CSCI203

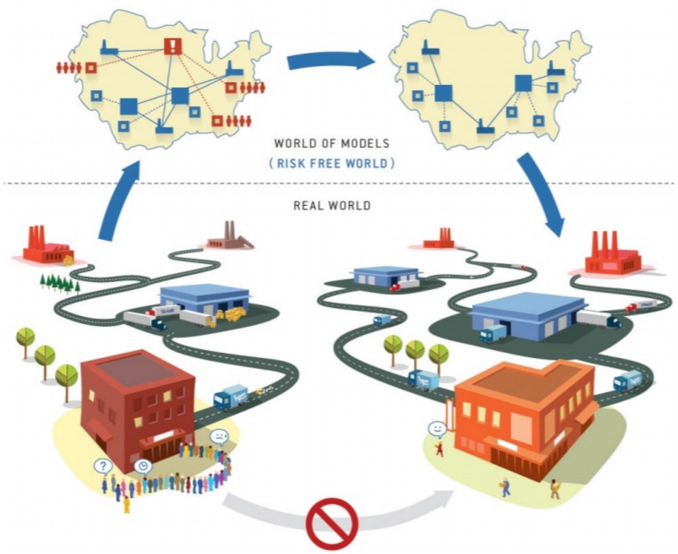
Algorithms and Data Structures

Simulation & Priority Queues

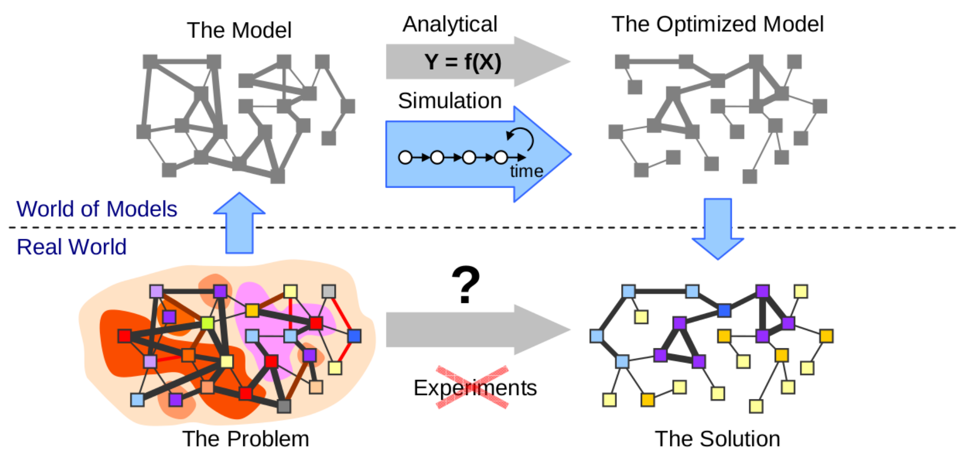
Lecturer: Dr. Zuoxia Yu
Room 3.116
Email: zyu@uow.edu.au

Simulation

- The production of a computer model of something, especially for the purpose of study



Modeling



Analytical vs. Simulation Modeling

Andrei Borshchev. 2013. The big book of simulation modeling: Multimethod modeling with anylogic 6. AnyLogic North America.

Types of Simulation

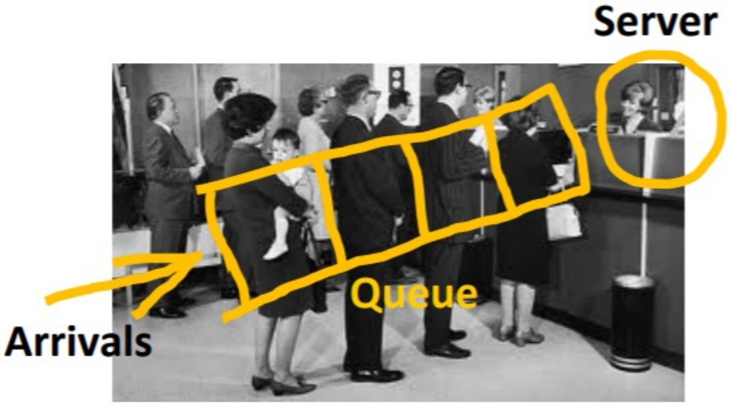
- Continuous:
  - Time is broken into discrete chunks (ticks)
  - Usually used to model a continuous process.
- Discrete:
  - Time can take any value
  - Usually used to model discrete events

