Introduction to Algorithms November 1, 2002
Massachusetts Institute of Technology 6.046J/18.410J
Professors Erik Demaine and Shafi Goldwasser Handout 20

Problem Set 5 Solutions

(Exercises were not to be turned in, but we're providing the solutions for your own interest.)

Exercise 5-1. First, we find the rank r of x using OS-RANK. Then we query the $(r+i)$th order statistic, which is x's ith successor. Both of these operations require $O(\log n)$ time.

Exercise 5-2. Yes, the black-heights can be maintained. If x is a black node, then its black height is one more than its children's black heights; if x is red, then its black height is its children's black heights. By Theorem 14.1, this information can be maintained without changing the $O(\log n)$ asymptotic running times of the tree operations.

Exercise 5-3. We modify the comparison operator to break left-endpoint ties by right endpoint. That is, $i_1 < i_2$ if and only if *min*$[i_1] <$ *min*$[i_2]$ or (*min*$[i_1] =$ *min*$[i_2]$ and *max*$[i_1] <$ *max*$[i_2]$). This ordering is consistent with what is assumed by the interval tree operations, so their running times and correctness remain. INTERVAL-SEARCH-EXACTLY(T, i) works in the following way: if the left endpoint of the root is less than *min*$[i]$, we go right; if greater, we go left. If equal, do the following: if the right endpoint of the root is less than *max*$[i]$, we go right; if greater, we go left. If equal, return the root.

Correctness follows from the following observations: any interval identical to i must have the same left endpoint, so the first comparison eliminates intervals whose left endpoints cannot possibly match. Furthermore, because left-endpoint ties are broken by right-endpoints, the second comparison eliminates intervals whose left endpoints match i's, but whose right endpoints cannot possibly match i's.

The running time is $O(\log n)$ because the algorithm does $O(1)$ work at each node it visits, and descends one level each time.

Exercise 5-4. For each point p in the set, sort all the other points by polar angle relative to p (exercise 33.1-3). Then test each pair of adjacent elements in the sorted list for collinearity with p, using the cross product (the points are collinear if the cross product is 0). This algorithm is correct because if three points a, b, c are collinear and b is between a and c, then sorting around a will find this triplet. Conversely, the algorithm only reports that three points are collinear when it conclusively determines this fact. Sorting requires $O(n \log n)$ time, and it is performed $O(n)$ times, for a total running time of $O(n^2 \log n)$.

Exercise 5-5. First, note that we can detect in $O(1)$ time if two disks, of radii r_1 and r_2, intersect: they do if and only if the squared distance between their centers is at most $(r_1 + r_2)^2$. This test can be performed using only multiplications and additions.

Our algorithm is very similar to ANY-SEGMENTS-INTERSECT: for each disk, we have 2 event points: one for the leftmost point of the disk, and one for its rightmost point. When encountering a leftmost point in the scan, we the y-coordinate of the center of the disk to the tree (with a pointer to the disk itself as auxiliary data). The rest of the algorithm remains the same.

The $O(n \log n)$ running time is obvious. The proof of correctness exactly mirrors the one in the book.

Problem 5-1. Omniscient adversaries.

(a) The adversarial strategy is the following: find a longest uninterrupted chain. Call its final element x, and the element immediately following the chain z. Insert an element y such that $x < y < z$. There are two cases: if y is not duplicated in any higher level, then the new structure now has a longest uninterrupted chain that is one element longer than in the old one. (This occurs with probability 1/2.) If y is duplicated in a higher level, then the original chain is still uninterrupted, so the longest uninterrupted chain in the new structure is always no shorter than it was in the old one.

(b) We prove that after employing the above strategy, the length of the longest uninterrupted chain is $n/10$, with probability $1 - 2^{-cn}$ for some constant $c > 0$. This qualifies as high probability, because the error probability is exponentially decaying with n.

