Chapter 4: The Queue ADT

CS401

Michael Y. Choi, Ph.D.

Department of Computer Science

Illinois Institute of Technology

Revised Nell Dale Presentation

Chapter 4: The Queue ADT

4.1 – The Queue
4.2 – The Queue Interface
4.3 – Array-Based Queue Implementations
4.4 – An Interactive Test Driver
4.5 – Link-Based Queue Implementations
4.6 – Application: Palindromes
4.7 – Queue Variations
4.8 – Application: Average Waiting Time
4.9 – Concurrency, Interference, and Synchronization

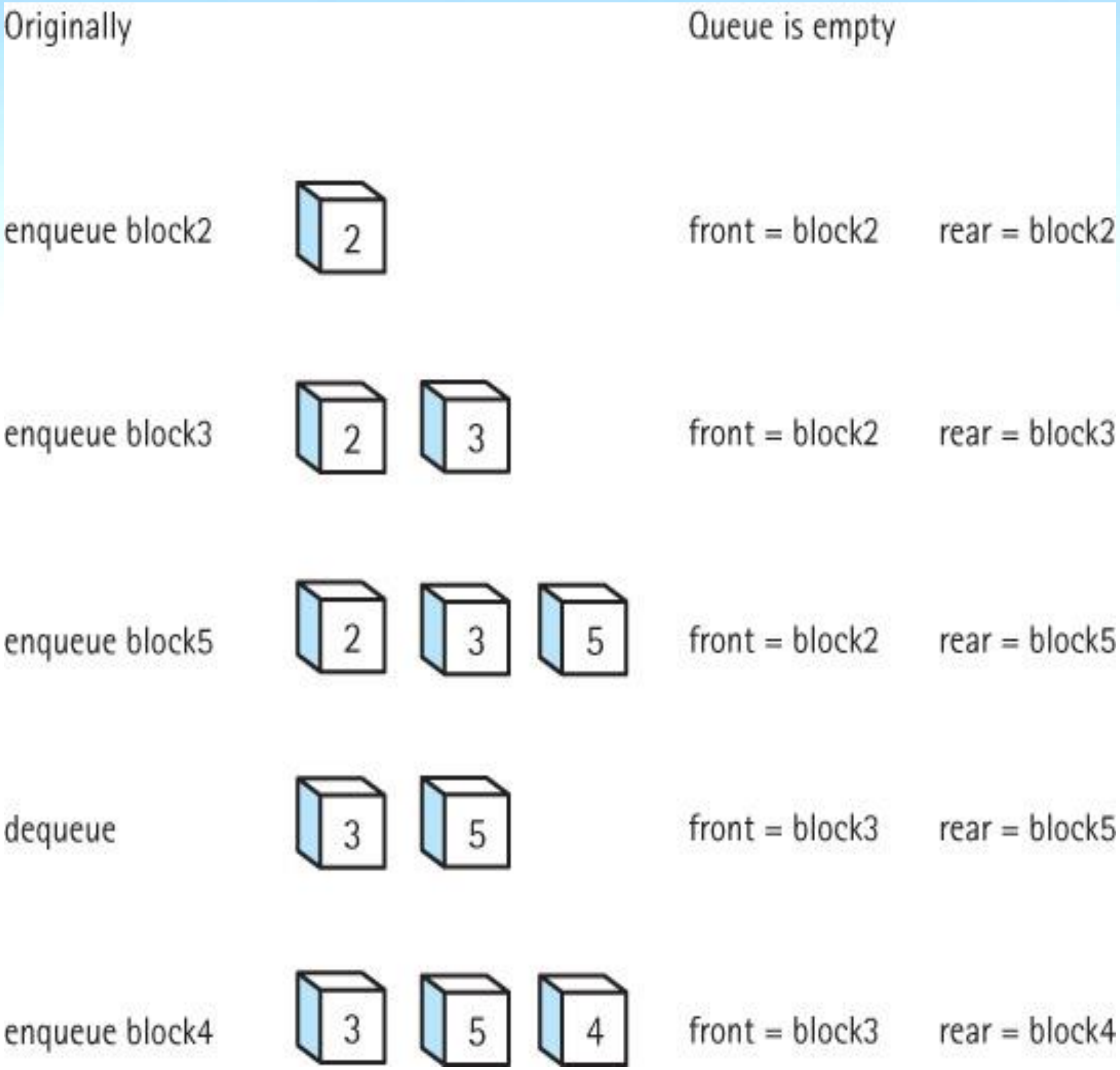
5.1 Queues

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

Operations on Queues

- Constructor
  - new - 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

Effects of Queue Operations



Using Queues

- 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.
- Our software queues have counterparts in real world queues. We wait in queues to buy pizza, to enter movie theaters, to drive on a turnpike, and to ride on a roller coaster. Another important application of the queue data structure is to help us simulate and analyze such real world queues

4.2 The Queue Interface

- We use a similar approach as with the Stack ADT.
- Our queues
  - are generic
  - queue related classes are held in `ch04.queues` package
- we define exceptions for both queue underflow and queue overflow
- we create a `QueueInterface`

QueueInterface

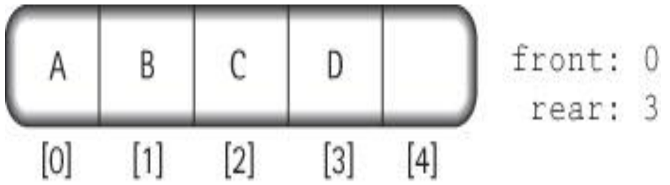
```
package ch04.queues;

public interface QueueInterface<T>
{
   void enqueue(T element) throws QueueOverflowException1;
   // Throws QueueOverflowException if this queue is full;
   // otherwise, adds element to the rear of this queue.

   T dequeue() throws QueueUnderflowException;
   // Throws QueueUnderflowException if this queue is empty;
   // otherwise, removes front element from this queue and returns it.

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

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

   int size();
   // Returns the number of elements in this queue.
}
```

Example Use of a Queue

- Instructors can now review and demonstrate the `RepeatStrings` application found in package `ch04.apps`.

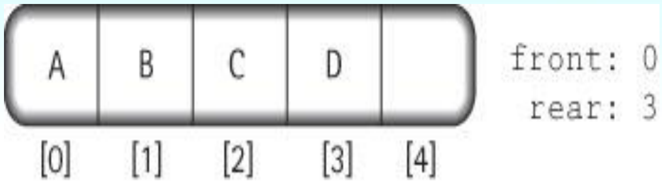
4.3 Array-Based Queue Implementations

- In this section we study two array-based implementations of the Queue ADT
  - a bounded queue version
  - an unbounded queue version
- We simplify some figures by using a capital letter to represent an element's information

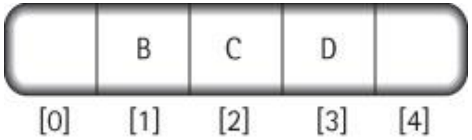


Fixed Front Design

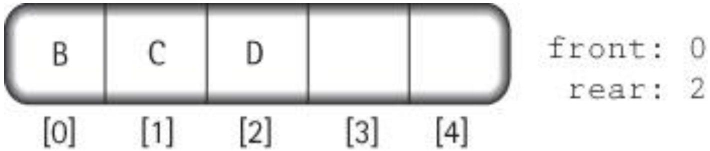
- After four calls to enqueue with arguments ‘A’, ‘B’, ‘C’, and ‘D’:



