University of
Central Lancashire
UCLan

CO2412 Computational Thinking

Tutorial 1.2 problem
List-like data structures
Python implementation

Where opportunity creates success

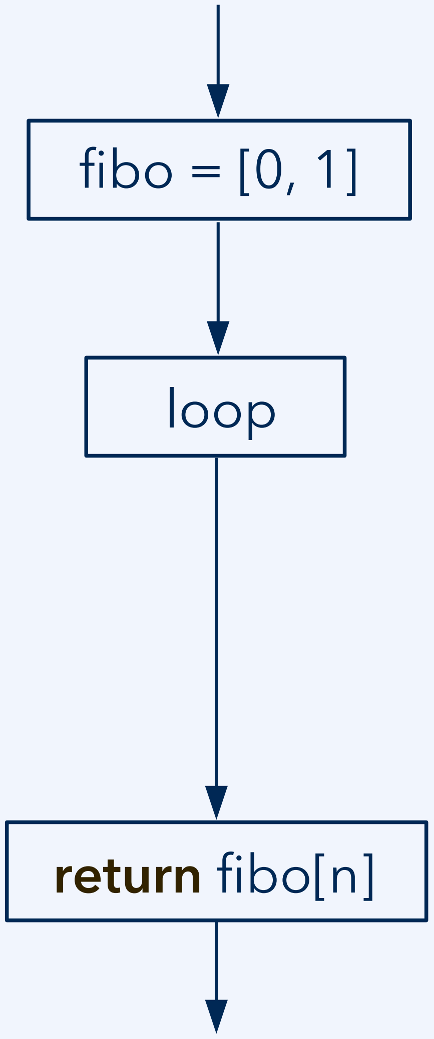
Tutorial 1.2 problem

CO2412

16th November 2021

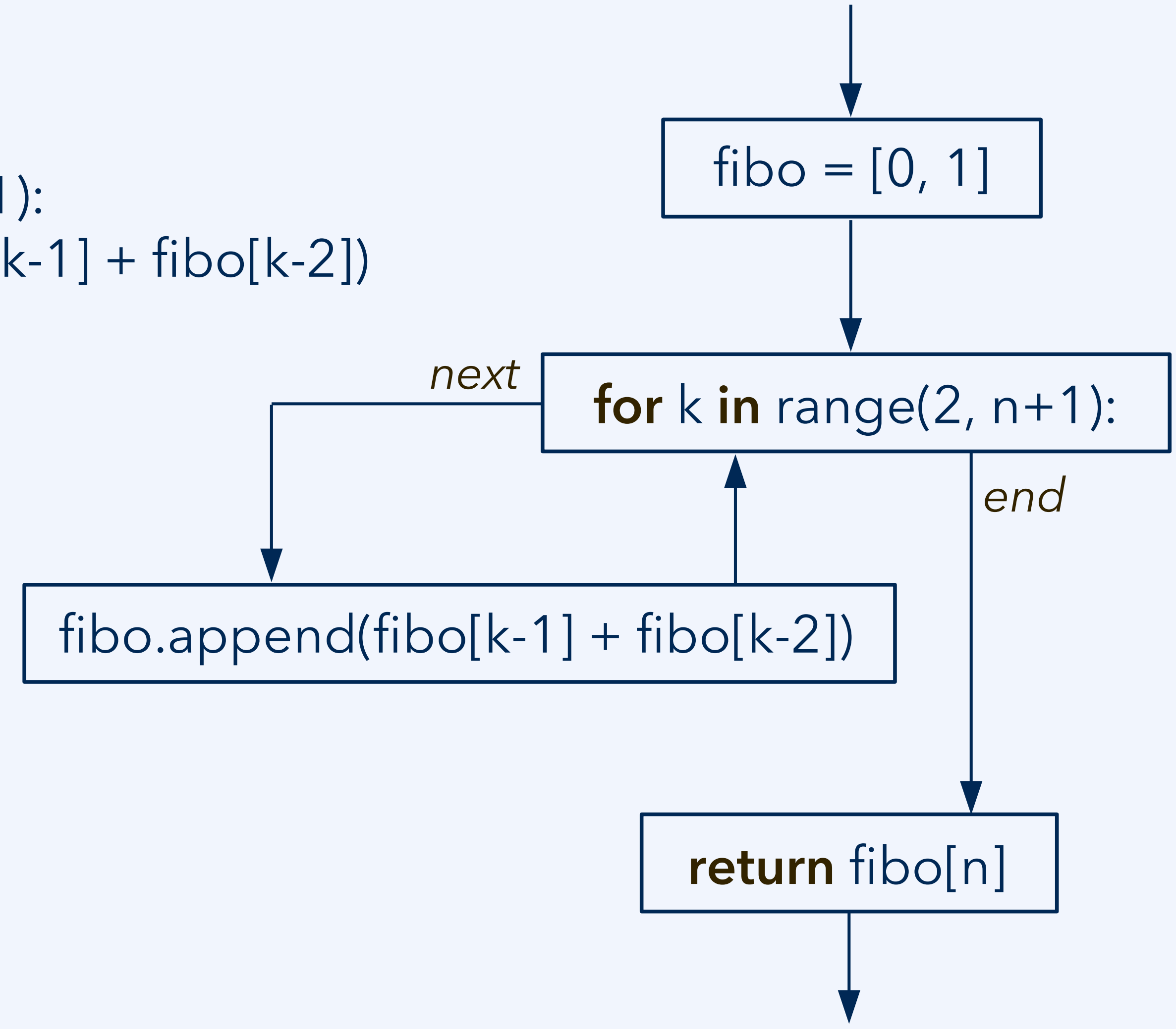
Iterative computation of Fibonacci numbers

```
def fibonacci_iter(n):
    fibo = [0, 1]
    for k in range(2, n+1):
        fibo.append(fibo[k-1] + fibo[k-2])
    return fibo[n]
```



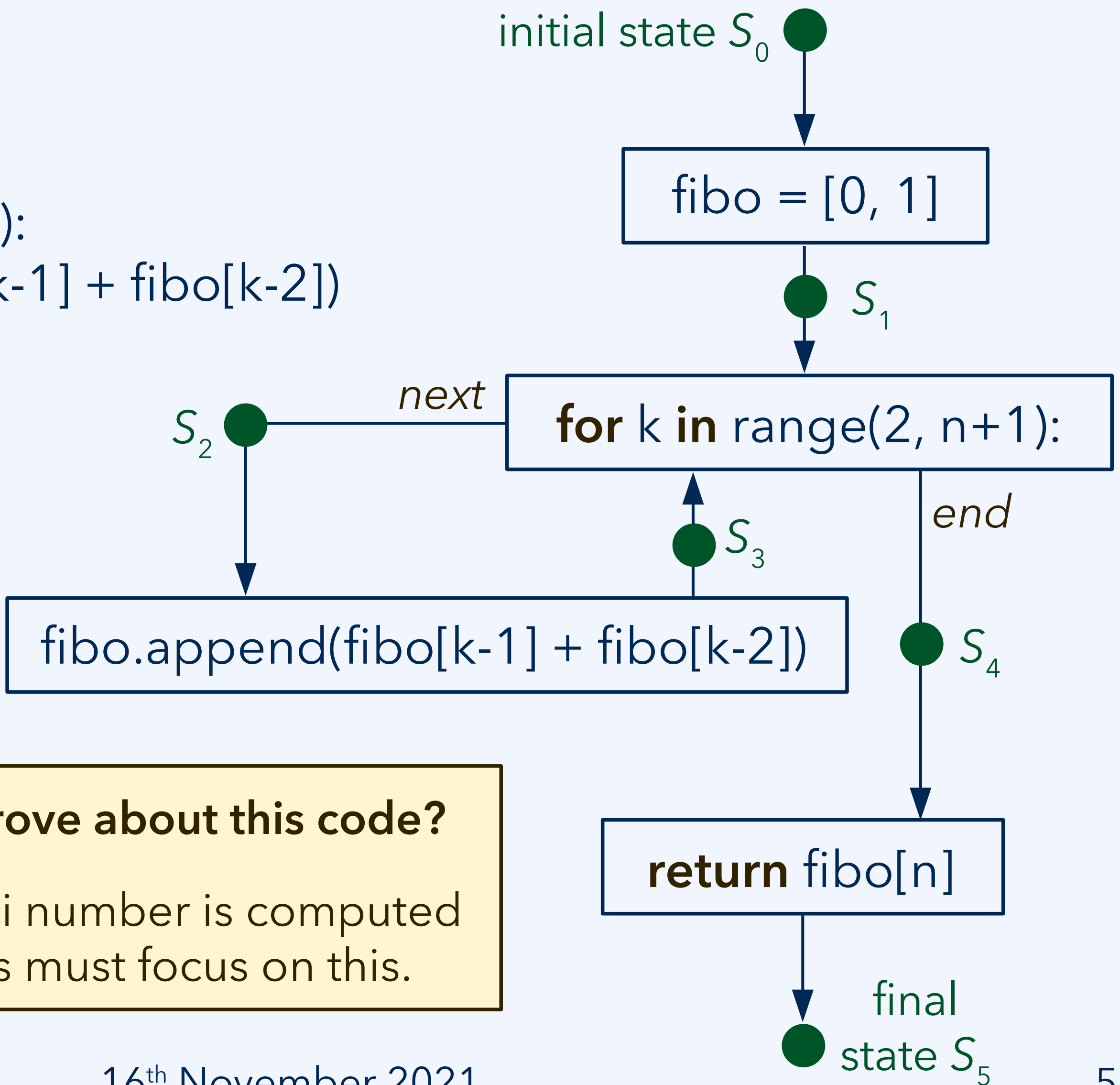
Iterative computation of Fibonacci numbers

```
def fibonacci_iter(n):
    fibo = [0, 1]
    for k in range(2, n+1):
        fibo.append(fibo[k-1] + fibo[k-2])
    return fibo[n]
```



Iterative computation of Fibonacci numbers

```
def fibonacci_iter(n):
    fibo = [0, 1]
    for k in range(2, n+1):
        fibo.append(fibo[k-1] + fibo[k-2])
    return fibo[n]
```



What do we want to prove about this code?

