INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Design and Analysis of Algorithms (MA 21007)

Assignment-4/Due: 28 Sep. 2015

Problem 1 Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

Problem 2 Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

Problem 3 Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (*Hint:* List the possibilities when $n = 3$.)

Problem 4 Let us define a **relaxed red-black tree** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree?

Problem 5 An **AVL tree** is a binary search tree that is **height balanced**: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra field in each node: $h[x]$ is the height of node x. As for any other binary search tree T, we assume that $root[T]$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (*Hint:* Prove that in an AVL tree of height h, there are at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, a node is first placed in the appropriate place in binary search tree order. After this insertion, the tree may no longer be height balanced. Specifically, the heights of the left and right children of some node may differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|h[right[x]] - h[left[x]]| \leq 2$, and alters the subtree rooted at x to be height balanced. (*Hint:* Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(x, z), which takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREE-INSERT from Section 12.3, assume that $key[z]$ has already been filled in and that $left[z] = \text{NIL}$ and $right[z] = \text{NIL}$; also assume that $h[z] = 0$. Thus, to insert the node z into the AVL tree T, we call AVL-INSERT($root[T], z$).

d. Give an example of an n-node AVL tree in which an AVL-INSERT operation causes $\Omega(\lg n)$ rotations to be performed.

Problem 6-1. Treaps

If we insert a set of n items into a binary search tree using TREE-INSERT, the resulting tree may be horribly unbalanced. As we saw in class, however, we expect randomly built binary search trees to be balanced. (Precisely, a randomly built binary search tree has expected height $O(\lg n)$.) Therefore, if we want to build an expected balanced tree for a fixed set of items, we could randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?

We will examine a data structure that answers this question in the affirmative. A **treap** is a binary search tree with a modified way of ordering the nodes. Figure 1 shows an example of a treap. As usual, each item x in the tree has a key *key*$[x]$. In addition, we assign *priority*$[x]$, which is a random number chosen independently for each x. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that (1) the keys obey the binary-search-tree property and (2) the priorities obey the min-heap order property. In other words,

- if v is a left child of u, then *key*$[v] <$ *key*$[u]$;
- if v is a right child of u, then *key*$[v] >$ *key*$[u]$; and
- if v is a child of u, then *priority*$(v) >$ *priority*(u).

(This combination of properties is why the tree is called a "treap": it has features of both a binary search tree and a heap.)

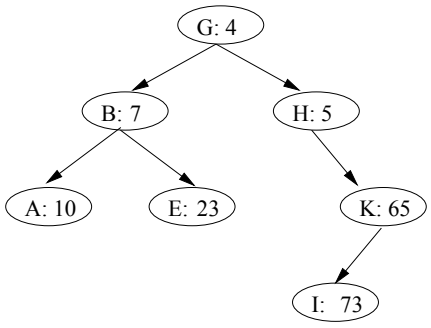

Figure 1: A treap. Each node x is labeled with *key*$[x]$: *priority*$[x]$. For example, the root has key G and priority 4.

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, each with an associated key, into a treap in arbitrary order. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities. In other words, *priority*$[x_i] <$ *priority*$[x_j]$ means that x_i is effectively inserted before x_j.

(a) Given a set of nodes $x_1, x_2, \ldots, x_n$ with keys and priorities all distinct, show that there is a unique treap with these nodes.

(b) Show that the expected height of a treap is $O(\lg n)$, and hence the expected time to search for a value in the treap is $O(\lg n)$.

Let us see how to insert a new node x into an existing treap. The first thing we do is assign x a random priority *priority*$[x]$. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 2.

(c) Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (*Hint:* Execute the usual binary search tree insert and then perform rotations to restore the min-heap order property.)

(d) Show that the expected running time of TREAP-INSERT is $O(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although searching and rotating have the same asymptotic running time, they have different costs in practice. A search reads information from the treap without modifying it, while a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant (in fact, less than 2)!

In order to show this property, we need some definitions, illustrated in Figure 3. The **left spine** of a binary search tree T is the path which runs from the root to the item with the smallest key. In other words, the left spine is the maximal path from the root that consists only of left edges. Symmetrically, the **right spine** of T is the maximal path from the root consisting only of right edges. The **length** of a spine is the number of nodes it contains.

