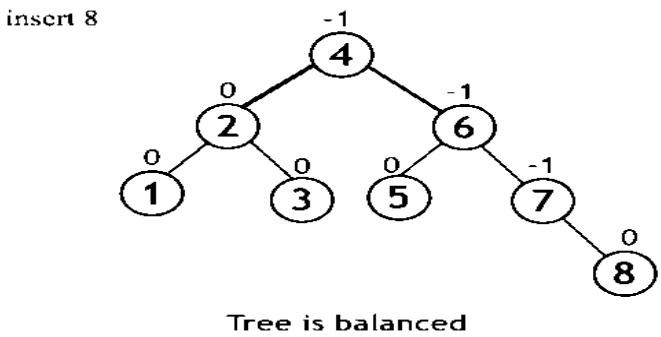
Step-3: If the Balance Factor of every node is 0 or 1 or -1 then go for next Operation.

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. Perform suitable Rotation to make it balanced and go for next operation.

Example-1:
Construct an AVL tree by inserting 1, 2, 3, 4, 5, 6, 7, 8

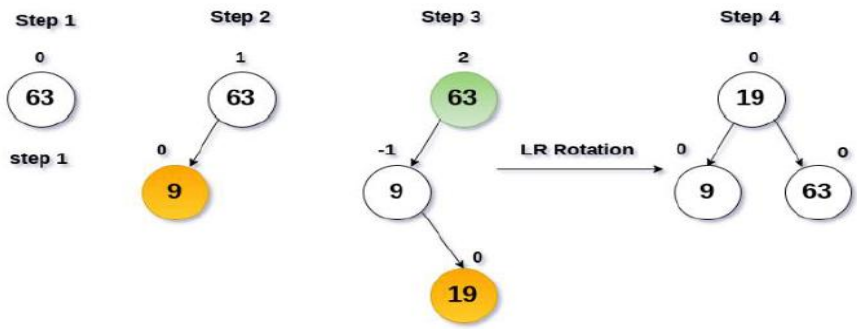


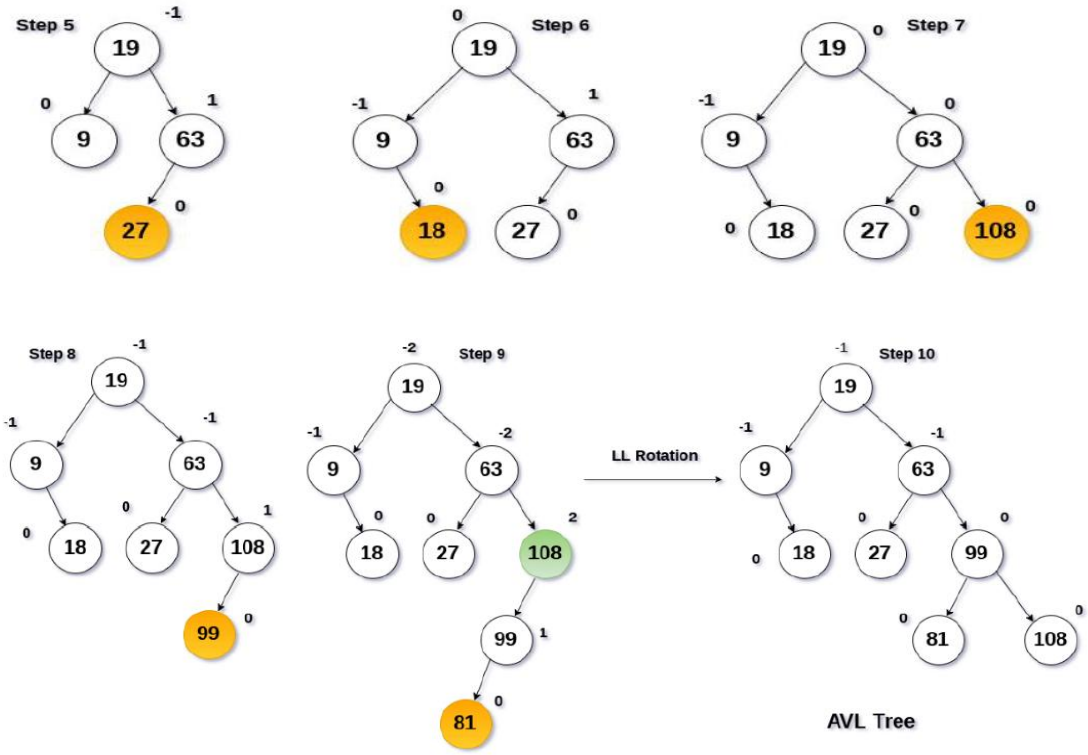




Example-2: Construct an AVL tree by inserting the following elements in the given order: 63, 9, 19, 27, 18, 108, 99, 81

Ans:

