CS1XC3 - Computer Science Practice and Experience: Development Basics

Assignment #6 – Linked Lists

Weight

7% of total course grade

Due date

Sunday March 14th at 11:59pm

Primary Learning Objectives

- Write a C program that manipulates linked lists.

Requirements

As starter code for this assignment, use the C code contained in as6_starter_code.zip file found on Avenue to Learn accompanying this assignment.

This starter code contains the same linked list type used in lectures and labs, and the same set of functions for working with linked lists that we created together in lecture and lab. The starter code also contains the following function declarations:

```
// Functions to implement (merge_sorted_lists is a bonus, it is not required)

void add_lists(Node *list1, Node *list2);

Node *duplicate_list(Node *head);

Node *efficient_delete_match(Node *head, int delete_value, int *num_deleted);

Node *merge_sorted_lists(Node *list1, Node *list2);
```

This assignment requires you to implement the add_lists, duplicate_list and efficient_delete_match functions; requirements are described below. The merge_sorted_lists function is an optional bonus that you can choose to complete or not, it is not required for full marks. Test code is provided in the main function to help you determine when your function is working correctly, and to clarify any confusion about the requirements provided below.

The function **add_lists** should add the value of each node contained in the same position in the lists together and store the result as the new value of list1's node at that position. If list1 is shorter than list2 or vice versa, than for nodes past the shorter of the two lists no addition needs to occur (and list1 should remain the same for these nodes). Use recursion to create this function (though not required, the function body can be created with 3 lines of code).

The function **duplicate_list** should create a new list on the heap that is a duplicate of the input list provided as an argument. The duplicate list should have the same number of nodes as the input list, and each node at each position in the list should have the same value as the node in the input list at that position in the input list. The function should return a pointer to the head of this new list.

The function **efficient_delete_match** should delete any nodes from the list that have a value matching the argument **delete_value** and it should count the number of deletions that have occurred with **num_deleted** (a pass-by-reference parameter). The function should return a pointer to the (potentially different) head of the resulting list (which could even be NULL/empty). The function should result in the same resulting list, and same num_deleted value, as the **delete_all_matches** function that we created in-class.

However the **delete_all_matches** function was inefficient... it works by repeatedly calling **delete_first_matc**h until no matches are found. The delete_first_match function traverses the list until the first match is found, and so if we repeatedly call the function, we are traversing the list again and again. Your **efficient_delete_match** function should traverse the list **only once**. You can use multiple loops in your code if like, perhaps traversing a portion of the list, and then another portion of the list, but your code should not repeatedly traverse the same elements of the list. The **delete_duplicates** function may be helpful as inspiration for how to achieve this. As a result, the performance should be vastly improved compared to **delete_all_matches**, and test code is provided to demonstrate this.

You can use any of the functions we created in lecture and in lab to help you create the above functions.

Bonus – 10 marks

You can optionally create the **merge_sorted_lists** function too for up to 10 bonus marks. This function should accept two *sorted lists* as input (sorted from lowest to highest values). The function should return a pointer to a list that has been created by merging and connecting the nodes in the two lists to form a new list that is also sorted from lowest to highest. The new list should contain all of the nodes that are in both lists, what should be different is how the nodes point to each other in order to form one merged list.

The function **should not** create any nodes (i.e. create a new list on the heap entirely using the values from the two previous lists). No new nodes should be created on the heap. The pointers of the existing lists should be re-arranged to form one new merged list that is ordered from lowest to highest.

Test Code

The main function contains test code to test all 4 functions. You may want to "comment out" this code by surrounding it with /* … */ multiline comments so that you can work on your functions, as the code assumes that all of the functions have been implemented. You can use this code to help verify that your functions are working correctly.

Note that in the case of merge_sorted_list some random values are used as part of the test, so your results will not be identical. And in the case of the performance test of the delete functions, the time will be different (but if you're running the code on the pascal server, a similar very large gap in performance should be evident).

The expected test code results are provided after the mark breakdown, and as a plaintext file on Avenue to Learn accompanying the assignment.

Submission

Save your solution in a file named **list.c** and put it in a zip file named **a6.zip** and submit it to the Assignment #6 dropbox on Avenue to Learn.

Marking scheme

This assignment is marked out of 100 total. If your grade exceed 100 marks due to the bonus question, you will receive 100 marks total (though of course, answering the bonus question can help ensure you reach 100 marks total).

Component	**Description**	**Marks**
add_lists	Does it use recursion to add the two list values as described in the requirements?	35
duplicate_lists	Is a duplicate list created, is it accurate?	35
efficient_delete_match	Does it delete list nodes with a matching value? Does it traverse the list only once? Does it keep track of the number of deletions?	30
Total:		100

Test Code Results

```
[brownek@pascal ~]$ ./demo



add_list test
****************************************


List 1...
Node 0: 5
Node 1: 4
Node 2: 3
Node 3: 2
Node 4: 1


List 2...
Node 0: 4
Node 1: 5
Node 2: 6
Node 3: 7
Node 4: 8


List 1 after add...
Node 0: 9
Node 1: 9
Node 2: 9
Node 3: 9
Node 4: 9


List 3...
Node 0: 2
Node 1: 1
```

```
List 4...
Node 0: 7
Node 1: 6
Node 2: 5
Node 3: 4


List 3 after add...
Node 0: 9
Node 1: 7


List 5...
Node 0: 7
Node 1: 6
Node 2: 5
Node 3: 4


List 6...
Node 0: 2
Node 1: 1


List 5 after add...
Node 0: 9
Node 1: 7
Node 2: 5
Node 3: 4




duplicate_list test
****************************************


List 7...
Node 0: 9
Node 1: 8
Node 2: 7
Node 3: 6
Node 4: 5
Node 5: 4
Node 6: 3
Node 7: 2
Node 8: 1
Node 9: 0


List 7 duplicate...
Node 0: 9
Node 1: 8
Node 2: 7
Node 3: 6
Node 4: 5
Node 5: 4
Node 6: 3
Node 7: 2
Node 8: 1
Node 9: 0




efficient_delete_match test
****************************************


List 8...
Node 0: 4
Node 1: 4
Node 2: 7
Node 3: 4
Node 4: 4
Node 5: 5
Node 6: 4
Node 7: 3
Node 8: 4
```

```
List 8 after delete...
Node 0: 7
Node 1: 5
Node 2: 3
Number of elements deleted: 6



Performance test of delete functions
****************************************


delete_all_matches: 0.633520s
elements deleted: 5000


efficient_delete_match: 0.000347s
elements deleted: 5000



merge_sorted_list test
****************************************


List 11...
Node 0: 2
Node 1: 2
Node 2: 9
Node 3: 62
Node 4: 62
Node 5: 62
Node 6: 62
Node 7: 74
Node 8: 77
Node 9: 97
```

```
List 12...
Node 0: 1
Node 1: 3
Node 2: 4
Node 3: 19
Node 4: 24
Node 5: 37
Node 6: 58
Node 7: 69
Node 8: 90
Node 9: 92


Merged list...
Node 0: 1
Node 1: 2
Node 2: 2
Node 3: 3
Node 4: 4
Node 5: 9
Node 6: 19
Node 7: 24
Node 8: 37
Node 9: 58
Node 10: 62
Node 11: 62
Node 12: 62
Node 13: 62
Node 14: 69
Node 15: 74
Node 16: 77
Node 17: 90
Node 18: 92
Node 19: 97
```