Final Project for CSC 122 001 Computer Science II

Chapter 19 Programming Challenges 1 – 10

From *Starting Out with C++, Early Objects*, 9e by Gaddis, Tony

Spring 2018
Submitted by Julius Ranoa

Table of Contents[1]

[1] Programming Challenge 8 Employee Tree had already been submitted as a previous homework assignment and thus is not included in this submission.

Source code can also be found at
https://github.com/JasonRanoa/SP18-CSC-122

Project 1 Simple Binary Search Tree Class

Objective: Write a class for implementing a simple binary search tree capable of storing numbers. The class should have member functions:
Programming Challenges 1 – 7 are compiled into one project file. The programming challenge the listed method belong to are indicated by a superscript at the end. (e.g. [PC1] for the first programming challenge.)

- An insert method that accepts an integer. This method should not use recursion directly or indirectly by calling a recursive function. [PC1]
- A search method that accepts an integer. The search function should work by calling a private recursive member function. [PC1]
- An inorder method that accepts a reference to an initially filled vector and fills it with the inorder list of numbers stored in the binary search tree. [PC1]
- A size method that returns the number of items (nodes) stored in the tree. [PC2]
- A *leafCount* member function that counts and returns the number of leaf nodes in the tree. Nodes with no children are considered leaf nodes. [PC3]
- A height method that computes and returns the height of the tree. The height of the tree is the number of levels it contains. [PC4]
- A width method that computes the width of the tree. The width of the tree is that largest number of nodes at the same level. [PC5]
- A tree copy constructor. [PC6]
- An overloaded assignment operator for the tree. [PC7]

All methods are invoked by a driver program to demonstrate correctness.

Implementation Overview:
A class named *BinaryTree* contains all the methods required by the challenges. A default constructor and destructor are also added for the clean-up after dynamic allocation.

This documentation will first present the source code for the header and implementation files. In the header file, the programming challenge each public member method fulfills is noted in a comment next to the method declaration. In the implementation file, the method definitions are sorted according to the which programming challenge it is used (either directly or as a helper function).

Files included[2]**:** (1) BinaryTree.h, (2) BinaryTree.cpp

[2] The driver program, main.cpp, will be included later. Since there needs to be several demonstrations, multiple versions of the driver program are included.

Source Code for BinaryTree.h

```
#ifndef CH19_PR1_7_SIMPLE_BINARY_SEARCH_TREE_CLASS_BINARYTREE_H
#define CH19_PR1_7_SIMPLE_BINARY_SEARCH_TREE_CLASS_BINARYTREE_H

#include <vector>

class BinaryTree {

private:
    struct TreeNode {
        double value;
        TreeNode * left;
        TreeNode * right;

        TreeNode(
            double argVal,
            TreeNode * argLeft = nullptr,
            TreeNode * argRight = nullptr
        ) {
            value = argVal;
            left = argLeft;
            right = argRight;
        }
    };

    TreeNode * root;

    // Private Methods
    void destroySubTree(TreeNode *); // Cleans up dynamic memory allocation
    bool search(double x, TreeNode *);
    void attachinorder(std::vector<double> &, TreeNode *);
    void countNodes(int &, TreeNode *);
    void countLeaves(int &, TreeNode *);
    TreeNode * duplicateNode(const TreeNode *);

public:
    BinaryTree() {
        root = nullptr;
    }
    ~BinaryTree() {
        destroySubTree(root);
    }
    void insert(double x); // Programming Challenge 1A.
    bool search(double x); // Programming Challenge 1B.
    void inorder(std::vector<double> &); // Programming Challenge 1C.
    int size(); // Programming Challenge 2
    int leafCount(); // Programming Challenge 3
    int height();// Programming Challenge 4
    int width(); // Programming Challenge 5
    // Programming Challenge 6 Tree Copy Constructor
    BinaryTree(const BinaryTree &);
    // Programming Challenge 7 Tree Assignment Constructor
    BinaryTree & operator=(const BinaryTree &);

};
```

Source Code for BinaryTree.cpp

```
#include <iostream>
#include "BinaryTree.h"

// CLEANING UP. Destroying subtrees.

void BinaryTree::destroySubTree(TreeNode * tree) {
    if (!tree) return;
    destroySubTree(tree->left);
    destroySubTree(tree->right);
    delete tree;
}

/*
 *    Programming Challenge 1A
 *    An insert function that does not use recursion,
 *    directly or indirectly by calling a recursive function.
 *
 */

void BinaryTree::insert(double x) {
    /*
     *     This implementation makes use to pointers to pointers,
     *     indicated by the **. This works but there's another way of
     *     doing it that's much easier.

    TreeNode ** ptrToNodePtr = &root;

    while (*ptrToNodePtr) {
        if ((**ptrToNodePtr).value == x) {
            return; // Do nothing.
        } else if (x < (**ptrToNodePtr).value) {
            ptrToNodePtr = &((**ptrToNodePtr).left);
        } else {
            ptrToNodePtr = &((**ptrToNodePtr).right);
        }
    }

    *ptrToNodePtr = new TreeNode(x);

     */

    // If tree node is empty...
    TreeNode * newNode = new TreeNode(x);
    if (!root) {
        root = newNode;
    } else {
        // If the tree node is not empty, we can just append the new item by
        // changing the value of the left/right pointer.
        TreeNode *currentNode, *oneAfter;
        currentNode = oneAfter = root;
        // We have two variables here: (1) currentNode, and
        // (2) oneAfter -- which is just used to test for nullptr (i.e. insertion points)
        while (currentNode->value != x && oneAfter != nullptr) {
            currentNode = oneAfter;
            if (x < currentNode->value) {
                oneAfter = currentNode->left;
            } else {
                oneAfter = currentNode->right;
            }
        }

        if (currentNode->value == x) return;
        else if (x < currentNode->value) {
            currentNode->left = newNode;
        } else {
            currentNode->right = newNode;
        }
        // Do note that we can't assign newNode to oneAfter
        // since doing so wouldn't affect the node on the tree.
    }

    return;
}

/*
 *    Programming Challenge 1B
 *    Create a search function that works by calling a private
 *    recursive function.
 *
 */

// Public Member Function
bool BinaryTree::search(double x) {
    return search(x, root);
}

// Private Member Function -- Recursive
bool BinaryTree::search(double x, TreeNode * tree) {
    if (!tree) {
        return false; // Found end of tree but no match.
    } else if (tree->value == x) {
        return true;
    } else if (x < tree->value) {
        return search(x, tree->left);
    } else {
        return search(x, tree->right);
    }
}

/*
 *     Programming Challenge 1C
 *     Create an in-order function that accepts an empty vector and fills
 *     it with the in-order list of numbers in the tree.
 *
 */

void BinaryTree::inorder(std::vector<double> & v) {
    attachinorder(v, root);
}

void BinaryTree::attachinorder(std::vector<double> & v, TreeNode * tree) {
    if (tree) {
        attachinorder(v, tree->left);
        v.push_back(tree->value);
        attachinorder(v, tree->right);
    }
}

/*
 *    Programming Challenge 2
 *    Get the size of the tree.
 *
 */

void BinaryTree::countNodes(int & count, TreeNode * tree) {
    if (!tree) return;
    else {
        count++;
        countNodes(count, tree->left);
        countNodes(count, tree->right);
    }
}

int BinaryTree::size() {
    int count = 0;
    countNodes(count, root);
    return count;
}

/*
 *    Programming Challenge 3
 *    Count the number of leaf nodes (i.e. nodes with no children) on the
 *    tree. In this implementation, two methods are used -- A public method
 *    and a private one for recursion.
 *
 */

void BinaryTree::countLeaves(int & leafCount, TreeNode * tree) {
    if (!tree) return;
    else if (!tree->left && !tree->right) {
        leafCount++;
    } else {
        countLeaves(leafCount, tree->left);
        countLeaves(leafCount, tree->right);
    }
}

int BinaryTree::leafCount() {
    int nLeaves = 0;
    countLeaves(nLeaves, root);
    return nLeaves;
}

/*
 *    Programming Challenge 4
 *    The height of tree is the number of levels it contains.
 *
 */
```

