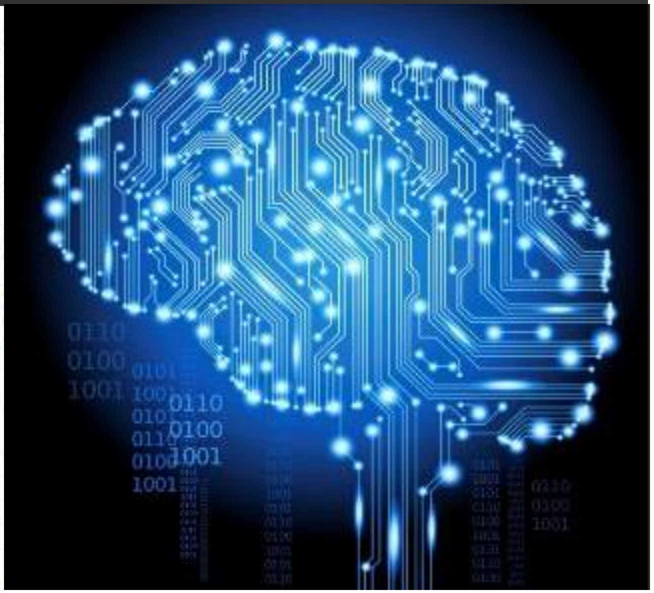
Information Technology

FIT1008&2085 Lecture 16
Lists with Arrays

Prepared by: M. Garcia de la Banda



Where we were at?

- **Last lecture we looked at Exceptions and Assertions**
- **We also had learnt several concepts including**
  - Data type, data structures, abstract data types (ADTs)
- **We had started to implement**
  - A list ADT
- **We had defined a few operations for them**
  - Create, access an element, compute the length
  - Check whether the list is empty
  - Check if an item is in the list using linear search
- **We had kept thinking about complexity**

Objectives for these two lectures

- **To finish implementing the list ADT**
  - More on linear search
  - Binary search
  - Deleting elements
  - Adding elements
- **Determine whether these operations suit sorted list**

- **In the process:**
  - To look "under the hood" at the array implementation
  - Keep practicing and becoming confortable:
    - Developing simple algorithms in Python
    - Computing their Big O time complexity

Time Complexity for sorted Linear Search

```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element:
            return True
        elif item < element:
            return False
    return False
```

? times

Access is constant K1

Comparison we don't know m1

Return is constant K2

Comparison we don't know m2

Return is constant K3

Return is constant K4

Best ≠ Worst

Some elements get a certain amount of processing

Other elements are not processed at all

Time complexity for sorted Linear Search

- **Best case?**
  - Loop stops in the first iteration
  - When? The wanted item is at the start of the list
    - K1 + m1 + K2 → O(m1)
- **Worst case?**
  - Loop goes all the way (n times, if n is the length of the list)
  - When? The wanted item is not found
    - (K1+m1+m2)*n + K4 → O((m1+m2)*n)
    - m1 and m2 are often the same (or max of the two) → O(m*n)

? times

```
def lin_search(sorted_list, item):
    for element in sorted_list:          Access is constant K1
        if item == element:              Comparison we don't know m1
            return True                  Return is constant K2
        elif item < element:             Comparison we don't know m2
            return False                 Return is constant K3
    return False                         Return is constant K4
```

An alternative (better/worse?) algorithm

```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element: #cannot be in
            return False
    return False #not found
```

- We modify the above algorithm to (differences in red):

