COSC230 Practical 6:
Binary Search Trees and Hash Tables
(Practical for Weeks 10–12)

The purpose of this practical is to understand and extend a simple binary search tree class, a red-black tree class, and to implement a simple hash table in `C++`. You will also become familiar with the STL implementations of red-black trees and hash tables.

Question 1: **Recursive binary search tree functions**

Download the file `bst.h` for the binary search tree class from myLearn. Both binary search trees and linked lists have a natural recursive structure. For example, a binary search tree is made up of a root node and its left and right subtrees. Each left and right subtree, in turn, is made up of a root node (i.e., a child of the original root node) and its left and right subtrees, and so on – all the way down to the leaves of the tree. In a similar way, a linked list is made up of a head node and its sublist; which in turn is made up of a head node and its sublist, and so on – all the way down to the end of the list. We will now implement recursive binary search tree functions to search for a key, to perform a tree traversal in different orders, and to find the height of each leaf node in a tree.

(a) Develop a recursive implementation of the search function:

```
Node<T>* search(Node<T>*, T).
```

Think about the condition when the Base is reached, and what the recursive strategy should be at each step of Recursion in terms of the binary search tree property and the left and right subtrees at each step. Remember that each step of Recursion should get closer to the Base.

(b) Use the `bst` class to construct a binary search tree. Now print its keys in sorted order by calling the function `inorder_walk` to traverse the BST. Following this, implement similar functions to perform preorder and postorder tree traversals, where keys are now printed in an order so that the root is printed before the left and right subtrees (preorder) and after the left and right subtrees (postorder). These functions will come in useful when you look at "expression trees" in Assignment 4.

(c) Make use of Part (b) to write a function `void inorder_leaf_height(Node<T>*, int)` to return the height of each leaf node in the tree along a simple path from the root to that leaf. Think about the condition that a leaf node satisfies that an internal node does not.

Question 2: **Improving the efficiency of a database program**

In *Practical 3* you used `std::list`, the STL implementation of a doubly linked list, to construct a database to manage airline reservations. Download that code from myLearn (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 `std::list` in `database.cc`, with the STL container `std::set` that is an implementation of a type of binary search tree called a red-black 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 3: **Red-Black Tree Insert helper functions**

Download the file `rbt.h` for the red-black tree class from myLearn. In the lectures, we did not look at `right_rotate`, or the second half of `insert-fixup`. In Part (a), your job is to complete these functions to obtain a complete RBT insert function. In Part (b), you will use this function to investigate worst-case behaviour when constructing binary search trees.

(a) Implement `void right_rotate(Node<T>*)` by copying and editing `left-rotate`, making use of the diagrams in the lecture slides "Insert into a Red-Black Tree". Complete the second half of `void insert_fixup(Node<T>*)` by copying the first half, and then interchanging "left" and "right" everywhere they appear.

(b) Use your `rbt` class insert function to construct a red-black tree from keys given in the following order: 1,2,3,4,5,6,7. Now construct a binary search tree with the same set of keys using the insert function from the `bst` class. Compare the heights of both trees using `ddd` and note any difference.

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

In this question, you will use the STL container `std::map` to generate a cross-reference table from a text file `text_excerpt.txt` available on myLearn. The `std::map` container is implemented as a red-black tree, and takes `key-value` pairs: where `key` is a unique key, and `value` is the value accessed through `key`. The keys are stored and maintained in sorted order as they are in `std::set`. 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. For this purpose, each `key` would be a word, while each `value` would be the head of a linked list storing all line numbers for that 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 50, 2800, 190000, 1500, 2000, 3300, 1200, 1700, and 1000 into the hash table?

Question 6: **Using STL hash tables**

Now use the STL container `std::unordered_map` which implements a hash table with a default hash function given by `std::hash`. Try the same key-set from Q5 and print the contents of each slot (a slot is called a bucket in STL language). How does the STL hash table compare with the one implemented in Q5?

Solutions

Question 1:

The recursive member functions are implemented as follows:

```
template<class T>
Node<T>* Binary_Search_Tree<T>::search(Node<T>* p, T k)
{
    if (p == 0 || k == p->key)
        return p;
    if (k < p->key)
        return search(p->left, k);
    else
        return search(p->right, k);
}

template<class T>
void Binary_Search_Tree<T>::preorder_walk(Node<T>* p)
{
    if (p != 0) {
        std::cout << p->key << " ";
        preorder_walk(p->left);
        preorder_walk(p->right);
    }
}

template<class T>
void Binary_Search_Tree<T>::postorder_walk(Node<T>* p)
{
    if (p != 0) {
        postorder_walk(p->left);
        postorder_walk(p->right);
        std::cout << p->key << " ";
    }
}
```

```
template<class T>
void Binary_Search_Tree<T>::inorder_leaf_height(Node<T>* p, int counter)
{
    counter++;
    if (p != 0) {
        inorder_leaf_height(p->left, counter);
        if (p->left == 0 && p->right == 0)
            std::cout << "(leaf=" << p->key << ", "
                << "height=" << counter-1 << ")" << " ";
        inorder_leaf_height(p->right, counter);
    }
}

// call as inorder_leaf_height(root, 0)
```

