Lab 4: Queue Management at a Grocery Store

1. Objectives

In this lab, you will be creating a computer simulation of a queue management system for a grocery store. The simulation will imitate customers, checkout registers and queues on checkout registers in a grocery store. The user will enter commands into the simulation to create customers, open and close checkout registers. Registers and customers will be added to queues implemented using linked list data structures. You will simulate customers in a “single” queue and in “multiple” queues.

2. Problem Statement

This assignment wants you to simulate different queue management strategies in a grocery store. Instead of testing out different queue management strategies in a real grocery store, we will build a computer simulation to test out different strategies.

2.1. Parser implementation

Initially, you are given starter code that **parses commands from the user**. In main.cpp, you will find

- `getMode()` function that gets the mode of the queueing system from the user: either single or multiple,
- `parseRegisterAction()`, `openRegister()` and `closeRegister()` functions that are used to parse the commands from the user that open and close registers,
- `addCustomer()` function that parses the commands, which adds customers to the queues,
- `getInt()` function that parses integers,
- `getDouble()` function that parses doubles, and
- `foundMoreArgs()` function that is used to detect whether more arguments were passed to the command than expected.

The starter code accepts commands in the following format.

- **Open a register:** register open <ID> <secPerItem> <setupTime> <timeElapsed>
- **Close a register:** register close <ID> <timeElapsed>
- **Add a customer:** customer <items> <timeElapsed>

<ID> is the ID of the register, <secPerItem> is the number of seconds the register takes to process one item, <setupTime> is the set up time of each register per customer, <items> is the number of items each customer has and <timeElapsed> is the number of seconds elapsed since the last command/action. For a user operating the simulation, the time between entering two commands can be a few seconds, but we can use <timeElapsed> to *simulate* several seconds, minutes or hours between two actions in the grocery store.

The starter code already checks for invalid inputs, too many or too few arguments and you aren't required to change it.

2.2. Objects in a Grocery Store

You will implement linked list based data structures to organize the objects in the grocery store.

To do so, you will implement four classes: `Customer`, `QueueList`, `Register` and `RegisterList`.

2.3. System Implementation

Users of the program can use the command line to open and close registers and add customers to queues. To organize the registers and customers, we will use linked list data structures. Initially, there is only one empty linked list of registers named `registerList`, which is a pointer to `RegisterList`. The instance of `RegisterList` has a member called `head`, which is initially set to NULL, but should later point to the first register in the linked list of registers.

A `register open` command would create a `Register` object. Each `Register` object stores the register's ID `ID`, speed at which the register processes items `secPerItem`, set up time for each customer `overheadPerCustomer` and the latest time the register was available `availableTime`. **The time at which the register opens or when they depart a customer is when they become available.** Since each instance of `Register` is a node in the linked list of registers, `Register` has a member called `next` that points to the next register in the linked list. Each `Register` object has `queue` of type `QueueList*` that is pointing to a linked list data structure holding customers to be served. Initially, when a register opens, `queue` should point to a **new** instance of `QueueList`. This new instance of `QueueList` has a member called `head` that should point to the customer being served at the head of the queue. Check Register.h in lab4_release.zip to see the noted data members.

A customer command would create a `Customer` object. Each `Customer` object stores the customer arrival time `arrivalTime`, number of items the customer has `numOfItems` and the departure time `departureTime`. Initially, the departure time is set to −1, and it gets updated with the true departure time when the customer departs. Check Customer.h in lab4_release.zip to see the noted data members.

When a customer arrives, it joins a queue. If the mode of the simulation is "multiple", the customer joins the queue `QueueList` of the register with least number of items. If several registers have the same least number of items, the customer should join the queue `QueueList` of the closest register to

the head of the register list `RegisterList`. Once a customer joins a queue, it cannot switch to another queue.

If the mode of the simulation is “single”, the customer becomes the head of the queue of the free register to get served. If there were several free registers, the customer joins the queue of the closest register to the head of the register list. If there were no free registers, then the customer joins the only **one** queue of customers waiting to be served named `singleQueue` in main.cpp. This is a linked list data structure of type `QueueList`.

