CSCI 132 Data Structures—Lab #5

Introduction

You have seen several data structures: `Bag`, `Set`, `Queue`, and `Stack`. And you have seen different implementations: link-based ones that can grow as large as needed, array-based implementations with a fixed size, and implementations using dynamically resizing arrays that can grow as large as needed.

In this lab you will need to select among all these different options, picking an appropriate data structure in each of several scenarios.

Setup: Log onto logos, and copy the code for the lab, as usual.

-
  ```
  cp -r ~csci132/labs/lab5 ~/labs/
  cd ~/labs/lab5/
  ```

Grocery Store Discrete Event Simulator

The program in `Simulator.cpp` is a *discrete event simulator* for a grocery store. It simulates one day of activity in a grocery store, with customers who arrive, pick up a shopping basket, shop for a while, wait to check out, return their basket, then depart. The program only simulates certain aspects of the situation, and ignores other details we are not interested in. For example, it has simulated customers, and simulated shopping baskets that the simulated customers pick up and carry with them as they shop. But we don't simulate the aisles in the grocery store, prices, or even the products for sale. Even so, the simulator can be useful for answering certain questions, like:

- If the store expects a certain number of customers to arrive in a typical day, how many baskets should the store have available? Arriving customers aren't happy if there are no free baskets.
- How many cashiers are needed to keep lines from growing too big?
- Do some shopping baskets get significantly more wear-and-tear than others? If a basket can only be used a limited number of times before they break, and usage is very uneven, then the store might need to replace the most-used baskets frequently, or perhaps shuffle the baskets to spread the wear more evenly.

In a discrete event simulator, the system is modeled as a sequence of discrete steps in time. We might start with time t=0, when the store opens for the morning. A customer might arrive 5 minutes later (time t=5), enter the store and shop for 20 minutes (finishing at time t=25), then wait in the checkout line for 10 minutes before departing (at time t=35).

Examine `main()` and other functions in `Simulator.cpp` to get an idea of how the simulator works. Compile the program using:

```
clang++ -g -Wall Simulator.cpp Customer.cpp Basket.cpp PrecondViolatedExcept.cpp -o sim
```

Then run the simulator like this: `./sim 8 3`

This will simulate `8` customers arriving at a store that has `3` shopping baskets. It will then print out a minute-by-minute record of events that happened at the store, followed by some summary statistics at the end.

Try running with `30` customers and `3` baskets. Notice that only 29 of the customers were successful, and for one of the customers there were no baskets available (this happens right around `t=535`, scroll back to see it in the output). So a simulation like this can be useful for planning purposes, for example, by suggesting how many baskets might be needed for a given number of customers, or virtually testing out different scenarios before trying it in the real world.

Note: the output is long, so you can either scroll up to see it all, or you can save the output to a file using `./sim 8 3 > output.txt` (this writes the output to the file "`output.txt`", which you can then view in vscode). Or, use `./sim 8 3 | head` or `./sim 8 3 | head -30` to see the first few, or the first 30, lines of output.

Allowing for Duplicate Arrival Times

There is a flaw in the simulator. For each customer, an arrival time is chosen randomly between 0 and 699. If you run the simulator with only a few customers, say 10 or 20 or 30, and examine the statistics printed at the end, the numbers should be sensible. But if we try many customers, e.g.: `sim 1000 5` to simulate 1000 customers with 5 baskets, the statistics printed at the end look incorrect, showing fewer than 1000 customers arriving.

Why does this happen? With only a few customers, it is likely that each customer will arrive at a different time. With many customers, it would be likely that multiple customers could have the same arrival time. In other words, there may be duplicate values among the list of arrival times. But the arrival times are stored in a `LinkedSet` of integers, which ignores duplicate values, and stores at most a single copy of each value. `LinkedSet` is not an appropriate data structure for this purpose.

Modify the code to use a more appropriate data structure for holding the arrival times. You will need to change the code in a few places:

- Change the `arrivals` variable declaration to use a more appropriate data type.
- Within `simulate_time_step()`, adjust the code to account for the possibility that duplicate values will be present in the `arrivals` variables. So if value `t` appears in `arrivals` *n* times, then `simulate_time_step()` should simulate the arrival of *n* customers, not just one customer.
- Anywhere else the `arrivals` variable is used, examine the code to see if adjustments are needed to account for duplicate values, or to account for differences between the old and new data types.

