CS 304 Lecture 5

The `Queue` ADT

Xiwei Wang, Ph.D.

Assistant Professor

Department of Computer Science

Northeastern Illinois University

Chicago, Illinois, 60625

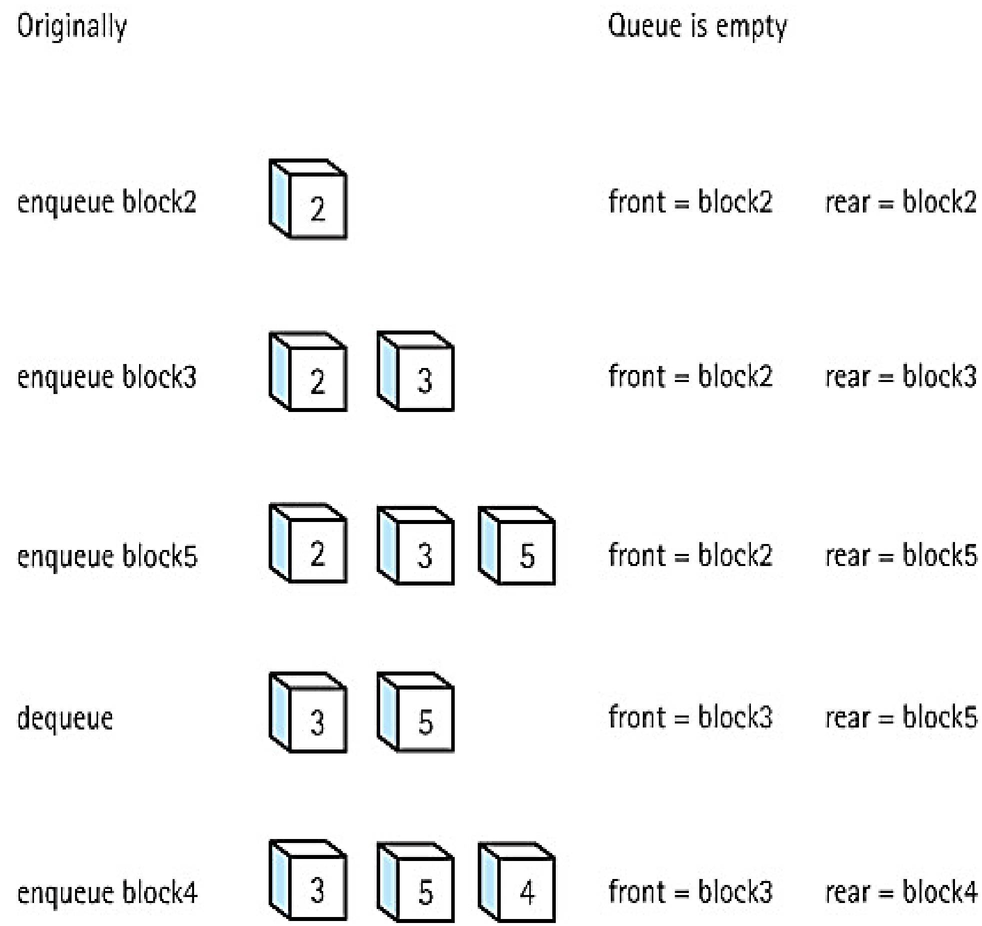
27 September 2016

Queues

- **Queue** - A structure in which elements are added to the rear and removed from the front; a "first in, first out" (FIFO) structure.

Queue operations

- Constructor - creates an empty queue
- Transformers
  - **`enqueue`** - adds an element to the rear of a queue
  - **`dequeue`** - removes and returns the front element of the queue



Using queues

- Examples of queues in real life situations
  - Cars waiting at a stop sign.
  - Customers waiting to check out in the checkout line.
  - Patients waiting outside the doctor's clinic.
- Queues in computer systems:
  - Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
  - Computer systems must often provide a "holding area" for messages between two processes, two programs, or even two systems. This holding area is usually called a "buffer" and is often implemented as a queue.

Formal specification of the **`Queue`** ADT

- Methods that are required by the **`Queue`** ADT
  - a constructor, **`enqueue`**, **`dequeue`**.
- Our **`Queue`** ADT is generic – the element type can be specified.
- We also need to
  - Identify and address any exceptional situations;
  - Determine boundedness;
  - Define the **`Queue`** interface or interfaces.
- Exceptional situations
  - **`dequeue`** – what if the queue is empty?
    - throw a **`QueueUnderflowException`**
    - plus define an **`isEmpty`** method for use by the application.
  - **`enqueue`** – what if the queue is full?
    - throw a **`QueueOverflowException`**
    - plus define an **`isFull`** method for use by the application.

Boundedness

- We support two versions of the **`Queue`** ADT
  - a bounded version
  - an unbounded version
- We define three interfaces
  - **`QueueInterface`**: features of a queue not affected by boundedness
  - **`BoundedQueueInterface`**: features specific to a bounded queue
  - **`UnboundedQueueInterface`**: features specific to an unbounded queue