A register closes with `register close` command. Regardless of the mode of the simulation, you can assume **a register will not close while it’s serving a customer or if it has a queue of customers**. This means you do not have to worry about where customers queued at a register will go if it closes. Once a register closes, we have to `delete` it from the linked list data structure of `RegisterList` instance holding all registers in the simulation.

2.4. System Update

Every time a command is entered, apart from adding a customer, opening or closing a register, the system time has to be updated using `expTimeElapsed` in main.cpp. The system time stored in `expTimeElapsed` starts from 0, and it gets incremented with the elapsed time every command is entered. Along with the update in time, each **register has to be checked**.

If the time taken to process the customer at the head of the queue of a register has passed, the customer has to depart. The time taken to process a customer is `secPerItem` of the register × `items` of the customer + `setupTime` of the register. The departure time would have to be carefully calculated. If the customer at the **head** of the queue arrived after the register became available, then the departure time is the processing time of the customer plus the arrival time of the customer. Otherwise, the departure time is the processing time of the customer plus the time the register became available. This has to be calculated in `calculateDepartTime` function in Register.h.

Once it is time for a customer to depart, its departure time is updated, and the customer should leave the queue and get added to a linked list of customers that exited the system `QueueList *doneList` in main.cpp. This linked list would be of help when we calculate some statistics about the wait time of the simulation.

If the simulation mode is “single”, as soon as a register departs a customer, another customer from the single queue should join this free register. **Hint: We have to depart customers in order of their departure times, NOT by the order of registers in the linked list, to ensure customers get served by the correct registers.** `calculateMinDepartTimeRegister` function in RegisterList class returns a pointer to the register that should depart a customer first. **Also, the elapsed time can be enough to depart more than one customer, so it is important to check that we have departed enough customers from each register.**

