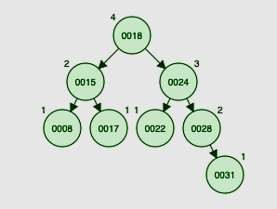
15B17CI371–Data Structures Lab ODD 2024

**Week 8-LAB B Practice Lab**

**Q1. Delete the following values form the AVL tree in following fig.**

**1)24**

**2)15**

**3)18**

**4)22**

**5)17**

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* left;

Node\* right;

int height;

};

int getHeight(Node\* node)

{

return node==nullptr?0:node->height;

}

Node\* createNode(int data)

{

Node\* newNode=new Node();

newNode->data=data;

newNode->left=nullptr;

newNode->right=nullptr;

newNode->height=1;

return newNode;

}

int getBalance(Node\* node)

{

if(node==nullptr)

return 0;

return getHeight(node->left)-getHeight(node->right);

}

Node\* rightRotate(Node\* y)

{

Node\* x=y->left;

Node\* T2=x->right;

x->right=y;

y->left=T2;

y->height=max(getHeight(y->left),getHeight(y->right))+1;

x->height=max(getHeight(x->left),getHeight(x->right))+1;

return x;

}

Node\* leftRotate(Node\* x)

{

Node\* y=x->right;

Node\* T2=y->left;

y->left=x;

x->right=T2;

x->height=max(getHeight(x->left),getHeight(x->right))+1;

y->height=max(getHeight(y->left),getHeight(y->right))+1;

return y;

}

Node\* insertNode(Node\* node,int data)

{

if(node==nullptr)

return createNode(data);

if(data<node->data)

node->left=insertNode(node->left,data);

else if(data>node->data)

node->right=insertNode(node->right,data);

else

return node;

node->height=1+max(getHeight(node->left),getHeight(node->right));

int balance=getBalance(node);

if(balance>1&&data<node->left->data)

return rightRotate(node);

if(balance<-1&&data>node->right->data)

return leftRotate(node);

if(balance>1&&data>node->left->data)

{

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

return rightRotate(node);

}

if(balance<-1&&data<node->right->data)

{

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

return leftRotate(node);

}

return node;

}

Node\* minValueNode(Node\* node)

{

Node\* current=node;

while(current->left!=nullptr)

current=current->left;

return current;

}

Node\* deleteNode(Node\* root,int data)

{

if(root==nullptr)

return root;

if(data<root->data)

root->left=deleteNode(root->left,data);

else if(data>root->data)

root->right=deleteNode(root->right,data);

else

{

if((root->left==nullptr)||(root->right==nullptr))

{

Node\* temp=root->left?root->left : root->right;

if(temp==nullptr)

{

temp=root;

root=nullptr;

}

else

\*root=\*temp;

delete temp;

}

else

{

Node\* temp=minValueNode(root->right);

root->data=temp->data;

root->right=deleteNode(root->right,temp->data);

}

}

if(root==nullptr)

return root;

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

int balance=getBalance(root);

if(balance>1&&getBalance(root->left)>=0)

return rightRotate(root);

if(balance>1&&getBalance(root->left)<0)

{

root->left=leftRotate(root->left);

return rightRotate(root);

}

if(balance<-1&&getBalance(root->right)<=0)

return leftRotate(root);

if(balance<-1&&getBalance(root->right)>0)

{

root->right=rightRotate(root->right);

return leftRotate(root);

}

return root;

}

void inorder(Node\* root)

{

if(root==nullptr)

return;

inorder(root->left);

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

inorder(root->right);

}

int main()

{

Node\* root=nullptr;

int elements[]={18,15,24,8,17,22,28,31};

int n=sizeof(elements)/sizeof(elements[0]);

for(int i=0;i<n;i++)

root=insertNode(root,elements[i]);

cout<<"Inorder traversal after insertion :\n";

inorder(root);

cout<<endl;

root=deleteNode(root,24);

cout<<"\nInorder traversal after deleting 24 :\n";

inorder(root);

root=deleteNode(root,15);

cout<<"\nInorder traversal after deleting 15 :\n";

inorder(root);

root=deleteNode(root,18);

cout<<"\nInorder traversal after deleting 18 :\n";

inorder(root);

root=deleteNode(root,22);

cout<<"\nInorder traversal after deleting 22 :\n";

inorder(root);