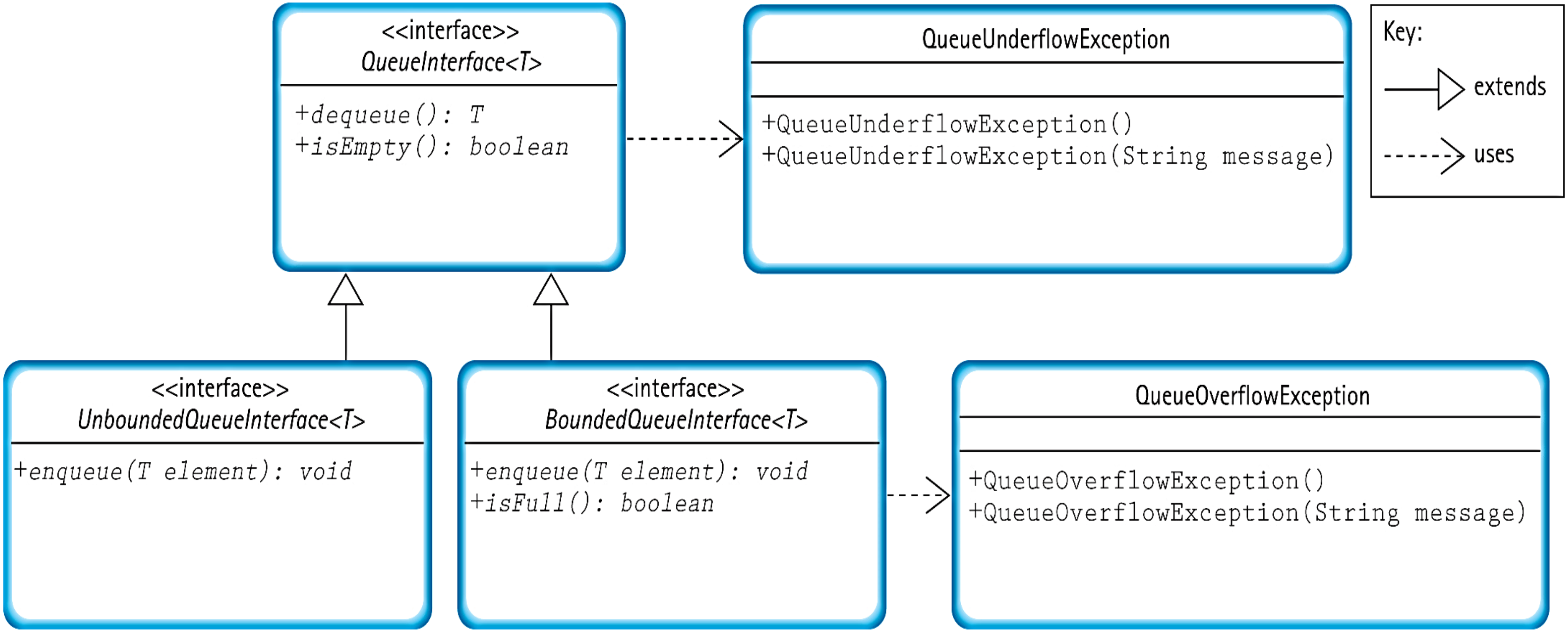
The interfaces of the **`Queue`** ADT

```
public interface QueueInterface<T>
{
  T dequeue() throws QueueUnderflowException;
  // Throws QueueUnderflowException if this queue is empty,
  // otherwise removes front element from this queue and
  // returns it.

  boolean isEmpty();
  // Returns true if this queue is empty, otherwise
  // returns false.
}
```

The interfaces of the **Queue** ADT

```
public interface BoundedQueueInterface<T> extends
QueueInterface<T>
{
  void enqueue(T element) throws QueueOverflowException;
  // Throws QueueOverflowException if this queue is full,
  // otherwise adds element to the rear of this queue.

  boolean isFull();
  // Returns true if this queue is full, otherwise returns
  // false.
}

public interface UnboundedQueueInterface<T> extends
QueueInterface<T>
{
  void enqueue(T element);
  // Adds element to the rear of this queue.
}
```

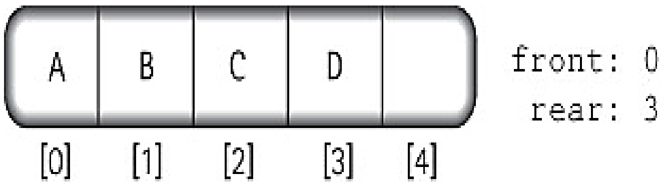
The interfaces of the `Queue` ADT



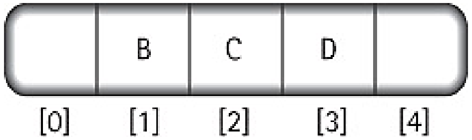
Array-based implementations - bounded

Fixed front design

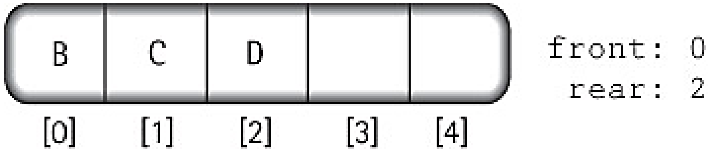
- After four calls to **`enqueue`** with arguments '**`A`**', '**`B`**', '**`C`**', and '**`D`**':



- **`dequeue`** the front element:



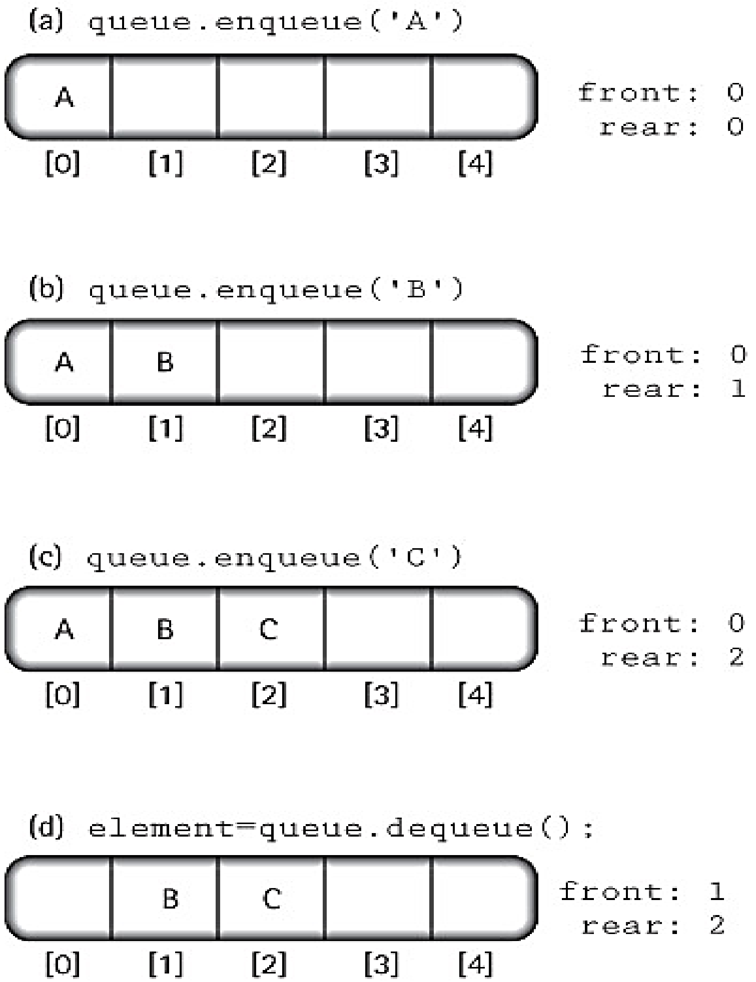
- Move every element in the queue up one slot:

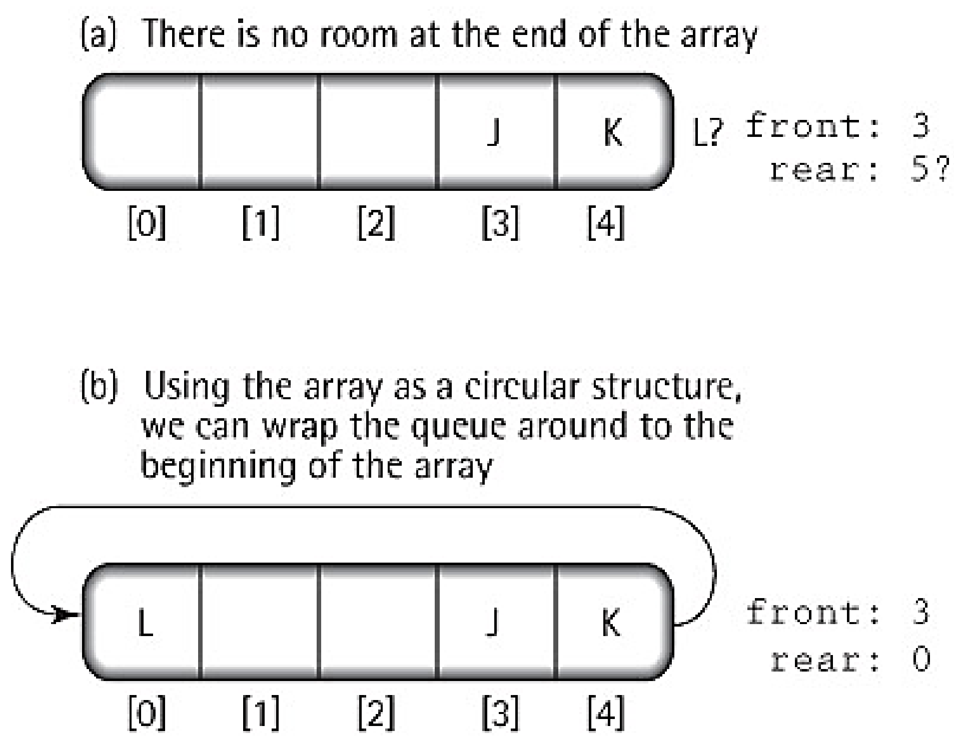


- The **`dequeue`** operation is inefficient, so we do not use this approach.

Array-based implementations - bounded

Floating front design





Array-based implementations - bounded

```
public class ArrayBndQueue<T> implements BoundedQueueInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] queue;              // array that holds queue elements
  protected int numElements = 0;    // number of elements in the queue
  protected int front = 0;          // index of front of queue
  protected int rear;               // index of rear of queue

  public ArrayBndQueue()
  {
    queue = (T[]) new Object[DEFCAP];
    rear =  DEFCAP - 1;
  }

  public ArrayBndQueue(int maxSize)
  {
    queue = (T[]) new Object[maxSize];
    rear =  maxSize - 1;
  }
```

The **enqueue** operation

```
public void enqueue(T element)
// Throws QueueOverflowException if this queue is full,
// otherwise adds element to the rear of this queue.
{
  if (isFull())
    throw new QueueOverflowException("Enqueue attempted on a
                                      full queue.");
  else
  {
    rear = (rear + 1) % queue.length;
    queue[rear] = element;
    numElements = numElements + 1;
  }
}
```

The **dequeue** operation

```
public T dequeue()
// Throws QueueUnderflowException if this queue is empty,
// otherwise removes front element from this queue and
// returns it.
{
  if (isEmpty())
    throw new QueueUnderflowException("Dequeue attempted on
                                           empty queue.");
  else
  {
    T toReturn = queue[front];
    queue[front] = null;
    front = (front + 1) % queue.length;
    numElements = numElements - 1;
    return toReturn;
  }
}
```

The `isEmpty` and `isFull` operations

```
public boolean isEmpty()
// Returns true if this queue is empty, otherwise returns
// false
{
  return (numElements == 0);
}

public boolean isFull()
// Returns true if this queue is full, otherwise returns
// false.
{
  return (numElements == queue.length);
}
```

Array-based implementations - unbounded

- The trick is to create a new, larger array, when needed, and copy the queue into the new array.
  - Since enlarging the array is conceptually a separate operation from enqueuing, we implement it as a separate **`enlarge`** method.
  - This method instantiates an array with a size equal to the current capacity plus the original capacity.
- We can drop the **`isFull`** method from the class, since it is not required by the unbounded **`Queue`** interface.
- The **`dequeue`** and **`isEmpty`** methods are unchanged.

