Updated: April 16, 2019

Filesystem For CS103L-Terminal: Doubly Linked List Edition.

1. Introduction

In this assignment, we will take your Lab 11 to the next level. You will design and implement a program that behaves very similarly to your Mac/Linux terminal or Windows cmd. In order to do that, we need to create a filesystem. With this filesystem in place, you will be able to execute some basic commands to interact with our CS103L terminal such as "***cd***" to change directory, "***rm***" to delete a file or directory, "***mkdir***" to create a directory, "***ls***" to list all items in the current directory, etc (see Section 5 for more commands that you can execute). For now, your filesystem will only support managing directories and text files. In addition, we will keep all our data in memory and you don't actually have to read from or write to an "actual" file. In other words, once you exit the program, all data in your filesystem will be deleted. This assignment should give you an idea on what you will learn in 104 (Data Structures), 350, and 402 (Operating Systems) - we are preparing you for the future.

```
ccc_v1_w_ZDYwZ_94281@runweb13:~$ ./filesys terminal
--------------------------------
Initailizing 103L OS ..........
CS103L_PA6_shell:root/$ mkdir popcorn
CS103L_PA6_shell:root/$ mkdir apple
CS103L_PA6_shell:root/$ mkfile fileA
CS103L_PA6_shell:root/$ ls
apple fileA(0) popcorn
CS103L_PA6_shell:root/$ ls -r
popcorn fileA(0) apple
CS103L_PA6_shell:root/$ write fileA
$> I'm writing to fileA
CS103L_PA6_shell:root/$ read fileA
I'm writing to fileA
CS103L_PA6_shell:root/$ cd popcorn
CS103L_PA6_shell:root/popcorn/$ mkdir insidePopcorn
CS103L_PA6_shell:root/popcorn/$ cd insidePopcorn
CS103L_PA6_shell:root/popcorn/insidePopcorn/$ pwd
root/popcorn/insidePopcorn/
CS103L_PA6_shell:root/popcorn/insidePopcorn/$ cd ..
CS103L_PA6_shell:root/popcorn/$ ls
insidePopcorn
CS103L_PA6_shell:root/popcorn/$ rm insidePopcorn
CS103L_PA6_shell:root/popcorn/$ ls

CS103L_PA6_shell:root/popcorn/$ cd
CS103L_PA6_shell:root/$ ls
apple fileA(20) popcorn
CS103L_PA6_shell:root/$
```

Under the hood, we will be using a doubly linked list. At this point, you should probably have an idea of what it is. Compared to a singly linked list, a doubly linked list helps us in multiple ways. For instance, 1). delete operation takes O(1) if the node to be deleted is given. This is because all you need to do is adjusting the pointers of its next and prev nodes. The information on these nodes can be retrieved directly from the node to be deleted itself. And 2). traverse the list can be done both ways (top -> bottom, bottom -> top). However, to implement a doubly linked list, you need to manage two pointers for each node. For more detail, check *Unit 15*. With that said, this data structure is perfect for our task.

2. What you will learn

In this assignment, you will practice and acquire the following skills:

- Dynamically memory allocation of structs and classes.
- Handling a great amount of pointers.
- Implementing multiple **recursive** functions.
- Creating a program using a doubly linked list as an underlying data structure.

3. Filesystem

What is a filesystem?

A *filesystem* is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk [1]. In the simplest form, a filesystem is a tree where each branch node is a directory and each leaf node is a file.

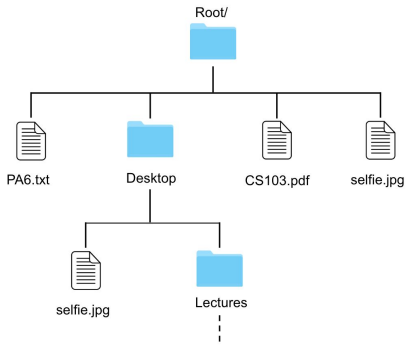

Figure 1 simple filesystem

How are we going to implement this using doubly linked lists?

In this assignment, you are given 2 classes and 2 structs. The 2 structs are *Node* and *Data* (see: *MList.h*). Figure 2 shows how these two structs interact with each other. Essentially what it means is each node has 2 pointers that point to its prev and next node and 1 pointer that points to its data.

