Queues

CS284

Structure of this week's classes

Queues

Applications

Implementation

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*

Example: Print Queue

- Operating systems use queues to
  - keep track of tasks waiting for a scarce resource
  - ensure 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

The Queue Interface (Sample) – `java.util` (1/2)

```
public interface Queue<E> extends Collection<E> {

// Returns entry at front of queue without removing it. If the
// queue is empty, throws NoSuchElementException
E element()

// Insert an item at the rear of a queue
boolean offer(E item)

// Return element at front of queue without removing it; returns null
E peek()

// Remove and return  entry from front of queue; returns null if queue
E poll()

// Removes entry from front of queue and returns it if queue not empty
E remove()
}
```

The Queue Interface – `java.util` (2/2)

Note:

- `Stack<E>` is a class (derived from Vector) but `Queue<E>` is an interface (derived from Collection)

Queues

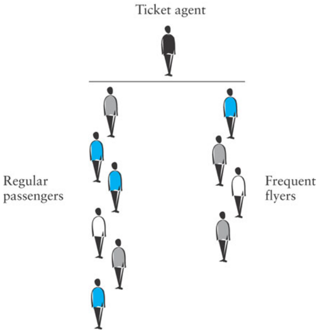
Applications

Implementation

Simulation

- Used to study the performance of a physical system by using a physical, mathematical, or computer model of the system
- Allows designers of a new system to estimate the expected performance before building it
- Can lead to changes in the design that will improve the expected performance of the new system
- Useful when the real system would be too expensive to build or too dangerous to experiment with after its construction
- System designers often use computer models to simulate physical systems
- A branch of mathematics called queuing theory studies such problems

Blue Sky Airlines (BSA) Example



- Two waiting lines:
  - regular customers
  - frequent flyers
- One ticket agent
- Determine average wait time for taking passengers from waiting lines
- Analyze various strategies:
  - take turns serving passengers from both lines (one frequent flyer, one regular, one frequent flyer, etc.)
  - serve the passenger waiting the longest
  - serve any frequent flyers before serving regular passengers

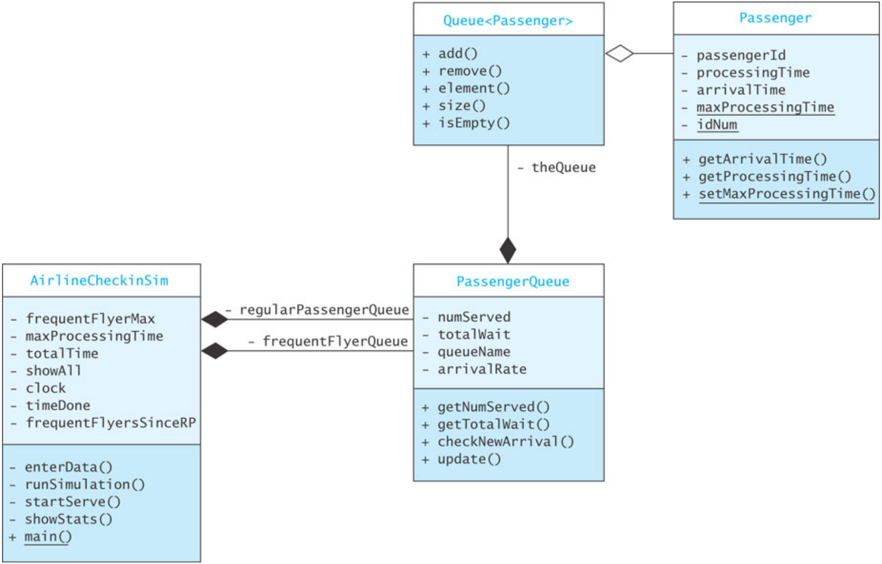
Blue Sky Airlines Example

- To run the simulation, we must keep track of the current time by maintaining a clock set to an initial time of zero
- The clock will increase by one time unit until the simulation is finished
- During each time interval, one or more of the following events occur(s):
  - a new frequent flyer arrives in line
  - a new regular flyer arrives in line
  - the ticket agent finishes serving a passenger and begins to serve a passenger from the frequent flyer line
  - the ticket agent finishes serving a passenger and begins to serve a passenger from the regular passenger line
  - the ticket agent is idle because there are no passengers to serve

Blue Sky Airlines Example

- We can simulate different serving strategies by introducing a simulation variable, `frequentFlyerMax` (> 0)
- `frequentFlyerMax` represents the number of consecutive frequent flyer passengers served between regular passengers
- When `frequentFlyerMax` is:
  - 1, every other passenger served will be a regular passenger
  - 2, every third passenger served will be a regular passenger a very large number, any frequent flyers will be served before regular passengers