Array-based implementations - unbounded

```
public class ArrayUnbndQueue<T> implements UnboundedQueueInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] queue;              // array that holds queue elements
  protected int origCap;            // original capacity
  protected int numElements = 0;    // number of elements n the queue
  protected int front = 0;          // index of front of queue
  protected int rear = -1;          // index of rear of queue

  public ArrayUnbndQueue()
  {
    queue = (T[]) new Object[DEFCAP];
    rear = DEFCAP - 1;
    origCap = DEFCAP;
  }

  public ArrayUnbndQueue(int origCap)
  {
    queue = (T[]) new Object[origCap];
    rear = origCap - 1;
    this.origCap = origCap;
  }
```

The `enlarge` operation

```
private void enlarge()
 // Increments the capacity of the queue by an amount
 // equal to the original capacity.
 {
   // create the larger array
   T[] larger = (T[]) new Object[queue.length + origCap];

   // copy the contents from the smaller array into the larger array
   int currSmaller = front;
   for (int currLarger = 0; currLarger < numElements; currLarger++)
   {
     larger[currLarger] = queue[currSmaller];
     currSmaller = (currSmaller + 1) % queue.length;
   }

   // update instance variables
   queue = larger;
   front = 0;
   rear = numElements - 1;
 }
```

The `enqueue` operation

```
public void enqueue(T element)
 // Adds element to the rear of this queue.
 {
   if (numElements == queue.length)
     enlarge();

   rear = (rear + 1) % queue.length;
   queue[rear] = element;
   numElements = numElements + 1;
 }
```

Applications: palindromes

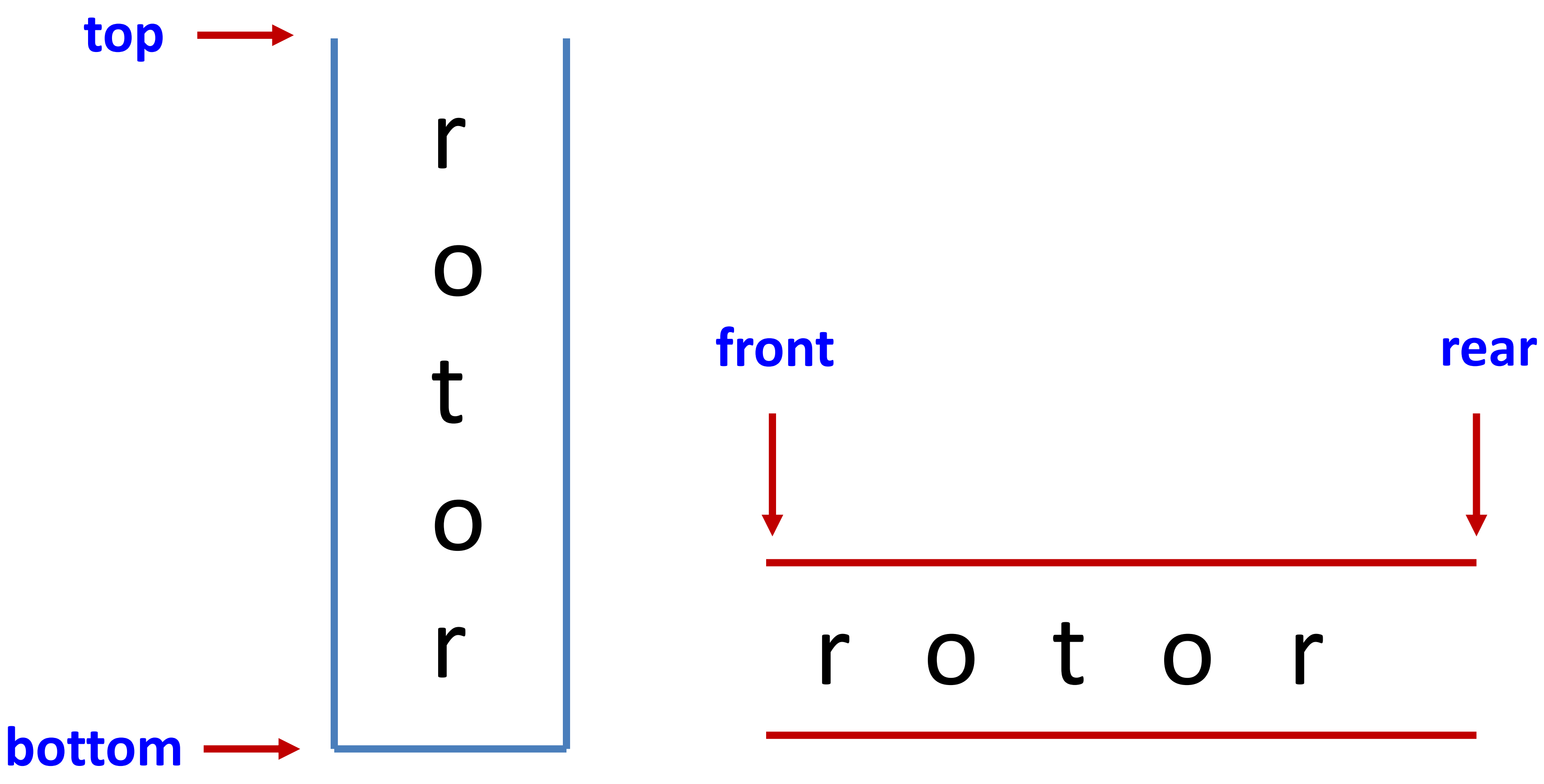
- Palindrome revisit: a string that is equal to itself when you reverse all characters.
- Our goal is to write a program that identifies palindromic strings.
  - We ignore blanks, punctuation and the case of letters.
- To help us identify palindromic strings we create a class called **`Palindrome`**, with a single exported static method **`test`**
  - **`test`** takes a candidate **`String`** argument and returns a **`boolean`** value indicating whether the string is a palindrome.
  - Since **`test`** is static we invoke it using the name of the class rather than instantiating an object of the class.
  - The **`test`** method uses both the **`stack`** and **`queue`** data structures.

