Introduction to Algorithms — October 24, 2003
Massachusetts Institute of Technology — 6.046J/18.410J
Professors Shafi Goldwasser and Silvio Micali — Handout 18

Problem Set 4 Solutions

Problem 4-1. Reconstructing Binary Trees Via Traversals

Recall the binary tree data structure; recall three algorithms for traversing the tree: the *inorder* traversal, the *preorder* traversal, and the *postorder* traversal.

(a) Suppose you are given the preorder traversal and the inorder traversal of a binary tree. Can you reconstruct the tree? If so, give an algorithm for doing so and prove its correctness. If not, give a counter example. Note that a binary tree is not necessarily a binary search tree.

Solution: Yes, we can reconstruct the tree. We shall give a constructive proof that the tree can be built using the preorder and inorder sequences. Clearly the first element in the preorder is the root. We can then search for this element in the inorder traversal. The elements of the tree are now uniquely partitioned into left and right subtrees by this element. (ie., the subsequence of elements preceding the root in the inorder sequence is the inorder sequence for the left sub-tree and similarly the subsequence of elements succeeding the root is the inorder sequence for the right sub-tree) We can then recursively construct the left and right sub-trees from the preorder and inorder subsequences.

(b) Suppose you are given the preorder and postorder traversals of a binary tree. Can you reconstruct the tree? If so, give an algorithm for doing so and prove its correctness. If not, give a counter example.

Solution: No. If we have left only or right only subtrees in the tree then the preorder and postorder traversals are not necessarily unique. See the following example. Both the trees have preorder (1 2 3) and postorder (3 2 1).

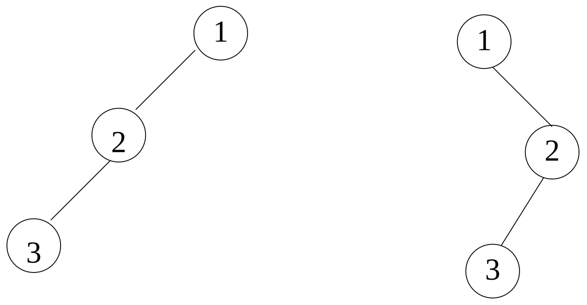

Figure 1: Two Trees with the same preorder (1 2 3) and postorder (3 2 1)

(c) Suppose you are given the preorder traversal of a binary tree. In addition, you are told that the tree is a binary search tree. Can you reconstruct the tree? If so, give an algorithm for doing so and prove its correctness. If not, give a counter example.

Solution: Yes. If it is known that the given tree is a binary search tree, then the inorder sequence of the tree is just the sorted order of the elements in the tree. Thus, from the preorder sequence we can obtain the inorder sequence by sorting the elements. We then can reconstruct the tree as in (a).

Problem 4-2. Successors In 2-3 Trees

A *2-3 tree* is a special B-tree in which each node which is not a leaf has 2 or 3 children, and every path from the root to a leaf is of the same length. Note that the tree consisting of a single node is a 2-3 tree.

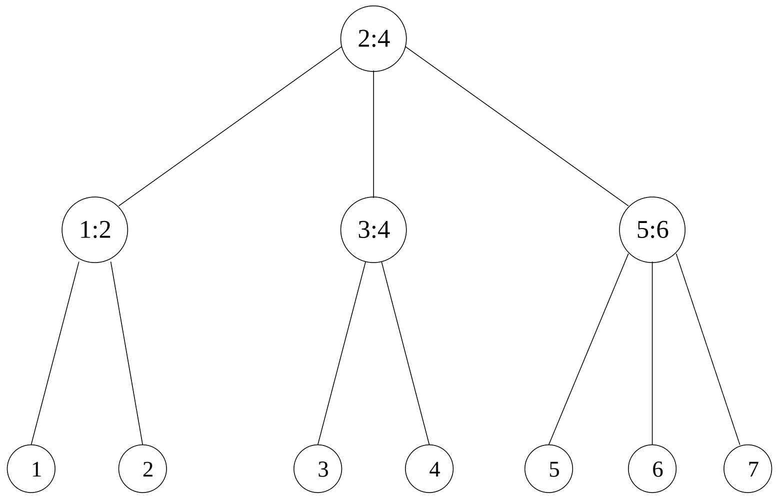

Figure 2: Seven keys stored in a 2-3 tree.

