Warmup Assignment #1

(100 points total)

Doubly-linked Circular List in C

Due 11:45PM 9/9/2022 (firm)

Assignment

(Please check out the Warmup 1 FAQ before sending your questions to the TAs, the course producers, or the instructor.)

The main purpose of this assignment is to develop an **efficient** doubly-linked circular list from scratch in C.

Electronic submissions only.

For the first part of the assignment, you need to implement a doubly-linked circular list. You need to create `"my402list.c"` to work with `"my402list.h"`. You would also need `"cs402.h"`. You must **not** alter the files provided on this web page. For the meaning of the functions, please see function definition below. After you have successfully implemented the doubly-linked circular list, you must use it to implement the **`sort`** command specified below.

We will **not** go over the lecture slides for this assignment in class. Although it's important that you are familiar with it. Please read it over. If you have questions, please e-mail the **instructor**.

Compiling

Please use a `Makefile` so that when the grader simply enters:

```
make warmup1
```

an executable named **warmup1** is created. Please make sure that your submission conforms to other general compilation requirements and README requirements.

Commandline Syntax & Program Output

The commandline syntax (also known as "usage information") for **warmup1** is as follows:

```
warmup1 sort [tfile]
```

Square bracketed items are optional. If `tfile` is not specified, your program should read from `stdin`. Unless otherwise specified, output of your program must go to `stdout` and error messages must go to `stderr`.

The meaning of the commands are:

sort : Produce a sorted transaction history for the transaction records in **`tfile`** (or `stdin`) and compute balances. The input file should be in the **`tfile` format**.

The output for various commands are as follows.

sort : Your job is to read in a tfile one line at a time. For each line, you need to check if it has the correct format. If the line is malformed, you should print an error message and quit your program. Otherwise, you should convert the line into an internal object/data structure, and insert the object/data structure into a list, sorted by the timestamp. If there is another object/data structure with **identical** timestamp, you should print an error message and quit your program.

After all the input lines are processed, you should output all the transactions in ascending order, according to their timestamps. The output must conform to the following format (please do not print the first 3 lines below, they are only for illustration purposes):

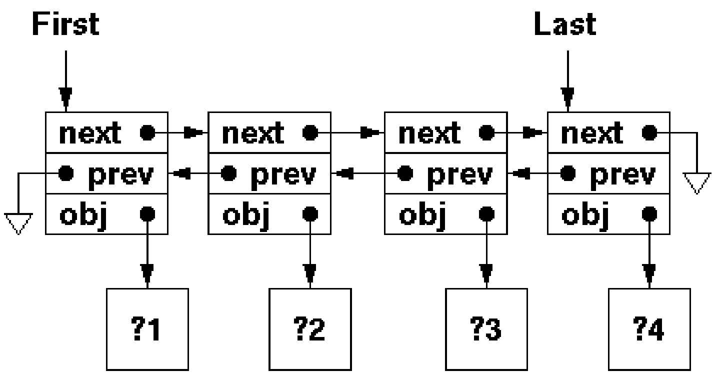
```
00000000011111111112222222222333333333344444444445555555555666666666677777777778
12345678901234567890123456789012345678901234567890123456789012345678901234567890

+-----------------+--------------------------+----------------+----------------+
|       Date      | Description              |         Amount |        Balance |
+-----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | ...                      |      1,723.00  |      1,723.00  |
| Wed Dec 31 2008 | ...                      | (       45.33) |      1,677.67  |
| Mon Jul 13 2009 | ...                      |     10,388.07  |     12,065.74  |
| Sun Jan 10 2010 | ...                      | (      654.32) |     11,411.42  |
+-----------------+--------------------------+----------------+----------------+
```

