Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 19
Sorted Lists
(Array Implementation)

FIT 1008
Introduction to Computer Science

MONASH University
Information Technology

SortedList ADT

- Sequence of items in increasing order
- Possible Operations:
  - **Create** a list
  - **Add item** to the list
  - **Delete** an **item** at a given position from the list
  - Check whether the list **is empty**
  - Check whether the list **is full**
  - Get the **length** of the list.

```python
class SortedList:

    def __init__(self, size):
        if size > 0:
            self.the_array = size*[None]
            self.count = 0

    def length(self):
        return self.count

    def is_empty(self):
        return self.count == 0

    def is_full(self):
        return self.count >= len(self.the_array)
```

Adding an element to a sorted list

- **Sorted list**: Element at position i is <= than that at postion i+1

- **Input:**
  - Sorted list
  - new_item to be added
- **Output:**
  - Sorted list
  - **False** if the list was full; **True**, then the list contains all original elements in the same order together with the new_item (postcondition)
- **Note**:
  - the “Sorted” is also a pre/postcondition

Example: add "Goat" to the sorted list.

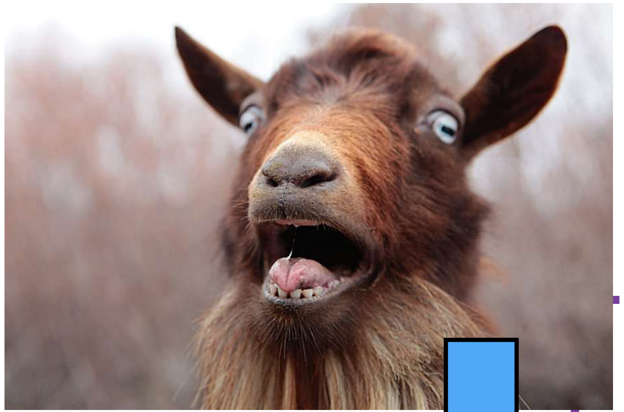
count: 4

the_array

0	1	2	3	4	5
Blobfish	Cheetah	Iguana	Tapir	???	???

Example: add "Goat" to the sorted list.

If there is space, find the correct position



Example: add “Goat” to the sorted list.

Order is maintained!

count: 5

the_array

Blobfish	Cheetah	Goat	Iguana	Tapir	???
0	1	2	3	4	5

If there is space, find the correct position

Make room by moving all to the right.

Put item in position.

Update count then **return True**

If the array has some space left:

find correct ***index*** at which to add item.

make room: move all items from **index** to **count-1** to the right

put item in position **index**.

increment **count**

return **True**.

else:

return **False**.

```python
def add(self, new_item):
    # easy if the list is empty
    if self.is_empty():
        self.the_array[self.count] = new_item
        self.count += 1
        return True
    # if the lis is not empty...
    has_place_left = not self.is_full()
    if has_place_left:
        # find correct position
        index = 0
        while index < self.count and new_item > self.the_array[index]:
            index+=1
        # now index has the correct position
        # we go backwards from count -1 up to index
        for i in range(self.count-1, index-1, -1):
            # "moving" the item in position i to position i+1
            self.the_array[i+1] = self.the_array[i]
        # insert new item
        self.the_array[index] = new_item
        # increment counter
        self.count+=1
    return has_place_left
```

Hence we can use the same delete as for a ListADT

```
def delete(self, index):
    valid_index = index >=0 and index < self.count
    if (valid_index):
        for i in range(index, self.count-1):
            self.the_array[i] = self.the_array[i+1]
        self.count -=1
    return valid_index
```

Print List

```
def print(self):
    for i in range(self.count):
        print(str(self.the_array[i]), end=" ")
```

Convert to string **whatever** is stored

What if the List contains more complex objects?

Overloading operators

- Any class can **redefine** certain special operations:
- By simply defining the **associated method** inside the class

Operation	Class Method
str(obj)	__str__(self)
len(obj)	__len__(self)
item in obj	__contains__(self,item)
y = obj[ndx]	__getitem__(self,ndx)
obj[ndx] = value	__setitem__(self,ndx,value)
obj == rhs	
obj < rhs	
...	
obj + rhs	
...	

Python checks whether the appropriate method is available to the object. If **not defined**, the **built-in operation** (if any) is used.