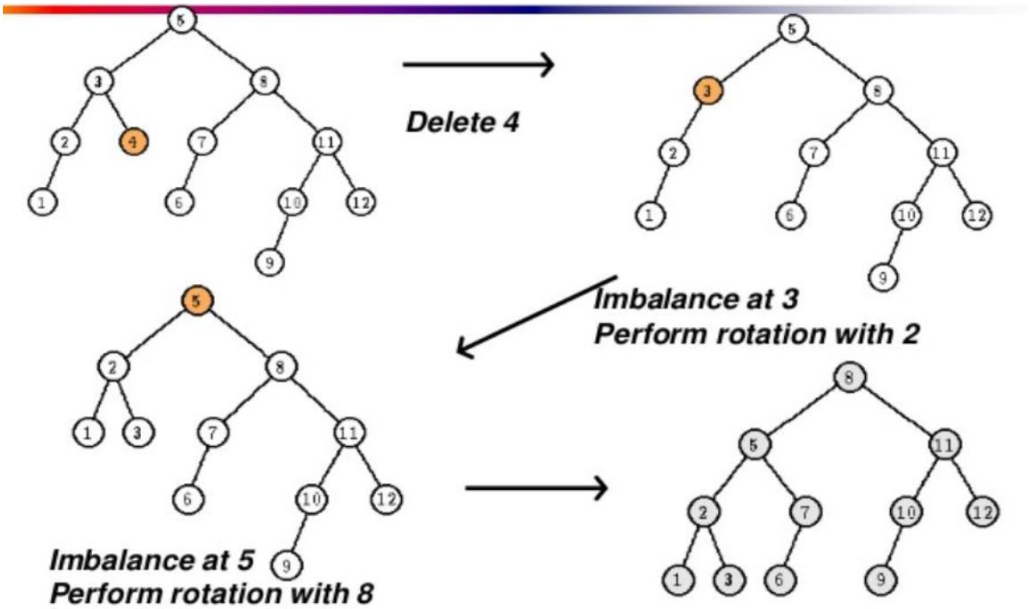




Deletion Operation in AVL Tree:

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, the Balance Factor conditions are needed to be checked. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

Example:



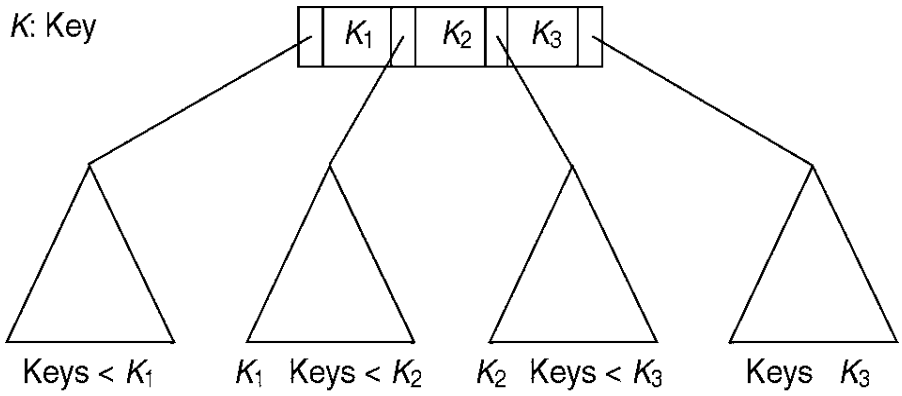
M- Way Search Tree:

The m-way search trees are multi-way trees which are generalised versions of binary trees where each node contains multiple keys (or data values).

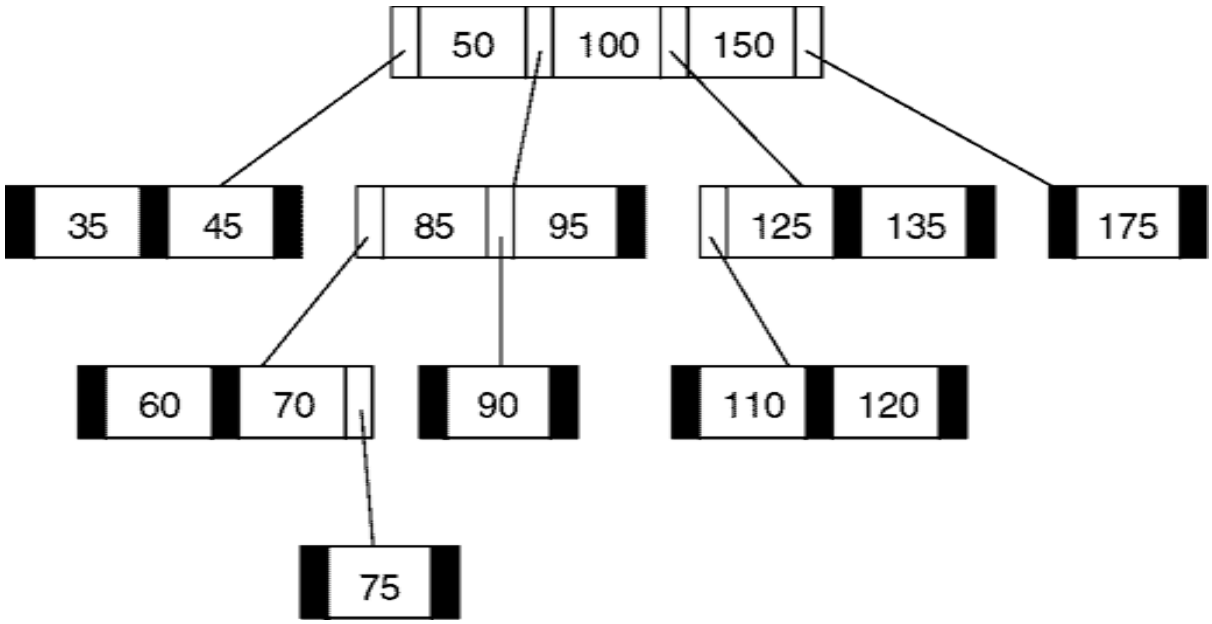
M-way trees have the following properties:

