**EE 367 Spring 2018**

**Homework 4 Total Points = 23**

**Problem 4.3-2** (page 87, 1 pt)

**Problem 4.3-8** (page 87, 1 pt)

**Problem 4.3-9** (page 88, 1 pt)  *[ You can use the substitution method and the Master theorem]*

**Problem 4.5-1** (page 96, 1 pt)

**Problem 4.5-4** (page 97, 1 pt)

**Problem 4.5(b,c)** (skip part (a), page 109, 1 pt*) [ Hint: Consider 3 chips and 2 are good and one is bad. Then consider 5 chips with 3 good and 2 bad.]*

**Problem 12.2-3** (page 293, 1 pt)

**Problem 12.2-6** (page 293, 1 pt) *[ This is confusing to read. Do some examples by drawing trees. This is related to the algorithm Tree-Successor on page 292, and that in some cases it must go up the tree to an ancestor to find a successor. ]*

**Problem 12.2-7** (page 293, 1 pt)

**Problem 12.3-3** (page 299, 1 pt)

**Problem 12.3-6** (page 299, 1 pt) *[Problem 12-1(b) on page 304 has a suggestion to implement a fair strategy ]*

**Problem 18.2-2** (page 497, 1 pt)

**Problem 18.2-7** (page 499, 1 pt) *[ Assume that the time to do a search is completely dominated by disk reads. ]*

The next Problem A is about operations in a 2-3 4 tree (this is a B-tree with minimum degree t = 2). To help understand the operation of insertion, the following is an example of the evolution of a tree as the following values are added in the order given:

C, O, R, N, F, L, A, K, E, S

C O R

C N

O

R

C F N

O

R

C

F O

R

L N

A C E

F O

R S

K L N

**Figure 1**. 2-3-4 Tree.

**Problem A** (1 pt). Show the evolution of a 2-3-4 tree when the input is in the following order:

S, E, K, A, L, F, N, R, O, C. You are not required to show every change, just the changes that lead to changes in the number of nodes, and also include the final tree.

**Problem B**. (1 pt). Consider the 2-3-4 tree in Figure 2. Delete “’J” and show what the final tree looks like. (Assume that nodes are merged only if rotating is not possible.)

G Q

B D

K

T

A

C

F

J

M

S

X

**Figure 1**. Another 2-3-4 Tree.

**Problem C**. (1 pt). Consider the 2-3-4 tree in Figure 2. Delete “’G” and show what the final tree looks like. (Assume that nodes are merged only if rotating is not possible.)

**Problem D** (1 pt) Show that equals .

**Problem E** (1 pt) Note that (this is a formal way to show that log(y) ≈ y-1 when y ≈ 1). Use this to show that .

**Problem F** (1 pt)

R

A

M

Y

L

X

F

If we were to traverse the tree, the list of visited nodes in pre-order is (here NULL means the node has no left-child or no right-child):

R, A, Y, NULL, NULL, L, NULL, NULL, M, NULL, F, X, NULL, NULL, NULL

Consider *another* binary tree T. If you were to traverse the tree, the list of visited nodes in pre-order is:

D, F, NULL, A, E, NULL, NULL, NULL, B, C, NULL, NULL, G, NULL, NULL

Draw the binary tree T.

**Problem G** (4 pts). Attached is a program tree.c that does the followning:

**Step 1**. It queries the user for the name of a data file that represents a binary tree (not necessarily a binary search tree). Here you should type “hw4.dat” which has data that represents the following tree (“hw4.dat” is attached). Note that the keys of nodes are alphabets.

a

b

c

root

d

e

y

h

x

f

g

z

**Figure 3**. Binary tree in hw4.dat.

**Step 2**. It displays the tree as follows

Tree root = a

a: b c

b: d e

d: y /

y: / /

e: h x

h: / /

x: / /

c: f g

f: / z

z: / /

g: / /

Each line indicates a node followed by its left child and right child. For example, the first line is for node ‘a’, and its left child is ‘b’ and its right child is ‘c’. The symbol ‘/’ means NULL. So for example, ‘x’ is a leaf.

**Step 3.** The program will *reverse* the binary tree so that a node’s children are switched, i.e., its left child becomes its right child and vice versa.

**Step 4.** It displays the tree as follows

Tree root = a

a: c b

c: g f

g: / /

f: z /

z: / /

b: e d

e: x h

x: / /

h: / /

d: / y

y: / /

Here, you can find that node ‘c’ has its children switched so it’s left child is ‘g’ and right child is ‘f’ whereas originally it’s left child was ‘f’ and its right child was ‘g’.

In the next steps, the program will find the lowest common ancestors of two nodes. For example, the lowest common ancestor of ‘x’ and ‘y’ in Figure 3 is ‘b’. The lowest common ancestor of ‘b’ and ‘h’ is ‘b’ because we will assume that a node is an ancestor of itself.

**Step 5.** It will display the lowest common ancestor of ‘x’ and ‘y’ which will be ‘b’

**Step 6.** It will display the lowest common ancestor of ‘b’ and ‘h’ which will be ‘b’

**Step 7**. It will attempt to find the lowest common ancestor of ‘x’ and ‘u’ but since there is none, it’ll indicate that no common ancestor exists.

The program tree.c is incomplete, and in particular the following functions:

* void reverse\_tree(struct node \*x): reverses the binary tree rooted at x
* struct node \* common(struct node \*x, char c0, char c1: Finds in a binary tree rooted at x, the lowest common ancestor of nodes with keys c0 and c1. It returns a pointer to the ancestor, or NULL if the ancestor does not exist.

Implement these functions. Note that “common” may require writing more functions and possibly structs.

Implementing ‘reverse\_tree’ is not too difficult. Implementing ‘common’ is more difficult.

**Submit your program tree.c by uploading it into laulima.**