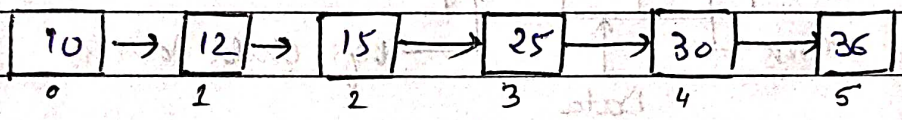
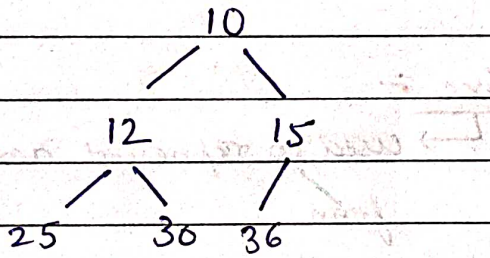
Trees

Binary Tree from Linked list :-

→ if root node is 'i' then the left and right nodes are $2*i+1$ and $2*i+2$





→ Head is always root node of linked list

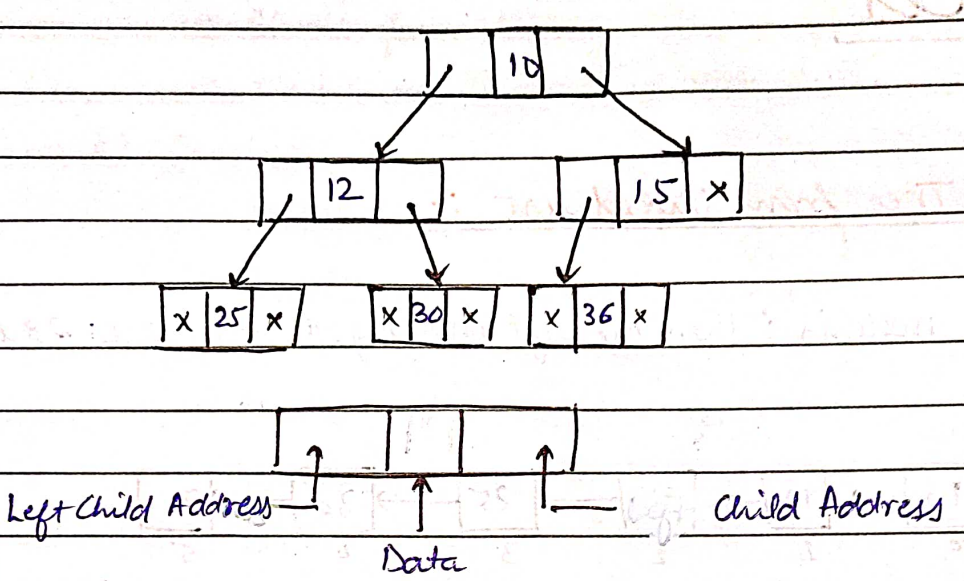
Algorithm :-

1. Create an empty queue
2. Make the first node of the list as root, and enqueue it to the queue
3. Until we reach end of the list follow →
   a. Dequeue one node from the queue. This is the current parent
   b. Traverse two nodes in the list, add them as children of the current parent.
   c. Enqueue the two nodes ~~from~~ to the queue

Time Complexity :- $O(n)$ → no. of nodes

Space Complexity :- $O(b)$ → maximum number of nodes at any level.

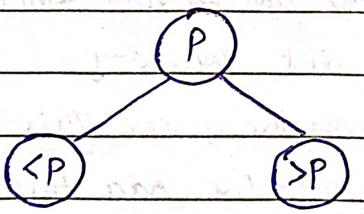
Representation :-



Binary Search Tree :-

↳ used to represent data in hierarchical form

→ Binary search tree is a tree in which left node should be smaller than parent node and right node should be greater than parent node.

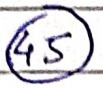


Advantages of Binary Search Tree :-

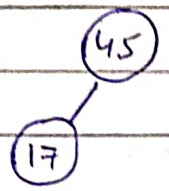
→ Searching element in binary search tree is easy.
→ Insertion and Deletion is faster than array and linked list.

Creating BST :-

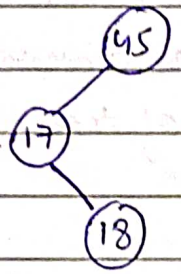
45, 17, 18, 20, 40, 4

→ 

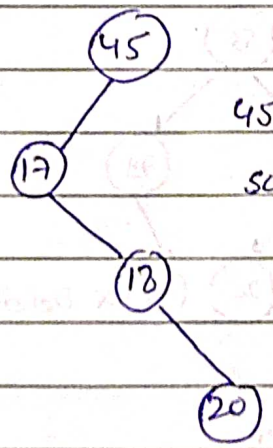
Insert 45.

