Question 2

Consider a binary search tree that contains n nodes with keys 1, 2, 3, ... , n.The shape of the tree depends on the order in which the keys have been inserted in the tree.

a) In what order should the keys be inserted into the binary search tree to obtain a tree with minimal height?

Keys should be inserted beginning with the middle key at the root and the middle of the halves below and above the middle in the left and right child respectively. Then repeat such that each child key is the middle of the half above or below the parent depending on whether it is the left or right child.

b) On the tree obtained in (a), what would be the worst-case running time of a find, insert, or remove operation? Use the big-Oh notation.

Worst case running times:
Find: $O(\log n)$
Insert: $O(\log n)$
Remove: $O(\log n)$

c) In what order should the keys be inserted into the binary search tree to obtain a tree with maximal height?

The root of the tree begins at one end (e.g. 1 or n) and the right child will be the next greater key when beginning at 1 or the left child is the next smaller beginning at n. Then repeat such that all left children are the next key smaller than the parent or all right children are the next key greater than the parent.

d) On the tree obtained in (c), what would be the worst-case running time of a find, insert, or remove operation? Use the big-Oh notation.

Worst case running times:
Find: $O(n)$
Insert: $O(n)$
Remove: $O(n)$

Question 3

Consider the following problem: Given an unsorted array A[0...n-1] of distinct integers, find the k-th smallest element (with k=0 giving the smallest element). For example, findKth([9 10 6 7 4 12] , 0) = 4 and findKth([1 7 6 4 15 10] , 4) = 10.

a) Complete the following pseudocode for the findKth algorithm.

Algorithm: findKth(A, start, stop, k)
Input: An unsorted array A[start...stop] of numbers, and a number k between 0 and stop-start-1
Output: Returns the k-th smallest element of A[start...stop]

```
//sort the array
//kth element is (start+k)

if (start<stop) then
   pivot=partition(A,start,stop)
   if (start+k==pivot) then return pivot
   else if (start+k<pivot) then return findKth(A,start,pivot-1,k-start)
   else if (start+k>pivot) then return findKth(A,pivot+1,stop,k-pivot-1)
```

b) …show that your algorithm runs in time O(n).

Assumptions (as given by question):

1. $n = 2^k \text{ } where \text{ } k \in Z \therefore k = \log_2 n$
2. "pivot" is always half way between start, stop
3. partition(start,stop) takes O(start-stop-1) time to run

In the worst case of running the algorithm, the recursive portion and pivot will run k times, but the running time of pivot changes and is the sum of $O(1) + O(2) + O(4) + \cdots + O(2^k)$. So running time looks more or less like this:

$$T(n) = \sum_{i=0}^{k} 2^i + ck + d; c \text{ } and \text{ } d \text{ } are \text{ } some \text{ } constants$$
$$T(n) = \frac{1 - 2^{\log_2 n}}{1 - 2} + c\log_2 n + d$$
$$T(n) = \frac{1 - n}{-1} + c\log_2 n + d$$
$$T(n) = n - 1 + c\log_2 n + d$$

My math probably has a mistake in there somewhere, but it doesn't change the general idea that the n component dominates the $c\log_2 n$, and so the running time is $O(n)$.

Question 4

a) Write the output being printed when weirdPreOrder(root) is executed on the following binary tree:

7 9 7 8 3 6 2 2 1

b) Write the output being printed when weirdPostOrder(root) is executed on the following binary tree:

9 9 8 7 6 2 1 2 3

c) Write the output being printed when queueTraversal(root) is executed.

7 3 9 2 6 8 9 1 2 7

d) Write the output being printed when stackTraversal(root) is executed. This is similar to what traversal method seen previously in class?

7 3 2 1 2 6 8 7 9 9

This is similar to the pre-order traversal method as seen in class.

Question 5

Write a recursive algorithm that tests if the trees rooted at two given treeNodes are isomorphic. Hint: if your algorithm takes more than 10 lines to write, you're probably not doing the right thing.

Algorithm isIsomorphic(treeNode A, treeNode B)
Input: Two treeNodes A and B
Output: Returns true if the trees rooted at A and B are isomorphic

```
//node.gLC, node.gRC represent node.getLeftChild() and node.getRightChild()
//respectively, while node.gV represents node.getValue()
//I'm also assuming I don't have to worry about null pointer exceptions since
//this is pseudocode

if (A==null AND B==null) then return true
else if (A.gV==B.gV) then
   if (A.gLC.gV==B.gLC.gV OR A.gLC.gV==B.gRC.gV) then
      if ( (isIsomorphic(A.gLC,B.gLC) AND isIsomorphic(A.gRC,B.gRC)) OR
         _(isIsomorphic(A.gRC,B.gLC) AND isIsomorphic(A.gLC,B.gRC) )
         return true
      else return false
else return false
```

Question 6

Let T be a heap storing n keys. Give the pseudocode for an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T).

Algorithm printAtMost(key x, treeNode n)
Input: a key x and a treeNode n (!)
Output: Prints the keys of all nodes of the subtree rooted at n with keys at most x

```
//Again, I’m not worrying about null pointer exceptions as I would in Java.
//I am also assuming a method call, getValue(), and getLeft/RightChild() are
//O(1)
//Algorithm runs in O(k+1) because if will not go into the “then” part of
//the if condition when the key > x so the number of operations performed
//will be at most 6k (three comparisons, two method calls, one print) plus
//the first comparison from the intial method call

if (n.getValue()<=x) then
   print n.getValue()
   printAtMost(x,n.getLeftChild())
   printAtMost(x,n.getRightChild())
```