COSC 2006: Data Structures I

The Queue ADT
Queue interface
Fixed array implementation
Dynamic array implementation
Linked implementation
Applications of queues
What is a queue?

- A queue simulates a waiting line.
- People enter at the back (rear) of the line
  - enqueue operation
- People leave at the front of the line
  - dequeue operation
- No other mutable access is possible

---

dequeuer

enqueue

FIFO
In Java queues are objects and we can define a Queue ADT interface as the specification of the queue operations. Any class that implements the interface "is a" Queue

Use the generic Object type for the elements stored in queue

```java
package stacksAndQueues;
public interface Queue {
    public void enqueue(Object e);
    public Object dequeue() throws EmptyQueueException;
    public Object front();
    public void clear(); // gives an empty queue
    public boolean isEmpty();
    public int size(); // number of elements in queue
    public String toString();
}
```
In Java 1.5 (Java 5) we can use a generic type instead of the Object type to define a generic queue. Then type casting is not necessary and auto boxing/unboxing can be used to easily work with stacks of primitive types.

```java
package stacksAndQueues;
public interface Queue<E> {
    public void enqueue(E e);
    public E dequeue() throws EmptyQueueException;
    public E front();
    public void clear(); // gives an empty queue
    public boolean isEmpty();
    public int size(); // number of elements in queue
    public String toString();
}
```
Different notations

The names for the different queue operations are not standardized. Some different notations are

- enqueue, insert       mutable
- dequeue, remove, delete  mutable
- front, first, getFront, getFirst  immutable

The name used in Main's book for the dequeue operation is confusing. He uses **getFront** but the name suggests an immutable operation.
Example of Queue operations

Following statements construct a queue of strings and perform some operations on it. The notation used is \([a,b,c,d]\) where \(a\) is at the front and \(d\) is at the rear.

```java
Queue<String> q = new LinkedListQueue<String>();
q.enqueue("Fred");       // q = ["Fred"]
q.enqueue("Jack");       // q = ["Fred", "Jack"]

String s = q.dequeue();  // q = ["Jack"]
                          // s is "Fred"

q.enqueue("Jill");       // q = ["Jack", "Jill"]
String a = q.front();    // a is "Jack"
```
Our first attempt at an implementation is to use an array such that index 0 is the front of the queue and the rear index increments each time an element is enqueued.

Here \( r \) is the index of the next available position at the rear of the queue. \( r = 0 \) corresponds to an empty queue.
Naive array implementation (2)

After enqueueing 4 elements the queue looks like

To save space we will not always show the objects referenced by e0, e1, e2, ...

After dequeueing an element the queue should look like

which can only be achieved by copying the elements e1, e2, e3 back one position in the array: dequeue is an $O(n)$ operation
To avoid the $O(n)$ dequeue operation we can introduce a front index and increment it each time we dequeue an element. Now an empty queue looks like

$$f = -1 \quad r = 0$$

Here $r$ is the index into the next available position at the rear of the queue and $f$ is the index of the element at the front of the queue.

$f = -1$ and $r = 0$ corresponds to an empty queue
Naive array implementation (4)

After enqueueing 3 elements the queue looks like

```
| f | 0 | r | 3 |
```
```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Now dequeue an element and enqueue an element

```
| f | 1 | r | 4 |
```
```
<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>d</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Now do two more pairs of dequeue - enqueue operations

```
| f | 3 | r | 6 |
```
```
<table>
<thead>
<tr>
<th>d</th>
<th>e</th>
<th>f</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Now r has gone beyond the last index and the queue is full even though there are three empty array elements

avoids copying but wastes space
Circular array implementation (1)

The solution is to consider the array as a circular array so that the element following the highest index (5 in the example) is the element at index 0.

Instead of

