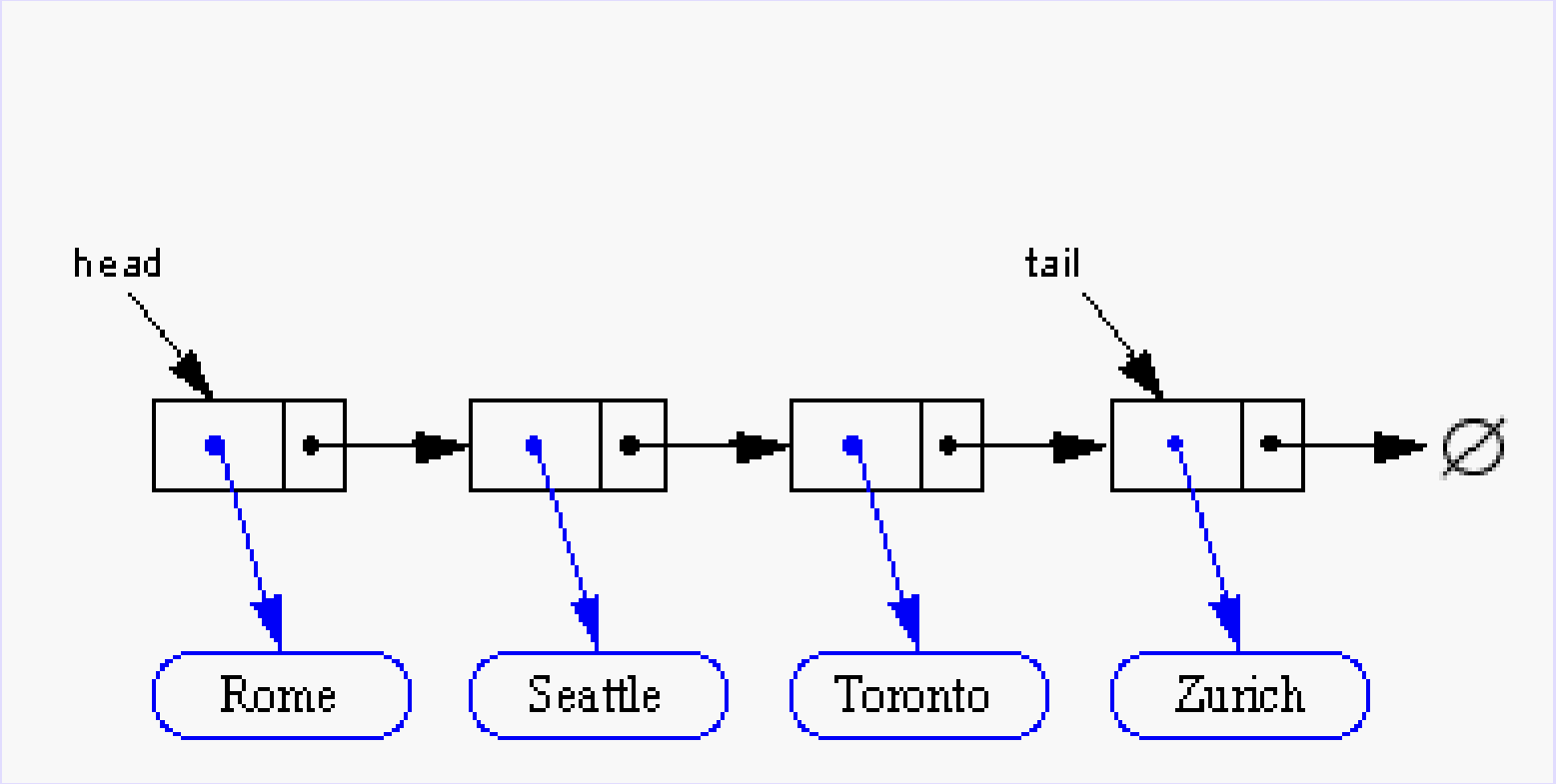
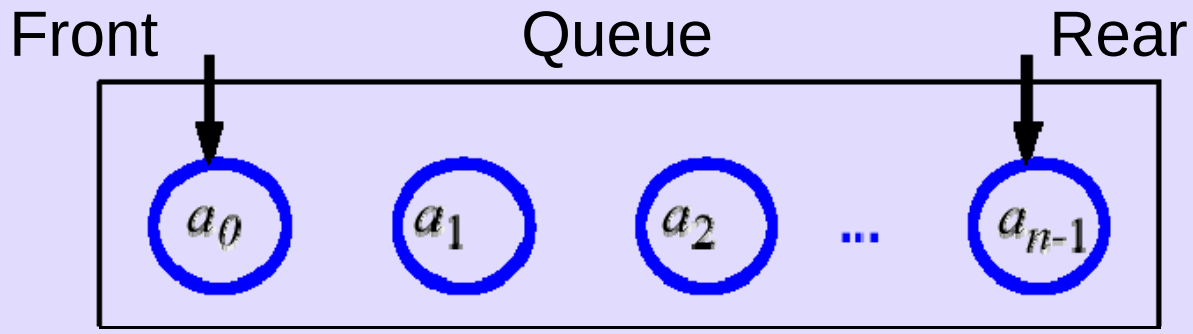
Queues and Linked Lists

Queues
Linked Lists
Double-Ended Queues



Queues

- A queue differs from a stack in that its insertion and removal routines follows the **first-in-first-out** (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)



Queues (2)

