Chapter 7-4, Lecture notes

A Summary of Position-Oriented ADTs

1 We done three distinctive abstract data type: the list, the stack, and the queue.
  1.1 They have a common theme: all of their operations are defined in terms of the positions of their data items.
  2.2 Stacks and queues greatly restrict the positions that their operations can affect – can only affect the end positions.
  3.3 The list does not have that restriction.

2 Stacks and queues are quite similar. The operations are paired off like:

- createStack and createQueue. – create the empty ADT
- Stack isEmpty and queue isEmpty – determine where any items exist in the ADT
- push and enqueue – insert a new item into one end of the ADT (top and back)
- pop and dequeue – the pop deletes the most recent item from the top of the stack and dequeue deletes the first item at the front of the queue
- Stack getTop and queue getFront – getTop retrieves the most recent item from the stack's top and getFront retrieves the most recent item from the front of the queue

3 The ADT list operations can be viewed as general versions for stack and queue.

- getLength – if you remove the restriction that stack and queue isEmpty, you obtain an operations that can count the number of items present
- insert – if you remove the restriction that push and enqueue can insert new items on only one position then you can have an operation to insert at any position
- remove – if you remove the restriction that pop and dequeue can delete from only one position, you would get an operations that can delete from any position in the list
- retrieve – if you remove the restriction that stack getTop and queue getFront can retrieve from only one position, you get an operation that can retrieve the item from any position in the list

Chapter 7-5, Application: Simulation

1 Simulation is a major application area for computers: a technique for modeling the behavior of both natural and human-made systems.
  1.1 Simulation can generate statistics to summarize how well an existing system is working or to predict how well a proposed system will work.

  1.2 So let's consider Ms. Simpson, president of the First City Bank of Springfield.

1.2.1 Customers at her bank complain how long they have to wait for service from one teller.

1.2.2 Rather than either stand at the bank all day with a stopwatch or temporarily hiring another teller, she decides to do a simulation.

1.3 First step in simulating a system like a bank is to construct a mathematical model that gets the relevant information about the system.

1.3.1 For instance: how many tellers does the bank employ, how often do customers arrive?

1.3.2 If the model is accurate, a simulation can derive accurate predictions about the system's overall performance.

1.3.3 For instance, a simulation could see what would happen with hiring another teller.

1.3.4 If the wait time goes down dramatically then the extra cost is justified.

1.4 Central to a simulation is the idea of simulated time – sort of like using a stopwatch to measure time elapsed.

1.4.1 So with a bank with only one teller, the start of the day starts at time 0.

1.4.2 The simulated time ticks away (probably in minutes) and certain events happen.

1.4.3 At time 12, the first customer arrives and, since there is no line, gets serviced immediately.

1.4.4 At time 20 the second customer arrives but since the first customer is not done, the second has to wait.

1.4.5 At time 38 the first customer leaves and the second can be serviced.

1.4.6 You need to keep track of the wait times. The first customer had 0 wait but the second had to wait 18 minutes.

1.4.7 Then you could get an average of wait times.

1.4.8 Mathematicians have learned to model events like people arriving by using probability theory.

1.4.9 The simulation uses these events and thus would be called "event-driven simulation".

1.4.10 The goal is to predict long-term average behavior – not predict occurrences of specific events.

1.5 You do not need a degree of mathematical sophistication for a simulation.

1.5.1 Suppose you already had sample arrival and transaction times in a file, like so:

```
20      5
22      4
23      2
30      3
```

1.5.2 First customer arrives at 20 and takes 5 minutes; second customer arrives but has to wait in line and so one.

1.5.3 The file is ordered by arrival time.

1.6 Now the following is the results of a simulation:

Time	Event
20	Customer 1 enters bank and begins transaction
22	Customer 2 enters bank and stands at end of line
23	Customer 3 enters bank and stands at end of line
25	Customer 1 departs; customer 2 begins transaction
29	Customer 2 departs; customer 3 begins transaction
30	Customer 4 enters bank and stands at the end of line
31	Customer 3 departs; customer 4 begins transaction
34	Customer 4 departs

