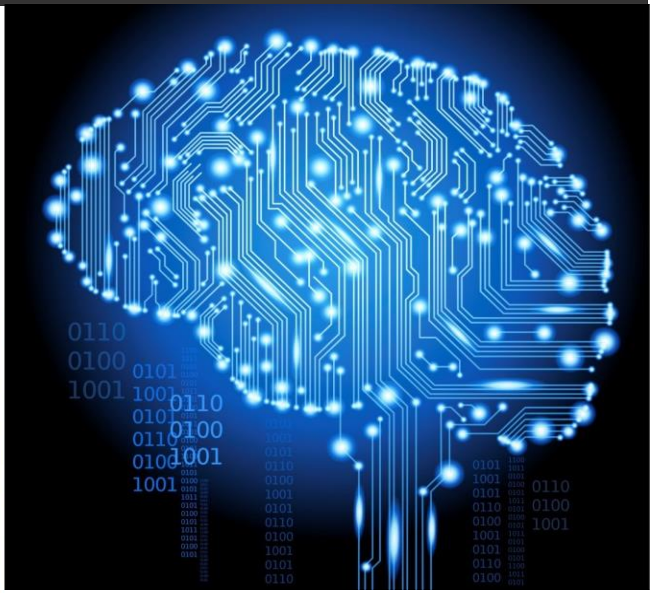
MONASH University

Information Technology

FIT1008 & 2085 Lecture 13

Abstract Data Types

Prepared by: M. Garcia de la Banda,
Pierre Le Bodic



Where are we at?

- **We are now familiar with Python basics**
- **Have learnt how to implement in Python:**
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- **Have learnt about time complexity**
- **Have started to become accustomed to think about:**
  - The use of invariants for improving our code
  - The properties of our algorithms (e.g., stable? incremental?)
  - Their Big O complexity

Objectives for this lecture

- **Understand the concepts of**
  - Data Types
  - Abstract Data Types (ADTs)
  - Data Structures

- **Start to implement our own ADTs for lists**
  - Implement some of their operations
  - Think about their properties
    - And modify them if appropriate
  - Compute their time complexity

Abstract Data Types

Data Types

- **Common concept in lower level languages (C, Java, …)**
- **Refers to a classification that determines:**
  - The possible values for that type
  - The meaning of those values
  - The operations that can be done on them
  - The way those values are implemented
- **Example: if a Java variable has type `int`**
  - Can take values from -2,147,483,648 to 2,147,483,647
  - Their meaning is that of an integer number
  - Can be used in all integer operations (add, subtract, etc)
  - Implemented using 32 bits and specific bytecode operations

Data Types

- **Knowing the implementation can have advantages**
  - Extra functionality
  - Speed
  - …
- **E.g., some languages implement `False/True` as `0/+`**
  - AND allow programs to use arithmetic with booleans
- **Using the implementation can have disadvantages:**
  - Lack of portability
  - Poor maintenance
  - …

Abstract Data Types (ADTs)

- **Often no need to know how types are implemented**
  - Data abstraction
- **An abstract data type:**
  - Provides information regarding:
    - The possible values of the type and their meaning
    - The operations that can done on them
  - BUT *not* on its implementation, i.e. how:
    - The values are stored
    - The operations are implemented
  - Users interact with the data only through the provided operations
- **In some languages abstraction mixed with hiding**
  - Like in Java: actively hides implementation (e.g., **`private`**)

Advantages of ADTs

- **Build programs without knowing their implementation**
  - Simplicity!
- **The implementation can change without affecting you**
  - Maintenance!
- **If several ADTs available, you could easily use any**
  - Flexibility and reusability!
- **Different compilers can use different implementations**
  - Portability!

Data Structures

- **At some point we must give ADTs an implementation**
- **Some ADTs (like lists) contain several data fields**
  - How do we organise the data? How do we access it?
- **That is what a data structure provides:**
  - A particular way in which data is physically organised (so that certain operations can be performed efficiently)
- **Example: the array data structure**
  - Fixed size
  - Data items are stored sequentially
  - Each item occupies the same amount of space

Physical organisation

This allows constant time access to any element (remember your MIPS!)

- **That looks VERY much like a Python list:**
  - Because Python lists are implemented using arrays

This is becoming confusing!

- **We have already talked about**
  - Data types
  - Data structures
  - Abstract Data types
- **And this is only part of the picture, we also have:**
  - Primitive (or built-in) data types versus user-defined
    - Readily available in a given programming language or not
  - Simple (or basic, or atomic) versus complex ones
    - Single data versus multiple data fields
- **What is the relation between them? It is all about:**
  - Abstraction level
  - Simple/complex data (single/multiple data)

A way to clarify things a bit (not gospel)

Higher level language (Python, Scribble)	Only ADTs (no details about implementation)	Primitive simple ADTs: integers, booleans,... Primitive complex ADTs: lists, strings, … Non primitive simple/complex ADTs: users can add any ADT they want.
Mid-level (Java)	Primitive data types plus user/library defined ADTs.	Same as below, plus non primitive simple/complex ADTs (implementation is hidden)
Lower level language (C, Fortran)	Primitive data types (both simple and complex). Details of implementation are known.	Primitive simple data types: int, short, float… Primitive complex data types (called data structures): arrays, strings Non primitive simple/complex data types: users can add anything like time, linked lists, array lists…
Assembly language instructions	32-bits registers and a few operations on them	Primitive simple data types: 8-, 16-, 32-bit signed/unsigned integer, float.
Hardware implementation	Bits and logic circuits	No real concept of type: bit, bytes, word …

For those interested in language evolution, but not examinable

Just remember that in this unit, we say:

- **Abstract Data types provide information about**
  - The possible values for that type
  - The meaning of those values
  - The operations that can be done on them
  - **Example: a list (however it is implemented – don’t care)**
- **Data Types provide the same info plus:**
  - The way those values are implemented
  - **Example: a list for which I know (and make use) how it is implemented**
- **Data Structures provide information about:**
  - A particular way in which data is physically organised in memory
  - **Example: an array**

The List ADT

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

Our List ADT

- **This week I will ask you to define you own list ADT**
  - Why on earth? They are already in Python!
- **Because you need 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)
  - Does Python have traditional arrays? (fixed size)
    - Yes, but they are a bit cumbersome

Implementing a List ADT using arrays

- **For now we will use Python lists as our arrays**
  - After all, they ARE implemented with arrays
- **This means our implementation can only use the list operations that are also array operations, that is:**
  - Create an array/list (e.g., x = [1,2,3])
  - Access an element in position P (e.g., item = x[i])
  - Obtain its length (e.g., n = len(x))
- **Do NOT use other python's list functions (e.g., append)**
  - The point is for you to implement these functions yourselves!
- **But this is an ADT. Should we hide the implementation?**
  - No, the abstraction comes from the user ignoring it, not form hiding it

Implementing your own List ADT

- **How do we start? Easy:**
  - Create a new file (called my_list.py)
  - Add any operation users will ever need to use!
- **What operations? We could do many…**
  - Create a list, access an element, compute the length
  - Determine whether is empty
  - Determine whether it has a given item
  - Find the position of an item (if in)
  - Add/delete an item
  - Delete/insert the item in position P
- **Let's create also an ADT for sorted lists:**
  - Lists whose elements are always sorted (sorted_list.py)
  - Same operations? We will see…

Lets start with the obvious

- **Our first functions are implemented using the list ones**

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

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

def length(the_list):
    return len(the_list)
```

The uppercase is not a typo, we will see later…

Simpler in MIPS: allocate space, store size, done!

For now: we assume the size of the list is the size of the array, i.e., no empty positions in the array…

- **Time complexity?**
  - That of the return statement
  - Which is O(1) for all (assuming the size is stored) except for creation, which is O(size) – would be O(1) for MIPS

Is the list empty?

- **Not really needed (users have `length`) but useful**

```
def is_empty(the_list):
    return len(the_list) == 0
```

- **Time complexity?**
  - Time to compute the length
    - Constant (K1)
  - Time to compare two integers
    - Constant (K2)
  - Time to return the value
    - Constant (K3)
  - K1+K2+K3 is some constant so → O(1)
- **Any properties of the list elements that affect big O?**

No! so best = worst

Determine whether an item is in a List

- **Input:**
  - List
  - Item
- **Output:**
  - **`True`** if the item is in the list, **`False`** otherwise
- **Plan for 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

You might have done this in the assignment for temperature frequencies

Several possibilities in Python

- **For those accustomed to indices:**

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

- **But Python can generates all elements within a list**
- **Which means there is and even easier way:**

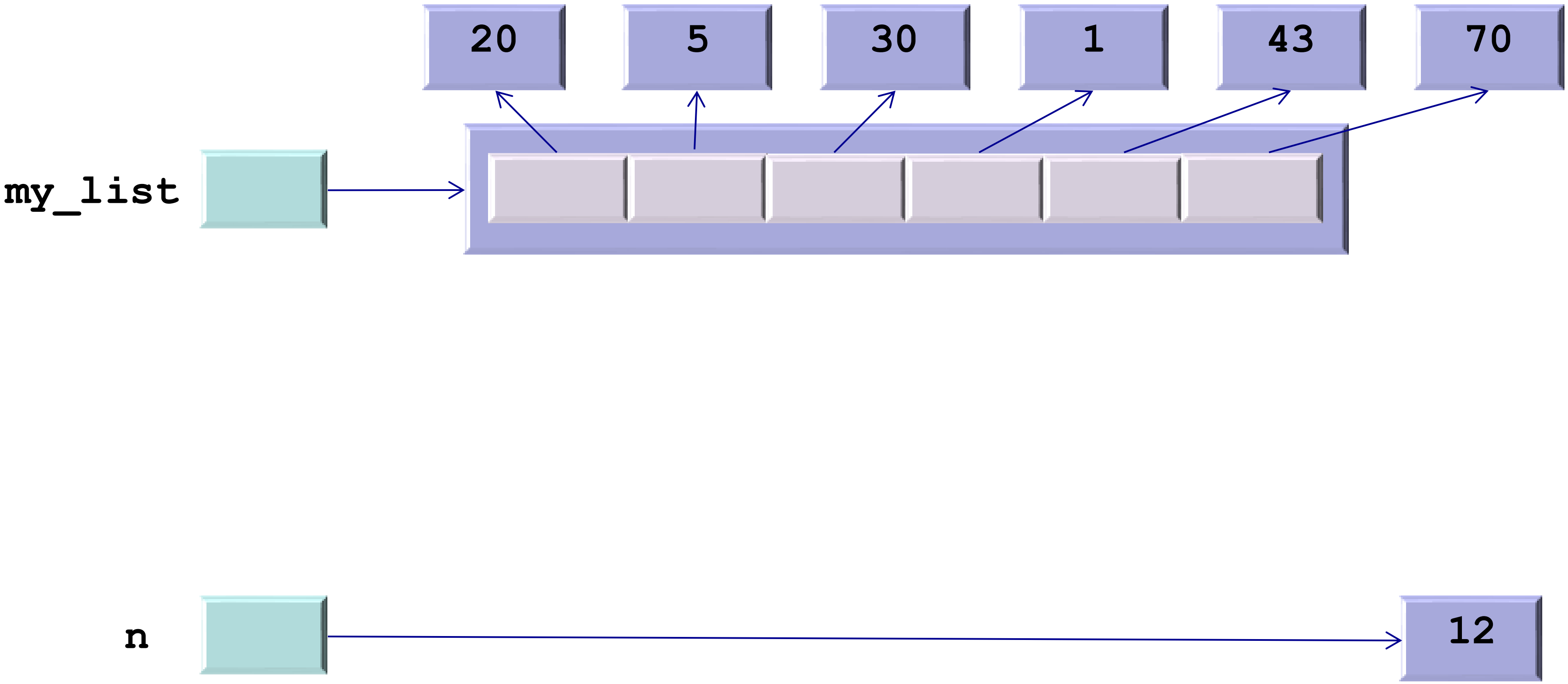
```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