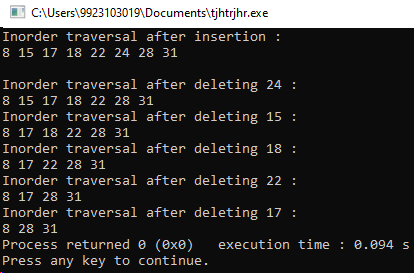
root=deleteNode(root,17);

cout<<"\nInorder traversal after deleting 17 :\n";

inorder(root);

}

**Output :**

****

**Q2. AVL Tree is one of the height-balanced binary search trees. Here,balancing of the BST is achieved using an additional field of balance factor associated with each node. Balance factor of a node,N is computed as the difference between the height of the left branch of N and the height of the right branch of N. In AVL tree,the balance factor of each node must be in the range between -1 and +1(*i.e.*-1,0,and1).**

**In context of AVL tree,write programs for following:**

* 1. **Compute the height of a binary tree/ binary search tree rooted at a node,N**

#include<iostream>

using namespace std;

struct Node

{

int data;

Node \*left,\*right;

Node(int value)

{

data=value;

left=right=nullptr;

}

};

Node\* insert(Node\* node,int data)

{

if(node==nullptr)

return new Node(data);

if(data<node->data)

node->left=insert(node->left,data);

else

node->right=insert(node->right,data);

return node;

}

void inOrderTraversal(Node\* root)

{

if(root!=nullptr)

{

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

int height(Node\* node)

{

if(node==nullptr)

return 0;

int leftHeight=height(node->left);

int rightHeight=height(node->right);

return max(leftHeight,rightHeight)+1;

}

int main()

{

int n,value;

cout<<"Input the number of elements : ";

cin>>n;

Node\* N=nullptr;

cout<<"Input the node values : ";

for(int i=0;i<n;i++)

{

cin>>value;

N=insert(N,value);

}

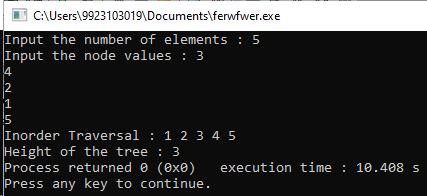
cout<<"Inorder Traversal : ";

inOrderTraversal(N);

cout<<"\nHeight of the tree : "<<height(N);

}

**Output :**

****

* 1. **Pre-order traversal of a binary search tree is given to you. Write a program to check whether the binary search tree formed with the given pre-order traversal is a valid AVL tree or not. E.g.,if pre-order traversal of a binary search tree is 20,10,15,18,30,25,and 40,then your program should display that it is not a valid AVL tree,whereas,your program should display display a valid AVL tree if the pre-order traversal is given as 20,15,18,30,25,and 40.**

#include<iostream>

#include<climits>

#include <cmath>

using namespace std;

struct Node

{

int data;

Node \*left,\*right;

Node(int value)

{

data=value;

left=right=nullptr;

}

};

Node\* constructBST(int pre[],int\* preIndex,int key,int min,int max,int n)

{

if(\*preIndex>=n)

return nullptr;

Node\* root=nullptr;

if(key>min&&key<max)

{

root=new Node(key);

\*preIndex=\*preIndex+1;

if(\*preIndex<n)

root->left=constructBST(pre,preIndex,pre[\*preIndex],min,key,n);

if(\*preIndex<n)

root->right=constructBST(pre,preIndex,pre[\*preIndex],key,max,n);

}

return root;

}

int height(Node\* node)

{

if(node==nullptr)

return 0;

int leftHeight=height(node->left);

int rightHeight=height(node->right);

return max(leftHeight,rightHeight)+1;

}

bool isBalanced(Node\* root)

{

if(root==nullptr)

return true;

int leftHeight=height(root->left);

int rightHeight=height(root->right);

if(abs(leftHeight-rightHeight)>1)

return false;

return isBalanced(root->left)&&isBalanced(root->right);

}

bool isAVL(int pre[],int n)

{

int preIndex=0;

Node\* root=constructBST(pre,&preIndex,pre[0],INT\_MIN,INT\_MAX,n);

return isBalanced(root);

}

int main()

{

int n;

cout<<"Input the number of elements : ";

cin>>n;

int pre[n];

cout<<"Input the preorder traversal of the tree : ";

for(int i=0;i<n;i++)

cin>>pre[i];

if(isAVL(pre,n))

