YEAH - Priority Queue
Anton Apostolatos

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

Functions

void enqueue(string s) // Inserts an element into the queue
string dequeue() // Returns and removes the first element placed
string peek() // Returns the first element placed
int size() // Returns the number of elements
bool isEmpty() // Returns whether the queue is empty
PriorityQueue: order items by rank

Functions

void enqueue(string s) // Inserts an element into the priority queue
string dequeueMin()  // Returns and removes the highest-ranked item
string peek()        // Returns the highest-ranked item
int size()           // Returns the number of elements
bool isEmpty()       // Returns whether the queue is empty
Order is lexicographic/alphabetic!

“Albus” < “Ginny” < “Harry” < “Hermione” < “Ronald” < “Tom Marvolo”

```cpp
PQueue pq;
pq.enqueue("There");
pq.enqueue("And");
pq.enqueue("Back");
pq.enqueue("Again");

cout << pq.dequeue << endl;
cout << pq.dequeue << endl;
```
A5: PQueue

- Unsorted Vector:
  - "Little"
  - "Teresa"
  - "Kevin"
  - "Paula"

- Sorted Singly-Linked List:
  - quokka!
  - dikdik!

- Unsorted Doubly-Linked List:
  - Dora Bianchi
  - Faye Whitaker
  - Marten Reed

- Binary Heap:
  - Represents a binary tree structure with numbers from 1 to 10.
Unsorted Vector

<table>
<thead>
<tr>
<th>&quot;Little&quot;</th>
<th>&quot;Teresa&quot;</th>
<th>&quot;Kevin&quot;</th>
<th>&quot;Paula&quot;</th>
</tr>
</thead>
</table>

Unsorted and **Vector wrapper** - Simplest to implement and think about!

- **Enqueue**: append to a vector!
- **Dequeue/peek**: scan the vector and find the smallest element

```
Tito  Jermaine  Marlon  Michael
```

```
dequeueMin()
```

```
Jermaine
```

```
Tito  Marlon  Michael
```
Sorted Singly-Linked List
Draw as you code!
You need to create a Linked List and enforce that all elements are stored in lexicographic order.

- **Enqueue**: look for its place in the list and place it there.
- **Dequeue/peek**: first element!

```
Gale Hawthorne  →  Katniss Everdeen  →  Peeta Mellark  →  Primrose Everdeen
```

```
Gale Hawthorne  →  Haymitch Abernathy  →  Katniss Everdeen  →  Peeta Mellark  →  Primrose Everdeen
```
Unsorted Doubly-Linked List
**Unsorted**, but every cell now has a **next** and **prev** pointer

**New functionality:** You can splice (remove) a cell without needing to keep a second pointer!

- **enqueue:** prepend new item to the list
- **dequeue/peek:** loop through list to find smallest element
Binary Heap

Slides by Chris Gregg!
A heap is a *tree-based* structure that satisfies the heap property:

*Parents have a higher priority than any of their children.*
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):
Circle the min-heap(s):
Binary Heaps

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…
Binary Heaps

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]
Heap Operations: enqueue(k)

enqueue (k)

How might we go about inserting into a binary heap?
Heap Operations: enqueue(k)

Insert item at element $\text{array}[\text{heap.size()}+1]$ (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
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)

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)
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
Heap Operations: dequeue()

• 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?
Tips and Tricks

- Height of a binary heap is $O(\log(n))$

- 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 Cell* would be *declared* as:

  ```
  Cell* helperFunction(Cell* ptr);
  ```

- And would be *implemented* as

  ```
  PQueue::Cell* PQueue::helperFunction(Cell* 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!
General questions?