An ordered set can be stored in a 2-3 tree by storing the elements of the set in the leaves of the tree. We assign the elements to leaves in increasing order from left to right. If a is an internal node of the tree, then two pieces of information are stored in it: $L[a]$ is the largest element stored in a leaf of the subtree whose root is the leftmost child of a; $M[a]$ is the largest element stored in the subtree whose root is the middle child of a. The values of L and M assigned to each internal node enable us to start at the root and search for an element in a manner analogous to binary search. Figure 2 provides an illustration.

We wish to add the SUCCESSOR operation to 2-3 trees. Given a leaf x in a 2-3 tree, the SUCCESSOR operation returns the leaf that comes after x in the sorted order that the 2-3 tree determines.

(a) Describe how the SUCCESSOR operation can be implemented on a 2-3 tree in $O(\lg n)$ time, where n is the number of elements stored in the tree.

Solution: For this part we assume that each node has a pointer to its parent. In order to find the SUCCESSOR of a leaf x, we initially go up the tree to the first ancestor of x that has a child to the right of the one we came from. That is, we find the first ancestor of x, such that the subtree in which x is located has a right sibling. Then we find the minimal leaf in the sibling subtree, by following the left children all the way to a leaf. The leaf that we find this way is the successor of x. The worst case running time of this operation is $O(\lg n)$, because we go up the tree once and down the tree once.

(b) Describe how the 2-3 tree data structure can be modified to support the SUCCESSOR operation in $O(1)$ time. Make sure you can still support INSERT, DELETE, and SEARCH in $O(\lg n)$ time.

Solution: We modify the 2-3 tree data structure by doubly linking the leaves of the tree. The SUCCESSOR operation can now be implemented in $O(1)$ time by simply taking the next element on the list. This modification does not affect the asymptotic running time of the SEARCH, INSERT, and DELETE operations.

- The SEARCH operation is not affected at all.
- The INSERT operation should now also insert the new leaf into the doubly linked list. One can get a handle on the leaf immediately before or immediately after the new leaf via the new leaf's parent. Splits occur only for internal nodes and do not involve the leaves.
- The DELETE operation should now also remove the deleted leaf from the doubly linked list. Merging of nodes is done only for internal nodes and does not affect the leaves ordering.

The modification of the 2-3 tree data structure introduces only an $O(1)$ extra work per operation, and therefore the SEARCH, INSERT, and DELETE operations can still be implemented in $O(\lg n)$ time.

(c) Describe how the 2-3 tree data structure can be augmented in $O(n)$ time using $O(1)$ additional space at each node, so that for any $i, 1 \leq i < n$, the i^{th} successor of an element–provided it exists–can be found in $O(\lg n)$ time.

Solution: We use a postorder traversal of the tree: Each leaf is assigned a value 1 and each non-leaf node is assigned the sum of the values of its children. Thus, each node v will be assigned a value $f(v)$, where $f(v)$ is the number of leaves in the subtree rooted at v. It takes $O(n)$ time to augment the data structure.

To find the i^{th} successor of an element x in the tree, we need to determine the order statistic of x, say it is k, and then find the element with order statistic $k + i$.

To determine the order statistic of a leaf element x, we keep a sum s of the values of $f(v)$ for the nodes v that are *to the left* of nodes on which we have branched. For example, suppose the root r has three children, c_1, c_2, c_3. If we branch to the second child (i.e. determine that element x is in the subtree rooted at c_2), then we add $f(c_1)$ to the sum s. If we branch to the third child, then we add both $f(c_1)$ and $f(c_2)$ to the sum s. If we branch on the first child, we add nothing to s. When we find the element x, if the current sum is s, then x has order statistic $k = s + j + 1$, where j is the number of siblings to the left of x.

