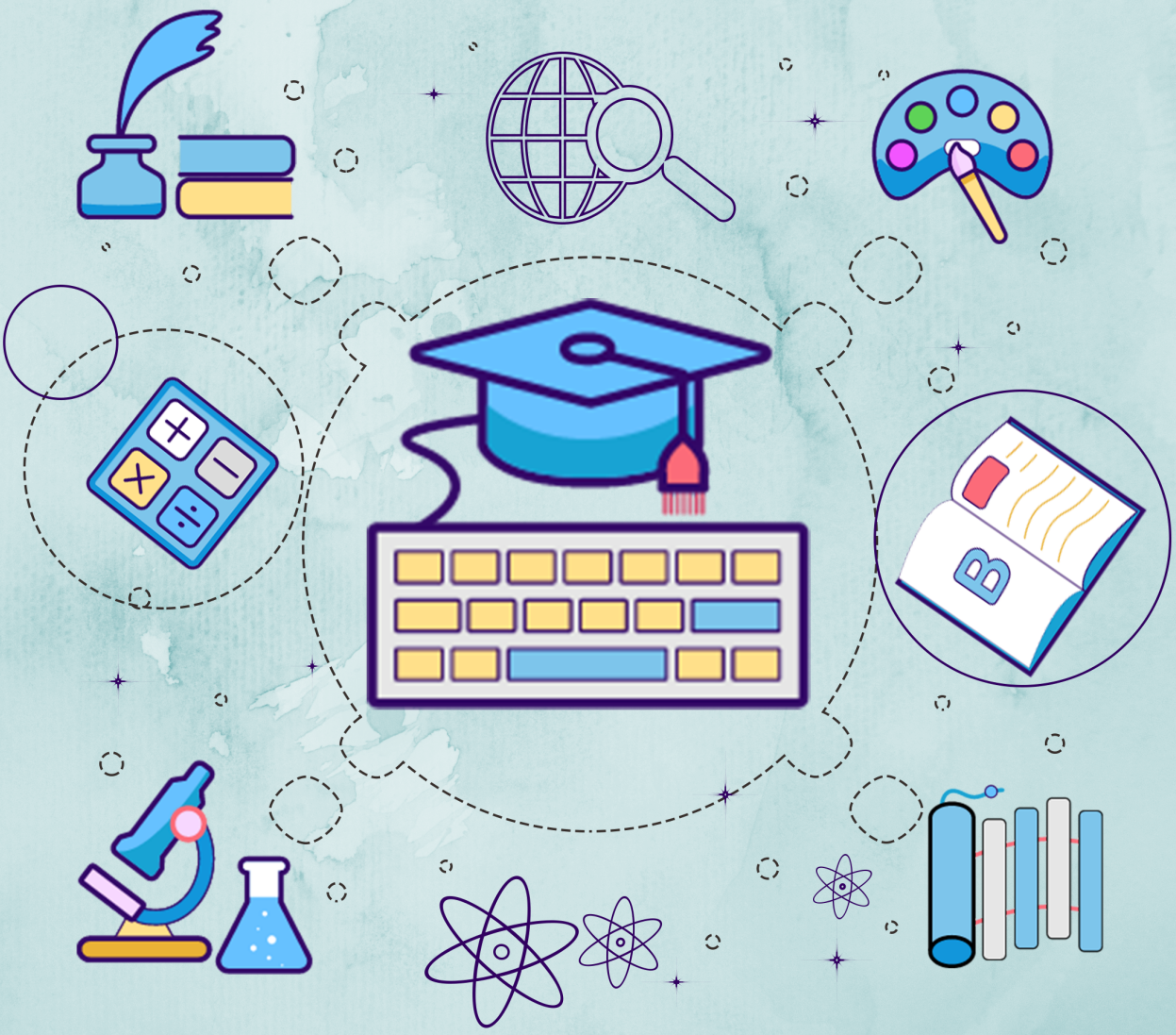
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

# KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

# KTU STUDY MATERIALS

## ALGORITHM ANALYSIS AND DESIGN

## CST 306

# Module 2

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

Module 2.

## AVL Trees

→ AVL Tree is a height balanced Binary Search Tree, in which the difference b/n the height of the left & right subtree is either -1 or or. -1, 0, or +1

→ AVL Trees are also called a self balancing binary Search tree.

→ AVL Trees Search time is O(logn)

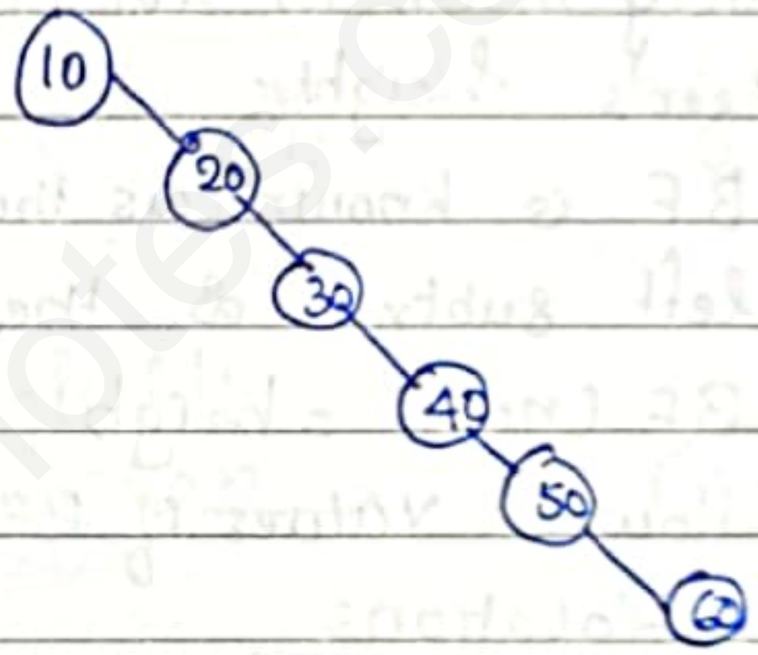
→ It is named after its inventors (AVL) Adelson, Velsky & Landis

Why AVL Trees?

To understand the need for AVL trees, let us look at some disadvantages of simple binary search trees.

→ Consider the fol. keys inserted in the given order in the BST

keys: 10, 20, 30, 40, 50, 60

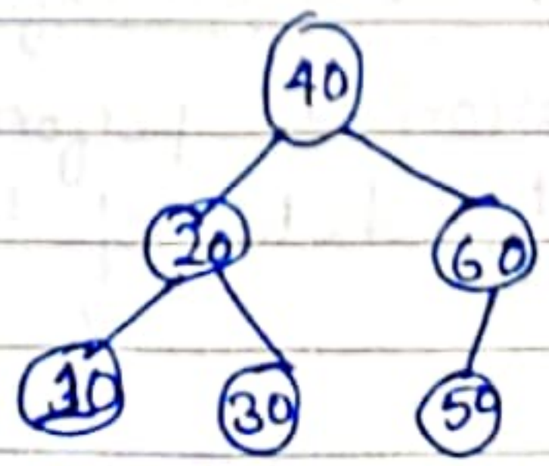



→ The height of the tree grows linearly in size when we insert the keys in increasing order of their value. Thus the search operation, at worst case, takes O(n).

→ On the other hand, if the height of the tree is balanced, we get better searching time.

→ Let us now look at the same keys but inserted in a diff. order.

key:- 40, 20, 30, 60, 50, 10



Here, the keys are the same, but since they are inserted in a diff. order, they take diff. positions, & the height of the tree remains balanced. Hence search will not take more than O(logn) for any element of the tree.

→ In AVL trees, we keep a check on the height of the tree during insertion operation.

→ Modifications are made to maintain the balanced height without violating the fundamental properties of BST.

## Balance Factor in AVL trees

Balance factor (BF) is a fundamental attribute of every node in AVL trees that helps to monitor the tree's height.

→ BF is known as the difference b/n the height of the left subtree & the right subtree.

→ BF (node) = height of left subtree - height of right subtree

→ Allowed values of BF are -1, 0 and +1.

## AVL Rotations

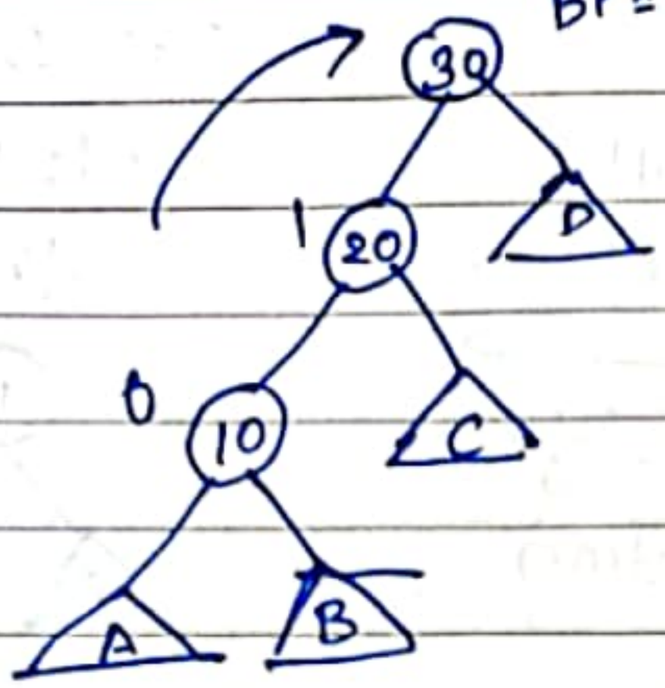
To make the AVL tree balance itself, when inserting or deleting a node from the tree, the fol. rotations are performed

→ LL Rotation (Left - Left Rotation)
→ RR Rotation (Right - Right Rotation)
→ LR Rotation (Left - Right Rotation)
→ RL Rotation (Right - Left Rotation)

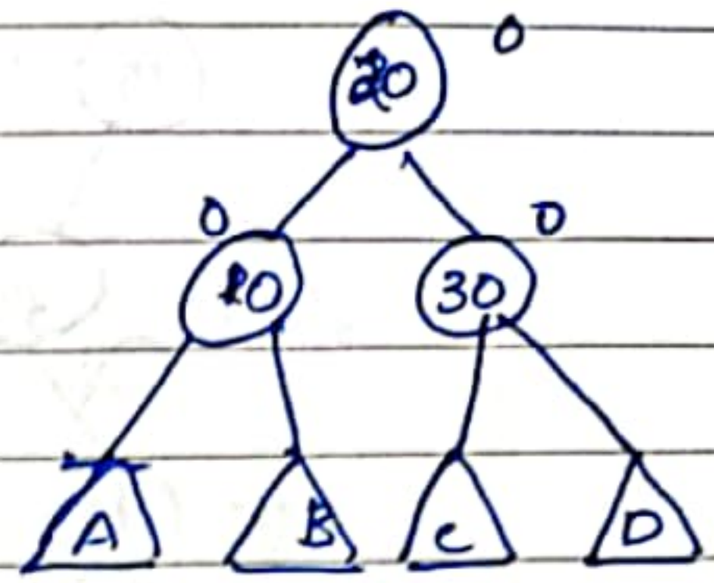
## LL Rotation (Left - Left Rotation)

This rotation is performed when a new node is inserted at the left child of the left subtree

eg:-




Single Right Rotation →

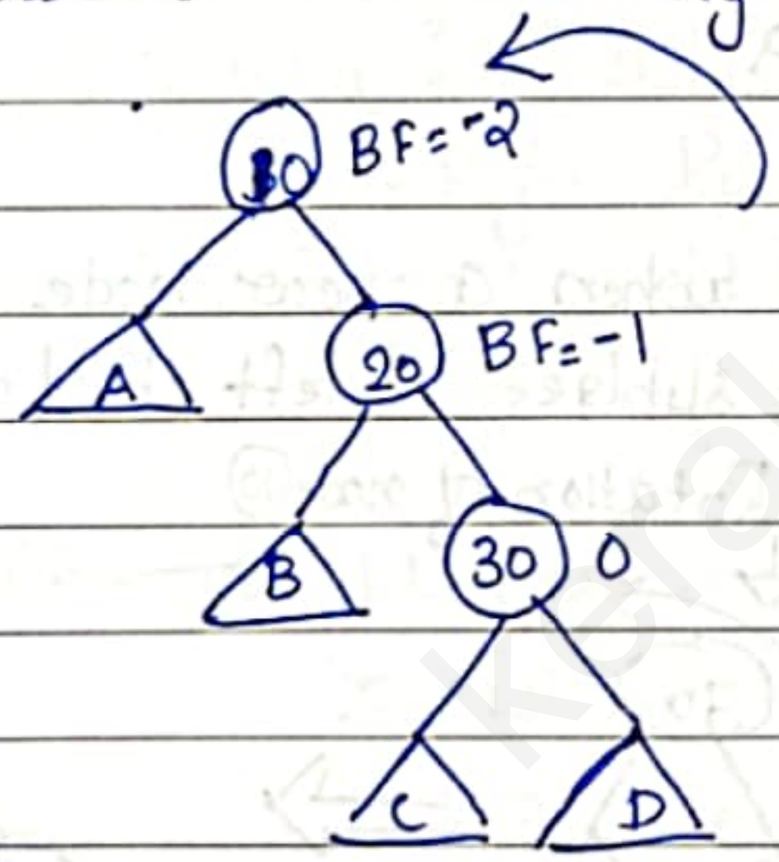



Tree imbalanced as BF(30) = +2

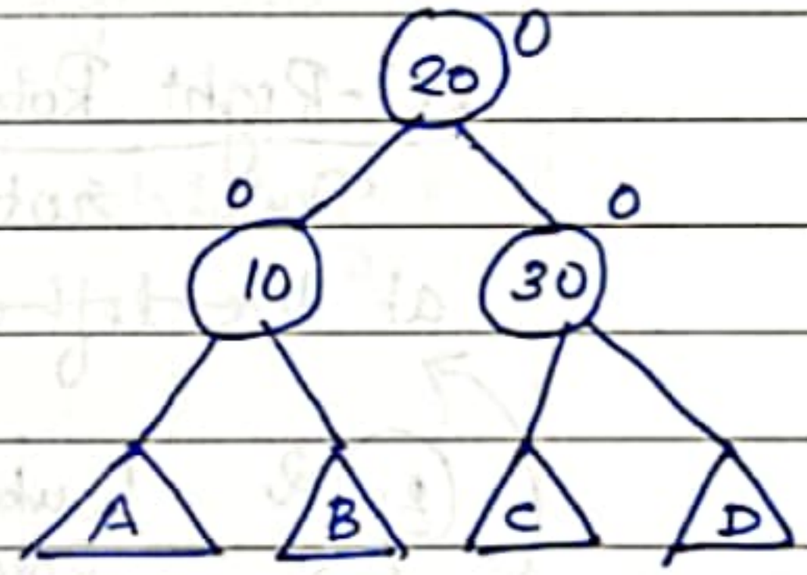
LL Rotation → { BF of a node = 2, BF of its left child = +1 }

## Right - Right Rotation.

This rotation is performed when a new node is inserted at the right child of the right subtree.




Single left Rotation →

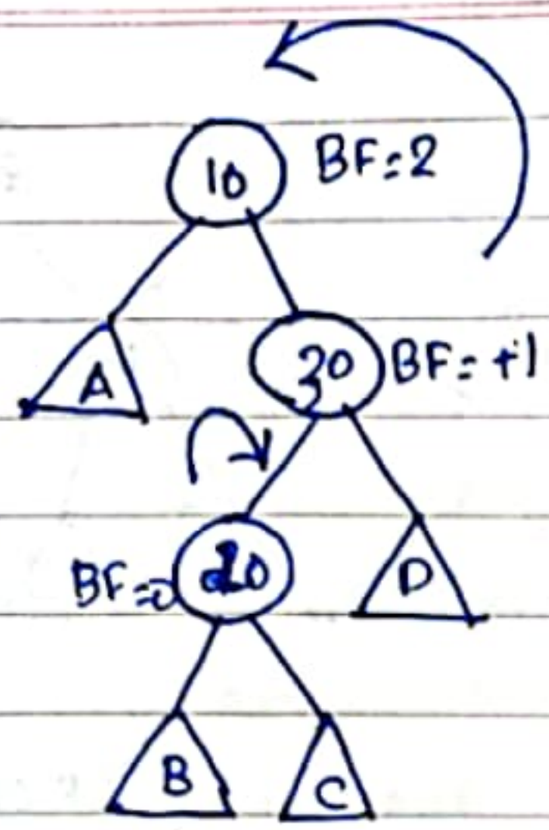



RR Rotation → BF of a node = -2
BF of its right child = -1

## Right left Rotation

This rotation is performed when a new node is inserted at the ~~right child of~~ the left subtree of the right child of the pivot node.

R-L Rotation → { BF of a node = -2, BF of its right child = +1 }

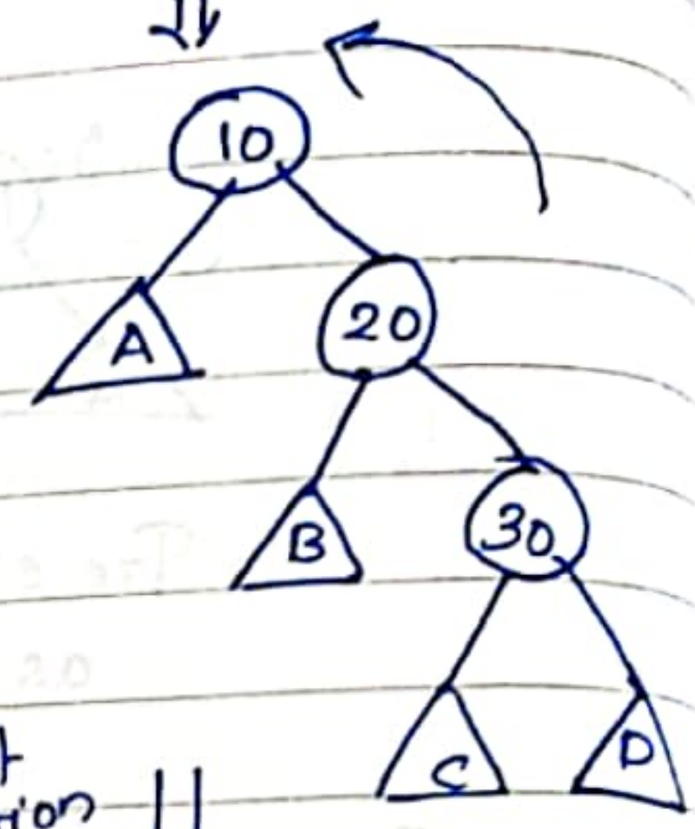



Double Rotation:

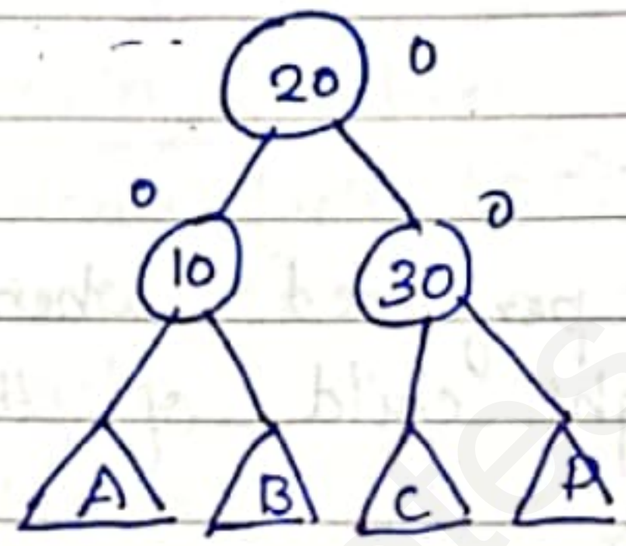
Right-left Rotation

Right

First left rotation of node (30) with BF=1

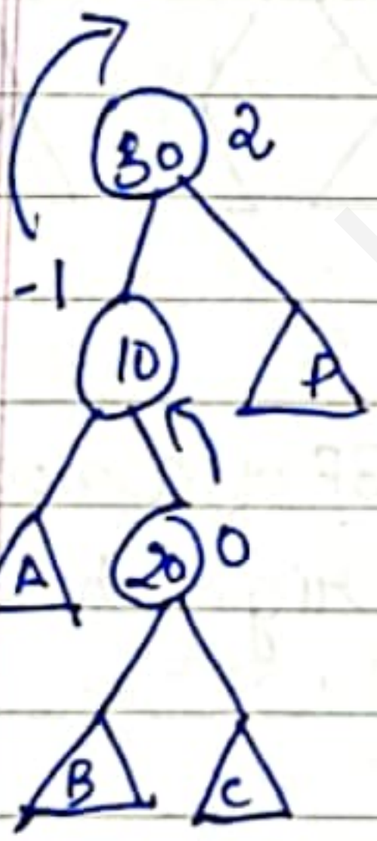



Left Rotation of node (10) with BF=2.




## Let-Right Rotation

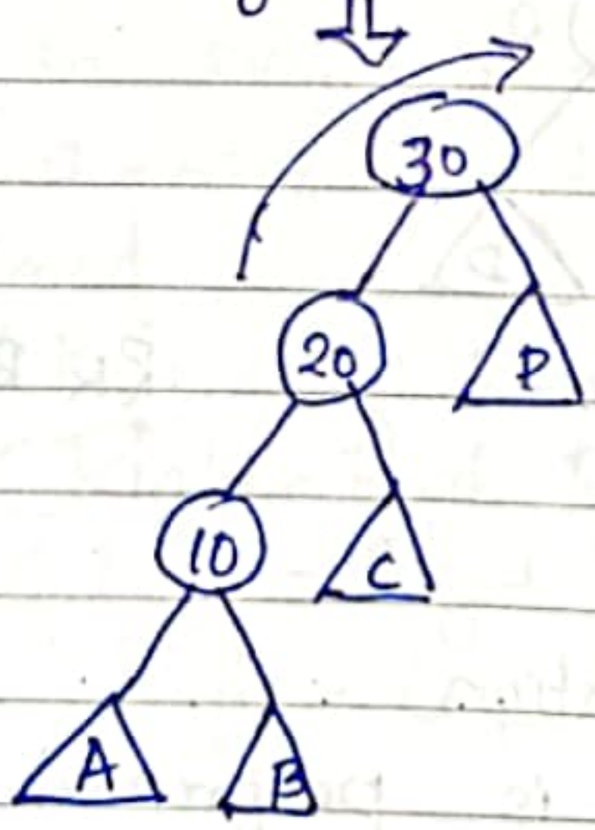
This rotation is performed when a new node is inserted at the ~~left child of~~ the right subtree of left child of pivot node



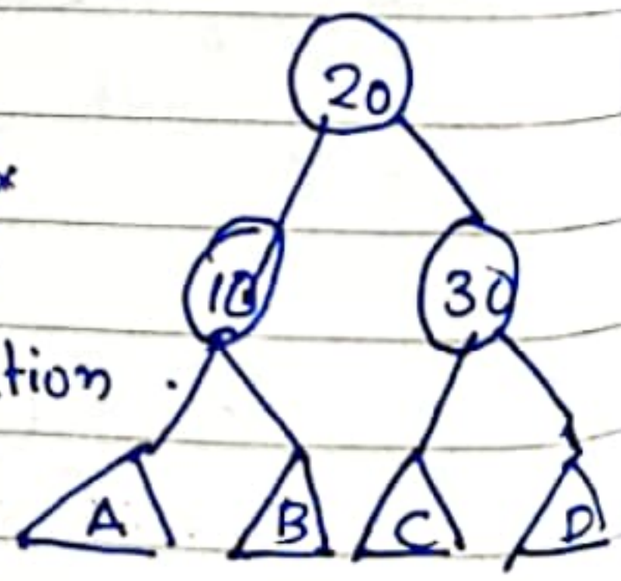

Double Rotation

Left Right Rotation

Left Rotation of node (10)




Right Rotation.




Left-Right Rotation.

⇒ Balance factor of a node = +2

Balance factor of its left child = -1

## Insertion Operation in AVL trees

Steps to follow for insertion.

Let the newly inserted node be w.

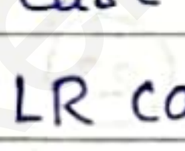
1) Perform BST insert for w.
2). Starting from w, travel up & find the first unbalance node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z & x be the grandchild of z that comes on the path from w to z.
3). Rebalance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases (rotations).
   (a) y is a left child of z & x is left child of Y - LL Rotation
   (b) y is a left child of z & x is a right child of Y - LR Rotation
   (c) Y is a right child of z & x is a right child of Y - Right-Right Rotation (RR rotation)
   (d) y is a right child of z & x is left child of Y - Right left case.

