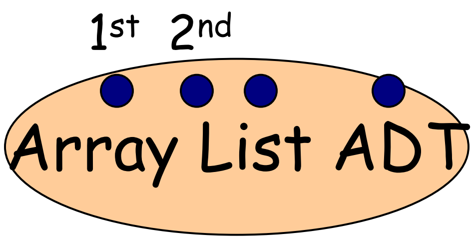
Lists

- Array-List ADT
  (also study extendible arrays)
- Positional-List ADT
- Sequence ADT

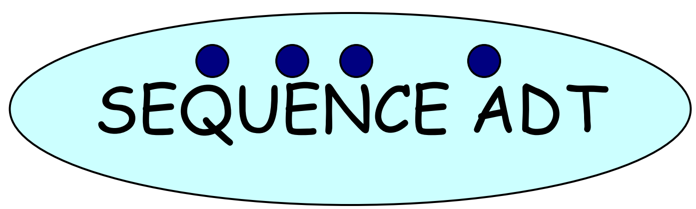
Lists or Sequences

LISTS or SEQUENCES= collection of elements in linear order



To be implemented by arrays. Access by "index"

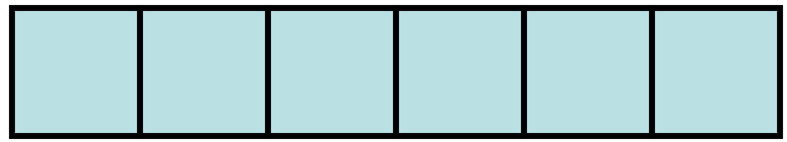
To be implemented by linked lists Access by "position" (or address)



Combination of both

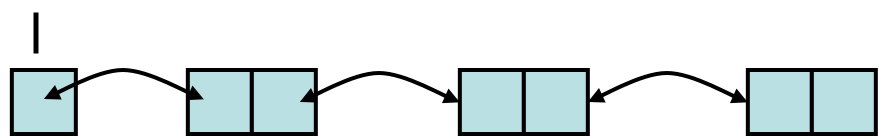
Review:
Basic Data Structures ("concrete" data structures)

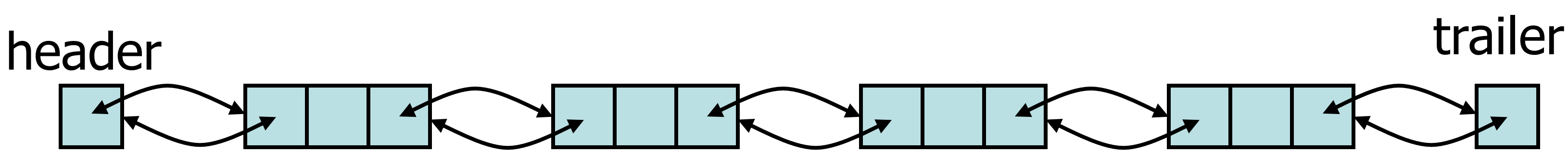
Array



Linked Lists

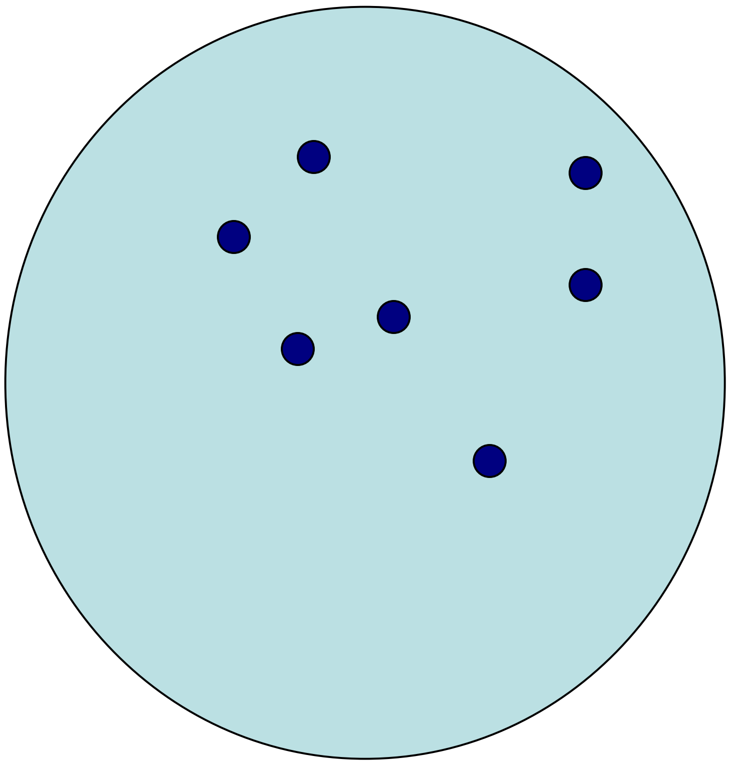
For example:





Abstract Data Types (ADT)

ADT is an abstraction of a data structure.
ADTs specify what can be stored and what operations can be performed.



Containers

Contains objects

I can INSERT

I can REMOVE

I can .....

Abstract Data Types seen so far

Insert = PUSH

Remove = POP

STACK

"last in first out"

Insert = ENQUEUE

Remove = DEQUEUE

QUEUE

"first in first out"

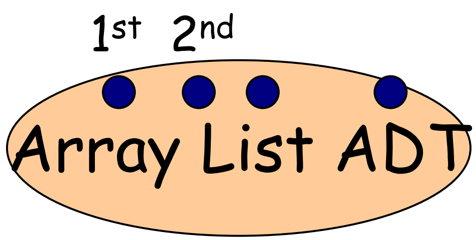
DEQUE

Insert: InsertFirst, InsertLast

Remove: RemoveFirst, RemoveLast

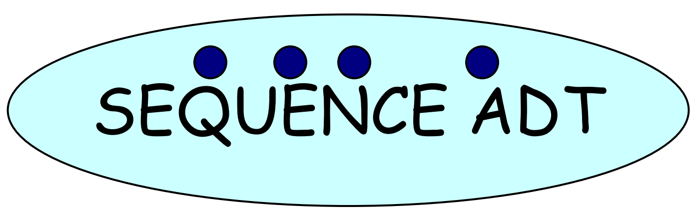
Lists or Sequences

LISTS or SEQUENCES= collection of elements in linear order



To be implemented by arrays. Access by "index"