Applications: palindromes

- The `test` method creates a `stack` and a `queue`.
- It then repeatedly pushes each input letter onto the stack, and also `enqueue`s the letter onto the queue.
- It discards any non-letter characters.
- To simplify comparison later, we `push` and `enqueue` only lowercase versions of the characters.
- After the characters of the candidate string have been processed, `test` repeatedly `pop`s a letter from the stack and `dequeue`s a letter from the queue.
- As long as these letters match each other the entire way through this process, we have a palindrome.

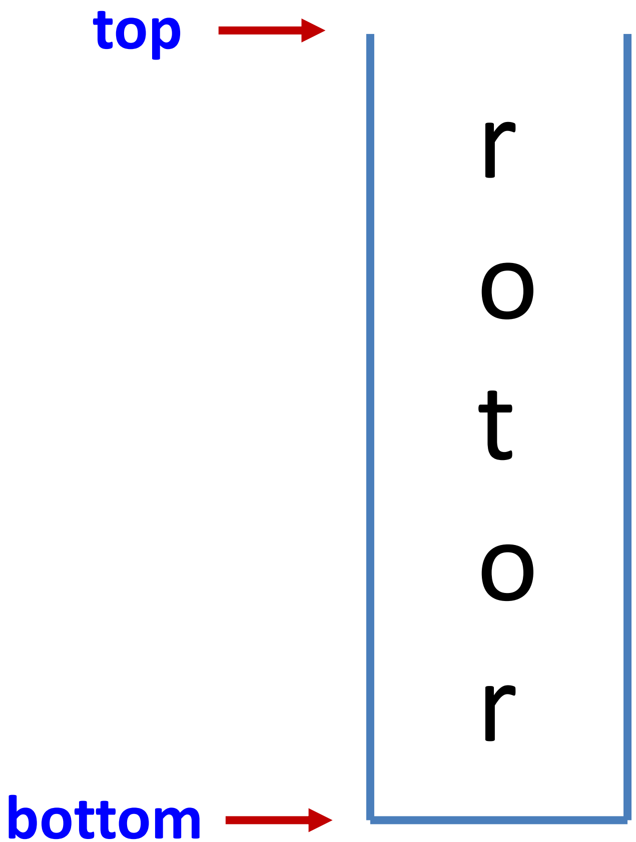
Applications: palindromes

`push` "rotor" onto the stack and **`enqueue`** it onto the queue



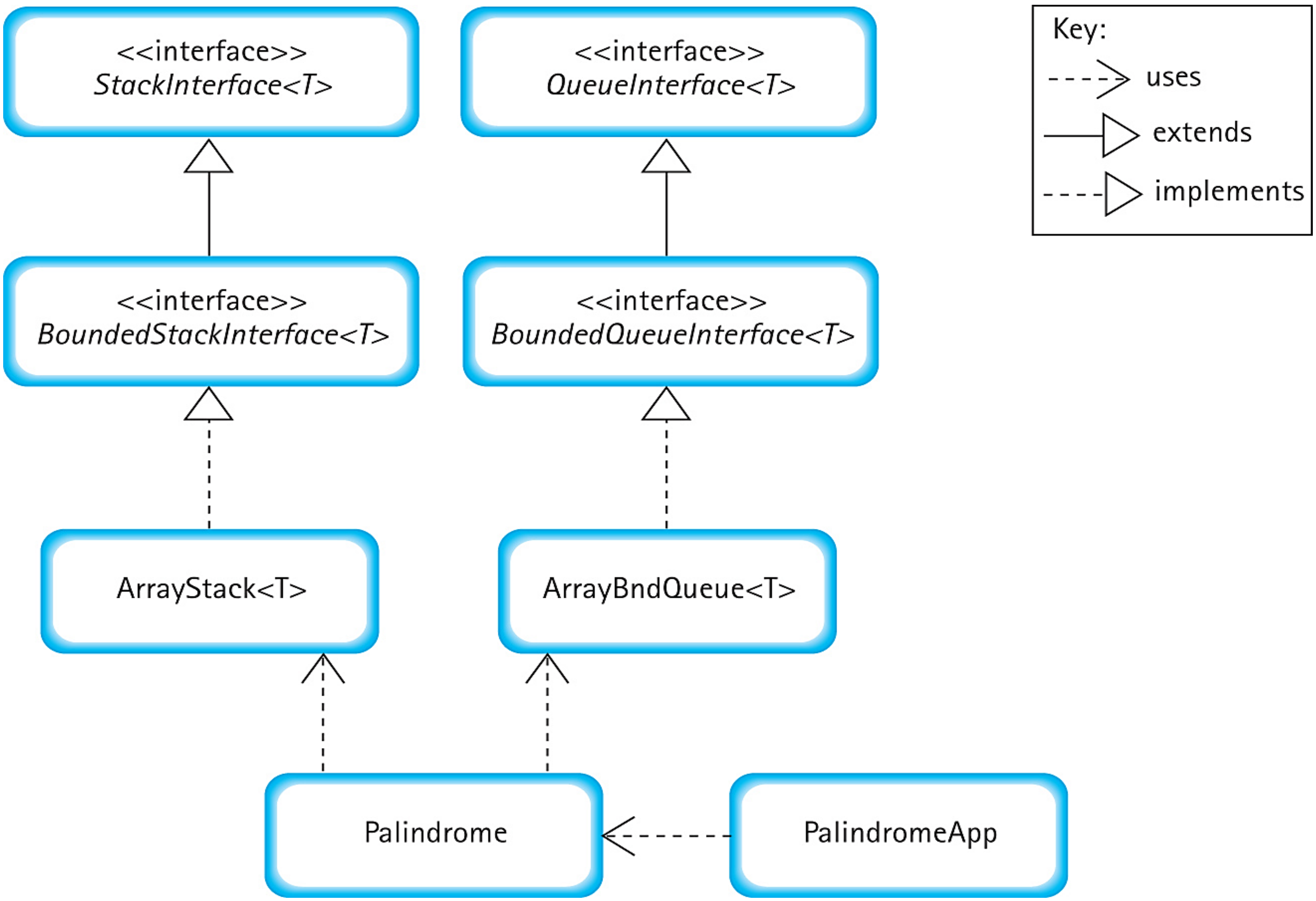
Applications: palindromes

`pop` letters from the stack and **`dequeue`** letters from the queue