Simulation Class Diagrams



Class Passenger

```
import java.util.*;

public class Passenger {
  // Data Fields
  /** The ID number for this passenger. */
  private int passengerId;

  /** The time needed to process this passenger. */
  private int processingTime;

/** The time this passenger arrives. */
  private int arrivalTime;

  /** The maximum time to process a passenger. */
  private static int maxProcessingTime;

  /** The sequence number for passengers. */
  private static int idNum = 0;
```

Class Passenger

```
/** Create a new passenger.
   @param arrivalTime The time this passenger arrives*/
public Passenger(int arrivalTime) {
  this.arrivalTime = arrivalTime;
  processingTime = 1+(new Random()).nextInt(maxProcessingTime);
  passengerId    = idNum++;
}
      /** Get the arrival time.
    @return The arrival time */
public int getArrivalTime() {
  return arrivalTime;
}
```

Class Passenger

```
/** Get the processing time.
      @return The processing time */
  public int getProcessingTime() {
    return processingTime;
  }

/** Get the passenger ID.
      @return The passenger ID */
  public int getId() {
    return passengerId;
  }

  /** Set the maximum processing time
      @param maxProcessingTime The new value */
  public static void setMaxProcessingTime(int maxProcessTime) {
    maxProcessingTime = maxProcessTime;
  }
}
```

Class PassengerQueue

```
import java.util.*;

public class PassengerQueue {
  // Data Fields
  /**  The queue of passengers. */
  private Queue<Passenger> theQueue;

  /** The number of passengers served. */
  private int numServed;
/** The total time passengers were waiting. */
  private int totalWait;

  /** The name of this queue. */
  private String queueName;

  /** The average arrival rate. */
  private double arrivalRate;
```

Class PassengerQueue

```
  // Constructor
  /** Construct a PassengerQueue with the given name.
      @param queueName The name of this queue
   */
 public PassengerQueue(String queueName) {
    numServed = 0;
    totalWait = 0;
    this.queueName = queueName;
    theQueue = new LinkedList<Passenger>();
  }

  /** Return the number of passengers served
      @return The number of passengers served
   */
  public int getNumServed() {
    return numServed;
  }
```

Class PassengerQueue

```java
/** Return the total wait time
      @return The total wait time
   */
  public int getTotalWait() {
    return totalWait;
  }

  /** Return the queue name
      @return - The queue name
   */
  public String getQueueName() {
    return queueName;
  }
```

Class PassengerQueue

```
/** Set the arrival rate
      @param arrivalRate the value to set
   */
  public void setArrivalRate(double arrivalRate) {
    this.arrivalRate = arrivalRate;
  }

  /** Determine if the passenger queue is empty
          @return true if the passenger queue is empty
   */
  public boolean isEmpty() {
    return theQueue.isEmpty();
  }

/** Determine the size of the passenger queue
      @return the size of the passenger queue
   */
  public int size() {
    return theQueue.size();
  }
```

Class PassengerQueue

```java
  /** Check if a new arrival has occurred.
      @param clock The current simulated time
      @param showAll Flag to indicate that detailed
                     data should be output
   */
public void checkNewArrival(int clock, boolean showAll) {
    if (Math.random() < arrivalRate) {
      theQueue.add(new Passenger(clock));
      if (showAll) {
        System.out.println("Time is "
                          + clock + ": "
                          + queueName
                          + "arrival, new queue size is"
                          + theQueue.size());
      }
    }
  }
```

Class PassengerQueue

```java
/** Update statistics.
      pre: The queue is not empty.
      @param clock The current simulated time
      @param showAll Flag to indicate whether to show detail
      @return Time passenger is done being served
   */
  public int update(int clock, boolean showAll) {
    Passenger nextPassenger = theQueue.remove();
    int timeStamp = nextPassenger.getArrivalTime();
    int wait = clock - timeStamp;
    totalWait += wait;
    numServed++;
    // continued
```

