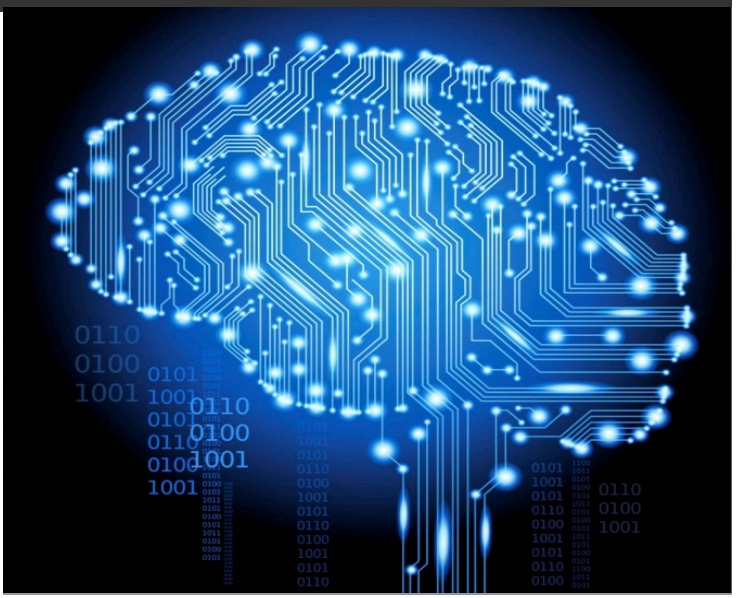
MONASH University

Information Technology

List ADT implemented using arrays

Prepared by:
Maria Garcia de la Banda
Revised by Pierre Le Bodic



Objectives for this lesson

- **Understand the List ADT:**
  - Main operations
  - Their complexity
- **Be able to:**
  - Implement the List ADT using arrays
  - Modify its operations and
  - Reason about their complexity
- **To be able to decide when it is appropriate to use Stacks, Queues or Lists**

MONASH University

List ADT

What is a List Abstract Data Type (or List ADT)?

- **An ADT that is used to store items**
- **That is very vague! What makes it a list?**
  1. Elements have an order (first, second, etc)
     - This does not mean they are sorted!
  2. Must have direct access to the first element (head)
  3. From one position you can always access the "next" (if any)
- **What else? The operations for the list ADT are not well defined**
  - Different texts/languages provide very different set of operations
- **They often agree on a core set of operations, which includes:**
  - Creating, accessing, computing the length
  - Testing whether the list is empty (not for full … we will see why)
  - Adding, deleting, finding, retrieving and modifying an element

Modification is the main difference between lists and tuples (mutables and immutables)

Our implementation of List ADT

- **In this lesson we will define and implement our own list ADT**
  - Why on earth? They are already in Python!
- **Again, because the learning objectives of this unit require you to:**
  - Learn to implement the operations yourself
    - You might need to program on a device with limited memory
  - Reason about the properties of these operations
  - Understand the changes in properties depending on implementation
- **What data structure do we use to implement it?**
  - We will start with arrays (later, linked nodes)
  - As for Stacks and Queues, we will use the **`ArrayR`** class for the arrays

Implementing your own List ADT

- **How do we start? Easy:**
  - Create a new file (called my_list.py)
  - Add a class List with any operations users will ever need to use!
- **But what operations? We could do many…**
  - Create a list, get/set the element at a given position, compute the length
  - Determine whether is empty
  - Find the position (or index) of an item (if in)
  - Insert an item in position P
  - Append an item (same as insert at the back)
  - Remove the first occurrence an item
  - Delete and return the item in position P (Python calls this pop; I find that confusing)
  - Clear the list

Abstract base List class

```
from abc import ABC, abstractmethod
from typing import TypeVar, Generic
T = TypeVar('T')

class List(ABC, Generic[T]):
    def __init__(self) -> None:
        self.length = 0

    @abstractmethod
    def __setitem__(self, index: int, item: T) -> None:
        pass

    @abstractmethod
    def __getitem__(self, index: int) -> T:
        pass

    @abstractmethod
    def append(self, item: T) -> None:
        pass

    @abstractmethod
    def insert(self, index: int, item: T) -> None:
        pass

    @abstractmethod
    def delete_at_index(self, index: int) -> T:
        pass

    @abstractmethod
    def index(self, item: T) -> int:
        pass
```

Called **pop** in Python

Very similar to the Stack and Queue classes: changes in blue

Main differences: methods **__get_item__**, **__set_item__**, **append**, **insert**, **delete_at_index**, **remove** and **index**

Also, no **is_full**… mmhh (see later)

```
def remove(self, item: T) -> None:
    index = self.index(item)
    self.delete_at_index(index)

def __len__(self) -> int:
    return self.length

def is_empty(self) -> bool:
    return len(self) == 0

def clear(self):
    self.length = 0
```

MONASH University

List ADT implemented with arrays, part I

Let's start implementing ArrayList

- **Create the class and its __init__ constructor:**

```
from referential_array import ArrayR
from abstract_list import List, T

class ArrayList(List[T]):
    MIN_CAPACITY = 1

    def __init__(self, max_capacity:int) -> None:
        List.__init__(self)
        self.array = ArrayR(max(self.MIN_CAPACITY,max_capacity))
```

- **Time complexity?**
  - Call to `ArrayR()` is linear on `max_capacity` (as seen for Stacks and Queues)
  - Everything else is constant, so the sum is O(`max_capacity`)
- **Any properties of the list elements that affect big O?** No! so best = worst
- **Invariant: valid elements between 0 and self.length -1, the head is at 0**

Now for the absolute basic methods

- **Accessing the elements and clearing the list:**

```
def __getitem__(self, index: int) -> T:
    return self.array[index]

def __setitem__(self, index: int, value: T) -> None:
    self.array[index] = value
```

- **Time complexity?**
  - The return statement, array access, length access and assignment are constant
  - Sum of constants is a constant so → O(1)
- **Any properties of the list elements that affect big O?** No! so best = worst

Finding the position of an item in the List

- **Input:**
  - List
  - Item
- **Output:**
  - The position of the first occurrence of the item, or `ValueError` if not there
- **This is basically a linear (sequential or serial) search:**
  - Start at one end of the list
  - Look at each element (advancing to the other end) until the element you are looking for is found

Dead easy, as you have done this in previous units, but will repeat to reinforce

How should we implement it in Python?

- **A possible implementation is:**

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

- **Wait! Python can generate all elements within a list!** `for element in list`
  - That would be clearer right?
- **However our ADT cannot generate this**
  - Yet! Once we look at iterators we will be able to
- **So for now we need to keep using indices (as you would have done in C)**

