COSC230 Practical 6: Binary Search Trees and Hash Tables (Practical for Weeks 10 and 11)

The purpose of this practical is to understand and extend a simple binary search tree class implemented in `C++`, and to learn how to use the STL implementations of balanced binary search trees. The readings for this practical are taken from Chapter 6 of the prescribed textbook: *Data Structures and Algorithms in `C++`*, Fourth Edition, Adam Drozdek.

Question 1: **Recursive search and insert functions**

Download code for a binary search tree class (in file `BST.h`) from Moodle. Try testing some of the member functions we went through in class and use `ddd` to visualize a binary search tree. Now think recursively: the root of a binary tree can have left and right children, which form the roots of two binary *subtrees*. These children in turn can each have left and right children (they would be children of children), forming the roots of four binary *subtrees of subtrees*, and so on. Now try implementing the member functions `recursive_search(BSTNode<T>*, const T&)` and `recursive_insert(BSTNode<T>*&, const T&)` which are recursive versions of the iterative members `search(BSTNode<T>*, const T&)` and `insert(const T&)`. Tail recursion is sufficient here. Remember that a recursive function requires an *anchor case*.

Question 2: **Balancing a binary search tree using a simple array method**

A tree is *height balanced* if, for any node `n`, the heights of the left and right subtrees of `n` differ by at most one. The aim of this question is to balance a binary search tree using a simple method that requires a sorted array of the tree node values. This sorted array can be generated from inorder traversal of an unbalanced binary search tree. The unbalanced tree is then deleted, and the `balance()` member function applied (it implements a recursive bisection of the sorted array, then inserts each middle element into the tree) to construct a balanced tree. An easy way to achieve this is to write a class, called `Balanced_BST<T>`, derived from `BST<T>` that includes an array data member (in this case a `vector`), and re-defines the `visit()` member function used in the traversal methods to insert node elements into this array. The following code can be used in `main()` to test your class:

```
Balanced_BST<int> a;

a.insert(2), a.insert(5), a.insert(7), a.insert(9);

a.balance();
// Check a is balanced by visualizing the tree in the Data Display Debugger
```

Question 3: **Improving the efficiency of a database program**

In *Practical 3* you used the STL implementation of a doubly linked list to construct a database to manage airline reservations. Download that code from Moodle (Weeks 8–9). In this practical, you will construct a database of cruise line reservations. A cruise line operates cruise ships, and typically works with much larger passenger lists than used for individual airline flights (therefore, time complexity becomes more of an issue). To do this, replace the STL container `list` in `database.cc`, with the STL container `set` that is an implementation of a type of balanced binary search tree. Some member functions will need to be re-named for the code to compile. Now use online `C++` documentation to comment on how the complexities of the member functions in your new database program have changed. What are the efficiencies that have been gained?

Question 4: **Generating a cross-reference table**

In this question, you will use the STL container `map` to generate a cross-reference table from a text file `text_excerpt.txt` available on Moodle. The `map` container allows indexing with any data type, and is therefore a generalization of arrays: which only allow consecutive unsigned integers to be used to index consecutive elements. Hash tables (see Question 5) are another data structure that allow indexing with any data type, but require purpose-built hash functions to achieve this. In `map`, indexing is done through `key-value` pairs: where `key` is a unique index, and `value` is the value accessed through `key`. The keys are also stored and maintained in sorted order to enable rapid access and modification of values. The STL implements `map` using a type of balanced binary search tree. The cross-reference table should contain all of the words from `text_excerpt.txt`, as well as the line numbers on which each word appears. These line numbers should be stored on linked lists associated with each word. To generate the table, for each word, iterate through the associated linked list. In this exercise, don't be concerned with parsing words or processing punctuation such as quotes and full stops – an advanced exercise would include stripping all words of punctuation before processing, but for simplicity, don't be concerned with that here.

Question 5: **Hashing with chaining**

Implement a hash table that resolves collisions by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \ mod \ 9$. What is the length of the longest linked list when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into the hash table?

