EECE.2160: ECE Application Programming

Fall 2017

Programming Assignment #10: Doubly-Linked Lists

Due **Monday, 12/18/17**, 11:59:59 PM

(Extra credit (≤ 5 pts on final average), no late submissions or resubmissions)

1. Introduction

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

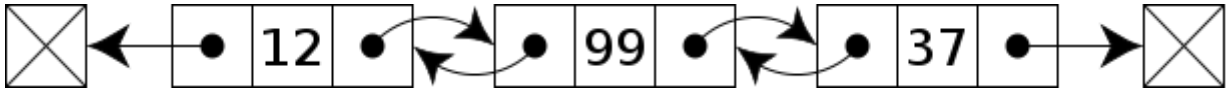


Figure 1: Doubly-linked list in which each node contains three fields--pointers to the previous and next nodes in the list, and a single integer as data. *(source: http://en.wikipedia.org/wiki/Doubly-linked_list)*

The boxes holding 'X' at the start and end of the list show the first and last nodes have NULL pointers for their previous and next pointers, respectively. Pointers to the first and last nodes, which allow you to traverse the list in either direction, are not shown.

The list you will implement is a sorted doubly-linked list in which each node stores two double-precision values, and the nodes are sorted from lowest to highest value. You will complete six functions, which allow you to 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 find and print functions, one for each possible traversal direction.

2. Deliverables

This assignment uses multiple files, each of which is provided on the course web page:

- ***prog10_main.c*****:** Main program. **Do not change the contents of this file.**
- ***DLList.h*****:** Header file that contains structure definitions and function prototypes to be used in this assignment. **Do not change the contents of this file.**
- ***DLList.c*****:** Definitions for the functions described in *DLList.h*. **You should only complete the functions in this file—do not change any of the #include statements, structure definitions, or function prototypes (i.e., function return types and arguments).**

To complete this assignment, you will complete each of the functions in *DLList.c*. If each function is properly written, the entire program will work correctly.

Submit all three files by uploading these files to your Dropbox folder. ***Place the files in directly in your shared folder—do not create a sub-folder to hold them.*** Ensure your file names match the names specified above. Failure to meet this specification will reduce your grade, as described in the program grading guidelines.

3. Specifications

The main program (***prog10_main.c***) recognizes five different commands, most of which call a function described in *DLList.h* and defined in *DLList.c*:

- **add**: Prompts the user to enter two values, then adds those values to the list using the **`addNode()`** function.
- **delete**: Prompts the user to enter a value, then removes that value from the list using the **`deleteNode()`** function.
- **find**: Prompts the user to enter a value, then searches the list for that value using both the **`findFWD()`** and **`findREV()`** functions.
- **print**: Prints the entire list using the **`printFWD()`** and **`printREV()`** functions.

DLList.h contains function prototypes as well as structure definitions. The doubly-linked list is defined using two structures:

- **DLNode:** A single node in the list, which contains four items:
  - **prev:** A pointer to the previous node in the list. This pointer is NULL if the node is the first entry in the list.
  - **next:** A pointer to the next node in the list. This pointer is NULL if the node is the last entry in the list.
  - **val1, val2:** Two double-precision values, which are the data stored in this node.
    - The list should be sorted so nodes are sorted from lowest to highest value, based on their val1 fields.
    - If two nodes have the same val1 values, they should be sorted from lowest to highest based on their val2 fields.
- **DLList:** A structure that contains two pointers, **firstNode** and **lastNode**, which point to the first and last nodes in the list, respectively.
  - If the list is empty, both pointers are NULL.
  - If the list holds only one node, both **firstNode** and **lastNode** point to that node.

You are responsible for completing each of the functions in ***DLList.c*** described below—again, **note that this file is the only one you should modify:**

`DLNode *findFWD(DLList *list, double v, int *num)`

Search **`list`** for a node in which only the val1 field matches **`v`**. Search starts with the first node in the list. The function should calculate the number of iterations required to find the value and store it at the address pointed to by **`num`**. Return a pointer to the node with the matching value if it is found, and return NULL otherwise.