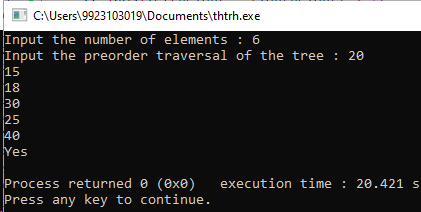
cout<<"Yes";

else

cout<<"No";

}

**Output :**

****

* 1. **You have been given two AVL trees A and B of height M and N respectively. Write a program to merge the AVL trees A and B into a new AVL tree C.**

#include<iostream>

#include<vector>

using namespace std;

struct Node

{

int data;

Node \*left,\*right;

Node(int value)

{

data=value;

left=right=nullptr;

}

};

Node\* insert(Node\* node,int data)

{

if(node==nullptr)

return new Node(data);

if(data<node->data)

node->left=insert(node->left,data);

else

node->right=insert(node->right,data);

return node;

}

void inOrder(Node\* root,vector<int>& nodes)

{

if(root==nullptr)

return;

inOrder(root->left,nodes);

nodes.push\_back(root->data);

inOrder(root->right,nodes);

}

Node\* sortedArrayToAVL(vector<int>& arr,int start,int end)

{

if(start>end)

return nullptr;

int mid=(start+end) / 2;

Node\* root=new Node(arr[mid]);

root->left=sortedArrayToAVL(arr,start,mid-1);

root->right=sortedArrayToAVL(arr,mid+1,end);

return root;

}

Node\* mergeTrees(Node\* A,Node\* B)

{

vector<int> nodesA,nodesB,mergedNodes;

inOrder(A,nodesA);

inOrder(B,nodesB);

int i=0,j=0;

while(i<nodesA.size()&&j<nodesB.size())

{

if(nodesA[i]<nodesB[j])

mergedNodes.push\_back(nodesA[i++]);

else

mergedNodes.push\_back(nodesB[j++]);

}

while(i<nodesA.size())

mergedNodes.push\_back(nodesA[i++]);

while(j<nodesB.size())

mergedNodes.push\_back(nodesB[j++]);

return sortedArrayToAVL(mergedNodes,0,mergedNodes.size()-1);

}

int main()

{

int n1,n2,value;

cout<<"Input the number of elements in tree A : ";

cin>>n1;

cout<<"Input the pre order traversal of tree A : ";

Node\* A=nullptr;

for(int i=0;i<n1;i++)

{

cin>>value;

A=insert(A,value);

}

cout<<"Input the number of elements in tree B : ";

cin>>n2;

cout<<"Input the pre order traversal of tree B : ";

Node\* B=nullptr;

for(int i=0;i<n2;i++)

{

cin>>value;

B=insert(B,value);

}

Node\* C=mergeTrees(A,B);

vector<int> mergedInOrder;

inOrder(C,mergedInOrder);

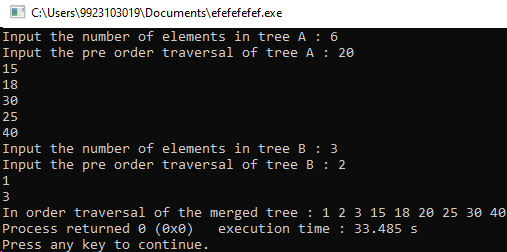
cout<<"In order traversal of the merged tree : ";

for(int val : mergedInOrder)

cout<<val<<" ";

}

**Output :**

****

* 1. **It is desired to delete entire sub-tree rooted at an intermediate node N in an AVL tree,A. It can be done in single step by making left and right children of N as NULL. so,Find node with value N and make left and right children of N as NULL. Call this modified tree as A’. Certainly,A’ will not be a valid AVL tree. If possible,apply known AVL rotations or your own designed rotations on A’ so that it will become a valid AVL tree.**

#include<iostream>

#include<algorithm>

using namespace std;

struct Node

{

int data;

Node \*left,\*right;

int height;

Node(int value)

{

data=value;

left=right=nullptr;

height=1;

}

};

int getHeight(Node\* node)

{

if(node==nullptr)

return 0;

return node->height;

}

int getBalance(Node\* node)

{

if(node==nullptr)

return 0;

return getHeight(node->left)-getHeight(node->right);

}

Node\* rightRotate(Node\* y)

{

Node\* x=y->left;

