CS 32 Solutions Week 3

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler.

Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please feel free to go to any of the LA office hours.

Concepts

Linked lists

1) 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);`

```
// 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
```

Time: 5-10 minutes

```
int cmpr(Node *head, int *arr, int arr_size) {
      int i = 0;
      Node *curr = head;
      while (i < arr_size && curr != nullptr && curr->data ==
      arr[i]) {
            i++;
            curr = curr->next;
      }
      return i-1;
}
```

2) 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 greater, and a negative integer if the second string is lexicographically greater.

Lexicographically: sorted alphabetically, like in a dictionary.

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
```

Time: 10 minutes

```
int compare(Node *list1, Node *list2)
{
    // Traverse both lists. Stop when either end of a linked
    // list is reached or current characters don't match
    while (list1 != nullptr && list2 != nullptr &&
                                  list1->c == list2->c)
    {
        list1 = list1->next;
        list2 = list2->next;
    }

    if (list1 == nullptr)  // list1 ran out
    {
        if (list2 == nullptr)  // both ran out at the same time
            return 0;
        else  // list2 continues after list1 ran out
            return -1;
    }
    else
    {
        if (list2 == nullptr)  // list1 continues after list2
ran out
            return 1;
        else  // there's a mismatching character
            return (list1->c < list2->c) ? -1 : 1 ;
    }
}
```

3) 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.

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

     LL(const LL& other) {
       if (other.head == nullptr)
         head = nullptr;
       else {
         head = new Node;
         head->val = other.head->val;
         head->next = nullptr;
```

```
            Node* thisCurrent = head;
            Node* otherCurrent = other.head;
            while (otherCurrent->next != nullptr) {
              thisCurrent->next = new Node;
              thisCurrent->next->val = otherCurrent->next->val;
              thisCurrent->next->next = nullptr;

              thisCurrent = thisCurrent->next;
              otherCurrent = otherCurrent->next;
            }
          }
        }

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

Time: 5-10 minutes

4) 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.

Time: 10-15 minutes

```
int LL::findNthFromLast(int n) {
  Node* p = head;
      // advance p forward by n, checking there are at least n
      // elements
    for (int i = 0; i < n; i++) {
      if (p == nullptr) {
        return -1;
      }
      p = p->next;
    }
    if (p == nullptr) {
      return -1;
    }

    Node* nthBeforeP = head;  // will lag n steps behind p
    while (p->next != nullptr) {
      p = p->next;
      nthBeforeP = nthBeforeP->next;
    }
    return nthBeforeP->val;
  }
```

5) 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*.

Ex: 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.

Time: 15 minutes

```
void LinkedList::rotateLeft(int n) {
  if(head == nullptr)
    return;
  int size = 1;
  Node* oldTail = head;
  while (oldTail->next != nullptr) {
    size++;
    oldTail = oldTail->next;
  }

  if (n % size > 0) {
    int headPos = n % size;
    Node* newTail = head;
    for (int x = 0; x < headPos - 1; x++) {
      newTail = newTail->next;
    }
    Node* newHead = newTail->next;

    newTail->next = nullptr;
    oldTail->next = head;
    head = newHead;
  }
}
```

6) 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
}
```

Time: 15 minutes

```
// The idea here is to reverse each node one step at
// a time with a previous and current pointer
// At the end prev should point to the last element in the
// original linked list

Node* reverse(Node* head) {
     Node* prev = nullptr;
     Node* curr = head;
     while (curr != nullptr) {
          // point "next" to the node after curr
          Node* next = curr->next;
          // make curr's next pointer point
          //to the node before curr
          curr->next = prev;
          // making prev point to curr and curr point to "next"
          // this is just advancing the linked list pointers
          prev = curr;
          curr = next;
     }
     // prev points to the last-examined node of the list,
which
     //is the head of the reversed linked list
     return prev;
}
```

7) 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
```

Time: 15-20 minutes

```
Node* combine(Node* h, Node* h2) {
     // checking that the lists aren't empty
     if (h == nullptr) {
          return h2;
     }
     if (h2 == nullptr) {
          return h;
     }

     Node* newList;

     // determining which should be the resultant head
     if (h->val <= h2->val) {
          newList = h;
          h = h->next;
     }
     else {
          newList = h2;
          h2 = h2->next;
     }

     Node* newNext = newList;

     // iterate through both given linked lists
     while (h != nullptr && h2 != nullptr) {
       // the next node in the combined list is the currently
       //examined node in either list that has the lesser value
          if (h->val <= h2->val) {
                    newNext->next = h;
```

```
                h = h->next;
            }
            else {
                newNext->next = h2;
                h2 = h2->next;
            }
            newNext = newNext->next;
        }
        // if one list was longer than the other
        // append the rest of the list to the new list
        if (h != nullptr) {
            newNext->next = h;
        }
        else if (h2 != nullptr) {
            newNext->next = h2;
        }

        // return the head of the combined list
        return newList;
    }
```