Multiple Queues

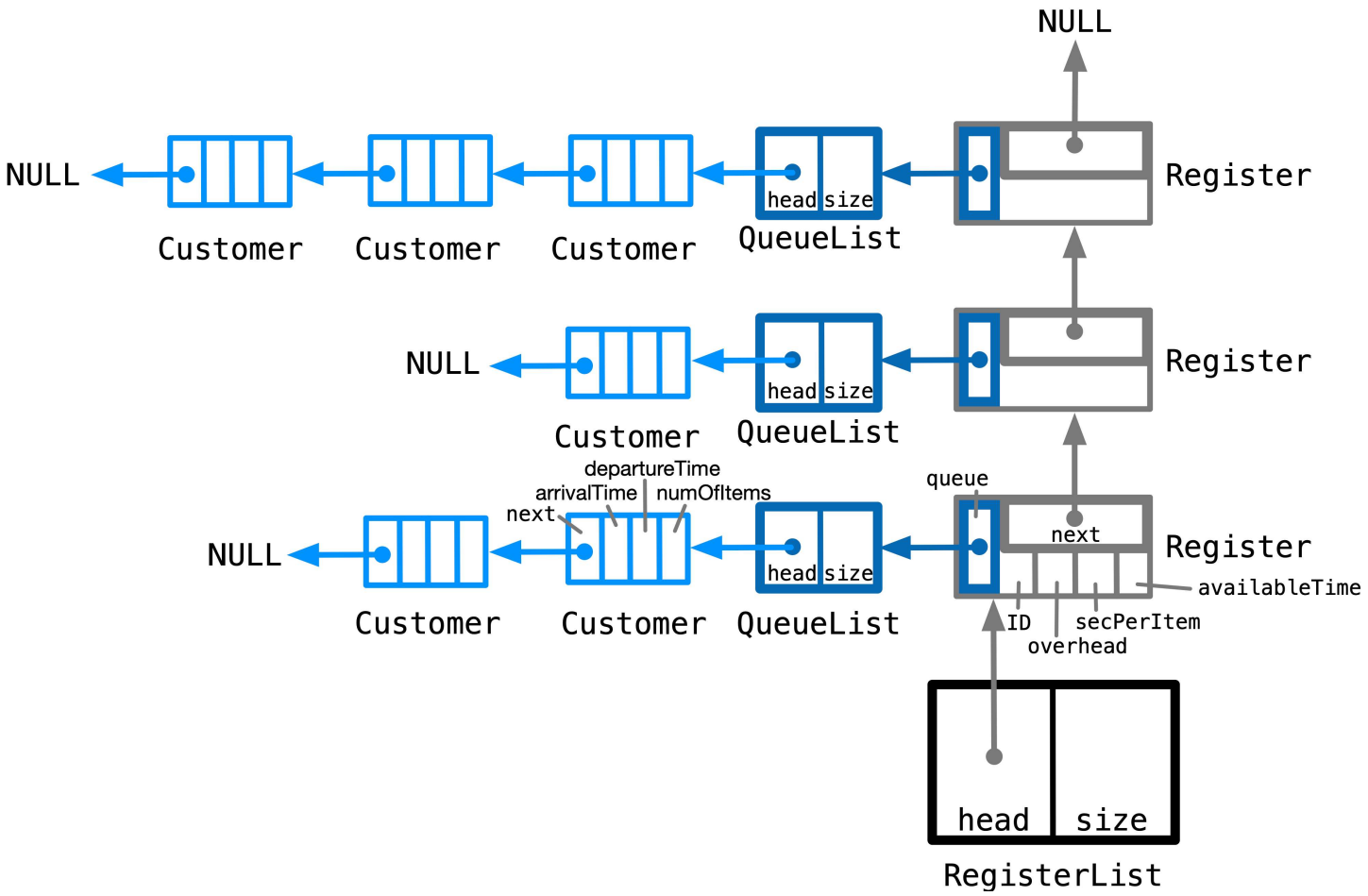

Figure 1: A snapshot of the grocery store adopting multiple queues

Single Queue

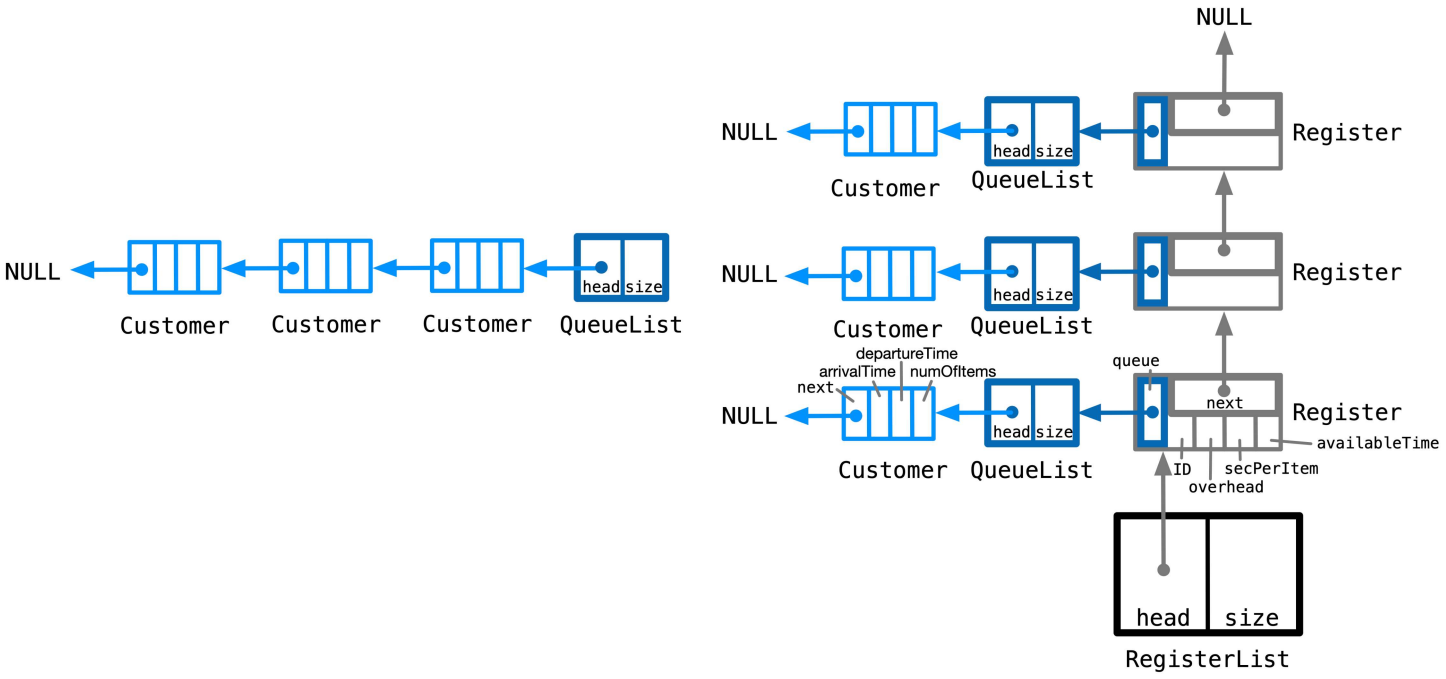

Figure 2: A snapshot of the grocery store adopting a single queue

2.5. End of simulation

Once the user enters an end of file character, the simulation should calculate some statistics. The program should calculate the wait time of each served customer in the linked list holding all served customers, and print the maximum, average and standard deviation of the wait time.

The wait time of customer i is the departure time minus the arrival time of a customer: x_i. The average wait time is the sum of all wait times of customers served divided by the number of customers served $\mu = \sum_{i=1}^{N} \frac{x_i}{N}$. The standard deviation is the sum of the difference between the wait time and average wait time squared divided by the number of customers served, all under the square root: $\sigma = \sqrt{\sum_{i=1}^{N} (x_i - \mu)^2 / N}$. You can assume that the number of customers served will be at least 1.

The examples in Section 4 further illustrate the above commands, the actions they take and the output they produce.

It is critical that your program deletes all dynamic data it allocates so as no memory leaks occur. In this assignment, memory leaks will be checked for and reported by exercise and the autotester, and *there is penalty for having memory leaks*. It is highly recommended that you use the valgrind memory checking program. A tutorial on valgrind as released with the previous lab. Please learn and use this tool.

2.6. Program Classes and Files

2.6.1. The Customer Class

The `Customer` class holds the properties of a customer, including its arrival time, departure time, number of items held and a pointer to next customer in line. The definition of the class appears in Customer.h. Examine the file and read through the comments to understand the variables and methods of the class. You must implement this class in the file Customer.cpp.

2.6.2. The QueueList Class