Consider the event that the length of the longest uninterrupted chain is less than $n/10$. This means that out of n coin flips, more than $9n/10$ of them came up "heads." Similarly to the analysis from class, we compute this probability to be at most:

$$\begin{aligned}
\binom{n}{9n/10}(1/2)^{9n/10} &= \binom{n}{n/10}(1/2)^{9n/10} \\
&\leq (10e)^{n/10}(1/2)^{9n/10} \\
&= 2^{(n \lg(10e))/10 - 9n/10} \\
&\leq 2^{3n/10 - 9n/10} \\
&= 2^{-6n/10}
\end{aligned}$$

(c) If there is an uninterrupted chain of length $\Omega(n)$, then for any element near the middle of the chain, the running time of SEARCH is $\Omega(n)$. Likewise, inserting any element near the middle of the chain requires $\Omega(n)$ time. A red-black tree, by contrast, *always* has $O(\log n)$-time SEARCH and INSERT operations, regardless of whether the adversary knows everything about the layout of the data structure. (In fact, because red-black trees are deterministic, an adversary can in fact know the exact layout of the tree at any point, simply by keeping an identical copy for itself.)

Problem 5-2. Survivor: MIT

(a) We put the elements $1, \ldots, n$ in a circular, doubly-linked list (i.e., each element has a pointer to both its successor and its predecessor, and the list "wraps around" from n to 1). Then we walk around the list, removing every mth element and appending it to a list containing the Survivor permutation. We repeat until the entire circular list is empty.

To create the doubly-linked list requires $O(n)$ time. To remove an element x from a doubly-linked list requires $O(1)$ time, because we need only modify the predecessor pointer of x's successor, and the successor pointer of x's predecessor. Appending to the end of the Survivor permutation list can also be done in $O(1)$ time. For each person kicked out, we traverse m pointers and do $O(1)$ pointer updates. Since n total people are kicked out, the running time is $O(mn)$. Correctness of the algorithm is apparent from the description of the problem.

(b) We maintain a dynamic order statistics red-black tree, described in the textbook and in lecture. First we insert every element $1, \ldots, n$, and first remove element m. After removing person x, the next person to be removed is (essentially) the mth successor of x. However, we must take into account "wrap-around," in case x doesn't have so many true successors. We can compute the "mth wrap-around successor" of x, immediately after removing x, as follows:

1. find the rank r of the predecessor of x,
2. find the number of elements k still in the tree,
3. find the element having rank $(r + m) \bmod k$ (or rank k, if $(r + m) \equiv 0 \bmod k$); this is the mth wrap-around successor.

We now analyze the running time of this procedure. Constructing the initial tree requires $O(n \log n)$ time. Deleting each element requires $O(\log n)$ time. Step 1 requires $O(\log n)$ time (see CLRS). Step 2 requires $O(1)$ time, simply by examining the augmented data at the root of the tree. Step 3 also requires $O(\log n)$ time. This loop is performed n times, for a total running time of $O(n \log n)$.

Problem 5-3. Accounting

Our data structure will be a red-black tree, ordered by the dates of the transactions, where each node x is augmented by a field *delta*, which is *the total change (increase) in the account balance over all nodes in x's subtree*. This information is merely the sum of all values of nodes in x's subtree (including x). It is clear that this information can maintained without affecting the running times of the red-black tree operations (by Theorem 14.1 from CLRS): $\mathit{delta}[x] = \mathit{sum}[x] + \mathit{delta}[\mathit{left}[x]] + \mathit{delta}[\mathit{right}[x]]$, and $\mathit{delta}[\mathit{nil}] = 0$.

From this description, INITIALIZE is trivial: simply construct an initially-*nil* red-black tree. Likewise, INSTRANS and DELTRANS are straightforward by Theorem 14.1: simply use the red-black tree INSERT and DELETE procedures, along with extra code to maintain *delta* according to the equations above.

BALANCE(*date*) is the only non-trivial operation. It is not enough to merely sum the *delta* values of all elements preceding *date*, because there may be up to n of them. Instead, we consider the path $(v_1, \ldots, v_k)$ from *date* to the root, where v_1 is the node for *date* and v_k is the root. Initially set an accumulator variable to *delta*$[$*left*$[v_1]]$. Then follow the path: whenever $v_{i-1} = \textit{right}[v_i]$ we add *sum*$[v_i]$ and *delta*$[$*left*$[v_i]]$ to the accumulator. The balance is the value of the accumulator at the end of the path.

The running time of this is $O(\log n)$, because the tree is balanced (so the path is short). To see correctness, we need to show that the value of every transaction that occurs before *date* is counted exactly once. Consider any date $y < \textit{date}$, and look at its path $(w_1, \ldots, w_m)$ to the root (where w_1 is the node for y, and w_m is the root). Then this path intersects the path from *date* to the root at some node, and is identical to it from that point on. The from the root downward, we have $w_m = v_k, w_{m-1} = v_{k-1}, \ldots, w_{m-(k-i)} = v_{k-(k-i)} = v_i$, and $v_{k-(k-i+1)} \neq v_{i-1}$ (where the paths diverge or one path ends).