- Dequeue the front element:



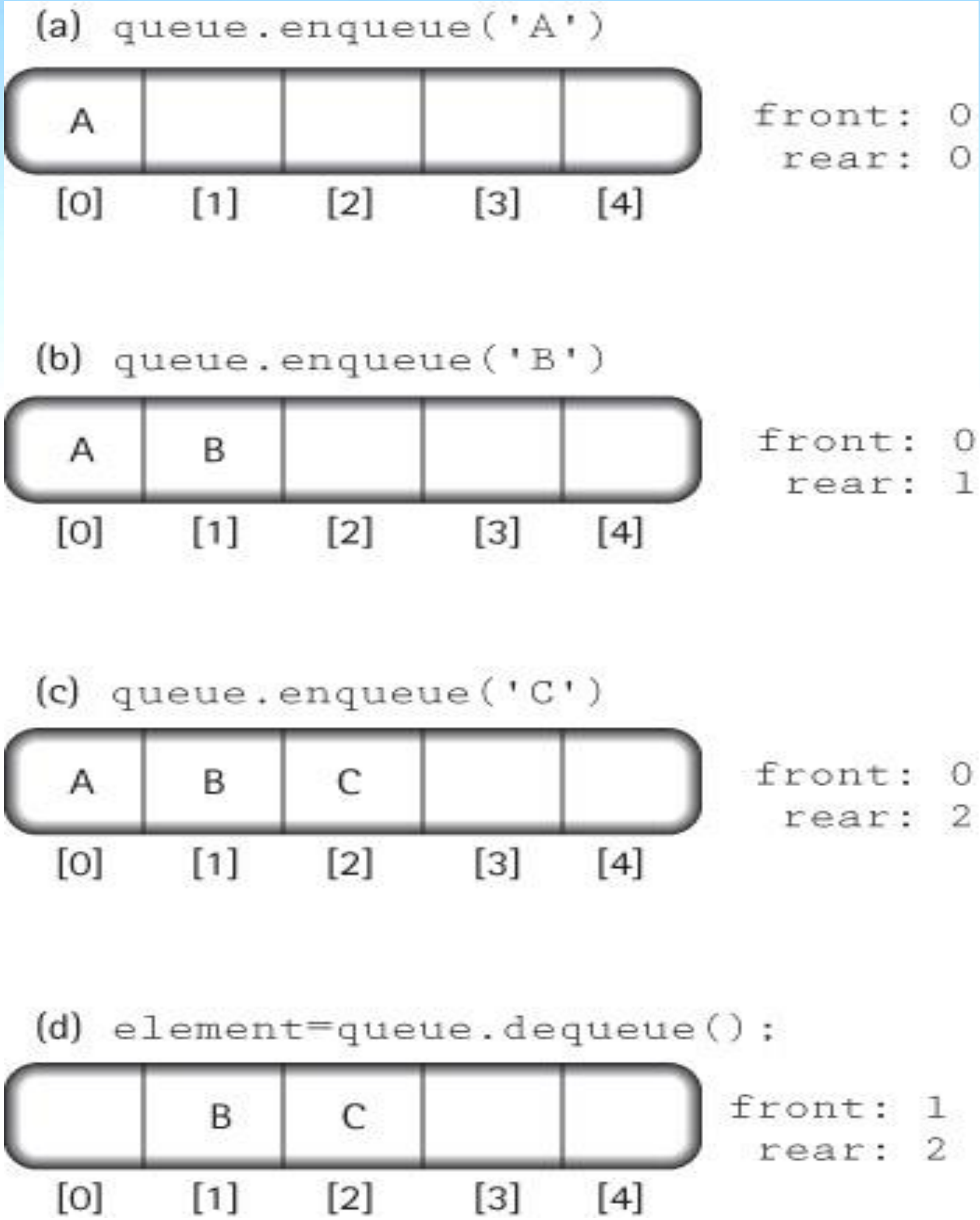
- Move every element in the queue up one slot



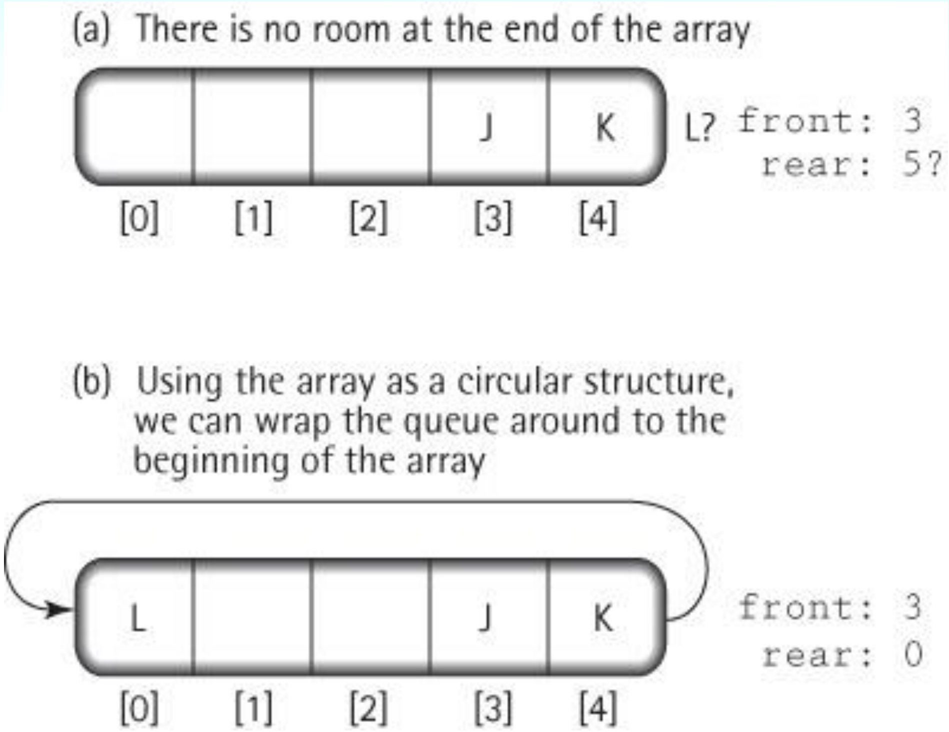
- The dequeue operation is inefficient, so we do not use this approach

Floating Front Design

We use this approach



Wrap Around with Floating Front Design



The `ArrayBoundedQueue` Class

```
package ch04.queues;

public class ArrayBoundedQueue<T> implements QueueInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] elements;           // 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 ArrayBoundedQueue()
  {
    elements = (T[]) new Object[DEFCAP];
    rear =  DEFCAP - 1;
  }

  public ArrayBounddQueue(int maxSize)
  {
    elements = (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) % elements.length;
    elements[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 = elements[front];
    elements[front] = null;
    front = (front + 1) % elements.length;
    numElements = numElements - 1;
    return toReturn;
  }
}
```

