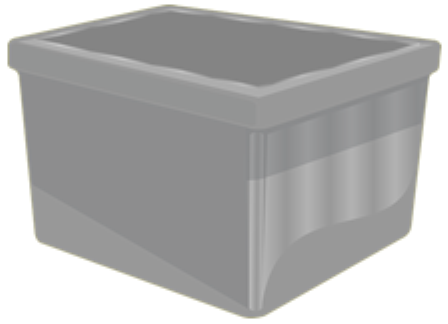
Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 20
Linked Lists

FIT1008&2085
Introduction to Computer Science

MONASH University
Information Technology



Container ADTs

	Array-based-implementation	Linked implementation
Stacks	Done	Done
Queues	Done	Done
Lists	Done	?

Objectives

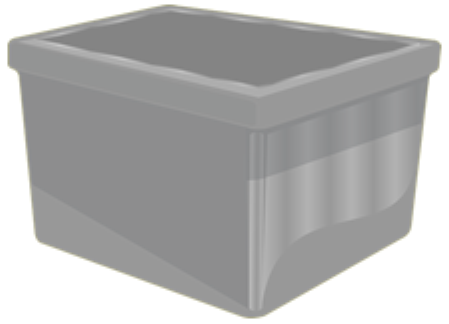
- To understand the use of linked data structures in implementing **Linked lists**

- To be able to:
  - Implement, use and modify linked lists.
  - Decide when is it appropriate to use them (as opposed to using the ones implemented with arrays)

List ADT

- Sequence of items

- Possible Operations:
    - Create a list
    - Insert an item before a given position in the list
    - Delete an item at a given position from the list
    - Check whether the list is empty
    - Check whether the list is full
    - Get the length of the list.



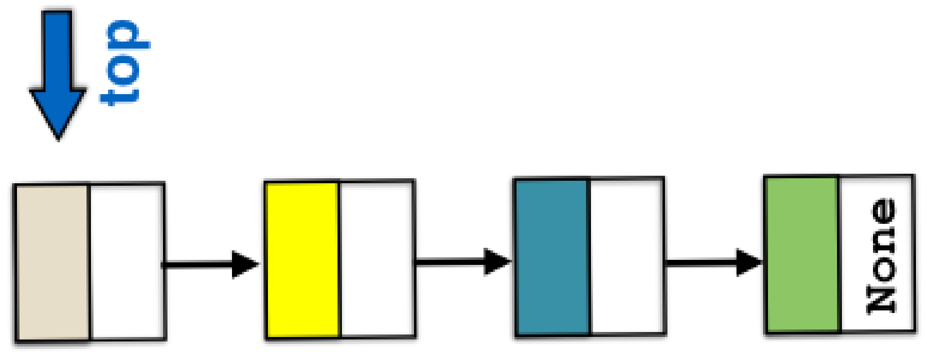
Container ADTs

	Array-based-implementation	Linked implementation
Stacks	Done	Done
Queues	Done	Done
Lists	Done	?

Access elements at specific locations: Stacks, Queues

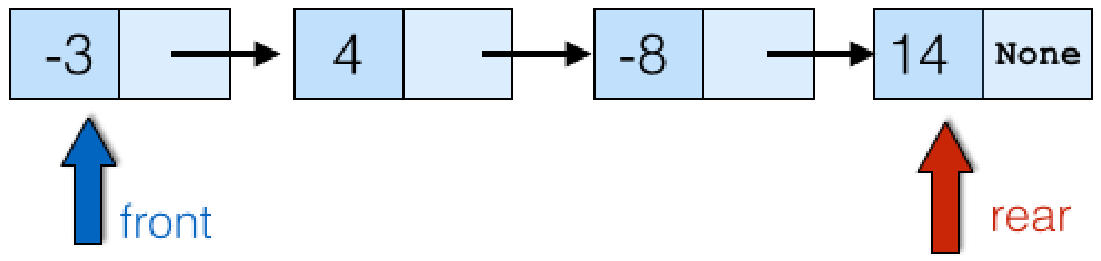
Access any element: Lists

Access the top element only



```
class Stack:
    def __init__(self):
        self.top = None
```

Append to rear
Serve front

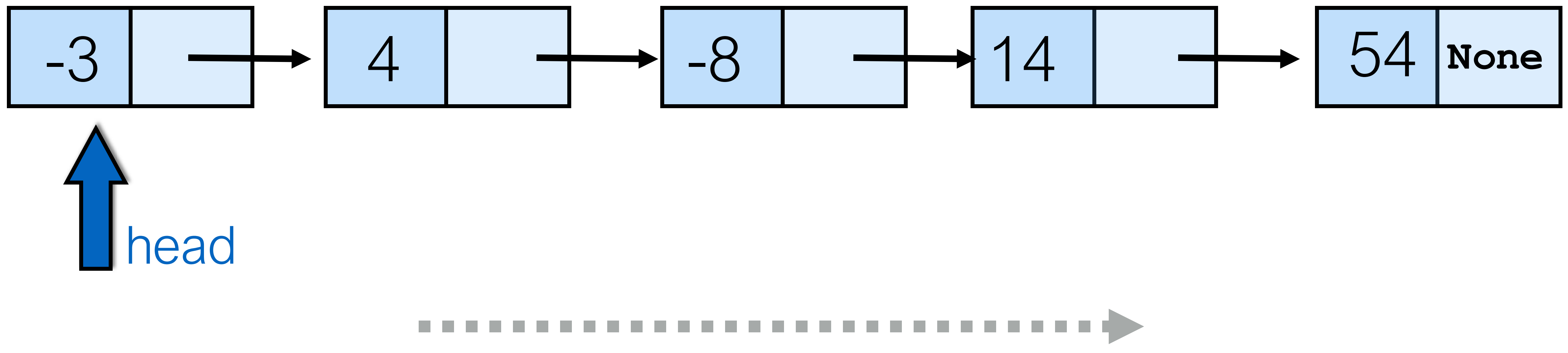


```
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
```

Access
any
Node

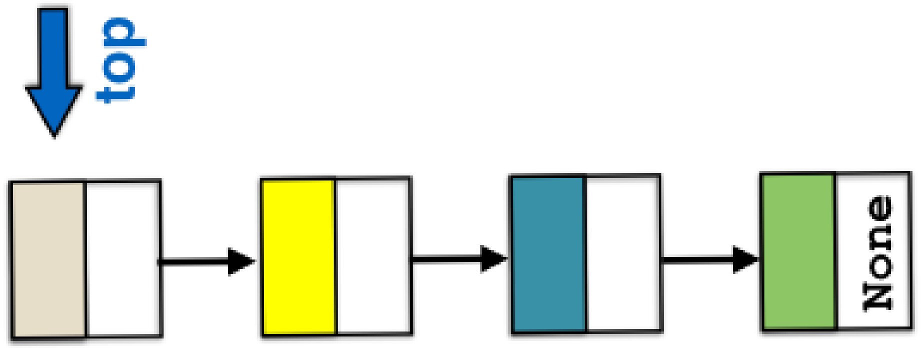
?

?



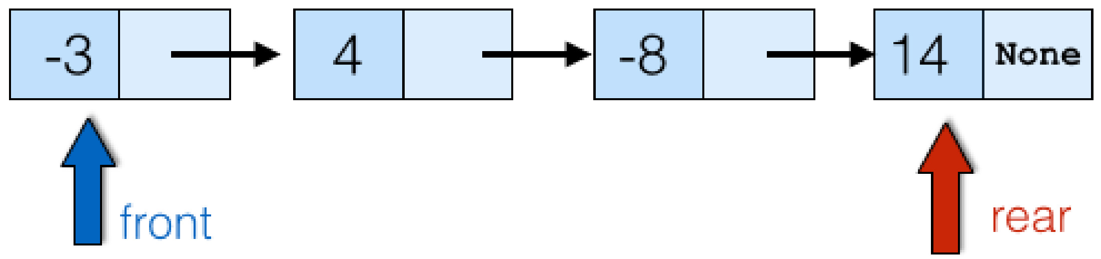
count from head to access elements

Access the top element only



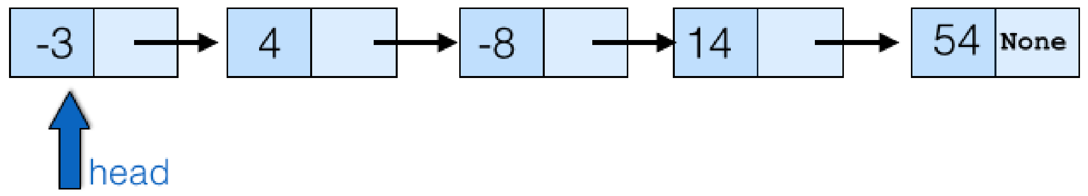
```
class Stack:
    def __init__(self):
        self.top = None
```

Append to rear
Serve front



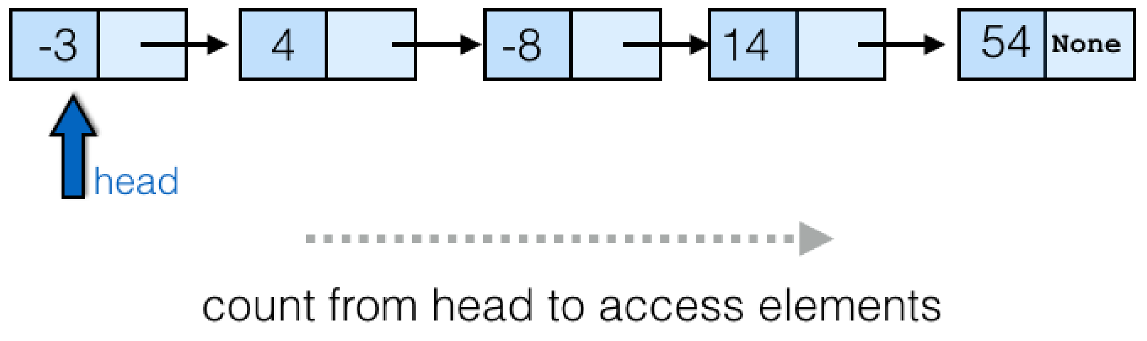
```
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
```

Access
any
Node



?

