1 June 2021

SEGMENT TREES :-

You are given a array of integers. You need to perform the following two operations

- Update (i, x) $\Rightarrow$ Set $A[i] = x$
- RMQ (l, r) $\Rightarrow$ return the index of min of the nos. $A[l], A[l+1], \ldots A[r]$

Ex

0	1	2	3	4	5	6	7
24	3	45	12	98	36	1	87

RMQ $(2,5) = 3$
RMQ $(1,4) = 1$
RMQ $(3,7) = 6$

BRUTE FORCE :-

Update will be done in $O(1)$
RMQ operation in the worst case can take $O(n)$

So if we do n operations, in worst case, the total complexity can be $O(n^2)$.

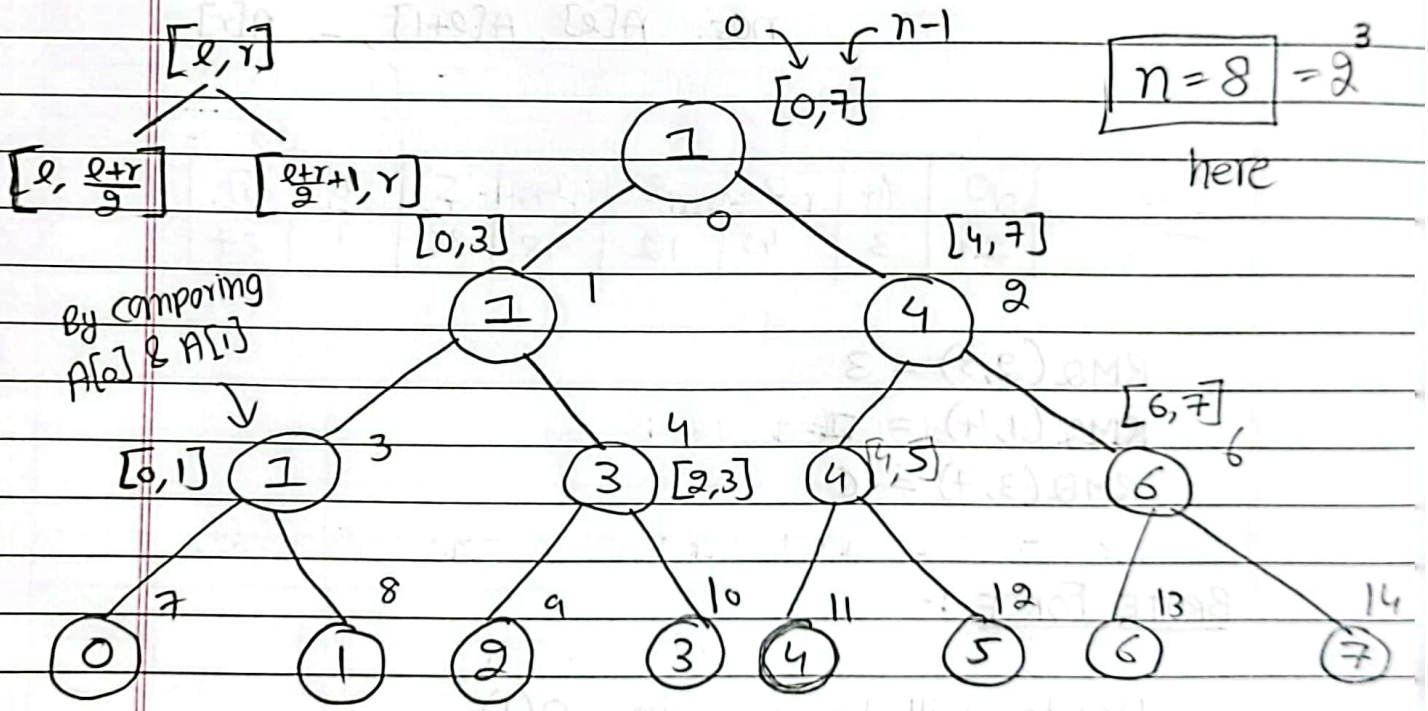
SEGMENT TREE METHOD :-

Both Update & RMQ shall be done in $O(\log n)$

So, if we do n operations, the overall complexity $O(n \log n)$.

$A =$

0	1	2	3	4	5	6	7
24	3	45	12	7	36	73	87



⇒ If we are given length of the array is some 2^k, then fine

Otherwise, we extend the array so that its length n is equal to some nearest 2^k and fill the additional entries with ∞.

Also Note → The no. of dummy entries that will be created is always gonna be less than the no. of original elements.

∵ B/w n & $2n$, there is always a no. which is 2^k.

⇒ So, now once n is fixed (either by extension or not) the no. of nodes that segment tree gonna have is $\boxed{2n-1}$

⇒ Segment Tree will be having an array representation is/in level order traversal just as we represented Binary heap.

Update (i, x) Operation :-

→ go to $(n-1+i)^{th}$ index in the ST array.

→ Update $A[ST[n-1+i]] = x$

→ Now compare the updated values and update the parent node if req. . keep moving up untill u reach the root node.

ie. parent node will be at location

$$\text{floor}\left(\frac{n-1+i-1}{2}\right) \Rightarrow \boxed{\text{floor}\left(\frac{\text{index}-1}{2}\right)}$$

index → index of node in ST whose parent is to be found

After going to porent node, compore its child nodes and keep moving up.

left child → 2^* index + 1
right child → 2^* index + 2

indx → index in ST whose children to be found

Hence complexity = $\boxed{O(\log n)}$

- RMQ (l, r) → Range Min. Query

We start with the node (root) and have 3 cases.

(i) If segment completely overlops the range

return ST[i] (the volve in that node)

(ii) If segment is completely out of range

return ∞

(iii) If segment partially overlaps

Then we proceed recursively to left and right

Complexity $\boxed{O(\log n)}$

- **RSQ(l,r)** → Range Sum Query.

⇒ Return the sum of all the numbers A[l], A[l+1] A[l+2], ---, A[r]

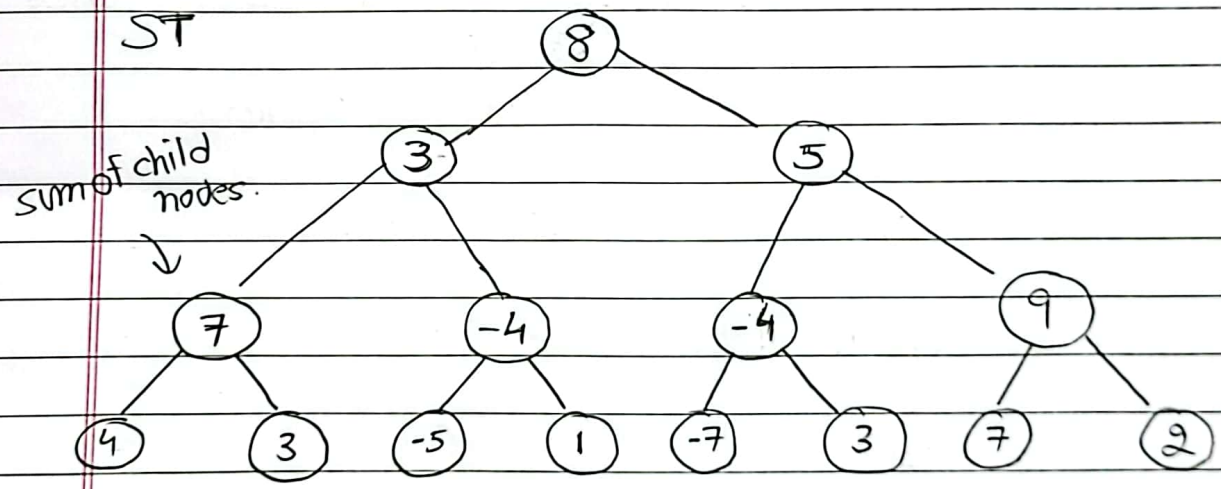
We define Presum(i) = A[0] + A[1] + -- A[i].

So $\boxed{RSQ(l,r) = Presum(r) - Presum(l-1)}$

A =

0	1	2	3	4	5	6	7
4	3	-5	1	-7	3	7	2

ST



To calculate Presum(i) → We start with the root node and go to ~~root~~ leaf node at index (n-1+i) i.e. where A[i] is present.

At any node → When we go left ⇒ We ignore right part completely.

When we go right ⇒ We add the value present in left child.

So, to calculate RSQ, we call Presum 2 times.

BIT

5 June, 2021

1 Binary Search Trees

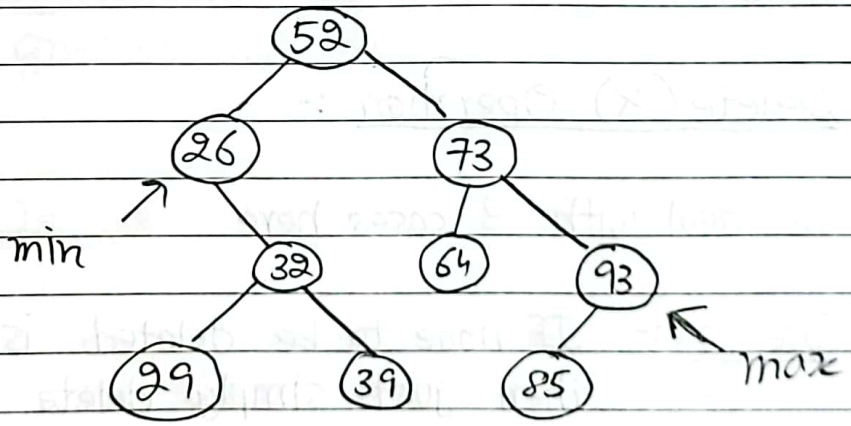
B.S.T. is a binary tree in which each node is associated with the key, at each node, following properties are satisfied :-

① The keys of all the nodes in the left subtree are less than the value of the key of present node.

② The keys of all the nodes in the right subtree are all greater than the value of key of present node.

They above properties we call **Binary search prop.**

Ex



- **To Find Max Element** → keep going right until that you can't go furthur. That node will contain max.

To Find Min Element → Keep going left from the root node until you can't go furthur. That node will contain min.

- Search (x) Operation :-

If the value of key at a node < X, search in the right subtree else search in the left subtree.

If u reach a null pointer, then X is not present in the tree.

- Insert (x) Operation :-

Search (x) till you reach NULL and insert it there.

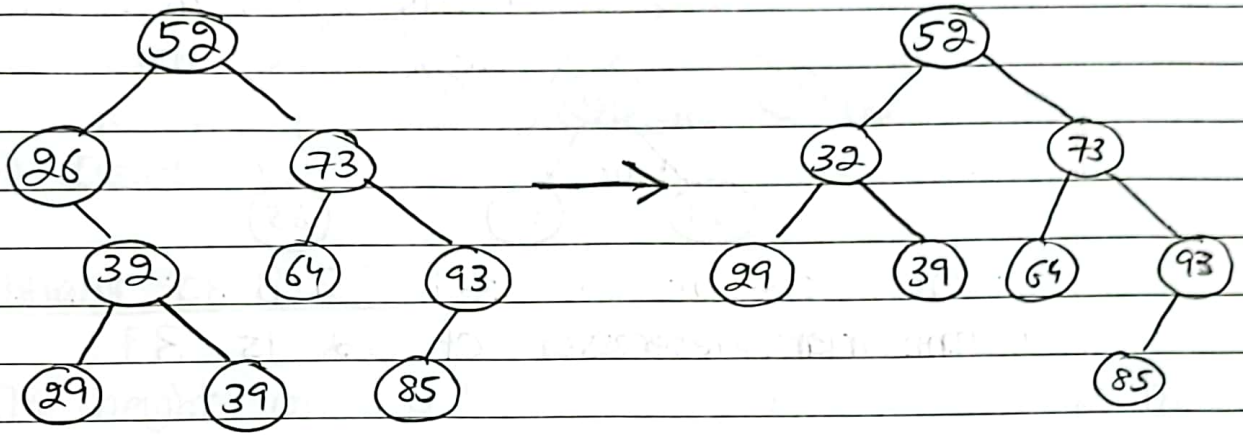
- Delete (x) Operation :-

We deal with 3 cases here

Case-0 :- If node to be deleted is a leaf node, then just simply delete it.

Case-I :- If node to be deleted has only one child, then replace it with that child.

Ex Delete 26.

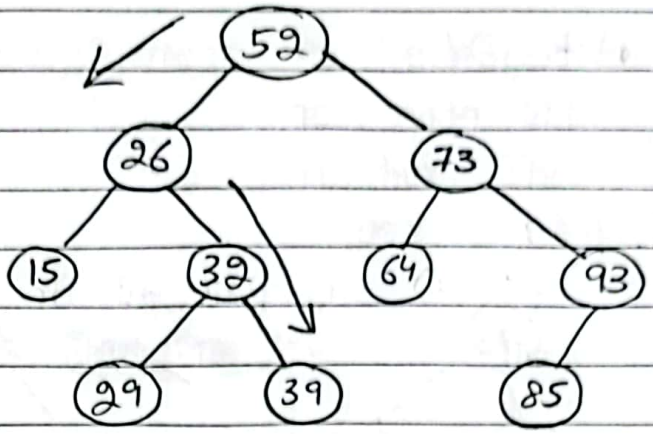


Case-II :- If node to be deleted has two children, replace the value of by its in-order predecessor's value, then delete the in-order predecessor.

In-Order predecessor of any key X is defined as the greatest no. smaller than X present in B.S.T.

To Find In-order predecessor :- Suppose we are at the node where X is present, then first move left (one step) and then keep moving right until you can't go further. The key in that node will be the in-order predecessor of X.

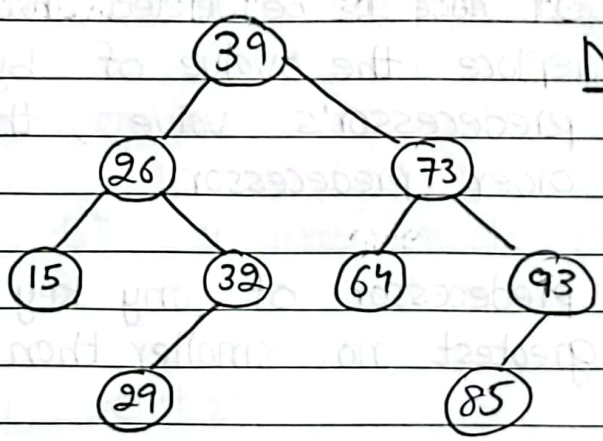
Ex



In-order predecessor of 52 is 39

So, if we want to delete 52

⇒



NOTE :- While deleting in-order predecessor, we will use either case 0 or 1

Similar is the in-order successor.

In-order Successor → of X is defined as the no. smallest of all the elements which are greater than X in B.S.T.

First go right and then keep going left

Complexity :-

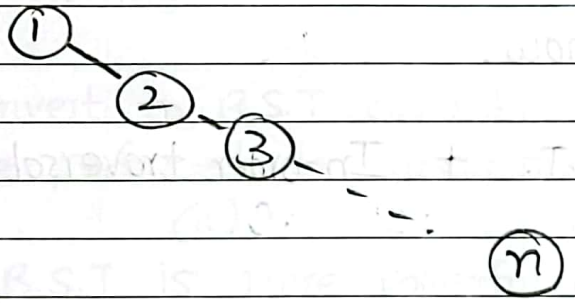
① Search
② Insert
④ Delete

$\Rightarrow O(\text{Height of the B.S.T}) = O(n)$

- **HEIGHT OF B.S.T :-**

The height of B.S.T can be as bad as linear in terms of no. of nodes in B.S.T.

Ex :- Insert all the no. in ascending order.



Degenerate tree

- **BALANCED BINARY SEARCH TREE :-** BBST

It is a B.S.T whose height is $O(\log n)$

Ex AVL Tree.

2 NOTE :- In-Order Traversal of a B.S.T gives us the sorted sequence of trees.

So, given a B.S.T, we can get the sorted sequence of the keys in $O(n)$.

Q Given a sequence of keys, can we build a B.S.T in linear time?

Ans is **NO**

Let us suppose that its possible.

Then we know

$$\underset{O(n)}{\text{Build B.S.T.}} + \underset{O(n)}{\text{In-order traversal}} = \underset{O(n)}{\text{sorting}} \quad \text{--- (1)}$$

But sorting is not possible in $O(n)$. Hence Build B.S.T cannot be $O(n)$.

So from eq. (1), we conclude that any algorithm to build B.S.T will take $\Omega(n \log n)$ time.

⇒ We build B.S.T. by calling Insert (x) function n times.

∴ Complexity = $\Omega(n \log n)$

- **B.S.T. vs. Binary Heap :-**

Building a Binary Heap by calling Insert/Add n times took $\Omega(n \log n)$ time, but we could build binary heap in linear time which is not possible in case of B.S.T.

Q Given a Binary heap, can we build the B.S.T in linear time ?

NO

⇒ Build Binary heap + convert to B.S.T + In-order = sorting.

if convert to B.S.T could be linear, then sorting can be performed in linear time which is not possible

Hence, B.S.T is more powerful data structure than heap.

- **B.S.T. vs. HASHING :-**

Search, Insert and Delete

Hashing :- $O(1)$ in avg. case.

B.B.S.T :- $O(\log n)$ in worst case

Some operations like Rank(x), can be done in $O(\log n)$ using BBST, but hashing takes linear time.

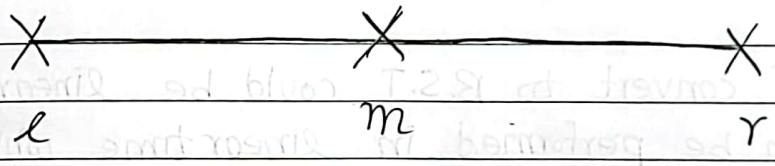
Conclusion :- So, if our dynamic data set involves only searching, deleting and inserting, implementing hashing is better option.

