KTU NOTES

The learning companion.

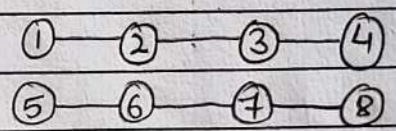
KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Module 2 → Advanced Data structures and Graph Algorithms: Self Balancing Trees → AVL Trees (Insertion and deletion operations with all rotations in detail, algorithms not expected): Disjoint Sets - Disjoint Set operations, Union and find algorithms.
DFS and BFS Traversals - Analysis, Strongly Connected Components of a directed graph, Topological sorting.

→ **Disjoint Set**: The disjoint set data structure is also known as Union-find data structure and merge find set. It is a data structure that contains a collection of disjoint or non-overlapping sets. The disjoint set means that when the set is partitioned into the disjoint subsets. The various operations can be performed on the disjoint subsets. In this, we can add new sets, we can merge new sets, and we can also find the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently.
The disjoint set can be defined as the subsets where there is no common element between the two sets.

Eg $S_1 = \{1, 2, 3, 4\}$
$S_2 = \{5, 6, 7, 8\}$

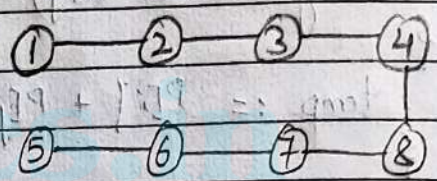


We have two subsets named S1 and S_2.
Since there is no common element between these two sets, we will not get anything if we consider the intersection between these two sets.
We can perform operations in such a case are: find & Union.

In case of find operation, we have to check that the element is present in which set.
Suppose we want to perform the Union operation on these two sets. First we have to check whether the elements on which we are performing the Union operation belong to different or same sets. If they belong to a different set, then we can perform the Union operation, otherwise not.

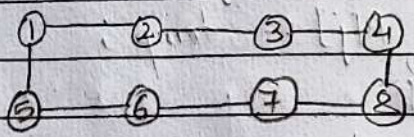
Eg. We want to perform Union b/w 4 and 8. Since 4 and 8 belongs to different sets, we can apply the Union operation. Once the Union is performed, the edge will be added b/w 4 and 8 as:

$S_1 \cup S_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}$



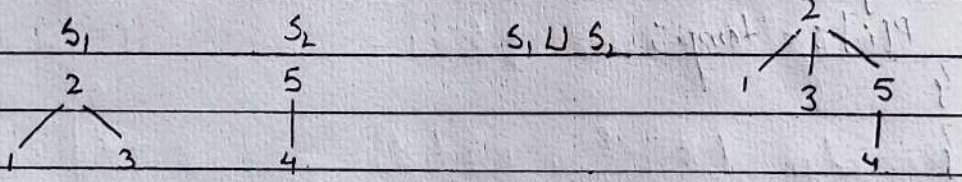
Suppose we add one more edge b/w 1 and 5. Then:

$S_3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$



Various operation on Disjoint Set: Find and Union.

Example:



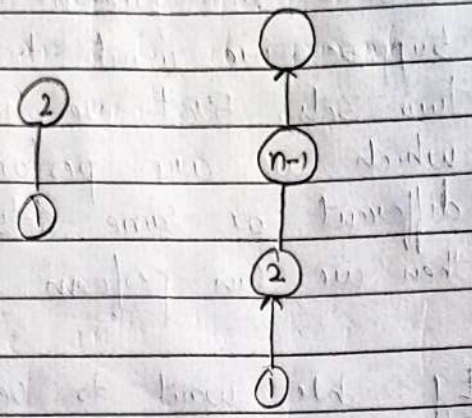
find (3) means in which set the element 3 is there i.e. 3 is in S_1

find (5) means in which set the element 5 is there i.e. 5 is in S_2.

Union Algorithm :

* Algorithm SimpleUnion (i, j)

```
{
  P[i] := j;
}
```



Time Complexity for 1 Union = 1
Time Complexity for n-1 Union = n-1

Tall Union = O(n).

* WeightedUnion (i, j)

```
{
  temp := P[i] + P[j];        // P[i] := - count (i)
{
  P[i] := j;
  P[j] := temp;
}
  else
  {
  P[j] := i;
  P[i] := temp;
  }
}
```

- Simple Union leads to high time complexity in some cases.
- Weighted Union is a modified union algorithm with weighting rule.
- Weighted union deals with making the smaller tree a subtree of the larger.
- Count of nodes can be placed as a negative number in the P[i] value of the root i.

FIND Algorithm

* Algorithm SimpleFind (i)

```
{
  while (P[i] >= 0) do
{
   i := P[i];
}
   return I;
}
```

Time Complexity for 1 find = i

Time complexity for n find = $1+2+\cdots+n-1+n = n(n+1)/2$.

Tall finds = $O(n^2)$.

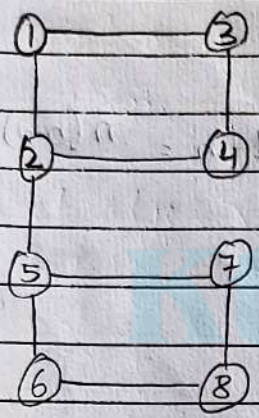
* Algorithm CollapsingFind (i)

```
{
   r := i;
   while (P[r] > 0) do
{
   r := P[i];
}
   while (i != r) do
{
   j := P[i];
   P[i] := r;
   i := j;
}
   return r;
}
```

- SimpleFind() leads to high time complexity in some cases.
- CollapsingFind() is a modified version of SimpleFind()
- If j is a node on the path from i to its parent and p[i] != root, Then set j to root.

Detecting Cycles in a Graph.

Consider an example to detect a cycle with the help of disjoint sets.



$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

Each vertex is labelled with some weight. There is a universal set with 8 vertices.

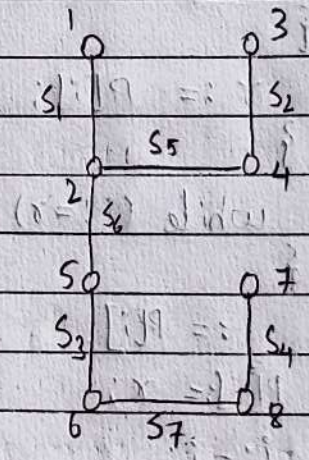
We will consider each edge one by one and form the sets.

Take edge 1,2 :- $S_1 = \{1, 2\}$

Take edge 3,4 :- $S_2 = \{3, 4\}$

Take edge 5,6 :- $S_3 = \{5, 6\}$

Take edge 7,8 :- $S_4 = \{7, 8\}$



Take edge 2,7 :- $S_5 = S_1 \cup S_2$

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

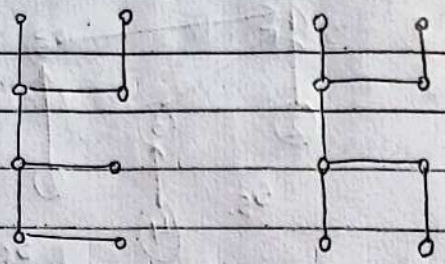
Take edge 2,5 :- $S_6 = S_5 \cup S_3$

$= \{1, 2, 3, 4, 5, 6\}$

Take edge 6,8 :- $S_7 = S_6 \cup S_4$

$= \{1, 2, 3, 4, 5, 6, 7, 8\}$

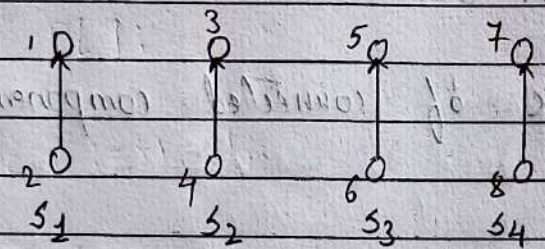
Take edge 5,7 :- We cannot cannot 5,7 as these belongs to same set S_7. Hence, no connection is possible.



Graphical Representation of disjoint Sets :

$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

We have universal set U and we will consider each edge one by one to represent set graphically.

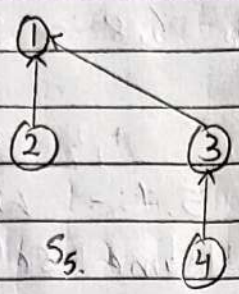


$S_1 = \{1, 2\}$, $S_2 = \{3, 4\}$, $S_3 = \{5, 6\}$, $S_4 = \{7, 8\}$.

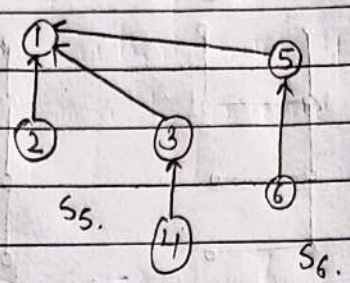
Now, we can consider the edge (2,4). Since 2 and 4 belongs to different sets, we need to perform the union operation. When we perform the union operation on two sets, i.e. S_1 and S_2, then 1 vertex would be the parent of vertex 3

$S_5 = \{2, 4\}$

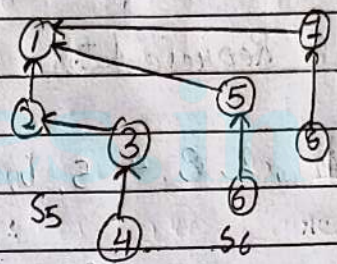
Next vertex : $S_6 = \{2, 5\}$
Parent of 2 is 1
Parent of 5 is 5.



Next edge: $S_7 = \{1, 3\}$
Parent of 2 is 1
Parent of 5 is 5.
Same parent, No need to perform any union operation.



Next edge: $S_8 = \{6, 8\}$
Parent of 6 is 5
Parent of 8 is 7.



⇒. Applications of Disjoint Set.

i) Used to keep track of connected component in an undirected graph.

ii) Used to detect cycles in a graph.

iii) Used to calculate the minimum spanning tree in Kruskal's algorithm.

iv) Used in maze generation problems.

Array Representation of Disjoint Set.

-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

Consider edge (1, 2) as 1 is the parent of itself Similarly 2 is the parent of itself. So we make vertex 2 is the ~~parent~~ child of 1. We add -2 at the index 1 where '-' indicates the vertex 1 is the parent of itself and 2 represents the no. of vertices in a set.

-2	1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

Next edge is 3, 4.

