Trees in Data Structure:

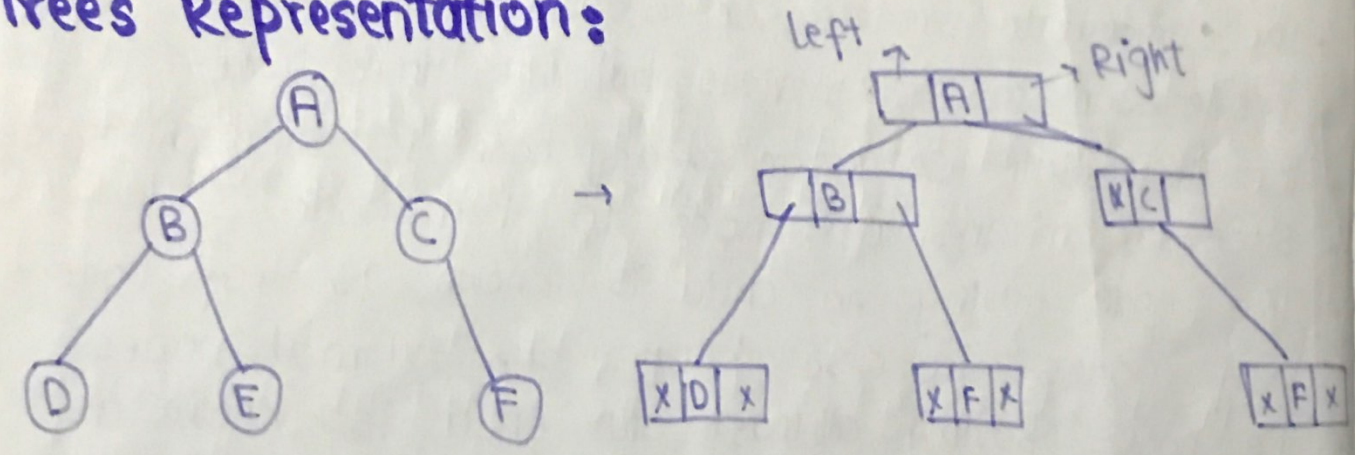
- Non-linear data structure
- Arranged in hierarcy way (Data Arranged)
- Simply example of parent to child or University → faculty means director to poen.
- We can only access the data from top to bottom.
- The data in tree is known as nodes and the top node is known as root node.
- Trees can be defined as a collection of entities (nodes) linked together to stimulate a hierarchy.
- The first node is root node and the root node has no parent.
- The immediate predecessor of any node is called parent node.
- Predecessor means previous node to that node.
- The immediate successor of any node is called child node.
- Sucessor means next node to that node.
- The node having no child is known as leaf node.
- Leaf nodes are also known as external nodes.
- The nodes having atleast one child is known as non-leaf nodes and that are other than leaf nodes. and non-leaf nodes are also known as internal nodes.
- Edge is the link between two nodes.
- Path is the way from source node to destination node.
- Any predecessor node on the path from root to that node is called ancesstors. (All previous)
- Any sucessor node on the path from that node to the leaf node is known as decendant. (All next).
- subtree are trees having node and all decendants.

- Siblings are the all the children of same parent of the node.
- Degree is the no. of children of the node.
- max degree among all nodes are degree of a tree.
- depth of the node is the legth of path from root to that node. (no. of edges).
- height of a node is the no. of edges in the longest path from that node to a leaf.
- level of node:- is equal to the depth of the node.
- height of tree is equal to the height of root node.

→ n nodes = n-1 edges

- Binary tree is the tree of maximum 2 childs.

Trees Representation:



→ Simply Representation of each node.

```
struct node {
    int data;
    struct node *left;
    struct node *right;
};
```

Working:-

- First they can enter the all left child of the node.
- If you enter -1 mean return 0 to the left child and they ask the right child of the lastest node.
- Basically it can store in the stack when the recursion create function is called at this place new stack is created and when you enter (-1) the stack is deleted and back to the position where create is call and then remaing lines for right child will be executed. When you enter value of right child they create another stack and ask the left child of the right one.
- When the left portion is completed they they shifted to the right portion and last return n.

means root = [n] → first node address.

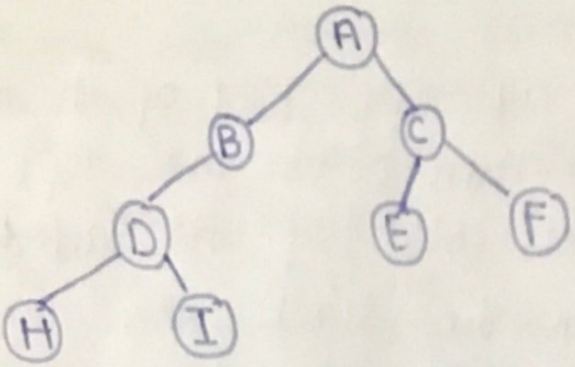
Array Representation of Binary Tree:

- Sequential Representation.
- Fill the index of Array of tree from left to Right.
- Using Array you don't know what is the parent and child of which node.
- Using formula you can calculate the parent and child of node.
- If a node is at i^{th} index

