YEAH - Patient Queue
Anton Apostolatos

Source: XKCD
Queue?
Queue: order items by when they were placed - first in, first out (FIFO)

Main Functions

void enqueue(string s) // Inserts an element into the queue
string dequeue() // Returns and removes the first element placed
PriorityQueue: order items by priority

Main Functions

void enqueue(string s, int priority)  // Puts element into priority queue
string dequeue()  // Returns and removes the highest-priority item
PatientQueue: order patients by priority

Main Functions

```c
void newPatient(string name, int priority) // Puts person into patient queue
string processPatient() // Returns and removes the highest-priority person
```
**KEEP THIS IN YOUR BAG, EMILY...**

<table>
<thead>
<tr>
<th>DEFCON (US)</th>
<th>UK VERSION</th>
<th>READINESS</th>
<th>WHEN ACTIVATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CRITICAL</td>
<td>Nuclear war imminent</td>
<td>Never</td>
</tr>
<tr>
<td>2</td>
<td>SEvere</td>
<td>Army ready in six hours</td>
<td>Cuban missile crisis, 1962</td>
</tr>
<tr>
<td>4</td>
<td>Moderate</td>
<td>Increased security</td>
<td>Sporadic in Cold War and ‘war on terror’</td>
</tr>
<tr>
<td>5</td>
<td>Low</td>
<td>Normal peacetime</td>
<td>Peacetime readiness</td>
</tr>
</tbody>
</table>

[Image of a missile]
Lower number = higher priority!

Note: time in queue is the tiebreaker

(“Ross” : 3) < ("Chandler" : 4) < ("Pheobe" : 8 )

PatientQueue pq;
pq.newPatient("Sam", 6);
pq.newPatient("Frodo", 3);
pq.newPatient("Pippin", 3);
pq.newPatient("Merry", 1);

cout << pq.processPatient() << endl;
cout << pq.processPatient() << endl;

Console

Merry
Frodo
Main Functions

`PatientQueue()` // Constructor for PatientQueue

`~PatientQueue()` // Destructor for PatientQueue

`void newPatient(string name, int priority)` // Puts element into patient queue

`string processPatient()` // Returns and removes the highest-priority item

`string frontName()` // Name of highest-priority patient

`int frontPriority()` // Priority of highest-priority patient

`void upgradePatient(string name, int newPriority)` // updates patient to
    // higher priority

`void clear()` // Removes all patients

`string toString()` // Returns the PatientQueue as a string
A5: Patient Queue

Unsorted Vector

Sorted Singly-Linked List

Extension: Binary Heap
Unsorted Vector

<table>
<thead>
<tr>
<th>“Pam”</th>
<th>“Dwight”</th>
<th>“Jim”</th>
<th>“Toby”</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>13</td>
<td>3</td>
<td>72</td>
</tr>
</tbody>
</table>
Unsorted and Vector wrapper - Simplest to implement and think about!

newPatient(string name, int priority): append to a vector!
processPatient() / front(): scan vector and find smallest element
A vector of what?!

Struct!

What does the struct store?

Up to you!
Questions?
Sorted Singly-Linked List

“George” 2

“Ringo” 5

“John” 8

“Paul” 9
Draw as you code!
struct PatientNode {
    string name;
    int priority;
    PatientNode* next;
}

// Constructor - each parameter is optional
PatientNode(string name, int priority, PatientNode* next);
You need to create a Linked List and enforce that all elements are stored in order of priority, with time in queue being tiebreaker.

**newPatient():** look for place in the linked list and place there.

**processPatient()/front():** first element!
Tips and Tricks

- Before writing any code, go through simple toy examples by hand to make sure your proposed solution’s logic is sound

- Don’t forget the semicolon after a struct or class definition!

