Data structures and Algorithms

Lecture 2a: Lists

[GT 2.2]

Dr. Julian Mestre

School of Computer Science

THE UNIVERSITY OF
SYDNEY

Abstract Data Types (ADT)

Type defined in terms of its data items and associated operations, not its implementation.

ADTs are supported by many languages, including Java and Python.

Abstract Data Types (ADT)

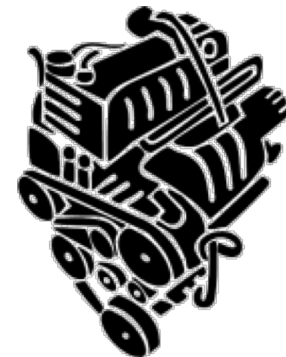Type defined in terms of its data items and associated operations, not its implementation.

Simple example: Driving a car

interface

implementation

Benefits of ADT approach

- Code is easier to understand if different issues are separated into different places.

- Client can be considered at a higher, more abstract, level.

- Many different systems can use the same library, so only code tricky manipulations once, rather than in every client system.

- There can be choices of implementations with different performance tradeoffs, and the client doesn't need to be rewritten extensively to change which implementation it uses.

Example: Reservation system

– We have a theatre with
500 named seats, e.g., "N31"

– What kind of data should be stored?

   – Seats names

   – Seats reserved or available.

   – If reserved, name of the person who reserved the seat.

– Operations needed?
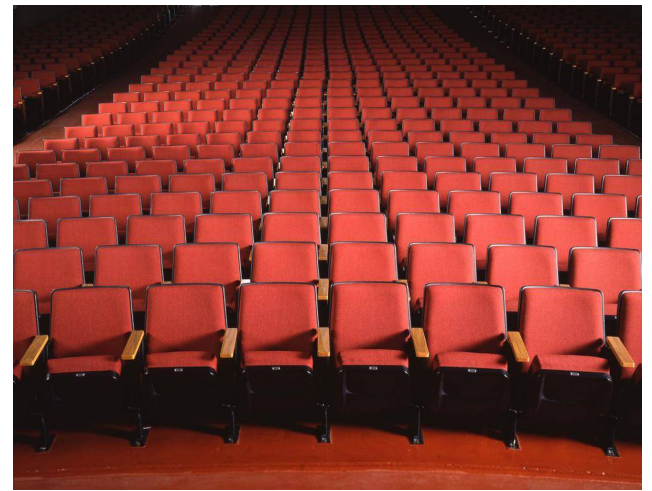
Example: Reservation system
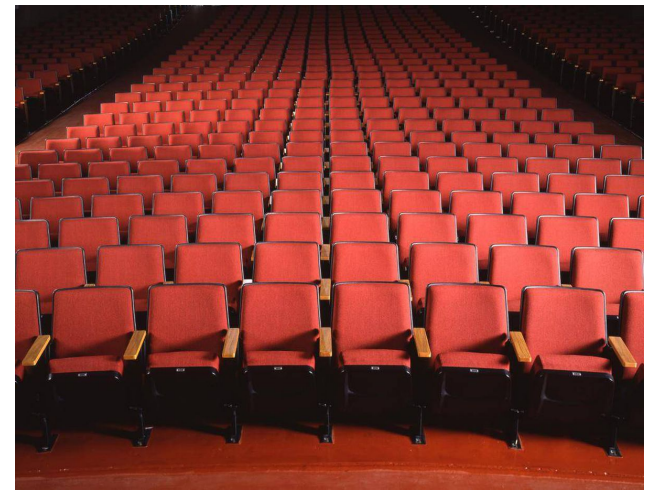
–   Operations needed?

  –   capacity_available() : number of available seats (integer)

  –   capacity_sold() : number of seats with reservations

  –   customer(x) : name of customer who bought seat x

  –   release(x) : make seat x available (ticket returned)

  –   reserve(x, n) : customer n buys ticket for seat x

  –   add(x) : install new seat whose name is x

ADT challenges

- Specify how to deal with the boundary cases
    - what to do if reserve(I, n) is invoked when I is already occupied?
    - what other cases can you think of?

- Do we need a new ADT? Could we use an existing one, perhaps by renaming the operations and tweaking the error-handling?
    - "Adapter" design pattern (see SOFT2201)
    - Could this example be mapped to an ADT you already know?

Abstract data types and Data structures

An abstract data type (ADT) is a specification of the desired behaviour from the point of view of the user of the data.

A data structure is a concrete representation of data, and this is from the point of view of an implementer, not a user.

Distinction is subtle but similar to the difference between a computational problems and an algorithm.

ADT in programming (Java)

–   ADT is given as an *interface*

–   An interface declares methods (with their names and signatures) but it does not contain code for the methods

–   Data structure implementation is a class that implements the interface and has a constructor to create objects.

–   Client code declares variables whose type implements the interface. Can call methods implemented by the underlying class

ADT in programming (Python)

–   ADT is given as an *abstract base class (abc)*

–   An abc declares methods (with their names and signatures) usually without providing code and we can't construct instances

–   A data structure implementation is a class that inherits from the abc, provides code for all the required methods (and perhaps others) and has a constructor

–   Client code can have variables that are instances of the data structure class and can call methods on these variables

Pseudocode

- – High-level description of an algorithm
- – More structured than English prose
- – Less detailed than a program
- – Preferred notation for describing algorithms
- – Hides program design issues

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  def **method** (**arg** [, **arg**…])

  Input …

  Output …

- Method call

  method (*arg* [, *arg*…])

- Return value

  return *expression*

- Expressions:

  ← : Assignment

  = : Equality testing

Pseudocode example

Pseudocode is not for a specific programming language, it is a simple way of describing a set of **instructions** that does not have to use specific syntax.

Greatest Common Divisor - Java

```java
public static int GCD(int a, int b) {
    if(b == 0){
        return a;
    }
    return GCD(b, a % b);
}
```

Pseudocode example

Pseudocode is not for a specific programming language, it is a simple way of describing a set of **<u>instructions</u>** that does not have to use specific syntax.

Greatest Common Divisor - Python

```python
def gcd(a,b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Pseudocode example

Pseudocode is not for a specific programming language, it is a simple way of describing a set of **instructions** that does not have to use specific syntax.

Greatest Common Divisor - Pseudocode