-2	1	-2	3	-1	-1	-1	-1
1	2	3	4	5	6	7	8

Next edge is 5, 6

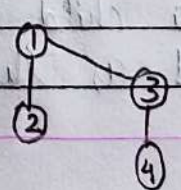
-2	1	-2	3	-2	5	-1	-1
1	2	3	4	5	6	7	8

Next edge is 7, 8.

-2	1	-2	3	-2	5	-2	7
1	2	3	4	5	6	7	8

Next edge is 2, 4

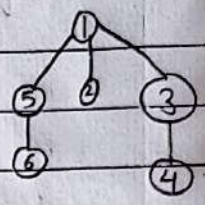
-4	1	1	3	-2	5	-2	7
1	2	3	4	5	6	7	8



Here we add -4 as it contains 4 vertices.

Next edge is 2, 5.
1 is the parent of 2 and 1 is the parent of itself. 5 is the parent of itself (-2). Now, we have to decide whether the vertex 1 or vertex 5 would become a parent. Since the weight of vertex 1 is -4, greater than vertex 5 i.e. -2. So when we apply the union operation, the vertex 5 would become a child of vertex 1.

-6	1	1	3	1	5	-2	7
1	2	3	4	5	6	7	8

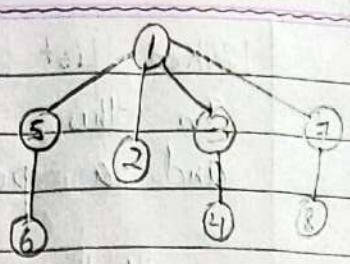


Next edge is 1, 3.
Parent of 1 is 1 and parent of 3 is 1.
∴ The parent of both the vertices are same, so we can say that there is a formation of cycle if we include the edge (1, 3).

Next edge is 6, 8.
Vertex 5 is the parent of vertex 6
vertex 1 is the parent of vertex 5.
Vertex 7 is the parent of vertex 8
vertex 7 is the parent of itself.

Height of vertex 1 (-6) is greater that vertex 7 (-2), we can make vertex 7 as child of vertex 1
We add 1 at the index 7 because 7 becomes a child of the vertex 1. We add -8 at the index 1 as the weight of the graph becomes 8.

-8	1	1	3	1	5	1	7
1	2	3	4	5	6	7	8.



Next edge is 5, 7

But parent of 5 and 7 is 1.

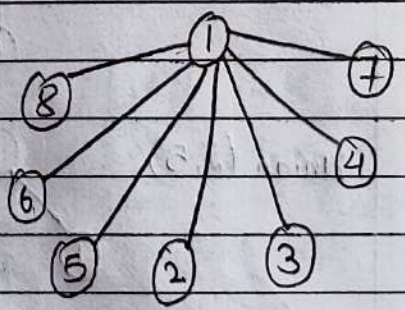
So it's not possible to make an edge.

Disadvantage of this is that : some nodes take more time to find its parent.

Collapsing find: In the above graph, if we want to find the parent of vertex 6, vertex 5 is the parent of vertex 6 so we moved to the vertex 5 and vertex 1 is the parent of vertex 5. To overcome this problem, we use collapsing find.

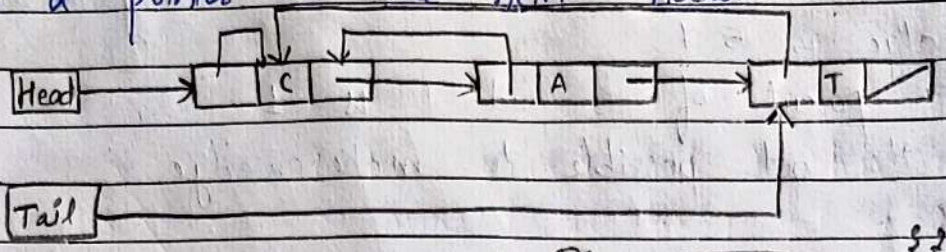
In this, once we know the parent of the vertex 6 which is 1 then we directly add the vertex 6 to the vertex 1. We will also update the array. In array, add 1 at the index 6 because parent of 6 is now 1. The process of directly linking a node to the direct parent of a set is klas collapsing find. Similarly, we can link 8 and 4 to 1

-8	1	1	3	1	1	1	7
1	2	3	4	5	6	7	8



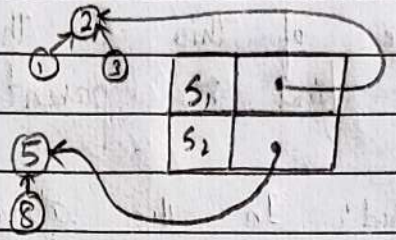
Linked List Representation of Disjoint Set:

In this, each node has a pointer to the head node and a pointer to the next node.



Eg: $S_1 = \{1, 2, 3\}$

$S_2 = \{5, 8\}$



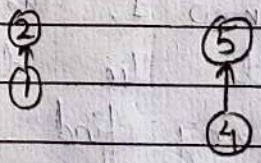
~~2 & 5~~ are the parents, we can choose any.

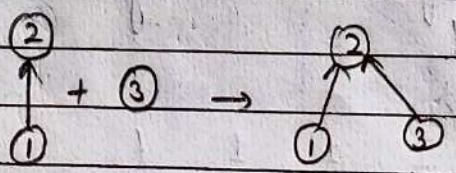
Makeset: The disjoint set also supports one other important operation called Makeset which creates a set containing only a given element in it. It is a procedure to form the set {x}.

Time complexity is O(1)

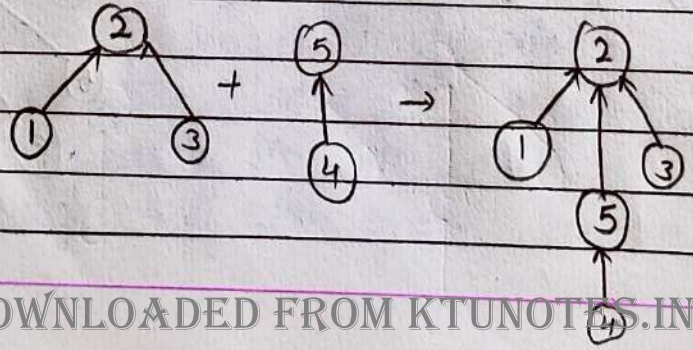
i) makeset(x) → makeset(1), makeset(2), makeset(3), makeset(4), makeset(5).

ii) Union(x, y) → union(1,2)
union(4,5)
union(1,3)





Union(2,5)



Improve Performance : Disjoint sets can be optimized by:

- Union by Rank
- Path by compression.

Union by Rank: Uses Find to determine the roots of the trees x and y belong to. If the roots are distinct, the trees are combined by attaching the root of one to the root of the other. If this is done naively, such as by always making x a child of y, the height of the trees can grow as O(n). We can optimize it by using union by rank.

Union by rank always attaches the shorter tree to the root of the taller tree.
To implement union by rank, each element is associated with a rank. Initially a set has one element and a rank of zero.
If we union two sets and:

- Both trees have the same rank. the resulting set's rank is one larger.
- Both trees have the different ranks - the resulting set's rank is the larger of the two. Ranks are used instead of height or depth because path compression will change the tree's heights over time

Worst case complexity : O(Logn)

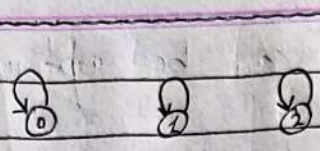
$S_1 \rightarrow i^{rank}$

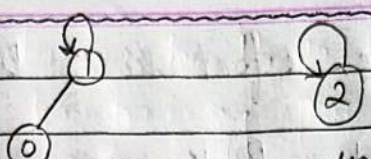
$S_2 \rightarrow j$

$i > j \Rightarrow Parent(i) \leftarrow j$ | diff. rank

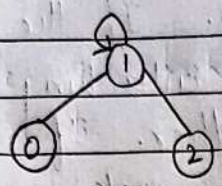
$i < j \Rightarrow parent(j) \leftarrow i$ | don't increment

$i = j \Rightarrow parent(i) \leftarrow j$, $parent(j) \leftarrow i$ | same rank then increment rank.





Union (1, 2)

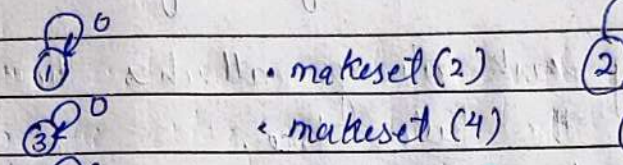


1 has rank 1 and 2 has rank 0 so make 1 as the parent of 2.

Eg $U = \{1, 2, 3, 4, 5\}$

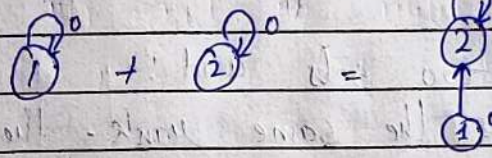
i) Makeset (x)

- Makeset (1)
- Makeset (3)
- makeset (5)
- makeset (2)
- makeset (4)

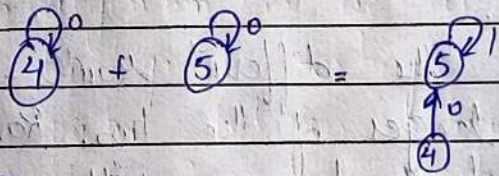


ii) Union (x, y)

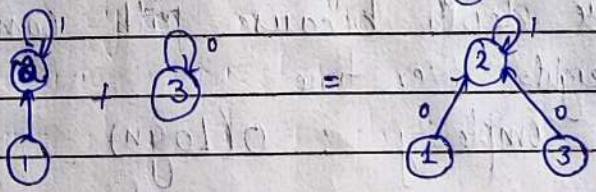
Union (1, 2)



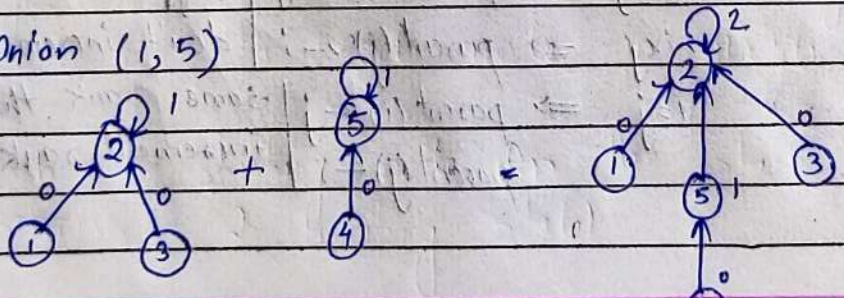
union (4, 5)



union (1, 3)

