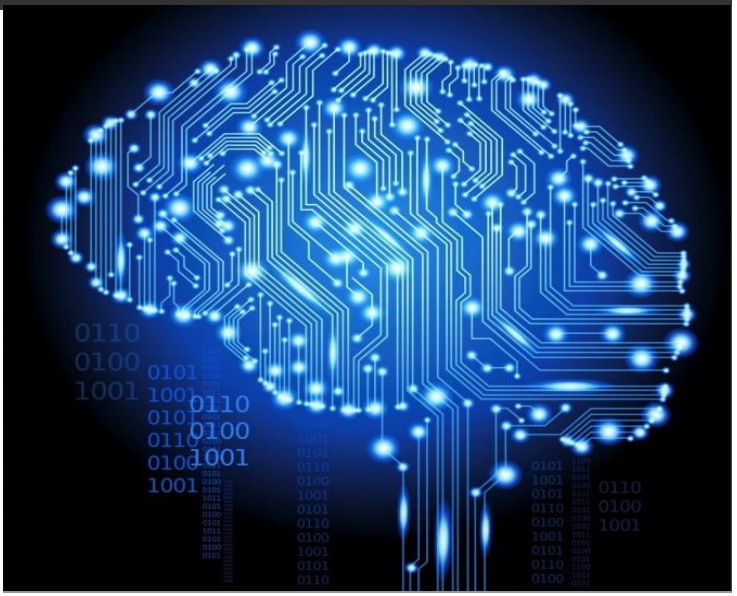
MONASH University

Information Technology

Linked Lists and Iterators

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor



Objectives for today lesson

- **To understand the importance of iterators**
- **To learn how to implement and use them**
- **To learn to make our classes iterable by creating iterators on them**

- **We will probably not have time for a brief look at higher-order programming but, for those interested, have a loot at how to implement functions/methods that:**
  - Receive functions as arguments
  - Return functions

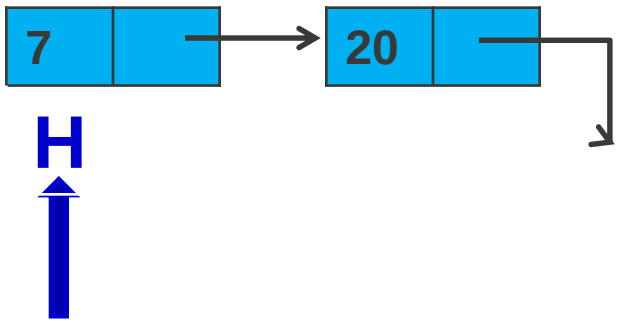
Recap: When to use linked nodes, when arrays for lists?

- **Linked Storage**
  - Unknown list size (no need to resize by copying)
    - If memory is a huge issue though and time is not a problem at all…
  - Many insertions and deletions needed within (rather than at the end)
    - If one does not have direct access to the positions though…
  - Most operations need traversal of the list from the first element
- **Contiguous Storage**
  - Known number of elements (no need for links)
  - Few insertions and deletions needed within (no shuffling)
  - Lots of searches on a sorted list (can use binary search, which is faster)
  - Access to elements by their positions (constant time)

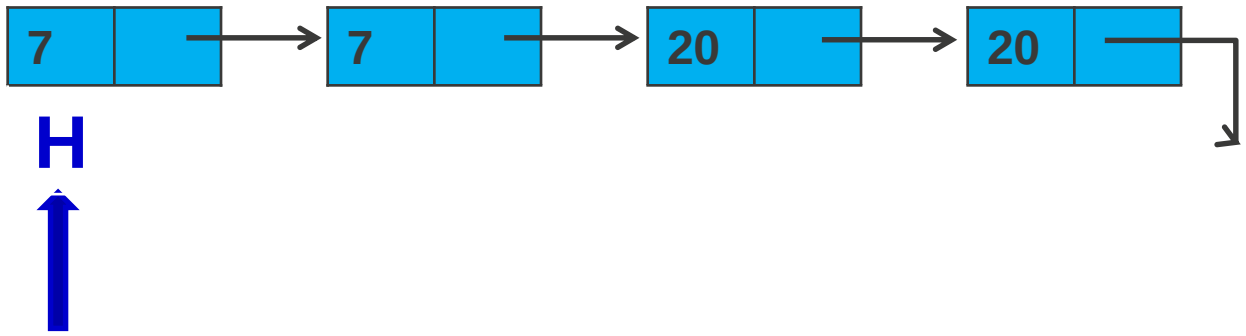
MONASH University

Motivating Iterators

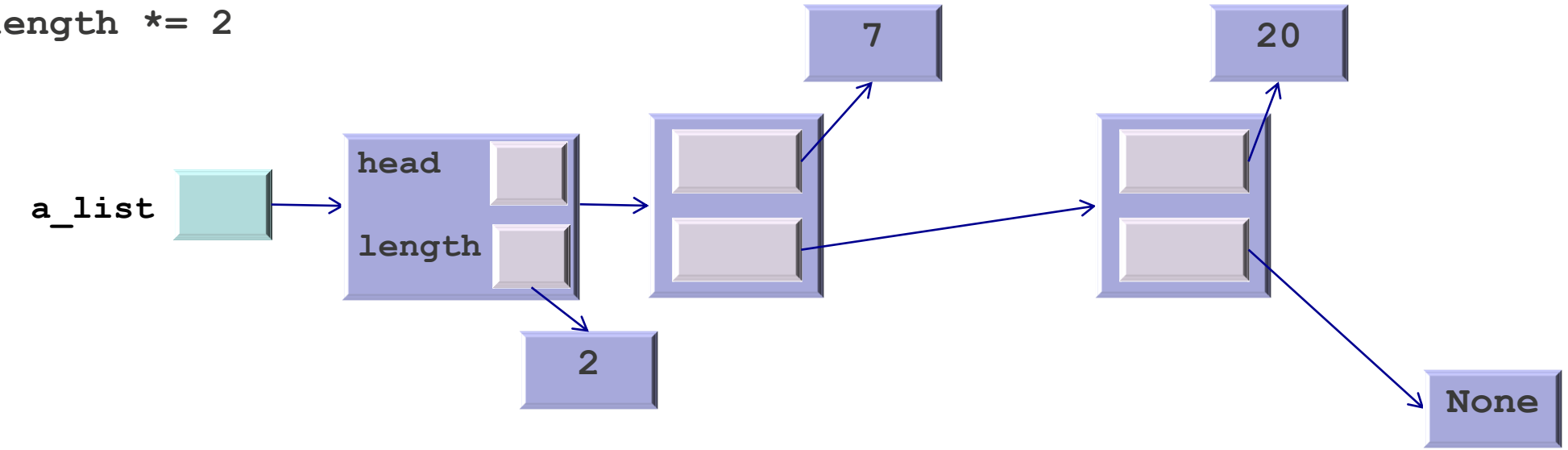
Define a method inside the linked class **`List`** that modifies the list by doubling every node. For example, if **`a_list`** is:



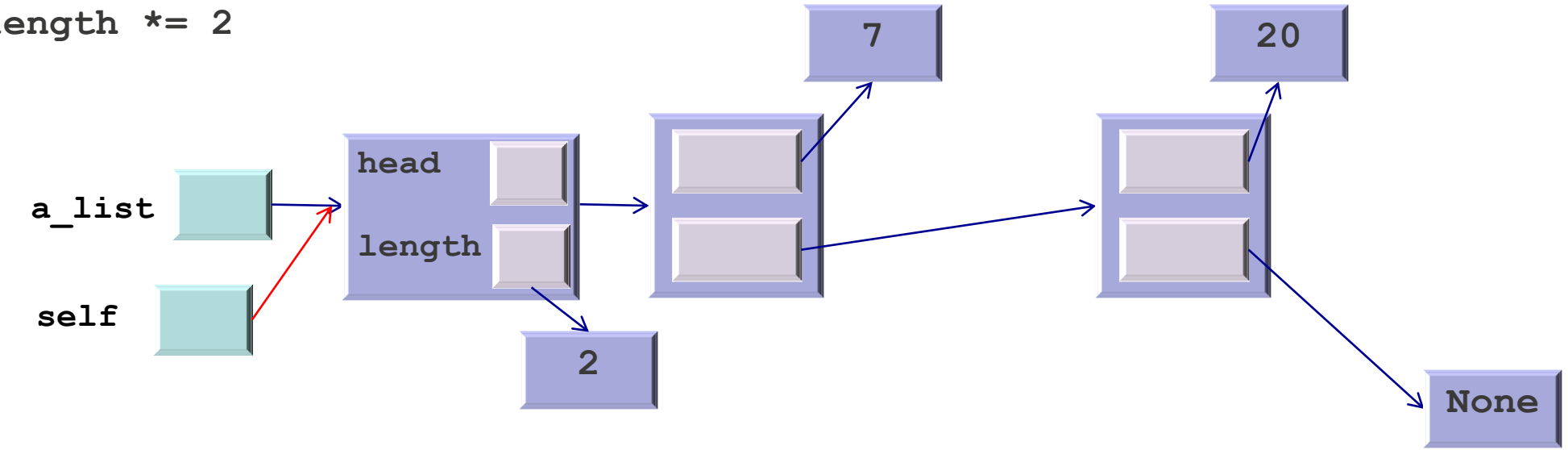
A call to **`a_list.double()`** will



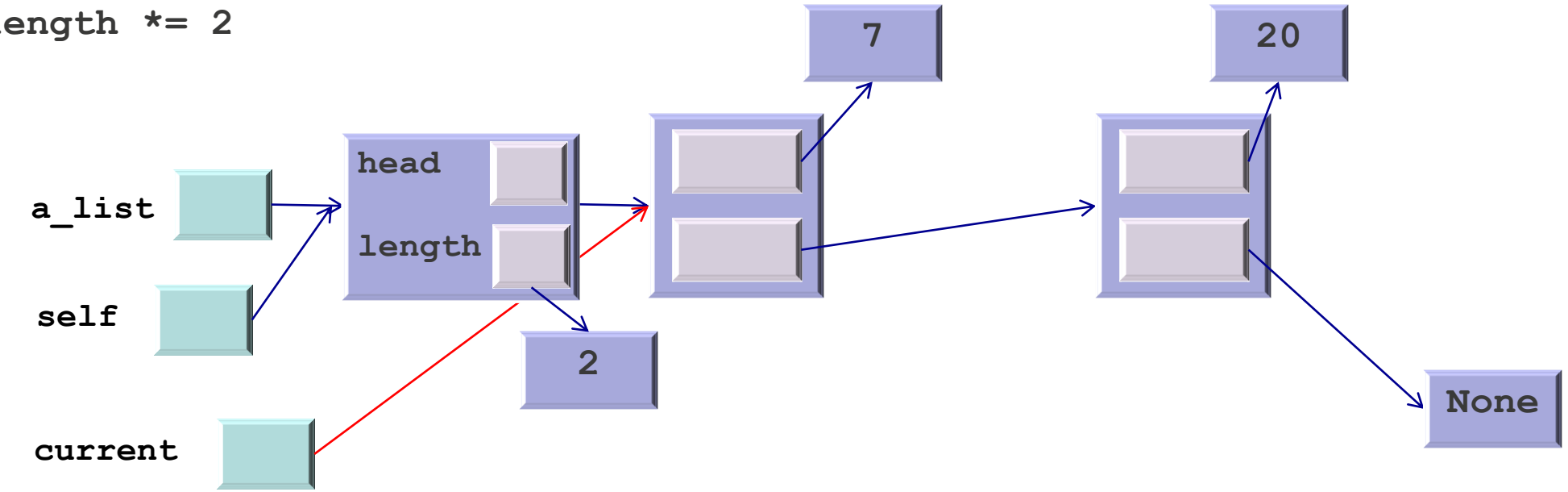
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



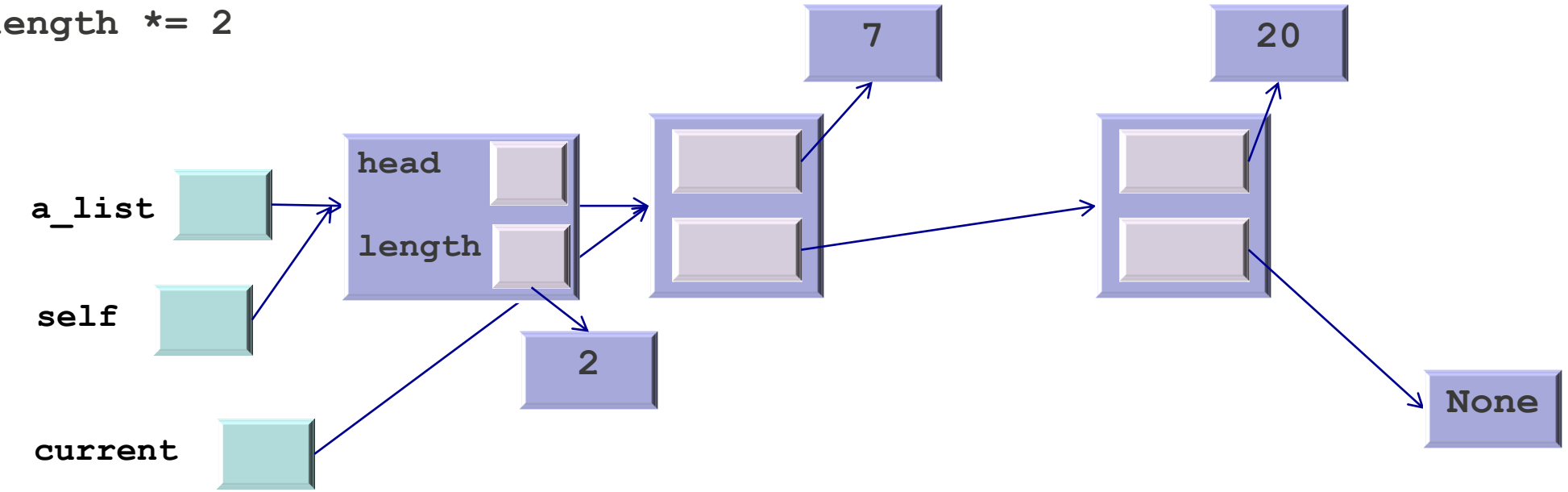
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



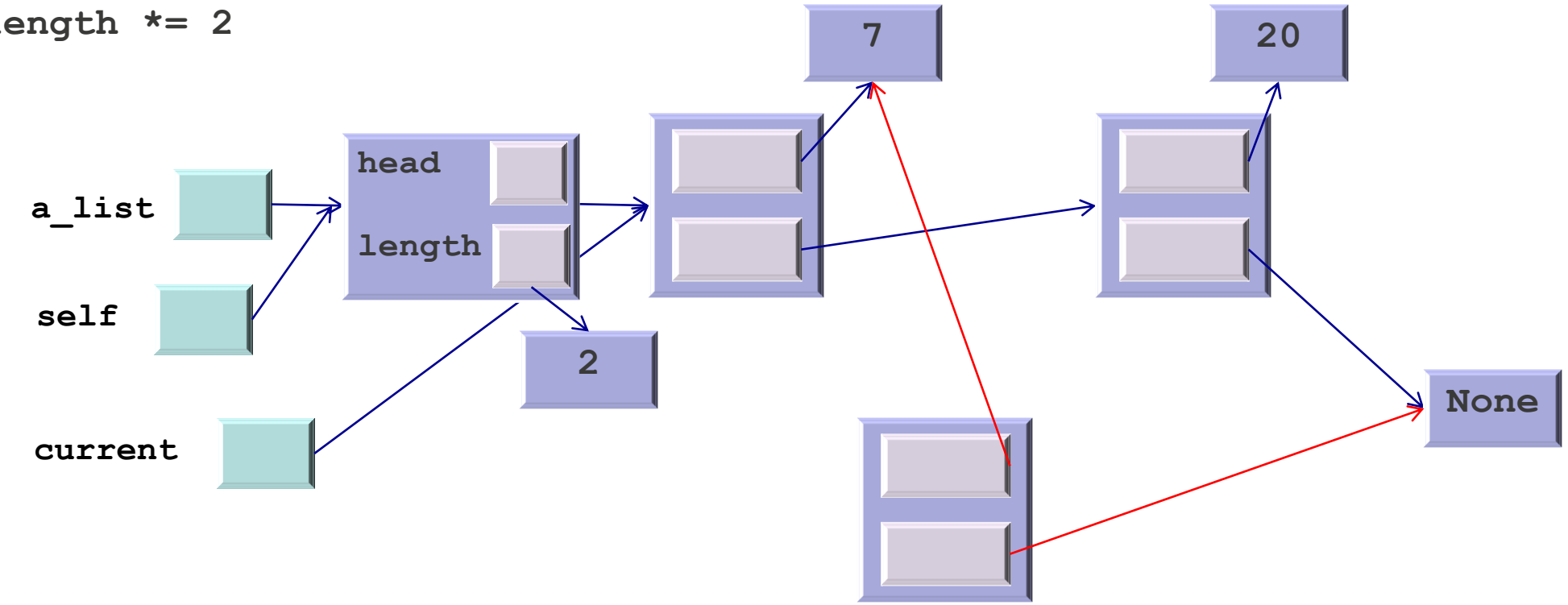
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



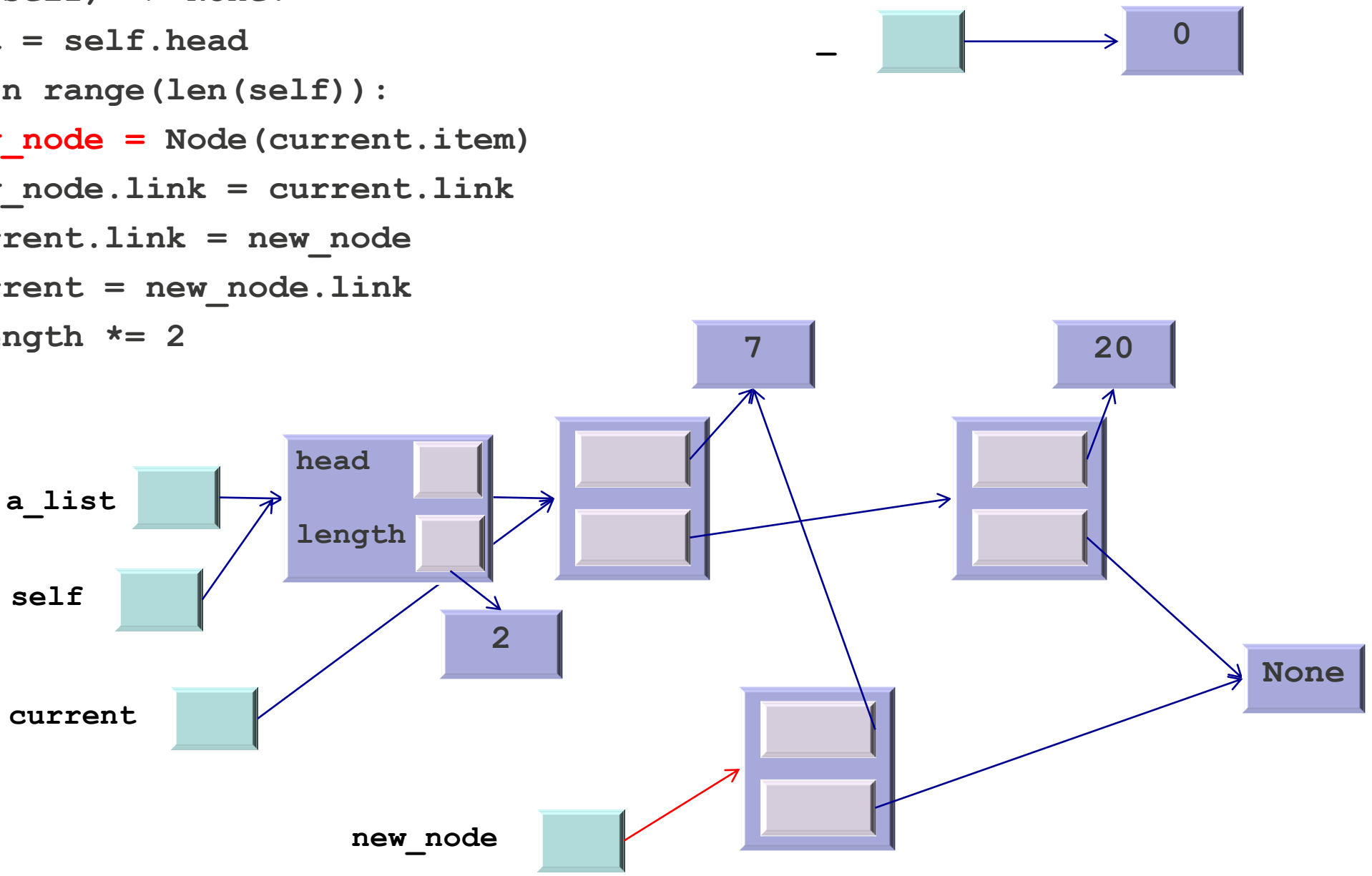
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