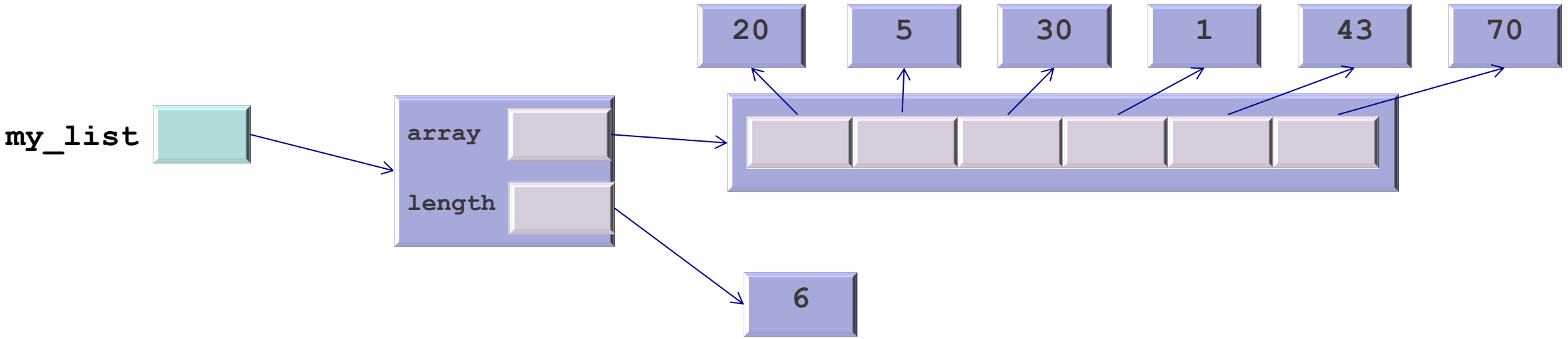
```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Assume, equivalent to `[20,5,30,1,43,70]`

Callee

Caller



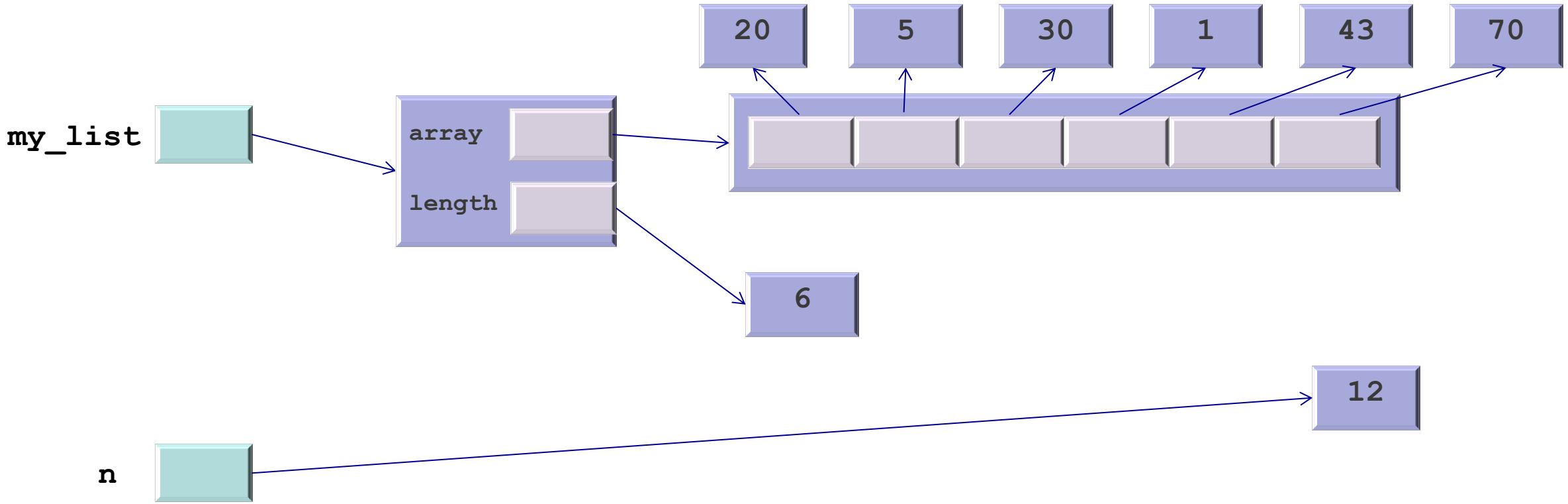
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



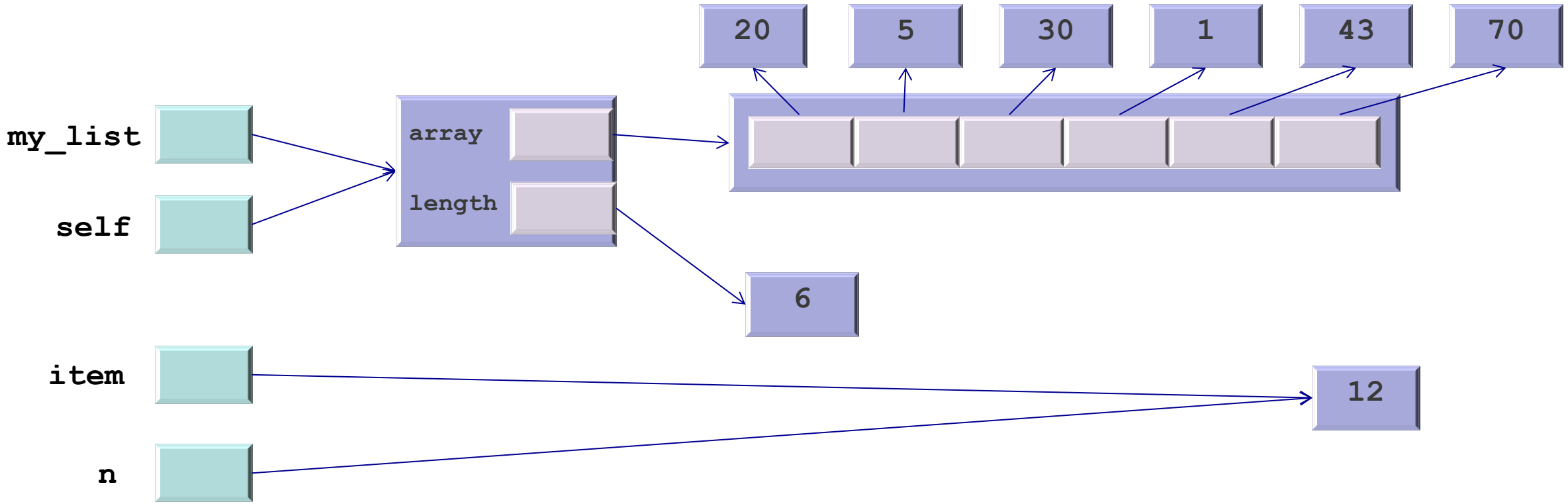
Assume, equivalent to
[20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



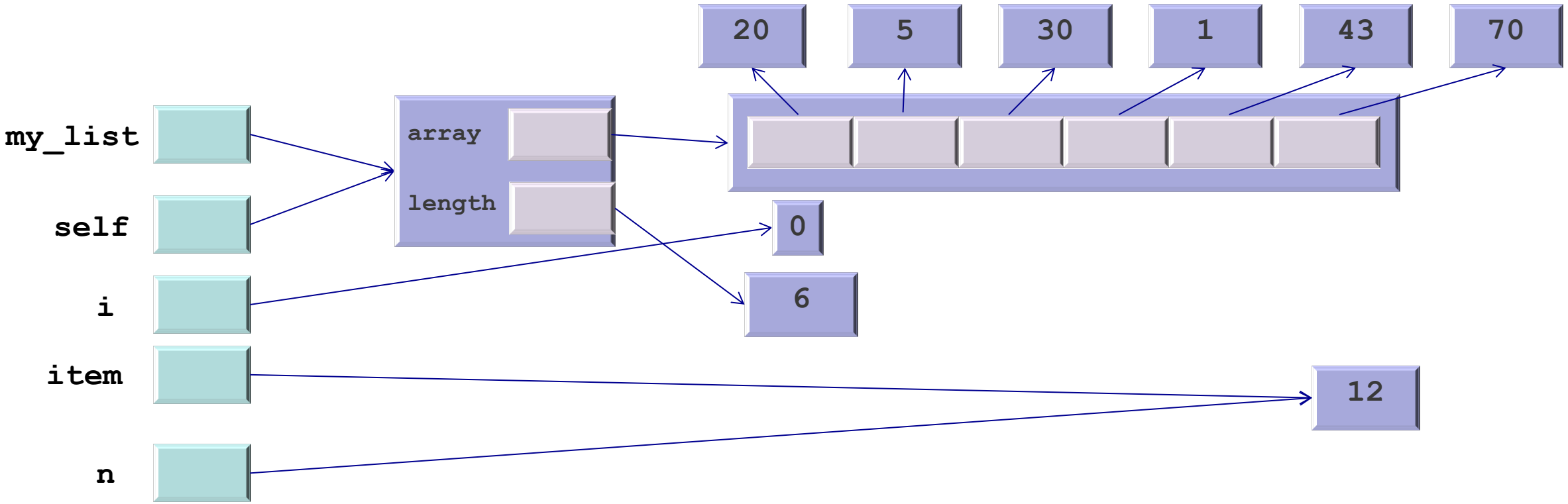
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



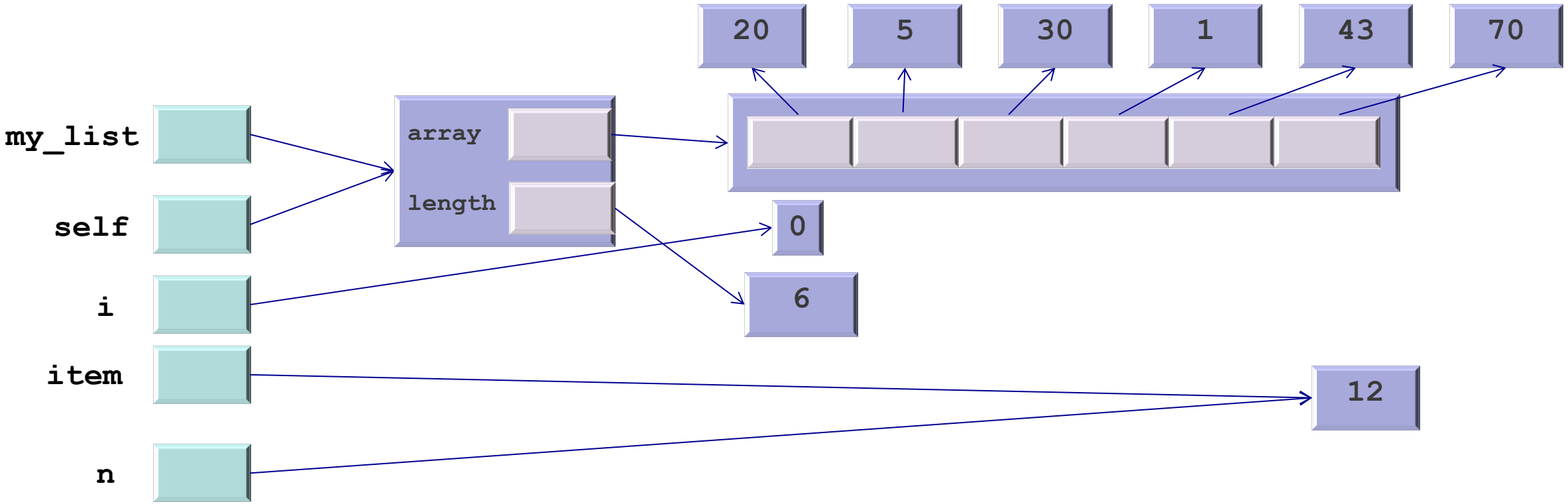
Assume, equivalent to
[20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



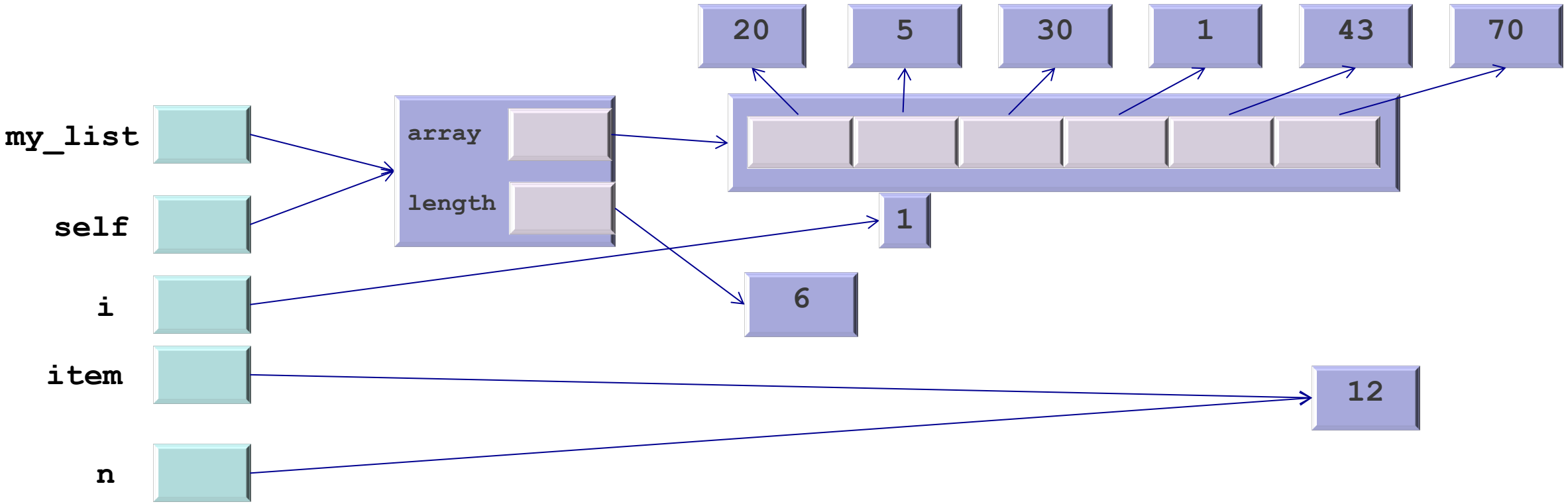
Assume, equivalent to
