Priority queue

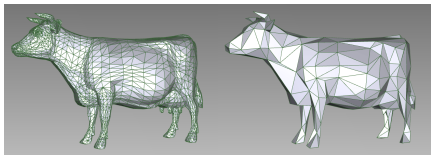
You already know that:

- A *stack* is a list that supports insertion on and removal from its top. It's a Last-In-First-Out, or LIFO, list.
- A *queue* is a list that supports insertion at the end and removal from the beginning. It's a First-In-First-Out, or FIFO, list. The British wait in a queue for their football tickets, Americans wait in a line.

The *priority queue* is another kind of list. Each item in a priority queue has a *priority*. New items are simply added to the list, along with their priority. When removal is performed, the item with the *smallest* priority is removed.

Priority queues are used in a number of applications. We name three:

- In event-driven simulations, events can be kept in a priority queue, with the priority being the time at which an event is to take place. Events are removed from the queue in chronological order and processed.
- Many articles on the web talk about simplifying the representation of objects. In one version, a priority queue contains a bunch of simplifications that can be made, with the priority being the cost of making that simplification. For example, the first cow on the right is represented by a bunch of triangles; the representation is simplified in the second cow with fewer triangles.



- Dijkstra's algorithm to compute the shortest path from a start node to all other nodes of a graph uses a priority queue in which the values are some of the nodes of the graph and the priority of a node is the shortest path found so far from the start node to that node.

Operations on a priority queue

The conventional operations on a priority queue are listed to the right. Method `add` returns true if `e` is actually added to the list and false if `e` is already in the list.

Methods `contains`, `remove`, and `updatePriority` are not always included. Their efficient implementation complicates the data structures usually used to implement a priority queue, as discussed briefly below.

Operations on priority queue of items of type E
int size()
boolean add(e, p) // add item e with priority p
E peek() // return item with minimum priority
E poll() // remove/return item with min priority
void clear() // remove all items

boolean contains(E e)
boolean remove(E e)
void updatePriority(e, p) // change e's priority to p

Implementing a priority queue

One can implement a priority queue naively in a Java `ArrayList` or something similar, but either method `add` or methods `peek` and `poll` will take linear time. That is because finding the item with minimum priority requires a linear search, unless method `add` did at least linear-time work to make `peek` and `poll` faster.

In 1964, J.W.J. Williams invented the *heap* data structure —not to implement priority queues but to develop sorting method heapSort. A *heap* is a tree with certain properties that can be stored in an array. Using a heap, methods `size` and `peek` take time O(1) for a heap of size `n`, while `add` and `poll` take time O(log `n`). Because of this, the heap is used extensively to implement priority queues.

Methods `contains`, `remove`, and `updatePriority` still take linear time unless additional data structures are used, because item `e` has to be found.

Look at JavaHyperText entry *heaps* for an extensive discussion of this important data structure.

When priorities are few

If there are just 2 or 3 (or a few more) different priorities, then the priority queue may be best implemented as a list of normal queues. After all, that is what airlines do at ticket counters. There are two priorities: first-class and everyone-else. First-class customers are in one queue and everyone else is in a second queue. In some applications, this is a good solution to the priority queue problem.

Java's class PriorityQueue

Class `java.util.PriorityQueue` implements an unbounded queue using a heap, To the right, we show some of its methods.

```
public class PriorityQueue<E> {
   public boolean add(E e) { … }
   public E peek() { … }
   public E poll()  { … }
   public void clear()
   public boolean contains(Object ob) { … }
   public boolean remove(Object ob) { … }
   …
```

Interestingly, method `add`, defined to the right, doesn't have a parameter for the priority. That is because either

(1) Type `E` has to implement interface `Comparable`, and its method `compareTo` is used to determine the ordering of items in the queue; OR

(2) When creating a `PriorityQueue` object, one can give the comparator as an argument of the call to the constructor.

Class `PriorityQueue` uses a heap and no other data structure. Therefore, methods `contains` and `remove` take time proportional to the size of the heap.

Class `PriorityQueue` also has an Iterator, allowing one to write a for-each loop over the elements of the queue.