© Ron Niebrugge

CSI2110

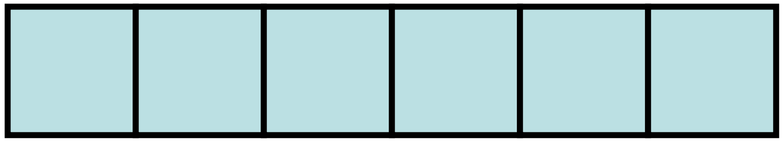
Data Structures and Algorithms

Prof. WonSook Lee

We learned…

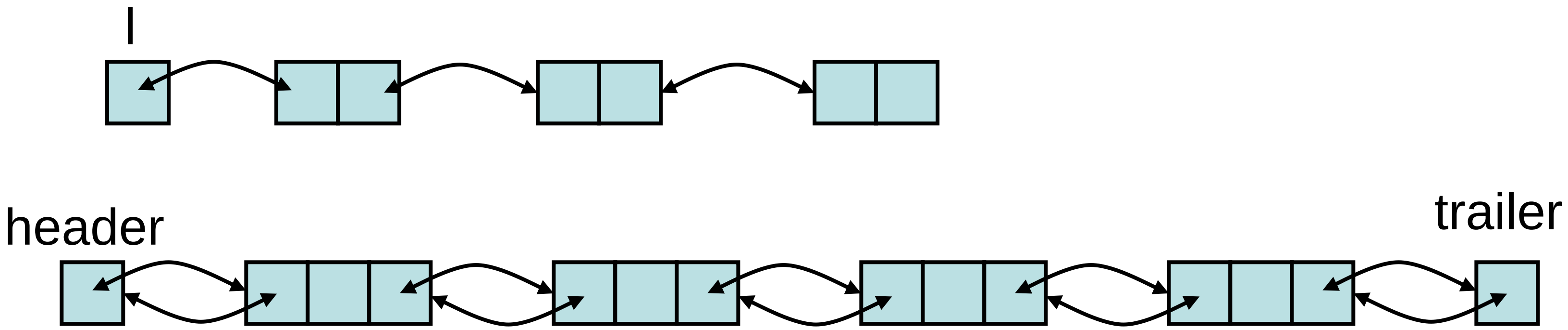
Review:
Basic Data Structures (“concrete” data structures)

Array



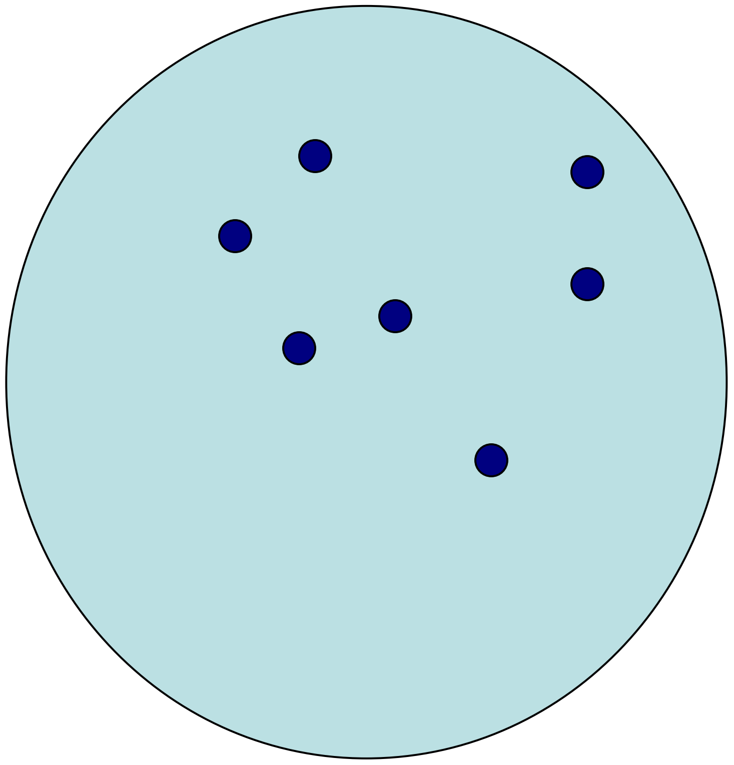
Linked Lists

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

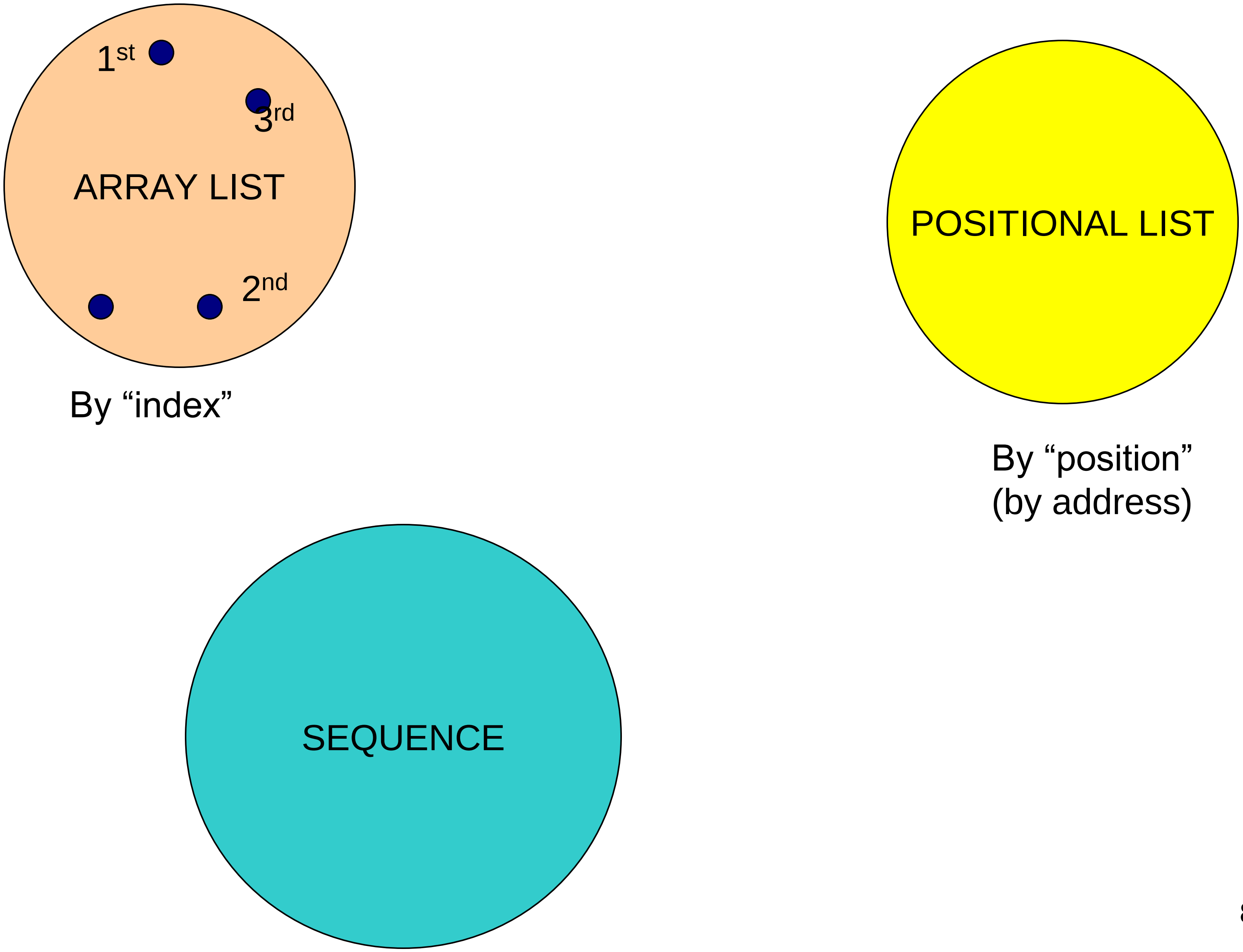
“first in first out”

DEQUE

Insert: InsertFirst, InsertLast
Remove: RemoveFirst RemoveLast

What are we going to see next ...

