Problem Set 5

MIT students: This problem set is due in lecture on *Day 20*.

Reading: Chapters 12 and 13

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered by the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation instructor and time, the date, and the names of any students with whom you collaborated.

MIT students: Each problem should be done on a separate sheet (or sheets) of three-hole punched paper.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

Exercise 5-1. Do exercise 12.2-9 on page 260 of CLRS.

Exercise 5-2. Do exercise 12.4-3 on page 268 of CLRS.

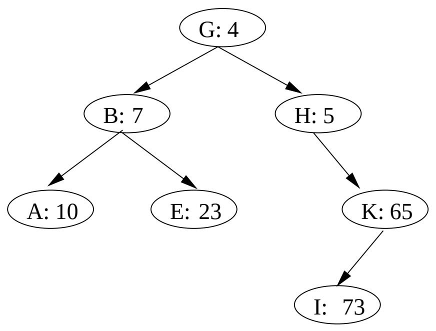

Figure 1: A treap. Each node x is labeled with *key*$[x]$: *priority*$[x]$. For example, the root has key G and priority 4.

Exercise 5-3. Do exercise 13.2-4 on page 279 of CLRS.

Exercise 5-4. Do exercise 13.4-7 on page 295 of CLRS.

Problem 5-1. Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. On the other hand, we know that randomly built binary search trees tend to be balanced. Therefore, a strategy that, on average, builds a balanced tree for a fixed set of items is to randomly permute the items and then insert them in that order into the tree.

But, 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. As usual, each node x in the tree has a key value *key*$[x]$. In addition, we assign *priority*$[x]$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- 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*$[u]$.
- 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.)

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, with associated keys, into a treap. 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, i.e., *priority*$[x_i] <$ *priority*$[x_j]$ means that x_i was inserted before x_j.

(a) Explain why a treap on n nodes is equivalent to a randomly built binary search tree on n nodes.

Solution:

Assigning priorities to nodes as they are inserted into a treap is the same as inserting the n nodes into a normal binary search tree in the (increasing) order defined by their priorities.
So if we assign the priorities randomly, we will get a random order of n priorities, which is the same as a random permutation of the n inputs, so we can view this as inserting the n items in random order i.e. a randomly built binary search tree.

(b) Conclude that the expected time to search for a value in the treap is $\Theta(\lg n)$.

Solution:

The time to search for an item that is in the treap is equal to the *depth* of that item. Now, we know that for an item x in a randomly built binary search tree, the expected depth of x is $\Theta(\lg n)$ (the expectation is taken over permutations of the n nodes, not the choice of x). Thus, the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. 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 insertion procedure and then perform rotations to restore the min-heap order property.)

Solution:

The hint gives the idea: do the usual binary search tree insert and then perform rotations to restore the min-heap order property.

TREAP-INSERT(T, x) inserts x into the treap T (by modifying T). It requires that x has defined key and $priority$ values. We have used the subroutines TREE-INSERT, RIGHT-ROTATE, and RIGHT-ROTATE as defined in CLRS.

```
TREAP-INSERT$(T, x)$
1  TREE-INSERT$(T, x)$
2  while $x \neq root[T]$ and $priority[x] < priority[p[x]]$
3      do if $x = left[p[x]]$
4            then RIGHT-ROTATE$(T, p[x])$
5            else LEFT-ROTATE$(T, p[x])$
```