Left child = $[(2 \times i) + 1]$

Right Child = $[(2 \times i) + 2]$

Parent = $\left\lfloor \frac{i-1}{2} \right\rfloor$ → ceil value



case 1

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

if a node is at i^{th} index

Left child = $[(2 \times i) + 1]$

Right child = $[(2 \times i) + 2]$

Parent = $\left\lfloor \frac{(i-1)}{2} \right\rfloor$ → ceil

case 2

A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

if a node is at i^{th} index

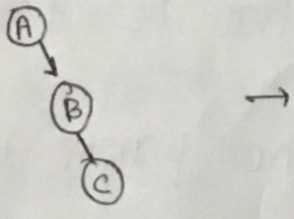
Left child = $[2 \times i]$

Right child = $[(2 \times i] + 1]$

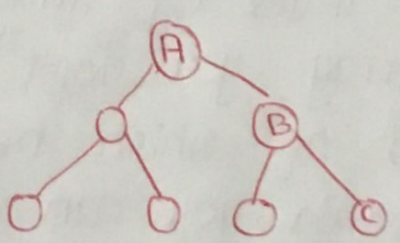
Parent = $\lfloor \frac{i}{2} \rfloor$ → ceil

→ This is only possible for complete binary trees means all the levels are completely filled expect the last level and in the last level the data are stored as left as possible.

→ If the tree are not complete first convert to the complete binary tree using empty nodes and then represent in an array.



→



A	B	C
0	1	2

X

A	-	B	-	-	-	C
0	1	2	3	4	5	6

✓

- This is the wastage of memory, but the right way to represent in an array.

Traversal of Binary Tree

Binary Tree & its types:

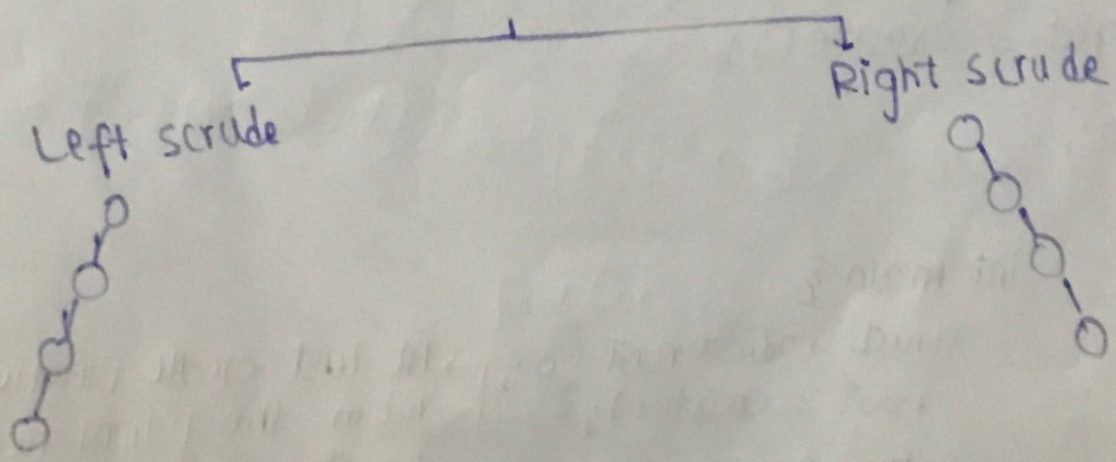
- Binary trees are the tree in which each nodes have at most 2 children. (0, 1, 2)
- Maximum no of nodes is possible at any level i is 2^i
- Maximum no. of nodes of height h = $2^{h+1} - 1$
- Minimum no. of nodes of heigh h = h + 1

Types:

- Full /proper/strict → Each node have either 0 or 2 children.
  - ↳ no. of leaf nodes = no. of internal node + 1
- complete Binary tree → All the nodes have maximum no. of childrens except last level and in the last level fill the nodes from left to right.
- Perfect Binary → All internal nodes have 2 childrens & all leaves are at same level.

→ Every perfect binary tree is full or complete binary.

- Degenerate Binary Tree → All the internal nodes have only one child.



Implementation of Binary Tree:

Write this code before creating the create function.

```
#include<iostream>
using namespace std;
Struct node{
        int data;
    Struct node * left;
    struct node * right;
};
```

Create (): →Function to implement a binary tree.

```
struct node *create() -> //pointer type bcz it return node
{ Struct node *n = new node(); // New Node
    int x;
 cout << "Enter data (-1 for no node)";
    cin >> x;
if (x == -1) {
        return 0;                    // Return to the previous stack
    }
  n -> data = x;
cout << "Enter Left child of" <<x;
  n -> left = create();              // Recursion
cout << "Enter Right child of " << x;
  n -> right = create();
  return n;
}
```

→ Main body:-

```
int main{
    struct node *root =0;
     root = create();
    return 0;
}
```

→ At last create function return the parent node to the root.

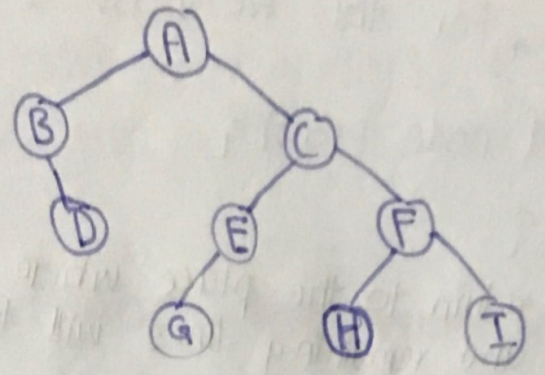
Traversal of Binary Tree:

- 3 types of traversal of a Binary Tree.

→ Inorder (left, root, right)

→ Preorder (root, left, right)

→ Postorder (left, right, root)



Inorder:- start from root first traverse the left part, then root and then traverse the right part.

B, D, A, G, E, C, H, F, I

Preorder:- start from the root first traverse the root then its left part and then the right part.