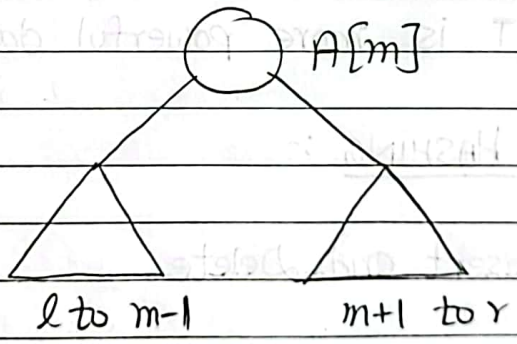
3 Codes of diff. functions in lms.

4 BUILD A B.S.T :-

given a sorted sequence of keys, build a B.S.T.

Particularly, we will build a B.B.S.T.





See function Create BBST in l.m.s.

Complexity :-

$$T(n) = O(1) + T(n/2) + T(n/2)$$

$$= O(1) + 2T(n/2)$$

$$= \boxed{O(n)}$$ Rem :- Seq. given is sorted

Height :- $H(n) = 1 + Max\{H(n/2), H(n/2)\}$

$$= 1 + H(n/2)$$

$$= \boxed{O(\log n)}$$

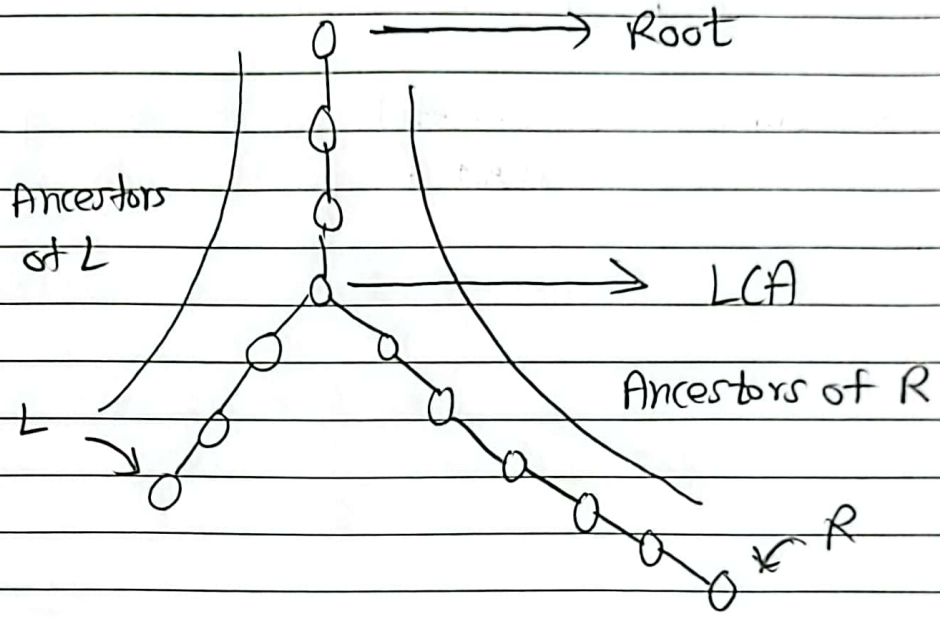
⇒ Given a sorted sequence of keys, a BBST can be build in linear time.

Given a sequence of keys, a BBST can be built in $O(n \log n)$

⇒ **Predecessor & Sucessor codes in l.m.s.**

- **LEAST COMMON ANCESTOR (LCA) :-**

LCA (L,R) is the node of min. height which is ancestor to both L & R.



See code to find LCA in l.m.s.

- **RANGE LIST :-**

Rangelist (l,r) is the list of all the nodes which are between l and r.

- **RANGE COUNT :-**

Range count (l,r) count the no. of nos. which are between l & r.

Summary :-

Build B.S.T : $O(n \log n)$
Predecessor & Successor : $O(H)$
LCA : $O(H)$

RangeList : $O(H+K)$
Rangecount : $O(H+K)$

P.T.O.

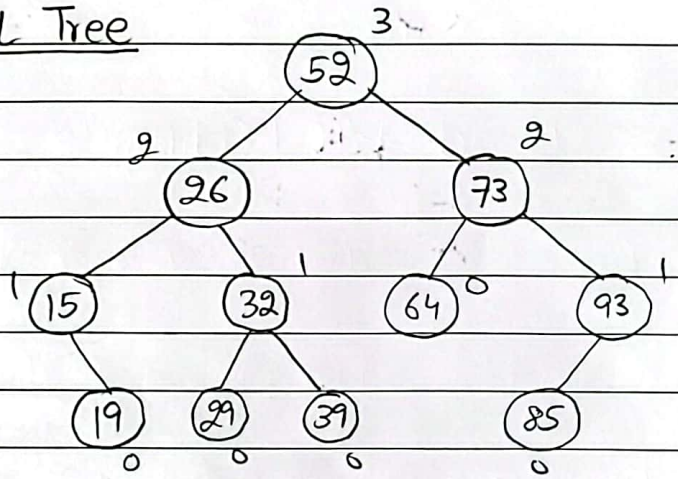
10th June, 2021

1 AVL TREE :-

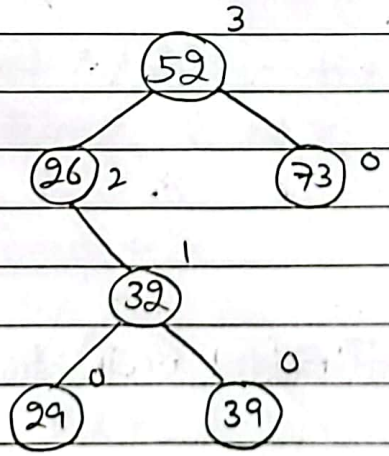
It is a B.S.T with the following property

For every node in the tree, the heights of the children, can differ by at most 1. of any node.

An AVL Tree



Not AVL Tree



AVL property is voilated at node 52 and 26

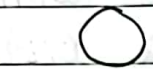
NOTE :- If any node has only 1 child, then the height of other child can be considered as -1 for refrences.

See case of 26 here.

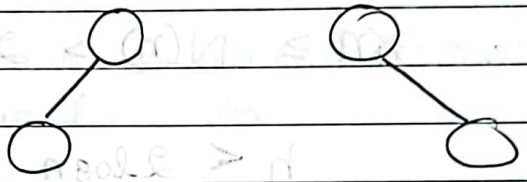
HEIGHT OF AVL TREE :-

Let N(h) be the min. no. of nodes required in an AVL tree of height h.

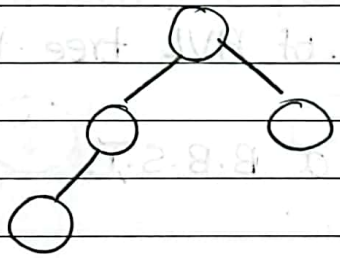
N(0) = 1



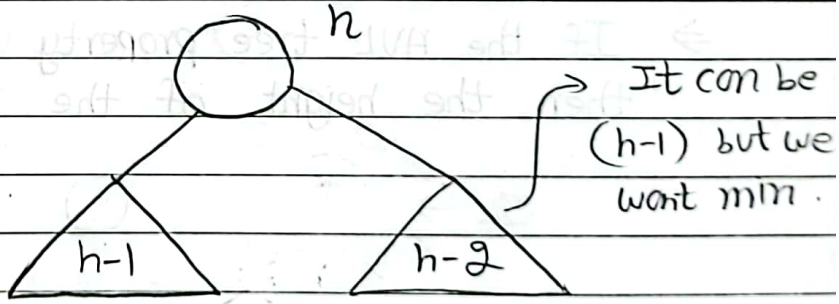
N(1) = 2



N(2) = 4



Generalising :-



$$N(h) = N(h-1) + N(h-2) + 1$$

$$> N(h-1) + N(h-2)$$

$$> N(h-2) + N(h-2)$$

$$> 2N(h-2)$$

Observation ⇒ $N(h) > 2^{(h/2)}$

So $N(h) > 2^{(h/2)}$

Consider any AVL tree with height h and n nodes. Then

$$n \geq N(h) > 2^{(h/2)}$$

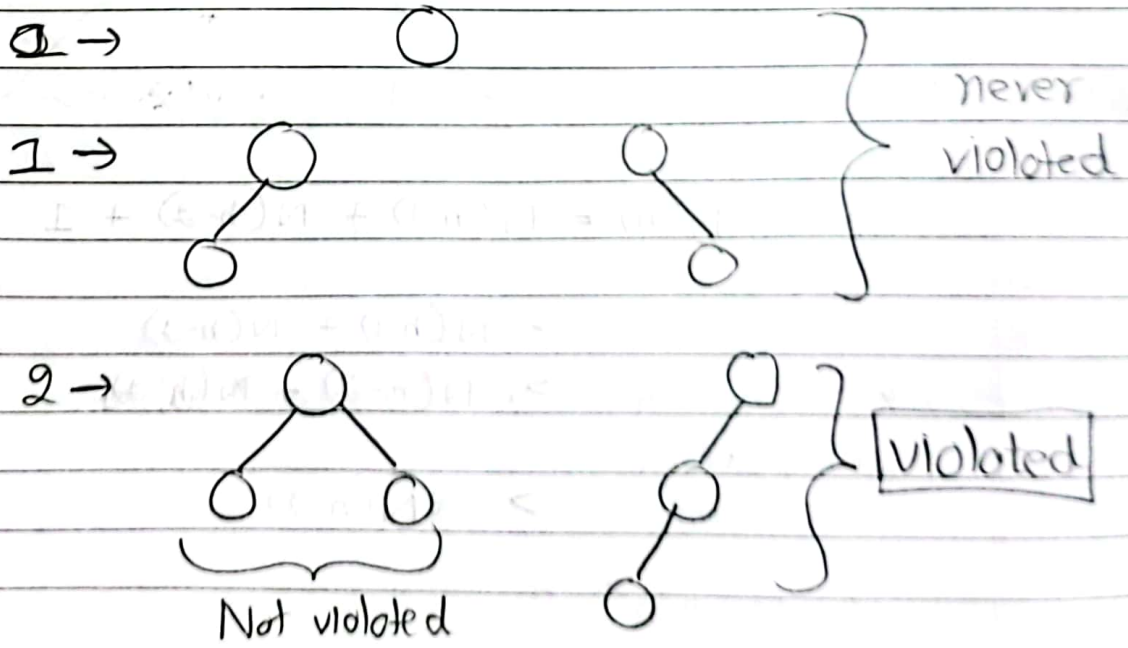
$$h < 2\log n$$

∴ Height of AVL tree is $\Theta(\log n)$

⇒ AVL tree is a B.B.S.T.

Imp. Note :-

⇒ If the AVL tree property is violated at a node, then the height of the node must be atleast 2.



2 ROTATIONS :-

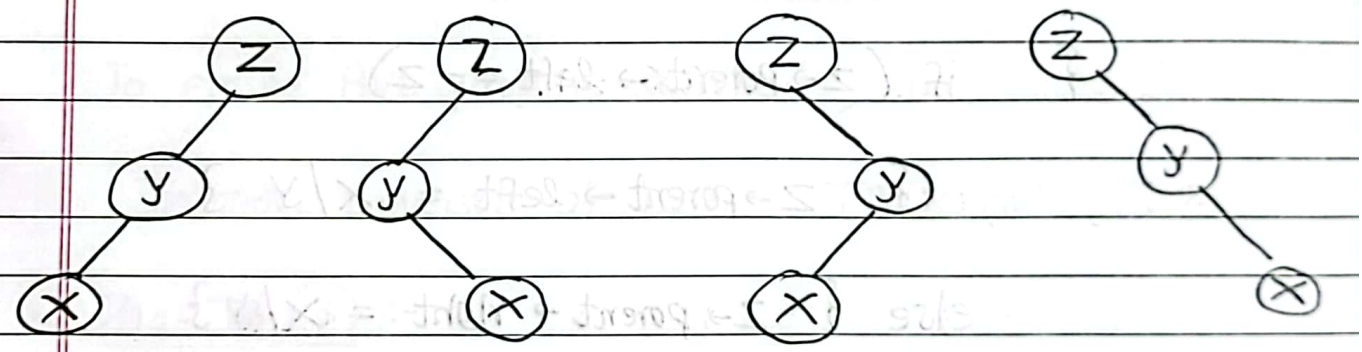
Z be a node where the AVL property is violated

Y be the child of Z such that $H(z) = H(y) + 1$

X be the child of Y such that $H(y) = H(x) + 1$

given that $H(z) > 1$, hence X and Y are valid nodes in the AVL tree.

So, now, there arises 4 possibilities.



0 = 00 — ↑↑ — Both left

1 = 01

2 = 10 — ↑ then ↖ — First right

3 = 11 — ↑↑ — Both right

Tree no. 0 & 3 → Zig Zig

1 & 2 → Zig Zag.

Note :- To differentiate among 4 of them, we when starting from Z, when we go right, append 1 and when we go left, append 0. Then we can convert this rep. into decimal.

i.e. we keep a count variable initialised to zero

When go Right → count = 2*count + 1
When go left → count = 2* count

⇒ Z be a node where AVL prop is violated.

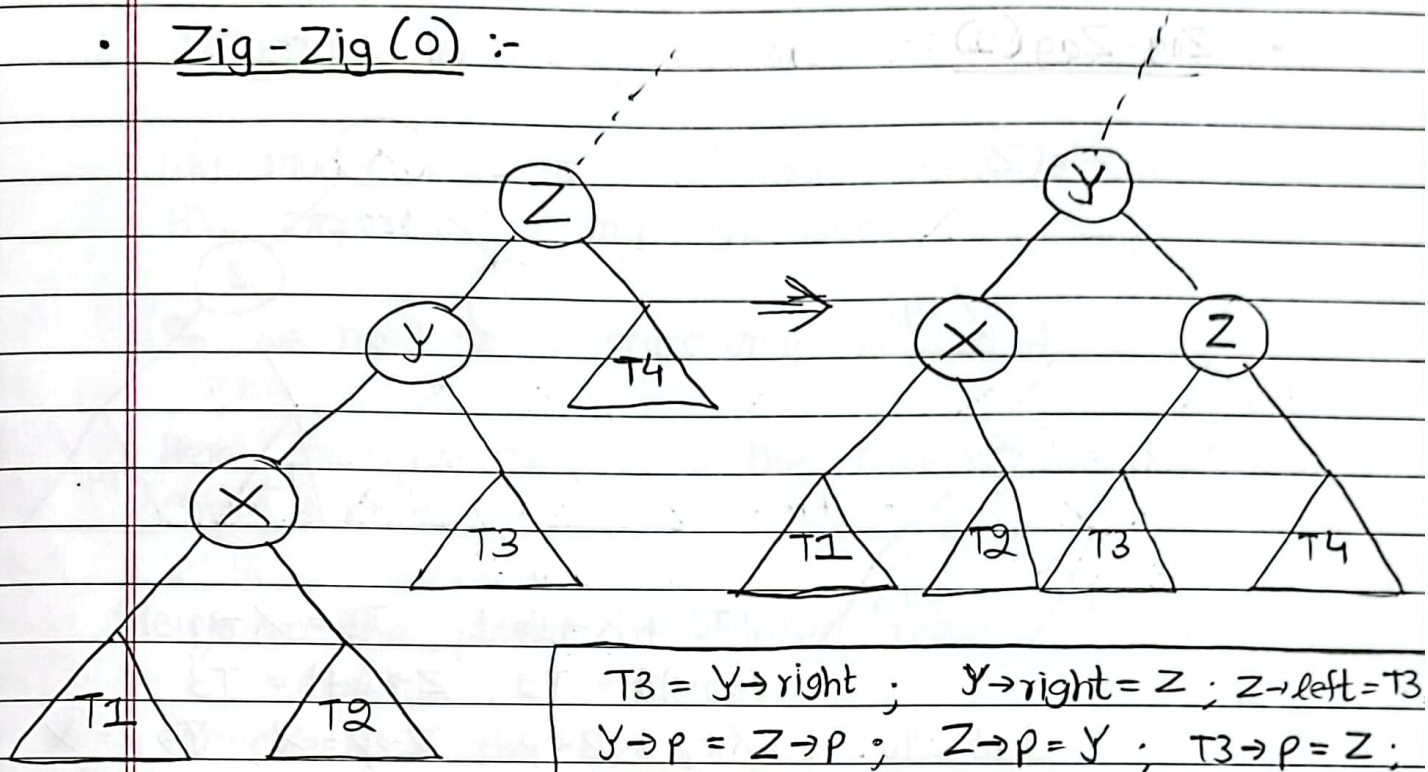
Replace the subtree tree rooted at Z with the subtree rooted at X or Y.

```
if (z→ parent)
{   if (z→ parent → left == z)
        { z→parent→ left = X/Y }
    else { z→ parent→ right = X/Y }
}
else X/Y become root of AVL tree
```

P.T.O.