Generalization…

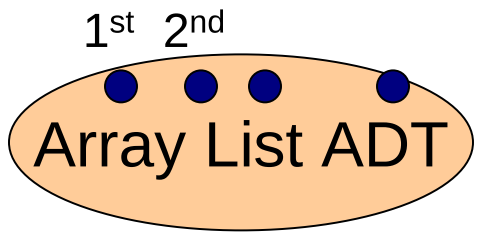

1st
3rd
ARRAY LIST
2nd
By “index”
POSITIONAL LIST
By “position”
(by address)
SEQUENCE

Lists

- Array-List ADT
- Positional-List ADT
- Sequence ADT

Lists or Sequences

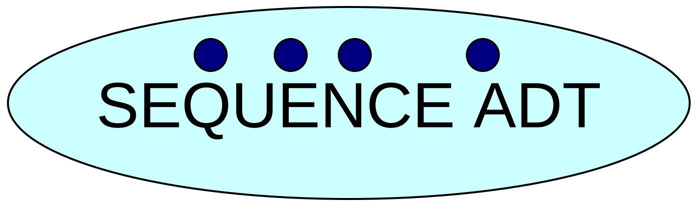
LISTS or SEQUENCES= collection of elements in linear order



Positional List ADT

To be implemented by arrays. Access by "index"

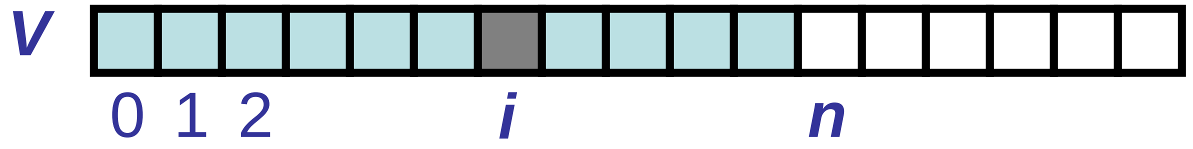
To be implemented by linked lists Access by "position" (or address)



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 $i > n - 1$

Adapter Pattern

- Two data structures A and B are often similar
- Adapt data structure B to be used as A
- Create a “wrapper class” A holding B

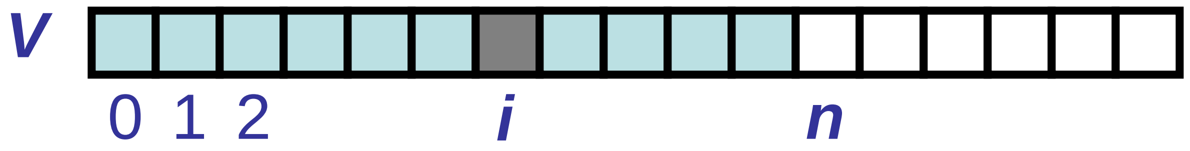
Examples:

- Regular array as an ArrayList, or
- ArrayList can be adapted as a Deque

Deque	ArrayList
getFirst(), getLast()	get(0), get(size()-1)
addFirst(e), addLast(e)	add(0,e), add(size(),e)
removeFirst(), removeLast()	remove(0), remove(size()-1)

Natural Implementation of Array-List: with an Array

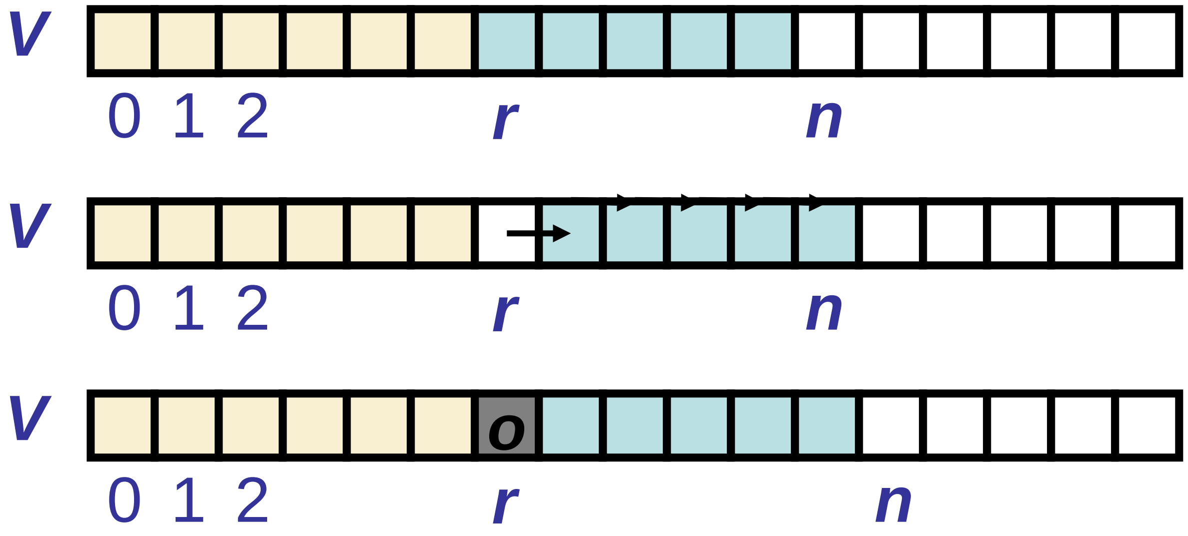
- Array **V** of size **N**
- A variable **n** keeps track of the size of the array-list (number of elements stored)
- Operation **get**(**i**) is implemented in **O**(1) time by returning **V**[**i**]