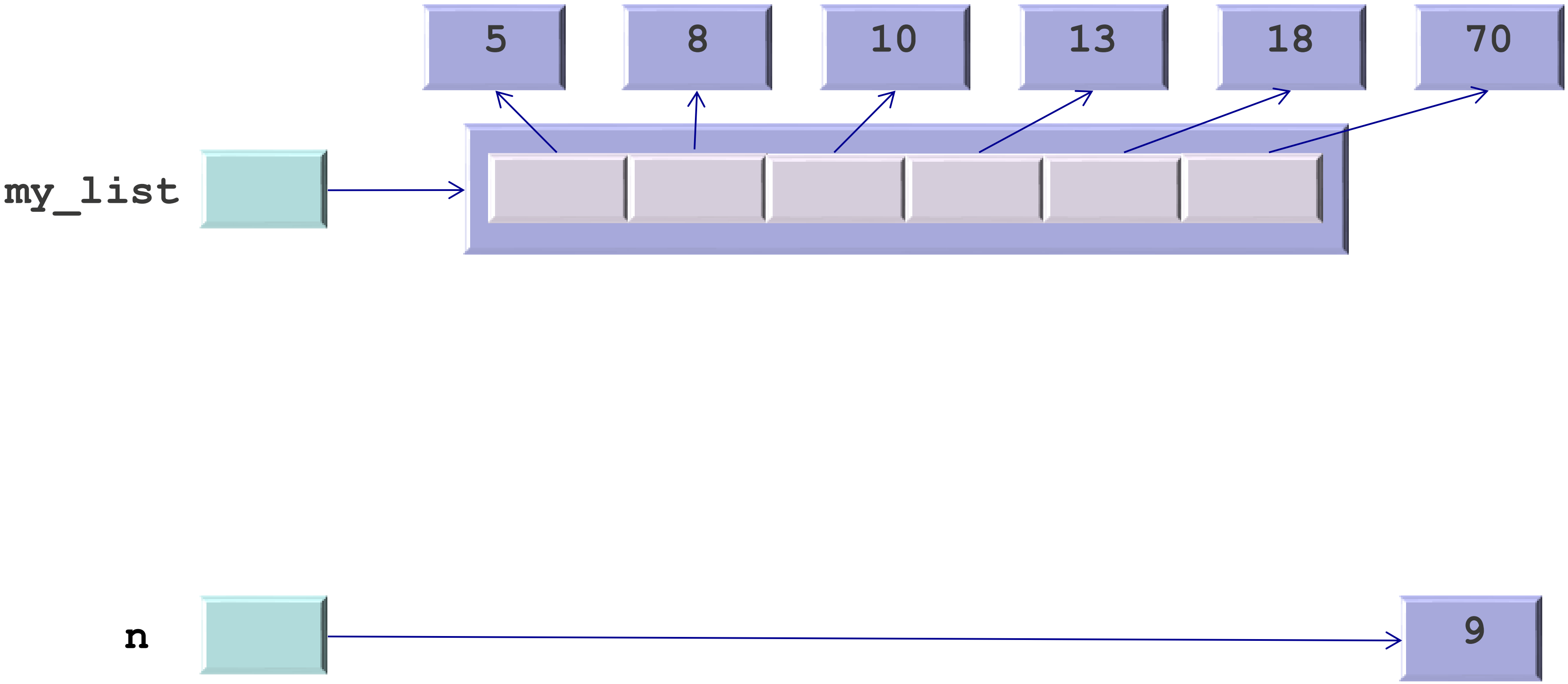
```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: #keep on going
            continue
        else: # found or know it cannot be in
            return(item == element)
    return False #not found
```

```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

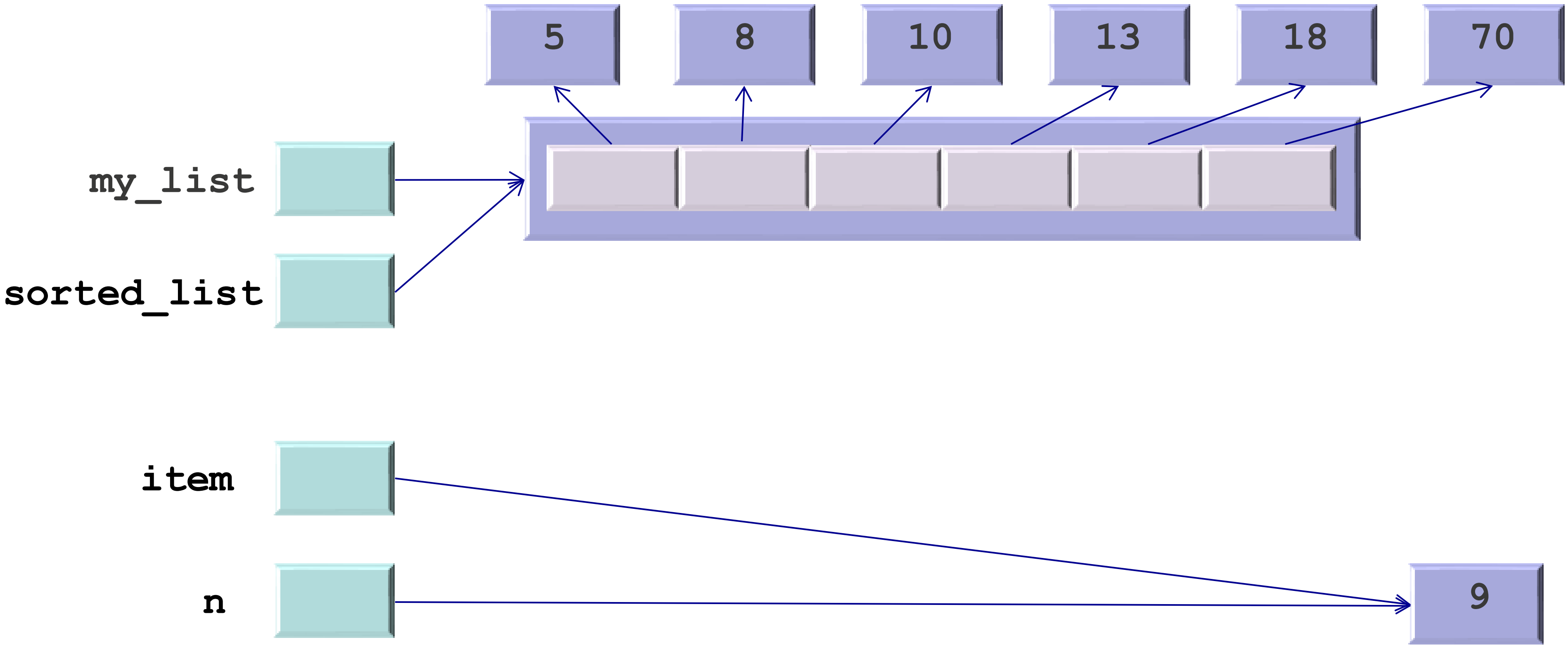



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

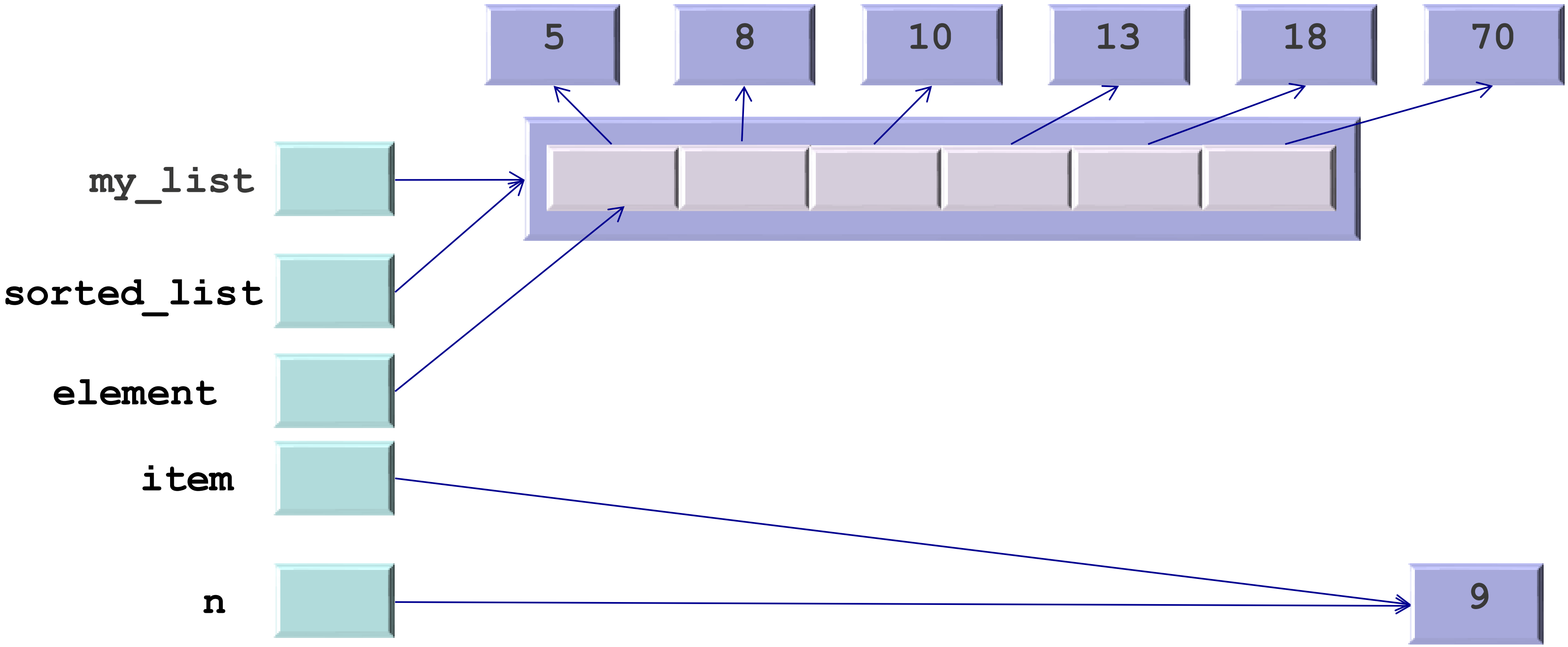



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

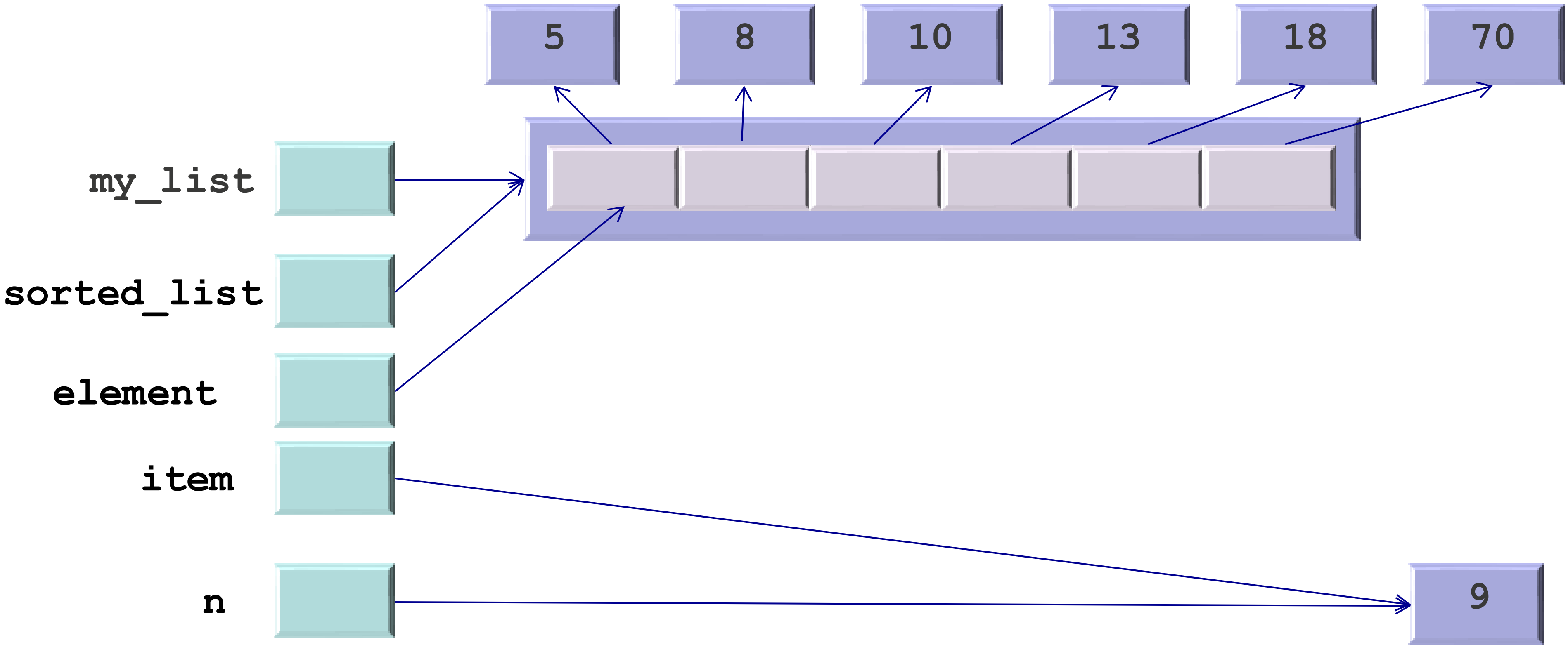



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

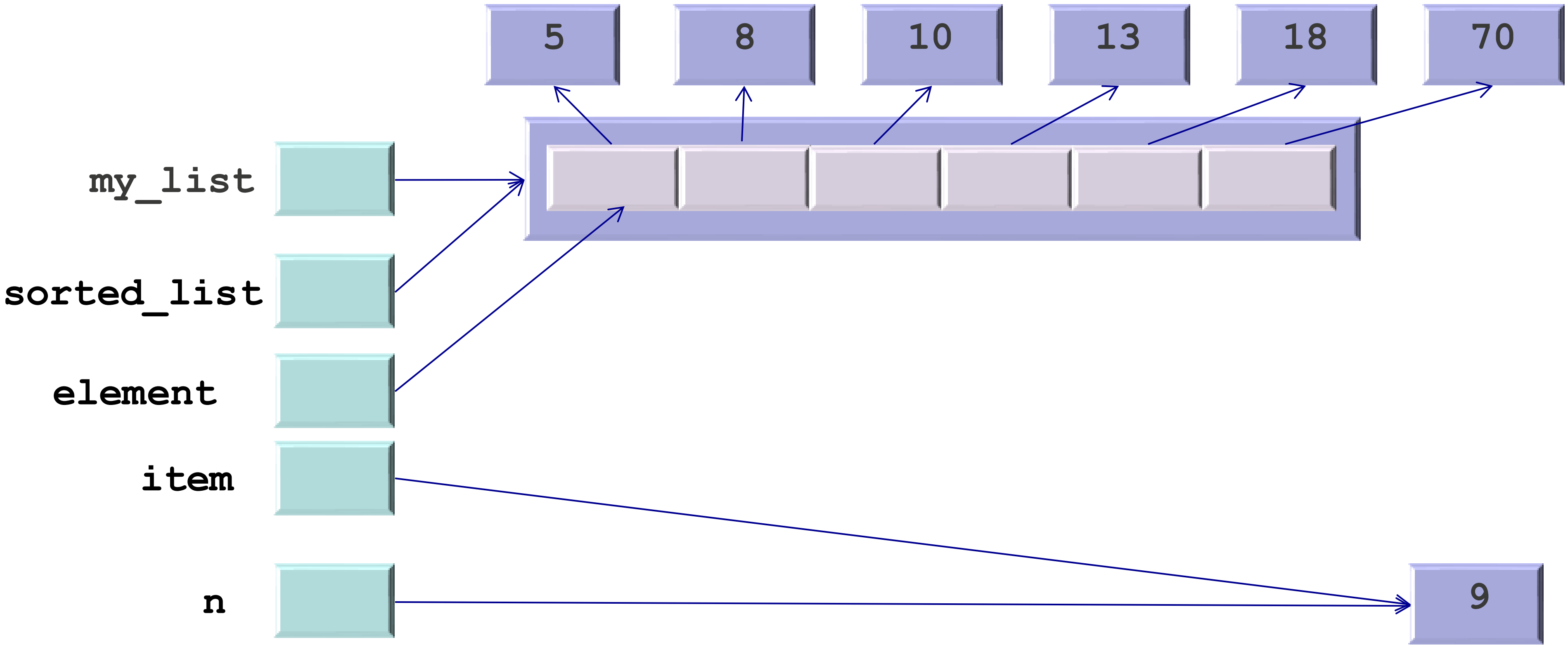



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

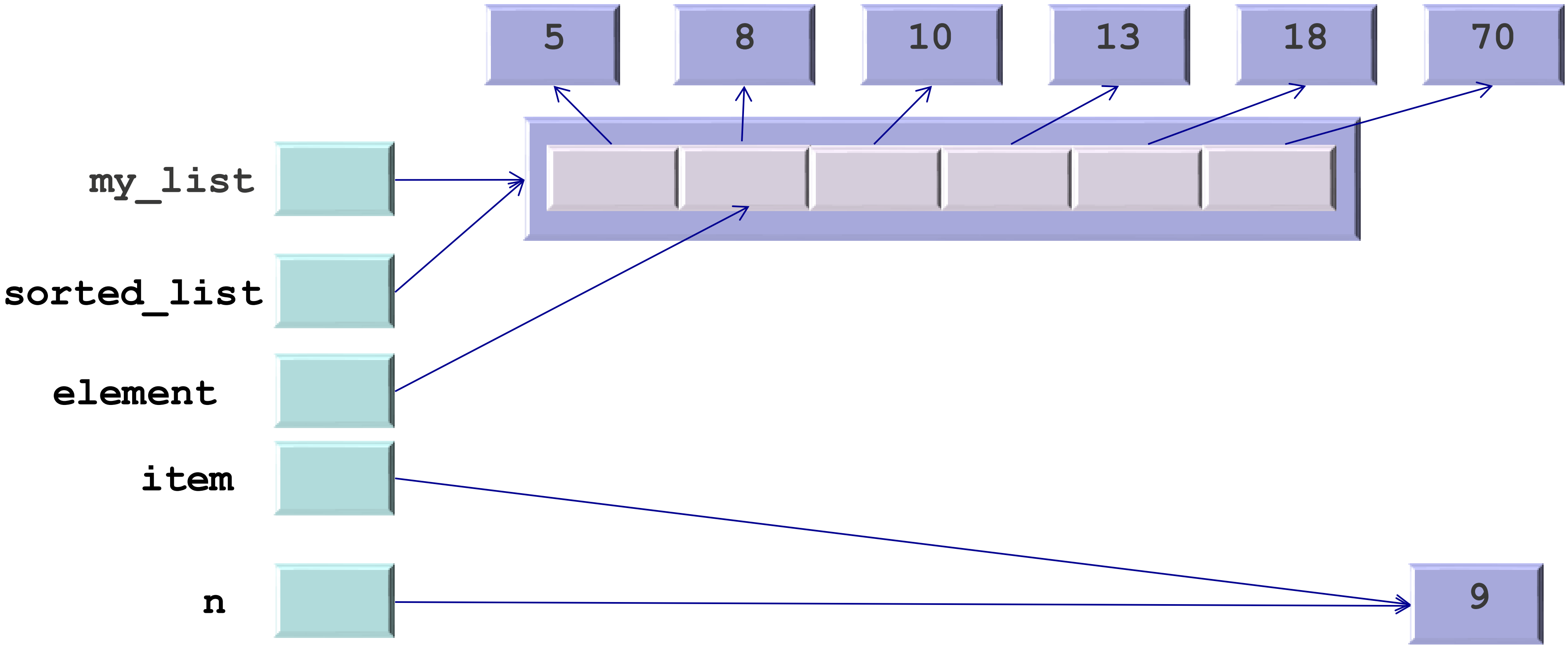



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

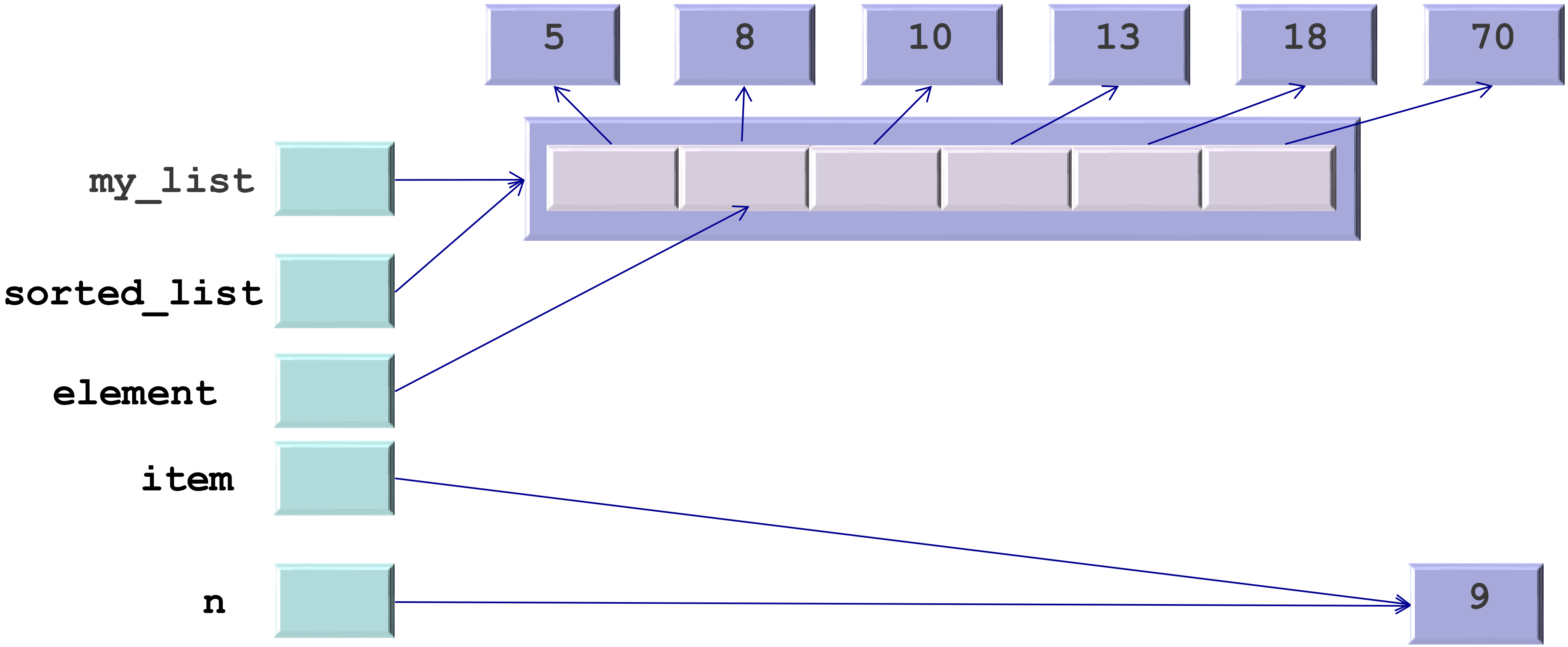



```python
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```python
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