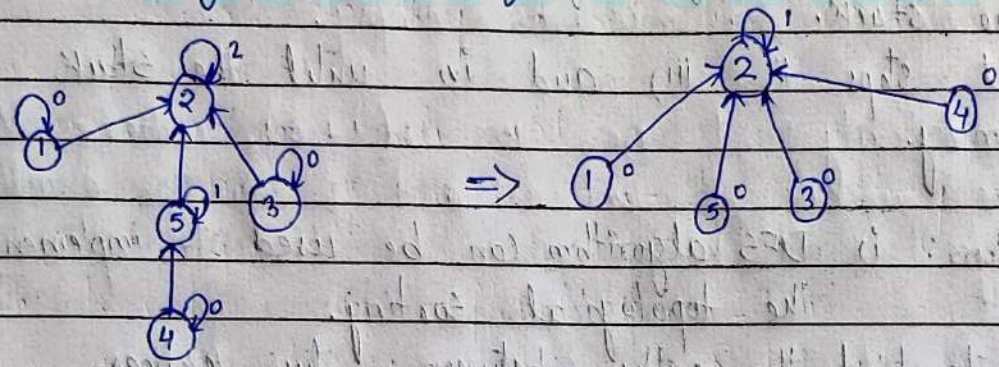


i) Union (1, 5)



Path Compression (collapsing rule) : Path compression is a way of flattening the structure of the tree whenever Find is used on it. Since each element visited on the way to a root is part of the same set, all of these visited elements can be reattached directly to the root. The resulting tree is much flatter, speeding up future operations not only on these elements, but also on those referencing them.

- If j is a node in the path from i to its root, and parent of i != root of i then set parent of i to root of i
- It's a method to reduce the height of a tree and make every operation faster.



MakeSet(x)	Find Set(x)

function Makeset(x)
{
if x is not parent,
then add x to disjoint set tree
x.parent = x
x.rank = 0
}
	function findset(x)
	{
	if x.parent != x
	x.parent = findset(x.parent)
	return x.parent
	}

⇒ DFS: stands for Depth First Search. It is a recursive algorithm that uses the backtracking principle.
The algorithm starts at the root node and examines each branch as far as possible before backtracking.
The step by step process to implement the DFS traversal:

i) Create a stack with the total number of vertices in the graph.
ii) Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
iii) After that, push a non-visited vertex to the top of the stack.
iv) Now, repeat steps iii) and iv) until no vertices are left to visit from the vertex on the stack's top.
v) If no vertex is left, go back and pop a vertex from the stack.
vi) Repeat steps ii), iii) and iv) until the stack is empty.

Applications: i) DFS algorithm can be used to implement the topological sorting.
ii) Used to find the paths between two vertices.
iii) Used to detect cycles in the graph.
iv) Used for one solution puzzles
v) Used to determine if a graph is bi-partite or not.

Algorithm: I. Set STATUS = 1 (ready state) for each node in Graph

II. Push the starting node on the stack and set its STATUS = 2 (waiting state)

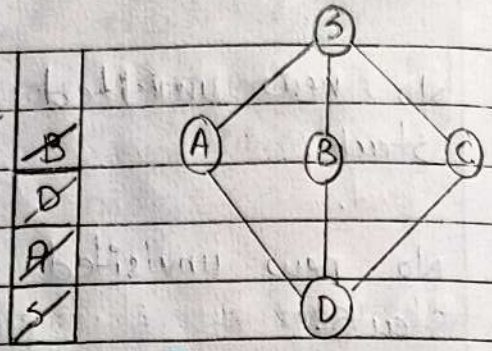
III. Repeat IV and V until Stack is empty.

IV. Pop the top node N. Process it and set its STATUS = 3 (processed state)

V. Push on the stack all the neighbors of node that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

VI. EXIT.



Eg i) Initialise the stack

ii) Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S.

iii) Now, we have three nodes and we can pick any of them. We are taking A. Mark A as visited and put it onto the stack. Explore any unvisited node from A.

iv) Visit D and mark it as visited and put onto the stack.

v) Visit B and mark it as visited and put onto the stack.

vi) B does not have any unvisited adjacent node. So, pop B from the stack.

vii) There is new unvisited node from D. Visit C and mark it as visited and push it onto the stack.

viii) No new unvisited node from C. pop C from the stack.

ix) No new unvisited node from D. pop D from the stack.

x) No new unvisited node from A. pop A from the stack.

xi) No new unvisited node from S. pop S from the stack.

xi) Now, stack has become empty.

Time complexity of DFS is $O(V+E)$, where V is the number of vertices and E is the number of edges in the graph.

Space complexity is $O(V)$.

Advantages: i) Consumes very less memory space.
ii) Reaches at the goal node in less time period than BFS if it traverses in a right path.
iii) It may find a solution without examining much of search because we may get the desired solution in the very first go.

Disadvantages: i) It is possible that may states keep re-occuring. There is no guarantee of finding the goal node.

ii) Sometimes the states may also enter into infinite loops.

⇒. BFS: Stands for Breadth First Search. It is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes.

BFS puts every vertex of the graph into two categories- visited and unvisited nodes. It selects a single node in a graph and after that, visits all the nodes adjacent to the selected node. It uses a queue.

Applications: i) BFS can be used to find the neighbouring locations from a given source location.

ii) Used to determine the shortest path and minimum spanning tree.

iii) Used in web crawlers to create web page indexes.

Algorithm: I. Set STATUS = 1 (ready status) for each node in Graph.

II. Enqueue the starting node A and set its STATUS = 2 (waiting status)

III. Repeat IV and V until QUEUE is empty.

IV. Dequeue a node N. Process it and set its STATUS = 3 (processed state).

V. Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state).

[END OF LOOP]

VI. Exit.

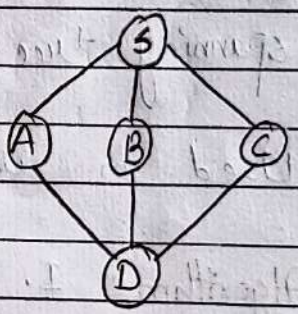
The step by step process to implement the BFS:

i) Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

ii) If no adjacent vertex is found, remove the first vertex from the queue.

iii) Repeat i) and ii) until the queue is empty.

Eg i) Initialize the queue.



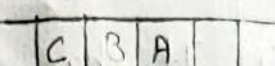
ii) We start from visiting S and mark it as visited.

iii) We can see unvisited nodes from S. We have three nodes from. We will choose A. mark it as visited and enqueue it in queue.

A				

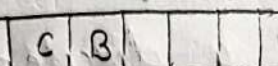
iv) Next, the unvisited node from S is B. Mark it as visited and enqueue it.

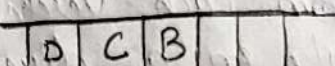
v) Unvisited node from S is C. Visit C and enqueue it.



vi) Now, S is left with no unvisited adjacent nodes. So, dequeue A and find nodes adjacent to A.

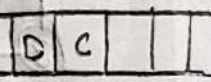
vii) From A we have D as unvisited adjacent node. Mark it as visited and enqueue it.



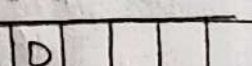


viii) At this stage, we are left with no unmarked nodes. But as per the algorithm dequeue all the nodes.

ix) No new node from B so dequeue B.



x) No new node from C so dequeue C.



xi) No new node from D. So dequeue D.

xii) Dequeue is empty.

xiii) Stop.

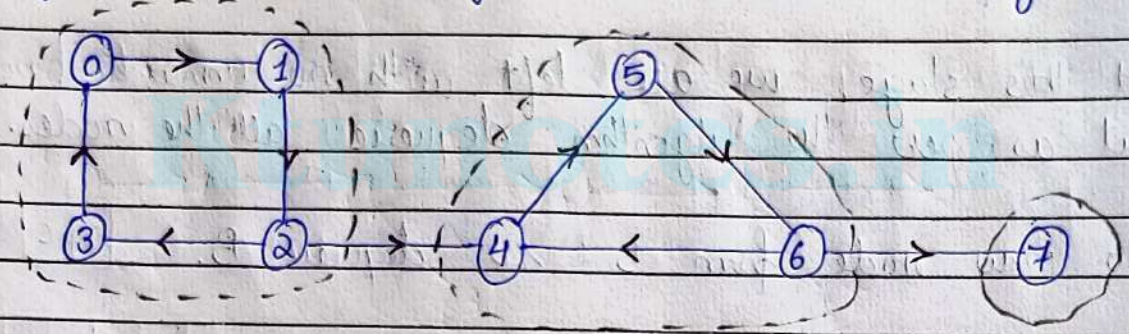
Advantages: i) In this procedure at any way it will find the goal.

ii) It does not follow a single unfruitful path for a long time. It finds the minimal solution in case of multiple paths.

Disadvantages : i) Consumes large memory space. Time complexity is more.

ii) It has long pathways, when all paths to a destination are on approximately the same search depth.

⇒ **Strongly Connected Graph :** A strongly connected component is the portion of a directed graph in which there is a path from each vertex to another vertex. It is applicable only on a directed graph. Single node is always a SCC.



In this, every vertex can reach the other vertex through the directed path.

Kosaraju's Algorithm : aims to find all strongly connected components of a given input graph. It uses DFS. It is used for SCC. It is a directed graph. It is a 2 pass algorithm

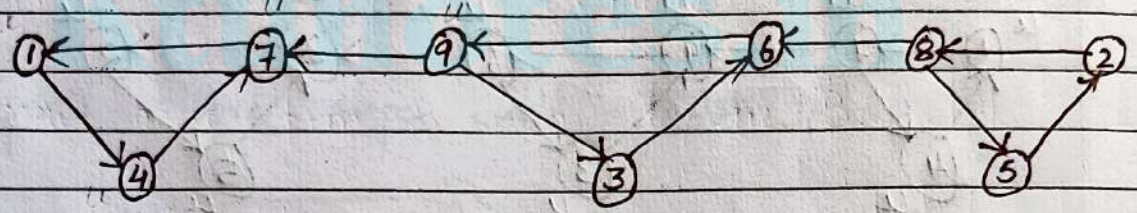
Pass 1 : i) Set all the vertices of graph G as invisible.

ii) Create an empty stack.

