COMP2823

Lecture 2a: Lists

[GT 2.2]

Dr. André van Renssen

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 Python.

Abstract Data Types (ADT)

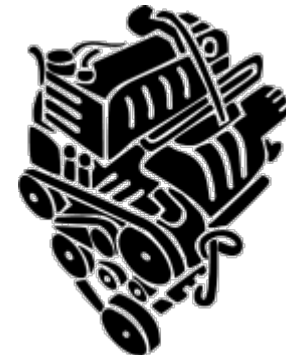Type defined in terms of its data items and associated operations, not its implementation.

Simple example: Driving a car

interface

implementation

Benefits of ADT approach

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

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

- Clients using the ADT need to learn the interface only once

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?
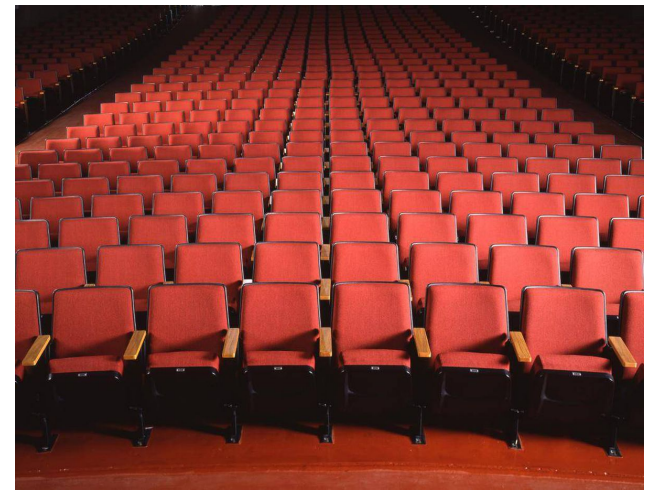
Example: Reservation system
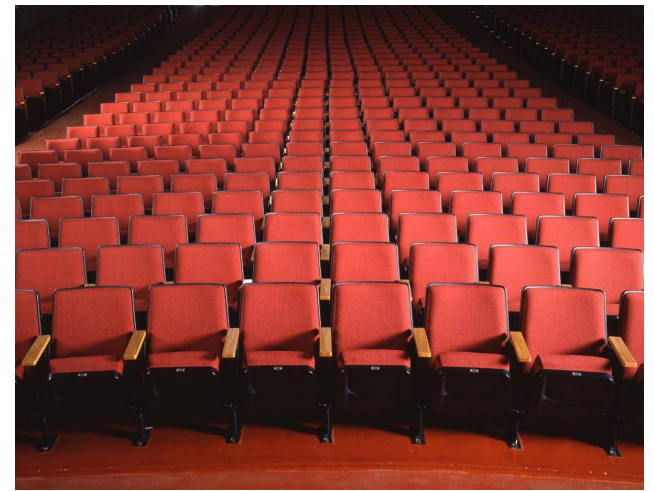
– Operations needed?

  – 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, y) : customer y buys ticket for seat x

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

  – get_available(): access available seats

ADT challenges

- Specify how to deal with the boundary cases
  - what to do if reserve(x, y) is invoked when x 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 (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

Index-Based Lists (List ADT)

An index-based list 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

insert(i, e) insert element e at index i existing elements with
               index $\geq$ 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
insert(0,A)	-	[A]
insert(0,B)	-	[B, A]
get(1)	A	[B, A]
set(2,C)	"error"	[B, A]
insert(2,C)	-	[B, A, C]
insert(4,D)	"error"	[B, A, C]
remove(1)	A	[B, C]
insert(1,D)	-	[B, D, C]
insert(1,E)	-	[B, E, D, C]
get(4)	"error"	[B, E, D, C]
insert(4,F)	-	[B, E, D, C, F]
set(2,G)	D	[B, E, G, C, F]

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
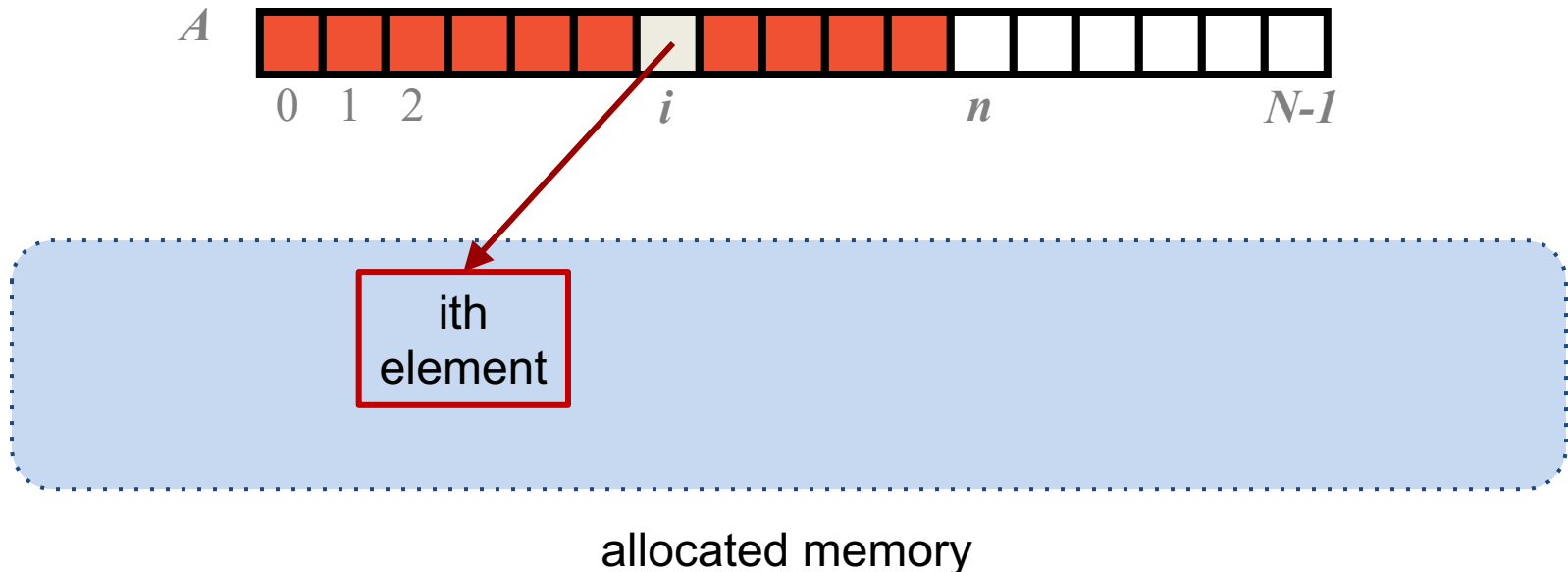
Array-based Lists: get(i)

The get(i) and set(i, e) methods are easy to implement by accessing A[i]
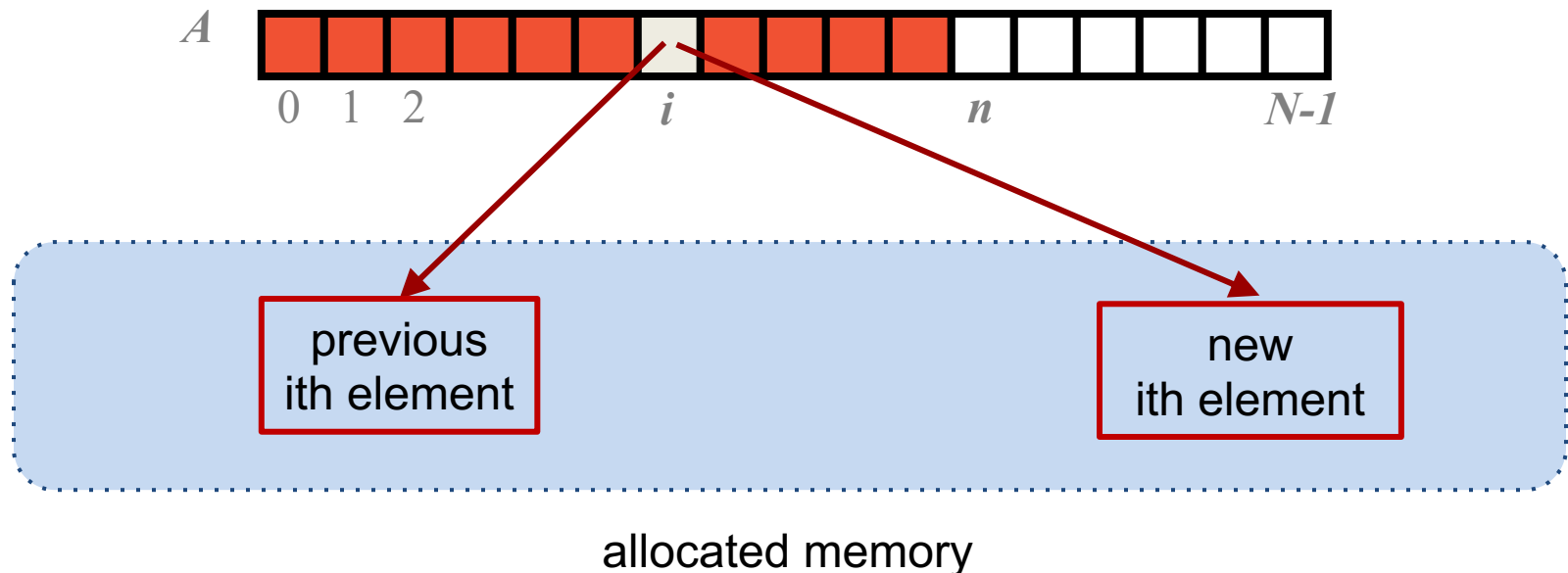
Must check that i is a legitimate index ($0 \le i < n$)

