Linked List Recursion Practices

Warm-up: Measuring the length of a linked list
Given a linked list, return the length of the linked list.

`int ll_len(Node* head);`

Remove Neighboring Duplicate Elements
Given a linked list, remove elements that appear immediately after an element equal to it.

`void ll_unique(Node* head);`

Example: 1 -> 2 -> 3 becomes 1 -> 2 -> 3; 1 -> 1 -> 2 -> 4 -> 4 -> 4 becomes 1 -> 2 -> 4 -> 5 -> 4

Partial Sum
Given a linked list, change its elements so that each element becomes the sum of itself and all elements that come before it in the original list.

`void ll_partial_sum(Node* head);`

Example: 1 -> 2 -> 3 becomes 1 -> 3 -> 6 (1 = 1, 3 = 1 + 2, 6 = 1 + 2 + 3)

Rotate
Given a linked list and an index $n$ such that the first $n$ are swapped to the end of the linked list. Return the new head.

`Node* ll_rotate(Node* head, int n);`

Example:
- `ll_rotate([1, 2, 3, 4, 5], 2)` gives `[3, 4, 5, 1, 2]`
- `ll_rotate([1, 2, 3, 4, 5], 3)` gives `[4, 5, 1, 2, 3]`
- `ll_rotate([1, 2, 3, 4, 5], 0)` gives `[1, 2, 3, 4, 5]`
- `ll_rotate([1, 2, 3, 4, 5], 5)` gives `[1, 2, 3, 4, 5]`
Lexicographical Compare

Given two linked lists, compare them lexicographically. This is similar to how you compare two strings (and the order they would appear in a dictionary), for example: “apple < application”, because “e < i”, the first letter by which they differ; “com < command” because they share the first 3 letters but “com” is shorter.

```c
int* ll_compare(Node* lhs, Node* rhs);
```

- You should return -1 if lhs is less than rhs
- You should return 1 if rhs is less than lhs
- You should return 0 if lhs and rhs are the same

Examples:
- `ll_compare( [1, 2, 3, 4], [1, 2, 3, 4] )` returns 0
- `ll_compare( [1, 2, 3, 4], [ ] )` returns 1.
- `ll_compare( [1, 2, 3, 4], [ 0, 1, 2, 3, 4 ] )` returns 1. (Because 0 < 1)
- `ll_compare( [1, 2, 3, 4], [ 1, 2, 3, 5 ] )` returns -1.

And good luck on the midterm exam :)}