COS 423 Optional Problem Set Due Tuesday May 10
Spring 2011
Tarjan

No collaboration. If you choose, you may turn this problem set in as a replacement for any single previous problem set. PLEASE INDICATE WHICH PREVIOUS PROBLEM SET YOU WISH TO REPLACE.

A *twin heap* is an implicit data structure consisting of a pair of heaps, one a min-heap, the other a max-heap, stored in a single two-dimensional array $H(i, j)$, with $i > 0$ and j in $\{0, 1\}$. The children of $H(i, j)$ are $H(2i, j)$ and $H(2i + 1, j)$; the parent of $H(i, j)$ is $H(i \text{ div } 2, j)$. The slots of H are filled in the order H(1,0), H(1, 1), H(2, 0), H(2, 1), H(3, 0), H(3, 1),… and emptied in the opposite order. The slots with $j = 0$ form a min-heap and those with $j = 1$ form a max-heap: $H(i, 0) \geq H(i \text{ div } 2, 0)$ and $H(i, 1) \leq H(i \text{ div } 2, 1)$. Your goal is to explore two different ways of tying the min-heap and the max-heap together.

1. A *min-max twin heap* is a twin heap with the additional constraint that $H(i, 0) \leq H(i, 1)$ for all i. Give algorithms for returning a minimum value, returning a maximum value, inserting a value, and deleting a value (given its position) in a min-max twin heap. Your algorithms for returning a minimum or maximum value should take O(1) time worst-case. Your algorithms for inserting or deleting a value should take $O(\lg n)$ time worst-case, where n is the number of values in the heap. Give the best bounds you can (including constant factors) for the number of binary comparisons between values required for each of these operations.

2. A *median twin heap* is a twin heap with the additional constraint that $H(1, 0) \geq H(1, 1)$ (and without the constraint in Problem 1). Give algorithms for returning the median of the values in the heap, for inserting a value, and for deleting a value (given its position) in a median twin heap. Your algorithm for returning the median should take O(1) time worst-case; your insertion and deletion algorithms should take $O(\lg n)$ time worst-case. Give the best bounds you can (including constant factors) for the number of binary comparisons between values required for each of these operations.

3. Consider the preflow push algorithm for computing a maximum flow (described in Lecture 21). A variant that we did not analyze is that of always choosing a vertex of highest label from which to do a push, or to label if such a push is not possible.

(a) Give an implementation of the highest-label selection rule such that the total running time of the algorithm with this rule is $O(nm)$ plus O(1) per non-saturating push.

(b) Give a careful and complete proof that with this rule the number of non-saturating pushes is $O(n^2m^{1/2})$. Make the constant factor in your bound as small as you can. Here is one way to obtain such a bound. Define the potential of an active vertex *v* to be the number of vertices (active or not) whose label is no higher than that of *v*. Define a *large push* to be a nonsaturating push that happens when there are at least *k* active vertices of maximum label; all other non-saturating pushes are *small*. Here *k* is a parameter whose value you are free to choose. Define a *phase* to consist of all pushes that take place between changes in the maximum of the labels of active vertices. Prove that the number of phases is $O(n^2)$ (get the best constant factor you can), that the number of small pushes per phase is at most *k*, and that any big push decreases the potential by at least *k*. Prove that the total increase in potential caused by labeling steps and saturating pushes is $O(n^2m)$ (with the best constant factor you can). Finally, combine these results with a good choice of *k* to get the desired bound on the number of non-saturating pushes.