Now, we look at diff. cases :-

- **Zig - Zig (0) :-**

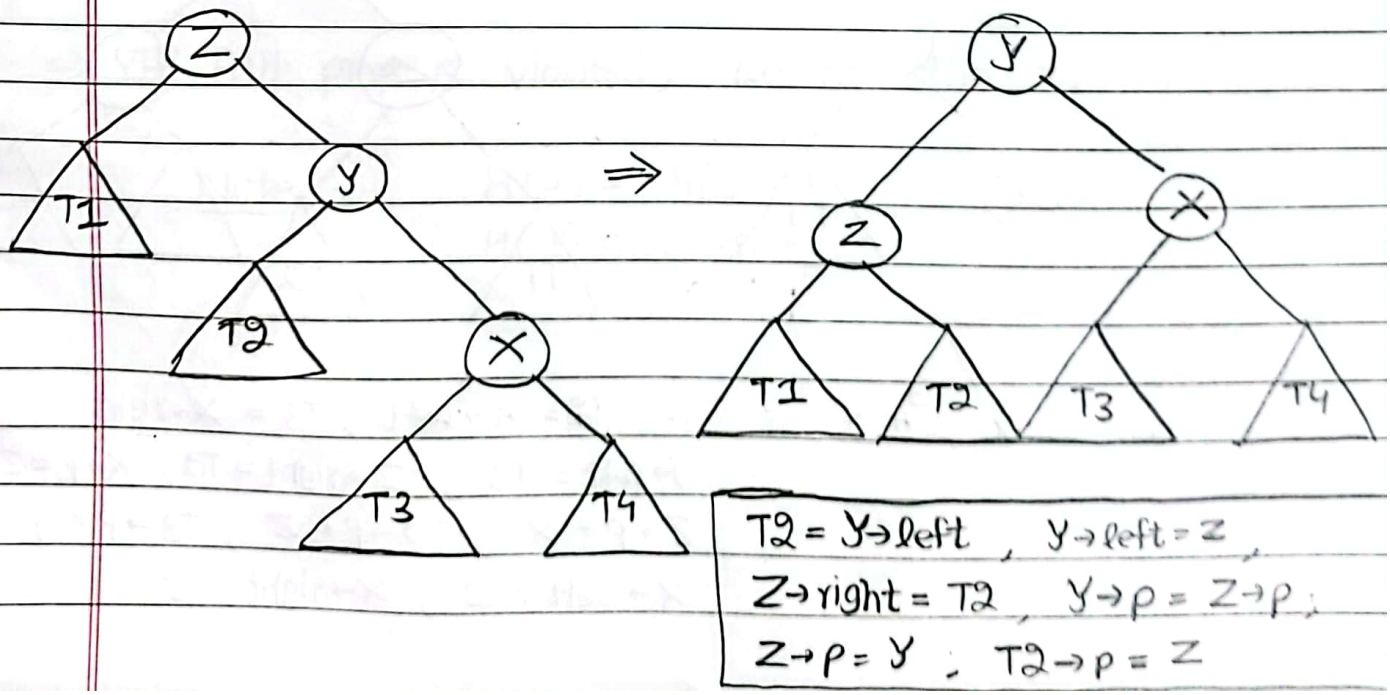


$T3 = Y \rightarrow right$; $Y \rightarrow right = Z$; $Z \rightarrow left = T3$
$Y \rightarrow p = Z \rightarrow p$; $Z \rightarrow p = Y$; $T3 \rightarrow p = Z$;

To ensure AVL prop is satisfied, we look at

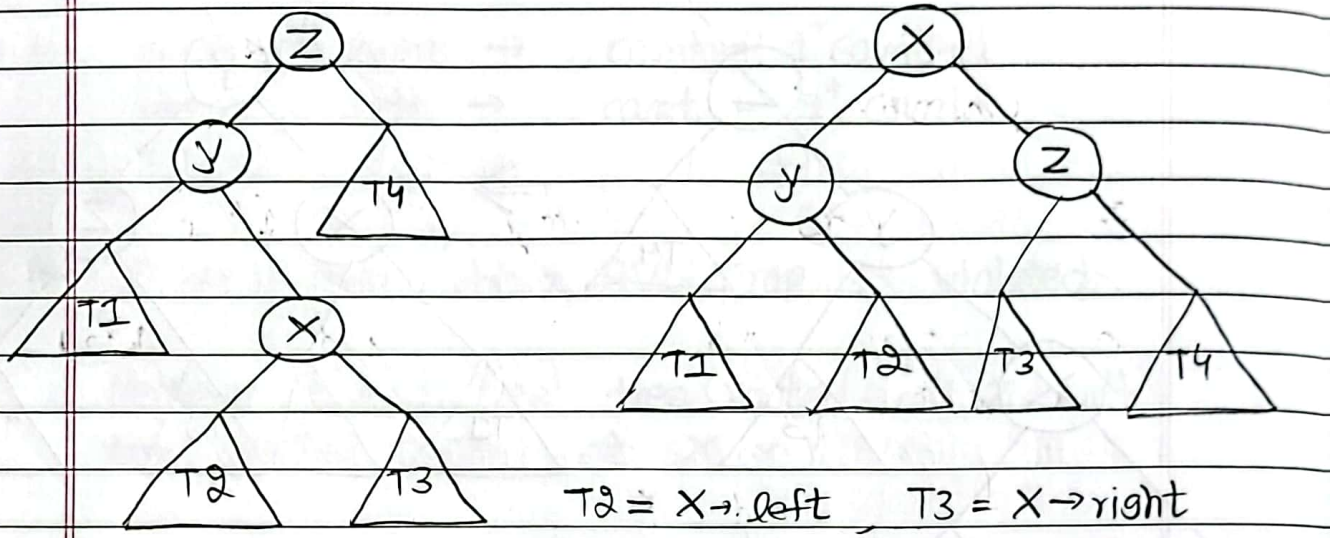
In order traversal of both = T1 X T2 Y T3 Z T4

- **Zig - Zig (3) :-**



$T2 = Y \rightarrow left$, $Y \rightarrow left = Z$,
$Z \rightarrow right = T2$, $Y \rightarrow p = Z \rightarrow p$;
$Z \rightarrow p = Y$, $T2 \rightarrow p = Z$

- **Zig-Zag (1) :-**



$T2 = X \rightarrow left$, $T3 = X \rightarrow right$

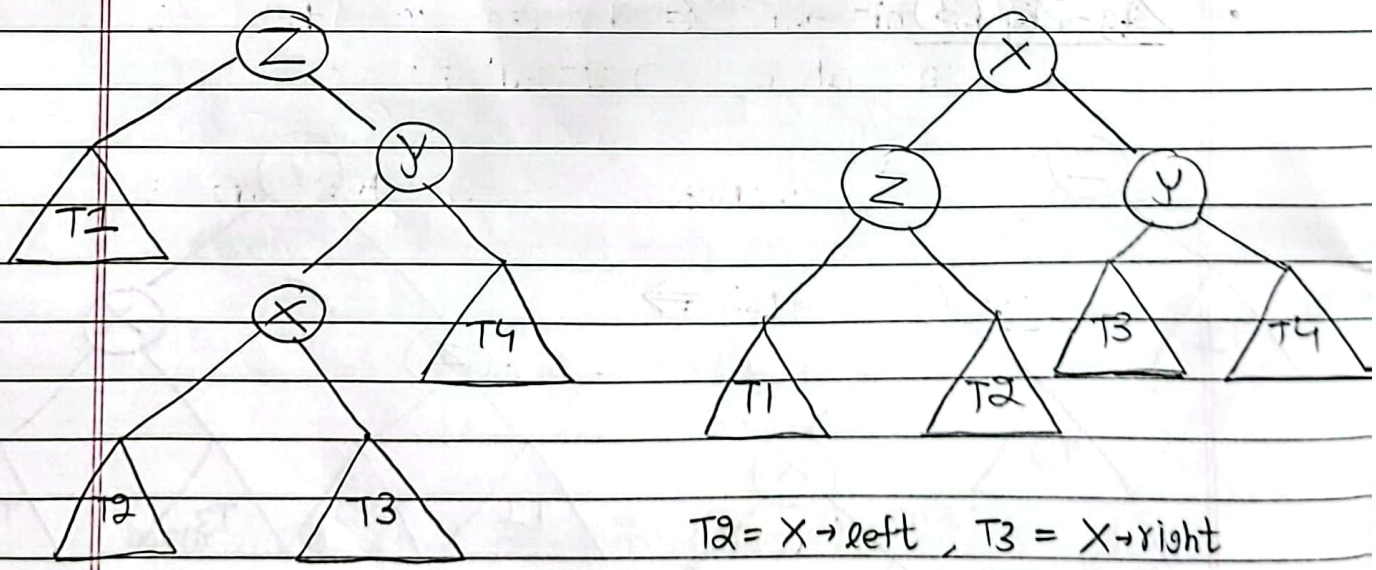
$y \rightarrow right = T2$, $Z \rightarrow left = T3$;

$X \rightarrow p = Z \rightarrow p$; $Z \rightarrow p = X$; $y \rightarrow p = X$

$T2 \rightarrow p = y$, $T3 \rightarrow p = Z$, $X \rightarrow left = y$

$X \rightarrow right = Z$

- **Zig Zag (2) :-**



$T2 = X \rightarrow left$, $T3 = X \rightarrow right$

$y \rightarrow left = T3$, $Z \rightarrow right \Rightarrow T2$, $X \rightarrow p = Z \rightarrow p$

$Z \rightarrow p = X$, $T2 \rightarrow p = Z$, $T3 \rightarrow p = y$

$X \rightarrow left = Z$, $X \rightarrow right = y$

• DELETION IN AVL TREE :-

We made 3 cases. 3rd case won't affect the AVL property in any AVL tree.

So, we need to discuss only 1st and 2nd case.

Here, first we search for the node to be deleted. Once deleted,

go to the parent of deleted node

① check if the AVL prop is violated.
② Update the height.

We keep on moving above until

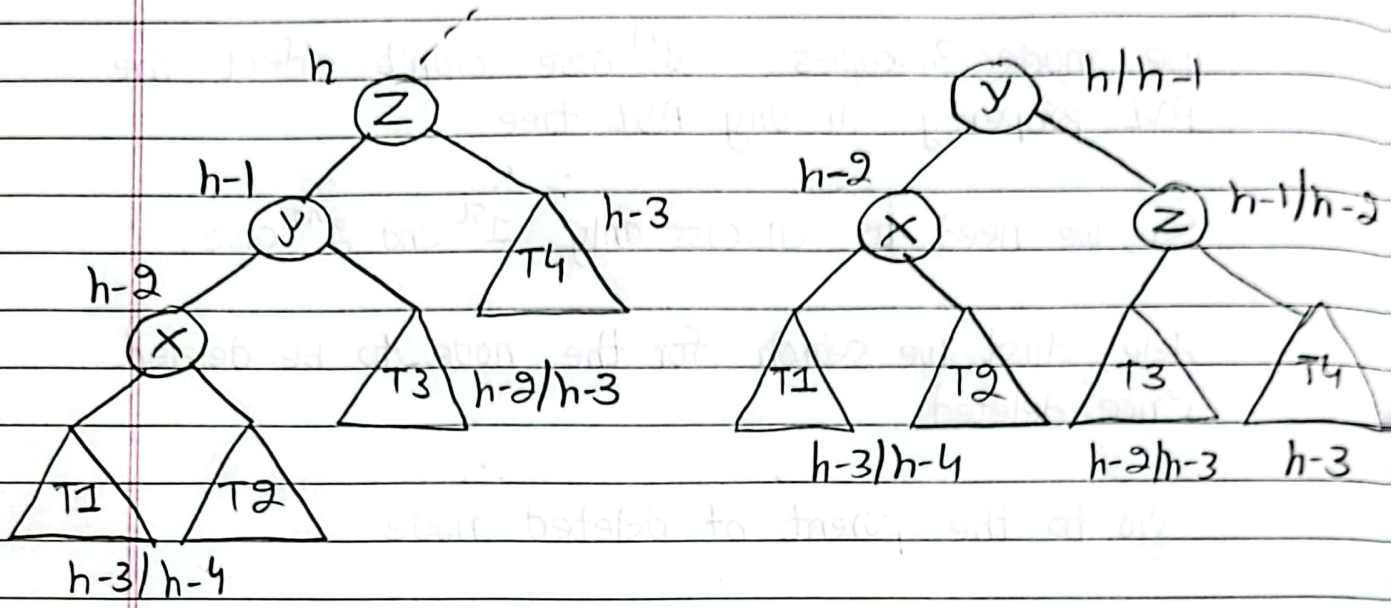
① we reach root node OR
② if height does not change and AVL prop is satisfied at any intermediate node.

⇒ If AVL prop is violated, we call the rotation function.

Note :- $H(z) = H(y)+1$
$H(y) = H(x)+1$

P.T.O.

- **Zig-Zig (0) - Deletion :-**

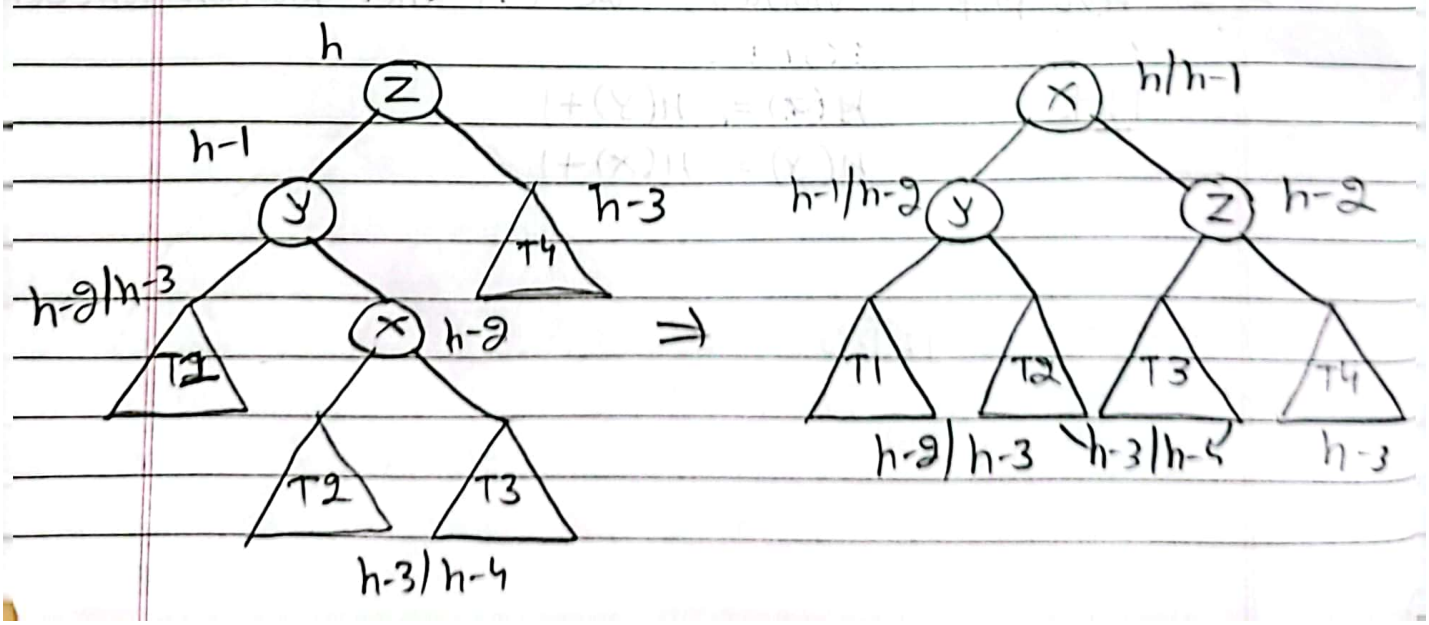


Clearly, we can tell that deletion happened at T4

After rotation, height of y can be h or h-1.

If height of y = h ⇒ We stop.
If height of y = h-1 ⇒ We go further up the tree.

- **Zig-Zag (1) - Deletion :-** Similar.



Similar for all other cases.

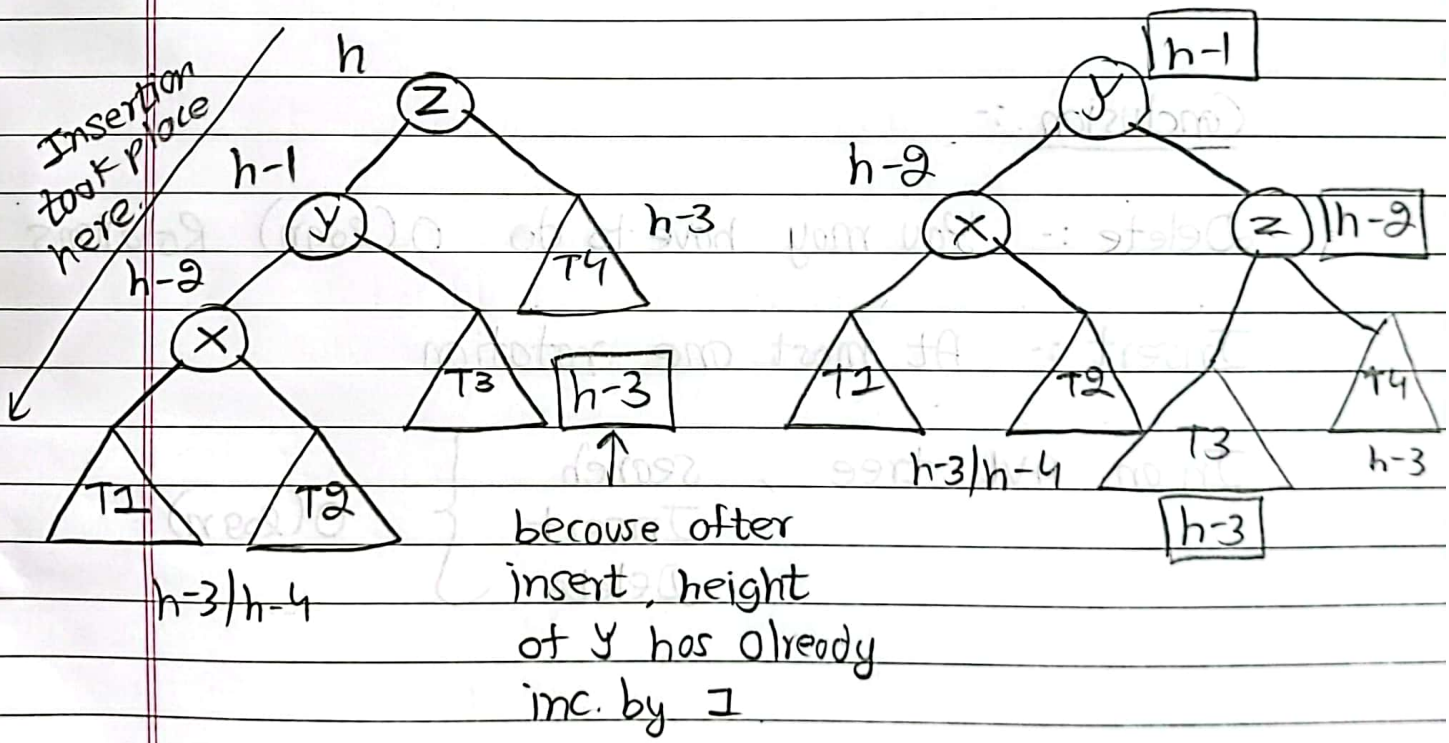
- **INSERTION IN AVL TREE :-**

Go to the parent node of the inserted node

① Update the height
② check if AVL prop is violated.

