LEARN DSA WITH C++

WEEK :: 10

LEARN DSA
WITH C++

PDF

COURSE INSTRUCTOR BY
ROHIT NEGI

MADE BY-
PRADUM SINGHA

Check Profile

My profile

LEARN DSA WITH C++

WEEK :: 10 **DAY: 01** **DATE: 13-06-2023**

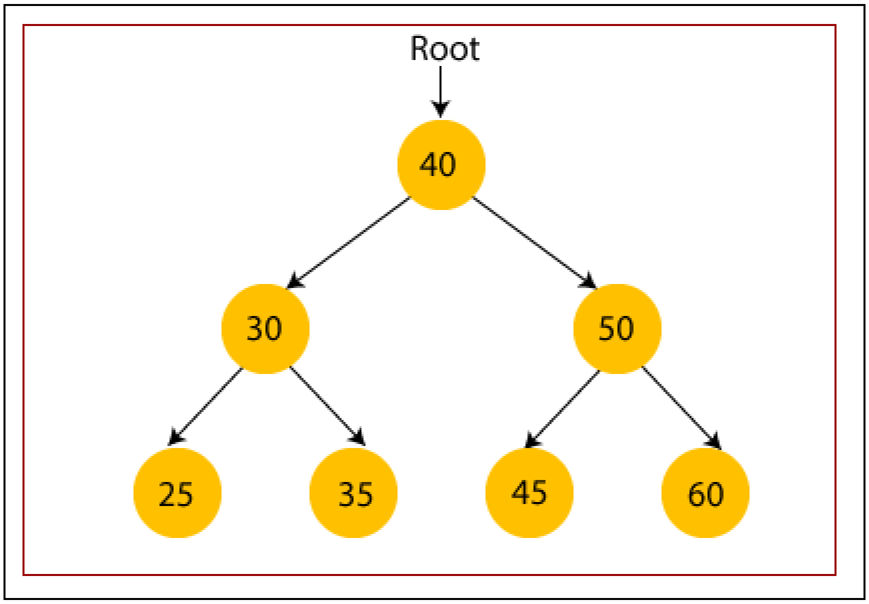
TREES

Construct Tree from Inorder & Preorder << [GeeksforGeeks] >>

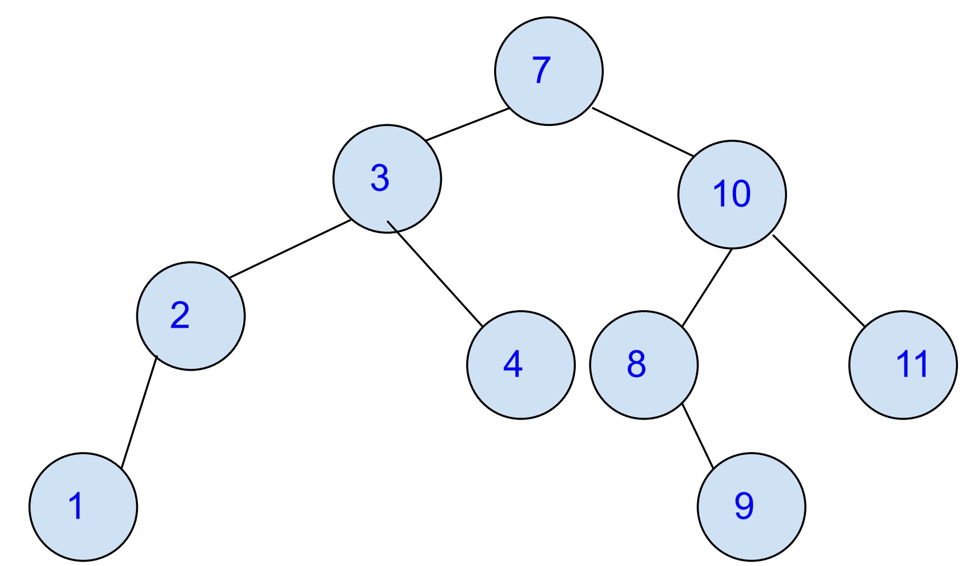
```
class Solution{
    public:

    int Find(int in[], int num, int start, int end)
    {
        for(int i= start; i<= end; i++)
        {
            if(in[i] == num)
            return i;
        }
    };

    Node * Tree(int in[], int pre[], int start, int end, int index)
    {
        if(start>end)
        return NULL;

        // Build the Node
        Node *root = new Node(pre[index]);

        // Find index in inorder Left & Right
        int i= Find(in, pre[index], start, end);

        root ->left = Tree(in, pre, start, i-1, index+1);
        root ->right = Tree(in, pre, i+1, end, index +(i-start) +1);

        return root;
    }

    Node* buildTree(int in[],int pre[], int n)
    {
        // Code here
        Node *root;
        root = Tree(in, pre, 0, n-1, 0);
        return root;
    }
};
```