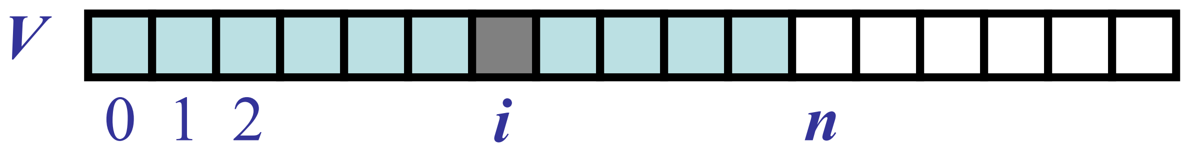
To be implemented by linked lists Access by "position" (or address)



Combination of both

Array-lists

- Can access any element directly, not just first or last.
- Elements are accessed by index (or rank), the number of elements which precede them (if starting from index 0).
- Typically implemented by an array



The Array-List ADT

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

-get(i): Return the element of S with index i; an error occurs if $i < 0$ or $i > n - 1$

-set(i,e): Replace the element at index i with e and return the old element; an error condition occurs if $i < 0$ or $i > n - 1$

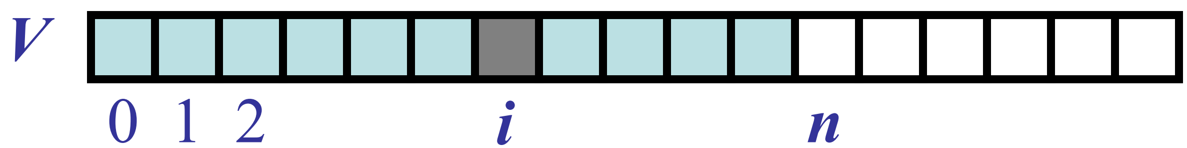
-add(i,e): Insert a new element into S which will have index i; an error occurs if $i < 0$ or $i > n$

-remove(i): Remove from S the element at index i; an error occurs if $i < 0$ or $i > n - 1$

(also support methods: size() and isEmpty())

Natural Implementation of Array-List: with an Array

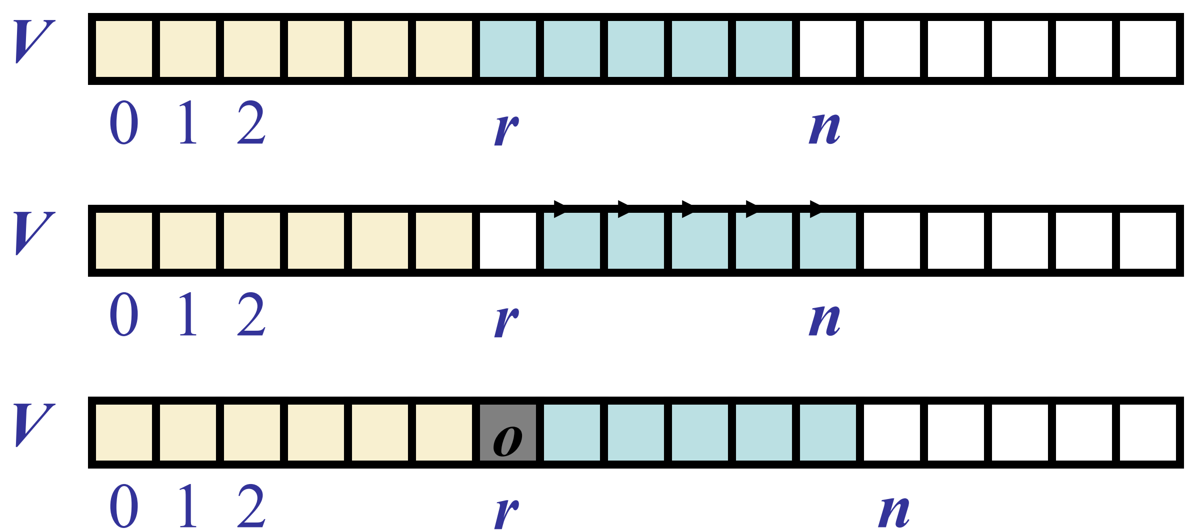
- Array *V* of size *N*
- A variable *n* keeps track of the size of the array-list (number of elements stored)
- Operation *get*(*i*) is implemented in *O*(1) time by returning *V*[*i*]



Insertion

- In operation *add*(*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

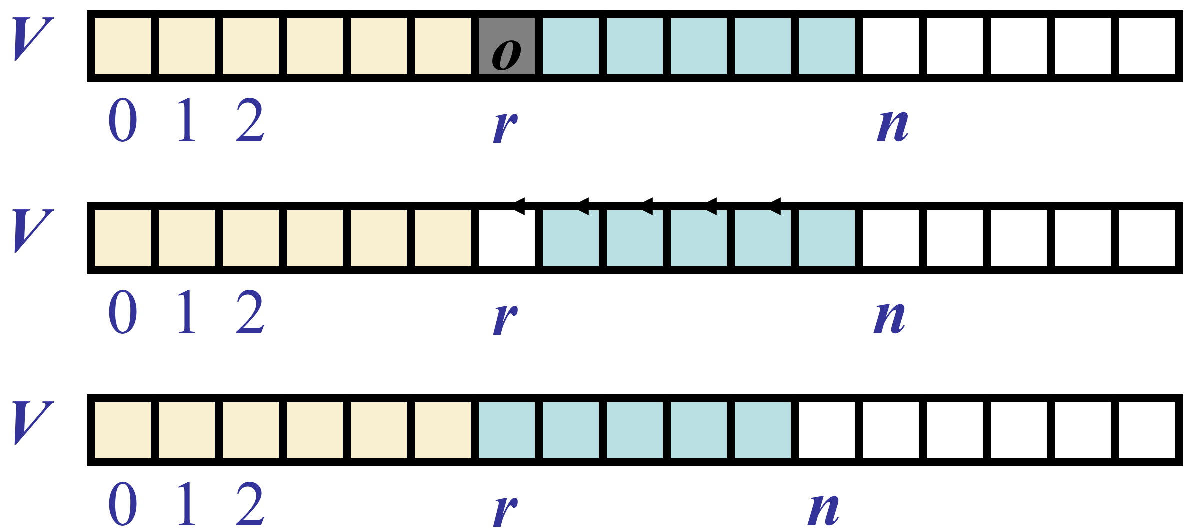
```
add(r,o):
  for i = n - 1, n - 2, ... , r do
      V[i+1] ← V[i]
  V[r] ← o
  n ← n + 1
```



Deletion

- In operation *remove*(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

```
remove(r):
  e ← V[r]
  for i = r, r + 1, … , n - 2 do
       V[i] ← V[i + 1]
  n ← n – 1
```



Java Implementation

```java
// public methods
/** Returns the number of elements in the array list. */
public int size( ) { return size; }
/** Returns whether the array list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the element at index i. */
public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];
}
/** Replaces the element at index i with e, and returns the replaced element. */
public E set(int i, E e) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

Java Implementation, 2

