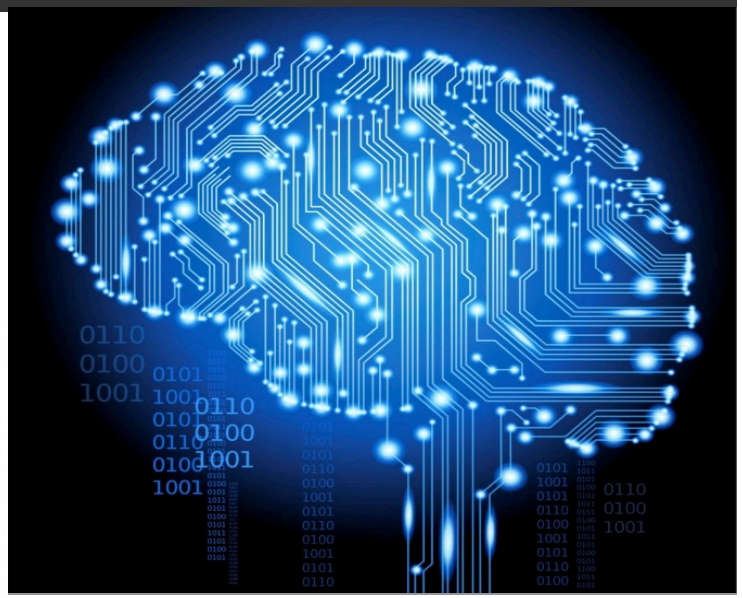
MONASH University

Information Technology

# Sorted List ADT using arrays

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



# Objectives for this lesson

- **Understand the Sorted List ADT:**
  - Main operations
  - Their complexity
- **Be able to:**
  - Implement the SortedList ADT using arrays
  - Modify its operations and
  - Reason about their complexity

MONASH
University

# Sorted List ADT

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

- **A List ADT where all elements are sorted (for us, in increasing order)**
  - That is an additional invariant of the class
- **Should it be a derived class of the List ADT?**
  - Some methods have the same meaning and possibly the same implementation:
    - `__getitem__`, `delete_at_index`, `remove`, `__len__`, `is_empty`, and `clear`
  - Others have the same meaning, but their implementation can be improved:
    - `index`, we will see how later
  - Unfortunately, some have no real meaning in sorted lists:
    - `append`, `insert` and `__setitem__`

So, we need a different kind of `add` method

- **So what do we do?**
- **The best solution is a bit involved for this unit (redefining List, etc) instead:**
  - We will simply create a new abstract class and a derived array one

# Abstract Sorted List class

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

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

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

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

    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
```

Identical to the abstract List class but with **add** and without **__setitem__**, **append**, and **insert**

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

MONASH University

# Sorted List ADT implemented with arrays

# What about SortedArrayList?

- **Again, it is almost identical to the implementation of ArrayList**
  - Which is why re-implementing is not great, but good enough for our unit
- **Let's first focus on the method whose implementation can be improved:**
  - `index`

# Let's think about index in the context of sorted lists

- **Can we use the same implementation as for `ArrayList`?**

```
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")
```

- **Yes! A linear search works for both**
- **Can we do something better…?**
- **Is there any property of a sorted list we can exploit?**
  - **Invariant**: every item is greater or equal than the previous one
- **Can we use this to stop the search earlier?**
  - Yes! If we find an element that is greater than **`item`**
  - We then know **`item` is not in the list**

# A possible implementation

- **One possibility is:**

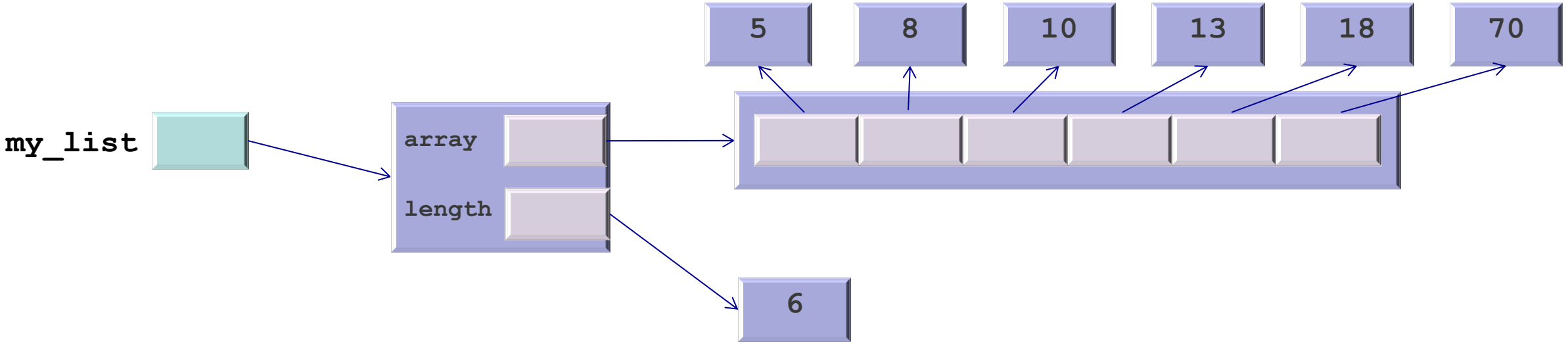
```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Item has been found