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

```python
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

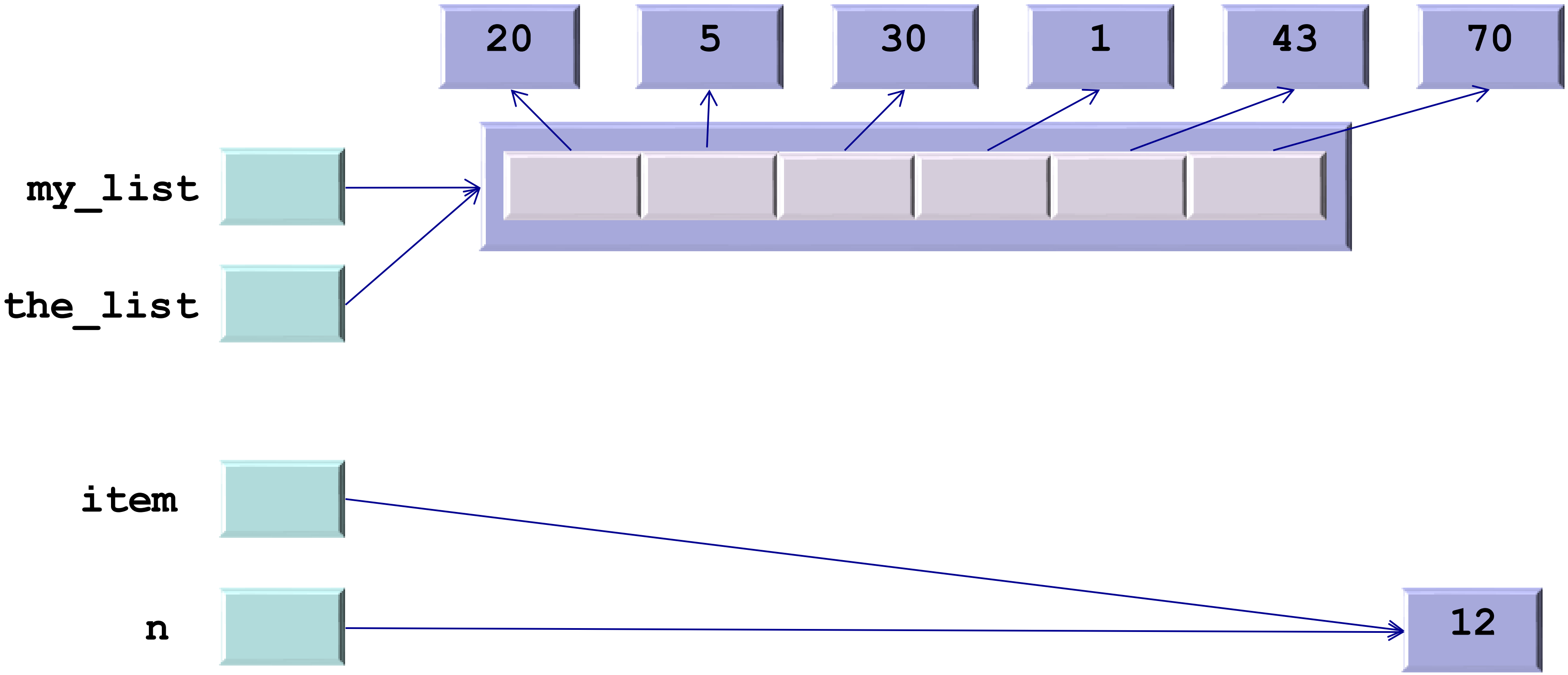



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

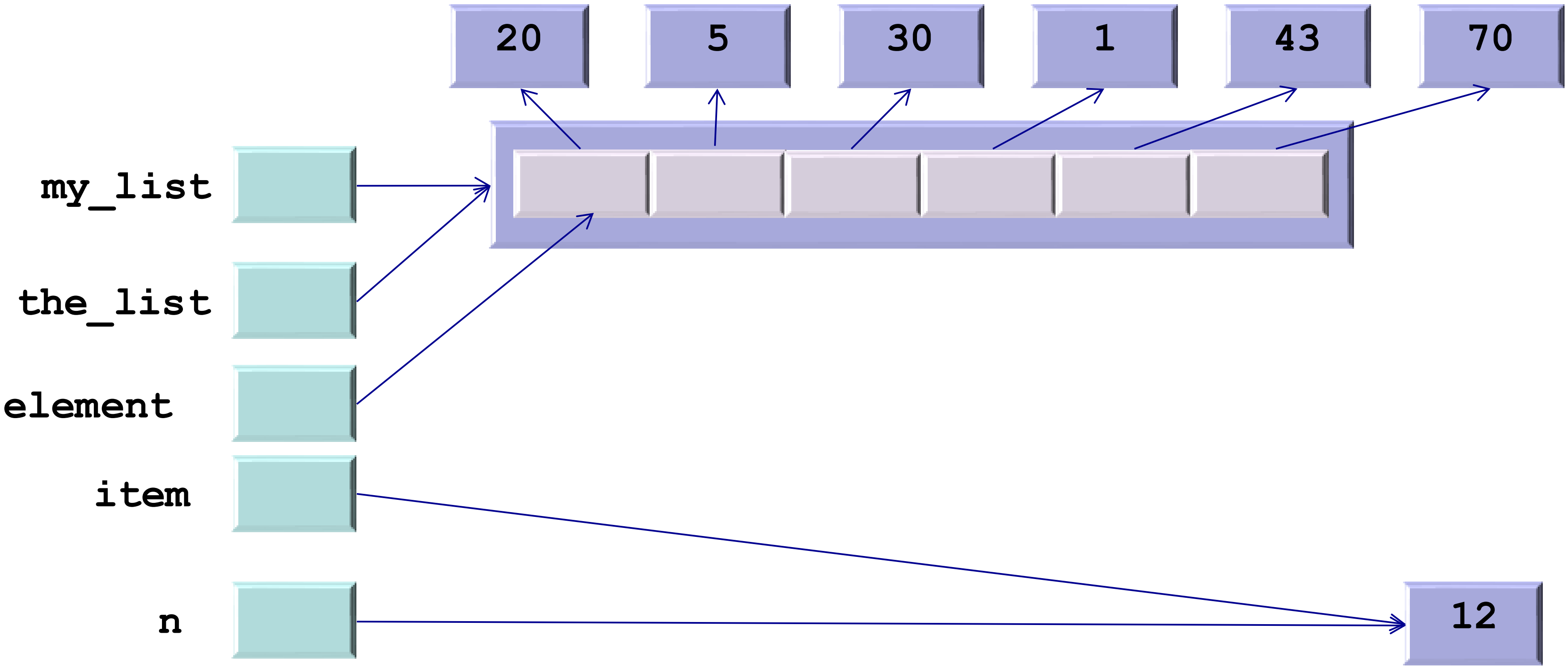



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

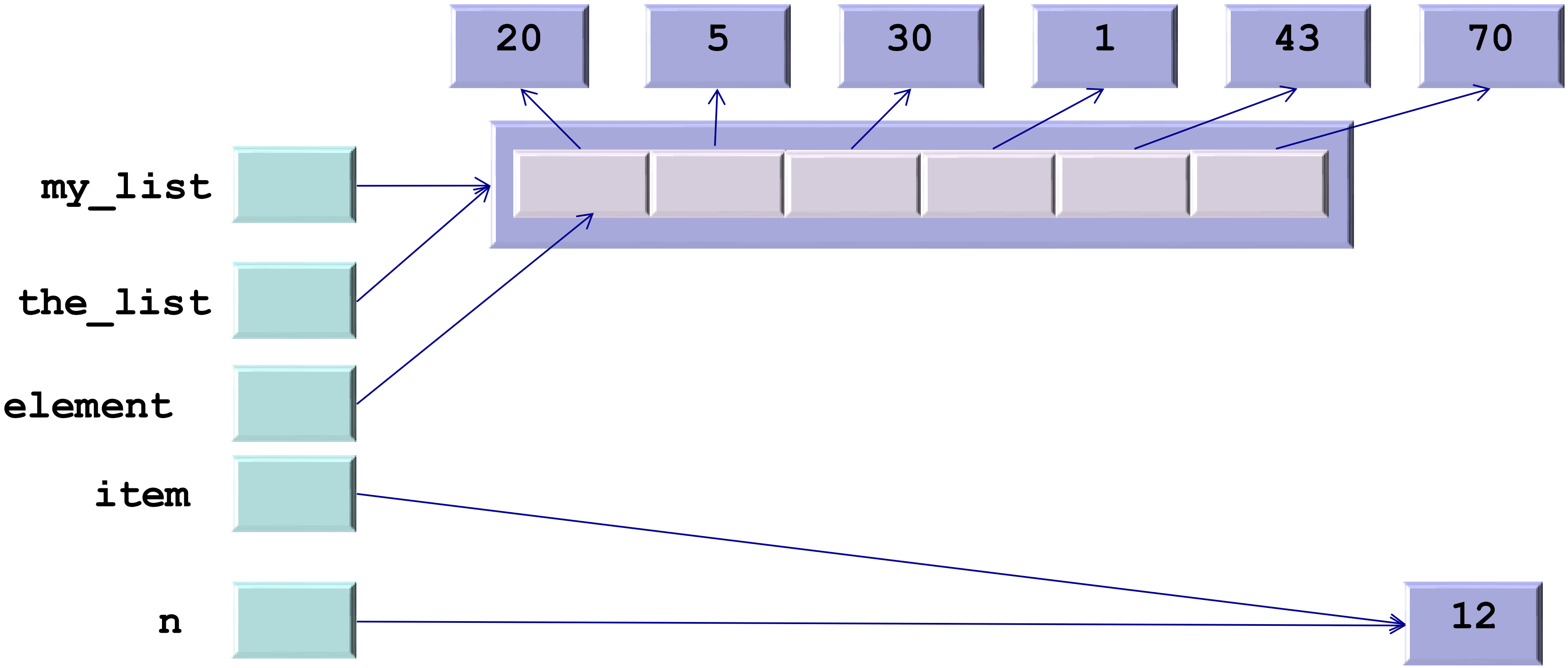



```python
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```python
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

