Problem Set 4: Amortization

This problem set is all about amortized efficiency and how to design powerful data structures that fit into that paradigm. In this problem set and the related individual assessment, you'll get to play around with the data structures we saw in lecture, 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 Thursday, May 12th at 3:15PM Pacific.
Problem One: Very Dynamic Prefix Parity

On Problem Set Two, you designed a data structure for the dynamic prefix parity problem. 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 \texttt{flip} operation, \( O(\log n / \log \log n) \).

Now, consider the very dynamic prefix parity problem (or VDPP). 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:

- **initialize()**, which constructs a new, empty VDPP structure.
- **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 VDPP 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.

Briefly argue correctness, and prove that you meet the required amortized time bounds by defining a potential function \( \Phi \) and working out the amortized costs of each operation. We’re expecting your amortized analysis to be written up in a manner similar to the formal analyses we did in lecture of the two-stack queue, dynamic array, and B-tree construction algorithm.
Problem Two: Meldable Heaps with Addition

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 also require the following operation:

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

Although meldable priority queue have `n` nodes in them, it’s possible to implement `add-to-all` without touching all of these nodes.

i. Show how to modify an eager (non-lazy) tournament heap so that `new-pq`, `insert`, `find-min`, `extract-min`, `meld`, and `add-to-all` each run in worst-case `O(log n)` time. Briefly argue correctness.

A note on this problem: if it weren’t for `meld`, you could support `add-to-all` in time `O(1)`. (Do you see why?) We recommend that, from the beginning, you think about how you’d `meld` together two different heaps that have had different Δk’s added to them.

ii. Show how to modify a lazy tournament heap so that `new-pq`, `insert`, `find-min`, `meld`, and `add-to-all` all run in amortized time `O(1)` and `extract-min` runs in amortized time `O(log n)`. Briefly argue correctness, then do an amortized analysis with the potential method to prove runtime bounds. Some hints:

1. 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`.

2. If you only propagate Δk’s during an `extract-min` as we suggest, you’ll run into some challenges trying to `meld` two lazy tournament 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 melds that have been done and then only actually combining together the heaps during an `extract-min`.

3. Depending on how you set things up, to get the proper amortized time bound for `extract-min`, you may need to define your potential function both in terms of the structure of the lazy tournament heaps and in terms of the auxiliary data structure hinted at by the previous point.

In your writeup, don’t just describe the final data structure all at once. Instead, walk us through the design. Explain why each piece is there, why it’s needed, and how the whole structure comes together. Briefly argue correctness, and prove that you meet the required amortized time bounds by defining a potential function `Φ` and working out the amortized costs of each operation.

We’re expecting your amortized analysis to be written up in a manner similar to the formal analyses we did in lecture of the two-stack queue, dynamic array, and B-tree construction algorithm.