Computer Science II — Homework 5 — Linked-List Functions

This assignment is due Thursday, March 1 at 11:59:59pm and is worth 70 points toward your homework grade. It focuses on implementation of linked-list operations, building on material covered in Lecture 10.

Starting with a simple, templated `Node<T>` class having all public member variables, you will write several functions that accomplish linked list operations. These functions will NOT be member functions of any class. (This step will be taken in Lecture 11 and Lab 6.) The form of the class is

```
template <class T>
class Node {
public:
  T value;
  Node* next;
};
```

This will be provided to you in a header file called `llist.h`. You may not change this declaration, except that you may add constructors if you wish. This header file also contains a templated function to print the contents of a list. You will need to add your function implementations to this header file. We will also provide you with a bare beginning of a main function in a file called `hw5_main.cpp` to test each of these functions. Your job will be to implement and **debug thoroughly** each of the functions, making sure that they can handle all special cases.

The following is an explanation of the functions, which must all be templated (see Lecture 10 notes). More details and usage for these functions may be seen in the provided main program. The functions are

- `add_to_back`. This is like the `push_back` function for the list class, except that the pointer to the head of the list is provided as an argument.
- `add_to_front`. This is like the `push_front` function for the list class, except that the pointer to the head of the list is provided as an argument.
- `remove_each`. This function should remove each node in the list that contains the specified value.
- `reverse`. This function should reverse the contents of the list **without creating any new nodes**. Hint: this function is perhaps easier to write recursively.
- `destroy_list`. This function should delete all nodes of the list and the resulting "list" should be empty.
- `ordered_insert`. Insert the new value so that the non-decreasing order of the list is maintained. We will not try to trick you by creating a non-ordered list and then calling the `ordered_insert` function.
- `ordered_merge`. Given pointers to the starts of two ordered lists, create a third list that contains all values from the original two lists in non-decreasing order. Return a

pointer to the first node in this list. If the same value appears in both lists, it should be repeated in the resulting list. The most credit will be given to a function that runs in time that is linear in the sum of the lengths of the two lists and **does not use** `ordered_insert`. On the other hand, we suggest that you start with the simplest solution you can think of, which may involve use of `ordered_insert`.

When you submit your homework, you may submit both `hw5_main.cpp` and `llist.h`, but we will not use your `hw5_main.cpp` in testing. When you submit we will test against the `hw5_main.cpp` that we provided. When we grade, we will test your functions against a more extended version of `hw5_main.cpp` that includes many more tests. We also require you to submit a brief `readme.txt` file. The template for this file is available on the course web page.

Final notes:

- We strongly urge that you implement and test these functions one at a time. You will be better off submitting a complete solution to some of the problems than submitting partial solutions to all of the problems.
- If you only debug your program on the `hw5_main.cpp` code provided, it is likely that you will miss many potential issues and therefore will not earn full credit. Write your own test cases.