Continuous Simulation - Clock driven

- Often dependent on complex mathematics.
- Often requires extreme computing resources.
  - Supercomputers
- We will not be looking at continuous simulation in this subject.

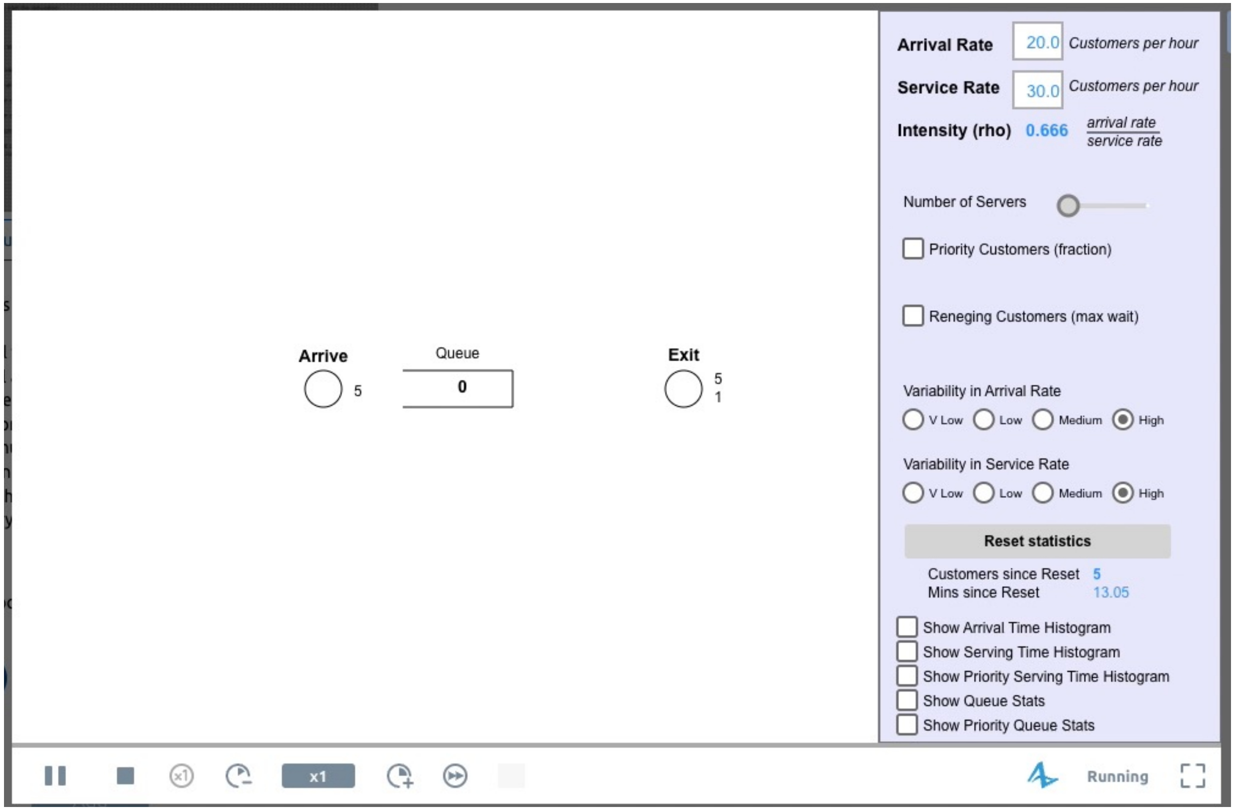
Discrete Simulation -Event driven

- Normally a lot less mathematically complex.
- Usually requires a lot less computer resources.
- E.g. Queue simulation
  - Widely used to evaluate queue-based processes
  - Shops
  - Production lines
  - Industrial processes



- We will look at some simple examples and see how we might implement them.

