COMP2823

Lecture 2: Lists

[GT 2.1-2.2]

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.

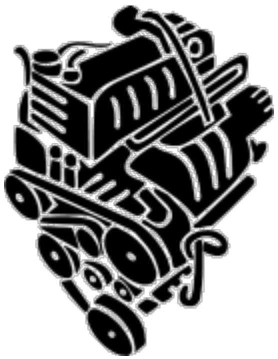
Abstract Data Types (ADT)

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

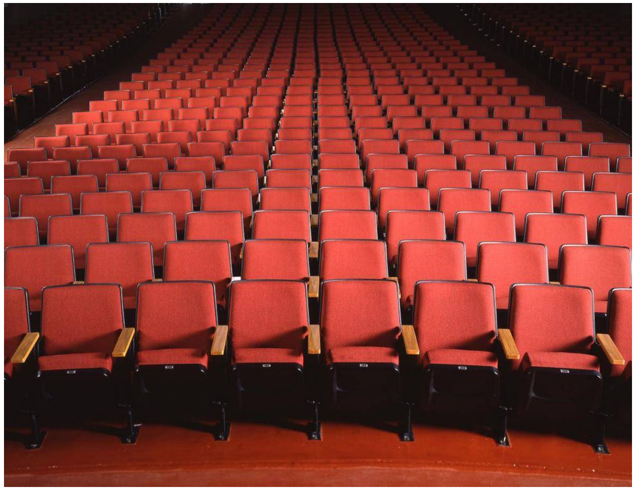
- Code is easier to understand if different functionalities are separated into different places.

- 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

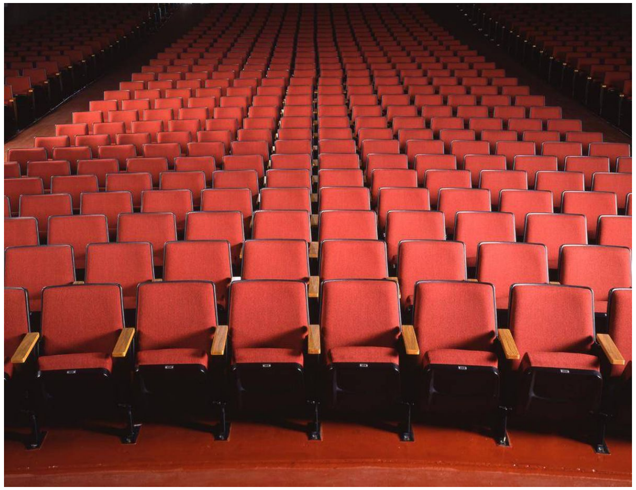


- We have a theatre with 500 named seats, e.g., “N31”

- 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
add(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

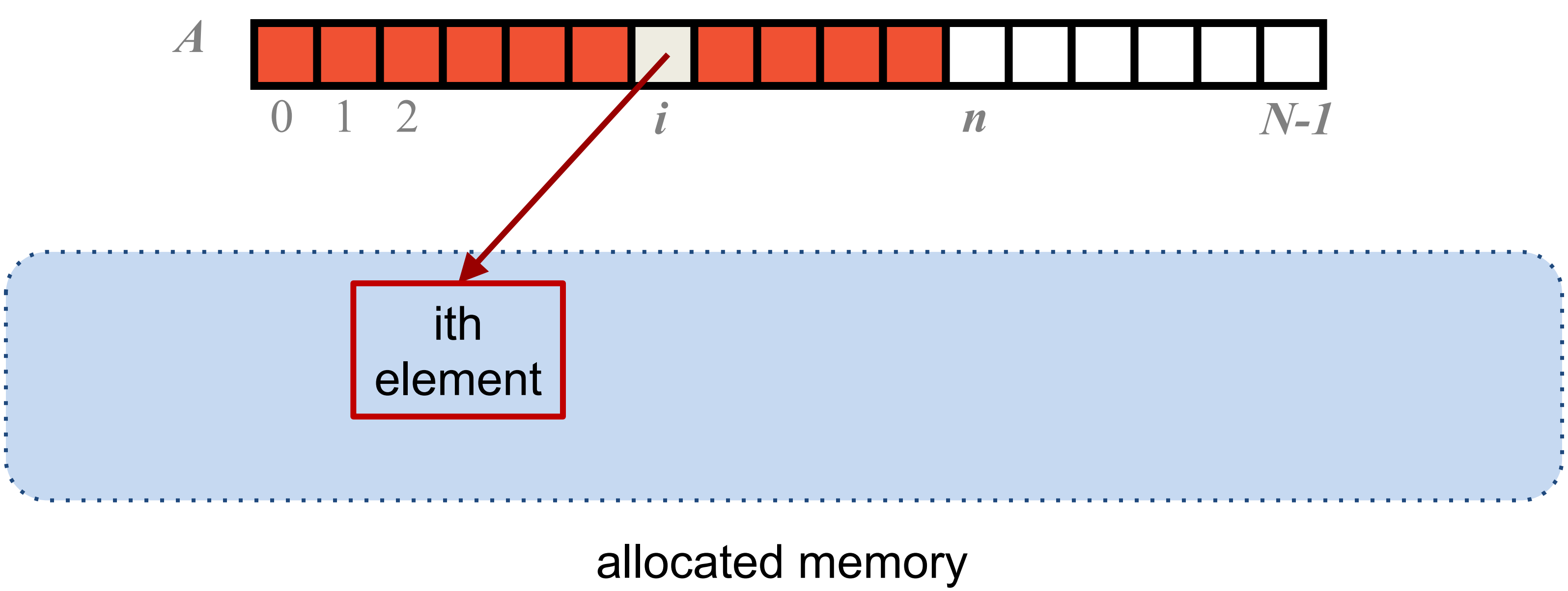
A sequence of List operations:

Method	Returned value	List content
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, G, C, F]

Array-based Lists

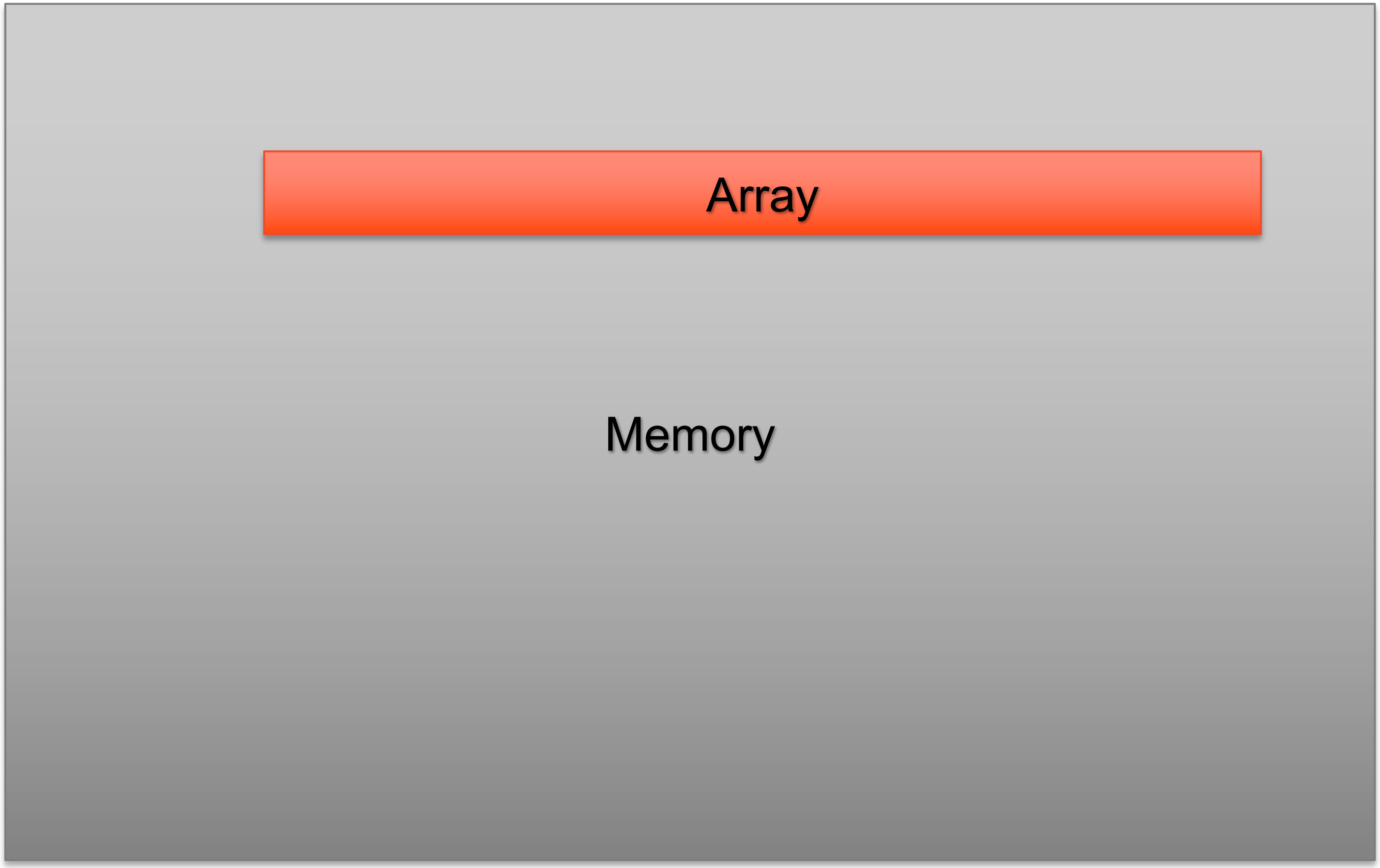
An option for implementing the list ADT is to use an array A, where A[i] stores (a reference to) the element with index i.

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



Array-based Lists

How is an array stored?

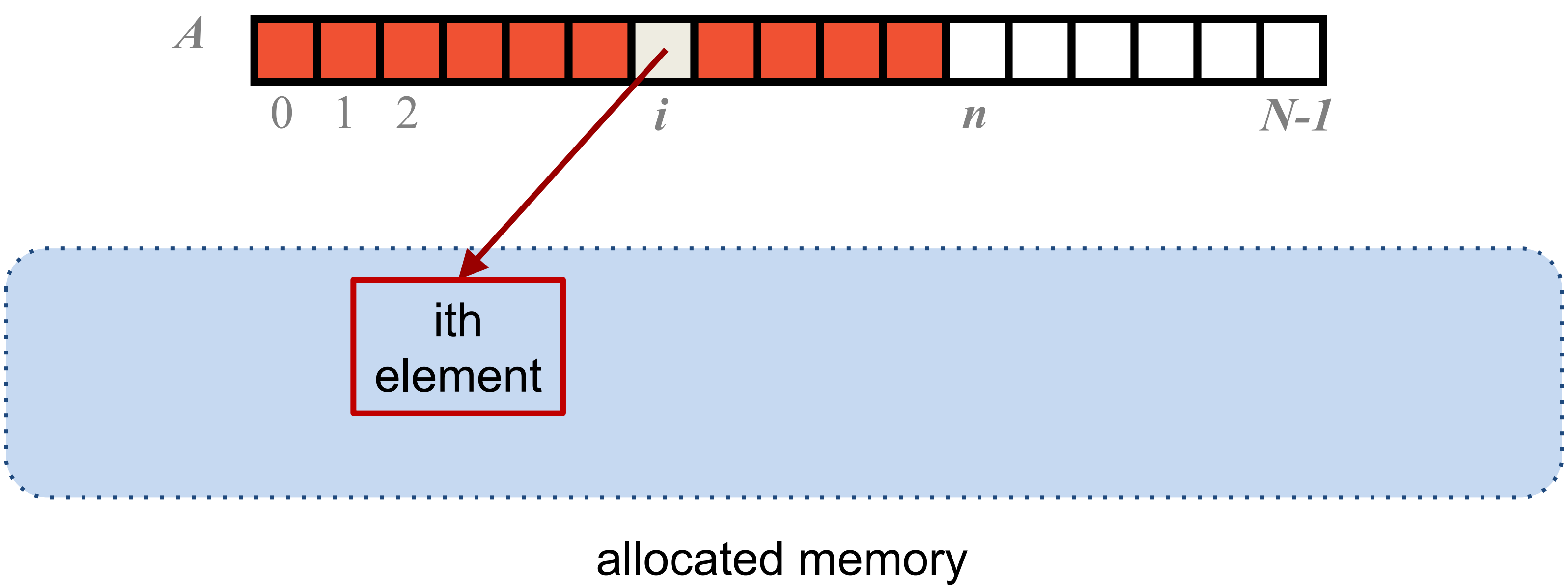


Array-based Lists: get(i)