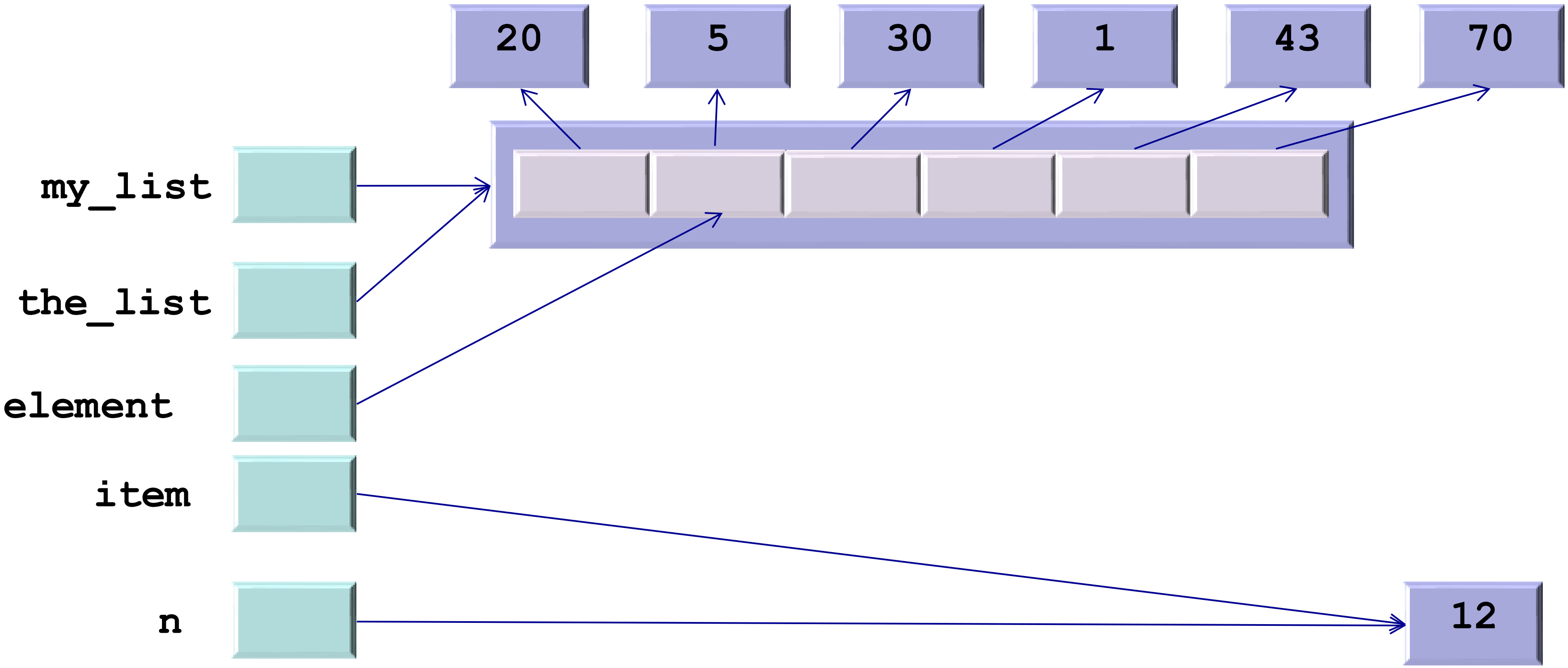



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

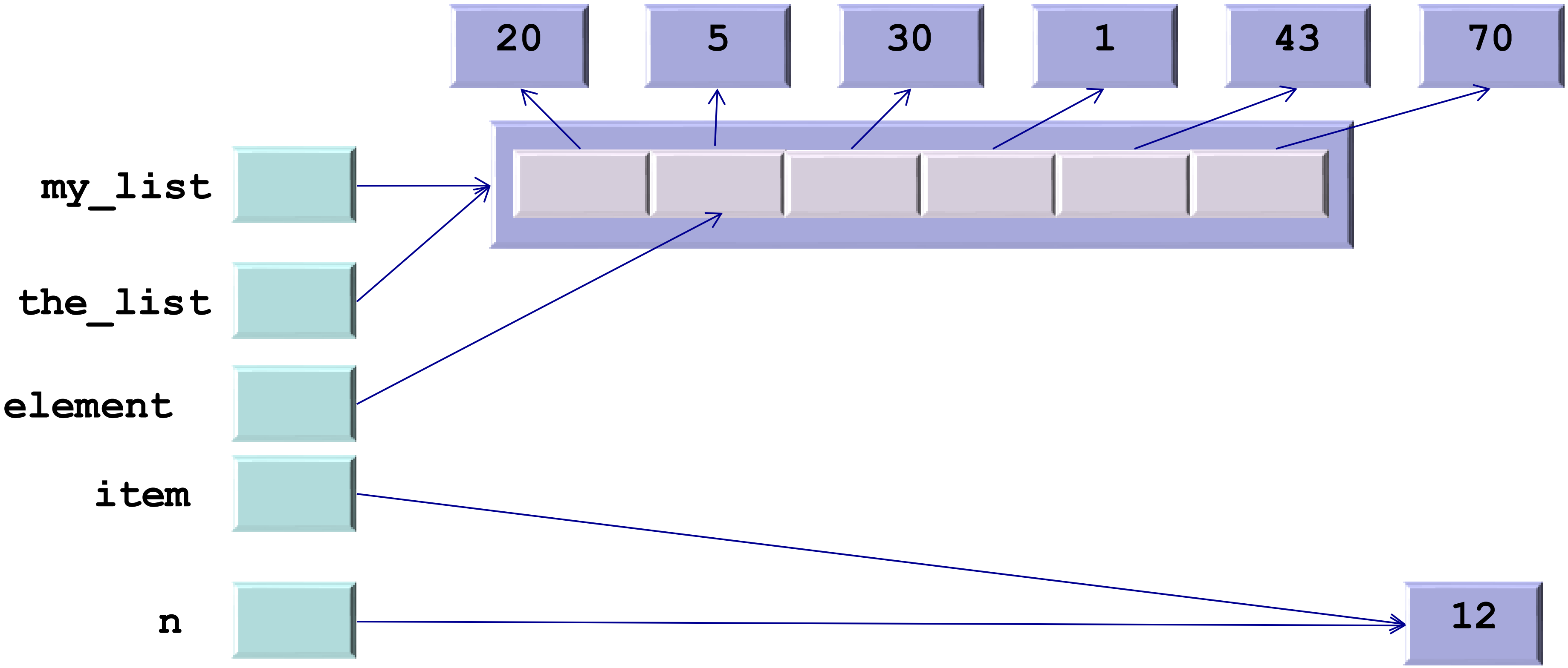



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

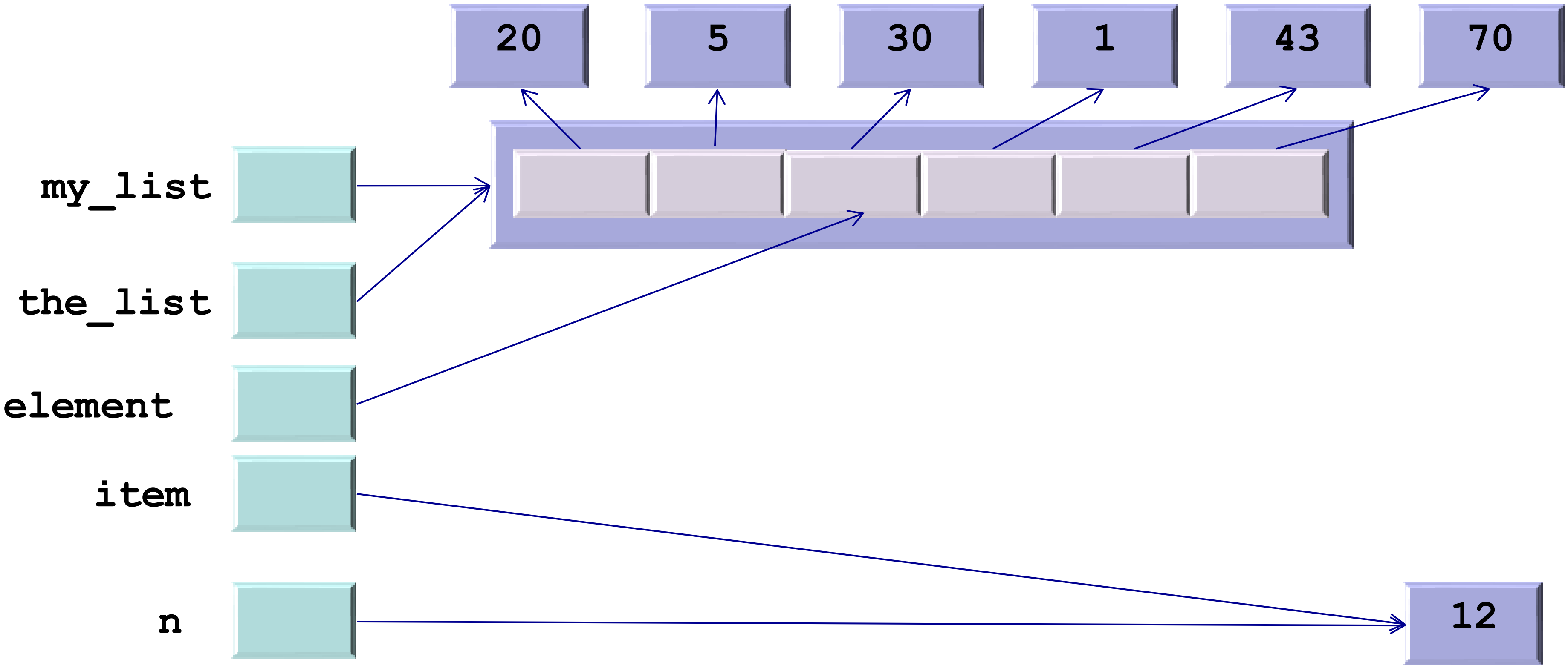



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

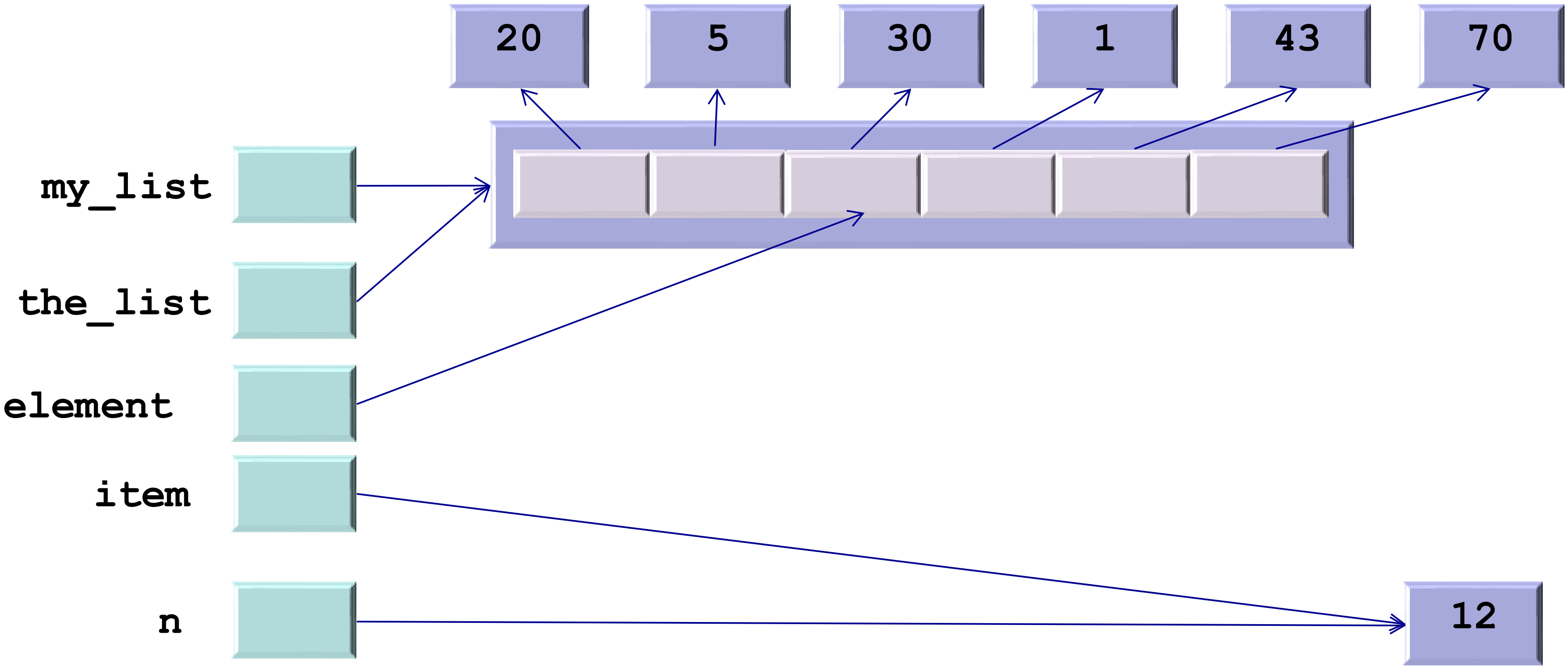



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

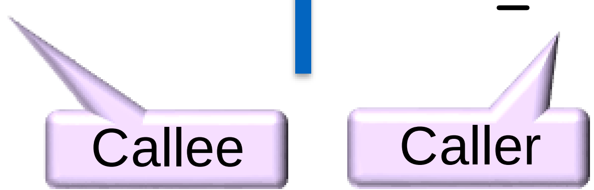
Callee

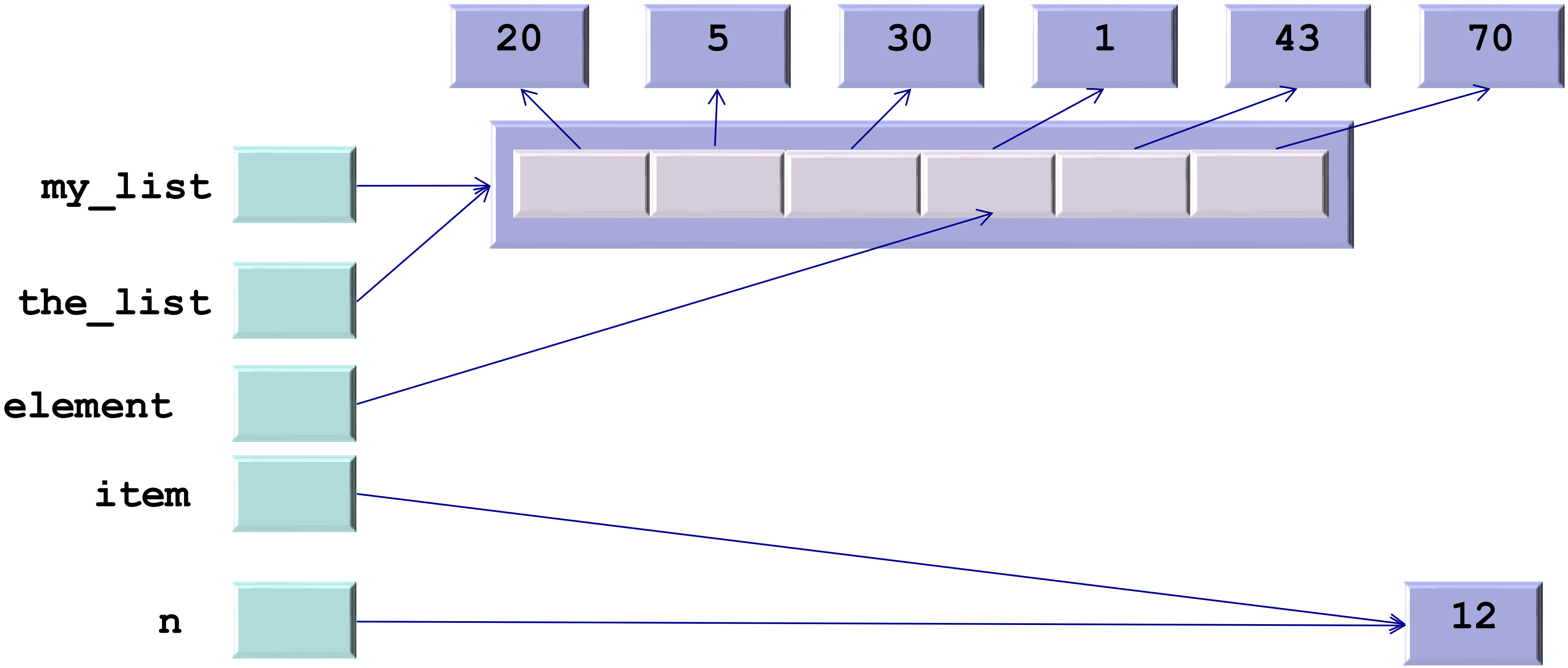
Caller




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

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```



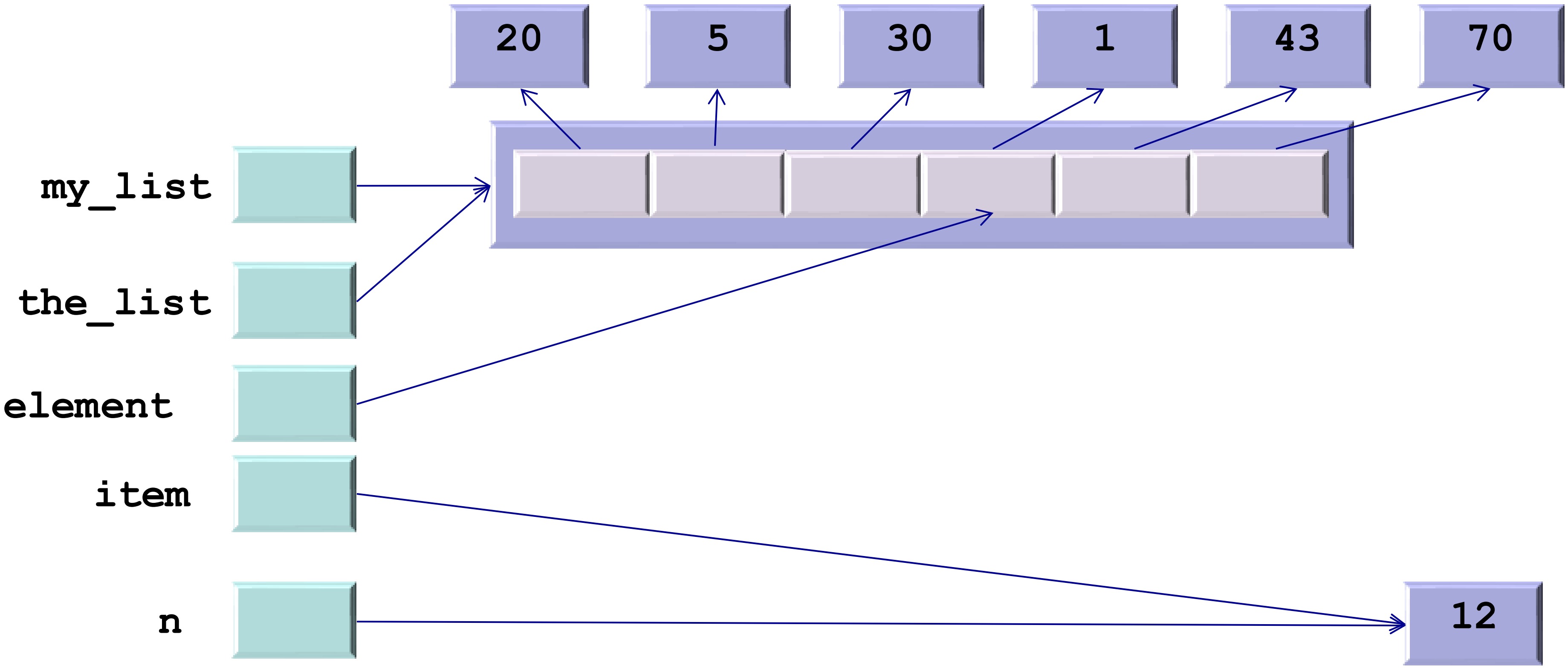



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

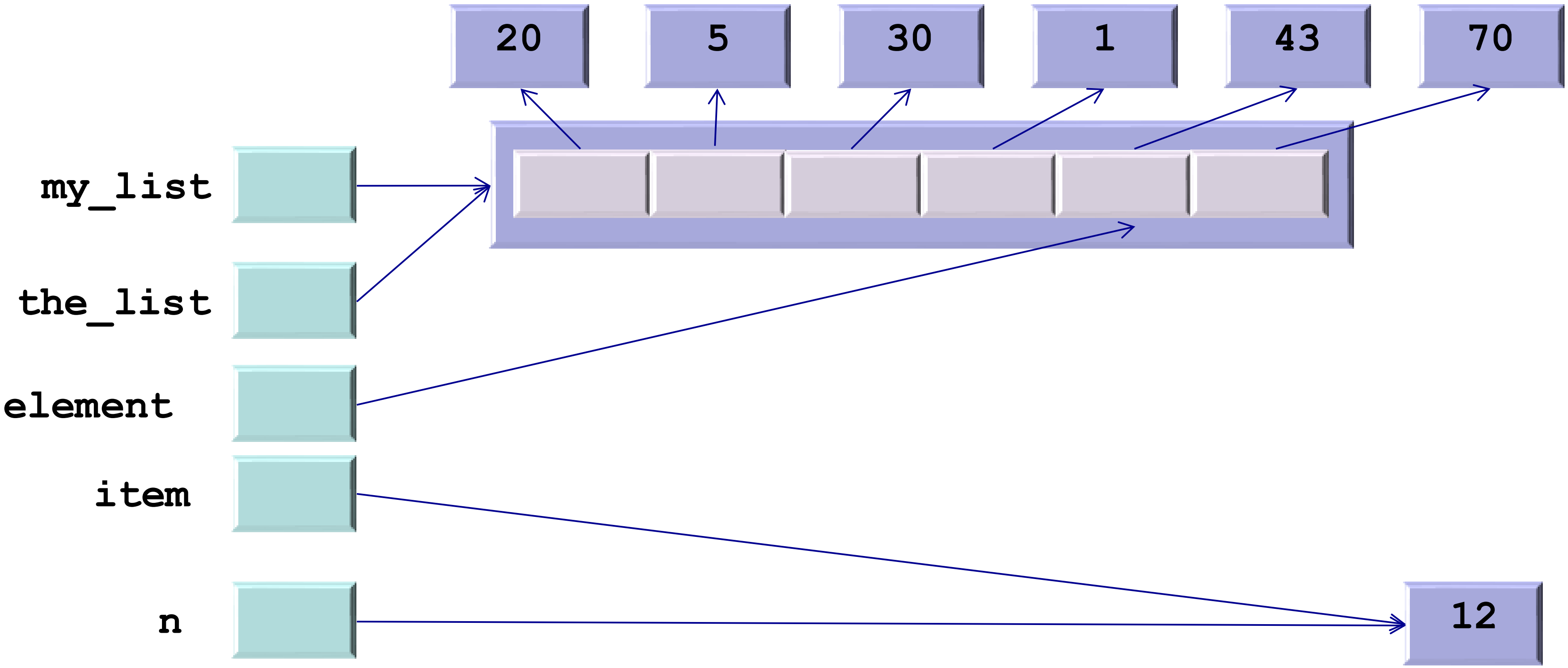



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

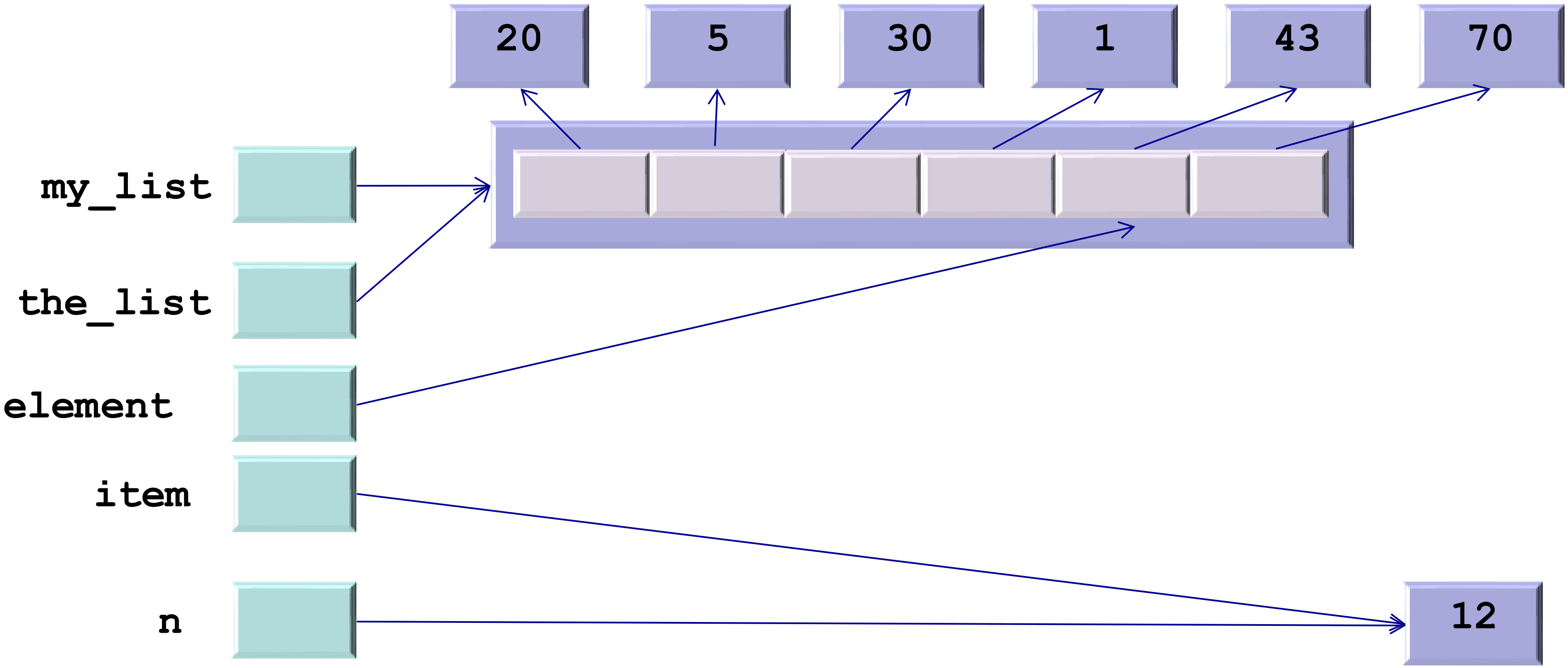



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

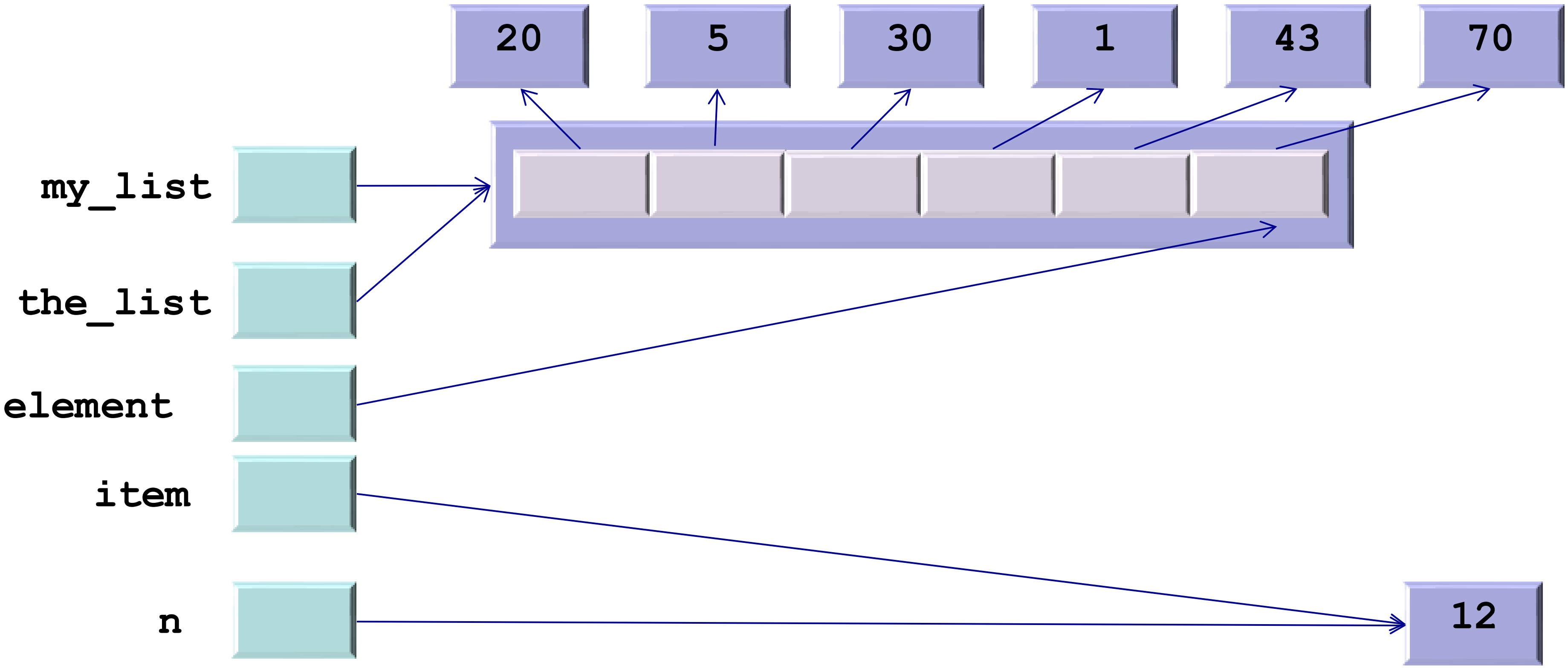



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

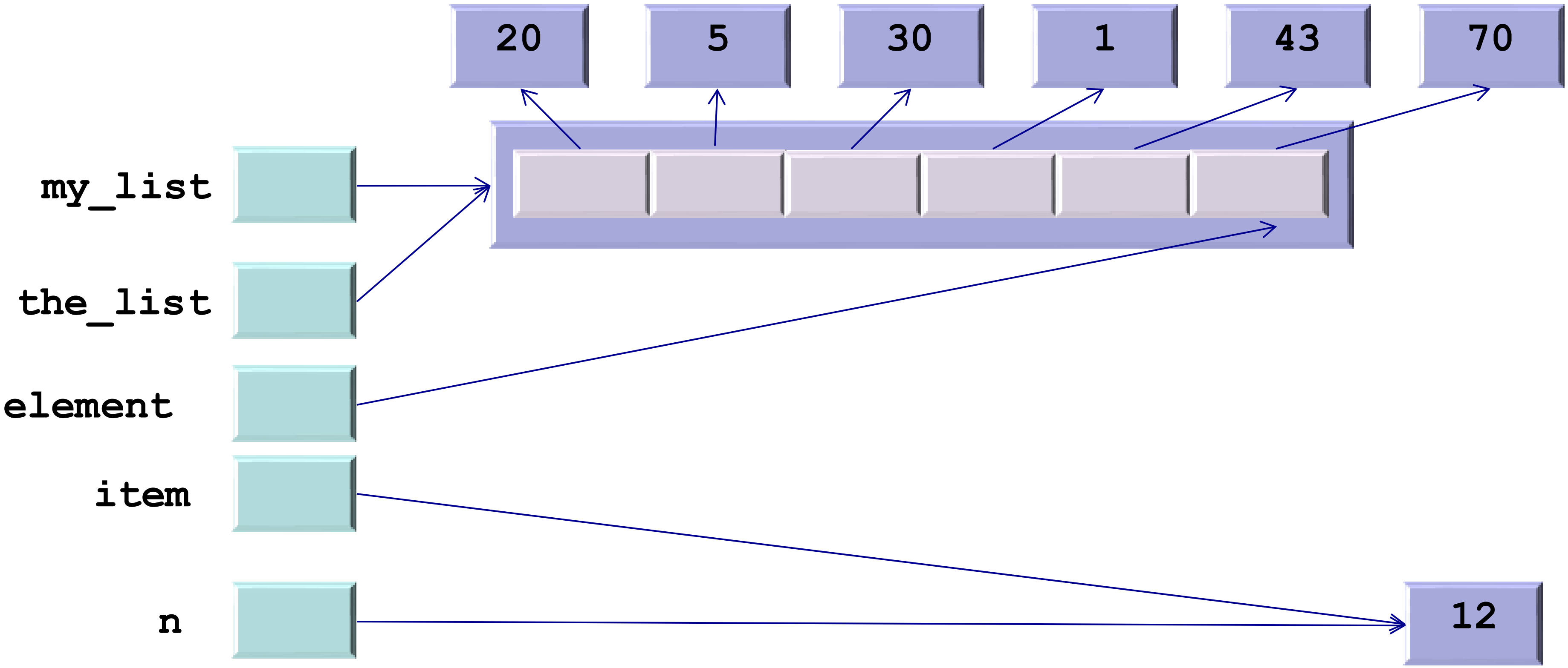



