Problem Set 4: Amortized Efficiency

In this problem set, you'll get to explore amortized efficient data structures and a number of their properties. You'll design a few new data structures and explore some of the ones from lecture in more detail. By the time you're done, you'll have a much deeper understanding of amortized efficiency and just why all these data structures are so nifty.

Due Thursday, May 12th at the start of lecture.

Problem One: Stacking the Deque (2 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 that 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 would instead implement a deque using three stacks.

Design a deque that is implemented on top of three stacks. Each of the above operations should run in amortized time O(1).

Problem Two: Palos Altos (2 Point)

Prove that for any positive integers *k* and *m*, there is a series of operations that, starting with an empty Fibonacci heap, causes the heap to have a tree of order *k* containing at least *m* nodes. This shows that, unlike the trees in binomial heaps, there is no upper bound on the number of nodes in a tree of any order in a Fibonacci heap.

Problem Three: Meldable Heaps with Addition (5 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;
- ***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 minimum spanning trees in directed graphs (often called "optimimum branchings"), 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. To get the proper amortized time bound for ***extract-min***, you will probably 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.

Problem Four: Static Optimality (5 Points)

In Problem Set Two, you explored weight-balanced trees in the context of trie construction. Interestingly, weight-balanced trees can also be used as nearly-statically-optimal binary search trees. As mentioned in lecture, if the weights assigned are the access probabilities of the keys, weight-balanced trees are never more than a factor of 1.5 off of the optimal static BST for those probabilities.

Here's a fast algorithm for building weight-balanced trees. In time $O(n)$, compute the total sum of the weights. Then, use the following recursive process:

- Scan from both ends of the array inward, summing up the weights as you go, until a node is found with the optimal balance between its left and right subarrays. You can detect this by looking for a spot where the absolute value of the weight difference increases; the spot right before that is the optimal splitting point. (Note that not all array elements will necessarily be scanned.)
- Make that node the root of the tree, and repeat this process on the halves of the array, passing down to the recursive calls the total weights of each subtree using the initial weight information and the relative weights discovered in the scan.

This algorithm somewhat resembles quicksort, so it's tempting to jump to the claim that it runs in time $\Theta(n \log n)$ in the best case and $\Theta(n^2)$ in the worst-case. However, this algorithm is much faster than that.

i. Prove that the above algorithm runs in worst-case time $O(n \log n)$.

In lecture, we talked about a number of wonderful theoretical properties of splay trees. How well do they hold up in practice? How do they compare against standard library tree implementations and against weight-balanced trees? Your task is to determine this for yourself.

ii. Download the assignment starter files from

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

and implement the `PerfectlyBalancedTree`, `WeightBalancedTree` and `SplayTree` classes. The `PerfectlyBalancedTree` type represents an optimally-balanced BST (that is, a tree that's as balanced as possible irrespective of the access probabilities). Some hints and advice:

1. Make sure not to recompute the sum of the weights in the subarrays at each level of the recursion, since otherwise the runtime might degrade to $\Theta(n^2)$ on some inputs.
2. Your implementation of splay trees should also never result in a stack overflow. To ensure this, you should implement *top-down splaying*, a variation of splaying described in Sleator and Tarjan's original paper (see Section 4), which uses $O(1)$ stack space. As with most research papers, the provided pseudocode skips a few key details, which you will need to figure out by looking at the diagrams and through thorough testing.
3. Once you've gotten your implementation working correctly, disable debugging and crank the optimization level all the way up by replacing `-O0 -g` in the Makefile with `-O3` and doing a clean build. Take a look at the times you get back – is that what you expected?

The "working set" test in the test cases uses an access pattern in which the input is broken apart into some fixed number of blocks. At each point in time there's an "active" block and all queries are directed at elements in that block. The net access pattern is uniformly random, but at any point in time it's highly skewed toward just a few elements.