→ 

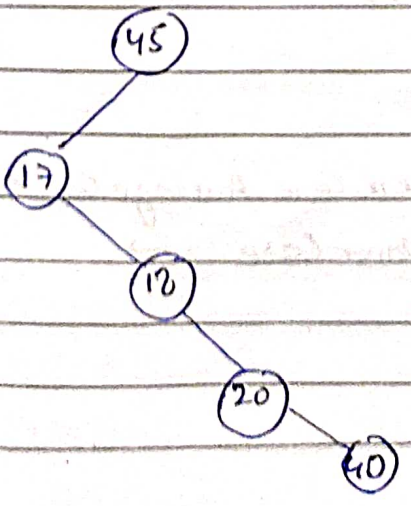
17 < 45 left side

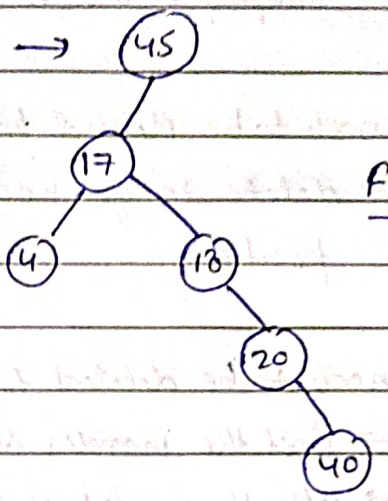
→ 

45 > 18 > 17
left of 45 right of 17

→ 

45 > 20 > 18 > 17
so left of 45 and right of 18

→ 

→ 

Final Tree

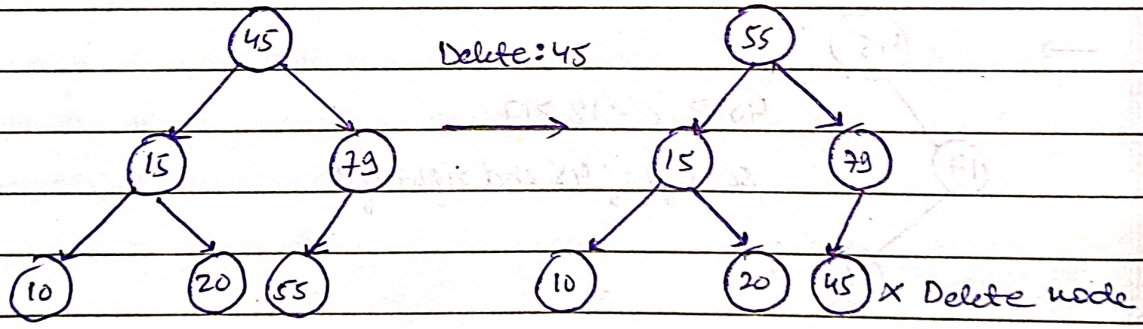
Deletion in binary search tree :-

1. If its the node i.e. to be deleted leaf node :-
   → Simply delete it

2. If node to be deleted has one child :-
   → delete target node move child node to target nodes position

3. If node to be deleted has two children :-
   → Find the inorder successor of the node to be deleted
   → After that replace the node with inorder successor until the target node is placed at the leaf of tree.

Representation :-



Inorder Traversal → 10, 15, 20, 45, 55, 79

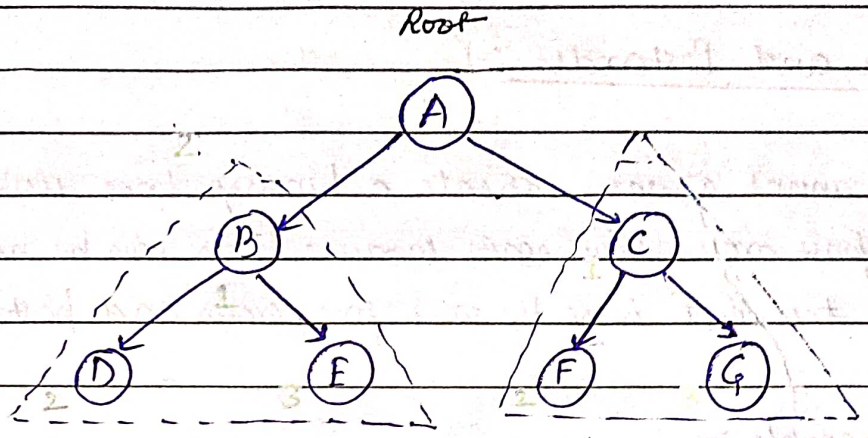
Inorder Successor of 45 is 55.

Time Complexity :-

→ Insertion, Deletion, Searching : Best Case, Average Case → O(log n)
: Worst Case → O(n)

→ Space Complexity :- O(n)

Tree Traversal Algorithms :-



Post Order Traversal :-

↳ Root node at last

Left, Right, Root

D → E → B → F → G → C → A

In Order Traversal :-

↳ Root node in middle

Left, Root, Right

D → B → E → A → F → C → G

Pre Order Traversal :-

↳ Root node at start

Root, left, Right

A → B → D → E → C → F → G

Construction of Binary Tree :-

→ **PreOrder and Postorder :-**

* We cannot always create a binary tree using pre-order and post order traversals because there can be multiple binary tree that have the same pre-order and post order traversals.