Each line is exactly 80 characters long (followed by a single `"\n"` character). The `Date` field spans characters 3 through 17. Please use `ctime()` to format the timestamp and remove unnecessary characters to make it look like what's in the table above. The `Description` field (shown as "`...`" above to mean whatever that's appropriate) spans characters 21 through 44. (If a description is too long, you must truncate it.) The `Amount` field spans characters 48 through 61. It must contain a decimal point with at least one digit to the left of the decimal point and exactly two digits to the right of the decimal point. If the number to the left of the decimal point is not 0, the first digit of this number must not be a 0. For a withdrawal, a pair of paranthesis must be used as indicated. If the amount of a transaction is more than or equal to 10 million, please print `?,???,???.??` (or `(?,???,???.??)`) in the `Amount` field to indicate that the amount is too large to fit in the display area. The `Balance` field spans characters 65 through 78. If a balance is negative, a pair of paranthesis must be used. If the absolute value of a balance is more than or equal to 10 million, please print `?,???,???.??` (or `(?,???,???.??)`) in the `Balance` field to indicate that the balance is too large to fit in the display area.

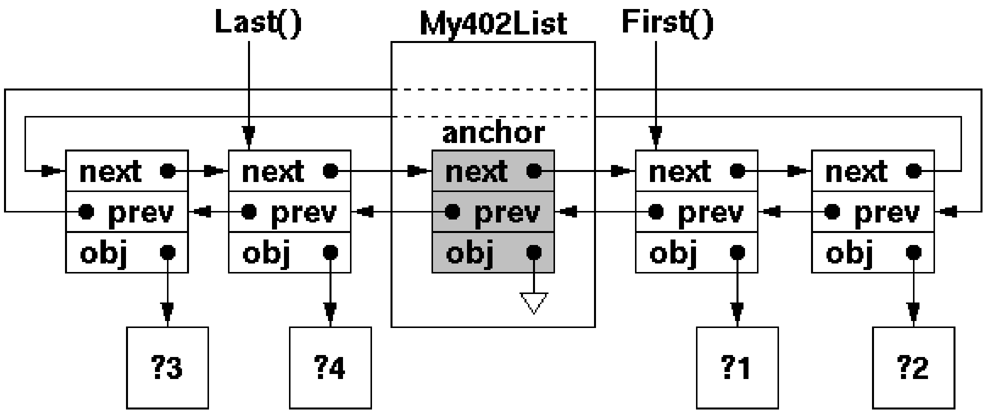
Pleaes output reasonable and useful error messages if the command is malformed or file does not exist or inaccessible.

My402List

A traditional doubly-linked list of length 4 looks like the following:



A corresponding `My402List` would look like the following:



The functions you need to implement has the following meaning (note that, for readability, all the functions are missing "`My402List`" before the name, and all of them are missing `(My402List*)` as the first argument as compare to what's in the actual header file, "`my402list.h`"; furthermore, other than the `Init()` function, you must assume that for all other functions, the first argument is a **valid list**; for the `Init()` function, you must assume that the first argument points to `sizeof(My402List)` bytes of valid memory):

`int Length()`
Returns the number of elements in the list.

`int Empty()`
Returns **TRUE** if the list is empty. Returns **FALSE** otherwise.

`int Append(void *obj)`
If list is empty, just add **`obj`** to the list. Otherwise, add **`obj`** after **`Last()`**. This function returns **TRUE** if the operation is performed successfully and returns **FALSE** otherwise.

`int Prepend(void *obj)`
If list is empty, just add **`obj`** to the list. Otherwise, add **`obj`** before **`First()`**. This function returns **TRUE** if the operation is performed successfully and returns **FALSE** otherwise.

`void Unlink(My402ListElem *elem)`
Unlink and `free()` **`elem`** from the list. Please do not `free()` the object pointed to by **`elem`** and do not check if `elem` is on the list.

`void UnlinkAll()`
Unlink and `free()` all elements from the list and make the list empty. Please do not `free()` the objects pointed to by the list elements.

`int InsertBefore(void *obj, My402ListElem *elem)`
Insert **`obj`** between **`elem`** and **`elem->prev`**. If **`elem`** is **`NULL`**, then this is the same as **`Prepend()`**. This function returns **TRUE** if the operation is performed successfully and returns **FALSE** otherwise. Please do not check if `elem` is on the list.