Creating a list

```
class List:
    def __init__(self, size):
        assert size > 0, "Size should be positive"
        self.the_array = size*[None]
        self.count = 0
```

```
object.__mul__(self, other)
```

Operation * is <u>overloaded</u> in Python's List type

https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

Operation	Class Method
str(obj)	__str__(self)
len(obj)	__len__(self)
item in obj	__contains__(self,item)
y = obj[ndx]	__getitem__(self,ndx)
obj[ndx] = value	__setitem__(self,ndx,value)
obj == rhs	__eq__(self,rhs)
obj < rhs	__lt__(self,rhs)
...	
obj + rhs	__add__(self,rhs)
...	

```
def __len__(self):
    return self.count

def is_empty(self):
    return len(self) == 0

def is_full(self):
    return len(self) >= len(self.the_array)
```

Operation	Class Method
str(obj)	__str__(self)
len(obj)	__len__(self)
item in obj	__contains__(self,item)
y = obj[ndx]	__getitem__(self,ndx)
obj[ndx] = value	__setitem__(self,ndx,value)
obj == rhs	__eq__(self,rhs)
obj < rhs	__lt__(self,rhs)
…	
obj + rhs	__add__(self,rhs)
…	

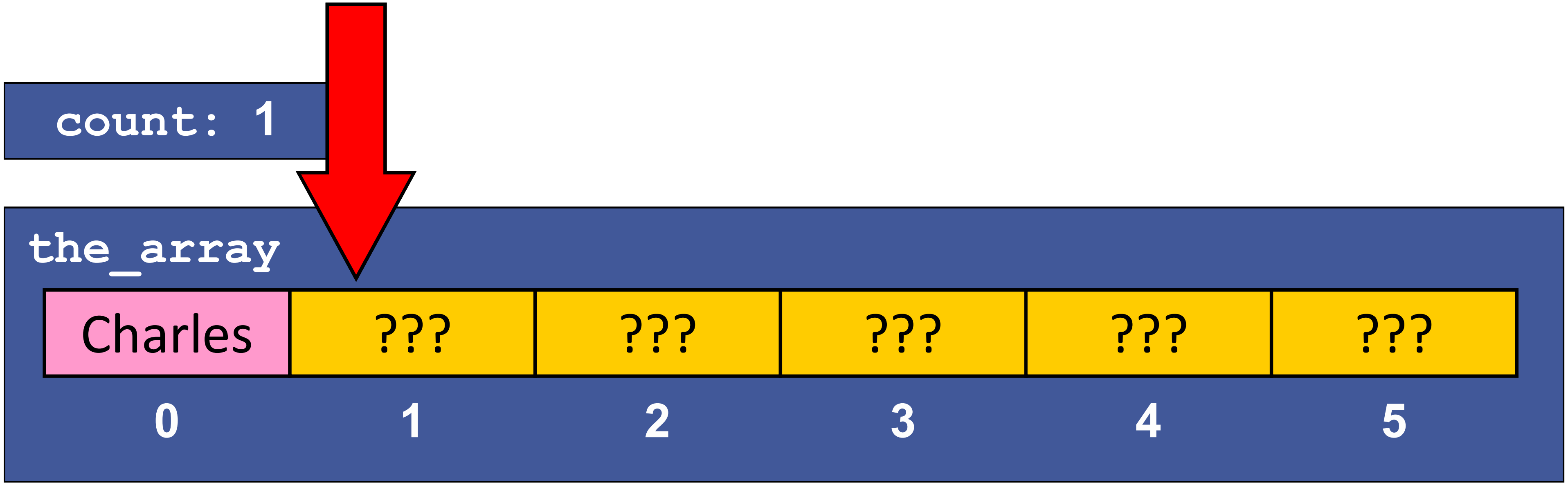
Searching

List ADT

- Sequence of items

- Possible **Operations**:
  - Add item
  - Remove item
  - Find item
  - Retrieve item
  - Next item
  - First item
  - Is last item
  - Is empty
  - Print
  - **Is an item in the list?**
  - **Find the first position of an item in the list.**

- Is an item in the list?
- Find the first position of an item in the list.



