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 Tuesday, May 7 at 2:30PM.

Problem One: Dynamic Overlap (8 Points)

Your task in this problem is to design and implement a data structure for the ***dynamic overlap problem***. In this problem, you'll design a data structure that keeps track of a set of intervals and supports queries of the form "how many of those intervals contain this particular value?"

Specifically, your task is to implement a data structure supporting the following operations:

- `construct`(), which creates a new, empty data structure;
- `insert`(*start*, *end*), which inserts the closed interval [*start*, *end*] into the data structure;
- `remove`(*start*, *end*), which removes the closed interval [*start*, *end*] from the data structure; and
- `intervals-containing`(*key*), which returns the number of intervals containing *key*.

The runtime of the `construct` should be O(1), and the runtime of the remaining three operations should each be O(log *n*), where *n* is the number of intervals present in the data structure.

Download the starter files for PS3 from `myth` at

`/usr/class/cs166/assignments/ps3`

and edit the `DynamicOverlap.h` and `.cpp` files with your answer.

In case you find it helpful, we've included an implementation of a balanced binary search tree in `Treap.h` and `Treap.cpp`. This is a balanced BST backed by a *treap*, a randomized data structure that we chose because it's fairly simple to code up and understand. You're welcome to do whatever you'd like with this data structure and these files – they're there for you to use however you see fit!

Some notes on this problem:

- Inserting multiple copies of the same interval is permissible. Each copy of the interval is treated as its own independent interval for the purposes of counting maximum overlap.
- The `remove` function should remove one copy of the interval [*start*, *end*] from the data structure, not all of them.
- You can assume that `remove` won't be called on an interval that doesn't exist in the data structure.
- Note that `intervals-containing` just needs to return *how many* intervals the key is contained in, not what those intervals are.
- Although technically a treap is a randomized data structure and therefore doesn't guarantee a worst-case time bound of O(log *n*) for its operations, for the purposes of this problem you can pretend that its guarantees are worst-case rather than average-case. We've provided you with a treap simply because it's a relatively simple balanced BST structure. Aside from the randomness in the treap, you should not use randomization in your solution.

To receive full credit, your code should compile with no warnings and should not have any memory errors. We'll test your code on the `myth` cluster. There's information about how to run the test driver in the `README` file.

Problem Two: Range Excision (2 Points)

Design and describe an algorithm that, given a red/black tree T and two values k_1 and k_2, deletes all keys between k_1 and k_2, inclusive, that are in T. Your algorithm should run in time O(log n + z), where n is the number of nodes in T and z is the number of elements deleted. You should assume that it's your responsibility to free the memory for the deleted elements and that deallocating a node takes time O(1).

Problem Three: Deterministic Skiplists (8 Points)

Although we've spent a lot of time talking about balanced trees, they are not the only data structure we can use to implement a sorted dictionary. Another popular option is the ***skiplist***, a data structure consisting of a collection of nodes with several different linked lists threaded through them.

Before attempting this problem, you'll need to familiarize yourself with how a skiplist operates. We recommend a combination of reading over the Wikipedia entry on skiplists and the original paper "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh (available on the course website). You don't need to dive too deep into the runtime analysis of skiplists, but you do need to understand how to search a skiplist and the normal (randomized) algorithm for performing insertions.

The original version of the skiplist introduced in Pugh's paper, as suggested by the title, is probabilistic and gives *expected* O(log *n*) performance on each of the underlying operations. In this problem, you'll use an isometry between multiway trees and skiplists to develop a fully-deterministic skiplist data structure that supports all major operations in *worst-case* time O(log *n*).

i. Briefly explain how to encode a multiway tree as a skiplist. Include illustrations as appropriate.

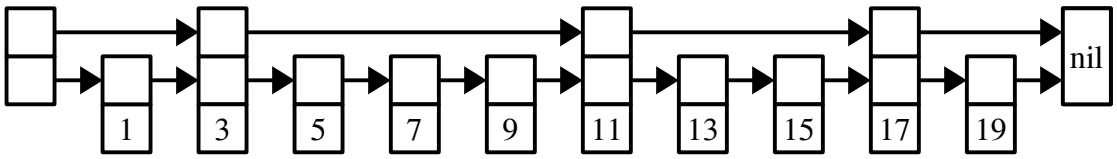
To design a deterministic skiplist supporting insertions, deletions, and lookups in time O(log *n*) each, we will enforce that the skiplist always is an isometry of a 2-3-4 tree.

ii. Using the structural rules for 2-3-4 trees and the isometry between multiway trees and skiplists you noted in part (i) of this problem, come up with a set of structural requirements that must hold for any skip list that happens to be the isometry of a 2-3-4 tree. To do so, go through each of the structural requirements required of a 2-3-4 tree and determine what effect they will have on the shape of a skiplist that's an isometry of a 2-3-4 tree.

Going forward, we'll call a skiplist that obeys the rules you came up with in part (ii) a ***2-3-4 skiplist***.

iii. Briefly explain why a lookup on a 2-3-4 skiplist takes worst-case O(log *n*) time.

iv. Based on the isometry you found in part (i) and the rules you developed in part (ii) of this problem, design a deterministic, (optionally amortized) O(log *n*)-time algorithm for inserting a new element into a 2-3-4 skiplist. Demonstrate your algorithm by showing the effect of inserting the value 8 into the skiplist given below:



Congrats! You've just used an isometry to design your own data structure! If you had fun with this, you're welcome to continue to use this isometry to figure out how to delete from a 2-3-4 skiplist or how to implement split or join on 2-3-4 skiplists as well.

Problem Four: Dynamic Prefix Parity (10 Points)

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:

- `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 subarray. (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, 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 `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.

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($\log_k n$), and `prefix-parity` take time O($k \log_k n$). 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 (i) 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:

   - It might help to 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. That is, 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?