Item cannot be in the list

Item is not in the list

- **Let's see why this is better**



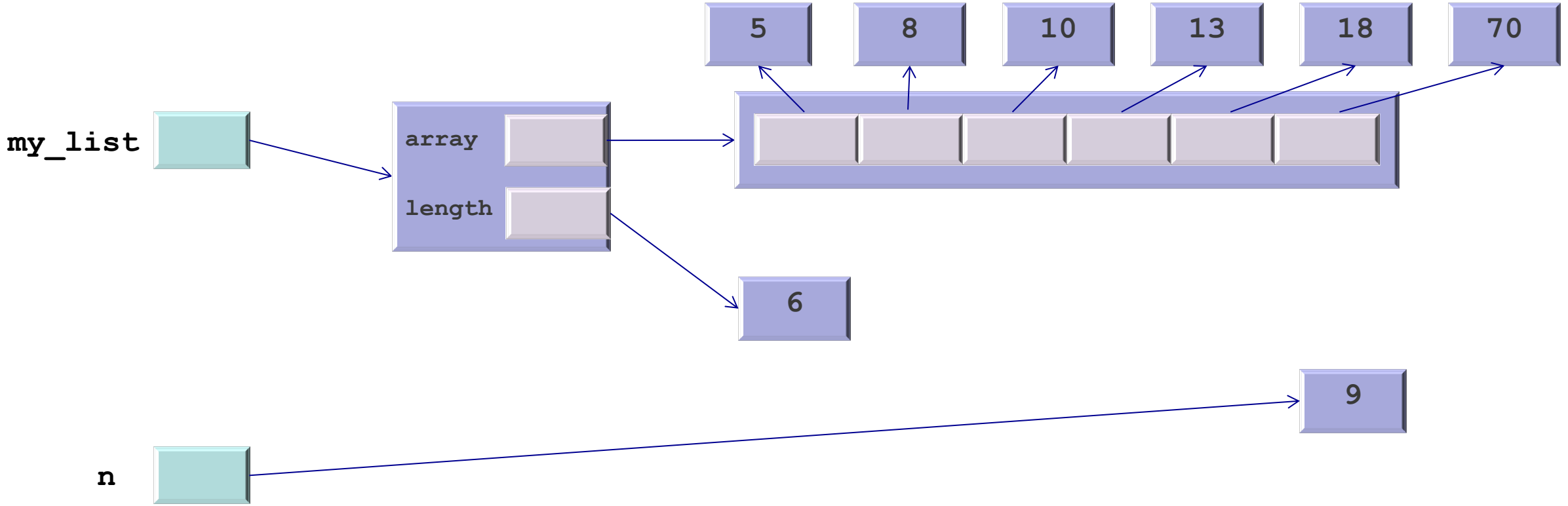

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



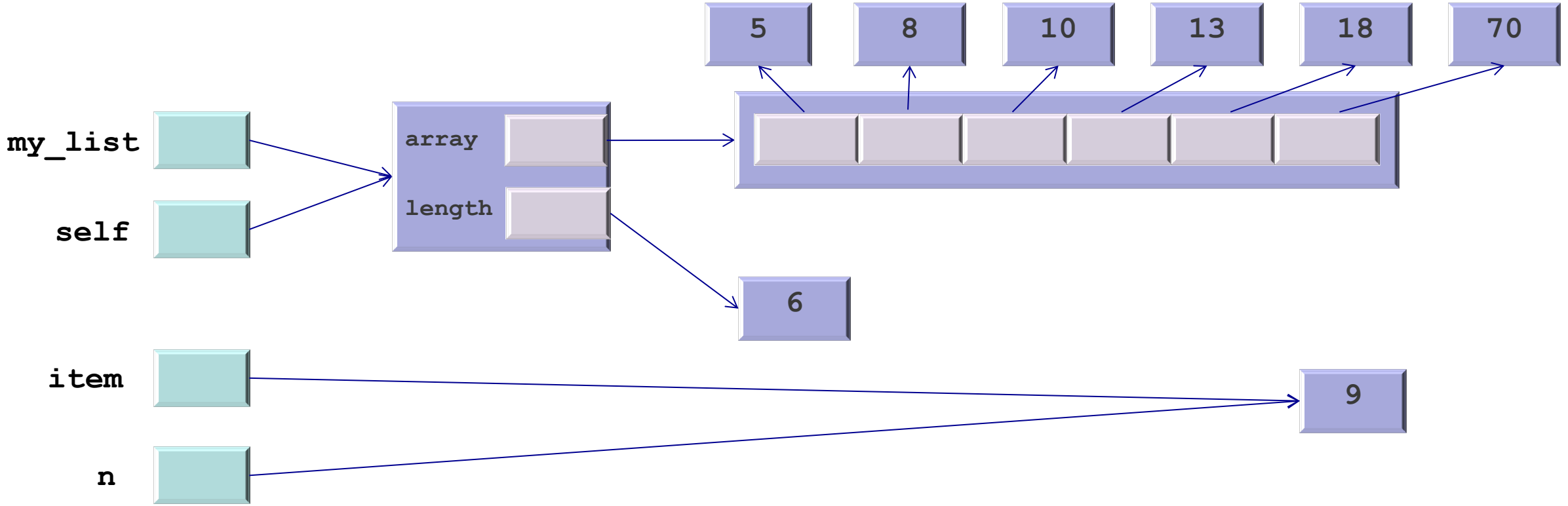

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



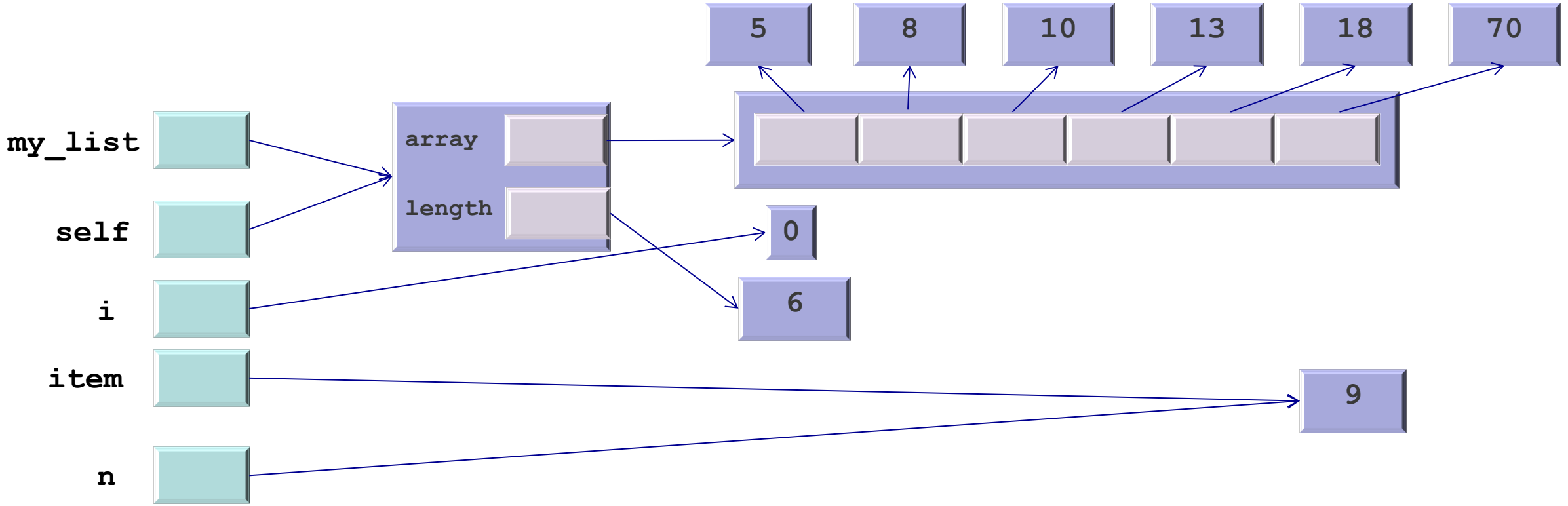

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



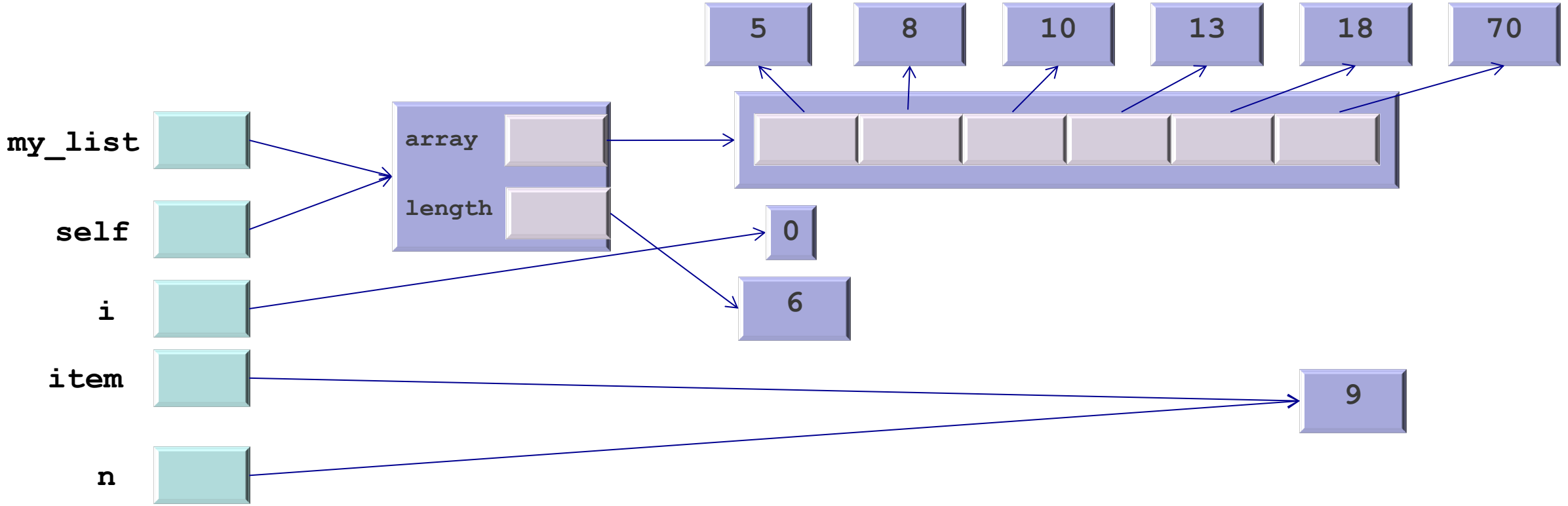

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



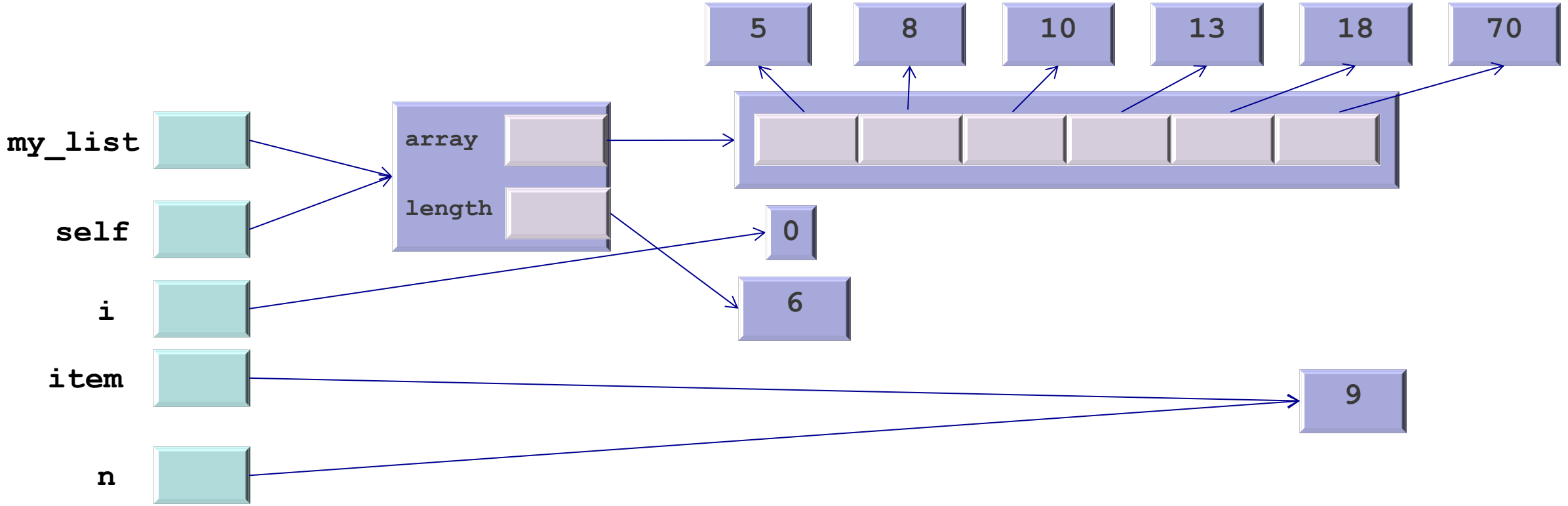

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



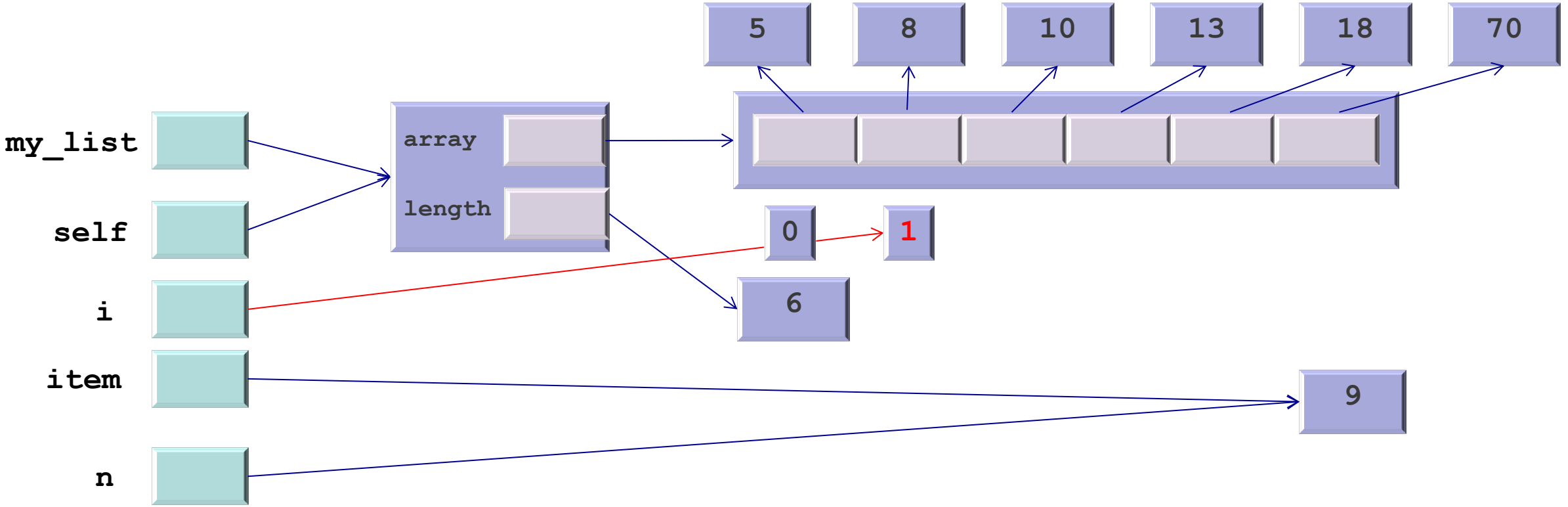

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



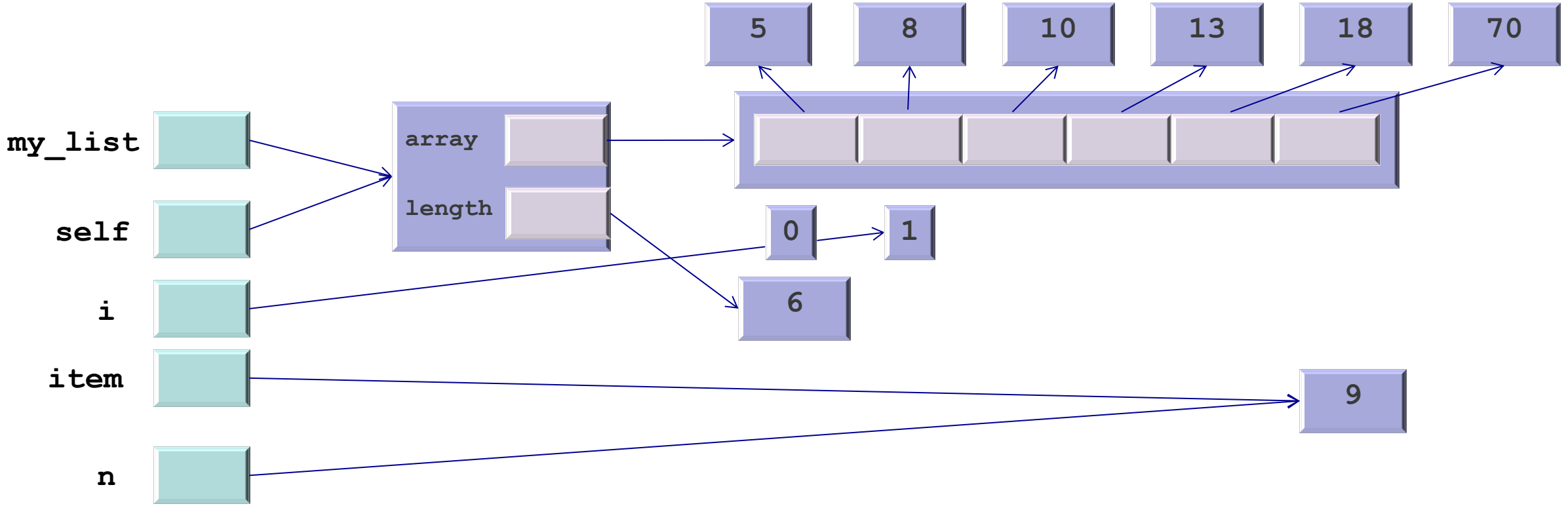