```
  /** Inserts element e to be at index i, shifting all subsequent elements later. */
  public void add(int i, E e) throws IndexOutOfBoundsException,
                                     IllegalStateException {
    checkIndex(i, size + 1);
    if (size == data.length)                    // not enough capacity
      throw new IllegalStateException("Array is full");
    for (int k=size-1; k >= i; k--)             // start by shifting rightmost
      data[k+1] = data[k];
    data[i] = e;                                // ready to place the new element
    size++;
  }
  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
  public E remove(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    for (int k=i; k < size-1; k++)              // shift elements to fill hole
      data[k] = data[k+1];
    data[size-1] = null;                        // help garbage collection
    size--;
    return temp;
  }
  // utility method
  /** Checks whether the given index is in the range [0, n-1]. */
  protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
    if (i < 0 || i >= n)
      throw new IndexOutOfBoundsException("Illegal index: " + i);
  }
}
```

Performance of Array-List with arrays

The space used by the data structure is $O(n)$

size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i,e)	$O(1)$
add(i,e)	$O(n)$
remove(i)	$O(n)$

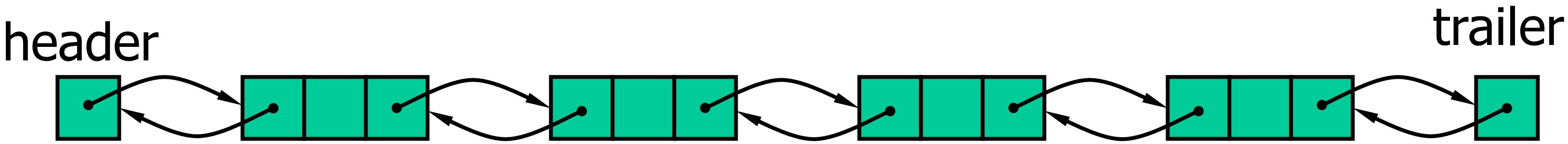
- In an *add* operation, when the array is full, instead of having an ERROR, we can replace the array with a larger one: extendable arrays (will see next).