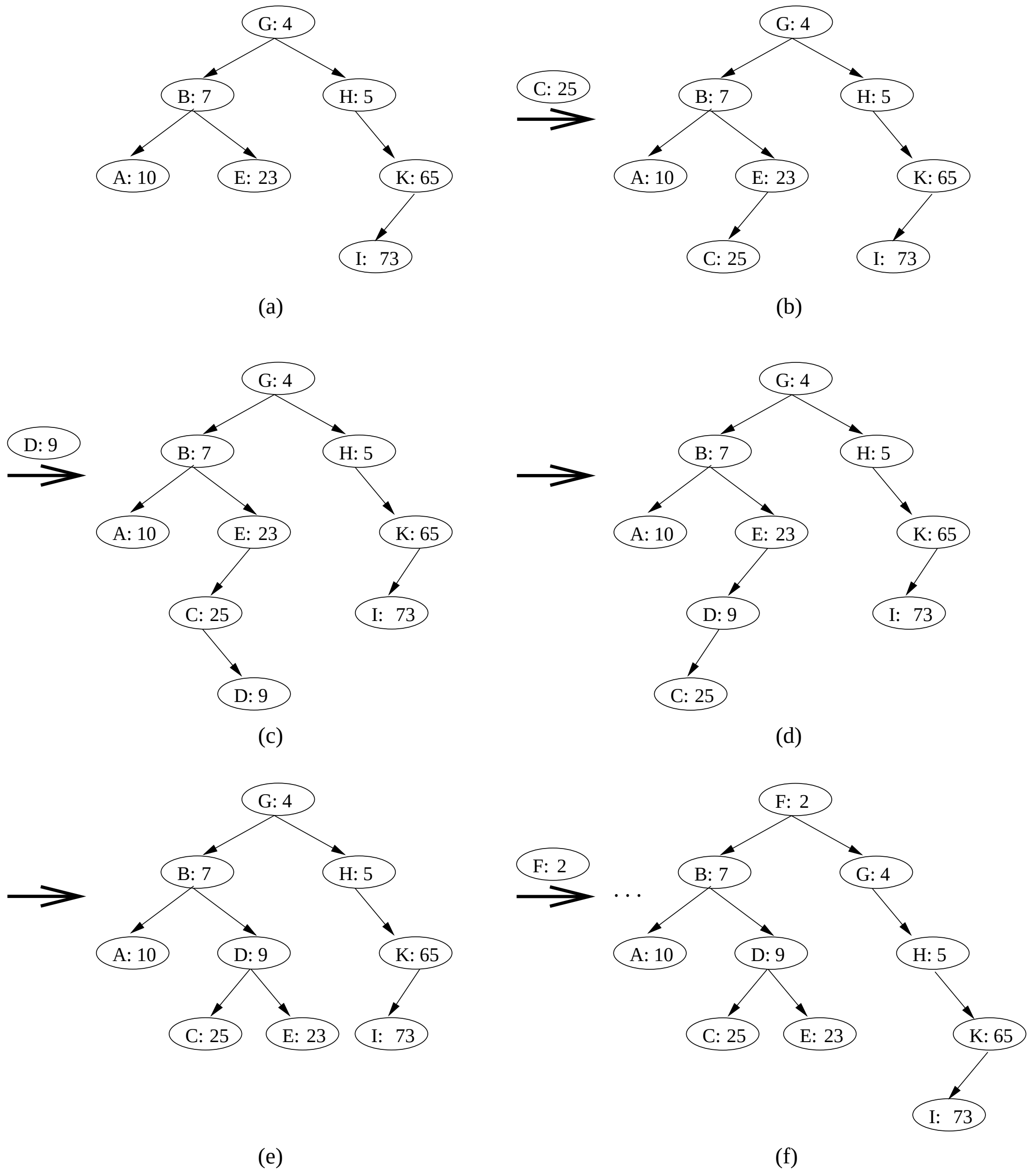

Figure 2: The operation of TREAP-INSERT. **(a)** The original treap, prior to insertion. **(b)** The treap after inserting a node with key C and priority 25. **(c)–(d)** Intermediate stages when inserting a node with key D and priority 9. **(e)** The treap after the insertion of parts (c) and (d) is done. **(f)** The treap after inserting a node with key F and priority 2.

Note that parent pointers simplify the code but are not necessary. Since we only need to know the parent of each node on the path from the root to x (after the call to TREE-INSERT), we can keep track of these ourselves.

