Csci 335 Assignment 2-Spring 2023

Due 11:00PM Thursday, March 9

This is an individual assignment.

- **Read and follow the contents of Programming Rules document on blackboard (Course Information Section).**
- **Submit only the files requested in the deliverables at the bottom of this description to Gradescope by the deadline.**

**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.

When you accept the assignment on GitHub Classrooms, you will be provided with the following starter code: `avl_tree.h, bst_tree.h, query_tree.cc, test_tree.cc, dsexceptions.h, README.md, Makefile and *.txt files` for expected output.

Part 1 (10 points)

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

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

The name of the file should be `sequence_map.h`

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.

c) Overload the `operator<<` for `SequenceMap`

~~d) Overload the `operator >>` for `SequenceMap`~~ DO NOT IMPLEMENT (d)

e) `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_`. 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. No need to submit your test functions.

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. Do not change this file. 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 **query_tree**. This program 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:

query_tree <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:

```
./query_tree rebase210.txt BST
```

The user should enter THREE strings (supposed to be recognition sequences) for instance:

CC'TCGAGG

TTA'TAA

TC'C

Your program should print in the standard output their associated enzyme acronyms. In the above example the output will be

AbsI

AanI PsiI

Not Found

I will test it with a file containing three strings and run your code like that:

```
./query_trees rebase210.txt BST < input_part2a.txt
./query_trees rebase210.txt AVL < input_part2a.txt
```

Part2(b) (45 points)

Next, create a test routine named **`test_tree`** that does the following in the sequence described below:

1. Parses the database and construct a search tree (this is the same as in Part2(a)).
2. Prints the number of nodes in your tree n.
3. Computes the average depth of your search tree, i.e. the internal path length divided by n.
    a. Prints the average depth.
    b. Prints 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 $\frac{6.9}{5.0} = 1.38$.
4. Searches (`find()`) the tree for each string in the `sequences.txt` file. Also counts the total number of recursive calls for all executions of `find()`.
    a. Prints the total number of successful queries (number of strings found).
    b. Prints the average number of recursion calls, i.e. #total number of recursion calls / number of queries.
5. Removes every other sequence in `sequences.txt` from the tree. Also counts the total number of recursion calls for all executions of `remove()`.

   a. Prints the total number successful removes.
   b. Prints the average number of recursion calls, i.e. #total number of recursion calls / number of remove calls.
6. Redo steps 2 and 3:
   a. Prints number of nodes in your tree.
   b. Prints the average depth.
   c. Prints the ratio of the average depth to $\log_2 n$.

The output of Part2(b) should be of the exact form:

2: <integer>

3a: <float>

3b: <float>

4a: <integer>

4b: <float>

5a: <integer>

5b: <float>

6a: <integer>

6b: <float>

6c: <float>

If you didn't complete a step, just print after the step number: Not Done

---

<u>For both Part2(a) and Part2(b)</u> you **must** write the test routine using templates so each tree can be used interchangeably. The trees should have identical interfaces.

Your program should run from the terminal as follows:

`test_tree <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

```
      ./test_tree rebase210.txt sequences.txt AVL >
output_avl.txt

      ./test_tree rebase210.txt sequences.txt BST >
output_bst.txt
```

Submission

On or before the due date listed on Bb, using **GitHub Classrooms**, submit (and only submit) the following files to Gradescope:

- Part 1:
  - `sequence_map.h`: Your original sequence_map.h will be reused for all further sections.
- Part 2A:
  - `query_tree.cc`
- Part 2B:
  - `test_tree.cc`
- Both Parts 2A & 2B:
  - ***bst_tree.h***
  - ***avl_tree.h***

- README file as described in Assignment 1 requirements and Programming Rules document.

In summary, submit your assignment by following these two steps:

1. Push all materials to GitHub Classrooms. The material includes (and only includes) the following files:
   - `README`
   - `sequence_map.h`
   - ***bst_tree.h***
   - ***avl_tree.h***
   - `query_tree.cc`
   - `test_tree.cc`
2. Do not change their names in any way—do not capitalize any letter. Do not delete any files.
3. Once you have your code on GitHub Classrooms, upload it to Gradescope **using GitHub Classrooms**. Instructions for all this can be found on YouTube. Make sure you run and test your code without Gradescope.