Data Structures and Algorithms

BSc. In Computing

Semester 5 Lecture 4

Lecturer: Dr. Simon McLoughlin

Queues

The abstract Data Type Queue

Simple Applications of the ADT Queue
- Reading a String of Characters
- Recognising Palindromes

Implementations of the ADT Queue
- An array-based implementation
- An implementation that uses the ADT List
- A reference-based implementation
- Comparing Implementations

A summary of position ADTs

The behaviour of a **queue** is characterised by the property of first-in, first-out. (**FIFO**)

- In this lecture we define the operations on the queue.
- We also discuss several strategies for implementing queues

The Abstract Data Type Queue

A queue is like a line of people.

- The first person to join a queue is the first person served and therefore the first person to leave the queue.

New items **enter** a queue at its back

Items **leave** a queue at its front

Operations on a queue occur only at its two ends

This gives the FIFO behaviour

The ADT Queue Operations

1. **Create** an empty queue
2. **Determine** whether a queue is empty
3. **Add** a new item into the queue
4. **Remove** from the queue the item that was added earliest
5. **Remove all** the items from the queue
6. **Retrieve** from the queue the item that was added earliest

Queues in real world situations and computer science

Queues are appropriate for very many **real world situations** where queuing and waiting for a service are involved.

- Queuing in a shop
- Queuing in a cafe
- Queuing in a bank
- Queuing at an ATM

They lend themselves very well to use in **computer science**.

- When you print a document from your PC, the lines of text etc are sent to the printer faster than the printer can print.

- To avoid printing difficulties, the lines are instead sent to a software-based spooler, that is, a software-based queue.

- This spooler removes jobs in FIFO order and trickle feeds the data in a print job to the printer.

Specification for the ADT queue Operations

createQueue()
// creates an empty queue

isEmpty()
// determines whether a queue is empty

enque(newItem) throws QueueException
// adds newItem at the back of a queue.
// Throws QueueException if the operation is not successful.

dequeue( ) throws QueueException
// retrieves and removes the front of a queue the item
// that was added earliest
// Throws QueueException if the operation is not successful.

dequeueAll()
// remove alltems from the front of a queue

peek() throws QueueException
// retrieves the front of a queue, that is,
// retrieves the item that was added earliest.
// Throws QueueException if the operation is not successful.
// The queue is unchanged

The diagram illustrates these operations with a queue of integers.

Notice that:

- **enqueue** inserts an item at the **back** of the queue
- **peek** looks at the item at the **front** of the queue
- **dequeue** deletes the item at the **front** of the queue

Operation	Queue after operation	Front
queue.createQueue()		
queue.enqueue(5)	5	
queue.enqueue(2)	5 2	
queue.enqueue(7)	5 2 7	
queueFront = queue.peek() (queueFront is 5)	5 2 7	
queueFront = queue.dequeue() (queueFront is 5)	5 2 7	
queueFront = queue.dequeue() (queueFront is 2)	2 7	

Some Queue Operations

Simple Applications of the ADT Queue

Reading a String of Characters

- When you enter characters at a keyboard, the system must retain them in the order in which you typed them.

- A queue could be used for this purpose.

Pseudocode

```
// read a string of characters from a single line of input into a queue
queue.createQueue()
while (not end of line)
{
     read a new character ch
     queue.enqueue( ch )
} //end while
```

Once the characters are in a queue the system can process them as necessary.

Recognising palindromes

A **palindrome** is a string of characters that read the same way from left to right and vice versa.

- navan
- radar
- abcba

In a previous lecture we learned that a **stack can be used to reverse the order** of occurrences.

A queue can be used to **preserve the order** of occurrences.

Therefore a queue and a stack can be used to determine whether a string is a palindrome.

Solution

Traverse the character string from left to right inserting each character into a **queue** and a **stack**.

- The **first** character in the string is at the **front of the queue**
- The **last** character in the string is at the **top of the stack**

Therefore:

- Characters removed **via a dequeue operation** from the **queue** (**FIFO**) will occur in the order in which they appear in the string

- Characters removed **via a pop operation** from the **stack** (**LIFO**) will occur in the opposite order.

We can then easily compare the characters at the front of the queue and the characters at the top of the stack.

If they are the **same**, we can delete them.

- This process is repeated until the ADTs become empty:

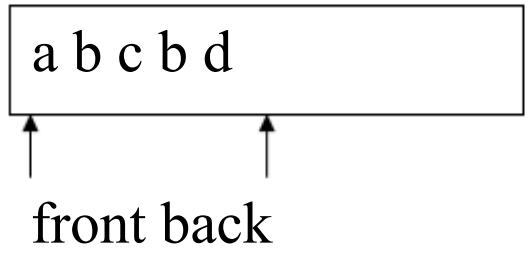
  → **palindrome**

If the two characters are **not the same:**

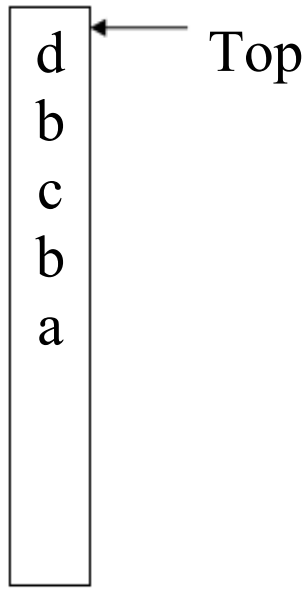
  → the string is **not a palindrome**

String: **abcbd**

Queue:



Stack:



The results of inserting a string into both a queue and a stack

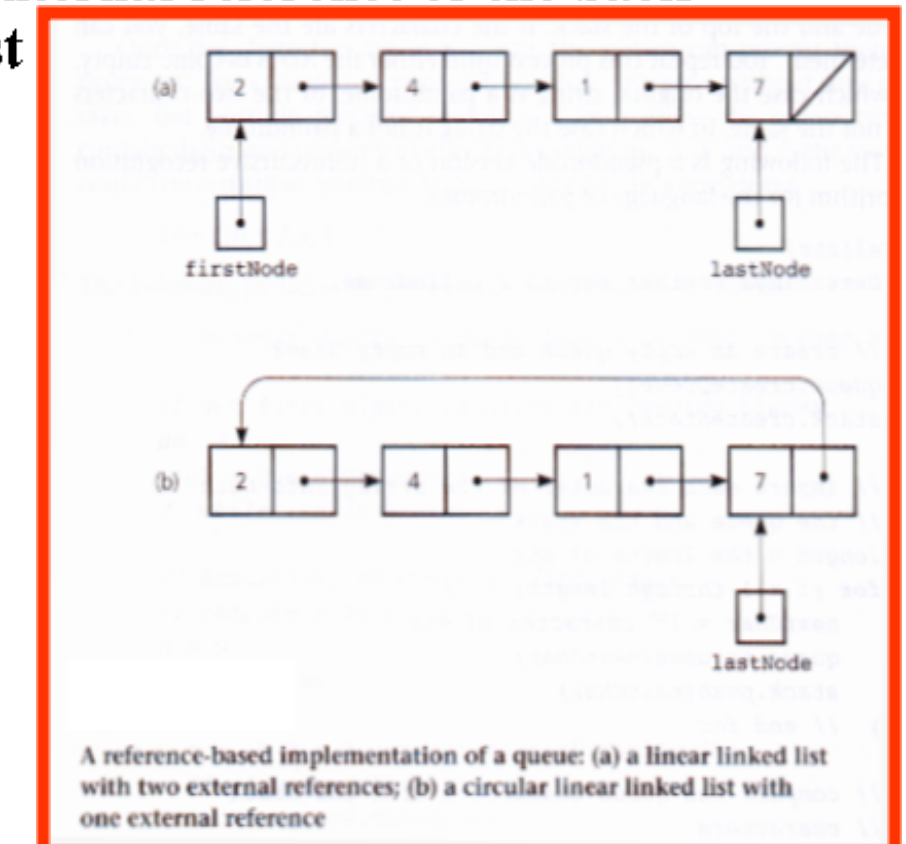
```
isPal(str)
// determines whether  str is a palindrome

1       //create an empty queue and an empty stack
        queue.createQueue()
        stack.createStack()
2       //insert each character of the string into both the stack and the queue
        length = the length of  str
        for (k = 1 through length
        {
              nextChar =  kth character of  str
              queue.enque(nextChar)
              stack.push(nextChar)
        } //end for
3       //compare the queue characters with the stack characters
        charactersAreEqual = true
        while (queue is not empty and  charactersAreEqual is true)
        {
              queueFront =  queue.dequeue()
              stackTop = stack.pop()
              if (queueFront not equal to  stackTop)
              {
                    charactersAreEqual = false
              } //end if
        } //end while
return charactersAreEqual
```

Implementations of the ADT Queue

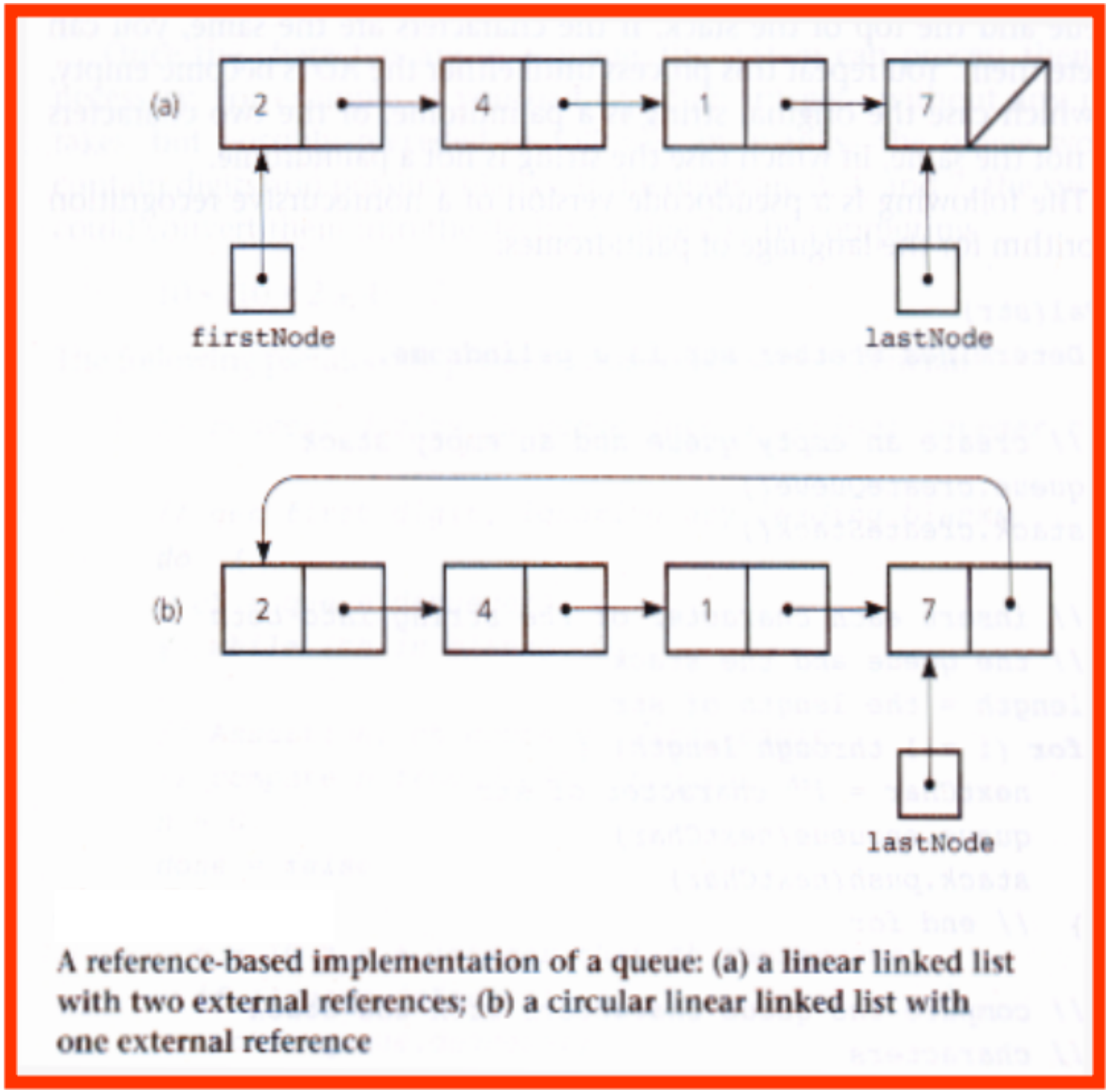
A Reference-based Implementation of a Queue

A reference-based implementation could use a linear linked list with two external references, one to the **front** and one to the **back**.

