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

Handout 4
September 21, 1999

Problem Set 2

Due: September 28, 1999.

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.

Problem 2. 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 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.

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 satisfies the static optimality theorem on 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. This is a good idea for a project in this course.)

Problem 4. Using the splay operation as a subroutine, implement the `successor` operation. This takes as input a pointer to the root of a tree, and restructures the tree so that the successor of the root becomes the root. The operation does not change the tree if the root is already the rightmost element of the tree. The new root is returned. You may make the assumption that every key in the tree is different.

(a) Explain why the amortized cost of your implementation is $O(\log n)$.

(b) Prove that the amortized running time of the successor operation is $O(1)$ if it is applied repeatedly until it finds the rightmost node in the tree. (Use a theorem that's stated in the JACM paper.)

* **Problem 5.** 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.

** **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).