Problem Set 4: Amortized Efficiency

This problem set is all about amortized efficiency and how to design powerful data structures that fit into that paradigm. You'll get to play around with the data structures we saw in lecture (two-stack queues, lazy binomial heaps, Fibonacci heaps, and splay trees), plus a few others from earlier. By the time you've finished this problem set, you'll have an excellent handle on how amortization works and how to think about problem-solving in a new way.

Due Tuesday, May 21st at 2:30PM.

Problem One: Stacking the Deque (3 Points)

A *deque* (***d***ouble-***e***nded ***que***ue, pronounced "deck") is a data structure that acts as a hybrid between a stack and a queue. It represents a sequence of elements and supports the following four operations:

- *deque*.`add-to-front`(*x*), which adds *x* to the front of the sequence.
- *deque*.`add-to-back`(*x*), which adds *x* to the back of the sequence.
- *deque*.`remove-front`(), which removes and returns the front element of the sequence.
- *deque*.`remove-back`(), which removes and returns the last element of the sequence.

Typically, you would implement a deque as a doubly-linked list. In functional languages like Haskell, Scheme, or ML, however, this implementation is not possible. In those languages, you might instead implement a deque using three stacks.

Design a deque implemented on top of three stacks. Each operation should run in amortized time O(1). For the purposes of this problem, do not use any auxiliary data structures except for these stacks.

Problem Two: Very Dynamic Prefix Parity (4 Points)

On Problem Set Three, you designed a data structure for the ***dynamic prefix parity problem***. If you'll recall, this data structure supports the following operations:

- `initialize`(*n*), which creates a new data structure representing an array of *n* bits, all initially zero. This takes time O(*n*).
- *ds*.`flip`(*i*), which flips the *i*th bit of the bitstring. This takes time O(log *n* / log log *n*).
- *ds*.`prefix-parity`(*i*), which returns the parity of the subarray consisting of the first *i* bits of the array. (the *parity* is 0 if there are an even number of one bits and 1 if there are an odd number of 1 bits). This has the same runtime as the `flip` operation, O(log *n* / log log *n*).

Now, let's consider the ***very dynamic prefix parity problem***. In this problem, you don't begin with a fixed array of bits, but instead start with an empty sequence. You then need to support these operations:

- *ds*.`append`(*b*), which appends bit *b* to the bitstring.
- *ds*.`flip`(*i*), which flips the *i*th bit of the bitstring.
- ds.`prefix-parity`(i), which returns the parity of the subarray consisting of the first i bits of the array.

Design a data structure for the very dynamic prefix parity problem such that all three operations run in *amortized* time O(log *n* / log log *n*), where *n* is the number of bits in the sequence at the time the operation is performed.

Problem Three: Palos Altos (3 Points)

The order of a tree in a Fibonacci heap establishes a lower bound on the number of nodes in that tree (a tree of order *k* must have at least F_{k+2} nodes in it). Surprisingly, the order of a tree in a Fibonacci heap does *not* provide an upper bound on how many nodes are in that tree.

Prove that for any natural numbers $k > 0$ and m, there's a sequence of operations that can be performed on an empty Fibonacci heap that results in the heap containing a tree of order *k* with at least *m* nodes.

Problem Four: Meldable Heaps with Addition (12 Points)

Meldable priority queues support the following operations:

- `new-pq`(), which constructs a new, empty priority queue;
- *pq*.`insert`(*v*, *k*), which inserts element *v* with key *k*;
- *pq*.`find-min`(), which returns an element with the least key;
- *pq*.`extract-min`(), which removes and returns an element with the least key; and
- `meld`(pq_1, pq_2), which destructively modifies priority queues pq_1 and pq_2 and produces a single priority queue containing all the elements and keys from pq_1 and pq_2.

Some graph algorithms, such as the Chu-Liu-Edmonds algorithm for finding the equivalent of a minimum spanning tree in directed graphs (an *optimum branching*), also require the following operation:

- *pq*.`add-to-all`(Δk), which adds Δk to the keys of each element in the priority queue.

Using lazy binomial heaps as a starting point, design a data structure that supports all `new-pq`, `insert`, `find-min`, `meld`, and `add-to-all` in amortized time O(1) and `extract-min` in amortized time O(log *n*).