Then we search for the element with order statistic $k + i$. To do this, we keep a running sum s of the values $f(v)$ for the nodes v that are *to the left* of nodes on which we have branched. Then suppose we are at node v with three children c_1, c_2, c_3. If $s + f(c_1) \leq k + i$, we branch on c_1, since the subtree rooted at c_1 would contain the element with order statistic $k + i$. Otherwise, if $s + f(c_1) + f(c_2) \leq k + i$, then we branch on c_2, etc. We continue until we find the $k + i^{th}$ element.

Problem 4-3. For an integer n, a permutation π of size n is a one-to-one function from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n\}$. Given a sequence of n items $x_1, x_2, \ldots, x_n$, we can *apply* π to that sequence to obtain the permuted sequence $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}$. For example, if $\pi(1) = 3$, $\pi(2) = 1$, and $\pi(3) = 2$, then π applied to the sequence M,A,F yields F,M,A.

Let π be a permutation of size n. If $i < j$ and $\pi(i) > \pi(j)$ then the pair $(\pi(i), \pi(j))$ is called an *inversion* of the permutation. A single element out of place may cause many inversions. For example, the permutation

$$\pi(1) = 5, \quad \pi(2) = 1, \quad \pi(3) = 3, \quad \pi(4) = 4, \quad \pi(5) = 2$$

has 6 inversions: $(5, 1)$, $(5, 3)$, $(5, 4)$, $(5, 2)$, $(3, 2)$, and $(4, 2)$.

(a) What permutation of size n has the most inversions? How many does it have?

Solution: The most inversions is the reverse-sorted permutation, i.e. $n, n-1, \ldots 2, 1$, which has $\frac{n(n-1)}{2}$ inversions. We can see that this is maximum, because every pair can only have one inversion and here every pair has an inversion.

(b) Give an algorithm that determines the number of inversions in any permutation of size n in $O(n \lg n)$ time. Prove that it is correct and analyze its running time.

Solution: Insert the elements into a (initially empty) balanced (i.e. AVL, RB) order statistics tree. When you insert the i^{th} element, you can use the order statistics to determine how many elements already in the tree are greater than i. We add this number to our counter that records the total number of inversions.

Our algorithm runs in $O(\lg n)$ time since the tree is always balanced and insert operations take $O(\lg n)$ time in RB tree. To determine the order statistic of the most recently inserted element, we use OS-RANK which determines the rank of an element in time proportional to the height of the tree, namely $O(\lg n)$ time.

Problem 4-4. Skip Lists

A random *skip list* is an alternative data structure to a binary search tree. It has the desirable property that an INSERT or a FIND procedure can be conducted in $O(\lg n)$ time.

A skip list is an ordered linked list of elements: each element is attached to it successor with a pointer. Additionally, some elements have pointers to elements farther down the list, allowing us to *skip* elements when searching for a particular item in the list. An example of a skip list is shown in Figure 3.

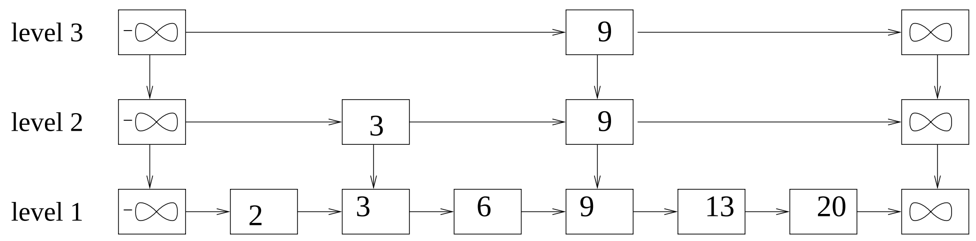

Figure 3: A skip list with height 3.

To create a skip list, we start with an "empty" skip list which contains two elements, ∞ and $-\infty$, with a pointer from ∞ to $-\infty$. Then for each element x that we want to insert into the skip list, we let the height of the element be h with probability $1/2^h$, i.e. we can toss a coin until it comes up heads on trial h. If we determine that the height of x should be k, then we insert x into the skip list and attach it with pointers to its successors at each of the first k levels. (If the height of element x is higher than that of any element already in the skip list, then we increase the height of the ∞ and $-\infty$ elements so that they have the same height as element x.)