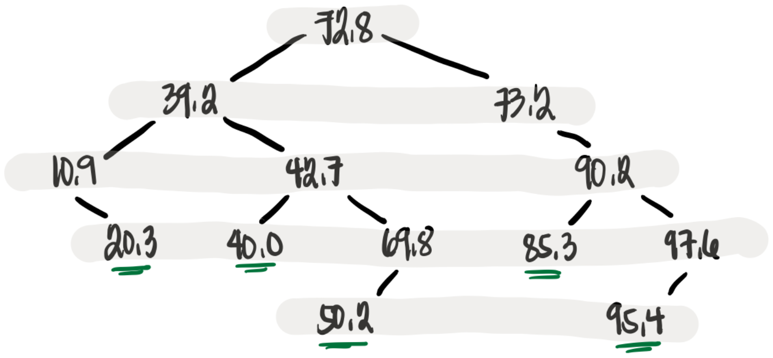
```
int BinaryTree::height() {
    std::vector<TreeNode *> currentLevel;
    std::vector<TreeNode *> nextLevel;
    int height = 0;

    // Start the count.
    if (root) currentLevel.push_back(root);

    // Children nodes.
    while (currentLevel.size() > 0) {
        height++;
        for (TreeNode * tree : currentLevel) {
            if (tree->left) nextLevel.push_back(tree->left);
            if (tree->right) nextLevel.push_back(tree->right);
        }
        currentLevel = nextLevel;
        nextLevel.clear();
    }

    return height;
}

/*
 *   Programming Challenge 5
 *   The width of a tree is the largest number of nodes at the same level.
 *
 */

int BinaryTree::width() {
    std::vector<TreeNode *> currentLevel;
    std::vector<TreeNode *> nextLevel;
    int width = 0;

    // Start the count.
    if (root) currentLevel.push_back(root);

    // Children nodes.
    while (currentLevel.size() > 0) {
        width = ( width > currentLevel.size() ) ? width : currentLevel.size();
        for (TreeNode * tree : currentLevel) {
            if (tree->left) nextLevel.push_back(tree->left);
            if (tree->right) nextLevel.push_back(tree->right);
        }
        currentLevel = nextLevel;
        nextLevel.clear();
    }

    return width;
}

/*
 *   Programming Challenge 6
 *   Implement a tree copy constructor.
 *
 */
```

```
BinaryTree::BinaryTree(const BinaryTree & copy) {
    root = duplicateNode(copy.root);
}

BinaryTree::TreeNode * BinaryTree::duplicateNode(const TreeNode * source) {
    if (!source) return nullptr;
    return new TreeNode(
        source->value,
        ( source->left ) ? duplicateNode(source->left) : nullptr,
        ( source->right ) ? duplicateNode(source->right) : nullptr
    );
}

/*
 *    Programming Challenge 7
 *    Implement an overloaded copy constructor.
 *
 */

BinaryTree & BinaryTree::operator=(const BinaryTree & source) {
    destroySubTree(root); // Just to cover my bases.
    root = duplicateNode(source.root);
    return *this;
}
```

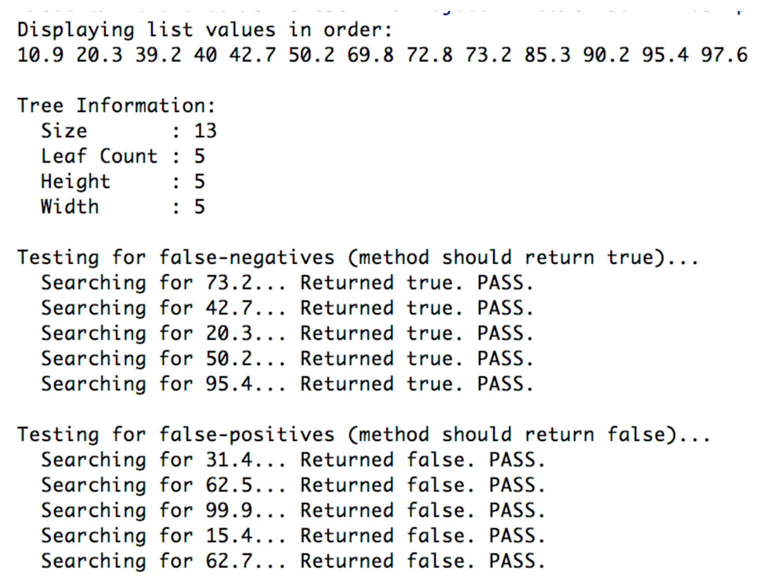
Driver Programs and Demonstrations

(1) Feeding the tree with test data, searching with the tree, displaying the values in order and present tree information (i.e. size, height, width, leaf count)

The following sample data is fed to the tree:



Screenshot of runtime:



```
Displaying list values in order:
10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6

Tree Information:
  Size       : 13
  Leaf Count : 5
  Height     : 5
  Width      : 5

Testing for false-negatives (method should return true)...
  Searching for 73.2... Returned true. PASS.
  Searching for 42.7... Returned true. PASS.
  Searching for 20.3... Returned true. PASS.
  Searching for 50.2... Returned true. PASS.
  Searching for 95.4... Returned true. PASS.

Testing for false-positives (method should return false)...
  Searching for 31.4... Returned false. PASS.
  Searching for 62.5... Returned false. PASS.
  Searching for 99.9... Returned false. PASS.
  Searching for 15.4... Returned false. PASS.
  Searching for 62.7... Returned false. PASS.
```