LL case - Right Rotate(z)
LR case - Left rotate(Y) then Right Rotate(z)
RR case - Left rotate(z).
RL case - Right rotate(Y) then Left Rotate (z)

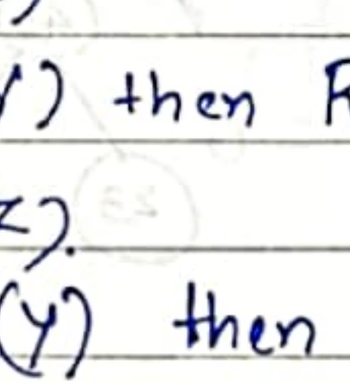
Q. Construct AVL tree for the fol. numbers.

14, 8, 12, 46, 23, 5, 77, 88, 20.

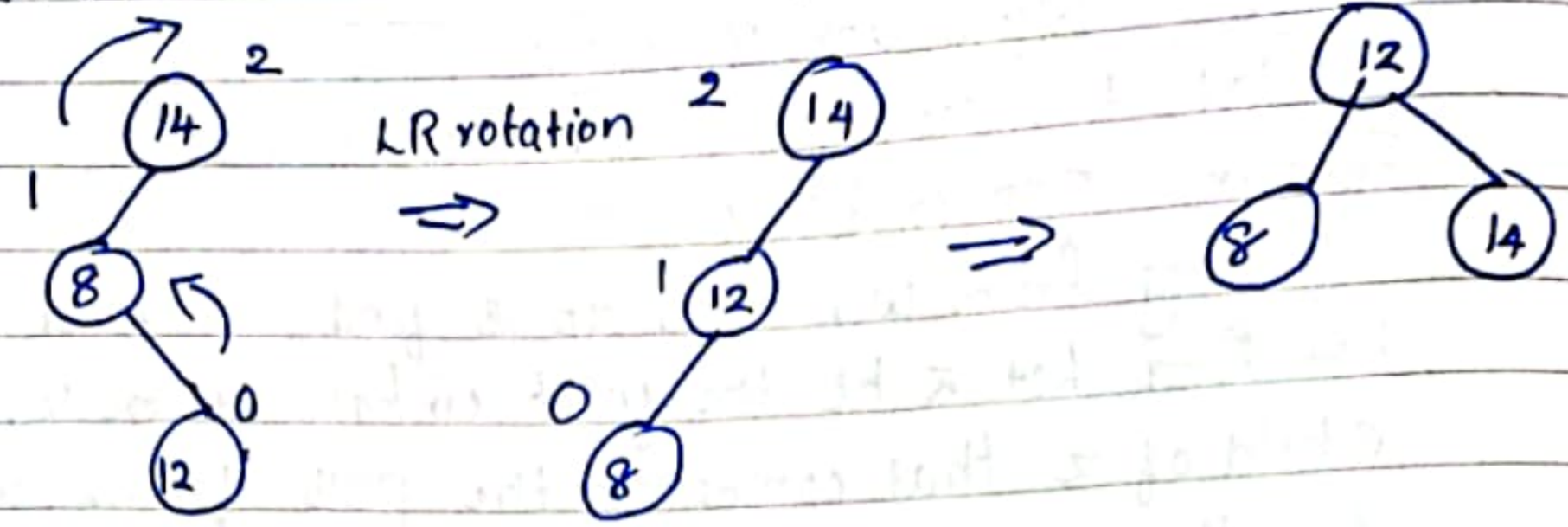
Insert 14



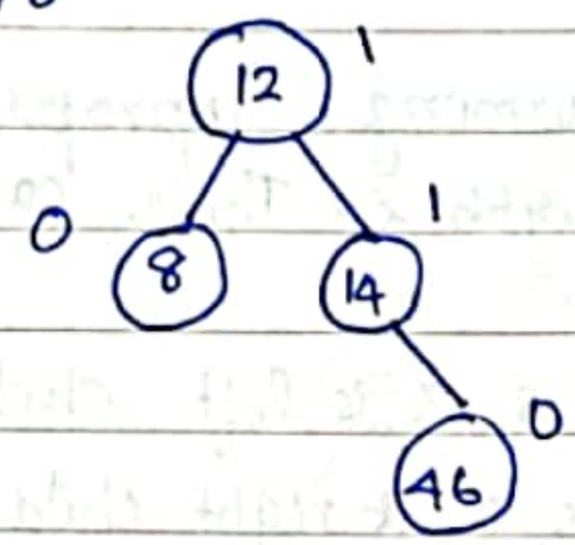

Insert 8



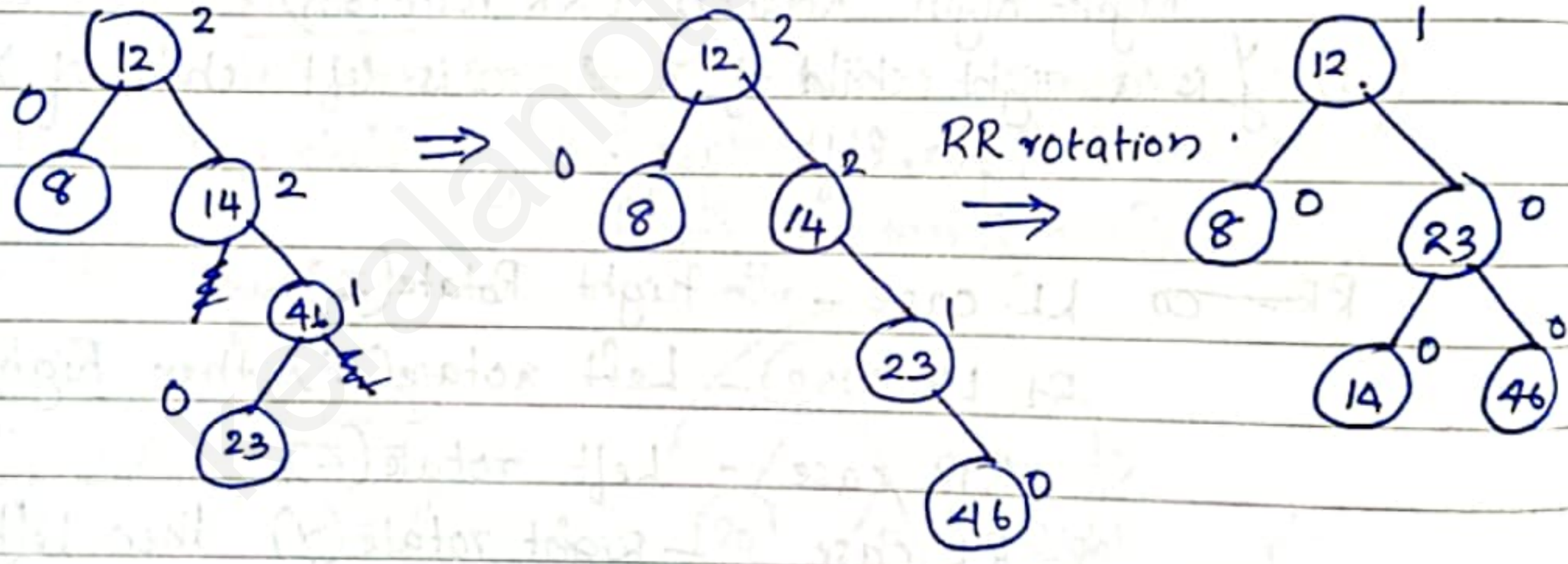
Insert 12




Insert 46

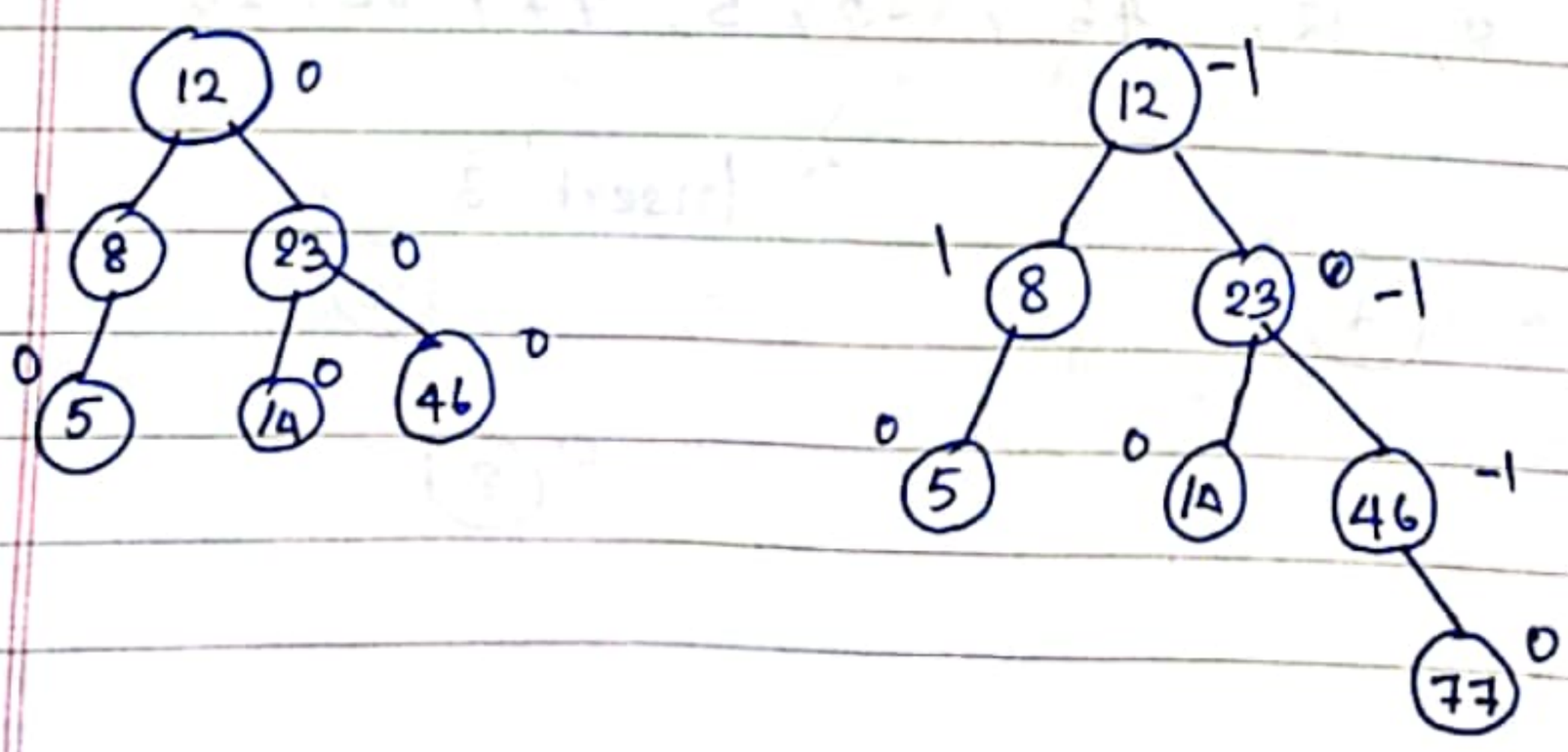



Insert 23

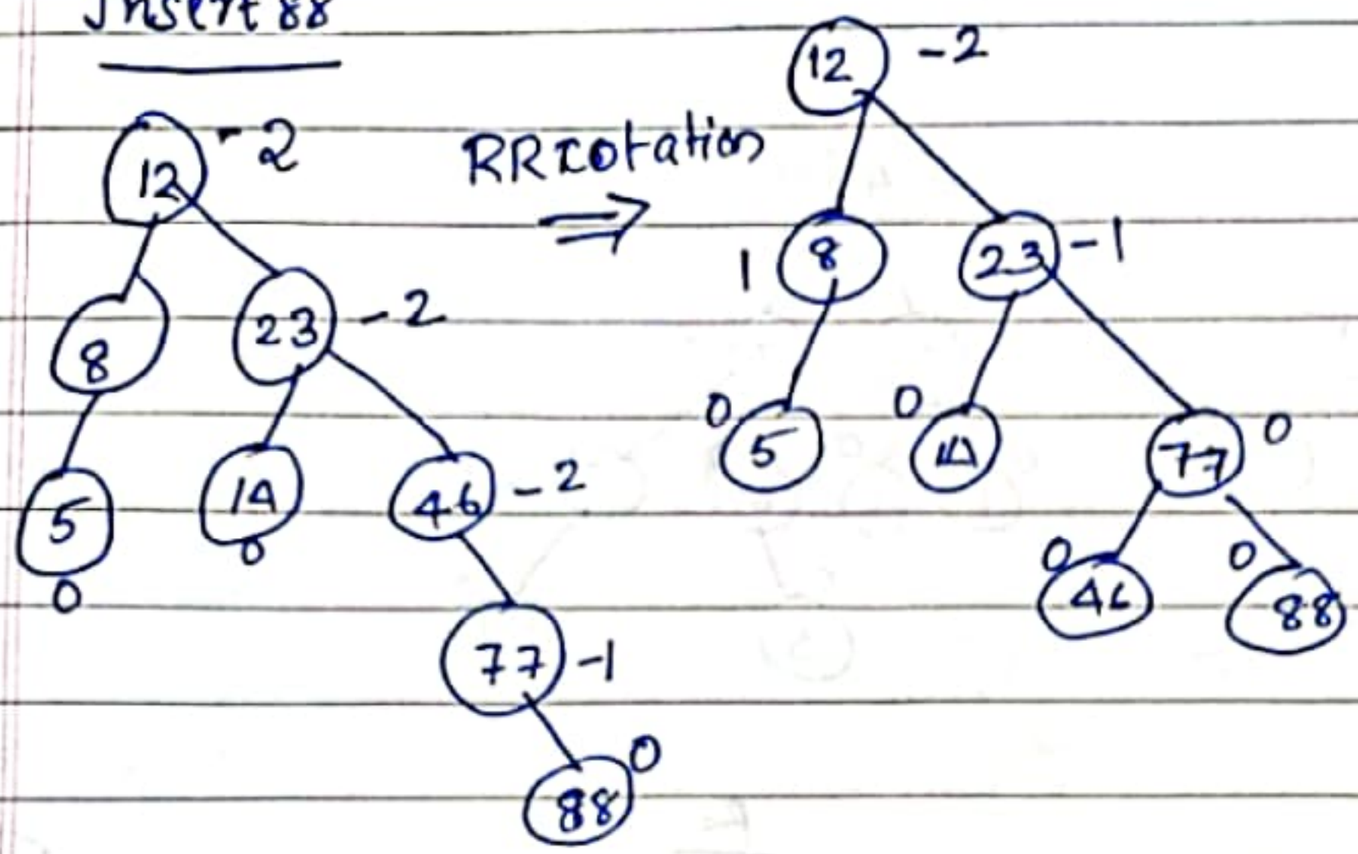



Insert 5

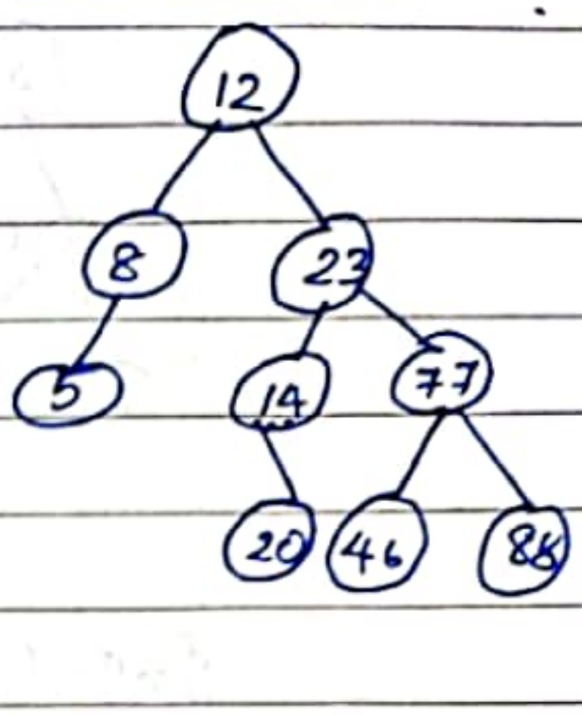
Insert 77



Insert 88




Insert 20




Q. Construct AVL trees for the fol. numbers.

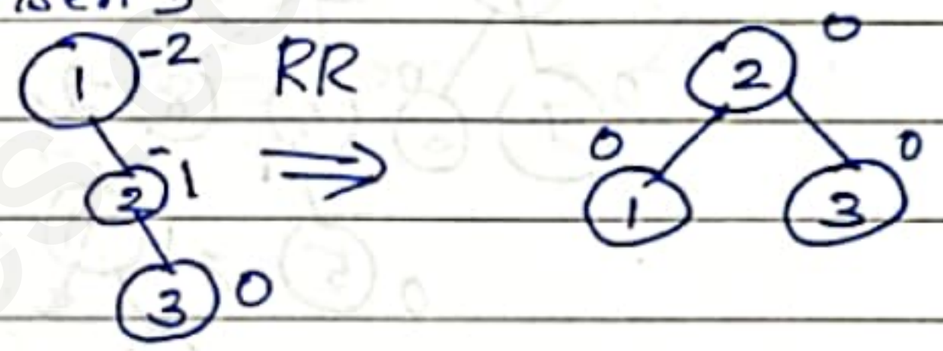
1, 2, 3, 4, 8, 7, 6, 5, 11, 10, 12

Soln

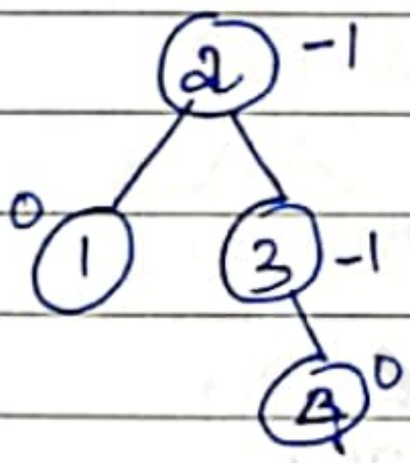
Insert 1


Insert 2

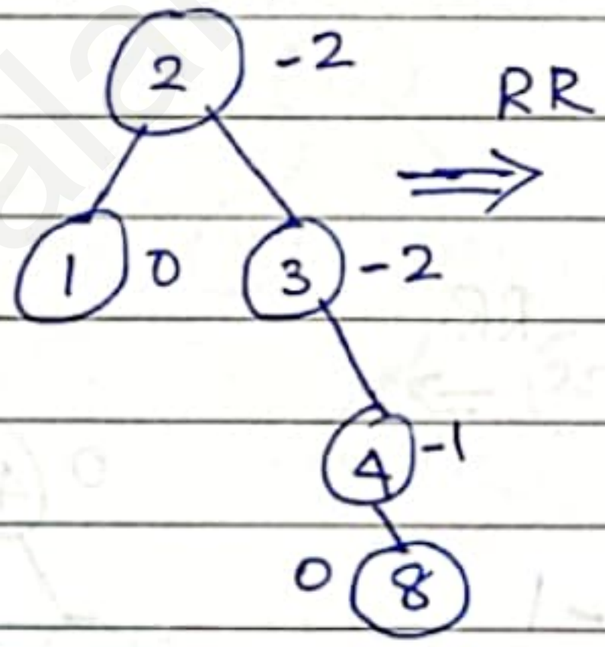

Insert 3

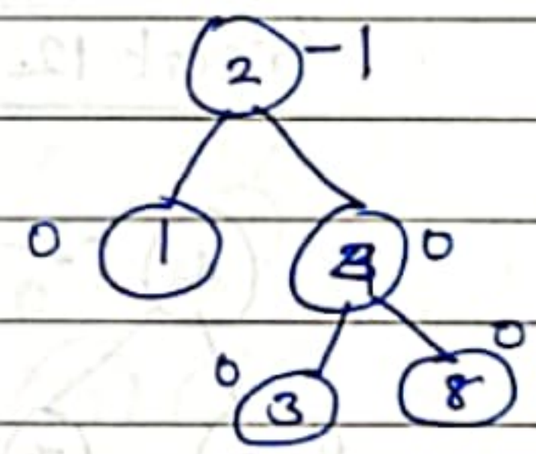



Insert 4



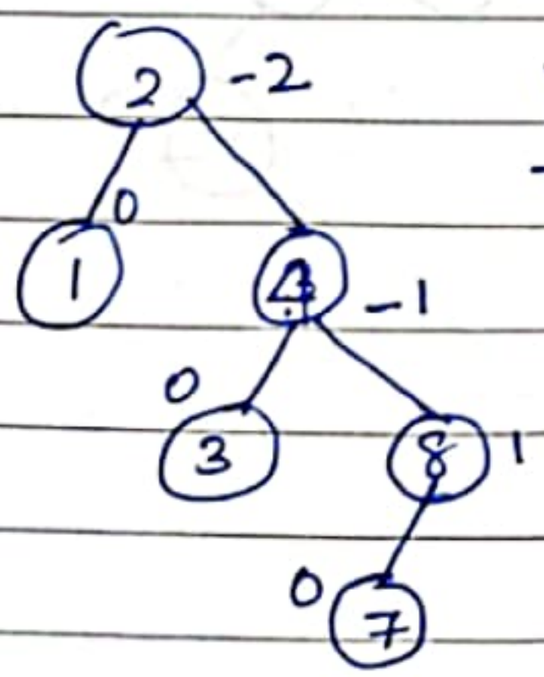

Insert 8.

