CSE101 Assignment
HW3: Life is all about balance

- All code must be written in C/C++.
- Please be careful about using built-in libraries or data structures. The assignment instructions will tell you what is acceptable, and what is not. If you have any doubts, please ask the instructors or TAs.

1 Problem description

This is a hard assignment. You have been forewarned. But you will learn a lot doing it.

Read this document carefully. Half the questions on Piazza can be answered by just reading the instructions. Also, the algorithm for this assignment requires a fair bit of thought, so read my suggestions carefully.

Main objective: We are going to design a fast data structure to perform insertions and *range* queries on text. Basically, your input file will have a (long) list of words to insert, interspersed with range queries. The latter determines the number of words seen so far in a given range. For example, if the range is "ab" and "bc", you have to find the number of words that are lexicographically between "ab" and "bc".

What makes this assignment particularly challenging is that your data structure will need to handle a million inserts and up to two million range queries. So you will have to build a data structure that is really efficient.

You cannot use any built in data structures for storing the words and answering range queries. You can use libraries for I/O and string processing.

Setup: You can access a Codio unit for this assignment. There is a directory "Wordrange" that contains a number of test input/output files, which shall be explained later. You must write all your code in that directory, and not in any subdirectory. There are also some testing scripts. Please check out the README for more details on that.

Format and Output: You should provide a Makefile. On running `make`, it should create an executable "wordrange". You should run the executable with *two* command line arguments: the first is an input file, the second is the output file. You must provide a README with an explanation of the usage and a description of the files involved. *Please cite any sources you used, such as online code, code from a previous course (that you may have written), or extensive discussions with someone.*

All your files must be of the form *.c, *.cpp, *.h, *.hpp. When we grade, all other code files will be deleted. (So do not try to script some part in another language.)

Each line of the input file will be of the following two forms:

i <STRING>

or

r <STRING1> <STRING2>

The first line above means insert the string into your data structure. If the string is already present, do not insert again. Each word should only appear once in the data structure.

The second line above means: count the number of strings (currently stored) that are lexicographically between STRING1 and STRING2. In other words, we want the number of all strings STR such that STRING1 $\leq$ STR $\leq$ STRING2, where comparison is lexicographic (in C++, this is just comparing by $<$ or $>$). This number, also called the range size, should be printed in the output file, in a separate line. You can assume that STRING1 $<$ STRING2.

For more clarity, look at the small test input/output files.

Data structure instructions: You cannot use inbuilt data structures in C++. (Actually, there probably isn't an inbuilt data structure that solves this assignment.)

You need to build a self-balancing tree that stores the words. Since that's what I taught in class, we will use the AVL tree for this assignment. Note that you do not need to implement deletions, just insert, find, and range queries.

How to do range queries? (I'm glad you asked.) A naive method would be to simply traverse the tree and find all keys in the given range. This would be a $\Theta(n)$ algorithm, where n is the number of nodes in the tree. Since you need to process a million such queries (and n is at least a few hundred thousand), this is not feasible.

There is actually a $\Theta(\log n)$ algorithm for answering range queries in AVL trees. But for this algorithm, you will need to store *subtree sizes* at every node. In other words, at every node, you need to store the number of descendants of that node. Once you have subtree sizes, range queries can be answered more quickly. I won't give the full solution, but let me give some hints.

As I always say, think recursion. Suppose you want to find the size of a range $[str1, str2]$ in the subtree rooted at some node, say x. First, compare the key at x with $str1$ and $str2$. You will have three possibilities for the comparisons (assuming, of course, that $str1 < str2$). Two of the those possibilities can be resolved by a *single* recursive call.

The third case is the interesting one, where you have to look at both children of x. But, if you think about it carefully, for the child subtrees, you only need to answer a "less than" or "greater than" query. These are queries where you need to find the number of nodes greater than (or less than) a given string.

These types of queries can again be handled by recursion. Here, you will notice (again thinking about it carefully) that if subtree sizes are already stored, then *it will save you recursive calls.* Indeed, "less than" or "greater than" queries can be answered by making *at most* one recursive call.

All in all, your procedure will only need to see $O(\log n)$ nodes for any query.

How to maintain your sanity: The easiest option is to avoid this assignment. But if you're reading this, you're already in the mood for some serious code writing. Ok then, read these tips to make your life much much easier.

- Understand the AVL insertion process well. The Wikipedia article (`https://en.wikipedia.org/wiki/AVL_tree`) is an excellent resource. There are *four* cases you need to code up.
- Use the visualization on the Schedule webpage to aid your understanding.
- Have your nodes store *heights* (for the AVL property) and *subtree sizes* (for the range queries).
- There are two rotations (left and right), that you should code up as separate functions. Be very very careful while coding them. Note that rotations affect heights and subtree sizes, so make sure you update this information. Height changes affect heights of ancestors, so make sure all updates are performed.
- Rotations and AVL fixes often access grandchildren of nodes, which (initially) can be null. This leads to many seg faults. Reduce your bugs (and code length) by having special functions that update the height/subtree/parent of a node, instead of directly modifying the field. This way, you have a safe method that first checks if the node is null before trying to modify the field.
- Rotations change the root of subtrees, which means the child pointer of the previous parent must change.
- The root is a special case. Rotations can change the root, so make sure you handle this special case.
- Recursion, recursion, recursion.
- Always keep in touch with your parent. One often forgets to update parent pointer, leading to errors that are very hard to trace. Every time a child changes, parents change.
- The simple test example only inserts numbers, so you can perform the same insertions in the AVL viz. Print out preorder traversals to make sure your tree looks the same as the viz.
- And did I mention recursion?

The test cases:

- simple-input.txt, simple-output.txt: This just inserts a few numbers, and is an excellent way to check if at least everything is working correctly.
- allwords-basic.txt, allwords-basic-output.txt: I got a list of all words in the English language from `https://github.com/dwyl/english-words`. Conveniently, these are in sorted order, so you can easily look up the range size using (say) vim. This file inserts all the words and then computes a few ranges at the very end.
- allwords-more-range.txt, allwords-more-range-output.txt: This has 1.8 million operations, just to challenge your code. My code runs in 16 seconds for this input.

2 Grading

You code should terminate within 1 minute for any input file with at most three million operations.

1. (10 points) For a full solution as described above, so handling three million operations in under a minute.
2. (7 points) If you can deal with around a million operations in a minute, you will get 7 points.
3. (5 points) Well, if you pass only pass some tiny test cases. You could get these points by simply coding up a standard BST, and range searching by reading the entire tree.

3 Some comments

Self-balancing trees are amazing. They are blistering fast, and handle complicated things like range queries. I processed 4.6 million queries in 31 seconds, and I didn't try too hard to optimize my code. 31 seconds!! Contrast this with using linked lists for HW1. This is the stuff that companies are made of. Indeed, the design of variants of these search trees have formed the basis of successful database companies.

BST are incredibly flexible. Just by storing the subtree sizes, range searches become fast. There are many different data structure questions that can solved by adapting BSTs.

I found this amazing website with a whole hoard of BST coding questions. If you can do all of these, you will nail any BST question.

`https://medium.com/@codingfreak/binary-tree-interview-questions-and-practice-problems-439df7e5ea1f`