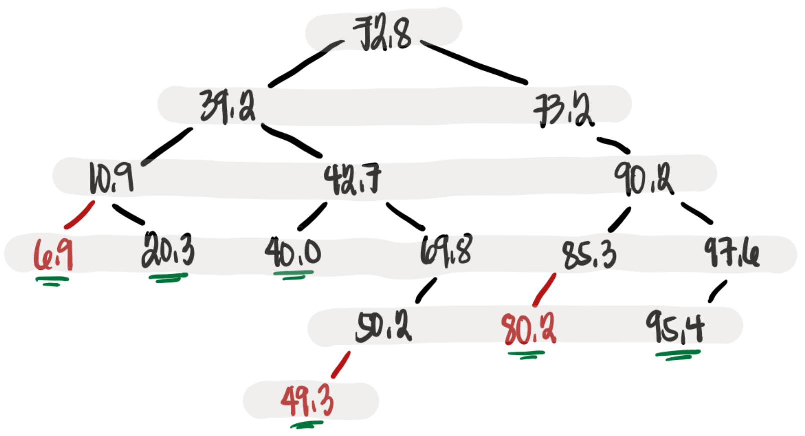
Source Code for main.cpp

```
#include <iostream>
#include <sstream>
#include <vector>
#include "BinaryTree.h"

void enterTestData(BinaryTree &);

int main() {
    BinaryTree treeOne;
    enterTestData(treeOne);
```

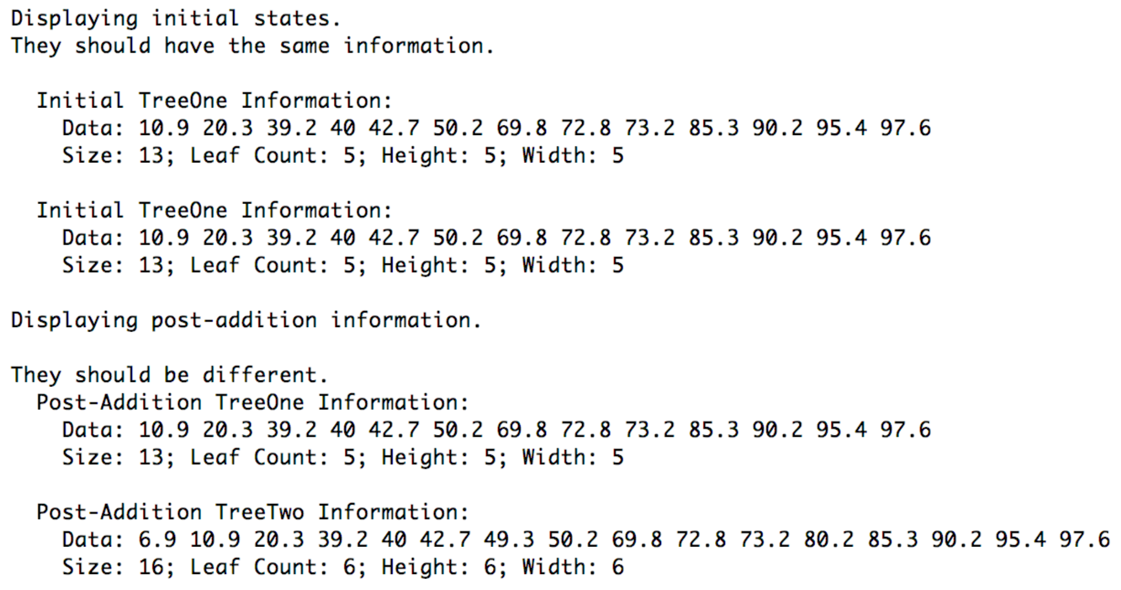
```
        std::vector<double> numList;
        treeOne.inorder(numList);

        // Display list
        std::cout << "Displaying list values in order: \n";
        for (double d : numList) {
            std::cout << d << " ";
        }
        std::cout << "\n\n";

        std::cout << "Tree Information: \n";
        std::cout << "  Size       : " << treeOne.size() << "\n"; // should be 13
        std::cout << "  Leaf Count : " << treeOne.leafCount() << "\n"; // should be 5
        std::cout << "  Height     : " << treeOne.height() << "\n"; // should be 5
        std::cout << "  Width      : " << treeOne.width() << "\n"; // should be 5
        std::cout << "\n";

        // Testing search functionality
        double positiveTests[] = { 73.2, 42.7, 20.3, 50.2, 95.4 };
        std::cout << "Testing for false-negatives (method should return true)... \n";
        for (double test : positiveTests) {
            std::cout << "  Searching for " << test << "... ";
            if (treeOne.search(test)) {
                std::cout << "Returned true. PASS. \n";
            } else {
                std::cout << "Returned false. FAIL. \n";
            }
        }
        std::cout << "\n";

        double negativeTests[] = { 31.4, 62.5, 99.9, 15.4, 62.7 };
        std::cout << "Testing for false-positives (method should return false)... \n";
        for (double test : negativeTests) {
            std::cout << "  Searching for " << test << "... ";
            if (treeOne.search(test)) {
                std::cout << "Returned true. FAIL. \n";
            } else {
                std::cout << "Returned false. PASS. \n";
            }
        }

        return 0;
    }

    void enterTestData(BinaryTree & bin) {
        std::string testData =
            "72.8 39.2 10.9 20.3 42.7 "
            "40.0 69.8 50.2 73.2 90.2 "
            "85.3 97.6 95.4";
        std::stringstream ss(testData);
        double temp;
        while (ss >> temp) {
            bin.insert(temp);
        }
        return;
    }
```

(2) Employing the copy constructor and testing for actual duplication of data, not just of pointers.

The same initial tree is fed to the program. A copy of the tree will be made and additional copies would be inserted. The second tree should have this structure (additional nodes in red):



Screenshot of Runtime:



```
Displaying initial states.
They should have the same information.

  Initial TreeOne Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

  Initial TreeOne Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

Displaying post-addition information.

They should be different.
  Post-Addition TreeOne Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

  Post-Addition TreeTwo Information:
    Data: 6.9 10.9 20.3 39.2 40 42.7 49.3 50.2 69.8 72.8 73.2 80.2 85.3 90.2 95.4 97.6
    Size: 16; Leaf Count: 6; Height: 6; Width: 6
```