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
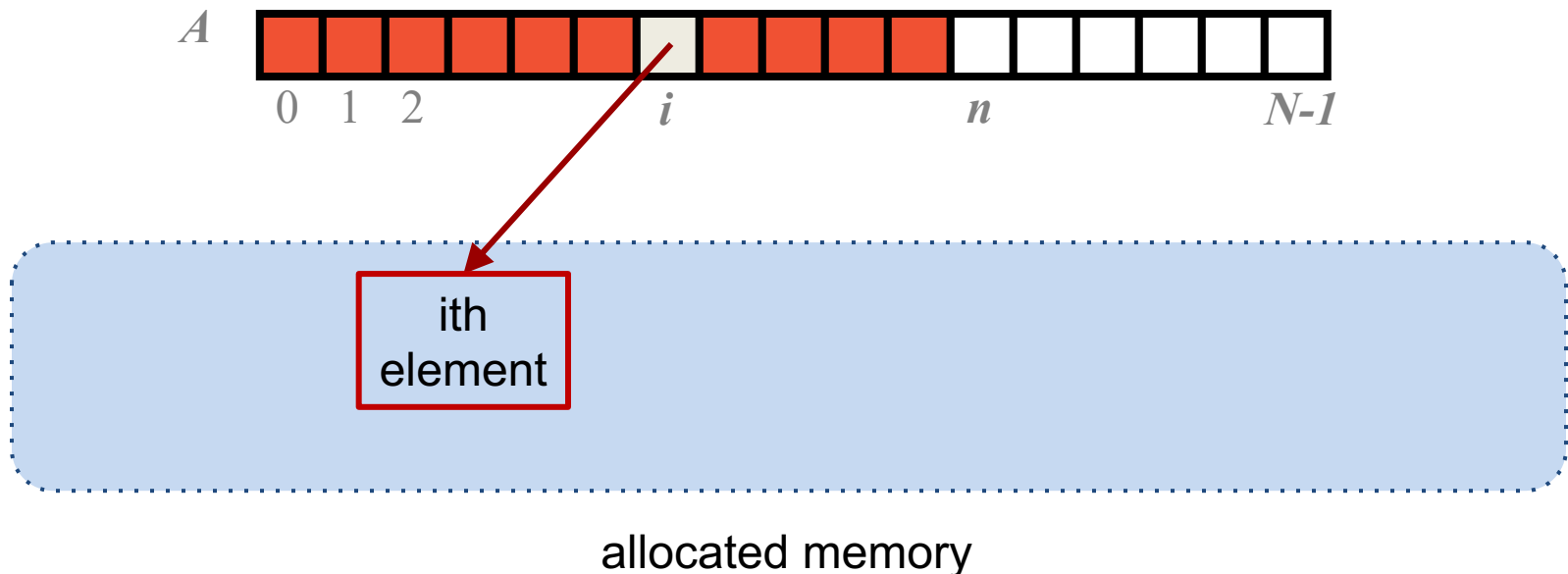
Array-based Lists: set(i,e)

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

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

allocated memory

Pseudo-code for get

```
def get(i):
    # input: index i
    # output: ith element in list
    if i < 0 or i ≥ n then
        return "index out of bound"
    else
        return A[i]
```

Time complexity of this operation is O(1) time, independent of the size of the array (N) or the represented list (n)

Pseudo-code for set

```
def set(i, e):
    # input: index i and value e
    # do: update ith element in list to e
    if i < 0 or i ≥ n then
        return "index out of bound"
    A[i] ← e
```

Time complexity of operation is $O(1)$ time, independent of the size of the array (N) or the represented list (n)

Array-based Lists: add(i,e)

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

Must check that there is space (n < N)

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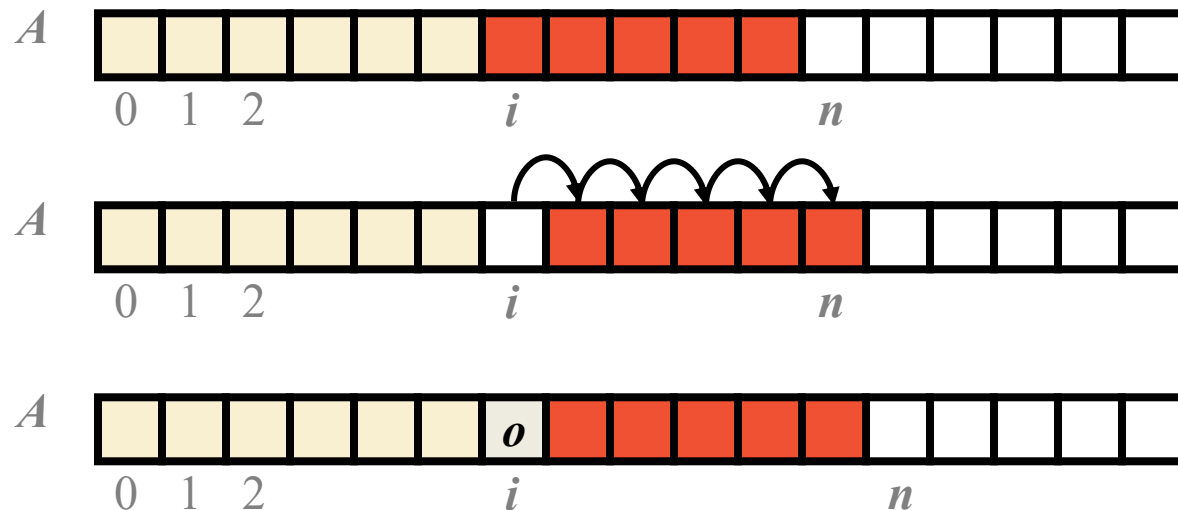
Pseudo-code for insertion

```
def insert(i, e):
    if i < 0 or i > n
        return "index out of bound"
    if n = N then
        return "array is full"

    for j in [n:i:-1] do
        A[j] ← A[j-1]
    A[i] ← e
    n ← n + 1
```

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 elements A[i + 1], …, A[n - 1]

