CSC 122 001 Computer Science II
Julius Ranoa

Chapter 19 Programming Challenge 8 *Employee Tree*

Requirements/Objectives of the Challenge:
Design an *EmployeeInfo* class that holds the following employee information:

Employee ID Number: an integer
Employee Name: a string

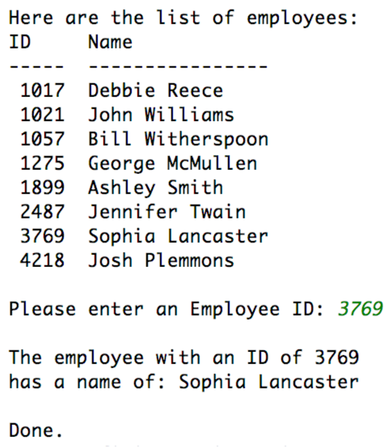
Implement a binary tree whose nodes hold an instance of the *EmployeeInfo* class. The nodes should be sorted on the Employee ID number.
Test the binary tree by inserting nodes with the following information: *(Test data included in source code).*
Your program should allow the user to enter an ID number, then search the tree for the number. If the number is found, it should display the employee's name. If the node is not found, it should display a message indicating so.

Implementation:
There are three files included in the submitted project:

(1) main.cpp – The driver program. The test data is included here in a function called *loadTestData*.
(2) BinaryTree.h – A template implementation for a binary tree. Instead of having objects for the nodes, shared pointers are used. An iterator class is also included, although I couldn't make it structure it like an STL iterator class due to the difference in structure for binary trees. Note that this implementation of a binary tree is incomplete and is only coded so far as to complete the requirements of the challenge. *(Although, the templating is completely unnecessary.)*
(3) EmployeeInfo.h – The header file for the *EmployeeInfo* class, as stated in the requirements. Overloaded functions for the ff. comparison operators: <, >, ==; are added to make the class work with the Binary Tree template.

Screenshots of Runtime:

(1) With a positive test.



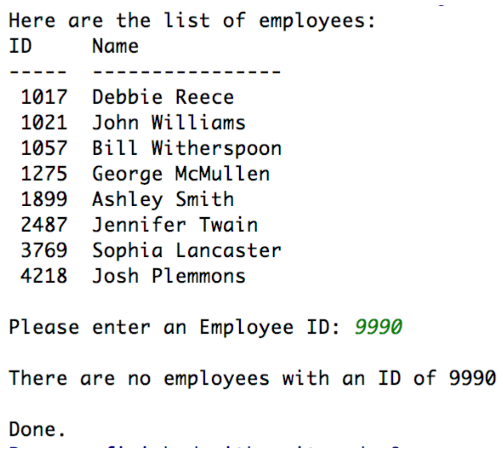
```
Here are the list of employees:
ID     Name
-----  ----------------
 1017  Debbie Reece
 1021  John Williams
 1057  Bill Witherspoon
 1275  George McMullen
 1899  Ashley Smith
 2487  Jennifer Twain
 3769  Sophia Lancaster
 4218  Josh Plemmons

Please enter an Employee ID: 3769

The employee with an ID of 3769
has a name of: Sophia Lancaster

Done.
```

(2) With a negative test.



```
Here are the list of employees:
ID     Name
-----  ----------------
 1017  Debbie Reece
 1021  John Williams
 1057  Bill Witherspoon
 1275  George McMullen
 1899  Ashley Smith
 2487  Jennifer Twain
 3769  Sophia Lancaster
 4218  Josh Plemmons

Please enter an Employee ID: 9990

There are no employees with an ID of 9990

Done.
```