Source Code for main.cpp

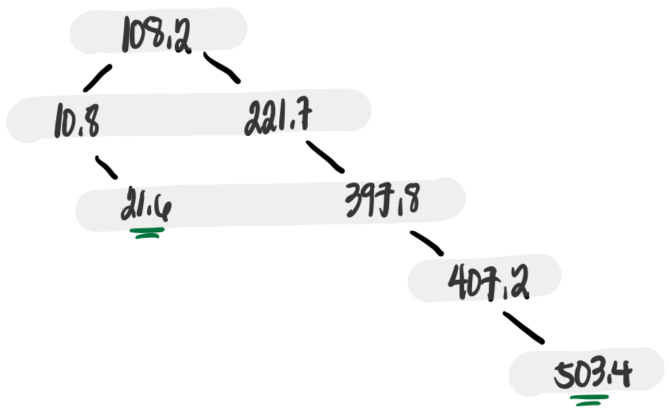
```cpp
#include <iostream>
#include <sstream>
#include <vector>
#include "BinaryTree.h"

void enterTestData(BinaryTree &);
void displayTreeInfo(BinaryTree * const, std::string);
```

```
int main() {
    BinaryTree treeOne;
    enterTestData(treeOne);

    std::vector<double> numList;
    treeOne.inorder(numList);
    BinaryTree treeTwo = treeOne; // Uses the copy constructor

    std::cout << "Displaying initial states. \n"
                 "They should have the same information. \n\n";
    displayTreeInfo(&treeOne, "Initial TreeOne");
    displayTreeInfo(&treeTwo, "Initial TreeOne");

    double test[] = { 6.9, 80.2, 49.3 };
    for (double d : test) {
        treeTwo.insert(d);
    }

    std::cout << "Displaying post-addition information. \n\n"
                 "They should be different. \n";
    displayTreeInfo(&treeOne, "Post-Addition TreeOne");
    displayTreeInfo(&treeTwo, "Post-Addition TreeTwo");

    return 0;
}

void enterTestData(BinaryTree & bin) {
    std::string testData =
        "72.8 39.2 10.9 20.3 42.7 "
        "40.0 69.8 50.2 73.2 90.2 "
        "85.3 97.6 95.4";

    std::stringstream ss(testData);

    double temp;
    while (ss >> temp) {
        bin.insert(temp);
    }
}

void displayTreeInfo(BinaryTree * const bin, std::string name) {
    std::cout << "   " << name << " Information: \n";
    std::vector<double> dList;
    bin->inorder(dList);
    std::cout << "    Data: ";
    for (double d : dList) {
        std::cout << d << " ";
    }
    std::cout << "\n";
    std::cout << "    Size: " << bin->size() << "; ";
    std::cout << "Leaf Count: " << bin->leafCount() << "; ";
    std::cout << "Height: " << bin->height() << "; ";
    std::cout << "Width: " << bin->width() << "\n";
    std::cout << "\n";
}
```

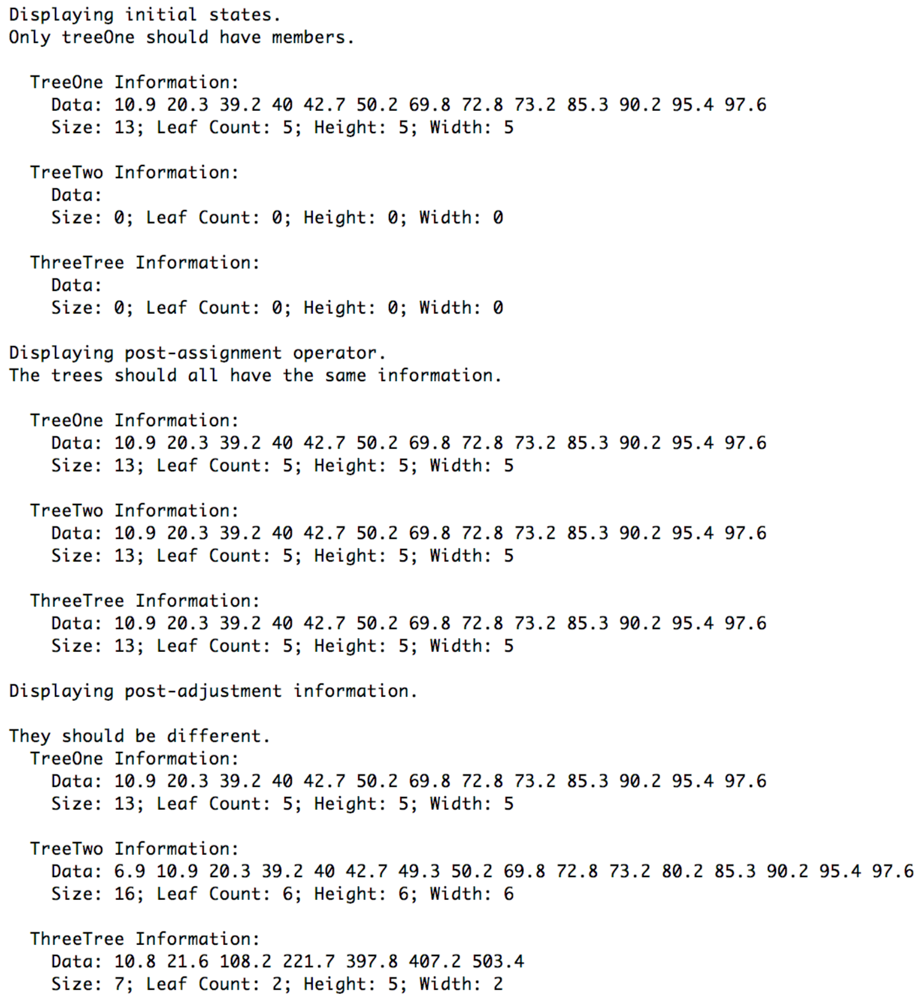
(3) Using the overloaded assignment operator and testing for actual duplication of nodes, not pointers.

The third demonstration is an alteration of the second demonstration, employing assignment operators to duplicate trees rather than copy constructors.

A third binary tree, the structure illustrated below, will be used as test data:



Screenshot of runtime.



```
Displaying initial states.
Only treeOne should have members.

  TreeOne Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

  TreeTwo Information:
    Data:
    Size: 0; Leaf Count: 0; Height: 0; Width: 0

  ThreeTree Information:
    Data:
    Size: 0; Leaf Count: 0; Height: 0; Width: 0

Displaying post-assignment operator.
The trees should all have the same information.

  TreeOne Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

  TreeTwo Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

  ThreeTree Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

Displaying post-adjustment information.

They should be different.
  TreeOne Information:
    Data: 10.9 20.3 39.2 40 42.7 50.2 69.8 72.8 73.2 85.3 90.2 95.4 97.6
    Size: 13; Leaf Count: 5; Height: 5; Width: 5

  TreeTwo Information:
    Data: 6.9 10.9 20.3 39.2 40 42.7 49.3 50.2 69.8 72.8 73.2 80.2 85.3 90.2 95.4 97.6
    Size: 16; Leaf Count: 6; Height: 6; Width: 6

  ThreeTree Information:
    Data: 10.8 21.6 108.2 221.7 397.8 407.2 503.4
    Size: 7; Leaf Count: 2; Height: 5; Width: 2
```

