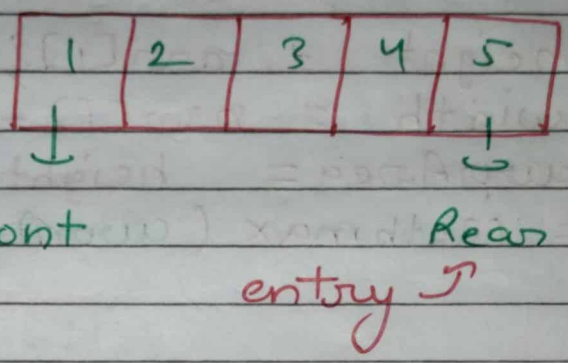
Queue



Operations :-

Add (Enque) → O(1)

Remove (Dequeue) → O(1)

Peek (Front) → O(1)

Queue using stack

```
static class Queue {

    static int rear;
    static int arr[];
    static int size;

    Queue ( int n) {
```

```
    arr = new int[n];
    size = n;
    rear = -1;
}

  public static boolean isEmpty (){

    return rear == -1;
  }

public static void add( int data ){

  if ( rear == size-1){
    System.out.println(" queue is Empty ")
    return;
  }
  rear = rear+1;
  arr[rear] = ~~arr~~ data;
}

public static int remove (){

  if ( isEmpty ()){

  System.out.println(" queue is Empty ");
    return -1;
  }

  int front = arr[0];
for( int i=0; i<rear; i++){
      arr[i] = arr[i+1];
  }
```

```
        rear --;
        return front;
    }

    public static int peek(){
    if ( isEmpty() ){
System.out.println("queue is Empty");
    return -1;
    }
        return arr[0];
    }
}
```

Circular queue → using array

```
Static class Queue {

    static int array[];
    Static int size;
    static int front;
    static int rear;

        Queue( int n ) {
            size = n;
            array = new int[n];
            front = rear = -1;
        }
```

```
public static boolean isEmpty ( ){
    return ( front == -1 && rear == -1);
}

public static boolean isFull () {
    return ((rear+1)%.size == front) ;
}

public static void add ( int data) {
  if ( isFull () ) {
    system.out.println(" Queue is full ! ");
    return;
  }

  if ( front == -1) {
      front = 0;
  }

  rear = (rear+1) %. size ;
  array [rear] = data;
}

public static int void remove ( ) {
  if ( isEmpty ()) {
  System.out.println(" Queue is Empty ! ");
  return -1;
  }
```

```
int result = array[front];

if( front == rear ){
    front = rear = -1;
}
else{
    front = (front + 1) % size;
}
return result;
}

public static int peek( ){

    if( isEmpty()){
    System.out.println(" Queue is Empty! ");
    return -1;
    }
    return array[front];
}
```

Queue using Linked list

```
Static class Node {
        int data;
        Node next;

        Node( int data){
            this.data = data;
```

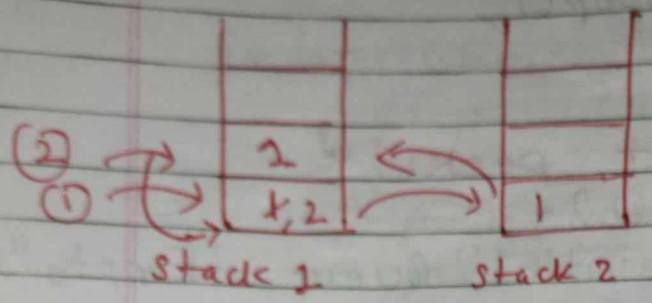
```
    this.next = null;
  }

  static class Queue{

static Node head = null;
static Node tail = null;

  public static boolean isEmpty(){

    return (head == null && tail==null)
  }

public static void add(int data){
  Node newNode = new Node(data);
   if( head == null) {
       head = tail = newNode;
    }
    tail.next = newNode;
     tail = newNode;
  }
   return

public static int remove(){

 if(isEmpty()){
 System.out.println(" Queue is Empty ");
 return-1;
 }
 result = head.data;
 if( head == tail){
```

```
    head = tail = null;
  }
  else {
      head = head.next;
    }

  return result;

}

public static ~~void~~ int peek() {

  if( isEmpty()) {

  System.out.println("Queue is empty!")
  return -1;
  }

  return head.data;

}
```

Queue using collection framwark

```
Queue<DataType> q = new
                ArrayDeque/LinkedList<>();
```

Queue using 2 stacks




```
Static class Queue {

 Static Stack<Integer> s1 = new stack<>();
 Static Stack<Integer> s2 = new stack<>();

  public static boolean isEmpty() {
        return s1.isEmpty();
  }

  public static void add(int data) {
        while (!s1.isEmpty()) {
            s2.push(s1.pop());
        }
        s1.push(data);
        while (!s2.isEmpty()) {
            s1.push(s2.pop());
        }
  }

  public static int remove() {
     If(s1.isEmpty()) {
        System.out.println("Queue is Empty");
```

```
        return -1;
    }
    return s1.pop();
}

public static int peek() {
    if( s1.isEmpty() ) {
        System.out.println(" Queue is Empty ");
        return -1;
    }
    return s1.peek();
}
}
```

Stack using 2 Queues

```
static class stack {
    static Queue<Integer> q1 = new
                    LinkedList<>();
    static Queue<Integer> q2 = new
                    LinkedList<>();

public static boolean isEmpty() {
    return ( q1.isEmpty && q2.isEmpty());
}
```

```
public static void push (int data) {

 ~~if (!q1.isEmpty) {~~
        ~~q2.push (q1.pop());~~
      ~~}~~
      ~~q1.push (data);~~
 ~~if (!q2.isEmpty) {~~
      ~~q1.push (q2.pop());~~
    ~~}~~

 if ( !q1.isEmpty()) {

      q1.push (data);
  }
  else {
        q2.push (data);
      }
 }

public static ~~void~~ int pop() {

 if( isEmpty()) {
   System.out.println(" stack is Empty");
   return -1;
  }
      int top = -1;
 if( !q1.isEmpty) {

  while( !q1.isEmpty) {
        top = q1.remove();
```

```
        if ( q1.isEmpty()) {
            break;
        }
        q2.add (top);
    }}
else {
    while (!q2.isEmpty) {
top = q2.remove();
    if ( q2.isEmpty()) {
            break ;
        }
        q1.add (top);
    }}

    return top ;
}

public static int peck() {

    /* remove the booth if conditions
    from the pop function in the
    both while loops and it will
    be work as peck() function */

}
```

First non-repeating letter in a stream of characters.

example a a b c c x b

output a, -1, b, b, b, b, x

Code

```
public static void streamStoring (String str
{
  Queue<character> q = new LinkedList<>(
  int freq[] = new int[26];

for (int i=0; i<str.length(); i++) {
   char chr = str.charAt(i);
   freq[chr - 'a']++;
   q.add(chr);

while ( !q.isEmpty && freq[q.peek -'a']
{
      q.remove();
}
if (q.isEmpty()){
  System.out.print(" "+ -1);
}
else {
System.out.print(" "+q.peek());
}
}
}
```

Interleave 2 Halves of a Queue (even length)

example	1	2	3	4	5	6	7	8	9	10
output	1	6	2	7	3	8	4	9	5	10

```
public static void interleave(Queue<Integer
                                q) {

Queue<Integer> firstHalf = new
                LinkedList<>();
 int size = q.size();

for( int i=0; i<size/2; i++) {
    firstHalf.add(q.remove());
  }
while( !firstHalf.isEmpty()) {
    q.add(firstHalf.remove());
    q.add(q.remove());
  }
}
```

Queue Reversal

code →

```
public void reverseQueue (Queue<Integer> q){

Stack<Integer> s = new Stack<>();

while (!q.isEmpty()) {
        s.push (q.remove());
    }

while (!s.isEmpty()) {
        q.add (s.pop());
    }
}
```

Deque
↓
Double ended queue

It allows us to add and remove a element in the queue from both of the side and it called deque we can use it from java collection framework

* available methods
  * addfirst()
  * addLast()
  * removefirst()
  * removeLast()
  * getFirst()
  * getLast()

peek()

Stack and Queue Using Deque

stack	queue

```
Deque<Integer> deque = new LinkedList<>();
```

```
push(int x) {
deque.addLast(x);
}
```

```
add() {
deque.addLast(x);
}
```

```
pop() {
return deque.removeLast();
}
```

```
remove() {
return deque.remove first();
}
```

```
peek() {
return deque.getLast();
}
```

```
peek() {
return deque.getfirst()
```

Tree

set of nodes

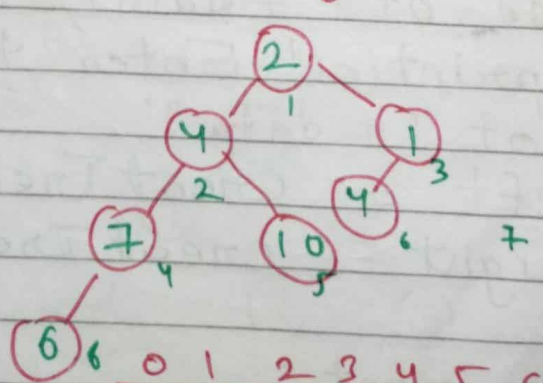
* every node can have a parent node.
* It may have zero or more child.
* The topmost node called the root node.
* The nodes which doesn't have any child called leaf nodes

Binary tree

* Maximum numbers of nodes at level i ⇒ 2^i.