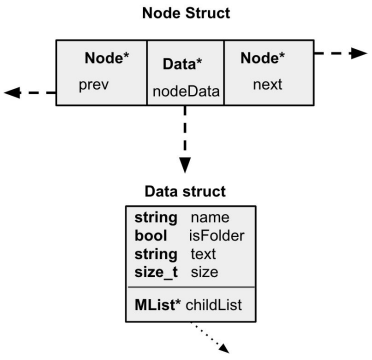

Figure 2- structs blueprint

Things to note:

- *isFolder* is a flag that tells you if the node's data is a file or directory type.
- If a node is a directory type, it should have a pointer (childList) that points to an *MList* object. This MList object holds a list of nodes for the directory that points to it. If we refer back to Figure 1, *Desktop* is a folder so it should have a pointer to an *MList object*. This *MList object* holds 2 nodes: a node that has a file data named "selfie.jpg" and a node that has a directory data named "Lectures".

Figure 3 gives you a glimpse on how we can visualize our filesystem using the two classes and two structs given. The bluish-green box is our OS class object, inside we have *MList* ***wd*** (**W**orking **D**irectory) and *MList dataStack*. The former tells us what directory we are currently in. You can think of it simply as a stack (Last In First Out - LIFO). If you want to "**cd**" to another directory, you can simply push a pointer to a data object (i.e., that directory) at the top of the wd list. If you want to go back to the previous directory (one level before), simply just pop your current directory off from the list. You can access what's inside the current directory through **wd**'s top(). Here in figure 3, since **wd** currently only contains 1 node, top() points to this node and it is the directory we are currently in (the root directory). You can see the list of data our root directory holds by simply accessing its childList (i.e., ***wd.top()->nodeData->childList***) so that you can perform tasks on that list like sorting, deleting, printing, searching, etc.

The latter list, dataStack, contains a list of pointers to all dynamically allocated data objects (files and folders) we have created within our OS class, again this list is a stack (LIFO). We can use this stack to help us deallocate the memory (think about why is that the case and how you can do that!).

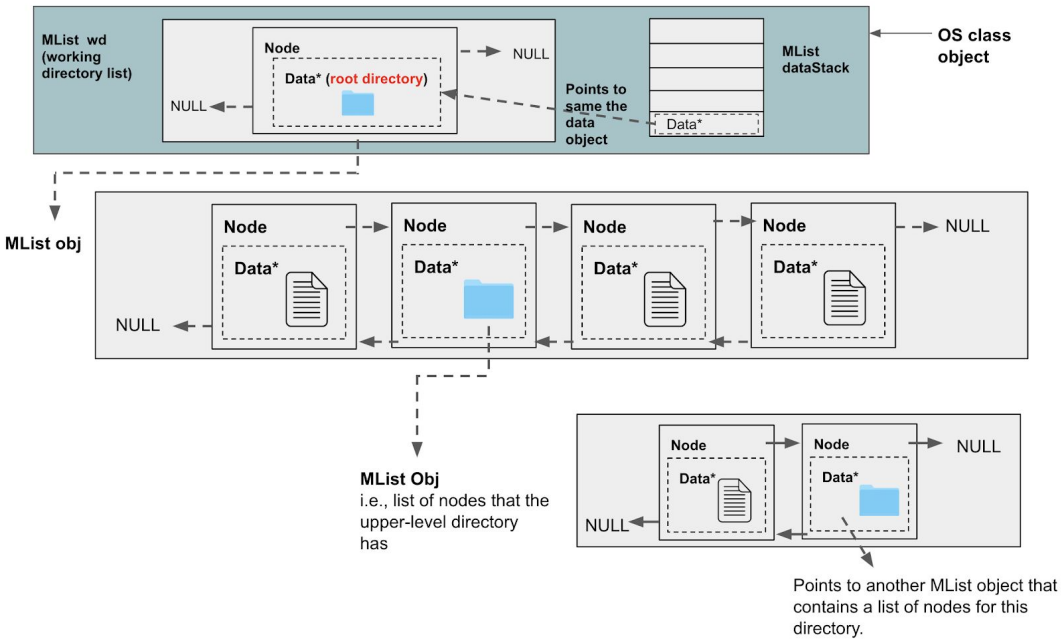

Figure 3: diagram showing the overview of our filesystem

As you can see, our filesystem is merely many doubly linked lists connected to each other.

