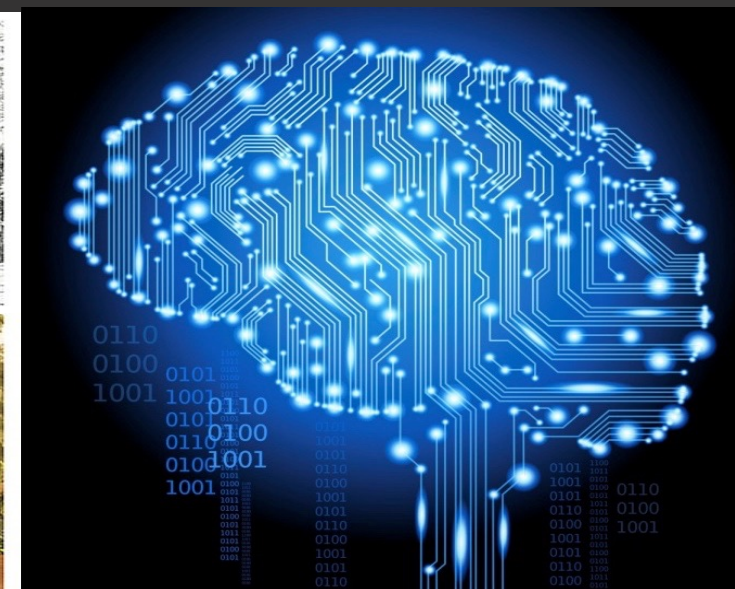
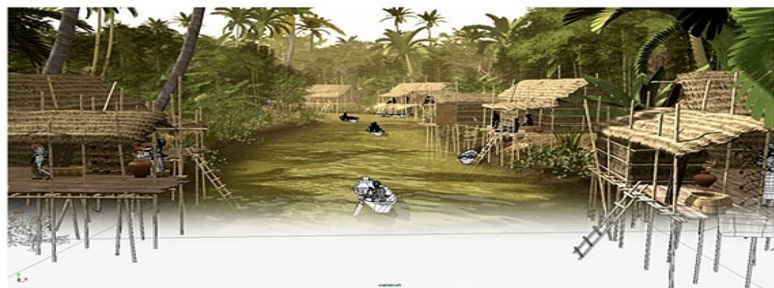
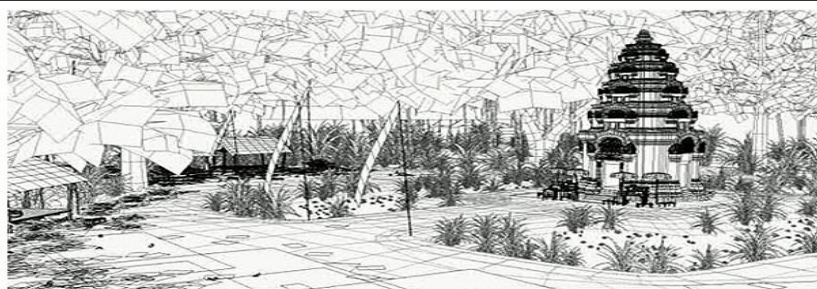




# Linked Stacks & Queues

Prepared by Maria Garcia de la Banda  
Updated by Brendon Taylor



# Objectives for this lesson

- **To understand the use of linked data structures in implementing**
  - Stacks
  - Queues
- **To be able to:**
  - Implement, use and modify linked stacks and linked queues
  - Decide when it is appropriate to use them (rather than arrays)

# Linked Stacks

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

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

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

    @abstractmethod
    def pop(self) -> None:
        pass

    @abstractmethod
    def peek(self) -> T:
        pass

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

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

# Remember:

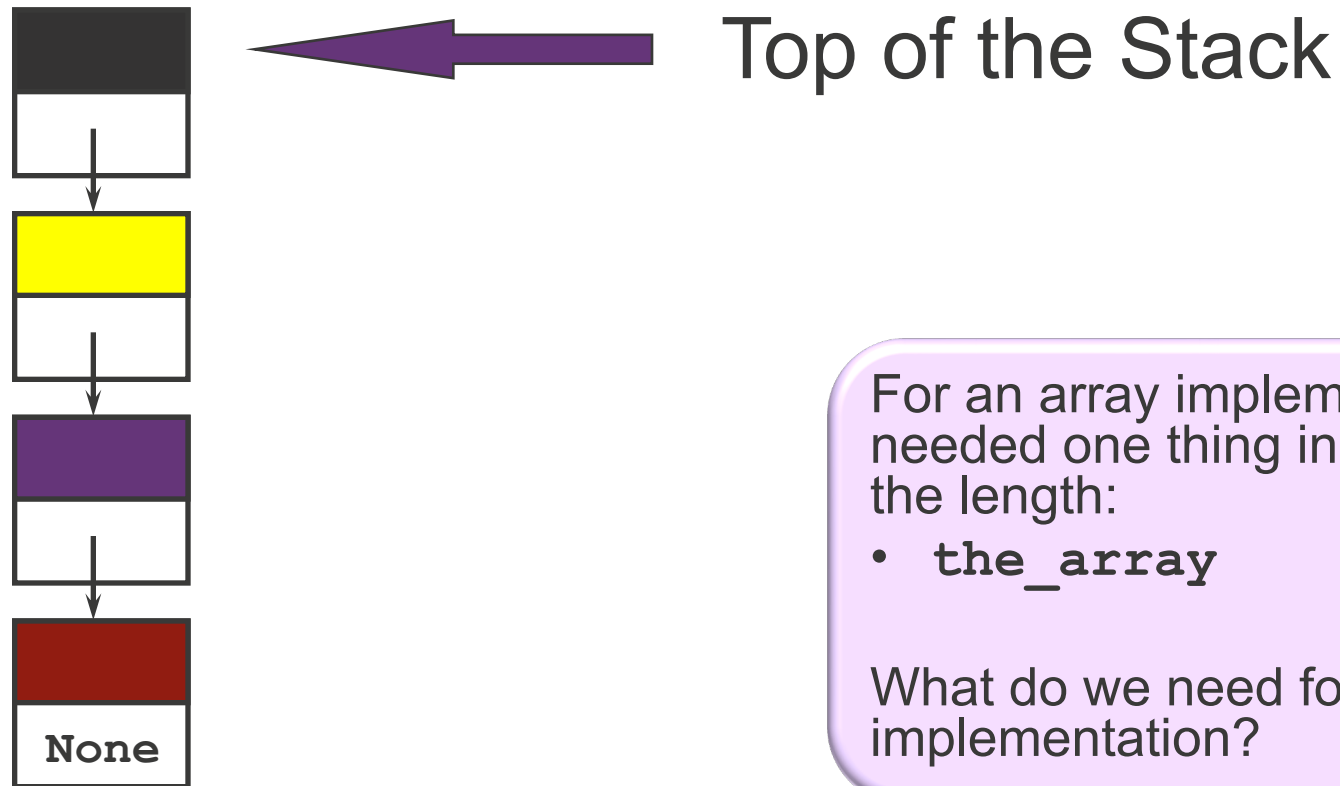
## Abstract base Stack class

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

```
@abstractmethod
def is_full(self) -> bool:
    pass
```

```
@abstractmethod
def __toString__(self) -> str:
    pass
```

# Linked Stack implementation



For an array implementation we needed one thing in addition to the length:

- `the_array`

What do we need for a linked implementation?

Nodes!

# Class for a Linked Stack

```
from typing import TypeVar
from abstract_stack import Stack
from node import Node
T = TypeVar('T')
```

No need for **size** when initialising the object

```
class LinkStack(Stack[T]):
    def __init__(self):
        Stack.__init__(self)
        self.top = None
```

```
    def is_full(self):
        return False
```

Big O?

O(1)

```
    def clear(self):
        Stack.clear()
        self.top = None
```

Did not do that for LinkLists,  
but it is good to free memory

# Push method for Linked Stacks

# Push: algorithm

- In the **array** implementation:

- If the array is full: raise exception (or resize, if we wanted to do that)
- Else
  - Add the item in the position marked by top (was the same as the length of the list)
  - Increase top

- In a **linked data structure**:

- Create a new node that contains the item
- We link it to the current top
- Make the new node the new top

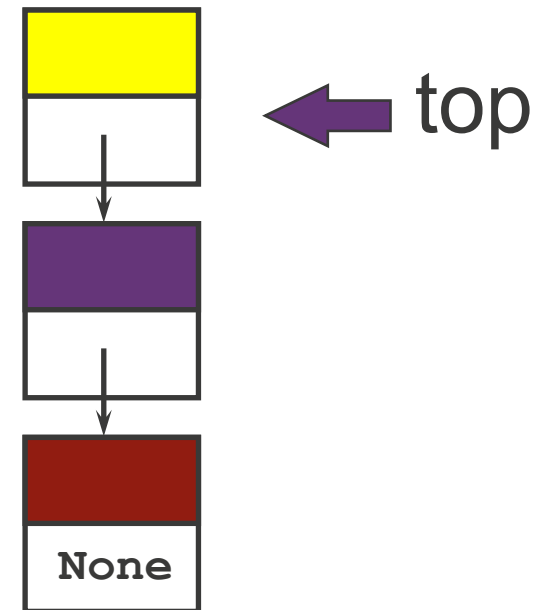
- No need for **is\_full** check

- If no more memory can be allocated:

- The system will raise an **exception**

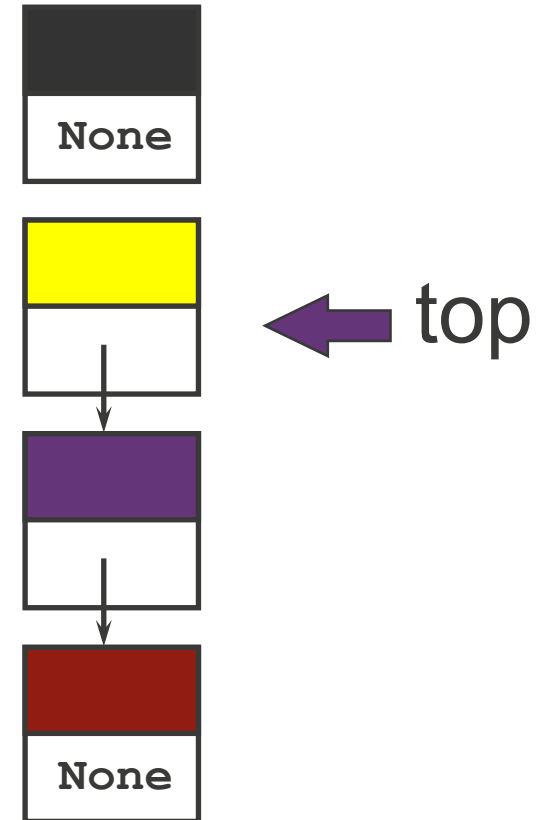


# Push: algorithm

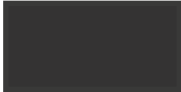


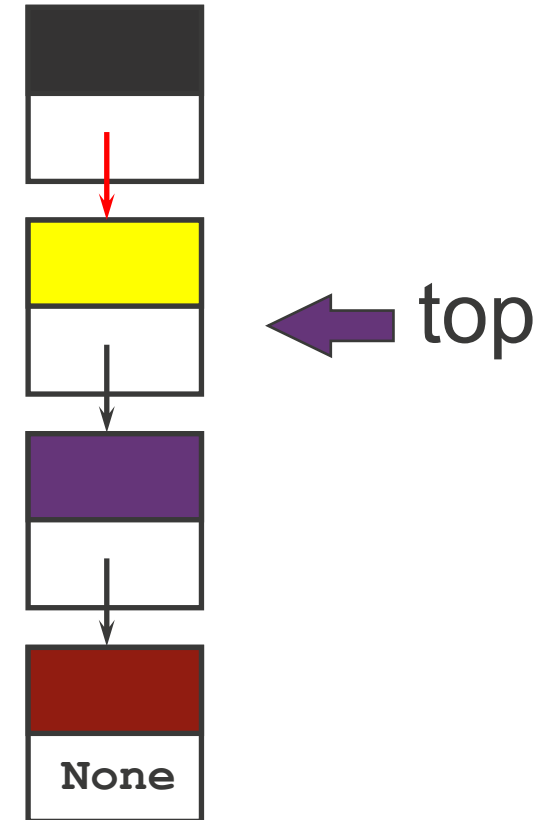
# Push: algorithm

- Create a new node for item

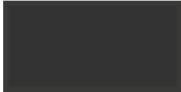


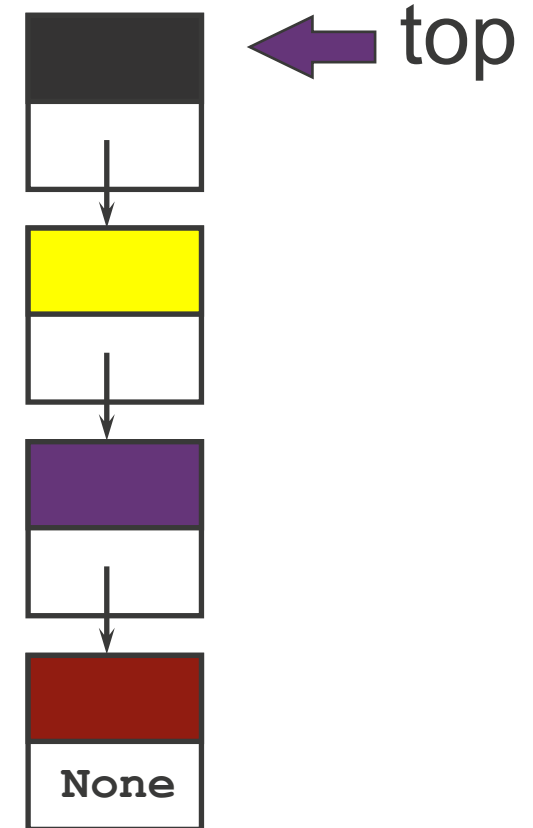
# Push: algorithm

- Create a new node for item 
- Link it to the current top node



# Push: algorithm

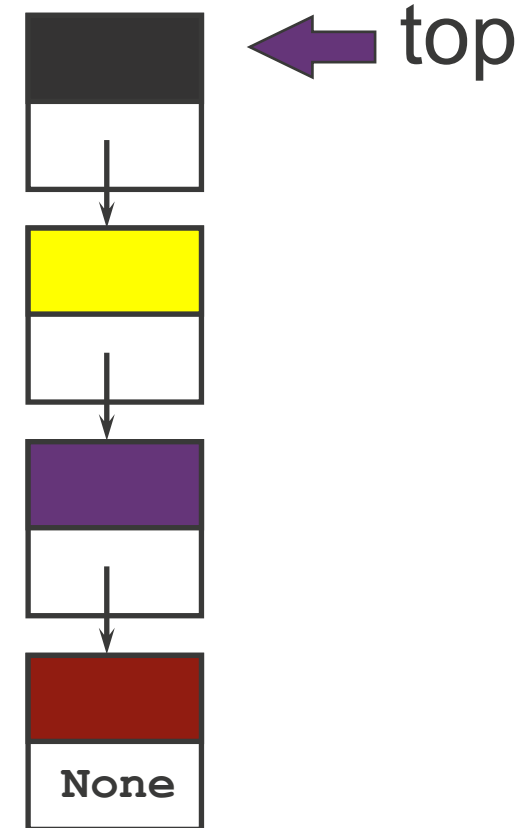
- Create a new node for item 
- Link it to the current top node
- **Make the new node the new top**



# Push: algorithm

- Create a new node for item
- Link it to the current top node
- Make the new node the new top

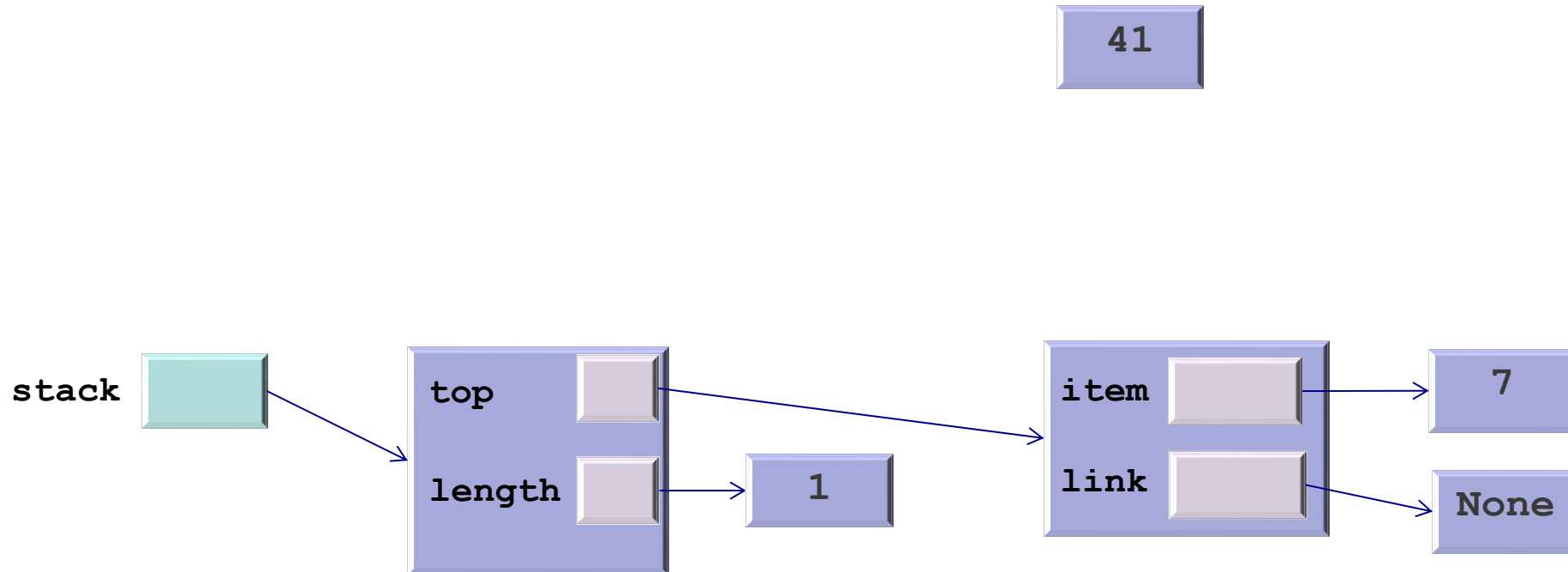
```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```



Consider a **stack**  
with a node whose  
item is **7**

Lets see the memory  
diagram for  
**stack.push(41)**

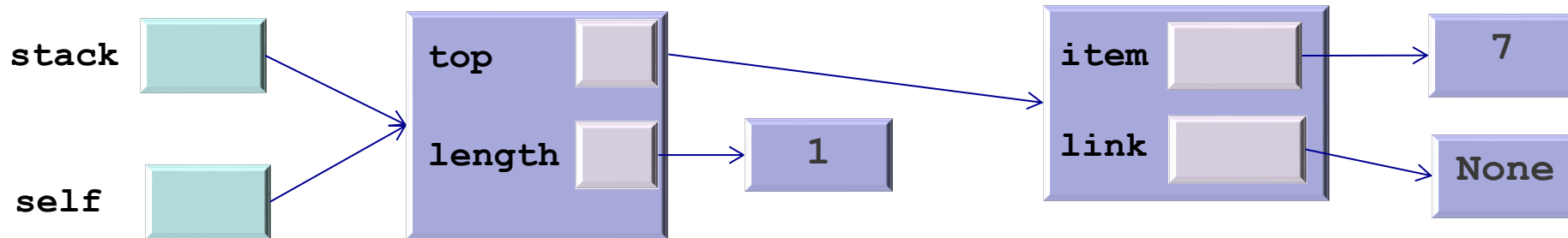
```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```



Consider a **stack**  
with a node whose  
item is 7

Lets see the memory  
diagram for  
**stack.push(41)**

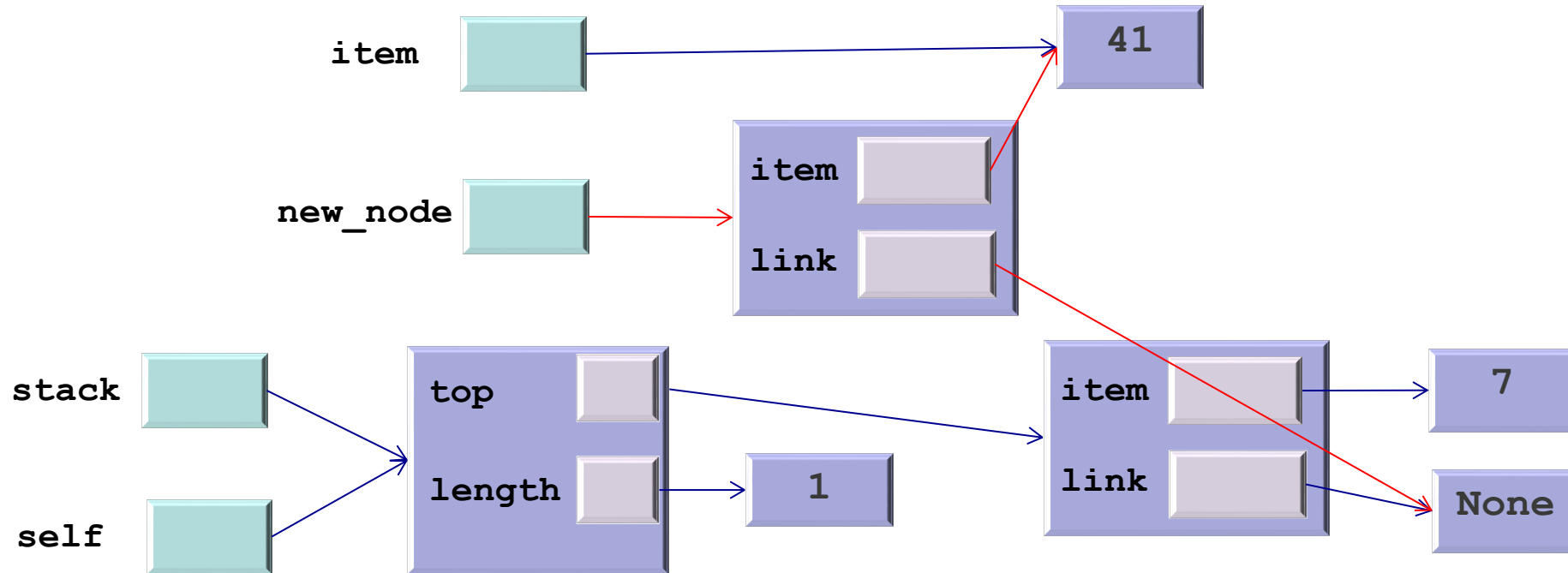
```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```



Consider a **stack**  
with a node whose  
item is 7

Lets see the memory  
diagram for  
**stack.push(41)**

```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```

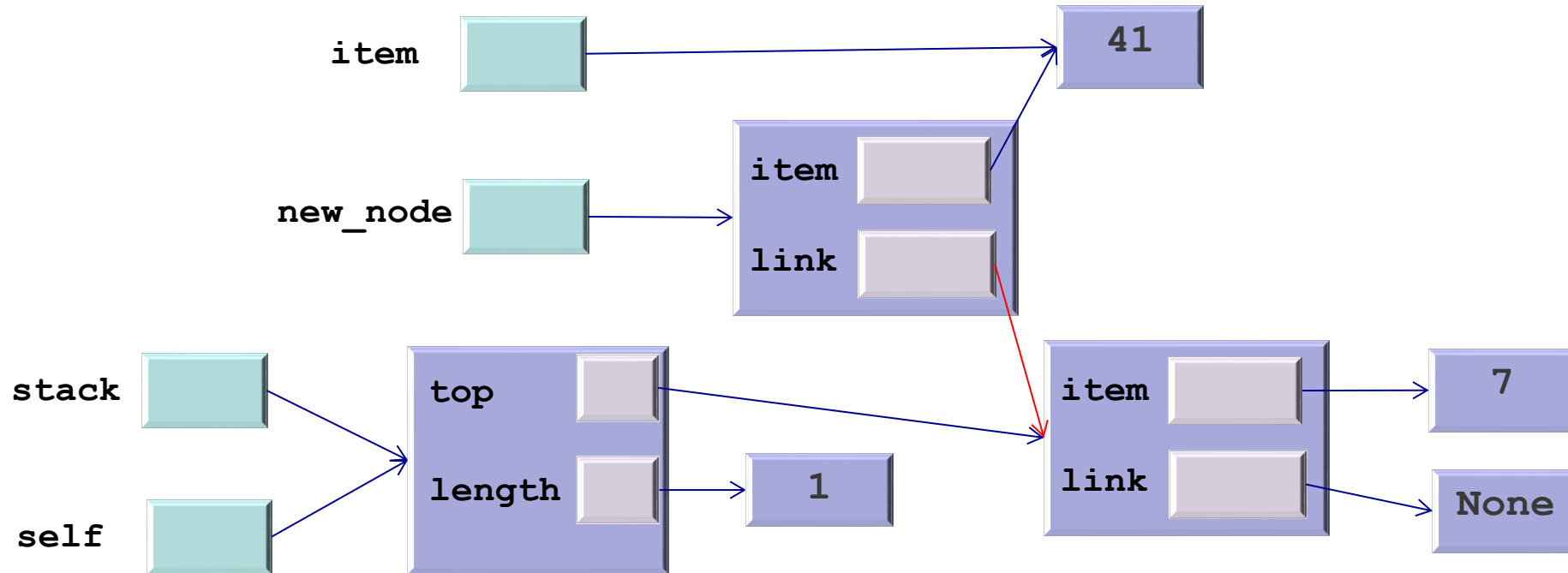




Consider a **stack**  
with a node whose  
item is 7

Lets see the memory  
diagram for  
**stack.push(41)**

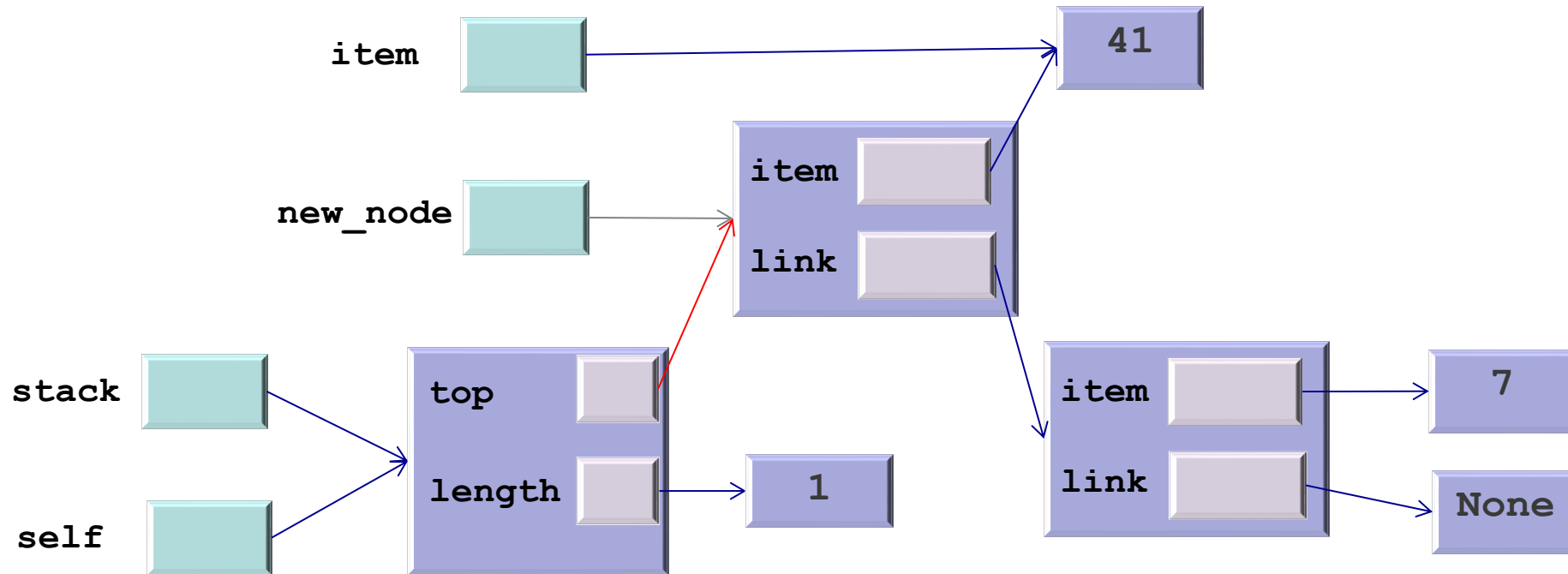
```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```



Consider a **stack**  
with a node whose  
item is 7

Lets see the memory  
diagram for  
**stack.push(41)**

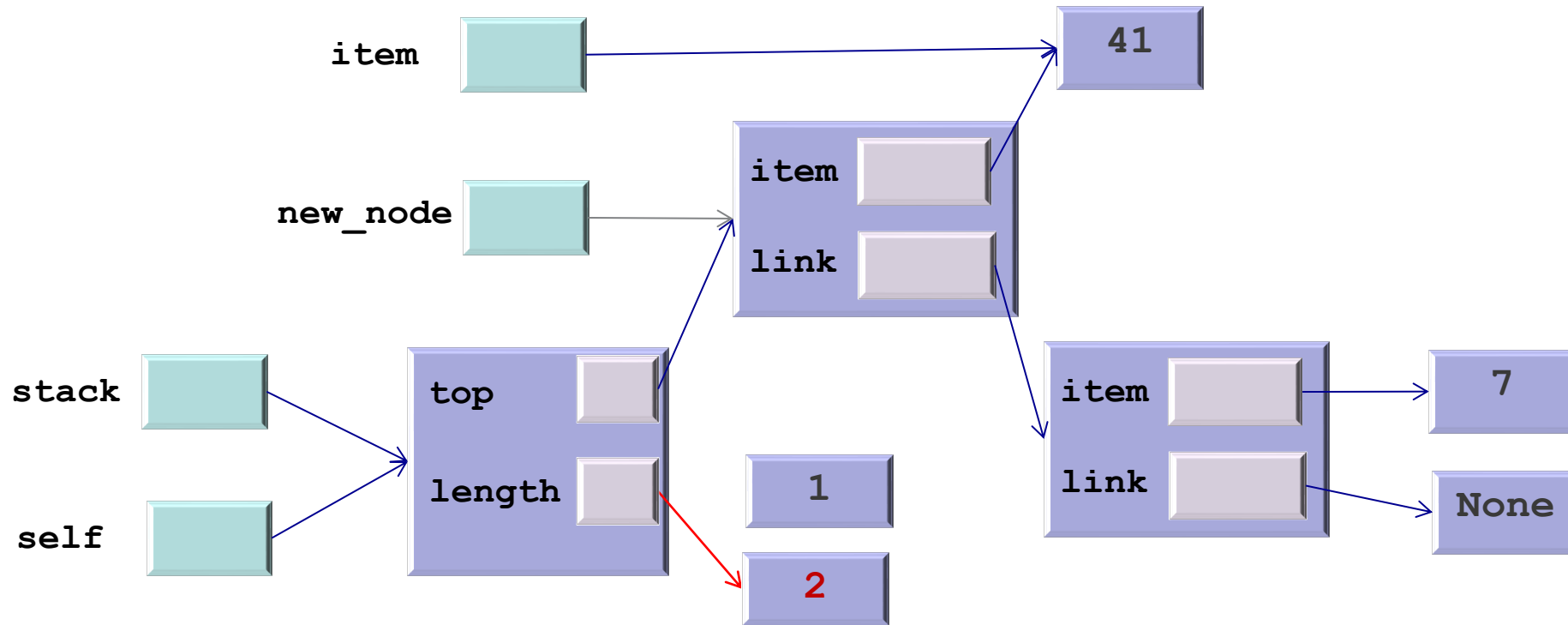
```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```