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

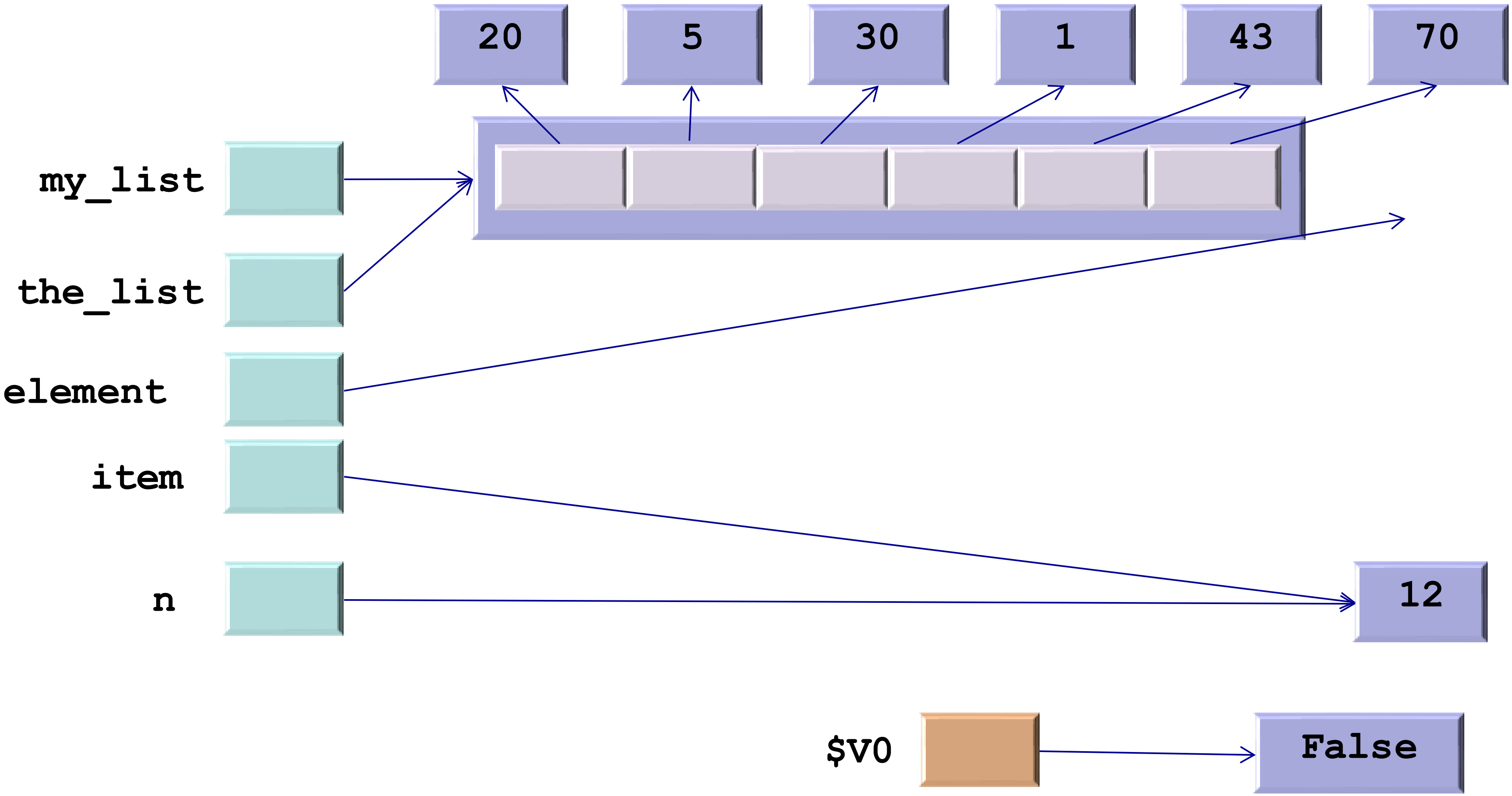



```
def lin_search(the_list, item):
    for element in the_list:
        if item == element:
            return True
    return False
```

Callee

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Caller




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

```
my_list = [20,5,30,1,43,70]
n = 12
lin_search(my_list, n)
```

Callee

Caller

Time Complexity

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

? times

Access is constant K1

Comparison we don't know m

Return is constant K2

Return is constant K3

We say comparison is O(m), where m depends on the size of what you are comparing. For integers m=1, 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.

Best ≠ Worst

Some elements get a certain amount of processing

Other elements are not processed at all

Time complexity for Linear Search

- **Best case?**
  - Loop stops in the first iteration
  - When? The wanted item is at the start of the list
    - $K1 + m + K2 \rightarrow O(m)$