```
| f | 3 | r | 6 |
```
we would have

```
| f | 3 | r | 0 |
```

the index r has wrapped around as though the array is circular
Circular array implementation (2)

Enqueueing one more element now gives

\[ \begin{array}{ccc}
  f & 3 & r \\
  \hline
  0 & 1 & 2 \\
\end{array} \quad \begin{array}{ccc}
  g & d & e & f \\
  \hline
  3 & 4 & 5 \\
\end{array} \]

and enqueueing one more element gives

\[ \begin{array}{ccc}
  f & 3 & r \\
  \hline
  0 & 1 & 2 \\
\end{array} \quad \begin{array}{ccc}
  g & h & d & e & f \\
  \hline
  3 & 4 & 5 \\
\end{array} \]

enqueueing one more element gives

\[ \begin{array}{ccc}
  f & 3 & r \\
  \hline
  0 & 1 & 2 \\
\end{array} \quad \begin{array}{ccc}
  g & h & i & d & e & f \\
  \hline
  3 & 4 & 5 \\
\end{array} \]
The condition that the queue is full is that $f = r$

<table>
<thead>
<tr>
<th>f</th>
<th>3</th>
<th>r</th>
<th>3</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>g</th>
<th>h</th>
<th>i</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

However if we now dequeue all 6 elements letting $f$ wrap around as we did for $r$ then we end up with

<table>
<thead>
<tr>
<th>f</th>
<th>3</th>
<th>r</th>
<th>3</th>
</tr>
</thead>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Now we have $f = r = 3$ for the full queue and we have $f = r = 3$ for the empty queue so how can we distinguish the empty queue from the full queue.
There are two ways to distinguish a full queue from an empty queue.

The first way is to keep track of a counter which is decremented each time we do a dequeue and incremented each time we do an enqueue. Then if there are N elements in the array the array is full when the count reaches N.

The second way is to use only N-1 of the N elements in the array. Then the following situation represents a full queue.
Circular array implementation (5)

Let's do it the second way to avoid counting elements. Losing one array element is not really a problem.

When elements are enqueued the rear index is incremented by 1 mod N so we use the formula $r = (r + 1) \mod N$

When elements are dequeued the front index is incremented by 1 mod N so we use the formula $f = (f + 1) \mod N$
If we are not counting elements then how can we determine how many elements are in the queue in terms of the indices \( f \) and \( r \)?

Case 1: \( f \leq r \) (no wrapping has occurred yet) \( r – f \) elements

Case 2: \( f > r \) (wrapping has occurred) \( r \) elements + \( N-f \) elements

Size: Int either case number of elements in queue is \((N – f + r) \mod N\)
ArrayQueue implementation (1)

ArrayQueue implements the Queue<E> interface using a circular array

```java
package stacksAndQueues;
public class ArrayQueue<E> implements Queue<E> {
    private E[] data; // the array
    private int f; // index of front element
    private int r; // index of next available

    // see next slide
```
Constructor creates the array and initializes the front and rear indices. Here $f = r$ denotes an empty queue and initially $r = 0$ indicates the next available position.

```java
public ArrayQueue(int maxCapacity) {
    if (maxCapacity < 1)
        throw new IllegalArgumentException(
            "maxCapacity less than 1");
    data = (E[]) new Object[maxCapacity];
    f = 0;
    r = 0;
}
```
ArrayQueue implementation (3a)

Enqueue and dequeue operations

```java
public void enqueue(E e) throws FullQueueException {
    if (size() == data.length - 1)
        throw new FullQueueException("...");
    data[r] = e;
    r = (r + 1) % data.length;
}
```
ArrayQueue implementation (3b)

Enqueue and dequeue operations

```java
public E dequeue() throws EmptyQueueException
{
    if (isEmpty())
        throw new EmptyQueueException("...");
    E element = data[f];
    data[f] = null; // help garbage collector
    f = (f + 1) % data.length;
}
```
front and clear methods. Note that clear creates a new array so old array can be garbage collected.

```java
public E front() throws EmptyQueueException {
    if (isEmpty())
        throw new EmptyQueueException("...");
    return data[f];
}

public E clear() {
    data = (E[]) new Object[data.length];
    f = r = 0;
}
```
isEmpty, size, and isFull methods. They are all one-liners that use f, r, and data.length.

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

public int size()
{
    return (data.length - f + r) % data.length;
}

public boolean isFull()
{
    return size() == data.length - 1;
}
```
**ArrayQueue implementation (6)**

**toString method** returns a string of the form \([a,b,c,...,z]\) where \(a\) is the front element and \(z\) is the rear element.

```java
public String toString()
{
    StringBuilder s = new StringBuilder("[");
    int start = f; count = 1;
    while (count < size())
    {
        if (count == size())
        {
            s.append(data[start]);
        }
        else
        {
            s.append(data[start] + ",");
        }
        start = (start + 1) % data.length;
        count++;
    }
    s.append("]");
    return s.toString();
}
```

toString method returns a string of the form \([a,b,c,...,z]\) where \(a\) is the front element and \(z\) is the rear element.
Queue Exercise

- Modify ArrayQueue to use dynamic arrays.
- Call the new class DynamicArrayQueue.
- Note that the reallocate method needs to consider two cases ($f \leq r$ and $f > r$)

\[ f \leq r \]

\begin{align*}
\text{occupied} & \rightarrow \\
\end{align*}

\[ f > r \]

\begin{align*}
2 & 1 \rightarrow \\
1 & 2
\end{align*}
Linked queue (1)