Consider a **stack**  
with a node whose  
item is 7

Lets see the memory  
diagram for  
**stack.push(41)**

```
def push(self, item: T):  
    new_node = Node(item)  
    new_node.link = self.top  
    self.top = new_node  
    self.length += 1
```



# Pop method for Linked Stacks

# Pop algorithm

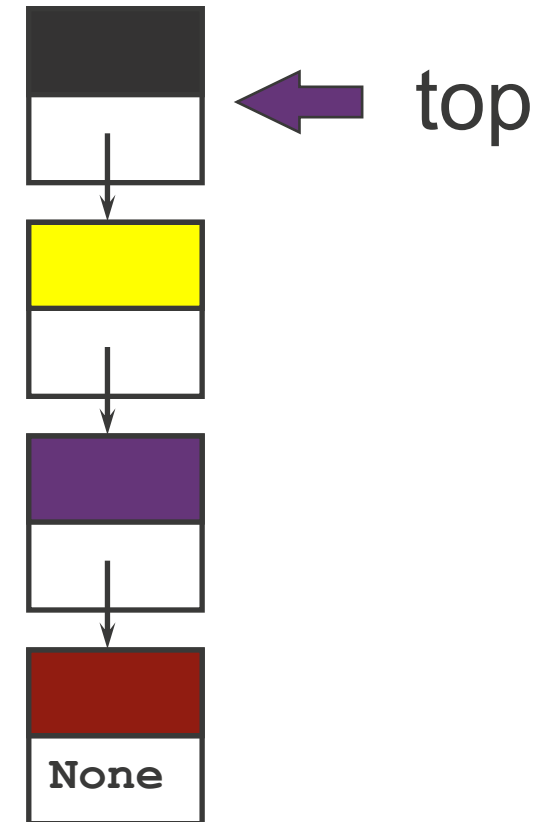
- **Array implementation:**

- If it is empty: raise exception
- Else:
  - Remember the top item
  - Decrease top
  - Return the item

- **Linked nodes:**

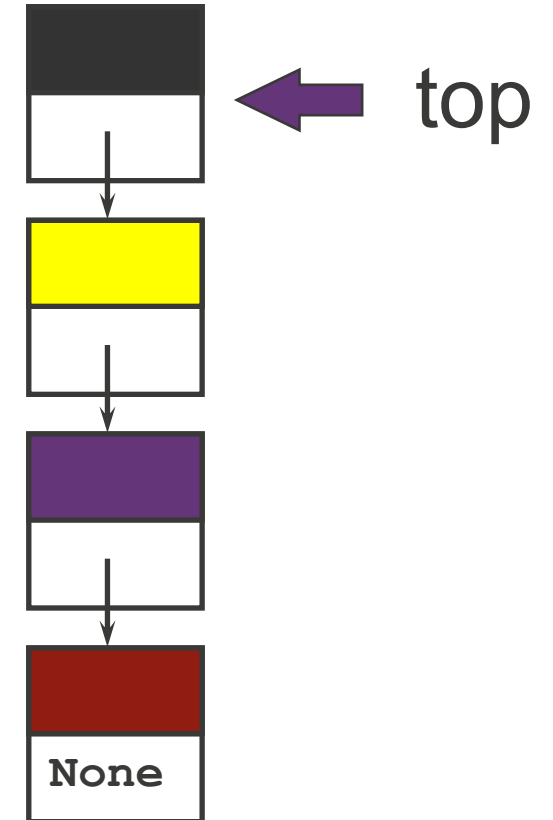
- Almost identical
- We simply move top along, rather than increase it

# Pop: algorithm



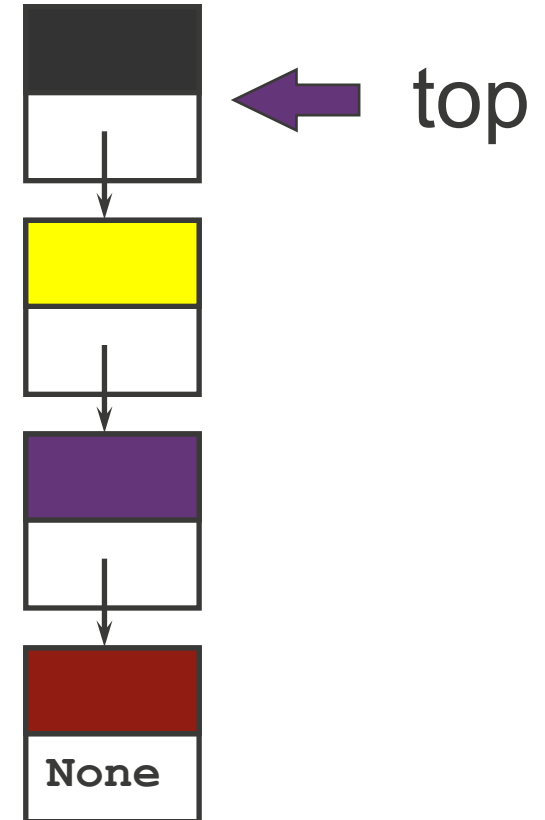
# Pop: algorithm

- Check if the stack is empty



# Pop: algorithm

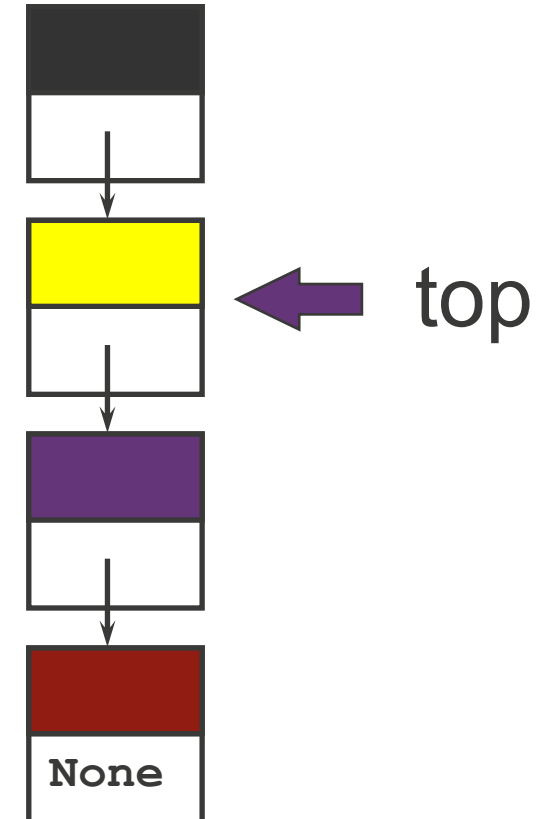
- Check if the stack is empty
- Remember the item in the top node





# Pop: algorithm

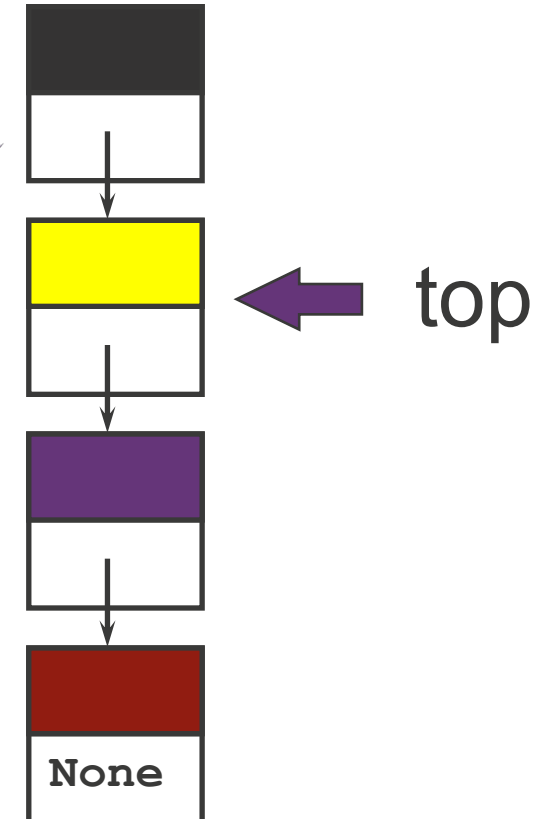
- Check if the stack is empty
- Remember the item in the top node
- **Make the next node the new top**



# Pop: algorithm

- Check if the stack is empty
- Remember the item in the top node
- Make the next node the new top
- **Return the item**

As usual, no need to do anything about this. Python will automatically free the memory



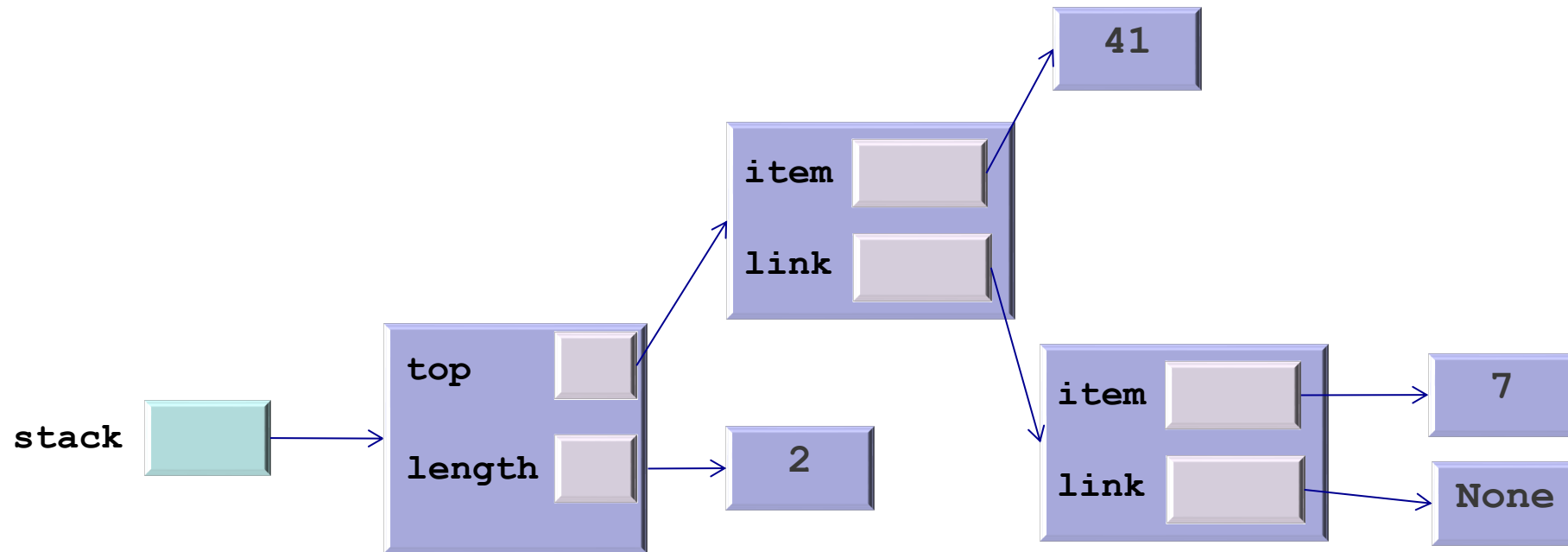
# Pop: algorithm and method

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```

**Complexity?  $O(1)$**

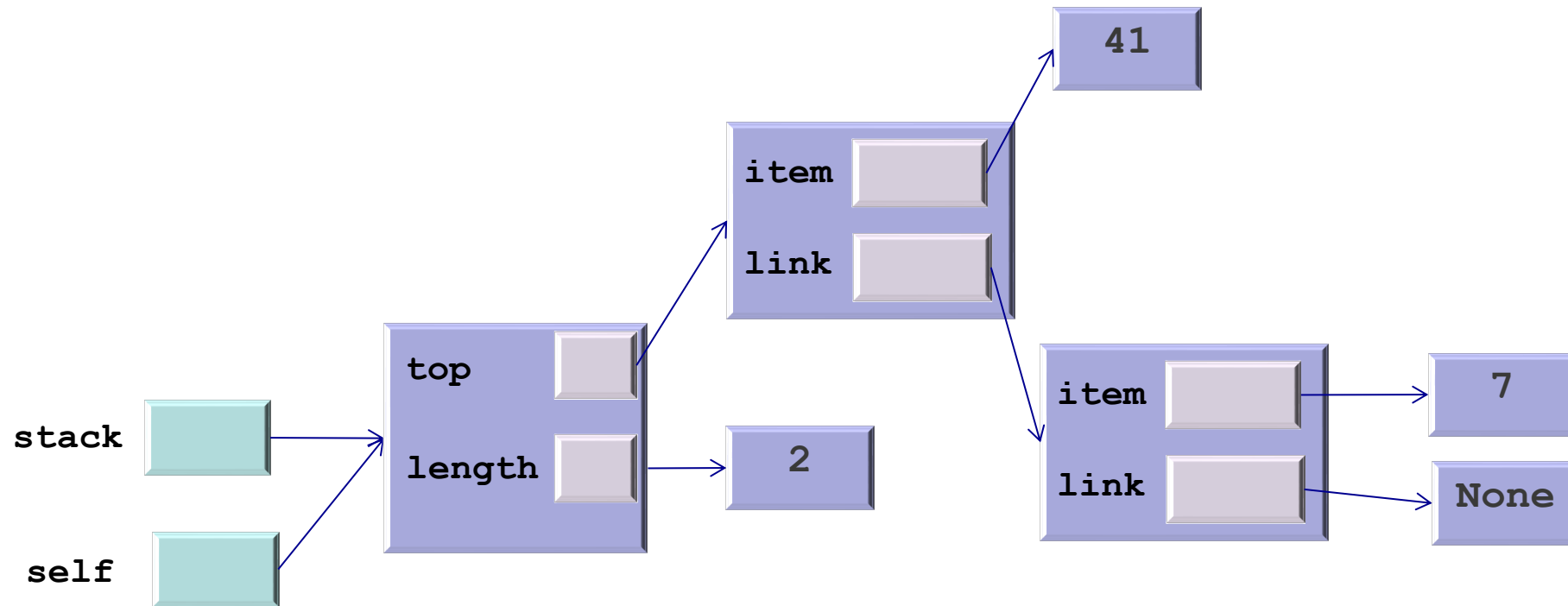
Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```



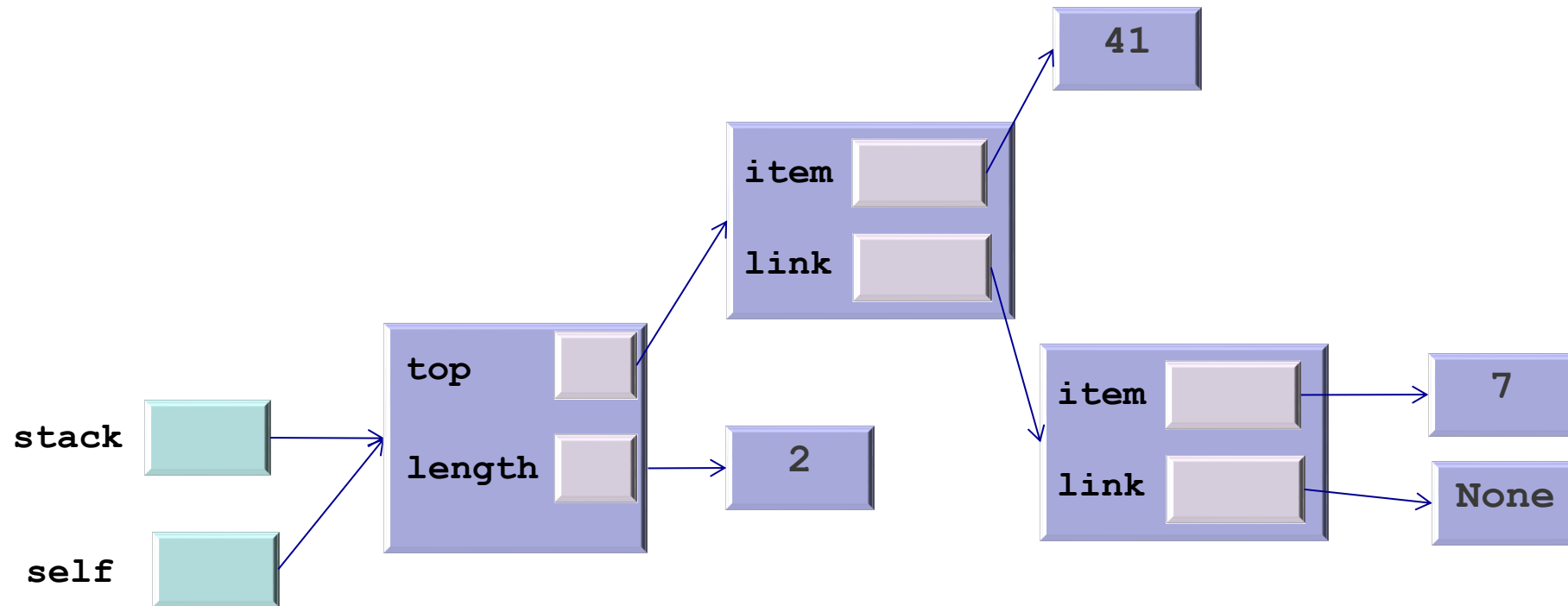
Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```



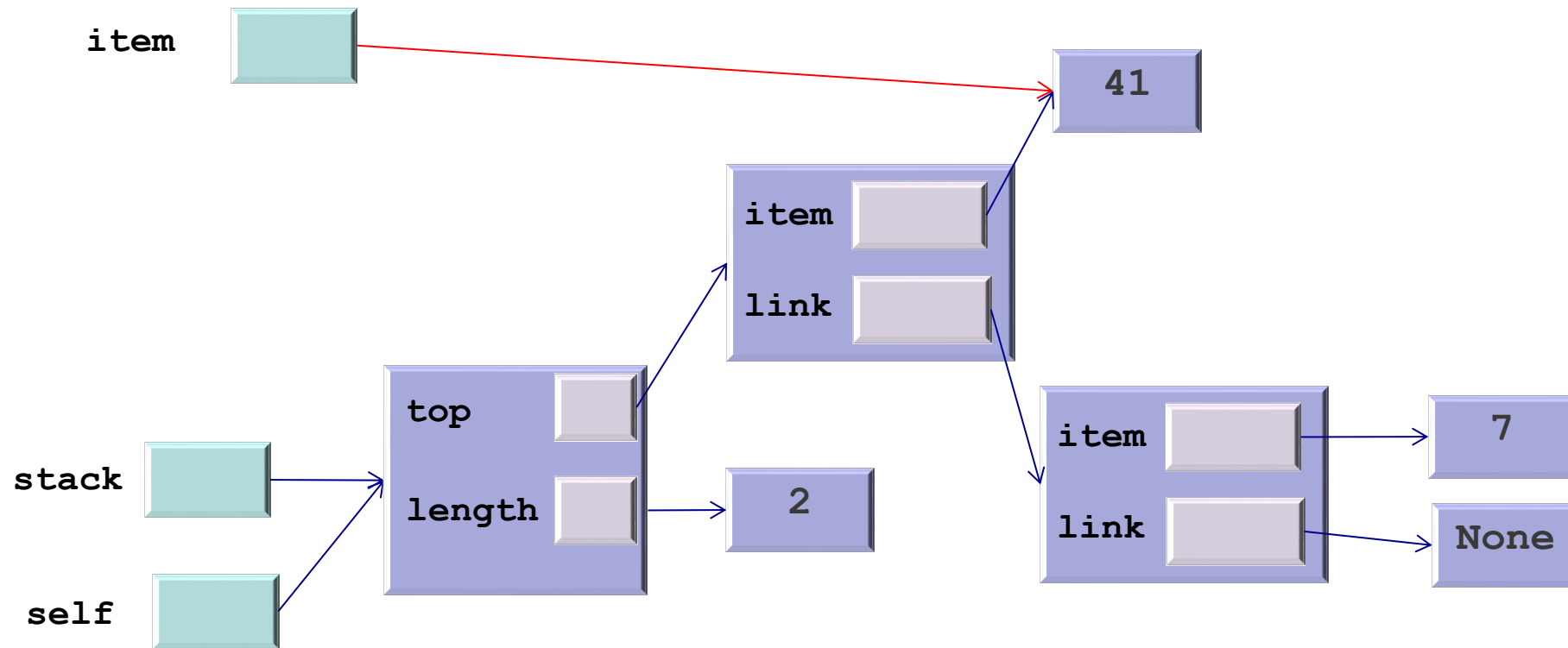
Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```



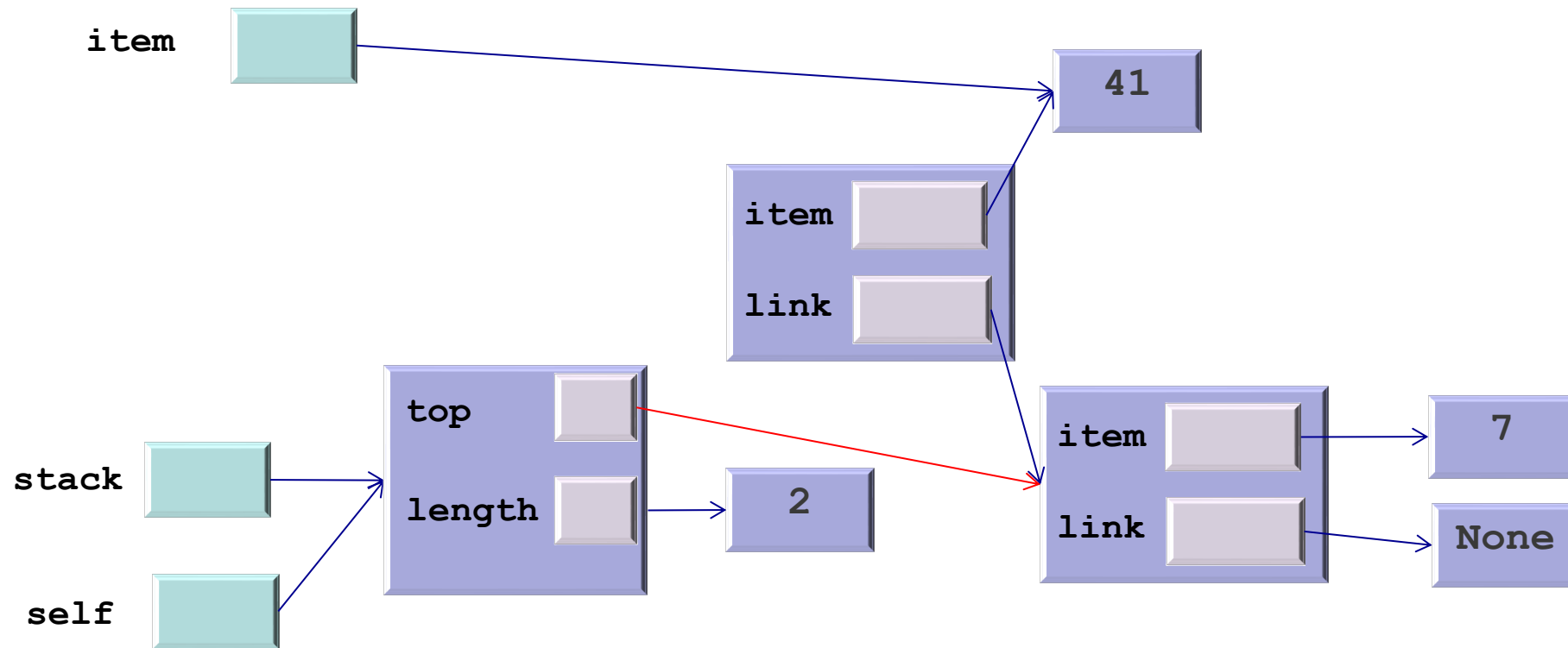
Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```



Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

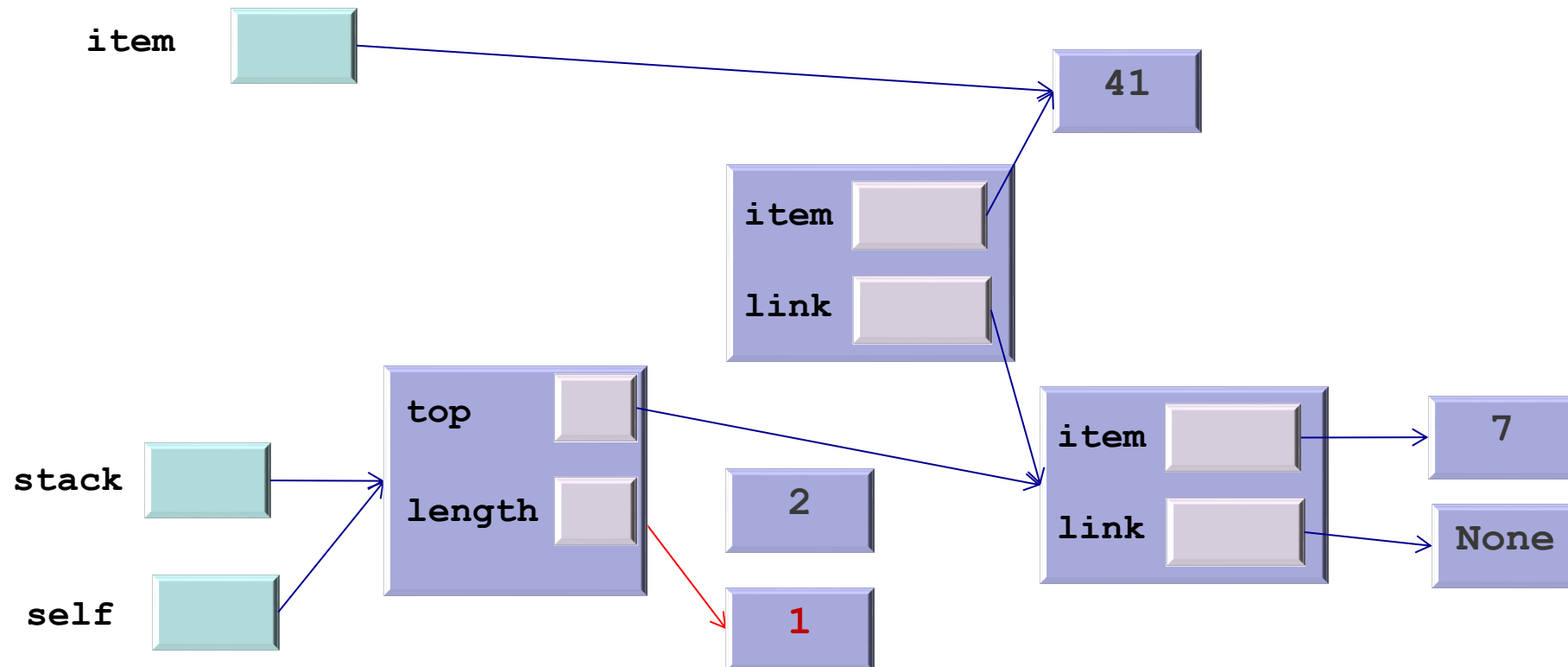
```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```





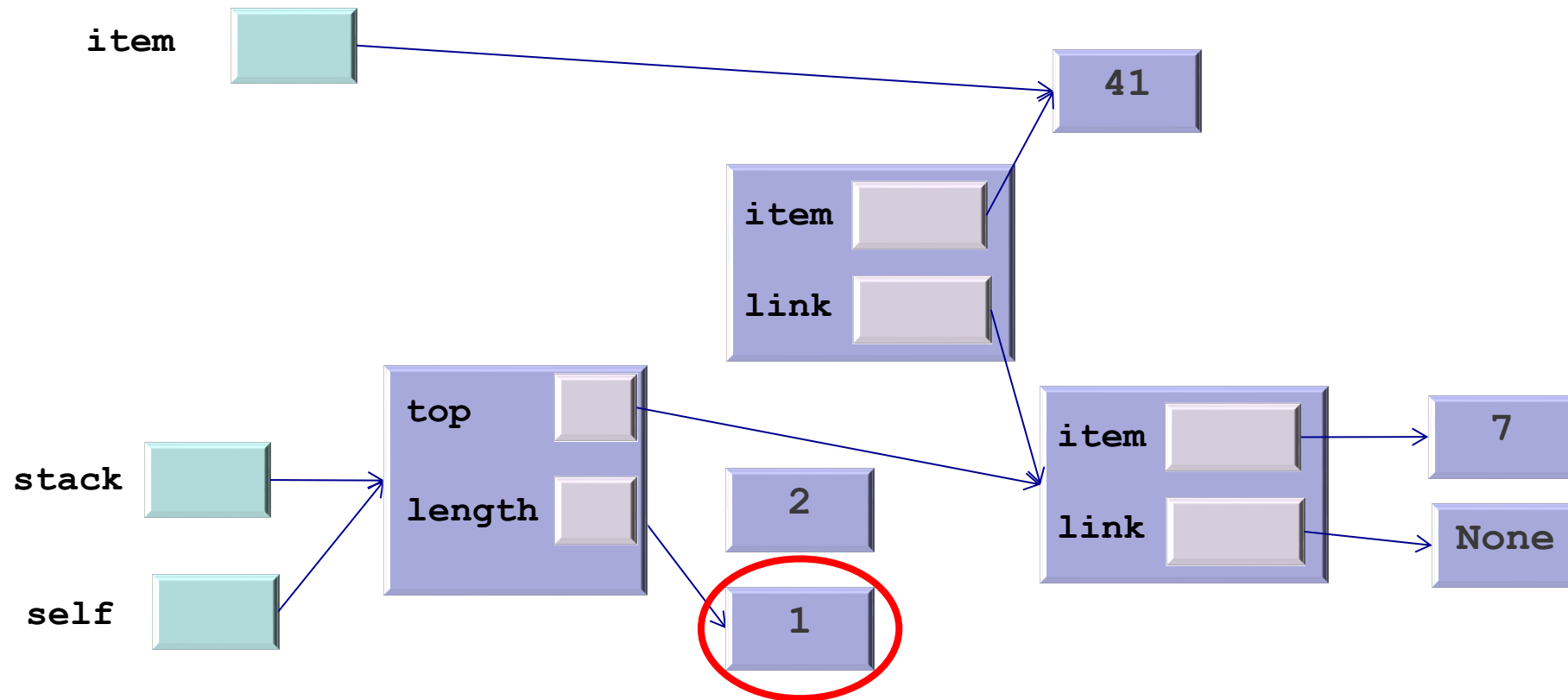
Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```



Consider a **stack**  
with two nodes whose  
items are **41** and **7**  
Let's see the memory  
diagram for  
**stack.pop()**

