Problem Set 8, Part I

Problem 1: Checking for keys below a value

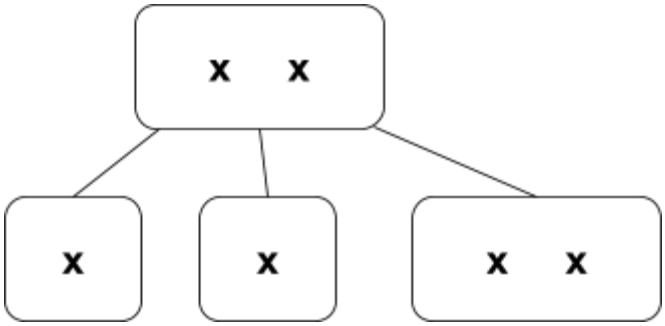
1-1) The best case time efficiency of this algorithm would be O(1) if the root node is null or empty. The worst case for a balanced tree would be O(n) when the value v is larger than every single value in the tree so the algorithm would need to traverse through every node of the tree. The worst case for an unbalanced tree would also be O(n) because there is one side is larger, height-wise, it would still have to traverse through the whole tree.

1-2)

```
private static boolean anySmallerInTree(Node root, int v) {
      if (root == null) {
            return false;
      } else if (root.key < v) {
            return true;
      } else {
            return anySmallerInTree(root.left, v);
      }
}
```

1-3) The best case time efficiency is O(1) if the tree has a null root. The worst-case time efficiency if the tree is balanced is O(log n) because it would traverse through the left subtree every time until true or false is reached. The worst case for an unbalanced tree would be O(log n) as well because it would still traverse through the left subtree of every recursive call.

Problem 2: Balanced search trees



Problem 3: Hash tables

3-1) linear

0	“ant”
1	“flea”
2	“bat”
3	“cat”
4	“goat”
5	“dog”
6	“bird”
7	“bison”

3-2) quadratic

0	
1	
2	
3	“cat”
4	“goat”
5	“bird”
6	“bison”
7	“dog”

3-3) double hashing

0	“ant”
1	“bat”
2	“flea”
3	“cat”
4	“goat”
5	“bison”
6	“bird”
7	“dog”

3-4) probe sequence: 3, 0, 5, 2, 4

3-5) table after the insertion:

0	bobcat
1	
2	“eel”
3	fly
4	
5	koala
6	
7	penguin

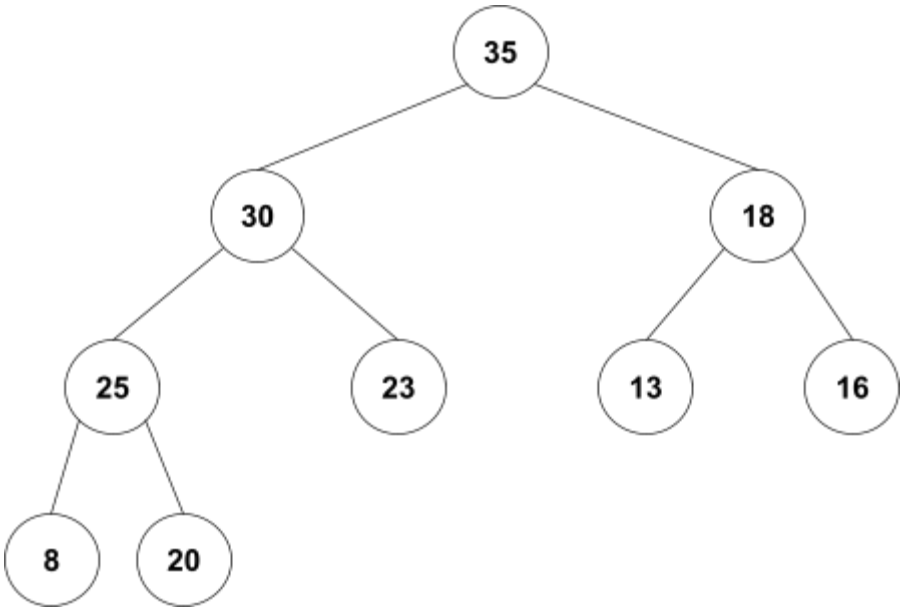
Problem 4: Complete trees and arrays

4-1) left child: 47 because 2*23 + 1 = 47
right child: 48 because 2*23 + 2 = 48
parent: 11 because (23 - 1) / 2 = 11