1. Each node has 0 .. m subtrees
2. A node with k < m subtrees, contains k-1 keys.
3. The key values of the first subtree are all less than the key value.
4. The data entries are ordered.
5. All subtrees are m-way trees.

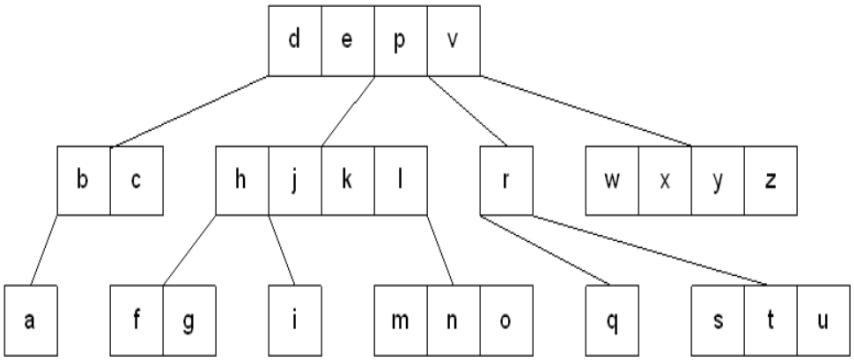
Example:



Example-1: A four way tree: i.e. multiway tree of order 4



Example-2: Multi-way tree of order 5



B –Tree:

- B-tree is a specialized m-way tree.

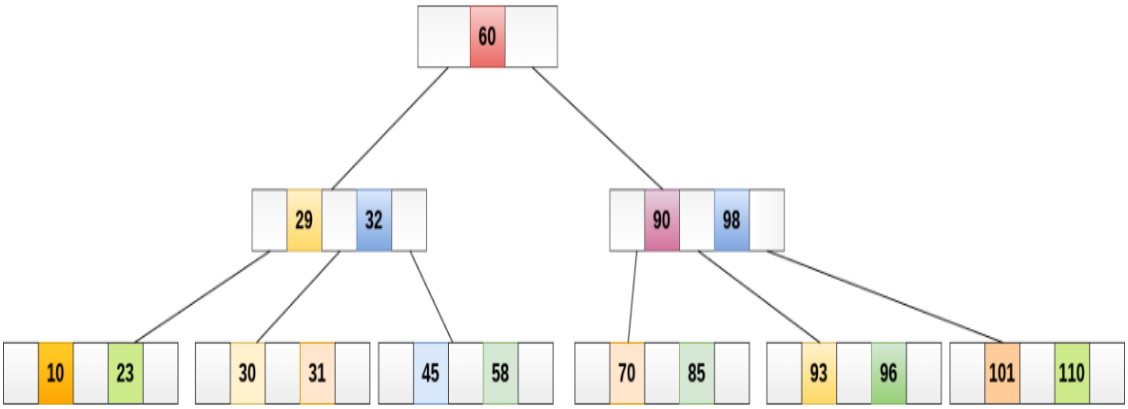
- A B-tree of order m can have at most m-1 keys and m children.

- A B-tree of order m contains all the properties of an M-way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

Example: A B-tree of order 4

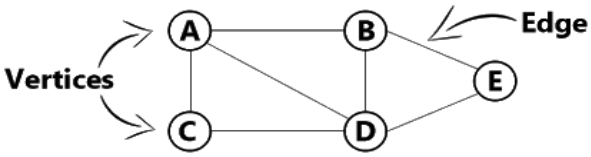


Graph

- Like trees, graph is also a non-linear data structure

- A **graph** G= (V, E) consists of 2 sets:
  - A set V, of **vertices** (or nodes) and
  - A set E, called set of **edges** (or arcs)

Example:



Graph G1 above contains 5 vertices and 7 edges:

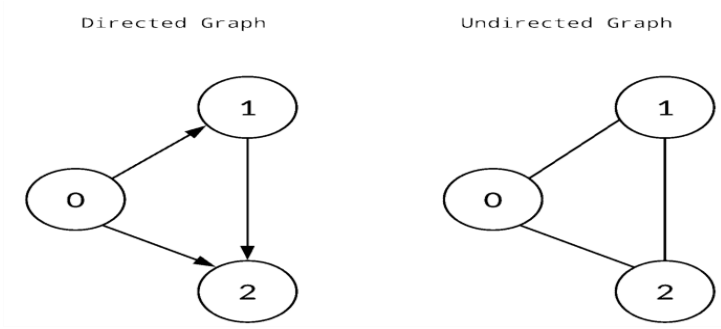
V={A, B, C, D, E}

E={(A,B), (A,C), (A, D), (B,D),(B,E), (C,D),(D,E)}

- Edges of the graph can be directed or undirected

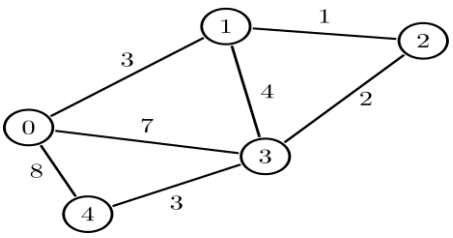
Digraph: If all edges of the graph are directed it is called a directed graph or diagraph.