```
def pop(self) -> T:
    if not self.is_empty():
        item = self.top.item
        self.top = self.top.link
        self.length -= 1
        return item
    else:
        raise ValueError("Stack is empty")
```



# Example: modify for using linked stacks

```
def reverse(string: str) -> str:
    my_stack = ArrayStack(len(string))

    for char in string:
        my_stack.push(char)

    output = ""

    while not my_stack.is_empty():
        char = my_stack.pop()
        output += char

    return output
```

What needs to change?

Only the class name for instantiating the object

That is the point of ADTs!

# Advantages/Disadvantages for Stacks

## ▪ Main advantages:

- Good to **resize**:
  - Push: never full so no need to copy, just add element at top
  - Pop: uses less memory when elements are popped
- Needs **less space** than the array, **if the array is relatively empty** (less than half)

## ▪ Main disadvantage:

- Needs **more space** (for the links) than the array, if the array is relatively full

## ▪ Other disadvantages:

- A bit **slower**
  - Still constant time but a bigger constant (create nodes, etc)

Note: Lack of random access is not a problem for a stack: its operations do not need this!

# Linked Queues

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

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

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

    @abstractmethod
    def serve(self) -> None:
        pass

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

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

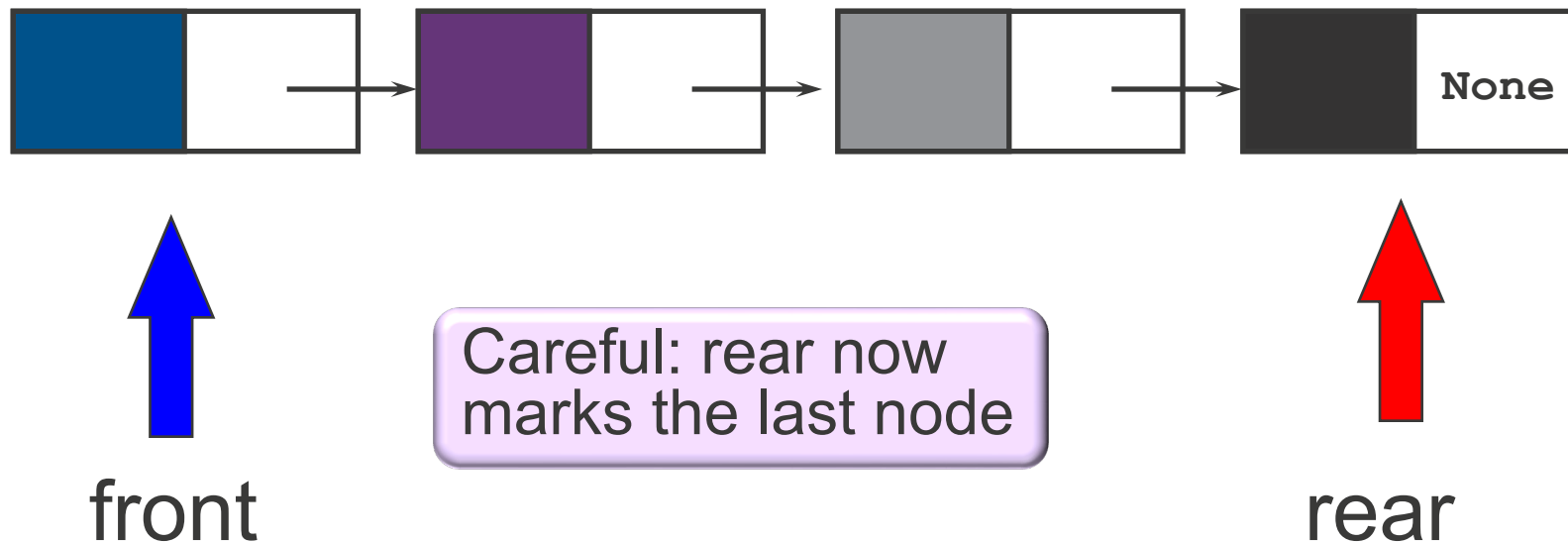
# Remember:

## Abstract base Queue class

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

```
@abstractmethod
def is_full(self) -> bool:
    pass
```

# Linked Queue



No need for circularity  
What do we need in  
the class?

# Class for Linked Queue

```
from typing import TypeVar
from abstract_queue import Queue
from node import Node
T = TypeVar('T')
```

```
class LinkQueue(Queue[T]):
    def __init__(self):
        Queue.__init__(self)
        self.front = None
        self.rear = None
```

The code must ensure that when **front** is **None**, **rear** is also **None**

```
def is_empty(self) -> bool:
    return self.front is None
```

```
def is_full(self) -> bool:
    return False
```

```
def clear(self) -> None:
    Queue.clear()
    self.front = None
    self.rear = None
```



# Linked Queues

## Append

# Append: algorithm

- **Linear array implementation:**

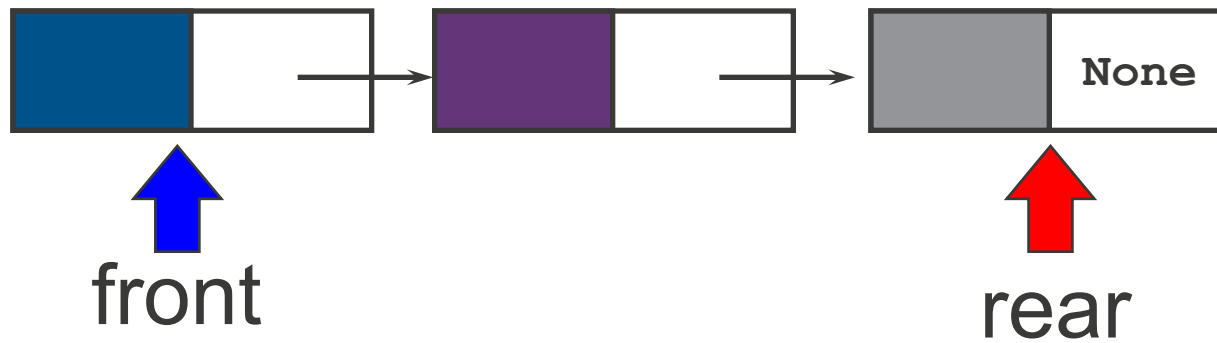
- If it is full: raise exception
- Else:
  - Increase rear
  - Add item at position marked by rear

- **In a linked list:**

- Create a new node that contains item and points to **None**
- Link the current rear to it
- Make the new node the new rear

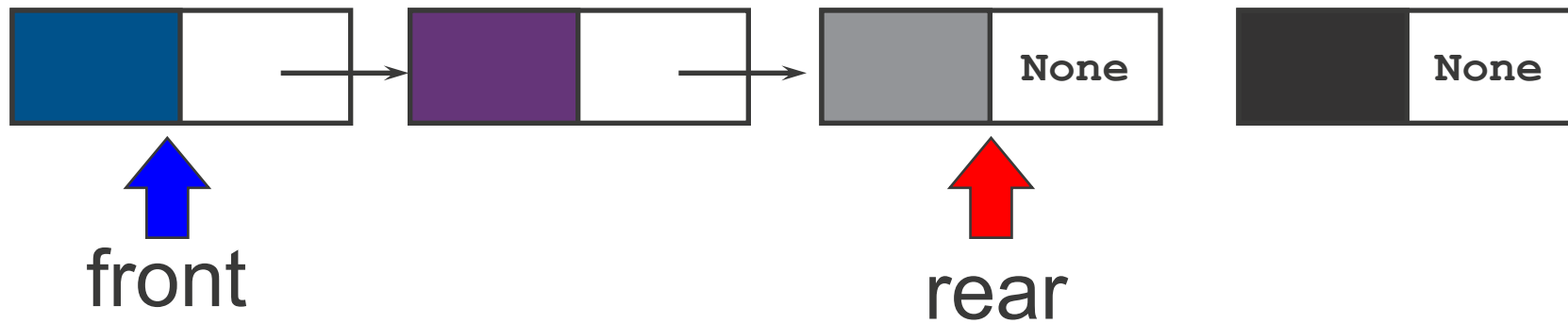
- **Again, no need for `is_full` check**

# Append: algorithm



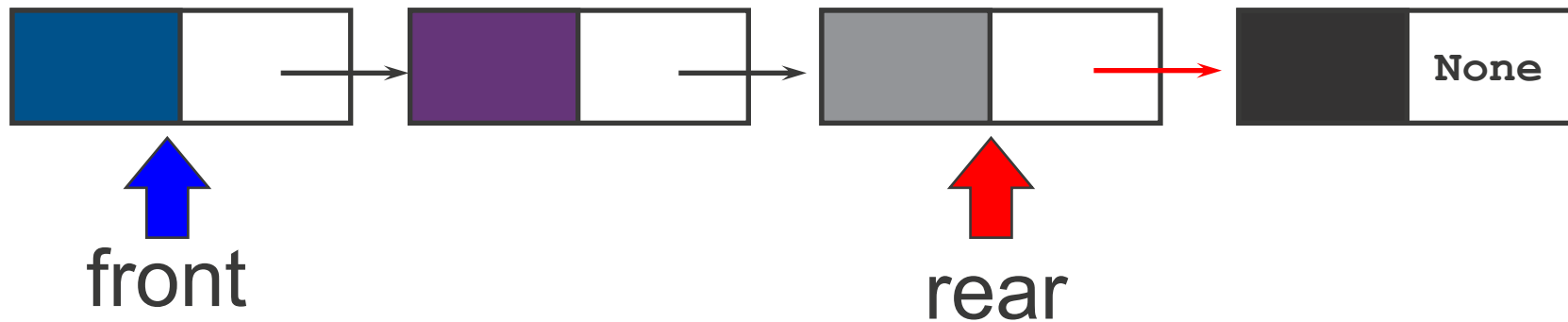
# Append: algorithm

- Create a new node for item



# Append: algorithm

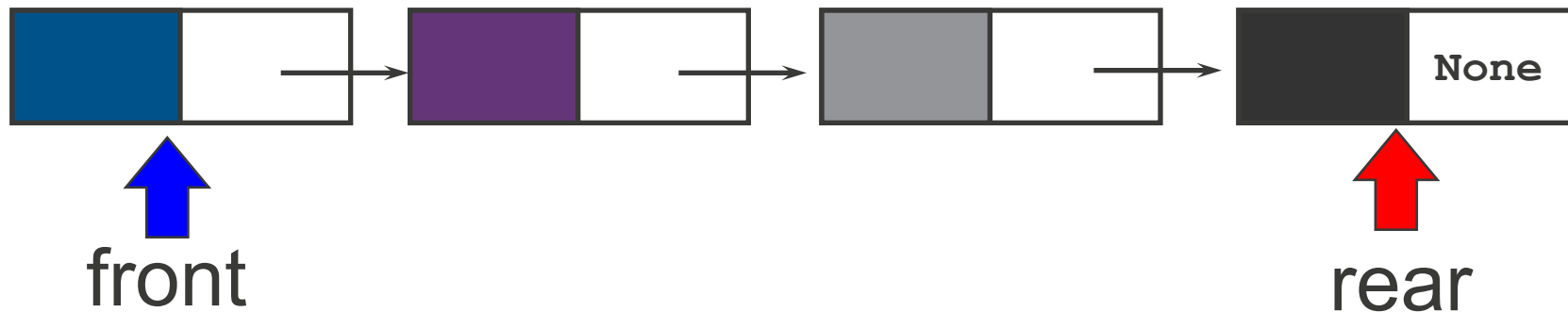
- Create a new node for item 
- Make a link from the current rear to the new node



# Append: algorithm

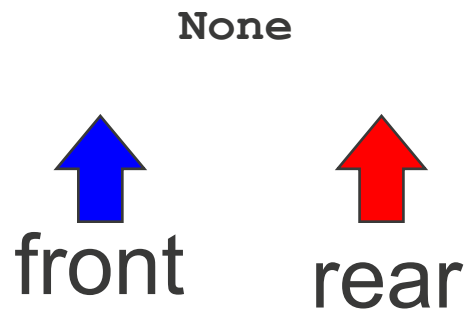
- Create a new node for item 
- Make a link from the current rear to the new node
- **The new node becomes the new rear**

Does this general algorithm always work?



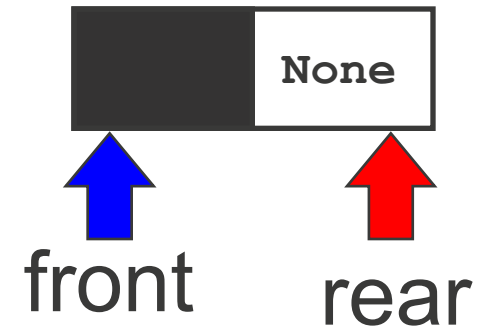
# Append: algorithm

- No, if the queue is empty, we must modify front too
- How?
  - Create a new node for item



# Append: algorithm

- No, if the queue is empty, we must modify front too
- How?
  - Create a new node for item 
- The new node become the new front and rear



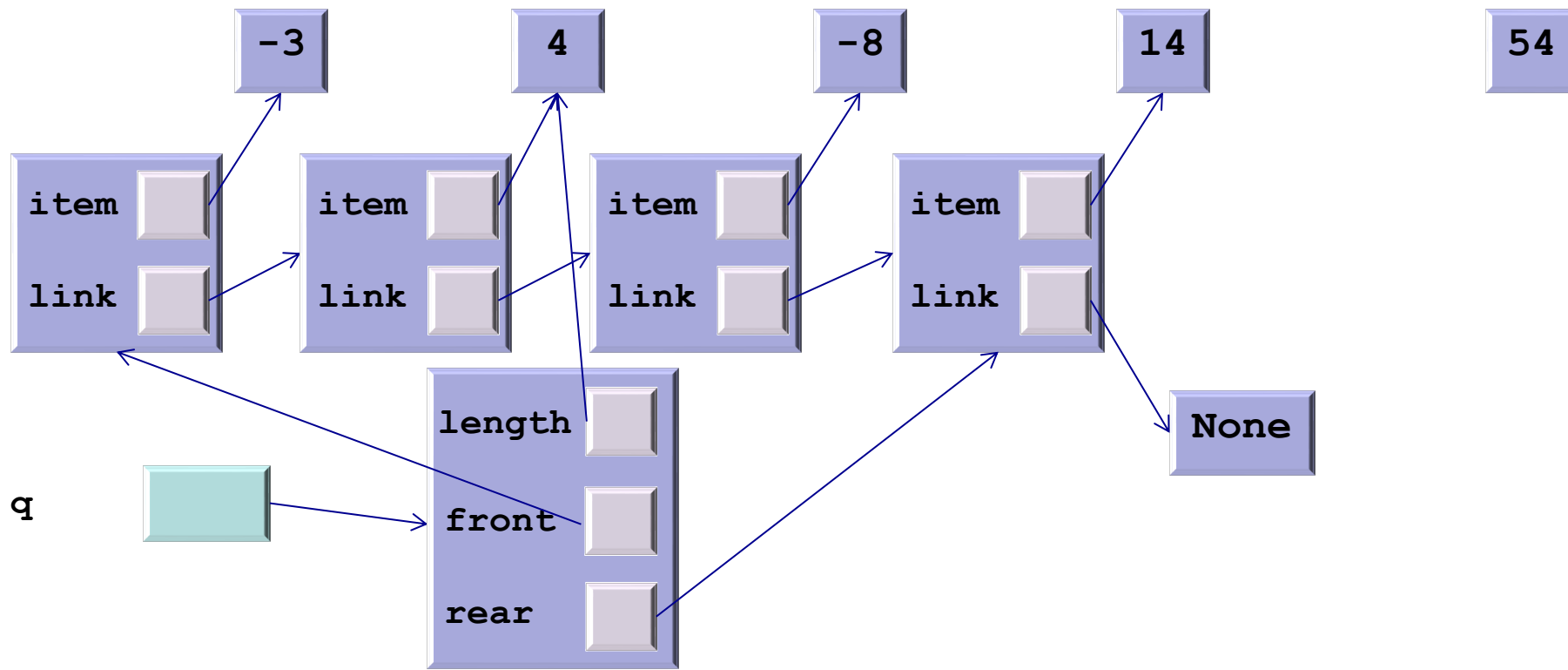


# Append method

...

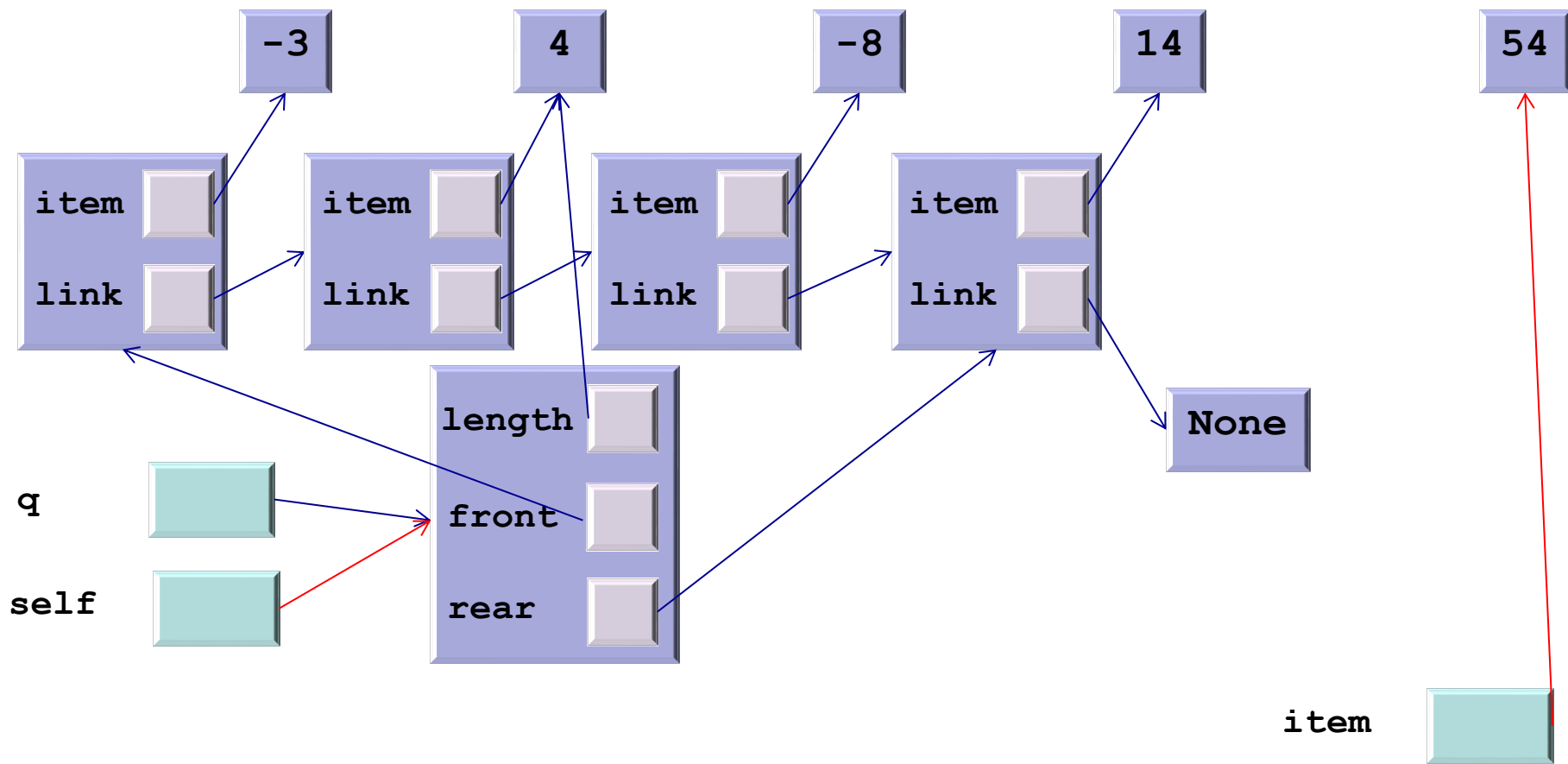
```
def append(self, item: T) -> None:
    new_node = Node(item) # create new node
    if self.is_empty():
        self.front = new_node # move head
    else:
        self.rear.link = new_node #link it in
    self.rear = new_node # move rear to new node
    self.length += 1
```

**Complexity?  $O(1)$**



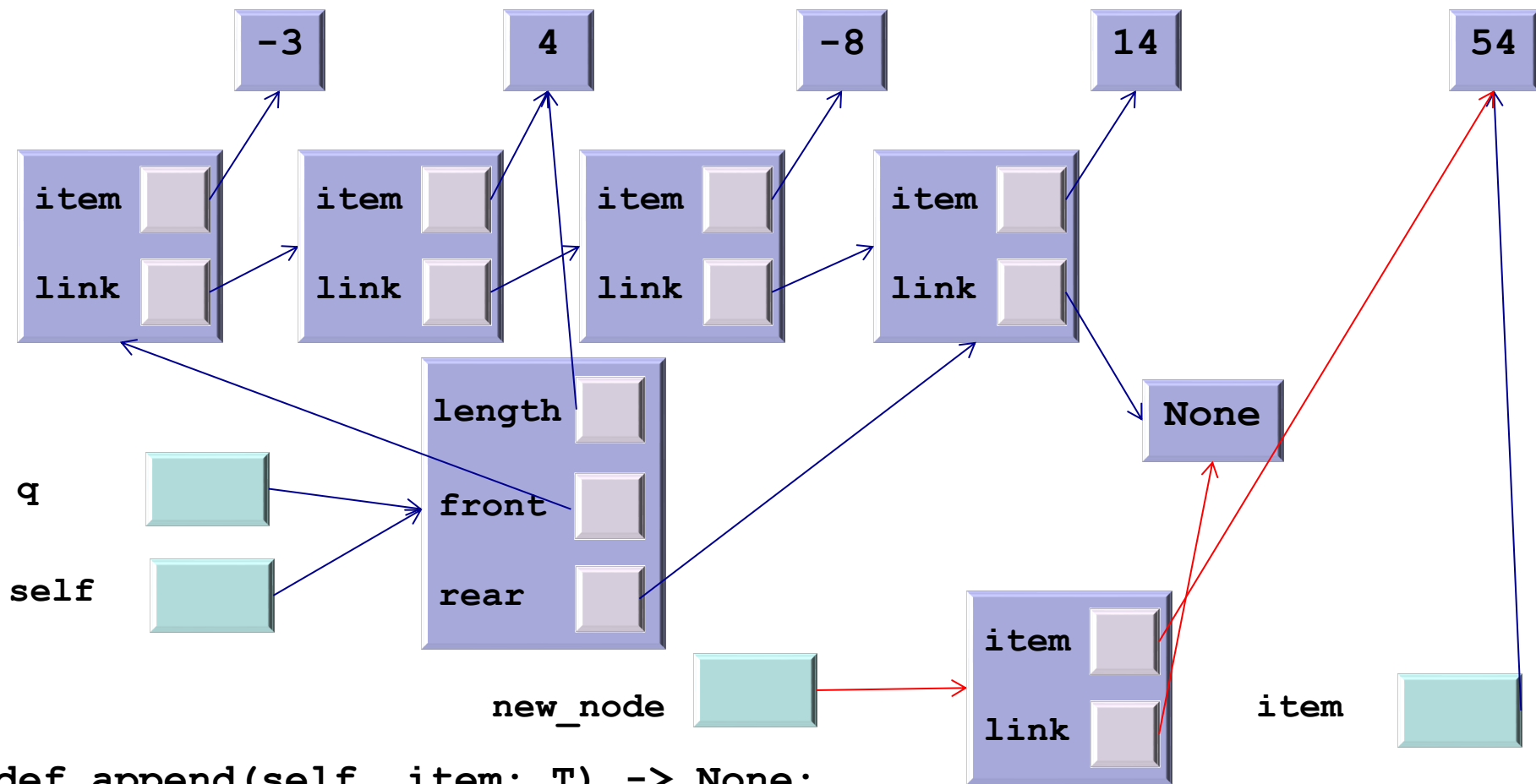
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



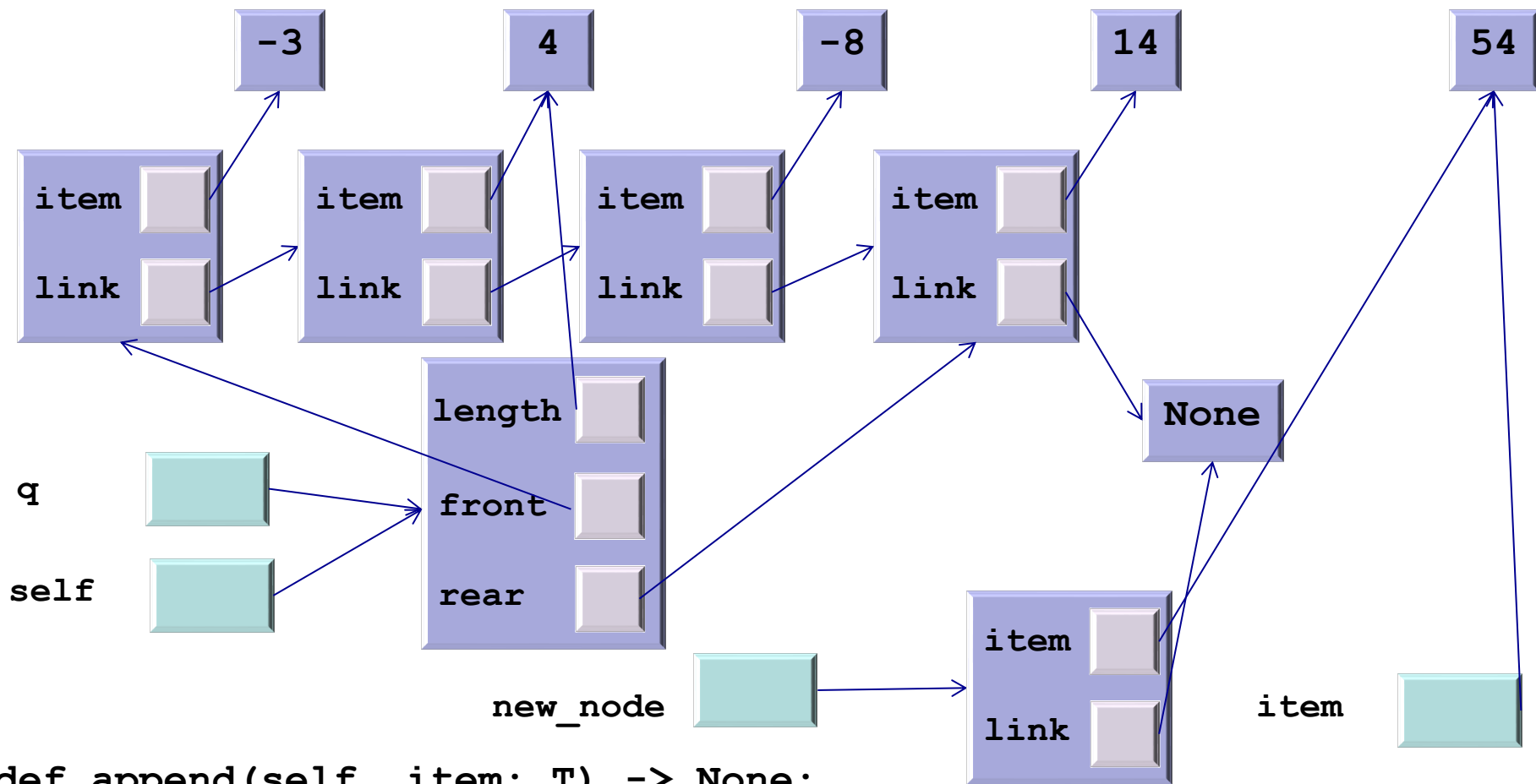
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

`q.append(54)`



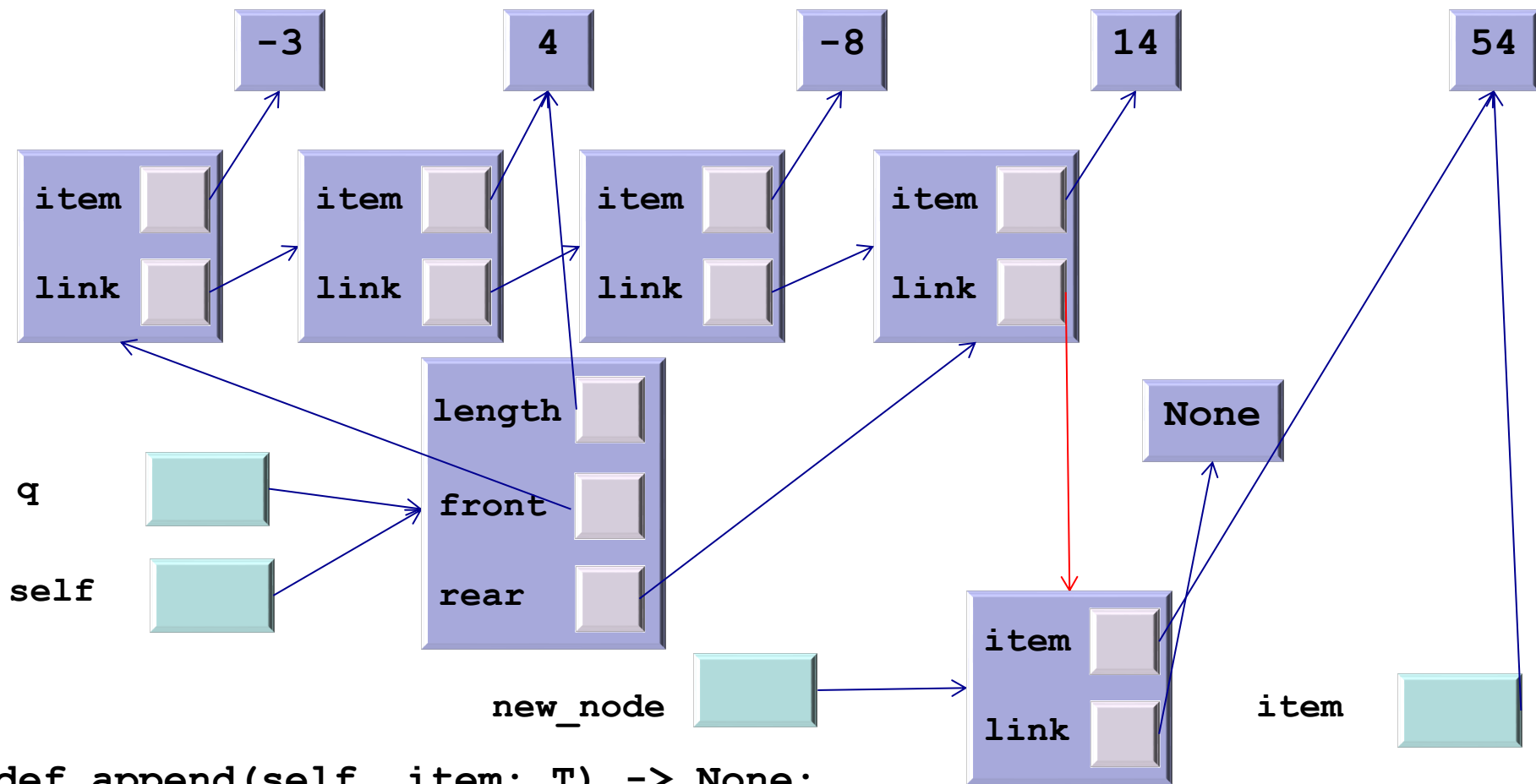
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



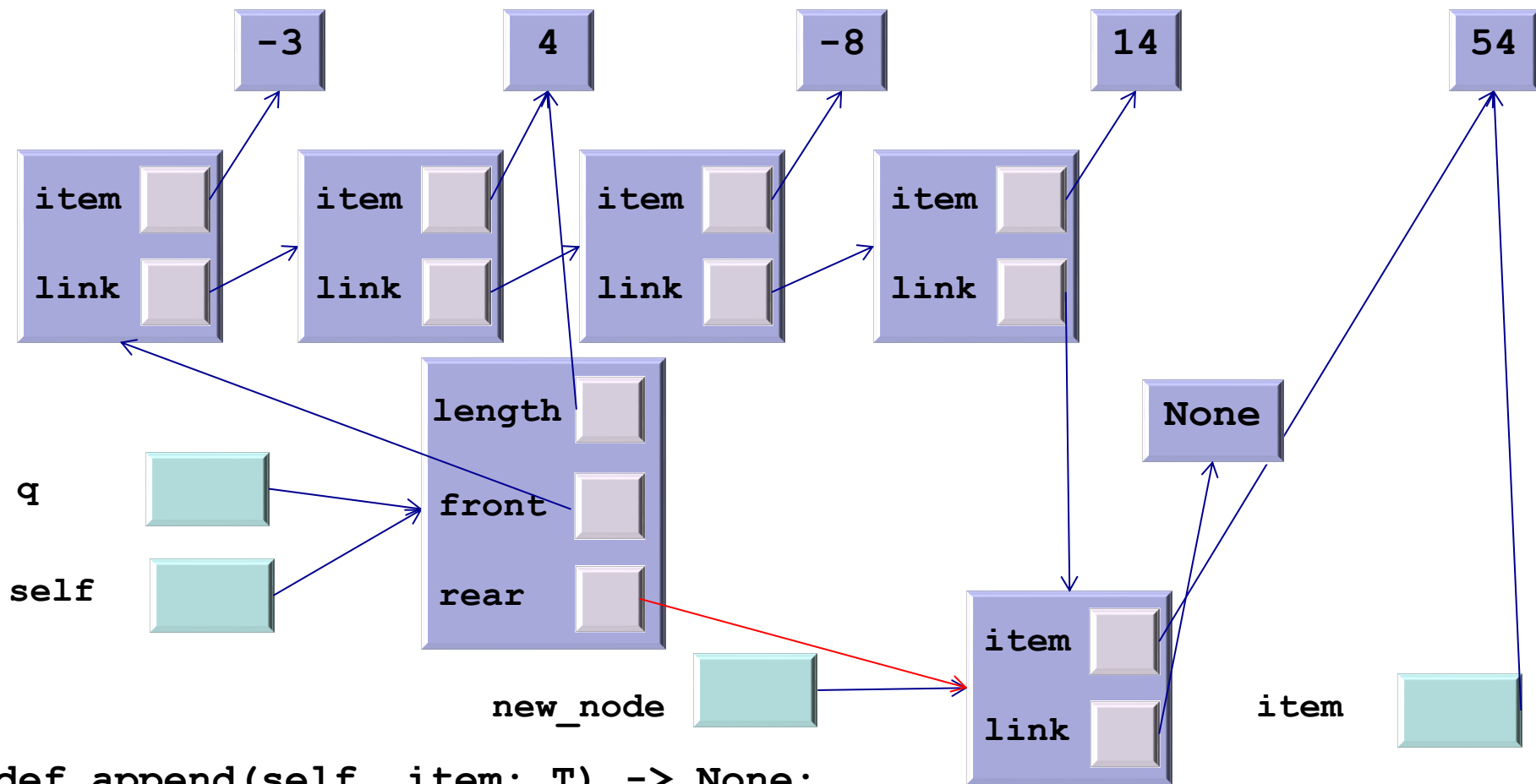
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



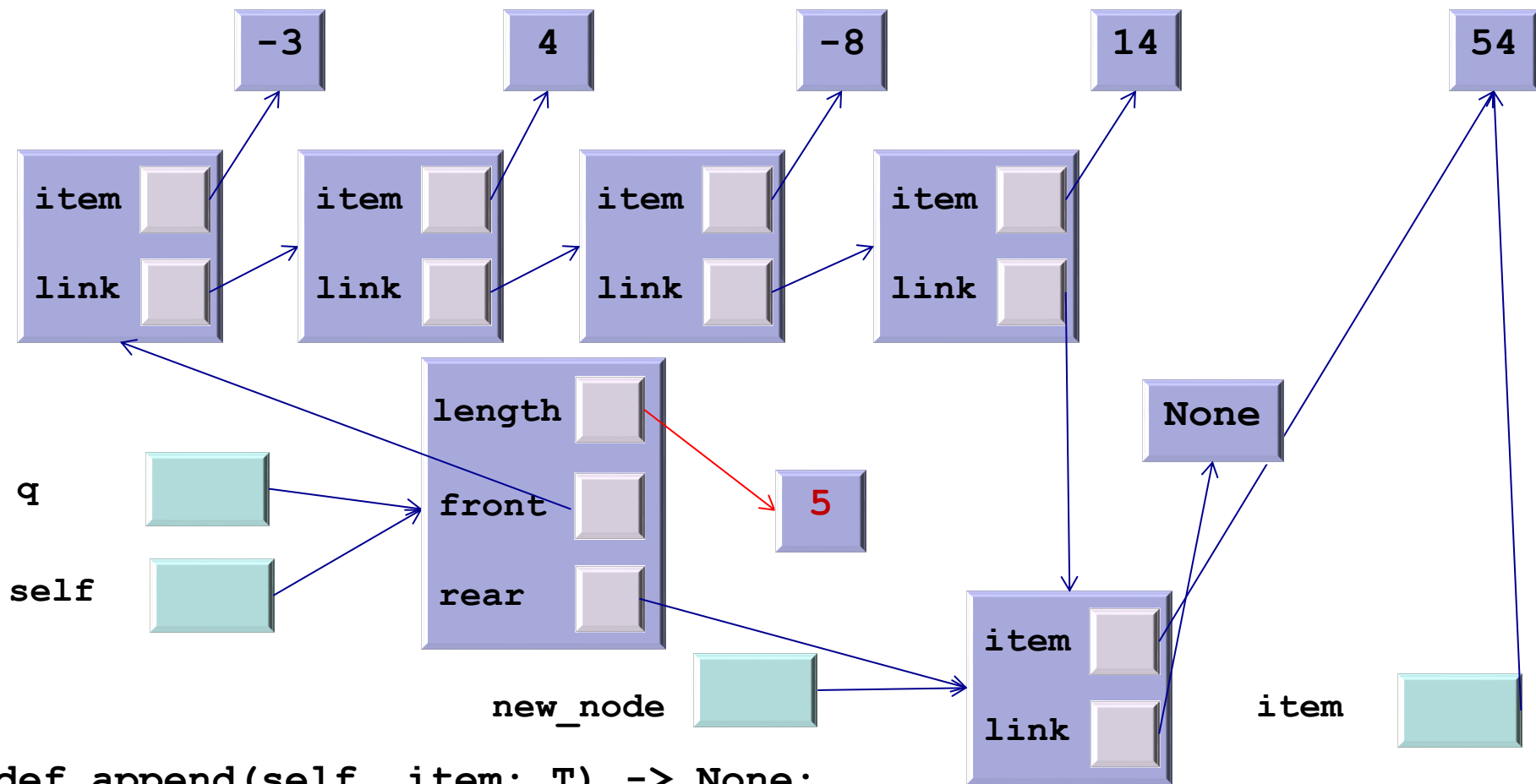
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

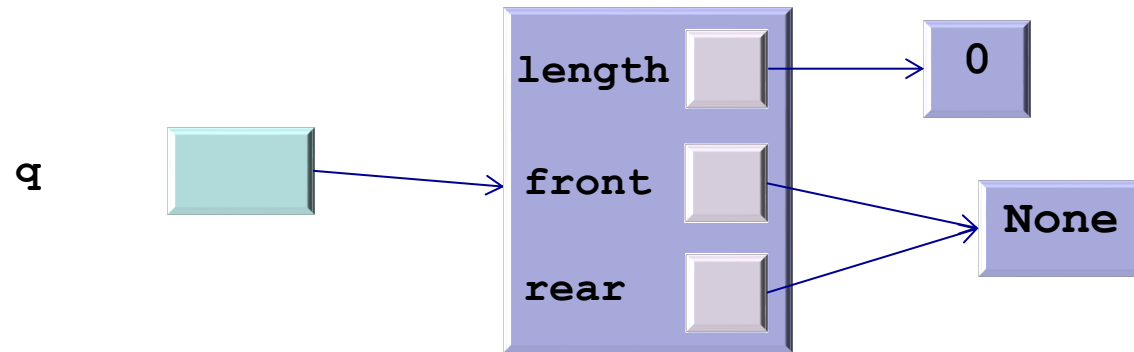
`q.append(54)`