BINARY SEARCH TREES

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of the left node must be smaller than the parent node, and the value of the right node must be greater than the parent node.



Create Tree :: { 7, 3, 10, 8, 2, 4, 11, 9, 1}



Time Complexity ::-

Worst Complexity :- O(N)
Average Complexity :- O(logN)

Create Binary Search Tree :-

```
#include<iostream>
using namespace std;
class Node
{
    public:
    int data;
```

```
    Node *left;
    Node *right;


    Node(int x)
    {
        data =x;
        left =NULL;
        right =NULL;
    }
};
Node *BST(Node *root, int value)
{
    if(!root)
    {
        root = new Node(value);
        return root;
    }
    // Left Side
    if(root -> data >value)
    {
        root ->left = BST(root->left, value);
        return root;
    }
    //Right Side
    else
    {
        root->right =BST(root ->right, value);
        return root;
    }
};
void inorder(Node *root)
{
    if(!root)
    return;

    inorder(root ->left);
    cout<<root->data<<" ";
    inorder(root ->right);

}
int main()
{
    int arr[10] = {10, 13, 4, 8, 11, 19, 2, 7, 18, 23};
    Node *root = NULL;
    for(int i=0; i<10; i++)
    root = BST(root, arr[i]);

    inorder(root);

return 0;
};
```

Search a node in BST << GeeksforGeeks >>

```
bool search(Node* root, int x) {
    // Your code here
    if(!root)
    return 0;

    if(root->data ==x)
    return 1;

    if(root->data >x)
    return search(root->left, x);

    if(root->data <x)
    return search(root->right, x);
}
```

Minimum element in BST << GeeksforGeeks >>

```
int minValue(Node* root) {
    // Code here
    if(!root)
    return -1;

    if(!root-> left)
    return root->data;

    return minValue(root ->left);
```

Kth largest element in BST << GeeksforGeeks >>

```
class Solution
{
    public:
    void Find (Node *root, int &k, int &ans)
    {
        if(!root || k<0)
        return;

        Find(root->right, k, ans);
        k--;
        if(k==0)
        {
            ans = root->data;
            return;
        }
        Find(root->left, k, ans);
    };

    int kthLargest(Node *root, int K)
    {
        //Your code here
        int ans;
        Find(root, K, ans);
```

```
        return ans;
    }
};
```

Kth Smallest Element In Tree << [InterviewBit]() >>

```
void inorder(TreeNode *A, vector<int>&v)
 {
     if(!A)
     return;

     inorder(A-> left, v);
     v.push_back(A->val);
     inorder(A->right, v);
 }
int Solution::kthsmallest(TreeNode* A, int B)
{
    vector<int>v;
    inorder(A,v);
    return v[B-1];
}
```

