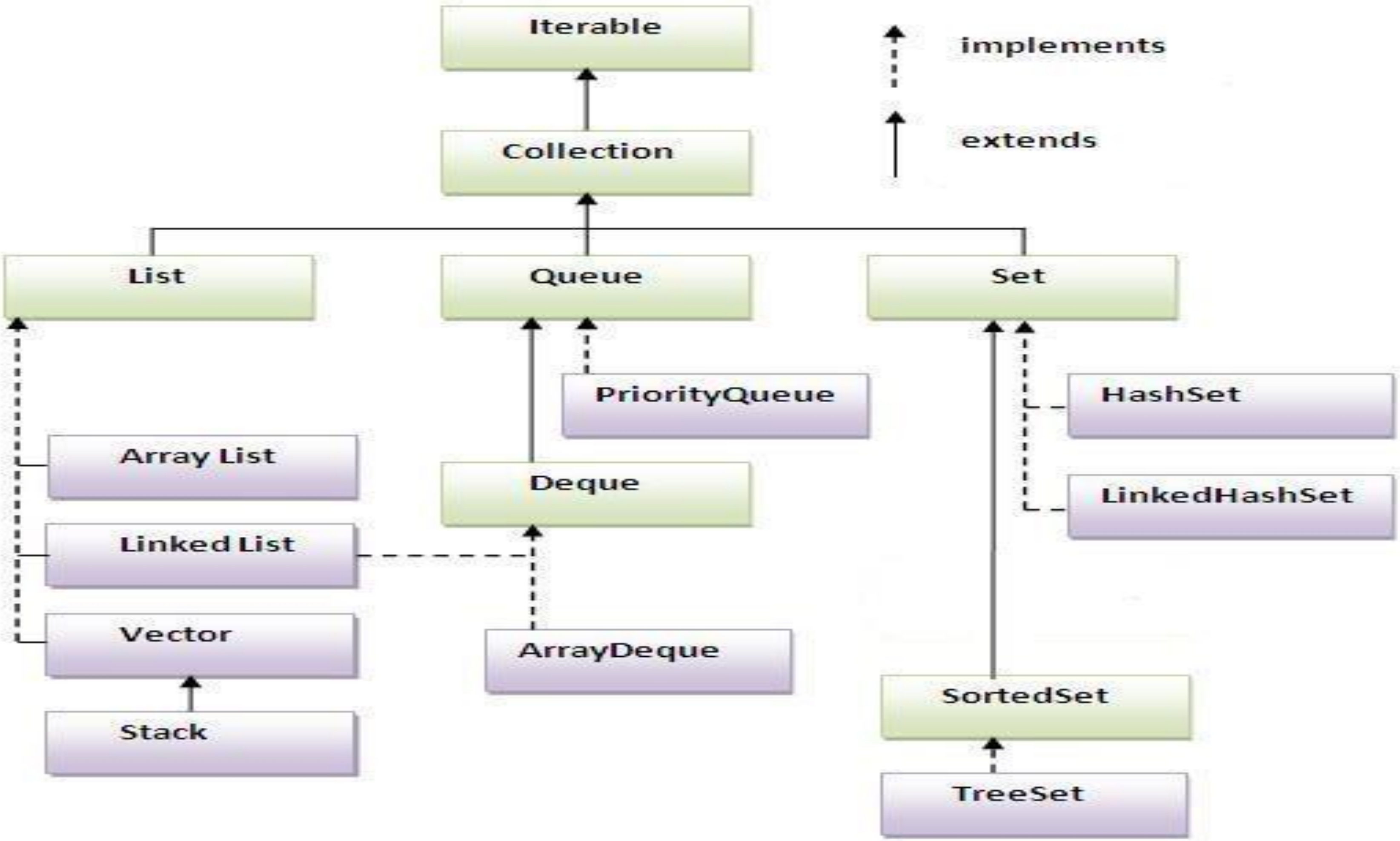
Lesson 4

Queues

Chapter Objectives

- To learn how to represent a waiting line (queue) and how to use the methods in the `Queue` interface for insertion (`offer` and `add`), removal (`remove` and `poll`), and for accessing the element at the front (`peek` and `element`)
- To understand how to implement the `Queue` interface using a single-linked list, a circular array, and a double-linked list
- To become familiar with the `Deque` interface and how to use its methods to insert and remove items from either end of a deque
- To understand how use `Queues` and random number generators to simulate the operation of a physical system that has one or more waiting lines

API Hierarchy



Queue

- The queue, like the stack, is a widely used data structure
- A queue differs from a stack in one important way
  - A stack is LIFO list – *Last-In, First-Out*
  - while a queue is FIFO list, *First-In, First-Out*

