

Course Name : Data Structures (تراکيب بيانات)

Other names :

- Data Structures in JAVA
- Data Structures in C#
- Data Structures in C
- Data Structures And Algorithms
- Programming Techniques
-

There are two questions that must be answered :

- How could I allocate the required memory space needed in my program ?
(Means : Data Structures for the program) First Term
- How could I use this space in my program ?
(Means : Algorithm for my program) Second Term

➔ Basically any program needs data structures and algorithm

Example in JAVA : `int x ; // Memory Allocation or Data Structure`
`x = 10/2 ; // Algorithm1`
`or`
`int y = 5 ; // Algorithm2`
`x = y;`

Contents :

CH1 • JAVA review

CH2 • Data Representation : (مشروح ضمن CH3)

- a) Fixed Memory Allocation
- b) Dynamic Memory Allocation

CH3• Linked Data Representation

- a) Single (Linear) Linked Lists
- b) Double Linked Lists
- c) Circular Linked Lists
- d) Ring Linked Lists
- e) Stack
- f) Queue

.....

CH4• Recursion

CH5•Linear Data Structures (Stacks , Queues)

CH6• Binary Trees :

- a) Complete Trees
- b) Heap Trees
- c) Binary Search Trees
- d) AVL Trees

.....

CH7• Heap Array , Heap Tree , Priority Queue

CH8• Graph

CH9• Hashing

References :

- 1- Data Structures in JAVA , Thomas A. Standish , Addison Wesley Longman
- 2- Data Structures and Alorithms , Iain T. Adamson , Springer
- 3- Lecture Notes
- 4- Other Materials

CH1 : JAVA Review :

Remember ➔

Definition of Data Type :

- *Predefined Data type* (or *From System Defined Data type*)
(Examples : int , float , char, boolean, double ,)
- *From user defined Data type*
(Examples : class Rectangle {...} ,)

Other characteristics :

- *Primitive Data Type* (Example : int x = 10)
- *NON-Primitive data Type* [Example : Rectangle x = new Rectangle(..)]

Functions (Methods) :

- call by value
- call by reference
- Recursion

Object Oriented Programming :

.....
.....

Examples :

1- Predefined Data type (Primitive Data Type)

Write a JAVA code to calculate the average of 10 integer values, entered from keyboard .

Solution1 :

```
int x1 , x2 , x3 , x4 , x5 ,x6 , x7 , x8 , x9 , x10 ; // variables declaration or
                                                    //data structure for the program

// Body of Alogrithm
{   Read (x1,x2,.....) ;
    ....
    Avg = ( x1 + x2 + ..... ) / 10 ;
    ....
}
```

Solution 2 :

```
int x ;                                // variables declaration or
                                      //data structure for the program

// Body of Algorithm
{ for( int i = 0 ; i <=9 ; ++i )
    { read(x) ;
      sum += x ;
    }
  avg = sum/10;
  ...
}
```

Solution3 :

```
int x[10] ;                            // variables declaration or
                                      //data structure for the program

// Body of Algorithm
{ for( int i = 0 ; i <=9 ; ++i )
    { read(x[i]) ;
      sum += x[i] ;
    }
  avg = sum/10;
  ...
}
```

2- From user defined Data type (NON-Primitive data Type)

```
public class Rectangle
{
    public int x ;
    public int y ;
    public int width ;
    public int height ;

    // constructors
    public Rectangle ( )
    {
        x = y = width = height = 0 ;
    }
    Rectangle(int x,int y,int width,int height )
    {
        this.x = x ;
        this.y = y ;
        this.width = width ;
        this.height = height ;
    }

    // Setter and getter
    .....
}
```

●Variable Declaration in any program using above class as Data Type :

1- Rectangle myRect = null; // initialized with null

2- Rectangle myRect=new Rectangle(); // 3 actions applied

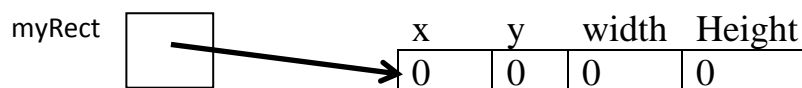
Meaning of 3 actions :

The above statement does 3 things :

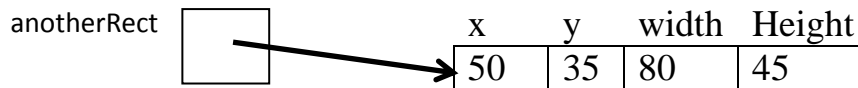
A- It allocates memory for a new **Rectangle** object

B- It initializes all of the object's data fields to zero

C- It returns a **reference** to this newly constructed object as the value of the constructor expression

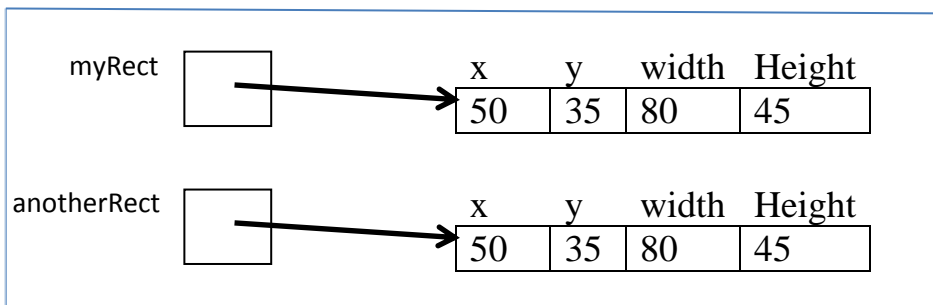


3- Rectangle anotherRect=new Rectangle(50,35,80,45);
 // 3 actions

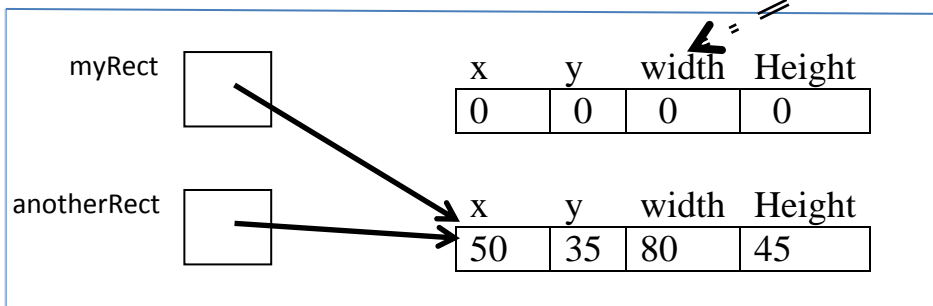


4- myRect = anotherRect ; // What will return this statement ?

Result1 : (**FALSE**)



Result2 : (**TRUE**)



Questions :

- If (myRect.x == 0) → ?
- anotherRect.y = myRect.y + 3 → ? answer : 0+3 = 3 OR 35+3 = 38
- S.O.P.(myRect.width) → ?

CH3 : Linked Data representation :

Meaning of :

- 1- Fixed Memory Allocation
- 2- Dynamic Memory Allocation

يشرح ضمن هذا الشبتر ومعلوماتك
لمقرر جافا من الفصل الثاني

Example : (*Fixed Memory Allocation*)

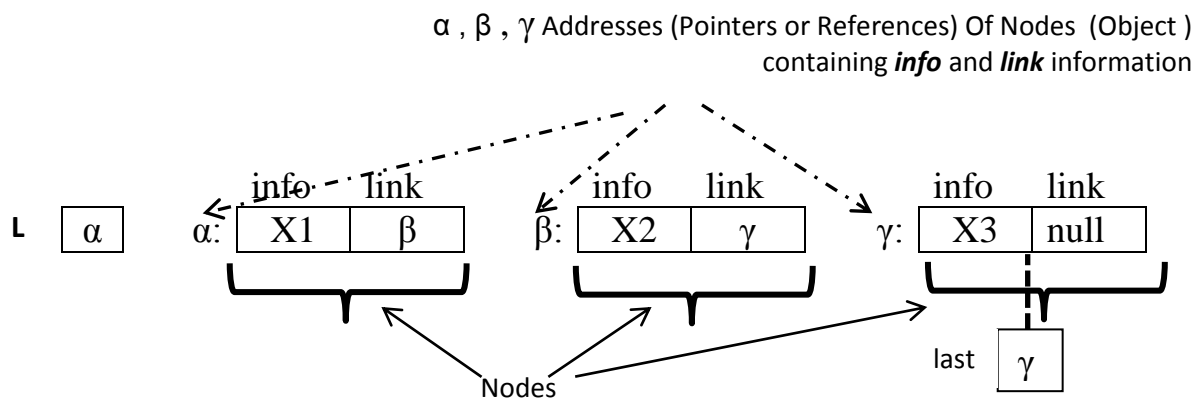
```
float y ; // Non dim variable
int x[10] ; // one dim array
char name[3][20] ; two dim array
```

```
for ( int i = 0 ; i <= 9 ; ++i )
    read(x[i]) ; // ERROR , if we try to insert a new element x[10]
```

► To understand **Dynamic Memory Allocation** :

3.1 Pointer (Reference) diagramming Notation

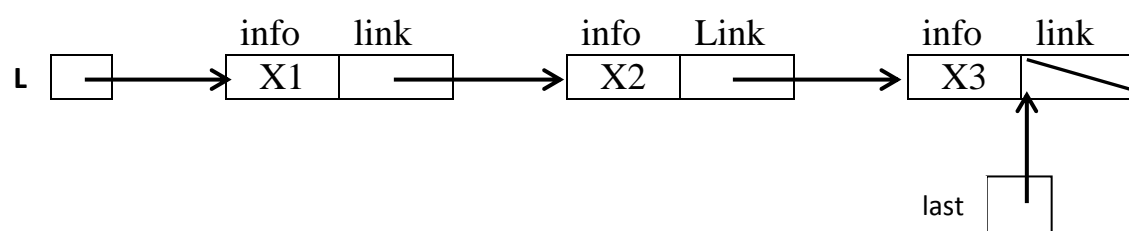
Notes : Every cell in Memory (Main or Secondary) has an Address.



The cell **L** refers to the Node, which contains the value X1 and

The cell **Last** refers to the Node, which contains the value X3


• **Alternately :**



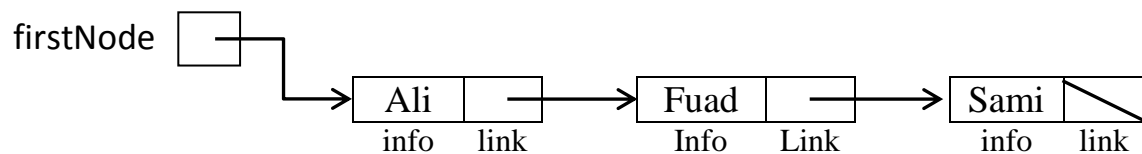
3.2 Single (Linear) Linked List :

► *Diagramming Representation of Single Linked List :*

Empty List :

• firstNode 

• NON-Empty List :



► *Definition of Single Linked List :*

Any SLL should have at least following properties :

- 1- Every Node contains the needed information(*same data type*) to be stored in memory and a reference (pointer) to next Node , if there is any one, otherwise null.
- 2- The address of the first node stored in extra cell (by example *firstNode*) , if (*firstNode == null*) → Empty List.
- 3- It is necessary to determine, which Node will be the last node with *link* value of last Node equal to null [**end of list**].

► *Declaration of Single Linked List : (Abstract Data type ADT)*

```
public class ListNode
{
    String info ;
    ListNode link ; // SelfReferential Pointer

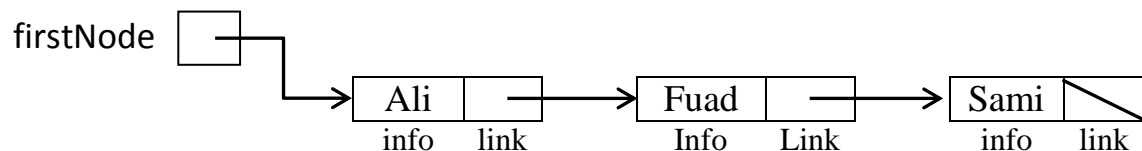
    // Constructors
    public ListNode ( )
    { this(null,null); }
    public ListNode(String info, ListNode link)
    { this.info = info ;
      this.link = link ;
    }
    .....
}
```


► *Manipulation Single Linked List :*

- Creation a new Single Linked Lists
- Printing
- Inserting new node (at Head , Between , At Last)
- Deleting a node (from Head , Between , from Last)
- Count the nodes
- Splitting up the list
- Concatenation two lists
- Updating the contents of the information in any node
-

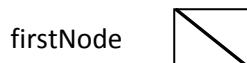
● Creation a new Single Linked Lists

Example1 : Creating a Single Linked List (*Naïve Approach1*)



- *Steps to create above Structures :*

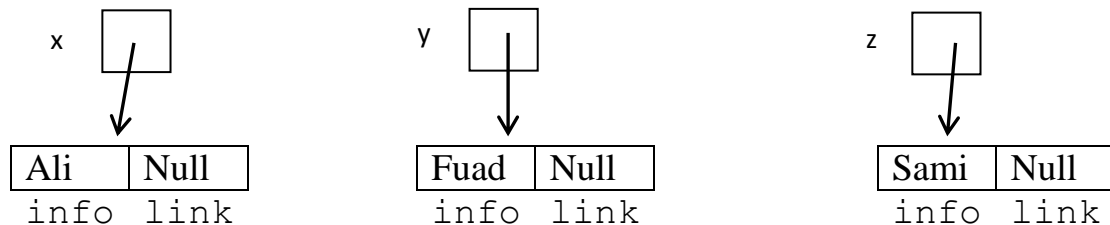
1. `ListNode firstNode = null ;`



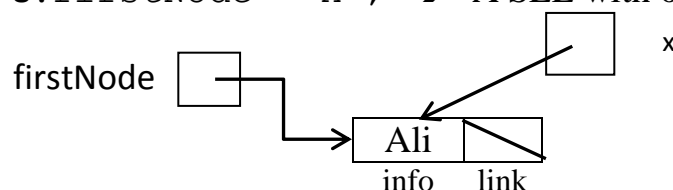
2. `ListNode x = new ListNode("Ali", null); // 3 actions`

3. `ListNode y = new ListNode("Fuad", null); // 3 actions`

4. `ListNode z = new ListNode("Sami" , null); // 3 actions`

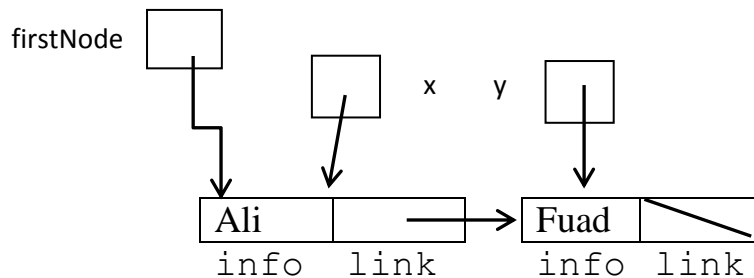


5. `firstNode = x ;` ➔ A SLL with only one Node :

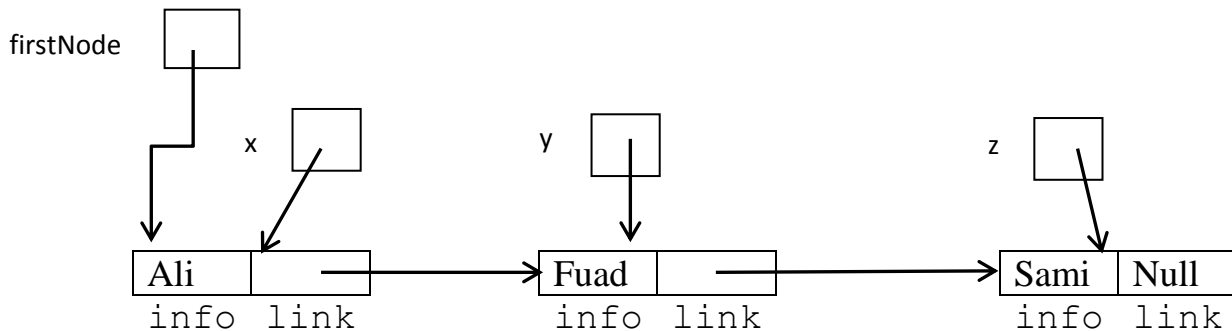


6. `firstNode.link = y ; // or x.link = y ; ➔`

A SLL with two Nodes :

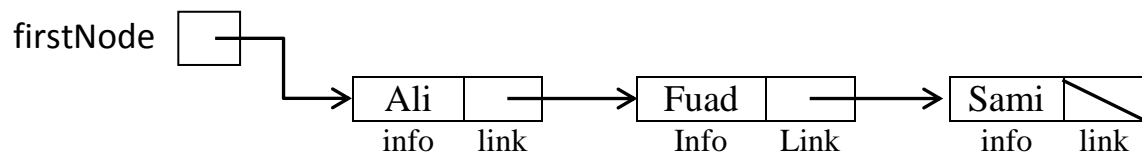


7. `firstNode.link.link = z ; // or x.link.link = z ;
// or y.link = z ; ➔` A SLL with three Nodes :



Disadvantage : needs to many variables x,y,z,..... to define Nodes

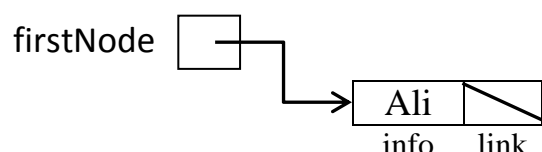
Example2 : Creating a Single Linked List (*Naïve Approach2*)



- Steps to create above Structures :

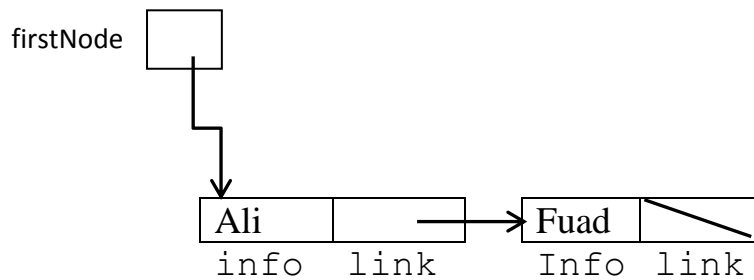
1. `ListNode firstNode=new ListNode("Ali",null); // 3 actions`

➔ A SLL with only one Node :



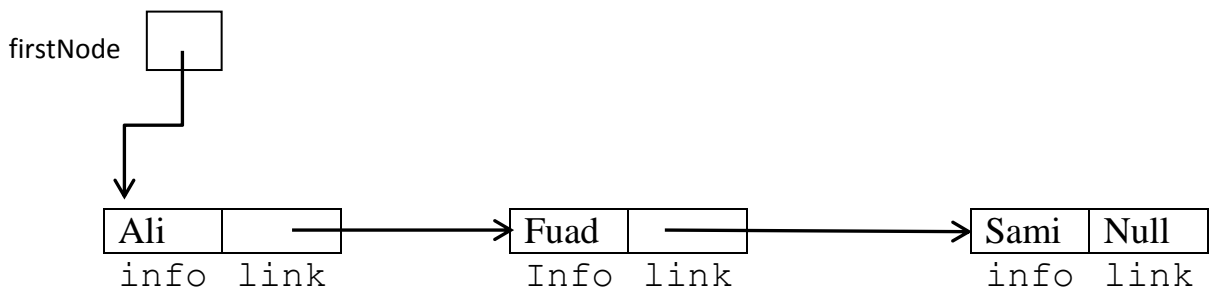
2. `firstNode.link = new ListNode("Fuad",null); //3 actions`

➔ A SLL with two Nodes :



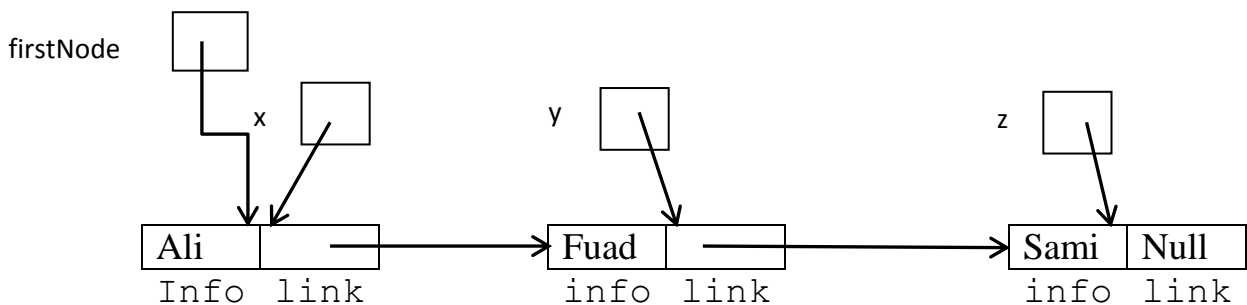
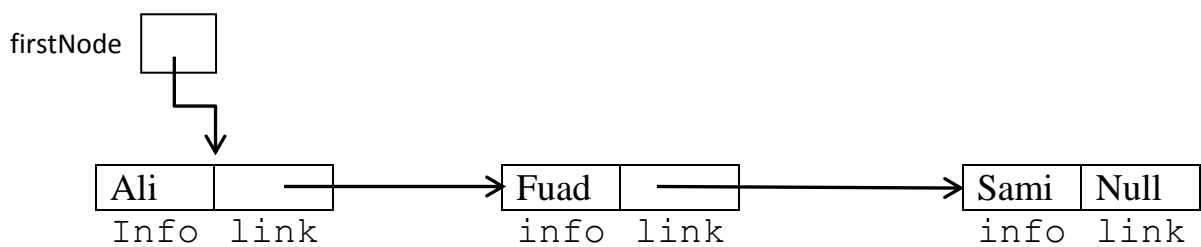
3. `firstNode.link.link=new ListNode("Sami",null); //3 actions`

➔ A SLL with three Nodes :



Disadvantage : How many times `firstNode.link.link.link.link.link....`

-
- Printing the contents of the above list : (naïve Approach)



- 1- S.O.P.(firstNode.info); // S.O.P.(x.info) → **Ali**
- 2- S.O.P.(firstNode.link.info); // S.O.P.(x.link.info)
//or S.O.P.(y.info) → **Fuad**
- 3- S.O.P.(firstNode.link.link.info); // S.O.P.(z.info)
//or S.O.P(y.link.info) or S.O.P.(x.link.link.info)
→ **Sami**

```

public class ListNode
{
    String info ;
    ListNode link ; // SelfReferential Pointer

    // Constructors
    public ListNode ( )
    { this(null,null); }
    public ListNode(String info, ListNode link)
    { this.info = info ;
      this.link = link ;}
//Setter and Getter
    public void setInfo( String info)
    { this.info = info;}

    public String getInfo()
    { Return this.info;}

    public void setLink(ListNode link)
    { this.link = link; }

    public ListNode getLink()
    { return this.link; }
}

Class Test
{ public static void main(String args[])
{
    ListNode firstNode = null ;
    ListNode x = new ListNode( "Ali", null ); // 3 actions
    ListNode y = new ListNode( );           // 3 actions
    ListNode z = new ListNode("Sami" , null); // 3 actions

    //y.info = "Fuad";
    y.setInfo("Fuad");

    //x.link = y ;
    x.setLink(y);

    //x.link.link = z ;
    x.link.setLink(z);

    firstNode = x ;

    //Printing the contents of the list
    System.out.println(firstNode.getInfo()+"\n"+firstNode.link.info
    +"\n"+firstNode.link.link.getInfo());

}
}

```

- Other manipulation of SLL (Like Searching , insertion , deletion , printing , ...)

Suppose we have following 3 classes :

1- //class to define Abstract Data Type ADT of Linked List

```
class ListNode
{   String info;
    ListNode link;

    // constructors
    .....
    // Setter and Getters
    ....
}
```

2. // class contains all possible manipulation methods (print , insert , delete , ...)

```
class LinkedList
{   ListNode firstNode ;
    int length;

    // methods to manipulate Single Linked Lists
    ..print (..)...;
    ..insert (..) ..;//insert last or at head or between
    ..delete ( ..) ..;//delete last or fromHead or between
    ..countNodes(..) .. ;
    .....}
```

3. // Class containing main statement ..

```
class Test
{   static void main(String args[])
    {
        //String name;
        //ListNode x ;
        ...
        LinkedList L = new LinkedList();
        ...
        // L.print();
        // x = L.listSearch(name);
        // L.insertNewSecondNode(name);
        // L.insertLast(name);
        // L.deleteLast();
        // L.reverse();
    } }
```

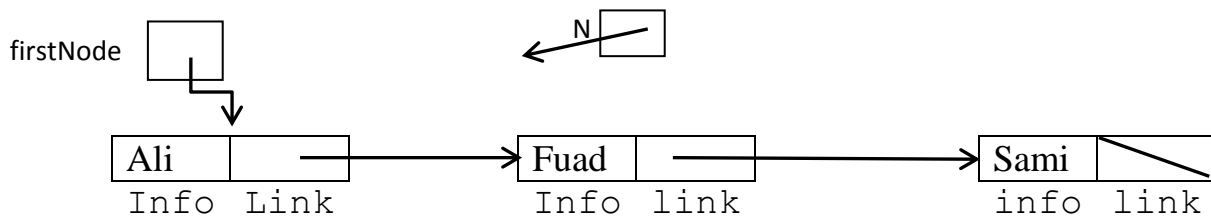
► **Methods to manipulate SLL (implemented in class LinkedList):**

1. Printing the contents of the SLL :

```
void print ( )
{
    ListNode N ;
    N = firstNode ;// Local variable

    while ( N != null )
    {
        S.O.P.(N.info);
        N = N.link;
    }
}
```

Example : Tracing L.print() , method calling

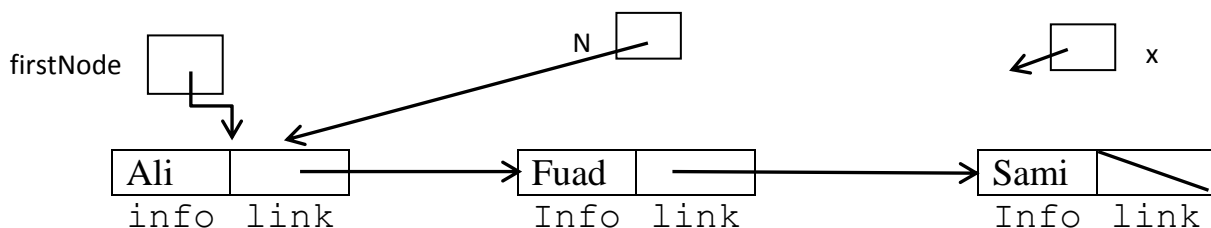


2. Searching the contents of the SLL :

```
ListNode listSearch (String info )
{
    ListNode N ;
    N = firstNode ;// Local variable

    while ( N != null )
    {
        if((info.equals(N.info))
            return N;
        else
            N = N.link;
    }
    Return N;
}
```

Example : Tracing x = L.listSearch(name) , method calling



3. Insert a new second Node into the list :

```
void inserNewSecondNode (String name )
{ if(length==0) //if(firstNode==null) //if(size()== 0)
    return ;
    ListNode newNode = new ListNode( ); // 3 actions
    newNode.info = name;
    newNode.link = firstNode.link;
    firstNode.link = newNode ;
    ++length;
}
```

