Implementations of Queue, Deque, and Priority Queue

Chapter 11
Pre-requisites: Chapter 10, Chapter 3
Objectives

• Implement ADT queue
  ▪ by using chain of linked nodes, or an array

• Implement ADT deque
  ▪ by using chain of doubly linked nodes
  ▪ Add/delete nodes at either end of doubly linked chain

• Implement ADT priority queue
  ▪ by using array or chain of linked nodes
Contents

• Implementations of Queue
  ▪ A Linked Implementation
  ▪ An Array-Based Implementation
  ▪ A Circular Array
  ▪ A Circular Array with One Unused Location
  ▪ Circular Linked Implementations of a Queue
  ▪ A Two-Part Circular Linked Chain
  ▪ Java Class Library: The Class `AbstractQueue`

• Implementation of Deque
  ▪ A Doubly Linked Implementation

• Possible Implementations of a Priority Queue
Queue Implementation: Linked

• 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;    private Node lastNode;
    public LinkedQueue() {    firstNode = null;    lastNode = null;    }
    public void enqueue(T newEntry) {    }
    public T dequeue() {    }
    public boolean isEmpty() {    return (firstNode == null) && (lastNode == null);    }
    public void clear() {    firstNode = null;    lastNode = null;    }
    private class Node {    private T data;    private Node next;
        // definitions of methods, e.g., getData, getNextNode, setNextNode, …
    }
    }

Figure 11-1 A chain of linked nodes that implements a queue
Queue.enqueue Implementation: Linked

- Note code for linked Implementation [Listing 11-1]

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

```java
private class Node     {
    private T data;
    private Node next;
    // definitions of methods, e.g., constructor, getData, getNextNode, setNextNode, …
}
```

- Test case: empty chain: Figure 11-2 (a) Before adding a new node (b) after adding it
Test case 2: nonempty chain:
Figure 11-3 (a) Before, (b) during, and (c) after adding a new node to the end

(a)

... → node

lastNode  newNode

(b)

... → node → node

lastNode  newNode

After executing
lastNode.setNextNode(newNode);

(c)

... → node → node

lastNode  newNode

After executing
lastNode = newNode;
public T dequeue () {
    T front = null;
    if (!isEmpty ()) {
        front = firstNode.getData ();
        firstNode = firstNode.getNextNode ();
        if (firstNode == null) lastNode = null;
    }
    return front;
}

Fig.11-4 (a) A queue of more than one entry; (b) after removing the front entry
Test case: **Queue with 1 entry**: Figure 11-5 (a) Before (b) after removing the front entry

```java
public T dequeue () {
    T front = null;
    if (! isEmpty ()) {
        front = firstNode.getData ();
        firstNode = firstNode.getNextNode ();
        if (firstNode == null) lastNode = null;
    }
    return front;
}
```
Queue Implementation: Linked

```java
public class LinkedQueue < T > implements QueueInterface < T > {
    private Node firstNode;     private Node lastNode;
    public LinkedQueue () { firstNode = null;  lastNode = null; }
    public void enqueue (T newEntry) {
        Node newNode = new Node (newEntry, null);
        if (isEmpty ())   firstNode = newNode;  else lastNode.setNextNode (newNode);
        lastNode = newNode;
    }
    public T dequeue () {
        T front = null;
        if ( ! isEmpty ())    { front = firstNode.getData (); firstNode = firstNode.getNextNode ();
            if (firstNode == null)  lastNode = null;        }
        return front;
    }
    public boolean isEmpty ()  { return (firstNode == null) && (lastNode == null); }
    public void clear ()     { firstNode = null;  lastNode = null;    }
    private class Node     {    private T data;    private Node next;
        // definitions of methods, e.g., getData, getNextNode, setNextNode, …
    }
}
```
Contents

• Implementations of Queue
  ▪ A Linked Implementation
  ▪ An Array-Based Implementation
  ▪ A Circular Array
  ▪ A Circular Array with One Unused Location
  ▪ Circular Linked Implementations of a Queue
  ▪ A Two-Part Circular Linked Chain
  ▪ Java Class Library: The Class AbstractQueue

• Implementation of Deque
  ▪ A Doubly Linked Implementation

• Possible Implementations of a Priority Queue
Deque Implementation: Doubly Linked

- Needs: ability to move *in two direction* from a node
  - Forward – removeFromFront(), Backward – removeFromBack()
- However, simple linked chain supports only one direction!
- This suggests a doubly linked chain

```java
public class LinkedDeque<T> implements DequeInterface<T> {
    private DLNode firstNode;     private DLNode lastNode;

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

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

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

    // addToBack, addToFront, removeFront, removeBack, …

    private class DLNode {private T data;  private DLNode next;  private DLNode previous;
        // assume constructor newDLNode( previous, data, next)
        // definitions of methods, e.g., getData(), getNextNode(), getPreviousNode(), …
    }
}
```

Figure 11-17 A doubly linked chain with head, tail references
public void addToBack (T newEntry) {  // pp. 296-297
  // assume constructor newDLNode( previous, data, next)
  DLNode newNode = new DLNode (lastNode, newEntry, null);
  if (isEmpty ())  firstNode = newNode;  else lastNode.setNextNode (newNode);
  lastNode = newNode;
}

Fig. 11-18: Case: nonempty deque
(a) after the new node is allocated;
(b) after the addition is complete

Exercise: Draw diagram to show behavior for empty deque.