A, B, D, C, E, G, F, H, I

Postorder:- First traverse the left part, then traverse the right part and in last traverse the root.

D, B, G, E, H, I, F, C, A

Inorder Implementation:

- We use Binary tree is already created using create function
- Same as create function follow the recursion principle.
- In the main just call the function with its name and pass the root node.

```
inorder(root);
```
, For this we pass a pointer to node Argument.

```
void inorder(struct node *root){
    if(root==0){
        return; → return to the place where it was called.
    }              and remaining lines will be executed.
inorder(root->left);      Left
cout<<root->data;         root
inorder(root->right);     Right
}
```

Note:- It can follow the same approach create stack when a single inorder function is called and executed until (root==0) then return to previous stack and for right child executed.

Preorder:

```
void preorder(struct node *root){
    if(root==0){
        return;
    }
cout<<root->data;
preorder(root->left);
preorder(root->right);
}
```

Postorder:

```
void postorder(struct node *root){
    if (root==0){
        return;
    }
postorder (root->left);
postorder (root->right);
cout << root -> data;
}
```

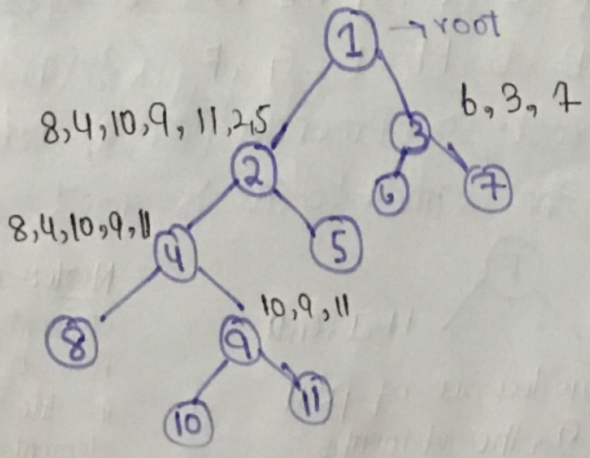
Construct a Binary Tree from Preorder & Inorder.

Preorder : 1,2,4,8,9,10,11,5,3,6,7 (root, left, right)

Inorder : 8,4,10,9,11,2,5,1,6,3,7 (left, root, right)

→ We can't find the root in the inorder because root is in between so find the root from preorder because root is on left.

→ After finding the root from preorder go on to the inorder check left to right the root no., the left elements from the node are the part of left and right are right part.



Note:

After find first root and its left and right part then again check the root of remaining parts by following the same procedure.

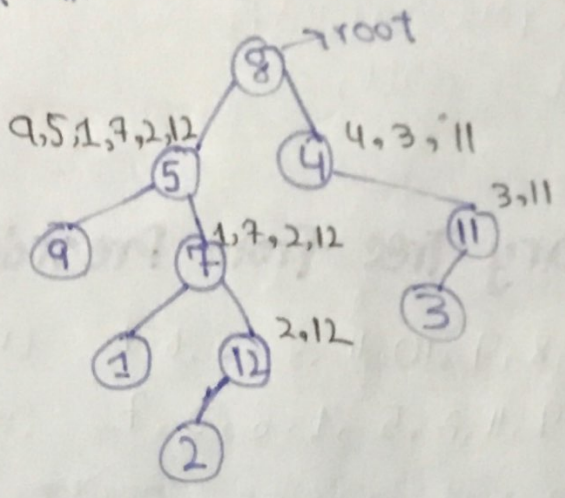
→ Again First and Inorder :- Inorder Traversal

Construct a Binary Tree from Post order & Inorder :

Postorder : 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 (left, right, root)

Inorder : 9, 5, 1, 7, 2, 12, 8, 4, 3, 11 (left, root, right).

→ Same not find root ~~from~~ in inorder so find root from postorder and in this case root is on right and check right to left in Inorder.



Construct a Binary Tree from postorder and Preorder:

→ When only postorder and preorder given it is not possible to construct a unique binary tree but you can construct a unique full binary tree.

Preorder : F, B, A, D, C, E, G, I, H (root, left, right)

Postorder : A, C, E, D, B, H, I, G, F (left, right, root)

→ Step 1 is find the root. So, Preorder (first) and Postorder (last) data will be the same. so, it is root. (F)

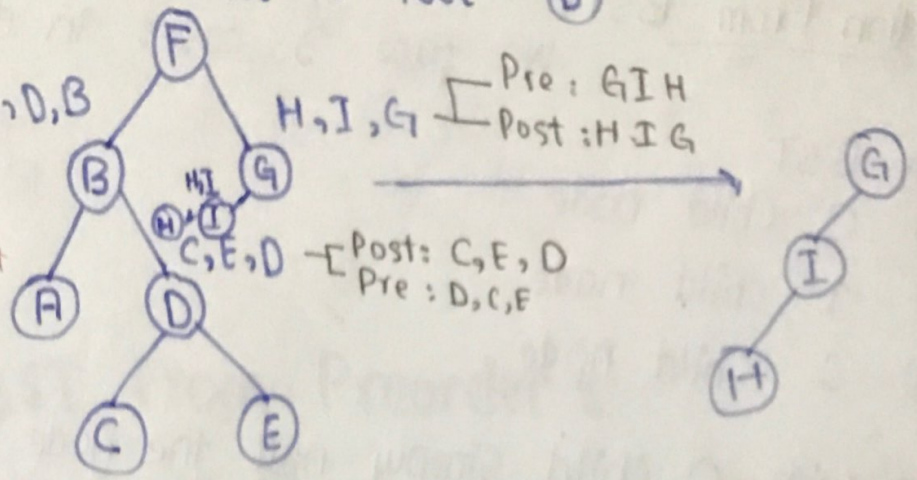
→ After this Find the predecessor of post order (previous) it is A, the elements before G are part of preorder and postorder already Find.