4. Implementation

- We provided skeleton code in Vocareum
- Use "**make**" to compile your program
- You will be implementing a series of functions in ***mlist.cpp*** and ***os.cpp*****. NOTE:** You should not modify any function parameters or add/remove any function.
- The overall program will be called ***filesys*** and implement multiple operations described below. Your program will take command line arguments to control which operation is performed.
- All traversal-based functions should be done using **recursion**. A comment is added to each function that needs to be implemented using recursion. Please read the comment for each function carefully so that you don't lose points.
- We have already implemented ***Filesys.cpp***, ***mlist.h***, and ***os.h*** for you. Please refer to them, especially the header files to get an overview on how everything comes together.
- You should complete all functions of ***MList.cpp*** first, then ***OS.cpp***. Most of the functions are very similar to what you have done in Lab 11. You can refer to that lab to get a jump start on this assignment. However, there are several new functions that you need to implement especially the ones that should be done using recursion.

In ***MList.cpp***, you need to implement (refer to code comments in the skeleton on how to code them):

- ***Constructor***
- ***Destructor***
- void ***push_top***(Data* d_item);
- void ***pop_top***();
- void ***push_bottom***(Data* d_item);
- void ***pop_bottom***();
- bool ***isEmpty***();
- void ***clear***(); **//recursion**
- void ***sortSelection***(Node* start, bool mode); **//recursion - Selection sort**
- void ***sortInsertion***(Node* start); **//recursion - Insertion sort**
- void ***swapNode***(Node* a, Node* b);
- void ***deleteNode***(Node* a);
- void ***removeNode***(Node* a);
- void ***insertAfter***(Node* a, Node* b);
- Node* ***search***(Node* start, string name); **//recursion**
- void ***traverseToTop***(Node* n_item, string s); **//recursion**
- void ***traverseToBottom***(Node* n_item, string s);**//recursion**

In ***OS.cpp***, you need to implement (refer to code comments on how to code them):

- ***Constructor***
- ***Destructor***
- void ***file***(string fname, bool isRead);
- void ***del***(string fname);
- void ***ls***(string option);
- void ***create_item***(string fname, bool isFolder);
- void ***rm***(string fname);
- void ***cd_dir***(string fname);
- void ***present_working_dir***();
- Node* ***search_item***(string s);

5. Test your implementation

- Once you finished your *MList.cpp,* you can test your program by running:

./filesys test_mlist

This should output:

```
- BASIC CHECKING - MODE 1 -----
true
g
g
false
f
k
true
- PRINT CHECKING - MODE 2 -----
T-> c d(4) l <-B
B-> l d(4) c <-T
T-> b(6) e(19) d(4) m(10) g(0) f(7) <-B
B-> f(7) g(0) m(10) d(4) e(19) b(6) <-T
- SWAP AND INSERT AFTER CHECKING - MODE 3 -----
T-> c <-B
B-> c <-T
T-> e(19) d(4) <-B
B-> d(4) e(19) <-T
T-> e(19) g(0) m(10) d(4) <-B
B-> d(4) m(10) g(0) e(19) <-T
T-> m(10) e(19) g(0) d(4) <-B
B-> d(4) g(0) e(19) m(10) <-T
T-> m(10) g(0) d(4) e(19) <-B
B-> e(19) d(4) g(0) m(10) <-T
- SORT CHECKING - MODE 4 -----
T-> m(10) <-B
B-> m(10) <-T
T-> b(6) d(4) e(19) f(7) g(0) m(10) <-B
B-> m(10) g(0) f(7) e(19) d(4) b(6) <-T
T-> m(10) <-B
B-> m(10) <-T
T-> g(0) d(4) b(6) f(7) m(10) e(19) <-B
B-> e(19) m(10) f(7) b(6) d(4) g(0) <-T
T-> m(10) <-B
B-> m(10) <-T
T-> b(6) d(4) e(19) f(7) g(0) m(10) <-B
B-> m(10) g(0) f(7) e(19) d(4) b(6) <-T
- DELETE CHECKING - MODE 5 -----
false
true
false
true
T-> f(7) <-B
B-> f(7) <-T
true
T-> d(4) <-B
B-> d(4) <-T
T-> e(19) m(10) <-B
B-> m(10) e(19) <-T
true
- SEARCH CHECKING - MODE 6 -----
true
true
false
```