There are now two cases: if $m - (k - i) = 1$, then the node for y (which is v_i) is on the path from *date* to the root. Because $y < \textit{date}$, it must be that v_{i-1} is the right child of v_i, so in fact the algorithm adds $\sum v_i$ to the accumulator. If $m - (k - i) > 1$, then the node for y must be in the left subtree of v_i (otherwise we would have $y > \textit{date}[v_i] \geq \textit{date}$ because the path to *date* diverges or ends at v_i). Then *date* is at v_i or is in the right subtree of v_i. If it is at v_i, then $v_i = v_1$ and the accumulator initially is set to the value of *delta*$[$*left*$[v_1]]$, which includes (exactly once) the value of the transaction on date y. If *date* is in the right subtree of v_i, then $v_{i-1} = \textit{right}[v_i]$, so *delta*$[$*left*$[v_i]]$ is added to the accumulator by the algorithm, and includes (exactly once) the value of y's transaction. This completes the proof of correctness.

Problem 5-4. Counting chord intersections

(a) Suppose $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. Because p_1 and p_2 are on a circle, to sort points by polar angle one simply sorts the points having non-negative y-coordinates by their x-coordinates in decreasing order, then the points having negative y-coordinates by their x-coordinates in increasing order. Therefore $p_1 < p_2$ if and only if: $y_1 > 0$ and $y_2 \leq 0$, or $y_1, y_2 \geq 0$ and $x_1 > x_2$, or $y_1, y_2 < 0$ and $x_1 < x_2$. This method of comparison is nice because it doesn't use any multiplications or additions.

Alternatively, we can use the cross-product: $p_1 < p_2$ if the path from the origin to p_1 to p_2 "turns left" (and does not cross from below the x-axis to above). That is, $p_1 < p_2$ if $p_2 \times p_1$ is negative, and NOT $(y_1 < 0$ and $y_2 \geq 0)$.

(b) We take the $2n$ endpoints of the chords, where each point also has a pointer to the other endpoint of its chord. We then sort the points in $O(n \log n)$ time using the comparison operator from above. We then make one pass over the sorted points, with j initially set to 1: at each point, if it is unlabelled, then label it A_j, label its corresponding endpoint B_j (by following the pointer), and increment j. If it is already labelled, do nothing.

It is clear from the description of the algorithm that in a sweep around the circle, A_j always precedes A_{j+1}. In addition, each chord has a unique j such that A_j and

B_j are its endpoints, because once a point is labelled its label never changes, and corresponding endpoints are always labelled with the same j. Finally, A_j precedes B_j because for a chord labelled with (A_j, B_j), the first endpoint encountered is labelled A_j.

The running time of this algorithm is $O(n \log n)$ for the sorting, and $O(n)$ for the sweep.

(c) We sort and label the endpoints as described above. Then we initialize c to zero and make a sweep over the sorted points, doing the following at each point: if the point is labelled A_j, add j to a dynamic order-statistics tree. If the point is labelled B_j, increment c by the size of the tree minus the rank of j in the tree, then remove j from the tree. After the loop, return c as the number of intersections.

For the proof of correctness, we take the following convention: we assign each intersection between two chords to the *first* chord encountered in a counterclockwise sweep. Now we argue correctness of the algorithm: consider a particular chord j having endpoints (A_j, B_j). The number of other chords which intersect this chord (according to our convention) is the number of A_ks which appear *after* A_j and *before* B_j, without a corresponding B_k in that range. By our labelling, all such A_ks will have $k > j$. At any point in the algorithm, the tree contains all k such that A_k has been encountered and B_k has not. Therefore the number of chords crossing chord j is the number of values k such that $k > j$ and k is in the tree; this is the size of the tree minus the rank of j in the tree. Because c is the sum of all these numbers, it is the total number of chord pairs which intersect.

For the running time, note that sorting takes $O(n \log n)$ time. In addition, insertion, deletion, and order statistic queries require $O(\log n)$ time apiece, and a constant number of such operations are performed for each of the $2n$ endpoints. Therefore the total running time is $O(n \log n)$.