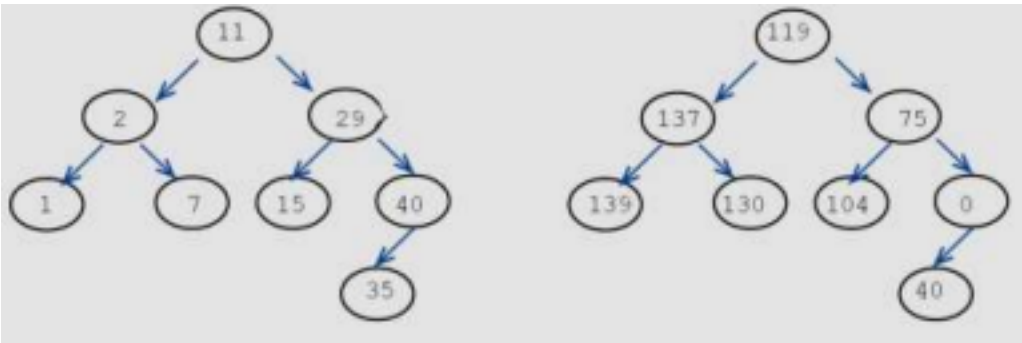
15B17CI371 – Data Structures Lab
ODD 2024
Week 8-LAB A
Practice Lab
[CO: C270.4]

Q1. Given a BST,transform it into greater sum tree where each node contains sum of all nodes greater than that node.




```
#include <iostream>
using namespace std;
struct Node
{
   int data;
   Node* left;
   Node* right;
};
Node* createNode(int data)
{
   Node* newNode=new Node();
   newNode->data=data;
   newNode->left=nullptr;
   newNode->right=nullptr;
   return newNode;
}
Node* insertNode(Node* root,int data)
{
   if(root==nullptr)
      return createNode(data);
   if(data<root->data)
      root->left=insertNode(root->left,data);
   else
      root->right=insertNode(root->right,data);
   return root;
}
void transformBST(Node* root,int& sum)
{
   if(root==nullptr)
      return;
   transformBST(root->right,sum);
   sum += root->data;
   root->data=sum-root->data;
   transformBST(root->left,sum);
}
void inorder(Node* root)
{
   if(root==nullptr)
```

```
    return;
  inorder(root->left);
  cout<<root->data<<" ";
  inorder(root->right);
}
int main()
{
  int n,data;
  Node* root=nullptr;
  cout<<"Input the number of nodes : ";
  cin>>n;
  cout<<"Input the nodes values : \n";
  for(int i=0;i<n;i++)
  {
    cin>>data;
    root=insertNode(root,data);
  }
  int sum=0;
  transformBST(root,sum);
  cout<<"Transformed BST(Greater Sum Tree) inorder traversal:\n";
  inorder(root);
}
```

Output :

```
Input the number of nodes : 8
Input the nodes values :
11
2
29
1
7
15
40
35
Transformed BST (Greater Sum Tree) inorder traversal:
139 137 130 119 104 75 40 0
Process returned 0 (0x0)   execution time : 26.207 s
Press any key to continue.
```

Q2. WAP to find the k^{th} smallest element in a BST.

```
#include <iostream>
using namespace std;
struct Node
{
  int data;
  Node* left;
  Node* right;
};
Node* createNode(int data)
{
  Node* newNode=new Node();
  newNode->data=data;
  newNode->left=nullptr;
  newNode->right=nullptr;
  return newNode;
```

```
}
Node* insertNode(Node* root,int data)
{
   if(root==nullptr)
      return createNode(data);
   if(data<root->data)
      root->left=insertNode(root->left,data);
   else
      root->right=insertNode(root->right,data);
   return root;
}
void findKthSmallest(Node* root,int& k,int& result)
{
   if(root==nullptr||k==0)
      return;
   findKthSmallest(root->left,k,result);
   k--;
   if(k==0)
   {
      result=root->data;
      return;
   }
   findKthSmallest(root->right,k,result);
}
int main()
{
   int n,data,k,result=-1;
   Node* root=nullptr;
   cout<<"Input the number of nodes : ";
   cin>>n;
   cout<<"Input the nodes in any order :\n";
   for(int i=0;i<n;i++)
   {
      cin>>data;
      root=insertNode(root,data);
   }
   cout<<"Input the value of k: ";
   cin>>k;
   int temp=k;
   findKthSmallest(root,temp,result);
   if(result!=-1)
      cout<<"The "<<k<<"th smallest element is: "<<result;
   else
      cout<<"The value of k is out of range.";
}
```

Output :

```
Input the number of nodes : 8
Input the nodes in any order :
4
8
0
1
3
9
2
5
Input the value of k : 2
The 2th smallest element is : 1
Process returned 0 (0x0)   execution time : 11.486 s
Press any key to continue.
```