- We can use a linked structure (list) to implement a LinkedQueue class.
- The head of the list is the front of the queue.
- The tail of the list is the rear of the queue.
- To enqueue an element means adding a new node at the tail of the list.
- To dequeue an element means removing a node at the head of the list.
- We use two references, head and tail.
Linked representation of the queue [e1, e2, e3, e4] with e1 the front element and e4 the rear element:
Here e1, e2, e3, e4 are references to the data objects shown in the picture.
Dequeue an element from the front:
Just link head to the second node.
The first node becomes an orphan

New list is [ e2, e3, e4 ]

One element list is a special case since tail will change

E element = head.data;
head = head.next;
size -- ;
return element;
Enqueueing an element e:
(1) make a new node
(2) Make the tail link reference it
(3) link tail to the new node

New list is
[ e1, e2, e3, e4, e ]

Step (1)
Step (2)
Step (3)

Empty list is special case

Node<E> newNode =
new Node<E>(e, null);
tail.next = newNode;
tail = newNode.

11/15/2006
The data fields are references to the head and tail nodes, and the size of the list. We use an inner class for the nodes.

package stacksAndQueues;
public class LinkedQueue<E> implements Queue<E> {
    private Node<E> head;
    private Node<E> tail;
    private int size;

    public LinkedQueue() {
        head = tail = null; // empty list
        size = 0;
    }

    // see next slides
LinkedQueue class (2)

Enqueue adds a new node at the tail of the list. The empty list is a special case.

```java
public void enqueue(E e) {
    Node<E> newNode = new Node<E>(e, null);
    if (size == 0) {
        head = newNode;
    } else {
        tail.next = newNode;
    }
    tail = newNode; // in either case
    size++;
}
```
Dequeue removes the node at the head of the list. The one element list is a special case since it becomes the empty list. Tail needs to be updated only if we have a one-element list.

```java
public E dequeue() throws EmptyQueueException {
    if (isEmpty())
        throw new EmptyQueueException("...");
    E element = head.data;
    head = head.next; // skip to second node
    size--;
    if (size == 0) // had 1 element list
    {
        tail = null;
    }
    return element;
}
```
front returns the element at the front of the queue without removing it.
clear returns the queue to its empty state

```java
public E front() throws EmptyQueueException
{
    if (isEmpty())
        throw new EmptyQueueException("...");
    return head.data;
}

public void clear()
{
    head = tail = null;
    size = 0;
}

public int size() { return size; }
```
toString returns a string representation of a queue in the format \([ e1, e2, ..., en \)] where e1 is element at the front of the queue and en is the element at the rear of the queue.

```java
toString() {  StringBuilder s = new StringBuilder("[");  Node<E> cursor = head;  while (cursor != null)  {    s.append(cursor.data.toString());    if (cursor.next != null) {      s.append(",");      cursor = cursor.next();    }  }  s.append("]");  return s.toString();}
```
The Node class is inside the LinkedQueue class

```java
private class Node<T>
{
    private T data;
    private Node<T> next;
    public Node()
    {
        this(null,null);
    }
    public Node(T data, Node<T> next)
    {
        this.data = data;
        this.next = next;
    }
}
} // end of LinkedQueue class
```
Circular linked queue

References to both the head and the tail

For circular queue tail references the head element so we only need a tail reference
Queue exercise (1)

- Rewrite LinkedQueue so that it uses a circular linked list having only a tail reference.
- Call the new class LinkedCircularQueue
- Draw picture for empty list
- Draw picture for 1-element list
- Draw picture for 2-element list

See next slides
Queue exercise (2)

Empty list

Two element list

One element list
Now we only have a reference to the tail. 
Instead of having a null next field at the end of the list we have a reference back to the head node

```java
package stacksAndQueues;
public class CircularLinkedQueue<E> implements Queue<E> {
    private Node<E> tail;
    private int size;

    public CircularLinkedQueue() {
        tail = null; // empty list
        size = 0;
    }
    // see next slides
```
CircularLinkedQueue class (2)

enqueue method

```java
public void enqueue(E e) {
    Node<E> newNode = new Node<E>(e, null);
    if (isEmpty()) {
        // 1-element list references itself
        newNode.next = newNode;
    } else {
        newNode.next = tail.next; // head
        tail.next = newNode;
    }
    tail = newNode; // tail refs last node
    size++;
}
```
CircularLinkedListQueue class (3)

