CSC148 winter 2016

test, assignment, linked list queues — week 5

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W16/
416-978-5899

February 11, 2016

Outline

test

assignment #1

linked list queues

where we've been

- class design and implementation
- composition and inheritance
- stacks, sacks, containers
- linked lists

where to look

class design: Week 1 course website examples, lab #1 (also solutions), course notes, How to think like a computer scientist.

composition and inheritance: Week 2 course website examples, lab #2 (also solutions), course notes

stacks, sacks, containers: Week 3 course website examples, lab #3 (also solutions)

linked lists: Week 4 course website examples, lab #4, How to think like a computer scientist

how to study

- look at material from all three lecturers
- work on a handout, or incomplete code, **before** looking at solution
- study groups can challenge each other, critique solutions
- office hour Tuesday 2–4:30, BA7172

What is it?

- Assignment 1 is a ride-sharing simulation
  - Riders request drivers to pick them up at their current location and drop them off somewhere else
  - Drivers request riders
- A text file is used to set up the initial riders and drivers
- ... and then the simulation runs, and we see what happens!
  - Useful for answering questions about real-world events
  - "How long did riders wait for a pickup, on average?"
  - "How much distance is traveled by drivers, on average?"

Starting the World

- We start the world of the simulation by using a text file of events
- The text file has only two types of events
  - **RiderRequest**: rider requests a driver
  - **DriverRequest**: driver requests a rider
  - i.e. the word before **Request** is the type of person doing the requesting
- Each rider has exactly one **RiderRequest** event in the file
- Each driver has exactly one **DriverRequest** event in the file, but **Pickup** or **Dropoff** events may generate new **DriverRequest**

Sample Text File

```
#At time 1, Dan exists
#Dan is at location 1,6, requests a driver, and is willing
#to wait 15 units of time for pickup before he cancels
# The 15 is the rider's "patience"
1 RiderRequest Dan 1,1 6,6 15

#At time 10, Arnold exists
#Arnold is at location 3,3, requests a rider,
#and his car moves 2 units of distance per unit time
10 DriverRequest Arnold 3,3 2
```

Locations

- think of the riders and drivers existing on an x-y plane, so their locations are simplified compared to a real city
- there is a **Location** object to represent these locations in the simulation
- distance from one location to another is the vertical distance plus horizontal distance
- so, what is the distance from **1,1** to **6,6**?

Other Events

besides **RiderRequest** and **DriverRequest** events, three other kinds of events can be generated during the simulation

Cancellation: cancels a waiting rider if they wait for pickup beyond their patience

Pickup: occurs when a driver picks up a rider

Dropoff: occurs when a driver drops-off a rider

event priorities

- each event has a priority, which is its timestamp
- events with smaller timestamps have higher priorities
- a priority queue is used to manage pending events

```
>>> pq = PriorityQueue()
>>> pq.add(Event(4))
>>> pq.add(Event(2))
>>> pq.add(Event(7))
>>> pq.remove().timestamp
2
```

Sample Text File: What Happens?

- What are all of the generated events?
- How long does Dan wait?
- What is Arnold's total distance traveled?
- What is Arnold's total distance traveled with a rider?
- Change Dan's patience from 15 to 10 — now what happens?

Dispatcher

- The dispatcher knows about the available drivers and riders
- It is also used to request a driver for a rider, request a rider for a driver, or cancel a rider request
- … but wait, don't events already do this kind of thing?
  - No — events don't do anything on their own
  - They ask the dispatcher to perform appropriate actions
  - Dispatcher is part of the "business logic" to make things happen

Monitor

- OK — so we have all of these events happening
- And we're supposed to return statistics (average wait time of riders, etc.) when the simulation is over
- How?
  - We use the monitor!
  - The monitor is our bookkeeper, keeping track of relevant data from which we compute our stats

Monitor...

- The monitor has two important methods
  - `notify`: events call this method to have the monitor record an activity
  - `report`: produces stats about the activities that the monitor has remembered
    - Each stat is computed by a separate private helper function

Events and Activities

- Why do we have **both** events and activities?
  - Events are used to move the simulation forward
    - They are active (cause things to happen)
  - Activities are used **only** in the monitor
    - They are passive (just used to record things)

something linked lists do better than lists?

list-based Queue has a problem: adding **or** removing will be slow.

symmetry with linked list

which end of a linked list would be best to add, which to remove? why??

build pop_front

... already have append

revisit Queue API

use an underlying LinkedList

revisit Stack API while we're at it

also use an underlying LinkedList

they're all Containers

stress drive them through **container_cycle**, in container_timer.py:

- list-based Queue
- linked-list-based Queue
- list-based Stack
- linked-list-based Stack

what matters is the growth rate

as Queue grows in size, list-based-Queue bogs down impossibly fast