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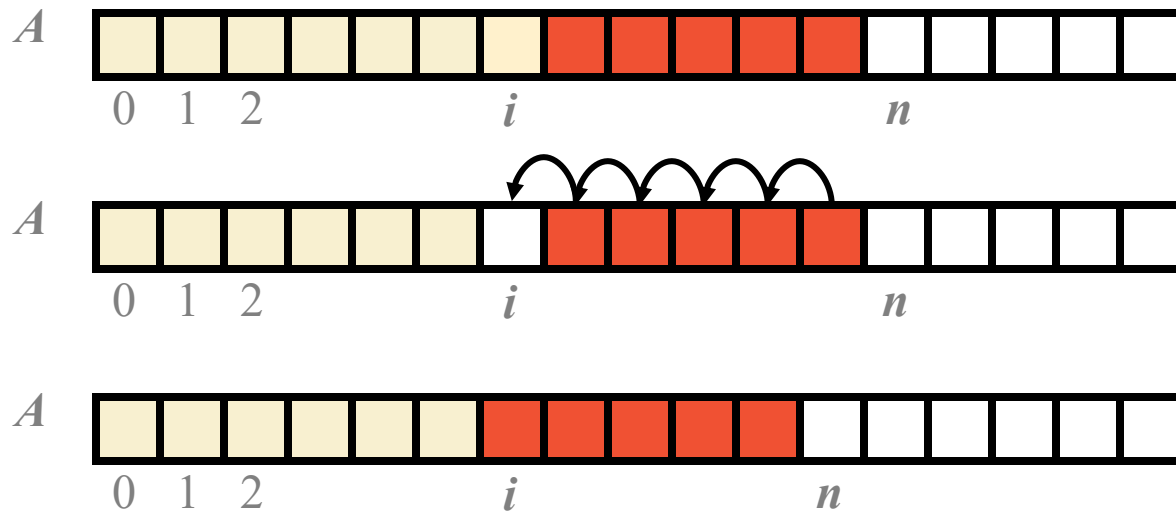
Pseudo-code for removal

```
def remove(i):
    if i < 0 or i ≥ n
        return "index out of bound"
    e ← A[i]
    if i < n-1
        for j in [i:n-1] do
            A[j] ← A[j+1]
    n ← n - 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 a dynamic array because the size of the array changes dynamically throughout the execution

Python's list type are implemented this way

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, but we need to be careful 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
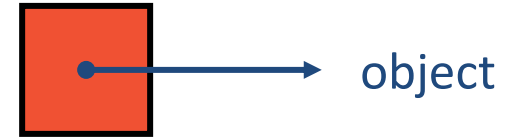
object

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.

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
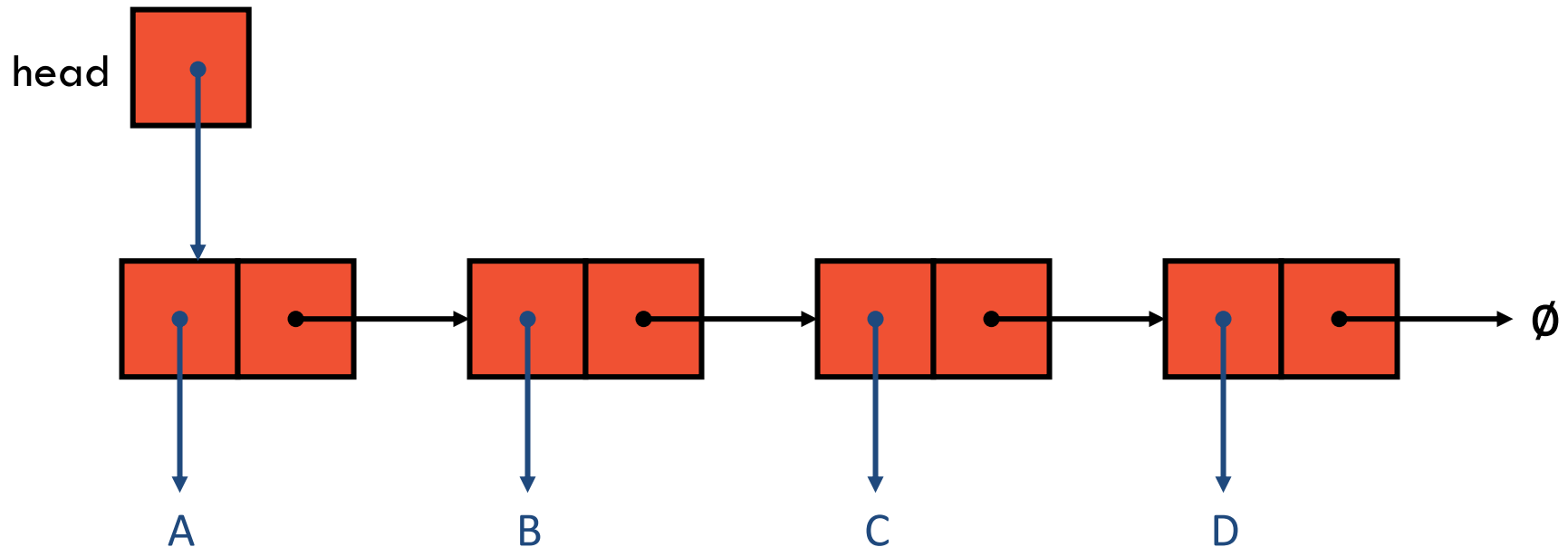
