CSC 273 – Data Structures

Lecture 7 - Queues, Deques, and Priority Queues

The ADT Queue

• A queue is another name for a waiting line
• Used within operating systems and to simulate real-world events
  – Come into play whenever processes or events must wait
• Entries organized first-in, first-out
The ADT Queue

Some everyday queues

The ADT Queue

• Terminology
  – Item added first, or earliest, is at the front of the queue
  – Item added most recently is at the back of the queue
• Additions to a software queue must occur at its back
• Client can look at or remove only the entry at the front of the queue
The ADT Queue

**Abstract Data Type: Queue**

- A collection of objects in chronological order and having the same data type

<table>
<thead>
<tr>
<th>Operations</th>
<th>Pseudocode</th>
<th>UML</th>
<th>Description</th>
</tr>
</thead>
</table>
| enqueue()  | enqueue(newEntry: T) | void | Task: Adds a new entry to the back of the queue.  
Input: newEntry is the new entry.  
Output: None. |
| dequeue()  | dequeue(): T | | Task: Removes and returns the entry at the front of the queue.  
Input: None.  
Output: Returns the queue's front entry.  
Throws an exception if the queue is empty before the operation. |

```plaintext
getFront()  
getFront(): T

Task: Retrieves the queue's front entry without changing the queue in any way.  
Input: None.  
Output: Returns the queue's front entry.  
Throws an exception if the queue is empty. |

isEmpty()  
isEmpty(): boolean

Task: Detects whether the queue is empty.  
Input: None.  
Output: Returns true if the queue is empty. |

clear()  
clear(): void

Task: Removes all entries from the queue.  
Input: None.  
Output: None. |
```
The ADT Queue

A queue of strings after (a) enqueue adds Jim; (b) enqueue adds Jess; (c) enqueue adds Jill; (d) enqueue adds Jane;
The ADT Queue

A queue of strings after (e) enqueue adds Joe;
(f) dequeue retrieves and removes Jim;
(g) enqueue adds Jerry;
(h) dequeue retrieves and removes Jess

Simulating a Waiting Line

A line, or queue, of people
Simulating a Waiting Line

WaitLine

Responsibilities
- Simulate customers entering and leaving a waiting line
- Display number served, total wait time, average wait time, and number left in line

Collaborations
- Customer

A CRC card for the class WaitLine

Simulating a Waiting Line

A diagram of the classes WaitLine and Customer
Simulating a Waiting Line

A diagram of the classes `WaitLine` and `Customer`

```
Customer

arrivalTime
transactionTime
customerNumber

getArrivalTime()
getTransactionTime()
getCustomerNumber()
```

Simulating a Waiting Line

```
Algorithm simulate(duration, arrivalProbability, maxTransactionTime)
    transactionTimeLeft = 0
    for (clock = 0; clock < duration; clock++)
    {
        if (a new customer arrives)
            numberOfArrivals++
            transactionTime = a random time that does not exceed maxTransactionTime
            nextArrival = a new customer containing clock, transactionTime, and
            a customer number that is numberOfArrivals
            line.enqueue(nextArrival)
        } if (transactionTimeLeft > 0) // If present customer is still being served transactionTimeLeft--
        else if (!line.isEmpty())
        {
            nextCustomer = line.dequeue()
            transactionTimeLeft = nextCustomer.getTransactionTime() - 1
            timeWaiting = clock - nextCustomer.getArrivalTime()
            totalTimeWaiting = totalTimeWaiting + timeWaiting
            numberServed++
        }
    }
```

