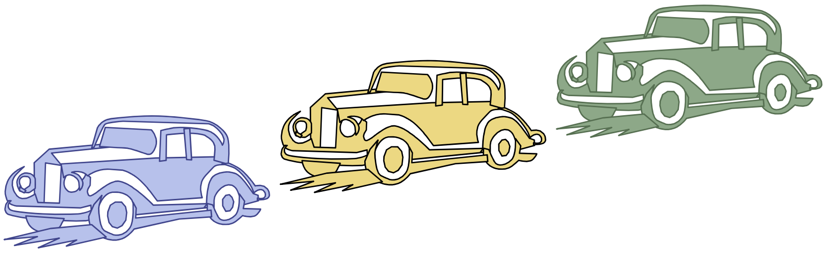
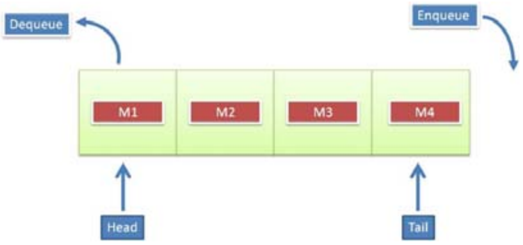
Lecture 4-2. Queues



Sunghyun Cho
Professor
Division of Computer Science
chopro@hanyang.ac.kr

Keywords

- Definition of Queue
- Implementation of Queue
- Usages of Queue
- Double-Ended Queue



The Queue ADT

- The Queue ADT stores arbitrary objects
- **Insertions and deletions follow the first-in first-out scheme**
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Example of Queue Operations

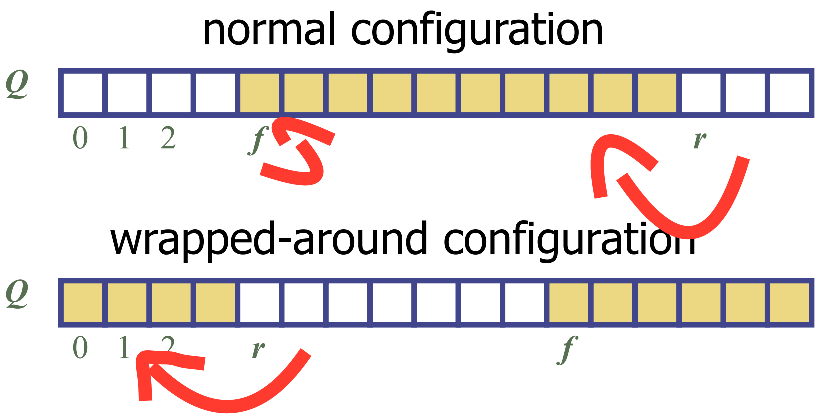
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the front element
  - r index immediately past the rear element
- Array location r is kept empty

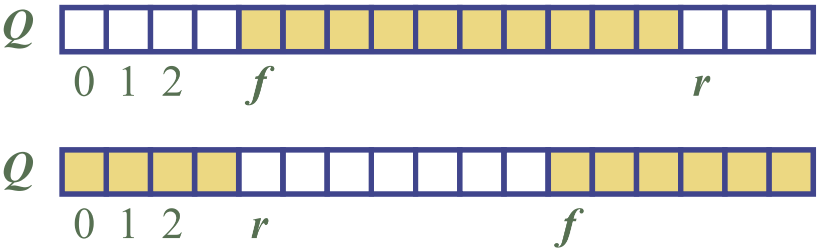

normal configuration

wrapped-around configuration

Queue Operations

- We **use the modulo operator** (remainder of division)

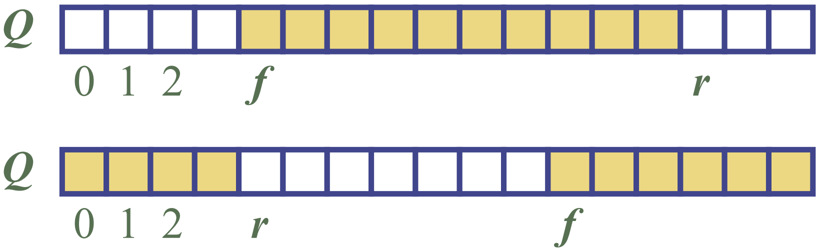
```
Algorithm size()
  return $(N - f + r) \bmod N$

Algorithm isEmpty()
  return $(f = r)$        Empty OR Full
```



