Homework 3

Aleck Zhao

September 28, 2017

1 Dumbbell Matching Revisited

Recall the dumbbell matching problem from the last homework. You designed a randomized algorithm which took $O(n \log n)$ comparisons. Prove a matching lower bound for deterministic algorithms: prove that any deterministic algorithm which solves the problem must make at least $\Omega(n \log n)$ comparisons.

Proof. Given any ordering of A, there is exactly 1 ordering of B that matches. Thus, this problem is equivalent to finding the correct permutation. Under the comparison model, this can be modeled as a decision tree, where the leaves are the different permutations of B, of which there are $n!$. Then the minimum number of comparisons is the depth of this decision tree, which is $\log n! = \Omega(n \log n)$. □

2 Algorithms for Sorted Arrays

Suppose that you are given a sorted array A of length n and two values x and y with $x < y$. Consider the following problem: determine a) how many elements of A are less than x, b) how many elements of A are at least x and at most y, and c) how many elements of A are larger than y. Find a function $f(n)$ and prove the following about it.

(a) Any algorithm for this problem in the comparison model must have running time $\Omega(f(n))$.

Proof. Claim: $f(n) = \log n$. Consider performing a binary search insert in A for x in the form of a decision tree to find the index of x. Each leaf corresponds to an index for x, so there are $n + 1$ leaves, and thus the minimum possible depth is $\log(n + 1)$, which is the minimum number of comparisons needed. Similarly, the minimum possible depth to find the index for y is $\log(n + 2)$. Thus, the minimum number of comparisons total is $\log(n + 1) + \log(n + 2) = \Omega(\log n)$. □

(b) There is an algorithm for the problem which runs in time $O(f(n))$ (give such an algorithm and prove running time and correctness).

Proof. Using the algorithm described above, we can find the indices of x and y using binary search in the sorted array, which takes $O(\log n)$. Once this is done, the index of x is the number of elements less than x, then the index of y minus the index of x is the number of elements in between, and n minus the index of y is the number of elements greater than y. The running time and correctness are due to the properties of a binary search. □

3 Range Queries

We saw in class how to use binary search trees as dictionaries, and in particular how to use them to do insert and lookup operations. Some of you might naturally wonder why we bother to do this, when hash tables already allow us to do this. While there are many good reasons to use search trees rather than hash tables, one informal reason is that search trees can in some cases be either used directly or easily extended to allow efficient queries that are difficult or impossible to do efficiently in a hash table.

An important example of this is a range query. Suppose that all keys are distinct. In addition to being able to insert and lookup (and possibly delete), we want to allow a new operation range(x, y) which is suppose to return the number of keys in the tree which are at least x and at most y.

In this problem we will only be concerned with normal binary search trees. Recall that in binary search trees of height h, inserts can be done in $O(h)$ time.

(a) Given a binary search tree of height h, show how to implement range(x, y) in $O(h + k)$ time, where k is the number of elements that are at least x and at most y.

Proof. Consider the following algorithm:

(1) Traverse the tree to find x, or the smallest element greater than x if x is not in the tree.

(2) Initialize a counter at 1, and perform an in-order traversal until y is found, incrementing at each step. If y is not found, stop at the smallest element greater than y, and decrement the final result.

(3) Return the counter.

The initial traversal to find x takes $O(h)$, and the in-order traversal of an entire tree takes $O(n)$, so since we are only traversing k elements, it takes $O(k)$, so the total runtime is $O(h + k)$. This algorithm is correct because we are performing an in-order traversal. □

Can we do this operation even faster? It turns out that we can! In particular, for a binary search tree of height h, we can do this in $O(h)$ time.

(b) Describe an extra piece of information that you will store at each node of the tree, and describe how to use this extra information to do the above range query in $O(h)$ time. (Hint: think of keeping track of a size.)

Solution. At each node x, store the size of its left subtree as $\ell(x)$. Then consider the following algorithm:

(1) Traverse the tree to find x, or the smallest element greater than x if x is not in the tree. Initialize a counter $c(x) = 1$, and increment by $\ell(x)$.

(2) Move to the parent p of x until we reach the root.

  (a) If x was a left child, then continue moving up the tree, recursing.

  (b) If x was a right child, then increment $c(x)$ by $\ell(p) + 1$ and continue moving up the tree, recursing.

  (c) The final value of $c(x)$ is the number of nodes with values less than x.

(3) Repeat steps (1) and (2) with y to get $c(y)$.

(4) The number of nodes $\geq x$ and $< y$ is thus $c(y) - c(x)$, so return $c(y) - c(x) + 1$ to include y.

This algorithm takes $O(h)$ to find x then $O(h)$ to bubble up to the root and, and same for y, so the total time is $O(h)$. This algorithm is correct because if (a), then x is less than its parent, and if (b), x is greater than its parent and therefore all of its parent's left subtree. □

(c) Describe how to maintain this information in $O(h)$ time when a new node is inserted (note that there are no rotations on an insert - it's just the regular binary search tree insert, but you need to update information appropriately).

Solution. When a new node is inserted, it is compared to existing nodes. Consider the algorithm:

(1) Insert the node x using regular BST insertion. Set $\ell(x) = 0$.

(2) Move to the parent p of x until we reach the root.

   (a) If x was a left child, then increment $\ell(p)$, and continue moving up the tree, recursing.

   (b) If x was a right child, then continue moving up the tree, recursing.

The insertion takes $O(h)$ time, and bubbling back up to the root also takes $O(h)$ time, so the total is $O(h)$ to update. The algorithm is correct because if (a), then inserting x increased the size of the left subtree of p by 1, and if (b), then the left subtree of p did not change. □