PROBLEM 1:  *(Frutti Tutti, List2Tree (15 points))*

The method below returns a roughly balanced binary tree containing the same elements as in a doubly-linked list so that an inorder traversal of the binary tree visits the values in the same order as they’re stored from first to last in the linked list. In other words, if the linked list is in sorted/lexicographical order the binary tree will be a search tree.

```java
public TreeNode convert(DListNode list){
    if (list == null) return null;
    if (list.next == null){
        return new TreeNode(list.info, null, null);
    }
    DListNode first = list;
    DListNode middle = list;

    while (list != null && list.next != null){
        list = list.next.next;
        middle = middle.next;
    }
    DListNode second = middle.next;
    middle.prev.next = null;
    return new TreeNode(middle.info, convert(first), convert(second));
}
```

**Part A (4 points)**

What is the big-Oh runtime complexity of `convert` for an N-node list. Justify your answer.

**Part B (2 points)**

Briefly, what is the purpose of the line below in the method shown above.

```java
middle.prev.next = null;
```

**Part C (3 points)**

The code above de-links nodes in the list and thus effectively “destroys” the list being converted to a tree. Explain how to add a few lines to reconstruct the list so that the method works, but the list isn’t de-linked/destroyed.

(this question continued)
Part D (3 points)
Explain why simply inserting nodes from a *sorted* linked list into a binary search tree using the code below results in $O(n^2)$ runtime for an n-node list.

```java
public TreeNode convert(DListNode list) {
    TreeNode root = null;
    while (list != null) {
        root = insert(root, list.info);
        list = list.next;
    }
    return root;
}
private TreeNode insert(TreeNode root, String info) {
    if (root == null) return new TreeNode(info, null, null);
    if (info.compareTo(root.info) <= 0) {
        root.left = insert(root.left, info);
    } else {
        root.right = insert(root.right, info);
    }
    return root;
}
```

Part E (3 points)
Explain both why changing the insert code above in **Part D** so that an AVL tree is created and added to (the insert code will change, it is not shown) will result in changing the complexity of the convert method and what the new complexity will be.
PROBLEM 2:  *(Finding Yourself (24 points))*

Use the *search* tree below in answering questions for this problem.

Part A (3 points)
What is the postorder traversal of the subtree rooted at the *narwhal* node. List the post-order values in order, one per line. You should have six lines.

Part B (3 points)
Consider the code below for method *size*. What is returned by the call *size(root)* where *root* references the giraffe node in the diagram above?

Part C (3 points)
What is the running time of the method *size* below for an N-node tree. Use big-Oh and justify your answer.

```java
private int size(TreeNode root){
    if (root == null) return 0;
    return 1 + size(root.left) + size(root.right);
}
```

(this problem continued)
Part D (3 points)
Suppose the last value added to the tree above is jaguar. Before the addition of this value the tree is an AVL/height-balanced tree. After the addition the tree is no longer height-balanced. When the tree is rebalanced, lion will be the right child of giraffe. What value will be in the node that is the right child of lion after the tree is re-balanced to be an AVL tree after the addition of jaguar?

Part E (6 points)
The $k^{\text{th}}$ largest value in a collection of items is the smallest value that is larger than $k$ elements. The $0^{\text{th}}$ largest is the smallest, it’s larger than no other element. In a sorted array $a$ the $k^{\text{th}}$ largest is simply $a[k]$. The method findKthArray below returns the $k^{\text{th}}$ largest value in a binary search tree.

For example, in the tree at the beginning of this problem fox is the 3rd largest element and monkey is the 8th largest.

Briefly explain both why the method works correctly and what its running time is to find the $k^{\text{th}}$ largest from a binary tree of $N$ elements that is roughly balanced. Use big-Oh, justify your answer which should be in terms of $N$ and $k$ (or just one of them).

```java
private void fill(TreeNode root, ArrayList<String> list){
    if (root != null){
        fill(root.left, list);
        list.add(root.info);
        fill(root.right, list);
    }
}

public String findKthArray(TreeNode root, int k){
    ArrayList<String> list = new ArrayList<String>();
    fill(root, list);
    return list.get(k);
}
```

(this problem continued)
Part F (6 points)

A student proposes the method below to find the $k^{th}$ largest element in a binary tree of $N$ elements. It works correctly.

Briefly explain both why the method works correctly and what its running time is to find the $k^{th}$ largest from a binary tree of $N$ elements that is roughly balanced. Use big-Oh, justify your answer which should be in terms of $N$ and $k$ (or just one of them).

```java
public String findKth(TreeNode root, int k){
    if (root == null) return null;
    
    int leftCount = size(root.left);
    
    if (leftCount == k) {
        return root.info;
    }
    else if (k < leftCount){
        return findKth(root.left,k);
    }
    else {
        return findKth(root.right,k-leftCount - 1);
    }
}
```