Remaining Queue Operations
(observers)

```
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 == elements.length);
}

public int size()
// Returns the number of elements in this queue.
{
  return numElements;
}
```

The `ArrayUnboundedQueue` Class

- 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 enqueing, 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 change the `isFull` method so that it always returns `false`, since an unbounded queue is never full
- The `dequeue` and `isEmpty` methods are unchanged

The `ArrayUnbndQueue` Class

```
package ch04.queues;

public class ArrayUnboundedQueue<T> implements QueueInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] elements;           // array that holds queue elements
  protected int origCap;            // original capacity
  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 ArrayUnboundedQueue()
  {
    elements = (T[]) new Object[DEFCAP];
    rear = DEFCAP - 1;
    origCap = DEFCAP;
  }

  public ArrayUnboundedQueue(int origCap)
  {
    elements = (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[elements.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] = elements[currSmaller];
    currSmaller = (currSmaller + 1) % elements.length;
  }

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

The enqueue operation

```
public void enqueue(T element)
// Adds element to the rear of this queue.
{
  if (numElements == elements.length)
    enlarge();
  rear = (rear + 1) % elements.length;
  elements[rear] = element;
  numElements = numElements + 1;
}
```

4.4 An Interactive Test Driver

- Act as an example use of the `ArrayBoundedQueue` class
- Can be used by students to experiment and learn about the Queue ADT and the relationships among its exported methods
- Will use elements of type `String` to be stored and retrieved from the ADT

The General Approach

```
Prompt for, read, and display test name
Determine which constructor to use
Obtain needed parameters
Instantiate a new instance of the ADT
while (testing continues)
{
  Display a menu of operation choices,
    one choice for each method exported by the ADT
    plus a “stop testing” choice
  Get the user’s choice and
    obtain any needed parameters
  Perform the chosen operation
    if an exception is thrown, catch it and
       report its message
    if a value is returned, report it
}
```

- Instructors can now walk through the code contained in `ITDArrayBoundedQueue.java` found in the `ch04.queues` package and demonstrate the running program.

4.5 Link-Based Queue Implementations

- In this section we develop a link-based implementation of an unbounded queue, and discuss a second link-based approach.
- For nodes we use the same `LLNode` class we used for the linked implementation of stacks.
- After discussing the link-based approaches we compare all of our queue implementation approaches.

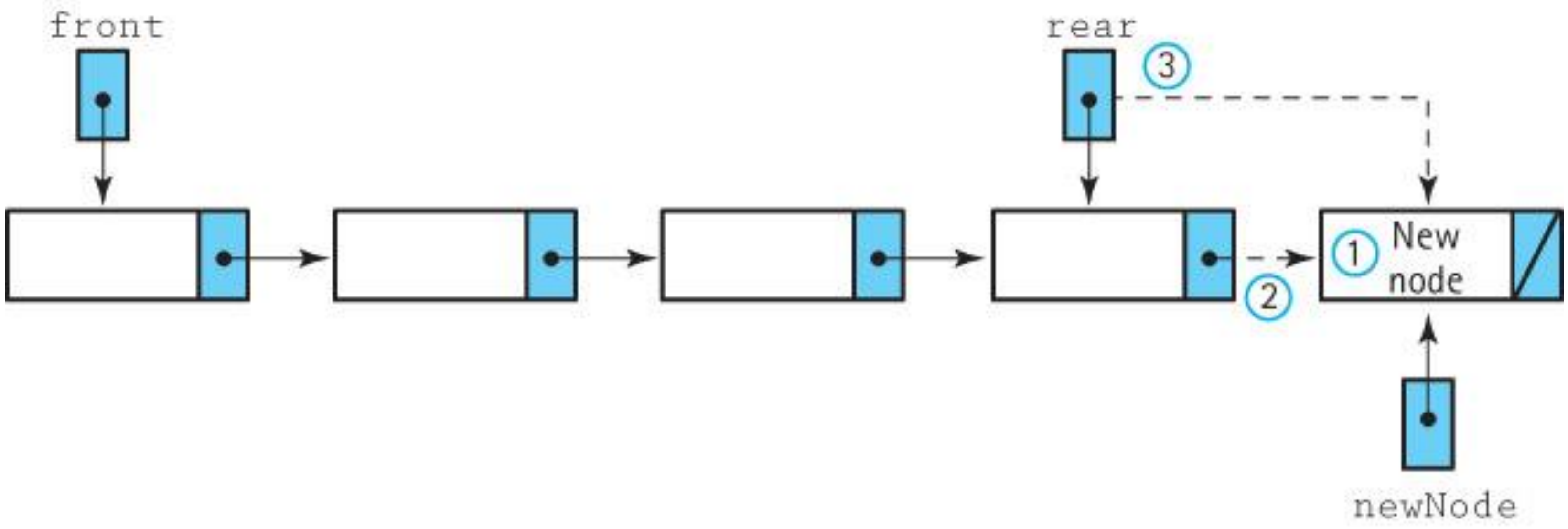
The LinkedQueue Class

```
package ch04.queues;
import support.LLNode;

public class LinkedQueue<T> implements QueueInterface<T>
{
  protected LLNode<T> front;     // reference to the front of this queue
  protected LLNode<T> rear;      // reference to the rear of this queue
  protected int numElements = 0; // number of elements in this queue

  public LinkedQueue()
  {
    front = null; rear = null;
  }
. . .
```

The enqueue operation

Enqueue (element)

1. Create a node for the new element
2. Add the new node at the rear of the queue
3. Update the reference to the rear of the queue
4. Increment the number of elements



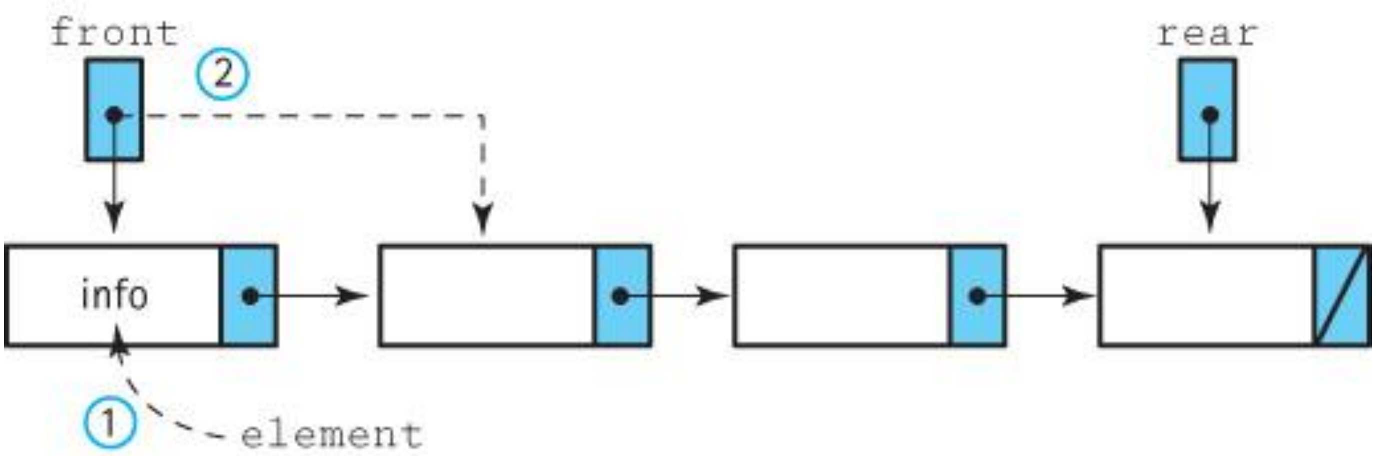
Code for the enqueue method

