The Queue ADT
Objectives

• Define the queue ADT
• Show how a queue can be used to solve problems
• Examine various queue implementations
• Compare queue implementations
Queues

- **Queue**: a linear collection whose elements are added at one end (the *rear* or *tail* of the queue) and removed from the other end (the *front* or *head* of the queue)

- A queue is a **FIFO** (first in, first out) data structure

- Any waiting line is a queue:
  - The check-out line at a grocery store
  - The cars at a stop light
  - An assembly line
Conceptual View of a Queue

Adding an element

Front of queue

New element is added to the rear of the queue
Conceptual View of a Queue

Removing an element

New front element of queue

Element is removed from the front of the queue
# Operations on a Queue

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue</td>
<td>Removes an element from the front of the queue</td>
</tr>
<tr>
<td>enqueue</td>
<td>Adds an element to the rear of the queue</td>
</tr>
<tr>
<td>first</td>
<td>Examines the element at the front of the queue without removing it</td>
</tr>
<tr>
<td>isEmpty</td>
<td>Determines whether the queue is empty</td>
</tr>
<tr>
<td>size</td>
<td>Determines the number of elements in the queue</td>
</tr>
<tr>
<td>toString</td>
<td>Returns a string representation of the queue</td>
</tr>
</tbody>
</table>
public interface QueueADT<T> {
    // Adds one element to the rear of the queue
    public void enqueue(T element);
    // Removes and returns the element at the front of the queue
    public T dequeue() throws EmptyCollectionE;
    // Returns without removing the element at the front of the queue
    public T first() throws EmptyCollectionE;
    // Returns true if the queue contains no elements
    public boolean isEmpty();
    // Returns the number of elements in the queue
    public int size();
    // Returns a string representation of the queue
    public String toString();
}
Uses of Queues in Computing

- Printer queue
- Keyboard input buffer
- GUI event queue (click on buttons, menu items)
A Caesar cipher is a substitution code that encodes a message by shifting each letter in a message by a constant amount $k$

- If $k$ is 5, $a$ becomes $f$, $b$ becomes $g$, etc.
- **Example**: $n$ $qtaj$ ofaf

- Used by Julius Caesar to encode military messages for his generals (around 50 BC)
- This code is fairly easy to break.
Using Queues: Coded Messages

• **Modern version**: ROT13
  
  • Each letter is shifted by 13
  
  • “used in online forums as a means of hiding spoilers, punchlines, puzzle solutions, and offensive materials from the casual glance” ([Wikipedia](https://en.wikipedia.org))
Using Queues: Coded Messages

• **An improvement**: change how much a letter is shifted depending on where the letter is in the message

• A **repeating key** is a sequence of integers that determine how much each character is shifted
  - Example: consider the repeating key
    
    3 1 7 4 2 5
  
  • The first character in the message is shifted by 3, the next by 1, the next by 7, and so on
  • When the key is exhausted, start over at the beginning of the key
Using Queues: Coded Messages

A **repeating key** is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

```
3 1 7 4 2 5
```