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



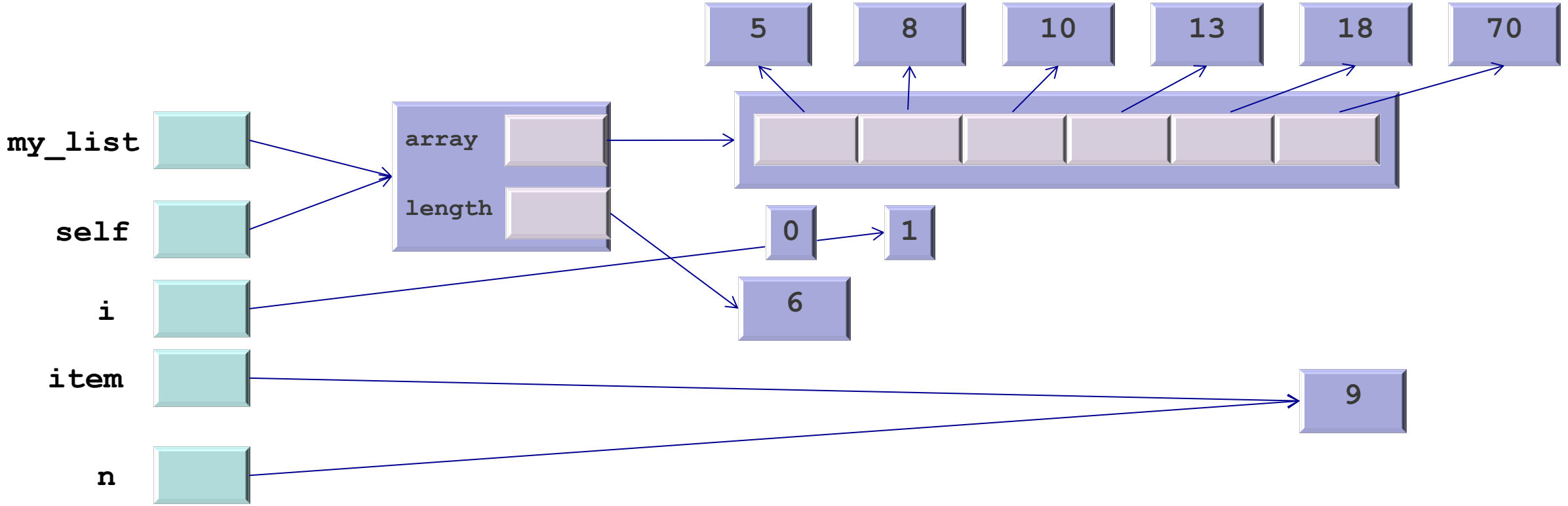

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



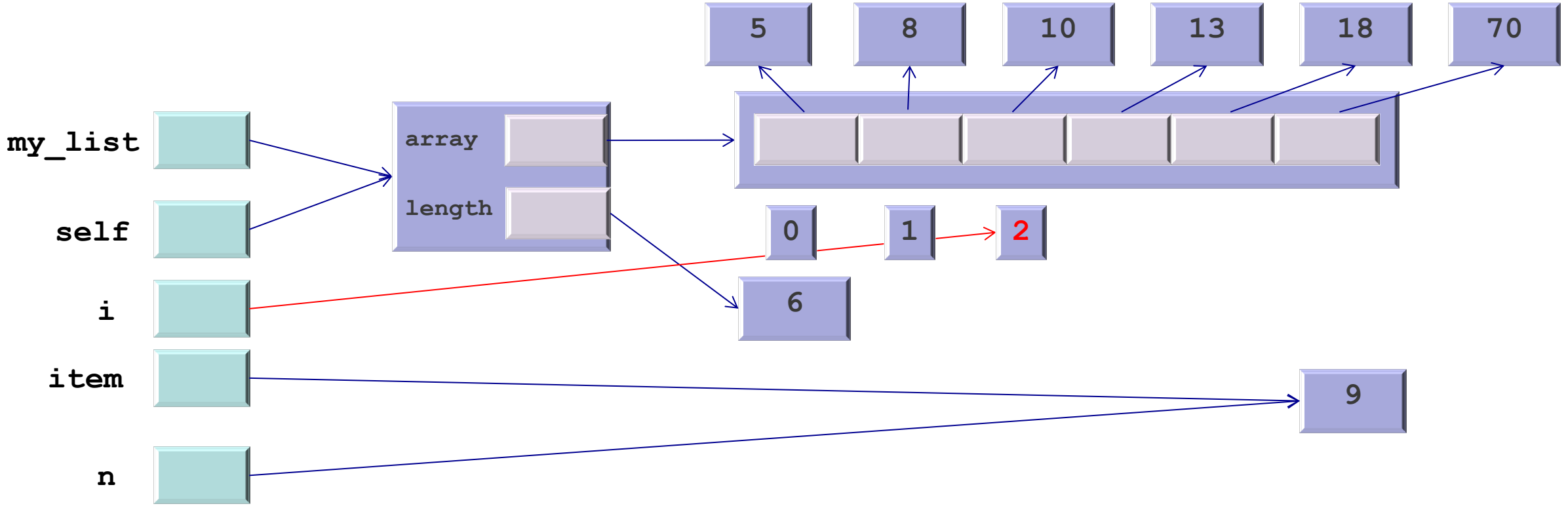

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



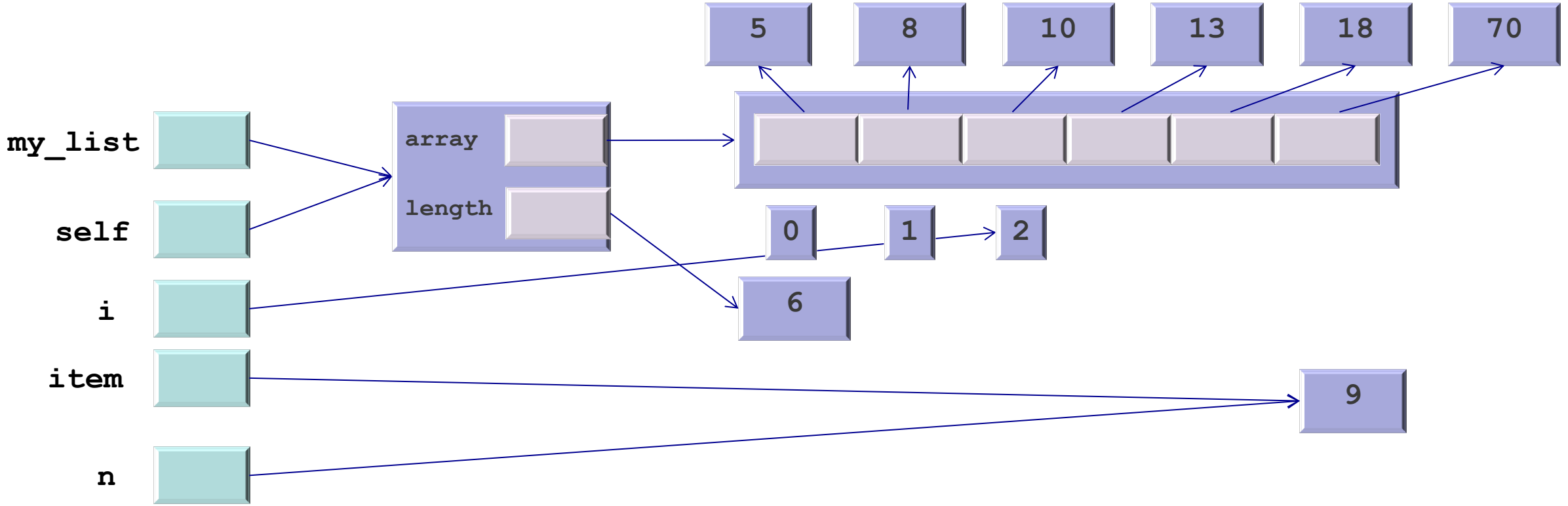

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



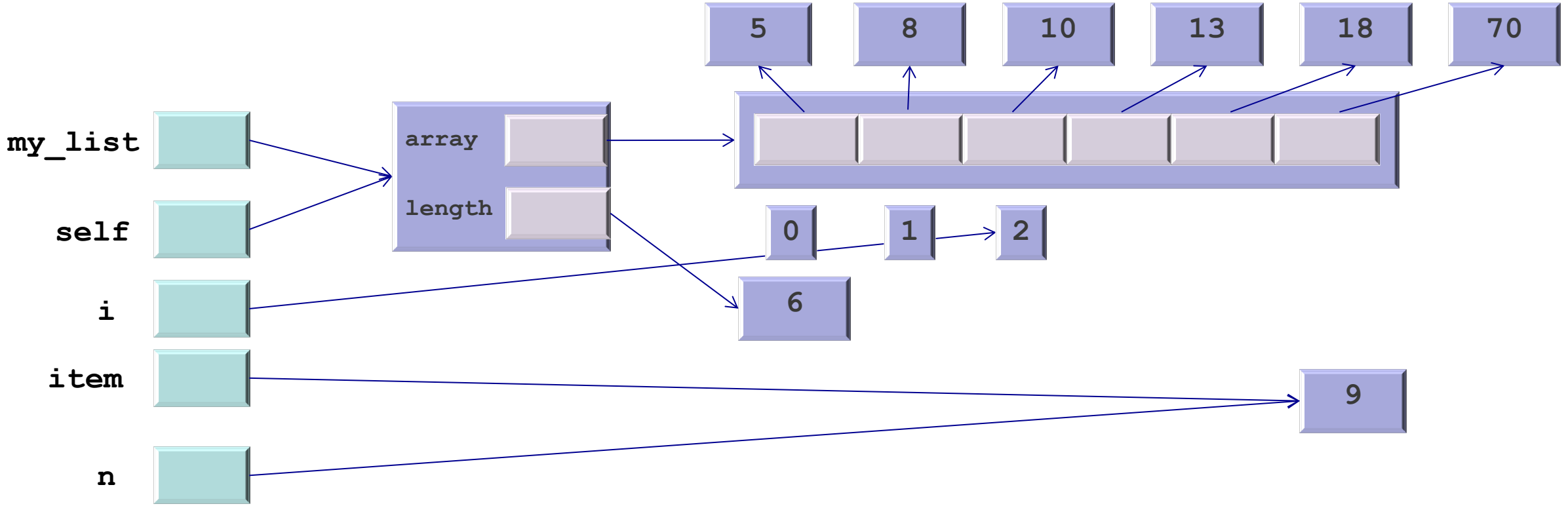

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]



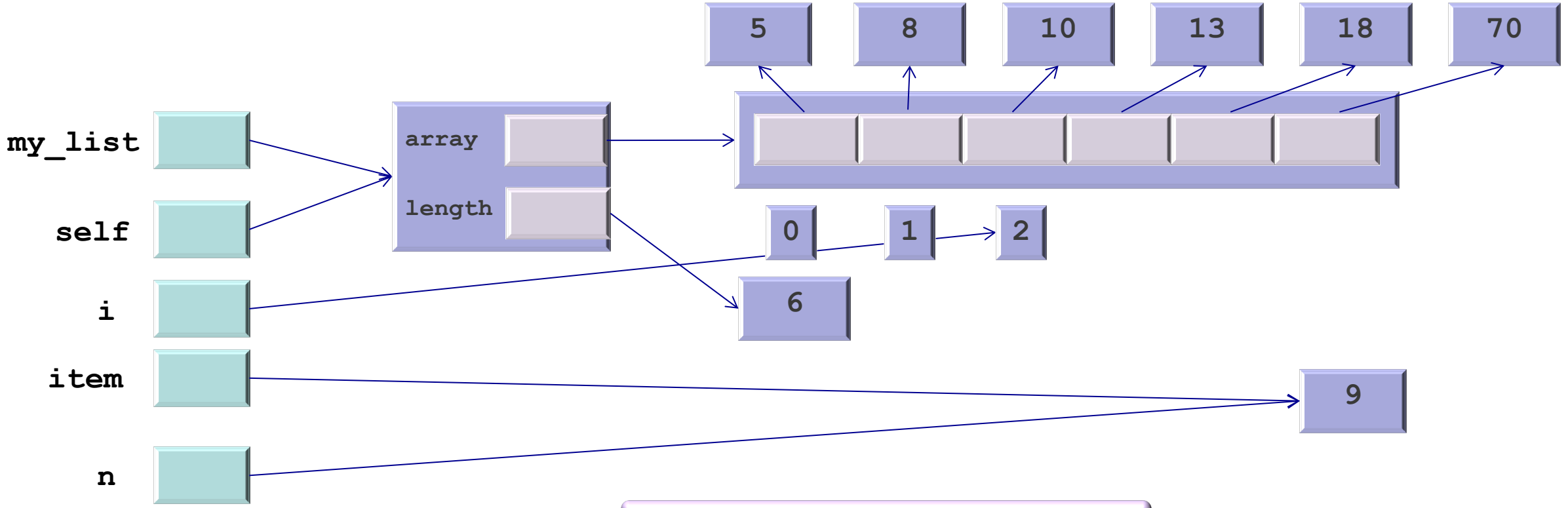

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item == self.array[i]:
            return i
        elif item < self.array[i]:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]




We don't need to look further!