```
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;
  numElements++;
}
```

The dequeue operation

Dequeue: returns Object

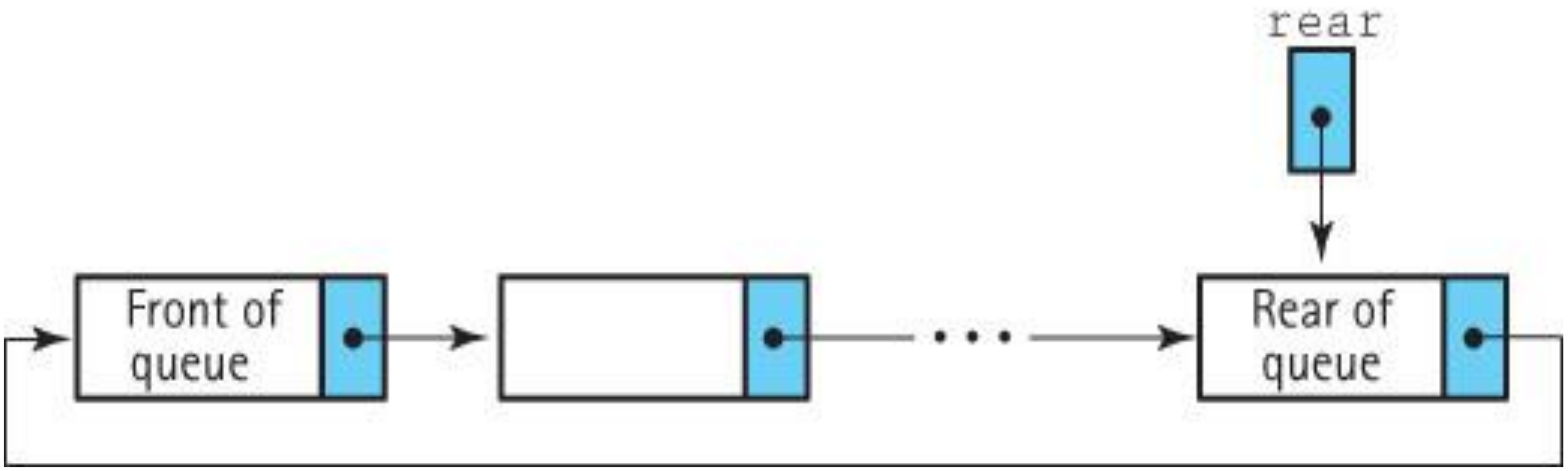
```
1. Set element to the information in the front node
2. Remove the front node from the queue
3. if the queue is empty
       Set the rear to null
4. Decrement the number of elements
5. return element
```



Code for the dequeue method

