Week 03 Lab Exercise

MyEd and DLLs

Objectives

- to illustrate file manipulation
- to implement a doubly-linked list ADT
- to design comprehensive test-cases
- to build a simple line-based text editor

Admin

Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week 03 Lab or in the Week 04 Lab
Submit	`give cs2521 lab03 DLList.c testList.c` or via WebCMS
Deadline	must be submitted by 11:59pm on Sunday 18 March

Aims

This lab aims to re-examine an idea that we have looked at a few times in lectures (without being explicit about it): **command interpreters**. More importantly, it aims to give you practice manipulating a useful data type: **doubly-linked lists**. Finally, it aims to demonstrate how files are manipulated in Linux. These ideas are tied together in a simple *text editor*, a program in the style of the classic Unix text editor `ed`, an editor which roamed the earth contemporaneously with the dinosaurs.

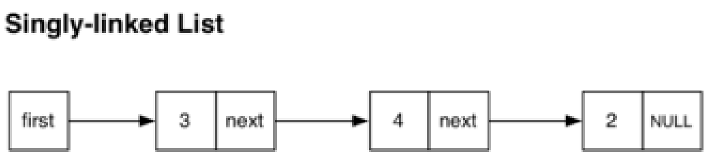
Background

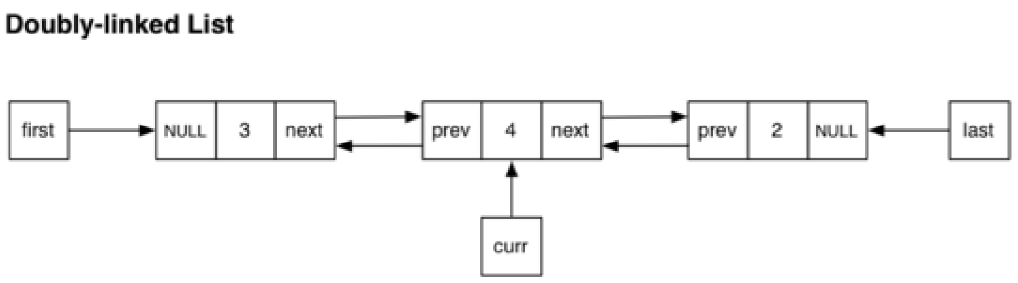
A text editor deals with a file of text, reads it, allows the user to modify it, and then writes it back. This implies that the editor has an internal representation that it deals with in between the reading and the writing. For this example, we consider that the internal representation is a sequence of lines, represented by a doubly-linked list of strings. Each node in the list represents one line in the file being edited.

Since a common operation is to "move around" the file, looking for lines to change, a doubly-linked list with a notion of a current node provides a convenient way to do this. The editor itself is implemented as a loop which: prints a prompt*, reads a command, carries out the command (possibly changing the state of the list), and then repeats. Many of the programs that we build for exploring data structures during this course will have a similar structure. (* The fact that it prints a prompt is an improvement on the real ed editor, which doesn't. Not prompting the user may be minimalist, but isn't great interface design).

Doubly-linked Lists

Doubly-linked lists are a variation on "standard" linked lists where each node has a pointer to the previous node as well as a pointer to the next node. The following diagram shows the differences:





As shown in the diagram, our version of doubly-linked lists also has a notion of a "current" node, and current can move backwards and forwards along the list. Our doubly-linked list does insertions either immediately before or immediately after the current node. Deletion always causes the current node to be removed from the list.

In this lab, we'll use a doubly-linked list ADT called `DLList` whose interface is given in the file

```
/home/cs2521/web/18s1/labs/week03/code/DLList.h
```

The representation of `DLList` is a doubly-linked list of `DLListNode`s, accessed via a record containing the following four fields:

- `nitems`: a count of the number of nodes/items in the list
- `first`: a pointer to the first node in the list
- `last`: a pointer to the last node in the list
- `curr`: a pointer to the "current" node in the list

Several important conditions must hold at all times during the lifetime of a `DLList`. Technically, these conditions are called *invariants*, and are checked for in the function `validDLList()`.

- Whenever the list is non-empty, it must have a defined current node. It is not possible for the current node pointer to be NULL if there is at least one node in the list.
- The counter `nitems` must always contain a value which is the same as the number of nodes in the list.