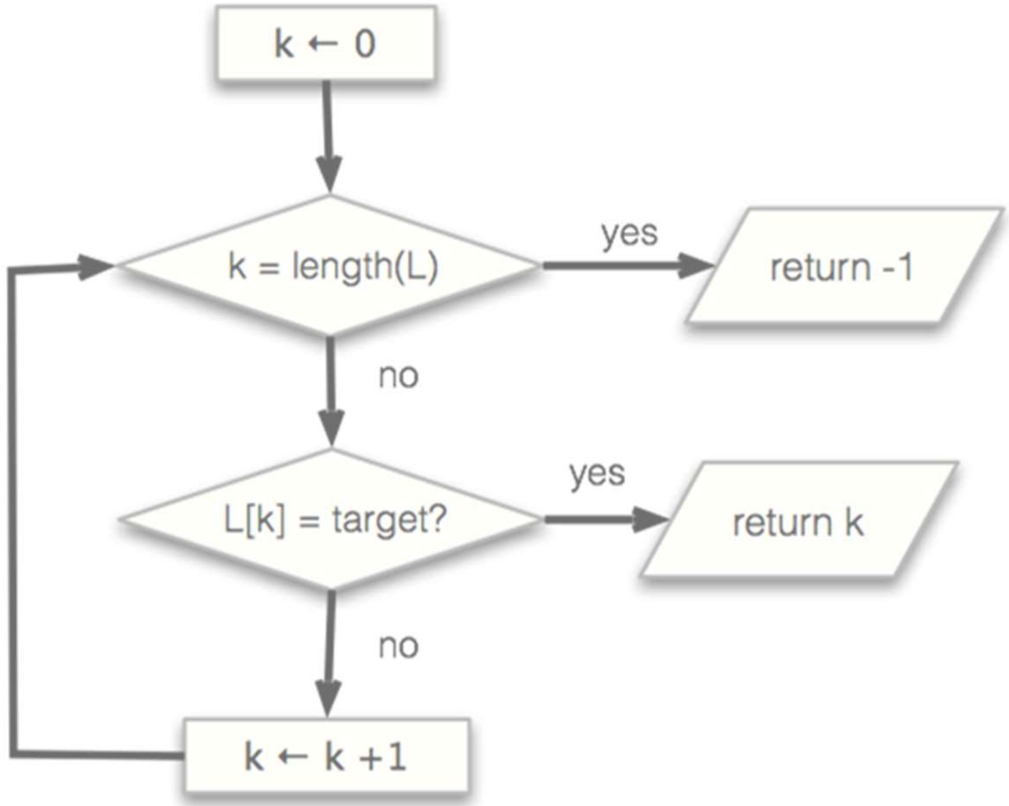
Item in List

```
>>> the_list = [1, 2, 3, 4, 5]
>>> x = 3
>>> x in the_list
True
>>> y = 8
>>> y in the_list
False
```

Item in List

```
def __contains__(self, item):
    for k in range(len(self)):
        if item == self.the_array[k]:
            return True
    return False
```