This class defines a linked list of `Customers`. It contains a single data member, `head`, which points to the first `Customer` in the list. It is defined in the file QueueList.h and the files contains comments that describe what the methods of the class do. You must implement this class in QueueList.cpp.

2.6.3. The Register Class

This class holds the properties of a `register`, which is a node in the linked list defined by `RegisterList`. It has six data members: `ID`, which stores the ID of the register, `secPerItem`, which stores the number of seconds it takes to process one item, `overheadPerCustomer`, which is the set up time a register takes per customer, `availableTime`, which stores the time at which the register was available either after the departure of a prior customer or when the register opened, `queue`, which

points to the `QueueList` object associated with the linked list of customers, and `next`, which is a pointer to the next `Register` in a list.

It is defined in the file Register.h and you must implement this class in the file Register.cpp. Again, read through the comments in Register.h to find out what the methods of the class do.

2.6.4. The RegisterList Class

This class defines a linked list of `Register`s. It contains two data members, `head`, which points to the first `Register` in the list, and `size`, which stores the number of registers in the list. It is defined in the file RegisterList.h and the files contains comments that describe what the methods of the class do. You must implement this class in RegisterList.cpp.

3. Coding Requirements

1. The code you will write shall be contained in only the source files ending in ".cpp": `main.cpp`, Customer.cpp, QueueList.cpp, Register.cpp and RegisterList.cpp. Skeletons of these files are released with the assignment's zip file. The zip file also contains corresponding ".h" files: Customer.h, QueueList.h, Register.h and RegisterList.h. These ".h" files contain comments that describe the classes defined in them and their methods. Please note that the ".h" files are **NOT** to be modified in any way. Modifying these files almost always results in a mark of 0 for the assignment.

