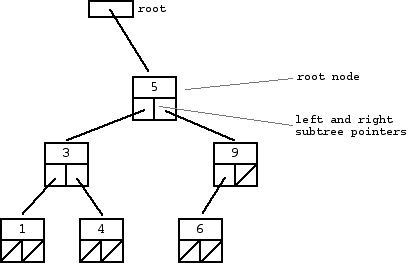
**Name:**

**Advanced Programming in C++**

**Lab Exercise 4/30/2024**

**Introduction to Binary Trees**

A **binary tree** is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree.



A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are less-or-equal to the node (<=), and all the elements in its right subtree are greater than the node (>). The tree shown above is a binary search tree -- the "root" node is a 5, and its left subtree nodes (1, 3, 4) are <= 5, and its right subtree nodes (6, 9) are > 5.

Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the 1 <= 3 and 4 > 3. Watch out for the exact wording in the problems -- a "binary search tree" is different from a "binary tree". The nodes at the bottom edge of the tree have empty subtrees and are called "leaf" nodes (1, 4, 6) while the others are "internal" nodes (3, 5, 9).

In this lab, you will be using the IntBinaryTree class which can be found in the Data Files folder. Answer the following questions about the IntBinaryTree class. The IntBinaryTree Class files (.h and .cpp) can be found in the

\\Ada\data files\C++\IntBinaryTree Class folder.

1. List the private members of the IntBinaryTree class.
2. List the private member functions of the IntBinaryTree class.
3. List the public member functions of the IntBinaryTree class.
4. What specifically happens when the tree is constructed?
5. What specifically happens when a node is removed from the tree?

**Problem 1**

In this exercise you will learn to work with the Binary Tree data structure. Write a main program that exercises the IntBinaryTree class. Your program should do the following:

1. Create an empty IntBinaryTree object called *tree*.

IntBinaryTree tree;

1. Add the following values to the tree; 45, 78, 38, 17, 9, 57, 88, 2, 19, 51.

tree.insertNode(45);

tree.insertNode(78);

tree.insertNode(38);

tree.insertNode(17);

tree.insertNode(9);

tree.insertNode(57);

tree.insertNode(88);

tree.insertNode(2);

tree.insertNode(19);

tree.insertNode(51);

1. Display an inorder, preorder, postorder traversal of the tree.

tree.displayInOrder();

tree.displayPostOrder();

tree.displayPreOrder();

1. Remove the values 57 and 38 from the tree.

tree.remove(57);

tree.remove(38);

1. Display an inorder, preorder, postorder traversal of the tree.

tree.displayInOrder();

tree.displayPostOrder();

tree.displayPreOrder();

**Problem 2**

Now that you have your IntBinaryTree class working, let us convert it to a template class called BinaryTree class.

Note: You do this the same way that you did for creating the template LinkedList class. You will need to fix BinaryTree.h and BinaryTree.cpp.

When you have completed these tasks, print your main.cpp program from Problem 1 and your new class files (BinaryTree.h and BinaryTree.cpp) for your templated class and attach them to this sheet and turn in.

Here is the class definition file:

// IntBinaryTree.h

#ifndef INTBINARYTREE\_H

#define INTBINARYTREE\_H

class IntBinaryTree

{

private:

struct TreeNode

{

int value;

TreeNode \*left;

TreeNode \*right;

};

TreeNode \*root;

void insert(TreeNode \*&, TreeNode \*&);

void destroySubTree(TreeNode \*);

void deleteNode(int, TreeNode \*&);

void makeDeletion(TreeNode \*&);

void displayInOrder(TreeNode \*);

void displayPreOrder(TreeNode \*);

void displayPostOrder(TreeNode \*);

public:

IntBinaryTree(); // Constructor

~IntBinaryTree(); // Destructor

void insertNode(int);

bool searchNode(int);

void remove(int);

void displayInOrder();

void displayPreOrder();

void displayPostOrder();

};

#endif

Here is the class implementation file

// IntBinaryTree.cpp

// Implementation file for the IntBinaryTree class

#include <iostream>

#include "IntBinaryTree.h"

using namespace std;

IntBinaryTree::IntBinaryTree() // Constructor

{

root = NULL;

}

IntBinaryTree::~IntBinaryTree() // Destructor

{

destroySubTree(root);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// insert accepts a TreeNode pointer and a pointer to a node. \*

// The function inserts the node into the tree pointed to by \*

// the TreeNode pointer. This function is called recursively. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::insert(TreeNode \*&nodePtr, TreeNode \*&newNode)

{

if (nodePtr == NULL)

nodePtr = newNode; // Insert the node.

else if (newNode->value < nodePtr->value)

insert(nodePtr->left, newNode); // Search the left branch

else

insert(nodePtr->right, newNode); // Search the right branch

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// insertNode creates a new node to hold num as its value, \*

// and passes it to the insert function. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::insertNode(int num)

{

TreeNode \*newNode; // Pointer to a new node.