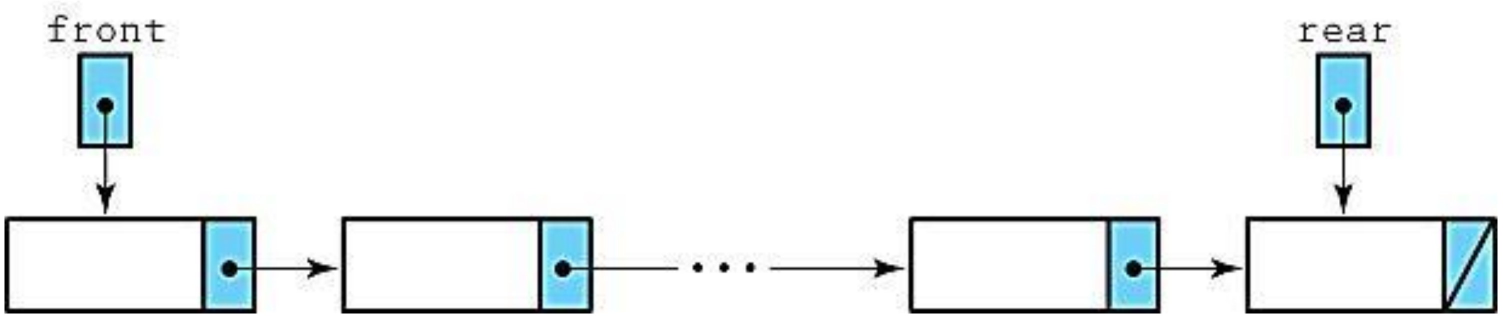
We get "rotor" from both the stack and the queue. They are identical to each other. Thus we've found a palindrome.

The classes


<<interface>>
StackInterface<T>
<<interface>>
QueueInterface<T>
Key:
uses
extends
implements
<<interface>>
BoundedStackInterface<T>
<<interface>>
BoundedQueueInterface<T>
ArrayStack<T>
ArrayBndQueue<T>
Palindrome
PalindromeApp

Linked-based implementations

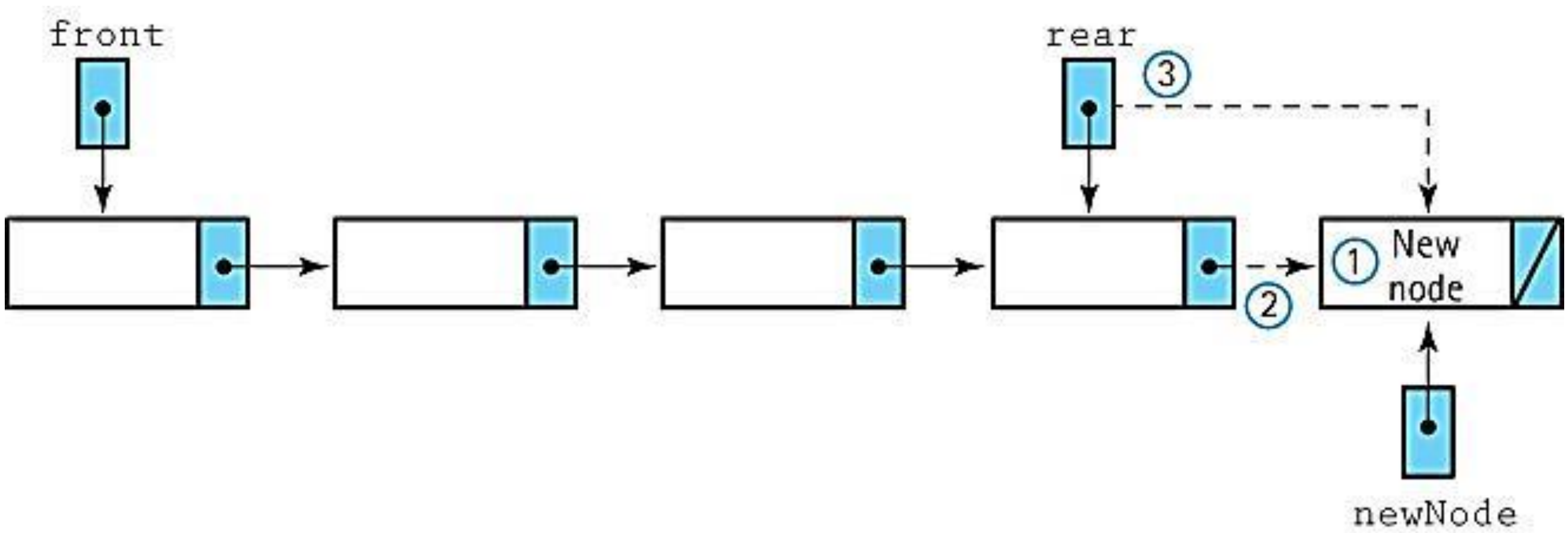
```
import support.LLNode;

public class LinkedUnbndQueue<T> implements
UnboundedQueueInterface<T>
{
  protected LLNode<T> front;   // reference to the front of
                               // this queue
  protected LLNode<T> rear;    // reference to the rear of
                               // this queue
  public LinkedUnbndQueue()
  {
    front = null;
    rear = null;
  }
. . .
```