Class PassengerQueue

```java
if (showAll) {
      System.out.println("Time is " + clock
                         + ": Serving "
                         + queueName
                         + " with time stamp "
                         + timeStamp);
    }
    return clock + nextPassenger.getProcessingTime();
  }

}
```

class AirlineCheckinSim

```
public class AirlineCheckinSim {

  // Data Fields
  /** Queue of frequent flyers. */
  private PassengerQueue frequentFlyerQueue =
      new PassengerQueue("Frequent Flyer");

  /** Queue of regular passengers. */
  private PassengerQueue regularPassengerQueue =
      new PassengerQueue("Regular Passenger");

  /** Maximum number of frequent flyers to be served
      before a regular passenger gets served. */
  private int frequentFlyerMax;

  /** Maximum time to service a passenger. */
  private int maxProcessingTime;

  /** Total simulated time. */
  private int totalTime;
```

class AirlineCheckinSim

```
/** If set true, print additional output. */
private boolean showAll;

/** Simulated clock. */
private int clock = 0;

/** Time that the agent will be done with the current passenger.*/
private int timeDone;

/** Number of frequent flyers served since the
    last regular passenger was served. */
private int frequentFlyersSinceRP;
```

class AirlineCheckinSim

```
private void runSimulation() {
  for (clock = 0; clock < totalTime; clock++) {
    frequentFlyerQueue.checkNewArrival(clock, showAll);
    regularPassengerQueue.checkNewArrival(clock, showAll);
    if (clock >= timeDone) {
      startServe();
    }
  }
}
```

class AirlineCheckinSim

```
private void startServe() {
   if (!frequentFlyerQueue.isEmpty()
       && ( (frequentFlyersSinceRP <= frequentFlyerMax)
           || regularPassengerQueue.isEmpty())) {
     // Serve the next frequent flyer.
     frequentFlyersSinceRP++;
     timeDone = frequentFlyerQueue.update(clock, showAll);
   }
   else if (!regularPassengerQueue.isEmpty()) {
     // Serve the next regular passenger.
     frequentFlyersSinceRP = 0;
     timeDone = regularPassengerQueue.update(clock, showAll);
   }
   else if (showAll) {
     System.out.println("Time is " + clock + " server is idle");
  }
}
```

class AirlineCheckinSim

```
/** Method to show the statistics. */
 private void showStats() {

   System.out.println
      ("\nThe number of regular passengers served was "
       + regularPassengerQueue.getNumServed());

   double averageWaitingTime =
       (double) regularPassengerQueue.getTotalWait()
       / (double) regularPassengerQueue.getNumServed();

   System.out.println(" with an average waiting time of "
                      + averageWaitingTime);

  // continues
```

class AirlineCheckinSim

```
    System.out.println("The number of frequent flyers served was "
                       + frequentFlyerQueue.getNumServed());
    averageWaitingTime =
        (double) frequentFlyerQueue.getTotalWait()
        / (double) frequentFlyerQueue.getNumServed();
    System.out.println(" with an average waiting time of "
                       + averageWaitingTime);

    System.out.println("Passengers in frequent flyer queue: "
                       + frequentFlyerQueue.size());
    System.out.println("Passengers in regular passenger queue: "
                       + regularPassengerQueue.size());
  }
}
```

Run a Simulation

You must supply:

- Expected number of frequent flyer arrivals per hour (arrival rate is this value / 60)
- Expected number of regular passenger arrivals per hour (arrival rate is this value / 60)
- The maximum number of frequent flyers served between regular passengers (`frequentFlyerMax`)
- Maximum service time in minutes (`maxProcessingTime`)
- Total simulation time in minutes (`totalTime`)

Run a Simulation

- Expected number of frequent flyer arrivals per hour (arrival rate is this value / 60): 240
- Expected number of regular passenger arrivals per hour (arrival rate is this value / 60): 120
- The maximum number of frequent flyers served between regular passengers (`frequentFlyerMax`): 3
- Maximum service time in minutes (`maxProcessingTime`): 4
- Total simulation time in minutes (`totalTime`): 60

```
The number of regular passengers served was 5
 with an average waiting time of 30.8
The number of frequent flyers served was 20
 with an average waiting time of 17.4
Passengers in frequent flyer queue: 40
Passengers in regular queue: 55
```

Queues

Applications

Implementation

Class `LinkedList` Implements the Queue Interface

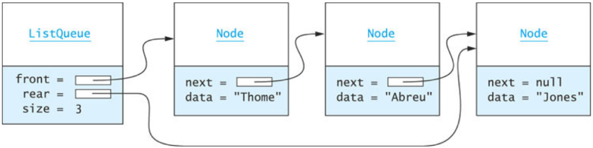
- The `LinkedList` class provides methods for inserting and removing elements at either end of a double-linked list, which means all Queue methods can be implemented easily
- The Java 5.0 LinkedList class implements the Queue interface

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

  - creates a new Queue reference, names, that stores references to String objects

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 $\mathcal{O}(1)$
- The number of elements in the queue is changed by methods insert and remove



Using a Single-Linked List to Implement a Queue

- A comment before beginning
- One might expect to start out with something like:

```
public class ListQueue<E> implements Queue<E> {
    ...
}
```

