The University of British Columbia

Department of Computer Science

Computer Science 500—Fundamentals of Algorithm Design and Analysis

Homework Assignment 3

Due: 2013 October 30

You are free to discuss problems with your fellow students. However, your actual homework submission must be prepared on your own. At the top of the first page of every homework submission you must provide a statement about external resources used in the preparation of your submission. This must clearly acknowledge all resources (including books, websites and discussions with fellow students) that you have used. Submissions missing this statement will not be graded.

1) We noted in class that $n-1$ comparisons are necessary and sufficient, in the worst case, to determine the maximum (or the minimum) element of an set of n numbers.

   a) Show that if we want to determine *both* the maximum and the minimum element, it suffices to use $\lceil 3n/2 \rceil - 2$ comparisons.

   b) We want to devise an *adversary strategy* to show that $\lceil 3n/2 \rceil - 2$ comparisons are also *necessary*, in the worst case.

      Our adversary will maintain the numbers (contestants) in four disjoint groups:

      Group A – contestants that have never played (and hence are still candidates for both the maximum and the minimum).

      Group B – contestants that have won one or more contests but have never lost (and hence are still candidates for the maximum).

      Group C – contestants that have lost one or more contests but have never won (and hence are still candidates for the minimum).

      Group D – contestants that have both lost and won at least once (and hence are no longer candidates for either the maximum or the minimum).

      The adversary ensures that group B contestants always win (against other types) and group C contestants always lose (against other types). Otherwise the outcome of contests is chosen arbitrarily.

      Show that, at any stage, an algorithm playing against this adversary is forced to make at least $f(A, B, C) = \lceil 3|A|/2 \rceil + |B| + |C| - 2$ more comparisons before it has determined the maximum and the minimum. (Hint: argue by induction on $f(A, B, C)$, considering all possible cases for the "next" comparison.)

   c) Argue that the lower bound ($\lceil 3n/2 \rceil - 2$) follows from part (b).

2) Suppose we are given a set S containing n elements drawn from the universe $\mathcal{U} = \{0, 1, \ldots, m-1\}$, and suppose that we are only able to make comparisons of the form "$x_i \geq c$?", where x_i is an element of S and c is some specified element of $\mathcal{U}$ (for example "$x_5 \geq 17$?").

   a) Show that $O(n\, lg\, |\mathcal{U}|)$ such *unary* predicate evaluations suffice to determine the maximum element of S, in the worst case.

b) Give the best (largest) lower bound that you can (expressed as a function of *both* n and $|\mathcal{U}|$) on the worst-case number of unary predicate evaluations needed to determine the maximum element of S.

3) Suppose that we are given $D[1:n]$, an array of real-valued keys sorted in increasing order, and we wish to use D as a *dictionary*, i.e. for each x_j in a sequence $x_1, x_2, \ldots, x_m$ of queries we want to return the location $\ell(j) \in \{1, \ldots, n\}$ of an element in $D[1:n]$ that minimizes $|D[\ell(j)] - x_j|$ (in which case we say that key $D[\ell(j)]$ has been *accessed*.)

It is not uncommon in certain applications that queries to a dictionary exhibit a kind of *locality of reference*: successive queries are frequently made to the same, or close to the same, key. In such situations, it is desirable to exploit this property to reduce the cost of successive searches. For $1 < j \leq n$, denote by Δ_j the value $|\ell(j) - \ell(j-1)|$.

a) Design an algorithm for searching array $D[1:n]$ that exploits locality of reference. Specifically, show that query x_j in the sequence $x_1, x_2, \ldots, x_n$ can be answered at a cost proportional to $1 + lg(1 + \Delta_j)$, for all $j > 1$. [Hint: start by designing an algorithm–a kind of blend of linear and binary search– whose cost, when the algorithm returns index i, is proportional to $1 + lg\, i$.]

b) Let $\mathcal{A}$ be any comparison-based algorithm for searching $D[1:n]$. Prove that for every k, $1 \leq k < lg\, n$, there exists an index $i_k \in [\ell(j-1) - 2^k,\ \ell(j-1) + 2^k]$ such that $\mathcal{A}$ requires at least k comparisons to search for x_j, knowing $\ell(j-1)$, when it turns out that $\ell(j) = i_k$. [Hint: recall the fact that every binary tree has at most 2^d leaves at depth d, for every $d \geq 0$.]

4) In class we discussed x-fast tries, a kind of augmented direct access table that provides the additional functionality of efficient predecessor and successor operations: specifically, it represents an arbitrary set S of keys drawn from a universe $\mathcal{U} = \{0, 1, \ldots, u-1\}$ and permit predecessor (respectively, successor) queries to be answered, for arbitrary elements q of $\mathcal{U}$, in time $O(lg\, lg\, u)$.

Recall that an x-fast trie starts with a binary tree T whose leaves correspond to the elements of $\mathcal{U}$. For a given subset $S \subset \mathcal{U}$ we *mark* all of the nodes on the path from each element of S to the root of T. Associated with every marked internal node z of T is the index of the minimum and maximum value keys in S that lie in the subtree of T rooted at z, and associated with every marked leaf z of T is the index of the predecessor and successor of z in S. The addresses (together with associated minimum and maximum values) of all marked nodes, are stored in a dictionary D (implemented as a direct access table) and this information is used to search for the lowest marked ancestor of a given query x.

Let us denote by ancestor(x, h) the address of the height h ancestor of x in T, that is the node at distance h from x on the path from x to the root of T. For an arbitrary node z in T, let us denote by left-neighbour(z) the address of the closest node to the left of z on the same level of T; right-neighbour(z) is defined similarly. (*All of these functions are easy to compute using the addressing scheme described in class. However, the details are not needed here.*)

a) Show that x-fast tries are particularly efficient in handling predecessor and successor queries when the query itself is an element of S.

Recalling our success (in Question 3, above) in designing search structures that exploit *locality of reference*, we are motivated to try and modify an x-fast trie, so that the cost of predecessor and successor queries is a function of Δ, the distance between query q and the closer of its predecessor or successor in S.

b) Prove that if none of ancestor(x, h), left-neighbour(ancestor(x, h)), or right-neighbour(ancestor(x, h)) are marked then $\Delta > 2^h$.

c) Prove that if ancestor(x, h) is unmarked but at least one of left-neighbour(ancestor(x, h)) or right-neighbour(ancestor(x, h)) is marked then we can find the predecessor and successor of x, using information stored at these marked nodes and at the leaves, in $O(1)$ time.

d) Prove that if ancestor(x, h) is marked then we can find the predecessor and successor of x, by searching for the lowest marked node on the path from x to ancestor(x, h), in $O(lg\, h)$ time.

e) Using the results from parts (b), (c) and (d) above, describe how to find the predecessor and successor of x in an x-fast trie in $O(lg\, lg\, \Delta)$ time, when $\Delta \geq 4$.