Preorder : B, A, D, C, E

Postorder : A, C, E, D, B

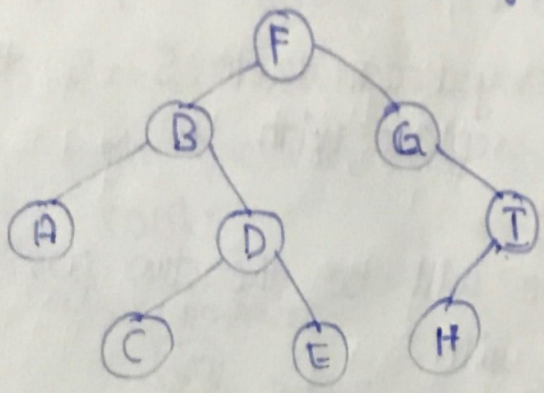
Note:- Check the 2nd element in preorder and it is B → then check B in the post order. All elements till B are part of left subtree, and remaining are right.

→Again First and last is root. (B)



• A will be the next element in preorder so till A will be the part of left tree in postorder.

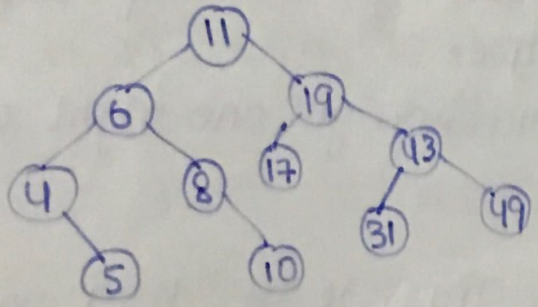
→This Binary Tree is not correct because we already say we can't construct a unique binary tree. The given tree is correct.



Binary Search Tree: (BST)

- Binary means each node of tree has atmost 2 childs.
- Left subtree value are lesser than that node and right subtree are greater than that node. (In BST)
- Construct BST of the given numbers from left to right.

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



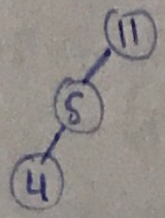
Deletion From BST:

We face 3 cases in deleting a node from BST.

① 0 child node
② 1 child node
③ 2 child node

① → with 0 child simply null the node and break the link of the node.

② → with 1 child simple a node can replace with his children.



→ you can delete 5, so the 5 can replace with 4.

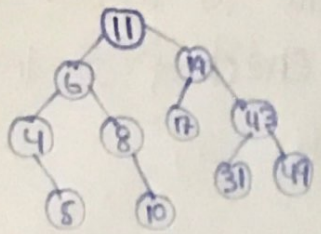
③ → with 2 childs, the will be the two possibilities to replace that node with
  (i) Inorder Predecessor
  (ii) Inorder Successor

(i) In inorder Predecessor you can replace with largest element in the left subtree.
→ To Find inorder predecessor go one Left and then find the right most child.

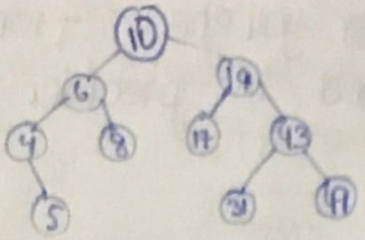
(ii) In inorder Successor you can replace with smallest element in the right subtree.
→ To Find inorder successor go one right and then find the left most child.

→ ~~After~~ Take Inorder Traversal of BST and your data is in sorted mood.

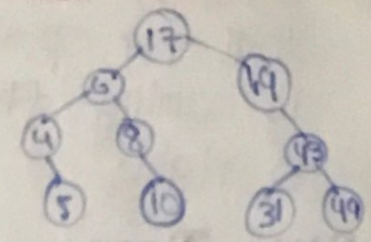
BST



Inorder Predecessor



Inorder Successor



• Delete (11)

Construct BST From Preorder :

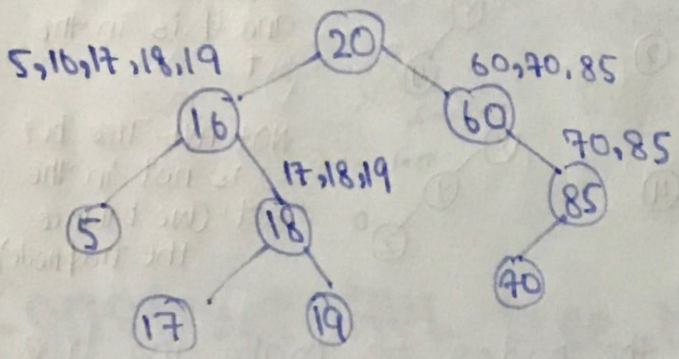
Preorder: 20, 16, 5, 18, 17, 19, 60, 85, 70 (Root, left, Right)

→ convert this preorder to Inorder

→ Inorder of BST always in ascending order.

Inorder : 5, 16, 17, 18, 19, 20, 60, 70, 85 (Left, Root, Right)

→ Find root from preorder bcz in preorder first node will be the root.



- Left elements from 20 will be the left subtree and right elements will be the right subtree in inorder.
- Out of subtree find which no. is coming first in preorder and that will be your root.
- left part from this root in inorder will left subtree and right for right.