Node implements Position

Each Node in a singly linked List stores
- its element, and
- a link to the next node.
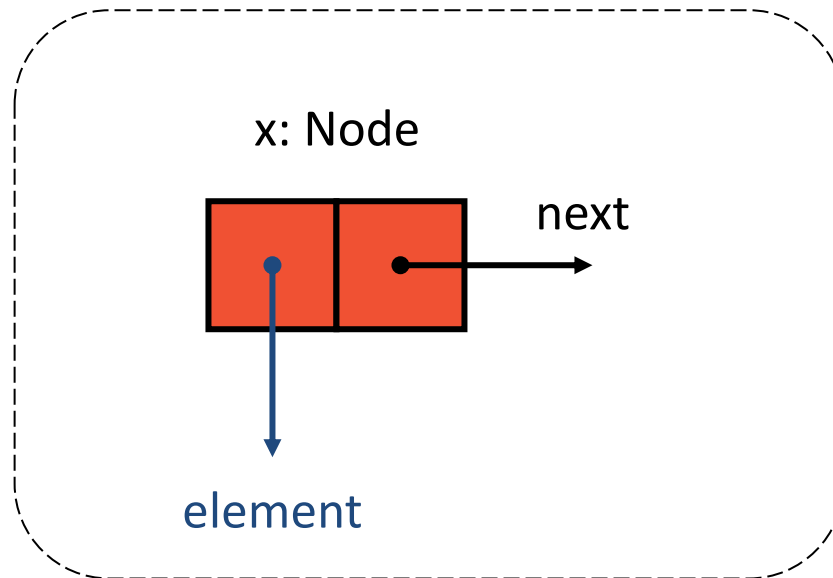
Advice on working with linked structures

- 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
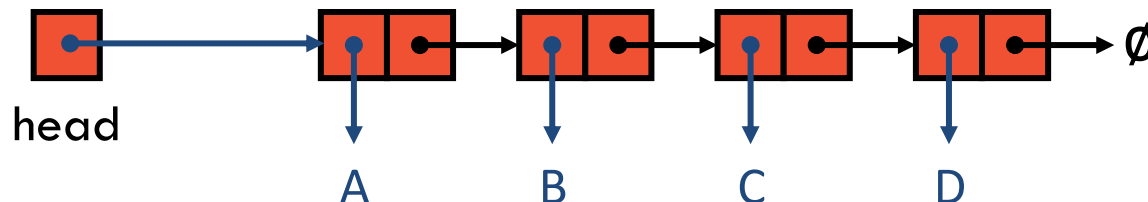
first()

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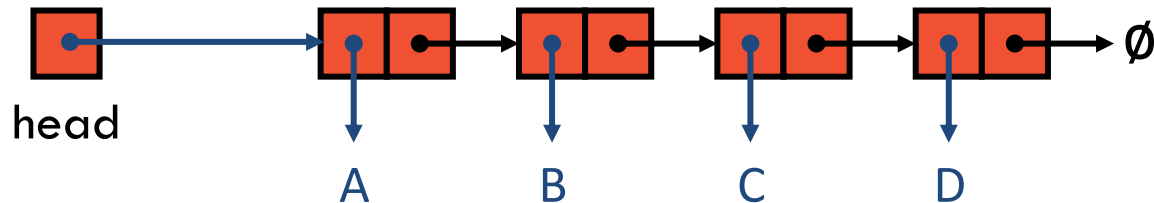
return?

first()

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

return head

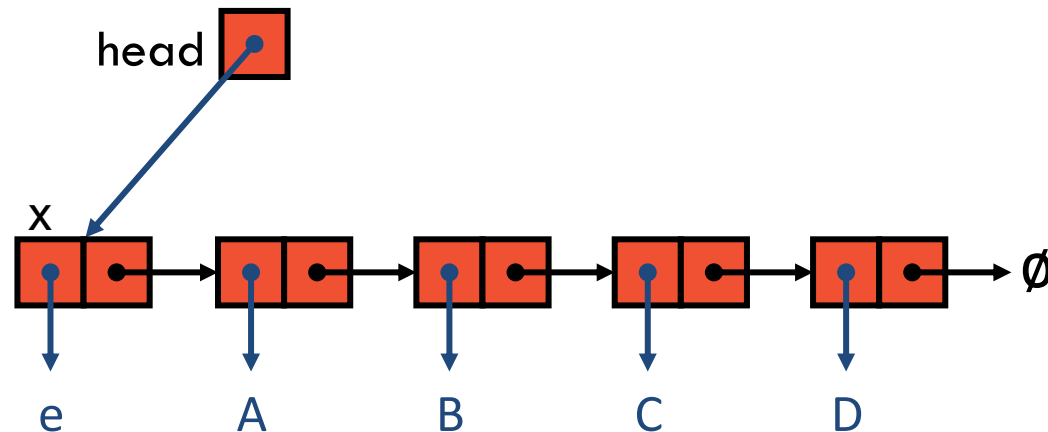
Time complexity?

insertFirst(e)

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

insertFirst(e)

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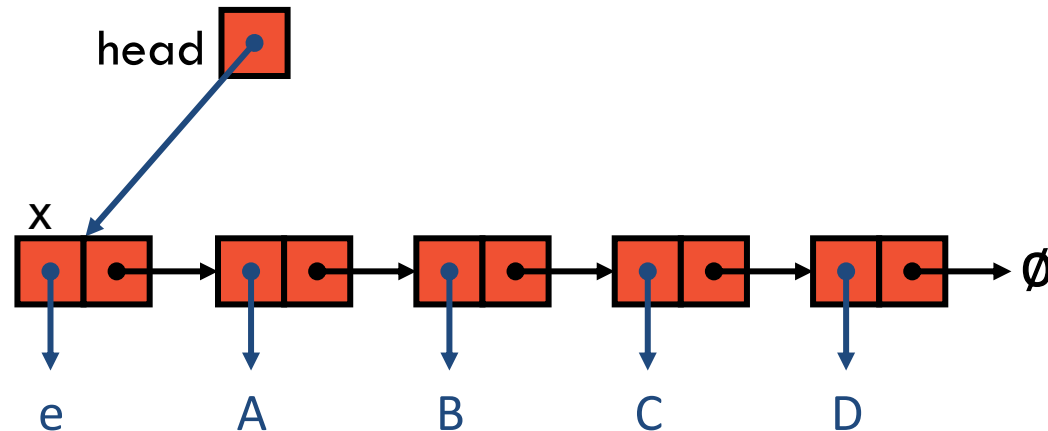4. Update list's head to point to x

insertFirst(e)

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

