Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 3
Wednesday, September 12, 2012

Problem Set 2

Due: Wednesday, September 19, 2012.

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. 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 large enough n, 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.

Problem 2. Let S be a search data structure (such as a red-black tree) that performs insert, delete and search in $O(\log n)$ time, where n is the number of elements stored. An empty data structure S can be created in $O(1)$ time.

We would like to construct a static data structure with n elements that is statically optimal in total access time, given the number of times an element is accessed in an access sequence. Recall that if item i is accessed $p_i m$ times in a sequence of m operations then the static-optimal access time is $O(m \sum p_i \log 1/p_i)$.

The data structure is constructed as follows. Search data structure S_k holds the 2^{2^k} most frequently occurring items in the access sequence. A search on v is done on $S_0, S_1, \ldots$ until an S_i holding v is encountered. Notice that all elements in S_i are held in S_{i+1}.

(a) Show that the above data structure is asymptotically comparable to the optimal static tree in terms of the total time to process the access sequence.

(b) Make the data structure capable of insert operations. Assume that the number of searches to be done on v is provided when v is inserted. The cost of insert should be $O(\log n)$ amortized time, and total cost of searches should still be optimal (non-amortized).

(c) Improve your solution to work even if the frequency of access is not given during the insert. Your data structure now matches the static optimality theorem of splay trees.

(d) Make your data structure satisfy the working set theorem on splay trees. Ignore the static optimality condition.

Problem 3. Describe a data structure that represents an ordered list of elements under the following three types of operations:

access(k)**:** Return the kth element of the list (in its current order).

insert(k, x)**:** Insert x (a new element) after the kth element in the current version of the list.

reverse(i, j) Reverse the order of the ith through jth elements.

For example, if the initial list is $[a, b, c, d, e]$, then access(2) returns b. After reverse(2,4), the represented list becomes $[a, d, c, b, e]$, and then access(2) returns d.

Each operation should run in $O(\log n)$ amortized time, where n is the (current) number of elements in the list. The list starts out empty.

Hint: First consider how to implement access and insert using splay trees. Then think about a special case of reverse in which the $[i, j]$ range is represented by a whole subtree. Use these ideas to solve the real problem. Remember, if you store extra information in the tree, you must state how this information can be maintained under various restructuring operations.

(This data structure is useful in efficiently implementing the Lin Kernighan heuristic for the travelling salesman problem.)

NONCOLLABORATIVE Problem 4. 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 5. 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?

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).

Problem 7. How long did you spend on this problem set?