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

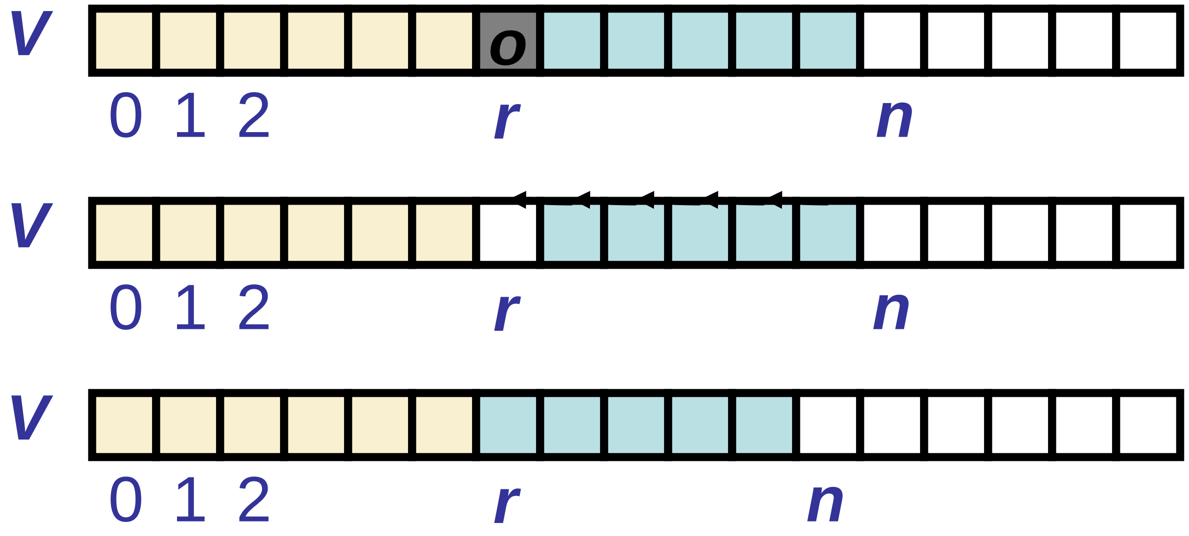
```
add(r,o):
    for i = n - 1, n - 2, ... , r do
            S[i+1] ← s[i]
    S[r] ← o
    n ← n + 1
```



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 ← 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 an array-list
  - The space used by the data structure is **O**(**n**)
  - **size**, **isEmpty**, **get** and **replace** run in **O**(1) time
  - **insert** and **remove** run in **O**(**n**) time
- In an **insert** operation, when the array is full, instead of having an ERROR, we can replace the array with a larger one: extendable arrays seen earlier

Performance (contd.)

- Time time complexity of the various methods:

size	**O**(1)
isEmpty	**O**(1)
get	**O**(1)
replace	**O**(1)
insert	**O**(**n**)
remove	**O**(**n**)

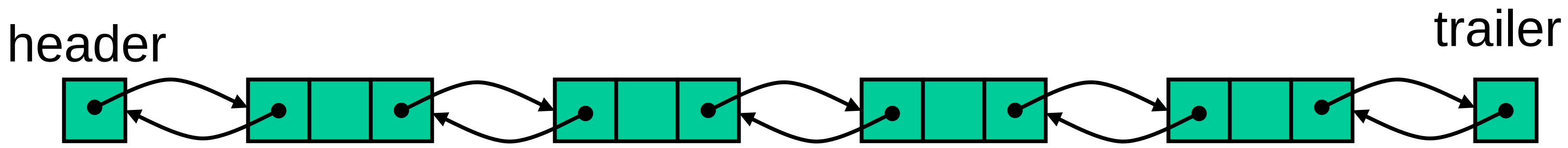
Class java.util.ArrayList<E>

- 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)

If we were to implement an array-list with a doubly linked list it would be quite inefficient !



get(rank) ?

Finding an element at a certain rank

```
Algorithm get(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;
```

Performance with linked list …

size	**O**(1)
isEmpty	**O**(1)
get	**O**(n)
replace	**O**(n)
insert	**O**(**n**)
remove	**O**(**n**)

Positional Lists

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

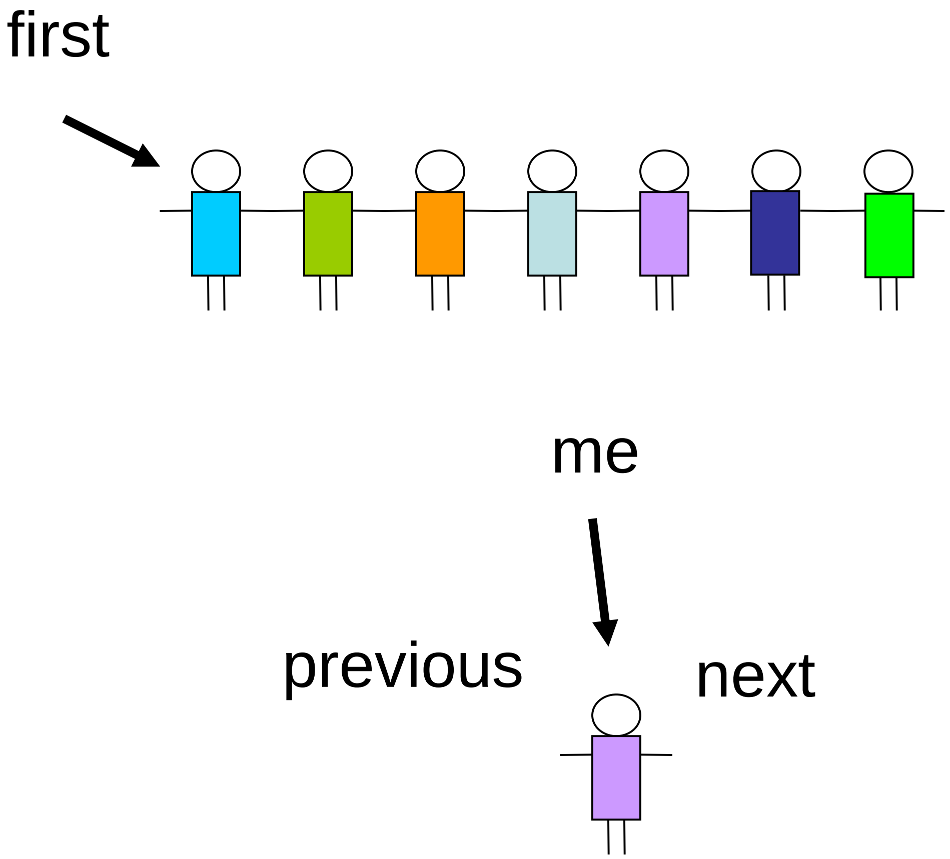
- Cannot access any element directly, can access just first or last.

