Csci 335 Assignment 2

Due Thursday, March 10th

**Programming: Using and Comparing Tree Implementations (100 points)

The goal of this assignment is to become familiar with trees and compare the performance of the basic binary search tree with the self-balancing AVL tree. You will also work with a real world data set and construct a generic test routine for comparing several different implementations of the tree container class. You are encouraged to use the book's implementation for the BST and AVL trees.

Part 1 (5 points)

First, create a class object named SequenceMap that has as private data members the following two:

```
string recognition_sequence_ ;
vector<string> enzyme_acronyms_;
```

Other than the big-five (note that you can use the defaults for all of them), you have to add the following:

a) A constructor `SequenceMap(const string &a_rec_seq, const string &an_enz_acro)`,that constructs a `SequenceMap` from two strings (note that now the vector `enzyme_acronyms_` will contain just one element, the `an_enz_acro`).
b) `bool operator<(const SequenceMap &rhs) const`, that operates based on the regular string comparison between the `recognition_sequence_` strings (this will be a one line function).
c) Overload the `operator<<` for `SequenceMap`.
d) `void Merge(const SequenceMap &other_sequence).` This function assumes that the object's `recognition_sequence_` and `other_sequence.recognition_sequence_` are equal to each other. The function `Merge()` merges the `other_sequence.enzyme_acronym_` with the object's `enzyme_acronym_.` This results in changing the calling object, meaning that `Merge()` is not a `const` function. The `other_sequence` object will not be affected.

This class (which is non-templated) will be used in the following programs. First test it with your own test functions to make sure that it operates correctly.

Part 2

Introduction to the problem

For this assignment you will receive as input two text files, `rebase210.txt` and `sequences.txt`. After the header, each line of the database file `rebase210.txt` contains the name of a restriction enzyme and possible DNA sites the enzyme may cut (cut location is indicated by a ‘) in the following format:

`enzyme_acronym/recognition_sequence/…/recognition_sequence//`

For instance the first few lines of `rebase210.txt` are:

```
AanI/TTA'TAA//
AarI/CACCTGCNNNN'NNNN/'NNNNNNNNGCAGGTG//
AasI/GACNNNN'NNGTC//
AatII/GACGT'C//
AbsI/CC'TCGAGG//
AccI/GT'MKAC//
AccII/CG'CG//
AccIII/T'CCGGA//
Acc16I/TGC'GCA//
Acc36I/ACCTGCNNNN'NNNN/'NNNNNNNNGCAGGT//
…
```

That means that each line contains one enzyme acronym associated with one or more recognition sequences. For example on line 2:
The enzyme acronym `AarI` corresponds to the two recognition sequences `CACCTGCNNNN'NNNN and 'NNNNNNNNGCAGGTG.`

Part 2(a) (45 points)

You will create a parser to read in this database and construct a search tree (either a regular BST or an AVL tree). For each line of the database and for each recognition sequence in that line, you will create a new `SequenceMap` object that contains the recognition sequence as its recognition_sequence_ and the enzyme acronym as the only string of its enzyme_acronyms_, and you will insert this object into the tree. This is explained with the following pseudo code:

```
Tree<SequenceMap> a_tree;
string db_line;
// Read the file line-by-line:
while (GetNextLineFromDatabaseFile(db_line)) {
   // Get the first part of the line:
   string an_enz_acro = GetEnzymeAcronym(db_line);
```

```
   string a_reco_seq;
   while  (GetNextRegocnitionSequence(db_line, a_rego_seq){
      SequenceMap new_sequence_map(a_reco_seq, an_enz_acro);
      a_tree.Insert(new_sequence_map);
   }  // End second while.
}  // End first while.
```

In the case that the new_sequence_map.recognition_sequence_ equals the recognition_sequence_ of a node X in the tree, then the search tree's `Insert()` function will call the `X.Merge(new_sequence_map)` function of the existing element. This will have the effect of updating the enzyme_acronym_ of X. Note, that this will be part of the functionality of the Insert() function. The Merge() will only be called in case of duplicates as described above. Otherwise, no Merge() is required and the `new_sequence_map` will be inserted into the tree.

To implement the above, write a test program named **`queryTrees`** which will use your parser to create a search tree and then allow the user to query it using a recognition sequence. If that sequence exists in the tree then this routine should print all the corresponding enzymes that correspond to that recognition sequence.

Your programs should run from the terminal as follows:

`QueryTrees <database file name> <flag>`

`<flag>` should be "BST" for binary search tree, and "AVL" for AVL tree.

For example you can write on the terminal:

QueryTrees rebase210.txt BST

Part2(b) (35 points)

Next, create a test routine named **`testTrees`** that does the following:

1. Parse the database and construct a search tree (this is the same as in Part2(a)).
2. Print the number of nodes in your tree n. Compute and print the average depth of your search tree, i.e. the internal path length divided by n. Also print the ratio of the average depth to $\log_2 n$. E.g., if average depth is 6.9 and $\log_2 n = 5.0$, then you should print $\frac{6.9}{5.0} = 1.38$.
3. Search the tree for each sequence in the `sequences.txt` file. Print the total number of successful queries. Count the total recursion calls for your find() function and print the average number of recursion calls, i.e. #total number of recursion calls / number of queries.

4. Remove every other sequence in `sequences.txt` from the tree. Print the total number successful removes. Count the total recursion calls for your delete() function and print the average number of recursion calls, i.e. #total number of recursion calls / number of remove calls.
5. Recompute the statistics in step 2 and print them.

For both Part2(a) and Part2(b) you **must** write the test routine using templates so each tree can be used interchangeably. The trees should have identical interfaces. The tree classes themselves should be well separated from the logic of the SequenceMap object.

Your program should run from the terminal as follows:

`TestTrees <database file name> <queries file name> <flag>`

`<flag>` should be "BST" for binary search tree, and "AVL" for AVL tree.

For example you can write on terminal

TestTrees rebase210.txt sequences.txt AVL

Part 3 (15 points)

Add to the AVL tree implementation a function that takes as input two keys k1 and k2 (assuming that k1 <= k2) prints all elements in the tree with keys between k1 and k2. In this implementation k1 and k2 will be two recognition sequences.

Your program should run as follows:

TestRangeQuery <database file name> <key1> <key2>

For example you can write:

`TestRangeQuery rebase210.txt CC\'TCGAGG T\'CCGGA`

Note that the \ is entered in the recognition sequences in order for the ' not to confuse the Linux shell.

Note: Because your program will be tested on Linux, you should make sure the input files are in the Unix file format and you design your program to expect Unix formatted files. These files already should be in Unix format but you can use the dos2unix and unix2dos tools to convert them back and forth. (Mac OS X now uses the Unix convention)