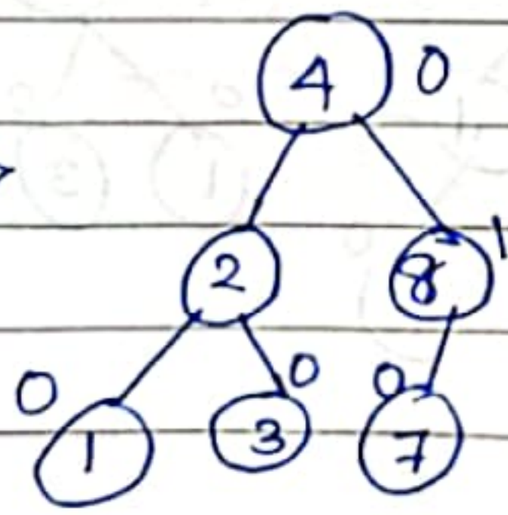


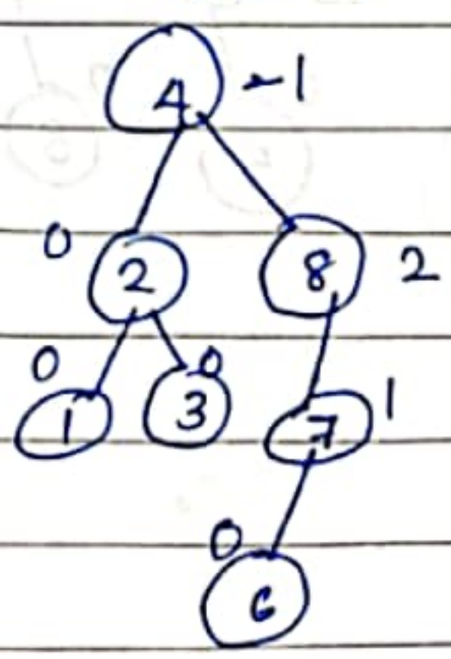



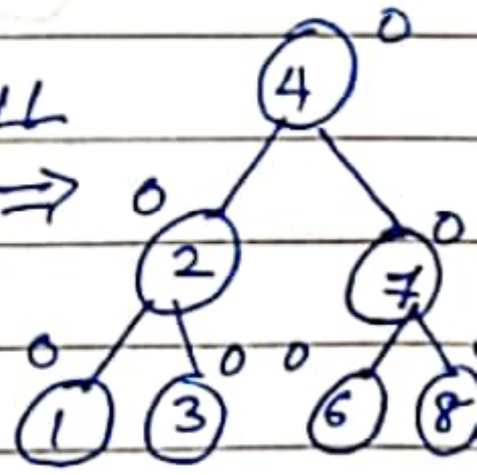

Insert 7

Insert 6

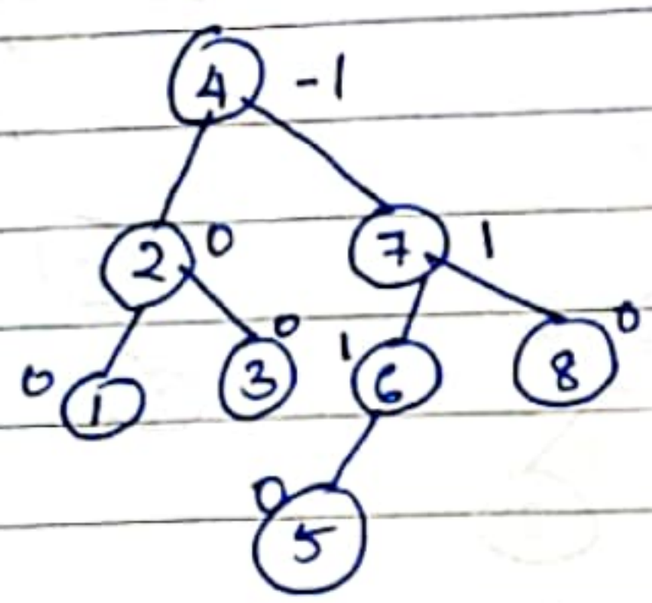


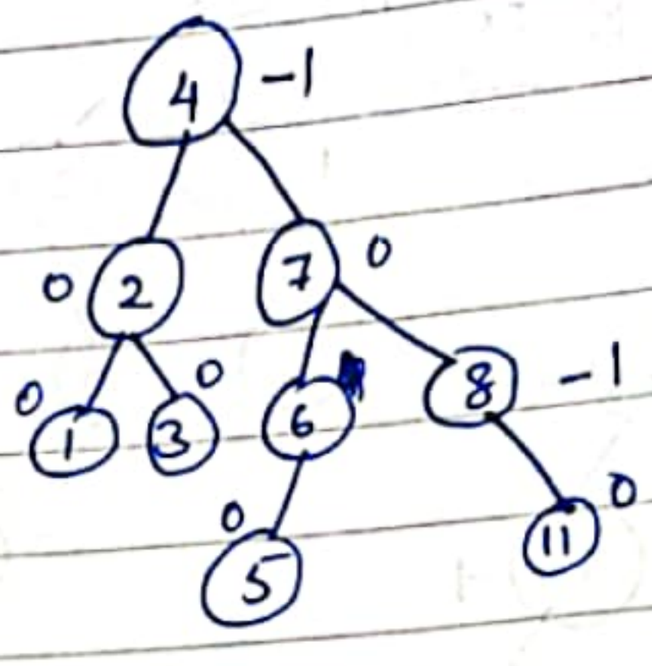



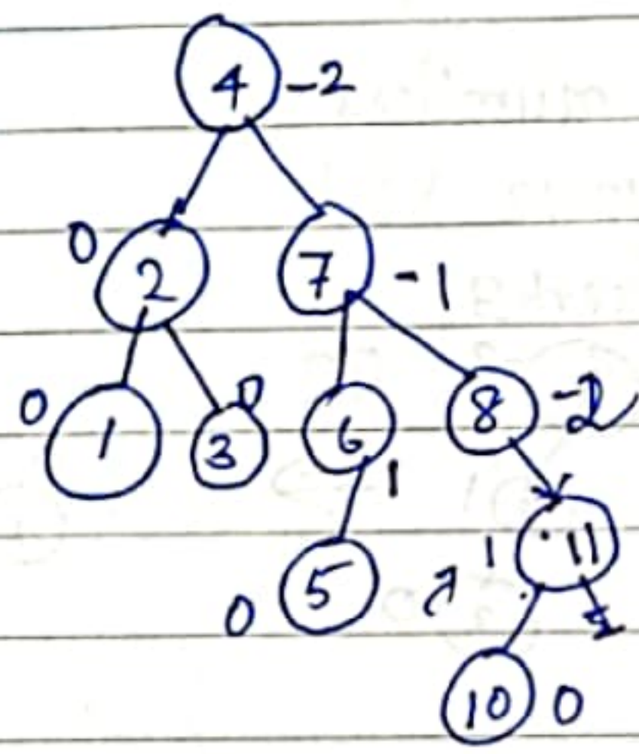
Insert 5




Insert 11

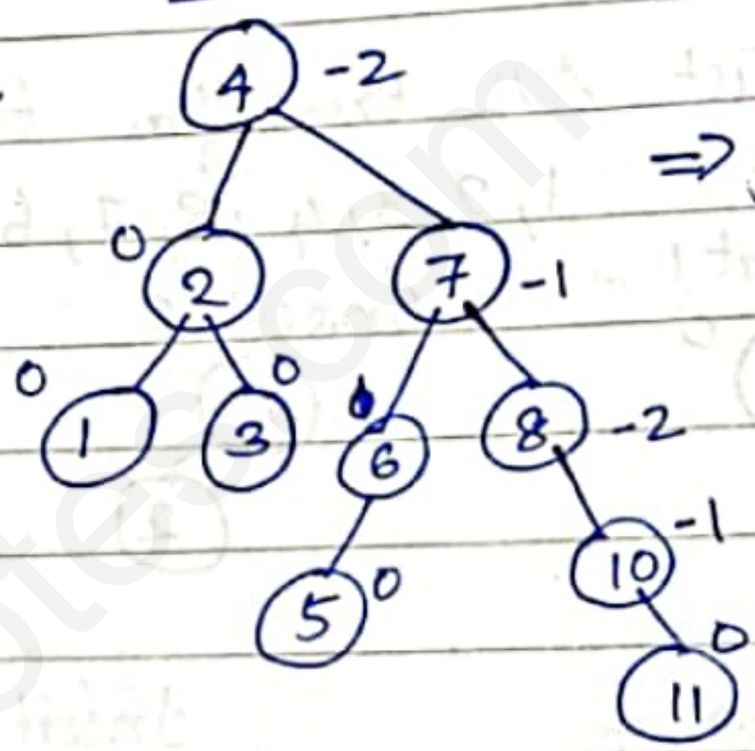



Insert 10.



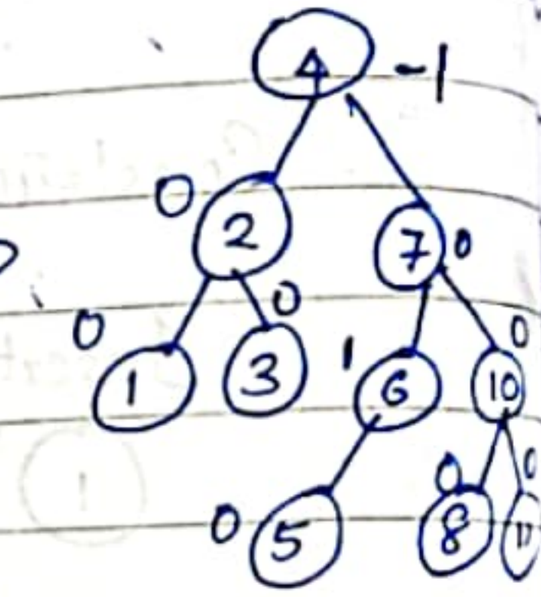

RL

⇒

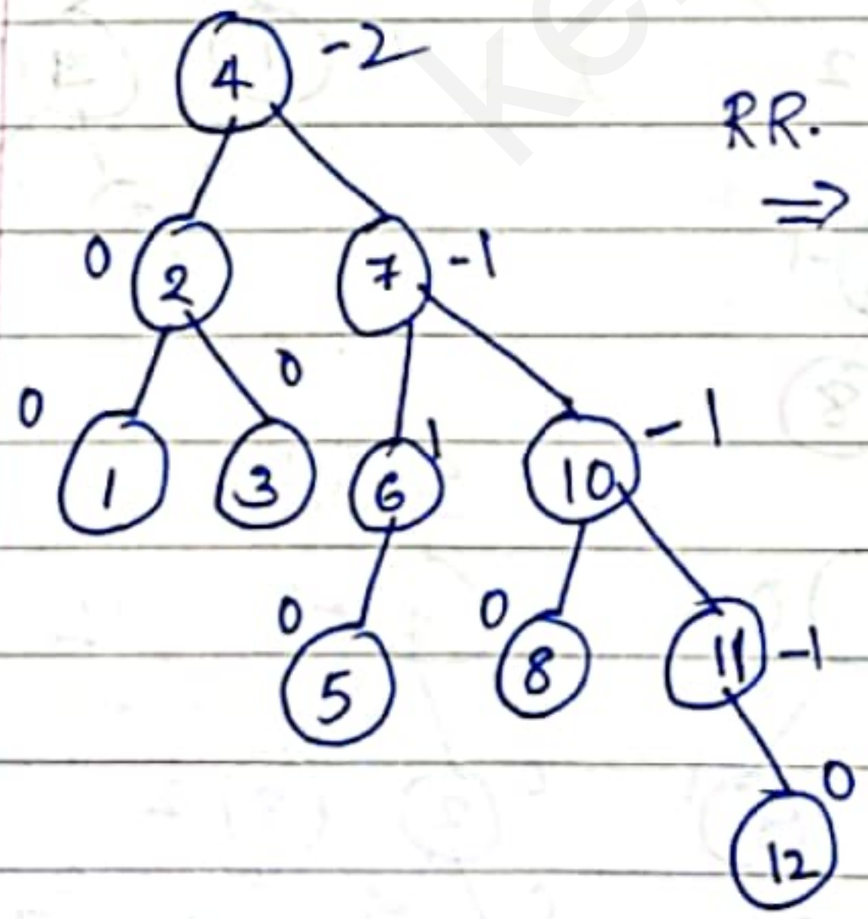



RR

⇒

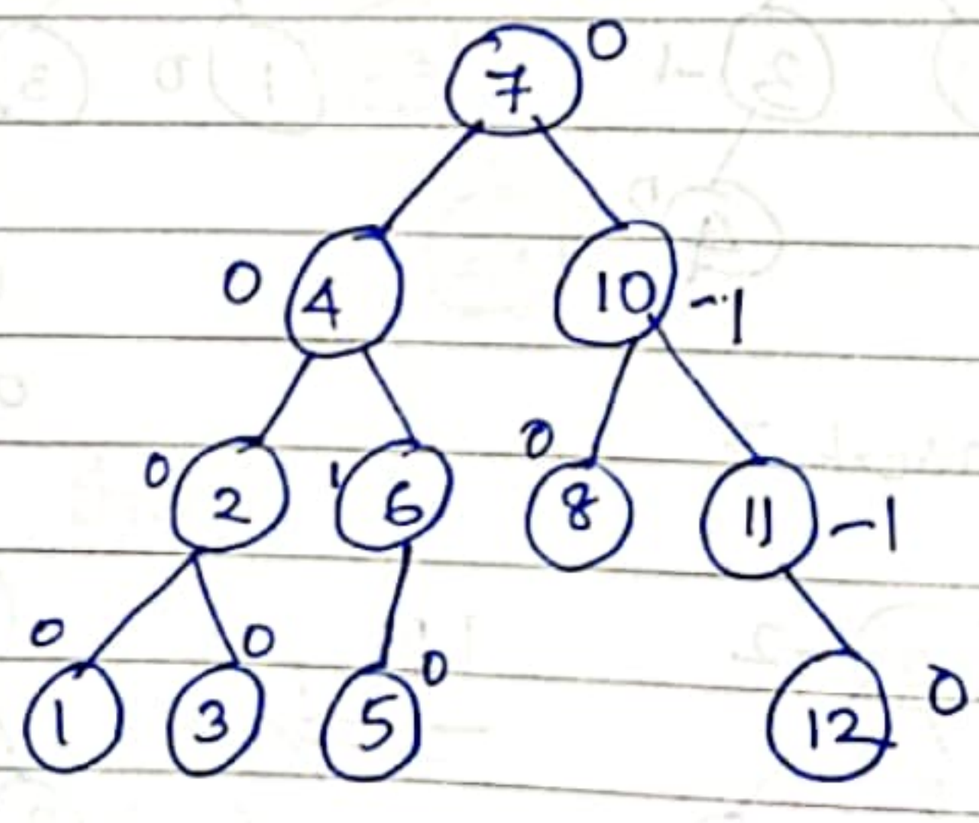



Insert 12




RR.

⇒

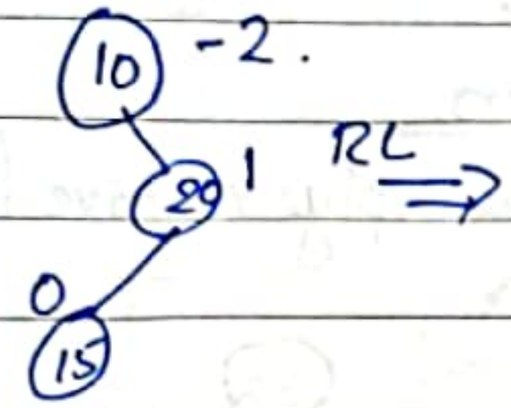


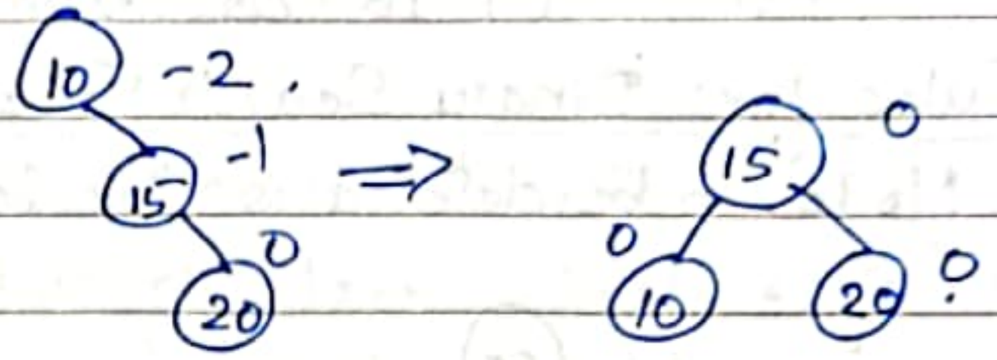
(3) Insert 10, 20, 15, 25, 30, 16, 18, 19 into an AVL. tree.

Insert 10 ⇒ (10)

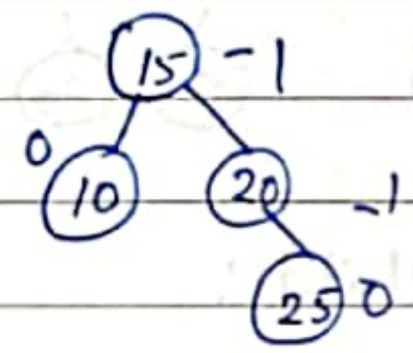
Insert 20 ⇒ (10) 0 — (20) -1

Insert 15

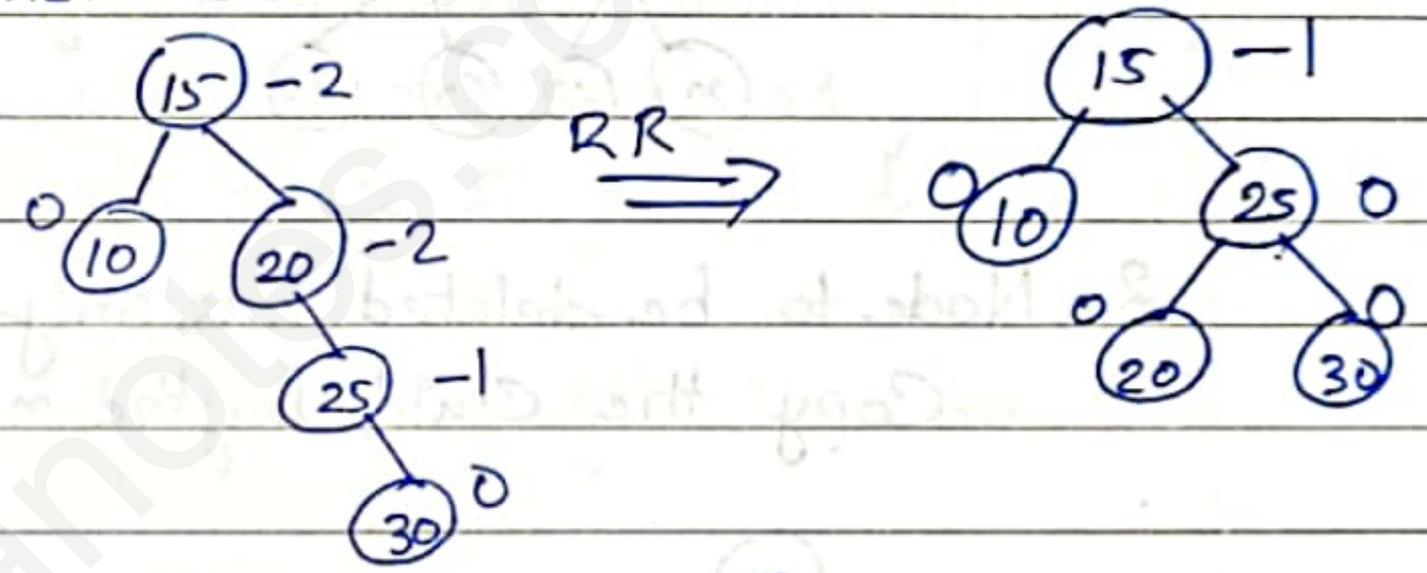


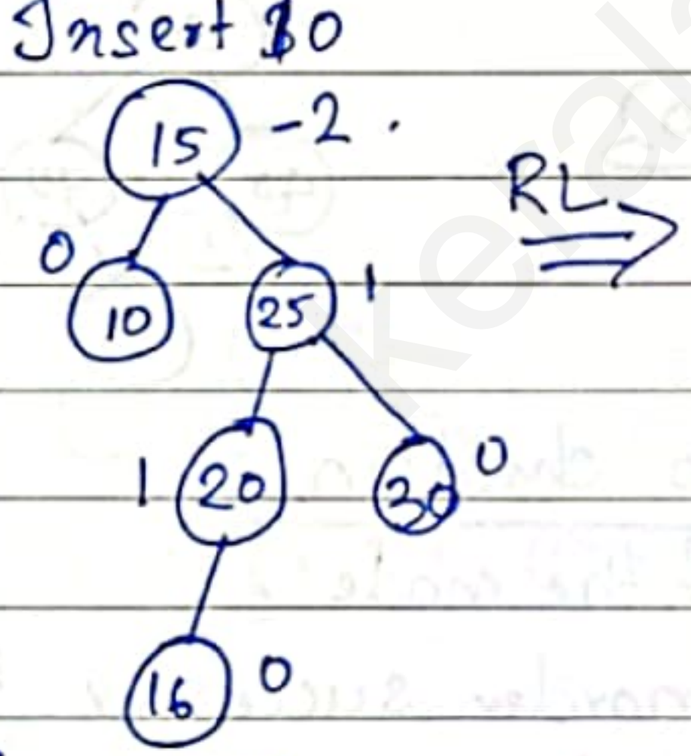

Insert 25

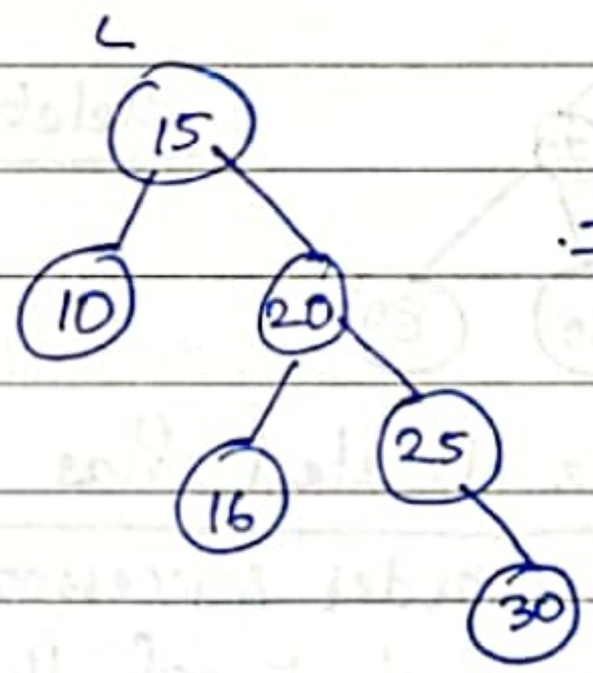


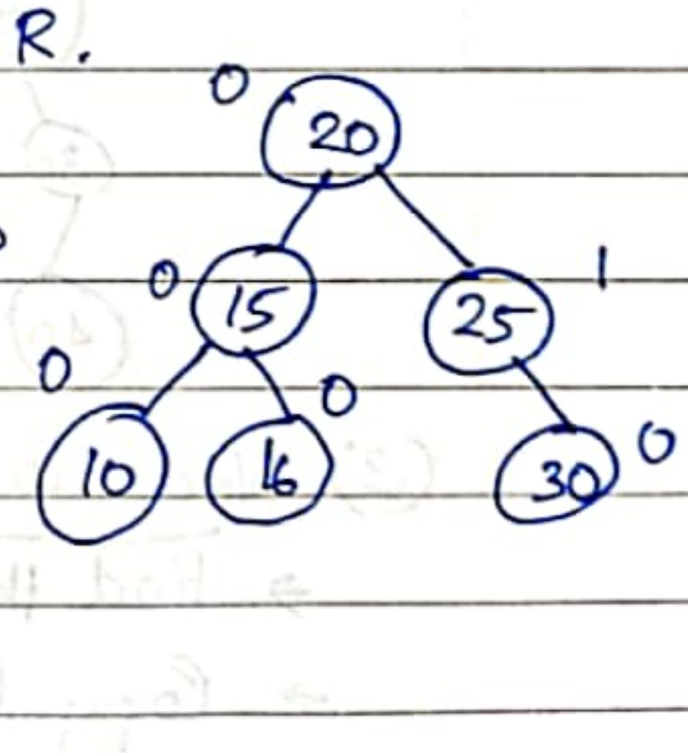

Insert 30




Insert 10




Insert 18




Insert 19

## Deletion in an AVL Tree.

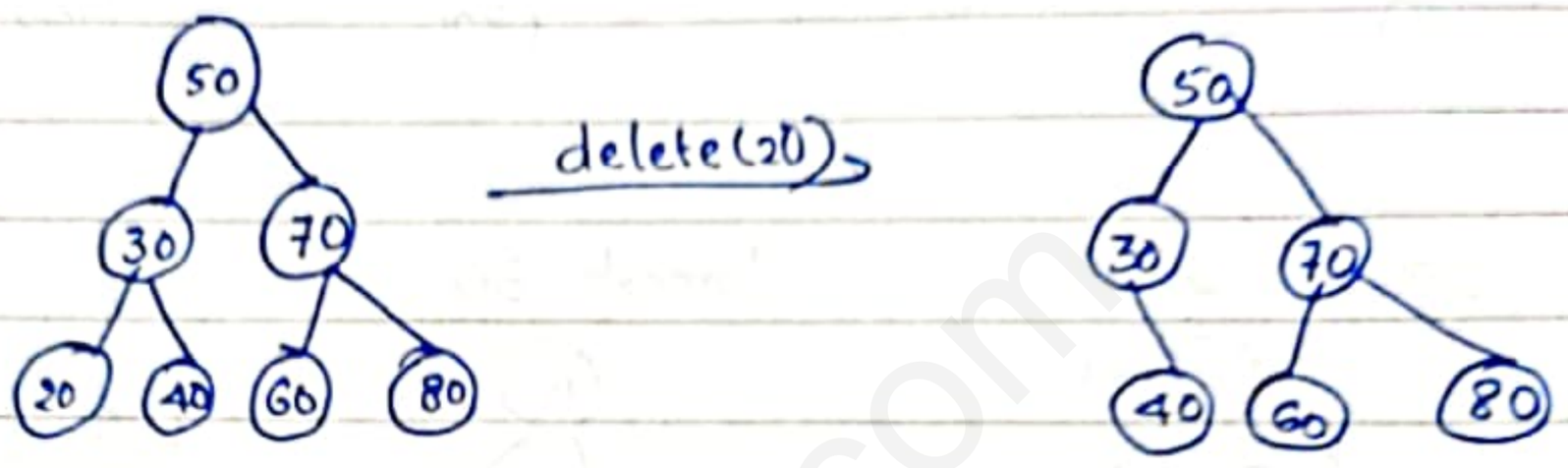
→ Deleting a node from an AVL tree follows the same rule of BST deletion.

→ Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced inorder to maintain it as an AVL tree.

