COMP9024 18s2 Data Structures and Algorithms

Assignment 1
Transport Card Manager

Change Log

We may make minor changes to the spec to address/clarify some outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the change log regularly.

Version 1: Released on 15 August 2018

Objectives

The assignment aims to give you more experience with

- dynamic data structures, specifically dynamic arrays and linked lists
- implementing (abstract) data types
- asymptotic runtime analysis

Admin

Marks	4 marks for stage 1 (correctness) 2 marks for stage 2 (correctness) 2 marks for stage 3 (correctness) 1 mark for complexity analysis (stages 2 & 3 only) 1 mark for style ——————— Total: 10 marks
Due	23:59 on **Monday** 27 August (week 6)
Late	1.5 marks (15%) off the ceiling per day late (e.g. if you are 25 hours late, your maximum possible mark is 7)

Aim

Your task is to write a program for reading and processing a collection of transport card records (stage 1) and for maintaining them (stages 2 & 3). You will have to complete wrapper code to test your implementations, and make sure to test a variety of conditions. You will also have to provide the time complexity of your functions for maintaining the card records.

Provided Code

A *zip file* is a way of compressing and packaging files. We have provided a zip file as a base for you to begin the assignment. You can download it here: `starterCode.zip`

When you `unzip` the package, you will find six files in your current directory:

- `main.c`: The incomplete main program.
- `cardRecord.h`: A header file that lists the functions in the card record implementation. *Do not change.*
- `cardRecord.c`: An incomplete library of functions for reading and writing single transport card records.
- `cardLL.h`: A header file that lists the functions in the linked list implementation. *Do not change.*

- `cardLL.c`: An incomplete library of functions for maintaining a linked list of transport card records.
- `Makefile`: Used to make an executable `main` from the main program and the two modules.

Stage 1 – Dynamic Array (4 marks)

For stage 1, you will need to implement functions in `cardRecord.c` and write code in `main.c` for a program that

1. accepts a positive number *n* as a single command line argument
2. uses a ***dynamic array on the heap*** to store *n* transport card records from user input
3. prints all card records
4. prints some statistical information

For each card, your program should prompt the user to

- "`Enter card ID: `" (expecting an 8-digit number, whose first digit must be non-zero)
- "`Enter amount: `" (expecting a floating point number between -2.35, the minimim balance, and 250, the maximum balance)

Your program should detect any invalid input and, where necessary, ask the user to re-enter data with the message "`Not valid. Enter a valid value: `".

The output should be as follows:

- for each card,
  - an opening line "`----------`"
  - a line "`Card ID: `" followed by the ID
  - a line "`Balance: `" followed by the balance on the card displayed as positive or negative $-value
  - a line "`Low balance`" if the balance is below $5.00
  - a closing line "`----------`"
- a line "`Number of cards on file: `" followed by the number of cards
- a line "`Average balance: `" followed by the average balance across all transport cards

Finally, you must ensure that all dynamically allocated memory has been freed upon termination of your program.

Example

Here is an example to show the desired behaviour of your program for stage 1:

```
prompt$ ./main 2
Enter card ID: 123456
Not valid. Enter a valid value: 12345678
Enter amount: 100
Enter card ID: 11111111
Enter amount: -5
Not valid. Enter a valid value: -2.3
----------
Card ID: 12345678
Balance: $100.00
----------
----------
Card ID: 11111111
Balance: -$2.30
Low balance
----------
Number of cards on file: 2
Average balance: $48.85
```

Note:

- Card records are printed in the order in which they have been entered.
- You may assume that no card ID is entered twice.
- All $-amounts get rounded to second decimal place when printed.
- Negative $-amounts should always be printed exactly as shown in the example.

Stage 2 – Linked List (2 marks)

Aim: For stage 2, you will implement most of the functions in `cardLL.c`, analyse the time complexity of your implementation and add code to `main.c`.

When you run the main program without command line argument, you will be presented with these options:

```
prompt$ ./main
Enter command (a,g,p,q,r, h for Help)>
```

If you type **h** for Help, you will see a more detailed list of commands:

```
  a - Add card record
  g - Get average balance
  h - Help
  p - Print all card records
  r - Remove card
  q - Quit
```

When you type **q** the program exits.

For this stage, you are to use a ***dynamic linked list*** to implement the following commands:

1. a – prompts the user to input valid data for a transport card record as above, inserts the record into the linked list and outputs `"Card added."`
2. g – outputs the total number of card records and the average balance across all transport cards in the list
3. p – prints all transport card records

4. Your program should include a time complexity analysis, in Big-Oh notation, for your functions in `cardLL.c`:
   - The size *n* of the input is given by the length of the linked list (i.e. the number of card records).
   - Each function should be preceded by two comment lines:

     ```
     // Time complexity: O(...)
     // Explanation: ...
     ```

   - You only need to provide time complexity for the functions that are also defined in `cardLL.h`.

