|  |  |
| --- | --- |
| **Application/ Program name:** | L4-1 |
| **Written by:** | Bailey Nichols |

|  |
| --- |
| **Purpose or problem definition:** |
| The purpose of this program is to demonstrate the use of binary trees as a data structure.   * I am to first define and implement a binary tree class, then create three instances called T1, T2, and T3 * Then I am to do a post-order traversal of T1 and while doing that insert each node into T2 * Do a preorder traversal of T2 and as above insert into T3 * Do an inorder traversal of T3 * Call swapsubtrees() function in T3 and then repeat the inorder traversal * Output the heights of the trees and the number of leaves in each of the three bSearchTree instances |
|  |
| **Program Procedures:** |
| The program must contain a binary tree structure and then inherit that class to preform search functions? <<why tho, you might think >> |
|  |
| **Algorithm/Processing/Conditions:** |
| **Inputs:** |
| Four of the traversal functions will be overwritten to take another tree, and the .cpp file written by the user should somehow accept a string. |
| **Processes:** |
| Two of the traversal functions will be overwritten to accept a tree and then insert the data into that tree as per the instructions. |
| **Outputs:** |
| There will be a couple functions in binarySearchTreeType.h that output stuff to console, otherwise console output will eb directed by the user. |
|  |
| **Notes & Restriction:** |
| \_\_\_\_Refrence\_\_\_\_  All code comes from this book or is highly inspired from it  Malik, D. S. (2018). *C++ Programming: Program Design Including Data Structures.* Pearson Education. Kindle Edition. Page 1373  So finally I have this working, on the 8th I re-wrote the classes and members again for the 4th time and in doing so while asking questions in an online chat group I was able to understand that you cannot access root directly in the binarySearchTreeType.h I had understood this when writing my own classes during ‘plan B’ but did not connect the dots to understand that the textbook was actually telling me to write the wrong code.  All the notes and code below are just added for the benefit of understanding how much work I did on this project and the process. They do not actually relate to the finished project as it is delivered.  So my first three attempts became failures at the debugging stage when one tiny issue ran into another and another and etc.  So plan B:  I just went through the slideshow and wrote functions on paper with a pencil and then took those and made the classes you wanted from very basic structures in my own time.  //This is the main.cpp program to run the header files  #include <iostream>  #include "binarySearchTreeType.h"  using namespace std;  int main()  {  binarySearchTreeType<int> t1;  binarySearchTreeType<int> t2;  binarySearchTreeType<int> t3;  int n;  cout << "Enter numbers seperated by spaces with the final number being -999" << endl;  cout << "[> ";      /\*cin >> n;  t1.insert(n);  while (n != -999) {  t1.insert(n);  cin >> n;  }  \*/  //for faster debugging  for (int i = 0; i < 10; ++i) {  t1.insert(i);  }  t1.postorderTraversal(t2);  cout << endl;  t2.postorderTraversal(t3);  cout << endl;  t3.inorderTraversal();  cout << endl;  t3.swapsubtree();  t3.inorderTraversal();  cout << endl;  cout << "T1 Leaf count: "<<t1.treeLeavesCount() <<  " Node count: " << t1.treeNodeCount() <<" Height: " << t1.treeHeight();  return 0;    }  //main.cpp ends here  This is a small program demonstrating that any pointer that is a nullptr will return false.  int main()  {  int \*c;    c = nullptr;  if (!c){  cout<<"Hella wurld"<<endl;  }  return 0;  }  I have a bunch of Notes on paper  I tried to do it via the textbook but man idk. Never have luck with the code from the book. All of this below is just code that didint work but I wanted to keep to look at.  #include <iostream>  //using namespace std;  using *namespace* std;  //definition of the node  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1373  *template* < *typename* elemType >  *struct* nodeType{  elemType info;  nodeType<elemType> \*left;  nodeType<elemType> \*right;  };  //definition of the class  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1373-1381  *template* < *typename* elemType >  *class* binaryTreeType{  *public:*  const binaryTreeType<elemType>& operator=(const binaryTreeType<elemType>&);  //overlaod the assignment operator  *bool* isEmpty() const;  //returns true if the tree is empty, otherwise returns false  *void* inorderTraversal() const;  //performs an inorder traversal of the tree  //postcondition: the tree has been traversed in order  *void* preorderTraversal() const;  //performs a preorder traversal of the tree  //postcondition: the tree has been traversed in preorder  *void* postorderTraversal() const;  //performs a postorder traversal of the tree  //postcondition: the tree has been traversed in postorder  *int* treeHeight() const;  //calculates the height of the tree  //postcondition: the height of the tree has been calculated  *int* treeNodeCount() const;  //calculates the number of nodes in the tree  //postcondition: the number of nodes in the tree has been calculated  *int* treeLeavesCount() const;  //calculates the number of leaves in the tree  //postcondition: the number of leaves in the tree has been calculated  *void* destroyTree();  //destroys the tree  virtual *bool* search(const elemType& *searchItem*) const = 0;  //searches for a specific value in the tree  //postcondition: the value has been searched for returns bool  virtual *void* insertNode(const elemType& *insertItem*) = 0;  //inserts a node into the tree  //postcondition: the node has been inserted if there is no duplicate  virtual *void* deleteNode(const elemType& *deleteItem*) = 0;  //deletes a node from the tree  //postcondition: the node has been deleted, unless there is no node with the value  // in the tree, then an error message is displayed  binaryTreeType(const binaryTreeType<elemType>& *otherTree*);  //copy constructor  binaryTreeType();  //default constructor  ~binaryTreeType();  //destructor  *protected:*  nodeType<elemType> \*root;  //pointer to the root of the tree  *private:*  *void* copyTree(nodeType<elemType> \*& *copy*, nodeType<elemType> \**otherTree*);  //copies the tree  //precondition: otherTree is assigned  *void* destroy(nodeType<elemType> \*& *tree*);  //destroys the tree  *void* inorder(nodeType<elemType> \**tree*) const;  //performs an inorder traversal of the tree  //postcondition: the tree has been traversed in order  *void* preorder(nodeType<elemType> \**tree*) const;  //performs a preorder traversal of the tree  //postcondition: the tree has been traversed in preorder  *void* postorder(nodeType<elemType> \**tree*) const;  //performs a postorder traversal of the tree  //postcondition: the tree has been traversed in postorder  *int* height(nodeType<elemType> \**tree*) const;  //calculates the height of the tree  //postcondition: the height of the tree is returned  *int* max(*int* *num1*, *int* *num2*) const;  //returns the larger of two numbers  //postcondition: the larger of two numbers is returned  *int* nodeCount(nodeType<elemType> \**tree*) const;  //calculates the number of nodes in the tree  //postcondition: the number of nodes in the tree is returned  *int* leavesCount(nodeType<elemType> \**tree*) const;  //calculates the number of leaves in the tree  //postcondition: the number of leaves in the tree is returned  };  //definitions of the class functions declared above  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1376- 1377  *template* < *typename* elemType >  *bool* binaryTreeType<elemType>::isEmpty() const{  return root == nullptr;  }  *template* < *typename* elemType >  binaryTreeType<elemType>::binaryTreeType(){  root = nullptr;  printf("Bailey Nichols \t L4-1 \t L4-1.cpp");  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::inorderTraversal() const{  inOrder(root);  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::preorderTraversal() const{  