Homework 4
CS 457

Damien Prieur

Question 1

You are given a red-black tree T with 15 internal nodes (nodes that hold key values) that form a *full* binary tree of height 3 (i.e., a full binary tree of height 4 if you include the NIL leaves). Can you assign colors to the nodes so that a call to RB-INSERT(T, z) for *any* new key value $z.key$ will cause RB-INSERT-FIXUP(T, z) to change the color of the root to red before switching it back to black? The initial assignment of colors needs to obey the red-black properties. If such a color assignment exists, then provide a sequence of 15 numbers whose insertion (in that order) would lead to such a tree, along with a figure of the resulting tree. If not, then explain why such an assignment cannot exist, using the fact that the tree needs to satisfy the red-black properties. You can use the applet `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html` and read through Chapter 13 from your textbook if you would like to better understand how red-black trees work.

Question 2

A full k-ary tree is a (rooted) tree whose nodes either have exactly k children (internal nodes) or have no children (leaves). *Using induction*, formally prove that every full k-ary tree that has x internal nodes has exactly $kx+1$ nodes in total. Note that for full *binary* trees, i.e., when $k = 2$, this would imply that the total number of nodes is $2x + 1$, which would verify what Cameron suggested in class (that the total number of nodes of full binary trees is always odd).

Question 3

Using proof by induction, show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap. Make sure to clearly state your base case and your inductive step arguments. Also note that you need to prove the aforementioned upper bound for the number of nodes of height h for *every* value of n and *every* n-element heap.

Question 4

You start a new tech business and you reach the point where you have offices both in the east coast and in the west coast. There are different operating costs involved in running your business in each location, so you may need to move back and forth across the coasts in order to minimize the total costs. However, there are moving costs involved in traveling from one coast to the other which complicate your problem. More formally, for each month $i = 1, \ldots, n$ you have an operating cost e_i associated with running your business in the east coast during that month and an operating cost w_i for the west coast. You can begin working in either coast, but after any subsequent travel from one coast to another you suffer a fixed travel cost of c. Given input $\{e_1, \ldots, e_n\}$, $\{w_1, \ldots, w_n\}$, and c, provide a $O(n)$ time algorithm that outputs a plan defining which coast you will work from each month, and minimizes your total cost (operating costs plus travel costs).

For example, if the number of months was $n = 5$, a possible plan could be to work from the east coast during the first two months, then spend the next two months working from the west coast, and return to the east coast for the fifth and final month. The total cost in this case would be $e_1 + e_2 + c + w_3 + w_4 + c + e_5$, where the two costs of c are due to the change of coast. Provide detailed pseudocode for your proposed algorithm, as well as an explanation regarding why it works and why its running time is indeed $O(n)$.

Question 5

You are planning to attend a music and arts festival that hosts multiple events that you are interested in, but many of these events may (partially) overlap. Assume that you get some value v_i from attending an event i, but you get this value only if you are there for the whole duration of the event; you get no value if you have to miss part of it. Given such a set of events, E, your goal is to choose a non-overlapping set of events, $E' \subseteq E$, to attend, aiming to maximize your total value. More formally, for each event $i \in E$ you have a start time s_i and a finish time f_i and you want to design an algorithm that runs in time $O(n \log n)$, where n is the total number of events, and the output of the algorithm is a set $E' \subseteq E$ such that for any two events $i, j \in E'$ we have $f_i \leq s_j$ or $f_j \leq s_i$ (no overlap) and the total value $\sum_{i \in E'} v_i$ is maximized. Provide detailed pseudocode for your proposed algorithm (you can use sorting as a subroutine), as well as an explanation regarding why it works and why its running time is indeed $O(n \log n)$.