```
def gcd(a, b)
    if b = 0 then
        return a
    else
        return gcd(b, a mod b)
```

Index-Based Lists (List ADT)

An index-based list (usually) supports the following operations:

size()   (int) number of elements in the store

isEmpty()   (boolean) whether or not the store is empty

get(i)   return element at index i

set(i, e)   replace element at index i with element e,
              and return element that was replaced

add(i, e)   insert element e at index i existing elements with
              index $>=$ i are shifted up

remove(i)   remove and return the element at index i existing
              elements with index $\geq$ i are shifted down

Example

A sequence of List operations:

Method	Returned value	List content
add(0,A)	-	[A]
add(0,B)	-	[B, A]
get(1)	A	[B, A]
set(2,C)	"error"	[B, A]
add(2,C)	-	[B, A, C]
add(4,D)	"error"	[B, A, C]
remove(1)	A	[B, C]
add(1,D)	-	[B, D, C]
add(1,E)	-	[B, E, D, C]
get(4)	"error"	[B, E, D, C]
add(4,F)	-	[B, E, D, C, F]
set(2,G)	D	[B, E, D, C, F]

List ADT: Example execution

Suppose mylist = [3, 4, 8, 1]

What is the result of executing the following sequence of operations?

operation	output	mylist
mylist.size()	4	[3, 4, 8, 1]
mylist.isEmpty()	False	[3, 4, 8, 1]
mylist.get(2)	8	[3, 4, 8, 1]
mylist.set(1, 5)	None	[3, 5, 8, 1]
mylist.size()	4	[3, 5, 8, 1]
mylist.remove(3)	None	[3, 5, 8]

Implementing an indexed List ADT

What sort of data structure should we use for a List ADT?

Any suggestions?

Array-based Lists

An option for implementing the list ADT is to use an array A, where $A[i]$ stores (a reference to) the element with index i.

If array has size N then we can represent lists of size $n < N$

allocated memory
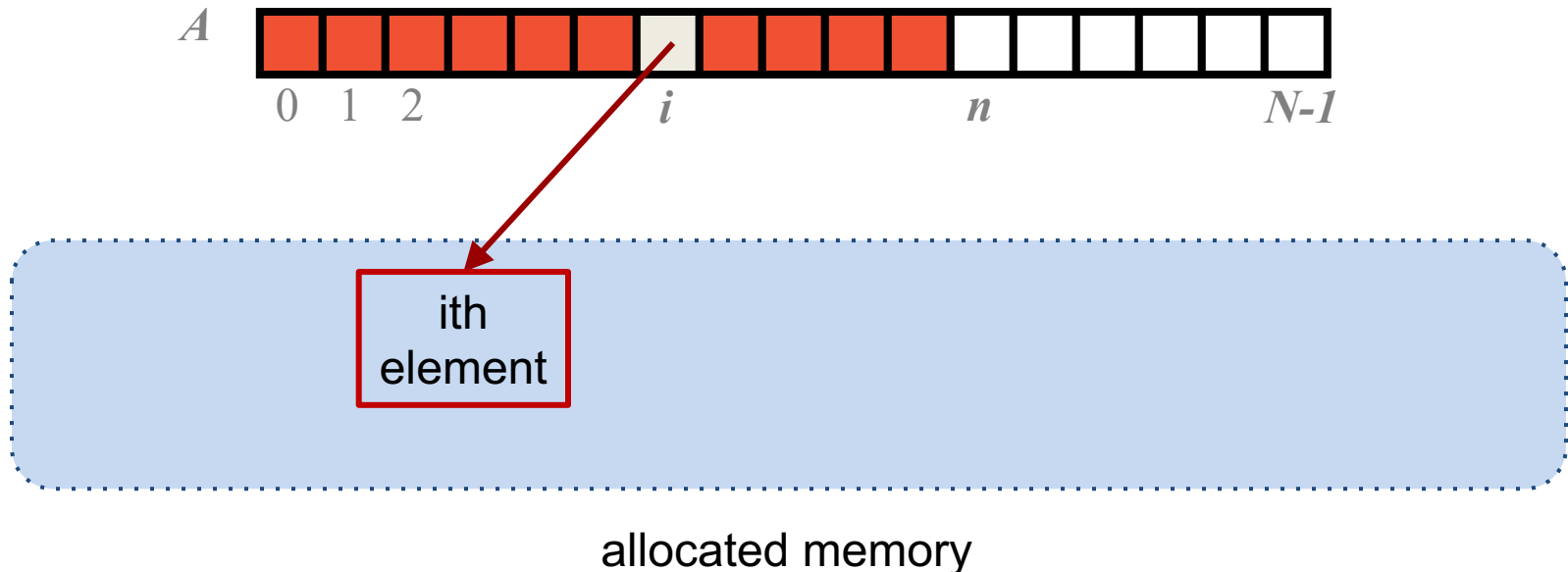
Array-based Lists: get(i)

The get(i) and set(i, e) methods are easy to implement by accessing A[i]

Must check that i is a legitimate index ($0 \leq i < n$)

Both operations can be carried out in in constant time (a.k.a. $O(1)$ time), independent of the size of the array

allocated memory
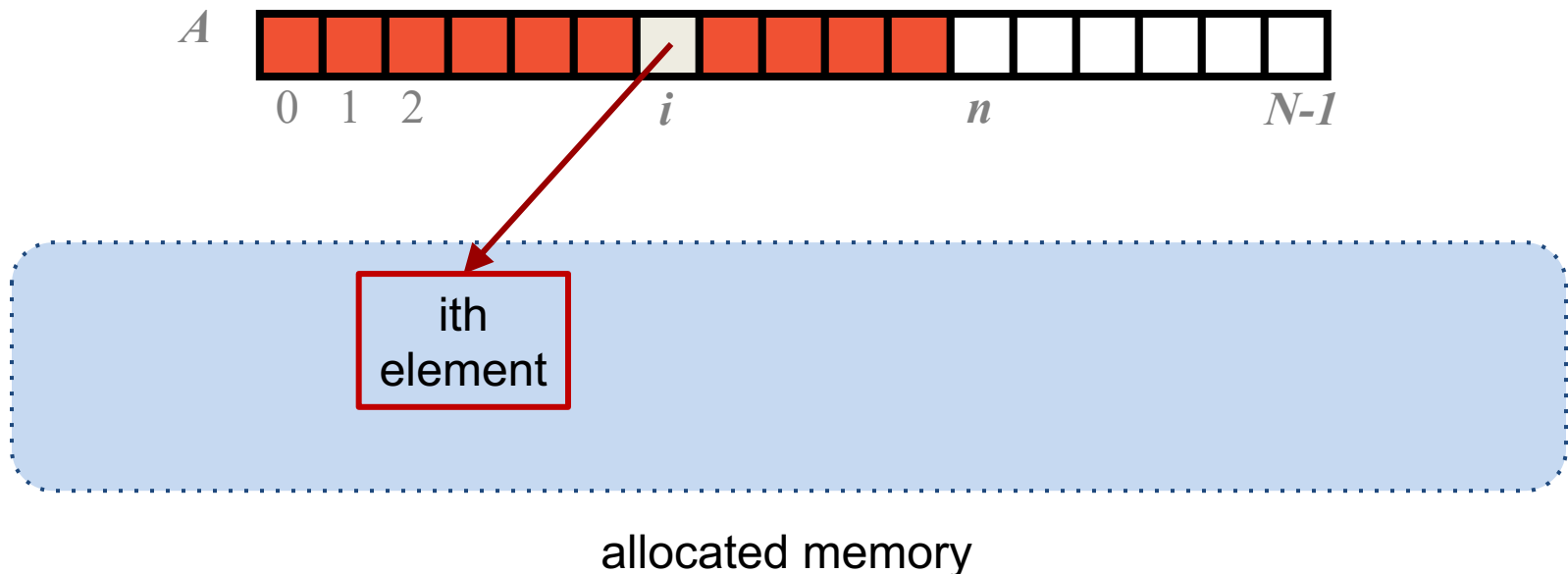
Array-based Lists: set(i,e)

The get(i) and set(i, e) methods are easy to implement by accessing A[i]

Must check that i is a legitimate index ($0 \le i < n$)

allocated memory
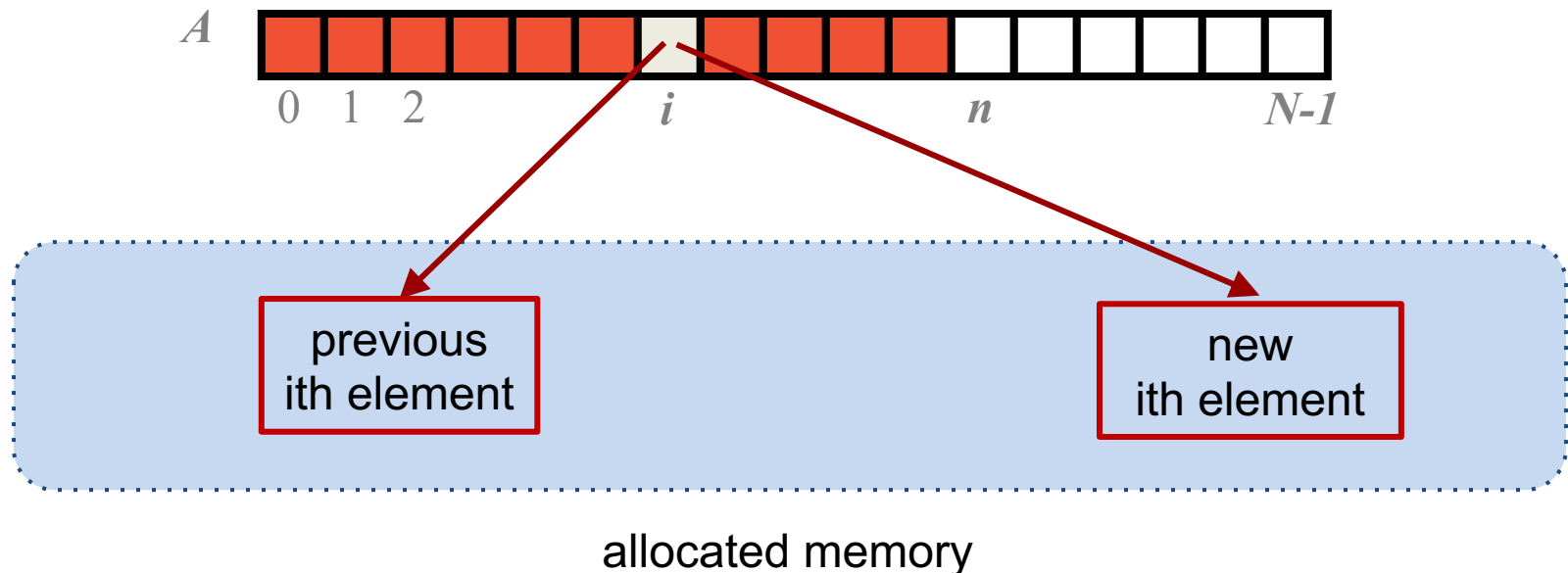
Pseudo-code for set and get

```
def get(i):
    if i < 0 or i >= n then
        return "index out of bound"
    else
        return A[i]

def set(i, e):
    if i < 0 or i >= n then
        return "index out of bound"
    A[i] ← e
```

Time complexity of both operations is $O(1)$ time, independent of the size of the array

Array-based Lists: add(i,e)

In an operation add(i, e), we must make room for the new element by shifting forward n - i elements A[i], ..., A[n - 1]

Must check that there is space (n < N)

What is the most time consuming scenario?
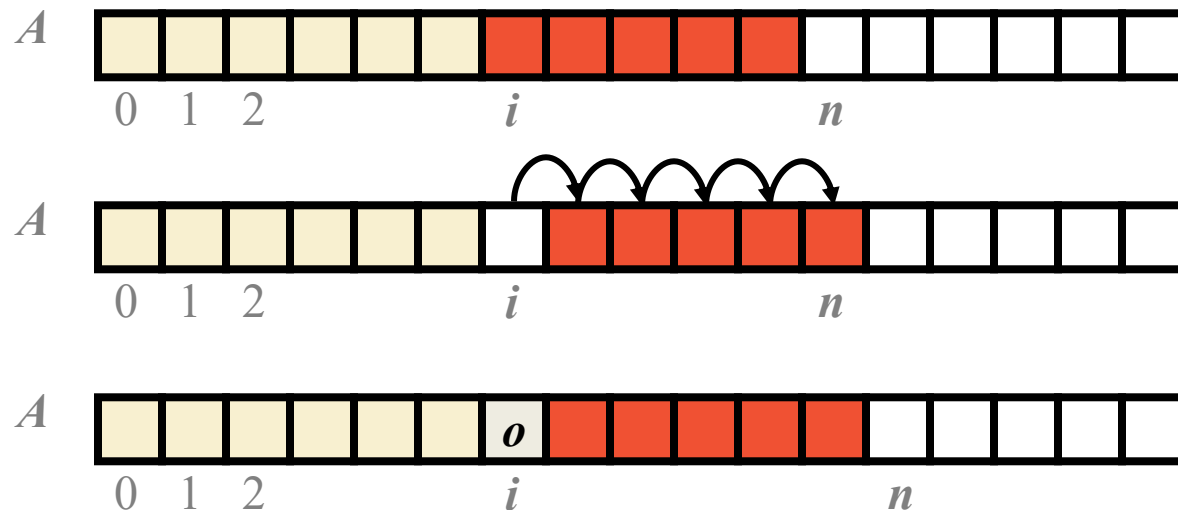
Pseudo-code for insertion