Single Queue and Server Simulation



https://cloud.anylogic.com/model/1c626205-7dfa-48ae-8f57-52cd89183afc?mode=SETTINGS

Scenario 1

- A single server queue.
  - Customers arrive at random intervals to be served
  - If the server is not busy the customer will be served immediately
  - If the server is busy the customer will join the end of the (possibly empty) queue.
  - When the server has finished with the customer the next customer (if any) begins service – first customer in queue.

Scenario 1…

- Events.
  - Customer arrives
  - Customer starts service
  - Customer ends service and leaves
- What we know:
  - When each customer arrives
  - How long they take to serve
- What we want to know:
  - How big is the queue on average?
  - How busy is the server?
  - What proportion of customers have to wait in a queue?

Scenario 1…

- Data
  - Input data is a file consisting of a set of records containing
    - Arrival time
    - Service time
  - Sorted by arrival time
    - 0.24 0.55
    - 0.59 0.16
    - 0.90 0.07
    - 1.87 0.69

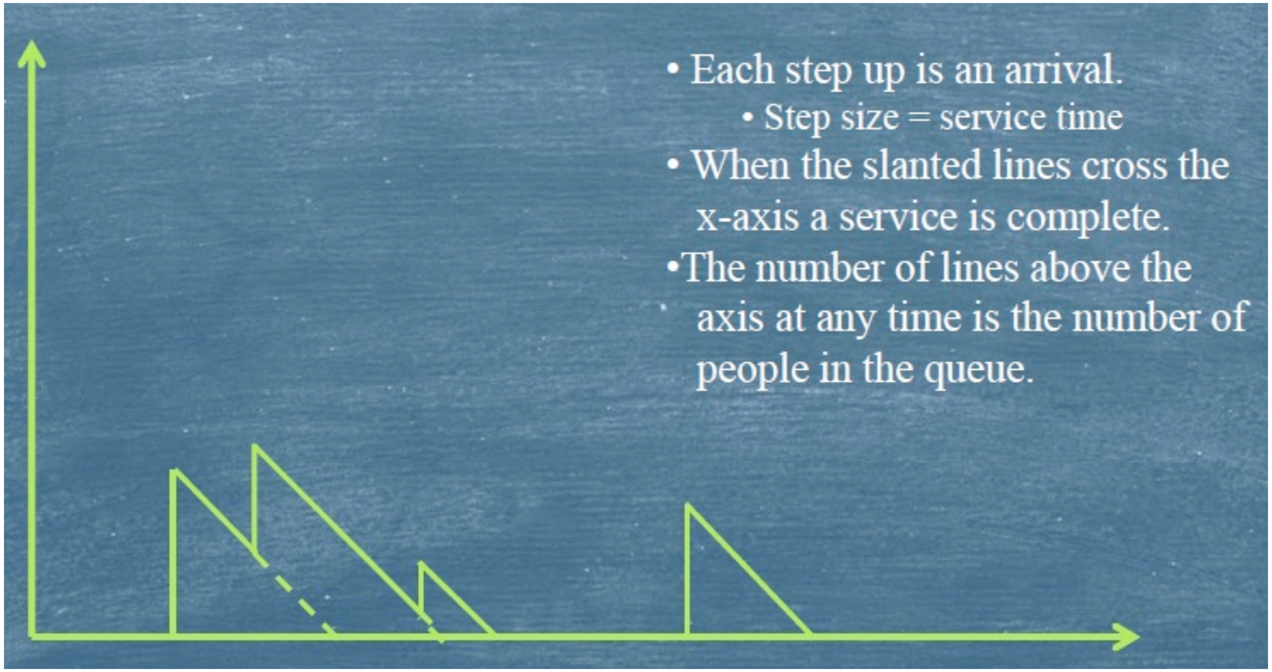
Scenario 1…

- Manual Simulation
  - From the data file we can get a feel for what is happening
  - At time 0.00 the simulation starts
    - The server is idle
    - The queue is empty
  - At time 0.24 the first customer arrives
    - The server is busy for the next 0.55 (until 0.79)
    - The queue is empty
  - At time 0.59 the second customer arrives
    - The server is still busy
    - The queue now contains 1 customer (customer 2)
  - At time 0.79 the server finishes with customer 1