Or you can run **each section** by providing a third command line argument. In this example below we are running only section 4 (sort checking).

```
ccc_v1_t_QgTo_145769@runweb13:~$ ./filesys test_mlist 4
- SORT CHECKING - MODE 4 -----
T-> m(10) <-B
B-> m(10) <-T
T-> b(6) d(4) e(19) f(7) g(0) m(10) <-B
B-> m(10) g(0) f(7) e(19) d(4) b(6) <-T
T-> m(10) <-B
B-> m(10) <-T
T-> g(0) d(4) b(6) f(7) m(10) e(19) <-B
B-> e(19) m(10) f(7) b(6) d(4) g(0) <-T
T-> m(10) <-B
B-> m(10) <-T
T-> b(6) d(4) e(19) f(7) g(0) m(10) <-B
B-> m(10) g(0) f(7) e(19) d(4) b(6) <-T
```

./filesys test_mlist <option>

There are 7 sections:

- **0** = run all sections
- **1** = basic
- **2** = print
- **3** = swap
- **4** = sort
- **5** = delete
- **6** = search

- Once you finished your *OS.cpp,* you can test your program by running (**Note**: functions in OS.cpp depend on MList, you should finish MList first):

./filesys test_os

This should output:

```
ccc_v1_t_QgTo_145769@runweb14:~$ ./filesys test_os
- TEST BASIC 1 -----
true
Bonny Charlie Federico(0) Holl(0) Marie Paola(0) Shaunta Terrian(0)
Terrian(0) Shaunta Paola(0) Marie Holl(0) Federico(0) Charlie Bonny
Jarrod Reinaldo(0)
false
root/Marie/Shuan/Jarrod/
Reinaldo(0) Jarrod
root/Marie/Shuan/
root/
Bonny Charlie Federico(0) Holl(0) Marie Paola(0) Shaunta Terrian(0)
- TEST BASIC 2 -----
root/
root/
d(0) o
o(0)
root/o/o/o/o/o/
o(0) c(0) a
b c(0) o
Error: cannot find file or directory 'orange'
c(0)
```

Or you can run **each section** by providing a third command line argument.
Ex.

./filesys test_os 1
./filesys test_os 2

- Once everything is in place, you can try out your program (CS103 terminal) by running

./filesys terminal

- There are **10** commands that you can use on your terminal
    i. **cd** - change directory. This command takes 0 or 1 option. If no option or "~" is provided (e.g., ***cd ~***), you will be redirected to the root directory. If you want to move one level up, you can type in ***cd ..*** (2 dots). Or, if you want to go into a directory, simply cd [dirname] (e.g., ***cd Desktop***)

    ii. **pwd** - present working directory. This command will tell you the absolute path to your current directory, starting from the root directory (e.g., root/directory1/directory2/)

    iii. **ls** - list files and folders. This command takes 0 or 1 option. If no option is provided, it should print whatever your current directory has. If -r is provided (i.e., ***ls -r***), it will print the list in reverse order. You can add -sort=size and -sort=name to sort the list by size or name alphabetically (i.e., ***ls -sort=size*** and ***ls -sort=name***).

    iv. **mkfile [***filename***]** - create a file, only if the file or folder with the given name doesn't already exist in the current directory.

    v. **mkdir [***dirname***]** - create a folder, only if the file or folder with the given name doesn't already exist in the current directory

    vi. **write [***filename***]** - write a text to a file, if a file already has a text, it will overwrite it. The amount of text inside the file determines the file size. You cannot write to a file that doesn't exist or is a folder type.

vii. **read [***filename***]** - read a text from a file. Cannot read a file if it doesn't exist or is a folder type.

viii. **rm [***file or dir name***]** - delete a folder or file.

ix. **help -** shows a list of available commands

x. **exit or quit** - terminate our terminal program.

6. Note

- Unless you correctly implement your MList class, you will get a lot of errors, including segmentation faults. This is expected. Once you get everything set up, these errors should all be gone, hopefully!
- Use lots of diagrams to help reason about your code!!
- Use ***`valgrind`*** to check for memory leaks (as well as other problems such as uninitialized variables or out-of-bounds memory access).
- Your sort implementation will be checked manually by the graders. Please make sure you use a ***correct algorithm*** as well as using ***recursion***.
- There should be **at most 2** loops in your ***MList.cpp***

References:

[1] Linux System Administrator Guides. https://www.tldp.org/LDP/sag/html/filesystems.html