Queue Abstract Data Type

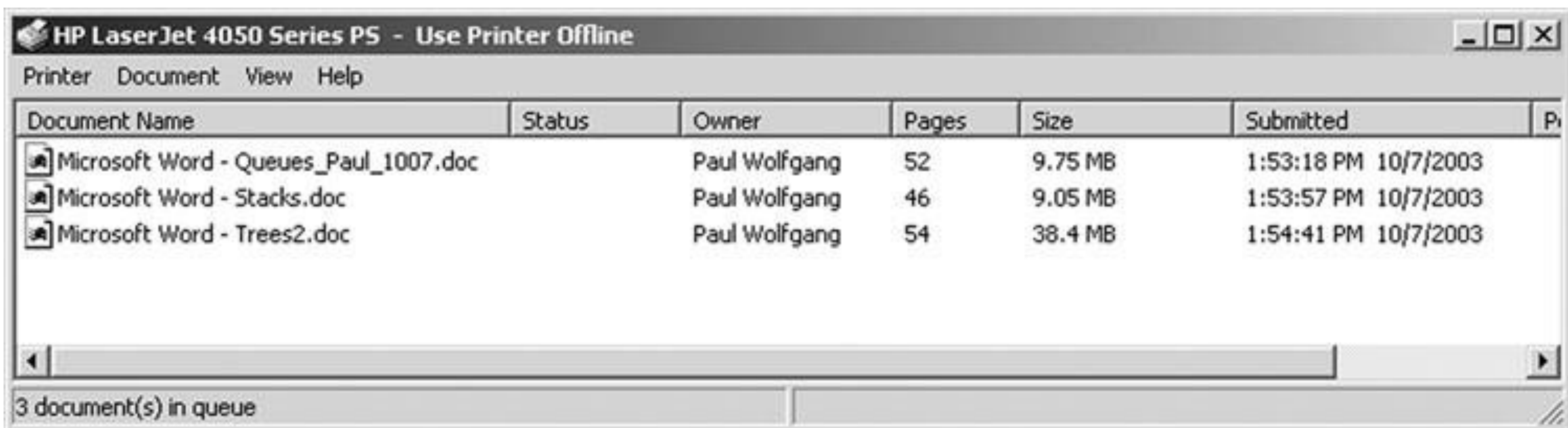
Section 4.1

Queue Abstract Data Type

- A queue can be visualized as a line of customers waiting for service
- The next person to be served is the one who has waited the longest
- New elements are placed at the end of the line

Print Queue

- Operating systems use queues to
  - keep track of tasks waiting for a scarce resource
  - ensure that the tasks are carried out in the order they were generated
- Print queue: printing is much slower than the process of selecting pages to print, so a queue is used

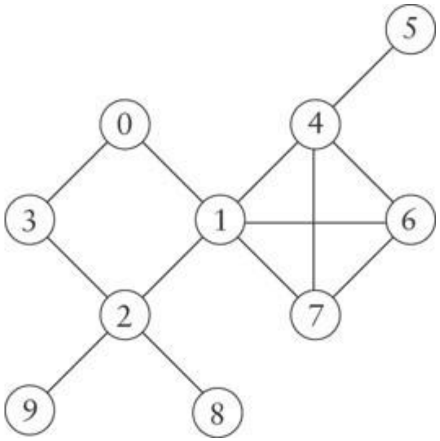


Unsuitability of a Print Stack

- Stacks are Last-In, First-Out (LIFO)
- The most recently selected document would be the next to print
- Unless the printer stack is empty, your print job may never be executed if others are issuing print jobs

Using a Queue for Traversing a Multi-Branch Data Structure

- A graph models a network of nodes, with links connecting nodes to other nodes in the network



- A node in a graph may have several neighbors
- Programmers doing a *breadth-first traversal* often use a queue to ensure that nodes closer to the starting point are visited before nodes that are farther away
- You can learn more about graph traversal in Chapter 10

Specification for a Queue Interface

Method	Behavior
`boolean offer(E item)`	Inserts `item` at the rear of the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the entry at the front of the queue and returns it; returns **`null`** if the queue is empty.
`E peek()`	Returns the entry at the front of the queue without removing it; returns **`null`** if the queue is empty.
`E element()`	Returns the entry at the front of the queue without removing it. If the queue is empty, throws a `NoSuchElementException`.

