**15B17CI371 – Data Structures Lab**

**ODD 2024**

**Week 9-LAB B**

**Practice Lab**

**Q1. Write a program for a B tree having functions for the following set of operations:**

**● Insert an element(no duplicates are allowed),**

**● Delete an existing element,**

**● Traverse the B Tree(in-order,pre-order,and post-order)**

**INPUT:**

**● Line 1 contains an integer NQ,the number of queries.**

**● Line 2 contains value for minimum number of child pointers of a B tree node.**

**● The next NQ lines contain queries and are of the form 'i xx'(Insert xx into a B tree)**

**or 'd xx'(Delete xx from a B tree).**

**OUTPUT:**

**● Output is a three line answer printing number of split operations,number of merge**

**operations,and the tree traversal of a B tree that results after the execution of all**

**NQ queries.**

**● First line is the total number of split operations performed.**

**● Second line is the total number of merge operations performed.**

**● Third line is the 'Inorder traversal'**

#include <iostream>

using namespace std;

class BTreeNode

{

int \*keys;

BTreeNode \*\*child;

int minDegree;

int n;

bool leaf;

public:

BTreeNode(int minDegree,bool leaf)

{

minDegree=minDegree;

leaf=leaf;

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

child=new BTreeNode \*[2\*minDegree];

n=0;

}

void traverseInOrder()

{

int i;

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

{

if(!leaf)

child[i]->traverseInOrder();

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

}

if(!leaf)

child[i]->traverseInOrder();

}

BTreeNode \*search(int k)

{

int i=0;

while(i<n&&k>keys[i])

i++;

if(keys[i]==k)

return this;

if(leaf)

return nullptr;

return child[i]->search(k);

}

void 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(child[i+1]->n==2\*minDegree-1)

{

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

if(keys[i+1]<k)

i++;

}

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

}

}

void splitChild(int i,BTreeNode \*y)

{

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

z->n=minDegree-1;

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

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

if(!y->leaf)

{

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

z->child[j]=y->child[j+minDegree];

}

y->n=minDegree-1;

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

child[j+1]=child[j];

child[i+1]=z;

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

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

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

n=n+1;

}

void 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);

if(child[idx]->n<minDegree)

fill(idx);

if(flag&&idx>n)

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

else

child[idx]->remove(k);

}

}

int findKey(int k)

{

int idx=0;

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

i++dx;

return idx;

}

void removeFromLeaf(int idx)

{

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

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

n--;

}

void removeFromNonLeaf(int idx)

{

int k=keys[idx];

if(child[idx]->n>=minDegree)

{

int pred=getPredecessor(idx);

keys[idx]=pred;

child[idx]->remove(pred);

}

else if(child[idx+1]->n>=minDegree)

{

int succ=getSuccessor(idx);

keys[idx]=succ;

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

}

else

{

merge(idx);

child[idx]->remove(k);

}

}

int getPredecessor(int idx)

{

BTreeNode \*cur=child[idx];

while(!cur->leaf)

cur=cur->child[cur->n];

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

}

int getSuccessor(int idx)

{

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

while(!cur->leaf)

cur=cur->child[0];

return cur->keys[0];

}

void fill(int idx)

{

if(idx != 0&&child[idx-1]->n>=minDegree)

borrowFromPrev(idx);

else if(idx != n&&child[idx+1]->n>=minDegree)

borrowFromNext(idx);

else

{

if(idx != n)

merge(idx);

else

merge(idx-1);

}

}

void borrowFromPrev(int idx)

{

BTreeNode \*c=child[idx];

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

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

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

if(!c->leaf)

{

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

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

}

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

if(!c->leaf)

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

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

c->n+=1;

sibling->n-=1;

}

void borrowFromNext(int idx)

{

BTreeNode \*c=child[idx];

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

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

if(!c->leaf)

c->child[c->n+1]=sibling->child[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->child[i-1]=sibling->child[i];

}

c->n+=1;

sibling->n-=1;

}

void merge(int idx)

{

BTreeNode \*c=child[idx];

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

c->keys[minDegree-1]=keys[idx];

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

c->keys[i+minDegree]=sibling->keys[i];

if(!c->leaf)

{

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

c->child[i+minDegree]=sibling->child[i];

}

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

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

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

child[i-1]=child[i];

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

n--;

delete sibling;

}

friend class BTree;

};

class BTree

{

BTreeNode \*root;

int minDegree;

int splits,merges;

public:

BTree(int minDegree)

{

root=nullptr;

minDegree=minDegree;

splits=0;

merges=0;

}

void traverseInOrder()

{

if(root != nullptr)

root->traverseInOrder();

cout<<endl;

}

void insert(int k)

{

if(root==nullptr)