Linear Search



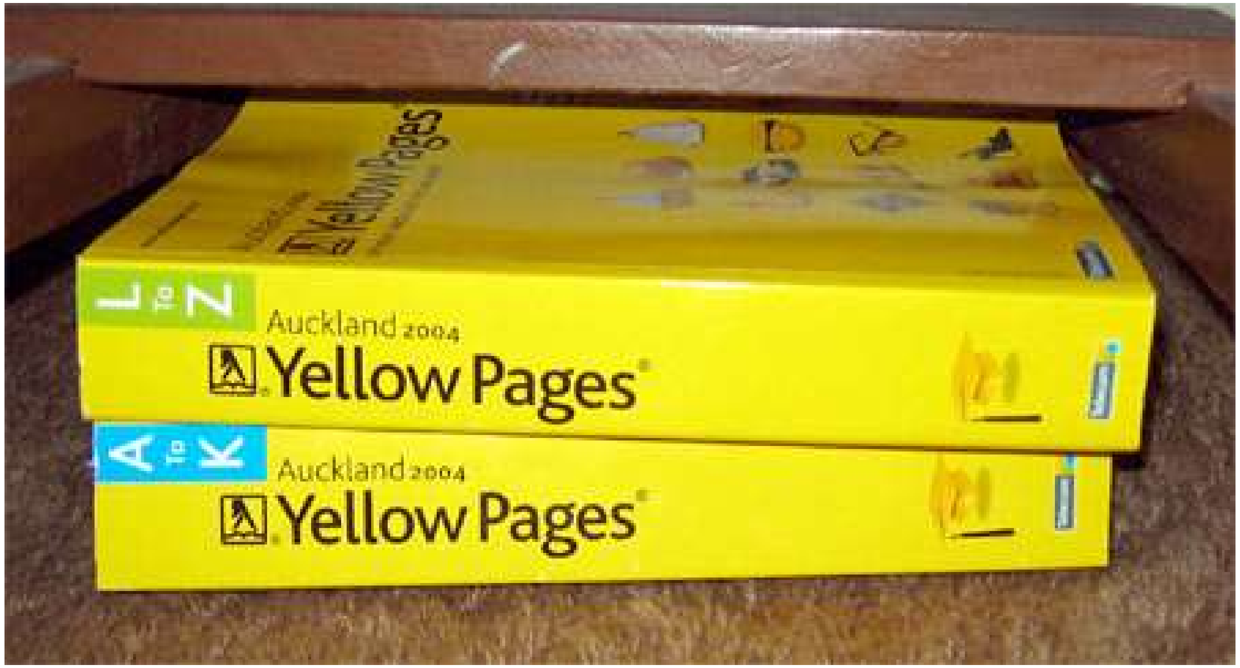
```
class SortedList:

    def __init__(self, size):
        if size > 0:
            self.the_array = size*[None]
            self.count = 0

     def __len__(self):
        return self.count

    def is_empty(self):
        return len(self) == (

    def is_full(self):
        return len(self) >=
```

```
def add(self, new_item):
    # easy if the list is empty
    if self.is_empty():
        self.the_array[self.count] = new_item
        self.count += 1
        return True
    # if the lis is not empty...
    has_place_left = not self.is_full()
    if has_place_left:
        # find correct position
        index = 0
        while index < self.count and new_item > self.the_array[index]:
            index+=1
        # now index has the correct position
        # we go backwards from count -1 up to index
        for i in range(self.count-1, index-1, -1):
            # "moving" the item in position i to position i+1
            self.the_array[i+1] = self.the_array[i]
        # insert new item
        self.the_array[index] = new_item
        # increment counter
        self.count+=1
    return has_place_left
```


```
def add(self, new_item):
    # easy if the list is empty
    if self.is_empty():
        self.the_array[self.count] = new_item
        self.count += 1
        return True
    # if the lis is not empty...
    has_place_left = not self.is_full()
    if has_place_left:
        # find correct position
        index = 0
        while index < self.count and new_item > self.the_array[index]:
            index+=1
        # now index has the correct position
        # we go backwards from count -1 up to index
        for i in range(self.count-1, index-1, -1):
            # "moving" the item in position i to position i+1
            self.the_array[i+1] = self.the_array[i]
        # insert new item
        self.the_array[index] = new_item
        # increment counter
        self.count+=1
    return has_place_left
```


List is always sorted!