1.6.1 A customer's wait time is the elapsed time between arrival in the bank and the start of the transaction.

1.6.2 You can obtain the average wait time from that.

1.7 There are two types of events you want to concentrate on:

- Arrival events – events indicate the arrival at the bank of a new customer. The input file specifies the times at which the arrival events occur – hence they are "external events". When a customer arrives, one of two things happen: if the teller is not busy, the customer gets immediate service or, if there is a line, the customer must wait at the end of the line
- Departure events – events indicate the departure from the bank of a customer who has finished his/her transaction. The simulation determines the times at which the departure happens – they are "internal events".

2 First attempt at simulation:

```
// initialize
currentTime = 0
Initialize the line to "no customers"

while (currentTime ≤ time of the final event)
{       if (an arrival event occurs at time currentTime)
                Process the arrival event

        if (a departure event occurs at time currentTime)
                Process the departure event

        // when an arrival event and departure event occur at the same time,
        // arbitrarily process the arrival event first

        ++currentTime
} // end while
```

2.1 Do you want to increment the time by one?

2.1.1 You would for a "time-driven simulation" where you would determine arrival and departure times at random and compare those times to "currentTime".

2.1.2 But if you have external events like times from a file, you can be interested only when the events actually happen and go directly from time to time.

2.1.3 Thus the pseudo code can be revised to do:

```
// Initialize the line to "no customer"

while (events remain to be processed)
{         currentTime = time of next event

          if (event is an arrival event)
                    Process the arrival event
          else
                    Process the departure event

          // when an arrival event and departure event occur at the same time,
          // arbitrarily process the arrival event first
} // end while
```

2.2 You have to determine the time of the next arrival so that you can implement: currentTime = time of next event

2.3 To make this determination, you must maintain an "event list".

2.3.1 This list would have all the arrival and departure times that will occur but have not yet.

2.3.2 The times would be in ascending order. The algorithm simply gets the time from the front of the list and advances the time to that and processes the event.

2.3.3 The difficulty is managing the list.

2.3.4 You could attempt to read the entire input file and create an event list of all arrival and departure time – a bit impractical.

2.3.5 Instead you kept the earliest unprocessed arrival event int the event list.

2.3.6 When you eventually process this event, you replace it in the event list with the next unprocessed arrival event – i.e. from the next item in the input file.

3 Also, you only need to keep one kind of event on the event list – the arrivals.

3.1 How do you determine the departure event?

3.2 It is the arrival time plus the length of the transaction or:
time of next departure = time service begins + length of transaction

3.3 You place the departure event on the event list.
  3.3.1 You can process an event when it is time for the event to occur.
  3.3.2 Do two general types of actions:
    - Update the line: Add or remove customers
    - Update the event list: Add or remove events

3.4 With customers arriving, they go to the back of the line.
  3.4.1 It is natural to have the queue with the arrival time and the length of transaction.
  3.4.2 The front of the line is the customer currently being services.
  3.4.3 However, the event list, since it is sorted by time, is not a queue (more on that later).
  3.4.4 You process an event like so:

```
//PROCESS AN ARRIVAL EVENT
// Update the event list
Delete the arrival event for customer C from the event list

if (new customer C begins transaction immediately)
        Insert a departure event for customer C into the event list (time of event =
        current_time + transaction length)

if (not at the end of the input file)
        Read a new arrival event and add it to the event list
        (time of event = time specified in file)
```

  3.4.5 The arrival event can occur either before or after the departure event.

```
//PROCESS A DEPARTURE EVENT
// Update the line
Delete the customer at the front of the queue
if (the queue is not empty)
        The current front customer begins transaction

// Update the event list
Delete the departure event from the event list
if (the queue is not empty)
        Insert into the event list the departure event for the customer now at
        the front of the queue (time of event = current time + transaction length)
```

4 After processing the departure event, you do not read another arrival event from the file.
  4.1 Assuming that the file has not been completely read, the event list will contain an arrival event whose time is earlier than any arrival still in the file.

