EECE.2160: ECE Application Programming

Summer 2016

Programming Assignment #9: Doubly-Linked Lists

Due **Friday, 6/24/16**, 11:59:59 PM

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.

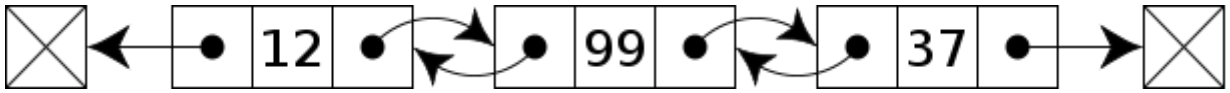


Figure 1: Doubly-linked list in which each node contains three fields--pointers to the previous and next nodes in the list, and a single integer as data. *(source: http://en.wikipedia.org/wiki/Doubly-linked_list)*

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 the data stored in each node is a string, and the nodes are sorted in alphabetical order. You will complete four functions, which allow you to add or delete a node, find a node containing a given string, or print the entire contents of the list.

2. Deliverables

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

- ***prog9_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 (***prog9_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 a word, then adds that word to the list using the **`addNode()`** function.
- **delete**: Prompts the user to enter a word, then removes that word from the list using the **`deleteNode()`** function.
- **find**: Prompts the user to enter a word, then searches the list for that word using the **`findNode()`** function.
- **print**: Prints the entire contents of the list using the **`printList()`** function.

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 three 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.
  - **word:** A string holding a single word, which is the data stored in this node.
    - The list should be sorted so that the words are stored in alphabetical order.
    - You can assume all words are written solely in lowercase letters.
- **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 *findNode(DLList *list, char *str)`

Search **`list`** for a node containing a word matching **`str`**. Return a pointer to this node if it is found, and return NULL otherwise.

`void printList(DLList *list)`

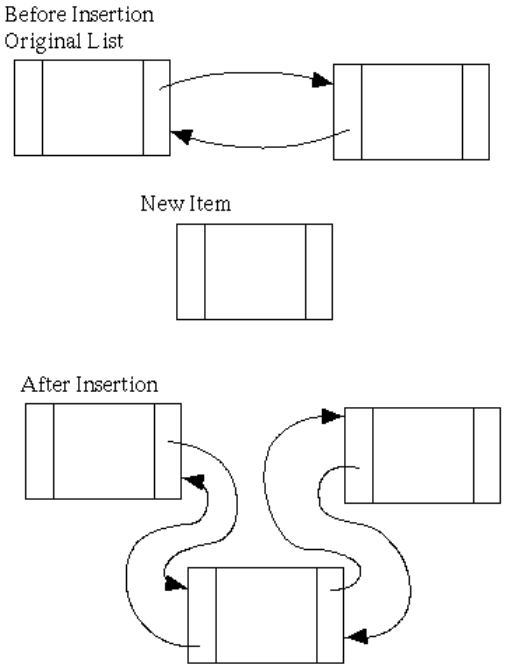
Go through the entire list and print the word stored in each node on its own line. If the list is empty, print “List is empty.”

3. Specifications (continued)

The other two functions to be completed are:

`void addNode(DLList *list, char *str)`

Create a new node containing the word **`str`**, then add that node to the list. A few 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--remember, your data is a dynamically allocated string.
  - 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—the new word must be stored in alphabetical order. 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)

`void delNode(DLList *list, char *str)`

Find the node containing the word **`str`**, 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 **`printList()`**, 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 **`printList()`**, by running the main program and alternating "add" and "print" commands.
3. Once you have handled all possible cases for **`addNode()`**, write the **`findNode()`** function.
   - Test this function by adding items to the list and then 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 lecture 12. In particular:

- The **`findNode()`** and **`printList()`** functions are virtually identical.
- The **`addNode()`** and **`delNode()`** functions are similar—the presence of 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 issues that you must account for in these cases are the following:

- 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.

4. Hints (continued)

Alphabetical sorting: Having a sorted list makes searching the list more efficient, because you can stop searching once you reach a point at which the value you're searching for would be out of order. For example, if you search a list of integers sorted from lowest to highest, and you're looking for the value 5, you can stop searching once you find any value greater than 5.

To handle string ordering, remember that the string compare functions, `strcmp()` and `strncmp()`, return a positive value if the first string is "greater than" the second and a negative value if the first string is "less than" the second. In other words, if the two strings are in alphabetical order, these functions return negative values. For example:

- If `n = strcmp("add", "subtract")` → `n < 0`
- If `n = strcmp("add", "addition")` → `n < 0`
- If `n = strcmp("add", "aardvark")` → `n > 0`

5. 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.

```
Enter command:  print

List is empty

Enter command:  add
Enter word to be added:  test

Enter command:  print

Contents of list:
test

Enter command:  add
Enter word to be added:  math

Enter command:  add
Enter word to be added:  western

Enter command:  print

Contents of list:
math
test
western

Enter command:  delete
Enter word to be deleted:  test
```

```
Enter command:  find
Enter word to be found:  math
math found in list

Enter command:  find
Enter word to be found:  test
test not found in list

Enter command:  print

Contents of list:
math
western

Enter command:  delete
Enter word to be deleted:  western

Enter command:  delete
Enter word to be deleted:  math

Enter command:  print

List is empty

Enter command:  exit
```