CSC115 Lecture 12

Chapters 7 & 8

Stacks & Queues

- Concepts
- ADT
- Implementation ideas

Next Class: implementation details

New Concept: Stack

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria

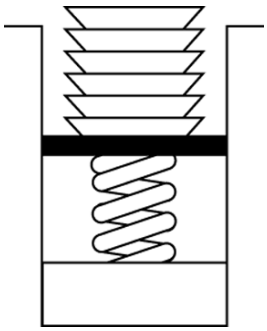


Figure 7-1

Stack of cafeteria dishes

New Concept: Queue

- A queue
  - First in, first out (FIFO) property
    - The first item added is the first item to be removed

Recall: Abstract Data Types

(from Lecture 4)

- An ADT is composed of
  - A collection of data
  - A set of operations on that data
- Specifications of an ADT indicate
  - What the ADT operations do,
    NOT: how to implement them
- Implementation of an ADT
  - Includes choosing a particular data structure

Recall: Abstract Data Types

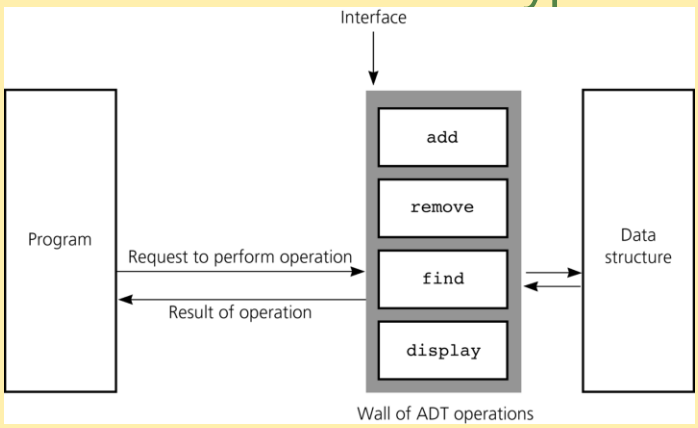


Figure 4-4

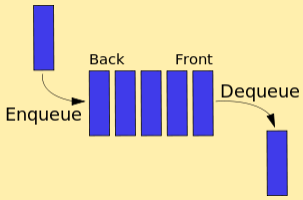
A wall of ADT operatio

So . . . .
Lets consider ADTs for Stack & Queue

ADT Stack

- ADT stack operations
  - Determine whether a stack is empty
  - Add a new item to the stack
  - Remove from the stack the item that was added most recently
  - Remove all the items from the stack
  - Retrieve from the stack the item that was added most recently

The Abstract Data Type Queue



Graphic from: wikipedia.org

- ADT queue operations
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove from the queue the item that was added earliest
  - Remove all the items from the queue
  - Retrieve from the queue the item that was added earliest

The ADT Stack ➔ Java's Interface

- Pseudocode for the ADT stack operations

```
isEmpty()
// Determines whether a stack is empty.

push(newItem) throws StackException
// Adds newItem to the top of the stack.
// Throws StackException if the insertion is
// not successful.
```

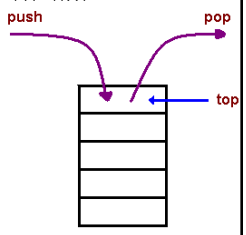
The ADT Stack ➔ Java's Interface

- Pseudocode for the ADT stack operations (Continued)

```
pop() throws StackException
// Retrieves and then removes the top of the stack.
// Throws StackException if the deletion is not
// successful.

popAll()
// Removes all items from the stack.

peek() throws StackException
// Retrieves the top of the stack. Throw
// StackException if the retrieval is not successful
```



The Abstract Data Type Queue

- Pseudocode for the ADT queue operations

```
isEmpty()
// Determines whether a queue is empty

enqueue(newItem) throws QueueException
// Adds newItem at the back of a queue. Throws
// QueueException if the operation is not
// successful
```

