15B17CI371–Data Structures

Lab ODD 2024

**Week10-LabB**

**Practice Lab**

**Q1. Write a program to A mid-sized electronic store wants to efficiently manage its**

**growing customer base. Each customer has a unique ID, and the company needs to**

**support fast insertions, searches, and deletions in their database. To achieve this,**

**they decide to use a Red-Black Tree for storing customer IDs. You are tasked with**

**implementing the Red-Black Tree to perform the following operations:**

**Insert: 125,-9,55,12, 45, 654, 78, 34, 120, 9, 67**

**Search: 654**

#include <iostream>

using namespace std;

enum Color { RED,BLACK };

struct Node

{

int data;

Node \*left,\*right,\*parent;

Color color;

Node(int data)

{

this->data=data;

left=right=parent=nullptr;

this->color=RED;

}

};

class RedBlackTree

{

Node \*root;

void rotateLeft(Node \*&node)

{

Node \*rightChild=node->right;

node->right=rightChild->left;

if(node->right != nullptr)

node->right->parent=node;

rightChild->parent=node->parent;

if(node->parent==nullptr)

root=rightChild;

else if(node==node->parent->left)

node->parent->left=rightChild;

else

node->parent->right=rightChild;

rightChild->left=node;

node->parent=rightChild;

}

void rotateRight(Node \*&node)

{

Node \*leftChild=node->left;

node->left=leftChild->right;

if(node->left != nullptr)

node->left->parent=node;

leftChild->parent=node->parent;

if(node->parent==nullptr)

root=leftChild;

else if(node==node->parent->left)

node->parent->left=leftChild;

else

node->parent->right=leftChild;

leftChild->right=node;

node->parent=leftChild;

}

void fixViolation(Node \*&node)

{

Node \*parent=nullptr;

Node \*grandparent=nullptr;

while(node != root&&node->color != BLACK&&node->parent->color==RED)

{

parent=node->parent;

grandparent=node->parent->parent;

if(parent==grandparent->left)

{

Node \*uncle=grandparent->right;

if(uncle != nullptr&&uncle->color==RED)

{

grandparent->color=RED;

parent->color=BLACK;

uncle->color=BLACK;

node=grandparent;

}

else

{

if(node==parent->right)

{

rotateLeft(parent);

node=parent;

parent=node->parent;

}

rotateRight(grandparent);

swap(parent->color,grandparent->color);

node=parent;

}

}

else

{

Node \*uncle=grandparent->left;

if(uncle != nullptr&&uncle->color==RED)

{

grandparent->color=RED;

parent->color=BLACK;

uncle->color=BLACK;

node=grandparent;

}

else

{

if(node==parent->left)

{

rotateRight(parent);

node=parent;

parent=node->parent;

}

rotateLeft(grandparent);

swap(parent->color,grandparent->color);

node=parent;

}

}

}

root->color=BLACK;

}

void inorderHelper(Node \*node)

{

if(node==nullptr)

return;

inorderHelper(node->left);

cout<<node->data <<(node->color==RED?"R " : "B ");

inorderHelper(node->right);

}

public:

RedBlackTree() { root=nullptr;}

void insert(const int &data)

{

Node \*node=new Node(data);

Node \*parent=nullptr;

Node \*current=root;

while(current != nullptr)

{

parent=current;

if(node->data<current->data)

current=current->left;

else

current=current->right;

}

node->parent=parent;

if(parent==nullptr)

root=node;

else if(node->data<parent->data)

parent->left=node;

else

parent->right=node;

if(node->parent==nullptr)

{

node->color=BLACK;

return;

}

if(node->parent->parent==nullptr)

return;

fixViolation(node);

}

void display()

{

inorderHelper(root);

cout<<endl;

}

bool search(int key)

{

return search(root,key);

}

bool search(Node\* node,int key)

{

if(node==NULL||node->data==key)

return node!=NULL;

if(key < node->data)

return search(node->left,key);

else

return search(node->right,key);