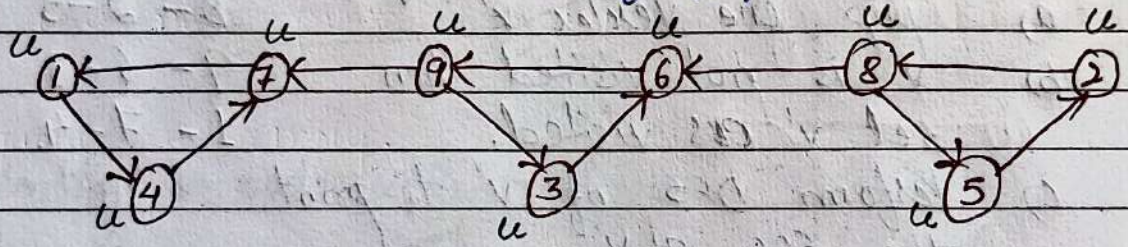
iii) Perform DFS traversal:
- a) on unvisited vertices.
- b) Set it as visited.
- c) Push in stack if the vertex has no unvisited neighbour (until all vertices are visited).

Pass 2: i) Reverse the graph G (reverse the direction).
ii) set all nodes as unvisited.
iii) While stack is not empty:
- a) Pop one vertex 'v'
- b) 'v' is not visited then set 'v' as visited.
- c) Perform DFS of 'v' to print the SCC of 'v'.

Q Find the strongly connected components.



Pass 1: i) Sett all the vertices of graph as unvisible



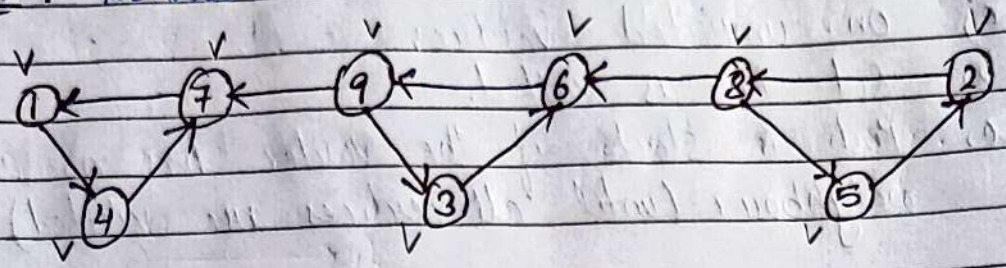
ii) Create an empty stack.

iii) Perform DFS traversal.
- on unvisited vertices
- set it as visited.
- Push in stack if the vertex has no unvisited neighbour.

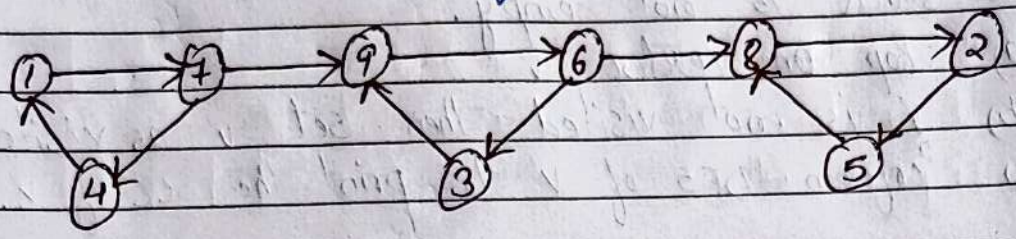
Pop 8 ←
pop
pop
9 ← Pop
Pop
pop
pop 1 ←
Pop
pop.

8
5
2
9
3
6
1
4
7

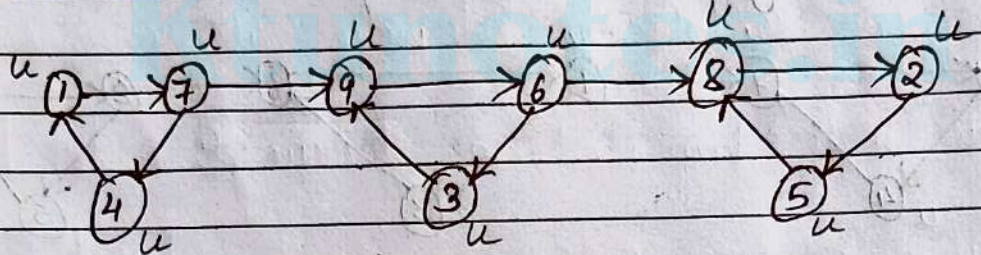
Pass 2: ~~Reverse~~



Pass 2: i) Reverse the graph G.



ii) Set all nodes as unvisited



iii) while stack is not empty:
a) pop one vertex v'
b) v' is not visited then set v' as visited.
c) Perform DFS of v' to print the SCC of v'.

8-2-5
9-6-3
1-7-4.

Applications: i) It has strongly connected component.

ii) It is used to find groups in social network. and to provide suggestions.

⇒. **Topological Sorting :** The topological sorting for a directed acyclic graph (DAG) is the linear ordering of vertices. For every edge U-V of a directed graph, the vertex u will come before vertex v in the ordering. After completing all nodes, we can simply display them from the stack.

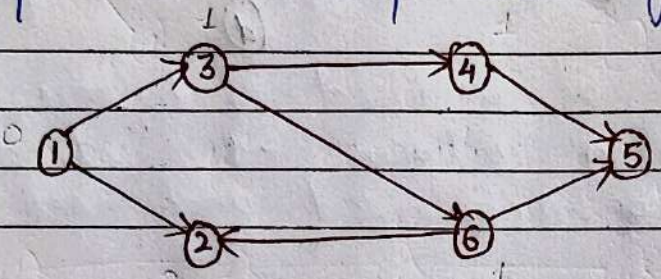
Algorithm : i) Identify a node with in-degree (no. of incoming edge)

ii) Add this node to the order.

iii) Remove this node and all its outgoing edges from the graph.

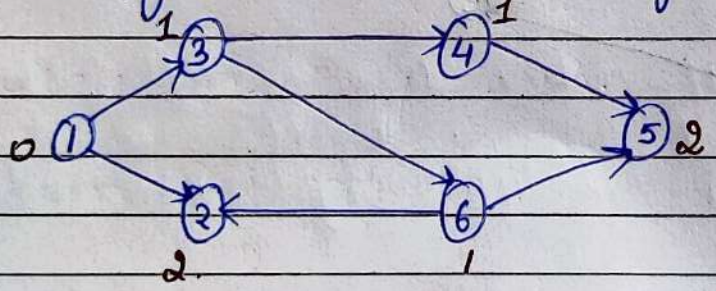
iv) Repeat above steps until graph is empty.

Eg1.



Topological Ordering
1 - 3 - 4
1 - 3 - 6
1 - 3 - 4 - 6 - 2 - 5
1 - 3 - 6 - 4 - 2 - 5
1 - 3 - 6 - 2 - 4 - 5.

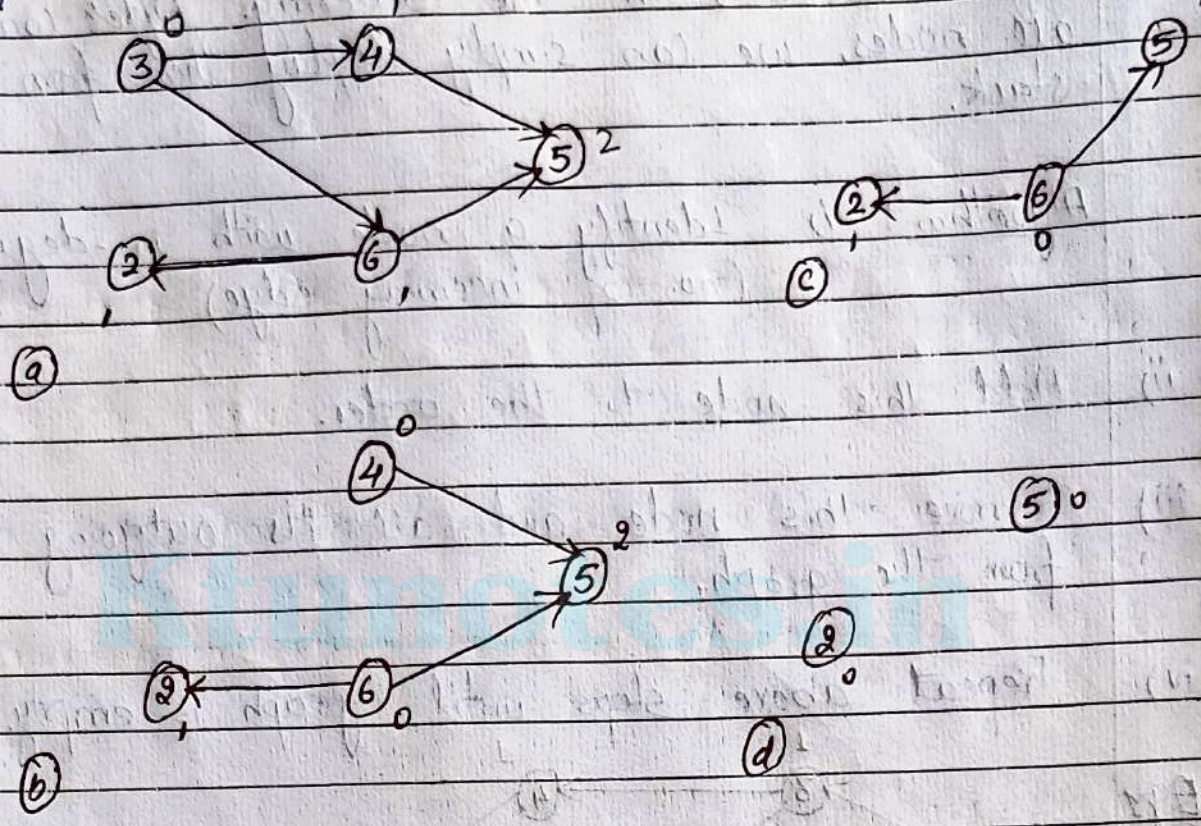
i) Identify a node with in-degree



ii) Add this node to the order

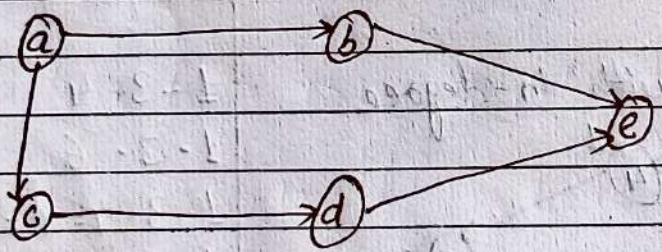
iii) Remove this node and all its outgoing edges from the graph.

iv) Repeat above steps until graph is empty.