Source Code for main.cpp

```cpp
#include <iostream>
#include <iomanip>
#include "EmployeeInfo.h"
#include "BinaryTree.h"

void loadTestData(BinaryTree<EmployeeInfo> &);

int main() {

    BinaryTree<EmployeeInfo> EmpList;
    loadTestData(EmpList);

    // Display the list of employees.
    std::cout << "Here are the list of employees: \n";
    std::cout << "ID     Name \n";
    std::cout << "-----  ---------------- \n";

    BinaryTree<EmployeeInfo>::Iterator i = EmpList.getInOrderIterator();
    for (; i.good(); ++i) {
        std::cout << std::setw(5) << i->getID() << "  ";
        std::cout << i->getName() << "\n";
    }
    std::cout << "\n";

    // Ask the user for an ID:
    unsigned argID;
    std::shared_ptr<EmployeeInfo> argInfo = nullptr;

    std::cout << "Please enter an Employee ID: ";
    std::cin >> argID;
    std::cout << "\n";
    argInfo = EmpList.extract(argID);

    if (argInfo) {
        std::cout << "The employee with an ID of " << argID << "\n"
                  << "has a name of: " << argInfo->getName() << "\n";
    } else {
        std::cout << "There are no employees with an ID of " << argID << "\n";
    }

    std::cout << "\nDone.";
    return 0;
}

void loadTestData(BinaryTree<EmployeeInfo> & BinTree) {
    // TEST DATA
    const int SIZE = 8;
    int EmpIDs[SIZE] = {
        1021, 1057, 2487, 3769, 1017, 1275, 1899, 4218
    };
    std::string EmpNames[SIZE] {
        "John Williams", "Bill Witherspoon", "Jennifer Twain",
        "Sophia Lancaster", "Debbie Reece", "George McMullen",
        "Ashley Smith", "Josh Plemmons"
    };

    for (int i = 0; i < SIZE; i++) {
        EmployeeInfo temp(EmpIDs[i], EmpNames[i]);
        BinTree.insert( temp );
    }
};
```

Source Code for BinaryTree.h

```
#ifndef CH19_PR8_EMPLOYEE_TREE_BINARYTREE_H
#define CH19_PR8_EMPLOYEE_TREE_BINARYTREE_H

#include <memory> // For the shared pointers.
#include <vector> // For the iterator.

#include <iostream>

/*
 * NOTE: This is a limited implementation of a binary tree class.
 *       Other functions that are not included such as node deletion
 *       were not needed to fulfill the programming challenge.
 */

template <class T>
class BinaryTree {

private:
    struct TreeNode {
        std::shared_ptr<T> value;
        TreeNode * left;
        TreeNode * right;

        TreeNode(
                std::shared_ptr<T> argValPtr,
                TreeNode * argLeft = nullptr,
                TreeNode * argRight = nullptr) {
            value = argValPtr;
            left = argLeft;
            right = argRight;
        }
    };
    TreeNode * root;

    // Private Methods
    void insert(TreeNode * &, std::shared_ptr<T>);
    void destroySubTree(TreeNode *);

public:
    BinaryTree() {
        root = nullptr;
    }
    ~BinaryTree() {
        destroySubTree(root);
    }

    bool search(const T, T&) const;
    std::shared_ptr<T> extract(T) const;
    void insert(T item);

// ITERATOR STUFF
public:
    /*
     * NOTE:
```

```
 * A basic iterator is added, just because I wanted to make one.
 * THIS DOES NOT HAVE THE FULL CAPABILITY OF AN STL ITERATOR.
 * ONLY THE METHODS NEEDED TO MAKE THE DRIVER PROGRAM RUN ARE ADDED.
 */
class Iterator {

private:
    std::vector< std::shared_ptr<T> > objPointers;
    unsigned maxIndex;
    unsigned currentIndex;

    // This function is friended by the Binary Tree class
    void addPointer(std::shared_ptr<T> newItem) {
        objPointers.push_back(newItem);
        maxIndex++;
    }

public:
    Iterator() {
        maxIndex = 0;
        currentIndex = 0;
    }
    friend class BinaryTree; // Allows the binary tree to access the pointer vector.

    bool good() {
        return ( currentIndex >= 0 ) && ( currentIndex < maxIndex );
    }
    std::shared_ptr<T> current() {
        if (currentIndex < maxIndex) {
            return objPointers[currentIndex];
        } else {
            return nullptr;
        }
    }
    void reset() {
        currentIndex = 0;
    }

    // Deferencing Operators
    T& operator*() {
        return *(objPointers[currentIndex]);
    }
    const T& operator*() const {
        return *(objPointers[currentIndex]);
    }
    std::shared_ptr<T> operator->() {
        return objPointers[currentIndex];
    }
    const std::shared_ptr<T> operator->() const {
        return objPointers[currentIndex];
    }

    // Increment Operators

    Iterator& operator++() {
        currentIndex++; // Overflows are handled by current() method
        return *this;
```

```cpp
        }
        Iterator& operator++(int) {
            currentIndex++; // There's no difference between post and pre
                            // in this implementation.
            return *this;
        }

        // Conversion Operators
        operator std::shared_ptr<T>() const {
            if (currentIndex < maxIndex) {
                return objPointers[currentIndex];
            } else {
                return nullptr;
            }
        }
        operator T * () const {
            if (currentIndex < maxIndex) {
                return objPointers[currentIndex];
            } else {
                return nullptr;
            }
        };

    };

// BinaryTree Iterator Generators
private:
    void addInOrderIterator(TreeNode * tree, BinaryTree<T>::Iterator & i);

public:
    BinaryTree<T>::Iterator getInOrderIterator() {
        BinaryTree<T>::Iterator i;
        addInOrderIterator(root, i);
        return i;
    }

};

// Templates

// PRIVATE METHODS

template <class T>
void BinaryTree<T>::insert(TreeNode *& tree, std::shared_ptr<T> newItem) {
    // If tree is empty.
    if (!tree) {
        tree = new TreeNode(newItem);
        return;
    }

    // If value is already in tree.
    if (*tree->value == *newItem) {
        return; // Do nothing.
    }

    if (*newItem < *tree->value) {
        insert(tree->left, newItem);
    } else {
        insert(tree->right, newItem);
    }

    return;
}

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

// PUBLIC METHODS

template <class T>
bool BinaryTree<T>::search(const T searchTerm, T & itemContainer) const {
    TreeNode * tree = root;
    while (tree) {
        if (*(tree->value) == searchTerm) {
            itemContainer = tree->value;
            return true;
        } else if (searchTerm < tree->value) {
            tree = tree->left;
        } else {
            tree = tree->right;
        }
    }
    return false;
}


template <class T>
std::shared_ptr<T> BinaryTree<T>::extract(const T searchTerm) const {
    TreeNode * tree = root;
    while (tree) {
        if (searchTerm == *(tree->value)) {
            return tree->value;
        } else if (searchTerm < *(tree->value)) {
            tree = tree->left;
        } else {
            tree = tree->right;
        }
    }
    return nullptr;
}

template <class T>
void BinaryTree<T>::insert(T item) {
    insert(root, std::make_shared<T>(item));
}

// BinaryTree Iterator Generator -- Implementation
template <class T>
void BinaryTree<T>::addInOrderIterator(TreeNode * tree, BinaryTree<T>::Iterator & i) {
    if (tree) {
        addInOrderIterator(tree->left, i);
        i.addPointer(tree->value);
        addInOrderIterator(tree->right, i);
    }
}

#endif //CH19_PR8_EMPLOYEE_TREE_BINARYTREE_H
```