The Abstract Data Type Queue

- Pseudocode for the ADT queue operations (Continued)

```
dequeue() throws QueueException
// Retrieves and removes the front of a queue.
// Throws QueueException if the operation is
// not successful.

dequeueAll()
// Removes all items from a queue

peek() throws QueueException
// Retrieves the front of a queue. Throws
// QueueException if the retrieval is not
// successful
```


Example Application: the ADT Stack: - Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces

    `abc{defg{ijk}{l{mn}}op}qr`
  - An example of unbalanced braces

    `abc{def}}{ghij{kl}m`

Checking for Balanced Braces

- Requirements for balanced braces
  - Each time you encounter a "}", it matches an already encountered "{"
  - When you reach the end of the string, you have matched each "{"

Checking for Balanced Braces

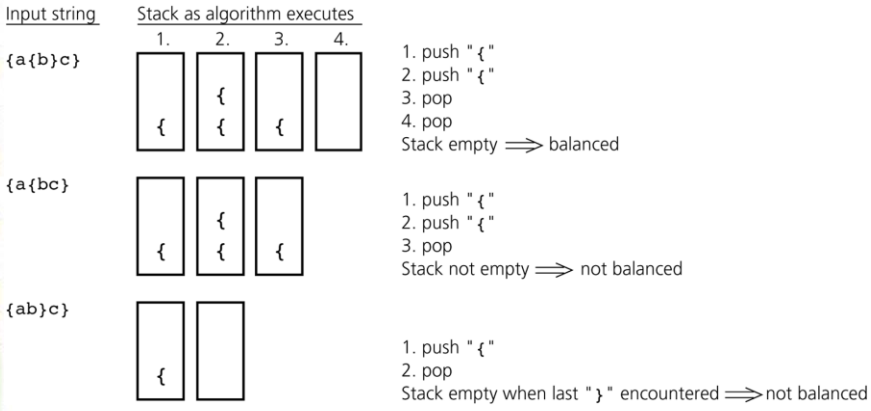


Figure 7-3

Traces of the algorithm that checks for balanced braces

Example: Balanced Braces: ie, javac

```
public class TestQuest4 {
      public static int theSum(int num) {

            if (num < 2) return 2*num;
           else  {
                  return theSum(num/2) + theSum(num-4);
            }
      }

      public static void main(String[] args) {
            int answer=theSum(6);
            System.out.println("The Sum = " + answer);
      }
}
```

Imagine: This brace is missing

Checking for Balanced Braces

- The exception `StackException`
  - A Java method that implements the balanced-braces algorithm should do one of the following
    - Take precautions to avoid an exception
    - Provide `try` and `catch` blocks to handle a possible exception

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list

Implementation of methods discussed in class:
- see Lecture 12 Code for result
- also, some implementation pictures next 2 slides

- `StackInterface`
  - Provides a common specification for the implementations
- `StackException`
  - Used by `StackInterface`
  - Extends `java.lang.RuntimeException`

Implementations of the ADT Stack

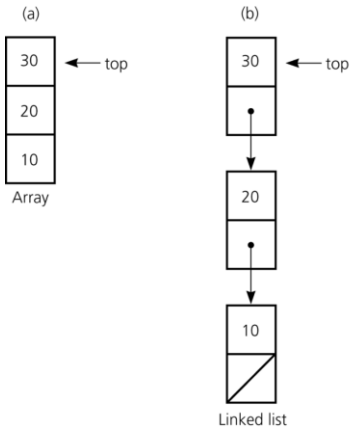


Figure 7-4
Implementation of the ADT stack that use a) an array; b) a linked list;

An Array-Based Implementation of the ADT Stack

- `StackArrayBased` class
  - Implements `StackInterface`
  - Instances
    - Stacks
  - Private data fields
    - An array of `Objects` called `items`
    - The index `top`