insertFirst(e)

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

insertFirst(e)

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

What is the time complexity?

insertFirst(e)

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

What is the time complexity? O(1)

removeFirst()

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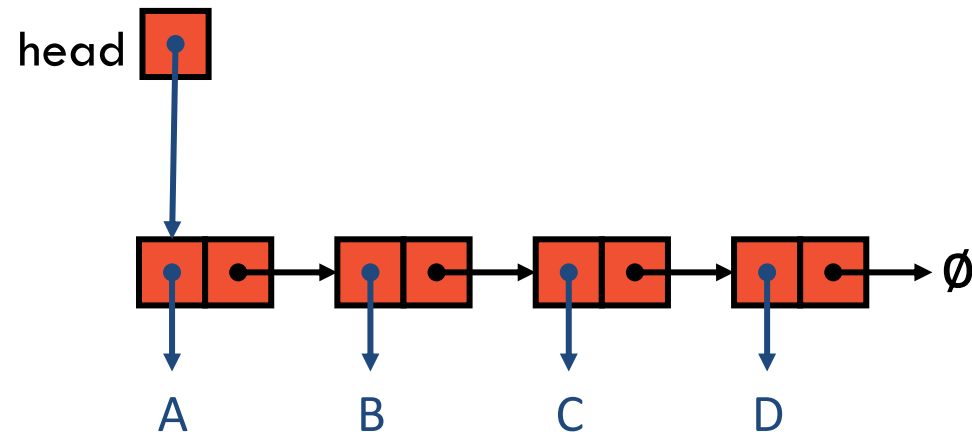
removeFirst()

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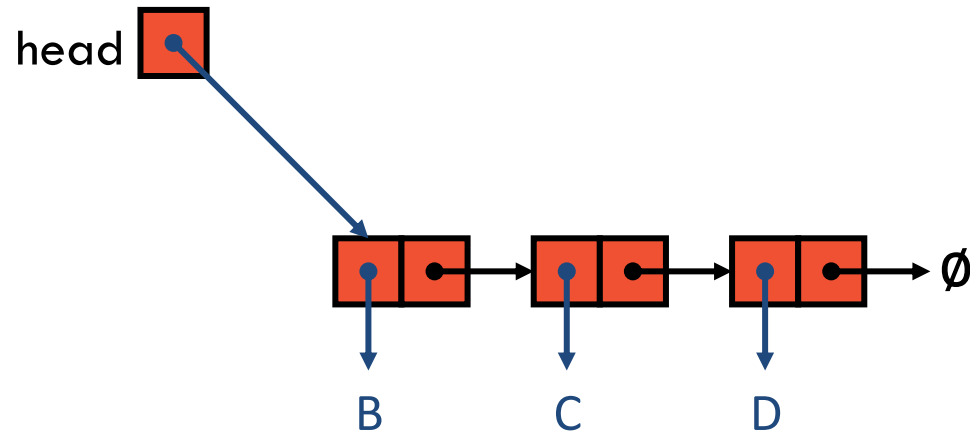2. Delete the former first node
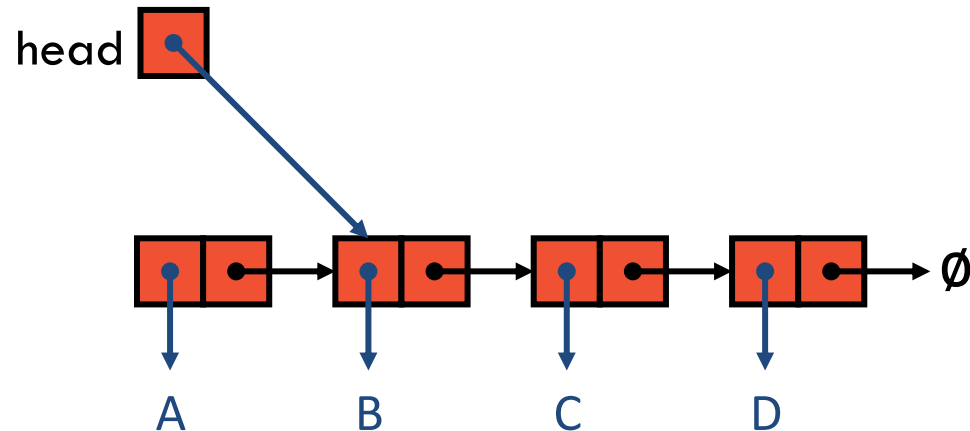
removeFirst()

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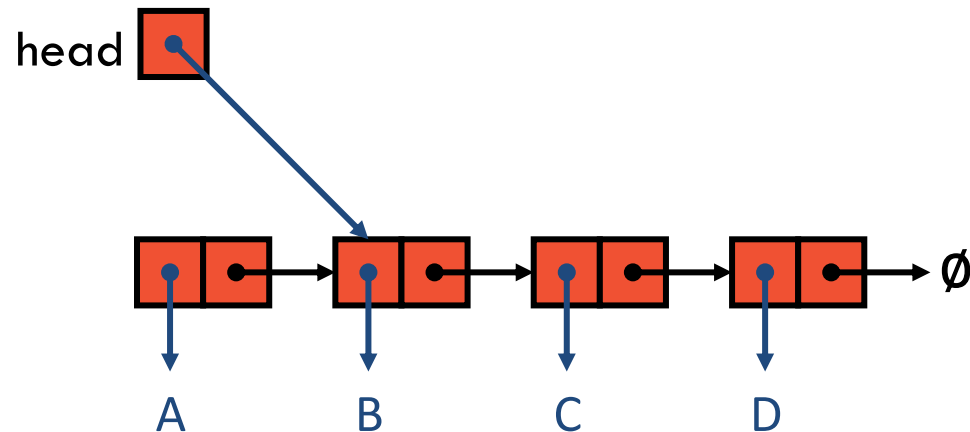
removeFirst()

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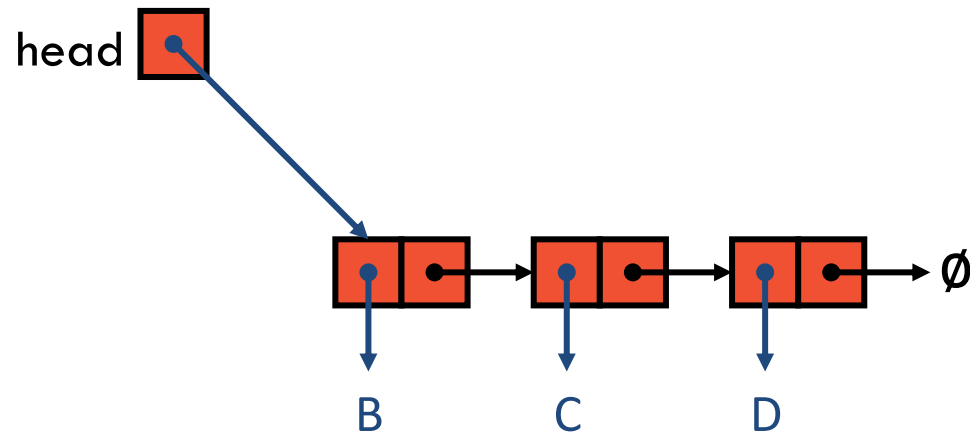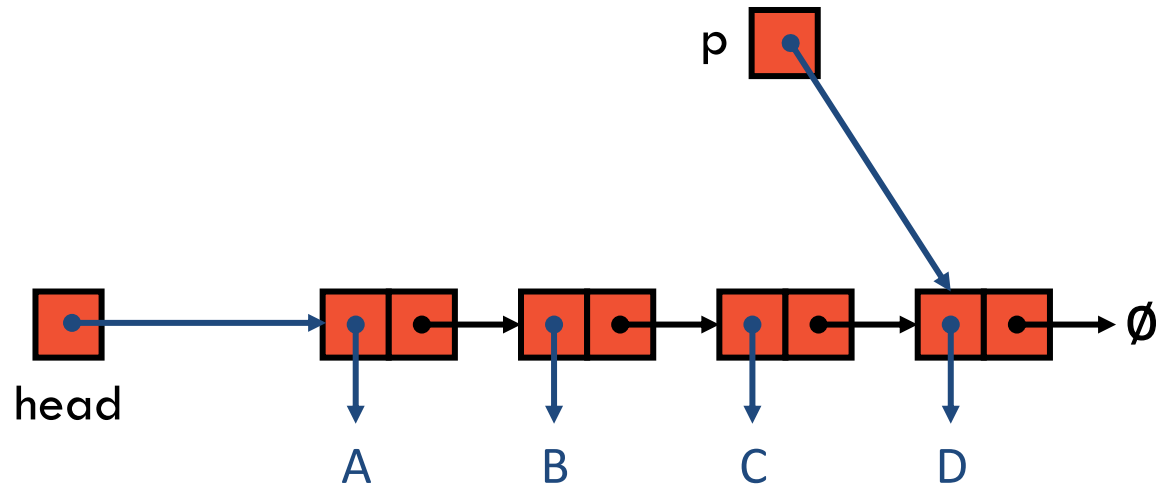
