Problem Set 7

MIT students: This problem set is due in lecture on *Monday, November 5*.

SMA students: This problem set is due after the video-conferencing session on *Wednesday, November 7*.

Reading: Chapters 17

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.

SMA students: Each problem should be done on a separate sheet (or sheets) of two-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 7-1. Do exercise 17.1-1 on page 409 of CLRS.

Exercise 7-2. Do exercise 17.3-4 on page 416 of CLRS.

Exercise 7-3. Do exercise 17.3-7 on page 416 of CLRS.

Exercise 7-4. Do exercise 17.4-1 on page 424 of CLRS.

Problem 7-1. Reducing the space in the van Emde Boas structure

In this problem, we will use hashing to modify the van Emde Boas data structure presented in lecture in order to reduce its space usage.

Recall the problem statement: In the ***fixed-universe successor*** problem, a data structure must maintain a dynamic subset S of the universe $U = \{0, \ldots, u-1\}$. The data structure must support the operations of inserting elements into S, deleting elements from S, finding the successor (next element in S) from any element in U, and finding the predecessor (previous element in S) from any element in U.

Recall the outline of the van Emde Boas data structure: The universe $U = \{0, \ldots, u-1\}$ is represented by a ***widget*** of size u. Each widget W of size $|W|$ stores an array *sub*$[W]$ of $\sqrt{|W|}$ ***recursive subwidgets*** *sub*$[W][0]$, *sub*$[W][1], \ldots,$ *sub*$[W][\sqrt{|W|}-1]$ each of size $\sqrt{|W|}$. In addition, each widget W stores a ***summary widget*** *summary*$[W]$ of size $\sqrt{|W|}$, representing which subwidgets are nonempty. Each widget W also stores its minimum element *min*$[W]$ separately from all the subwidgets. Finally, each widget W maintains the value *max*$[W]$ of its maximum element.

For reference, the van Emde Boas algorithms for insertion and finding successors in $O(\lg \lg u)$ time are given as follows. For any widget W, and for any x in the universe of possible elements in W, define *high*(x) and *low*(x) to be nonnegative integers so that $x =$ *high*$(x)\sqrt{|W|} +$ *low*(x). Thus, *high*(x) and *low*(x) are both less than $\sqrt{|W|}$, and represent the high-order and low-order halves of the bits in the binary representation of x.

```
vEB-INSERT(x, W)
if x < min[W]
    then exchange x ↔ min[W]
if subwidget sub[W][high(x)] is nonempty, that is, min[sub[W][high(x)]] ≠ NIL
    then vEB-INSERT(low(x), sub[w][high(x)])
    else min[sub[W][high(x)]] ← low(x)
         vEB-INSERT(high(x), summary[W])
if x > max[W]
    then max[W] ← x
```

vEB-SUCCESSOR(x, W)
if $x <$ *min*$[W]$
  then return *min*$[W]$
if *low*$(x) <$ *max*$[$*sub*$[W][$*high*$(x)]]$
  then $j \leftarrow$ vEB-SUCCESSOR(*low*(x), *sub*$[W][$*high*$(x)]$)
    return *high*$(x)\sqrt{|W|} + j$
  else $i \leftarrow$ vEB-SUCCESSOR(*high*(x), *summary*$[W]$)
    return $i\sqrt{|W|}$ + *min*$[$*sub*$[W][i]]$

(a) Argue that the van Emde Boas data structure uses $\Theta(u)$ space. (*Hint:* Derive a recurrence for the space $S(u)$ occupied by a widget of size u.)

Consider the following modifications to the van Emde Boas data structure.

1. Empty widgets are represented by the value NIL instead of being explicitly represented by a recursive construction.
2. The structure *sub*$[W]$ containing the subwidgets

$$sub[W][0], sub[W][1], \ldots, sub[W][\sqrt{|W|} - 1]$$

   is stored as a dynamic hash table (as in Section 17.4 of CLRS) instead of an array. The key of a subwidget *sub*$[W][i]$ is i, so we can quickly find the ith subwidget *sub*$[W][i]$ by a single search in the hash table *sub*$[W]$.
3. As a consequence of the first two modifications, the hash table *sub*$[W]$ only stores the *nonempty* subwidgets. The NIL values of the empty subwidgets are not even stored in the hash table. Thus, the space occupied by the hash table *sub*$[W]$ is proportional to the number of nonempty subwidgets of W.

Whenever we insert an element into an empty (NIL) widget, we ***create*** a widget using the following procedure, which runs in $O(1)$ time:

CREATE-WIDGET(x) ▷ Returns a new widget containing just the element x.
allocate a widget structure W
min$[W] \leftarrow x$
max$[W] \leftarrow x$
summary$[W] \leftarrow$ NIL
sub$[W] \leftarrow$ a new empty dynamic hash table
return W

In the next two problem parts, you will develop the insertion and successor operations for this modified van Emde Boas structure. It suffices to simply describe the necessary changes from the vEB-INSERT and vEB-SUCCESSOR operations detailed above. In any case, you should give special attention to the interaction with the hash table *sub*$[W]$.

(b) Give an efficient algorithm for inserting an element into the modified van Emde Boas structure, using CREATE-WIDGET as a subroutine.

(c) Give an efficient algorithm for finding the successor of an element in the modified van Emde Boas structure.

(d) Using known results, argue that the running time of your modified insertion and successor algorithms run in $O(\lg \lg u)$ expected time, under the assumption of simple uniform hashing.

(e) Prove that the space occupied by the modified data structure is $O(n)$. You may ignore the possibility of deletions, and assume that only insertions and successor operations are performed.

Problem 7-2. The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color modifications. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color modifications.

(a) Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n + 1)$st node causes $\Omega(\lg n)$ color modifications. Then describe a legal red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color modifications.

Although the worst-case number of color modifications per operation can be logarithmic, we shall prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case.

(b) Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (hint: look at Figures 13.5, 13.6, and 13.7)

We shall first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T. Assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

(c) Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T. Argue that $\Phi(T') = \Phi(T) - 1$.

(d) Node insertion into a red-black tree using RB-INSERT can be broken down into three parts. List the structural modifications and potential changes resulting from lines 1-16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

(e) Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node x,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red ,} \\ 1 & \text{if } x \text{ is black and has no red children ,} \\ 0 & \text{if } x \text{ is black and has one red child ,} \\ 2 & \text{if } x \text{ is black and has two red children .} \end{cases}$$

Now we redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x) ,$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T.

(f) Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

(g) Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

(h) Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.