Node\* T2=x->right;

x->right=y;

y->left=T2;

y->height=max(getHeight(y->left),getHeight(y->right))+1;

x->height=max(getHeight(x->left),getHeight(x->right))+1;

return x;

}

Node\* leftRotate(Node\* x)

{

Node\* y=x->right;

Node\* T2=y->left;

y->left=x;

x->right=T2;

x->height=max(getHeight(x->left),getHeight(x->right))+1;

y->height=max(getHeight(y->left),getHeight(y->right))+1;

return y;

}

Node\* insert(Node\* node,int data)

{

if(node==nullptr)

return new Node(data);

if(data<node->data)

node->left=insert(node->left,data);

else if(data>node->data)

node->right=insert(node->right,data);

else

return node;

node->height=1+max(getHeight(node->left),getHeight(node->right));

int balance=getBalance(node);

if(balance>1&&data<node->left->data)

return rightRotate(node);

if(balance<-1&&data>node->right->data)

return leftRotate(node);

if(balance>1&&data>node->left->data)

{

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

return rightRotate(node);

}

if(balance<-1&&data<node->right->data)

{

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

return leftRotate(node);

}

return node;

}

Node\* findNode(Node\* root,int N)

{

if(root==nullptr||root->data==N)

return root;

if(N<root->data)

return findNode(root->left,N);

return findNode(root->right,N);

}

Node\* deleteSubtree(Node\* root,int N)

{

Node\* node=findNode(root,N);

if(node != nullptr)

{

node->left=nullptr;

node->right=nullptr;

}

return root;

}

Node\* balanceAVL(Node\* node)

{

if(node==nullptr)

return nullptr;

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

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

node->height=1+max(getHeight(node->left),getHeight(node->right));

int balance=getBalance(node);

if(balance>1&&getBalance(node->left)>=0)

return rightRotate(node);

if(balance>1&&getBalance(node->left)<0)

{

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

return rightRotate(node);

}

if(balance<-1&&getBalance(node->right) <= 0)

return leftRotate(node);

if(balance<-1&&getBalance(node->right)>0)

{

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

return leftRotate(node);

}

return node;

}

void showChildren(Node\* root)

{

if(root==nullptr)

return;

cout<<"Parent: "<<root->data;

if(root->left)

cout<<",Left Child: "<<root->left->data;

else

cout<<",Left Child: NULL";

if(root->right)

cout<<",Right Child: "<<root->right->data;

else

cout<<",Right Child: NULL";

cout<<endl;

showChildren(root->left);

showChildren(root->right);

}

void inOrder(Node\* root)

{

if(root==nullptr)

return;

inOrder(root->left);

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

inOrder(root->right);

}

int main()

{

int n,N,value;

cout<<"Input the number of elements : ";

cin>>n;

Node\* A=nullptr;

cout<<"Input the node values : ";

for(int i=0;i<n;i++)

{

cin>>value;

A=insert(A,value);

}

cout<<"\nShowing the children of each parent node:\n";

showChildren(A);

cout<<"Input the value of Node N : ";

cin>>N;

A=deleteSubtree(A,N);

A=balanceAVL(A);

cout<<"In order Traversal of modified AVL tree :\n";

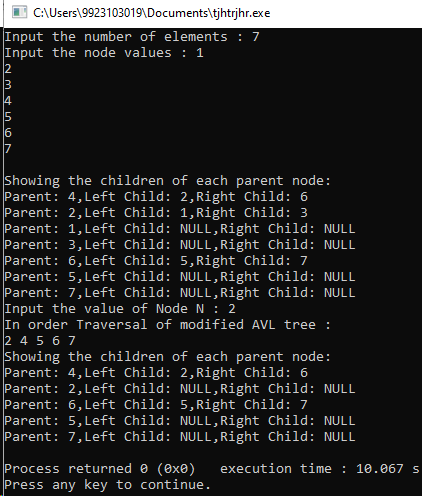
inOrder(A);

cout<<"\nShowing the children of each parent node:\n";

showChildren(A);

}

**Output :**

****

* 1. **Let us consider,E as an element stored in a node N of an AVL tree,A. Considering an update operation,which updated the element E by ±Δ. With updated value as E±Δ at node N,A may not be a valid AVL tree. Call the updated tree as A`. Write a program to make the tree A` as a valid AVL tree.**

#include<iostream>

#include<algorithm>

using namespace std;

struct Node

{

int data;

