CS 473: Undergraduate Algorithms, Spring 2009
Homework 5

Written solutions due Tuesday, March 9, 2009 at 11:59:59pm.

---

1. Remember the difference between stacks and queues? Good.

   (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard methods PUSH and POP.

   (b) A *quack* is an abstract data type that combines properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

   - **Push:** add a new item to the left end of the list;
   - **Pop:** remove the item on the left end of the list;
   - **Pull:** remove the item on the right end of the list.

   Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. Again, you are *only* allowed to access the stacks through the standard methods PUSH and POP.

2. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   Describe and analyze an algorithm to purge an *arbitrary* n-node dirty binary search tree in $O(n)$ time, using at most $O(\log n)$ space (in addition to the tree itself). Don't forget to include the recursion stack in your space bound. An algorithm that uses $\Theta(n)$ additional space in the worst case is worth half credit.

3. Some applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Maintaining these secondary structures usually complicates algorithms for keeping the top-level search tree balanced.

   Let T be an arbitrary binary tree. Suppose every node v in T stores a secondary structure of size $O(size(v))$, which can be built in $O(size(v))$ time, where $size(v)$ denotes the number of descendants of v. Performing a rotation at any node v now requires $O(size(v))$ time, because we have to rebuild one of the secondary structures.

   (a) **[1 pt]** Overall, how much space does this data structure use *in the worst case*?

   (b) **[1 pt]** How much space does this structure use if the primary search tree T is perfectly balanced?

   (c) **[2 pts]** Suppose T is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

(d) **[3 pts]** Now suppose T is a scapegoat tree, and that rebuilding the subtree rooted at v requires $\Theta(size(v) \log size(v))$ time (because we also have to rebuild the secondary structures at every descendant of v). What is the *amortized* cost of inserting a new element into T?

(e) **[3 pts]** Finally, suppose T is a treap. What's the worst-case *expected* time for inserting a new element into T?