Construct BST From Postorder :

Postorder : 5, 17, 19, 18, 16, 70, 85, 60, 20 (Left, Right, Root)

→ Convert this postorder to inorder

→ Inorder of BST always in ascending order.

Inorder : 5, 16, 17, 18, 19, 20, 60, 70, 85 (Left, Root, Right)

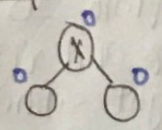
→ Find root from postorder bcz in post order last node will be the root.

- Construct same as Preorder but the only difference is that for postorder you check from right to left which is coming first and in preorder you check from left to right.

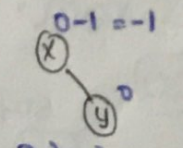
AVL Tree:

- It is a BST (Binary Search Tree).
- height of left subtree - height of right subtree = {-1, 0, 1}
  → This difference is called BALANCE FACTOR.
- Insert data in AVL same as BST. Left is smaller and right is greater.
- Duplicate value is not allowed in BST.
- Check Balance Factor on each node.

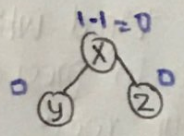
Example: (x, y, z)

① 

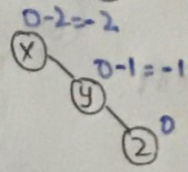
AVL-Tree bcz 0 is in set.

② 

AVL-Tree bcz 0 and -1 is in the set

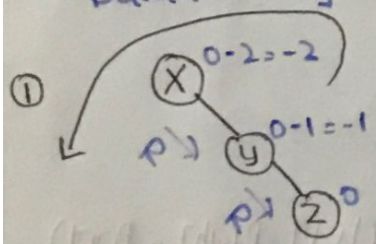
③ 

AVL-Tree bcz 0 is in the set

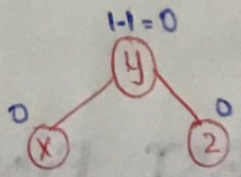
④ 

Not AVL-Tree bcz -2 is not in the set. (we balance the tree/node)

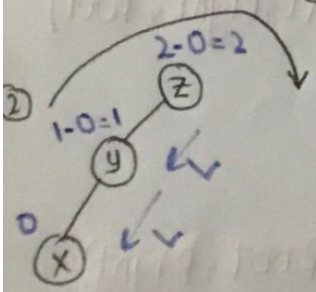
→ For Non-AVL Trees we balance the nodes to make the tree balance by rotating the node.

① 

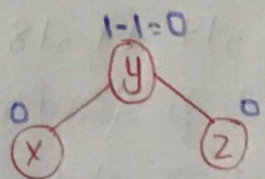
Left Rotation → After



AVL

② 

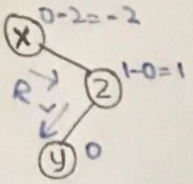
Right Rotation → After



AVL

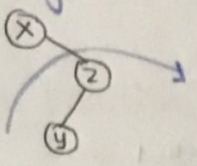
- Tree unbalance bcz again and again Left Insertion.
- We always maintain the order Bcz it has BST.

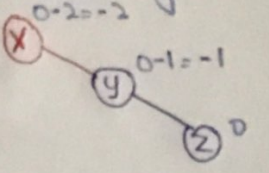
③ X, Z, y



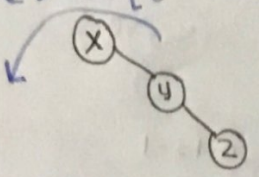
• unBalance Bcz of Right left rotation. To balance this type of tree need two rotations.

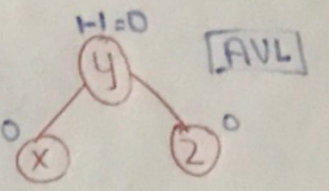
(i) Right rotation of only z and y.



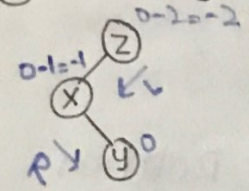


(ii) Left Rotation



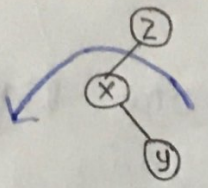


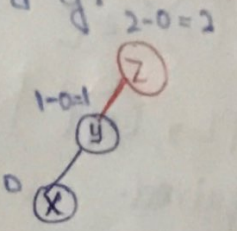
④ Z, X, y



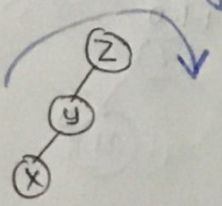
• Same Two Rotation Required.

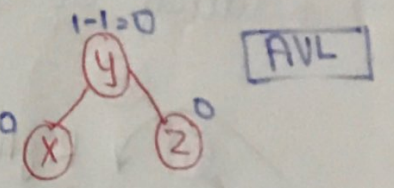
(i) Left rotation of x and y.





(ii) Right Rotation



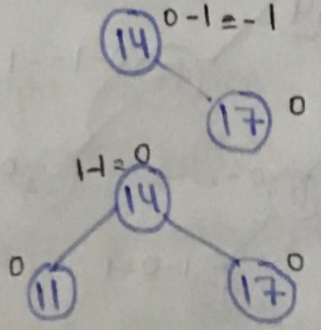


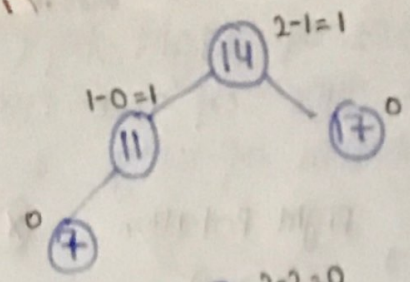
Construct AVL tree by inserting the following Data.