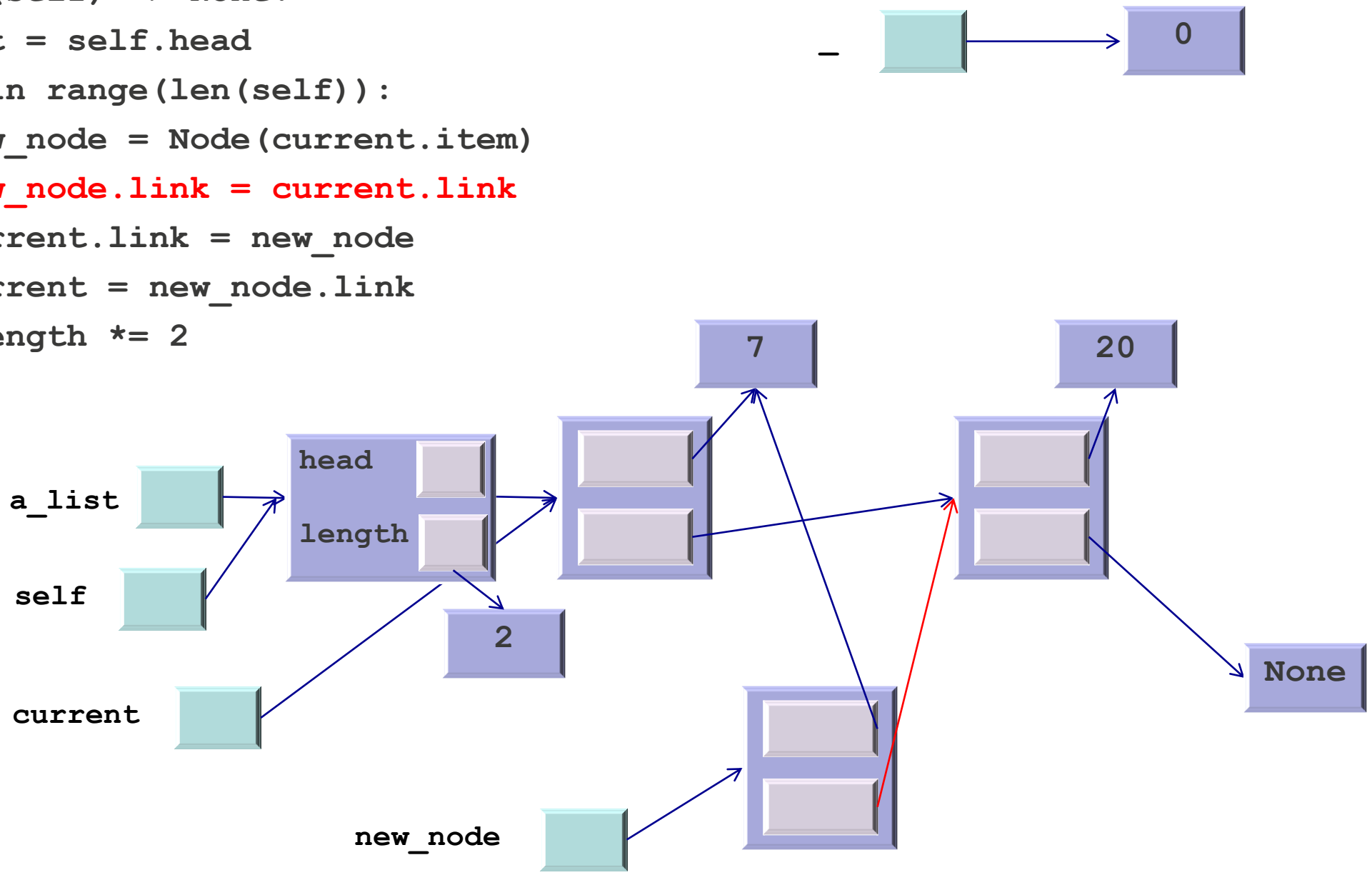
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



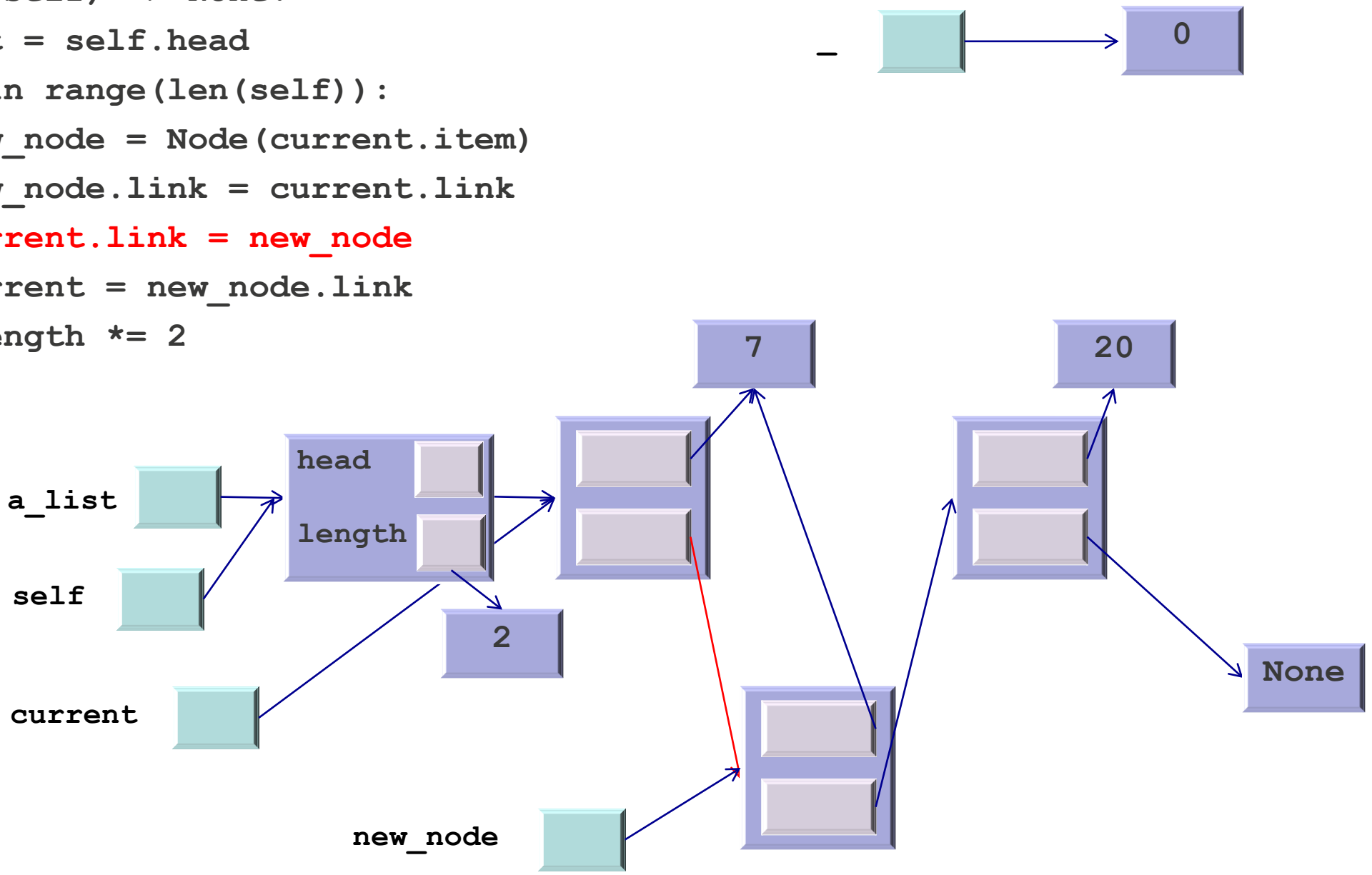
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



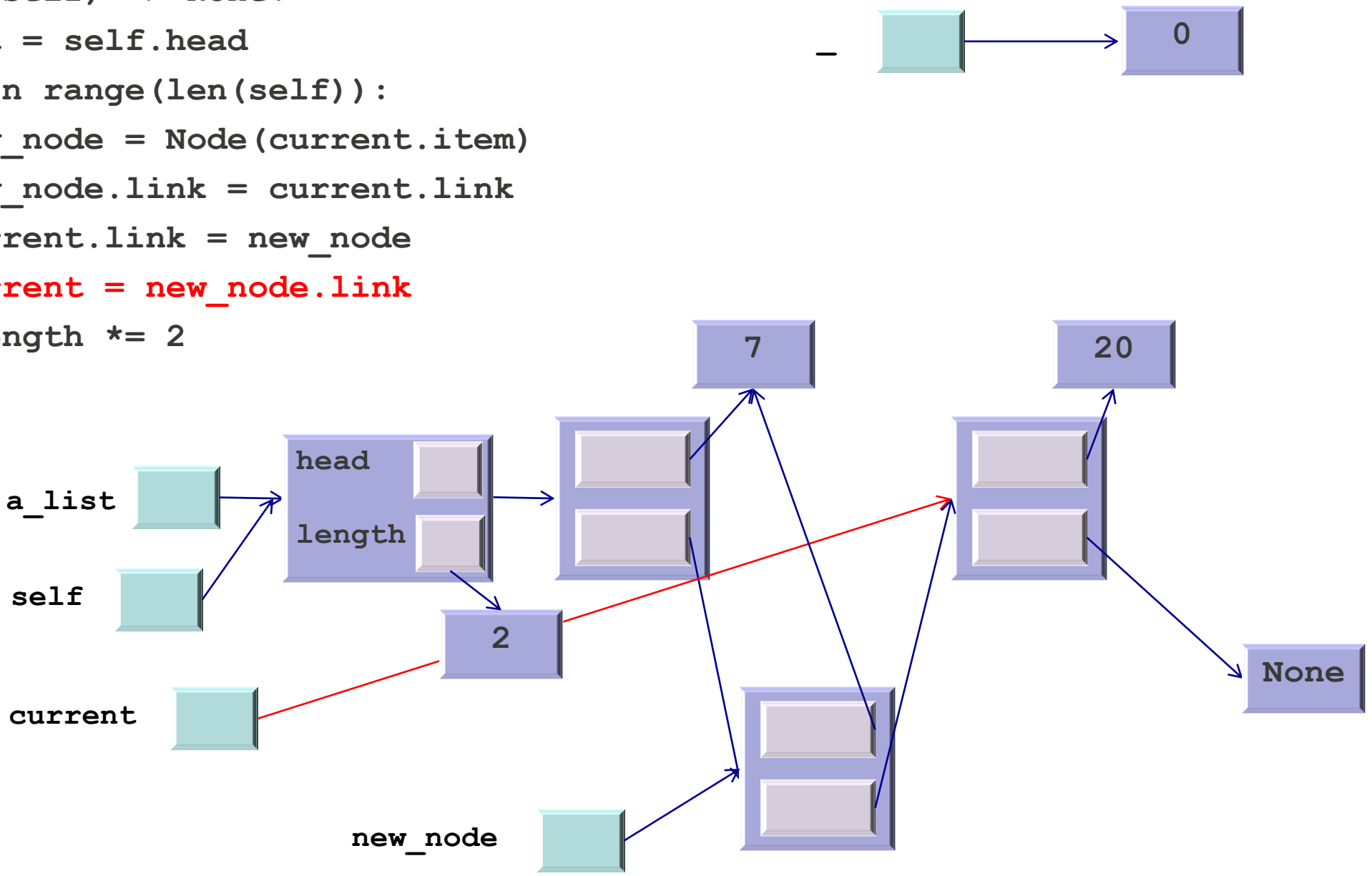
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



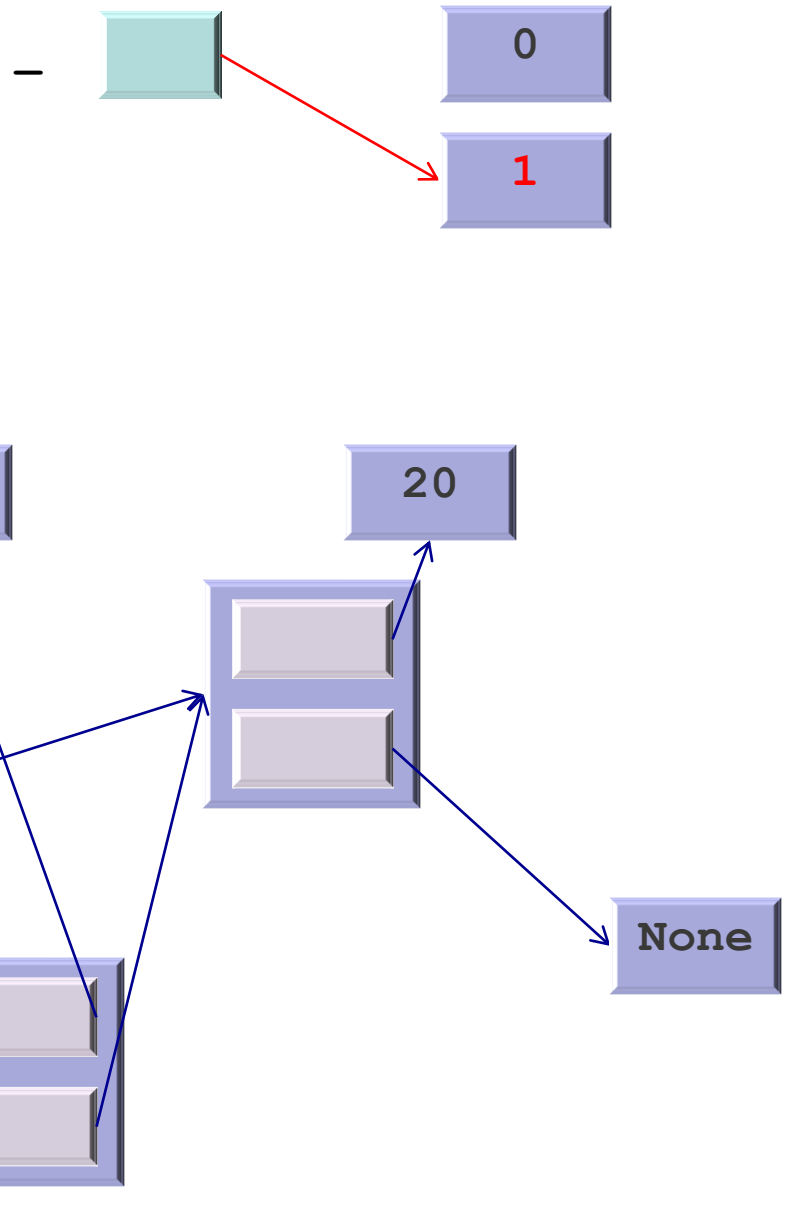
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



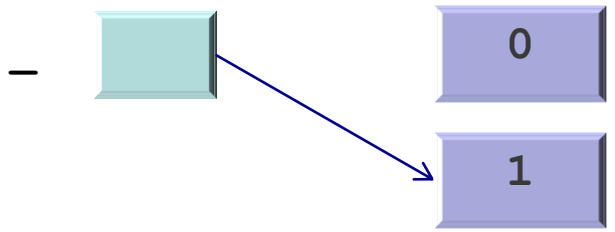
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```

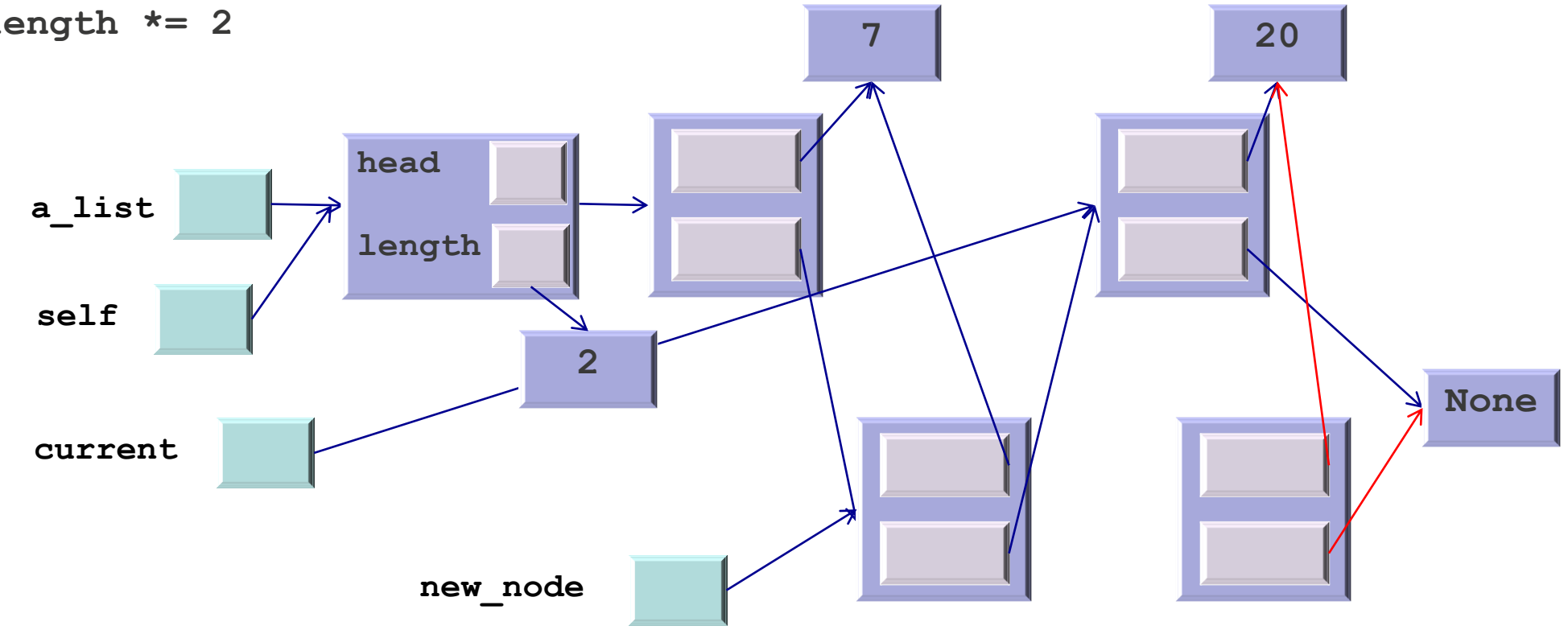


```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```

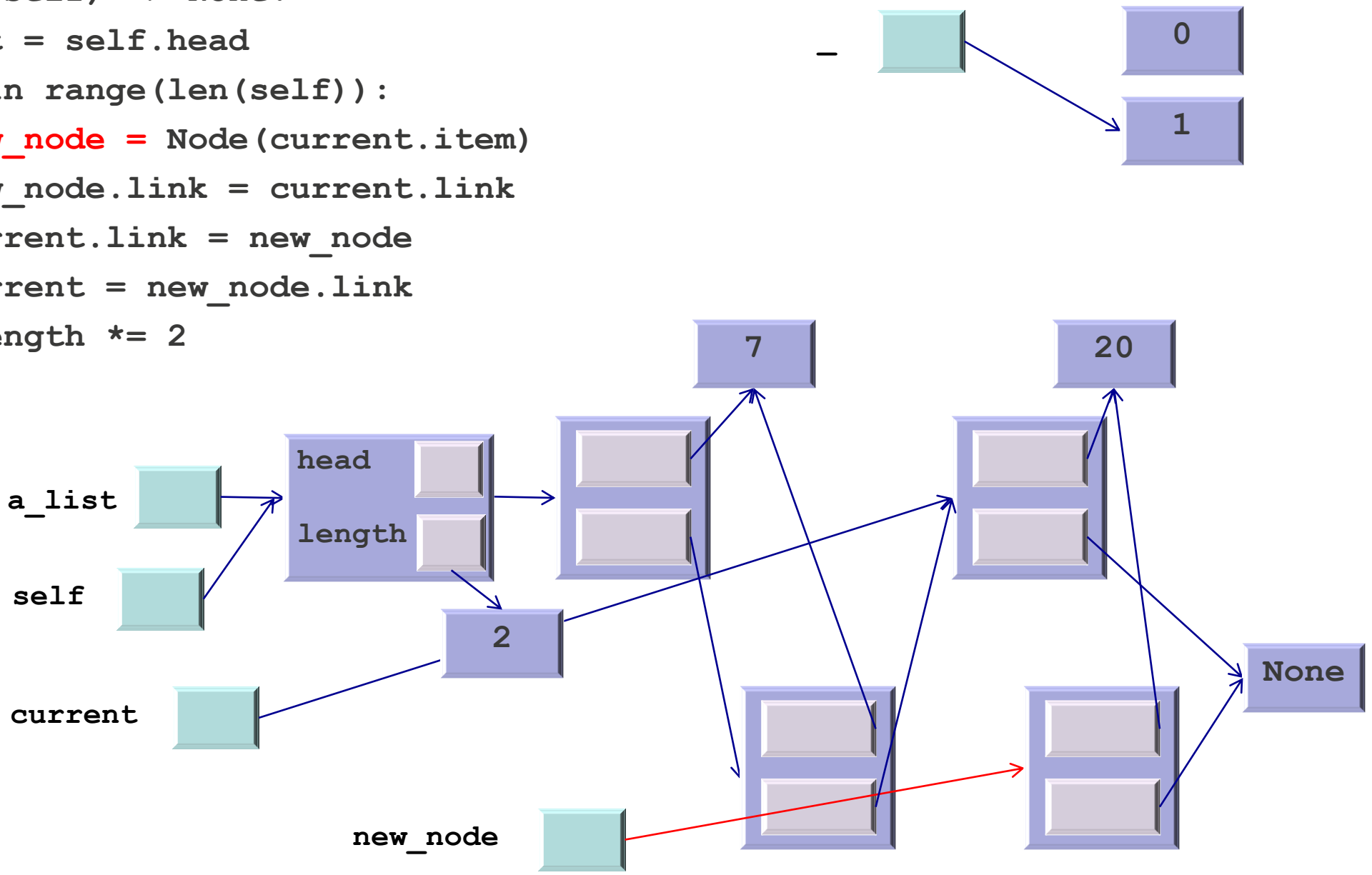