Some hints:

1. You may find it useful, as a warmup, to get all these operations to run in time O(log *n*) by starting with an *eager* binomial heap and making appropriate modifications. You may end up using some of the techniques you develop in your overall structure.
2. Try to make all operations have worst-case runtime O(1) except for `extract-min`. Your implementation of `extract-min` will probably do a lot of work, but if you've set it up correctly the amortized cost will only be O(log *n*). This means, in particular, that you will only propagate the Δk's through the data structure in `extract-min`.
3. If you only propagate Δk's during an `extract-min` as we suggest, you'll run into some challenges trying to `meld` two lazy binomial heaps with different Δk's. To address this, we recommend that you change how `meld` is done to be even lazier than the lazy approach we discussed in class. You might find it useful to construct a separate data structure tracking the `meld`s that have been done and then only actually combining together the heaps during an `extract-min`.
4. Depending on how you've set things up, to get the proper amortized time bound for `extract-min`, you may need to define a potential function both in terms of the structure of the lazy binomial heaps and in terms of the auxiliary data structure hinted at by the previous point.

As a courtesy to your TAs, please start off your writeup by giving a high-level overview of how your data structure works before diving into the details.

Problem Five: Splay Trees in Practice (8 Points)

In lecture, we talked about a number of wonderful theoretical properties of splay trees. How well do they hold up in practice? In this problem, you'll find out!

Download the assignment starter files from

```
/usr/class/cs166/assignments/ps4/
```

and extract them somewhere convenient. There are a number of files there, of which you should only need to edit the `SplayTree.h` and `SplayTree.cpp` file. These starter files contain a partial implementation of a `SplayTree` type. Finish the implementation by implementing the `contains` member function. This should perform a lookup and, since this is a splay tree, perform a splay.

In class we presented bottom-up splaying where, after each query is performed, we applied a series of rotations from the queried node up to the root. This approach is a messy in practice because we need to keep track of the access path. There are a few ways to do this, all of which have some problems:

- We could have each node store a pointer back up to its parent. This increases the overhead associated with each node, which impacts cache-friendliness and reduces the maximum size of the trees that we can hold in memory. Plus, it requires additional pointer reassignments during rotations.
- We could implement splaying recursively, using the call stack to remember the nodes we've visited. This can cause problems with stack overflows if the tree gets too imbalanced and we do an extremely deep lookup.
- We could maintain a dynamically-allocated list of the nodes that we visited during a lookup. This requires extra memory allocations per lookup, which slows down the implementation.

In your implementation, we'd like you to use the ***top-down splaying*** procedure described in Section 4 of Sleator and Tarjan's paper on splay trees. Top-down splaying works by reshaping the tree during the top-down tree search and requires only O(1) auxiliary storage space.

You'll need to do a little legwork to figure out top-down splaying from Sleator and Tarjan's description. As with most research papers, the provided pseudocode skips over some C++-level details, which you will need to figure out by looking at the diagrams and through thorough testing. Don't just translate the code directly; draw some pictures to make sure you understand how top-down splaying works and what, conceptually, it's trying to accomplish. You might want to look at the operational description of the rotate-to-root heuristic as a starting point for understanding how top-down splaying has the same result as bottom-up splaying.

There are two versions of top-down splaying presented in Section 4. Implement the full rather than the simplified version of top-down splaying, since the simplified version is slower in practice.

You can test your implementation by running `./explore`. Once you've implemented everything, edit the `Makefile` to crank the optimizer up to maximum (set the flags `-Ofast -march=native`), rebuild everything, and run `./run-tests`. Those tests include sequential and reverse sequential access (great for trees with the dynamic finger property), tests doing lookups focused on small batches of elements (great for trees with the working set property), tests where lookups tend to be close to previous lookups (also great for trees with the dynamic finger property), tests over uniform distributions (great for trees with the balance property), and tests over Zipfian distributions (great for trees with the entropy property). Do those numbers surprise you? See anything interesting there?

In case you haven't seen Zipfian distributions before, we recommend checking them out on Wikipedia – they arise naturally in a bunch of places. Higher values of the parameter correspond to more and more skewed distributions.