Check for BST << [GeeksforGeeks]() >>

```
class Solution {
public:
    // Function to check whether a Binary Tree is BST or not.
    void Find(Node* root, int& prev_val, bool& ans) {
        if (!root || !ans)
            return;

        // Left Side
        Find(root->left, prev_val, ans);

        if (prev_val >= root->data) {
            ans = false;
            return;
        }
        prev_val = root->data; // Update prev_val

        // Right Side
        Find(root->right, prev_val, ans);
    }
    bool isBST(Node* root) {
        int prev_val = INT_MIN;
        bool ans = true;
        Find(root, prev_val, ans);
        return ans;
    }
};
```

LEARN DSA WITH C++

WEEK :: 10 **DAY: 02** **DATE: 14-06-2023**

BINARY SEARCH TREES QUESTION

Sum of k smallest elements in BST << [GeeksforGeeks] >>

```
void Find(Node *root, int &k, int &total)
{
    if(!root || k<0)
    return;

    Find(root->left,k,total);
    k--;
    if(k>=0)
    total += root->data;

    Find(root->right, k, total);
}

int sum(Node* root, int k)
{
    // Your code here

    int total = 0;
    Find(root, k, total);
    return total;
}
```

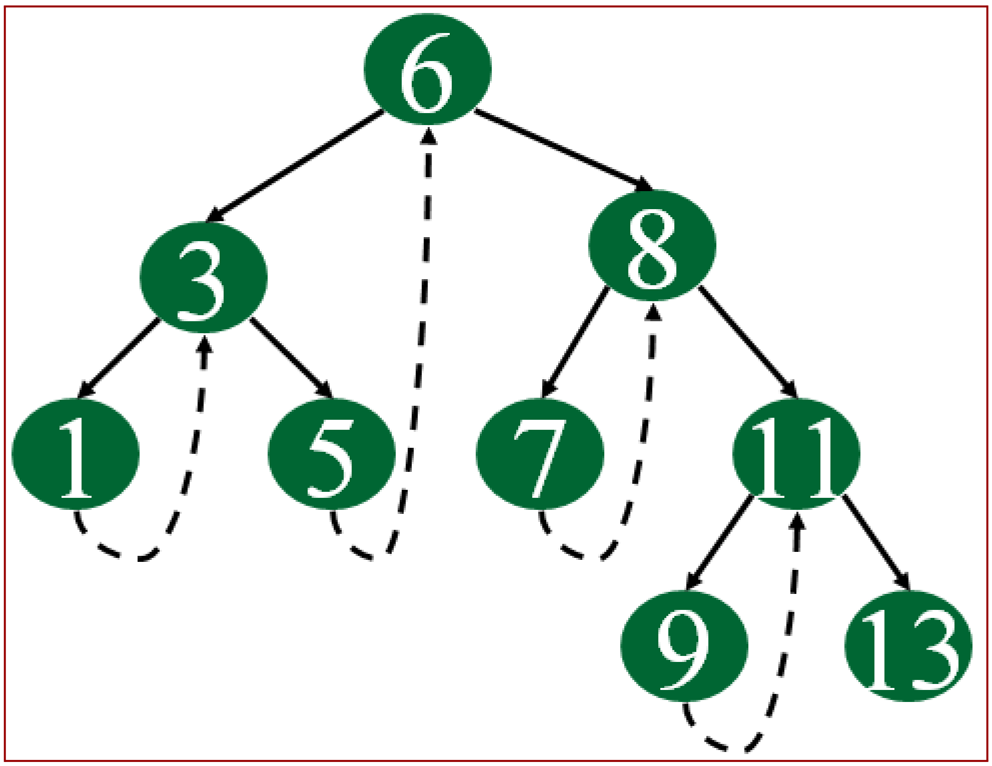
Inorder Successor in BST << [GeeksforGeeks] >>

```
class Solution {
public:
    Node* inOrderSuccessor(Node* root, Node* x) {
        if (x->right != NULL) {
            // If the right subtree of x exists,
            // find the minimum value in that subtree.
            x = x->right;
            while (x->left != NULL)
                x = x->left;
            return x;
        } else {
            // If the right subtree of x is NULL,
            // traverse the tree to find the successor.
            Node* successor = NULL;
            while (root != NULL) {
                if (x->data < root->data) {
                    successor = root;
                    root = root->left;
                } else if (x->data > root->data) {
                    root = root->right;
                } else {
                    break;
```

```
                }
            }
            return successor;
        }
    }
};
```

MORRIS TRAVERSAL

Morris traversal is a tree traversal algorithm that does *not* use Recursion, Stack, and Queue.