preOrder(root);  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::postorderTraversal() const{  postOrder(root);  }  *template* < *typename* elemType >  *int* binaryTreeType<elemType>::treeHeight() const{  return height(root);  }  *template* < *typename* elemType >  *int* binaryTreeType<elemType>::treeNodeCount() const{  return nodeCount(root);  }  *template* < *typename* elemType >  *int* binaryTreeType<elemType>::treeLeavesCount() const{  return leavesCount(root);  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::inorder(nodeType<elemType> \**tree*) const{  if(*tree* != nullptr){  inOrder(*tree*->left);  cout << *tree*->info << " ";  inOrder(*tree*->right);  }  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::preorder(nodeType<elemType> \**tree*) const{  if(*tree* != nullptr){  cout << *tree*->info << " ";  preOrder(*tree*->left);  preOrder(*tree*->right);  }  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::postorder(nodeType<elemType> \**tree*) const{  if(*tree* != nullptr){  postOrder(*tree*->left);  postOrder(*tree*->right);  cout << *tree*->info << " ";  }  }  *template* < *typename* elemType >  *int* binaryTreeType<elemType>::height(nodeType<elemType> \**tree*) const{  if(*tree* == nullptr){  return 0;  }  else{  *int* leftHeight = height(*tree*->left);  *int* rightHeight = height(*tree*->right);  if(leftHeight > rightHeight){  return leftHeight + 1;  }  else{  return rightHeight + 1;  }  }  }  *template* < *typename* elemType >  *int* binaryTreeType<elemType>::max(*int* *num1*, *int* *num2*) const{  if(*num1* > *num2*){  return *num1*;  }  else{  return *num2*;  }  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::copyTree(nodeType<elemType> \*& *copy*, nodeType<elemType> \**otherTree*){  if(*otherTree* != nullptr){  *copy* = new nodeType<elemType>;  *copy*->info = *otherTree*->info;  copyTree(*copy*->left, *otherTree*->left);  copyTree(*copy*->right, *otherTree*->right);  }  else{  *copy* = nullptr;  }  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::destroy(nodeType<elemType> \*& *tree*){  if(*tree* != nullptr){  destroy(*tree*->left);  destroy(*tree*->right);  delete *tree*;  *tree* = nullptr;  }  }  *template* < *typename* elemType >  *void* binaryTreeType<elemType>::destroyTree(){  destroy(root);  }  *template* < *typename* elemType >  binaryTreeType<elemType>::binaryTreeType(const binaryTreeType<elemType>& *otherTree*){  if(*otherTree*.root == nullptr){  root = nullptr;  }  else{  copyTree(root, *otherTree*.root);  }  }  *template* < *typename* elemType >  binaryTreeType<elemType>::~binaryTreeType(){  destroyTree();  }  *template* < *typename* elemType >  const binaryTreeType<elemType>& binaryTreeType<elemType>::operator=(const binaryTreeType<elemType>& *otherTree*){  if(this != *otherTree*){  if(root != nullptr){  destroy(root);  }  if(*otherTree*.root == nullptr){  root = nullptr;  }  else{  copyTree(root, *otherTree*.root);  }  }  return \*this;  }  #include "binaryTree.h"  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1383-1387  *template* <*class* elemType>  *class* binarySearchTree : *public* binaryTreeType<elemType>  {  *public:*  *bool* search(const elemType& *searchItem*) const;  //Function: determines whether searchItem is in the binary search tree.  //Precondition: searchItem is an object of the elemType class.  //Postcondition: returns true if searchItem is in the binary search tree, false otherwise.  *void* insert(const elemType& *insertItem*);  //Function: inserts insertItem into the binary search tree.  //Precondition: insertItem is an object of the elemType class.  //Postcondition: If insertItem is not in the binary search tree, insertItem is inserted.  *void* deleteNode(const elemType& *deleteItem*);  //Function: deletes deleteItem from the binary search tree.  //Precondition: deleteItem is an object of the elemType class.  //Postcondition: If deleteItem is in the binary search tree, deleteItem is deleted.  *void* deleteFromTree(nodeType<elemType>\* &*pTree*);  //Function: deletes the node pointed to by pTree.  //Precondition: pTree is a pointer to a node in the binary search tree.  //Postcondition: If pTree is not a null pointer, the node pointed to by pTree is deleted.  //functions for the assignment  *void* inorderTraversal() const;  //Function: traverses the binary search tree in inorder and displays the data in each node.  *void* preorderTraversal() const;  //Function: traverses the binary search tree in preorder and displays the data in each node.  *void* postorderTraversal() const;  //Function: traverses the binary search tree in postorder and displays the data in each node.  };  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1385  *template* <*class* elemType>  *bool* binarySearchTree<elemType>::search(const elemType& *searchItem*) const  {  nodeType<elemType>\* p = this->rootPtr;  while (p != nullptr)  {  if (p->data == *searchItem*)  return true;  else if (p->data > *searchItem*)  p = p->leftPtr;  else  p = p->rightPtr;  }  return false;  }  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1386  *template* <*class* elemType>  *void* binarySearchTree<elemType>::insert(const elemType& *insertItem*)  {  nodeType<elemType>\* current; //pointer to traverse the tree  nodeType<elemType>\* trailCurrent= nullptr; //pointer behind current  nodeType<elemType>\* newNode; //pointer to create the node  //init newNode  newNode = new nodeType<elemType>; //create the node  newNode->data = *insertItem*; //insert the data in the node  newNode->leftPtr = nullptr; //set the left child pointer to null  newNode->rightPtr = nullptr; //set the right child pointer to null  //now this is where the fun begins  if (this->rootPtr == nullptr) //if the tree is empty  this->rootPtr = newNode; //insert the node at the root  else  {  current = this->rootPtr;  while (current != nullptr)  {  trailCurrent = current;  if (*insertItem* < current->data)  current = current->leftPtr;  else  current = current->rightPtr;  }  if (*insertItem* < trailCurrent->data)  trailCurrent->leftPtr = newNode;  else  trailCurrent->rightPtr = newNode;  }  }  //--------------------------------------------------------------------------------------------------------------------------------------------------  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1387  *template* <*class* elemType>  *void* binarySearchTree<elemType>::deleteFromTree(nodeType<elemType>\* &*p*){  nodeType<elemType>\* current;  nodeType<elemType>\* trailCurrent;  nodeType<elemType>\* temp;  if (*p* == nullptr){  cout << "Cannot delete empty node." << endl;  return;  }  else if (*p*->leftPtr == nullptr && *p*->rightPtr == nullptr){  temp = *p*;  *p* = nullptr;  delete temp;  }  else if (*p*->leftPtr == nullptr){  temp = *p*;  *p* = temp->rightPtr;  delete temp;  }  else if (*p*->rightPtr == nullptr){  temp = *p*;  *p* = *p*->leftPtr;  delete temp;  }  else{  current = *p*->leftPtr;  trailCurrent = nullptr;  while (current->rightPtr != nullptr){  trailCurrent = current;  current = current->rightPtr;  }  *p*->data = current->data;  if (trailCurrent == nullptr)  *p*->leftPtr = current->leftPtr;  else  trailCurrent->rightPtr = current->leftPtr;  delete current;  }//end else  }  //--------------------------------------------------------------------------------------------------------------------------------------------------  //Malik, D. S. (2018). C++ Programming: Program Design Including Data Structures. Pearson Education. Kindle Edition. Page 1391  *template* <*class* elemType>  *void* binarySearchTree<elemType>::deleteNode(const elemType& *deleteItem*)  {  nodeType<elemType>\* current; //pointer to traverse the tree  nodeType<elemType>\* trailCurrent; //pointer behind current  *bool* found = false;  if (this->rootPtr == nullptr)  cout << "Cannot delete from an empty tree." << endl;  else  {  current = this->rootPtr;  trailCurrent = nullptr;  while (current != nullptr && !found)  {  if (current->data == *deleteItem*)  found = true;  else  {  trailCurrent = current;  if (current->data > *deleteItem*)  current = current->leftPtr;  else  current = current->rightPtr;  }  }  if (current == nullptr)  cout << "The item to be deleted is not in the tree." << endl;  else if (found)  {  if (current == this->rootPtr) //Case 1: Deleting the root  deleteFromTree(this->rootPtr);  else if (trailCurrent->leftPtr == current) //Case 2: Deleting a left child  deleteFromTree(trailCurrent->leftPtr);  else //Case 3: Deleting a right child  deleteFromTree(trailCurrent->rightPtr);  }  }  }  // I wrote these but they are super simple they just call the inherited functions from the binary tree class  *template* <*class* elemType>  *void* binarySearchTree<elemType>::inorderTraversal() const  {  inorder(this->rootPtr);  }  *template* <*class* elemType>  *void* binarySearchTree<elemType>::postorderTraversal() const  {  postorder(this->rootPtr);  }  *template* <*class* elemType>  *void* binarySearchTree<elemType>::preorderTraversal() const  {  preorder(this->rootPtr);  }  #pragma once  #include <iostream>  //first the treenode struct  *template* <*class* T>  *struct* treenode  {  T data;  treenode<T> \*left;  treenode<T> \*right;  treenode()  {  left = nullptr;  right = nullptr;  }  };  //  //now the binaryTreeType class  *template* <*class* T>  *class* binaryTreeType  {  *public:*  //overload operator  binaryTreeType<T>& operator=(const binaryTreeType<T>&);  //constructor  binaryTreeType();  //destructor  ~binaryTreeType();  //insert  *void* insert(const T&);  //count leaves  *int* count\_leaves();  //height of binaryTreeType  *int* height();  //print the binaryTreeType in order  *void* inorderPf();  //  treenode<T> \*root;  //  };  //  //now the implementation  //  //constructor  *template* <*class* T>  binaryTreeType<T>::binaryTreeType()  {  root = nullptr;  }  //insert  //Malik D.S. C++ Programming Program Design Including Data Structures (MindTap Course List) Kindle Edition (2014) pg 1386  *template* <*class* T>  *void* binaryTreeType<T>::insert(const T& *insertItem*)  {  std::cout<< "Flag";  treenode<T>\* current; //pointer to traverse the tree  treenode<T>\* trailCurrent= nullptr; //pointer behind current  treenode<T>\* newNode; //pointer to create the node  //init newNode  newNode = new treenode<T>; //create the node  newNode->data = *insertItem*; //insert the data in the node  newNode->left = nullptr; //set the left child pointer to null  newNode->right = nullptr; //set the right child pointer to null  //now this is where the fun begins  if (this->root == nullptr) //if the tree is empty  this->root = newNode; //insert the node at the root  else  {  current = this->root; //set current to the root  while (current != nullptr)  {  trailCurrent = current;  if (*insertItem* < current->data)  current = current->left;  else  current = current->right;  }  if (*insertItem* < trailCurrent->data)  trailCurrent->left = newNode;  else  trailCurrent->right = newNode;  }  }  //count leaves  *template* <*class* T>  *int* binaryTreeType<T>::count\_leaves()  {  *int* count = 0;  treenode<T> \*current = root;  while(current != nullptr)//this is the count leaves loop  {  if(current->left == nullptr && current->right == nullptr)//if current is a leaf, increment count  {  count++;  }  current = current->left;//otherwise, go left  }  return count;  }  //height  *template* <*class* T>  *int* binaryTreeType<T>::height()  {  *int* height = 0;  treenode<T> \*current = root;  while(current != nullptr)//this is the height loop  {  if(current->left == nullptr && current->right == nullptr)//if current is a leaf, increment height  {  height++;  }  current = current->left;//otherwise, go left  }  return height;  }  //  //now the destructor  *template* <*class* T>  binaryTreeType<T>::~binaryTreeType()  {  treenode<T> \*current = root;  treenode<T> \*temp;  while(current != nullptr)//this is the destructor loop  {  if(current->left == nullptr && current->right == nullptr)//if current is a leaf, delete current  {  temp = current;  current = current->left;  delete temp;  }  else  {  current = current->left;//otherwise, go