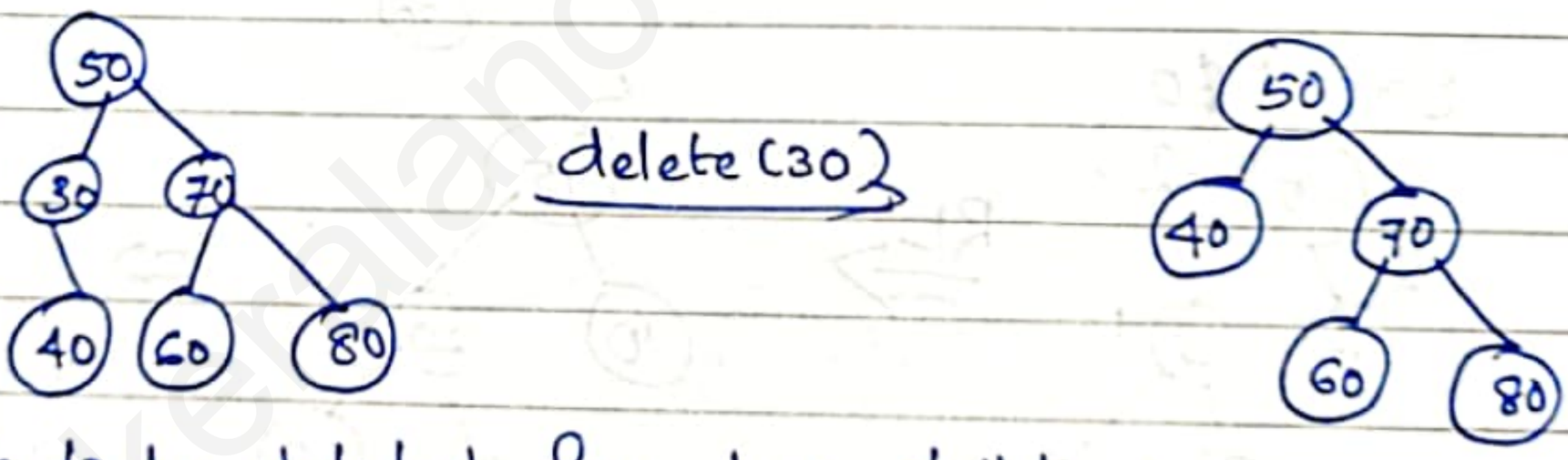
### Rules for Binary Search Tree Deletion

1. Node to be deleted is the leaf: Simply remove from the tree

eg:-




2. Node to be deleted has only one child:
   Copy the child to the node & delete the child.

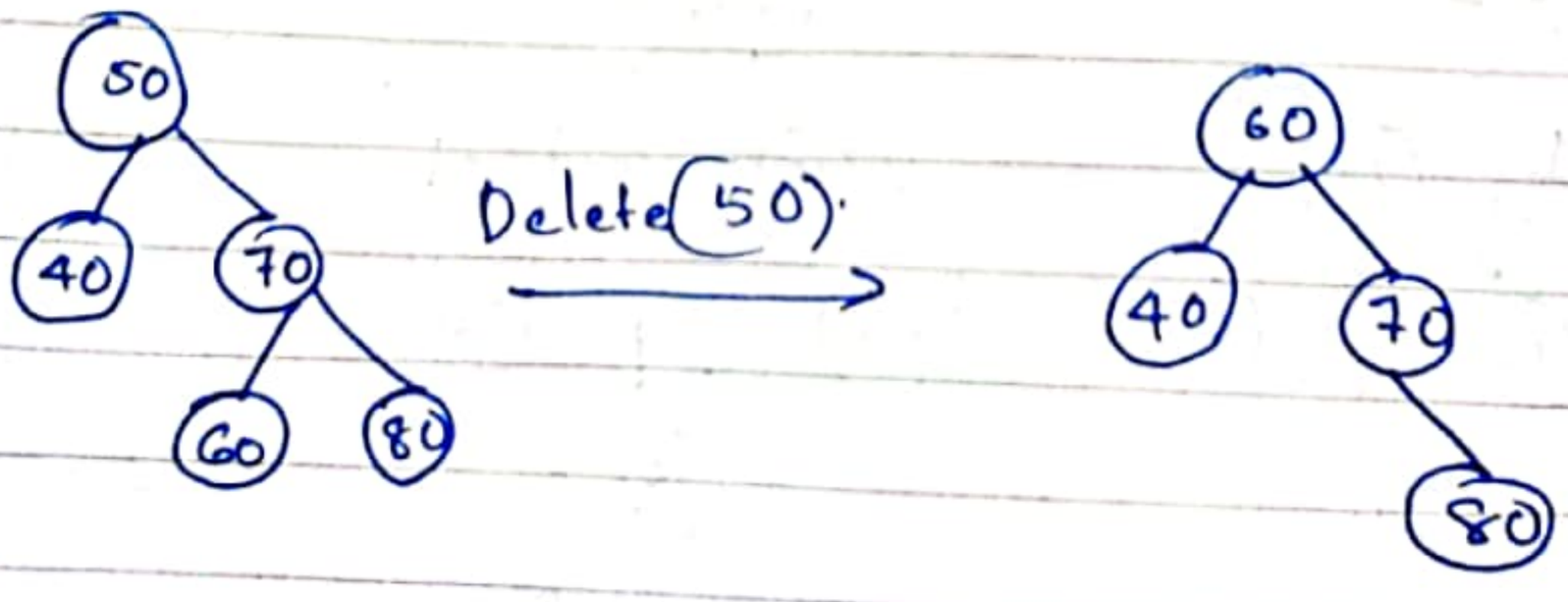



(3) Node to be deleted has two children.

→ Find the inorder successor of the node.

→ Copy the contents of the inorder successor to the node & delete the inorder successor.

→ Note that inorder predecessor can also be used.

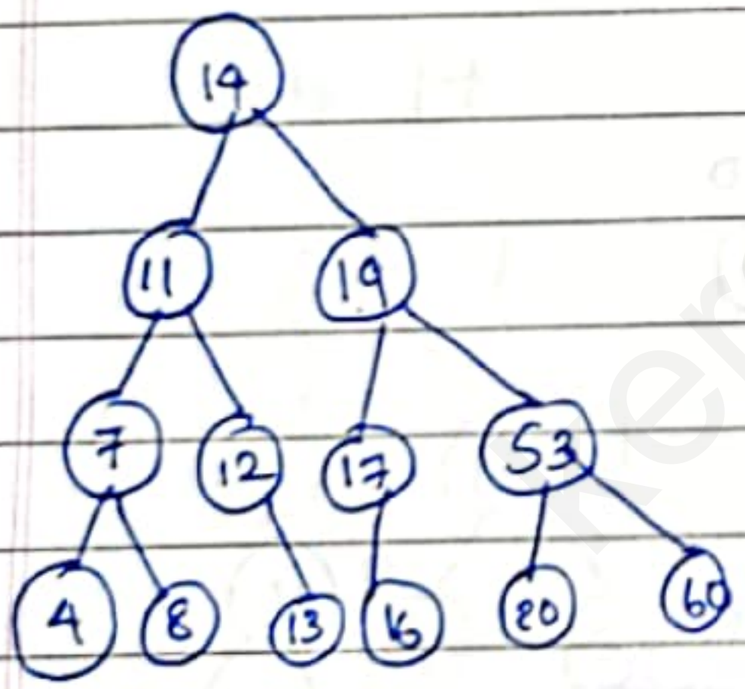


Inorder predecessor of a node in a BST = largest node in the left subtree.
Inorder Successor → Smallest node in the right subtree.

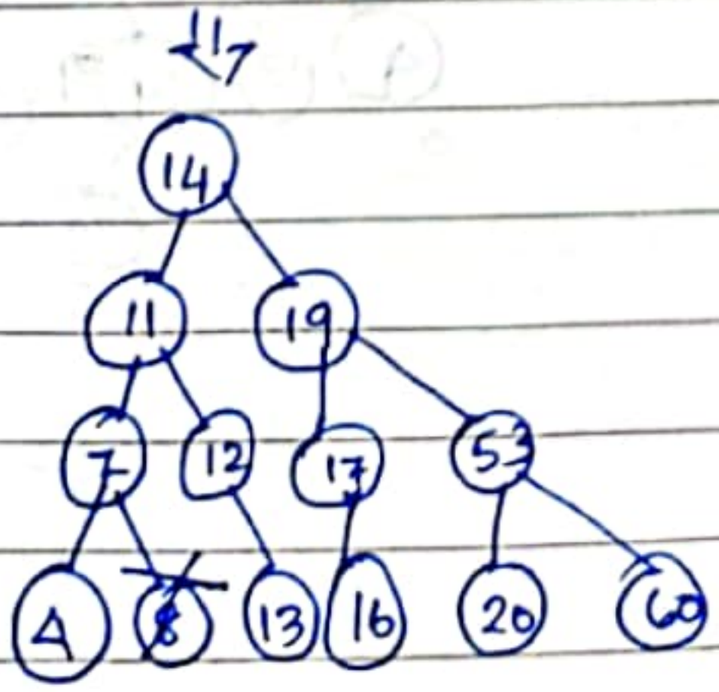
Steps in AVL tree deletion

1. Search the node which is to be deleted.
Cases: If the node to be deleted is not a leaf node then
Simply delete that node. ie if x is a leaf delete x
case2: If x has one child, use it to replace x.
case3: if x has two children, replace x with its inorder successor or inorder predecessor. Then delete
2. Traverse back up the path towards the root node checking balance factor of every node along the path.
3. If the B.F of a node is not -1, 0, or 1, then balance the tree using appropriate single or double rotations.

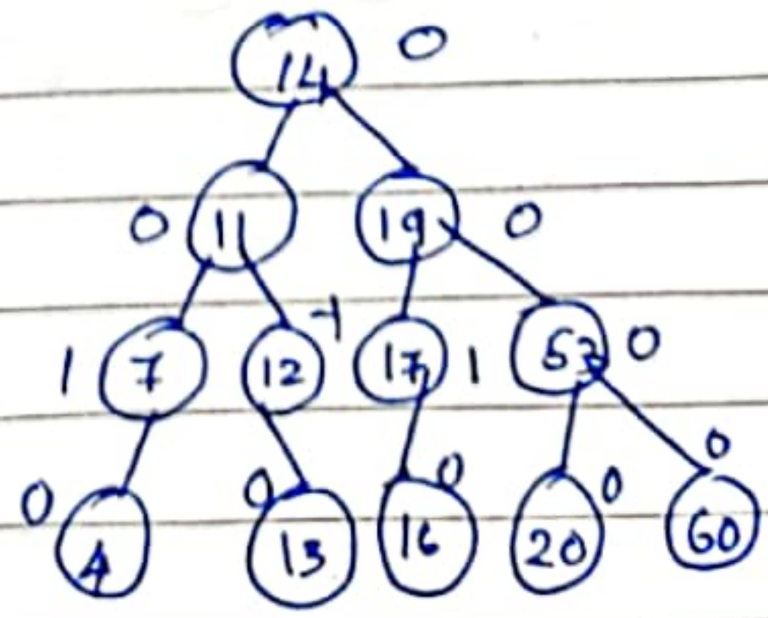
Q Delete 8, 7, 11, 14, 17 from the gol. AVL tree.



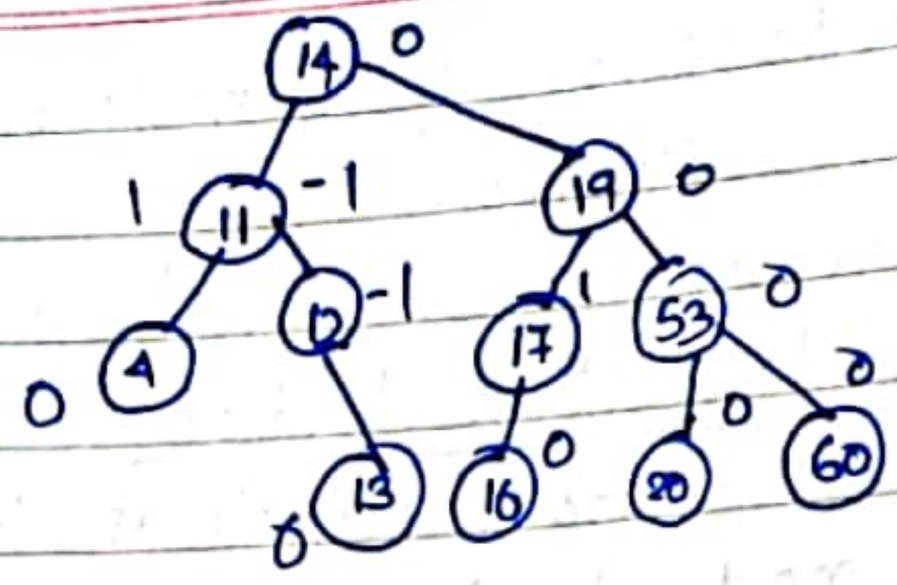

Soln. Delete 8




⇒



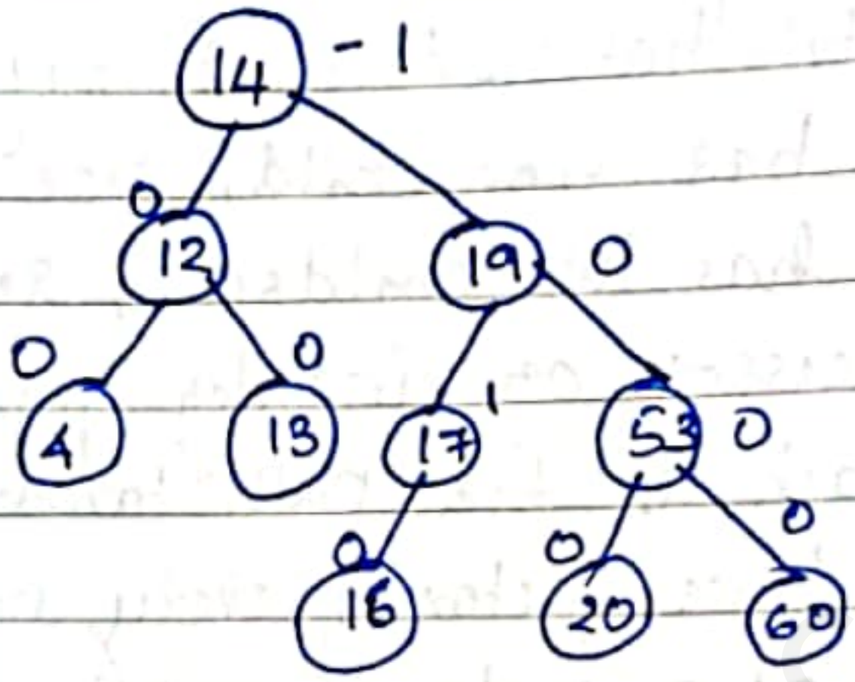
Delete 7 =>




⇒

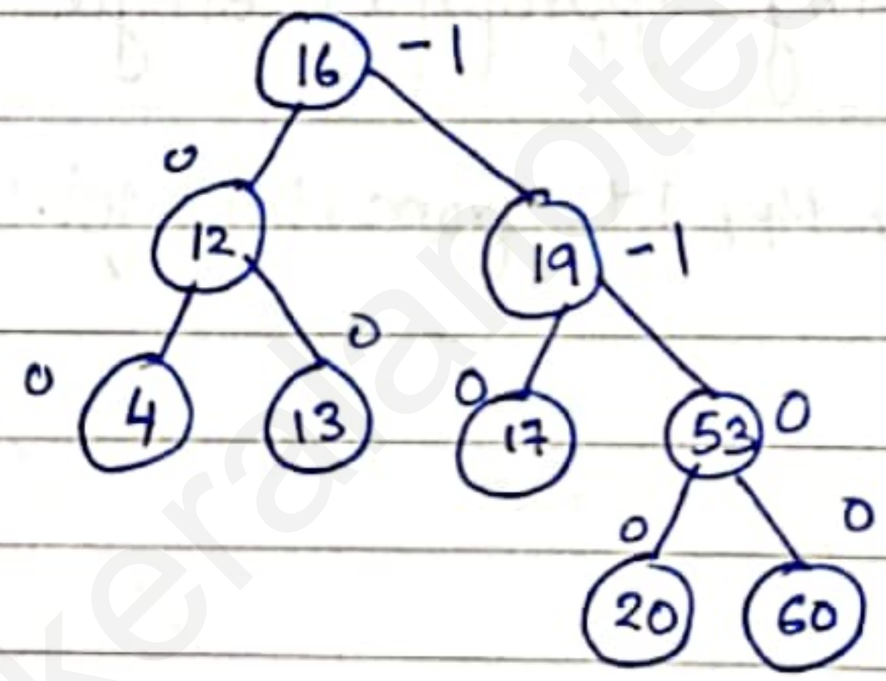
Node 11 has 2 children.
Take inorder successor of 11

Delete 11

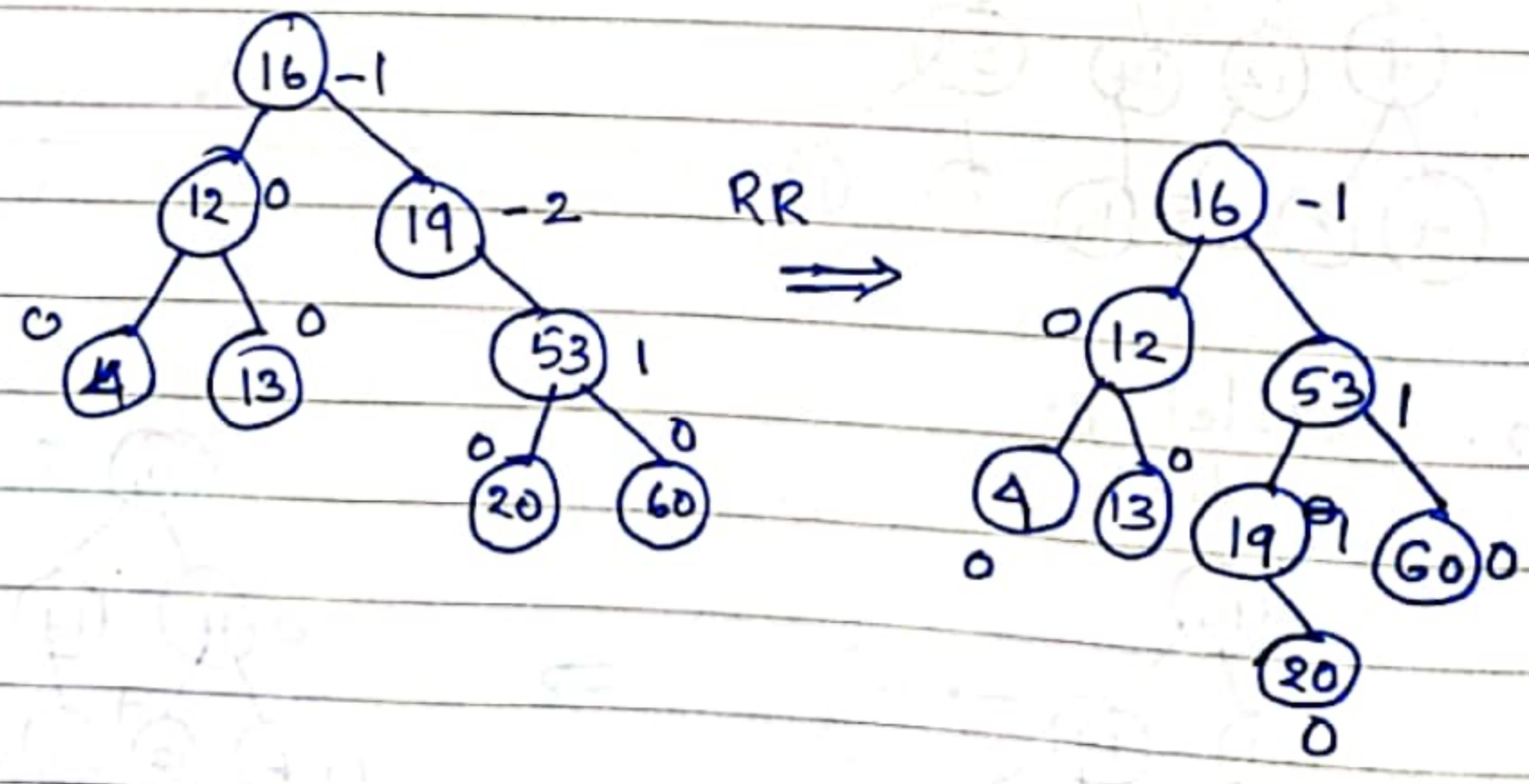



Delete 14.

Take inorder successor of 14 => 16.



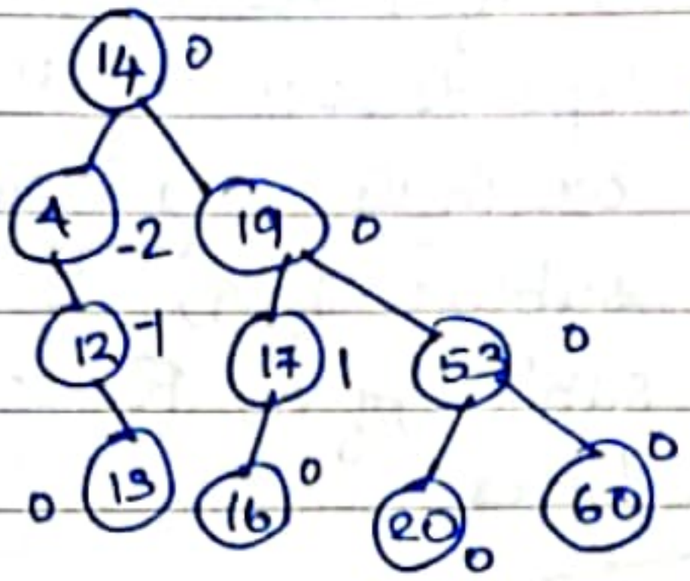

Delete 17.



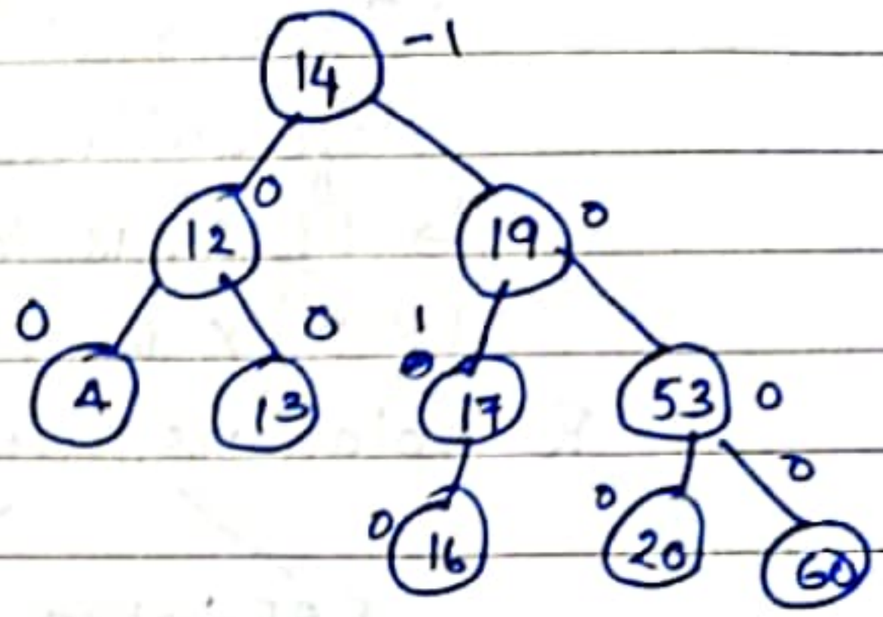
2nd soln.

Delete 11 ⇒ If we take inorder predecessor.

⇒

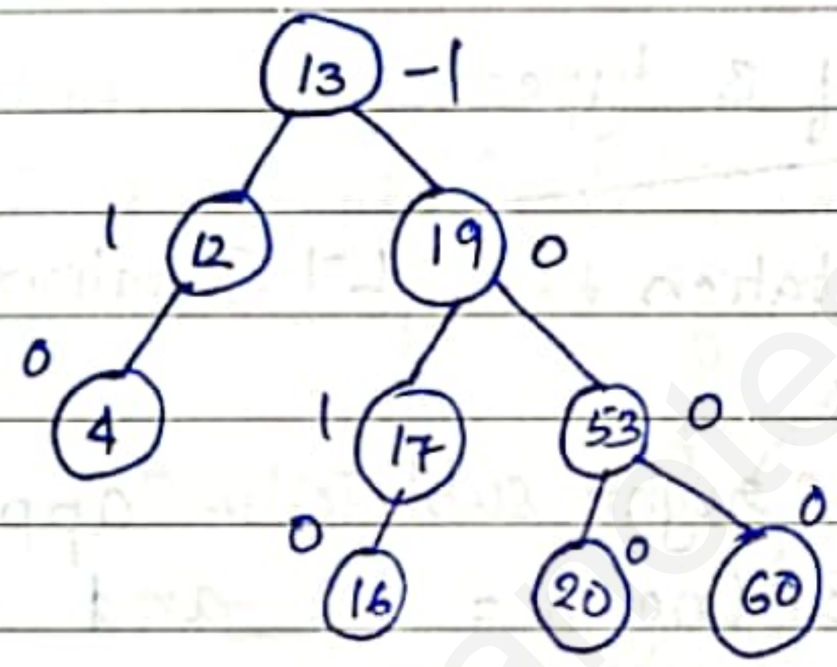



RR. ⇒

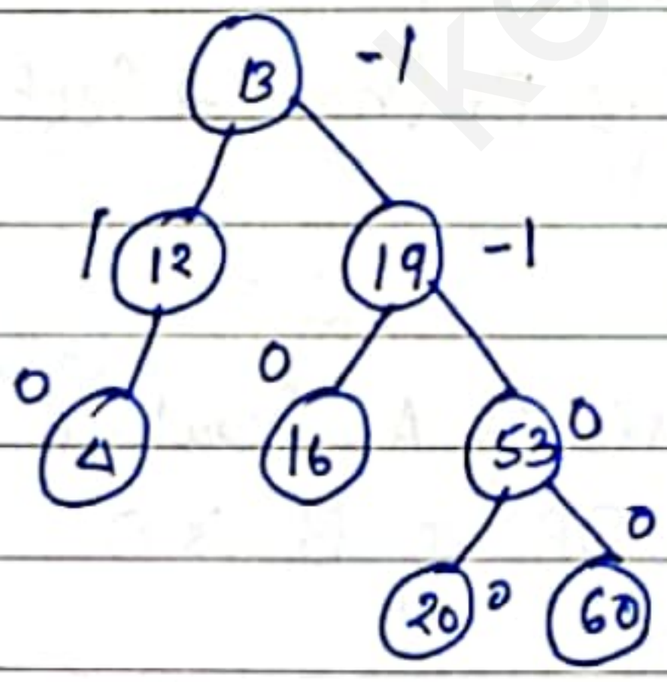



Take inorder predecessor of 14

Delete 14 ⇓




Delete 17



## AVL Tree Deletion - Rotations

⇒ On deletion of node X, if node A becomes critical (BF=±2) node
  ↳ Type of rotation depends on whether X is in the left subtree of A or in its right subtree.
  ↳ If X is in left subtree of A, L rotation is applied
  ↳ If X is in right subtree of A, R rotation is performed

R-rotations are of 3 types

- R0 Rotation
- R1 Rotation
- R-1 Rotation.