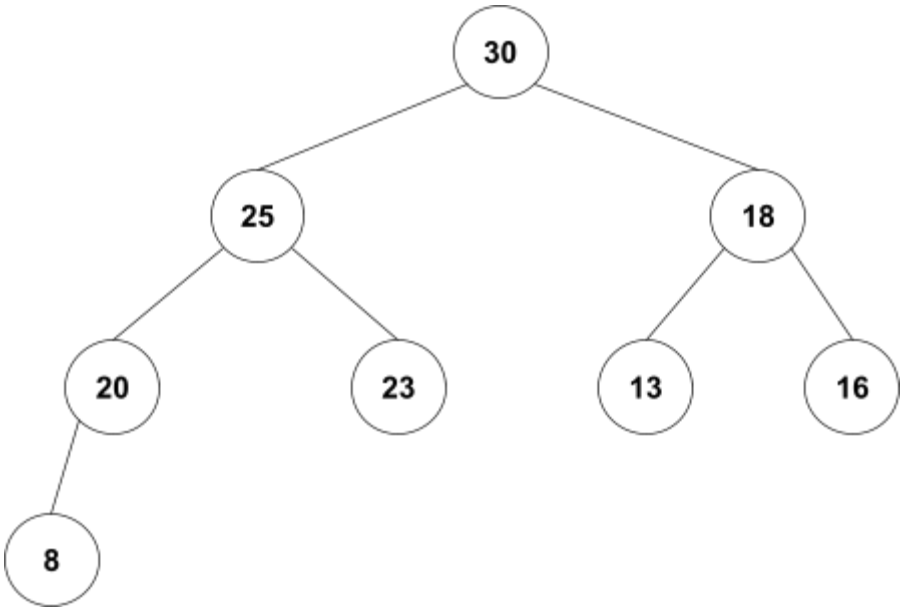
4-2) To find the height of a complete tree, you have to use $\log_2(n)$, and if there are 2023 nodes, there would be a height of 10 because the tree is balanced.

4-3) The rightmost leaf node is always the right child of its parent because, in a complete tree, every level is filled from left to right, so the rightmost would always be the right child of its parent node.

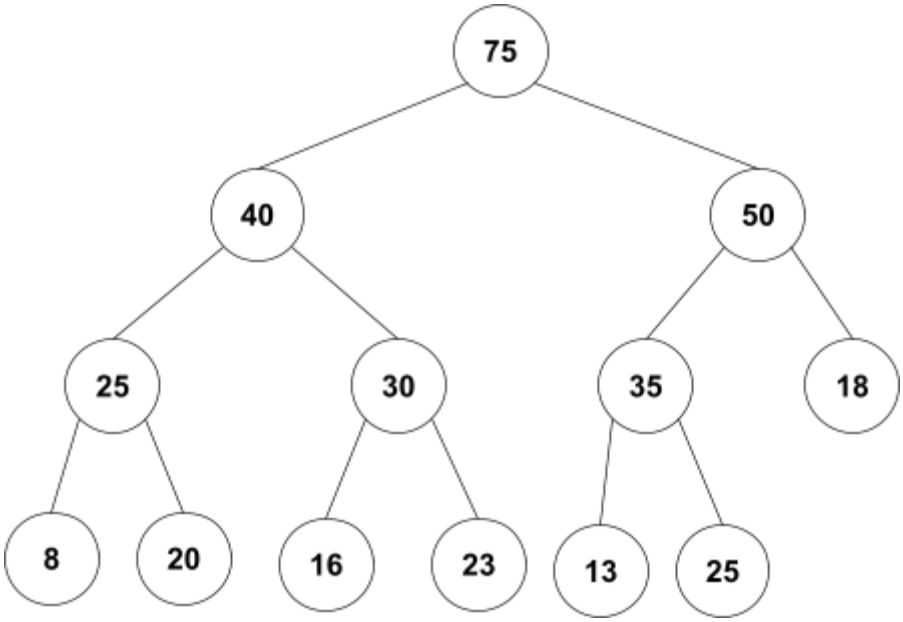
Problem 5: Heaps
5-1)
after one removal



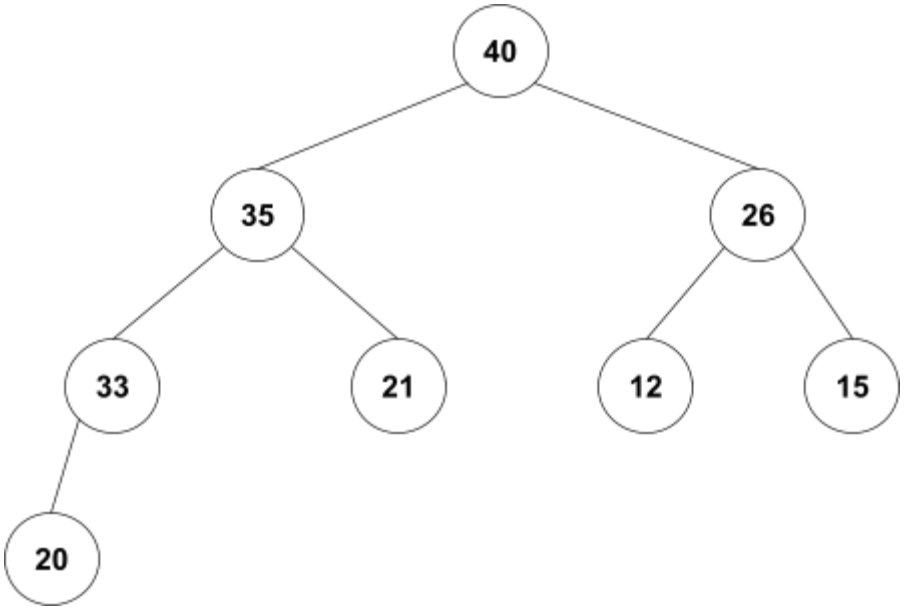
after a second removal
(copy your revised diagram from part 1 here, and edit it to show the result of the second removal)



5-2)



Problem 6: Heaps and heapsort
6-1)



6-2) {40, 35, 26, 33, 21, 12, 15, 20}

6-3) after first: {35, 33, 26, 20, 21, 12, 15, 40}
second: {33, 21, 26, 20, 15, 12, 35, 40}