Ex. Compare addToBack() with enQueue() (slide 6).
// assume constructor newDLNode( previous, data, next)
public void addToBack (T newEntry) {
    DLNode newNode = new DLNode (lastNode, newEntry, null);
    if (isEmpty ())  firstNode = newNode;  else lastNode.setNextNode (newNode);
    lastNode = newNode;
}

// Ex. Fill in the blanks A, B, …, G to provide a definition for addToFront()
public void addToFront (T newEntry) {  // pp. 297
    DLNode newNode = new DLNode (A, B, C);
    if (isEmpty ())  D = newNode;  else C.E (newNode);
    C = newNode;
}

Choices for A, B, …, E:
(1) newEntry (2) firstNode (3) lastNode (4) Null (5) setNextNode (6) setPreviousNode
// assume constructor newDLNode( previous, data, next)
public void addToBack (T newEntry) {
    DLNode newNode = new DLNode (lastNode, newEntry, null);
    if (isEmpty ()) firstNode = newNode;
    else lastNode.setNextNode (newNode);
    lastNode = newNode;
}

public void addToFront (T newEntry) {  // pp. 297
    DLNode newNode = new DLNode (null, newEntry, firstNode);
    if (isEmpty ()) lastNode = newNode;
    else firstNode.setPreviousNode (newNode);
    firstNode = newNode;
}
public T removeFront () { // pp. 297
    T front = null;
    if (!isEmpty ()) { front = firstNode.getData (); firstNode = firstNode.getNextNode ();
        if (firstNode == null) lastNode = null;
        else firstNode.setPreviousNode (null);
    }
    return front;
}

Fig.11-19, pp. 298
(a) A deque w >= 2 entries
(b) after removing front and returning front

Ex.: Diagram behavior for empty deque & singleton deque.

Ex. Compare addToBack() with enQueue() (slide 7-8).
public T removeFront () { // pp. 297
    T front = null;
    if (!isEmpty ()) {front = firstNode.getData (); firstNode = firstNode.getNextNode ();
        if (firstNode == null) lastNode = null; else firstNode.setPreviousNode (null);
    }
    return front;
}

// Ex. Fill in the blanks A, B, …, G to provide a definition for addToFront()
public T removeBack () { // pp. 298
    T back = null;
    if (!isEmpty ()) {back = A.getData (); A = A.getPreviousNode ();
        if (A == null) B = null; else A.setPreviousNode (null);
    }
    return back;
}

Choices for A, B, C:
(1) newEntry (2) firstNode (3) lastNode (4) null (5) setNextNode (6) setPreviousNode

Copyright ©2012 by Pearson Education, Inc. All rights reserved
public T removeFront () {
    T front = null;
    if (!isEmpty()) {
        front = firstNode.getData();
        firstNode = firstNode.getNextNode();
        if (firstNode == null) lastNode = null;
        else firstNode.setPreviousNode(null);
    }
    return front;
}

public T removeBack () { // pp. 298
    T back = null;
    if (!isEmpty()) {
        back = lastNode.getData();
        lastNode = firstNode.getPreviousNode();
        if (lastNode == null) firstNode = null; else lastNode.setNextNode(null);
    }
    return back;
}
Contents

• Implementations of Queue
  ▪ A Linked Implementation
  ▪ An Array-Based Implementation
  ▪ A Circular Array
  ▪ A Circular Array with One Unused Location
  ▪ Circular Linked Implementations of a Queue
  ▪ A Two-Part Circular Linked Chain
  ▪ Java Class Library: The Class AbstractQueue

• Implementation of Deque
  ▪ A Doubly Linked Implementation

• Possible Implementations of a Priority Queue
Possible Implementations of a Priority Queue

Figure 11-20 Two possible implementations of a priority queue using (a) an array; (b) a chain of linked nodes
Contents

• Implementations of Queue
  ▪ A Linked Implementation
  ▪ An Array-Based Implementation
  ▪ A Circular Array
  ▪ A Circular Array with One Unused Location
  ▪ Circular Linked Implementations of a Queue
  ▪ A Two-Part Circular Linked Chain
  ▪ Java Class Library: The Class `AbstractQueue`

• Implementation of Deque
  ▪ A Doubly Linked Implementation

• Possible Implementations of a Priority Queue
Queue Implementation: Array Based

• **Listing 11-2**: Array named `queue`,
  - `queue[0]` is front
  - `frontIndex, backIndex` are indices of front and back of queue

• Now to decide …
  - With `queue[0]` always as front, must shift elements
  - Instead, move `frontIndex`
  - Then we run off the end of the array!? 
Figure 11-6 An array that represents a queue without moving any entries: (a) initially; (b) after removing the entry at the front twice;
Figure 11-6 An array that represents a queue without moving any entries: (c) after several more additions and removals; (d) after two additions that wrap around to the beginning of the array;
Java Class Library: The Class AbstractQueue

• Methods provided
  - 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()
End

Chapter 11