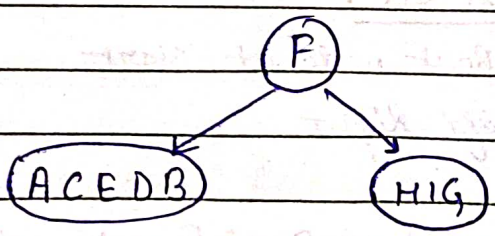
Example :-

Pre Order :- F, B, A, D, C, E, G, I, H (NLR)
Post Order :- A, C, E, D, B, H, I, G, F (RLN)

Construction :-

1. First the first element of pre order and last element of post order is root. so insert Root

2. Find Successor of F in pre order i.e. B so split Post Order into two halfs i.e left subtree contains A, C, E, D, B and Right subtree contains H, I, G

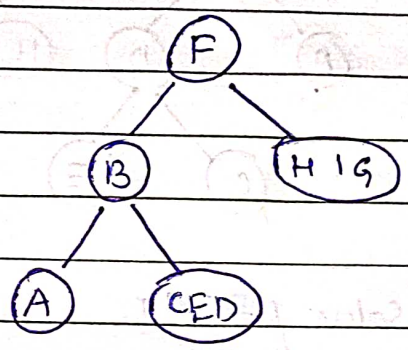


Post Order → ACEDB
Pre Order → BADCE → All the elements of post-order in pre-order sequence from main sequen

3. Now similarly as step-2 divide →

Pre-Order → BADCE

Post-Order → ACEDB

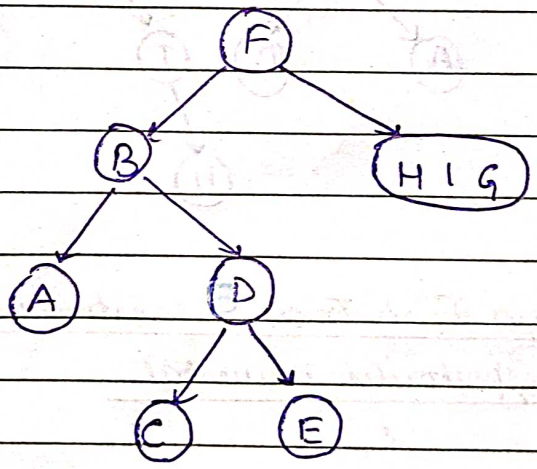


Pre-Order → ADCE

Post-Order → CED

4. Find for Pre-Order → DCE

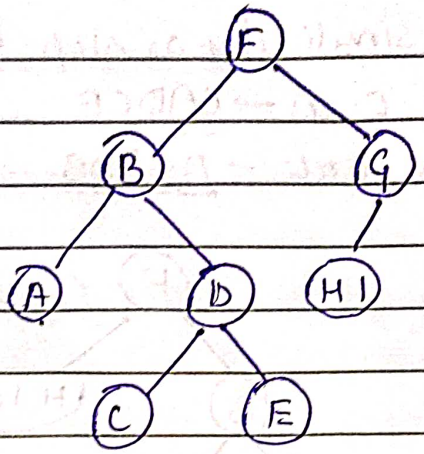
Post-Order → CED



5. Now solve for HIG

Pre-Order → GIH

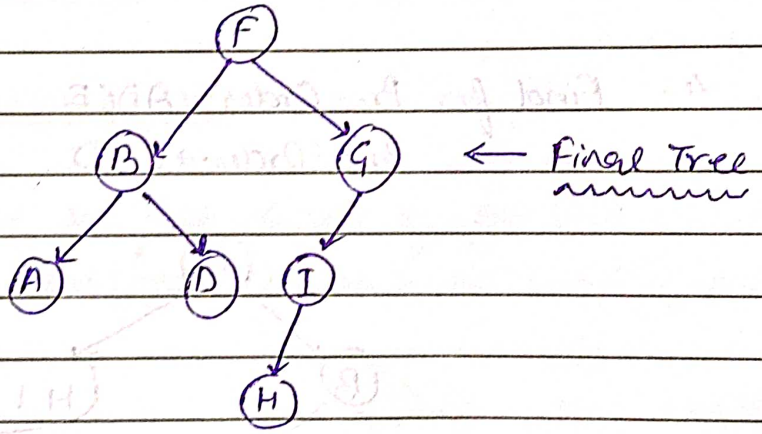
Post-Order → HIG



6. Solve for H I

Pre-Order — I H

POst-Order — H I



← Final Tree

Easy and Fast Trick to construct a binary tree using preorder, postorder traversal.

Pre-Order → F B A D C E G I H (NLR)

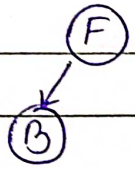
Post-Order → A C E D B H I G F (LRN)

P.T.O

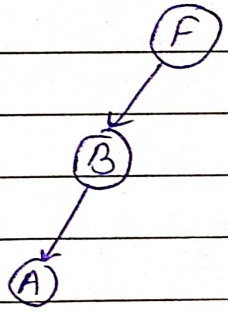
1. F is the root node

* traverse after each step in post-order format.