Question 2:

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. This means that displaying and saving the passenger list are now $\Theta(n)$ instead of $\Theta(n \log_2 n)$. 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 3:

(a) The RBT insert helper functions are given as follows:

```
template<class T>
void Red_Black_Tree<T>::right_rotate(Node<T>* p)
{
    Node<T>* q = p->left;
    p->left = q->right;
    if (q->right != sntl)
        (q->right)->par = p;
    q->par = p->par;
    if (p->par == sntl)
        root = q;
    else if (p == (p->par)->left)
        (p->par)->left = q;
    else
        (p->par)->right = q;
    q->right = p;
    p->par = q;
}
```

```
template<class T>
void Red_Black_Tree<T>::insert_fixup(Node<T>* p)
{
    while ((p->par)->color == 'r') {
        if (p->par == ((p->par)->par)->left) {
            Node<T>* q = ((p->par)->par)->right;
            if (q->color == 'r') {
                (p->par)->color = 'b';
                q->color = 'b';
                ((p->par)->par)->color = 'r';
                p = (p->par)->par;
            }
            else {
                if (p == (p->par)->right) {
                    p = p->par;
                    left_rotate(p);
                }
                (p->par)->color = 'b';
                ((p->par)->par)->color = 'r';
                right_rotate((p->par)->par);
            }
        }
        else {
            Node<T>* q = ((p->par)->par)->left;
            if (q->color == 'r') {
                (p->par)->color = 'b';
                q->color = 'b';
                ((p->par)->par)->color = 'r';
                p = (p->par)->par;
            }
            else {
                if (p == (p->par)->left) {
                    p = p->par;
                    right_rotate(p);
                }
                (p->par)->color = 'b';
                ((p->par)->par)->color = 'r';
                left_rotate((p->par)->par);
            }
        }
    }
    root->color = 'b';
}
```

(b) The `main` function is given as:

```
#include "rbt.h"

int main()
{
    Red_Black_Tree<int> T1;

    T1.insert(1), T1.insert(2), T1.insert(3), T1.insert(4);
    T1.insert(5), T1.insert(6), T1.insert(7);

    return 0;
}
```

and similarly for `bst.h`. You cannot include both classes simultaneously, as they both name a `Node` class that will clash. The key point to note in `ddd` is that the height of the red-black tree is 3, while the height of the (vanilla) binary search tree is 6.

Question 4:

```
// Generates 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;

    // 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) {
        list<int> 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: ";
    list<int> 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:

```
#include <iostream>
#include <vector>
#include <list>

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

int main()
{
    // Hash table given by a 9-element array of empty linked lists.
    std::list<char> Table[9];

    // Some key-value pairs stored in a vector array.
    std::vector<std::pair<int,char> > key_value =
        {{50,‘a’},{2800,‘b’},{190000,‘c’},{1500,‘d’},{2000,‘e’},
        {3300,‘f’},{1200,‘g’},{1700,‘h’},{1000,‘i’}};

    // Inserting key-value pairs into the hash table.
    std::vector<std::pair<int,char> >::const_iterator it;
    for (it = key_value.begin(); it != key_value.end(); it++) {
        Table[hash_function((*it).first)].push_back((*it).second);
    }

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

    return 0;
}
```

Question 6:

```
#include <unordered_map>
#include <vector>
#include <iostream>

int main()
{
    // Hash table.
    std::unordered_map<int,char> Table;

    // Some key-value pairs stored in a vector array.
    std::vector<std::pair<int,char> > key_value =
        {{50,‘a’},{2800,‘b’},{190000,‘c’},{1500,‘d’},{2000,‘e’},
        {3300,‘f’},{1200,‘g’},{1700,‘h’},{1000,‘i’}};

    // Inserting key-value pairs into the hash table.
    std::vector<std::pair<int,char> >::const_iterator it;
    for (it = key_value.begin(); it != key_value.end(); it++) {
        Table.insert(*it);
    }

    // Printing the hash table contents.
    for (unsigned i = 0; i != Table.bucket_count(); i++) {
        std::cout << "slot " << i  << ": ";
        auto i1 = Table.begin(i);
        auto i2 = Table.end(i);
        if (i1 == i2)
            std::cout << 0;
        else {
            for (; i1 != i2; i1++) {
                std::cout << (*i1).second << " ";
            }
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;

    return 0;
}
```