left  }  }  }  //overload operator  *template* <*class* T>  binaryTreeType<T>& binaryTreeType<T>::operator=(const binaryTreeType<T>& *right*)  {  if(this != &*right*)//if this is not the same as right  {  treenode<T> \*current = root;  treenode<T> \*temp;  while(current != nullptr)//this is the destructor loop  {  if(current->left == nullptr && current->right == nullptr)//if current is a leaf, delete current  {  temp = current;  current = current->left;  delete temp;  }  else  {  current = current->left;//otherwise, go left  }  }  root = nullptr;  current = *right*.root;  while(current != nullptr)//this is the copy constructor loop  {  if(current->left == nullptr && current->right == nullptr)//if current is a leaf, insert current  {  insert(current->data);  }  else  {  insert(current->data);  }  current = current->left;//otherwise, go left  }  }  return \*this;  }  //----------------------------------------------------------------------------------------------------------------------  //Mali D.S. C++ Programming Program Design Including Data Structures (MindTap Course List) Kindle Edition (2014) pg 1386  //print the binaryTreeType in order  *template* <*class* T>  *void* binaryTreeType<T>::inorderPf()  {  treenode<T> \* current = root;  if (root == nullptr)  {  std::cout << "The tree is empty" << std::endl;  }  else  {  while (current != nullptr)  {  std::cout << current->data << " ";  if (current->left == nullptr)  {  current = current->right;  }  else  {  current = current->left;  }  }  }  }  //----------------------------------------------------------------------------------------------------------------------  //now the searchTreeType class  *template* <*class* T>  *class* bSearchTree : *public* binaryTreeType<T>  {  *public:*  //constructor  bSearchTree();  *void* inorderTraversal();  *void* preorderTraversal(bSearchTree<T>&);  *void* postorderTraversal(bSearchTree<T>&);  *void* swapSubtrees();  };  //  //now the implementation  //  //constructor  *template* <*class* T>  bSearchTree<T>::bSearchTree()  {  binaryTreeType<T>::root = nullptr;  }  //inorder traversal  //Malik D.S. C++ Programming Program Design Including Data Structures (MindTap Course List) Kindle Edition (2014) pg 1386  *template* <*class* T>  *void* bSearchTree<T>::inorderTraversal()  {  treenode<T> \*current = this->root;  if (this->root == nullptr)  {  std::cout << "The tree is empty" << std::endl;  }  else  {  while (current != nullptr)  {  std::cout << current->data << " ";  //tree.insert(current->data);  if (current->left == nullptr)  {  current = current->right;  }  else  {  current = current->left;  }  }  return;  }  }  //----------------------------------------------------------------------------------------------------------------------  //Malik D.S. C++ Programming Program Design Including Data Structures (MindTap Course List) Kindle Edition (2014) pg 1386  //Knuth. The Art of Computer Programming, Vol. 1 (3rd ed.), pp 318-321.  *template* <*class* T>  *void* bSearchTree<T>::preorderTraversal(bSearchTree<T>& *tree*)  {  treenode<T> \*current = this->root;  if(this->root == nullptr)  {  std::cout << "The tree is empty" << std::endl;  }  else  {  while(current != nullptr)  {  if(current->left == nullptr)  {  current = current->right;  }  current = current->left;  std::cout << current->data << " ";  *tree*.insert(current->data);  }  return;  }  }  //----------------------------------------------------------------------------------------------------------------------  //Wikipedia the free encyclopedia, Tree Traversal  *template* <*class* T>  *void* bSearchTree<T>::postorderTraversal(bSearchTree<T>& *tree*)  {  treenode<T> \*current = this->root;  if(this->root == nullptr)  {  std::cout << "The tree is empty" << std::endl;  }  else  {  while(current != nullptr)  {  std::cout << current->data << " ";  *tree*.insert(current->data);  if (current->left == nullptr)  {  current = current->right;  }  current = current->left;  }  return;  }  } |
|  |
| **Comments:** |
| That was less than fun. |