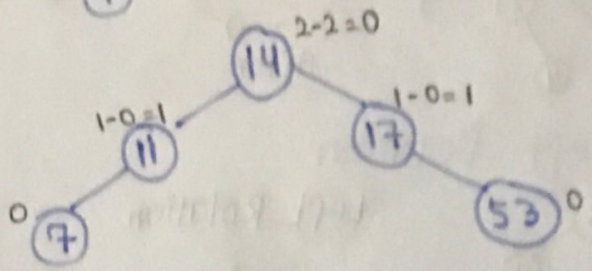
14, 17, 11, 7, 53, 4, 13, 12, 8, ~~60, 19, 16, 20~~

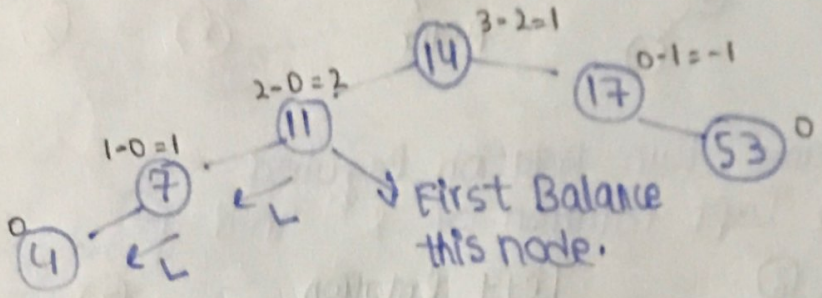
→ Insert same as BST.

→ Check Balance Factor After inserting each node.



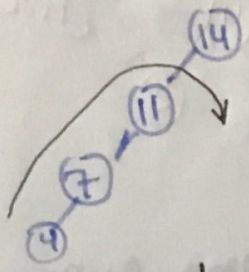




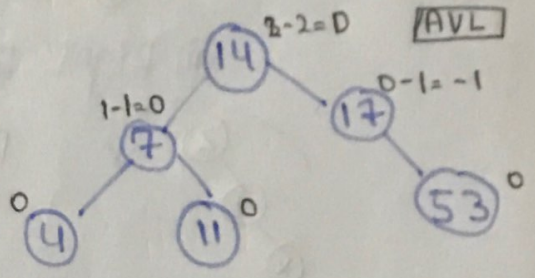


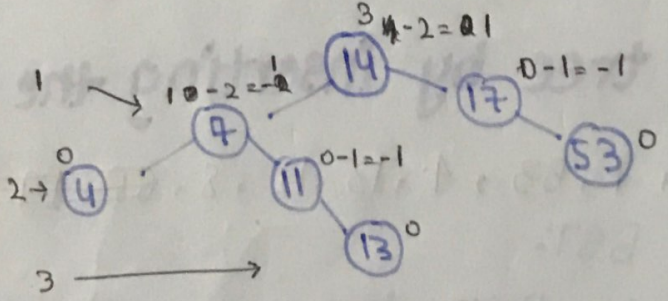
→ Left, Left Insertion means only one Right Rotation Required.