Algorithm for `simulate`
Simulating a Waiting Line

A simulated waiting line
public class WaitLine {
    private QueueInterface<Customer> line;
    private int numberOfArrivals;
    private int numberServed;
    private int totalTimeWaited;

    public WaitLine() {
        line = new LinkedQueue<>();
        reset();
    }

    // Simulates a waiting line with one serving
    // agent.
    //    duration - The number of simulated
    //        minutes
    //    arrivalProbability -
    //        A real number between 0 and 1,
    //        and the probability that a
    //        customer arrives at a given time
    //    maxTransactionTime -
    //        The longest transaction time for
    //        a customer
    public void simulate(int duration,
                         double arrivalProbability,
                         int maxTransactionTime) {
        int transactionTimeLeft = 0;
for (int clock = 0; clock < duration; 
clock++)
{
    if (Math.random() < arrivalProbability) {
        numberOfArrivals++;
        int transactionTime = (int)(Math.random()
            * maxTransactionTime + 1);
        Customer nextArrival
            = new Customer(clock,
                transactionTime, numberOfArrivals);
        line.enqueue(nextArrival);
        System.out.println("Customer 
            + numberOfArrivals
            + " enters line at time " + clock
            + ". Transaction time is 
            + transactionTime");
    }

    if (transactionTimeLeft > 0)
    transactionTimeLeft--;
    else if (!line.isEmpty())
    {
        Customer nextCustomer = line.dequeue();
        transactionTimeLeft =
            nextCustomer.getTransactionTime() - 1;
        int timeWaited
            = clock - nextCustomer.getArrivalTime();
        totalTimeWaited
            = totalTimeWaited + timeWaited;
        numberServed++;
        System.out.println("Customer 
            + nextCustomer.getCustomerNumber()
            + " begins service at time " + clock
            + ". Time waited is 
            + timeWaited");
    }
// Displays summary results of the simulation
public void displayResults() {
    System.out.println();
    System.out.println("Number served = 
        + numberServed);
    System.out.println("Total time waited = 
        + totalTimeWaited);
    double averageTimeWaited =
        ((double)totalTimeWaited) / numberServed;
    System.out.println("Average time waited = 
        + averageTimeWaited);
    int leftInLine
        = numberOfArrivals - numberServed;
    System.out.println("Number left in line = 
        + leftInLine);
}

// Initializes the simulation
public final void reset() {
    line.clear();
    numberOfArrivals = 0;
    numberServed = 0;
    totalTimeWaited = 0;
} // end reset
Computing the Capital Gain in a Sale of Stock

A CRC card for the class StockLedger

A diagram of the classes StockLedger and StockPurchase
public class StockLedger {
    private QueueInterface<StockPurchase> ledger;

    public StockLedger() {
        ledger = new LinkedQueue<>();
    } // end default constructor

    // Records a stock purchase in this ledger.
    //    sharesBought - The number of shares purchased.
    //    pricePerShare - The price per share.
    public void buy(int sharesBought, double pricePerShare) {
        while (sharesBought > 0) {
            StockPurchase purchase = new StockPurchase(pricePerShare);
            ledger.enqueue(purchase);
            sharesBought--;
        }
    } // end buy
// Removes from this ledger any shares that
// were sold and computes the capital gain or
// loss.
//    sharesSold   The number of shares sold.
//    pricePerShare The price per share.
//    Returns the capital gain (loss). */
public double sell(int sharesSold,
                   double pricePerShare) {
    double saleAmount
        = sharesSold * pricePerShare;
    double totalCost = 0;

    while (sharesSold > 0) {
        StockPurchase share = ledger.dequeue();
        double shareCost = share.getCostPerShare();
        totalCost = totalCost + shareCost;
        sharesSold--;
    }

    // Gain or loss
    return saleAmount - totalCost;
}
StockPurchase.java

// An immutable class that represents the
// purchase of one share of stock.
public class StockPurchase {
    private double cost;

    public StockPurchase(double costPerShare) {
        cost = costPerShare;
    } // end constructor

    public double getCostPerShare() {
        return cost;
    }
}

Computing the Capital Gain in a
Sale of Stock

A queue of:
(a) individual shares of stock;
(b) grouped shares
Java Class Library: The Interface **Queue**

Methods provided
- `add`
- `offer`
- `remove`
- `poll`
- `element`
- `peek`
- `isEmpty`
- `size`

The ADT Deque

- A double ended queue
- *Deque* pronounced “deck”
- Has both queuelike operations and stacklike operations
The ADT Deque

An instance $d$ of a deque

DequeInterface.java

// An interface for the ADT deque.