{

root=new BTreeNode(minDegree,true);

root->keys[0]=k;

root->n=1;

}

else

{

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

{

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

s->child[0]=root;

s->splitChild(0,root);

splits++;

int i=0;

if(s->keys[0]<k)

i++;

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

root=s;

}

else

root->insertNonFull(k);

}

}

void remove(int k)

{

if(!root)

return;

root->remove(k);

if(root->n==0)

{

BTreeNode \*tmp=root;

if(root->leaf)

root=nullptr;

else

root=root->child[0];

delete tmp;

}

}

int getSplits()

{

return splits;

}

int getMerges()

{

return merges;

}

};

int main()

{

int NQ,minDegree;

cin>>NQ>>minDegree;

BTree btree(minDegree);

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

{

char op;

int val;

cin>>op>>val;

if(op=='i')

btree.insert(val);

else if(op=='d')

btree.remove(val);

}

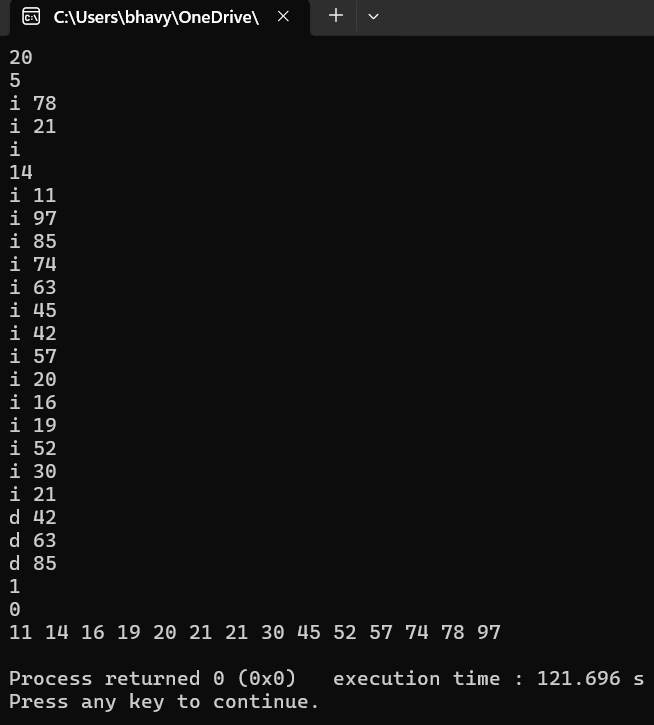
cout<<btree.getSplits()<<endl;

cout<<btree.getMerges()<<endl;

btree.traverseInOrder();

}

**Output :**



**Q2. Write a program to print the nodes of a Threaded Binary Tree from a given**

**Binary Tree.**

**Hint: We will do a reverse in-order traversal,which means we will go to the right child**

**first. Then in the recursive call,we will pass an additional parameter which is the**

**previously visited node. If the right pointer of a node is NULL and the previously visited**

**node is not NULL,we will point the right of the node to the previously visited node and**

**set the boolean rightThread variable to true. The previously visited node should not be**

**changed when making a recursive call to the right subtree,and the real previous visited**

**node should be passed when making a recursive call to the left subtree.**

#include <iostream>

using namespace std;

struct Node

{

int data;

Node\* left;

Node\* right;

bool rightThread;

Node(int val)

{

data=val;

left=right=nullptr;

rightThread=false;

}

};

void convertToThreaded(Node\* root,Node\*& prev)

{

if(root==nullptr)

return;

convertToThreaded(root->right,prev);

if(root->right==nullptr&&prev != nullptr)

{

root->right=prev;

root->rightThread=true;

}

prev=root;

convertToThreaded(root->left,prev);

}

void inOrderTraversal(Node\* root)

{

if(root==nullptr)

return;

Node\* curr=root;

while(curr != nullptr)

{

while(curr->left != nullptr)

curr=curr->left;

while(curr != nullptr)

{

cout<<curr->data<<" ";

if(curr->rightThread)

curr=curr->right;

else

break;

}

curr=curr->right;

}

}

int main()

{

Node\* root=new Node(10);

root->left=new Node(5);

root->right=new Node(20);

root->left->left=new Node(1);

root->right->left=new Node(15);

root->right->right=new Node(30);

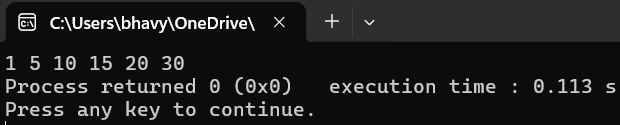
Node\* prev=nullptr;

convertToThreaded(root,prev);

inOrderTraversal(root);

}

**Output :**

****