- However, since `Queue` is a subinterface of other interfaces (namely, `Collection<E>` and `Iterable<E>`), many additional operations would have to be implemented

Using a Single-Linked List to Implement a Queue

- It is best to start off with the abstract class `AbstractQueue` since it implements all operations except for:
  - public boolean offer(E item)
  - public E poll()
  - public E peek()
  - public int size()
  - public Iterator<E> iterator()
- Our implementation shall concentrate on these

```
public class ListQueue<E> extends AbstractQueue<E>
    implements Queue<E> {
   ...
}
```

Using a Single-Linked List to Implement a Queue

```java
import java.util.*;
public class ListQueue<E> extends AbstractQueue<E>
    implements Queue<E> {

  // Data Fields
  /** Reference to front of queue. */
  private Node<E> front;
  /** Reference to rear of queue. */
  private Node<E> rear;
  /** Size of queue. */
  private int size;
```

Using a Single-Linked List to Implement a Queue

```
/** Node is building block for single-linked list. */
private static class Node<E> {
  private E data;
  private Node next;

  /** Creates a new node with a null next field.
      @param dataItem The data stored
   */
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }
  /** Creates a new node that references another node.
      @param dataItem The data stored
      @param nodeRef The node referenced by new node
   */
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
} //end class Node
```

Using a Single-Linked List to Implement a Queue

```java
/** Insert an item at the rear of the queue.
    post: item is added to the rear of the queue.
    @param item The element to add
    @return true (always successful)    */
public boolean offer(E item) {
  // Check for empty queue.
  if (front == null) {
    rear = new Node<E> (item);
    front = rear;
  }
  else {
```

Using a Single-Linked List to Implement a Queue

```
    else {
      // Allocate a new node at end, store item in
      // it, and
      // link it to old end of queue.
      rear.next = new Node<E>(item);
      rear = rear.next;
    }
    size++;
    return true;
  }
```

Using a Single-Linked List to Implement a Queue

```
  /** Return the item at the front of the queue without removi
       @return The item at the front of the queue if successful
   */
  public E peek() {
    if (size == 0)
      return null;
    else
      return front.data;
  }
}
```

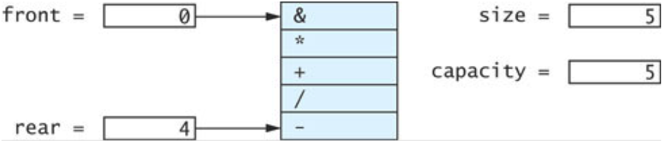
Using a Single-Linked List to Implement a Queue

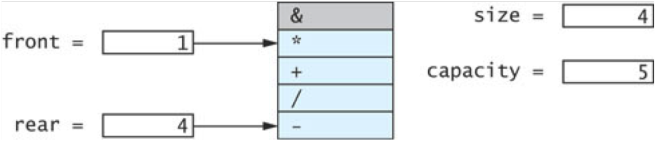
```java
/** Remove the entry at the front of the queue and
     return it if the queue is not empty.
     post: front references item that was 2nd in queue.
     @return Item removed if successful, null othw    */
 public E poll() {
   E item = peek(); // Retrieve item at front.
   if (item == null)
     return null;
   if (size==1) {    // Queue has one item
       front = null;
       rear  = null;
   } else {          // Queue has two or more items
       front = front.next;
   }
   size--;
   return item; // Return data at front of queue.
 }
```

Implementing a Queue Using a Circular Array

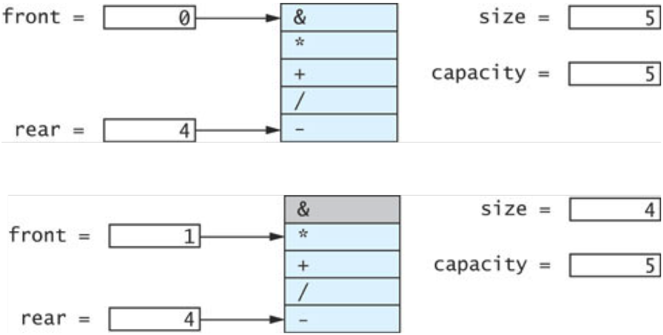
- The time efficiency of using a single- or double-linked list to implement a queue is acceptable
- However, there are some space inefficiencies
- Storage space is increased when using a linked list due to references stored in the nodes
- Array Implementation
  - Insertion at rear of array is constant time $\mathcal{O}(1)$
  - Removal from the front is linear time $\mathcal{O}(n)$ if we shift all elements
  - Removal from rear of array is constant time $\mathcal{O}(1)$
  - Insertion at the front is linear time $\mathcal{O}(n)$ if we shift all elements
- We can avoid these inefficiencies in a circular array

Implementing a Queue Using a Circular Array (cont.)





