Queues

Outline and Required Reading:

- Queues (§ 4.2 except 4.2.4)
Queue ADT

Queue – linear data structure organized according to “first-in/first-out” (FIFO) principle

- queue items are removed in exactly the same order as they were added to the queue
- item access and deletion is restricted to the first element (front), while item insertion is restricted to the last place in the queue (rear)
- queue is open at its ends - items cannot be added in the middle

Examples – print server, scheduler in an OS, scheduler in automated on-line systems, packet forwarding in networks

“flow through pipe”
Queue ADT: Interface

**Fundamental Methods**
(Transformers)

- `public void enqueue(Object element);`  /* insert an element at the rear of the queue */
- `public Object dequeue();`  /* remove the element at the front of the queue */  /* QueueEmptyException if stack is empty */

**Supporting Methods**
(Observers)

- `public int size();`  /* return the # of objects in the queue */
- `public boolean isEmpty();`  /* return true if the queue is empty */
- `public Object front();`  /* get the front element without removing it */  /* QueueEmptyException if stack is empty */
Possible Queue ADT Errors

(1) **underflow** - trying to dequeue() or front() on an empty queue
   - can occur regardless of the underlying data structure employed

(2) **overflow** - trying to enqueue() to an already full queue
   - can occur **only** if the underlying data structure has a fixed capacity
Queue ADT: Array-Based Implementation

**Trivial Array-Based Implementation** – let Q[0] be the front of the queue (index of the queue front = 0, at all times)

**Drawback** – too costly! on every dequeue, all remaining elements must be moved forward (Θ(n) run time)

Explain why, in this case, dequeue() has Θ(n), not O(n), run time!
Queue ADT: Array-Based Implementation (cont.)

“Wrap Around” Implementation

- when the rear index reaches the end of array start using available locations at the front
- \( f \) — index to the cell of Q storing the front
- \( r \) — index to the next available cell of Q (rear +1)

- rules for incrementing \( r \) and \( f \) in circular array:
  - enqueue: \( r = (r + 1) \mod N \)
  - dequeue: \( f = (f + 1) \mod N \)
Queue ADT: Array-Based Implementation (cont.)

“Wrap Around”

Problems

(1) $r < f$ - how to determine Q size ?

(2) $f = r$ - is Q empty or full ?

```
      0  1  2  3  4  5
      f  1  2  3  4  5
      r  1  2  3  4  5

size = r - f
```

```
      0  1  2  3  4  5
      0  1
      r  1

size = ?
```

```
      0  1  2  3  4  5
      0  1  2  3  4  5
      f  r  1

full, empty ?
```

“Wrap Around”

Solutions

(1) max allowed queue size: (N-1) if $r > f$, works fine, i.e. N has no effect

(2) size: $size = (r + N - f) \mod N$ if $r < f$ implies that, in fact, there is an N in between, hence (r+N)
public class ArrayQueue implements Queue {

    public static final int CAPACITY = 1000;
    private Object Q[];
    private int f,r;

    public ArrayQueue() {
        this(CAPACITY); }

    public ArrayQueue(int cap) {
        Q = new Object[capacity];
        f=0;
        r=0; }

    public int size() {
        if (r>f) return (r-f);
        else return (r+q.length-f); }

    public boolean isEmpty() {
        return (r == f); }

    cont.
public void enqueue(Object obj) throws QueueFullException {
    if (size() == q.length-1)
        throw new QueueFullException("Queue overflow.");
    Q[r] = obj;
    r = (r+1)%Q.length;
}

public Object front() throws QueueEmptyException {
    if (isEmpty())
        throw new QueueEmptyException("Queue is empty.");
    return Q[f];
}

public Object dequeue() throws QueueEmptyException {
    Object elem;
    if (isEmpty())
        throw new QueueEmptyException("Queue is empty.");
    elem = Q[f];
    Q[f] = null;
    f = (f+1)%Q.length
    return elem; }

Avoided in LL implementation.
Queue ADT: Performance of Array-Based Implement.

**Run Time** – Good! all methods run in constant $O(1)$ time (no loops or recursions)

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>front</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>enqueue</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Space Usage** – Poor! $O(N)$, where $N$ – array size, $n$ – current # of elements in the queue, $N>n$

**General Note** – problems arise if attempting to enqueue $N$ objects
Queue ADT: Singly Linked List - Based Implementation

Instance Variables

private Node head;
/* reference to the head of the SLL */

private Node tail;
/* reference to the tail of the SLL */

private int size;
/* current number of elements in the queue */

Choose: front of Q ↔ head of SLL, rear of Q ↔ tail of SLL. Why?!
public void enqueue(Object obj) {
    Node node = new Node();
    node.setElement(obj);
    node.setNext(null);
    if (size == 0)
        head = node;
    else
        tail.setNext(node);
    tail = node;
    size ++;
}
public Object dequeue() throws QueueEmptyException {
    Object obj;
    if (size == 0)
        throw new QueueEmptyException("Queue is empty.");
    obj = head.getElement();
    head = head.getNext();
    size --;
    if (size == 0)
        tail = null;
    return obj; }

Queue ADT: Linked List - Based Implement. (cont.)
Queue ADT: Performance of LL - Based Implement.

**Run Time** — Good! all methods run in constant $O(1)$ time

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>isEmpty</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>front</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>enqueue</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Space Usage** — Good! $O(n)$, $n$ – current # of elements in the stack

**General Note** — no problems with size/overflow
## Queue ADT: Array vs. Linked List Implementation

<table>
<thead>
<tr>
<th>Properties of Queues</th>
<th>Array Implementation</th>
<th>Linked List Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered access</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>additions/removals at a cursor</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>frequent resizing</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>implementation complexity / cost</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>

In Linked List implementation:
1) each element must contains reference “next”
2) for every en/de-queued element, method setNext()/getNext() must be called
Queue ADT: Questions

Q.1 Assume we want to implement Queue using a SLL, which has only one reference (e.g.) to the head of SLL.

What is the main disadvantage of this implementation? What would be the time complexity of the main queue operations, in this case?

Q.2 (textbook C-4.2)
Describe how to implement the queue ADT using two stacks. What is the running time of the enqueue() and dequeue() methods?

Q.3 (textbook C-4.7)
Describe in pseudo-code a linear-time algorithm for reversing a queue Q.