}

};

int main()

{

RedBlackTree tree;

int n;

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

cin>>n;

int elements[n];

cout<<"Input the elements : ";

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

{

cin>>elements[i];

tree.insert(elements[i]);

}

cout<<"Inorder Traversal of tree : ";

tree.display();

cout<<"\nInput a number to search : ";

int ksearch;

cin>>ksearch;

if(tree.search(ksearch))

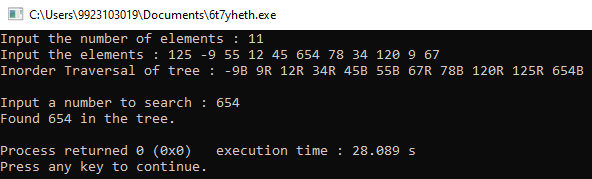
cout<<"Found "<<ksearch<<" in the tree.";

else

cout<<ksearch<<" not found in the tree.";

}

**Output :**



**Q2. Your task is to manage a large set of prices of stationary items, represented as**

**integers. The company wants to ensure that its database of property prices remains**

**balanced for quick lookups and updates. To achieve this, they decide to use an AVL**

**Tree due to its ability to maintain balance after each insertion and deletion**

**operation.**

* **Calculates the Balance Factor for each node in the AVL tree.**
* **Performs various Rotations to maintain balance in the tree.**
* **For the given item ids, create the AVL tree and calculate the number**

**of LL, LR, RR and RL rotations required**

**○ 90, 56, 23, 12, 3, 6, 1, 80, 6, 2, 61, 99, 45, 55, 22**

#include <iostream>

#include <queue>

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,int &ll,int &rr,int &lr,int &rl)

{

if(node==nullptr)

return createNode(data);

if(data<node->data)

node->left=insertNode(node->left,data,ll,rr,lr,rl);

else if(data>node->data)

node->right=insertNode(node->right,data,ll,rr,lr,rl);

else

return node;

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

int balance=getBalance(node);

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

{

rr++;

return rightRotate(node);

}

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

{

ll++;

return leftRotate(node);

}

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

{

lr++;

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

return rightRotate(node);

}

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

{

rl++;

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

return leftRotate(node);

}

return node;

}

void levelOrder(Node \*root)

{

if (!root) return;

queue<Node \*> q;

q.push(root);

while (!q.empty())

{

Node \*node = q.front();

q.pop();

cout << node->data << " ";

if (node->left) q.push(node->left);

if (node->right) q.push(node->right);

}

}

int main()

{

Node\* root=nullptr;

int ll=0,rr=0,lr=0,rl=0;;

int elements[]={90,56,23,12,3,6,1,80,6,2,61,99,45,55,22};

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

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

root=insertNode(root,elements[i],ll,rr,lr,rl);

cout<<"Level Order of the tree : ";

levelOrder(root);

cout << "\nLL rotations: " << ll << endl;

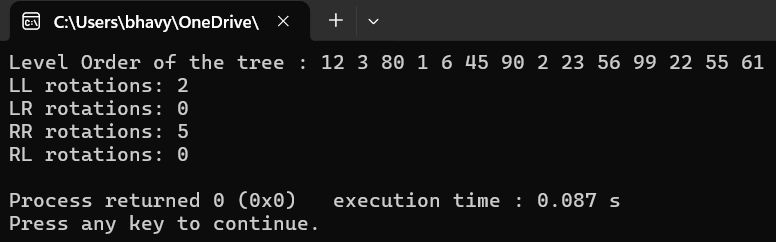
cout << "LR rotations: " << lr << endl;

cout << "RR rotations: " << rr << endl;

cout << "RL rotations: " << rl << endl;

}

**Output :**

****

**Q3. For the above-mentioned insertions, write a program to create a B-Tree of**

**(a) Max degree=3**

**(b) Max degree=3**

**Delete the elements 80, 1, 56, and 90.**

#include <iostream>

using namespace std;

class BTreeNode

{

int \*keys;

int t;

