ICS 311, Spring 2016, Problem Set 04, Topics 7 & 8

Solutions

Due by midnight Tuesday 2/16

Copyright (c) 2016 Daniel D. Suthers & Nodari Sitchinava. All rights reserved. These solution notes may only be used by students in ICS 311 Spring 2016 at the University of Hawaii.

#1. Peer Credit Assignment

1 Point Extra Credit for replying

Please list the names of the other members of your peer group for class 1/25 and 1/27 and the number of points you think they deserve for their participation in group work on the two days combined.

- You have a total of 6 points to allocate across all of your peers.
- You can distribute the points equally, give them all to one person, or do something in between.
- You need not allocate all the points available to you.
- *You cannot allocate any points to yourself!* Points allocated to yourself will not be recorded.

#2. 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

2 pts each lettered problem, 0.5 for a,b,f(n); 0.5 for case and epsilon, and 1.0 for final solution.

(a) T(n) = 2T(n/4) + √n
Answer:
a=2, *b*=4, f(*n*) = $\sqrt{n} = n^{1/2}$.
Compare $n^{1/2}$ to $\Theta(n^{\log_4 2}) = \Theta(n^{1/2})$ ***(2 is the square root of 4)*.**
Case 2: T(*n*) = Θ(√*n* lg*n*).

(b) T(n) = 3T(n/9) + n

Answer:
a=3, b=9, f(n) = n.
Let ε = 1/2.
f(n) = n = Ω($n^{\log_b a + \varepsilon}$) = Ω($n^{1/2 + 1/2}$).
Check regularity: 3f(n/9) ≤ cf(n). Let c = 1/2. Then 3(n/9) ≤ n/2, for all n.
Case 3: T(n) = Θ(n).

(c) T(n) = 7T(n/3) + n
Answer:
a = 7, b = 3, f(n) = n.
Let ε = $\log_3 7 - 1$
f(n) = n = O($n^{\log_3 7 - \varepsilon}$)
Case 1: T(n) = Θ($n^{\log_3 7}$).

#3. Substitution (7 pts)

Use substitution *as directed below* to solve

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

It is strongly recommended that you read page 85-86 "Subtleties" before trying this!

(a) First, use the result from the Master Method in 2c as your "guess" and inductive assumption. (When removing Θ, just use one constant c rather than c_1 and c_2.) Take the proof up to where it fails and say where and why it fails. Remember that the steps are:

1. Write the definition of T(n)
2. Substitute the inductive hypothesis into this guess on the right hand side.
3. Simplify the right hand side algebraically to try to derive the exact form of your guess.

Answer (3 pts):
Let us guess T(n) = $cn^{\log_3 7}$

T(n) = 7T(n/3) + n	// definition of T(n)
= $7(c(n/3)^{\log_3 7}) + n$	// substitute inductive hypo
= $7c((n/3)^{\log_3 7}) + n$	// pull c out
= $7c(n^{\log_3 7} / 3^{\log_3 7}) + n$	// (a/b)^k = (a^k)/(b^k)
= $7c(n^{\log_3 7}/7) + n$	// definition of log
= $cn^{\log_3 7} + n$	// 7/7 = 1

But this is strictly greater than $cn^{\log_3 7}$. It has a term "n" that we want to get rid of.

(b) Redo the proof, but subtracting *dn* from the guess to construct a new guess. Don't forget to use the new guess when you do substitution. This time it should succeed: once you get close to your guess you will solve for d to determine the exact guess.

Answer (4 pts):
We'll get rid of that "n" by subtracting it from our guess. The subtracted "n" will be inside our inductive hypothesis, so we'll multiply it by some constant that will make it cancel the extra "n" once the algebra is worked out. We don't know what constant will make this work, but we'll just give it a name "d" and find out later.

guess $T(n) = cn^{\log_3 7} - dn$

$T(n) = 7T(n/3) + n$	// definition of T(n)
$= 7(c(n/3)^{\log_3 7} - dn/3)) + n$	// substitute inductive hypo
$= 7(c(n^{\log_3 7} / 3^{\log_3 7} - dn/3)) + n$	// (a/b)^k = (a^k)/(b^k)
$= 7(cn^{\log_3 7} / 7 - dn/3) + n$	// definition of log
$= cn^{\log_3 7} - (7/3)dn + n$	// multiply 7 through

We must get the exact form of the inductive hypothesis, $cn^{\log_3 7} - dn$.
When will $cn^{\log_3 7} - (7/3)dn + n$ be equal to $cn^{\log_3 7} - dn$?
When $-(7/3)dn + n = -dn$, or

$n + dn = (7/3)dn$
$1 + d = (7/3)d$
$1 = (4/3)d$
$d = 3/4$

Thus, with d = 3/4, we have shown that
$T(n) = cn^{\log_3 7} - dn = cn^{\log_3 7} - (¾)n$.
(We had a "d" term to allow us to make this adjustment!)