Time complexity?

removeFirst()

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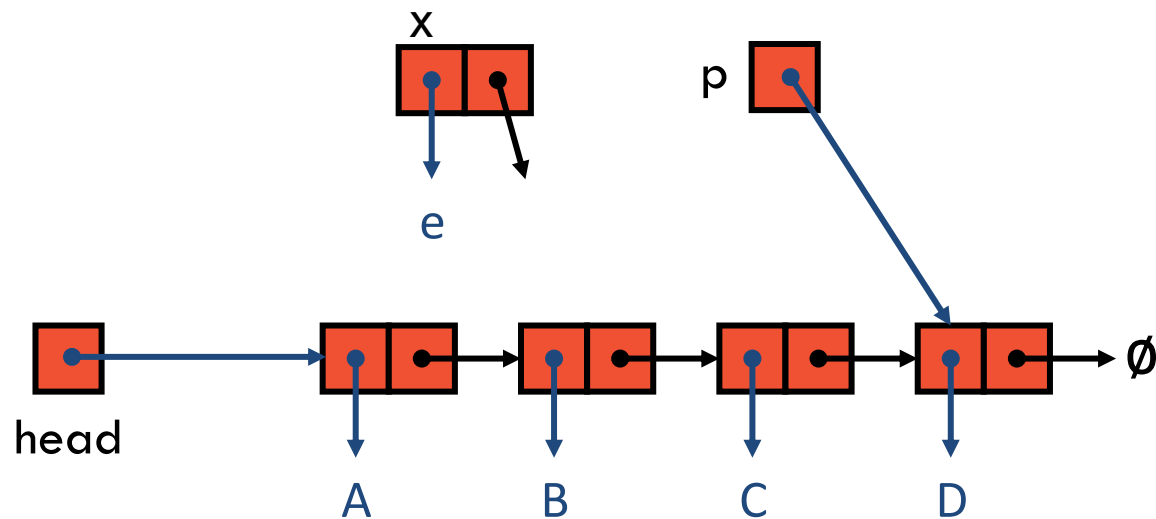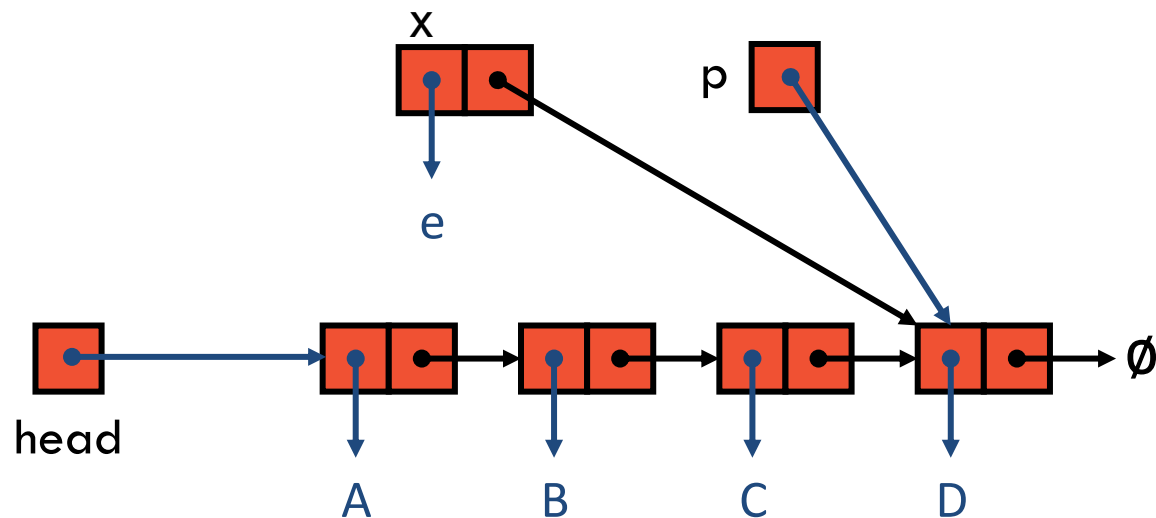2. Delete the former first node

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
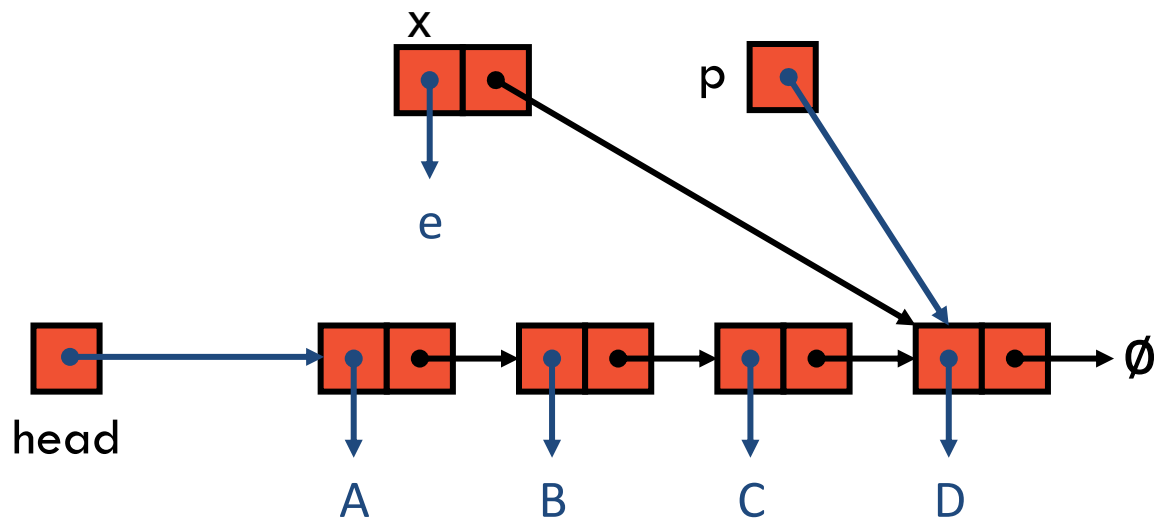
insertBefore(p,e)

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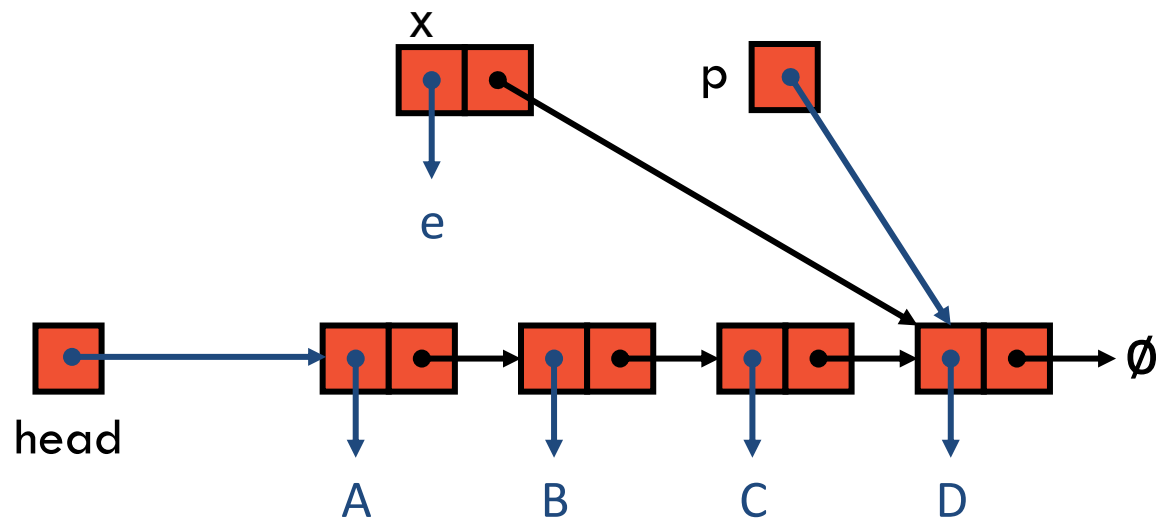

What's the next step?

insertBefore(p,e)

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

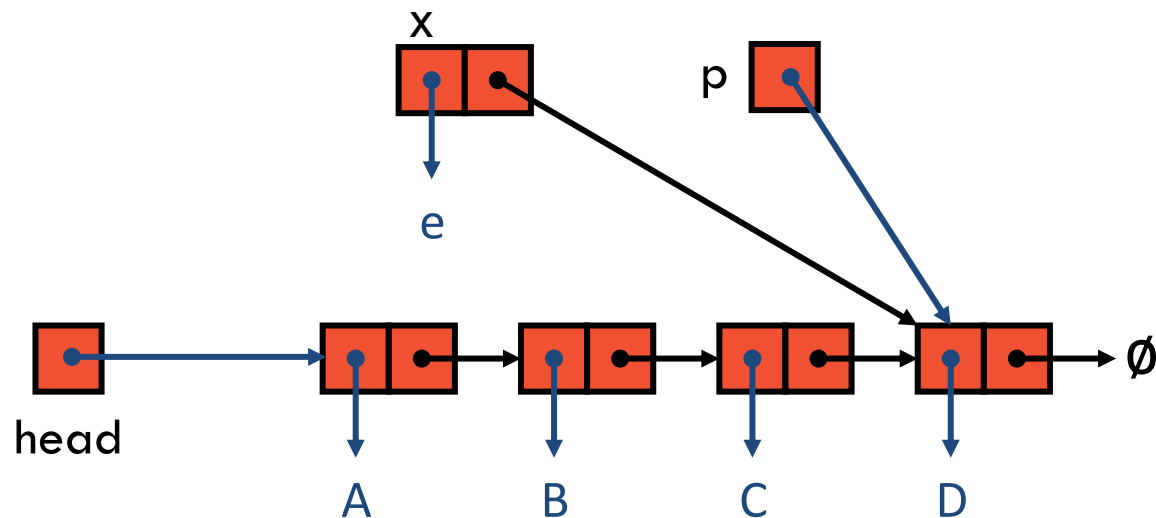What's the next step? Find the predecessor of x. How?

insertBefore(p,e)

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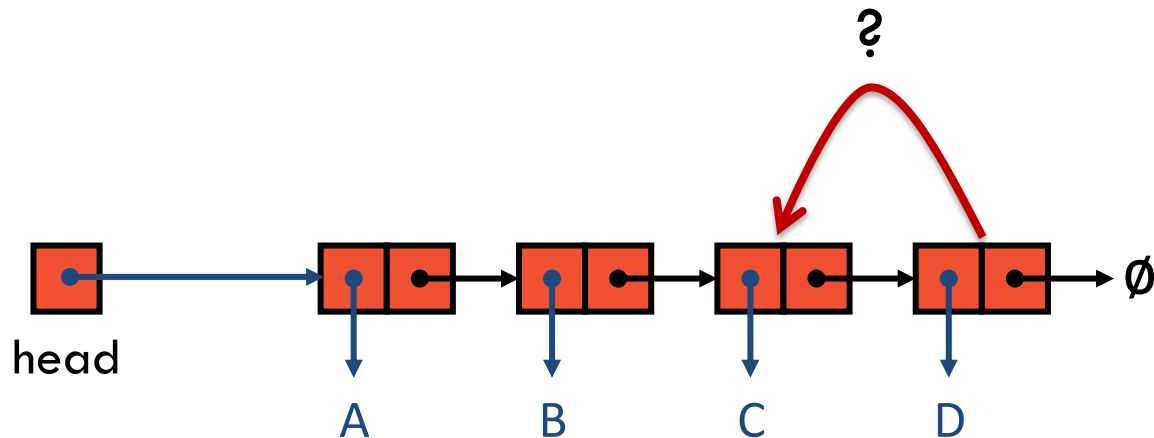
insertBefore(p,e)

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.