```
def add(r, e):
    if n = N then
        return "array is full"
    if r < n then
        for i in [n-1, n-2, … , r] do
            A[i + 1] ← A[i]
    A[r] ← e
```

Time complexity is O(n) in the worst case

Array-based Lists: remove(i)

In an operation remove(i), we need to fill the hole left at position i by shifting backward n - i - 1 elements A[i + 1], …, A[n - 1]

Must check that i is a legitimate index (0 ≤ i < n)
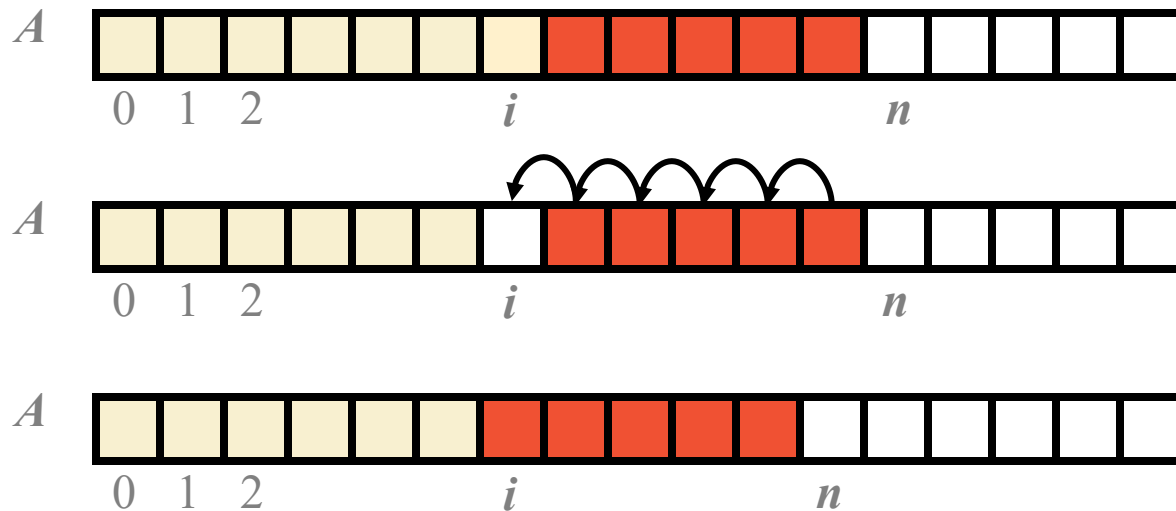
Pseudo-code for removal

