Summer 2014 CSCI 402

Warmup Assignment #1

(100 points total)

Doubly-linked Circular List in C

Due 11:45PM 6/5/2014 (firm)

(Please check out the FAQ before sending your questions to the TA or the instructor.)

Assignment

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 youare 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 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 transcations 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):

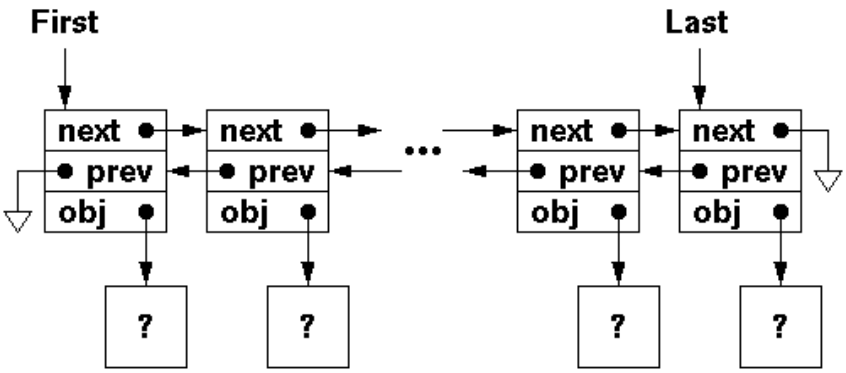
```
0000000001111111111222222222233333333334444444444555555555566666666667777777778
1234567890123456789012345678901234567890123456789012345678901234567890123456789

+-----------------+--------------------------+----------------+----------------+
|       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 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. 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. 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.

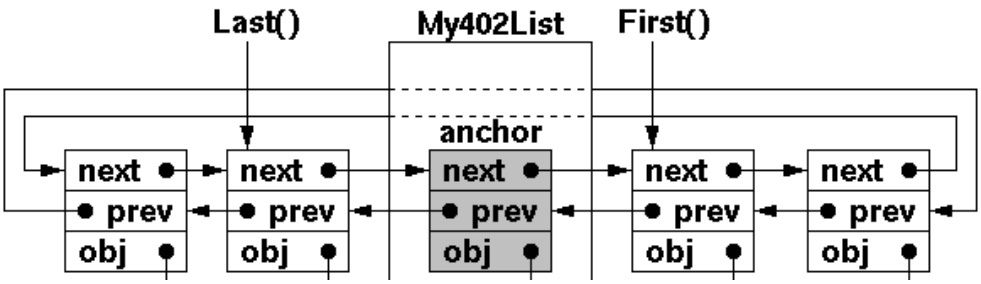
Pleaes output reasonable and useful error messages if the command is malformed or file does not exist or inaccessible.

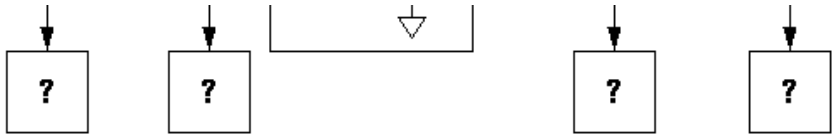
My402List

A traditional doubly-linked list looks like the following:



A corresponding `My402List` would look like the following:





The functions you need to implement has the following meaning (note that all the functions are missing "`My402List`" before the name and `(My402List*)` as the first argument as compare to the actual implementation in the actual header file, "`my402list.h`"):

`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)`
[BC: updated 05/25/14] 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)`
[BC: updated 05/25/14] 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 delete **`elem`** from the list. Please do not delete the object pointed to by **`elem`** and do not check if `elem` is on the list.

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

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 `My420List` must allow your list to be traversed in the above way. Please use `listtest` to verify that your implementation is correct.

`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.) The fields are:

- Transcation type (single character: `"+"` for deposit or `"-"` for withdrawal).
- Transcation time (a UNIX timestamp, please see `man -s 2 time` on `nunki.usc.edu`). The value of this field must be between 0 and the timestamp that correspond to the current time. (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). The number to the left of the decimal point can be at most 7 digits (i.e., < 10,000,000). The transcation amount must have a positive value.
- Transcation 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.

Grading Guidelines

The grading guidelines has been made available. Please run the scripts in the guidelines on

The grading guidelines has been made available. Please run the scripts in the guidelines on `nunki.usc.edu` or `aludra.usc.edu` (after the grading account is setup properly). For now, 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.

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!
- The Solaris workstations in the ISD lab in SAL have the same setup as nunki.usc.edu. So, if you are logged on to one of these workstations, please do your development locally and not to overload nunki unnecessarily.
- 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 your 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. On `nunki.usc.edu` or `aludra.usc.edu`, the command you can use to create a gzipped tarfile is:

```
/usr/usc/bin/gtar cvzf warmup1.tar.gz MYFILES
```

Where **`MYFILES`** is the 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** if you submit extra binary files, such as `warmup1`, `.o`, `core`, or files that can be **generated** from the rest of your submission.

Please note that the 2nd commandline argument of the `gtar` command above is the **output** filename of the `gtar` command. So, if you omit `warmup1.tar.gz` above, you may accidentally replace one of your files with the output of the `gtar` command. So, please make sure that the first commandline argument is **`cvzf`** and the 2nd commandline argument is **`warmup1.tar.gz`**.

A README template file is provided here. You should save it as your `README` file and fill it out with your documentation information before you create your submission file.

Here is a sample command for creating your `warmup1.tar.gz` file (your command will vary depending on what files you want to submit):

```
/usr/usc/bin/gtar cvzf warmup1.tar.gz *.c *.h Makefile README
```

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.

[BC: paragraph added 05/26/14]
For this assignment, to make sure that you did not modify the provided "`my402list.h`", "`cs402.h`", and "`listtest.c`" files, if you include them in your submission, they will be discarded. We will use our copy of these files (please read the grading guidelines for details) when we grade your assignment. (If you include them and they are identical to what's in the spec, it's perfectly fine.)

You need to run **`bsubmit`** to submit warmup1.tar.gz to the submission server. Please use the following command:

```
[BC: fixed 05/26/14]
~csci551b/bin/bsubmit upload \
  -email `whoami`@usc.edu \
  -event merlot.usc.edu_80_1372906710_42 \
  -file warmup1.tar.gz
```

Please note that the quotation marks surrounding `whoami` are **back-quote** characters and not single quotes. It's best if you just copy and paste the above command into your console and not try to type the whole command in.

If the command is executed successfully, the output should look like the sample mentioned in the submission web page. If it doesn't look like that, please fix your command and rerun it until it looks right. 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 everything you have submitted is everything you wanted us to grade.

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

[*Last updated Thu May 29 2014*] [*Please see copyright regarding copying.*]