L-Rotations are of 3 types

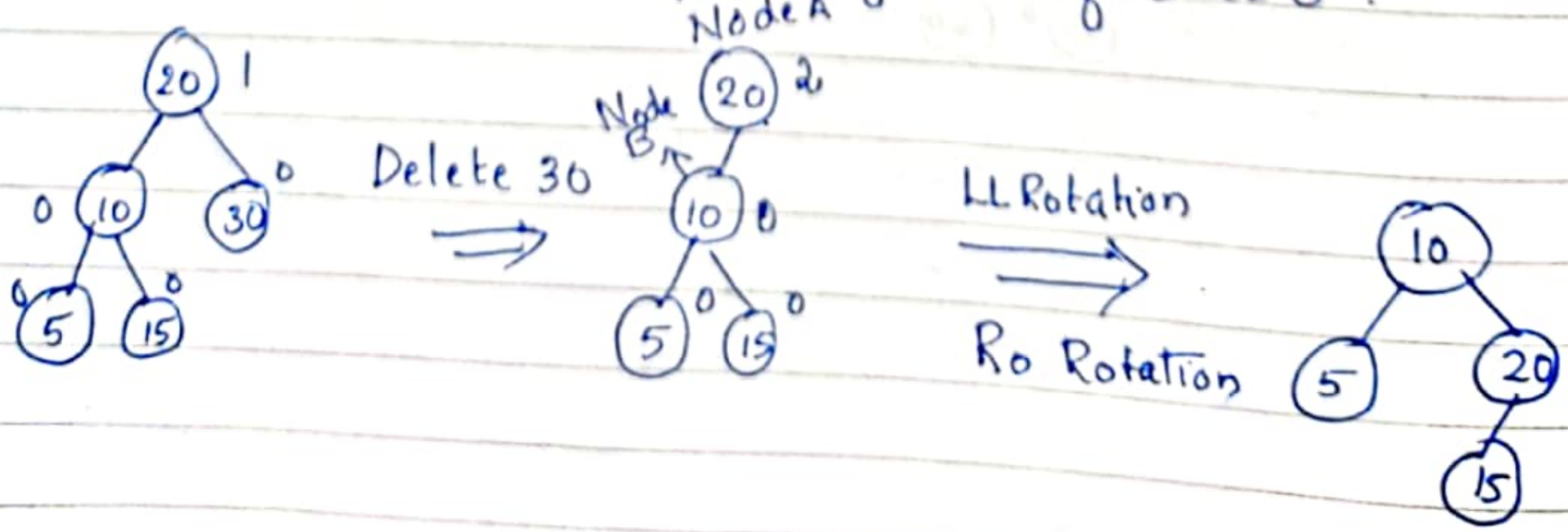
- L0 Rotation
- L1 Rotation
- L-1 Rotation.

Case 1: Deleting from right subtree - apply R Rotations.

* R0 Rotation: If BF(node) = +2 and BF(node→left child) =0 ⇒ Similar to LL Rotation.
* R1 Rotation :- If BF(node) = +2 & BF(node→left child) :-1 ⇒ Similar to LL Rotation.
* R-1 Rotation :- If BF(node) = +2 & BF(node→ left child) :-1 ⇒ Similar to LR Rotation.

R0 Rotation:

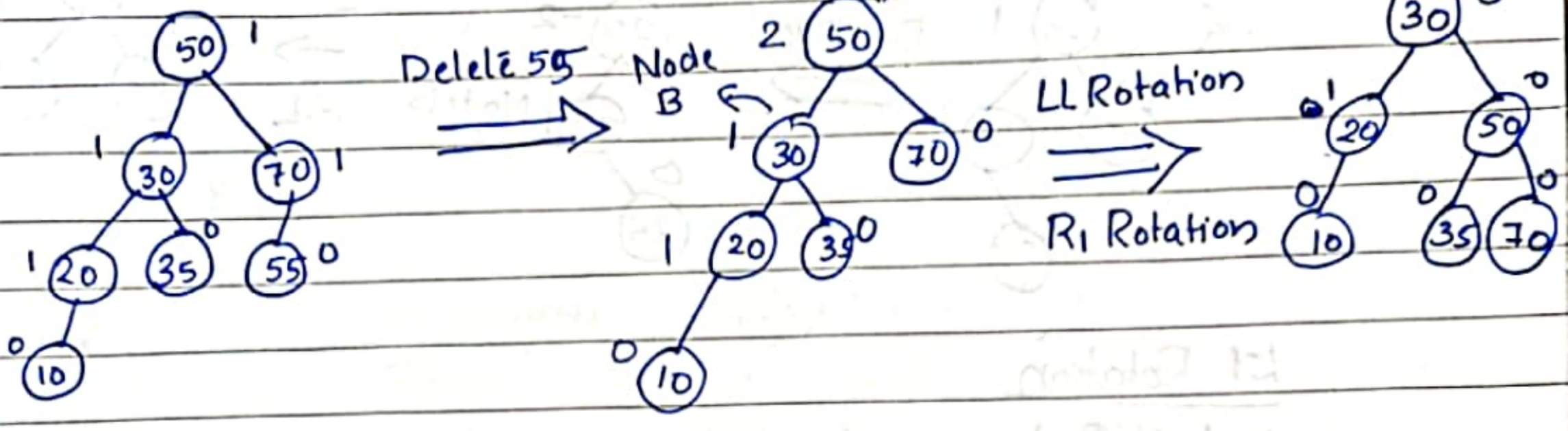
→ Let B be root of left subtree of A (critical node) (BF=±2).
→ R0 Rotation is applied if BF of B is 0.



## R1 Rotation

→ Let B be root of left subtree of A (critical node)

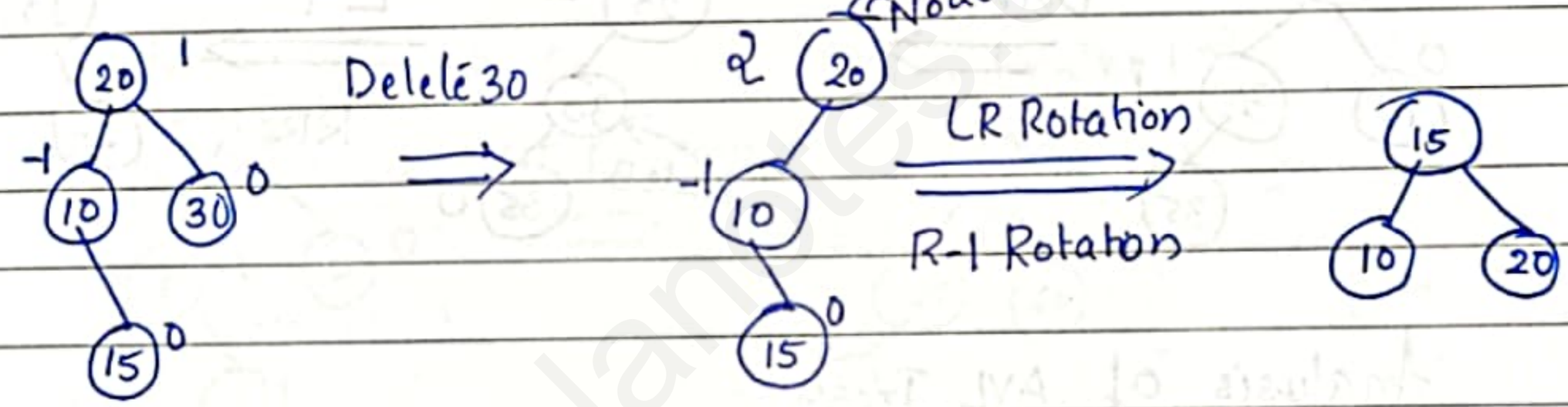
⇒ R1 rotation is applied if balance factor of B is 1.




## R-1 Rotation

Let B be root of left or right subtree of A (Critical node)

R-1 Rotation is applied if balance factor of B is -1

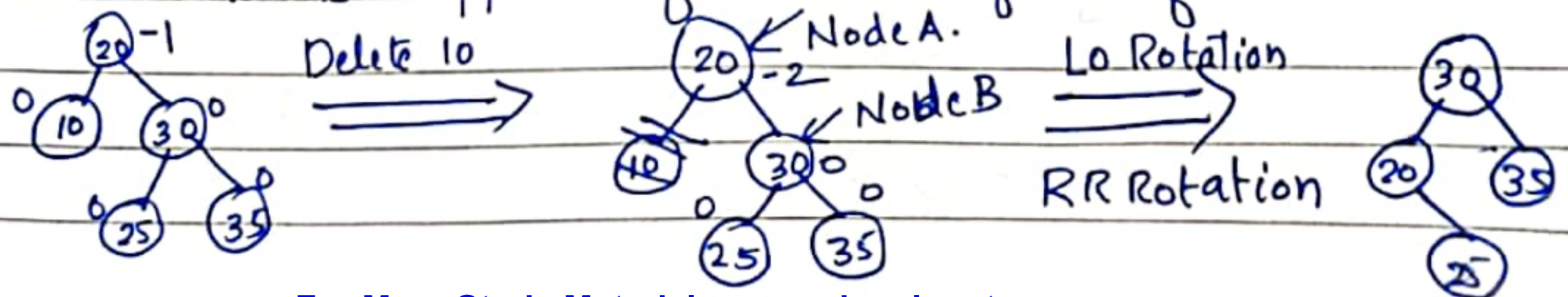



Case 2:- Deleting from left subtree - apply L Rotations.

* LO Rotation :- If BF(node) = -2 & BF(node → right child) = 0
  ⇒ Similar to RR Rotation.
* L1 Rotation :- If BF(node) = -2 & BF(node → right child)
  = +1 ⇒ Similar to RL Rotation.
* L-1 Rotation :- If BF(node) = -2 & BF(node → right child) = -1
  ⇒ Similar to RR Rotation.

## LO Rotation

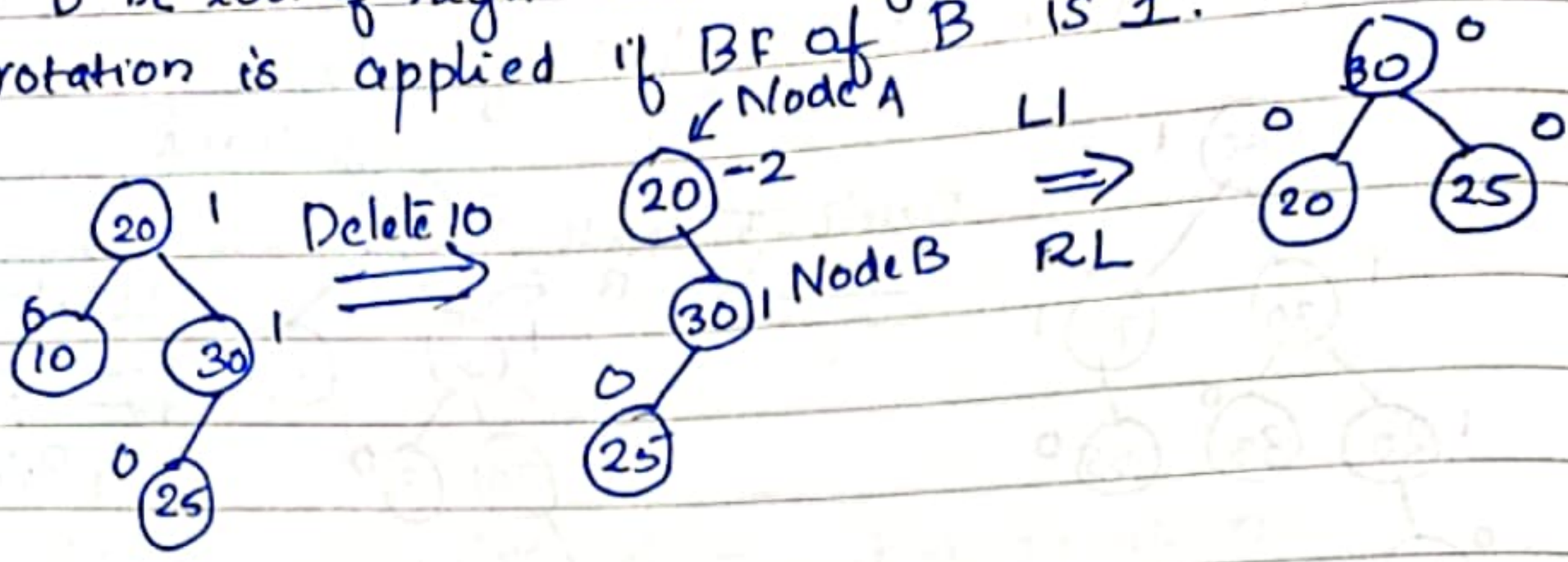
* Let B be the root of right subtree of A (critical node)
* LO Rotation is applied if balance factor of B is 0.



## LI Rotation :.

⇒ Let B be root of right subtree of A (critical node)

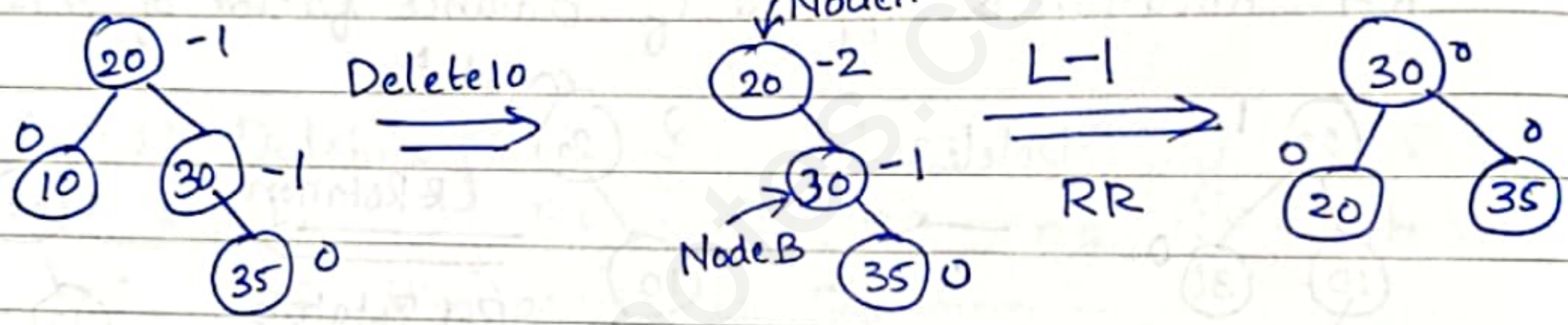
⇒ LI rotation is applied if BF of B is 1.




## L-1 Rotation

⇒ Let B the root of right subtree of A (critical node)

⇒ L-1 rotation is applied if BL of B is -1




## Analysis of AVL Trees

→ The insertion of a node is going to take O(logn) time because the tree is balanced.

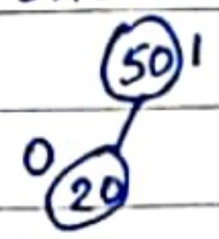
⇒ Also, the unbalances in the insertion process get fixed after a fixed number of rotations. So the entire process is O(logn) time.

→ We can delete any node in constant time ie O(1) & also fix the unbalance with a fixed number of rotation in O(1) time but since the unbalance might propogate above the tree, the entire deletion process takes O(logn) time.

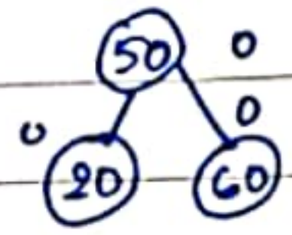
Q. Construct AVL tree for the fol. sequence of numbers
50, 20, 60, 10, 8, 15, 32, 46, 11, 48.

Insert 50

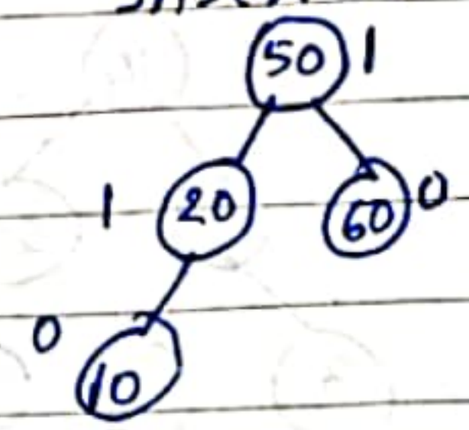

Insert 20



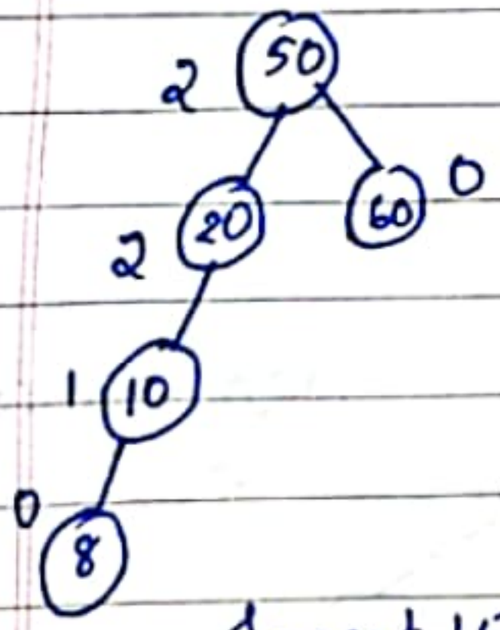

Insert 60



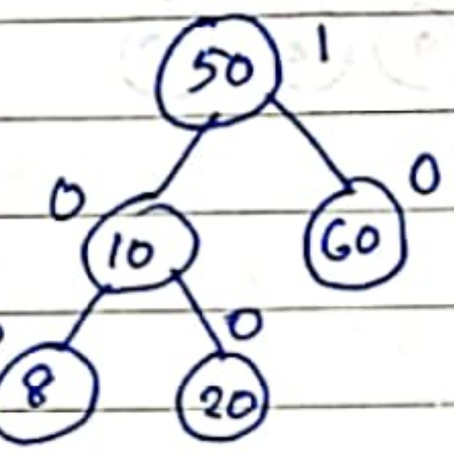

Insert 10



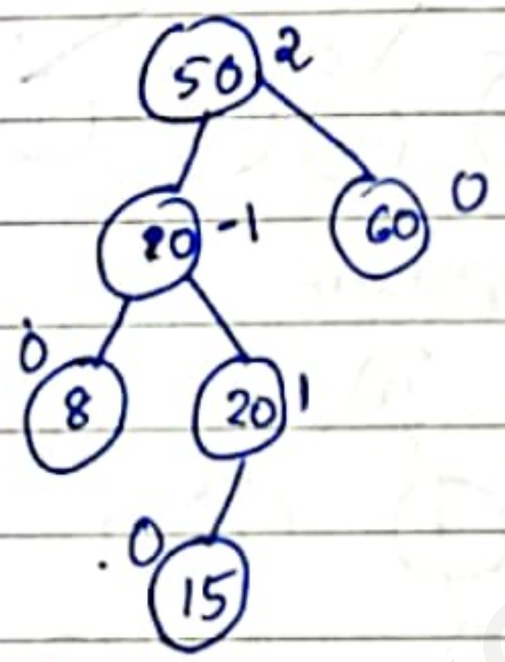

Insert 8



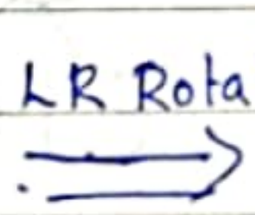

LL Rotation



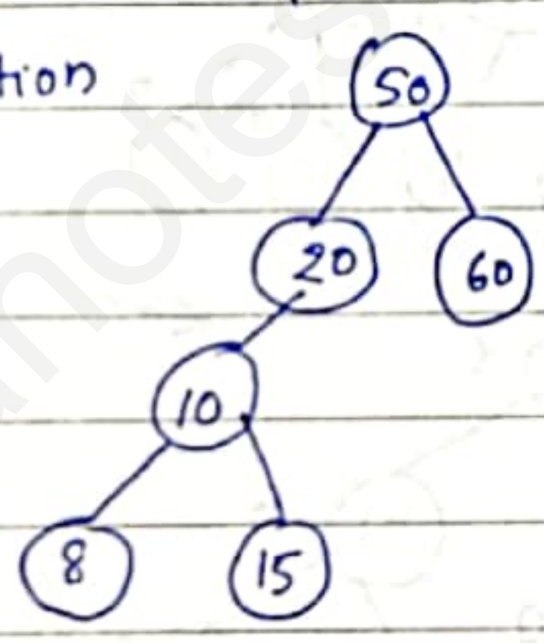

Insert 15



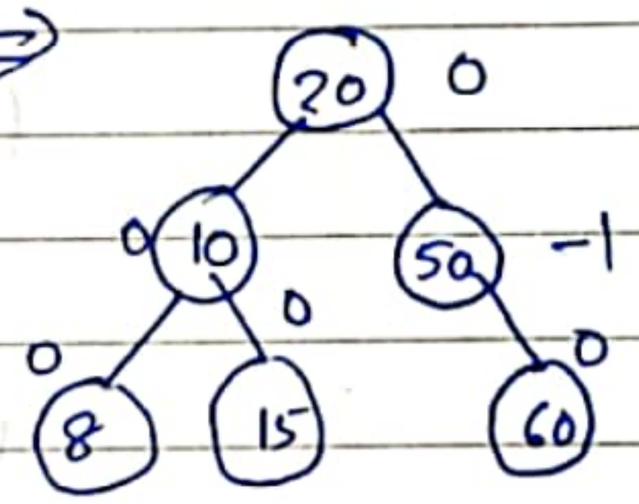

LR Rotation



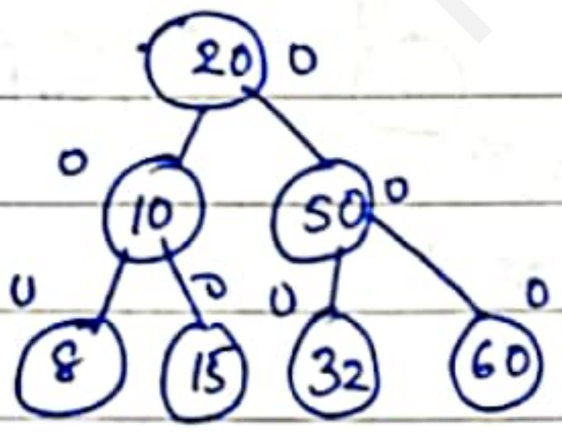
RR Rotation (10)



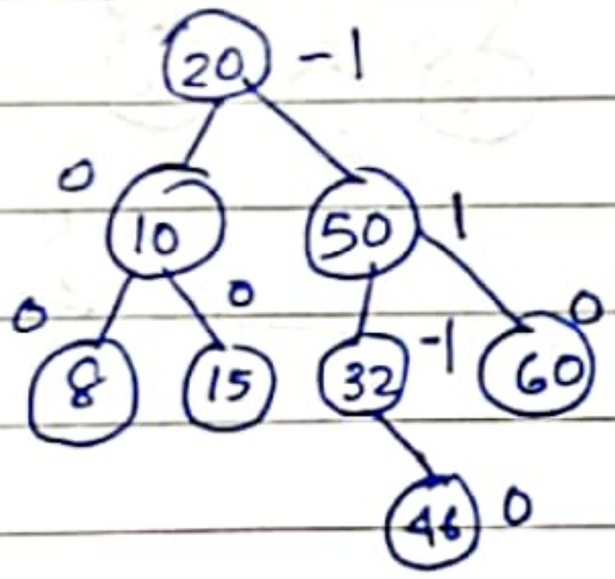

LL Rotation (50)



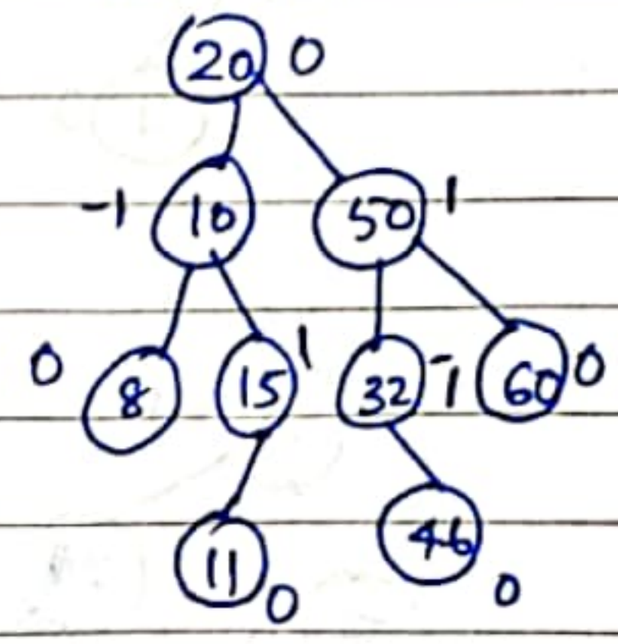

Insert 32.



