An Array-Based Implementation

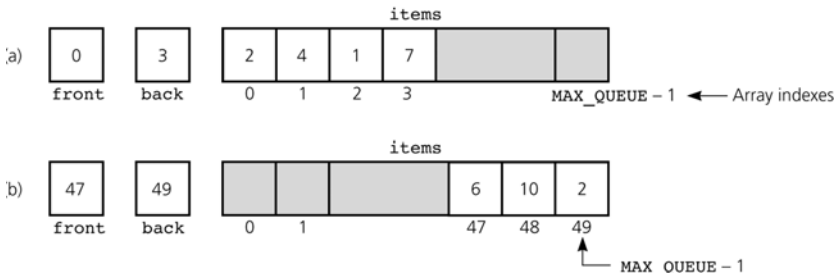

Figure 8-8

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

© 2006 Pearson Addison-Wesley. All rights reserved

An Array-Based Implementation

- A circular array eliminates the problem of rightward drift

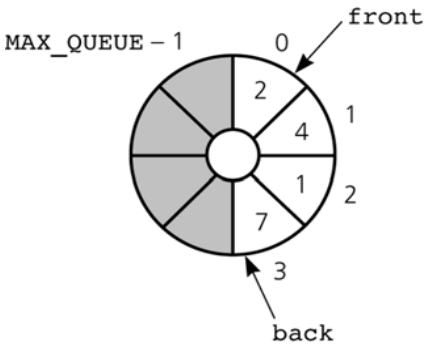

Figure 8-9

A circular implementation of a queue

© 2006 Pearson Addison-Wesley. All rights reserved

An Array-Based Implementation

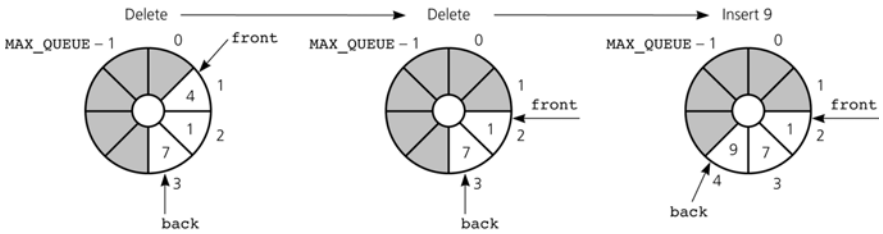

Figure 8-10

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

© 2006 Pearson Addison-Wesley. All rights reserved

An Array-Based Implementation

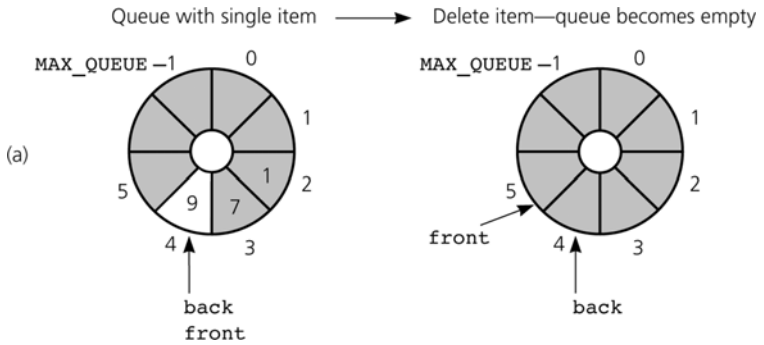

Figure 8-11a

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

© 2006 Pearson Addison-Wesley. All rights reserved

An Array-Based Implementation

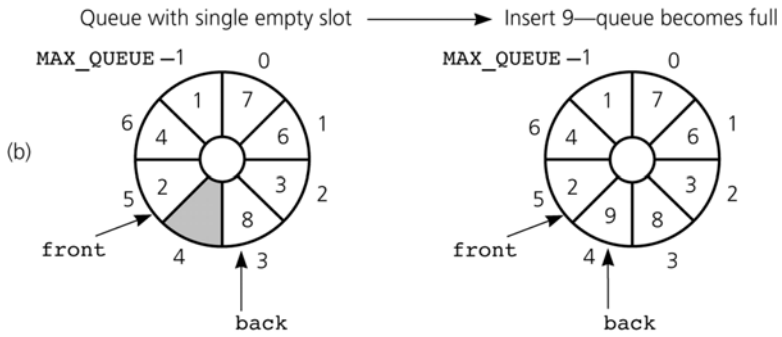

Figure 8-11b

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

© 2006 Pearson Addison-Wesley. All rights reserved

An Array-Based Implementation

- To detect queue-full and queue-empty conditions
  - Keep a count of the queue items
- To initialize the queue, set
  - `front` to 0
  - `back` to `MAX_QUEUE - 1`
  - `count` to 0