BTreeNode \*\*C;

int n;

bool leaf;

public:

BTreeNode(int \_t,bool \_leaf);

void traverse();

BTreeNode \*search(int k);

int findKey(int k);

void insertNonFull(int k);

void splitChild(int i,BTreeNode \*y);

void remove(int k);

void removeFromLeaf(int idx);

void removeFromNonLeaf(int idx);

int getPred(int idx);

int getSucc(int idx);

void fill(int idx);

void borrowFromPrev(int idx);

void borrowFromNext(int idx);

void merge(int idx);

friend class BTree;

};

class BTree

{

BTreeNode \*root;

int t;

public:

BTree(int \_t)

{

root=nullptr;

t=\_t;

}

void traverse()

{

if(root != nullptr)

root->traverse();

}

BTreeNode \*search(int k)

{

return(root==nullptr)?nullptr:root->search(k);

}

void insert(int k);

void remove(int k);

};

BTreeNode::BTreeNode(int t1,bool leaf1)

{

t=t1;

leaf=leaf1;

keys=new int[2\*t-1];

C=new BTreeNode \*[2\*t];

n=0;

}

int BTreeNode::findKey(int k)

{

int idx=0;

while(idx<n&&keys[idx]<k)

++idx;

return idx;

}

void BTreeNode::remove(int k)

{

int idx=findKey(k);

if(idx<n&&keys[idx]==k)

{

if(leaf)

removeFromLeaf(idx);

else

removeFromNonLeaf(idx);

}

else

{

if(leaf)

return;

bool flag=((idx==n)?true:false);

if(C[idx]->n<t)

fill(idx);

if(flag&&idx>n)

C[idx-1]->remove(k);

else

C[idx]->remove(k);

}

}

void BTreeNode::removeFromLeaf(int idx)

{

for(int i=idx+1;i<n;i++)

keys[i-1]=keys[i];

n--;

}

void BTreeNode::removeFromNonLeaf(int idx)

{

int k=keys[idx];

if(C[idx]->n>=t)

{

int pred=getPred(idx);

keys[idx]=pred;

C[idx]->remove(pred);

}

else if(C[idx+1]->n>=t)

{

int succ=getSucc(idx);

keys[idx]=succ;

C[idx+1]->remove(succ);

}

else

{

merge(idx);

C[idx]->remove(k);

}

}

int BTreeNode::getPred(int idx)

{

BTreeNode \*cur=C[idx];

while(!cur->leaf) cur=cur->C[cur->n];

return cur->keys[cur->n-1];

}

int BTreeNode::getSucc(int idx)

{

BTreeNode \*cur=C[idx+1];

while(!cur->leaf)

cur=cur->C[0];

return cur->keys[0];

}

void BTreeNode::fill(int idx)

{

if(idx != 0&&C[idx-1]->n>=t)

borrowFromPrev(idx);

else if(idx != n&&C[idx+1]->n>=t)

borrowFromNext(idx);

else

{

if(idx != n)

merge(idx);

else

merge(idx-1);

}

}

void BTreeNode::borrowFromPrev(int idx)

{

BTreeNode \*child=C[idx];

BTreeNode \*sibling=C[idx-1];

for(int i=child->n-1;i>=0;i--)

child->keys[i+1]=child->keys[i];

if(!child->leaf)

{

for(int i=child->n;i>=0;i--)

child->C[i+1]=child->C[i];

}

child->keys[0]=keys[idx-1];

if(!child->leaf)

child->C[0]=sibling->C[sibling->n];

keys[idx-1]=sibling->keys[sibling->n-1];

child->n+=1;

sibling->n-=1;

}

void BTreeNode::borrowFromNext(int idx)

{

BTreeNode \*child=C[idx];

BTreeNode \*sibling=C[idx+1];

child->keys[child->n]=keys[idx];

if(!child->leaf) child->C[child->n+1]=sibling->C[0];

keys[idx]=sibling->keys[0];

for(int i=1;i<sibling->n;i++)

sibling->keys[i-1]=sibling->keys[i];

if(!sibling->leaf)

