CSIS 3103

Ch 4: Queues

The *Queue* ADT

A ***queue*** is a collection ADT where elements are maintained in the order they were inserted and accessed using the first-in-first-out (FIFO) access protocol.

Operations

1. *Add (Offer)*: Add an element to the back of the queue.
2. *Peek*: If the queue is not empty, return the element that is at the front of the queue.
3. *Remove*: If the queue is not empty, delete and return the element that is at the front of the queue.

Queue ADT

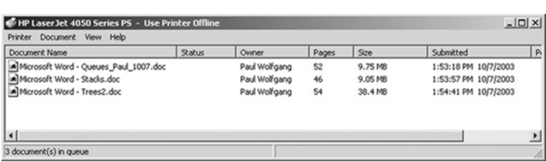
- Can visualize a queue as a line of customers waiting for service
- The next person to be served is the one who has waited the longest
- New customers arrive at the end of the line

Operating Systems Use Queues

- To manage tasks waiting for a scarce resource
- Ensure that the tasks are carried out in the order that they were generated
  - Processes waiting for access to a shared CPU
  - Print jobs waiting for the printer

A Print Queue

Printing is much slower than the process of selecting a print job and so a queue is used



Print Stack? No Way!

- Stacks are last-in, first-out (LIFO)
- The most recently selected document would be the next to print
- Unless the "print stack" is empty, your print job may never get executed if others are issuing print jobs

`Queue` Interface Specification

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`.

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

`LinkedList` Implements the `Queue` Interface

`LinkedList` provides methods for inserting and removing elements at either end of a double-linked list

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

creates a new `Queue` that stores references to `String` objects

`LinkedList` Implements the `Queue` Interface

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

- The actual object referenced by `q` is a `LinkedList<String>`
- Because `q` is a `Queue<String>` reference, it can only call `Queue` methods (actually there are a few non-queue operations included)

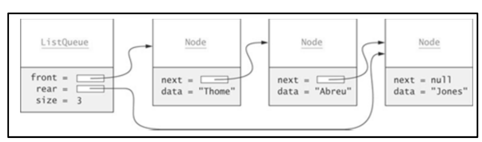
Implementing a `Queue` with a Double-Linked List

- 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

Implementing a Queue with a Single-Linked List

- Class `ListQueue` contains a collection of `Node<E>` objects
- Elements are added at the rear of a queue and removed from the front
- Need references to the first and last list nodes

Implementing a Queue with a Single-Linked List



Class invariant: size ≥ 0 and
size $> 0 \Rightarrow$ `front` references the "oldest" node and
`rear` references the "newest" node

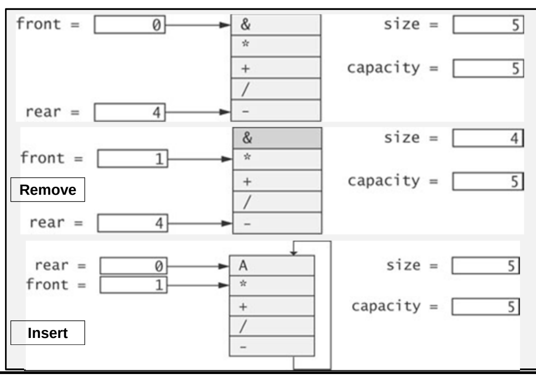
Linked Implementations Compared

- Time efficiency of using a single- or double-linked list to implement a queue are comparable
- But there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored at each list node (especially for double-linked)

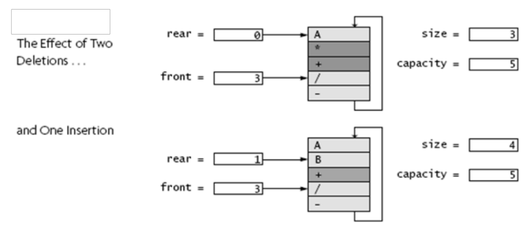
Implementing a Queue with a Circular Array

- Array implementation of lists
  - Add and remove at the rear of array is constant time
  - Add and remove at the front is linear time
- A "circular array" supports the best of both

Queue as a Circular Array

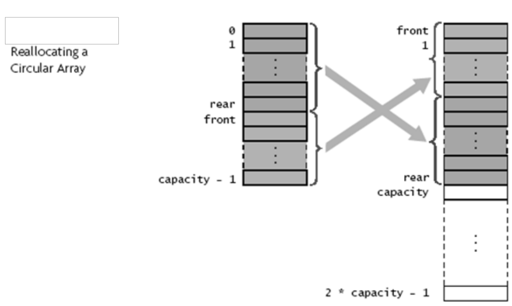


Implementing a Queue with a Circular Array



Class invariant: size ≥ 0 and
front = (rear – size + 1 + capacity) % capacity

Implementing a Queue with a Circular Array



Implementing `ArrayQueue<E>.Iter`

- Like `ListQueue<E>`, we must implement the missing `Queue` methods and an inner class `Iter` to fully implement the `Queue` interface
- `Iter.remove` method throws an `UnsupportedOperationException` because it would violate the contract for a queue to remove an item other than the first one

Comparing Implementations

- Comparable computation times for all three implementations
- Linked-list implementations require more storage because of the extra space required for the links
  - Each single-linked list node stores two references
  - Each double-linked list node stores three references