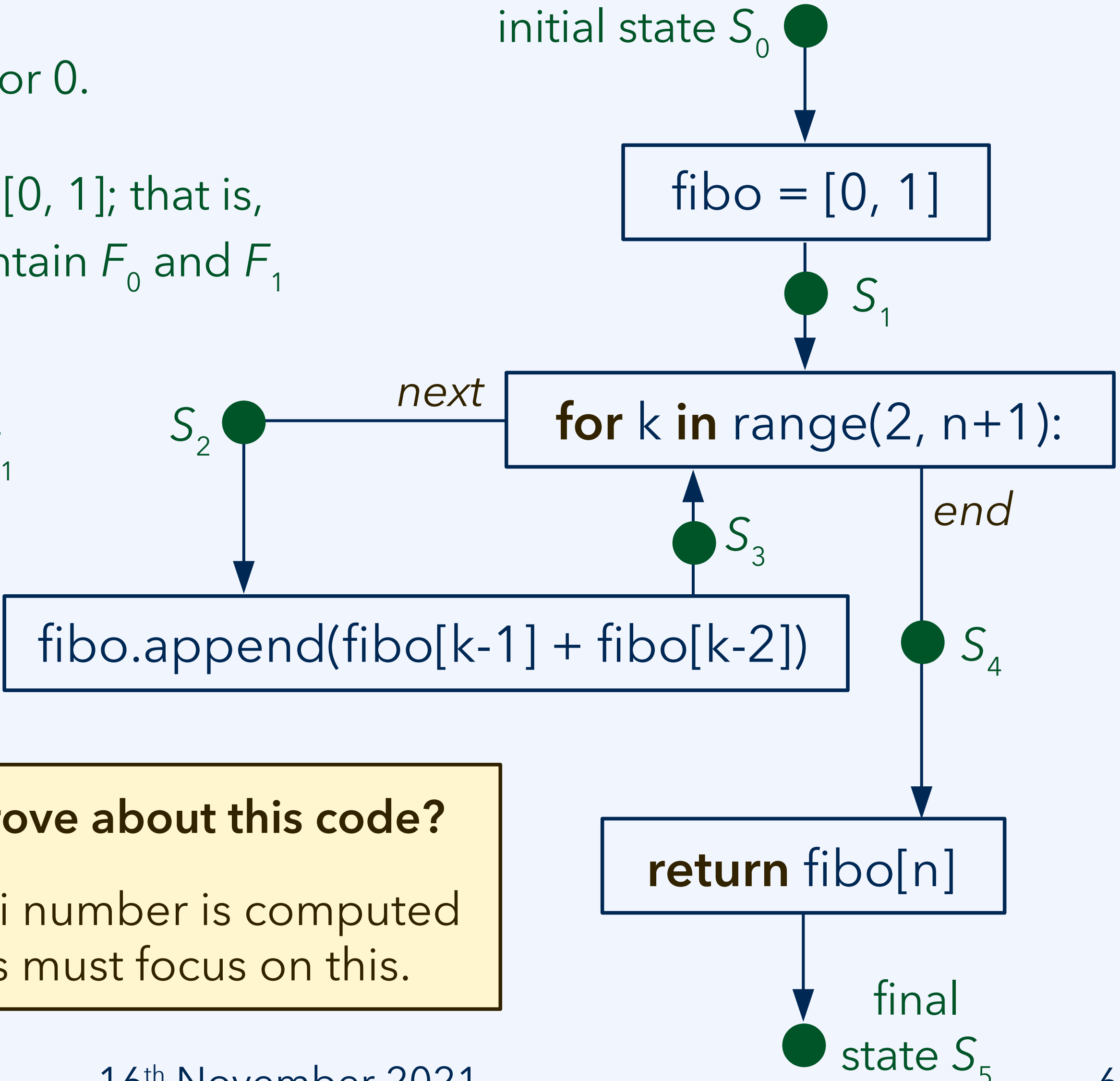
… that the *n*th Fibonacci number is computed correctly. Our analysis must focus on this.

Execution states: First iteration

S_0: n is a natural number or 0.

S_1: as above; also, fibo is [0, 1]; that is, fibo[0] and fibo[1] contain F_0 and F_1

S_2: k ≡ 2 and n ≥ 2; fibo contains F_0 and F_1



What do we want to prove about this code?

… that the *n*th Fibonacci number is computed correctly. Our analysis must focus on this.

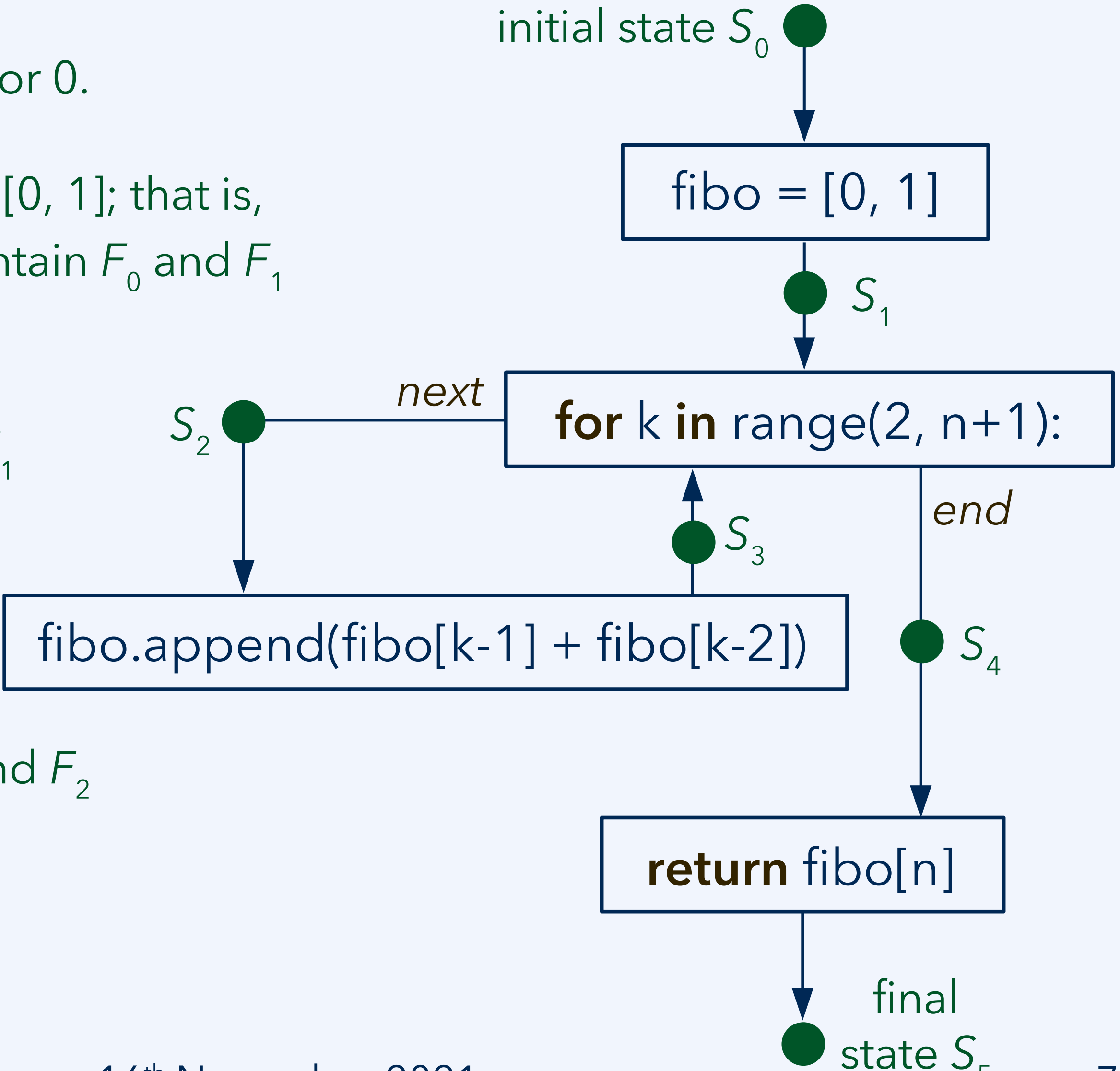
Execution states: First iteration

S_0: n is a natural number or 0.

S_1: as above; also, fibo is [0, 1]; that is, fibo[0] and fibo[1] contain F_0 and F_1