- The **queue** supports three fundamental methods:
  - **New():***ADT – Creates an empty queue*
  - **Enqueue(S:***ADT,* **o:***element***):***ADT* - Inserts object *o* at the rear of the queue
  - **Dequeue(S:***ADT***):***ADT* - Removes the object from the front of the queue; an error occurs if the queue is empty
  - **Front(S:***ADT***):***element* - Returns, but does not remove, the front element; an error occurs if the queue is empty

Queues (3)

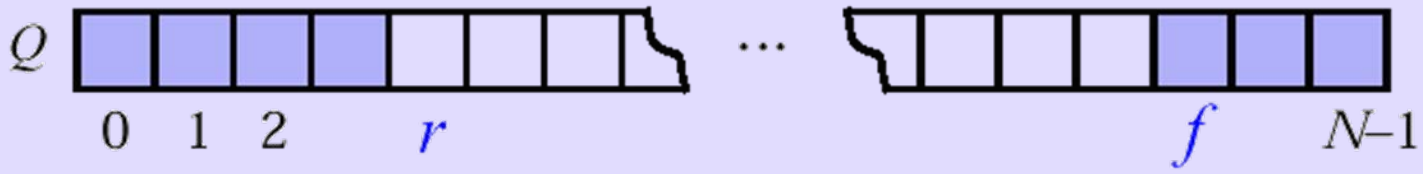
- These support methods should also be defined:
  - **Size(S:*ADT*):*integer***
  - **IsEmpty(S:*ADT*):*boolean***
- Axioms:
  - **Front(Enqueue(New(), v))** = **v**
  - **Dequeque(Enqueue(New(), v)) = New()**
  - **Front(Enqueue(Enqueue(Q, w), v)) = Front(Enqueue(Q, w))**
  - **Dequeue(Enqueue(Enqueue(Q, w), v)) = Enqueue(Dequeue(Enqueue(Q, w)), v)**

An Array Implementation

- Create a queue using an array in a circular fashion
- A maximum size *N* is specified.
- The queue consists of an N-element array *Q* and two integer variables:
  - *f*, index of the front element (head – for dequeue)
  - *r*, index of the element after the rear one (tail – for enqueue)

An Array Implementation (2)

- “wrapped around” configuration



- what does *f*=*r* mean?

An Array Implementation (3)

- Pseudo code

Algorithm **size()**
return *(N-f*+r) **mod** *N*

Algorithm **isEmpty()**
return *(f*=r)

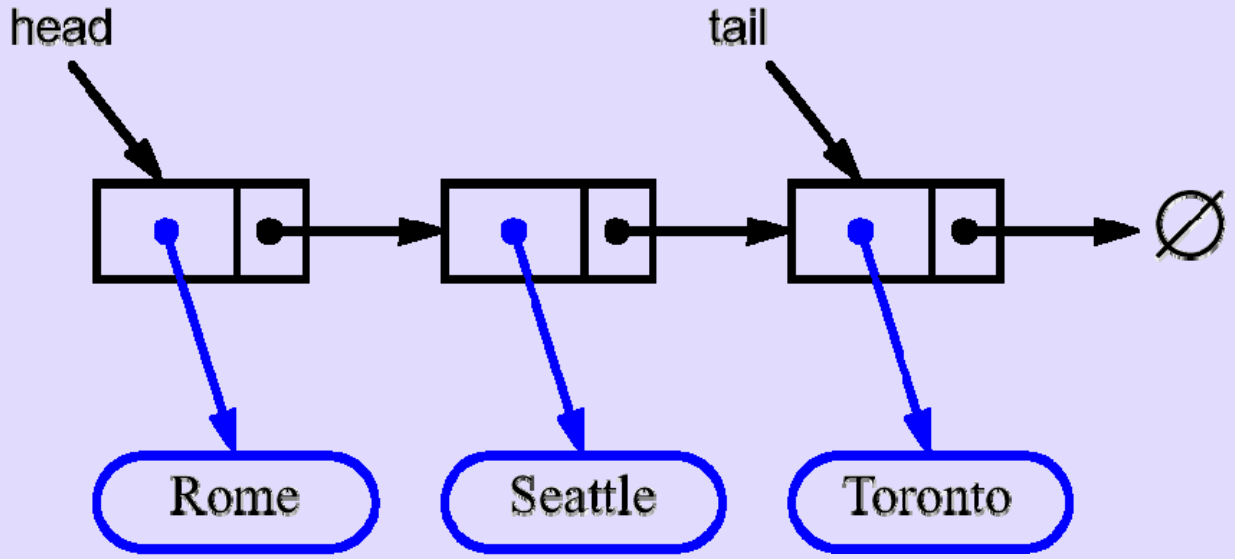
Algorithm **front()**
if isEmpty() **then**
return Queueemptyexception
return Q[f]

Algorithm **dequeue()**
if isEmpty() **then**
return Queueemptyexception
Q[f] ← null
f ←*(f*+1)**mod***N*

Algorithm **enqueue(o)**
if size = *N* - 1 **then**
return Queuefullexception
Q[r] ← *o*
r ← *(r* +1)**mod***N*

