Discrete Event Simulation

CS1316: Representing Structure and Behavior

Motivation: a simulation…

- There are three Trucks that bring product from the Factory.
  - On average, they take 3 days to arrive.
  - Each truck brings somewhere between 10 and 20 products—all equally likely.
- We've got five Distributors who pick up product from the Factory with orders.
  - Usually they want from 5 to 25 products, all equally likely.
- It takes the Distributors an average of 2 days to get back to the market, and an average of 5 days to deliver the products.
- Question we might wonder: How much product gets sold like this?

Don't use a Continuous Simulation

- We don't want to wait that number of days in real time.
- We don't even care about every day.
  - There will certainly be timesteps (days) when *nothing* happens of interest.
- We're dealing with different probability distributions.
  - Some uniform, some normally distributed.
- Things can get out of synch
  - A Truck may go back to the factory and get more product before a Distributor gets back.
  - A Distributor may have to wait for multiple trucks to fulfill orders (and other Distributors might end up waiting in line)

Running a DESimulation

```
Welcome to DrJava.
> FactorySimulation fs = new
  FactorySimulation();
> fs.openFrames("D:/temp/");
> fs.run(25.0)
```

Story

- Discrete event simulation
  - Simulation time != real time
- Key ideas:
  - Queues
    - What makes a queue a queue is its behavior not its code.
    - Priority queues allow objects to cut in line (if they are high enough priority)
  - Different kinds of randomness
    - Uniformity
    - Normally distributed populations of events
  - Straightening time
    - Inserting it into the right place
    - Sorting it afterwards

Discrete vs. Continuous: No time loop

- In a discrete event simulation: There is no time loop.
  - There are events that are scheduled.
  - At each **run** step in the event loop, the next scheduled event with the *lowest* time gets processed.
    - The current time is then *that* time, the time that that event is supposed to occur.
- Key idea: We have to keep the list of scheduled events *sorted* (in order)

DES Agents don't act()

- In a discrete event simulations, agents don't act().
  - Instead, they wait for events to occur.
  - They schedule new events to correspond to the *next* thing that they're going to do.
- Key idea: Events get scheduled "stochastically" (at times that depend on probabilities).

DES Agents get blocked

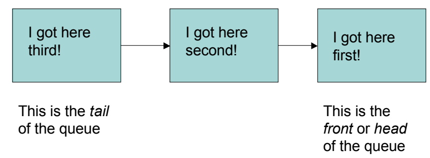
- Agents can't do everything that they want to do.
- If they want product (for example) and there isn't any, they get *blocked.*
  - They can't schedule any new events until they get unblocked.
- Many agents may get blocked awaiting the same resource.
  - More than one Distributor may be awaiting arrival of Trucks
- Key: We have to keep track of the Distributors waiting *in line* (*in the* ***queue***)

Story

- Discrete event simulation
  - Simulation time != real time
- Key ideas:
  - Queues
    - What makes a queue a queue is its behavior not its code.
    - Priority queues allow objects to cut in line (if they are high enough priority)
  - Different kinds of randomness
    - Uniformity
    - Normally distributed populations of events
  - Straightening time
    - Inserting it into the right place
    - Sorting it afterwards

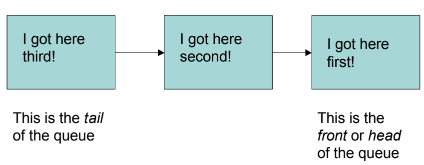
Key idea #1: Queues again (contrast stacks)

- First-In-First-Out List
  - First person in line is first person served

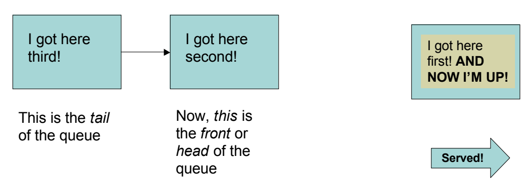


First-in-First-out

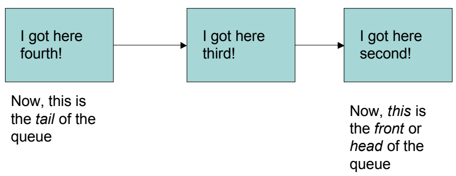
- New items *only* get added to the tail.
  - Never in the middle
- Items *only* get removed from the head.



As items leave, the head shifts



As new items come in, the tail shifts



What can we do with queues?

- *push(anObject):* Tack a new object onto the tail of the queue
- *pop*(): Pull the end (head) object off the queue.
- *peek()*: Get the head of the queue, but don’t remove it from the queue.
- *size*(): Return the size of the queue

A queue is a queue, no matter what lies beneath.