```
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;
    numElements--;
    return element;
  }
}
```

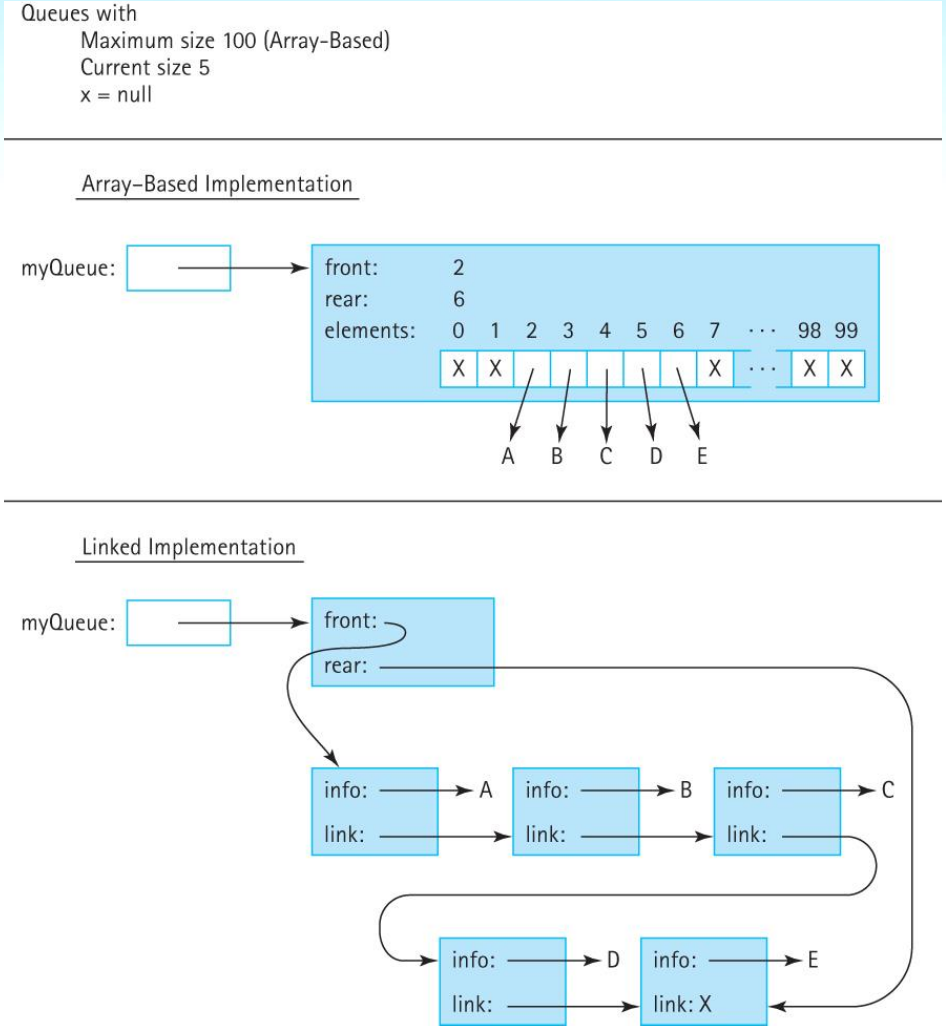
An Alternative Approach - A Circular Linked Queue



Comparing Queue Implementations

- Storage Size
  - Array-based: takes the same amount of memory, no matter how many array slots are actually used, proportional to current capacity
  - 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 `ArrayUnboundedQueue` the size “penalty” can be minimized but the `enlarge` method is O(N)

Comparing Queue Implementations



4.6 Application: Palindromes

- Examples
  - A tribute to Teddy Roosevelt, who orchestrated the creation of the Panama Canal:
    - A man, a plan, a canal—Panama!
  - Allegedly muttered by Napoleon Bonaparte upon his exile to the island of Elba:
    - Able was I ere, I saw Elba.
- Our goal is to write a program that identifies Palindromic strings
  - we ignore blanks, punctuation and the case of letters

The `Palindrome` Class

- 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

The `test` method approach

- The `test` method creates a stack and a queue
- It then repeatedly pushes each input letter onto the stack, and also enqueues 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 pops a letter from the stack and dequeues a letter from the queue
- As long as these letters match each other the entire way through this process, we have a palindrome

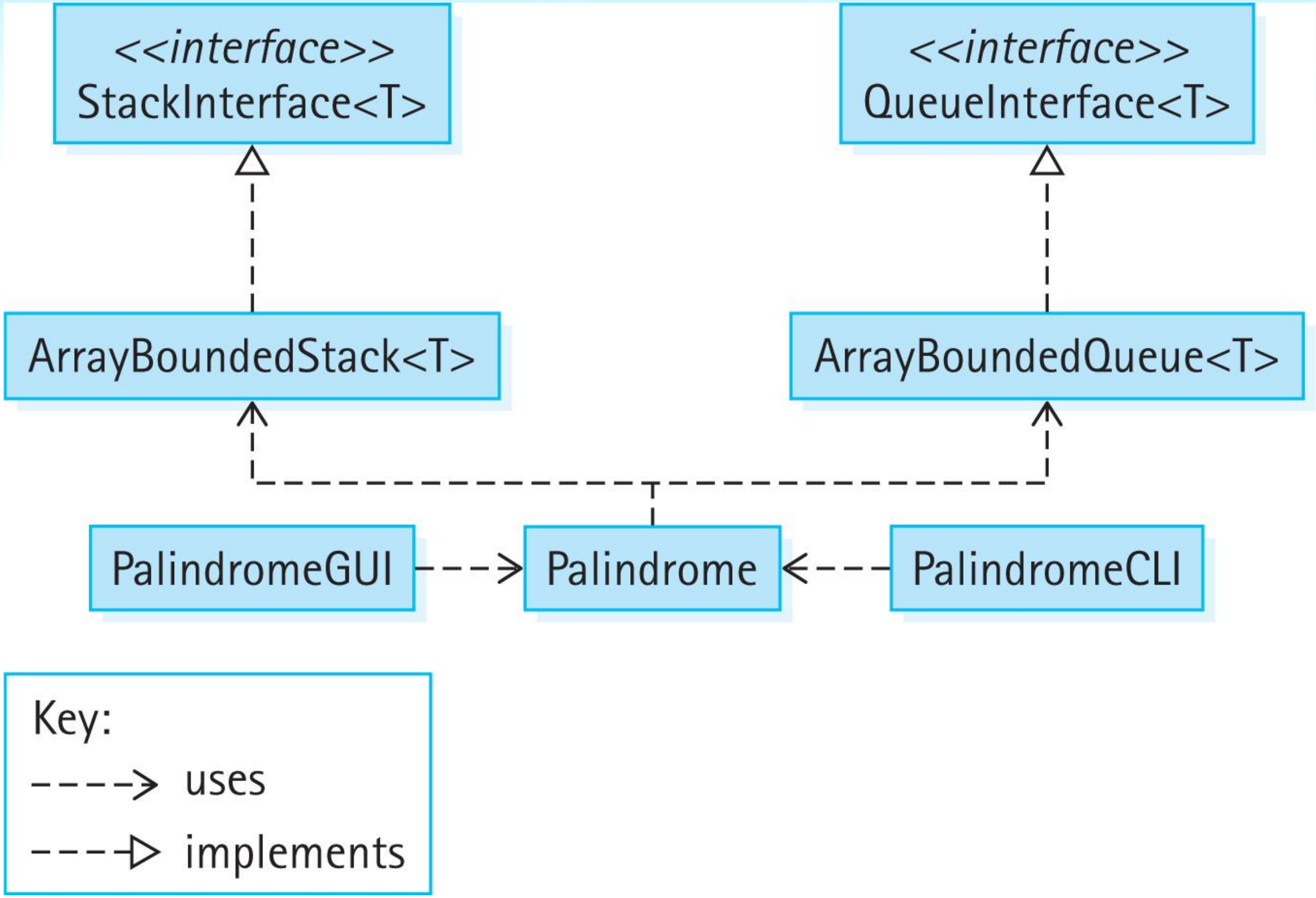
Test for Palindrome (String candidate)

```
Create a new stack
Create a new queue

for each character in candidate
  if the character is a letter
    Change the character to lowercase
    Push the character onto the stack
    Enqueue the character onto the queue

Set stillPalindrome to true

while (there are still more characters in the structures
      && stillPalindrome)
  Pop fromStack from the stack
  Dequeue fromQueue from the queue
  if (fromStack != fromQueue)
    Set stillPalindrome to false

return (stillPalindrome)
```

Code and Demo

- Instructors can now walk through the code contained in `Palindrome.java` in the `ch04.palindromes` package, and `PalindromeCLI.java` and/or `Palindrome.GUI` both in the `ch04.apps` package, and demonstrate the application.

Program Architecture



4.7 Queue Variations

- We consider some alternate ways to define the classic queue operations.
- We look at additional operations that could be included in a Queue ADT, some that allow us to “peek” into the queue and others that expand the access rules
- We review the Java Standard Library queue support.

Exceptional Situations

- Our queues throw exceptions in the case of underflow or overflow.
- Another approach is to prevent the over/underflow from occurring by nullifying the operation, and returning a value that indicates failure
  - boolean enqueue(T element) adds element to the rear of this queue; returns true if element is successfully added, false otherwise
  - T dequeue()returns null if this queue is empty, otherwise removes front element from this queue and returns it

Inheritance of Interfaces

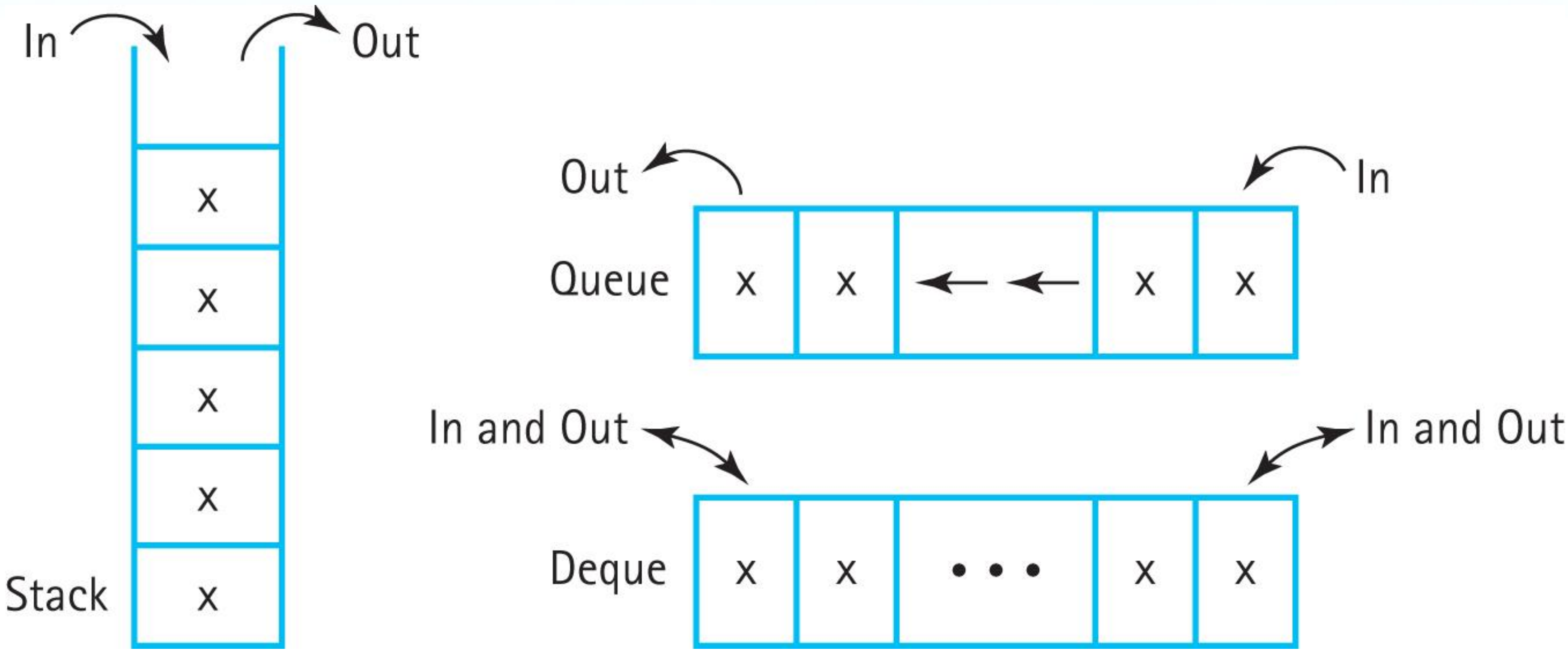
- Java supports inheritance of interfaces.
- In fact, the language supports multiple inheritance of interfaces—a single interface can extend any number of other interfaces.
- Suppose interface B extends interface A. Then a class that implements interface B must provide concrete methods for all of the abstract methods listed in both interface B and interface A.

The Glass Queue