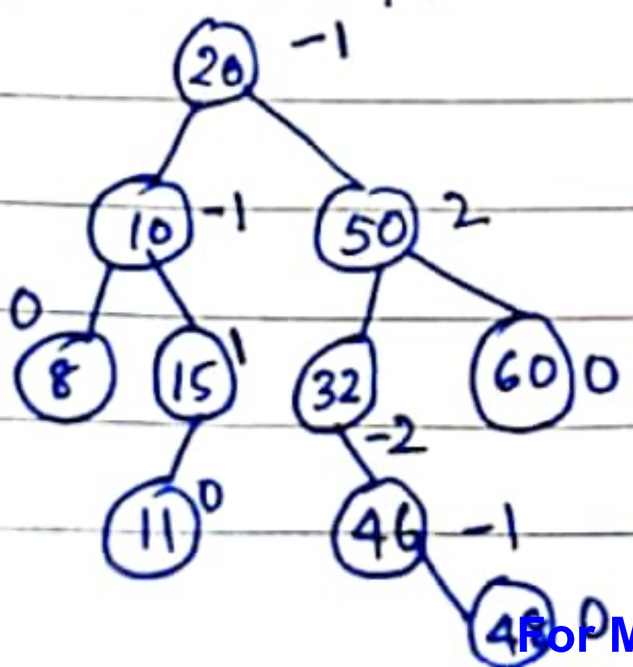

Insert 46



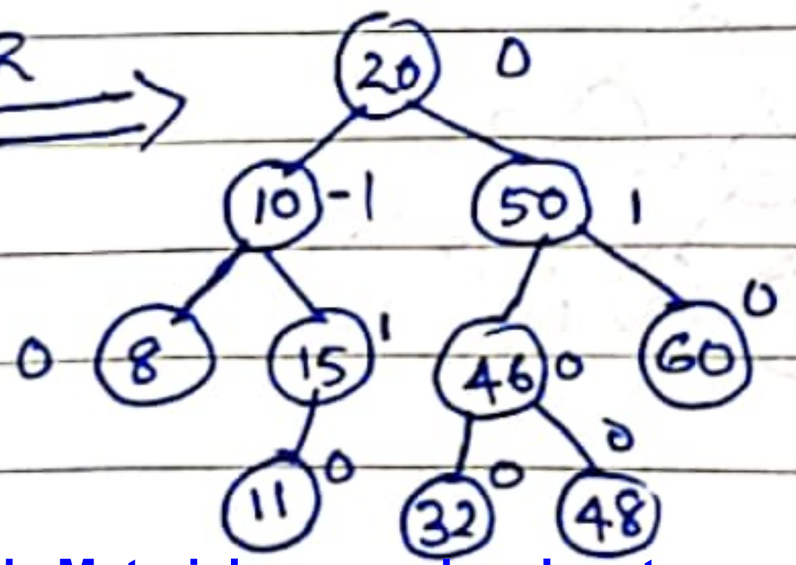

Insert 11




Insert 48

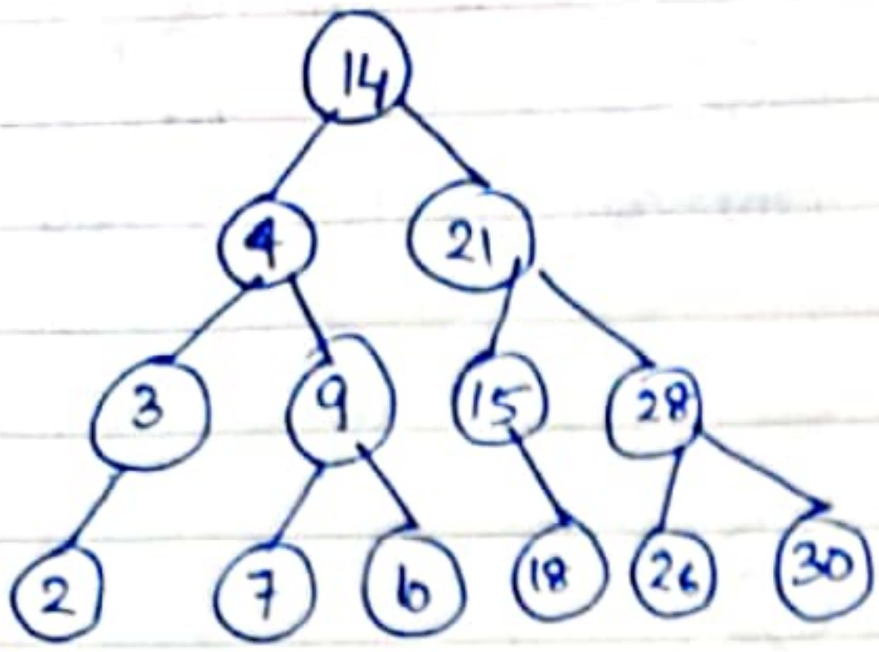



RR

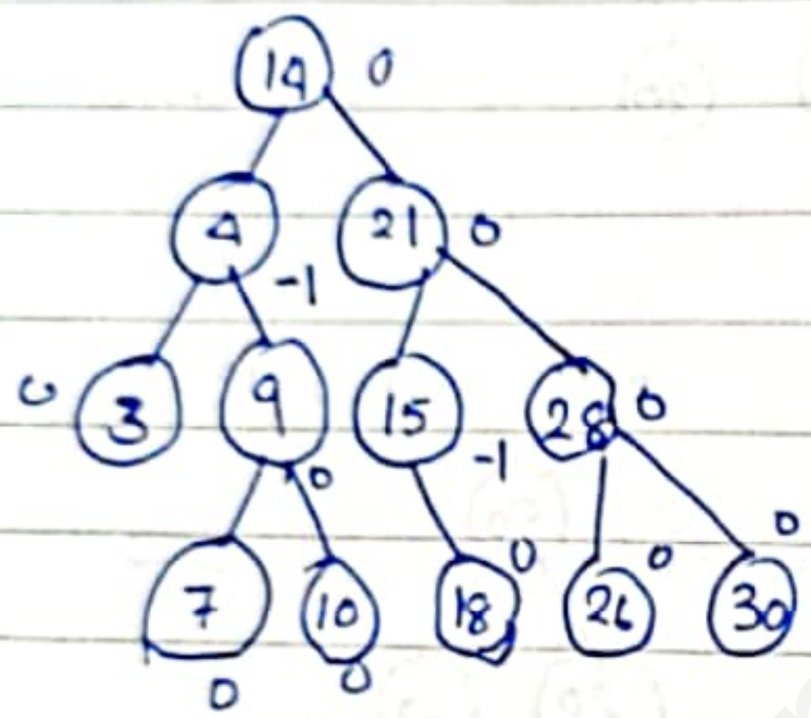


Q. Delete the fol. nodes in sequences in the fol AVL tree.

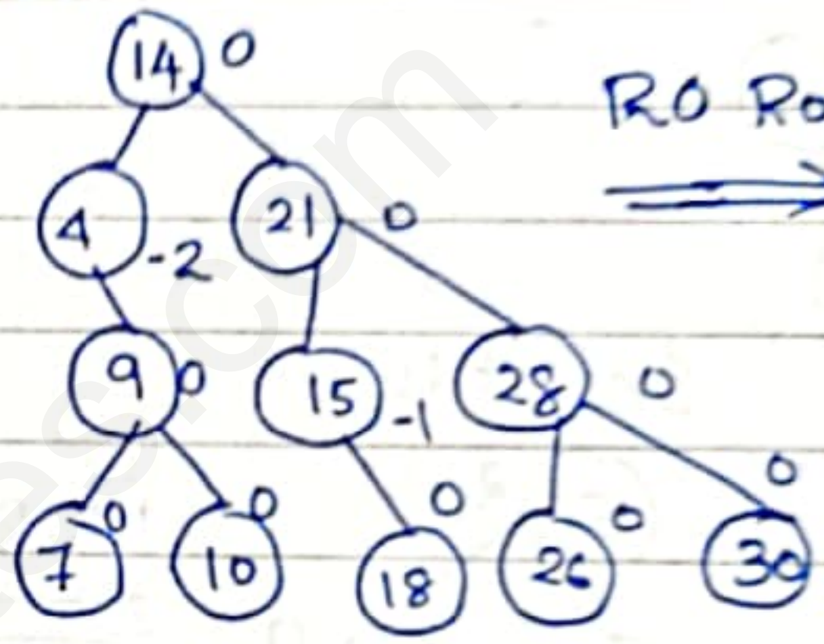
2, 3, 10, 18, 4, 9, 14, 7, 15




Delete 2

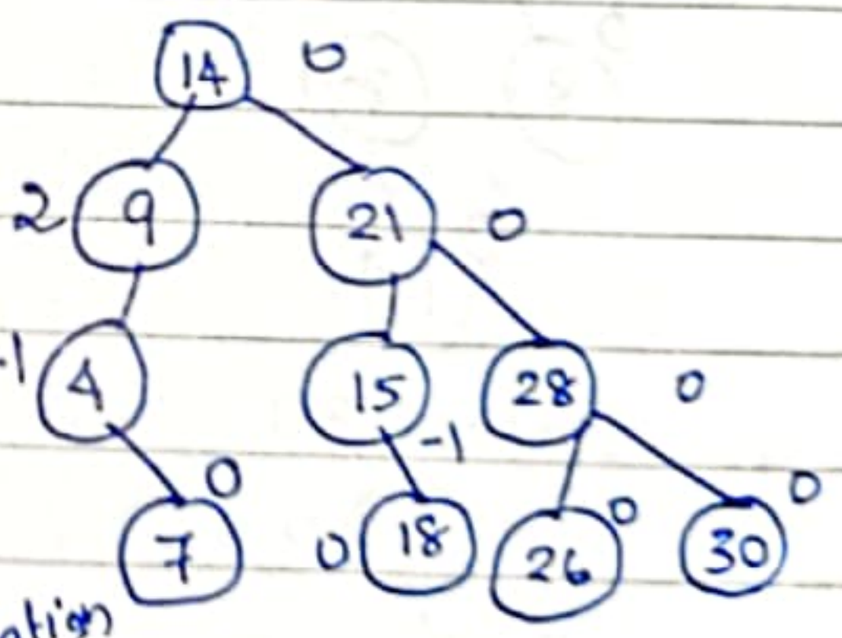



Delete 3



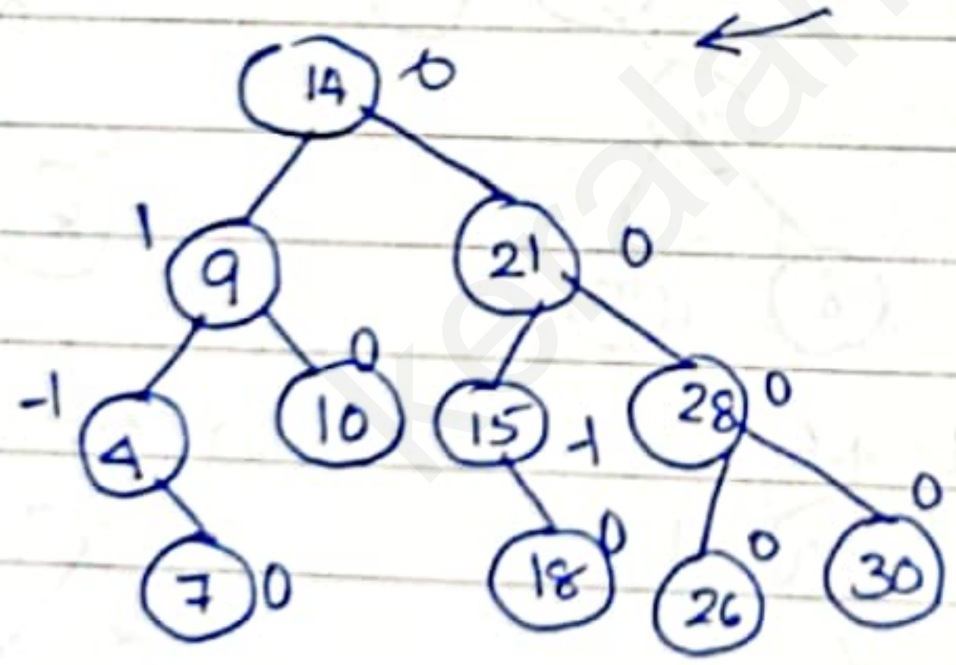

RO Rotation

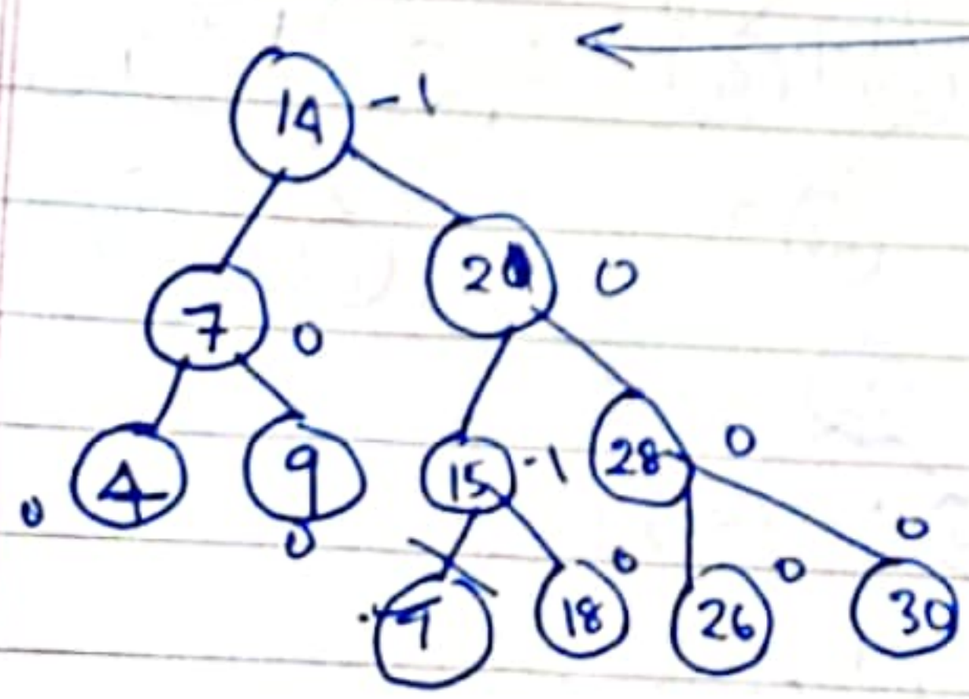
Delete 10



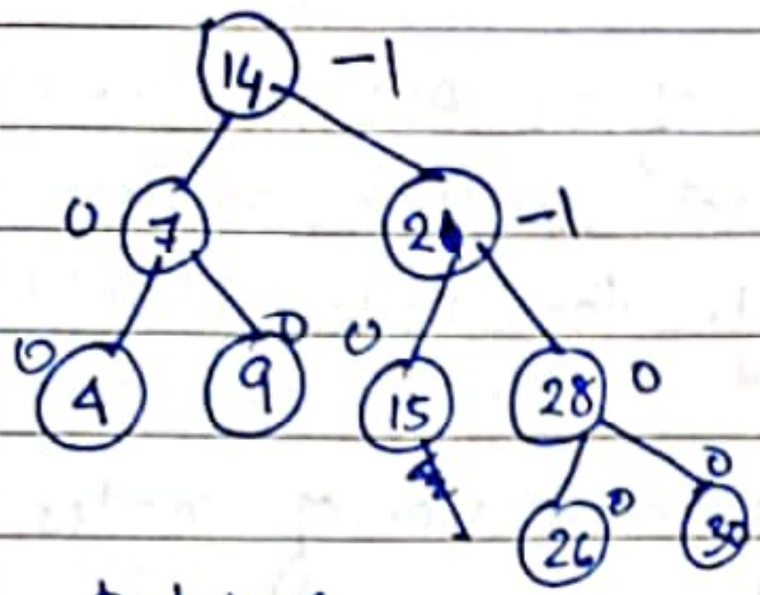

R-1 Rotation

LR rotation

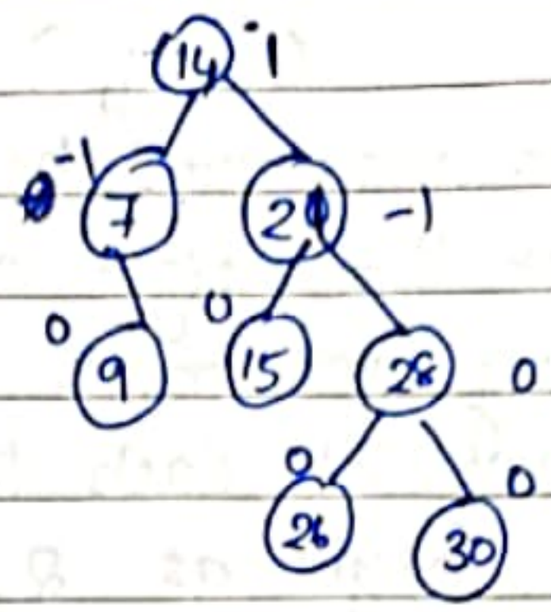


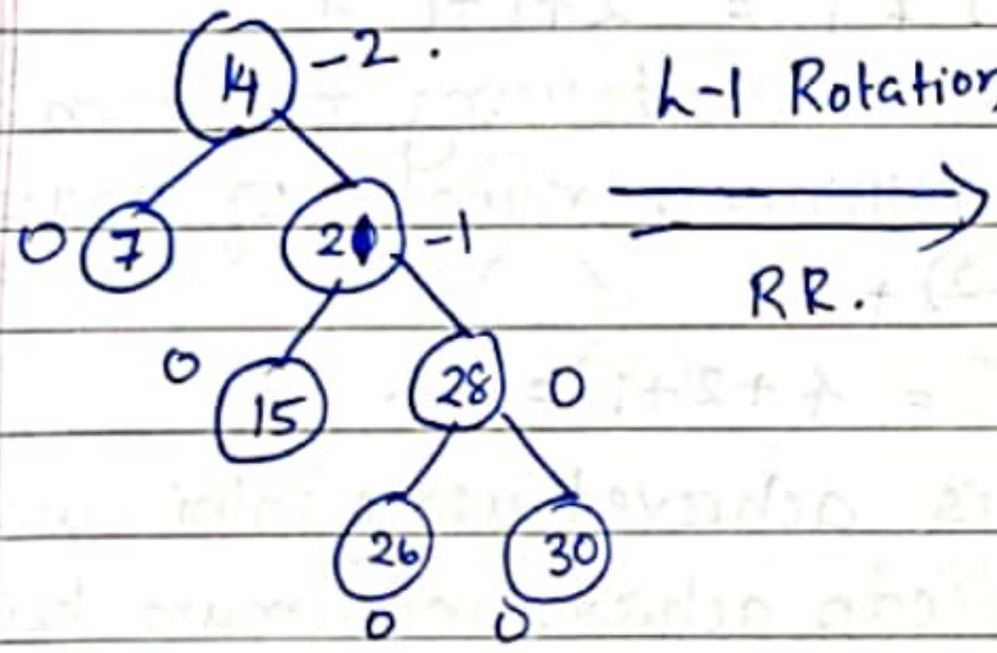
Delete 18




Delete 4



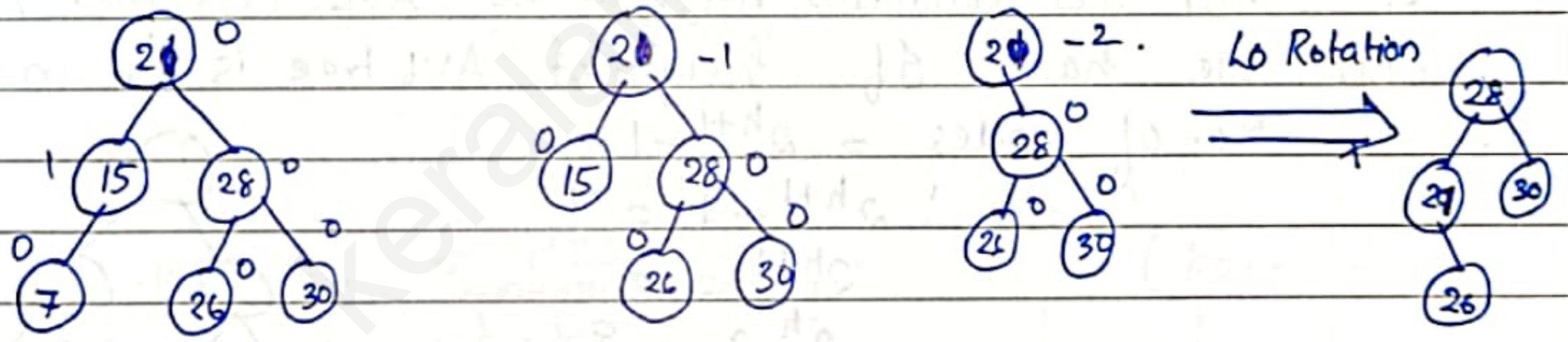

Delete 9




Delete 14

Delete 7

Delete 15




Key points on AVL trees

→ If there are n nodes in AVL tree, minimum height of AVL tree is floor( $\log_2 n$ ).

→ If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.

→ If height of AVL tree is h, maximum no. of nodes can be $2^{h+1} - 1$

→ Minimum number of nodes in a tree with height h can be represented as

$N(h) = N(h-1) + N(h-2)$ for $n \geq 2$ where $N(0) = 1$ & $N(1) = 2$

→ The complexity of searching, inserting & deletion in AVL tree is $O(\log_2 n)$.

Q. What is the maximum height of an AVL tree with 7 nodes? Assume that the height of a tree with a single node is 0.

Soln For finding maximum height, the nodes should be at minimum at each level.

Assuming height as 2, minimum no. of nodes required

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

$N(2) = N(1) + N(0) + 1 = 2 + 1 + 1 = 4$

It means, height 2 is achieved using minimum 4 nodes.

Assuming height as 3, minimum number of nodes req:

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

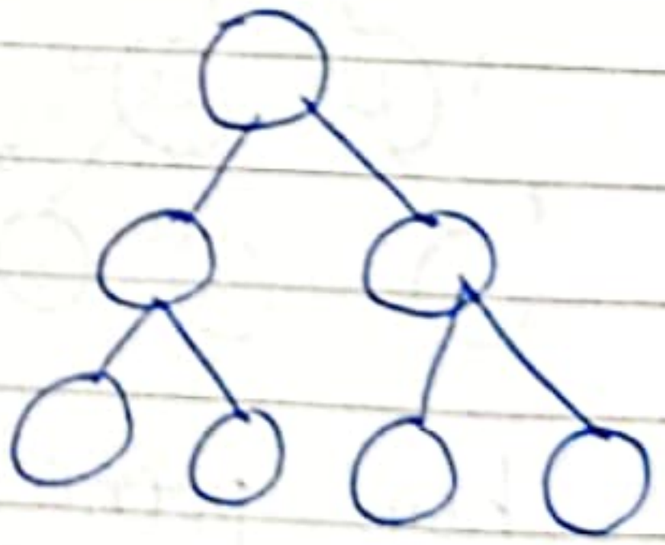
$N(3) = N(2) + N(1) + 1 = 4 + 2 + 1 = 7$.

It means, height 3 is achieved using minimum 7 nodes.

∴ Using 7 nodes, we can achieve maximum height as 3

Q. Find the minimum height of the AVL tree with 7 nodes.

Soln We have, If height of AVL tree is h, maximum no. of nodes $= 2^{h+1} - 1$



$\therefore 2^{h+1} - 1 = 7$

$2^{h+1} = 8$

$2^h \cdot 2 = 8$

$2^h = 4$

$\therefore h = 2$ //

∴ Minimum height of AVL tree with 7 nodes = 2.

## Topological Sorting

Topological sorting for a Directed Acyclic graph (DAG) is a linear ordering of vertices such that for every directed edge (u,v), vertex u comes before v in the ordering.

→ Topological sorting for a graph is not possible if the graph is not a DAG.

