Queue and Its Implementation

Tessema M. Mengistu
Department of Computer Science
Southern Illinois University Carbondale
tessema.mengistu@siu.edu
Room - 3131
Outline

• Queue ADT
• Linked List Based Implementation
• Array Based Implementation
• Vector Based Implementation
• Variants of Queue
  – Double Ended Queue - Deque
  – Priority Queue
Queue ADT

• Another name for a waiting line
• Organizes its entries according to the order in which they were added
• Has a characteristics of First in, first out (FIFO)
  – The first element entered the queue is the first element to be processed
• Has two ends – back (rear) and front
Queue

- All additions to a queue are at its back (rear)
  - Called *enqueue*
  - The recent item added
- All removal from the queue is at its front
  - Called *dequeue*
  - The earliest item added

![Queue Diagram]

- **Add:** 5 15 12 9 10
- **Remove:**
- **Rear:**
- **Front:**
Queue

- Used within operating systems
- Simulate many real world events
ADT Queue

• Data
  – A collection of objects in chronological order and having the same data type

• Operations
  – enqueue(newEntry):void
  – dequeue():T
  – getFront():T
  – isEmpty():boolean
  – clear():void
public interface QueueInterface<T>
{
    /** Adds a new entry to the back of the queue. 
     * @param newEntry an object to be added */
    public void enqueue(T newEntry);

    /** Removes and returns the entry at the front of this queue. 
     * @return either the object at the front of the queue or, if the 
     * queue is empty before the operation, null */
    public T dequeue();

    /** Retrieves the entry at the front of this queue. 
     * @return either the object at the front of the queue or, if the 
     * queue is empty, null */
    public T getFront();

    /** Detects whether this queue is empty. 
     * @return true if the queue is empty, or false otherwise */
    public boolean isEmpty();

    /** Removes all entries from this queue. */
    public void clear();
} // end QueueInterface
Example

```java
QueueInterface<String> myQueue = new LinkedListQueue<String>();
myQueue.enqueue("Jim");
myQueue.enqueue("Jess");
myQueue.enqueue("Jill");
myQueue.enqueue("Jane");
myQueue.enqueue("Joe");

String front = myQueue.getFront(); // returns "Jim"
System.out.println(front + " is at the front of the queue.");

front = myQueue.dequeue(); // removes and returns "Jim"
System.out.println(front + " is removed from the queue.");

myQueue.enqueue("Jerry");
front = myQueue.getFront(); // returns "Jess"
System.out.println(front + " is at the front of the queue.");

front = myQueue.dequeue(); // removes and returns "Jess"
System.out.println(front + " is removed from the queue.");
```
Java Class Library

• Interface **Queue**
  - public boolean add(T newEntry)
  - public boolean offer(T newEntry)
  - public T remove()
  - public T poll()
  - public T element()
  - public T peek()
  - public boolean isEmpty()
  - public void clear()
  - public int size()
Linked Implementation of a Queue

• Consider chain of linked nodes
  – Head reference insufficient
  – Must also have tail reference

• Which should be front of queue?
  – Head easier to be front of queue for entry removal
  – Adding entries at tail/back of queue easily done
public class LinkedQueue<T> implements QueueInterface<T>
{
    private Node firstNode; // references node at front of queue
    private Node lastNode; // references node at back of queue

    public LinkedQueue()
    {
        firstNode = null;
        lastNode = null;
    } // end default constructor

    < Implementations of the queue operations go here. >
    ...

    private class Node
    {
        private T data; // entry in queue
        private Node next; // link to next node

        < Constructors and the methods getData, setData, getNextNode, and setNextNode are here. >
        ...
    } // end Node

} // end LinkedQueue
enqueue Method

• If empty
enqueue Method

- If not empty
enqueue Method

```java
public void enqueue(T newEntry)
{
    Node newNode = new Node(newEntry, null);
    if (isEmpty())
        firstNode = newNode;
    else
        lastNode.setNextNode(newNode);
    lastNode = newNode;
} // end enqueue
```
dequeue Method