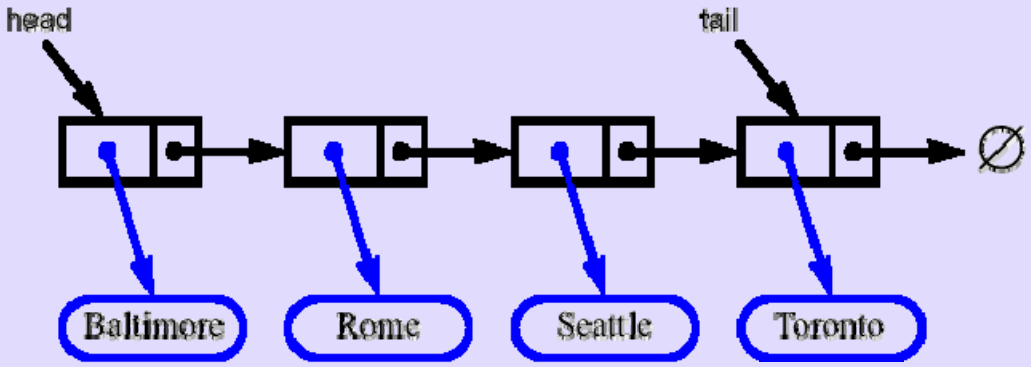
Implementing Queue with Linked List

- Nodes *(data, pointer)* connected in a chain by links

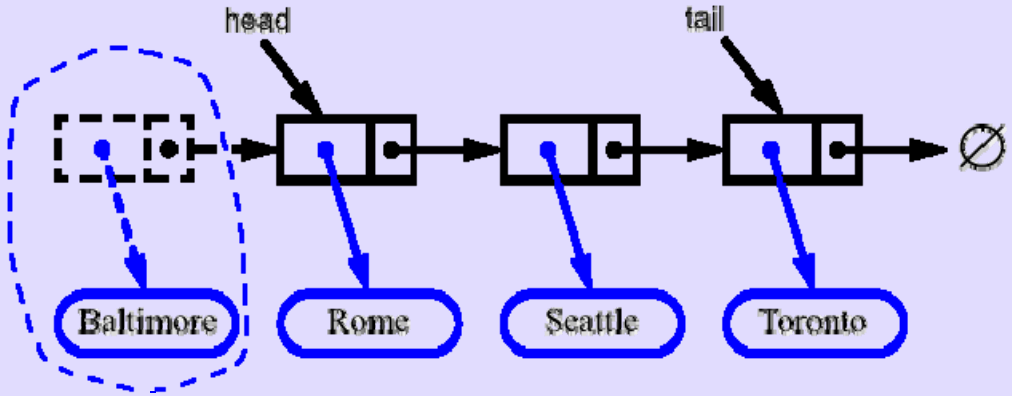


- The head of the list is the front of the queue, the tail of the list is the rear of the queue. ***Why not the opposite?***

Linked List Implementation



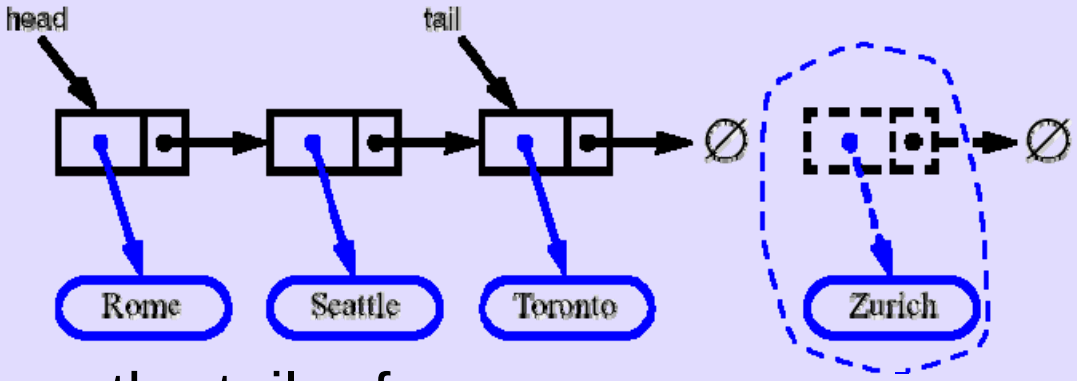
- Dequeue - advance head reference



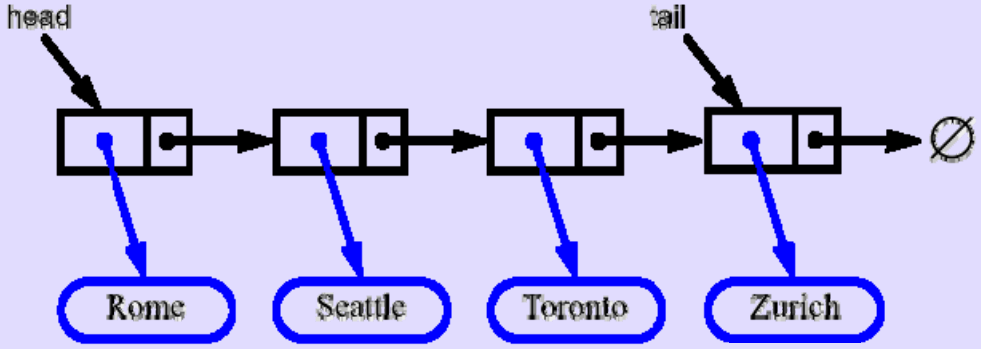
- Inserting at the head is just as easy

Linked List Implementation (2)

- Enqueue - create a new node at the tail



- chain it and move the tail reference



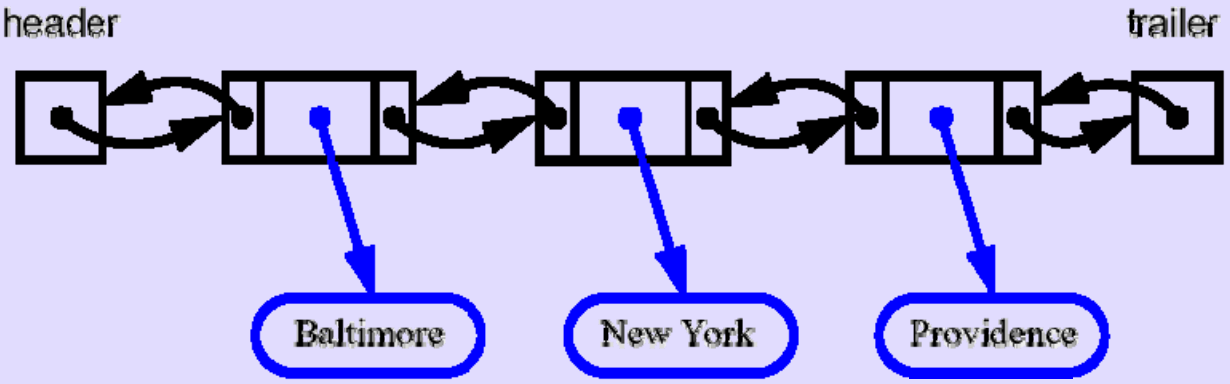
- How about removing at the tail?