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

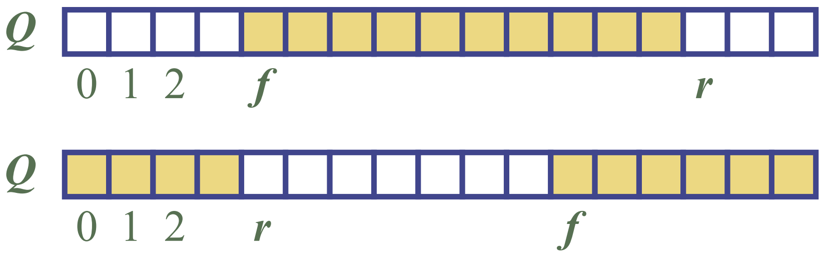
```
Algorithm enqueue(o)
  if $size() = N - 1$ then
    throw FullQueueException
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$
```



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
  if isEmpty() then
    throw EmptyQueueException
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return $o$
```



Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class

```
public interface Queue<E> {
  public int size();
  public boolean isEmpty();
  public E front()
      throws EmptyQueueException;
  public void enqueue(E element);
  public E dequeue()
       throws EmptyQueueException;
}
```

Linked List based Queue

- Using linked list to implement the queue ADT
  - use singly linked list
  - for efficiency
    - choose **the front of the Q** to be at the **head** of the list
    - choose **the rear of the Q** to be at the **tail** of the list
  - in order to perform primitive operations in constant time, we keep track of the references to both the head and tail nodes of the list

- JAVA code
  - **refer to Code Fragment 6.8 (textbook p. 224)**
  - **refer to Code Fragment 6.9 (textbook p. 226)**

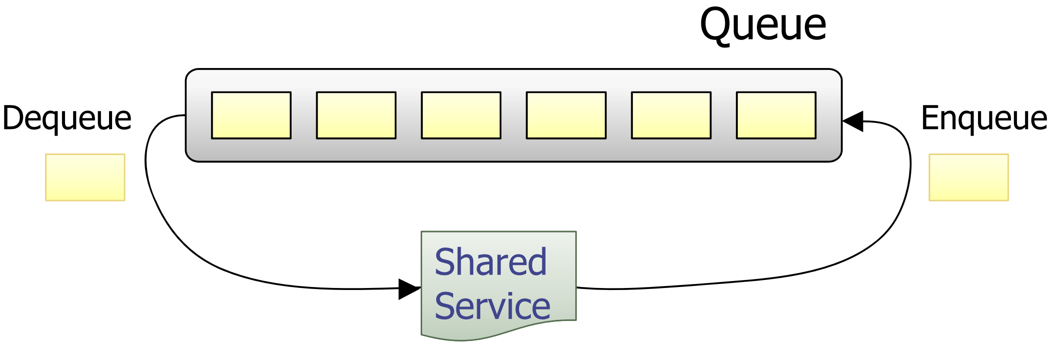
Linked List based Queue (Performance)

- Pros.
