CpSc 840: Design and Analysis of Algorithms

Instructor: Dr. Brian Dean — Spring 2013
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 11:00-12:15
Handout 8: Quiz #1 Solutions. — Lehostsky 246

Some of the following questions ask you to design an algorithm. Please do so *clearly* and also *concisely*, in English (pseudocode is generally not necessary). For each algorithm, please also briefly justify its running time. Standard results from class or the textbook can be used as "black boxes" without extra elaboration. Partial credit will be awarded for valuable insight or slightly slow but correct solutions, as long as they are not overly complicated. Unless otherwise stated, use the RAM model of computation. Be sure to keep an eye on the clock and pace yourself well. Good luck!

1. High Frequency (200,000 points). Given an array $A[1 \ldots n]$, a *high-frequency* element is a value x that occurs a maximum number of times in A. For example, if $A = [4, 6, 1, 2, 6, 1, 1, 3, 6]$, then both 1 and 6 are high-frequency elements, since no other elements have more occurrences. Give a fast comparison-based algorithm for computing a high-frequency element in A.

The simplest solution is to sort in $O(n \log n)$ time and then scan the array once, keeping track of the longest run of identical elements. Alternatively, we could insert the elements one by one into a balanced BST augmented with frequency counts, so that if we try to insert an element already present, we simply increment its frequency count.

2. High Frequency, Revisited (200,000 points). Argue that any comparison-based algorithm for solving the previous problem must take $\Omega(n \log n)$ time in the worst case.

An algorithm with running time faster than $O(n \log n)$ would violate the $\Omega(n \log n)$ worst-case lower bound for the element uniqueness problem, since all elements in A are unique if and only if a high-frequency element occurs only once in A.

3. Piles (200,000 points). Bored in CpSc840 lecture one day, Shiree draws a picture of a set of rectangles in her notebook.

Curiously, Shiree notes that if any two rectangles touch, then one is strictly contained within the other. Moreover, if two rectangles r and r' are contained within the same larger rectangle, then either r must be contained within r' or vice-versa. As a result, the rectangles form what Shiree calls "piles", where a pile is defined as a maximal chain of rectangles where each contains the next. The figure above consists of 4 piles of rectangles. Shiree would like to know the number of piles in her drawing. Help her by devising a fast algorithm that counts the number of piles in a set of n rectangles satisfying the conditions above. Each rectangle is specified by the coordinates of its corner points.

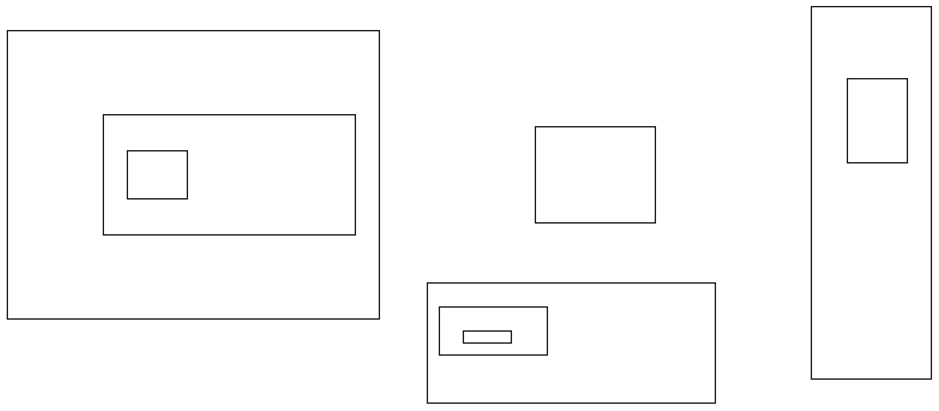

Figure 1: A collection of rectangles.

