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 Tuesday, May 19th at 2:30PM Pacific.
Problem One: Stacking the Deque

A deque (double-ended queue, 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.

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. Briefly argue correctness. Then, perform two different amortized analyses to show that you meet the runtime bounds, one using the banker’s method and one using the potential method. For the banker’s method, if an operation places down credits, tell us where those credits are placed, and if an operation spends credits, tell us which specific credits it spends and why those credits are guaranteed to be there. For the potential method, clearly define your potential $\Phi$ and explain why your potential begins at zero and is nonnegative.

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(\Delta k)`, which adds $\Delta k$ to the keys of each element in the priority queue.

Design a data structure that supports `new-pq`, `insert`, `find-min`, `meld`, and `add-to-all` in amortized time $O(1)$ and `extract-min` in amortized time $O(\log n)$. Briefly argue correctness, then do an amortized analysis with either the banker’s method or the potential method to prove runtime bounds. Some hints:

1. As a warmup, get all these operations to run in worst-case time $O(\log n)$ by starting with an eager binomial heap and making appropriate modifications. Your ultimate data structure will likely be based on lazy binomial heaps, but starting eager may give you some useful insights for later.

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 $\Delta k$’s through the data structure in `extract-min`.

3. If you only propagate $\Delta k$’s during an `extract-min` as we suggest, you’ll run into some challenges trying to `meld` two lazy binomial heaps with different $\Delta k$’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`.

4. Depending on how you set things up, to get the proper amortized time bound for `extract-min`, you may need to define a potential function or place credits 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.

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.