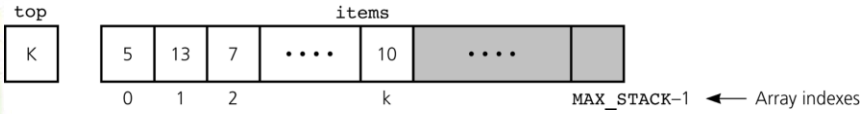


Figure 7-5

An array-based implementation

A Reference-Based Implementation of the ADT Stack

- A reference-based implementation
  - Required when the stack needs to grow and shrink dynamically
- `StackReferenceBased`
  - Implements `StackInterface`
  - `top` is a reference to the head of a linked list of items

A Reference-Based Implementation of the ADT Stack

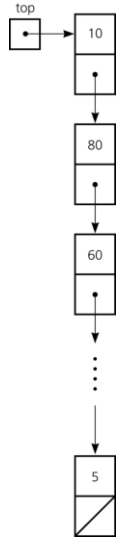


Figure 7-6

A reference-based implementation

The Abstract Data Type Queue

- Queues
  - Are appropriate for many real-world situations
    - Example: A line to buy a movie ticket
  - Have applications in computer science
    - Example: A request to print a document

The Abstract Data Type 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()`	5 2 7 (`queueFront` is 5)
`queueFront = queue.dequeue()`	5 2 7 (`queueFront` is 5)
`queueFront = queue.dequeue()`	2 7 (`queueFront` is 2)

Figure 8-2

Some queue operations

Simple Applications of the ADT Queue: Reading a String of Characters

- A queue can retain characters in the order in which they are typed

```
queue.createQueue()
while (not end of line) {
  Read a new character ch
  queue.enqueue(ch)
}
```

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

Recognizing Palindromes

- A palindrome
  - A string of characters that reads the same from left to right as its does from right to left
- To recognize a palindrome, a queue can be used in conjunction with a stack
  - A stack can be used to reverse the order of occurrences
  - A queue can be used to preserve the order of occurrences

Recognizing Palindromes

- A nonrecursive recognition algorithm for palindromes
  - As you traverse the character string from left to right, insert each character into both a queue and a stack
  - Compare the characters at the front of the queue and the top of the stack

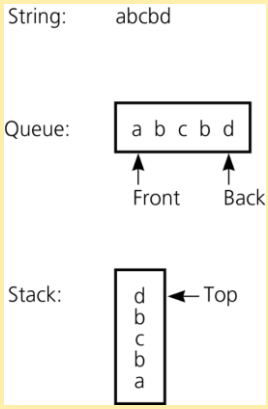


Figure 8-3

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

Implementations of the ADT Queue

- A queue can have either
  - An array-based implementation
  - A reference-based implementation

A Reference-Based Implementation

- Possible implementations of a queue
  - A linear linked list with two external references
    - A reference to the front
    - A reference to the back

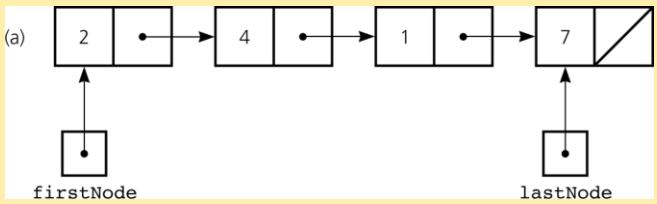


Figure 8-4a

A reference-based implementation of a queue: a) a linear linked list with two external references

A Reference-Based Implementation

- Possible implementations of a queue (Continued)
  - A circular linked list with one external reference
    - A reference to the back