We can, however, get by with a **single external reference to the back** if we decide to use a **circular linked list**



A reference-based implementation of a queue: (a) a linear linked list with two external references; (b) a circular linear linked list with one external reference

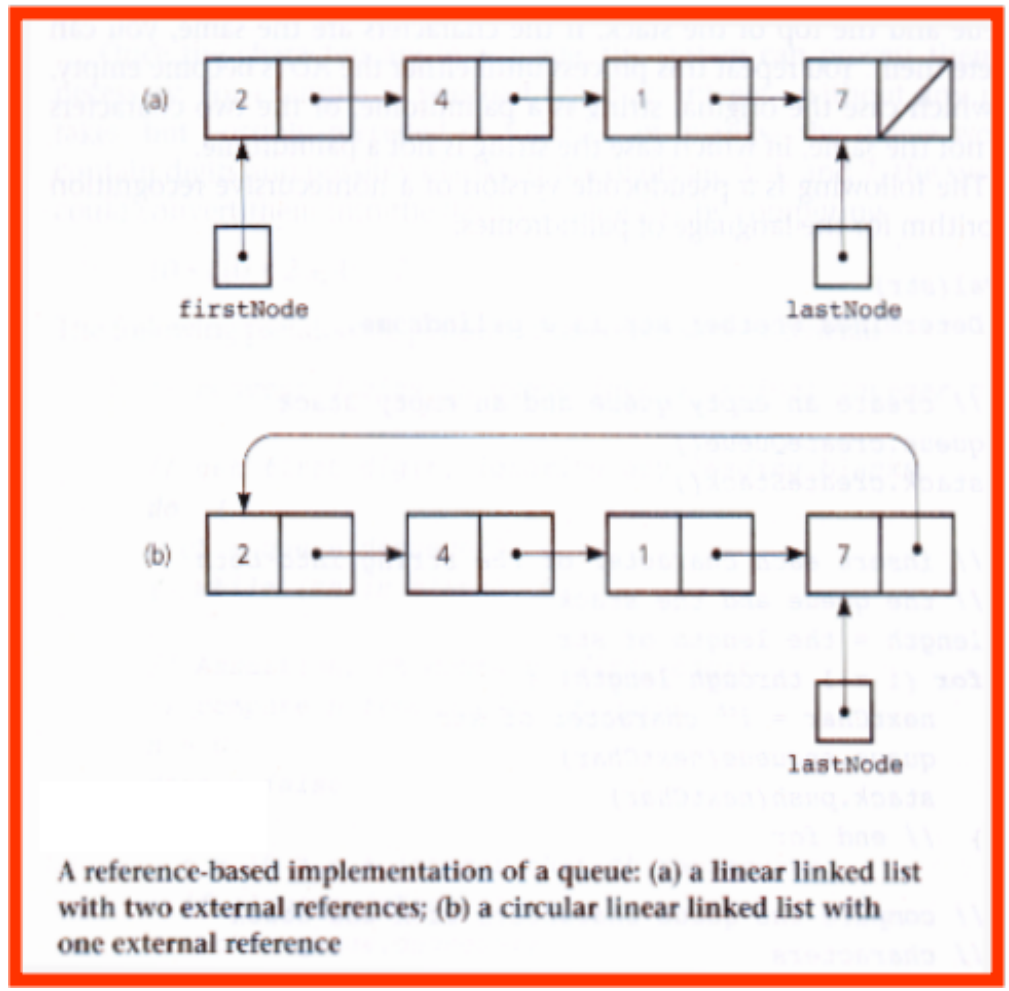


A reference-based implementation of a queue: (a) a linear linked list with two external references; (b) a circular linear linked list with one external reference

When a circular linked list represents a **queue**, …

- the **node at the back** of the queue references the **node at the front**
- **lastNode** references the node at the **back** of the queue
- **lastNode.getNext()** references the **node at the front**

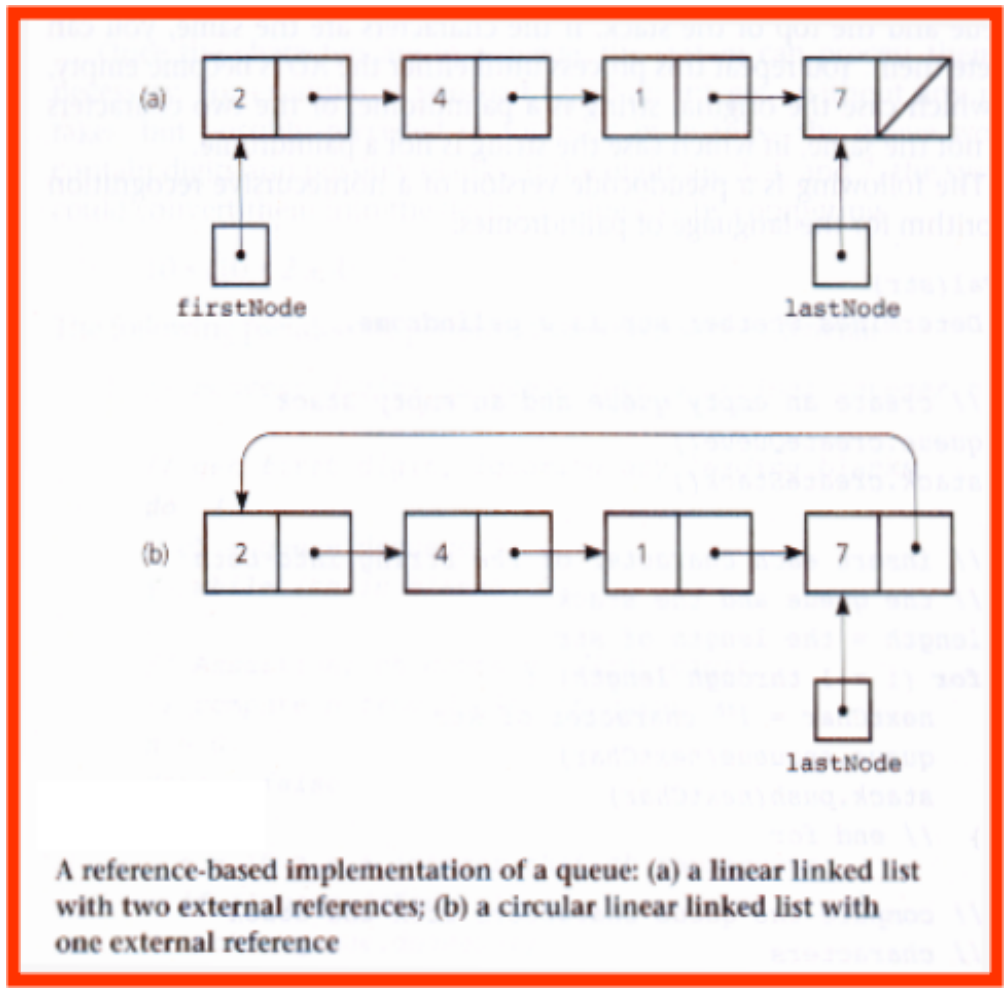
Insertion at the back and deletion from the front are easy.



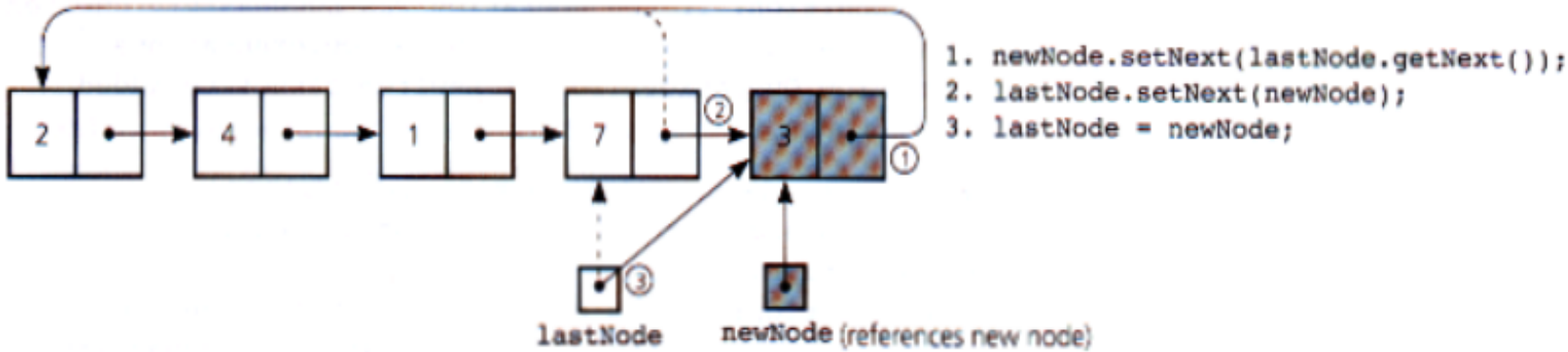
A reference-based implementation of a queue: (a) a linear linked list with two external references; (b) a circular linear linked list with one external reference

Inserting a new node, which **newNode** references, at the **back of the queue** requires three references changes:

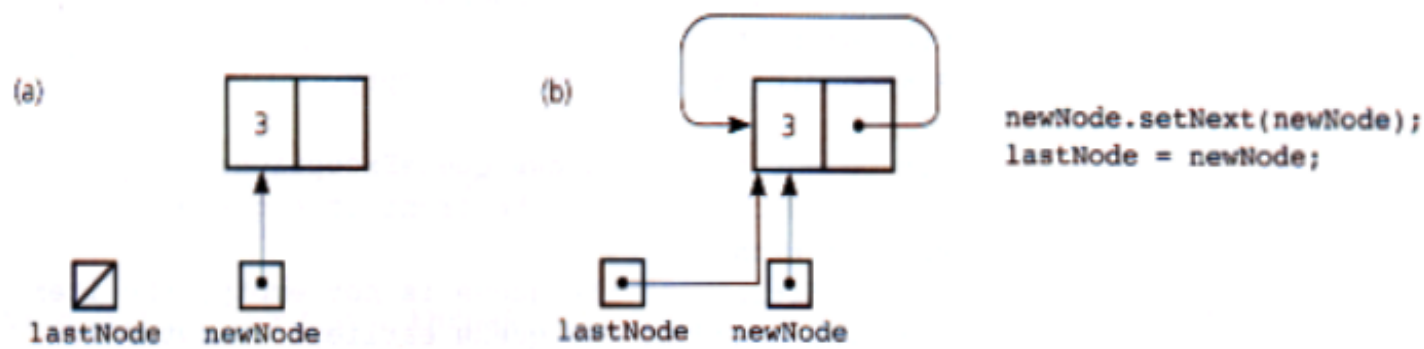
- The **next** reference in the **new** node,
- The **next** reference in the **back** node
- The **external** reference **lastNode**



A reference-based implementation of a queue: (a) a linear linked list with two external references; (b) a circular linear linked list with one external reference



Inserting an item into a nonempty queue

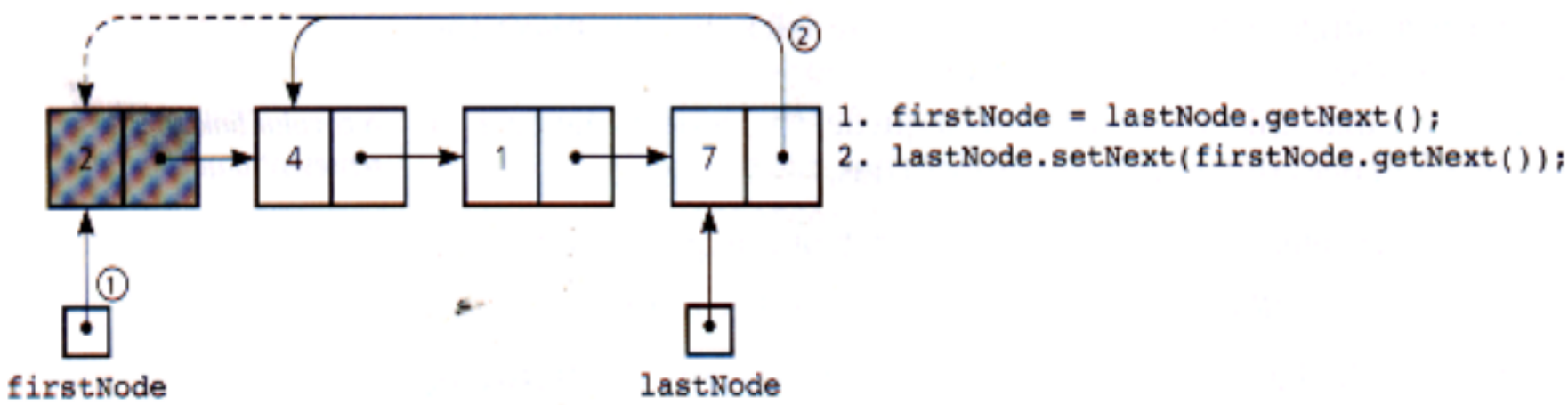


Inserting an item into an empty queue: (a) before insertion; (b) after insertion

- The diagram above shows these changes and indicates the order in which they must occur.
- The **addition** of an item to an **empty queue** is a special case.