deprove method: 3 cases, dequeue in a one-element queue, dequeue in a more than one-element queue, dequeue in an empty queue.

```java
public E dequeue()
    throws EmptyStackException
{
    if (isEmpty())
        throw new EmptyQueueException("...");
    E element = tail.next.data;
    if (tail.next == null) // 1-elem --> empty
        tail = null;
    else // more than one element
        tail.next = tail.next.next;
    size--;
    return element;
}
```
public E front()
    throws EmptyStackException
    {
        if (isEmpty())
            throw new EmptyQueueException("...");
        return tail.next.data;
    }
CircularLinkedListQueue class (5)

clear, isEmpty, size

```java
public void clear()
{
    tail = null;
    size = 0;
}

public boolean isEmpty()
{
    return (size == 0)
}

public int size() { return size; }
```
public String toString()
{
    StringBuilder s = new StringBuilder("[");
    if (tail != null) // list not empty
    {
        Node<E> head = tail.next;
        Node<E> cursor = head;
        while (cursor.next != head)
        {
            s.append(cursor.data.toString());
            if (cursor.next != head)
                s.append(",");
            cursor = cursor.next;
        }
        s.append(cursor.data.toString());
    }
    s.append("]"); return s.toString();
}
CircularLinkedQueue class (7)

The inner Node<E> class

```java
private class Node<T>
{
    private T data;
    private Node<T> next;

    public Node()
    {
        this(null, null);
    }

    public Node(T data, Node<T> next)
    {
        this.data = data;
        this.next = next;
    }
}
} // end CircularLinkedQueue
```
A Deque is a double ended queue. It has enqueue and dequeue operations at both ends of the queue.

It has the following operations at the front
- insertFront, removeFront, front

It has the following operations at the rear
- insertRear, removeRear, rear

Also we can add the methods
- clear, isEmpty, toString, size
Deque (2)

- removeFront
- insertFront
- insertRear
- removeRear
Java 5 Deque interface

A Deque has insert and remove operations at both ends

```java
class Deque{
    public void insertFront(E e);
    public void insertRear(E e);
    public E removeFront() throws EmptyDequeException
    public E removeRear() throws EmptyDequeException
    public E front() throws EmptyDequeException
    public E rear() throws EmptyDequeException
    public void clear(); // gives an empty queue
    public boolean isEmpty();
    public int size(); // number of elements in deque
    public String toString();
}
```
Linked Deque Implementation

- A singly linked list can be used to implement a deque.
- However, not all the insert and remove operations can be done in O(1).
- For example, `removeRear` is O(n) since we need to do a link traversal to locate the node previous to the tail node (previous node becomes new tail node).
- With a doubly linked list all operations are O(1).
Doubly linked nodes (1)

- So far our nodes have had a data part and a link part but the link allows us to traverse the list only in one direction (forward).
- To traverse a list in both directions we can use doubly linked nodes having a data part a next link and a previous link.
- We will use a doubly linked structure to implement the Deque interface.
Here we can traverse the list from left to right starting at head and we can traverse the list from right to left starting at tail.

We will use a doubly linked structure to implement a Deque so we need to know how to insert and delete list elements at the head and the tail.

A general list would also need to have operations to insert in the middle.
The doubly linked node class DLNode will be an inner class of the LinkedDeque class. There are two references in each node, a next reference and a previous reference.

```java
public class DLNode<T> {
    private T data;
    private DLNode<T> prev;
    private DLNode<T> next;
    public DLNode() { this(null, null, null); }

    public DLNode(T data, DLNode<T> prev,
                  DLNode<T> next) {
        this.data = data;
        this.prev = prev; this.next = next;
    }
}
```
Sentinel nodes (1)

- It is more complicated to manage links and special cases for a doubly linked list than a singly linked list.
- Therefore we will use the concept of a sentinel node to simplify the logic.
- Sentinel nodes hold no data.
- They simply mark the beginning and end of a doubly linked list so that there are no special cases. All insertions and deletions follow the same logic.
Header and Trailer nodes

- **Header**: a reference to a DLNode that has a next link but null previous link. The data part is not used.

- **Trailer**: a reference to a DLNode that has a previous link but null next link. The data part is not used.

List is between these nodes so there are no cases: all operations take place between two nodes.
List with header and trailer nodes

header

List goes here

trailer
The empty list is just a header sentinel node and a trailer sentinel node referencing each other.

The header sentinel node has a next link referencing the trailer sentinel node and a null previous link.

