**Chapter 8:**

**R-81.**

**a. Which node is the root?**

/user/rt/courses/is the root.

**b. What are the internal nodes?**

/user/rt/courses/,cs016/, cs252/, homeworks/, programs/, projects/, papers/, demos

**c. How many descendants does node cs016/ have?**

9 descendants and dashes:

grades, homeworks/, programs/, hw1, hw2,hw3, pr1, pr2, pr3.

**d. How many ancestors does node cs016/ have?**

1 ancestor:

/user/rt/courses/.

**e. What are the siblings of node homeworks/?**

grades and programs

**f. Which nodes are in the subtree rooted at node projects/?**

Papers, demos, buylow, sellhigh, and market

**g. What is the depth of node papers/?**

3

**h. What is the height of the tree?**

4

**R-8.4 What is the running time of a call to T. height2(p) when called on a**

**position p distinct from the root of T? (See Code Fragment 8.5.)**

**def height2(self, p): # time is linear in size of subtree**

**if self.is leaf(p):**

**return 0**

**else: 66 return 1 + max(self. height2(c) for c in self.children(p))**

**Code Fragment 8.5: Method height2 for computing the height of a subtree rooted at a position p of a Tree.**

**Running time:** O(n): each node is accessed once

**R-8.21 In what order are positions visited during a preorder traversal of the tree of Figure 8.8?**

(-),(/),(x),(+),(3),(1),(3),(+),(-),(9),(5),(2),(+),(x),(3),(-),(7),(4),(6)

19 nodes

(follow the tree from the leftmost element)

**C-8.42 Describe how to clone a LinkedBinaryTree instance representing a (not necessarily proper) binary tree, with use of the add left and add right methods.**

Def \_copy( self, p)

If p == None:

Return None

New\_p =self.Node(p.\_element)

New\_p.\_left = self.\_copy(p.\_left)

New\_p.\_right = self.\_copy(p.\_right)

Return new\_p

Def copy(self)

New\_root = self.\_copy(self.\_root)

Return BinaryTree(new\_root)

This is the method I worked on in class, with refinement from the professor.

**Chapter 9**

**R-9.3 What does each remove min call return within the following sequence of priority queue ADT methods: add(5,A), add(4,B), add(7,F), add(1,D), remove min( ), add(3,J), add(6,L), remove min( ), remove min( ), add(8,G), remove min( ), add(2,H), remove min( ), remove min( )?**

(1,D), (3,J), (4,B), (5,A), (2,H), (6,L)

**R-9.5 The min method for the UnsortedPriorityQueue class executes in O(n) time, as analyzed in Table 9.2. Give a simple modification to the class so that min runs in O(1) time. Explain any necessary modifications to other methods of the class.**

**class Entry(object):**

**def getKey(self):**

**pass**

**def getValue(self):**

**pass**

**def insert(self, key, value):**

**checkKey(key) # auxiliary key-checking method (could throw exception)**

**newest = New Entry<>(key, value)**

**list.addLast(newest)**

**return newest**

**def min():**

**if(list.isEmpty()) return None**

**return findMin().getElement()**

**def removeMin():**

**if list.isEmpty():**

**return None**

**return list.remove(findMin())**

**R-9.21 Show all the steps of the algorithm for removing the entry (16,X) from the heap of Figure 9.1, assuming the entry had been identified with a locator**

First we delete a node from the array. Then we replace that node with the right-most node from the lowest level of nodes on the binary tree. We have to replace it with the parent node because the value of that replacement node is less than the parent node.

**C-9.26 Show how to implement the stack ADT using only a priority queue and one additional integer instance variable**

I did this in the given stack class and just copied the relevant code here without the full class or main.

**# Stack ADT**

**def push(self, v):**

**self.counter += 1**

**self.insert(Integer(-1\*self.counter), v)**

**def pop(self):**

**self.counter -= 1**

**return removeMin()**