Deletion from the **front** of the queue is simpler that **insertion** at the **back**

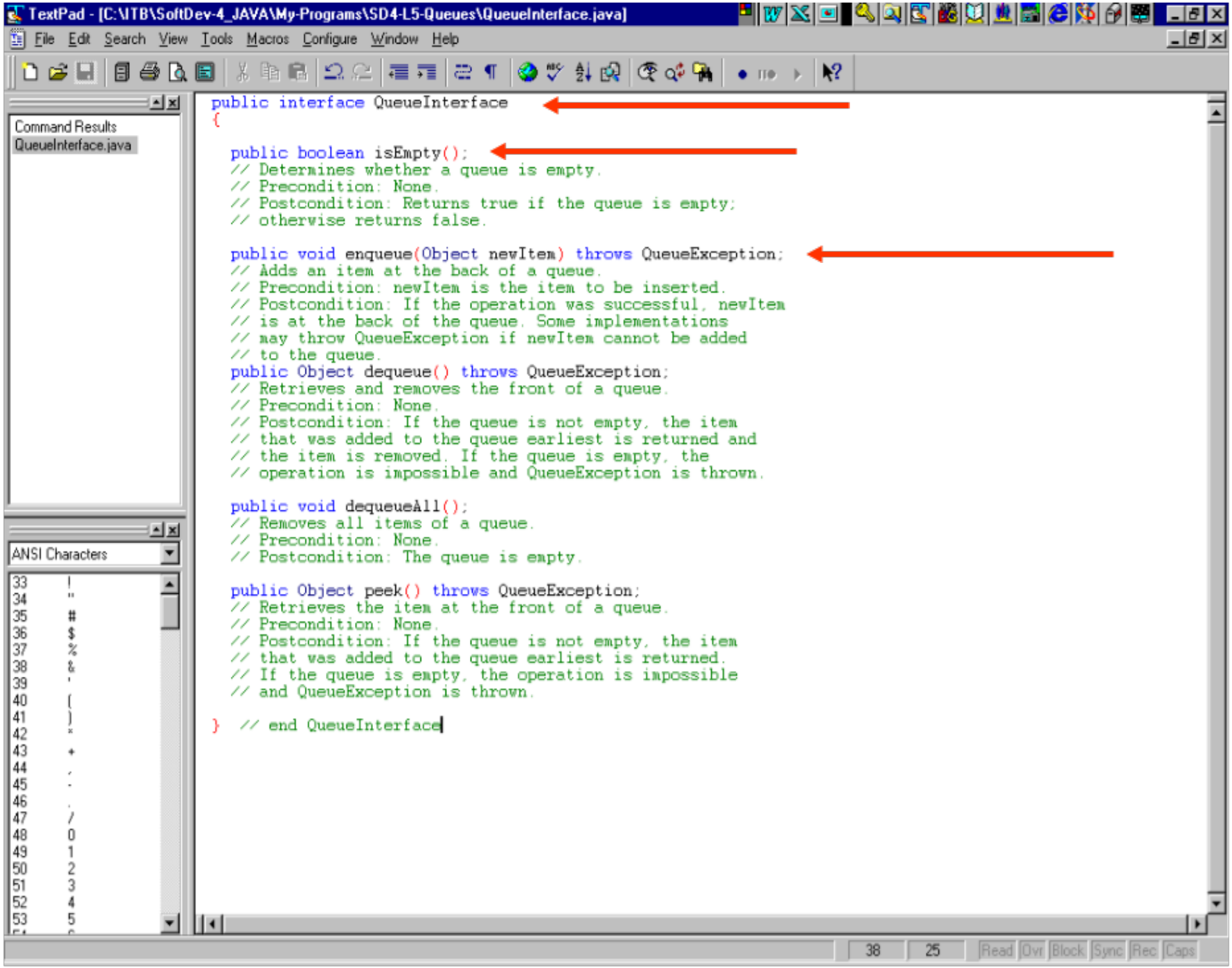
- The diagram shows the removal of the front item of a queue that contains more than one item.
- Notice that we need to change only one reference within the queue.
- Deletion from a queue with only one node is a special case.



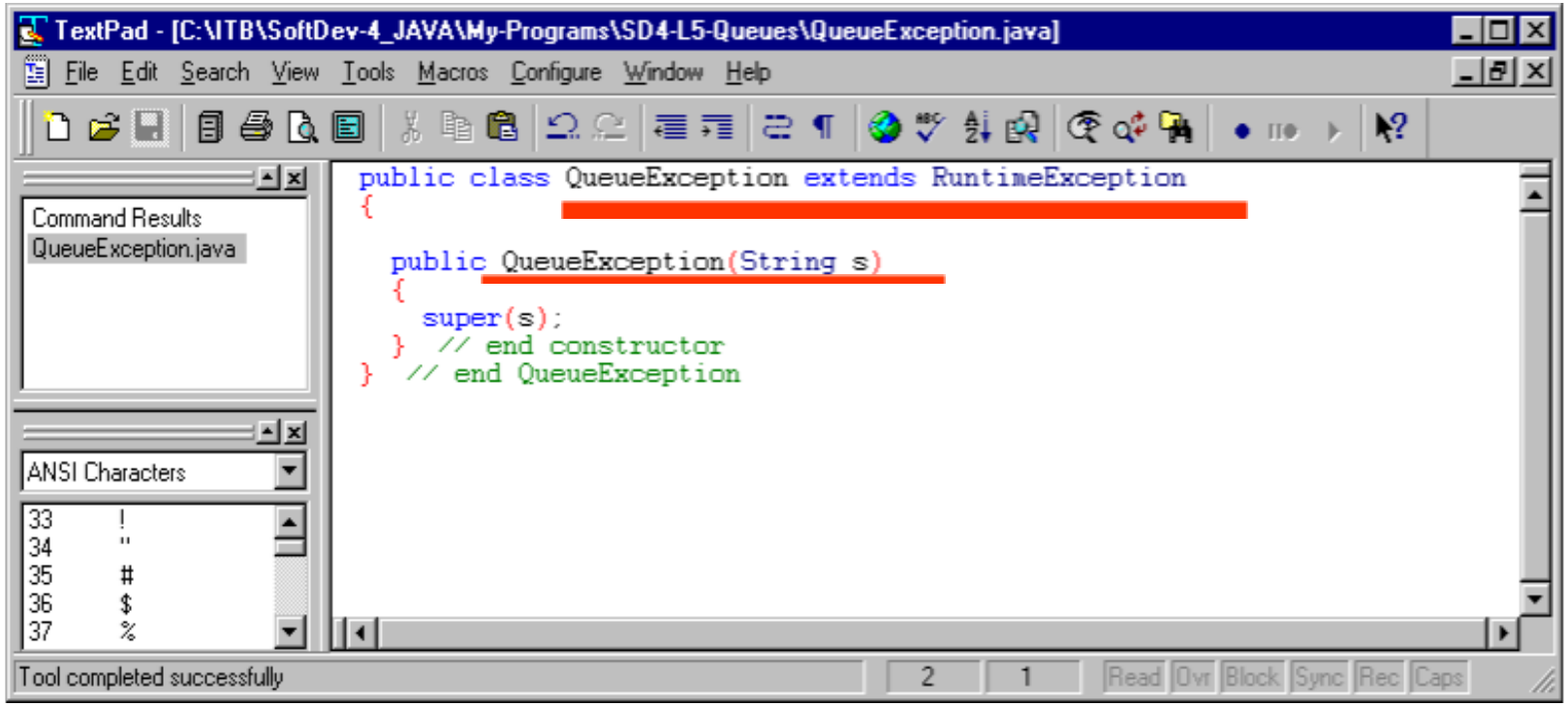
Deleting an item from a queue of more than one item

