/\*

Write a function, swapSubTrees, that swaps all of the left and right subtrees

of a binary tree. Add this function to the class binaryTreeType and create a

program to test this function.

\*/

#include <iostream>

using namespace std;

// Class binaryTreeType

//Definition of the node

template <class elemType>

struct nodeType

{

 elemType info;

 nodeType<elemType> \*lLink;

 nodeType<elemType> \*rLink;

};

//Definition of the class

template <class elemType>

class binaryTreeType

{

public:

 const binaryTreeType<elemType>& operator=(const binaryTreeType<elemType>&);

 //overload assignment operator

 bool isEmpty() const;

 //checks if tree is empty

 //postcondition: returns T if empty, F otherwise

 void inorderTraversal()const;

 //does inorder traversal

 //postcondition: nodes printed in inorder sequence

 void preorderTraversal() const;

 //does preoder traversal of tree

 //postcondition: nodes printed in preorder sequence

 void postorderTraversal() const;

 //does postorder traversal

 //postcondition: nodes printed in postorder sequence

 int treeHeight() const;

 //determines height of tree

 //postcondition:returns height of tree

 int treeNodeCount()const;

 //counts nodes in the tree

 //postcondition: returns number of nodes

 int treeLeavesCount()const;

 //counts leaves in the tree

 //postcondition: returns number of leaves

 void destroyTree();

 //destroys tree

 // postcondition: deallocates memory space of each node, root = NULL

 virtual bool search(const elemType& searchItem) const = 0;

 //determines if searchItem is in tree

 //postcondition: returns true if searchItem found, fals otherwise

 virtual void insert(const elemType& insertItem) = 0;

 //inserts insertItem into tree

 //postcondition: if no node has same info as insertItem, a node with

 // the info insertItem is created and inserted into tree

 virtual void deleteNode(const elemType& deleteItem) = 0;

 //deletes deletItem from tree

 //postcondition: if a node with same info as deleteItem is found it is

 //deleted from the tree. If tree is empty or delteItem not in tree

 //a message is printed

 binaryTreeType(const binaryTreeType<elemType>& otherTree);

 //copy constructor

 binaryTreeType();

 //default constructor

 ~binaryTreeType();

 //Destructor

protected:

 nodeType<elemType> \*root;

private:

 void copyTree(nodeType<elemType>\* &copiedTreeRoot,

 nodeType<elemType>\* otherTreeRoot);

 //copies the tree to which otherTreeRoot points

 //postcondition:pointer copiedTreeRoot points to the root of the copied

 //tree

 void destroy(nodeType<elemType>\* &p);

 //destroys the tree to which p points

 //postcondition: memory occupied by tree to which p points is

 //deallocated. P = NULL;

 void inorder(nodeType<elemType> \*p)const;

 //in order traversal of tree to which p points

 //postcondition: nodes of tree to which p points printed in inorder

 //sequence

 void preorder(nodeType<elemType> \*p)const;

 //pre order traversal of tree to which p points

 //postcondition: nodes of tree to which p points printed in preorder

 //sequence

 void postorder(nodeType<elemType> \*p)const;

 //post order traversal of tree to which p points

 //postcondition: nodes of tree to which p points printed in postorder

 //sequence

 int height(nodeType<elemType> \*p)const;

 //determines height of tree to which p points

 //postcondition: retursn height of tree to which p points

 int max(int x, int y)const;

 //determines the larger of x and y

 //postcondition: returns larger of z and y

 int nodeCount(nodeType<elemType> \*p)const;

 //counts number of nodes in tree to which p points

 //postcondition: returns number of nodes in tree to which p points

 int leavesCount(nodeType<elemType> \*p)const;

 //counts number of leaves in tree to which p points

 //postcondition: returns number of leaves in tree to which p points

/\*

 void swapSubTrees();

 //swaps all of the left and right subtrees of binary tree

 \*/

};

template<class elemType>

bool binaryTreeType<elemType>::isEmpty() const

{

 return (root == NULL);

}

template<class elemType>

binaryTreeType<elemType>::binaryTreeType()