Linked-based implementations

`Enqueue(element)`

1. Create a node for the new element
2. Insert the new node at the rear of the queue
3. Update the reference to the rear of the queue



Linked-based implementations

Code for **`enqueue`**:

```
public void enqueue(T element)
// Adds element to the rear of this queue.
{
  LLNode<T> newNode = new LLNode<T>(element);

  if (rear == null)
    front = newNode;
  else
    rear.setLink(newNode);

  rear = newNode;
}
```

Linked-based implementations

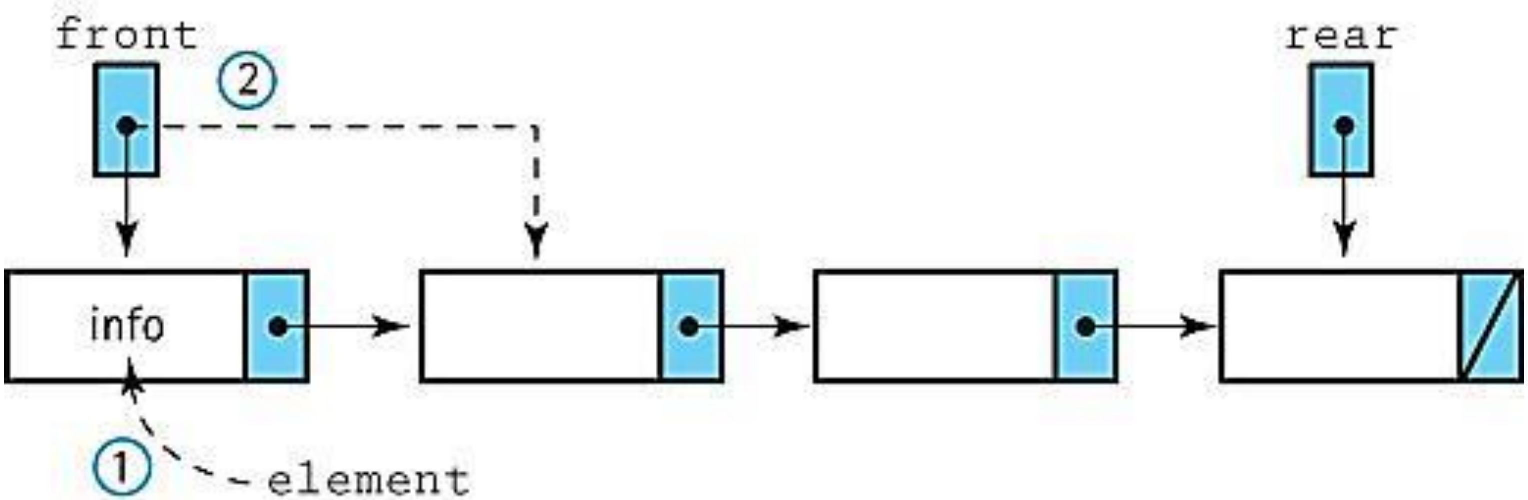
`Dequeue: returns Object`

1. Set element to the information in the front node
2. Remove the front node from the queue

    if the queue is empty

        Set the rear to **`null`**

    return element

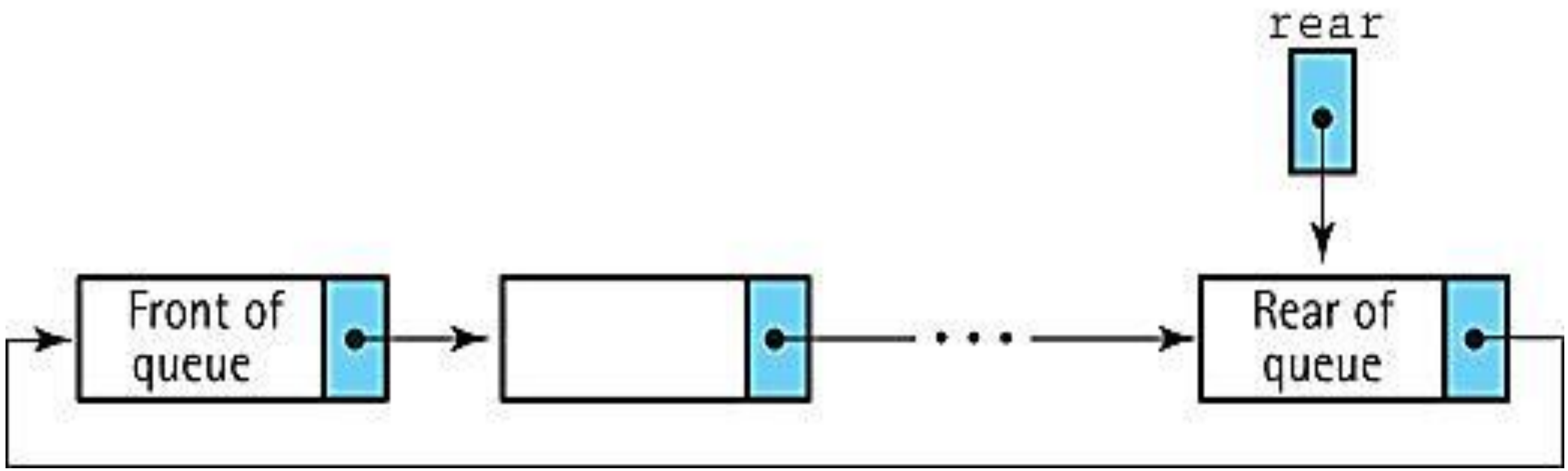


Linked-based implementations

Code for **dequeue**:

```
public T dequeue()
// Throws QueueUnderflowException if this queue is empty,
// otherwise removes front element from this queue and returns
// it.
{
  if (isEmpty())