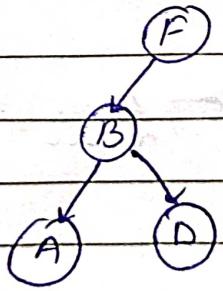
2. (B) in pre order is the next successor of (F) and in post order (B) is left of (F)



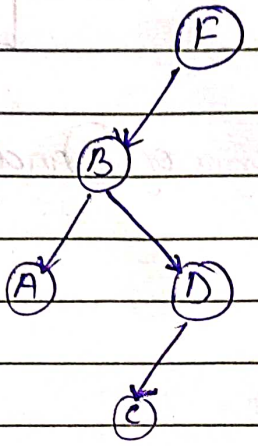
3. Now, (B) in pre order next successor is (A) and (A) is left to the (B) in postorder.



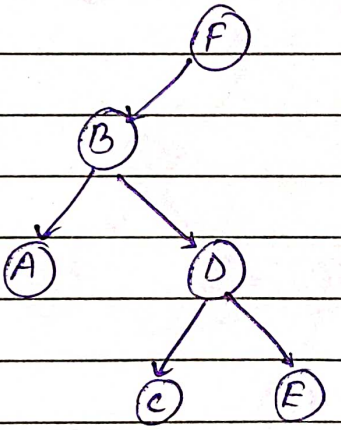
4. Now, (A) in pre-order (D) is successor and in post order (d) is right of (A) but ~~since~~ insert (D) to the right of (B) pre-order traversa



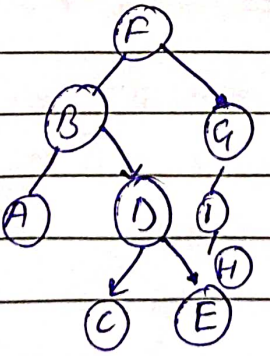
5. Now (C) in preorder the (C) is left of (D) in post order



6. Now (E) and (E) is to the right of (C) in post order



7. Now insert (G) is to the right of (B) traversed LRN

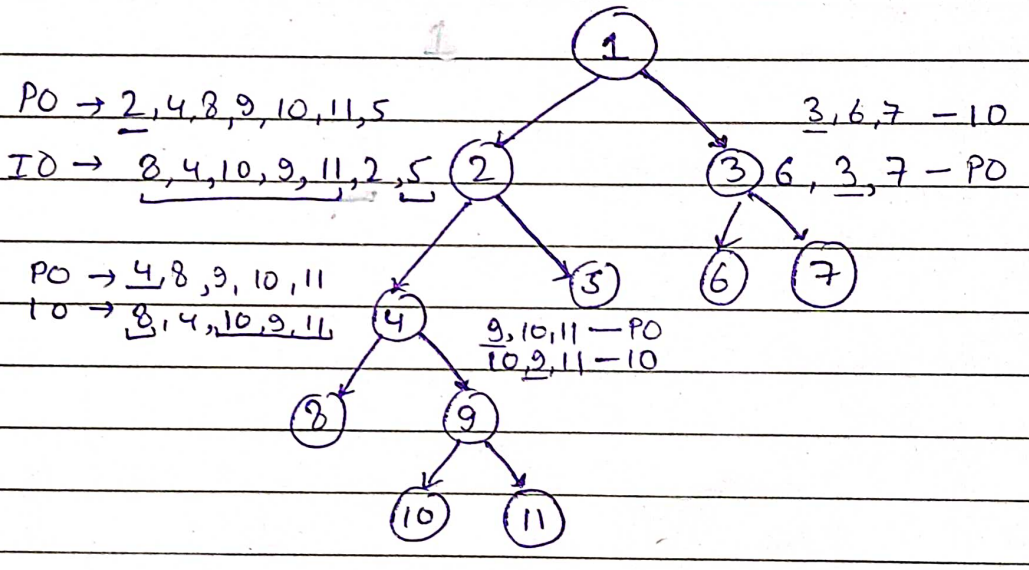


← Final

→ Pre Order and In Order :-

Pre-Order :- 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 (N, L, R)
In-Order :- 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 (L, Root, R)

1. The first element from Pre-Order will be root and ~~the~~ in the In order the element selected in preorder the left element from the selected pre order element will be left sub tree and to the right will be right sub tree



→ Post Order and In Order :-

1. Same as Pre-Order and In Order but traverse from Post-Order from Right to left.

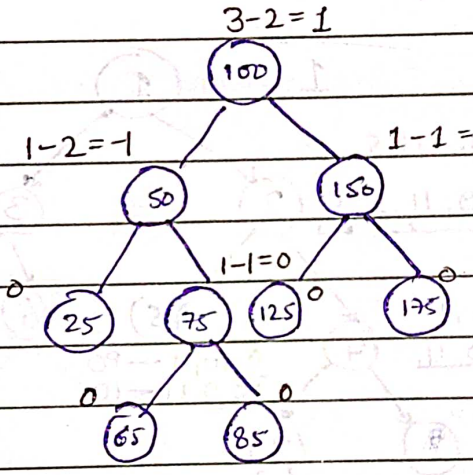
Post-Order :- 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 ←
In-Order :- 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

AVL tree :-

→ height balanced binary search tree

→ Tree is said to be balanced if balance factor of each node is between -1 to 1, other wise it is unbalanced and need to be balanced.