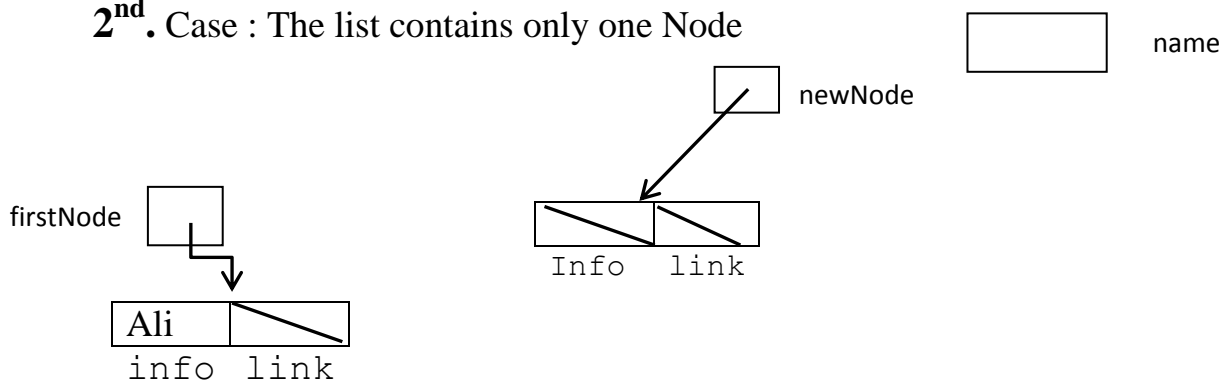
Example : Tracing L.insertNewSecondNode(name), method calling

3 Cases to be considered :

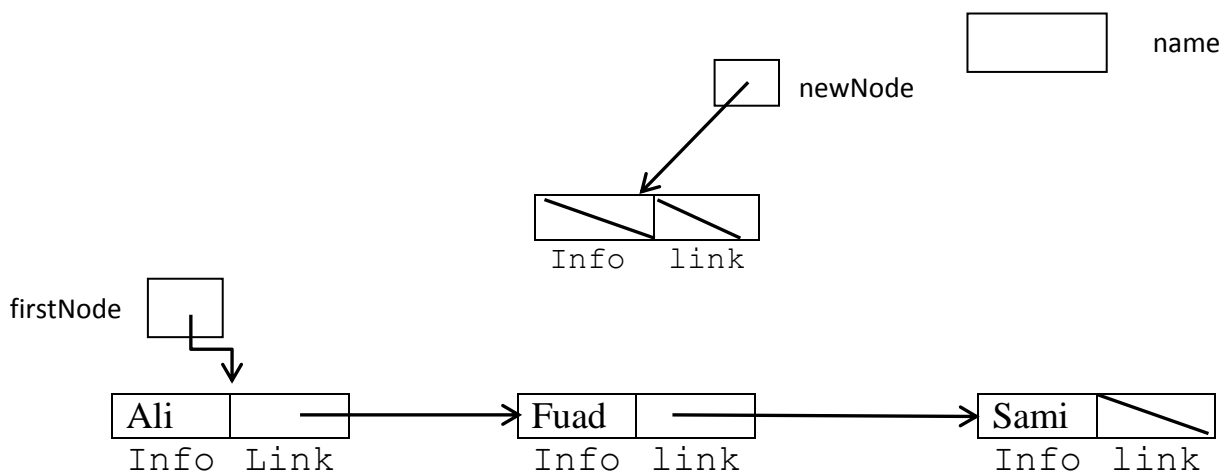
1st.Case : The list is empty

firstNode  → Nothing to insert

2nd.Case : The list contains only one Node



3rd.Case : The list contains many Nodes



4. Insert a new Node to the last of the list :

```
void insertLast(String info)
{
    ListNode N = new ListNode();

    N.info = info;
    N.link = null;

    if (firstNode == null)
        firstNode = N;
    else
    {
        ListNode P = firstNode;
        while (P.link != null)
            P = P.link;
        P.link = N;
    }
    length++;
}
```

Example : Tracing L.insertLast(name), method calling

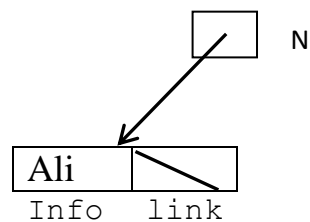
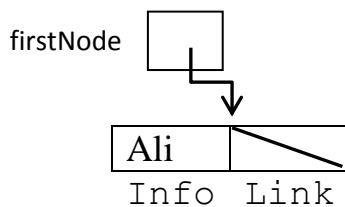
2 Cases to be considered :

1st.Case : The list is empty

Ali name

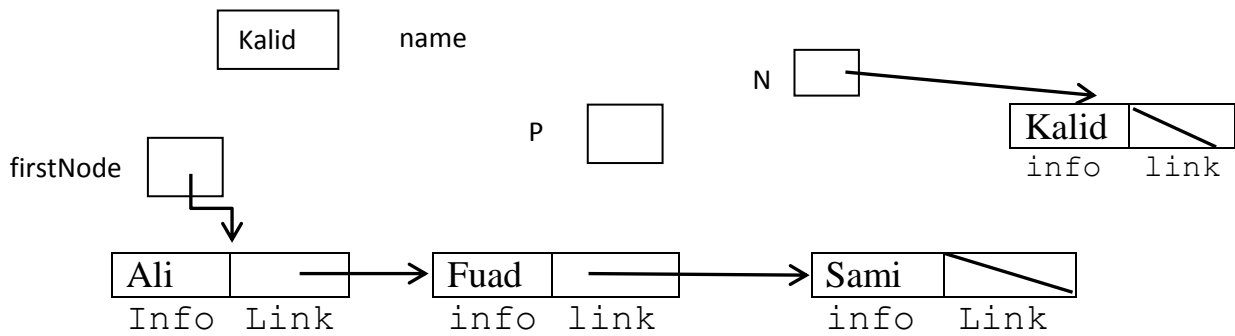
firstNode  ➔ insert Ali ➔

Output ➔

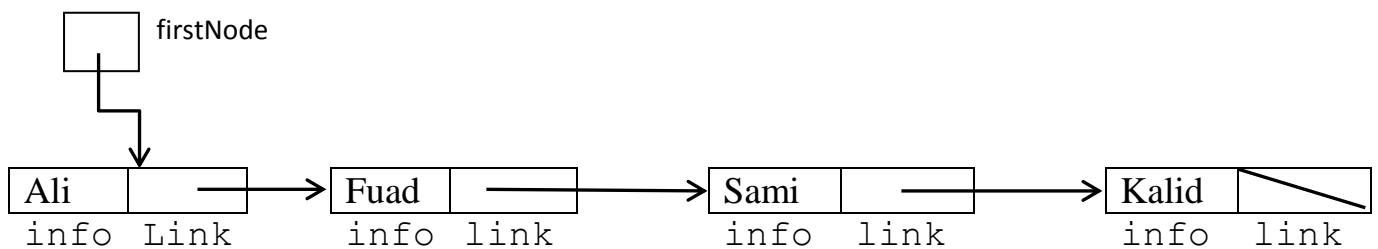


2nd. Case : The list contains many Nodes

Insert "**Kalid**" to the Last :



Output →



5. Delete the last Node from the list :

```
void deleteLast()
{
    ListNode prev , current;

    if (firstNode != null)
    {
        if (firstNode.link == null)
        {
            firstNode = null;
            length--;
        }
        else
        {
            prev = firstNode;
            current = firstNode.link;
            while (current.link != null)
            {
                prev = current;
                current = current.link;
            }
            prev.link = null;
            length--;
        }
    }
}
```

```

    }
  }
}

```

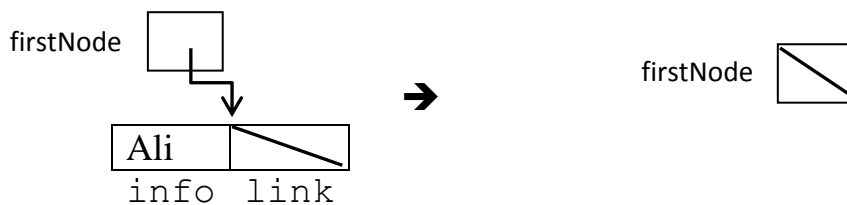
Example : Tracing `L.deleteLast()`, method calling

3 Cases to be considered :

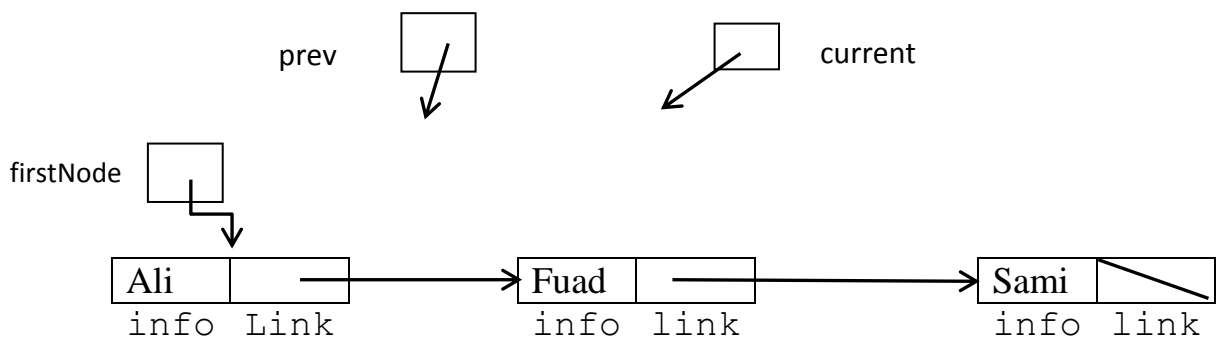
1st.Case : The list is empty

firstNode  → Nothing to delete

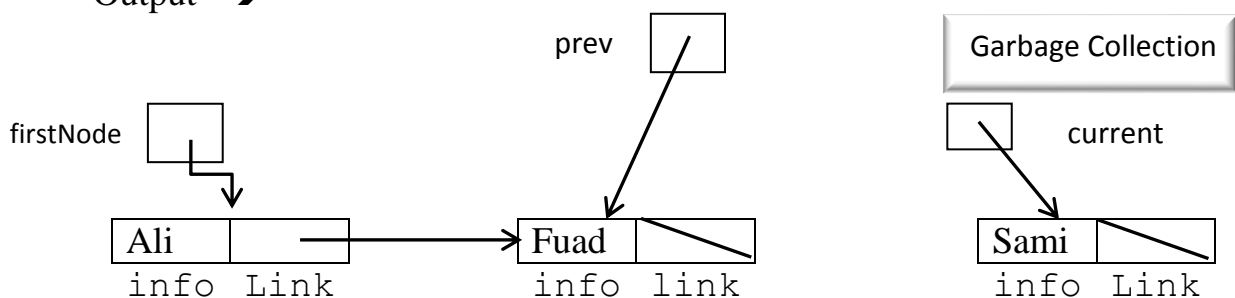
2nd. Case : The list contains only one Node



3rd.Case : The list contains many Nodes



Output →



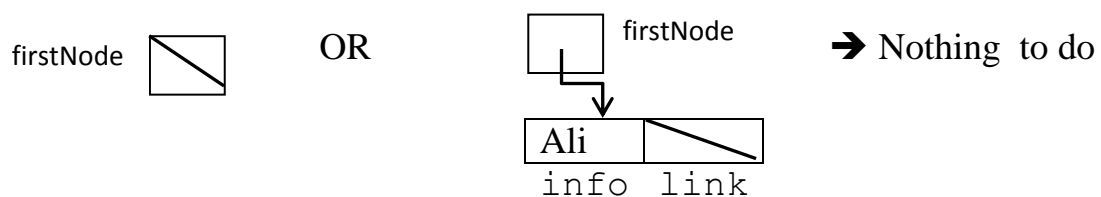
6. Reverse the list :

```
void reverse ( )
{
    ListNode R , N , L1 ;
    R = null ;
    L1 = firstNode ;
    while ( L1 != null )
    {
        N = L1 ;
        L1 = L1.link;
        N.link = R ;
        R = N ;
    }
    firstNode = R ;
}
```

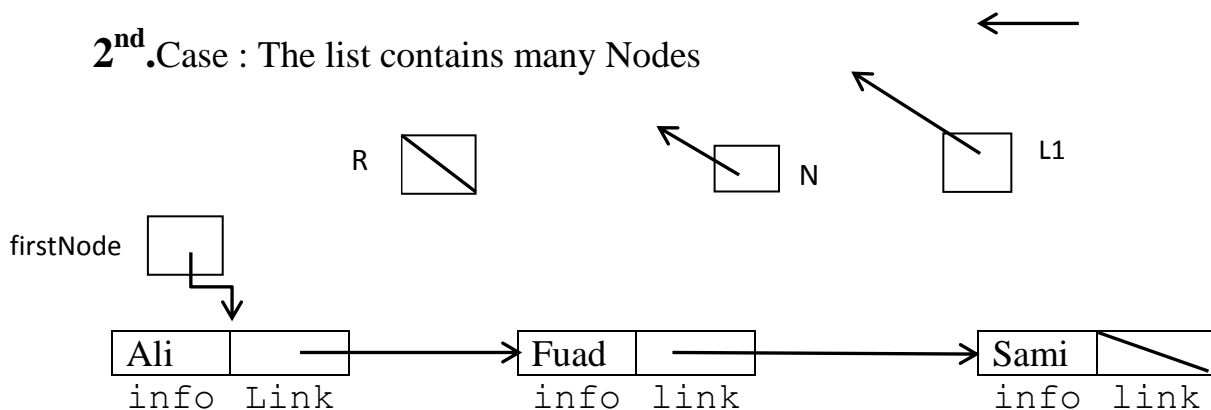
Example : Tracing L.reverse(), method calling

2 Cases to be considered :

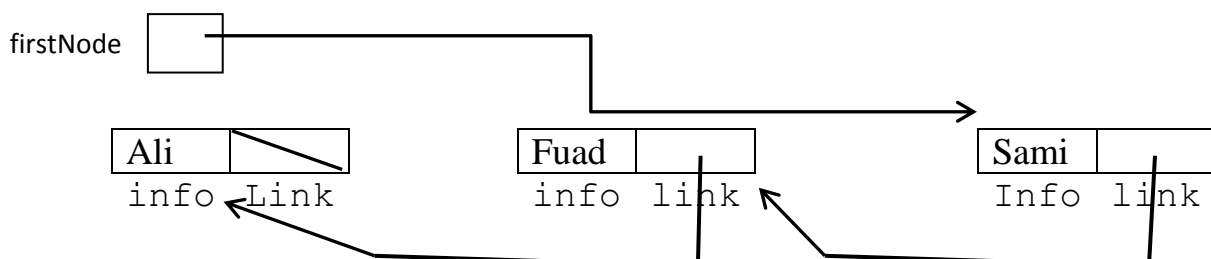
1st.Case : The list is empty OR the list contains only one Node



2nd.Case : The list contains many Nodes



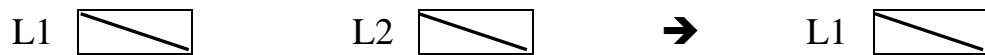
Output ➔



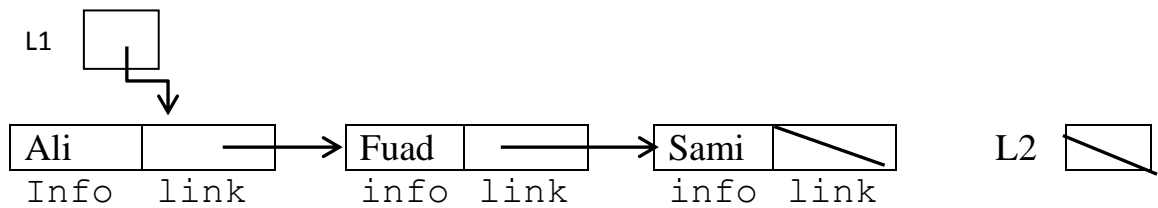
7. Concatenation two Lists (L1 with L2) :

Diagramming representation :

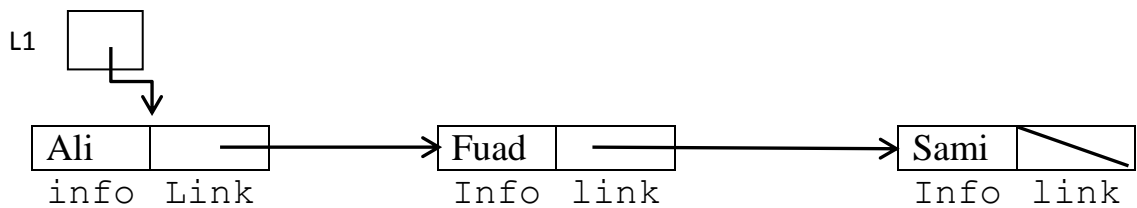
A- L1 and L2 are empty :



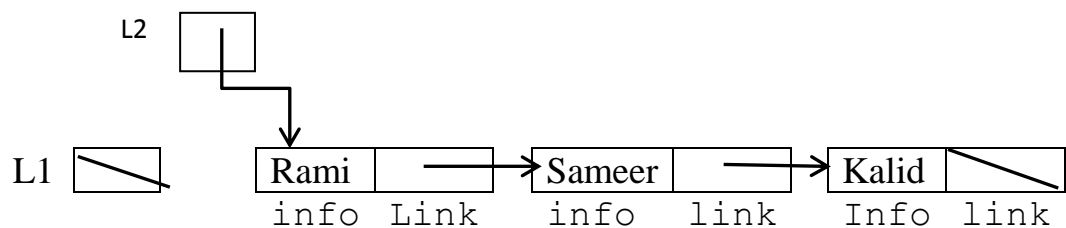
B- (L1 != null) and (L2 == null) :



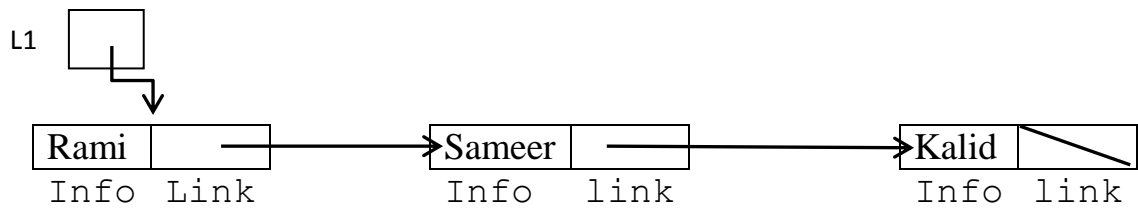
Output → L1 unchanged



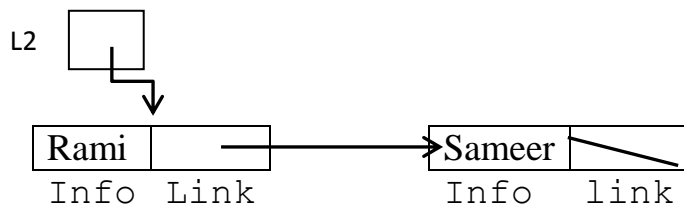
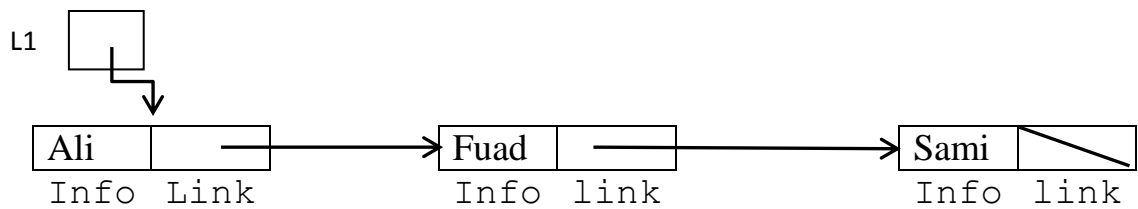
C- (L1 == null) and (L2 != null) :



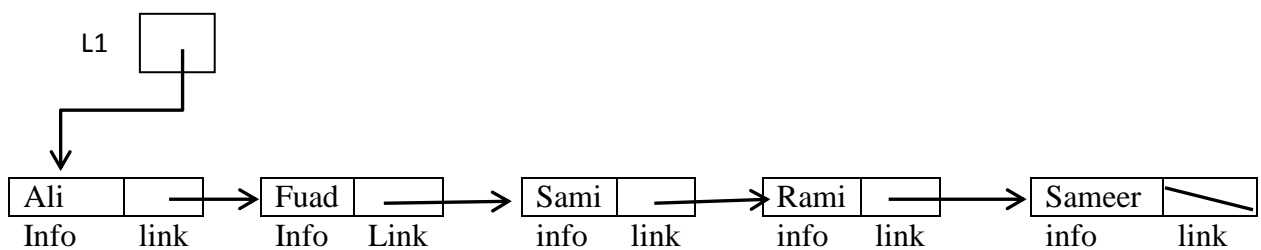
Output ➔ L1 changed to L2



D- (L1 != null) and (L2 != null) :



Output ➔ L1 changed



HomeWork :

Try to edit the following program and test it, then do the following :

- Write a method, which inserts a new Node at Head.**
- Write a method, which deletes the Node from Head.**
- Write a method** `concat (L1, L2)`

```
public class ListNode
{
    protected String info;
    protected ListNode link;

    public ListNode()
    {
        this(null,null);
    }

    public ListNode(String info , ListNode link)
    {
        this.info = info;
        this.link = link;
    }

    public void setInfo(String info)
    {
        this.info = info;
    }

    public String getInfo()
    {
        return this.info;
    }

    public void setLink(ListNode link)
    {
        this.link = link;
    }

    public ListNode getLink()
    {
        return this.link;
    }
}
```

```

public class LinkedList
{
    ListNode firstNode;
    int length;

    public int size()
    { return length;}

    public void print()
    { ListNode N;
      N = firstNode;

      while (N != null)
      { System.out.println(N.info);
        N = N.link; }
      System.out.println("NUMBER OF info IN THE LIST = "+size());
    }

    public void inserNewNode(String info)
    {
        if (length == 0) return ;
        ListNode newNode = new ListNode();

        newNode.info = info;
        newNode.link = firstNode.link;

        firstNode.link = newNode;
        length++;
    }

    public void insertLast(String info)
    {
        ListNode N = new ListNode();

        N.info = info;
        N.link = null;

        if (firstNode == null)
            firstNode = N;
        else
        { ListNode P = firstNode;
          while (P.link != null)
            P = P.link;
          P.link = N; }
        length++;
    }

    public void deleteLast()
    {
        ListNode prev , current;

        if (firstNode != null)
        {
            if (firstNode.link == null)
            {
                firstNode = null;
            }
        }
    }
}

```

```

        length--;
    }
    else
    {
        prev = firstNode;
        current = firstNode.link;
        while (current.link != null)
        {
            prev = current;
            current = current.link;
        }
        prev.link = null;
        length--;
    }
}
}
}

```

```

import javax.swing.*;
class Test
{
    public static void main(String args[])
    {
        // ListNode x = new ListNode("ali",null);
        // ListNode y = new ListNode();
        // ListNode z = new ListNode("fuad",null);
        // ListNode n = new ListNode();

        // ++++++
        // FIRST IDEA TO CREATE A LIST WITH 3 ELEMENTS AND INSERT
        // NEW ELEMENT AFTER THE FIRST ELEMENT :

        //y.info = "ahmed";
        //n.info ="sami";
        //x.link = y;
        //x.link.link = z;

        //Insert new Node
        //n.link=x.link;
        //x.link = n;

        // ++++++
        // SECOND IDEA TO CREATE A LIST WITH 3 ELEMENTS AND INSERT
        // NEW ELEMENT AFTER THE FIRST ELEMENT :

        // y.setInfo("ahmed");
        // n.setInfo("sami");
        // x.setLink(y);
        // x.link.setLink(z);

        // Insert new Node
        // n.setLink(x.link);
        // x.setLink(n);
    }
}

```



```

//System.out.println(x.getInfo()+"\n"+x.link.Info+"\n"+x.link.
//link.Info+"\n"+
//x.link.link.link.Info);

// ++++++
// THIRD IDEA TO CREATE A LIST WITH 3 ELEMENTS :

    ListNode M = new ListNode();

    // CREATE A LINKED LIST WITH ONE ELEMENT :
    M.info = "Ali";
    M.link = null;

    // APPEND TWO ELEMENTS TO THE LIST :
    M.link = new ListNode();
    M.link.info = "Sami";
    M.link.link = new ListNode();
    M.link.link.info = "Fuad";
    M.link.link.link = null;

    LinkedList L = new LinkedList();

    L.firstNode = M;
    L.length = 3;

//System.out.println(M.getName()+"\n"+M.link.name+"\n"+M.link.
//link.name);

// OUTPUT THE CONTENTS OF THE LIST TO THE SCREEN BEFORE INSERT
// NEW ELEMENT:
    L.print();

    String inserM = JOptionPane.showInputDialog("ENTER 1 TO AFTER
OR\n 2 TO INSER AT END OR\n 3 TO DELETE AT END ");

    int n ;

    n = Integer.parseInt(inserM);

    if (n == 1)
    { String newElement=JOptionPane.showInputDialog("ENTER NEW
ELEMENT ");
      L.inserNewNode(newElement);
    }
    else if(n == 2)
    {String newElement=JOptionPane.showInputDialog("ENTER NEW
ELEMENT ");
      L.inserLast(newElement);
    }
    else L.deleteLast();

// OUTPUT THE CONTENTS OF THE LIST TO THE SCREEN AFTER INSERT
// NEW ELEMENT:
L.print();
}
}

```


3.3 Other Linked Data Structures :

- ▶ Two Way Linked Lists (Double Linked Lists)
- ▶ Two way Ring (Circular)
- ▶ Binary Trees
- ▶

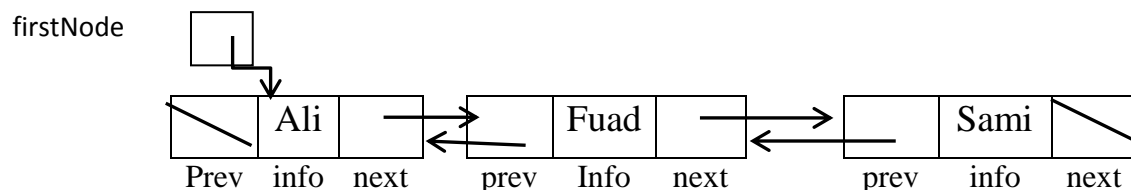
Two Way Linked Lists (Double Linked Lists) :

▶ *Diagramming Representation :*

Empty List :

• firstNode 

• NON-Empty List :



▶ *Definition of Double Linked List :*

Any DLL should have at least following properties :

- 1- Every Node contains the needed information(*same data type*) to be stored in memory and two references (pointers), one (*next*) to next Node , if there is any one, and the other (*prev*) to the previous Node , if there is any one.
- 2- The address of the first node stored in extra cell (by example *firstNode*) , if (*firstNode == null*) → Empty List.
- 3- It is necessary to determine, which Node will be the last Node with *next* value of last Node equal to null [**end of list**], and which Node will be the first Node with *prev* value equal to null.