5. You must ensure that all dynamically allocated memory is properly freed.

Example

Here is an example to show the desired behaviour of your program for stage 2:

```
prompt$ ./main
Enter command (a,g,p,q,r, h for Help)> a
Enter card ID: 33333333
Enter amount: -1
Card added.
Enter command (a,g,p,q,r, h for Help)> a
Enter card ID: 11111111
Enter amount: 41
Card added.
Enter command (a,g,p,q,r, h for Help)> g
Number of cards on file: 2
Average balance: $20.00
Enter command (a,g,p,q,r, h for Help)> p
```

```
-----------------
Card ID: 11111111
Balance: -$1.00
Low balance
-----------------
-----------------
Card ID: 33333333
Balance: $41.00
-----------------
Enter command (a,g,p,q,r, h for Help)> q
Bye.
```

Note:

- As before, you can assume that all card IDs will be different.
- For stage 2, new records should always be added at the *beginning* of the list
  - so they are displayed in reverse order when printed.

Stage 3 – Ordered Linked List (2 marks)

Aim: For stage 3, you will modify and complete the implementation in `cardLL.c` and also complete `main.c`:

1. Extend the functionality of the command `a` (adding a record):
   - *New* cards should now be inserted into the list in *ascending order* of the card ID.
   - *Existing* cards should have their balance updated by the specified transaction amount, and the updated card record should be printed.
2. Implement the command `r` to
   - prompt the user to input an 8-digit card ID
   - remove the card record if it is in the list, and otherwise print the message `"Card not found."`.
3. Your program `cardLL.c` should include a time complexity analysis, in Big-Oh notation, for all your functions that are also defined in `cardLL.h`.
4. You must ensure that all dynamically allocated memory is properly freed.

Example

Here is an example to show the desired behaviour of your program for stage 3:

```
prompt$ ./main
Enter command (a,g,p,q,r, h for Help)> a
Enter card ID: 11111111
Enter amount: 50
Card added.
Enter command (a,g,p,q,r, h for Help)> r
Enter card ID: 33333333
Card not found.
Enter command (a,g,p,q,r, h for Help)> a
Enter card ID: 33333333
Enter amount: 5
Card added.
Enter command (a,g,p,q,r, h for Help)> p
------------------------
Card ID: 11111111
Balance: $50.00
------------------------
------------------------
Card ID: 33333333
Balance: $5.00
------------------------
Enter command (a,g,p,q,r, h for Help)> a
Enter card ID: 33333333
Enter amount: -1.10
------------------------
```

```
Card ID: 33333333
Balance: $3.90
Low balance
-----------------------
Enter command (a,g,p,q,r, h for Help)> g
Number of cards on file: 2
Average balance: $26.95
Enter command (a,g,p,q,r, h for Help)> r
Enter card ID: 33333333
Card removed.
Enter command (a,g,p,q,r, h for Help)> g
Number of cards on file: 1
Average balance: $50.00
Enter command (a,g,p,q,r, h for Help)> q
Bye.
```

Testing

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program for the corresponding assignment, i.e.* `assn1`*. It expects to find the programs* `main.c, cardRecord.c, cardLL.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun assn1
```

Please note: Passing the dryrun tests does not guarantee that your program is correct. You should thoroughly test your program with your own test cases.

Submit

Note: Before you submit, you must ensure that your program compiles without errors or warnings on a CSE machine using the compiler options `-Wall -Werror -std=c11`

For this project you will only need to submit program files (.c files). You can either submit through WebCMS3 or use the submit command

```
prompt$ give cs9024 assn1 main.c cardRecord.c cardLL.c
```

Do not forget to add the time complexity to your source code file `cardLL.c`.

You can submit as many times as you like — later submissions will overwrite earlier ones. You can check that your submission has been received on WebCMS3 or by using the following command:

```
prompt$ 9024 classrun -check assn1
```

Marking

This project will be marked on functionality in the first instance, so it is very important that the output of your program be ***exactly*** correct as shown in the examples above. Submissions which score very low on the automarking will be looked at by a human and may receive a few marks, provided the code is well-structured and commented.

Programs that generate compilation errors will receive a very low mark, no matter what other virtues they may have. In general, a program that attempts a substantial part of the job and does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

Style considerations include:

- readability
- structured programming
- good commenting

The main objective of this assignment is to give you more experience with dynamic data structures:

- You will receive 0 marks for stage 1 if your program is not using the heap for a dynamic array, and 0 marks for stage 2 & 3 if your program is not using a dynamic linked list.
- You will also lose marks if your program has not freed all dynamically allocated memory upon termination.

Plagiarism

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar projects in previous years, if applicable) and serious penalties will be applied, particularly in the case of repeat offences.

- ***Do not copy ideas or code from others***
- ***Do not use a publicly accessible repository or allow anyone to see your code, not even after the deadline***

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Policy Statement
- UNSW Plagiarism Procedure

Help

See FAQ for some hints on how to get started and for troubleshooting tips.

Finally …

Have fun! Michael