```
//----------------------------------------------------------------------------
// GlassQueueInterface.java      by Dale/Joyce/Weems              Chapter 4
//
// Interface for a class that implements a queue of T and includes
// operations for peeking at the front and rear elements of the queue.
//----------------------------------------------------------------------------
package ch04.queues;
public interface GlassQueueInterface<T> extends QueueInterface<T>
{
  public T peekFront();
  // If the queue is empty, returns null.
  // Otherwise, returns the element at the front of this queue.

  public T peekRear();
  // If the queue is empty, returns null.
  // Otherwise, returns the element at the rear of this queue.
}
```

```
package ch04.queues;
public class LinkedGlassQueue<T> extends LinkedQueue<T>
                                 implements GlassQueueInterface<T>
{
  public LinkedGlassQueue()
  {
    super();
  }

  public T peekFront()
  {
    if (isEmpty())
       return null;
    else
       return front.getInfo();
  }

  public T peekRear()
  {
    if (isEmpty())
      return null;
    else
      return rear.getInfo();
  }
 }
```

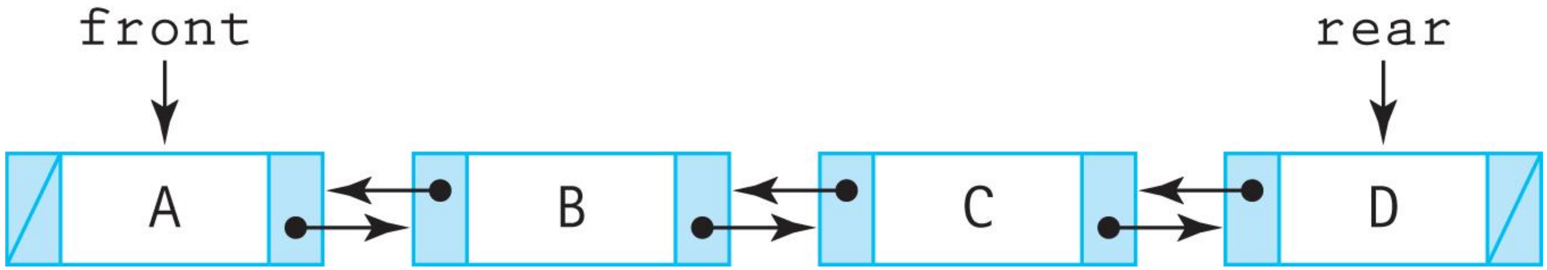
The Double-Ended Queue: Deque



```java
package ch04.queues;
public interface DequeInterface<T>
{
  void enqueueFront(T element) throws QueueOverflowException;
  // Throws QueueOverflowException if this queue is full;
  // otherwise, adds element to the front of this queue.

  void enqueueRear(T element) throws QueueOverflowException;
  // Throws QueueOverflowException if this queue is full;
  // otherwise, adds element to the rear of this queue.

  T dequeueFront() throws QueueUnderflowException;
  // Throws QueueUnderflowException if this queue is empty;
  // otherwise, removes front element from this queue and returns it.

  T dequeueRear() throws QueueUnderflowException;
  // Throws QueueUnderflowException if this queue is empty;
  // otherwise, removes rear element from this queue and returns it.

