**C-2.8**

We need to maintain two int ‘head’ and ‘tail’ representing the start of the array and the end respectively.

Also a capacity variable ‘cap’ is also needed. We need a circular queue array.

**Algorithm** get (index) :

return element at (head + index) mod cap;

**Algorithm** insertion(element, index):

If ((cap – head + tail) mod cap = cap - 1) then return Error: Queue full

If (index = head) then Q[head] <- element

head <- (head - 1) mod cap

If (index = tail) then Q[tail] <- element

tail <- (tail + 1) mod cap

//other index not concerned

return

**Algorithm** remove(index):

If (head = tail) then return Error: Queue empty

If (index = head) then

Temp <- Q[head]

Q[head] <- null

head <- (head + 1) mod cap

If (index = tail) then

Temp <- Q[tail]

Q[tail] <- null

tail <- (tail - 1) mod cap

//other index not concerned

return Temp

**C-2.20**

**Algorithm** levelOrderTraversal (BinaryTree T):

Q = new Queue()

Q.enqueue(T root())

while (! Q.isEmpty()) do

v <- Q.dequeue()

if (isInternal(v)) then

Q.enqueue(leftChild(v))

Q.enqueue(rightChild(v))

This would scan through each element one time, thus linear

**A-2.2**

**Algorithm** LCA (BinaryTree root, BinaryTree x, BinaryTree y):

if (root = null) or (x = root) or (y = root) then

return root

BinaryTree left <- LCA (leftChild(root), x, y)

BnaryTree right <- LCA (rightChild(root), x, y)

if left = null then return right

if right = null then return left

return root

The running time should be O(height).

**R-3.6**

**Algorithm** findSmallest (BinaryTree root):

s <- root

while leftChild(s) != null do

s <- leftChild(s)

return s

The running time is also O(height).

**C-3.3**

**Algorithm** findAllElements (Key k, BinaryTree root):

if isExternal(root) then return null

if key(root) = k then

left <- findAllElements (k, leftChild(root))

right <- findAllElements (k, rightChild(root))

return left and root and right

else if key(root) < k then

return findAllElements (k, rightChild(root))

else

return findAllElements (k, rightChild(root))

Binary search cost O(log n), while array.length = n. Assume we’ve found s number of duplicates then the total time should be O(log n + s)

**A-3.2**

Use the element at as the root. then use recursive function to place the nodes from A[0] to and from to A[n-1]. For each element the recursive function should use O(1) to move them, thus, the total running time is O(n).

**R-4.4**

For set {1,2,3,4} the AVL tree is

2

1 3

4

For set {4,3,2,1} the AVL tree is

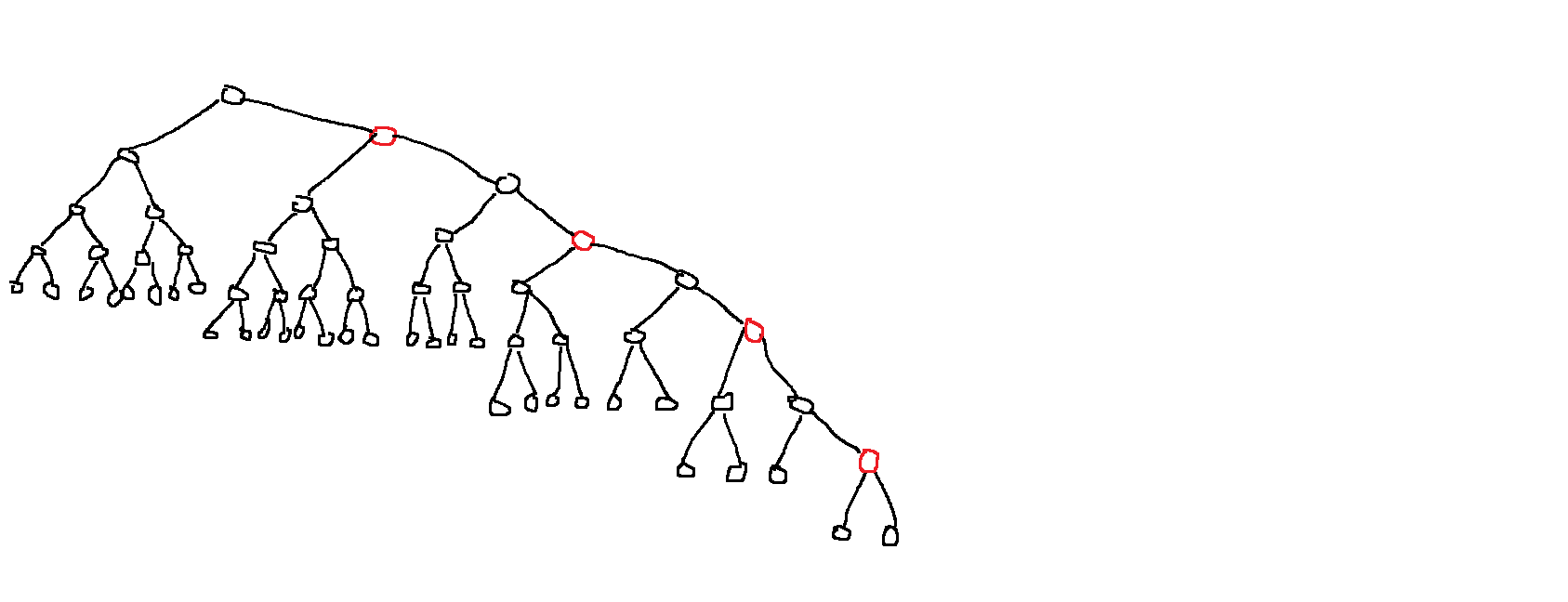
3

2 4

1

At least they have different root, which makes the Prof wrong.

**R-4.7**

To get the minimum number of nodes we should make sure the right-most path have the largest amount of possible red nodes, thus the number of black nodes we needed would be minimized.

Note that all the external nodes in the picture are null.

This tree is done following the rules I said with height 8 and 61 nodes. Thus, the minimum nodes is 61.

**A-4.4**

Functions needed:

1. List out names ordered by ZIP
2. Within each ZIP, names are ordered in alphabetical.
3. List out names with given ZIP.

We can store the ZIP codes and names in a costumed linked list with two values: an integer ZIP code and a string list with names ordered alphabetically. We use a balanced binary search tree to store the ZIP codes.

To list out names ordered with ZIP, we iterate every ZIP codes and print out each of the names corresponding to the code. Since we need to go over each names, the running time would be O(n).

To list out names in a given ZIP, we need O(log n) to get to the particular node, then we need O(s) time for s names returned. Thus, we need O(log n + s) running time in total. The insertion and removals would go to a specific node, which cost O(log n).