* Maximum number of nodes ~~at~~ in a tree with height h is (2^h-1)

* Array (storing)



Array =

0	1	2	3	4	5	6	7	8	9	10
X	2	4	1	7	10	4		6		

L = 2×i

R = 2×i+1

Parent = i/2

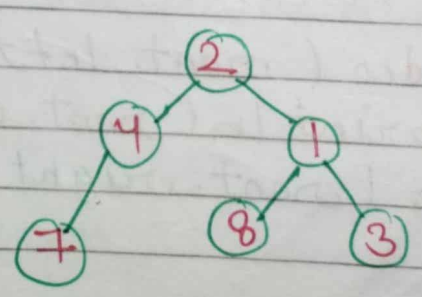
```
package tree;
import java.util.*;

public class main () {

static Scanner sc = new Scanner(System.in);

Public static void main( String[] args) {

        creatTree();
}

static Node creatTree () {

Node root = null;
System.out.println("Enter data");
 int data = sc.nextInt();

If(data == -1) {
        return null;
  }

System.out.println("Enter the left
        node of "+data);
System.out.println("Enter the right
    node of " + data);
    root.left = creatTree();
    root.right = creatTree();

    return root;
}}
```

```
Class Node {
    Node left, right;
    int data;
        Node (int data) {
            this.data = data;
        }
}
```

Binary tree traversals

1. Inorder traversal (L Root R)
2. Preorder traversal (Root L R)
3. Post order traversal (L R Root)

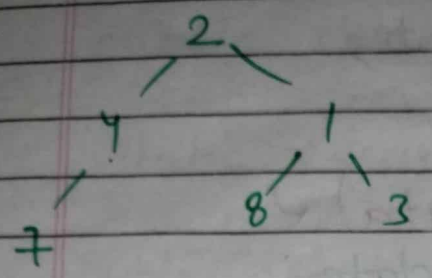
example



In Order → 7 4 2 8 1 3

2 Pre order

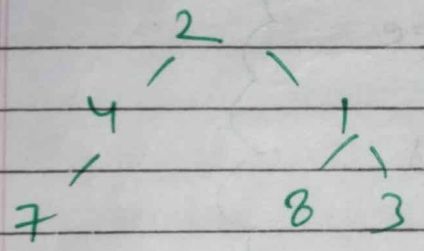
Root L R
↳ Node



~~2 7~~ 4
2, 4, 7, 1, 8, 3

(2) Post order

L R Root
↳ Node



7, 4, 8, 3, 1, 2

In order code

↳ Proceed Node first traversing L and the R

```
public void InOrder(Node root){

if( root == null) { return; }

        InOrder( root.left);
System.out.println( root.data);
        InOrder( root.right);

}
```

Pre Order Code

↳ Proceed N first then travest L and R

```
Public void PreOrder( Node root){

  If(root == null) return;

  System.out.println(root.data);
    PreOrder (root.left);
    PreOrder (root.Right);

}
```

Post Order Code

↳ travers L then R and then Proceed N

```
Public void postOrder ( Node root) {

  If( root == null) return ;

    postOrder (root.left ) ;
    post Order ( root.Right ) ;

  System.out.println( root.data );

}
```

Pre Order Code

↳ Procced N first then travest L and R

```
Public void PreOrder( Node root) {

  If(root == null) return;

  System.out.println(root.data);
    PreOrder (root.left);
    PreOrder (root.Right);

}
```

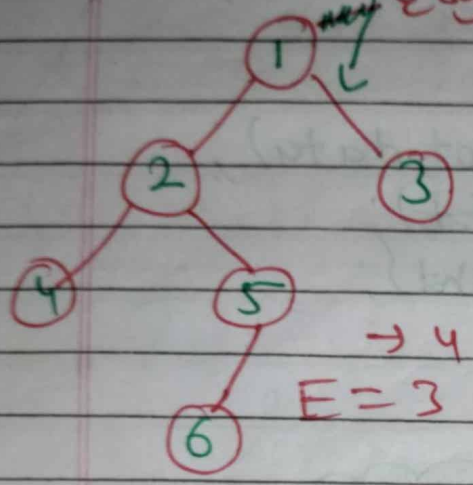
Post Order Code

↳ travers L then R and then Procced N

```
Public void postOrder ( Node root) {

  If( root == null) return ;

   postOrder (root.left ) ;
   post Order ( root.Right ) ;

  System.out.println( root.data );

}
```

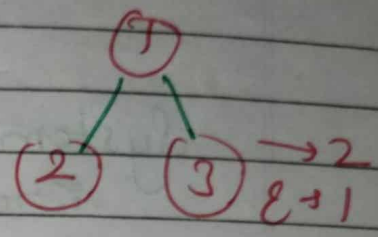
Height of Binary Tree

↳ farest leaf node



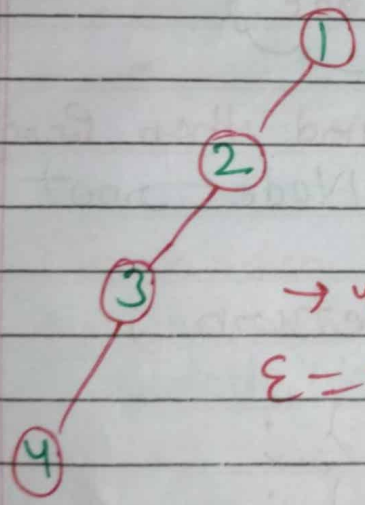
→ 4
E = 3

1 → 1
E = 0



→ 2
E → 1

null → 0

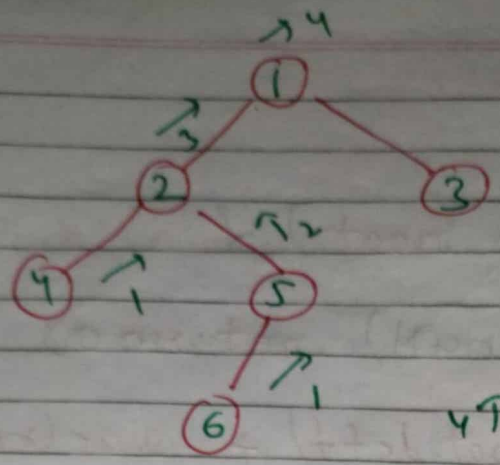


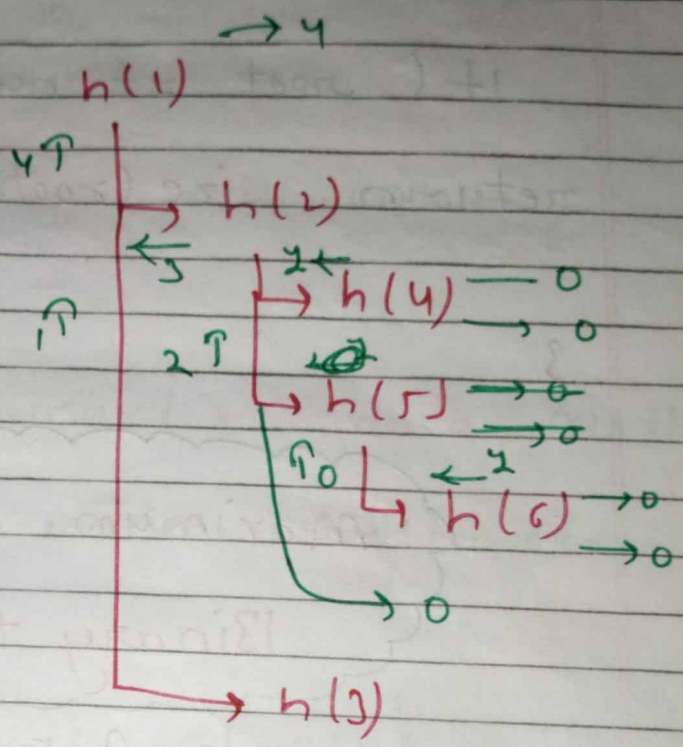
→ 4
E = 3

Code

```
int height( Node root) {
  if ( root == null) return 0;
  return Math.max( height ( ~~left~~ root.left),
      height (root.right) ) + 1;
}
```

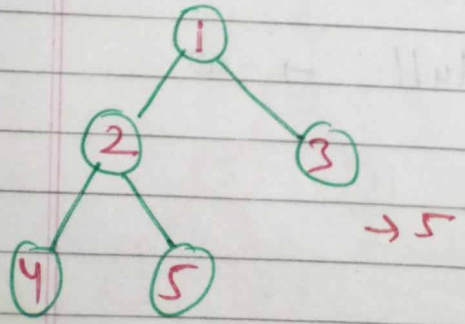
Recursive callstack

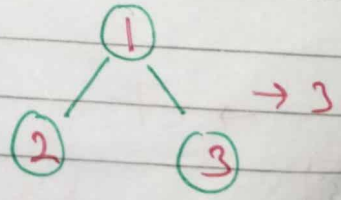




Size of Binary tree

↳ How many nodes in a B tree.





(1) → 1

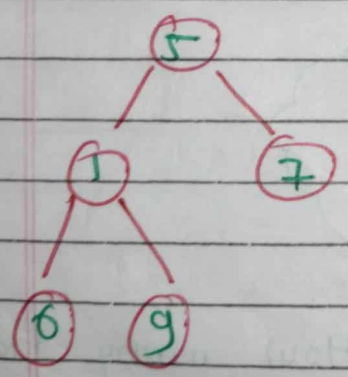
null → 0

Code

```
int size ( Node root ) {
    if ( root == null) return 0;
return size (root.left) + size (root.left)
                    +1;
}
```

Maximum In a Binary tree

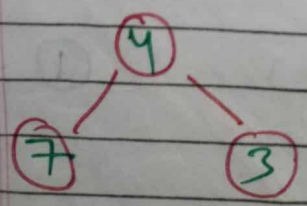
↳ A node having maximum value data.



→ 9

4 → 4

Null → $-\infty$



→ 7

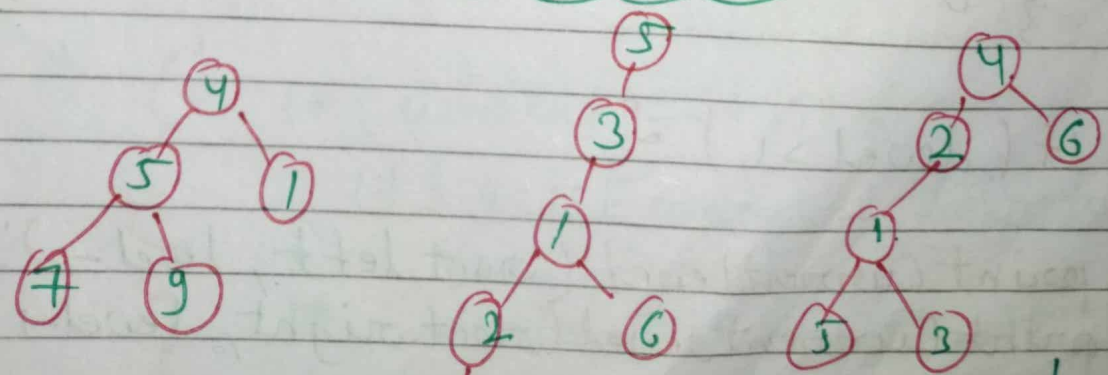
Code

```
int maximumNode ( Node root) {
  if( root = null ) {
  return Integer . MIN-VALUE ;
  }

return math.max ( root.data , math.max(
maximumN(root. left) , maximumNode (root.Right)) ;
}
```

Time Complexity → O(n)

Space Complexity → O(n)

Level Order traversal



(4 5 1 7 9) (5 3 1 2 6 4) (4 2 6 1 5 3)

Code

```
~~static~~ void ~~printLevel~~(Node ~~root~~, int
level, int height) {

(1) Static void printLevelOrder(Node root,
int height) {

for(int i=1; i<=height; i++) {

printCurrentLevel(~~Node~~ root, i);

}

printCurrentLevel(Node root, int level)
{
if(root == null) return;

if(level ==1) {
System.out.println(root.data);
}

if(level >1) {

printCurrentLevel(root.left, level-1);
printCurrentLevel(root.right, level-1);
}}
```