Performance of Array-List with arrays

- Time time complexity for operations:

size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i,e)	$O(1)$
add(i,e)	$O(n)$
remove(i)	$O(n)$

BAD IDEA to implement an Array-list with a doubly linked list as it t would be quite inefficient !




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

size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(n)$
set(i,e)	$O(n)$
add(i,e)	$O(n)$
remove(i)	$O(n)$

Class java.util.ArrayList<E>

- Inherits from
  - java.util.AbstractCollection<E>
  - java.util.AbstractList<E>
- Implements
  - Iterable<E>
  - Collection<E>
  - List<E>
  - RandomAccess

Implementation with extendable arrays

- The methods
  - size(), isEmpty(), get(int) and set(int,E) in time O(1)
  - add(int,E) and remove(int) in time O(n)

Extendable/Dynamic Array-based Array List

- Let push(o) be the operation that adds element o at the end of the list
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We **compare** the **incremental strategy** and the **doubling strategy** by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

Incremental strategy : $n \leftarrow n+c$ (ex: $n \leftarrow n+100$)

- Over n push operations, we replace the array $k = n/c$ times, where c is a constant. (ex: $k = n/100$)
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of a push operation is $O(n)$

Doubling Strategy Analysis

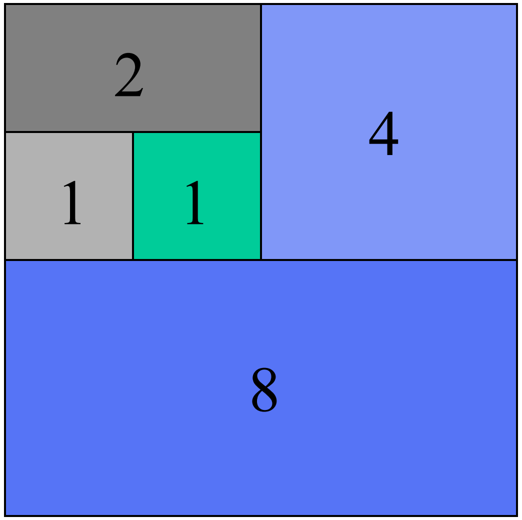
Doubling strategy : $n \leftarrow 2n$