Time Complexity :

2.



Applications: i) Finding cycle in a graph.
ii) Operation system deadlock detection.
iii) Dependency Resolution
iv) Sentence ordering
v) Critical path Analysis
vi) Job Scheduling
vii) Data Serialization.

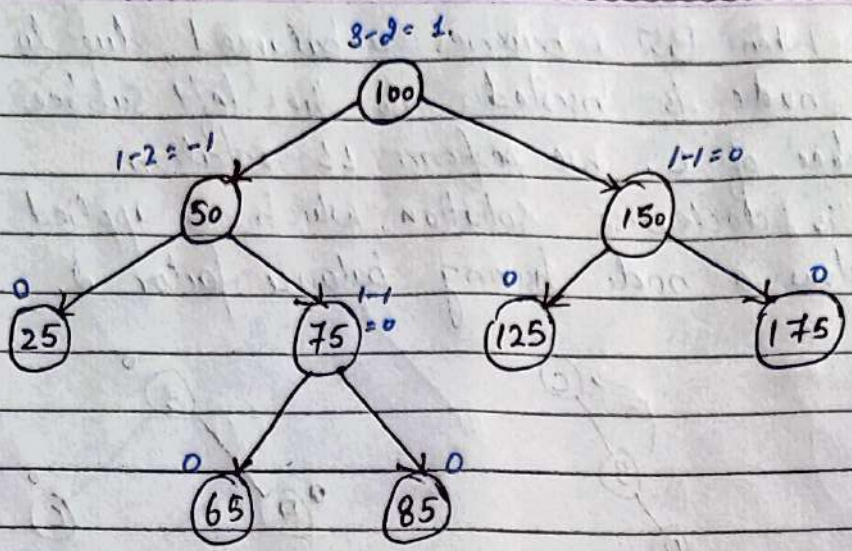
⟹ AVL Trees: AVL Tree is invented by GM Adelson, Velsky and EM Landis in 1962. The tree is named AVL in honour of its Inventors.

AVL tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
Tree is said to be balanced if balance factor of each node is in between -1 to 1, 0 otherwise, the Tree will be unbalanced and need to be balanced.

Balance factor (K) = height (left(K)) - height (right(K)).

- If balance factor of node is 1, then left subtree is one level higher than the right subtree.
- If balance factor is 0, then left & right subtree contains equal height.
- If balance factor is -1, then left subtree is one level lower than the right subtree.

Eg.



AVL Rotations: We peoform rotation in AVL tree only in case if Balance factor is other than -1, 0 and 1. It includes:

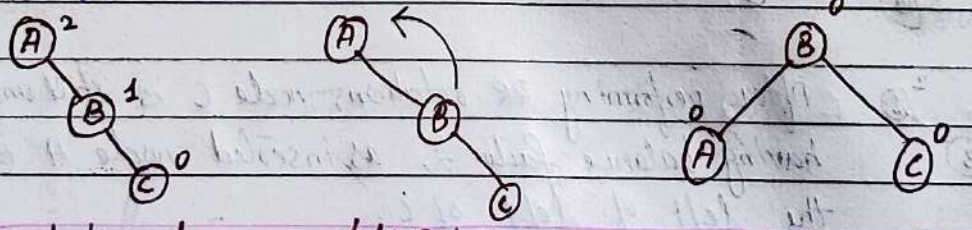
I. Single Rotation → LL
→ RR

II. Double Rotation → LR
→ RL.

For a treee to be unbalanced, minimum height must be atleast 2.

i) RR Rotation: When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, we peoform RR. RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor 2.

Eg



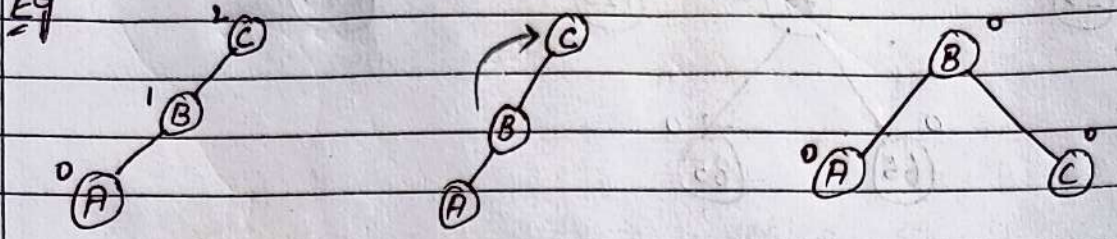
Right unbalanced tree

Left Rotation

Balanced.

ii) **LL Rotation**: When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, we perform LL rotation.
LL Rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

Eg

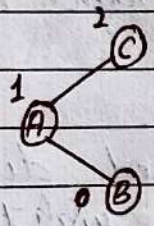


left Unbalanced tree | Right Rotation | Balanced Tree

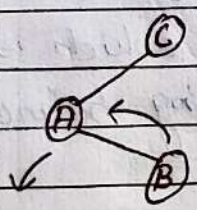
iii) **LR Rotation**: Double rotations are bit tougher than single rotation.

LR Rotation = RR + LL

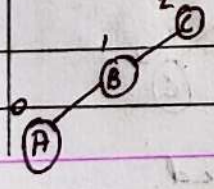
First RR rotation is performed on subtree and then LL rotation is performed on full tree.



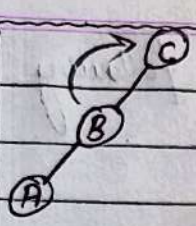
A node B has inserted into the right of left subtree.



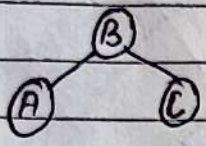
LR = RR + LL so RR is performed first. Now node A, has become the left subtree of B.



After performing RR rotation, node C is still unbalanced, having balance factor 2, as inserted node A is in the left of left of C.

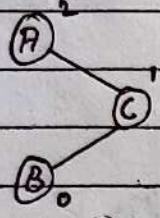


Perform LL clockwise rotation on full tree, ie on node C. node C has now become the right subtree of node B, A is left subtree of B.

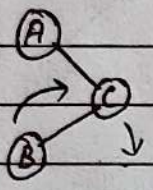


BST is balanced now.

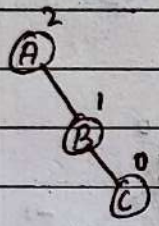
iii) RL Rotation : RL = LL + RR.
First, LL is performed then RR.



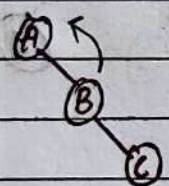
A node B has been inserted into the left subtree of C. and A became an unbalanced node. So, Inserted node is in the left subtree of right subtree of A.



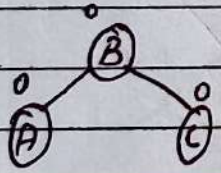
Perform LL on subtree rooted at C and node C become the right subtree of B.



After performing LL rotation, node A is still unbalanced ie. having balance factor 2 because of the right subtree of the right subtree.



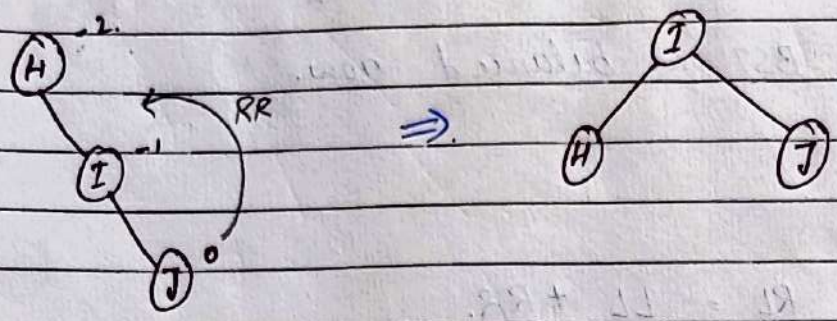
Perform RR rotation on full tree i.e. on node A. node C became the right subtree of node B and node A will become left subtree of B.



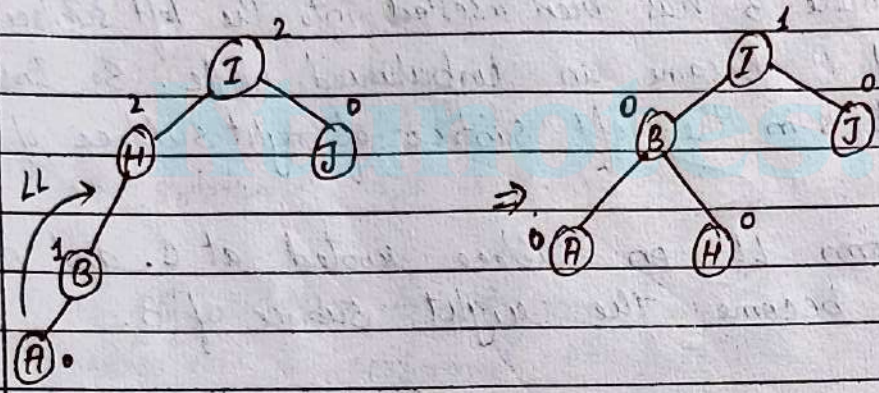
BST is now balanced.

Examples: 1. Construct an AVL tree for the following:
H, I, J, B, A, E, C, F, D, G, K, L.