Time complexity → $O(n^2)$

Next code in $O(n)$

```
static void printLevelOrder( Node root) {

Queue<Node> q = new LinkedList<>();
  q.add(root);
                q.add(null);
while( ! q.isEmpty() ) {

  Node current = q.poll();
System.out.print( current.data + " ");

  If( current.left != null) {
      q.add( current.left);
  }

  If( current.right != null) {
      q.add( current.right);
  }
}
}
```

```
If( current == null) {
    If( q.isEmpty()) { return; }
        q.add(null);
        System.out.println();
    }   continue;
```

Another type of code

for → Pre Order
In-Order
Post-Order
Level ~~list~~ Order

```
class solution {

    class Node
    {
        int data ;
        Node left, right ;

        Node () { }
        Node (int data) {
        this.data = data ;
        this.left = this.right = null;
        }
```

(1) InOrder

```
public static List<Integer> findInOrder (
                Node root) {

List<Integer> InOrderList = new
            ArrayList<>();

    if( root == null) { return InOrderList; }
```

addAll

```
InOrder.add( FindInOrder(root.left));
InOrder.add( root.data );
InOrder.addAll( FindInOrder( root.right));
  return InOrderList ;
}
```

(2) Pre-order

```
Public static List<Integer> PreOrder( Node root){
  List<Integer> PreOrderList = new
                ArrayList<>();
  If( root == null) { return PreOrderList; }
  PreOrderList.add (root.data);
PreOrderList.addAll( PreOrder(root.left));
PreOrderList.addAll( PreOrder(root.right));
    return PreOrderList ;
}
```

(3) Post-order:

```
Public static List<Integer> PostOrder( Node root){
```

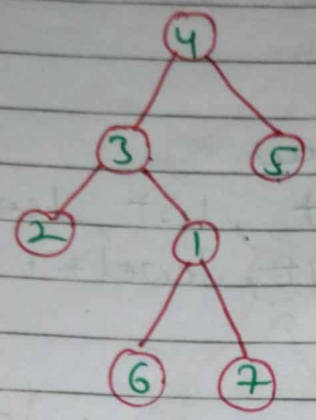
```
List<Integer> PostList = new ArrayList<>();
if(root = null) {
    return PostList;
  }

PostList.addAll(PostOrder(root.left));
PostList.addAll(PostOrder(root.Right));
PostList.add(root.data);

return PostList;
}
```

(4) Level-Order

```
public static List<Integer> levelOrder(Node root){

List<Integer> levelList = new ArrayList<>();

if(root == null) {
    return levelList;
  }

Queue<Node> q = new LinkedList<>();

    q.add(root);

while(!q.isEmpty()){
```

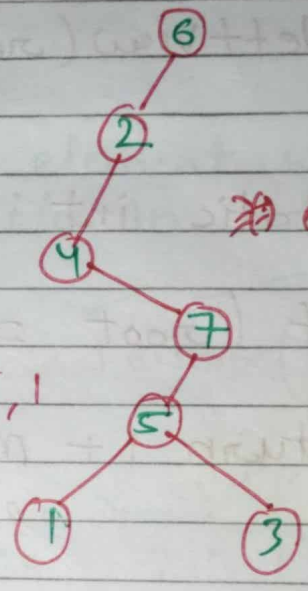
```
Node current = ~~q.pop~~ q.poll();

~~find~~
    levelList.add ( current.dada);

If ( current.left != null ) {
        q.add(current.left);
    }

If ( current.right != null) {
        q.add ( current.right);
    }
}
    return levelList ;

}
}
```

Left view and Right view



Ex 4, 5, 1, 7

Ex 4, 3, 2, 6



Ex 6, 2, 4, 7, 5, 1

Ex 6, 2, 4, 7, 5, 1

Code

```
static void LeftView ( Node root) {

    int maxDepth = maxDepth (root);

List<Node> list = new ArrayList<> (collections.
                nCopies ( maxDepth , null));

  setLeftView( root, list , 0) ;

for( int i=0 ; i<list.size(); i++) {
System.out.println(list.get(i).data+ " ");
}}
```

```
static void SetLeftView( Node root,
    List<Node> list , int level) {

  if (root == null) return ;

  list.Set (level. root);
  // It will give left view
  SetLeftView (root.right , list, level+1);
  SetLeftView (root.left , list, level+1);
}

  static int maxDepth( Node root) {

  if (root == null) return 0;

  return 1 + mathmax( maxDepth (root.left
                , maxDepth(root.right));

}
```

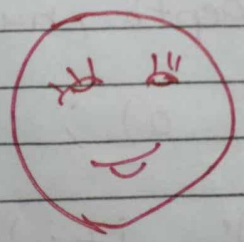
1st

2nd

1st ' " "

2nd ' " "

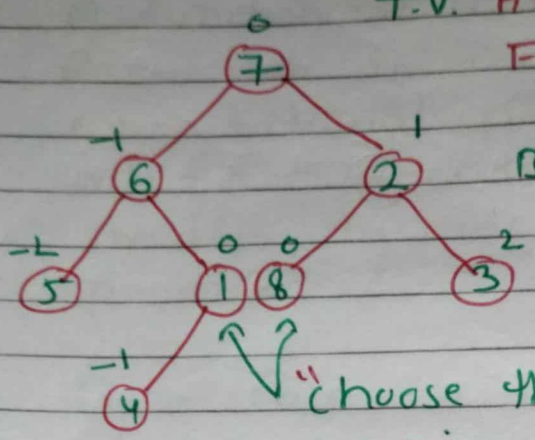
(replace 1st line with 2nd line and it will give right view)



Keep doing hard work =) =)

*important Topic

Top-view and Bottom view



T.V. At same level which is seen First from Top.

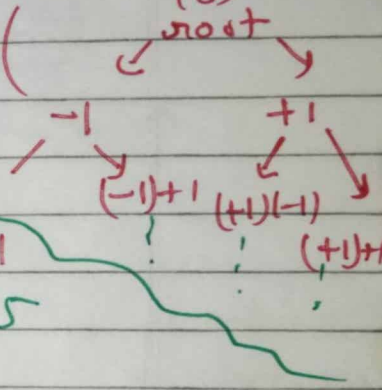
B.V. At same level which is seen first from bottom.

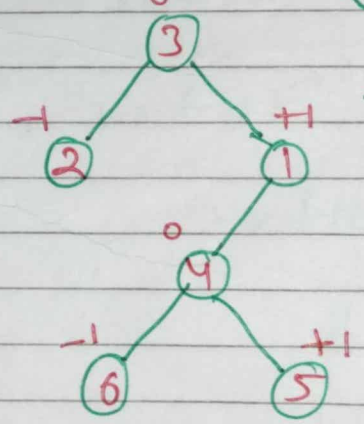
"choose the element which is comming after in this condition"

T.V. 5, 6, 7, 2, 3

B.V. 5, 4, 8, 2, 3

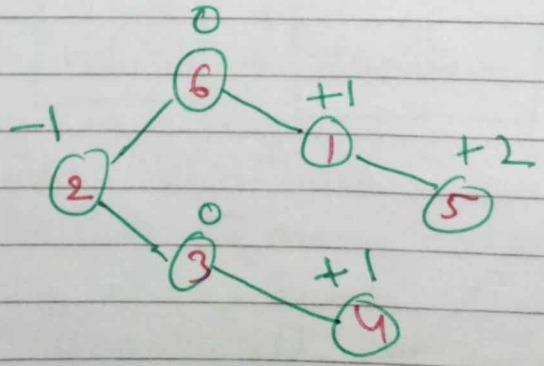
horizonal distane





T.V. 2 3 1

B.V. 6 4 5



T.V. 2, 6, 1, 5

B.V. 2, 3, 4, 5

* Min absolute Difference Pairs :-

→ Given two arrays A and B of equal length n. Pair each element of array A to an element in array B, such that sum S of absolute differences of all the pairs is minimum.

* first we will sort the array

```
public class Pairs {

    public static void main (String args[]) {

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

    Arrays.sort (A);
    Arrays.sort (B);

      int minDiff = 0;

    for( int i=0; i<A.length; i++) {
      minDiff += Math.abs ( A[i]-B[i]);
    }

  System.out.println( minDiff );
}
}
```

Max Length chain of Pairs

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c,d) can come after pair (a,b) if b<c.
Find the longest chain which can be formed from a given set of pairs.

pairs = (5,24) (39,60), (5,28), (27,40), (50,90)
ans = 3

```
Public class check{
 public static void main( String args[]){

int pair[][] = { {5,24}, {39,60}, {5,28},
    {27,40}, {50,90} };

Arrays.sort ( pair, Comparator.comparing
                Double ( o -> o[1]));

 int chainLength = 1;
 int chainEnd = pair[0][1];

 for (int i=1 ; i< pair.length ; i++){
     if ( pair[i][0] < chainEnd ){
         chainLength++;
         chainEnd = pair[i][1];
     }}
 System.out.println( chainLength); }
```

Indian Coins

We are given an infinite supply of denominations [1, 2, 5, 10, 20, 50, 100, 500, 2000]. Find min no. of coins/notes to make change for a value V

V = 121

ans = 3(100+20+1)

V = 590

ans = 4(500+50+20+20)

Code

```
public class Coins {
public static void main( String[] args) {

 int coins[ ] = { 2000, 500, 100, 50, 20, 10,
                  5, 2, 1 }
 int price = 590;
 int i = 0;
ArrayList< Integer> ans = new ArrayList<>();
 while ( price != 0 ) {
      if ( price >= coins [i] ) {
            price = price - coins [i];
ans.add (notes[i]) }
      if ( money < coins [i] ) {
                i++;
          }
  }
  for( int i=0 ; i< ans.size() ; i++) {
      System.out.print ( ans.get (i)+ " ");
  }}
```

Binary Search Tree

* What is a BST

↳ Binary Tree
↳ Left subtree Node < Root
↳ Right Subtree Node > Root
↳ Left and Right Subtrees are also BST with no duplicates.

Special Property :-
↳ Inorder traversal of BST gives a sorted sequence

Strategy:-
↳ Most problems will be solved using recursion i.e. by dividing into subproblems and making recursive calls on subtrees

Heaps → priority queues.

Concept :- (Priority queues)

we sort the data on the bases of priority.

Priority Queues in JCF :-

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

1	2	3	4	5

O(logn) pq.add(data);
O(1) pq.peek() → heighest priority element
O(logn) pq.remove()

→ By default smallest element will be priority element.

