Csci 335 Assignment 2

Due Sunday October 11, 2015 @ Noon

Using and Comparing Tree Implementations

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 as well as lazy vs non-lazy deletion for the 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 may use the book's implementation for the BST and AVL trees but will have to implement lazy deletion variants yourself.

First, create a data structure named SequenceMap that has as data members a string and a set of strings. This will be the data that we store in our trees and passed in to the template tree classes. This class should be comparable using the string member as the comparison key by overloading the `operator<` function. This class should also have a merge(SequenceMap other) method that tells a search tree what to do in case of duplicates, this is described below.

Second, modify the BST and AVL tree implementations to count the number of recursive calls to the insert, contains, and remove methods. Also create a new template class that implements the AVL tree using lazy deletion.

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 " **'** " [single quote, no spaces]) in the following format:

`enzyme_acronym/recognition_sequence/…/recognition_sequence//`

You will create a parser to read in this database and construct a search 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 search key and the enzyme acronym in the set of strings and you will insert this object into the tree. In the case of a duplicate on insertion, the search tree will call the x.merge(SequenceMap other) function of the existing element, x, in the tree which will add the newly created other SequenceMap's enzyme acronym to the set of enzyme acronyms contained by the SequenceMap in the tree.

Now create a small 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 activate on that recognition sequence. Note that the recognition sequence should include the '-symbol which indicates the location of the cut.

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

1. Parse the database and construct a search tree. Print the total number of recursive calls to `insert` after processing the entire database.
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 and the total number of recursive calls to the `contains` method.
4. Remove every other sequence in `sequences.txt` from the tree. Print the total number successful removes and number of recursive calls to the `remove` method.
5. Recompute the statistics in step 2 and repeat the search in step 3 printing the number of calls to `contains`. Note these should not change with lazy deletion.

*You **must** write the test routine using templates (or inheritance) so each tree can be used interchangeably.* The trees should have identical interfaces. Code shared by both programs in this assignment should be included or linked from a separate file. The tree classes themselves should be well separated from the logic of the SequenceMap object.

Good design practices are an important part of the grade for this assignment.

Your programs should run from the terminal as follows:

```
queryTrees <database file name> <flag>
```

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

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

Create a table comparing the statistics you observed for each of the four search tree variants. Include a caption comparing these results to what you would expect given the analysis of these data structures.

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)