(e) Consider the treap T immediately after x is inserted using TREAP-INSERT. Let C be the length of the right spine of the left subtree of x. Let D be the length of the left spine of the right subtree of x. Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

We will now calculate the expected values of C and D. For simplicity, we assume that the keys are $1, 2, \ldots, n$. This assumption is without loss of generality because we only compare keys.

For two distinct nodes x and y, let $k =$ *key*$[x]$ and $i =$ *key*$[y]$, and define the indicator random variable

$$X_{i,k} = \begin{cases} 1 & \text{if } y \text{ is a node on the right spine of the left subtree of } x \text{ (in } T\text{)}, \\ 0 & \text{otherwise.} \end{cases}$$

(f) Show that $X_{i,k} = 1$ if and only if (1) *priority*$[y] >$ *priority*$[x]$, (2) *key*$[y] <$ *key*$[x]$, and (3) for every z such that *key*$[y] <$ *key*$[z] <$ *key*$[x]$, we have *priority*$[y] <$ *priority*$[z]$.

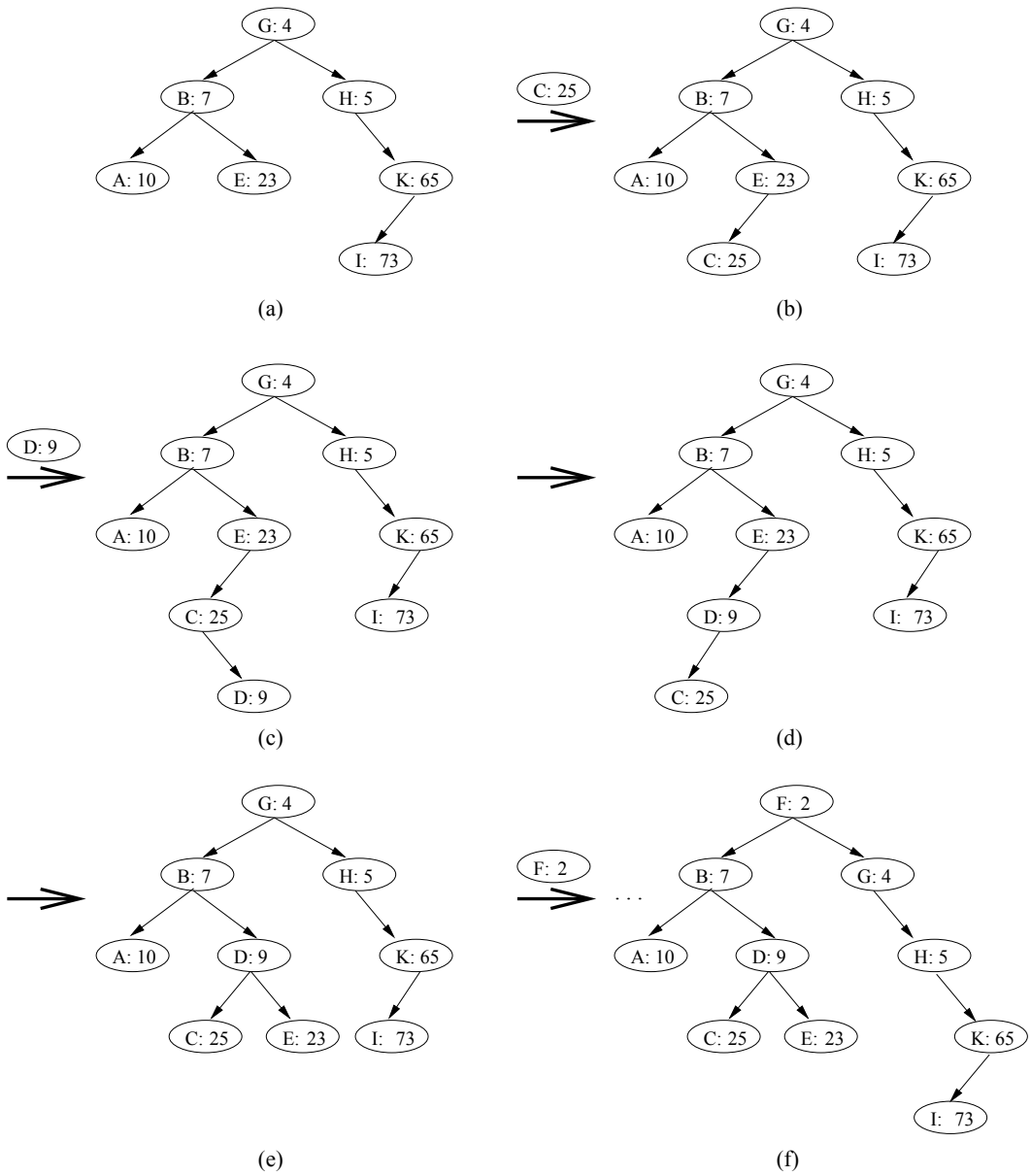

Figure 2: Operation of TREAP-INSERT. As in Figure 1, each node x is labeled with *key*$[x]$: *priority*$[x]$. **(a)** Original treap prior to insertion. **(b)** The treap after inserting a node with key C and priority 25. **(c)–(d)** Intermediate stages when inserting a node with key D and priority 9. **(e)** The treap after insertion of parts (c) and (d) is done. **(f)** The treap after inserting a node with key F and priority 2.

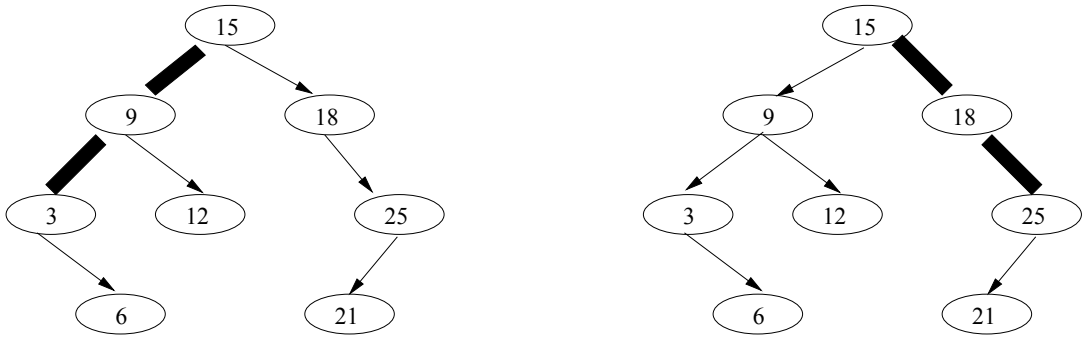

Figure 3: Spines of a binary search tree. The left spine is shaded in **(a)**, and the right spine is shaded in **(b)**.

(g) Show that

$$\Pr\{X_{i,k} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

(h) Show that

$$\mathrm{E}[C] = \sum_{j=1}^{k-1} \frac{1}{j(j+1)} = 1 - \frac{1}{k}.$$

(i) Use a symmetry argument to show that

$$\mathrm{E}[D] = 1 - \frac{1}{n-k+1}.$$

(j) Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

Problem 6-2. Being balanced