► ***Declaration of Double Linked List : (Abstract Data type ADT)***

```
public class ListNode
{
    String info ;
    ListNode prev ; // SelfReferential Pointer1
    ListNode next ; // SelfReferential Pointer2

    // Constructors
    public ListNode ( )
    { this(null,null,null); }

    public ListNode(String info, ListNode prev , ListNode next)
    { this.info = info ;
      this.prev = prev ;
      this.next = next ;
    }
    .....
}
```

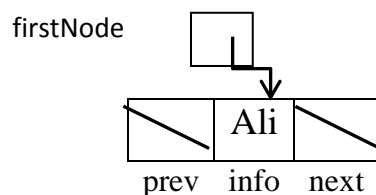
► ***Manipulation Double Linked List :***

- Creation a new Double Linked Lists
- Printing
- Inserting new node (at Head , Between , At Last)
- Deleting a node (from Head , Between , from Last)
- Splitting up the list
- Concatenation two lists
- Updating the contents of the information in any node
-

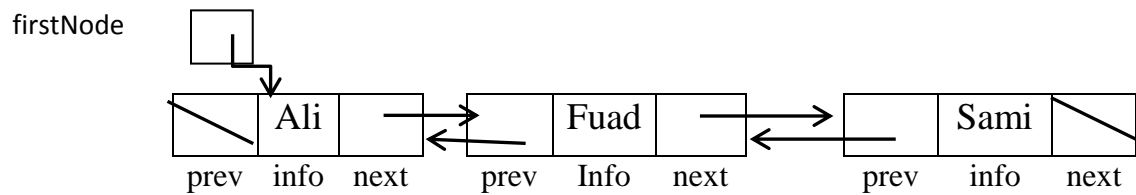
Examples :

- Creating a Double Linked List (Naïve Approach)

```
ListNode firstNode = new ListNode("Ali",null,null); //3 actions
```



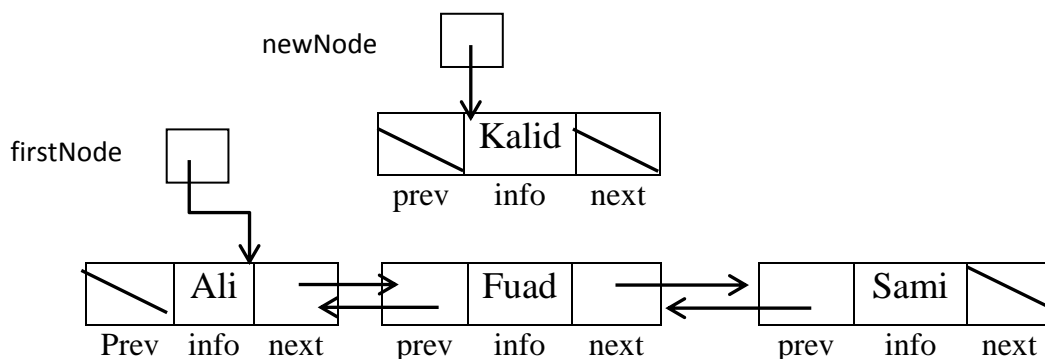
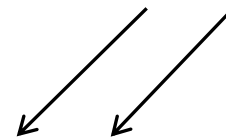
- Printing the value "Sami" :



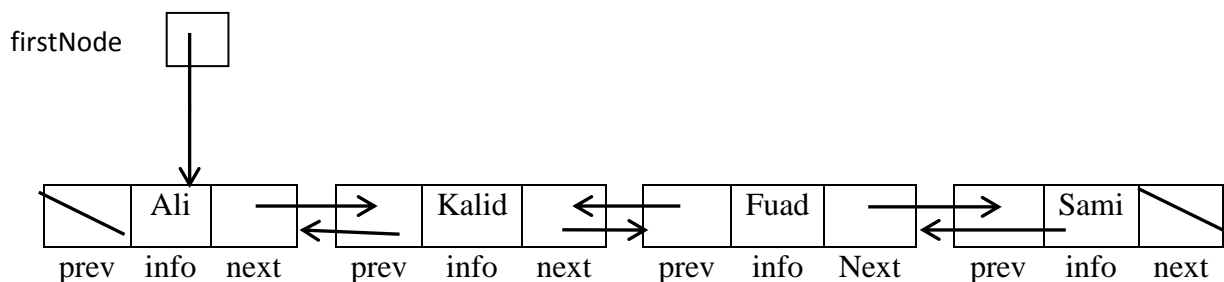
```
S.O.P. (firstNode.next.next.info) ; // OR
S.O.P. (firstNode.next.next.prev.next.info); //OR
.....
```

-Insert new Node with value "Kalid" between "Ali" and "Fuad" :

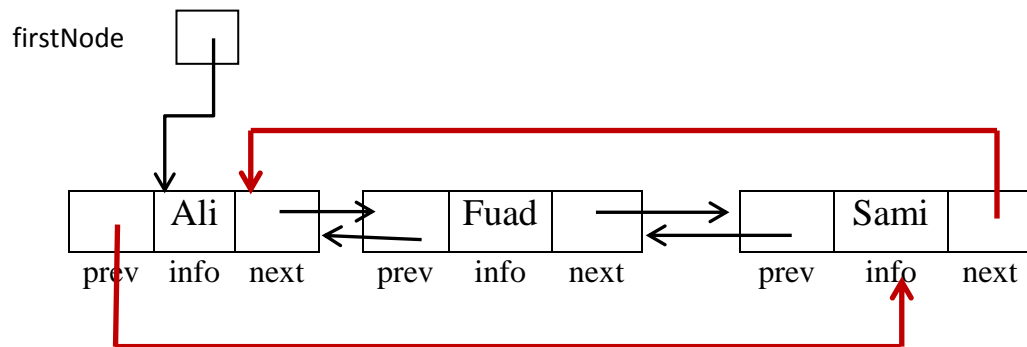
```
ListNode newNode = new ListNode("Kalid",null,null) ;
newNode.next = firstNode.next;
newNode.prev = firstNode;
firstNode.next = newNode;
newNode.next.prev = newNode ;
```



Output ➡



Two Way Ring (Circular) :



- **Printing the value "Ali" :**

S.O.P.(firstNode.info); // OR
S.O.P.(firstNode.next.prev.info) ; // OR
S.O.P.(firstNode.next.next.next.info); // Or
S.O.P.(firstNode.prev.prev.prev.info);

NOT PRACTICAL

CH4 : Recursion :

This chapter is an introduction that aims at converging the heart of the concepts. It also gives you opportunities to build your skills for dealing with recursion . Any method, which represents repeated work, could be implemented in only two ways (*Iteration [Looping] or Recursion*).

Examples :

1- Factorial Example :

$\text{fact}(0) = 0! = 1$, $\text{fact}(1) = 1! = 1$, $\text{fact}(2) = 2! = 2*1 = 2$

$\text{fact}(3) = 3! = 3*2*1 = 6$, $\text{fact}(4) = 4! = 4*3*2*1 = 24$

....

• Iteration (Looping) :

```
int fact ( int n )
{
    int f = 1;

    for ( int i = 2 ; i <= n ; ++i )
        f *= i ;

    return f ;
}
```

Tracing : method Call

$z = \text{fact}(0) = 1$

$z = \text{fact}(1) = 1$

...

$z = \text{fact}(4) = 24$

n	?	f	1	i	2
				fact	?

• Recursion1 :

```
int fact ( int n )
{ if ( n <= 1 ) // ((n==0) || (n==1))
    return 1;
  else
    return n*fact(n-1);
}
```

$z = \text{fact}(0) = 0$

$z = \text{fact}(1) = 1$

...

$z = \text{fact}(4) = 24$

n	?
---	---

fact	?
------	---



1

2

3

● Recursion2 :

```
int fact ( int n )
{
    int x , y ;

    if ( n <= 1 )
        return 1;
    else
    {   x = n-1 ;
        y = fact (x) ;
        return n*y;
    }
}
```

Try alone :

Tracing : method Call

z = fact(0)=1

z = fact(1) = 1

... .

z = fact(4) = 24 ?

2- SumSquares Example (between two integers) :

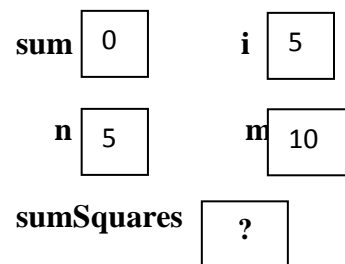
$$\begin{aligned}\text{sumSquares} (5 , 10) &= 5*5 + 6*6 + 7*7 + 8*8 + 9*9 + 10*10 = \\ &= 25 + 36 + 49 + 64 + 81 + 100 = 355\end{aligned}$$

● Iteration (Looping) :

```
int sumSquares(int n , int m )    // n<=m
{ int i , sum ;

    sum = 0 ;
    for ( i = n ; i <= m ; ++i )
        Sum += i*i;
    return sum ;
}
```

Tracing : method Call



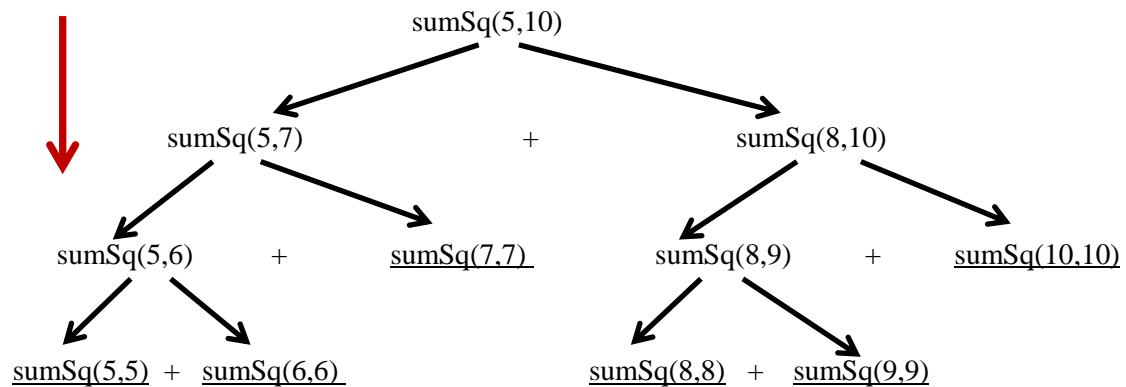
● Recursion :

```
int sumSquares( int n , int m )    // n<= m
{
    int middle ;

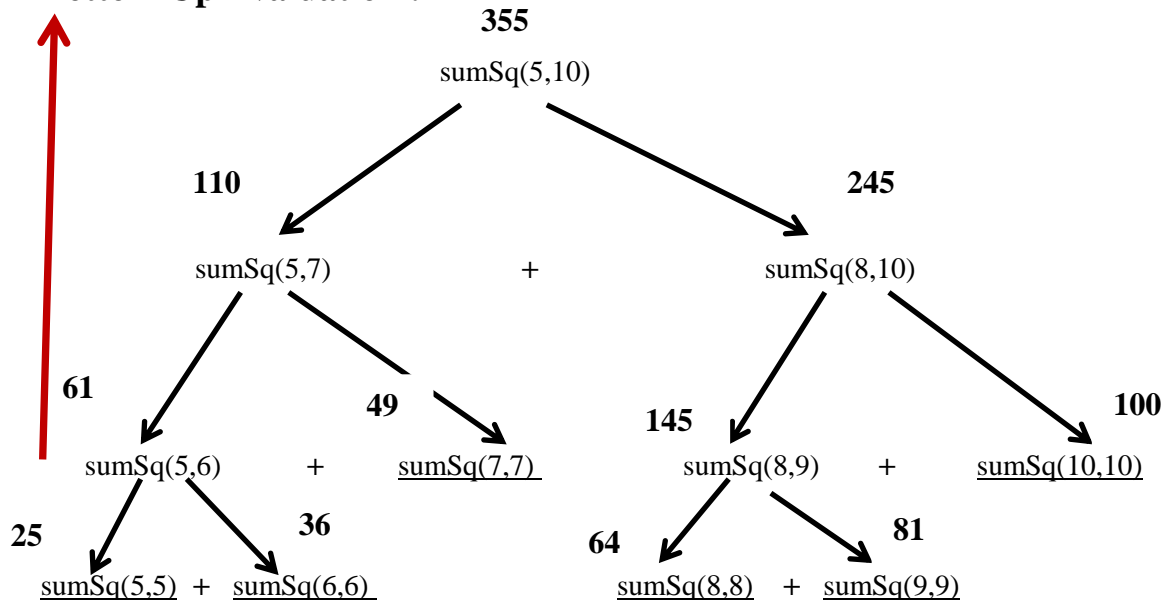
    if ( n == m )
        return m*m;
    else
    { middle = (n+m)/2 ;
      return sumSquares(n,middle) + sumSquares(middle+1,m) ;
    }
}
```


Call Trees and Traces :

Top Down Call :



Bottom Up Evaluation :



3- Fibonacci number Example : (Recursion)

n	0	1	2	3	4	5	6	7	8
fib(n)	0	1	1	2	3	5	8	13	21

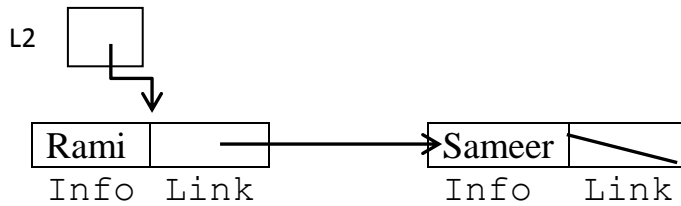
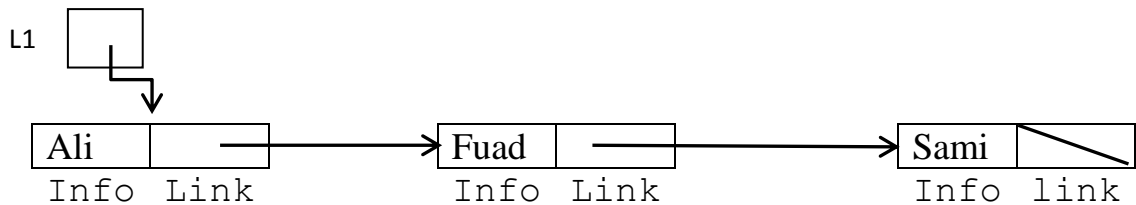
```
int fib(int n)
{ if ((n == 0) || (n == 1))
  return n;
  else return fib(n-1) + fib(n-2);
}
```

Evaluate the following Statement :

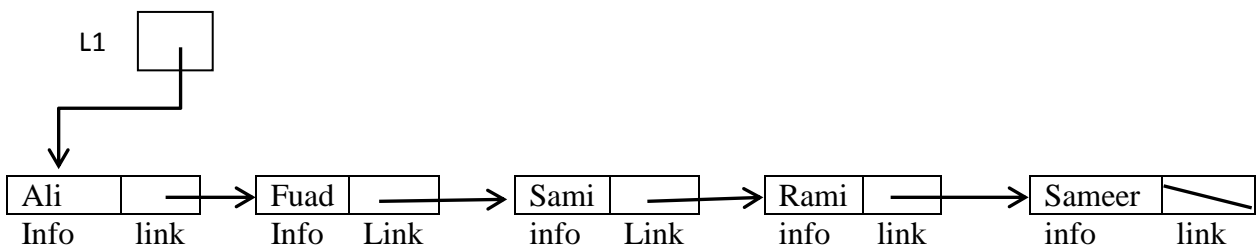
$Z = \text{fib}(4)$

4- Concatenation Two Lists (L1 and L2) :

```
ListNode concat( ListNode L1 , ListNode L2)
{
    if ( L1 == null)
        return L2;
    else
    { L1.link = concat(L1.link , L2);
      return L1;
    }
}
```

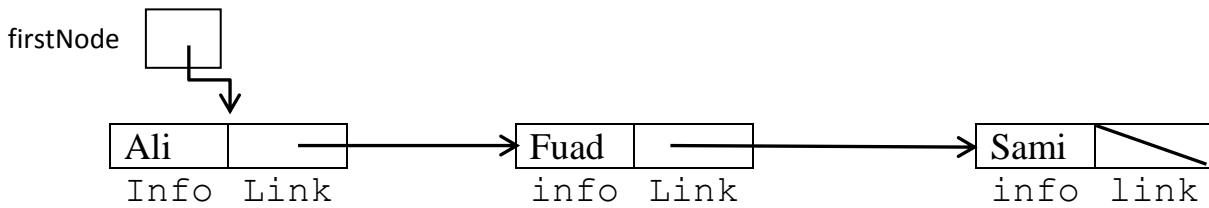


Output ➔ L1 changed

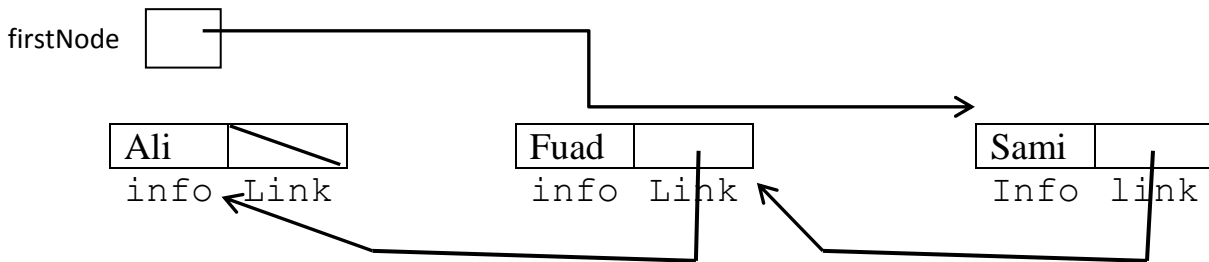


5- Reverse the List :

Input →



Output →



```
a- void reverse( )
    {   firstNode = reverse1(firstNode); }
```

```
b- ListNode reverse1 (ListNode L)
{
    if ( L == null )
        return null;
    else
    {   ListNode head = L;
        ListNode tail L.link;

        head.link = null ;
        return concat( reverse1( tail ) , head ) ;
    }
}
```

- Try to implement the method ***print*** () recursively.

CH5● Linear Data Structures (Stacks , Queues)

5.1 Stacks :

- A stack is a special case of Single (Linear) Linked Lists, where it works under the principle of *LIFO* (**Last In First Out**). It means :
 - Inserting a new Object at head of the Single Linked List and
 - Deleting an Object from head of the Single Linked List.

- A stack , S , is a sequence of items (Objects) on which the following operations are defined :
 1. Construct an initially empty stack, S.
 2. Determine whether or not the stack, S , is *empty*
 3. *Push* (insert new object at the top (Head) of stack.
 4. If S is NonEmpty, *pop* an object from the top (Head) of stack , S.
 5. If S is NonEmpty, *peek* at the top (Head) of stack S by reading a copy of top object of S without deleting it.

● Implementing of Stack :

```
class StackNode // Stack ADT
{
    Object item;
    StackNode link;

    // constructors
    .....
}

class Stack // methods ...
{
    StackNode topNode; //firstNode

    //Methods to manipulate Stack
    boolean empty()
    { return (topNode == null; }

    void push (object X) // insert at top (Head)
    { StackNode newNode = new StackNode();
      newNode.item = X;
      newNode.link = topNode;
      topNode = newNode;
    }

    Object pop () // Delete from top (Head)
    { if (topNode == null ) // if (empty())
      return null;
      else
      { StackNode tempNode = topNode ;
        topNode = topNode.link ;
        return tempNode.item;
      }
    }

    Object peek() // Make copy of top Object (Head)
    { if (topNode == null )
      return null;
      else return topNode.item;
    }
    ....
} //End of class Stack

Class Test //main Program
{
    ....main()
    ....
    Stack S;

    S.empty();
    S.push(X);
    Y = S.pop();
    Z = peek() ;
    ...
}
```

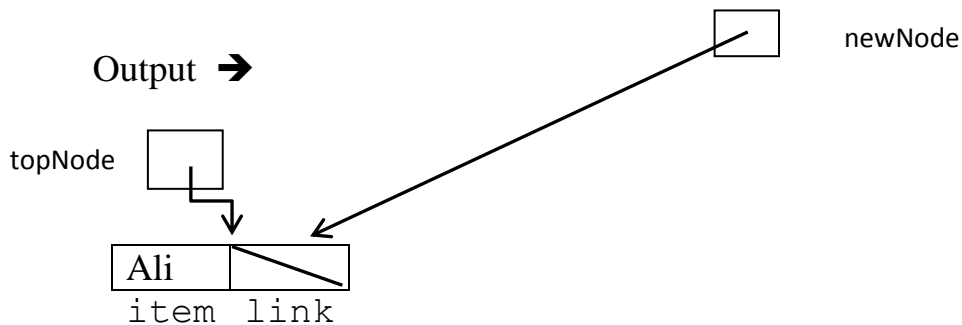
• Diagramming Representation of push() :

S.push(X) ;

1st.Case : The Stack is empty

Ali X

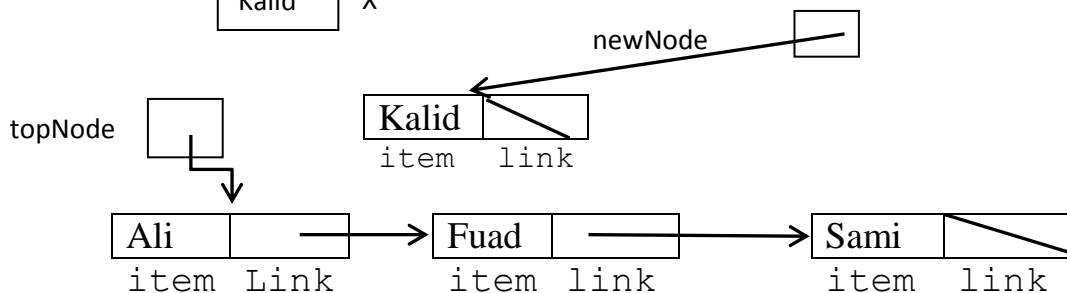
topNode  → push Ali →



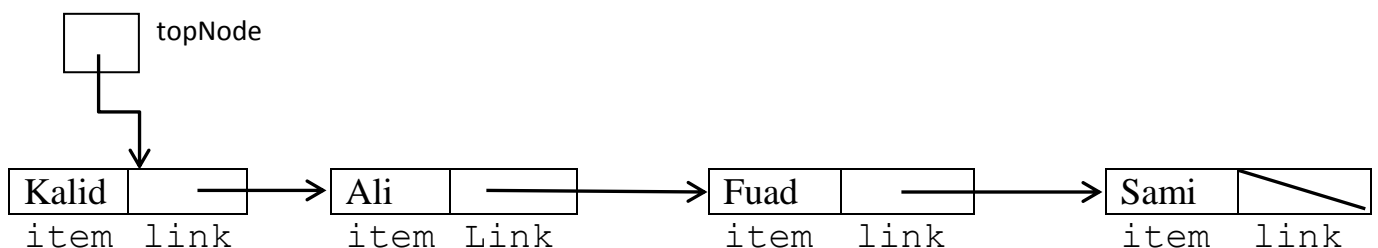
2nd. Case : The list contains many Nodes

Insert (*push*) "Kalid" at Head :

Kalid X



Output →



• Diagramming Representation of pop() :

$Y = S.pop();$

1st.Case : The Stack is empty → Nothing to do

2nd. Case : The list contains only one Node :

Input →

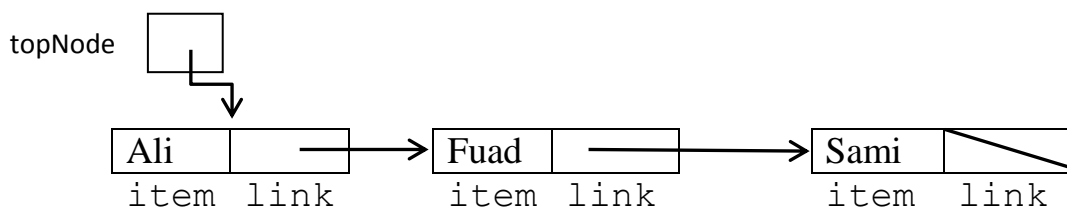


Output → Empty stack : `topNode` `Y`

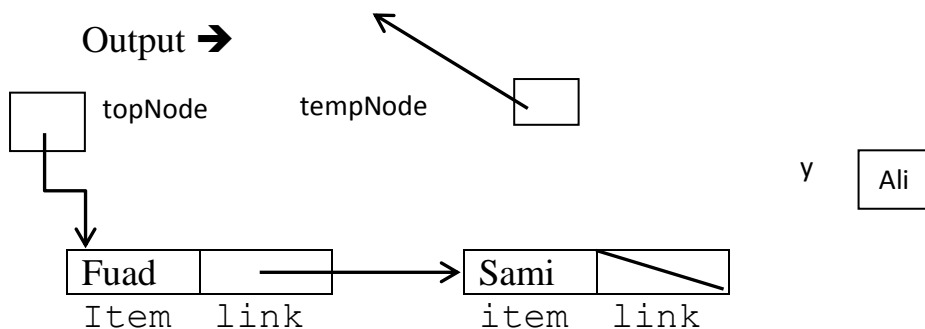
3rd. Case : The list contains many Nodes

Delete *pop()* from Head :

Input →



Output →



● Applications of Stack :

- 1- Converting infix Expressions to postfix Expressions
- 2- Evaluation of postfix Expressions
- 3- Check up for balanced parenthesis and Brackets.
- 4- Evaluation of Recursion

1- Converting infix Expressions to postfix Expressions :

Lukas's Theory to represent a math. Expression :-

- 1- $x+y$ (*infix Expression*)
- 2- $xy+$ (*postfix Expression*)
- 3- $+xy$ (*prefix Expression*)

Examples : Approach to convert infix Expression to postfix Expression

We discuss only Expressions containing following Symbols :

- Operands (Variables) like : $a, b, c, \dots, A, B, C, \dots$
- Operators (Operations) like : $+, -, *, /, \%, ^$ (Power)
- Parenthesis or (Brackets) like : $(,), [,], \{, \}$
- Special Symbols for the begin and end of Expressions like : $|, \vdash$

► **Idea :** *Converting higher operator to first from infix to postfix representation and then removing all parenthesis and brackets from Expression.*

Ex1 : Given $a+b*c$

(*infix*): $a+b*c \longrightarrow a+(bc*) \longrightarrow a(bc*)+ \longrightarrow abc*+ (Postfix)$

Ex2 : $(a+b)*c$

(*infix*): $(a+b)*c \longrightarrow (ab+)*c \longrightarrow (ab+)c* \longrightarrow ab+c* (Postfix)$

Ex3 : $a+b-c$

(*infix*): $a+b-c \longrightarrow (ab+)-c \longrightarrow (ab+)c- \longrightarrow ab+c- (Postfix)$

Algorithm to convert an infix Expression to postfix Expression :

Enclose Expression between $\{$ and $\}$ symbols then reading Expression from left to right :

- 1- If reading $\{$ symbol \rightarrow *push*($\{$) into Stack.
- 2- If reading an operand \rightarrow output operand to the output Stream.
- 3- If reading left Bracket or Parenthesis { , [, (\rightarrow *push* into Stack.
- 4- If reading an operator \rightarrow compare incoming operator with the top of Stack using *peek*() :
 - (a) If top of Stack one of $\{$, { , [, (\rightarrow *push* incoming operator into Stack.
 - (b) If top of Stack an operator with lower precedence than the incoming operator \rightarrow *push* incoming operator into Stack.
 - (c) If top of Stack an operator with same or higher than the incoming operator \rightarrow *pop*() top of Stack and then again with (a) , (b) or (c).
- 5- If reading right Bracket or Parenthesis } ,] ,) \rightarrow
 - *pop*() all of stack until reaching left Bracket or Parenthesis { , [, (
 - { } , [] , () disappeared.
- 6- If reading $\}$ symbol \rightarrow
 - *pop*() all of stack until reaching $\{$ symbol
 - $\{$, $\}$ disappeared

Examples :

- 1- $A+B*C$
- 2- $(A+B)*C$
- 3- $A*B*C$
- 4- A^B^C (إشارة ^ تعبير عن الاس)
- 5- $(A*(C/D)+B)^2$

► Solutions :

1- $A+B*C$:

Enclose Expression between $\{$ and $\}$ →

$\{ A+B*C \}$ then :

Input Stream	Stack	Output Stream
$\{$	$\{$	
A	$\{$	A
+	$\{ +$	
B	$\{ +$	B
*	$\{ * +$	
C	$\{ * +$	C
$\}$	Empty	$* +$

Output → $ABC*+$

Incoming Operator

Top of Stack



2- $(A+B)*C$:

Enclose Expression between $\{$ and $\}$ →

$\{ (A+B)*C \}$ then :

Input Stream	Stack	Output Stream
$\{$	$\{$	
($\{ ($	
A	$\{ ($	A
+	$\{ + ($	
B	$\{ + ($	B
)	$\{$	+
*	$\{ *$	
C	$\{ *$	C
$\}$	Empty	$*$

Output → $AB+C*$

Incoming Operator

Top of Stack



3- $A * B * C$:

Enclose Expression between $|$ and $| \rightarrow$

$| A * B * C |$ then :

Input Stream	Stack	Output Stream
$ $	$ $	
A	$ $	A
*	* $ $	
B	* $ $	B
*	* $ $	*
C	* $ $	C
$ $	Empty	*

Output $\rightarrow AB * C *$

Incoming Operator

Top of Stack



4- $A^B \wedge C$: (\wedge (blue Power) will be evaluated before the (black Power) $^$)

It means : \wedge (blue Power) is higher than (black Power) $^$

Explaining : Suppose $A = 2$, $B = 3$ $C = 2$

$\rightarrow A^B \wedge C = 2^3 \wedge 2 = 2^9 = 512$ (CORRECT ANSWER)

But Not : $2^3 \wedge 2 = (2^3)^2 = 8^2 = 64$ (WRONG!!!)