- Only one element
dequeue Method

• More than one Elements

(a)

firstNode
Entry at front of queue

Entry at back of queue

lastNode

(b)

firstNode
front
Returned to client

Entry at front of queue

Entry at back of queue

lastNode
public T dequeue()
{
    T front = null;
    if (!isEmpty())
    {
        front = firstNode.getData();
        firstNode = firstNode.getNextNode();
        if (firstNode == null)
            lastNode = null;
    } // end if

    return front;
} // end dequeue
Other Methods

```java
public T getFront()
{
    T front = null;
    if (!isEmpty())
        front = firstNode.getData();
    return front;
} // end getFront

public boolean isEmpty()
{
    return (firstNode == null) && (lastNode == null);
} // end isEmpty

public void clear()
{
    firstNode = null;
    lastNode = null;
} // end clear
```
Array-Based Implementation of a Queue

- Array named `queue`
  - `queue[0]` is front
  - `frontIndex, backIndex` are indices of front and back of queue

![Diagram of queue implementation with array and indices]
Array Based Implementation ...

• What happens during dequeue?
  – With `queue[0]` always as front, must shift elements
    • Not efficient
  – Instead, move `frontIndex`
Array Based Implementation ...

- Then we run off the end of the array!?  

( )

- Solution ?
  - Expand?
    - left many spaces unoccupied
  - Use unoccupied spaces
Array Based Implementation ...

- Once the queue reaches the end of the array, we can add subsequent entries to the queue at the beginning of the array.

- The array behave as **circular**
  - Its first location follows its last one
Array Based Implementation ...

- Increment indices with modulo operator

\[
\text{backIndex} = (\text{backIndex} + 1) \mod \text{queue.length}; \\
\text{frontIndex} = (\text{frontIndex} + 1) \mod \text{queue.length};
\]
Array Based Implementation ...

(c) 0 1 47 48 49

\[ \text{Entry at front of queue} \quad \text{Entry at back of queue} \]

(d) 0 1 47 48 49

\[ \text{Entry at back of queue} \quad \text{Entry at front of queue} \]

frontIndex \quad backIndex

Array Based Implementation ...

- How do we know the queue is full?

$$\text{frontIndex} = \text{backIndex} + 1$$
Array Based Implementation ...

- How do we know the queue is Empty?

\[
\text{frontIndex} = \text{backIndex} + 1
\]
Array Based Implementation ...

• Problem
  – No way to decide whether the queue is empty or full using index

• Solution
  – Have a counter variable and test the variable
    • The enqueue and dequeue methods should manipulate this variable – inefficient
  – Leave one array location unused
Circular Array with One Unused Element

- Allows detection of empty Vs. full queue
  - Examine `frontIndex`, `backIndex`

(a) Empty queue
(b) Full queue
(c) Full queue
(d) Full queue
(e) Full queue
(f) Full queue
• Any pattern?
  – full
    \[ \text{frontIndex} = (\text{backIndex} + 2) \% \text{queue.length} \]
  – Empty
    \[ \text{frontIndex} = (\text{backIndex} + 1) \% \text{queue.length} \]
public class ArrayQueue\<T\> implements QueueInterface\<T\> {
    private T[] queue; // circular array of queue entries and one unused
                        // location
    private int frontIndex;
    private int backIndex;
    private static final int DEFAULT_INITIAL_CAPACITY = 50;

    public ArrayQueue() {
        this(DEFAULT_INITIAL_CAPACITY);
    } // end default constructor

    public ArrayQueue(int initialCapacity) {
        // the cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        T[] tempQueue = (T[]) new Object[initialCapacity + 1];
        queue = tempQueue;
        frontIndex = 0;
        backIndex = initialCapacity;
    } // end constructor

    // Implementations of the queue operations go here.

} // end ArrayQueue
dequeue Method

(a) 

0 | ... | 5 | 6 | 7 | ... | 49

Entry at front of queue
Entry at back of queue

5 frontIndex
7 backIndex

(b) 

0 | ... | 5 | 6 | 7 | ... | 49

front
Returned to client
Entry at front of queue
Entry at back of queue

6 frontIndex
7 backIndex

(c) 

0 | ... | 5 | 6 | 7 | ... | 49

front
Returned to client
Entry at front of queue
Entry at back of queue

6 frontIndex
7 backIndex
public T dequeue()
{
    T front = null;
    if (!isEmpty())
    {
        front = queue[frontIndex];
        queue[frontIndex] = null;
        frontIndex = (frontIndex + 1) % queue.length;
    } // end if
public T getFront()
{
    T front = null;

    if (!isEmpty())
        front = queue[frontIndex];

    return front;
} // end getFront
enqueue Method

```java
public void enqueue(T newEntry)
{
    ensureCapacity();
    backIndex = (backIndex + 1) % queue.length;
    queue[backIndex] = newEntry;
} // end enqueue
```