[20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



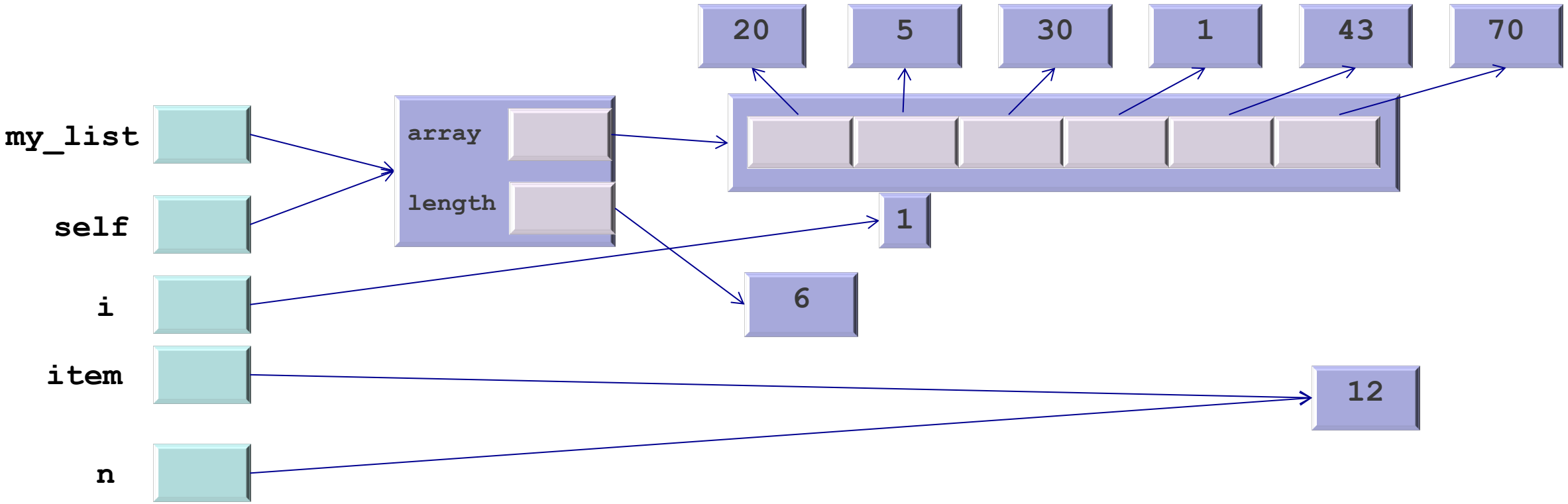
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



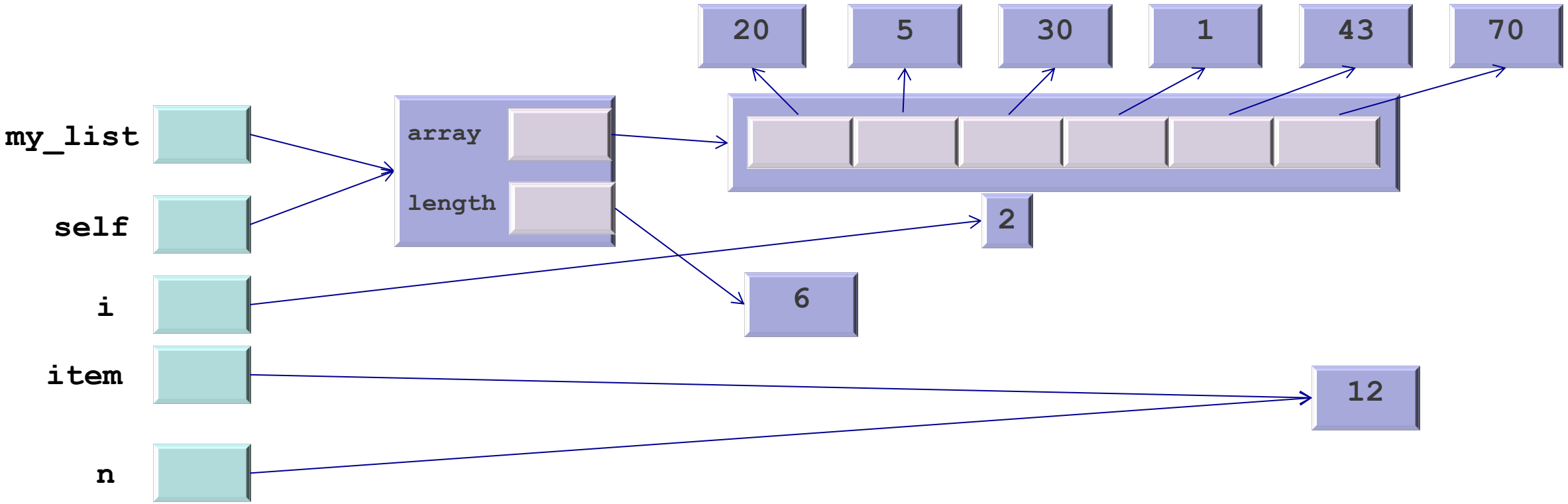
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



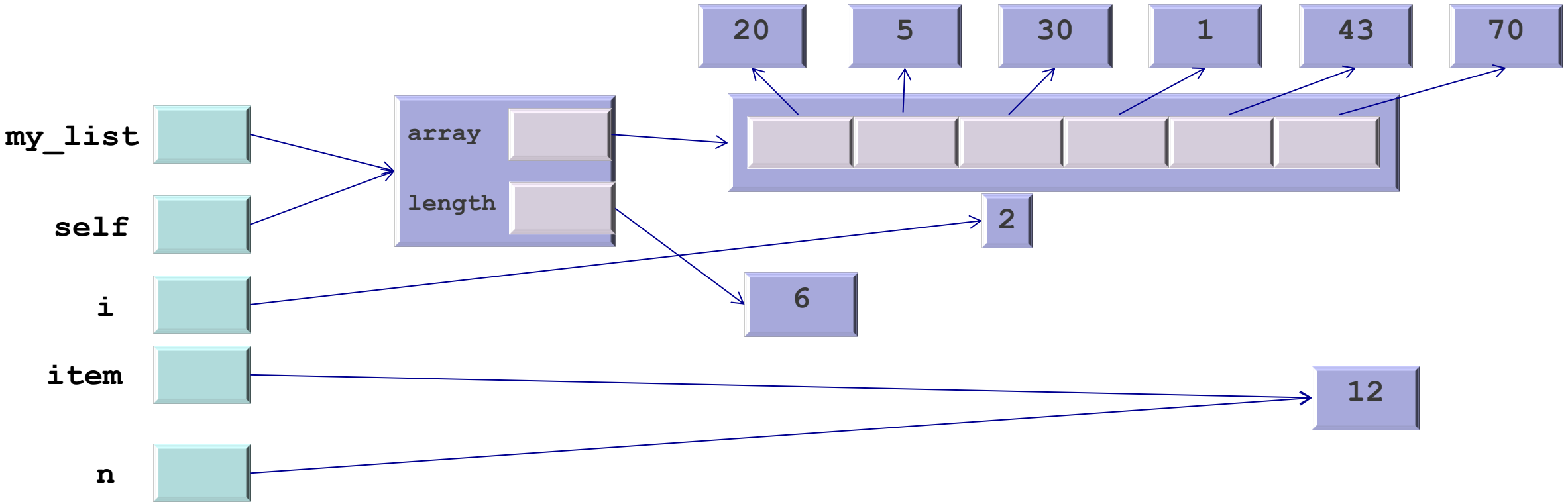
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



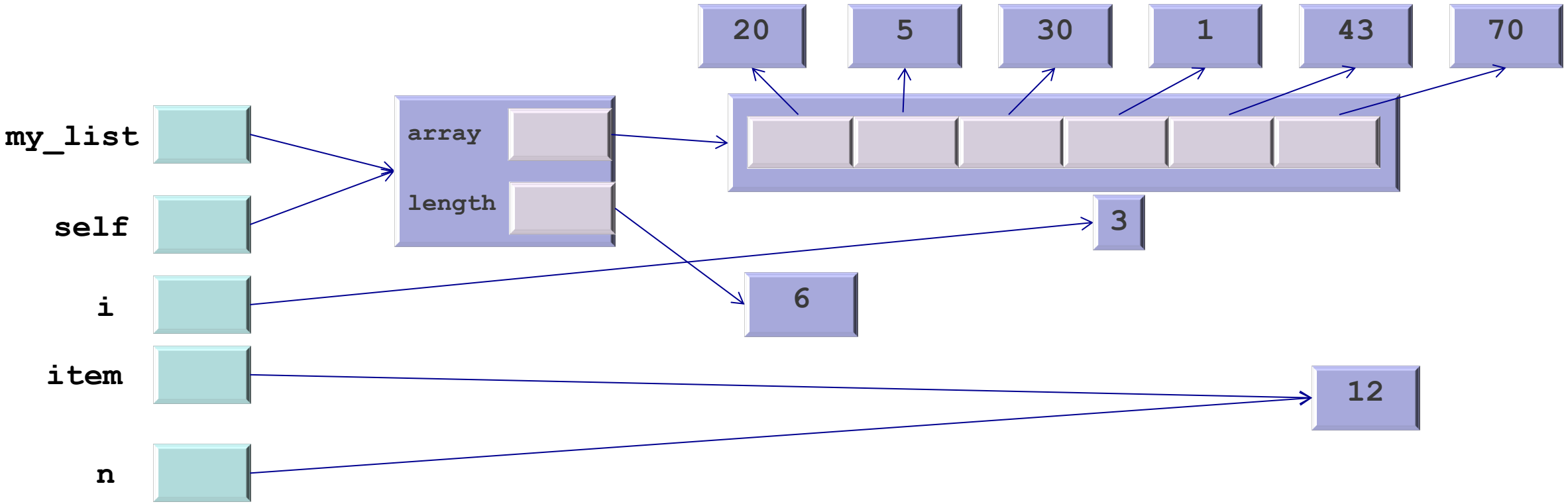
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



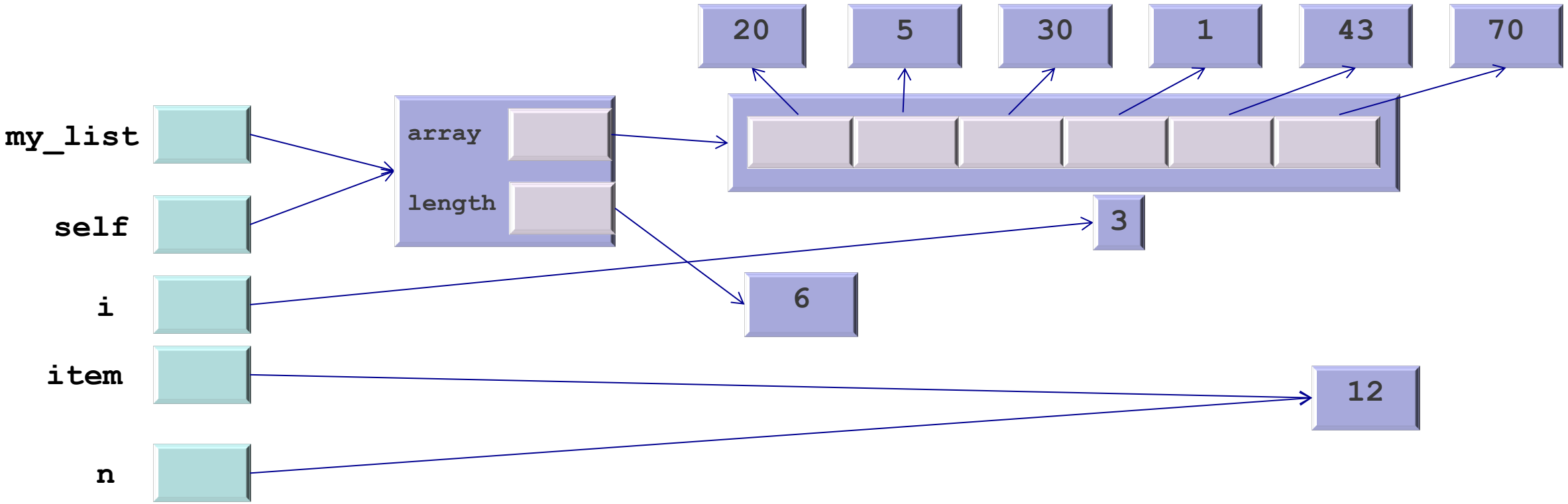
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



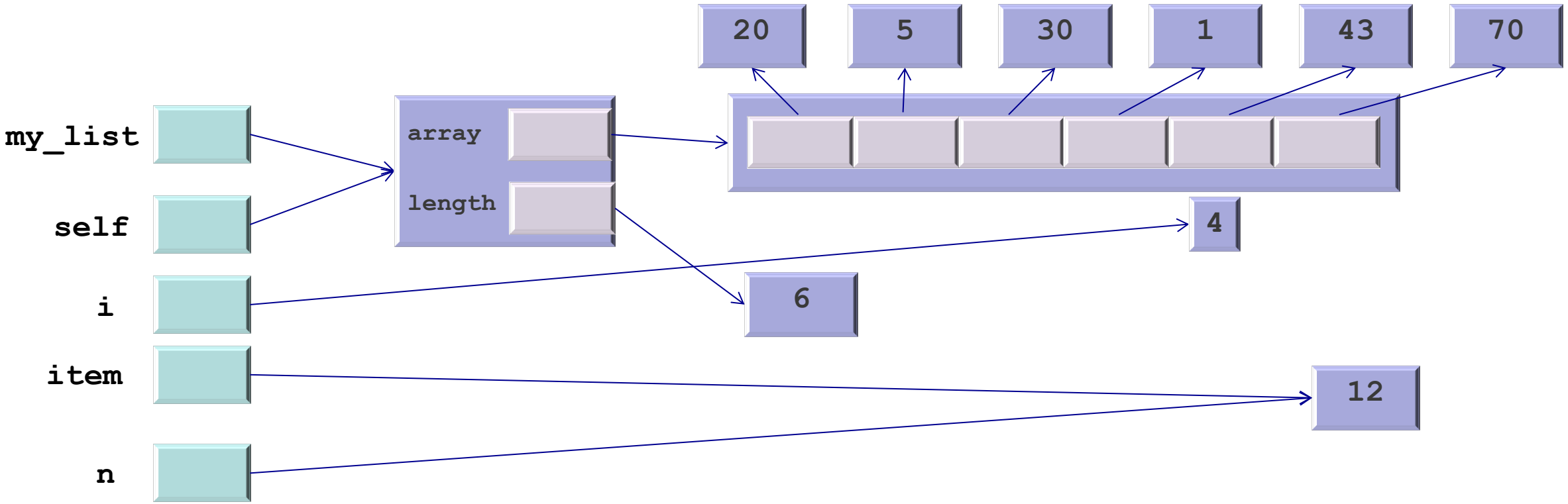
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



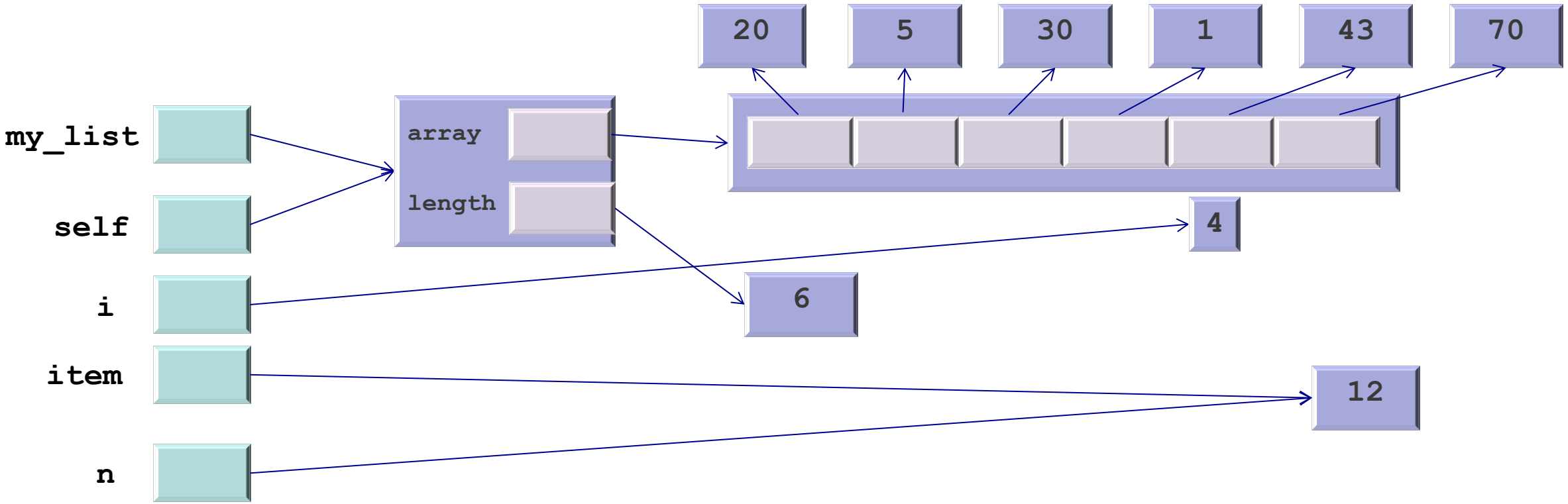