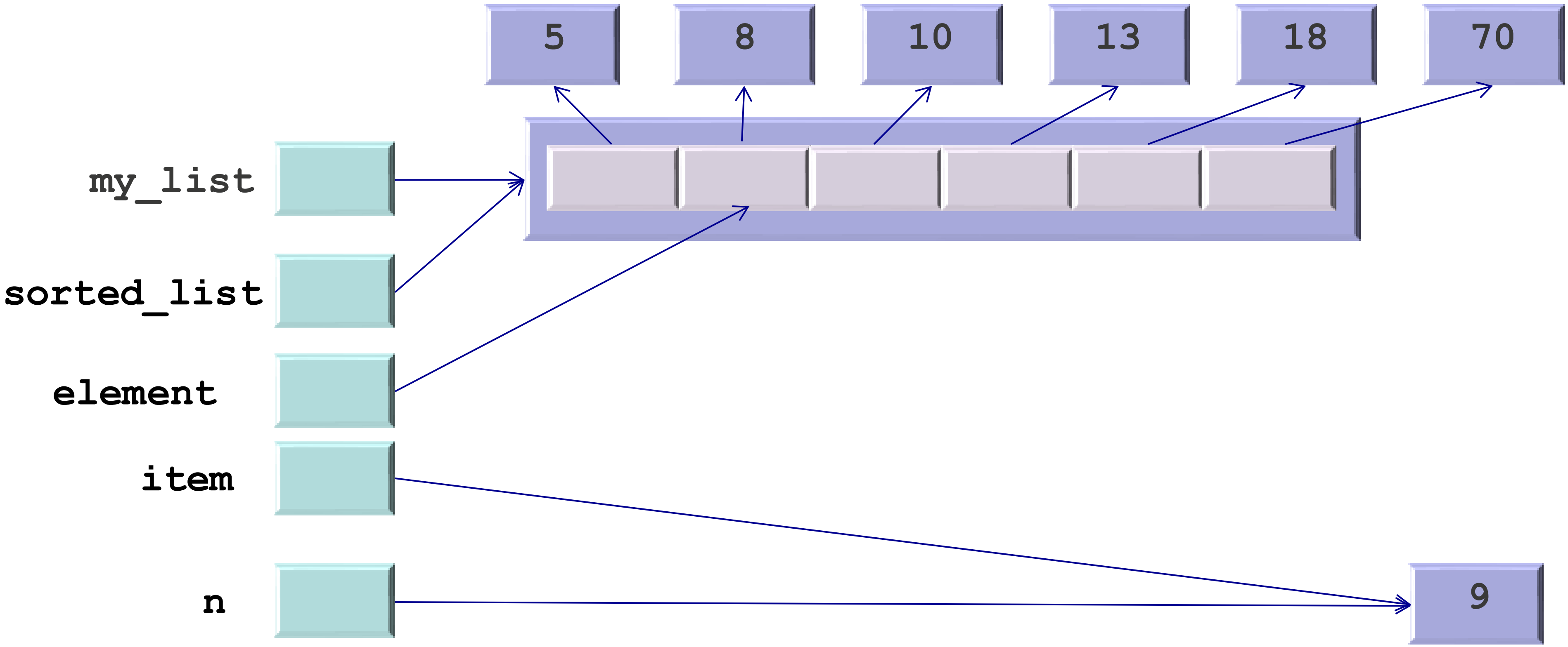



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

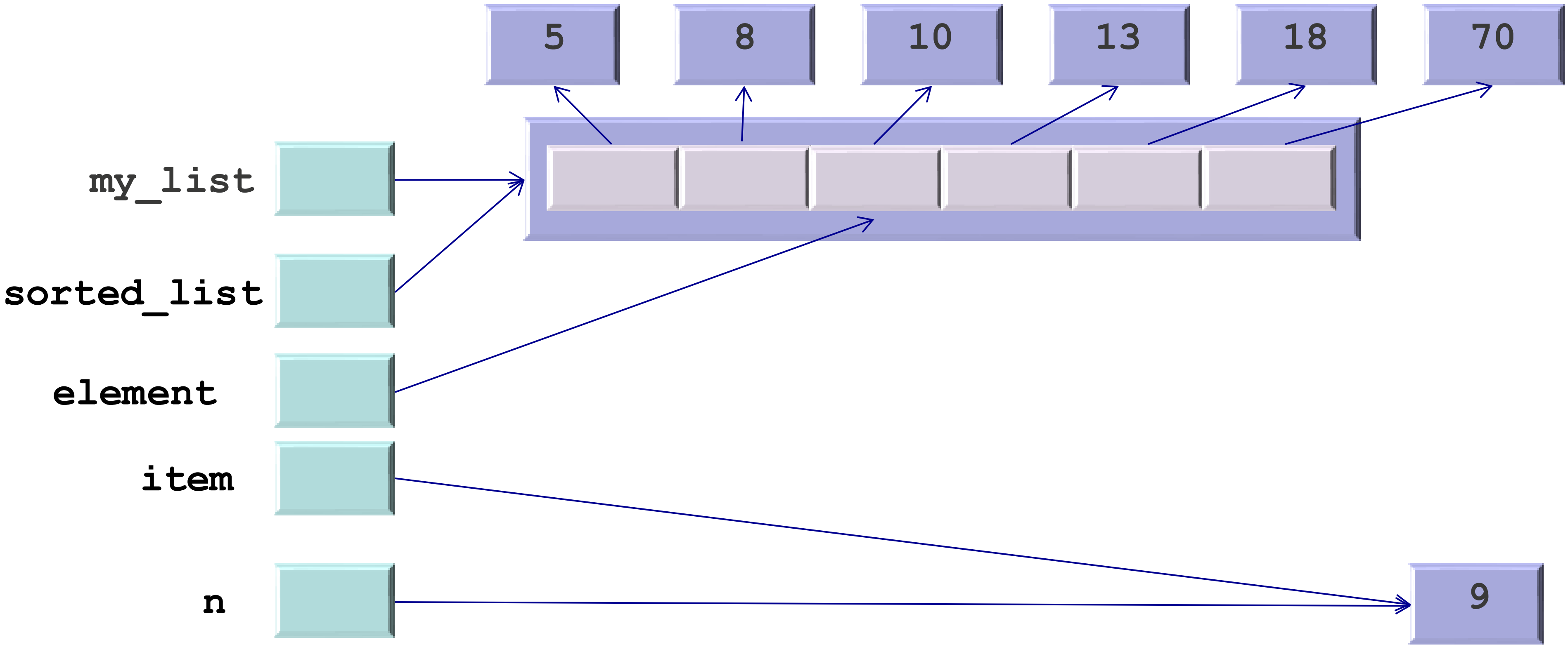



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

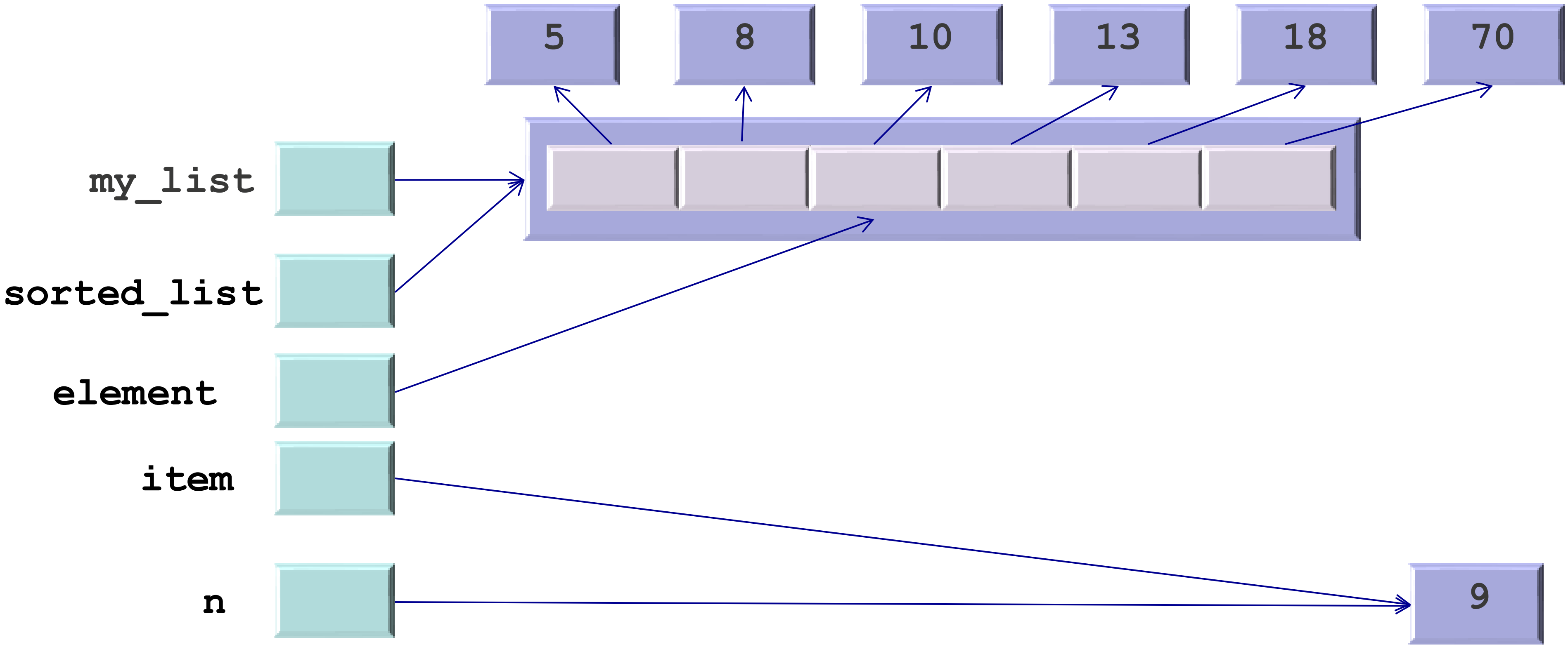



```python
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```python
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

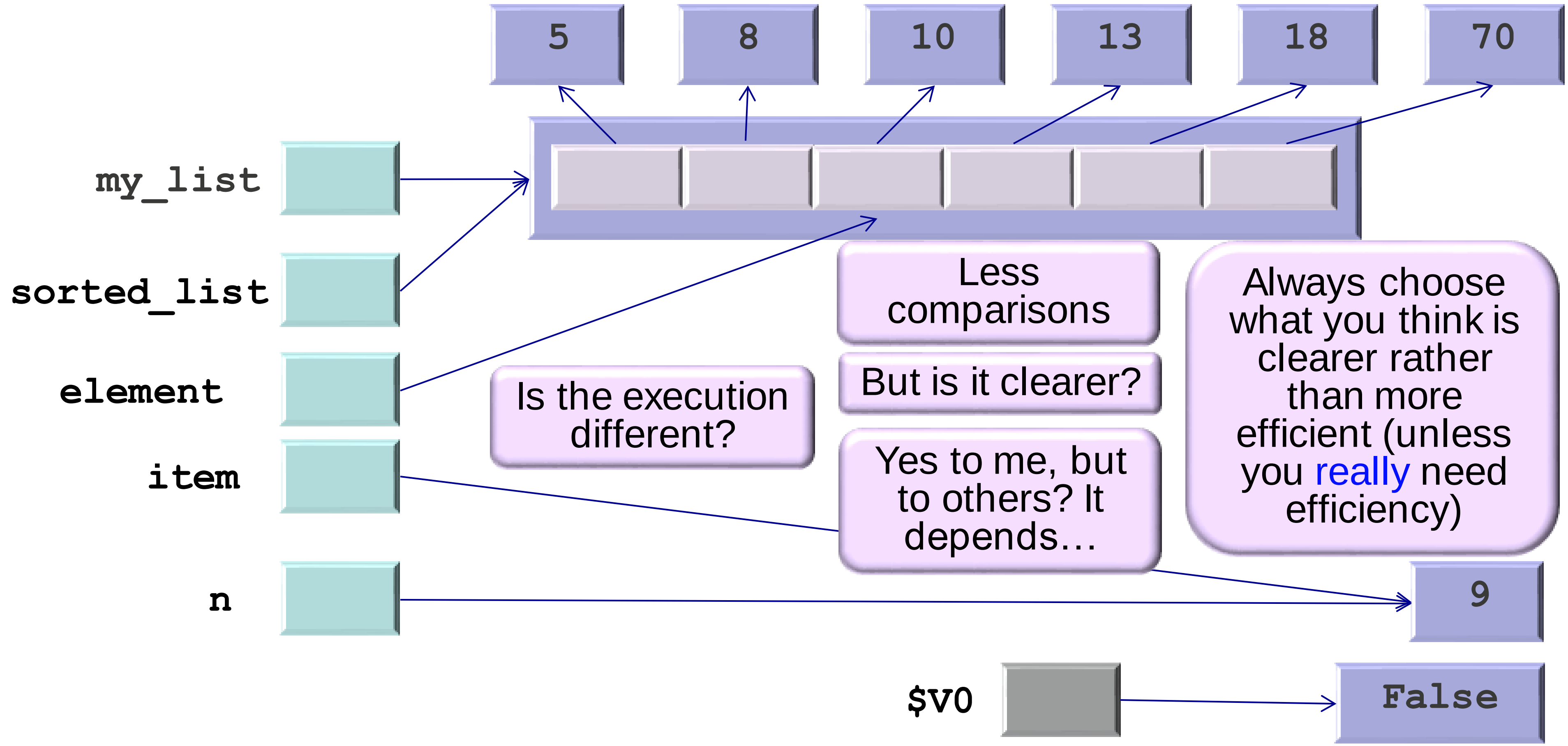



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller




```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: # keep
            continue
        else:# found or cannot be in
            return(item == element)
    return False #not found
```

```
my_list = [5,8,10,13,18,70]
n = 9
lin_search(my_list, n)
```

Callee

Caller

Making it as clear as possible

- **It is tempting to write our last linear search algorithm:**

```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: #keep going
            continue
        else:# found or know it cannot be
            return(item == element)
    return False #not found
```

- **As:**

```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item > element: #keep going
            continue
        elif item == element: #found
            return True
        else: #know it cannot be in
            return False
    return False #not found
```

Resist the temptation!
Always use `return A`
rather than:

```
    if A:
        return True
    else:
        return False
```

and use `return not A`
rather than:

```
    if A:
        return False
    else:
        return True
```

Modify `lin_search` to find the position

- If the item is found, return its position in the list
- If not, return `None` (indicates it didn't find it)

```
def lin_search(the_list, item):
    for index in range(len(the_list)):
        if item == the_list[index]:
            return True
    return False
```

Index version

