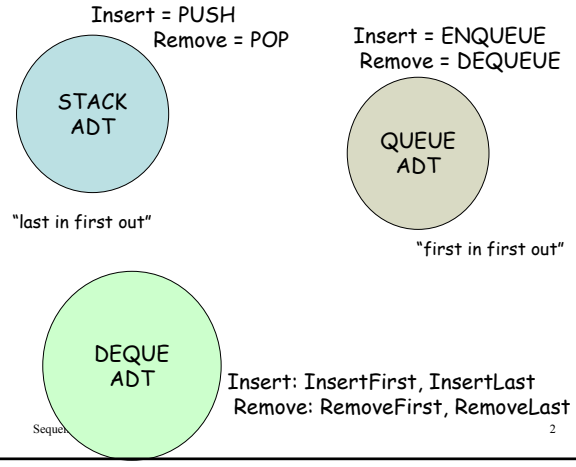
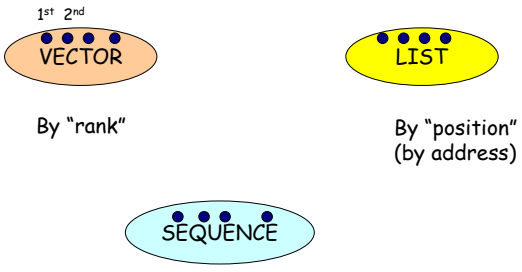
Vectors, Lists, and Sequences



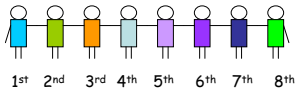
Vector, List, and Sequence ADTs are collections of linearly arranged elements and provide methods for accessing, inserting, and removing arbitrary elements.



Vectors ADT

- Can access any element directly, not just first or last.
- Elements are accessed by rank, the number of elements which precede them.

VECTOR



Questions like:

Who is 5th ?

The Vector ADT

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

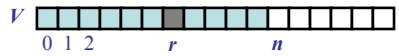
-elemAtRank(r):	Return the element of S with rank r; an error occurs if r < 0 or r > n -1
-replaceAtRank(r,e):	Replace the element at rank r with e and return the old element; an error condition occurs if r < 0 or r > n - 1
-insertAtRank(r,e):	Insert a new element into S which will have rank r; an error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

Applications of Vectors

- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Natural Implementation: with an Array

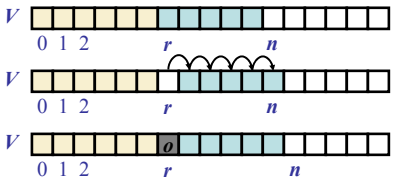
- Array ***V*** of size ***N***
- A variable ***n*** keeps track of the size of the vector (number of elements stored)
- Operation ***elemAtRank***(***r***) is implemented in $O(1)$ time by returning ***V***[***r***]



Insertion

- In operation ***insertAtRank***(***r***, ***o***), we need to make room for the new element by shifting forward the ***n*** - ***r*** elements ***V***[***r***], …, ***V***[***n*** - 1]
- In the worst case (***r*** = 0), this takes $O(n)$ time

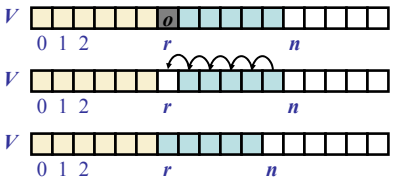
```
insertAtRank(r,o):
    for i = n - 1, n - 2, ... , r do
        S[i+1] ← s[i]
    S[r] ← o
    n ← n + 1
```



Deletion

- In operation ***removeAtRank***(***r***), we need to fill the hole left by the removed element by shifting backward the ***n*** - ***r*** - 1 elements ***V***[***r*** + 1], …, ***V***[***n*** - 1]
- In the worst case (***r*** = 0), this takes $O(n)$ time

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



Performance

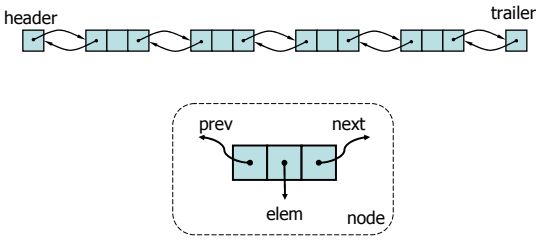
- In the array based implementation of a Vector
  - The space used by the data structure is $O(n)$
  - ***size***, ***isEmpty***, ***elemAtRank*** and ***replaceAtRank*** run in $O(1)$ time
  - ***insertAtRank*** and ***removeAtRank*** run in $O(n)$ time
- In an ***insertAtRank*** operation, when the array is full, instead of having an ERROR, we can replace the array with a larger one