```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```

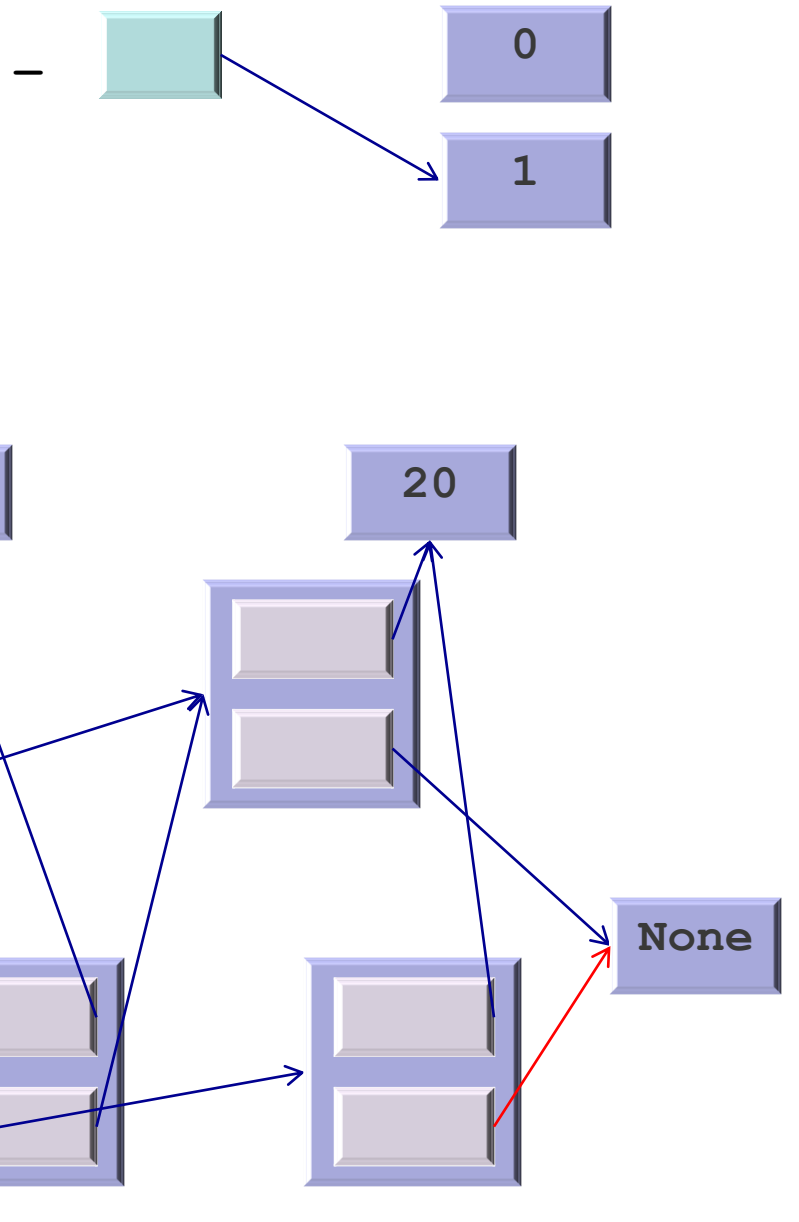




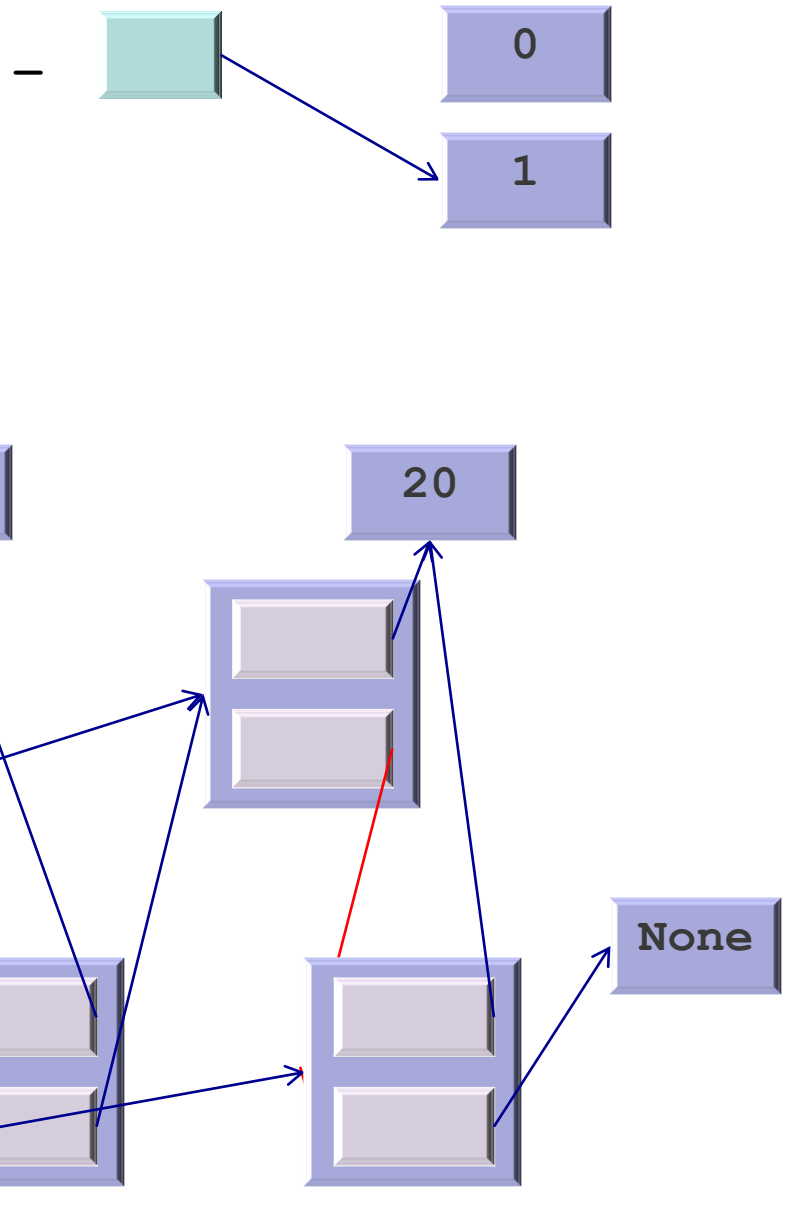
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



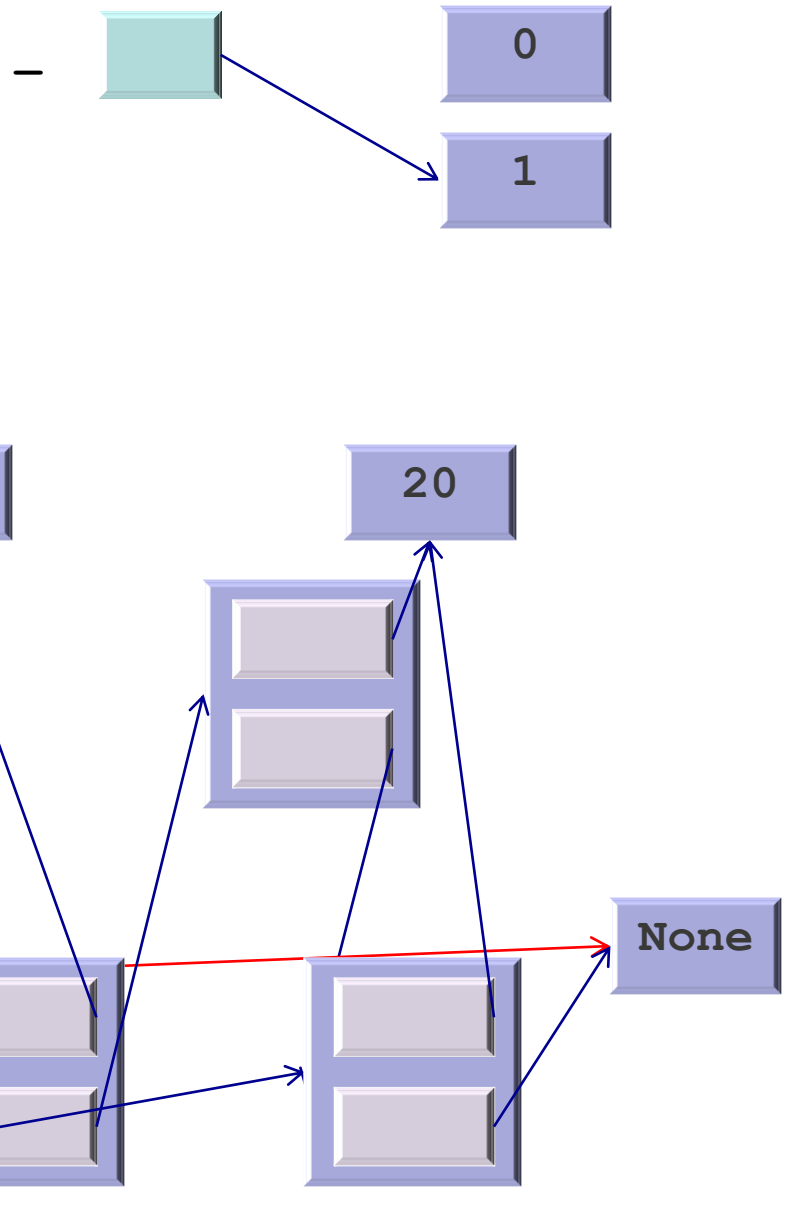
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



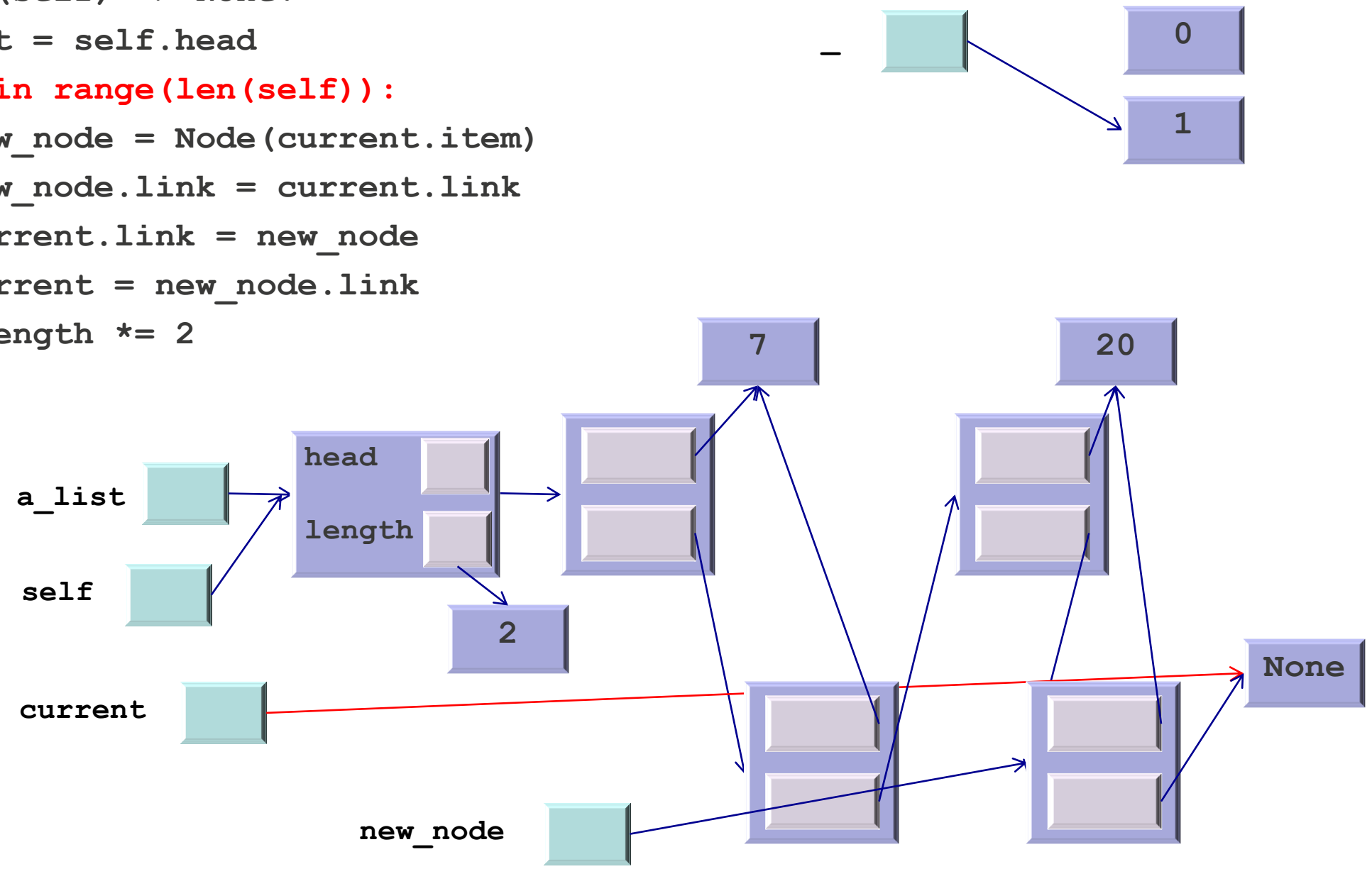
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



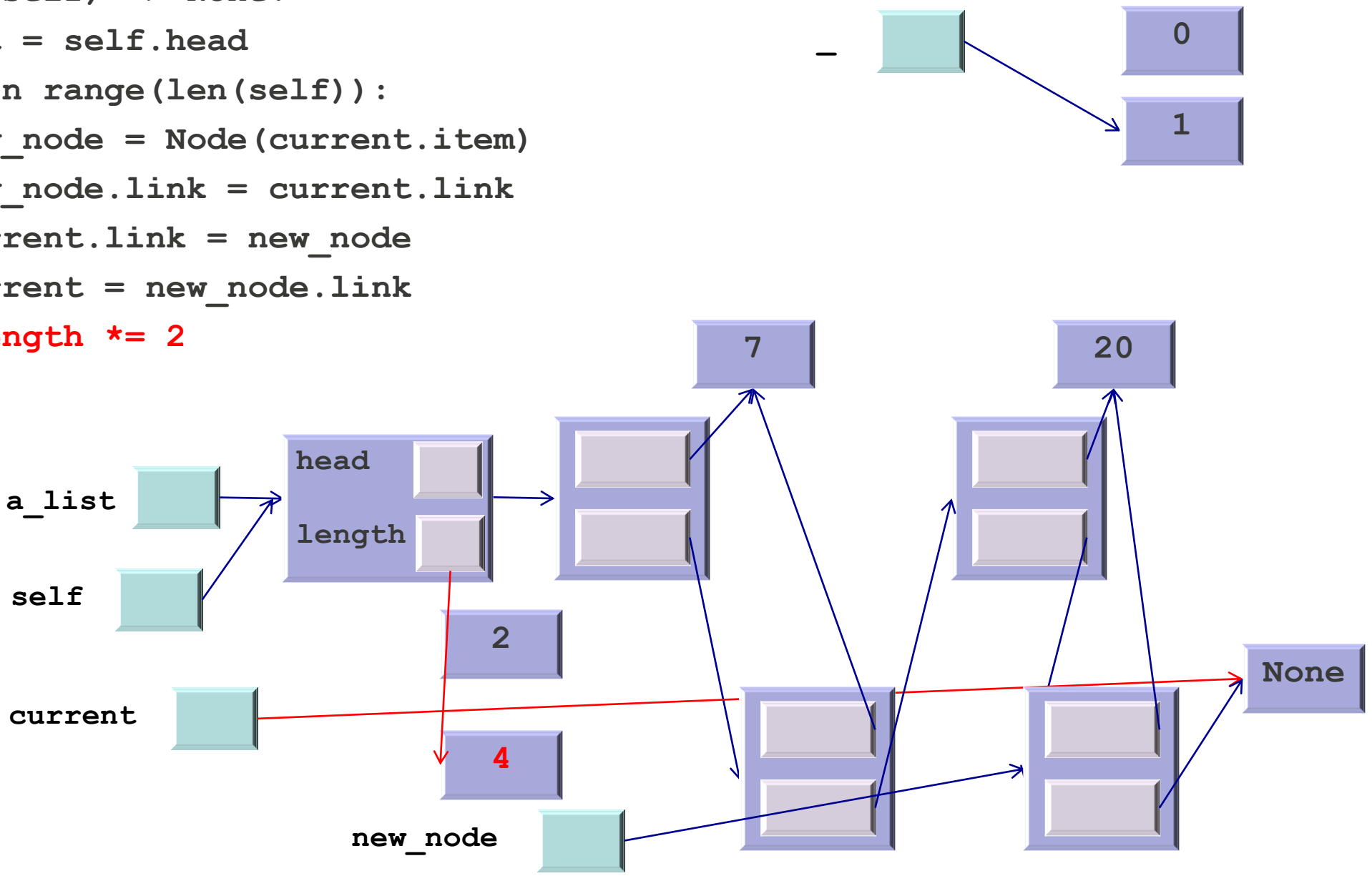
```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```



```
def double(self) -> None:
    current = self.head
    for _ in range(len(self)):
        new_node = Node(current.item)
        new_node.link = current.link
        current.link = new_node
        current = new_node.link
    self.length *= 2
```

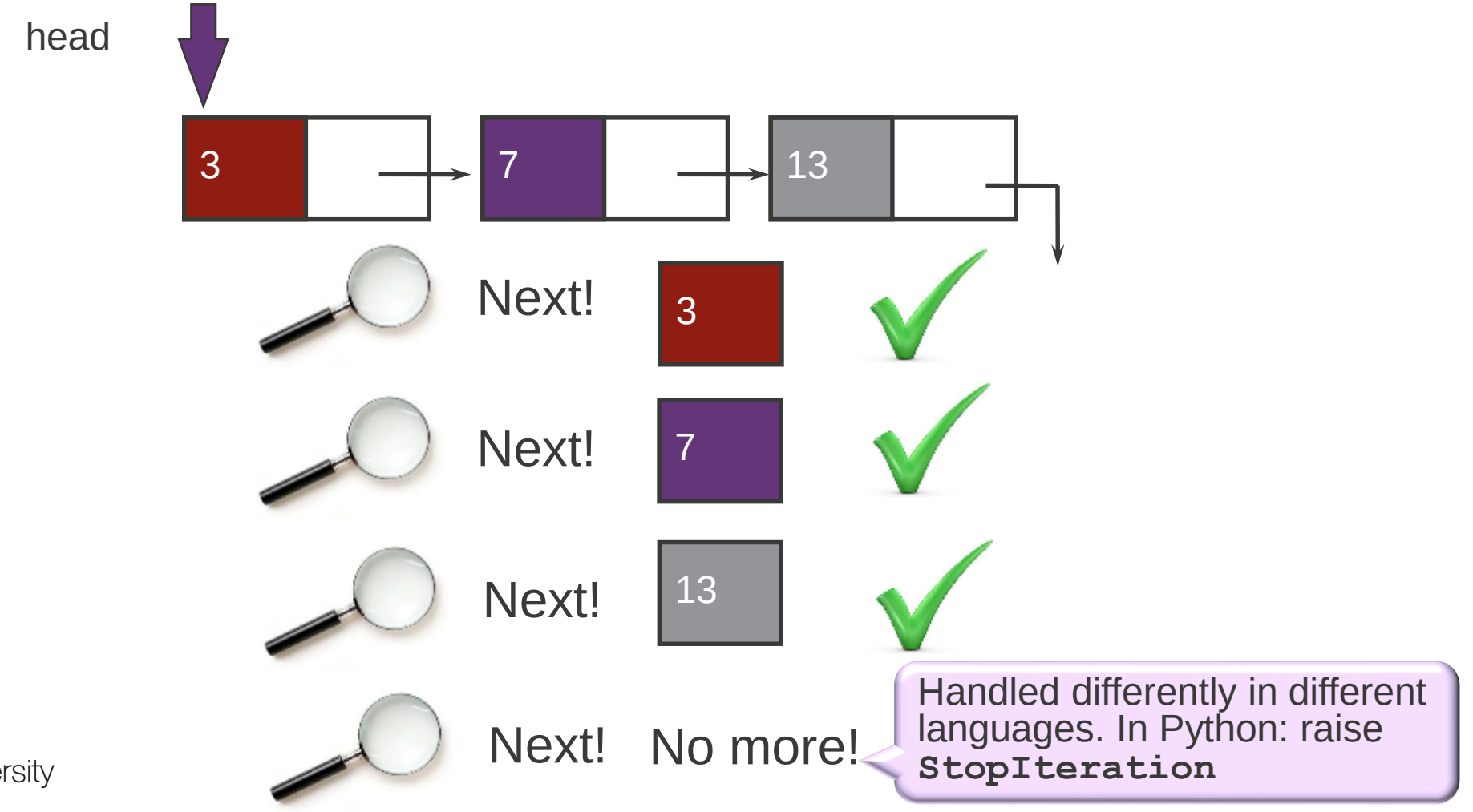


Motivation for iterators

- **How can we define `double()` from outside the class?**
  - Need to traverse the list
  - Need to add nodes into a list
- **How would we even traverse the list from outside the class?**
  - We would have to access **`self.link`**
  - This means accessing the implementation (not very abstract!)

Traversing a list as a user

- **What do users do when they need to check all elements are >0?**

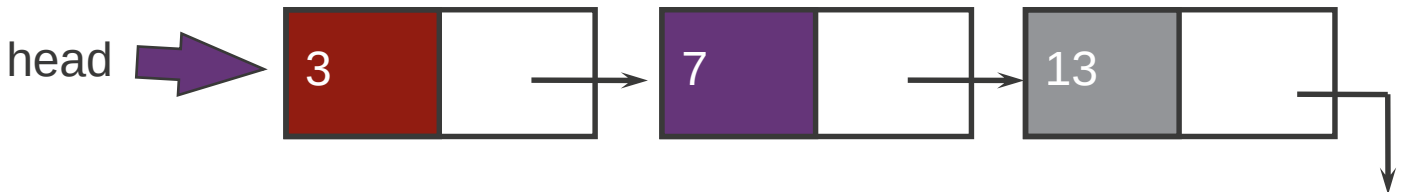


MONASH
University

Implementing Iterators

How do we do this?

- **We need a way of creating an iterator object to start the iterations**
  - An object other than the list because:
    - It needs to change (move through the list) without changing the list
    - We might need several iterators on a list
  - In Python, iterator objects are returned by method `__iter__`
- **This means we need an iterator class to create the objects:**
  - Its `__init__` performs the "beginning-of-iteration" initialisation
    - In our example: it makes sure we start from 3
- **The class needs a way of returning the next element**
  - In our example, first 3, then 7, then 13, and then `StopIteration`
  - In Python, this is implemented by the `__next__` method



How does this work from the outside?

- **Python lists and strings (among others) are iterable:**

Overloaded function. Python finds the `__iter__` from the `String` class

```
>>> x = "abc"
>>> it1 = iter(x)
>>> it1
<iterator object at 0x10a9f2>
>>> next(it1)
'a'
>>> next(it1)
'b'
>>> it2 = iter(x)
>>> next(it2)
'a'
>>> next(it1)
'c'
```

`it1` is an iterator, not a list

`it1` and `it2` are different objects and, thus, independent