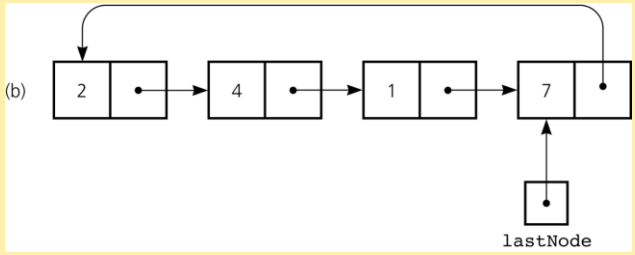


Figure 8-4b

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

A Reference-Based Implementation

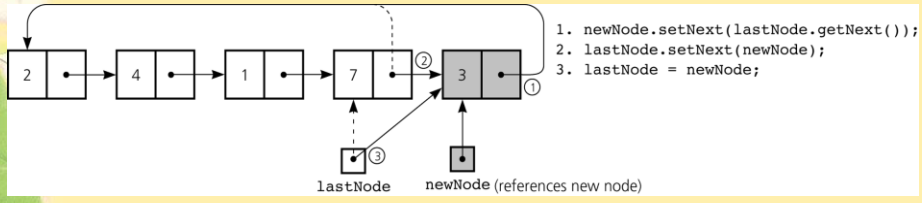


Figure 8-5

Inserting an item into a nonempty queue

A Reference-Based Implementation

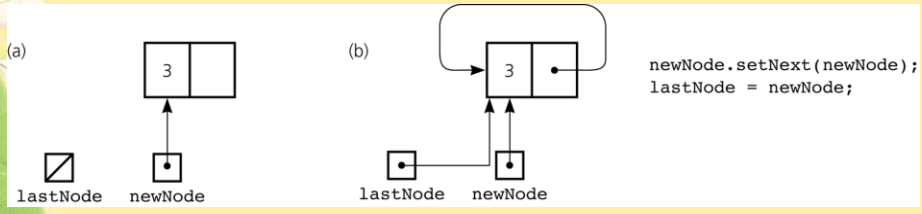


Figure 8-6

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

A Reference-Based Implementation

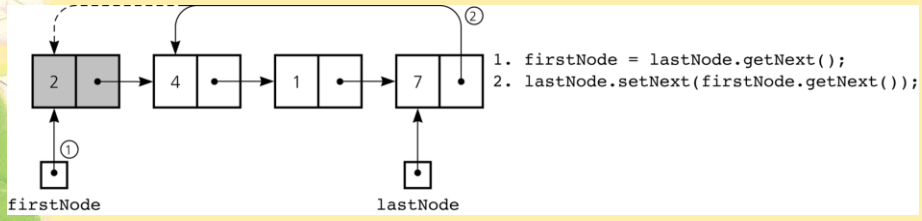


Figure 8-7

Deleting an item from a queue of more than one item

An Array-Based Implementation

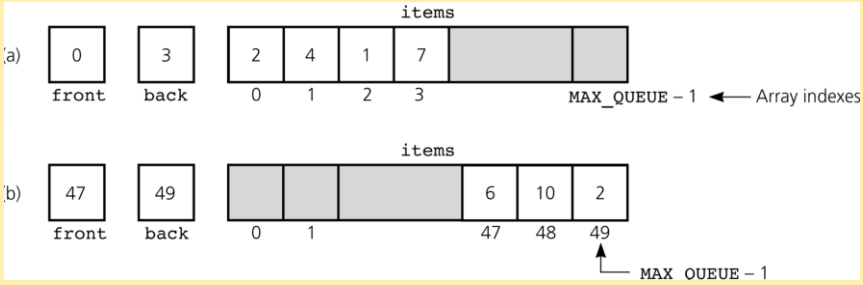


Figure 8-8

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

An Array-Based Implementation

- A circular array eliminates the problem of rightward drift

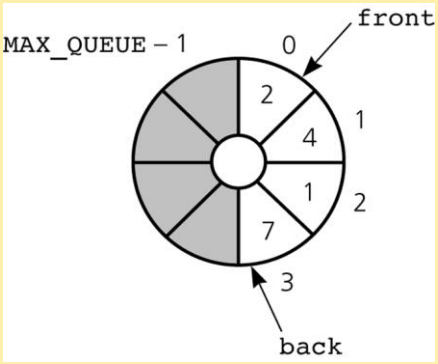


Figure 8-9

A circular implementation of a queue

An Array-Based Implementation

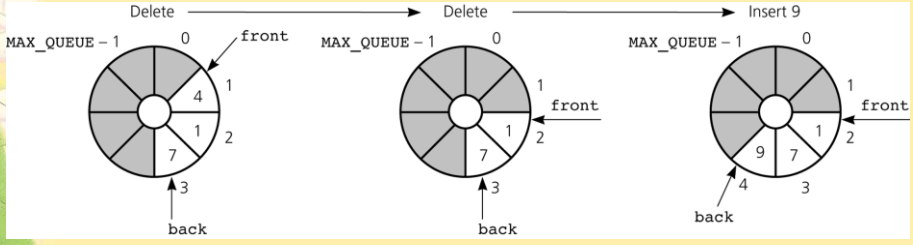


Figure 8-10

The effect of some operations of the queue in Figure 8-8

An Array-Based Implementation