Double-Ended Queue

- **A double-ended queue**, or **deque**, supports insertion and deletion from the front and back
- The deque supports six fundamental methods
  - **InsertFirst(S:***ADT,* **o:***element***):***ADT* - Inserts *e* at the beginning of deque
  - **InsertLast(S:***ADT,* **o:***element***):***ADT* - Inserts *e* at end of deque
  - **RemoveFirst(S:***ADT***):***ADT* – Removes the first element
  - **RemoveLast(S:***ADT***):***ADT* – Removes the last element
  - **First(S:***ADT***):***element* and **Last(S:***ADT***):***element* – Returns the first and the last elements

Doubly Linked Lists

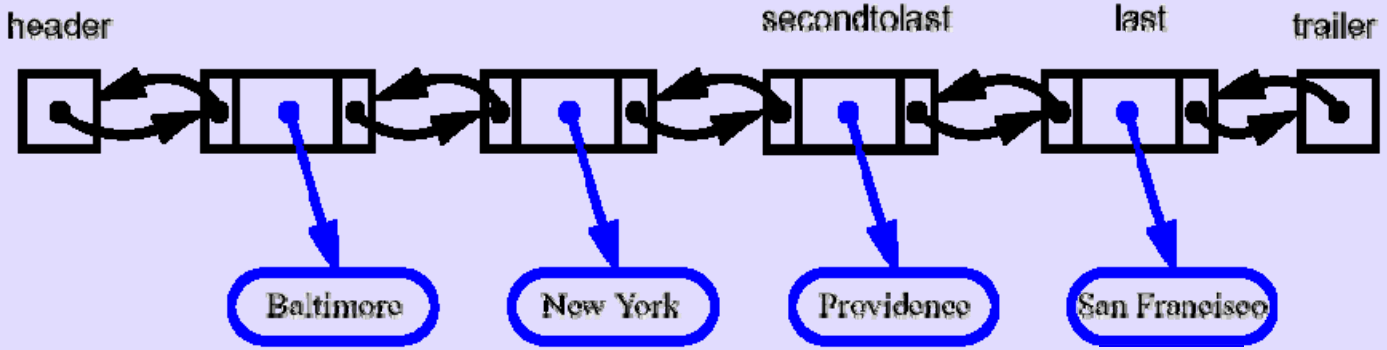
- Deletions at the tail of a singly linked list cannot be done in constant time
- To implement a deque, we use a **doubly linked list**