(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

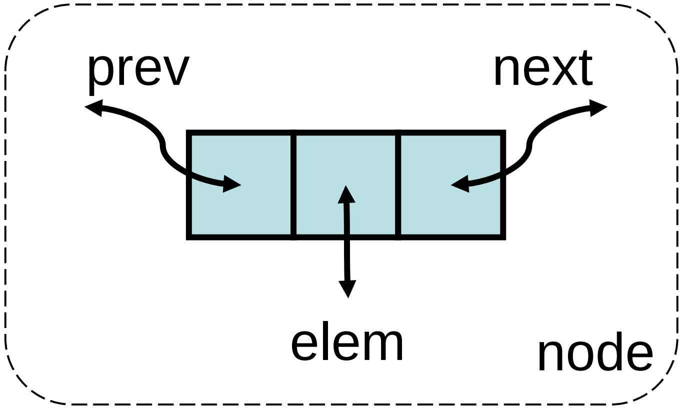
The Positional-List ADT

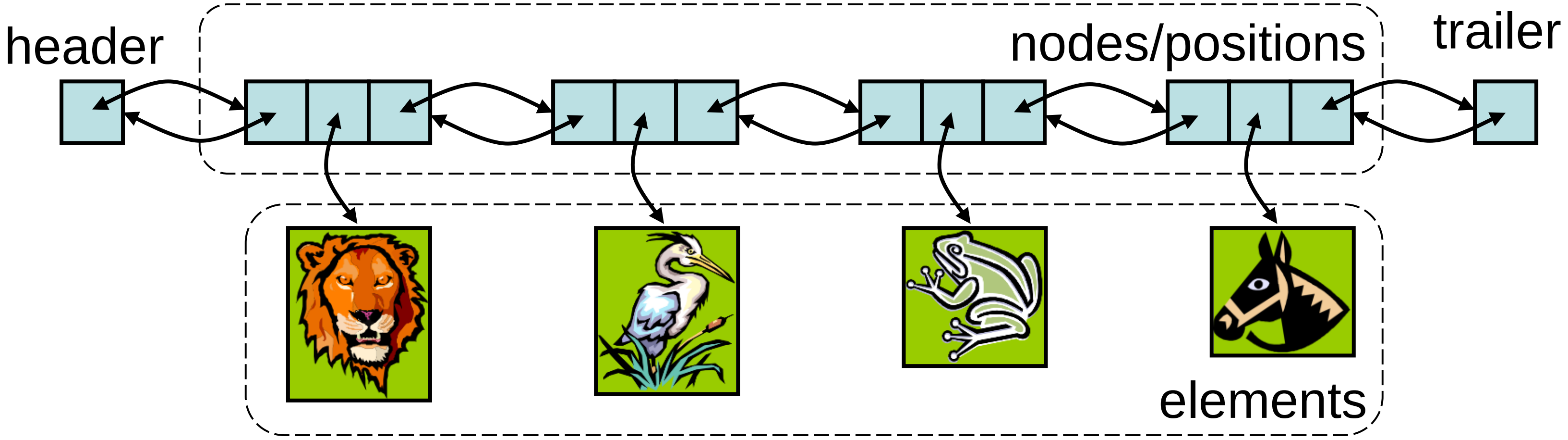
ADT with position-based methods

• 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

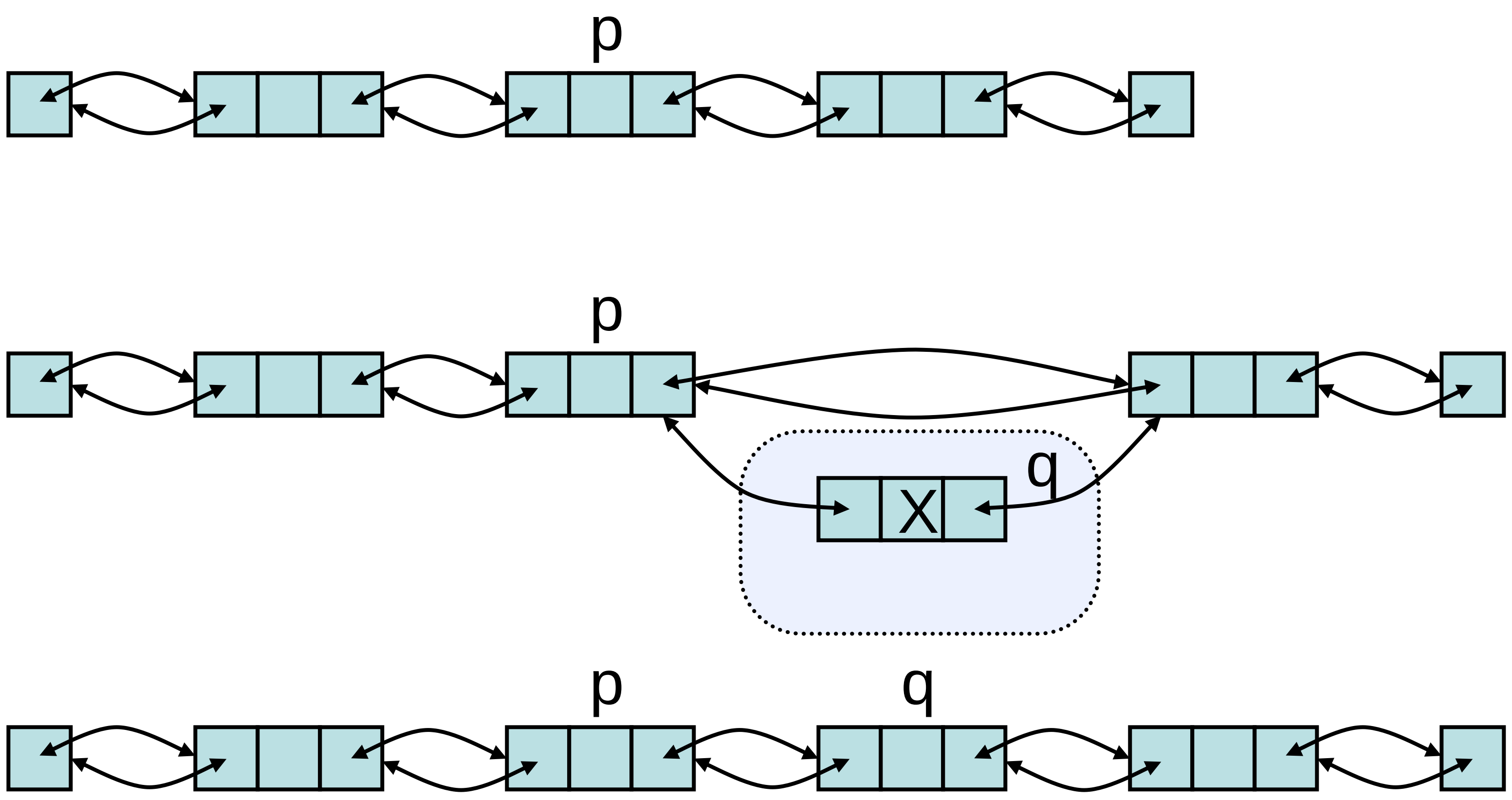
- A doubly linked list provides a natural implementation of the Positional-List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





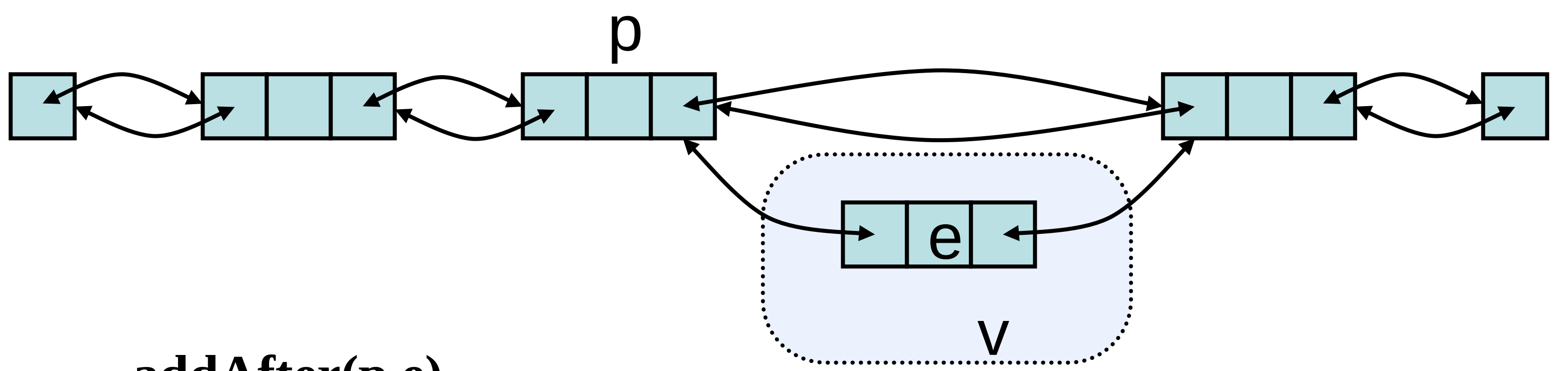
Insertion

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



Insertion

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



addAfter(p,e)

