Individual Assessment 4: Amortization

This is an individual assessment, and, as the name suggests, must be completed individually. Specifically, you're not allowed to work with a partner, and you should not discuss these problems with other students in CS166. However, the course staff are happy to answer clarifying questions on Ed-Stem (if you do, please post the question privately) or in our office hours.

Due Thursday, May 12th at 3:15PM Pacific.
Problem One: Abdication Heaps

Abdication heaps have a number of moving parts that can be tricky to catch the first time around. This problem explores those details to make sure you’re solid on the data structure and its analysis.

i. An extract-min operation in an abdication heap has two steps. First, we perform the standard tournament heap extract-min procedure. Next, we remove the crowns from all the overtall trees. Prove that after the first step – right before we remove crowns from overtall trees – there are at most $O(\log n)$ trees in the heap.

ii. We picked $\Phi = T + 2z(\alpha)B$. Where would the amortized analysis go wrong if we dropped the leading $2z(\alpha)$ coefficient on the $B$ term? Be specific.

Problem Two: 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 six operations:

- $\text{deque.push_front}(x)$, which adds $x$ to the front of the sequence.
- $\text{deque.push_back}(x)$, which adds $x$ to the back of the sequence.
- $\text{deque.front}()$, which returns (but does not remove) the front element of the sequence.
- $\text{deque.back}()$, which returns (but does not remove) the last element of the sequence.
- $\text{deque.pop_front}()$, which removes (but does not return) the front element of the sequence.
- $\text{deque.pop_back}()$, 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 code up an implementation of your three-stack deque. 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 IA4 from myth from

```bash
/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

- $\text{stack.size}()$, which returns the size of the stack;
- $\text{stack.empty}()$, which returns whether the stack is empty;
- $\text{stack.push}(x)$, which pushes onto the stack;
- $\text{stack.top}()$, which returns the top element of the stack; and
- $\text{stack.pop}()$, 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 $\Phi$ 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.