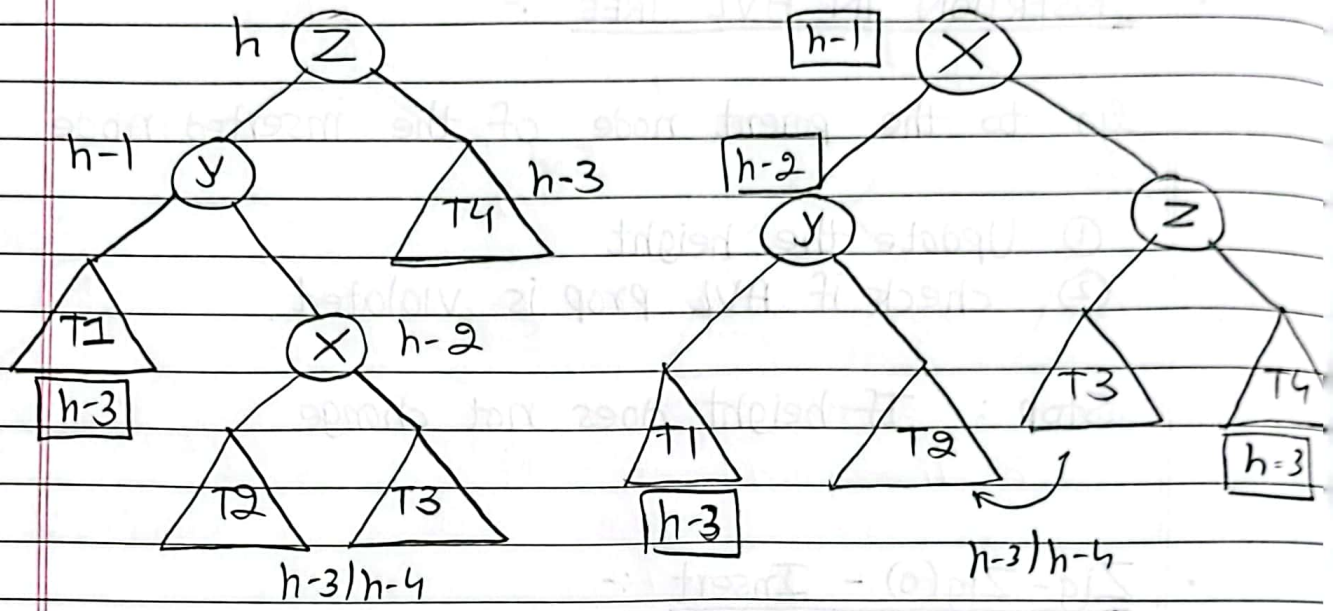
Stop : If height does not change

- **Zig-Zig(0) - Insert :-**



Note :- Height of subtree rooted at Y is fixed in all the cases.

Zig-Zag (1) - Insert :-



Conclusion :-

Delete :- You may have to do O(logn) Rotations.

Insert :- At most one rotation

In an AVL tree, search, Insert, Delete } O(log n)

Graphs

$G(V,E)$ is called graph

V is a finite set

& E is a subset of $V * V$ $\quad [V*V = \{(a,b) \mid a,b \in V\}]$

E is called relation on set V

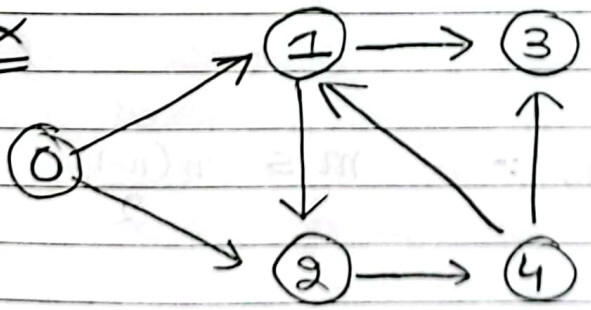
Let V be a finite set, $\boxed{n}$ nos.
We may assume $V = \{0,1,2 \ldots, n-1\}$

These are called nodes/vertices of graph.

E is a subset of $V*V$ is called edges of a graph
$\boxed{m}$ is the number of edges in the graph

When $(i,j) = e$ is an edge, we say i & j are neighbours or i and j incident on edge e.

Ex



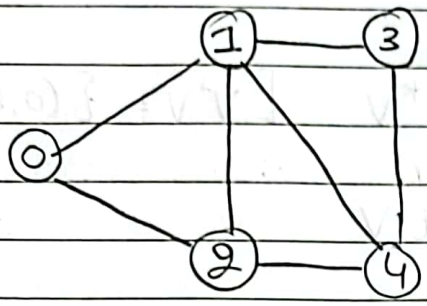
$E = \{(0,1), (0,2), (1,2), (1,3), (2,4), (4,1), (4,3)\}$

$n = 5$ & $m = 7$

Directed graph
(Arrows present)

Directed graph → The edge relation is not symmetric i.e. if (a,b) is an edge does not mean that (b,a) is an edge.

Undirected graph →



$E = \{(0,1), (0,2), (1,2), (1,3), (2,4), (4,1), (4,3)\}$

$n = 5$ & $m = 7$

The edge relation is symmetric here. So no arrows required.

Simple graphs :-

These are graphs without self loops. We will study simple graphs.

i.e. (a,a) is not a edge

NOTE :-

In Undirected graphs :- $m \leq \frac{n(n-1)}{2}$ $\left({}^{n}C_{2}\right)$

In directed graphs :- $m \leq n(n-1)$ (2 times)

If in any graph :-

m is of $\Theta(n^2)$ → Dense graph
m is max → Complete graph
m is $\Theta(n)$ → Sparse graph

- Weighted graphs :-

Edges are mapped to some real no. (called weight of that edge)

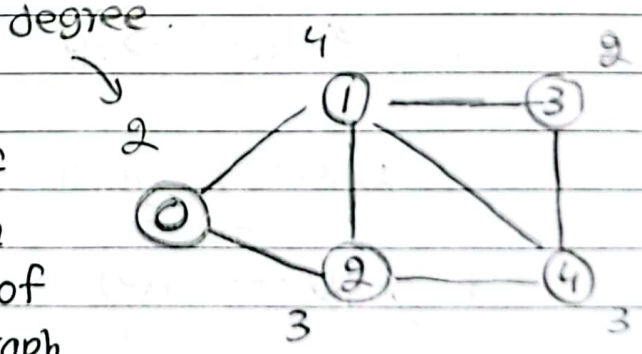
$$W : E \rightarrow R$$

It would be distance b/w 2 cities.

Weight of an edge could be negative

- Degree of a Node :-

Degree of a node is the no. of edges incident on the node.



Note :- Sum of degree of nodes in the graph is twice the no. of edges in the graph.

In case of Directed graphs :-

- In-degree of a node is the number of incoming edges to the node.

- Out degree of a node is the no. of in outgoing edges to the node.

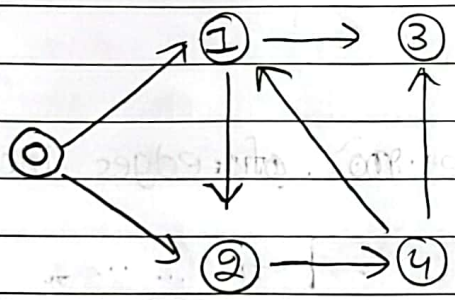
- Sum of in-degree of nodes = m
  Sum of out-degree of nodes = m.

2 Data Structures to Store Graphs :-

(i) ADJACENCY MATRIX :- takes n^2 space

A[n][n] is a Boolean matrix

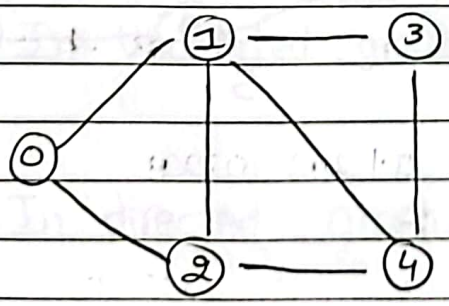
A[i][j] = true iff (i,j) is an edge of the graph.



Directed graph

Adj matrix.

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

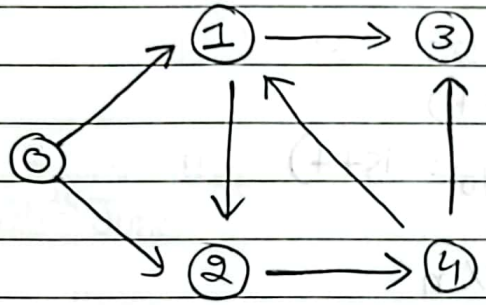


Undirected graph

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

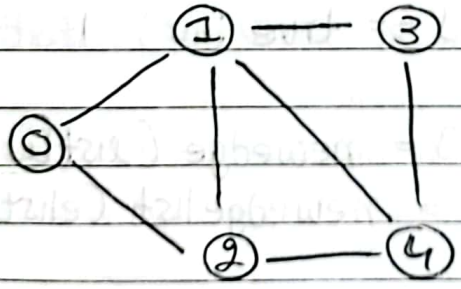
- In case of undirected graph, Adj matrix is a symmetric matrix
- Diagonal elements will be 0 in both cases because they are simple graphs.
- If edges are weighed, then instead of boolean matrix, take a float matrix & A[i][j] = weighed value.

(ii) ADJANCY LIST :-



0	→ 1 → 2
1	→ 2 → 3
2	→ 4
3	
4	→ 1 → 3

Directed graph → O(n+m) space



0	→ 1 → 2
1	→ 0 → 2 → 3 → 4
2	→ 0 → 1 → 4
3	→ 1 → 4
4	→ 1 → 2 → 3

Undirected graph → O(n+2m) space

- If edges are weighed, then in the struct of L.L., just add one more data type.

- **EDGE LIST :-** O(m) space.

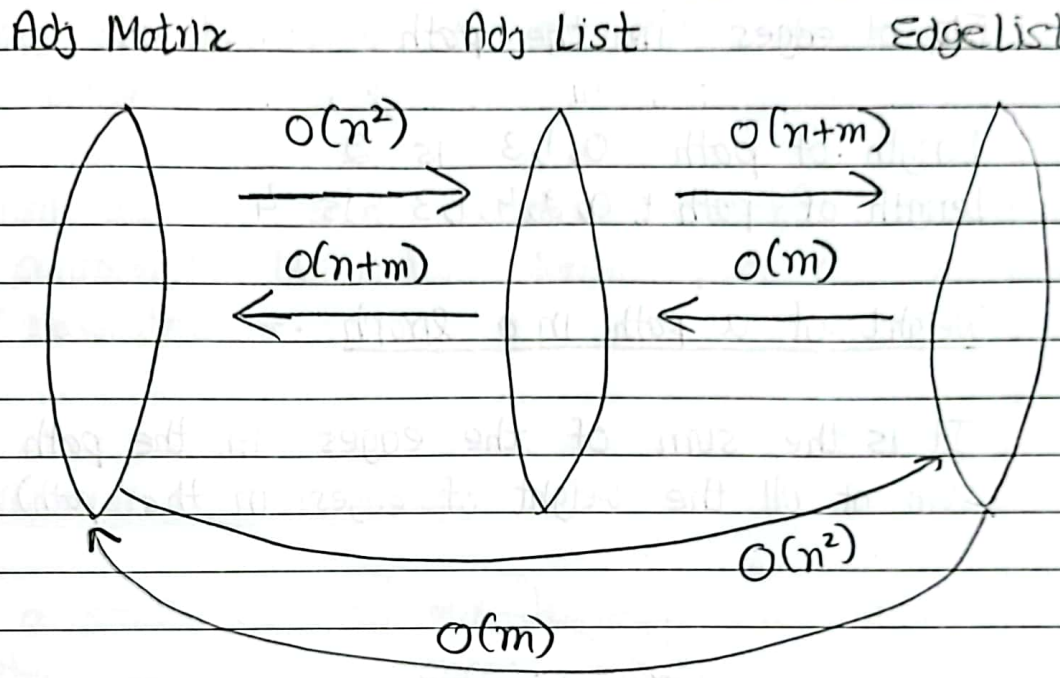
It is simply L.L. of edges. Each node in L.L. will store i, j and weight (if req.).

```
struct EdgeList {

 int i, j;
 struct EdgeList *next; }
```

Random Graph :-

```
for (i=0 ; i<n , i++)
  for (k=0 ; k<n/10 : k++)

{   j= rand() % n

    if ( ! A[i][j])

    {   A[i][j] == true

        list [i] = newedge (list[i], j);
        elist = newedgelist (elist, i, j);

        m++;

    }
}
```

Conversion :-



24th June

1 Paths in a graph :-

Path is a sequence of nodes $v_1, v_2 \ldots v_k$ such that (v_i, v_{i+1}) is an edge of the graph.

Ex 0, 1, 3 is a path

⇒ Cycle is a path in which first node is same as the last node.

1, 2, 4, 1 is a cycle.

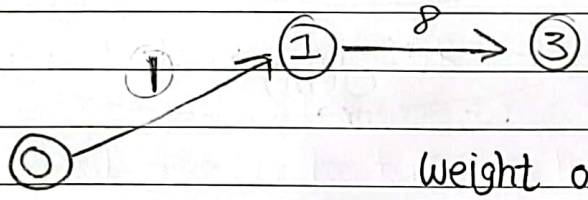
Length of a path :-

No. of edges in the path

Length of path 0,1,3 is 2
Length of path 0,2,4,1,3 is 4

Weight of a path in a graph :-

It is the sum of the edges in the path (basically sum of all the weight of edges in the path)



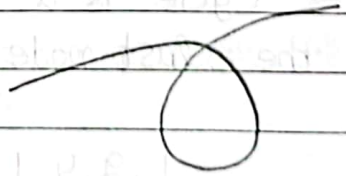
Weight of path 0,1,3 is
1+8 = 9

- ## SHORTEST PATH PROBLEM :-

Given two nodes s and t, find the shortest weight path.

Shortest path is always a simple path (i.e. it does not contain cycle.

⇒ If the weight of circle is +ve, then it is not shortest path.



⇒ If the weight of circle is -ve, then we just keep rooming in the circle ∞ times because everytime the weight dec. It is not acceptable.

- **LONGEST PATH PROBLEM :-**

Among all the paths b/w s and t, find a path with max weight.

⇒ We will observe that Shortest Path Prob. can be solved in polynomial time but longest path problem is a N-P hard problem.

- **HAMILTONIAN PATH PROBLEM :-**

Find a simple path which connects all the nodes of the graph exactly once.

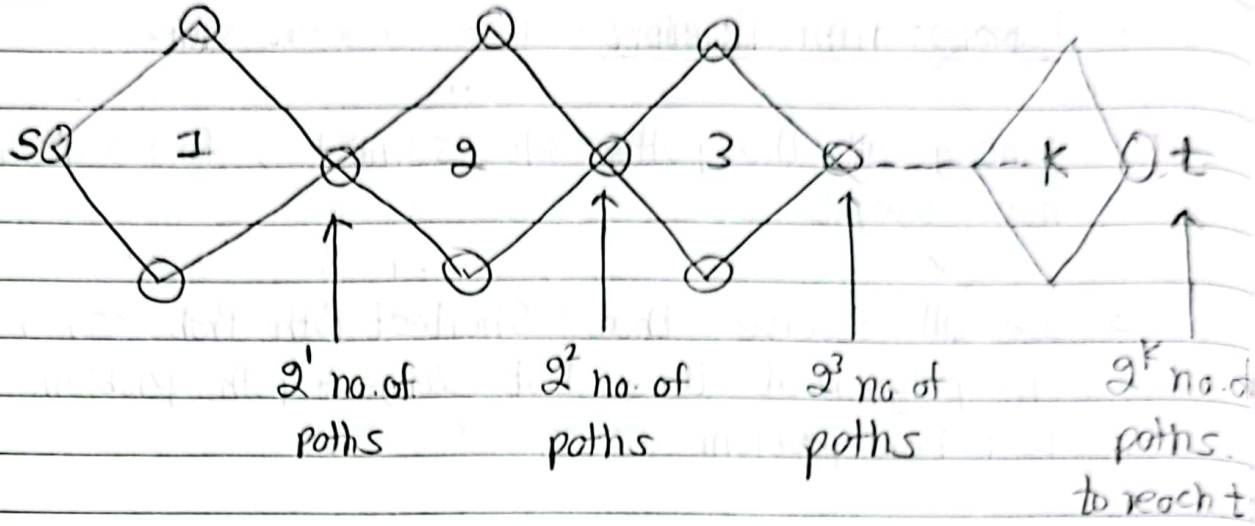
Note :-

→ Length of longest Path is n-1 ⇔ There is a Hamiltonian Path in the graph

→ If length of longest path < n-1 ⇒ There is no Hamiltonian Path in graph

- **Exponential Paths :-**

The no. of paths in a graph can be exponential in terms of no. of nodes & no. of edges in the graph.



In above case, the no. of paths from s to t is 2^k

and $n = 3k+1$ & $m = 4k$

⇒ So we cannot use brute force method to solve shortest path problem because it can take exponential time.

- **Single Source Shortest Path (SSSP) :-**

Find the shortest path from a source node s, to every other node in the graph.

It is more than single source single destination path problem.

- **Dijkstar's Algorithm** :- Works only when all weights are positive

  → $O((E+V)\log V)$ time
  → Uses Adjacency List

- **Bellman - Ford Algo :-** → O(VE) time
  → Uses Edge List

- **ALL PAIR SHORTEST PATH (APSP) :-**

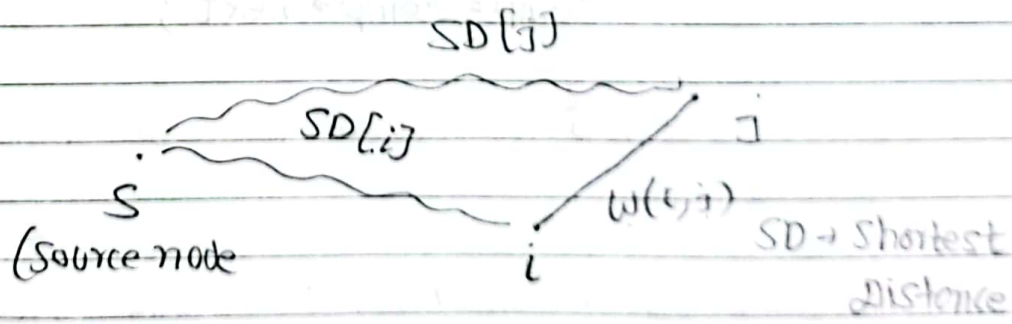
Find the shortest path between every pair of nodes in the graph.