Create a new node v

Correct order??

v.setNext(p.getNext())

v.setElement(e)

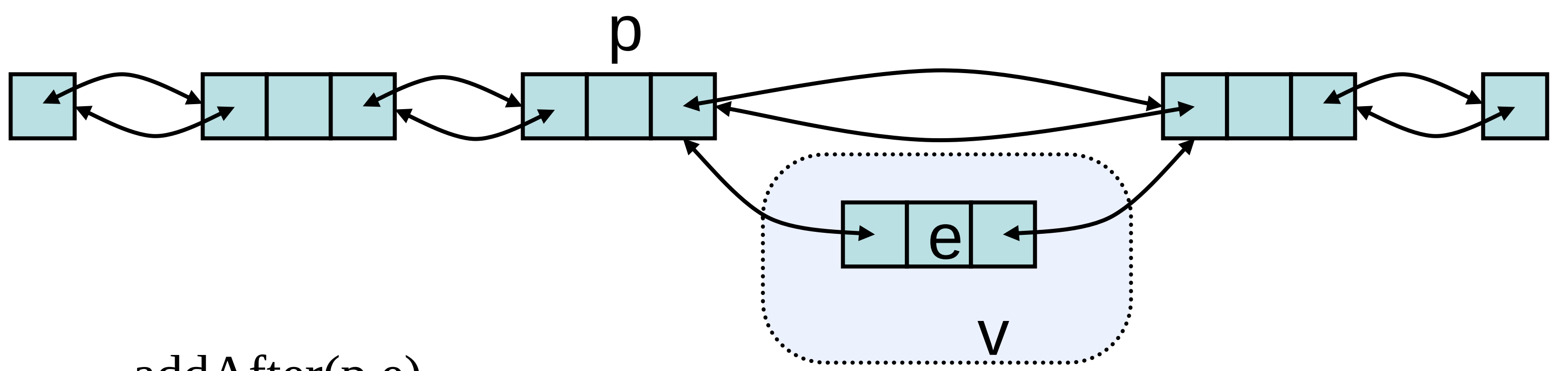
(p.getNext()).setPrev(v)

v.setPrev(p)

p.setNext(v)

Insertion

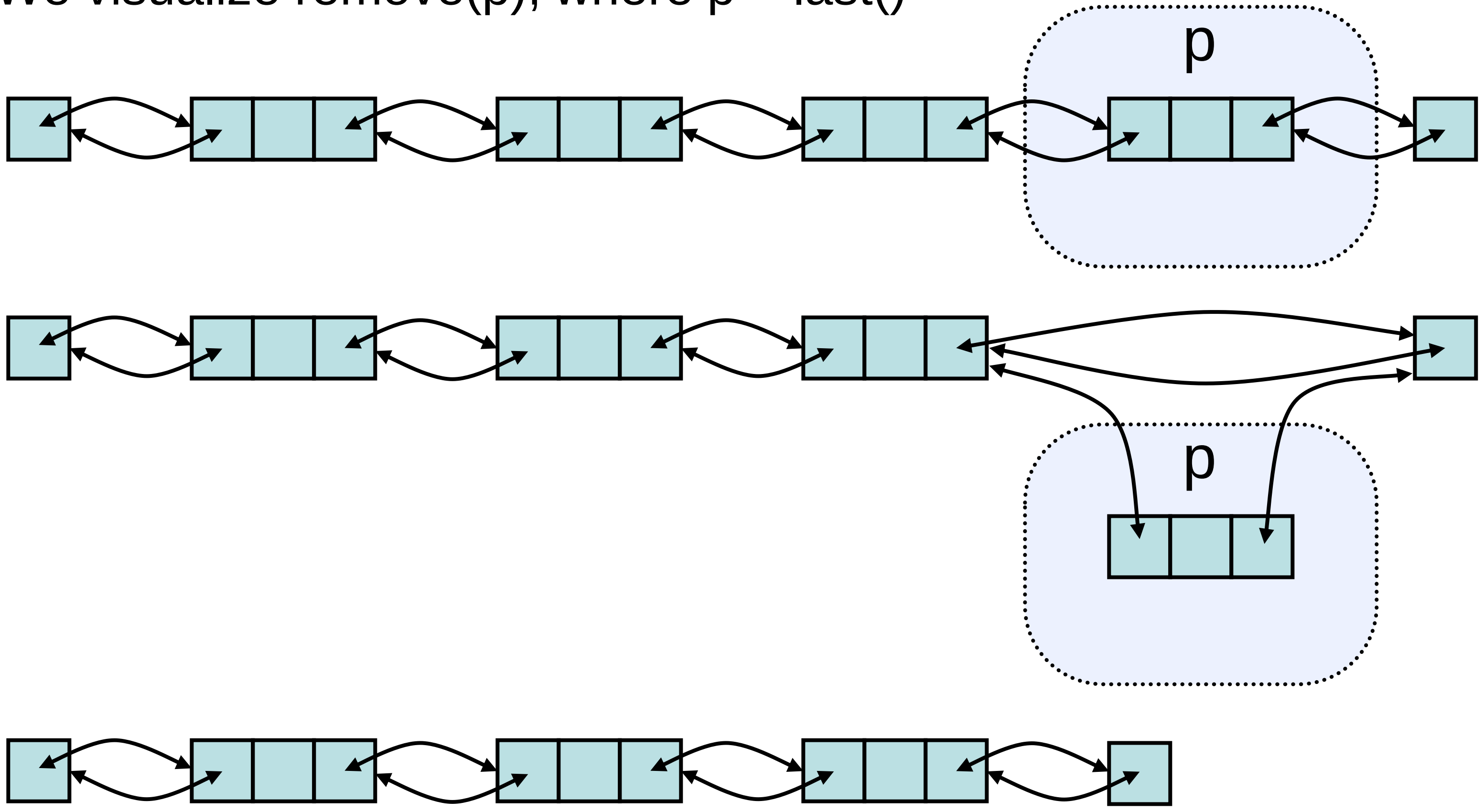
- We visualize operation addAfter(p, e), which returns position v