```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

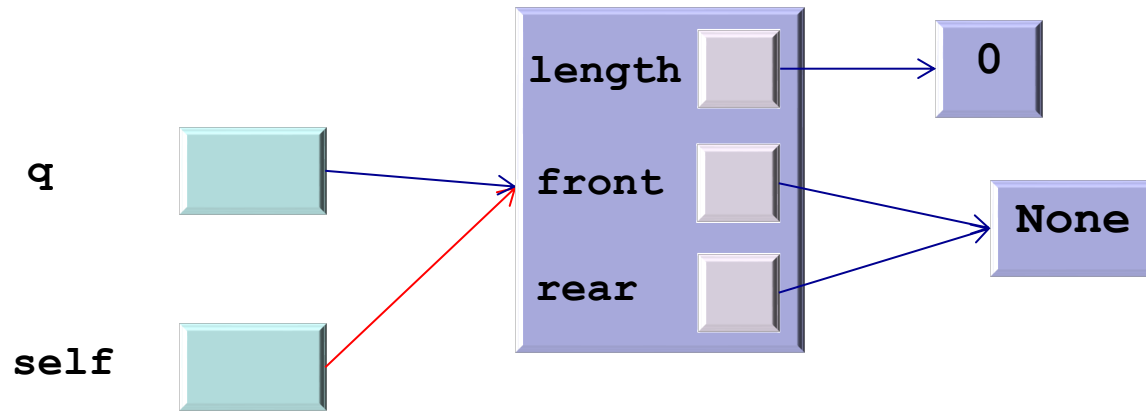
q.append(54)





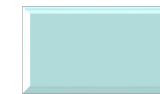
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



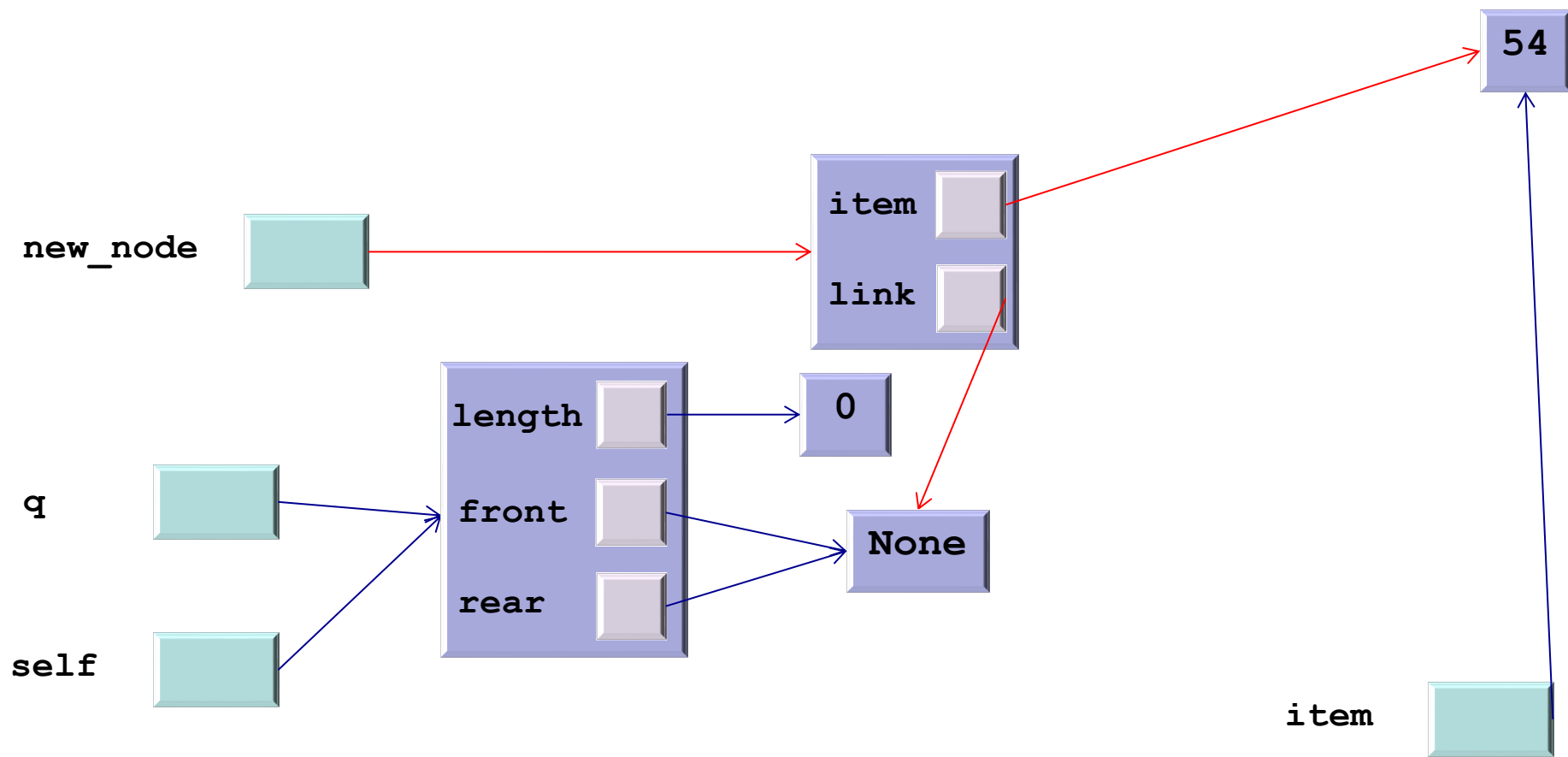
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

item



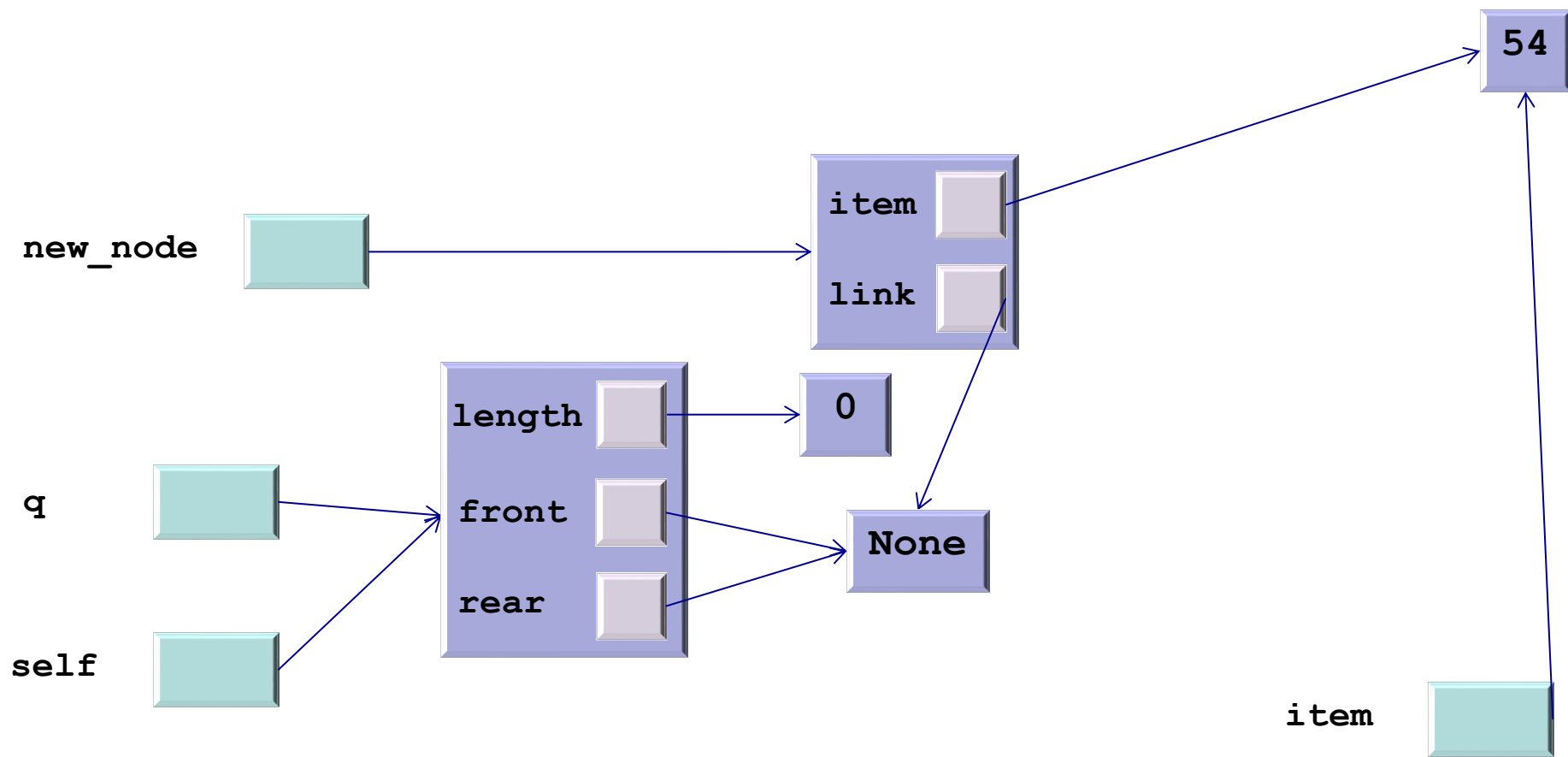
54

`q.append(54)`



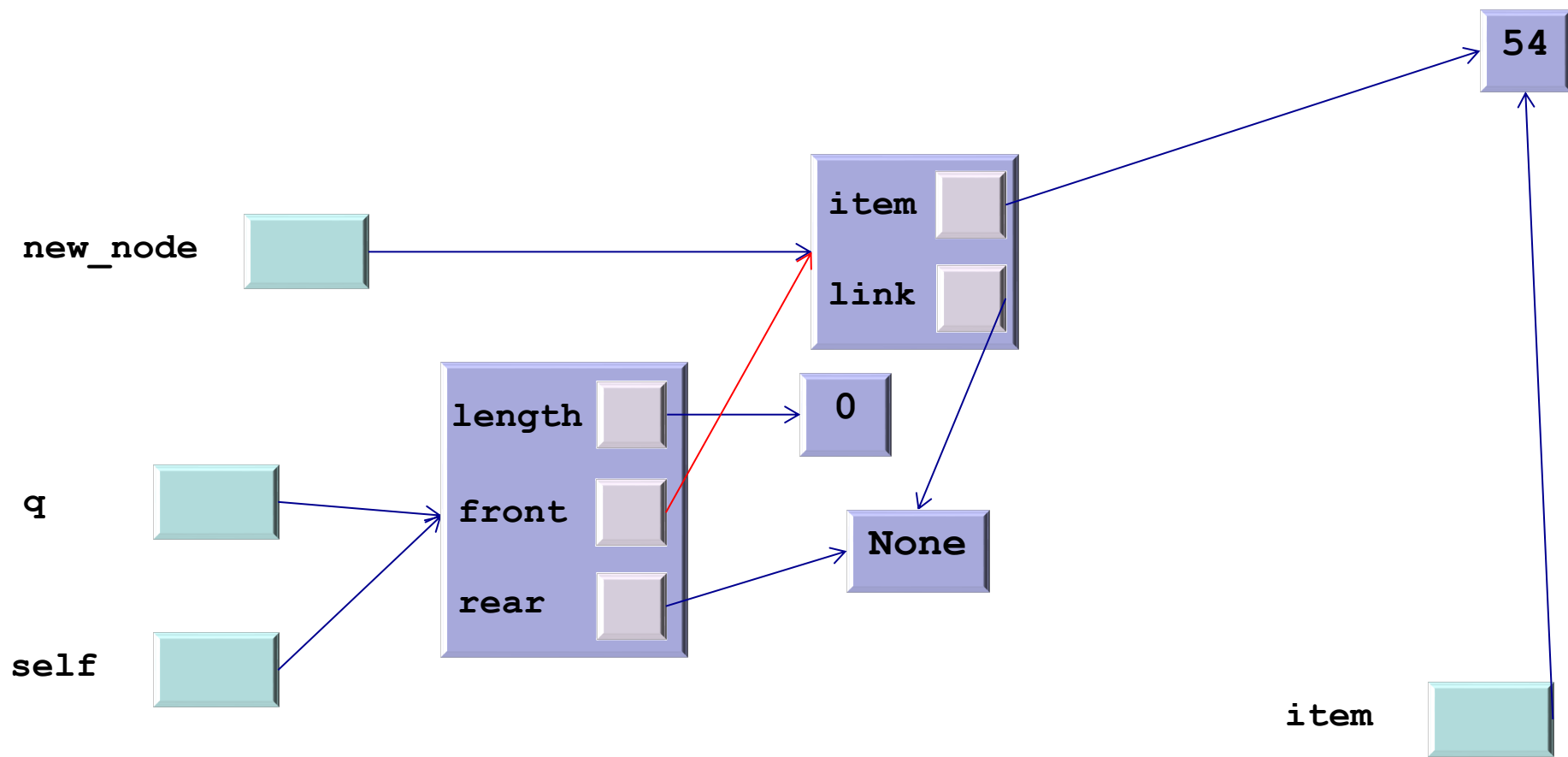
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



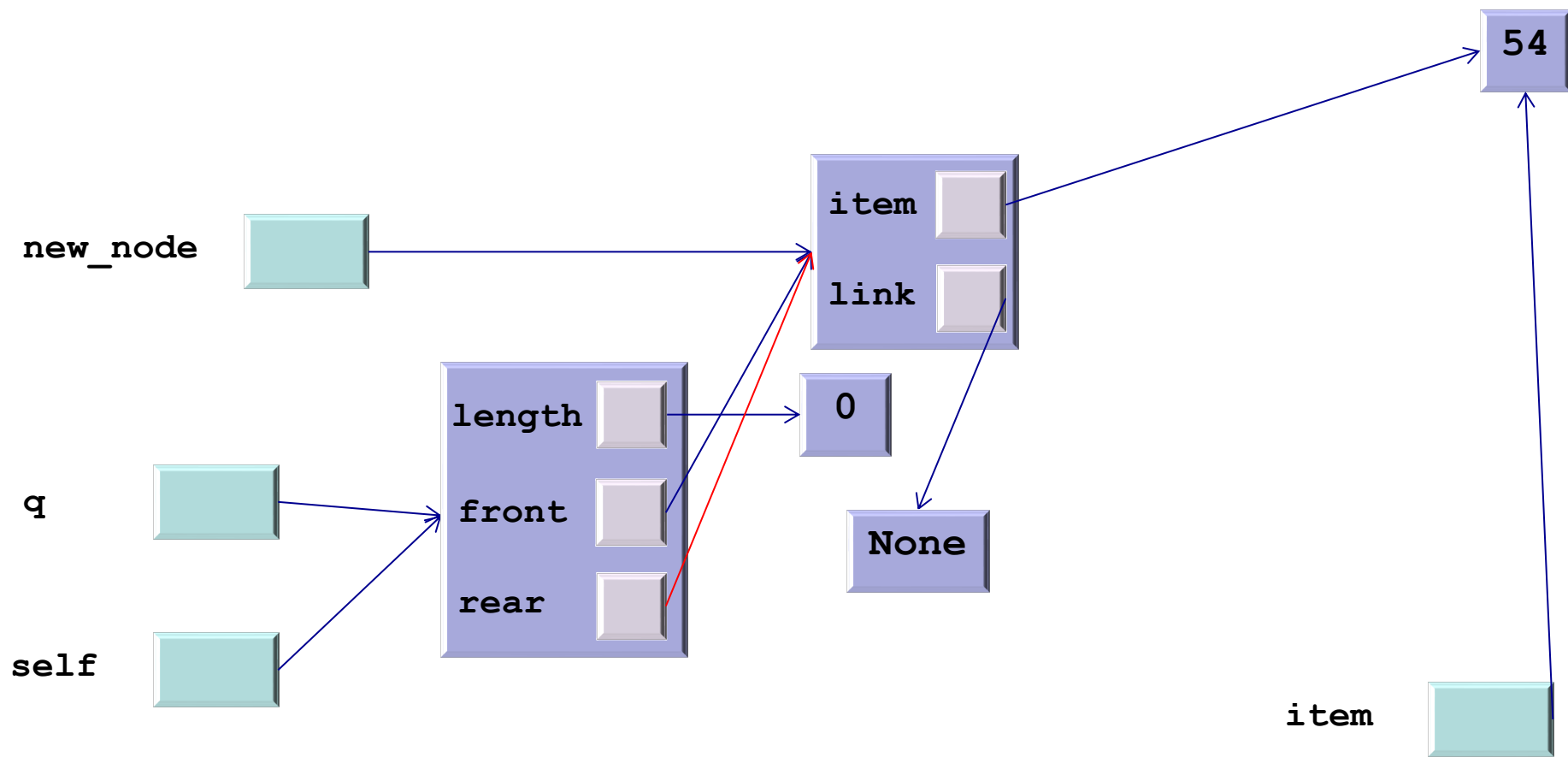
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

`q.append(54)`



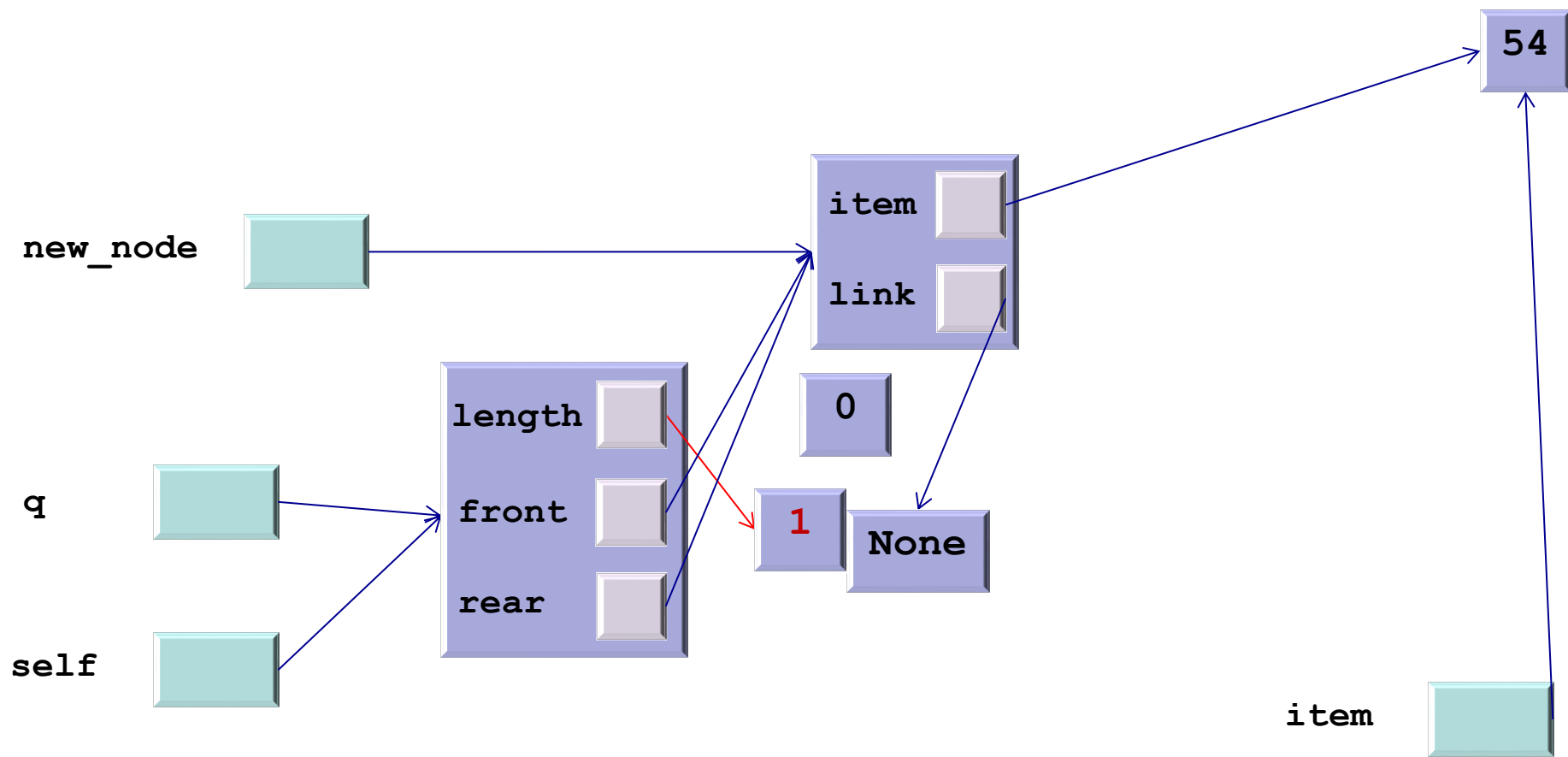
```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)