  - each of the methods of the singly linked list implementation of the queue ADT runs in $O(1)$ time
  - avoid the need to specify a max. size for the Q (which is required in the array-based queue implementation)

- Cons.
  - increasing the amount of space used per element
  - implementation is more complicated

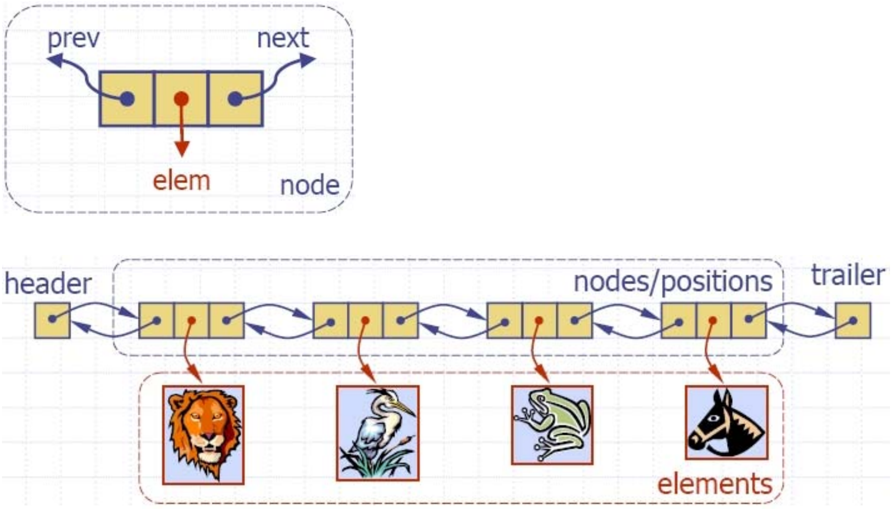
Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



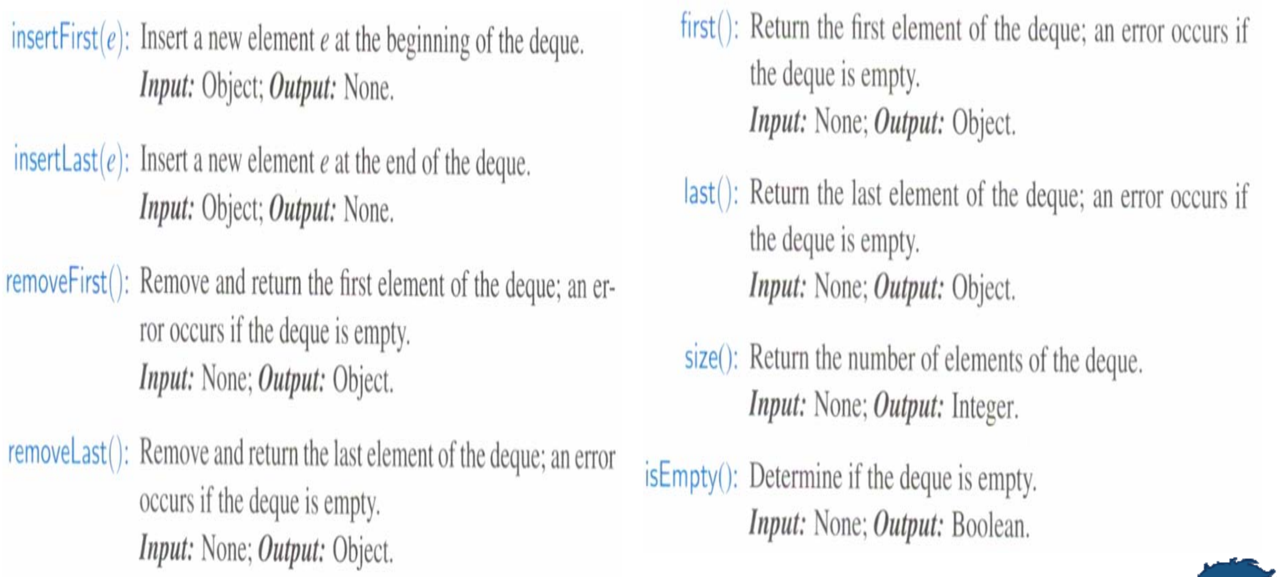
Keywords

- Double-Ended Queues



Double-Ended Queues (Deque)

- Dequeue ADT



insertFirst(*e*): Insert a new element *e* at the beginning of the deque.
Input: Object; ***Output:*** None.

insertLast(*e*): Insert a new element *e* at the end of the deque.
Input: Object; ***Output:*** None.

removeFirst(): Remove and return the first element of the deque; an error occurs if the deque is empty.
Input: None; ***Output:*** Object.

removeLast(): Remove and return the last element of the deque; an error occurs if the deque is empty.
Input: None; ***Output:*** Object.

first(): Return the first element of the deque; an error occurs if the deque is empty.
Input: None; ***Output:*** Object.

last(): Return the last element of the deque; an error occurs if the deque is empty.
Input: None; ***Output:*** Object.

size(): Return the number of elements of the deque.
Input: None; ***Output:*** Integer.