`DLNode *findREV(DLList *list, double v, int *num)`

Same as **`findFWD()`**, but the search begins with the last node and traverses the list in reverse.

3. Specifications (continued)

Three of the other four functions to be completed are:

`void printFWD(DLList *list)`

Go through the entire list, starting with the first node, and print the values stored in each node on their own line. If the list is empty, print "List is empty."

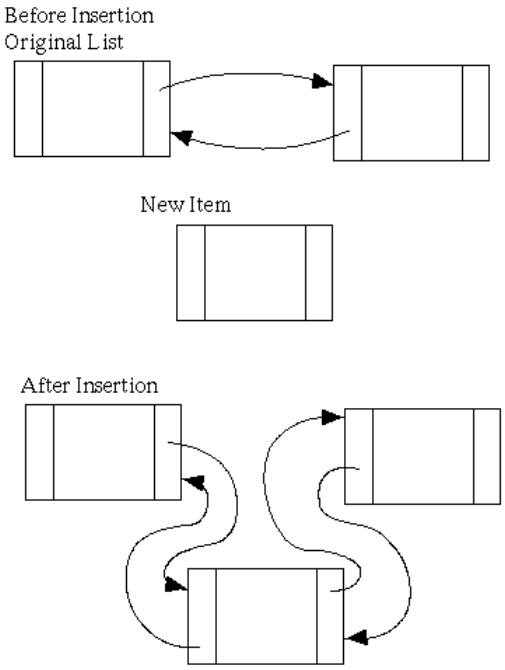
`void printREV(DLList *list)`

Same as **`printFWD()`**, but prints the last node first and traverses the list in reverse.

`void addNode(DLList *list, double v1, double v2)`

Create a new node holding values **`v1`** and **`v2`**, then add that node to the list. Notes:

- The process for inserting a node in a doubly-linked list is below, with pointer changes shown in the figure *(source: http://www.cs.grinnell.edu/~walker/courses/161.sp12/modules/lists/reading-lists-double.shtml)*:
  - Create a new node
  - Place the data in the node (**v1** in the **val1** field, **v2** in **val2**)
  - Set the **prev** and **next** pointers inside the new node to point to the correct nodes.
  - Modify **next** in the node before the new one.
  - Modify **prev** in the node after the new one.
- This function must maintain the list order—all **val1** fields should be sorted from lowest to highest, with **val2** fields used to sort nodes with matching **val1** fields. You must therefore find the correct location before inserting the node into the list.



- There are three special cases to account for:
  - The new node is the only thing in the list (i.e., list is empty at start of function)
  - The new node becomes the first node in the list (but list contains other nodes)
  - The new node becomes the last node in the list (but list contains other nodes)

3. Specifications (continued)

The final function to be completed is:

```
void delNode(DLList *list, double v)
```

Find the node containing the word **`v`** **(only testing the val1 field of each node)**, then remove that node from the list. If no matching node is found, do not modify the list. A few notes:

- This function essentially does the opposite of the **`addNode()`** function, once the node to be removed has been found:
  - Modify **next** in the node before the chosen node.
  - Modify **prev** in the node after 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.
- There are, once again, three special cases to account for:
  - The node to be removed is the only thing in the list (both first and last)
  - The node to be removed is the first node in the list (but not also the last node)
  - The node to be removed is the last node in the list (but not also the first node)

4. Hints

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

1. Start with the two print functions, and at least test the case where the list is empty.
2. Next, write the **`addNode()`** 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 word you add will become either the first or last item in the list.
   - You can test the operation of this function, as well as the print functions, by running the main program and alternating “add” and “print” commands.
3. Once you have handled all possible cases for **`addNode()`**, write the **`findFWD()`** and **`findREV()`** functions.
   - Test these functions by adding items to the list and using the “find” command.
4. Finally, write the **`delNode()`** 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.