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

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

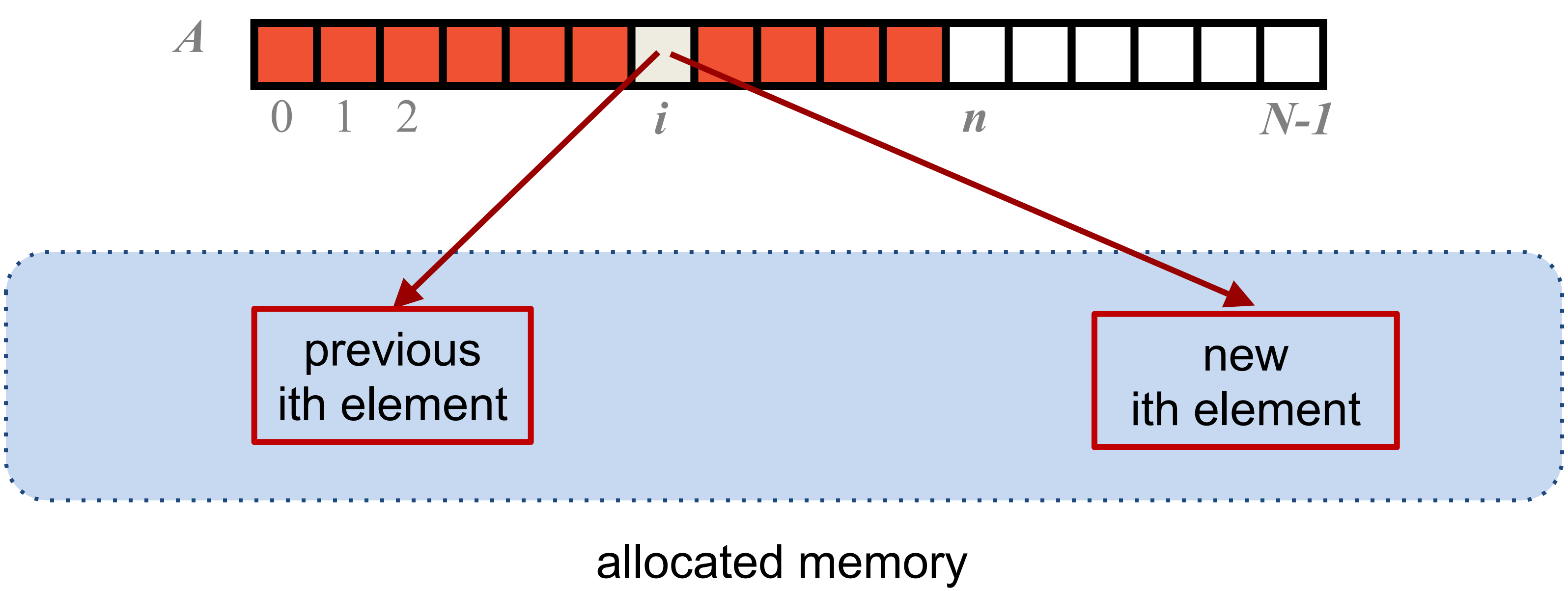
Both operations can be carried out in in constant time (a.k.a. O(1) time), independent of the size of the array



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 ($0 \leq i < n$)



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"
   result ← A[i]
   A[i] ← e
   return result
```

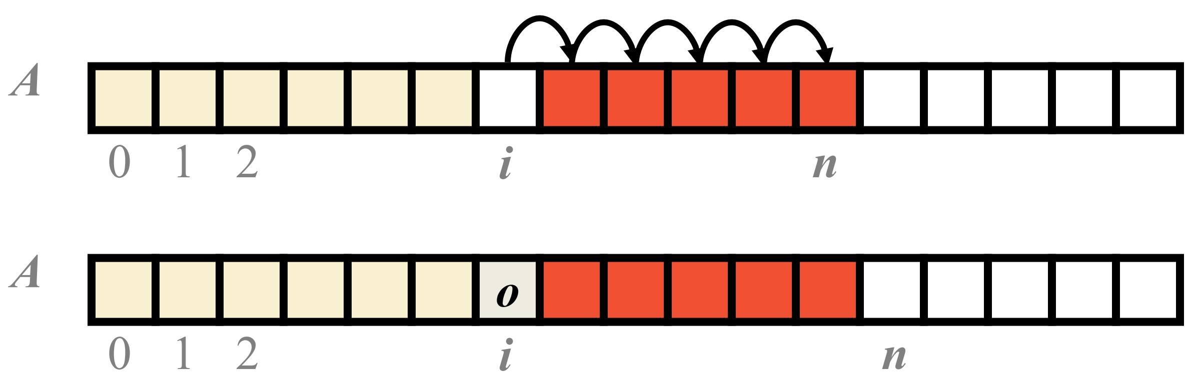
Time complexity of operation is O(1) time, independent of the size of the array (N) or the represented list (n)

Array-based Lists: add(i,e)

In an operation add(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

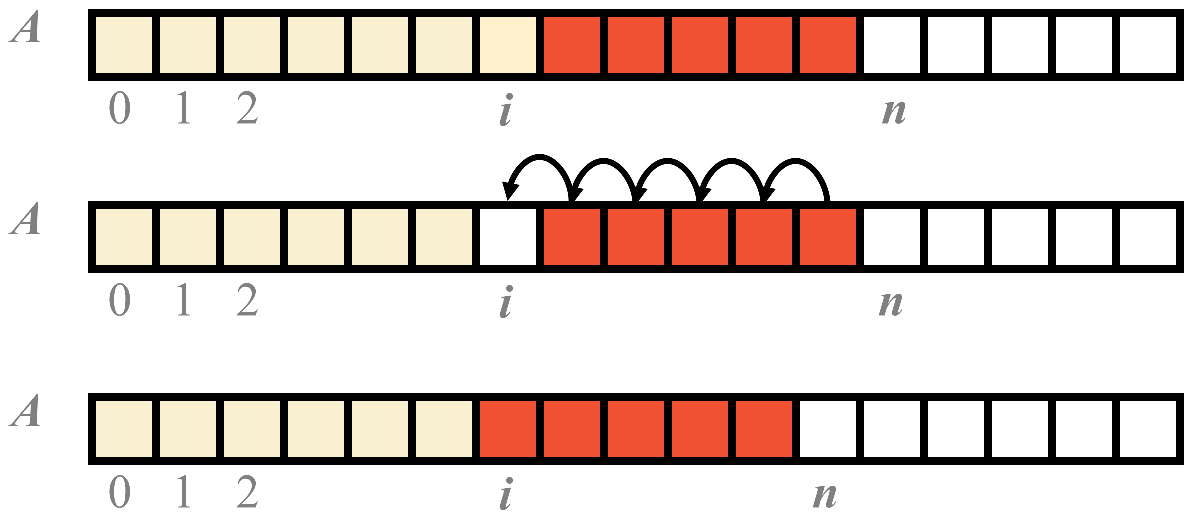
```
def add(i, e):
    if i < 0 or i > n
        return “index out of bound”
    if n = N then
        return “array is full”

    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 ($0 \leq i < n$)



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:

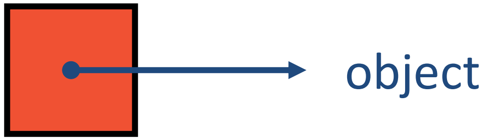
– space used is O(n)

Time complexity:

– both get and set take O(1) time

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

Positional Lists



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:

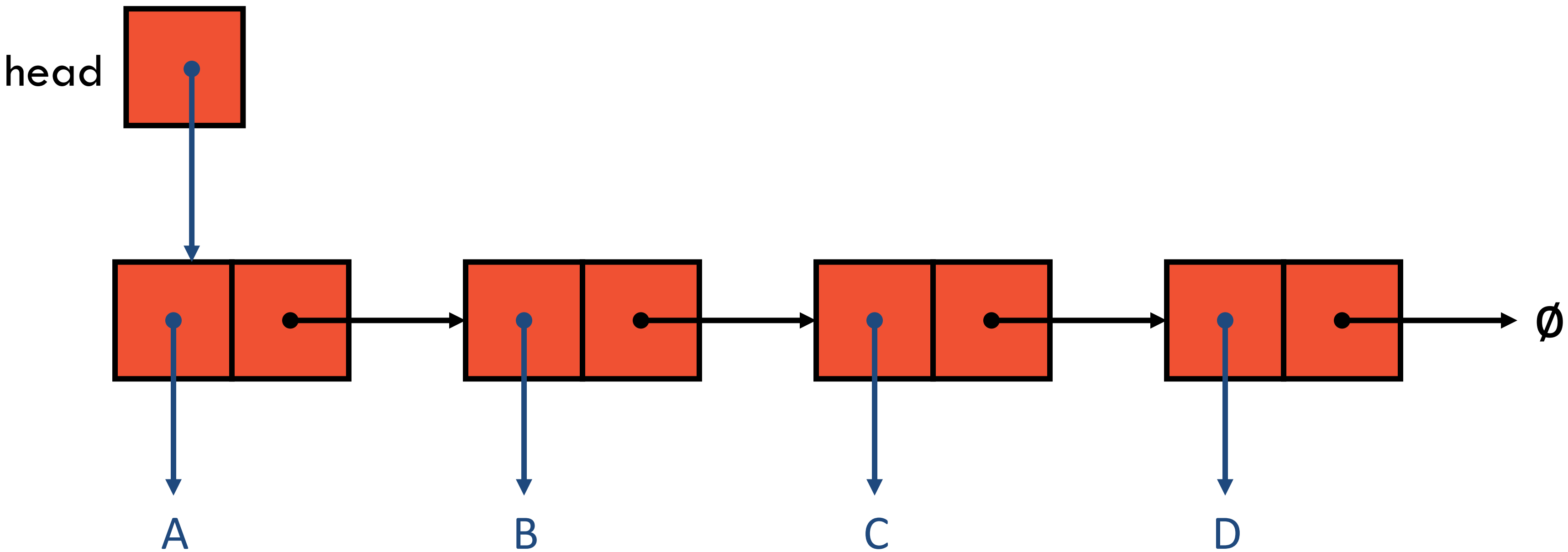
element() : return the element stored at the position instance

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

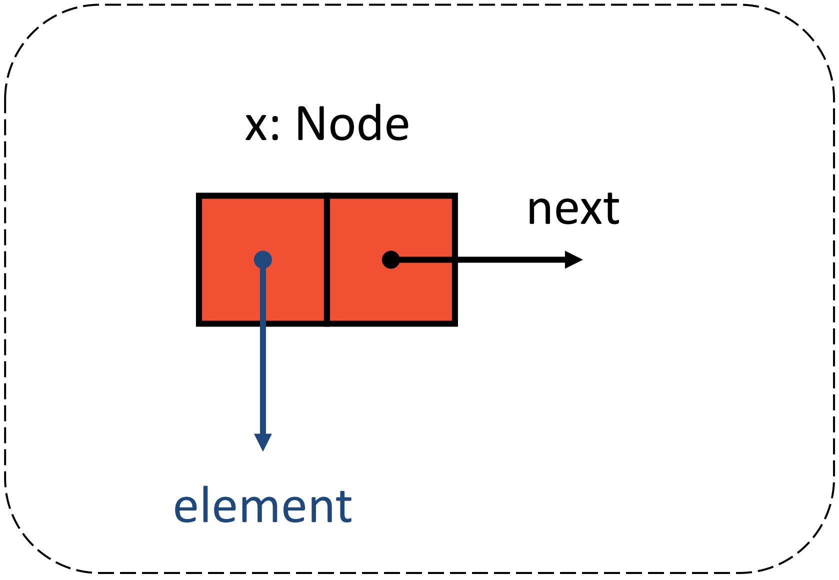
- A concrete data structure
- A sequence of Nodes, each with a reference to the next node
- List captured by reference (head) to the first Node



Node implements Position

Each Node in a singly linked List stores

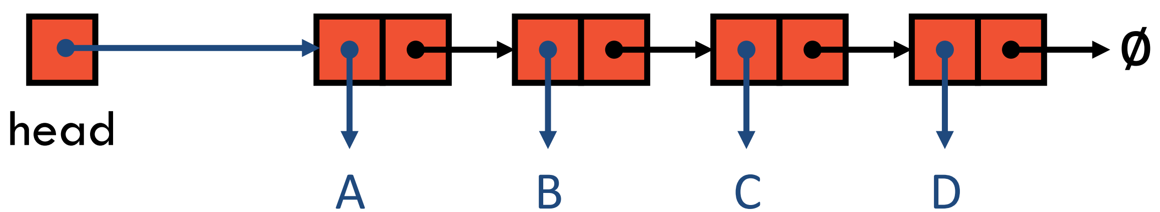
- its element, and
- a link to the next node.