```
def lin_search_index(the_list, item):
    for index in range(len(the_list)):
        if item == the_list[index]:
            return index
    return None
```

- Btw, **`None`** is a constant; the only value of type **`NoneType`**.
- You could also raise an exception
- Could also use -1 or any value to mean Not there!

Binary Search

- **We can use it if the list is**
  - Sorted (for our algorithm, in ascending order)
  - Implemented with an array (we will see why later)
- **The algorithm is simple:**

If (value == middle element)
value is found
else if (value < middle element)
search left-half of list with the same method
else
search right-half of list with the same method

Binary Search Case 1: val == a[mid]

val = 10

low = 0, high = 8

mid = (0 + 8) // 2 = 4

a:

1	5	7	9	10	13	17	19	27
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
low				mid				high

Return `True`

Binary Search Case 2: val > a[mid]

val = 19

low = 0, high = 8

mid = (0 + 8) // 2 = 4

new low = mid + 1 = 5

a:	1	5	7	9	10	13	17	19	27
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
	low				mid	new low			high

Keep on searching using the same algorithm

Binary Search Case 3: val < a[mid]

val = 7

low = 0, high = 8

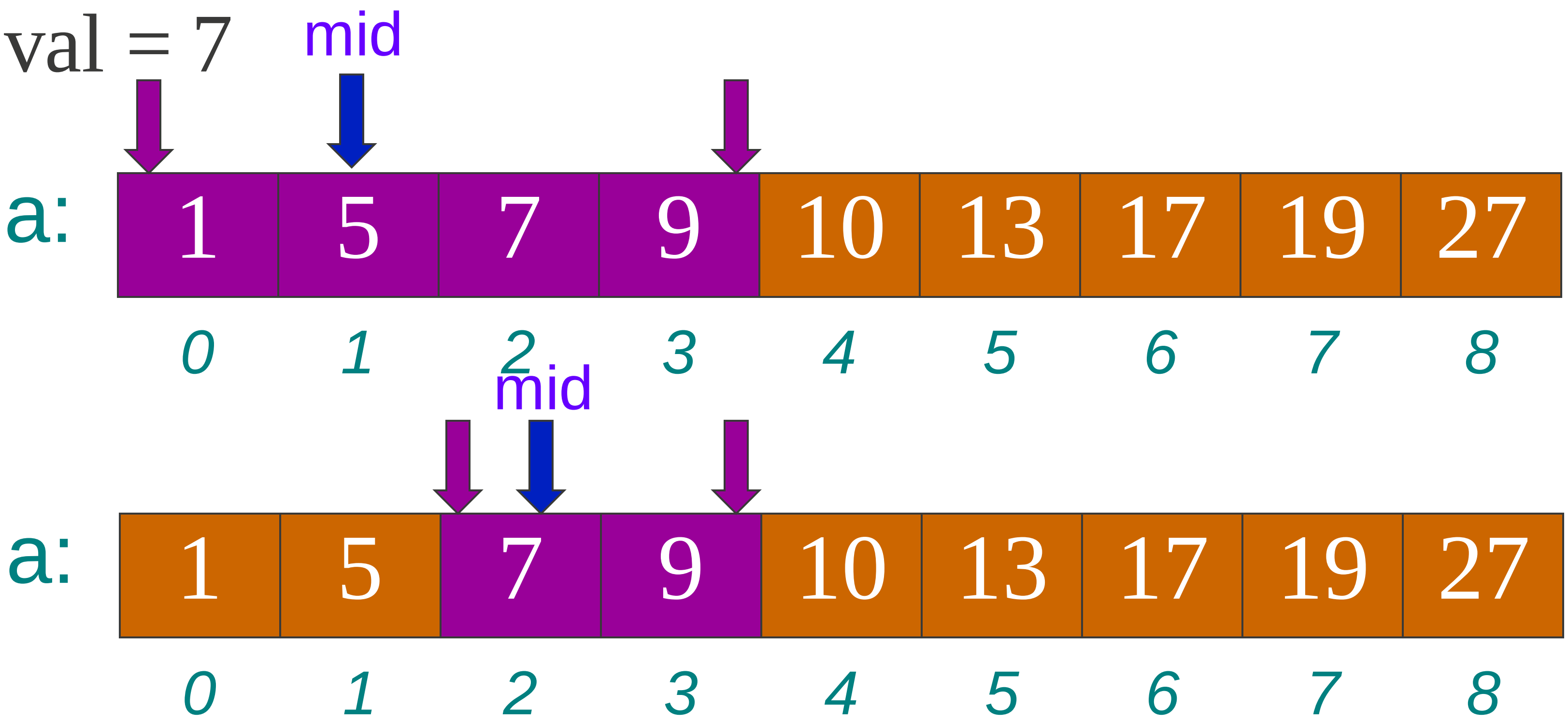
mid = (0 + 8) // 2 = 4

new high = mid -1 = 3

a:									
	1	5	7	9	10	13	17	19	27
	0	1	2	3	4	5	6	7	8
	low			new high	mid				high

Keep on searching using the same algorithm

Binary Search Case 3: val < a[mid] (cont)



Return `True`

Binary Search Case 4: val not in

val = 11

Then `low = 5`, `high = 4`

Return `False`

Implementing Binary Search in Python

```
def binary_search(sorted_list,item):
    low = 0
    high = len(sorted_list)-1
    while low <= high:
        mid = (low+high)//2
        if sorted_list[mid] > item:
            high = mid-1
        elif sorted_list[mid] == item:
            return True
        else:
            low = mid+1
    return False
```

?

1	5	7	9	10	13	17	19	27
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
low				mid				high

Complexity?

Every operation here is either O(1) except comparisons, which are O(m) , where m is again the size of the element being compared

Best ≠ Worst

Some elements get a certain amount of processing; others none

Time Complexity for Binary Search

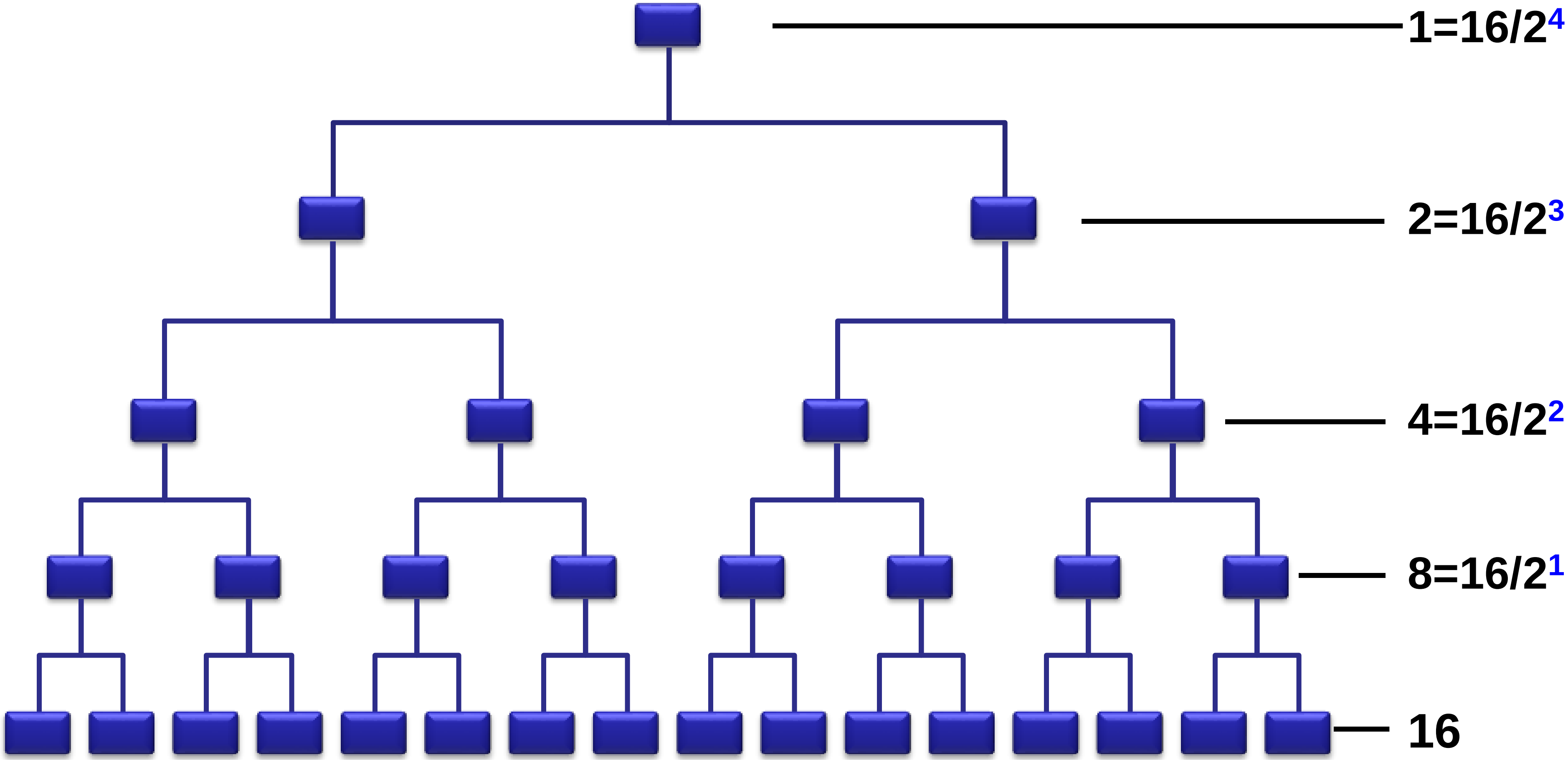
- **Size of the array being searched: n (many!)**
- **Best case?**
  - Loop stops immediately. When?
    - Item in the middle
  - m + some constants → O(m)
- **Worst case?**
  - Loop goes all the way: When?
    - Item not found
  - Each iteration (without finding) does two comparisons (O(m)) plus a fixed number of other operations
  - But how many times? $\log_2 n$
  - So, O(m*log n)

Calculating the Worst Case Complexity

- After 1 bisection $n/2$ items
- After 2 bisections $n/4 = n/2^2$ items
- After 3 bisections $n/8 = n/2^3$ items
- . . .
- After *b* bisections $n/2^b = 1$ item

$$b = \log_2 n$$

Another way of looking at it



Binary Search: why sorted and array?

- **We said we can use Binary Search if the list is**
  - Sorted (for our algorithm, in ascending order)
  - Implemented with an array
- **Why sorted?**
  - Otherwise we cannot guarantee that the item we are looking for is NOT in the half we discard
- **Why implemented using an array?**
  - We need to access any element in the list
  - We need to do that efficiently:
    - We need constant time access
    - Arrays ensure that is always the case (as seen in MIPS)

Adding and deleting elements

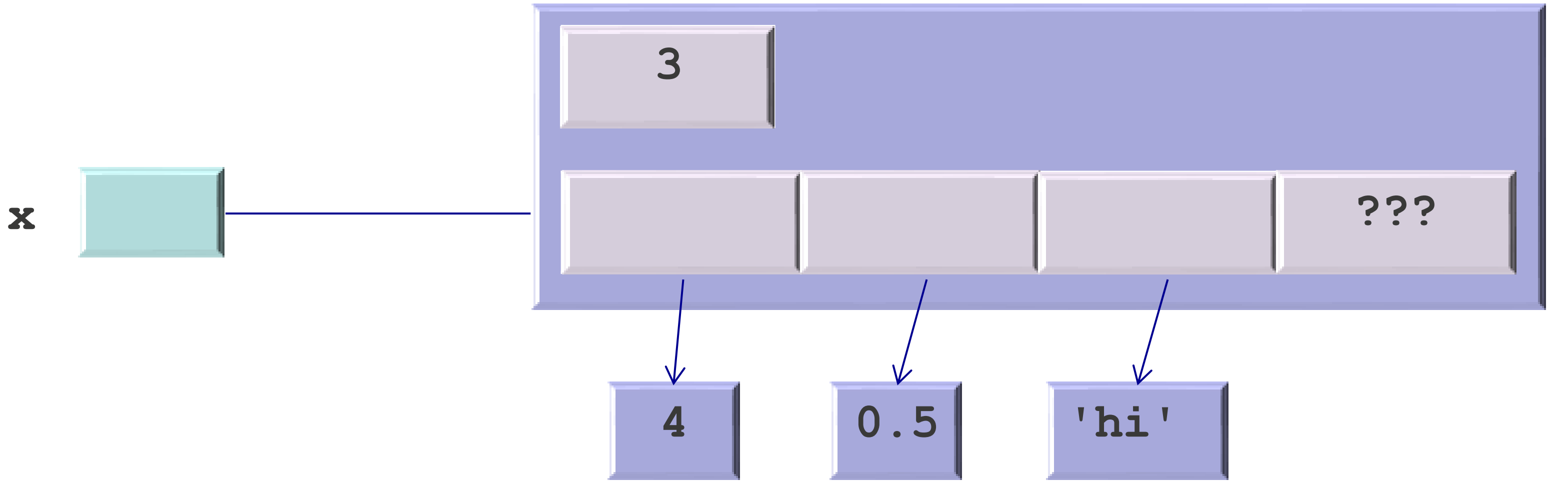
- **Up to now we have only:**
  - Traversed lists
  - Swapped elements
  - Compared elements
- **We now want to add and delete elements**
  - This means changing the size of the list
- **How do we do this using only create, access, length?**
  - We cannot… without copying all elements into a bigger list
- **If we use other python list operations (`del`,`append`,etc)**
  - Miss: that is exactly what we are trying to do ourselves!
- **We will instead "mimic" the use of arrays**
  - Not a waste of time! (you will need it for other languages)

Looking under the hood

- **Many implementation of lists use arrays**
- **As we said: arrays have fixed size (never changes)**
  - Needs to be known when they are created
  - It is always known (kept with the array)
- **But the number of elements in lists might change!**
- **So, lists implemented with arrays need 2 things:**
  - The array itself already with a given big size
    - Some cells in the array will be empty (until it is full)
  - The number of elements currently in the list (its length)
    - That is, how many array positions are used

Last week we saw this

- **When I said that list implementation is closer to this:**



- **Where the 3 says that only the first 3 cells in the array are used**

We are going to use a simplified version of this, with colours and arrows to distinguish the used cells form the unused ones

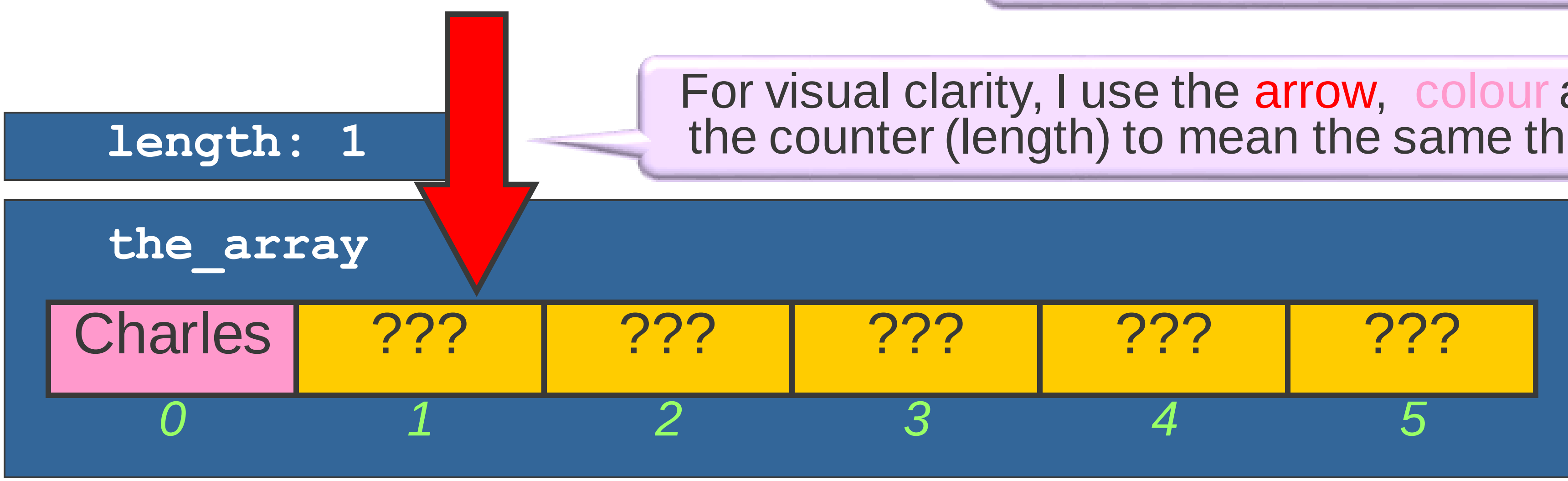
Visualising lists implemented with arrays

- **Consider a list defined:**
  - Over an array of size 6
  - Currently with one element (Charles)
- **We will visualise it like this:**

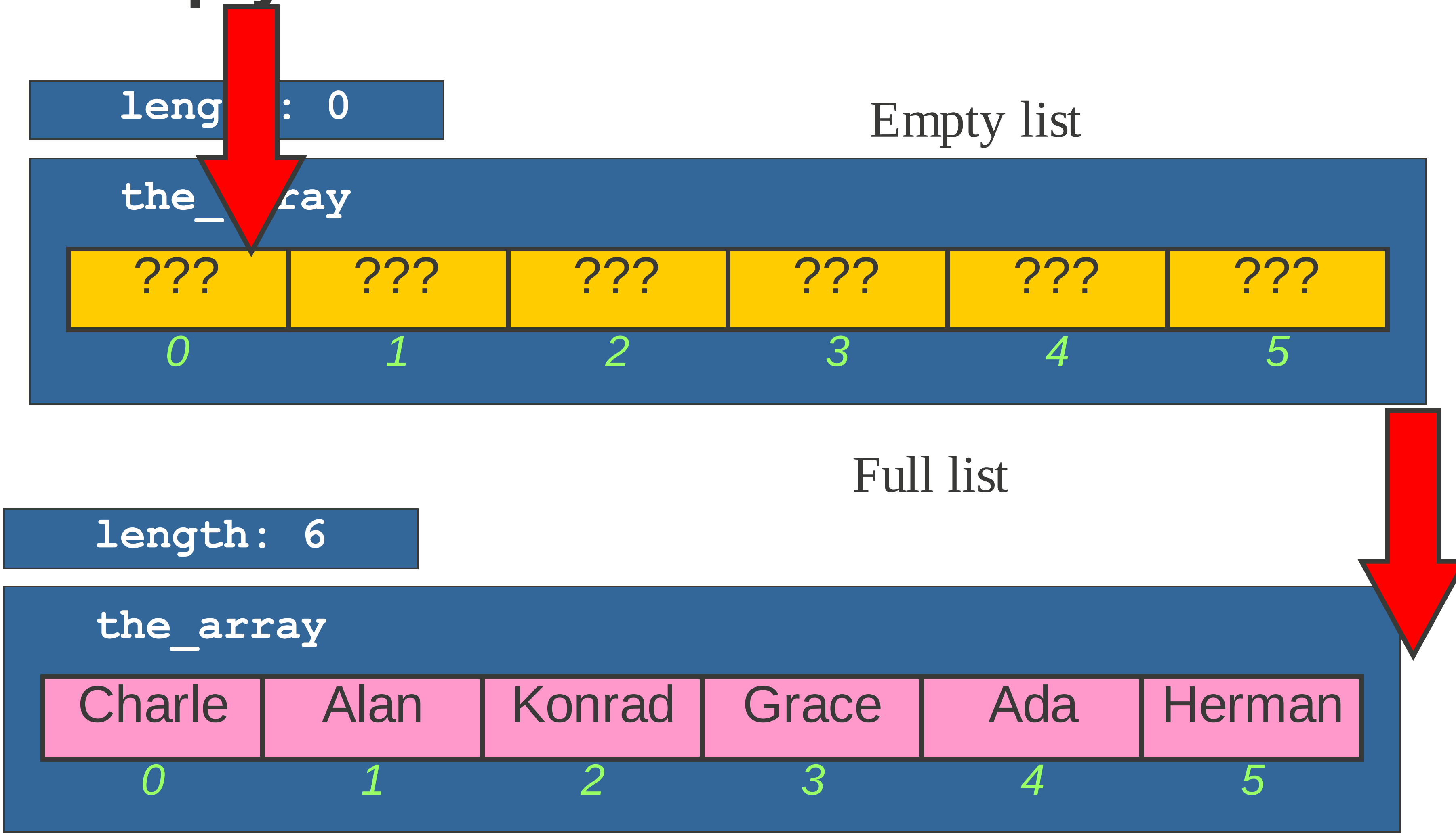
Invariant: the `length` points to the first free position in the array

In other words: valid data appear in the `0..length-1` positions

For visual clarity, I use the arrow, colour and the counter (length) to mean the same thing



Empty vs Full



How do we implement this in Python?

- **We need something that stores two things:**
  - The length (number of arrays cells used) and the array
- **We could use tuples: a sequence of elements**
  - Like a list but once created, cannot add, delete, reassign items