Enclose Expression between $|$ and $| \rightarrow$

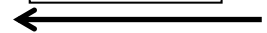
$| A^B \wedge C |$ then :

Input Stream	Stack	Output Stream
$ $	$ $	
A	$ $	A
$^$	$^ $	
B	$^ $	B
\wedge	$\wedge^ $	
C	$\wedge^ $	C
$ $	Empty	$\wedge^$

Output $\rightarrow ABC \wedge^$

Incoming Operator

Top of Stack



5- $(A*(C/D)+B)^2$:

Enclose Expression between $\{$ and $\}$ \rightarrow

$\{ (A*(C/D)+B)^2 \}$ then :

Input Stream	Stack	Output Stream
$\{$	$\{$	
$($	$(\{$	
A	$(\{$	A
*	$*(\{$	
$($	$*(\{$	
C	$*(\{$	C
/	$/(*(\{$	
D	$/(*(\{$	D
)	$*(\{$	/
+	$+*(\{$	*
B	$+*(\{$	B
)	$\{$	+
$^$	$^{\{$	
2	$^{\{$	2
$\}$	Empty	$^$

Output $\rightarrow ACD/*B+2^$

Incoming Operator

Top of Stack



2- Evaluation of postfix Expressions :

Algorithm :

Repeating:

if (reading an operand) \rightarrow push into stack ;

else (reading an operator) \rightarrow

{ pop the first two tops operands to memory;

and evaluating the operator (swap operands if

operator NON-Commutative one of [- , / , % , ^])

using the two just popped operands;

}

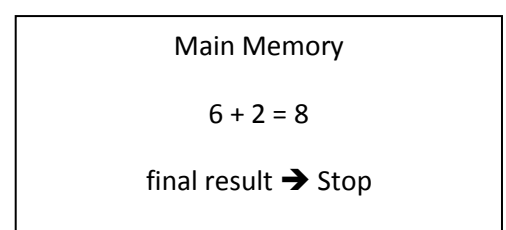
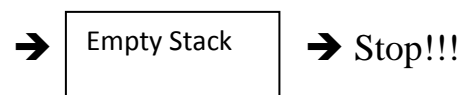
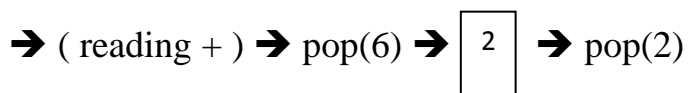
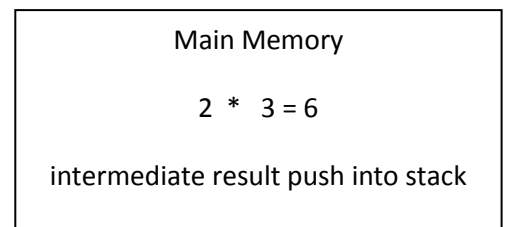
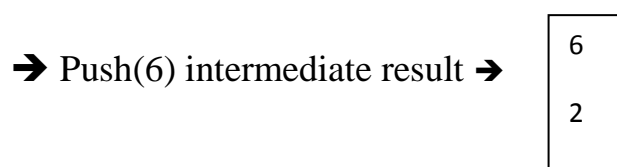
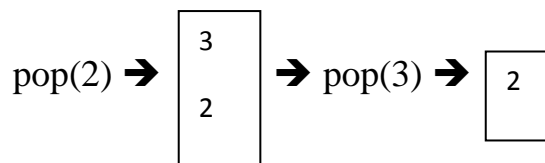
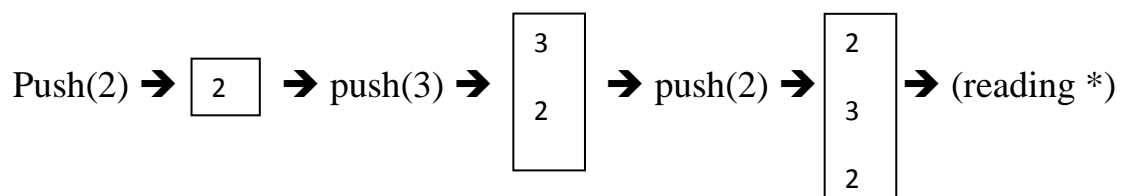
until empty Stack;

Examples :

Ex1 :

► Infix: $A+B*C = 2+3*2 = 2+6 = 8$, where $A = 2$, $B = 3$, $C = 2$

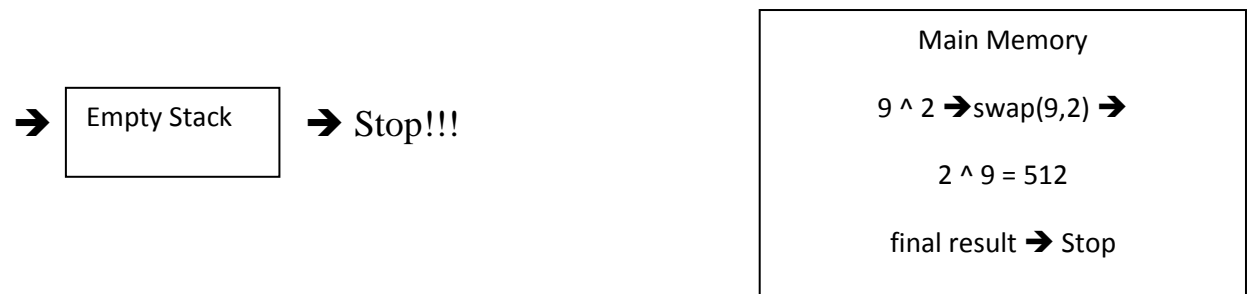
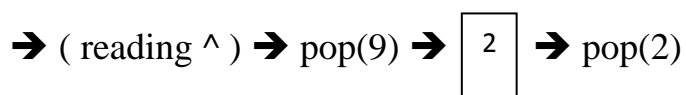
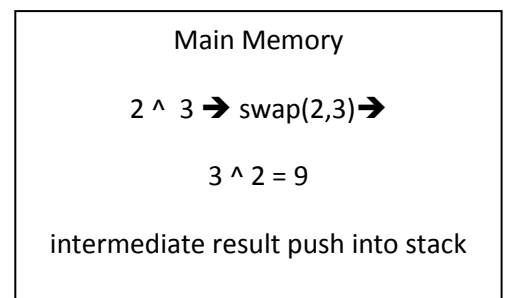
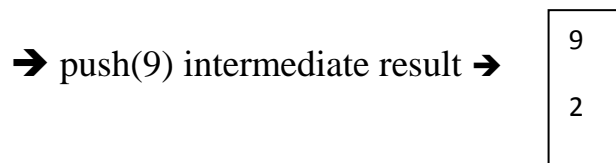
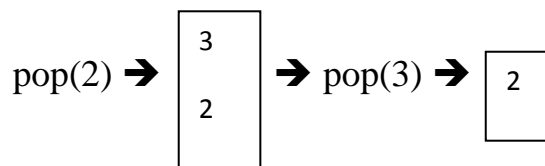
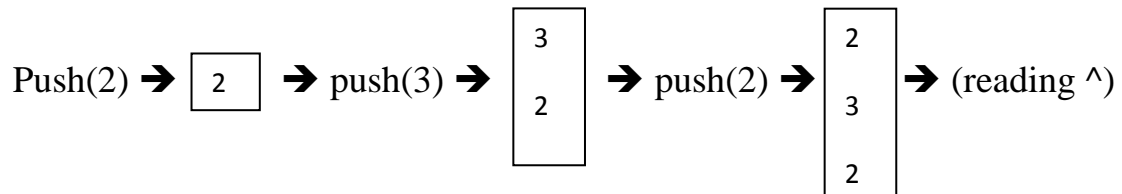
Postfix : $ABC*+= 2\ 3\ 2\ *\ +$, where $A = 2$, $B = 3$, $C = 2$



Ex2 :

► Infix : $A^B C = 2^3 2 = 2^{(3^2)} = 2^9 = 512$, where $A=2, B=3, C=2$

Postfix : $ABC^{^^} = 232^{^^}$



3- Check up for balanced parenthesis and Brackets :

Algorithm :

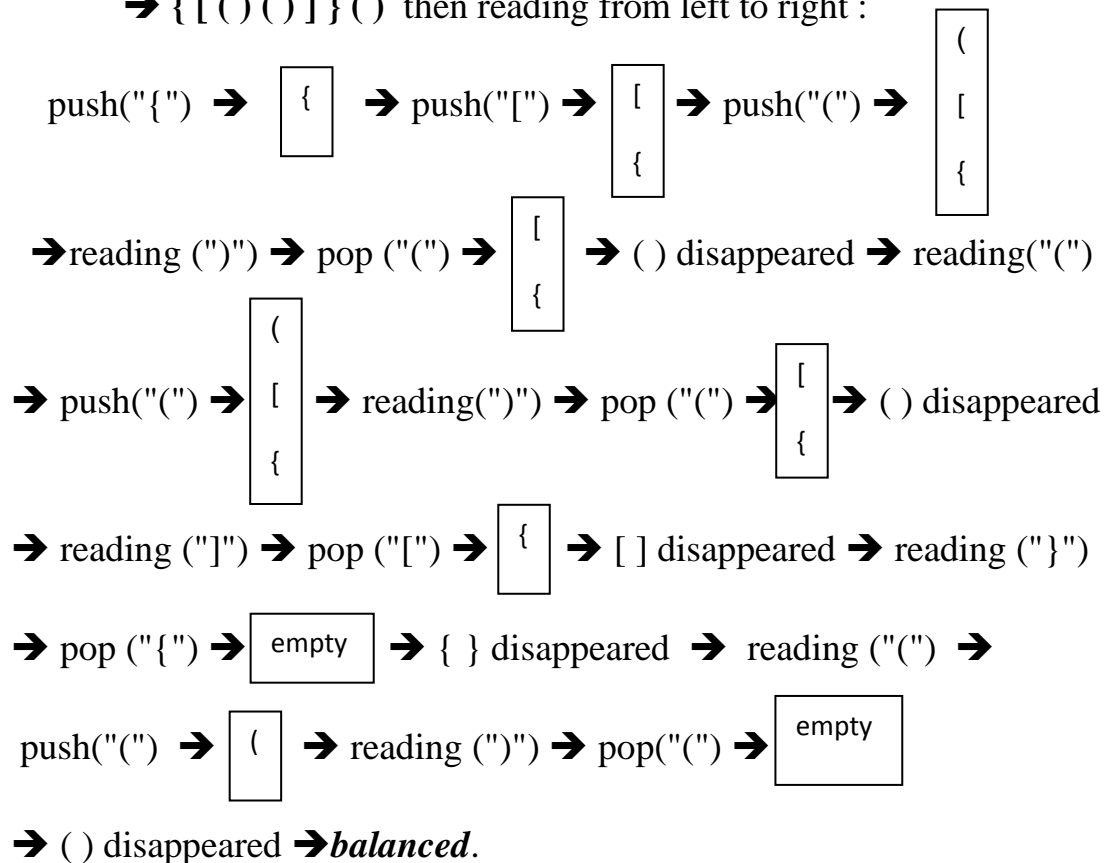
- 1- Ignore all (operands , operators) excepting only parenthesis or Brackets.
 - 2- Repeating:
 - if* reading *left* parenthesis "(" or *left* Bracket "[", "{"
 - push into stack ;
 - else* reading *right* parenthesis ")" or *right* Bracket "]", "}"
 - pop the top of stack and comparing with incoming right parenthesis or right bracket, **if** same shape then disappeared () , [] **or** { }.
 - else** error!!! (Exit algorithm) .
- until empty Stack;

Examples :

Ex1 :

► Infix expression : $\{a^2 - [(c - d)^2 + (e - f)^2]\} * \sin(x - y)$

→ { [() ()] } () then reading from left to right :



Ex2 :

► Infix expression : (a * sin (x) - A[i,j] / (cos (x) + { p - q) / { x - y })

→ (() [] (() { }) { }) then reading from left to right :

push("(") →

(

 → push("(") →

(
(

 → reading ("") → pop("(")

→

(

 → () disappeared → reading("[") → push("[") →

[
(

→ reading("]") → pop("[") →

(

 → [] disappeared

→ reading "(" → push "(" →

(
(

 → reading "(" → push "(" →

(
(
(

→ reading ("") → pop("(") →

(
(

 → () disappeared → reading ("{"

→

{
(
(

 → reading ("") → pop ("{" →

(
(

 {) **no matching**

→ ERROR (STOP reading) → Exit Algorithm .

4- Evaluation of Recursion :

Example :

Evaluate the following statement using stack :

$T = \text{fact}(4);$ Where **fact()** defined as follows :

```
int fact ( int n )
{ int x , y ;
```

```
  if (( n == 0 ) || ( n == 1 ) )
```

```
    return 1 ;
```

```
  else
```

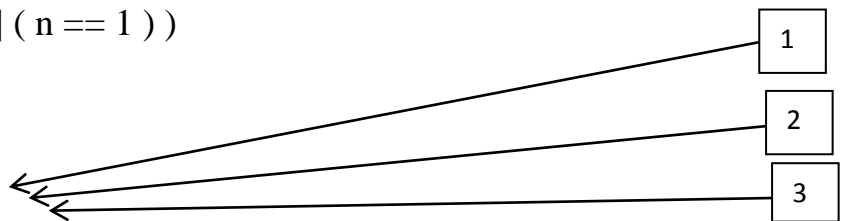
```
    { x = n - 1 ;
```

```
      y = fact(x) ;
```

```
      return n * y ;
```

```
    }
```

```
  }
```

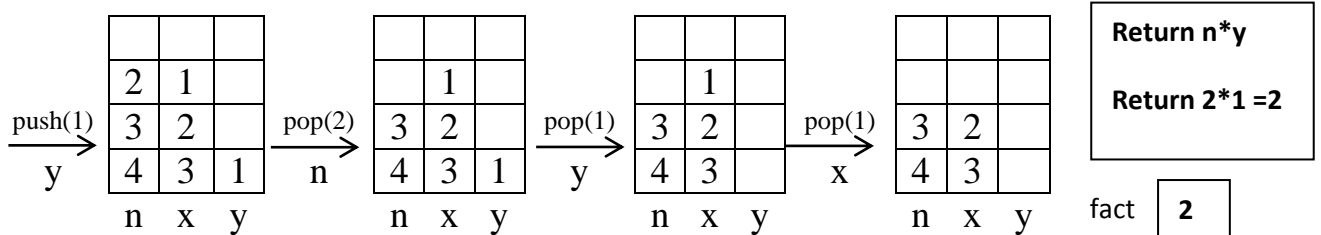
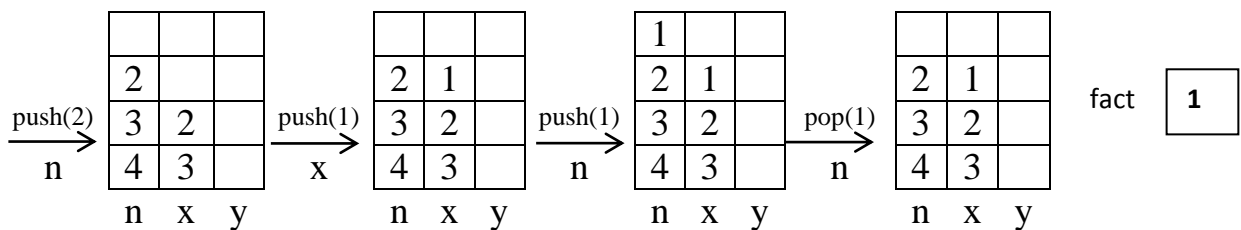
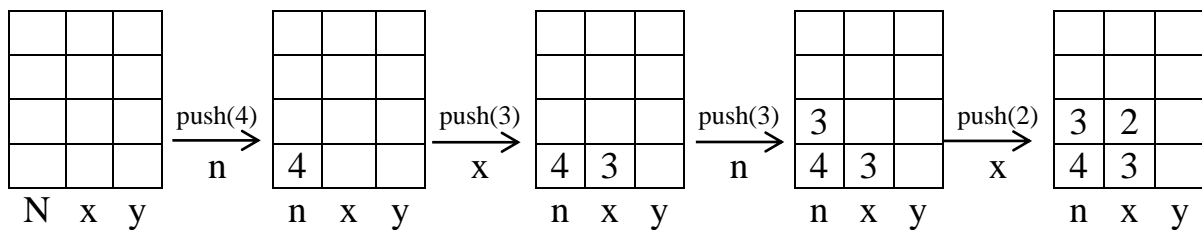


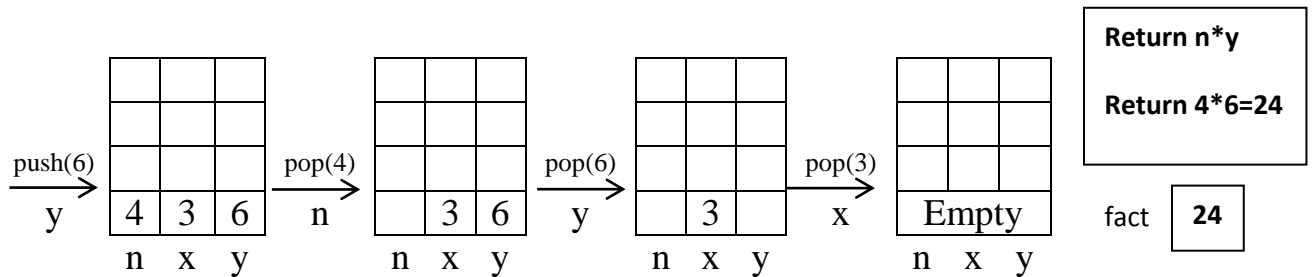
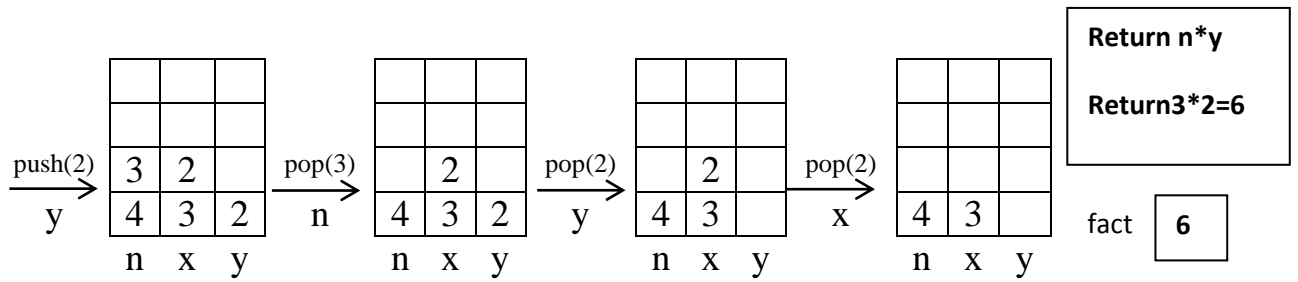
► Solution :

1- For each formal parameter define a stack.

2- For each local variable define a stack.

➔ There are 3 stacks (in Example **n** : formal and **x** , **y** : local)





Other Examples :

1- Evaluate the following statement using stack :

$T = \text{fib}(3)$; Where **fib** () defined as follows :

```
int fib(int n)
{ if ((n == 0) || (n == 1))
  return n;
else
  return fib(n-1) + fib(n-2);
}
```

2- Evaluate the following statement using stack :

$T = \text{fact}(4)$; Where **fact** () defined as follows :

```
int fact(int n)
{ if ( n <= 1 )
  return 1;
else
  return n*fact(n-1);
}
```

3- Evaluate the following statement using stack :

T=sumSquares (4,6); Where **sumSquares ()** defined as follows :

```
int sumSquares( int n , int m )
{
    int middle ;

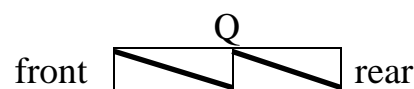
    if ( n == m )
        return m*m;
    else
        {middle = (n+m)/2 ;
         return sumSquares(n,middle)+sumSquares(middle+1,m);
        }
}
```

5.2 Queue :

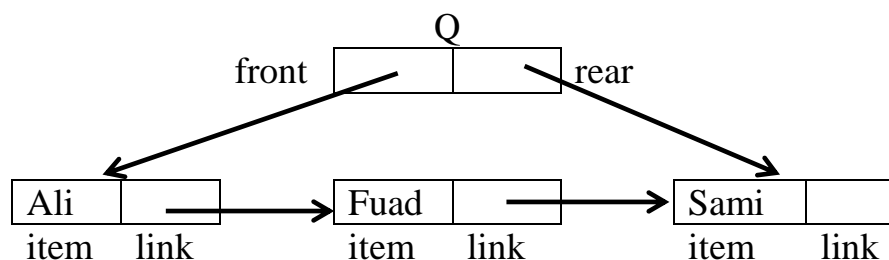
- A queue is a special case of Single (Linear) Linked Lists, where it works under the principle of *FIFO* (**F**irst **I**n **F**irst **O**ut). It means :
 - Inserting a new Object at last of the Single Linked List and
 - Deleting an Object from head of the Single Linked List.
- A queue , Q , is a sequence of items (Objects) on which the following operations are defined :
 1. Construct an initially empty queue , Q.
 2. Determine whether or not the queue, Q , is *empty*
 3. *Enqueue* (insert new object at the last of queue).
 4. If Q is NonEmpty, *Dequeue* (delete an object from the top of queue , Q).
- **Diagramming representation of Queue :**

Define two pointers (*front* , *rear*), where *front* points to the first Node and *rear* points to the last Node in the queue.

- Empty Queue :



- NON Empty Queue :



● Implementing of Stack :

```
class QueueNode // Queue ADT
{
    Object item;
    QueueNode link;

    // constructors
    .....
}

class Queue // methods ...
{
    QueueNode front; //firstNode
    QueueNode rear; //lastNode

    //Methods to manipulate Stack

    boolean empty()
    {
        return (front == null; } //or (front == null)&& (rear == null)

    void enqueue (object X) // insert at last
    {
        QueueNode temp = new QueueNode();
        temp.item = X;
        temp.link = null;
        if ( rear == null )
        {
            front = rear = temp;
        }
        else
        {
            rear.link = temp;
            Rear = temp;
        }
    }

    Object dequeue () // Delete from top (Head)
    {
        if (empty())
            return null;
        else
        {
            Object tempItem = front.item ;
            front = front.link;
            if (front == null)
                rear = null;

            return tempItem;
        }
    }
}

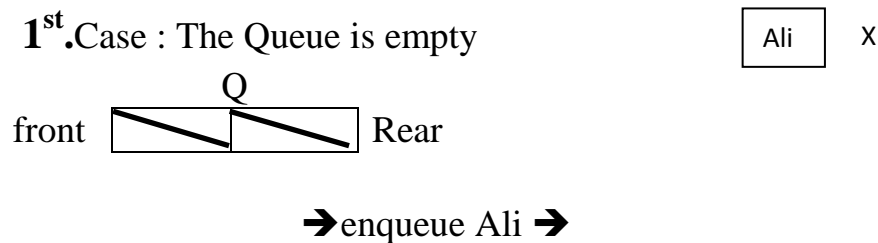
Class Test //main Program
{
    ....main()
    ...
    Queue Q;

    Q.empty();
    Q.enqueue(X);
    Y = Q.dequeue();
    ...
}
```

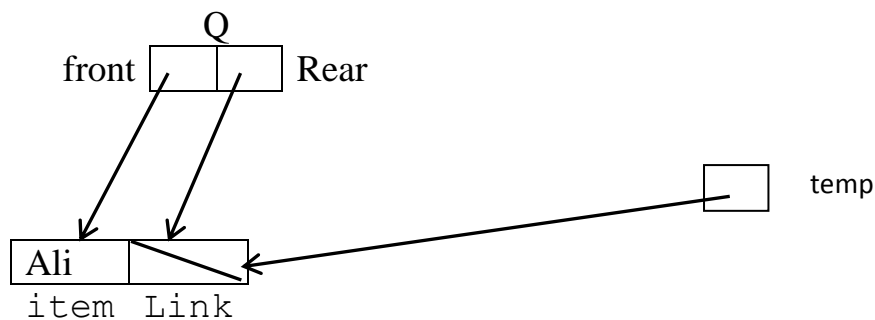
• **Diagramming Representation of enqueue() :**

Q.enqueue (X) ;

1st.Case : The Queue is empty

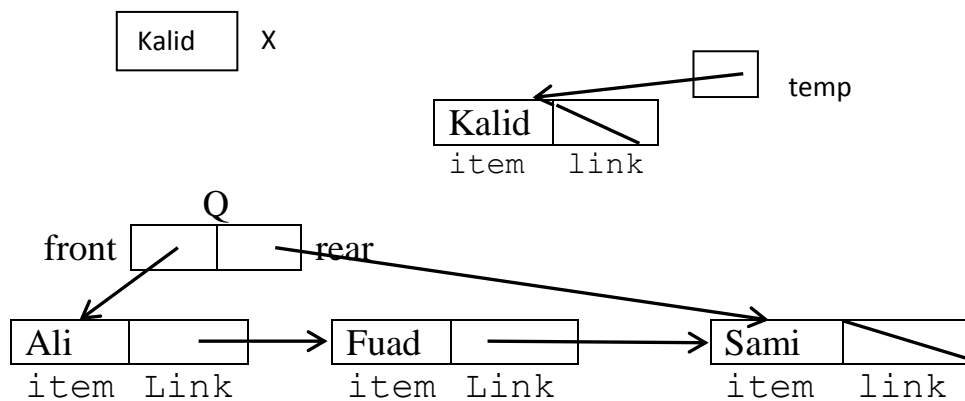


Output →

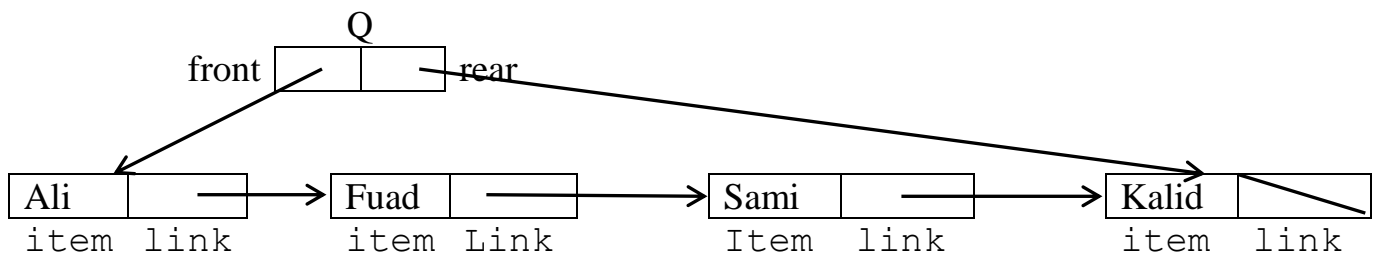


2nd. Case : The list contains many Nodes

Insert (*enqueue*) "**Kalid**" at last :



Output →



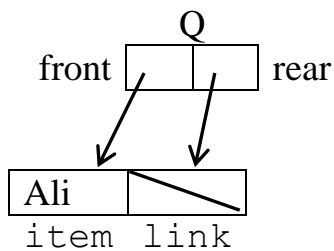
• **Diagramming Representation of dequeue() :**

$Y = Q.dequeue();$

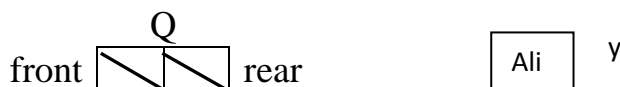
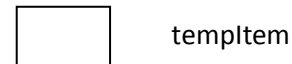
1st.Case : The Queue is empty → Nothing to do

2nd. Case : The list contains only one Node :