S_2: k ≡ 2 and n ≥ 2; fibo contains F_0 and F_1

S_3: k ≡ 2 and n ≥ 2; fibo contains F_0, F_1, and F_2



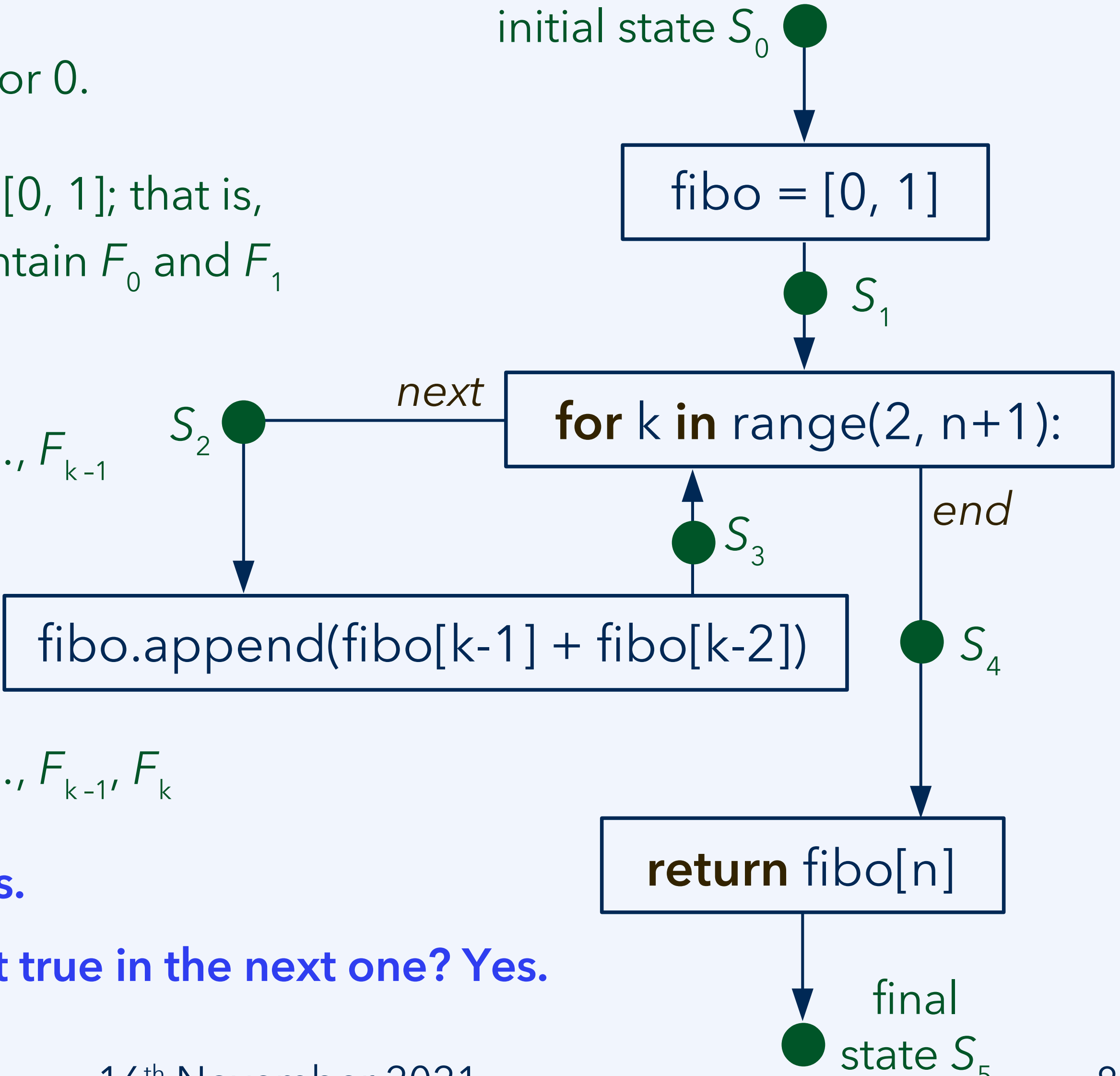
Execution states: Loop invariants

S_0: n is a natural number or 0.

S_1: as above; also, fibo is [0, 1]; that is, fibo[0] and fibo[1] contain F_0 and F_1

S_2: 2 ≤ k ≤ n; fibo contains $F_0, F_1, \ldots, F_{k-1}$

S_3: 2 ≤ k ≤ n; fibo contains $F_0, F_1, \ldots, F_{k-1}, F_k$



Is it true the first time? Yes.

If true in one iteration, is it true in the next one? Yes.

Execution states and proof of correctness

S_0: n is a natural number or 0.

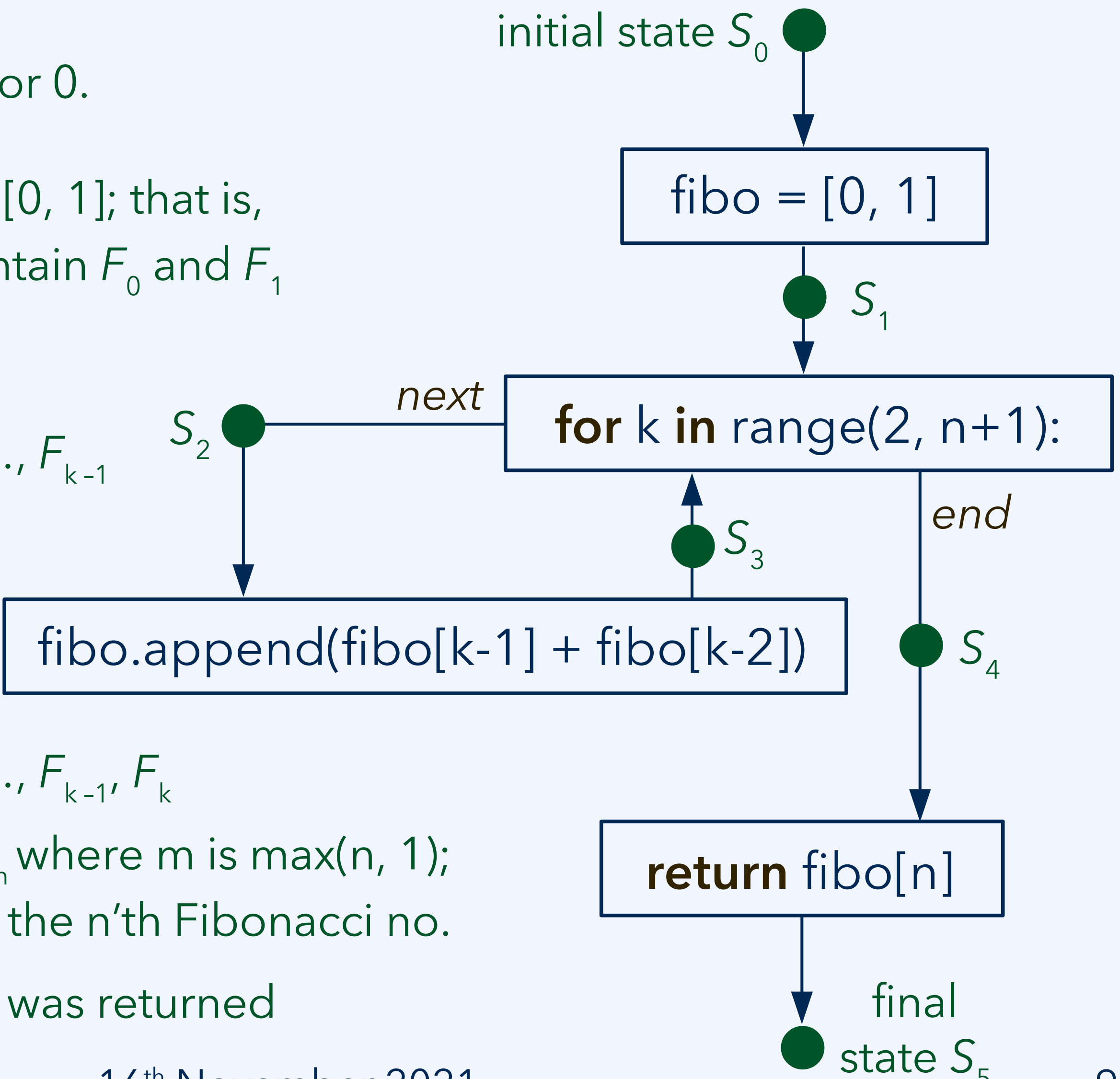
S_1: as above; also, fibo is [0, 1]; that is, fibo[0] and fibo[1] contain F_0 and F_1

S_2: $2 \le k \le n$; fibo contains $F_0, F_1, \ldots, F_{k-1}$

S_3: $2 \le k \le n$; fibo contains $F_0, F_1, \ldots, F_{k-1}, F_k$

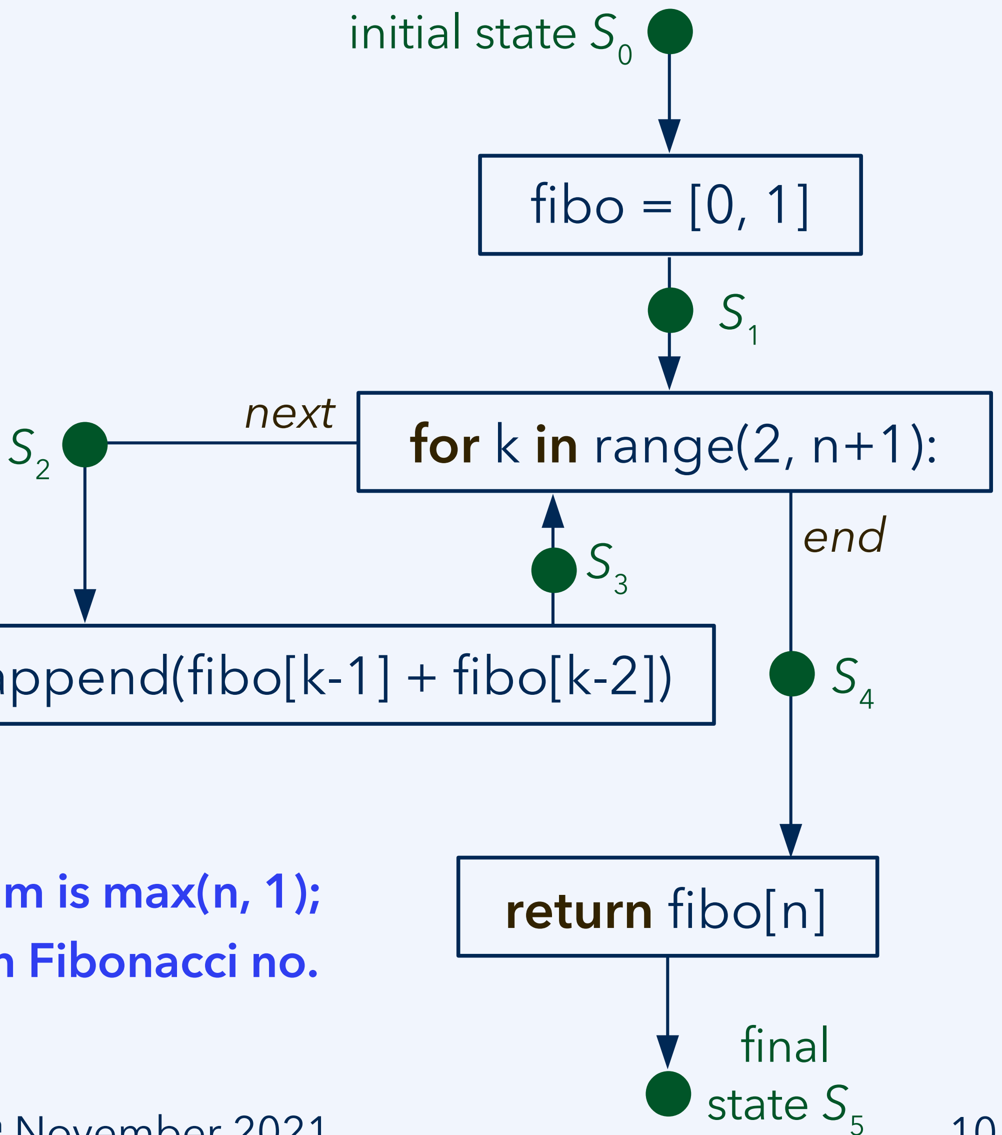
S_4: fibo contains $F_0, \ldots, F_m$ where m is max(n, 1); in particular, fibo[n] is the n'th Fibonacci no.

