Problem Set 8, Part I

Problem 1: Checking for keys above a value

1-1) The time efficiency of this algorithm is O(n). This recursive algorithm returns true only if it finds a key greater than the integer input, so it will check for every key in the tree if there is no node greater than the input. The closer the first possible greater node is to the root node, the earlier the algorithm will stop, so the time complexity is determined by the keys and shape of the tree.

The best-case scenarios are 1) in a tree that at least has a complete level 0 and level 1, the first possible greater node is the left child of the root, and the left child is a leaf node; 2) in an unbalanced tree whose root node does not have a left child, the first possible greater node is the right child of the root. In the above situations, since the operation in each recursive call is almost 1 step, and there are few recursive calls, the big-O notation will be O(1).

The worst-case scenario for a balanced tree is that the algorithm cannot find any greater node or the first possible greater node is at the rightmost bottom-most. The worst-case scenario for an unbalanced tree is that in a tree that only has right children and whose nodes are added from a sorted array, the algorithm cannot find any greater node or the first possible greater node is at the bottom-most. In either case, the algorithm has to go down every node in the tree with a length of n, so the big-O notation for the worst cases is O(n)

1-2)

```
private static boolean anyGreaterInTree(Node root, int v) {
      if (root == null) {
            return false;
      } else if (root.key > v) {
            return true;
      } else {
            boolean anyGreater = anyGreaterInTree(root.right, v);
            return anyGreater;
      }

}
```

1-3) The time complexity of the updated algorithm is O(log(n)). Same as the old version, the new algorithm hits the base case and returns true only if it finds a key in the right subtrees greater than the integer input, so its time efficiency is still determined by the keys of the tree. The shape of the tree is no longer a limiting factor, because the updated algorithm only checks for the right children.

The best-case scenario is that regardless of the shape of the tree, the root itself or the first right child of the root is the first possible greater node. Again, since the operation in each recursive call is almost 1 step, and there are few recursive calls, the big-O notation will be O(1).

The worst-case scenario for a balanced tree is that it goes down every node in the right subtree and does not find a greater node. The worst-case scenario for an unbalanced tree is that in a tree that only has right children and whose nodes are added from a sorted array, the algorithm cannot find any greater node after going down the entire tree, or the first possible greater node is at the bottom-most. As both worst cases only consider right subtrees, this

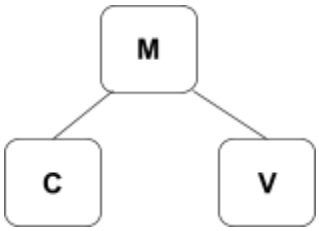
algorithm divides the problem into half for each recursive call~~, so the big-O notation for the worst cases is O(log(n)).~~ So, the big-O notation for the worst case of a balanced tree is O(logn), and that for the worst case of an unbalanced tree is O(n).

Problem 2: Balanced search trees

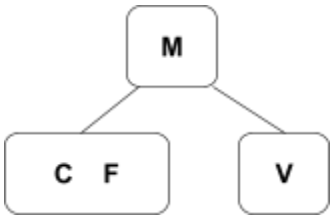
Adding V:

Adding C:

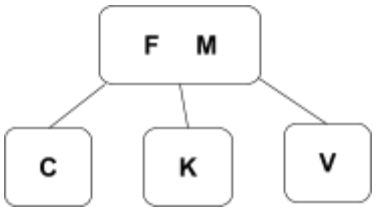
Adding M:



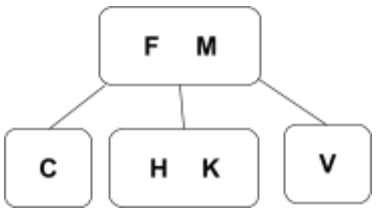
Adding F:



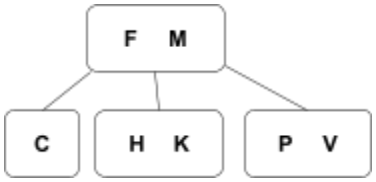
Adding K:



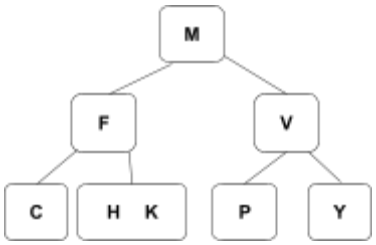
Adding H:



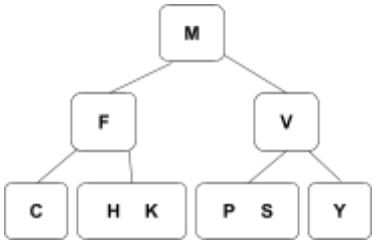
Adding P:



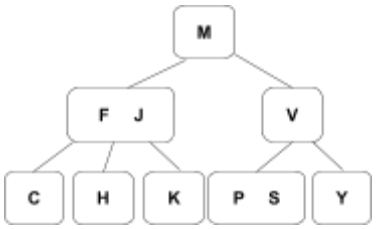
Adding Y:



Adding S:



Adding J:



Problem 3: Hash tables

3-1) linear

0	you
1	a
2	to
3	the
4	my
5	their
6	bring
7	do

3-2) quadratic

0	
1	bring
2	to
3	the
4	
5	their
6	my
7	

3-3) double hashing

0	bring
1	do
2	to
3	the
4	
5	their
6	my
7	a

For 3-1, “go” causes overflow since there is no more space to add a new key to the hash table.