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

Callee

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Caller

Assume, equivalent to [5,8,10,13,18,70]

# Big O Time Complexity for index

So, the same as for unsorted!
Why is it better then?

More efficient (rather than scalable), as it will often stop sooner

- **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 + 2m) +constant → O(len(self)*m) – or O(len(self)*($Comp_{==}$+$Comp_{<}$))

```
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
        elif item < self.array[i]:      Comparison is O(m)
            raise ValueError("item not in list")    irrelevant
    raise ValueError("item not in list")            irrelevant
```

? times

MONASH University

# Alternative Implementation

# An alternative (better/worse?) algorithm

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

Item has been found

Item cannot be in the list

Item is not in the list

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

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

Continue looking

Item has been found

Item cannot be in the list

Item is not in the list

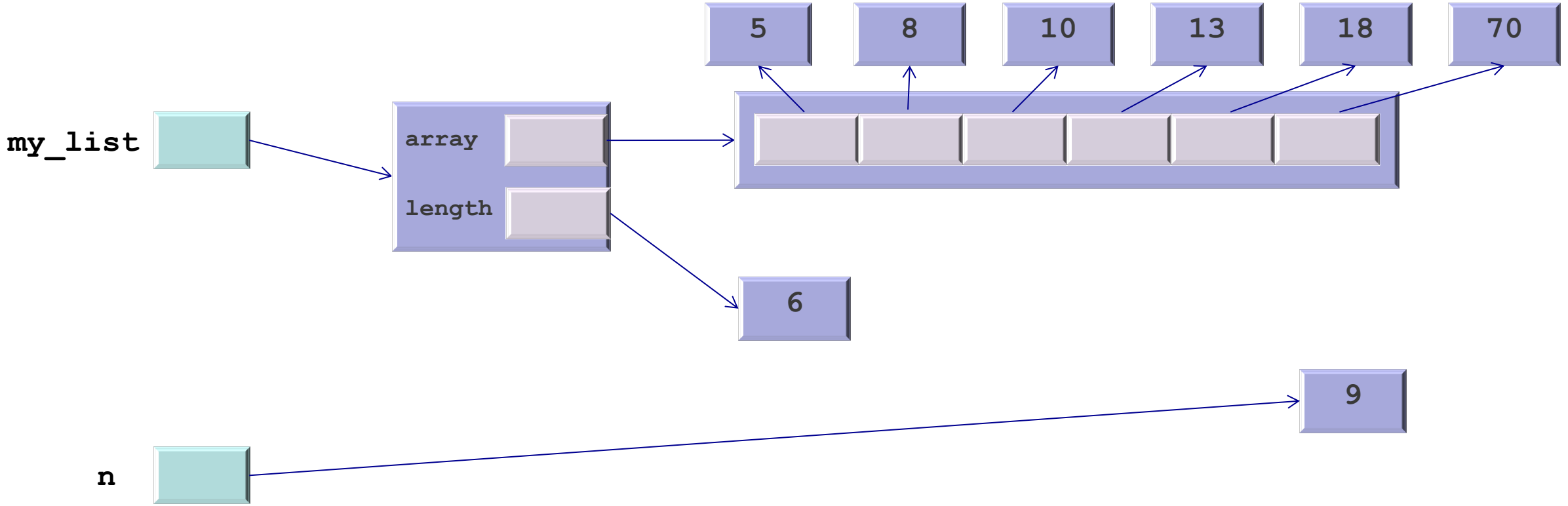
```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to
[5,8,10,13,18,70]



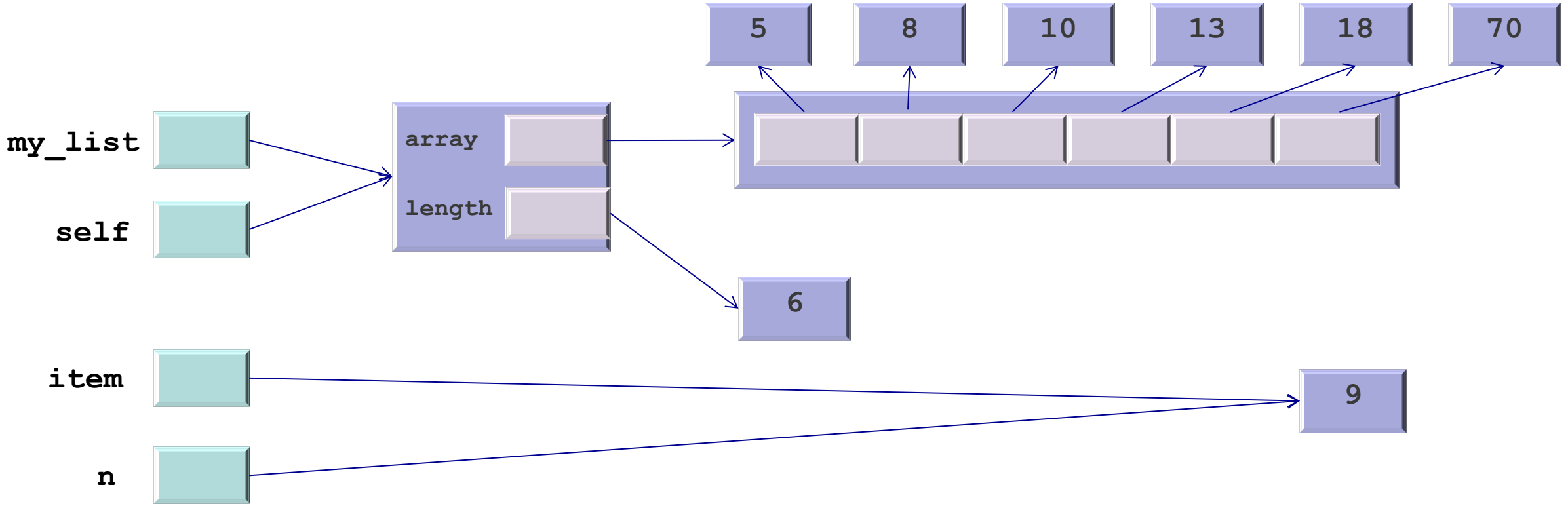

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



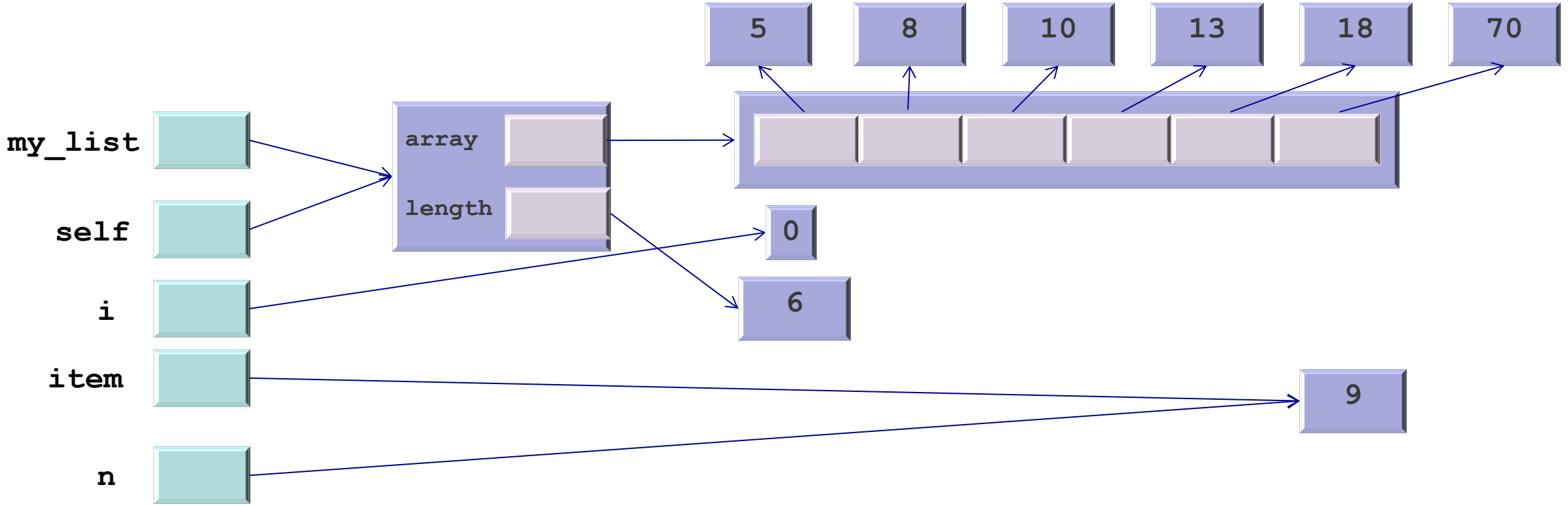

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



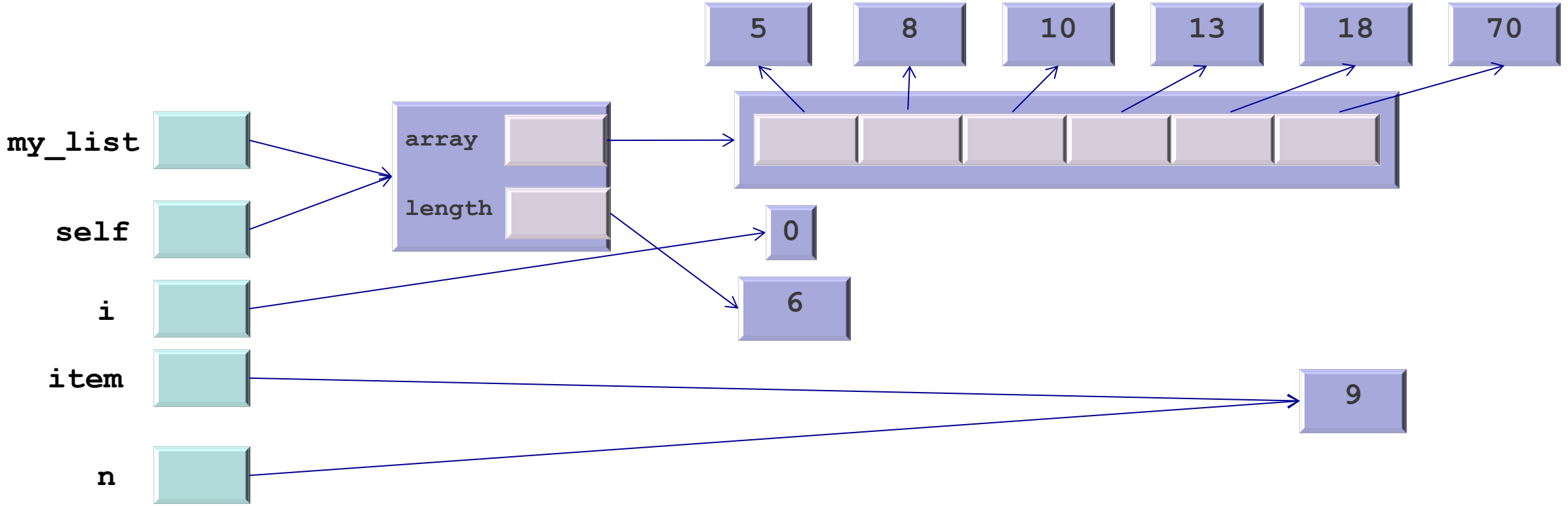

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



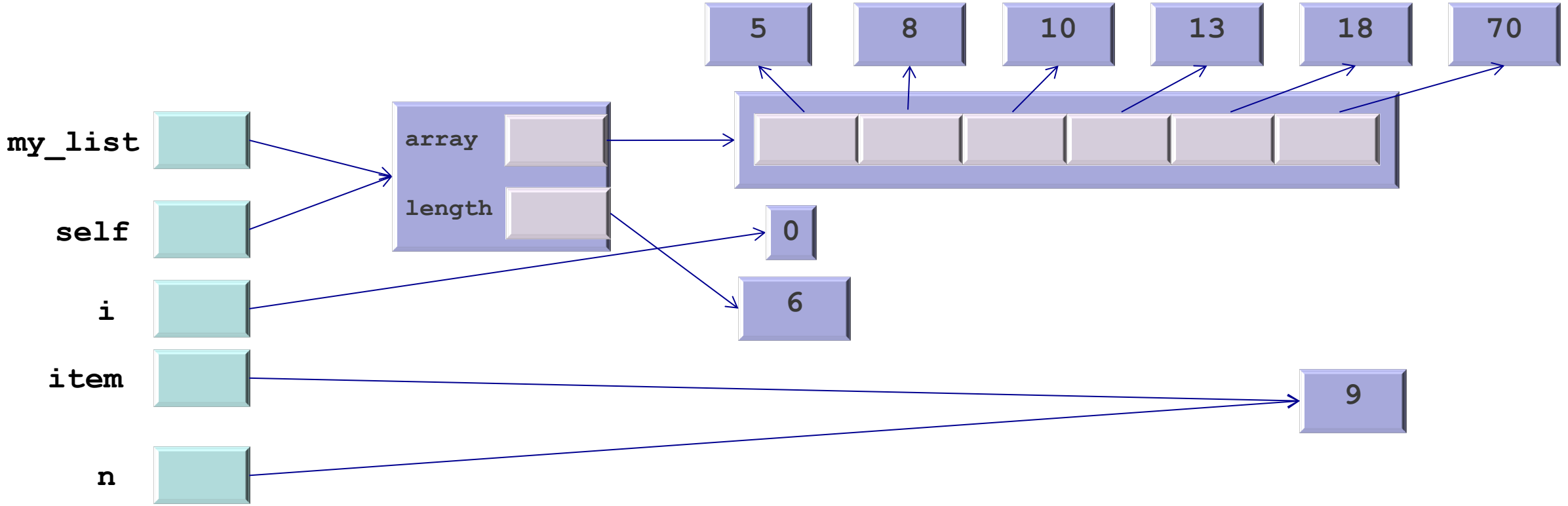