Create Morris Traversal ::-

```
while(root)
{
   if(!root ->left)
    {
      cout<<root->data;
      root= root ->right;
     }
   else
    {
      Node *curr = root ->left;
      while(curr ->right && curr -> right ! = root)
        {
           curr = curr ->right;
        }
```

```
if(!curr ->right)
 {
   curr ->right = root;
   root = root ->left;
 }
else
 {
    curr -> right = NULL;
    cout << root->data;
    Root = root->right;
}
```

Inorder Traversal <<[GeeksforGeeks]>>

```
class Solution {
  public:
    // Function to return a list containing the inorder traversal of the tree.
    vector<int> inOrder(Node* root) {
        // Your code here
        vector<int>ans;

        while(root)
        {
            // Left doesn't exist
            if(!root ->left)
            {
                ans.push_back(root->data);
                root = root->right;
            }
            // Left Exist
            else
            {
                Node *curr = root->left;
                //Rightmost Node
                while(curr->right && curr ->right !=root)
                curr = curr ->right;

                //from the link
                if(!curr ->right)
                {
                    curr -> right = root;
                    root = root ->left;
                }
                else
                {
                    curr ->right = NULL;
                    ans.push_back(root ->data);
                    root = root ->right;
                }
            }
        }
        return ans;
    }
};
```

Inorder Successor in BST << [GeeksforGeeks] >>

```
class Solution{
  public:
    // returns the inorder successor of the Node x in BST (rooted at 'root')
    Node * inOrderSuccessor(Node *root, Node *x)
    {
        bool flag =0;
        while(root)
        {
            // Left doesn't exist
            if(!root ->left)
            {
                if(flag ==1)
                return root;
                if(root ==x)
                flag =1;
                root = root->right;
            }
            // Left Exist
            else
            {
                Node *curr = root->left;
                //Right Most Node
                while(curr->right && curr ->right !=root)
                curr = curr ->right;

                //from the link
                if(!curr ->right)
                {
                    curr -> right = root;
                    root = root ->left;
                }
                else
                {
                    curr ->right = NULL;
                    if(flag ==1)
                    return root;
                    if(root ==x)
                    flag =1;
                    root = root ->right;
                }
            }
        }

    }
};
```

Flatten binary tree to linked list << [GeeksforGeeks] >>

```
class Solution
{
    public:
    void flatten(Node *root)
    {
        //code here
```

```
        while(root)
        {
            // root left exist
            if(root ->left)
            {
                Node *curr = root ->left;
                while(curr ->right)
                {
                    curr = curr->right;
                };
                curr ->right = root ->right;
                root ->right = root->left;
                root ->left =NULL;
            }
            //Root left doesn't exist
            else
            {
                root = root->right;
            }
        }
    }
};
```

LEARN DSA WITH C++

WEEK :: 10 **DAY: 03** **DATE: 15-06-2023**

BINARY SEARCH TREES HARD QUESTION

Preorder to PostOrder << [GeeksforGeeks]() >>

```cpp
class Solution{
public:
    //Function that constructs BST from its preorder traversal.
    Node *Find(int pre[], int min, int max, int &index, int size)
    {
        if(index >= size || pre[index] > max)
            return NULL;

        Node *root = new Node;
        root->data = pre[index];
        index++;

        root->left = Find(pre, min, root->data, index, size);
        root->right = Find(pre, root->data, max, index, size);

        return root;
    }

    Node* post_order(int pre[], int size)
    {
        int min = INT_MIN, max = INT_MAX;
        int index = 0;
        return Find(pre, min, max, index, size);
    }
};
```