The following is the **interface specification for queues**

Note that **enqueue**, **dequeue** and **peek** may throw **QueueException**

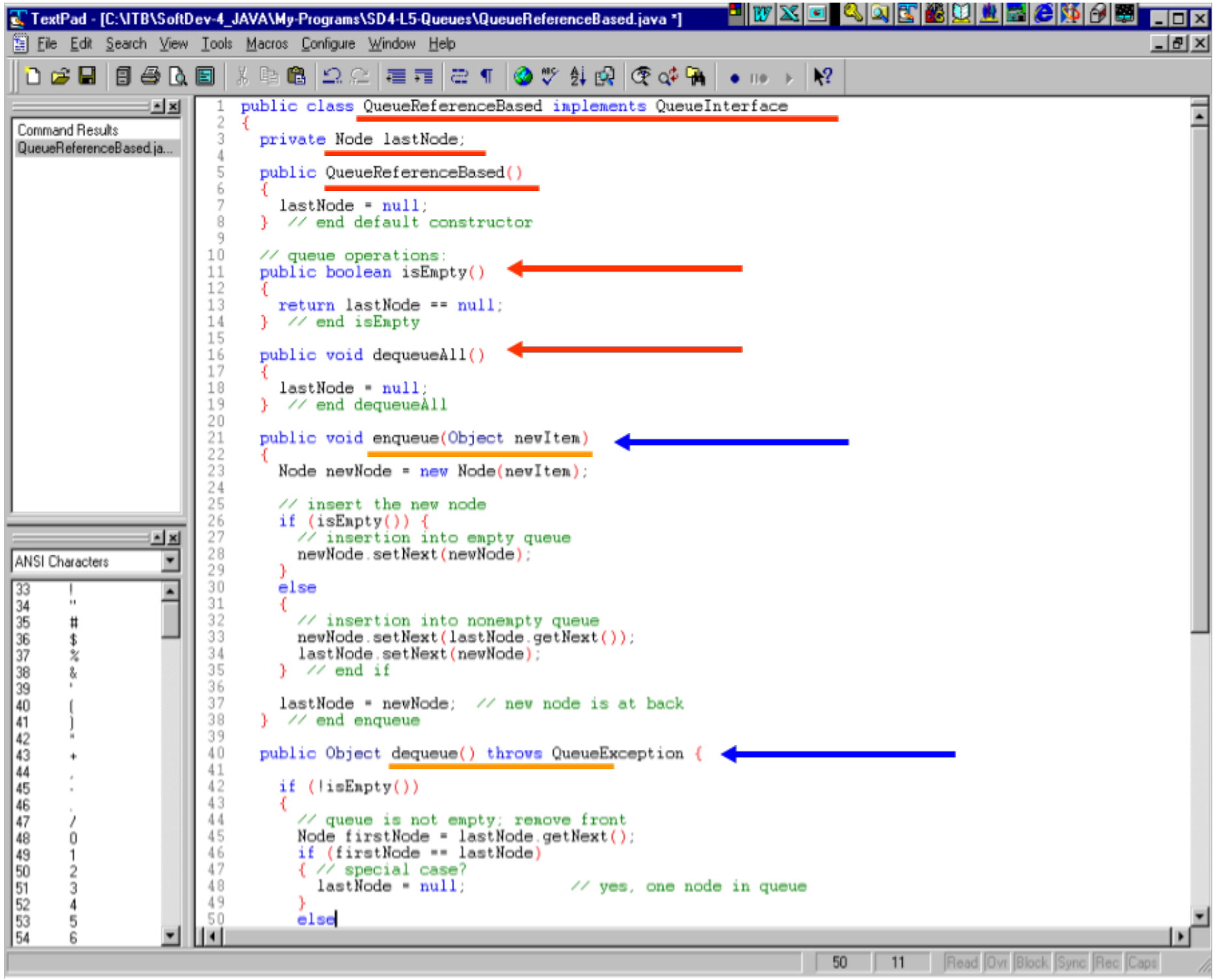

TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueInterface.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueInterface.java
ANSI Characters
public interface QueueInterface
{
public boolean isEmpty();
// Determines whether a queue is empty.
// Precondition: None.
// Postcondition: Returns true if the queue is empty;
// otherwise returns false.
public void enqueue(Object newItem) throws QueueException;
// Adds an item at the back of a queue.
// Precondition: newItem is the item to be inserted.
// Postcondition: If the operation was successful, newItem
// is at the back of the queue. Some implementations
// may throw QueueException if newItem cannot be added
// to the queue.
public Object dequeue() throws QueueException;
// Retrieves and removes the front of a queue.
// Precondition: None.
// Postcondition: If the queue is not empty, the item
// that was added to the queue earliest is returned and
// the item is removed. If the queue is empty, the
// operation is impossible and QueueException is thrown.
public void dequeueAll();
// Removes all items of a queue.
// Precondition: None.
// Postcondition: The queue is empty.
public Object peek() throws QueueException;
// Retrieves the item at the front of a queue.
// Precondition: None.
// Postcondition: If the queue is not empty, the item
// that was added to the queue earliest is returned.
// If the queue is empty, the operation is impossible
// and QueueException is thrown.
} // end QueueInterface
38 25 Read Ovr Block Sync Rec Caps

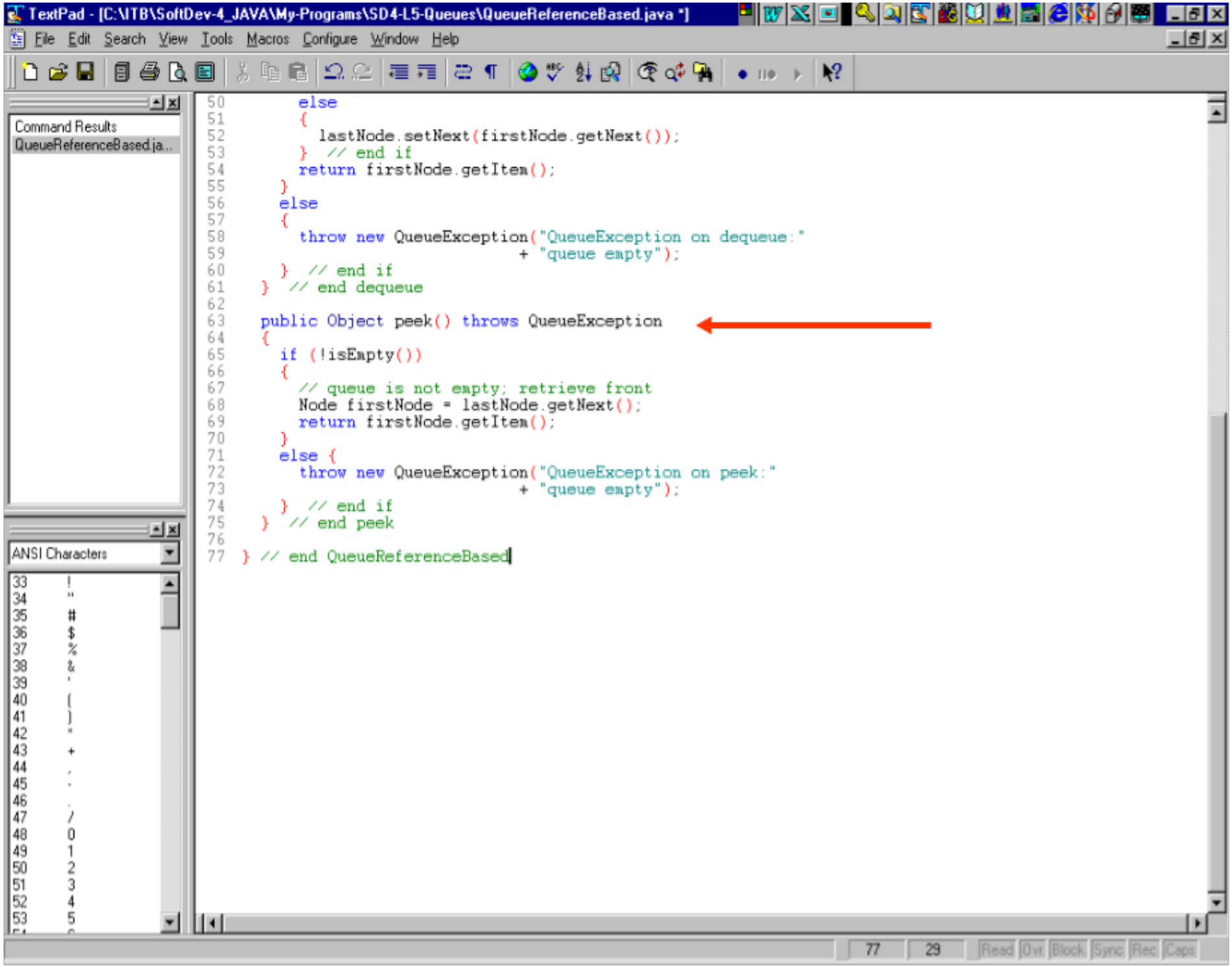
The **QueueException** class appears next.



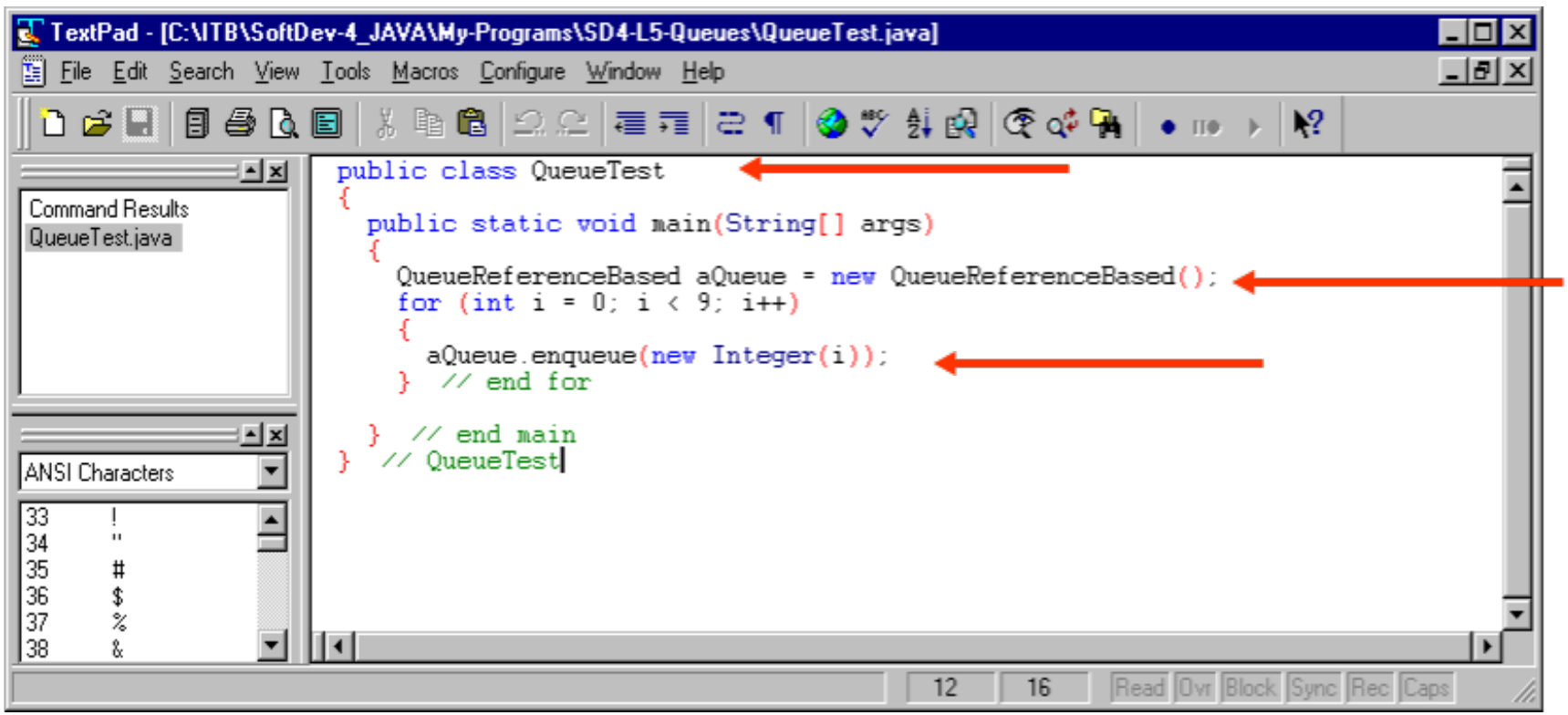
The following class is a reference-based implementation of the ADT queue

The implementation uses the **Node** class developed in an earlier lecture.


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueReferenceBased.java *]
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueReferenceBased.ja...
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
49 1
50 2
51 3
52 4
53 5
54 6
1 public class QueueReferenceBased implements QueueInterface
2 {
3 private Node lastNode;
4
5 public QueueReferenceBased()
6 {
7 lastNode = null;
8 } // end default constructor
9
10 // queue operations:
11 public boolean isEmpty()
12 {
13 return lastNode == null;
14 } // end isEmpty
15
16 public void dequeueAll()
17 {
18 lastNode = null;
19 } // end dequeueAll
20
21 public void enqueue(Object newItem)
22 {
23 Node newNode = new Node(newItem);
24
25 // insert the new node
26 if (isEmpty()) {
27 // insertion into empty queue
28 newNode.setNext(newNode);
29 }
30 else
31 {
32 // insertion into nonempty queue
33 newNode.setNext(lastNode.getNext());
34 lastNode.setNext(newNode);
35 } // end if
36
37 lastNode = newNode; // new node is at back
38 } // end enqueue
39
40 public Object dequeue() throws QueueException {
41
42 if (!isEmpty())
43 {
44 // queue is not empty; remove front
45 Node firstNode = lastNode.getNext();
46 if (firstNode == lastNode)
47 { // special case?
48 lastNode = null; // yes, one node in queue
49 }
50 else
50 11
Read Ovr Block Sync Rec Caps