```
>>> next(it1)
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
StopIteration
>>> next(it2)
'b'
>>> next(it2)
'c'
>>> next(it2)
Traceback ….StopIteration
>>> x
'abc'
```

`x` has not changed at all!

How do we make the LinkList class iterable?

- **We need to create an iterator class for the list**
  - A class with the **`__init__`**, **`__iter__`** and **`__next__`** methods

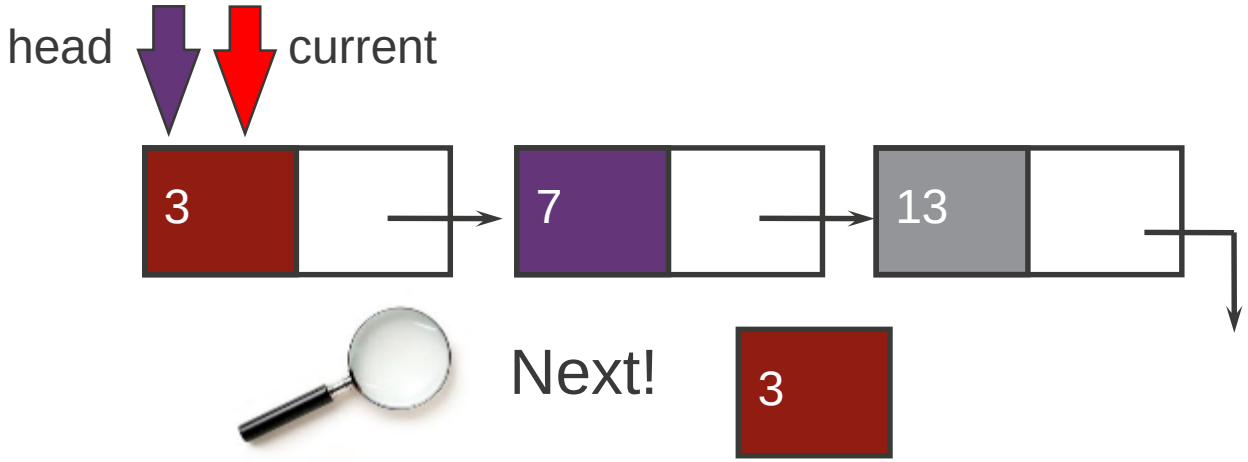
```
class LinkListIterator(Generic[T]):
    def __init__(self, node: Node[T]) -> None:
        self.current = node

    def __iter__(self):
        return self

    def __next__(self) -> T:
```

Takes a list node as argument, and initialises the iterator to start at that node

Must return an object with a **`__next__`** method. This one simply returns itself

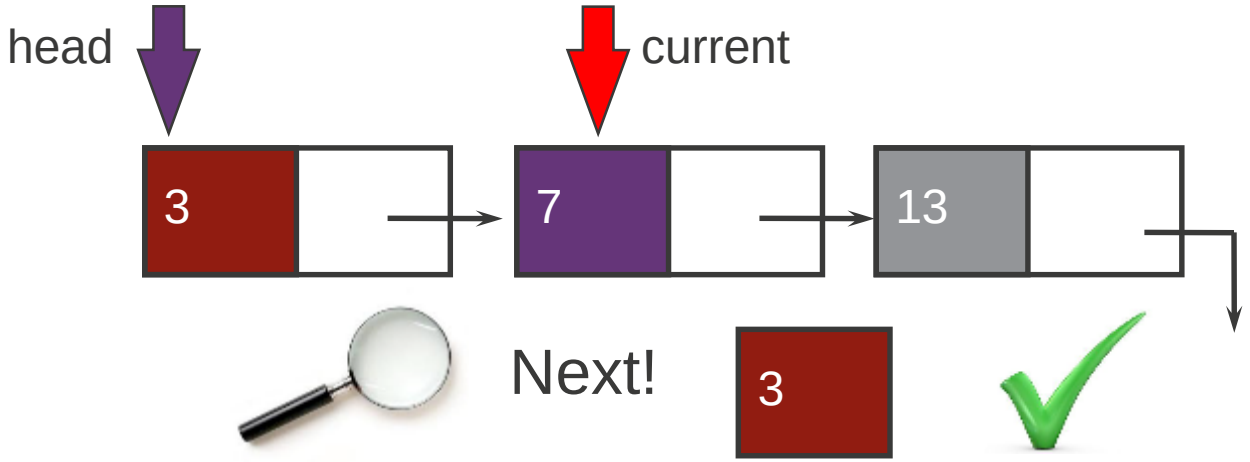
How do we implement `__next__`?

- **We need a reference `current` that starts at the `head`**



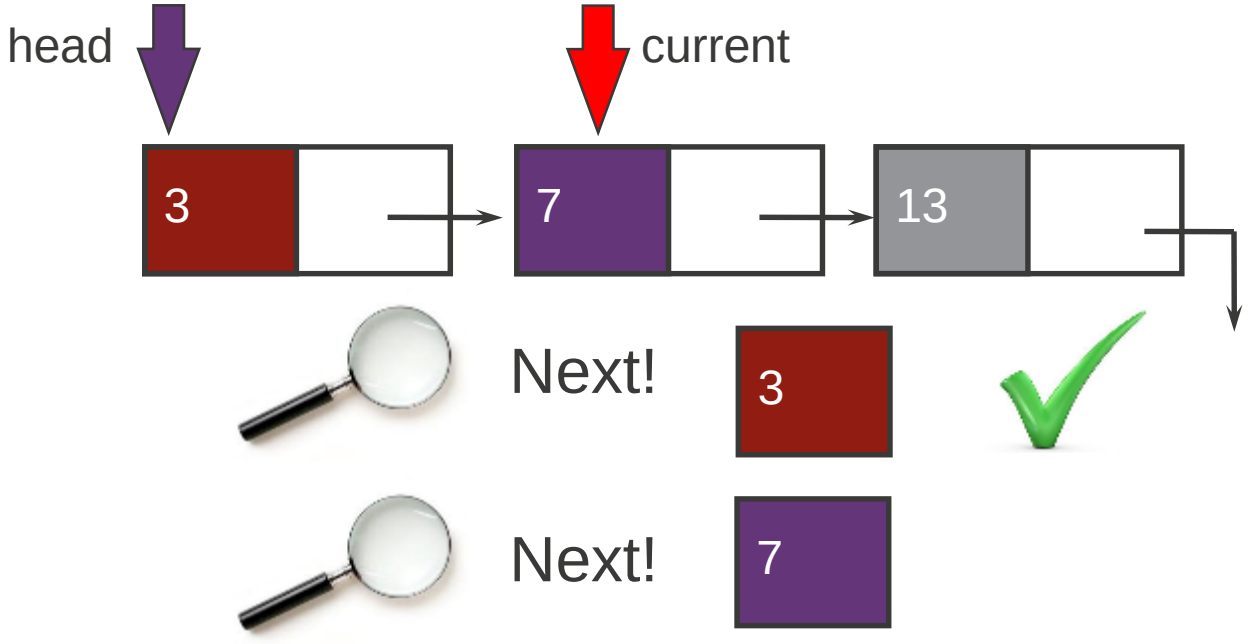
How do we implement `__next__`?

- **We need a reference `current` that starts at the `head`**



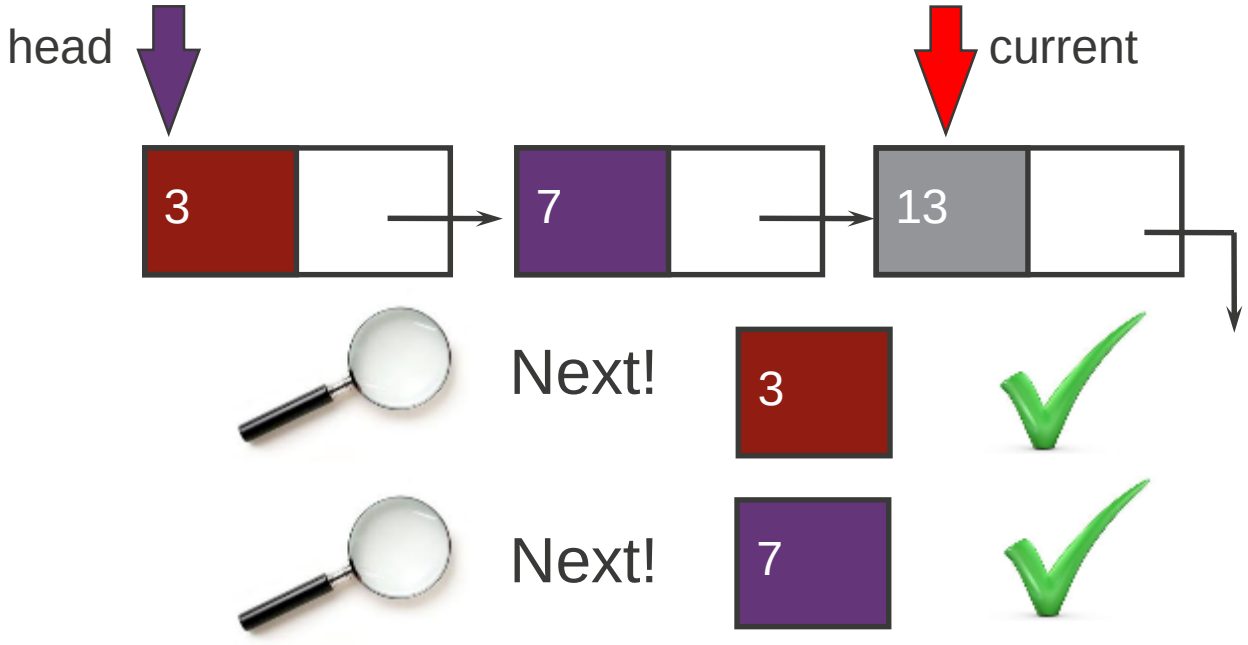
How do we implement `__next__`?

- **We need a reference `current` that starts at the `head`**



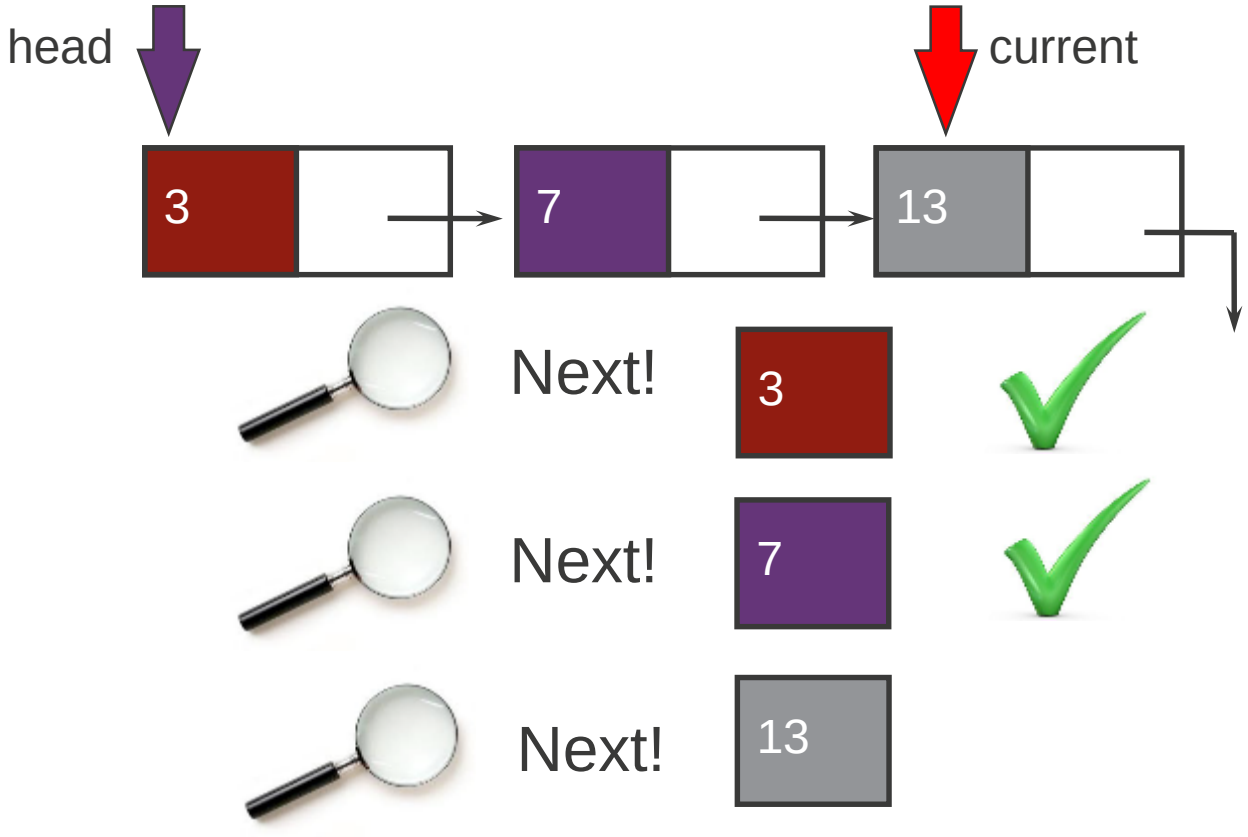
How do we implement `__next__`?

- **We need a reference `current` that starts at the `head`**



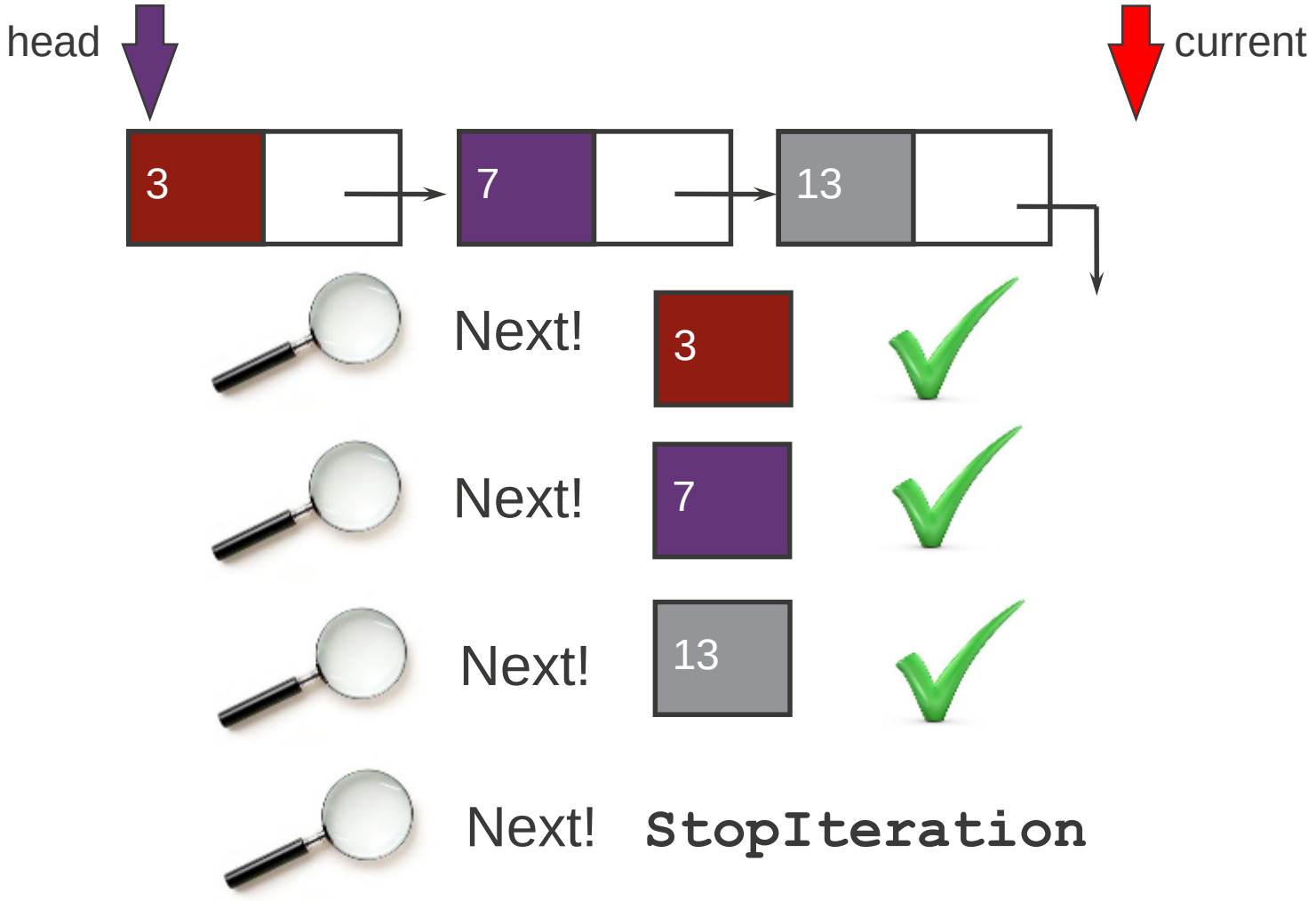
How do we implement `__next__`?

- **We need a reference `current` that starts at the `head`**



How do we implement `__next__`?

- **We need a reference `current` that starts at the `head`**



How do we make the List class iterable?

- **We need to create an iterator class for the list**
  - A class with the `__init__`, `__iter__` and `__next__` methods

```
class LinkListIterator(Generic[T]):
    def __init__(self, node: Node[T]) -> None:
        self.current = node

    def __iter__(self):
        return self

    def __next__(self) -> T:
        if self.current is not None:
            item = self.current.item
            self.current = self.current.link
            return item
        else:
            raise StopIteration
```

Takes a list node as argument, and initialises the iterator to start at that node

Must return an object with a `__next__` method. This one simply returns itself

If not finished, remember current item and advance

Why not `self.current != None`?

Because users might have re-defined `__eq__(self, rhs)`

If finished, raise `StopIteration`

An equivalent way of defining __next__

- **We have defined `__next__` as:**

```
def __next__(self) -> T:
    if self.current is not None:
        item = self.current.item
        self.current = self.current.link
        return item
    else:
        raise StopIteration
```

- **But we could also have said:**

```
def __next__(self) -> T:
    try:
        item = self.current.item
    except AttributeError:
        raise StopIteration
    else:
        self.current = self.current.link
        return item
```

Which one is better? Whichever one is clearer (different people…)

How is this connected to the LinkList class?

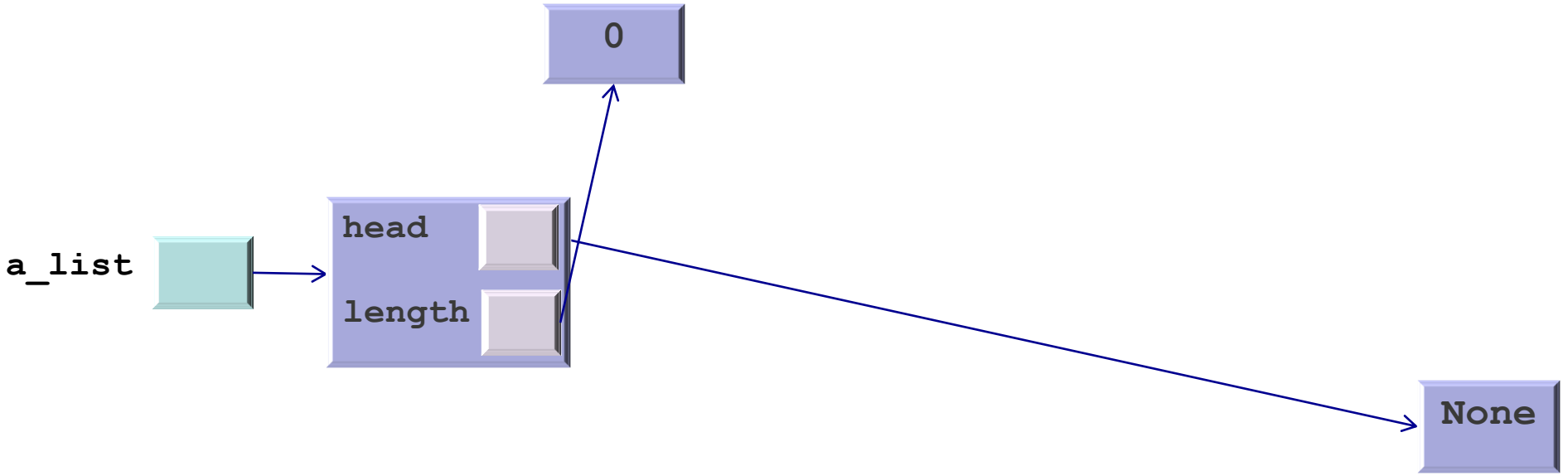
- **The `LinkList` class needs to have an `__iter__` method too**
- **Which does what?**
  - Return a list iterator object initialised to the head of the list

```
class LinkList(List[T]):

    ...

    def __iter__(self) -> LinkListIterator[T]:
        return ListIterator(self.head)
```

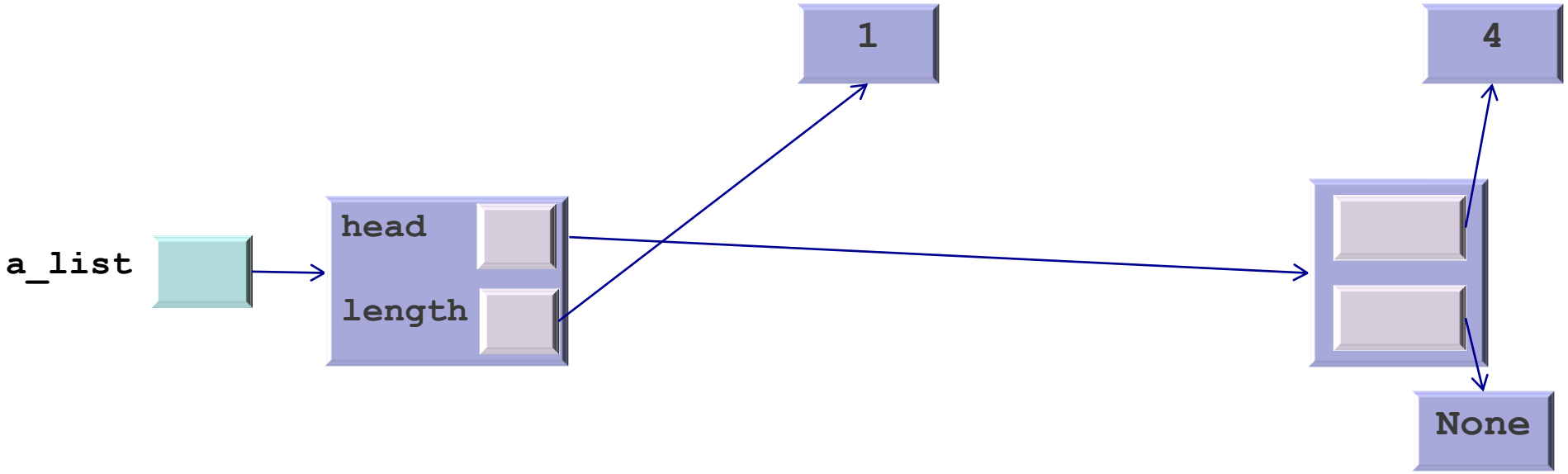
This points to the **`List`** object

Somewhere inside the **`List`** class

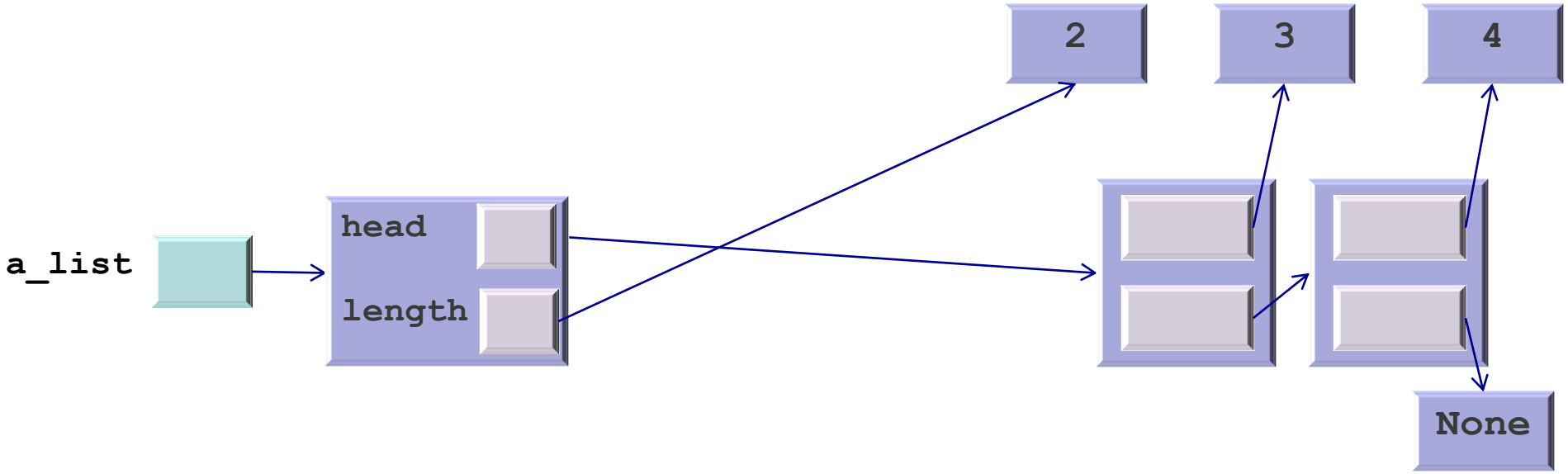
This points to a **`Node`** object (the one at the head of the list), which remember, is the input argument for the **`__init__`** in the **`ListIterator`** class



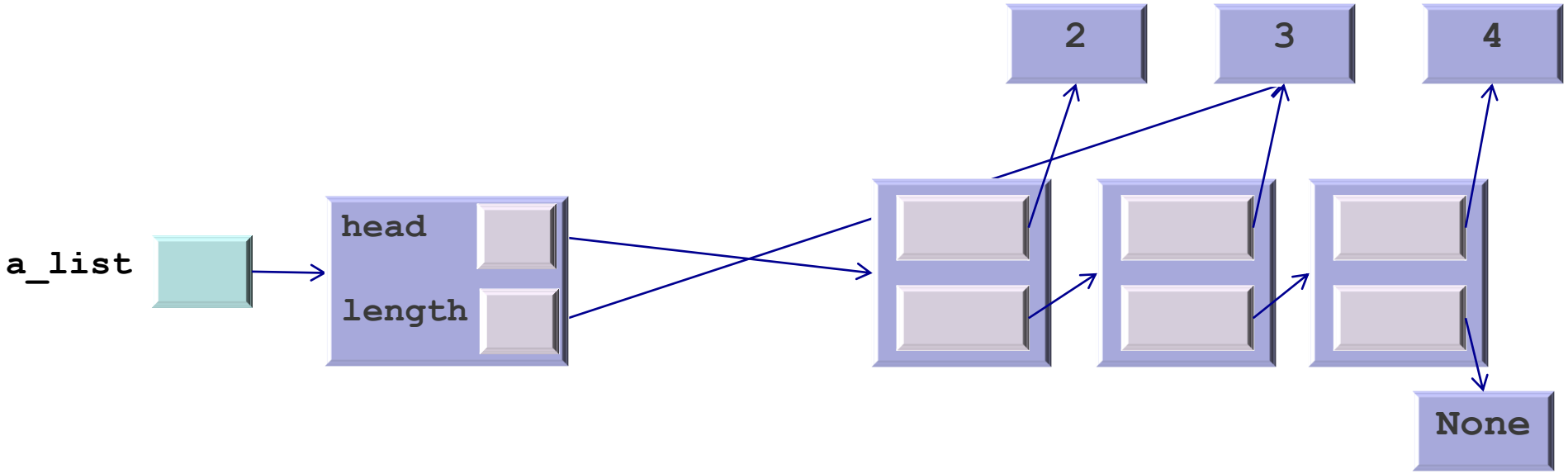

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
```