Find the Closest Element in BST << [GeeksforGeeks]() >>

```cpp
class Solution
{
    public:
    //Function to find the least absolute difference between any node
      //value of the BST and the given integer.
    int minDiff(Node *root, int K)
    {
        //Your code here
        if(!root)
        return INT_MAX;

        if(K==root ->data)
        return 0;

        else if(K>root ->data)
        return min(K- root->data, minDiff(root ->right, K));

        else
```

```
        return min(root ->data -K, minDiff(root ->left, K));
    }
};
```

Fixing Two swapped nodes of a BST << GeeksforGeeks >>

```
class Solution {
  public:

  void Find(Node *root, Node *&prev, Node *&first, Node *&second)
  {
      if(!root)
      return;

      //left side visit
      Find(root ->left, prev, first, second);

      // Apply operation in nodes
      if(prev)
      {
          //value decrease
          if(prev ->data > root ->data)
          {
              // First decrease
              if(!first && !second)
          {
              first = prev;
              second = root;
          }
          // second decrease
          else
          {
              second = root;
          }
        }

      };
      prev = root;

      // Right Side visit
      Find(root ->right, prev, first, second);

  };

    struct Node *correctBST(struct Node *root) {
        // code here
        Node *prev = NULL, *first =NULL, *second = NULL;
        Find(root, prev, first, second);
        int data = first ->data;
        first ->data = second ->data;
        second ->data = data;
        return root;
    }
};
```

LEARN DSA WITH C++

WEEK :: 10 **DAY: 04** **DATE: 16-06-2023**

BINARY SEARCH TREES HARD QUESTION MORE

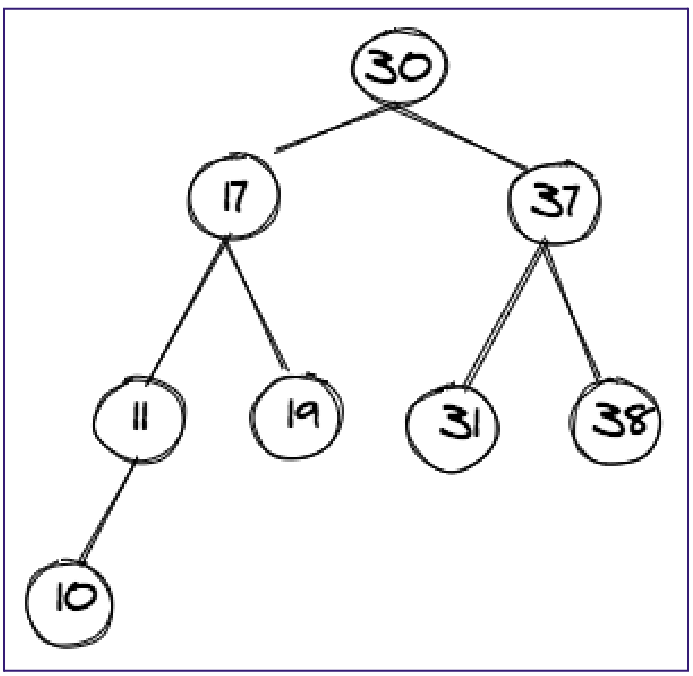
Delete a node from BST << [GeeksforGeeks] >>

```cpp
Node *minValue(Node *root)
{
    Node *current = root;
    while(current && current ->left)
    current = current ->left;

    return current;
}

Node *deleteNode(Node *root, int X) {
    // your code goes here

    if(!root)
    return NULL;

    // X value is equal to root value
    if(root->data == X)
    {
        // 0 child
        if(!root ->right && !root ->left)
        {
            delete root;
            return NULL;
        }
        // 1 left child
        else if(root ->left && !root ->right)
        {
            Node *temp = root ->left;
            delete root;
            return temp;
        }
        // 1 Right child
        else if(!root ->left && root ->right)
        {
            Node *temp = root ->right;
            delete root;
            return temp;
        }
        // 2 child
        else
        {
            Node *temp = minValue(root ->right);
            root ->data = temp ->data;
            root ->right = deleteNode(root ->right, temp ->data);
        }
    }
    // X value is Less then root value
    else if(X <root ->data)
```

```
    {
        root ->left = deleteNode(root ->left, X);
    }
    // X value is Greater then root value
    else
    {
        root ->right = deleteNode(root ->right, X);
    }
    return root;
}
```