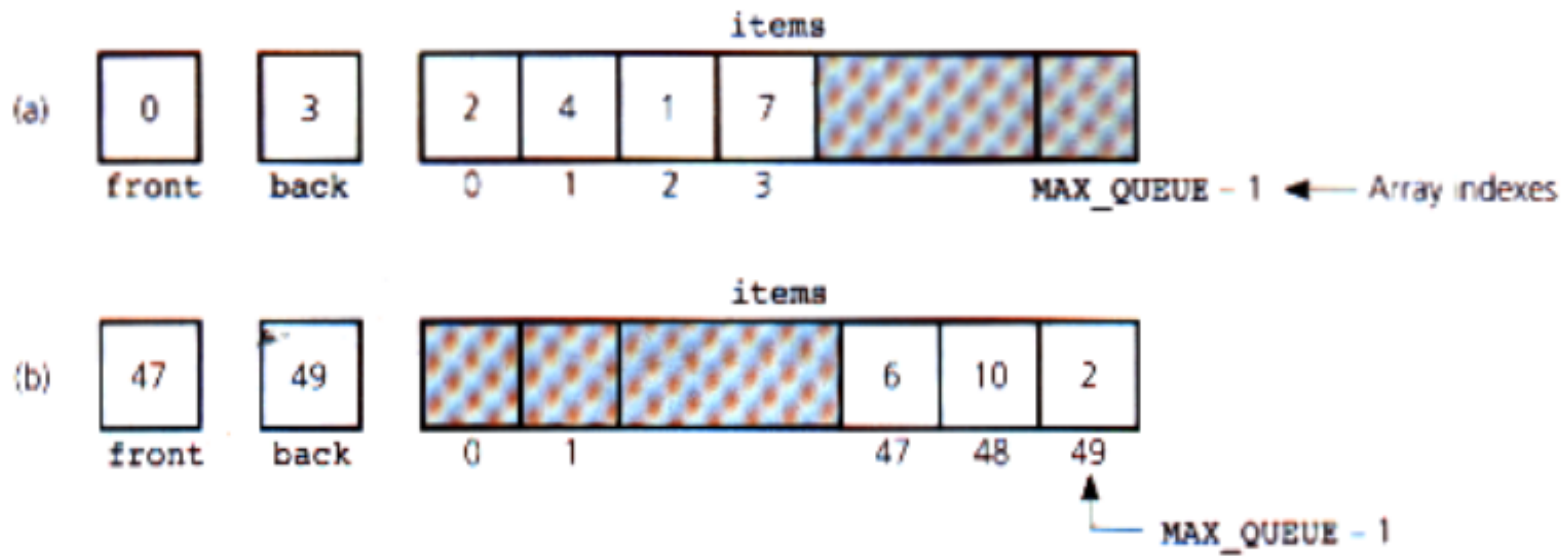

TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueReferenceBased.java *]
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueReferenceBased.ja...
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
49 1
50 2
51 3
52 4
53 5
50 else
51 {
52 lastNode.setNext(firstNode.getNext());
53 } // end if
54 return firstNode.getItem();
55 }
56 else
57 {
58 throw new QueueException("QueueException on dequeue:"
59 + "queue empty");
60 } // end if
61 } // end dequeue
62
63 public Object peek() throws QueueException
64 {
65 if (!isEmpty())
66 {
67 // queue is not empty; retrieve front
68 Node firstNode = lastNode.getNext();
69 return firstNode.getItem();
70 }
71 else {
72 throw new QueueException("QueueException on peek:"
73 + "queue empty");
74 } // end if
75 } // end peek
76
77 } // end QueueReferenceBased
77
29
Read Ovr Block Sync Rec Caps

A program that uses this implementation would look something like the following:



An Array-based Implementation

An array can be used to implement a queue if a **fixed-sized** does not present a problem within the solution.

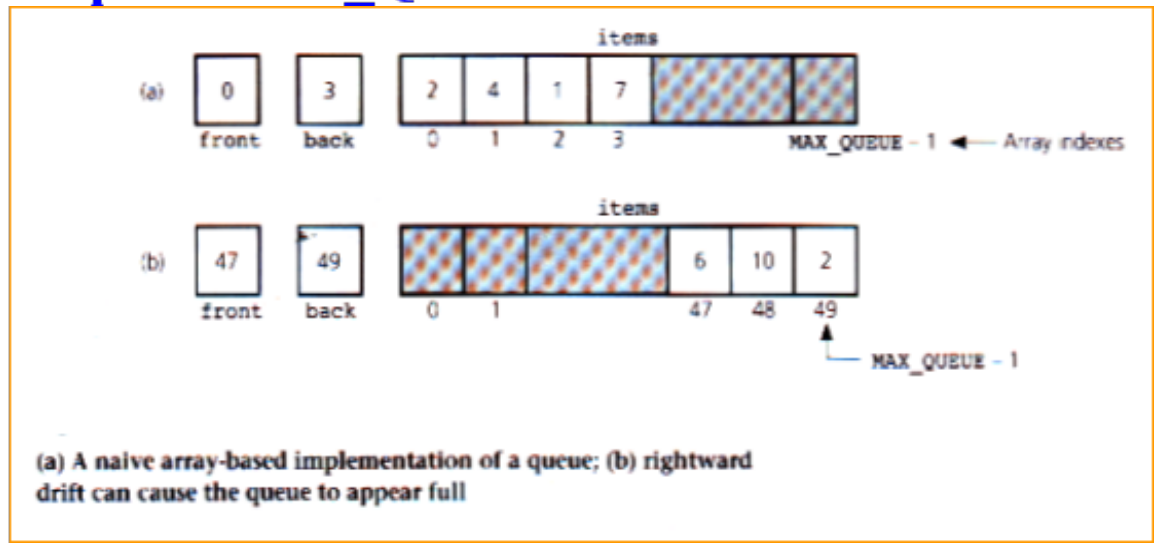


(a) A naive array-based implementation of a queue; (b) rightward drift can cause the queue to appear full

The **indexes of the front and back items**in the queue are **front** and **back**, respectively.

Initially **front** is 0 and **back** is –1.

- To **insert** a new item into the queue, you increment back and place an item in **items[back].**
- To **delete** an item, you simply increment **front**.
- The queue is **empty** whenever **back is less thanfront**.
- The queue is **full** when **back equals MAX_QUEUE – 1**

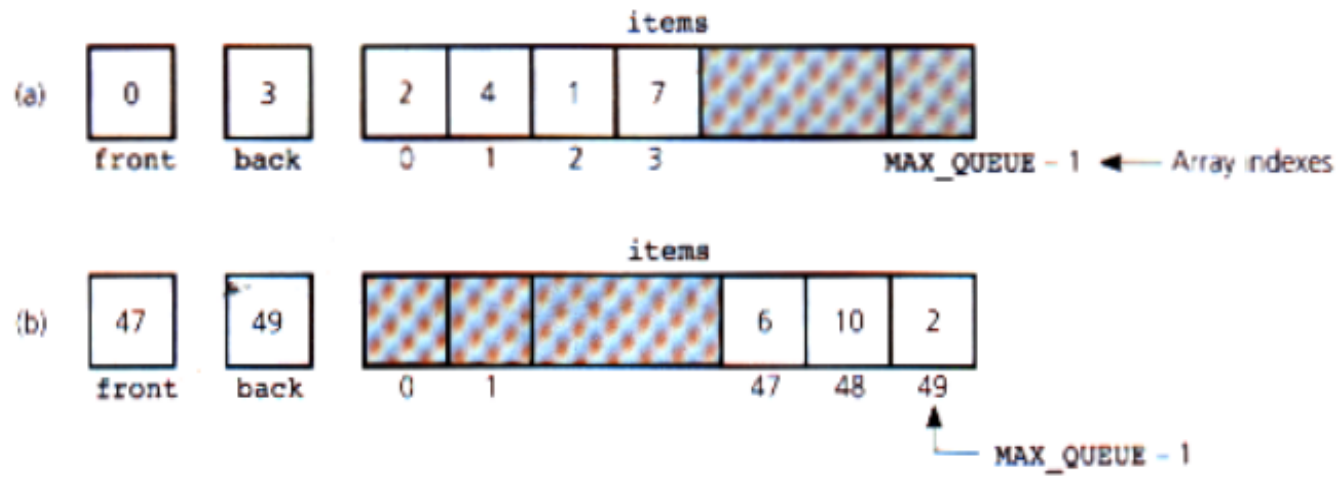


(a) A naive array-based implementation of a queue; (b) rightward drift can cause the queue to appear full

The **problem** with this strategy is **rightward shift**.

After a **sequence of additions and removals** …

- the items in the queue will drift towards the end of the array, and
- **back could equal MAX_QUEUE-1** even when the queue contains only a few items.



(a) A naive array-based implementation of a queue; (b) rightward drift can cause the queue to appear full

A possible solution is to shift array elements to the left, …

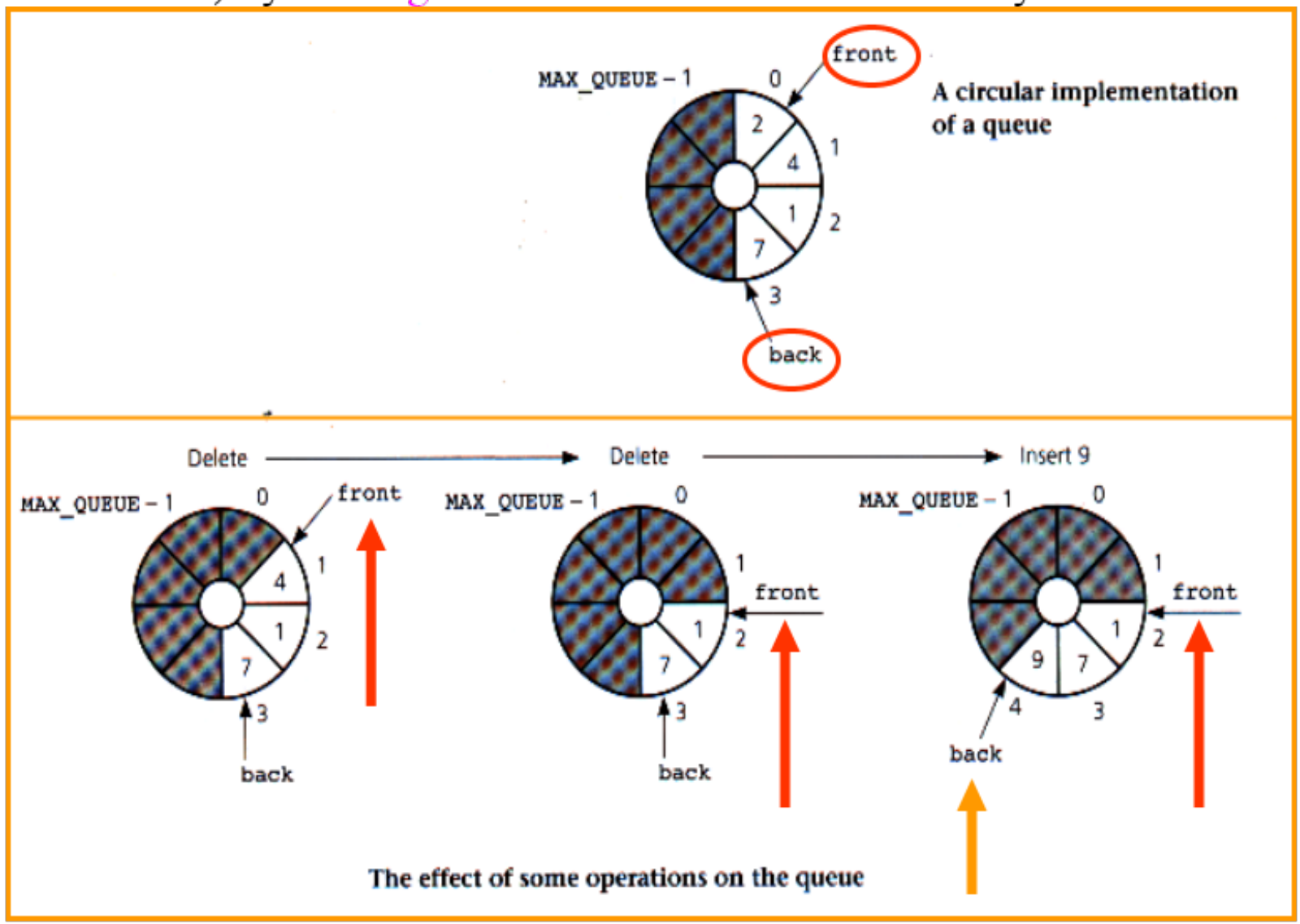
- either **after each deletion** or
- **whenever back equals MAX_QUEUE-1**.

This will guarantee that the queue can always contain up to MAX_QUEUE elements.

Shifting is not very satisfactory, a s it would dominate the **performance cost** of the implementation

A more elegant solution is possible by viewing the array as circular.

You advance the queue indexes **front** (to delete an item) and **back** (to insert an item) by moving them clockwise around the array.