isEmpty(): Determine if the deque is empty.
Input: None; ***Output:*** Boolean.

Deque ADT (cont.)

- Deque : A DS that **supports insertion and deletion at both the front and the rear of the queue**

- Deque ADT

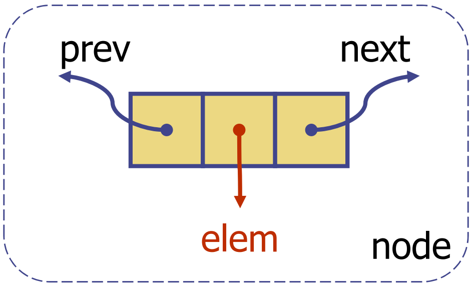
	input	output
– insertFirst(*e*)	Object	None
– insertLast(*e*)	Object	None
– removeFirst()	None	Object
– removeLast()	None	Object
– first()	None	Object
– last()	None	Object
– size()	None	Integer
– isEmpty()	None	Boolean

Example of Deque Operations

Operation	*Output*	*D*
insertFirst(3) –		(3)
insertFirst(5) –		(5, 3)
removeFirst()	*5*	(3)
insertLast(7) –		(3, 7)
removeFirst()	*3*	(7)
removeLast()	*7*	()
removeFirst()	*"error"*	()
isEmpty()	*true*	()

Doubly Linked List

- **A doubly linked list provides a natural implementation of the Deque**
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



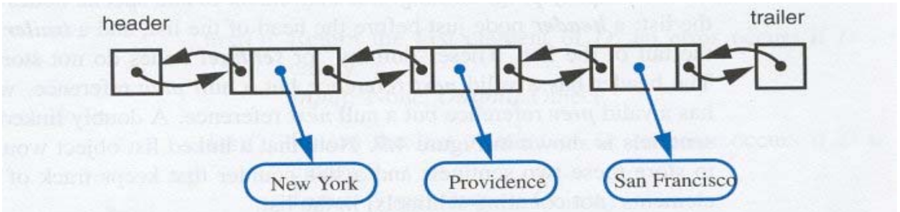
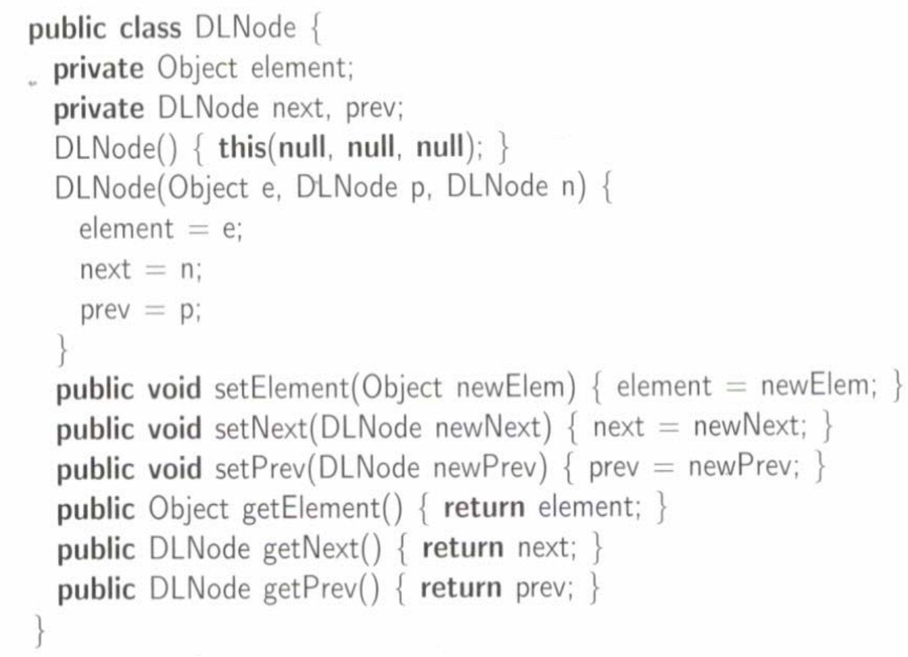


Figure 4.9: A doubly linked list with sentinels, header and trailer, marking the ends of the list. An empty list would have these sentinels pointing to each other. We do not show the null prev pointer for the header nor do we show the null next pointer for the trailer.

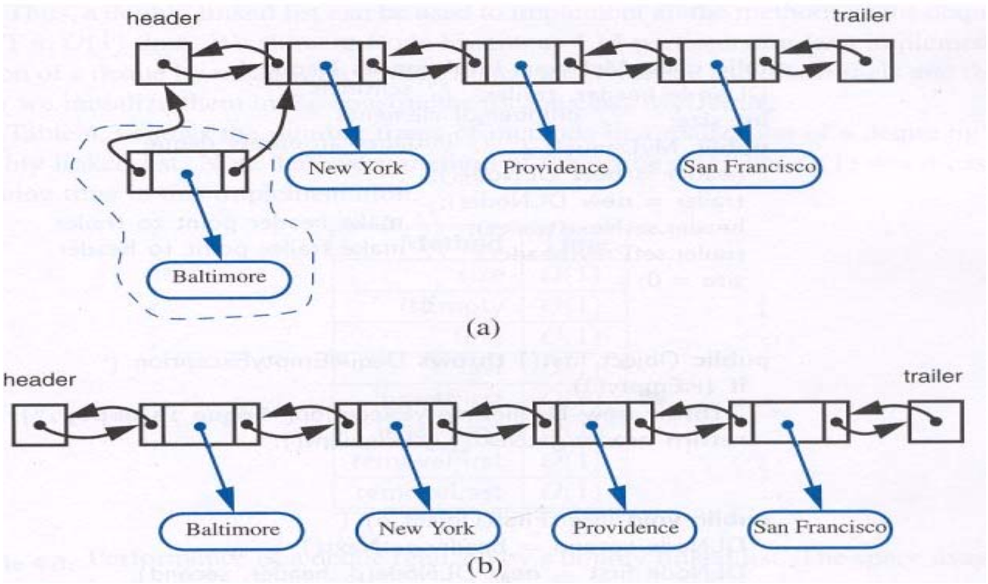
Implementation of a Deque with a Doubly Linked List