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



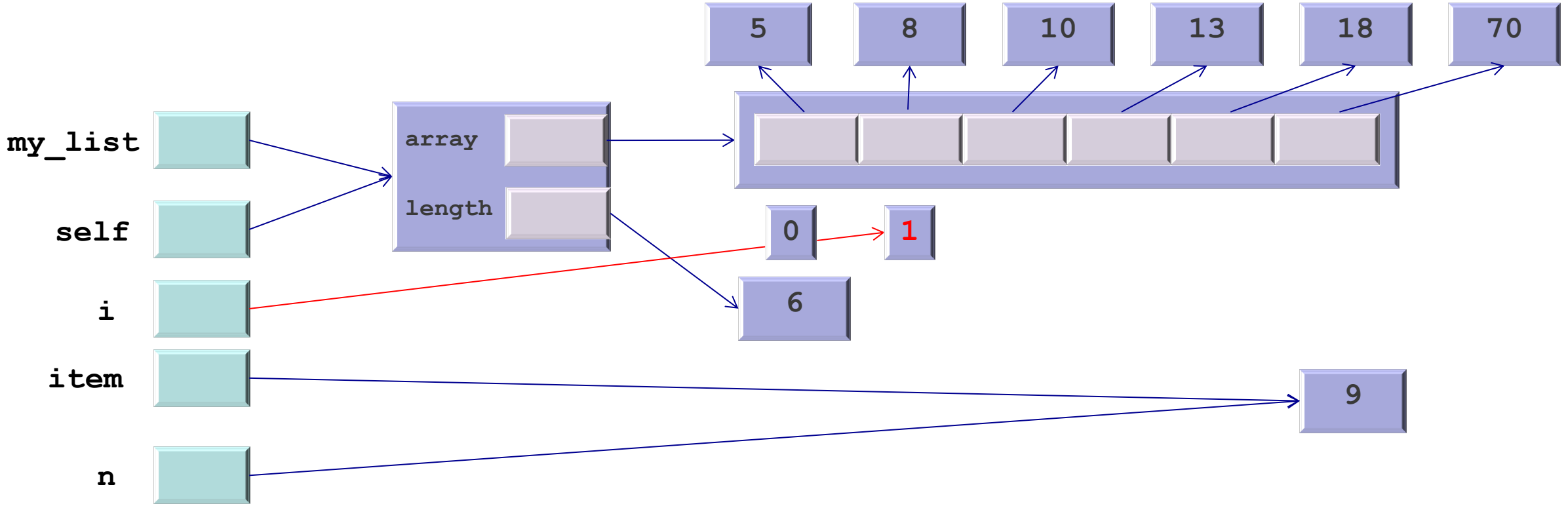

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



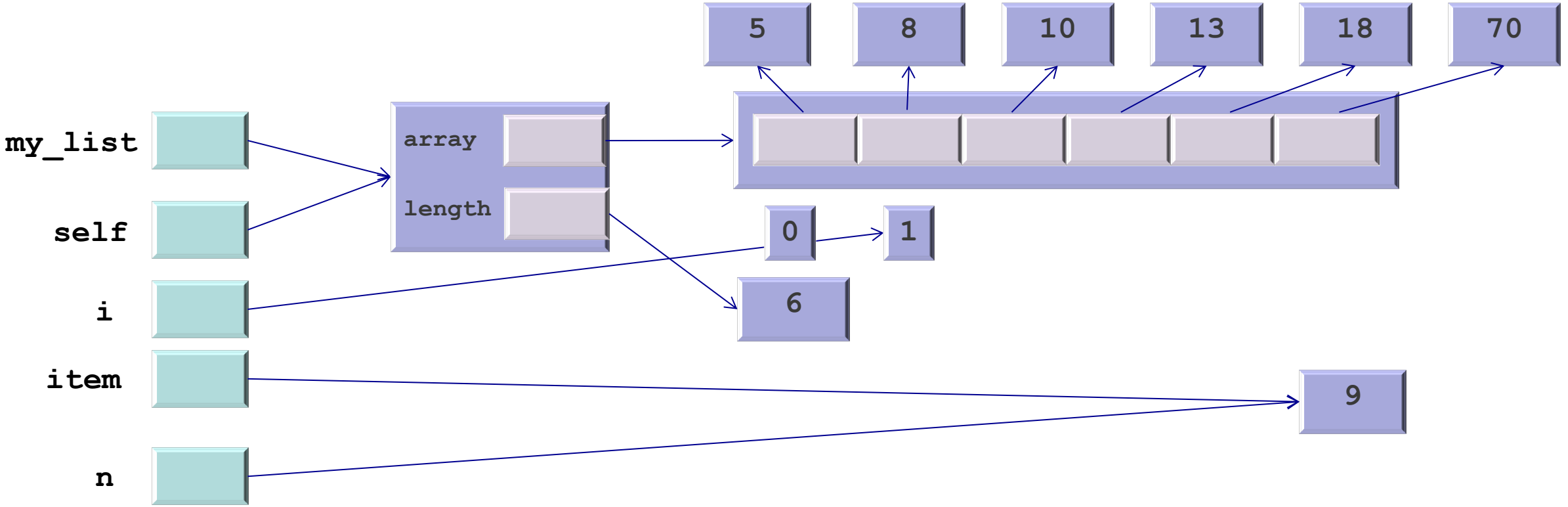

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



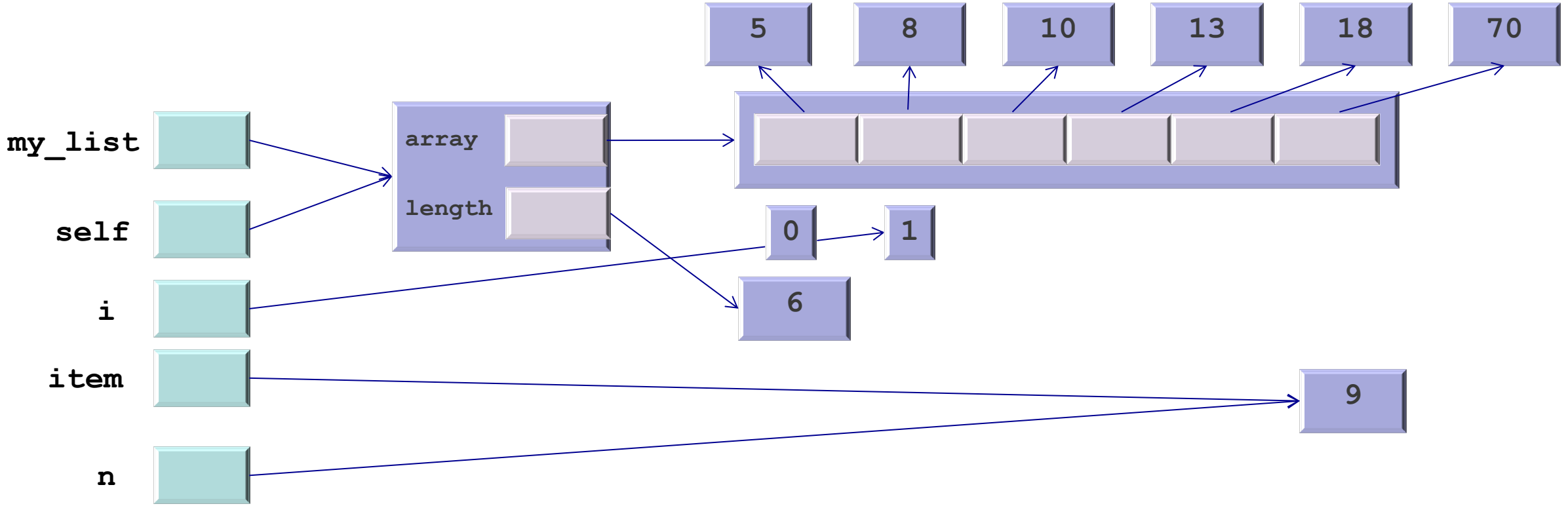

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



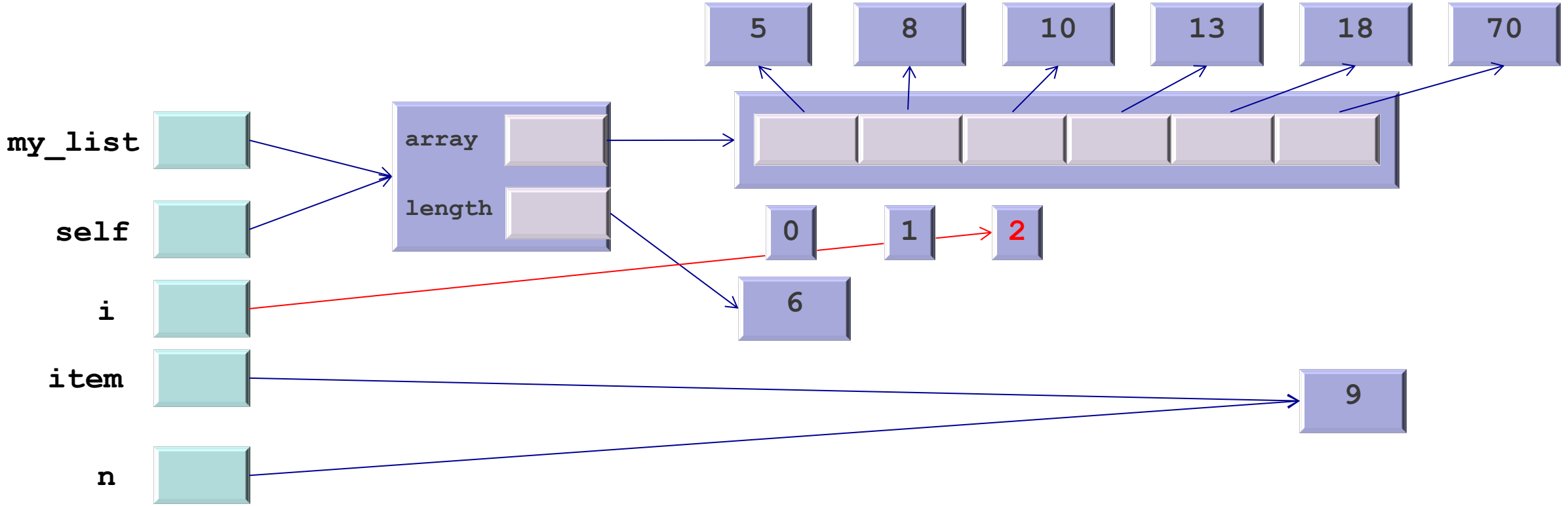

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



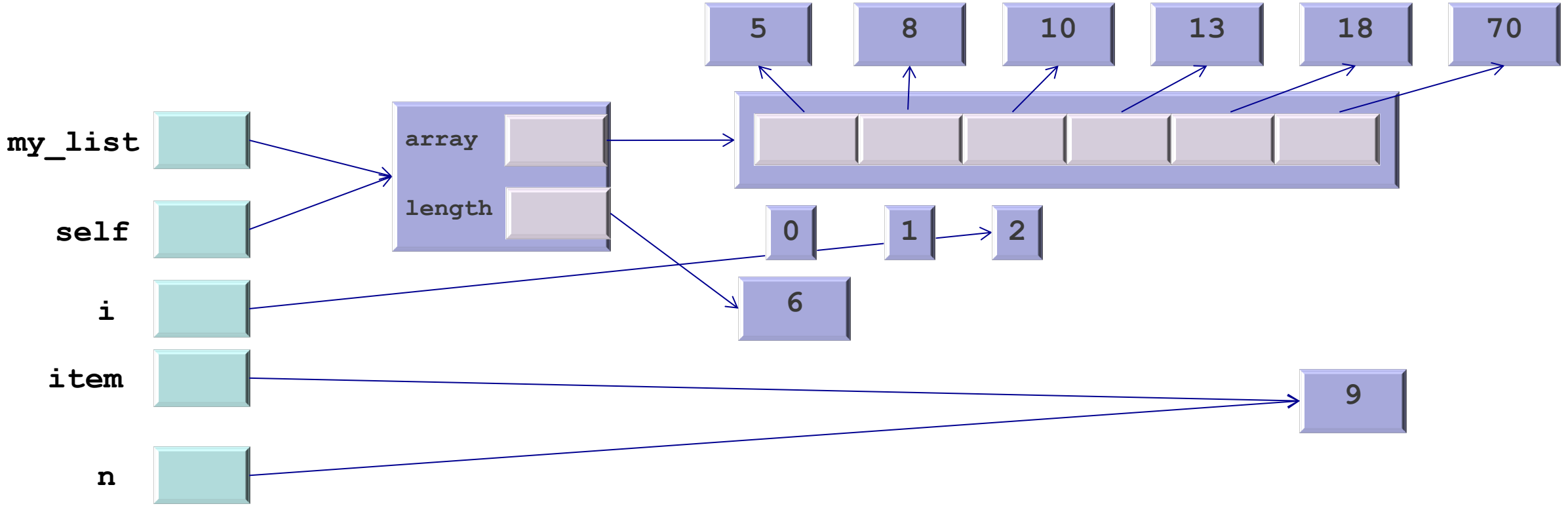

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



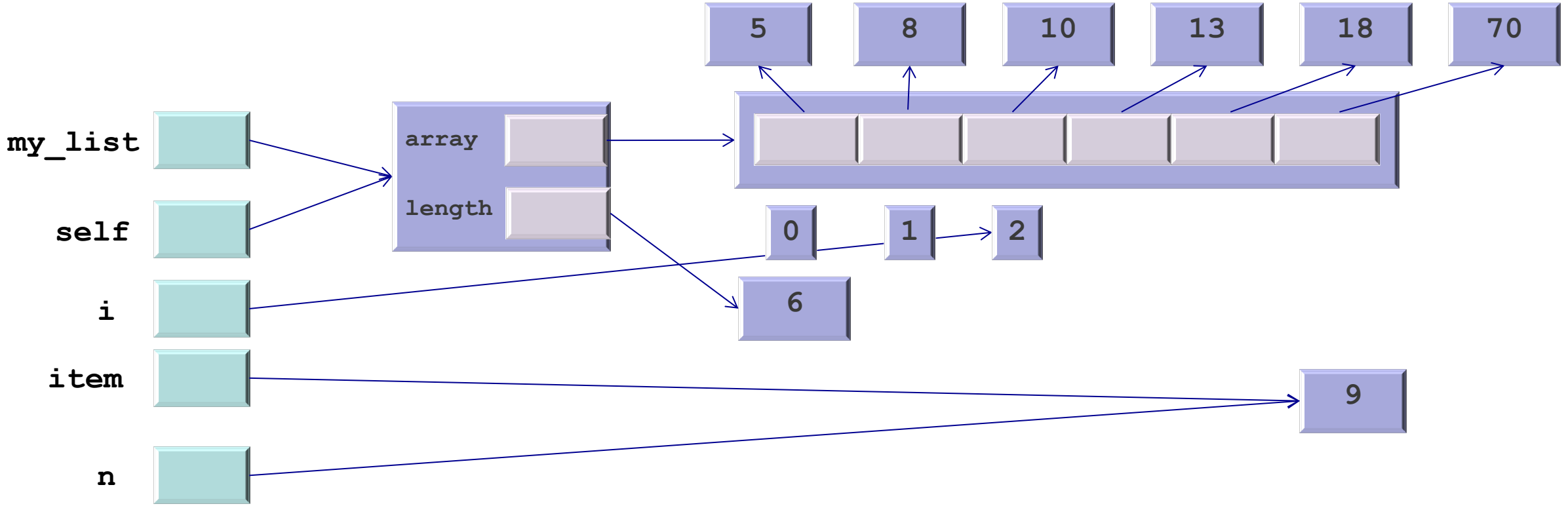

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



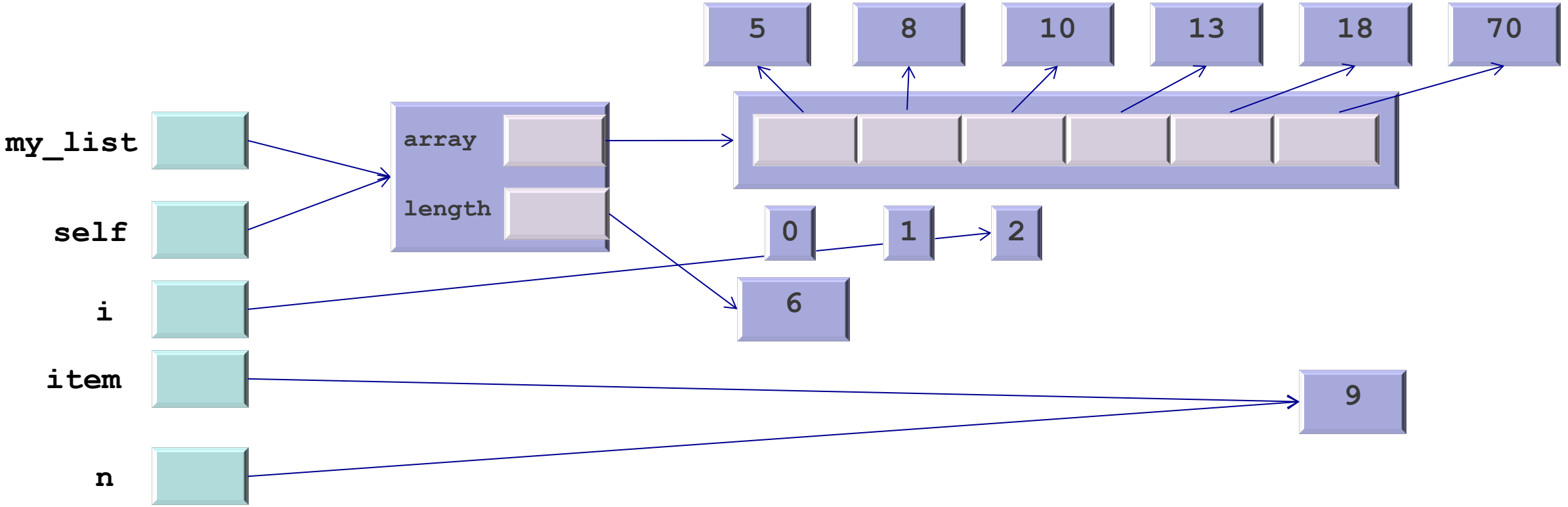

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]



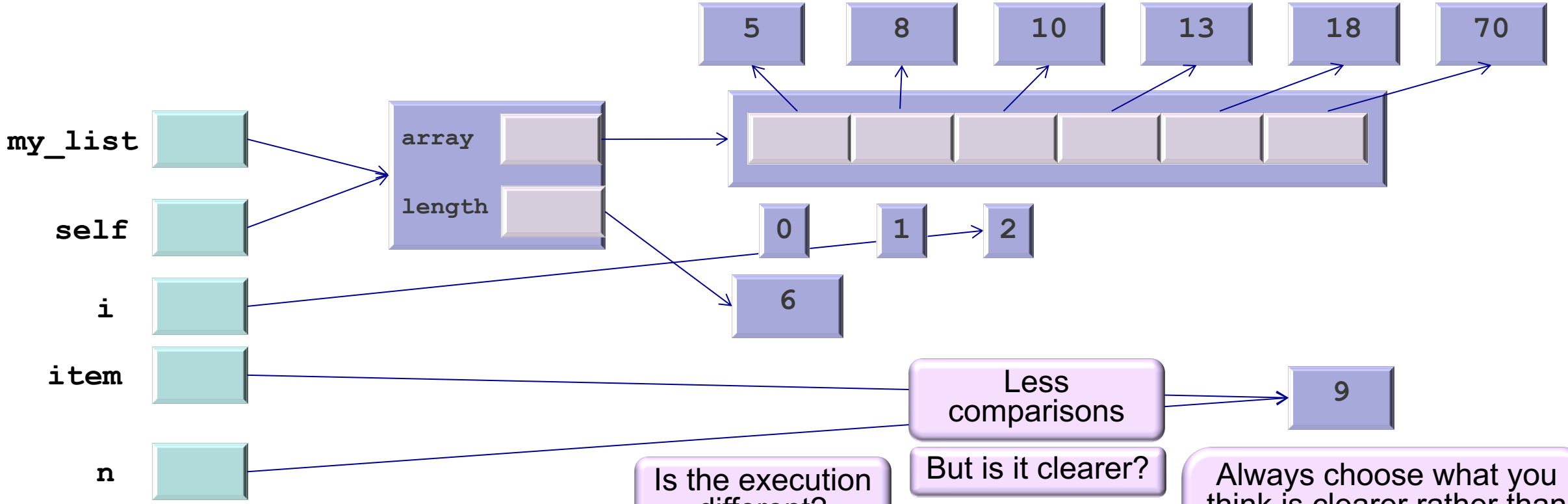