    - The server stays busy for the next 0.16 (until 0.95)
    - The queue is empty

```
0.24 0.55
0.59 0.16
0.90 0.07
1.87 0.69
…
```

Graphs of Queues

- We can represent what is happening with a graph:



The Algorithm

- Starting to design the algorithm.
- Data structures
  - We need to hold the queue
    - What should we put in it?
  - We need to keep track of the time
  - We need to know if the server is busy
    - If so we need to know when they will finish
  - We need to know when the next customer will arrive
  - We need to track statistics

The Algorithm

- Starting to design the algorithm.
- Procedures
  - Initialise the simulation
  - Process an arrival
  - Process a service completion
  - Finish the simulation
- Once the simulation is running how do we decide what to do next?
  - Compare the next arrival time with the end of service time

The Algorithm

- Initialise the simulation

```
time = 0
busy = false
queue = empty
read next_arrival, next_service
```

- Set up for statistics collection

The Algorithm…

- The main program loop

```
initialise
repeat
    if busy = true then
       if service_end < next arrival then
           process_service_end
       else
           process_arrival
    else
           process_arrival
    fi
until arrival file is empty and busy = false
finish
```

The Algorithm…

- Process an arrival

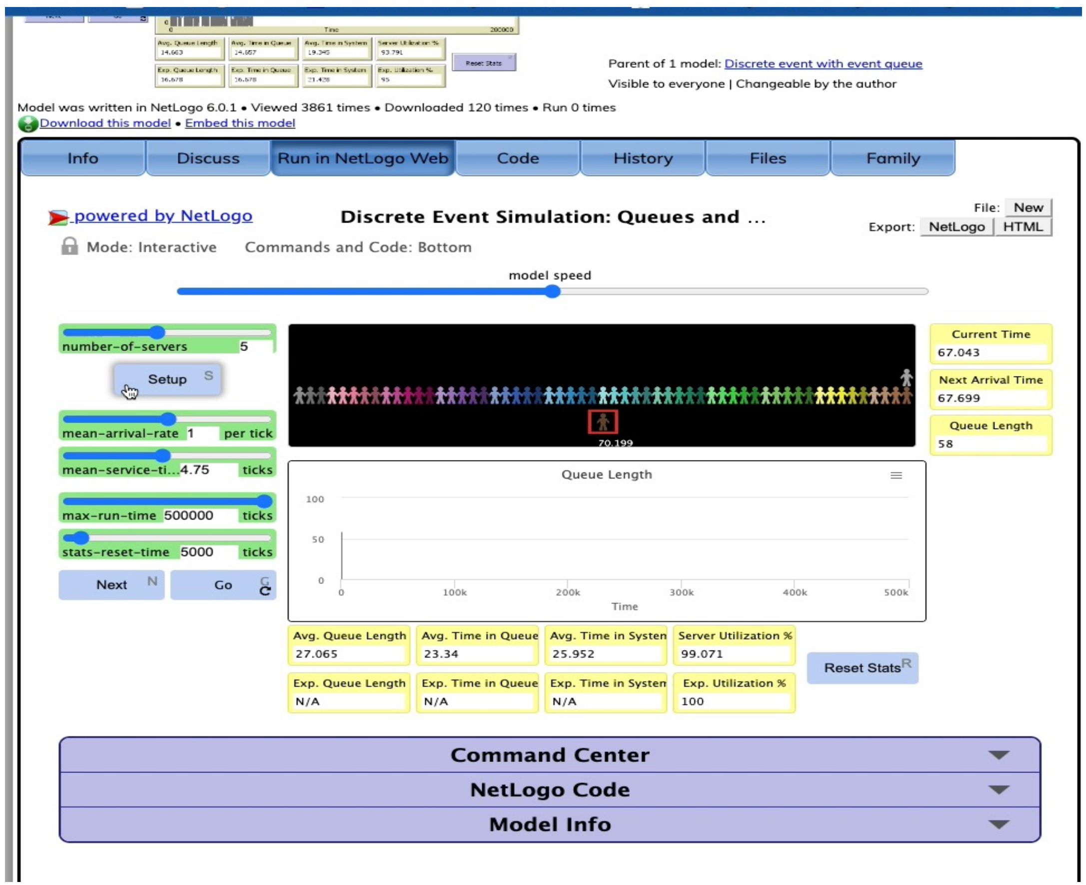
```
time = next_arrival
if busy then
      enqueue (next_service)
else
      busy = true
      service_end = time + next_service
fi
read (next_arrival next_service)
```