```
public class DLNode {
  private Object element;
  private DLNode next, prev;
  DLNode() { this(null, null, null); }
  DLNode(Object e, DLNode p, DLNode n) {
    element = e;
    next = n;
    prev = p;
  }
  public void setElement(Object newElem) { element = newElem; }
  public void setNext(DLNode newNext) { next = newNext; }
  public void setPrev(DLNode newPrev) { prev = newPrev; }
  public Object getElement() { return element; }
  public DLNode getNext() { return next; }
  public DLNode getPrev() { return prev; }
}
```

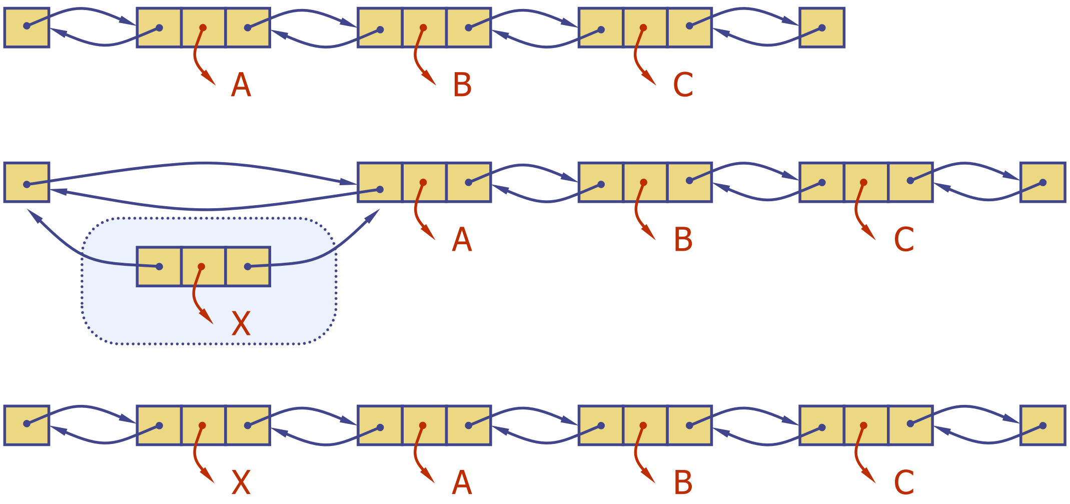
Update Operations

- Insertion at the head



Insertion

- We visualize operation insertFirst(X)

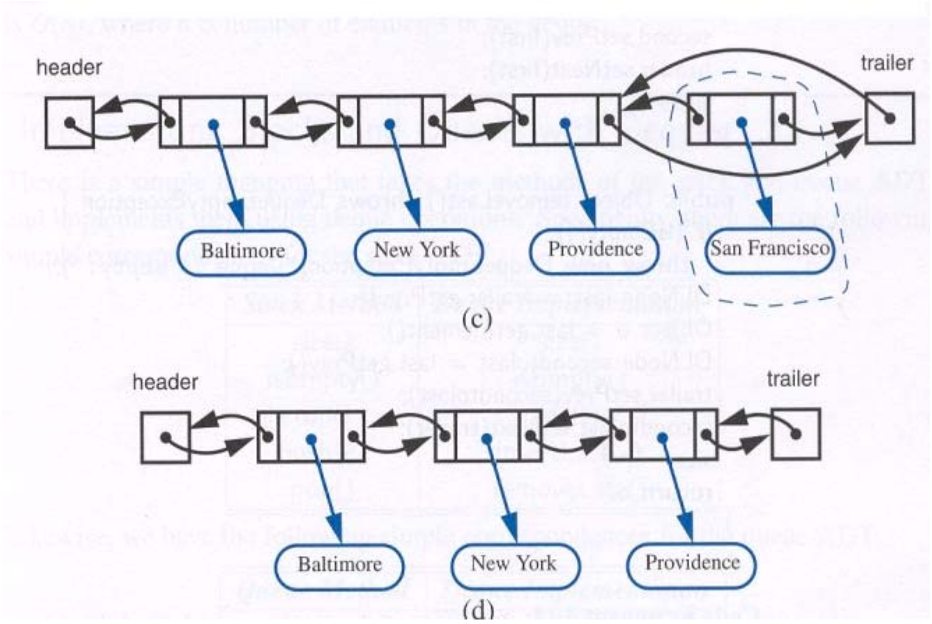


Insertion Algorithm