© 2006 Pearson Addison-Wesley. All rights reserved

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;
```

© 2006 Pearson Addison-Wesley. All rights reserved

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

© 2006 Pearson Addison-Wesley. All rights reserved

An Array-Based Implementation

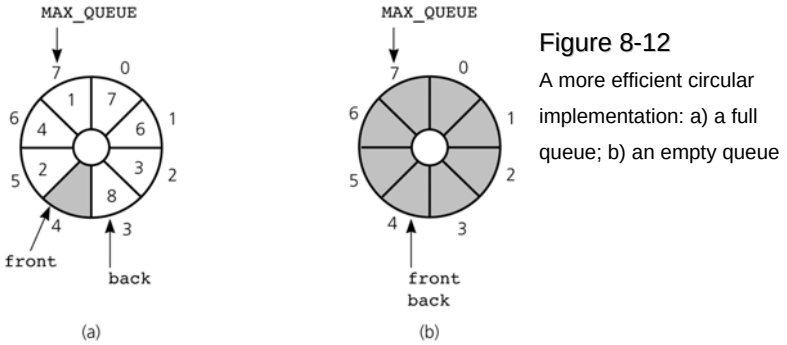

Figure 8-12
A more efficient circular implementation: a) a full queue; b) an empty queue

© 2006 Pearson Addison-Wesley. All rights reserved

An Implementation That Uses the ADT List

- If the item in position 1 of a list `list` represents the front of the queue, the following implementations can be used
  - `dequeue()`
    `list.remove(1)`
  - `peek()`
    `list.get(1)`

© 2006 Pearson Addison-Wesley. All rights reserved

An Implementation That Uses the ADT List

- If the item at the end of the list represents the back of the queue, the following implementations can be used
  - `enqueue(newItem)`
    `list.add(list.size()+1, newItem)`

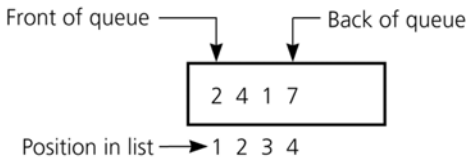

Figure 8-13
An implementation that uses the ADT list

© 2006 Pearson Addison-Wesley. All rights reserved

The Java Collections Framework Interface **`Queue`**

- JCF has a queue interface called **`Queue`**
- Derived from interface **`Collection`**
- Adds methods:
  - `element:` retrieves, but does not remove head
    - Throws an exception if empty
  - `offer:` inserts element into queue
    - Returns false if unable to add
  - `peek:` retrieves, but does not remove head
    - Returns null if empty
  - `poll:` retrieves and removes head
    - Returns null if empty
  - `remove:` retrieves and removes head
    - Throws an exception if empty

© 2006 Pearson Addison-Wesley. All rights reserved

A Summary of Position-Oriented ADTs

- Position-oriented ADTs
  - List
  - Stack
  - Queue
- Stacks and queues
  - Only the end positions can be accessed
- Lists
  - All positions can be accessed

© 2006 Pearson Addison-Wesley. All rights reserved

A Summary of Position-Oriented ADTs

- Stacks and queues are very similar
  - Operations of stacks and queues can be paired off as
    - `createStack` and `createQueue`
    - Stack `isEmpty` and queue `isEmpty`
    - `push` and `enqueue`
    - `pop` and `dequeue`
    - Stack `peek` and queue `peek`

© 2006 Pearson Addison-Wesley. All rights reserved

A Summary of Position-Oriented ADTs

- ADT list operations generalize stack and queue operations
  - `length`
  - `add`
  - `remove`
  - `get`

© 2006 Pearson Addison-Wesley. All rights reserved

Application: Simulation

- Simulation
  - A technique for modeling the behavior of both natural and human-made systems
  - Goal
    - Generate statistics that summarize the performance of an existing system
    - Predict the performance of a proposed system
  - Example
    - A simulation of the behavior of a bank

© 2006 Pearson Addison-Wesley. All rights reserved

Application: Simulation

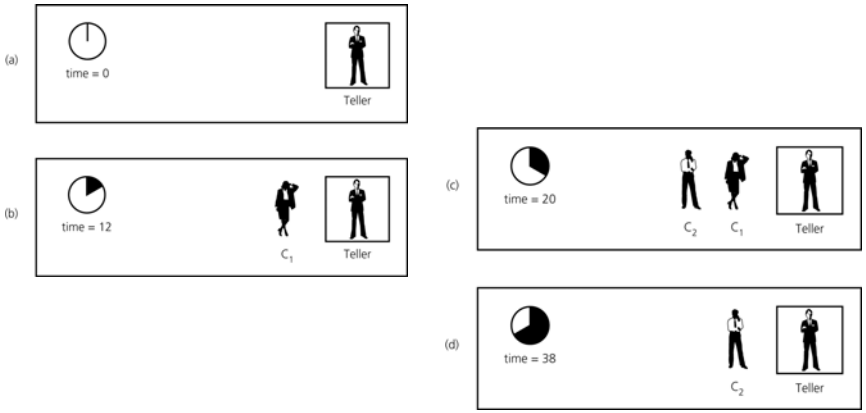

Figure 8-14a ... 8-14d

A bank line at time a) 0; b) 12 c) 20; d) 38

© 2006 Pearson Addison-Wesley. All rights reserved

Application: Simulation

- The bank simulation is concerned with
  - Arrival events
    - Indicate the arrival at the bank of a new customer
    - External events: the input file specifies the times at which the arrival events occur
  - Departure events
    - Indicate the departure from the bank of a customer who has completed a transaction
    - Internal events: the simulation determines the times at which the departure events occur

© 2006 Pearson Addison-Wesley. All rights reserved

Application: Simulation

- An event list is needed to implement an event-driven simulation
  - An event list
    - Keeps track of arrival and departure events that will occur but have not occurred yet
    - Contains at most one arrival event and one departure event

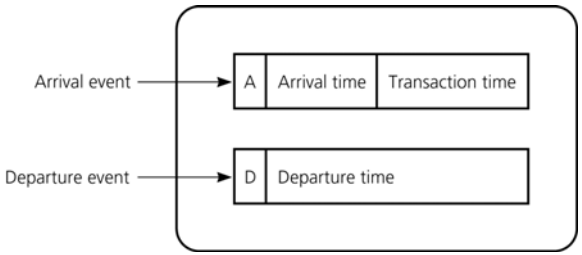

Figure 8-15

A typical instance of the event list

© 2006 Pearson Addison-Wesley. All rights reserved

The ADT Table

- The ADT table, or dictionary
  - Uses a search key to identify its items
  - Its items are records that contain several pieces of data

City	Country	Population
Athens	Greece	2,500,000
Barcelona	Spain	1,800,000
Cairo	Egypt	9,500,000
London	England	9,400,000
New York	U.S.A.	7,300,000
Paris	France	2,200,000
Rome	Italy	2,800,000
Toronto	Canada	3,200,000
Venice	Italy	300,000

Figure 12-1

An ordinary table of cities

© 2006 Pearson Addison-Wesley. All rights reserved

The ADT Table

- Operations of the ADT table
  - Create an empty table
  - Determine whether a table is empty
  - Determine the number of items in a table
  - Insert a new item into a table
  - Delete the item with a given search key from a table
  - Retrieve the item with a given search key from a table
  - Traverse the items in a table in sorted search-key order

© 2006 Pearson Addison-Wesley. All rights reserved

The ADT Table

- Pseudocode for the operations of the ADT table