// Create a new node and store num in it.

newNode = new TreeNode;

newNode->value = num;

newNode->left = newNode->right = NULL;

// Insert the node.

insert(root, newNode);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// destroySubTree is called by the destructor. It \*

// deletes all nodes in the tree. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::destroySubTree(TreeNode \*nodePtr)

{

if (nodePtr)

{

if (nodePtr->left)

destroySubTree(nodePtr->left);

if (nodePtr->right)

destroySubTree(nodePtr->right);

delete nodePtr;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// searchNode determines if a value is present in \*

// the tree. If so, the function returns true. \*

// Otherwise, it returns false. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool IntBinaryTree::searchNode(int num)

{

TreeNode \*nodePtr = root;

while (nodePtr)

{

if (nodePtr->value == num)

return true;

else if (num < nodePtr->value)

nodePtr = nodePtr->left;

else

nodePtr = nodePtr->right;

}

return false;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// remove calls deleteNode to delete the \*

// node whose value member is the same as num. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::remove(int num)

{

deleteNode(num, root);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// deleteNode deletes the node whose value \*

// member is the same as num. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::deleteNode(int num, TreeNode \*&nodePtr)

{

if (num < nodePtr->value)

deleteNode(num, nodePtr->left);

else if (num > nodePtr->value)

deleteNode(num, nodePtr->right);

else

makeDeletion(nodePtr);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// makeDeletion takes a reference to a pointer to the node \*

// that is to be deleted. The node is removed and the \*

// branches of the tree below the node are reattached. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::makeDeletion(TreeNode \*&nodePtr)

{

TreeNode \*tempNodePtr; // Temporary pointer, used in reattaching the

// left subtree.

if (nodePtr == NULL)

cout << "Cannot delete empty node.\n";

else if (nodePtr->right == NULL)

{

tempNodePtr = nodePtr;

nodePtr = nodePtr->left; // Reattach the left child

delete tempNodePtr;

}

else if (nodePtr->left == NULL)

{

tempNodePtr = nodePtr;

nodePtr = nodePtr->right; // Reattach the right child

delete tempNodePtr;

}

// If the node has two children.

else

{

// Move one node the right.

tempNodePtr = nodePtr->right;

// Go to the end left node.

while (tempNodePtr->left)

tempNodePtr = tempNodePtr->left;

// Reattach the left subtree.

tempNodePtr->left = nodePtr->left;

tempNodePtr = nodePtr;

// Reattach the right subtree.

nodePtr = nodePtr->right;

delete tempNodePtr;

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The displayInOrder member function displays the values \*

// in the subtree pointed to by nodePtr, via inorder traversal. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::displayInOrder(TreeNode \*nodePtr)

{

if (nodePtr)

{

displayInOrder(nodePtr->left);

cout << nodePtr->value << endl;

displayInOrder(nodePtr->right);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The displayPreOrder member function displays the values \*

// in the subtree pointed to by nodePtr, via preorder traversal. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::displayPreOrder(TreeNode \*nodePtr)

{

if (nodePtr)

{

cout << nodePtr->value << endl;

displayPreOrder(nodePtr->left);

displayPreOrder(nodePtr->right);

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The displayPostOrder member function displays the values \*

// in the subtree pointed to by nodePtr, via postorder traversal.\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void IntBinaryTree::displayPostOrder(TreeNode \*nodePtr)

{

if (nodePtr)

{

displayPostOrder(nodePtr->left);

displayPostOrder(nodePtr->right);

cout << nodePtr->value << endl;

}

}

// Overloaded display functions that pass the root to their overloaded counterparts

void IntBinaryTree::displayInOrder()

{

displayInOrder(root);

}

void IntBinaryTree::displayPreOrder()

{

displayPreOrder(root);

}

void IntBinaryTree::displayPostOrder()

{

displayPostOrder(root);

}