class DequeInterface<T> {
    // Adds a new entry to the front/back of this queue.
    // newEntry - An object to be added
    public void addToFront(T newEntry);
    public void addToBack(T newEntry);
// Removes and returns the front/back entry of
// this dequeue.
// Returns - the object at the front/back of
// the dequeue.
// Throws EmptyQueueException if the dequeuer
// is empty before the operation
public T removeFront();
public T removeBack();

// Retrieves the front/back entry of this
// dequeue.
// Returns - the object at the front/back
// of the dequeue.
// Throws EmptyQueueException if the dequeuer
// is empty before the operation
public T getFront();
public T getBack();
The ADT Deque

A comparison of operations for a stack \( s \), a queue \( q \), and a deque \( d \): (a) add; (b) remove; (c) retrieve
The ADT Deque

// Read a line
d = a new empty deque
while (not end of line)
{
    character = next character read
    if (character == 
        d.removeBack()
    else
        d.addToBack(character)
}
// Display the corrected line
while (!d.isEmpty())
   System.out.print(d.removeFront())
   System.out.println()

DequeInterface.java

// An interface for the ADT deque.

public interface DequeInterface<T> {
   // Adds a new entry to the front/back of
   // this deque.
   // newEntry - An object to be added
   public void addToFront(T newEntry);
   public void addToBack(T newEntry);
// Removes and returns the front/back entry
// of this deque
// Returns – the object at the front/back of
// the deque
// Throws EmptyQueueException if the dequeuer
// is empty before the operation
public T removeFront();
public T removeBack();

// Retrieves the front/back entry of this
// dequeue.
// Returns – the object at the front/back of
// the dequeue.
// Throws EmptyQueueException if the dequeuer
// is empty before the operation
public T getFront();
public T getBack();
// Detects whether this dequeue is empty.
// Returns - true if the queue is empty,
// or false otherwise
public boolean isEmpty();

// Removes all entries from this dequeue
public void clear();
}

Computing the Capital Gain in a Sale of Stock

// Records a stock purchase in this ledger.
// sharesBought - The number of shares purchased.
// pricePerShare - The price per share
public void buy(int sharesBought,
    double pricePerShare) {
    StockPurchase purchase = new StockPurchase
        (sharesBought, pricePerShare);
    ledger.addToBack(purchase);
}

Method **buy** creates an instance of **StockPurchase**
and places it at the back of the deque
Computing the Capital Gain in a Sale of Stock

// Removes from this ledger any shares that
// were sold and computes the capital gain or
// loss.
// sharesSold - The number of shares sold
// pricePerShare - The price per share
// Returns  The capital gain (loss).
public double sell(int sharesSold,
                   double pricePerShare) {
    double saleAmount = sharesSold * pricePerShare;
    double totalCost = 0;

    The method sell is more involved

    while (sharesSold > 0) {
        StockPurchase transaction
            = ledger.removeFront();
        double shareCost
            = transaction.getCostPerShare();
        int numberOfShares
            = transaction.getNumberOfShares();

        if (numberOfShares > sharesSold) {
            totalCost = totalCost
                + sharesSold * shareCost;
            int numberToPutBack
                = numberOfShares - sharesSold;
            StockPurchase leftOver = new StockPurchase
                (numberToPutBack, shareCost);

            sharesSold -= numberToPutBack;
        } else {
            totalCost += shareCost;
        }
    }

    return saleAmount - totalCost;
}
// Return leftover shares
// Note: Loop will exit since sharesSold
// will be <= 0 later
ledger.addToFront(leftOver);
}
else
    totalCost = totalCost
        + numberOfShares * shareCost;

    sharesSold = sharesSold - numberOfShares;
} // end while

return saleAmount - totalCost; // Gain or loss
}

Java Class Library: The Interface Deque

Methods provided
• addFirst, offerFirst
• addLast, offerLast
• removeFirst, pollFirst
• removeLast, pollLast
• getFirst, peekFirst
• getLast, peekLast
• isEmpty, clear, size
• push, pop
ADT Priority Queue

- Consider how a hospital assigns a priority to each patient that overrides time at which patient arrived.
- ADT priority queue organizes objects according to their priorities
- Definition of “priority” depends on nature of the items in the queue