Use a sweep line approach: sort all the corner points on x, then sweep across the scene from left to right. When we reach the left edge of a rectangle, insert its two corner points into a y-ordered balanced BST; when we reach the right edge, we delete the two points from the BST. Further, before we insert the first point in any rectangle, we check if its predecessor and successor in the tree come from the same rectangle – if so, this point is part of a nested rectangle; if not, we increment a global count of the number of piles (so each pile is counted once when we process its outermost rectangle). Alternatively, the number of piles is equal to the number of innermost rectangles, which are the rectangles that have no other corner points inside them. So we can preprocess all $4n$ corner points into a 2D range tree with fractional cascading, and then use n 2D range counting queries, one per rectangle, to identify which ones are the innermost rectangles.

4. No Duplicates (200,000 points). Sakti is writing code to implement an AVL tree. Unfortunately, she has stayed up too late studying for a CpSc 840 quiz and in her tired state, accidentally modifies the *insert* routine as follows:

```
Insert(x):
  Call find(x) to check if x is already present in the structure
  If so, call delete(x) and then insert(x+1)
  Otherwise, insert the value x as in a normal AVL tree.
```

Sakti is concerned that a single call to *insert* might take a very long time, since it might cause a propagating chain of calls to *insert* in the worst case. For example, if 3, 4, and 5 are already present in the tree and *insert*(3) is called, then 3, 4, and 5 will be deleted and the value 6 will ultimately be inserted. However, please argue that the amortized running time of *insert* is only $O(\log n)$, where n is an upper bound on the number of elements that are ever present in the structure during its existence. For full credit, use potential functions; partial credit can still be obtained by using other methods of amortized analysis.

Let $T = O(\log n)$ denote the worst-case running time for an operation in our AVL

tree, and let $\phi = 2mT$, where m is the number of elements currently in the structure. Observe that ϕ satisfies the conditions required for a potential function: $\phi = 0$ initially, and $\phi \geq 0$ always. Now suppose we perform a call to *insert* that causes k finds, k deletions, and ultimately one insertion into the AVL tree. The actual cost of this operation is at most $(2k+1)T$, but the number of elements in the tree drops by $k-1$, so the amortized cost is at most $(2k+1)T + \Delta\phi = (2k+1)T - 2(k-1)T = 4T = O(\log n)$.

5. Stratified Heaps (200,000 points). A common misconception about heaps is that if x is an element on the level below element y, then $x \geq y$. This is certainly true if x is a child of y due to the heap property, but it is not true in general, as shown in the valid heap on the left, where the element 8 appears on the level below the element 12.

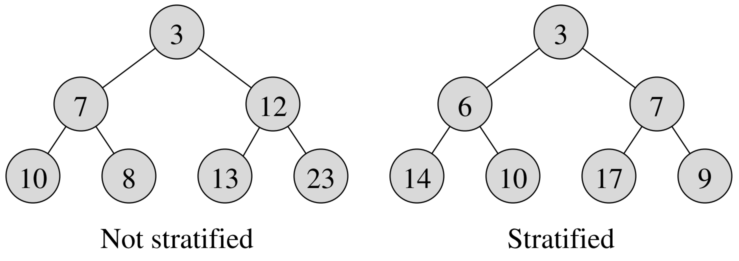

Figure 2: Examples of non-stratified vs. stratified heaps.

Let us call a heap *stratified* if it does satisfy the more stringent "heap property" above. In a stratified heap, all the elements on one level of the heap are no larger than all the elements in the next level of the heap. Since stratified heaps look harder to build than normal heaps due to this more demanding requirement, please either (i) show that you can still build a stratified heap in $O(n)$ time, *or* (ii) argue that in the comparison model, any algorithm for building a stratified heap must take $\Omega(n \log n)$ worst-case time.

We can build a stratified heap in $O(n)$ time using divide and conquer: for simplicity assume all elements are distinct, and that the bottom row of the heap is full (we can add dummy elements of infinite value to ensure this). Then find the median in $O(n)$ time and put all elements $\geq$ the median in the bottom row. On top of this bottom row, place a stratified heap that is recursively computed from the left-over elements. The running time satisfies $T(n) = T(n/2) + O(n)$, which solves to $T(n) = O(n)$.