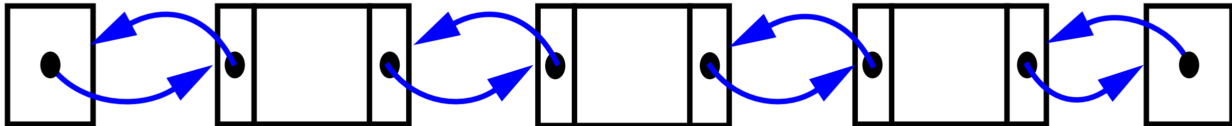
SEQUENCES

- Vectors
- Positions
- Lists
- General Sequences
- Bubble Sort Algorithm



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 $r > n - 1$

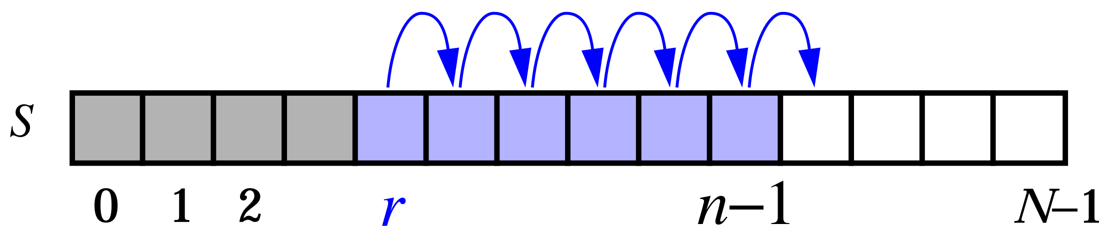
  - insertAtRank(r,e):
    Insert a new element into S which will have rank r; an error occurs if $r < 0$ or $r > n - 1$

  - removeAtRank(r):
    Remove from S the element at rank r; an error occurs if $r < 0$ or $r > n - 1$

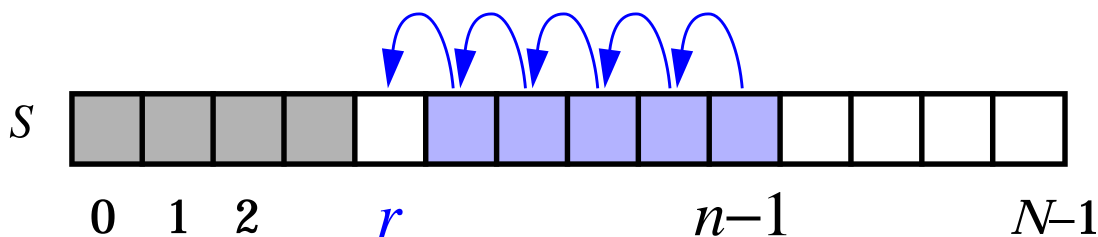
Array-Based Implementation

- Some Pseudo-Code:

Algorithm insertAtRank(r,e):
  for $i = n - 1, n - 2, \ldots, r$ **do**
    $S[i+1] \leftarrow s[i]$
  $S[r] \leftarrow e$
  $n \leftarrow n + 1$



Algorithm removeAtRank(r):
  $e \leftarrow S[r]$
  for $i = r, r + 1, \ldots, n - 2$ **do**
    $S[i] \leftarrow S[i + 1]$
  $n \leftarrow n - 1$
  return



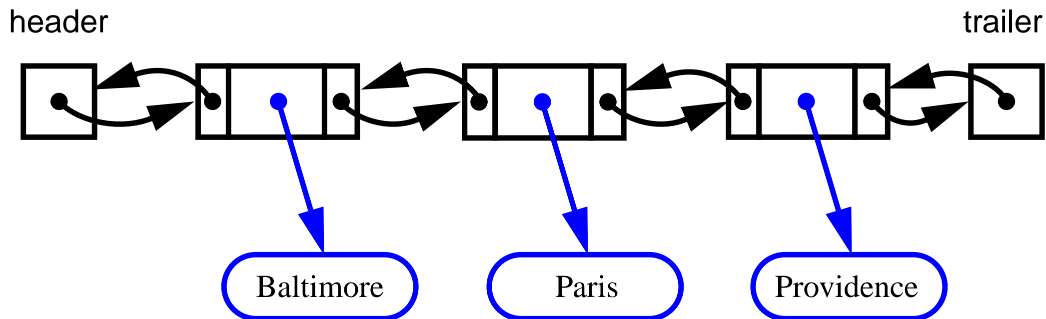
Array-Based Implementation (contd.)