Performance (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



Finding an element at a certain rank

```
Algorithm nodeAtRank (rank)
        if (rank <= size()/2) { //scan forward from head
                                node ← header.next
                                for (int i=0; i < rank; i++)
                                node ← node.next
    }else { // scan backward from the tail
                                node ← trailer.prev
                                for (int i=0; i < size()-rank-1 ; i++)
                                node ← node.prev
    }
    return node;
```

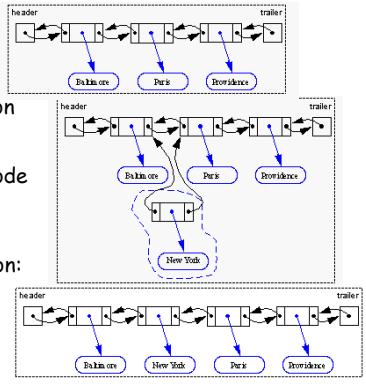

Inserting at a certain rank

insertAtRank

the list before insertion

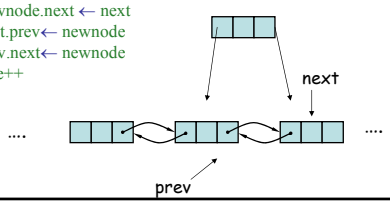
creating a new node for insertion:

the list after insertion:



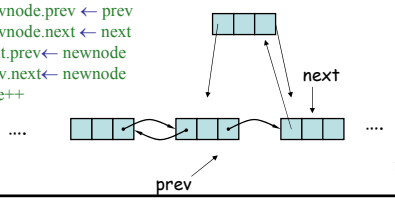
newnode: node to insert.

```
Algorithm insertAtRank (rank,element)
                if (rank < 0 or rank > size())
                        ERROR
                next ← nodeAtRank(rank) // the new node
                                        //will be right before this
                prev ← next.prev  // the new node will be right after this

                newnode.prev ← prev
                newnode.next ← next
                next.prev← newnode
                prev.next← newnode
                size++
```



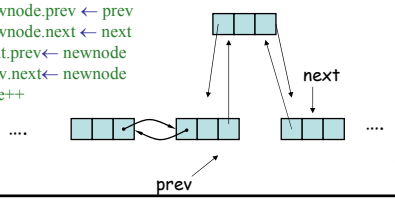
newnode: node to insert.

```
Algorithm insertAtRank (rank,element)
        if (rank < 0 or rank > size())
                ERROR
        next ← nodeAtRank(rank) // the new node
                                //will be right before this
        prev ← next.prev  // the new node will be right after this

        newnode.prev ← prev
        newnode.next ← next
        next.prev← newnode
        prev.next← newnode
        size++
```



newnode: node to insert.

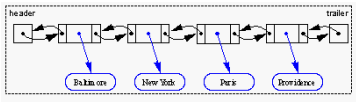
```
Algorithm insertAtRank (rank,element)
        if (rank < 0 or rank > size())
                ERROR
        next ← nodeAtRank(rank) // the new node
                                //will be right before this
        prev ← next.prev  // the new node will be right after this

        newnode.prev ← prev
        newnode.next ← next
        next.prev← newnode
        prev.next← newnode
        size++
```



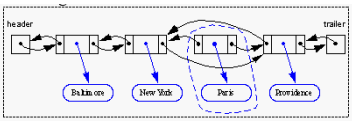
Removing at a certain rank

removeAtRank

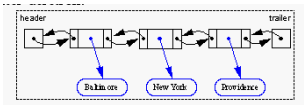
the list before deletion:



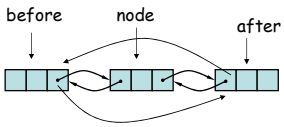
deleting a node



after deletion:




```
Algorithm removeAtRank (rank)
        if (rank < 0 or rank > size()-1)
                ERROR
        node ← nodeAtRank(rank) // node to be removed
        after ← node.next       // node after it
        before ← node.prev      // node before it
        before.next ← next
        after.prev ← prev
        size--
        return node.element  // returns the element of the deleted node
```



Performance

- ***elemAtRank*** , ***insertAtRank, removeAtRank, and replaceAtRank*** run in *O*(n) time

- ***size***, ***isEmpty***, ***: O(1)***

Lists ADT

Container of elements that store each element at a position and that keeps these positions arranged in a linear order

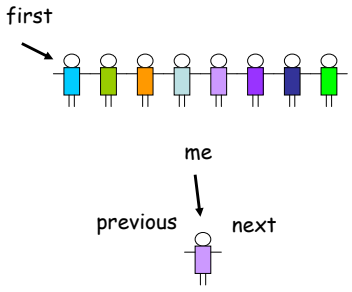
Cannot access any element directly, can access just first or last.

