Practice Exam Problems: Binary Trees and Binary Search Trees (BSTs)

For the following problems, use this node struct:

```
typedef struct node
{
      int data;
      struct node *left, *right;
} node;
```

1. Warm-up exercise: Insert the following values into a binary search tree (BST), one by one. Show the resulting BST:

   `6  10  17  11  3  5  2  21  13  12  1`

   Show the output from in-order, pre-order, and post-order traversals of the resulting tree.

2. Write a recursive function that compares two given binary trees. It returns 1 if the two trees are different, and it returns 0 otherwise. The function signature is:

   ```
   int treeDiff(node *a, node *b);
   ```

3. Write a recursive function that counts how many nodes in a tree have a single child. The function signature is:

   ```
   int countOneChild(node *root);
   ```

4. Write a recursive function that returns a pointer to the node containing the largest element in a BST. The function signature is:

   ```
   node *largest(node *root);
   ```

5. Write a recursive function that counts how many nodes are in a binary tree. The function signature is:

   ```
   int countNodes(node *root);
   ```

6. Write a recursive function that counts how many leaf nodes are in a binary tree. The function signature is:

   ```
   int countLeafNodes(node *root);
   ```

7. Write a recursive function that counts how many nodes in a binary tree have values greater than a given number, *key*. The function signature is:

   ```
   int countGreater(node *root, int key);
   ```

8. Write a recursive function that counts how many nodes in a BST have values greater than a given number, *key*. The function signature is:

   ```
   int BST_countGreater(node *root, int key);
   ```

9. Give the best-, worst-, and average-case Big-Oh runtimes for the functions above.

Practice Exam Problems: Linked Lists, Stacks, and Queues

1. Write an iterative (non-recursive) *insertNode()* function that inserts a node at the end of a linked list and a *frontInsert()* function that inserts a node at the front of a linked list. Modify the function(s) so that they maintain a tail pointer. The function signatures are:

```
node *insertNode(node *head, int data);

node *frontInsert(node *head, int data);
```

2. Re-write the *insertNode()* function so that it has no return value. Instead, pass a pointer to the head pointer. (For further explanation, see Practice Exam Problem #13 from the Linked Lists (Part 1 of 2) notes from Webcourses.) The function signature is:

```
void insertNode(node **head, int data);
```

3. Re-write *insertNode()* as a recursive function.

4. Write both recursive and iterative versions of a *printList()* function.

5. Write a recursive *printList()* function that prints a list in reverse order.

6. Write both recursive and iterative version of a *destroyList()* function that frees all nodes in a linked list.

7. Write a function that deletes the *N*th element from a linked list. If the linked list doesn't even have *N* nodes, don't delete any of them. Try writing this recursively and iteratively. The function signature is:

```
node *deleteNth(node *head, int n);
```

8. Write a function that deletes every other element in a linked list.

9. Write a function that deletes all nodes containing even integers from a linked list.

10. Implement *insertNode()*, *frontInsert()*, *deleteLast()*, *deleteFirst()*, and *deleteNth()* functions for doubly linked lists. Repeat these exercises with doubly linked lists in which you maintain a tail pointer. How does the tail pointer affect Big-Oh runtime for these functions? Are any of these functions more efficient for doubly linked lists with tail pointers than they are for singly linked lists with tail pointers?

11. Implement *push()* and *pop()* operations for stacks using linked lists. Do the same for *enqueue()* and *dequeue()*. The function prototypes are:

```
void push(Stack *s, int data);
int pop(Stack *s);
```

```
void enqueue(Queue *q, int data);
int dequeue(Queue *q);
```

12. In the previous problem, what would the Stack and Queue structs look like if we want each of those functions to be O(1)?

13. Code up a version of MergeSort that works on linked lists instead of arrays. (Aside: If we call *MergeSort()* on an array of size *n*, how many recursive calls will be made?)

14. Write a function that uses a stack (of characters) to reverse the characters in a string. Re-write the function using a queue instead of a stack.