```
createTable()
// Creates an empty table.

tableIsEmpty()
// Determines whether a table is empty.

tableLength()
// Determines the number of items in a table.

tableInsert(newItem) throws TableException
// Inserts newItem into a table whose items have
// distinct search keys that differ from newItem's
// search key. Throws TableException if the
// insertion is not successful
```

© 2006 Pearson Addison-Wesley. All rights reserved

The ADT Table

- Pseudocode for the operations of the ADT table (Continued)

```
tableDelete(searchKey)
// Deletes from a table the item whose search key
// equals searchKey. Returns false if no such item
// exists. Returns true if the deletion was
// successful.

tableRetrieve(searchKey)
// Returns the item in a table whose search key
// equals searchKey. Returns null if no such item
// exists.

tableTraverse()
// Traverses a table in sorted search-key order.
```

© 2006 Pearson Addison-Wesley. All rights reserved

Selecting an Implementation

- Categories of linear implementations
  - Unsorted, array based
  - Unsorted, referenced based
  - Sorted (by search key), array based
  - Sorted (by search key), reference based

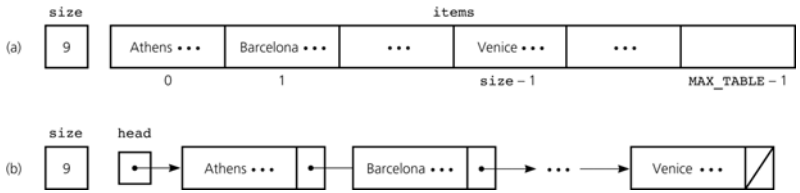

Figure 12-3

The data fields for two sorted linear implementations of the ADT table for the data in Figure 12-1: a) array based; b) reference based

© 2006 Pearson Addison-Wesley. All rights reserved

Selecting an Implementation

- A binary search implementation
  - A nonlinear implementation

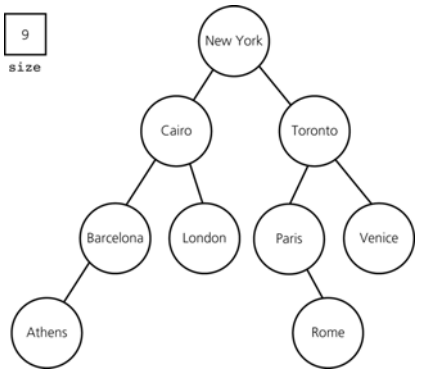

Figure 12-4
The data fields for a binary search tree implementation of the ADT table for the data in Figure 12-1

© 2006 Pearson Addison-Wesley. All rights reserved

Selecting an Implementation

- The binary search tree implementation offers several advantages over linear implementations
- The requirements of a particular application influence the selection of an implementation
  - Questions to be considered about an application before choosing an implementation
    - What operations are needed?
    - How often is each operation required?

© 2006 Pearson Addison-Wesley. All rights reserved