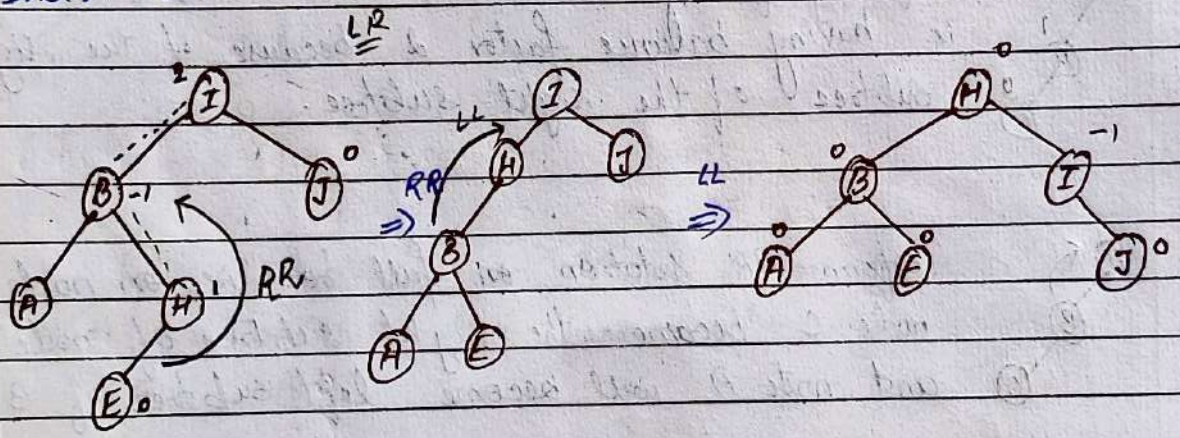
i) Insert H, I, J.



ii) Insert B, A.

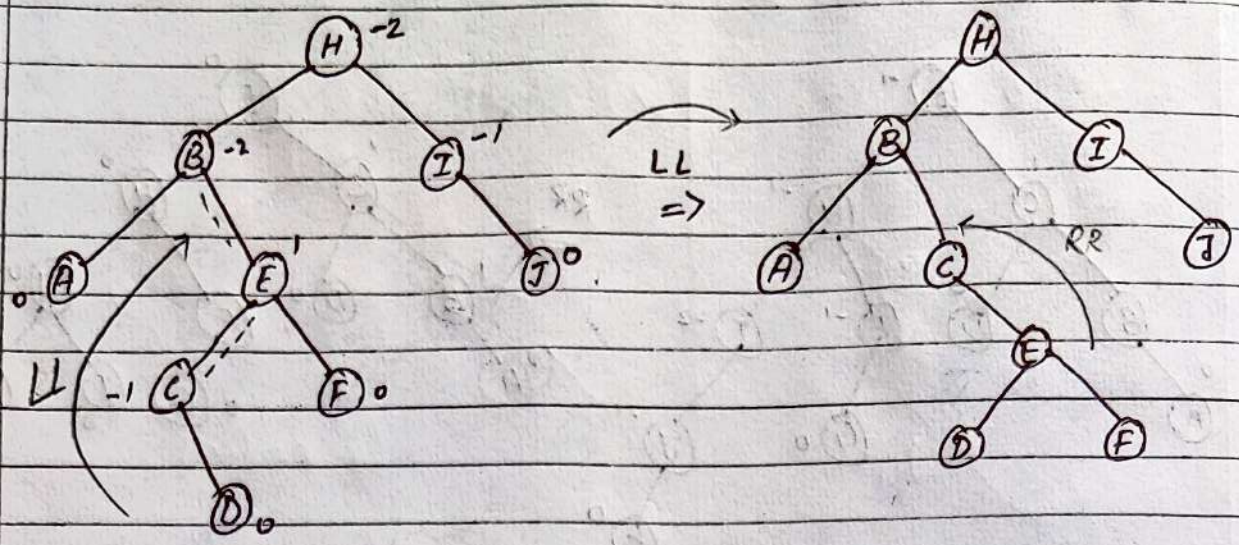


iii) Insert E

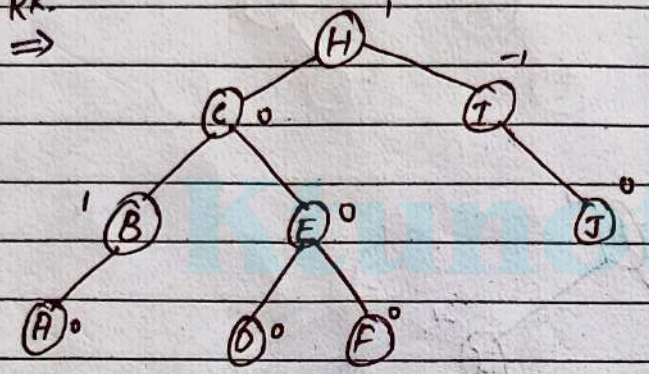


iv) Insert C, F, D.

RL

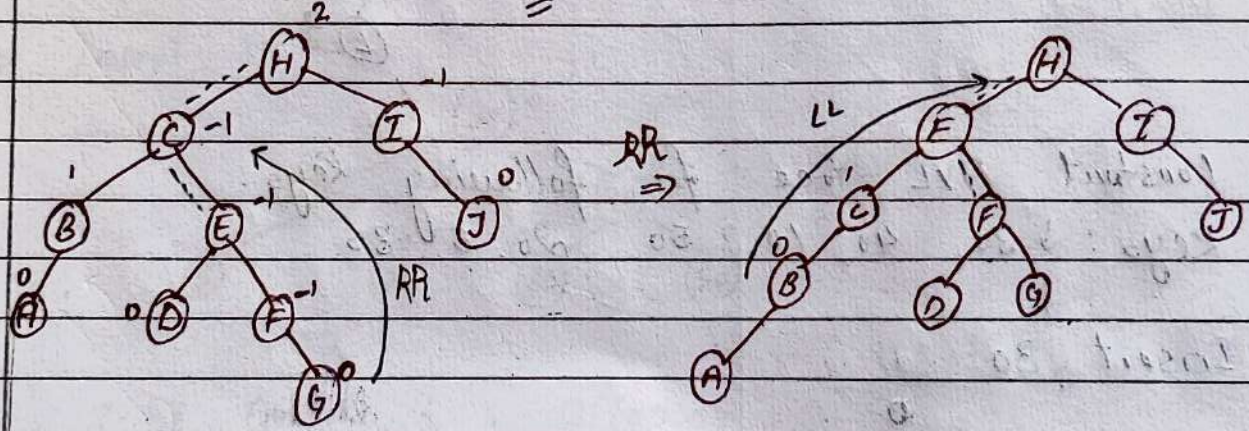


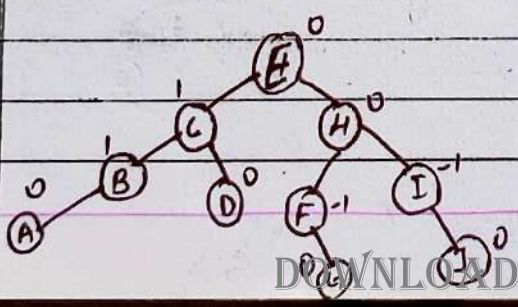
RR



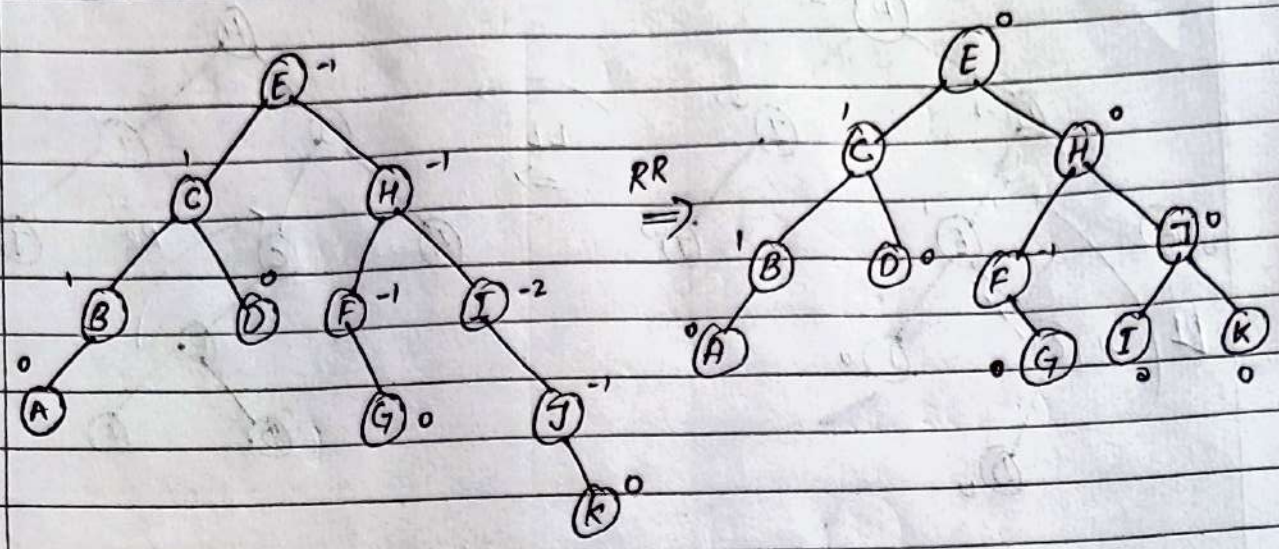
v) Insert G

LR

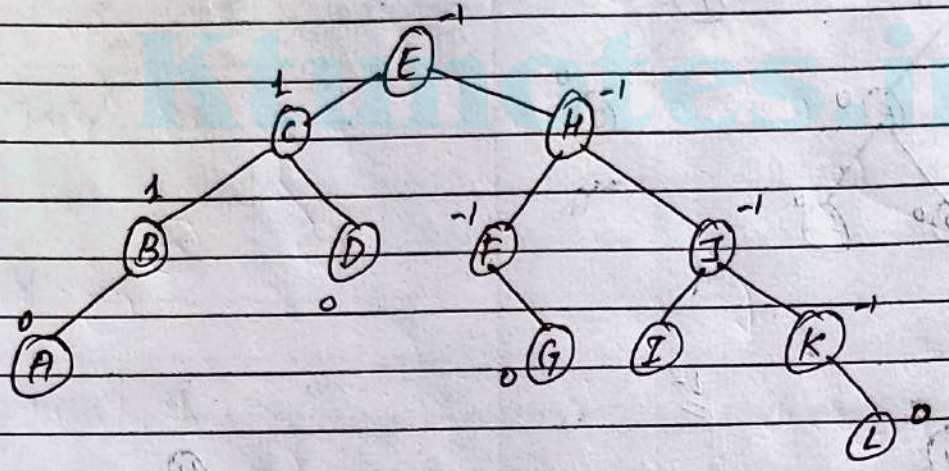




vi) Insert K



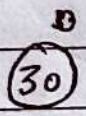
vii) Insert L



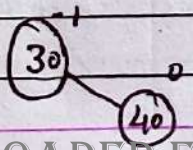
2. Construct AVL tree for following keys:

I. Keys: 30, 40, 10, 50, 20, 5, 35.

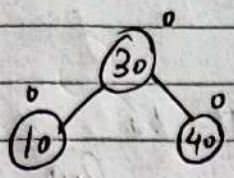
i) Insert 30



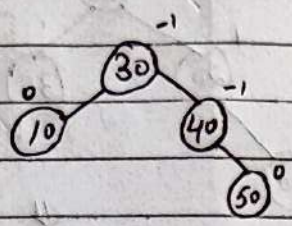
ii) Insert 40



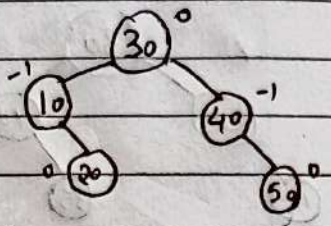
iii) Insert 10.