```
def remove(i):
    if i < 0 or i >= n
        return "index out of bound"
    e ← A[i]
    if i < n-1
        for j in [r, r+1, … , n-2] do
            A[j] ← A[j+1]
    return e
```

Time complexity is O(n) in the worst case

Summary of (static) array-based Lists

Limitations:

- can represent lists up to the capacity of the array (n vs N)

Space complexity:

- space used is $O(N)$, whereas we would like it to be $O(n)$

Time complexity:

- both get and set take $O(1)$ time
- both add and remove take $O(n)$ time in the worst case

Dynamic array

Instead of using an array of fixed length, we can get a larger array each time we run out of space, and copy existing element there

This approach is called an dynamic array because the size of the array changes dynamically throughout the execution

How the Java ArrayList class and Python list type are implemented

Copying elements to new array makes the add operation sometimes much slower, but worst-case time is still $O(n)$

We can also shrink array if we want to be memory efficient

Need to be careful regarding how often we change array

Summary of dynamic array-based Lists

Space complexity:

– space used is $O(n)$

Time complexity:

– both get and set take $O(1)$ time

– both add and remove take $O(n)$ time in the worst case

Positional Lists
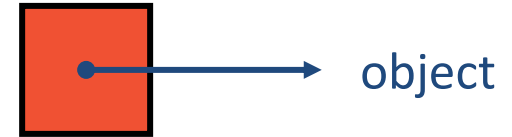
object

ADT for a list where we store elements at "positions"

Position models the abstract notion of place where a single object is stored within a container data structure. Giving a a unified view of diverse ways of storing data:

- a **cell** of an array implementation of a List

- a **node** in a linked implementation of a List

Unlike index, this keeps referring to the same entry even after insertion/deletion happens elsewhere in the collection.

Position offers just one method:

    element() : return the element stored at the position instance

Positional Lists – Operations

size()   (int) number of elements in the store

isEmpty()   (boolean) whether or not the store is empty

first()   return position of first element (null if empty)

last()   return position of last element (null if empty)

before(p)   return position immediately before p (null if p is first)

after(p)   return position immediately after p (null if p last)

insertBefore(p, e) insert e in front of the element at position p

insertAfter(p, e)   insert e following the element at position p