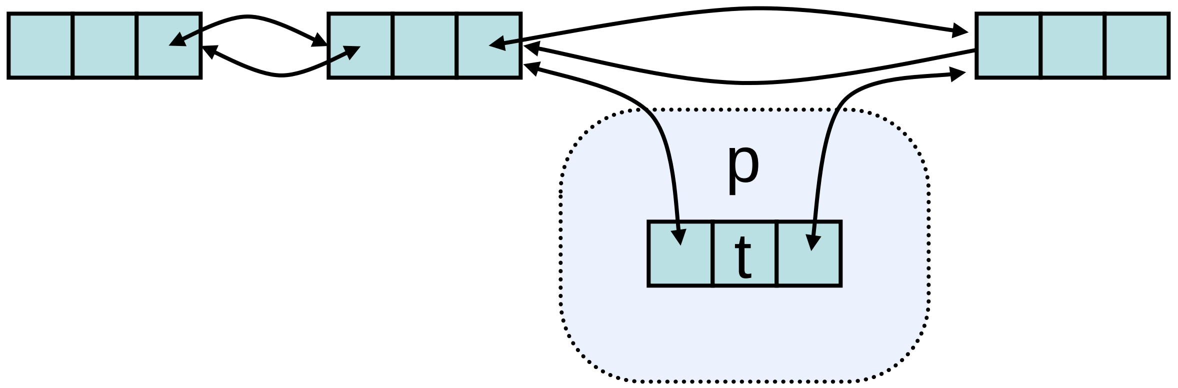


```
addAfter(p,e)
        Create a new node v
        v.setElement(e)
        v.setPrev(p)
        v.setNext(p.getNext())
        (p.getNext()).setPrev(v)
        p.setNext(v)
```

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**)
  - The space used by each position of the list is **O**(1)
  - All the operations of the Positional-List ADT run in **O**(1) time

A more general ADT: Sequence ADT

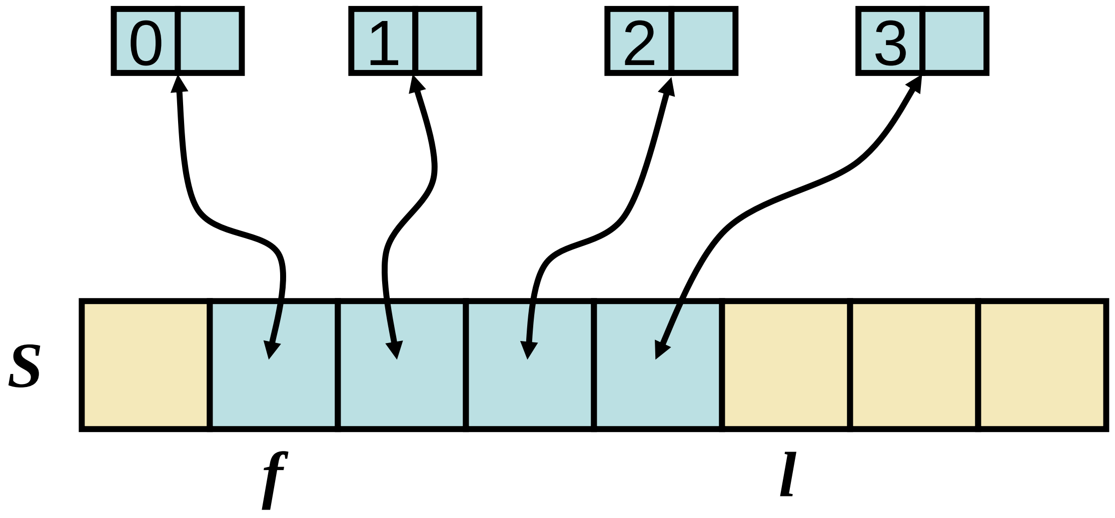
- Combines the Array-List and Positional-List ADT

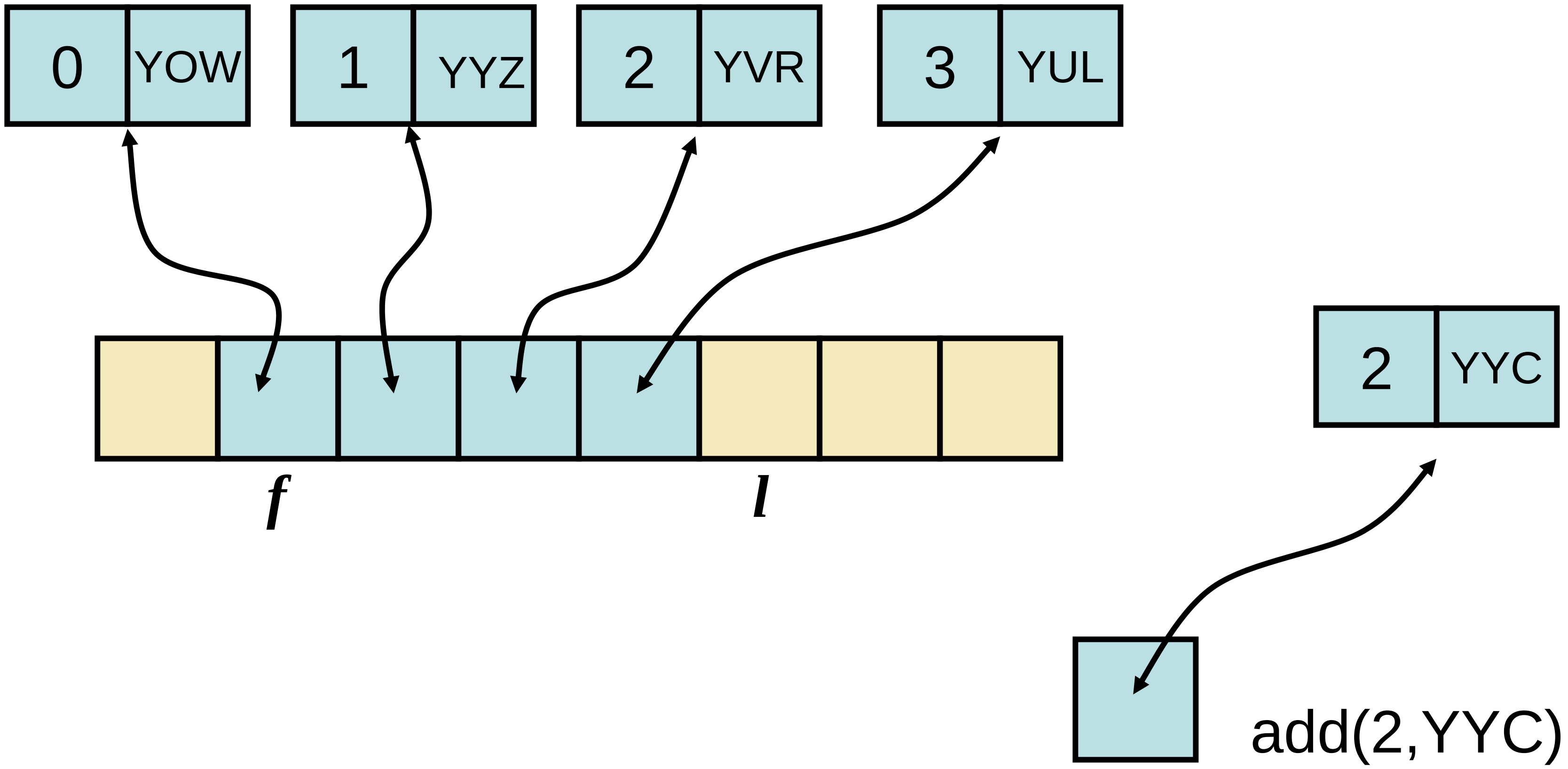
- Adds methods that bridge between index and positions