`int InsertAfter(void *obj, My402ListElem *elem)`
Insert **`obj`** between **`elem`** and **`elem->next`**. If **`elem`** is **`NULL`**, then this is the same as **`Append()`**. This function returns **TRUE** if the operation is performed successfully and returns **FALSE** otherwise. Please do not check if `elem` is on the list.

`My402ListElem *First()`
Returns the first list element or **`NULL`** if the list is empty.

`My402ListElem *Last()`
Returns the last list element or **`NULL`** if the list is empty.

`My402ListElem *Next(My402ListElem *elem)`
Returns **`elem->next`** or **`NULL`** if **`elem`** is the last item on the list. Please do not check if `elem` is on the list.

`My402ListElem *Prev(My402ListElem *elem)`
Returns **`elem->prev`** or **`NULL`** if **`elem`** is the first item on the list. Please do not check if `elem` is on the list.

`My402ListElem *Find(void *obj)`
Returns the list element **`elem`** such that **`elem->obj`** == **`obj`**. Returns **`NULL`** if no such element can be found.

`int Init()`

Initialize the list into an empty list. Returns **TRUE** if all is well and returns **FALSE** if there is an error initializing the list.

Assuming that you have a list of `(Foo*)` objects, a typical way to traverse the list from first to last is as follows:

```
void Traverse(My402List *list)
{
    My402ListElem *elem=NULL;

    for (elem=My402ListFirst(list);
            elem != NULL;
            elem=My402ListNext(list, elem)) {
        Foo *foo=(Foo*)(elem->obj);

        /* access foo here */
    }
}
```

Your implementation of `My402List` must allow your list to be traversed in the above way. Please use `listtest` to verify that your implementation is correct. Please understand that it is **your job to read and understand** the code in "`listtest.c`" and figure out how you must write the code in "`my402list.c`" so that `listtest` compiles and runs perfectly.

If you are not familiar with pointers in C, please take a look at my review on pointers.

`tfile` Format

A **`tfile`** (transaction file) is an ASCII text file. Each line in a **`tfile`** contains 4 string fields with <TAB> characters being the delimeters (i.e, each line contains exactly 3 <TAB> characters.) Each line (includingn the last line) in a `tfile` must end with a "`\n`" character. As a result, the last character of a valid `tfile` must always be a "`\n`" character. The fields are:

- Transaction type (single character: "`+`" for deposit or "`-`" for withdrawal).
- Transaction time (a UNIX timestamp, please see `man -s 2 time` on your 32-bit Ubuntu 16.04 system). The value of this field must be > 0 and < the timestamp that correspond to the current time. Also, since this is a number, the first digit must not be zero. (Since the largest unsigned integer is 4,294,967,295, if the length of the string of this field is more than or equal to 11, you can safely assume that the timestamp is bad.)
- Transaction amount (a number followed by a period followed by two digits; if the number is not zero, its first digit must not be zero). The number to the left of the decimal point can be at most 7 digits (i.e., < 10,000,000). The transaction amount must have a positive value.
- Transaction description (textual description, cannot be empty). A description may contain leading space characters, but you must remove them before proceeding. After leading space characters have been removed, a transaction description must not be empty.

The lines are not sorted in any order. Furthermore, if a line is longer than 1,024 characters (including the '\n' at the end of a line), it is considered an error.

If you encounter an error when you process the input file, you should print an error message and quit your program. You must not process additional input lines. Please also note that a valid file must contain at least one transaction.

A sample `tfile` is provided here as test.tfile.

Testing Your Doubly-linked Circular List

To make sure that your implementation of the doubly-linked circular list is correct, we have provided a test program, `listtest.c` and a corresponding `Makefile`:

- `listtest.c`
- `Makefile`

Put these files together with your implementation of `my402list.c` and the provided `my402list.h` and `cs402.h` and type "`make`". You should get an executable named **`listtest`**.

If you do:

```
./listtest
```

no output must be produced. You can also run:

```
./listtest -debug
```

to have the program output some debugging information.