remove(p)   remove and return the element at position p

Singly Linked List

- A concrete data structure

- A sequence of Nodes, each with a reference to the next node

- List captured by reference (head) to the first Node
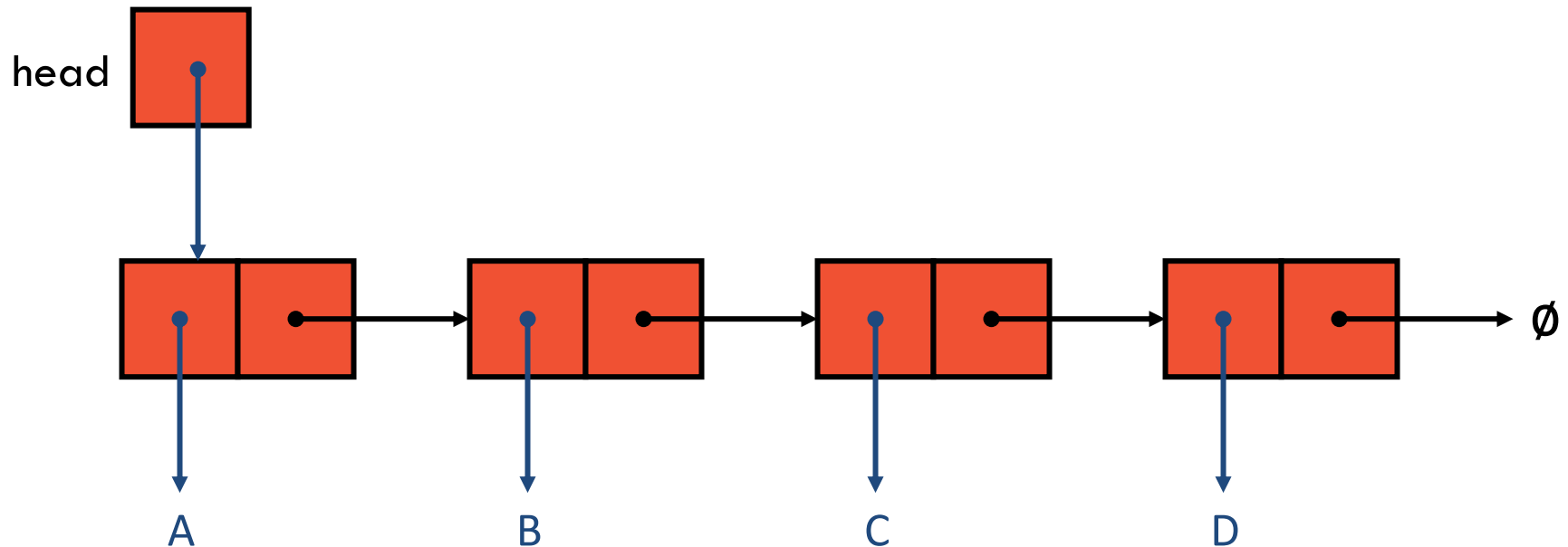
Node implements Position

Each Node in a singly linked List stores
- its element, and
- a link to the next node.

n: Node

next

element
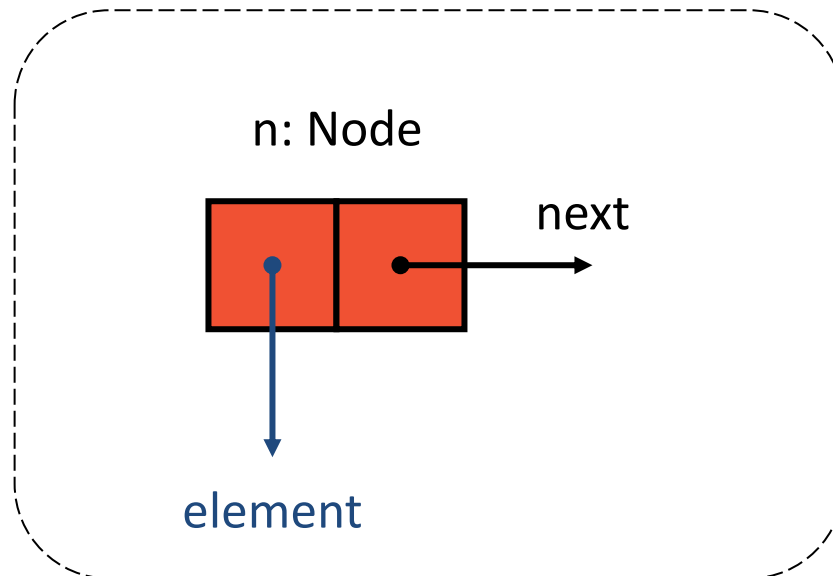
Advice on working with linked structures

– Draw the diagram showing the state.

– Show a location where you place carefully each of the instance variables (including references to nodes).

– Be careful to step through dotted accesses e.g. p.next.next

– Be careful about assignments to fields e.g.
    p.next = q or p.next.next = r
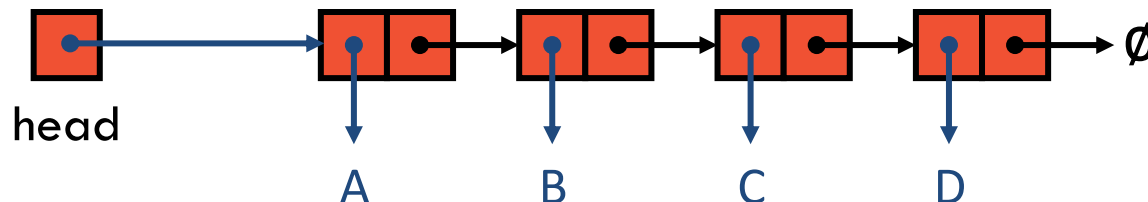
first()

first() : return position of first element (null if empty)

return?

head

A   B   C   D
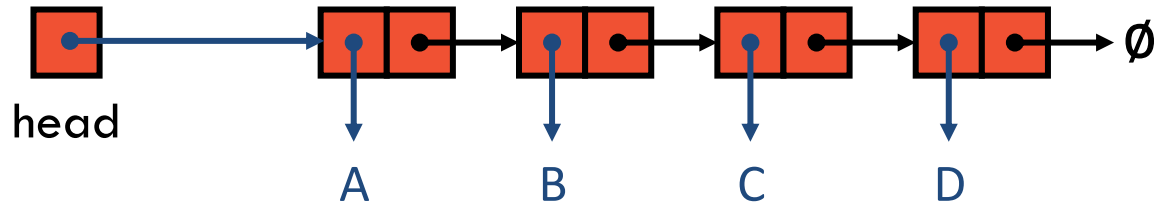
first()

first() : return position of first element (null if empty)

return head

Time complexity?

insertFirst(e)

1. Instantiate a new node *n*
2. Set e as element of n
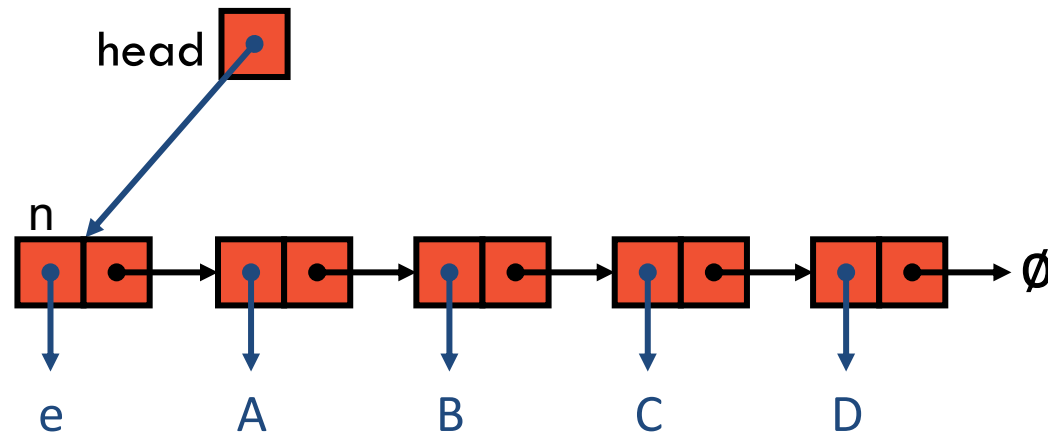3. Set n.next to point to (old) head
4. Update list's head to point to n

insertFirst(e)

1. Instantiate a new node **n**
2. Set e as element of n
3. Set n.next to point to (old) head
4. Update list's head to point to n

insertFirst(e)

1. Instantiate a new node **n**
2. Set e as element of n
3. Set n.next to point to (old) head
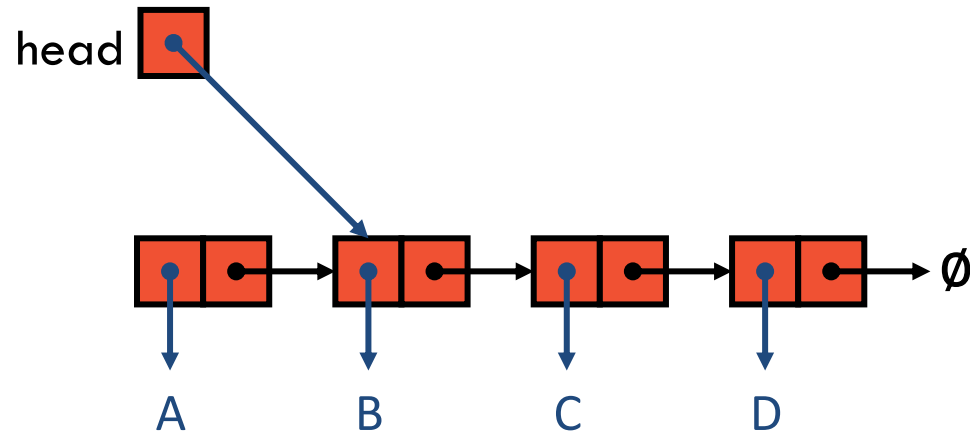4. Update list's head to point to n

insertFirst(e)

1. Instantiate a new node **n**
2. Set e as element of n
3. Set n.next to point to (old) head
4. Update list's head to point to n

insertFirst(e)

1. Instantiate a new node **n**
2. Set e as element of n
3. Set n.next to point to (old) head
4. Update list's head to point to n

What is the time complexity?

insertFirst(e)

1. Instantiate a new node **n**
2. Set e as element of n
3. Set n.next to point to (old) head
4. Update list's head to point to n

What is the time complexity? O(1)

removeFirst()

1. Update head to point to next node
2. Delete the former first node
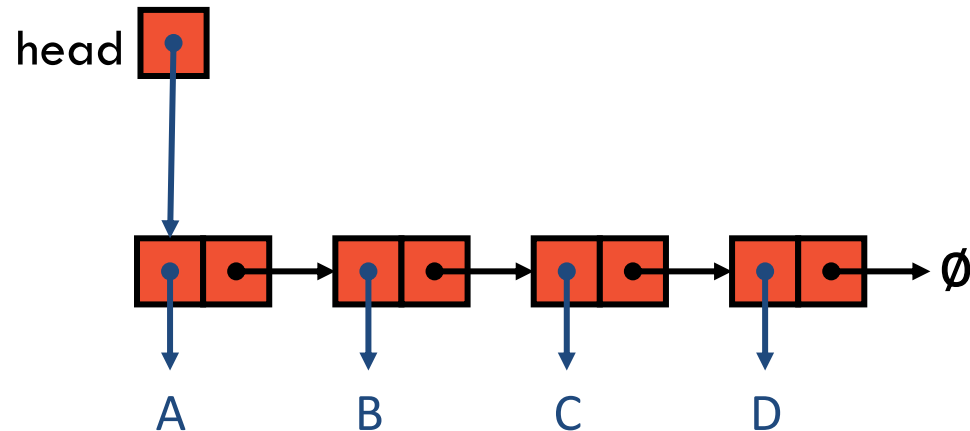
removeFirst()

1. Update head to point to next node
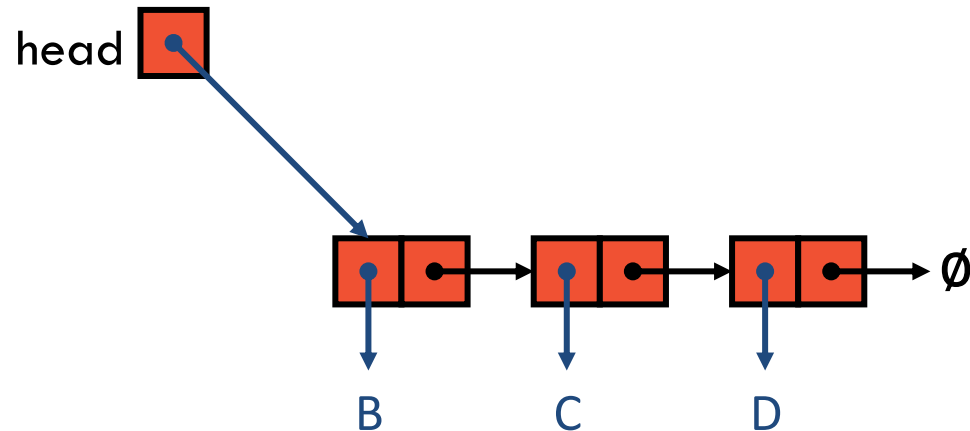2. Delete the former first node

removeFirst()

1. Update head to point to next node