Binary Search



Source:http://www.geekets.com/

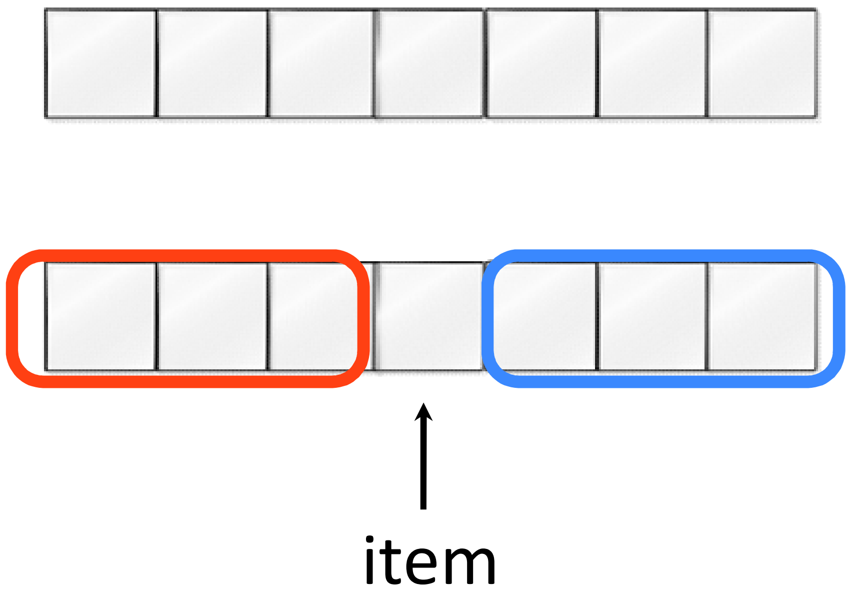
Binary Search Assumptions

- The list is sorted

- We can random access the list
  (you can get the value of any position in the list)

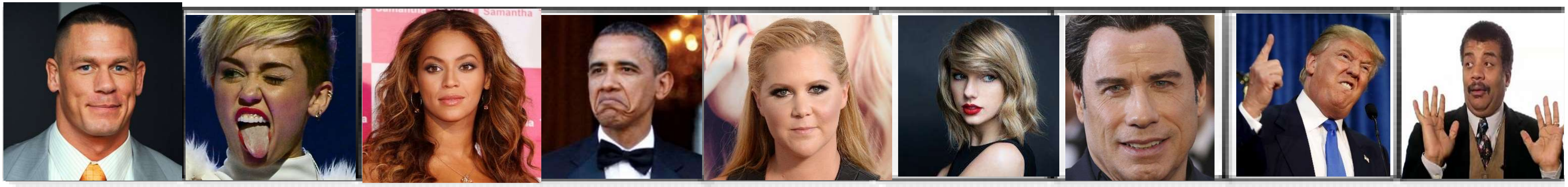
Binary Search

```
item ⟵ the item in the middle of the list

if (item = target)
{
        return True
}

if (target < item)
{
        search the first part of the list
}

if (target > item)
{
        search the second part of the list
}
```



item < target   item = target   item > target

Remember how I found TayTay?



John Cena, Miley Cyrus, Beyonce' Knowles, Barack Obama, Amy Schumer, Taylor Swift, John Travolta, Donald Trump, Neil Degrasse Tyson

Binary Search

Case 2: `target > the_array[mid]`

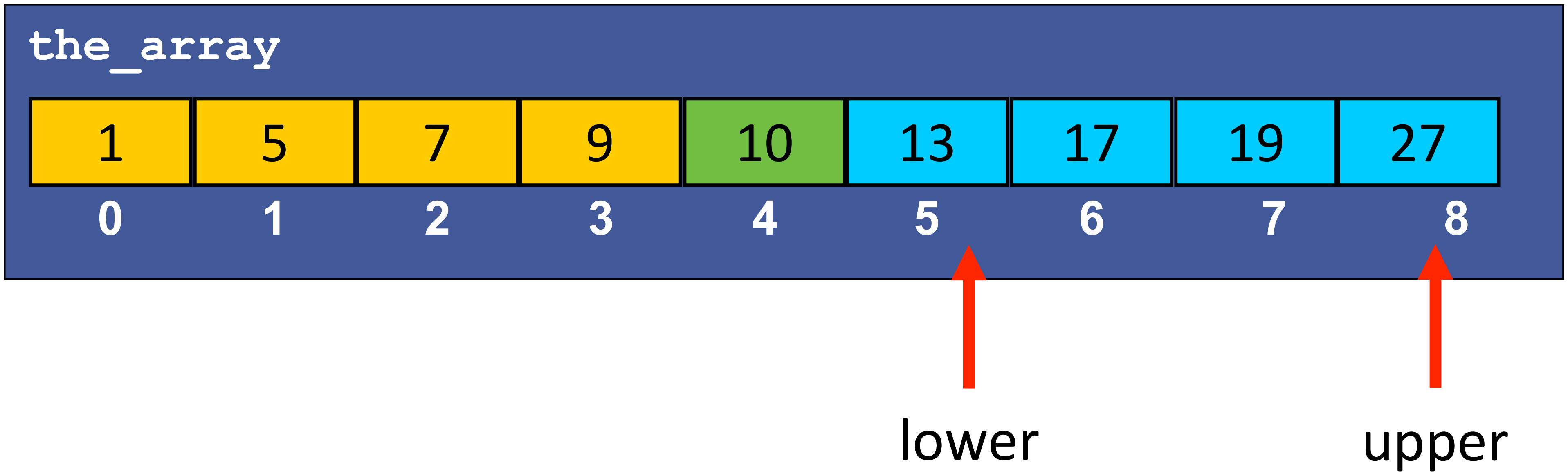
target = 19

lower = 0, upper = 8

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

update **lower** and keep searching....

