1. **Assume LinkedList names = new LinkedList();  
   Draw the names list resulting from the following code snippet:  
   names.addFirst("John");  
   names.addLast("Paul");  
   names.addFirst("George");  
   names.addLast("Ringo");**

The names list resulting from the code snippet will be as follows:

[George, John, Paul, Ringo]

1. **Assuming the final names list from question 1, draw the names list resulting from the following code snippet:  
   String name = (String)names.removeLast();  
   names.addFirst(name);**

The names list resulting from the code snippet above will be as follows:

[Ringo, George, John, Paul]

1. **We improved the run-time efficiency of our addLast method by adding a tail Node reference to our original LinkedList class.  Is the run-time efficiency of the removeLast method improved?  Why (not)?  If not, how can we improve the efficiency of the removeLast method?**

No, the run-time efficiency of the removeLast method will not be improved by adding a tail node reference. This is because in order to remove the last node, we need a reference to the second-to-last node, rather than the tail node. However, because we do not currently have this reference, the removeLast method must still traverse the list to the next-to-last node and then remove the tail node. We can improve the efficiency of the removeLast method by adding a reference to the next-to-last node so that the tail node can be quickly deleted.

1. **Suppose you are stranded on a desert island on which LIFO stacks are plentiful, but no other data structures (arrays, lists, sets, maps, etc.) exist.  You need a FIFO queue.  How can you implement a queue using two stacks?  
   Describe each of the queue operations (isEmpty, enqueue, dequeue, and peek) in terms of the stack operations (isEmpty, push, pop, and peek).  Your answer can be in the form of algorithms or Java methods.  
   Also, assuming all stack operations are O(1), what is the run-time efficiency (i.e., O(...)) of each queue operation?**

You can implement a queue using two stacks. To do so, we will push the data onto the top of the first stack for the enqueue method if the first stack is empty. If the first stack is not empty, then pop everything in it and push it to the second stack. Then, push the new data to the first stack and pop and push everything from the second stack back onto the first stack. For dequeue, we will throw an error if both the first stack is empty. Otherwise, we will pop the element from the first stack and return it. For the queue peek method, we will again throw an error if the first stack is empty. Otherwise, we will use the stack peek method to return the element at the top of the first stack (which is the top of the queue) without having to remove it. Finally, for the queue isEmpty method, simply check if the size of the first stack is 0. If so, return true, otherwise return false. In this situation, the run-time efficiency of enqueue would be O(n). Additionally, dequeue, peek, and isEmpty would be O(1).

1. **Suppose you are stranded on a desert island on which queues are plentiful, but no other data structures (arrays, lists, sets, maps, etc.) exist.  You need a stack.  How can you implement a stack using two queues?  
   Describe each of the stack operations (isEmpty, push, pop, and peek) in terms of the queue operations (isEmpty, enqueue, dequeue, and peek).  Your answer can be in the form of algorithms or Java methods.  
   Also, assuming all queue operations are O(1), what is the run-time efficiency (i.e., O(...)) of each stack operation?**In order to make a stack using two queues, we need to make sure that the newest element is at the top of the first queue. We will use the second queue to keep the newest element at the top of the first queue. For the push method (adds element to end of queue), we will simply enqueue the element to the first queue if the first queue is empty. If the first queue is not empty, then enqueue the element to the second queue and then dequeue and enqueue everything in the first queue to the second queue. Finally, dequeue and enqueue each element back to the first queue. For the pop method, we will throw an exception if the first queue is empty. If it isn’t empty, we will use the queue’s dequeue operation to remove the top element from the first queue. For peek, we will again throw an error of the first queue is empty. If not, we will use the queue peek operation to return the top element without removing it. Finally, for isEmpty, simply check if the size of the first queue is 0. If so, return true, otherwise return false. In this situation, the run-time efficiency of push would be O(n). The run-time efficiency of pop, peek, and isEmpty would all still be O(1).
2. **Describe a recursive algorithm for finding the maximum number of siblings in a tree.**

One way to go about finding the maximum number of siblings in a given tree would be to use recursion to traverse trough each node in the tree. So, we can start by using recursion to move through all of the nodes on the left, for example. For each node that is not equal to null, add one to a counter variable. However, once a node is equal to null, do not add anything to the counter variable and terminate the recursive calls. Once the nodes on the left have been traversed through, go through the same recursive process for the right nodes, adding one for each node that is not null and ending the process when you come across a null node. Finally, return the value of the counter variable, as this will be the maximum number of siblings in the given tree.

This countSiblings method could look something like the following:

int counter = 0;

counter = counter + 1 + countSiblings(leftTree);

counter = counter + 1 + countSiblings(rightTree);

return counter;

1. **Describe a recursive algorithm for finding the total path length of a tree. The total path length is the sum of the lengths of all paths from the root to the leaves. (The length of a path is the number of nodes on the path.)**

We can recursively find the total path length of a tree using the number of nodes in each level of the tree. We can calculate this by multiplying the number of nodes on a given level of the tree by the level of the tree, then adding up all of the levels. So, we want to use recursion to traverse through each level of the tree. While in each level, count the total number of nodes, multiply it by the level number, and add this number to a sum variable. Then, recursively call the same method for the new level using the updated sum. Once you go all the way through the entire tree (i.e., the nodes on the next level are null), return the current value of the sum variable and terminate the recursive calls. This sum variable will leave you with the total path length of the given tree.

1. **What is the difference between a binary tree and a binary search tree?**

A binary tree is a tree (data type that consists of nodes and arcs) whose elements have a maximum of two children. However, a binary search tree is an organized binary tree data structure where the left subtree only contains lesser elements, and the right subtree only contains greater elements. Additionally, each subtree is also a binary search tree, and there are no duplicate nodes in binary search trees.

1. **Describe an algorithm for deciding whether two binary trees have the same shape.**

To do this, we need to traverse through each binary tree and check if nodes exist in the same spot for each. In order to complete this process, we can use the following algorithm: first, if both trees are empty, return true. Else if both trees are not empty, use recursion to check all of the subtrees on the left for both trees. Then, use recursion to check all of the subtrees on the right for both trees. If true is returned in the last two steps (i.e., both trees have nodes in all of the same spots on both the left and the right) then return true. Finally, if one tree is empty and one tree isn’t empty, return false as the trees will not have the same shape. So, the recursive method should accept two parameters – in this case each parameter would be a reference to one of the trees. In conclusion, this recursive method will go through each tree at the same time and return false if one tree has a child when the other does not, and true if that is not the case.

1. **Assuming case-insensitive, lexicographic order, insert the following eleven words into an initially-empty binary search tree:  
   Mary had a little lamb. Its fleece was white as snow.  
   Draw the resulting tree.**

Lexicographic order = dictionary order – ex: A comes before B, M comes after H, etc.

Case-insensitive, so Its is not before everything else. **Diagram

Description automatically generated**

1. **What is the result of printing the tree from question 10 using preorder, inorder, and postorder traversal?**

Preorder: Mary had a fleece as little lamb its was snow white

Inorder: a as fleece had its lamb little Mary snow was white

Postorder: as fleece a its lamb little had snow white was Mary