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



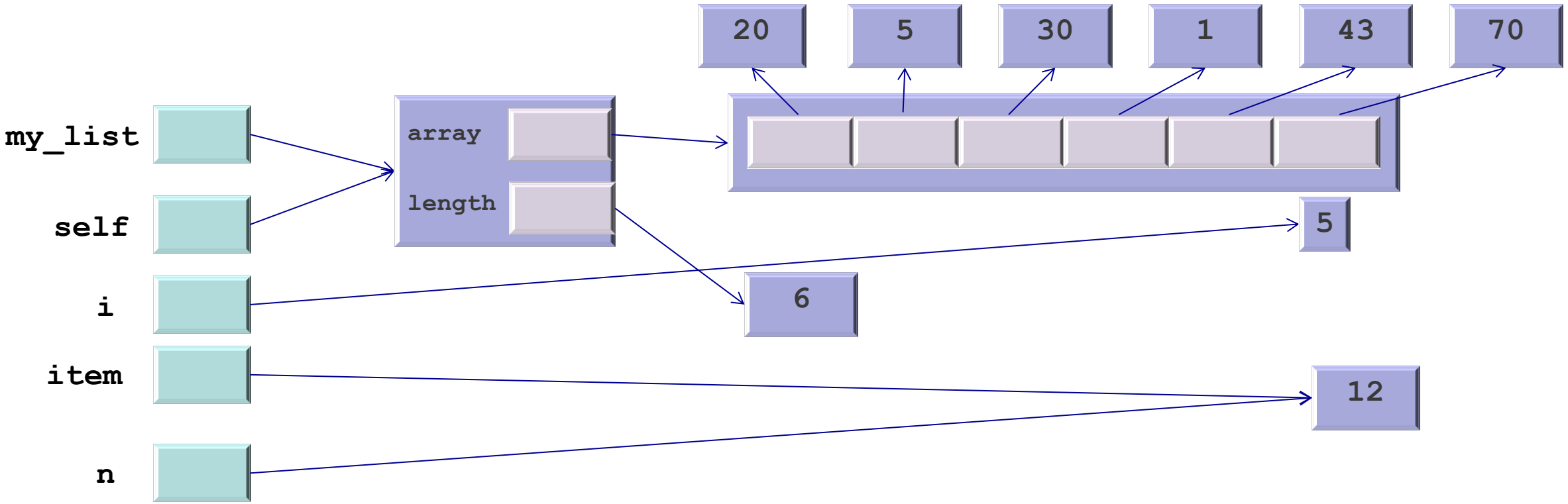
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Callee

Caller



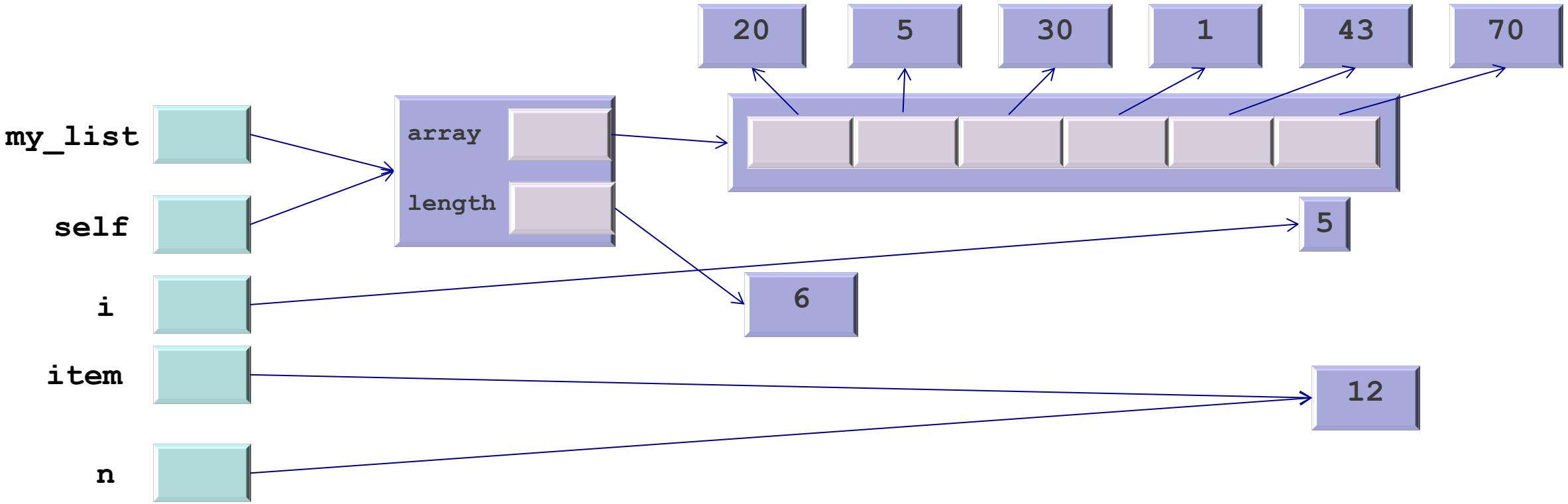
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



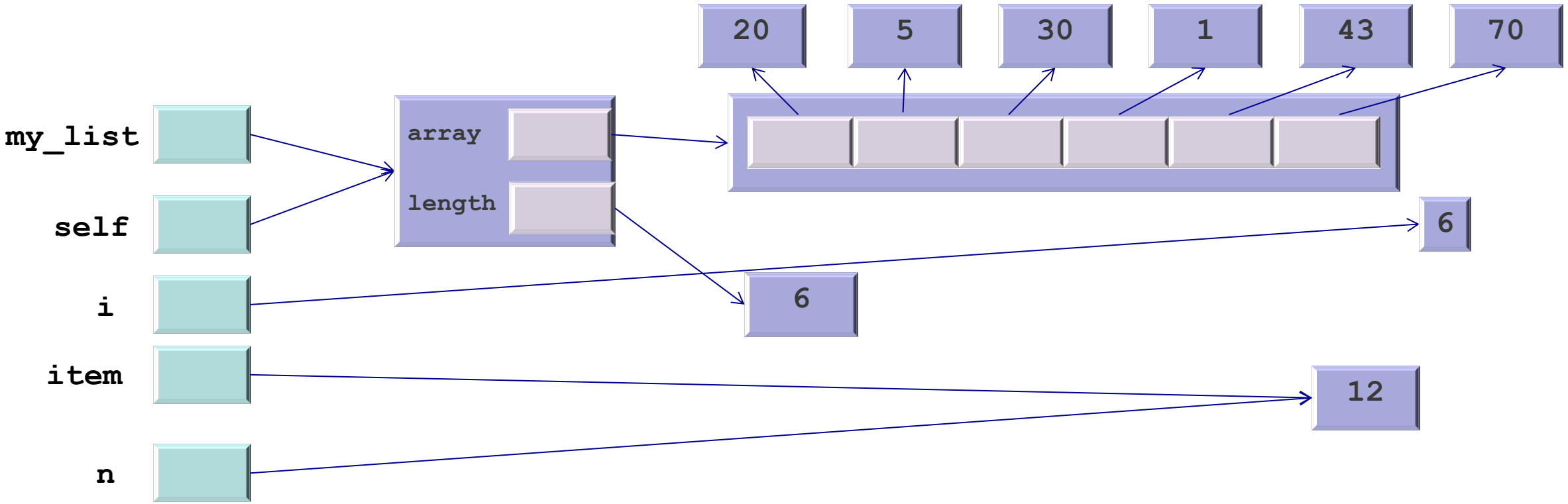
Assume, equivalent to
`[20,5,30,1,43,70]`

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



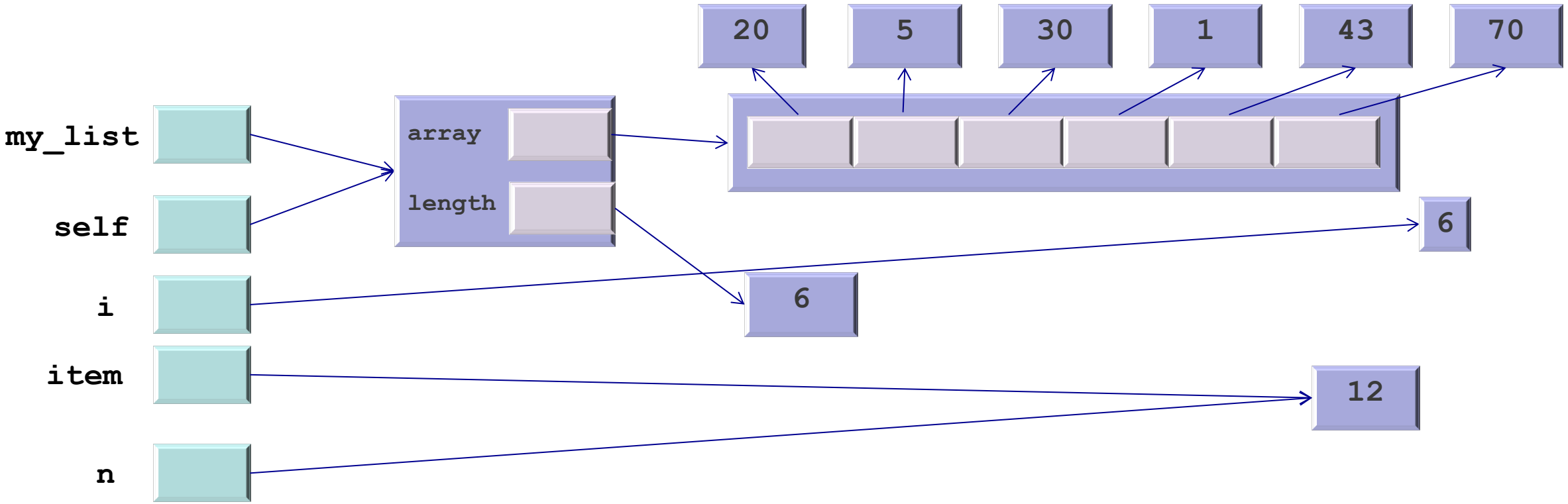
Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller



Assume, equivalent to [20,5,30,1,43,70]

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list[0] = 20; … ; my_list[5] = 70
n = 12
my_list.index(n)
```

