CSCI 335 Spring 2020

# HW 2: Lists and Trees

## 100 points (85 points deliverables, 15 points for design)

**Due 11pm March 2, 2020**

**This is an individual assignment**

**Instructions:**

1. **Read and follow the contents of 335 Spring2202 335 Programming Rules document on the blackboard (Course Information Section).**
2. **Read the assignment description below.**
3. **Submit only the files requested in the deliverables at the bottom of this description to gradescope by the deadline.**

**Learning Outcome: The goal of this assignment is to become familiar with lists and trees and compare the performance of 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 AVL tree. Acknowledge the sources you use in the README file.**

**Q1: C++11** **10 points**

Implement the STL find routine that returns the *iterator* containing the first occurrence of *x* in the range that begins at *start* and extends up to but not including *end.* If *x* is not found, *end* is returned. This is a non-class (global function) with signature

**template <typename Iterator, typename object>**

**Iterator my_find(Iterator start, iterator end, const object & x);**

**Q1 Deliverable:** Submit the code for the routine in a file titled HW2Q1.cc and a README file.

**Q2: AVL Trees** **75 points**

**Files provided:** avl_tree.h, query_tree.cc, test_tree.cc, test_tree_mod.cc, dsexceptions.h, a README.txt, a Makefile and 6 text (.txt) files. The output should look exactly as described in Part 1 and Part 2a and 2b to obtain the full grade (consider this part of exact API specification).

**Part 1 (15 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 after the constructor is called 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 in order to print out the enzyme_acronyms vector, for a given recognition sequence. Refer to assignment 1 for example of overloading the operator signature.

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_. 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) (25 points)**
You will create a parser to read in this database and construct 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  (GetNextRecognitionSequence(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** 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:

**query_tree <database file name>**

For example you can write on the terminal:

./query_tree rebase210.txt

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
We will test it with a file containing three strings and run your code like that:
./query_trees rebase210.txt < input_part2a.txt

**Part2(b) (20 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 and 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 and 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

Your program should run from the terminal as follows:

**test_tree <database file name> <queries file name>**

For example you can write on terminal
./test_tree rebase210.txt sequences.txt

**Part2(c) (15 points)**

You will use the avl_tree.h code you have written for Part2(b) and you will modify it in order to implement double rotations directly instead of calling the two single rotations. Name your modified implementation avl_tree_p2c.h. Run the exact same routines as in Part2(b), but now with your modified AVL implementation. The executable should be named test_tree_mod. The results should be the same as in Part2(b).
For example you can write on terminal

./test_tree_mod rebase210.txt sequences.txt

---

You will be given a mandatory Makefile, along with some code to start (start of main functions query_tree.cc test_tree.cc test_tree_mod.cc

---

Q2 Deliverables: You should submit these files:

- README file (as discussed in class)

- Part 1:
  sequence_map.h: Your original sequence_map.h will be reused for all further sections.

- Part 2A: (Modify avl_tree code by adding functions)
  query_tree.cc
  avl_tree_p2a.h

- Part 2B:
  test_tree.cc
  avl_tree_p2b.h

- Part 2C:
  test_tree_mod.cc
  avl_tree_p2c.h