```
Algorithm insertFirst(e):
    Create a new node v
    v.setElement(e)
    v.setPrev(header)    {link v to header}
    v.setNext(header.getNext())   {link v to head's successor}
    (header.getNext()).setPrev(v) {link head's old successor to v}
    header.setNext(v)   {link head to its new successor, v}
    return null
```

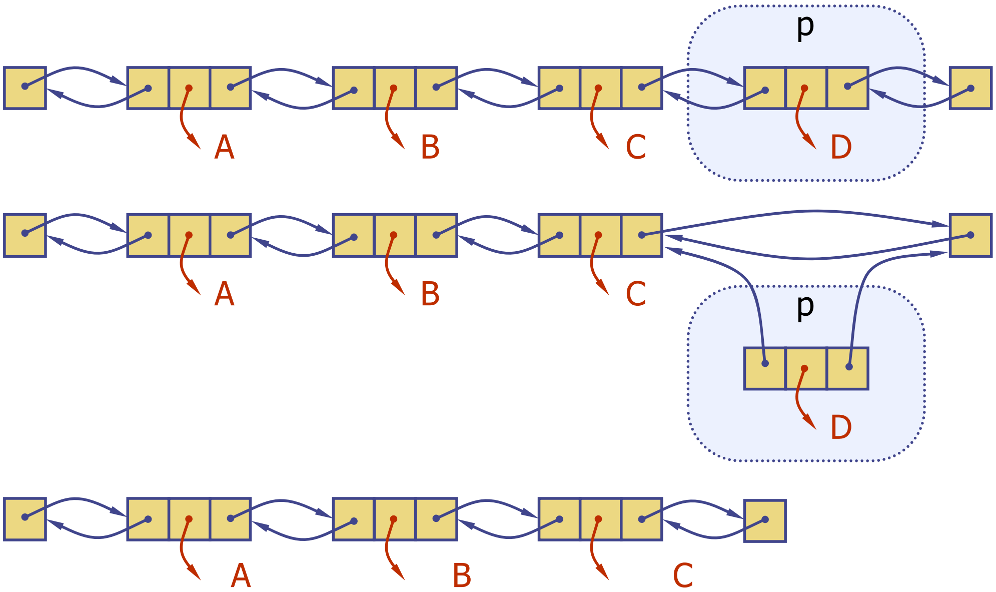
Update Operations (cont.)

- Deletion at the tail



Deletion

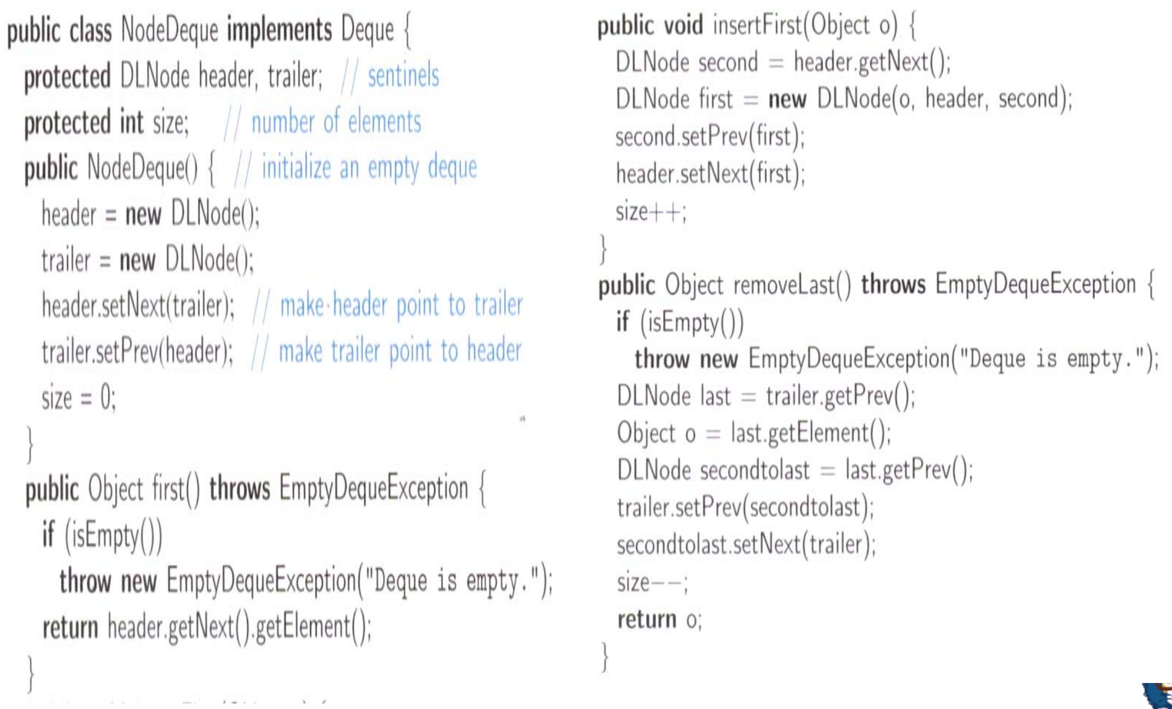
- We visualize remove(p), where p = last()



Deletion Algorithm