- **Floyd - Warshall Algo :-** → $O(n^3)$
  → Uses adjacency matrix

- **BELLMAN - FORD ALGORITHM :-**

Uses Edge list rep. for the graph.

```
struct Edgelist {

int i, j; float w;
struct Edgelist *next; };
```



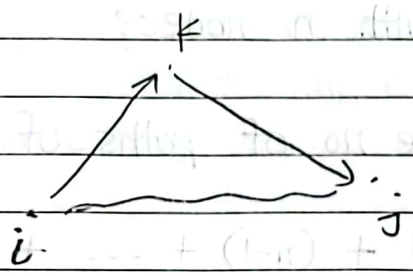
if SD[i] + w(i,j) < SD[j]

then update SD[j] to SD[i] + w(i,j)

```
Void BellmonFord (struct Edgelist *elist, float SD[],
                  int s, int n)
{   struct EdgeList *temp; int l;
    for (l=0, l<n, l++)
       { SD[l] = INT_MAX ; }
    SD[s] = 0;   (Distance from s to s is 0 only)
    for (l=0, l<n, l++)
    {     temp = elist
          while (temp) {
              if (SD[temp→j] > SD[temp→i] + temp→w)
                 { SD[temp→j] = SD[temp→i] + temp→w }
          temp = temp→ next;
          }
    }
```

FLOYD - WARSHALL ALGO :-

Find shortest path from i to j



Check weight of the shortest path from i to j is shorter than weight of shortest path from i to k plus from k to j ?

Code in lms.

2 No. of Paths :-

How many distinct simple paths are there in a complete graph with n nodes?

Let n_ℓ be the no. of paths of length ℓ.

n_1 :- $\underbrace{(n-1) + (n-1) + \dots + (n-1)}_{n}$

$n_1 = n * (n-1) \rightarrow \underbrace{(n-1) + (n-1) + \dots (n-1)}_{n}$

$n_2 = \underbrace{n * (n-1)}_{\text{You select path of length 1}} * \underbrace{(n-2)}_{\text{Then you can select } (n-2) \text{ paths for each path of length 1 u selected before.}}$

$\vdots$

$n_{n-1} = n * (n-1) * (n-2) \dots * 1$

$\therefore$ total no. of distinct paths. $\Rightarrow$

$$\sum_{i=1}^{n-1} n_i = n! \sum_{i=2}^{n-1} \frac{1}{(n-i)!}$$

26th June

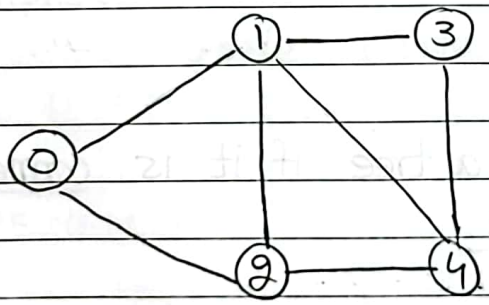
1 MIN/MAX SPANNING TREE :-

This problem is defined only on Undirected Graphs. So, we will be talking only of undirected graphs.

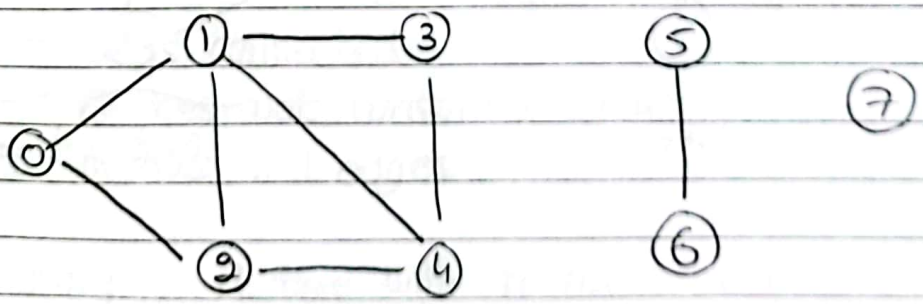
Terminolgies :-

→ Two nodes are connected if there is a path between them.

→ An undirected graph is connected if there is a path between every pair of nodes in the graph. (Note → Here we do not talking about direct path).



Connected graph



Unconnected graph

In the cases of not-connected graphs, we talk about a term connected component

⇒ A connected component of G is a maximal connected subgraph of G.

$$C\text{-}C. = \{[0,1,2,3,4], [5,6], [7]\}$$

$$= \{[0,1,2,3,4], [5,6], [7]\}$$

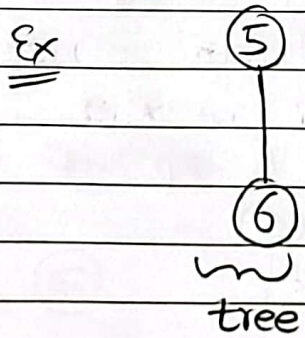
Hence, there are 3 connected components in the graph

Note → The set of nodes is the disjoint union of set of nodes in the connected components.

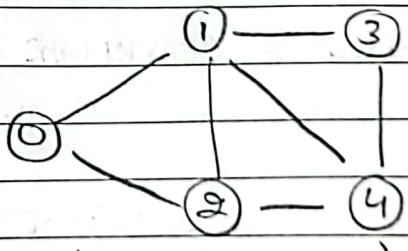
- TREES :-

An undirected graph is a tree if it is connected and does not contain a cycle.

Ex



tree

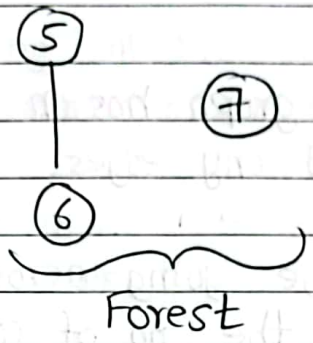


Not a tree

- **FOREST :-** → Collection of trees.

An undirected graph is a forest if it does not contain a cycle. (It may or may not be connected).

Ex



So, Basically forest is union of trees.

⇒ The Binary trees we studied are special kind of graphs.

Where (node, node → parent) is an edge.

- **THEOREM :-**

Let G be an undirected graph with n nodes. Any two of the following will imply the third.

1. G is connected
2. G does not contain a cycle
3. G has n-1 edges.

Corollary → A tree with n nodes will have exactly n-1 edges.

Observation :-

Let G be an undirected graph with n nodes. Let us start with only these isolated nodes and start adding edges in any order.

Note that initially graph has n connected components as we have not added any edges.

- If we add an edge going across the connected components, then the no. of connected components decreases by one.
- If we add an edge going within the connected component then we are introducing a cycle in the graph.

Now Proof :-

⇒ 1 and 2 implies 3

Since G does not contain a cycle, after adding K edges, we will have n-k connected components.

But 1 says G is connected.

$$\therefore \quad n-k = 1 \quad \Rightarrow \quad k = n-1$$

⇒ 2 and 3 implies 1

Since G does not contain a cycle, after adding k edges we will have n-k connected components.

So after adding all the edges, there will be n-(n-1) = 1 connected components, implying the graph is connected.

⇒ 1 and 3 implies 2

If k edges has gone across the connected component then we have n-k c.c.

But graph is connected, implying k = n-1

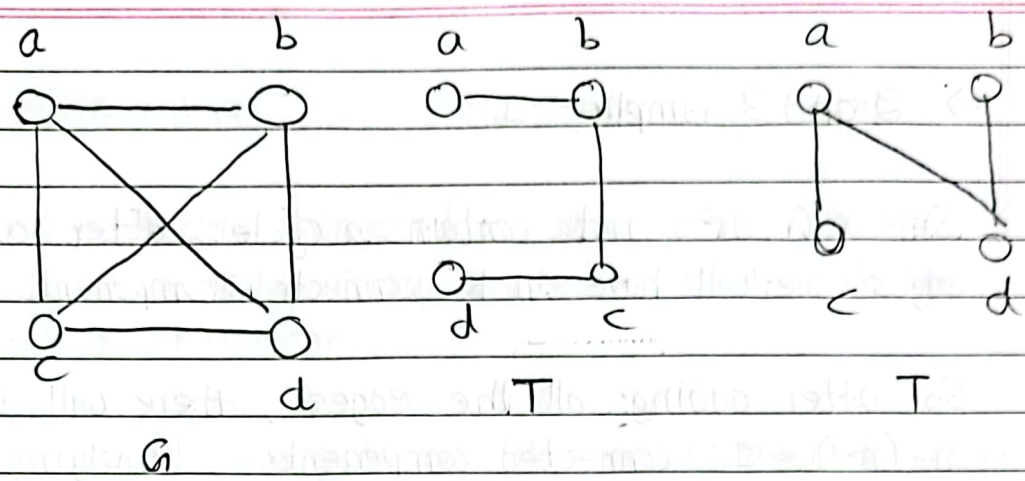
All edges has gone across the C.C. Hence no cycle in the path.

- **SPANNING TREE** :-

Let G be a graph of n nodes/vertices.

Spanning tree T of G is a subgraph of G, which is a tree with n nodes (i.e. T covers all the nodes of graph G). T will contain (n-1) edges.

Thus :-
→ The no. of nodes in T = No. of nodes in G
→ T will have (n-1) edges.



So, a graph can have more than one spanning tree

- **WEIGHT OF T :-**

Weight of spanning tree is defined as the sum of the weights of the edges of the spanning tree.

- **MST :-**

Minimum Spanning Tree (MST) is a spanning tree with min weight.

Maximum Spanning Tree (MST) is a spanning tree with max weight.

Both can be solved in polynomial time.

2 CAYLEY'S THEOREM :-

The no. of distinct spanning trees in a complete graph with n nodes is

$$n^{n-2}$$

Observation :- A tree with n nodes ($n \geq 2$) will have atleast 2 nodes with degree one (leaf nodes). Any tree will have atleast 2 leaf nodes. Also, a degenerate tree have such 2 nodes.

Proof :- Let us assume the observation is not true. then let deg(i) be degree of any node.

$$\sum \deg(i) \geq 2n \quad \cdots (1)$$

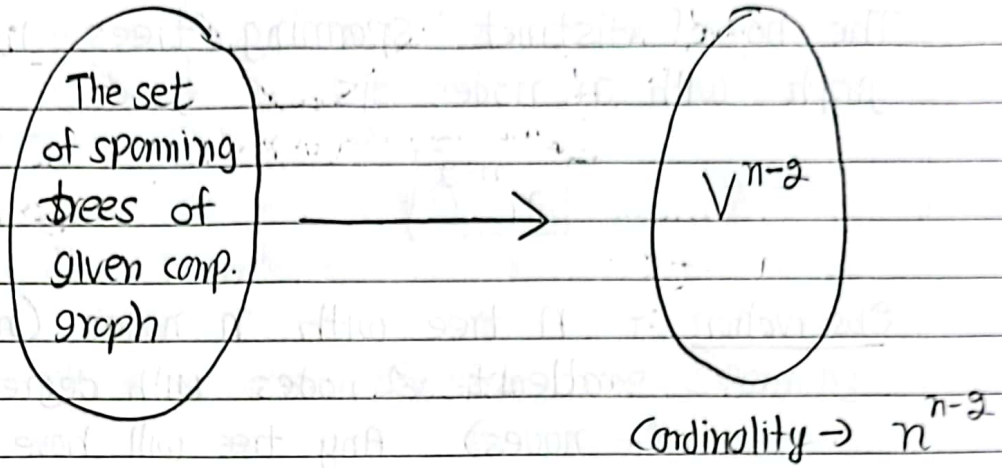
But we also know that tree has (n-1) edges. Each edge inc. degree of graph/tree by 2

$$\therefore \sum \deg(i) = 2(n-1) \quad \cdots (2)$$

From (1) & (2) $\Rightarrow$ $-2 > 0$ Not possible

Hence the obs. is true. (Similar we can consider the case when only 1 node with degree 1)

Proof of Cayley's Theorem :-



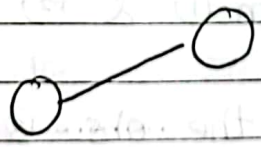
Cardinality → n^{n-2}

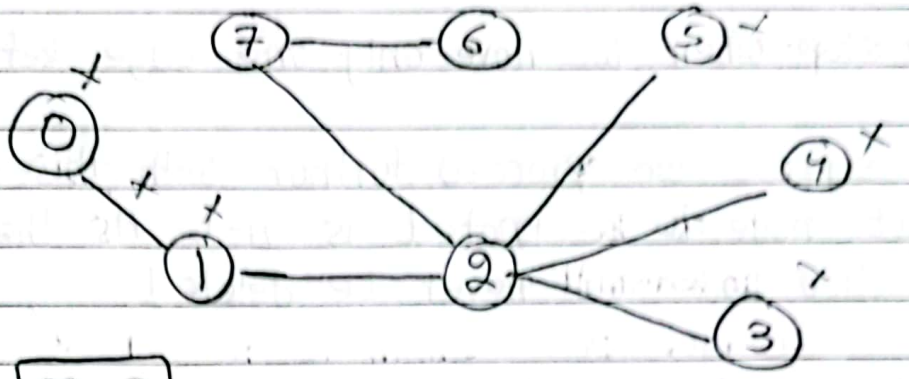
Showing the bijection between the sets will proove the theorem.

We will see mapping both ways.

I→ Given a spanning tree with n nodes, pick a degree one node with least label and delete the node along with the edge and note down the label of the other end point of the edge that is deleted.

Continue this till only one edge is left





$n = 8$

(i) Pick 0 and delete. Note 1.

(1)

(ii) Pick 1 and delete. Note 2

(1, 2)

(iii) Pick 3 and delete. Note 2

(1, 2, 2)

(iv) Pick 4 & 5 and delete. Note 2 & 2

(1, 2, 2, 2, 2)

(vi) Pick 2 and delete. Note 7

(1, 2, 2, 2, 2, 7)

Now we stop with remaining nodes 6 and 7

We stop when we have only one edge left

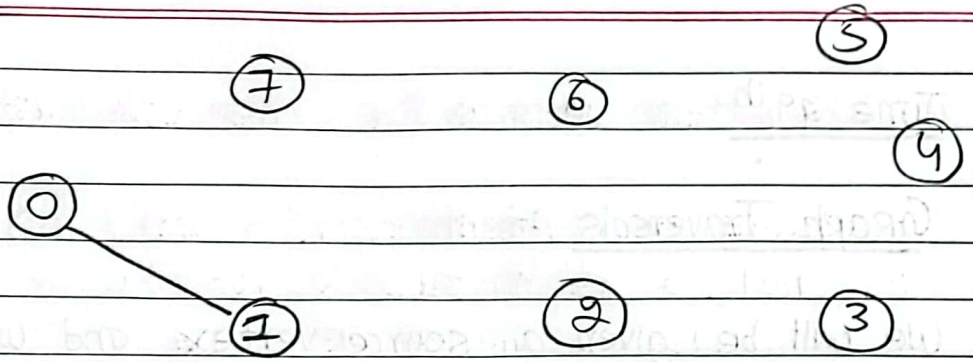
If suppose we proceed furthur with this algo, the last node to be noted is n-1. As the highest labelled node will never be deleted.

But we will not write it in our noting array because in a tree, there should be atleast 2 nodes with degree one.

In this algo, we do delete & note step (n-2) times.

II → Given a vector of length n-2. Pick a node with least label that has not been picked so for and also it is not part of the vector. Connect that node with the first ~~vector~~ node present in the vector and delete that node from the vector.

Repeat this until the vector is empty. Connect the not picked node with n-1 node finally.



given → (1, 2, 2, 2, 2, 7)

(i) Pick 0 and connect to 1. Remove 1

(2, 2, 2, 2, 7)

(ii) Pick 1 and connect to 2. Remove 2

(2, 2, 2, 7)

(iii) Pick 3, 4 & 5 and connect to 2. Remove 2's

(7)

(vi) Pick 2 and connect it to 7. Remove 7

( )

We stop now. We see that we got back our original tree.

Rem :- At last, we have to connect not picked node (i.e. 6) with n-1 (i.e 7)

June 29th

1 Graph Traversals :-

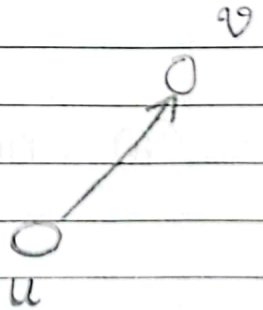
We will be given a source vertex and we have to visit every vertex which is reachable from the source vertex.

If the graph is disconnected, then we have to apply this algo on each connected components of the graph.

Given :- A graph in adjacency list rep. and a source node s.

We maintain a DS (Data Structure)

```
Add (DS, s)

while (DS != empty)
{   u = Delete (DS)

    For every edge (u, v), if

    1. v was deleted  - Do nothing
    2. v is not added - add v
    3. v is in the DS - update.

}
```



} we do one of them

Note → Here update will depend on the DS we use.

- DS is queue - do nothing
- DS is stack - Add to stack
- DS is priority queue -
    - Min - Decrease key
    - Max - Inc. key.

- Phi Function :-

Phi(s) = -1

while (DS) {
  u = Delete(DS)
  For every edge (u, v)

Whenever v is added or updated

Phi(v) = u // v is being discovered by u

Note :- (Phi(v), v)) for all $v \in V \setminus \{s\}$ is a tree

We will call it the tree associated with this traversal.

1 →
2
3

⇒ We get diff algorithms if we use diff DS.

- Queue → Breadth First Search (BFS)
- Stack → Depth First Search (DFS)
- Min Priority Queue → Dijskstra
- Priority Queue → Prims
    - Min → Min. Spanning tree
    - Max → Max. Spanning tree.

2 BREADTH FIRST SEARCH :-

Given :- A source vertex and Adjacency list rep. of graph.