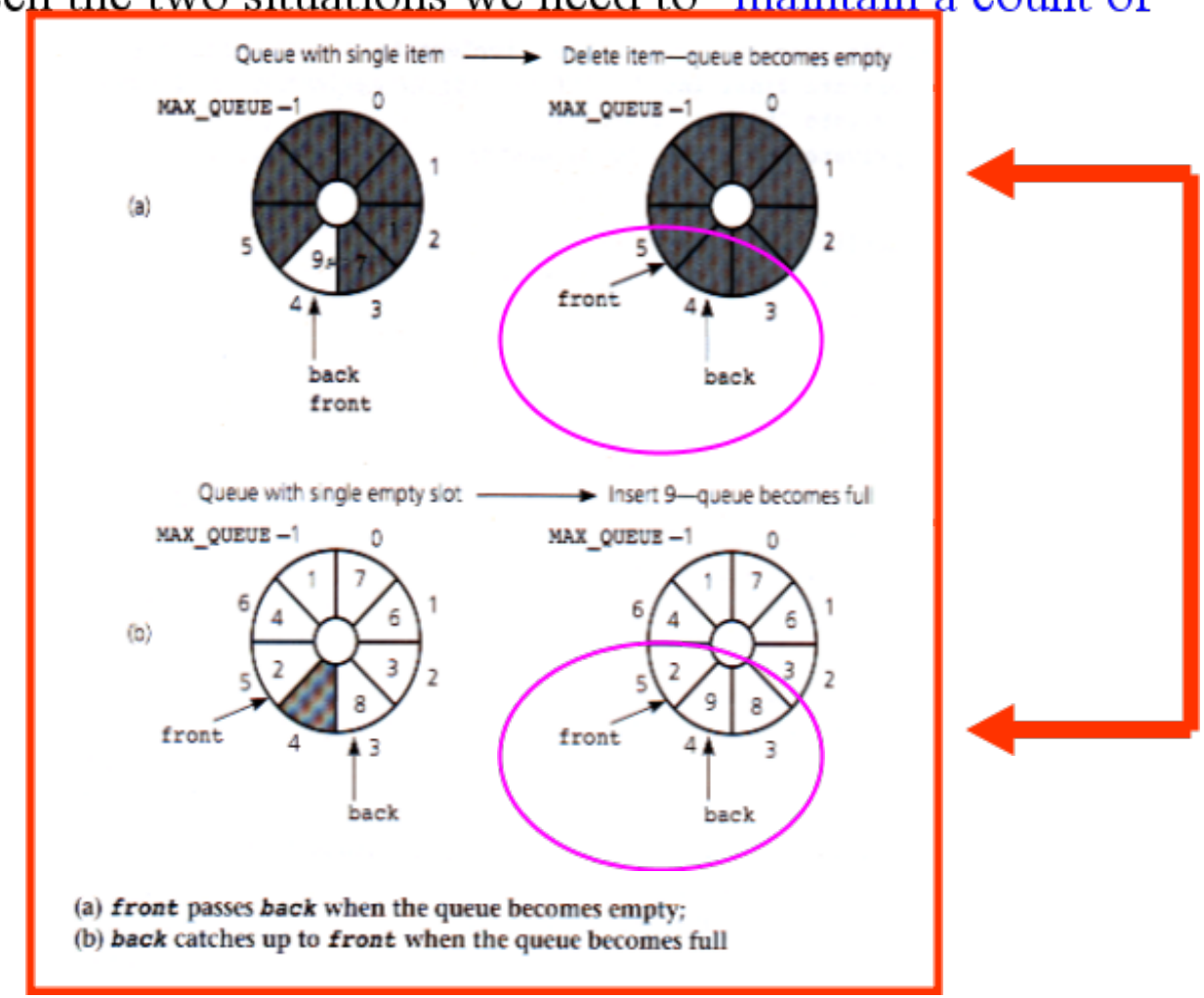


The only difficulty with this involves detecting the queue-empty and queue-full conditions.

The **queue-empty** condition:

- **front is one slot ahead of back** could also indicate a **full** queue.

To distinguish between the two situations we need to maintain a count of items in the queue.



(a) ***front*** passes ***back*** when the queue becomes empty;
(b) ***back*** catches up to ***front*** when the queue becomes full

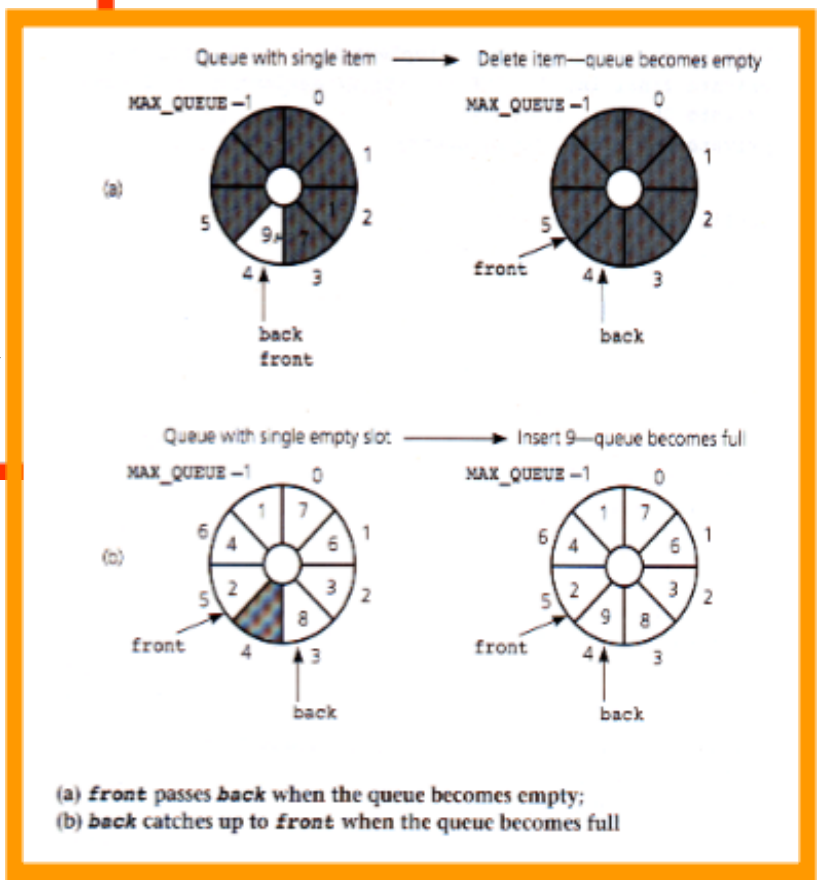
Enforce the wraparound effect in a circular queue by using modulo arithmetic.

```
//Inserting into a queue
back = (back + 1) % MAX_QUEUE;

items[back] = newItem;

++count;


//Deleting from the front of a queue

front = (front + 1) % MAX_QUEUE;

--count;
```

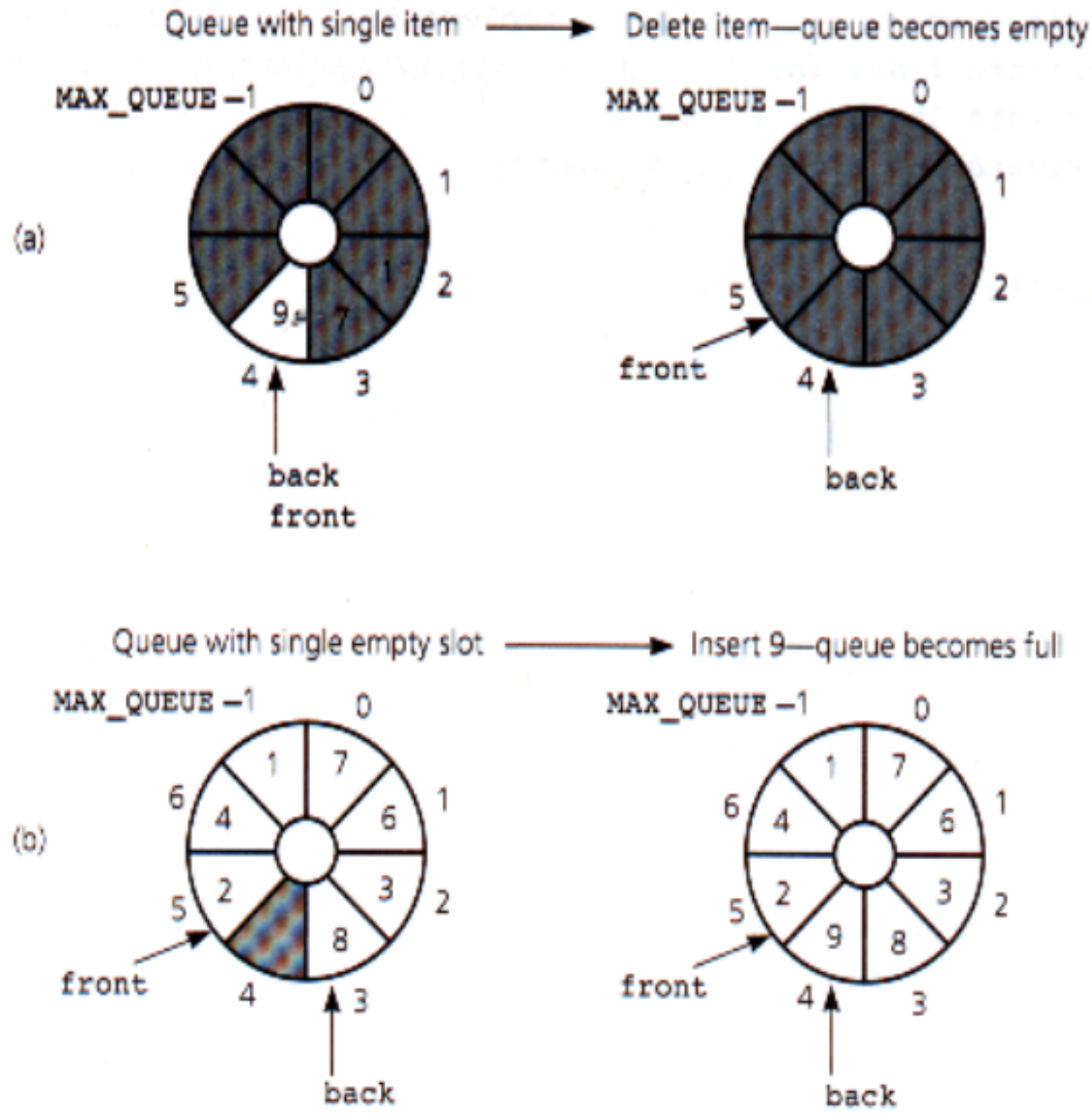
Etc.

See:

- **enque method**
- **dequeue method**

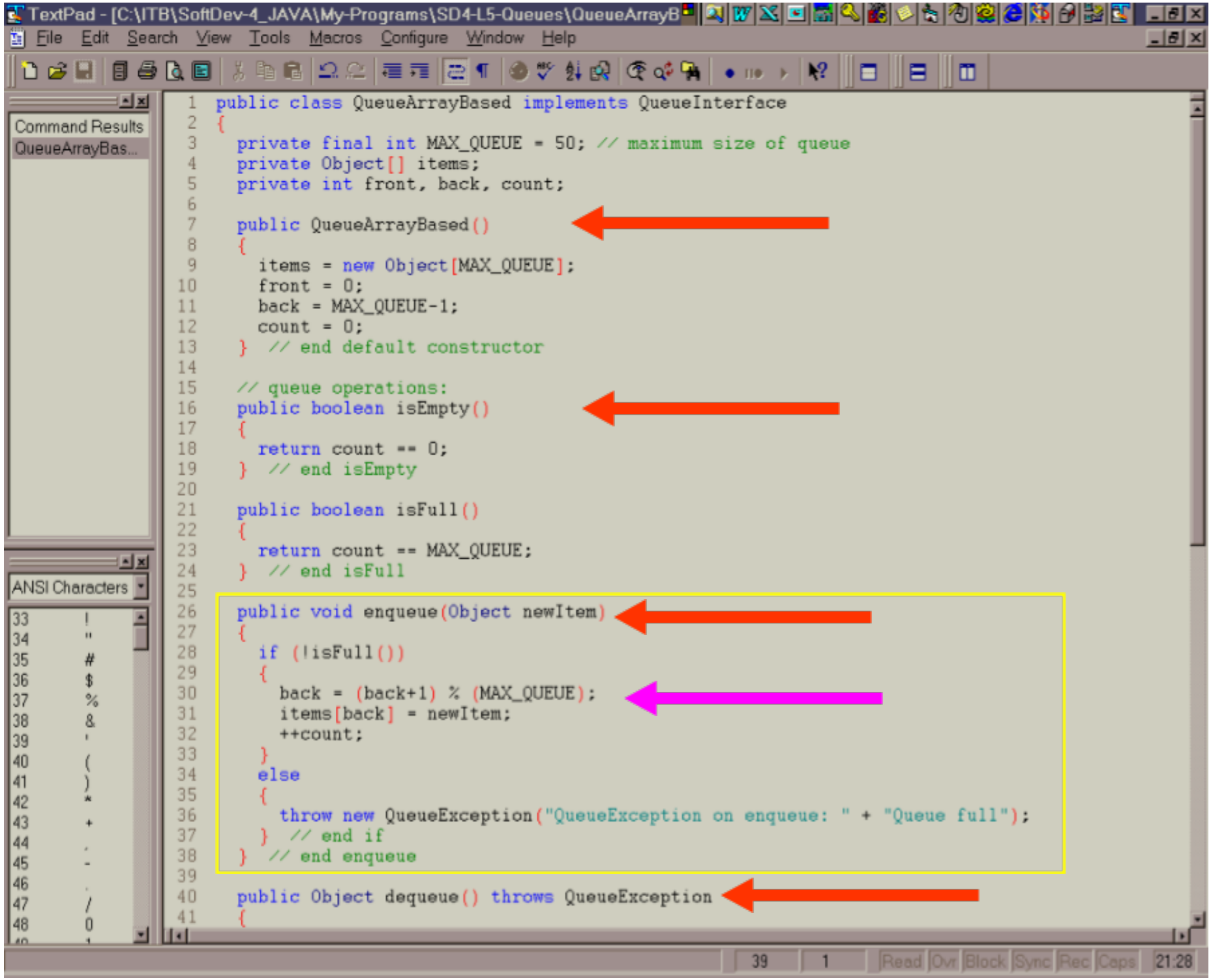


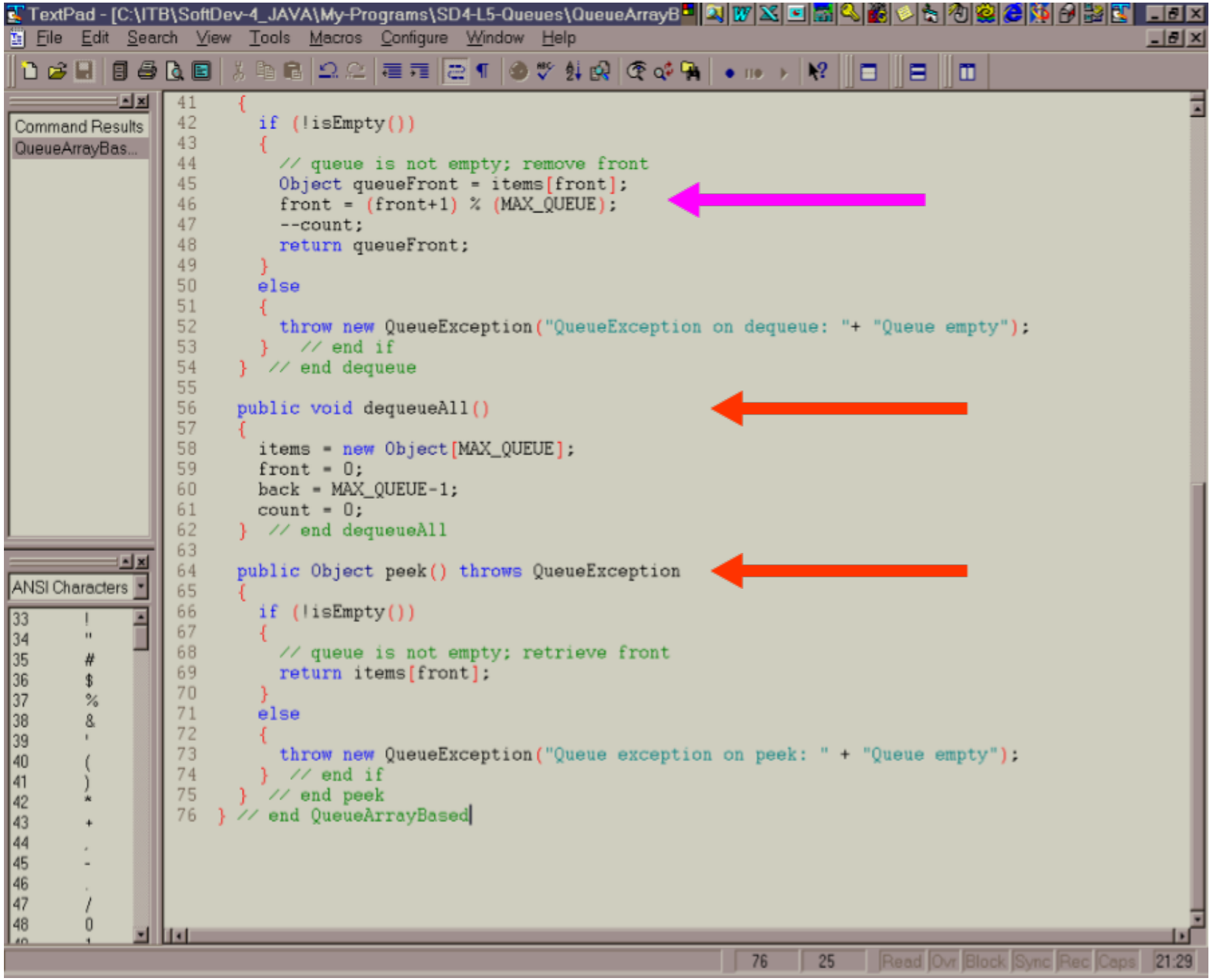
(a) **front** passes **back** when the queue becomes empty;
(b) **back** catches up to **front** when the queue becomes full



(a) ***front*** passes ***back*** when the queue becomes empty;
(b) ***back*** catches up to ***front*** when the queue becomes full

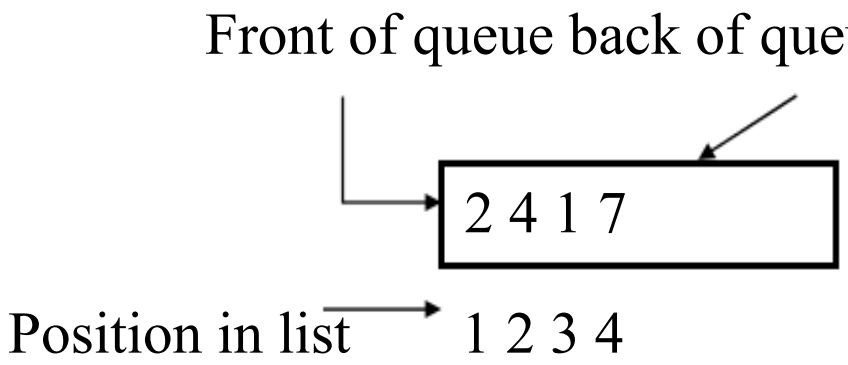
The following class is an array-based implementation of the ADT queue that uses a circular array.


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueArrayB
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueArrayBas...
ANSI Characters
public class QueueArrayBased implements QueueInterface
{
private final int MAX_QUEUE = 50; // maximum size of queue
private Object[] items;
private int front, back, count;
public QueueArrayBased()
{
items = new Object[MAX_QUEUE];
front = 0;
back = MAX_QUEUE-1;
count = 0;
} // end default constructor
// queue operations:
public boolean isEmpty()
{
return count == 0;
} // end isEmpty
public boolean isFull()
{
return count == MAX_QUEUE;
} // end isFull
public void enqueue(Object newItem)
{
if (!isFull())
{
back = (back+1) % (MAX_QUEUE);
items[back] = newItem;
++count;
}
else
{
throw new QueueException("QueueException on enqueue: " + "Queue full");
} // end if
} // end enqueue
public Object dequeue() throws QueueException
{
39
1
Read Ovr Block Sync Rec Caps
21:28


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueArrayB
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueArrayBas...
ANSI Characters
41 {
42 if (!isEmpty())
43 {
44 // queue is not empty; remove front
45 Object queueFront = items[front];
46 front = (front+1) % (MAX_QUEUE);
47 --count;
48 return queueFront;
49 }
50 else
51 {
52 throw new QueueException("QueueException on dequeue: "+ "Queue empty");
53 } // end if
54 } // end dequeue
55
56 public void dequeueAll()
57 {
58 items = new Object[MAX_QUEUE];
59 front = 0;
60 back = MAX_QUEUE-1;
61 count = 0;
62 } // end dequeueAll
63
64 public Object peek() throws QueueException
65 {
66 if (!isEmpty())
67 {
68 // queue is not empty; retrieve front
69 return items[front];
70 }
71 else
72 {
73 throw new QueueException("Queue exception on peek: " + "Queue empty");
74 } // end if
75 } // end peek
76 } // end QueueArrayBased
76 25 Read Ovr Block Sync Rec Caps 21:29

An Implementation that uses the ADT List

We can use the ADT List to represent a queue



An Implementation that uses the ADT List

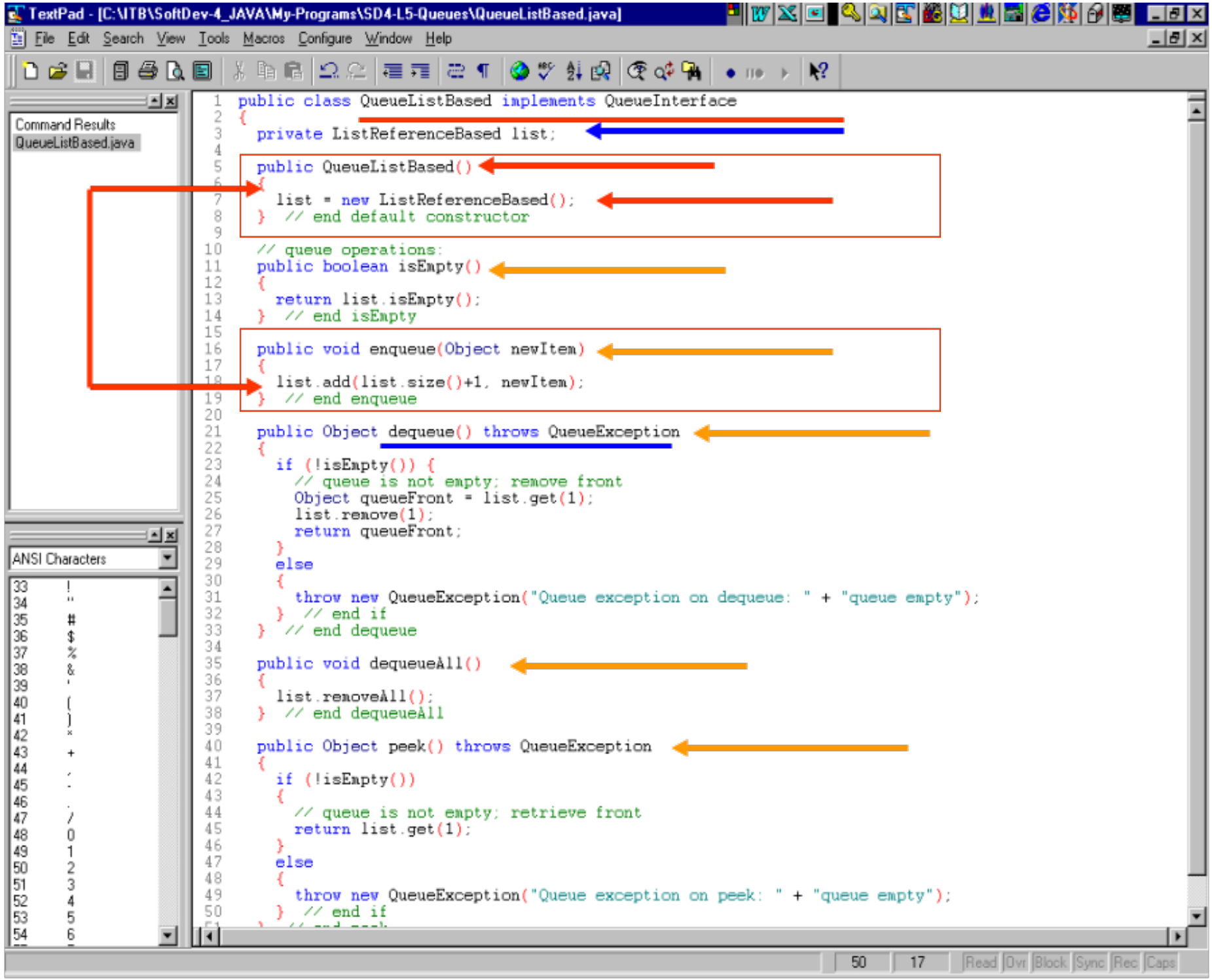
If the **item in position 1** of a list **aList** represents the **front** of the queue…