BALANCE BINARY SEARCH TREE

Property of BBST :-
-1 <= (Left Height - Right Height) <= 1



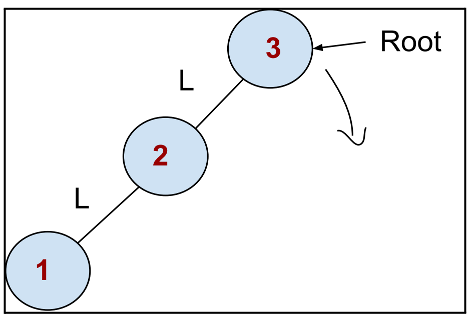
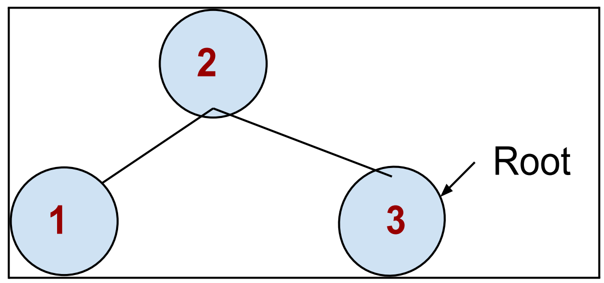
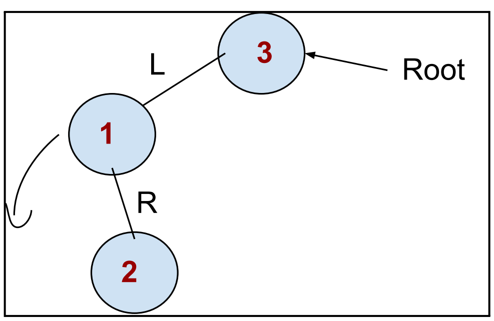
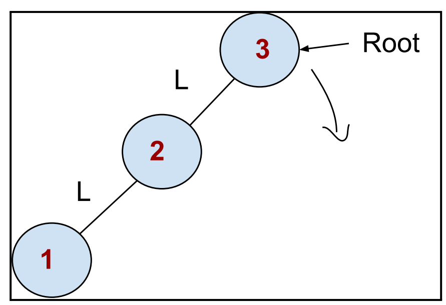
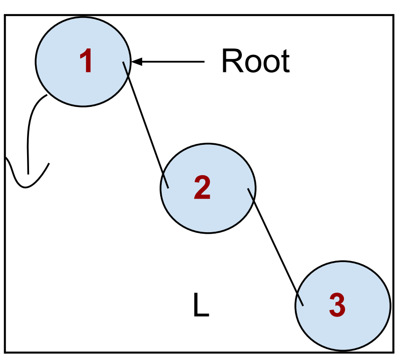
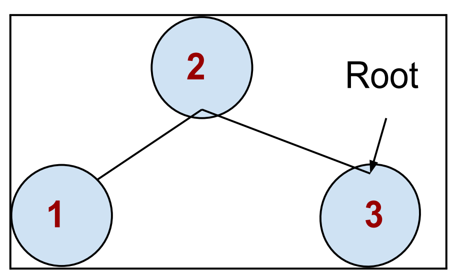
Create Balance Binary Search Tree :-