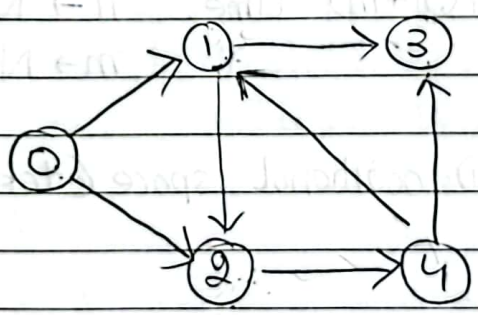
We will maintain DS → Queue and an array which will keep track of visited nodes V[] & Phi function.

```
for(i=0; i<n; i++)
{   Phi(i) = -2;
    V[i] = false; }

Enqueue(Q,s);
Phi(s) = -1;
V[s] = true;
```

// V[i] = True will indicate that this node has been added to Queue

```
while (Q) {
    u = Dequeue (Q) ;
    For every edge (u,v)  → traverse through Adjacency
                            list with head = u
    {   if (V[v] == false)
        {   Enqueue (Q, v) ;
            Phi(v) = u ;
            V[v] = true ;
        }
    }
}
```

Ex



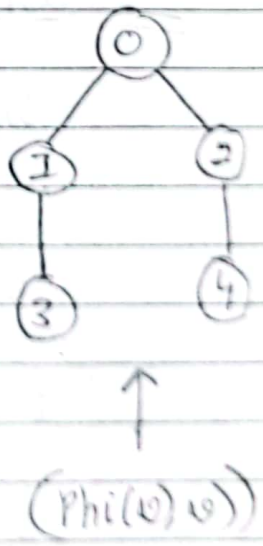
Before

i	0	1	2	3	4
Phi	-1	-2	-2	-2	-2
V	f	f	f	f	f

After

i	0	1	2	3	4
Phi	-1	0	0	1	2
V	T	T	T	T	T

BFS Tree



⇒ The tree we get by looking at (Phi(v), v) except for source node.

We call this tree as **BFS Tree**

⇒ In BFS, we start with the source vertex and then we visit all the nodes which are immediately adjacent to that node. Similarly we proceed further. We visit that nodes which haven't been visited before.

⇒ If we apply BFS on the Binary tree, we get is basically level order traversal of that Binary tree.

⇒ BFS takes $O(n+m)$ time. $n \rightarrow$ No. of nodes
$m \rightarrow$ No. of edges.

BFS takes $O(n)$ additional space (taken by queue)

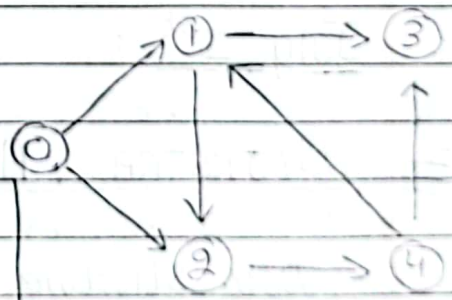
3 DEPTH FIRST SEARCH :-

We will use **DS = Stack**

```
for(i=0; i<n; i++)
{   Phi(i) = -2;
    V[i] = false;
}
```

V[i] == true will indicate that node has been deleted from the stack

```
Push(s,s) ;
Phi(s) = -1 ;

while (s) {

    u = Pop(s)

    if ( V[u] == folse)

    {       V[u] = trve
    }

      ← For every edge (u,v)

            {   if (V[v] == folse)

                    {  Push(s,v) ;
                       Phi(v) = u ;
                    }
            }
    }
}
```



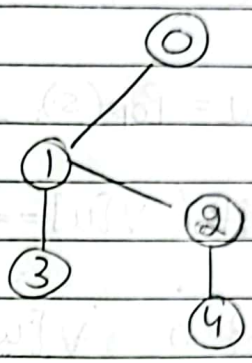
Before

i	0	1	2	3	4
Phi	-1	-2	-2	-2	-2
V	T	F	F	F	F

After

i	0	1	2	3	4
Phi	-1	0	1	1	2
V	T	T	T	T	T

See collection of ((Phi(v)), v) and we get DFS Tree.



DFS Tree

⇒ In DFS, we keep on moving furthur and furthur in the graph until we cant go more. The we kind of backtrack.

⇒ DFS takes O(m) time & O(m) additional space.

⇒ The algo we discussed is iterative DFS.

July 1st :-

1 DIJKSTRA'S ALGORITHM FOR SHORTEST PATH

Data Structure → Min heap / Min priority queue.

Note :- All the weights must be positive

Priority is given to D[node]

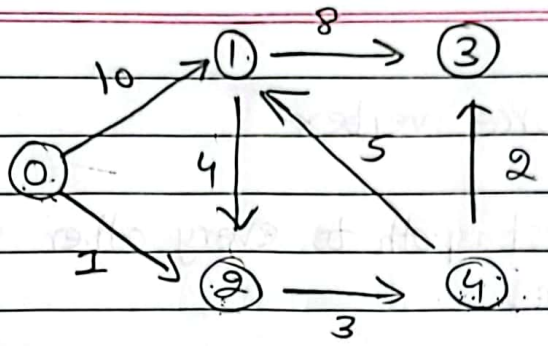
Given :- A source vertex

To Find :- Shortest path to every other vertex

```
for(i=0 ; i<n ; i++)
{   Phi(i) = -2 ;
    V[i] = false ;
    D[i] = infinity ;
}
D[s] = 0 ; → DecKey (s, 0)
Phi(s) = -1 ;

while (PQ)
{   u = DeleteMin()
    V[u] = true ;

    For every edge (u,v) {

        if (V[v] == false && D[v] > D[u] + w(u,v)) {

            D[v] = D[u] + w(u,v) → DecKey(v, D[u]
                                   + w(u
            Phi(v) = u }

    }
}
```

V[i] will be true when i^{th} node is deleted from priority queue.

D[i] will give us shortest path from source node to i^{th} node



Before :-

i	0	1	2	3	4
Phi	-1	-2	-2	-2	-2
V	T	F	F	F	F
D	0	∞	∞	∞	∞

PQ ⇒

0	0
node	D[node]

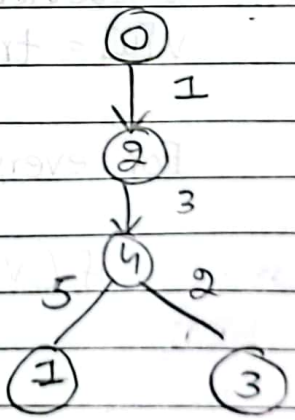
After :-

i	0	1	2	3	4
Phi	-1	4	0	4	2
V	T	T	T	T	T
D	0	9	1	6	4

gives shortest dist. →

⇒ D[i] will give us length/weight of shortest path. For the path, we can refer to the Tree

⇒ The weight of edges of tree is same as corresponding edges in the graph.



Dijkstra's shortest path tree

Complexity :-

n DeleteMin() $\rightarrow \log n$

m Dec.key()

$\Rightarrow$ Min Heap $\rightarrow$ $O((n+m)\log n)$

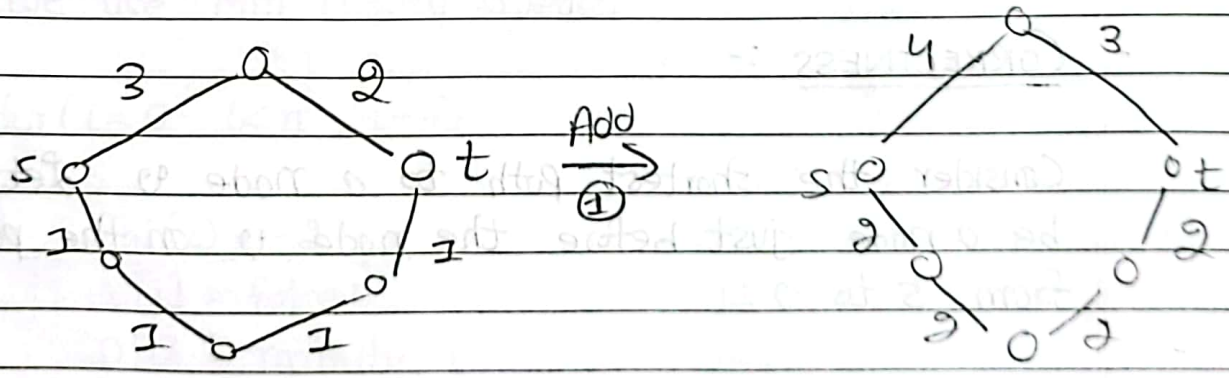
Fib. Heap $\rightarrow$ $O(n\log n + m)$

Imp Points :-

1. This algo fails when any edge in the graph has negative weight.

2. If we use Max Heap instead of Min Heap, this algo does NOT compute the longest path.

3. If we add any +ve/-ve constant to all the edges weights The shortest path of the original graph may change w.r.t. new graph



Shortest path from s to t changes

- **CLAIM :-**

If $v_1, v_2, \ldots, v_n$ is the order in which the nodes are deleted from the Min heap then

$$D[v_1] \leq D[v_2] \leq \cdots \leq D[v_n]$$

Proof :- By induction

Consider the node i, let v_i and $D[v_j] < D[v_i]$ let j be the smallest such j.

When i was deleted $D[v_i] \leq D[v_j]$

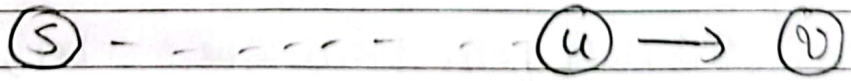
Even in future, if $D[v_j]$ dec, it can decrease to

$$D[v_j] = D[v_k] + w(k,j) \quad \text{for some } k,\ i \leq k \leq j$$

$$D[v_j] = D[v_k] + w(k,j) \geq D[v_i] + w(k,j) > D[v_i]$$

- **CORRECTNESS :-**

Consider the shortest path to a node v. Let u be a node just before the node v on the path from s to v.



We will show by induction.

for the source node $D[s] = 0$ is trivial.

So, As $w(u,v) > 0 \Rightarrow D[u] < D[v]$, so u must be deleted before v.

(We are assuming that till u, we have correctly calculated the shortest path ie. induction proof)

When u was deleted, edge (u,v) should have been considered and D[v] can not be > than [D[u] + w(u,v).

2 PRIMS ALGORITHM :- It is only applicable for undirected graphs.

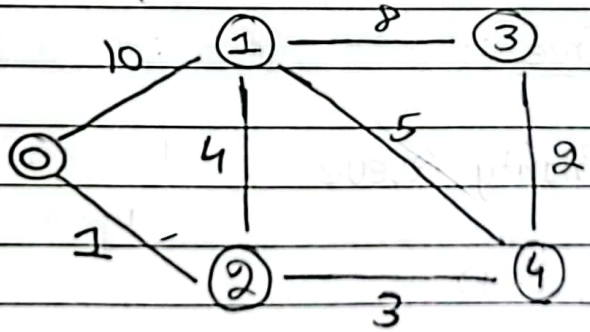
Min Spanning Tree :-

We use Min Priority Queue

```
for(i=0; i<n; i++)
{   Phi(i) = -2;
    V[i] = false;
    D[i] = infinity;
}
```

V[i] = true when that element is deleted from priority queue

```
D[s] = 0;          // DecKey (s, 0)
Phi(s) = -1;

while (PQ)
{   u = DeleteMin()
    V[u] = true;

    for every edge (u,v)
    { if (V[v] == false && D[v] > w(u,v))
      { D[v] = w(u,v)   // DecKey (v, w(u,v))
        Phi(v) = u;
      }
    }
}
```



0	0

Before :-

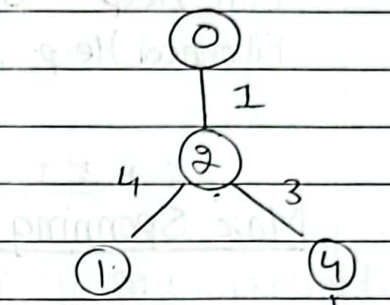
i	0	1	2	3	4
Phi	-1	-2	-2	-2	-2
V	T	F	F	F	F
D	0	∞	∞	∞	∞

After :-

i	0	1	2	3	4
Phi	-1	2	0	4	2
V	T	T	T	T	T
D	0	4	1	2	3

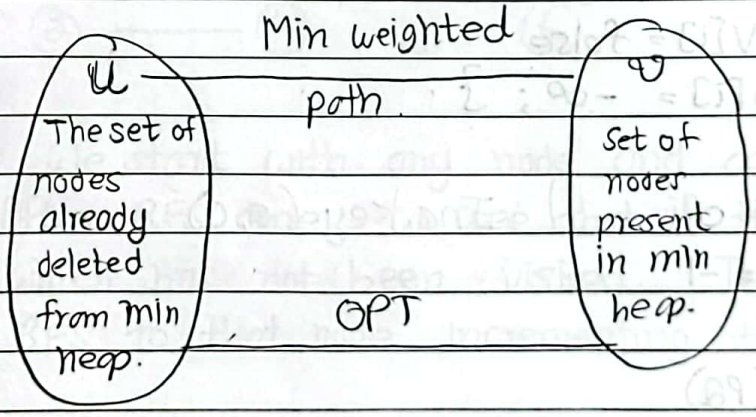
⇒ Weight of min spanning tree :-

$$\sum_{i=0}^{n-1} D[i] = 0+4+1+2+3$$
$$= 10$$



Min Spanning tree

- CORRECTNESS :-



Let OPT be the min spanning Tree

Add the edge (u,v) to OPT, weight of the maximum weighted edge should be $w(u,v)$.

OPT should choose at least one edge going across whose weight is some as $w(u,v)$.

Complexity :-

n DeleteMin()
m Dec.key()

Min Heap $O((n+m)\log n)$
Fibonacci Heap $O(n\log n + m)$

Max Spanning Tree :-

We use Max Heap / Max priority queue.

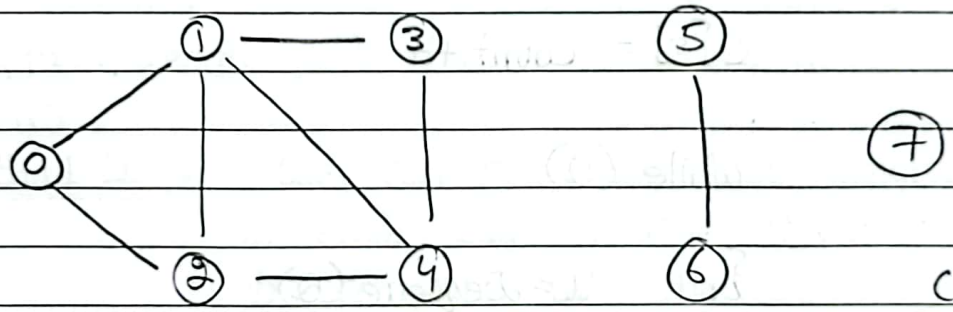
```
for(i=0, i<n, i++)
{   Phi(i) = -2
    V[i] = folse
    D[i] = -∞ ; }

D[s] = 0        || Inc.key(s, 0)
Phi(s) = -1

while (PQ)

{   u = DeleteMax()
    V[u] = true

    for every edge (u,v)

    {   if (V[u]==folse && D[v] ≤ w(u,v))
```

```
        { d[v] = w(u,v)          // Inc.key (v, w(u,v))
          Phi(v) = u;  }
    }
}
```

3rd July

1. BFS Application : Connected Components

We have to count the no. of C.C. in any undirected graph using BFS.



Procedure :- We start with any node and apply BFS. After BFS done, we see that if there is any node which has not been visited. Then we apply BFS to that node incrementing the count.

We stop when all the nodes are visited.

Code on next page.

```
for (i=0 ; i<n ; i++)
{   C[i] = 0 }  →  C[i] will be non zero when the node i is visited
count = 0
for (i=0 ; i<n , i++)
{   if (C[i] == 0)
    {   s = i
        Enqueue (Q, s)
        C[s] = count++          || Applying BFS on each C.C.
        while (Q)
        {   u = Dequeue (Q)
            For every edge (u,v)
            {   if (C[v] == 0)
                {   Enqueue (Q, v)
                    C[v] = cant ;
                }
            }
        }
    }
}
```

So, given a Connected graph, the no. of c.c. and labelling the c.c to which a node belongs to can be done in $O(n+m)$ time.

2 BFS Application : Shortest Length Path

- Given a graph (directed or undirected)
- Length of a path is the no. of edges in the path.
- So, given a source node s, we have to find the shortest length path from s to every node in the graph.

Method-I :- We can set the edge weight as 1 and for each edge, use dijstra's algorithm to compute the shortest length path

It takes $O((n+m)\log n)$ time.

Method-II :- We can use BFS to compute shortest length path.

It taken $O(n+m)$ time.

Code on next pg.

```
for(i=0; i<n; i++)
{   Phi(i) = -2
    V[i] = false        V[i] = true when ith node is
    L[i] = infinity       visited
}

Enqueue (Q,s)
Phi(s) = -1;  V[s] = true ;  L[s] = 0 ;

while (Q)
{   u = Dequeue (Q)

    For every edge (u,v)
    { if ( V[v] == false)
        {   Enqueue (Q, v)
            Phi(v) = u
            V[u] = true
            L[v] = L[u] +1
        }
    }
}
```

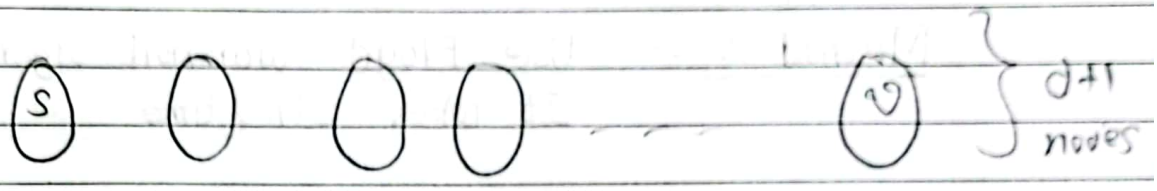
So, finally | L[i] = length of shortest length path of node i from s

L[i] is also the level of node i in the graph

- If $L[v]$ = infinite $\Rightarrow$ That node is not reachable from s.

- **CORRECTNESS** :-

If d is the length of the shortest length path from s to v. $L[v] = d$.