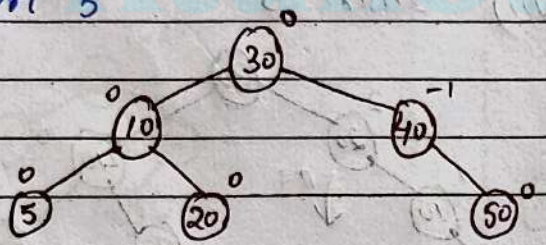
iv) Insert 50



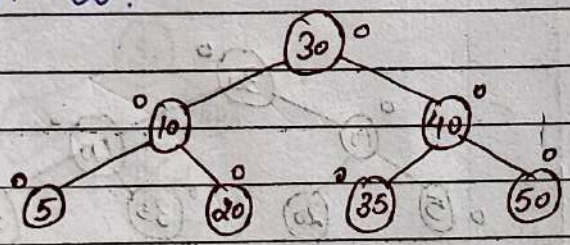
v) Insert 20



vi) Insert 5



vii) Insert 35.

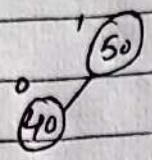


Time Complexity = O(log n)

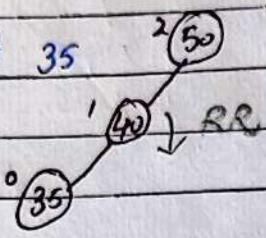
II. Keys : 50, 40, 35, 30, 20, 10, 5.

i) Insert 50

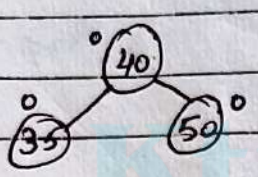
ii) Insert 40



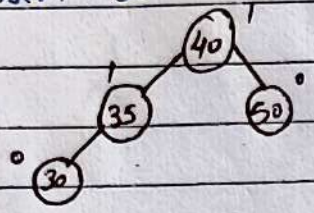
iii) Insert 35



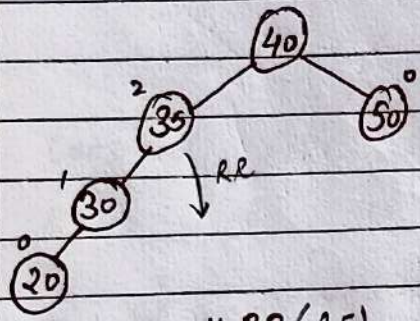
⇓ RR(50)



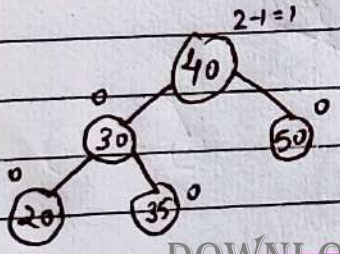
iv) Insert 30



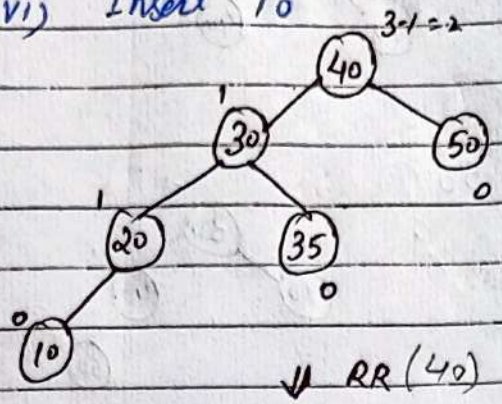
v) Insert 20



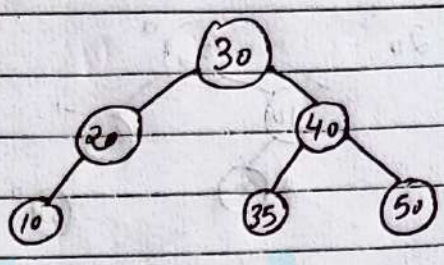
⇓ RR(35)



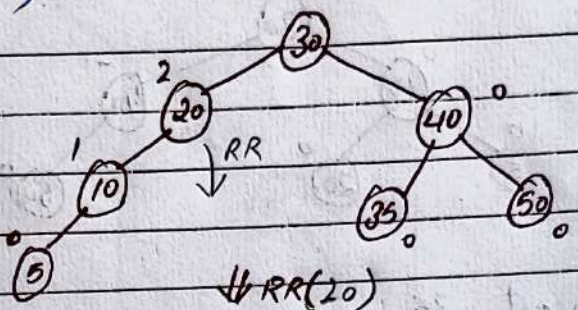
vi) Insert 10



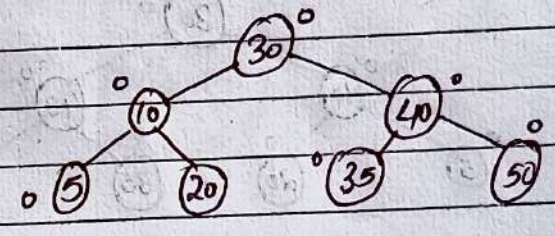
⇓ RR(40)



vii) Insert 5



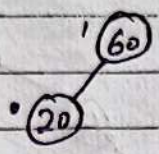
⇓ RR(20)



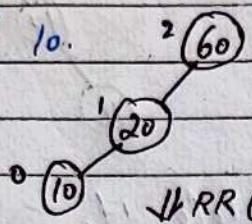
III) Keys : 60, 20, 10, 40, 50, 30, 70.

i) Insert 60

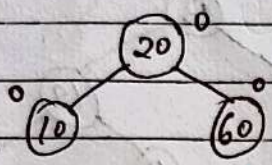
ii) Insert 20



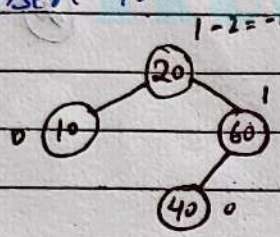
iii) Insert 10.



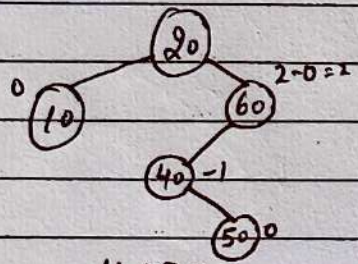
⇓ RR (60)



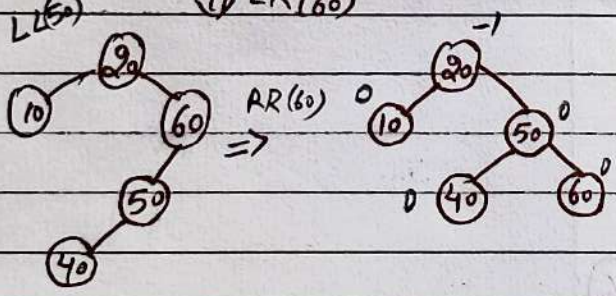
iv) Insert 40



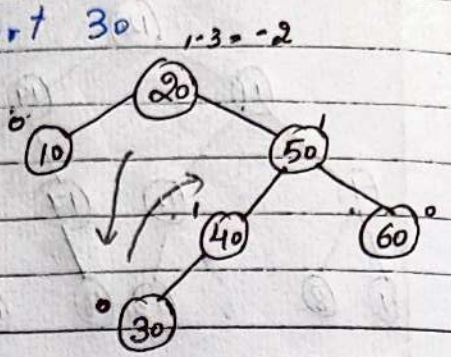
v) Insert 50



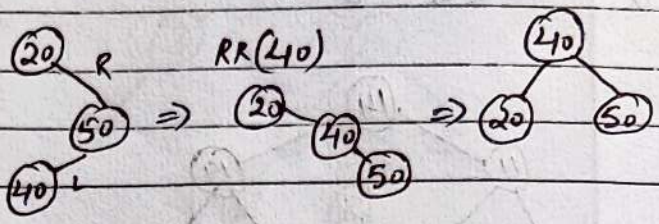
⇓ LR (60)

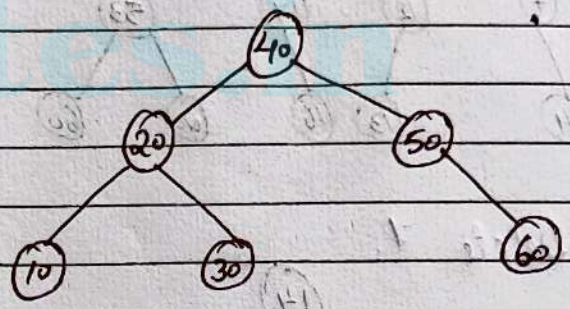


vi) Insert 30

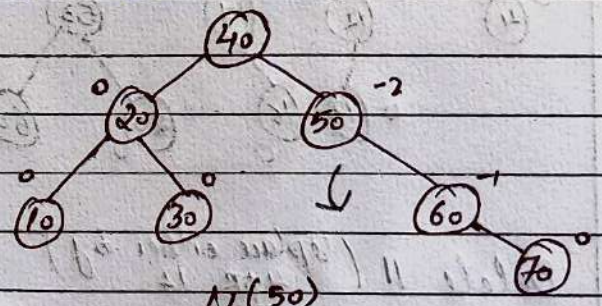


⇓

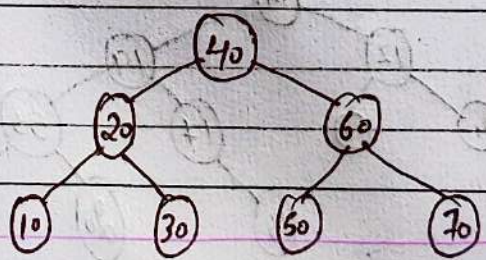




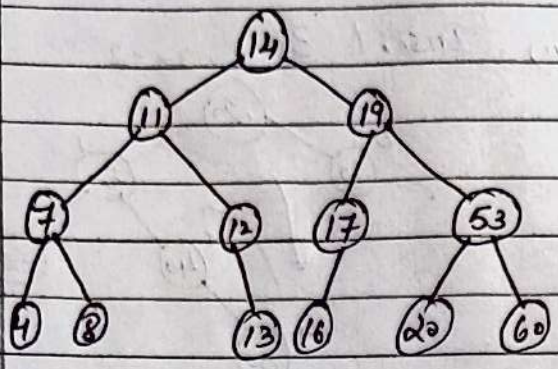
vii) Insert 70.