```
Algorithm remove(p):
    t = p.element          {a temporary variable to hold the
                            return value}
    (p.getPrev()).setNext(p.getNext())      {linking out p}
    (p.getNext()).setPrev(p.getPrev())
    p.setPrev(null)        {invalidating the position p}
    p.setNext(null)
    return t
```

Implementation of Dequeue with a doubly linked list.



```
public class NodeDeque implements Deque {
  protected DLNode header, trailer;  // sentinels
  protected int size;    // number of elements
  public NodeDeque() {  // initialize an empty deque
    header = new DLNode();
    trailer = new DLNode();
    header.setNext(trailer);  // make header point to trailer
    trailer.setPrev(header);  // make trailer point to header
    size = 0;
  }
  public Object first() throws EmptyDequeException {
    if (isEmpty())
      throw new EmptyDequeException("Deque is empty.");
    return header.getNext().getElement();
  }
```

```
public void insertFirst(Object o) {
  DLNode second = header.getNext();
  DLNode first = new DLNode(o, header, second);
  second.setPrev(first);
  header.setNext(first);
  size++;
}
public Object removeLast() throws EmptyDequeException {
  if (isEmpty())
    throw new EmptyDequeException("Deque is empty.");
  DLNode last = trailer.getPrev();
  Object o = last.getElement();
  DLNode secondtolast = last.getPrev();
  trailer.setPrev(secondtolast);
  secondtolast.setNext(trailer);
  size--;
  return o;
}
```

Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with $\boldsymbol{n}$ elements is $\boldsymbol{O}(\boldsymbol{n})$
  - The space used by each position of the list is $\boldsymbol{O}(1)$
  - All the operations of the List ADT run in $\boldsymbol{O}(1)$ time
  - Operation element() of the Position ADT runs in $\boldsymbol{O}(1)$ time

Q & A

한양대학교 ERICA 캠퍼스