- The `Queue` interface implements the `Collection` interface (and therefore the `Iterable` interface), so a full implementation of `Queue` must implement all required methods of `Collection` (and the `Iterable` interface)

Class `LinkedList` Implements the `Queue` Interface

- The `LinkedList` class provides methods for inserting and removing elements at either end of a double-linked list, which means all `Queue` methods can be implemented easily
- The Java 5.0 `LinkedList` class implements the `Queue` interface

```
Queue<String> names = new LinkedList<String>();
```

  - creates a new `Queue` reference, `names`, that stores references to `String` objects
  - The actual object referenced by `names` is of type `LinkedList<String>`, but because `names` is a type `Queue<String>` reference, you can apply only the `Queue` methods to it

Maintaining a Queue of Customers

Section 4.2

Maintaining a Queue of Customers

- Write a menu-driven program that maintains a list of customers
- The user should be able to:
  - insert a new customer in line
  - display the customer who is next in line
  - remove the customer who is next in line
  - display the length of the line
  - determine how many people are ahead of a specified person

Designing a Queue of Customers

- Use `JOptionPane.showOptionDialog()` for the menu
- Use a queue as the underlying data structure
- Write a `MaintainQueue` class which has a `Queue<String>` component `customers`

Data Field	Attribute
`private Queue<String> customers`	A queue of customers.
Method	**Behavior**
`public static void processCustomers()`	Accepts and processes each user's selection.

Designing a Queue of Customers (cont.)

Algorithm for `processCustomers`

```
1.  while the user is not finished
2.      Display the menu and get the selected operation
3.      Perform the selected operation
```

Algorithm for determining the position of a Customer

```
1.  Get the customer name
2.  Set the count of customers ahead of this one to 0
3.  for each customer in the queue
4.      if the customer is not the one sought
5.          increment the counter
6.      else
7.          display the count of customers and exit the loop
8.  if all the customers were examined without success
9.      display a message that the customer is not in the queue
```

Implementing a Queue of Customers

- Listing 4.1(`MaintainQueue`, page 202)
- Listing 4.2 (`method processCustomers` in `Class MaintainQueue`, pages 203-204)

Implementing the Queue Interface

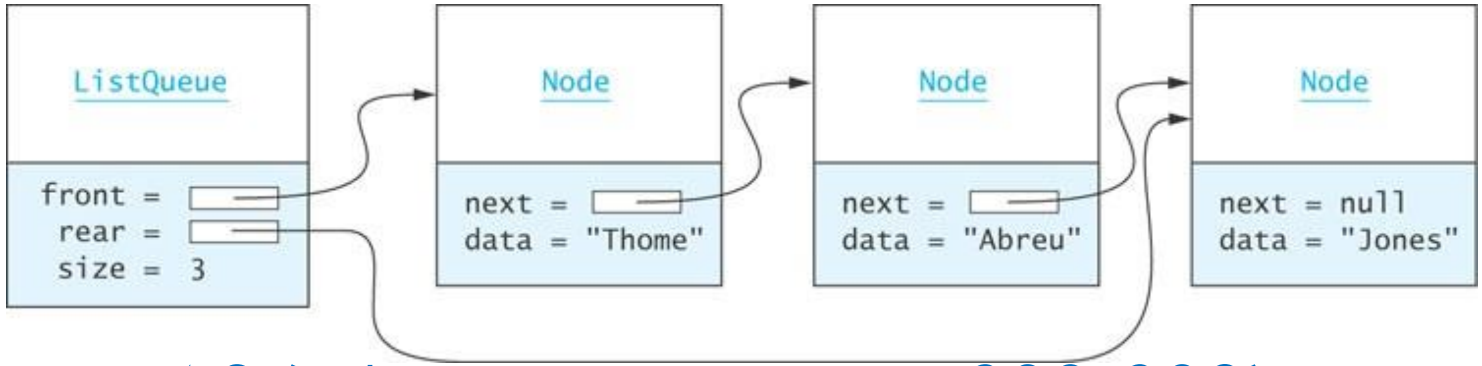
Section 4.3

Using a Double-Linked List to Implement the `Queue` Interface

- Insertion and removal from either end of a double-linked list is O(1) so either end can be the front (or rear) of the queue
- Java designers decided to make the head of the linked list the front of the queue and the tail the rear of the queue

Using a Single-Linked List to Implement a `Queue`

- Insertions are at the rear of a queue and removals are from the front
- We need a reference to the last list node so that insertions can be performed at O(1)
- The number of elements in the queue is changed by methods insert and remove



- Listing 4.3 (`ListQueue`, pages 208-209)

Implementing a Queue Using a Circular Array

- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
  - Insertion at rear of array is constant time O(1)
  - Removal from the front is linear time O(*n*)
  - Removal from rear of array is constant time O(1)
  - Insertion at the front is linear time O(*n*)

Comparing the Three Implementations

- Computation time
  - All three implementations are comparable in terms of computation time
  - All operations are O(1) regardless of implementation
  - Although reallocating an array is O(*n*), its is amortized over *n* items, so the cost per item is O(1)

Comparing the Three Implementations (cont.)

