CS 222: System Programming

Spring 2020
Lab #7: Singly Linked Lists

Due Thursday**, 04/09/20, 11:59 PM** (no extension**)**

1. Introduction

This lab deals with the combination of dynamic memory allocation and structures to create a common data structure known as a singly-linked list, shown in Figure 1.

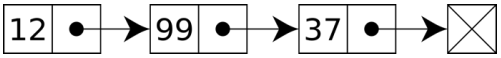

Figure 1: A linked list whose nodes contain two fields: an integer value and a pointer to the next node. The last node is linked to NULL pointer used to signify the end of the list. *(source: https://en.wikipedia.org/wiki/Linked_list)*

The head pointer to the first node, which allow you to traverse the list, is not shown.

The list you will implement has the option being an unsorted or sorted singly-linked list decided by the user. If being a sorted linked list, the nodes will be **sorted** from lowest to highest value. You will complete five functions (excluding main function), which allow you to read user's command into a dynamically resizable string, add or delete a node, find a node containing a given value, or print the entire contents of the list. There are two versions of the add, find and delete functions, one for unsorted linked list the other for sorted – the unsorted version is given.

2. Deliverables

This assignment uses multiple files, each of which is provided on Canvas:

- ***lab7_llist.c*****:** Main program. **Complete the file as indicated inline.**
- ***lab7_functions.h*****:** Header file that contains structure definitions and function prototypes to be used in this lab. **Do not change the contents of this file.**
- ***lab7_functions.c*****:** Definitions for the functions described in *lab7_functions.h*. **You should only complete the functions in this file—do not change any of the #include directives, structure definitions, or function prototypes (i.e., function return types and arguments).**
- ***Makefile***: A Makefile that allows you to compile your program into a final executable named `lab7`.

To complete this lab, you will complete *lab7_functions.c* and *lab7_llist.c*. If each function is properly written, the entire program will work correctly. Zip all 4 files into **lab7.zip** and upload only the zip file to Canvas. Ensure your file names match the names specified above. Failure to meet this specification will reduce your grade by 10 points.

3. Specifications

The main program (***lab7_llist.c***) will first prompt the user to make a choice on the type of the linked list – sorted or unsorted, then stick with the choice.

- **'Y' or 'y':** the list is sorted.
- **'N' or 'n':** the list is unsorted.
- **Anything else:** program won't proceed to the next stage if choice is invalid.

In addition to linked list type choice question, `main()` function also recognizes five different commands, most of which call a function described in *lab7_functions.h* and defined in *lab7_functions.c*:

- **add**: Prompts the user to enter a number, then adds that number to the list using either the **`addNode()`** or **`addSortedNode()`** function.
- **delete**: Prompts the user to enter a number, then removes that word from the list using either the **`deleteNode()`** or **`deleteSortedNode()`** function.
- **find**: Prompts the user to enter a number, then searches the list for that number using either the **`findNode()`** or **`findSortedNode()`** function.
- **print**: Prints the entire contents of the list using the **`printList()`** function.
- **exit:** free the memory of the entire list and exit the program.

lab7_functions.h contains function prototypes as well as structure definitions. The singly-linked list is defined using the `LLnode` structure:

- **LLnode:** A single node in the list, which contains two fields:
  - **value:** An integer number, which is the data stored in this node.
    - If the list is determined as a sorted linked list by the user, the values should be stored in ascending order.
  - **next:** A pointer to the next node in the list. This pointer is NULL if the node is the last entry in the list.

lab7_functions.c contains the functions to be implemented as described below—again, **<u>you only need to complete the given functions, nothing else in that file. You should NOT modify the function prototypes or other variables</u>:**

`char *readCmd()`

Reads string from standard input and dynamically allocates space. The function starts with a one-character-long dynamically allocated string to account for the NULL terminator. Assuming user input string terminates with a newline `'\n'`, the function then reads in the string character by character. When reading in each character, the function will increase the size of the dynamic string by one character size. Once all characters have been processed, the dynamic string need to be NULL-terminated and returned.

Function should have proper guard on failed memory allocation.

3. Specifications (continued)

One of the other four functions to be completed is:

`LLnode *findSortedNode(LLnode *list, int v)`

Search **`list`** for a node containing a number matching **`v`**. Return a pointer to this node if it is found and return NULL otherwise. No need to search the entire list as the list is sorted.

`void printList(LLnode *list)`

Go through the entire list and print the number stored in each node on its own line. If the list is empty, print "List is empty".

`LLnode *addSortedNode(LLnode *list, int v)`

Create a new node holding value **`v`**, then add that node to the list. Notes:

- This function must maintain the list order—the new number **`v`** must be stored in ascending order. You must therefore find the correct location before inserting the node into the list.
- If the function fails to allocated the memory space for the new node, it will report the error, free all the existing nodes back to the heap memory, then exit gracefully (refer to **`addNode()`** function).