Source code for main.cpp

```cpp
#include <iostream>
#include <sstream>
#include <vector>
#include "BinaryTree.h"

void enterTestData(BinaryTree &);
void displayTreeInfo(BinaryTree * const, std::string);

int main() {
    BinaryTree treeOne, treeTwo, treeThree;
    enterTestData(treeOne);

    std::vector<double> numList;
    treeOne.inorder(numList);

    std::cout << "Displaying initial states. \n"
                 "Only treeOne should have members. \n\n";
    displayTreeInfo(&treeOne, "TreeOne");
    displayTreeInfo(&treeTwo, "TreeTwo");
    displayTreeInfo(&treeThree, "ThreeTree");

    treeThree = treeTwo = treeOne; // this uses an assignment operator.

    std::cout << "Displaying post-assignment operator. \n"
                 "The trees should all have the same information. \n\n";
    displayTreeInfo(&treeOne, "TreeOne");
    displayTreeInfo(&treeTwo, "TreeTwo");
    displayTreeInfo(&treeThree, "ThreeTree");

    // Adding stuff to treeTwo.
    double testTwo[] = { 6.9, 80.2, 49.3 };
    for (double d : testTwo) {
        treeTwo.insert(d);
    }

    double testThree[] = { 108.2, 221.7, 397.8, 407.2, 503.4, 10.8, 21.6 };
    treeThree = BinaryTree(); // Resetting treeTree
    for (double d : testThree) {
        treeThree.insert(d);
    }

    std::cout << "Displaying post-adjustment information. \n\n"
                 "They should be different. \n";
    displayTreeInfo(&treeOne, "TreeOne");
    displayTreeInfo(&treeTwo, "TreeTwo");
    displayTreeInfo(&treeThree, "ThreeTree");

    return 0;
}

void enterTestData(BinaryTree & bin) {
    std::string testData =
        "72.8 39.2 10.9 20.3 42.7 "
```

```
            "40.0 69.8 50.2 73.2 90.2 "
            "85.3 97.6 95.4";

    std::stringstream ss(testData);

    double temp;
    while (ss >> temp) {
        bin.insert(temp);
    }
}

void displayTreeInfo(BinaryTree * const bin, std::string name) {
    std::cout << "  " << name << " Information: \n";
    std::vector<double> dList;
    bin->inorder(dList);
    std::cout << "    Data: ";
    for (double d : dList) {
        std::cout << d << " ";
    }
    std::cout << "\n";
    std::cout << "    Size: " << bin->size() << "; ";
    std::cout << "Leaf Count: " << bin->leafCount() << "; ";
    std::cout << "Height: " << bin->height() << "; ";
    std::cout << "Width: " << bin->width() << "\n";
    std::cout << "\n";
}
```

Project 2 Cousins

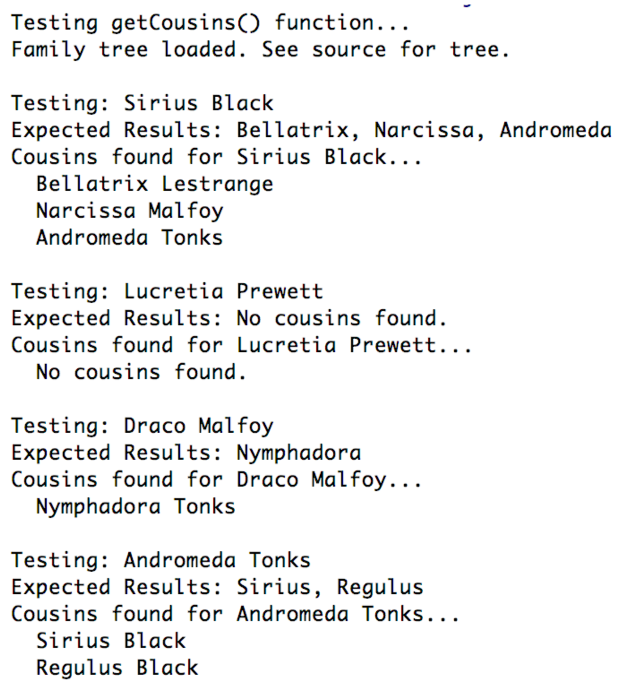
Objective: Building on Program 19-5, write a function that takes a pointer to a *Person* object and produces a list of that person's cousins.

Implementation Notes:
In the project folder, the source code for the *Person* class is included since it's needed to run the project. Do note that the source code is unmodified and all the work for this project is done in the driver program (i.e. main.cpp). For the sake of completeness, the exported source code will be included as well.

The function with this signature – `std::vector<Person *> getCousins(Person *);`
contains all the logic needed to extract the cousins from a given Person object, given that relationships have already been defined. Cousins are defined as the children of each parent's siblings.

The test logic is also already included in the source code. See code comments for more details.

Screenshot of Runtime:



```
Testing getCousins() function...
Family tree loaded. See source for tree.

Testing: Sirius Black
Expected Results: Bellatrix, Narcissa, Andromeda
Cousins found for Sirius Black...
  Bellatrix Lestrange
  Narcissa Malfoy
  Andromeda Tonks

Testing: Lucretia Prewett
Expected Results: No cousins found.
Cousins found for Lucretia Prewett...
  No cousins found.

Testing: Draco Malfoy
Expected Results: Nymphadora
Cousins found for Draco Malfoy...
  Nymphadora Tonks

Testing: Andromeda Tonks
Expected Results: Sirius, Regulus
Cousins found for Andromeda Tonks...
  Sirius Black
  Regulus Black
```

Files included: (1) main.cpp, (2) Person.h[3], (3) Person.cpp[3]