2. Delete the former first node
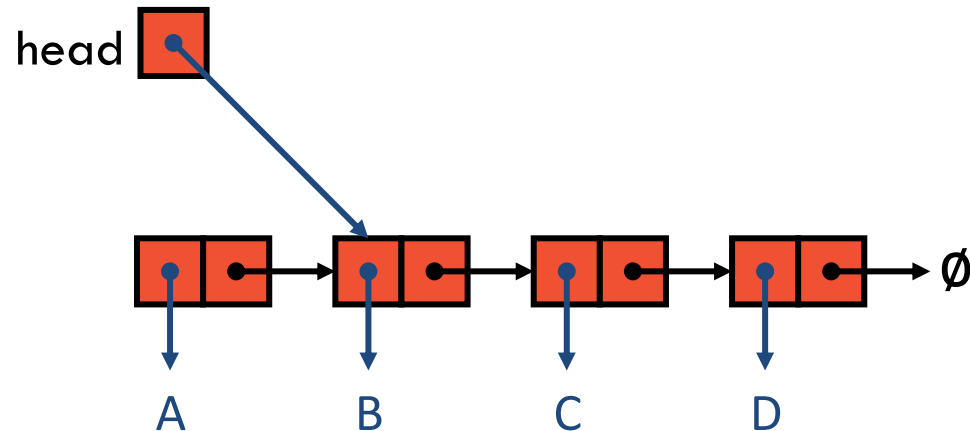
removeFirst()

1. Update head to point to next node
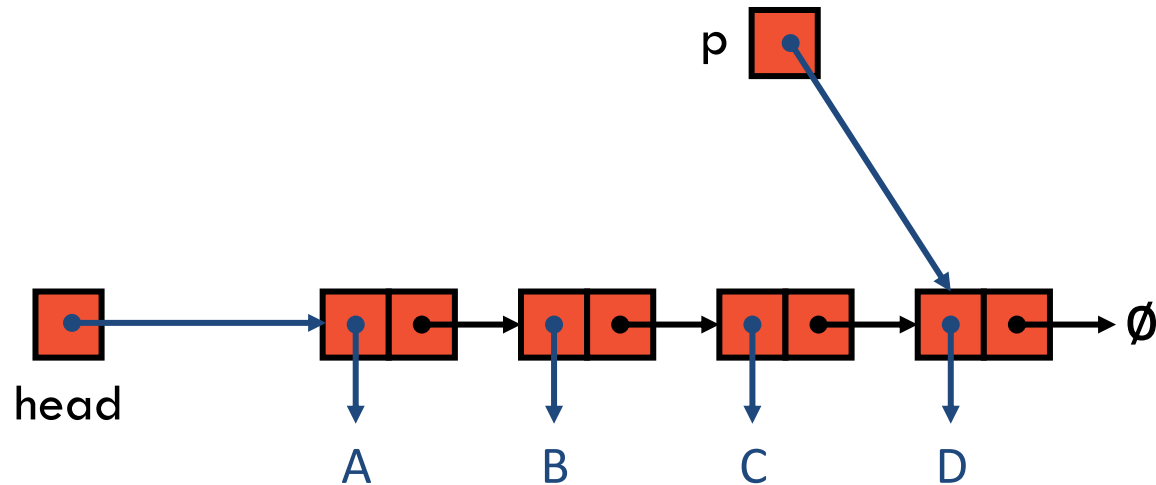2. Delete the former first node
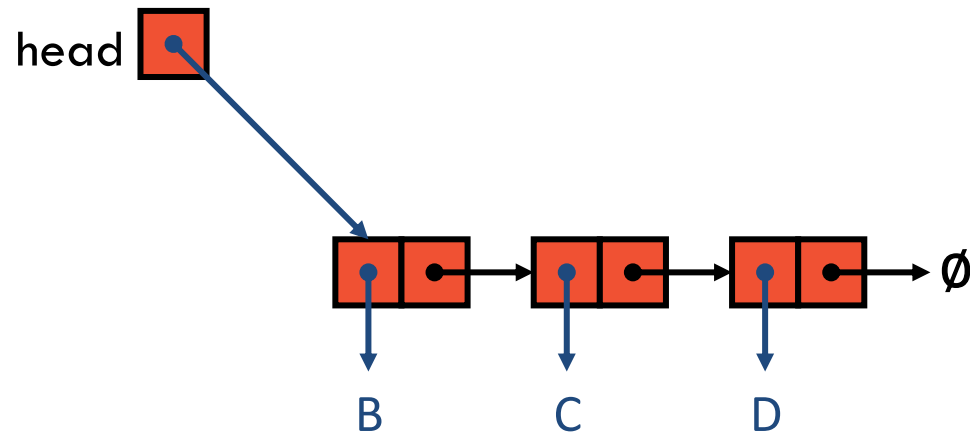
Time complexity?

removeFirst()

1. Update head to point to next node
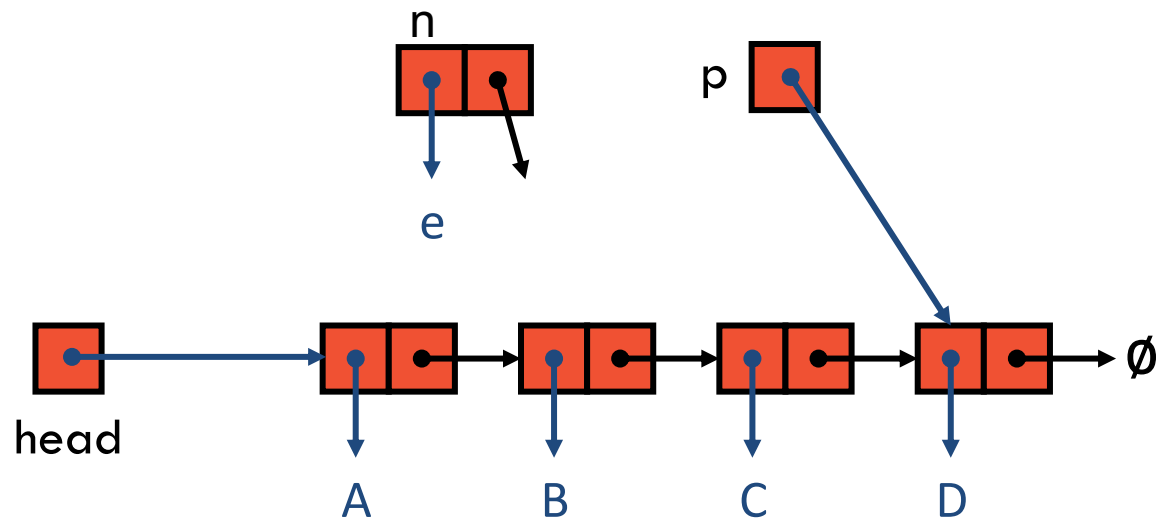2. Delete the former first node
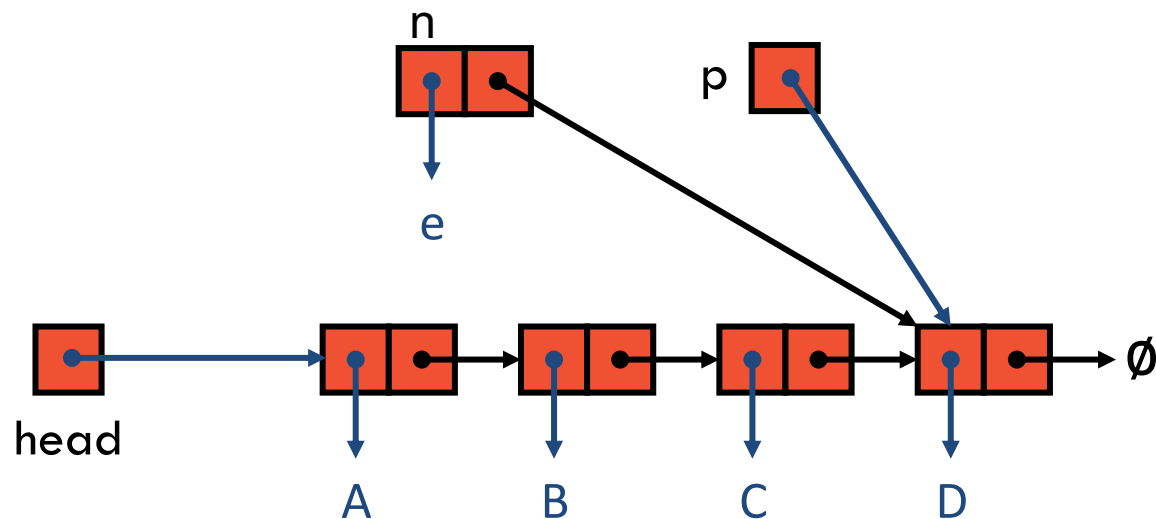
Time complexity? O(1)

insertBefore(p,e)

insertBefore(p,e) : insert e in front of the element at position p

insertBefore(p,e)

insertBefore(p,e) : insert e in front of the element at position p
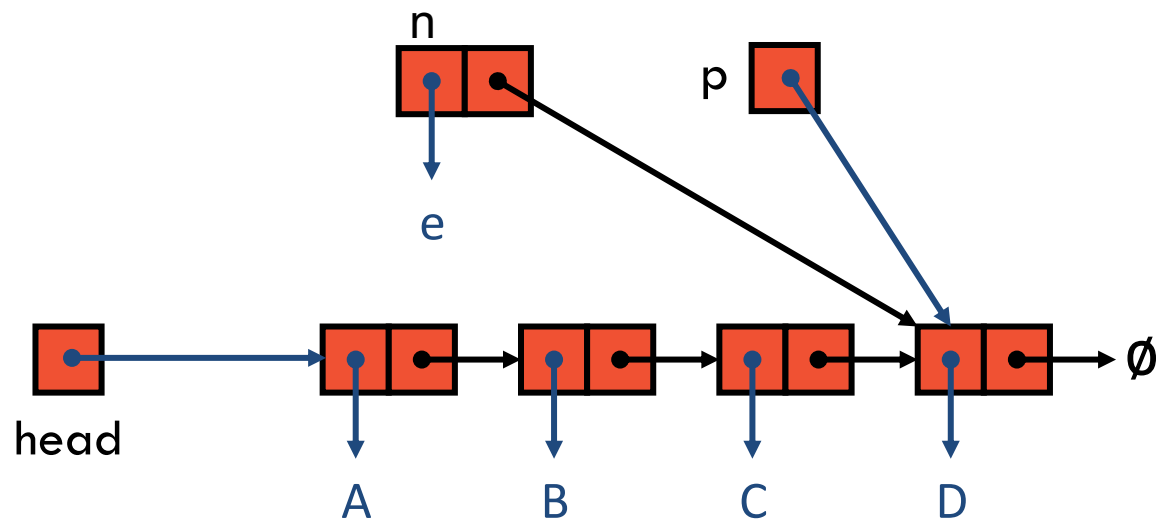
insertBefore(p,e)

insertBefore(p,e) : insert e in front of the element at position p
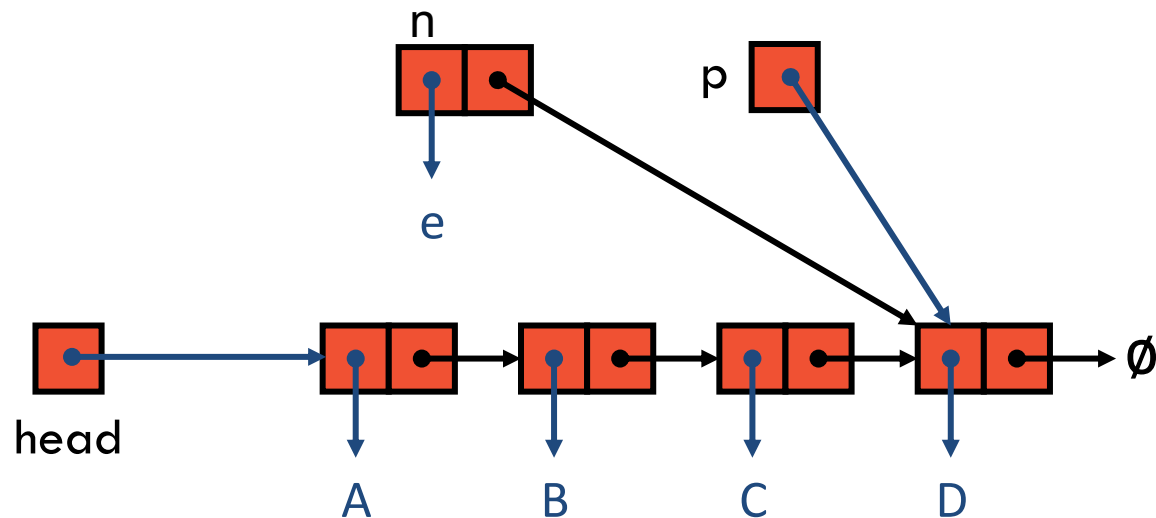

What's the next step?

insertBefore(p,e)

insertBefore(p,e) : insert e in front of the element at position p

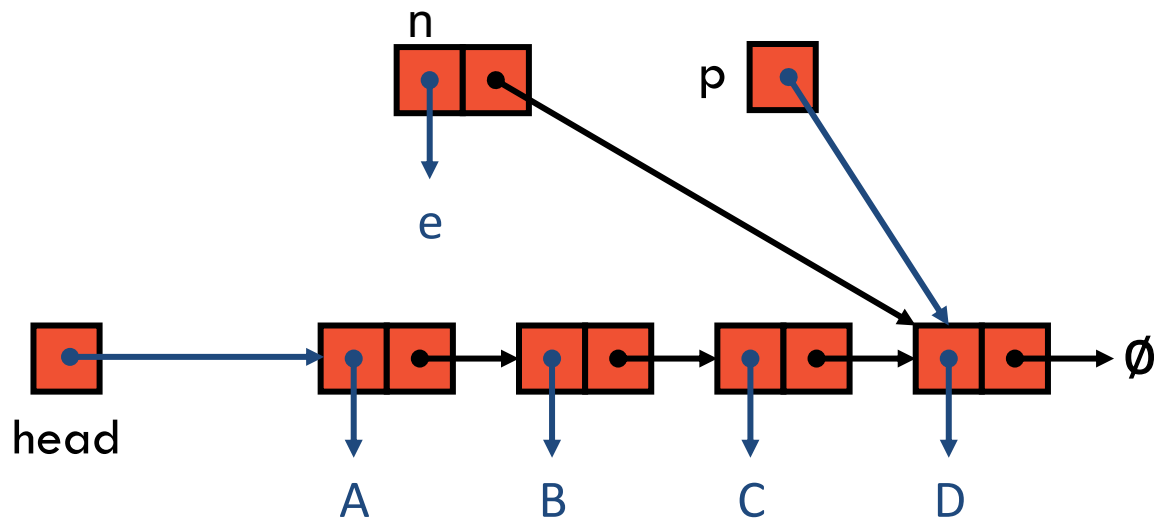**What's the next step?** Find the predecessor of n. How?

insertBefore(p,e)

To find the predecessor of p we need to follow the links from the "head". Time complexity?
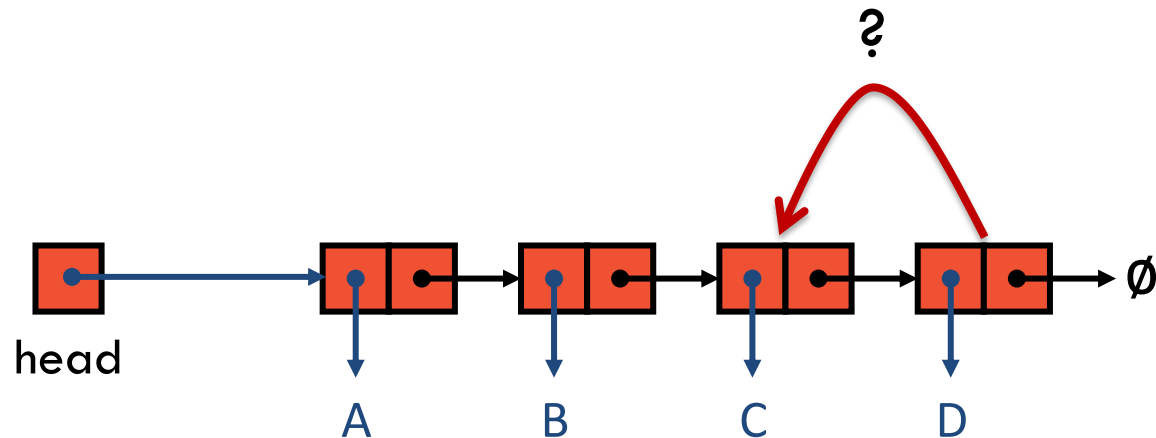
insertBefore(p,e)

To find the predecessor of p we need to follow the links from the "head". Time complexity: O(n)

insertBefore(p,e)

– There is no constant-time way to find the predecessor of a node in a Singly Linked List.
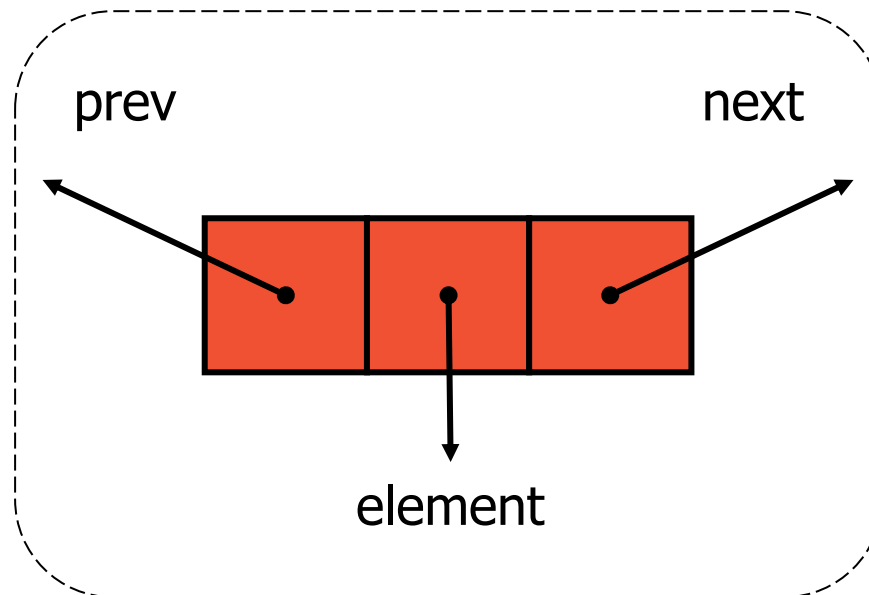
Another attempt

A very natural way to implement a positional list is with a doubly-linked list, so that it is easy/quick to find the position before.

Each Node in a Doubly Linked List stores

–   its element, and
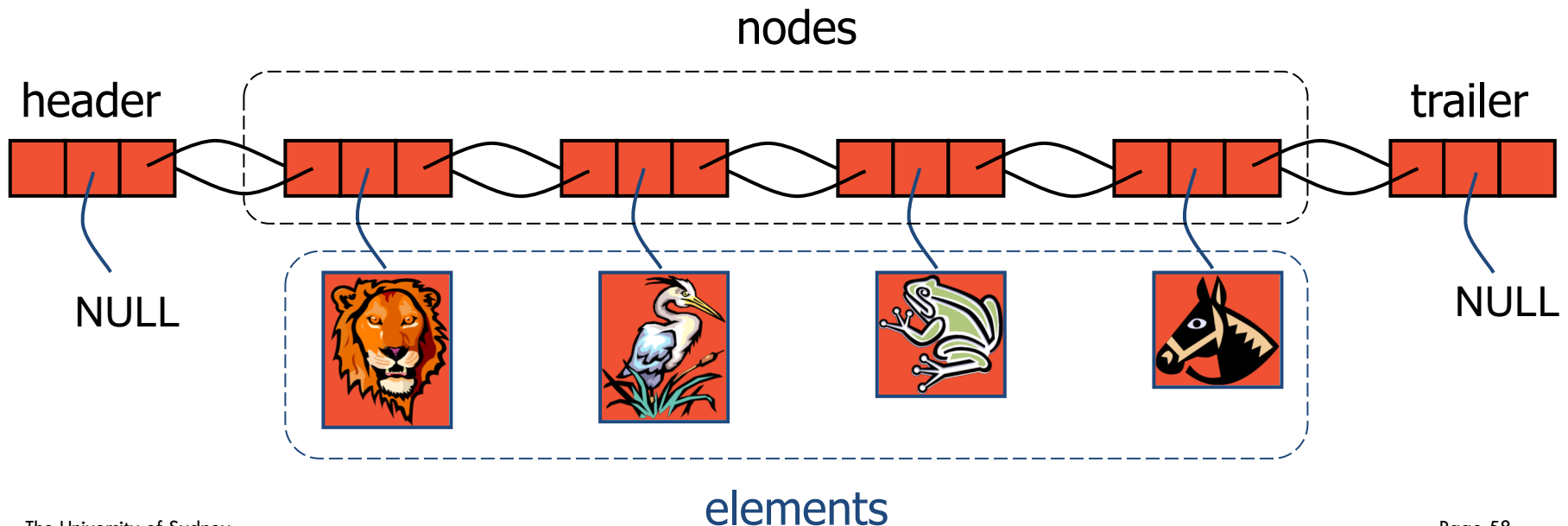
–   a link to the previous and next nodes.

prev   next

element

Doubly Linked Lists

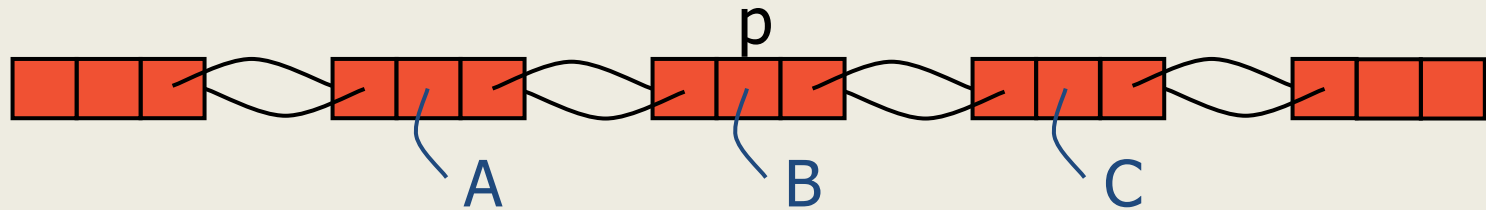A concrete data structure
- A sequence of Nodes, each with reference to prev and to next
- List captured by references to its Sentinel Nodes

nodes
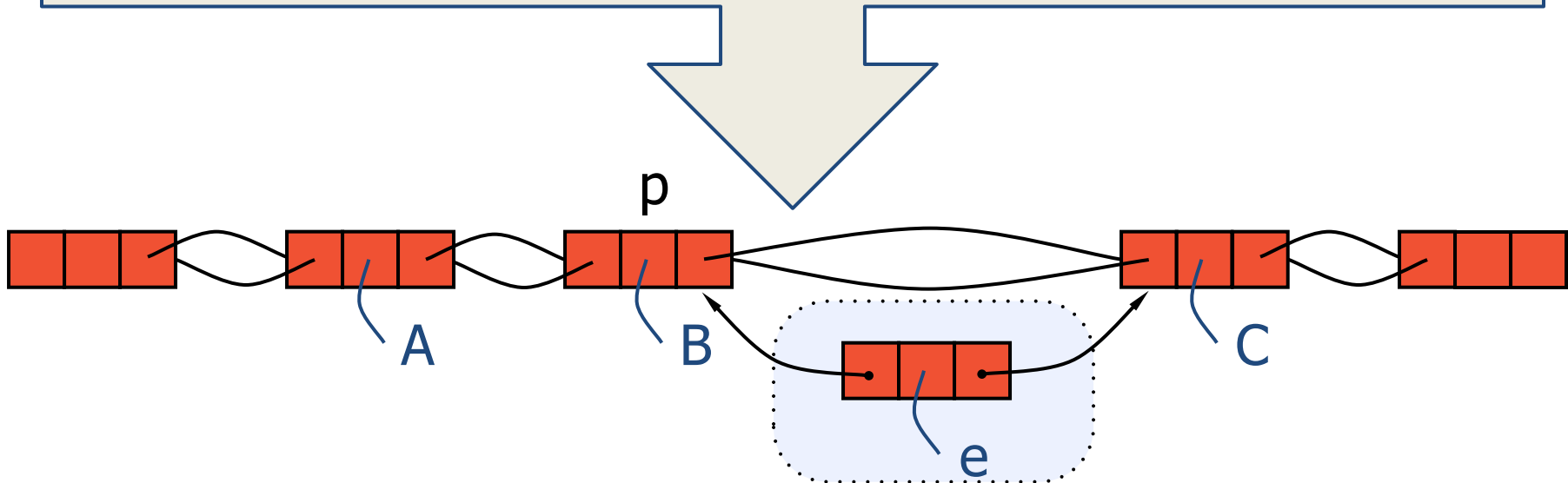
header

trailer

NULL

NULL

elements

insertAfter(p,e) – step 1

Instantiate new Node v with element set to e.
Update v.previous to point to p and v.next to point to p.next.

insertAfter(p,e) – step 2

Update p.next.prev ← p and p.next ← v
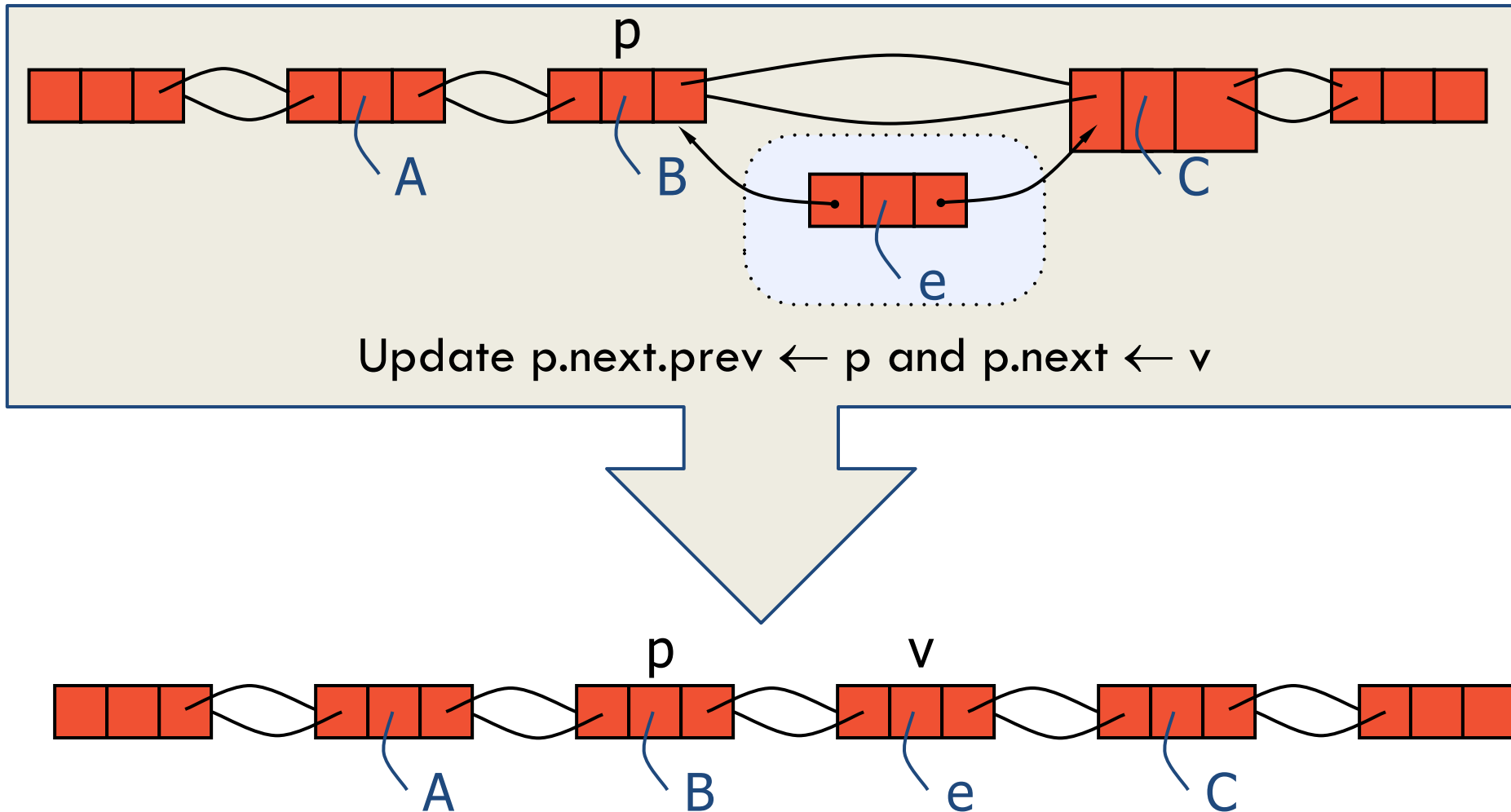
insertAfter(p,e)

– Insert a new node with element e between p and its successor.
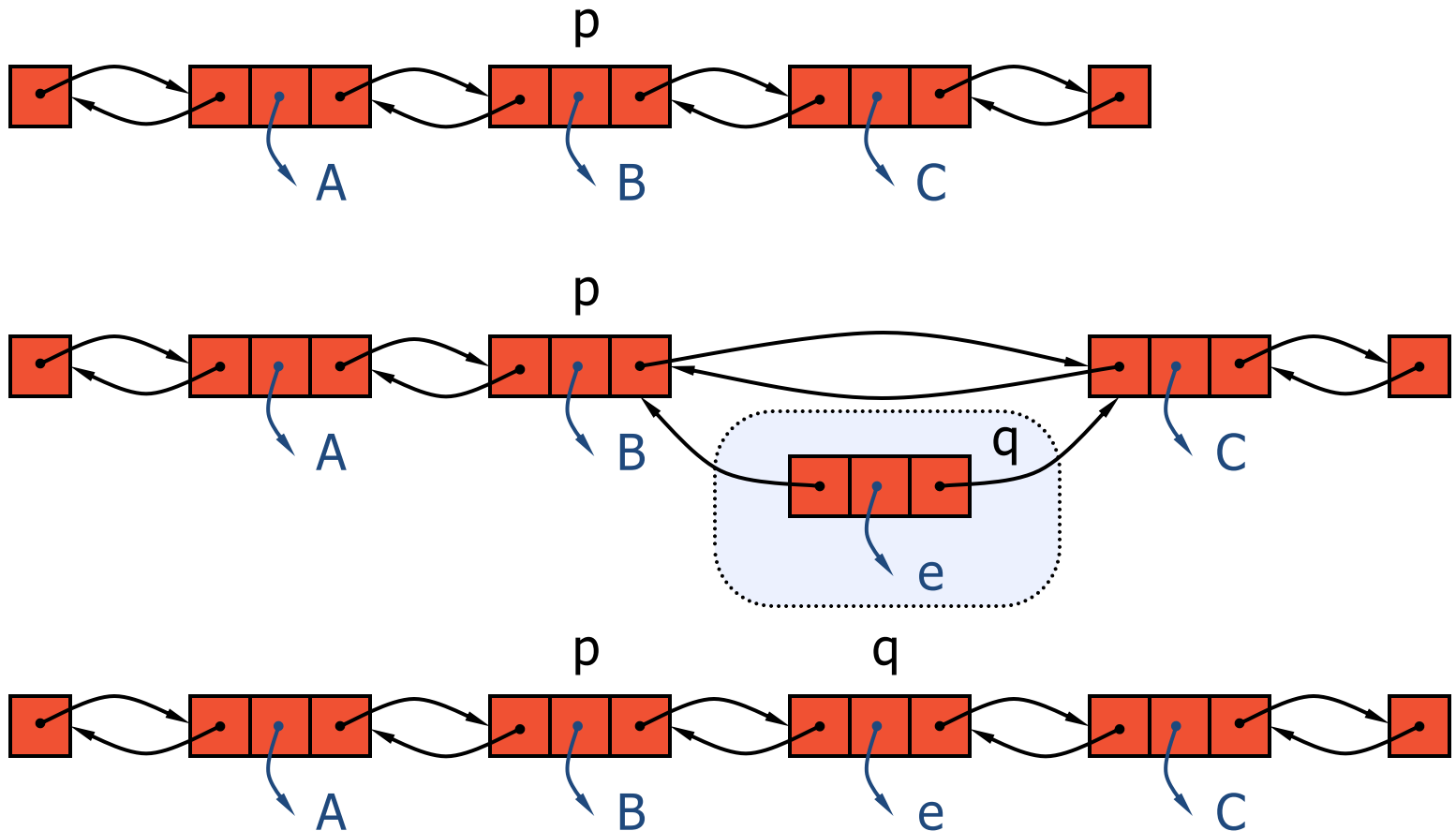
Pseudo-code