```
def index(self, item: T) -> int:
    for i in range(len(self)):
        if item > self.array[i]:
            continue
        elif item == self.array[i]:
            return i
        else:
            raise ValueError("item not in list")
    raise ValueError("item not in list")
```

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]




Less comparisons

But is it clearer?

Is the execution different?

Yes to me, but to others? It depends…

Always choose what you think is clearer rather than more efficient (unless you really need efficiency)

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

```
my_list = ArrayList(6)
my_list.add(5); … ; my_list.add(70)
n = 9
my_list.index(n)
```

Callee

Caller

Assume, equivalent to [5,8,10,13,18,70]

MONASH University

# Binary Search

# Binary Search

- **Linear search for sorted list is only better when the element is not found**
- **We would like something that is always better: Binary Search**
- **We can use it if the list is**
  - Sorted (for our algorithm, in ascending order)
  - Implemented with using arrays (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
```

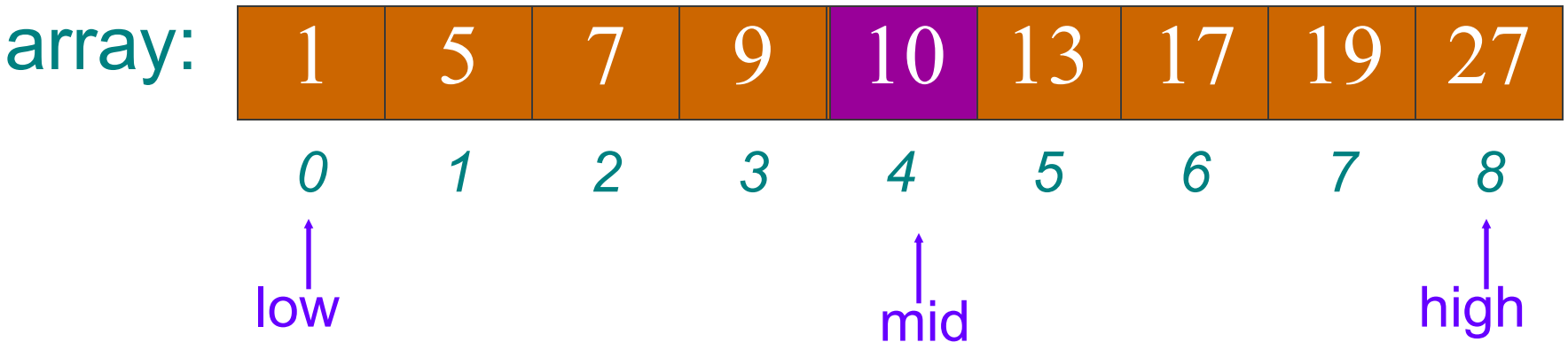
# Binary Search Case 1: val == array[mid]

val = 10

low = 0, high = 8

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

Different visualization to emphasise the array indices




Return **4**

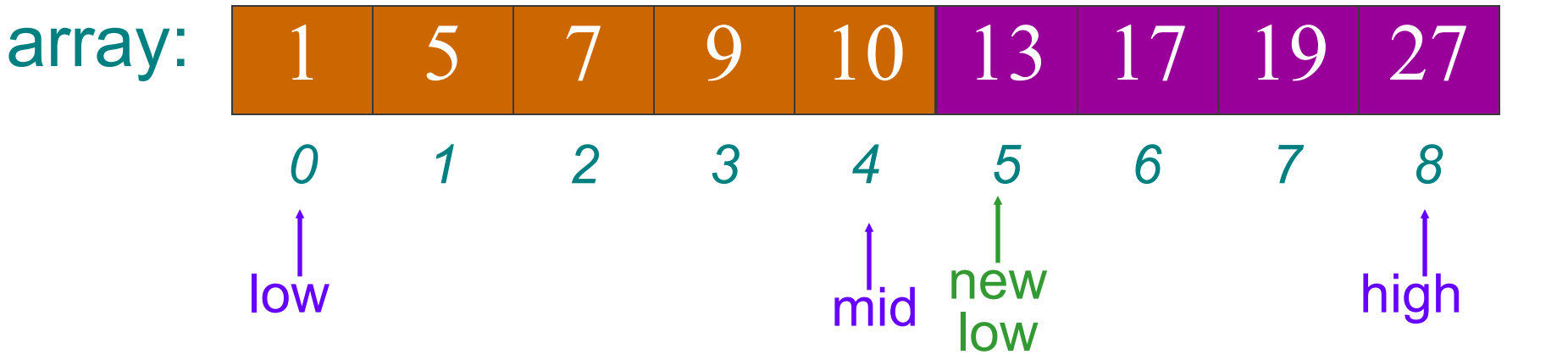
# Binary Search Case 2: val > array[mid]

val = 19

low = 0, high = 8

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

new low = mid + 1 = 5




Keep on searching using the same algorithm

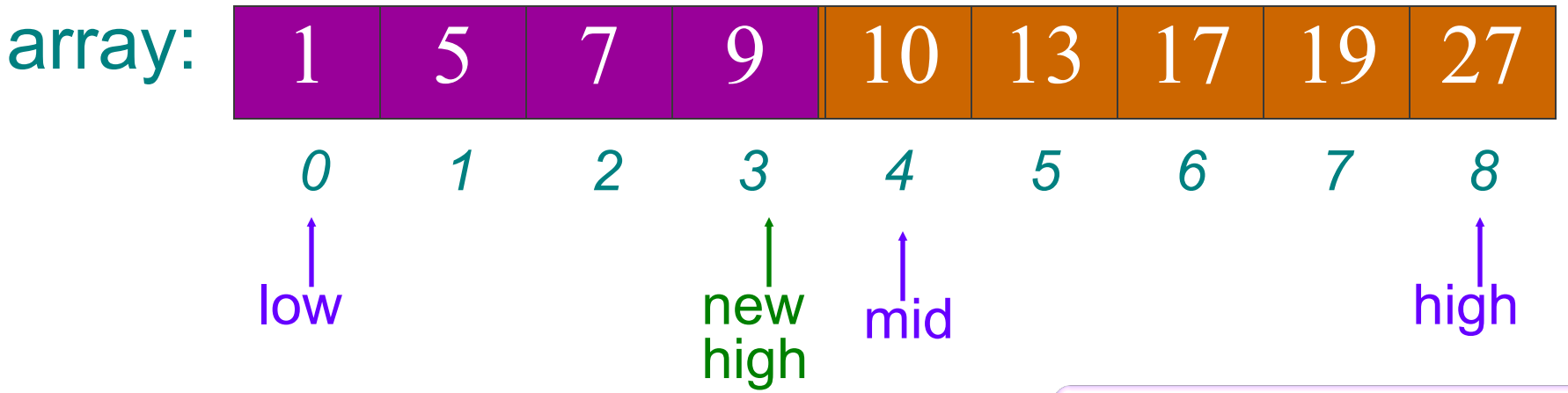
# Binary Search Case 3: val < array[mid]

val = 7

low = 0, high = 8

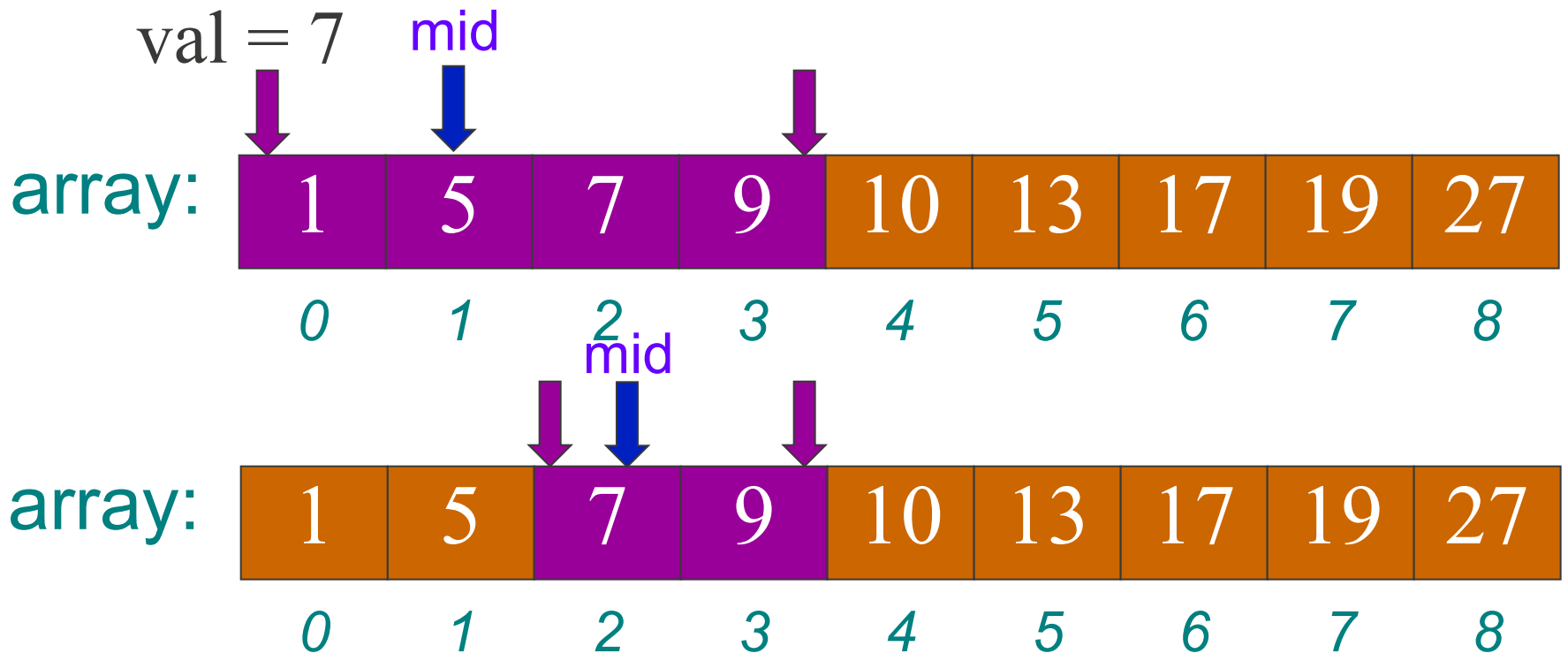
mid = (0 + 8) // 2 = 4

new high = mid -1 = 3




Keep on searching using the same algorithm

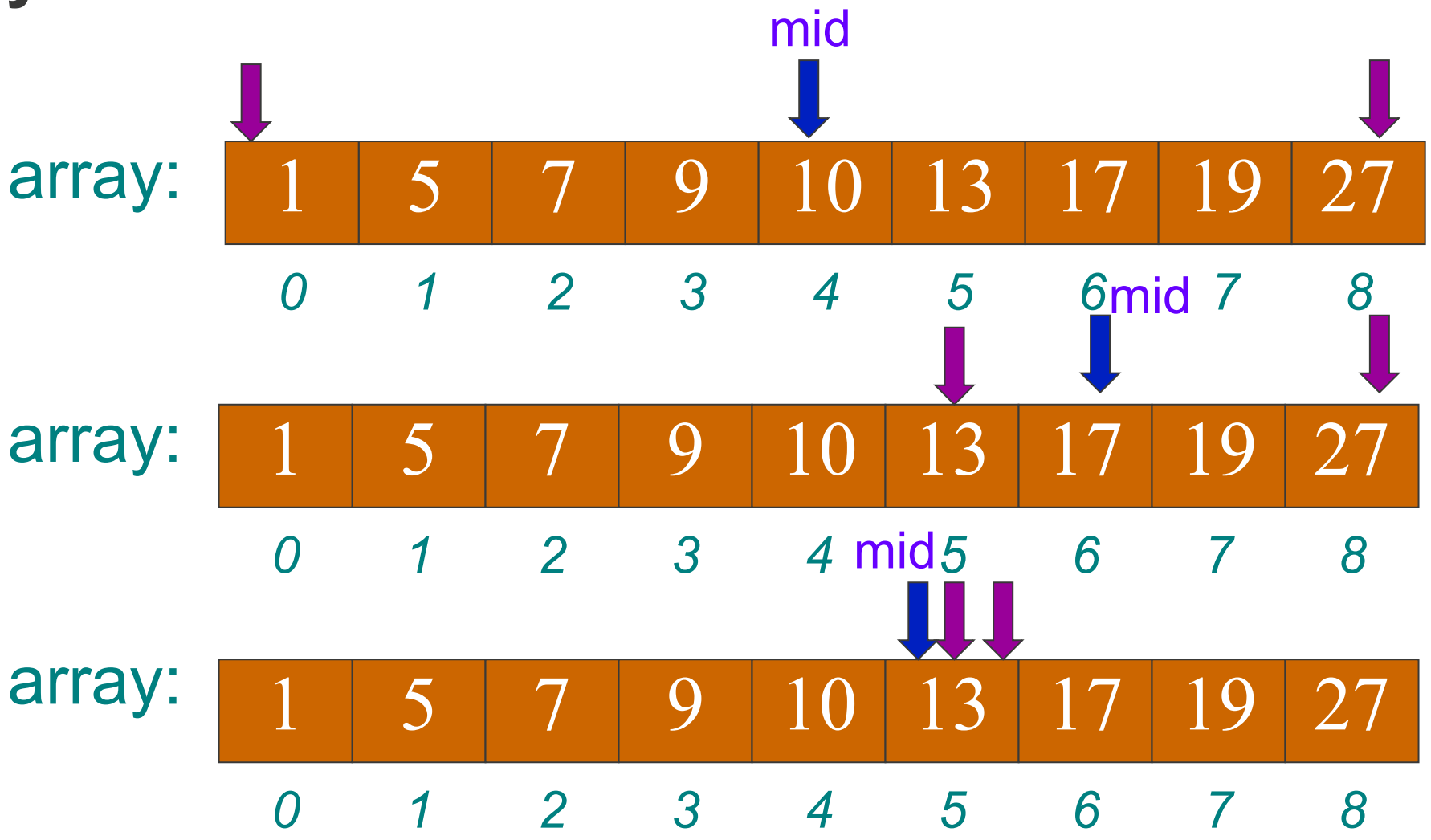
# Binary Search Case 3: val < array[mid] (cont)