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 = 3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series



Positional Lists

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

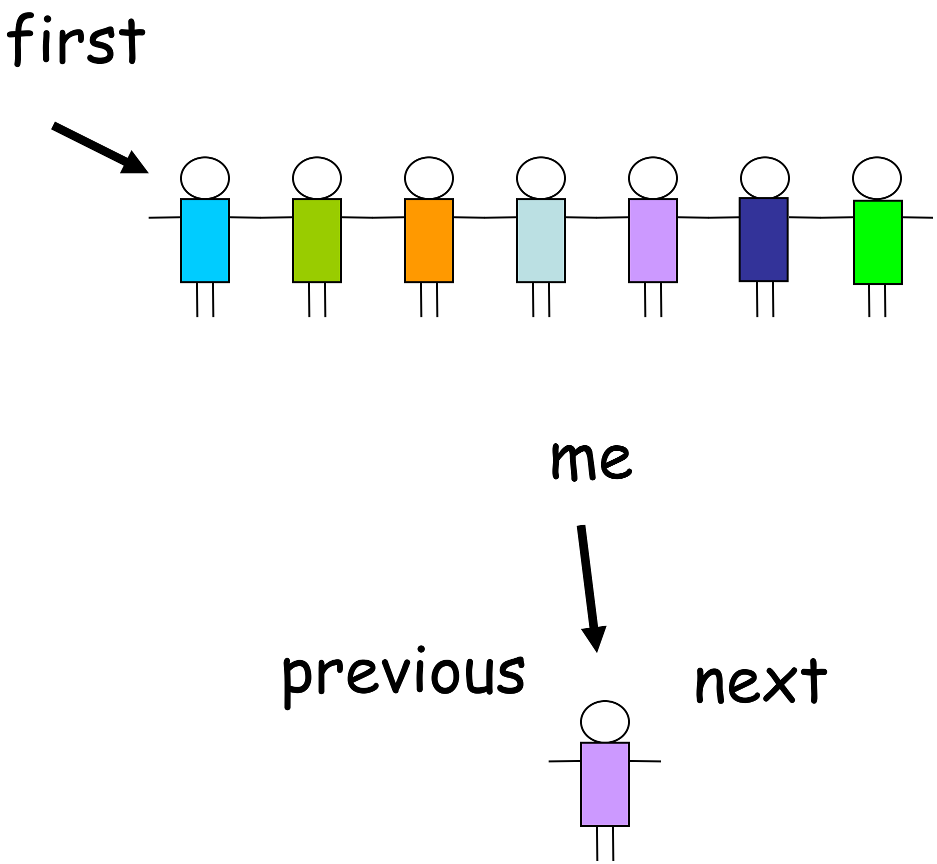
- Cannot access any element directly, can access just first or last.

(node) (address)

- Elements are accessed by position. (place)

Positions are defined relatively to other positions

(before/after relation)



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

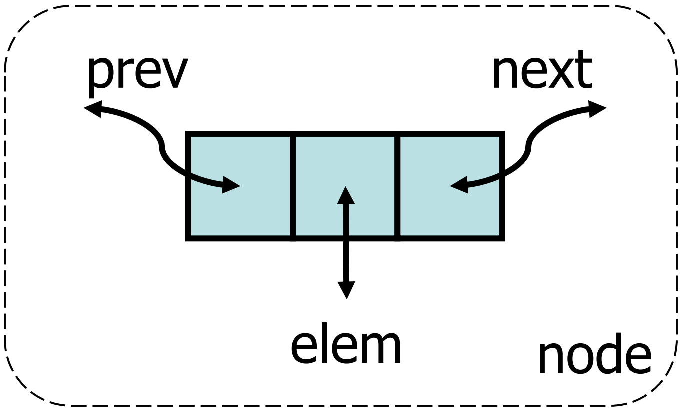
The Positional-List ADT

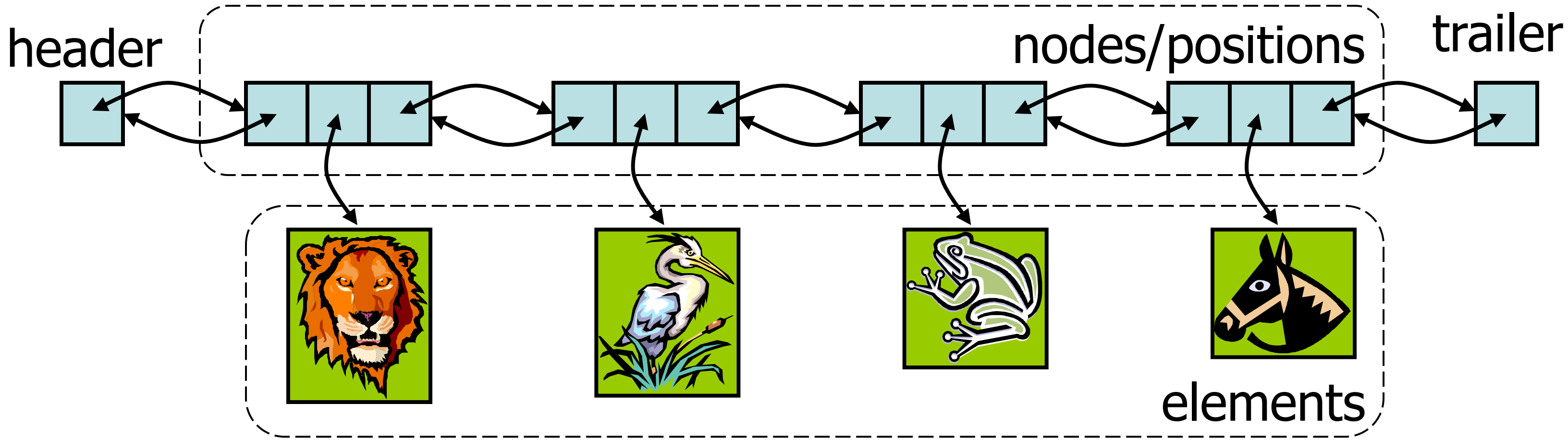
ADT with position-based methods

- generic methods — size(), isEmpty()
- accessor methods — first(), last()
  before(p), after(p)
- update methods
  addFirst(e), addLast(e)
  addBefore(p,e), addAfter(p,e)
  set(p,e), remove(p)

Natural Implementation: with a Linked List

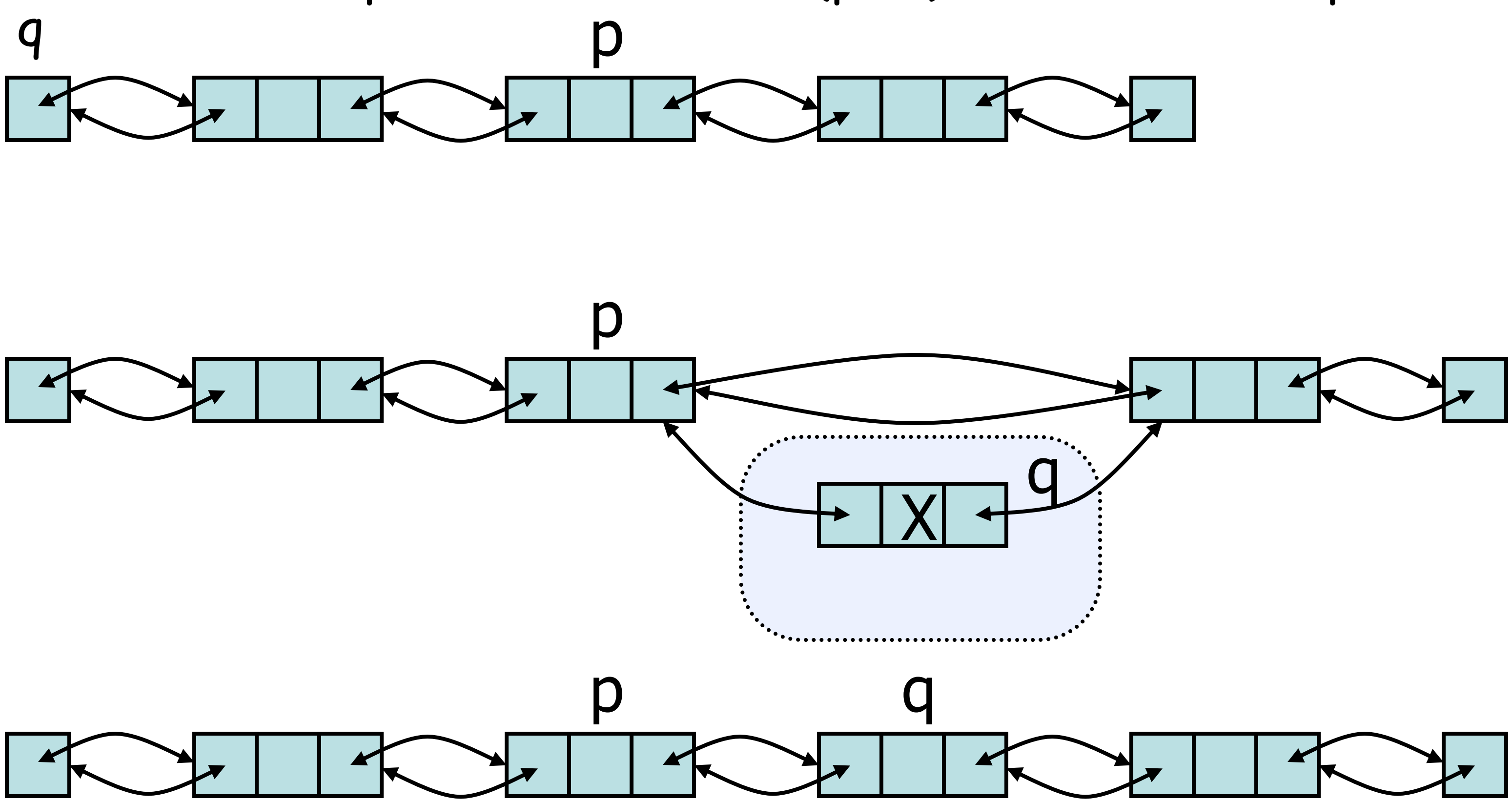
- A doubly linked list provides a natural implementation of the Positional-List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





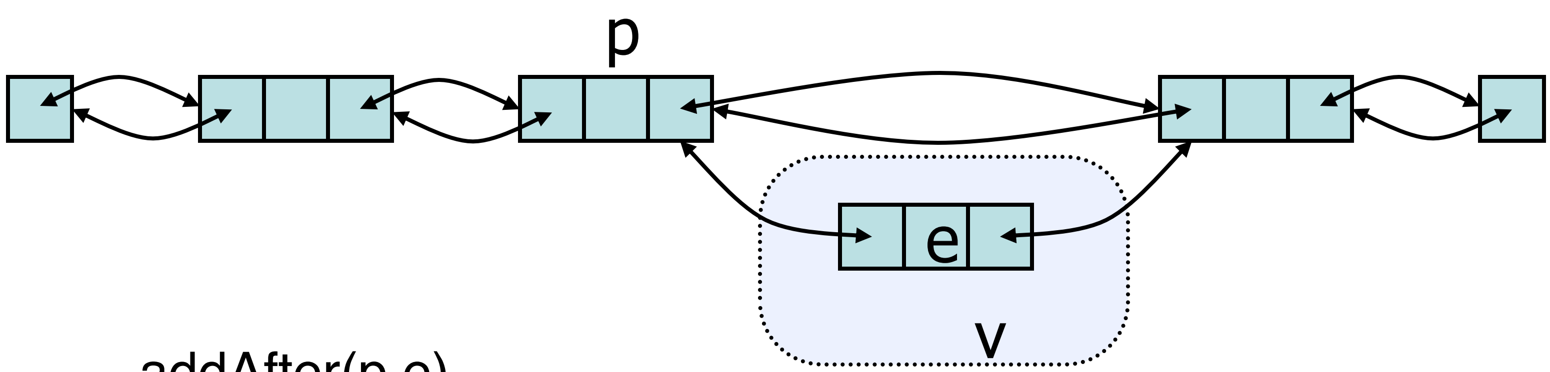
Insertion

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



Insertion

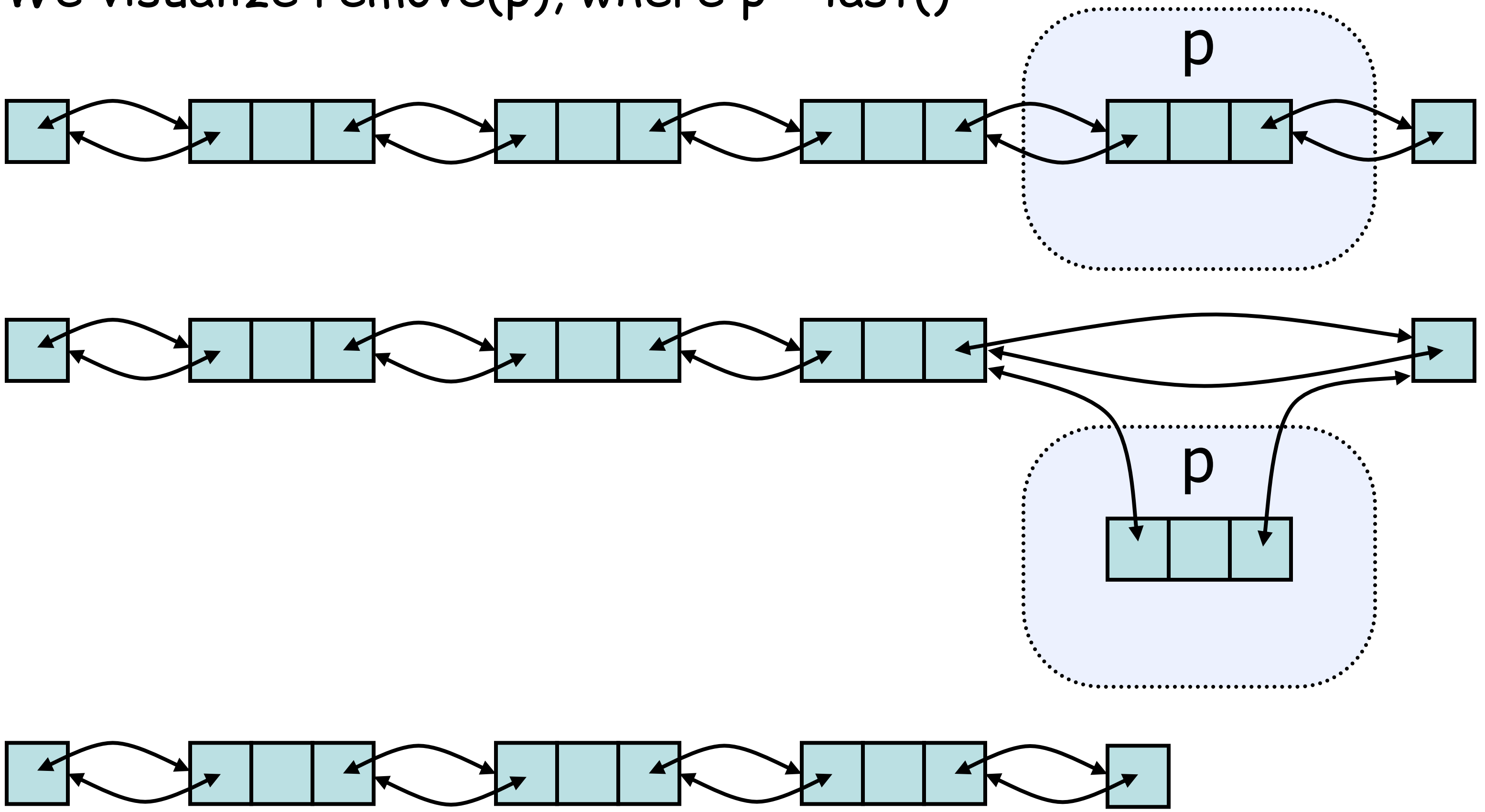
- We visualize operation addAfter(p, e), which returns position q

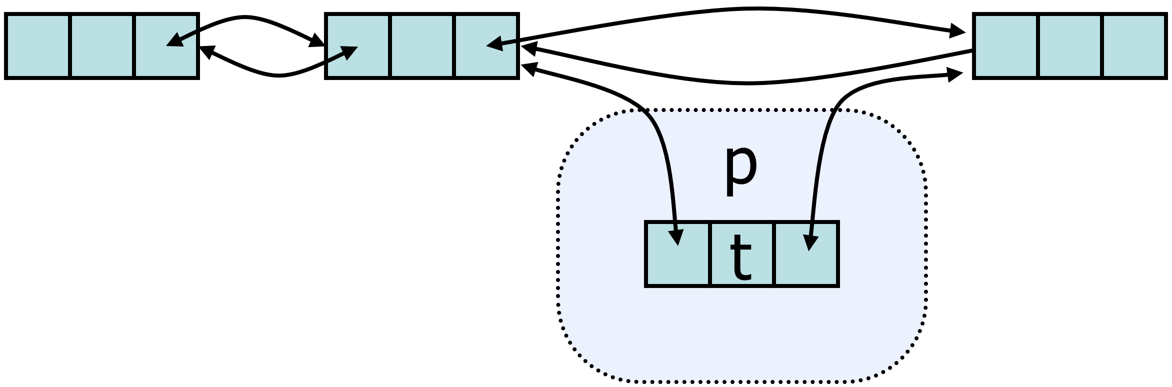


addAfter(p,e)
  Create a new node v
  v.setElement(e)
  v.setPrev(p)
  v.setNext(p.getNext())
  (p.getNext()).setPrev(v)
  p.setNext(v)

Deletion

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





remove(p)