Source Code for EmployeeInfo.h

```cpp
#ifndef CH19_PR8_EMPLOYEE_TREE_EMPLOYEEINFO_H
#define CH19_PR8_EMPLOYEE_TREE_EMPLOYEEINFO_H

#include <string>
#include <iostream>

class EmployeeInfo {

private:
    int EmpID;
    std::string EmpName;

public:
    EmployeeInfo() {
        EmpID = 0;
        EmpName = "";
    }
    EmployeeInfo(int id, std::string name = "") {
        EmpID = id;
        EmpName = name;
    }
    int getID() const {
        return EmpID;
    }
    std::string getName() const {
        return EmpName;
    }

};

/*
 * The following functions are made assuming that each employee
 * will have a unique ID and that two EmployeeInfo objects
 * with the same ID refer to the same employee.
 *
 * THESE ARE HERE JUST TO MAKE THE CLASS WORK WITH THE BINARY TREE.
 * I couldn't make the BinaryTree template to accept a
 * function pointer as argument with the == operator as the default.
 * This is the next best thing.
 */
inline bool operator< (const EmployeeInfo &lhs, const EmployeeInfo &rhs) {
    return ( lhs.getID() < rhs.getID() );
}

inline bool operator> (const EmployeeInfo &lhs, const EmployeeInfo &rhs) {
    return ( lhs.getID() > rhs.getID() );
}

inline bool operator== (const EmployeeInfo &lhs, const EmployeeInfo &rhs) {
    return ( lhs.getID() == rhs.getID() );
}

#endif //CH19_PR8_EMPLOYEE_TREE_EMPLOYEEINFO_H
```