{

for(int i=1;i <= sibling->n;i++)

sibling->C[i-1]=sibling->C[i];

}

child->n+=1;

sibling->n-=1;

}

void BTreeNode::merge(int idx)

{

BTreeNode \*child=C[idx];

BTreeNode \*sibling=C[idx+1];

child->keys[t-1]=keys[idx];

for(int i=0;i<sibling->n;i++)

child->keys[i+t]=sibling->keys[i];

if(!child->leaf)

{

for(int i=0;i <= sibling->n;i++)

child->C[i+t]=sibling->C[i];

}

for(int i=idx+1;i<n;i++)

keys[i-1]=keys[i];

for(int i=idx+2;i <= n;i++)

C[i-1]=C[i];

child->n+=sibling->n+1;

n--;

delete sibling;

}

void BTree::insert(int k)

{

if(!root)

{

root=new BTreeNode(t,true);

root->keys[0]=k;

root->n=1;

}

else

{

if(root->n==2\*t-1)

{

BTreeNode \*s=new BTreeNode(t,false);

s->C[0]=root;

s->splitChild(0,root);

int i=(s->keys[0]<k)?1:0;

s->C[i]->insertNonFull(k);

root=s;

}

else root->insertNonFull(k);

}

}

void BTreeNode::insertNonFull(int k)

{

int i=n-1;

if(leaf)

{

while(i>=0&&keys[i]>k)

{

keys[i+1]=keys[i];

i--;

}

keys[i+1]=k;

n=n+1;

}

else

{

while(i>=0&&keys[i]>k)

i--;

if(C[i+1]->n==2\*t-1)

{

splitChild(i+1,C[i+1]);

if(keys[i+1]<k)

i++;

}

C[i+1]->insertNonFull(k);

}

}

void BTreeNode::splitChild(int i,BTreeNode \*y)

{

BTreeNode \*z=new BTreeNode(y->t,y->leaf);

z->n=t-1;

for(int j=0;j<t-1;j++)

z->keys[j]=y->keys[j+t];

if(!y->leaf)

{

for(int j=0;j<t;j++)

z->C[j]=y->C[j+t];

}

y->n=t-1;

for(int j=n;j>=i+1;j--)

C[j+1]=C[j];

C[i+1]=z;

for(int j=n-1;j>=i;j--)

keys[j+1]=keys[j];

keys[i]=y->keys[t-1];

n=n+1;

}

void BTree::remove(int k)

{

if(!root) return;

root->remove(k);

if(root->n==0)

{

BTreeNode \*tmp=root;

if(root->leaf)

root=nullptr;

else

root=root->C[0];

delete tmp;

}

}

void BTreeNode::traverse()

{

int i;

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

{

if(!leaf)

C[i]->traverse();

cout<<" "<<keys[i];

}

if(!leaf)

C[i]->traverse();

}

int main()

{

BTree t(3);

int arr[]={90,56,23,12,3,6,1,80,6,2,61,99,45,55,22};

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

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

t.insert(arr[i]);

cout<<"Traversal of B-tree after insertion:"<<endl;

t.traverse();

cout<<endl;

t.remove(80);

t.remove(1);

t.remove(56);

t.remove(90);

cout<<"Traversal of B-tree after deletions:"<<endl;

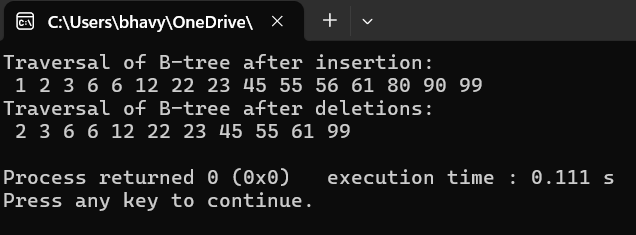
t.traverse();

cout<<endl;

return 0;

}

**Output :**

****