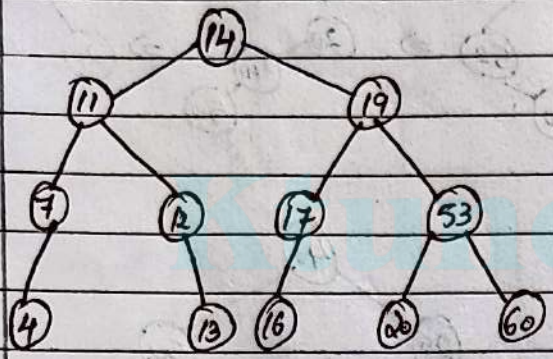
RL(50)



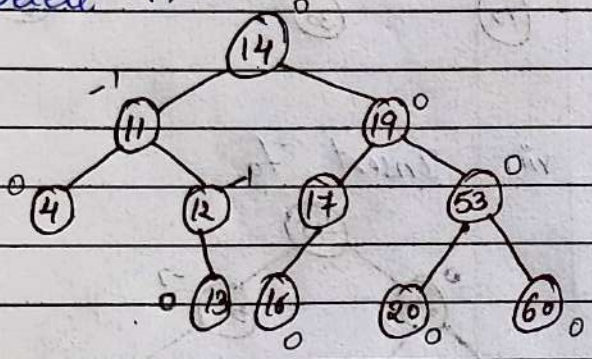
3. Delete 8, 7, 11, 14, 17 from the given AVL tree



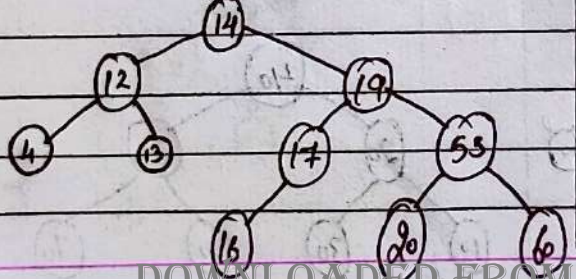
i) Delete 8



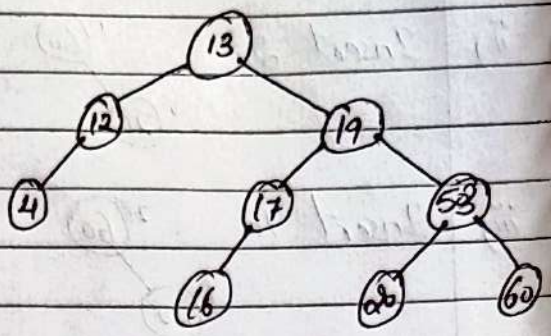
ii) Delete 7.



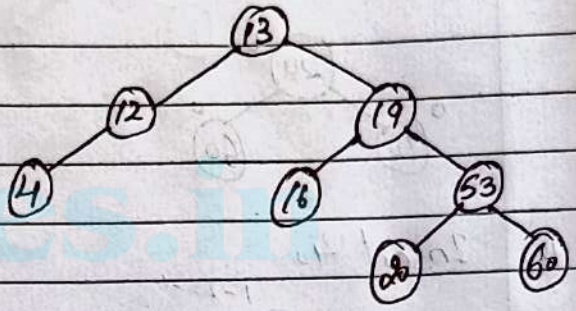
iii) Delete 11 (replace either by 4 OR 12)



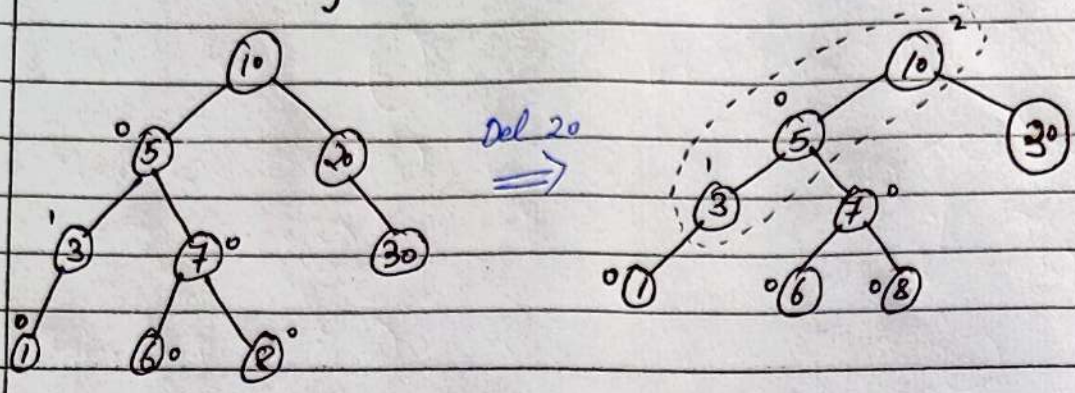
iv) Delete 14 (left higher (or) right lowest, replace by 13 or 16)

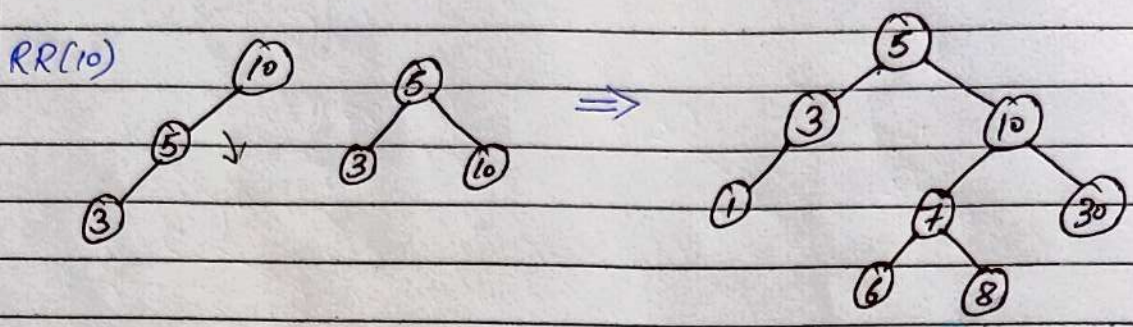


v) Delete 17.



4. Delete 20 from the given AVL tree.



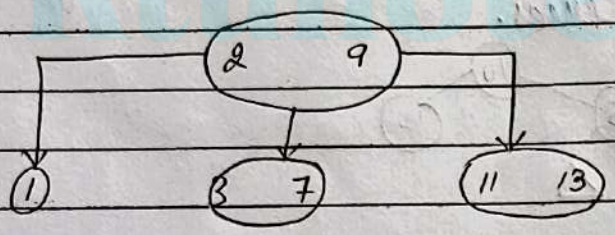


⇒ Other Self - Balanced BST Trees :

Height Balanced BST Trees that automatically keeps height as small as possible when insertion and deletion operations are performed on trees.
Time Complexity is $O(\log n)$.

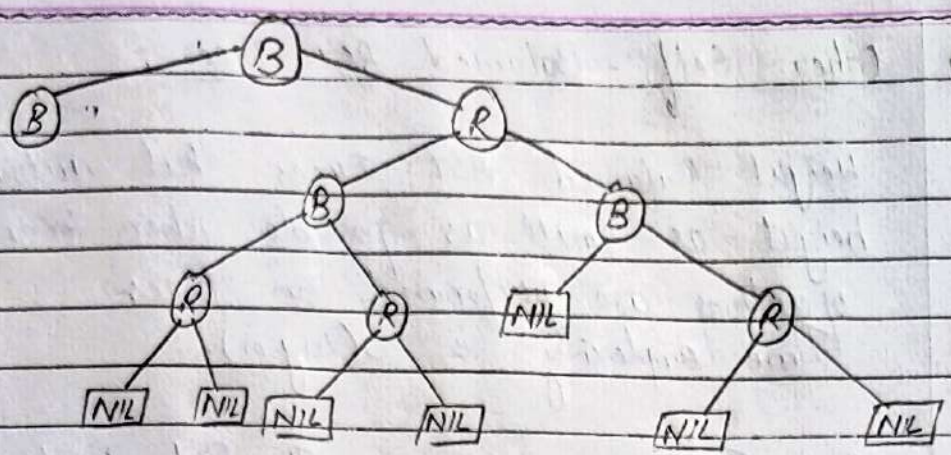
I. 2-3 Tree
II. Red-Black Tree
III. AVL Trees
IV. Splay Tree
V. Weight-Balanced Tree
VI. B-Tree

I. 2-3 Tree: A 2-3 tree is a tree data structure, where every node with children (internal node) has either two children (2-node) and one data element or three children (3-node) and 2 data elements.

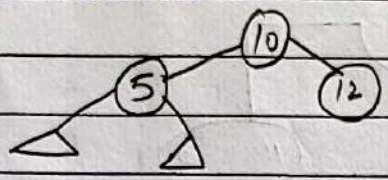


II. Red-Black Tree: A red black tree is a kind of self balancing binary search tree where each node has an extra bit and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.

- Root is always a black node.
- Red node cannot have red child or red parent.
- NIL (leaf node) are always black.
- Every path from a node to any of its descented will node has same no. of black node



III. **Splay Tree**: A splay tree contains the same operations as a Binary search tree i.e. Insertion, Deletion and searching, but it also contains one more operation, i.e. splaying. So, all the operations in the splay tree are followed by splaying. Splay trees are not strictly balanced trees, but they are roughly balanced trees.



IV. **Weight-Balanced Tree**: is a binary tree in which for each node the number of nodes in the left subtree is atleast half and atmost twice the number of ~~nnnn~~ nodes in the right subtree.

V. **B-Tree**: is a tree data structure that keeps data sorted and allows searches, insertions and deletion in logarithmic amortized time