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

Solution:

TREAP-INSERT first inserts an item in the tree using the normal binary search tree insert and then performs a number of rotations to restore the min-heap property.

The normal binary search tree insert always places the new item at a new leaf of tree. Therefore the expected time to insert an item into a treap is the expected height of a randomly built binary search tree, which is $O(\lg n)$. Since the height of every binary tree is $\Omega(\lg n)$ (a complete tree has the smallest height and its height is $\Omega(\lg n)$), the expected height is $\Theta(\lg n)$.

The maximum number of rotations occurs when the new item receives a priority less than all priorities in the tree. In this case it needs to be rotated from a leaf to the root. An upper bound on the expected number of rotations is therefore the expected height of a randomly built binary search tree, which is $\Theta(\lg n)$. Since each rotation take constant time, the expected time to rotate is $\Theta(\lg n)$.

Thus the expected running time of TREAP-INSERT is $\Theta(\lg n + \lg n) = \Theta(\lg n)$.

Problem 5-2. 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*$[x_1] \le$ *key*$[x] \le$ *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 (as given in the textbook, on pages 280 and 288 respectively) 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.

Solution:

Starting at the root, we can proceed to a leaf, couting the number of black nodes on the path. This does not require any extra storage in the nodes of the tree and will take $\Theta(\lg n)$ time. Since RB-INSERT and RB-DELETE also run in $\Theta(\lg n)$ time, the asymptotic running time is not increased.

While descending through T, we decrement *bh*$[T]$ by 1 everytime we encounter a black node. The black-height of a node, N, is then *bh*$[T]$ minus the number of black nodes encountered (excluding node N itself). This decrement can be done in $O(1)$ time per node visited.

We wish to implement the operation RB-JOIN(T_1, x, T_2), which may destroy 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*$[T_1] \ge$ *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*$[T_2]$.

Solution:

Since T_1 is a binary search tree, the largest element at any level is on the rightmost path. So, we decend down the rightmost path, calculating *bh* at each node (as described in the previous part), until we reach the black node whose black-height is *bh*$[T_2]$, which is what we want. Thus the running time is at most the height of the tree, i.e. $O(\lg n)$. (Calculating the black-height takes $O(1)$ per node, as shown in the previous part).

(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.

Solution:

Insert x into where y was in T_1. Form $T_y \cup \{x\} \cup T_2$ by letting T_y be the left subtree of x, and T_2 be the right subtree of x. Given that this join operation is such that $\mathit{key}[x_1] \leq \mathit{key}[x] \leq \mathit{key}[x_2]$ where $x_1 \in T_1$ and $x_2 \in T_2$, the binary search tree property is maintained and this operation takes $O(1)$ time.

Consider the following red-black properties:

- every node is either red or black
- every leaf is black
- for each node, all paths from the node to descendant leaves contain the same number of black nodes

(d) What color should we make x so that the above red-black properties are maintained?

Solution:

We should make x red. Since T_y already has black-height = $\mathit{bh}(T_y)$, x must be red to maintain the same black-height, $\mathit{bh}[T_y \cup \{x\} \cup T_2]$ = $\mathit{bh}(T_y)$

Consider the following red-black properties:

- the root is black
- if a node is red, then both its children are black

(e) Describe how the above two properties can be enforced in $O(\lg n)$ time.

Solution:

Use RB-INSERT-FIXUP on the new tree, to perform the recoloring and rotations necessary to enforce these two properties. We know that RB-INSERT-FIXUP runs in $O(\lg n)$ time, thus we conclude that the enforcement can be done in $O(\lg n)$ time.

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

Solution:

RB-JOIN is implemented by using all the previous parts: The black-height can be calculated and maintained in $O(1)$ time. The required black node, y, can be found in $O(\lg n)$ time. Then, the join is done in $O(\lg n)$ time, and finally, after assigning x the right color, the red-black tree properties can be enforced in $O(\lg n)$ time. So the total runing time is $O(\lg n)$