Advice on working with linked structures

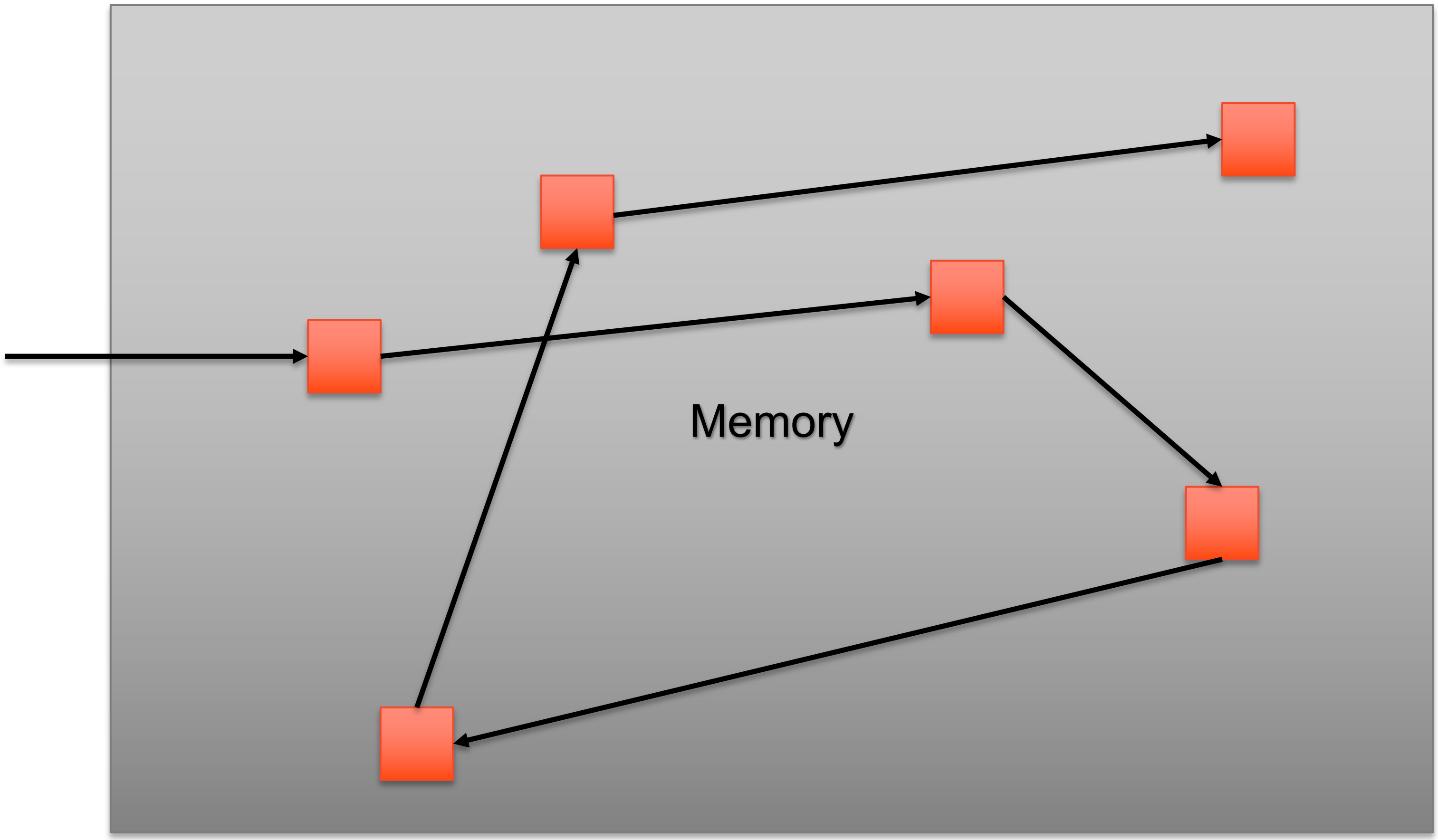
- Draw the diagram showing the state.
- Show a location where you place carefully each of the instance variables (including references to nodes).
- Be careful to step through dotted accesses e.g. p.next.next
- Be careful about assignments to fields e.g.

  p.next ← q or p.next.next ← r



Linked Lists

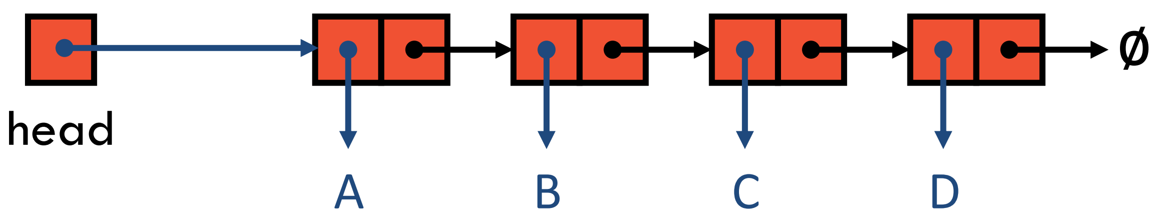
How are linked lists stored?



first()

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

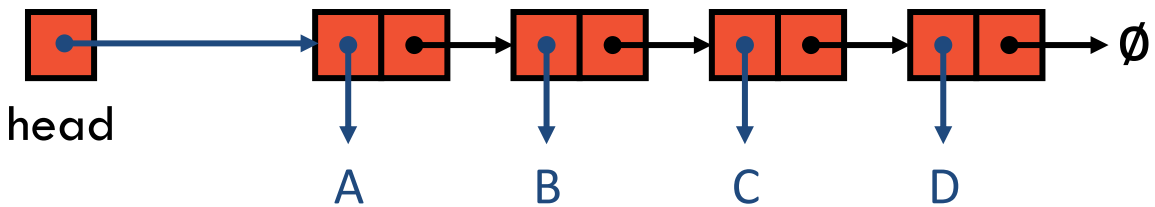
return?



first()

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

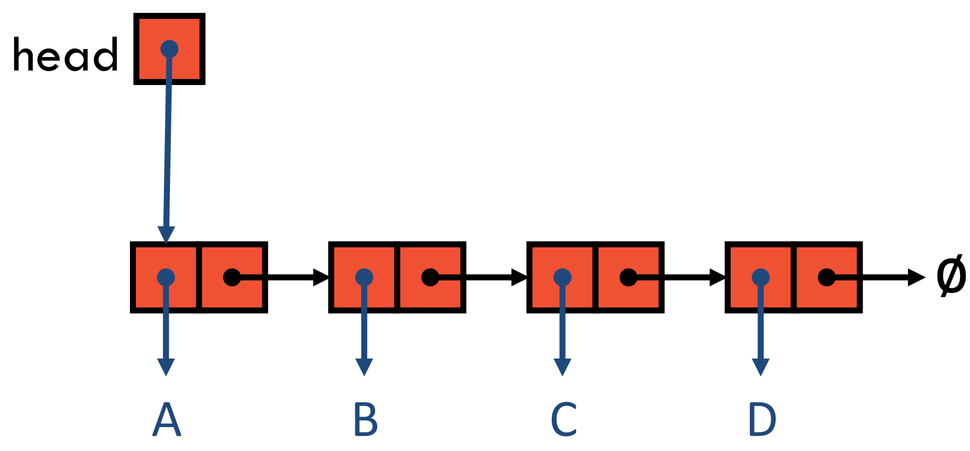
return head



Time complexity?

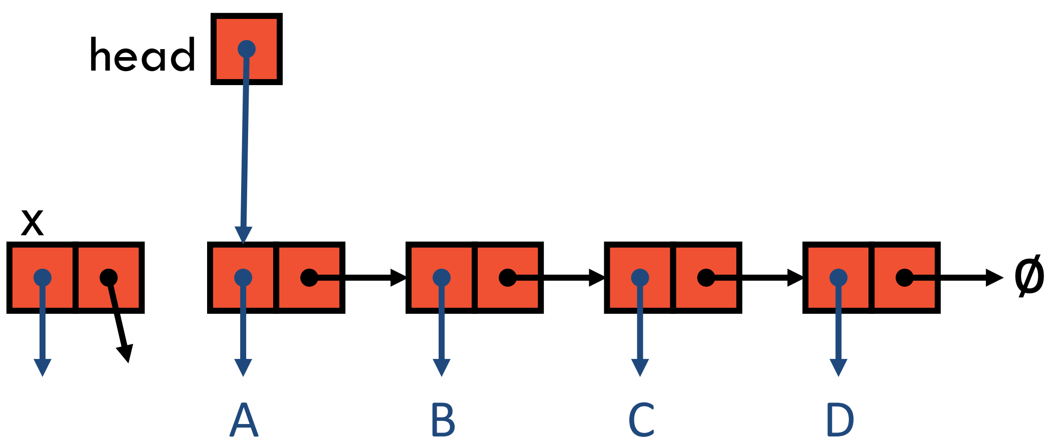
insertFirst(e)

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



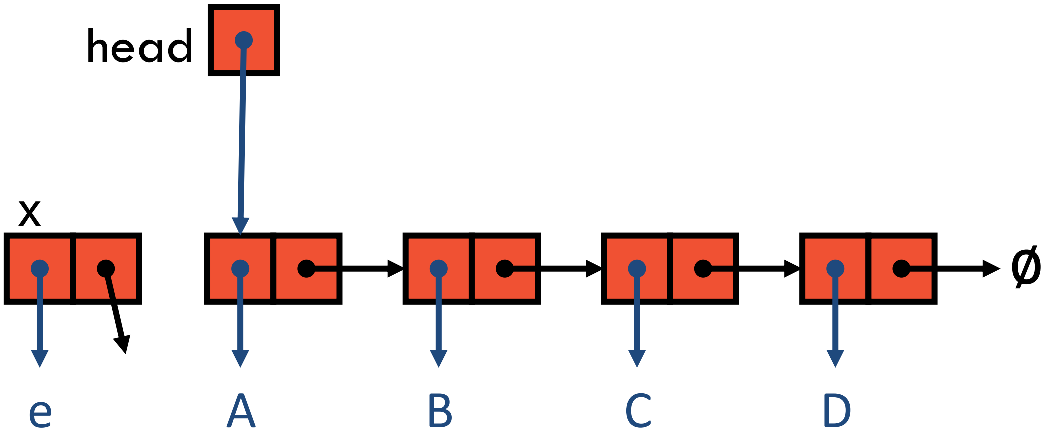
insertFirst(e)