Input →



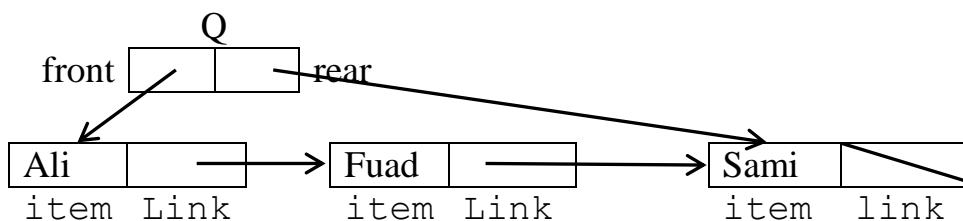
Output → Empty Queue:



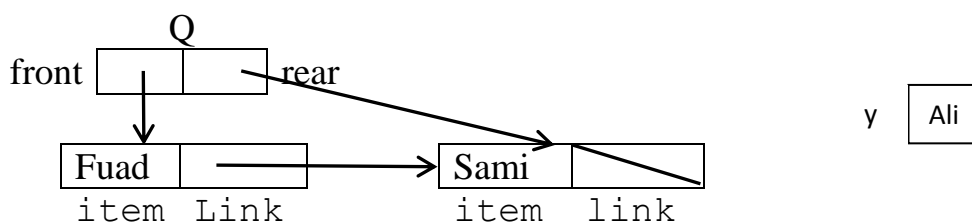
3rd. Case : The list contains many Nodes

Delete **dequeue()** from Head :

Input →



Output →

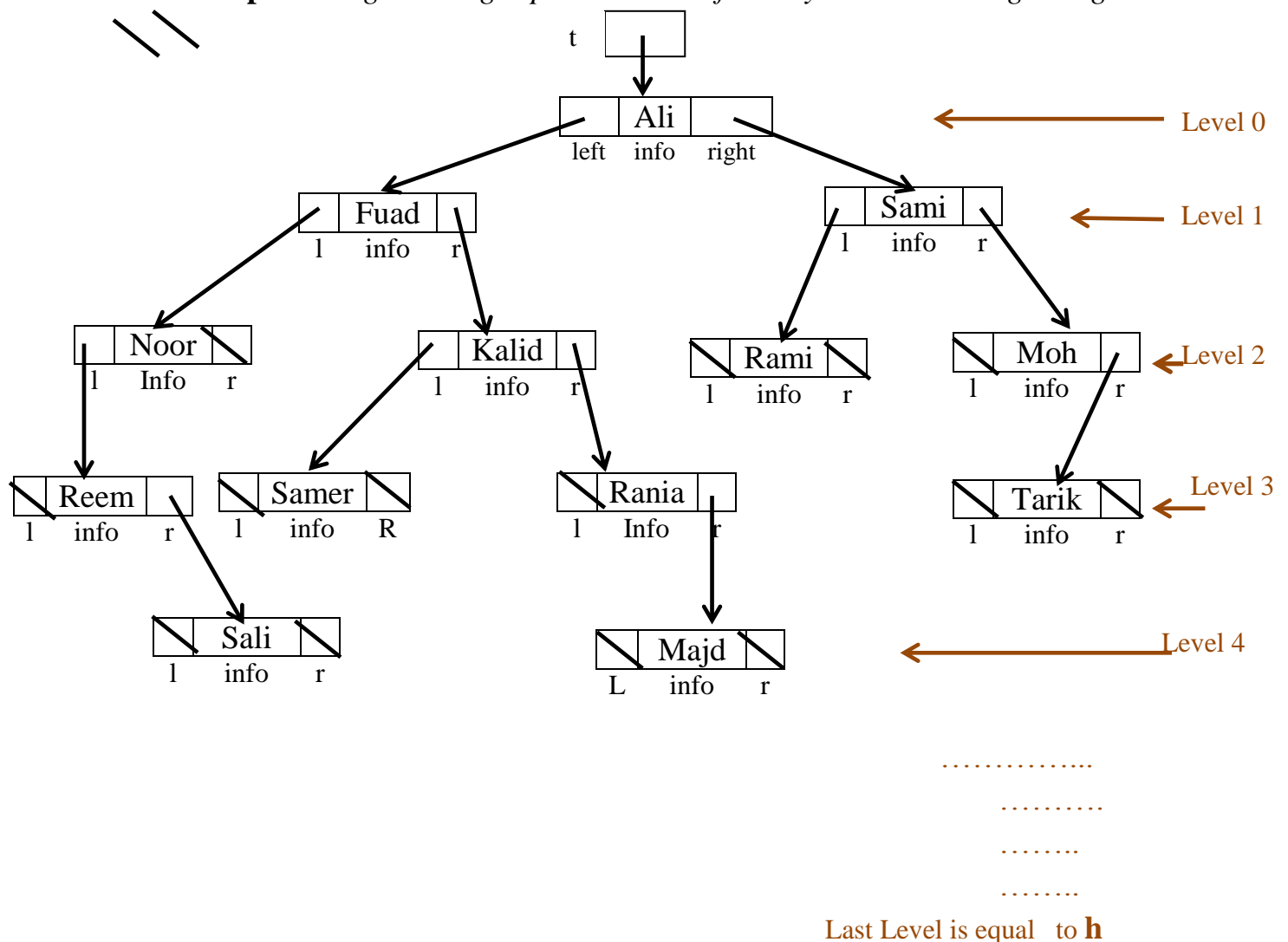


CH6• Binary Trees :

6.1. Basic concept and terminology :

A Binary Tree (B.T.) is a special kind of Linked Lists in which each Node can have at most two children (pointers) : they are distinguished as a left child and a right child. The order of the Nodes matters (we cannot just swap left and right), so it is an ordered tree. The subtree rooted at the left child of a node is called its left subtree and the subtree rooted at the right child of a node is called its right subtree.

Example : Diagramming representation of binary tree containing string



Some definitions using above figure :

Any Node above could be :

- 1 - **Root** of the B.T. : Like the Node with value *Ali* (*B.T. has only one Root*)
- 2- All other Nodes called **NON-Root**
- 3- **Leaf** (Leaves) : Like the Nodes containing (*Rami , Tarik , Majd , Sali , Samer*)
- 4- All other Nodes called **NON-Leaf**
- 5- **Parent** : Which has left or right pointer (or both) (Like Node with value *Ali , Fuad ,Sami , Noor , Kalid , Moh , reem , rania*)
- 6- **Child** (left or right) : Which has parent (except the Root)
- 7- **Ancestor** : Like Ancestors of Node with value "*Majd*" are : *Rania , Kalid , Fuad , Ali*
- 8- **Descendant** : Like all nodes after the Root are Descendants for the Root.

Other definitions :

- 1- **Level** : e.g. Root is in Level 0 ,
- 2- **Height of Binary Tree** :is equal to no of Last Level (Means equal to *h*)
- 3- **Max No of Node** : is equal to $2^{h+1} - 1$
- 4- **Max No of Leaves** : is equal to 2^h

● Some examples to print the contents of the Nodes (Naïve Approach) :

- S.O.P.(t.left.right.left.info) → "Samer"
- S.O.P.(t.left.left.left.right.info)→ "Sali"
- S.O.P.(t.info) → "Ali"

6.2. Implementing a Binary Tree (Naïve Approach) :

- Abstract Data type of B.T.

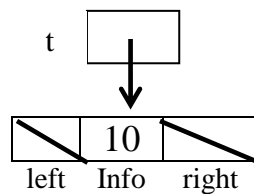
```
public class TreeNode
{
    int info;
    TreeNode left;
    TreeNode right;

    public TreeNode ()
    {
    }

    public TreeNode(int info , TreeNode left , TreeNode right)
    {
        this.info = info;
        this.left = left;
        this.right = right;
    }
}
```

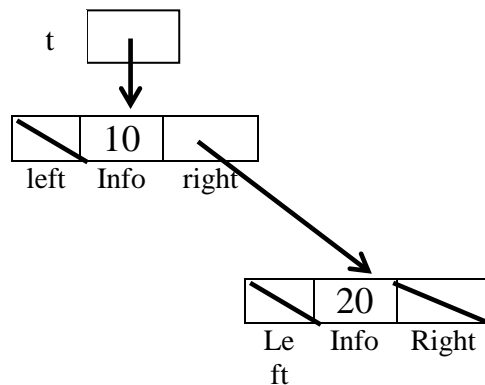
- Creating a new B.T. :

```
TreeNode t = new TreeNode (10 , null,null);
```

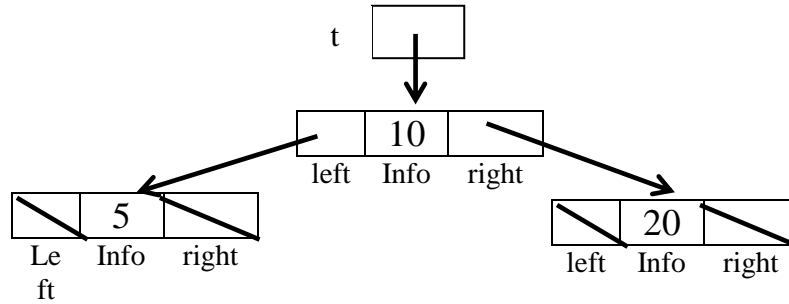


- Inserting new Objects always as leaf :

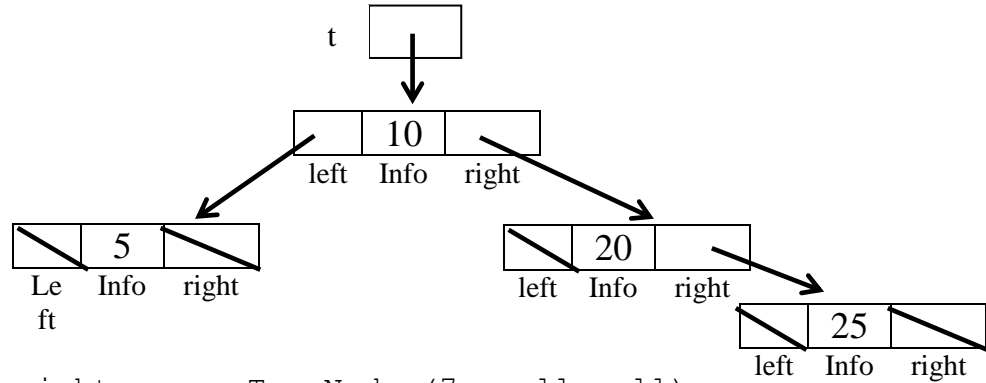
```
t.right = new TreeNode (20 , null,null);
```



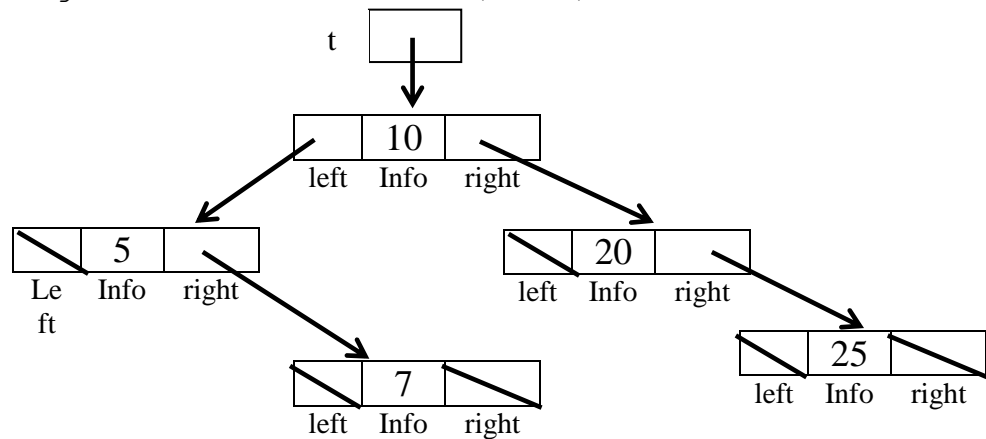
```
t.left = new TreeNode (5 , null , null);
```



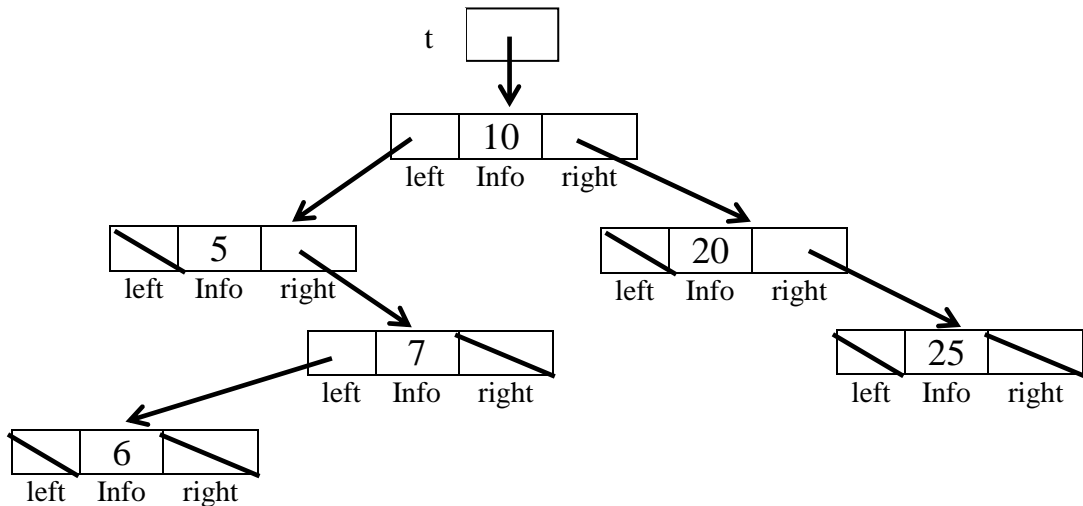
```
t.right.right = new TreeNode (25 , null , null);
```



```
t.left.right = new TreeNode (7 , null , null);
```



```
t.left.right.left = new TreeNode (6 , null , null);
```



6.3. Traversing a Binary Tree :

We often need to perform an operation on every node in a tree and sometimes the order matters. There are multiple approaches to tree traversals and we will discuss three of them:

- *inOrder Traversing*
- *preOrder Traversing*
- *postOrder Traversing*

6.3.1. InOrder Traversing : (Recursive Approach)

- process the nodes in the left subtree of a node .
- process the node itself .
- process the nodes in the right subtree of a node.

In the above, the word "***process***" is intentionally vague. The actual action depends on what exactly we need to do. If we want to print the content of the binary tree, then "***process***" means "***print***". If we need to perform some other computation based on the data stored in the nodes, then "***process***" means "***perform the computation***". We implement process as "***visit***", which prints the content of each Node :

```
void visit (Object a )
{
    S.O.P.(a);
}
```

Here is the code for a recursive *inOrder traversal* algorithm :

```
void inOrder ( TreeNode T )
{
    if ( T != null )
    { inOrder ( T.left);
      visit ( T.info);
      inOrder(T.right);
    }
}
```

6.3.2. *preOrder Traversing : (Recursive Approach)*

The *preOrder traversal* algorithm is intuitively described by the sequence of three steps mentioned already above:

- process the node itself .
- process the nodes in the left subtree of the node .
- process the nodes in the right subtree of the node.

The code for a recursive *preOrder* algorithm does not differ much :

```
void preOrder ( TreeNode  T )
{
    if ( T != null )
    {
        visit ( T.info);
        preOrder ( T.left);
        preOrder(T.right);
    }
}
```

6.3.3. *postOrder Traversing : (Recursive Approach)*

The *postOrder traversal* algorithm is intuitively described by the sequence of three steps mentioned already above:

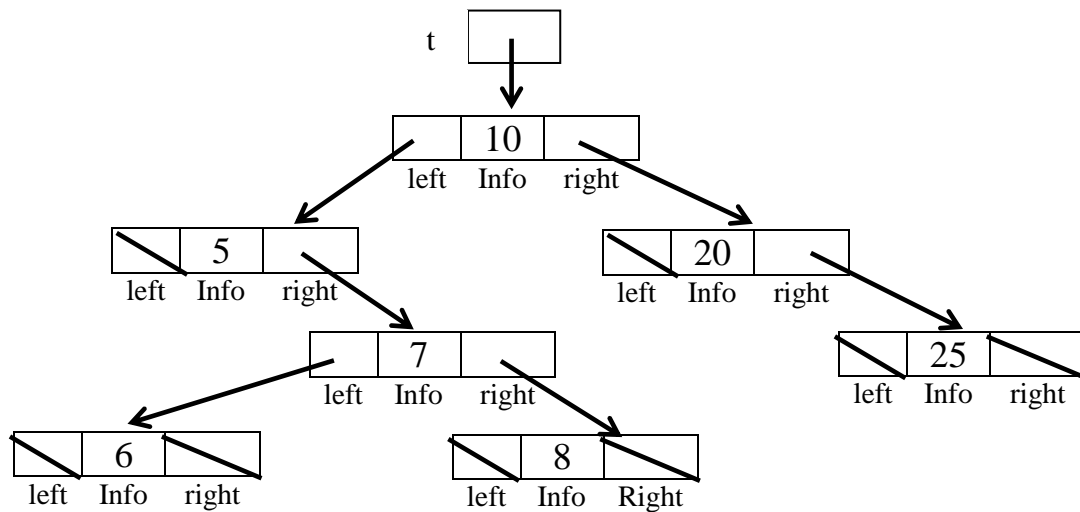
- process the nodes in the left subtree of the node .
- process the nodes in the right subtree of the node.
- process the node itself .

The code for a recursive *postOrder* algorithm does not differ much :

```
void postOrder ( TreeNode  T )
{
    if ( T != null )
    {
        postOrder ( T.left);
        postOrder(T.right);
        visit ( T.info);
    }
}
```

Example :

Traverse the following B.T. using the 3 algorithms :

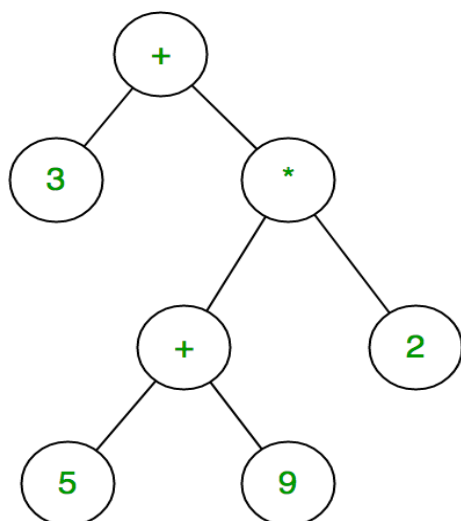


- inOrder Traversing : 5 , 6 , 7 , 8 , 10 , 20 , 25
- preOrder Traversing : 10 , 5 , 7 , 6 , 8 , 20 , 25
- postOrder Traversing : 6 , 8 , 7 , 5 , 25 , 20 , 10

6.4. Types of Binary Tree :

6.4.1. Arithmetic Expression Binary Tree :

The expression Binary Tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for $3 + ((5+9)*2)$ would be:



• How to construct an Expression Binary Tree :

Idea :

Suppose we have following Expression format :

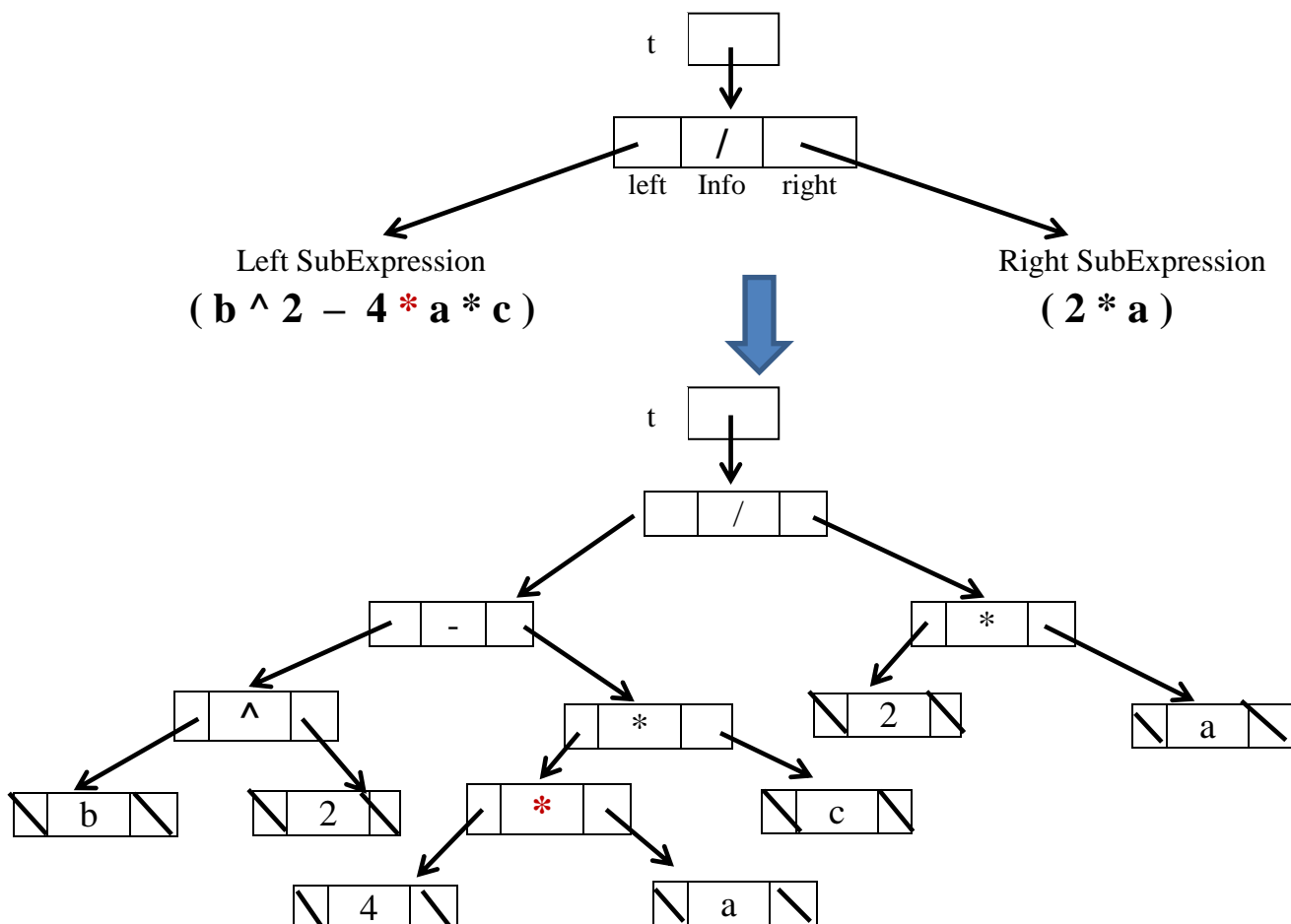
LeftSubExpression Operator RightSubExpression

- Construct root containing an operator, which will be evaluated to last (*using precedence table*).
 - Construct left subtree containing the left subExpression of the operator (*recursively*).
 - Construct right subtree containing the right subExpression of the operator (*recursively*).
- (HINT : all leaves contain operands and all NON-leaf Nodes contain operators)

Example :

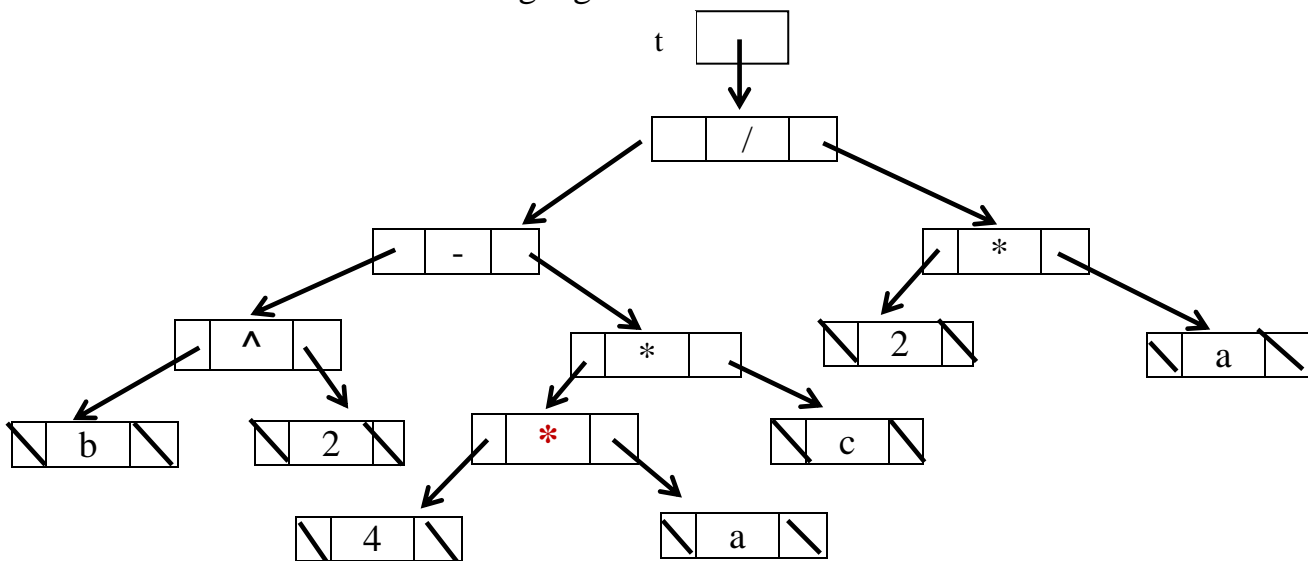
Construct an Expression Tree for the following Expression :

$$(b^2 - 4 * a * c) / (2 * a)$$



Example :

Traverse the following Expression Tree using *InOrder* , *PreOrder* and *PostOrder* Traversing algorithms :



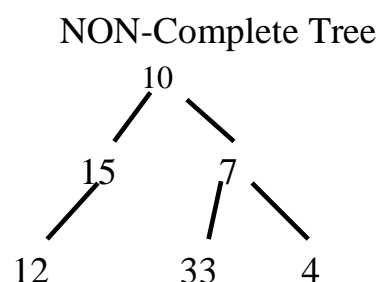
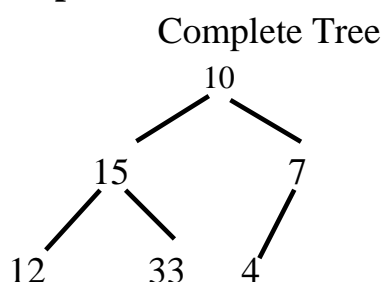
●Applying Traversing algorithms →

- InOrder Traversing : (b ^ 2 - 4 * a * c) / (2 * a)
- PreOrder Traversing : /-^b2**4ac*2a
- PostOrder Traversing : b2^4a*c*-2a*/

6.4.2. Complete Binary Tree :

A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys (objects) as left as possible. (*means : inserting a new key (object) must be levelwise left to right*)

Examples :



6.4.3. Binary Search Tree :

- **Predefinition :**

A *key* is defined as any integral value (Like int , char , Boolean , enum).