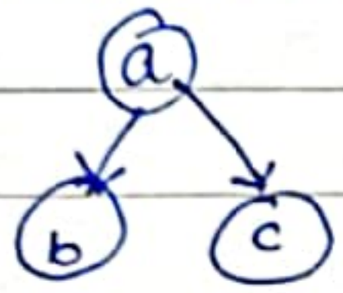
→ eg:-


Topological sorting — a, b.

There can be more than one topological sorting for a graph.

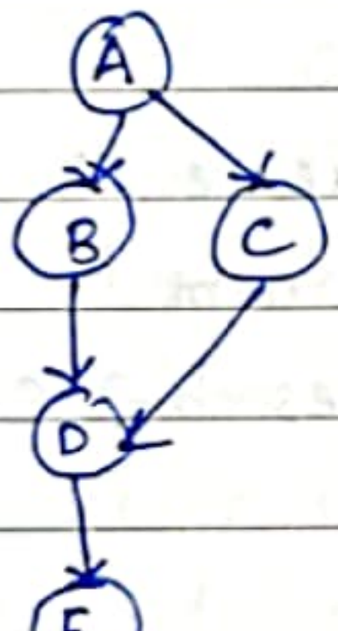
eg:-

(1)




Topological sorting - a b c
or
a c b.

(2)

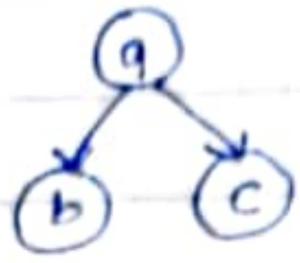



A B C D E or A C B D E.

## Algorithms for Topological Sorting (Kahn's Algorithm)

```
Topological sort (G)
{
    Store each vertex's indegree in an array.
    Initialize a queue with vertices having indegree zero.
    while there are vertices remaining in the queue
    {   Dequeue and o/p a vertex.
        Decrement Indegree of all vertices adjacent to it by 1
        Enqueue any of these vertices whose
        Indegree became zero
    }
}
```

eg :-




Array

| | |
|---|---|
| a | 0 |
| b | 1 |
| c | 1 |

Queue

| |
|---|
| a. |

Dequeue a

Decrement Indegree of b & c by 1 (adjacent to a)

Enqueue b & c.

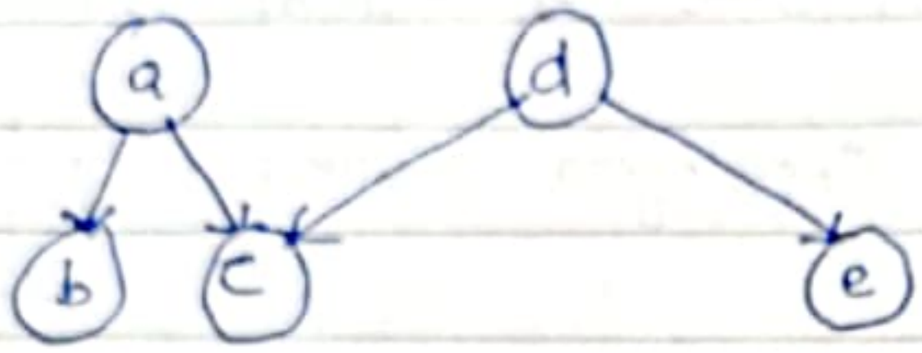
Enqueue:

| | |
|---|---|
| a | 0 |
| b | 0 |
| c | 0 |

| | |
|---|---|
| b | c |

Dequeue b and then c. ∴ O/P → a, b, c

eg:2.




Array

| | |
|---|---|
| a | 0 |
| b | 1 |
| c | 2 |
| d | 0 |
| e | 1 |

Queue

| | |
|---|---|
| a | d |

↑ Enqueue a, d

Dequeue a: ⇒ O/P → a

Decrement in degree of vertices b & c by 1

Enqueue b.

| | |
|---|---|
| a | 0 |
| b | 0 |
| c | 1 |
| d | 0 |
| e | 1 |

| | |
|---|---|
| d | b |

↗ O/P → a, d

Dequeue d & decrement indegree of vertices c & e by 1

Enqueue c, e.

| | |
|---|---|
| a | 0 |
| b | 0 |
| c | 0 |
| d | 0 |
| e | 0 |

| | | |
|---|---|---|
| b | c | e |

↗ O/P → a, d, b

↗ O/P → a, d, b, c

Dequeue b, Dequeue c, Dequeue & e

Topological sorting → a, d, b, c, e.

## Time Complexity of Topological Sort.

Time Complexity of topolical sorting is $O(V+E)$, where $V$ = Vertices, $E$ = Edges.

To find out the time complexity of the algorithm, let's try to find the time complexity of each steps.

→ To determine the indegree of each vertex, we have to iterate through all the edges of the graph. So time complexity is $O(E)$

→ Next, to look for vertices with zero indegree. This will require to iterate through the entire array that stores the indegree of each node. The size of the array is equal to $V$. So the time complexity of this step is ~~$O(V)$~~ $O(V)$

→ Next, for each ~~node~~ vertex with an indegree of zero, we ~~remove~~ it ~~from the qu~~ we decrement the indegree of those vertices adjacent to it by 1. In the entire run of the algorithm the no. of times, we have to perform decrement operation is equal to the no. of edges. So it will take $O(E)$ time.

→ Whenever we decrement the indegree of a node, we check the Indegree of the node becomes zero or not. If zero we add it to the queue. In the entire run of the algorithm, it can occur at max $(V-1)$ times. So the run time of this opern is $O(V)$

→ So adding up all the individual run times, the time complexity of topological sort is $O(V+E)$

→ This is the best case, avg case & ~~best~~ worst case time complexity of the algo since we perform the same steps, regardless of how the elements are organized.

## ~~Topo~~ Space complexity of Topological sorting

Space Complexity is $O(V)$

↳ We have to create an array to store the indegree of all vertices. This will require $O(V)$ space

→ We have to store the vertices with indegree 0 in a queue (or stack). In the worst case, it will store all vertices, requiring O(V) space.
→ Finally we need to store all the vertices in sorted order in an array requiring O(V) space.
→ Adding all three we arrive at the space Complexity is O(V)

## Applications of Topological Sorting

⇒ Scheduling jobs from the given dependencies among jobs
⇒ Determing the order of compilation tasks in a compiler
⇒ Instruction scheduling
⇒ Data Serialization.

## Graphs

A graph is a data structure that consists of following 2 components.
1) A finite set of vertices (nodes)
2) A finite set of ordered pair of the form (u,v) called as edge.

### Representation of a graph.

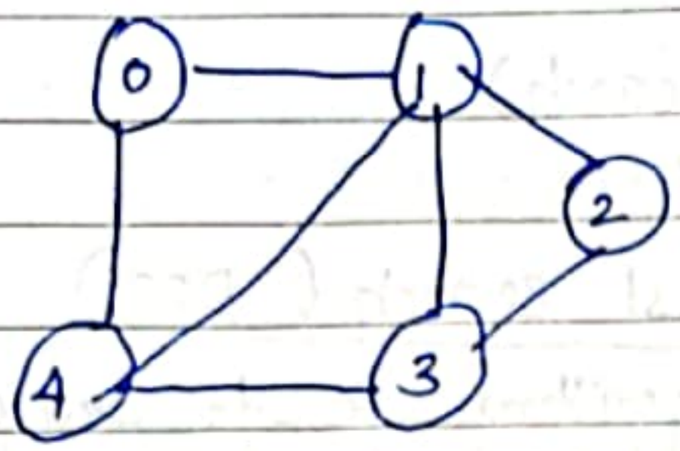
1) Adjacency Matrix
2) Adjacency list.

### Adjacency Matrix

→ Adjacency matrix is a 2D array of size VxV, where V is the no. of vertices in a graph.
→ Let the 2D array be adj[][], a slot adj[i][j]=1 indicates that there is an edge from vertex i to j.
→ Adjacency matrix for undirected graph is always symmetric
→ Adjacency matrix is also used to represent weighted graphs
→ If adj[i][j] = w, then there is an edge from vertex i to

vertex j with weight w.

eg:-




Adjacency matrix

| adj = | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 0 | 1 | 0 |

Space complexity - $O(V^2)$

Time complexity - Computing all neighbours of a vertex takes $O(V)$ time $\therefore$ Time complexity = $O(V^2)$

Adjacency list Representation

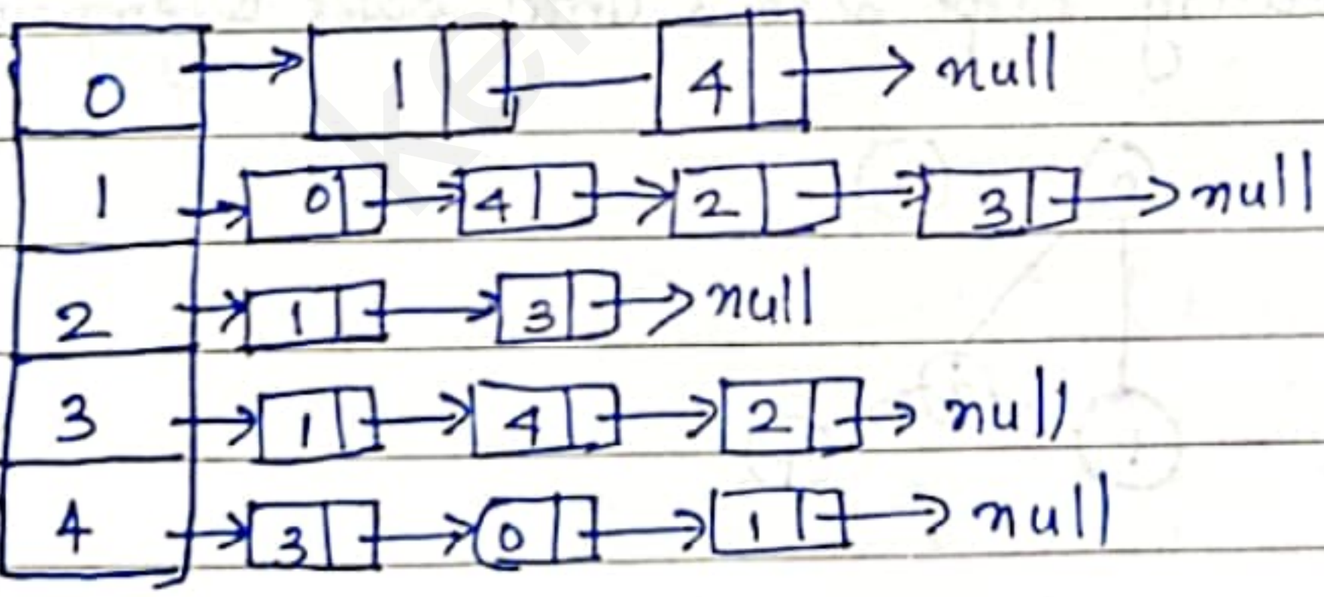
→ An array of lists is used.

→ The size of the array is equal to the no. of vertices.

→ Let the array be A[]. An entry A[i] represents the list of vertices adjacent to ith vertex.

→ This representation can also be used to represent a weighted graph.

→ The weights of edges can be represented as lists of pairs.




If n is the no. of vertices & m is the no. of edges in a graph, the time complexity of building such a list is O(m).

Space Complexity is O(n+m).

## Graph traversals

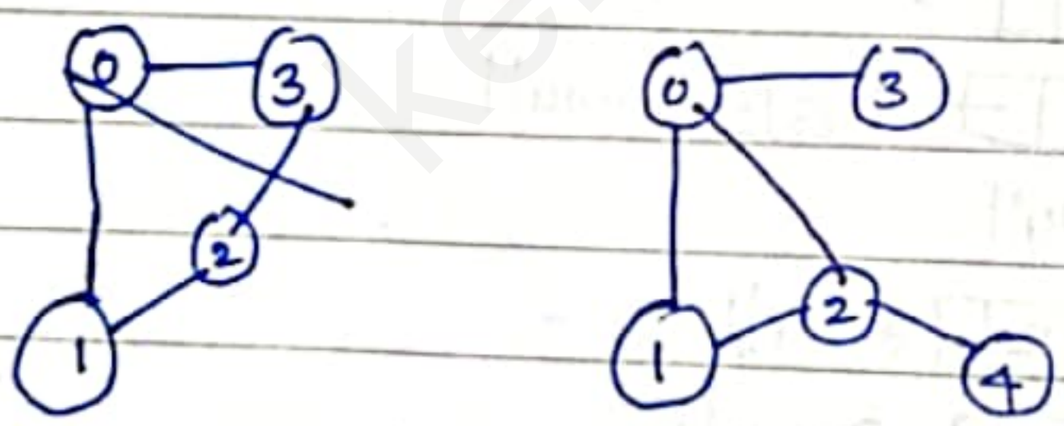
DFS ( Depth first Search)
BFS ( Breadth first search)

## ~~Breadth f~~ Depth first search ( DFS)

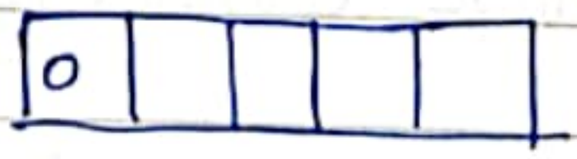
→ It is a recursive algorithm ~~se~~ to search all the vertices of a tree or a graph.
→ A DFS implementation puts each vertex of the graph into one of the two categories.
1. Visited
2. Not Visited.

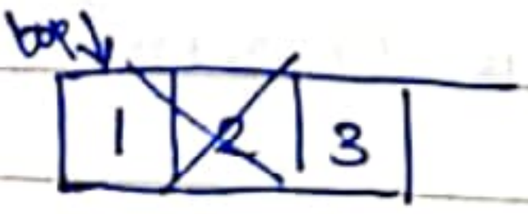
The DFS algorithm works as follows.
1. Starts by ~~putting~~ taking any one of the graph's vertices on ~~top of~~ a ~~stack~~. and make it visited. Add its neighbors to stack
2. Take the top item of the stack & add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 & 3 until stack is empty.

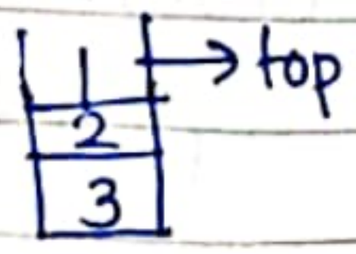
eg:-



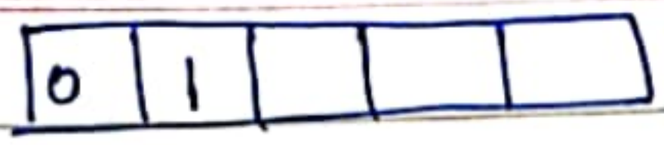

We start from vertex 0. The algorithm starts by putting it in the visited list & putting all its adjacent vertices in the stack.

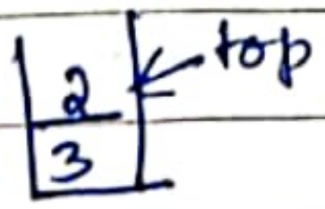
Visited 


stack 

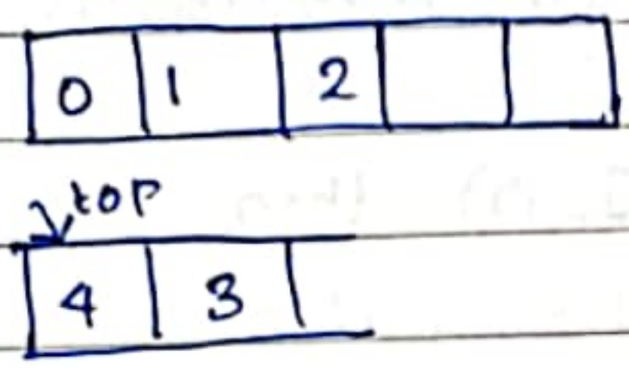

stack 

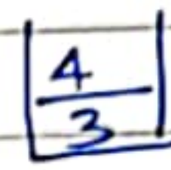

Next we visit the element at the top of stack ie 1 & go to its adjacent nodes. Since 0 has already visited, we visit



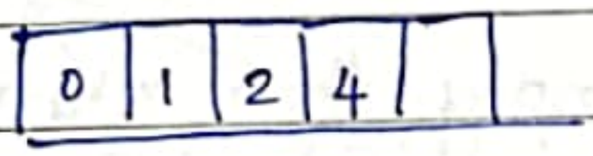




Vertex 2 has an unvisited vertex (adjacent) 4, so we add that to the top of the stack. & visit it.




Ver Next we visit the element at the top of stack ie 4 & go to its adjacent nodes. No adjacent nodes for 4





Next we visit 3, it doesnot have any unvisited adjacent nodes, so we have completed Depth first traversal of the graph. stack → empty

Algorithm

Given an undirected graph (directed) $G = (V, E)$ with n vertices and an array visited [] initially set to 0
For any node i, visited [i] = 1 if i has already been visited. The graph G & array visited [] are global.
Let S be a stack of unvisited vertices. initially empty.

```
Algorithm BFT (G, n)
{ for i=1 to n do
        visited [i]=0
    for i=1 to n do
        if (visited [i]= 0) then DFS(i);
}
```

```
Algorithm DFS(v)
{   u=v;
    visited [v]=1;
    repeat
    {
        for all verticeswadjacent from u do
        {
            if (visited [w]=0) then
            {
                push w to S;
                visited [w]=1;
            }
        }
        If S is empty then return;
        u=Pop S;        visited [u]=1;
    } until(false)
}
```

OR

## Complexity Analysis of Depth first Search.

### Time Complexity

→ Let V be the no. of vertices & E be the no. of edges in the graph.

→ If the graph is represented as adjacency list:

* Here each node maintains a list of all its adjacent edges.
* For each node, we discover all its neighbours by traversing its adjacency list just once in linear time.
* For a directed graph, the sum of sizes of the adjacency lists of all the nodes is E. So the time complexity in this case case is $O(V) + O(E) = O(V+E)$.
* For an undirected graph, each edge appears twice. The time complexity for this case is $O(V) + O(2E)$

Visited [i] has to be initialized to 0, $1 \le i \le n$. This takes O(V) time

$\sim O(V+E)$

→ If the graph is represented as an adjacency matrix (a $V \times V$ array).

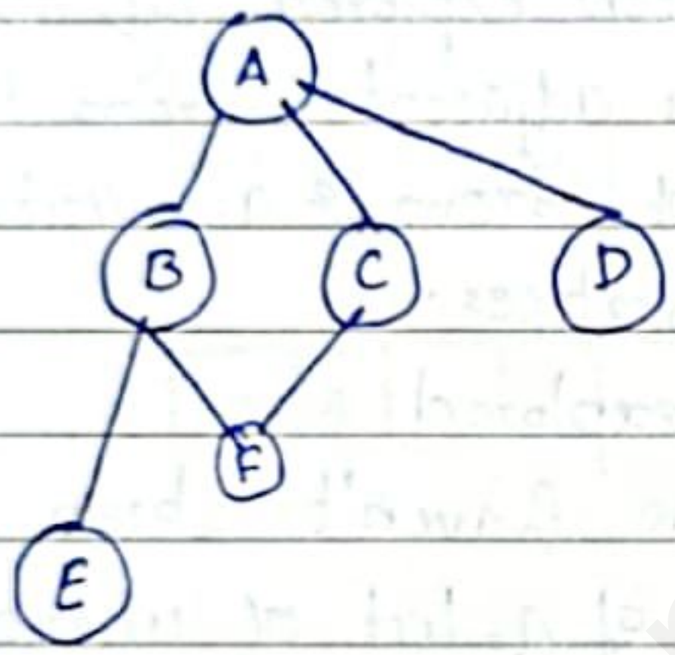
* For each node, we will have to traverse an entire row of length V in the matrix to discover all its outgoing edges.

∴ Time complexity is $O(V \times V) = O(V^2)$.

Space Complexity

→ Since we are maintaining a stack to keep track of the last visited node, in worst case, the stack could take up to the size of the nodes in the graph. Hence the space complexity is $O(V)$.

eg:-




Depth First Traversal
= A B E F C D

BFS = A B C D E F.

→ Recursive version of Algorithm DFS(V).

```
Algorithm DFS(v)
{
    visited [v] = 1
    for each vertex w adjacent from v do
    {
        if (visited [w] = 0) then DFS(w)
    }
}
```

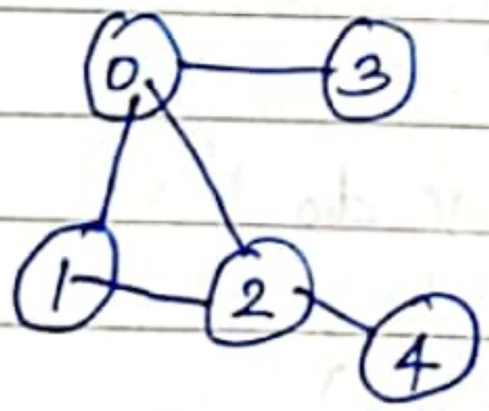
## Applications of DFS

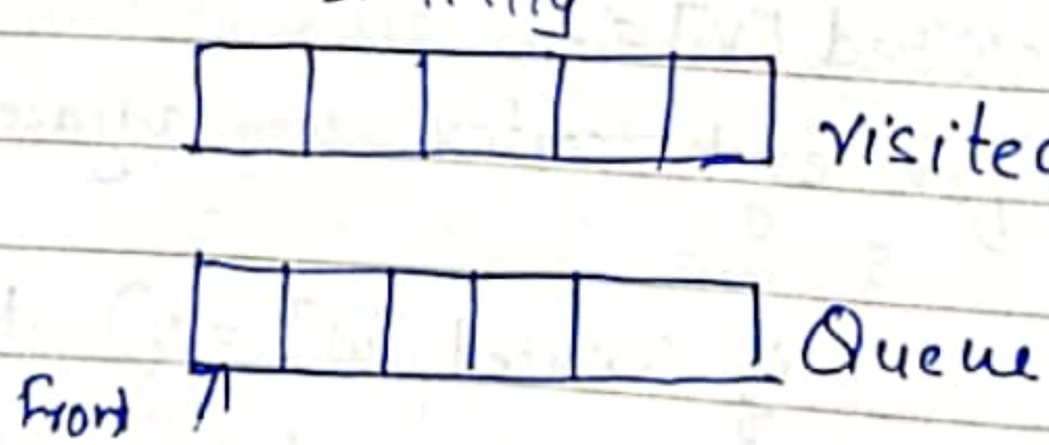
1) Detecting cycle in a graph
2) Path finding
3) Topological sorting
4) To test if a graph is bipartite.
5) Finding Strongly Connected Components of a graph.
6) Solving puzzles with only one solution.

## Breadth first Search (BFS)

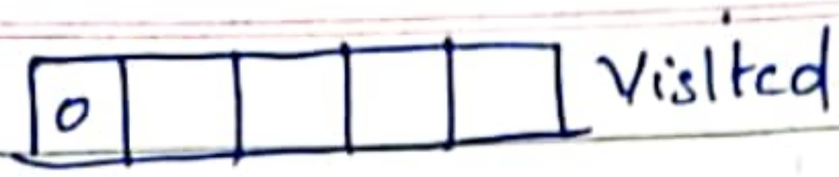
→ In BFS we start at vertex v and mark it as visited.
→ The vertex v is at this time said to be unexplored.
→ A vertex is said to have been explored by an alg, when the alg has visited all vertices adjacent from it.
→ All unvisited vertices adjacent from v are visited next.
→ These are new unexplored vertices.
→ Vertex v has now been explored
→ The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices.
→ The first vertex on this list is the next to be explored.
→ Exploration continues until no unexplored vertex is left.
→ List of unexplored vertices operates as a queue.

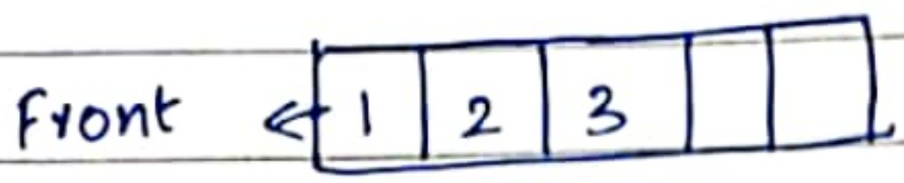
## BFS example



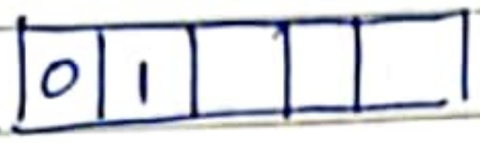



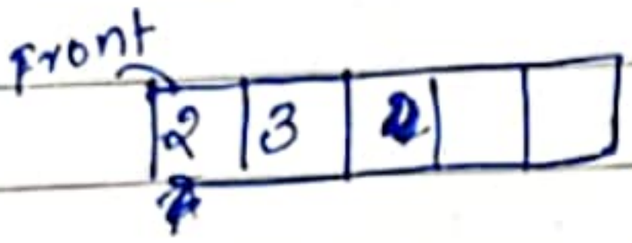

We start from vertex 0, the BFS alg. starts by putting it in the visited list and adding its adjacent vertices to queue.



