Queues

More with Comparable

Priority Queues
Queueing Theory

Agner Krarup Erlang
Queues
Queue Interface

- **null is returned from** `getFront()` **and** `dequeue()` **when queue is empty**
- **return false from** `offer` **when cannot add to queue.**

```java
public interface QueueInterface<E> {
    int size();
    boolean isEmpty();
    E getFront();  // peek
    boolean enqueue(E e);
    E dequeue();
    void clear();
}
```
## Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>output</th>
<th>Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(5)</td>
<td>TRUE</td>
<td>{5}</td>
</tr>
<tr>
<td>enqueue(3)</td>
<td>TRUE</td>
<td>{5, 3}</td>
</tr>
<tr>
<td>dequeue()</td>
<td>5</td>
<td>{3}</td>
</tr>
<tr>
<td>enqueue(7)</td>
<td>TRUE</td>
<td>{3, 7}</td>
</tr>
<tr>
<td>dequeue()</td>
<td>3</td>
<td>{7}</td>
</tr>
<tr>
<td>getFront()</td>
<td>7</td>
<td>{7}</td>
</tr>
<tr>
<td>dequeue()</td>
<td>7</td>
<td>{}</td>
</tr>
<tr>
<td>dequeue()</td>
<td>null</td>
<td>{}</td>
</tr>
</tbody>
</table>
Array-based Queue

- An array of size $n$ in a circular fashion
  - `frontLoc`: index of the front element
    - where objects are read
  - `count`: number of stored elements
  - `rearLoc`: index of rear element
    - where objects are added

\[
\begin{array}{cccccccc}
Q & 0 & 1 & 2 & f & r \\
\hline
\text{normal configuration} & & & & & & & \\
\text{wrapped-around configuration} & & & & & & & \\
0 & 1 & 2 & r & f \\
\end{array}
\]
Circular Array and Queue

- Normal configuration
  - Queue `Q` with elements `0`, `1`, `2`, `f`, `r`.

- Wrapped-around configuration
  - Queue `Q` with elements `0`, `1`, `2`, `r`, `f`. Elements wrap around.
Performance and Limitations for array-based Queue

• Performance
  ▫ let \( n \) be the number of objects in the queue
  ▫ The space used is \( O(n) \)
  ▫ Each operation runs in time \( O(1) \)

• Limitations
  ▫ Max size is limited and can not be changed
  ▫ Adding to a full queue returns false (enqueue method)
public class ArrayQueue<Q> implements QueueInterface<Q> {
    private static final int DEFAULT_CAPACITY = 42;
    private Q[] backingArray;
    private int count;
    private int frontLoc;

    public ArrayQueue() {
        this(DEFAULT_CAPACITY);
        count = 0;
        frontLoc = 0;
        backingArray = (Q[]) new Object[DEFAULT_CAPACITY];
    }
    public ArrayQueue(int qSize) {
        count = 0;
        frontLoc = 0;
        backingArray = (Q[]) new Object[qSize];
    }
    public int size() {
        return count;
    }
}
Queue offer Method (enqueue)

boolean offer(E e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted queue, this method is generally preferable to add(E), which can fail to insert an element only by throwing an exception.

Parameters:
e - the element to add

Returns:
true if the element was added to this queue, else false
public class ArrayQueue<Q> implements QueueInterface<Q> {
    private static final int CAPACITY = 42;
    private Q[] backingArray;
    private int count;
    private int frontLoc;
    
    boolean enqueue(Q e); // add item to queue   // write with class
    Q dequeue(); // remove item from queue   // write in groups
public class Sal<C> {

• Problem
  • how to guarantee that the Generic class C has an ordering ...
    • Homework 4 convert to string and compare those strings
  • That method is less than optimal .... Why?
• It would be better to require that items have an ordering
  • or at least that items know ordering with respect to each other.

• In Java — require the Comparable interface
  
  public class Sal<C extends Comparable<C>> {
Comparable Interface

• Part of Java language
• Idea, give a way for classes to define a total ordering of instances
• Java classes that implement:
  • String
  • All descendants of Number
  • Lots of others
The Comparable Interface

- public interface Comparable<T>
  This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method. Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.

  The natural ordering for a class C is said to be consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

  It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.

  For example, if one adds two keys a and b such that (!a.equals(b) && a.compareTo(b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

  Virtually all Java core classes that implement Comparable have natural orderings that are consistent with equals. One exception is java.math.BigDecimal, whose natural ordering equates BigDecimal objects with equal values and different precisions (such as 4.0 and 4.00).

  For the mathematically inclined, the relation that defines the natural ordering on a given class C is: 

  $$\{ (x, y) \text{ such that } x.\text{compareTo}(y) \leq 0 \}.$$
int compareTo(T o)
Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementor must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation sgn(expression) designates the mathematical signum function, which is defined to return one of -1, 0, or 1 according to whether the value of expression is negative, zero or positive.

Parameters:
o - the object to be compared.
Returns:
a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
Comparable Interface
(even shorter)

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```

- return 0 if they are equal
- return <0 if caller is less than compared
- return >0 if caller greater than compared
  - Integer v3 = Integer.valueOf(3).
  - v3.compareTo(4) ==> -1
  - "DQ".compareTo("DR") ==>- 3
Comparable Rabbit

```java
public class Rabbit implements Comparable<Rabbit> {
    private final int ID;
    private final String nickname;
    public Rabbit(int id, String nn) {
        this.iD = id;
        this.nickname = nn==null ? makeName() : nn;
    }

    // implement Comparable interface so that rabbits
    // are sorted based on their ID.
```
Priority Queue

• Rather than FiFo, remove items according to their priority
  • Implement:
    • same methods as queue(?)
  • Others needed?

```java
public class PriorityQueue<B extends Comparable<B>>
    extends ArrayList<B>
    implements QueueInterface<B>

public class PriorityQueueSAL<P extends Comparable<P>>
    extends SALextending<P>
    implements QueueInterface<P>
```
PriorityQueue

- Implementation a trivial extension on SAL!!!
- Small difference
  - Usually PQ are on K,V pairs where
    - K — the priority
    - V — the item in the queue
- Q: Does K,V pair matter?