Call a family of trees **balanced** if every tree in the family has height $O(\lg n)$, where n is the number of nodes in the tree. (Recall that the **height** of a tree is the maximum number of edges along any path from the root of the tree to a leaf of the tree. In particular, the height of a tree with just one node is 0.)

For each property below, determine whether the family of binary trees satisfying that property is balanced. If you answer is "no", provide a counterexample. If your answer is "yes", give a proof (hint: it should be a proof by induction). Remember that being balanced is an *asymptotic* property, so your counterexamples must specify an infinite set of trees in the family, not just one tree.

(a) Every node of the tree is either a leaf or it has two children.

(b) The size of each subtree can be written as $2^k - 1$, where k is an integer (k is *not* the same for each subtree).

(c) There is a constant $c > 0$ such that, for each node of the tree, the size of the smaller child subtree of this node is at least c times the size of the larger child subtree.

(d) There is a constant c such that, for each node of the tree, the heights of its children subtrees differ by at most c.

(e) The average depth of a node is $O(\lg n)$. (Recall that the **depth** of a node x is the number of edges along the path from the root of the tree to x.)

Problem 7

Join operation on red-black trees

The **join** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $key[x_1] \leq key[x] \leq key[x_2]$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Given a red-black tree T, we store its black-height as the field $bh[T]$. Argue that this field can be maintained by RB-INSERT and RB-DELETE without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

We wish to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $bh[T_1] \geq bh[T_2]$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $bh[T_2]$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how properties 2 and 4 can be enforced in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $bh[T_1] \leq bh[T_2]$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.