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



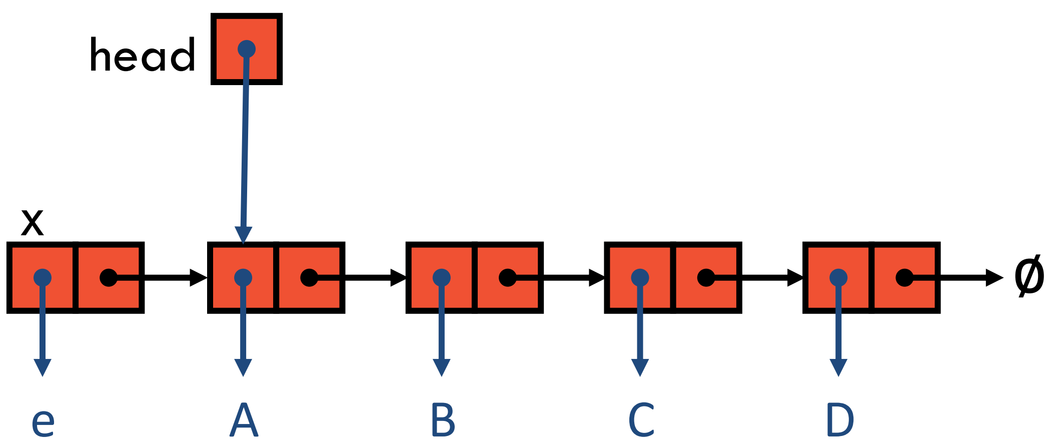
insertFirst(e)

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)

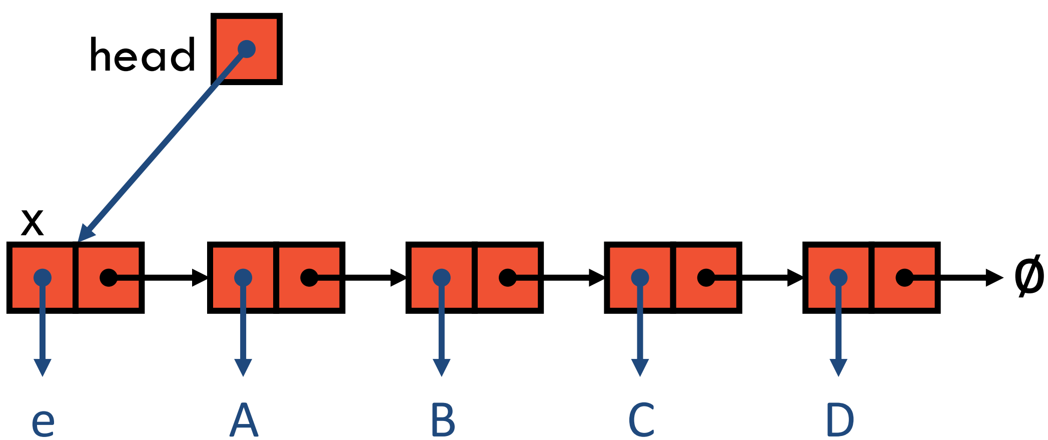
1. Instantiate a new node **x**
2. Set e as element of x
3. Set x.next to point to (old) head
4. Update list's head to point to x



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

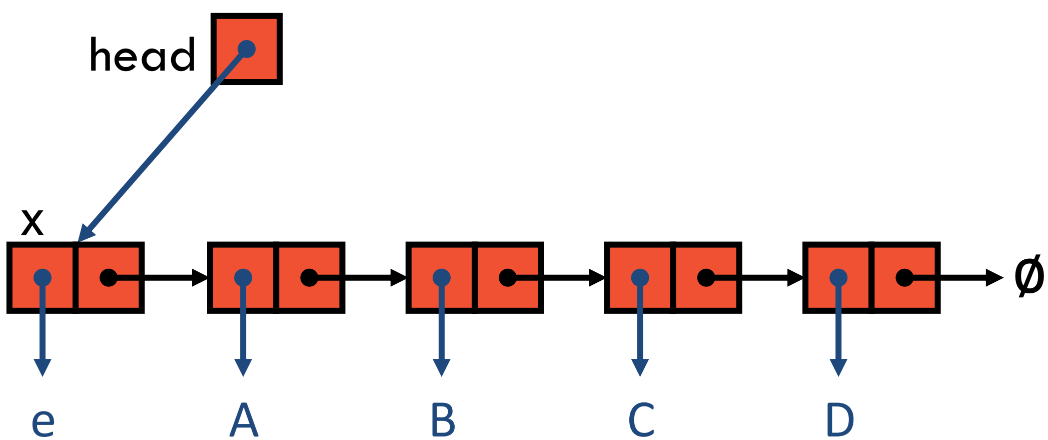
What is the time complexity?



insertFirst(e)

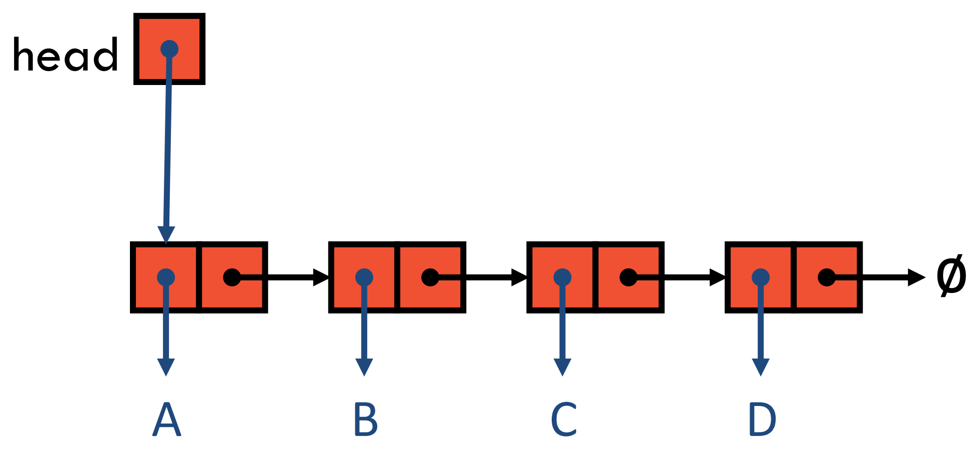
1. Instantiate a new node ***x***
2. Set e as element of x
3. Set x.next to point to (old) head
4. Update list's head to point to x

What is the time complexity? O(1)



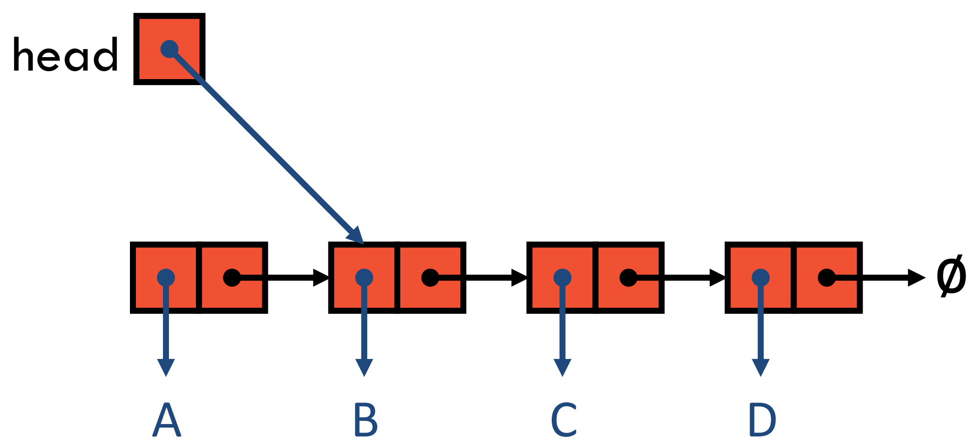
removeFirst()

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



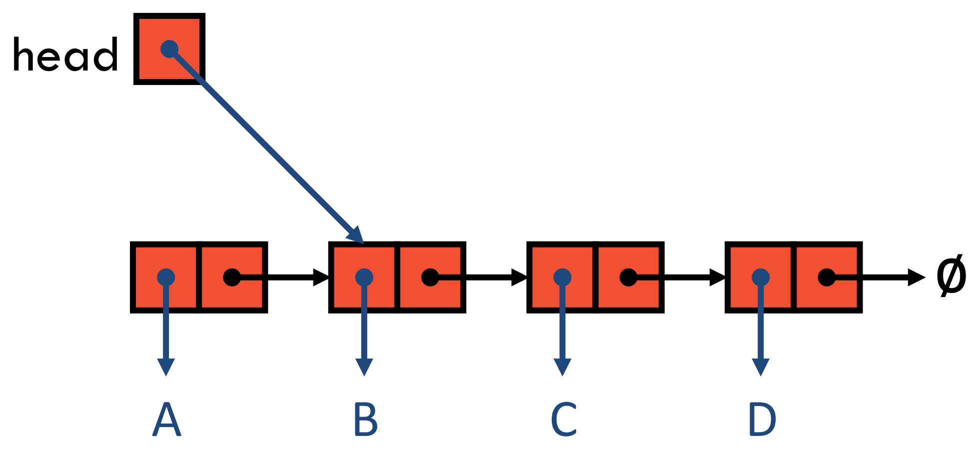
removeFirst()

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



removeFirst()

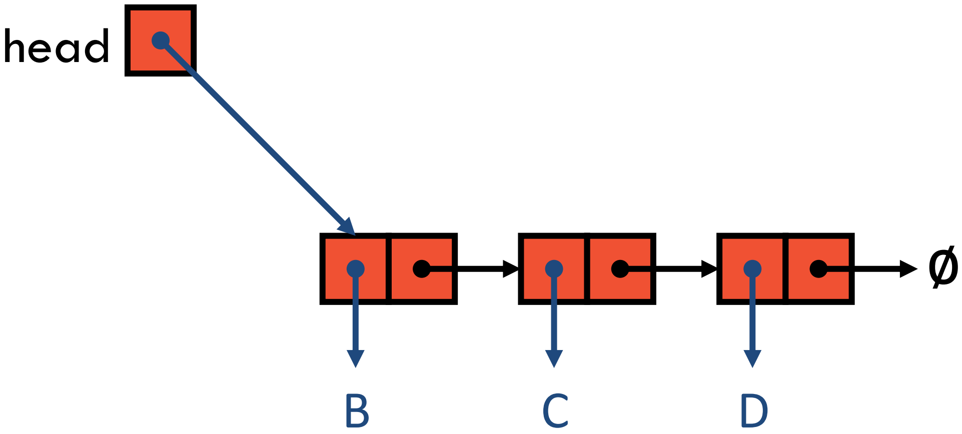
1. Update head to point to next node
2. Delete the former first node



removeFirst()

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

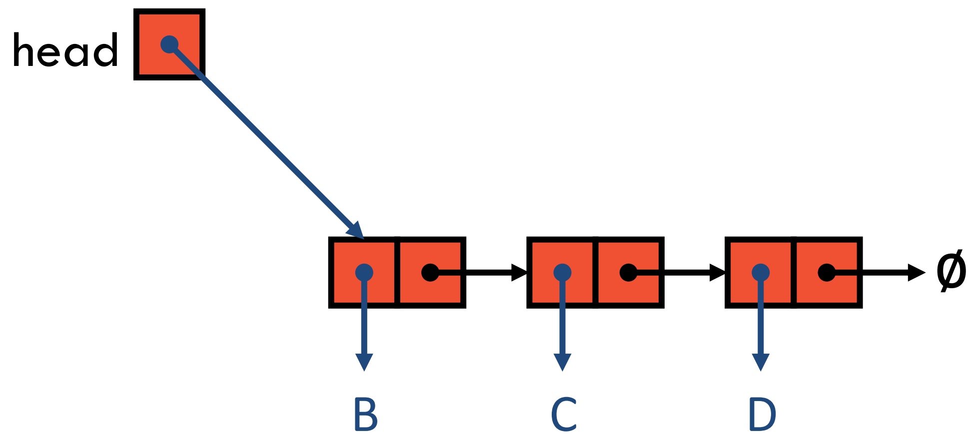
Time complexity?



removeFirst()