Caller

Big O Time Complexity

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
    raise ValueError("item not in list")
```

? times

Access is constant

Comparison we don't know m

Return is constant

Raising is not part of Big O

We say comparison is O(m), where again m depends on the size of what you are comparing. For integers m = constant number of bits. For strings m is the length of the string. For an array is the length of the array multiplied by the size of its elements, and so on. If we want to be more general, use $Comp_{==}$

Best ≠ Worst

Some elements get a certain amount of processing

Other elements are not processed at all

Big O Time Complexity for index

This affects the Big O of method **`remove`**: it calls **`index`**

- **Best case?**
  - Loop stops in the first iteration
  - When the item we are looking for is at the start of the list
    - constant + m + constant → O(m) -- or O($Comp_{==}$) if we use the general form
- **Worst case?**
  - Loop goes all the way (len(self) times)
  - When the item we are looking for is at the end of the list
    - len(self)*(constant + m) +constant → O(len(self)*m) – or O(len(self)*$Comp_{==}$)

```
def index(self, item: T) -> int:
    for i in range(len(self)):          Length access is constant
        if item == self.array[i]:       Comparison is O(m)
            return i                    Return is constant
    raise ValueError("item not in list") irrelevant
```

? times

MONASH University

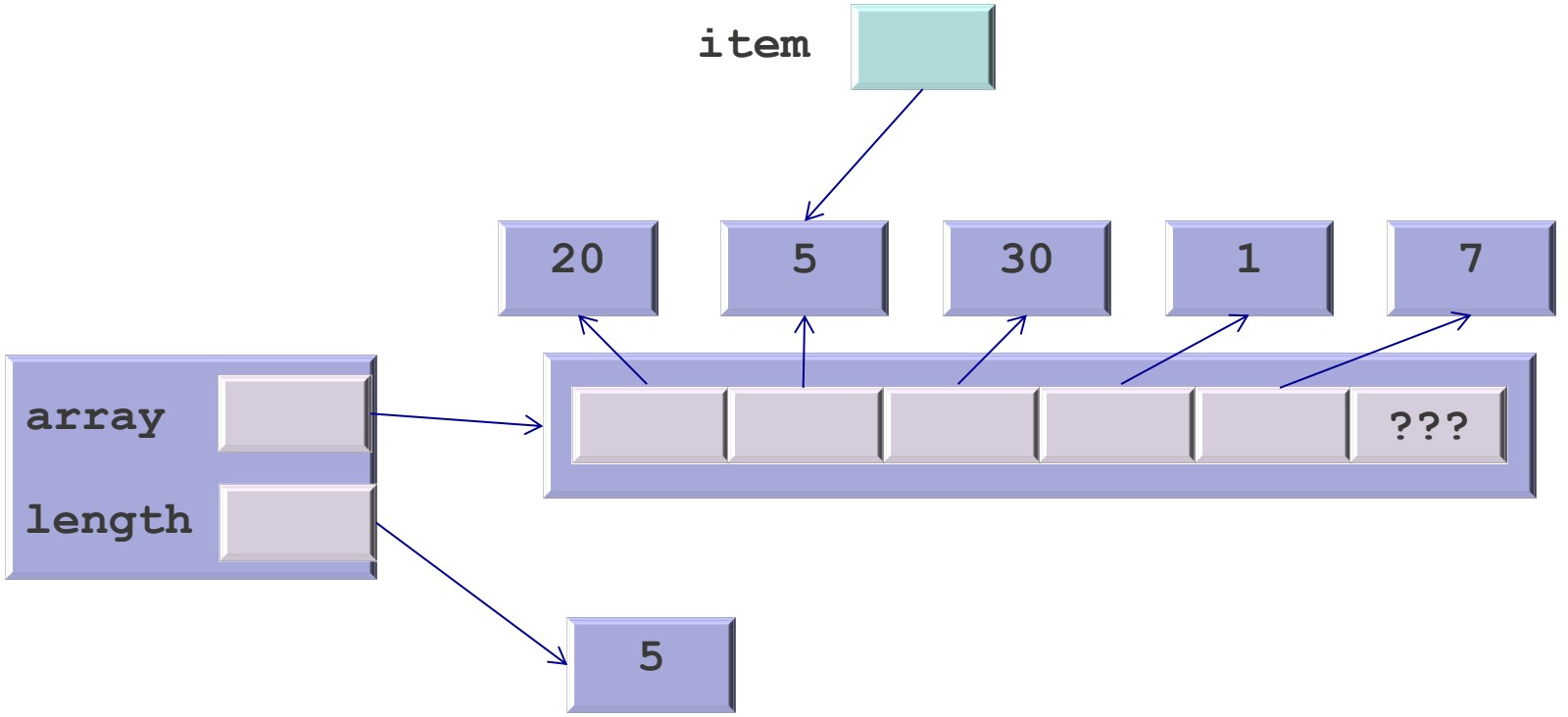
List ADT implemented with arrays, part II

Time for implementing `delete_at_index`

- **What exactly do we want to do?**
  - Given a list and the index of position P of the item to be deleted
  - Finish with a list that:
    - Has exactly the same elements as before
    - EXCEPT for the item that was in P, which is now not in the list
  - And return that item
- **This is a little bit vague…**
  - What if the position of index P is not in the list? (<0 or >= len(self))
    - Raise an `IndexError`
  - Do the non-removed elements need to maintain their relative order?
    - Yes! since the elements in a list are meant to be "ordered" (not sorted, but ordered)
    - If we didn't care: simply move the last element to the position of the deleted element

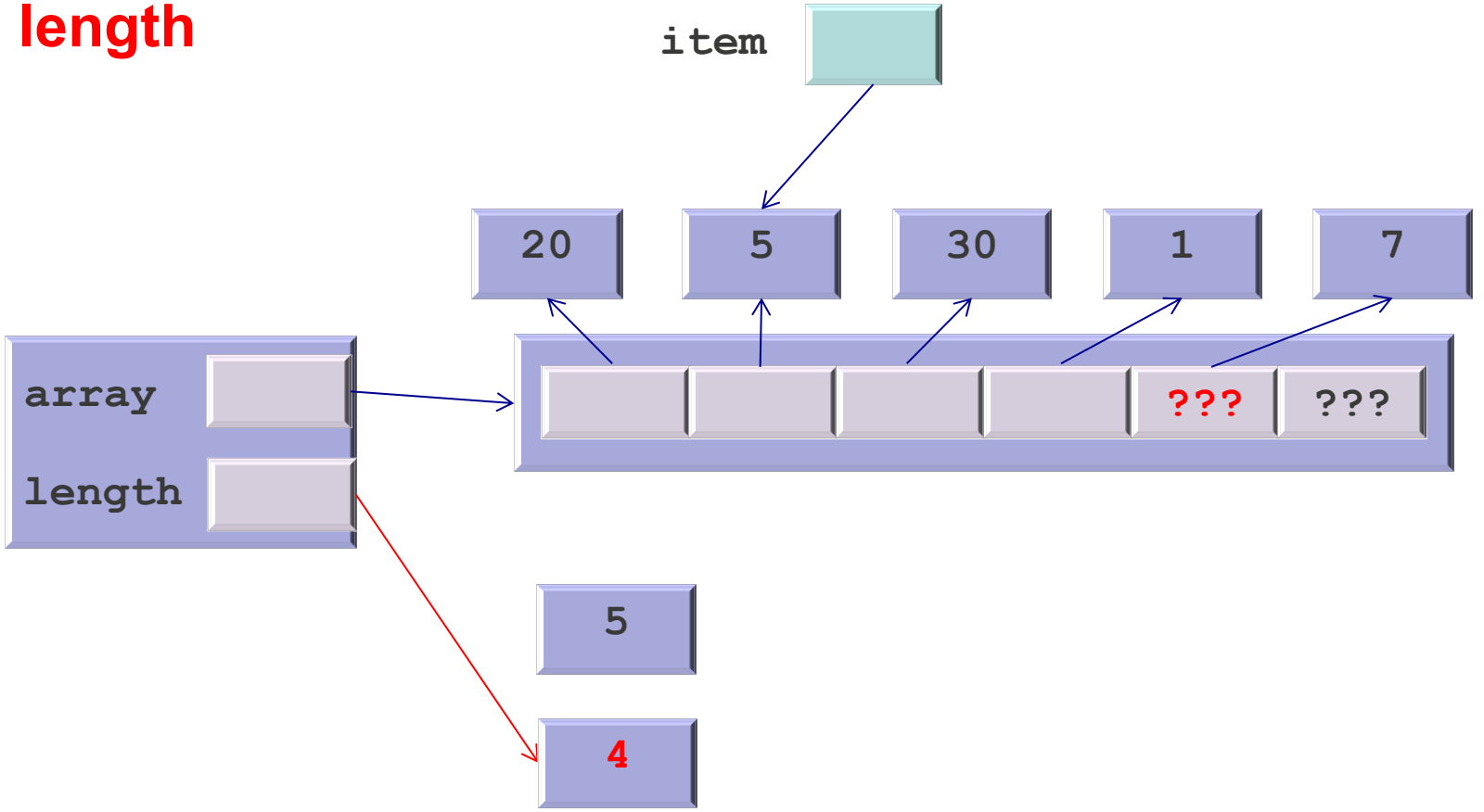
Deleting and returning the element in position 1

- **Put the element in position 1 in a temporary variable (say `item`)**



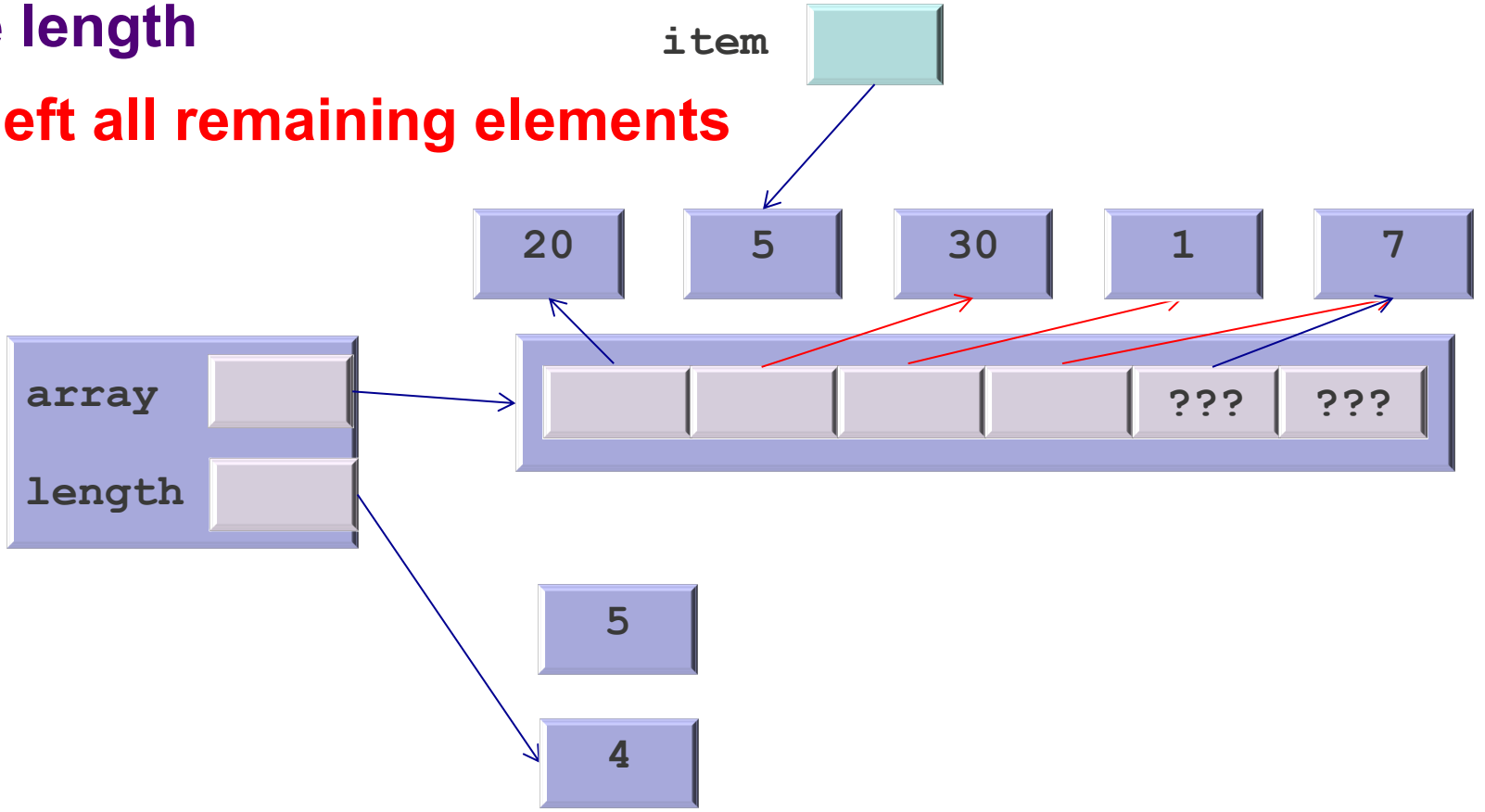
Deleting and returning the element in position 1

- Put the element in position 1 in a temporary variable (say `item`)
- Decrement the length



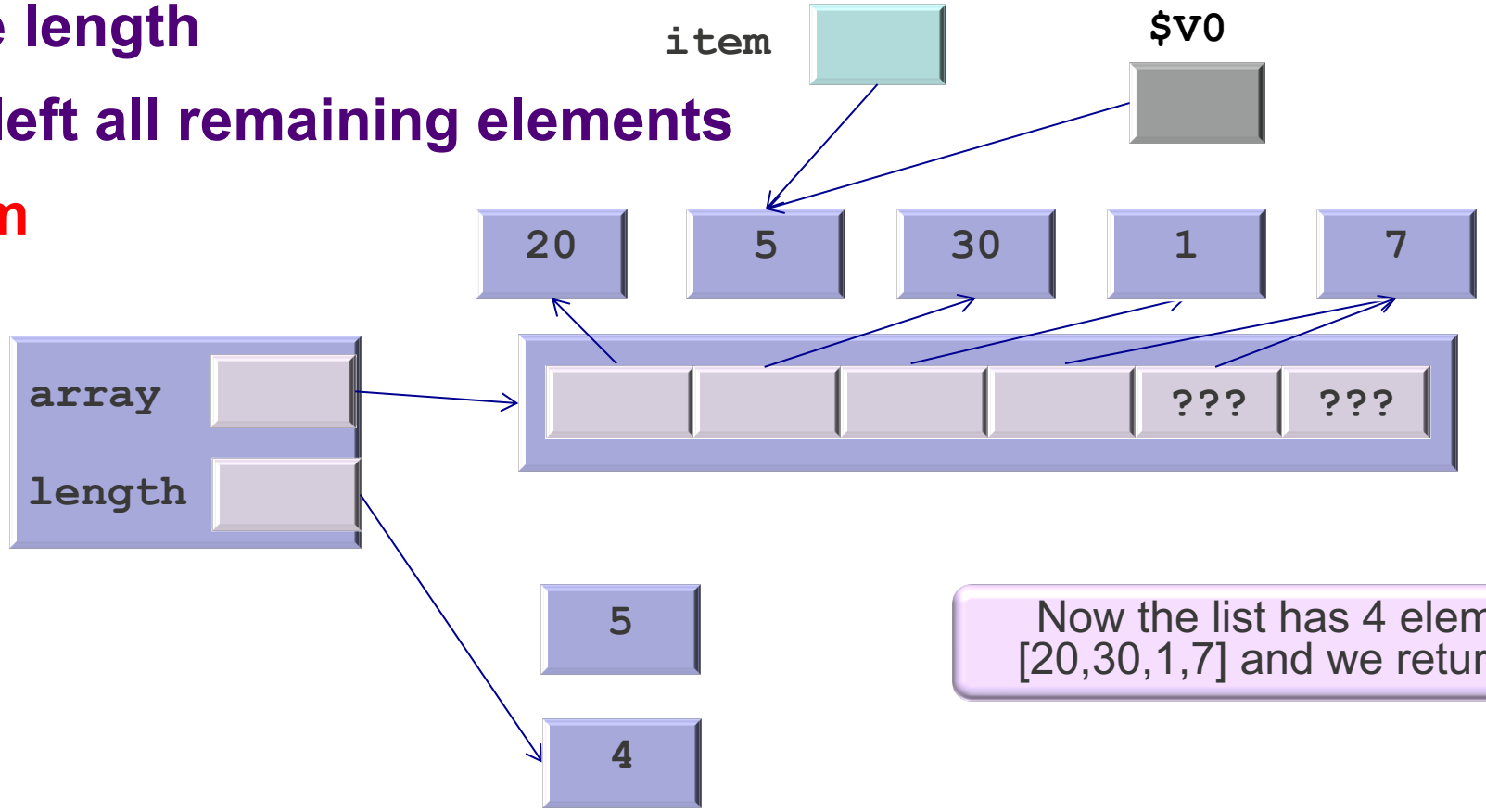
Deleting and returning the element in position 1

- Put the element in position 1 in a temporary variable (say `item`)
- Decrement the length
- Shuffle to the left all remaining elements



Deleting and returning the element in position 1

- Put the element in position 1 in a temporary variable (say `item`)
- Decrement the length
- Shuffle to the left all remaining elements
- Return the item



Now the list has 4 elements [20,30,1,7] and we returned 5

Method `delete_at_index`

- **A possible implementation:**