Sample Printout

If you download the sample `tfile`, i.e., test.tfile and run:

```
./warmup1 sort test.file
```

You should get the following printout (indented so it's easier to read):

```
+-----------------+--------------------------+----------------+----------------+
|       Date      | Description              |         Amount |        Balance |
+-----------------+--------------------------+----------------+----------------+
| Thu Aug 21 2008 | Initial deposit          |      1,723.00  |      1,723.00  |
| Wed Dec 31 2008 | Phone bill               | (       45.33) |      1,677.67  |
| Mon Jul 13 2009 | Dear parents             |     10,388.07  |     12,065.74  |
| Sun Jan 10 2010 | Beemer monthly payment   | (      654.32) |     11,411.42  |
+-----------------+--------------------------+----------------+----------------+
```

The above printout is provided in test.tfile.out (648 bytes in size). Please understand that your printout must be **identical** to test.tfile.out, down to every byte. Also, please be careful when you download test.tfile.out with a web browser. Your web browser may **modify** this file! For example, if you are on Windows, since Windows text files are different from Unix text files, Windows may add invisible characters to this file to make it into a Windows text file. Therefore, you should inspect the file size after you have it downloaded. If the file size is not 648 bytes, then you need to dowload it using a different method (for example, right click on test.tfile.out and select "Save Link As").

Grading Guidelines

The grading guidelines and grading data (in `w1data.tar.gz`) have been made available. Please understand that the grading guidelines is **part of the spec** and the grader will stick to it when grading. Please run the scripts in the guidelines on a standard 32-bit Ubuntu 16.04 system. You should read the scripts to understand exactly how your assignment will be graded. It is possible that there are bugs in the guidelines. If you find bugs, please let the instructor know as soon as possible.

The grading guidelines is the **only** grading procedure we will use to grade your program. No other grading procedure will be used. To the best of our effort, we will only **change** the **testing data** for grading but not the commands. (We may make minor changes if we discover bugs in the script or things that we forgot to test.) It is strongly recommended that you run your code through the scripts in the grading guidelines.

A copy of the grading scripts are made available here (the `.out` files are what the output suppose to look like):

```
section-A.csh         section-A.out
section-B.csh         section-B.out
```

After you have download the above "`.csh`" files, please put them in the same directory as `warmup1` and run the following command:

```
chmod 755 section-?.csh
```

Please also follow the beginning part of the grading guidelines to unpack the grading data file (i.e., `w1data.tar.gz`) so that the script can work properly.

By the way, please do not run these grading scripts in a shared folder. For unknown reasons, running the grading scripts from within a shared folder may not work correctly!

Miscellaneous Requirements and Hints

- Please read the general programming FAQ if you need a refresher on file I/O and bit/byte manipulications in C.
- You must **NOT use any external code segments** to implement this assignment. You must implement all these functionalities from scratch.
- You must **not use any arrays** to implement list functionalities. You must dynamically allocate all elements in a list.
- For the **`sort`** command, you must use the doubly-linked circular list developed in this assignment.
- If the size of the input file is large, you **must not** read the whole file into a large memory buffer and then process the file data. You must read the file **incrementally**.

- It's important that **every byte** of your data is read and written correctly. You will **lose a lot of points** if one byte of data is generated incorrectly! The grading of this assignment will be **harsh** and you must make your code to work according to the posted grading guidelines.

- Please follow the UNIX convention that, when your output is an ASCII file (such as the output of the `sort` command), append '\n' in the last line of the output if it's not a blank line. (This way, you don't get the commandline prompt appearing at the wrong place on the screen.)

- String I/O functions such as `fgets()`, `scanf()`, and `printf()` are really meant for inputing/outputing *strings*. Do **not** use them to input/output binary data! Do **not** use them to input/output binary data (unless you are sure what you are doing)!

- Start working on this **early**! Please don't complain to the instructor that this assignment is too tedious or it takes too much work just to parse the commandline. Get it done early and get it done right!