[3] Taken from *Starting Out with C++, Early Objects 9e*, Gaddis

Source Code for main.cpp

```
#include <iostream>
#include <set>
#include "Person.h"

/*
 *    Programming Challenge 9
 *    Building on Program 19-5, write a function that
 *    takes a pointer to a Person object and produces a list of
 *    that Person's cousins.
 *
 *    Implementation:
 *    The Person class from Program 19-5 is unmodified.
 *    The getCousins() function is implemented in main.cpp.
 *
 */


std::vector<Person *> getCousins(Person *);

std::vector<Person *> produceTestData();
void doTesting(Person *, std::string);

int main() {
    // Test Data
    std::vector<Person *> test = produceTestData();

    std::cout << "Testing getCousins() function... \n";
    std::cout << "Family tree loaded. See source for tree. \n\n";

    doTesting(test[5], "Bellatrix, Narcissa, Andromeda"); // Sirius
    doTesting(test[2], "No cousins found."); // Lucretia
    doTesting(test[14], "Nymphadora"); // Draco
    doTesting(test[13], "Sirius, Regulus"); // Andromeda

    // Clear test data.
    for (Person * t : test) {
        delete t;
    }
    return 0;
}

std::vector<Person *> getCousins(Person * person) {
    if (!person) return std::vector<Person *>();
    // Return empty vector.

    // NOTE: Sets are used to avoid name duplication.
    // The insert function does nothing if key is already there.
    // Get the list of parents for child.
    std::set<Person *> parents;
    for (int i = 0; i < person->getNumParents(); i++) {
        parents.insert(person->getParent(i));
    }

    // Get the list of grandparents.
    std::set<Person *> grandparents;
    for (Person * parent : parents) {
        for (int i = 0; i < parent->getNumParents(); ++i) {
            grandparents.insert(parent->getParent(i));
        }
    }

    // Get the list of children of the grandparents
    std::set<Person *> parentSiblings;
    for (Person * grandparent : grandparents) {
        for (int i = 0; i < grandparent->getNumChildren(); ++i) {
            parentSiblings.insert(grandparent->getChild(i));
        }
    }
    // Remove the current parents to get list of parent siblings
    for (Person * parent : parents) {
        parentSiblings.erase(parent);
    }

    // Now, get the children of the p-siblings to get cousins
    std::set<Person *> cousinSet; // To avoid duplication.
    for (Person * parentSibling : parentSiblings) {
        for (int i = 0; i < parentSibling->getNumChildren(); i++) {
            cousinSet.insert(parentSibling->getChild(i));
        }
    }

    return std::vector<Person *>(cousinSet.begin(), cousinSet.end());
}

std::vector<Person *> produceTestData() {
    /*
     *    Black Family Tree (Test Data)
     *    Arcturus --- Melania    Pollux --- Irma
     *              |                     |
     *          ---------           -----------
     *         |        |          |          |
     *    Lucretia    Orion --- Walburga   Cygnus --- Druella
     *                       |                     |
     *                  ---------         -------------------------
     *                 |        |        |            |           |
     *              Sirius   Regulus  Bellatrix   Narcissa   Andromeda
     *                                               |           |
     *                                             Draco    Nymphadora
     *
     */
    std::vector<Person *> people = {
        new Person( "Arcturus Black III", male ), // 0
        new Person( "Melania Macmillan", male ), // 1
        new Person( "Lucretia Prewett", female ), // 2

        new Person( "Orion Black", male ), // 3
        new Person( "Walburga Black", female ), // 4
        new Person( "Sirius Black", male ), // 5
        new Person( "Regulus Black", male ), // 6

        new Person( "Pollux Black", male ), // 7
        new Person( "Irma Crabbe", female ), // 8

        new Person( "Cygnus Black", male ), // 9
        new Person( "Druella Rosier", female ), // 10

        new Person( "Bellatrix Lestrange", female ), // 11
        new Person( "Narcissa Malfoy", female ), // 12
        new Person( "Andromeda Tonks", female ), // 13

        new Person( "Draco Malfoy", male ), // 14
        new Person( "Nymphadora Tonks", female ), // 15
    };

    // Establish relationships.
    people[0]->addChild(people[2]);
    people[1]->addChild(people[2]);

    people[3]->addChild(people[5]); people[4]->addChild(people[5]);
    people[3]->addChild(people[6]); people[4]->addChild(people[6]);

    people[7]->addChild(people[4]); people[7]->addChild(people[4]);
    people[7]->addChild(people[9]); people[7]->addChild(people[9]);

    people[9]->addChild(people[11]); people[10]->addChild(people[11]);
    people[9]->addChild(people[12]); people[10]->addChild(people[12]);
    people[9]->addChild(people[13]); people[10]->addChild(people[13]);

    people[12]->addChild(people[14]);
    people[13]->addChild(people[15]);

    return people;
}

void doTesting(Person * test, std::string expectedResults) {
    std::vector<Person *> testResults;

    std::cout << "Testing: " << test->getName() << "\n"
              << "Expected Results: " << expectedResults << " \n";
    std::cout << "Cousins found for " << test->getName() << "... \n";
    testResults = getCousins(test);
    if (testResults.size() == 0) {
        std::cout << "  No cousins found. \n";
    } else {
        for (Person * p : testResults) {
            std::cout << "  " << p->getName() << "\n";
        }
    }

    std::cout << "\n";
}
```

Source Code for Person.h

```
#ifndef QUIZ9_PR19_5_DOCUMENTATION_SOURCEFILE_PERSON_H
#define QUIZ9_PR19_5_DOCUMENTATION_SOURCEFILE_PERSON_H

#include <vector>
#include <string>
#include <iostream>
#include <cstdlib>
using namespace std;

enum Gender{male, female};

// Person class represents a person participating in a genealogy.
class Person
{
    string name;
    Gender gender;
    vector<Person *> parents;
    vector<Person *> children;
    void addParent(Person *p){ parents.push_back(p); }
public:
    Person (string name, Gender g)
    {
        this->name = name;
        gender = g;
    }
    Person *addChild(string name, Gender g);
    Person *addChild(Person *p);

    friend ostream &operator << (ostream &out, Person p);

    // Member functions for getting various Person info
    string getName() const{ return name; };
    Gender getGender() const{ return gender; };
    int getNumChildren() const{ return children.size(); }
    int getNumParents() const{ return parents.size(); }
    Person *getChild(int k) const ;
    Person *getParent(int k) const;

};

#endif //QUIZ9_PR19_5_DOCUMENTATION_SOURCEFILE_PERSON_H
```

Source Code for Person.cpp

```
#include "Person.h"

//*************************************************************
// Create a child with specified name and gender, and        *
// set one of the parents to be this person.                  *
// Add the new child to the list of childfen for this person *
//*************************************************************
Person *Person::addChild(string name, Gender g)
```

```
{
    Person *child = new Person(name, g);
    child->addParent(this);     // I am a parent of this child
    children.push_back(child);  // This is one of my children
    return child;
}

//************************************************************
// Add a child to the list of children for this person       *
//************************************************************
Person *Person::addChild(Person* child)
{
    child->addParent(this);    // I am a parent of this child
    children.push_back(child); // This is one of my children
    return child;
}

//*********************************************************
// Return a pointer to the specified parent               *
//*********************************************************
Person *Person::getParent(int k) const
{
    if (k < 0 || k >= parents.size())
    {
        cout << "Error indexing parents vector." << endl;
        exit(1);
    }
    return parents[k];
}

//**********************************************************
// Return a pointer to a specified child                   *
//**********************************************************
Person *Person::getChild(int k) const
{
    if (k < 0 || k >= children.size())
    {
        cout << "Error indexing children's vector." << endl;
        exit(1);
    }
    return children[k];
}

//****************************************************
// Overloaded stream output operator                 *
//****************************************************
ostream & operator<<(ostream & out, Person p)
{
    out << "<person name = " << p.name << ">" << '\n';
    if (p.parents.size() > 0)
        out << "   <parents>" << ' ';
    for (int k = 0; k < p.parents.size(); k++)
    {
        out << " " << p.parents[k]->name << ' ';
    }
    if (p.parents.size() > 0)
        out << " </parents>" << "\n";
    if (p.children.size() > 0)
        out << "    <children>" << ' ';
    for (int k = 0; k < p.children.size(); k++)
    {
        out << " " << p.children[k]->name << ' ';
    }
    if (p.children.size() > 0)
        out << " </children>" << "\n";
    out << "</person>" << "\n";
    return out;
}
```

Project 3 Prefix Representation of Binary Trees

Objective: Write a method that prints the prefix representation of binary trees.[4]

[4] See code comments in the main.cpp file for more information.