- **Worst case?**
  - Loop goes all the way (n times, if n is the length of the list)
  - When? The wanted item is not found
    - $(K1+m)*n + K3 \rightarrow O(m*n)$

```
def lin_search(the_list, item):
    for element in the_list:        Access is constant K1
        if item == element:         Comparison we don't know m
            return True             Return is constant K2
    return False                    Return is constant K3
```

? times

What about linear search in sorted lists?

- **Can we use the same implementation?**

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

- **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?**
  – If we find an element that is greater than `item`

What about linear search in sorted lists?

- **Let's modify the code to work on sorted lists**
- **One possibility is:**

```
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
```

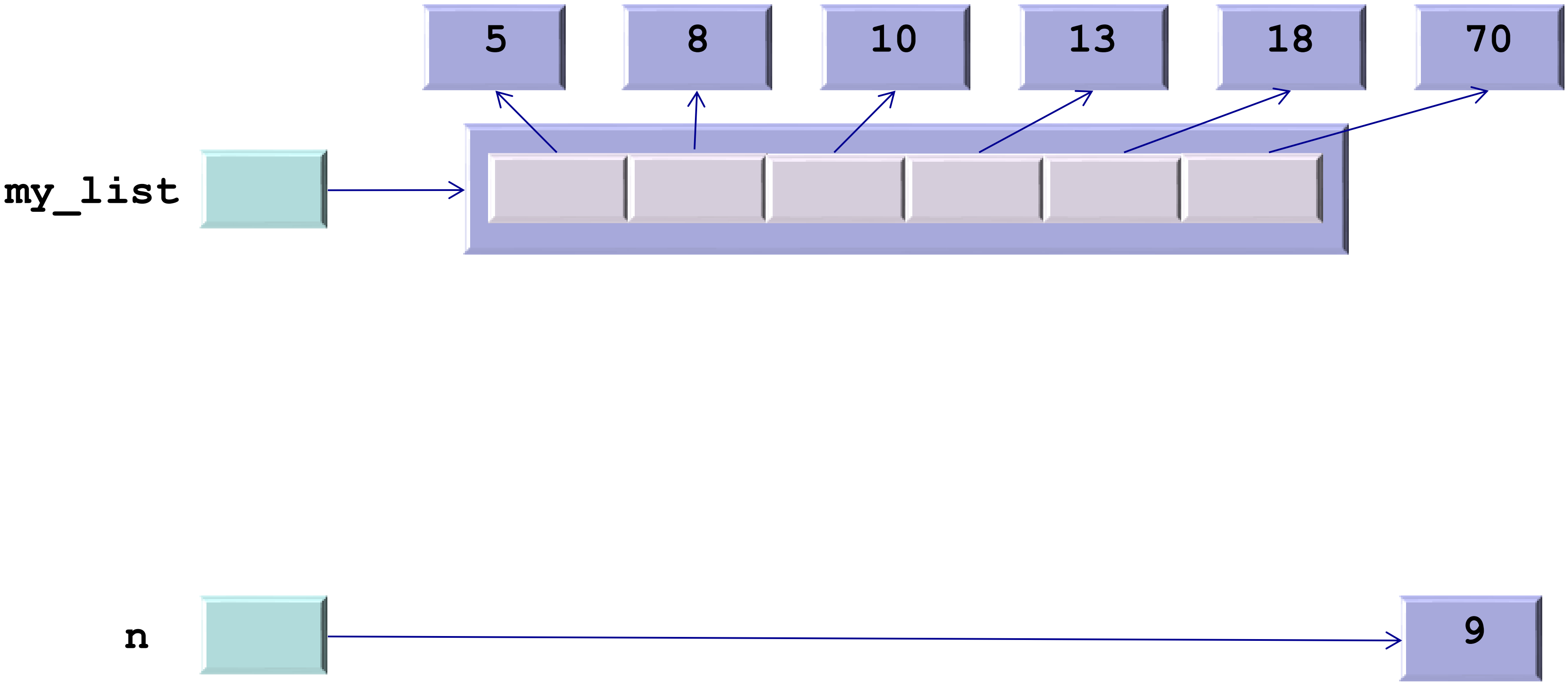
You could also **break** out of the loop at this point and let the outter **return** handle it