```
def append(self, item: T) -> None:
    new_node = Node(item)
    if self.is_empty():
        self.front = new_node
    else:
        self.rear.link = new_node
    self.rear = new_node
    self.length += 1
```

q.append(54)

# Linked Queues Serve



# Serve: algorithm

- **Linear array implementation:**

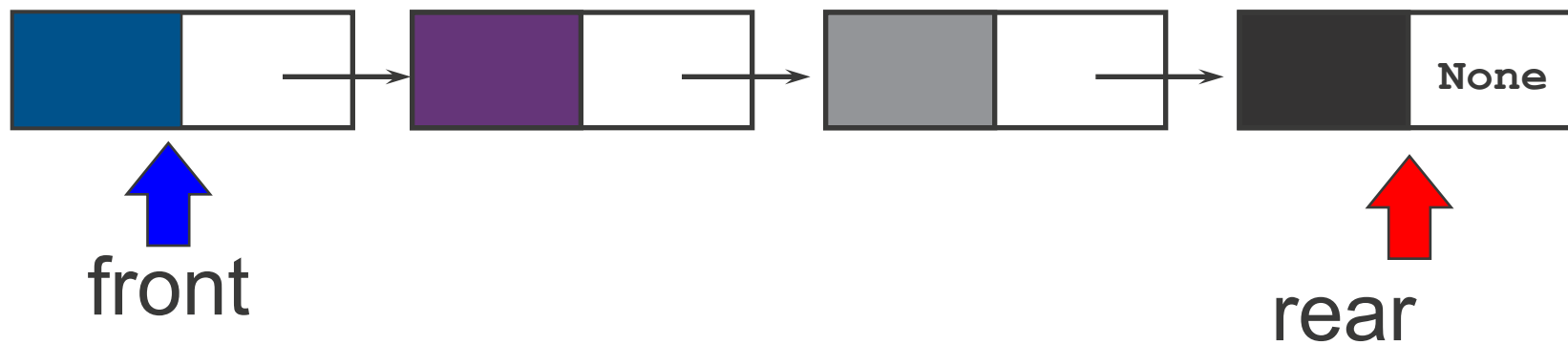
- If it is empty: raise exception
- Else:
  - Remember item to return
  - Increase front
  - Return item

- **In a linked list**

- Almost identical
- We simply move front along rather than increase it

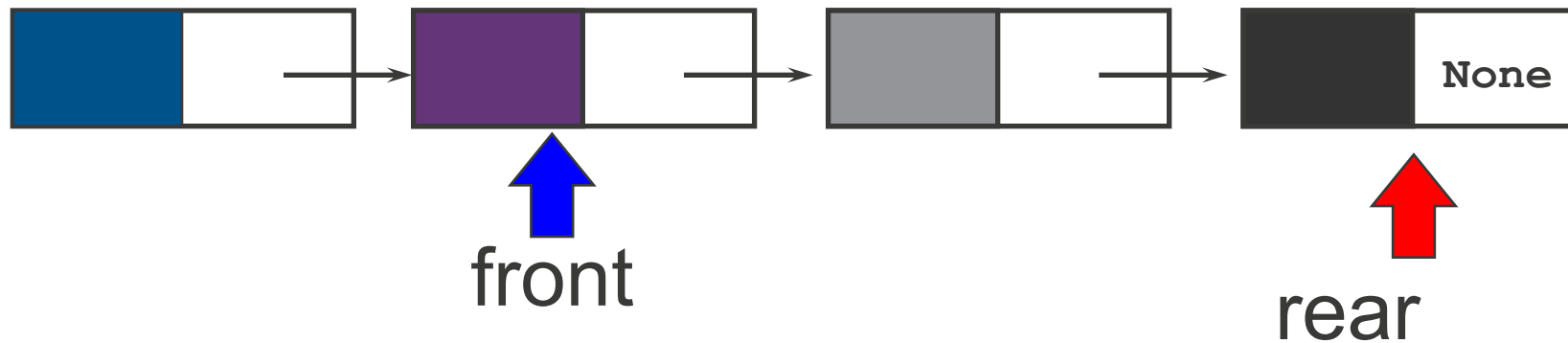
# Serve: algorithm

- Remember the item in the front node



# Serve: algorithm

- Remember the item in the front node
- Make the next node the new front

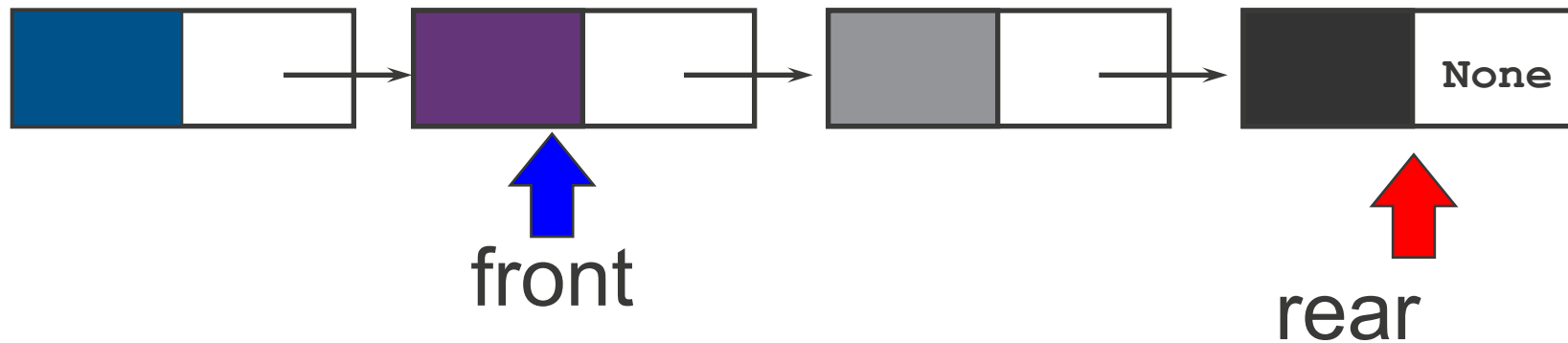


# Serve: algorithm

- Remember the item in the front node
- Make the next node the new front
- **Return the item**

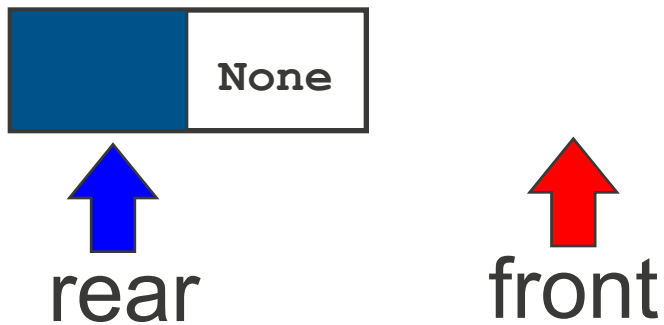


Does this general algorithm always work?



# Serve: algorithm

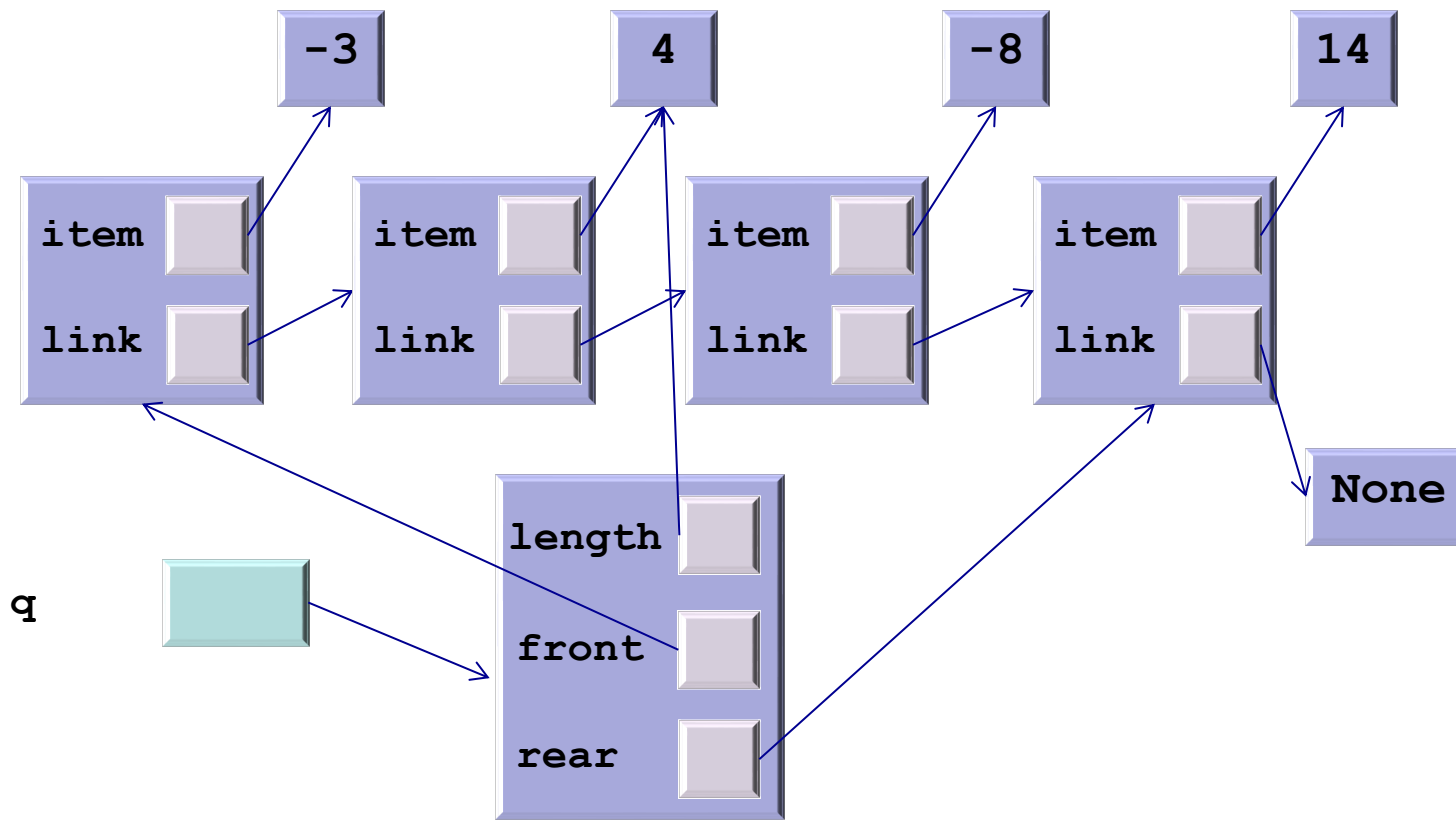
- If the Queue becomes empty, we end up in a possibly dangerous configuration
- Better to set rear to `None`



# Serve method

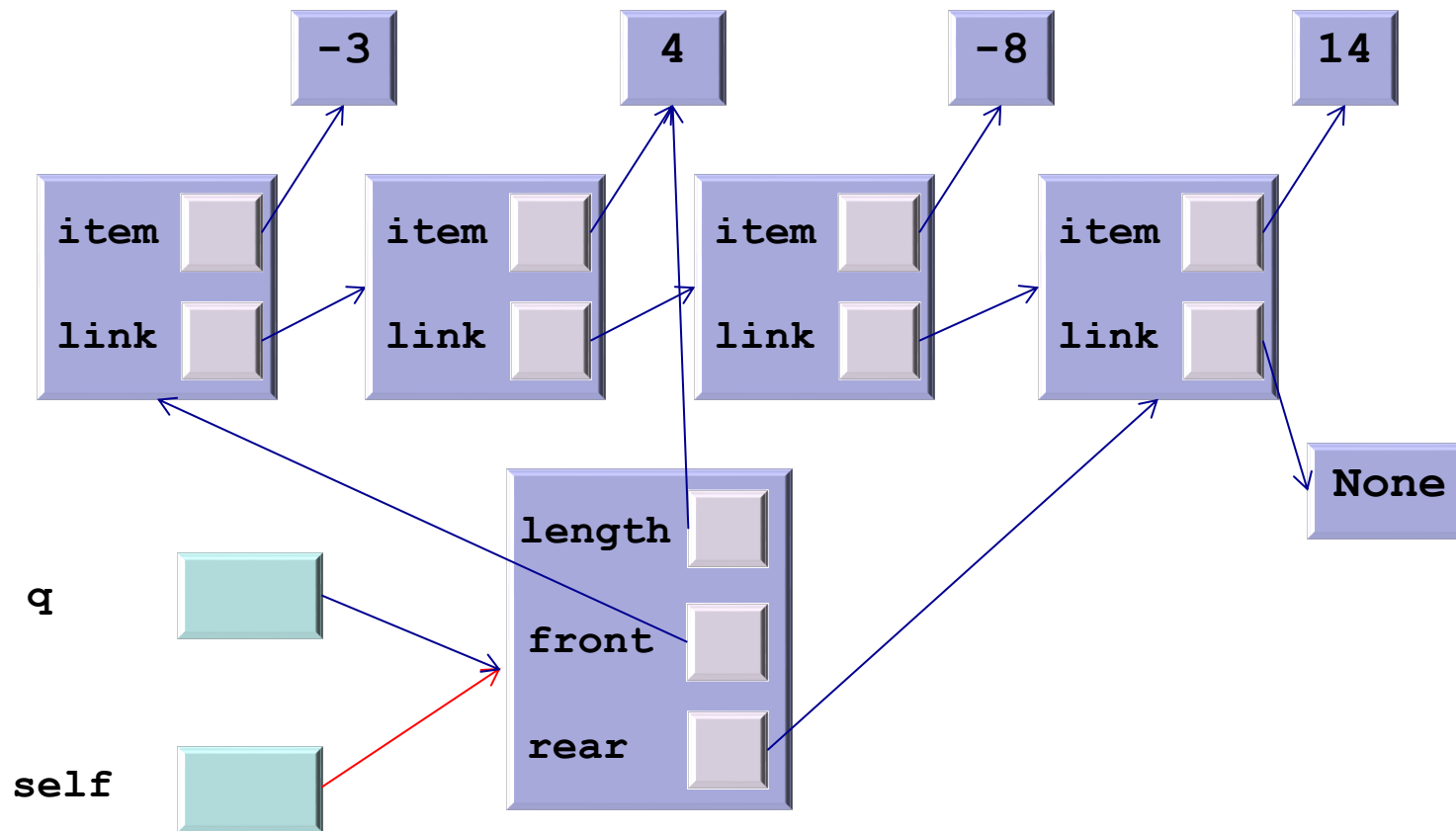
```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```

**Complexity?  $O(1)$**



`q.serve()`

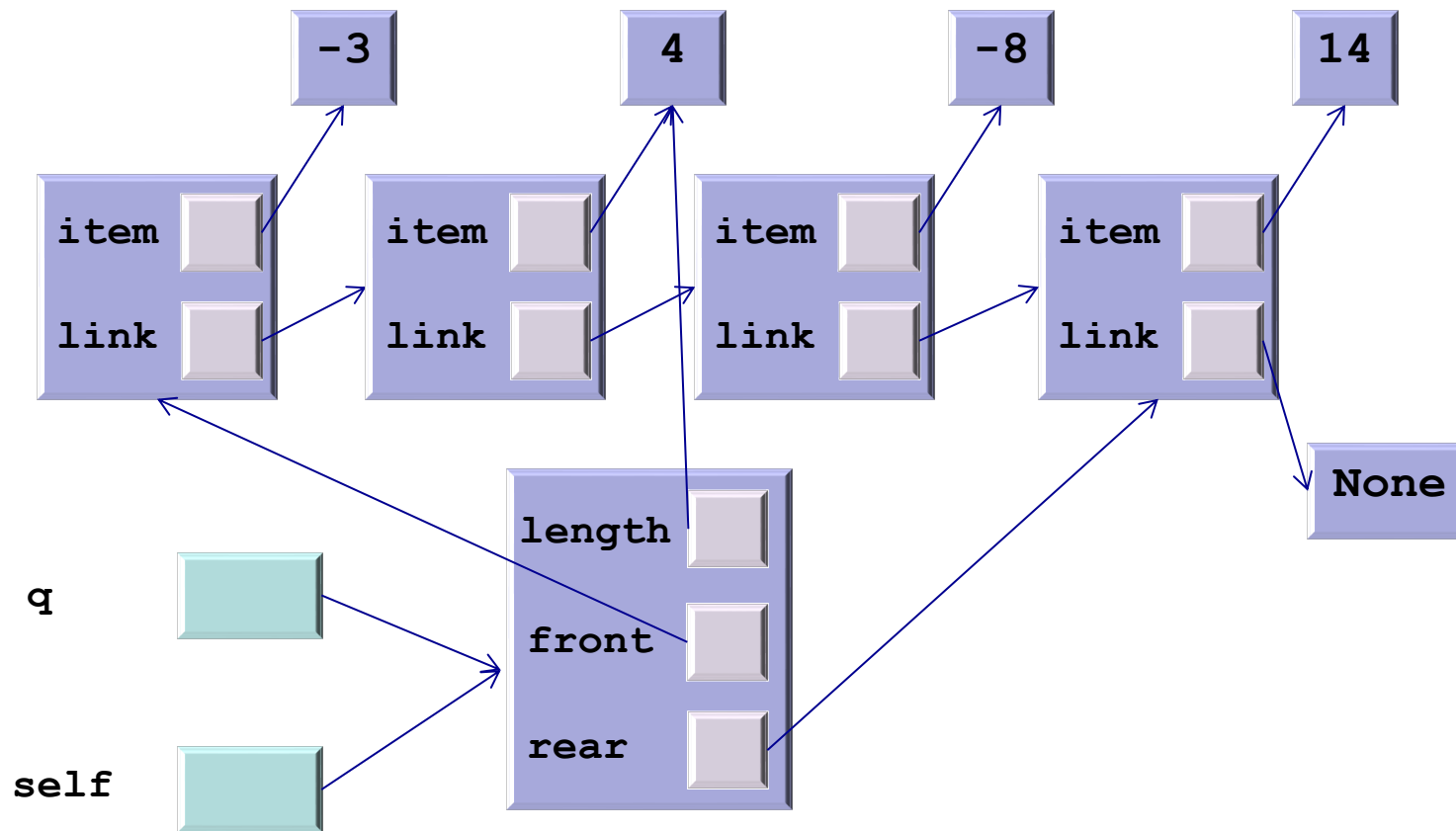
```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



**q. serve()**

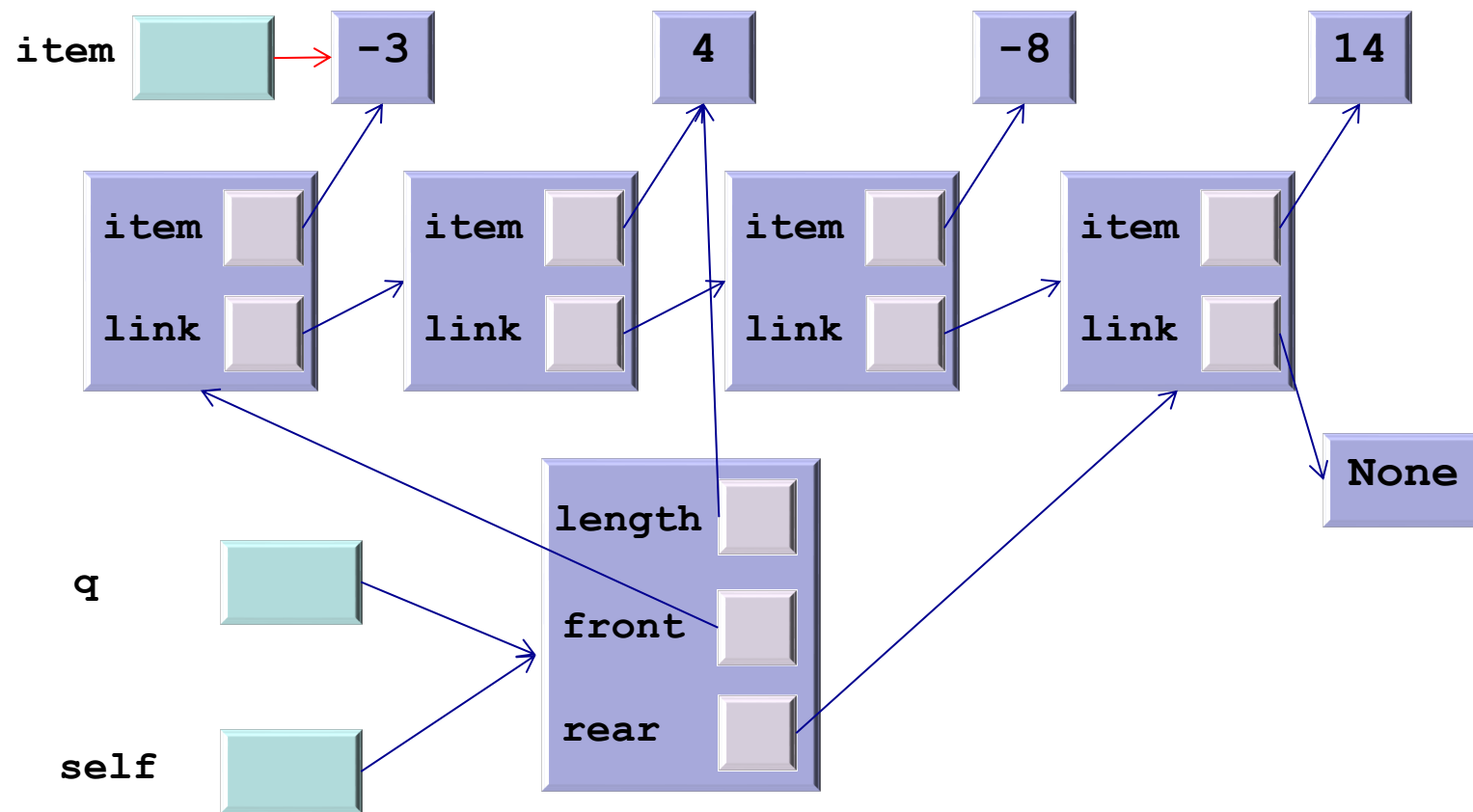
```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```





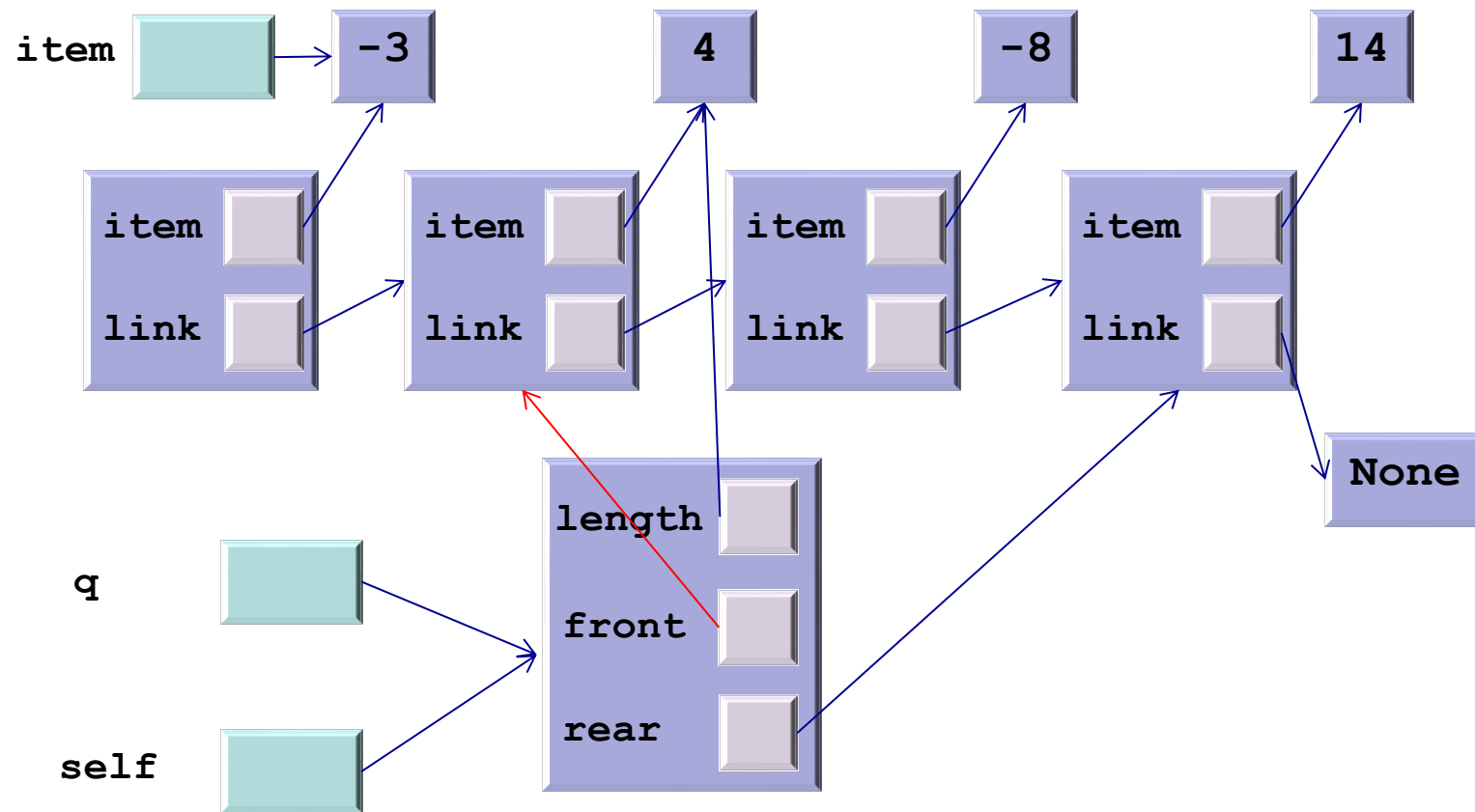
**q. serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



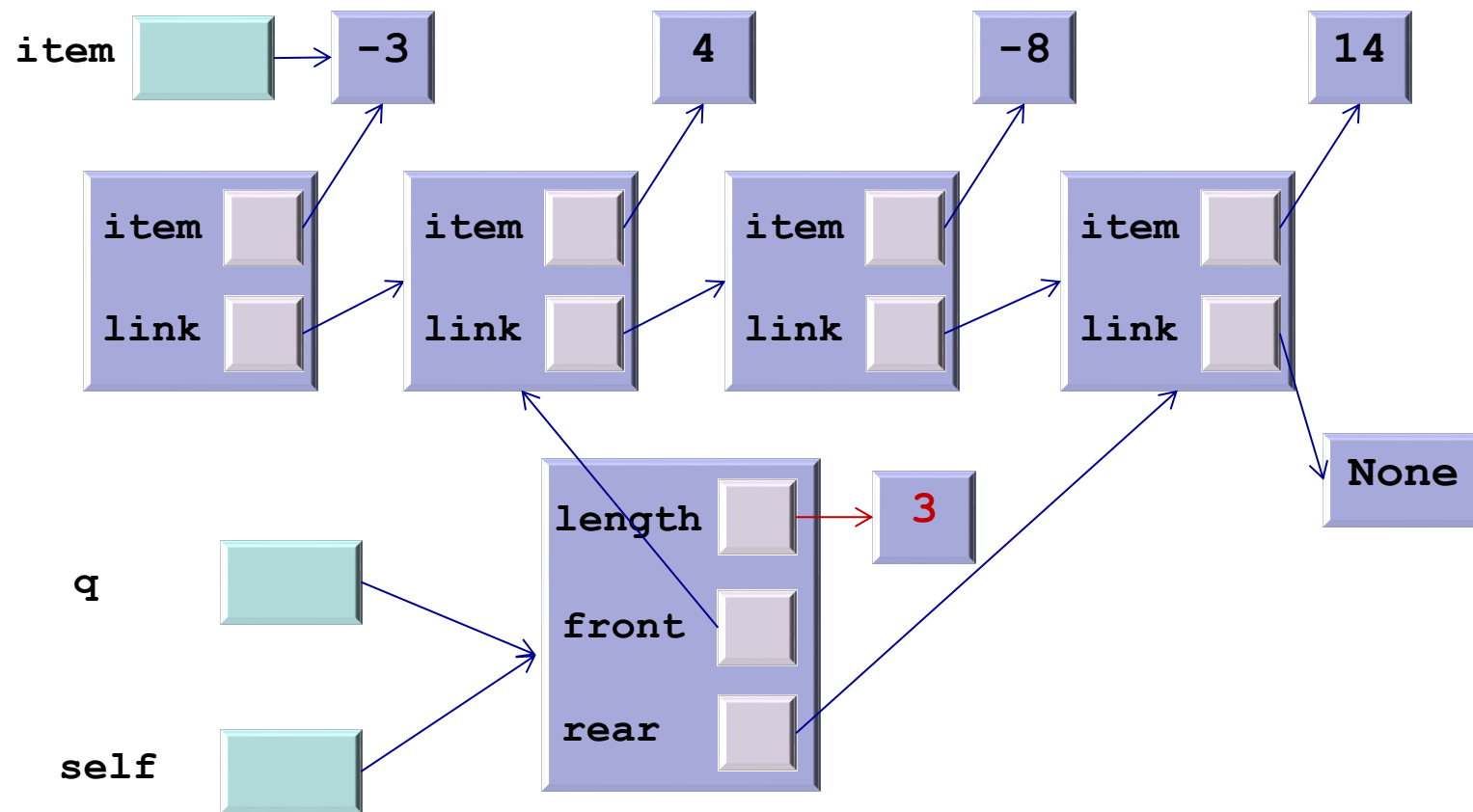
**q.serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