Implementation Notes:

The project asked to modify the main function of Program 19-4 but the provided class – *IntBinaryTree* – doesn't have a method that allows an external function access to the information needed to fulfill this challenge. Solving this issue would require modifying the source for the *IntBinaryTree* class so I've decided to create a method of the *IntBinaryTree* to print the prefix representation.

Most of the code in the *IntBinaryTree* class remains unchanged except for the last few lines. In the header file, a private method named *treePrint()* with 2 arguments and a public method also named *treePrint()* but without any arguments are included. The latter is defined inline but the definition of the first method is in the last few lines of the implementation file. Comments are added to indicate this.

For brevity, snippets of the declaration and definition of the mentioned functions are included below:

From IntBinaryTree.h

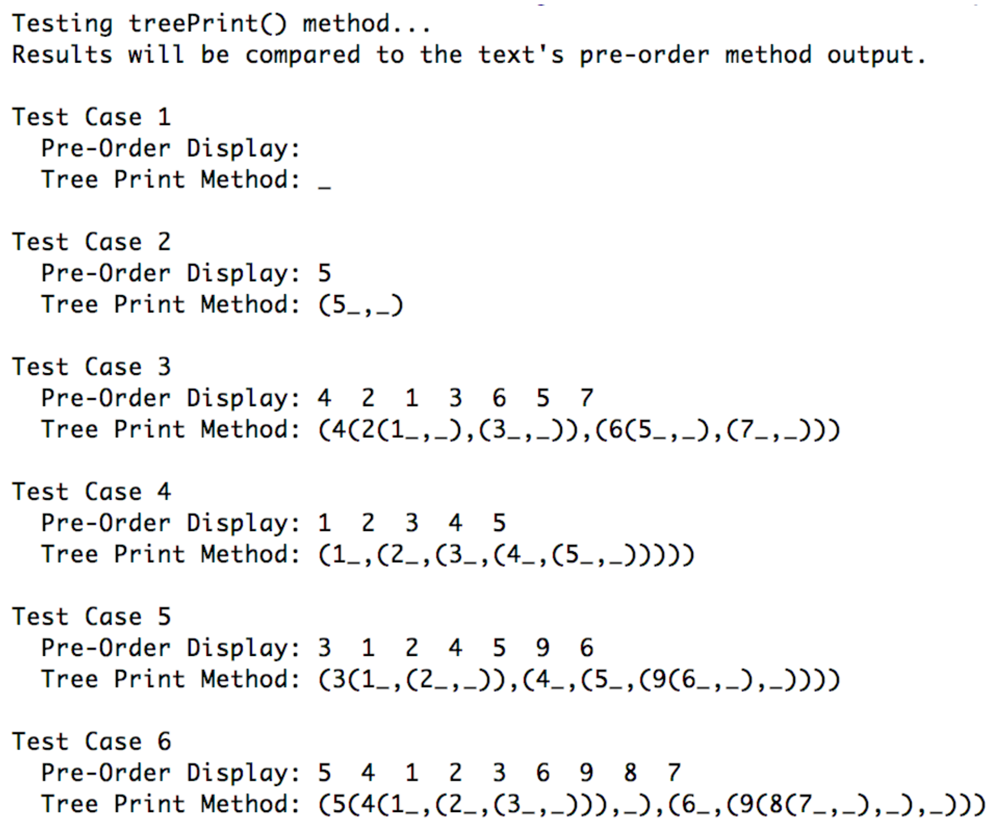
```
private:
    void treePrint(TreeNode *, std::ostream &);

public:
    void treePrint() { treePrint(root, std::cout); };
```

From IntBinaryTree.cpp

```
void IntBinaryTree::treePrint(IntBinaryTree::TreeNode * tree, std::ostream & out) {
    if (!tree) out << "_";
    else {
        out << "(" << tree->value;
        treePrint(tree->left, out);
        out << ",";
        treePrint(tree->right, out);
        out << ")";
    }
}
```

There are six test cases for the demonstration. All are defined in the source code below.

Screenshot of Runtime:



```
Testing treePrint() method...
Results will be compared to the text's pre-order method output.

Test Case 1
  Pre-Order Display:
  Tree Print Method: _

Test Case 2
  Pre-Order Display: 5
  Tree Print Method: (5_,_)

Test Case 3
  Pre-Order Display: 4  2  1  3  6  5  7
  Tree Print Method: (4(2(1_,_),(3_,_)),(6(5_,_),(7_,_)))

Test Case 4
  Pre-Order Display: 1  2  3  4  5
  Tree Print Method: (1_,(2_,(3_,(4_,(5_,_)))))

Test Case 5
  Pre-Order Display: 3  1  2  4  5  9  6
  Tree Print Method: (3(1_,(2_,_)),(4_,(5_,(9(6_,_),_))))

Test Case 6
  Pre-Order Display: 5  4  1  2  3  6  9  8  7
  Tree Print Method: (5(4(1_,(2_,(3_,_))),_),(6_,(9(8(7_,_),_),_)))
```

Files Included: (1) main.cpp, (2) IntBinaryTree.h[5], (3) IntBinaryTree.cpp[5]

[5] Most of the source code taken from *Starting Out with C++, Early Objects 9e*, Gaddis, as part of the challenge requirement.

Source Code for main.cpp

```cpp
#include <iostream>
#include <vector>
#include "IntBinaryTree.h"

/*
 *    Programming Challenge 10
 *    Prefix Representation of Binary Trees
 *
 *    Rules:
 *    1. The prefix representation of an empty binary tree is a single underscore.
 *    2. The prefix presentation of a non-empty binary tree is (v L, R),
 *       where v represents the value stored in root and
 *             L and R are the prefix representations of the left and right subtrees.
 *
 *    Modify the binary tree class of Program 19-1 to add the ff. member functions.
 *    1. void treePrint():
 *       This public member function will print the prefix representation of a
 *       binary tree object to standard output.
 *    2. void treePrint(TreeNode * root, ostream& out) const:
 *       This private member function will print the prefix representation of the
 *       binary tree with a given root to a given output stream.
 *
 */

int main() {
    // Loading test data
    std::vector<std::vector<int>> testPool = {
        { }, // intentionally kept empty
        { 5 }, // (5_,_)
        { 4, 2, 6, 1, 3, 5, 7 }, // 4 leaf nodes, 3 levels.
        { 1, 2, 3, 4, 5 }, // Only one leaf node.
        { 3, 1, 4, 5, 9, 2, 6 }, // First seven digits of pi
        { 5, 4, 6, 1, 2, 3, 9, 8, 7 } // Tree forms a diamond.
    };

    std::cout << "Testing treePrint() method... \n";
    std::cout << "Results will be compared to the text's pre-order method output. \n\n";

    for (int i = 0; i < testPool.size(); ++i) {
        IntBinaryTree tree;
        for (int num : testPool[i]) { tree.insert(num); }

        std::cout << "Test Case " << (i + 1) << "\n";
        std::cout << "  Pre-Order Display: ";
        tree.showPreOrder();
        std::cout << "\n";
        std::cout << "  Tree Print Method: ";
        tree.treePrint();
        std::cout << "\n\n";
    }

    return 0;
}
```

