Problem Set 6, Part I

Problem 1: Checking for keys above a value

1-1) O(n) because the anyGreaterInTree method traverse all the keys even if the greater value is found.

1-2)

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

}
```

1-3)

The efficiency is depending on the keys.

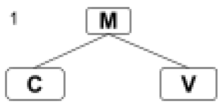
The Best case is O(1) when the root is larger than v.

The Worst case balanced is O(n) when the method have to go through all the nodes.

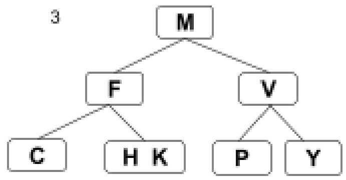
The Worst case unbalanced is O(n) when the method have to go through all the nodes.

Problem 2: Balanced search trees

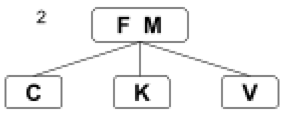
1



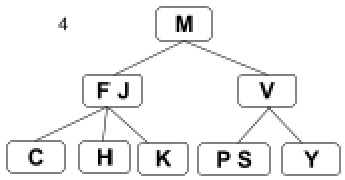
3



2



4



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	do
5	their
6	my
7	

3-3) double hashing

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

3-4) probe sequence: 3, 6, 1, 4, 7

3-5) table after the insertion:

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

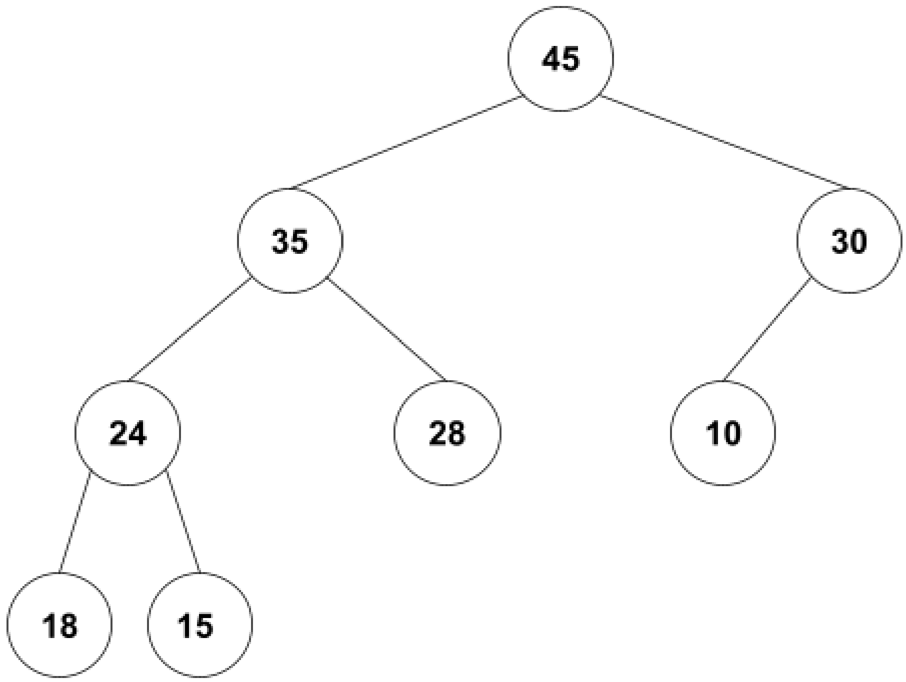
Problem 4: Complete trees and arrays

4-1) left child: 81, a[2*40 + 1]
right child: 81, a[2*40 + 2]
Parent:16, a[(40 - 1) / 2]

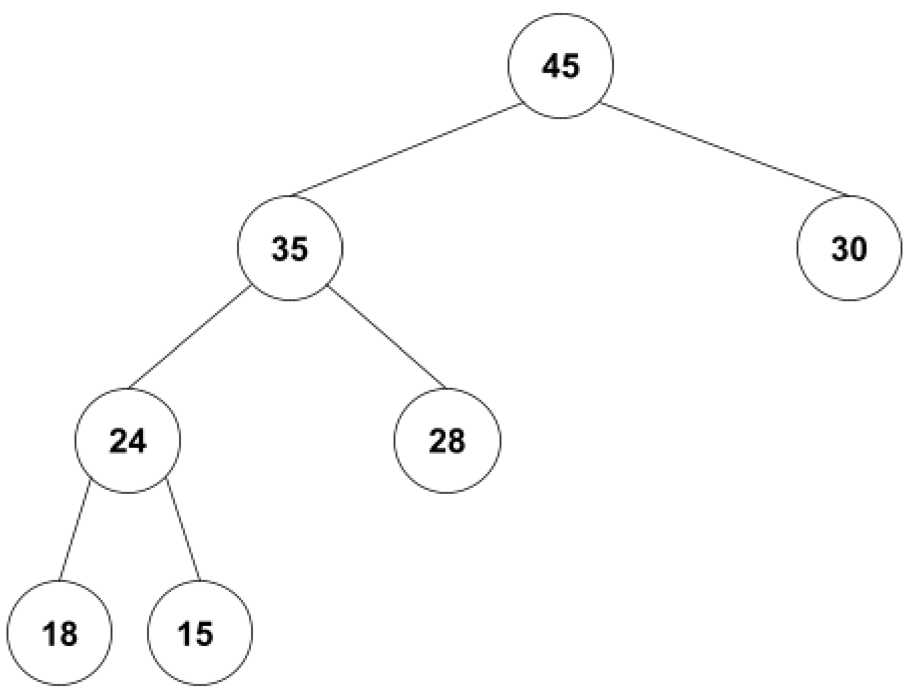
4-2) 6, Floor(log2(112))

4-3) Left child, because node 111 is the bottom node, the parent node is a[110 / 2] = 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)