    throw new QueueUnderflowException("Dequeue attempted on
                                       empty queue.");
  else
  {
    T element;
    element = front.getInfo();
    front = front.getLink();
    if (front == null)
      rear = null;
    return element;
  }
}
```

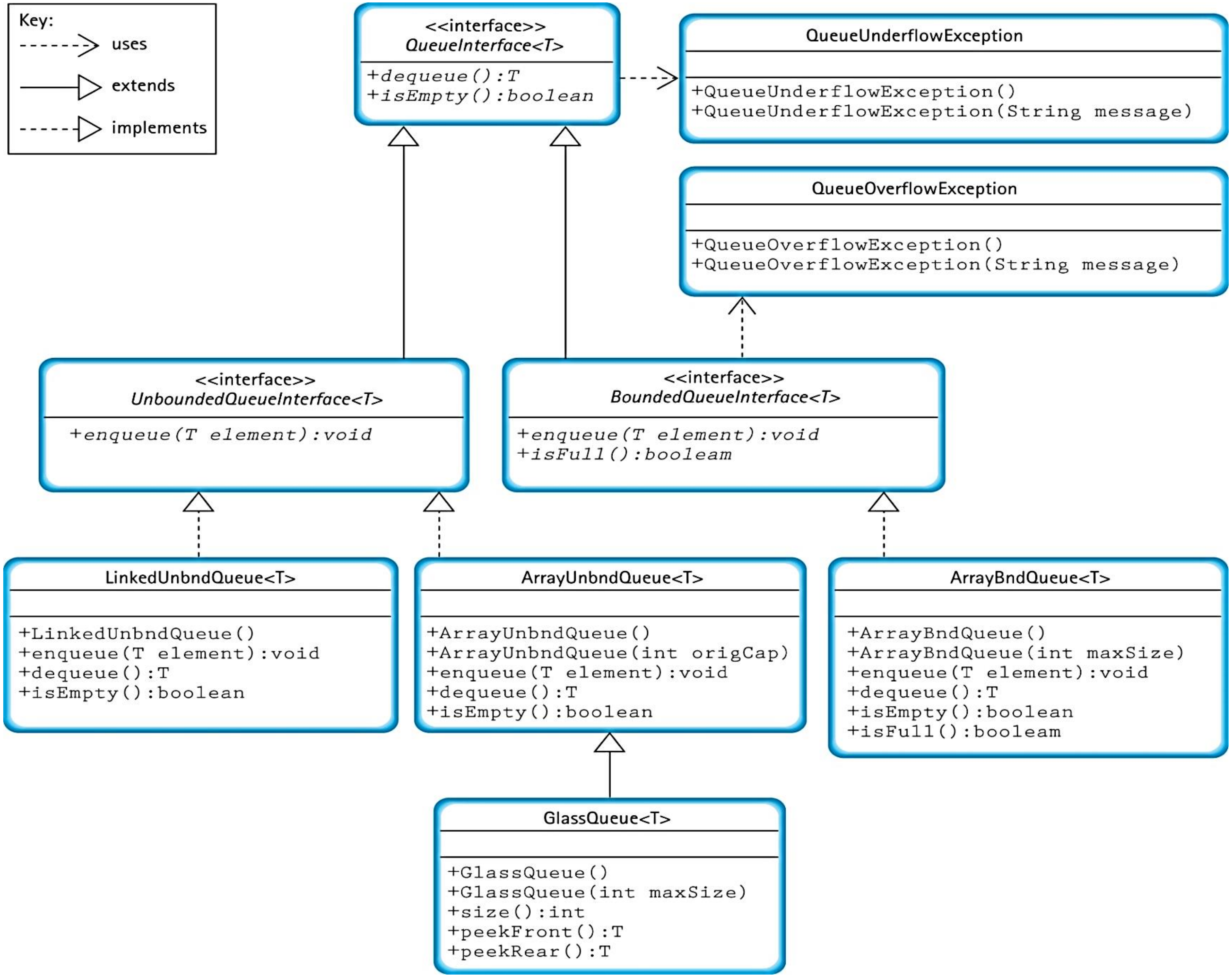
Circular linked Queue



Comparing Queue implementations

- Storage Size
  - Array-based: takes the same amount of memory, no matter how many array slots are actually used, proportional to maximum size.
  - Link-based: takes space proportional to actual size of the queue (but each element requires more space than with array approach).
- Operation efficiency
  - All operations, for each approach, are O(1).
  - Except for the constructors:
    - Array-based: O(N)
    - Link-based: O(1)
- Special Case – For the **`ArrayUnbndQueue`** the size "penalty" can be minimized but the **`enlarge`** method is O(N).

Comparing Queue implementations



Action items

- Read book chapter 5.