```
>>> x = [5,'h',6]
>>> tup = (3,x)
>>> tup
(3,[5,'h',6])
>>> x[2]=7
>>> tup
(3,[5,'h',7])
>>> tup[1]
[5,'h',7]
```

Modifying **x** affects the tuple

Can be accessed with list syntax

```
>>> (a,b) = tup
>>> a
3
>>> b
[5,'h',7]
>>> tup[0]=2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:'tuple' object does not
support item assignment(4,[5,'h',3])
```

Also accessed by *pattern matching* or *tuple unpacking*

BUT, cannot reassign items

Its immutable

Lists implemented with arrays - again

- **Cannot use tuples (are immutable!)**
- **We will use a Python list with two elements:**
  - The length of the list (number of “array” cells used)
  - The “array” itself (another Python list)
- **Have the operations changed?**
  - A bit

Lists implemented with arrays - again

- **How do we define the `def List(size)` function?**
- **As before but indicating the list is empty**
- **For example, if `size` is 5 we could create:**
  - Something like `[0,[None,None,None,None,None]]`
  - We don't have to use `None`:
    - Could use anything, like `[0,[1,3,0.5,'a',10]]`
    - Since `length` tells us the first non-valid position in the list
  - But it is customary to use `None` for "unintialised" variables
- **We saw how to use `[None]*5` to create the array**
- **Aside: in Python there is a more powerful way:**
  - Using the concept of list comprehension

Aside: List comprehensions

- **Used to define a list using mathematic-like notation**
  – By allowing us to create a list from another list
- **For example, in maths you might say:**
  – A = {3*x : x in {0 ... 9}}
  – B = {1, 2, 4, 8, ..., 2^{10}}
  – C = {x | x in A and x even}
- **In Python, you can easily define these:**

```
>>> A = [3*x for x in range(10)]
>>> B = [2**i for i in range (11)]
>>> C = [x for x in A if x % 2 == 0]
>>> A;B;C
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
[0, 6, 12, 18, 24]
```

Lists implemented with arrays - again

```
def List(size):
    return [0,[None]*size]

def get_item(the_list, index):
    return the_list[1][index]

def length(the_list):
    return the_list[0]

def is_empty(the_list):
    return the_list[0] == 0

def is_full(the_list):
    return the_list[0] >= len(the_list[1])
```

`[None for _ in range(size)]` is not as clear for this and slower

This first extracts the array ([1]) and then the item in position **`index`**

Could we simply return **`the_list[index]`**?

No, that would return the length, the array, or give an error

Could we return **`len(the_list)`**?

No, that would always return 2

What is the Big O time complexity of these functions?

They are all constant (not really for creation but we will assume it is), since they only access array elements, assign variables and compare integers. So O(1)

More clearly: meaningful variable names

```
def List(size):
    return [0,[None]*size]

def get_item(the_list, index):
    return the_list[1][index]

def length(the_list):
    return the_list[0]

def is_empty(the_list):
    return the_list[0] == 0
```

```
def List(size):
    length = 0
    the_array = [None]*size
    return [length,the_array]

def get_item(the_list, index):
    the_array = the_list[1]
    return the_array[index]

def length(the_list):
    length = the_list[0]
    return length

def is_empty(the_list):
    length = the_list[0]
    return length == 0
```

What about using what we just defined:

```
return length(the_list) == 0
```

Lists implemented with arrays - again

```
def linear_search(the_list, item):
    [length,the_array] = the_list
    for index in range(length):
        if item == the_array[index]:
            return True
    return False
```

Identical to the definition we used except the red (which “unpacks” the two elements of **the_list**)

Same time complexity?

Yes! Slightly bigger constant but still a constant

- **This was not quite the best definition. The best was:**

```
def is_in(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

- **Note: we cannot directly iterate over the elements**
  - Some positions in the array do not have valid content

Lists implemented with arrays - again

```
def binary_search(sorted_list,item):
    [length,the_array] = sorted_list
    low = 0
    high = length-1
    while low <= high:
        mid = (low+high)/2
        if the_array[mid] > item:
            high = mid-1
        elif the_array[mid] == item:
            return True
        else:
            low = mid+1
    return False
```

Again, almost identical to the definition we used (except the red)

- **Again, they still have the same time complexity**

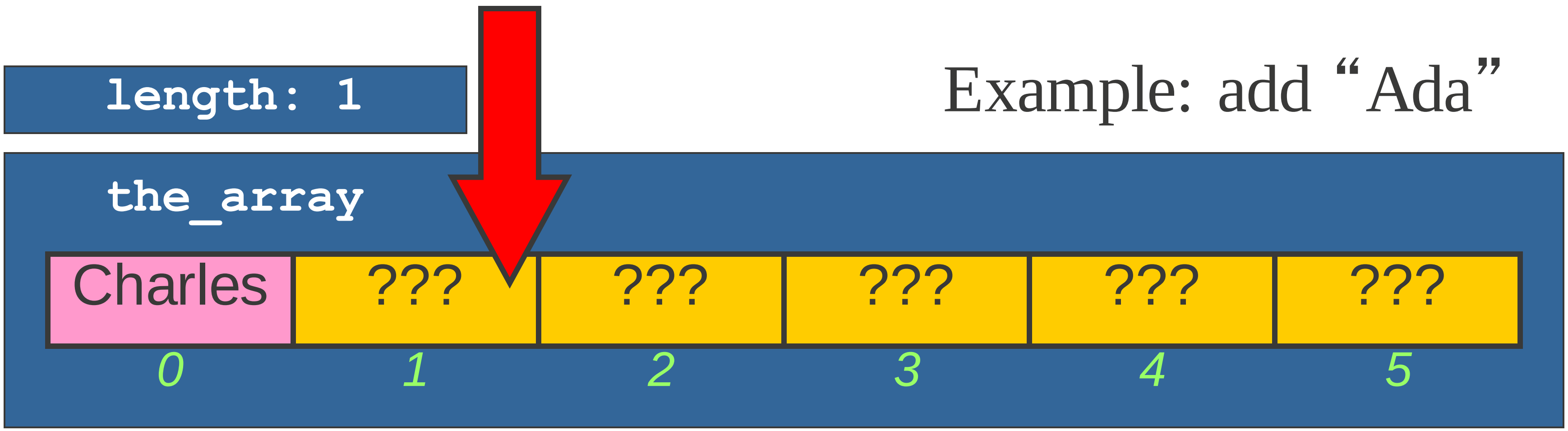
Adding an element to a list

- **Lets start by deciding what exactly do we want to do**
  - Input:
    - List (in our case: array + length)
    - Element to be added
  - Output:
    - List
    - Contains all original elements in the same order AND the input one (this is the post-condition)

