Problem Set 3: Balanced Trees

This problem set explores balanced trees, augmented search trees, data structure isometries, and how those techniques can be used to find clever solutions to complex problems. By the time you’ve finished this problem set, you’ll have a much deeper understanding for how these concepts relate to one another. Plus, you’ll have designed and implemented several truly beautiful data structures!

Due Thursday, May 7th at 2:30PM Pacific time.
Problem One: Order Statistics Trees

In this problem, you'll take an implementation of a red/black tree that only supports insertions and lookups, then convert into an order statistics tree by adding support for the \texttt{rankOf} and \texttt{select} operations. The \texttt{select} operation is the one we talked about in lecture: it takes in a number \( k \), then returns the \( k \)th order statistic. The \texttt{rankOf} operation is a sort of inverse of \texttt{select}: it takes in a key, then returns the number of elements in the red/black tree smaller than the key.

Download the starter files for PS3 from \texttt{myth} at

```
/usr/class/cs166/assignments/a3
```

and edit the \texttt{RedBlackTree.h} and \texttt{.cpp} files with your solution.

Some notes on this problem:

- You are free to edit whatever parts of the provided starter code that you see fit to edit, provided that (1) you still back the data structure with a red/black tree and (2) all operations run in time \( O(\log n) \), except for the destructor (time \( O(n) \)) and \texttt{printDebugInfo} (can be whatever you'd like). We don't think you will need to do much surgery on the provided \texttt{RedBlackTree} type, so if you find yourself fundamentally rewriting large parts of the code, chances are you're missing an easier solution.

- We've provided two different test harnesses. One of them (\texttt{./run-tests}) will run random inputs into the tree and see how the operations work. The other (\texttt{./explore}) can be used to interactively test your tree or to run small scripts on the tree. We recommend using both in the course of coding this up.

To receive full credit, your code should compile with no warnings and should not have any memory errors (use \texttt{valgrind} to check this). We'll test your code on the \texttt{myth} cluster. There's information about how to run the test drivers in the \texttt{README} file.
Problem Two: Dynamic Prefix Parity

Consider the following problem, called the dynamic prefix parity problem. Your task is to design a data structure that logically represents an array of \( n \) bits, each initially zero, and supports these operations:

- \( \text{initialize}(n) \), which creates a new data structure for an array of \( n \) bits, all initially 0;
- \( \text{ds.flip}(i) \), which flips the \( i \)th bit; and
- \( \text{ds.prefix-parity}(i) \), which returns the parity of the subarray consisting of the first \( i \) bits of the array. (The parity of a subarray is zero if the subarray contains an even number of 1 bits and is one if it contains an odd number of 1 bits. Equivalently, the parity of a subarray is the logical XOR of all the bits in that array).

It's possible to solve this problem with \( \text{initialize} \) taking \( O(n) \) time such that \( \text{flip} \) runs in time \( O(1) \) and \( \text{prefix-parity} \) runs in time \( O(n) \) or vice-versa. (Do you see how?) However, by using balanced trees, it's possible to do significantly better than this.

i. Let's begin with an initial version of the data structure. Describe how to use augmented binary trees to solve dynamic prefix parity such that \( \text{initialize} \) runs in time \( O(n) \) and both \( \text{flip} \) and \( \text{prefix-parity} \) run in time \( O(\log n) \). Argue correctness and justify your runtime bounds.

ii. Explain how to revise your solution from part (i) of this problem so that instead of using augmented binary trees, you use augmented multiway trees. Your solution should have \( \text{initialize} \) take time \( O(n) \), \( \text{flip} \) take time \( O(\log k n) \), and \( \text{prefix-parity} \) take time \( O(k \log n) \). Here, \( k \) is a tunable parameter representing the number of keys that can be stored in each node in the multiway tree. Argue correctness and justify your runtime bounds.

We didn't explicitly discuss the idea of augmenting multiway trees in lecture, but we hope that the generalization isn't too tricky, especially since your tree never changes shape.

iii. Using the Method of Four Russians, modify your data structure from part (ii) so that \( \text{initialize} \) still runs in time \( O(n) \), but both \( \text{flip} \) and \( \text{prefix-parity} \) run in time \( O(\log n / \log \log n) \).

This last step is probably the trickiest part. Here are some hints:

- In Fischer-Heun, the Method of Four Russians took the form of “share solutions to subproblems when you can.” Here, think of the Method of Four Russians as a “divide, precompute, and conquer” approach. That is, break the problem down into multiple smaller copies of itself, precompute all possible answers to the smaller versions of those problems, then solve the overall problem by looking up precomputed answers where appropriate. This will be less about explicitly sharing answers to subproblems and more about having the answers to all possible small problems written down somewhere. Do you see how your solution to part (ii) implicitly breaks the bigger problem down into lots of smaller copies?
- Remember that \( \log_k b = \log b / \log k \) thanks to the change-of-basis formula.
- All basic arithmetic operations are assumed to take time \( O(1) \). However, floating-point operations are not considered basic arithmetic operations, nor are operations like “count the number of 1 bits in a machine word” or “find the leftmost 1 bit in a machine word.”
- An array of bits can be thought of as an integer, and integers can be used as indices in array-based lookup structures.
- Be precise with your choice of block size. Constant factors matter!

As usual, argue correctness. Be sure to justify your runtime bounds precisely – as with the Fischer-Heun structure, your analysis will hinge on the fact that there aren’t “too many” subproblems to compute the answers to all of them.

Pat yourself on the back when you finish this problem. Isn’t that an amazing data structure?