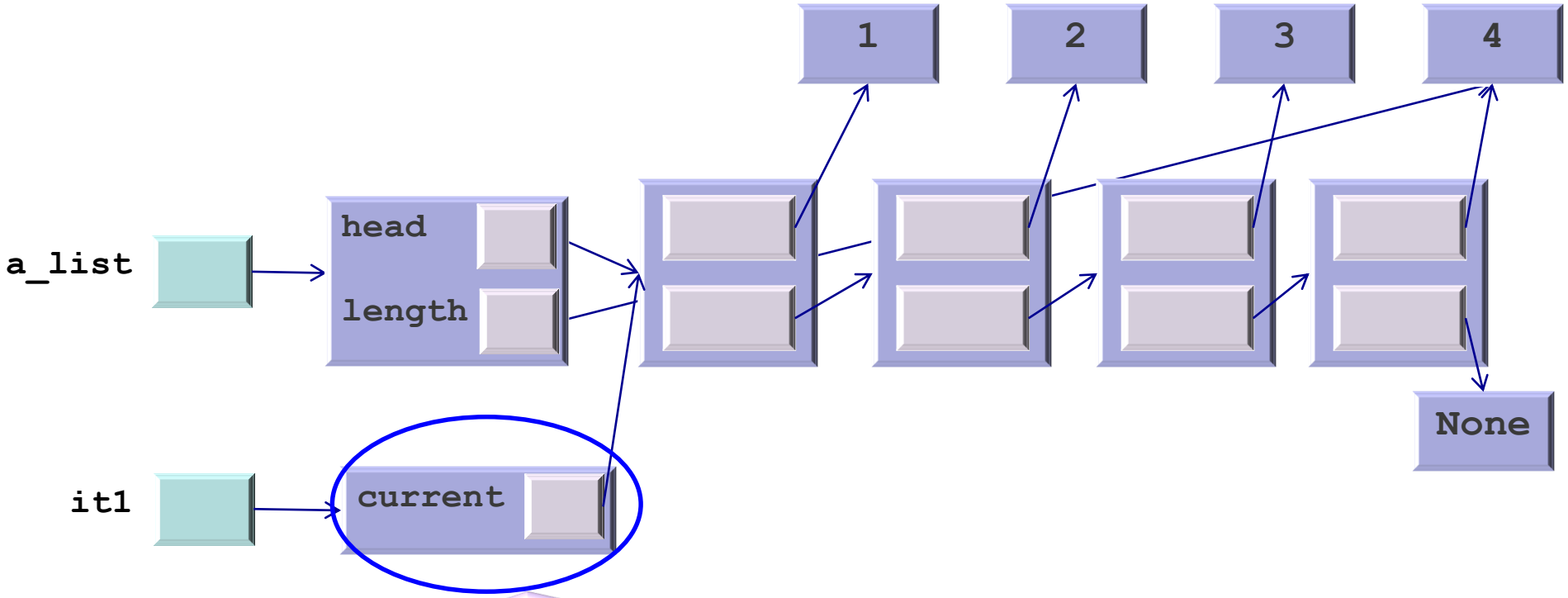

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
```




```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
```



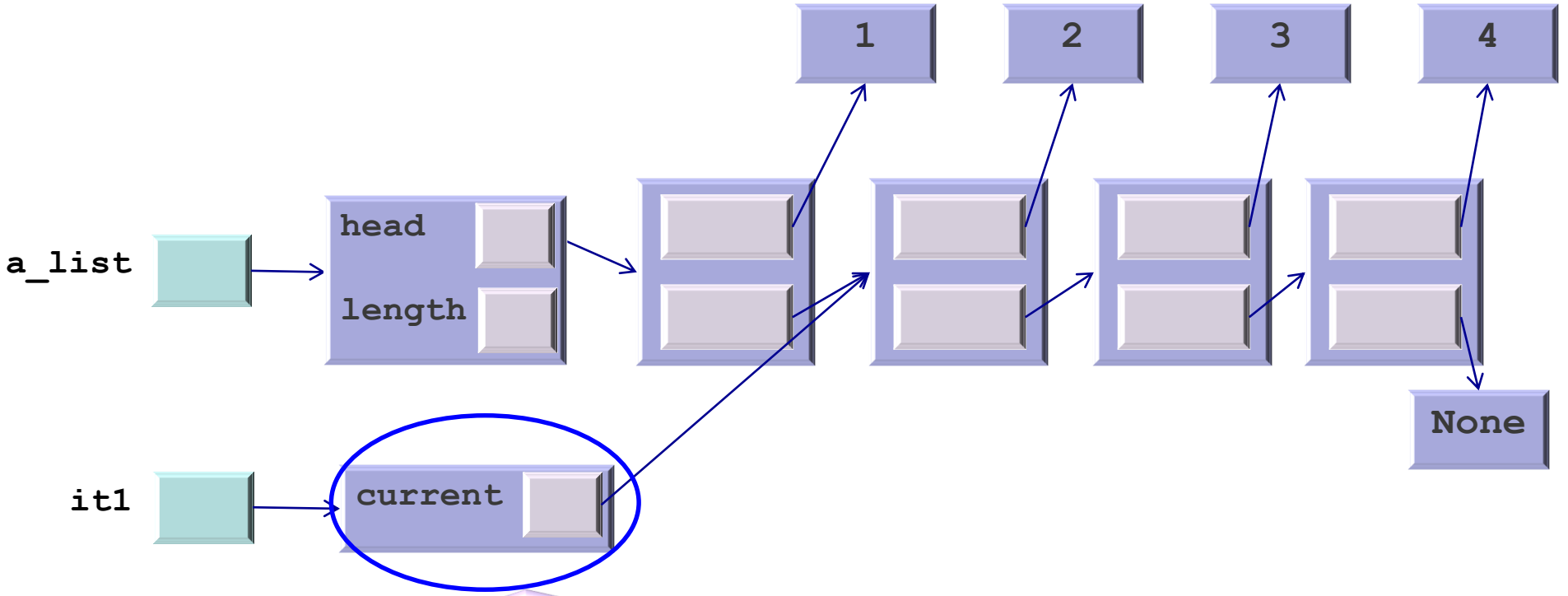

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
```



This is my **LinkListIterator** object

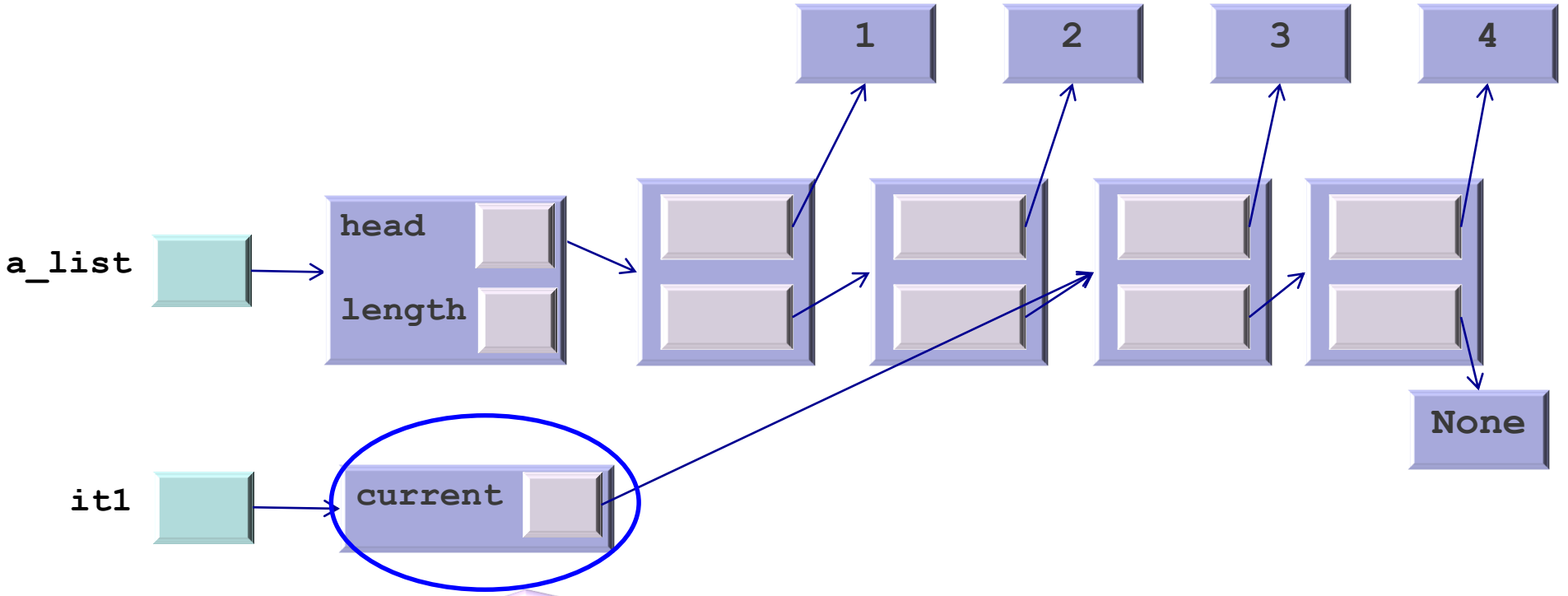
```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> it1 = iter(a_list)
>>> next(it1)
```

```
def __next__(self) -> T:
    if self.current is not None:
        item = self.current.item
        self.current = self.current.link
        return item
    else:
        raise StopIteration
```

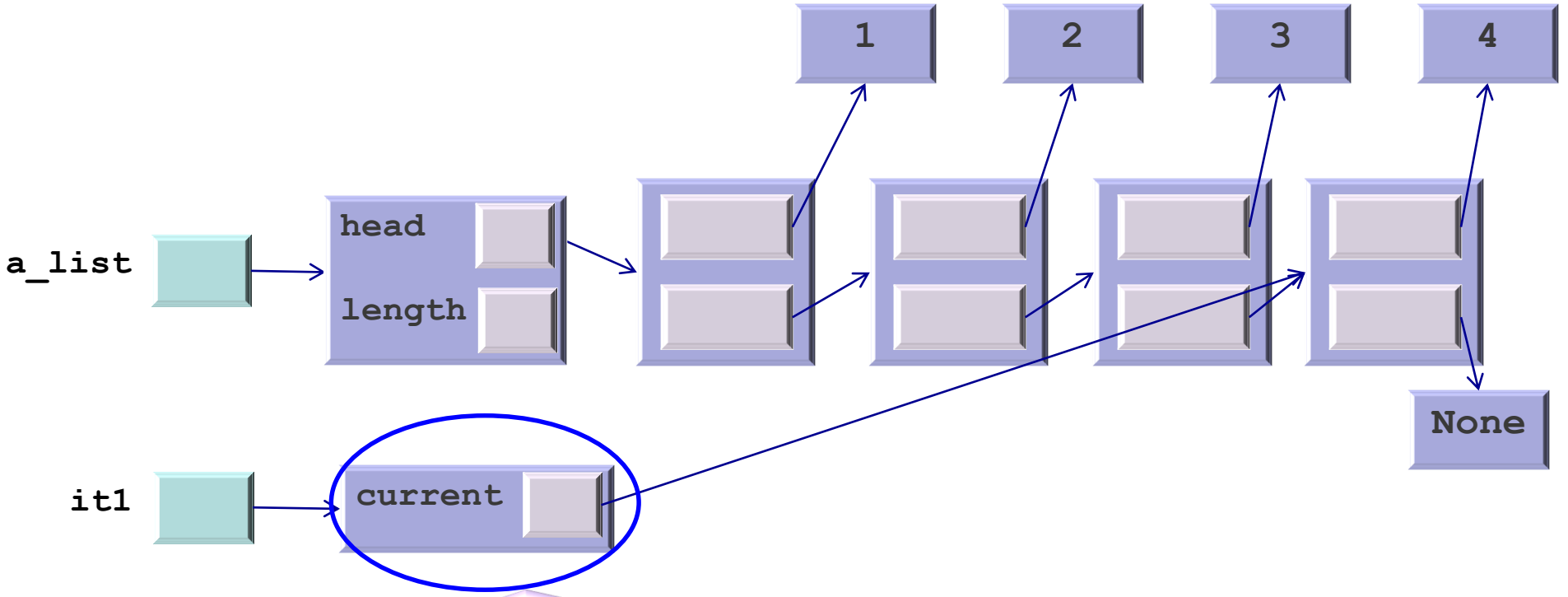


This is my **LinkListIterator** object

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> it1 = iter(a_list)
>>> next(it1)
1
>>> next(it1)
```

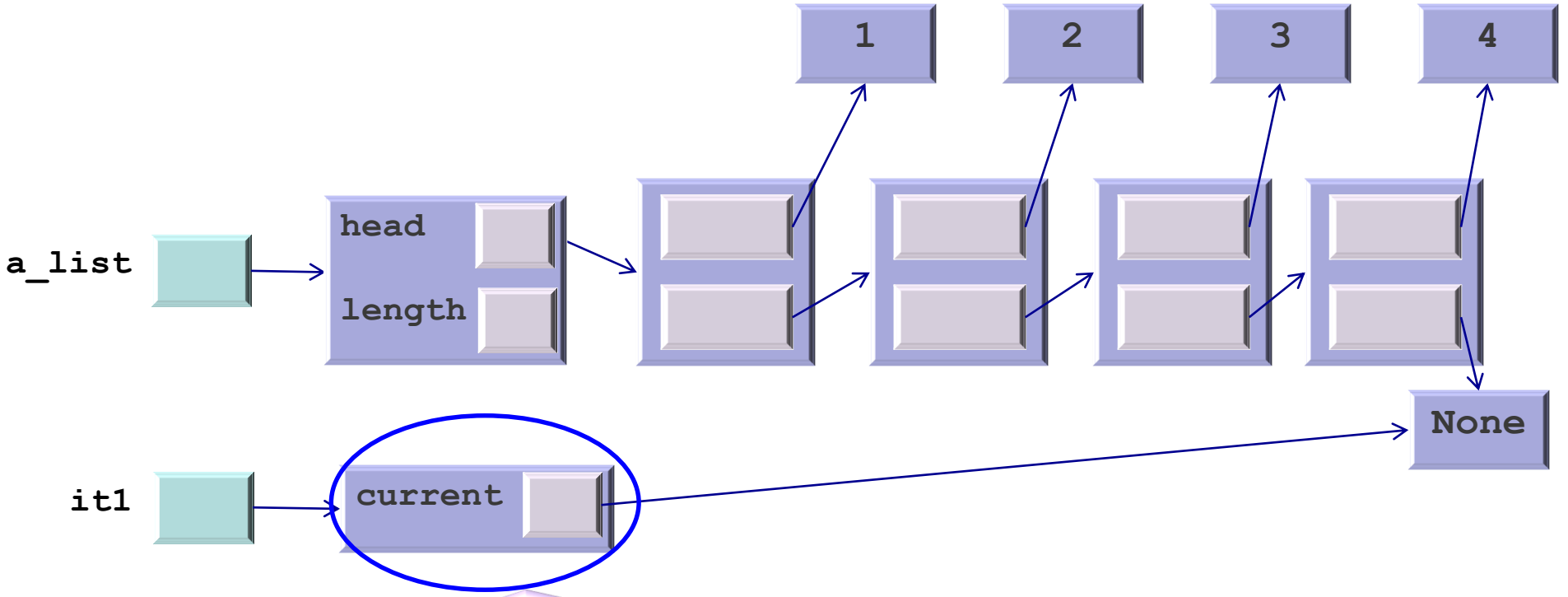



```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> it1 = iter(a_list)
>>> next(it1)
1
>>> next(it1)
2
>>> next(it1)
```



This is my **LinkListIterator** object

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> it1 = iter(a_list)
>>> next(it1)
1
>>> next(it1)
2
>>> next(it1)
3
>>> next(it1)
```

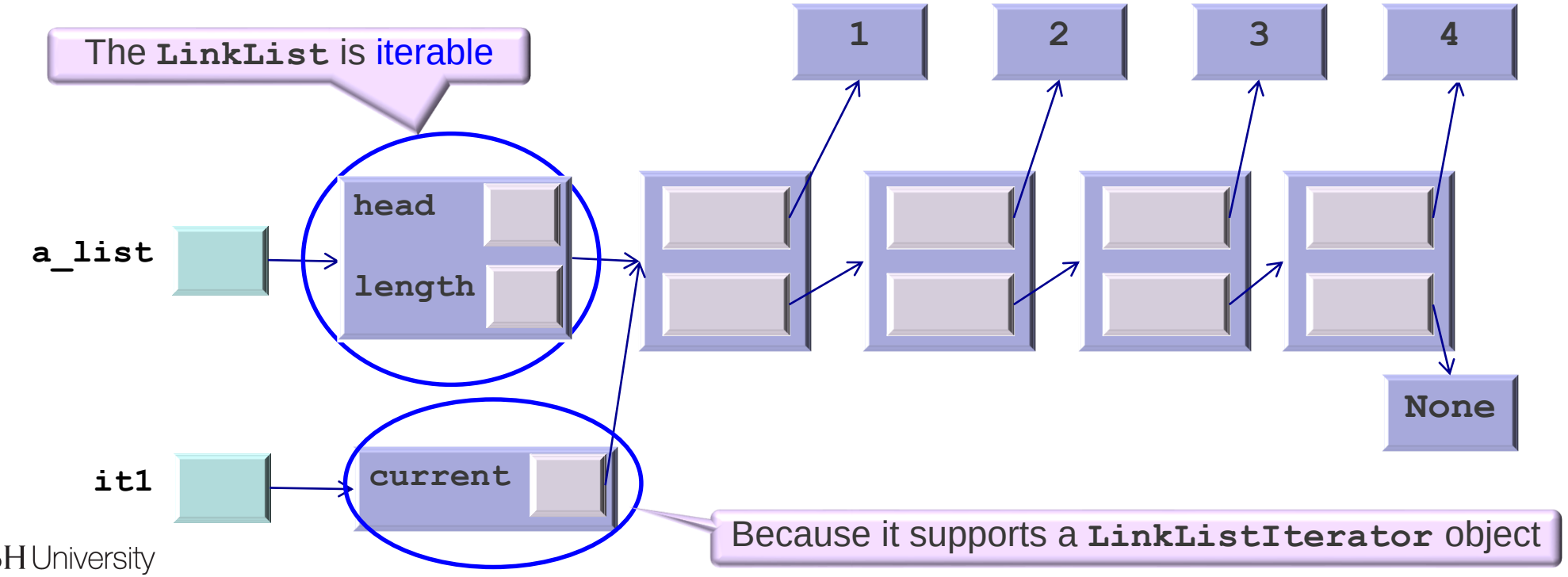


This is my **LinkListIterator** object

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> it1 = iter(a_list)
>>> next(it1)
1
>>> next(it1)
2
>>> next(it1)
3
>>> next(it1)
4
>>> next(it1)
Traceback …:
  File ... in __next__
    raise StopIteration
```

Iterables versus iterators

- **We have made our `List` class iterable, since**
  - It defines an `__iter__` method that returns an iterator on the list
- **The objects of the `LinkListIterator` class are iterators, since**
  - They support the `__iter__` and `__next__`



Now that our `LinkList` class is iterable…

- **We can use the `for` notation on it:**

```
>>> a_list = LinkList()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> for item in a_list
...     print(item)
...
1
2
3
4
>>>
```

So easy!

MONASH University

Using Iterators

So let's use it

- Define `positive(a_list)` which returns `True` if all items are >0
- Assume you are a user: outside the class, no access to internals
- One possible solution is:

```
def positive(a_list: LinkList[T]) -> bool:
    for item in a_list:
        if item <= 0:
            return False
    return True
```

Too easy! Not even have to explicitly call `iter` or `next`

Complexity?

Best: O(1) when the first element is <=0

Worst : O(N) when all elements are >0

List comprehensions and Iterable classes

- **List comprehension: create a list from another list using mathematic-like notation:**