Adding an element to a list (cont)

- Recall: **`length`** indicates the first empty position (if any)
- This gives us an idea for a possible plan

Example: add "Ada"

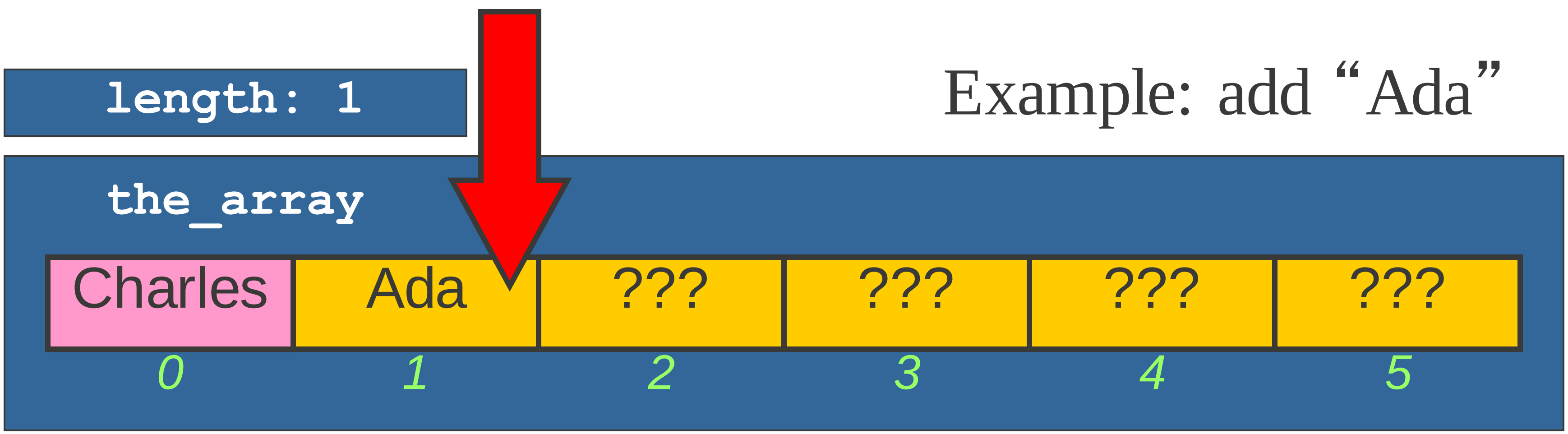


A list of 1 element

Adding an element to a list (cont)

- Recall: **`length`** indicates the first empty position (if any)
- This gives us an idea for a possible plan
  - Add the item at position **`length`**

Example: add "Ada"

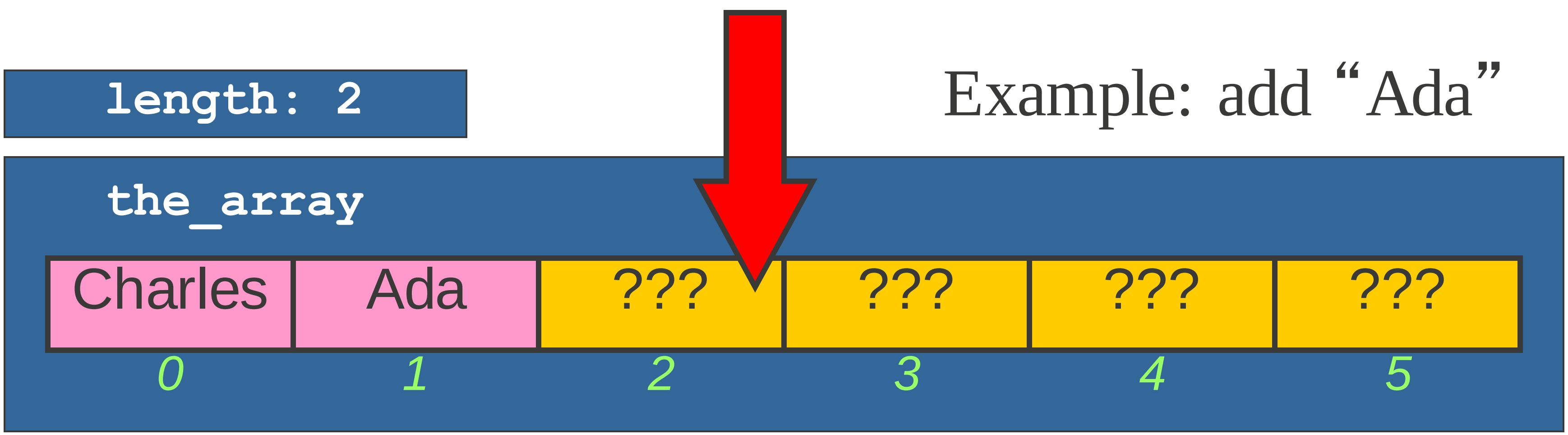


A list of 1 element

Adding an element to a list (cont)

- Recall: **`length`** indicates the first empty position (if any)
- This gives us an idea for a possible plan
  - Add the item at position **`length`**
  - Increment **`length`**

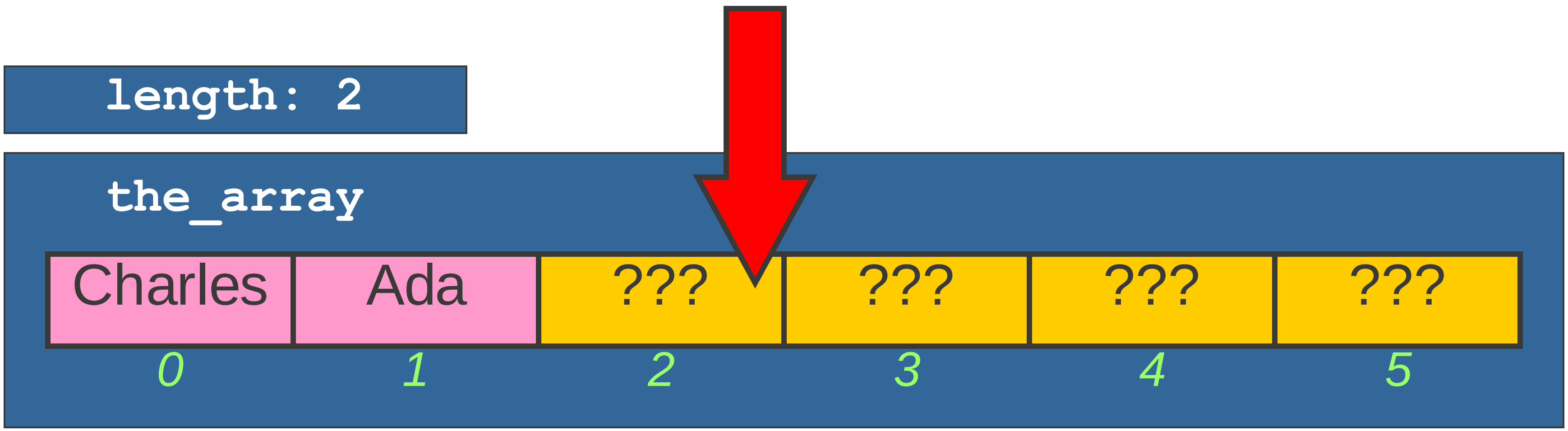
Example: add "Ada"



A list of 2 elements

Adding an element to a list (cont)

- **Why did we add Ada at the end of the list?**
  - Because `length` gave us easy access to an empty spot
- **Why not at the beginning (position 0)?**
  - Because would have to move Charles somewhere



A list of 2 elements

Adding an element to a list (cont)

- **Lets review our algorithm (add item to `length`, increment `length`)**
- **Does it always work?**
- **What are we trying to do?**
  - Add an item
- **We are assuming we can always add**
- **What if it is full? What to do then?**
  - One possibility: return `True` if we can, `False` otherwise
  - This changes the output AND the postcondition
  - Remember: Python does not do that (lists are never full…)

Revision: Main steps for alg development

- **Step 1: Understand the problem**
  - Relationship between input/output (ours was wrong)
- **Step 2: Devise a plan**
  - Think in terms of a small example
- **Step 3: Carry it out**
  - Write is as an algorithm (finite sequence of steps)
  - Apply it to your small example
- **Step 4: Review it**
  - Any cases for which it does not work? Then review
  - Improvements

Anything else?

- **Our algorithm has an extra postcondition:**
  - If **`True`** is returned, the added element appears last
- **Should we then call it `add` or `add_last`?**
  - I would say **`add_last`**
  - Lists are meant to be ordered even if not sorted
    - position IS important
  - Still, many list ADTs call it add (in Python it is **`append`**)
- **But users might not be interested in any order!**
  - Then, create an **`add`** function that
    - Calls **`add_last`** (or **`add_first`**, or whatever)
    - Indicates the element might be added in any position

Function add_last

```
def add_last(the_list, item):
    has_space_left = not is_full(the_list)
    if has_space_left:
        [length,the_array] = the_list
        the_array[length] = item
        the_list[0] = length + 1
    return has_space_left
```

Careful! You cannot say `length += 1`, as that modifies variable `length`, not `the_list[0]`

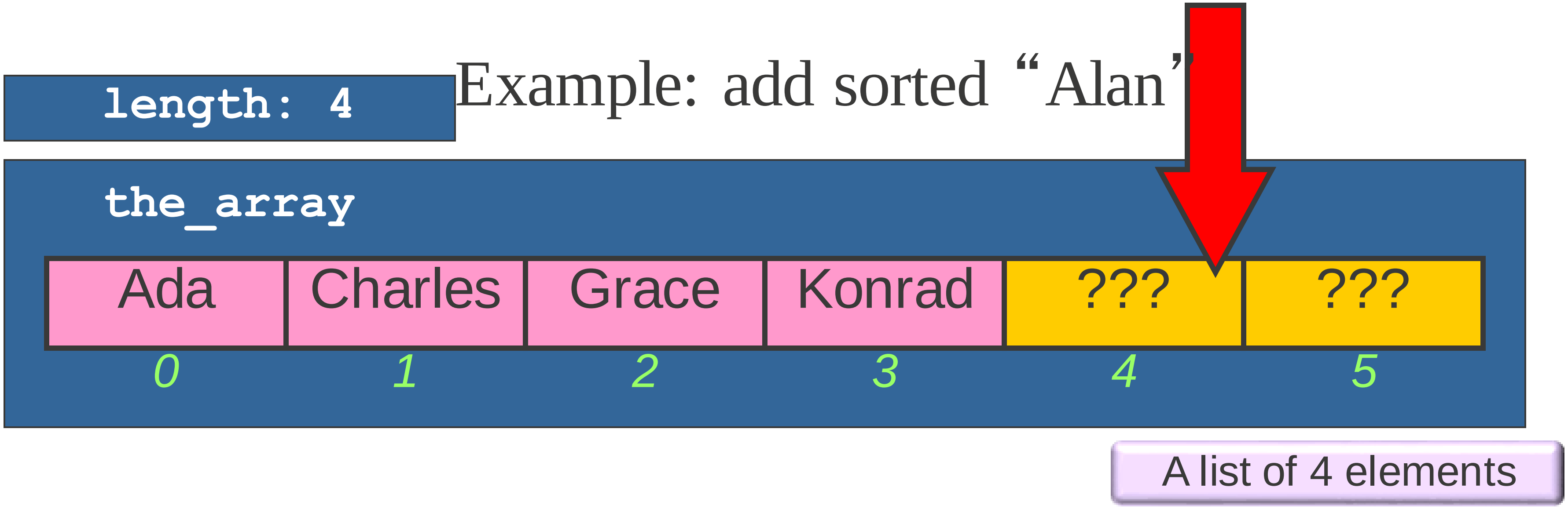
- **What is the big O time complexity?**
  - Every basic operation (access, assignment, addition) is constant
  - What about `is_full(the_list)`?
  - It was constant too, so O(1)

Adding an element to a sorted list

- **What if we are dealing with sorted lists?**
  - Element at position i is <= than that at postion i+1
- **What exactly do we want to do?**
  - Input:
    - Sorted list
    - Element to be added
  - Output:
    - Sorted list
    - Boolean: if false the list was full; if true, it contains all original elements in the same order AND the new one (postcondition)
  - Note:
    - the "Sorted" is also a pre/postcondition (might or might not be part of the type)