- Time complexity of the various methods:

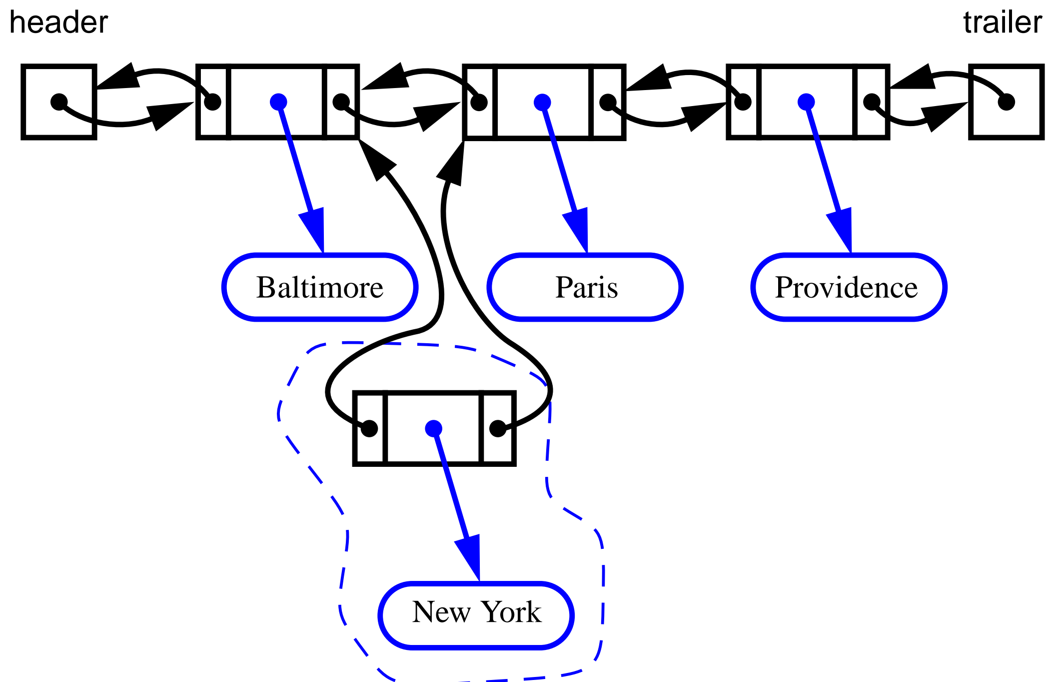
Method	Time
size	$O(1)$
isEmpty	$O(1)$
elemAtRank	$O(1)$
replaceAtRank	$O(1)$
insertAtRank	$O(n)$
removeAtRank	$O(n)$

Implementation with a Doubly Linked List

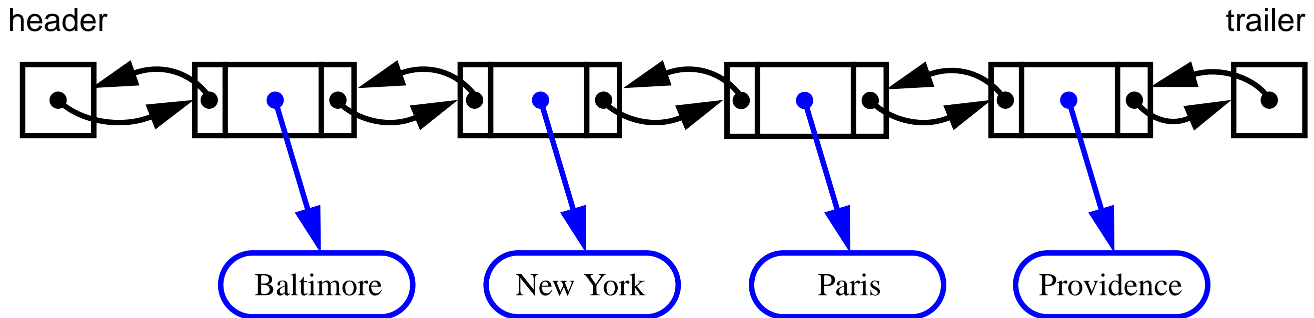
- the list before insertion:



- creating a new node for insertion:



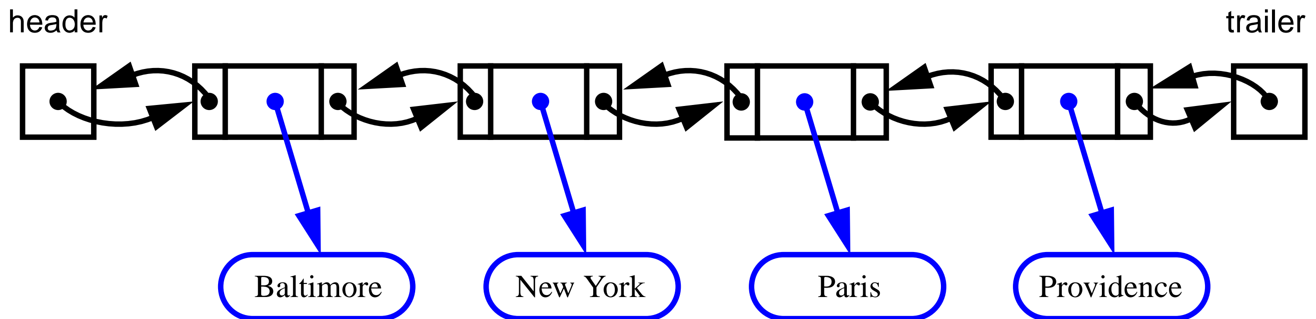
- the list after insertion:



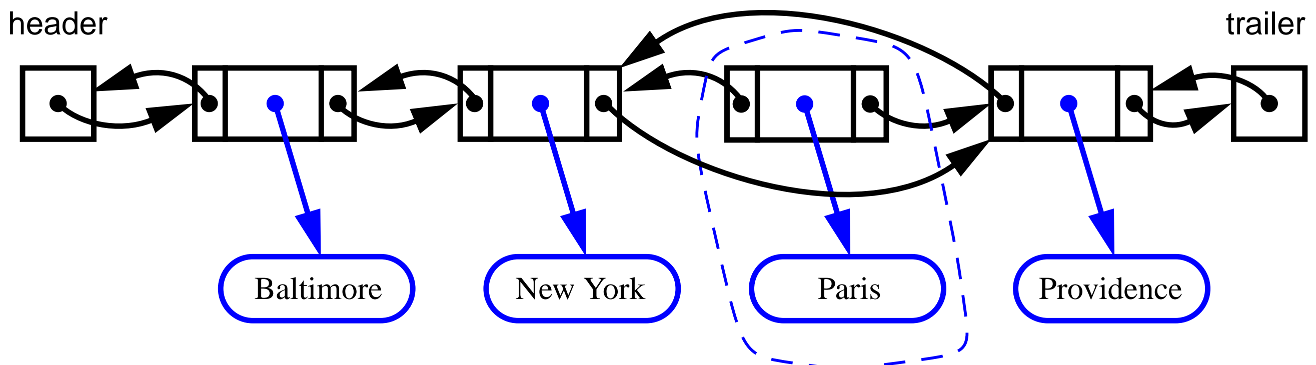

```
public void insertAtRank (int rank, Object element)
  throws BoundaryViolationException {
    if (rank < 0 || rank > size())
      throw new BoundaryViolationException(“invalid rank”);
    DLNode next = nodeAtRank(rank); // the new node
                          //will be right before this
    DLNode 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

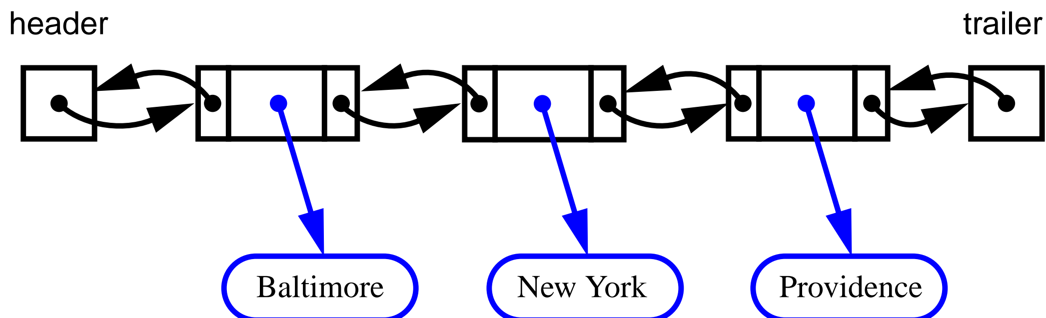
- the list before deletion:



- deleting a node:



- after deletion:



Java Implementation

- code for deletion of a node

```
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 (cont.)