```
def insert_after(pos, elem):
    // insert elem after pos
    // assuming it is a legal pos

    new_node ← create a new node
    new_node.element ← elem
    new_node.prev ← pos
    new_node.next ← pos.next
    pos.next.prev ← new_node
    pos.next ← new_node

    return new_node
```

remove(p)

- Remove a node, p, from a doubly-linked list.
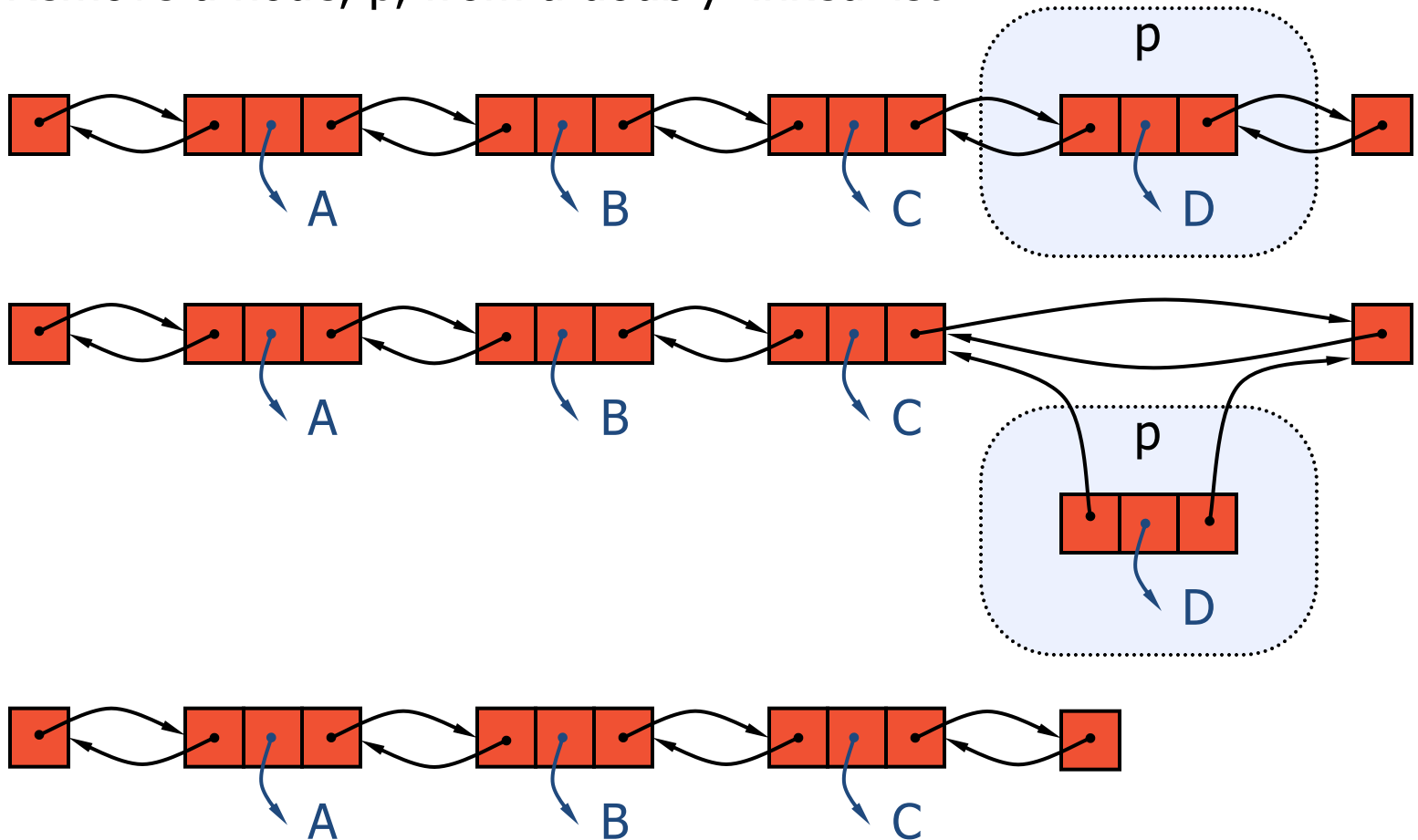
Pseudo-code