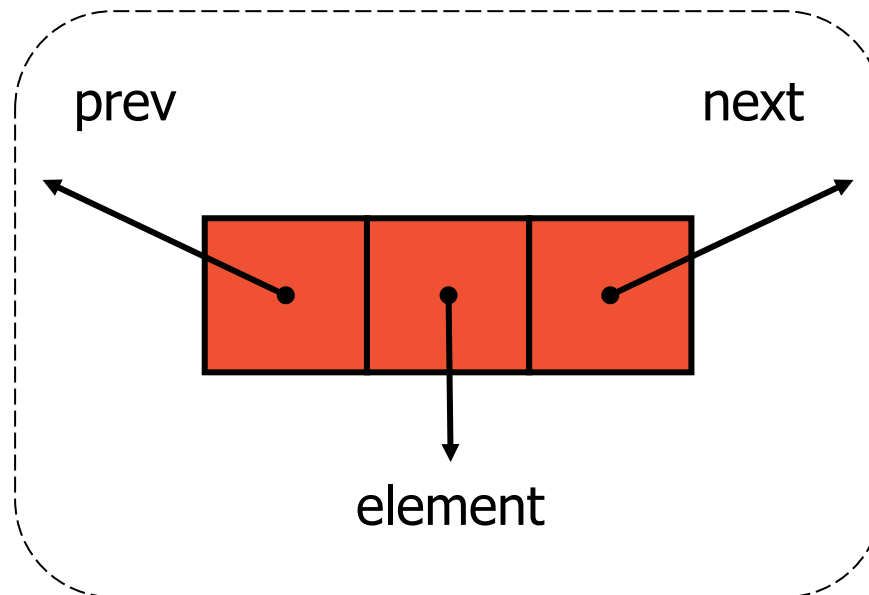
Another attempt

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.

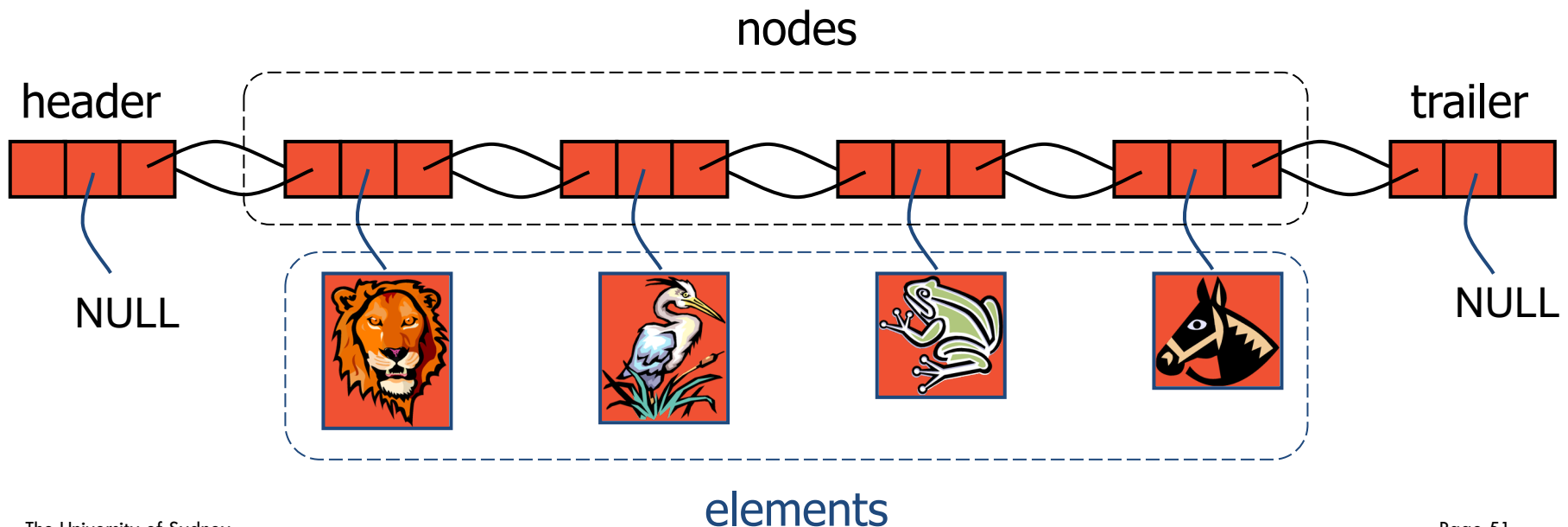Each Node in a Doubly Linked List stores

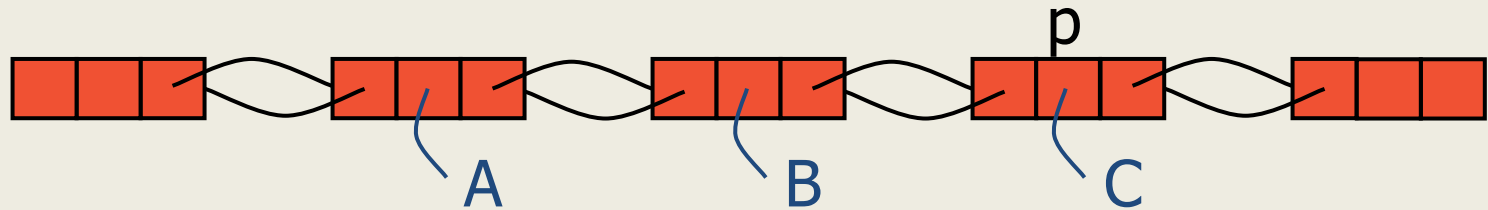- its element, and
- a link to the previous and next nodes.

Doubly Linked Lists

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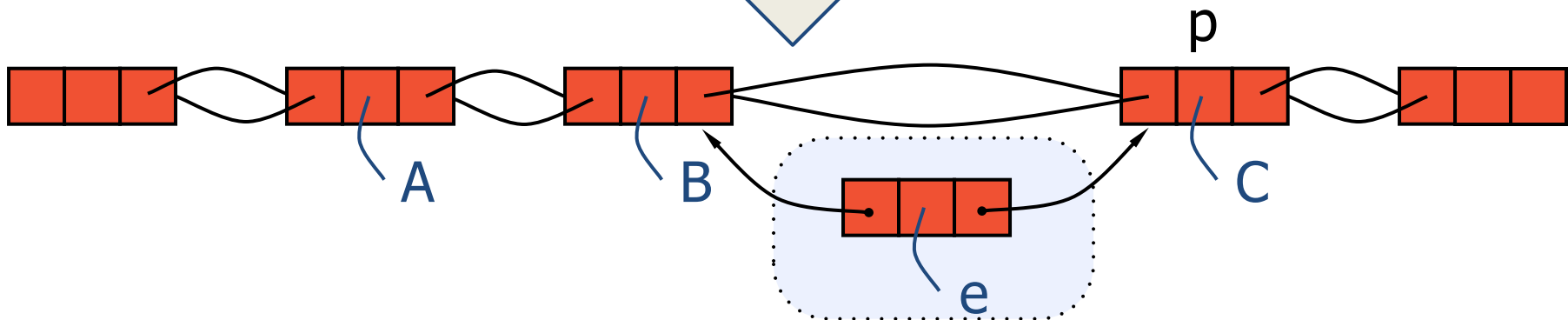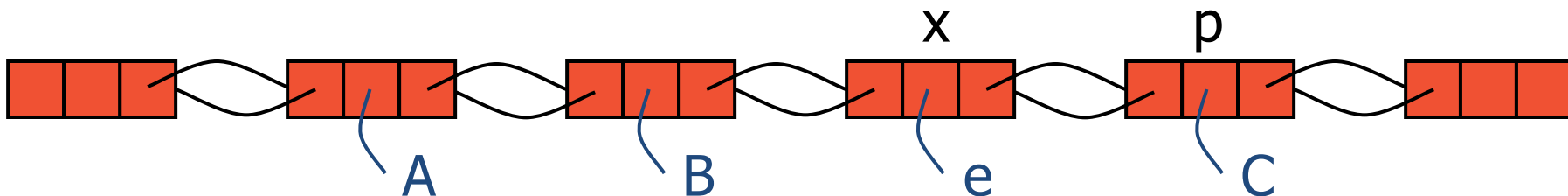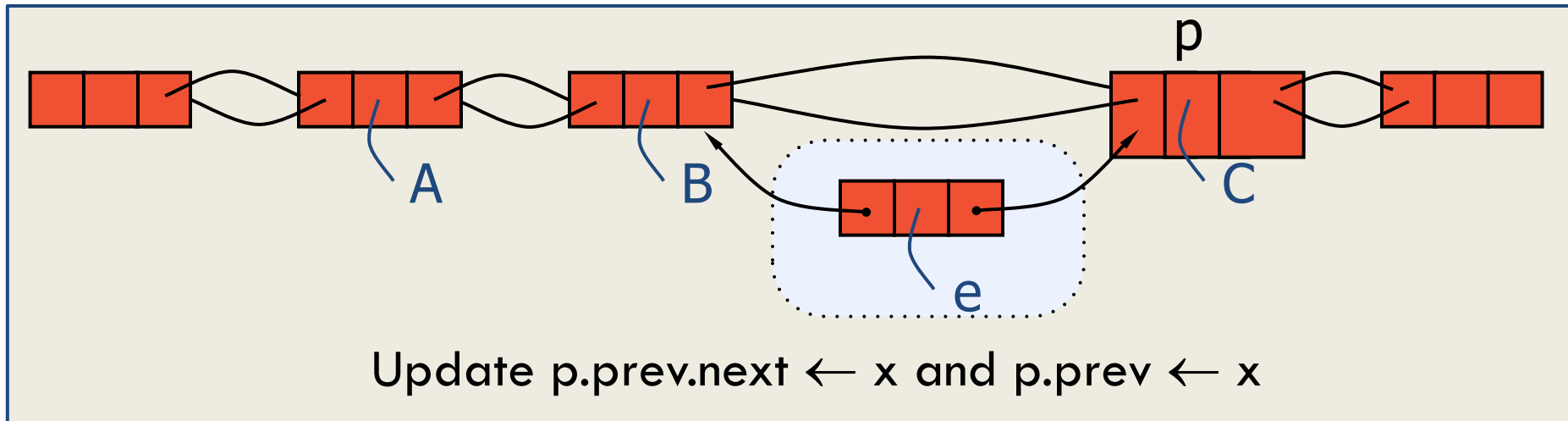- List captured by references to its Sentinel Nodes

insertBefore(p,e) – step 1

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

insertBefore(p,e) – step 2

Update p.prev.next ← x and p.prev ← x

insertBefore(p,e)

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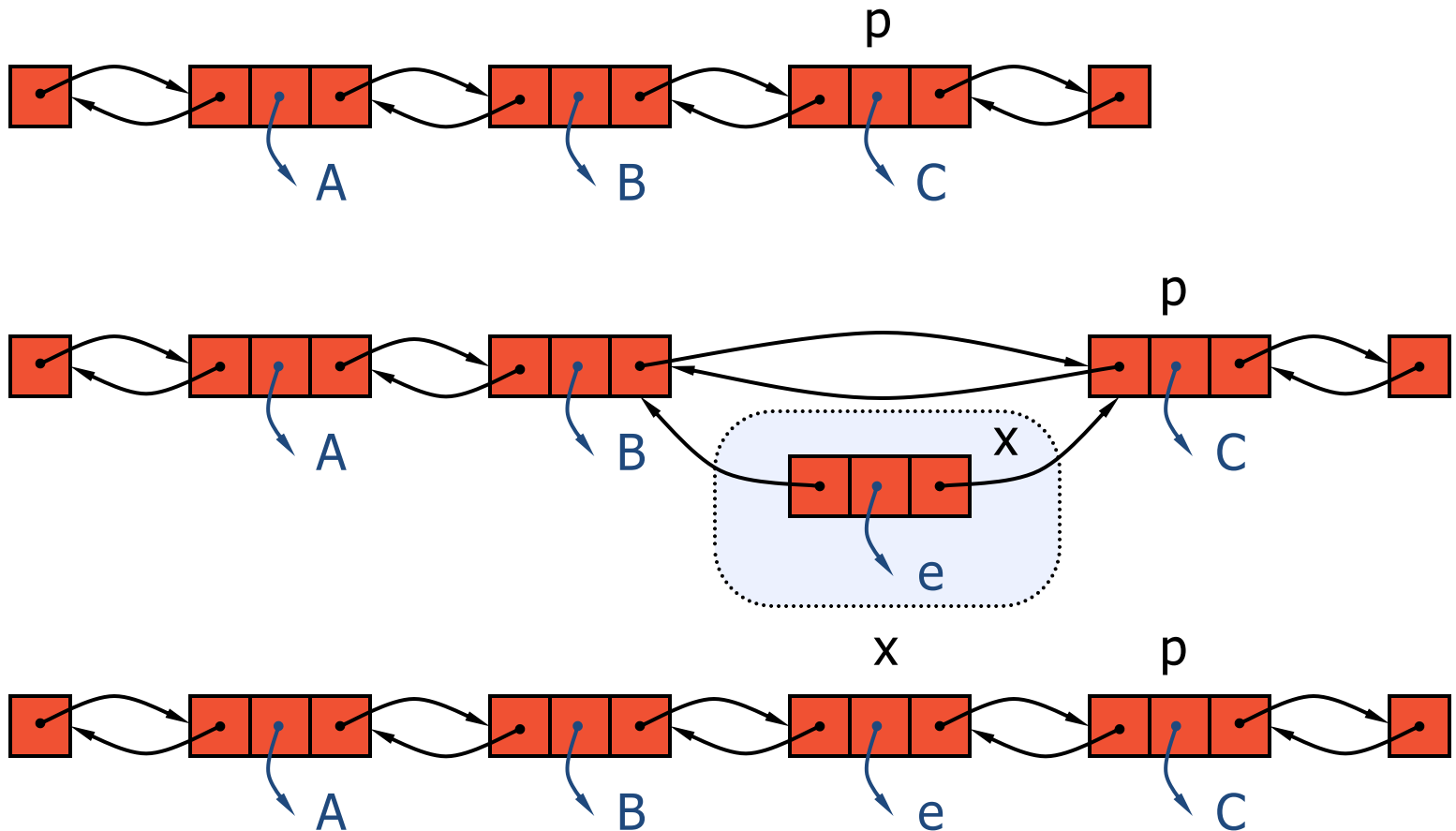
Pseudo-code