PriorityQueueInterface.java

// An interface for the ADT priority queue.
public interface PriorityQueueInterface<T extends Comparable<? super T>> {
    // Adds a new entry to this priority queue
    // newEntry - An object to be added.
    public void add(T newEntry);
// Removes and returns the entry having the
// highest priority
// Returns - either the object having the
// highest priority or, if the priority
// queue is empty before the operation,
// null
public T remove();

// Retrieves the entry having the highest
// priority
// Returns - either the object having the
// highest priority or, if the
// priority queue is empty, null
public T peek();

// Detects whether this priority queue is
// empty
// Returns - true if the priority queue is
// empty, or false otherwise
public boolean isEmpty();

// Gets the size of this priority queue
// Returns - the number of entries currently
// in the priority queue
public int getSize();

// Removes all entries from this priority
// queue
public void clear();
Tracking Your Assignments

Assignment

course—the course code

getCourseCode()

task—a description of the assignment
date—the due date

getTask()

getDueDate()

compareTo()

A diagram of the class Assignment

AssignmentLog

log—a priority queue of assignments

addProject(newAssignment)

addProject(courseCode, task, dueDate)

getNextProject()

removeNextProject()

A diagram of the class AssignmentLog
import java.sql.Date;
// A class that represents a log of assignments
// ordered by priority.

public class AssignmentLog {
    private PriorityQueueInterface<Assignment> log;
    
    public AssignmentLog() {
        log = new LinkedPriorityQueue<>();
    }
    
    The class AssignmentLog

    public void addProject
       (Assignment newAssignment) {
        log.add(newAssignment);
    }

    public void addProject(String courseCode,
                           String task, Date dueDate) {
        Assignment newAssignment = new Assignment
                                (courseCode, task, dueDate);
        addProject(newAssignment);
    }

    public Assignment getNextProject() {
        return log.peek();
    }
}
public Assignment removeNextProject() {
    return log.remove();
}

Java Class Library: The Class PriorityQueue

Basic constructors and methods
- PriorityQueue
- add
- offer
- remove
- poll
- element
- peek
- isEmpty, clear, size
Linked Implementation of a Queue

A chain of linked nodes that implements a queue

LinkedQueue.java

public final class LinkedQueue<T> implements QueueInterface<T> {
    // References node at front of queue
    private Node firstNode;
    // References node at back of queue
    private Node lastNode;

    public LinkedQueue() {
        firstNode = null;
        lastNode = null;
    }

    An outline of a linked implementation of the ADT queue
The Private class **Node**

```java
private class Node {
    private T data; // Entry in queue
    private Node next; // Link to next node

    // Constructors and methods getData, setData,
    // getNxtNode, setNextNode go here
}
```

**LinkedQueue.java**

```java
public void enqueue(T newEntry) {
    Node newNode = new Node(newEntry, null);

    if (isEmpty())
        firstNode = newNode;
    else
        lastNode.setNextNode(newNode);

    lastNode = newNode;
}
```

*The definition of **enqueue**

*Performance is O(1)*
Linked Implementation of a Queue

(a) Before adding a new node to an empty chain
(b) After adding it

linked implementation of a queue

(a) Before
(b) During adding a new node to the end
of a nonempty chain that has a tail reference
Linked Implementation of a Queue

(c) After adding a new node to the end of a nonempty chain that has a tail reference

```
public T getFront() {
    if (isEmpty())
        throw new EmptyQueueException();
    else
        return firstNode.getData();
}

Retrieving the front entry
```
Linked Implementation of a Queue

public T dequeue() {
   // Might throw EmptyQueueException
   T front = getFront();
   assert firstNode != null;
   firstNode.setData(null);
   firstNode = firstNode.getNextNode();

   if (firstNode == null)
      lastNode = null;

   return front;
}

Removing the front entry

Linked Implementation of a Queue

(a)

- firstNode
- Entry at front of queue
- Entry at back of queue
- lastNode

(b)

- firstNode
- Returned to client
- Entry at front of queue
- Entry at back of queue
- lastNode

a) A queue of more than one entry
b) After removing the entry at the front of the queue
Linked Implementation of a Queue