Return 2

# Binary Search Case 4: val not in

val = 11



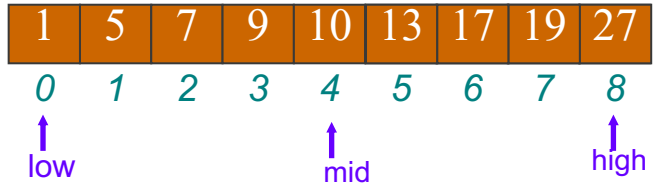

Then `low = 5, high = 4`

Raise `ValueError`

# Implementing index using Binary Search in Python

```
def index(self, item: T) -> None:
    low = 0
    high = len(self)-1
    while low <= high:
        mid = (low+high)//2
        if self.array[mid] > item:
            high = mid-1
        elif self.array[mid] == item:
            return mid
        else:
            low = mid+1
    raise ValueError("item not in list")
```

? times




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

- **Length of the list being searched: len(self)**
- **Best case?**
  - Loop stops immediately. When?
    - Item in the middle
  - m + some constants → O(m)
- **Worst case?**
  - Loop goes all the way: When?
    - Item is at one of the extremes
  - Each iteration (without finding) does one/two comparisons (O(m)) plus a fixed number of other operations
  - But how many times? $\log_2 n$
  - So, O(m*log n)

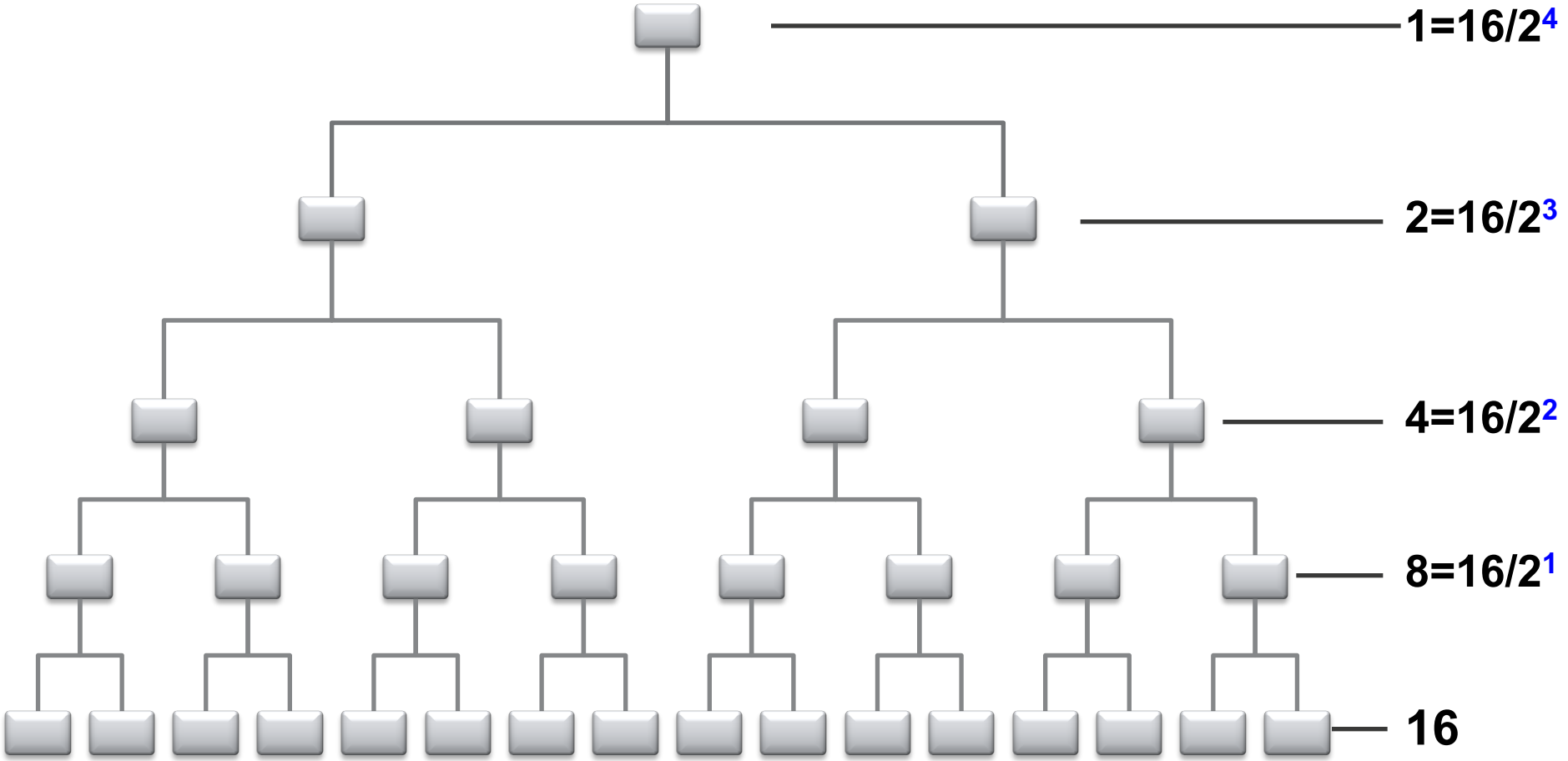
# Calculating the Worst Case Complexity

- 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)

MONASH University

# The add method

# The add method for sorted lists

- **Given an item, insert it in its appropriate position in the sorted list**
- **We can do this by putting together (similar) code from unsorted list:**
  1. Find the index of the position the item should go in
  2. Make space for the item by shuffling all elements from index to the right
  3. Put the item in the now empty space marked by index
  4. Increment the length
- **Something like:**

```
def add(self, item: T) -> None:
    index = self.__index_to_add(item)
    self.__make_space(index)
    self.array[index] = item
    self.length += 1
```

Find the index where item should be added

Make space for it

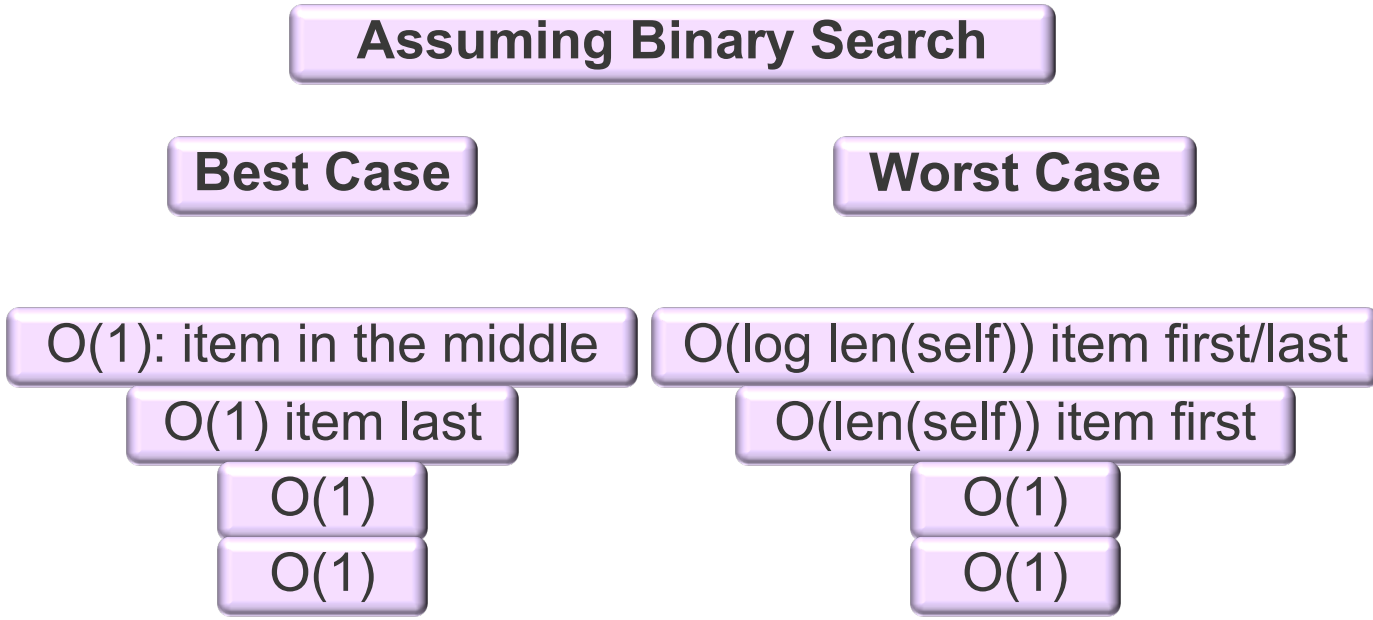
Put item in the space

Increase the length

- **I will leave this as an exercise for you!**

# Complexity of add

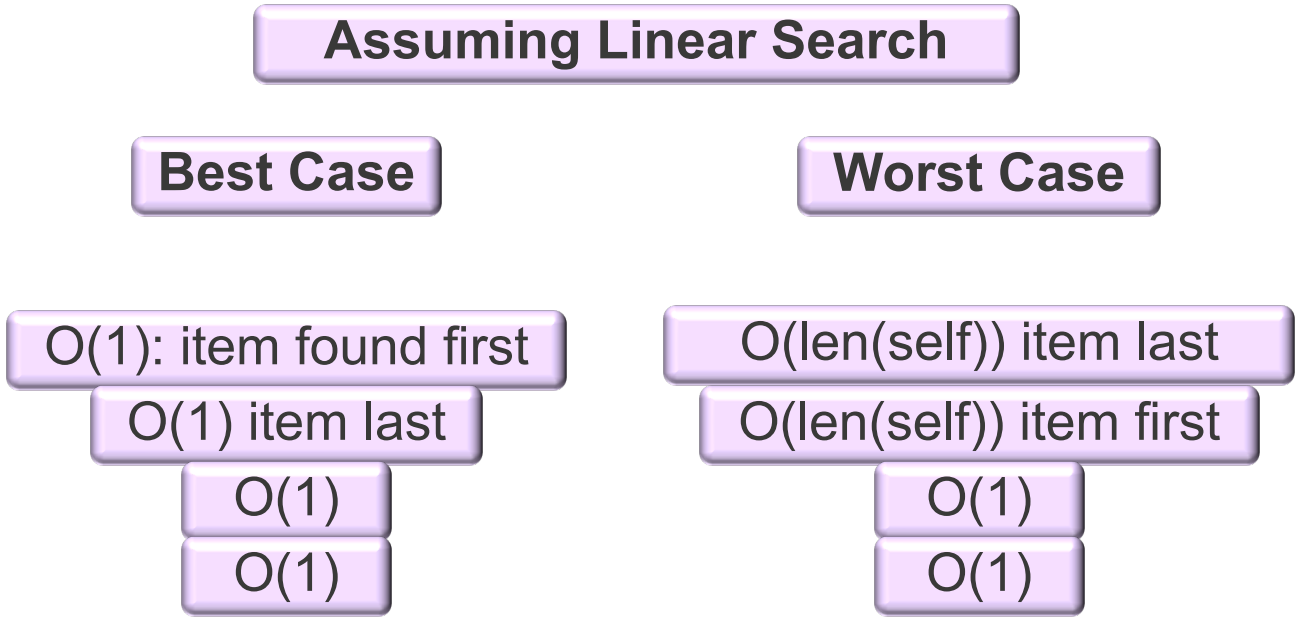
```
def add(self, item: T) -> None:
    index = self.__index_to_add(item)
    self.__make_space(index)
    self.array[index] = item
    self.length += 1
```




- **What will be the best case then?**
  - The best cases of the first two calls do not align
  - If item is last, we get O(log len(self)) + O(1) which gives O(log len(self))
  - If item is in the middle, we get O(1) + O(len(self)), which gives O(len(self))
  - If item is first, we get O(log len(self)) + O(len(self)), which gives O(len(self))
  - So the best case is when the item is last, and the worst when is first

# Complexity of add

```
def add(self, item: T) -> None:
    index = self.__index_to_add(item)
    self.__make_space(index)
    self.array[index] = item
    self.length += 1
```




- **What will be the best case then?**
  - The best cases of the first two calls do not align again
  - If item is last, we get O(len(self)) + O(1) which gives O(len(self))
  - If item is first, we get O(1)+ O(len(self)), which gives O(len(self))
  - So the best case and worst cases are the same

# Summary

- **We now understand the SortedList ADT and are able to:**
  - Implement the SortedList ADT using arrays
  - Implement several versions of its main operation
    - Search for an element's position in the list (`index`)
  - Reason about their complexity
- **Also able to decide when it is appropriate to use Stacks, Queues, Lists or Sorted lists**
  - Based on their methods and their invariants

# 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]
```