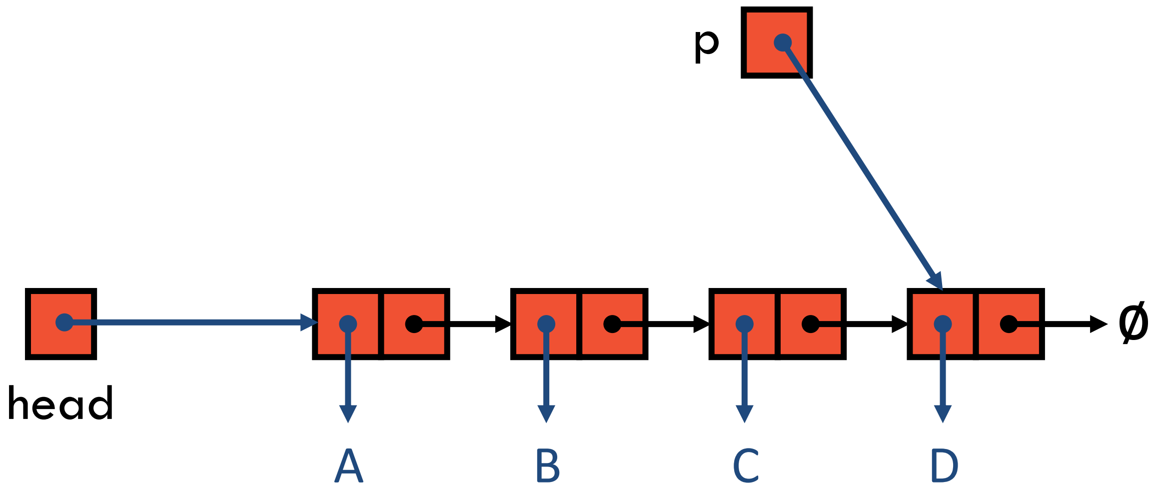
1. Update head to point to next node
2. Delete the former first node

Time complexity? O(1)



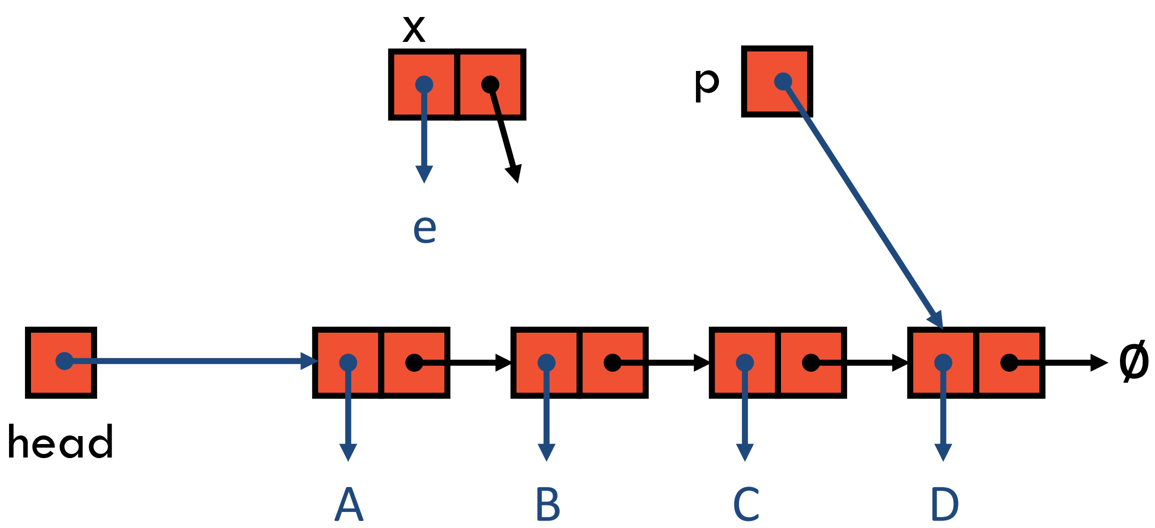
insertBefore(p,e)

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



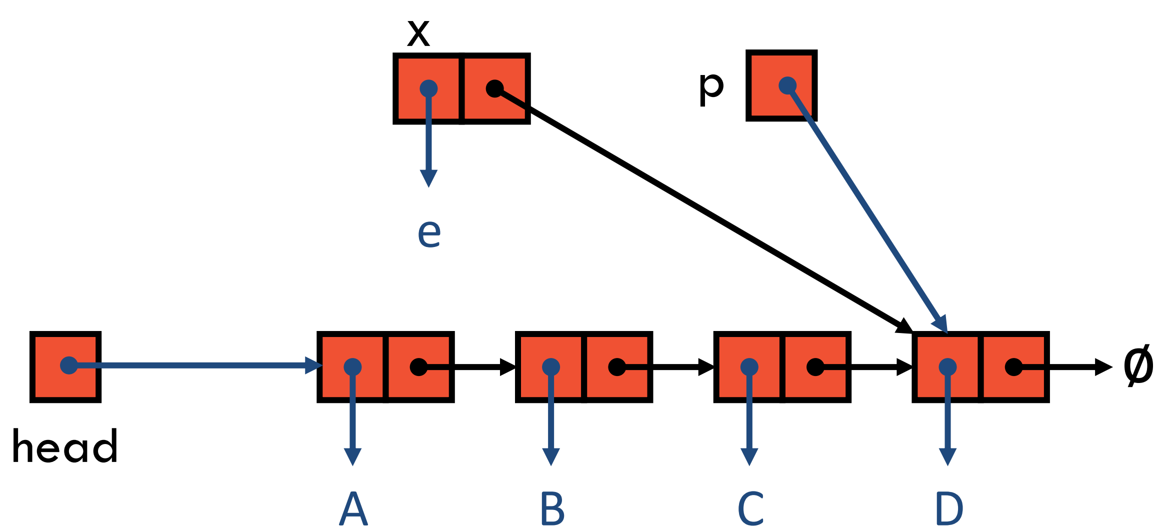
insertBefore(p,e)

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



insertBefore(p,e)

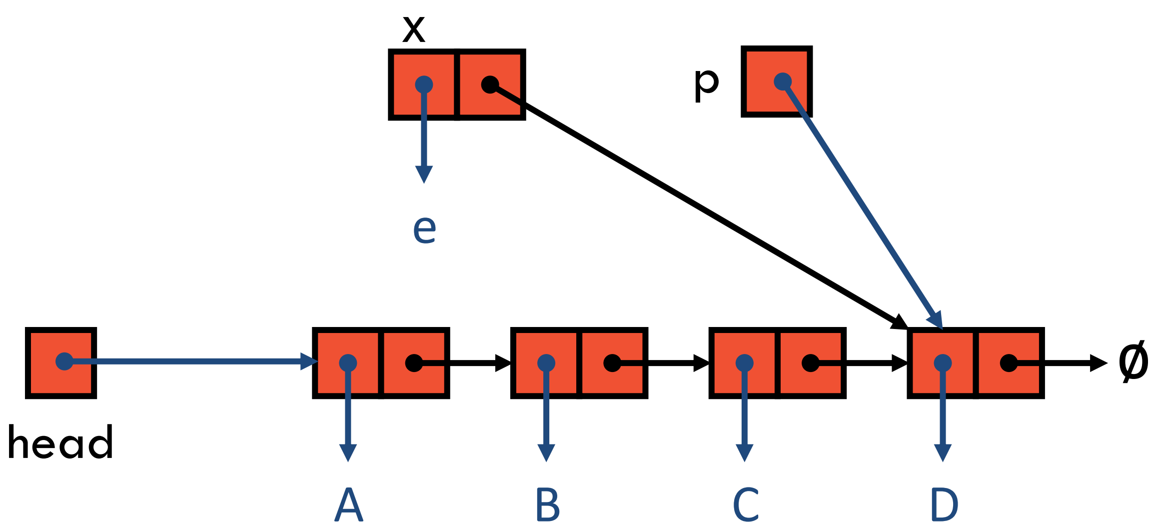
insertBefore(p,e) : insert e in front of the element at position p



What's the next step?

insertBefore(p,e)

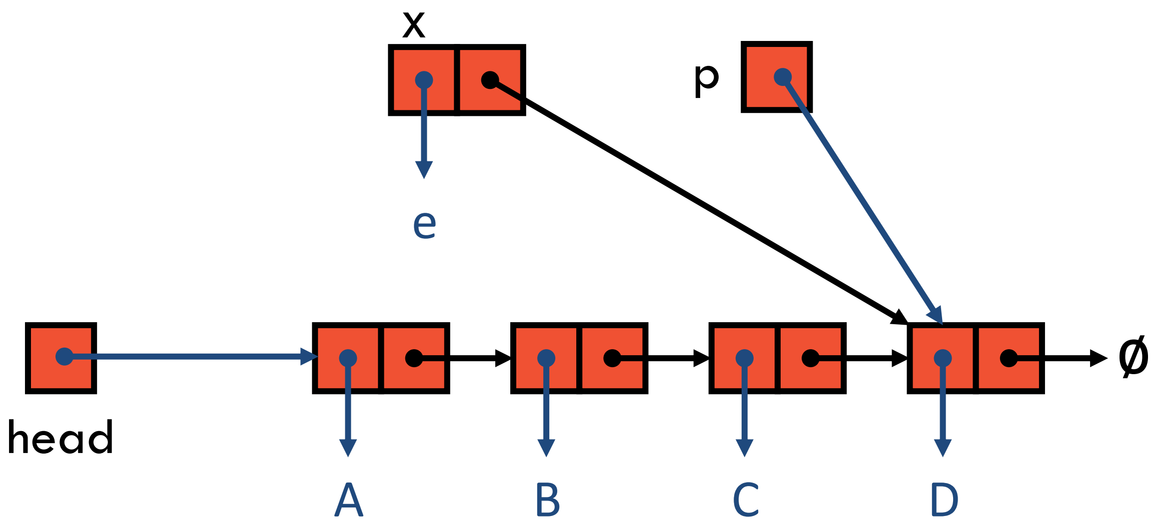
insertBefore(p,e) : insert e in front of the element at position p



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

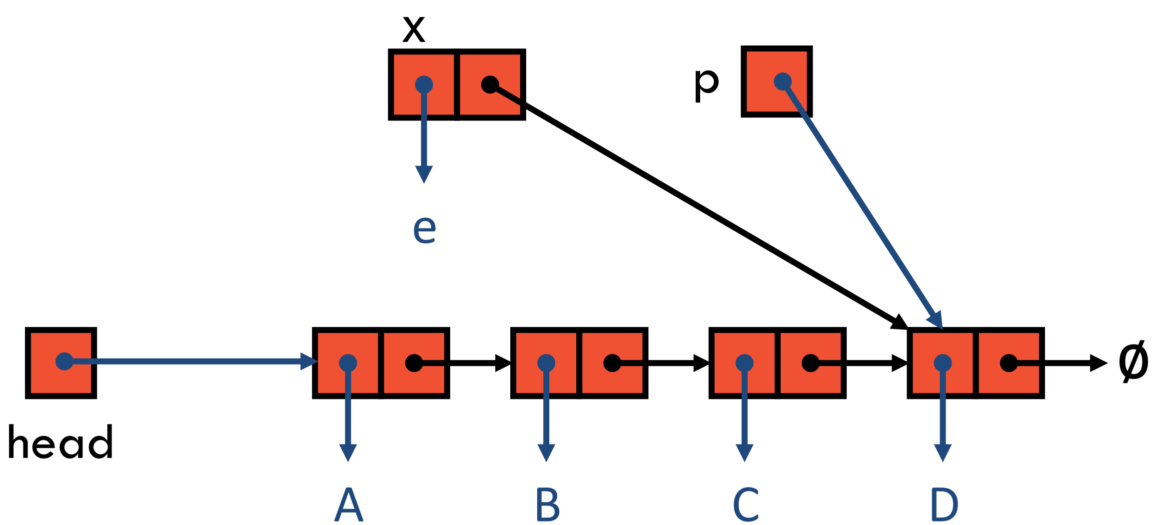
insertBefore(p,e)

To find the predecessor of p we need to follow the links from the “head”. Time complexity?



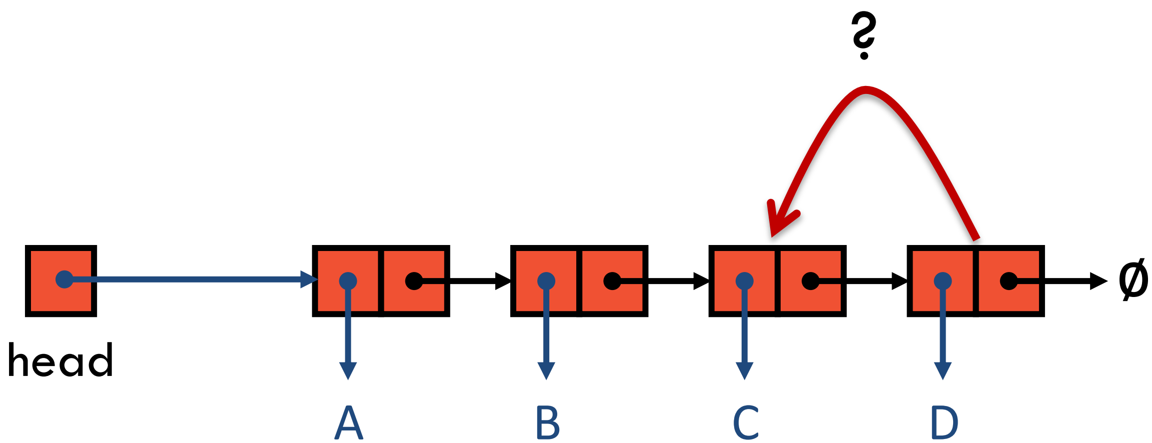
insertBefore(p,e)

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.

