Problem Set 2

Due: Wednesday, September 21, 2005.

Notice that one problem is marked **noncollaborative.** As you might expect, this problem should be done without any collaboration.

Problem 1. 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.

Problem 2. 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 (implying that there is some access sequence that turns a tree into a path).

Problem 3. Let S be a search data structure 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 will construct a static data structure with n elements that is worst-case optimal in total access time, given the number of times an element is accessed in an access sequence.

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. Recall from class that the statically optimal data structure achieves average access time $O(-\sum p_i \log p_i)$ where p_i is the fraction of accesses to item i.

(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 **worst case** 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 satisfies the same static optimality theorem as splay trees.

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

Problem 4. Worked example.

(a) Build an uncompressed suffix trie for "banana\$". Show the structure and node traversal path for each suffix insertion. Mark the suffix links that are actually used as shortcuts in the efficient construction algorithm.

(b) Draw the compressed suffix tree for "banana\$".

NONCOLLABORATIVE Problem 5. In this problem, we will see how to construct a suffix tree on multiple texts, and what some useful properties of such a suffix tree are. Suppose you are given n texts $T_1, T_2, \ldots T_n$.

(a) Suppose you build a common suffix tree of all the texts $T_1, T_2, \ldots T_n$, i.e., a trie that contains all the suffixes of all the n texts. Argue that you can do this in time $O(|T_1| + |T_2| + \ldots |T_n|)$. Be careful not to produce suffixes that cross from one text to another (which would happen if you simply concatenated all the texts).

(b) Suppose that we add a different unique terminating symbol $\$_i$ to each of the texts T_i. Consider a node N in the common suffix tree, and let s be the string corresponding to this node (i.e., the string on the path from the root to the node). How can you determine whether the string s is a substring of all the n texts by looking at the subtree rooted at N?

(c) Using the above approach, explain how you can find the largest common substring of the two texts T_1, T_2 in time $O(|T_1| + |T_2|)$ (there's a simple generalization to more texts).