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

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

Callee

Caller

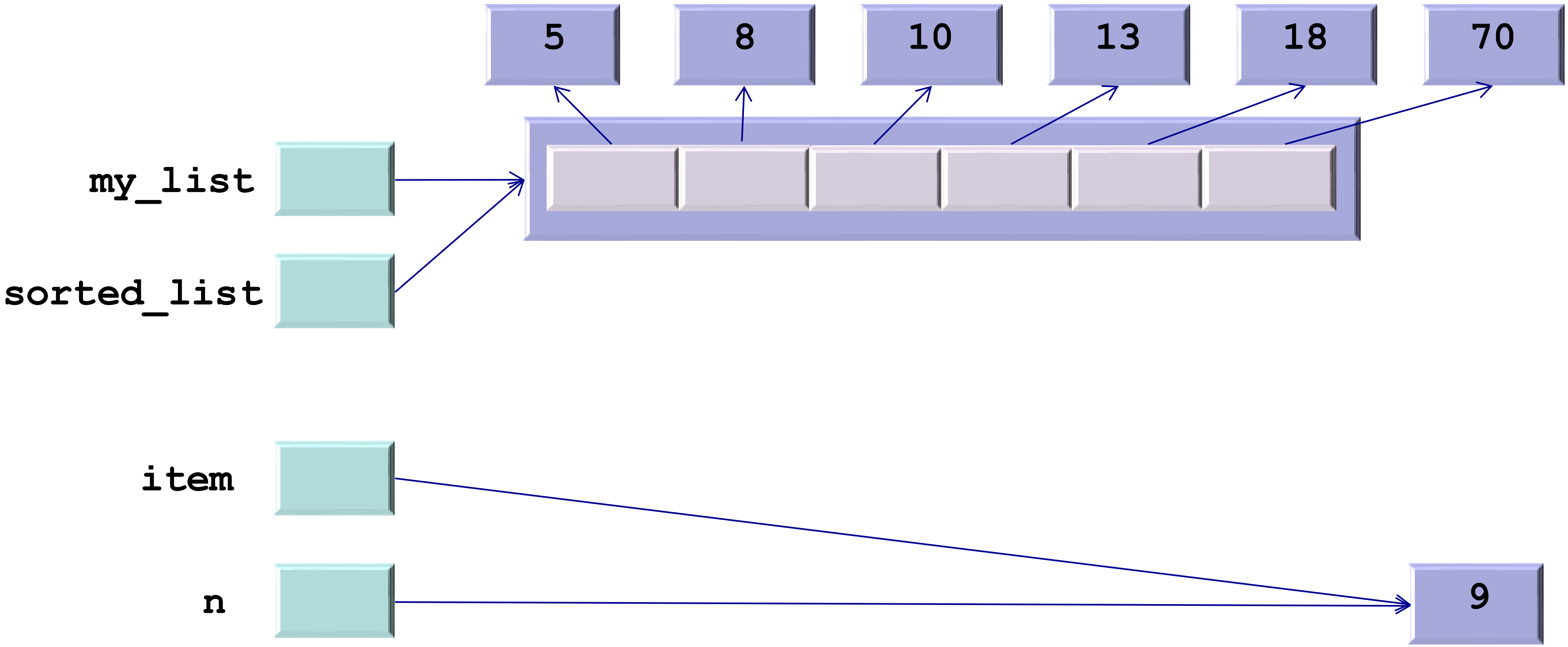



```python
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

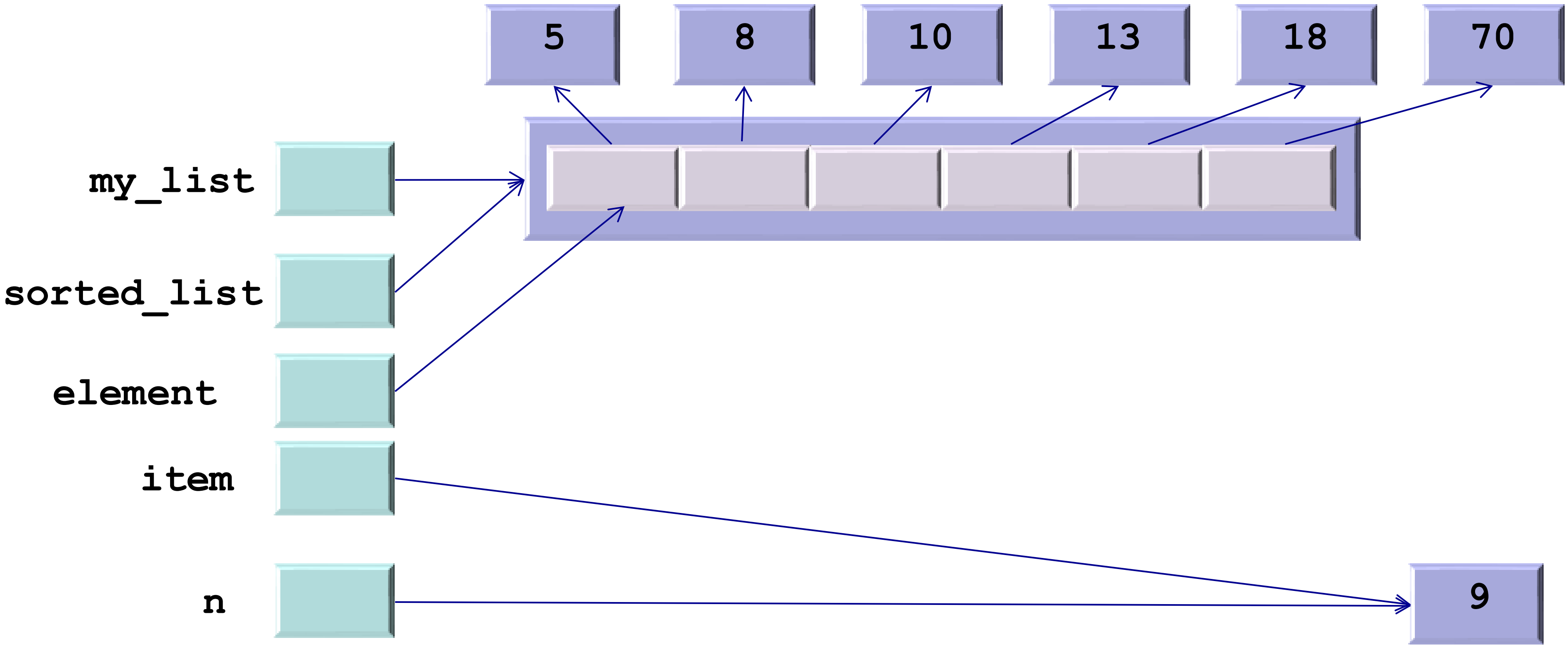



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

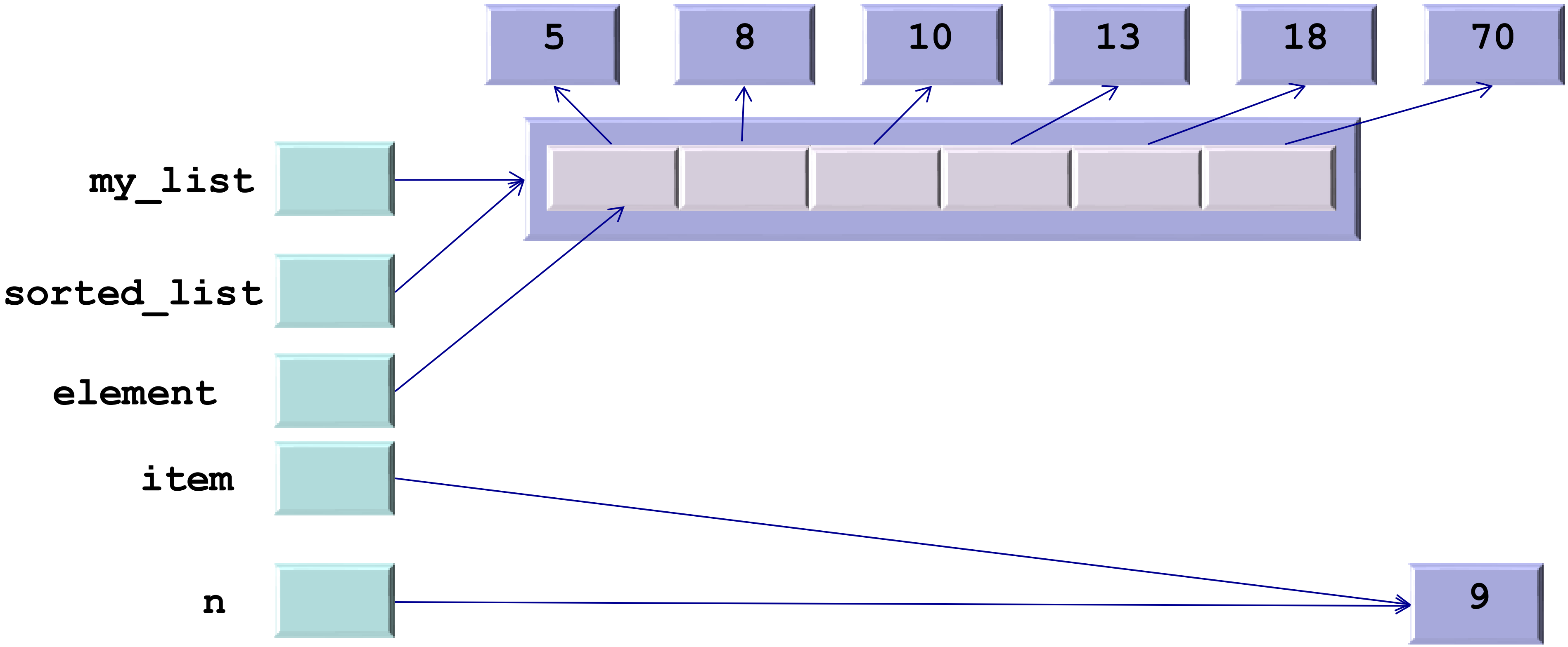



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

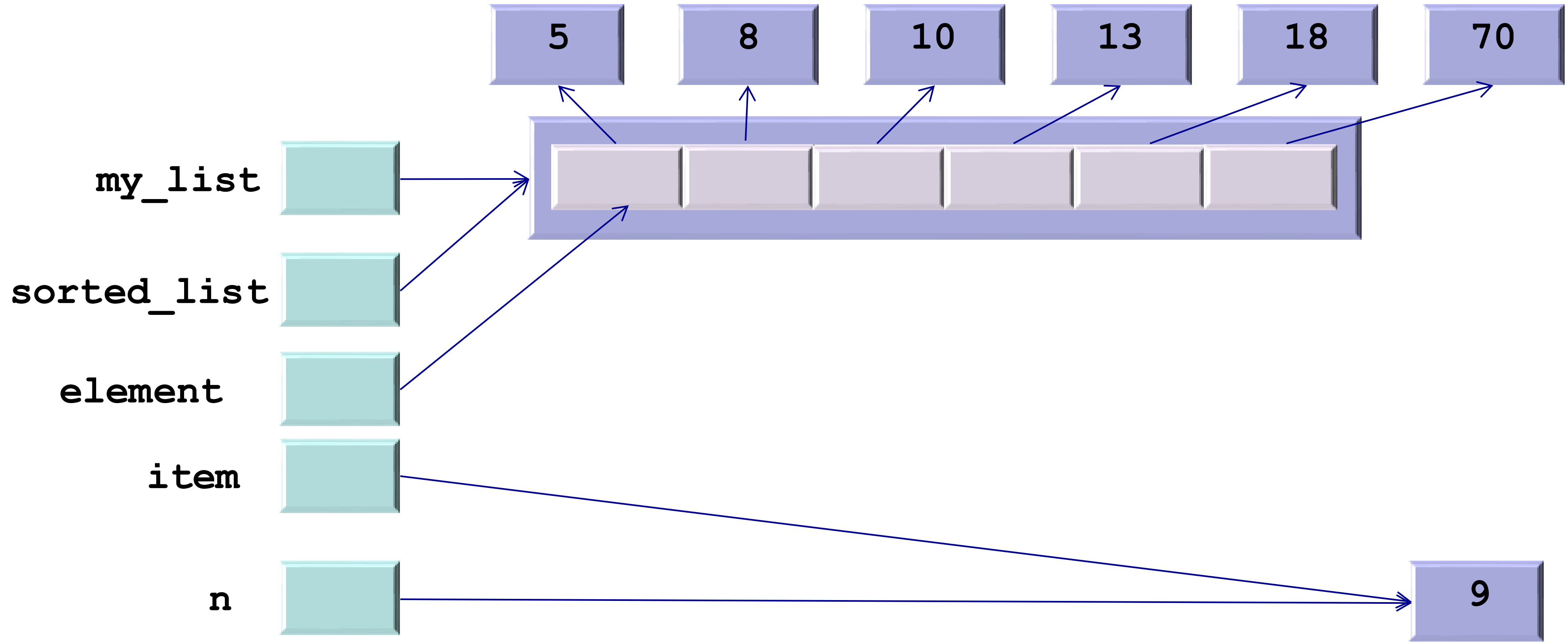