(a) A queue of one entry;
(b) After removing the entry at the front of the queue

```java
public boolean isEmpty() {
    return (firstNode == null) && (lastNode == null);
}

public void clear() {
    firstNode = null;
    lastNode = null;
}
```

Public methods `isEmpty` and `clear`
Array-Based Implementation of a Queue: Circular Array

An array that represents a queue without moving any entries:
(a) initially
(b) after removing the entry at the front twice

Circular Queue

An array that represents a queue without moving any entries:
(c) several more additions, removals;
(d) after two additions that wrap around to beginning of array
A circular array that represents a queue:
(a) when full
(b) after removing two entries
(c) after removing three more entries

A circular array that represents a queue:
(d) after removing all but one entry
(e) after removing the remaining entry
public final class ArrayQueue<T> implements QueueInterface<T> {
    // Circular array of queue entries and one unused location
    private T[] queue;
    private int frontIndex; // Index of front entry
    private int backIndex; // Index of back entry
    private boolean initialized = false;
    private static final int DEFAULT_CAPACITY = 3;
    private static final int MAX_CAPACITY = 10000;

    public ArrayQueue() {
        this(DEFAULT_CAPACITY);
    }

    public ArrayQueue(int initialCapacity) {
        checkCapacity(initialCapacity);
        @SuppressWarnings("unchecked")
        T[] tempQueue = (T[]) new Object[initialCapacity + 1];
        queue = tempQueue;
        frontIndex = 0;
        backIndex = initialCapacity;
        initialized = true;
    }
}
Circular Array with One Unused Location

public void enqueue(T newEntry) {
    checkInitialization();
    ensureCapacity();
    backIndex
        = (backIndex + 1) % queue.length;
    // Index of location after current
    // back of queue
    queue[backIndex] = newEntry;
}

Adding to the back

Circular Array with One Unused Location

public T getFront() {
    checkInitialization();
    if (isEmpty())
        throw new EmptyQueueException();
    else
        return queue[frontIndex];
}

Retrieving the front entry
Circular Array with One Unused Location

(a) A seven-location circular array that contains at most six entries of a queue

(b) Empty queue

(c) Full queue

A seven-location circular array that contains at most six entries of a queue

Circular Array with One Unused Location

(d) A seven-location circular array that contains at most six entries of a queue

(e) Full queue

(f)
Circular Array with One Unused Location

A seven-location circular array that contains at most six entries of a queue

Circular Array with One Unused Location

An array-based queue: (a) initially
Circular Array with One Unused Location

An array-based queue: (b) after removing its front entry by incrementing `frontIndex`;

Circular Array with One Unused Location

An array-based queue: (c) after removing its front entry by setting `queue[frontIndex]` to null and then incrementing `frontIndex`
Circular Array with One Unused Location

```java
public T dequeue() {
    checkInitialization();
    if (isEmpty())
        throw new EmptyQueueException();
    else {
        T front = queue[frontIndex];
        queue[frontIndex] = null;
        // Index of new front of queue
        frontIndex = (frontIndex + 1) % queue.length;
        return front;
    }
}
```

*Implementation of dequeue*

---

Doubling the size of an array-based queue
ensureCapacity()

// Doubles the size of the array queue
// if it is full.
// Precondition: checkInitialization has been
// called.
private void ensureCapacity() {
    if (frontIndex
        == ((backIndex + 2) % queue.length)) {
        // If array is full, double size of array
        T[] oldQueue = queue;
        int oldSize = oldQueue.length;
        int newSize = 2 * oldSize;

        // Queue capacity is 1 fewer than array
        // length
        checkCapacity(newSize - 1);

        // The cast is safe because the new array
        // contains null entries
        @SuppressWarnings("unchecked")
        T[] tempQueue = (T[]) new Object[newSize];
        queue = tempQueue;

        // Number of queue entries = oldSize - 1
        // index of last entry = oldSize - 2
        for (int index = 0; index < oldSize - 1;
            index++) {
            queue[index] = oldQueue[frontIndex];
            frontIndex = (frontIndex + 1) % oldSize;
        }
    }
}
```java
frontIndex = 0;
backIndex = oldSize - 2;
}
}

public boolean isEmpty() {
    return frontIndex ==
            (backIndex + 1) % queue.length);
} // end isEmpty
```