S_5: the n'th Fibonacci no. was returned



Fibonacci code: Space efficiency

```
def fibonacci_iter(n):
    fibo = [0, 1]
    for k in range(2, n+1):
        fibo.append(fibo[k-1] \
                    + fibo[k-2])
    return fibo[n]
```



List size: O(n)

S_4: fibo contains F_0, …, F_m where m is max(n, 1); in particular, fibo[n] is the n'th Fibonacci no.

Fibonacci code: Memory optimization

```
def fibonacci_iter(n):
    fibo = [0, 1]
    for k in range(2, n+1):
        fibo.append(fibo[k-1] \
                        + fibo[k-2])
    return fibo[n]
```

List size: O(n)

S_4: fibo contains F_0, …, F_m where m is max(n, 1);
in particular, fibo[n] is the n'th Fibonacci no.

```
def fibonacci_iter(n):
    if n == 0:
        return 0
    F_k_minus_one, F_k = 0, 1  # k = 1
    for k in range(2, n+1):
        F_k_minus_two = F_k_minus_one
        F_k_minus_one = F_k
        F_k = F_k_minus_one \
                + F_k_minus_two
    return F_k  # k = n
```

Fibonacci code: Memory optimization

O(n) space code

```
def fibonacci_iter(n):
    fibo = [0, 1]
    for k in range(2, n+1):
        fibo.append(fibo[k-1] \
                          + fibo[k-2])
    return fibo[n]
```

List size: O(n)

S_4: fibo contains F_0, …, F_m where m is max(n, 1); in particular, fibo[n] is the n'th Fibonacci no.

O(1) space code

```
def fibonacci_iter(n):
    if n == 0:
        return 0
    F_k_minus_one, F_k = 0, 1  # k = 1
    for k in range(2, n+1):
        F_k_minus_two = F_k_minus_one
        F_k_minus_one = F_k
        F_k = F_k_minus_one \
                  + F_k_minus_two
    return F_k  # k = n
```

constant number of elementary variables

O(1) space

University of Central Lancashire UCLan

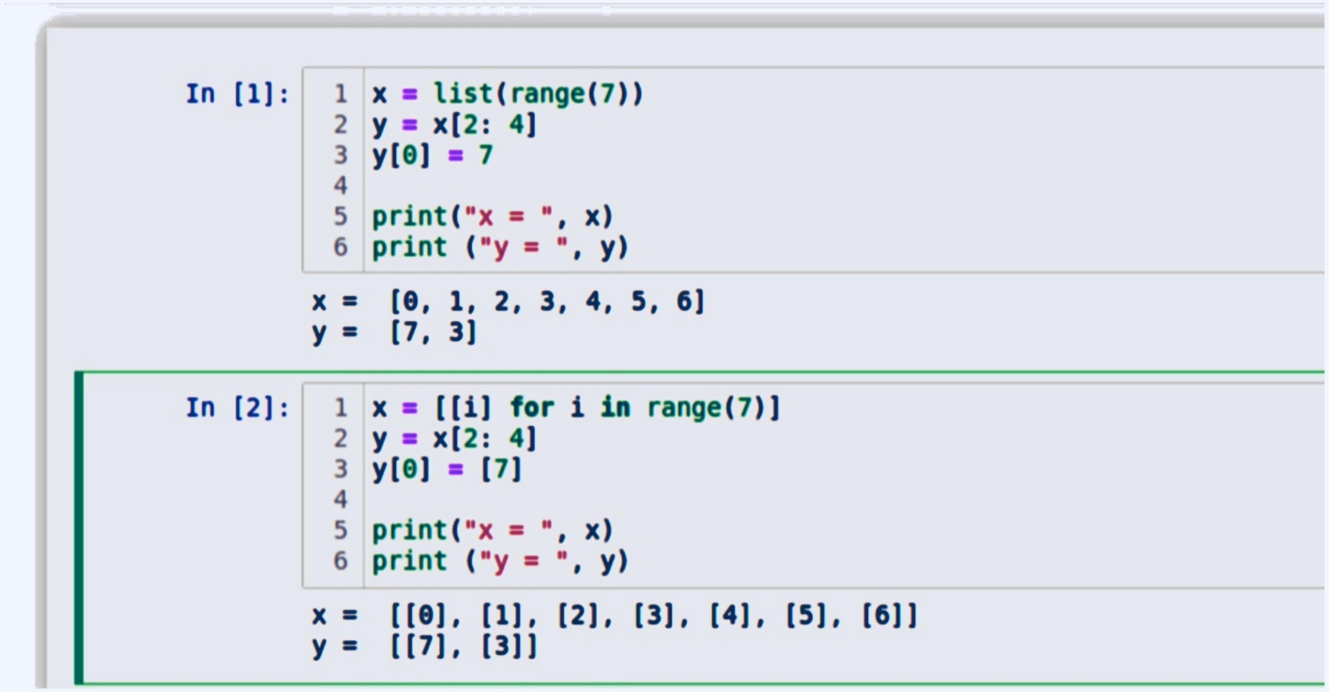
List-like data structures

CO2412

16th November 2021

Dynamic arrays

Lists in Python are implemented as dynamic arrays. Their elements behave in the same way as Python variables do in general: For elementary data types such as numbers, they contain the value, otherwise they are object references.



In [1]:
```
x = list(range(7))
y = x[2: 4]
y[0] = 7

print("x = ", x)
print ("y = ", y)
```
```
x =  [0, 1, 2, 3, 4, 5, 6]
y =  [7, 3]
```

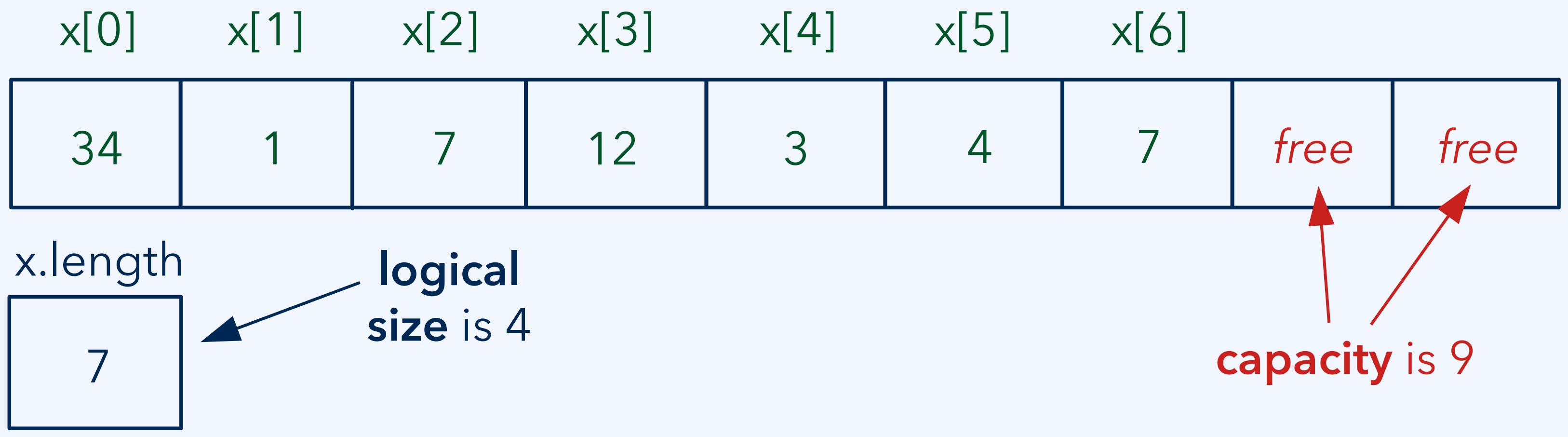
In [2]:
```
x = [[i] for i in range(7)]
y = x[2: 4]
y[0] = [7]

print("x = ", x)
print ("y = ", y)
```
```
x =  [[0], [1], [2], [3], [4], [5], [6]]
y =  [[7], [3]]
```

When a sublist such as x[2: 4] is created from x, the sublist elements are copied.

Dynamic arrays

Dynamic data structures can change in size and/or structure at runtime. For an array, this can be implemented by **allocating reserve memory** for any elements that may be appended in the future. When the capacity of the dynamic array is exhausted, all of its contents need to be shifted to another position in memory.

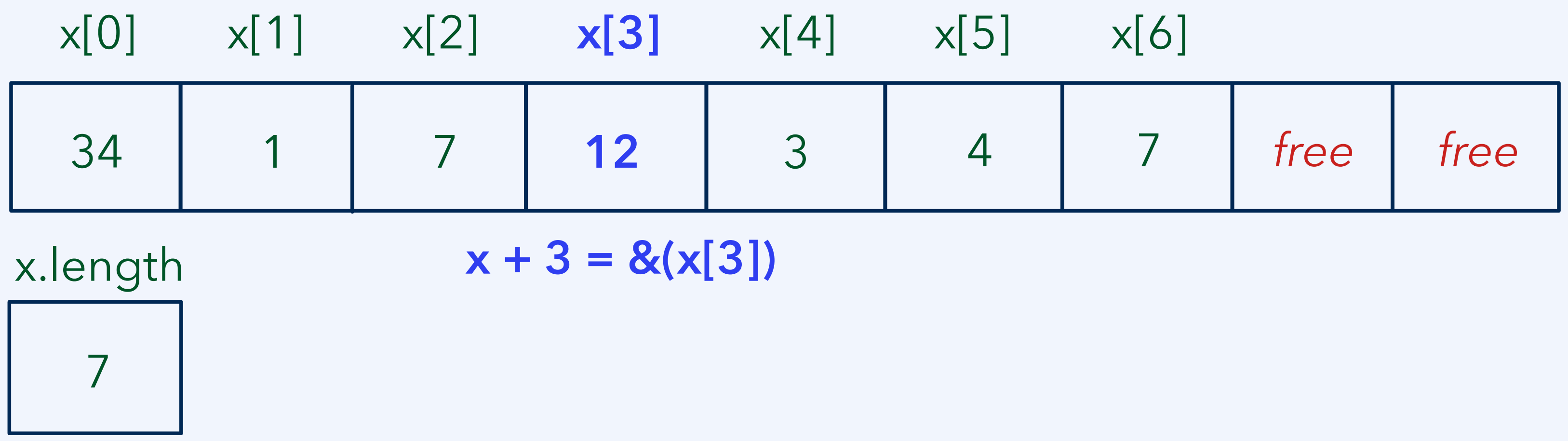
An array contains a sequence of elements of the same type, arranged **contiguously in memory**. The compiler/interpreter, and in some languages the programmer, can use **pointer arithmetics for converting indices to addresses**.