Linked Lists

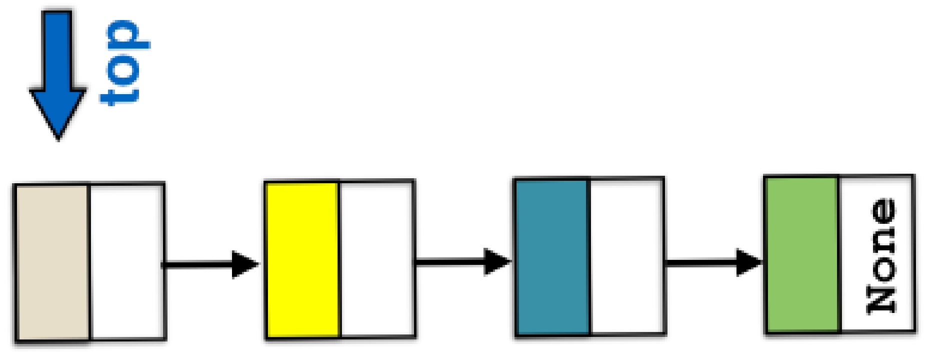


- What instance variables have we used for stacks and queues?
  - **Stacks**: top (only place where we push and pop elements from)
  - **Queues**: front and rear (we append to the rear, serve from the front)
- What instance variables do we need for **lists**?Only one component: a reference to the **head node**
- From there, we can access every other node

Not strictly necessary,
but it will be useful

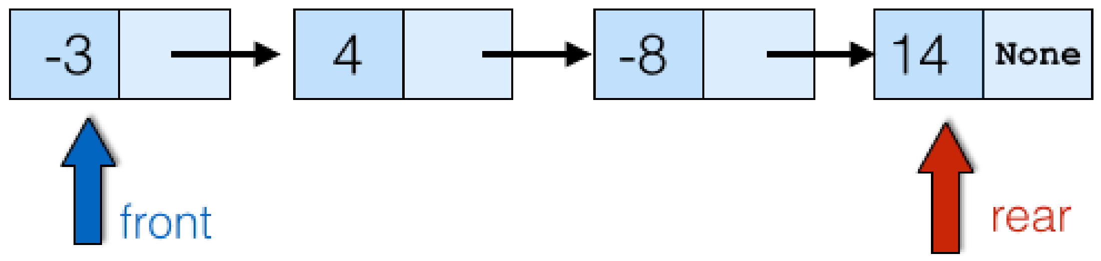
```
class List:
    def __init__(self):
        self.head = None
        self.count = 0
```

Access the top element only



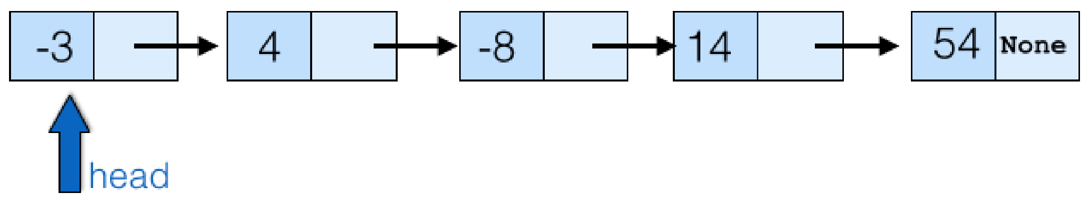
```
class Stack:
    def __init__(self):
        self.top = None
```

Append to rear
Serve front

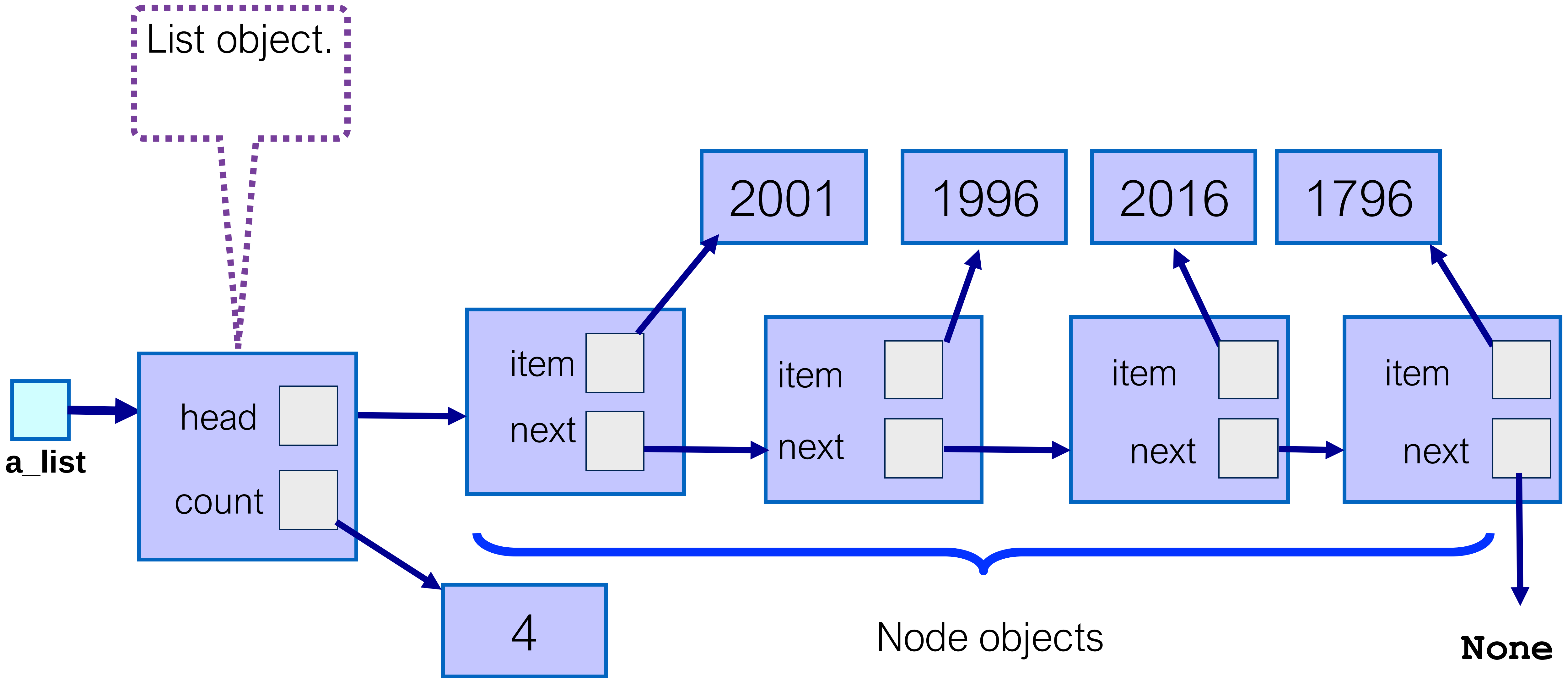


```
class Queue:
    def __init__(self):
        self.front = None
        self.rear = None
```

Access
any
Node



```
class List:
    def __init__(self):
        self.head = None
        self.count = 0
```


List object.
2001
1996
2016
1796
item
next
item
next
item
next
item
next
a_list
head
count
4
Node objects
None

```
class List:
```

```python
class List:
    def __init__(self):
        self.head = None
        self.count = 0

    def is_empty(self):
        return self.count == 0
```

An empty list will also have head of None

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

Linked lists are never full

```python
    def reset(self):
        self.__init__()

    def __len__(self):
       return self.count
```

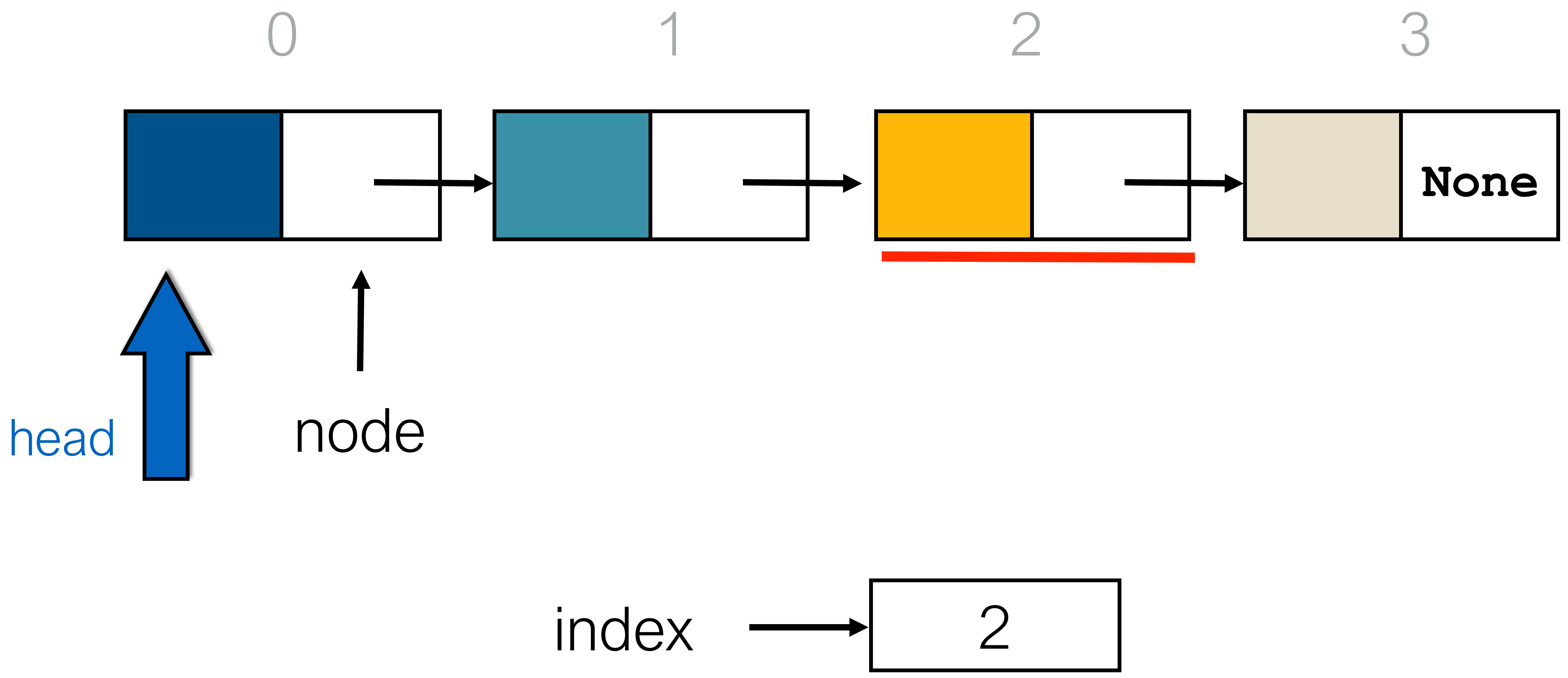
Insert and Delete

- **insert**(index, item)
  - Inserts **item** before position **index** in the list