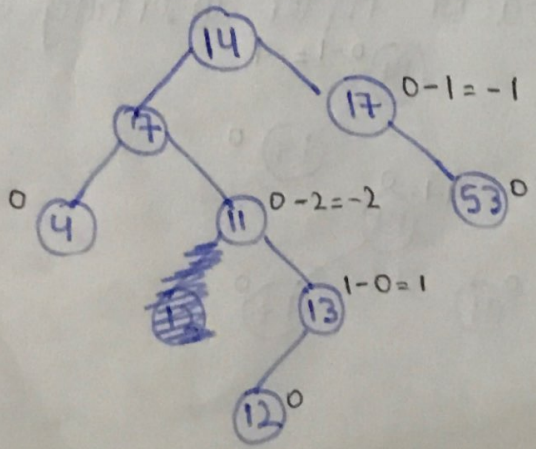
Ex → Rotation of only 4, 7, 11



Right Rotation After →

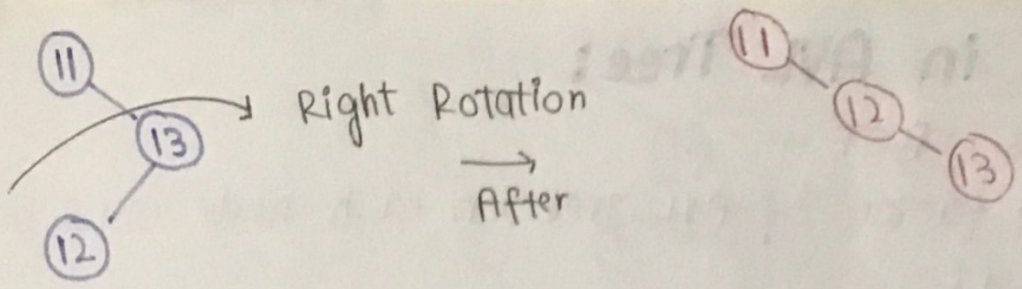


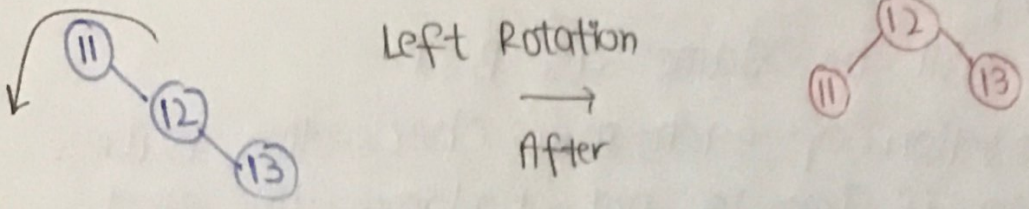




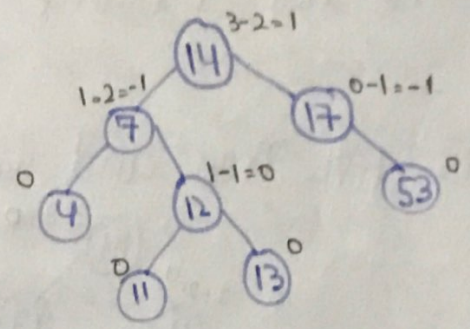
• 11 will be the unbalanced, due to R, L

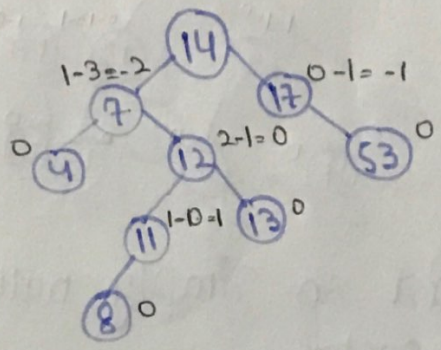
So, two rotations req. right, left.



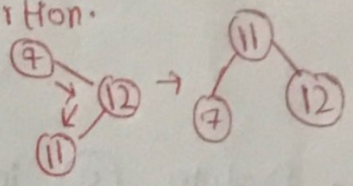


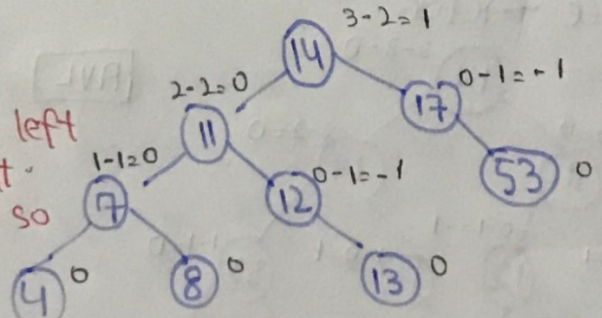
- Median element will be the root and remaining 2 will be the childs.





- 7 will be the effected node due to right-left insertion.





- 8 will be the left part of 11, but already 7 there so 8 will be the right of 7 due to BST.

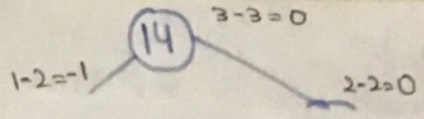
AVL Rotations:-

Left, Left → Right Rotation

Right, Right → Left Rotation

Left, Right → Left Rotation, Right Rotation

Right, Left → Right Rotation, Left Rotation

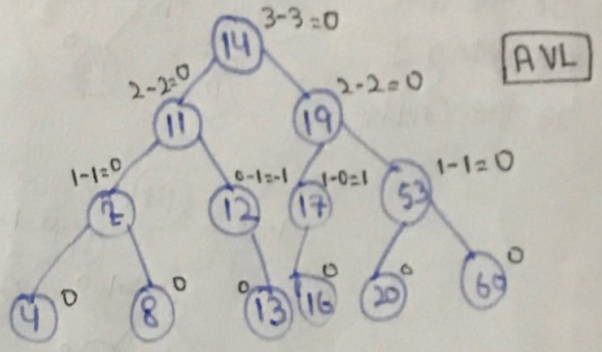


AVL

Deletion in AVL Tree:

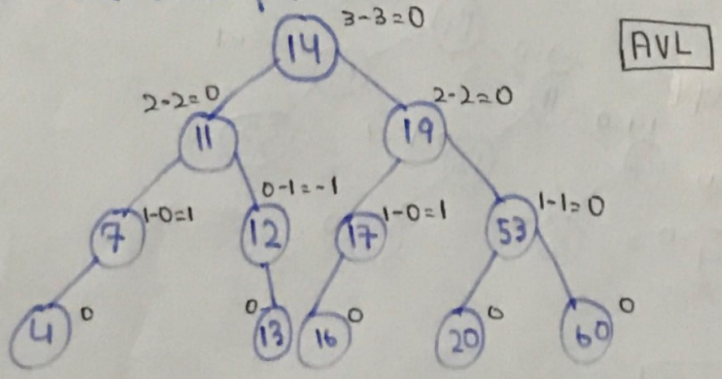
- AVL is BST
- Balance factor of AVL tree on each node must be $\{1, 0, -1\}$.
- Deletion will be same as BST.
- After Deletion of each node check the Balance factor again. if Tree is not balance the first balance the tree.
- Example the given is AVL Tree and we delete:

8, 7, 11, 14, 17



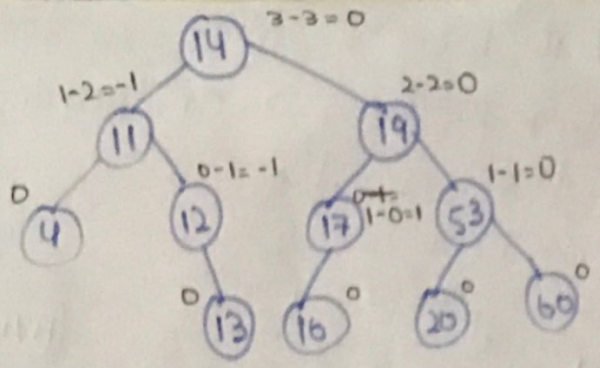
AVL

- Delete (8) in the last so simply null the link, and check again balance factor.