- ensureCapacity() – reading assignment
Other Methods

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

public void clear()
{
    while(!isEmpty())
        dequeue();
}
```
Vector Based Implementation of a Queue

• Front of queue at beginning of vector
• Vector *add* method used at back of queue
• Remove from front of queue
  – Vector takes care of moving elements
  – No indices needed
• Vector manages additional space as needed
import java.util.Vector;

/**
 * A class that implements a queue of objects by using a vector.
 * @author Frank Carrano
 */

public class VectorQueue<T> implements QueueInterface<T>
{
    private Vector<T> queue; // queue's front entry is first in the vector

    public VectorQueue()
    {
        queue = new Vector<T>(); // vector doubles in size if necessary
    } // end default constructor

    public VectorQueue(int initialCapacity)
    {
        queue = new Vector<T>(initialCapacity);
    } // end constructor

    <Implementations of the queue operations go here.>
    
} // end VectorQueue
Vector Based Implementation ...

• enqueue method

```java
public void enqueue(T newEntry)
{
    queue.add(newEntry);
} // end enqueue
```

• getFront method

```java
public T getFront()
{
    T front = null;
    if (!isEmpty())
        front = queue.get(0);
    return front;
} // end getFront
```
Vector Based Implementation ...

- dequeue method

```java
public T dequeue()
{
    T front = null;
    if (!isEmpty())
        front = queue.remove(0);
    return front;
} // end dequeue
```

- isEmpty method

```java
public boolean isEmpty()
{
    return queue.isEmpty();
} // end isEmpty
```

- clear method

```java
public void clear()
{
    queue.clear();
} // end clear
```
Efficiency of Vector Based Implementation

• Since we add entries to one end of a queue and remove them from the other end, the vector implementation inherently moves its entries after each removal.
  – dequeue() is O(n)
  – Other methods O(1)
• Exercise
  – Create a queue that can contain Strings
  – Add 5 strings to the queue
  – Remove the first two strings from the queue
  – Add additional three strings
  – Display the content of the queue
Double Ended Queue

• Allows add, remove, or retrieve entries at both the front and back of a queue
• In short called deque – pronounced as “deck”
• Has queue like operations and stack like operations
  – addToBack() and removeFront() – queue
  – addToFront() and removeFront() – stack
  – getFront(), getBack(), and removeBack()
Deque ADT

```java
public interface DequeInterface<T>
{
    public void addToFront(T newEntry);
    public void addToBack(T newEntry);
    public T removeFront();
    public T removeBack();
    public T getFront();
    public T getBack();
    public boolean isEmpty();
    public void clear();
} // end DequeInterface
```
Deque ...

The stack s, queue q, or deque d

(a) Add
- s.push(item)
- d.addToFront(item)
- q.enqueue(item)
- d.addToBack(item)

(b) Remove
- s.pop()
- q.dequeue()
- d.removeFront()
- d.removeBack()

(c) Retrieve
- s.peek()
- q.getFront
- d.getFront()
- d.getBack()
Deque ...

• Output?

DequeInterface<String> myDeque = new LinkedListDeque<String>();
myDeque.addToFront("Jim");
myDeque.addToBack("Jess");
myDeque.addToFront("Jill");
myDeque.addToBack("Jane");
String name = myDeque.getFront();
myDeque.addToBack(name);
myDeque.removeFront();
myDeque.addToFront(myDeque.removeBack());
Doubly Linked Implementation of a Deque

• We need a way to traverse the liked nodes from both ends
  – Doubly linked list
public class LinkedDeque<T> implements DequeInterface<T>
{
    private DLNode firstNode; // references node for front of deque
    private DLNode lastNode;  // references node for back of deque

    public LinkedDeque()
    {
        firstNode = null;
        lastNode = null;
    } // end default constructor

    < Implementations of the 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 and the methods getData, setData, getNextNode, setNextNode, 
        getPreviousNode, and setPreviousNode are here. >
        ...
    } // end DLNode

} // end LinkedDeque
addToBack() Method

(a) newNode

(b) lastNode
public void addToBack(T newEntry) 
{
    DLNode newNode = new DLNode(lastNode, newEntry, null);

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

    lastNode = newNode;
} // end addToBack
addToFront() Method