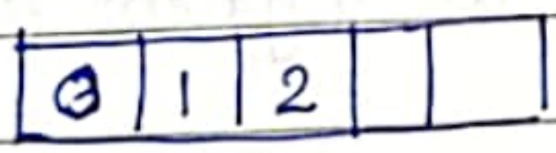



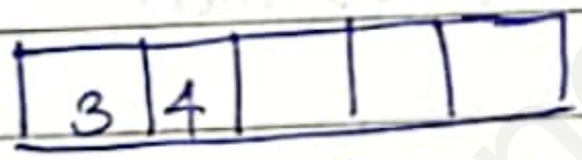

Visit the first neighbour of 0, ie 1 and add its adjacent neighbors to Queue. Neighbors of 1, 0 & 2, 0visited.



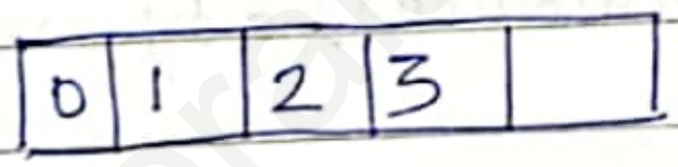



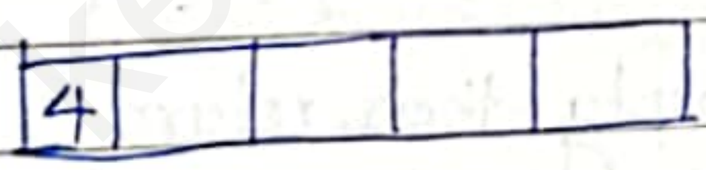

Visit Vertex 2 and put its neighbor 4 on to queue.



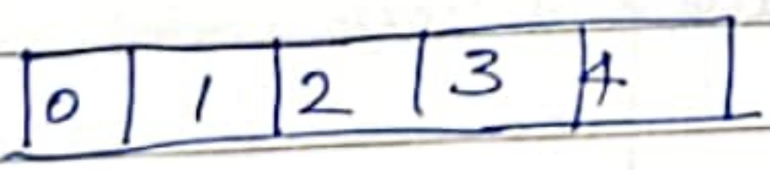



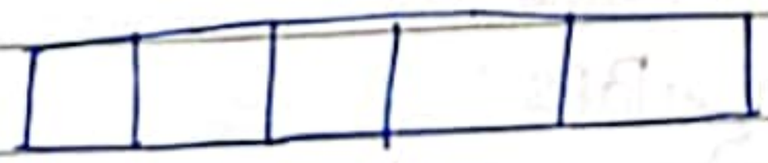

Visit 3 & 3, has no unvisited neighbors.




Visit 4




Algorithm

A breadth first search of G is carried out beginning at vertex v. For any unvisited node i, visited[i]=1, if i has already been visited. The graph G and array visited[] are global; visited[] is initialized to zero. q is a queue of unexplored vertices.

```
Algorithm BFT (G, n)
{ for i=1 to n do
      visited [i] = 0
  for i = 1 to n do
      if ( visited [i] = 0) then BFS(i);
}

Algorithm BFS(v)
{
  u = v;
  visited [v] = 1;
  repeat {
      for all vertices w adjacent from u do
          {
              if (visited [w] = 0) then
                {
                  Add w to q;
                  visited [w] = 1;
                }
          }
      if q is empty then return;
      Delete the next element, u from q;
  } until(false)
}.
```

Complexity Analysis of BFS

Same as DFS. → study DFS analysis.

Adjacency list

Time Complexity = $\Theta(V+E)$

Adjacency matrix, Time Complexity = $O(V^2)$

Space Complexity :- Using a queue, Worst case ⇒ size of the queue might be V .! Space complexity = $O(V)$

Applications of BFS

1) Shortest Path &
   Finding the Shortest Path & Minimum Spanning Tree.
   for unweighted graph.
2) Peer to Peer Networks: In Peer to Peer n/ws like BitTorrent, BFS is used to find all neighbors.
3) Crawlers in search Engines: Crawlers build index using BFS
4) Social Networking Websites: In social n/ws, we can find people within a given distance 'k' from a person using BFS till 'k' levels.
5) GPS Navigation s/m: BFS is used to find all neighboring locations.
6) Broadcasting in n/ws.
   e/e: - - .

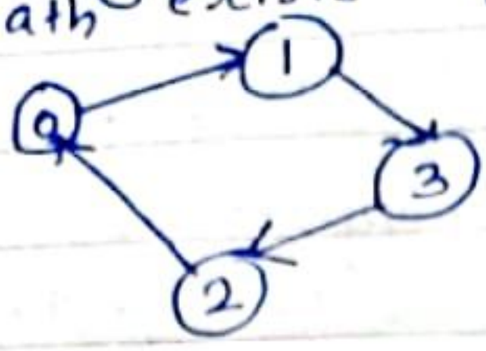
Comparison

| BFS | DFS |
|---|---|
| 1. BFS traverses according to tree level | 1. DFS traverses according to tree depth. |
| 2. BFS uses Queue to find shortest Path | 2. DFS uses stack to find S.P. |
| 3. Here siblings are visited before the children | 3. Here children are visited before siblings |
| 4. BFS is better when target is closer to source. | 4. DFS is better when target is far from source. |
| 5. As BFS considers all neighbours, So it is not suitable for decision trees used in puzzle games | 5. DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation we stop. |
| - there is no backtracking | 6. DFS uses backtracking. |
| | 7. DFS is faster. |

## Strongly Connected Components of a Directed Graph.

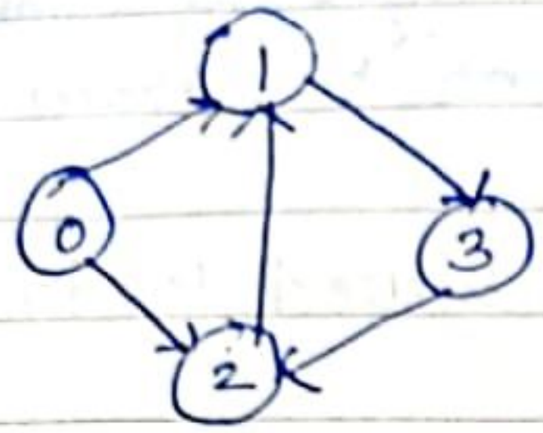
A directed graph is said to be a strongly connected graph, if every vertex is reachable from every other vertex. A path exists bln every all pair of vertices (SCG)

eg:-




→ Strongly connected graph.

eg:-

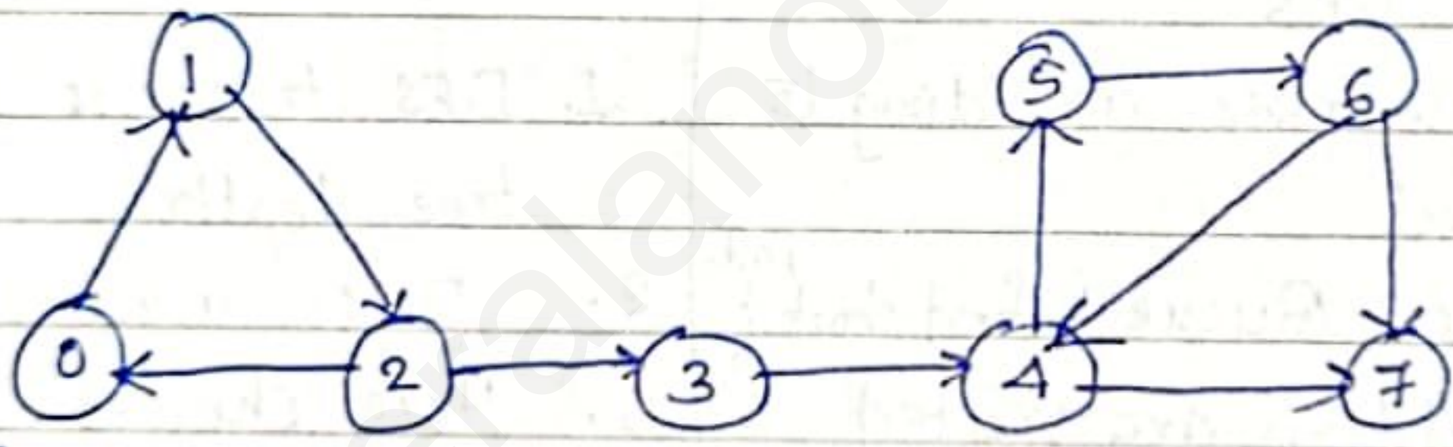



→ Not strongly connected graph.

→ A strongly connected component is the portion of a directed graph in which there is a path from each vertex to another vertex. (SCC)

→ ~~It is applicable only on a directed graph.~~

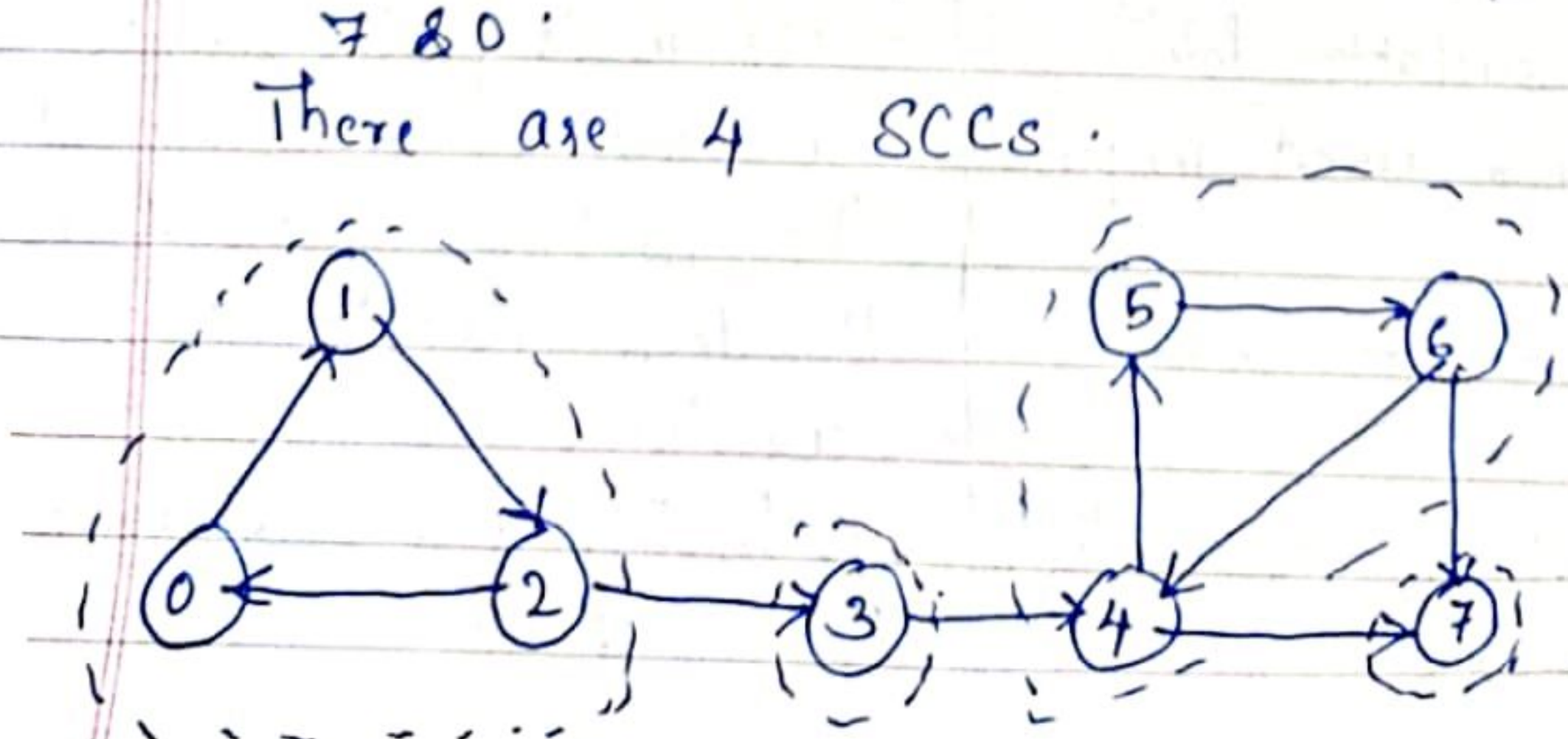
Q.




Is this graph an SCG? What are the SCCs?

Soln This graph is not an SCG because there is no path exists bln 3 & 2, 3 & 0, 3 & 1 also bln 7 & 4, 7 & 5, 7 & 6, 7 & 3, 7 & 2, 7 & 1 & 7 & 0.

There are 4 SCCs.




SCC1 – 0, 1, 2

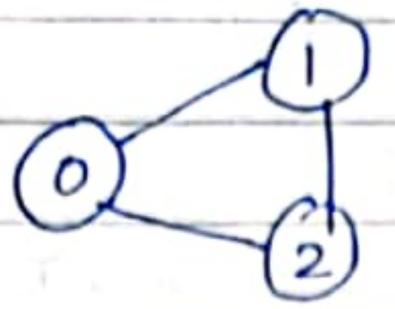
SCC2 → 3

SCC3 → 4, 5, 6

SCC4 → 7.

Note:- All the nodes in a component are always strongly connected in an Undirected graph.

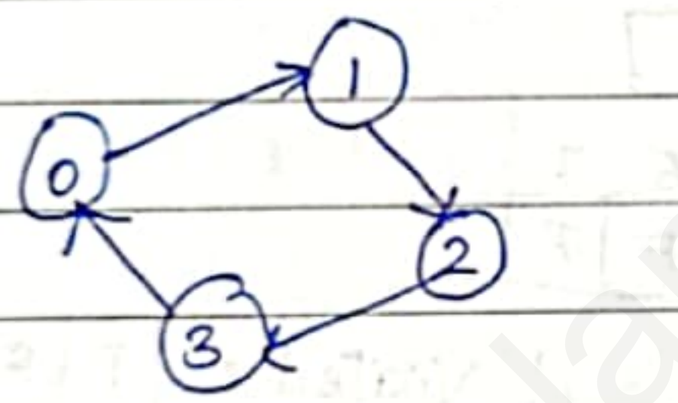
eg:-




It is an SCG also.

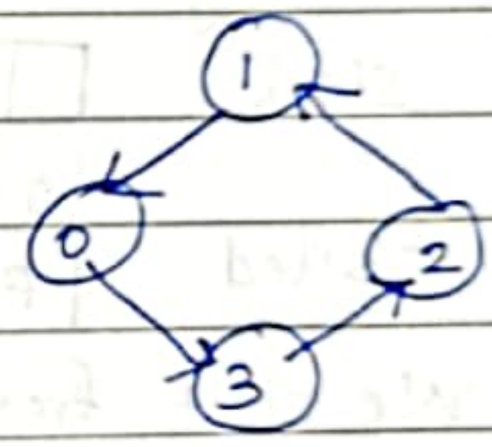
Note:- On reversing all the edges of the graph, the type of the graph won't change.
ie If it is an SCG, then reversing edges also makes the graph turns to be SCG.
If the graph is not an SCG, then reversed graph also won't be an SCG.
The SCCs in the graph & the SCCs in the reversed graph also remains same.

eg:-



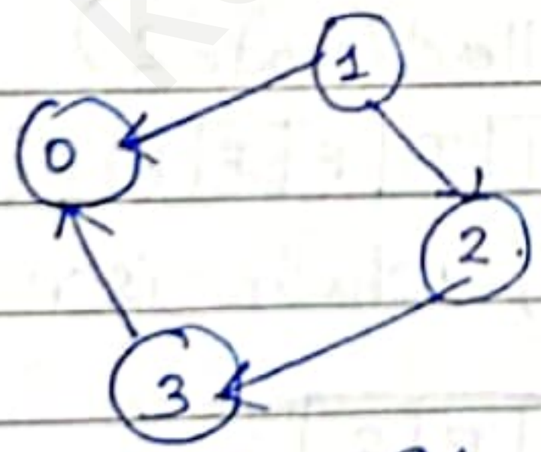

It is an SCG

Reverse the edges



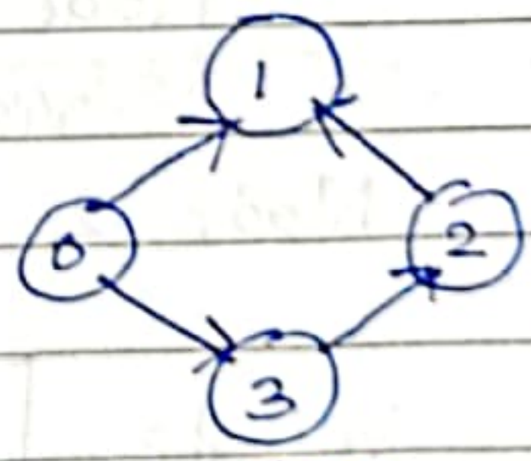

SCG.

eg:- 2:-



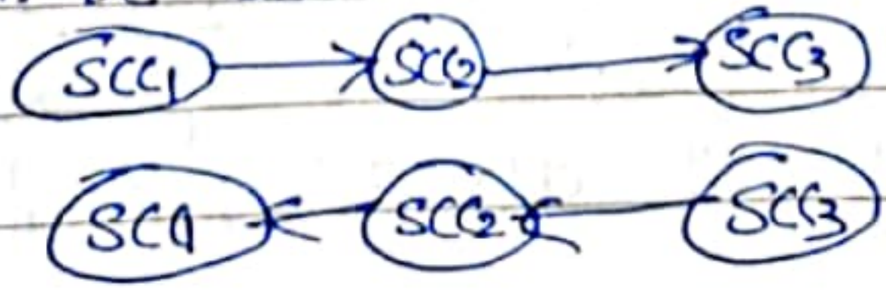

Not an SCG

Reverse the edges




Not an SCG.

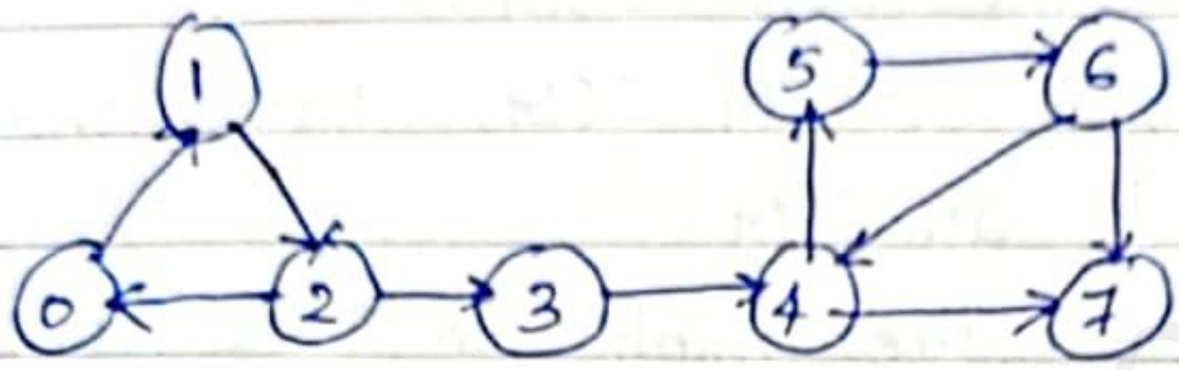
ie If we take the transpose of a graph, SCC will remain as SCC. Direction b/n components are reversed.
This property can be used to detect SCCs.



## Kosaraju's Algorithm to find SCCs

(1) Perform DFS traversal of graph. Push nodes to stack before returning.
(2) Find the transpose graph by reversing the edges.
(3) Pop nodes one by one from stack and again do DFS on the modified graph. Each successful DFS gives one strongly connected component.

Example.




Initially stack is empty. Visited list is set to false.

Stack 

Visited

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

We start from node 0 ∴ Visited = | T | F | F | F | F | F | F | F |

Node 0 will call node 1.

∴ Visited = | T | T | F | F | F | F | F | F |

Node 1 will called dfs(node 2)

Visited = | T | T | T | F | F | F | F | F |

Node 2 will call 0, which is visited, then call n

Visited =

| T | T | T | T | F | F | F | F |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Node 3 will call dfs(4)

Visited =

| T | T | T | T | T | F | F | F |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

like this 4 will call dfs(5). 5 will call dfs(6), 6 will call dfs(7), 4 already visited.

∴ Visited =

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

From 7 there are no outgoing edges : So the alg return back from 7 to 6. When we are returning back 7 is pushed on to stack.

Stack 7

stack 7

From 6, the outgoing edges are 7 & 4 which are already visited. So push 6 and return back to 5

Stack= 7 6

push 5 &

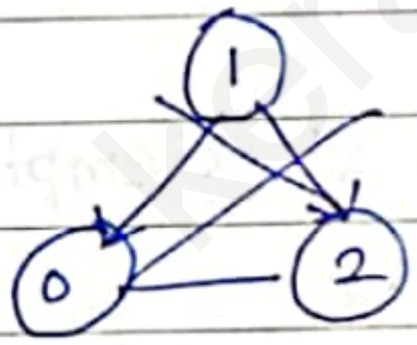
From 5, 6 is already visited, so go back to 4. From Push 5

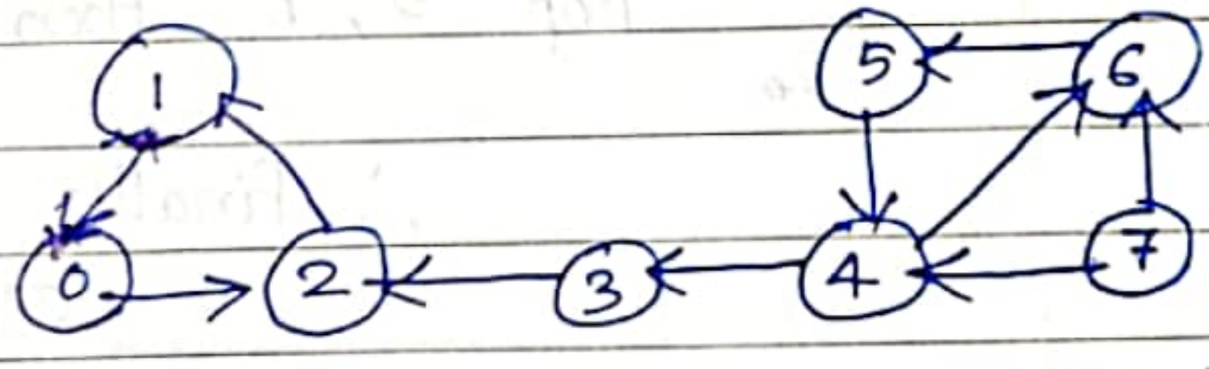
∴ Proceeding like this at the end of the backtracking

Stack = 7 6 5 4 3 2 1 0

We have completed step 1.

Next step is to find the transposed graph.




Next we proceed to 3rd step. Keep visited array to False

Pop 0 & perform DFS.

Stack = 7 6 5 4 3 2 1

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Visited = F | F | F | F | F | F | F | F |
| | T | T | T | | | | |

I St SCC → 0,2,1

Pop 1 from stack. Since 1 is already visited we won't call dfs(1). Pop 2, 2 is also visited So dfs(2) is not invoked.

Stack = | 7 | 6 | 5 | 4 | 3 |

Pop 3 and call dfs (3)

Stack = | 7 | 6 | ~~5~~ | 4 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Visited = | ~~F~~ T | ~~F~~ T | ~~F~~ T | ~~F~~ T | F | F | F | F |

& From 3, there is only one outgoing edge 2, but 2 is already traversed. Therefore dfs(2) is not invoked.

∴ So $SCC_2 = 3$.

Pop 4 & repeat the same process.

Stack = | 7 | 6 | 5 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Visited = | ~~F~~ T | ~~F~~ T | ~~F~~ T | ~~F~~ T | ~~F~~ T | ~~F~~ T | ~~F~~ T | F |

∴ So $SCC_3 = 4, 6, 5$.

Pop 5, 6 then 7.

Do

∴ Finally we get stack empty.

$SCC_4 = 7$.

∴ Four strongly connected components.

$SCC_1 = 0, 1, 2.$

$SCC_2 = 3$

$SCC_3 = 4, 5, 6$

$SCC_4 = 7.$

Time Complexity: The above alg calls DFS, finds reverse of the graph & again call DFS.

DFS take $O(V+E)$ for a graph represented using

adjacency list.
- Reversing a graph also takes $O(V+E)$ time.

∴ Time complexity = $O(V+E)$

Applications of SCCs.
→ Used to solve 2-satisfiability problems.
→ Used in social n/wing sites, to depict the group of people who are friends of each other or who have any common interest.
→ Used to convert a graph into a Direct Acyclic graph of strongly connected components.