- A problem with the circular array implementation
  - `front` and `back` cannot be used to distinguish between queue-full and queue-empty conditions

An Array-Based Implementation

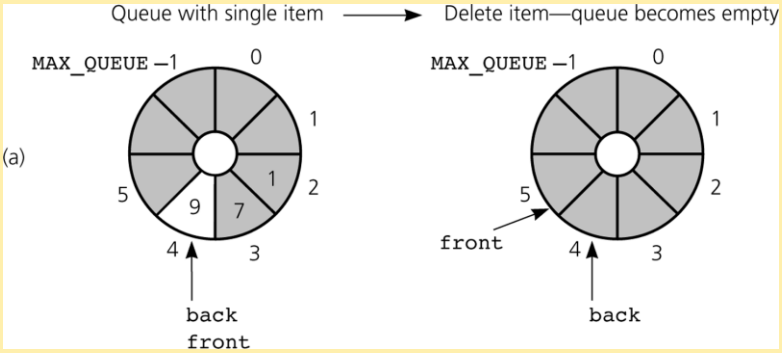


Figure 8-11a

a) ***front*** passes ***back*** when the queue becomes empty

An Array-Based Implementation

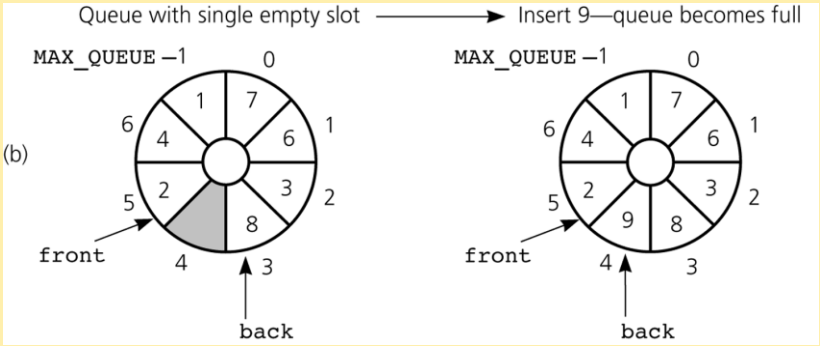


Figure 8-11b

b) ***back*** catches up to ***front*** when the queue becomes full

An Array-Based Implementation

- Inserting into a queue

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

- Deleting from a queue

```
front = (front+1) % MAX_QUEUE;
--count;
```


An Array-Based Implementation

- Variations of the array-based implementation
  - Use a flag `full` to distinguish between the full and empty conditions
  - Declare `MAX_QUEUE + 1` locations for the array items, but use only `MAX_QUEUE` of them for queue items

An Array-Based Implementation

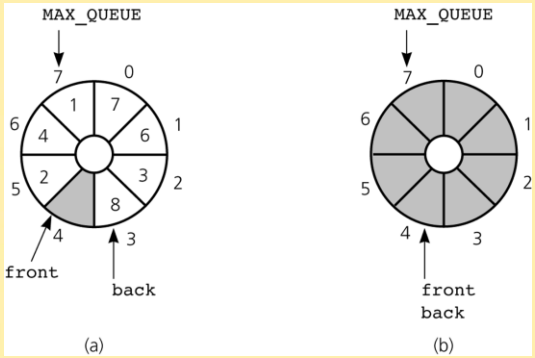


Figure 8-12

A more efficient circular implementation: a) a full queue; b) an empty queue