Dynamic arrays: Efficiency analysis

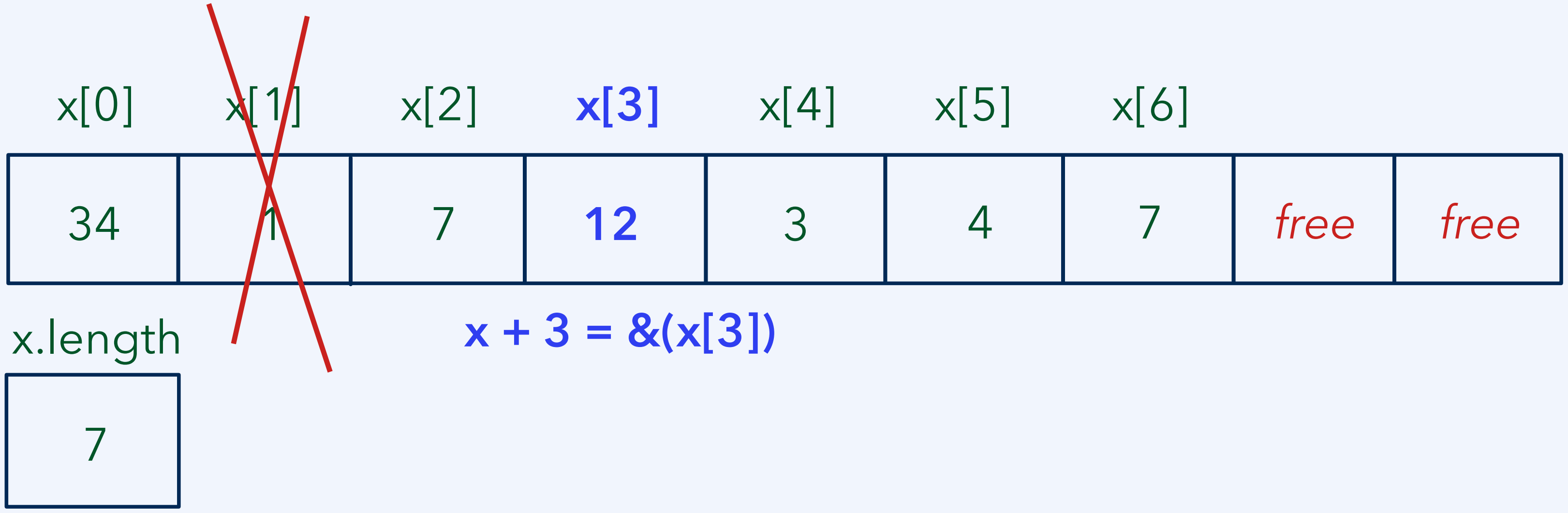
- **Read/write access to an array element: O(1) time.**
  Address of the i-th element computable by pointer arithmetics.

An array contains a sequence of elements of the same type, arranged **contiguously in memory**. The compiler/interpreter, and in some languages the programmer, can use **pointer arithmetics for converting indices to addresses**.



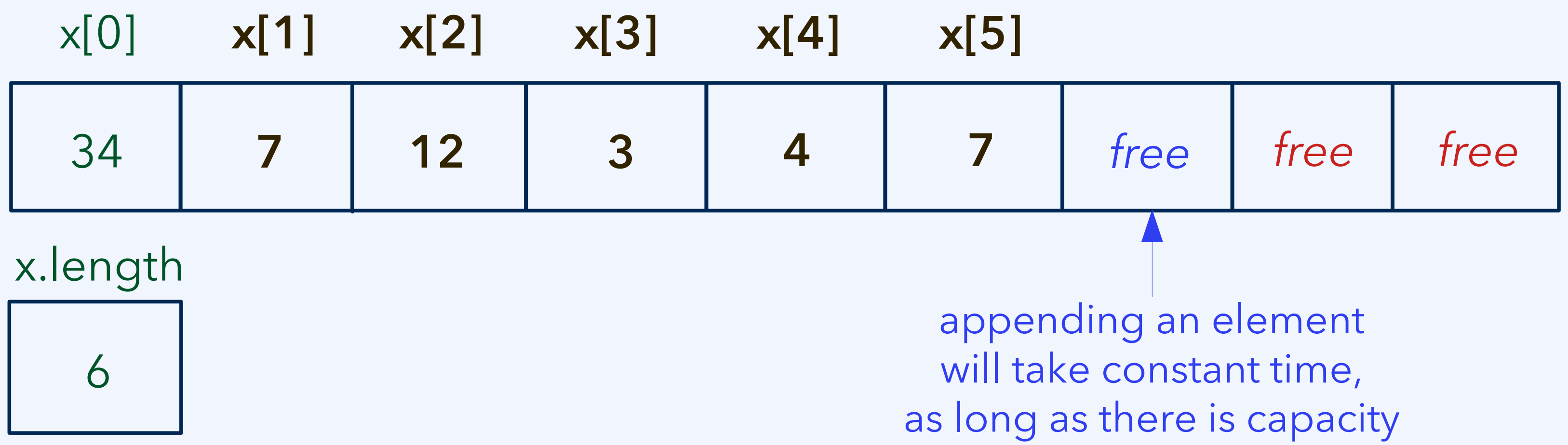
Dynamic arrays: Efficiency analysis

- **Read/write access to an array element: O(1) time.**
  Address of the i-th element computable by pointer arithmetics.
- **Deleting an element from the array?** O(1) at the end, O(*n*) elsewhere.
  All the elements with greater indices need to be shifted.



Dynamic arrays: Efficiency analysis

- Read/write access to an array element: O(1) time.
  Address of the i-th element computable by pointer arithmetics.
- Deleting an element from the array: **O(1) at the end, O(*n*) elsewhere. All the elements with greater indices need to be shifted.**
- **Extending the array by one element?** O(1) at the end, if there is capacity. O(*n*) elsewhere, or if the capacity of the dynamic array is exhausted.

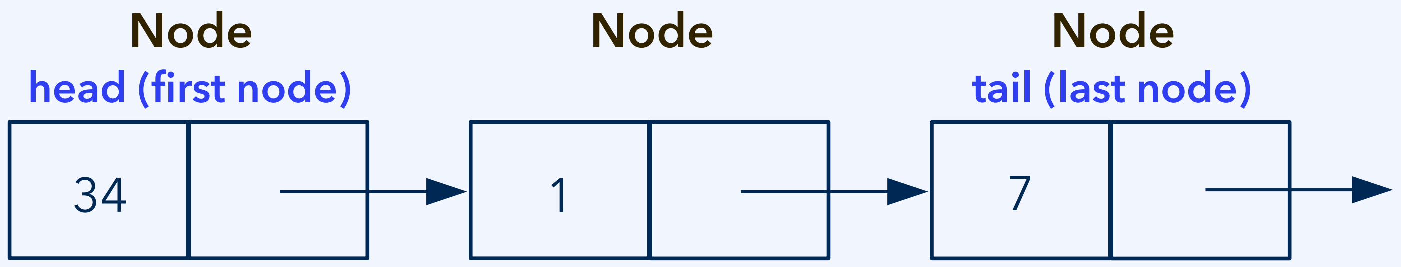

x[0]
x[1]
x[2]
x[3]
x[4]
x[5]
34
7
12
3
4
7
free
free
free
x.length
6
appending an element will take constant time, as long as there is capacity

Linked lists

Linked lists are **dynamic data structures**.

They differ from dynamic arrays in that their elements are **not contiguous in memory**. Therefore, pointer increments (*e.g.*, p++ as it would be in C/C++) cannot be used to proceed from one data item (element) to the next.

Instead, the linked list consists of **nodes**.

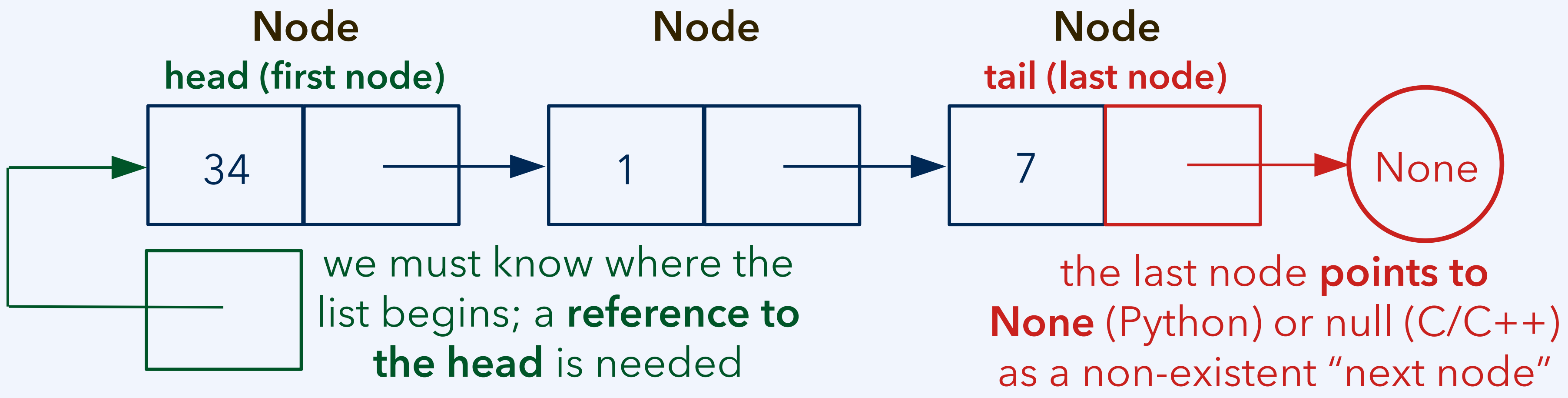


Linked lists

Linked lists are **dynamic data structures**.

