CMSC 132, Object-Oriented Programming II **Summer 2016**

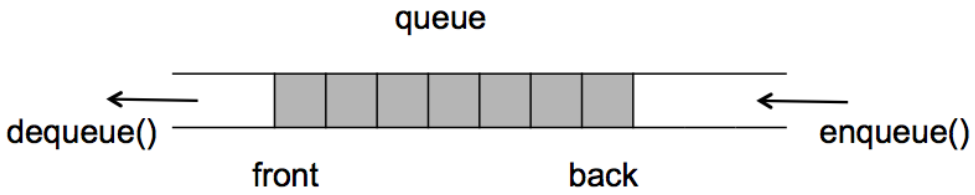
Lecture 9:

Lecturer: Anwar Mamat

9.1 QUEUE

Queue is a First-In-First-Out (FIFO) data structure, in which the first element added to the queue will be the first one to be removed.

Figure 9.1: Queue



Listing 1: Queue Interface

```
1 public interface Queue<T> extends Iterable<T> {
2     void enqueue(T t);
3     T dequeue();// throws EmptyStackException;
4     T peek() ;//throws EmptyStackException;
5     boolean isEmpty();
6     int size();
7 }
```

If we have a Queue Q:

```
1 Q.enqueue(10);
```

```
1 [  ]
2 [10] <-- First
```

If we add numbers 20,30

```
1 Q.enqueue(20);
2 Q.enqueue(30);
```

```
1 [  ]
2 [10] <-- First
3 [20]
4 [30] <-- Last
```

If we pop a number:

```
Q.dequeue();
```

```
[  ]
[20]<-- First
[30] <--Last
```

Peek returns the top item, but it does not remove the item. If we pop a number:

```
int i = Q.peek();//returns the top item;
```

```
[  ]
[20]<-- top
[10]
```

9.1.1 Array based implementation of Queue

In this section, we use a resizable array to implement queue. Array "Items" holds the elements of the stack. items[first] and items[last] always point to the head and tail of the queue. When an item is pushed, the item is stored in items[last]. to remove an item, items[first] will be removed. To avoid moving array elements when the items[first] is removed, a circular array is used. when "first==array.length", then "first = first%array.length", so that it wraps around. Same rule applies to "last".

Listing 2: Array Based Queue

```
import java.util.Iterator;
import java.util.NoSuchElementException;
/**
 *  Based on the implementation by author Robert Sedgewick and Kevin Wayne
 */
public class ArrayQueue<E> implements Queue<E> {
    private E[] q;             // queue elements
    private int N = 0;           // number of elements on queue
    private int first = 0;       // index of first element of queue
    private int last  = 0;       // index of next available slot

    /**
     * Initializes an empty queue.
     */
    public ArrayQueue() {
        // cast needed since no generic array creation in Java
        q = (E[]) new Object[2];
    }

    /**
     * Is this queue empty?
     * @return true if this queue is empty; false otherwise
     */
    public boolean isEmpty() {
        return N == 0;
    }

    /**
     * Returns the number of items in this queue.
     * @return the number of items in this queue
     */
    public int size() {
        return N;
    }

    // resize the underlying array
```

```
37      private void resize(int max) {
38          assert max >= N;
39          E[] temp = (E[]) new Object[max];
40          for (int i = 0; i < N; i++) {
41              temp[i] = q[(first + i) % q.length];
42          }
43          q = temp;
44          first = 0;
45          last  = N;
46      }
47
48      /**
49       * Adds the item to this queue.
50       * @param item the item to add
51       */
52      public void push(E item) {
53          // double size of array if necessary and recopy to front of array
54          if (N == q.length) resize(2*q.length);   // double size of array
55          q[last++] = item;                        // add item
56          if (last == q.length) last = 0;          // wrap-around
57          N++;
58      }
59
60      /**
61       * Removes and returns the item on this queue that was least recently added.
62       * @return the item on this queue that was least recently added
63       * @throws java.util.NoSuchElementException if this queue is empty
64       */
65      public E pop() {
66          if (isEmpty()) throw new NoSuchElementException("Queue underflow");
67          E item = q[first];
68          q[first] = null;                         // to avoid loitering
69          N--;
70          first++;
71          if (first == q.length) first = 0;        // wrap-around
72          // shrink size of array if necessary
73          if (N > 0 && N == q.length/4) resize(q.length/2);
74          return item;
75      }
76
77      /**
78       * Returns the item least recently added to this queue.
79       * @return the item least recently added to this queue
80       * @throws java.util.NoSuchElementException if this queue is empty
81       */
82      public E peek() {
83          if (isEmpty()) throw new NoSuchElementException("Queue underflow");
84          return q[first];
85      }
86
87      /**
88     * Returns an iterator that iterates over the items in this queue in FIFO order.
89     * @return an iterator that iterates over the items in this queue in FIFO order
90      */
91      public Iterator<E> iterator() {
92          return new ArrayIterator();
93      }
94
95      private class ArrayIterator implements Iterator<E> {
96          private int i = 0;
97          public boolean hasNext()  { return i < N;                               }
98          public void remove()      { throw new UnsupportedOperationException();  }
99
100         public E next() {
101             if (!hasNext()) throw new NoSuchElementException();
102             E item = q[(i + first) % q.length];
103             i++;
104             return item;
```

```
105         }
106     }
107
108    /**
109      * Unit tests the ArrayQueue data type.
110      */
111     public static void main(String[] args) {
112         Queue<Integer> q = new ArrayQueue();
113         for(int i = 1; i <= 5; i++){
114             q.push(i);
115         }
116         while(!q.isEmpty())
117         {
118             System.out.print(q.peek()+",");
119             System.out.print(q.pop()+",");
120         }
121         System.out.println("\n");
122     }
123
124 }
```

