CS 32 Worksheet Week 3

Concepts: Linked lists

1. (10 min) Write a function **`cmpr`** that takes in a linked list and an array and returns the largest index up to which the two are identical. The function should return -1 if no values match starting from the beginning.

Assume the following declaration of `Node`:

```
struct Node {
   int data;
   Node* next;
};

// Function declaration: cmpr(Node* head, int* arr, int arr_size);
```

Examples:

```
// head -> 1 -> 2 -> 3 -> 5 -> 6
int a[6] = {1, 2, 3, 4, 5, 6};
cout << cmpr(head, a, 6); // Should print 2

int b[7] = {1, 2, 3, 5, 6, 7, 5};
cout << cmpr(head, b, 7); // Should print 4

int c[3] = {5, 1, 2};
cout << cmpr(head, c, 3); // Should print -1

int d[3] = {1, 2, 3};
cout << cmpr(head, d, 3); // Should print 2
```

2. (10 min) Given two linked lists where every node represents a character in a word. Write a function compare() that works similarly to strcmp(), i.e., it returns 0 if both strings are the same, a positive integer if the first linked list is lexicographically[1] greater, and a negative integer if the second string is lexicographically greater.

The header of your function is given as:

```
int compare(Node* list1, Node* list2)
```

Assume the following declaration of `Node`:

```
struct Node {
      char c;
      Node* next;
};
```

Example:

```
If list1 = a -> n -> t
   list2 = a -> r -> k
then compare(list1, list2) < 0

If list1 = b -> e -> a -> n -> s
   list2 = b -> e -> a -> n
then compare(list1, list2) > 0
```

[1] Sorted alphabetically, like in a dictionary.

3. (10 min) The following is a class definition for a linked list, called `LL`, and for a node, called `Node`. Class `LL` contains a single member variable - a pointer to the head of a singly linked list. Struct `Node` contains an integer value, and a node pointer, `next`, that points to the next node in the linked list. Your task is to implement a copy constructor for `LL`. The copy constructor should create a new linked list with the same number of nodes and same values.

Note: This is not a complete class implementation, but it doesn't affect the problem.

```
class LL {
  public:
    LL() { head = nullptr; }

  private:
    struct Node {
        int val;
        Node* next;
    };
    Node* head;
};
```

4. (15 min) Using the same class LL from the last problem, write a function `findNthFromLast` that returns the value of the Node that is n Nodes before the last Node in the linked list. Consider the last Node to be 0 Nodes before the last Node, the second-to-last Node to be 1 Node before the last Node, etc.

```
int LL::findNthFromLast(int n);

// findNthFromLast(2) should return 4 when given the following linked list:

// head -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

If the nth from the last Node does not exist, *findNthFromLast* should return -1. You may assume all values that are actually stored in the list are nonnegative.

5. (15 min) Suppose you have a struct **Node** and a class **LinkedList** defined as follows:

```
struct Node {
   int val;
   Node* next;
};

class LinkedList {
 public:
   void rotateLeft(int n); // rotates head left by n
   // Other working functions such as insert and printItems
 private:
   Node* head;
};
```

Write a function *rotateLeft* function such that it rotates the linked list to the left, *n* times. Rotating a list left consists of shifting elements left, such that elements at the front of the list loop around to the back of the list. The new start of the list should be stored in *head*.

Example:

Suppose you have a **LinkedList** object `numList`, and printing out the values of *numList* gives the following output, with the head pointing to the node with 10 as its value:
10 -> 1 -> 5 -> 2 -> 1 -> 73

Calling `numList.rotateLeft(3)` would alter `numList`, so that printing out its values gives the following, new output, with the head pointing to the node with 2 as its value:
2 -> 1 -> 73 -> 10 -> 1 -> 5

The `rotateLeft` function should accept only integers greater than or equal to 0. If the input does not fit this requirement, it may handle the case in whatever reasonable way you desire.

6. (15 min) Write a function that takes in the head of a singly linked list, and returns the head of the linked list such that the linked list is reversed. The function modifies the arrangement of the nodes; do not create any new nodes.

Example:

Original: LL = 1 → 2 → 3 → 4 → 5 Reversed: LL = 5 → 4 → 3 → 2 → 1

We can assume the Node of the linked list is implemented as follows:

```
// Linked list node
struct Node
{
    int data;
    Node* next;
};

Node* reverse(Node* head) {
      // Fill in this function
}
```

7. (20 min) Write a function `combine` that takes in two **sorted** linked lists and returns a pointer to the start of the resulting combined **sorted** linked list. You may write a helper function to call in your function `combine`.

Assume the following declaration of `Node`:

```
struct Node {
      int val;
      Node* next;
};
```

The header of your function is given as:

```
Node* combine(Node* h, Node* h2)
```

Example:

h: `head -> 1 -> 3 -> 6 -> 9`
h2: `head2 -> 7 -> 8 -> 10`

```
Node* res = combine(head, head2);
```

should result in

```
res -> 1 -> 3 -> 6 -> 7 -> 8 -> 9 -> 10
```