Module 3 Creativity Exercises

3.6.17) describe and pseudocode findAllElements(k)

Assuming the tree allows duplicate entries and those dupes would be stored to the right if its predecessor has the same value. Ex:

```
5
 \
  5
```

We can approach this the same way as a typical BST traversal but just modifying it to account for the fact the dupe values can exist and in that case to traverse right if k is found or continue your typical root.left and root.right traversals until an leaf node is encountered.

Ill assume we have the head of the tree given as a root variable T and will access the value in the same manner the chapters showed using key(node).

```
#code start
findAllElements(k):
    root ← T
    //helper function for traversal
    findElements(root, k, ans):
        if root is external node:
            return ans
        if k == key(root):
            ans.append(root)
            # need to continue since dupes and unbalanced
            return findElements(root.rightChild(), k, ans)
        elif k > key(root):
            return findElements(root.rightChild(), k, ans)
        else:
            return findElements(root.leftChild(), k, ans)

    ans ← findElements(root, k, [])
    return ans
```

Time complexity: O(h + s)

Since it is not stated that the BST is balanced and we can have dupes its possible to have a tree composed of the same value down the the leaf node and in that case we would need to traverse all nodes of the tree making this O(h + s) assuming s is just accessing the object value.

4.7.25)

Since duplicates are not allowed in a balanced search tree by definition (assuming binary in this case) our approach is very similar to the previous question except we do not have to continue traversal after k is found or we reach an external node.

Once again assumption is that we have access to a tree T. Will assign to root just for convenience and better naming conventions

```
findAllElements(k):
    root ← T
    // defining a helper function that takes a node as a parameter
    findElement(root, k):
        if root is external node:
            return null
        if k == key(root):
            return root
        elif k > key(root):
            return findElements(root.rightChild(), k)
        else:
            return findElements(root.leftChild(), k)

    return findElement(root, k)
```

Time Complexity: O(logn + s)

A balanced tree will always be dividing the search candidates in half at each traversal depending on whether the seach value is greater than or less than the node. So at most you will always traverse logn of the tree + s assuming s is just the time to access the value.