The Algorithm…

- Process a service completion

```
time = service_end
if queue_empty then
      busy = false
else
      service_end = time + dequeue()
fi
```

Single Queue and Multi-server Simulation



http://modelingcommons.org/browse/one_model/5086#model_tabs_browse_nlw

Getting Complicated

- What if there is more than one server?
- Two possible situations
  - One queue for all servers (like a bank)
  - One queue per server (like a supermarket)

“One Queue to Serve Them All…”

- One queue for all servers
  - If all servers are busy add arrival to queue
    - Otherwise make one of the idle servers busy
  - When a server finishes, have them serve the head of the queue
    - Otherwise make them idle

Single Queue & Multi-server

- The events we are interested in are now:
  - Customer arrives
  - Server 1 finishes
  - Server 2 finishes
  - …
  - Server n finishes
- How do we keep track of which event will happen next?
- Do we really need to know which servers are busy or can we just keep track of how many are busy?

Single Queue & Multi-server

- Keeping track of servers:
  - Two possible solutions
  - An array of servers with busy[i] and end_time[i]
    - This lets us track who is doing what
    - Finding what happens next is in O(n)
  - A heap of end times (smallest on top)
    - This does not let us track who is doing what
    - Finding what happens next is O(log n)
  - Can we get the best of both worlds?

Single Queue & Multi-server

- Using a heap to represent the events (ascending order to events)
  - If we are clever, we can store all event times on the heap
  - All we need is a second array telling us what is next
  - If we are really clever, we can partition the heap into two parts:
    - The heap itself
    - The idle servers
- How do we manipulate the heap as events occur?

The algorithm

- The main program loop

```
initialise
Repeat
        If heap[0]'s type is service end
          process_service_end
        else
          process_arrival
        fi
until heap is not empty
finish
```

Single Queue & Multi-server

- Customer arrives

```
//make an idle server busy or add the new arrival to the queue
time = heap[0]
read next arrival into heap[0], next_service_time
siftdown(heap)
if n_busy < n_servers then //have an idle server
       n_busy = n_busy + 1
       //allocate the new arrival to an idle server
       heap[n_busy] = time + service_time
       siftup(heap)

else
       enqueue(service_time) //all servers are busy
fi
```

Single Queue & Multi-server

- Server finishes

```
time = heap[0]
//make the server idle
if queue_empty then
      heap[0] = heap[n_busy]
      n_busy = n_busy - 1
Else
      //serve the head of the queue
      heap[0] = time + dequeue()
fi
siftdown(heap)
```

Single Queue & Multi-server

- In summary:
  - Heap Grows if a customer arrives and a server is idle
  - Heap shrinks if a customer is served and the queue is empty
  - Heap stays the same size otherwise
  - Just an example of the idea! Actually, we need to distinguish the type event type of heap[0] each time we read it.
- If we keep a second array (id) initially filled with integers $1 \cdots n$, we can use it to track who is doing what
  - 0 is the next arrival time
  - 1 is server 1's completion time
  - 2 is server 2's completion time
  - …
  - n is server n's completion time