Proof is by induction :-

For s node, $L[s] = 0$

Now we assume that for i^{th} node $L[i]$ is computed correct.

Now $L[i+1] = L[i] + 1$ which is correct

$\therefore$ it worked for $i+1$ as well.

- **DIAMETER OF A GRAPH** :-

Diameter of a graph is the maximum distance b/w two nodes of the graph.

Distance between two nodes is the weight of the shortest path between the nodes.

Note → This is not same as the longest path in the graph.

Concept → We find the shortest path between every pair of node in the graph. Then we pick the max out of them.

Method-I :- Use Floyd-Worshall algorithm. It takes $O(n^3)$ time.

Then in linear time, we can find max among the values calculated by floyd-Worshall algorithm.

But if all the edges have unit weight, then the distance b/w 2 nodes is the length of the shortest length path b/w the nodes.

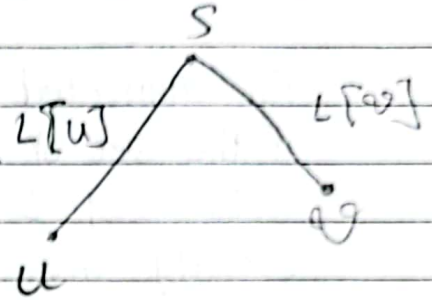
So, in this case, we can compute diameter in $O((n+m)n)$ time by calling BFS from each node of the graph.

- OBSERVATION :-

Let G be an undirected connected graph

Let L be the max level of any node when applied BFS from a node s.

Let d be the diameter of the graph and let u and v be the nodes such that d is the length of the shortest length path b/w u and v.



$$d \leq L[u] + L[v]$$

$$\leq 2L$$

$$\boxed{L \leq d \leq 2L}$$

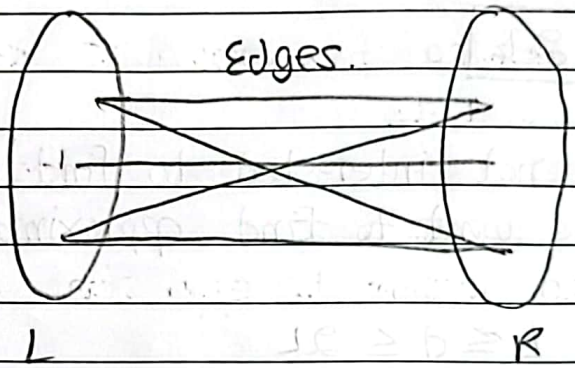
So, if we are not interested to find the exact value of diameter, we want to find approximate value, Find L and $L \leq d \leq 2L$

This relation is applicable for undirected graph only.

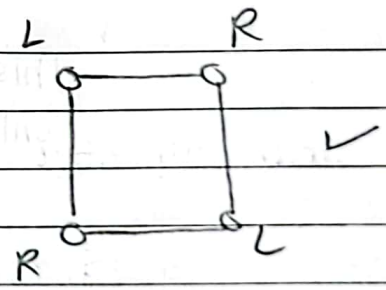
3 BFS Applications : Bipartite Graphs

- **BIPARTITE GRAPHS :-**

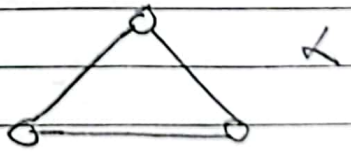
An undirected graph is said to be a bipartite graph if the set of nodes can be divided into two disjoint sets say L and R such that every edge has exactly one end point in L.



Is a Biported graph ⇒



Not a Biported graph ⇒



• COLOURABLE GRAPHS :-

→ An undirected graph is said to be a K- coloured graph, if every node can be assigned one of the colours from {1, 2, --, K} such that for every edge (u,v), $c(u) \neq c(v)$

$c(u)$ → color of u
$c(v)$ → colour of v.

→ If $K > 2$, checking if a graph is K-colouroble or not is a NP hard problem.

Both ways ↓

⇒ **An Undirected graph is Bipartite iff it is 2-colouroble**

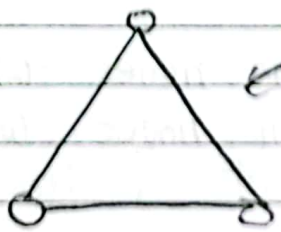
→ I can colour all the nodes in set L of one colour and all the nodes in set R of another colour.

• KONIG THEOREM :-

→ **An undirected graph is 2-colouroble ⇔ it does not contain an odd cycle.**

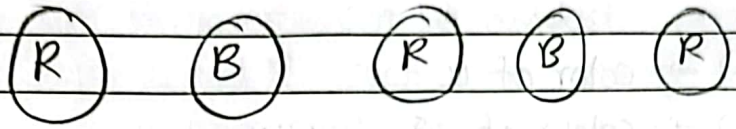
Odd cycle → In a cycle, **no. of edges = No. of nodes = odd no.** it is called odd cycle.

Ex



Proof :-

I If the graph has a odd cycle, then it is not 2-colouroble.



As the no. of nodes is odd, so if I start marking nodes with any colour, first and last node will have same colour. Thus when we connect them, it does not remain 2-colouroble.

II If the graph does not contain odd cycle, then it is 2-colouroble.

→ Fix a node v
→ $L = \{u \mid$ there is odd length path from v to $u\}$
→ $R = \{u \mid$ there is even length path from v to $u\}$

→ A node is in either L or R
→ There is no edge between two nodes of L
→ There is no edge between two nodes of R

So, the code is same as we wrote 2 pages back to calculate level of each node.

L will contains nodes whose level is odd
R will contain nodes whose level is even.

Then, for every edge in the graph (u, v)

if (L[v] %2 != L[u] %2), then the given graph is not a bipartite graph (even if one edge only does not satisfy the relation, then we conclude that the graph is not bipartite graph).

Thus, the whole algo is linear time.

6th July :-

1 RECURSIVE DFS :-

No stack. Maintain 2 arrays D=[] & F=[] indicating discovery time and finishing time.

```
DFS visit (G)                          // Note → D[] & F[] and
{   for (i=0; i<n; i++)                   count are global
    {   Phi(i) = -2;   D[i] = 0;          variables
        V[i] = false;  F[i] = 0;
    }

    count = 1;

    for (i=0; i<n; i++)

    {   if (V[i] == false) { Phi(i) = -1; DFS(G,i) }
    }
}
```

```
DFS (G, u)

{   V[u] = true ;
    D[u] = count++ ;

    for every edge (u, v)

    {   if ( V[u] == false)

        {   Phi(v) = u ;
            DFS (G, v) ;
        }
    }

    F[u] = count++ ;                    // O(m) algo
}
```

Paste DFS slides. (4 slides)

We can update iterative DFS to compute D[] & F[]

2 Properties of Discovery and Finish times.

D[ ] → n distinct values
F[ ] → n distinct values.

Every node is associated with a closed interval

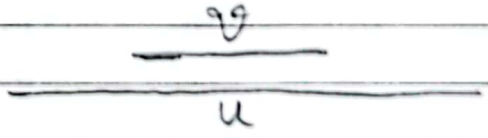
[D[u], F[u]]

u was discovered before v

⇒ Let u and v be 2 nodes such that D[u] < D[v], then one of the following should be true

in diff cc ← F[u] < D[v]

in same cc. F[v] < F[u]



Application: For 2 nodes u and v, if [D[u], F[u]] ⊂ [D[v], F[v]]

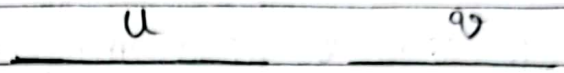
then v is ancestor of u

and if [D[u], F[u]] & [D[v], F[v]] are disjoint, then

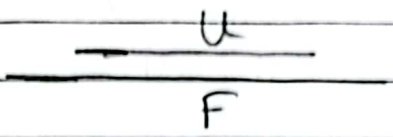
u & v are in diff. subtrees.

⇒ Let u and v be two nodes such that F[u] < F[v] then one of the following should be true

- F[u] < D[v]

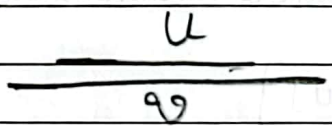


- D[v] < D[u]



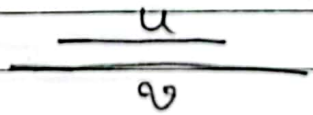
Theorem - 1 :-

Let (u,v) is an edge in the graph such that F[u]<F[v] then there must be a path in the graph from v to u.



Theorem - 2 :-

If there is a path from u to v in the graph s.t. F[u] < F[v], then there must be a path in the graph from v to u.



July 8th

Applications of DFS → Topological Sort of DAG.

- **DIRECTED ACYCLIC GRAPH (DAG) :-**

DAG is a directed graph which does not contain a cycle.

Apply DFS on G, then list the nodes in the dec. order of finish times.

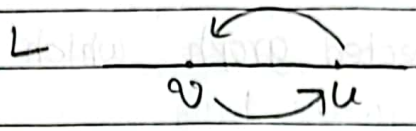
Code is similar to recursive DFS.

```
DFS (G,u) {

V[u] = true
D[i] = count++;

For every edge (u,v)

{ if (V[v] == false)

    { Phi(v) = u ; DFS(G,v) ; }

}

F[u] = count++;    L[n-c] = u ; c++ ;

}
```

This array will contain topological sort

Now, pick $i<j$ and let $v = L[i]$ and $u = L[j]$ then $F[u] < F[v]$

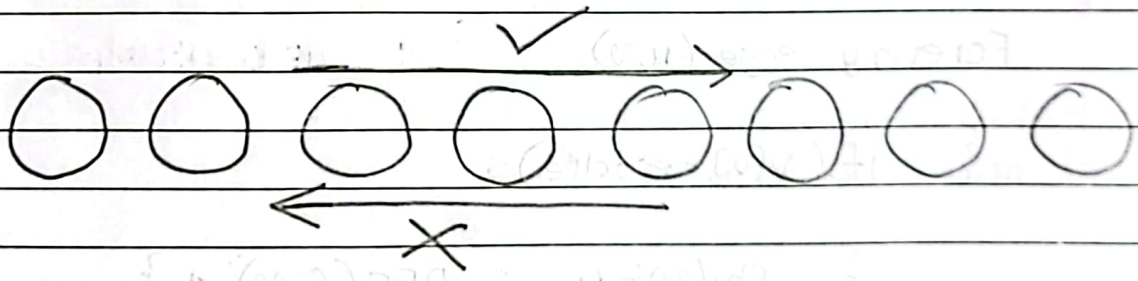
This implies that if (u,v) is on edge in the graph then there must be a path from v to u in the graph.



But then the cycle will be formed and graph will not remain DAG.

Ah So, if we are given DAG, then (u,v) cannot be an edge.

There will be only forward edges and no backward edges b/w nodes present in array L.



Only forword dependencies.

2 SHORTEST AND LONGEST PATH IN DAG :-

We can find both shortest and longest paths from a source node to every node in the graph in linear time

```
Paths(s)
{   for(i=0 i<n i++)
    {   Long[i] = Min      Phi1(i) = Phi2(i) = -2
        Short[i] = Max
    }

Long[s] = 0          Phi(s) = Phi2(s) = -1
Short[s] = 0

for (i=0, i<n, i++)
{   u = L[i]          // → L is the array for topological sort.
    for every edge (u,v)
    {   if ( short[v] > short[u] + w(u,v))
        {   sh[v] = short[u] + w(u,v) ; Phi(v) = u ;
        }
        if ( Long[v] < Long[u] + w(u,v))
        {   Long[v] = Long[u] + w(u,v), Phi2(v) = u;
        }
    }
}
```

3 STRONGLY CONNECTED COMPONENTS

We studied connected components in case of undirected graph.

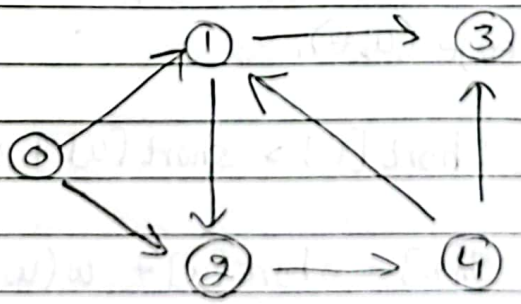
In case of directed graph, we define S.C.C.

⇒ A directed graph is disjoint union of S.C.C.

⇒ A S.C.C of a directed is a sub-graph such that

1. Given u and v in S.C.C, there must be a path from u to v and a path from v to u in the graph.

2. Given u in S.C.C and v not in S.C.C, then there should not be a path from u to v and v to u in the graph. There may be a path in one dir. but not in both dir.



S.C.C. are

(i) {0} (ii) {1,2,4} (iii) {3}

For single nodes, check that there is no outgoing edge

- **TRANSPOSE OF A GRAPH :-** // Just reverse the edges

Given $G(V, E)$

For every edge (u, v), add an edge (v, u) in E^T

Then $G^T(V, E^T)$ is called transpose of the graph.

→ Transpose can be found in $O(n+m)$ time

→ S.C.C. of G and G^T are same.

- **TO FIND NO. OF S.C.C. :-**

We must know both G and G^T

Step-I :- Apply DFS on G and find topological sort array $L[]$.

Step-II :- We will apply DFS on G^T

```
DFS visit (G^T)
{  for (i=0. i<n, i++)
       V[i] = false

   scc = 0 ;        // variable to count scc

   for (i=0 : i<n ; i++)
       if (L[i] == false)
}                  { scc++; DFS (G^T, L[i]) }
```

```
DFS (G^T, u)
{   V[u] = true
    SCC[u] = sec;
    for every edge (u,v) in G^T
    {   if (V[v] == false)
            { DFS (G^T, v) ; }
    }
}
```

CORRECTNESS :- We must proove both points in S.C.C.

2nd point :-

Strongly Connected Components

Among all the nodes that are not yet visited pick the node with maximum finish time and apply DFS on the Transpose of the graph.

1. Given u and v in SCC, there must be path from u to v and a path from v to u in the graph.
2. Given u in SCC and v not in SCC, there should not be be path from u to v and a path from v to u in the graph.

If I<J there can not be any edge form SCC(I) to SCC(J) in the G^T

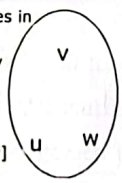
2nd pt proof

1st point →

Strongly Connected Components

Let u and w be two nodes in the same SCC.
Let this SCC obtained by applying DFS starting at the node v

F[v]>F[u] and F[v]>F[w]

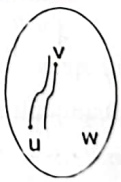


Strongly Connected Components

In G^T
path from v to u

So in G
path from u to v
And F[v]>F[u]

path from v to u



Strongly Connected Components

In G^T
path from v to w

So in G
path from w to v
And F[v]>F[w]

path from v to w

Strongly Connected Components

In G
path from u to v
path from v to w
So in G
path from u to w

Similarly
path from w to u

10th July

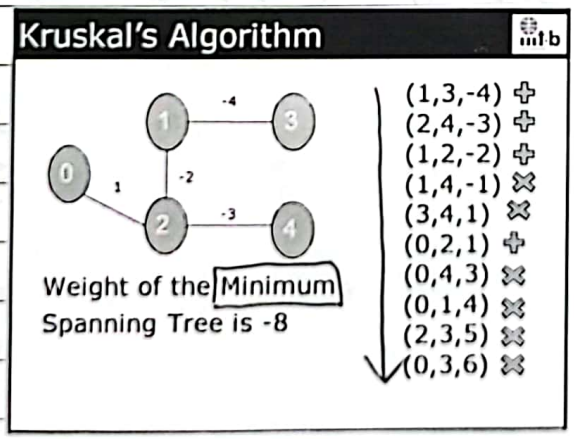
1 KRUSKAL'S ALGORITHM FOR MST :-

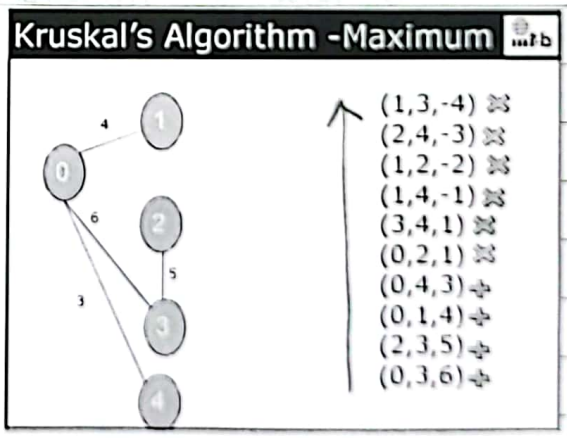
Min Spanning Tree :- Given a graph G.

Start with MST with only nodes (and no edges)

Consider the edges of G in the increasing order of weights (can be done by sorting the edgelist rep of G)

- If the edge goes across the c.c., merge the c.c.
- If the edge goes with in the c.c., ignore the edge.





Analysis :-

⇒ Sorting the edges based on their weights takes O(mlogn) time.

- How do you check if the given edge (u,v) goes across the c.c or goes within the c.c.

We can apply BFS/DFS from u on the MST and check if u is connected to v.

This makes krushkal algorithm O(mn) buts Prims Algo is O((n+m) logn)

2 Disjoint Union Data Structure :-

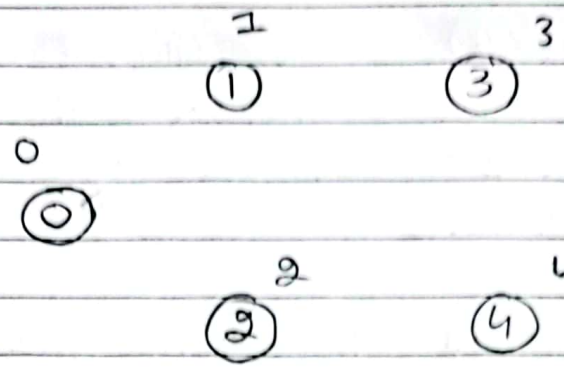
U have a universal set U = {0, 1, 2, ... , n-1}

→ Makeset → Make n sets each containing just one node (ie. singleton disjoint sets)

→ Disjoint (i,j) → We want to know that whether elements i & j belong to the same set or diff. set.