```
t ← p.element
(p.getPrev()).setNext(p.getNext())
(p.getNext()).setPrev(p.getPrev())
p.setPrev(null)
p.setNext(null)
return t
```

Performance

- In the implementation of the Positional-List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$

All the operations of the Positional-List ADT
size(), isEmpty(), addFirst(e), addLast(e)
addBefore(p,e), addAfter(p,e), set(p,e), remove(p)
run in $O(1)$ time.

A more general ADT: Sequence ADT

- Combines the Array-List and Positional-List ADT providing all of its operations plus **bridge methods**.

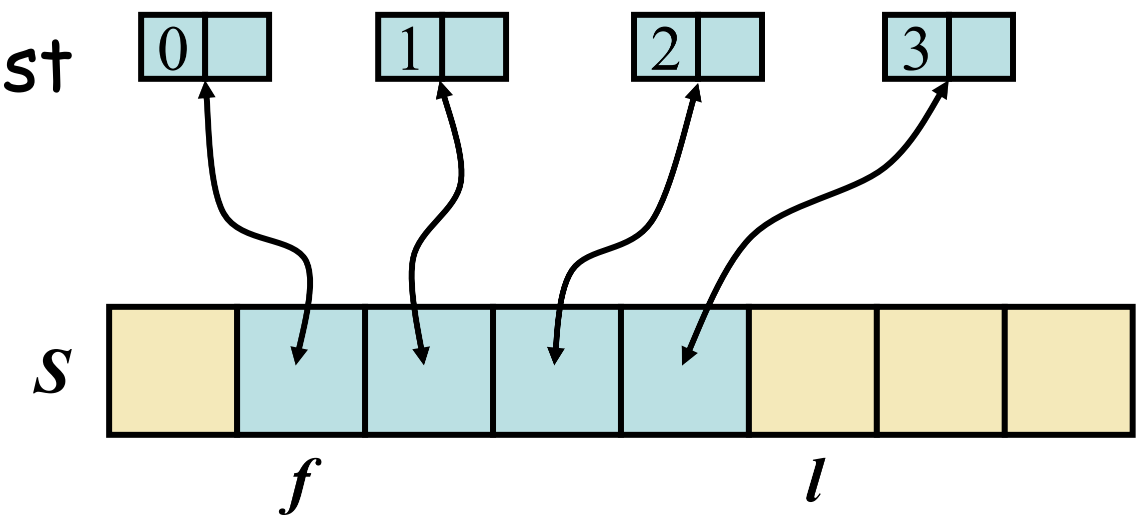
- Adds methods that bridge between index and positions

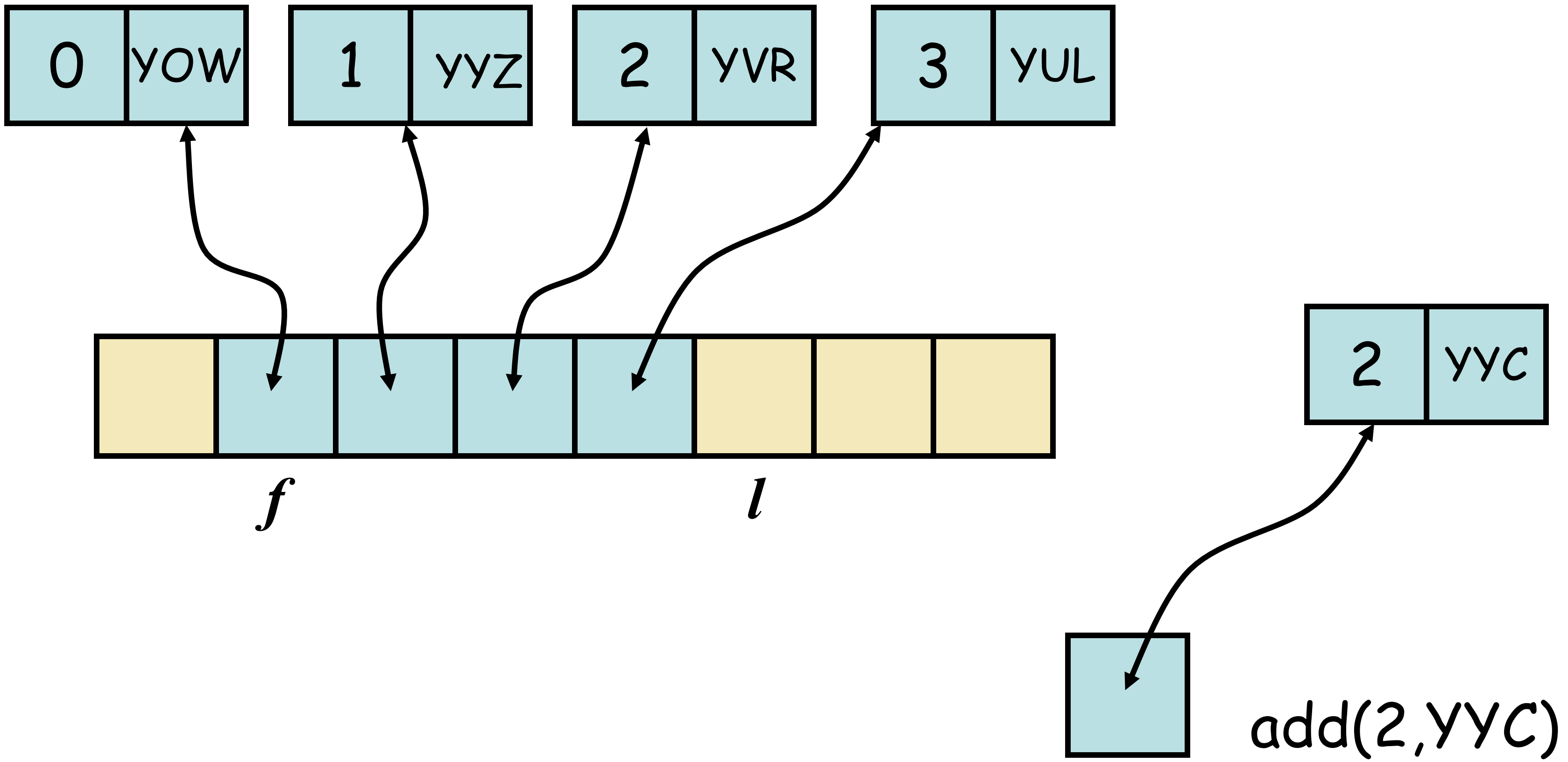
  -atIndex(i) returns a position

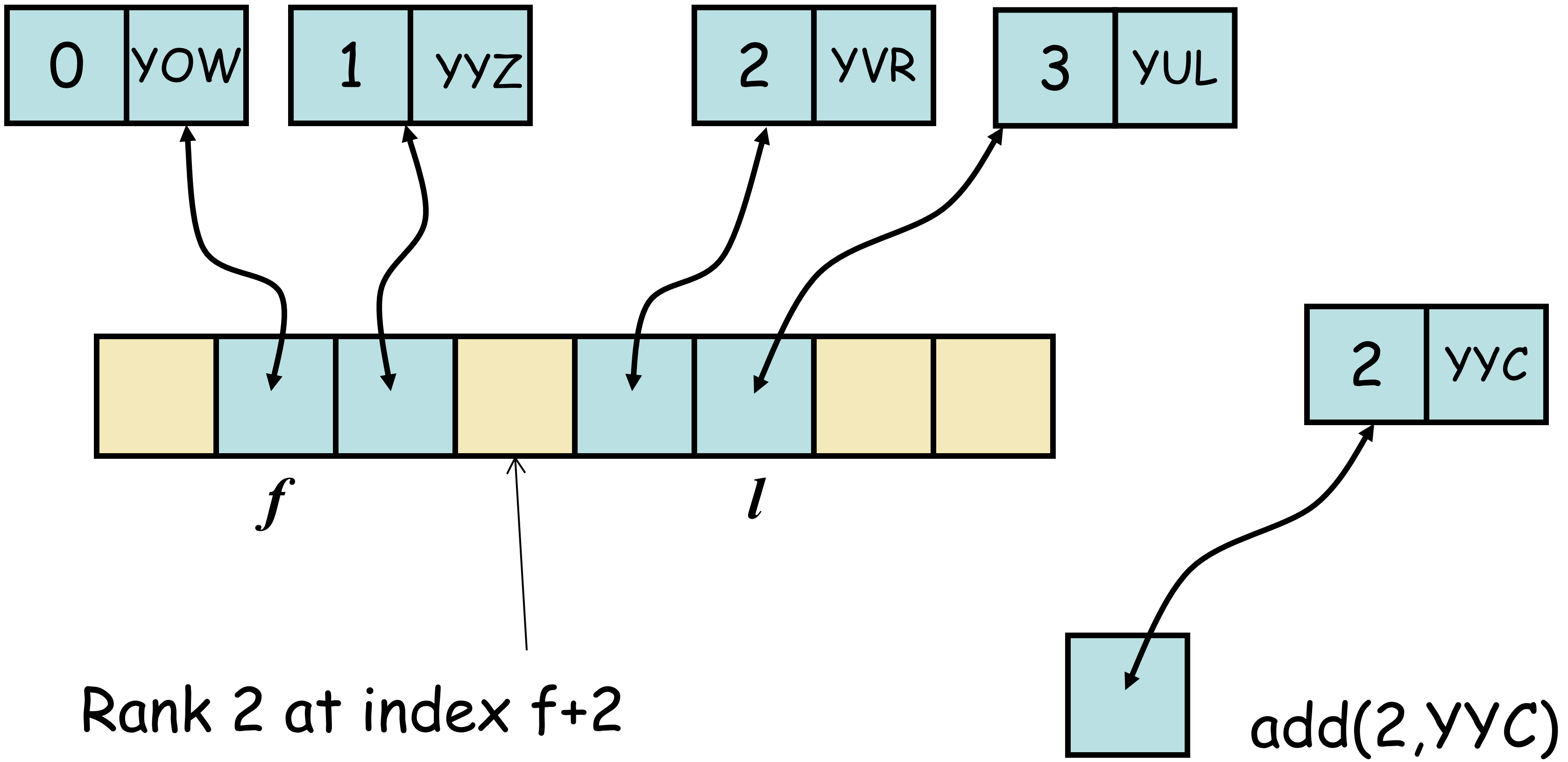
  -indexOf(p) returns an integer index

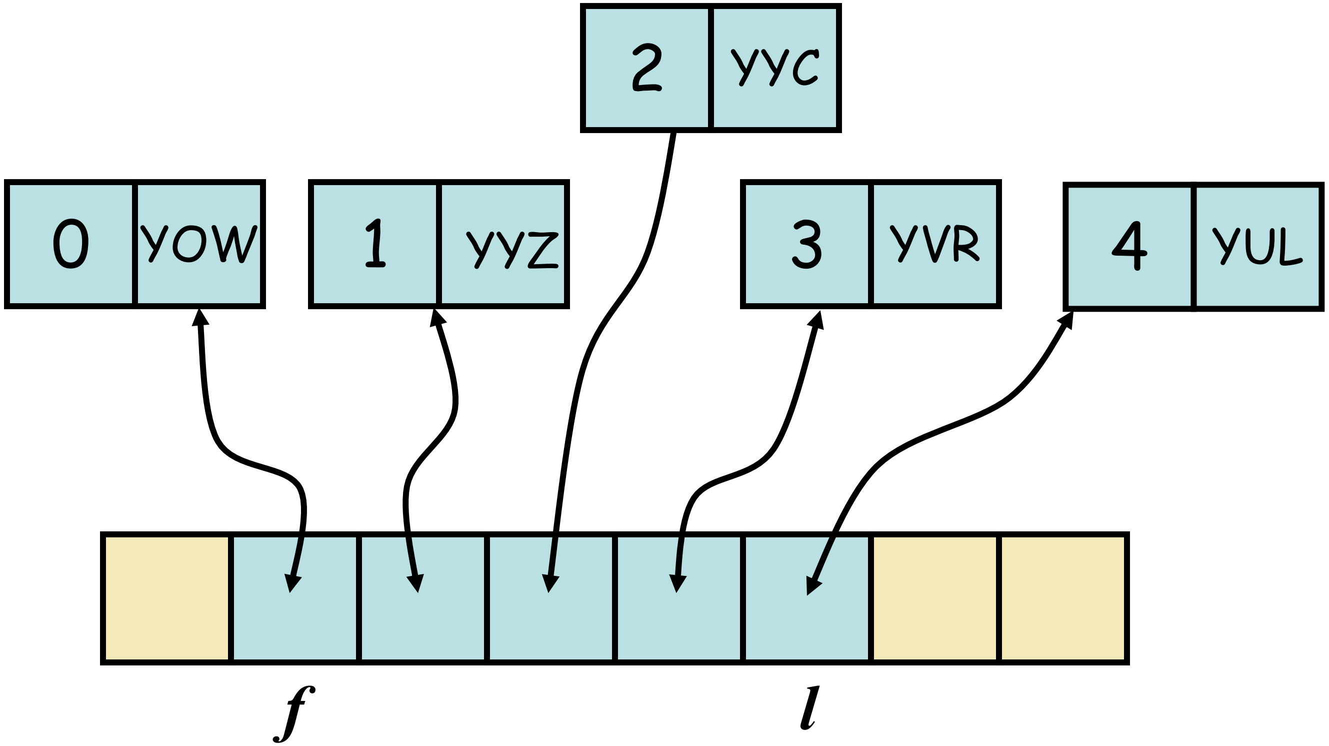
An array-based Implementation

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

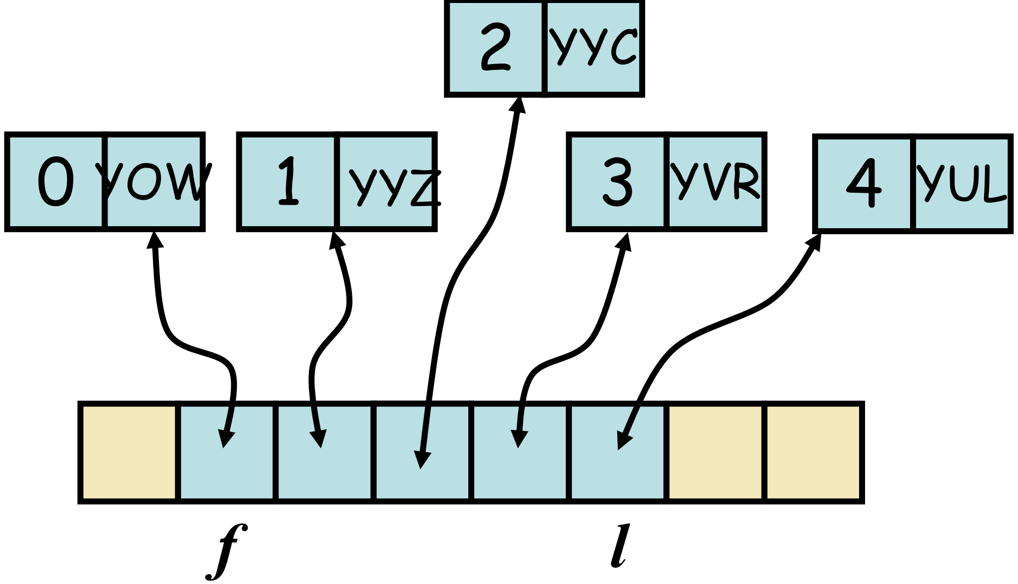



0
YOW
1
YYZ
2
YVR
3
YUL
f
l
2
YYC
add(2,YYC)