Formula → balance factor (k) = height (left(k)) - height (right(k))



Complexity :-

Time Complexity in Search, Insert, Delete Operation :-

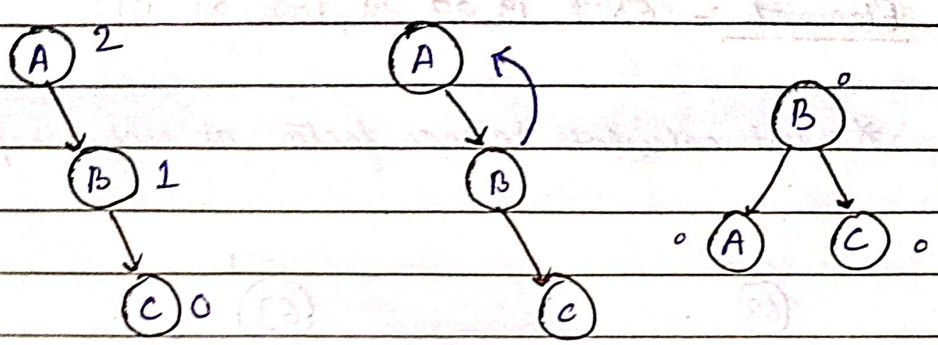
Average Case :- O(log n)

Worst Case :- O(log n)

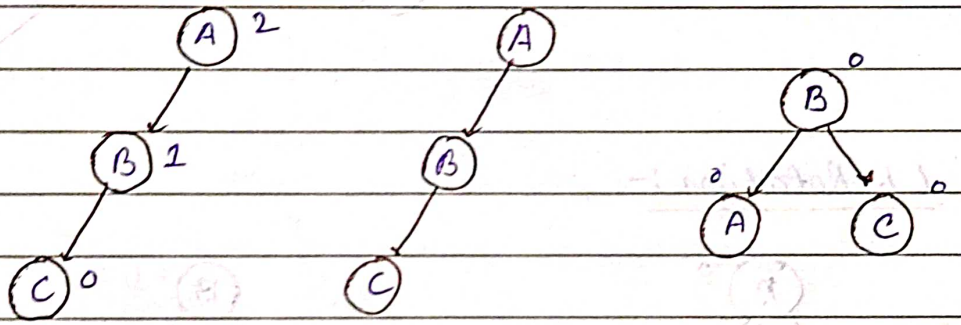
Space :- Average and Worst :- O(n)