Undirected graph: If all edges of the graph are undirected it is called undirected graph.



Weighted graph: A graph is called a weighted graph if all the edges are labelled with some weights.

Example:

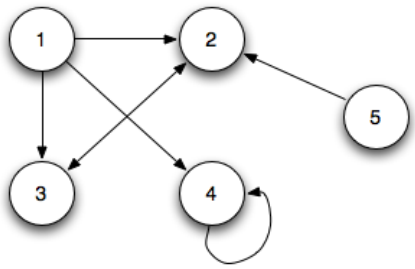


Adjacent Vertices: A vertex v_i is adjacent to (or neighbour of) another vertex v_j if there is an edge from v_i to v_j .

Example: vertex 1 is adjacent to vertex 2, 3, and 4 in below figure.

Self loop: If there is an edge whose start and end vertices are same i.e. (v_i , v_i) then it is called a self loop or loop.

Example: The edge (4, 4)



Degree: Total number of edges connected to a vertex.

Indegree: Total number of incoming edges connected to a vertex is called the indegree of that vertex.

Example: indegree of vertex 2 =3 in above figure

Outdegree: Total number of outgoing edges connected to a vertex is called the outdegree of that vertex.

Example: outdegree of vertex 5 =1 in above figure

Path: Sequence of vertices in which each pair of successive vertices is connected by an edge

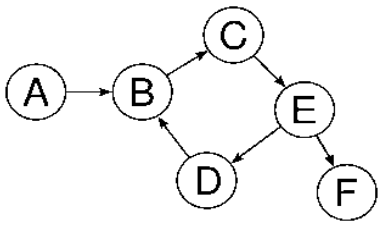
Example: path to reach from vertex 0 to 3 in below figure:

Path-1: 0-1-3,

Path-2: 0-2-3

Cycle: A path that starts and ends on the same vertex

Example:



Graph contains cycle: B-C-E-D-B

<u>Representation of Graphs:</u>

Graphs can be represented using 2 approaches:

1. Sequential or Matrix representation
2. Linked list representation

1. Matrix Representation:

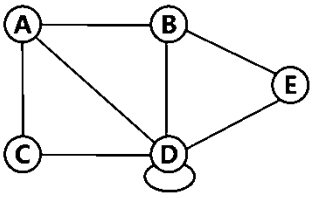
There are 2 commonly used matrices to represent graphs:

A. Adjacency Matrix
B. Incident Matrix

A. **Adjacency Matrix:**

- In this representation, the graph is represented using a matrix of size:
  total number of vertices X *total number of vertices*.
- i.e a graph with 5 vertices is represented using a matrix of size 5 X 5.
- 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.

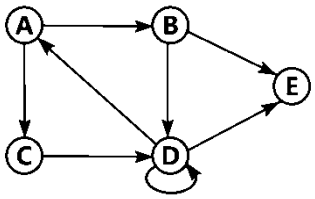
Example-1: undirected graph representation



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

For any edge (A,B) in undirected graph, matrix[A][B]=1, also matrix[B][A]=1

Example-2: directed graph representation



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

For any edge (A,B) in directed graph, matrix[A][B]=1, but matrix[B][A]=0

For self loop for the edge(D,D), matrix[D][D]=1

B. Incidence Matrix

- In this representation, the graph is represented using a matrix of size:

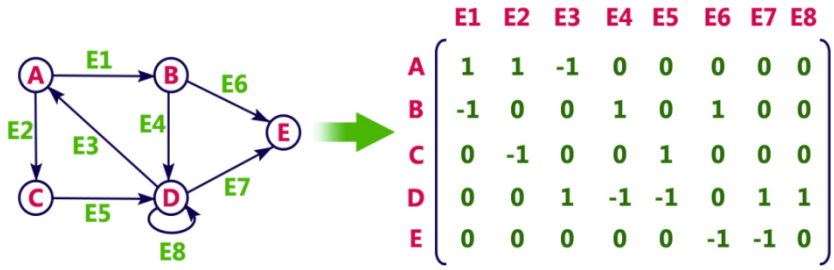
  total number of vertices X *total number of edges.*
- i.e a graph with 5 vertices and 8 edges is represented using a matrix of size 5 X 8.
- In this matrix, rows represent vertices and columns represents edges.
- This matrix is filled with 0 or 1 or -1.
- Here,

  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.

