CS 423: Theory of Algorithms
Spring, 2000 Due: Tuesday, April 4
Tarjan

**Extra Credit Problems**

Extra credit problems are intended to be a little more challenging than the regular problems. Please work on them entirely on your own (no collaboration). Scores on extra credit problems may be used to boost your grade or to earn an A+.

1. The purpose of this problem is to fix problem 4, problem set 3, which was taken from CLR, problem 22.3-4, page 450. Let us redefine the UNION operation so that it returns the value "true" if its inputs are initially in different sets and "false" otherwise.

   UNION $(x, y)$: If $x$ and $y$ are in different sets, combine these sets into one and return *true*; otherwise, merely return *false*.

   We call a UNION **successful** if it returns true and **unsuccessful** otherwise. Our goal is to build a data structure that supports intermixed MAKE-SET, UNION, and FIND-SET operations with the following amortized time bounds, where $n$ is the total number of UNION operations:

   MAKE-SET: $O(1)$

   UNION; $O(1)$ if successful, $O(\alpha(n))$ if not.

   FIND-SET: $O(1)$ if the operation occurs after all UNION operations, $O(\alpha(n))$ if not.

   Observe that these bounds imply that a sequence of $m$ operations in which all UNIONs are successful and come before all FIND-SETs takes $O(m)$. Compare this with the (false) original statement of problem 4, problem set 3.

   The data structure is the usual tree structure, with each node having a rank, initially zero for a node in a single-node tree. We perform FIND-SET in the usual way, with path compression.

   To perform UNION $(x, y)$, we traverse the find paths from $x$ and $y$ concurrently, taking a step from the current node of smaller rank (breaking a tie arbitrarily), until reaching a root on one path. Suppose the root reached is $r$, on the find path from $x$. We continue traversing the find path from $y$ until reaching a node $s$ that is either a root or has rank strictly larger than that of $r$. If $r = s$ the UNION is unsuccessful; we compress the paths from $x$ and $y$ to $r$ (making $r$ the parent of every node on both paths except itself) and return *false*. Otherwise, if **rank**$(r) <$ **rank** $(s)$, we make $s$ the parent of $r$, compress the paths from $x$ and $y$ to $s$, and return *true*; if **rank** $(r) \geq$ **rank**$(s)$ we increase **rank** $(r)$ by 1 if **rank** $(r) =$ **rank** $(s)$, we make $r$ the parent of $s$ ($s$ must be a root), we compress the paths from $x$ and $y$ to $r$, and we return *true*. This method relies on the existence of ranks, and in particular on maintaining the invariant that ranks strictly increase along find paths.

   (a) Give pseudo-code implementaion of UNION.

   (b) Prove the amortized bounds claimed above.

2. The purpose of this problem is to add insertions and deletions to linear heaps (see problem 2, mid-term exercise). Insertions are easy to handle, but deletions are much tougher.

(a) suppose we modify the **make heap** operation for linear heaps so that it creates a new, empty heap, and we add an operation **insert** $(i, a, b, h)$, which adds a new item $i$ with key $ax + b$ to linear heap $h$. Describe how to implement the linear heap operatoins **make heap**, **insert**, and **find min**, so that the worst-case times for the operations are $O(1)$ for **make heap** and $O(\log n)$ for an **insert** or **find min** on an $n$-time heap. **Hint:** Use a red-black tree to store the items, sorted on $a$, with ties broken on $b$, and items whose keys cannot be minimum deleted.

(b) Suppose we add to the operations in (a) the operation of deleting an item in a linear heap, given a pointer to the location in the data structure representing the item. We also impose a **progressivity** condition on the **find min** operations: successive $x$-parameters are non-decreasing. Describe and analyze an implementation of progressive linear heaps with the following amortized time bounds: $O(1)$ for **make heap**; $O((\log n)^2)$ for **insert**, **delete**, and **find min** on an $n$-item heap.

**Hint:** For simplicity, assume that no two items have identical keys. (Identical keys could be handled by storing all items with the same key in a list at a single node.) Store the items in the **external** nodes of a red-black tree, ordered as in (a). To facilitate searching, store in each internal node a pointer to its successor (external) node. Define $x_0$, the **current** $x$, to be the $x$-parameter of the most recent **find min** operation. Store at each internal node the item among its descendants with minimum key for $x = x_0$. Finally, for two items $i$ and $j$, define the **switch value** $x_{ij}$ to be the value of $x$ for which the keys of $i$ and $j$ are equal, and $-\infty$ if $i$ and $j$ have equal $a$-values (so their keys are never equal). Note that for $x < x_{ij}$ one of the keys will be smaller and for $x > x_{ij}$ the other key will be smaller. Store in each internal node the minimum amoung the switch times of all pairs of items stored in siblings that are its descendents. Use the switch times to help update the minimum-key items as $x_0$ increases.

**Comment:** Actually, it is possible to obtain $O((\log n)^2)$ time bounds in the worst case for **insert** and **delete** and $O(\log n)$ worst-case for **find min**, without imposing the progressivity condition but the data structure is more complicted thean the one described in the hint. The latter is a nice illustration of the use of both heap order and symmetric order in a single balanced tree. Others include **treaps**, which are randomized search trees, and **priority search trees**, which support $1\frac{1}{2}$-dimensional range queries.

**Research Problems:** (Solve one, and become an author or co-author on a research paper.)

2(c) Prove or disprove $O((\log n)^2)$ amortized bounds if a splay-tree is used in 2(b) instead of a red-black tree.

2(d) Find a data structure that reduces the amortized bounds in 2(b) to $O(\log n)$. (I know an $O(\log n \log\log n)$ solution.)