lower = mid+1 = 5



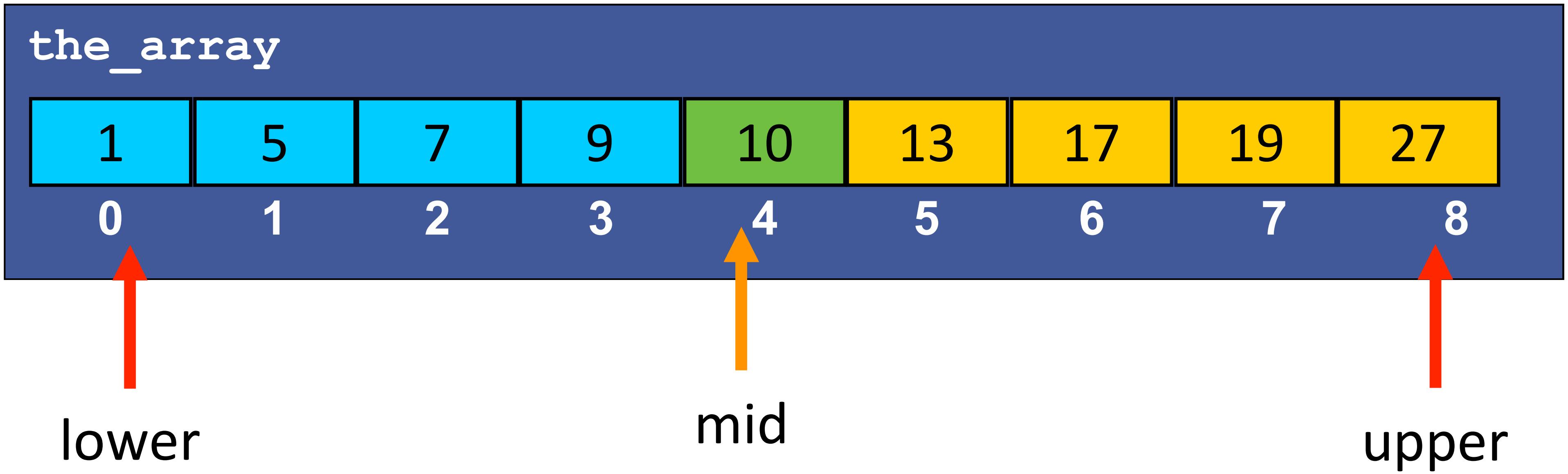
Binary Search

<u>Case 3:</u> `target < the_array[mid]`

target = 7

lower = 0, upper = 8

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



Binary Search

<u>Case 4:</u> `target` not in the list

target = 11

lower = 0, upper = 8

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

lower = 5, upper = 8