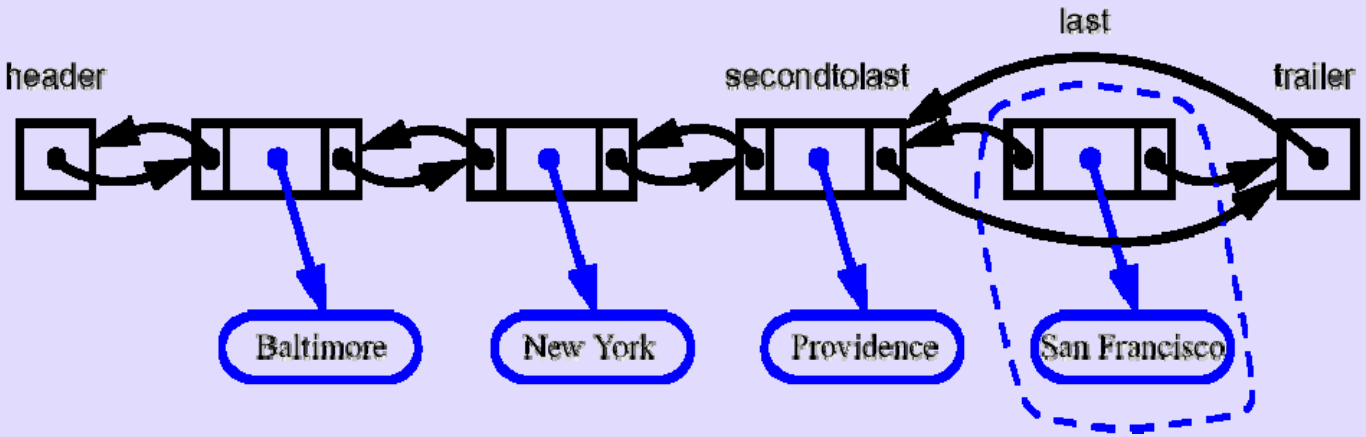


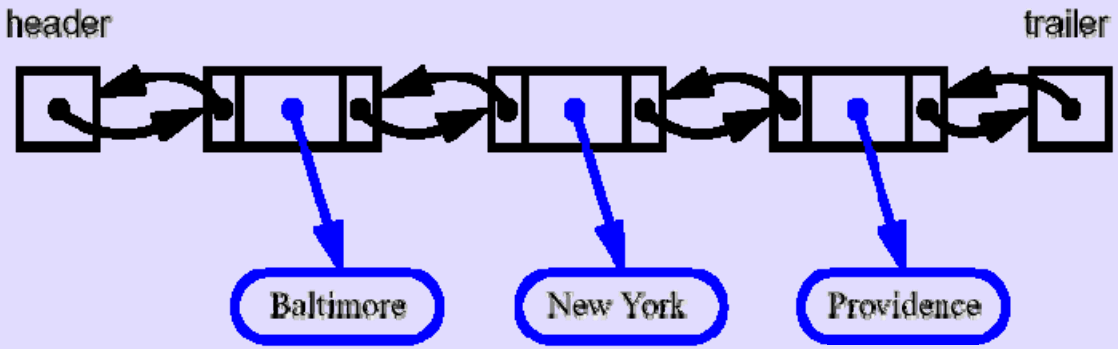
- A node of a doubly linked list has a **next** and a **prev** link
- Then, all the methods of a deque have a constant (that is, O(1)) running time.

Doubly Linked Lists (2)

- When implementing a doubly linked lists, we add two special nodes to the ends of the lists: the header and trailer nodes
  - The header node goes before the first list element. It has a valid next link but a null prev link.
  - The trailer node goes after the last element. It has a valid prev reference but a null next reference.
- The header and trailer nodes are **sentinel** or “dummy” nodes because they do not store elements


header
secondtolast
last
trailer
Baltimore
New York
Providence
San Francisco


last
header
secondtolast
trailer
Baltimore
New York
Providence
San Francisco


header
trailer
Baltimore
New York
Providence

Stacks with Deques

- Implementing ADTs using implementations of other ADTs as building blocks

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(o)	insertLast(o)
pop()	removeLast()

Queues with Deques

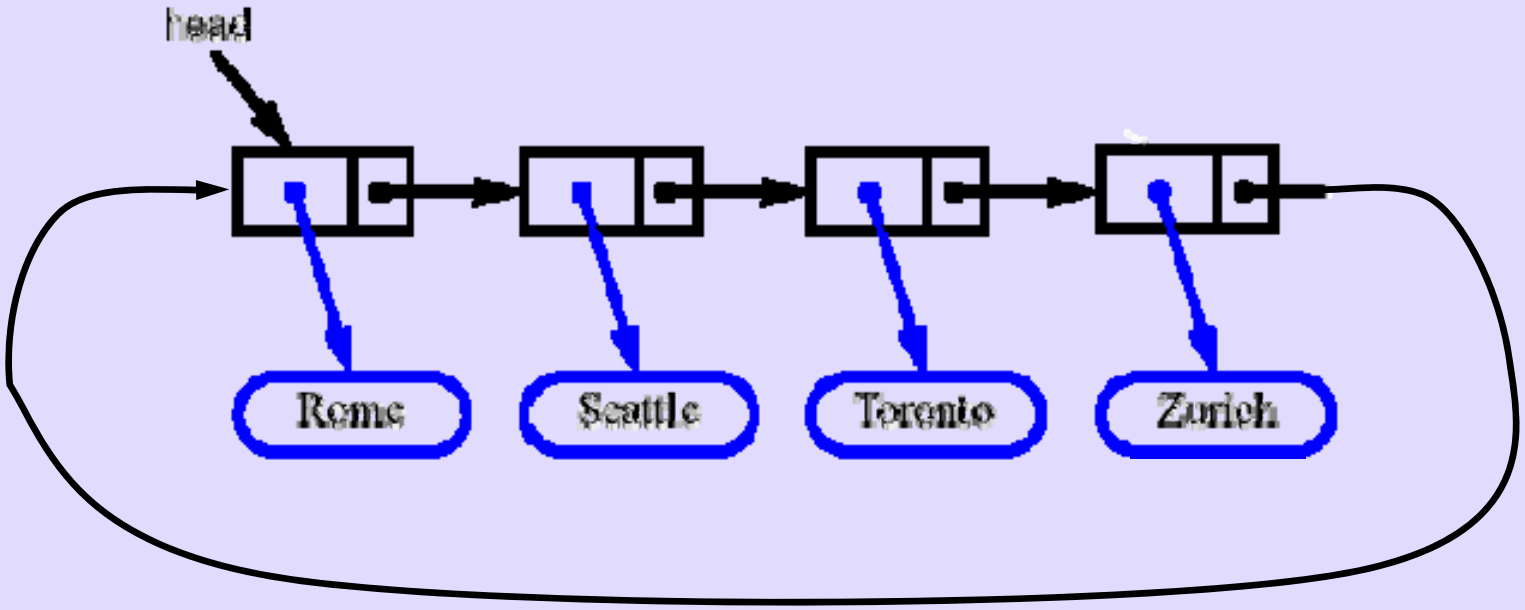
Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	insertLast(o)
dequeue()	removeFirst()

The Adaptor Pattern

- Using a deque to implement a stack or queue is an example of the adaptor pattern. Adaptor patterns implement a class by using methods of another class
- In general, adaptor classes specialize general classes
- Two such applications
  - Specialize a general class by changing some methods eg implementing a stack with a deque.
  - Specialize the types of objects used by a general class eg defining an IntegerArrayStack class that adapts ArrayStack to only store integers.