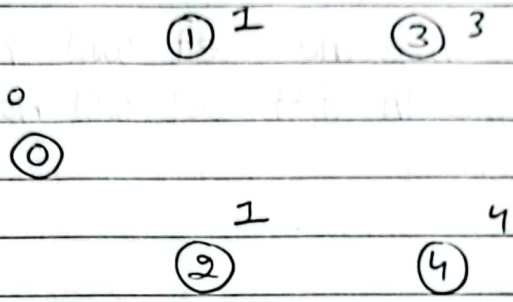
→ Union (i,j) → Replace the set containing i and set containing j with the union of these 2 sets.

Example :-

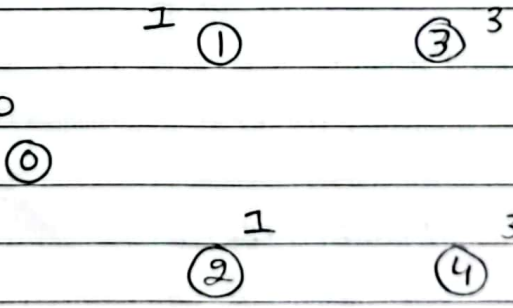


Initially all disjoint

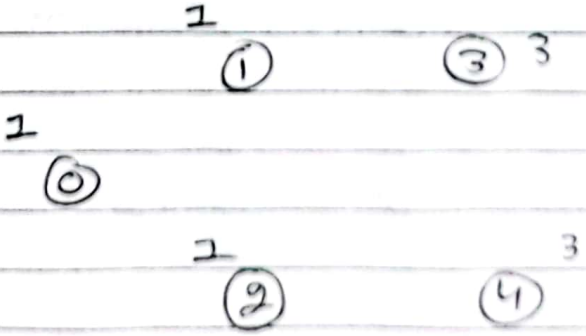
I Union (1,2) → Change colour of 2 to 1



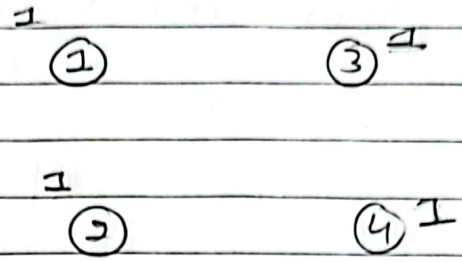
II Union (3,4) → Change colour of 4 to 3



III Union (0,2) → Change colour of 0 to colour of 2
(Because we have to do min)



IV Union(0,4) → We change the colour of any one set to the other.



⇒ Elements in same set will have same colour and elements in diff set will have diff. colours.

Disjoint Union

- C[i] – the color of the element set to which the element i belong to.
- S[C[i]] – number of elements in the set whose color is C[i]
- List[C[i]] – list of elements in the set whose color is C[i]

Make Set

```
For (i=0;i<n:++i)
   C[i]=i;
   List[i]=newnode(List[i],i);
   S[i]=1;
```

Disjoint Union

- Disjoint (i,j) check if C[i]=C[j]

Elements i and j belong to the same set iff C[i]=C[j]

Takes O(1)

Disjoint Union

- Union (i,j)

Take the smaller set and merge it with the bigger set.

```
if(S[C[i]]<S[C[j]])
   S[C[j]]+=S[C[i]];
   S[C[i]]=0;
```

Delete each element in the List[C[i]] and add it to the List[C[j]] and change the color of each element in C[i] to C[j]

Disjoint Union

- Union (i,j)

Take the smaller set and merge it with the bigger set.

S[C[i]]=S[C[j]]=n/2;

One union can take O(n) time.

Disjoint Union

- Union (i,j)

Take the smaller set and merge it with the bigger set.

We shall show that n union takes O(n log n) time.

Each element can change its color at most O(log n) times.

Disjoint Union

Theorem: If an element has changed its color k times then the size of the set to which it belongs should be $\geq 2^k$.

$S[C[i]] \geq 2^k$.

Proof by Induction : k=0 $S[C[i]] \geq 2^{0-1}$.

Disjoint Union

Theorem: If an element has changed its color k times then the size of the set to which it belongs should be $\geq 2^k$.

When an element changes its color kth time. It has already changed its color k-1 times by induction.

$S[C[i]] \geq 2^{k-1}$.

It will be merged with a bigger set.

So after the kth color change

$S[C[i]] \geq 2^{k-1} + 2^{k-1} \geq 2^k$

Disjoint Union

Theorem: If an element has changed its color k times then the size of the set to which it belongs should be $\geq 2^k$.

Size of the Universal Set is n.

So any element can change its color at most log n times.

n union can take at most O(n log n) time

Disjoint Union –Next Class

MakeSet – O(n) time.

Disjoint (i,j) – O(1) Amortized

Union(i,j) – O(1)

Kruskal makes m calls to Disjoint and n-1 calls to Union would take O(n+m) time.

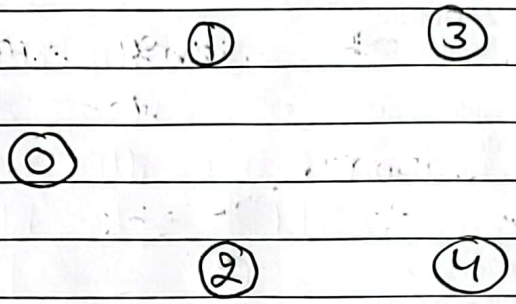
So Complexity of Kruskal is O(mlog n+ m+n) which is O(m log n)

3 This class

Disjoint (i, j) ⇒ O(logn) ⇒ O(1) (Amortized)
Union (i, j) ⇒ O(1)

Now, we will apply the previous concept differently.

Let P[i] be denoting parent of node i.

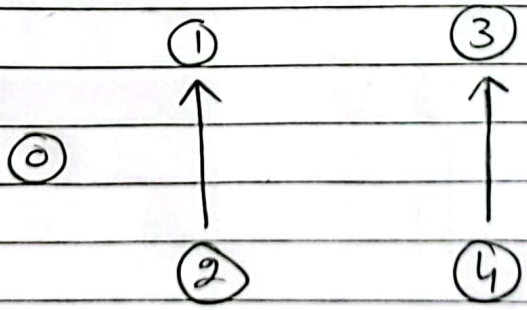


Initially P[i] = i, Each set is represented by a tree.

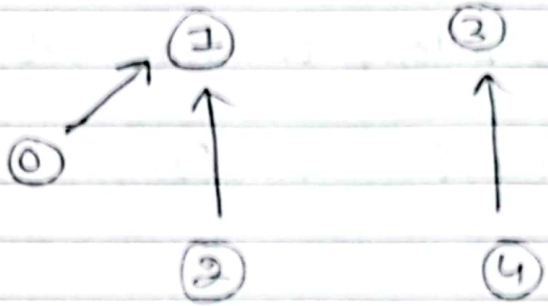
I. Union (1,2) & Union (3,4)

↓ 2 becomes child of 1

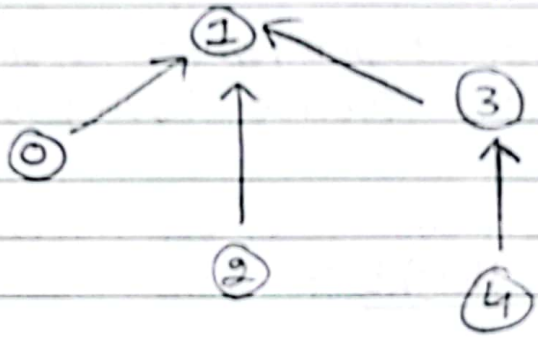
↓ 4 becomes child of 3



II Union(0,2) → 0 becomes child of 1.



III Union (4, 2) → Subtree with fewer no of elements become child of other tree.



⇒ Elements in the same set should return the same parent (root) of that tree.

We define a function Find(i) which returns the root of the tree to which node i belongs to.

```
Find(i) {

    if (P[i] == i)  return i;

    else return Find(P[i]);

}
```

- Disjoint (i, j)

⇒ i and j belong to some set iff

$$\text{Find}(i) = \text{Find}(j)$$

- Union (i, j)

⇒ Union By Rank :-

- Subtree with fewer no. of elements become the child of a subtree having more no. of elements.

⇒ Union By Height :-

Subtree with smaller height become the child of a subtree with longer height.

⇒ Here the assumption is that you call union only after you call disjoint and if they are in 2 diff. sets.

Suppose the given task is Union(a, b)

⇒ a = Find(i)
b = Find(j)

if (a != b) Union(a, b)

⇒ We don't change Union function for the disjoint function here.

⇒ Each Union is O(1)

Each disjoint is 2Find()

Each find is O(h), h is the height of the tree

We can think of colour of a node as the root of the subtree.

→ No node can change its colour more than (log n) no. of times. That means height of tree is O(log n). [See th^m $S[c[i]] \geq 2^k$]

This is in case of Union By Rank

In case of Union by Height :-

Thm → A tree of height h will have $\geq 2^h$ elements

Proof :- By induction, if $h = 0$, then tree has one node.

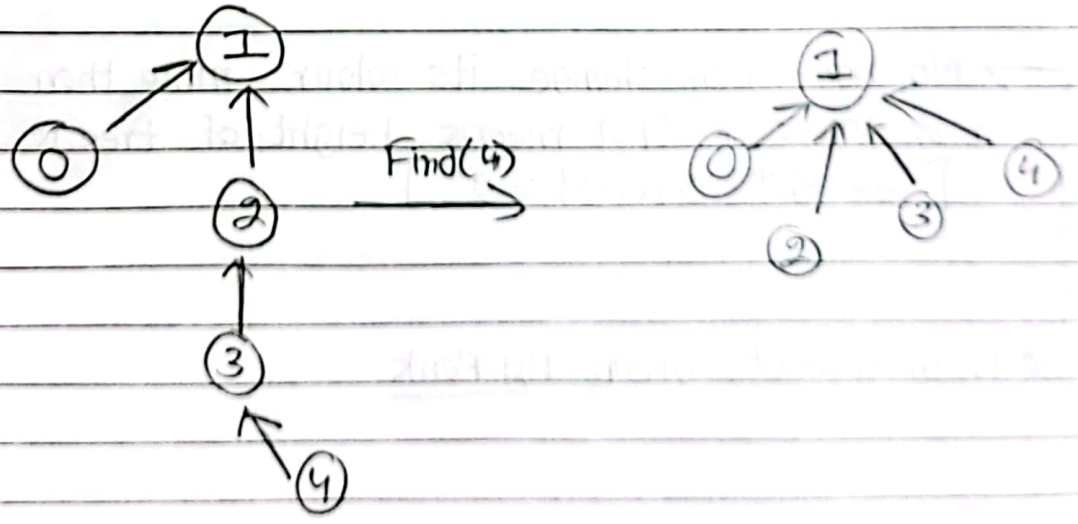
⇒ Height of a tree increases only when the height of both the trees to be merged has the same height.

$\geq 2^{h-1} + 2^{h-1} \geq 2^h$ elements

→ $h \leq \log n$

- PATH COMPRESSION :-

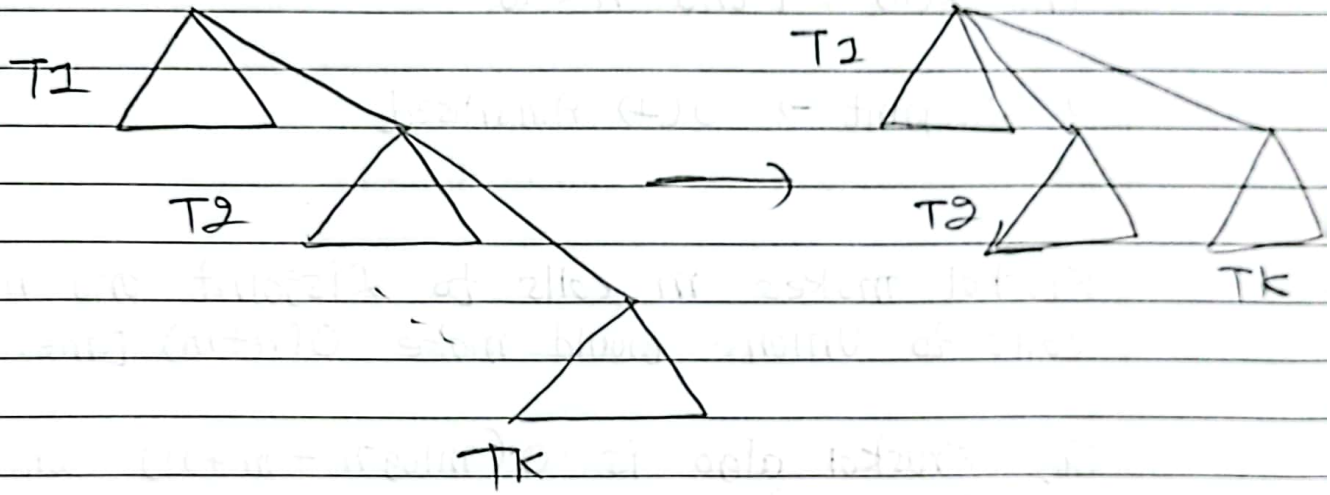
When we call find(), we do more than just simply returning the root.



Suppose we called Find(4).

So, the process is that when u troverse the path from 4 to 1, change the parent pointer of all the nodes we are passing through (2 & 3) to 1.

In Future, if we have to find if 3 & 4 belong to same set, it will be easy to answer in future.




```
Find(i)
{   if(P[i]==i) return i;
    else return P[i] = Find(P[i]);
}
```

So,

Make set → $O(n)$ time

Disjoint → $O(\alpha(n))$ Amortised

Union → $O(1)$

Where $\alpha(n)$ is the inverse Ackermann function and $\alpha(n) < 4$ and $n < 10^{600}$

⇒ Disjoint → $O(1)$ Amortised.

Kruskal makes m calls to Disjoint and $n-1$ calls to Union would make $O(n+m)$ time.

So, Kruskal algo is $O(m \log n + m + n)$ which is $O(m \log n)$

13th July

- **UPDATING MST :-**

How we can efficiently update min & max spanning tree if the graph is dynamic.

I Let $G(V, E, w)$ and $G'(V, E, w')$ be two graphs such that $w'(u,v) = w(u,v) + k$ ($k \rightarrow$ constant $\in$ I) for every edges in the graph.

If ST is any spanning tree, the no. of edges in ST is $n-1$.

$$\sum w'(u,v) = \sum (w(u,v) + k)$$

$$= k(n-1) + \sum w(u,v)$$

$\Rightarrow$ MST of G is also a MST of G'

II $w'(u,v) = k * w(u,v)$ ($k \rightarrow$ constant (+ve))

$$\sum w'(u,v) = \sum (k * w(u,v))$$

$$= k * \sum (w(u,v))$$

$\Rightarrow$ MST of G is also MST of G'

Note :- In II part, if K = -ve constant, then max ST becomes min ST and viceversa.

In II part, shortest path also doesn't change in new graph unlike part I (as we seen time before).

⇒ Following operations will be performed on min Spanning tree.

Similar could be done for max.

Deleting an edge

Let (u,v) is Deleted from the graph.

If(u,v) is not an edge of MST, do nothing.

If(u,v) is an edge of the MST, delete the edge from the MST, it will create two connected components , among all the edges going across , pick an edge of minimum weight and add it the MST.

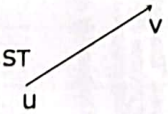
BFS(s)

BFS on u

```
for (i=0;i<n:++i)
V[i]=false;

Enqueue(Q,s);
V[s]=true;
```

BFS(u)

```
While(Q){
u=Dequeue(Q);
For every edge (u,v)in the MST
if (V[v]==false){
    Enqueue(Q,v);
        V[v]=true;}
}}
```



Deleting edge

```
BFS(u);
Min=INT_MAX;

For every edge (u,v) in the graph
If(V[v]!=V[u] && w(u,v) <min)
{min=w(u,v); a=u;b=v;}

Add the edge (a,b) to the MST
O(E) time.
```

Adding an edge

Let (u,v) is Added to the graph.

Add(u,v) is an edge of the MST, it will create a cycle; delete the maximum weighted edge in the cycle.

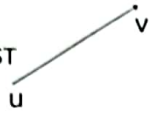
BFS on u

BFS(s)

```
for (i=0;i<n; ++i){
V[i]=false;
M[i]=INT_MIN;
}
Enqueue(Q,s);
V[s]=true;

//M[i]= maximum weighted edge reachable from s;
```

BFS(u)

```
While(Q){
u=Dequeue(Q);
For every edge (u,v)in the MST
if (V[v]==false){
    Enqueue(Q,v);
        V[v]=true;
If(M[u]<w(u,v)) {M[v]=w(u,v);A[v]=v;B[v]=u;}
else {M[v]=M[u];A[v]=A[u];B[v]=B[u];}
}}}
```



Adding edge

Add the edge (u,v) to the MST.
BFS(u);

Delete the edge (A[v],B[v]) from the MST

Note that w(A[v],B[b])=M[v];

O(V);

Updating edge

Update(u,v,w) -set w(u,v) = w

If Increase Key w>w(u,v)
-Similar to Deleting the key -O(E)

If Decrease Key w<w(u,v)
-Similar to Adding the key -O(V)

If (u,v) is an edge in MST
Delete the key (edge)

else do nothing (just update the graph)

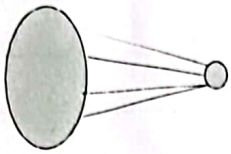
If (u,v) is not an edge in MST, then Add operation in MST

else do nothing

Adding a node

G(V,E,w) be the graph and MST be its Minimum Spanning Tree.

A new node and k weighted edges incident on this node are added to the graph.



Adding a node

G(V,E,w) be the graph and MST be its Minimum Spanning Tree.

A new node and k weighted edges incident on this node are added to the graph.

- Add k edges using the previous algorithm. O(kn) time

Adding a node

G(V,E,w) be the graph and MST be its Minimum Spanning Tree.

A new node and k weighted edges incident on this node are added to the graph.

- Add k edges using the previous algorithm. O(kn) time
- Add k edges to the MST to compute a Graph G' and apply Kruskal's algorithm to find MST.This takes O(n log n) time.