Technique: Append Function

Level: Easy

The append function is especially useful in Linked List problems. On the face, it is pretty simple. But once you know it by heart, you can apply it to a ton of problems quickly.

To Implement: A function that adds a node to a linked list.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few lines quickly to plan out your solution. Your actual solution should be in a real language and use good syntax.)

append(Node n)
  if head is null (list empty)
     n is new head
     n is new tail
  else
     add n to tail
     n is new tail
```

Time Complexity: O(1) since we know the tail

Space Complexity: O(1)

```
public static class LinkedList {
    Node head;
    Node tail;

    public LinkedList() {
        head = null;
        tail = null;
    }

    public void append(Node toAdd) {
        if (head == null) {
            head = toAdd;
        } else {
            tail.setNext(toAdd);
        }
        tail = toAdd;
    }
}
```

Level: Easy

You are given a Linked List with nodes that have values 0, 1 or 2. Sort the linked list.

Questions to Clarify:
Q. Do I need to sort in place?
A. No, I want you to rearrange the existing nodes.

Q. How do you want the output?
A. Return the Head and Tail as a pair.

Solution:
We initialize 3 new linked lists for each value (0,1,2).

Then, we go through the list and use the append function to place each node in the appropriate list.

After that, we simply join the 3 lists (0 -> 1 -> 2). This is the output.

Follow Up Question: How would you generalize this problem for n value types (0,1,2,..n).
In our solution, we explicitly declare 3 variables and 3 if statements every time.
This is ok because we know there are only 3 values (0,1,2). For n values, we will create an array of size *n*, and append accordingly.

Pseudocode:

```
init 3 lists
loop through input list
     append node to appropriate list
combine 3 lists using append function
```

Test Cases:
Edge Cases: empty list, null elements, invalid value
Base Cases: one node, 2 nodes
Regular Cases: all values present, one value only

Time Complexity: O(n)

Space Complexity: O(1).
This is we only allocate 3 new Linked Lists. We don't create any new nodes in these lists. We merely re-use nodes from the input list.

```
public static LinkedList sortList(LinkedList input) {
    if (input == null)
        return new LinkedList(); // empty list
```

```
    LinkedList list0 = new LinkedList();
    LinkedList list1 = new LinkedList();
    LinkedList list2 = new LinkedList();
    Node current = input.head;

    while(current != null) {
        switch (current.getData()) {
            case 0: list0.append(current); break;
            case 1: list1.append(current); break;
            case 2: list2.append(current); break;
            default: throw new IllegalArgumentException(
                    "Invalid value: " + current.getData());
        }
        current = current.getNext();
    }

    // set tails to null
    if (list0.tail != null)
        list0.tail.next = null;

    if (list1.tail != null)
        list1.tail.next = null;

    if (list2.tail != null
        list2.tail.next = null;

    // attach lists in sequence
    LinkedList result = new LinkedList();
    appendList(list0, result);
    appendList(list1, result);
    appendList(list2, result);

    return result;
}

void appendList(LinkedList toAppend, LinkedList original) {
    if (toAppend == null || toAppend.head == null)
        return;
    original.append(toAppend.head);
    original.tail = toAppend.tail;
}
```