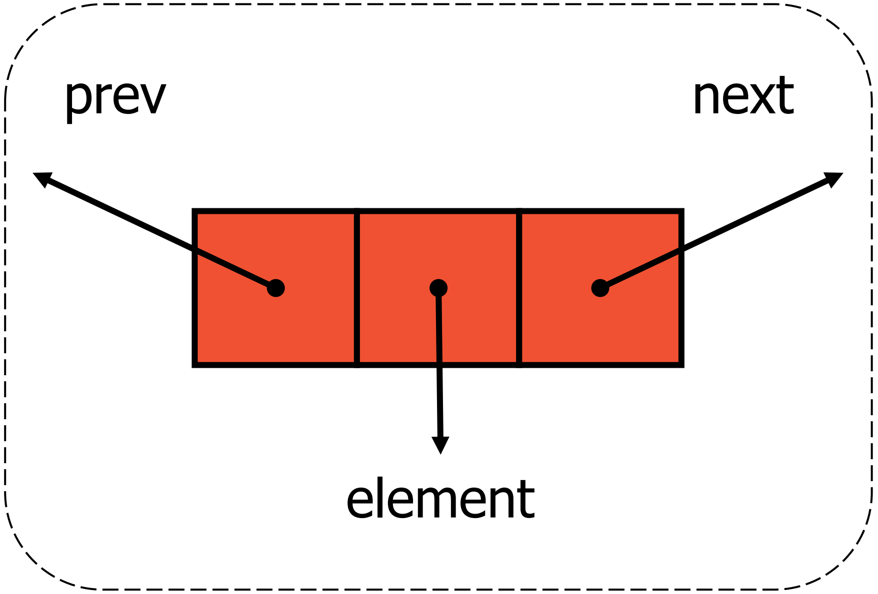


Another attempt

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

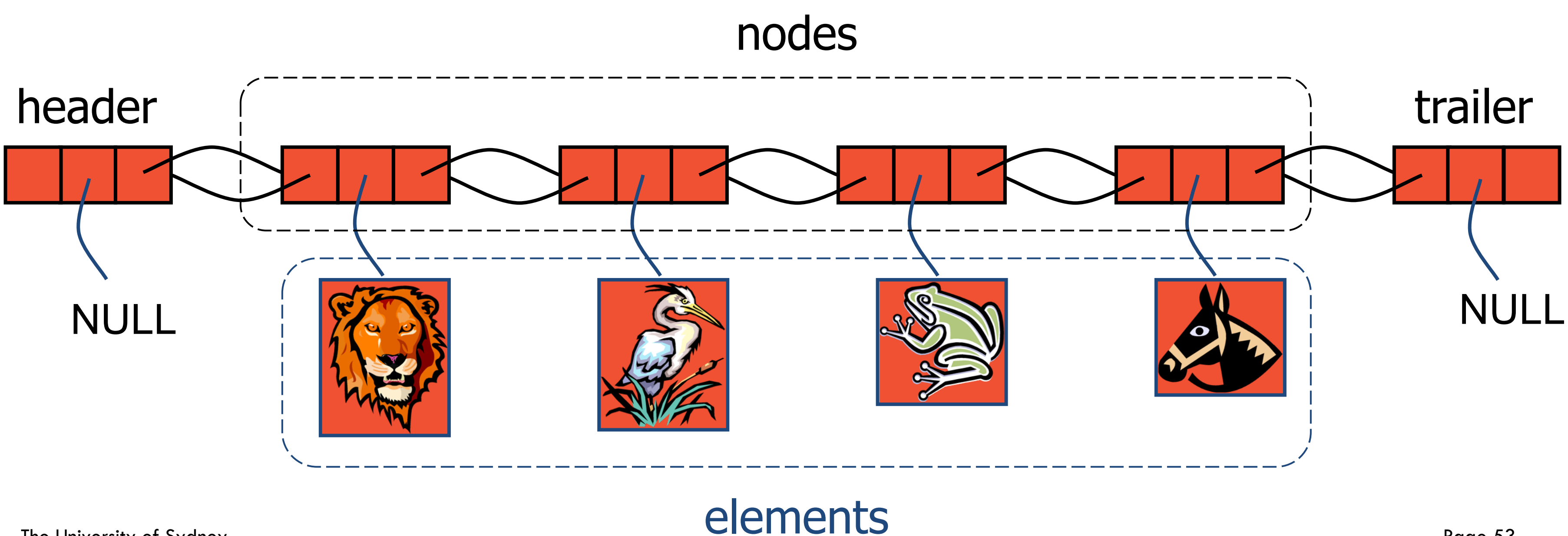
- its element, and
- a link to the previous and next nodes.



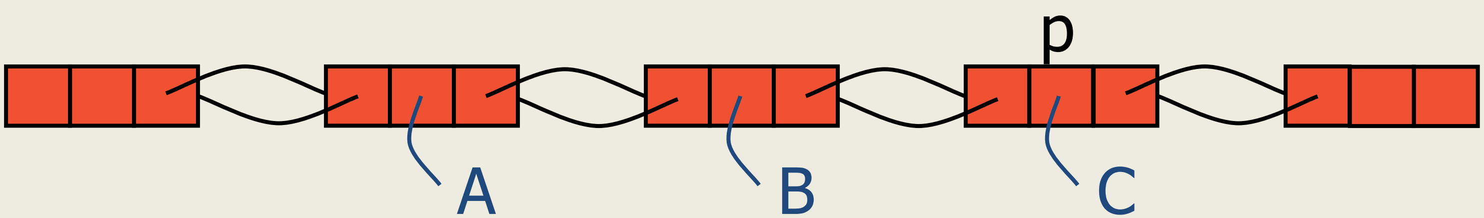
Doubly Linked Lists

A concrete data structure

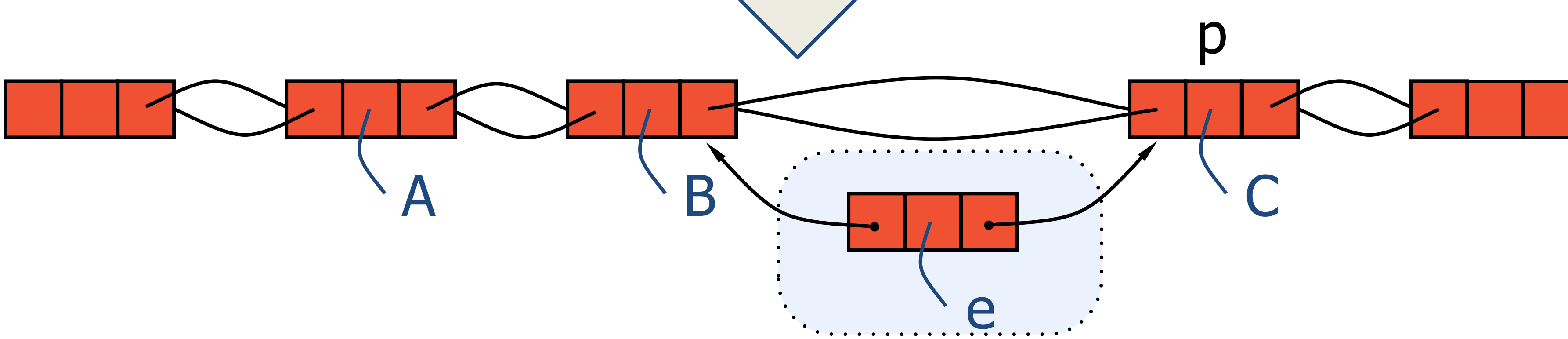
- A sequence of Nodes, each with reference to prev and to next
- List captured by references to its Sentinel Nodes