message: *knowledge*

encoded message:

```
queue:
```

```
3 1 7 4 2 5
```

```
Using Queues: Coded Messages

A **repeating key** is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

\[ 3 \ 1 \ 7 \ 4 \ 2 \ 5 \]

message: knowledge
encoded
message: n

\[ \text{dequeued: 3} \]

\[ \text{queue:} \]
\[ 1 \ 7 \ 4 \ 2 \ 5 \]
Using Queues: Coded Messages

A *repeating key* is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key 3 1 7 4 2 5

message: knowledge
encoded message: n

queue: 1 7 4 2 5 3
Using Queues: Coded Messages

A *repeating key* is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

\[3 \ 1 \ 7 \ 4 \ 2 \ 5\]

message: *knowledge*

encoded

message: *no*

dequeued: 1

queue: 7 4 2 5 3
Using Queues: Coded Messages

A *repeating key* is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key 3 1 7 4 2 5

message: knowledge
encoded
message: no

queue: 7 4 2 5 3 1
Using Queues: Coded Messages

A *repeating key* is a sequence of integers that determine by how much each character in a message is shifted. Consider the repeating key

3  1  7  4  2  5

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

message: *knowledge*

encoded

message: *novangjhl*

queue: 4  2  5  3  1  7
Using Queues: Coded Messages

• We can use a queue to store the values of the key
  • dequeue a key value when needed
  • After using it, enqueue it back onto the end of the queue

• So, the queue represents the constantly cycling values in the key
Using Queues: Coded Messages

• See *Codes.java* in the sample code page of the course’s website
  • Note that there are *two* copies of the key, stored in two separate queues
    • The encoder has one copy
    • The decoder has a separate copy

• Why?
Using Queues: Ticket Counter Simulation

• Simulate the waiting line at a movie theatre:
  • Determine how many cashiers are needed to keep the customer wait time under 7 minutes

• **Assume:**
  • Customers arrive on average every 15 seconds
  • Processing a request takes two minutes once a customer reaches a cashier

• See *Customer.java, TicketCounter.java* in the sample code page of the course’s website
# Results of Ticket Counter Simulation

<table>
<thead>
<tr>
<th>Number of Cashiers</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time (in seconds)</td>
<td>5317</td>
<td>2325</td>
<td>1332</td>
<td>840</td>
<td>547</td>
<td>355</td>
<td>219</td>
<td>120</td>
<td>120</td>
<td>120</td>
</tr>
</tbody>
</table>
Queue Implementation Issues

• What do we need to implement a queue?
  • A data structure (container) to hold the data elements
  • A variable to indicate the front of the queue
  • A variable to indicate the rear of the queue
Queue Implementation Using a Linked List

• A queue can be represented as a linked list of nodes, with each node containing a data item
• We need two pointers for the linked list
  • A pointer to the beginning of the linked list (front of queue)
  • A pointer to the end of the linked list (rear of queue)
• We will also have a count of the number of items in the queue
Linked Implementation of a Queue

A queue $q$ containing four elements
Discussion

• What are the values of front and rear if the queue is empty?

• What are their values if there is only 1 element?
Queue After Adding Element

New element is added in a node at the end of the list, \textit{rear} points to the new node, and \textit{count} is incremented.
Queue After a **dequeue** Operation

Node containing ▀ is removed from the front of the list (see previous slide), **front** now points to the node that was formerly second, and **count** has been decremented.
Java Implementation

• The queue is represented as a linked list of nodes:
  • We will again use the LinearNode class
  • `front` is a reference to the head of the queue (beginning of the linked list)
  • `rear` is a reference to the tail of the queue (end of the linked list)
  • The integer `count` is the number of nodes in the queue
public class LinkedQueue<T> implements QueueADT<T> {
/**
 * Attributes
 */
private int count;
private LinearNode<T> front, rear;

/**
 * Creates an empty queue.
 */
public LinkedQueue() {
    count = 0;
    front = rear = null;
}
public void enqueue (T element) {
    LinearNode<T> node = new LinearNode<T> (element);

    if (isEmpty( ))
        front = node;
    else
        rear.setNext (node);

    rear = node;
    count++;
}
// Removes the element at the front of the queue and returns a reference to it. Throws an EmptyCollectionException if the queue is empty.

public T dequeue () throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException("queue");
    T result = front.getElement();
    front = front.getNext();
    count--;
    if (isEmpty())
        rear = null;
    return result;
}
Array Implementation of a Queue

• **First Approach:**
  • Use an array in which index 0 represents one end of the queue (the *front*)
  • Integer value *count* represents the number of elements in the array (so the element at the rear of the queue is in position count - 1)

• **Discussion:** What is the challenge with this approach?
An Array Implementation of a Queue

A queue aq containing four elements

front

queue 0 1 2 3 4 5 ...

aq

count 4
Queue After Adding an Element

The element is added at the array location given by the value of `count` and then count is increased by 1.
Queue After Removing an Element

Element is removed from array location 0, remaining elements are shifted forward one position in the array, and then count is decremented.

