A purple text on a black background

AI-generated content may be incorrect.

**LAB 12 - BST and AVL**

Name: Omar Afef Mohammed

Roll number: SU92-BSSEM-S24-086

Section: 3A

Teacher: Mr. Rasikh Ali

Subject: DSA LAB

Date: May 4, 2025

Tasks:

1. Insert and Traverse for BST

**ANSWER**

**CODE**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* right;

Node\* left;

Node(int d){

data=d;

right=NULL;

left=NULL;

}

};

class BST{

public:

Node\* head;

BST(){

head=NULL;

}

void insert(int d){

if(head==NULL){

head=new Node(d);

return;

}

Node\* temp=head;

while(true){

if(d>temp->data){

if(temp->right==NULL){

temp->right=new Node(d);

return;

}

temp=temp->right;

}

else if(d<temp->data){

if(temp->left==NULL){

temp->left=new Node(d);

return;

}

temp=temp->left;

}

else{

cout<<temp->data<<"="<<d<<endl;

return;

}

}

}

void display(Node\* temp){

cout<<temp->data<<" ";

}

void Travers() {

TraversHelper(head);

}

void TraversHelper(Node\* temp) {

if (temp == NULL) return;

display(temp);

TraversHelper(temp->left);

TraversHelper(temp->right);

}

};

int main(){

BST me;

me.insert(1);

me.insert(2);

me.insert(3);

me.insert(4);

me.insert(4);

me.insert(5);

me.insert(6);

me.insert(6);

me.Travers();

}

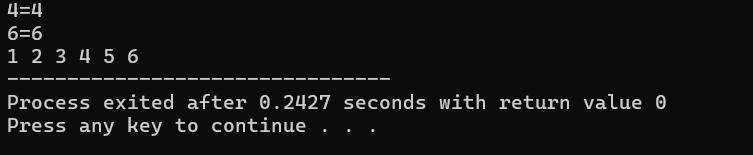
**How it works:**

1. Node class:  
   Each node stores:
   * data (an integer)
   * left (left child)
   * right (right child)
2. BST class:
   * Contains a pointer head to the root node.
   * insert(d):
     + If tree is empty, set head as new node.
     + Else, go left if d < current, go right if d > current.
     + If equal, do nothing (or print that it’s a duplicate).
   * Travers():
     + Calls helper to print tree in preorder (root, left, right).
3. main() function:
   * Inserts values 1–6 (some duplicates).
   * Then calls Travers() to print them.

**Why it works:**

1. Binary Search Tree rule:  
   Left child < Node < Right child — this is maintained in insert().
2. No duplicates:  
   When d == current->data, it returns — avoiding duplicates.
3. Traversal:  
   Recursive preorder prints nodes in a consistent pattern.
4. Dynamic structure:  
   Uses pointers (linked nodes), so the tree grows as needed — no size limit.

**OUTPUT**

****

2. Insert and Traverse for AVL

**ANSWER**

**CODE**

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node\* right;

Node\* left;

Node(int d){

data=d;

right=NULL;

left=NULL;

}

};

class BST{

public:

Node\* head;

BST(){

head=NULL;

}

void insert(int d){

if(head==NULL){

head=new Node(d);

return;

}

Node\* temp=head;

while(true){

if(d>temp->data){

if(temp->right==NULL){

temp->right=new Node(d);

return;

}

temp=temp->right;

}

else if(d<temp->data){

if(temp->left==NULL){

temp->left=new Node(d);

return;

}

temp=temp->left;

}

else{

cout<<temp->data<<"="<<d<<endl;

return;

}

}

}

void display(){

printTree(head);

cout<<"\n\nend here\n\n";

}

void printTree(Node\* root, int space = 0, int indent = 5) {

if (root == NULL) return;

space += indent;

// Print right subtree first

printTree(root->right, space);

// Print current node

cout << endl;

for (int i = indent; i < space; i++) cout << " ";

cout << root->data << "\n";

// Print left subtree

printTree(root->left, space);

}

// Recursive function to calculate height of a subtree

int getHeight(Node\* root) {

if (root == NULL) return 0;

return 1 + max(getHeight(root->left), getHeight(root->right));

}

// Function to check if a node is AVL-balanced

int isAVL(Node\* node) {

if (node == NULL) return 0;

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

int balance = leftHeight - rightHeight;

return balance;

}

// Right Rotation

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

return x; // New root

}

// Left Rotation

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

return y; // New root

}

Node\* fixAVL(Node\* node) {

if (node == NULL) return NULL;

int balance = isAVL(node);