For 3-2, “do” causes overflow since its probing sequence is 2-3-6-3-2-3-6-3-2…, which is a loop of 2-3-6-3. As positions 2, 3, and 6 have been occupied, and the probing does not jump to any empty cell, “do” cannot be added to the hash table.

For 3-3, “you” causes overflow since its probing sequence is 3-7-3-7-3-7…, which is a loop of 3-7. As positions 3 and 7 have been occupied, and the probing does not jump to any empty positions, “you” cannot be added to the hash table.

3-4) probe sequence: try 3, 6, 9%8 = 1(hit the first removed cell), 12%8=4, 15%8=7(hit the first never-been-removed empty cell)

3-5) table after the insertion:

0	list
1	try
2	
3	our
4	
5	
6	linked
7	

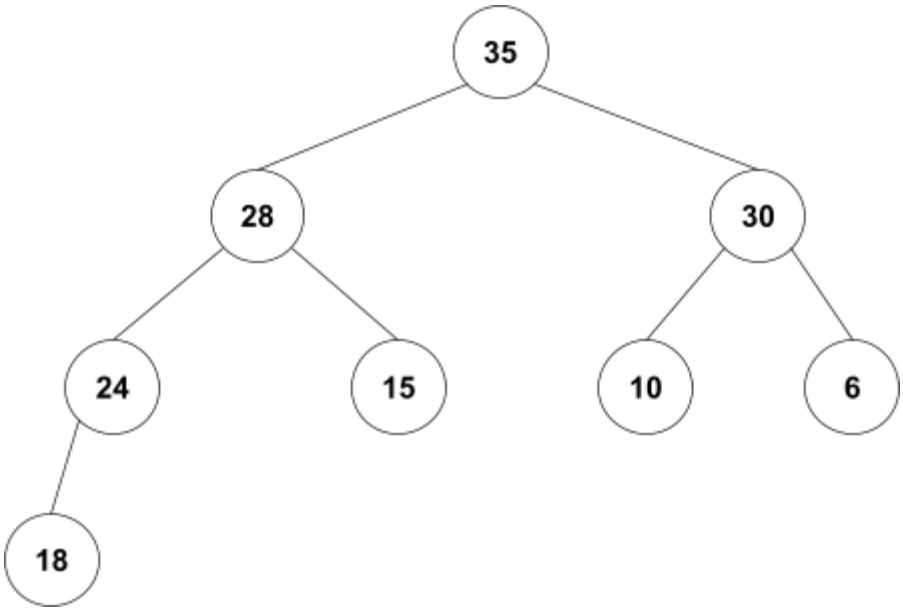
Problem 4: Complete trees and arrays

4-1) left child: at position **81** of the array (40 * 2 + 1)
right child: at position **82** of the array (40 * 2 + 2)
parent: at position **19** of the array ((40-1) / 2)

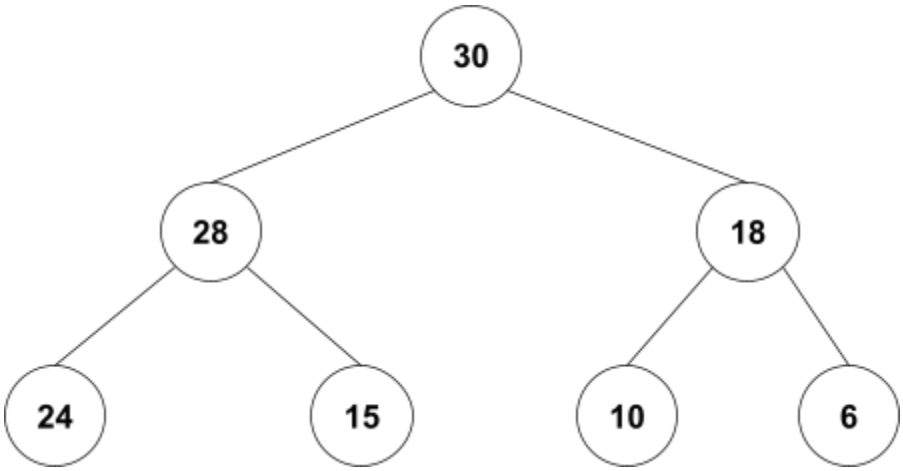
4-2) In a complete tree, the number of nodes doubles from a level to the next level, which means the number of nodes from level 0 to level (h-1) equal the sum of a geometric sequence with a common ratio of 2 and with an unknown number of terms. Therefore the sum should be 2^n - 1, where n is the number of terms. By equating 2^n-1 to 112, we solve n by calculating log base 2 of 112, which turns out to be approximately 6.81. This means that the (h-1) level of the complete tree is level 6, and level 7 is the final level which has not been completed. Thus, the height of this complete tree is 7.

4-3) The rightmost leaf node in the bottom level is node 111. Let's assume that node 111 is a right child of its parent, and its parent is node x, then 2*x + 2 = 111. We solve this equation, and x equals 54.5. Since every node in this tree is an integer node, x cannot be 54.5, and we cannot perform rounding to 54.5. Therefore, node 111 is not a right child. If node 111 is a left child of its parent x, then 2*x + 1 = 111, and x must be an integer. It turns out that x equals 55, so node 111 is a valid left child of node 55.

Problem 5: Heaps
5-1)
after one removal



after a second removal
(copy your revised diagram from part 1 here, and edit it to show the result of the second removal)



5-2)