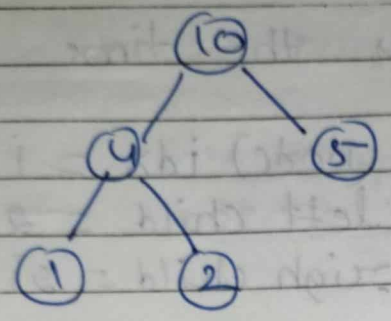
→ To change priority with largest

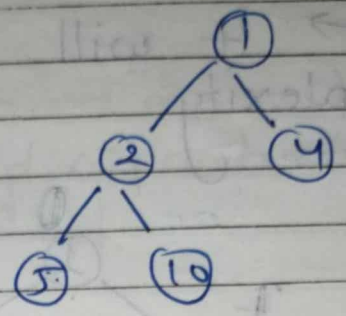
```
PriorityQueue<Integer> pq = new
PriorityQueue<>(Comparator.reverseOrder());
```

To Compain objech :-

```
public class Classroom {

  static class Student implements
      Comparable<student> {

    String name;
    int rank;

    public static student (string name,
                           int rank) {
        this.name = name;
        this.rank = rank;
    }

    @ Override

    public int compareTo (student s2) {

      return this.rank - s2.rank;
    }
```

Heap:-





max heap
max elem priority ↑
~~mini~~

min heap
min elem ~~prior~~ priority ↑

Visualize → Heap (BT)
Implement → Array.

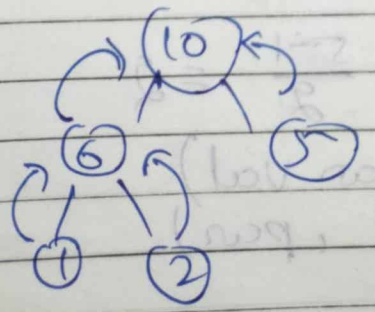
Properties:-

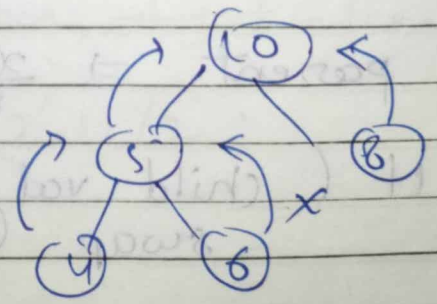
(i) at most 2 children. (BT)

(ii) Complete Binary tree.

(iii) Heap Order property

children >= Parent (min Heap)
children <= Parent (max Heap)

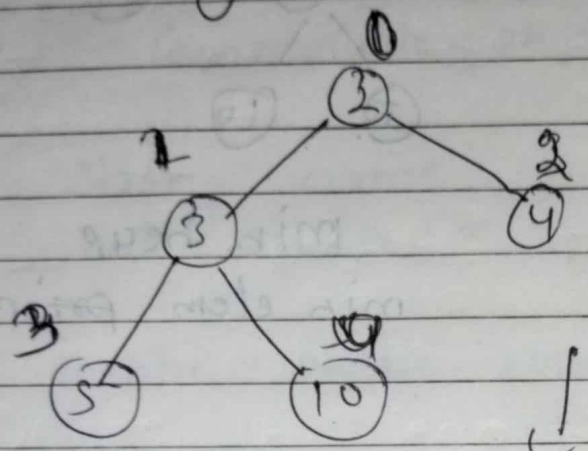




Heap is not implemented as a class:-

↳ It will increase the time complexity

(node) idx = i
left child = 2i+1
righ child = 2i+2



↓ Heap

Array :

2	3	4	5	10
0	1	2	3	4

So we will implement our heap as array and arraylist

⇒ **Insert in Heap:-**

2	3	4	5	10	1
0	1	2	3	4	5

Steps ①
(i) add at last
(ii) fix heap

$$\text{parent} \Rightarrow \frac{x-1}{2} = \frac{5-1}{2} = 2$$

if (child val < par val)
swap (child , par)

→ step 2

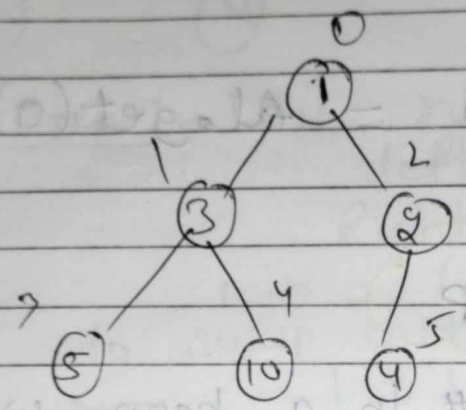
arr = | 2 | 3 | 1 | 5 | 10 | 4 |
0 1 2 3 4 5

now par → $\frac{x-1}{2} = \frac{2-1}{2} = \frac{1}{2} = 0$

swap (child, par)

| 1 | 3 | 2 | 5 | 10 | 4 |

→



code :-

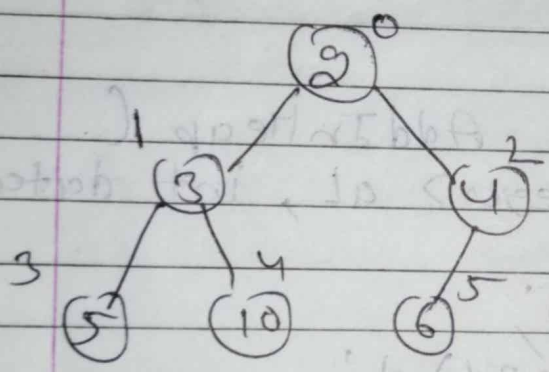
```
public static void AddInHeap (
    ArrayList<Integer> al, int data){

    al.add (data);
    int x = al.size () -1;
    int par = (x-1) /2 ;

while ( al.get(x) < al.get (par) ){
    int temp = al.get(x);
    al.set ( x, al.get (par) );
    al. set (par, ~~al.get~~ (temp) );
    x = par;
    par = (x-1) /2 ;
}
```

Get min in Heap (peek())

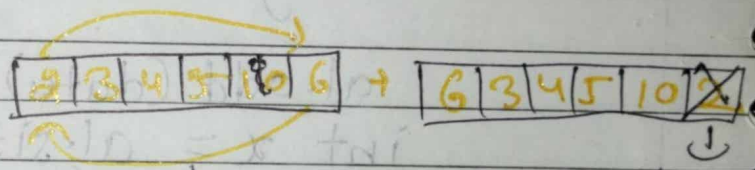
In an heap the minimum element is root so it will be at the starting index of arraylist.

So. minVal = Al.get(0);

Delete in Heap



In a heap we delete the root node of tree or the first element of arrlist



step 2 :- Swap 1st and last element

delete

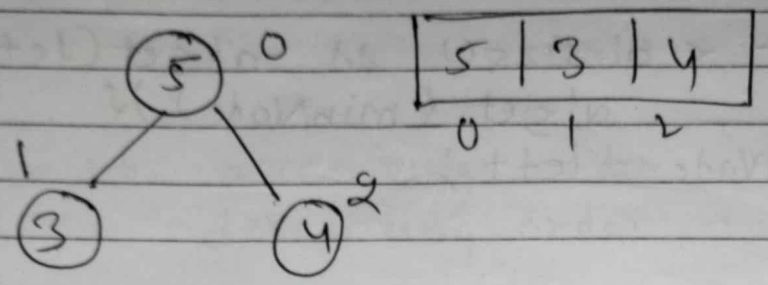
(2) remove last indx
arr.remove(arr.size()-1);

(3) fix my heap → heapify

to fix it we will do some opration called heapify let see it how does it works.

Heapify

O(log n)



<u>steps:-</u> (i) root = i

left ⇒ 2xi+1

Right → 2xi+2

min = (i, 2i+1, 2i+2)

to swap with root to min.

So now | 3 | 5 | 4 |

<u>code</u>

```
RemoveHeap( ArrayList<Integer> al ){

    int data = al.get(0);
    int temp = al.get(0);
    ~~al.get~~ al.set (0, al.get (al.size()-1));
     al.set ( al.size()-1, temp);

     al.remove ( al.size()-1);
     Heapify ( al, 0);

}


public static void Heapify ( ArrayList
                <Integer> al , int indx){
    int minNode = indx;
```

```
int left = (2*indx)+1;
int right = (2*indx)+2;

if ( left < al.size() && al.get(left) <
            al.get (minNode) ){
    minNode = left;
}

if ( Right < al.size() && al.get(right)
        < al.get (minNode) ) {
    minNode = right;
}

if (indx != minNode) {

    int temp = al.get (indx);
    al.set (indx, al.get (minNode));
    al.set (minNode, temp);

    Heapify ( al, minNode);
}
```

Page No.
Date

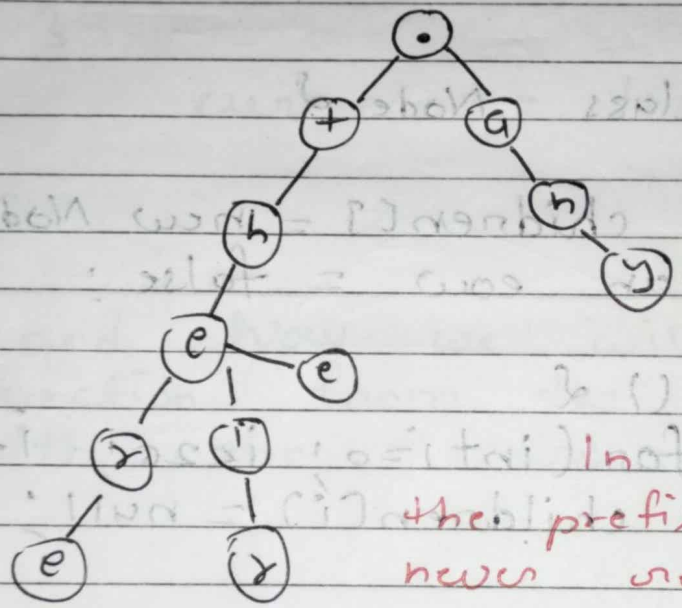
Heap Sort → O(nlogn)

arr = 1, 2, 4, 5, 3

→ If ascending order → creat max heap
→ If descending order → create min heap

Tries

words [] = "the", "a", "those", "their", "any", "thee"

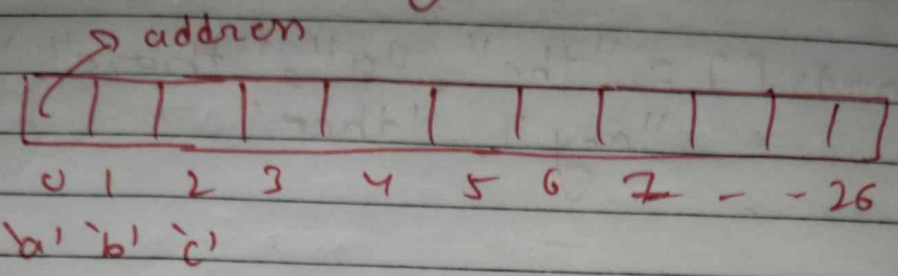


In tries we store the prefix of and we never repeat them

Creating a Trie

```
class Node {
    Node[] children;
    boolean endOfWord;
}
```