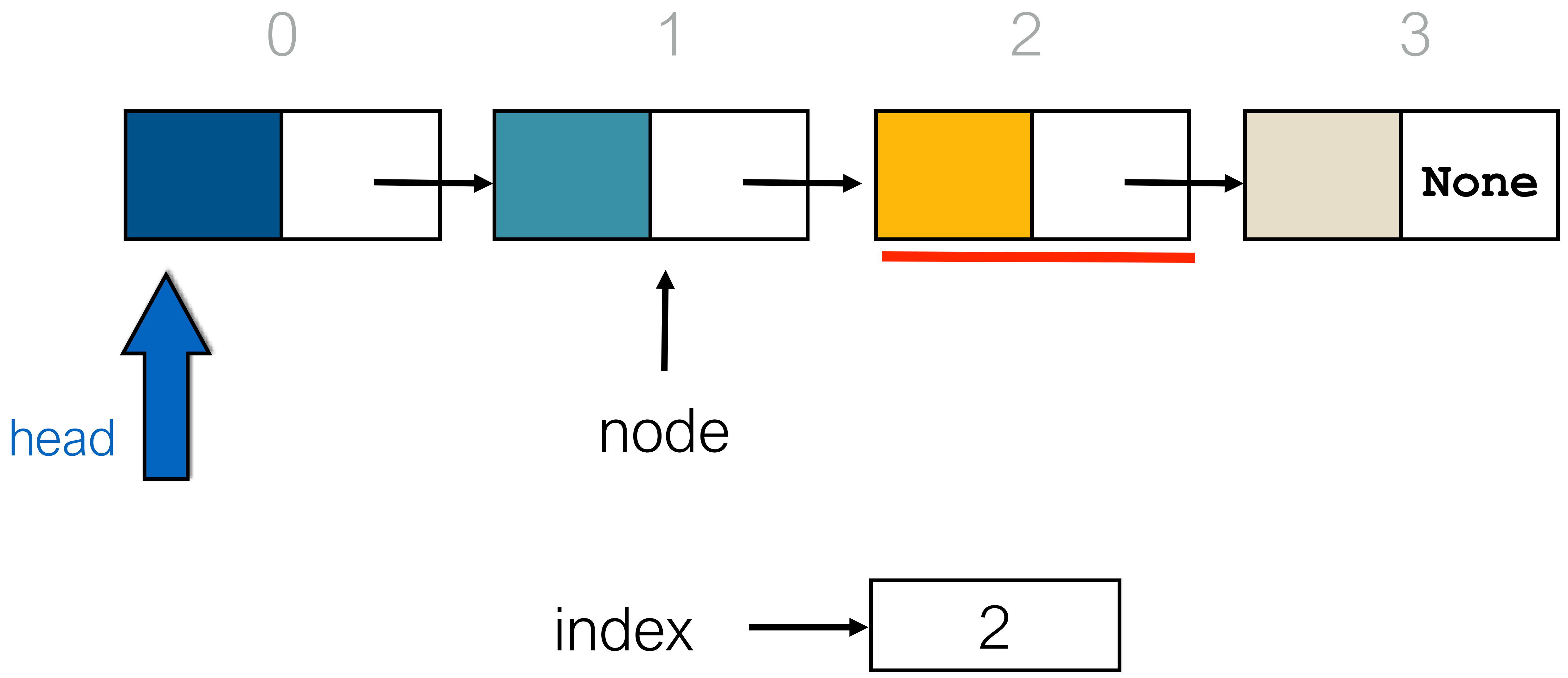
- **delete**(index)
  - Removes the **item** at position **index** in the list
  - Raises IndexError if the list is empty or the index is out of range
  - Similar to pop(index) in Python's list ADT

- Both require **_get_node**(self, index)
  - Returns a reference to the node at position index.
  - Internal "private" method

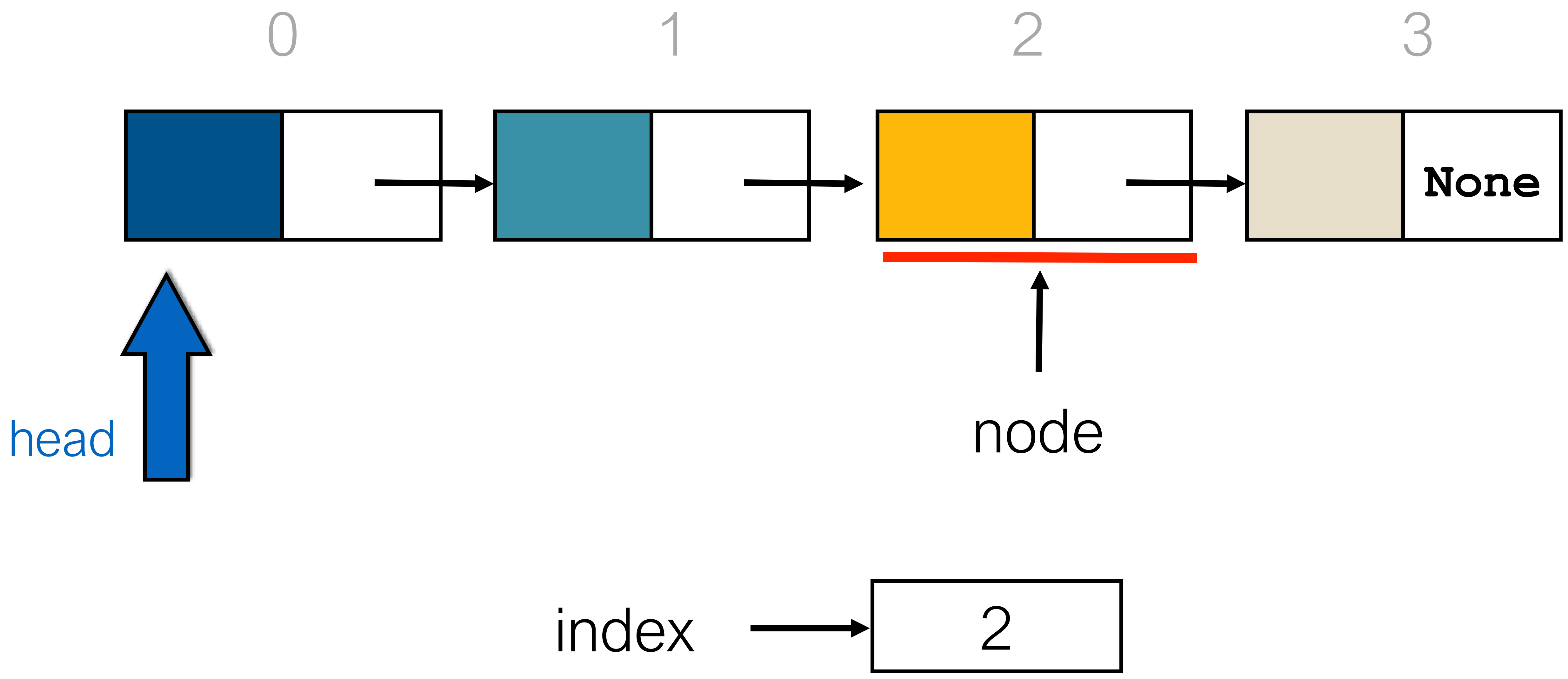
_get_node(self, **index**)



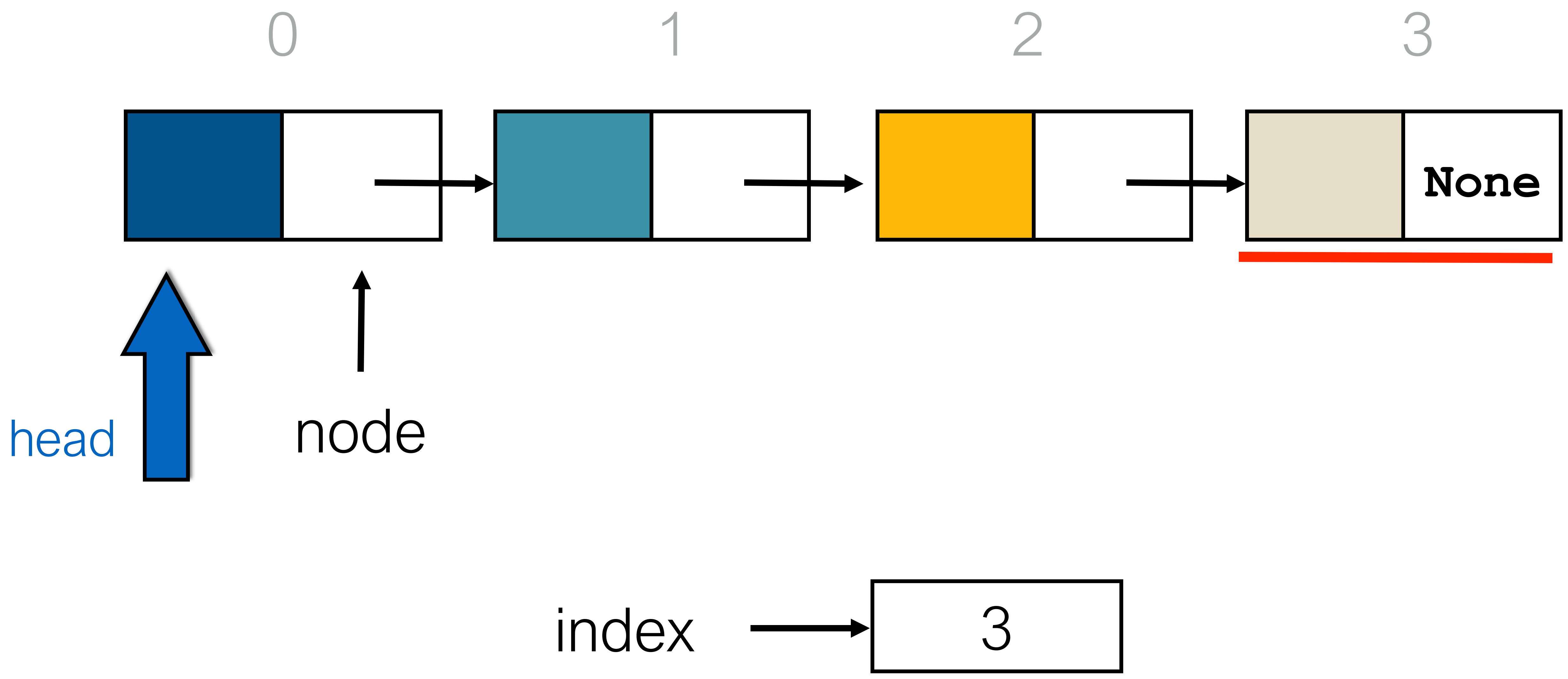
_get_node(self, **index**)



