CSE 222 HOMEWORK #6

Orhan Aksoy

09104302

TABLE OF CONTENTS

1. Problem Description

Instead of the regular queue implementation which works with a pure FIFO mechanism, a priority queue is to be designed where each item can be assigned priorities. The only extra requirement for the priority queue over the regular queue is that the items with higher priorities should be processed before the items with lower priorities.

1.1. Requirements

1.1.1. The project shall be comprised of a test application which drives a priority queue interface, and a generic priority queue class definition that implements this interface for any type provided by the user.

1.1.2. The queue shall accept new items, together with a priority number assigned to them.

1.1.3. The queue shall allow peeking at the front of the queue

1.1.4. The queue shall allow removing an item from the front of the queue.

1.1.5. Both peeking and removing methods shall have two versions. The first version returns *null* if the queue is empty and the other generates a *NoSuchElementException*.

1.1.6. The item at the front of the queue shall not have a priority less than any item in the queue.

1.1.7. The items with the same priorities shall be ordered in the queue with respect to their time of arrival to the queue. The first item getting into the queue shall come before the other items which came later.

2. Analysis

In a regular queue, the classic FIFO implementation would suffice, where a simple array could successfully handle the queue operations with a little bit of customization:

Adding a new item at the end of the queue could be a simple ‘add’ command for an array. To remove from the front of the queue, a ‘remove’ call would be enough. However, the memory reallocation overhead would make this mechanism inefficient, since this reallocation would be repeated continuously as new items get into the queue in time.

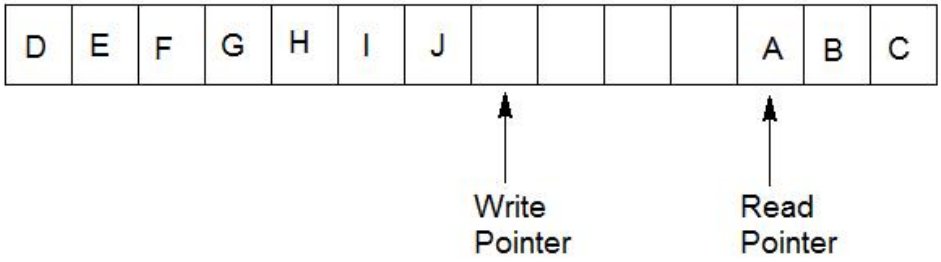
A better approach is to use a simple java array with an initial size. On this array, two pointers are to be used, one of which point to the point where a new item will be inserted in the next addition; the other pointer points to the cell on the array where the next 'remove/peek' shall return. When a pointer meets the end of the java array, the pointer jumps back to the beginning (circular operation). If addition operation increments the write pointer and the write pointer overlap with the read pointer, this means that the queue is full. In this case, the array size is doubled, and the whole mechanism works the same way as before.

In case of a priority queue, an ordered array of circular queues could be used. In this case, each queue in the array shall represent a unique priority. The addition method ensures that the queue array is ordered with respect to priorities, and the removing/peeking methods start using the first queue in the array.

3. Design and Implementation

3.1. Circular Queue Implementation

The memory organization of the simple circular queue described in the Analysis section is as follows:



A circular queue. The next peek / remove operation will return 'A', and the read pointer shall proceed to 'B'. The next add operation will write next to J and the write pointer will move one cell to the right.

As described on the picture, successive remove and add operations will move the read and write pointers to the right. When a pointer hits the end of the array, it jumps back to the beginning of the array.

A problem occurs when the remove operations are not frequent enough to read all the items that the add operations put into the queue. In this case, the read and write pointers overlap just after an add operation. When this happens, a new array is allocated with double size, and

the current array content is moved to the new array, and the pointers are set to the correct values.

3.2. Priorities Handled by Multiple Circular Queues

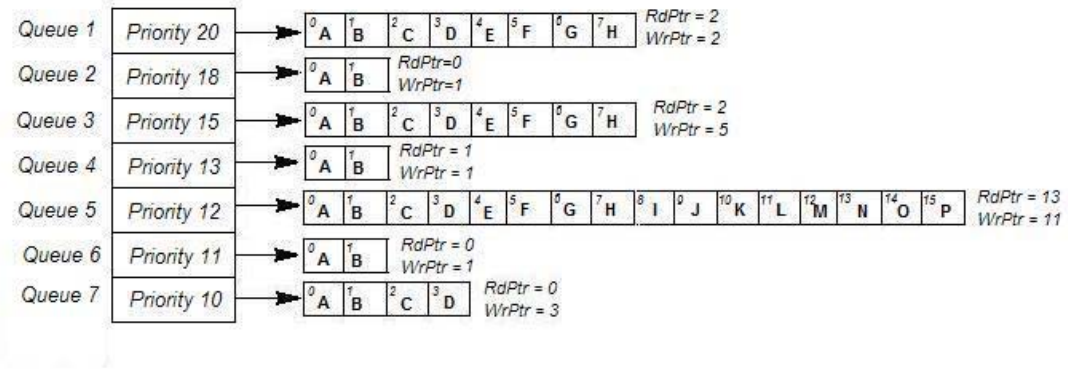
In order to enhance this mechanism to handle items with priorities, an ordered array of circular queues can be used as described in the analysis section. In this solution, an array of circular queues is created with no content. Each circular queue has unique priority assigned to it and every new item that has this priority is handled by this queue. The array is always kept ordered according to their priorities, so the item at the front of the first queue will be used in the first remove/peek operation. The mechanism works as follows:

When the first item gets added to the queue, a new circular queue is created with a priority attribute set to the item's priority. From that point on, every new item with this priority will use this circular queue. This queue is added to the queue array as the first queue.

When a new item gets added, its priority is checked against all stored queue priorities. If a match is found, the item is added to the end of that circular queue. If a match is not found, a new queue is created and put in the correct place in the queue array to keep it ordered.

During peeking/removal, the first queue in the array is used first. When this queue is empty, the second queue can be used.

The memory organization of this solution in an example is as follows:



The front of the queue is the front of the first non-empty queue. The priority 20 queue is empty, since the read and writes pointers are overlapping. The next queue has a priority of 18, and the queue is not empty. SO, he item at the front of the queue is 'A' at position 0 of Queue 2.

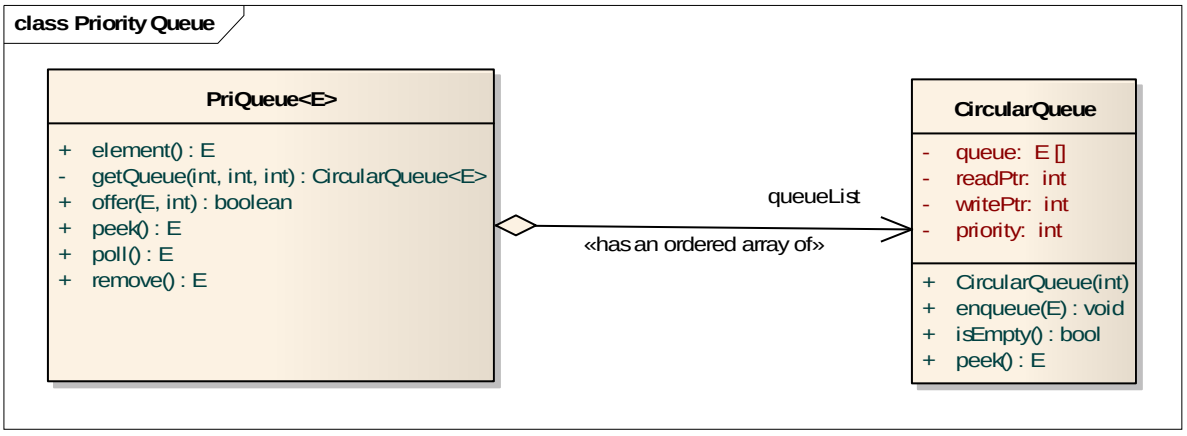
The states of each queue are explained as follows:

- In Queue 1, both the read and write pointers point to position 2. This means that there's no more item on the queue. The last item that was removed was "B". The next write will be at position 2.
- In Queue 2, the write pointer points to position 1, but read pointer points to position 0. This means that the next remove will return "A" and the read pointer will move to 1. The next write will be at position 0.
- In Queue 3, there are 3 more items to be read: C, D and E.
- In Queue 4, both read and write pointers point at the same point, so there's no item to read.
- In Queue 5, the read pointer points to 13, and write pointer to 11. This means that the following items to be returned are: N, O, P, A, B, C, D, E, F, G, H, I, J and K.
- In Queue 6, the read pointer points to 0 and write pointer to 1. If a new add is attempted on the queue, the item will be written at position 1, and the write pointer will move to position 0. After the write operation, the overlapping of read and write pointers will initiate reallocation of the circular queue, and the size will be doubled.
- In Queue 7, the next three removes will return A, B and C.

3.3. Implementation

The priority queue is implemented using two classes: *PriQueue* class and an inner class *CircularQueue*.

CircularQueue class provides the functionality described in section 3.1. Class PriQueue<E> is a generic class that contains an array of circular queues. It provides the queue interface specified in the textbook.