Here we are ~~avi~~ having an array having index 0 to 26



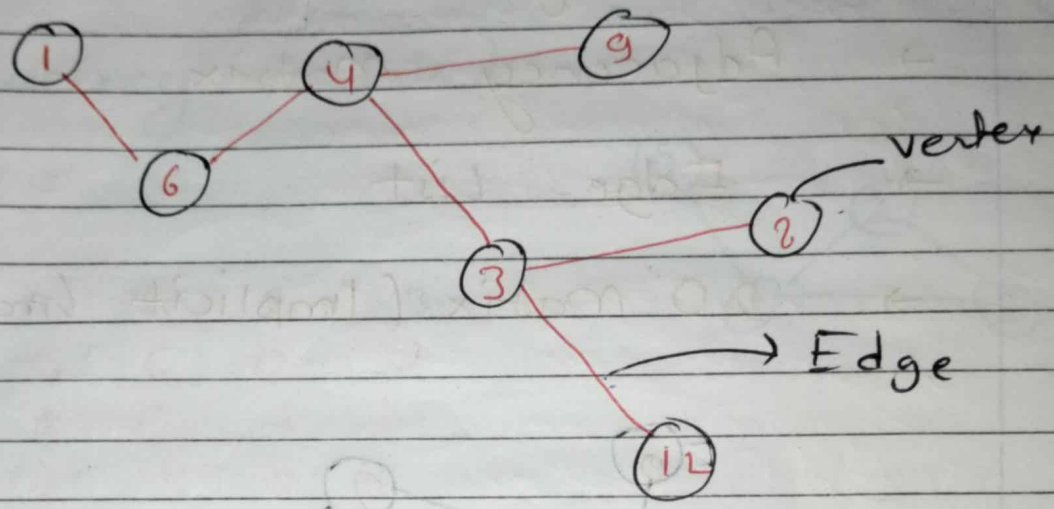

```
static class Node {

    Node children[] = new Node[26];
    boolean eow = false;

    Node() {
        for (int i=0; i<26; i++){
            children[i] = null;
        }
    }
}

public static Node root =
                new Node();

public static void insert (String word)
{
    Node curr = root;
    int
    for (level =0; level< word.length();
                level++){
```

```
int idex = word.charAt(level) - 'a';

if( curr.children[idex] == null) {

curr.children[idex] = new Node();
}

curr = curr.children[idex];

}

curr.eow = true;

}
```

// and how we will call insert function from the main function with the word.

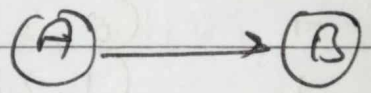
Graph

→ It is a network of nodes.

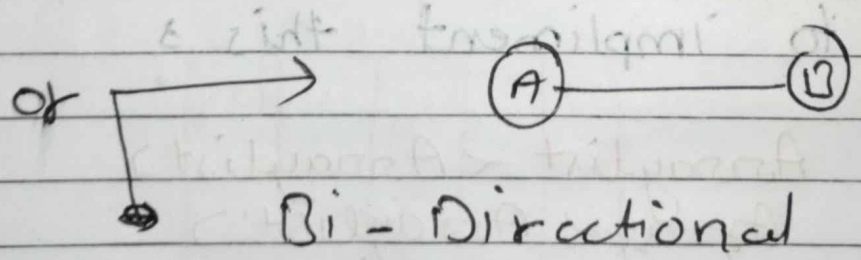


Types of Edges

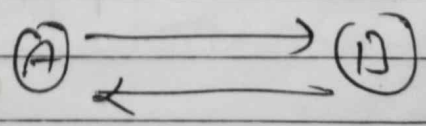
→ Uni Directional



→ Un - Directed



Bi - Directional



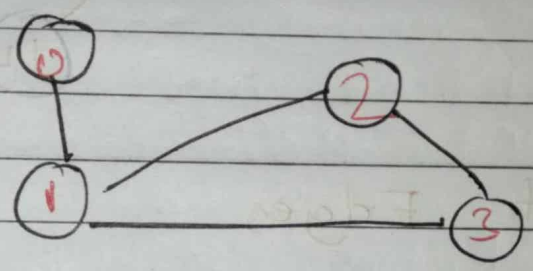
Storing a Graph

→ Adjacency list

→ Adjacency matrix

→ Edge List

→ 2D matrix (Implicit Graph)



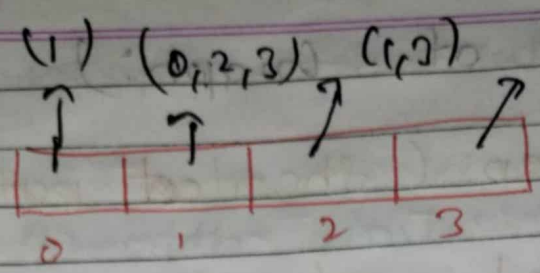
0 → (1)
1 → (0, 2, 3)
2 → (1, 3)
3 → (1, 2)

(Adjacency list)

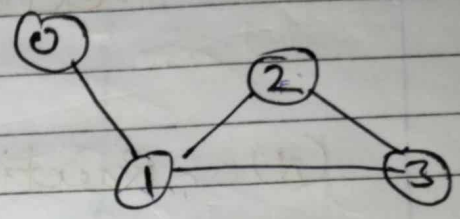
Ways to implement this :

(i) Arraylist < Arraylist >
(ii) ~~Array < Arraylist >~~
(iii) hashmap < int, list >

Vertex (key) — Value



(Adjacency matrix)

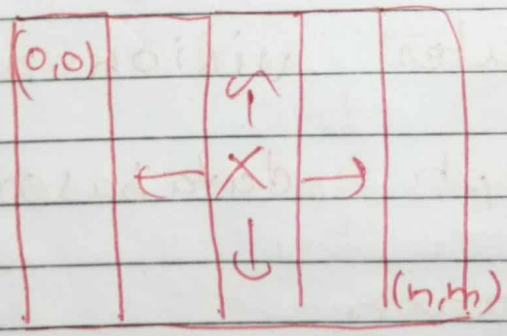


	0	1	2	3
0	0	1	0	0
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0

(Edge list)

Edges = { {0,1}, {1,2}, {1,3}, {2,3} }

(Implicit Graph)



(Applications of Graph:)

(a) Maps (shortest path)

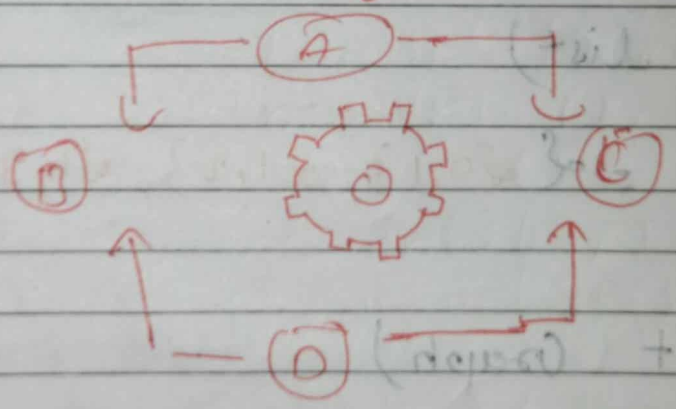
(b) Social networks

(c) Delivery network (shortest cyclic Route)

(d) Routing algorithms

(e) Machine learing (Computation Graphs)

(f) Dependency Graph



(g) Computer vision

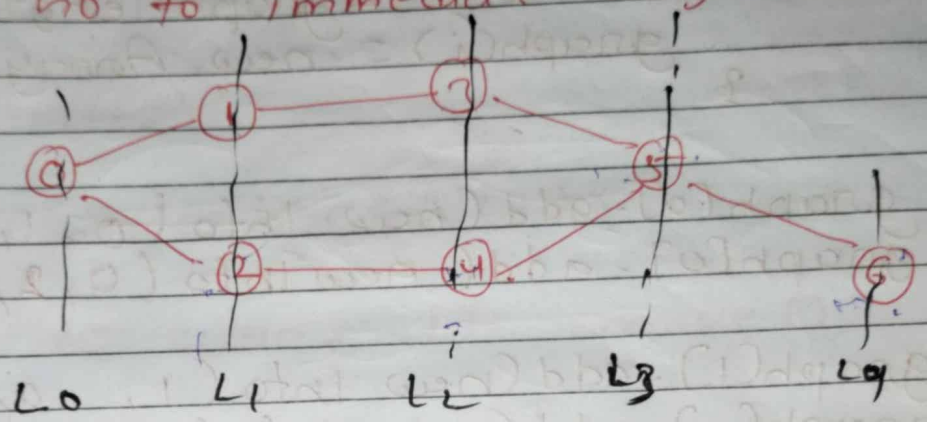
(H) Graph databases

(i) Regearch

Graph traversals

(i) Breadth first search (BFS) :-

↳ Go to immediate neighbors first



→ here will use the logic of first in first out so we will use queue data structure.

Code:-

~~Cre~~

Creating a class Edge

```
static class Edge {
        int scr;
        int dst;
        int weight;

public Edge( int scr, int dst, int w)
{
    this.scr = scr;
    this.dst = dst; this.weight = w;}
```

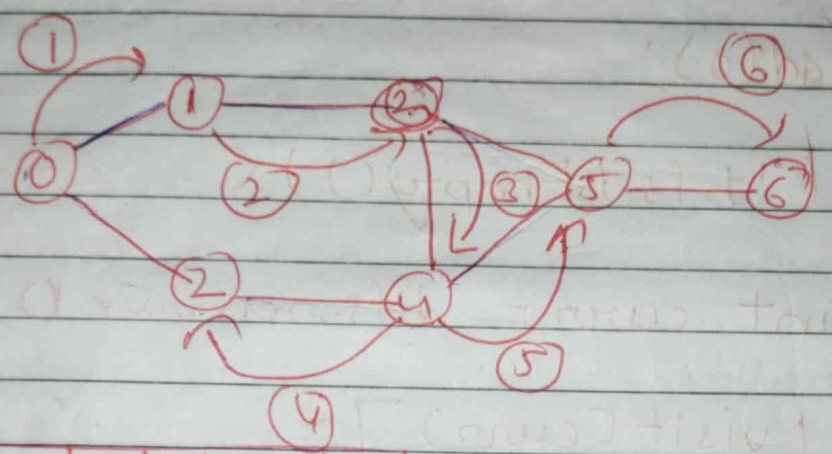
```
public static void createGraph (
    ArrayList<Info> graph[] ){
  for( int i=0; i<graph.length; i++){
        graph[i] = new ArrayList<>();
    }

  graph[0].add(new Info(0,1,0));
  graph[0].add(new Info(0,2,0));

  graph[1].add(new Info(1,0,0));
  graph[1].add(new Info(1,3,0));

  graph[2].add(new Info(2,0,0));
  graph[2].add(new Info(2,4,0));

  graph[3].add(new Info(3,1,0));
  graph[3].add(new Info(3,5,0));

  graph[4].add(new Info(4,2,0));
  graph[4].add(new Info(4,5,0));

  graph[5].add(new Info(5,3,0));
  graph[5].add(new Info(5,4,0));
  graph[5].add(new Info(5,6,0));

  graph[6].add(new Info(6,5,0));
```