mid =(5+8)//2 = 13//2 = 6

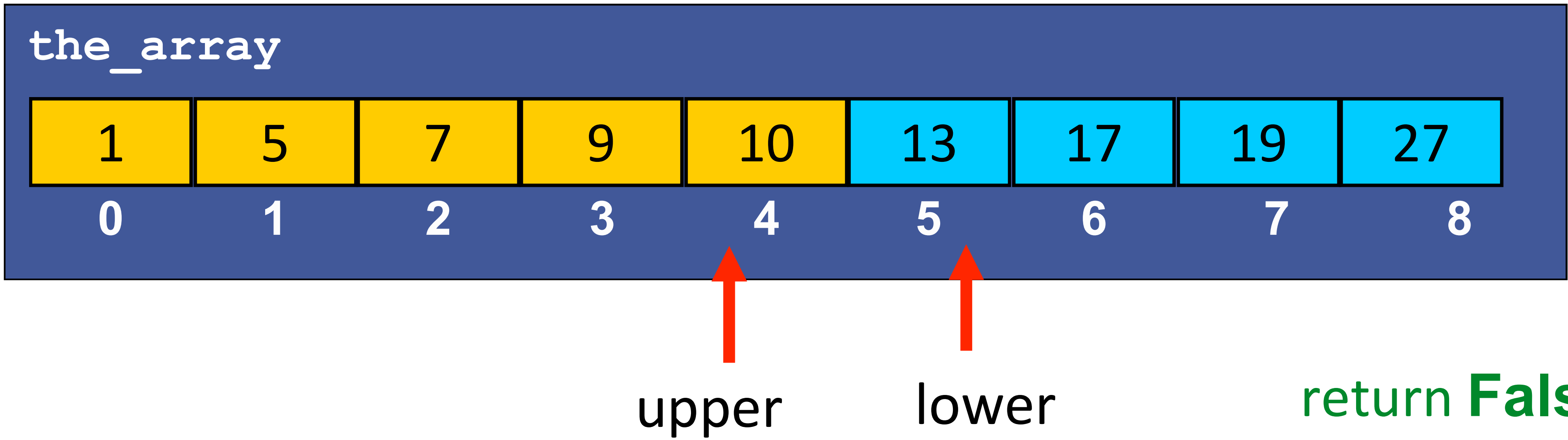
lower = 5, upper = 5

mid =(5+5)//2 = 10//2 = 5

upper = mid -1

upper < lower

if Target is not in the List



return **False**

Algorithm: BinarySearch(target, L[0..n-1])

Search for target in L

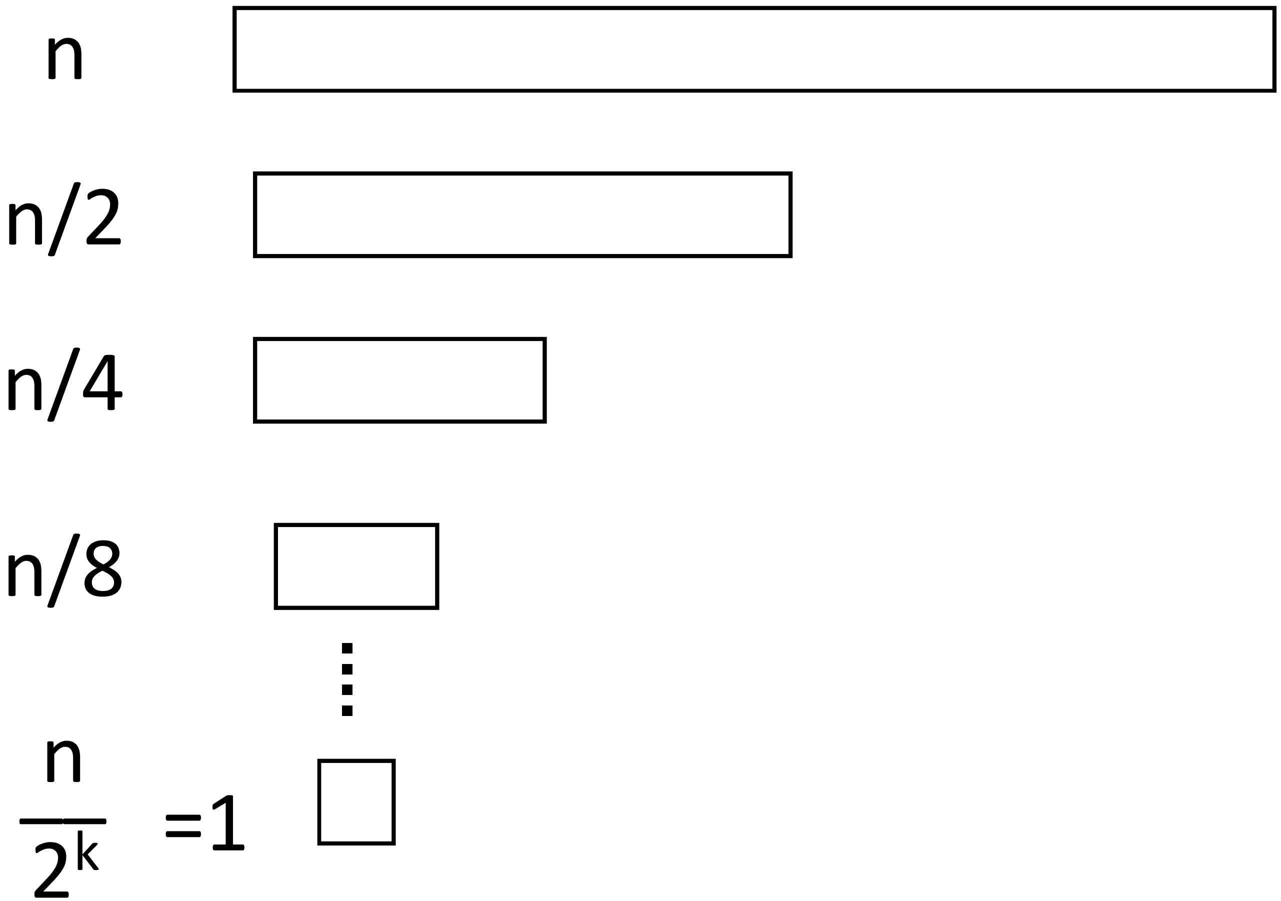
Input: target, and a <u>sorted</u> list L[0, n-1]