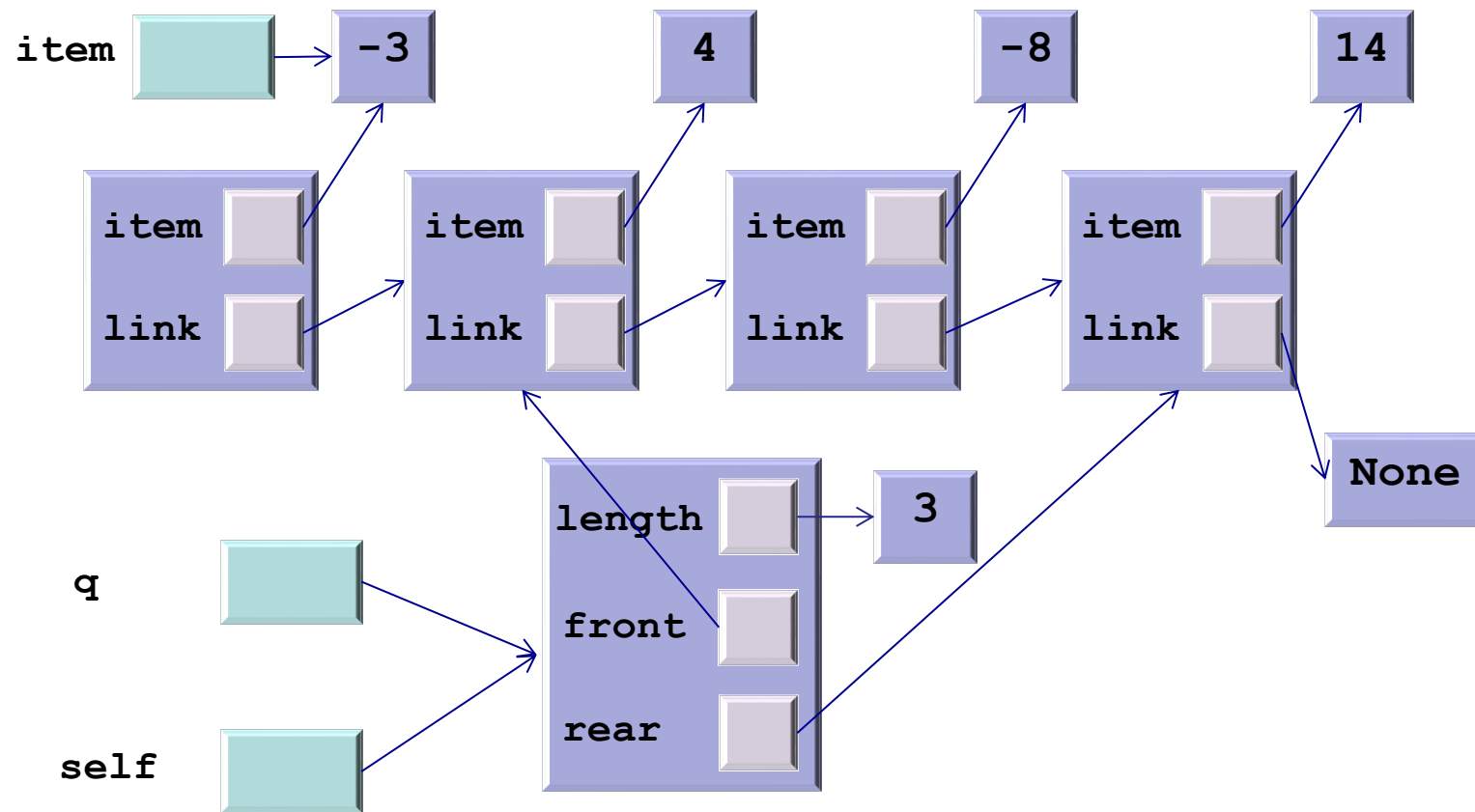
**q. serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



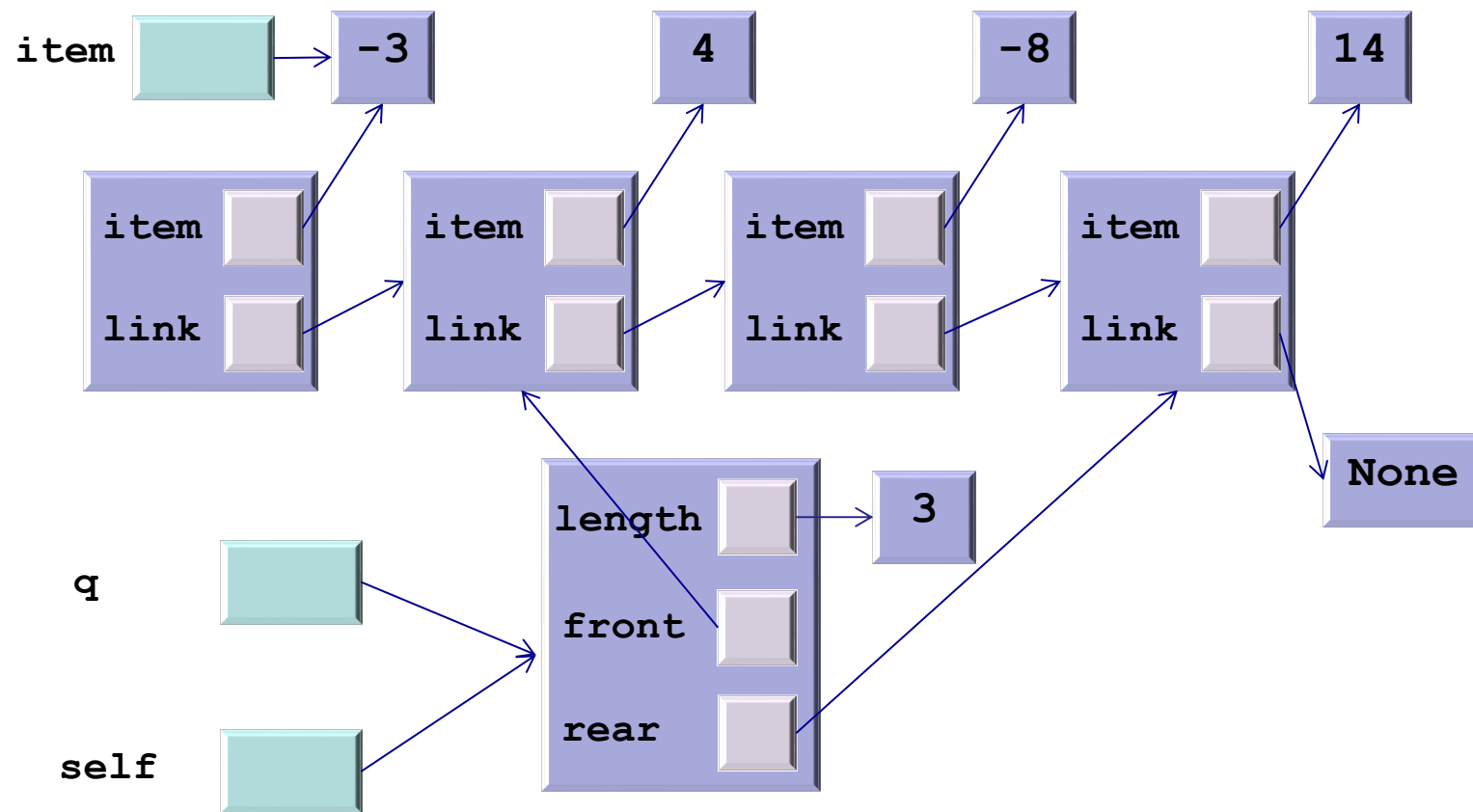
**q. serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



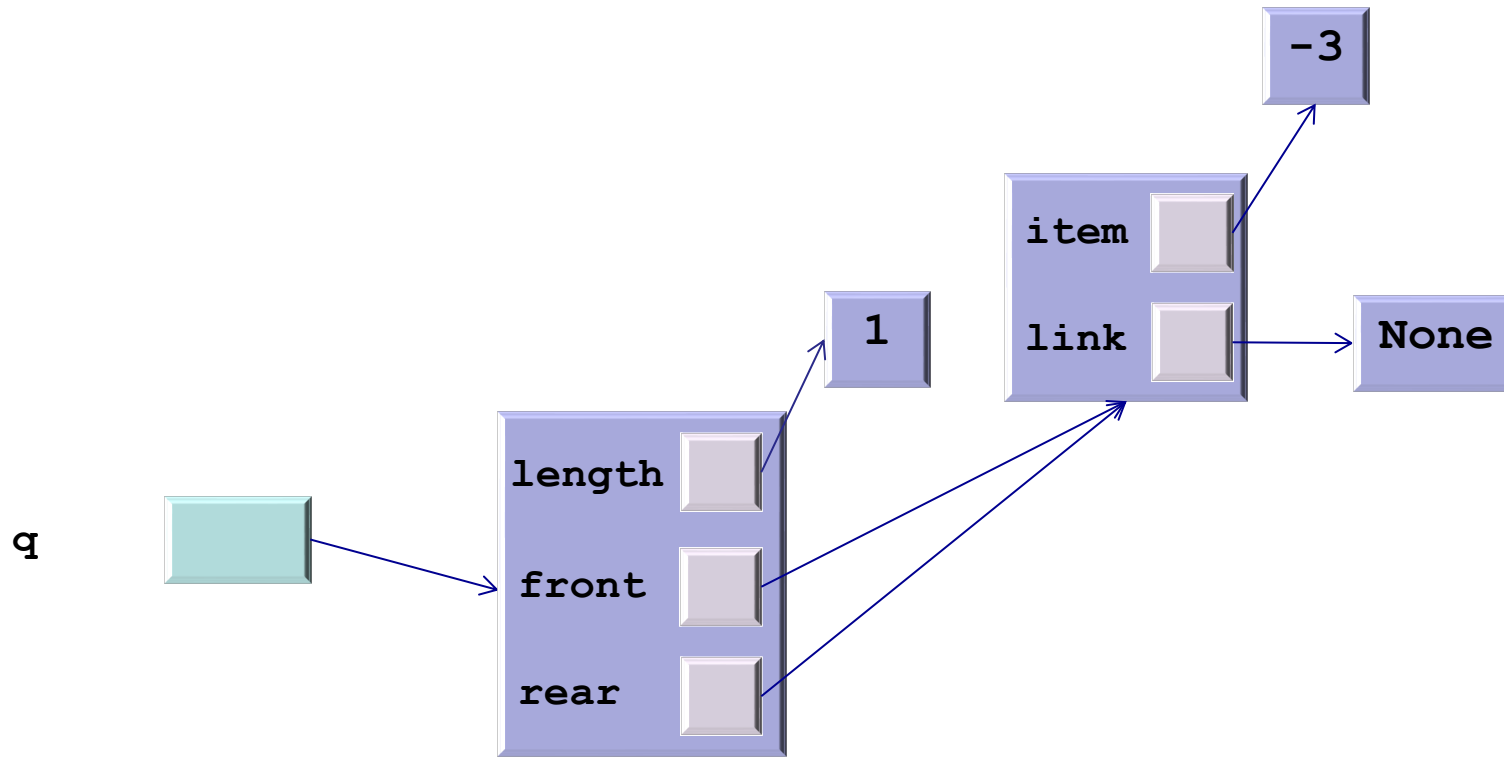
**q.serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



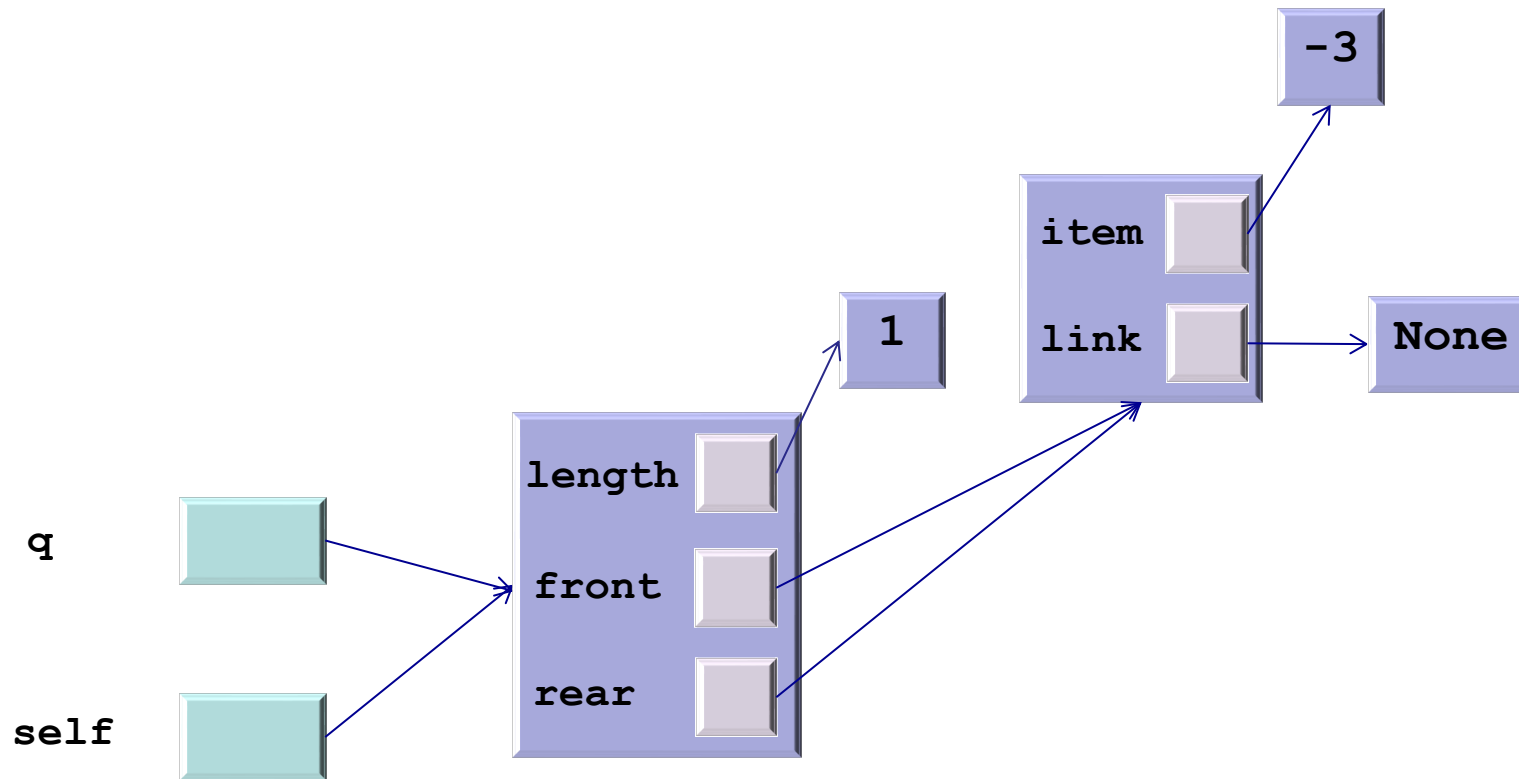
**q.serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



**q.serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```

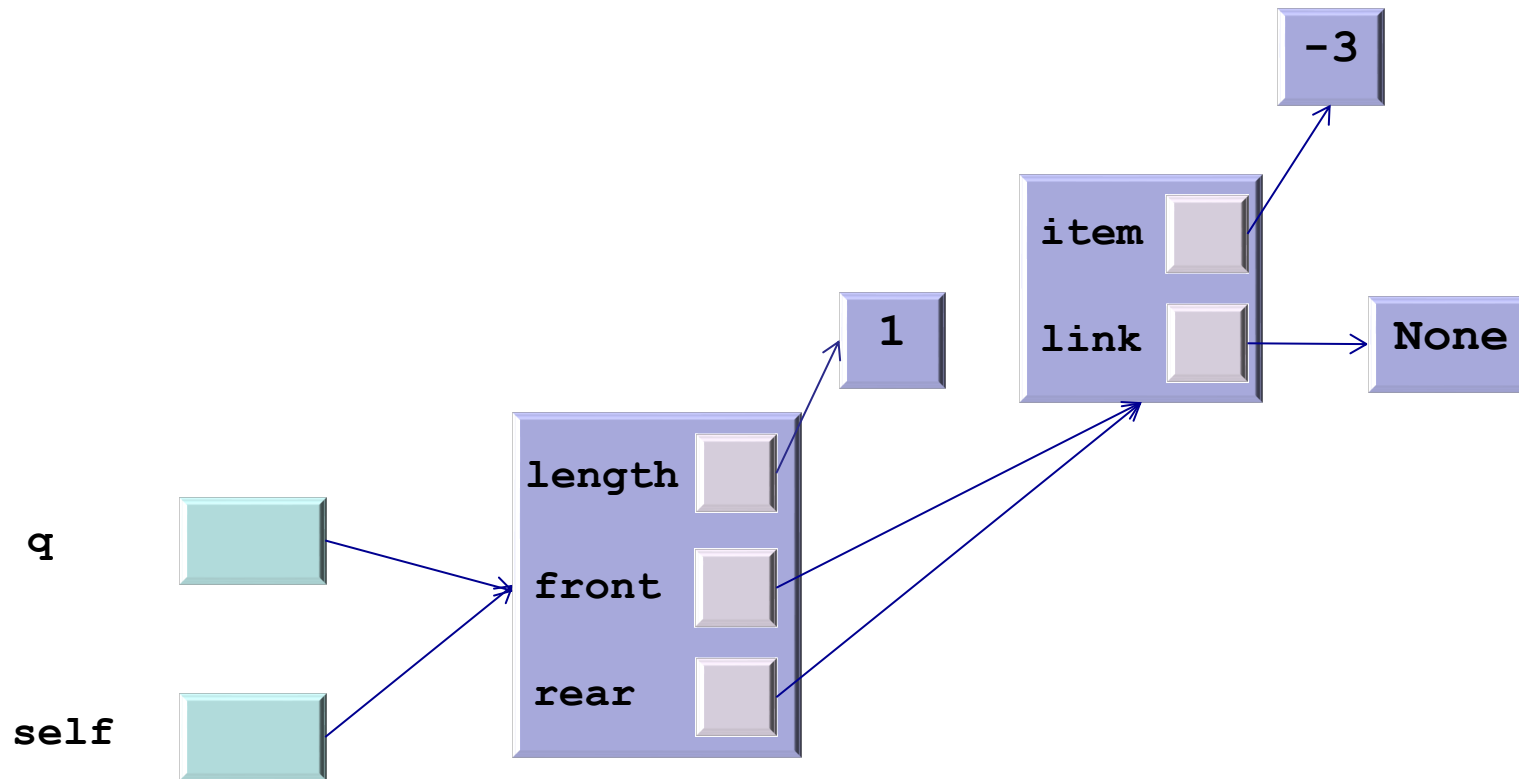


**q.serve()**

```

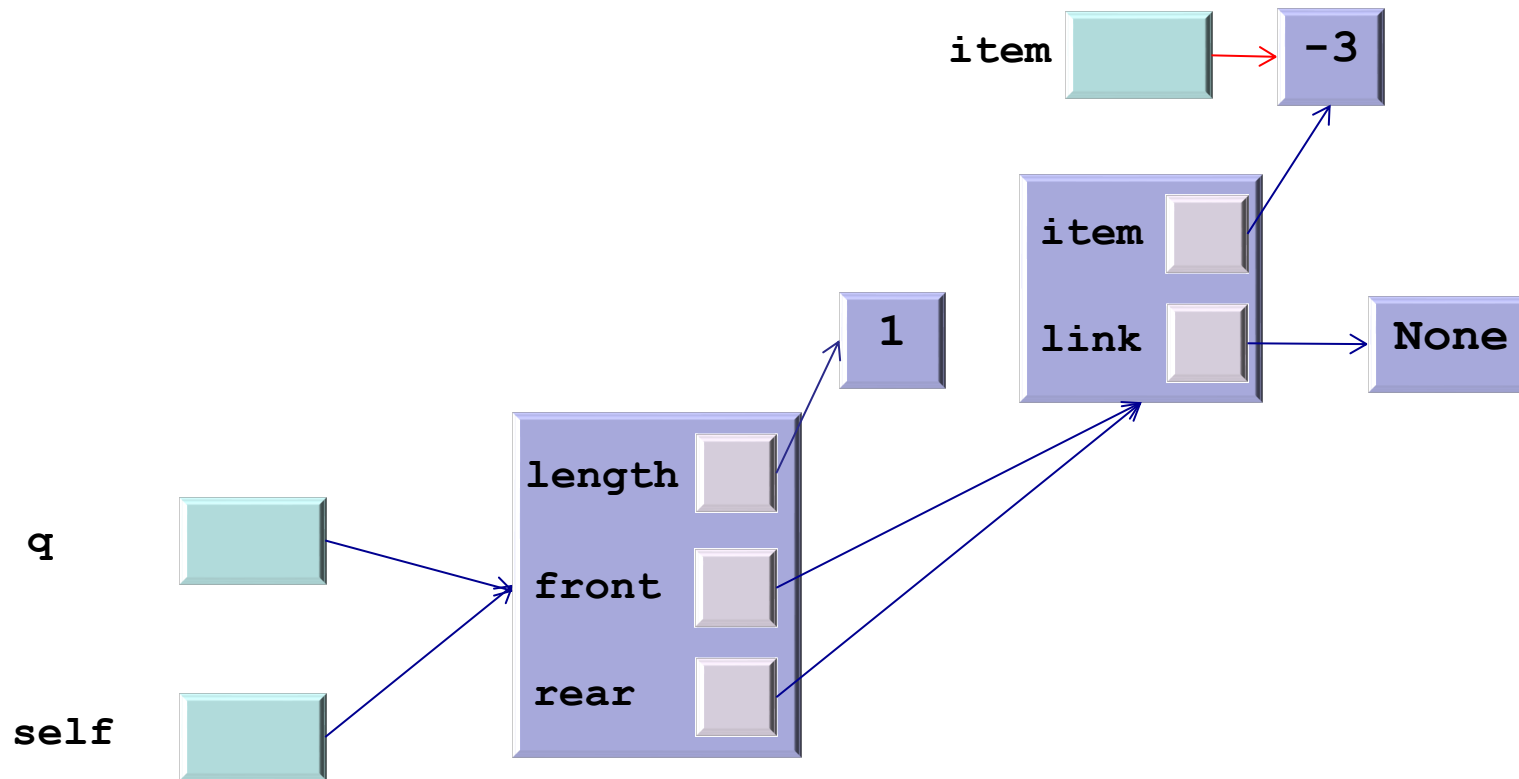
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
  
```





**q.serve()**

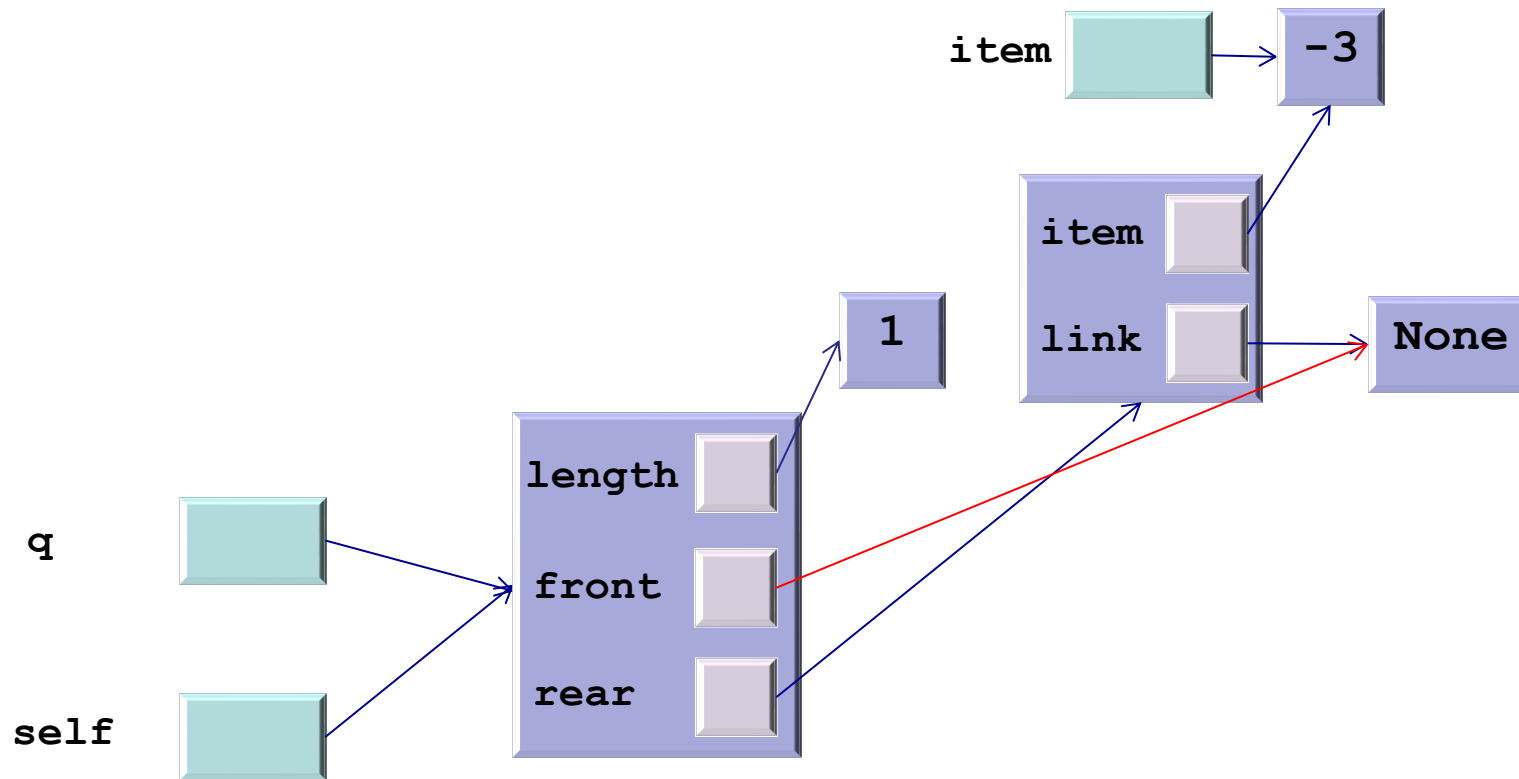
```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



**q. serve()**

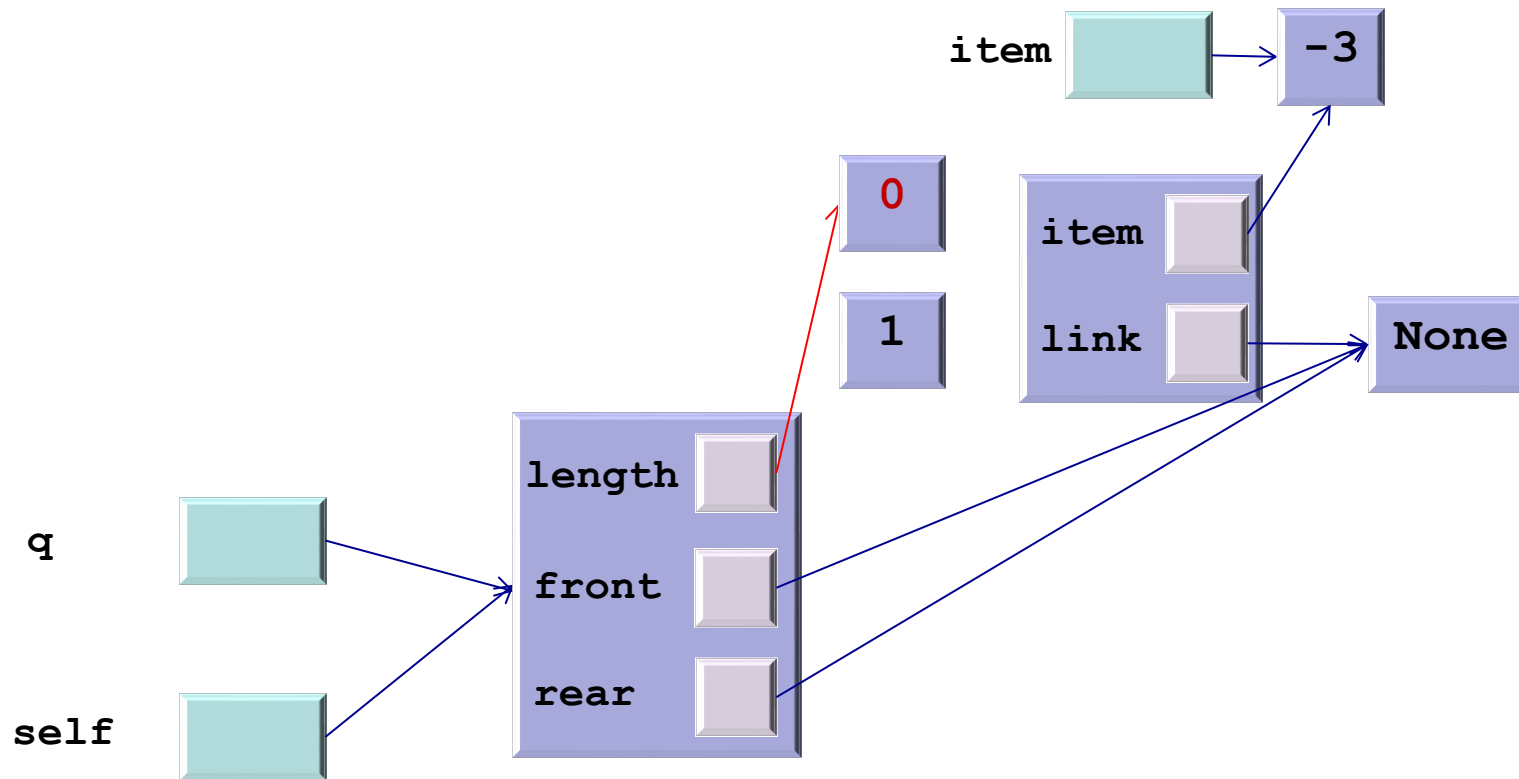
```