- Bad idea to declare multiple pointers on the same line:

  ```
  Node * head, tail;
  ```
Tips and Tricks: Continued

- Nested structs are weird. If we create a cell inside of PQueue then a helper function that returns a PatientNode* would be declared as:

  \[
  \text{PatientNode}^* \ \text{helperFunction} (\text{PatientNode}^* \ \text{ptr});
  \]

- And would be implemented as

  \[
  \text{PQueue}::\text{PatientNode}^* \ \text{PQueue}::\text{helperFunction} (\text{PatientNode}^* \ \text{ptr});
  \]

- Do your best to make your size functions not O(n)! → how?

- We’ll ask you for Big-O of every function you write!
Questions?
Extension: Binary Heap
Binary Heaps

A heap is a tree-based structure that satisfies the heap property:

Parents have a higher priority than any of their children.
Binary Heaps

• There are two types of heaps:

Min Heap
(root is the smallest element)

Max Heap
(root is the largest element)
There are no implied orderings between siblings, so both of the trees below are min-heaps:
• Circle the min-heap(s):

![Binary Heap Diagram]
• Circle the min-heap(s):
Heaps are completely filled, with the exception of the bottom level. They are, therefore, "complete binary trees":

- **complete**: all levels filled except the bottom
- **binary**: two children per node (parent)
- **height**: $\log(n)$
Binary Heaps

What is the best way to store a heap?

We could use a node-based solution, but…
It turns out that an array works **great** for storing a binary heap!

We will put the root at index 1 instead of index 0 (this makes the math work out just a bit nicer).
Binary Heaps

The array representation makes determining parents and children a matter of simple arithmetic:

For an element at position $i$:
- left child is at $2i$
- right child is at $2i+1$
- parent is at $\lfloor i/2 \rfloor$
Heap Operations

Remember that there are three important priority queue operations:

- **peek()**: return an element of h with the smallest key.
- **enqueue(e)**: insert element e into the heap.
- **dequeueMin()**: removes the smallest element from h.

We can accomplish this with a heap!
Heap Operations: peek()

peek()

Just return the root!

return heap[1]
How might we go about inserting into a binary heap?

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>10</th>
<th>8</th>
<th>12</th>
<th>11</th>
<th>14</th>
<th>13</th>
<th>22</th>
<th>43</th>
</tr>
</thead>
</table>
Heap Operations: enqueue(k)

Insert item at element $array[heap.size() + 1]$ (this probably destroys the heap property)

Perform a “bubble up” operation:
- Compare the added element with its parent
  - if in correct order, stop
  - If not, swap and repeat
Start by inserting the key at the first empty position. This is always at index `heap.size()+1`. 
Heap Operations: enqueue(9)

Start by inserting the key at the first empty position. This is always at index `heap.size()+1`. 
Heap Operations: enqueue(9)

Look at parent of index 10, and compare: do we meet the heap property requirement?
Heap Operations: enqueue(9)

Look at parent of index 10, and compare:
do we meet the heap property
requirement?

No -- we must swap.
Heap Operations: enqueue(9)
Heap Operations: enqueue(9)

Look at parent of index 5, and compare: do we meet the heap property requirement?

No -- we must swap.
Heap Operations: enqueue(9)
Heap Operations: enqueue(9)
Heap Operations: enqueue(9)

No swap necessary between index 2 and its parent. We're done bubbling up!
Demo!

http://www.cs.usfca.edu/~galles/visualization/Heap.html
• How might we go about removing the minimum?

dequeue()
Heap Operations: dequeue()

We are removing the root, and we need to retain a complete tree: replace root with last element.

“bubble-down” or “down-heap” the new root:

- Compare the root with its children:
  - if in correct order, stop.
  - if not, swap with smallest child, and repeat
Heap Operations: dequeue()
Heap Operations: dequeue()

Remove root (will return at the end)
Heap Operations: dequeue()

Move last element (at `heap[heap.size()]`) to the root (this may be unintuitive!) to begin bubble-down
Heap Operations: dequeue()

Compare children of root with root: swap root with the smaller one (why?)
Heap Operations: dequeue()

Keep swapping new element if necessary. In this case: compare 13 to 11 and 14, and swap with smallest (11).
Heap Operations: dequeue()

13 has now bubbled down until it has no more children, so we are done!
Questions?