They differ from dynamic arrays in that their elements are **not contiguous in memory**. Therefore, pointer increments (*e.g.*, p++ as it would be in C/C++) cannot be used to proceed from one data item (element) to the next.

Instead, the linked list consists of **nodes**.

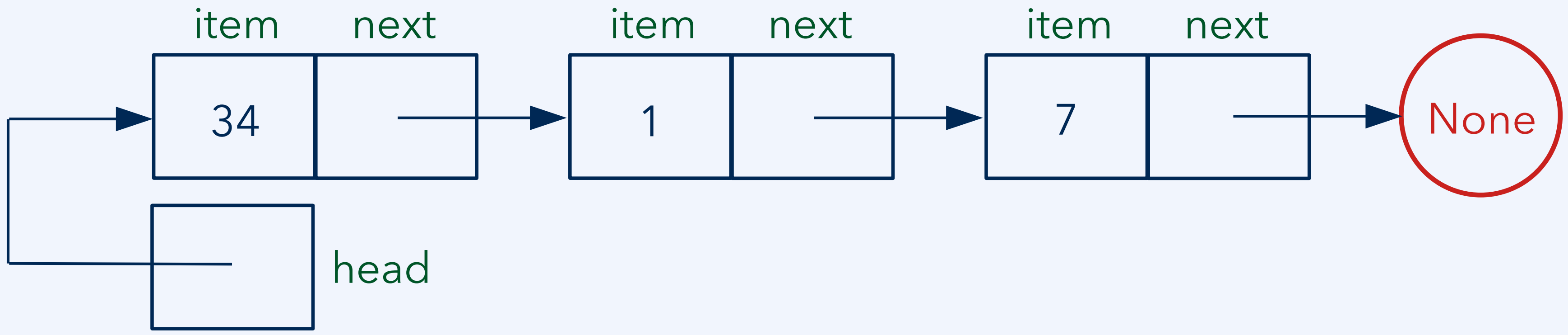

Node
head (first node)
Node
Node
tail (last node)
34
1
7
None
we must know where the list begins; a reference to the head is needed
the last node points to None (Python) or null (C/C++) as a non-existent "next node"

Linked lists

Linked lists are **dynamic data structures**.

They differ from dynamic arrays in that their elements are **not contiguous in memory**. Therefore, pointer increments (*e.g.*, p++ as it would be in C/C++) cannot be used to proceed from one data item (element) to the next.

Instead, the linked list consists of **nodes**, where each **item** is combined with a link (*i.e.*, a pointer in C/C++, an object reference in Python) to the **next node**. When this is all that is provided, the data structure is called a **singly linked list**.

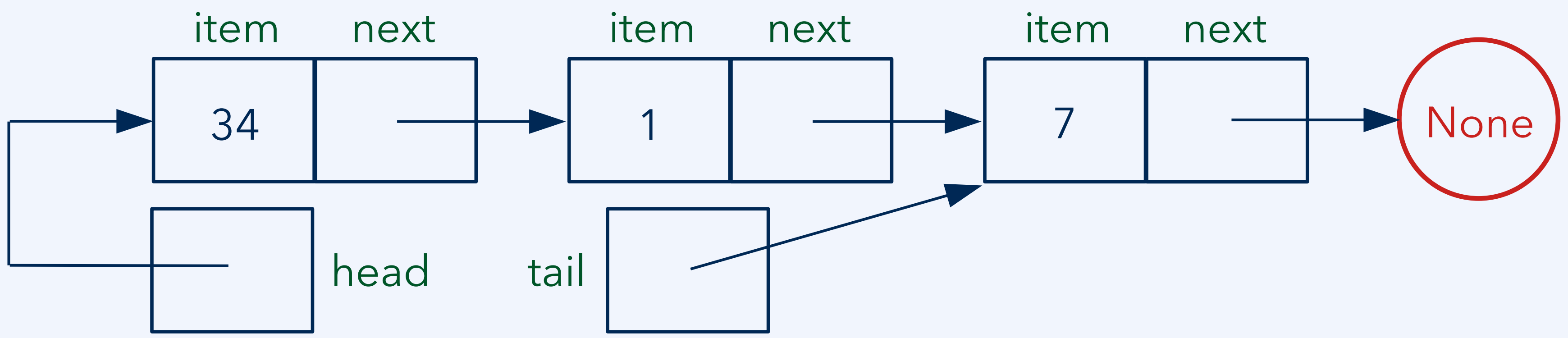


Linked lists

The list contains a reference to its **head**, and may also contain one to its **tail**.

Linked lists differ from dynamic arrays in that their elements are **not contiguous in memory**. Therefore, pointer increments (*e.g.*, p++ as it would be in C/C++) cannot be used to proceed from one data item (element) to the next.

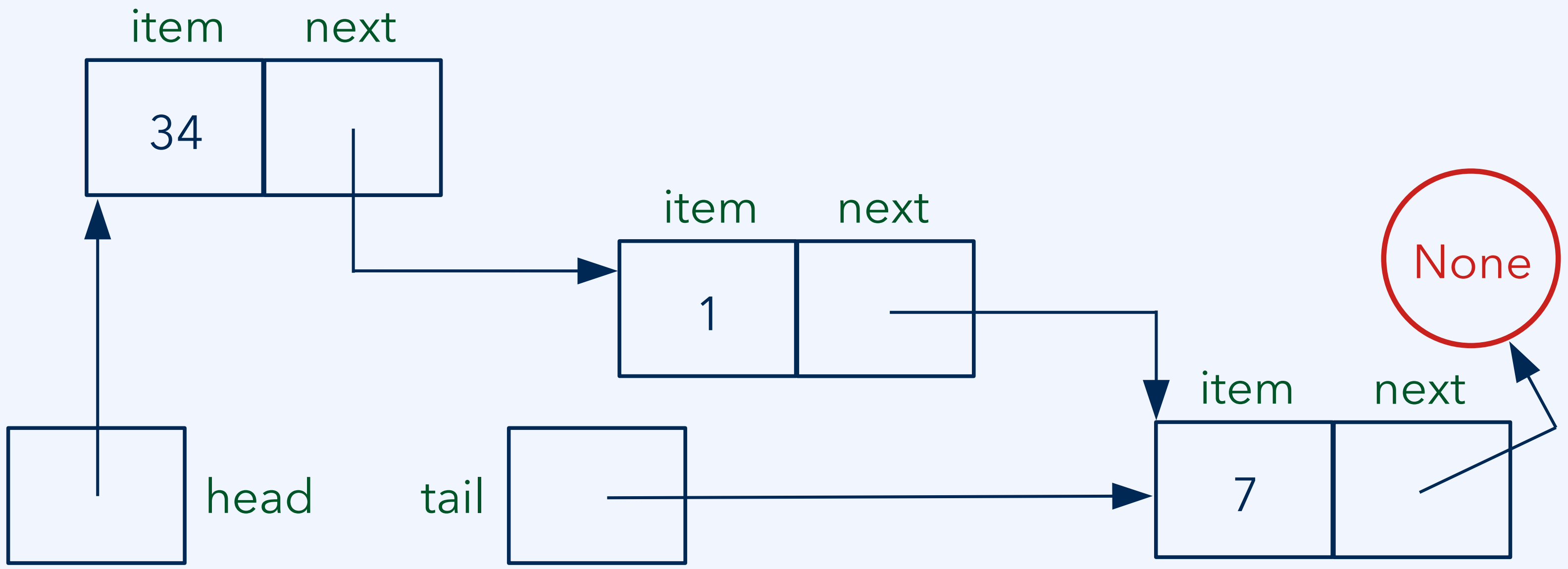
Instead, the linked list consists of **nodes**, where each **item** is combined with a link (*i.e.*, a pointer in C/C++, an object reference in Python) to the **next node**. When this is all that is provided, the data structure is called a **singly linked list**.



Singly linked lists: Inserting an element

If a **reference to a node** is given, another item can be **inserted after** that node in constant time; by accessing the linked-list object, it takes constant time to insert a new head at the beginning (**push**) or a new tail at the end (**append**).

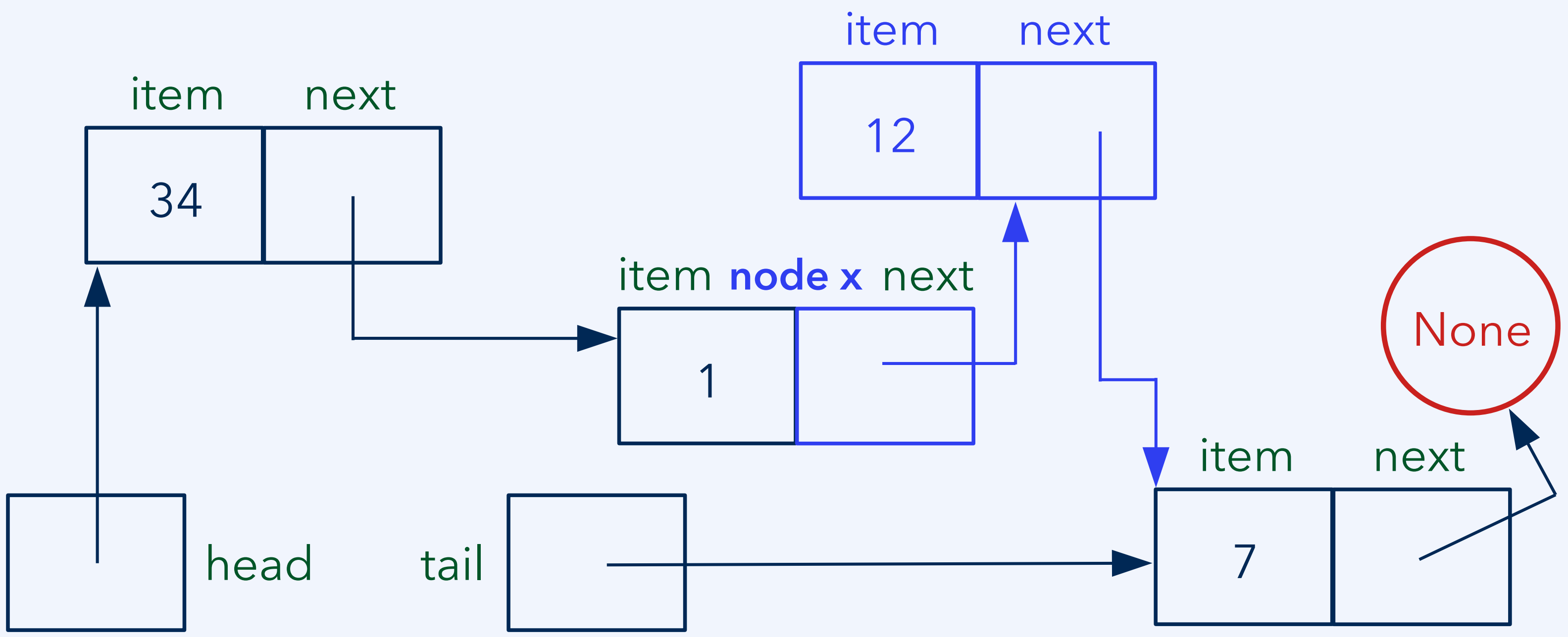
Walking *n* elements forward and doing an insertion there takes O(*n*) time.