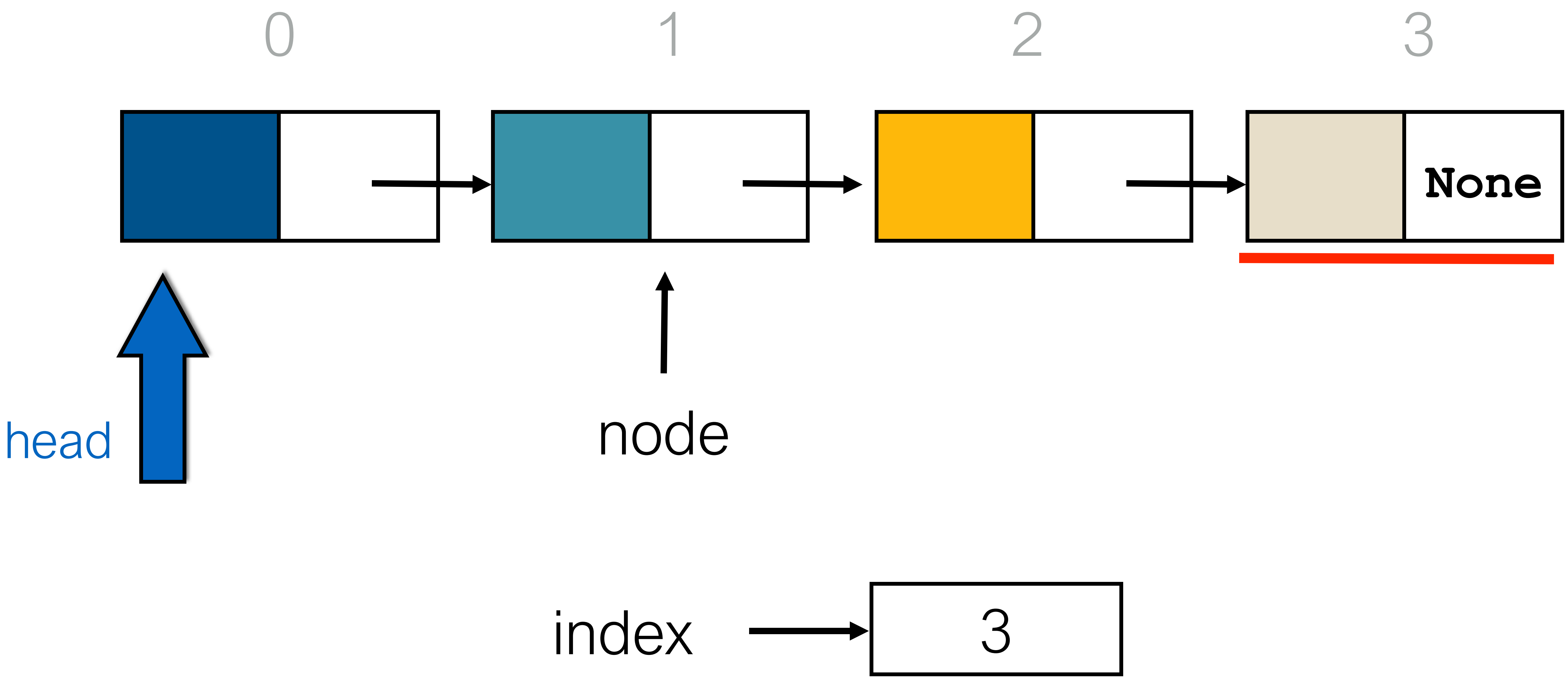
_get_node(self, **index**)



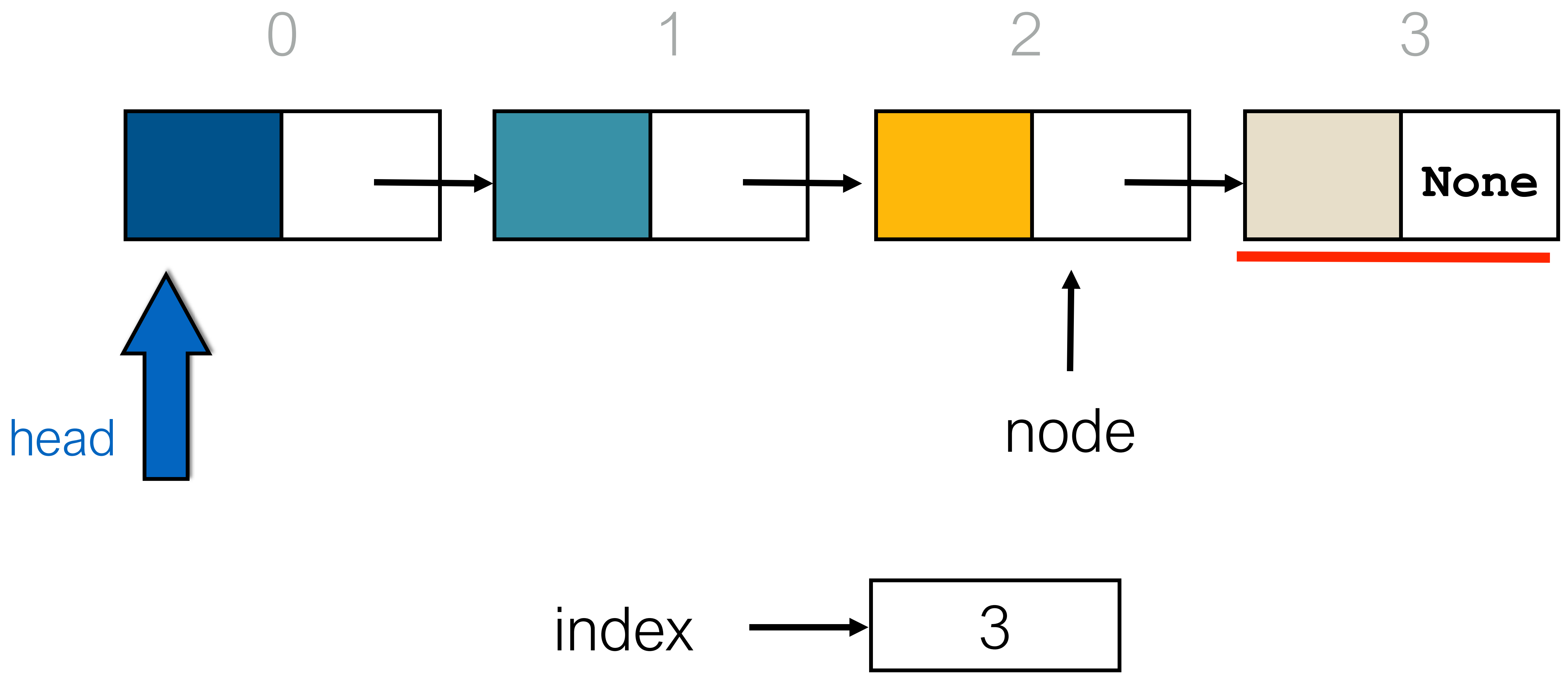
_get_node(self, **index**)



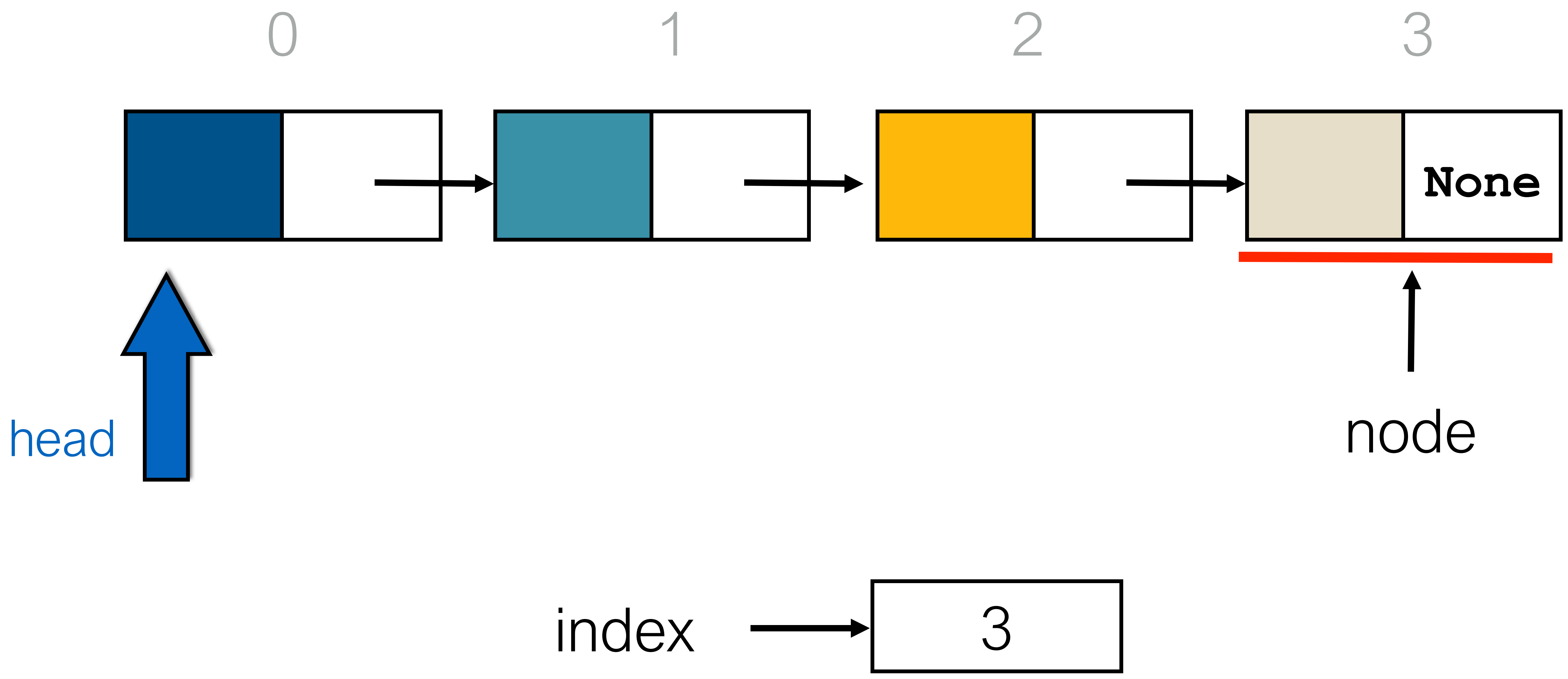
_get_node(self, **index**)



_get_node(self, **index**)



_get_node(self, **index**)



_get_node(self, **index**)