Example:



For vertex A:

outgoing edges are: E1, E2 => matrix[A][E1] and matrix[A][E2]=1

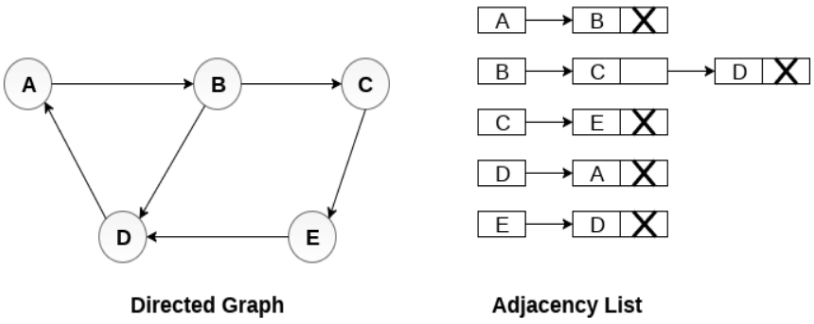
incoming edge: E3 => matrix[A][E3]=-1

all other entries for A^{th} row in matrix are 0

2. Linked Representation:

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list.

Example:



Graph Traversal:

- The graph traversal is used to decide the order of vertices visited in the search process.

i.e. using graph traversal all the vertices of the graph are visited without getting into loops.

- There are two graph traversal techniques used:

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

1. **DFS Traversal:** *(uses stack)*

- DFS algorithm starts with an initial node of the graph G, and then goes to deeper and deeper until the searching node or the node which has no neighbour is found.

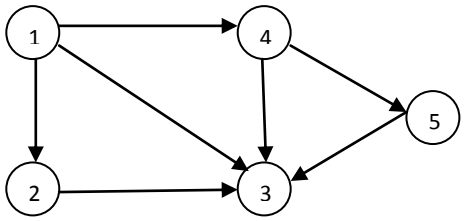
- The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

- The Stack data structure is used to implement DFS traversal
- The maximum size of stack used is the total number of vertices in the graph.

Algorithm:

Step 1: Initialize all nodes in graph G to ready state
Step 2: Push the starting node on to the stack and change its state to waiting state
Step 3: Repeat Steps 4 and 5 until stack is empty
Step 4: Pop the top node N from stack, process it and change its state to processed state
Step 5: Push all neighbours of processed node N that are in the ready state on to the stack and change their state to waiting state

Example:



G

Node	Adjacent list
1	2, 3, 4
2	3
3	-
4	3, 5
5	3

Maximum stack size= 5
Initially stack is empty
Initialize all vertices to ready state and select the starting state.
Let the starting vertex be vertex 1.
The sequences of steps executed using DFS algorithm are:

Stack	Ready State	Waiting State	Processed State
	1, 2, 3, 4, 5	-	-
1	2, 3, 4, 5	1	-
4 3 2	5	2, 3, 4	1
5 3 2	-	2, 3, 5	1, 4
3 2	-	2, 3	1, 4, 5
2	-	2	1, 4, 5, 3
	-	-	**1, 4, 5, 3, 2**

The stack is now empty.
So DFS traversal of graph G starting at node 1 is complete.
The nodes are processed in the order: **1, 4, 5, 3, 2.**

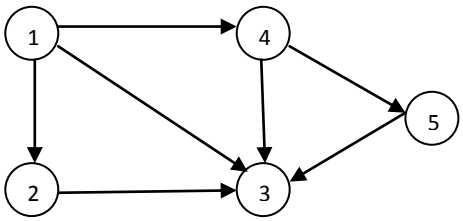
2. BFS Traversal: *(uses Queue)*

- It starts from the starting node and explores all of the neighbours at present depth before moving to next depth level.
- The queue data structure is used to implement BFS traversal
- The maximum size of queue used is the total number of vertices in the graph.

Algorithm:

Step 1: Initialize all nodes in graph G to ready state
Step 2: Insert the starting node in to the queue and change its state to waiting state
Step 3: Repeat Steps 4 and 5 until queue is empty
Step 4: Remove the front node N from queue, process it and change its state to processed state
Step 5: Insert all neighbours of processed node N that are in the ready state in to the queue and change their state to waiting state

Example:



Node	Adjacent list
1	2, 3, 4
2	3
3	-
4	3, 5
5	3

Maximum queue size= 5
Initially queue is empty
Initialize all vertices to ready state.
Let the starting state be vertex 1.
The sequences of steps executed using BFS algorithm are:

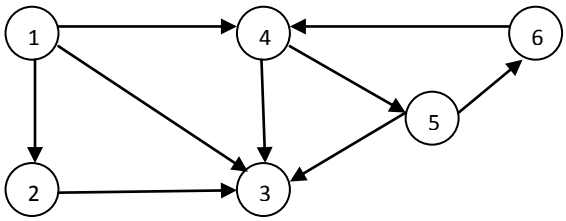
Queue	**Ready State**	**Waiting State**	**Processed State**
	1, 2, 3, 4, 5	-	-
1	2, 3, 4, 5	1	-
2 3 4	5	2, 3, 4	1
3 4	5	3, 4	1, 2
4	5	4	1, 2, 3
5	-	5	1, 2, 3, 4
	-	-	**1, 2, 3, 4, 5**

The queue is now empty. So BFS Traversal of graph G starting at node 1 is complete.
The nodes are processed in the order: **1, 2, 3, 4, 5.**

Questions to practice:

1. Find the sequence of steps executed and the order in which nodes are processed using DFS and BFS on above graph with starting vertex 2.