Singly linked lists: Inserting an element

If a **reference to a node** is given, another item can be **inserted after** that node in constant time; by accessing the linked-list object, it takes constant time to insert a new head at the beginning (**push**) or a new tail at the end (**append**).

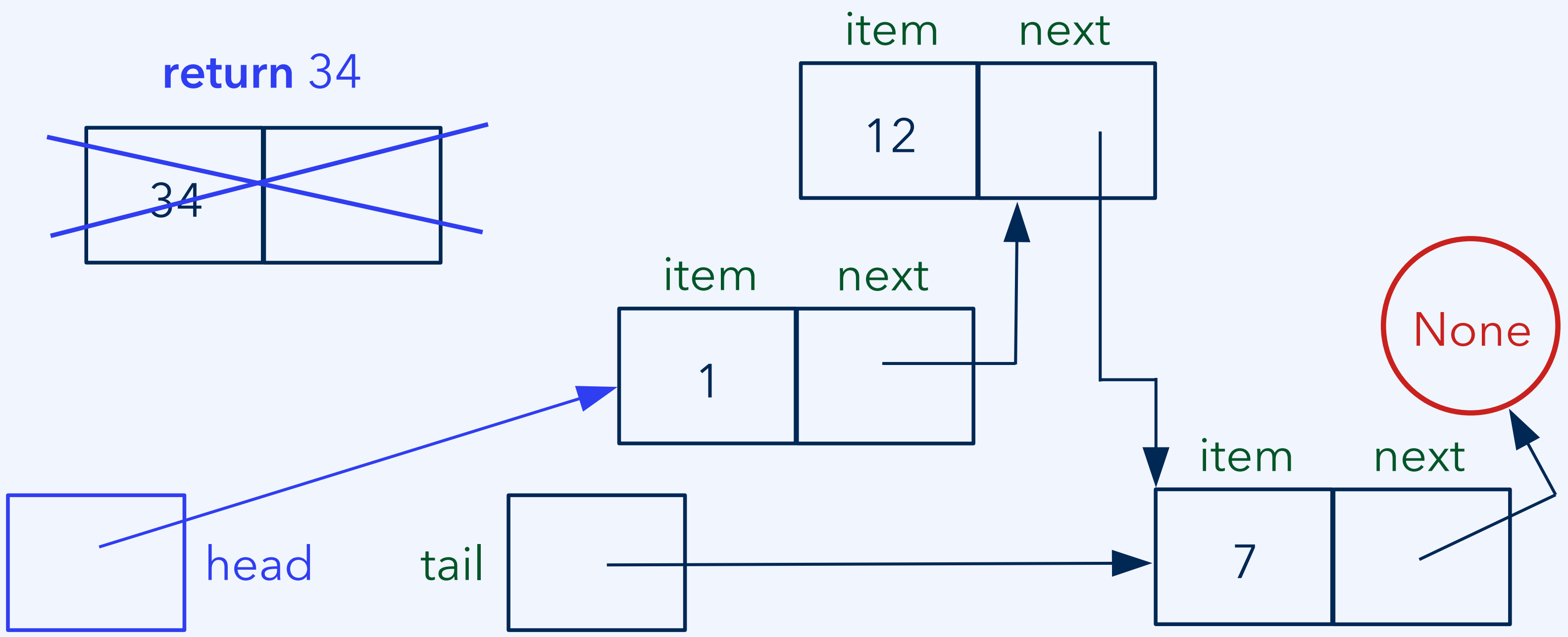
Example: Insert 12 after node x, to which we already have a reference.



Singly linked lists: Deleting an element

If a **reference to a node** is given, **deleting the subsequent node** from the list takes constant time; by accessing the linked-list object, it takes constant time to remove and return the **head item** (**pop** operation in a **stack** data structure).

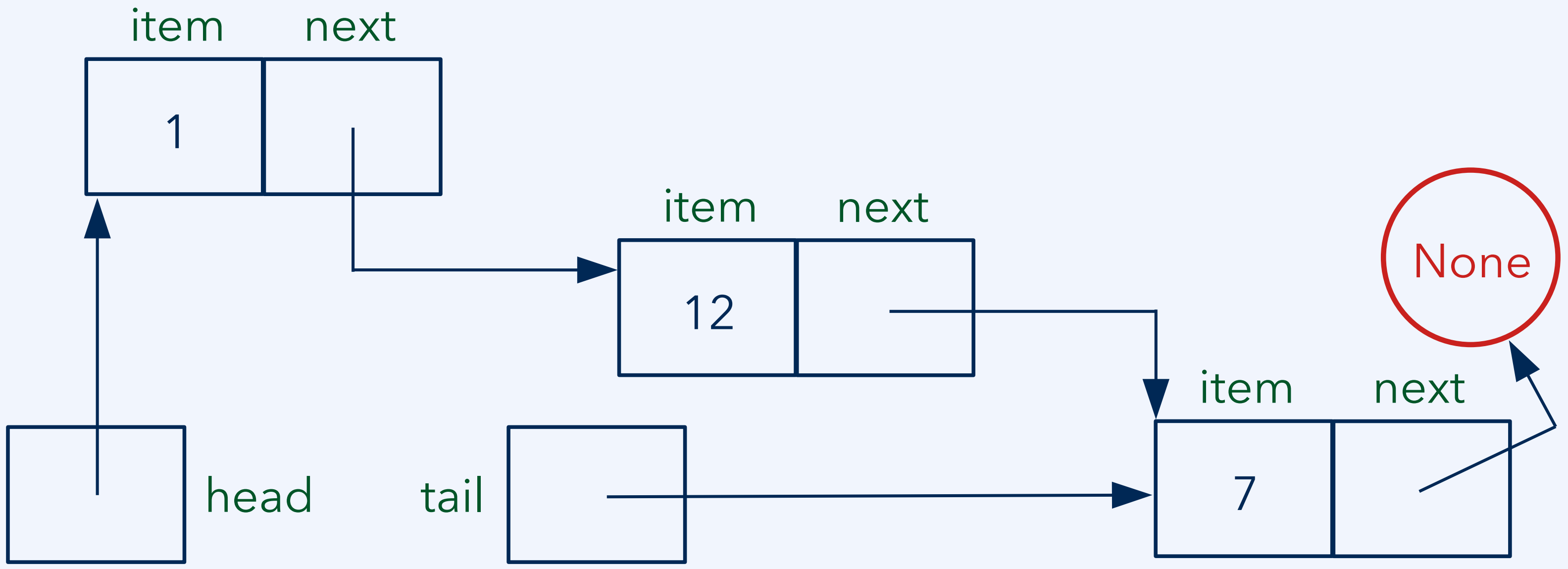
Walking *n* elements forward and doing an deletion there takes O(*n*) time.



Singly linked lists: Summary

In a **singly linked list**, nodes contain references (pointers) to the **next node**. This makes it possible to **iterate forward** in constant time, but *not backward*. **Insertion/deletion after** a given element (*not before* it!) takes **constant time**.

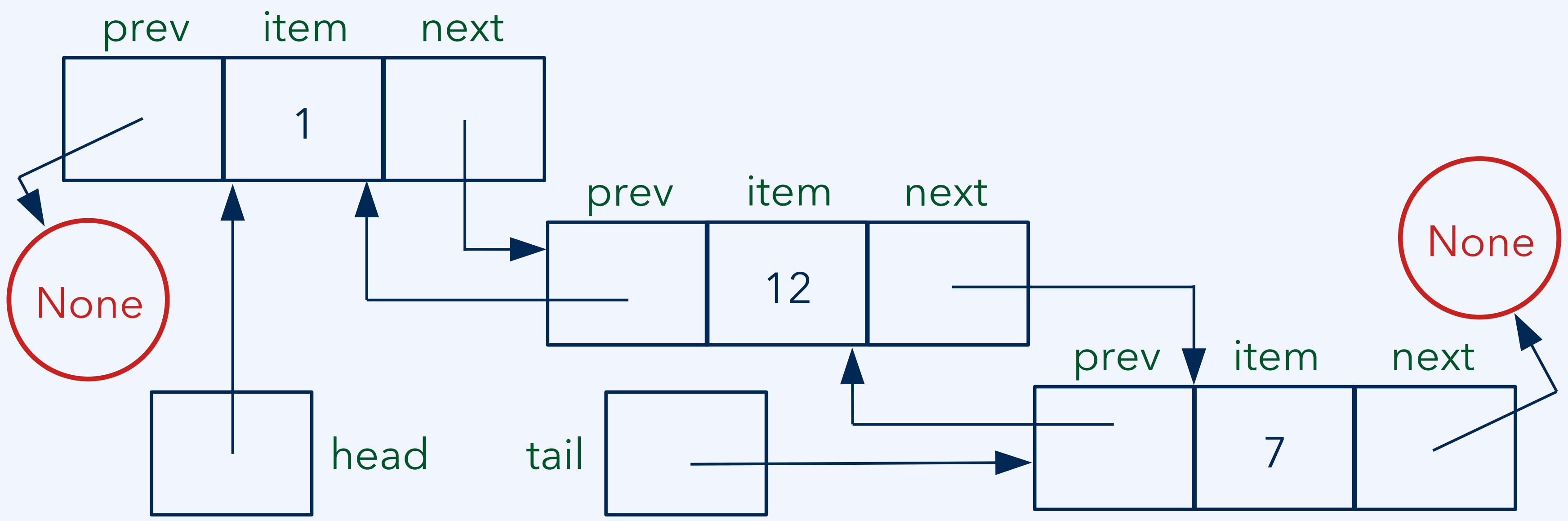
Recall that **for dynamic arrays**, general **insertion/deletion takes O(n) time**.



