CS5008 Lab 5: Linked Lists

The purpose of this lab is to give you experience with linked lists and writing recursive functions on linked lists. Recall that recursive functions are written such that recursive calls of the function lead toward base case(s). For each of the recursive functions in this lab, the base case of lists being NULL is done for you. You are to complete the recursive cases in the functions.

This lab has a total of 100 possible

Choose a new partner for this lab – I encourage pair programming.

Objectives

Upon completion of the laboratory exercise, you will be able to do the following:

- Identify the base case(s) and recursive case(s) for the recursive functions.
- Traverse linked lists.
- Create linked lists.
- Merge two linked lists.

Part 1:

Get the lab6 file. Download Lab 5 from class Github page to your repository on locate `link_list_lab.c` and open it with your editor

You can focus on the first 130 (or so) lines of the file. This has the same implementation of linked lists as discussed in lab class, so the code should be familiar. The four new function prototypes are listed above the main function.

Part 2: lengthRec – Fix the recursive case

1. Examine the `lengthRec` function definition. This function is supposed to return the number of nodes in the linked list. An iterative version of length is defined later in the file. This one, however, is supposed to be recursive. Update the "else" case so that it recursively calls `lengthRec` on `list->next` and returns this function call plus 1. (one line of code)

2. In the main function, update the code to update the contents of `lst1` so that it has 5 nodes where the values of the nodes are in ascending order.

3. In the main function, write the code to update the contents of `lst2` so that it has 8 nodes where the values of the nodes are in ascending order. Have some of these values be the same as values in lst1 and have some be different. Print the list.

4. Compile and run your code.

```
gcc -o link_list link_list_lab.c
link_list
```

If you want to use the debugger, include the –g flag when compiling.

Part 3: freeList – Fix the recursive case

1. Examine the `freeList` function. The function should be recursive. It should recursively free `list->next` and then free the list. (Two lines of code)

2. The function call to free lst1 and lst2 are already in main. Compile and run your code.

3. Why must `list->next` be free'ed before `list`?

____

<u>Checkpoint 1 [40 points]: Show your lab instructor/assistant your program running and show your function definitions. Show your answer to the question above.</u>

Part 4: copyList – Fix the recursive case

1. Examine the `copyList` function. Update the code so that the function makes a new node with the value of `list->num` and recursively calls `copyList` to assign the `next` value of the new node. (should be one line of code using the function call for makeNode)

2. In main, create a new list called lst3 and copy lst1 to lst3.

```
Node * lst3 = copyList(lst1);
print(lst3);
```

3. At the bottom of main, free `lst3`.

4. Do any other modification and testing to be sure `copyList` is working correctly.

5. What is the running time for `copyList` (suppose the linked list has N items)? O(______)

Part 5: mergeList – Fix the recursive cases

1. Examine the `mergeList` function. The base cases are done for you. You should complete the recursive cases so that a new node is made with the appropriate value and its `next` field is the recursive call on the two appropriate lists. (each recursive case should be one line of code)

For example, if lst1 has 2 4 5 8 10 12
and lst 2 has 1 2 5 8

then, the function should return the making of a new node with value 1 (`lst2->num` is greater than `lst1->num`) and recusively merge `lst1` and `lst2->next`.

Each case should be one line of code in `mergeList`.

2. Why do the recursive calls for `mergeList` eventually reach a base case?

____

3. Suppose list1 has 200 items and list2 has 200 items. How many calls does `mergeList` make to merge the two lists (in worst case)? ____________________

Checkpoint 2 [40 points]: Show your lab instructor/assistant your program running and your code. Show your answer to the question above.

If you finish early and want more practice, try writing the code for one or more of the following functions:

- `insertInOrder` – takes an int n and a pointer to a list; inserts a new node with num n into the list, keeping the list sorted in ascending order
- `reverse` – takes a list and returns a new list with the nodes from list in reverse order
- `numInRange` – takes a lower bound min and upper bound max and a list and returns the number of nodes with values >= min and <= max

Part 5: Opaque object design and make files.

Over the last couple of weeks, we have been working towards using header files in C and definition files. We also have looked at opaque objects, I.e a strategy to conceal our known data type using a void pointer.
In this section, you will create a list_list.h file and a link_list.c file. The header file will be your data structures blueprint and will store the function definitions and the opaque object definition. Make sure that the file is guarded using conditional if or "pragma once" as done in the vectors class.

You definition file shall have the function definitions and the definition for your known type.

Make sure that any functions that were using the known type will now only using the opaque type but internally they will be working with the known type.

Lastly create a makefile that will be used to compile and run your work.

Checkpoint 3 [20 points]: Show your lab instructor/assistant your program running and show your multiple files and opaque objects and your make file

If any checkpoints are not finished, they are due in one week. You may submit screenshots, code files, and answers to questions electronically (via Canvas) or submit printouts of these to the TA.
Remember to test your work on Khoury server and also to push it to Github