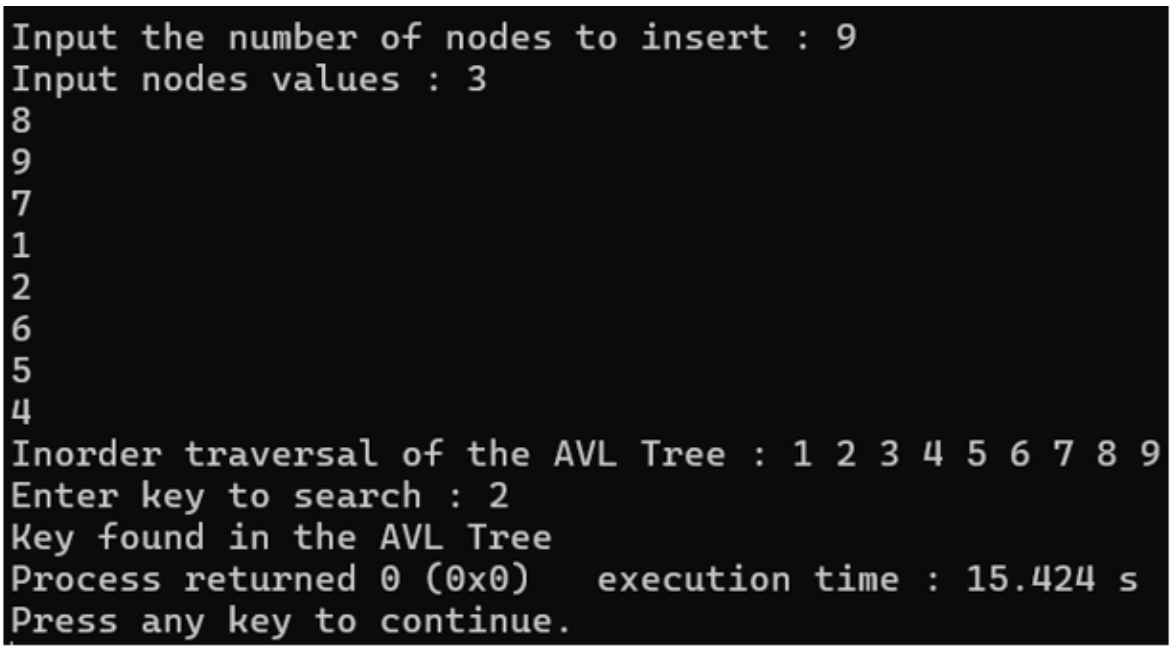
Q3. Implement an AVL Tree that supports insertion and search operations efficiently while maintaining its balanced property.

```
#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;
}
bool search(Node* root,int key)
{
   if(root==nullptr)
      return false;
   if(root->data==key)
      return true;
   else if(key<root->data)
      return search(root->left,key);
   else
      return search(root->right,key);
}
void inorder(Node* root)
{
   if(root==nullptr)
      return;
   inorder(root->left);
```

```
    cout<<root->data<<" ";
    inorder(root->right);
}
int main()
{
    Node* root=nullptr;
    int n,data,key;
    cout<<"Input the number of nodes to insert : ";
    cin>>n;
    cout<<"Input nodes values : ";
    for(int i=0;i<n;i++)
    {
        cin>>data;
        root=insertNode(root,data);
    }
    cout<<"Inorder traversal of the AVL Tree : ";
    inorder(root);
    cout<<"\nEnter key to search : ";
    cin>>key;
    if(search(root,key))
        cout<<"Key found in the AVL Tree";
    else
        cout<<"Key not found in the AVL Tree";
}
```

Output :



Q4. WAP to implement AVL tree for the following elements:

a. 21,26,30,9,4,14,28,18,15,10,2,3,7

b. Also,WAP to delete 30,14,10 nodes.

```
#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[]={21,26,30,9,4,14,28,18,15,10,2,3,7};
    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,30);
    root=deleteNode(root,14);
    root=deleteNode(root,10);
    cout<<"Inorder traversal after deleting 30,14,10 :\n";
    inorder(root);
}
```

Output :

```
Inorder traversal after insertion :
2 3 4 7 9 10 14 15 18 21 26 28 30
Inorder traversal after deleting 30, 14, 10 :
2 3 4 7 9 15 18 21 26 28
Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
```

Q5. Develop functionality within an AVL Tree to handle deletions efficiently while preserving the tree's balanced nature. Also show the left and right children of each parent node.