{

 root == NULL;

}

template<class elemType>

void binaryTreeType<elemType>::inorderTraversal() const

{

 inorder(root);

}

template<class elemType>

void binaryTreeType<elemType>::preorderTraversal() const

{

 preorder(root);

}

template<class elemType>

void binaryTreeType<elemType>::postorderTraversal() const

{

 postorder(root);

}

template<class elemType>

int binaryTreeType<elemType>::treeHeight() const

{

 return height(root);

}

template<class elemType>

int binaryTreeType<elemType>::treeNodeCount() const

{

 return nodeCount(root);

}

template<class elemType>

int binaryTreeType<elemType>::treeLeavesCount() const

{

 return leavesCount(root);

}

template<class elemType>

void binaryTreeType<elemType>::inorder(nodeType<elemType> \*p) const

{

 if (p != NULL)

 {

 inorder(p->lLink);

 cout << p->info << " ";

 inorder(p->rLink);

 }

}

template<class elemType>

void binaryTreeType<elemType>::preorder(nodeType<elemType> \*p) const

{

 if(p != NULL)

 {

 cout << p->info << " ";

 preorder(p->lLink);

 preorder(p->rLink);

 }

}

template<class elemType>

void binaryTreeType<elemType>::postorder(nodeType<elemType> \*p) const

{

 if(p != NULL)

 {

 postorder(p->lLink);

 postorder(p->rLink);

 cout << p->info << " ";

 }

}

template<class elemType>

int binaryTreeType<elemType>::height(nodeType<elemType> \*p) const

{

 if(p == NULL)

 return 0;

 else

 return 1 + max(height(p->lLink), height(p->rLink));

}

template<class elemType>

int binaryTreeType<elemType>::max(int x, int y) const

{

 if(x > y)

 return x;

 else

 return y;

}

//nodeCount and leavesCount are left out for me to code if I need them

template<class elemType>

void binaryTreeType<elemType>::copyTree(nodeType<elemType>\* &copiedTreeRoot,

 nodeType<elemType>\* otherTreeRoot)

{

 if(otherTreeRoot == NULL)

 copiedTreeRoot = NULL;

 else

 {

 copiedTreeRoot = new nodeType<elemType>;

 copiedTreeRoot->info = otherTreeRoot->info;

 copyTree(copiedTreeRoot->lLink, otherTreeRoot->lLink);

 copyTree(copiedTreeRoot->rLink, otherTreeRoot->rLink);

 }//end copyTree

}

template<class elemType>

void binaryTreeType<elemType>::destroy(nodeType<elemType>\* &p)

{

 if(p != NULL)

 {

 destroy(p->lLink);

 destroy(p->rLink);

 delete p;

 p = NULL;

 }

}

template<class elemType>

void binaryTreeType<elemType>::destroyTree()

{

 destroy(root);

}

//copy constructor

template<class elemType>

binaryTreeType<elemType>::binaryTreeType

 (const binaryTreeType<elemType>& otherTree)

{

 if(otherTree.root == NULL) //otherTree is empty

 root = NULL;

 else

 copyTree(root, otherTree.root);

}

//destructor

template<class elemType>

binaryTreeType<elemType>::~binaryTreeType()

{

 destroy(root);

}

//overload the assignment operator

template<class elemType>

const binaryTreeType<elemType>& binaryTreeType<elemType>::

 operator=(const binaryTreeType<elemType>& otherTree)

{

 if(this != &otherTree) //avoid self copy

 {

 if(root != NULL) //if tree not empty

 destroy(root); //destroy the tree

 if(otherTree.root == NULL) //if tree not empty

 root = NULL;

 else

 copyTree(root, otherTree.root);

 }//end else

 return \*this;

}

/\*

//here is my swapSubTrees function to be written and tested

//I think my algorithm below will work if node type is a pointer

template<class elemType>

void binaryTreeType<elemType>::swapSubTrees()

{

nodeType Temp; = root->lLink;

root->llink = root->rLink;

root->rlink = Temp;

delete Temp; // do I need to do this?

}

\*/