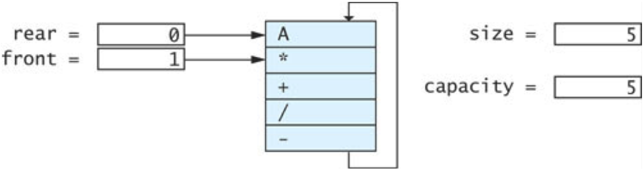
Implementing a Queue Using a Circular Array (cont.)



Now we add A

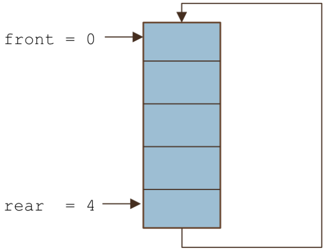
Implementing a Queue Using a Circular Array (cont.)

We add A



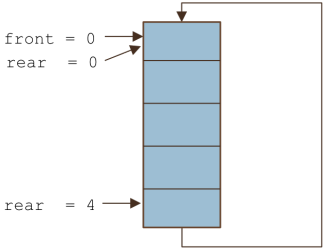
Implementing a Queue Using a Circular Array (cont.)

```
ArrayQueue q = new ArrayQueue(5);
```




```
public ArrayQueue(int initCapacity) {
  capacity = initCapacity;
  theData = (E[])new Object[capacity];
  front = 0;
  rear = capacity - 1;
  size = 0;
}
```

Implementing a Queue Using a Circular Array (cont.)

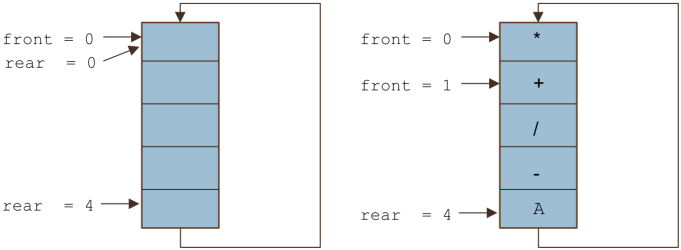



```java
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

Let's see an example

Implementing a Queue Using a Circular Array (cont.)

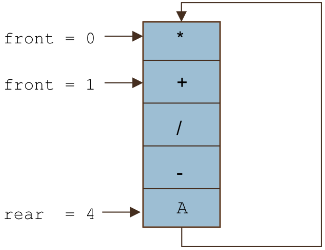
q.offer('*');q.offer('+');q.offer('/');q.offer('-');q.offer('A');




```
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

Implementing a Queue Using a Circular Array (cont.)

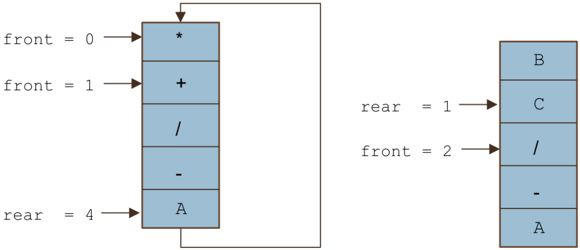
`next = q.poll();next = q.poll();`




```
public E poll() {
  if (size == 0) {
    return null
  }
  E result = theData[front];
  front = (front + 1) % capacity;
  size--;
  return result;
}
```

Implementing a Queue Using a Circular Array (cont.)

`q.offer('B');q.offer('C')`




```
public boolean offer(E item) {
  if (size == capacity) {
    reallocate();
  }
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

Implementing a Queue Using a Circular Array (cont.)

```java
private void reallocate() {
  int newCapacity = 2 * capacity;
  E[] newData = (E[])new Object[newCapacity];
  int j = front;
  for (int i = 0; i < size; i++) {
    newData[i] = theData[j];
    j = (j + 1) % capacity;
  }
  front = 0;
  rear = size - 1;
  capacity = newCapacity;
  theData = newData;
}
```

Comparing the Three Implementations

Computation time

- All three implementations (double-linked list, single-linked list, circular array) are comparable in terms of computation time
- All operations are $\mathcal{O}(1)$ regardless of implementation
- Although reallocating an array is $\mathcal{O}(n)$, it is amortized over n items, so the cost per item is $\mathcal{O}(1)$

Comparing the Three Implementations

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
- All three implementations (double-linked list, single-linked list, circular array) are comparable in terms of computation time