Source Code for IntBinaryTree.h

```
class IntBinaryTree
{
private:
    // The TreeNode struct is used to build the tree.
    struct TreeNode
    {
        int value;
        TreeNode *left;
        TreeNode *right;
        TreeNode(int value1,
                 TreeNode *left1 = nullptr,
                 TreeNode *right1 = nullptr)
        {
            value = value1;
            left = left1;
            right = right1;
        }
    };

    TreeNode *root;      // Pointer to the root of the tree

    // Various helper member functions.
    void insert(TreeNode *&, int);
    void destroySubtree(TreeNode *);
    void remove(TreeNode *&, int);
    void makeDeletion(TreeNode *&);
    void displayInOrder(TreeNode *) const;
    void displayPreOrder(TreeNode *) const;
    void displayPostOrder(TreeNode *) const;

public:
    // These member functions are the public interface.
    IntBinaryTree()       // Constructor
    {
        root = nullptr;
    }
    ~IntBinaryTree()      // Destructor
    {
        destroySubtree(root);
    }
    void insert(int num)
    {
        insert(root, num);
    }
    bool search(int) const;
    void remove(int num)
    {
        remove(root, num);
    }
    void showInOrder(void) const
    {
        displayInOrder(root);
    }
    void showPreOrder() const
    {
        displayPreOrder(root);
    }
    void showPostOrder() const
    {
        displayPostOrder(root);
    }

// Programming Challenge 10 Method Declarations
private:
    void treePrint(TreeNode *, std::ostream &);

public:
    void treePrint() { treePrint(root, std::cout); };

};
```

Source Code for IntBinaryTree.cpp

```
#include <iostream>
#include "IntBinaryTree.h"
using namespace std;

//*************************************************
// This version of insert inserts a number into    *
// a given subtree of the main binary search tree. *
//*************************************************
void IntBinaryTree::insert(TreeNode * &tree, int num)
{
    // If the tree is empty, make a new node and make it
    // the root of the tree.
    if (!tree)
    {
        tree = new TreeNode(num);
        return;
    }

    // If num is already in tree: return.
    if (tree->value == num)
        return;

    // The tree is not empty: insert the new node into the
    // left or right subtree.
    if (num < tree->value)
        insert(tree->left, num);
    else
        insert(tree->right, num);
}

//**************************************************
// destroySubTree is called by the destructor. It   *
// deletes all nodes in the tree.                   *
//**************************************************
void IntBinaryTree::destroySubtree(TreeNode *tree)
```

```
{
    if (!tree) return;
    destroySubtree(tree->left);
    destroySubtree(tree->right);
    // Delete the node at the root.
    delete tree;
}

//***************************************************
// searchNode determines if a value is present in   *
// the tree. If so, the function returns true.      *
// Otherwise, it returns false.                     *
//***************************************************
bool IntBinaryTree::search(int num) const
{
    TreeNode *tree = root;

    while (tree)
    {
        if (tree->value == num)
            return true;
        else if (num < tree->value)
            tree = tree->left;
        else
            tree = tree->right;
    }
    return false;
}

//********************************************
// remove deletes the node in the given tree *
// that has a value member the same as num.  *
//********************************************
void IntBinaryTree::remove(TreeNode *&tree, int num)
{
    if (tree == nullptr) return;
    if (num < tree->value)
        remove(tree->left, num);
    else if (num > tree->value)
        remove(tree->right, num);
    else
        // We have found the node to delete.
        makeDeletion(tree);
}

//***********************************************************
// makeDeletion takes a reference to a tree whose root      *
// is to be deleted. If the tree has a single child, the    *
// the tree is replaced by the single child after the       *
// removal of its root node. If the tree has two children   *
// the left subtree of the deleted node is attached at      *
// an appropriate point in the right subtree, and then      *
// the right subtree replaces the original tree.            *
//***********************************************************
void IntBinaryTree::makeDeletion(TreeNode *&tree)
{
    // Used to hold node that will be deleted.
    TreeNode *nodeToDelete = tree;

    // Used to locate the  point where the
    // left subtree is attached.
    TreeNode *attachPoint;

    if (tree->right == nullptr)
    {
        // Replace tree with its left subtree.
        tree = tree->left;
    }
    else if (tree->left == nullptr)
    {
        // Replace tree with its right subtree.
        tree = tree->right;
    }
    else
        //The node has two children
    {
        // Move to right subtree.
        attachPoint = tree->right;

        // Locate the smallest node in the right subtree
        // by moving as far to the left as possible.
        while (attachPoint->left != nullptr)
            attachPoint = attachPoint->left;

        // Attach the left subtree of the original tree
        // as the left subtree of the smallest node
        // in the right subtree.
        attachPoint->left = tree->left;

        // Replace the original tree with its right subtree.
        tree = tree->right;
    }

    // Delete root of original tree
    delete nodeToDelete;
}

//***********************************************************
// This function displays the values  stored in a tree    *
// in inorder.                                             *
//***********************************************************
void IntBinaryTree::displayInOrder(TreeNode *tree) const
{
    if (tree)
    {
        displayInOrder(tree->left);
        cout << tree->value << "  ";
        displayInOrder(tree->right);
    }
}

//***********************************************************
// This function displays the values stored in a tree     *
// in inorder.                                             *
```

```
//*******************************************************
void IntBinaryTree::displayPreOrder(TreeNode *tree) const
{
    if (tree)
    {
        cout << tree->value << "  ";
        displayPreOrder(tree->left);
        displayPreOrder(tree->right);
    }
}

//*******************************************************
// This function displays the values  stored  in a tree   *
// in postorder.                                          *
//*******************************************************
void IntBinaryTree::displayPostOrder(TreeNode *tree) const
{
    if (tree)
    {
        displayPostOrder(tree->left);
        displayPostOrder(tree->right);
        cout << tree->value << "  ";
    }
}

// Programming Challenge 10 Definition

void IntBinaryTree::treePrint(IntBinaryTree::TreeNode * tree, std::ostream & out) {
    if (!tree) out << "_";
    else {
        out << "(" << tree->value;
        treePrint(tree->left, out);
        out << ",";
        treePrint(tree->right, out);
        out << ")";
    }
}
```