Single Queue & Multi-server

- Customer arrives (updated version)

```
time = heap[0]
read next arrival into heap[0], next_service_time
siftdown(heap)
if n_busy < n_servers then
      n_busy = n_busy + 1
      heap[n_busy] = time + service_time
      siftup(heap, id)
      service_time = next_service_time
else
      enqueue(service_time)
Fi
```

Single Queue & Multi-server

- Server finishes (updated version)

```
time = heap[0]
if queue_empty then
      swap (id[0], id[n_busy])
      heap[0] = heap[n_busy]
      n_busy = n_busy - 1
else
      heap[0] = time + dequeue()
fi
siftdown(heap, id)
```

Single Queue & Multi-server

- Every time we move an entry in the heap we move the corresponding entry in the id array.
- If the top of the id array is a zero, the next event is an arrival
- If the top of the id array is non_zero, the next event is a service completion for server `id[0]`
- The simulation starts with the first arrival time in `heap[0]` and `n_busy = 0`

Single Queues & Multi-server

- Define a record (structure), e.g.

```
Type event = record
    Time: float
    eventType: char
    serviceDuration: float
    …
```

Single Queue & Multi-server

- Customer arrives (version 3)

```
Event evt0, evt1
time= next arrival
read next arrival and service_time into evt1
evt1.eventType = 'a'
heap[0] = evt1
siftdown(heap)  // based on time field
if n_busy < n_servers then
        n_busy = n_busy + 1
        evt0.time = time + evt0.serviceDuration
        evt0.eventType = 's'
        heap[n_busy] = evt0
        siftup(heap)
else
        enqueue(evt0)
fi
```

Single Queue & Multi-server

- Server finishes (version 3)

```
time = heap[0].time
if queue_empty then
      heap[0] = heap[n_busy]
      n_busy = n_busy - 1
else
      heap[0]= dequeue()
      heap[0].time = time + heap[0].serviceDuration
fi
siftdown(heap)
```

An Example

of servers = 1

0.24 0.55
0.59 0.16
0.90 0.07
1.87 0.69
…

process	time	heap	queue
initialization		(0.24, a, 0.55), n_busy=0	
1 (arrival)	0.24	(0.59,a,0.16)(0.24+0.55,s, 0.55]) n_busy=1	
2 (arrival)	0.59	(0.79,s,0.55) (0.90,a,0.07) n_busy=1	(0.59, 0.16)
3 (service)	0.79	(0.90,a,0.07) (0.79+0.16, s, 0.16) n_busy=1	
4 (arrival)	0.90	(0.95,s,0.16) (1.87,a,0.69) n_busy=1	(0.90,0.07)
5 (service)	0.95	(0.95+0.07,s,0.07) (1.87,a,0.69) n_busy=1	
6 (service)	1.02	(1.87, a,0.69) n_busy=0	
7 (arrival)	1.87	(1.87+0.69, s,0.69)	
8 (service)	2.56		

An Example

of servers = 2

0.24 0.55
0.59 0.16
0.90 0.07
1.87 0.69
…

Process	time	heap	queue
initialization		(0.24, a, 0.55), n_busy=0	
1 (arrival)	0.24	(0.59,a,0.16) (0.24+0.55,s, 0.55]) n_busy=1	
2 (arrival)	0.59	(0.75,s,0.16) (0.79,s,0.55) (0.90,a,0.07) n_busy=2	
3 (service)	0.75	(0.79, s, 0.55) (0.90,a,0.07) n_busy=1	
4 (service)	0.79	(0.90,a,0.07) n_busy=0	
5 (arrival)	0.90	(0.97,s,0.07) (1.87,a,0.69) n_busy=1	
6 (service)	0.97	(1.87,a,0.69) n_busy=0	
7 (arrival)	1.87	(1.87+0.69,s,0.69) n_busy=1	
8 (service)	2.56		