```
>>> A = [3*x for x in range(10)]
>>> B = [x for x in A if x % 2 == 0]
>>> A;B
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
[0, 6, 12, 18, 24]
```

Both `range()` and list `A` are iterable

- **Common operations on an iterator's output include:**
  - Perform some operation for every element (e.g., `3*x`)
  - Select a subset of elements that meet a condition (e.g., `x % 2 == 0`)
- **List comprehensions allow you to do that AND return a list**
- **Generator expressions allow you to do that AND return an iterator**

```
>>> A = (3*x for x in range(10))
>>> next(A)
0
```

You can now do things like this

Equivalent definition

- **We said one possible solution for `positive(a_list)` was:**

```
def positive(a_list: LinkList[T]) -> bool:
    for item in a_list:
        if item <= 0:
            return False
    return True
```

- **Another (more pythonic) solution is:**

```
def positive(a_list: LinkList[T]) -> bool:
    return [] == [e for e in a_list if e <= 0]
```

I do not expect you to program like this though… not just yet

Let's use it again

- **Define `max(a_list)` to compute the maximum item `a_list`**
- **Assume you are a user: outside the class, no access to internals**
- **One could write the following:**

```
def max(a_list: LinkList[T]) -> int:
    for item in a_list:
        if max < item:
            max = item
    return max
```

But this does not work. Why?

Local variable **`max`** has not been initialised before the comparison. Python will say:

```
UnboundLocalError:local variable
'max' referenced before assignment
```

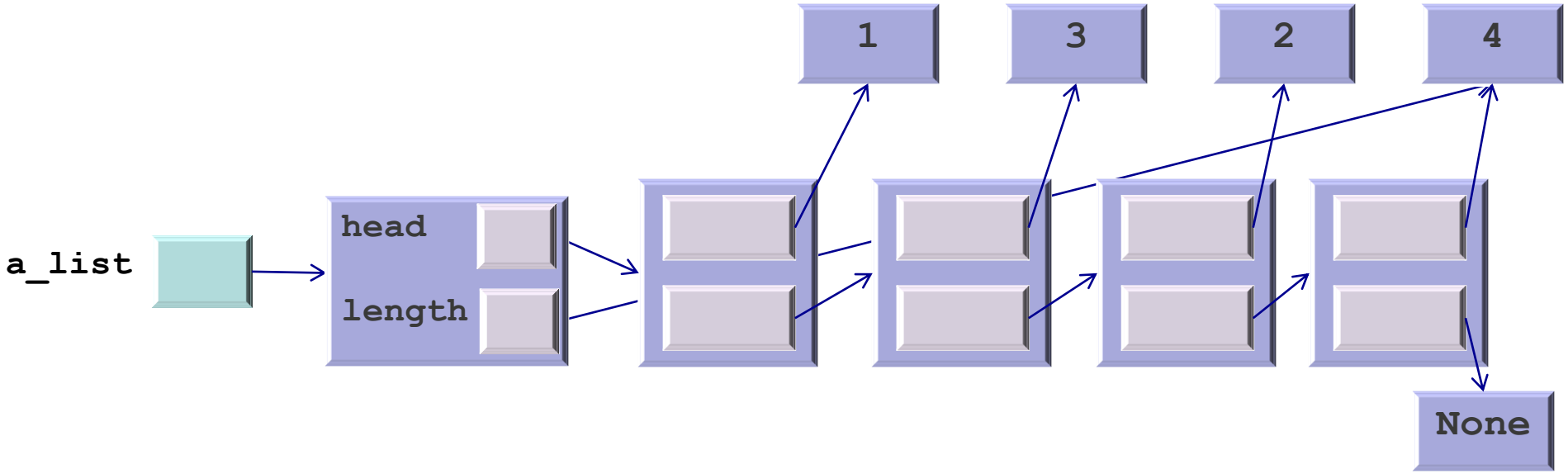
- **What do you need to do to fix it?**
  - Give a (reasonable) initial value to max (that of an element of the list)
- **What does the function do if the list is empty?**
  - Raise an exception

Let's use it again (cont)

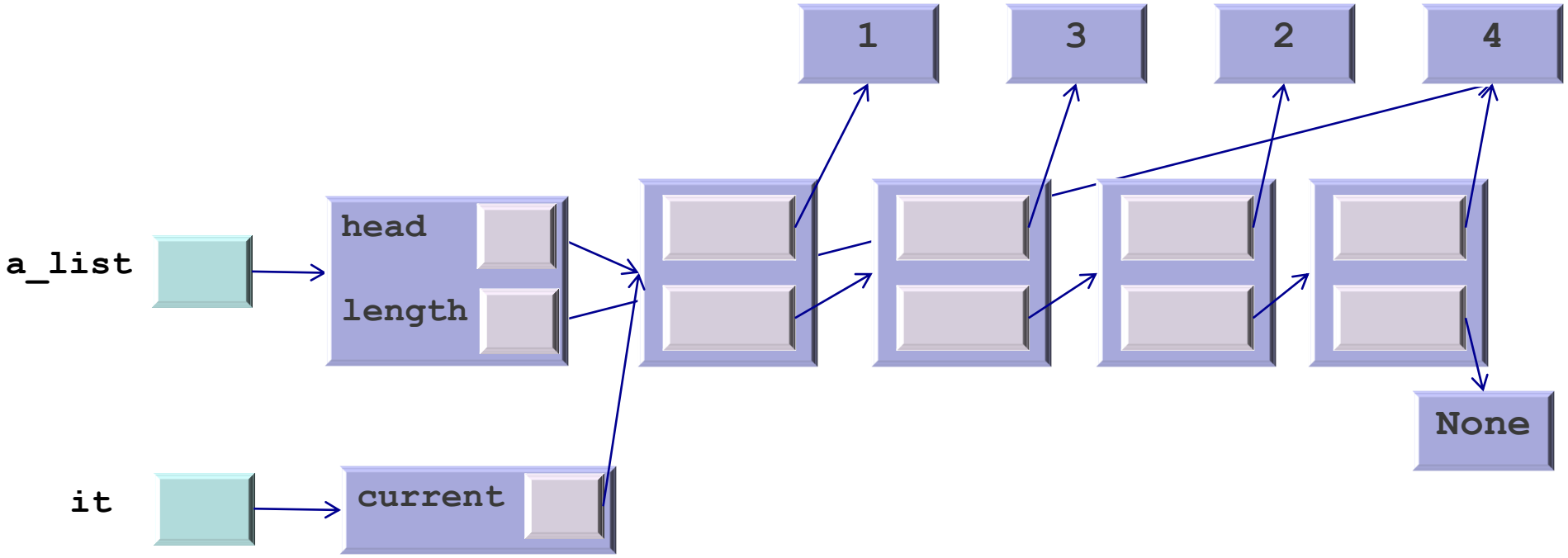
- **One possibility (there are many!):**

Iterators are also iterable!

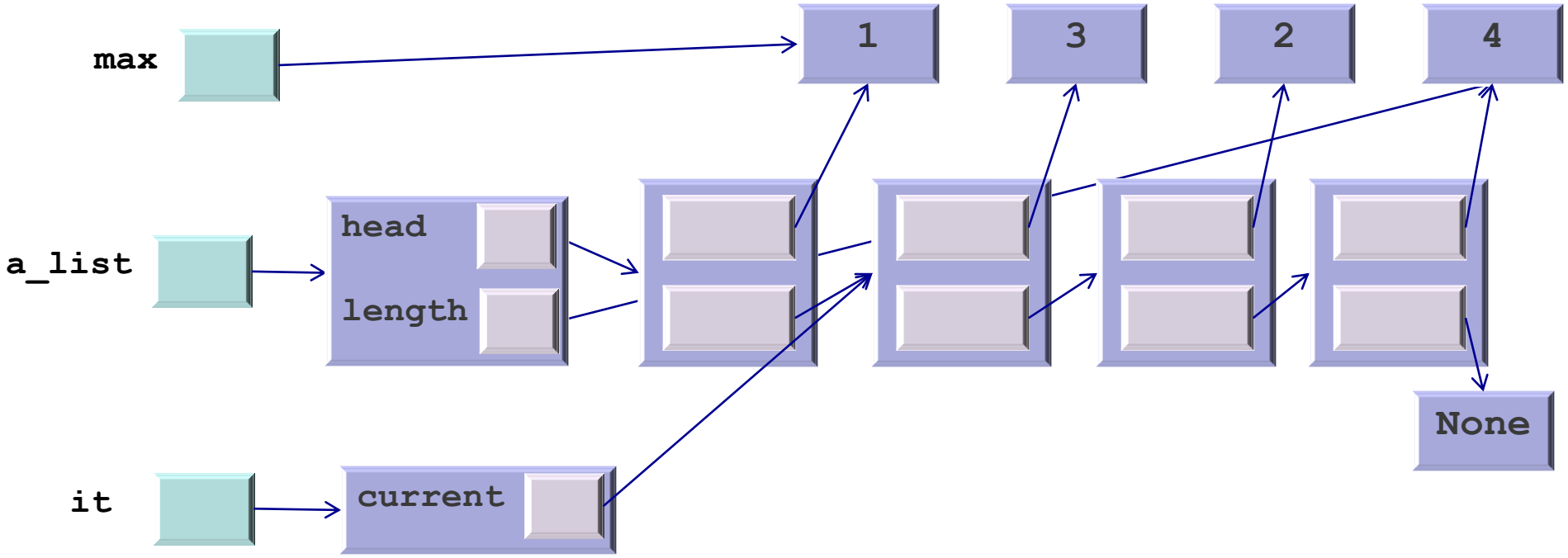
```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list) # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:  # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