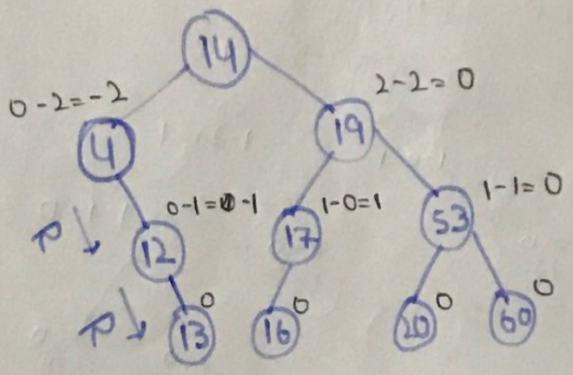


AVL

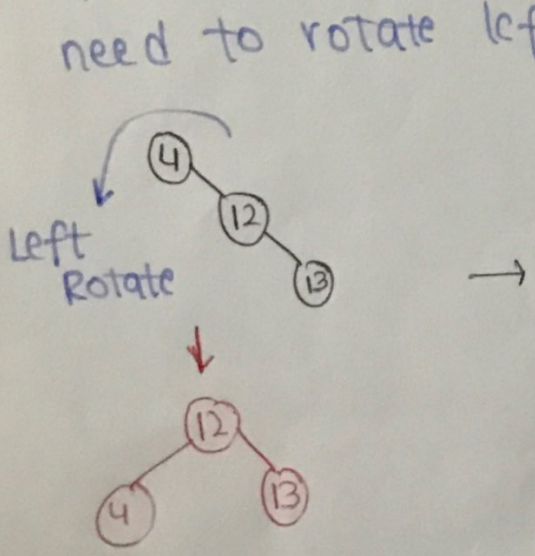
- This is still AVL no need to rotate any node.
- Delete (7) it has one child so replace child with the parent.

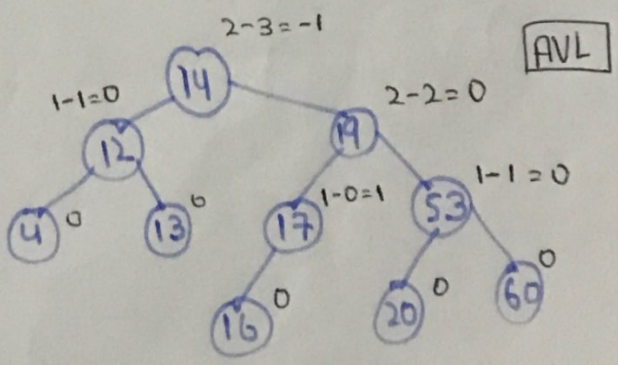


- This is still AVL no need to Balance.
- Delete (11) it has 2 child so it has 2 cases one is to replace with its inorder predecessor or ~~into~~ second is to replace with its inorder successor.
- inorder predecessor is (4)
- inorder successor is (12)



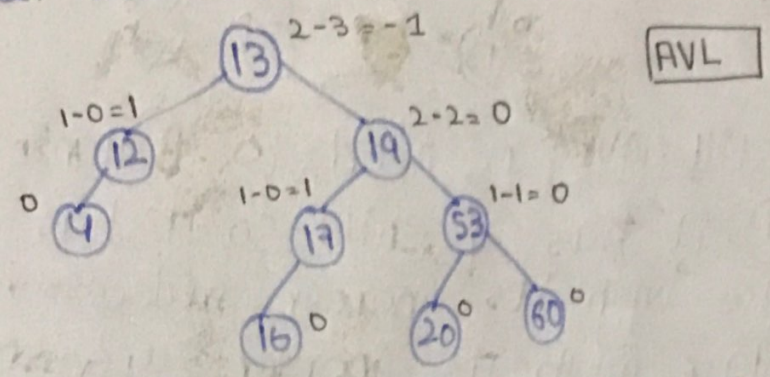
- (4) is the effected node so need to maintain balance.
- It is inbalance due to right, right insertions so need to rotate left.





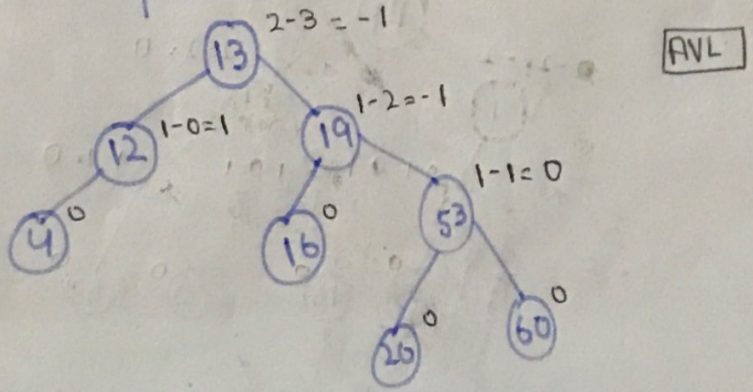
↳ This is Now AVL

- Now Delete (14) it has 2 childs so same 2 steps first we finds its Inorder predecessor or Inorder Successor.
  - → inorder predecessor = 13
  - → inorder successor = 16



AVL

- This is AVL so no need to balance/rotate.
- Now Delete (17) it has one child so replace the child with the parent.



AVL

- This is still AVL so no need to balance/rotate.