Output: If target is in L, returns the index of the first item with that value. Otherwise returns -1.

```
lower ⟵ 0
upper ⟵ n-1
while (lower ≤ upper) do {
    mid ⟵ ⌊ (lower + upper)/2 ⌋
    if (target = L[mid])
        return True
    if (target < L[mid])
        upper ⟵ mid – 1
    if (target > L[mid])
        lower ⟵ mid + 1
}
return False
```

Binary Search

```
def __contains__(self, item):
    lower = 0
    upper = len(self) -1
    while lower <= upper:
        mid = (lower + upper)//2
        if self.the_array[mid] == item:
            return True
        elif self.the_array[mid] > item:
            upper = mid - 1
        else:
            lower = mid + 1
    return False
```



$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log_2(n) = k$$

Binary Search: **Assumptions**

- To use Binary Search we need:
  - List to be **sorted**
  - 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 access to <u>any element</u> in the list
  - We need to do that efficiently: **constant time** access
  - Arrays ensure that is always the case

list.index(item)

- Finds the **first index** of an item in the list.
- Raises `ValueError` if item is not in list.

```
>>> the_list = [1, 2, 2, 4, 5]
>>> the_list.index(2)
1
>>> the_list.index(8)
Traceback ...
     .....
ValueError: 8 is not in list
```

`list.`**`index`**`(`*x*`)`

Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

https://docs.python.org/3/tutorial/datastructures.html

Sorted List

`list.`**`index`**`(`*x*`)`

Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

Sorted List

?

```
def _binary_search(self, item):
    lower = 0
    upper = len(self) -1
    while lower <= upper:
        mid = (lower + upper)//2
        if self.the_array[mid] == item:
            return mid
        elif self.the_array[mid] > item:
            upper = mid - 1
        else:
            lower = mid + 1
    return -1
```

now it <u>returns</u> a valid index **or** -1

Sorted List

```python
def __contains__(self, item):
    if self._binary_search(item) == -1:
        return False
    return True
```

Sorted List

```
def index(self, item):
    position = self._binary_search(item)
    if position == -1:
        raise ValueError("Element " + str(item) + " is not in the list")
    while position >=0 and self.the_array[position] == item:
        position -= 1
    return position + 1
```

Check if there are copies of the target that appear earlier

Sorted List

```
def index(self, item):
    position = self._binary_search(item)
    if position == -1:
        # item is not in the list
        raise ValueError("Element " + str(item) + " is not in the list")
    # there is at least a copy and is in position position
    while position >=0 and self.the_array[position] == item:
        # search back until we find the first appearance
        position -= 1
    return position + 1
```

Best-case time complexity O(1)

you do better than linear?

```
def index(self, item):
    position = self._binary_search(item)
    if position == -1:
        # item is not in the list
        raise ValueError("Element " + str(item) + " is not in the list")
    # there is at least a copy and is in position position
    while position >=0 and self.the_array[position] == item:
        # search back until we find the first appearance
        position -= 1
    return position + 1
```

Summary

- Implementing lists using arrays:
  - Class structure for a list
  - Add an element to an unsorted list
  - Add an element to a sorted list
  - Delete an element