- Our description of the queue *minus* the implementation is an example of an ***abstract data type (ADT)***.
  - An abstract type is a description of the methods that a data structure knows and what the methods do.
- We can actually write programs that use the abstract data type *without* specifying the implementation.
  - There are actually *many* implementations that will work for the given ADT.
  - Some are better than others.

Key idea #2:
Different kinds of random

- We’ve been dealing with *uniform* random distributions up until now, but those are the *least* likely random distribution in real life.
- How can we generate some other distributions, including some that are more realistic?

Visualizing a uniform distribution

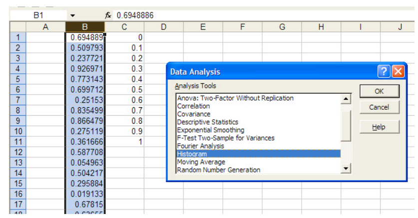
By writing out a tab and the integer, we don't have to do the string conversion.

```
import java.util.*;  // Need this for Random
import java.io.*;  // For BufferedWriter

public class GenerateUniform {
  public static void main(String[] args) {
    Random rng = new Random(); // Random Number Generator
    BufferedWriter output=null; // file for writing

   // Try to open the file
    try {
      // create a writer
      output =
        new BufferedWriter(new FileWriter("D:/cs1316/uniform.txt"));
    } catch (Exception ex) {
      System.out.println("Trouble opening the file.");
    }
      // Fill it with 500 numbers between 0.0 and 1.0, uniformly
       distributed
      for (int i=0; i < 500; i++){
        try{
        output.write("\t"+rng.nextFloat());
        output.newLine();
      } catch (Exception ex) {
        System.out.println("Couldn't write the data!");
        System.out.println(ex.getMessage());
        }
      }
      // Close the file
       try{
        output.close();}
      catch (Exception ex)
      {System.out.println("Something went wrong closing the file");}
}
}
```

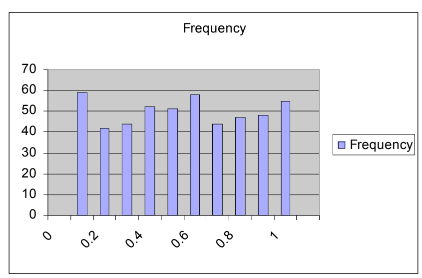
How do we view a distribution? A Histogram



Then graph the result

	A	B
1	Bin	Frequency
2	0	0
3	0.1	11
4	0.2	12
5	0.3	9
6	0.4	7
7	0.5	8
8	0.6	12
9	0.7	10
10	0.8	8
11	0.9	13
12	1	10
13	More	0
14		

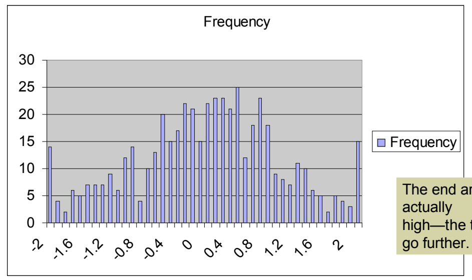
A Uniform Distribution



A Normal Distribution

```
// Fill it with 500 numbers between -1.0 and 1.0, normally distributed
    for (int i=0; i < 500; i++){
      try{
      output.write("\t"+rng.nextGaussian());
      output.newLine();
    } catch (Exception ex) {
      System.out.println("Couldn't write the data!");
      System.out.println(ex.getMessage());
      }
    }
```

Graphing the normal distribution



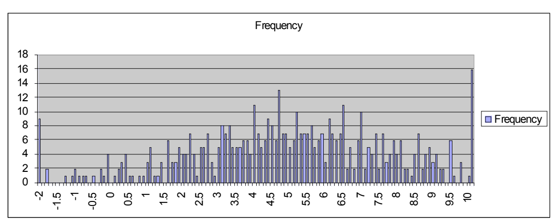
The end aren't actually high—the tails go further.

How do we shift the distribution where we want it?

```
// Fill it with 500 numbers with a mean of 5.0 and a
  //larger spread, normally distributed
    for (int i=0; i < 500; i++){
      try{
      output.write("\t"+((range * rng.nextGaussian())+mean));
      output.newLine();
    } catch (Exception ex) {
      System.out.println("Couldn't write the data!");
      System.out.println(ex.getMessage());
      }
    }
```

Multiply the random **nextGaussian()** by the range you want, then add the mean to shift it where you want it.

A new normal distribution



Key idea #3: Straightening Time

- Straightening time
  - Inserting it into the right place
  - Sorting it afterwards
- We'll actually do these in reverse order:
  - We'll add a new event, then sort it.
  - Then we'll insert it into the right place.