   You may make use of helper functions to split up the code for readability and to make it easier to re-use. These functions (and their prototypes) **must** be in one of the aforementioned ".cpp" files. That is, you must not add any new ".h" or ".cpp" files.

2. Input and output must be done **only** using the C++ standard library streams `cin` and `cout`.

3. The stream input operator `>>` and associated functions such as `fail()` and `eof()` shall be used for all input. C-style IO such as printf() and scanf() shall not be used.

4. Strings shall be stored using the C++ library type `string`, and operations shall be done using its class members, not C-style strings.

5. C-library string-to-integer conversions (including but not limited to atoi(), strtol(), etc.) shall not be used.

6. The Standard Template Library (STL) shall not be used. The point of this assignment is for you to create your own linked list implementation.

7. Your program shall not leak memory. You will be penalized for leaking memory.

4. Examples

4.1. An Annotated Example

The program when first started, is ready to receive the mode of the simulation, either single or multiple

```
Welcome to ECE 244 Grocery Store Queue Simulation
Enter "single" if you want to simulate a single queue or "multiple" to simulate multiple queues:
```

Now the user types single to simulate a single queue.

```
> single
```

To which the program should respond with a message indicating the successful selection.

```
Simulating a single queue ...
```

The user then opens a register with ID 101 processing items at a rate of 1 item per second, setup time of 3 seconds and at time 0.

```
> register open 101 1 3 0
```

To which the program should respond with the message for a successful opening of the register.

```
Opened register 101
```

The user then adds a new customer with 5 items after 110 seconds.

```
> customer 5 110
```

To which the program should respond with the message for a successful entry of the customer, and indicate if the customer was queued to a free register and its ID.

```
A customer entered
Queued a customer with free register 101
```

The user then wants to add a new customer but misspells “customer”.

```
> custom 15 2
```

To which the program should say that this is an invalid operation.

```
Invalid operation
```

The user corrects “customer”, but misses one argument.

```
> customer 15
```

To which the program should mention that there were few arguments.

```
Error: too few arguments
```

The user then corrects the command to add a customer with 15 items after 2 seconds from the previous action.

```
> customer 15 2
```

The program should respond with a message indicating the successful entry of the customer, and that there is no free registers. This is because register 101 is busy with the customer that entered at 110.

```
A customer entered
No free registers
```

The user opens a register but mistakenly adds too many arguments.

```
> register open 102 2 2 1 0
```

To which the program should respond saying there are too many arguments.

```
Error: too many arguments
```

The user corrects the command and opens a register with ID 102 processing 2 items per second, set up time of 2 seconds per customer and after 1 second from the previous action, which is technically at 113.

```
> register open 102 2 2 1
```

To which the program should respond with a message indicating that the register is opened. Since a customer is there waiting in the single queue, it should be queued to the new free register. The program should say so in a message as well.

```
Opened register 102
Queued a customer with free register 102
```

The user entered the same command again mistakenly.

```
> register open 102 2 2 1
```

To which the program responded that there is already a register with this ID open.

```
Error: register 102 is already open
```

The user added another customer with 47 items at 118 seconds.

```
> customer 47 5
```

Since the time elapsed is 118, the simulation should depart the customer with 5 items at register 101. Then the customer entered at 118 should go to register 101 since it's now free.

```
Departed a customer at register ID 101 at 118
A customer entered
Queued a customer with free register 101
```

The user then closes register 101 after 50 seconds, which is at 168.

```
> register close 101 50
```

The program should then print that departures of customers happened at 145 at register 102 and at 168 at register 101. Notice the order of prints goes by the earliest first. Finally, the program prints that the register is closed.

```
Departed a customer at register ID 102 at 145
Departed a customer at register ID 101 at 168
Closed register 101
```

The user then closes register 103, which is not there to which the program responds with a message indicating that this register is not open.

```
> register close 103 0
Error: register 103 is not open
```

The user closes register 102, to which the program replies with a successful closure message.

```
> register close 102 0
Closed register 102
```