9.1.2 Linked List based implementation of Queue

In this section, we implement the queue using singly linked list. We always add and remove the nodes at the head of the linked list.

Listing 3: Linked List Based Queue

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3  /**
4  *  Based on the implementation of Robert Sedgewick and Kevin Wayne
5   */
6  public class LinkedQueue<E> implements Queue<E> {
7      private int N;         // number of elements on queue
8      private Node first;    // beginning of queue
9      private Node last;     // end of queue
10
11     // helper linked list class
12     private class Node {
13         private E item;
14         private Node next;
15     }
16
17     /**
18      * Initializes an empty queue.
19      */
20     public LinkedQueue() {
21         first = null;
22         last  = null;
23         N = 0;
24     }
25
26     /**
27      * Is this queue empty?
28      * @return true if this queue is empty; false otherwise
29      */
30     public boolean isEmpty() {
31         return first == null;
32     }
33
34     /**
35      * Returns the number of items in this queue.
36      * @return the number of items in this queue
```

```
37      */
38     public int size() {
39         return N;
40     }
41
42     /**
43      * Returns the item least recently added to this queue.
44      * @return the item least recently added to this queue
45      * @throws java.util.NoSuchElementException if this queue is empty
46      */
47     public E peek() {
48         if (isEmpty()) throw new NoSuchElementException("Queue␣underflow");
49         return first.item;
50     }
51
52     /**
53      * Adds the item to this queue.
54      * @param item the item to add
55      */
56     public void enqueue(E item) {
57         Node oldlast = last;
58         last = new Node();
59         last.item = item;
60         last.next = null;
61         if (isEmpty()) first = last;
62         else           oldlast.next = last;
63         N++;
64     }
65
66     /**
67      * Removes and returns the item on this queue that was least recently added.
68      * @return the item on this queue that was least recently added
69      * @throws java.util.NoSuchElementException if this queue is empty
70      */
71     public E dequeue() {
72         if (isEmpty()) throw new NoSuchElementException("Queue␣underflow");
73         E item = first.item;
74         first = first.next;
75         N--;
76         if (isEmpty()) last = null;   // to avoid loitering
77         return item;
78     }
79
80     /**
81      * Returns a string representation of this queue.
82      * @return the sequence of items in FIFO order, separated by spaces
83      */
84     public String toString() {
85         StringBuilder s = new StringBuilder();
86         for (E item : this)
87             s.append(item + "␣");
88         return s.toString();
89     }
90  /**
91      * Returns an iterator that iterates over the items in this queue in FIFO order.
92      * @return an iterator that iterates over the items in this queue in FIFO order
93      */
94     public Iterator<E> iterator()  {
95         return new ListIterator();
96     }
97
98     // an iterator, doesn't implement remove() since it's optional
99     private class ListIterator implements Iterator<E> {
100        private Node current = first;
101
102        public boolean hasNext()  { return current != null;                     }
103        public void remove()      { throw new UnsupportedOperationException();  }
104
```

```
105         public E next() {
106             if (!hasNext()) throw new NoSuchElementException();
107             E item = current.item;
108             current = current.next;
109             return item;
110         }
111     }
112      /**
113      * Unit tests the LinkedQueue data type.
114      */
115     public static void main(String[] args) {
116         Queue<Integer> q = new LinkedQueue();
117         for(int i = 1; i <= 5; i++){
118             q.enqueue(i);
119         }
120         while(!q.isEmpty())
121         {
122             System.out.print(q.peek()+",");
123             System.out.print(q.dequeue()+",");
124         }
125         System.out.println("\n");
126     }
127 }
```