Page No.
Date

```
public static void BFS(ArrayList<Edge>
                graph[]){

  Queue<Integer> bfs = new LinkedList<>();

boolean visit[] = new boolean[graph.length];

  bfs.add(0);

  while(!bfs.isEmpty()){

      int curr = bfs.remove();

  if[ !visit[curr] ]{
      system.out.print(curr+" ");
        visit[curr] = true;

for(int i=0; i<graph[curr].size() ; i++){

    Into e = graph[curr].get(i);
    bfs.add (e.dst);
  }
  }
}
}
```

DFS :- depth first search

→ In this ~~data~~ ~~stru~~ method we visit all the neibars of the current node.



d	1	3	4	2	5	6

Code

```
DFS (ArrayList<Edge> graph[],
int curr, boolean[] vist) {
  // visit the current node
System.out.print(curr + " ");
    vist[curr] = true


// visite all neighabur of it
for (int i=0; i<grap[curr].size(); i++){

  Info e = grap[curr].get(i);
  if ( ! vist[e.dst] ) {
      DFS ( graph, e.dst, vist);
  }}}
```

Connected components :-

If we want to travers all the dis-connected parts of the Graph then we have to create and write function in which we will write the complete original code and we will call it from a function in which we have a loop from 0 to graph length

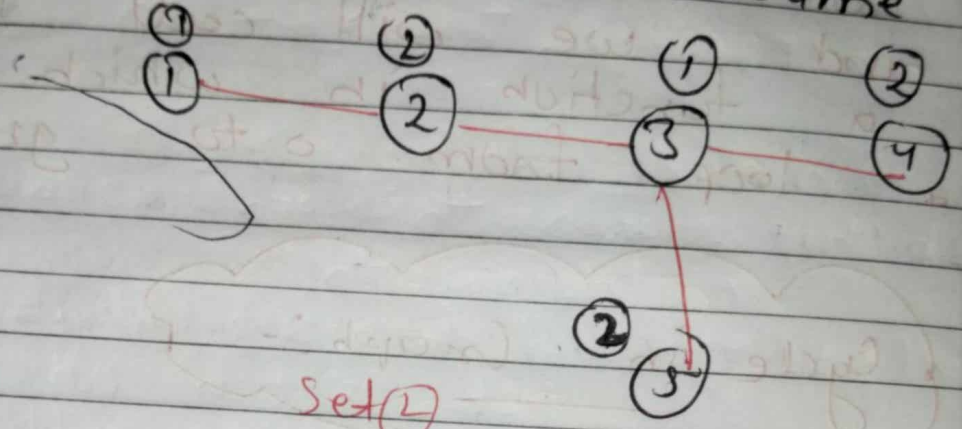
Cycle in Graph :-

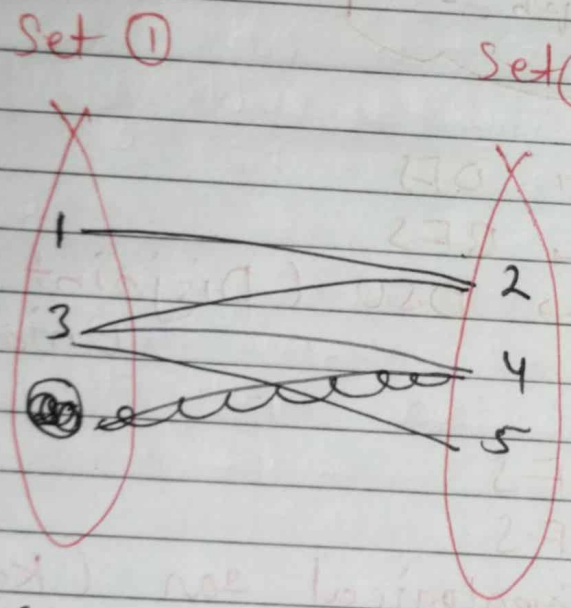
Undirected → DFS
→ BFS
→ DSU (Disjoint Set Union)

Directed → DFS
BFS
→ Topological sort (Kahn's algorithm)

Bipartite Graph

→ If an Graph can be represented as collection of vertexes in two sets such that the neighbors should not be in same set





Conditions for bipartite Graph :-

(i) If it is an Acyclic — TRUE

(ii) If it is an Even Cyclic — TRUE

(iii) If it is an odd cyclic — FALSE

Cycle Detection in Directed Graph

→ first we will create a graph and then we will create two functions to detected cycle in Graph

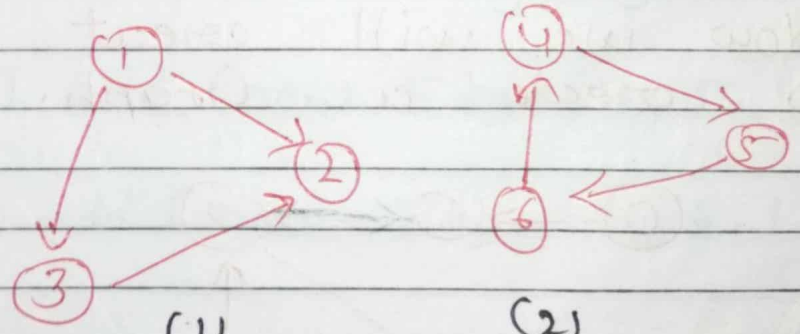
→ 1st function → this function will call our util function to detect the cycle for every unvisited node.

→ 2nd function → Second function will Detect the cycle in the Graph

Why we are using two function ?

If we will do all these things in a single function then we may left dis-connected graphs to traverse

like :-



(1) (2)

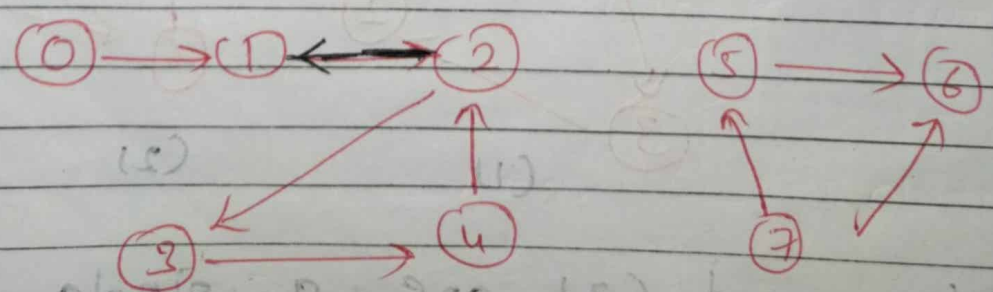
(1), and (2) are a single graph so, we have to traverse all the nodes.

Code

```
public class Graph {
    public static class Edge {
        int source;
        int dst;
        int weight;
    }

    public Info (int source, int dst,
                 int weight) {
        this.source = source;
        this.dst = dst;
        this.weight = weight;
    }
}
```

In above class we will store using all the necessary info. of a node.

Now we will creat a function to create a Graph.



```
public static void createGraph(
    ArrayList<Info> graph[7]){

for(int i=0; i<graph.length; i++){
    graph[i] = new ArrayList<>();
}

// the above loop will create an
array list at every index of
the array

    graph[0].add(new Edge(0,1,1));

    graph[1].add(new Edge(1,2,1));

    graph[2].add(new Edge(2,3,1));

    graph[3].add(new Edge(3,4,1));

    graph[4].add(new Edge(4,2,1));

    graph[5].add(new Edge(5,6,1));

    graph[7].add(new Edge(7,5,1));

    graph[7].add(new Edge(7,6,1));

}
    // Hense we have created a
Directed Graph, In this question
we can skip to the weight.
```

// Now we will create our first function which will call to our Util function for every unvisited ~~Edge~~ Node.

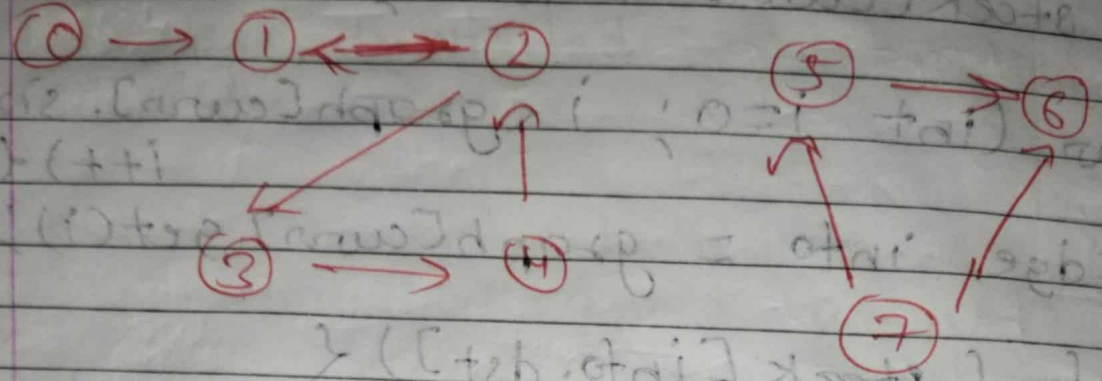
```
public static boolean isCycle(
    ArrayList<~~Info~~ Edge> graph[]){
boolean stack[] = new boolean[graph.length]
boolean visit[] = new boolean[graph.length]
for(int i=0; i<graph.length; i++){
    if(!visit[i]){
        if(isCycleUtil(graph, stack,
                        visit, i)){
            return true;
        }
    }
}
return false;
}
```

// Now we will create our util function.

```
public static boolean isCycleUtil(
ArrayList<Edge> graph[], boolean
    stack[], boolean visit[], int cur){
```

```
visit[curr] = true;
stack[curr] = true;

for (int i=0; i<graph[curr].size();
                                i++){
Edge info = graph[curr].get(i);

if (stack[info.dst]){
        return true;
}

else if( !visit[info.dst] && isCycle(Oth)(
    graph, stack, visit, info.dst)){

    return true;
}
}  stack[curr] = false;
    return false;
}

public static void main(String[] args){

ArrayList<Edge> grap[] = new ArrayList<
                                    [7];
createGraph(grap);

boolean ans = isCycle(grap);

System.out.print(ans);
}}
```

Explanation :-



Stack =

0	1	2	3	4	5	6	7

visit =

0	1	2	3	4	5	6	7

→ isCycle function will call for 0 to the Util function.

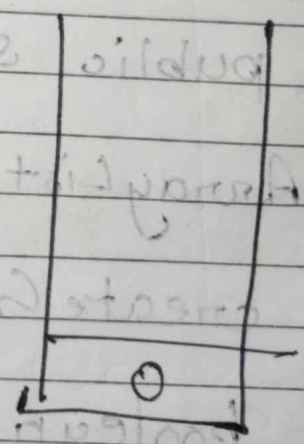
→ Now in Util function in stack and in visit we make 0 index as true.