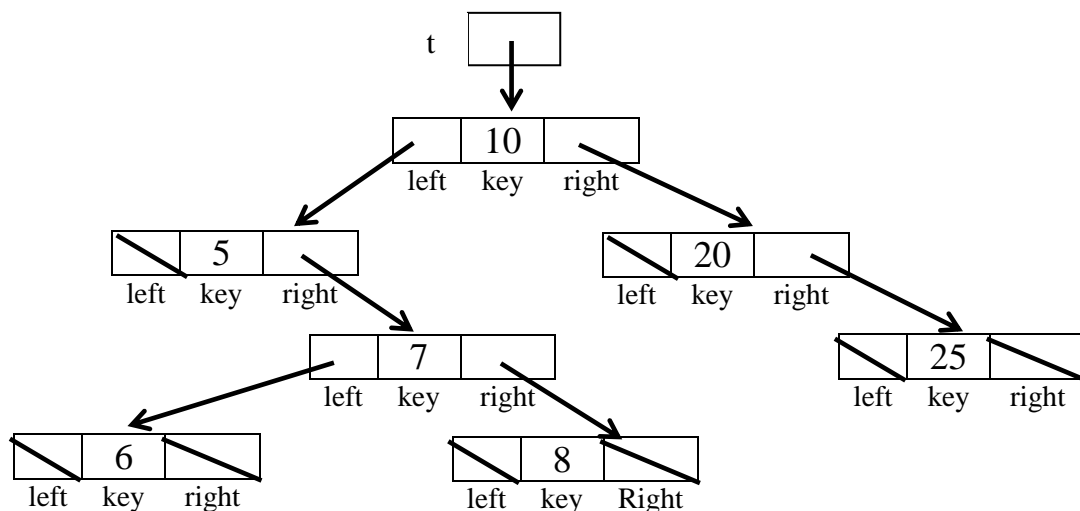
- **Definition of BST :**

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

- The value of the *key* of the left sub-tree is less than the value of its parent (root) node's *key*. (**recursively**)
- The value of the *key* of the right sub-tree is greater than the value of its parent (root) node's *key*. (**recursively**)

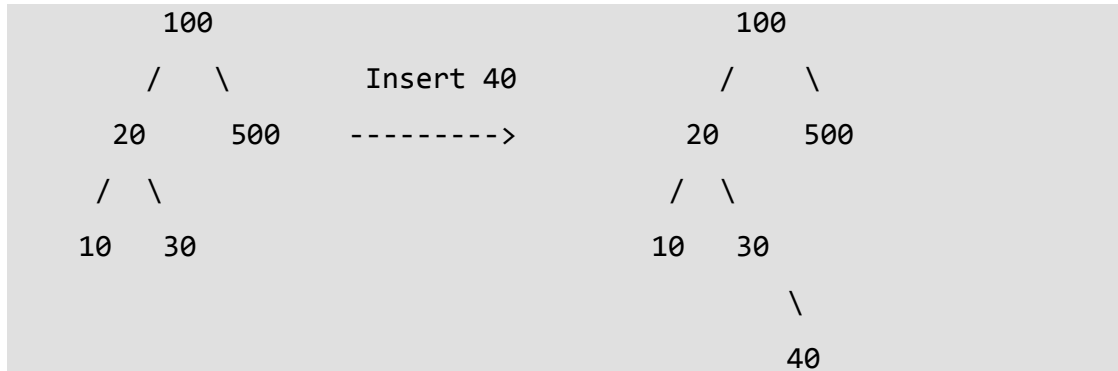
`left_subtree (keys) < node (key) < right_subtree (keys)`

Example : Binary Search Tree B.S.T.



• Insertion in BST :

A new key is always inserted at the **leaf**. We start searching a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a left or right child of the leaf node.



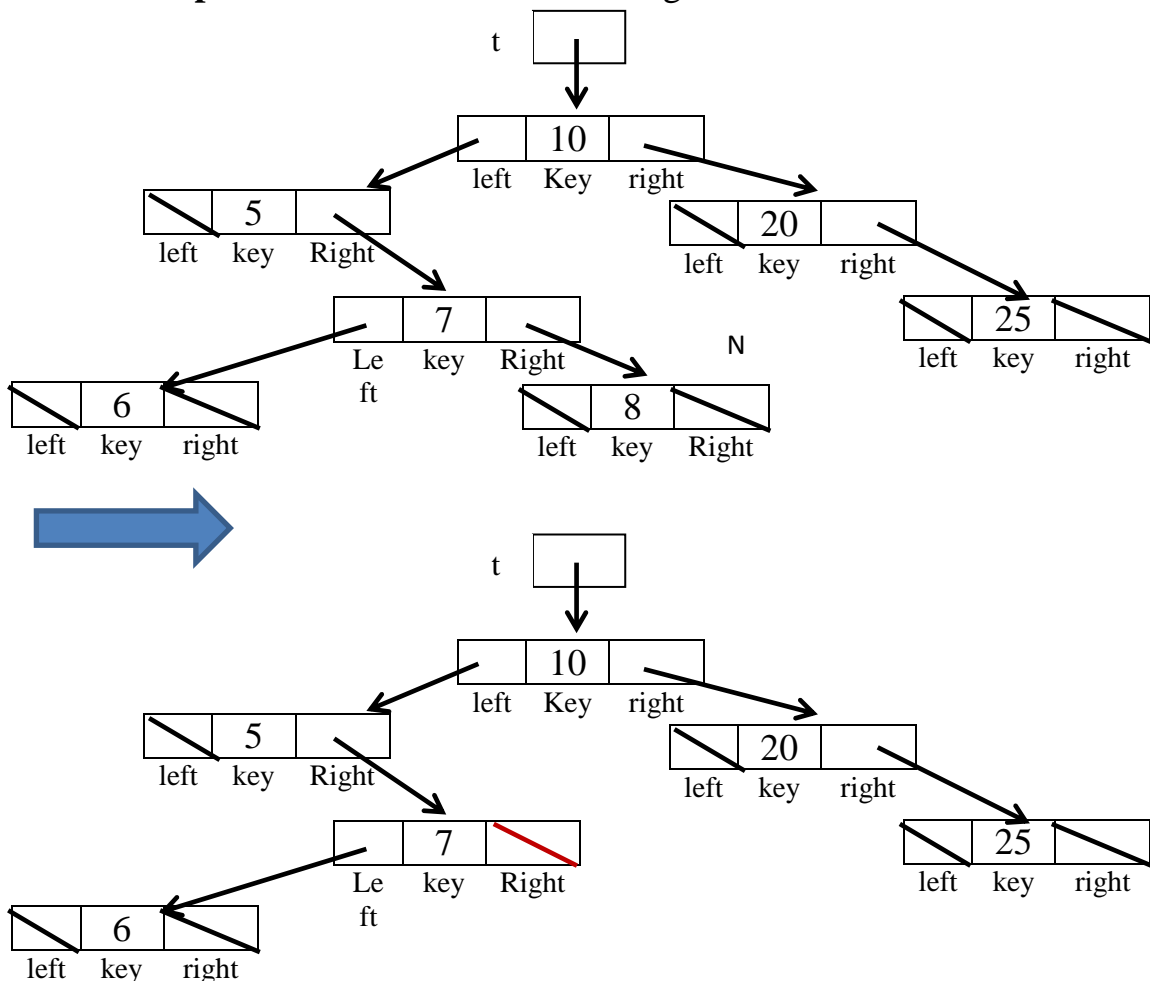
• Deletion from BST :

Suppose N is the node with the key, which will be deleted →

Then we have 3 cases to consider :

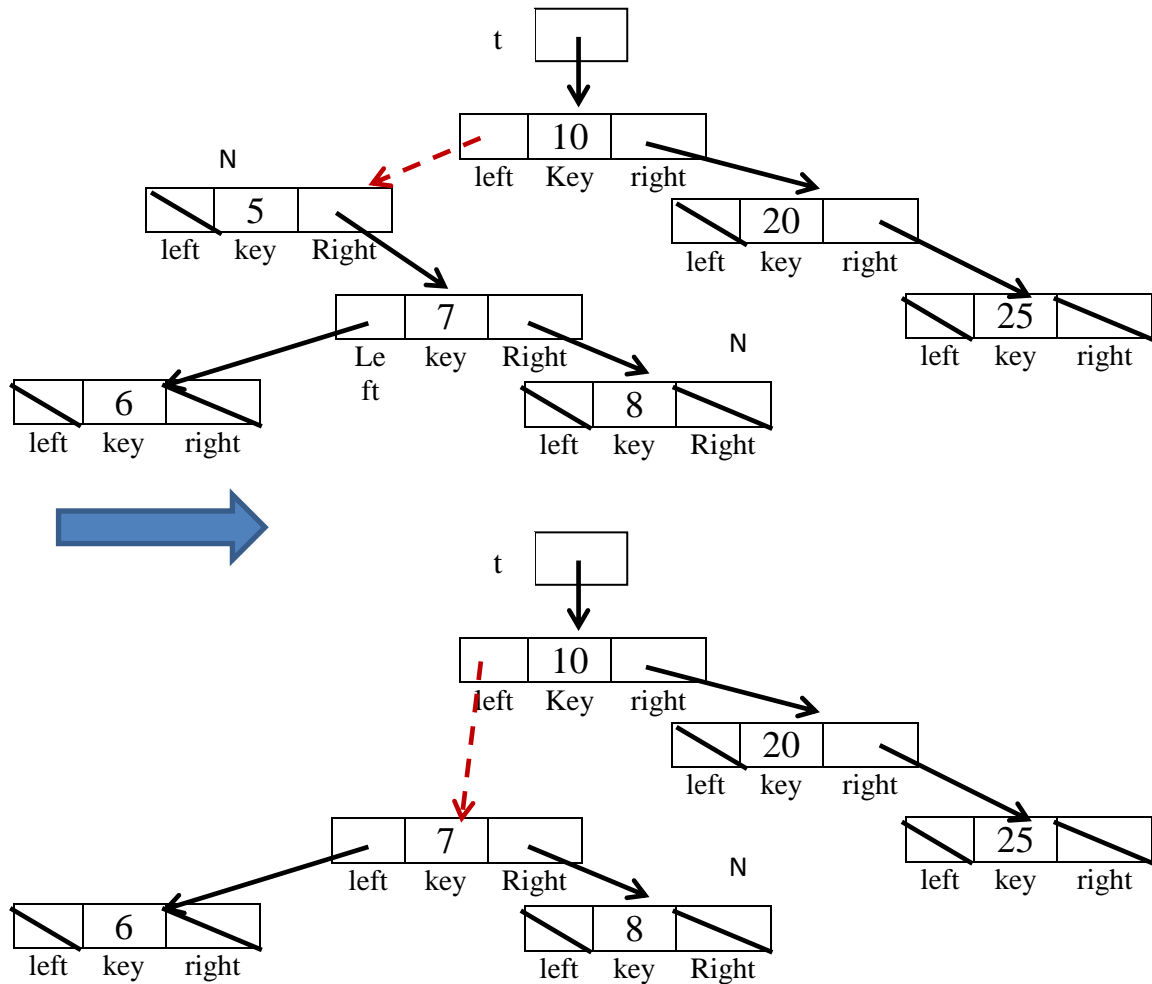
1- If N defined as leaf → pointer which points to N becomes null.

Example : delete 8 from the following BST :



2- If N NON-Leaf with one child (left or right) \rightarrow pointer which points to N , will point to the left or right child of N .

Example : Delete 5 from the following BST :



3- If N NON-Leaf with two children \rightarrow Make following steps

A- Looking for X Node as replacement for N :

(There are two approaches to find X) :

- X is the mostleft Node in the right subtree of N or
- Using inOrder Traversing to find X (X comes direct after N in the inOrder Traversing)

[Be sure : X either Leaf or NON-Leaf with right child only]

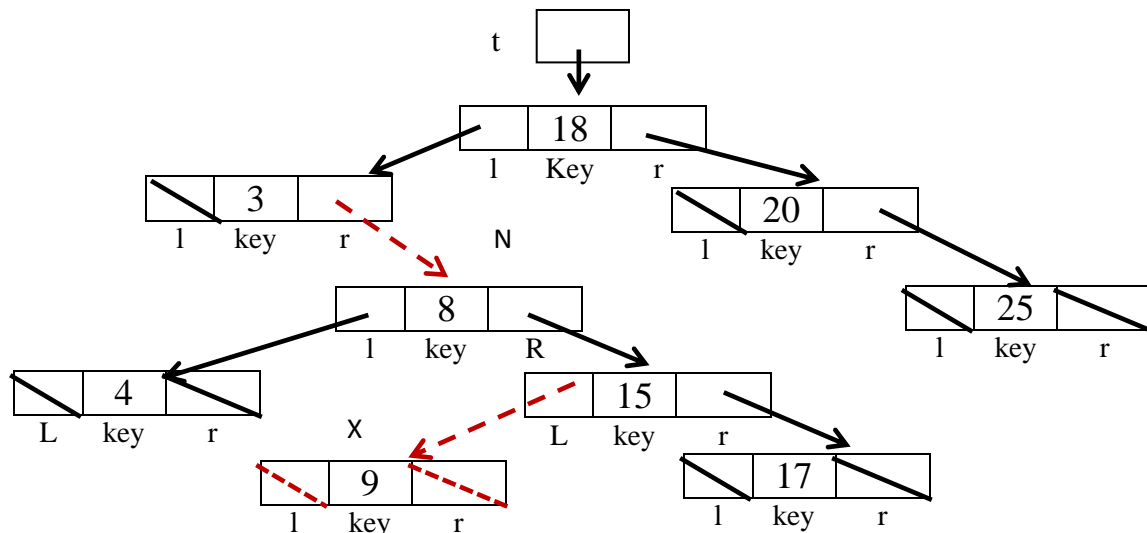
B- Pointer, which points to N , will point to X .

C- Pointer, which points to X , will point to the right child of X .(if there is any one otherwise null)

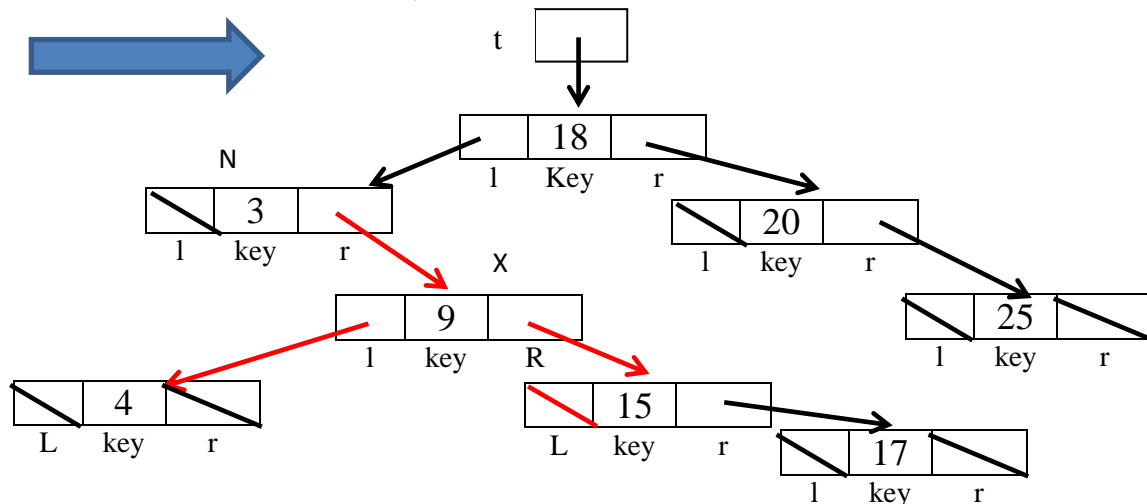
D- Left Pointer of X will point to the left child of N .

E- Right Pointer of X will point to the right child of N .

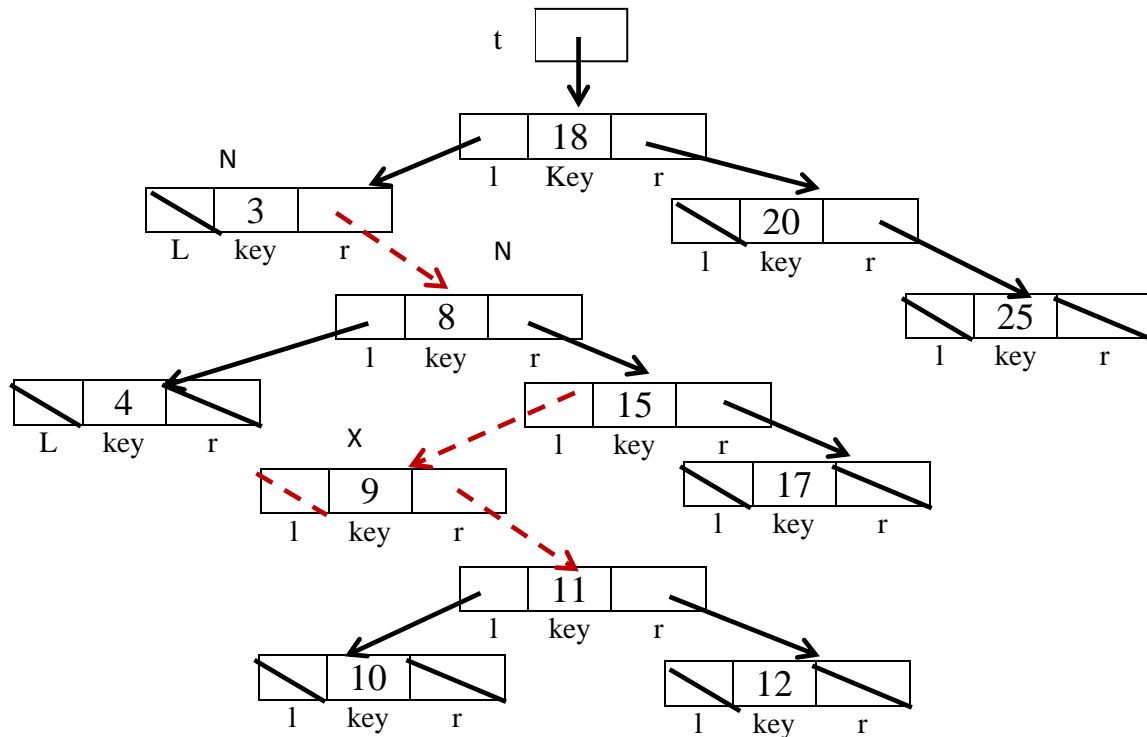
Example1 : Delete 8 from the following BST :



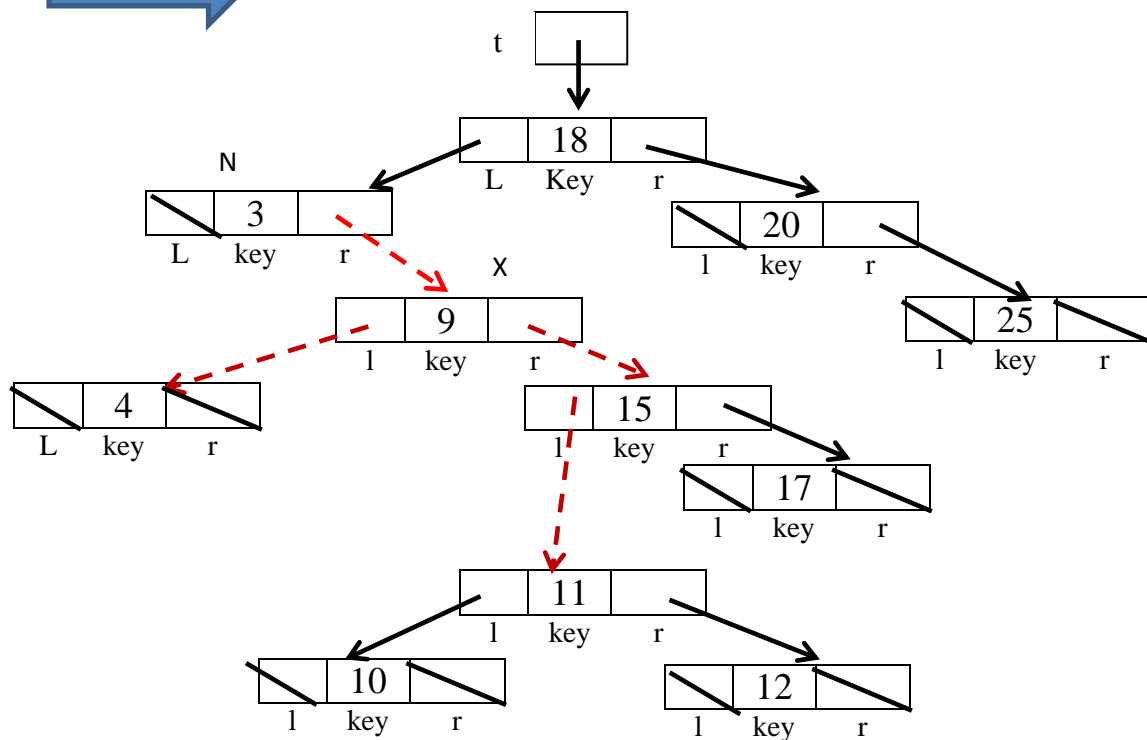
Inorder Traversing : 3 , 4 , 8 , 9 , 15 , 17 , 18 , 20 , 25
N , X



Example2 : Delete 8 from the following BST :



Inorder Traversing : 3 , 4 , 8 , 9 , 10 , 11 , 12 , 15 , 17 , 18 , 20 , 25
N , X



●BINARY SEARCH TREE (INSERT – DELETE – TRAVERSING)

// TreeNode Defining

```
public class TreeNode
{
    protected int info;    //key
    protected TreeNode left;
    protected TreeNode right;

    public TreeNode ()
    {
        }

    public TreeNode(int info,TreeNode left,TreeNode right)
    {
        this.info = info;
        this.left = left;
        this.right = right;
    }
}
```

```
public class Tree
{
    public TreeNode  root ;
}
```

//METHODS TO MANIPULATE BINARY SEARCH TREES

```
public TreeNode insertNew(int a , TreeNode T)
{
    TreeNode p;

    if (T==null)
    {
        p = new  TreeNode ( ) ;
        p.info = a ;
        p.left = null ;
        p.right = null ;
        return p ;
    }
    else if ( a < T.info)
    {
        T.left = insertNew(a,T.left);
        return T;
    }
    else if( a > T.info)
    {

```

```

        T.right = insertNew(a,T.right);
        return T;
    }
    else
    {
        System.out.println(a+" already presents ");
        return null;
    }
}

public void insert ( int  a )
{
    root = insertNew ( a , root ) ;
}

public TreeNode deleteElement(int a,TreeNode T)
{
    TreeNode p , q , s;

    if (T == null)
    { System.out.println(a + " Not found") ;
      return T ; }
    else if ( a  <  T.info )
    {
        T.left  =  deleteElement ( a , T.left ) ;
        return T ; }
    else if ( a  >  T.info )
    {
        T.right  =  deleteElement ( a , T.right ) ;
        return T ;
    }
    else    // ELEMENT FOUND!!!!
    { q=T;
      s=T;

      if (q.right == null)
          T=q.left; // T WITH ONE LEFT CHILD OR T IS NULL
          // T WITH ONE RIGHT CHILD OR T IS NULL
      else if (q.left==null)
          T=q.right; // T WITH ONE RIGHT CHILD OR T IS NULL
      else    // T HAS TWO CHILDREN
          { p  =  q.right ;
            // P POINTS TO THE NODE N' AND S.LEFT WILLPOINTS TO THE RIGHT
            // CHILD OF N' IF P HAS RIGHT CHILD, NULL OTHERWISE....
            while ( p.left  !=  null )
            { s = p ;
              p = p.left ;  }
            // REPLACE N WITH N'
            T=p;

```

```

        T.left = q.left ;
        s.left = p.right ;
        if ( q.right == p )
            T.right = p.right ;
        else
            T.right = q.right ;
    }
    return T;
} }

public void delete ( int a )
{ root = deleteElement ( a , root ) ;}

public void visit ( int a )
{ System.out.println ( a ) ;}

public void preOrder ( TreeNode T )
{
    if ( T != null )
    {
        visit ( T.info ) ;
        preOrder ( T.left ) ;
        preOrder ( T.right ) ;
    }
}

public void inOrder ( TreeNode T )
{
    if ( T != null )
    {
        inOrder ( T.left ) ;
        visit(T.info) ;
        inOrder ( T.right ) ;
    }
}

public void postOrder ( TreeNode T )
{
    if ( T != null )
    {
        postOrder ( T.left ) ;
        postOrder ( T.right ) ;
        visit ( T.info ) ;
    }
}

}

```

```

import javax.swing.*;
class TreeTest
{
    public static void main ( String args[] )
    {
        TreeNode T1 = new TreeNode ( 50 , null , null ) ;
        Tree Z = new Tree () ;
        Z.root = T1;
        String inserM = JOptionPane.showInputDialog( "-1 to
exit\n"+"1 to insert\n"+"2 to delete\n"+"3 to traverse" ) ;
        int n ;
        String c ;
        int m ;
        String insertNewElement , deleteElement ;
        int in ;

        n = Integer.parseInt ( inserM ) ;
        while ( n != -1)
        {
            if ( n == 1)
            {
                insertNewElement =
JOptionPane.showInputDialog("Enter new element to insert");
                in = Integer.parseInt ( insertNewElement ) ;
                Z.insert ( in ) ;
            }
            else if ( n == 2 )
            {
                deleteElement =
JOptionPane.showInputDialog("Enter element to delete");
                in = Integer.parseInt ( deleteElement ) ;
                Z.delete ( in ) ;
            }
            else if ( n == 3 )
            {
                c = JOptionPane.showInputDialog ( "1 for
PREORDER\n"+"2 for INORDER \n"+"3 for POSTORDER\n" ) ;
                m = Integer.parseInt ( c ) ;
                if ( m == 1 )
                    Z.preOrder ( Z.root ) ;
                else if ( m == 2 )
                    Z.inOrder ( Z.root ) ;
                else
                    Z.postOrder ( Z.root ) ;
            }
            inserM = JOptionPane.showInputDialog ( "-1 to
exit\n"+"1 to insert\n"+"2 to delete\n"+"3 to traverse" ) ;
            n = Integer.parseInt ( inserM ) ;
        }
    }
}

```

CH7● Heap Array , Heap Tree , Priority Queue

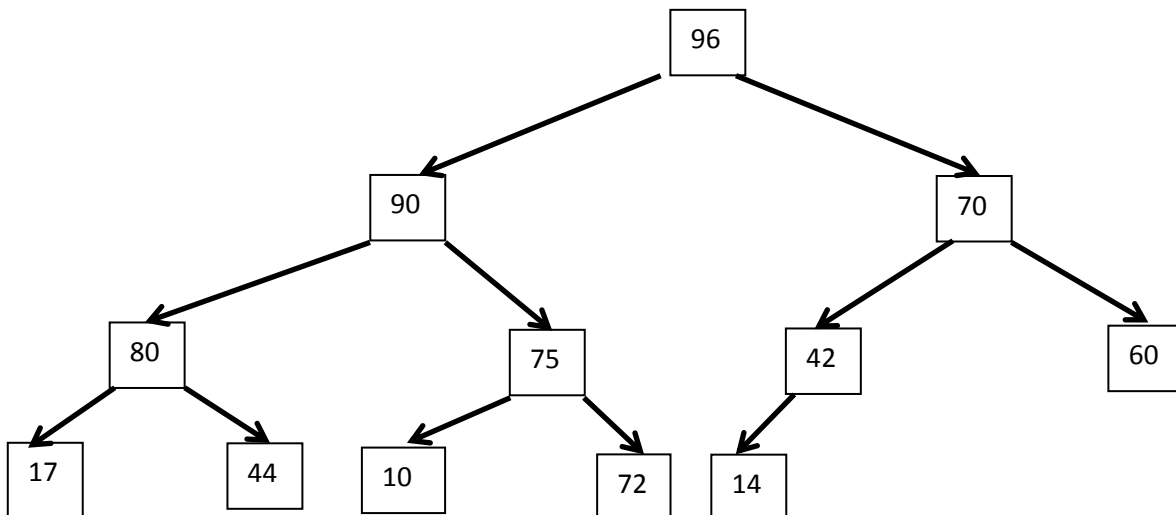
7.1. Heap Tree :

● Definition :

A Heap Tree is complete binary tree with following additionally characteristics :

- Root Node contains the greatest key.
- Each Node key is greater than the keys of its children (if there is any one).
- All leaves are in the last two adjacent levels.