```
def delete_at_index(self, index: int) -> T:
    item = self.array[index]
    self.length -= 1
    for i in range(index,self.length):
        self.array[i] = self.array[i+1]
```

When the loop finishes, the item that was in `self.length` is now copied to the left, and that position is now "empty". Thus, the invariant holds again.

- **Wait! What about the pre-condition!**
  - If the element is not there, `index` already raises the `IndexError`
  - So that's OK, no point in handling it to just throwing it again!
- **And what about Big O complexity?**
  - Best case: `index` is the last position. Thus, the loop does not start, so O(1)
  - Worst case: `index` is the head. Thus the loop runs fully, so O(len(self))

What about the Big O complexity of `remove`?

- **This method was defined as part of the abstract class as:**

```
def remove(self, item: T) -> None:
    index = self.index(item)
    self.delete_at_index(index)
```

- **We said that the Big O of:**
  - Index:
    - Best O($Comp_{==}$): if **`item`** is found at the head of the list
    - Worst O(len(self)*$Comp_{==}$): if **`item`** is found last
  - Delete at index:
    - Best case: O(1) when **`index`** is last
    - Worst case: O(len(self)) when **`index`** is the head of the list
- **The best case of one is the worst case of the other**

If element is the head, we get $O(Comp_{==} + len(self))$

If element is the last, we get $O(Comp_{==} * len(self))$

MONASH University

List ADT implemented with arrays, part III

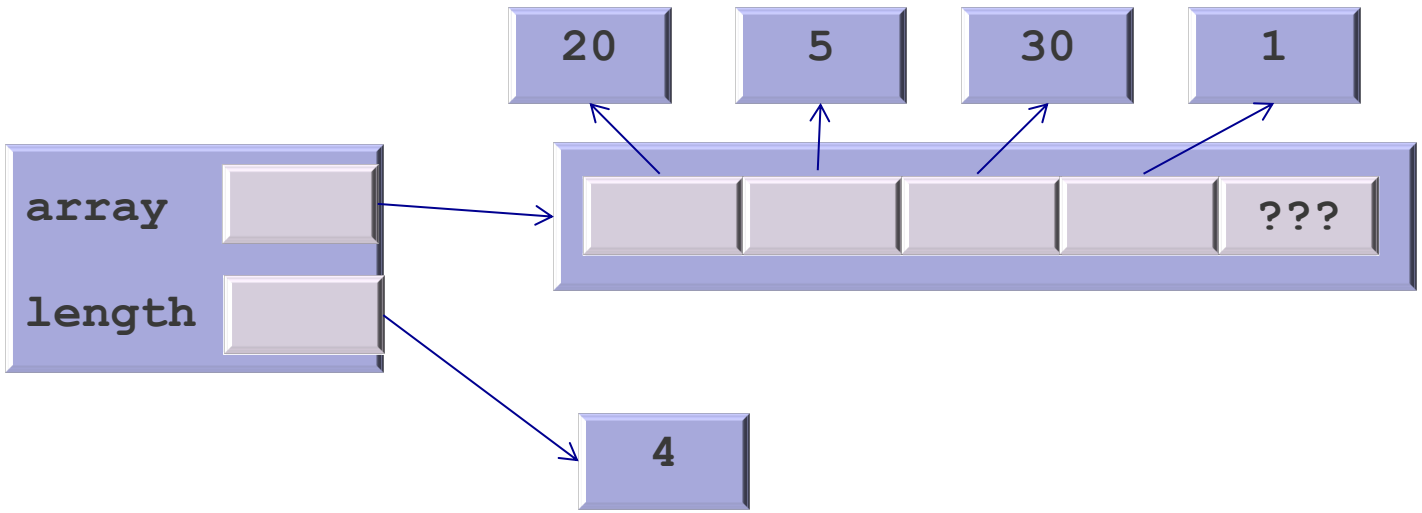
Time for implementing append

- **What exactly do we want to do?**
  - Given a list and the item to be appended
  - Finish with a list that:
    - Has exactly the same elements as before
    - And has the new item as the last elements in the list
- **This is a little bit vague…**