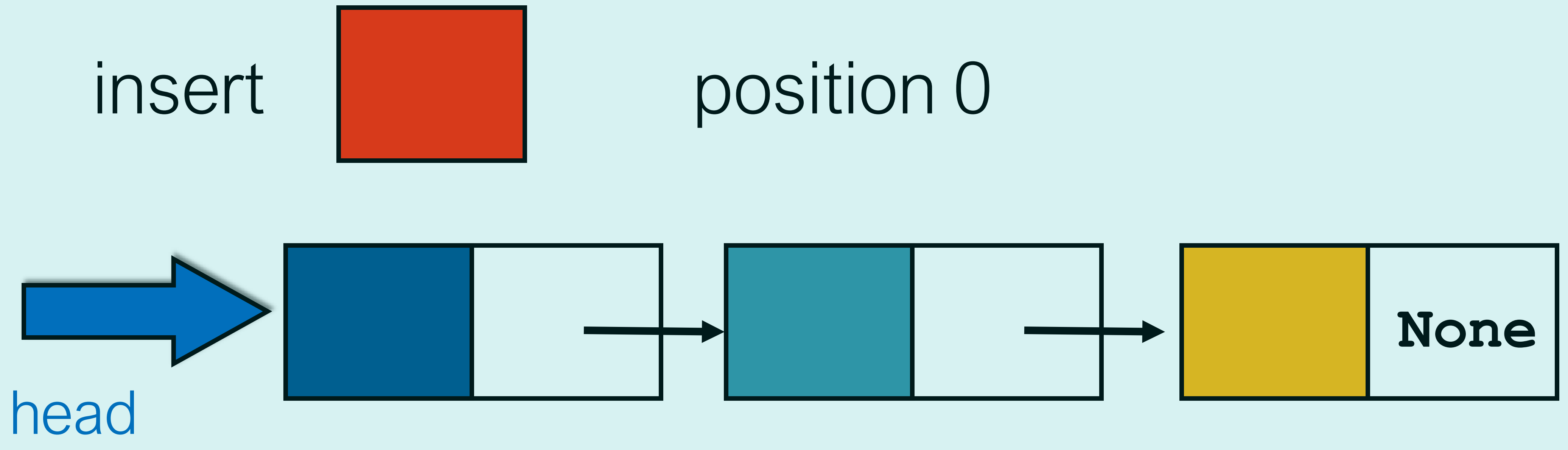
- check if **index** is within range
- set a variable **node**, pointing to Node referred by head
- set a **counter** to 0
- while **counter** is less than **index**
    - follow link to <u>next node</u>
    - increment **counter**
- return **node**

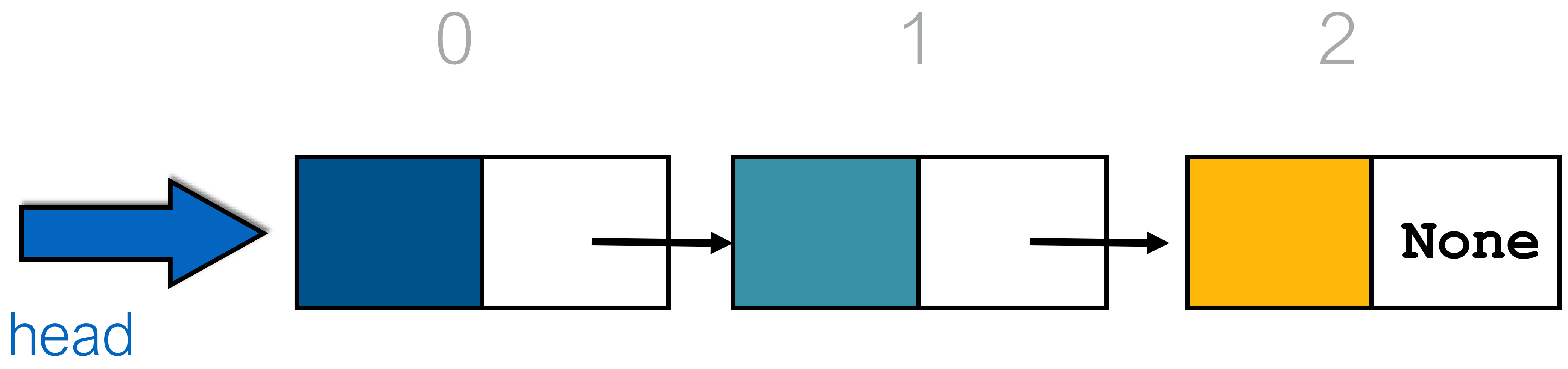
```python
def _get_node(self, index):
```

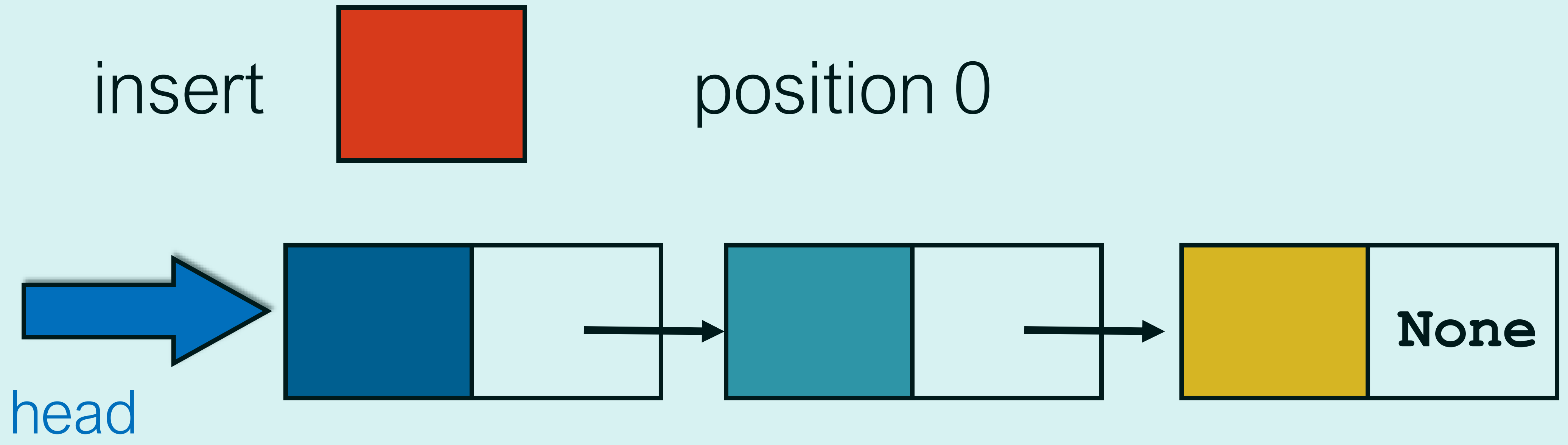
```python
def _get_node(self, index):
    assert 0 <= index < self.count, "Index out of bounds"
    node = self.head
    for _ in range(index):
        node = node.next
    return node
```

Only works if self.count updated consistently
Otherwise might try to access a next of None