Exercising an EventQueue

We're stuffing the EventQueue with events whose times are *out* of order.

```
public class EventQueueExercisor {
  public static void main(String[] args){
    // Make an EventQueue
    EventQueue queue = new EventQueue();

    // Now, stuff it full of events, out of order.
    SimEvent event = new SimEvent();
    event.setTime(5.0);
    queue.add(event);

    event = new SimEvent();
    event.setTime(2.0);
    queue.add(event);

    event = new SimEvent();
    event.setTime(7.0);
    queue.add(event);

    event = new SimEvent();
    event.setTime(0.5);
    queue.add(event);

    event = new SimEvent();
    event.setTime(1.0);
    queue.add(event);

    // Get the events back, hopefull in order!
    for (int i=0; i < 5; i++) {
      event = queue.pop();
      System.out.println("Popped event
        time:"+event.getTime());
    }
  }
}
```

If it works right, should look like this:

```
Welcome to DrJava.
> java EventQueueExercisor
Popped event time:0.5
Popped event time:1.0
Popped event time:2.0
Popped event time:5.0
Popped event time:7.0
```

Implementing an EventQueue

```
import java.util.*;

/**
 * EventQueue
 * It's called an event "queue," but it's not really.
 * Instead, it's a list (could be an array, could be a linked list)
 * that always keeps its elements in time sorted order.
 * When you get the nextEvent, you KNOW that it's the one
 * with the lowest time in the EventQueue
 **/
public class EventQueue {
  private LinkedList elements;

  /// Constructor
  public EventQueue(){
    elements = new LinkedList();
  }
```

Mostly, it's a queue

```
public SimEvent peek(){
  return (SimEvent) elements.getFirst();}

 public SimEvent pop(){
  SimEvent toReturn = this.peek();
  elements.removeFirst();
  return toReturn;}

 public int size(){return elements.size();}

 public boolean empty(){return this.size()==0;}
```

Two options for add()

```
/**
  * Add the event.
  * The Queue MUST remain in order, from lowest time to
    highest.
  **/
 public void add(SimEvent myEvent){
 // Option one: Add then sort
 elements.add(myEvent);
 this.sort();
 //Option two: Insert into order
 //this.insertInOrder(myEvent);
 }
```

There are *lots* of sorts!

- Lots of ways to keep things in order.
  - Some are faster – best are O(n log n)
  - Some are slower – they're always $O(n^2)$
  - Some are $O(n^2)$ in the worst case, but on average, they're better than that.
- We're going to try an *insertion sort*

Useful Link on Sorting

- http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html

Includes animations to show how sorting algorithms differ in behavior and performance

How an insertion sort works

- Consider the event at some position (1..n)
- Compare it to all the events *before* that position *backwards*—towards 0.
  - If the comparison event time is *LESS THAN* the considered event time, then shift the comparison event down to make room.
  - Wherever we stop, that's where the considered event goes.
- Consider the next event…until done

Option #2: Put it in the *right* place

```
/**
  * Add the event.
  * The Queue MUST remain in order, from lowest time to
    highest.
  **/
public void add(SimEvent myEvent){
// Option one: Add then sort
//elements.add(myEvent);
//this.sort();
//Option two: Insert into order
this.insertInOrder(myEvent);
}
```

insertInOrder()

Again, trace it out to convince yourself that it works!

```
/**
  * Put thisEvent into elements, assuming
  * that it's already in order.
  **/
 public void insertInOrder(SimEvent
     thisEvent){
  SimEvent comparison = null;

  // Have we inserted yet?
  boolean inserted = false;
  for (int i=0; i < elements.size(); i++){
    comparison = (SimEvent)
     elements.get(i);
// Assume elements from 0..i are less than
     thisEvent
    // If the element time is GREATER,
     insert here and
    // shift the rest down
    if (thisEvent.getTime() <
     comparison.getTime()) {
      //Insert it here
      inserted = true;
      elements.add(i,thisEvent);
      break; // We can stop the search loop
    }
  } // end for

  // Did we get through the list without
     finding something
  // greater?  Must be greater than any
     currently there!
  if (!inserted) {
    // Insert it at the end
    elements.addLast(thisEvent);}
 }
```

Finally: A Discrete Event Simulation

- Now, we can assemble queues, different kinds of random, and a sorted EventQueue to create a discrete event simulation.

Running a DESimulation

Welcome to DrJava.

```
> FactorySimulation fs = new
  FactorySimulation();
> fs.openFrames("D:/temp/");
> fs.run(25.0)
```

What we see (not much)