Doubly linked lists

In a **doubly linked list**, each node also contains a reference (or pointer) to the **previous node**. This facilitates traversal in **both directions and inserting** a new data item **before** any given node (rather than only after it), all in constant time.

Singly linked lists require two variables per data item (item and next).
Doubly linked lists require three variables per data item (prev, item, and next).



University of Central Lancashire UCLan

Summary: Efficiency analysis

- Read/write access to a data item at position *k*
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(*k*) time, *i.e.*, O(*n*) in the average/worst case

- Iterating over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists;
    in a *singly* linked list, this is limited to one direction, forward

- Deleting a data item at position *k*
  - For a dynamic array, O(1) at the end, O(*n* – *k*) in general
  - For a singly linked list, O(1) at the head, or if we have a reference to the element at position *k*–1; otherwise, in general, O(*k*)

Above, *n* is the length (logical size) of the dynamic array or linked list.

Remark: For **dynamic arrays**, *lookup* and jumping to an index are in O(1). It is only *rearranging the elements in memory* that, in general, can take linear time.

Summary: Efficiency analysis

- Read/write access to a data item at position k
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(k) time, *i.e.*, O(n) in the average/worst case
  - For a doubly linked list, O(min(k, n – k)), which is still effectively O(n)

- Iterating over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists

- Deleting a data item at position k
  - For a dynamic array, O(1) at the end, O(n – k) in general
  - For a singly linked list, O(1) at the head, or if we have a reference to the element at position k–1; otherwise, in general, O(k)
  - For a doubly linked list, O(1) at the head or tail, or if we have a reference to that region of the list; in general, O(min(k, n – k))

Remark: For **linked lists**, *insertion/deletion as such* takes constant time, once the node has been localized. However, *getting to the node* can take O(n) time.

Summary: Efficiency analysis

- Read/write access to a data item at position k
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(k) time, *i.e.*, O(n) in the average/worst case
  - For a doubly linked list, O(min(k, n – k)), which is still effectively O(n)

- Iterating over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists

- Inserting an additional data item at position k
  - For a dynamic array, O(n) in the worst case, *i.e.*, whenever the capacity is exhausted; with free capacity, O(1) at the end, O(n – k) elsewhere
  - For a singly linked list, O(1) at the head or tail, or if we have a reference to the element at position k–1; Otherwise, in general, O(k)
  - For a doubly linked list, O(1) at the head or tail, or if we have a reference to that region of the list; in general, O(min(k, n – k))

Application: Stacks and queues

- **Stacks** function by the principle **"last in, first out" (LIFO)**
  - Can be implemented using a singly linked list:
    - » Attach (push) new elements at the head of the list only
    - » Detach (pop) elements from the head of the list only

- **Queues** function by the principle **"first in, first out" (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - » Attach (push) new elements at the tail of the list only
    - » Detach (pop) elements from the head of the list only

All these operations can be carried out in constant time.

Application: Stacks and queues

- **Stacks** function by the principle **"last in, first out" (LIFO)**
  - Can be implemented using a singly linked list:
    - » Attach (push) new elements at the head of the list only
    - » Detach (pop) elements from the head of the list only
  - Can be implemented using a dynamic array:
    - » Attach (push) new elements at the end of the array only
    - » Detach (pop) elements from the end of the array only
- **Queues** function by the principle **"first in, first out" (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - » Attach (push) new elements at the tail of the list only
    - » Detach (pop) elements from the head of the list only

All these operations can be carried out in constant time;
in case of the push operation for the dynamic array, subject to free capacity.

University of Central Lancashire
UCLan

Python implementation

See “linked_list” Jupyter Notebook example

University of Central Lancashire UCLan

Implementing data structures as Python classes

For implementing data structures (one of the main learning outcomes from this module), it is usually helpful to rely on object-oriented programming.

In comparison to OOP syntax from other programming languages, Python syntax is analogous in many ways. Major differences to C++ include:

- Object variables are object references, behaving similar to pointers.
- The argument “self” (analogous to “this” in C++) needs to be mentioned as the first argument in each method definition.
- There are no private properties and methods – all is public. You are advised to begin names with an underscore if they are for internal use.

See https://docs.python.org/3/tutorial/classes.html

Performance of linked lists vs. dynamic arrays

Python lists (dynamic arrays) are very efficient for many purposes.

However, deletions and insertions anywhere except at the very end are handled inefficiently, compared to linked lists. In the linked_list Jupyter Notebook, this is demonstrated by the following example task:

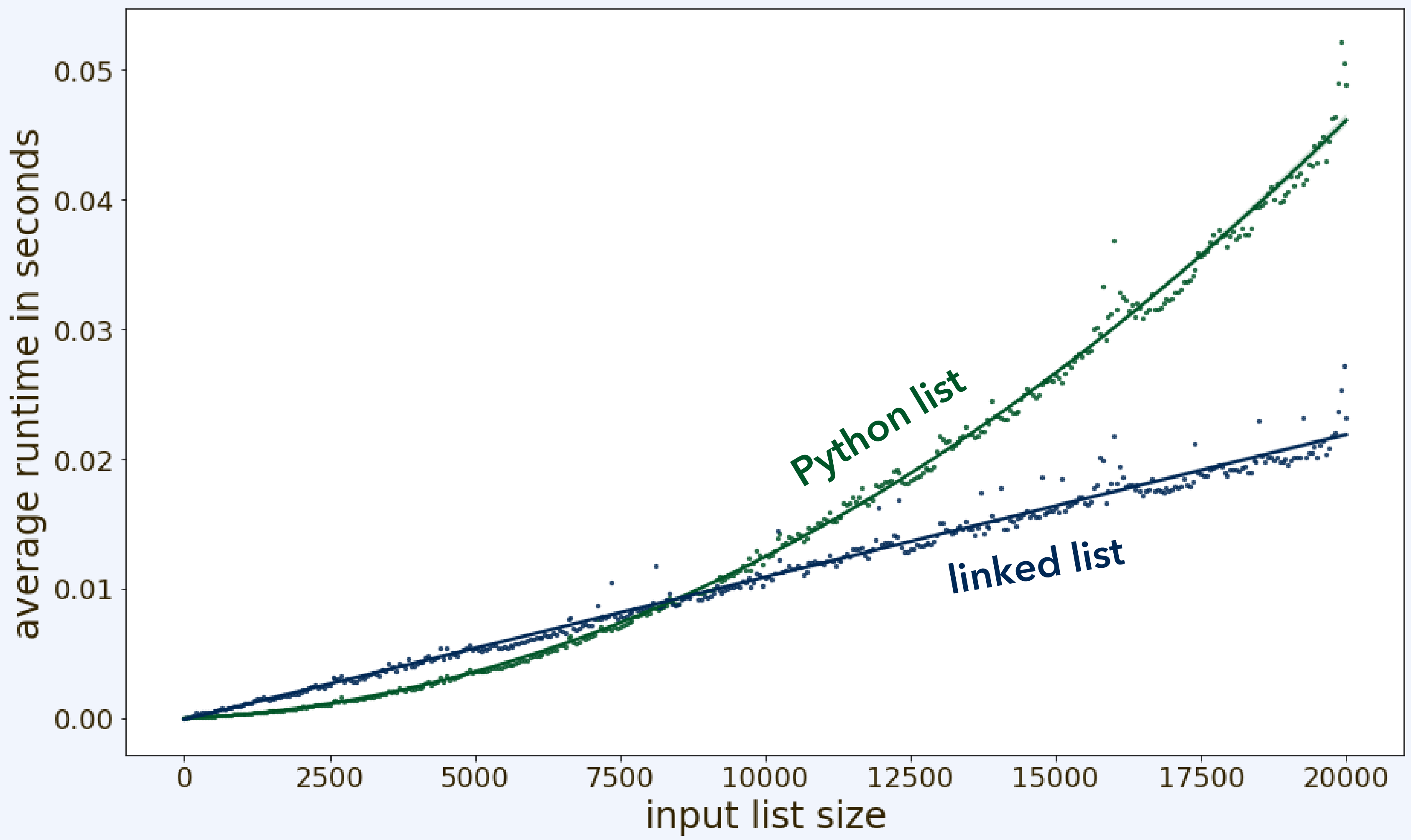
A list with *n* elements is given. Iterate over the whole list, and for each element:

- If it is a multiple of 3, delete it from the list;
- If it has a remainder of 1 upon division by three, do nothing;
- If it has a remainder of 2, insert a copy of the element right next to it.

In this way, *e.g.*, [19, 12, 20, 12, 4] is modified to become [19, 20, 20, 4].

See "linked_list" Jupyter Notebook example

Performance of linked lists vs. dynamic arrays



University of Central Lancashire UCLan

Glossary

CO2412

16th November 2021

Glossary

From your point of view:

- What are the most **important new concepts** that were discussed?
- What essential terms have we been using **without a clear definition**?
- What concepts have we been using with **different meanings in different contexts**, so that a clarification would be helpful?
- Has there been any **unexplained jargon** or technical terminology?

Any such expressions would be suitable for the glossary.

Suggestions are welcome at any time, particularly now, right after the lecture.

University of
Central Lancashire
UCLan

CO2412
Computational Thinking

Tutorial 1.2 problem
List-like data structures
Python implementation

Where opportunity creates success