(address)

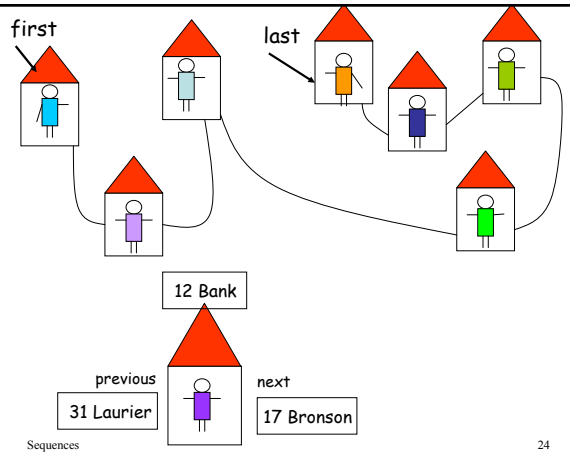
Elements are accessed by position. (place)

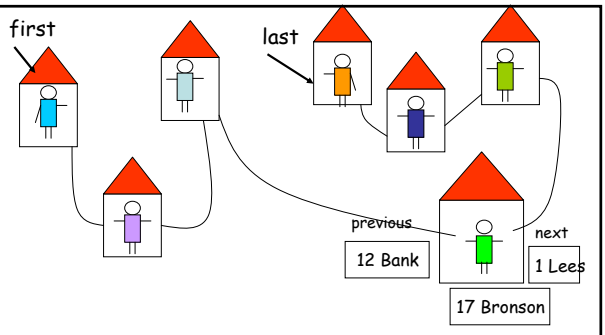
Positions are defined relatively to other positions

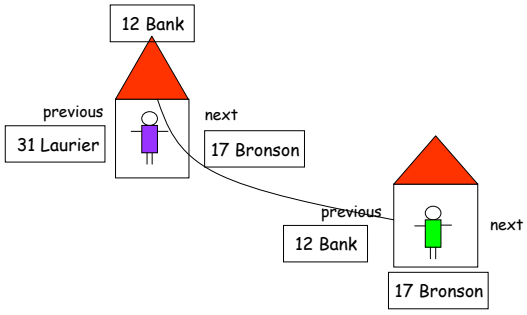
(before/after relation)

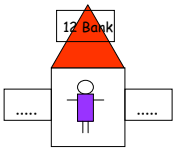


I don't know my "rank" - there is no notion of rank.
I only know who is next and who is before









If I know an address, questions like:

Who is after 12 Bank ?

Who is first ?
Who is after first ?

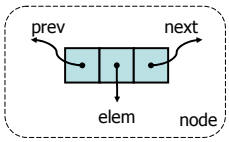
The List ADT

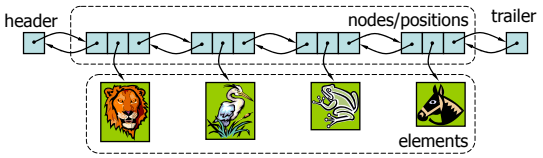
ADT with position-based methods

- 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)

Natural Implementation: with a Linked List

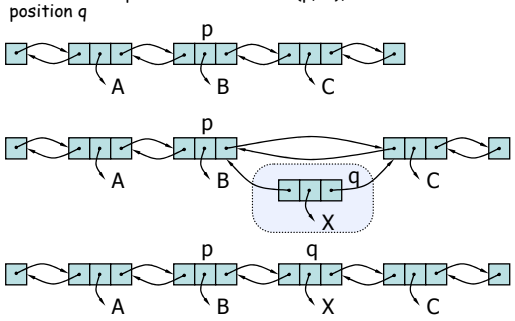
- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