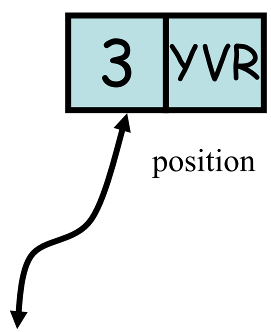

0
YOW
1
YYZ
2
YVR
3
YUL
f
l
Rank 2 at index f+2
2
YYC
add(2,YYC)



Change all other ranks



atIndex(i) Direct access to the position at index f+i

indexOf(position): 

Immediate access to the corresponding index

Sequence: Array-based Implementation

addFirst, addBefore, addAfter, remove

$O(n)$

Also: add, remove based on the index

$O(n)$

Other methods

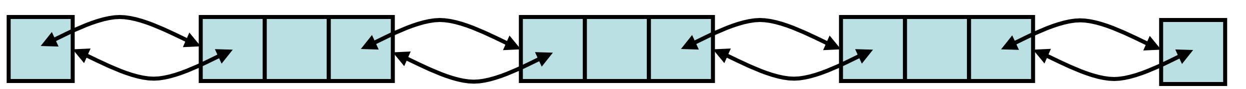
$O(1)$

Sequence: Implementation with Doubly Linked List

All methods are inherited ….

Bridges:

atIndex(i), indexOf(p): O(n)



Must traverse the list

Summary: Implementation of Sequences using Array

Need to move elements

add(i,e), addFirst,addBefore,addAfter ---- O(n)

remove(index), remove(position) ---- O(n)

Bridges: atIndex(i), indexOf(p): ---- O(1)

get(i), set(i,e) ----- O(1)

Because the position contains also the index

Summary: Implementation of Sequences using Doubly-linked lists

addFirst,addBefore,addAfter, remove(position) ---- O(1)

add(i,e)
remove(index) ---- O(n)

Bridges: atIndex(i), indexOf(p): ---- O(n)

Need to traverse to find an index

get(i), set(i,e) ----- O(n)

Appendix about: Iterators

- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a

hasNext(): Returns true if there is at least one additional element in the sequence, and false otherwise.

next(): Returns the next element in the sequence.

The Iterable Interface

- Java defines a parameterized interface, named Iterable, that includes the following single method:
  - iterator( ): Returns an iterator of the elements in the collection.
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
- Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

The for-each Loop

- Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax:

```
for (ElementType variable : collection) {
  loopBody                                    // may refer to "variable"
}
```

is equivalent to:

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                                    // may refer to "variable"
}
```