Example :

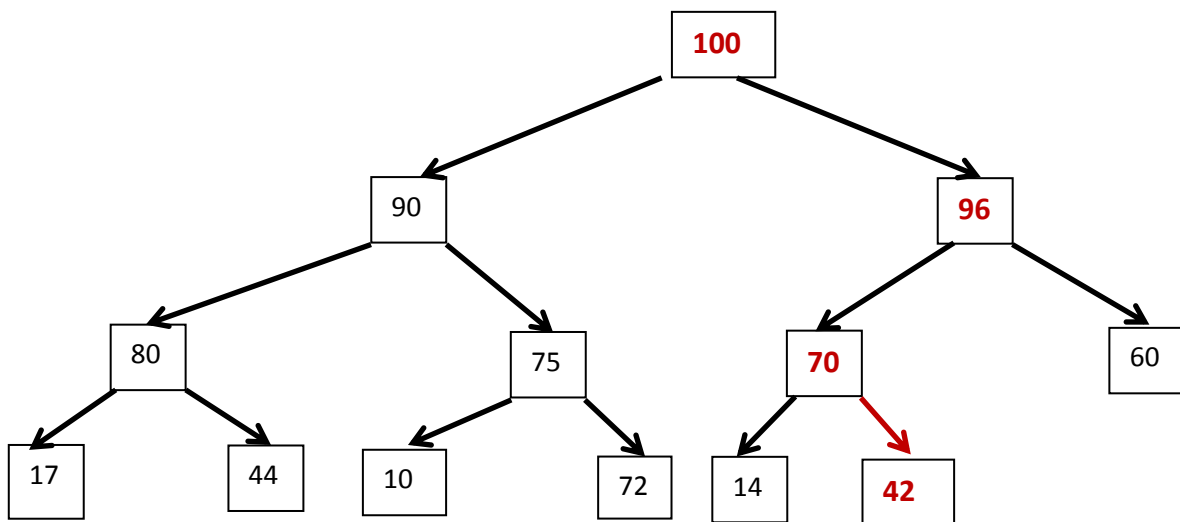
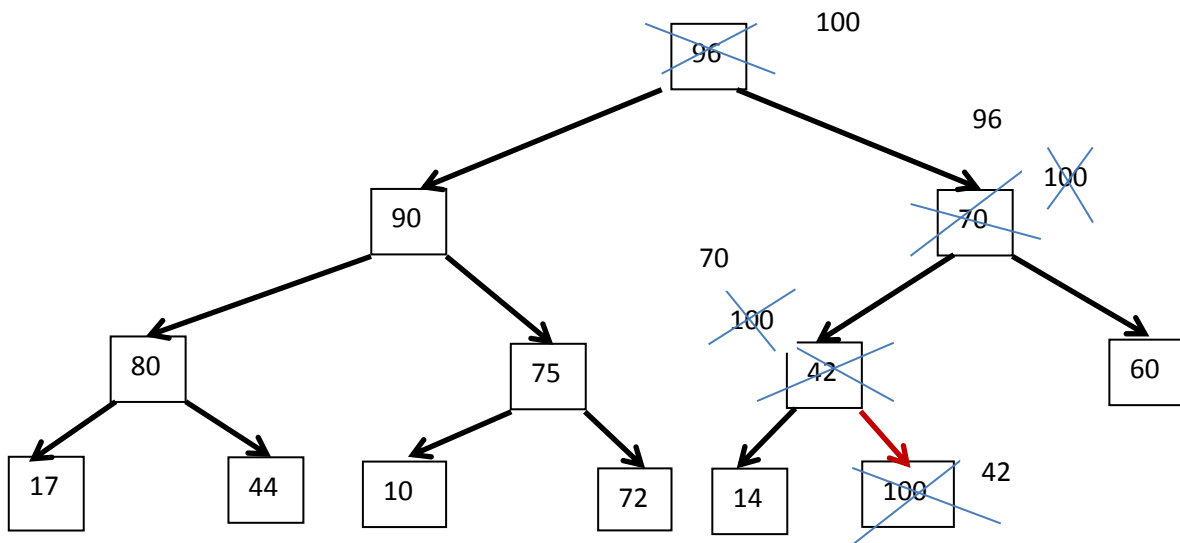


●Insert a new Node in Heap Tree :

Insert new Node levelwise left to right (Like complete Binary Tree), **additionally** with swap (new node with its parent) , if new node's key greater than the key of parent. (*recursively*)

Example :

Insert 100 into the following Heap Tree :



●Delete a node from Heap Tree :

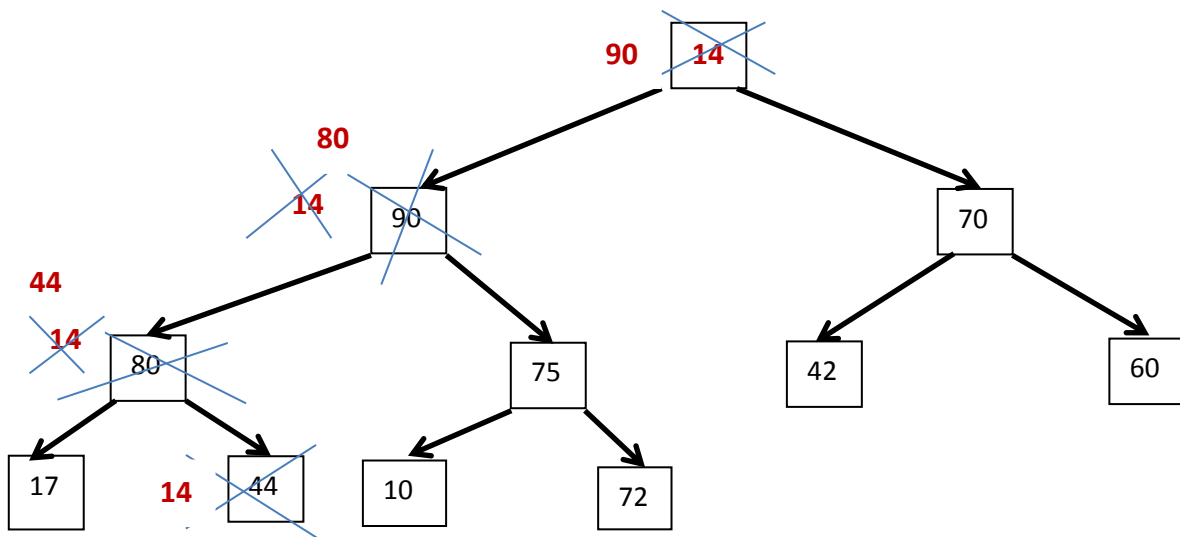
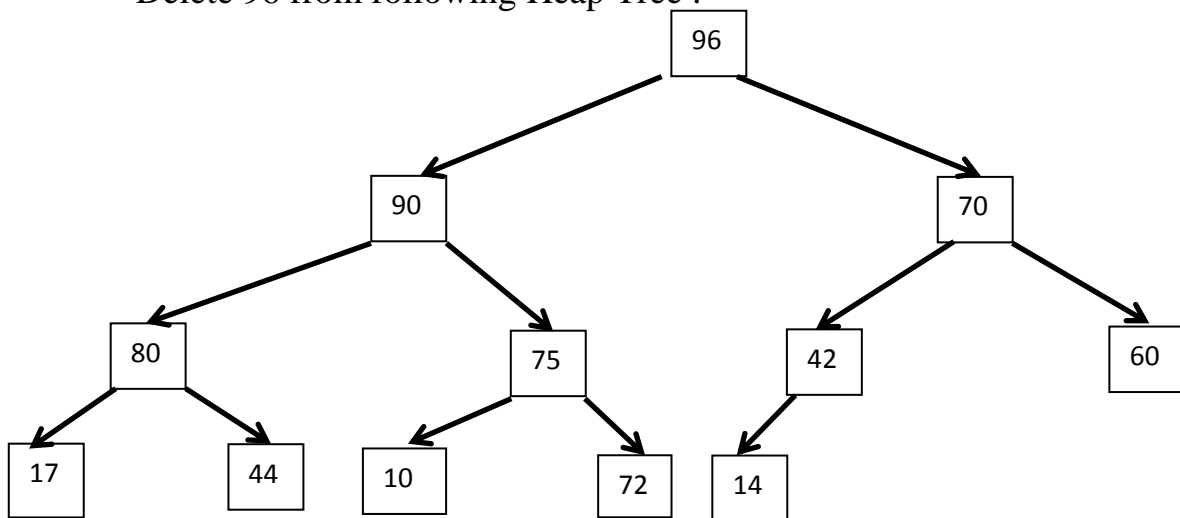
Deletion is always from Root and then restructuring the tree as follows :

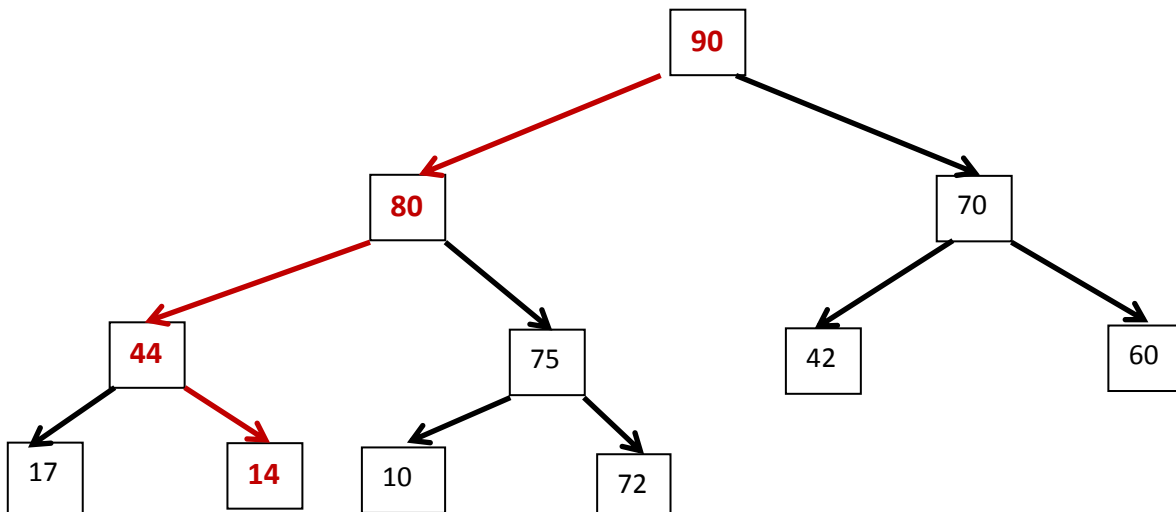
- Replace Root key with the key from mostRight node in the last level
- If the new Root key is less than the max of its children →

Swap (Root's key with max (left child , right child)) *[recursively]*

Example :

Delete 96 from following Heap Tree :





●Heap Array :

Definition :

A Heap Array H is a one dimensional array with length $2^{h+1} - 1$, where h the height of a Heap Tree. And for any index i:

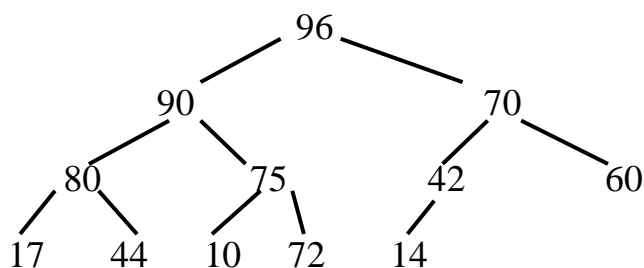
$$H[i/2] > H[i] > \max (H[2*i] , H[2*i+1])$$

OR

parent > H[i] > max (left child , right child) .. comparing with Heap Tree

➔ The key in H[1] is the greatest key

Example : Given Heap Tree with h = 3 ➔ Max No of Nodes = $2^{h+1} - 1 = 15$



➔ Heap Array indexed by 1..15

H	96	90	70	80	75	42	60	17	44	10	72	14			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

We will discuss three concepts of Heap Array :

- 1- Insertion new object in Heap Array**
- 2- Deletion object from Heap Array**
- 3- Make Heap Array from any given array**

HINT : Insertion and Deletion from Heap Array refer to the FIFO with priority.

● **Insertion in a Heap Array :**

Idea :

- Insert new key last in index i (**FIFO with priority**) and then
- **Bubbling up** the new key forward (if possible) :

While (true) // key is not in the correct position

```
{ if ( H[i] > H[i/2] )  
    ➔ swap ( H[i] , H[i/2])  
}
```

Example1 :

Insert 35 into the following heap array :

H	96	90	70	80	75	42	60	17	44	10	72	14			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

➔

H	96	90	70	80	75	42	60	17	44	10	72	14	35		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

STOP!!!!

Example2 :

Insert 100 into the following heap array :

H	96	90	70	80	75	42	60	17	44	10	72	14			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

➔

H	96	90	70	80	75	42	60	17	44	10	72	14	100		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

➔ $i = 13 \rightarrow H[13] = 100 > H[i/2] = H[6] = 42 \rightarrow \text{swap}(100, 42)$

H	96	90	70	80	75	100	60	17	44	10	72	14	42		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

➔ $i = 6 \rightarrow H[6] = 100 > H[i/2] = H[3] = 70 \rightarrow \text{swap}(100, 70)$

H	96	90	100	80	75	70	60	17	44	10	72	14	42		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

➔ $i = 3 \rightarrow H[3] = 100 > H[i/2] = H[1] = 96 \rightarrow \text{swap}(100, 96)$

H	100	90	96	80	75	70	60	17	44	10	72	14	42		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

STOP!!!!

●Deletion from Heap Array :

Idea :

- Delete always from $H[1]$ (**FIFO with priority**) through replacing $H[i]$ with the last key : $\text{REPLACE}(H[1], \text{Last})$ and then

- **Trickling Down** :

$i = 1$

While (true) // key is not in the correct position

{ if ($H[i] < \max (H[2*i] , H[2*i+1])$)

$\rightarrow \text{swap} (H[i] , \max (H[2*i] , H[2*i+1]))$

$i = 2*i , i = 2*i + 1$ }

Example :

Delete 96 from following heap array :

H	96	90	70	80	75	42	60	17	44	10	72	14			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Replace (96 , 14) \rightarrow

H	14	90	70	80	75	42	60	17	44	10	72				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$i = 1 \rightarrow H[1] = 14 < \max (H[2] , H[3]) = \max (90,70) = 90 \rightarrow \text{YES} \rightarrow \text{swap}$
 $(H[1],H[2]) \rightarrow \text{swap}(14,90) \rightarrow$

H	90	14	70	80	75	42	60	17	44	10	72				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$i = 2 \rightarrow H[2] = 14 < \max (H[4] , H[5]) = \max(80,75) = 80 \rightarrow \text{YES} \rightarrow \text{swap}$
 $(H[2],H[4]) \rightarrow \text{swap}(14,80) \rightarrow$

H	90	80	70	14	75	42	60	17	44	10	72				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$i = 4 \rightarrow H[4] = 14 < \max (H[8] , H[9]) = \max(17,44) = 44 \rightarrow \text{YES} \rightarrow \text{swap}$
 $(H[4],H[9]) \rightarrow \text{swap}(14,44) \rightarrow$

H	90	80	70	44	75	42	60	17	14	10	72				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

STOP!!!!

●Make Heap (Heapifying)

Idea :

1- Calculate N : No of keys in the array.

2- Starting at index $i = N/2$: (**Outer loop**)

A- if ($H[i] < \max (H[2*i] , H[2*i+1])$)

→ Trickling down : swap ($H[i]$, $\max (H[2*i] , H[2*i+1])$)

B- Calculate new index j of $H[i]$: (if $H[j]$ has any children) [inner loop]

 if ($H[j] < \max (H[2*j] , H[2*j+1])$)

→ Trickling down : swap ($H[j]$, $\max (H[2*j] , H[2*j+1])$)

C- Calculate $i = i - 1$

3- while (true $\backslash\backslash$ $i >= 1$) goto step 2 else

4- Stop

Example :

Make Heap Array from the following given array :

H	5	10	27	60	59	62	14	73
	1	2	3	4	5	6	7	8

1- Calculate $i = N/2 = 8/2 = 4$

$H[4] = 60 < \max(H[8], H[9]) = \max(H[8]) = 73 \rightarrow \text{YES} \rightarrow \text{swap}(60, 73)$

→

H	5	10	27	73	59	62	14	60
	1	2	3	4	5	6	7	8

→ New position for 60 : $j = 8 \rightarrow$ Nothing to do

2- Calculate $i = i - 1 = 3$:

$H[3] = 27 < \max(H[6], H[7]) = \max(62, 14) = 62 \rightarrow \text{YES} \rightarrow \text{swap}(27, 62)$

→

H	5	10	63	73	59	27	14	60
	1	2	3	4	5	6	7	8

→ New position for 27 : $j = 6 \rightarrow \text{Nothing to do}$

3- Calculate $i = i - 1 = 2$:

$H[2] = 10 < \max(H[4], H[5]) = \max(73, 59) = 73 \rightarrow \text{YES} \rightarrow \text{swap}(10, 73)$

→

H	5	73	63	10	59	27	14	60
	1	2	3	4	5	6	7	8

→ New position for 10 : $j = 4 \rightarrow \text{[inner loop]}$

$j = 4$:

$H[4] = 10 < \max(H[8], H[9]) = \max(H[8]) = 60 \rightarrow \text{YES} \rightarrow \text{swap}(10, 60)$

→

H	5	73	63	60	59	27	14	10
	1	2	3	4	5	6	7	8

→ New position for 10 : $j = 8 \rightarrow \text{[Nothing to do]}$

4- Calculate $i = i - 1 = 1$:

$H[1] = 5 < \max(H[2], H[3]) = \max(73, 63) = 73 \rightarrow \text{YES} \rightarrow \text{swap}(5, 73)$

→

H	73	5	63	60	59	27	14	10
	1	2	3	4	5	6	7	8

→ New position for 5 : $j = 2 \rightarrow \text{[inner loop]}$

$j = 2$:

$H[2] = 5 < \max(H[4], H[5]) = \max(H[4], H[5]) = 59 \rightarrow \text{YES} \rightarrow \text{swap}(5, 59)$

→

H	73	60	63	5	59	27	14	10
	1	2	3	4	5	6	7	8

→ New position for 5 : $j = 4 \rightarrow$

$H[4] = 5 < \max(H[8], H[9]) = \max(H[8]) = 10 \rightarrow \text{YES} \rightarrow \text{swap}(5, 10)$

→

H	73	60	63	10	59	27	14	5
	1	2	3	4	5	6	7	8

→ New position for 5 : $j = 8 \rightarrow [\text{Nothing to do}]$

STOP!!!

CH8● Graph

8.1 Basic concepts and Terminology :

● Definition of Graph :

▪ A Graph is a non-linear data structure consisting of **nodes** and **edges**. The nodes are sometimes also referred to as **vertices** and the edges are lines that connect any two nodes in the graph.

▪ **Formally(mathematical)** : a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices :

$G = (V, E)$ an ordered pair of two sets :

$V = \{ v_0, v_1, v_2, \dots, v_n \}$ set of all nodes (vertices) in G

$E = \{ e_1, e_2, e_3, \dots \}$ set of all edges in G

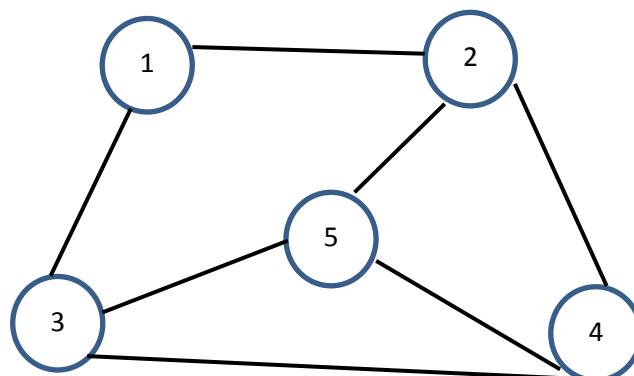
For any edge $e \in E : e = \{ v_i, v_j \}$, where v_i is the **Source Vertex** and v_j is the **Sink Vertex**.

Example :

$G = (V, E)$, where

$V = \{ 1, 2, 3, 4, 5 \}$

$E = \{ \{1,2\}, \{1,3\}, \{2,4\}, \{2,5\}, \{3,5\}, \{3,4\}, \{4,5\} \}$

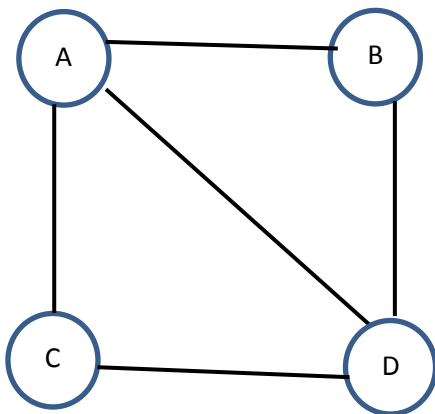


8.2 Types of Graph :

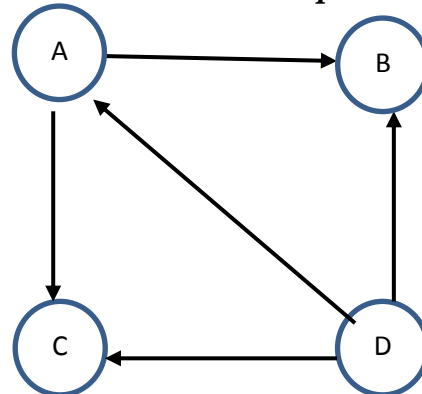
- 1- Undirected Graph
- 2- Directed Graph
- 3- Undirected Weighted Graph
- 4- Directed Weighted Graph

A– Diagramming Representation :

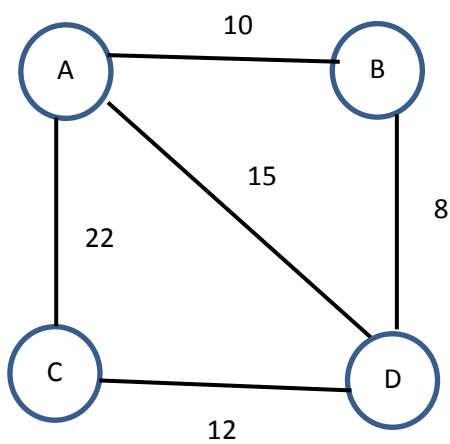
1- Undirected Graph



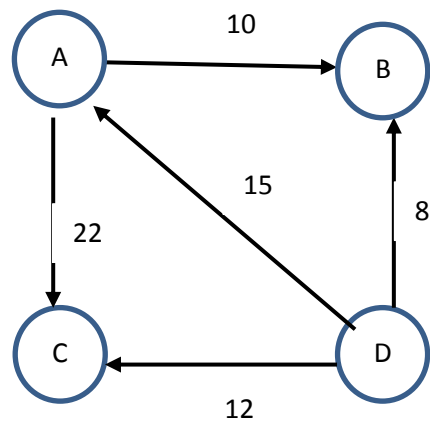
2- Directed Graph



3- Undirected Weighted Graph



4- Directed Weighted Graph



B- Formal (mathematical) Representation :

1- Undirected Graph :

$G = (V, E)$, where

$V = \{ A , B , C , D \}$

$E = \{ \{A,B\} , \{A,C\} , \{A,D\} , \{B,D\} , \{C,D\} \}$ and

For any edge $e \in E : e = \{ v_i , v_j \} = \{ v_j , v_i \} \rightarrow$

The graph is "*symmetric*"

By example : $\{A,B\} = \{B,A\}$

2- Directed Graph :

$G = (V, E)$, where

$V = \{ A , B , C , D \}$

$E = \{ (A,B) , (A,C) , (D,A) , (D,B) , (D,C) \}$ and

For any edge $e \in E : e = (v_i , v_j) \neq (v_j , v_i) \rightarrow$

The graph is "*NON-symmetric*"

By example : $(A,B) \neq (B,A)$

- To define weighted Graphs, we need to define weight function :

For any graph (directed or undirected), $G = (V, E)$

$w : E \longrightarrow \text{any Value (integer , float ,)}$

For any edge $e \in E : e = (v_i , v_j)$ OR $e = \{ v_i , v_j \}$

$w(e) = \text{value} , \quad \text{if there is an edge from } v_i \text{ to } v_j$

OR

$w(e) = \infty \quad \text{otherwise}$

3- Undirected Weighted Graph :

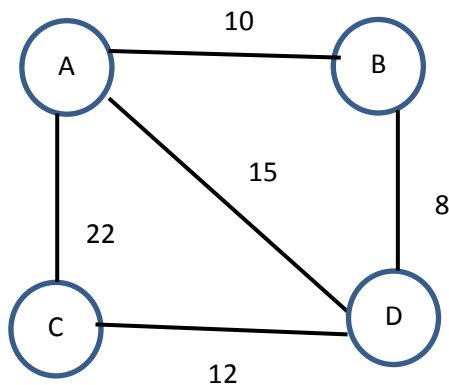
$G = (V, E)$, where

$V = \{ A , B , C , D \}$

$E = \{ \{A,B\} , \{A,C\} , \{A,D\} , \{B,D\} , \{C,D\} \}$ and

For any edge $e \in E : e = \{ v_i , v_j \} = \{ v_j , v_i \} \rightarrow$

The graph is "*symmetric*"



By example : $\{A,B\} = \{B,A\}$

$w(\{A,B\}) = 10 = w(\{B,A\})$

$w(\{B,C\}) = \infty$

4- Directed Weighted Graph :

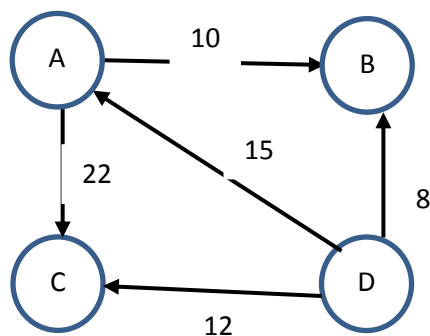
$G = (V, E)$, where

$V = \{ A , B , C , D \}$

$E = \{ (A,B) , (A,C) , (D,A) , (D,B) , (D,C) \}$ and

For any edge $e \in E : e = (v_i , v_j) \neq (v_j , v_i) \rightarrow$

The graph is "*NON-symmetric*"



By example : $(A,B) \neq (B,A)$

$w((A,B)) = 10$, But $w((B,A)) = \infty$

$w((B,C)) = \infty$

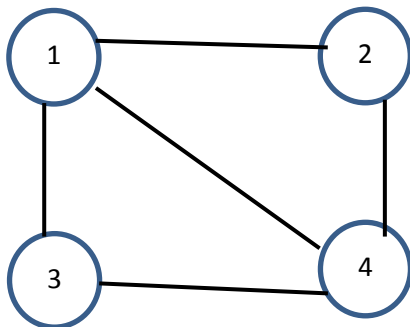
C- Implementing of Graph :

There are two approaches to implement (program) a Graph :

1- Fixed memory allocation using "*Adjacent Matrix*":

Exampe :

1- Undirected Graph



int adj [4][4] ;

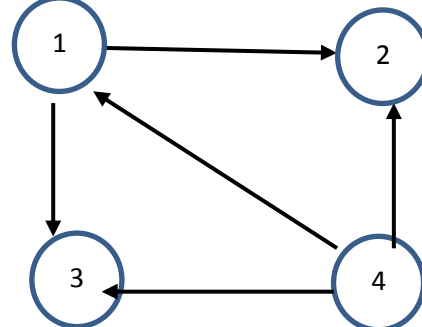
adj :

	1	2	3	4
1	0	1	1	1
2	1	0	∞	1
3	1	∞	0	1
4	1	1	1	0

Symmetric Graph ,

if for all i , j : $\text{adj}[i][j] = \text{adj}[j][i]$;

2- Directed Graph



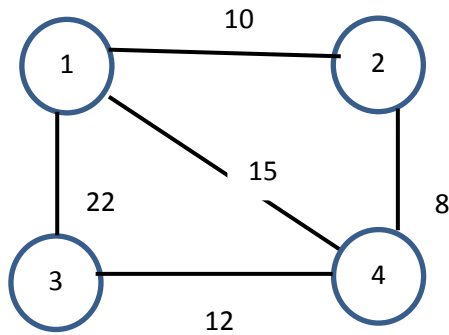
adj :

	1	2	3	4
1	0	1	1	∞
2	∞	0	∞	∞
3	∞	∞	0	∞
4	1	1	1	0

NON-Symmetric Graph

if for some i , j : $\text{adj}[i][j] \neq \text{adj}[j][i]$;

3- Undirected Weighted Graph



int adj [4][4] ;

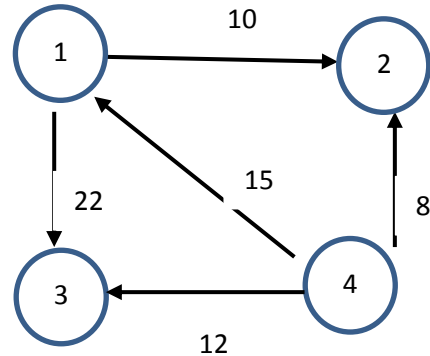
adj :

	1	2	3	4
1	0	10	22	15
2	10	0	∞	8
3	22	∞	0	12
4	15	8	12	0

Symmetric Graph ,

if for all i, j : $\text{adj}[i][j] = \text{adj}[j][i]$;

4- Directed Weighted Graph



adj :

	1	2	3	4
1	0	10	22	∞
2	∞	0	∞	∞
3	∞	∞	0	∞
4	15	8	12	0

NON-Symmetric Graph

if for some i, j : $\text{adj}[i][j] \neq \text{adj}[j][i]$;

2- Dynamic memory allocation using "Adjacency Lists" :

First, we will discuss some definitions for given Graph $G = (V, E)$:

• **Adjacency sets** and **Degrees** of undirected and directed Graphs :

- V_x set of all vertices adjacent to vertex x :

$V_x = \{ y \mid \{x,y\} \in E \}$ // Undirected Graph

$V_x = \{ y \mid (x,y) \in E \}$ // Directed Graph

- Degree of vertex x : number of edges e in which x is one of the endpoints of edge e .