2. Find the sequence of steps executed and the order in which nodes are processed using DFS and BFS on below given graph.



Sorting Algorithms

1. Bubble Sort:

- Bubble sort proceeds by scanning the list from left to right and whenever a pair of adjacent keys found out of order, those elements are swapped. This process continues till all the elements of the array are in sorted order.

Algorithm:

```
void bubbleSort( int a[ ], int n )
        {
         int i, j, temp;
         for(i = 0; i< n-1; i++)
                {
                for(j = 0; j < n-i-1; j++)
                        {
                         if(a[ j ] > a[ j+1] )
                                {
                                temp = a[ j ];
                                a[ j ] = a[ j+1 ];
                                a[ j + 1] = temp;
                                }
                        }
                }
        }
```

Example: a:

44	55	33	88	77	22	11	66

44	**55**	33	88	77	22	11	66

44	33	55	**88**	77	22	11	66

44	33	55	77	**88**	22	11	66

44	33	55	77	22	**88**	11	66

44	33	55	77	22	11	**88**	66

44	33	55	77	22	11	66	**88**

At the end of first pass, the largest element is bubbled up to the last array position.

Continuing the same process at the end of 2nd pass the output will be:

33	44	55	22	11	66	**77**	**88**

Continuing the same process at the end of 3rd pass the output will be:

33	44	22	11	55	**66**	**77**	**88**

..
...
...

Finally form the sorted array:

11	**22**	**33**	**44**	**55**	**66**	**77**	**88**

2. Selection Sort

The selection sort algorithm selects the largest element from the unsorted array in each iteration and place it in its appropriate position till the whole array is sorted. This is also called as *push-down sort*.

Note: Instead of maximum, the minimum element can also be used.

Algorithm:

```
void selectionSort(int a[], int n)
        {
        int i, j, max, pos;
        for (i=n-1; i>0; i--)
                {
        max = a[0];
                pos=0;
        for (j =1; j <=i; j++)
                        {
                if (a[j] >max)
                                {
                                max=a[j];
                                pos=j
                                }
                }
                a[pos]=a[i];
                a[i]=max;
                }
        }
```

Example:

a:

30	20	35	14	90	25	32

30	20	35	14	**90**	25	32

max=90 swap with last position element i.e.32

30	20	**35**	14	32	25	90

max=35 swap with last position not sorted yet. i.e.25

30	20	25	14	**32**	35	90

max=32, same position so no exchange

30	20	25	14	32	35	90

max=30, swap with 14

14	20	**25**	30	32	35	90

max=25, same position so no exchange

14	**20**	25	30	32	35	90

max=20, same position so no exchange

14	20	25	30	32	35	90

max=14, same position so no exchange

14	20	25	30	32	35	90

Sorted array

3. Insertion Sort

It sorts the values by inserting them into their appropriate positions in an already sorted sub array.

Algorithm:

```
void insertionSort(int a[], int n)
        {
        int i, key, j;
        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;
                }
        }
```

Example

a:

30	20	35	14	90	25	32

30	**20**	35	14	90	25	32

20	30	**35**	14	90	25	32

20	30	35	**14**	90	25	32

14	20	30	35	**90**	25	32

14	20	30	35	90	**25**	32

14	20	25	30	35	90	**32**

14	20	25	30	32	35	90

Sorted array

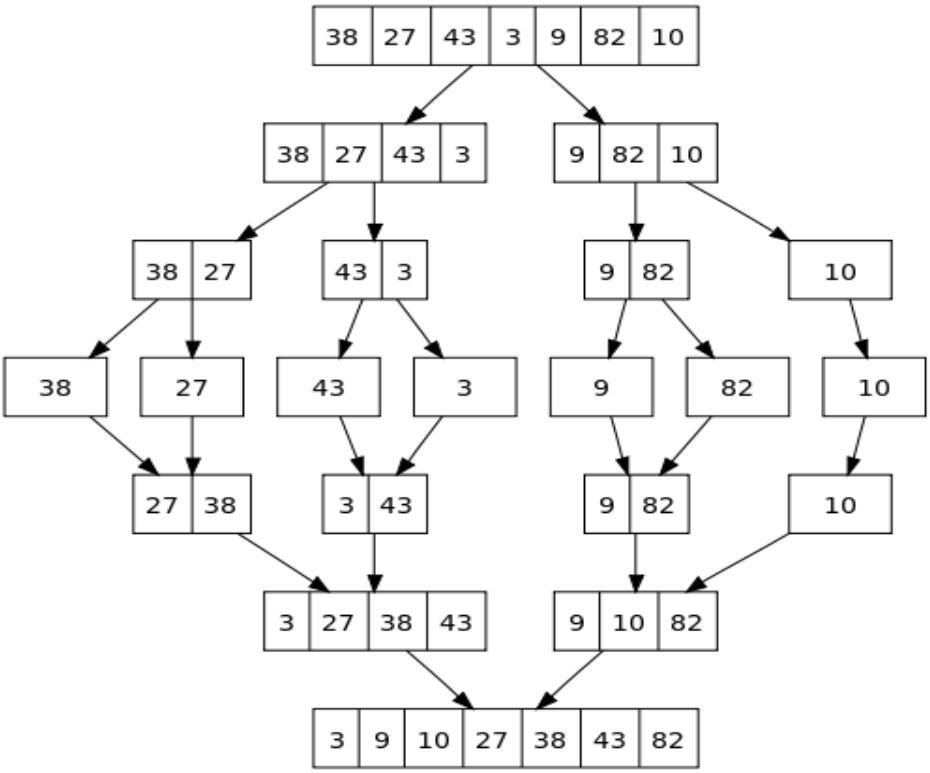
4. Merge Sort:

Merge Sort follows the **Divide and Conquer** mechanism to sort a given set of numbers recursively, hence consuming less time compared to other sorting algorithms.

The concept of Divide and Conquer involves three steps:

1. **Divide:** Divide the problem into multiple small problems.
2. **Conquer:** Conquer the subproblems by solving them.
3. **Combine:** Combine the solutions of the subproblems to find the solution of the actual problem.

Example:



Algorithm:

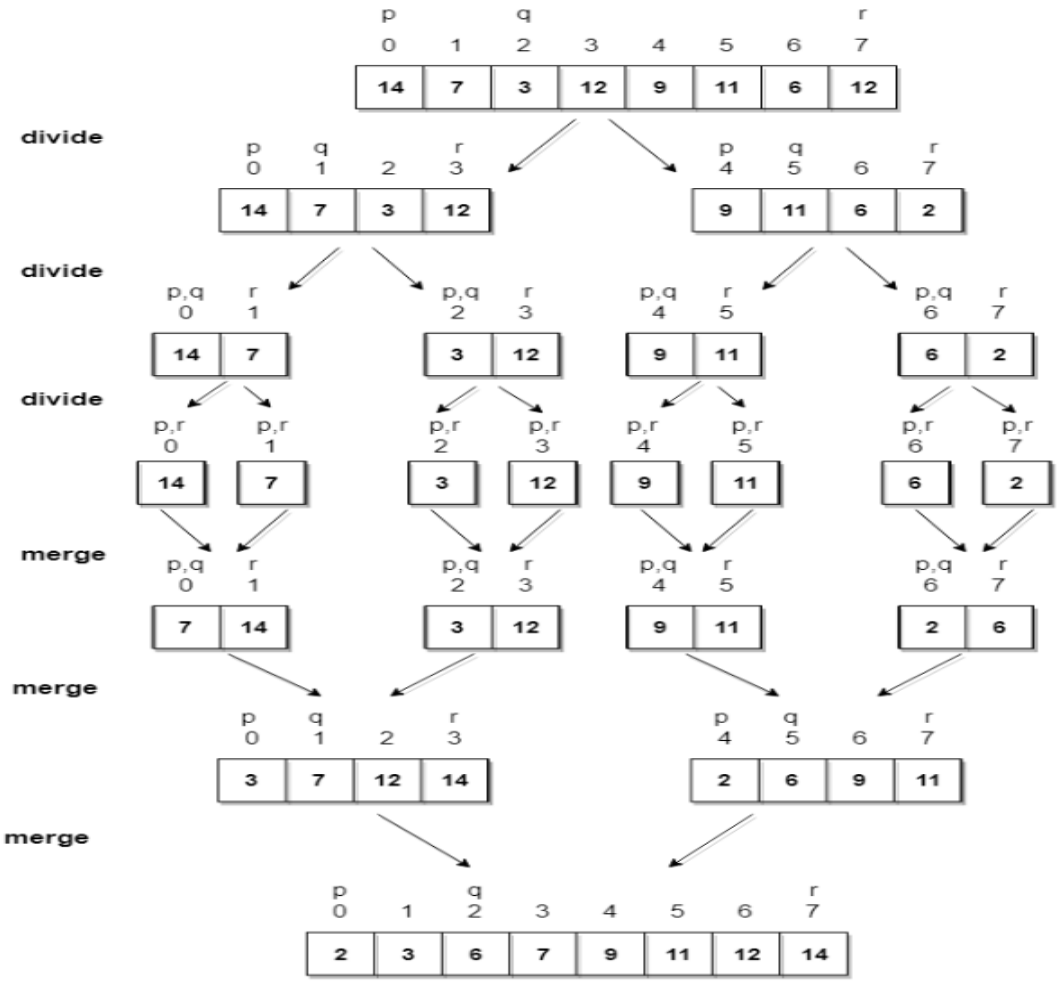
// merge sort function

```
void mergeSort(int a[], int p, int r)
{
    int q;
    if(p<r)
     {
        q=(p+r)/2;
        mergeSort(a,p,q);
        mergeSort(a,q+1,r);
        merge(a,p,q,r); //function to merge the subarrays
    }
}
```