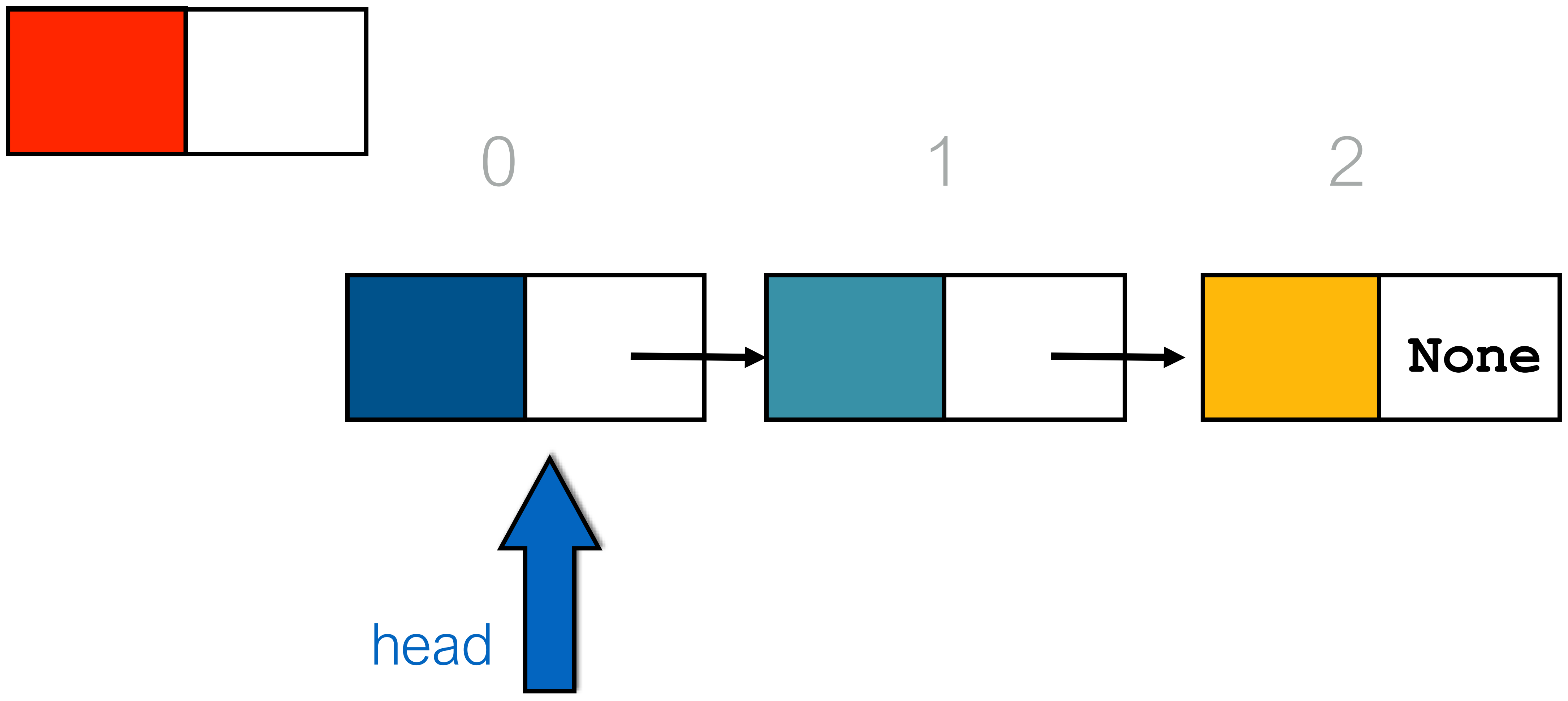
Insert


insert
position 0
head
None


0
1
2
head
None

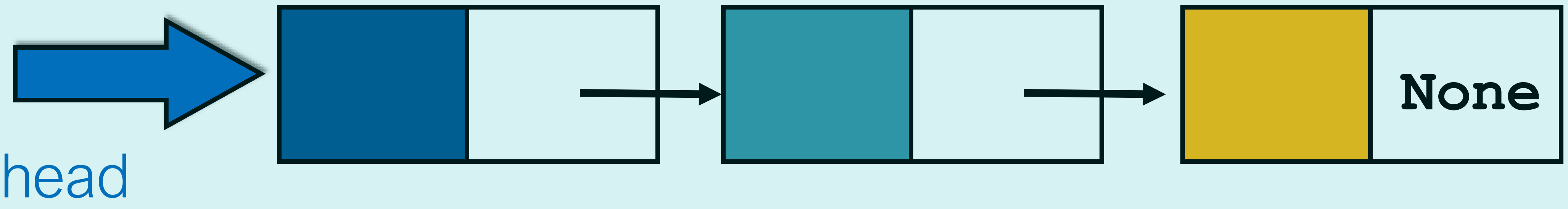


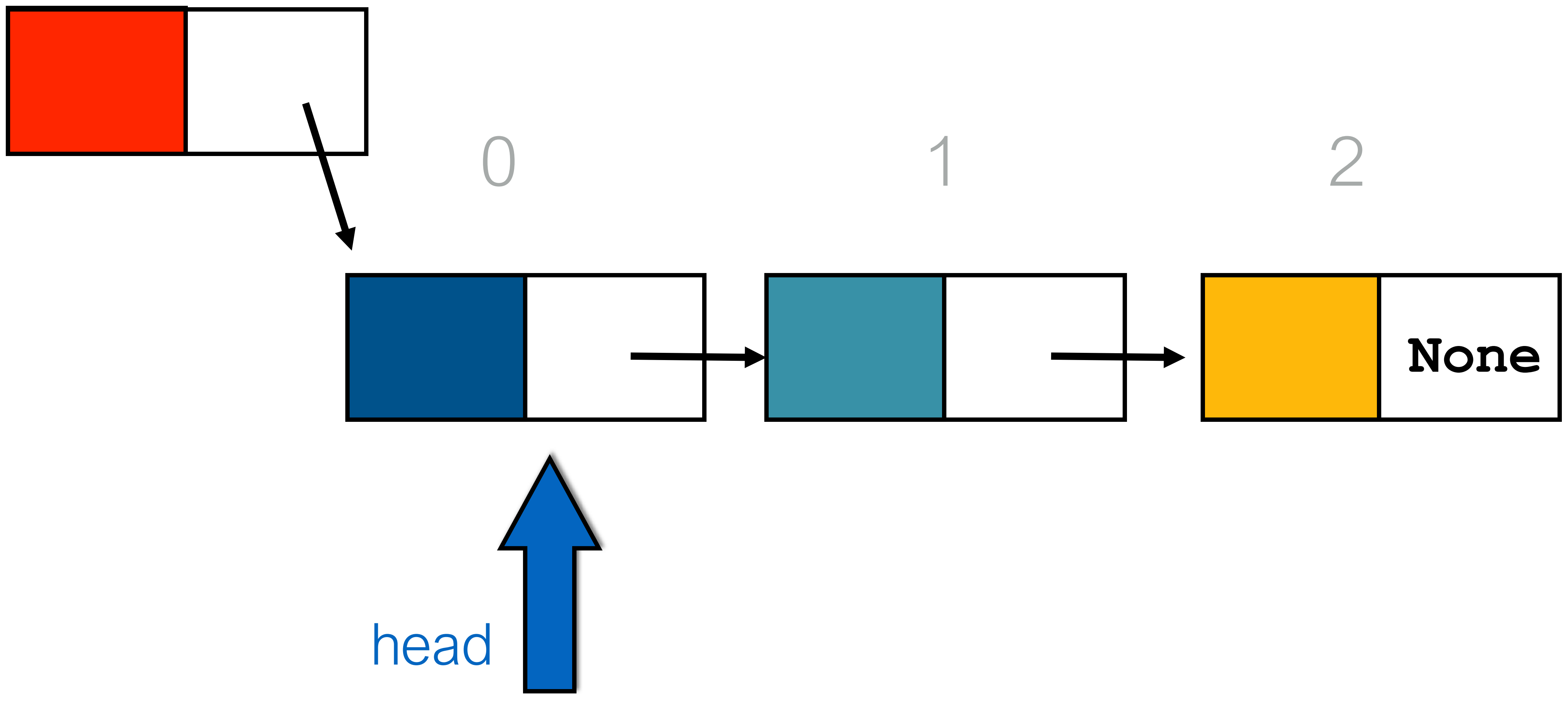
insert
position 0
head
None

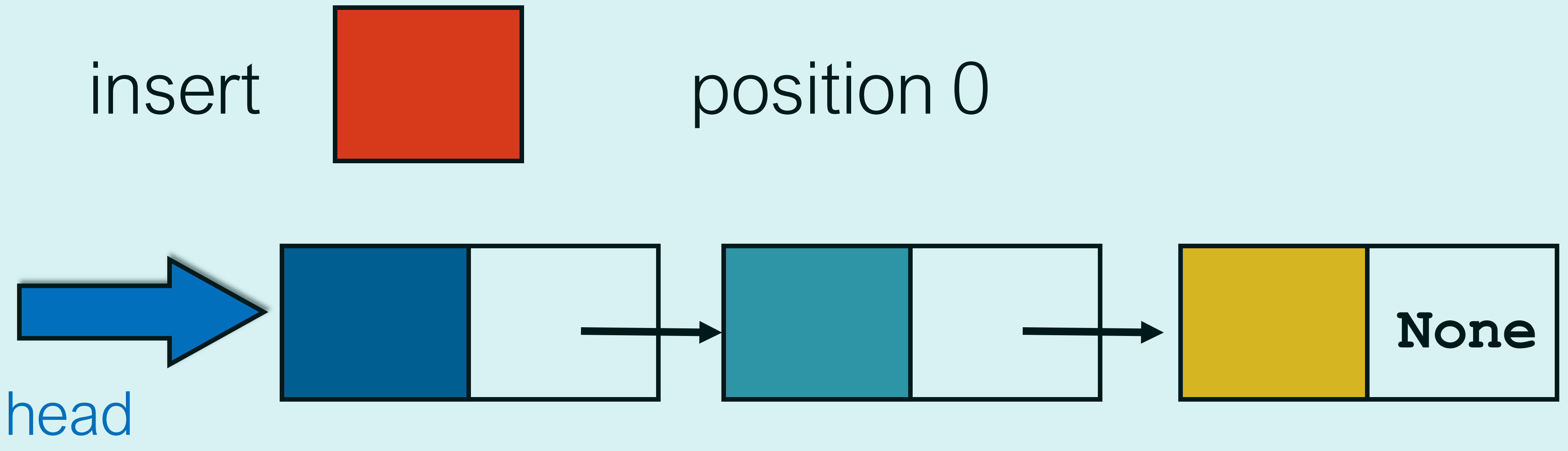