- Storage
  - Linked-list implementations require more storage due to the extra space required for the links
    - Each node for a single-linked list stores two references (one for the data, one for the link)
    - Each node for a double-linked list stores three references (one for the data, two for the links)
  - A double-linked list requires 1.5 times the storage of a single-linked list
  - A circular array that is filled to capacity requires half the storage of a single-linked list to store the same number of elements,
  - but a recently reallocated circular array is half empty, and requires the same storage as a single-linked list

Queue Applications

Section 4.4

Queue Applications

When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc

The Deque Interface

Section 4.5

`Deque` Interface

- A deque (pronounced "deck") is short for double-ended queue
- A double-ended queue allows insertions and removals from both ends
- The Java Collections Framework provides two implementations of the `Deque` interface
  - `ArrayDeque`
  - `LinkedList`
- `ArrayDeque` uses a resizable circular array, but (unlike `LinkedList`) does not support indexed operations
- `ArrayDeque` is the recommend implementation

Deque Interface (cont.)

Method	Behavior
`boolean offerFirst(E item)`	Inserts `item` at the front of the deque. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`boolean offerLast(E item)`	Inserts `item` at the rear of the deque. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`voidaddFirst(E item)`	Inserts `item` at the front of the deque. Throws an exception if the item could not be inserted.
`voidaddLast(E item)`	Inserts `item` at the rear of the deque. Throws an exception if the item could not be inserted.
`E pollFirst()`	Removes the entry at the front of the deque and returns it; returns **`null`** if the deque is empty.
`E pollLast()`	Removes the entry at the rear of the deque and returns it; returns **`null`** if the deque is empty.
`E removeFirst()`	Removes the entry at the front of the deque and returns it if the deque is not empty. If the deque is empty, throws a `NoSuchElementException`.
`E removeLast()`	Removes the item at the rear of the deque and returns it. If the deque is empty, throws a `NoSuchElementException`.
`E peekFirst()`	Returns the entry at the front of the deque without removing it; returns **`null`** if the deque is empty.
`E peekLast()`	Returns the item at the rear of the deque without removing it; returns **`null`** if the deque is empty.
`E getFirst()`	Returns the entry at the front of the deque without removing it. If the deque is empty, throws a `NoSuchElementException`.
`E getLast()`	Returns the item at the rear of the deque without removing it. If the deque is empty, throws a `NoSuchElementException`.
`boolean removeFirstOccurrence(Object item)`	Removes the first occurrence of item in the deque. Returns **`true`** if the item was removed.
`boolean removeLastOccurrence(Object item)`	Removes the last occurrence of item in the deque. Returns **`true`** if the item was removed.
`Iterator<E> iterator()`	Returns an iterator to the elements of this deque in the proper sequence.
`Iterator<E> descendingIterator()`	Returns an iterator to the elements of this deque in reverse sequential order.

Deque **Example**

Deque Method	Deque d	Effect
d.offerFirst('b')	b	'b' inserted at front
d.offerLast('y')	by	'y' inserted at rear
d.addLast('z')	byz	'z' inserted at rear
d.addFirst('a')	abyz	'a' inserted at front
d.peekFirst()	abyz	Returns 'a'
d.peekLast()	abyz	Returns 'z'
d.pollLast()	aby	Removes 'z'
d.pollFirst()	by	Removes 'a'

Deque Interface (cont.)

- The `Deque` interface extends the `Queue` interface, so it can be used as a queue
- A deque can be used as a stack if elements are pushed and popped from the front of the deque
- Using the `Deque` interface is preferable to using the legacy `Stack` class (based on `Vector`)

Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()
empty()	isEmpty()

Priority Queues

- More specialized data structure.
- Similar to Queue, having front and rear.
- Items are removed from the front according to the priority.
- Items are ordered by key value so that the item with the lowest key (or highest) is always at the front.
- Items are inserted in proper position to maintain the order.
- Eg: Used in multitasking operating system.
- Eg: PriorityQueueSale.java

Priority Queues (cont.)

- In a print queue, sometimes it is more appropriate to print a short document that arrived after a very long document
- A *priority queue* is a data structure in which only the highest-priority item is accessible (as opposed to the first item entered)

Insertion into a Priority Queue

```
pages = 1                pages = 4
title = "web page 1"     title = "history paper"
```

After inserting document with 3 pages

```
pages = 1                pages = 3        pages = 4
title = "web page 1"     title = "Lab1"   title = "history paper"
```

After inserting document with 1 page

```
pages = 1                pages = 1           pages = 3        pages = 4
title = "web page 1"     title = "receipt"   title = "Lab1"   title = "history paper"
```

`PriorityQueue` **Class**

- Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4.

Method	Behavior
`boolean offer(E item)`	Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the smallest entry and returns it. If the queue is empty, returns **`null`**.
`E peek()`	Returns the smallest entry without removing it. If the queue is empty, returns **`null`**.
`E element()`	Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException`.