Finally, the user stops the simulation by entering an eof character, to which the program should print the statistics of the wait time.

```
> ^D
```

```
Finished at time 168
Statistics:
Maximum wait time: 50
Average wait time: 30.3333
Standard Deviation of wait time: 17.2498
```

4.2. Full session

The following is an example session. Note that the text from the prompt (>) up to the end of the line is typed by the user, whereas the prompt and line without a prompt are program output.

```
Welcome to ECE 244 Grocery Store Queue Simulation!
Enter "single" if you want to simulate a single queue or "multiple" to simulate
multiple queues:
> multiple
Simulating multiple queues ...
> register open 101 1 3 0
Opened register 101
> customer 5 110
A customer entered
Queued a customer with quickest register 101
> customer 15 2
A customer entered
Queued a customer with quickest register 101
> customer 2 3
A customer entered
Queued a customer with quickest register 101
> customer 17 5
Departed a customer at register ID 101 at 118
A customer entered
Queued a customer with quickest register 101
> customer 30 2
A customer entered
Queued a customer with quickest register 101
> register open 102 2 2 1
Opened register 102
> customer 11 1
A customer entered
Queued a customer with quickest register 102
> customer 32 2
A customer entered
Queued a customer with quickest register 102
> register open 103 1.5 3 5
Opened register 103
> customer 47 3
A customer entered
Queued a customer with quickest register 103
> customer 7 2
Departed a customer at register ID 101 at 136
A customer entered
Queued a customer with quickest register 102
> customer 10 3
A customer entered
Queued a customer with quickest register 103
```

```
> customer 52 15
Departed a customer at register ID 101 at 141
Departed a customer at register ID 102 at 148
A customer entered
Queued a customer with quickest register 102
> customer 15 22
Departed a customer at register ID 101 at 161
A customer entered
Queued a customer with quickest register 101
> customer 20 110
Departed a customer at register ID 101 at 194
Departed a customer at register ID 103 at 207.5
Departed a customer at register ID 101 at 212
Departed a customer at register ID 102 at 214
Departed a customer at register ID 103 at 225.5
Departed a customer at register ID 102 at 230
A customer entered
Queued a customer with quickest register 101
> register close 101 25
Departed a customer at register ID 101 at 309
Closed register 101
> register close 102 27
Departed a customer at register ID 102 at 336
Closed register 102
> register close 103 0
Closed register 103
> ^D
Finished at time 338 Statistics:
Maximum wait time: 182
Average wait time: 59.8462
Standard Deviation of wait time: 45.2431
```

5. Procedure

Create a sub-directory called `lab4` in your ece244 directory, and set its permissions so no one else can read it. Download the `lab4.zip` file, un-zip it and place the resulting files in the `lab4` directory.

In the release, there are 4 ".h" files: Customer.h, QueueList.h, Register.h and RegisterList.h. The files define the various classes, as described in Section 2.6. **You may NOT modify these files** to add to or delete from their content. Modifying the files commonly results in a mark of zero for the assignment.

In the release, there are also 5 ".cpp" files in which you will add your code: main.cpp, Customer.cpp, QueueList.cpp, Register.cpp and RegisterList.cpp. In the first file, you will complete the code that implements the simulation. In the rest you will implement the various classes described in Section 2.6.

There is a reference executable released with the assignment to help you better understand the specifications. You can run it using this command `~ece244i/public/main-ref` on ECF. When in doubt, run the executable to observe its behaviour.

To run your own executable, run `make` then `./main`. The public test cases are in files named `1`, `2`, `3`, `4` in the lab4_release.zip file. You may channel all the input from file `1`, for example, by running `./main < 1`.

The `~ece244i/public/exercise` command will also be helpful in testing your program. You should exercise the **executable**, i.e., `main`, using the command: `~ece244i/public/exercise 4 main`

As with previous assignments, some of the exercise test cases will be used by the autotester during marking of your assignment. We will not provide all the autotester test cases in exercise, however, so you should create additional test cases yourself and ensure you fully meet the specification listed above.

6. Deliverables

Submit the `main.cpp`, `Customer.cpp`, `QueueList.cpp`, `Register.cpp` and `RegisterList.cpp` files as lab 4 using the command
`~ece244i/public/submit 4`