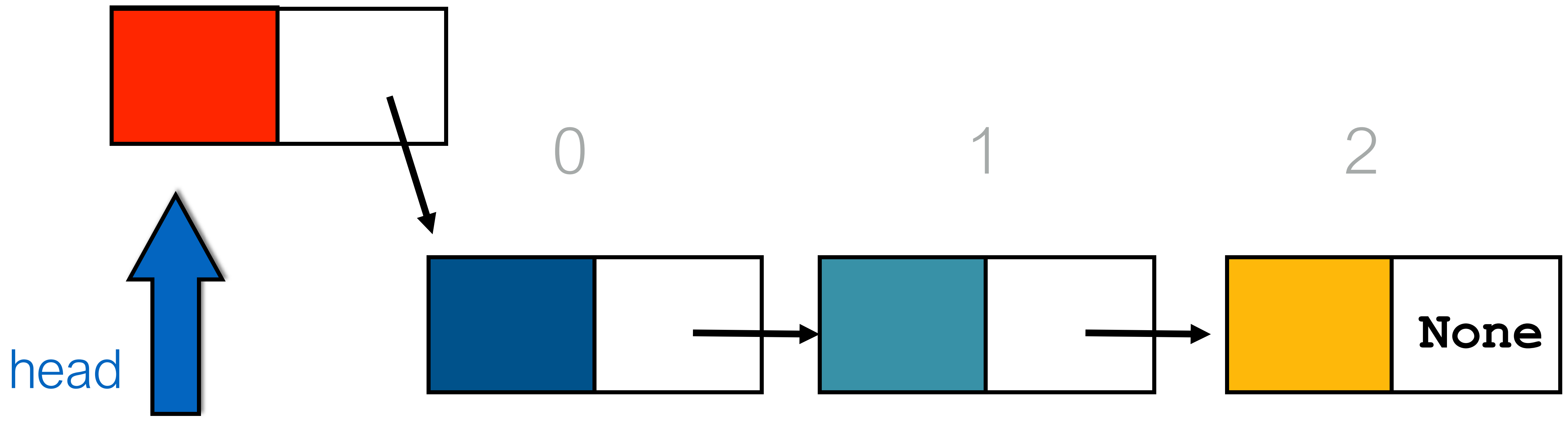
0
1
2
None
head

insert position 0





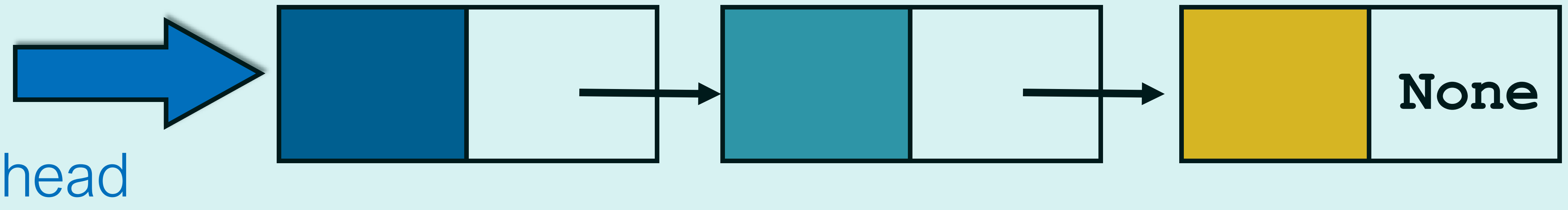


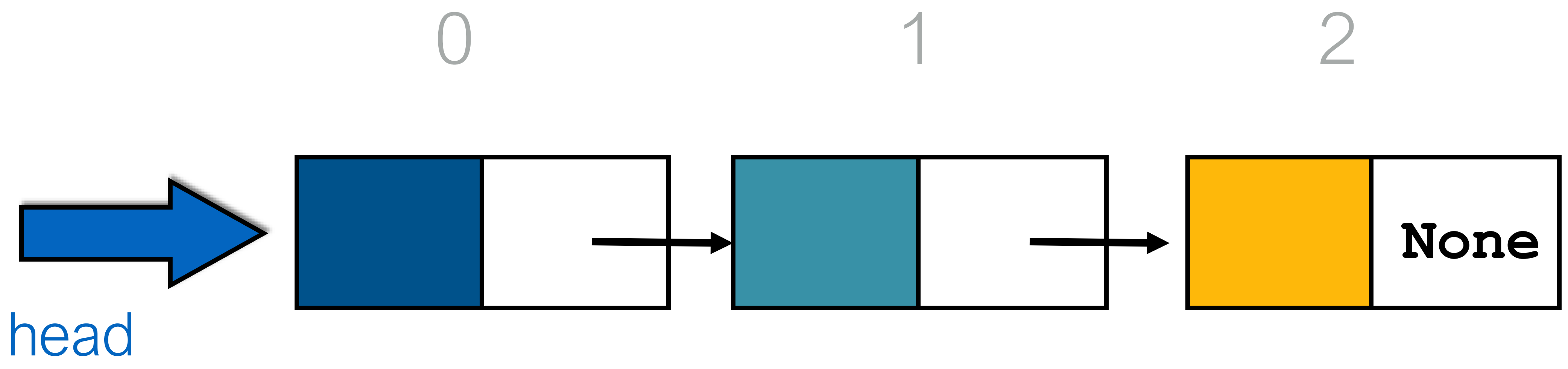


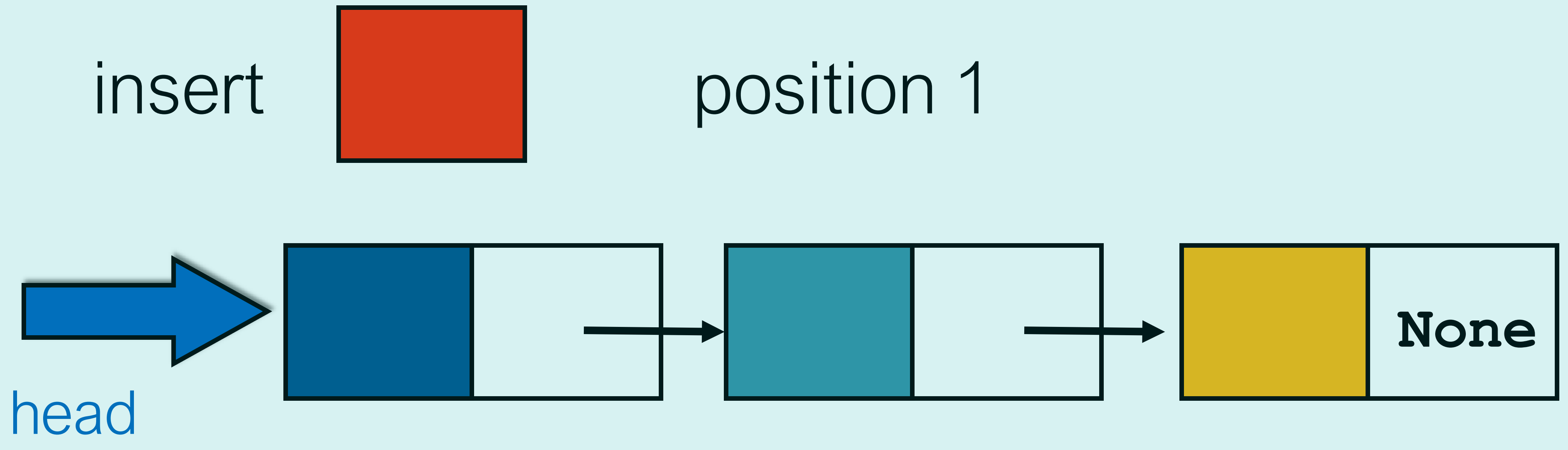
Very similar to **push** in a Stack, if <u>position is 0</u>