def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
  
```



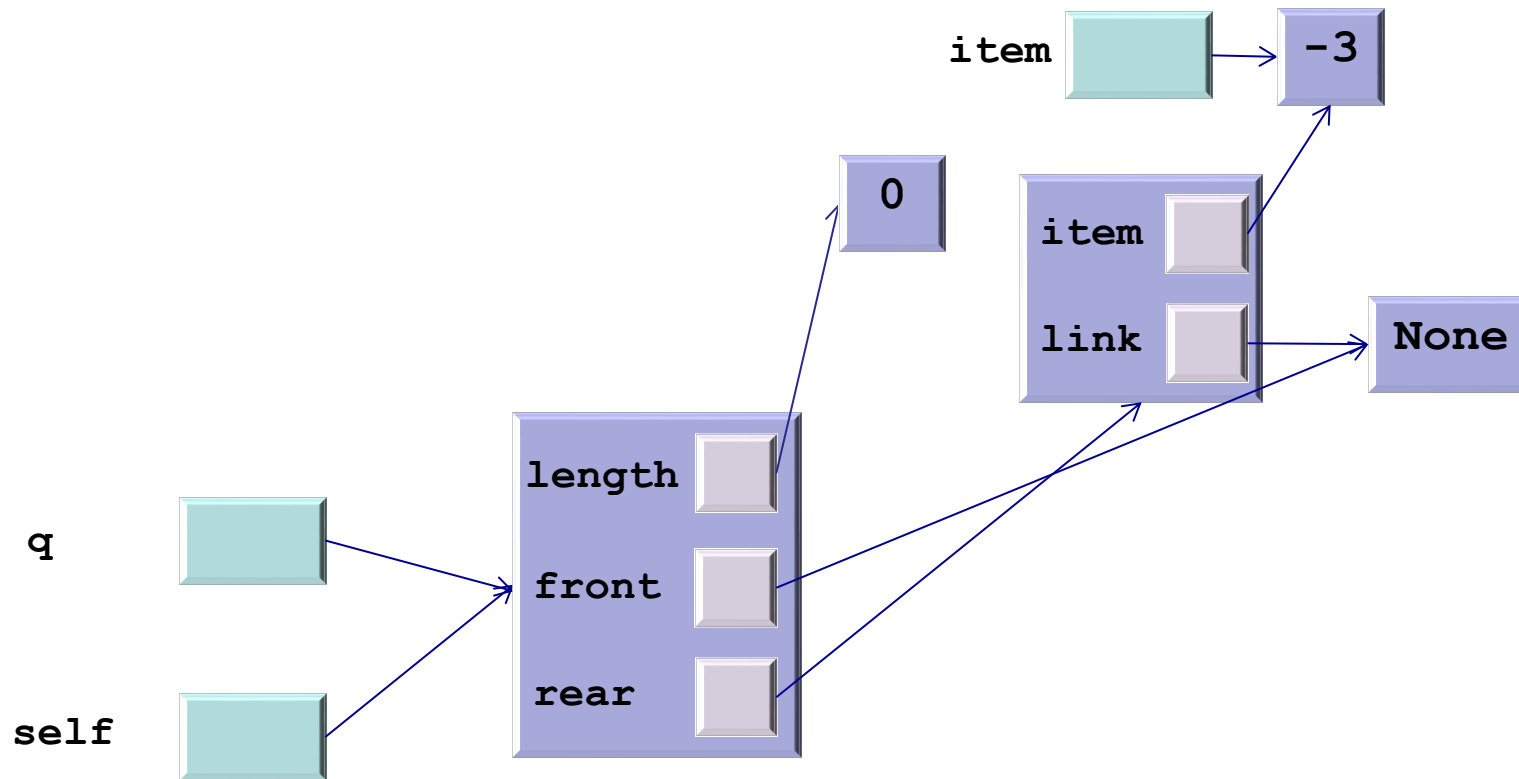
`q.serve()`

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



`q. serve()`

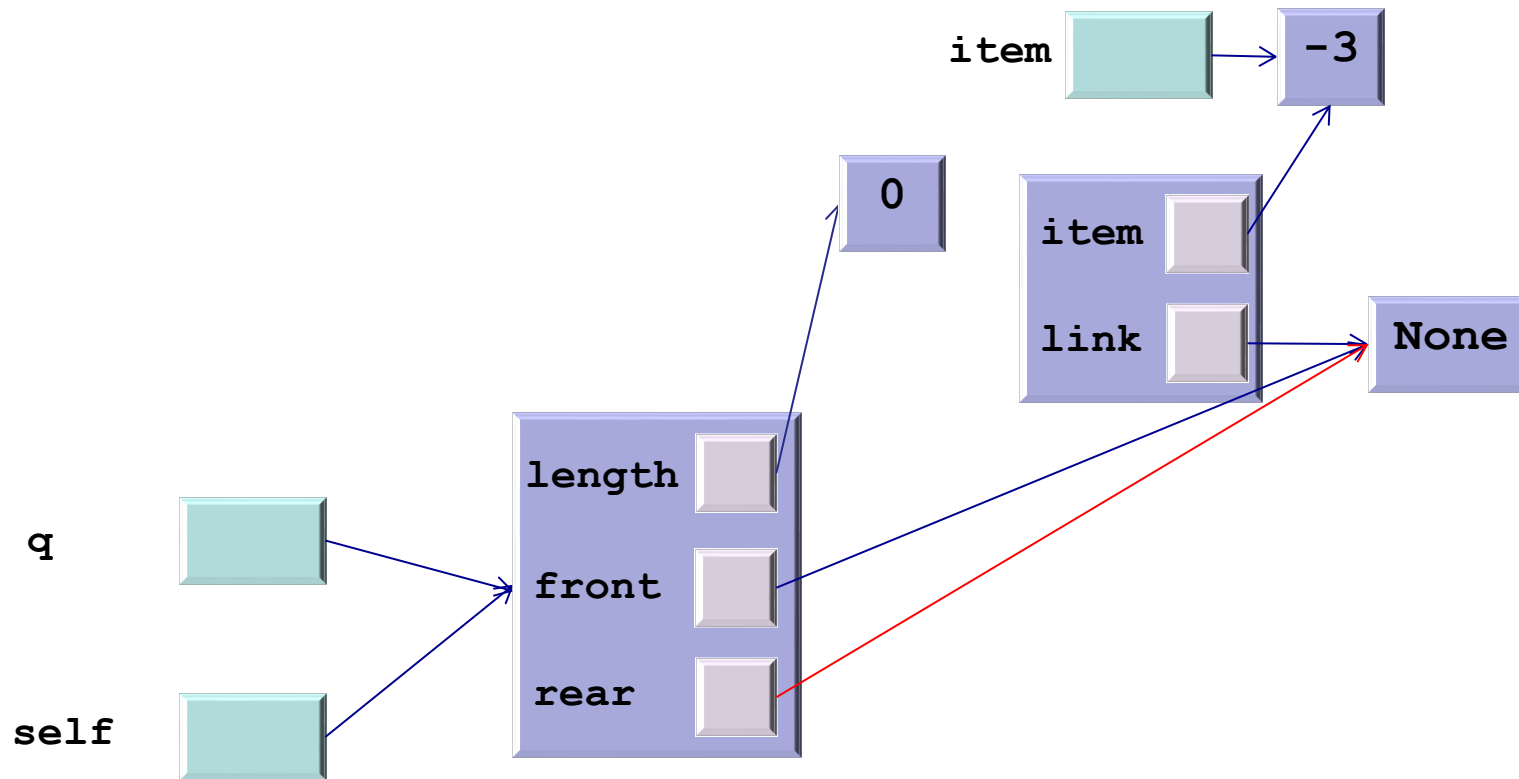
```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



**q. serve()**

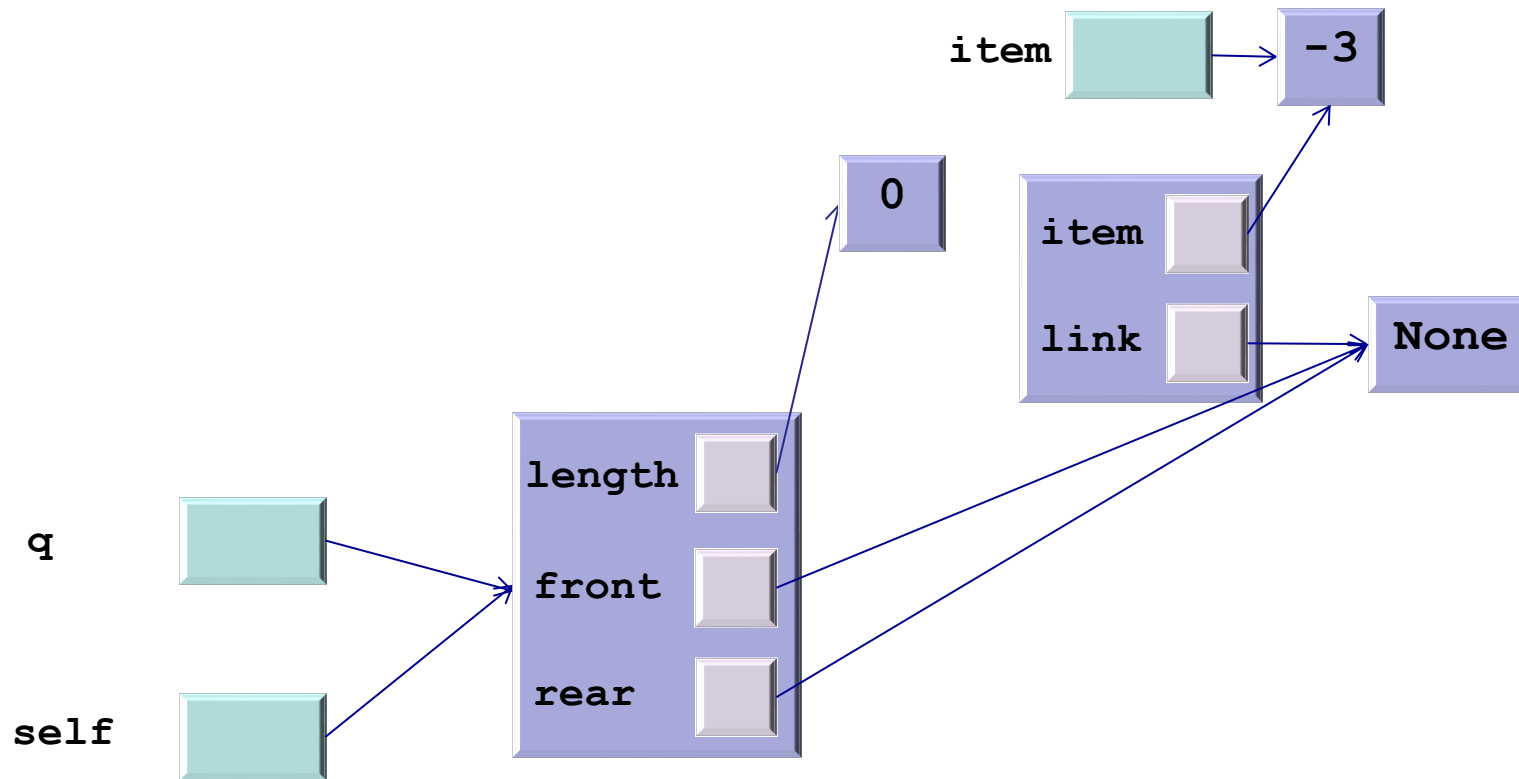
```

def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
  
```



**q.serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```



**q. serve()**

```
def serve(self) -> T:
    if not self.is_empty():
        item = self.front.item # store the item to serve
        self.front = self.front.link # move front
        self.length -= 1
        if self.is_empty(): # if now empty
            self.rear = None # move rear
        return item
    else:
        raise ValueError("Queue is empty")
```

# Using LinkQueue



Exercise: add to **LinkQueue** a method that halves the queue by deleting the nodes at index 1, 3, 5, etc

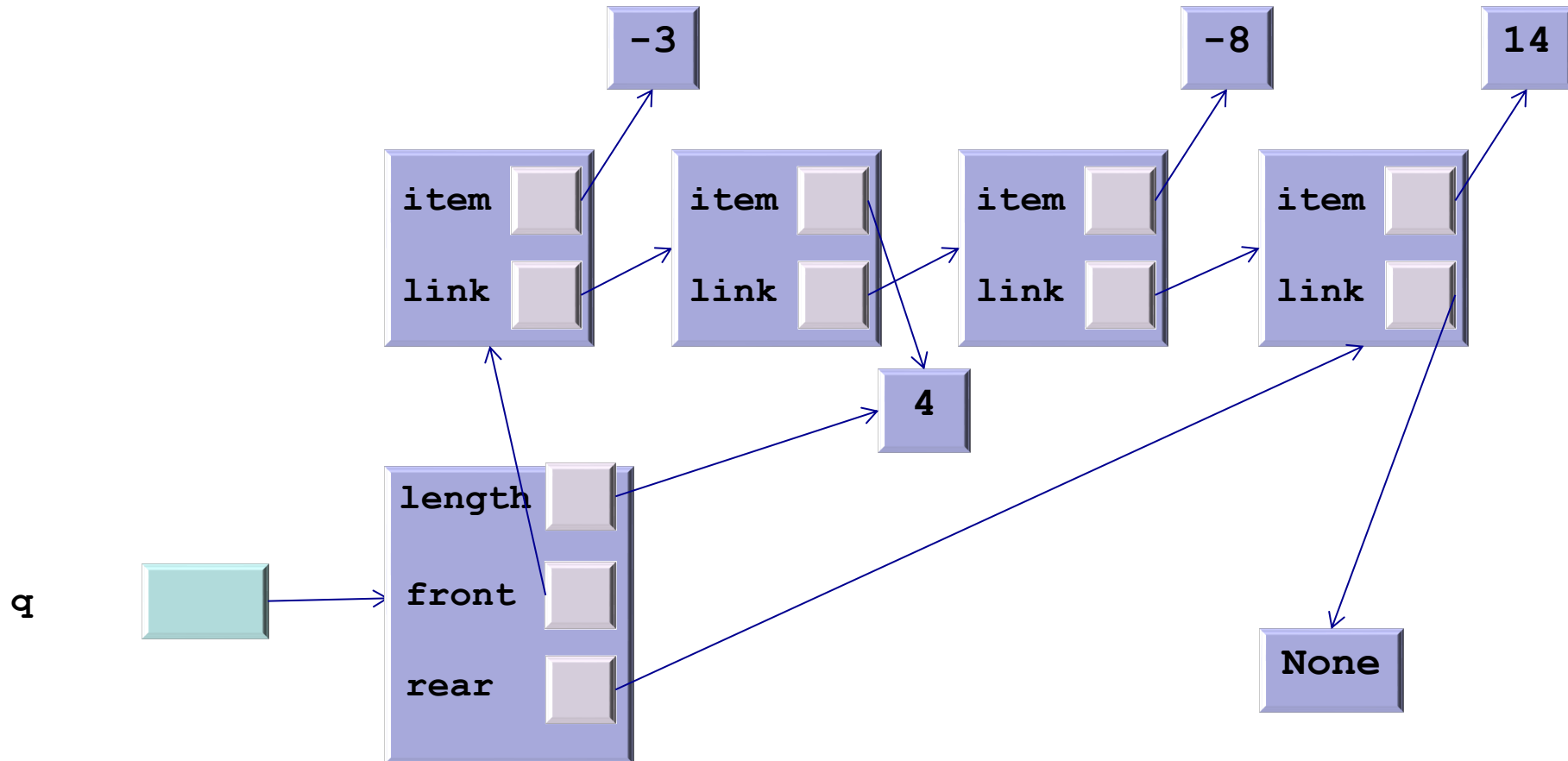
```
def halve(self) -> None:
    current = self.front
    # while at least two elements not traversed in the queue
    while current is not None and current.link is not None:
        if current.link is self.rear: # if even node is last
            self.rear = current # move rear up
        current.link = current.link.link # bypass odd node
        current = current.link # keep on traversing - next two
    length -= 1
```

```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
    length -= 1

```

**q.halve()**

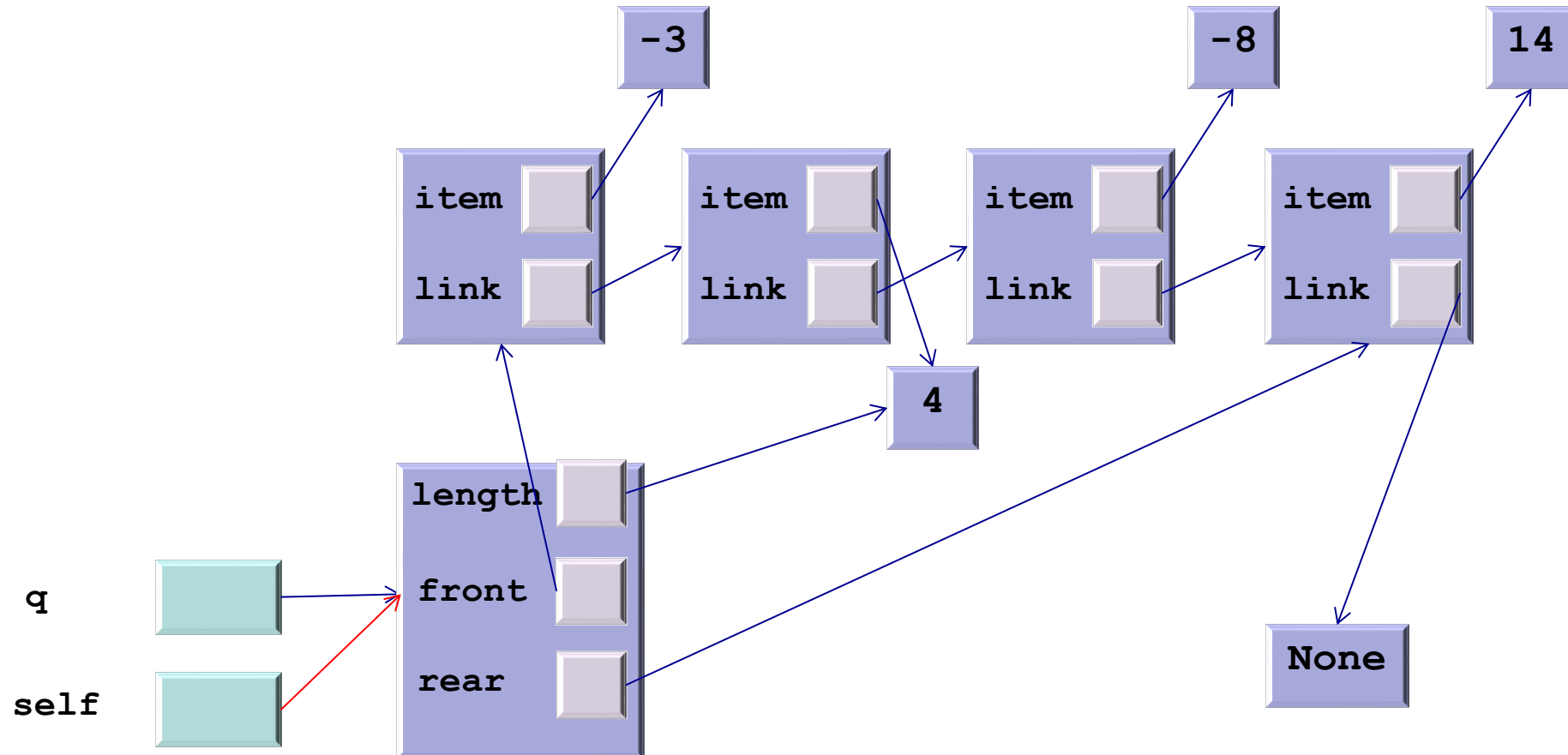


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
    length -= 1

```

q.halve()

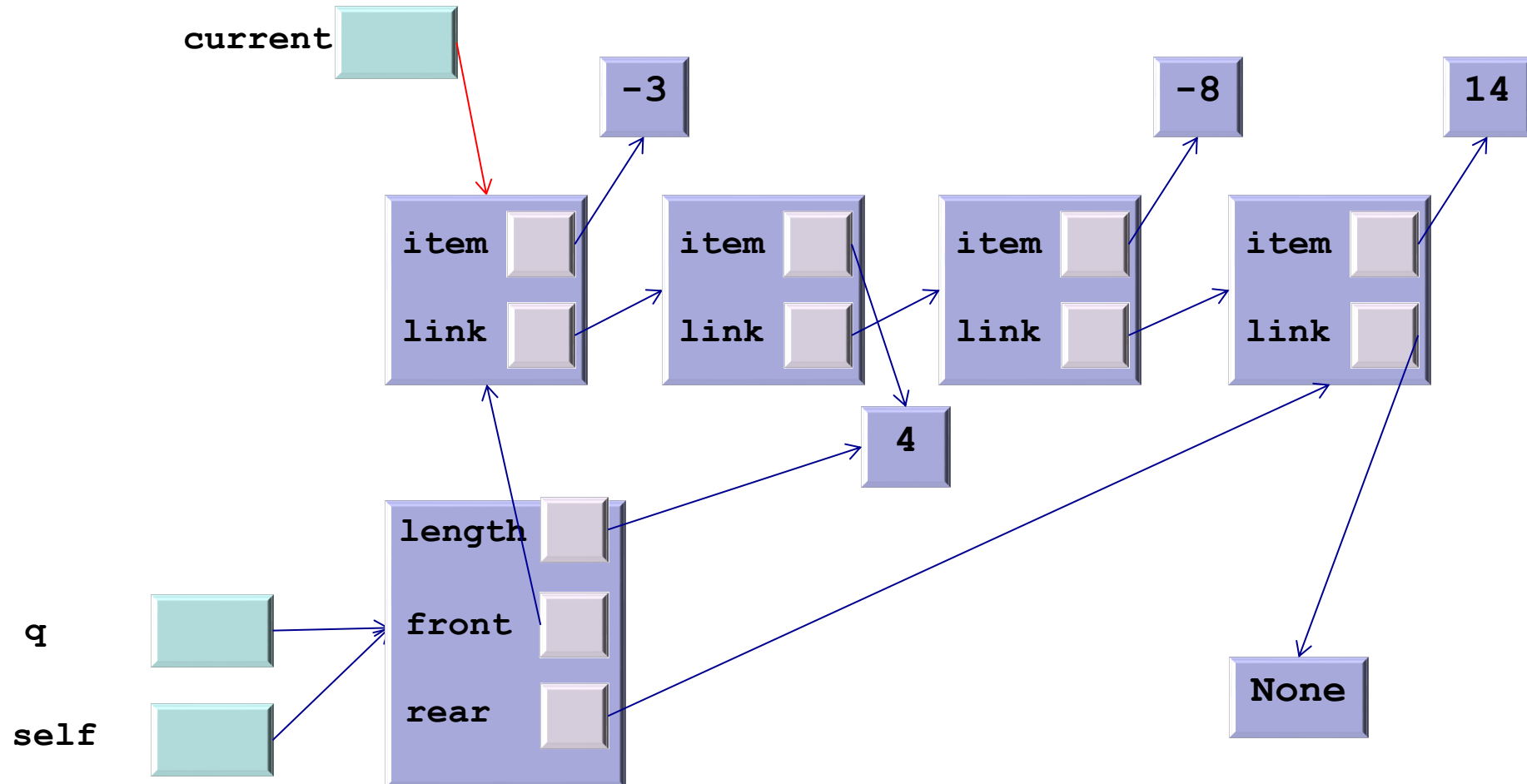


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
        length -= 1

```

q.halve()

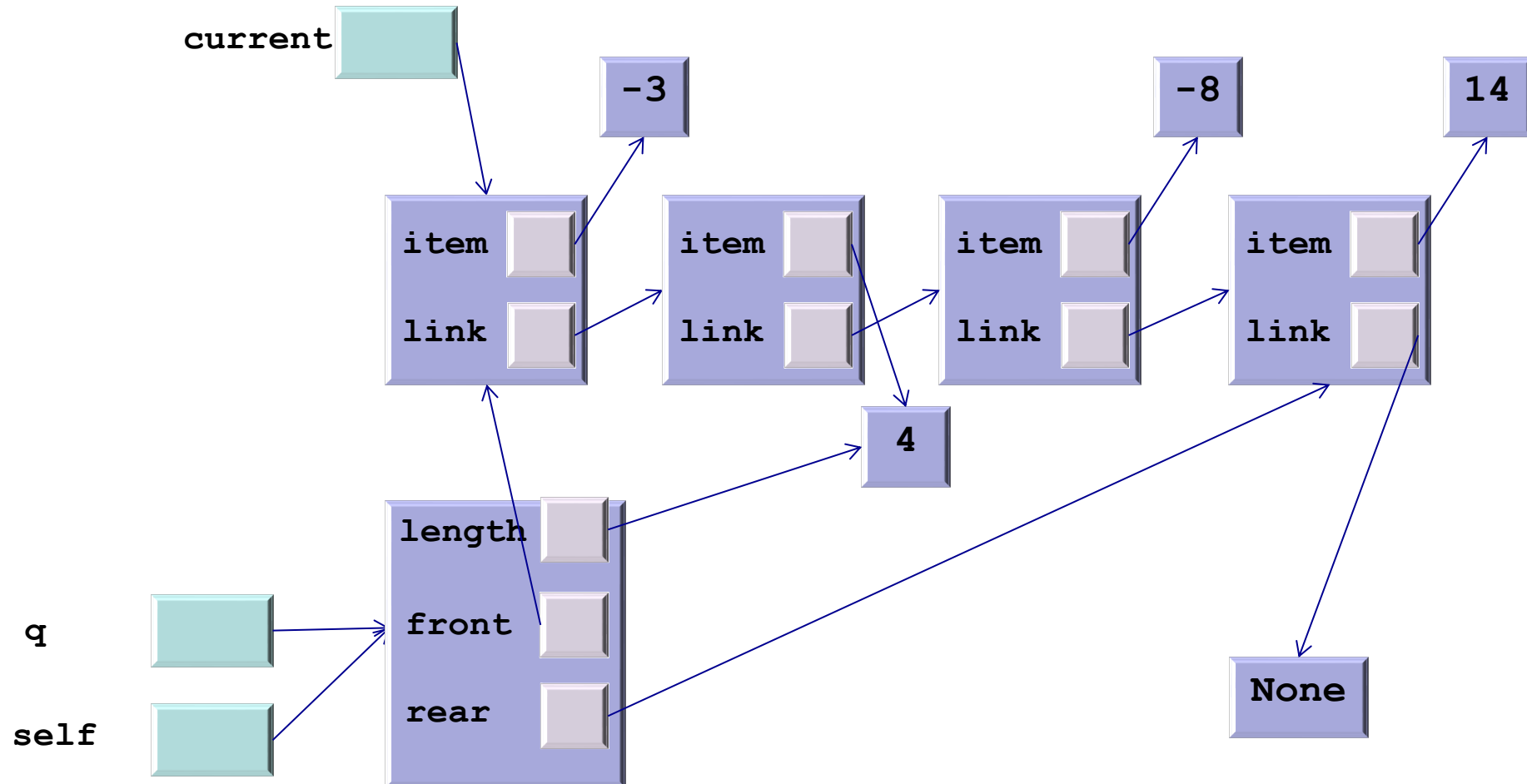


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
        length -= 1

```

q.halve()

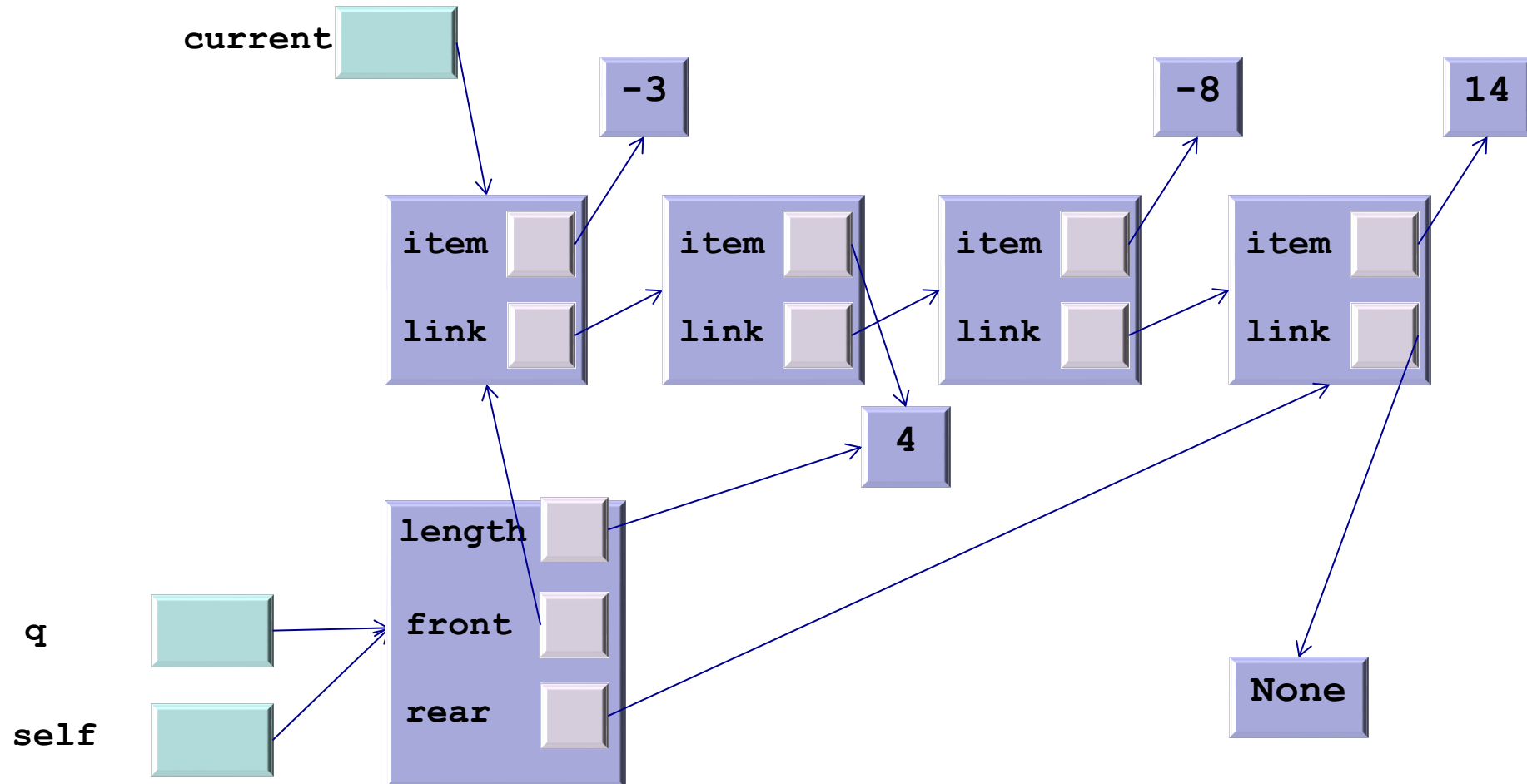


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
    length -= 1

```

q.halve()