Checkpoint 1:
Finish the updates described above. Test your code with a variety of parameters to ensure the right number of customers is always simulated.

When you are confident in your code, you can submit this part using: `~csci132/bin/submit`

Basket Collection and Distribution

The `baskets` variable is used to keep track of the collection of baskets waiting to be picked up by customers. This is declared as a `LinkedSet` of `Basket` pointers. However, this data structure is unordered, and arriving customers pick baskets at random from among all available baskets. This isn't very realistic, since most stores organize their baskets neatly rather than having a jumbled pile. This matters if the simulator is going to help answer questions about the distribution of wear-and-tear on baskets. In other words, `LinkedSet` is not an appropriate data structure for the `baskets` variable.

Modify the code to use a more appropriate data structure for holding the baskets waiting to be picked up by customers. Modify the variable declaration, then examine each place the variable is used to make any needed adjustments. In particular:

- Customers picking up a basket should not pick randomly.
- Customers returning a basket should not simply toss the basket into a jumble.

Hint: You will need to modify some methods in `Customer` as well.
When you are confident in your code, you can submit this part using: `~csci132/bin/submit`

Basket Usage Histogram

The simulator keeps track of how many minutes each individual basket is used by a customer. At the end of the simulation, it calculates and prints out the maximum usage for any basket (`maxUsage`) and the minimum usage (`minUsage`). A detailed histogram would be more informative.

Modify the code to print out a histogram, with the usage times grouped into 10-minute bins. Use the format below as an example of what to print out:

```
Basket maximum usage: 499 min
Basket minimum usage: 365 min
Basket usage histogram:
1 baskets used for 360 min
2 baskets used for 370 min
2 baskets used for 380 min
0 baskets used for 390 min
3 baskets used for 400 min
4 baskets used for 410 min
2 baskets used for 420 min
7 baskets used for 430 min
3 baskets used for 440 min
2 baskets used for 450 min
1 baskets used for 460 min
2 baskets used for 470 min
0 baskets used for 480 min
1 baskets used for 490 min
```

Hint: Declare a `hist` variable using an appropriate data structure. For each basket, take the usage of the basket, divide by 10, and add that to the data structure. Then, for each value between `minUsage/10` and `maxUsage/10`, use the values in `hist` to print out one line of the histogram as shown above.

Checkpoint 2:
Finish all the tasks described in the "Basket Collection and Distribution" and "Basket Usage Histogram" sections above.

Organizing the Checkout Lane

The `checkout` variable keeps track of the customers who are waiting at the checkout. It is declared as a `LinkedSet` of `Customer` pointers. Customers are added to the `checkout` set when they are ready to check out (so long as the set isn't too big). But to simulate the actions of a cashier, the simulator removes customers one at a time in a *random* order from the set (because the set data structure is not ordered). So this is not an appropriate data structure if we want to model the effects of customers having to wait in line to check out.

Modify the code to use a more appropriate data structure for keeping track of customers waiting to check out. Again, you will need to modify the variable declaration, then examine each place the variable is used to make any needed adjustments:

- Customers who are ready to check out should join the *back* of the line.
- When simulating the actions of a cashier, the customer at the *front* of the line should be selected to check out next.

Note: there is code in `simulate_shopper` to enforce a constraint that at most 10 customers will be waiting in line for the cashier at any one time. Some of our data structures have a fixed size and already have code to enforce this maximum size. By choosing an appropriate data structure for the `checkout` variable, some of this size-checking code becomes unnecessary.

- Simplify (or eliminate) some of the size-checking code in `simulate_shopper`.

What to Turn In

Submit the following using the command: `~csci132/bin/submit`

- A file named `lab5.txt` with your name and the names of people you worked with during this lab. A full "discussion log" is not necessary. There were no questions to answer in this lab.
- Your modified C++ files.

Be sure you add your name and date to the prologue for each file you modify.