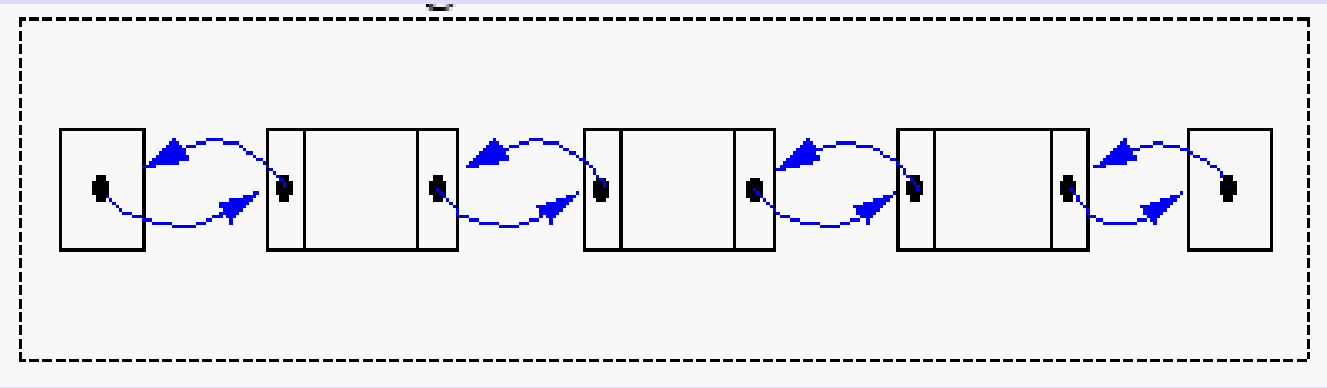
Circular Lists

- No end and no beginning of the list, only one pointer as an **entry point**
- *Circular doubly linked list with a sentinel* is an elegant implementation of a stack or a queue



Sequences

- Vectors
- Positions
- Lists
- General Sequences



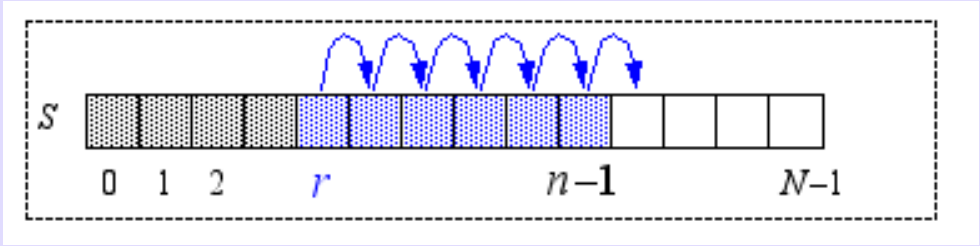
The Vector ADT

A sequence S (with n elements) that supports the following methods:

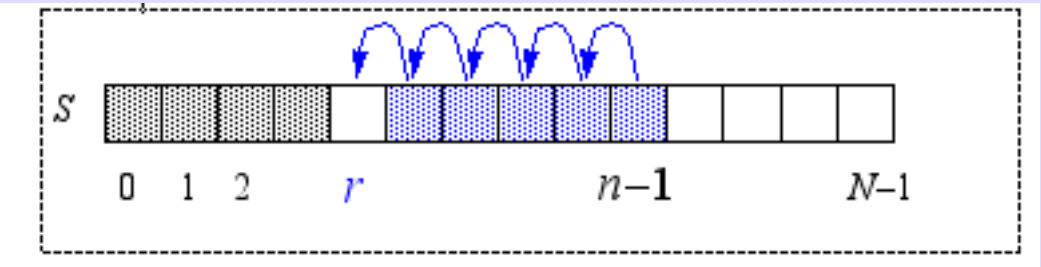
- elemAtRank(r): Return the element of S with rank r; an error occurs if r < 0 or r > n -1
- replaceAtRank(r,e): Replace the element at rank r with e and return the old element; an error condition occurs if r < 0 or r > n - 1
- insertAtRank(r,e): Insert a new element into S which will have rank r; an error occurs if r< 0 or r > n
- removeAtRank(r): Remove from S the element at rank r; an error occurs if r < 0 or r > n - 1

Array-Based Implementation

Algorithm insertAtRank(r,e):
for i = n – 1 downto r do
    S[i+1] ← S[i]
S[r] ← e
n ← n + 1



Algorithm removeAtRank(r):
e ← S[r]
for i = r to n - 2 do
    S[i] ← S[i + 1]
n ← n - 1
return



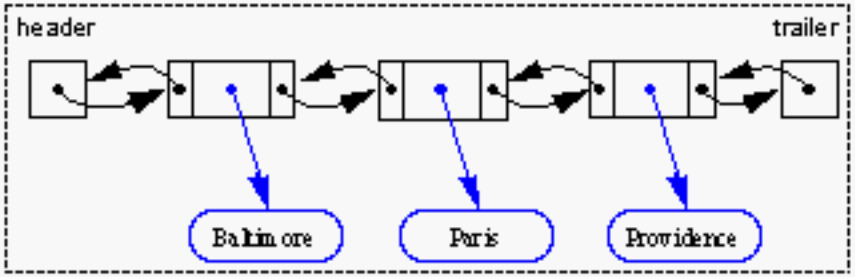
Array-Based Implementation (contd.)