// Left heavy

if (balance > 1) {

if (isAVL(node->left) < 0) {

// Left-Right Case

node->left = rotateLeft(node->left);

}

// Left-Left Case

return rotateRight(node);

}

// Right heavy

if (balance < -1) {

if (isAVL(node->right) > 0) {

// Right-Left Case

node->right = rotateRight(node->right);

}

// Right-Right Case

return rotateLeft(node);

}

return node; // Already balanced

}

Node\* fixWholeTree(Node\* node) {

if (node == NULL) return NULL;

// Fix left and right subtrees first

node->left = fixWholeTree(node->left);

node->right = fixWholeTree(node->right);

// Then fix this node

return fixAVL(node);

}

};

int main(){

BST me;

me.insert(1);

me.insert(2);

me.insert(3);

me.insert(4);

me.insert(5);

me.insert(6);

me.insert(0);

me.insert(-1);

me.insert(-2);

me.display();

me.head=me.fixWholeTree(me.head);

me.display();

}

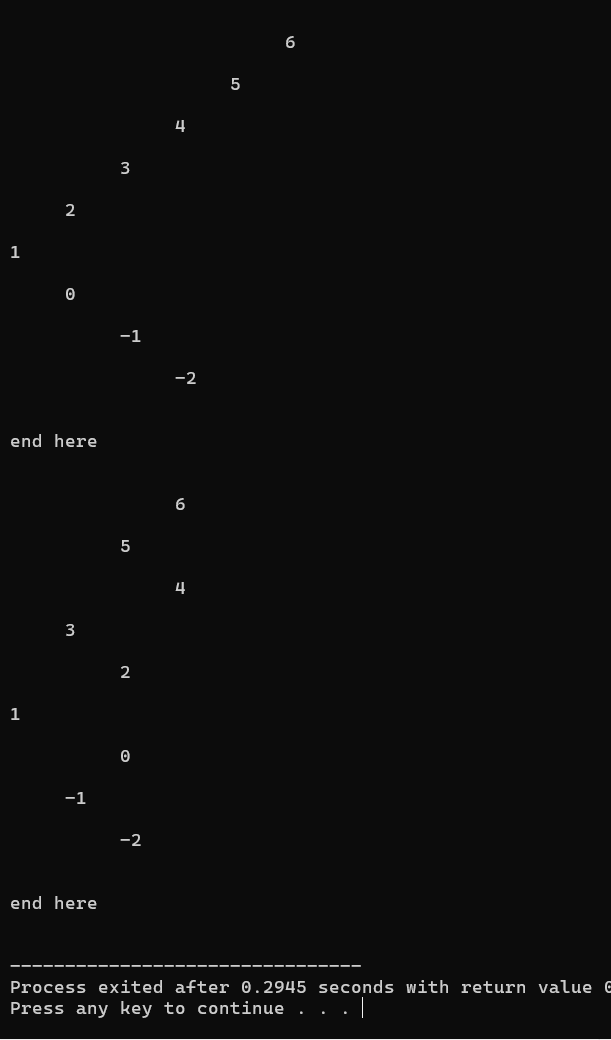
**How it works (step by step):**

1. Node class:  
   Same as before — stores data, left, and right.
2. insert(d):  
   Same BST insert — no balancing during insert.
3. display() and printTree():  
   Shows the tree sideways:
   * Right child on top
   * Left child at bottom  
     (uses indentation to show levels)
4. getHeight():  
   Recursively finds the height of a subtree.
5. isAVL():  
   Checks balance factor: left height - right height.
   * AVL allows only -1, 0, or +1
   * If outside this, it's unbalanced
6. Rotations:
   * rotateLeft(): Fix right-heavy imbalance
   * rotateRight(): Fix left-heavy imbalance
7. fixAVL():  
   Fixes one node if it's unbalanced using appropriate rotation(s).
8. fixWholeTree():  
   Fixes the entire tree recursively:
   * First fixes subtrees (bottom-up)
   * Then fixes current node
9. main():
   * Inserts 9 values (some make it unbalanced)
   * Displays tree (unbalanced)
   * Fixes AVL balance
   * Displays tree again (balanced)

**Why it works:**

1. Binary Search Tree rules:  
   Insertion keeps correct left < root < right order.
2. AVL Tree rules:  
   After insertion, fixWholeTree() ensures every node has a balance factor between -1 and +1.
3. Rotations fix imbalance:
   * LL → Right Rotation
   * RR → Left Rotation
   * LR → Left + Right Rotation
   * RL → Right + Left Rotation
4. Display function helps visualize the tree easily.

**OUTPUT**

****