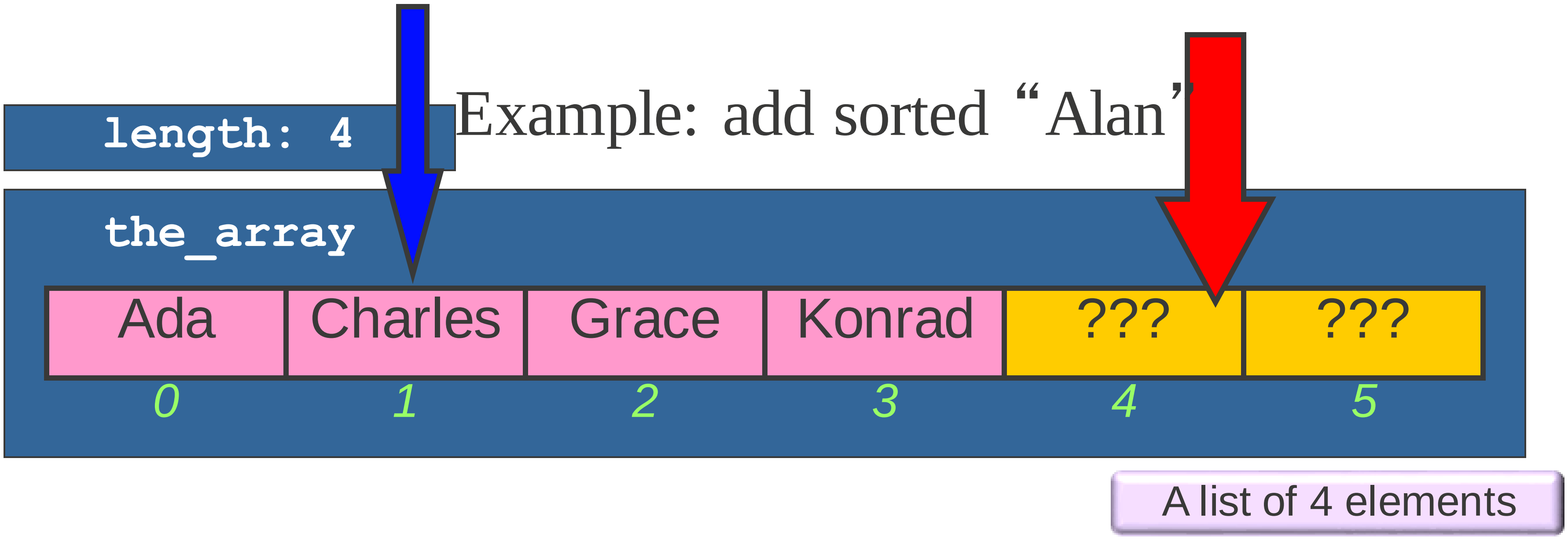
Sorted List: Add Sorted

- If there is space:



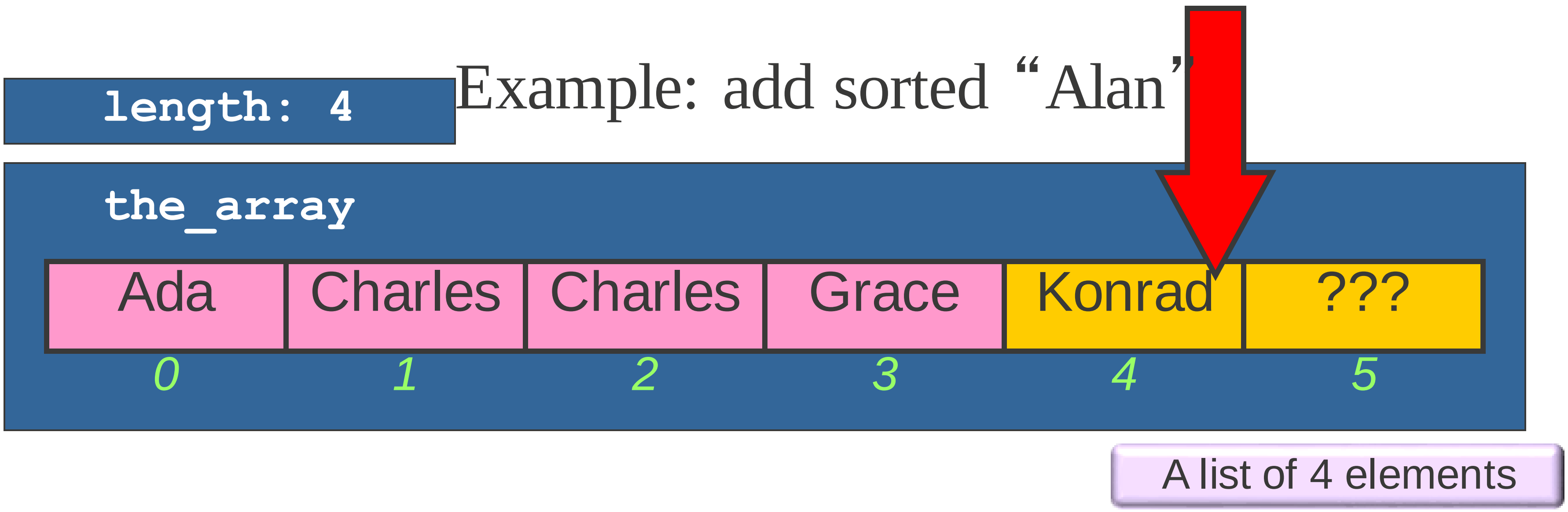
Sorted List: Add Sorted

- If there is space
  - Find correct position



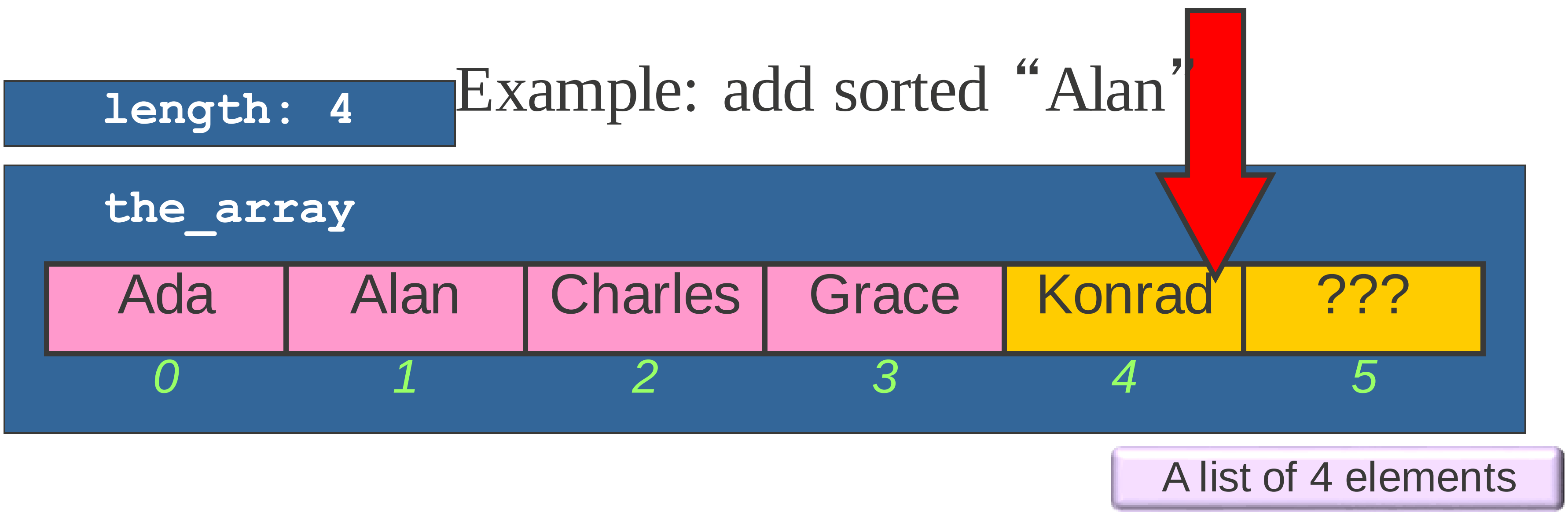
Sorted List: Add Sorted

- If there is space:
  - Find correct position
  - Make room by moving all to the right



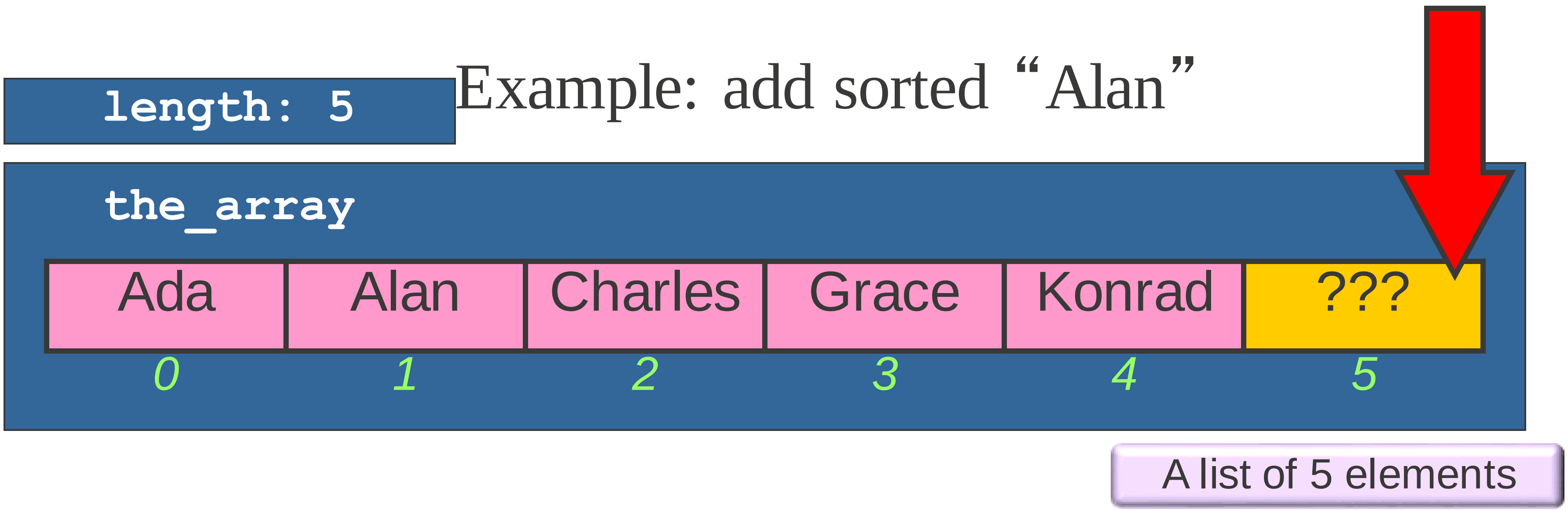
Sorted List: Add Sorted

- If there is space:
  - – Find correct position
  - – Make room by moving all to the right
  - – Put item in that position



Sorted List: Add Sorted

- If there is space:
  - Find correct position
  - Make room by moving all to the right
  - Put item in that position
  - Update `length` count



Sorted List: Add Sorted

- If there is space:
  - Find correct position
  - Make room by moving all to the right
  - Put item in that position
  - Update **`length`** count
  - Return **`True`**

Alphabetical order is maintained

Example: add sorted "Alan"

`length: 5`

`the_array`

Ada	Alan	Charles	Grace	Konrad	???
0	*1*	*2*	*3*	*4*	*5*

A list of 5 elements

Sorted List: Add Sorted

- **Do we really need to find the position first?**
- **Why not behave as if we were in insertion sort?**
  - Find the position P while shuffling elements > than item

If the array has some space left
    start at the rightmost element
    move to the right any element greater than item until P
    put item in position P
    increment $length$
    return true
else
    return false (no addition performed)

Method add_sorted

20	25	30	31	43	70			
0	1	2	3	4	5	6	7	8

```
def add_sorted(sorted_list, item):
    has_space_left = not is_full(sorted_list)
    if has_space_left:
        [length,the_array] = sorted_list
        i = length
        while i>0 and the_array[i-1]> item: #make room
            the_array[i] = the_array[i-1]
            i -= 1
        the_array[i] = item #put item in place
        sorted_list[0] = length+1 #increment lengh
    return has_space_left
```

One iteration of insertion sort

Let's see how it works using Python Tutor

Time complexity for add_sorted

20	25	30	31	43	70			
0	1	2	3	4	5	6	7	8

- **We have a single loop with O(m) operations**
  - Where m is the size of the elements (for the comparison)
- **Best case?**
  - The item is the greatest element: loop stops immediately
  - O(m)
- **Worst case?**
  - The item is the smallest element: loop goes all the way
  - O(m*n) where n is the size of the list

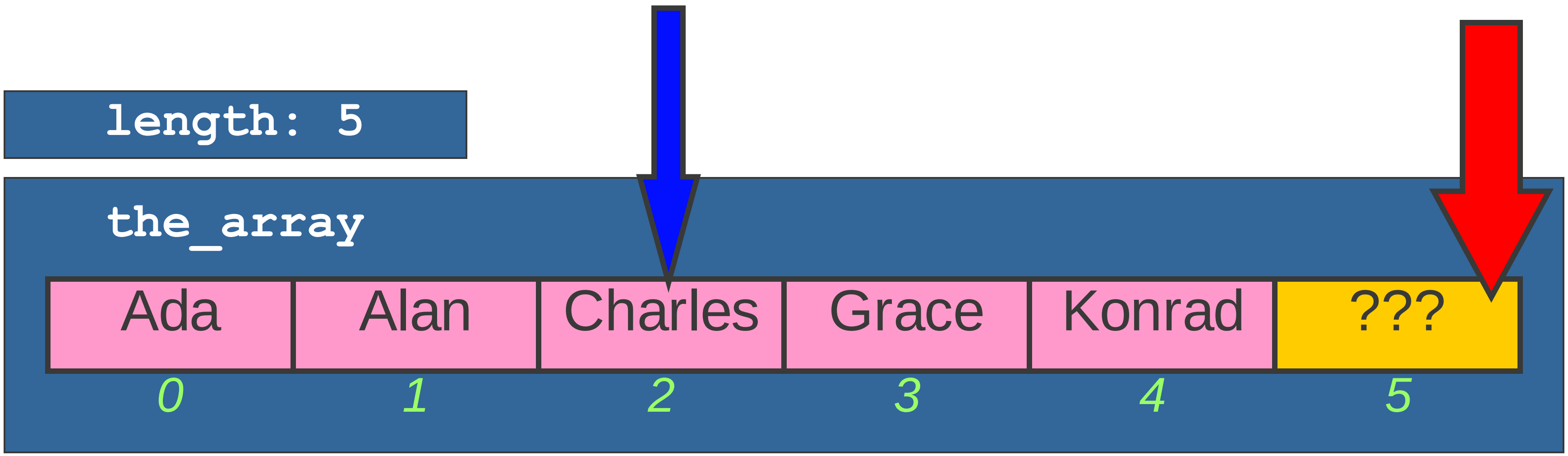
Deleting an element from a list

- **What exactly do we want to do?**
  - Given a list and the item to be deleted
  - Finish with a list that:
    - Has exactly the same elements as before
    - EXCEPT for the item, which is now not in the list
- **This is a little bit vague…**
  - What if item occurs several times in the list?
    - We delete only the first occurrence
  - Do the remaining elements need to appear in the initial order?
    - Let’s say yes (we will see later how to do it differently)