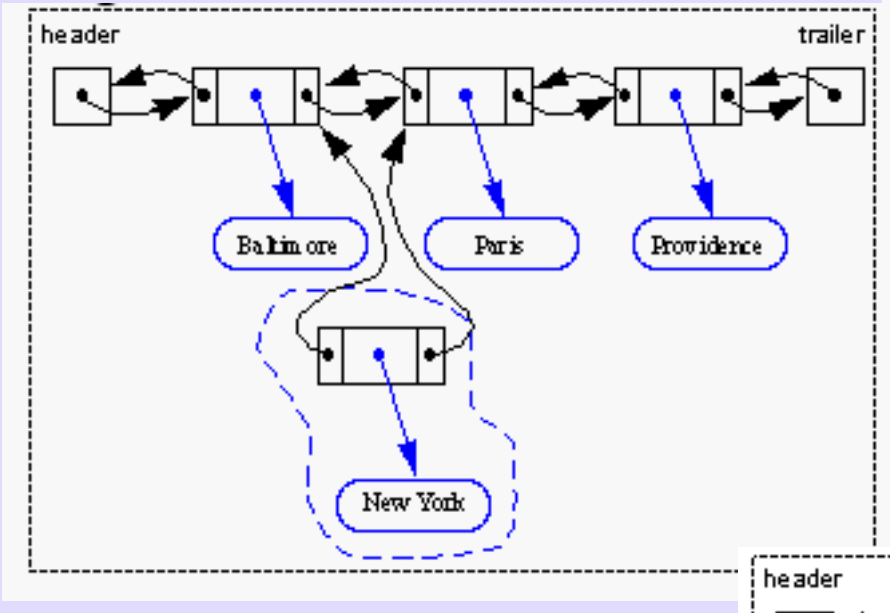
Time complexity of the various methods:

Method	Time
size	$O(1)$
isEmpty	$O(1)$
elemAtRank	$O(1)$
replaceAtRank	$O(1)$
insertAtRank	$O(n)$
removeAtRank	$O(n)$

Implem. with a Doubly Linked List

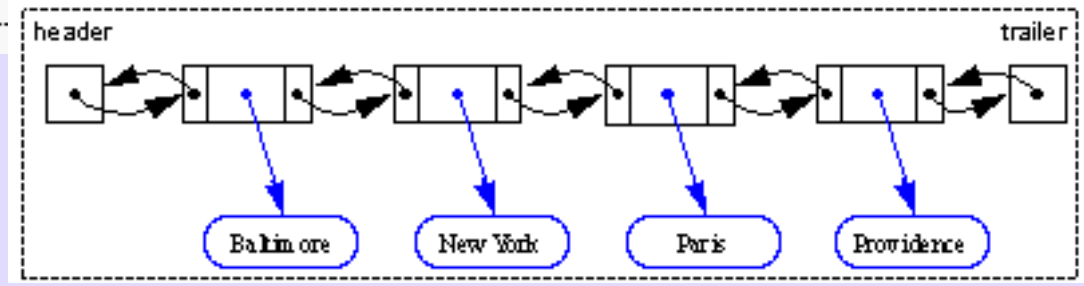
the list before insertion





creating a new node for insertion:

the list after insertion

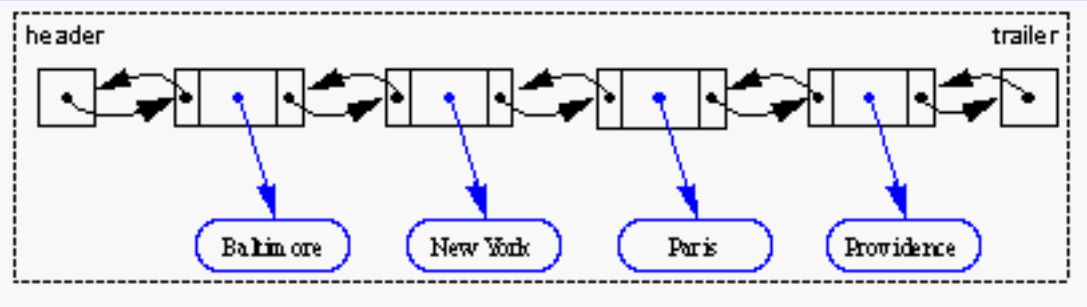


Java Implementation

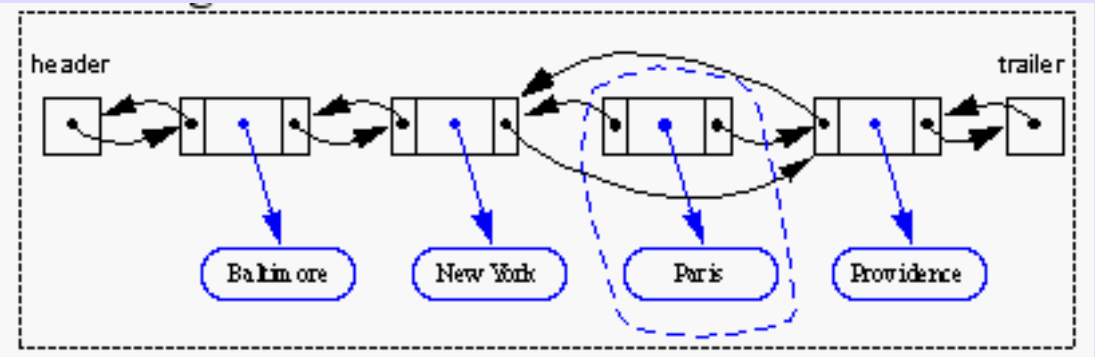
```
public void insertAtRank (int rank, Object element)
    throws BoundaryViolationException {
if (rank < 0 || rank > size())
    throw new BoundaryViolationException("invalid rank");
DLNode next = nodeAtRank(rank);
                        // the new node will be right before this
DLNode prev = next.getPrev();
                          // the new node will be right after this
DLNode node = new DLNode(element, prev, next);
                      // new node knows about its next & prev.
                  // Now we tell next & prev about the new node.
next.setPrev(node);
prev.setNext(node);
size++;
}
```