Node \*left,\*right;

int height;

Node(int value)

{

data=value;

left=right=nullptr;

height=1;

}

};

int getHeight(Node\* node)

{

if(node==nullptr)

return 0;

return node->height;

}

int getBalance(Node\* node)

{

if(node==nullptr)

return 0;

return getHeight(node->left)-getHeight(node->right);

}

Node\* rightRotate(Node\* y)

{

Node\* x=y->left;

Node\* T2=x->right;

x->right=y;

y->left=T2;

y->height=max(getHeight(y->left),getHeight(y->right))+1;

x->height=max(getHeight(x->left),getHeight(x->right))+1;

return x;

}

Node\* leftRotate(Node\* x)

{

Node\* y=x->right;

Node\* T2=y->left;

y->left=x;

x->right=T2;

x->height=max(getHeight(x->left),getHeight(x->right))+1;

y->height=max(getHeight(y->left),getHeight(y->right))+1;

return y;

}

Node\* insert(Node\* node,int data)

{

if(node==nullptr)

return new Node(data);

if(data<node->data)

node->left=insert(node->left,data);

else if(data>node->data)

node->right=insert(node->right,data);

else

return node;

node->height=1+max(getHeight(node->left),getHeight(node->right));

int balance=getBalance(node);

if(balance>1&&data<node->left->data)

return rightRotate(node);

if(balance<-1&&data>node->right->data)

return leftRotate(node);

if(balance>1&&data>node->left->data)

{

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

return rightRotate(node);

}

if(balance<-1&&data<node->right->data)

{

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

return leftRotate(node);

}

return node;

}

Node\* findNode(Node\* root,int N)

{

if(root==nullptr||root->data==N)

return root;

if(N<root->data)

return findNode(root->left,N);

return findNode(root->right,N);

}

Node\* updateNode(Node\* root,int N,int delta)

{

Node\* node=findNode(root,N);

if(node != nullptr)

{

node->data+=delta;

}

return root;

}

Node\* balanceAVL(Node\* node)

{

if(node==nullptr)

return nullptr;

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

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

node->height=1+max(getHeight(node->left),getHeight(node->right));

int balance=getBalance(node);

if(balance>1&&getBalance(node->left)>=0)

return rightRotate(node);

if(balance>1&&getBalance(node->left)<0)

{

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

return rightRotate(node);

}

if(balance<-1&&getBalance(node->right) <= 0)

return leftRotate(node);

if(balance<-1&&getBalance(node->right)>0)

{

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

return leftRotate(node);

}

return node;

}

void inOrder(Node\* root)

{

if(root==nullptr)

return;

inOrder(root->left);

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

inOrder(root->right);

}

int main()

{

int n,N,value,delta;

cout<<"Input the number of elements : ";

cin>>n;

Node\* A=nullptr;

cout<<"Input the node values : ";

for(int i=0;i<n;i++)

{

cin>>value;

A=insert(A,value);

}

cout<<"Input the value of Node N : ";

cin>>N;

cout<<"Input the value of delta to be added to the value of node N : ";

cin>>delta;

A=updateNode(A,N,delta);

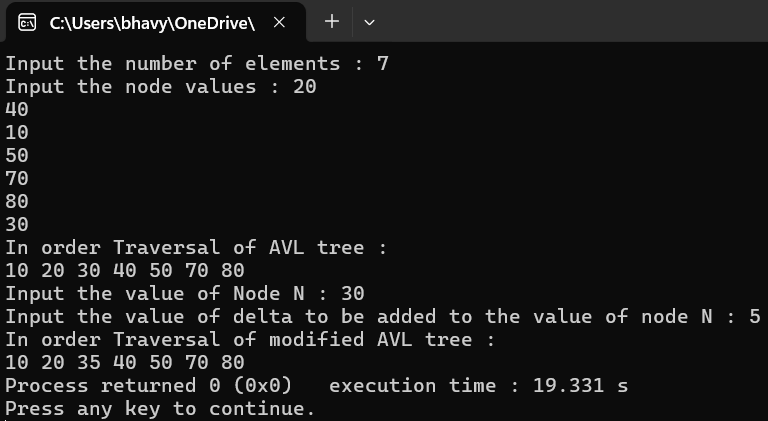
A=balanceAVL(A);

cout<<"In order Traversal of modified AVL tree :\n";

inOrder(A);

}

**Output :**

****