Submission

All assignments are to be submitted electronically (including the required "README" file). To submit your work, you must first `tar` all the files you want to submit into a "tarball" and `gzip` it to create a **gzipped tarfile** named **`warmup1.tar.gz`**. Then you upload **`warmup1.tar.gz`** to the Bistro system. The command you can use to create a gzipped tarfile is:

```
    tar cvzf warmup1.tar.gz MYLISTOFFILES
    ls -l warmup1.tar.gz
```

Where `MYLISTOFFILES` is a list of file names that you are submitting (you can also use wildcard characters if you are sure that it will pick up only the right files). **DO NOT** submit your compiled code, just your source code and README file. **Two point will be deducted** for each **binary file** included in your submission (e.g., `warmup1`, `.o`, `.gch`, `core`, etc.). The last command shows you how big the created "`warmup1.tar.gz`" file is. If "`warmup1.tar.gz`" is larger than 1MB in size, the submission server will not accept it.

Please note that the 2nd commandline argument of the `tar` command above is the **output** filename of the `tar` command. So, if you omit `warmup1.tar.gz` above, you may accidentally replace one of your files with the output of the `tar` command and there is no way to recover the lost file (unless you have made a backup copy). So, please make sure that the first commandline argument is **`cvzf`** and the 2nd commandline argument is **`warmup1.tar.gz`**.

A `w1-README.txt` template file is provided here. You must save it as your `w1-README.txt` file and follow the instructions in it to fill it out with appropriate information and include it in your submission. You must not delete a single line from `w1-README.txt`. Please make sure that you satisfy all the README requirements.

Here is an example commands for creating your `warmup1.tar.gz` file:

```
    tar cvzf warmup1.tar.gz Makefile *.c *.h w1-README.txt
```

If you use an IDE, you need to modify the commands above so that you include **ALL** the necessary source files and subdirectories and make sure you have not forgotten to include a necessary file in order for the grader to compile your code on a "clean" system.

You should read the output of the above commands carefully to make sure that `warmup1.tar.gz` is created properly. If you don't understand the output of the above commands, you need to learn how to read it! It's your responsibility to ensure that `warmup1.tar.gz` is created properly.

To check the content of `warmup1.tar.gz`, you can use the following command:

```
    tar tvzf warmup1.tar.gz
```

Please read the output of the above command carefully to see what files were included in `warmup1.tar.gz` and what are their file sizes and make sure that they make sense.

Please enter your **USC e-mail address** and your **submission PIN** below. Then click on the **Browse** button and locate and select your submission file (i.e., `warmup1.tar.gz`). Then click on the **Upload** button to submit your `warmup1.tar.gz`. (Be careful what you click! Do **NOT** submit the wrong file!) If you see an error message, please read the dialogbox carefully and fix what needs to be fixed and repeat the procedure. If you don't know your submission PIN, please visit this web site to have your PIN e-mailed to your USC e-mail address.

When this web page was **last loaded**, the time at the submission server (**merlot.usc.edu**) was **30Aug2022-15:57:53**. **Reload** this web page to see the **current time** on **merlot.usc.edu**.

USC E-mail:	**`@usc.edu`**
Submission PIN:	

Event ID (read-only):	merlot.usc.edu_80_1557931083_236
Submission File Full Path:	Choose File No file chosen
	Upload

If the command is executed successfully and if everything checks out, a **ticket** will be issued to you to let you know "what" and "when" your submission made it to the Bistro server. The next web page you see would display such a ticket and the ticket should look like the sample shown in the submission web page (of course, the actual text would be different, but the format should be similar). Make sure you follow the Verify Your Ticket instructions to verify the SHA1 hash of your submission to make sure what you did not accidentally submit the wrong file. Also, an e-mail (showing the ticket) will be sent to your USC e-mail address. Please read the ticket carefully to know exactly "what" and "when" your submission made it to the Bistro server. If there are problems, please contact the instructor.

It is extreme important that you also **verify your submission** after you have submitted `warmup1.tar.gz` electronically to make sure that every you have submitted is everything you wanted us to grade. If you don't **verify your submission** and you ended up submit the wrong files, please understand that due to our fairness policy, there's absolutely nothing we can do.

Finally, please be familiar with the Electronic Submission Guidelines and information on the bsubmit web page.