ICS 311 Fall 2020, Problem Set 04, Topics 7 & 8

Firstname Lastname <uhumail@hawaii.edu> , Section 1
Due by midnight Tuesday Sept. 29nd. 40 points total.

#1. Master Method Practice (6 pts)

Use the Master Method to give tight Θ bounds for the following recurrence relations. Show *a*, *b*, and *f*(*n*). Then explain why it fits one of the cases, choosing ε where applicable. Write and *simplify* the final Θ result

(a) $T(n) = 3T(n/9) + n$

(b) $T(n) = 7T(n/3) + n$

(c) $T(n) = 2T(n/4) + \sqrt{n}$

#2. Solving a Recurrence (10 pts)

Solve the following recurrence by analyzing the structure of the recursion tree to arrive at a "guess" and using substitution to prove it. Justify all steps to show that you understand what you are doing.

$T(n) = T(2\sqrt{n}) + c$ if $n > 8$
$T(n) = c$ if $n \leq 8$

#3. Binary Search Tree Proof (11 pts)

The procedure for deleting a node in a binary search tree relies on a fact that was given without proof in the notes:

> **Lemma:** If a node X in a binary search tree has two children, then its successor S has no left child and its predecessor P has no right child.

In this exercise you will prove this lemma. Although the proof can be generalized to

duplicate keys, for simplicity assume no duplicate keys. The proofs are symmetric, so we start by proving for the successor. We rule out where the successor cannot be to narrow down to where it must be. Drawing pictures may help.

(a) Prove by contradiction that the successor S cannot be an ancestor or cousin of X, so S must be in a subtree rooted at X.

(b) Identify and prove the subtree of X that successor S must be in.

(c) Show by contradiction that successor S cannot have a left child.

(d) Indicate how this proof would be changed for the predecessor.

#4. Deletion in Binary Search Trees (5 pts)

Consider Tree-Delete (CLRS page 298), where `x.left`, `x.right` and `x.p` access the left and right children and the parent of a node x, respectively.

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4     TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)  // successor
 6      if y.p != z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
 10       TRANSPLANT(T, z, y)
 11       y.left = z.left
 12       y.left.p = y
```

(a) How does this code rely on the lemma you just proved in problem 3? Be specific, referring to line numbers. Be careful that you are using the actual lemma above, and not a similar fact proven elsewhere.

(b) When node *z* has two children, we arbitrarily decide to replace it with its successor. We could just as well replace it with its predecessor. (Some have argued that if we choose randomly between the two options we will get more balanced trees.) Rewrite Tree-Delete to use the predecessor rather than the successor. Modify this code just as you need to and underline or boldface the changed portions.

#5. Constructing Balanced Binary Search Trees (8 pts)

Suppose you have some data keys sorted in an array `A` and you want to construct a *balanced binary search tree* from them. Assume a tree node representation `TreeNode` that includes instance variables `key`, `left`, and `right`. (No `p` needed in this problem.)

(a) Write pseudocode (or Java if you wish) for an algorithm that constructs the tree and returns the root node. (We won't worry about making the enclosing BinaryTree class instance.) You will need to use methods for making a new TreeNode, and for setting its left and right children.

Hints: Think about how BinarySearch works on the array. Which item does it access first in any given subarray it is called with? How can we set up the tree so that a search sequence for a given key examines the same keys as it does in the array?

(b) What is the Θ time cost to construct the tree, assuming that the array has already been sorted? Justify your answer.

(c) Compare the expected runtime of BinarySearch on the array to the expected runtime of BST TreeSearch in the tree you just constructed. Have we saved time?