```
def remove(pos):
    // remove pos from the list
    // assuming it is a legal pos

    pos.prev.next ← pos.next
    pos.next.prev ← pos.prev

    return pos.element
```

Performance

– A linked list can perform all of the accessor and update operations for a positional list in constant time.

– Space complexity is O(n)

Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insertBefore(p, e)	$O(1)$
insertAfter(p, e)	$O(1)$
remove(p)	$O(1)$

+ size() and isEmpty() in O(1)

Array or Linked List implementation?

Linked List

- good match to positional ADT
- efficient insertion and deletion
- simpler behaviour as collection grows
- modifications can be made as collection iterated over
- space not wasted by list not having maximum capacity

Arrays

- good match to index-based ADT
- caching makes traversal fast in practice
- no extra memory needed to store pointers
- allow random access (retrieve element by index)

Iterators

Abstracts the process of stepping through a collection of elements one at a time by extending the concept of position

Implemented by maintaining a cursor to the "current" element

Two notions of iterator:

- snapshot freezes the contents of the data structure (unpredictable behaviour if we modify the collection)
- dynamic follows changes to the data structure (behaviour changes predictably)

Iterator ADT (in Java)

hasNext()   checks that the collection is non-empty and
that the cursor is not after the last element

next()   returns the element after the cursor (cursor advances)

```java
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
}

while (iterator.hasNext()){
    E value = iterator.next();
    // do something using value
}
```

Iterables types (in Java)

Iterables are objects with an iterator() method

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```

Java has a convenient syntax for using the iterator associated with an Iterable class

```
List<Integer> values;
int sum = 0;
for (Integer i : values) {
    sum += i;    // implicit conversion
}
```

ListIterator (Java) example

add(e) add e at the current cursor position

set(e) replace element the current cursor position with e

remove() remove current position

next() move cursor forward and return element it points to

hasNext() check whether there is a position after the cursor

previous() move cursor backwards and return element it points to

hasPrevious() check whether there is an element before the cursor

Iterators in Python

iter(obj) returns an iterator of the object collection

To make a class is iterable define the method __inter__(self)

The method __iter__() returns an object having a next() method

Calling next() returns the next object and advances the cursor or raises StopIteration()

Iterators in Python

```python
for x in obj:
    // process x
```

Is equivalent to:

```python
it = x.__iter__()
try:
    while True:
        x = it.next()
        // process x
except StopIteration:
    pass
```