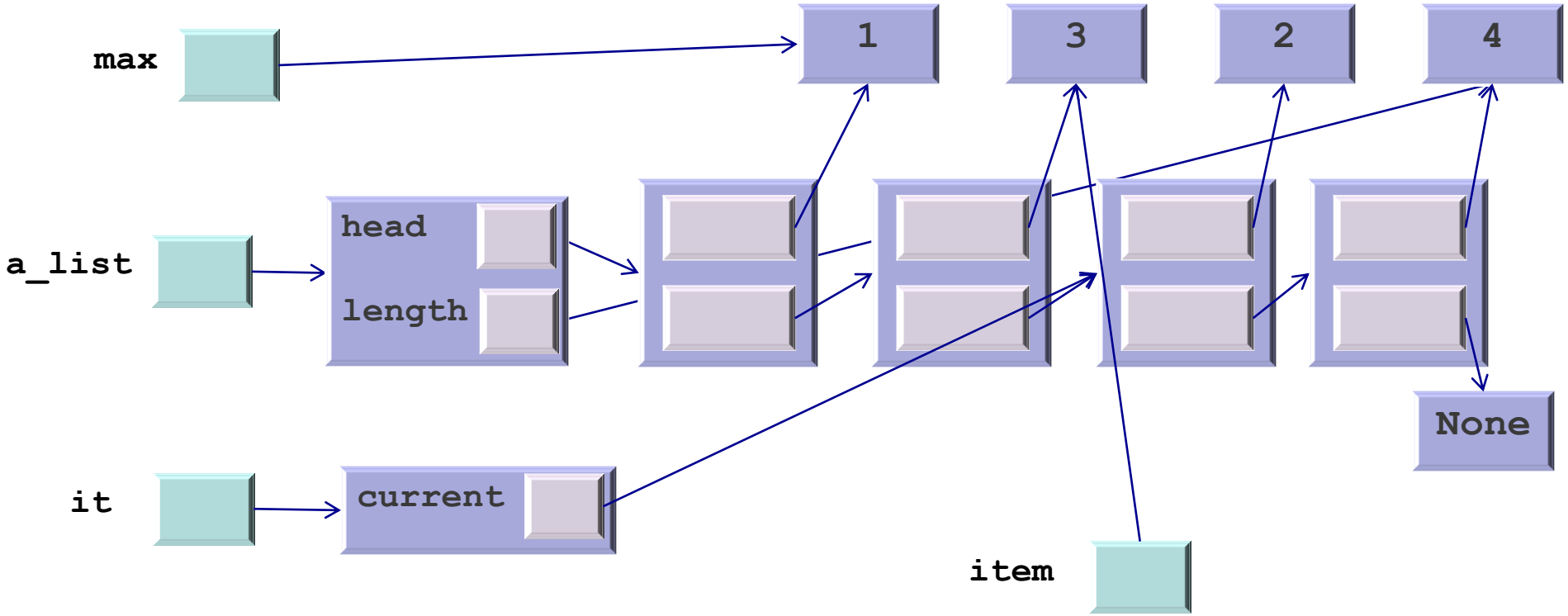

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



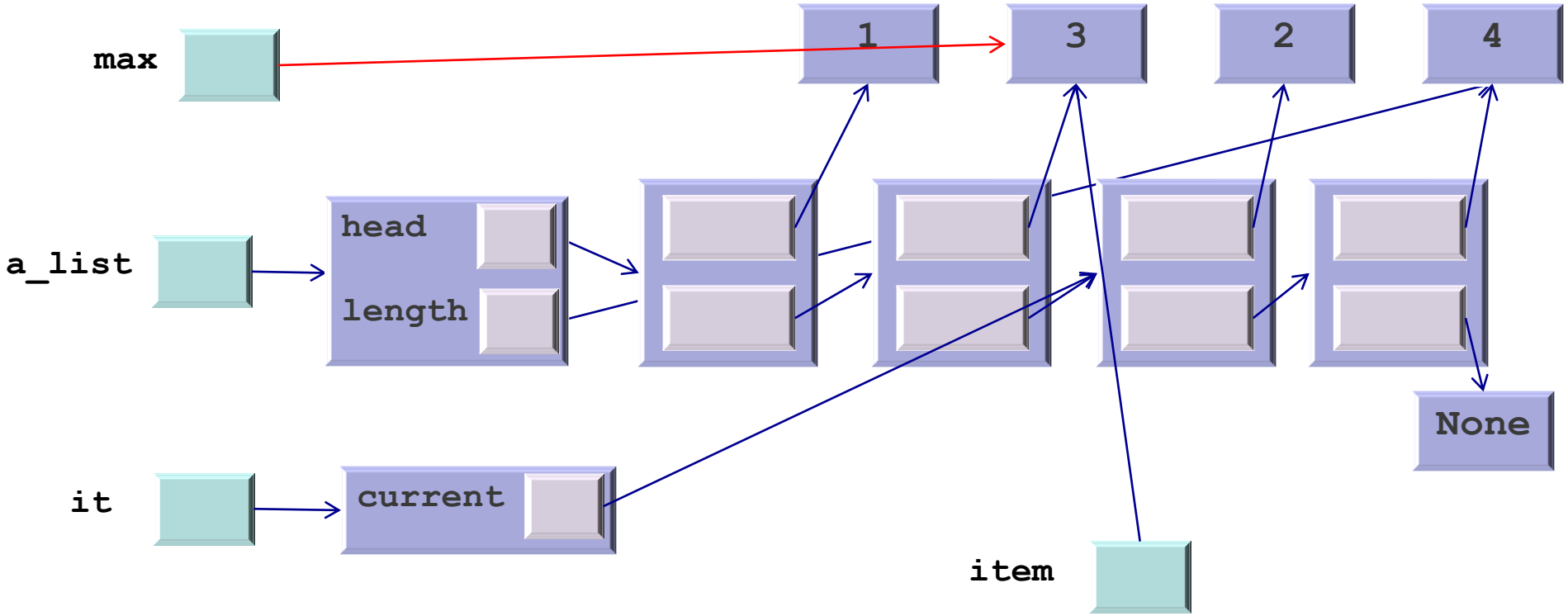

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



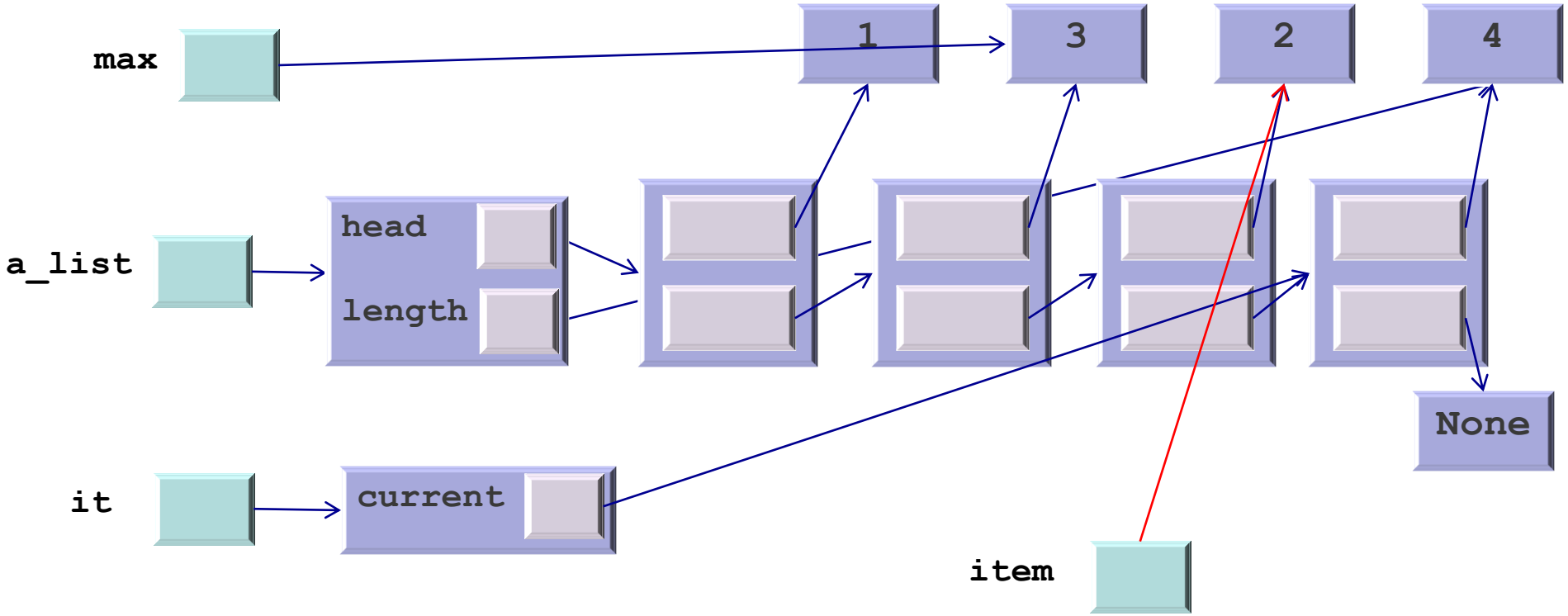

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



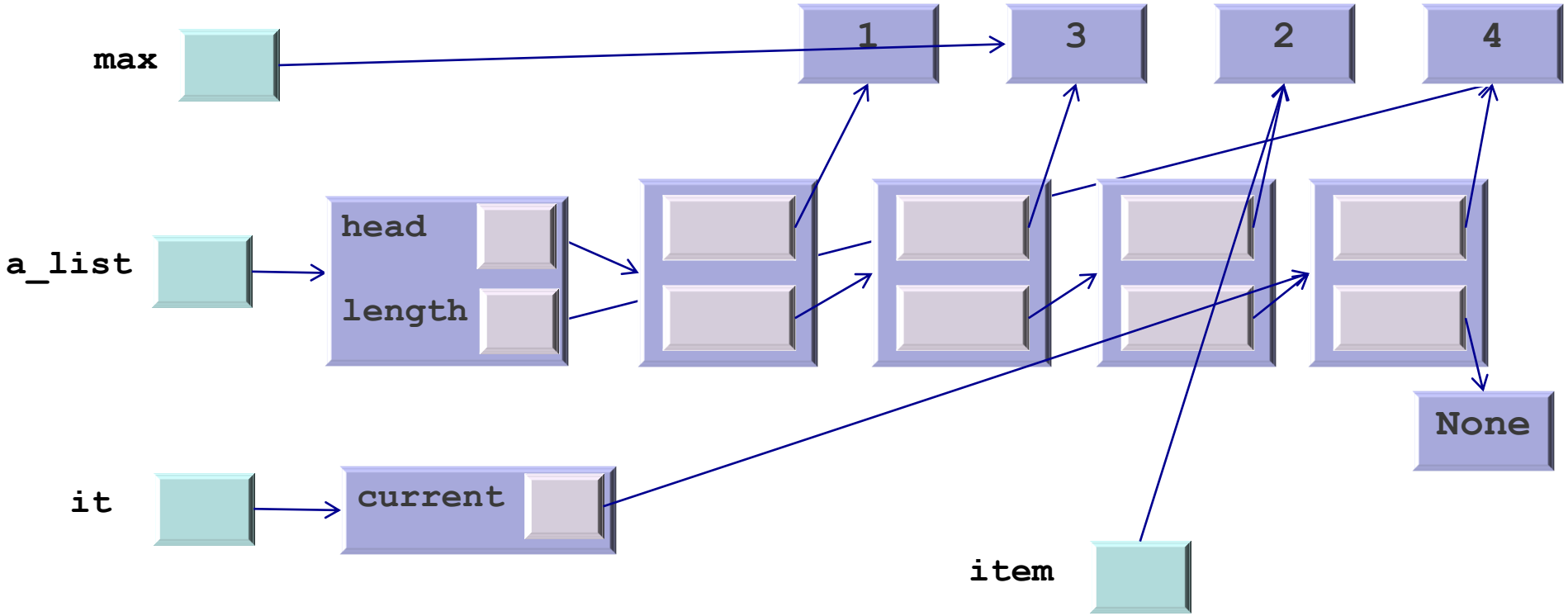

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



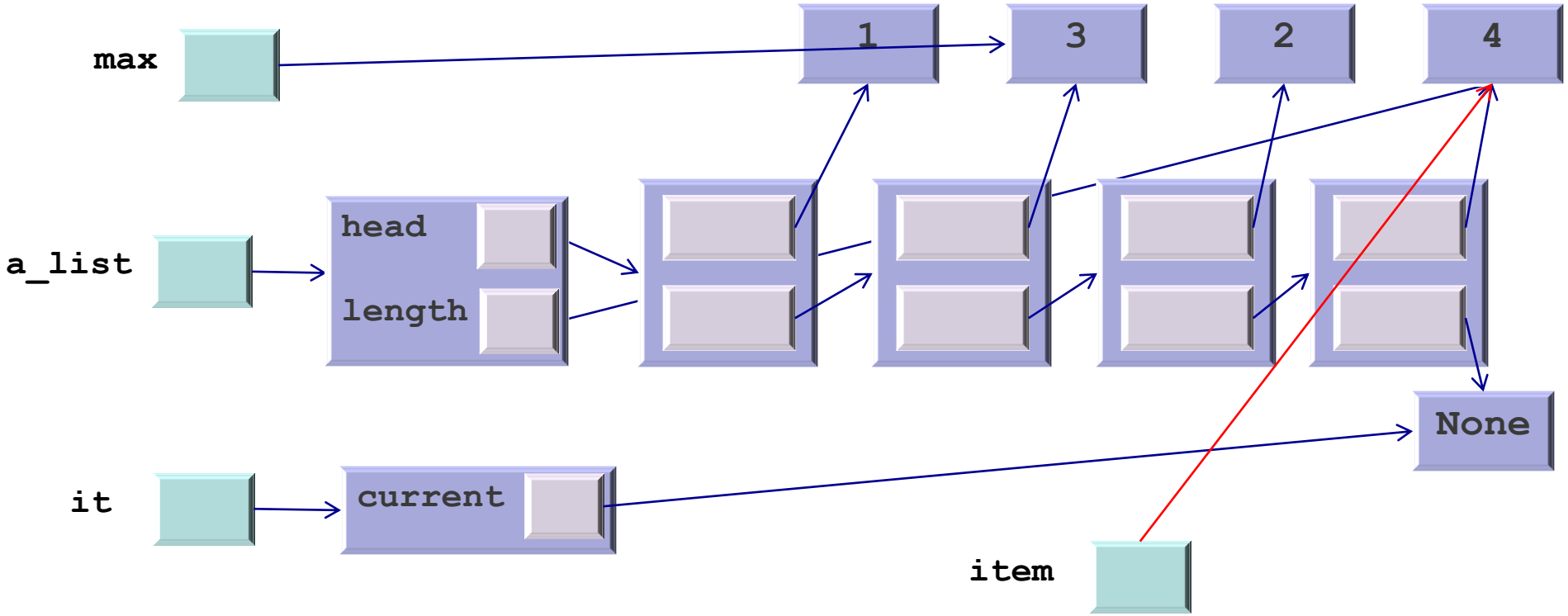

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



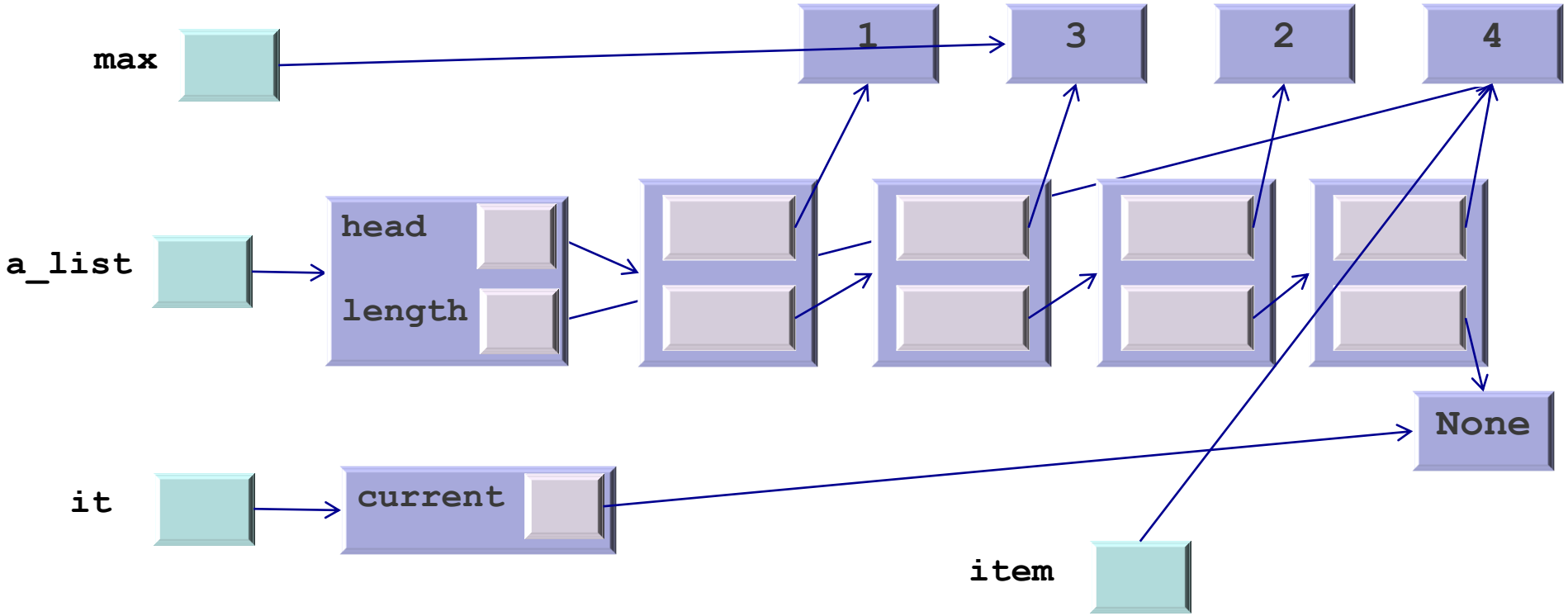

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



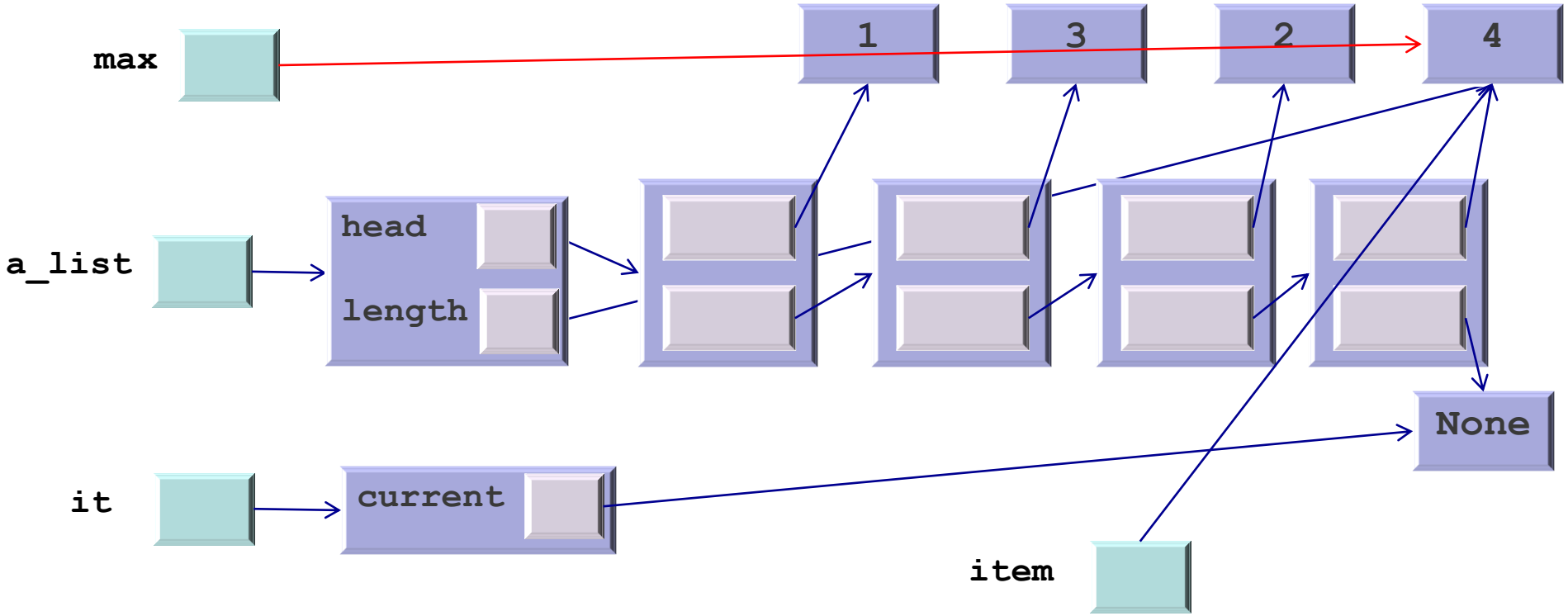

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



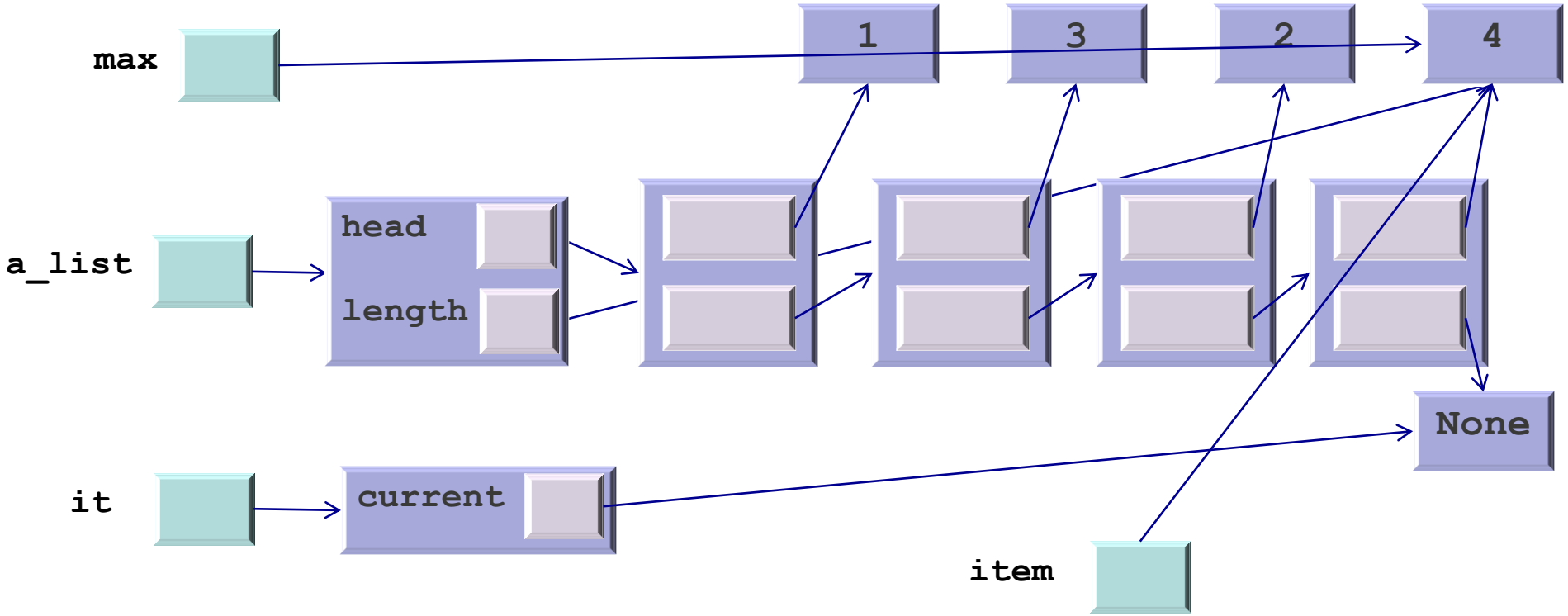

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```



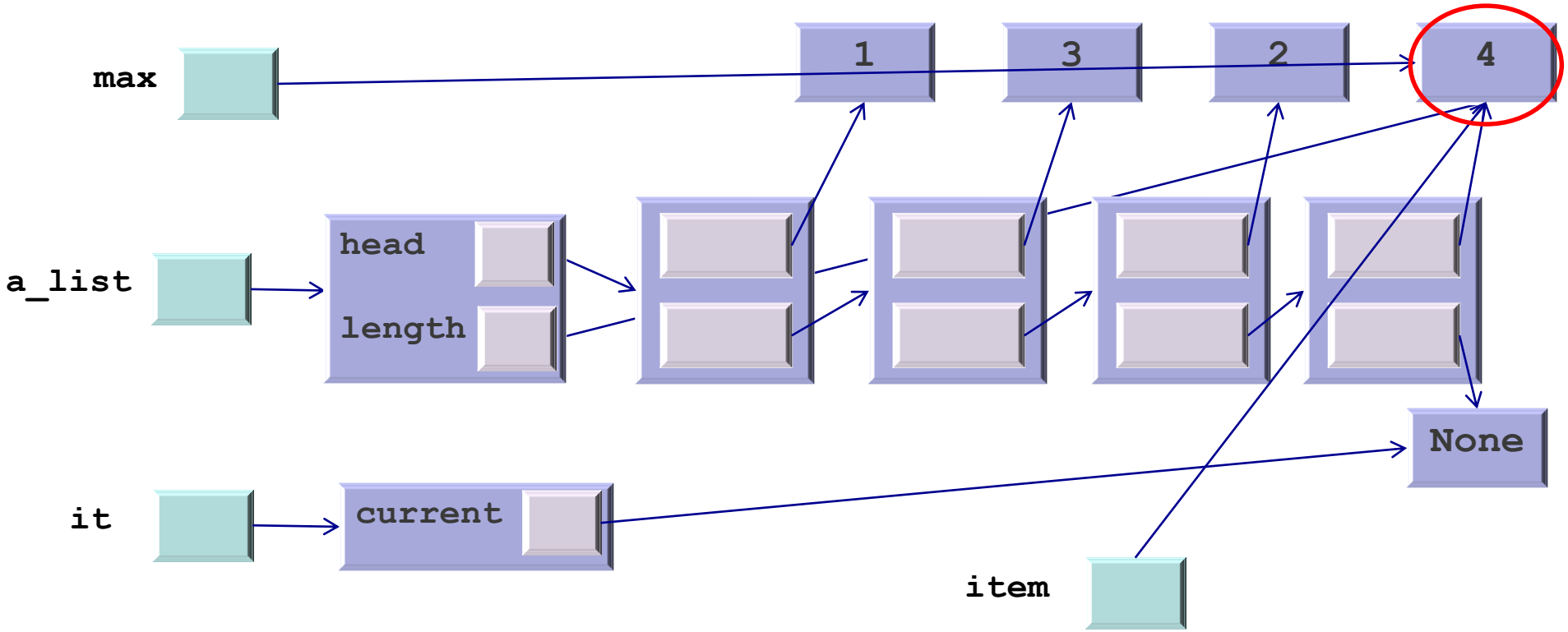

```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```




```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```




```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```




```
def max(a_list: LinkList[T]) -> int:
    it = iter(a_list)       # construct an iterator
    try:
        max = next(it)      # get the first element
    except StopIteration:   # if empty, next(it) raises an exception
        raise Exception("The list is empty")
    else:
        for item in it:     # traverse the rest of the list
            if max < item:
                max = item
        return max
```

MONASH University

A Modifying Iterator

Can we now do double as users?

- **We started this to be able to define the function `double(a_list)`**
  - And do so from outside the class
- **Can we now do that? What do we need?**
  - To traverse the list and add new nodes as we do so
- **Can our iterator add nodes?**

Nope!

- **So we need to create a different kind of iterator**
  - One that allows us to modify the iterable object

A more versatile `LinkListIterator` class

- **Additional operations we will define:**
  - delete: returns the item pointed by **`current`** and deletes the node
  - add: adds an item right before **`current`** (to do in the pracs)
  - has_next: returns **`True`** if there is a next item to be processed (i.e., if **`current`** is not at the end of list)
  - peek: returns the next item (the one pointed by **`current`**) without moving along. Raises **`StopIteration`** if there is no next.
- **Since we need to add and delete:**
  - We are going to need not only a **`current`** but also a **`previous`**
  - Plus some times we will need to change the head of the list
    - We are also going to keep the list itself

```
class LinkList(List[T]):
…
def __iter__(self) -> LinkListIterator[T]:
return LinkListIterator(self)
```

The list, not just the head node

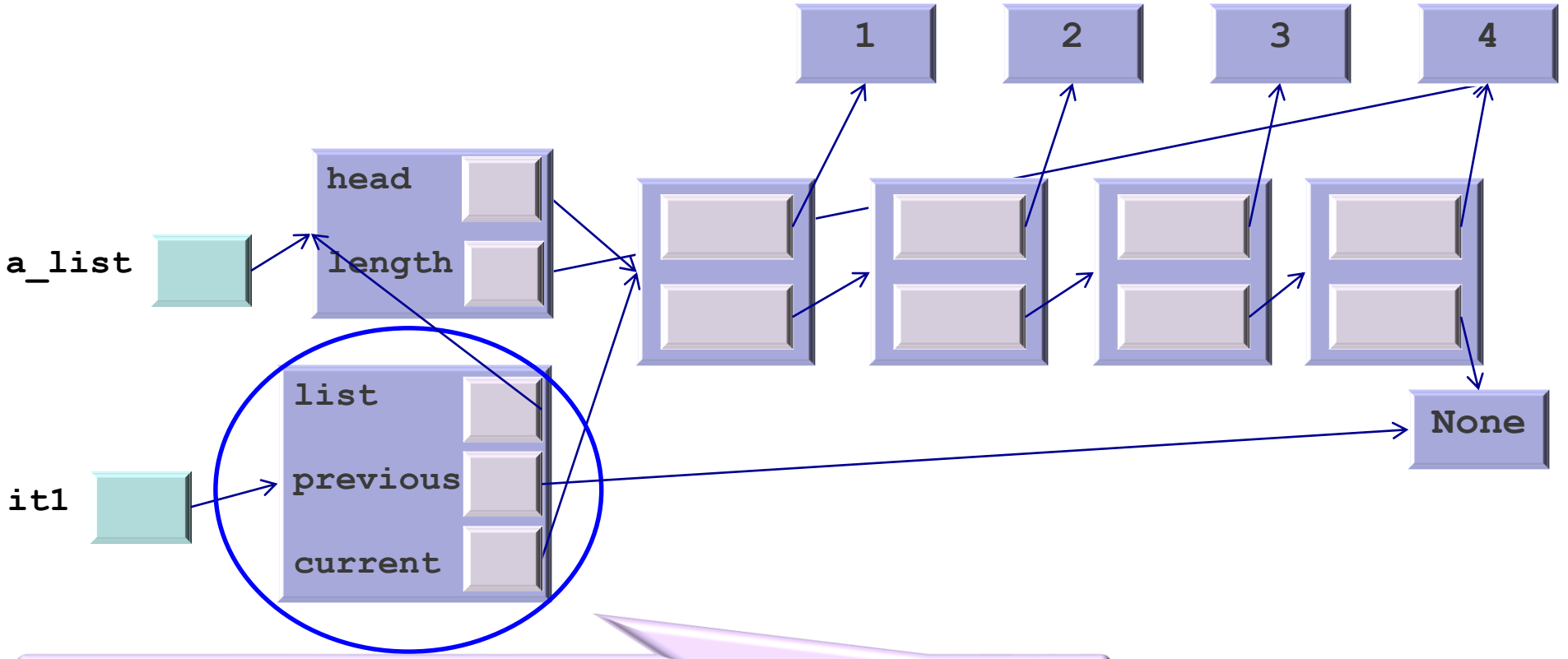
Rethinking `LinkListIterator`

```
class LinkListIterator(Generic[T]):
    def __init__(self, a_list: LinkList[T]) -> None:
        self.list = a_list
        self.previous = None
        self.current = a_list.head

    def __iter__(self):
        return self

    def __next__(self) -> T:
        if self.current is not None:
            item = self.current.item
            self.previous = self.current
            self.current = self.current.link
            return item
        else:
            raise StopIteration
```

The list, not just the head node

We store the list, previous and current

Differences marked in red



This is my new more versatile **`LinkListIterator`** object

```
>>> a_list = List()
>>> a_list.insert(0,4)
>>> a_list.insert(0,3)
>>> a_list.insert(0,2)
>>> a_list.insert(0,1)
>>> it1 = iter(a_list)
>>>
```

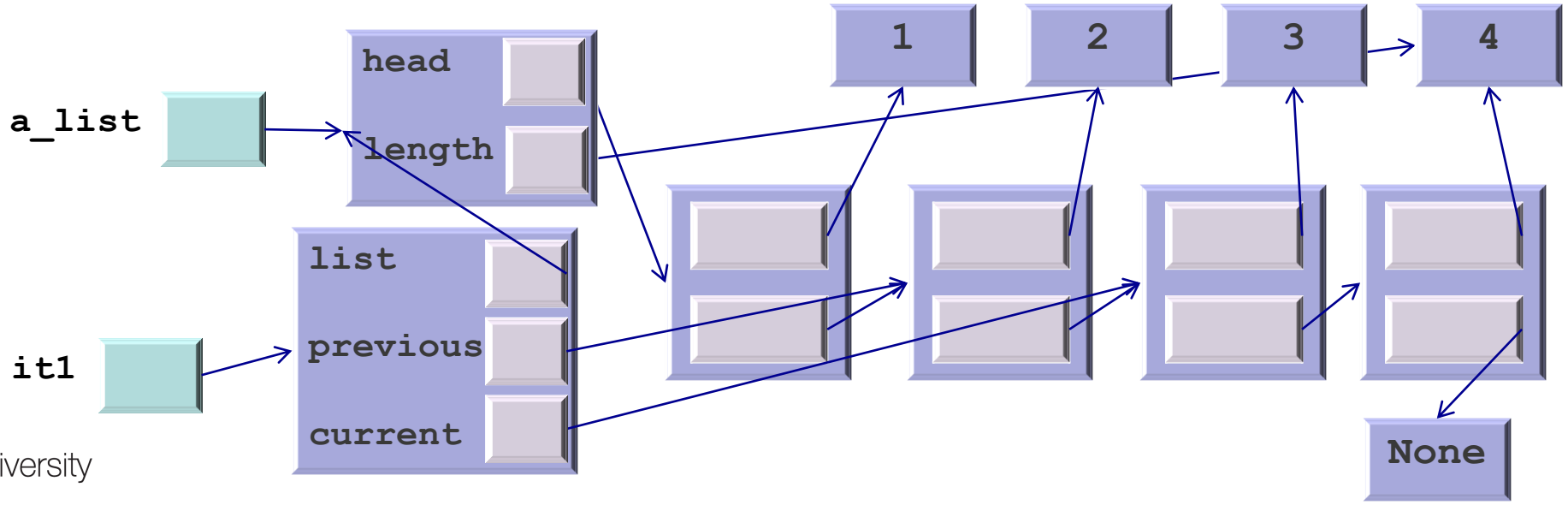
Rethinking `LinkListIterator` (cont)

```
def has_next(self):
    return self.current is not None
```

True if there is at least one item to process