  - The original elements must maintain their relative order
  - If the array is full:
    - We could raise an exception (as we did for Stacks and Queues)
    - Instead: we are going to increase the size of the array (for now, double it)

This is why there is no `is_full` method

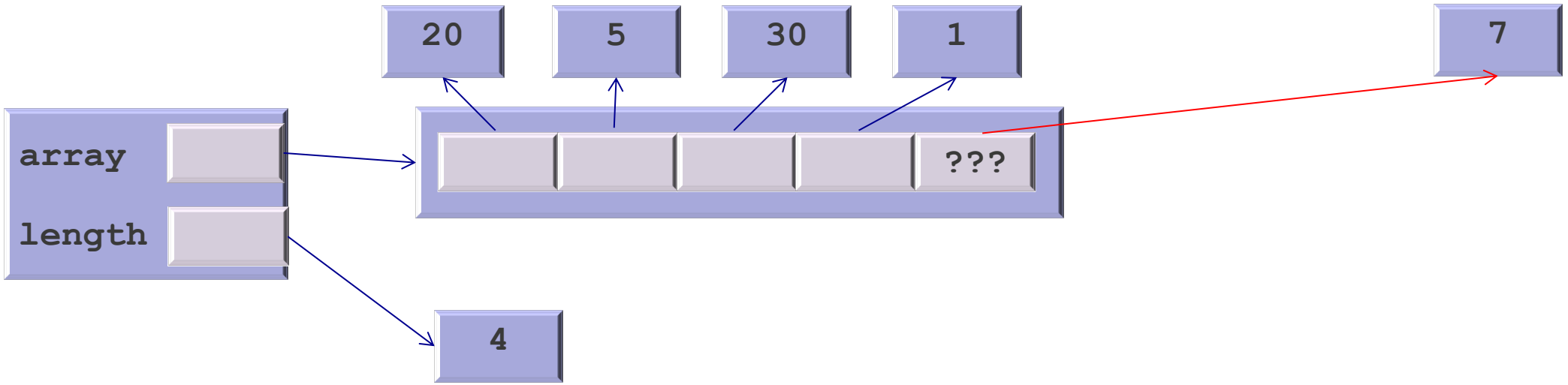
Appending an element to a list: an example

- **Check whether the array is full**



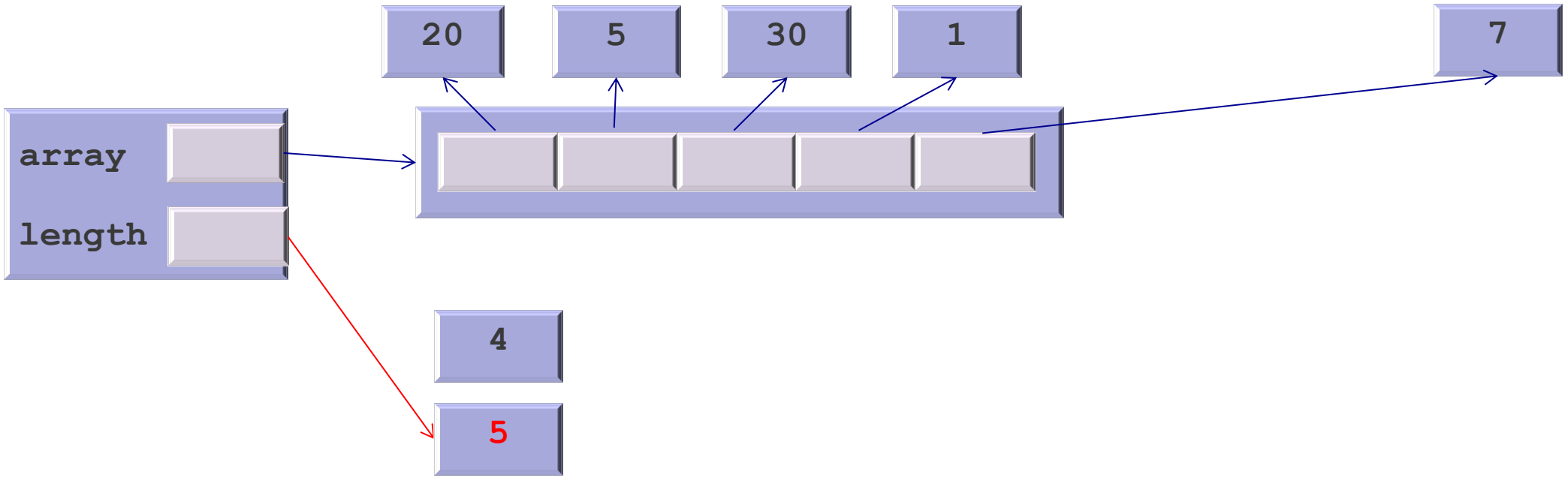
Appending an element to a list: an example

- Check whether the array is full
- If not, assign the item to be inserted (say 7) to the first empty cell



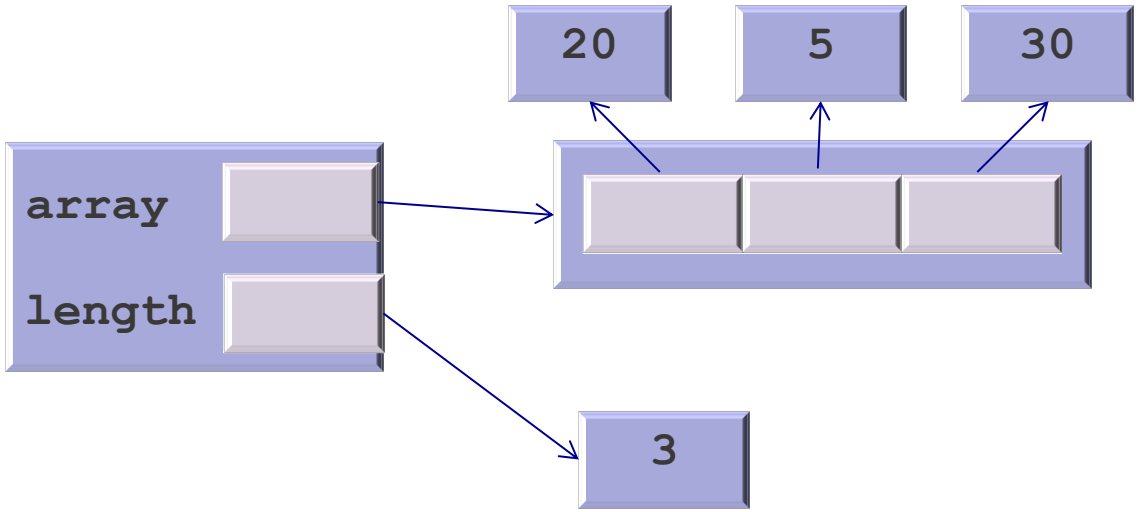
Appending an element to a list: an example

- Check whether the array is full
- If not, assign the item to be inserted (say 7) to the first empty cell
- Increment length



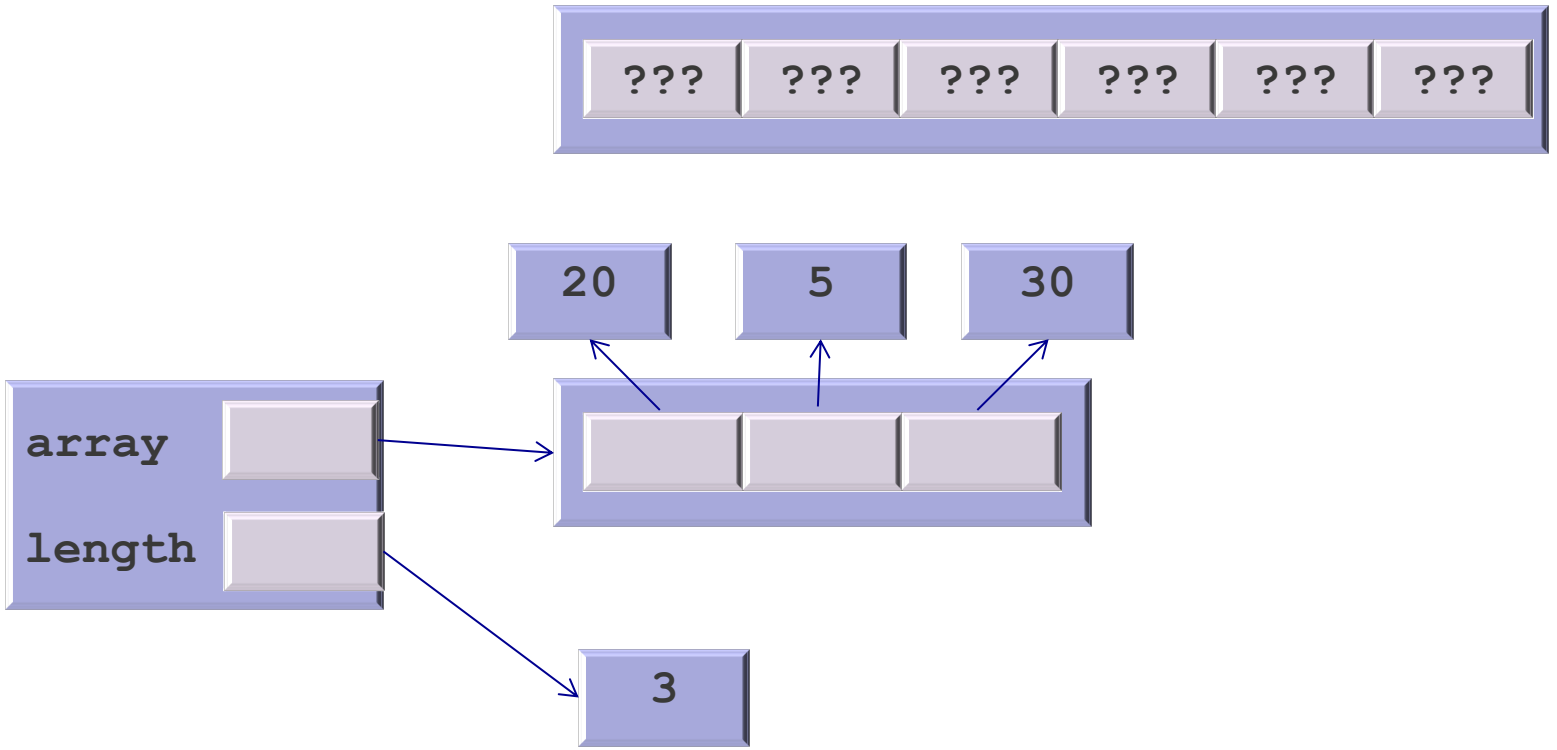
Appending an element to a list: another example

- **Check whether the array is full**



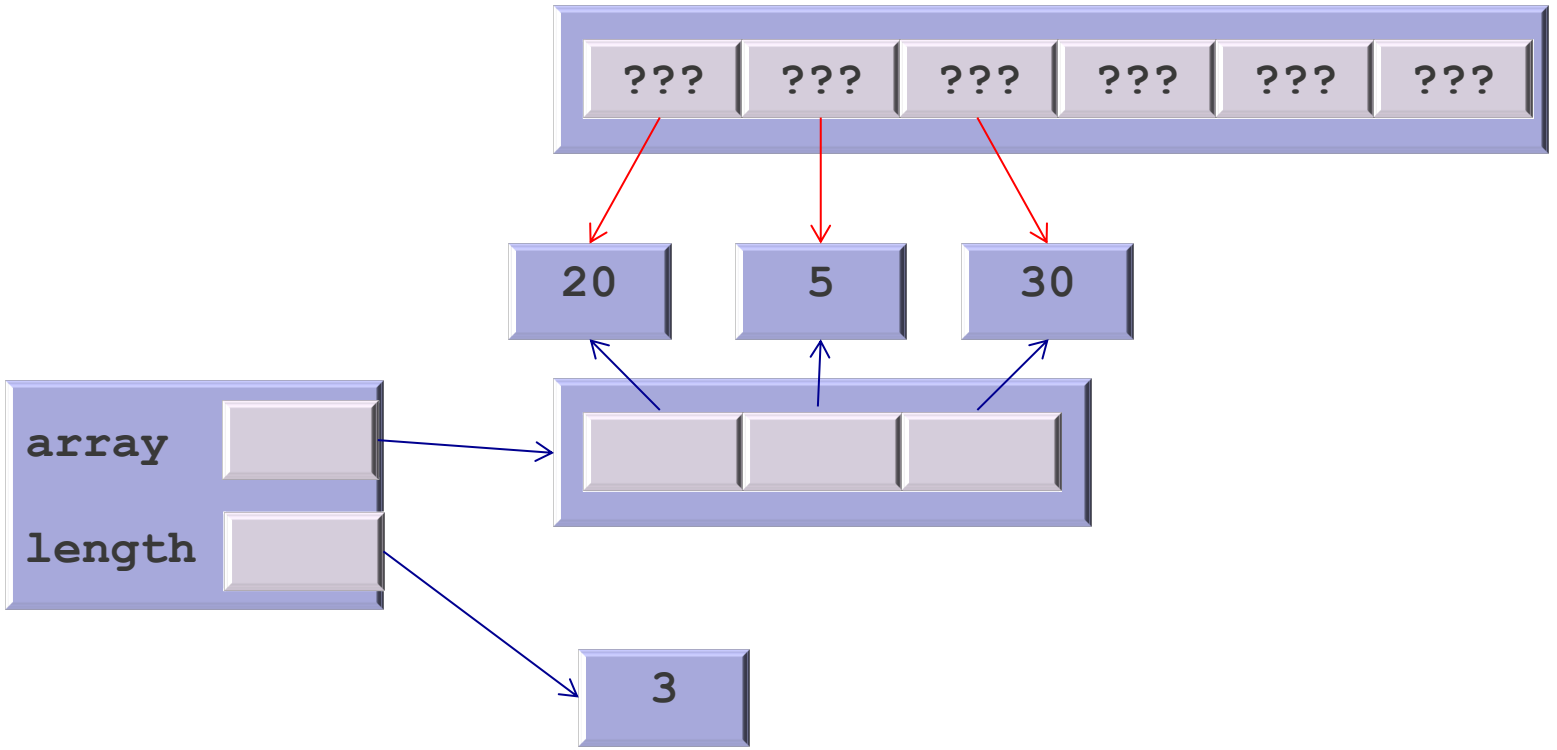
Appending an element to a list: another example

- Check whether the array is full
- If it is, create a new array with more capacity (say double)



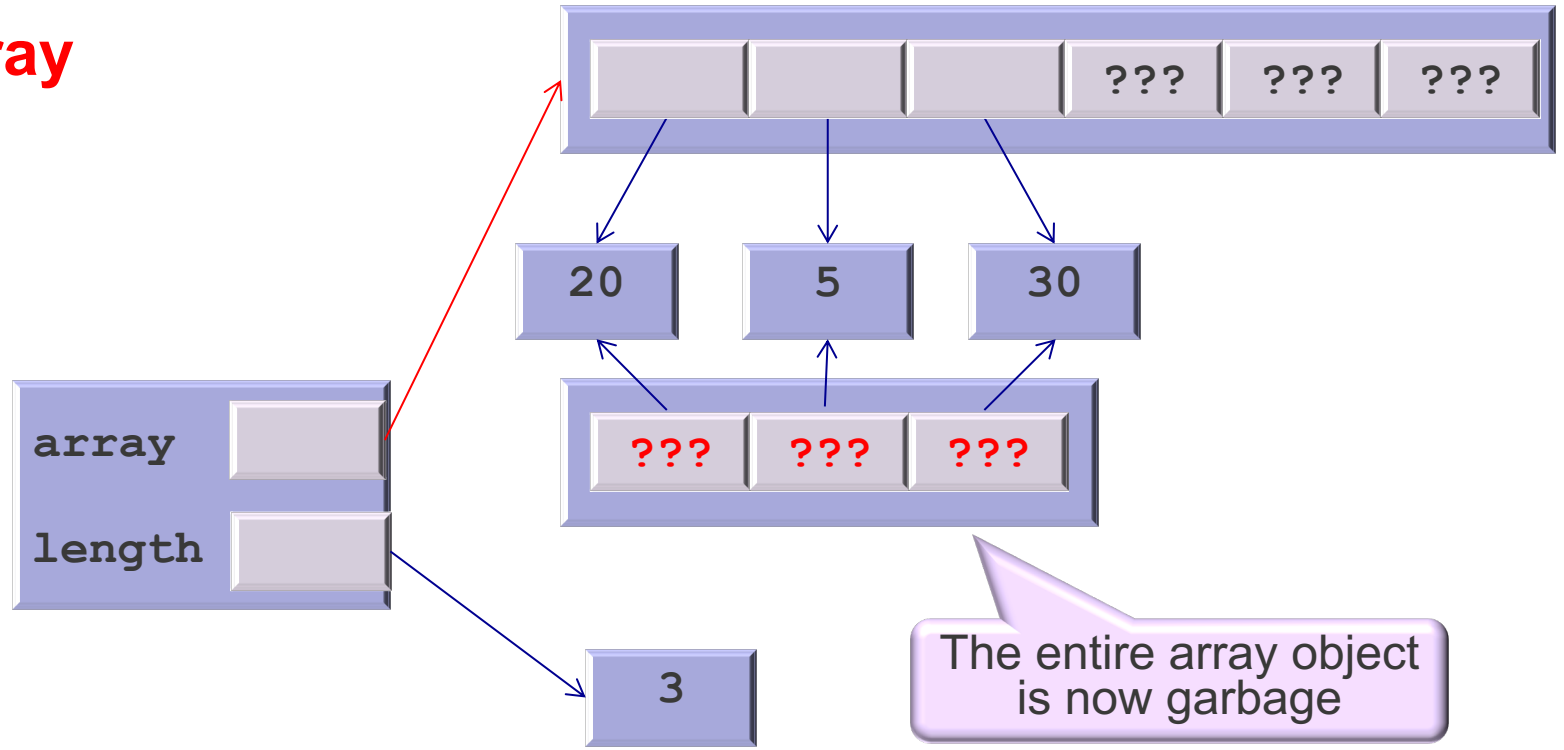
Appending an element to a list: another example

- Check whether the array is full
- If it is, create a new array with more capacity (say double)
- Copy the elements across



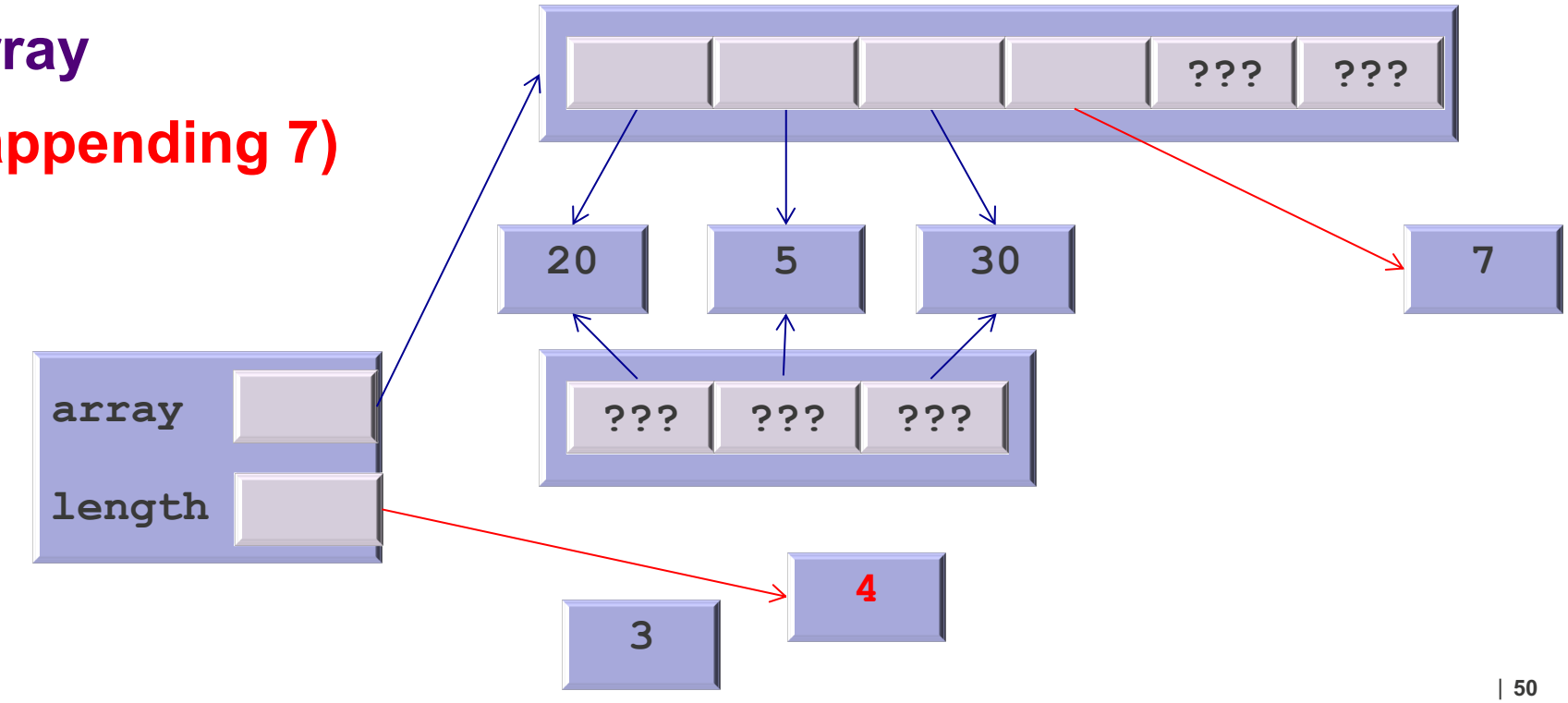
Appending an element to a list: another example

- Check whether the array is full
- If it is, create a new array with more capacity (say double)
- Copy the elements across
- Make this the new array



Appending an element to a list: another example

- Check whether the array is full
- If it is, create a new array with more capacity (say double)
- Copy the elements across
- Make this the new array
- Proceed as before (appending 7)



Method append

- **A possible implementation:**