Multiple Queues

- In this case we have an array of n queues, 1 per server
- When a customer arrives and all servers are busy we place the customer on one of the queues (which one?)
- When a server finishes we only make them busy if their queue is not empty
- *NOTE: This means that we can have a queue even if a server is idle.*

Multi-Queue & Multi-server

- Customer arrives

```
time = heap[0]
read next arrival into heap[0], next_service_time
siftdown(heap)
//assume the new arrival will be allocated to the idle server
//by default. How to modify the algorithm if not?
if n_busy < n_servers then
      n_busy = n_busy + 1
      heap[n_busy] = time + service_time
      siftup(heap)
      service_time = next_service_time
else
      k = the server selected for adding the new arrival
      enqueue(k, service_time)
fi
```

Multi-Queue & Multi-server

- Server k finishes

```
time = heap[0]
k = the server finishes
if queue_k_empty then
      heap[0] = heap[n_busy]
      n_busy = n_busy - 1
 else
      heap[0] = time + dequeue(k)
fi
siftdown(heap)
```

Priority Queues

- How would we handle the situation where customers are given different service priorities?
  - One queue for each priority empty the highest priority queue first
  - This is only efficient if there are a small number of priorities
- What do we do if each priority may be different?
  - E.g. priority is a float between 0 and 1
  - 0 is the lowest customer priority
  - 1 is the highest priority customer
  - We have an infinite number of different priorities so we can't have a queue for each one.

Priority Queues

- A priority queue is a data structure for maintaining a set of elements, each with an associated value called a key.
- In a normal queue, elements come off in first-in-first-out (FIFO) order, so the first element in the queue is the top element.
- In a priority queue, the element with the largest key is always on the top, no matter what order it or the other elements were inserted.
- Some uses for priority queues:
  - OS scheduling algorithms
  - Huffman's algorithm
  - Service for VIPs

Priority Queues - Basic Operation

- `insert(pQueue, elt)`
  - Insert an element into the queue and place it in the right position in the queue.
- `remove(pQueue, elt)`
  - Extract the element with top priority and remove it from the queue
  - adjust the queue as needed
    - This depends on implementation

Priority Queues - Naïve Implementation

- A naive implementation of these operations is to represent the queue as a linked list L
- Insert just puts the elements onto the linked list

```
procedure naïve-insert (L, key)
    M = new node with key
    listadded(L, M)
    return
```

- This operation is $\Theta(1)$ since we're not worried about keep the list in any order

Priority Queues - Naïve Implementation

- Remove an element via searching for the element

```
procedure remove(L)
    M = L
    max = head (M)
    while M is not empty
        if (head(M) > max)
        max = head(M)
       M = tail(M)
    elihw
    remove (L, max)
    return max
```

- The algorithm is $\Theta(n)$
  - if the list has n elements, this algorithm is $\Theta(n)$ since it must iterate exactly n times.
  - Only $\Theta(1)$ work is needed to delete an element from a reasonably implemented linked list

Priority Queues - Heap Implementation

- We can use a max heap to improve the implementation, since a heap always keeps its maximum element in the first element.

- Insert just puts the elements onto the linked list

```
procedure heap-insert (Heap, key)
    M = new node with key
    Add(Heap, M)
    return
```

- This could possibly go from a leaf up to the root, taking $O(\log n)$ time, compared with $O(1)$ of the naive version.

Priority Queues - Heap Implementation

- Remove the max element from the priority queue

```
procedure heap-remove(Heap)
    max = remove(Heap)
    return max
```

- The algorithm is $O(\log n)$, same as `remove(Heap)`
  - Taking the first element at $O(1)$
  - Maintain the heap property using `siftDown()` is $O(\log n)$

Priority Queues

- The solution is to replace the queue with a heap ordered on priority (descending order)
- Each time we remove a customer from the heap we
  - Move the last entry to the top of the heap
  - Reduce the heap size by one
  - Sift down the top entry (based on the priority)
- Each time we add a customer to the heap we:
  - Increase the heap size by one
  - Add the customer to the end of the heap
  - Sift up the last entry (based on the priority)

Related References

- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 6.4
- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 6.5