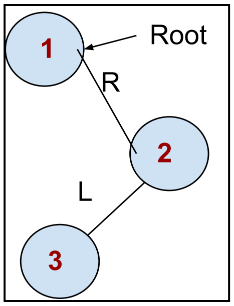
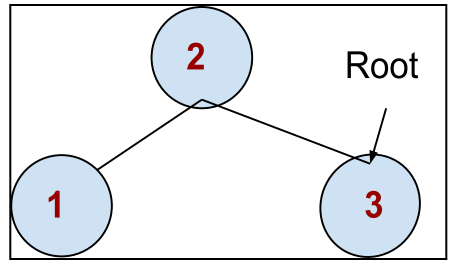
```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;

class Node
{
    public:
    int data;
```

```
    Node *left, *right;

    Node(int value)
    {
        data = value;
        left = right =NULL;
    }
};

Node *CreateBBST(vector<int> &v, int start, int end)
{
    if(start>end)
    return NULL;

    int mid = start + (end - start)/2;

    Node *root = new Node(v[mid]);

    root ->left = CreateBBST(v, start, mid-1);
    root ->right = CreateBBST(v, mid+1, end);
    return root;
};

void preorder(Node *root)
{
    if(!root)
    return;

    cout<<root->data<<" ";
    preorder(root ->left);
    preorder(root ->right);
}

int main()
{
    // Size of Array
    int n;
    cin>>n;
    vector<int>v(n);
    for(int i=0; i<n; i++)
    cin>>v[i];

    // sort the array
    sort(v.begin(), v.end());
    Node *root = CreateBBST(v, 0, n-1);
    preorder(root);

return 0;
};
```

BBST ages case :-	
Rotate -> left	**Rotate -> right**

Rotateleft(Node *root)
{
  Node *temp = root +right;
  root->right = temp ->left;
  temp ->left = root;
 	
  return temp;
}
	RotateRight(Node *root)
	{
	  Node * temp = root ->left;
	  root ->left = temp ->right;
	  root ->right = root;
	
	  return temp;
	}

Rotation for Balance of Binary Search Tree :-	
 1	 Right Rotation (root)
 2	 1st -> Left Rotation(root ->left) 2nd -> Right Rotation(root)
 3	 Left Rotation(root)

 4	 First make it

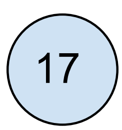
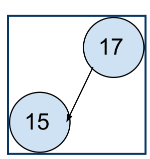
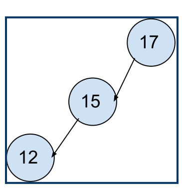
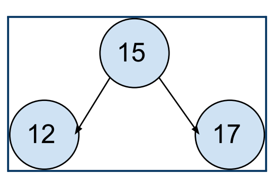
LEARN DSA WITH C++

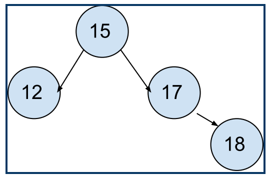
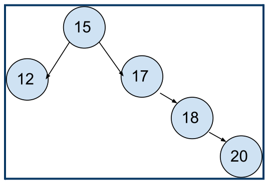
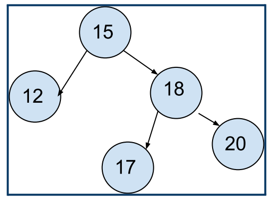
WEEK :: 10 **DAY**: 05 **DATE**: 27-06-2023

BALANCE BINARY SEARCH TREES AND AVL

How to Know which side unbalance of tree:-
abs(left height - right height) > 1 —> unbalance balance = Left height - Right Height; balance >1 —-> left side (unbalance) balance <-1 —--> Right side (unbalance)

Where Unbalance in Tree :-
Compare height in Node - which side more value that side unbalance

