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

Solutions

40 points total. Point allocation for problem parts are for TA guidance only: TAs may redistribute if needed to reflect how the student response is structured with appropriate credit.

#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

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

(a) $T(n) = 3T(n/9) + n$
Solution:
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) ≤ *c*f(*n*). Let *c* = 1/2. Then 3(*n*/9) ≤ *n*/2, for all *n*.
Case 3: T(*n*) = Θ(*n*).

(b) $T(n) = 7T(n/3) + n$
Solution:
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($n^{\log_3 7}$).

If we have a divide and conquer recurrence of the form

$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1, b > 1$$

then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

(c) $T(n) = 2T(n/4) + \sqrt{n}$
Solution:
a = 2, b = 4, f(*n*) = $\sqrt{n}$ = $n^{1/2}$

f(n) = $n^{1/2}$ = Θ($n^{log_4 2}$)
Case 2: T(n) = Θ($n^{log_4 2}\ lg\ n$) = Θ($\sqrt{n}\ lg\ n$)

#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. (Do not try to use the Master Method.) Justify all steps to show that you understand what you are doing.

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

Solution: (The following is adapted from Nodari's nice solution. Googling the problem will return another solution based on a change of variables. If they use that, it goes against our instructions and is probably plagiarism.)

To make a reasonable guess, let's see how many layers are in the recursion.

For an input *n*, the first recursive call is on size n^(½), then n^(¼), then n^(⅛), etc. In general, the *k*-th level of the recursion is called on n^(½^k). At the bottom layer, we have n^(½^k)=2, so we can solve for k:

2 = n^(½^k)
lg 2 = (½^k) lg n Take the log base 2 of both sides.
1 = (½^k) lg n
2^k = lg n
k = lg lg n Take the log base 2 of both sides again

So the depth of the recursion is k = lg lg n. Since each level of the recursion requires *c*, a good guess of the runtime is to multiply c by the number of layers: T(n) = c lg lg n. We test this using substitution.

T(n) = T(√n) + c Recurrence relation (given)
= c · lg lg(√n)) + c Substitution
= c · lg lg(n^0.5) + c
= c · lg(0.5 lg n) + c
= c · (lg(1/2) + lg lg n) + c
= c · (− 1 + lg lg n) + c
= − c + c · lg lg n + c

$= c \cdot \log \log n$ QED

#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. (*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 (5 pts as indicated for these logical moves or their equivalent):

- **(1)** 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.

- **(2)** 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

- **(1)** 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.

- **(1)** 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 (2pts):
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 (2pts):
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 the predecessor.

Answer (2pts):
Change "successor" to "predecessor"; swap "left" and "right"; and
flip the inequalities, including the words "smaller" and "greater" as well as <
and >.

#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.

Answer (2 pts):
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.

(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 (3 pts, 0.5 for each line modified as shown below. 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
```

#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?

Answer (5 pts, 1 for each line of code or equivalent logic below):
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, the following would be called with the low and high indices of the data in the array:

```
BuildBalancedBST (A,low,high)
1  if low > high return NIL          // subtree is empty
2  mid = floor(low + (high - low)/2) // divide into two equal parts
3  leftTree = BuildBalancedBST (A, low, mid-1)
4  rightTree = BuildBalancedBST (A, mid+1, high)
5  return new TreeNode (A[mid], leftTree, rightTree)
```

(Code need only be logically equivalent, not exact. For example, they might write 3, 4 and 5 as one nested call without the temporary variables leftTree and rightTree: give the 3 points for the logic of the two calls and for combining the results correctly.)

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

Answer (2 pts):
To construct the tree it takes Θ(n) time. Each call removes one 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.