Amortized analysis of 2-3-4 trees (50 points)

Recall that in a 2-3-4 tree, whenever we insert a new key we immediately split (on our way down the tree) any node we see that is full (has 3 keys in it).

(a) (15 points) Show that in the worst case, the number of splits that we do in a single operation can be $\Omega(\log n)$. In other words, show that for there is a series of $n$ inserts so that the next insert causes $\Omega(\log n)$ splits. You can be informal here — just explain at a high level what such a sequence would look like and why it would result in $\Omega(\log n)$ splits.

Let’s try to get around this worst-case bound by using amortized analysis. We will try to prove that the amortized number of splits is only $O(1)$. Note that this does not mean that the amortized running time of an insert is $O(1)$ (since this is not true); it just means that the amortized number of splits is $O(1)$. So think of “cost” not as “time”, but as “number of splits”. Since we didn’t talk about them in class, feel free to assume there are no delete operations.

(b) (15 points) Since we only have to split a node when it’s full, a natural approach is to have a bank on every node which works as follows. When a node is first created its bank is initialized to 0. When a node becomes full, we add 1 to its bank. Then when we split a node we use the 1 in its bank to pay for the split. In other words, a node’s bank is 1 if the node is full and 0 otherwise.

This argument unfortunately does not work. Explain why this banking scheme does not imply that the amortized number of splits is $O(1)$.

(c) (20 points) Show how to modify this argument to work. That is, use a bank at each node to prove that the amortized number of splits is $O(1)$. Hint: the previous part shows that this bank cannot just equal 1 if it is full and 0 otherwise.

2 Stacks and Queues (50 points)

Recall that a stack is a LIFO (last-in-first-out) data structure which supports the following operations: $\text{Push}(x)$ adds $x$ to the top of the stack, and $\text{Pop}()$ removes and returns the element on top of the stack. A queue is a FIFO (first-in-first-out) data structure which supports the following operations: $\text{Push}(x)$ adds $x$ to the queue, and $\text{Pull}()$ removes and returns the oldest element in
the queue. (See Section 10.1 of CLRS if you are not familiar with these: note that I have renamed
Enqueue to Push, and Dequeue to Pull).

Suppose that we have access to two stacks \(A\) and \(B\). Doing a push or a pop on either \(A\) or \(B\) costs
1 unit of time. We now implement a queue using \(A\) and \(B\) as follows:

\[
\begin{align*}
\text{Push}(x): \\
& 1. \quad A.\text{Push}(x) \\
\text{Pull}(): \\
& 1. \quad \text{If } B \text{ is nonempty, return } B.\text{Pop}() \\
& 2. \quad \text{Otherwise, pop each element from } A \text{ and push it into } B, \text{ except return the final element of } A \text{ (the last one popped) rather than push it onto } B.
\end{align*}
\]

(a) (15 points) Prove that this algorithm is correct, i.e. it does implement a Queue. Hint: think
about which invariants are maintained by this algorithm.

(b) (15 points) Use a potential function argument to prove that the amortized runtime of Push
and Pull is \(O(1)\).

Now suppose that you want more than a queue: you want a quack, which is both a queue and
a stack. That is, you want to support the following operations:

- **QuackPush\((x)\)**: add \(x\) to the quack.
- **QuackPop()**: remove and return the newest (most recently added) item from the quack.
- **QuackPull()**: remove and return the oldest item from the quack.

In order to do this, you are given a little bit more flexibility: you are given a third stack \(C\), where
the cost of pushing or popping on \(C\) is also 1 unit of time.

(c) (20 points) Show how to implement a quack using these three stacks so that the amortized
time of any of the three quack operations is \(O(1)\). As always, prove both correctness and
running time.