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, 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 16th at noon Pacific.

Problem One: Stacking the Deque

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 six operations:

- *deque*.`push_front`(*x*), which adds *x* to the front of the sequence.
- *deque*.`push_back`(*x*), which adds *x* to the back of the sequence.
- *deque*.`front`(), which returns (but does not remove) the front element of the sequence.
- *deque*.`back`(), which returns (but does not remove) the last element of the sequence.
- *deque*.`pop_front`(), which removes (but does not return) the front element of the sequence.
- *deque*.`pop_back`(), which removes (but does not return) the last element of the sequence.

Your goal in this problem is to design a deque using three stacks such that each operation runs in amortized time O(1), under the assumption that each operation on a stack takes time O(1). You may not use any auxiliary or helper data structures other than these stacks. You'll do this in two steps. First, you'll design a data structure meeting these time bounds and code up your implementation. Next, you'll write up an analysis of the three-stack deque to explain why each operation takes amortized time O(1).

i. Copy the starter files for PS4 from `myth` from

`/usr/class/cs166/assignments/a4`

to a local directory of your own choosing, then edit `ThreeStackDeque.h` and `.cpp` with your solution. Your implementations of `front`, `back`, `pop_front`, and `pop_back` should throw exceptions of type `std::out_of_range` if the deque is empty.

In C++, stacks are represented by the type `std::stack<T>`, which is defined in the `<stack>` header. The operations on stacks that you'll need to use are

- *stack*.`size`(), which returns the size of the stack;
- *stack*.`empty`(), which returns whether the stack is empty;
- *stack*.`push`(*x*), which pushes onto the stack;
- *stack*.`top`(), which returns the top element of the stack; and
- *stack*.`pop`(), which pops the stack (but does not return the top element).

Make sure your solution compiles without warnings (`-Wall -Werror -Wpedantic`), runs cleanly under `valgrind`, and is commented so beautifully that it could be a museum piece. Note that running a program under `valgrind` markedly slows it down, so don't worry if you're failing the time tests when `valgrind` is engaged.

ii. Give a brief description of your data structure in plain English, the way you'd write up a solution to a non-coding question. Then, define a potential function Φ and use the potential method to argue that the amortized cost of each of the six operations on your three-stack deque is O(1). You do ***not*** need to argue correctness. 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: 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 `flip` operation, O(log *n* / log log *n*).

Now, consider the ***very dynamic prefix parity problem*** (or VDDP). 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 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.

Problem Three: Palos Altos

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 positive natural numbers *k* and *m*, there's a sequence of operations that can be performed, starting on an empty Fibonacci heap, so that the result Fibonacci heap's collection of trees includes a tree of order *k* that has at least *m* nodes. (This shows that there is no upper bound on the number of nodes in a tree of a given order; for any number *m*, you can force the tree to have at least *m* nodes.)

Illustrate the sequence of operations you get with $k = 2$ and $m = 8$ and show the resulting tree. You don't need to draw out each step, but should provide enough detail to convince us (and yourself!) that your sequence indeed produces the correct tree.

Problem Four: 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) binomial heap so that `new-pq`, `insert`, `find-min`, `extract-min`, and `add-to-all` each run in worst-case O(log n) time and the `meld` operation runs in time O(log m + log n), where m and n are the sizes of the respective heaps. 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 binomial 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 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`.
   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 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 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.