  boolean isFull();
  boolean isEmpty();
  int size();
}
```

A good approach for implementing Deque

- Double Linked List:



- See `DLLNode` in package `support`

Queues in the Java Standard Library

- A Queue interface was added to the Java Library Collection Framework with Java 5.0 in 2004.
- Elements are always removed from the “front” of the queue.
- Two operations for enqueuing: add, that throws an exception if invoked on a full queue, and offer, that returns a boolean value of false if invoked on a full queue.

Queues in the Java Standard Library

- As with the library Stack, the library Queue was supplanted by the Deque with the release of Java 6.0 in 2006
  - it requires operations allowing for additions, deletions, and inspections at both ends of the queue
- There are four library classes that implement the Deque interface: ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, and LinkedList.

4.8 Application: Average Waiting Time

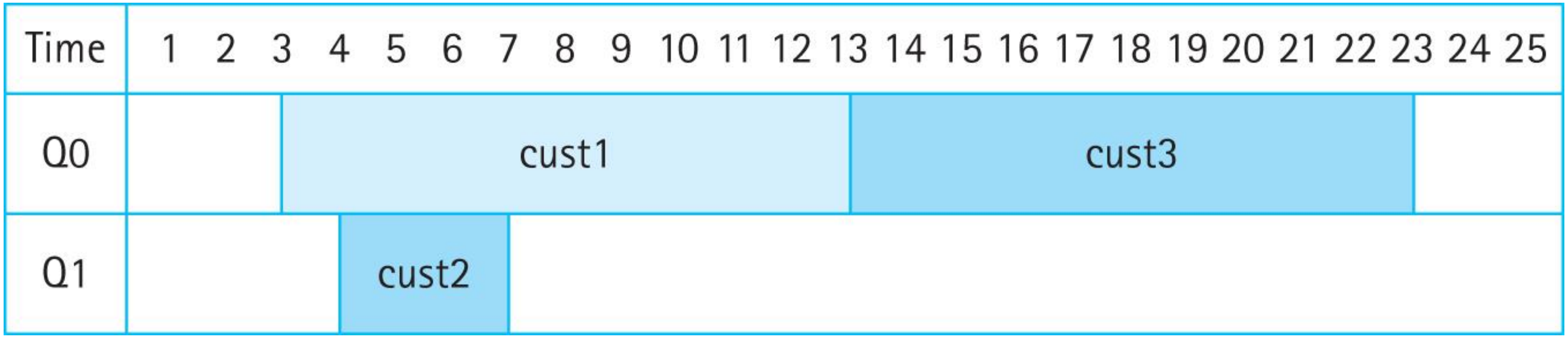
- We create a program that simulates a series of customers arriving for service, entering a queue, waiting, being served, and finally leaving the queue.
- It tracks the time the customers spend waiting in queues and outputs the average waiting time.

Definitions

- Arrival time: the time a customers arrives
- Service time: time customer needs
- Departure time: the time customer leaves
- Turnaround time: Departure time – Arrival time
- Wait time: Turnaround time – Service Time

Simple Example

Customer	Arrival Time	Service Time
1	3	10
2	4	3
3	5	10
4	25	7

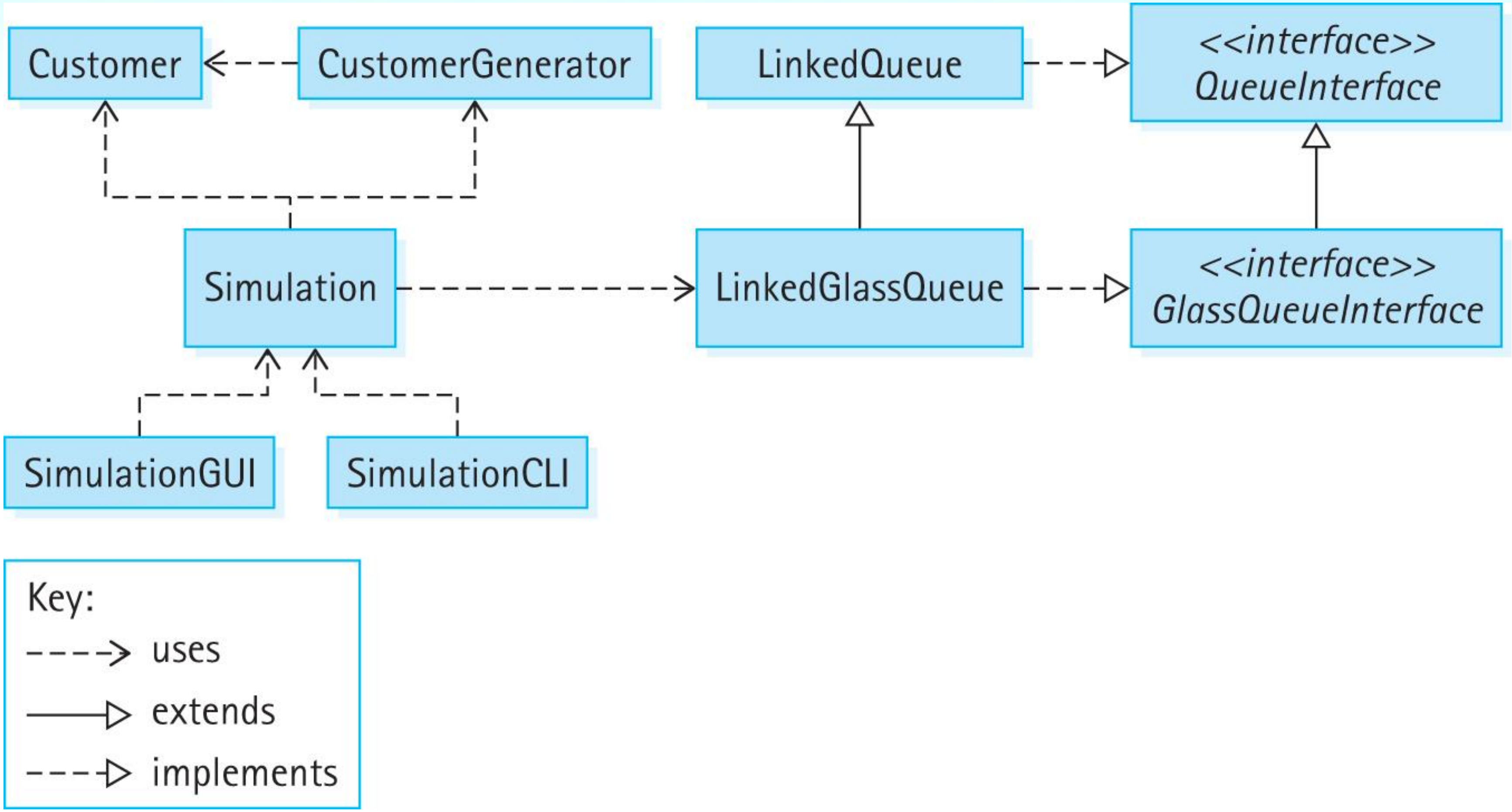


And so on

Simple Example Results

Customer	Arrival Time	Service Time	Finish Time	Wait Time
1	3	10	13	0
2	4	3	7	0
3	5	10	23	8
4	25	7	32	0

Program Architecture



Sample Run of `SimulationCLI`

```
Enter minimum interarrival time: 0
Enter maximum interarrival time: 10
Enter minimum service time: 5
Enter maximum service time: 20
Enter number of queues: 2
Enter number of customers: 2000
Average waiting time is 1185.632

Evaluate another simulation instance? (Y=Yes): y
Enter number of queues: 3
Enter number of customers: 2000
Average waiting time is 5.7245

Evaluate another simulation instance? (Y=Yes): n
Program completed.
```

4.9 Concurrency, Interference, and Synchronization

- Multitask: Perform more than one task at a time
- Concurrency: Several interacting code sequences are executing simultaneously
  - through interleaving of statements by a single processor
  - through execution on several processors

Counter Class

```
//----------------------------------------------------------------------
// Counter.java by Dale/Joyce/Weems Chapter 4
//
// Tracks the current value of a counter.
//----------------------------------------------------------------------
package ch04.threads;
public class Counter
{
    private int count;
    public Counter()
    {
        count = 0;
    }

    public void increment()
    {
        count++;
    }

    public int getCount()
    {
        return count;
    }
}
```

Demo One - Basic

```
package ch04.concurrency;
import ch04.threads.*;