Solutions

Question 1:

The member functions are implemented as follows:

```
template<class T>
T* BST<T>::recursive_search(BSTNode<T>* p, const T& el) const
{
    if (p != 0) {
        if (el == p->el)                            // Anchor case
            return &p->el;
        else if (el < p->el)
            return recursive_search(p->left, el);   // Tail recursion
        else
            return recursive_search(p->right, el);
    }
    else
        return 0;
}

template<class T>
void BST<T>::recursive_insert(BSTNode<T>*& p, const T& el)
{
    if (p == 0)                         // Anchor case
        p = new BSTNode<T>(el);
    else if (el < p->el)
        recursive_insert(p->left, el);  // Tail recursion
    else
        recursive_insert(p->right, el);
}
```

Question 2:

```
// Class derived from BST to balance a binary search tree using an array method.
#include "BST.h"

template<class T>
class Balanced_BST: public BST<T> {
public:
    void balance()
    {
        BST<T>::inorder();                                      // Create sorted array
        BST<T>::~BST();                                         // Delete old tree
        BST<T>::balance(tree_array, 0, tree_array.size()-1);
                                                                // Build new balanced tree
    }
protected:
    void visit(BSTNode<T>* p)                                   // re-define "visit"
        { tree_array.push_back(p->el); }
    std::vector<T> tree_array;                                  // new vector data member
};
```

Question 3:

The member functions `push_back` and `remove` are replaced by `insert` and `erase`, and `sort` is no longer necessary as values are now stored and maintained in sorted order. The member `push_back` is worst-case $\Theta(1)$, while its replacement `insert` is worst-case $\Theta(\log_2 n)$; which is still very efficient. The benefit is that now values are stored in sorted order instead of order of arrival. The members `find` and `remove` go from worst-case $\Theta(n)$, to worst-case $\Theta(\log_2 n)$; which is a significant improvement (decrease) in time complexity.

Question 4:

```
// Generate a cross-reference table from "text_excerpt.txt".
// Does not process punctuation or parse words.

#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <list>
#include <sstream>
```

```
using namespace std;

int main()
{
    map<string, list<int> > words;
    map<string, list<int> >::iterator it;
    string word, line;
    int line_number = 1;
    list<int> line_number_list;

    // Read in words and line numbers from a file
    ifstream infile("text_excerpt.txt");
    while (getline(infile, line)) {
        istringstream iss(line);
        while (iss >> word) {
            words[word].push_back(line_number);
        }
        ++line_number;
    }
    infile.close();

    // Generate a cross-reference table
    for (it = words.begin(); it != words.end(); ++it) {
        line_number_list = (*it).second;
        cout << (*it).first << " ";
        while (!line_number_list.empty()) {
            cout << line_number_list.front() << " ";
            line_number_list.pop_front();
        }
        cout << endl;
    }
    // Or index directly off "word" variable
    cout << "Harry appears on line numbers: ";
    line_number_list = words["Harry"];
    while (!line_number_list.empty()) {
        cout << line_number_list.front() << " ";
        line_number_list.pop_front();
    }
    cout << endl;
    return 0;
}
```

Question 5:

```
// A simple hash table implementation using chaining.

#include <iostream>
#include <array>
#include <list>
using namespace std;

int hash_function(int k) { return k % 9; }

int main()
{
    // Using STL containers "array" and "list".
    list<int> Table[9];                             // 9-element array of list values.
    array<int,9> key = {5,28,19,15,20,33,12,17,10};
    array<int,9> value(key);                        // "value" is just a copy of "key"
                                                    // but would usually be different.

    // Inserting key-value pairs into hash table.
    for (int i = 0; i != 9; ++i) {
        Table[hash_function(key[i])].push_back(value[i]);
    }

    // Printing out hash table contents.
    list<int>::iterator i1, i2;
    for (int i = 0; i != 9; ++i) {
        cout << "slot " << i << ": ";
        i1 = Table[i].begin();
        i2 = Table[i].end();
        if (Table[i].empty()) {
            cout << 0; }
        else {
            for (; i1 != i2; ++i1) {
                cout << *i1 << " ";
            }
        }
        cout << endl;
    }

    return 0;
}
```