```
def insert_before(pos, elem):
    // insert elem before pos
    // assuming it is a legal pos

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

    return new_node
```

remove(p)

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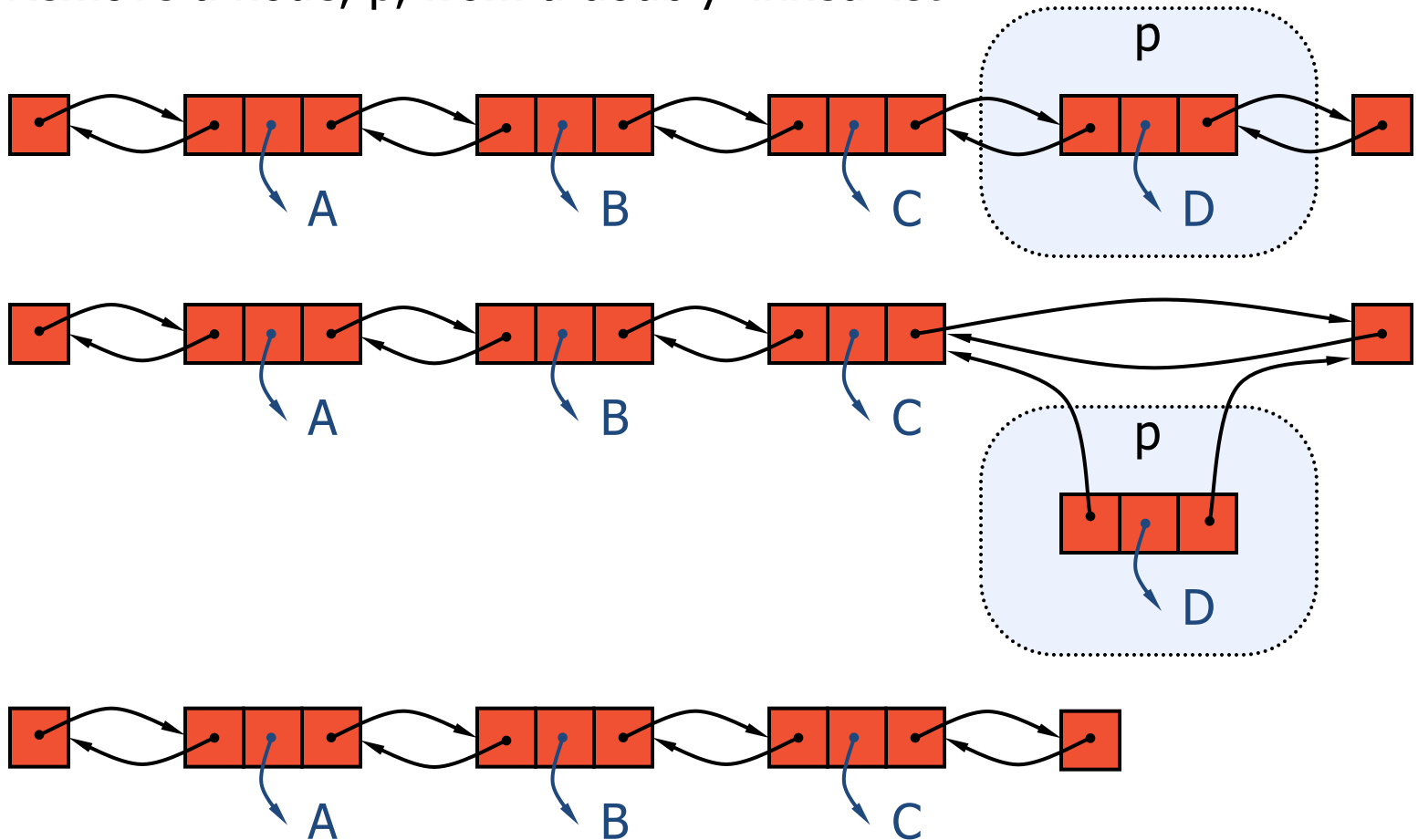
Pseudo-code

```
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)

Time complexity is O(1) for all operations

Method	Time
first()	O(1)
last()	O(1)
before(p)	O(1)
after(p)	O(1)
insert_before(p, e)	O(1)
insert_after(p, e)	O(1)
remove(p)	O(1)
size()	O(1)
is_empty()	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)

Iterators in Python

iter(obj) returns an iterator of the object collection

To make a class 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

```
for x in collection:
    [process x]
```

Is equivalent to:

```
it ← x.__iter__()
try:
    while True:
        x ← it.next()
        [process x]
except StopIteration:
    pass
```