- Predecessors of vertex x : // **For only Directed Graph**

$\text{PRED}(x) = \text{set of all vertices } y \in V, \text{ such that } (y,x) \in E$

- Successors of vertex x : // **For only Directed Graph**

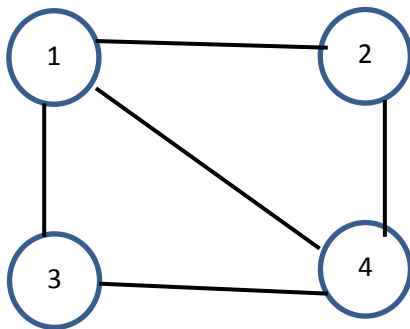
$\text{SUCC}(x) = \text{set of all vertices } y \in V, \text{ such that } (x,y) \in E$

- Indegree of vertex x : the number of $\text{PRED}(x)$

- Outdegree of vertex x : the number of $\text{SUCC}(x)$

Exampe :

1- Undirected Graph

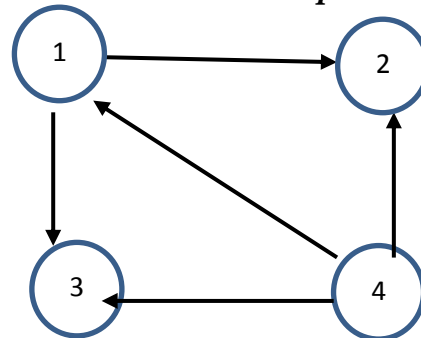


$V = \{ 1, 2, 3, 4 \}$

$x = 1 \rightarrow V_1 = \{ 2, 3, 4 \}$

$\text{Deg}(x) = \text{Deg}(1) = 3$

2- Directed Graph



$V = \{ 1, 2, 3, 4 \}$

$x = 1 \rightarrow V_1 = \{ 2, 3 \}$

$\text{Deg}(x) = \text{Deg}(1) = 3$

$\text{PRED}(x) = \text{PRED}(1) = \{ 4 \}$

$\text{SUCC}(x) = \text{SUCC}(1) = \{ 2, 3 \}$

$\text{Indeg}(x) = \text{Indeg}(1) = 1$

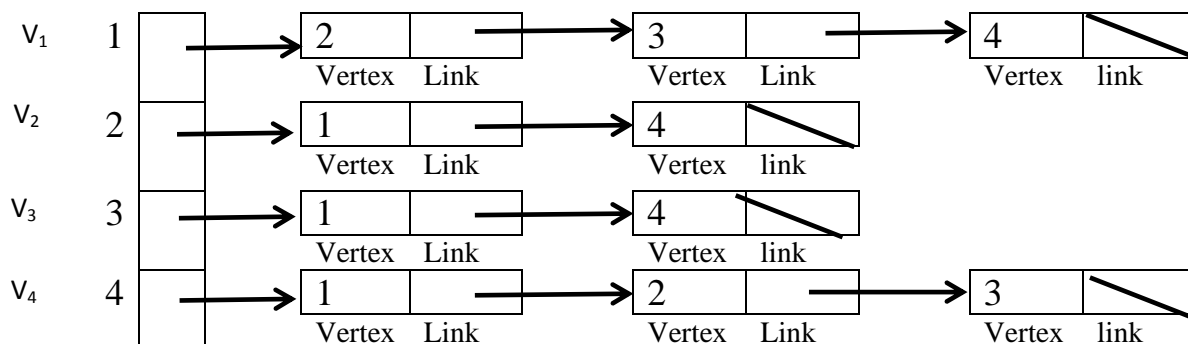
$\text{Outdeg}(x) = \text{Outdeg}(1) = 2$

```

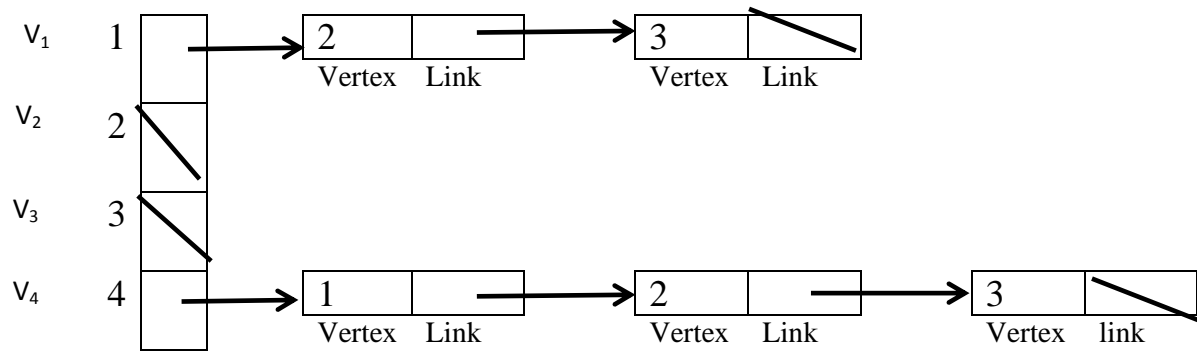
class GraphNode
{
    int vertex ;
    GraphNode link ;
    .....
}
  
```

`GraphNode adj[4]; // undirected or directed Graph`

Undirected Graph : Above Undirected Graph



Directed Graph : Above Directed Graph



8.3 Graph Traversing :

Graph traversing means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversing, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

The Graph has two types of traversing algorithms. These are called :

1- *Depth First Search (DFS)*

2- *Breath First Search (BFS)*

1- Depth First Search (DFS) :

Idea : (Algorithm)

a- Make following initializations :

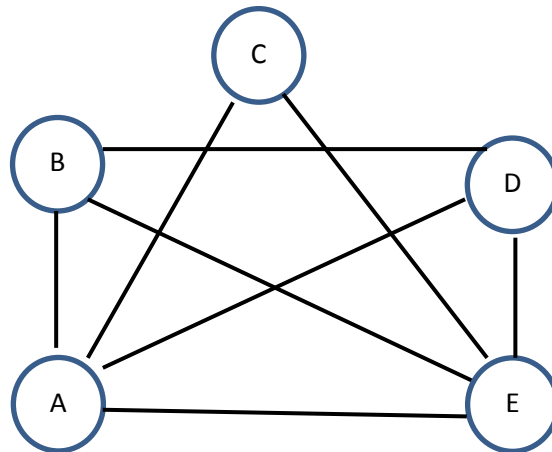
- Determine Start Vertex.
- Define a Stack S , push(Start Vertex , S).
- Define a one dimensional Array (*visited*) with length equal to the number of Vertices in the Graph initialized with false values.
- Define the *adjacent lists* for the Graph.

b- Body of algorithm :

```
Repeat
  pop ( S , top ) ;
  if ( ! visited[top] )
  {
    visit ( top ) ;
    visited[top] = true ;
    push into stack S all unvisited vertices adjacent to vertex just popped;
  }
until empty ( S ) ;
```

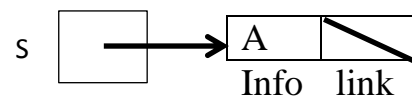
Example :

Traverse the following graph starting at vertex A using **DFS** algorithm :



a- Init :

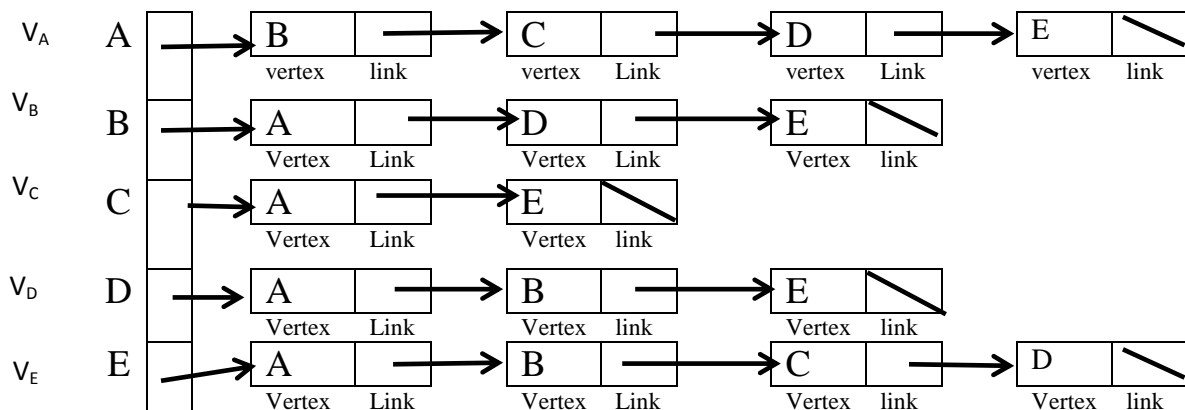
- push(S,A) →



- visited


A	F
B	F
C	F
D	F
E	F

- Adjacent Lists :



b- Body of algorithm :

repeat

pop (S,top) \rightarrow s  and top = A

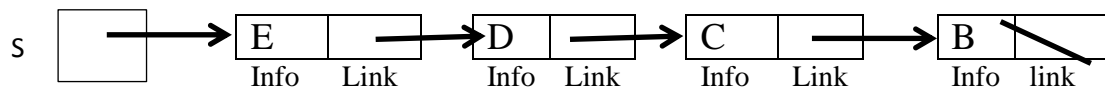
if (!visited[A]) \rightarrow True \rightarrow

1- visit(top) \rightarrow output : A

2- visited[top] = visited [A] = true \rightarrow

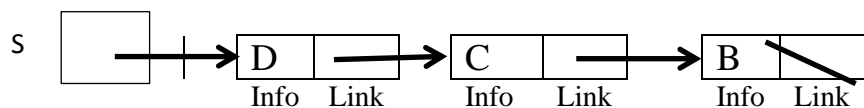
visited	
A	T
B	F
C	F
D	F
E	F

3- push(B,C,D,E) \rightarrow // push all unvisited vertices adjacent to A



Stack is not empty ? \rightarrow yes \rightarrow again

pop (S,top) \rightarrow top = E



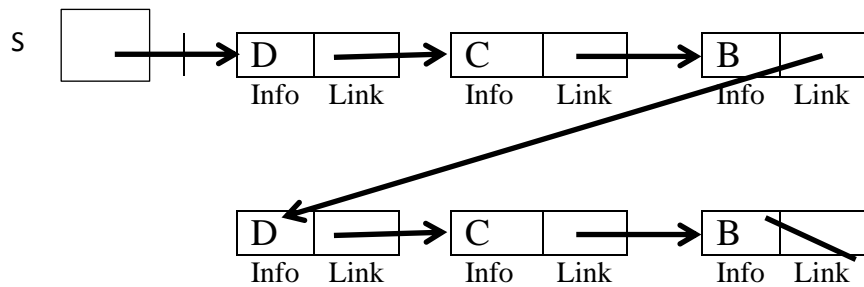
if (!visited[E]) \rightarrow True \rightarrow

1- visit(top) \rightarrow output : A , E

2- visited[top] = visited [E] = true \rightarrow

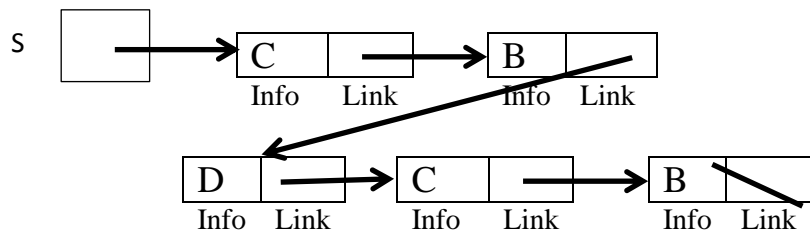
visited	
A	T
B	F
C	F
D	F
E	T

3- push(B,C,D) → // push all unvisited vertices adjacent to E



Stack is not empty ? → yes → again

pop (S,top) → top = D



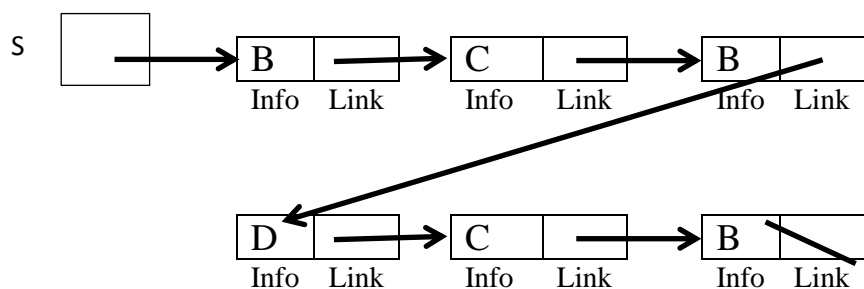
if (!visited[D]) → True →

1- visit(top) → output : A , E , D

2- visited[top] = visited [D] = true →

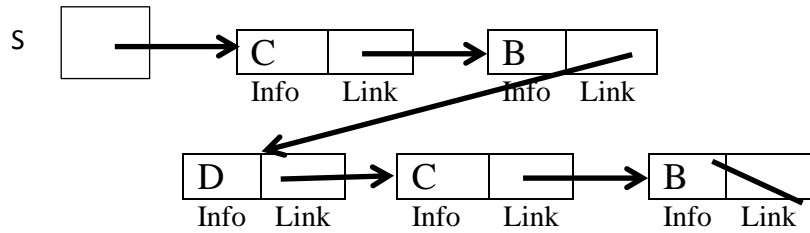
visited	
A	T
B	F
C	F
D	T
E	T

3- push(B) → // push all unvisited vertices adjacent to D



Stack is not empty ? → yes → again

pop (S,top) → top = B



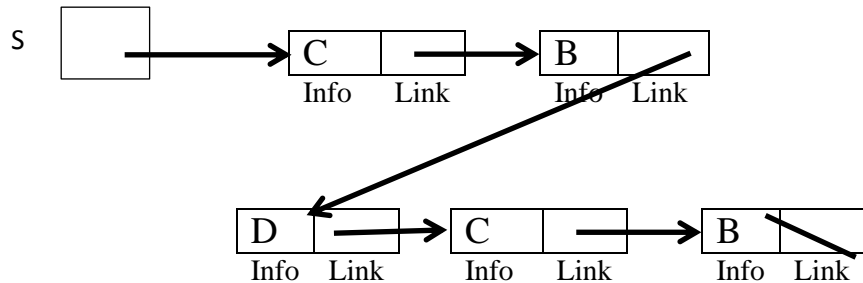
if (!visited[B]) → True →

1- visit(top) → output : A , E , D , B

2- visited[top] = visited [B] = true →

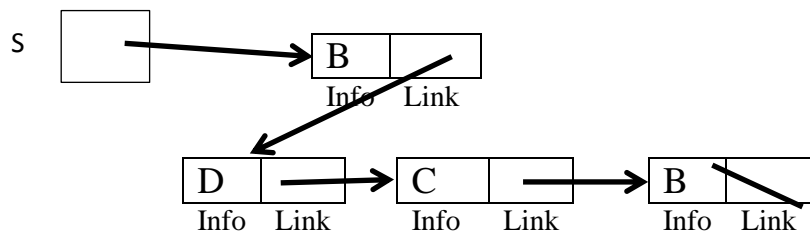
visited	
A	T
B	T
C	F
D	T
E	T

3- **Nothing to push** // all vertices adjacent to B are visited



Stack is not empty ? → yes → again

pop (S,top) → top = C



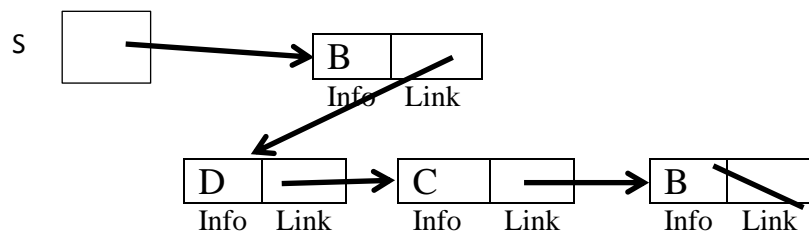
if (!visited[C]) → True →

1- visit(top) → output : A , E , D , B ,C

2- visited[top] = visited [C] = true →

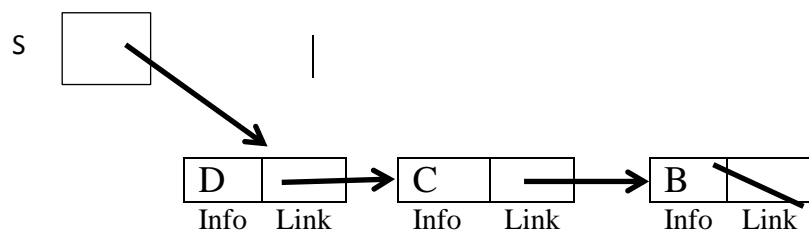
visited	
A	T
B	T
C	T
D	T
E	T

3- **Nothing to push** // all vertices adjacent to C are visited



Stack is not empty ? → yes → again

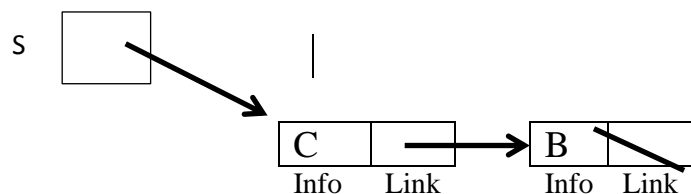
pop (S,top) → top = B



if (!visited[B]) → False → Nothing to do

Stack is not empty ? → yes → again

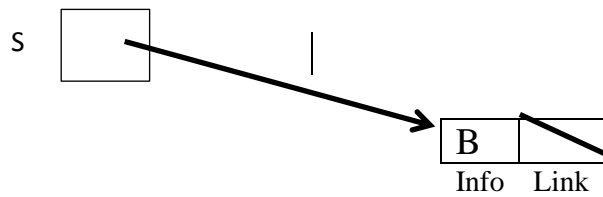
pop (S,top) → top = D



if (!visited[D]) → False → Nothing to do

Stack is not empty ? → yes → again

pop (S,top) → top = C



if (!visited[C]) → False → Nothing to do

Stack is not empty ? → yes → again

pop (S,top) → top = B

Final Output : A , E , D , B , C



if (!visited[B]) → False →

Nothing to do

Stack is not empty ? → No → STOP!!!!

2- Breadth First Search (BFS) :

Idea : (Algorithm)

a- Make following initializations :

- Determine Start Vertex.
- Define a Queue Q , enqueue(Start Vertex , Q).
- Define a one dimensional Array (visited) with length equal to the number of Vertices in the Graph initialized with false values.
- Define the adjacent lists for the Graph.

b- Body of algorithm :

BEGIN

enqueue (i , Q) ; // i = Start vertex

visited[i] = true ;

while (! empty(Q))

{

 dequeue (Q , first) ;

 visit(first) ;

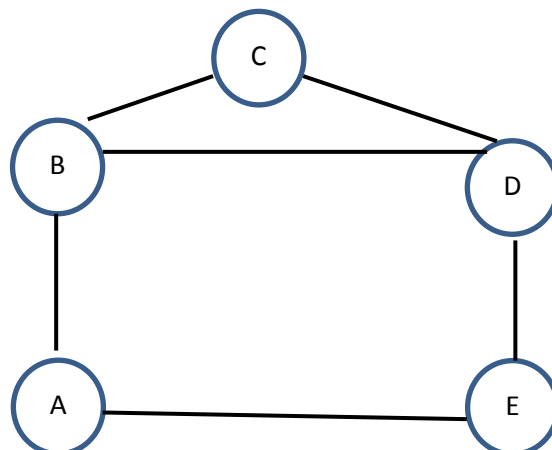
 Enqueue all '*unvisited*' vertices *adjacent* to first and marking them '*visited*';

}

END;

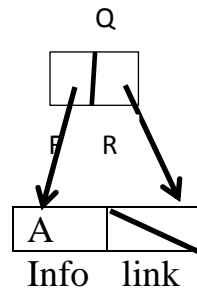
Example :

Traverse the following graph starting at vertex A using **BFS** algorithm :



a- Init :

- Start Vertex : $i = A$
- enqueue(A , Q) →



- visited

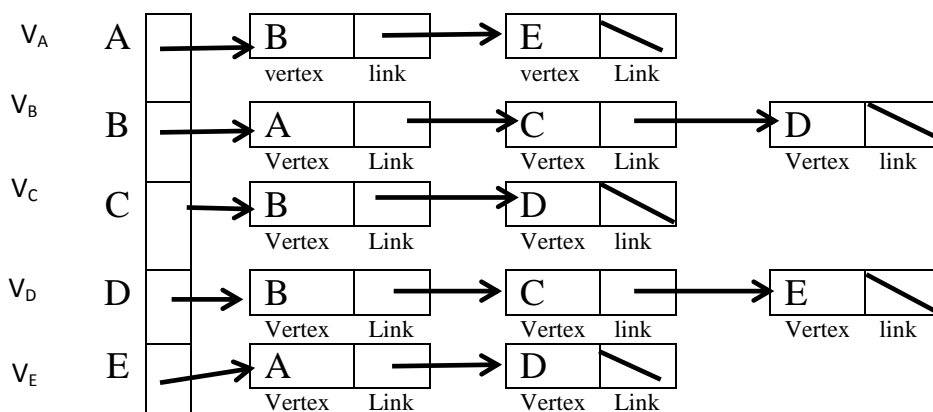
A	F
B	F
C	F
D	F
E	F

- visited[A] = true →

- visited

A	T
B	F
C	F
D	F
E	F

- Adjacent Lists :

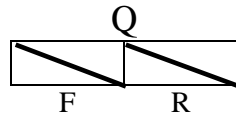


b- Body of algorithm :

while (!empty(Q)) → yes

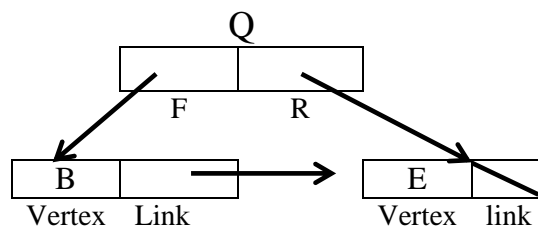
1- dequeue(Q , first) → first = A

2- visit(first) → output : A



visited	
A	T
B	F
C	F
D	F
E	F

3- enqueue(B , E) → // enqueue all unvisited vertices adjacent to A
// and marking them 'visited'

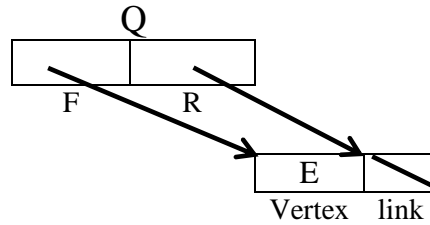


visited	
A	T
B	T
C	F
D	F
E	T

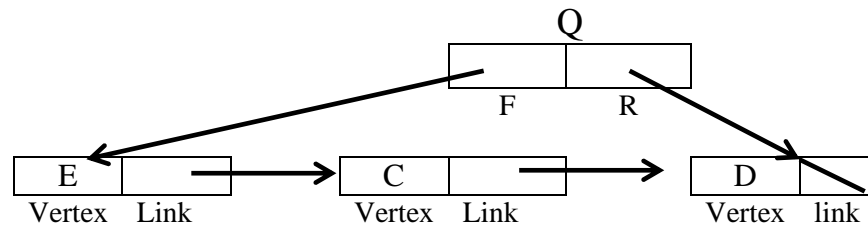
while (!empty(Q)) \rightarrow yes

1- dequeue(Q , first) \rightarrow first = B

2- visit(first) \rightarrow output : A , B



3- enqueue(C , D) \rightarrow // enqueue all unvisited vertices adjacent to B
// and marking them 'visited'

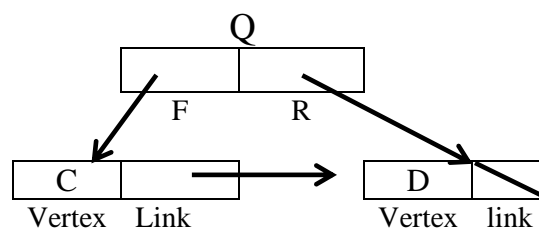


visited	
A	T
B	T
C	T
D	T
E	T

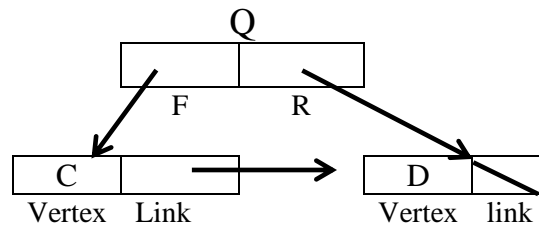
while (!empty(Q)) \rightarrow yes

1- dequeue(Q , first) \rightarrow first = E

2- visit(first) \rightarrow output : A , B , E



3- Nothing to enqueue → // all vertices adjacent to E are 'visited'



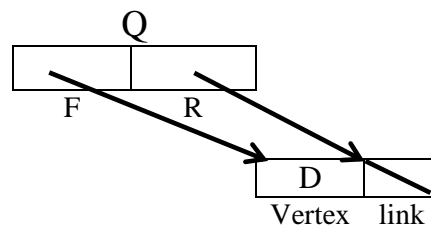
visited

A	T
B	T
C	T
D	T
E	T

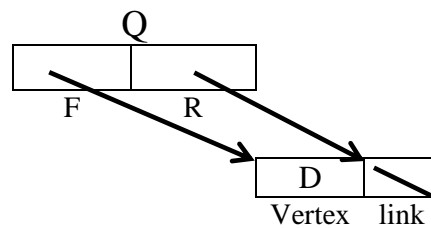
while (!empty(Q)) → yes

1- dequeue(Q , first) → first = C

2- visit(first) → output : A , B , E , C



3- Nothing to enqueue → // all vertices adjacent to C are 'visited'



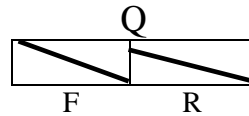
visited

A	T
B	T
C	T
D	T
E	T

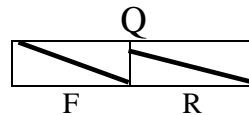
while (!empty(Q)) \rightarrow yes

1- dequeue(Q , first) \rightarrow first = D

2- visit(first) \rightarrow output : A , B , E , C , D



3- Nothing to enqueue \rightarrow // all vertices adjacent to D are 'visited'



visited	
A	T
B	T
C	T
D	T
E	T

while (!empty(Q)) \rightarrow No \rightarrow Nothing to do
STOP!!!

FINAL Output : A , B , E , C , D