- code for finding a node at a certain rank

```
private DLNode nodeAtRank (int rank) {
  // auxiliary method to find the node of the
  // element with the given rank. We make
  // auxiliary methods private or protected.
  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

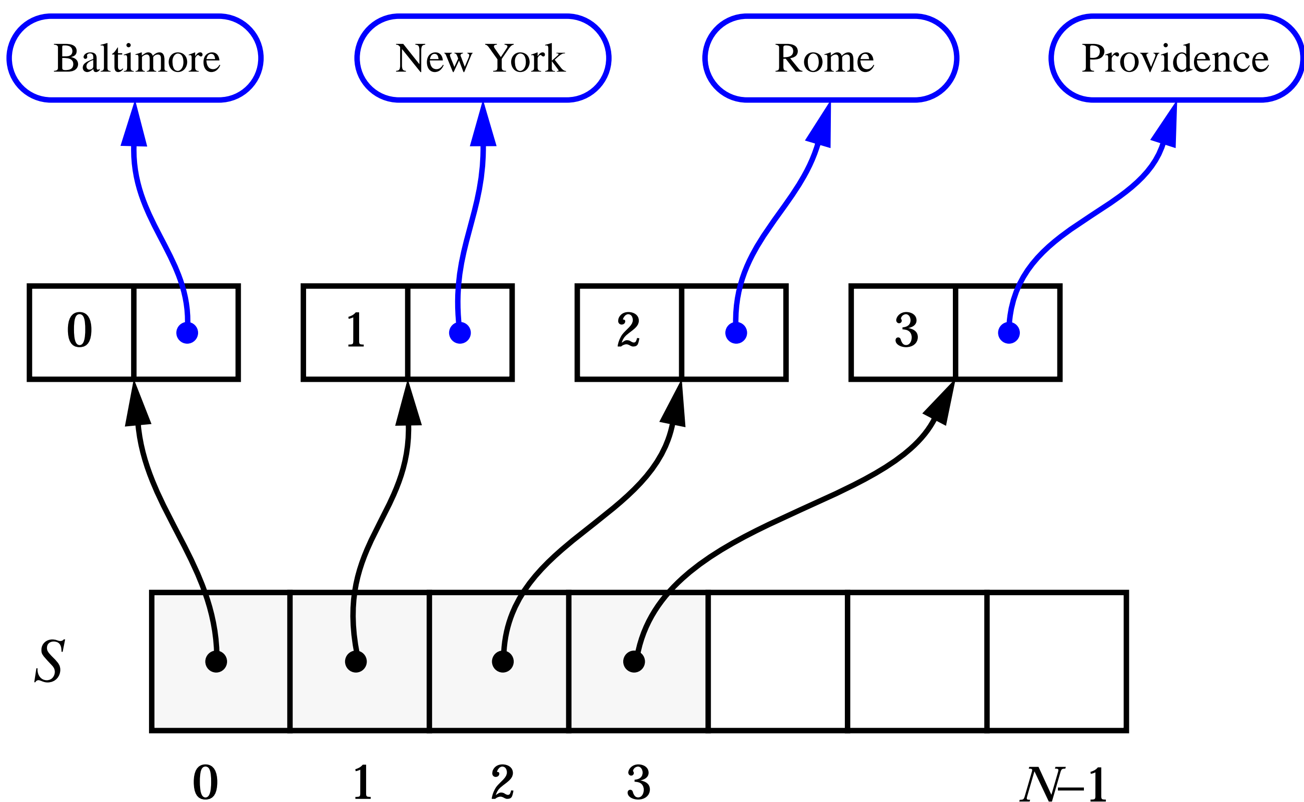
- We introduce the ***Position*** ADT
- Initiutve notion of “place” of an element.
- Positions have only one method:
  element(): Return the element at this position
- Positions are defined relatively to other positions (before/after relation)
- Positions are not tied to an element or rank

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()