The book-keeping operations (e.g. `newDLList()`, `getDLList()`, etc.) are relatively straightforward, so we'll describe only the operations relevant for this lab in detail.

char *DLListCurrent(DLList L)

Return a pointer to the string value for the current node. It is an error to call this function if the list contains no nodes.

int DLListMove(DLList L, int n)

Move the current pointer either backwards or forwards `n` positions from the current position. Move forwards in `n` is positive, and move backwards if `n` is negative. If current reaches either end of the list before it has moved `n` positions, it stops at the end node. If the movement stops with current at either the first or last node, return a value of 1; otherwise, return a value of 0.

int DLListMoveTo(DLList L, int n)

Set the current pointer to point to the n^{th} node in the list, where nodes are indexed starting from 1. If `n` is smaller than 1, the current node will be set to the first node. If `n` is larger than the number of nodes in the list, the current node will be set to the last node. The return value has the same behaviour as the return value of `DLListMove()`.

void DLListBefore(List, char *)

Insert a new node in the list immediately before the current node. The new node becomes the current node. If the node is inserted before the first node, it also becomes the new first node.

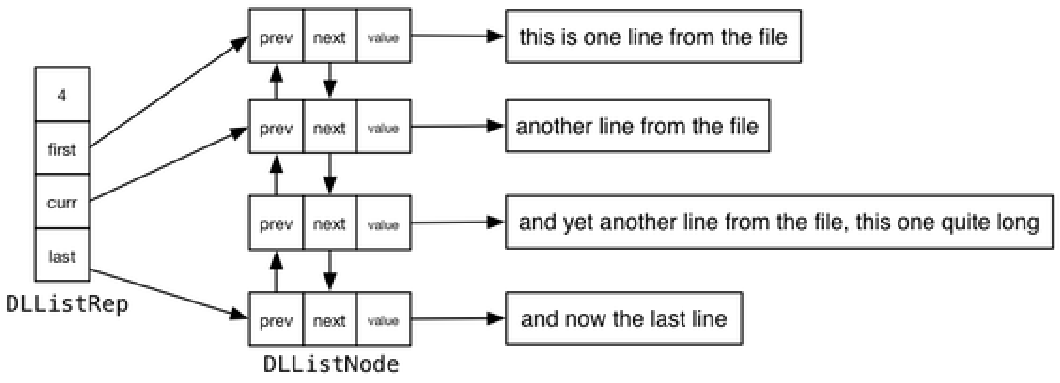
void DLListAfter(List, char *)

Insert a new node in the list immediately after the current node. The new node becomes the current node. If the node is inserted after the last node, it also becomes the new last node.

void DLListDelete(List)

Remove the current node from the list (and free its memory, including the memory used to hold the string). The current node becomes the node immediately following the node that was removed. If the removed node is the last node (i.e. the node at the end of the list), then the current node is set to the new last node. If the node removed was the only node in the list, then current becomes NULL. If `DLListDelete()` is called with an empty list, it should simply return without making any changes to the list.

Note that the supplied `DLList` ADT implements a doubly-linked list of strings. The strings are created outside the list nodes, as in the diagram below:


4
first
curr
last
DLListRep
prev
next
value
DLListNode
this is one line from the file
another line from the file
and yet another line from the file, this one quite long
and now the last line

The storage used by the strings needs to be managed separately to the storage used by the list nodes, although still as part of the implementation of the `DLList` ADT.

`myed`: a Text Editor

Text editors are programs that allow you to manipulate text files interactively; they provide a user interface where you can load a file, make changes to it, and then save an updated version of the file. Text editors like `myed` are seriously retro and hark back to the 60's, when screen-based editors like `vi` and `emacs` didn't exist, and graphical editors like `gedit` weren't even a gleam in some inventor's eye (except maybe Doug Engelbart).

The `myed` program provides an extremely simple command-line interface where you enter one-letter commands to manipulate the loaded file. Despite its primitive interface, `myed` can actually make changes to text files. Here's an example session with the editor (using the normal notational conventions: what the system prints is in `this font`, what you type is in **`this font`**, comments are in `small grey font`).

```
$ ./myed text                // start the editor, loading a file called "text"
> %                          // show all of the lines in the file
this is                      // the current line is the first line
a small file
containing a few lines
of boring text
> .                          // show the current line
this is
> i                          // insert a line before the first line
new first line               // you type the new line here
> %                          // show all of the lines again, so you can see the changes
new first line               // note that the new line is now the current line
this is
a small file
containing a few lines
of boring text
```

```
> n                          // move to the next line
this is
> n                          // move to the next line
a small file
> n                          // move to the next line
containing a few lines
> -2                         // move back two lines
this is
> w                          // save a copy of the modified file
> q                          // quit the editor
$ ls                         // list the files in the current directory
...  text  text.new ...      // should see the original file and the new version
$ cat text.new               // look at the contents of the new text file
new first line
this is
a small file
containing a few lines
of boring text
$
```

The best way to get a feeling for how the editor works is to play with it. There's a working version in the class directory, which you can execute on the CSE lab machines via the command:

```
$ /home/cs2521/web/18s1/labs/week03/myed FileName
```

Choose any old text file you like as the filename. Once you have an editor prompt, the command `?` will tell you what other commands are available.