The trailer sentinel node has a previous link referencing the header sentinel node and a null next link.
One element list

Here is a one-element list \([ e1 ]\) with sentinel nodes

![Diagram of a one-element list with sentinel nodes]
Here is a two-element list \([e_1, e_2]\) with sentinel nodes
InsertFront(e) (1)
InsertFront(e) (2)

`DLNode<E> second = header.next`
InsertFront(e) (3)

```
DLNode<E> newNode = new Node<E>(e, header, second);
```
InsertFront(e) (4)

second.prev = newNode
InsertFront(e)  (5)

header.next = newNode

diagram showing node connections
InsertFront(e) (6)
InsertFront(e) (7)
Exercise

- Draw pictures for the other operations
- `removeFront()`
- `insertRear(e)`
- `removeRear()`
- Write the code
We can use a doubly linked list with sentinels header and trailer to implement the Deque interface.

```java
package stacksAndQueues;
public class LinkedDeque<E> implements Deque<E> {
    private DLNode<E> header;
    private DLNode<E> trailer;
    private int size;

    // continued next slide
```
The constructor needs to construct an empty list and set size to 0. Save code by calling the clear() method in the constructor.

```java
public LinkedDeque()
{
    clear();
}

public void clear()
{
    header = new DLNode<E>(null,null,null,null);
    trailer = new DLNode<E>(null,null,null,null);
    header.next = trailer;
    trailer.prev = header;
    size = 0;
}
```
**LinkedDeque class (3)**

*insertFront neads to create a new Node at the head of the list and link it to the header and the second node*

```java
public void insertFront(E e) {
    // create new node and have header reference it

    // skip over sentinal: first node will now be the second one
    DLNode<E> second = header.next;
    DLNode<E> newNode =
        new DLNode<E>(e, header, second);
    second.prev = newNode;
    header.next = newNode;
    size++;
}
```
Using symmetry insertFront can easily be modified to get the implementation of insertRear (use penultimate instead of second and use trailer instead of header).

```java
public void insertRear(E e) {
    // create new node and have trailer reference it
    DLNode<E> penultimate = trailer.prev;
    DLNode<E> newNode =
        new DLNode<E>(e, penultimate, trailer);
    penultimate.next = newNode;
    trailer.prev = newNode;
    size++;
}
```
removeFront

```java
public E removeFront()
    throws EmptyDequeueException
{
    if (isEmpty())
    {
        throw new EmptyDequeueException("...");

        DLNode<E> front = head.next;
        E data = front.data;
        DLNode<E> second = front.next;
        header.next = second;
        second.prev = header;
        size--;
        return data;
    }
}
```
**LinkedDeque class (5b)**

*Use symmetry to write removeRear*

```java
public E removeRear()
    throws EmptyDequeException
{
    if (isEmpty())
        throw new EmptyDequeException("...");
    DLNode<E> rear = trailer.prev;
    E data = rear.data;
    DLNode<E> penultimate = rear.prev;
    trailer.prev = penultimate;
    penultimate.next = trailer;
    size--;
    return data;
}
```
LinkedDeque class (6a)

**front method.**

```java
public E front()
    throws EmptyDequeException
{
    return header.next.data;
}
```
LinkedDeque class (6b)

rear method

```java
public E rear()
    throws EmptyDequeException
{
    return trailer.prev.data;
}
```
Write the isEmpty and size methods

```java
public boolean isEmpty()
{
    return size == 0;
}

public int size()
{
    return size;
}
```
Write the isEmpty and size methods

```java
public String toString()
{
    StringBuilder s = new StringBuilder("[");
    DLNode<E> cursor = header.next;
    while (cursor != trailer)
    {
        s.append(cursor.data.toString());
        if (cursor.next != trailer)
            s.append(",");
        cursor = cursor.next;
    }
    s.append("]");
    return s.toString();
}
```
Now include the DLNode class as an inner class

```java
public class DLNode<T>
{
    // done earlier
}
} // end LinkedQueue class
```
Stack as a Deque

The Stack<E> interface can be implemented using the LinkedDeque class as an adapter class to obtain the class DequeStack:

- **Stack operation**    **Deque operation**
  - push          insertFront
  - pop           removeFront
  - peek          front

Here the top end of the stack corresponds to the front end of the deque (see code in Eclipse project)
Queue as a Deque

- The Queue<E> interface can be implemented using the LinkedDeque class as an adapter class to obtain the class DequeQueue:

  - **Queue operation** | **Deque operation**
  - enqueue | insertRear
  - dequeue | removeFront
  - front | front

- Here the queue front corresponds to the deque front and the queue rear corresponds to the deque rear (see Eclipse project)