CpSc 8400: Design and Analysis of Algorithms

Instructor: Dr. Brian Dean — Spring 2014
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 11:00-12:15
Handout 9: Quiz #1 Solutions — Vickery 100

1. Common Elements (4 points). You are given two n-element balanced binary search trees (AVL trees) as input. Please describe a fast comparison-based algorithm for deciding whether there is any key that appears in both trees.

Perform an in-order traversal of both trees ($O(n)$ time), then merge these sorted sequences into one sorted sequence ($O(n)$ time). Scan through and look for consecutive pairs of equal elements ($O(n)$ time).

2. Recurrences (1+1+1 points). Please give asymptotic $\Theta(\cdot)$ solutions to the following recurrences. As a base case for each one, $T(n) = O(1)$ if $n = O(1)$.

$T(n) = 2T(n/2) + \Theta(\log n)$ **Solution:** $T(n) = \Theta(n)$.
$T(n) = 3T(n/2) + \Theta(\log n)$ **Solution:** $T(n) = \Theta(n^{\log_2 3})$.
$T(n) = 2T(n/3) + \Theta(\log n)$ **Solution:** $T(n) = \Theta(\log^2 n)$.

3. Min-Quack (8 points). Recall from lecture that we can easily build a *min-stack* supporting the operations *push*, *pop*, and *find-min* (which reports but does not remove the minimum element in structure) all in $O(1)$ worst-case time. In this problem, we use min-stacks to build a more powerful data structure which we call a *min-quack*.

We all know that a queue is a sequential data structure supporting insertion at one end and removal from the other, and a stack is a sequence supporting insertion and removal from the same end. A *quack*, as you might guess from its amusing name, is a combination of the two: it maintains a sequence of elements and supports the ability to insert or delete at *either* end. Accordingly, a quack must support the operations *insert-left*, *insert-right*, *delete-left*, *delete-right* which insert and delete elements on the left and right sides of the sequence. For example, if the quack holds the sequence "1 2 3 4 5" and we call *insert-right*(6), the quack would then hold "1 2 3 4 5 6". If we then called *delete-left*, the quack would hold "2 3 4 5 6".

We want to build a *min-quack*, supporting also a *find-min* operation. Please show how to use a pair of min-stacks to implement a *min-quack* where *insert-left*, *insert-right*, *delete-left*, *delete-right* and *find-min* all run in $O(1)$ amortized time. Use potential functions in your amortized analysis for full credit.

Use a pair of "back to back" min-stacks (call them A and B), just as with the min-queue we built in lecture. However, whenever we attempt to delete from an empty

min-stack, we spend linear time and "recenter" the two stacks, moving half the elements into each. Letting n_A and n_B denote the number of elements in A and B, we use $\phi = |n_A - n_B|$ be our potential function. Neglecting the cost of re-centering, an insertion or deletion takes 1 unit of actual time, plus at most 1 unit of increase to ϕ, for a total amortized cost of at most 2. Re-centering takes $n_A + n_B$ units of actual time, but this is accompanied by a drop in potential from $n_A + n_B$ to at most 1, so the amortized cost is at most 1. Hence, every operation has amortized cost at most 3.

4. Min-Quack, Part II (4 points). Please design a min-quack data structure (defined in the previous problem) where all five fundamental operations (*insert-left*, *insert-right*, *delete-left*, *delete-right* and *find-min*) take $O(\log n)$ worst-case time.

A standard double-ended queue (i.e., a "quack") is easy to implement using either a circular array or doubly-linked list, so that *insert-left*, *insert-right*, *delete-left*, and *delete-right* all take $O(1)$ worst-case time. We store our elements in this structure *and* also in a binary heap; we maintain pointers between the corresponding elements in both structures, so when we delete from the first structure, we can delete the corresponding element also from the heap. The heap supports *find-min* in $O(1)$ time and insertion / deletion in $O(\log n)$ worst-case time.

5. Housing Prices (6 points). There are n houses located at various locations along a one-dimensional street (think of this as a number line). For each house $i = 1 \ldots n$, you are told its location x_i, as well as its price p_i. The numbers $x_1 \ldots x_n$ are integers in the range $1 \ldots n^2$, and the prices $p_1 \ldots p_n$ are integers. A house i is *overvalued* if there exists another house of at most half its price within some distance D – that is, if there exists another house j such that $p_j \leq p_i/2$ and $|x_i - x_j| \leq D$. Given D, Please give a fast algorithm for counting the number of overvalued houses.

First sort the houses so $x_1 \leq \ldots \leq x_n$ in $O(n)$ time using radix sort. Then scan a window through this ordering (we can think of this as moving two pointers in tandem defining the left and right endpoints of the window, one keeping distance $-D$ from the current house and the other keeping distance $+D$). We store the houses within this window in a min-queue, and as we visit each house we compare its price p with the price p' of the minimum house in the queue, incrementing the number of overvalued houses if $p' \leq p/2$. Since we perform n insertions, n deletions, and n *find-min* calls in the min-queue, and each of these run in $O(1)$ time, the total time is $O(n)$.