Rotations in AVL tree :-

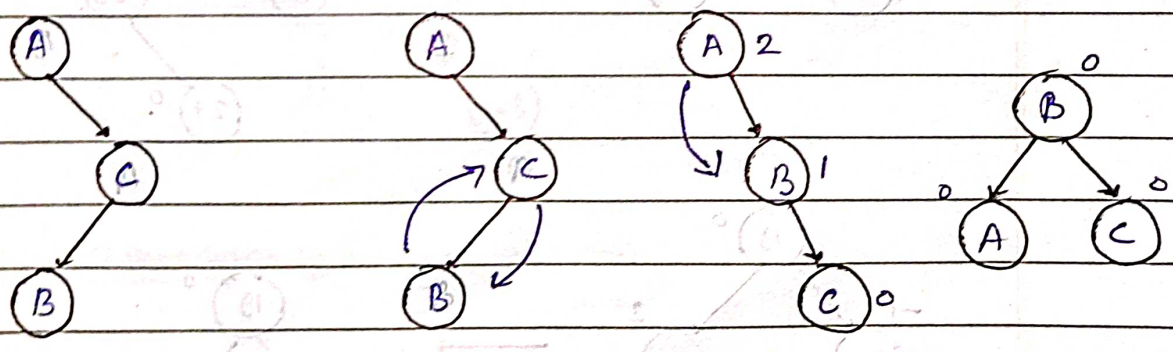
RR →



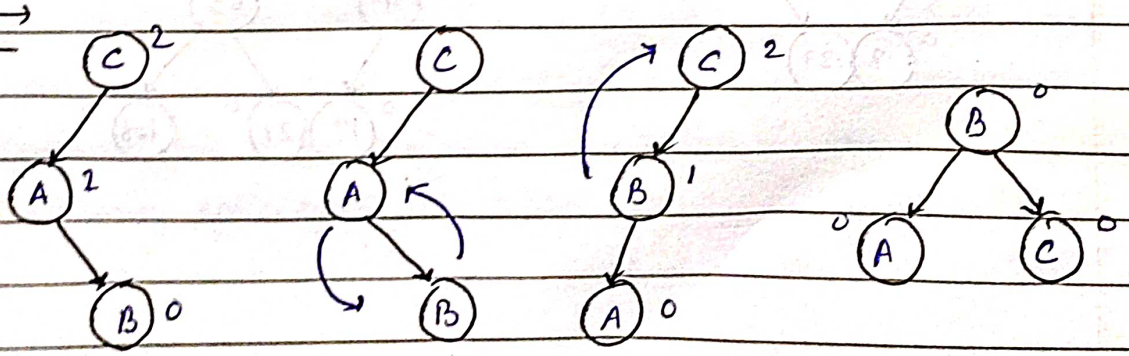
LL →



RL →



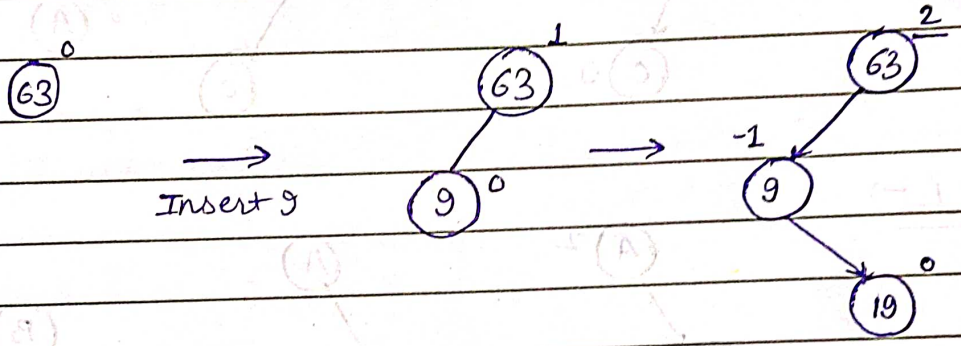
LR →



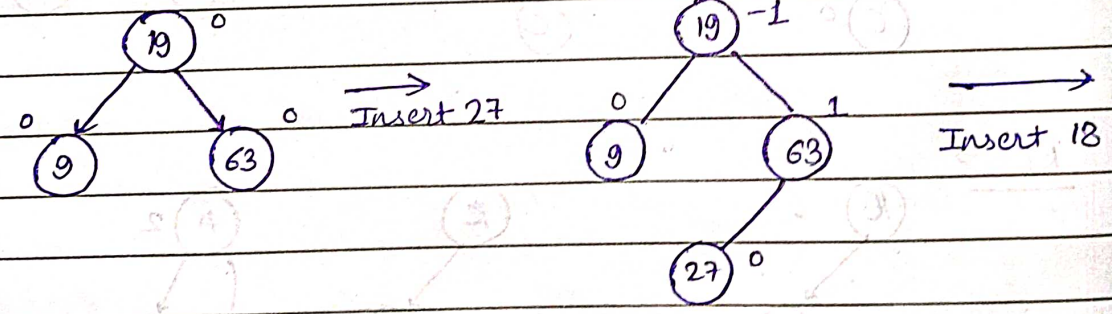
Insertion in AVL tree :-

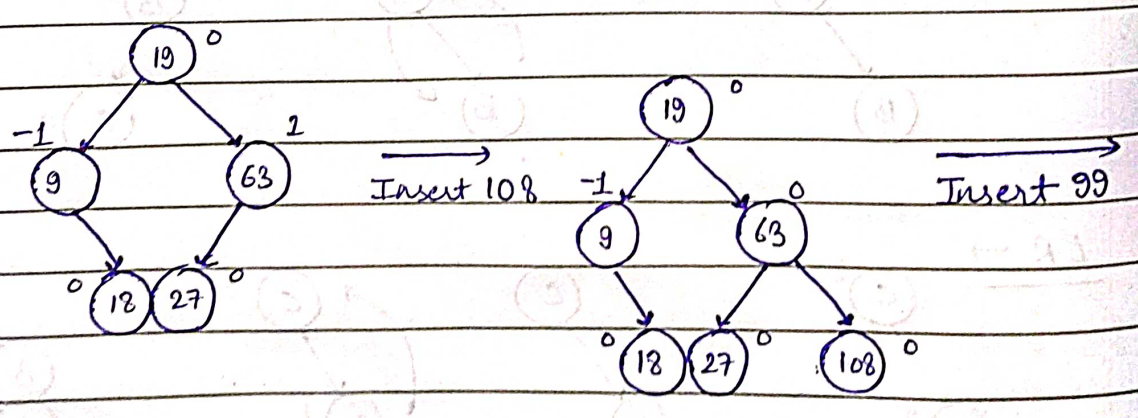
Elements :- 63, 9, 19, 27, 18, 108, 99, 81