```java
public void addToFront(T newEntry) {
    DLNode newNode = new DLNode(null, newEntry, firstNode);

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

    firstNode = newNode;
} // end addToFront
```
removeFront() Method

(a)

```
firstNode
```

Entry at front of deque

(b)

```
firstNode
```

front

Returned to client

Entry at front of deque
public T removeFront()
{
    T front = null;
    if (!isEmpty())
    {
        front = firstNode.getData();
        firstNode = firstNode.getNextNode();
        if (firstNode == null)
            lastNode = null;
        else
            firstNode.setPreviousNode(null);
    } // end if

    return front;
} // end removeFront
public T removeBack()
{
    T back = null;
    if (!isEmpty())
    {
        back = lastNode.getData();
        lastNode = lastNode.getPreviousNode();

        if (lastNode == null)
            firstNode = null;
        else
            lastNode.setNextNode(null);
    } // end if

    return back;
} // end removeBack
Other Methods

- `getFront()`

```java
public T getFront()
{
    T front = null;
    if (!isEmpty())
        front = firstNode.getData();
    return front;
} // end getFront
```

- `getBack()`?
Other Methods

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

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

- Better clear() implementation??
Java Class Library

• Interface **Deque** – extends Queue
  - public void addFirst(T newEntry)
  - public boolean offerFirst(T newEntry)
  - public void addLast(T newEntry)
  - public boolean offerLast(T newEntry)
  - public T removeFirst()
  - public T pollFirst()
  - public T removeLast()
  - public T pollLast()
  - public T getFirst()
  - public T peekFirst()
  - public T getLast()
  - public T peekLast()
  - public boolean isEmpty()
  - public void clear()
  - public int size()
Java Class Library

• Class `ArrayDeque`
  – Implements `Deque`

• Note – has methods appropriate for `deque`, `queue`, and `stack`
  – Could be used for instances of any of these

• Constructors
  – `public ArrayDeque()`
  – `public ArrayDeque(int initialCapacity)`
Priority Queue

• Organizes objects according to their priorities

• Example
  – Bank Vs Hospital ER

• What exactly is a priority depends on the context of the application

• By making the objects *Comparable*, we can hide this detail in the objects’ method *compareTo*
public interface PriorityQueueInterface<T extends Comparable<? super T>>
{
    /** Adds a new entry to this priority queue. */
    @param newEntry an object */
    public void add(T newEntry);

    /** Removes and returns the item with the highest priority. */
    @return either the object with the highest priority or, if the priority queue is empty before the operation, null */
    public T remove();

    /** Retrieves the item with the highest priority. */
    @return either the object with the highest priority or, if the priority queue is empty, null */
    public T peek();

    /** Detects whether this priority queue is empty. */
    @return true if the priority queue is empty, or false otherwise */
    public boolean isEmpty();

    /** Gets the size of this priority queue. */
    @return the number of entries currently in the priority queue */
    public int getSize();

    /** Removes all entries from this priority queue */
    public void clear();
} // end PriorityQueueInterface
Priority Queue

• Example

```java
PriorityQueueInterface<String> myPriorityQueue =
    new LinkedPriorityQueue<String>();
myPriorityQueue.add("Jane");
myPriorityQueue.add("Jim");
myPriorityQueue.add("Jill");
String name = myPriorityQueue.remove();
myPriorityQueue.add(name);
myPriorityQueue.add("Jess");
```
Priority Queue

• Priority can be implemented using Array, linked List, or Vector

• If a linked chain contains the entries in a priority queue, the entry with the highest priority should occur at the beginning of the chain, where it is easy to remove
Java Class Library

- Class **PriorityQueue** constructors and methods
  - public PriorityQueue()
  - public PriorityQueue(int initialCapacity)
  - public boolean add(T newEntry)
  - public boolean offer(T newEntry)
  - public T remove()
  - public T poll()
  - public T element()
  - public T peek()
  - public boolean isEmpty()
  - public void clear()
  - public int size()