(a) Show that if we insert n elements into an initally empty random skip list, the expected space required is $O(n)$. You can assume that an element x uses one unit of space at each level in which it appears and that each pointer uses one unit of space.

Solution: We can show that each element uses $O(1)$ space in expectation. By linearity of expectation, the total space requirement of a random skip list is $O(n)$.

An element uses at most 3 units of space at each level in which it appears. With probability $\frac{1}{2^j}$ an element i uses $3j$ units of space. Let X_{ij} be the indicator variable that indicates whether or not element i uses exactly j units of space. Thus, the expected number of units of space per element is

$$E[\sum_{j=1}^{\infty} X_{ij}] = \sum_{j=1}^{\infty} E[X_{ij}] = \sum_{j=1}^{\infty} 3\frac{j}{2^j} = 3\sum_{j=1}^{\infty} \frac{j}{2^j}.$$

As shown in the solutions to problem 3-1, this sum at most 3, so the expected number of space units per element is at most 9.

(b) Show that if we insert n elements into an initally empty random skip list, the expected number of levels, i.e. the maximum height of an element is at most $\lg n + O(1)$.

Solution: Let X_i be an indicator random variable that is 1 if the maximum level is at least i and 0 otherwise. We want to compute $E[\sum_{i=1}^{\infty} X_i] = \sum_{i=1}^{\infty} E[X_i]$.

Note that X_i is 1 with probability at most $\min\{1, \frac{n}{2^i}\}$. This is because each element has height at least i with probability $1/2^i$. If we union bound over all n elements, we get that X_i is 1 with probability at most $n/2^i$.

Thus,

$$\sum_{i=1}^{\infty} E[X_i] \leq \sum_{i=1}^{i=\lg n} 1 + \sum_{i=\lg n}^{\infty} \frac{n}{2^i} = \lg n + \sum_{i=\lg n}^{\infty} \frac{n}{2^i}.$$

Note that the second part of the summation, $\sum_{i=\lg n}^{\infty} \frac{n}{2^{\lg n}} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \cdots \leq 2$. So $\sum_{i=1}^{\infty} E[X_i] = \lg n + O(1)$.

(c) Let x_i denote the element in a skip list that has value x at level i. A *parent* of an element x_i in a skip list is the maximum valued element of value at at most x at level $i + 1$. For example, in Figure 3, 9_2 is the parent of 9_1, 13_1, and 20_1.

Prove that the expected number of children, i.e. elements on level i with the same parent on level $i + 1$, of an element y_{i+1} is $O(1)$.

Solution: Note that if an element x_{i+1} has c children, then each child has height exactly i. Given that an element has height at least i, the probability that it has height at least $i + 1$ is $1/2$. Let X_j be the indicator random variable that is a 1 if j_i is a child of x_{i+1} and 0 otherwise. Note that x_{i+1} has exactly one child with probability $1/4$, exactly two children with probability $2/8$, exactly three children with probability $3/16$, etc. So the expected number of children of element y_{i+1} is:

$$\sum_{j=1}^{\infty} E[X_j] = \sum_{j=1}^{\infty} \frac{j}{2^{j+1}} \leq 2,$$

which is $O(1)$.

(d) Suppose we follow the convention that we never insert any element into the skip list with height more than $100 \lg n$. In other words, to assign a height to an element x, we flip a coin until it comes up heads on trial h. Then we let the height be $\min\{h, 100 \lg n\}$. Use this to determine the expected time for an INSERT procedure.

Solution: This follows from the fact that in a search path, we branch on an element x_i and then probe only the children of x_i (at level $i - 1$). Since from part **(c)**, we know that, in expectation, each element has a constant number of children on each level, it follows by linearity of expectation that the expected time of an INSERT procedure is $O(\lg n)$. Note that in the top level, all of the elements can be viewed as children of the same parent.