5 There is no typical form that an event list takes.

5.1 However, the event list has four possible configurations:

- Initially, the event list contains an arrival event A after you read the first arrival event from the input file but before you process it: Event list: A(initial state)
- Generally, the event list for this simulation contains exactly two events: one arrival event A and one departure event D. Either the departure event is first or the arrival event is first like so:
  Event list: D A (general case – next event is a departure)
  Or
  Event list: A D (general case – next event is an arrival)
- If the departure event is first and that event leaves the teller's line empty, a new departure event does not replace the just-processed event. In this case the event list is: Event list: A (a departure leaves the teller's line empty)
- If the arrival event is first and if, after it is processed, you are at the end of the input file, the event list contains only a departure event: Event list: D (input exhausted)

5.2 So the event list would be ordered by the time therefore then event list cannot be a queue.

5.3 Let's see the simulation:

```
+simulate()
// performs the simulation

    Create an empty queue bankQueue to represent the bank line
    Create an empty event list eventList

    Get the first arrival event from the input file
    Place the arrival event in eventList

    while (eventList is not empty)
    {       newEvent = the first event in eventList

            if (newEvent is an arrival event)
                    processArrival(newEvent, arrivalFile, eventList, bankQueue)
            else
                    processDeparture(newEvent, eventList, bankQueue)
    } // end while

+processArrival(in arrivalEvent:Event, in arrivalFile:File,
                  inout anEventList:EventList, inout bankQueue:Queue)
// Processes an arrival event

    atFront = bankQueue.isEmpty()        // present queue status

    // update the bankQueue by inserting the customer, as described in arrivalEvent,
    //   into the queue
    bankQueue.enqueue(arrivalEvent)

    // update the event list
    Delete arrivalEvent from anEventList

    if (atFront)
    {       // the line was empty, so new customer is at front of the line and begins
            // transaction immediately
            Insert into anEventList a departure event that corresponds to the
                new customer and has currentTime = currentTime + transaction length
    } // end if

    if (not at end of input file)
    {       Get the next arrival event from arrivalFile
            Add the event – with time as specified in the input file – to anEventList
    } // end if

+processDeparture(in departureEvent:Event, inout anEventList:EventList,
                    inout bankQueue:Queue)
// Processes a departure event

    // update the line by deleting the front customer
    bankQueue.dequeue()

    // update the event list
    Delete departureEvent from anEventList
    if (!bankQueue.isEmpty())
    {       // customer at front of line begins transaction
            Insert into anEventList a departure event that corresponds to the
              Customer now at the front of the line and has currentTime =
              currentTime + transaction length
    } // end if
```

5.4 Let's finish it with:

```
+createEventList()
// Creates an empty event list

+destroyEventList()
// Destroys an event list

+isEmpty():boolean {query}
// Determines whether an event list is empty

+insert(in anEvent:Event)
```

// Inserts anEvent into the event list so that events are ordered by time. If an
//   arrival event and a departure event have the same time, the arrival event
//   precedes the departure event

+delete()
// Deletes the first event from an event list

+retrieve(out anEvent:Event)
// Sets anEvent to the first event in an event list

6   The following is a trace of a simulation (at least the beginning)

Time	Action	bankQueue (front to back)	anEventList (beginning to end)
0	Read file, place event in anEventList	(empty)	A 20.5
20	Update anEventList and bankQueue: Customer 1 enters bank	20 5	(empty)
	Customer 1 begins transaction, create departure event	20 5	D 25
	Read file, place event in anEventList	20 5	A 22 4 D 25
22	Update anEventList and bankQueue: Customer 2 enters bank	20 5 22 4	D 25
	Read file, place event in an EventList	20 5 22 4	A 23 2 D 25
23	Update anEventList and bankQueue: Customer 3 enter bank	20 5 22 4 23 2	D 25
	Read file, place event in anEventList	20 5 22 4 23 2	D 25 A 30 3
25	Update anEventList and bankQueue: Customer 1 departs	22 4 23 2	A 30 3
	Customer 2 begins transaction, create departure event	22 4 23 2	D 29 A 30 3