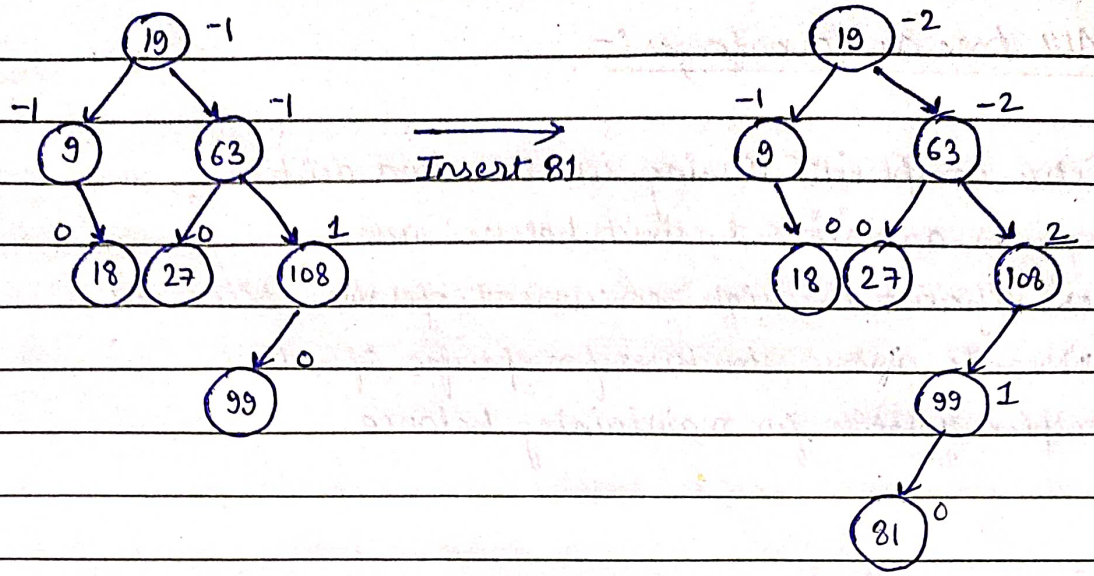
* must calculate balance factor at each step :-



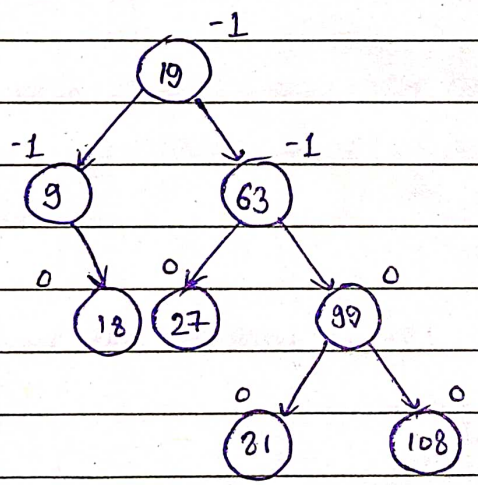
LR Rotation :-







LL Rotation :-



Final Tree

AVL Tree Advantages :-

1. Efficient search operation with fixed time complexity.
2. Quick insertion and retrieval with O(logn) time complexity.
3. Consistent worst case time complexity for various operations.
4. Relatively simple implementation compared to other trees.
5. More memory efficient.

AVL tree disadvantages :-

1. Extra complexity during insertion and deletion.
2. Requires adherence to strict balance rules.
3. Less efficient in high concurrent operation scenarios
4. Slower than some structures for specific operations.
5. Complex rotation for maintaining balance.

B+tree :-

→ Allows efficient insertion, deletion
→ B+ tree stores record in leaf node while B tree stores keys and records in both internal and leaf nodes
→ All leaf nodes are connected in a single linked list format.
→ B+ tree are designed to handle records that cannot fit in main memory. Internal nodes are stored in main memory, and leaf nodes in secondary memory storage.

Properties of B+ tree :-

1. Every node will have maximum (m) children and m-1 keys and minimum of m/2 children and $\frac{m-1}{2}$ keys, where m is order except the root node.
   keys → values inside a node

2. All paths must end at same level.

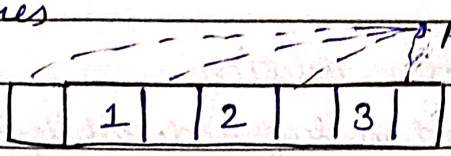
Insertion :-

Order - 4 elements 1, 2, 3, 4, 5, 6, 7

Order = 4
Maximum Children (m) = 4
Minimum Children (m/2) = 2

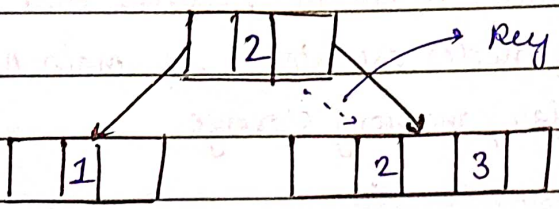
Maximum Keys (m-1) = 3
Minimum Keys ($\lceil \frac{m-1}{2} \rceil$) = 1

1. Maximum we can store 3 Keys (elements or values)
Insert 3 values

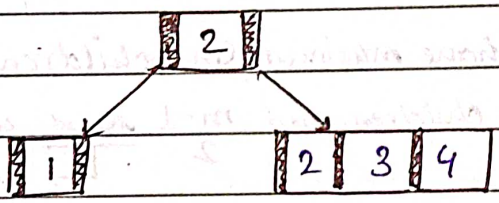


*Follow rules of BST

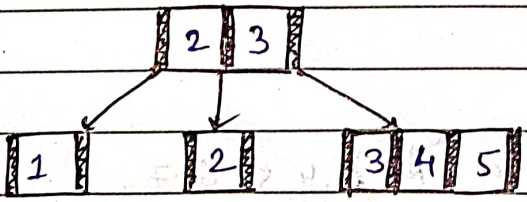
2. Adding 4 will overflow, split from middle