3.4. Algorithms

3.4.1. Adding algorithm (Method *offer*)

The algorithm for the *offer* method is as follows:

```
If the queue array size is zero
        Create a new queue, make it the first item in the queue list and put the item in this queue
Else
        Find the queue with the same priority as the input item.
        (Use binary search in the aray as the array is ordered)
        If an array is found with the same priority
                Enqueue the item in this queue
        Else
                Create a queue with the priority of the input item.
                Enqueue the item in this queue
        End if
End if
```

3.4.2. remove() and element() methods' algorithms

The algorithm for the *remove* and *element* methods is as follows:

```
Find the first non-empty queue in the queue array
If there's no non-empty queue
        Throw NoSuchElementException
Dequeue the first item and return it.
if remove() is called
        remove the first item
```

3.4.3. poll() and peek() methods' algorithms

The algorithm for the *remove* and *element* methods is as follows:

```
Find the first non-empty queue in the queue array
If there's no non-empty queue
        Return null
Dequeue the first item and return it.
if poll() is called
        remove the first item
```

3.4.4. Circular Queue : enqueue algorithm

The algorithm for the *enqueue* methods is as follows:

Put the item at the cell pointed by the 'writePtr' pointer.
Increment the writePtr pointer. If it is equal to the size of the array, reset it to zero..
If the write pointer overlaps with the read pointer (after write operation only)
    Reallocate a new array with double size.
    Transfer the items from the old array starting from the read pointer until the end of the array.
    Keep on transferring from the beginning until the write pointer.
    Make readpointer to zero, and write pointer to the length of the old array
End if.

3.4.5. Circular Queue : dequeue algorithm

The algorithm for the *dequeue* methods is as follows:

If the read pointer equal to the write pointer, return false.
Return the item pointed by the 'read' pointer and increment the read pointer.
If the read pointer is equal to the size of the array, reset it to ero.

4. Testing

All tests are done in 'main' with different titles. A single run of the application will execute all tests (Test 1 to test 6). The outputs of all tests can be observed consecutively in a single run.

4.1. Simple queue functionality

Three items (5, 7 and 8) are enqueued with the same priority. The same order is observed upon remove calls.

// Test 1 code snippet:

```
q.offer(5, 1);
q.offer(7, 1);
q.offer(8, 1);

while (q.peek() != null) {
   System.out.println("Test1: Dequeue: " + q.poll());
}
```

4.2. Priority queue functionality

Three items (5, 7 and 8) are enqueued with different priorities (1, 5 and 3 respectively). The items are observed with the order of their priorities, irrespective of their arrival order:

// Test 2 code snippet:

```
q.offer(5, 1);
q.offer(7, 5);
q.offer(8, 3);

while (q.peek() != null) {
   System.out.println("Test2: Dequeue: " + q.poll());
}
```

4.3. Per queue array reallocation.

Although the initial array size of queues' are 4, five items (5, 10, 15, 22, 44) are enqueued with the same priority (3). The same items are observed in their arrival order.

// Test 3 code snippet:

```
q.offer(5, 3);
q.offer(10, 3);
q.offer(15, 3);
q.offer(22, 3);
q.offer(44, 3);

while (q.peek() != null) {
   System.out.println("Test3: Dequeue: " + q.poll());
}
```

4.4. remove/element methods throwing NoSuchElementException.

Element() and remove() methods are called on empty queues, and exception are observed:

// Test 4 code snippet:

```
try {
   q.offer(5, 3);
   q.remove();
   q.remove();
} catch (Exception e) {
   System.out.println("Exception received during remove() : " + e.getMessage());
}

try {
   q.offer(5, 3);
   q.remove();
   q.element();
} catch (Exception e) {
   System.out.println("Exception received during element() : " + e.getMessage());
}
```

4.5. peek/poll methods returning null on empty queues.

poll() and peek() methods are called on empty queues, and return value are observed:

// Test 5 code snippet:

```
System.out.println ("q.peek() returned " + q.peek());
System.out.println ("q.poll() returned " + q.poll());
```

4.6. Priority queue operation

Different items with different priorities are added, and the prioritized output is observed:

// Test 6 code snippet:

```
q.offer(3, 6);
q.offer(4, 3);
q.offer(5, 6);
q.offer(6, 3);
q.offer(300, 14);
q.offer(7, 6);
q.offer(8, 3);
q.offer(9, 6);
q.offer(100, 9);
q.offer(10, 3);
q.offer(11, 6);
q.offer(12, 3);
q.offer(200, 12);
q.offer(13, 6);
q.offer(14, 3);
q.offer(400, 14);

while (q.peek() != null) {
   System.out.println("Test 6 : Dequeue : " + q.poll());
}
```

The expected output is:

Test 6 : Dequeue : 300
Test 6 : Dequeue : 400
Test 6 : Dequeue : 200
Test 6 : Dequeue : 100
Test 6 : Dequeue : 3
Test 6 : Dequeue : 5
Test 6 : Dequeue : 7
Test 6 : Dequeue : 9
Test 6 : Dequeue : 11
Test 6 : Dequeue : 13

Test 6 : Dequeue : 4
Test 6 : Dequeue : 6
Test 6 : Dequeue : 8
Test 6 : Dequeue : 10
Test 6 : Dequeue : 12
Test 6 : Dequeue : 14