Input 1:
AVL Insert: 40,20,60,10,30,50,70,5,15,25,35,45,55,65,75 AVL Delete: 5
Output 1:
10,15,20,25,30,35,40,45,50,55,60,65,70,75

Input 2:
AVL Insert: 10,15,20,25,35,40,45,50,55,60,65,70,75
AVL Delete: 60
Output 2:
10,15,20,25,35,40,45,50,55,65,70,75

```
#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)
{
   return node==nullptr?0: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);
}
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);
}
int main()
{
    int n,data;
    Node* root=nullptr;
    cout<<"Input the number of nodes : ";
    cin>>n;
    cout<<"Input the nodes values :\n";
```

```
    for(int i=0;i<n;i++)
    {
        cin>>data;
        root=insertNode(root,data);
    }
    cout<<"Inorder traversal after insertion:\n";
    inorder(root);
    cout<<"\nInput the node value to be deleted : ";
    cin>>data;
    root=deleteNode(root,data);
    cout<<"Inorder traversal after deleting 5:\n";
    inorder(root);
    cout<<"\nShowing the children of each parent node:\n";
    showChildren(root);
}
```

Output :

```
Input the number of nodes : 15
Input the nodes values :
40
20
60
10
30
50
70
5
15
25
35
45
55
65
75
Inorder traversal after insertion:
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
Input the node value to be deleted : 5
Inorder traversal after deleting 5:
10 15 20 25 30 35 40 45 50 55 60 65 70 75
Showing the children of each parent node:
Parent: 40, Left Child: 20, Right Child: 60
Parent: 20, Left Child: 10, Right Child: 30
Parent: 10, Left Child: NULL, Right Child: 15
Parent: 15, Left Child: NULL, Right Child: NULL
Parent: 30, Left Child: 25, Right Child: 35
Parent: 25, Left Child: NULL, Right Child: NULL
Parent: 35, Left Child: NULL, Right Child: NULL
Parent: 60, Left Child: 50, Right Child: 70
Parent: 50, Left Child: 45, Right Child: 55
Parent: 45, Left Child: NULL, Right Child: NULL
Parent: 55, Left Child: NULL, Right Child: NULL
Parent: 70, Left Child: 65, Right Child: 75
Parent: 65, Left Child: NULL, Right Child: NULL
Parent: 75, Left Child: NULL, Right Child: NULL

Process returned 0 (0x0)   execution time : 33.331 s
Press any key to continue.
```

Q6. Problem Statement: You are given a set of key-value pairs to insert into an AVL tree. After the insertions,you need to perform a range query that partially overlaps with the existing keys. The test should validate that the tree accurately includes only the keys that fall within the specified range,even if they are close to the boundaries.

Insert: Insert a key-value pair into the tree.

Range Query: Given two keys,low and high,return the sum of values for all keys within this range(inclusive).

Insertions:

Insert(10,100)
Insert(20,200)
Insert(30,300)
Insert(40,400)
Insert(50,500)

Range Query:

Perform a range query from 25 to 45.

Expected Output:

The expected output for the range query should be the sum of values of the keys 30 and 40,which are the ones falling within the 25 to 45 range. Thus,the output should be 300+400= 700.

Insert:

10,100
20,200
30,300

Range

10 30

Output:

600

```
#include <iostream>
#include <utility>
using namespace std;
struct Node
{
   int key;
   int value;
   Node* left;
   Node* right;
   int height;
};
int getHeight(Node* node)
{
   return node==nullptr?0:node->height;
}
Node* createNode(int key,int value)
{
   Node* newNode=new Node();
   newNode->key=key;
   newNode->value=value;
   newNode->left=nullptr;
   newNode->right=nullptr;
   newNode->height=1;
   return newNode;
}
int getBalance(Node* node)
{
   return node==nullptr?0: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 key,int value)
{
    if(node==nullptr)
        return createNode(key,value);
    if(key<node->key)
        node->left=insertNode(node->left,key,value);
    else if(key>node->key)
        node->right=insertNode(node->right,key,value);
    else
        return node;
    node->height=1+max(getHeight(node->left),getHeight(node->right));
    int balance=getBalance(node);
    if(balance>1&&key<node->left->key)
        return rightRotate(node);
    if(balance<-1&&key>node->right->key)
        return leftRotate(node);
    if(balance>1&&key>node->left->key)
    {
        node->left=leftRotate(node->left);
        return rightRotate(node);
    }
    if(balance<-1&&key<node->right->key)
    {
        node->right=rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
int rangeQuery(Node* root,int low,int high)
{
    if(root==nullptr)
        return 0;
    if(root->key<low)
        return rangeQuery(root->right,low,high);
    if(root->key>high)
        return rangeQuery(root->left,low,high);
    return root->value+rangeQuery(root->left,low,high)+rangeQuery(root->right,low,high);
}
int main()
{
    Node* root=nullptr;
```

```
    int n,data,key,low,high;
    cout<<"Input the number of nodes to insert : ";
    cin>>n;
    for(int i=0;i<n;i++)
    {
        cout<<"Input the key : ";
        cin>>key;
        cout<<"Input the data : ";
        cin>>data;
        root=insertNode(root,key,data);
    }
    cout<<"Input the lower limit of range : ";
    cin>>low;
    cout<<"Input the higher limit of range : ";
    cin>>high;
    int result=rangeQuery(root,low,high);
    cout<<"Range Query Result("<<low<<" to "<<high<<") : "<<result;
}
```

Output :