Stack

0	1	2
T		

visit

0	1	2
T		

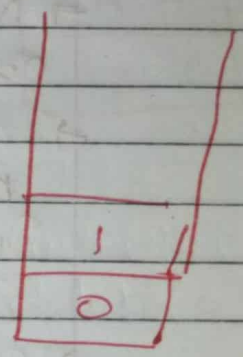


→ Now, inner for loop will find the neighbour of 0, which is

→ It will check if y is in stack or not, it will return false

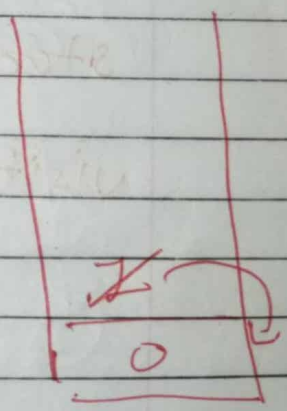
→ Now, 1, is not visited than way util function will be called for 1, Now it will make 1st indx of stack and visit as true

	0	1	2
Stack	T	T	F
visit	T	T	F



→ Now in for loop we will try to find neighbour of 1 but we don't have so

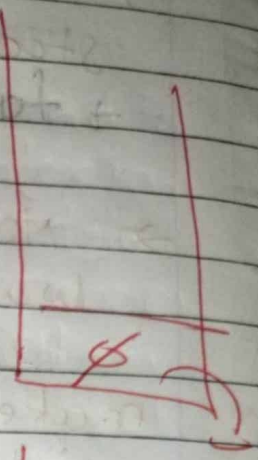
	0	1
stack	T	F
visit	T	T



→ Now recursion stack will calle for 0 again but we have traversed all the neighbours of 0, so it will also be removed from stack.

	0	1
Stack	F	F

	0	1	2
visite	T	T	



→ Now util function will have call. for 2,

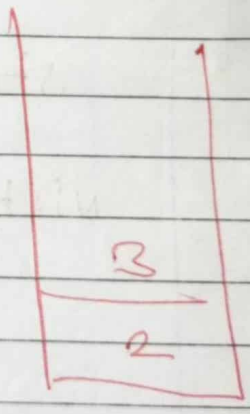
→ It will check for the neighbours of 2 and we will have 3

→ 3, is not in stack

→ 3, is un-visited so we will again call util funct for 3,

	0	1	2	3
stack	F	F	T	T

	0	1	2	3
visit	T	T	T	T



→ Now utile find heighbour of 3, that is 4

→ 4 is not in stack

→ 4, is un-visited

→ So util function will be

→ this will be done before for loop

called for 4

→ Stack :

0	1	2	3	4
F	F	T	T	T

→ visit :

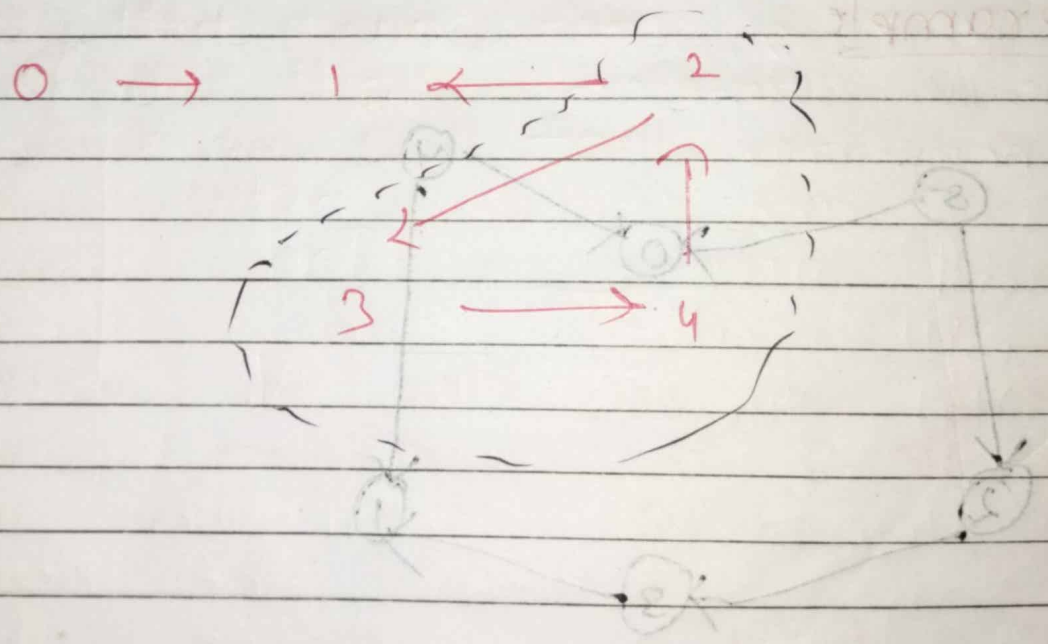
0	1	2	3	4
T	T	T	T	T

Now, ~~until~~ in for loop we will find the neighbour of 4, that is 2,

4
3
2

→ 2, is already in stack so, it will return true

hence we have ~~deted~~ detected a cycle in a graph



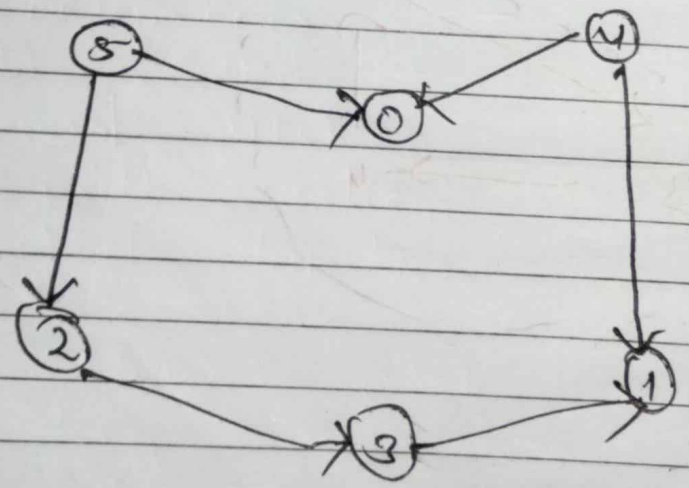
Topological sorting

→ In this sorting we find the linear order of vertexes such that every direct edge $u \to v$ the vertex u comes ~~after~~ befor v in the Order

→ This sorting is used for any Directed Acyclic Graph (DAG)

DAG ?
The graphs which have directed edges and have no cycles.

example



→ 5, 4, 2, 3, 1, 0

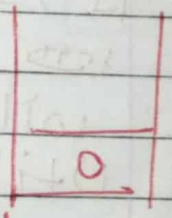
How will the aproch work.

→ To complete this task we can apply DFS ont the Graph and can store the visited vertices in the stack

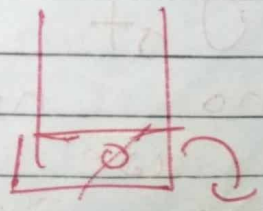
→ And hense we will ~~av~~ have the last visited ~~not~~ vertex at top and first visited vertex at bottom, that ~~way~~ why we will ~~push~~ pop() all the element from the stack and we will ~~gate~~ a correct order.

Dry Run:-

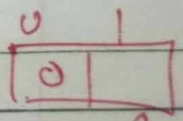
→ first we will call for 0 to our Util function



→ Now we will see for the neighbour of 0 th and we will get nothing, so we will return



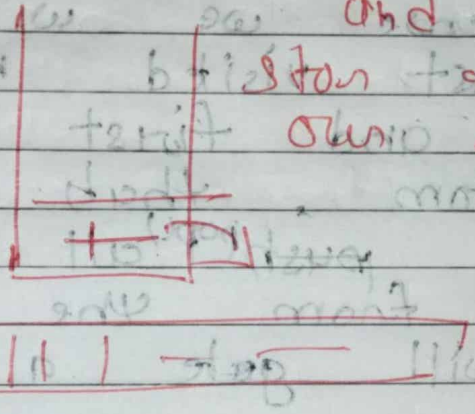
and add 0 to stack



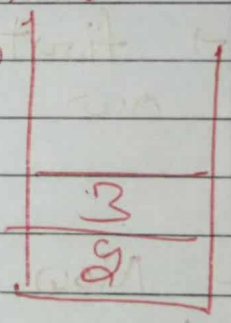
Now our function will call for

to these util function

→ and util function we try to find neighbour of 1 and we have no neighbour that's why we will return from here in call stack and we will store this 1 in our stack



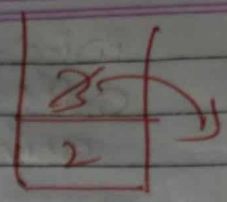
→ Now our function will call for 2, to our util function util function will find the neighbors of 2, which is 3 and it is not visited so we will called the util function for 3.



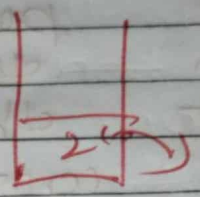
→ Now util function will try to find the neighbour of 3 and we have no neighbour of 3, so we will return from here in call stack and add 3 to stack.

Stack

0	1	3



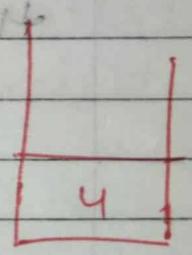
→ Now 2 have visited all its neighbour so again return from here in call stack



add 2 to stack

0	1	3	2

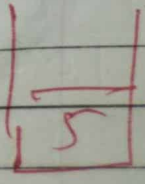
→ Now function will call for 4, to util function



→ 4, have two neighbour 0 and 1 and they are visited before so return from call stack and add 4 to stack

0	1	3	2	4

→ Now function will calle for 5 to util function and 5 have two neighbour 0, and 2 ~~the~~ and they both have visited so



we will return from here in call stack and add 5 to our stack

Stack

0	1	3	2	4	5

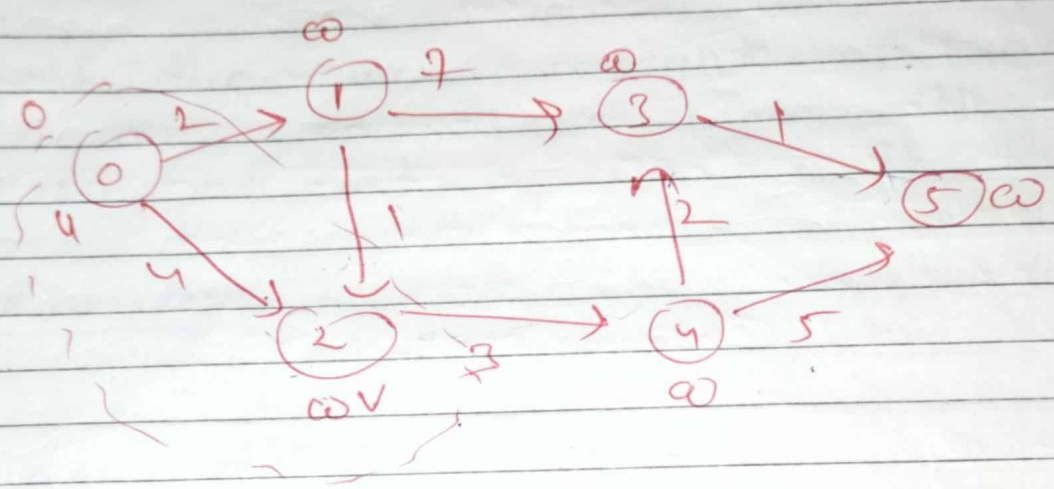
→ Now pop all the elements from this stack and print then it will be the correct path

5 4 2 3 1 0

Note:- have a boolean type array and marke T to the index which vertex is visited.