Circular Linked Implementations of a Queue

A circular linked chain with an external reference to its last node that
(a) has more than one node
(b) has one node
(c) is empty
A two-part circular linked chain that represents both a queue and the nodes available to the queue.
Two-Part Circular Linked Chain

A two-part circular linked chain that represents a queue:
(d) after removing the front entry;
(e) after adding one more entry

Two-Part Circular Linked Chain

// A class that implements the ADT queue by
// using a two-part circular chain of nodes

public final class TwoPartCircularLinkedQueue<T> implements QueueInterface<T> {
    // References first node in queue
    private Node queueNode;

    // References node after back of queue
    private Node freeNode;
public TwoPartCircularLinkedQueue() {
    freeNode = new Node(null, null);
    freeNode.setNextNode(freeNode);
    queueNode = freeNode;
} // end default constructor

// Implementations of queue operations follow
// below...

private class Node {
    private T data; // Queue entry
    private Node next; // Link to next node
}
} // end TwoPartCircularLinkedQueue
Two-Part Circular Linked Chain

(a) A chain with nodes available for additions to a queue
(b) the chain after one addition
(c) the chain after another addition
Two-Part Circular Linked Chain

```java
public void enqueue(T newEntry) {
    freeNode.setData(newEntry);
    if (isChainFull()) {
        // Allocate a new node and insert it after
        // the node that freeNode references
        Node newNode = new Node(null,
                                 freeNode.getNextNode());
        freeNode.setNextNode(newNode);
    } // end if
    // Order O(1)
    freeNode = freeNode.getNextNode();
}
```

```java
public T getFront() {
    if (isEmpty())
        throw new EmptyQueueException();
    else
        return queueNode.getData();
}
```

```java
public T dequeue() {
    // Might throw EmptyQueueException
    T front = getFront();
    assert !isEmpty();

    queueNode.setData(null);
    queueNode = queueNode.getNextNode();
    return front;
}
```
public boolean isEmpty() {
    return queueNode == freeNode;
} // end isEmpty

private boolean isChainFull() {
    return queueNode == freeNode.getNextNode();
}
Doubly Linked Implementation of a Deque

public class LinkedDeque<T> implements DequeInterface<T> {

    // References node at front of deque
    private DLNode firstNode;
    // References node at back of deque
    private DLNode lastNode;

    public LinkedDeque() {
        firstNode = null;
        lastNode = null;
    }

    // Deque operations go here

    private class DLNode {
        private T data;  // Deque entry
        private DLNode next;  // Link to next node
        private DLNode previous; // Link to previous node

        // Constructors, accessors and mutators go here
    }
}
**Doubly Linked Implementation of a Deque**

Adding to the back of a nonempty deque:

(a) after the new node is allocated;
(b) after the addition is complete
```java
public void addToFront(T newEntry) {
    DLNode newNode = new DLNode(null, newEntry, firstNode);

    if (isEmpty())
        lastNode = newNode;
    else
        firstNode.setPreviousNode(newNode);

    firstNode = newNode;
}
```

```java
public T removeFront() {
    // Might throw EmptyQueueException
    T front = getFront();

    assert (firstNode != null);
    firstNode = firstNode.getNextNode();

    if (firstNode == null)
        lastNode = null;
    else
        firstNode.setPreviousNode(null);

    return front;
}
```
Doubly Linked Implementation of a Deque

(a) A deque containing at least two entries;
(b) after removing the first node and obtaining a reference to the deque’s new first entry.

removeBack() – O(1)

```java
public T removeBack() {
    // Might throw EmptyQueueException
    T back = getBack();

    assert (lastNode != null);
    lastNode = lastNode.getPreviousNode();

    if (lastNode == null)
        firstNode = null;
    else
        lastNode.setNextNode(null);

    return back;
}
```
Doubly Linked Implementation of a Deque

Two possible implementations of a priority queue using
(a) an array;
(b) a chain of linked nodes