Implementation with a Doubly Linked List

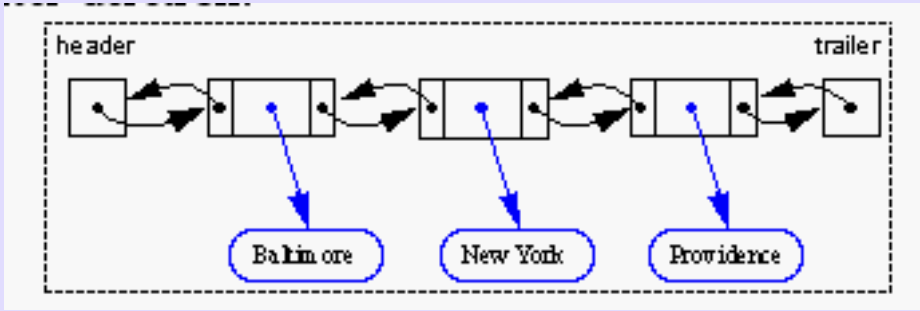
the list before deletion



deleting a node



after deletion



Java Implementation

```
public Object removeAtRank (int rank)
          throws BoundaryViolationException {
if (rank < 0 || rank > size()-1)
      throw new BoundaryViolationException("Invalid rank.");
DLNode node = nodeAtRank(rank);   // node to be removed
DLNode next = node.getNext();                  // node before it
DLNode prev = node.getPrev();                    // node after it
prev.setNext(next);
next.setPrev(prev);
size--;
return node.getElement();
                        // returns the element of the deleted node
 }
```

Java Implementation (contd.)

```
private DLNode nodeAtRank (int rank) {
      //auxiliary method to find node of element with given rank
DLNode node;
if (rank <= size()/2) {                    //scan forward from head
   node = header.getNext();
   for (int i=0; i < rank; i++)
        node = node.getNext();
} else {                                   //scan backward from the tail
         node = trailer.getPrev();
         for (int i=0; i < size()-rank-1 ; i++)
               node = node.getPrev();
}
return node;
}
```

Nodes

- Linked lists support the efficient execution of ***node-based operations:***
- removeAtNode(Node v) and insertAfterNode(Node v, Object e), would be O(1).
- However, node-based operations are not meaningful in an array-based implementation because there are no nodes in an array.
- Nodes are implementation-specific.
- **Dilemma:**
  - If we do not define node based operations, we are not taking full advantage of doubly-linked lists.
  - If we do define them, we violate the generality of ADTs.

From Nodes to Positions

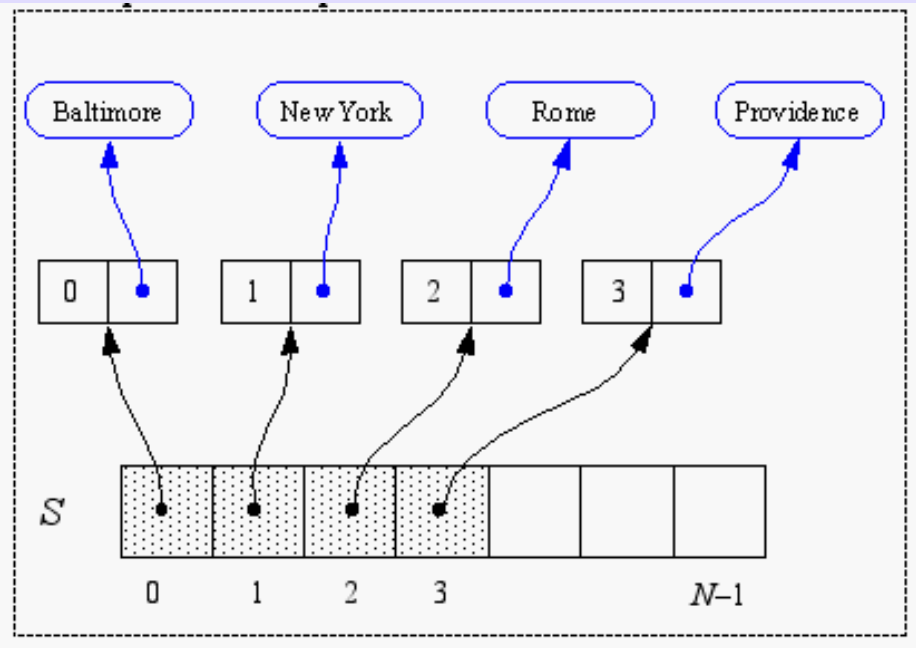
- We introduce the ***Position*** ADT
- Intuitive notion of “place” of an element.
- Positions have only one method: element(): Returns the element at this position
- Positions are defined relative to other positions (before/after relation)
- Positions are not tied to an element or rank

The List ADT

- ADT with position-based methods
- generic methods: size(), isEmpty()
- query methods: isFirst(p), isLast(p)
- accessor methods: first(), last(), before(p), after(p)
- update methods
  - swapElements(p,q), replaceElement(p,e)
  - insertFirst(e), insertLast(e)
  - insertBefore(p,e), insertAfter(p,e)
  - remove(p)
- each method takes O(1) time if implemented with a doubly linked list

The Sequence ADT

- Combines the Vector and List ADT (multiple inheritance)
- Adds methods that bridge between ranks and positions
  - atRank(r) returns a position
  - rankOf(p) returns an integer rank



- An array-based implementation needs to use objects to represent the positions

Comparison of Sequence Implementations

Operations	Array	List
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last	$O(1)$	$O(1)$
before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Iterators

- Abstraction of the process of scanning through a collection of elements
- Encapsulates the notions of “place” and “next”
- Extends the position ADT
- Generic and specialized iterators
- ***ObjectIterator***: hasNext(), nextObject(), object()
- ***PositionIterator:*** nextPosition()
- Useful methods that return iterators: elements(), positions()