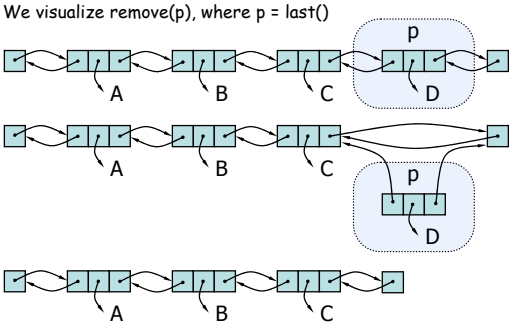
Insertion

- We visualize operation insertAfter(p, X), which returns position q



Deletion

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



Performance

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

A more general ADT: The Sequence

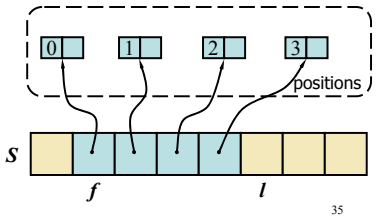
- Combines the Vector and List ADT
- Adds methods that bridge between ranks and positions (i.e. provides access to its elements using both ranks and positions)
  - atRank(r) returns a position
  - rankOf(p) returns an integer rank

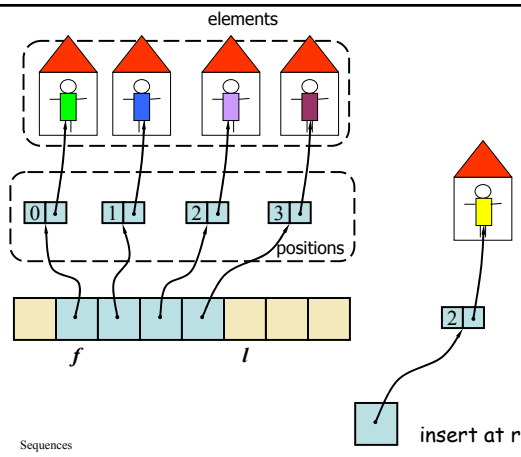
Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

Array-based Implementation

- Circular array storing positions
- A position object stores:
  - Element
  - Rank
- Indices *f* and *l* keep track of first and last positions



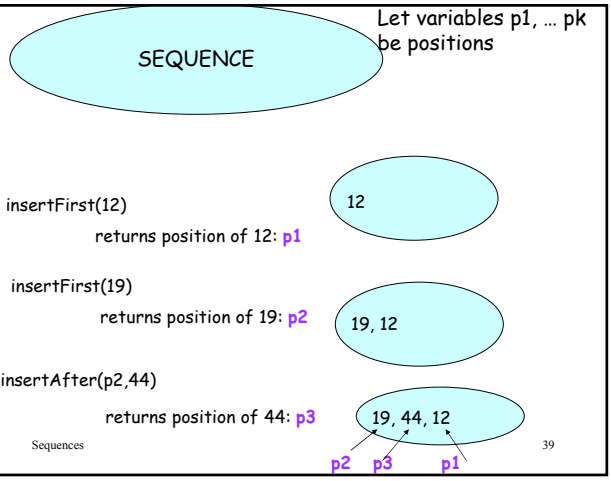


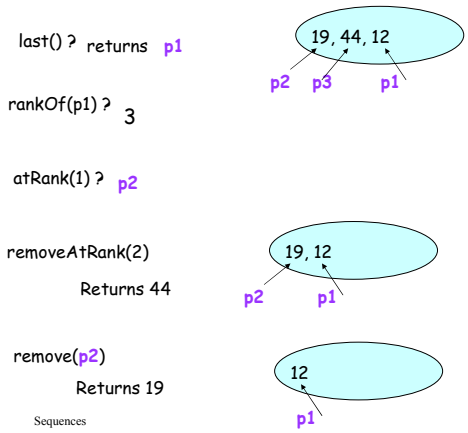
Implementation with Doubly Linked List

- Position methods are O(1)
- Rank methods require scanning through the list: O(n)

Sequence Implementations

Operation	Array	List
size, isEmpty	O(1)	O(1)
atRank, rankOf, elemAtRank	O(1)	***O(n)***
first, last, 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)





Traversing a Sequence (march through the elements) depends on the actual implementation ….

Accessing the "next element" in a Sequence depends on the actual implementation ….

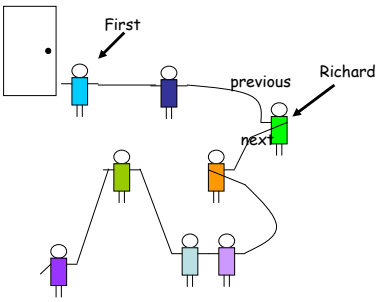
We would like to have a general way of doing this ….

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()
  next()
- ***PositionIterator*** elements()
  positions()

LIST



If I know Richard, questions like

Who is after Richard ?

SEQUENCE