```
def append(self, item: T) -> None:
    if len(self) == len(self.array):
        new_array = ArrayR(self.__newsize())
        for i in range(len(self)):
            new_array[i] = self.array[i]
        self.array = new_array
    self.array[len(self)] = item
    self.length += 1
```

Resize

Copy

Returns the new capacity, according to whatever formula you want to define

Pyhon uses a growth rate of 1.125 plus a constant, obtaining a growth pattern: 0, 4, 8, 16, 25, 35, 46, 58, 72, 88…

Shouldn't be O(self.__newsize())?

No, since self.__newsize() will return len(self) multiplied by a constant.

- **And what about Big O complexity?**
  - Best case: O(1) when the list is not full
  - Worst case: O(len(self)) when the list is full due to ArrayR and to the loop
    - __newsize should be O(1) since it only does mathematical calculations

Time for implementing insert

- **What exactly do we want to do?**
  - Given a list, the item and the position at which the item should be inserted
  - Finish with a list that:
    - Has exactly the same elements as before
    - And has the new item in the given position
- **Again, a little bit vague…**
  - The original elements must maintain their relative order
  - If the array is full, we must increase the size of the array
- **I will leave this one for you (it simply combines what you have seen before)**
  - Once you do, **`append`** should be redefined to call **`insert`**!

Reuse is your friend!

```
def append(self, item: T) -> None:
    self.insert(len(self), item)
```

Summary

- **We now understand the List ADT:**
  - Main operations
  - Their complexity
- **We are able to:**
  - Implement the List ADT using arrays
  - Modify its operations and
  - Reason about their complexity
- **To be able to decide when it is appropriate to use Stacks, Queues or Lists**

Aside: 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]
```

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]
```