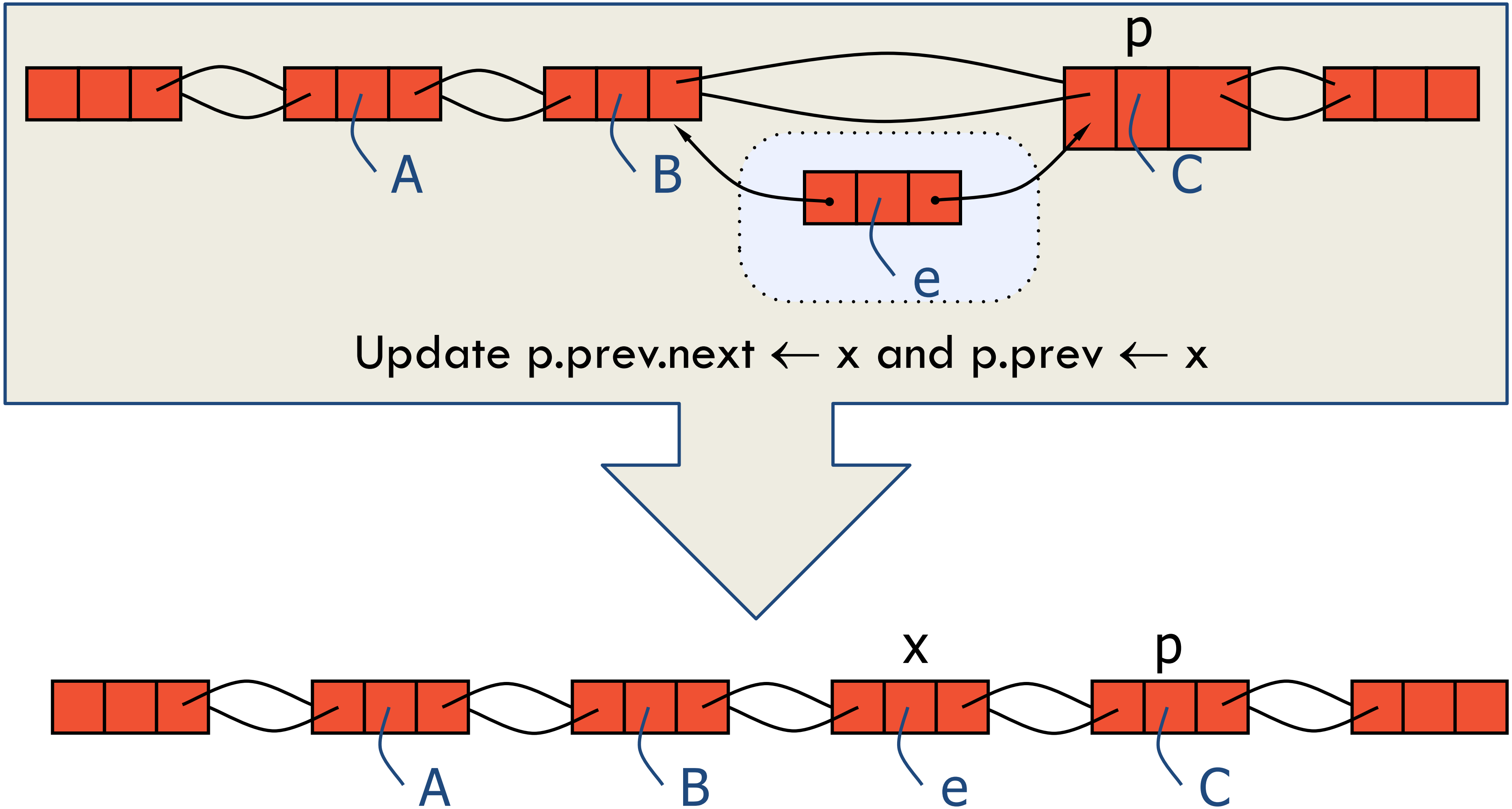
insertBefore(p,e) – step 1



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

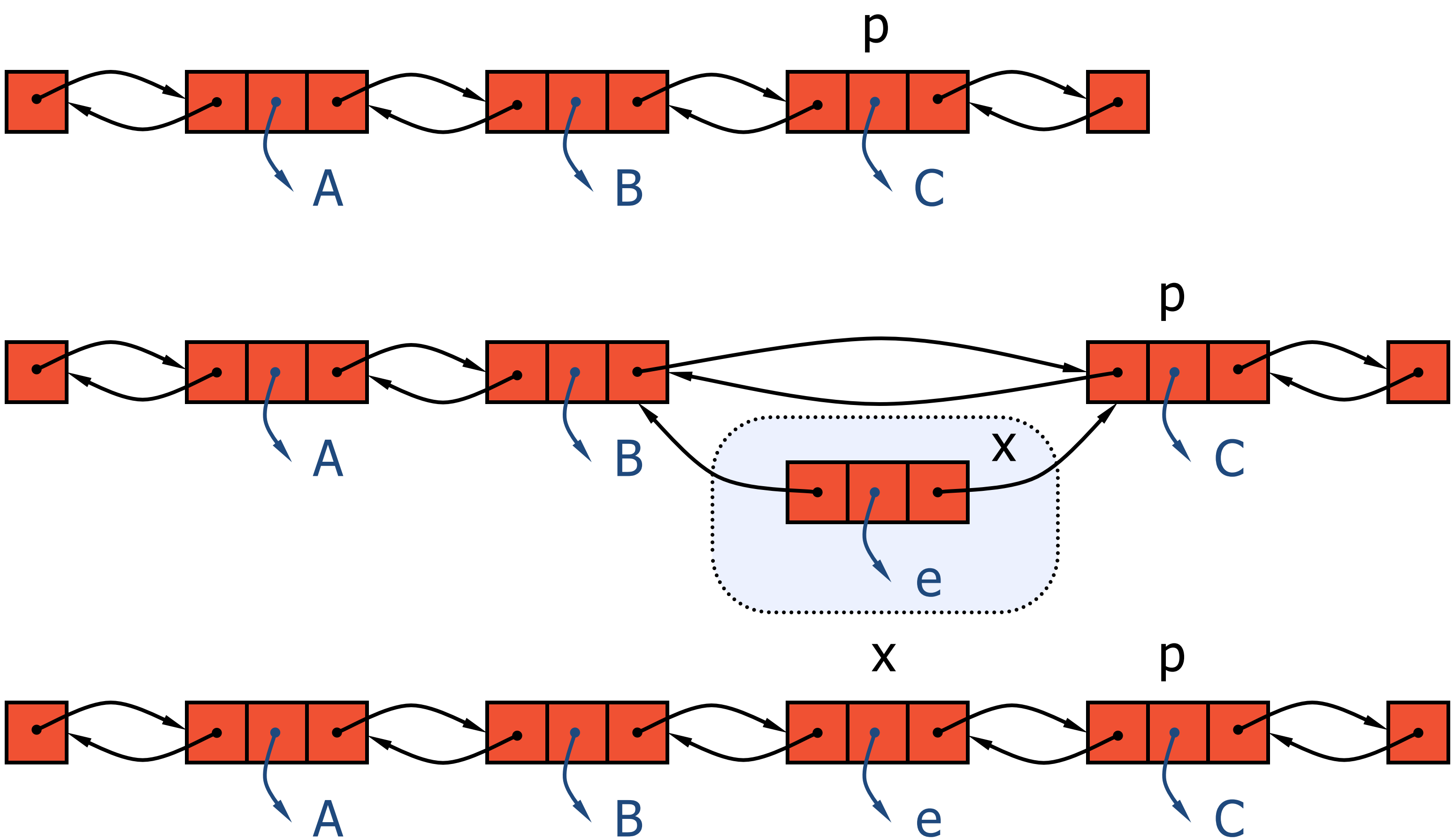


insertBefore(p,e) – step 2



insertBefore(p,e)

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

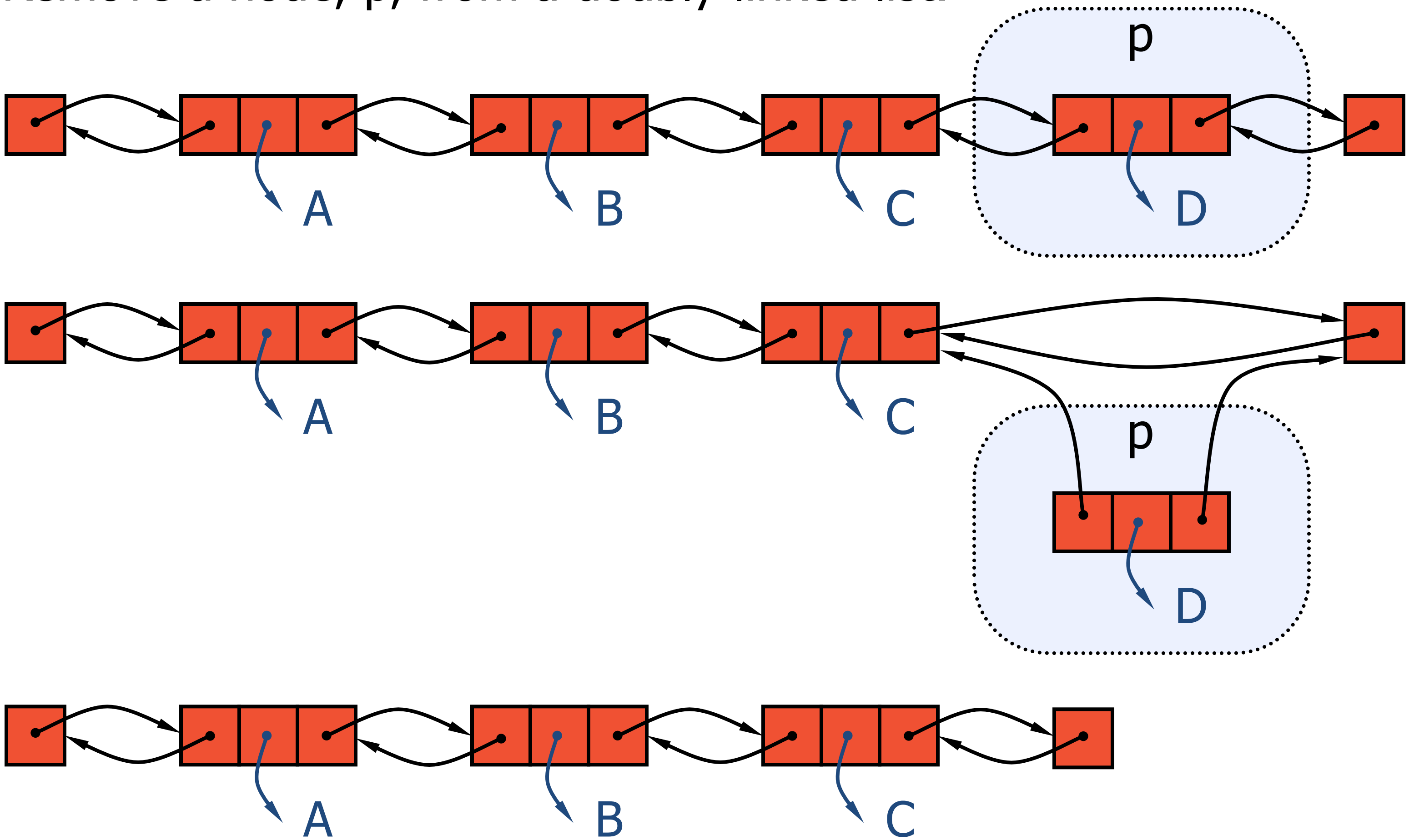


Pseudo-code

```
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 (doubly) 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 __iter__(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
```

Stacks and queues

These ADTs are restricted forms of List, where insertions and removals happen only in particular locations:

- stacks follow last-in-first-out (LIFO)
- queues follows first-in-first-out (FIFO)

So why should we care about a less general ADT?

- operations names are part of computing culture
- numerous applications
- simpler/more efficient implementations than Lists

Stack ADT

Main stack operations:

- push(e): inserts an element, e
- pop(): removes and returns the last inserted element

Auxiliary stack operations:

- top(): returns the last inserted element without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

Stack Example

operation	returns	stack
push(5)	-	[5]
push(3)	-	[5, 3]
size()	2	[5, 3]
pop()	3	[5]
isEmpty()	False	[5]
pop()	5	[]
isEmpty()	True	[]
push(7)	-	[7]
push(9)	-	[7, 9]
top()	9	[7, 9]
push(4)	-	[7, 9, 4]
pop()	4	[7, 9]

Stack Applications

Direct applications

- Keep track of a history that allows undoing such as Web browser history or undo sequence in a text editor
- Chain of method calls in a language supporting recursion
- Context-free grammars

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Method Stacks

The runtime environment keeps track of the chain of active methods with a stack, thus allowing recursion

When a method is called, the system pushes on the stack a frame containing

- Local variables and return value
- Program counter

When a method ends, we pop its frame and pass control to the method on top

```
main() {
  int i ← 5;
  foo(i);
}

foo(int j) {
  int k;
  k ← j+1;
  bar(k);
}

bar(int m) {
  …
}
```

bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

Parentheses Matching

Each "(", "{", or "[" must be paired with a matching ")", "}", or "]"

- correct: ( )(( )){([( )])}
- correct: ((( )(( )){([( )])}))
- incorrect:)(( )){([( )])}
- incorrect: ({[ ])}
- incorrect: (

Scan input string from left to right:

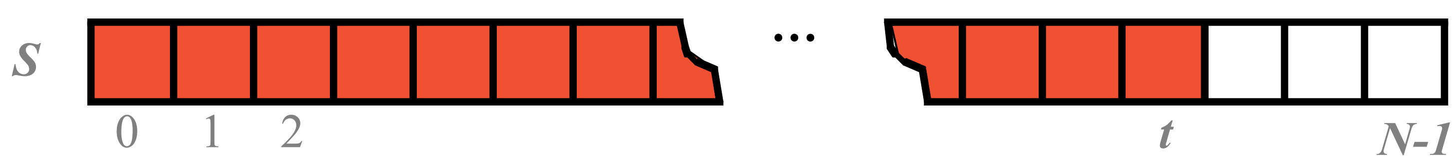
- If we see an opening character, push it to a stack
- If we see a closing character, pop character on stack and check that they match

Stack implementation based on arrays

A simple way of implementing the Stack ADT uses an array:

- Array has capacity N
- Add elements from left to right
- A variable t keeps track of the index of the top element

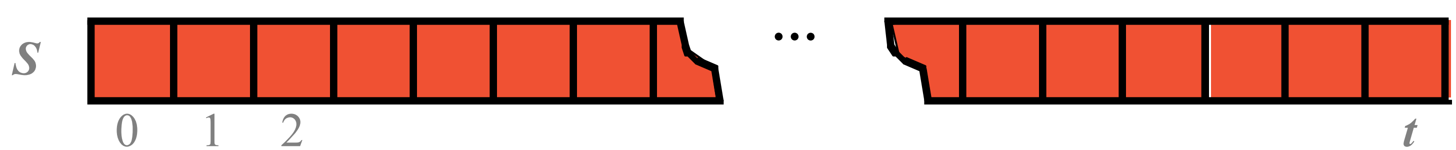
```
def size()
  return t + 1

def pop()
  if isEmpty() then
    return null
  else
    t ← t − 1
    return S[t + 1]
```



Stack implementation based on arrays

- The array storing the stack elements may become full
- A push operation will then either grow the array or signal a "stack overflow" error.

```
def push(e)
  if t = N – 1 then
    return "stack overflow"
  else
    t ← t + 1
    S[t] ← e
```