The last function to be completed is:

`LLnode *delSortedNode(LLnode *list, int v)`

Find the node containing the number **`v`**, then remove that node from the list. If no matching node is found, do not modify the list. No need to search the entire list as the list is sorted. A few notes:

- This function essentially does the opposite of the **`addSortedNode()`** function, once the node to be removed has been found:
    - Modify **next** in the node before the chosen node so that it points passing the chosen node.
    - Remove the chosen node.
        - Removal of a node implies that any space that was dynamically allocated when creating the node must be deallocated to remove it.

4. Hints

Design process: I would suggest handling the program in the following order:

1. Start with the **`readCmd()`**, use the debugger or a **`printf()`** function to check the contents in the string immediate after reading in.
2. Finish the **`main()`** function. Then test it with unsorted linked list functions that are already given.
3. Move on to the **`printList()`**, and at least test the case where the list is empty.
4. Next, write the **`addSortedNode()`** function. At least two of the first three cases you test will have to be special cases, since the list starts out as an empty list, and the second number you add will become either the first or last item in the list.
    - You can test the operation of this function, as well as **`printList()`**, by running the main program and alternating "add" and "print" commands.
5. Once you have handled all possible cases for **`addSortedNode()`**, write the **`findSortedNode()`** function.
    - Test the function by adding items to the list and using the "find" command.
6. Finally, write the **`delSortedNode()`** function.
    - Test this function by adding items to the list, using the "delete" command, and then using the "print" command to show the results. Be sure to test all of the special cases.

If you encounter errors, running your program in the debugger is the most effective way to find them. Recall that the debugger offers the ability to "step into" a function (F11 in Visual Studio) so that you can see each step within the function you have written, or simply "step over" (F10) the function and treat a function call as a single statement.

5. Grading

Deduction Code	Description	Points Deducted
1	Program does not compile, but some code was given.	-60
2	Comments	-10
2a	No header comments	-4
2b	Body of program contains no single line of comment	-4
2c	Other comments such as comments for key variables. Header comments is present but missing some key components	-2
3	File names are incorrect (including source files and Makefile)	-10
4	Inconsistent program style • Indentation, braces, descriptive variable names, etc. • -1 for each occurrence and maxed at -5	-5
5	Input error	-10
5a	Program interprets more commands than the given five. (-5)	-5
5b	Program cannot interpret some valid commands. (-5)	-5
6	Output error	-55
6a	Output error due to incorrect readCmd() function i. String is not dynamically resized. (-5) ii. No guard on allocation failure. (-5) iii. String is not Null-terminated. (-5)	-10
6b	Output error due to incorrect add, find, or delete function i. Each function is worth 10 points. ii. No guard on allocation failure. (-5)	-30
6c	Output error due to incorrect main() function (-5 for each until maxed) i. Incorrect calling on the functions under each command ii. Linked list not freed before ending the program	-10
6d	Printed information does not match with the given sample output exactly (-1 for each occurrence until maxed) i. There should be no extra information printed ii. User prompts and error messages should match	-5
7	Lack of a Makefile or Makefile cannot compile the code	-10
7a	Makefile does not have a recipe for cleaning the object files and executable. (-5)	-5
7b	Makefile does not separate the compilation and linking steps. (-5)	-5

6. Test Cases

Your output should match these test cases exactly for the given input values. I will use these test cases in grading of your program, but will also generate additional cases that will not be publicly available. Note that these test cases may not cover all possible program outcomes. You should create your own tests to help debug your code and ensure proper operation for all possible inputs.

The test cases were generated in Xcode, so I've copied and pasted the output below, rather than showing a screenshot of the output window. User input is underlined, although it won't be when you run the program.

Test case 1:

```
Use sorted linked list? (Y/N): X
Invalid choice X

Use sorted linked list? (Y/N): n

Enter command: adding it
Invalid command adding it

Enter command: print
List is empty

Enter command: printing
Invalid command printing

Enter command: add
Enter number to be added: 20

Enter command: add
Enter number to be added: 5

Enter command: add
Enter number to be added: 10

Enter command: print
10
5
20

Enter command: delete
Enter number to be deleted: 19
19 not found in list
```

6. Test Cases (continued)

```
Enter command: delete
Enter number to be deleted: 20
20 was deleted from list

Enter command: find
Enter number to be found: 20
20 not found in list

Enter command: quit
Invalid command quit

Enter command: exit
```

Test case 2:

```
Use sorted linked list? (Y/N): Y

Enter command: add
Enter number to be added: 20

Enter command: add
Enter number to be added: 5

Enter command: add
Enter number to be added: 10

Enter command: print
5
10
20

Enter command: find
Enter number to be found: 10
10 found in list

Enter command: add
Enter number to be added: 15

Enter command: print
5
10
15
20
```

6. Test Cases (continued)

```
Enter command: delete
Enter number to be deleted: 19
19 not found in list

Enter command: delete
Enter number to be deleted: 10
10 was deleted from list

Enter command: print
5
15
20

Enter command: exit
```