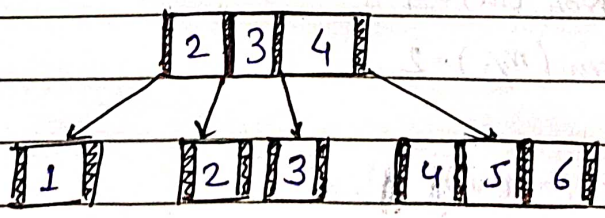
3. Add 4 now



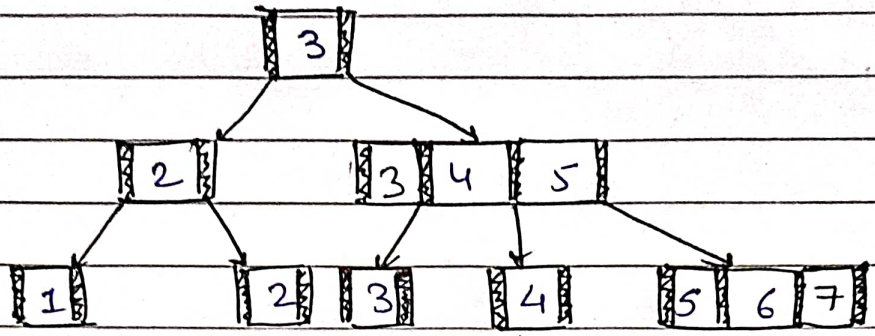
4. Adding 5 will overflow split and add 5



5. Add 6 will overflow split and Add 6.



7. Add 7 will overflow so spliting will cause overflow in node [2,3,4] so split this node.



B+ tree Advantages :-

1. Keys are used for indexing.
2. search is faster as data is only stored in leaf nodes.
3. Equal number of disk access to fetch record.
4. Height is always balanced compared to B tree.
5. Data stored is accessible by both sequential and direct manner.

B+ tree disadvantages :-

1. Complex split and merge operation.
2. Lack strict balance due to height variations.
3. Performance may vary from AVL tree.
4. Searching may be slower than AVL
5. Not as much efficient require specific requirement applications.

Time and Space Complexity :- $O(\log n)$ and $O(n)$

Time ↵ ↳ space

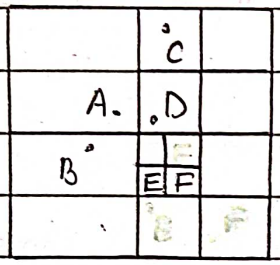
Quad Tree and Oct Tree :-

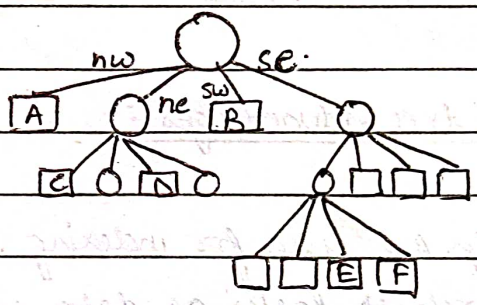
1. Quadtree :-

→ 2D spatial tree structure, commonly used in image compression.

→ Used in collision Detection in 2D.

→ Graphical Information System (GIS)





Advantages :-

→ Efficient Space partitioning : provides improved indexing and searchi

→ Adaptability : adapts changes in the distribution of object.

Disadvantages :-

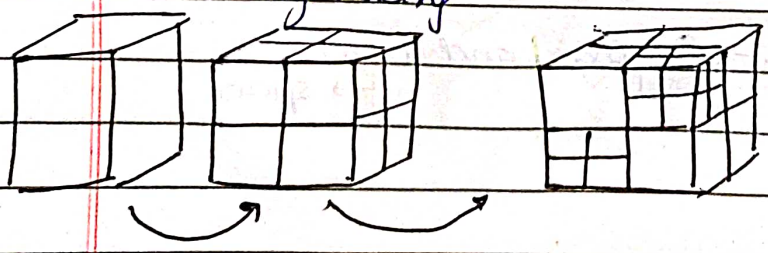
→ Grid Alignment : Irregular distributions of grid objects

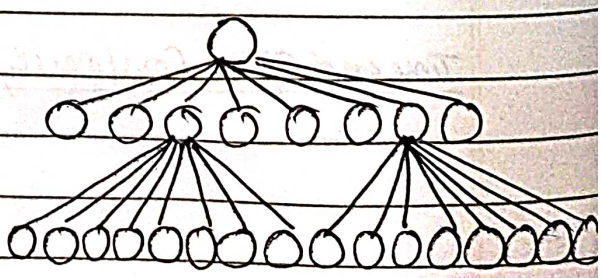
→ Potential Node Overlap : nodes overlap increasing memory usage.

2. Oct Tree :-

→ 3D spatial tree, commonly used for 3D applications like graphic rendering and collision detection.

→ Ray Tracing





Advantages :-

→ Space Efficiency : saves space by representing empty or uniform regions.
→ Fast Spatial Query : Searching more fast.
→ Dynamic Adaptability : dynamically adapt changes in distribution of object

Disadvantages :-

→ Complex Implementation : due to dynamic adaptability.
→ Memory Overhead : overweight for small data sets.