# Random Queue

Dennis Sadeler Shapira Jonas Lomholdt

February 22, 2015

## Contents

**1 The RandomQueue data structure** **1**

**2 Method descriptions** **1**

2.1 RandomQueueIterator inner class . . . . . . . . . . . . . . . . . . . . 2

**3 Testing** **3**

## 1 The RandomQueue data structure

Our RandomQueue is essentially a generic collection of items. Initially when constructing a RandomQueue object we are instantiating two fields consisting of an array field containing slots available for element insertion and an int field specifying the size of the RandomQueue.

In the constructor we are by default instantiating an array of size two. When the client enqueues an item the array is automatically resized to double its size, if its capacity is reached. Also, if the client dequeues elements from the RandomQueue, it will automatically resize itself, by shrinking to half its size, when there are only 1/4 of elements left. Both enqueueing and dequeueing is done at random, by swapping a random element with the last index element in the RandomQueue. The RandomQueue data structure is iterable, and implements the iterable interface. When invoked, the iterator will return a representation of the RandomQueue in shuffled/random order.

## 2 Method descriptions

`isEmpty` Returns a boolean of true or false if the the size of the RandomQueue is equal to 0.

`size` Returns the size of the RandomQueue that is stored in the field rq_size.

`getRandomIndex` The getRandomIndex only contains a conditional statement that return either 0 if the size of the elements in the RandomQueue is equal to 0 (the RandomQueued size is specified in the passing parameter of the method) or a random integer between 0 and the size of RandomQueue.

`resize` The resize method initially creates a new RandomQueue object with the size that is specified by the received parameter (the resize method is called from either the enqueue or dequeue method). The original RandomQueue is then looped through and each item is copied from the specific index into the same index in the newly created RandomQueue.

`enqueue` The enqueue method selects a random index in the RandomQueue and moves the item stored at that index, to the end of the queue. The item that is about to be inserted, is then stored at the random indexed spot, thus obtaining randomness while enqueueing.

`dequeue` The dequeue method works very similar to the enqueue method. A random index is selected from the RandomQueue. The last item in the queue (here understood as the last item in the array) is then inserted at the random index location, while still maintaining a reference to the object that was located at the random index to begin with. We can now remove the reference to the last object in the array (since it has been moved to the random index), and set it's value to null to avoid loitering. The size count is then decremented, and the randomly selected object is returned.

`sample` In the sample method the getRandomIndex method is called(see getRandomIndex method description) which returns a random integer that is used to get an element from RandomQueue, this element is consequently returned.

`toString` The toString method loops through the array of elements and consecutively concatenates them into a string which is then returned.

`iterator` Returns a RandomQueueIterator.

## 2.1 RandomQueueIterator inner class

`RandomQueueIterator` The RandomQueueIterator constructor creates a temporary array of exactly the size of elements contained in the RandomQueue. Each element from the original queue is copied over to the temporary queue at random. This is done by choosing a random index relative to the elements contained in the temporary array, and swapping the last indexes and randomly selected element.

`hasNext` Returns either true orr false if the currentIndex is less than the RandomQueue size.

`next` Since the currentIndex field specifies the pointer in the iteration, the next method returns the element from the RandomQueue with that index and makes sure to increment currentIndex as well.

## 3 Testing

For testing we have implemented a `toString` method, that returns a string representation of the RandomQueue. Since we know how the queue is supposed to act behind the scenes in theory, we can visually confirm that our initial hypothesis holds, by running a series of tests. The queue should be able to dequeue at random, sample at random and be randomly iterable for each instantiated iterator. By running the provided client code we can confirm that this is the case. After each "test" provided in the client code, we simply printed the queue, to confirm that it was acting accordingly (e.g. when dequeueing the element must be deleted from the queue). Again we can visually confirm that our assumptions about it holds. The client code also provides an iterator test. In the iterator test we can confirm that we are iterating at random by noticing, that each colour from the first shuffle, and the two remaining colours are in different order than the entire second shuffle. Based on the above, we believe that our data structure is complete in its entirety, and that we are not violating any of the time complexity restraints that was given to us.