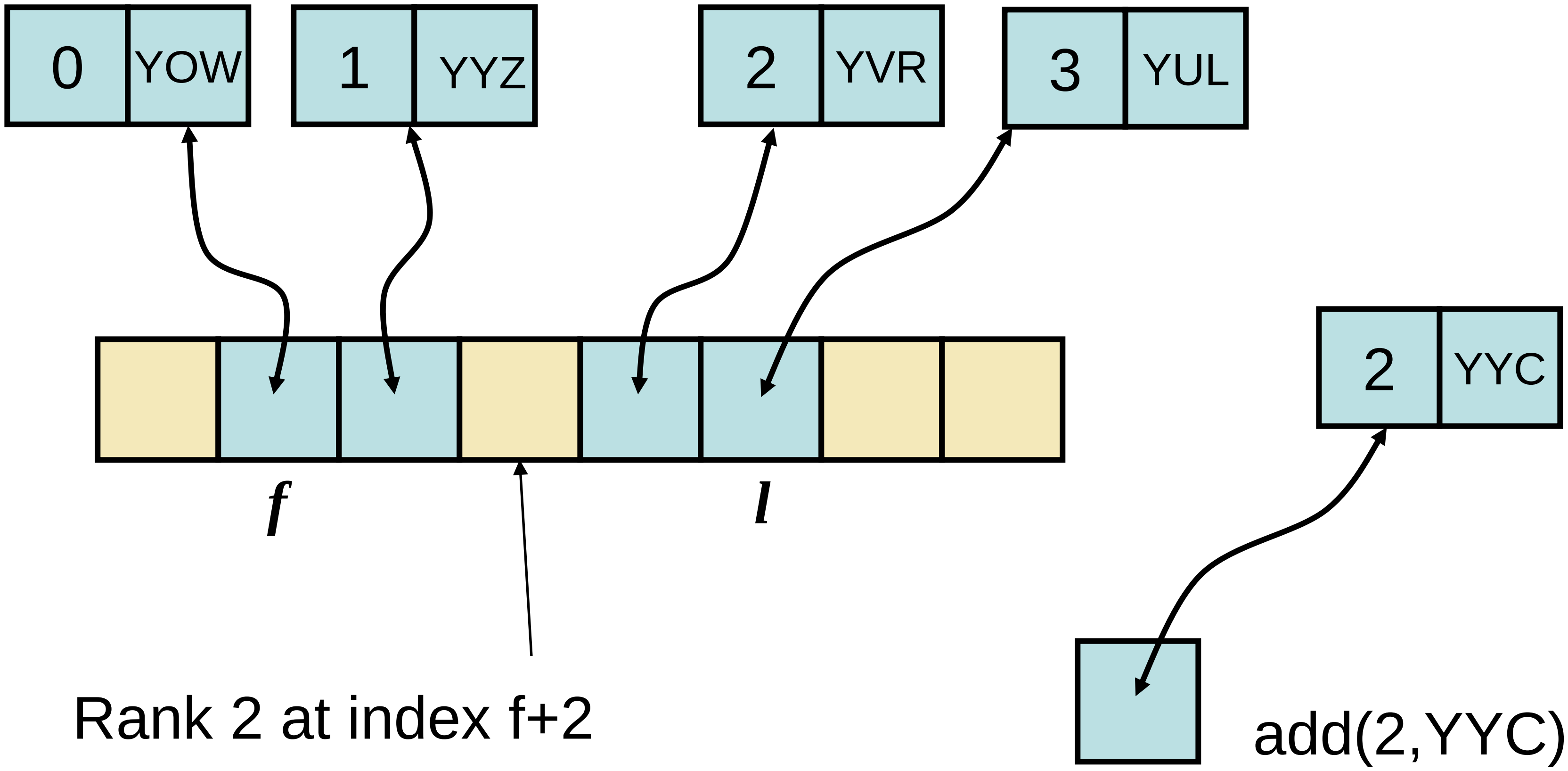
-atIndex(i)	returns a position
-indexOf(p)	returns an integer index