```python
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

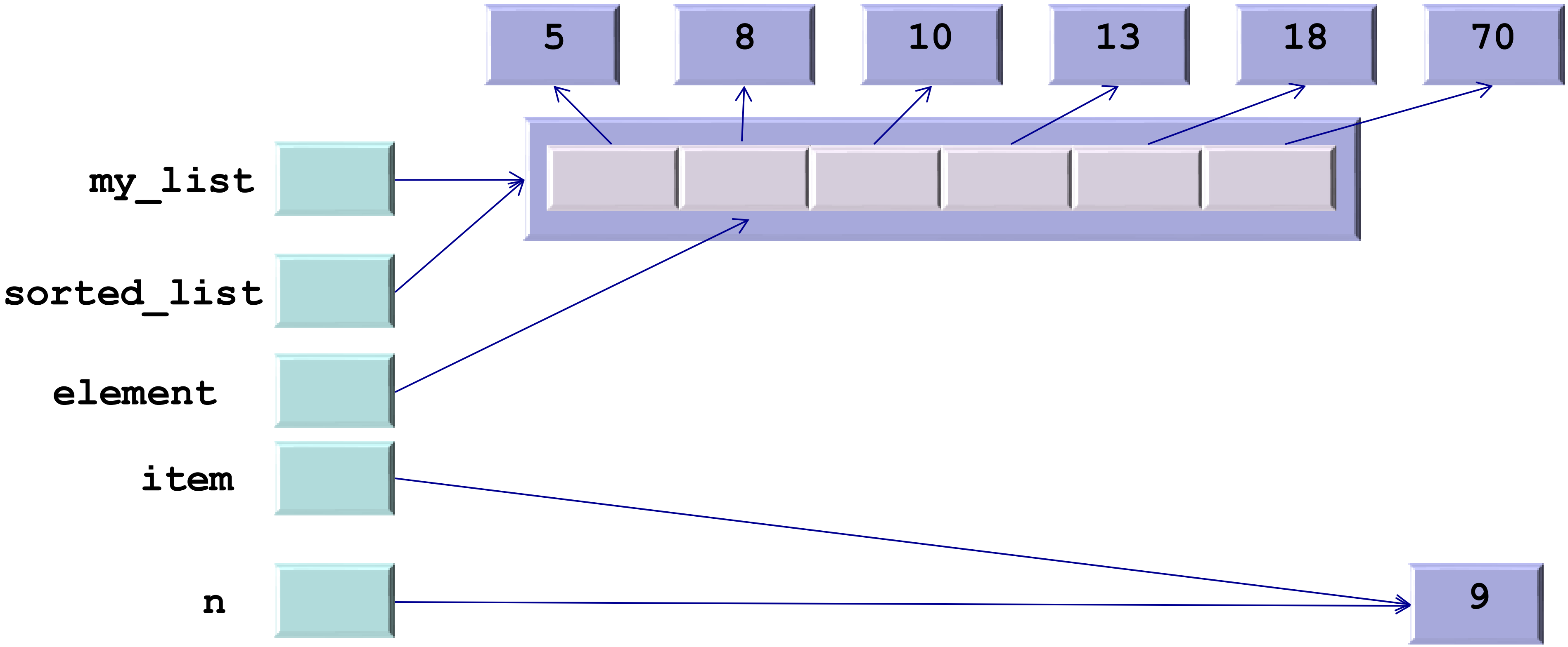



```python
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

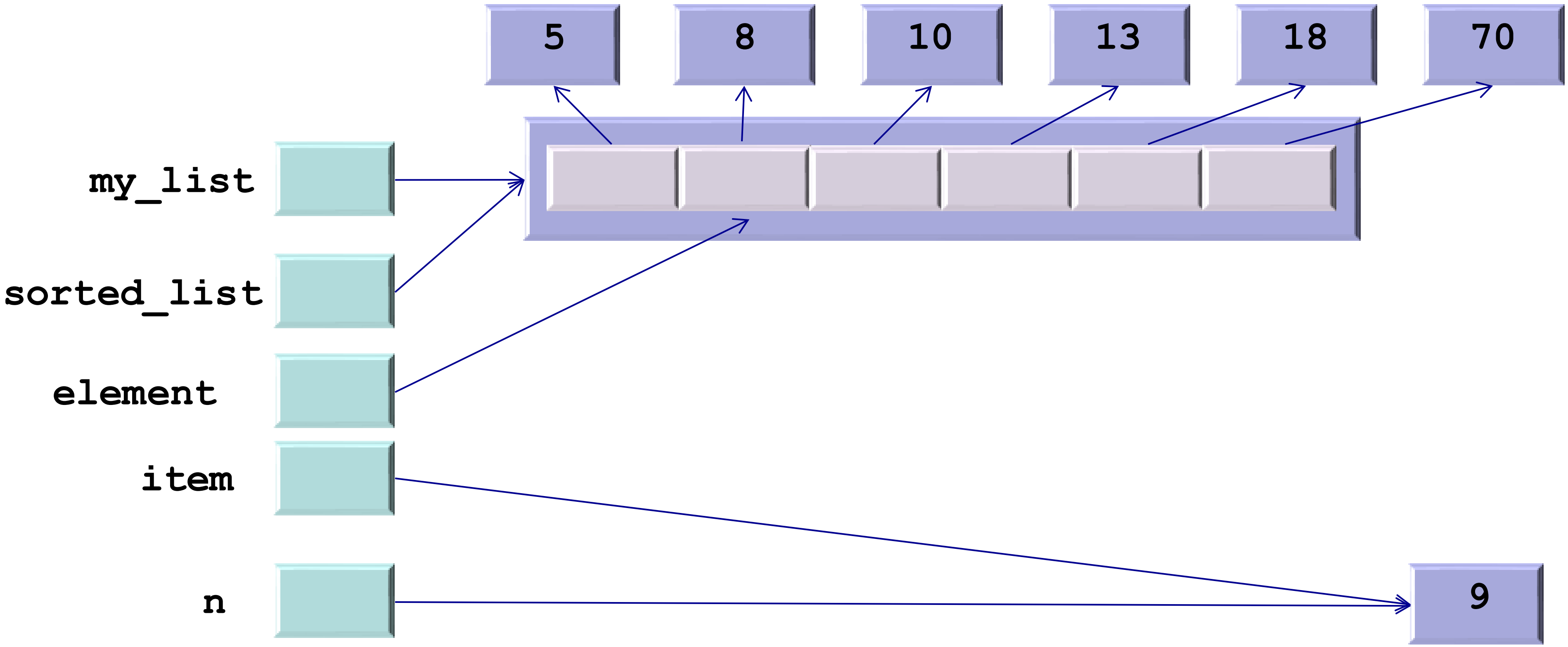



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

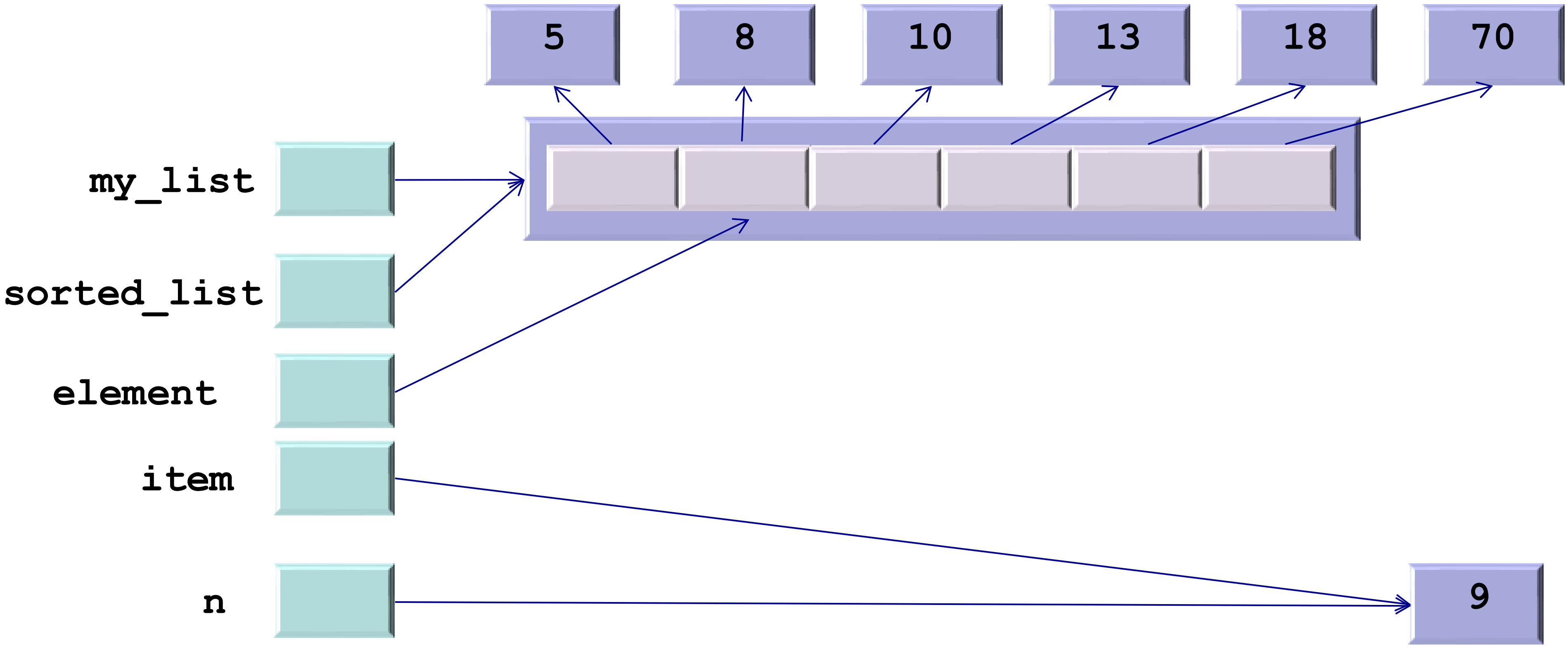



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

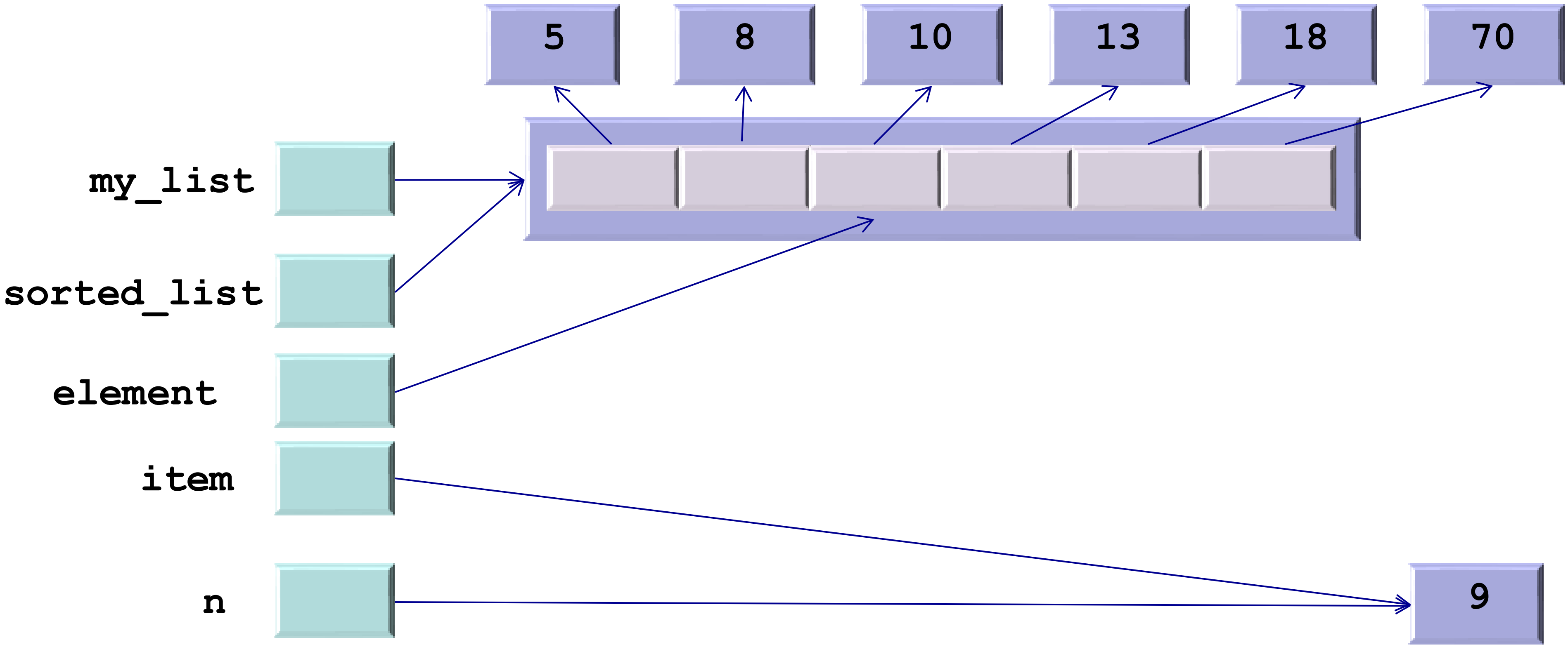



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

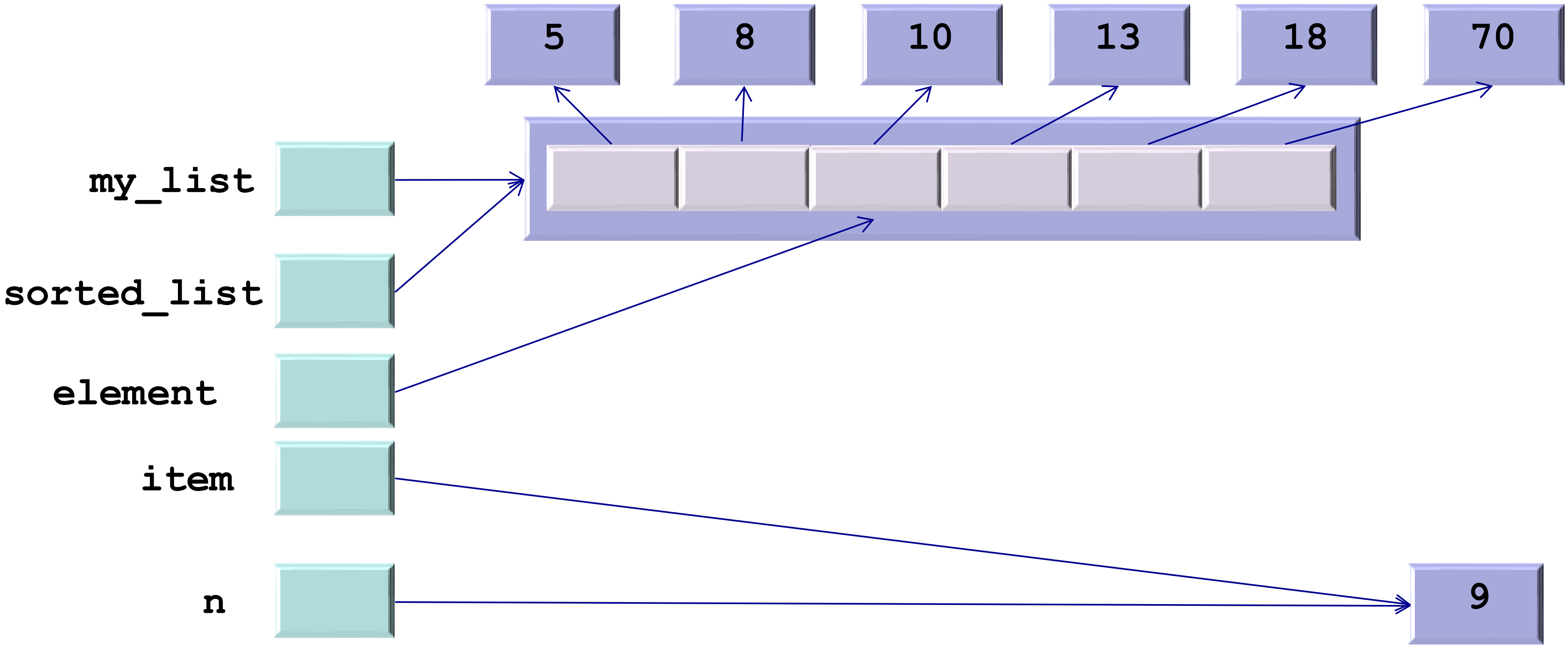