How to create BBST:-		
When root NULL		
Input = 15		17 height = 2 15 height =1
Input = 12	 	12 height = 1 15 height = 2 17 height = 3 || (0-2)=2 unbalance

Input = 18		**18 height = 1** **17 height = 2** **15 height = 3 ||(2-1)=1** **12 height = 1** **Balance**
Input = 20	 	**20 height = 1** **18 height = 2** **17 height = 3 || (2-0)=2** **Unbalance** **15 height = 3 || (2-1)=1** **Balance**

Step create BBST:-
1. **First enter the element in our Tree.** 2. **Left side —-> Right** 3. **Till it find NULL : Create itself & return** 4. **Check the balancing : - first update the right; -Balance =left -right** **Balance > 1 = unbalance left side : LL & RL** **Balance <-1 = unbalance right side : RR & RL**

Balancing the Binary Search Tree:-

```
#include<iostream>
using namespace std;

class Node
{
    public:
    int data, height;
    Node *left, *right;

    Node(int value)
    {
        data = value;
        height =1;
        left = right =NULL;
    }
};

int getHeight(Node *root)
{
    if(!root)
    return 0;

    return root ->height;
};
```

```
void updataHeight(Node *root)
{
    int leftHeight = getHeight(root ->left);
    int rightHeight = getHeight(root ->right);
    root ->height = 1+ max(leftHeight, rightHeight);
};

Node *rotateRight(Node *root)
{
    Node *temp = root ->left;
    root ->left = temp ->right;
    temp ->right = root;

    updataHeight(root);
    updataHeight(temp);

    return temp;
};

Node *rotateLeft(Node *root)
{
    Node *temp = root ->right;
    root ->right = temp ->left;
    temp ->left = root;

    updataHeight(root);
    updataHeight(temp);

    return temp;
};

Node *Balance(Node *root)
{
    if(!root)
    return NULL;

    // Update the height
    updataHeight(root);

    // Balance factor = left - right height
    int balance = getHeight(root ->left) - getHeight(root ->right);

    // Balance > 1 Left Unbalance
    if(balance>1)
    {
        // left left unbalance
        if(getHeight(root ->left -> left) >= getHeight(root ->left ->right))
        {
            root = rotateRight(root);
        }
        //Left Right Unbalance
        else
```

```
        {
            root ->left = rotateLeft(root ->left);
            root = rotateRight(root);
        }
    }
    // Balance < -1 Right Unbalance
    else if(balance< -1)
    {
        // Right Right unbalance
        if(getHeight(root ->right -> right) >= getHeight(root ->right ->left))
        {
            root = rotateLeft(root);
        }
        // Right Left Unbalance
        else
        {
            root ->right = rotateRight(root ->right);
            root = rotateLeft(root);
        }
    }
        return root;

};

Node *insertBST(Node *root, int value)
{
    if(!root)
    {
        return new Node(value);
    };
    if(value <root ->data)
    {
        root ->left = insertBST(root ->left, value);
    }
    else
    {
        root ->right = insertBST(root ->right, value);
    }
    return Balance(root);
};

void inorder(Node *root)
{
    if(!root)
    return;

    inorder(root ->left);
    cout<<root ->data<<" ";
    inorder(root ->right);
};

void preorder(Node *root)
```

```
{
    if(!root)
    return;

    cout<<root ->data<<" ";
    preorder(root ->left);
    preorder(root ->right);
};

int main()
{
    Node *root = NULL;
    int value;
    while(1)
    {
        cin>>value;
        if(value != -1)
        root = insertBST(root, value);
        else
        break;
    }

    inorder(root);
    cout<<endl;
    preorder(root);

return 0;
};
```

Time Complexity :-	
BBST: Balance Binary Search Tree	**BST: Binary Search Tree**
Insertion = O(logN) **Deletion = O(logN)**	**Insertion = O(N)** **Deletion = O(N)**