If you play with the supplied `myed`, I can guarantee that you will run the `%` command a lot. Editing is better when you get immediate feedback on the changes you've made, which led to the development of screen-based editors like `vi` and `emacs` soon after the novelty of editing in the `ed` style wore off (and once graphical displays were available). If you're keen to experience the original `ed`, it's still available under Linux.

Setting Up

Set up a directory for this lab under your `cs2521/labs` directory, change into that directory, and run the following command:

```
$ unzip /home/cs2521/web/18s1/labs/week03/lab03.zip
```

If you're working at home, download `lab03.zip`, unzip it there, and then work on your local machine.

If you've done the above correctly, you should now find the following files in the directory:

`Makefile`	a set of dependencies used to control compilation
`DLList.h`	interface definition for the List ADT
`DLList.c`	(partial) implementation for the List ADT
`testList.c`	main program for testing the List ADT
`myed.c`	main program for a simple line-based text editor
`text`	a simple text file that you can use to test `myed`

Once you've got these files, the first thing to do is to run the command

```
$ make
```

This will produce messages about compiling with `gcc` and will put two executable files in your directory (along with some `.o` files).

`testL`

This is the executable for the `testList.c` program. As supplied, it runs a number of simple tests on the `DLList` datatype. You can use it by running `./testL`, typing in a few lines of text and then using contrl-D to end the input; the program should display all of the lines you typed and not produce any `assert` errors. You will need to add more extensive tests as part of this Lab.

`myed`

This is the executable for the `myed.c` program, which implements a very simple line-based text editor. It takes as argument a file name, reads this file into a doubly-linked list of lines, and then supports various actions on the lines. See the `myed.c` file for details on what commands are available, or type the **?** command inside the editor to get help.

Since `myed` is a text editor, it needs to be invoked by giving it a file to edit. The `lab.zip` archive contains a small text file (called `text`) that you can use with the editor. Invoke the editor using a command like:

```
./myed  text
```

Task 1

The `myed` program is complete, but doesn't work because some of the functions in `DLList.c` are not complete. You should complete these functions so that the behaviour of your `myed` program is the same as the behaviour of the sample solution:

```
$ /home/cs2521/web/18s1/labs/week03/myed  text
```

You can execute the sample solution on the CSE workstations (only), by typing the full path name exactly as above. It would be a good idea to play with the sample solution at least briefly to get an idea of how it is supposed to work.

Note that since the two `myed` programs read from `stdin` and write to `stdout`, one automatic way of testing them would be to devise a set of editing scripts (files containing sequences of `myed` commands), and then apply each script with both your `myed` and the sample `myed` and using `diff` to ensure that they both produce the same output, e.g.

```
$ ./myed text < FileContainingEditCommands > out1
$ /home/cs2521/web/18s1/labs/week03/myed text < FileContainingEditCommands > out2
$ diff out1 out2
```

In order to establish the correctness of your `myed`, you should develop a number of editing scripts, each one being more devious than the previous in attempting to crash your editor (and the sample one, if you can ... if you do work out how to crash the sample editor, let me know, and I'll fix it).

Task 2

Implementing the `DLList` functions to make `myed` work serves two purposes: (a) getting a complete editor, (b) `myed` providing a simple test harness for the `DLList` ADT. However, the way that `myed` exercises the ADT is by no means complete, and more testing is needed.

The `testList.c` program provides a skeleton in which to build a more serious testing framework. At present, it is minimal; it simply uses the `getDLList()` function to read some strings from standard input into a `List`, displays the `List` and then does a sanity check on the representation. You should expand `testList.c` to provide a more complete set of tests to exercise the `DLList` functions.

Note that writing a detailed test suite for all functions in the ADT is quite time consuming. You should at least write comprehensive test cases for the three functions that you implemented. This means that you should have a test for each of the possible states in which each function might be invoked (e.g. empty list, list with one node, etc.). At the least, each test should:

- show the state of the list before the operation (including `curr` and `nitems`)
- indicate what operation is about to be performed
- invoke the operation

- display the state of the list after the operation
- run the `validDLList()` check on the list

You can simply use visual inspection of the output to check whether the operations have worked correctly, and, of course, check with `validDLList()` after each operation. It is not necessary to write automatic checks of the precise state, since this would require writing multiple functions almost as complex as `validDLList()`. If you wish to modify `putDLList()` to give a better display of the `DLList` state (e.g. use short lines and display all of them on a single output line), that would be useful (but not essential).

Submission

You need to submit two files: `DLList.c` and `testList.c`. Submit these either via the `give` command in a terminal window or from WebCMS, and then show your tutor, who'll give you feedback on your coding style, your test quality, and award a mark.

Have fun, *jas*