Stack implementation based on arrays

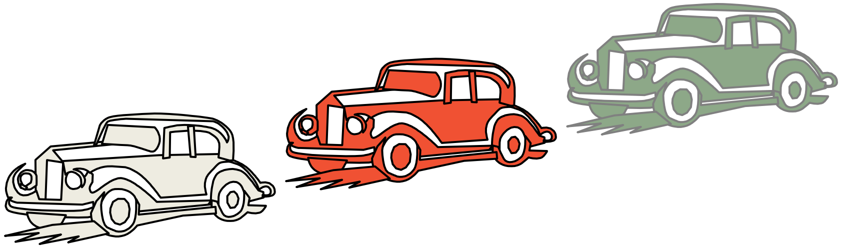
Performance

- The space used is $O(N)$
- Each operation runs in time $O(1)$

Qualifications

- Trying to push a new element into a full stack causes an implementation-specific exception or
- Pushing an item on a full stack causes the underlying array to double in size to make room for new element

Queue ADT



Main queue operations:

- enqueue(e): inserts an element, e, at the end of the queue
- dequeue(): removes and returns element at the front of the queue

Auxiliary queue operations:

- first(): returns the element at the front without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

Boundary cases:

- Attempting the execution of dequeue or first on an empty queue signals an error or returns null

Queue Example

Operation	*Output*	*Queue*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Queue applications

Buffering packets in streams, e.g., video or audio

Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

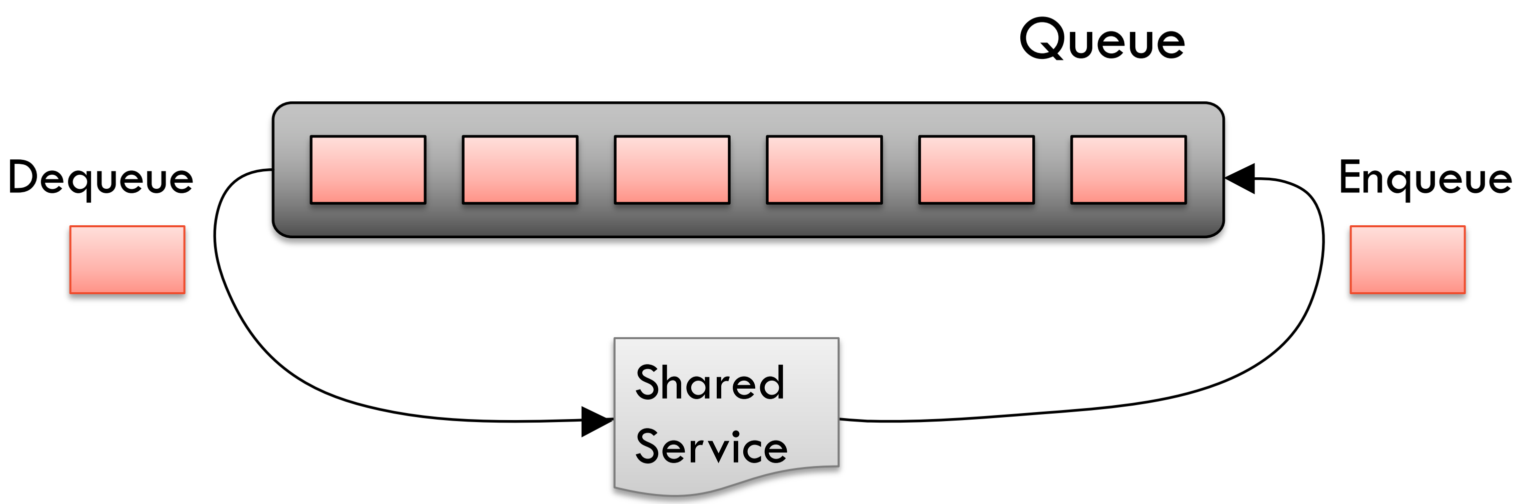
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Queue application: Round Robin Schedulers

Implement a round robin scheduler using a queue Q by repeatedly performing the following steps:

1. $e \leftarrow$ Q.dequeue()
2. Service element e
3. Q.enqueue(e)



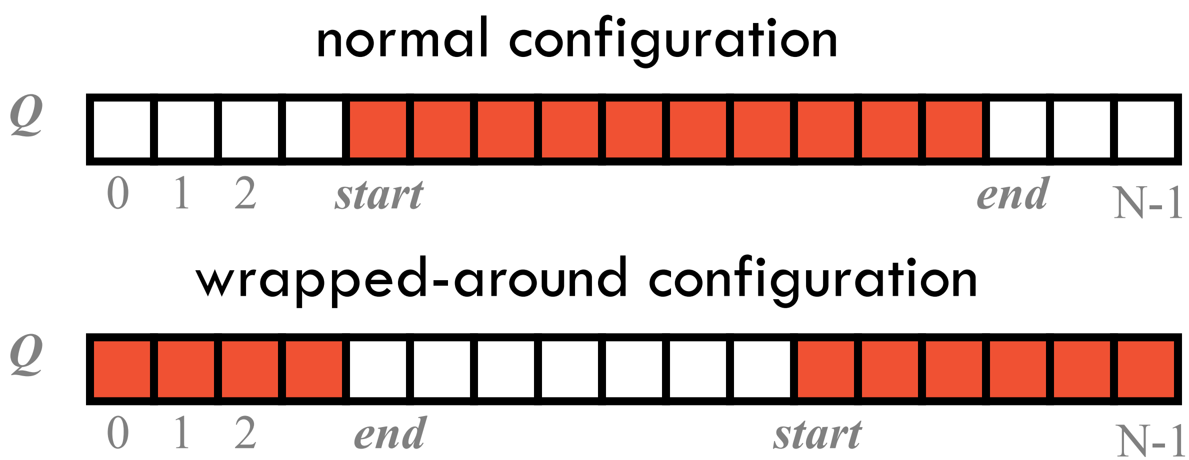
Queue implementation based on arrays

Use an array of size N in a circular fashion

Two variables keep track of the front and size

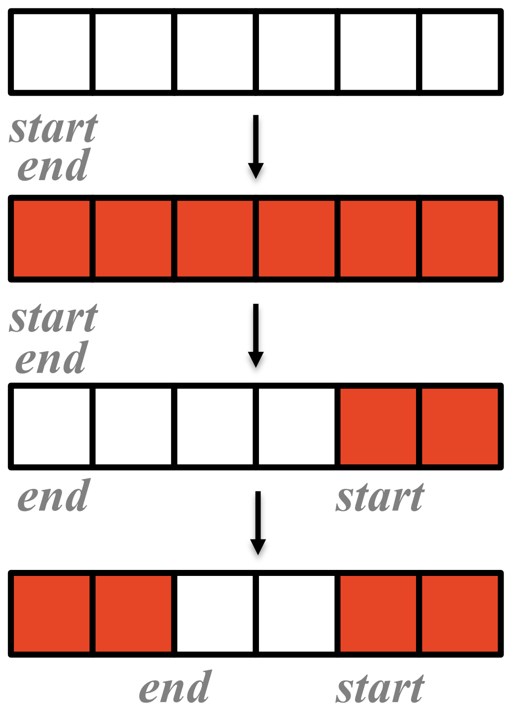
- start : index of the front element
- end : index past the last element
- size : number of stored elements

These are related as follows end = (start + size) mod N, so we only need two, start and size



How to get in a wrapped-around configuration

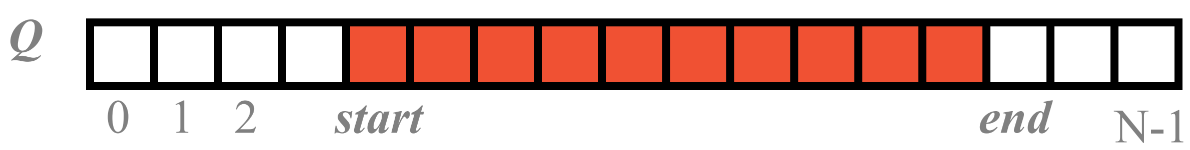
- Enqueue N elements
- Dequeue k < N elements
- Enqueue k’ < k elements

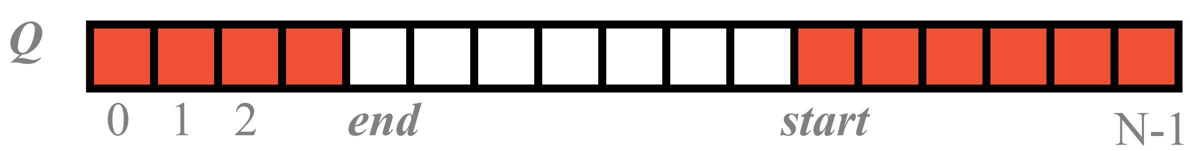


Queue Operations: Enqueue

Return an error if the array is full. Alternatively, we could grow the underlying array as dynamic arrays do

```
def enqueue(e)
  if size = N then
    return "queue full"
  else
    end ← (start + size) mod N
    Q[end] ← e
    size ← size + 1
```



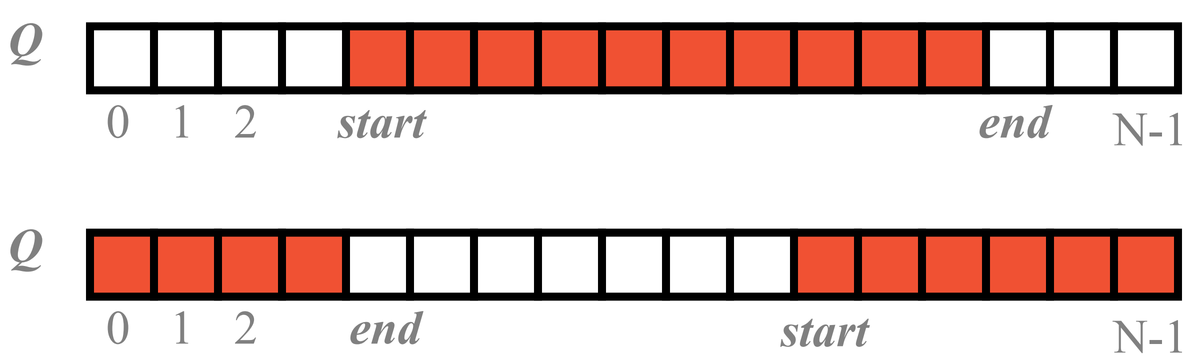


Queue Operations: Dequeue

Note that operation dequeue returns error if the queue is empty

One could alternatively return null

```
def dequeue()
  if isEmpty() then
    return “queue empty”
  else
    e ← Q[start]
    start ← (start + 1) mod N
    size ← (size - 1)
    return e
```



Queue implementation based on arrays

Performance

- The space used is $O(N)$
- Each operation runs in time $O(1)$

Stack implementation based on dynamic arrays

```
def push(e)
  # if we run out of space, double size of S
  if t = N - 1 then
    aux ← new array[2*N]
    for j in [0:N] do
      aux[j] ← S[j]
    S ← aux
    N ← 2*N
  t ← t + 1
  S[t] ← e
```

Amortized analysis

Back to the array-based implementation of lists. Every time we run out of space we double the underlying array.

Recall that worst-case analysis of push takes $O(n)$ time since we may need to double the size of underlying array

Let $T(i)$ be the complexity of i-th operation

$$T(i) = \begin{cases} O(i) & \text{if } i = 2^k \\ O(1) & \text{if } i \neq 2^k \end{cases}$$

Amortized analysis

Starting from the empty list, the total complexity is linear:

Let $T(i)$ be the complexity of the i-th operation

$$\begin{aligned}\Sigma_{i<n} T(i) &= \Sigma_{i\neq 2^k} O(1) + \Sigma_{k<\log n} O(2^k) \\ &= O(n) + O(2^{\log n}) \\ &= O(n)\end{aligned}$$

So even though a single operation can take $O(n)$, we can amortize (average) that cost among n operation.

Amortized analysis definition

We say that a sequence of n operation has $O(f(n))$ amortized time complexity if in the worst-case the total amount of work done by the n operations is no more than $O(n\ f(n))$

For the dynamic array implementation using stack. If we double the size of the array then append takes $O(1)$ amortized time.

Improved space usage

Current solution is wasteful because if we pop too many items the underlying array may be much larger than the current size of the stack

What if we halve the size of S every time $t < N / 2$?

What if we halve the size of S every time $t < N / 4$?

This week

Tutorial Sheets 1: Available on Ed

Quiz 1: Available on Canvas

Assignment 1: Available on Ed and Gradescope