```
def peek(self):
    try: # In case someone tries to peek over the line
        return self.current.item
    except AttributeError:
        raise StopIteration("no more elements in list")
```

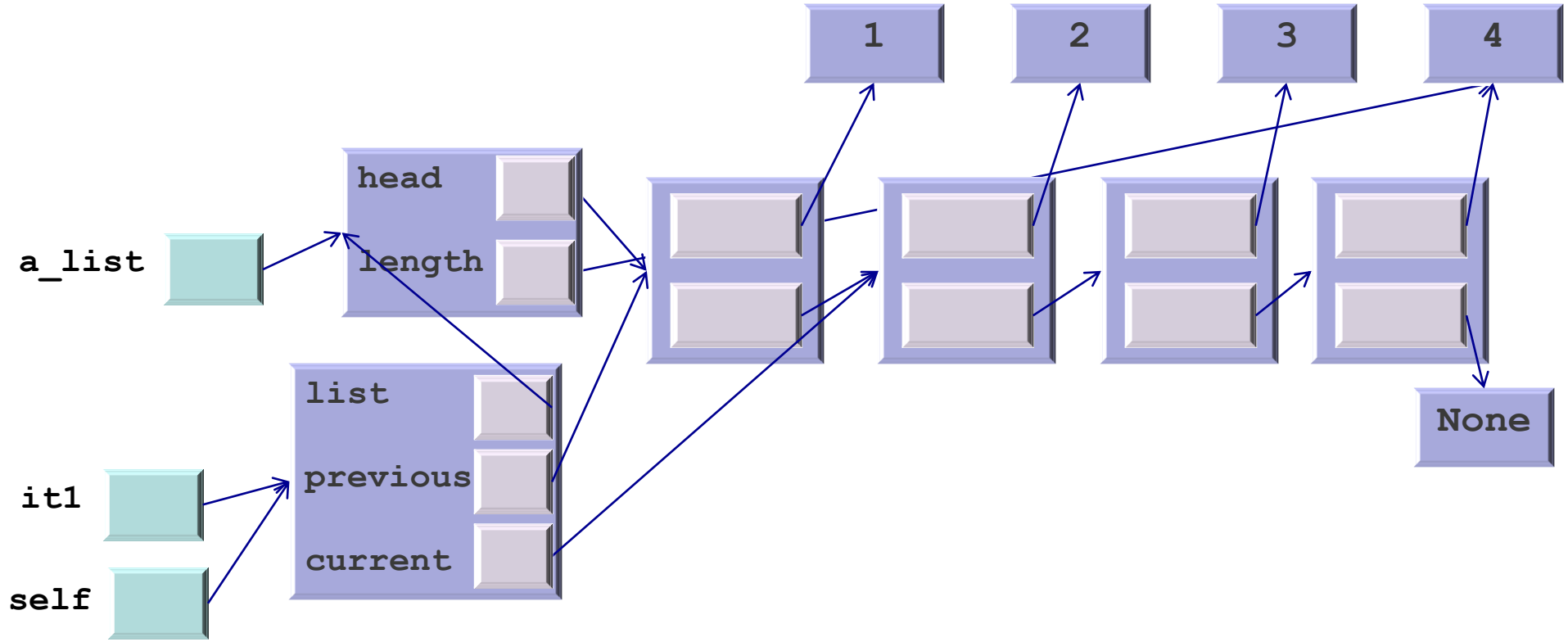
Returns the next item without moving along. Raises `StopIteration` if no next.



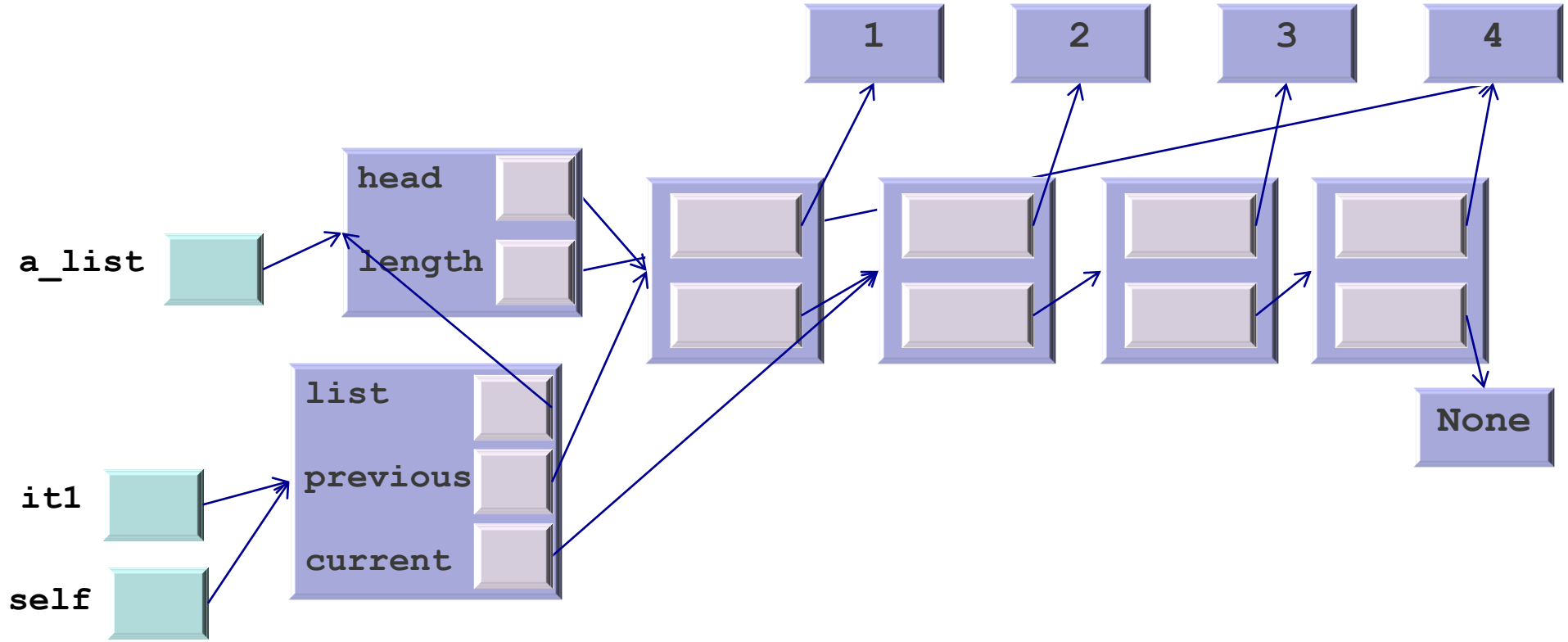
Rethinking `LinkListIterator` (cont)

Returns the next item and deletes the associated node. Raises `StopIteration` if no next.

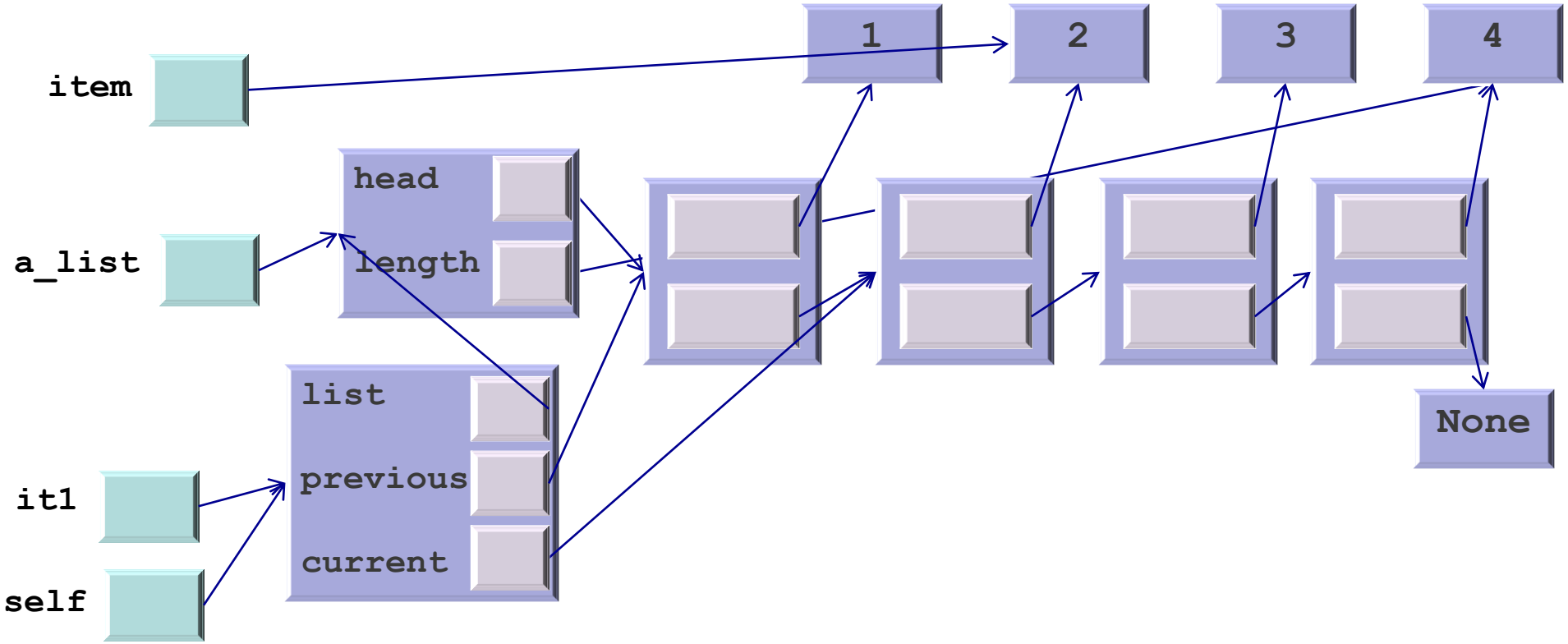
```
def delete(self) -> T:
    if not self.has_next(): # we reached the end
        raise StopIteration("no more elements in list")
    else:
        item = self.current.item
        self.current = self.current.link

        if self.previous is None: #it is the first element
            self.list.head = self.current
        else:                          #not the first element
            self.previous.link = self.current
        self.list.length -= 1

        return item
```



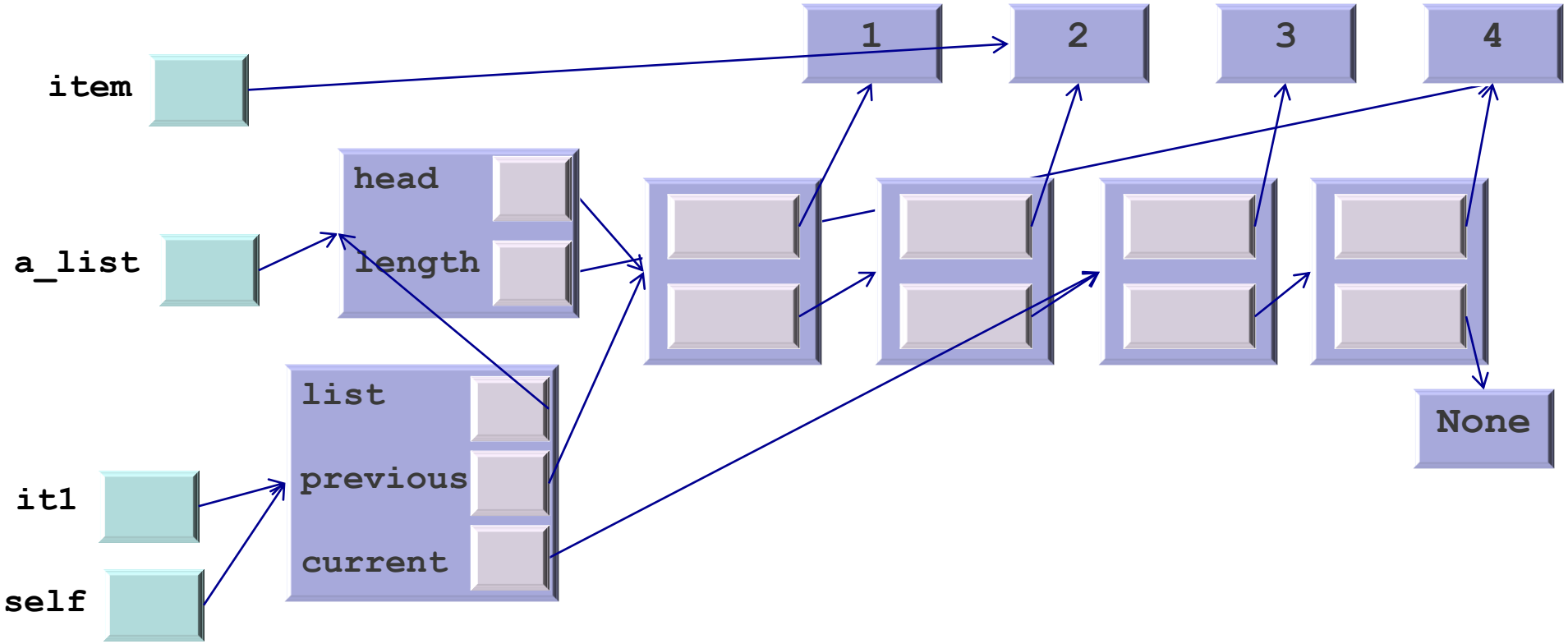

```
    def delete(self) -> T:
        if not self.has_next(): # we reached the end
            raise StopIteration("no more elements in list")
        else:
            item = self.current.item
            self.current = self.current.link
            if self.previous is None: #it is the first element
                self.list.head = self.current
            else:                          #not the first element
                self.previous.link = self.current
            self.list.length -= 1
            return item
```



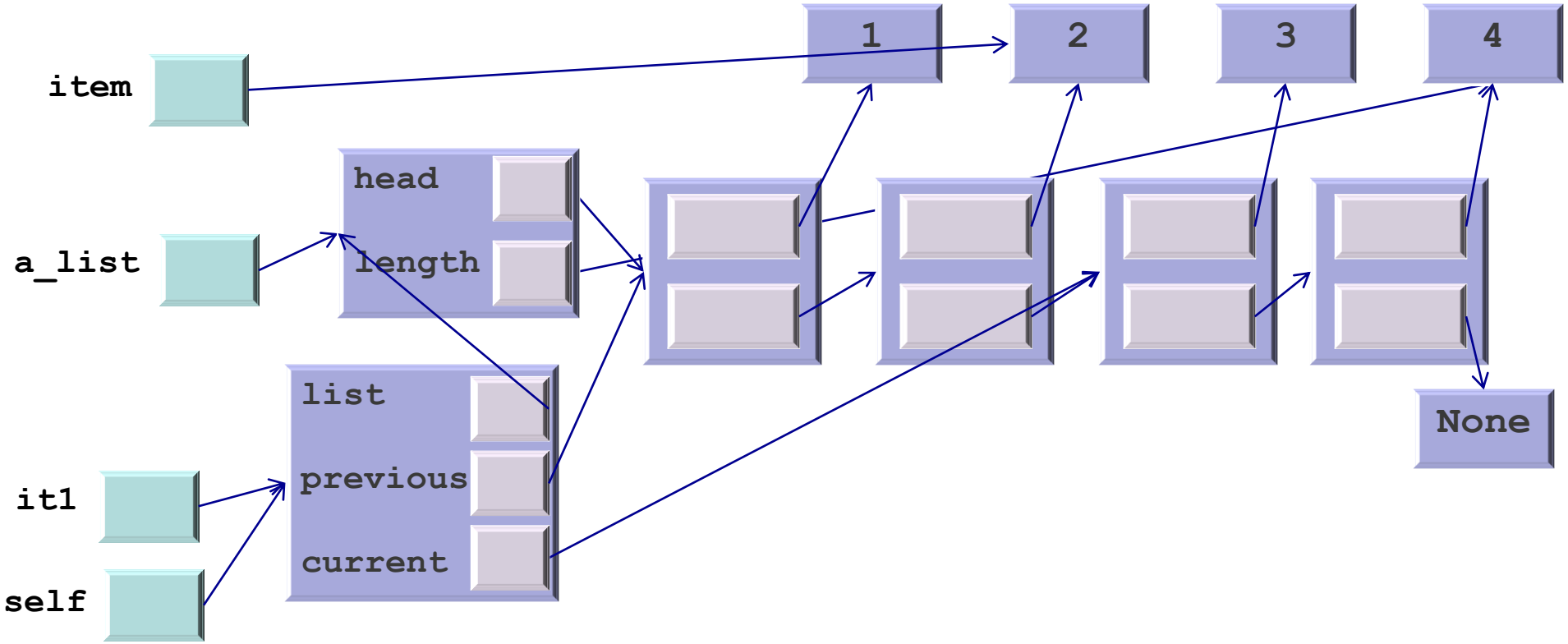

```
    def delete(self) -> T:
        if not self.has_next(): # we reached the end
            raise StopIteration("no more elements in list")
        else:
            item = self.current.item
            self.current = self.current.link
            if self.previous is None: #it is the first element
                self.list.head = self.current
            else:                          #not the first element
                self.previous.link = self.current
            self.list.length -= 1
            return item
```



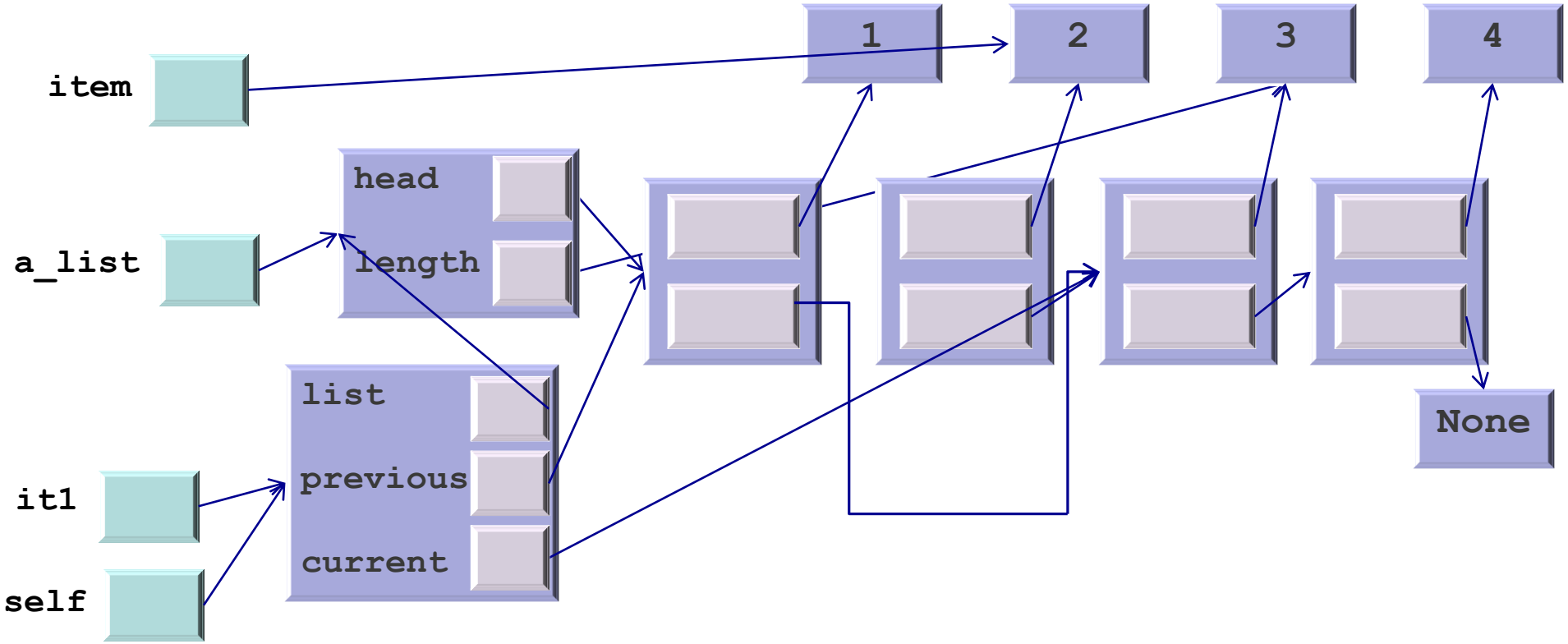

```
def delete(self) -> T:
    if not self.has_next(): # we reached the end
        raise StopIteration("no more elements in list")
    else:
        item = self.current.item
        self.current = self.current.link
        if self.previous is None: #it is the first element
            self.list.head = self.current
        else:                          #not the first element
            self.previous.link = self.current
        self.list.length -= 1
        return item
```




```
def delete(self) -> T:
    if not self.has_next(): # we reached the end
        raise StopIteration("no more elements in list")
    else:
        item = self.current.item
        self.current = self.current.link
        if self.previous is None: #it is the first element
            self.list.head = self.current
        else:                          #not the first element
            self.previous.link = self.current
        self.list.length -= 1
        return item
```




```
    def delete(self) -> T:
        if not self.has_next(): # we reached the end
            raise StopIteration("no more elements in list")
        else:
            item = self.current.item
            self.current = self.current.link
            if self.previous is None: #it is the first element
                self.list.head = self.current
            else:                          #not the first element
                self.previous.link = self.current
            self.list.length -= 1
            return item
```




```
    def delete(self) -> T:
        if not self.has_next(): # we reached the end
            raise StopIteration("no more elements in list")
        else:
            item = self.current.item
            self.current = self.current.link
            if self.previous is None: #it is the first element
                self.list.head = self.current
            else:                          #not the first element
                self.previous.link = self.current
            self.list.length -= 1
            return item
```

MONASH University

Basics of Higher Order Functions in Python

Functions as first-class objects/entity

- **Being a first-class object means being like any other object:**
  - Functions can then be assigned to variables
  - Passed as arguments to a function/method
  - Returned by a function/method
- **A higher order function is one that:**
  - Takes another function as argument and/or
  - Returns another function as argument
- **In Python functions (and thus methods) are first class objects**
  - So you can define higher order functions

```
>>> def my_call(f,x):
...     f(x)
...
```

```
>>> def my_function(x):
...     print(x)
...
>>> my_call(my_function, "hello")
hello
```

Takes a function as argument

Functions as first-class objects (cont)

- **You can also define a function that returns another function:**

```
>>> def return_f(x):
...     def f(y):
...         return x+y
...     return f
...
>>> g = return_f(30)
>>> g
<function return_f….>
>>> g(2)
32
```

`g` is bound to a function that adds 30 to its argument

```
>>> def return_h():
...     def h(x,y):
...         return x+y
...     return h
...
>>> j = return_h()
>>> j
<function return_h….>
>>> j(30,2)
32
```

`j` is bound to a function that adds its two arguments

Let's combine the two

- **Let's define a function that:**
  - Receives a list `a_list` and a function `f` as input
  - Returns a new function `h` that receives another function `g` and applies first `g` and then `f` to every element in `a_list`

```
def combine(f,a_list):
    def h(g):
        return [f(g(x)) for x in a_list]
    return h
```

```
>>> def plus_10(x):
...     return x + 10
...
>>> def mult_by2(x):
...     return x*2
...
```

```
>>> h = combine(plus_10,[1,2,3,4])
>>> h(mult_by2)
[12, 14, 16, 18]
>>>
```

Summary

- **We have looked at how to make lists iterable by implementing an iterator for them**
  - A simple traversal iterator
  - A modifying iterator (more Java like)
- **Briefly looked at higher order functions in Python**