Problem Set 2: 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, April 22nd at 2:30PM Pacific

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 `rankOf` and `select` operations. The `select` operation is the one we talked about in lecture: it takes in a number *k*, then returns the *k*th order statistic. The `rankOf` operation is a sort of inverse of select: it takes in a key, then returns the number of elements in the red/black tree smaller than the key. (The key in question doesn't actually have to be in the tree.)

Copy the starter files for PS2 from `myth` from

```
/usr/class/cs166/assignments/a2
```

to a local directory of your own choosing, then edit `RedBlackTree.h` and `.cpp` 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 `printDebugInfo` (can be whatever you'd like). We don't think you will need to do much surgery on the provided `RedBlackTree` type, so if you find yourself fundamentally rewriting large parts of the code, chances are you're missing an easier solution.
- You can run all tests by executing

  ```
  ./run-tests
  ```

  and following the prompts.
- The `select` function should throw a `std::out_of_range` exception if the index to select is invalid (for example, trying to select the fifth item in a tree with four elements). However, the `rankOf` function should work just fine regardless of whether the key is in the tree.

To receive full credit, your code should compile with no warnings (`-Wall -Wpedantic -Werror`) and should not have any memory errors (use `valgrind` to check this). We'll test your code on the `myth` cluster. Submit by running the normal CS166 submit script.

Problem Two: Dynamic Prefix Parity

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

- `initialize`(n), which creates a new data structure for an array of n bits, all initially 0;
- *ds.*`flip`(i), which flips the ith bit; and
- *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 `initialize` taking O(n) time such that `flip` runs in time O(1) and `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. In this problem, you'll work through a series of smaller milestones that culminates in a theoretically optimal result.

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 `initialize` runs in time O(n) and both `flip` and `prefix-parity` run in time O(log n). Argue correctness and justify your runtime bounds.

   Some things to think through as you do this:

   - You have the luxury of working with a tree where insertions and deletions will never happen; the number of items is specified in `initialize(n)` and never changes after that. Therefore, provided you can give initial values to any extra information you're caching at each node, and provided that you can update that information in O(log n) total time after a `flip` operation, you can meet the required time bounds.
   - The order statistics tree works because each node only needed to store information about what was to its left. That worked because rank queries purely care about the node and its left subtree. See if you can use that idea here as well.
   - A powerful technique we haven't yet encountered, but which might be useful to you here: consider making your tree such that only the leaf nodes store actual data, with all the internal nodes just serving as a way to join smaller subtrees together and cache relevant data.
   - When designing an augmented tree, it often helps to first solve the problem on a static array using a divide-and-conquer algorithm. So consider doing the following: suppose you had an array of n bits. Could you design a divide-and-conquer algorithm for computing prefix parities that has the recurrence relation $T(n) = 2T(n / 2) + O(1)$? If so, you can often translate your idea into an augmented tree by caching, at each node in the tree, the value that would be returned by running that divide-and-conquer algorithm on the elements in that tree.

(Continued on the next page...)

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 `initialize` take time $O(n)$, `flip` take time $O(\log_b n)$, and `prefix-parity` take time $O(b \log_b n)$. Here, b is a tunable parameter. Argue correctness and justify your runtime bounds.

Some things to think about as you do this:

- In an order statistics tree, we store one piece of information per node, since of a node's two children we only needed to care about the left child. Now that you have a multiway tree where each node can have multiple children, you may want to store multiple pieces of extra information per node.
- In lecture, we saw an algorithm that builds a B-tree from a set of n sorted keys in time $O(n)$, annotating each node with a pointer to its parent. That algorithm can easily be modified to also store, for each node, which child it is of its parent (e.g. first child, third child, etc.) without changing the overall runtime. It can also be modified to build a B-tree with a particular set of n leaves in time $O(n)$. Feel free to use this algorithm as a black box.

iii. Using the Method of Four Russians, modify your data structure from part (ii) so that `initialize` still runs in time $O(n)$, but both `flip` and `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 $\log_x y = \log y / \log x$ 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? It turns out that this data structure is asymptotically optimal (it can only be improved by reducing constant factors, not its asymptotic runtimes) under the assumption that you have just a single processor and that memory is segmented into individual machine words of size $\Omega(\log n)$. Understanding how this lower bound works could make for an excellent final project!