// function to merge the subarrays

```
void merge(int a[], int p, int q, int r)
     {
     int b[5];   //same size of a[]
     int i,j,k;
     k=0;
     i=p;
     j=q + 1;
     while(i<=q && j<=r)
          {
          if(a[i]<a[j])
               {
               b[k]=a[i];
               k++;
               i++;
               }
          else
               {
               b[k]=a[j];
               k++;
               j++;
               }
          }
     while(i <= q)
          {
          b[k]=a[i];
          k++;
          i++;
          }
     while(j <= r)
          {
          b[k]=a[j];
          k++;
          j++;
          }
  // copying back the sorted list from b[] to a[]
    for(i=p;i<=r;i++)
          {
        a[i]=b[i];
          }
     } //end of merge function
```

Example:



5. Quick Sort:

- Like Merge Sort, Quick Sort also follows the **Divide and Conquer** mechanism to sort a given set of numbers recursively.
- The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively.
- The partition of the list is performed based on a selected **pivot** element from 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.

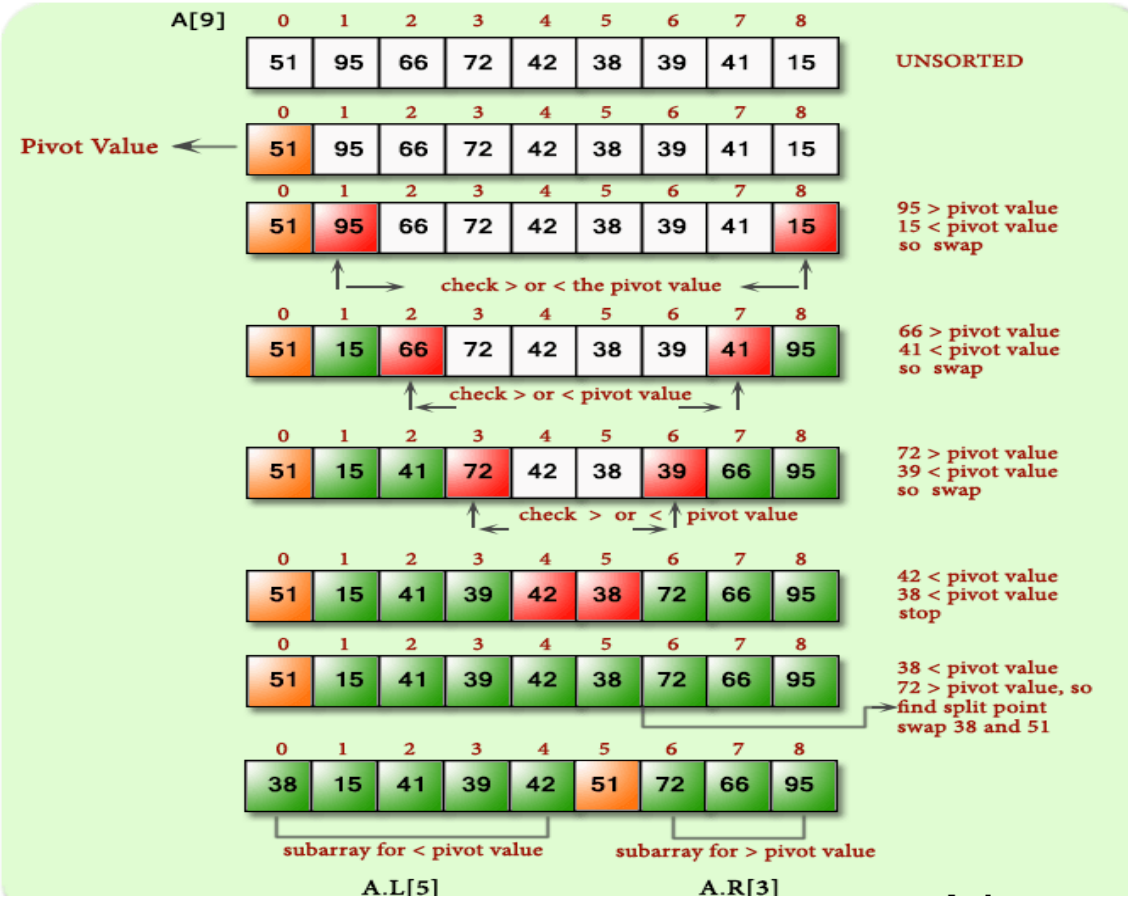
- Pivot element can be any element from the array:

  it can be the first element, or

  the last element or

  any random element.

Example-1: Considering first element as pivot



Applying the same process of partitioning recursively on left and right sub arrays, finally the obtained sorted array will be:

15	38	39	41	42	51	66	72	95

Algorithm:

```
void quicksort(int a[], int p, int r)
{
    int q;
    if(p < r)
    {
        q = partition(a, p, r);
        quicksort(a, p, q);
        quicksort(a, q+1, r);
    }
}

int partition(int a[], int p, int r)
{
     int pivot, i, j, temp;
     pivot = a[p];  // selecting first element as pivot
     i=p-1;
     j=r+1
     while(1)
     {
          do
          {
               j=j-1;
          }while(a[j]>pivot);
          do
          {
               i=i+1;
          }while(a[i]<pivot);
          if(i<j)
          {
               temp=a[i];
               a[i]=a[j];
               a[j]=temp;
          }
          else
               return(j);
     }
}
```