CS513: Design and Analysis of Data Structures and Algorithms
Homework 2

Due November 07, 2012

Next to each question, there is an indication of how hard we think each question is. Your answers should be as concise as possible while also fully explaining your solution. Please make an effort to write legibly.

The following questions are from the 2\textsuperscript{nd} edition of the CLRS book. The text of the questions is placed here to avoid issues with different versions of the book.

12.1-5 (easy) Argue that since sorting \( n \) elements takes \( \Omega(n \log n) \) 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 \( \Omega(n \log n) \) time in the worst case.

12.3-3 (medium) We can sort a given set of \( n \) numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

12.3-5 (easy) Is the operation of deletion “commutative” in the sense that deleting \( x \) and then \( y \) from a binary search tree leaves the same tree as deleting \( y \) and then \( x \)? Argue why it is or give a counterexample.

13.2-5 (hard) We say that a binary search tree \( T_1 \) can be right-converted to binary search tree \( T_2 \) if it is possible to obtain \( T_2 \) from \( T_1 \) via a series of calls to RIGHT-ROTATE. Give an example of two trees \( T_1 \) and \( T_2 \) such that \( T_1 \) cannot be right-converted to \( T_2 \). Then show that if a tree \( T_1 \) can be right-converted to \( T_2 \), it can be right-converted using \( O(n^2) \) calls to RIGHT-ROTATE.

13.4-4 (easy) In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel \text{nil}[T]\

(Algorithm is given below.)

13.4-5 (easy) In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees \( \alpha, \beta, \ldots, \zeta \), and verify that each count remains the same after the transformation. When a node has a color attribute \( c \) or \( c' \), use the notation \( \text{count}(c) \) or \( \text{count}(c') \) symbolically in your count.

(Figure is given below.)
13.4 Deletion

- since $y$ could not have been the root if it was red, the root remains black.

The node $x$ passed to $\text{RB-DELETE-DELETE-FIXUP}(T, x)$ is one of two nodes: either the node that was $y$'s sibling before $y$ was spliced out if $y$ had a child that was not the sentinel $\text{nil}$, or, if $y$ had no children, $x$ is the sentinel $\text{nil}$. In the latter case, the unconditional assignment in line 7 guarantees that $x$'s spare parent, whether $x$ is a key-bearing internal node or the sentinel $\text{nil}$.

We can now examine how the procedure $\text{RB-DELETE-DELETE-FIXUP}(T, x)$ restores the red-black properties to the search tree.

$$\text{RB-DELETE-DELETE-FIXUP}(T, x)$$

1. while $x \neq \text{root}[T]$ and $\text{color}[x] = \text{BLACK}$
2. do if $x = \text{left}[p[x]]$
3. then $w \leftarrow \text{right}[p[x]]$
4. if $\text{color}[w] = \text{RED}$
5. then $\text{color}[w] \leftarrow \text{BLACK}$ ▷ Case 1
6. $\text{color}[p[x]] \leftarrow \text{RED}$ ▷ Case 1
7. $\text{LEFT-ROTATE}(T, p[x])$ ▷ Case 1
8. $w \leftarrow \text{right}[p[x]]$ ▷ Case 1
9. if $\text{color}[\text{left}[w]] = \text{BLACK}$ and $\text{color}[\text{right}[w]] = \text{BLACK}$
10. then $\text{color}[w] \leftarrow \text{RED}$ ▷ Case 2
11. $x \leftarrow p[x]$ ▷ Case 2
12. else if $\text{color}[\text{right}[w]] = \text{BLACK}$
13. then $\text{color}[\text{left}[w]] \leftarrow \text{BLACK}$ ▷ Case 3
14. $\text{color}[w] \leftarrow \text{RED}$ ▷ Case 3
15. $\text{RIGHT-ROTATE}(T, w)$ ▷ Case 3
16. $w \leftarrow \text{right}[p[x]]$ ▷ Case 3
17. $\text{color}[w] \leftarrow \text{color}[p[x]]$ ▷ Case 4
18. $\text{color}[p[x]] \leftarrow \text{BLACK}$ ▷ Case 4
19. $\text{color}[\text{right}[w]] \leftarrow \text{BLACK}$ ▷ Case 4
20. $\text{LEFT-ROTATE}(T, p[x])$ ▷ Case 4
21. $x \leftarrow \text{root}[T]$ ▷ Case 4
22. else (same as then clause with “right” and “left” exchanged)
23. $\text{color}[x] \leftarrow \text{BLACK}$

If the spliced-out node $y$ in $\text{RB-DELETE-DELETE-FIXUP}(T, x)$ is black, three problems may arise.

First, if $y$ had been the root and a red child of $y$ becomes the new root, we have violated property 2. Second, if both $x$ and $p[y]$ (which is also $p[x]$) were red, then we have violated property 4. Third, $y$'s removal causes some path that previously contained $y$ to have one fewer black node. Thus, property 5 is now violated by any ancestor of $y$ in the tree. We can correct this problem by saying...
Figure 13.7  The cases in the while loop of the procedure RB-DELETE-FIXUP. Darkened nodes have color attributes BLACK, heavily shaded nodes have color attributes RED, and lightly shaded nodes have color attributes represented by $c$ and $c'$, which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. In each case, the configuration on the left is transformed into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by $x$ has an extra black and is either doubly black or red-and-black. The only case that causes the loop to repeat is case 2. (a) Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes $B$ and $D$ and performing a left rotation. (b) In case 2, the extra black represented by the pointer $x$ is moved up the tree by coloring node $D$ red and setting $x$ to point to node $B$. If we enter case 2 through case 1, the while loop terminates because the new node $x$ is red-and-black, and therefore the value $c$ of its color attribute is RED. (c) Case 3 is transformed to case 4 by exchanging the colors of nodes $C$ and $D$ and performing a right rotation. (d) In case 4, the extra black represented by $x$ can be removed by changing some colors and performing a left rotation (without violating the red-black properties), and the loop terminates.