public class Demo01
{
  public static void main(String[] args)
  {
     Counter c = new Counter();
     c.increment();
     c.increment();
     c.increment();
     System.out.println("Count is: " + c.getCount());
   }
 }
```

The output of the program: Count is: 3

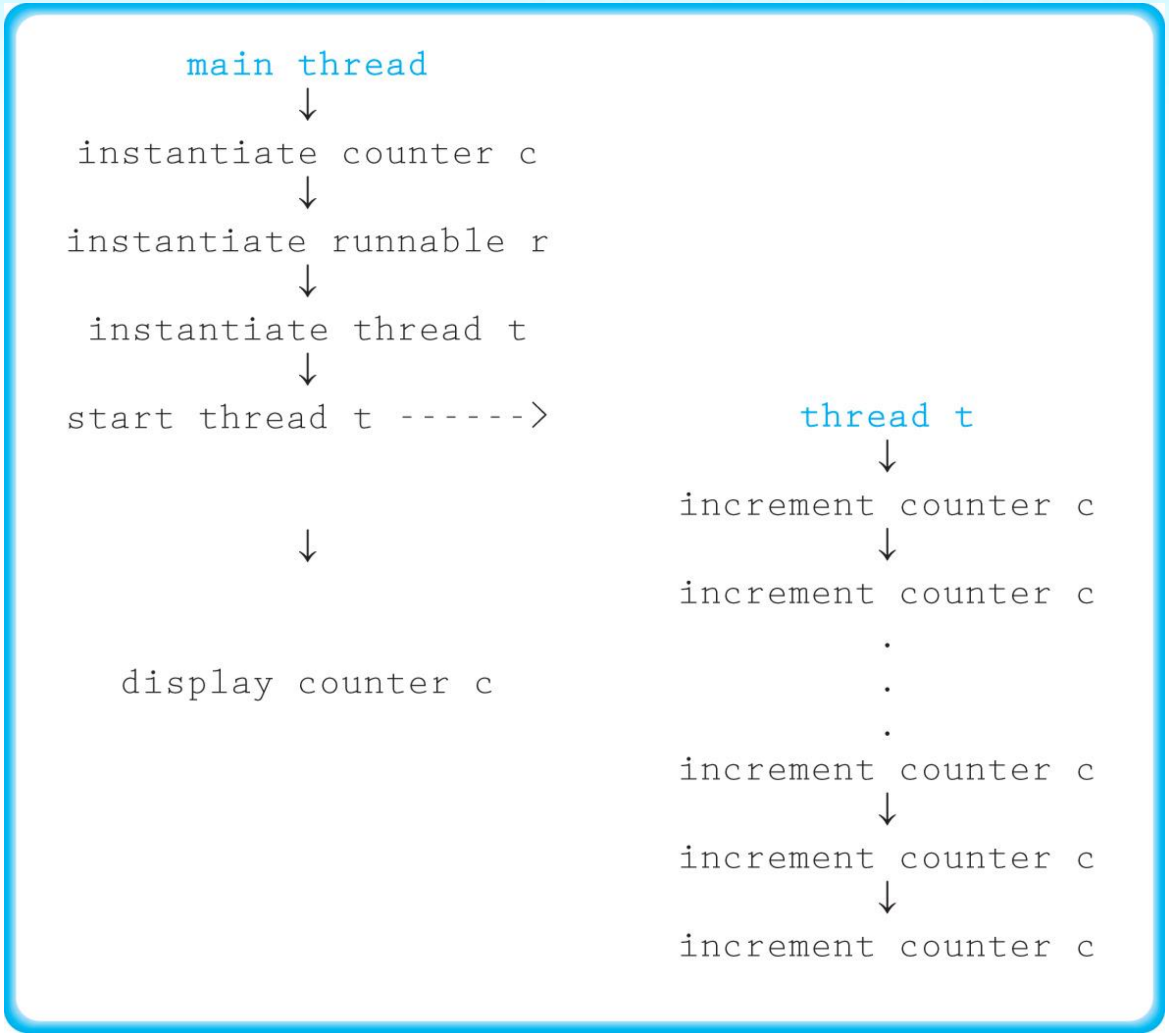
Demo Two - Threads

```
package ch04.threads;
public class Increase
        implements Runnable
{
  private Counter c;
  private int amount;
  public Increase (Counter c, int amount)
  {
    this.c = c; this.amount = amount;
  }

  public void run()
  {
  for (int i = 1; i <= amount; i++)
    c.increment();
  }
}
```

```
package ch04.concurrency;
import ch04.threads.*;
public class Demo02
{
  public static void main(String[] args) throws
    InterruptedException
  {
    Counter c = new Counter();
    Runnable r = new Increase(c, 10000);
    Thread t = new Thread(r);
    t.start();
    System.out.println("Count is: " + c.getCount());
  }
}
```

Output Varies: 86, 3024, 457 ????

Demo Two - Threads



Demo Three - Join

```
package ch04.threads;
public class Increase
        implements Runnable
{
  private Counter c;
  private int amount;
  public Increase (Counter c, int amount)
  {
    this.c = c; this.amount = amount;
  }

  public void run()
  {
  for (int i = 1; i <= amount; i++)
    c.increment();
  }
}
```

```
package ch04.concurrency;
import ch04.threads.*;
public class Demo03
{
  public static void main(String[] args) throws
    InterruptedException
  {
    Counter c = new Counter();
    Runnable r = new Increase(c, 10000);
    Thread t = new Thread(r);
    t.start();
    t.join();
    System.out.println("Count is: " + c.getCount());
  }
}
```

Output is 10000

Demo Four - Interference

```java
package ch04.threads;
public class Increase
        implements Runnable
{
   private Counter c;
   private int amount;
   public Increase (Counter c, int amount)
   {
      this.c = c; this.amount = amount;
   }

   public void run()
   {
   for (int i = 1; i <= amount; i++)
      c.increment();
   }
}
```

```java
package ch04.concurrency;
import ch04.threads.*;
public class Demo04
{
   public static void main(String[] args)
         throws InterruptedException
   {
      Counter c = new Counter();
      Runnable r1 = new Increase(c, 5000);
      Runnable r2 = new Increase(c, 5000);
     Thread t1 = new Thread(r1);
     Thread t2 = new Thread(r2);
     t1.start();   t2.start();
     t1.join();    t2.join();
     System.out.println("Count is: " + c.getCount());
   }
}
```

Output Varies: 9861, 9478 ????

Demo Four - Interference

Thread t1	Thread t2
Step 1: obtains value 12	
	Step 2: obtains value 12
Step 3: increments value to 13	
Step 4: stores the value 13	
	Step 5: increments value to 13
	Step 6: stores the value 13

Demo Five - Synchronization

```
// SyncCounter.java
// Tracks the current value of a counter.
// Provides synchronized access

package ch04.threads;
public class SyncCounter
{
   private int count;
   public SyncCounter()
   {
      count = 0;
   }

   public synchronized void increment()
   {
      count++;
   }

   public int getCount()
   {
       return count;
   }
}
```

```
// The IncreaseSync class is identical to Increase
// cass except that it accepts a SyncCounter instead
// of Counter as its first parameter
package ch04.concurrency;
import ch04.threads.*;
public class Demo05
{
   public static void main(String[] args) throws
       InterruptedException
   {
      SyncCounter sc = new SyncCounter();
      Runnable r1 = new IncreaseSync(sc, 5000);
      Runnable r2 = new IncreaseSunc(sc, 5000);
      Thread t1 = new Thread(r1);
      Thread t2 = new Thread(r2);
      t1.start(); t2.start();
      t1.join(); t2.join();
      System.out.println("Count is: " + sc.getCount());
   }
}
```

Output is 10000

A Synchronized Queue

- Similarly the synchronized keyword can be used to control access to an entire or selected parts of a data structure
- See the subsection “A Synchronized Queue”

Our Queue Architecture