Problem Set 2

Due: Wednesday, September 23, 2011.

Collaboration policy: collaboration is strongly encouraged. However, remember that

1. You must write up your own solutions, independently.
2. You must record the name of every collaborator.
3. You must actually participate in solving all the problems. This is difficult in very large groups, so you should keep your collaboration groups limited to 3 or 4 people in a given week.
4. No bibles. This includes solutions posted to problems in previous years.

Problem 1. Devise a way to avoid initializing large arrays. More specifically, develop a data structure that holds $n$ items according to an index $i \in \{1, \ldots, n\}$ and supports the following operations in $O(1)$ time (worst case) per operation:

- **init** Initializes the data structure to empty.
- **set**$(i, x)$ places item $x$ at index $i$ in the data structure.
- **get**$(i)$ returns the item stored in index $i$, or “empty” if nothing is there.

Your data structure should use $O(n)$ space and should work regardless of what garbage values are stored in that space at the beginning of the execution. Hint: use extra space to remember which entries of the array have been initialized. But remember: the extra space also starts out with garbage entries!

Problem 2. Some splaying counterexamples.

(a) In class, I stated that single rotations “don’t work” for splay trees. To demonstrate this, consider a degenerate $n$-node “linked list shaped” binary tree where each node’s right child is empty. Suppose the (only) leaf is splayed to the root by single rotations. Show the structure of the tree after this splay. Generalizing, argue that there is a sequence of $n/2$ splays that each take at least $n/2$ work.

(b) Now from the same starting tree, show the final structure after splaying the leaf with (zig-zig) double rotations. Explain how this splay has made much more progress than single rotations in “improving” the tree.
(c) Given the theorem about access time in splay trees, it is tempting to conjecture that splaying does not create trees in which it would take a long time to find an item. Show that this conjecture is false by showing that for \( n \geq 4 \), it is possible to restructure any binary tree on \( n \) nodes into any other binary tree on \( n \) nodes by a sequence of splay operations. Conclude that it is possible to make a sequence of requests that cause the splay tree to achieve any desired shape. **Hint:** start by showing how you can use splay operations to make a specified node into a leaf; then recurse.

**NONCOLLABORATIVE Problem 3.** In this problem, we’re going to develop a less slick but hopefully more intuitive analysis of why splay trees have low amortized search cost. We need to argue that on a long search, all but \( O(\log n) \) of the work of the descent and of the follow-on splay is paid for by a potential decrease from the splay. We’ll use the same potential function as before—the sum of node ranks (multiplied by some constant). Also as before, we’ll analyze one double rotation at a time, and argue that the potential decrease for each double rotation cancels the constant work of doing that double rotation.

(a) As in class, a double rotation will involve 3 nodes on the search path: \( x \), its parent \( y \), and its grandparent \( z \). Call this triple **biased** if over \( 9/10 \) of \( z \)’s descendants are below \( x \), and **balanced** otherwise. Argue that along the given search path, there can be at most \( O(\log n) \) balanced triples.

(b) Argue that when a biased triple is rotated, the potential decreases by a constant, paying for the rotation. Do so by observing that rank of \( x \) only increases by a small constant, while the ranks of \( y \) or \( z \) decrease by a significantly larger constant. Do this for both the ZIG-ZIG and ZIG-ZAG rotations.

(c) Argue that when a balanced triple is rotated, the potential increases by at most \( 2(r(z) - r(x)) \) (again, consider both rotation types).

(d) Conclude that enough potential falls out of the system to pay for all the biased rotations, while the real work and amount of potential introduced by the balanced rotation is \( O(\log n) \), which thus bounds the amortized cost.

**Problem 4.** The slowest part of multi-level-bucket heaps was the scan for the next nonempty bucket in a block. Suppose that instead of maintaining the set of a block’s nonempty buckets in an array, you kept it in a standard binary heap (note this adds no data structural complexity, since we already have the necessary array handy). Discuss how the runtimes of operations would change. By balancing parameters, argue that you can implement a heap with amortized runtimes of \( O(\sqrt{\log C}) \) for insert, decrease key, and delete min, yielding a shortest paths algorithm with runtime \( O((m + n)\sqrt{\log C}) \). **Optional:** can you achieve a runtime of \( O(m + n\sqrt{\log C}) \) by tweaking this method?
**Problem 5.** Our Van Emde Boas construction gave a high-speed priority queue, but with a little more work it can turn into a high-speed “binary search tree” (which still supports find- and delete- min). Augment the van Emde Boas priority queue to track the maximum as well as the minimum of its elements, and use the augmentation to support the following operations on integers in the range \{0, 1, 2, \ldots, u - 1\} in $O(\log \log u)$ worst-case time each and $O(u)$ space total:

**Find** ($x, Q$): Report whether the element $x$ is stored in the structure.

**Predecessor** ($x, Q$): Return $x$’s predecessor, the element of largest value less than $x$, or null if $x$ is the minimum element.

**Successor** ($x, Q$): Return $x$’s successor, the element of smallest value greater than $x$, or null if $x$ is the maximum element.

**OPTIONAL Problem 6.** Prove the *dynamic optimality conjecture*: over any given access sequence, splay trees take time proportional to the best possible (pointer based) data structure for the problem, even if that data structure is allowed to adjust itself during accesses (the adjustment time counts toward the overall cost, of course). Partial credit will be given for proving special cases of the conjecture (for particular kinds of access sequences).

**OPTIONAL Problem 7.** Can a van Emde Boas type data structure be combined with some ideas from Fibonacci heaps to support insert/decrease-key in $O(1)$ time and delete-min in $O(\log \log u)$ time?

**Problem 8.** How long did you spend on this problem set?