Deleting an element from a list

- Find the position of the element

Example: delete "Charles"

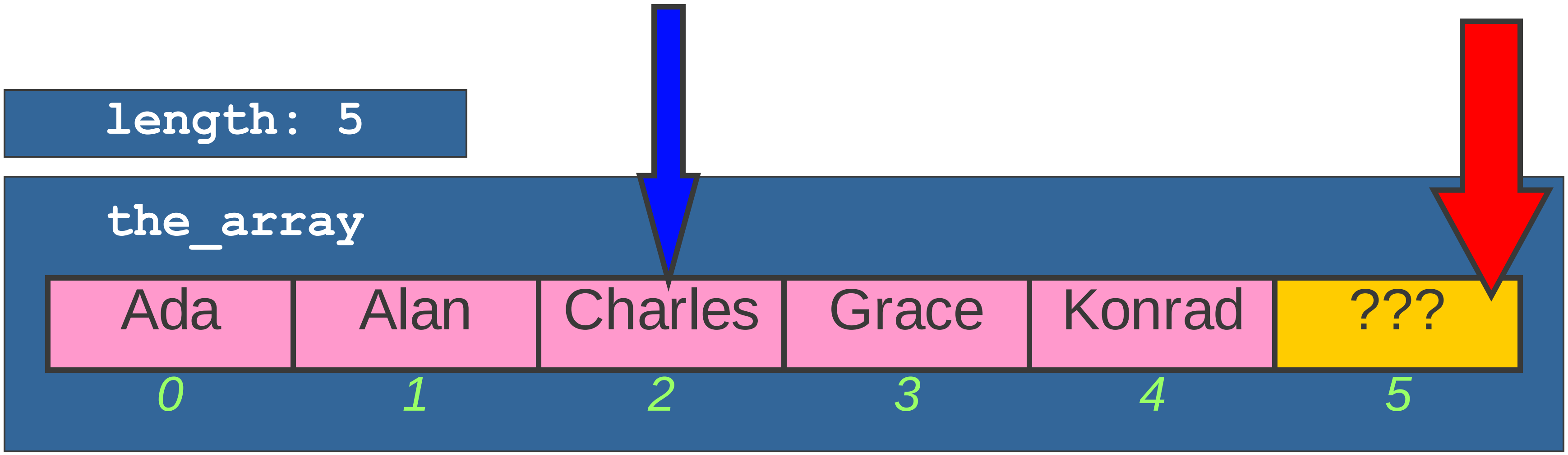


A list of 5 elements

Deleting an element from a list

- Find the position of the element
- Shuffle the items after the deleted item to the left

Example: delete “Charles”

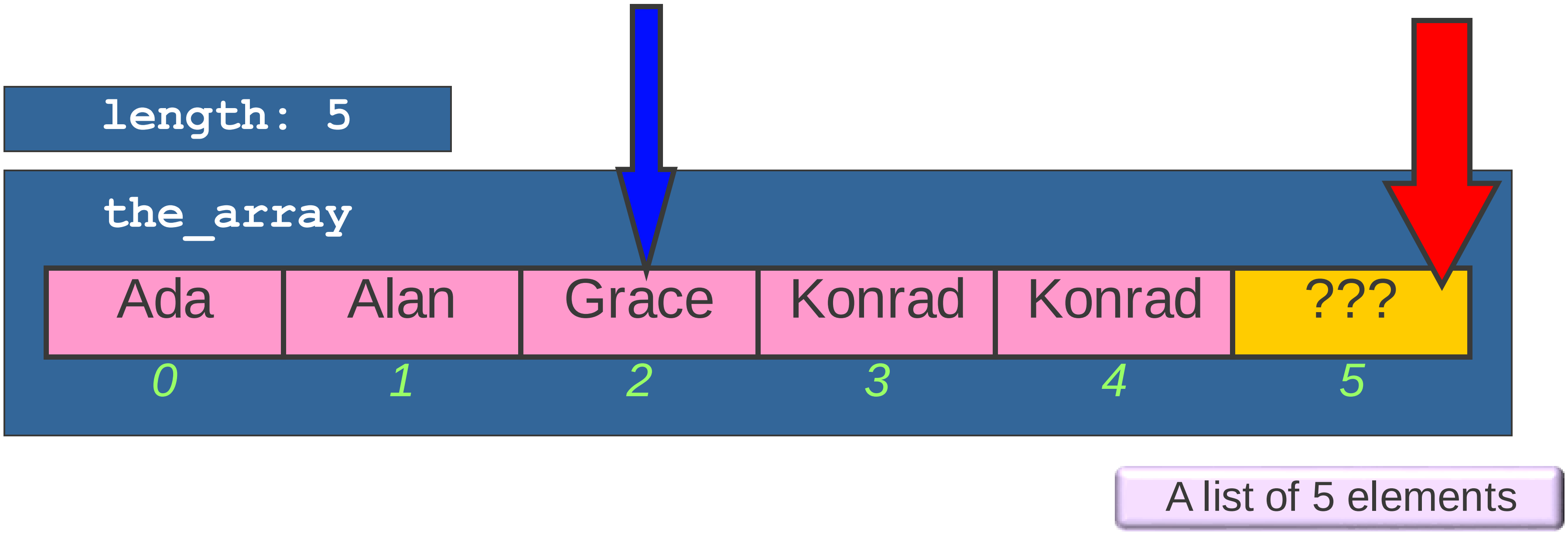


A list of 5 elements

Deleting an element from a list

- Find the position of the element
- Shuffle the items after the deleted item to the left

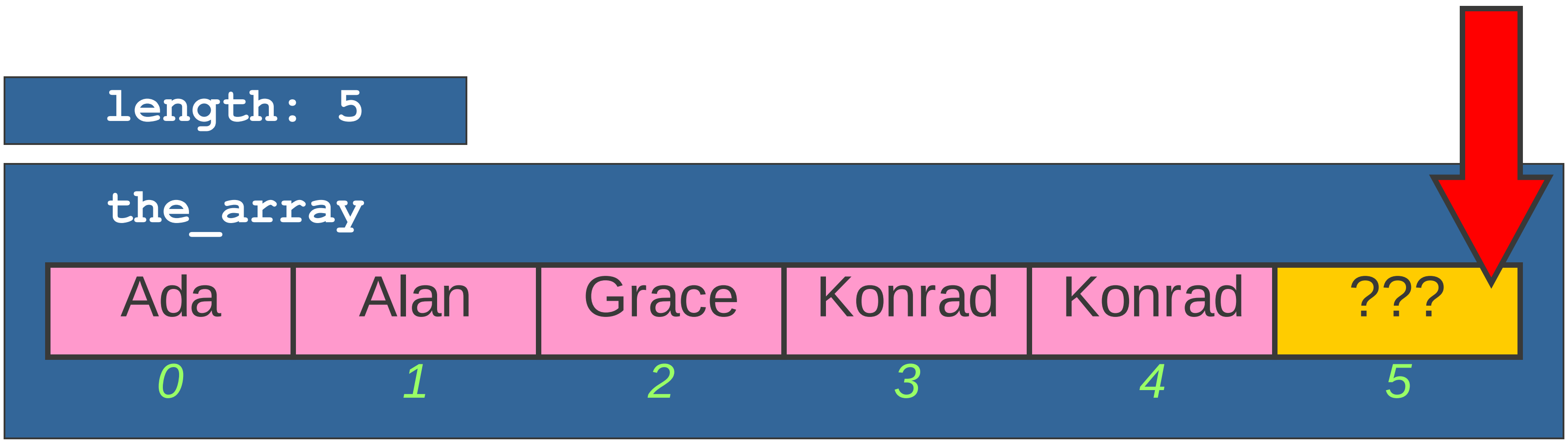
Example: delete "Charles"



Deleting an element from a list

- Find the position of the element
- Shuffle the items after the deleted item to the left
- Decrement `length`

Example: delete "Charles"

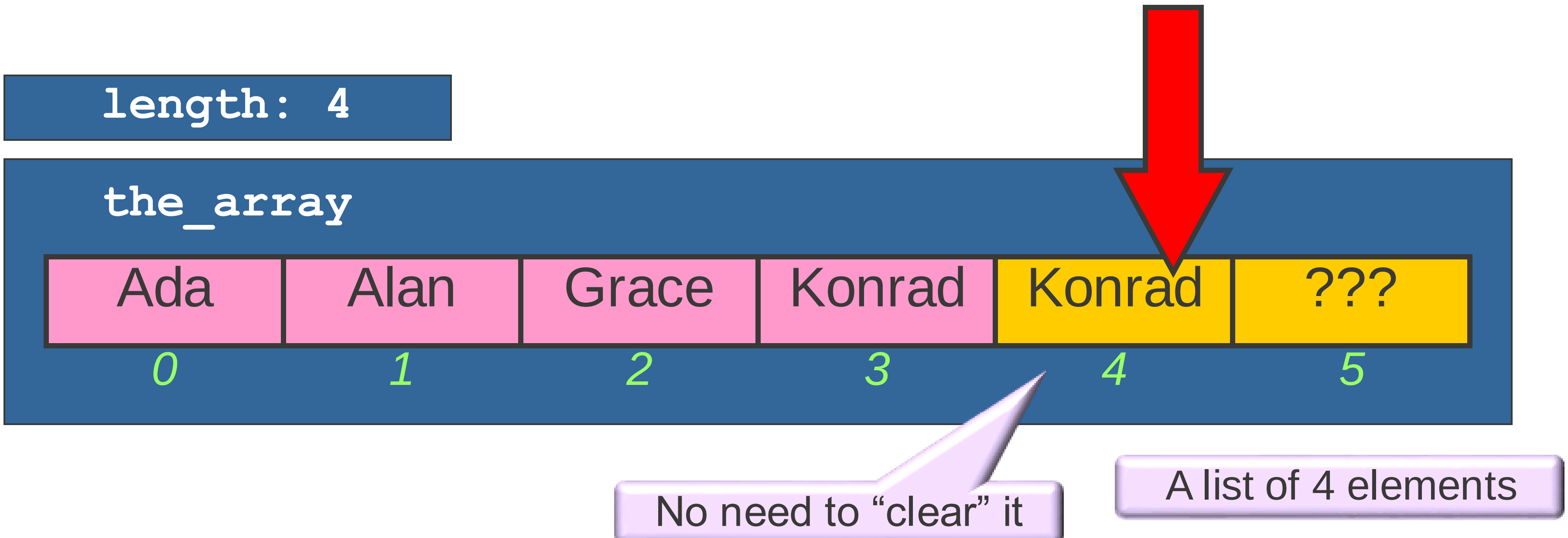


A list of 5 elements

Deleting an element from a list

- Find the position of the element
- Shuffle the items after the deleted item to the left
- Decrement **`length`**

Example: delete "Charles"



Alternatives

- **As we said, in our algorithm:**
  - Every element in the final list maintains its relative position
- **Thus, it could be used for sorted lists**
- **If we do not care about the relative order:**
  - We could simply swap the found element with the last one
  - Much simpler and faster

- **In FIT2085 we are going to assume we care**

Deleting an element from a list

- **Does this general algorithm always work?**
- **What are we trying to do now?**
  - Delete an item
- **We are assuming we can always delete it**
- **When can we not delete it?**
  - When the item is not there
- **What to do then?**
  - One possibility: return true if we can, false otherwise

List: Delete algorithm

Find the position P **at which the item appears**

If not found

return False (no deletion performed)

else

delete: move all $P+1$ **to** $length\text{-}1$ **items to the left**

decrement $length$

return True

Function delete_item

```
def delete_item(the_list, item):
    pos = index(the_list,item)
    found = (pos is not None)
    if found:
        [length,the_array] = the_list
        for i in range(pos,length-1):
            the_array[i] = the_array[i+1]
        the_list[0] = length-1
    return found
```

length-pos

Finds the position at which `item` appears in the list

A better version of `pos != None`, we will see later why

20	25	30	31	43	70			
0	1	2	3	4	5	6	7	8

Some elements get a constant amount? Depends

Time complexity for delete_item

All multiplied by M (size of elements) of course

- **We have two loops**
  - The search loop: best case $O(1)$, worst $O(N)$ or $O(\log_2 N)$
  - The shuffle loop: best case $O(1)$, worst $O(N)$
- **Best case? (the shuffle loop stops immediately)**
  - Not found + no shuffle
    - $O(\log_2 N) + O(1) \approx O(\log_2 N)$ (binary search)
    - $O(N) + O(1) \approx O(N)$ (linear search)
- **Worst case? (the shuffle loop goes all the way)**
  - Find it at the start of the list + shuffle all
    - $O(\log_2 N) + O(N) \approx O(N)$ (binary search)
    - $O(1) + O(N) \approx O(N)$ (linear search)

List slices

- **Python slices simplify the "making room" step**

```
for i in range(pos,length-1):
     the_array[i] = the_array[i+1]
```

```
>>> x = [0,1,2,3,4,5]
>>> x[1:3]
[1, 2]
>>> x[0:4]
[0, 1, 2, 3]
>>> x[:2]
[0,1]
```

```
>>> x[2:]
[2, 3, 4, 5]
>>> x[3:6]= x[2:5]
>>> x
[0, 1, 2, 2, 3, 4]
>>>
```

- **With slices: no need to write the loop (copy "in block"):**

```
the_array[pos:length-1] = the_array[pos+1:length]
```

Summary

- **Array representation**
- **Tuples and slices in Python**
- **Algorithms, methods and complexity of:**
  - Linear search
  - Binary search
  - Deleting elements
  - Adding elements