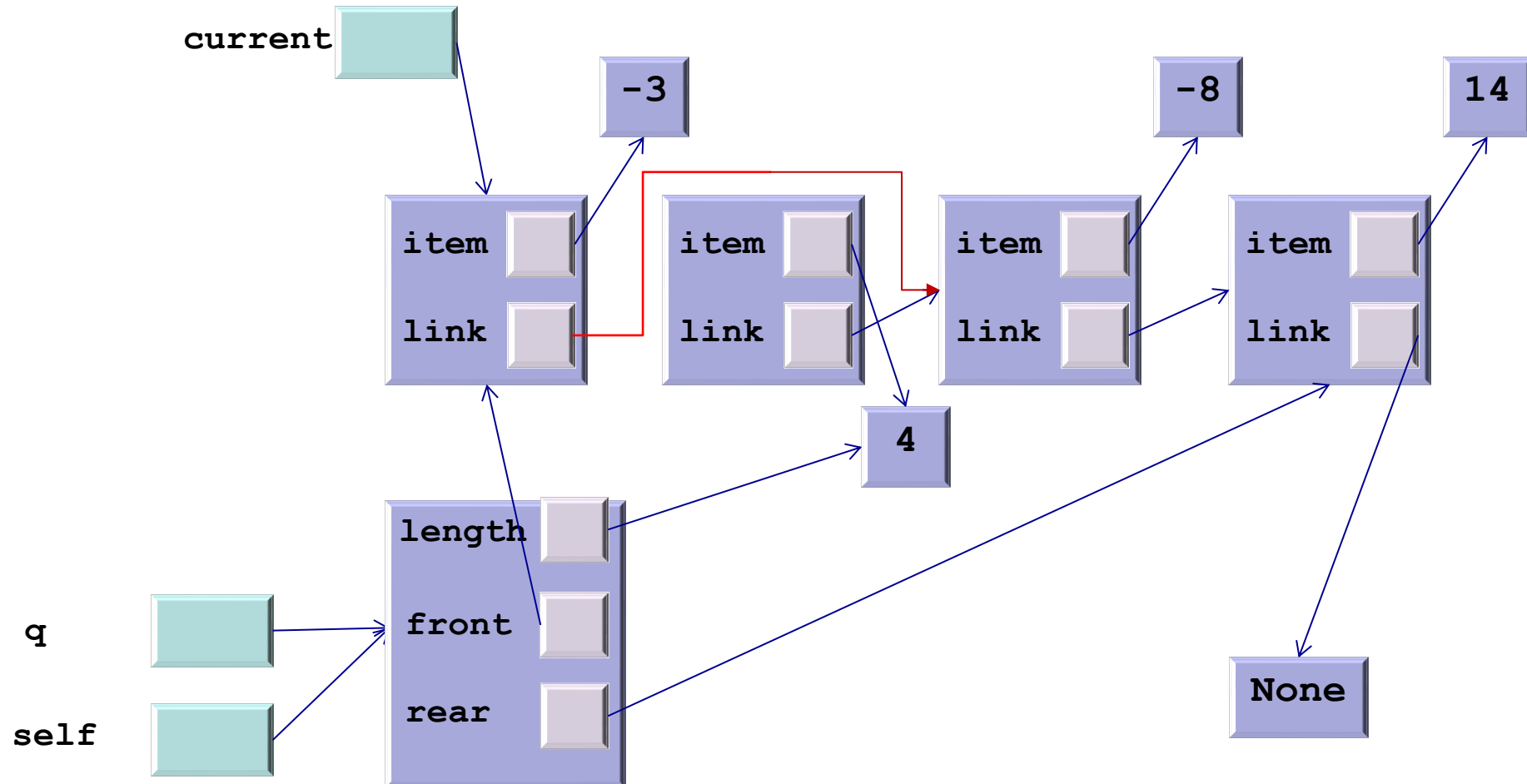


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
            current.link = current.link.link
            current = current.link
        length -= 1

```

q.halve()

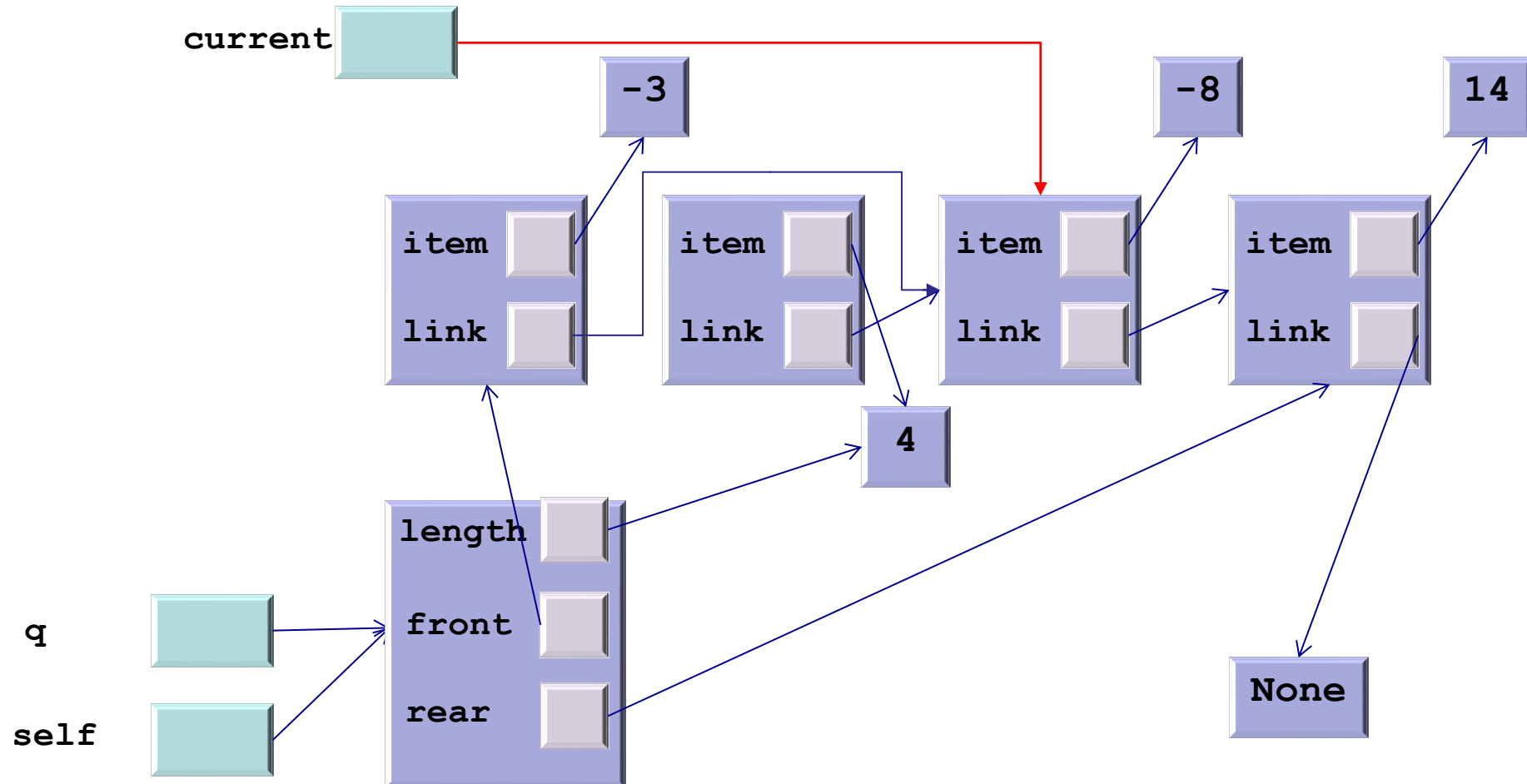


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
        length -= 1

```

q.halve()



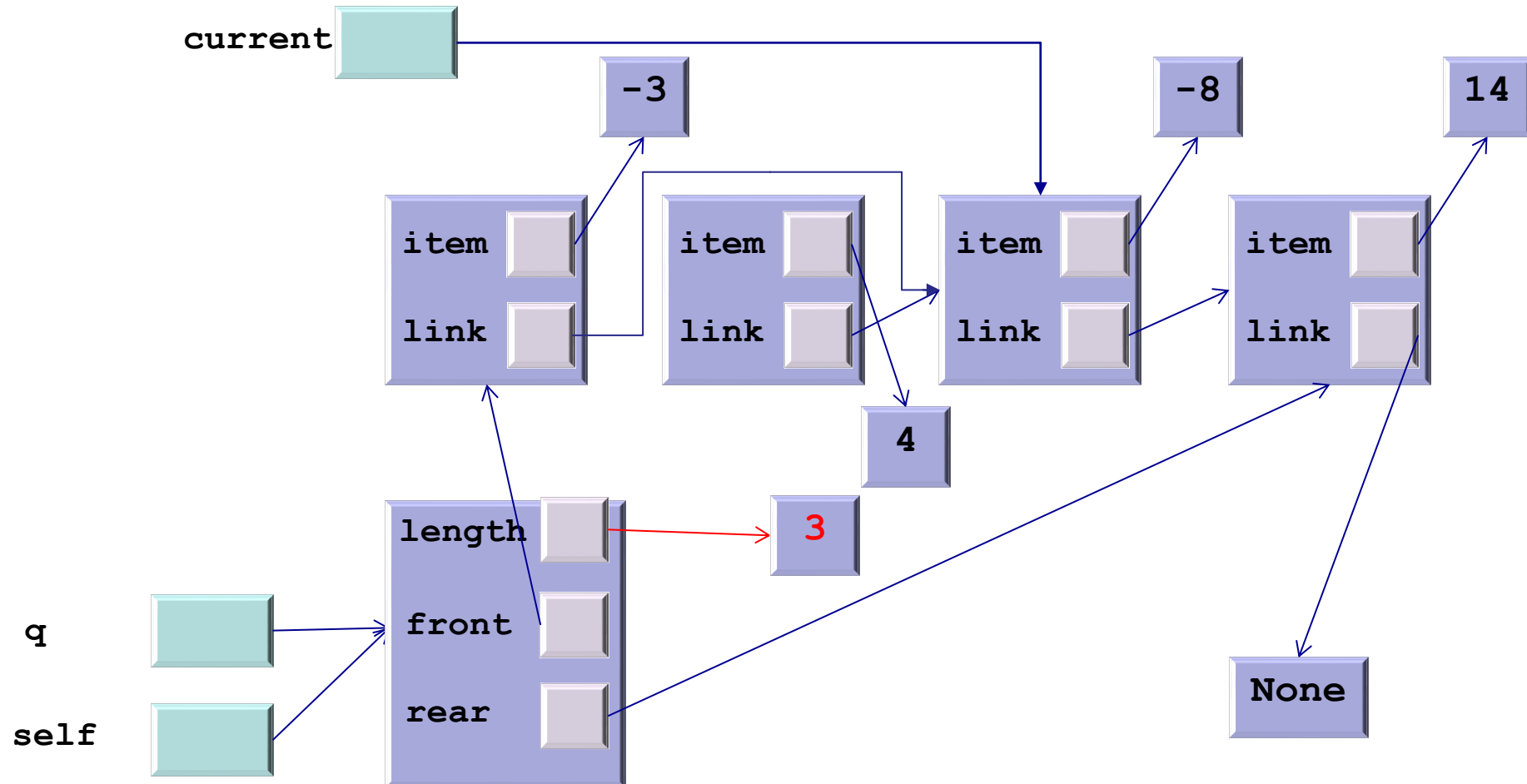


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
    length -= 1

```

q.halve()

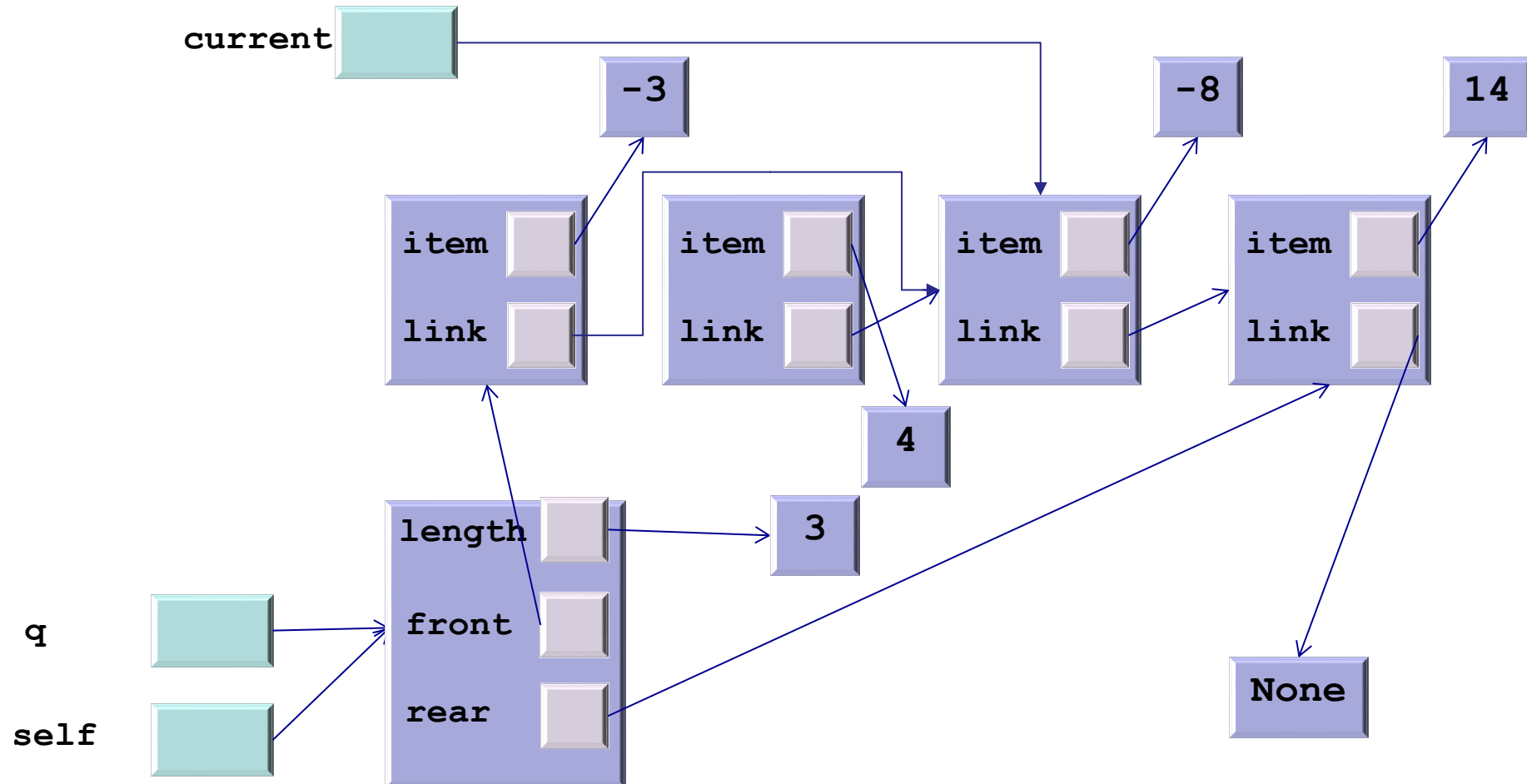


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
        length -= 1

```

q.halve()

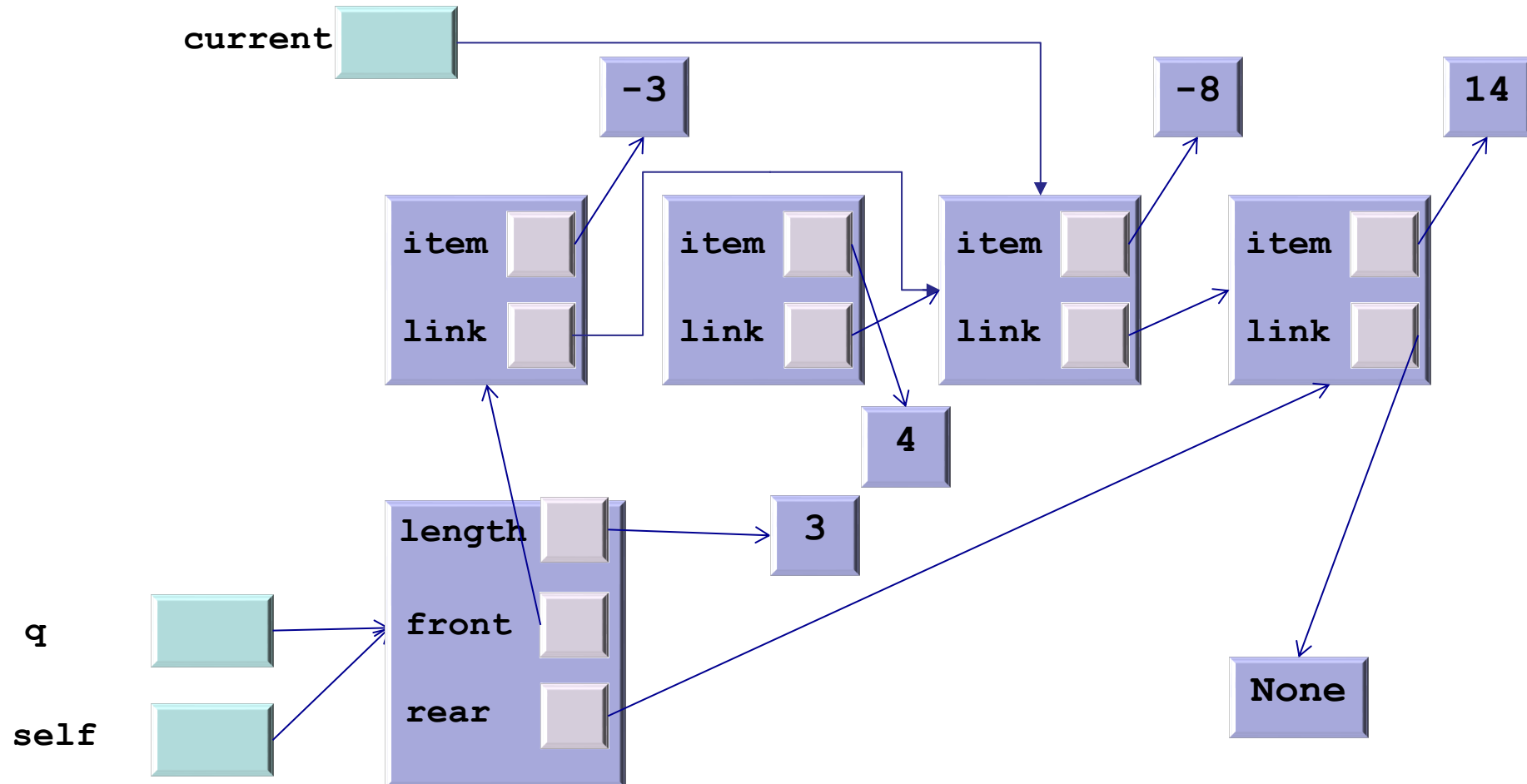


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
        length -= 1

```

q.halve()

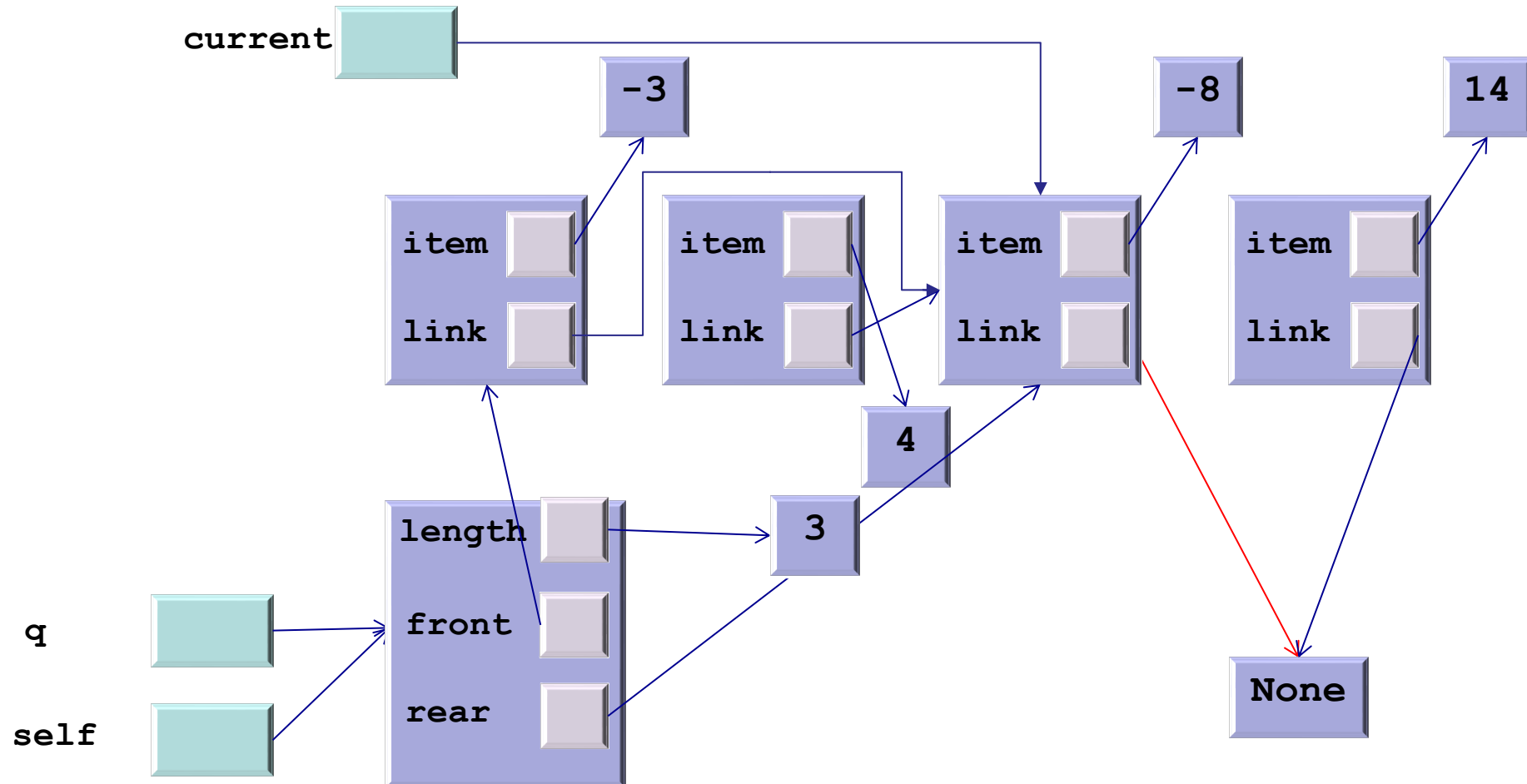


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
    length -= 1

```

q.halve()

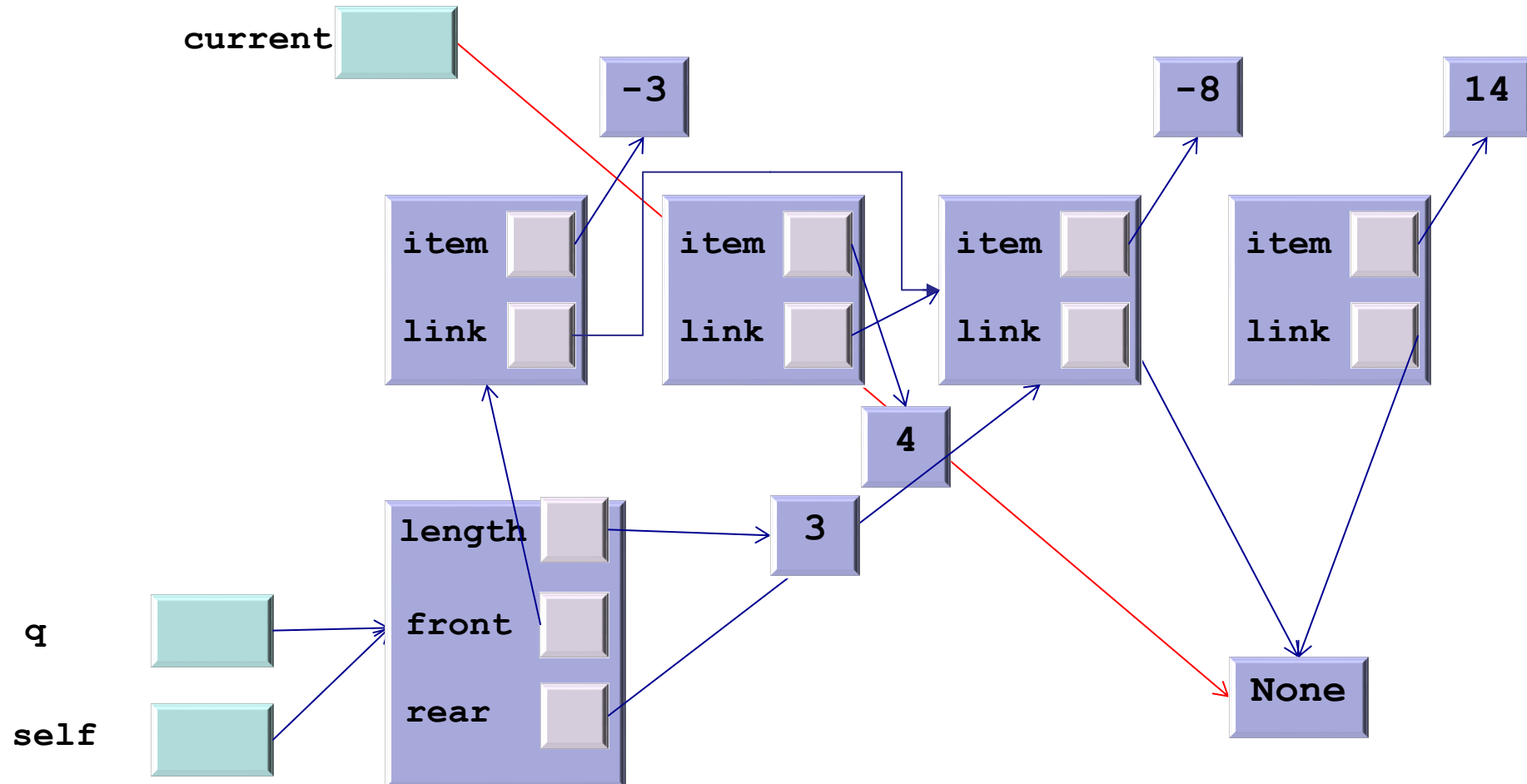


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
        length -= 1

```

q.halve()

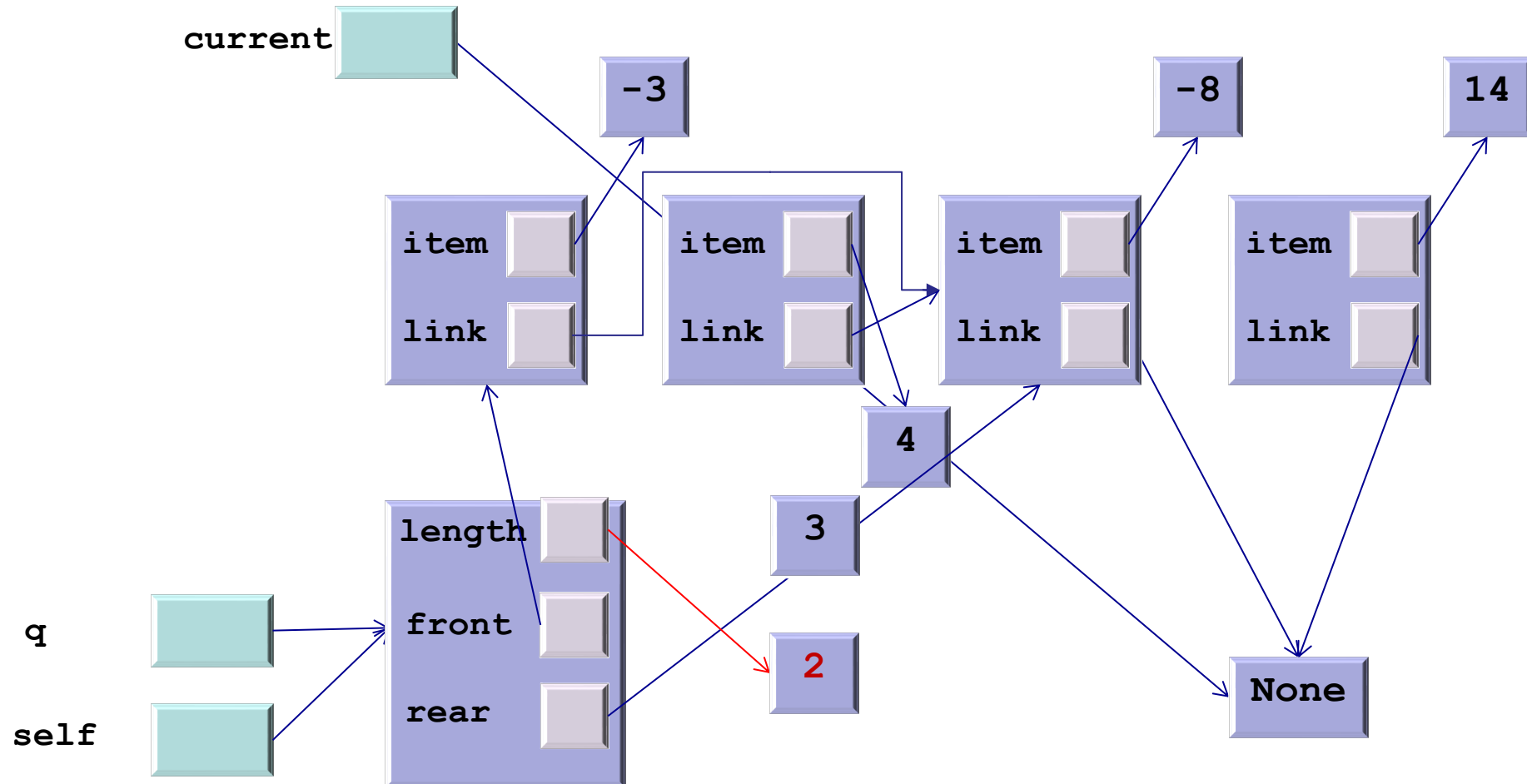


```

def halve(self) -> None:
    current = self.front
    while current is not None and current.link is not None:
        if current.link is self.rear:
            self.rear = current
        current.link = current.link.link
        current = current.link
    length -= 1

```

q.halve()



# Remember: Abstract Data Types

- **Any data type provides:**
  - **Storage** for a collection of data items
  - A set of **operations** to interact with the data
- **An abstract one does not provide any info on how:**
  - The storage is **organised**
  - The operations are **implemented**
- **Users can only interact with the data through the provided operations**

***Separates the WHAT from the HOW  
and ignores the HOW***

# Abstract Data Types (cont)

- **Example: a Stack ADT has operations:**
  - push, pop, is\_empty, reset, etc
  - Implementation? Could be array, could be linked, or something else, a user does not know
- **As a user I just need to know its operations**
- **Do not confuse Data Type with Data Structure**
  - Data Structure: particular way in which the data is organised (structured) in memory
  - The way a given Data Type is implemented



# Abstract Data Types: pros and cons

- **Main advantage: maintenance**

- Changing the implementation of the ADT does not mean changing the user's code

- **Main disadvantage: efficiency**

- Having access to the implementation (ADT as an inner class) might allow good programmers to improve time/space performance

# Abstract Data Types: advice

- **Always design your data types abstractly**
  - Use the class methods if you can (even as god!)
- **Late modifications to its implementation will not affect the rest of your code**
- **Readability** is also improved: use meaningful names for operations
- **Correctness** easier to verify: after proper testing to all methods

# Summary

- **We now understand how to use linked data structures in implementing**
  - Stacks
  - Queues
- **We are able to:**
  - Implement, use and modify linked stacks and linked queues
  - Decide when it is appropriate to use them (rather than arrays)