```
[0, 1, 2, 3, 4, ...]
```

```
queue
```

```
count
```
public class ArrayQueue<T> implements QueueADT<T> {
    private final int DEFAULT_CAPACITY = 100;
    private int count;
    private T[] queue;

    public ArrayQueue() {
        count = 0;
        queue = (T[])(new Object[DEFAULT_CAPACITY]);
    }

    public ArrayQueue (int initialCapacity) {
        count = 0;
        queue = (T[])(new Object[initialCapacity]);
    }
}
// Adds the specified element to the rear of the queue,
// expanding the capacity of the queue array if
// necessary.

public void enqueue(T element) {
    if (size() == queue.length) {
        expandCapacity();
        queue[count] = element;
    }
    count++;
}
// Removes the element at the front of the queue and returns
// a reference to it. Throws anEmptyCollectionException if the
// queue is empty.

public T dequeue() throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException(“Empty queue”);
    T result = queue[0];
    count--;
    // shift the elements
    for (int i = 0; i < count; i++)
        queue[i] = queue[i+1];
    queue[count] = null;
    return result;
}
Second Approach: Queue as a Circular Array

• If we do not fix one end of the queue at index 0, we will not have to shift elements

• **Circular array** is an array that conceptually loops around on itself
  - The last index is thought to “precede” index 0
  - In an array whose last index is \( n \), the location “before” index 0 is index \( n \); the location “after” index \( n \) is index 0

• We need to keep track of where the **front** as well as the **rear** of the queue are at any given time
Circular Array Implementation of a Queue

- **front**: 3
- **rear**: 8
- **count**: 5

The diagram illustrates a circular array with indices 0 to 10, representing a queue. The front and rear indices are marked with blue and red boxes, respectively. The count is indicated by a yellow box. The queue is depicted by the arrows and numbers, showing the sequence and structure of the circular array implementation.
A Queue Straddling the End of a Circular Array

cq

front
rear
queue
count
Circular Queue Drawn Linearly

Queue from previous slide

```
cq

98
front

2
rear

4
count

0 1 2 3 4 ...
96 97 98 99

Queue from previous slide
```
Circular Array Implementation

• When an element is enqueued, the value of \( \text{rear} \) is incremented
• But it must take into account the need to loop back to index 0:

\[
\text{rear} = (\text{rear}+1) \mod \text{queue.length};
\]

• Can this array implementation also reach capacity?
Example: array of length 4
What happens?

Suppose we try to add one more item to a queue implemented by an array of length 4

The queue is now full. How can you tell?
Add another item!
Need to expand capacity…

We can’t just double the size of the array and copy values to the same positions as before: circular properties of the queue will be lost.

These locations should be in use.
We *could* build the new array, and copy the queue elements into contiguous locations beginning at location *front*:
Or, we could copy the queue elements in order to the *beginning* of the new array.
New element is added at \( \text{rear} = (\text{rear}+1) \mod \text{queue.length} \)

See \textit{expandCapacity()} in \texttt{CircularArrayQueue.java}
Pseudocode for the Enqueue Operation Using a Circular Array Implementation of a Queue

**Algorithm enqueue(element)**

- if queue is full then expandQueue()
- rear = (rear + 1) mod size of queue
- queue[rear] = element
- ++count

**Algorithm expandQueue()**

- q = new array of size 2 * size of queue
- copied = 0 // number of elements copied to the larger array
- i = 0 // index of next entry in array q
- j = front // index of next entry in array queue
- while copied < count do { // copy data to new array
  - q[i] = queue[j]
  - ++i
  - j = (j + 1) mod size of queue
  - ++copied
}
- rear = count – 1 // position of last element in the queue
- front = 0
- queue = q
Enqueue Operation in Java

```java
public void enqueue (T element) {
    if (count == queue.length) expandQueue();
    rear = (rear + 1) % queue.length;
    queue[rear] = element;
    ++count;
}
```
Enqueue Operation in Java

```java
public void expandQueue() {
    T[] q = (T[]) new Object[2*queue.length];
    copied = 0; // number of data items copied to larger array
    i = 0;    // index of next entry in array q
    j = front; // index of next entry in array queue
    while (copied < count) {
        q[i] = queue[j];
        ++i;
        j = (j + 1) % queue.length;
        ++copied;
    }
    rear = count - 1;
    front = 0;
    queue = q
}
```
Algorithm dequeue() {
    if queue is empty then ERROR
    result = queue[front]
    count = count – 1
    front = (front + 1) mod (size of array queue)
    return result
}

Where mod is the modulo operator (or modulus or remainder), denoted % in Java.
public T dequeue() {
    if (isEmpty())
        throw new EmptyQueueException();
    result = queue[front];
    count = count - 1;
    front = (front + 1) % queue.length;
    return result;
}