#4. Binary Search Tree Proof (7 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. (*Hints: Rule out where the successor cannot be to narrow down to where it must be. Draw Pictures!!!*)

(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.

Answer (4 pts, one for each statement or equivalent):

- The node X cannot be a right child or right descendent of its successor S because then X would have a greater key than S (or could potentially if we allow equal keys), contradicting the assumption that S is a successor.

- The node X cannot be a left child or left descendent of its successor S because X has a right child and the nodes in the right subtree of X have keys greater than X, but they must also be smaller than their ancestor S, meaning for some node Z in the right subtree of X, we can have X < Z < S, contradicting the assumption that S is the successor of X

- X and S cannot be "cousins", sharing a common parent or ancestor, because then the cousin would be between X and S, contradicting successorship of S.

- Thus, S cannot be above X in the tree; nor can it be in a different subtree; it must be below X.

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

Answer (1pt):
By the above argument, S is in the subtree rooted either at the left child of X or the right child of X. If it was rooted at the left child of X, then by BST property
S < X, which violates the definition that S is successor of X. Therefore, S must be in the subtree rooted at the right child of X.

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

Answer (1pt):
Assume S has a left child Y. Then Y < S. But we know that S is in the subtree rooted at the right child of X, and by BST property all nodes in that subtree have keys larger than X.key. Then X < Y < S. I.e. S cannot be a successor of X -- a contradiction.

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

Answer (1pt):
Change "successor" to "predecessor"; swap "left" and "right"; and

flip the inequalities, including the words "smaller" and "greater" as well as < and >.

#5. Deletion in Binary Search Trees (3 pts)

Consider Tree-Delete (page 298, and also copied below).

```
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?

Answer (1 pt):
Line 11 replaces y.left with the subtree under z.left. The lemma assures us that we are not overwriting (losing track of) any subtree as y.left is empty.

I suspect we will also get answers saying that line 5 assumes that the successor will be the leftmost child (and hence having no left child) in z's right subtree. This is actually a different fact, proven page 291: if there is a right subtree the successor must be the minimum of this tree. Half point for this.

(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.

Answer (2 pts, 0.25 for each instance I suppose. Note: the code will still be correct if

ALL of the instances of ‘left’ and ‘right’ are swapped, even in lines 1-4, but change to lines 1-4 is not necessary.**)**

```
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-MAXIMUM(z.left)  // Predecessor
 6      if y.p != z
 7          TRANSPLANT(T, y, y.left)
 8          y.left = z.left
 9          y.left.p = y
 10       TRANSPLANT(T, z, y)
 11       y.right = z.right
 12       y.right.p = y
```

#6. Constructing Balanced Binary Search Trees (7 pts)

Suppose you have some data keys sorted in an array 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`.

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: First, identify the array location of the key that would have to be the root of the balanced BST. Now think about how BinarySearch works on the array. Which item does it access first in any given subarray it is called with? Using a similar strategy a simple recursive algorithm is possible.

Answer (5 pts):
It’s like binary search, except that you go into BOTH sides to build subtrees around the middle element (and so don’t need to compare keys). For example:

```
BuildBalancedBST (A,low,high)
     if low > high // subtree is empty
```

```
    return NIL
mid = floor(low + (high - low)/2) // divide into two equal
parts
leftTree = BuildBalancedBST (A, low, mid-1)
rightTree = BuildBalancedBST (A, mid+1, high)
return new TreeNode (A[mid], leftTree, rightTree)
```

b. What is the Θ cost to construct the tree? Justify your answer.

Answer (1 pt):
To construct the tree it takes Θ(n) time. Each call removes element of the input in constant time and recurses on the rest, building the result in constant time. There are no more than n
elements to remove. Note: Θ(n lg n) if we include the cost to sort.
More formally, the recurrence is T(n) = 2T(n/2) + O(1), and with the Master Method, f(n) = O(1) is $O(n^{\log 2 - e}) = O(n^{1-e})$, so T(n) = $\Theta(n^{\log 2}) = \Theta(n)$.

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?

Answer (1 pt):

Expected time of BinarySearch is O(lg n); expected time of search in the BST is also O(lg n) as it is a balanced tree.

So you don't save any big-O time; it's only worth the Θ(n) time if you need the actual tree.