An array-based Implementation

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



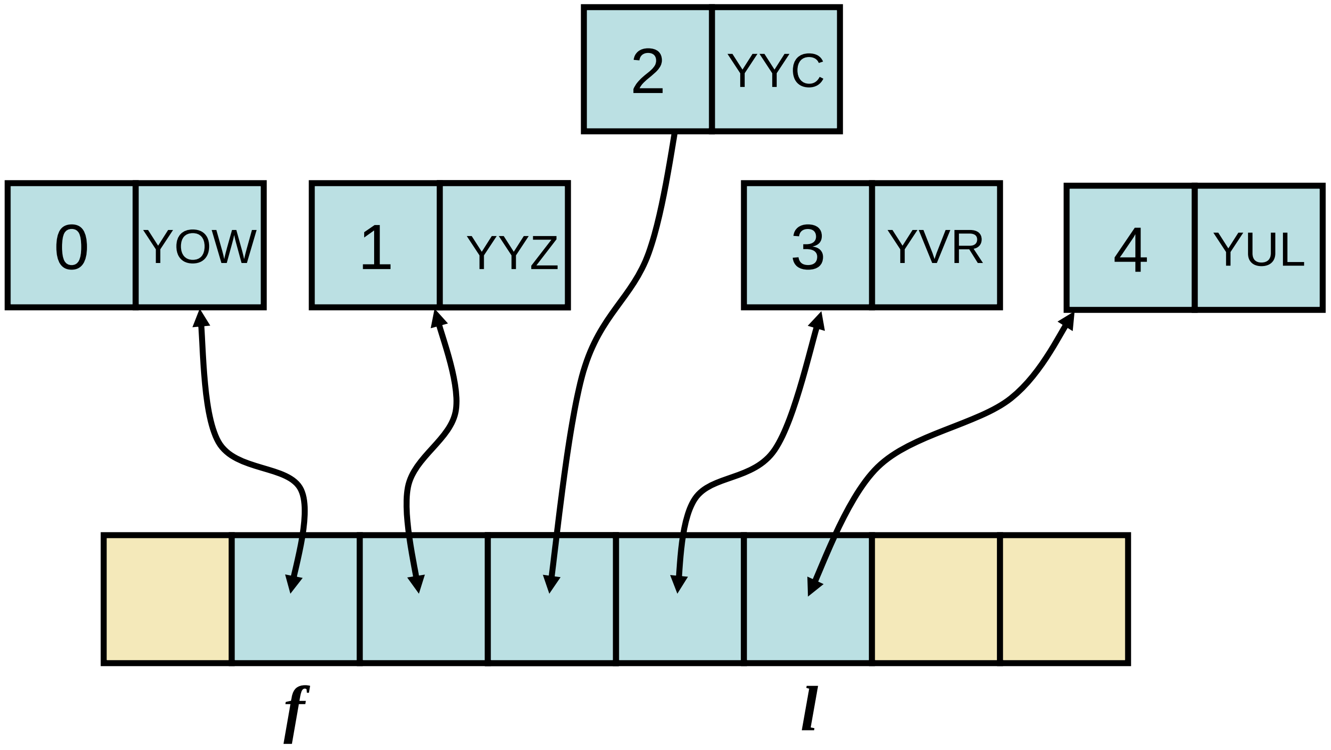

0
YOW
1
YYZ
2
YVR
3
YUL
f
l
2
YYC
add(2,YYC)



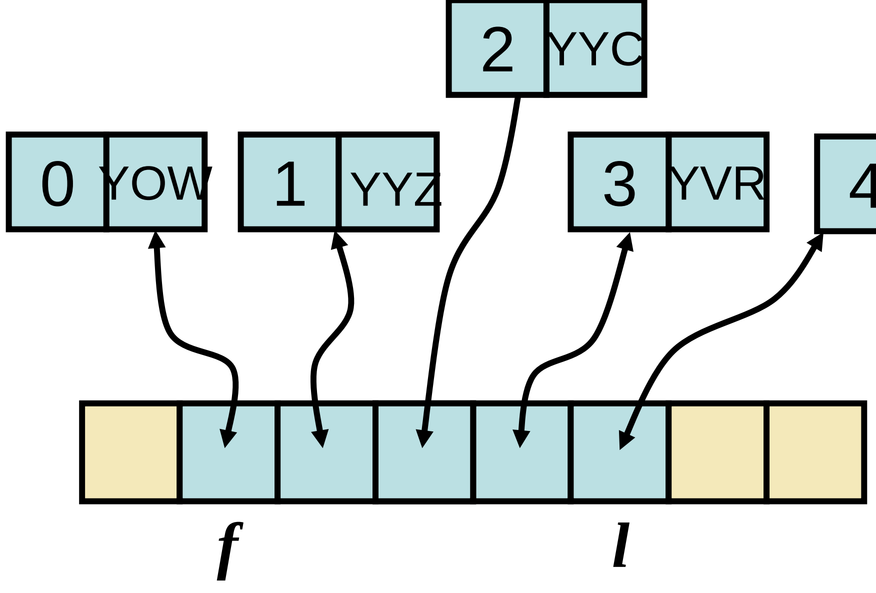
Rank 2 at index f+2

add(2,YYC)

i = f+r

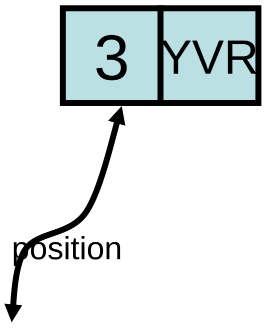


Change all other ranks



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

indexOf(position):



Immediate access to the corresponding index

Array-based Implementation

addFirst, addBefore, addAfter, remove

O(n)

Also: add,   remove based on the index

O(n)

Other methods

O(1)

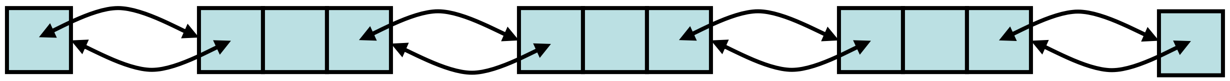
Implementation
with Doubly Linked List

All methods are inherited ….

a b c d e f g H ı i j k l

Bridges:

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



Must traverse the list

Summary: Array-based implementation of Sequences

Need to move elements

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

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

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

r = f-i

p=r, i=f+r

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

Because the position contains also the index

Summary: Implementation of Sequences by Doubly-linked lists

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

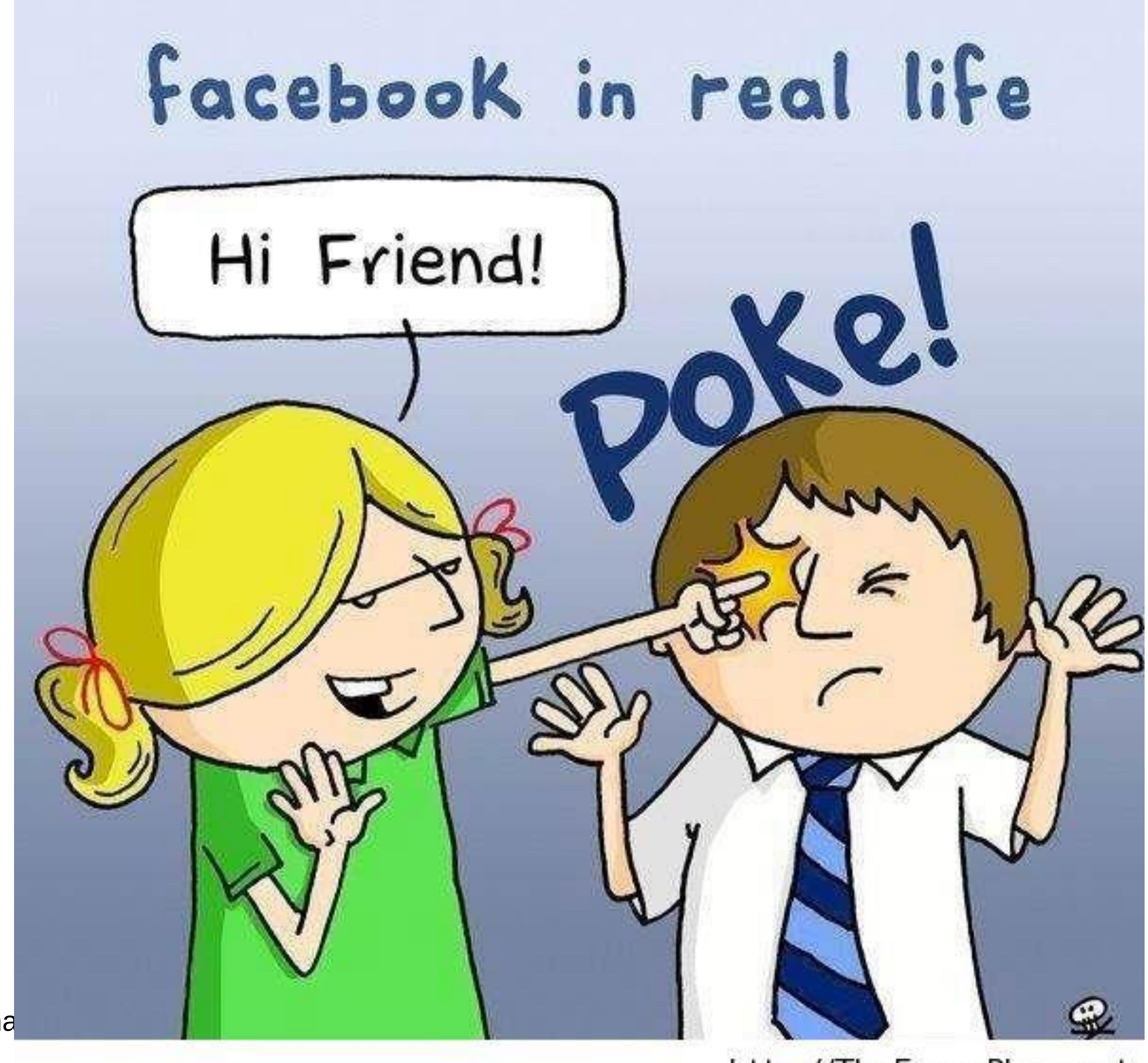
add(i,e)

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)


facebook in real life
Hi Friend!
Poke!
http://TheFunnyPlace.net