Similarities: Please note that many of the functions are similar to those used for a sorted singly-linked list, which was discussed in lectures 33-36. In particular:

- The **`findFWD()`** and **`printFWD()`** functions are virtually identical to the **`findNode()`** and **`printList()`** functions.
- The **`addNode()`** and **`delNode()`** functions are similar—an additional pointer in each node makes the functions slightly more complicated, but also makes it easier to identify the nodes before and after the one being added or deleted.

Handling first and last nodes: The **`addNode()`** and **`delNode()`** functions must each deal with three special cases involving the first and last nodes in the list:

- The **DLList** structure contains pointers to the first (**firstNode**) and last (**lastNode**) nodes in the list. Therefore, any operation that changes what node is first or last must also change the appropriate pointer in the **DLList** structure, not just the **prev** and **next** pointers in the **DLNode** structures within the list.
- In each node at one end of the list, at least one of the pointers in that node is a NULL pointer. In the first node, **prev** is NULL; in the last node, **next** is NULL. You must account for these NULL pointers when working with these nodes.

5. Grading

As noted in class, grading for this program will be stricter than normal. In particular:

- A program that does not compile will receive a grade of 0.
- A program that does not generate proper output will receive minimal credit.
    - If your functions work but the program doesn't show the results, you will receive some credit for writing functions that appear correct but will be penalized heavily for making it impossible to test those functions.

The general point breakdown will be:

- Find functions (**`findFWD()`**, **`findREV()`**): 25 points
    - Each function must traverse list in proper order, return the correct node pointer, and count the number of iterations to find the value if present.
- Print functions (**`printFWD()`**, **`printREV()`**): 25 points
    - Print functions must print list in proper order with correct formatting.
- **`addNode()`**: 25 points
    - Function must maintain a sorted list based on val1 values, with val2 values used to sort nodes that duplicate val1 values.
- **`delNode()`**: 25 points
    - Function must delete appropriate node. **For full credit, this function should only use a single node pointer, rather than the "prev/cur" pointer pair required in a singly-linked list.**

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 lab, 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.

I've copied and pasted the output from each test case below, rather than showing a screenshot of the output window. User input is underlined in each test case, but it won't be when you run the program.

```
Enter command:  add
Enter values to be added:  1.2 3.4

Enter command:  add
Enter values to be added:  5.6 7.8

Enter command:  add
Enter values to be added:  0.1 0.1

Enter command:  add
Enter values to be added:  3 3.33

Enter command:  print

Contents of list (low to high):
0.10 0.10
1.20 3.40
3.00 3.33
5.60 7.80

Contents of list (high to low):
5.60 7.80
3.00 3.33
1.20 3.40
0.10 0.10

Enter command:  find
Enter value to be found:  1.2
1.20 found in node: 1.20 3.40
Forward search: 2 iterations
Reverse search: 3 iterations

Enter command:  add
Enter values to be added:  4 2

Enter command:  add
Enter values to be added:  4 6
```

6. Test Cases (continued)

```
Enter command:  add
Enter values to be added:  4 4

Enter command:  print

Contents of list (low to high):
0.10 0.10
1.20 3.40
3.00 3.33
4.00 2.00
4.00 4.00
4.00 6.00
5.60 7.80

Contents of list (high to low):
5.60 7.80
4.00 6.00
4.00 4.00
4.00 2.00
3.00 3.33
1.20 3.40
0.10 0.10

Enter command:  find
Enter value to be found:  4.1
4.10 not found in list

Enter command:  find
Enter value to be found:  3.4
3.40 not found in list

Enter command:  delete
Enter value to be deleted:  5.6

Enter command:  delete
Enter value to be deleted:  0.1

Enter command:  delete
Enter value to be deleted:  3
```

6. Test Cases (continued)

```
Enter command:  print

Contents of list (low to high):
1.20 3.40
4.00 2.00
4.00 4.00
4.00 6.00

Contents of list (high to low):
4.00 6.00
4.00 4.00
4.00 2.00
1.20 3.40

Enter command:  exit
```