```python
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

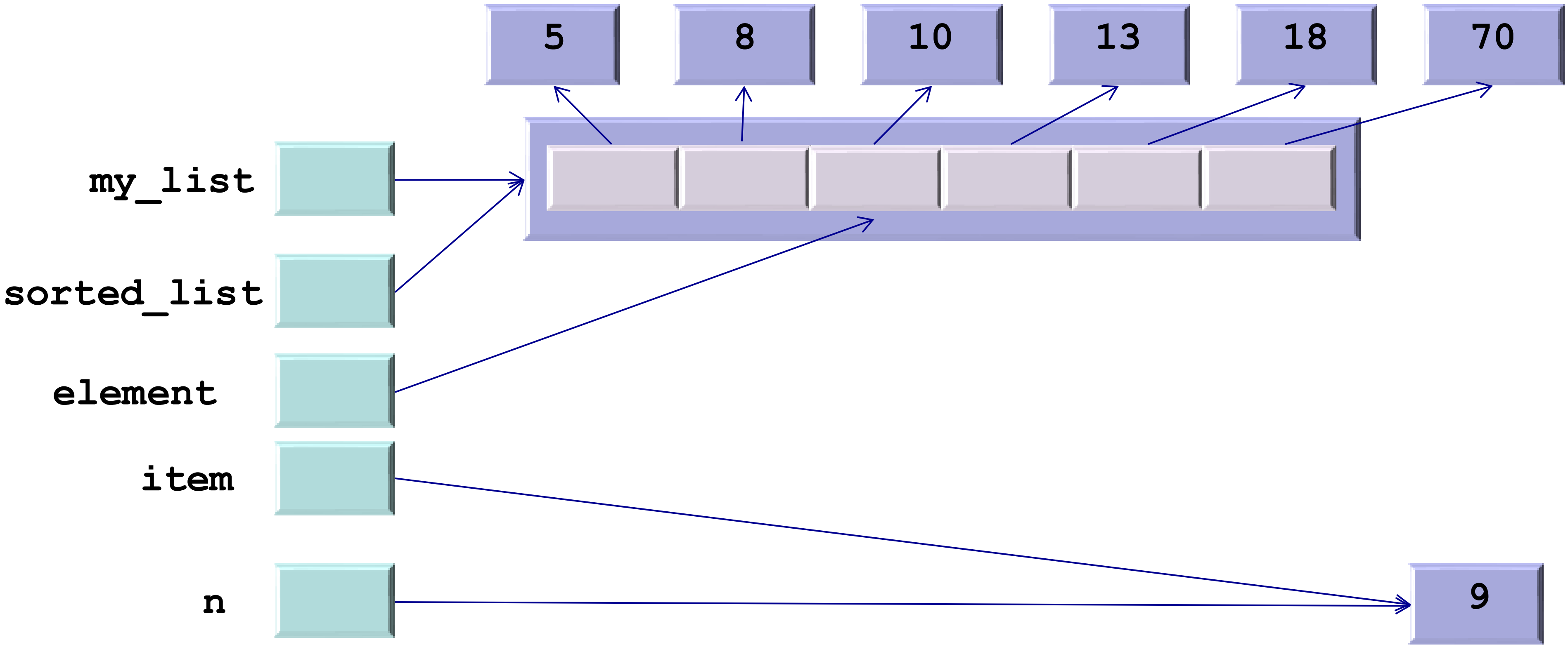



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

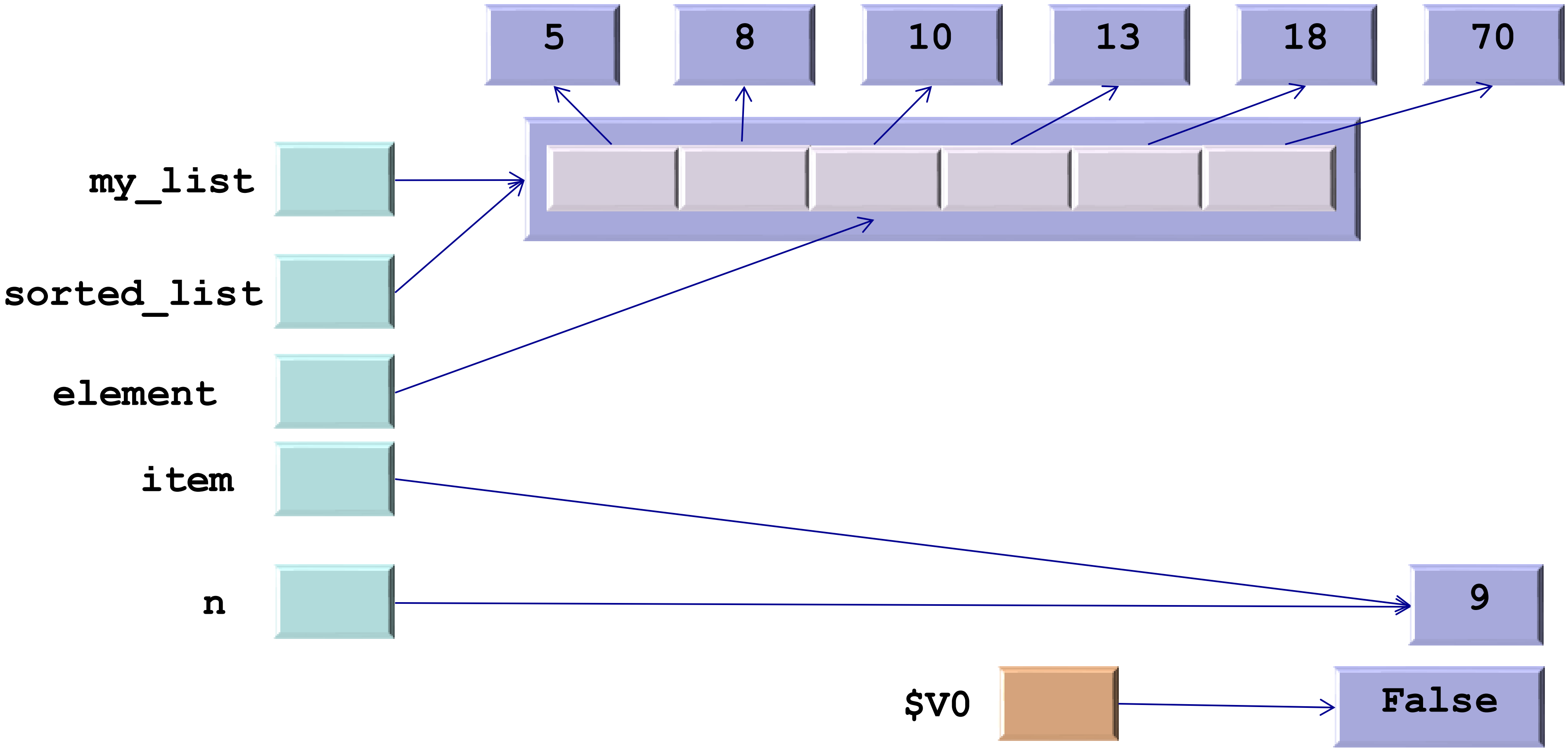



```
def lin_search(sorted_list, item):
    for element in sorted_list:
        if item == element: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

```
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: #found
            return True
        elif item < element:#not in
            return False
    return False #not found
```

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

Callee

Caller

Summary

- **Abstract Data Types**
- **Data Structures**
- **Implementing our own ADTs for**
  - Lists
  - Sorted lists
- **Algorithms, methods and complexity of:**
  - Checking if a list is empty
  - Finding an element is in the list using linear search