Dijkstra Algorithm :-

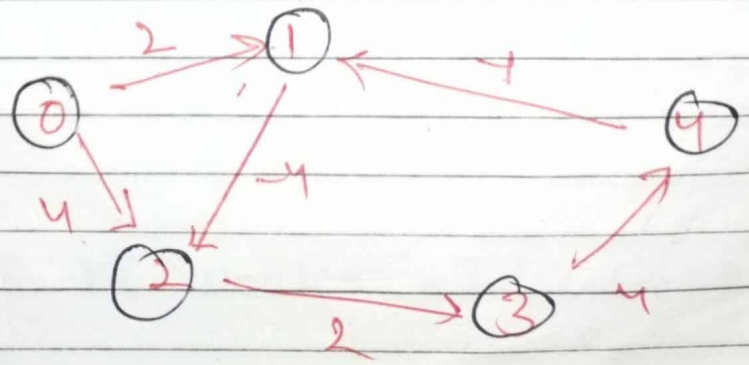
→ If we want to find the sortest path from a source to destination then we can use of Dijkstra Algorithm.



if dist[u] + weight < dist[v]
then dist[v] = dist[u] + weight ;

Bellman ford Algorithm

→ Shortest paths from the source to all vertices (negative edges)



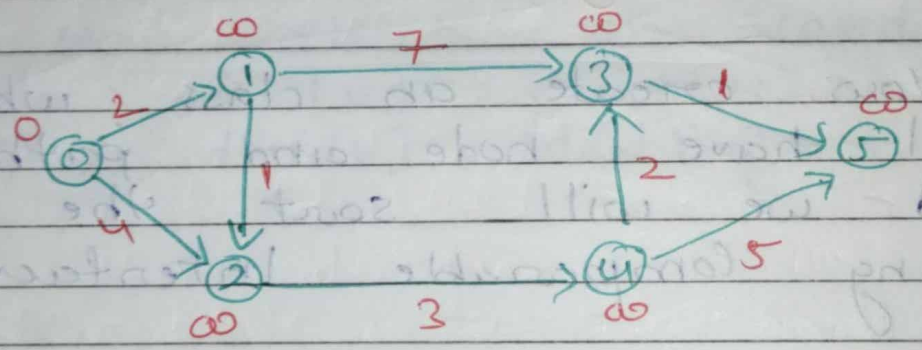
Note :- This algo does not work for Negative weight Cycles.

Dijkstra Algo.

* To find the sortest path from source to destination we use of dijkstra Algarithm

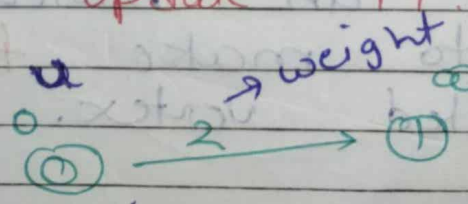
* If the given weight of vertexes are -ve then we will not use of it.

<u>example:-</u>



In this algo we do an simple opration if the cost to go destination from source is less then destination's vertex's cost then we will update it.

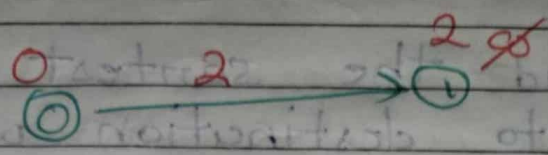
let's



vertex[u] + weight < vertex[v]

$0 + 2 < \infty$

So, we will update destination cost



what will we do ?

* We will be having and array of to collect distance of all vertex, initialize the source distance as zero, and others as ∞ (Integer.MAX_VALUE).

* Now create an class which will have node and path and we will sort the path using Comparable Interface

* Now create an priority queue and insert the first object with node source and path as zero,

* Create an boolean type visited array to make true for visited vertex.

* Now run a while loop while pq is not empty.

* remove current object from PQ and check if it's node is visited or not.
If not then get its all Edges (neighbours) by an for loop,

* get 3 information by neighbour Edge type object
1 → vertex u (source)
2 → vertex v (destination)
3 → weight

* Now check an condition that
if dist[v] > (dist[u] + weight)
then dist[v] = (dist[u] + weight)
distence

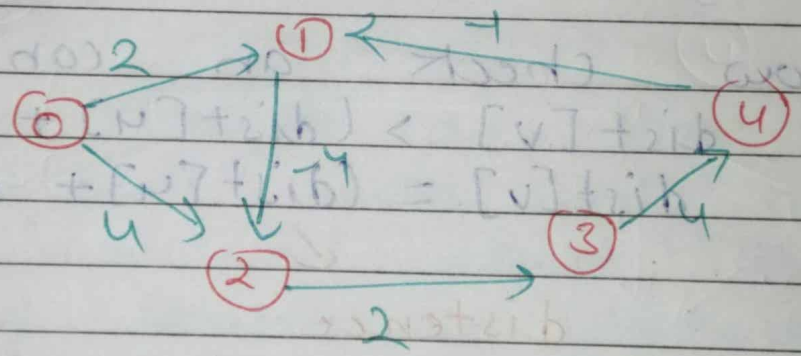
If this condition is true then insert one more Info type object in PQ with neighbour vertex and it's distence

Bellman ford Algo

→ This algo is used to find the shortest path from the source to all vertices

→ Negative edges are allowed

<u>example</u>



⇒ In this algo we will run a loop.
if we have V vertices then loop will run for V-1 times

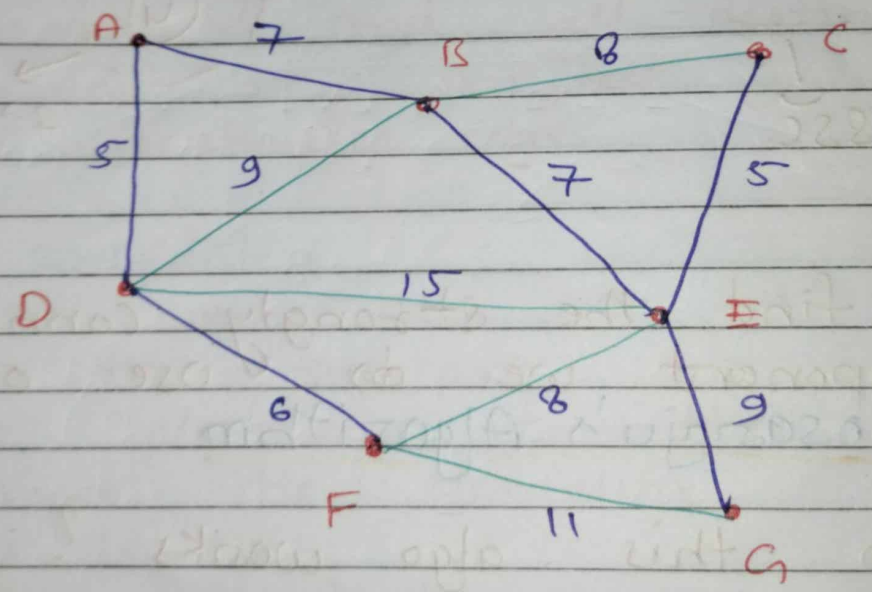
→ all V-1 time we will find the sortest path for all vertexes

Prim's Algorithm

* Used to find minimum spanning tree

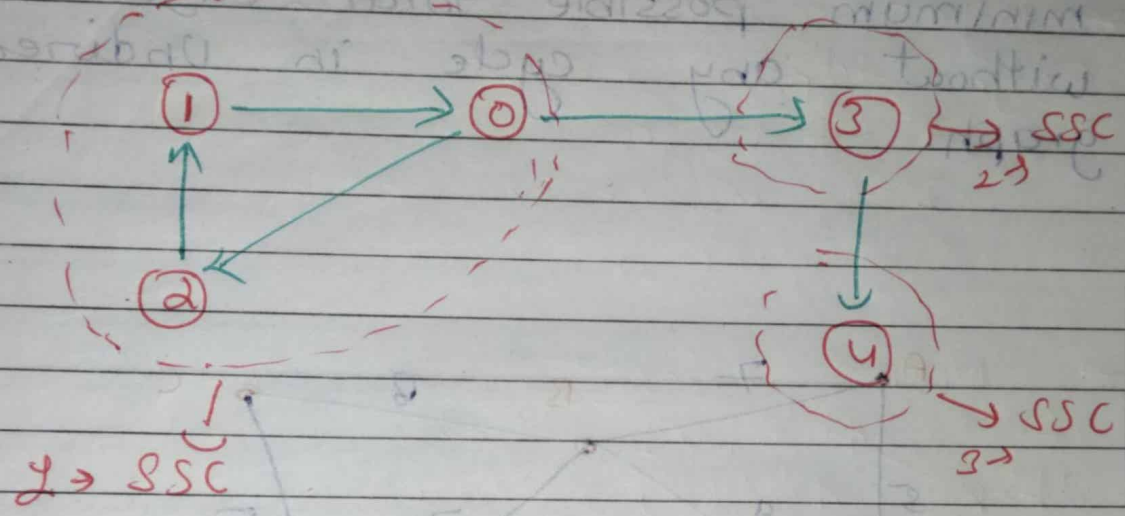
MST ?

A sub set of a Graph having minimum possible total edge weight without any cycle in Undirected graph.



Strongly Connected Component

→ SSC is a component in which we can reach every vertex of the component from every other vertex in that component.



To find the strongly connected Component we do use of Kosaraju's Algorithm

How this algo works ?

In this algo we will do three simple steps

1 step

first we will get the order of vertex's in which we should

performing traversal.

To find this order we will do Topological sort and we will get elements in the stack

2nd step.

Now we will find the transpose of the oringhal graph.

→ To find transpose we will just creat an temp graph and will copy all the sources as destination and all destinations as sources of oringhal graph of an perticular edge.

3rd step

Now we will Do the DFS on the transpose and we use the elements of the stack to traverse the elements

→ we will just pop() an element and then call for the dfs function for it, if it is not visited.