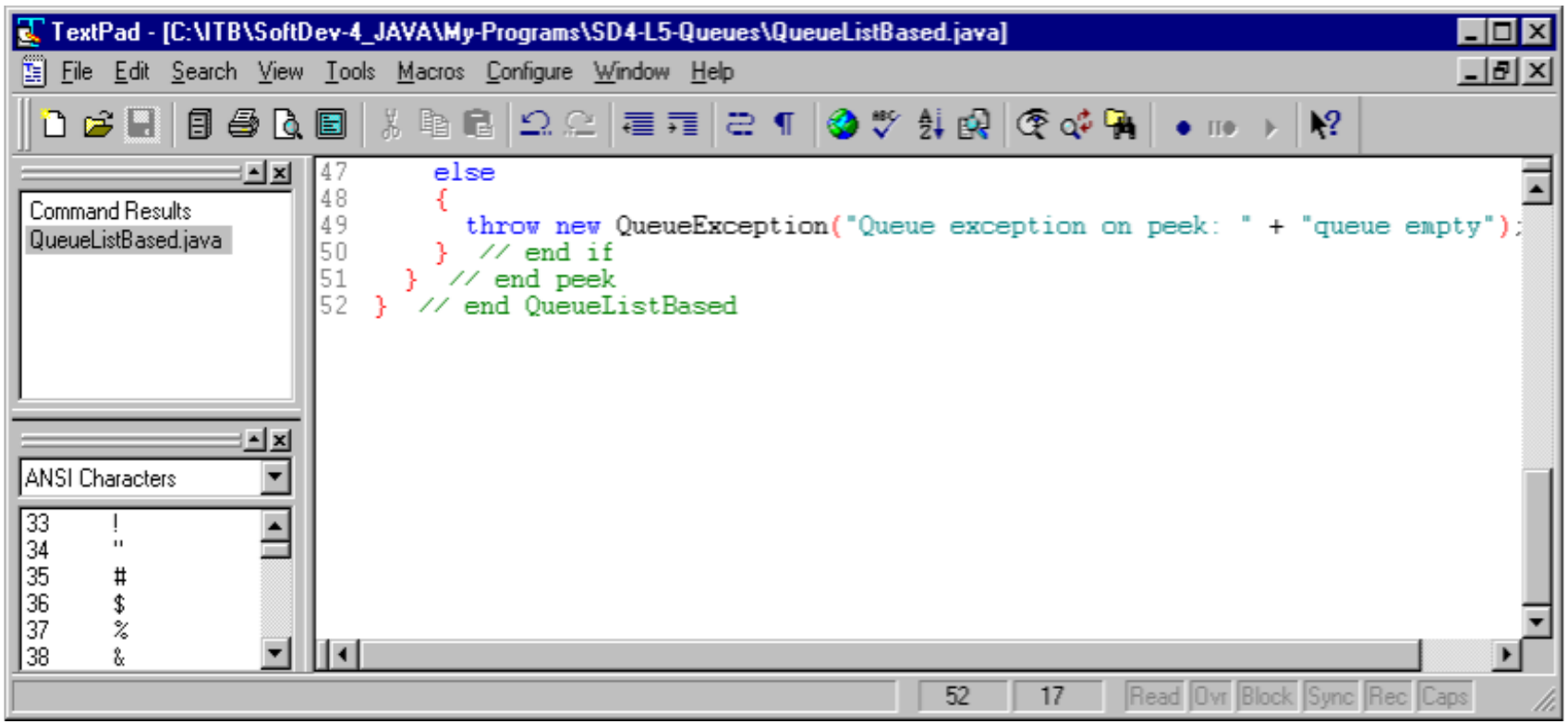
- We can implement the operation **dequeue()** as **aList.remove**(1)
- The operation peek() as **aList.get**(1)

If we **let the item at the end** of the list represent the **back** of the queue,

- We can implement the operation **enqueue(newItem)** as **aList.add**(**aList.size**() + 1, newItem)

The following class uses an instance of **ListReferenceBased** to represent the **queue**


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueListBased.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueListBased.java
ANSI Characters
public class QueueListBased implements QueueInterface
{
private ListReferenceBased list;
public QueueListBased()
{
list = new ListReferenceBased();
} // end default constructor
// queue operations:
public boolean isEmpty()
{
return list.isEmpty();
} // end isEmpty
public void enqueue(Object newItem)
{
list.add(list.size()+1, newItem);
} // end enqueue
public Object dequeue() throws QueueException
{
if (!isEmpty()) {
// queue is not empty; remove front
Object queueFront = list.get(1);
list.remove(1);
return queueFront;
}
else
{
throw new QueueException("Queue exception on dequeue: " + "queue empty");
} // end if
} // end dequeue
public void dequeueAll()
{
list.removeAll();
} // end dequeueAll
public Object peek() throws QueueException
{
if (!isEmpty())
{
// queue is not empty; retrieve front
return list.get(1);
}
else
{
throw new QueueException("Queue exception on peek: " + "queue empty");
} // end if
50
17
Read Ovr Block Sync Rec Caps


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L5-Queues\QueueListBased.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
QueueListBased.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
47 else
48 {
49 throw new QueueException("Queue exception on peek: " + "queue empty");
50 } // end if
51 } // end peek
52 } // end QueueListBased
52
17
Read Ovr Block Sync Rec Caps

Comparing Implementations

Implementations of the ADT Queue may use either a linear linked list, a circular linked list, an array, a circular array, or the ADT list to represent the items in a queue.

- We have seen the details of three of these implementations.
- All of the implementations of the ADT queue are either array-based or referenced-based.
- The choice between an array-based or reference-based is primarily about fixed vs. dynamic size.

A Summary of position-oriented ADTs

We have seen **position-oriented** ADTs

- List
- Stack
- Queue

Each has a common theme:

- All of the operations are defined in terms of the positions of their data items.
- Stacks and queues greatly restrict the positions that their operations can affect - Only their end positions can be accessed.
- The list removes this restriction.

Stacks are very similar to **queues**:

- **createStack and createQueue**. These operations create an empty ADT of the appropriate type.
- **Stack isEmpty and queue isEmpty**. These operations determine whether any items exist in the ADT.
- **Operations push and enqueue** These operations insert a new item into one end (the top and back, respectively) of the ADT.
- **Operations pop and dequeue** . The pop operation deletes the most recent item, which is at the top of the stack, and the dequeue deletes the first item, which is at the front of the queue.
- **Stack peek and queue peek**. Stack peek retrieves the most recent item, which is at the top of the stack, and queue peek retrieves the first item, which is at the front of the queue.

The **ADT List** allows you to insert into, delete from, and inspect the item at any position of the list.

The **ADT List** has the most flexible operations.

Axioms of the ADT Queue

1 `(queue.createQueue()).isEmpty() = true`

2 `(queue.enqueue(item)).isEmpty() = false`

3 `(queue.createQueue()).dequeue() = error`

4 `((queue.createQueue()).enqueue(item)).dequeue() = queue.createQueue()`

5 `queue.isEmpty() = false ⇒ (queue.enqueue(item)).dequeue() = (queue.dequeue()).enqueue(item)`

6 `(queue.createQueue()).peek() = error`

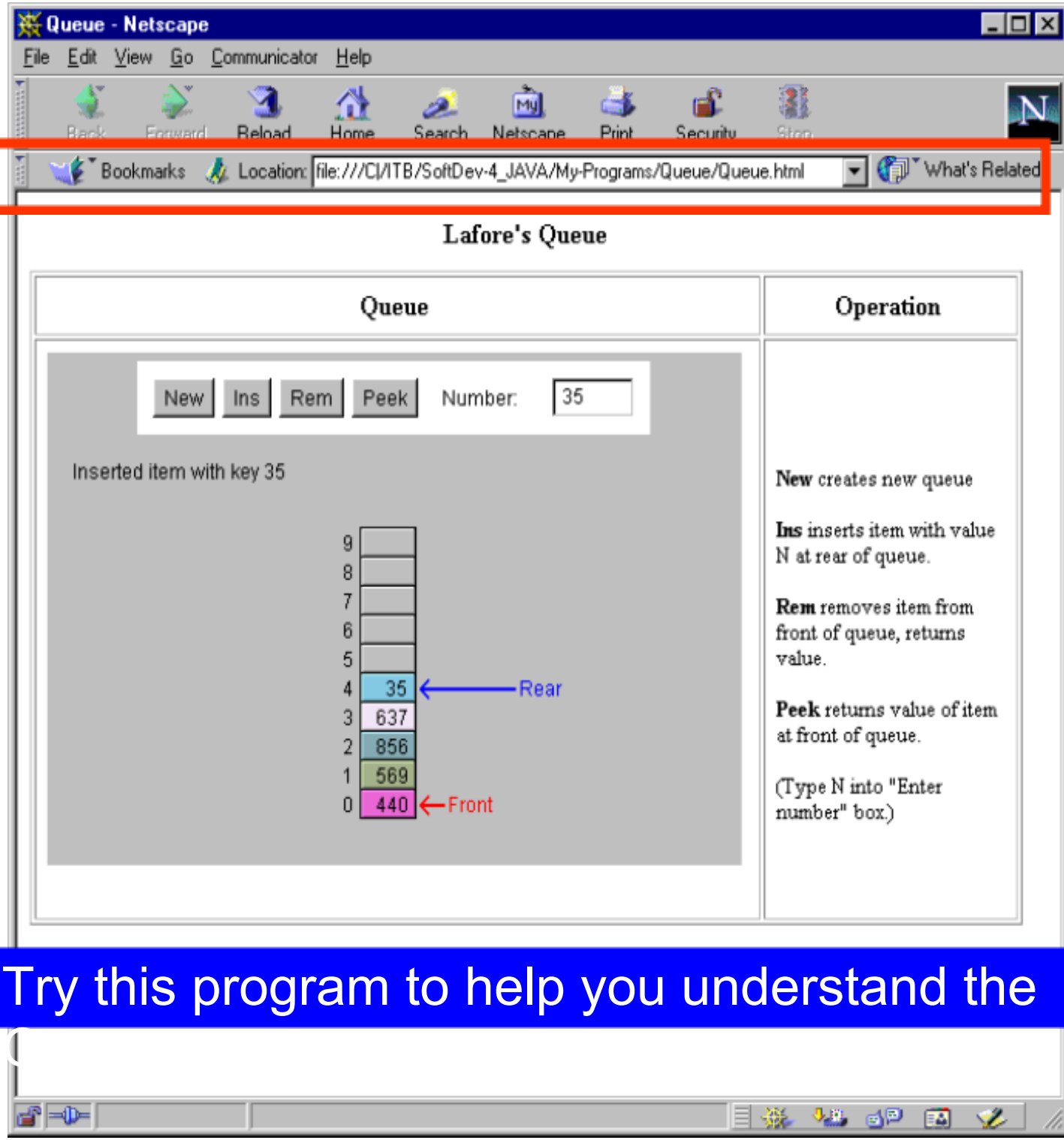
7 `((queue.createQueue()).enqueue(item)).peek() = item`

8 `queue.isEmpty() = false ⇒ (queue.enqueue(item)).peek() = queue.peek()`

Programs to do this week:

1 Create an **ADT Queue** implemented over a Circular Array

- Implement the ADT Queue as a Java class over an Circular Array.
- This class must implement the ADT operations as public methods of the class
- Explain the **enqueue**, **dequeue**, **peek** methods with specific referenc e to the contract and the Java code. Use **diagrams** to support your answer
- Write a small driver program that demonstrates the ADT Queue.
- This program should print results to the screen in some simple manner. Ideally this should be done via a GUI display us ing a graphical metaphor of the queue!
- Exercise all of the methods to prove that t he data is processed correctly.
- Write a method to display the queue contents (perhaps this should return a formatted string).
- Lastly write a method boolean isPal(String s), that uses a stack and a queue to determine is the string 's' is a palindrome.



Try this program to help you understand the