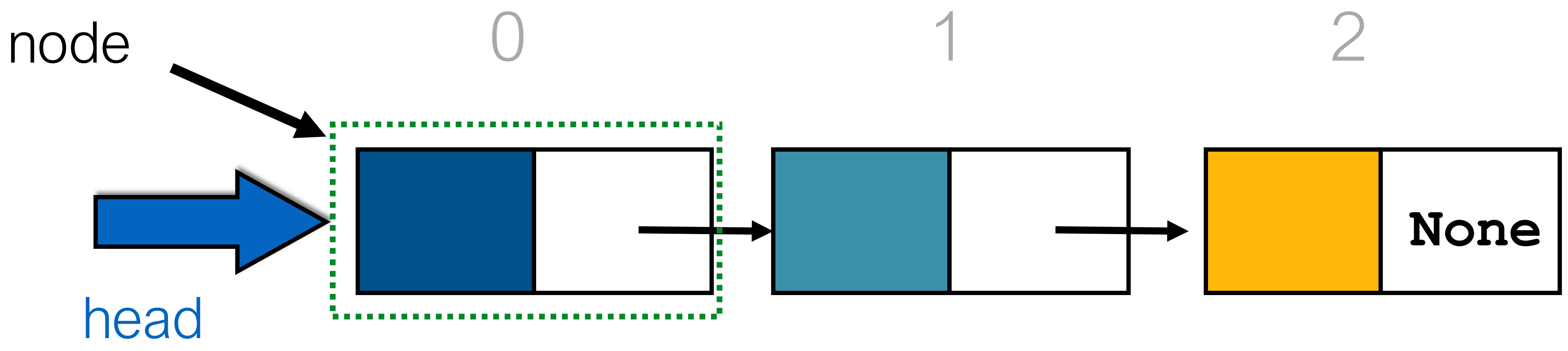
position $i > 0$

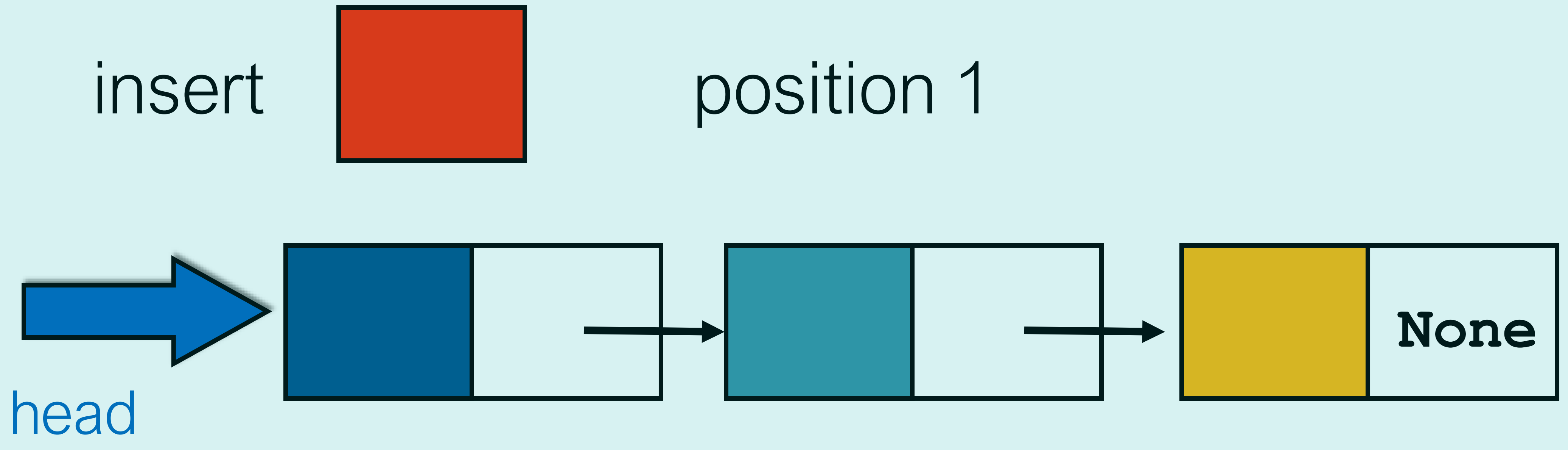
insert position 1

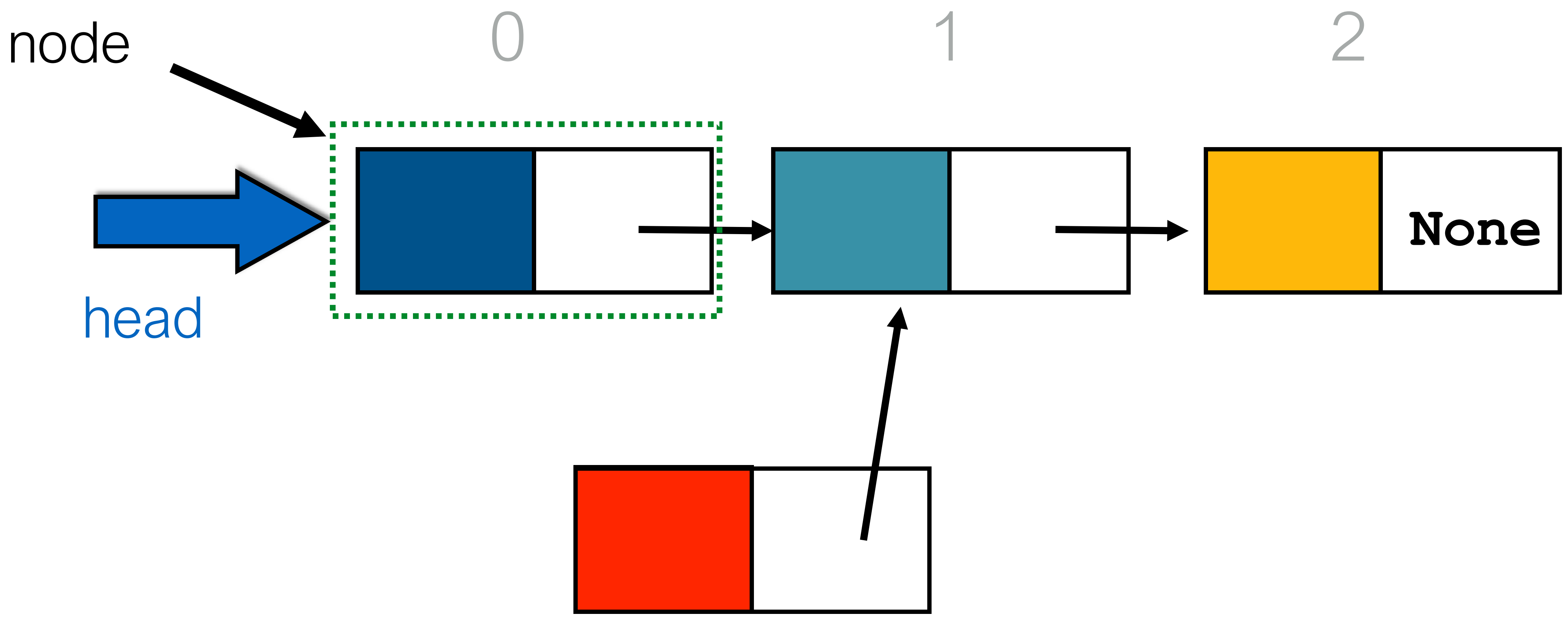


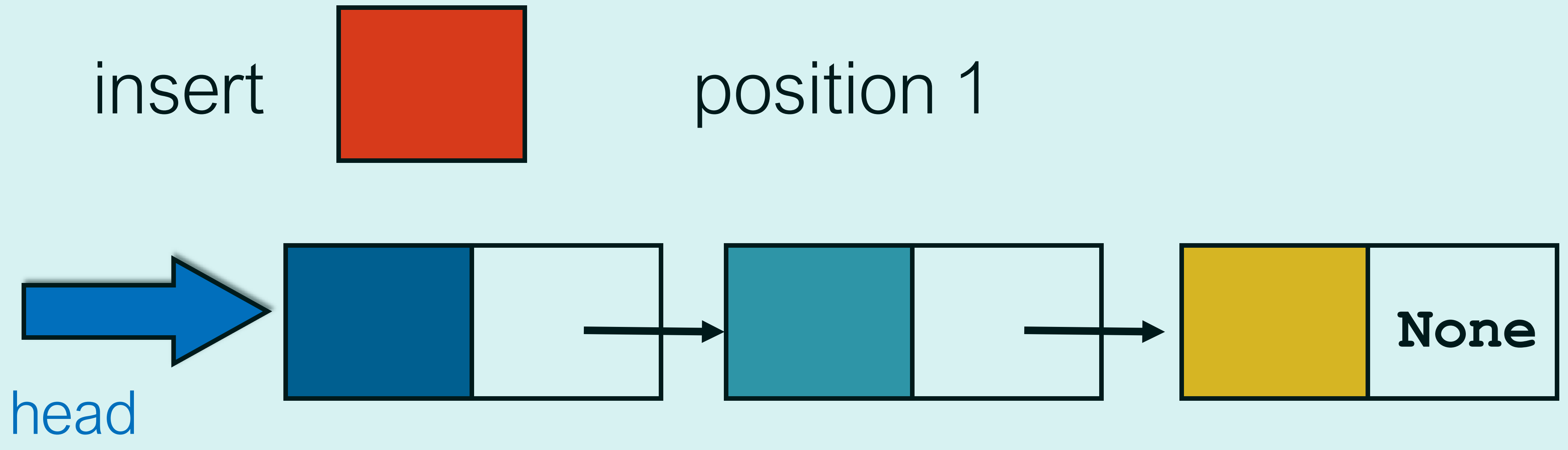


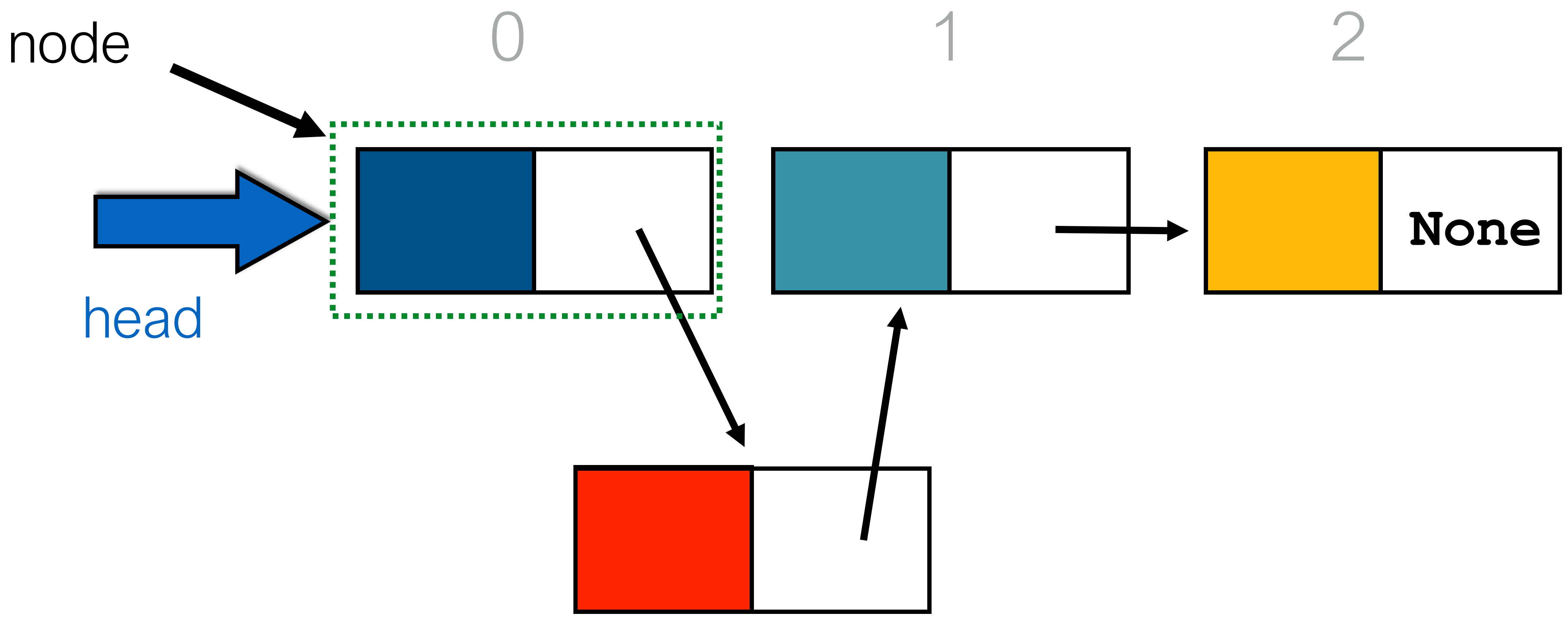

insert
position 1
None
head


node
0
1
2
None
head


insert
position 1
None
head


node
0
1
2
None
head

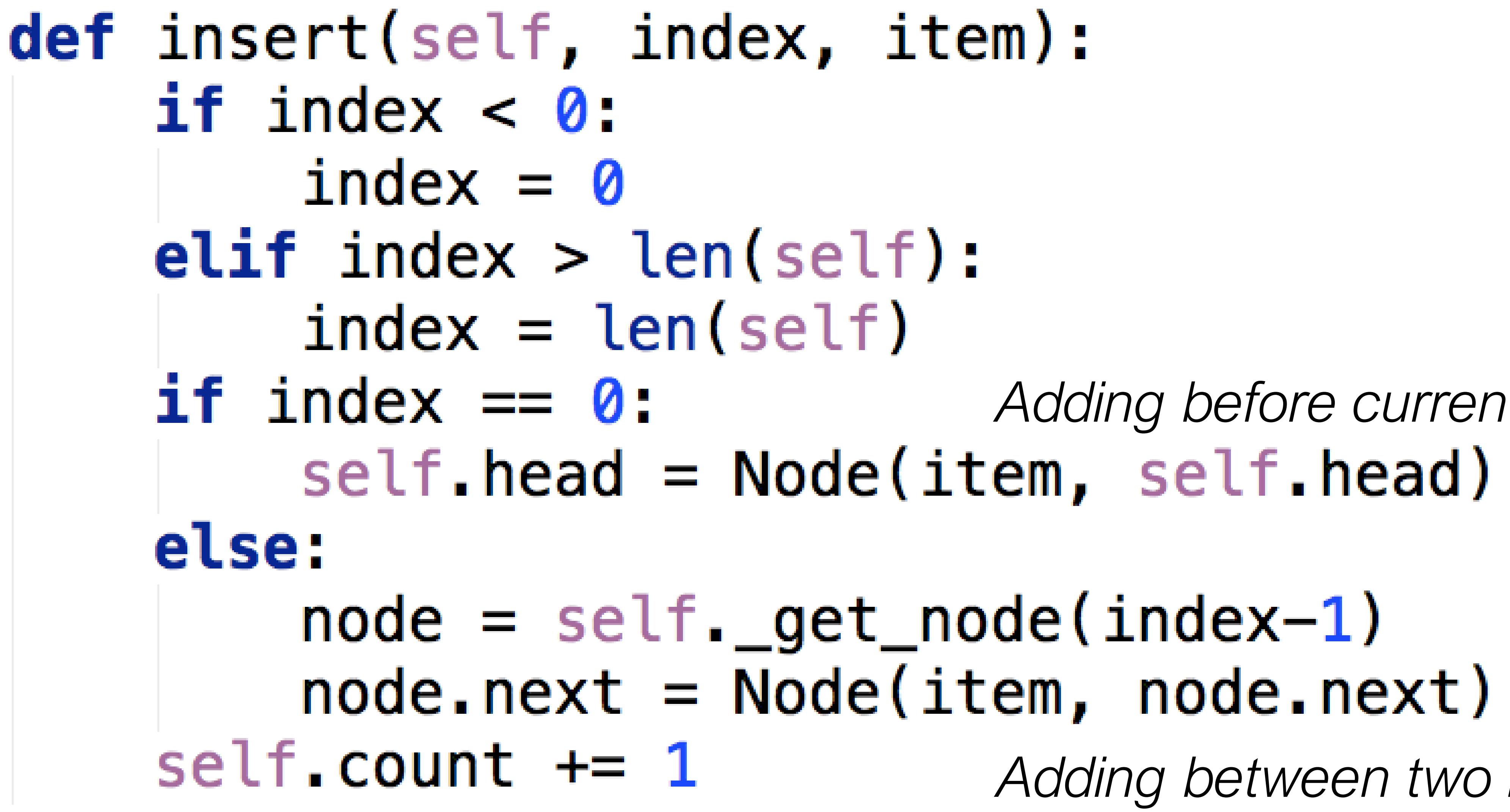

insert
position 1
None
head


node
0
1
2
None
head

insert

```
def insert(self, index, item):
```

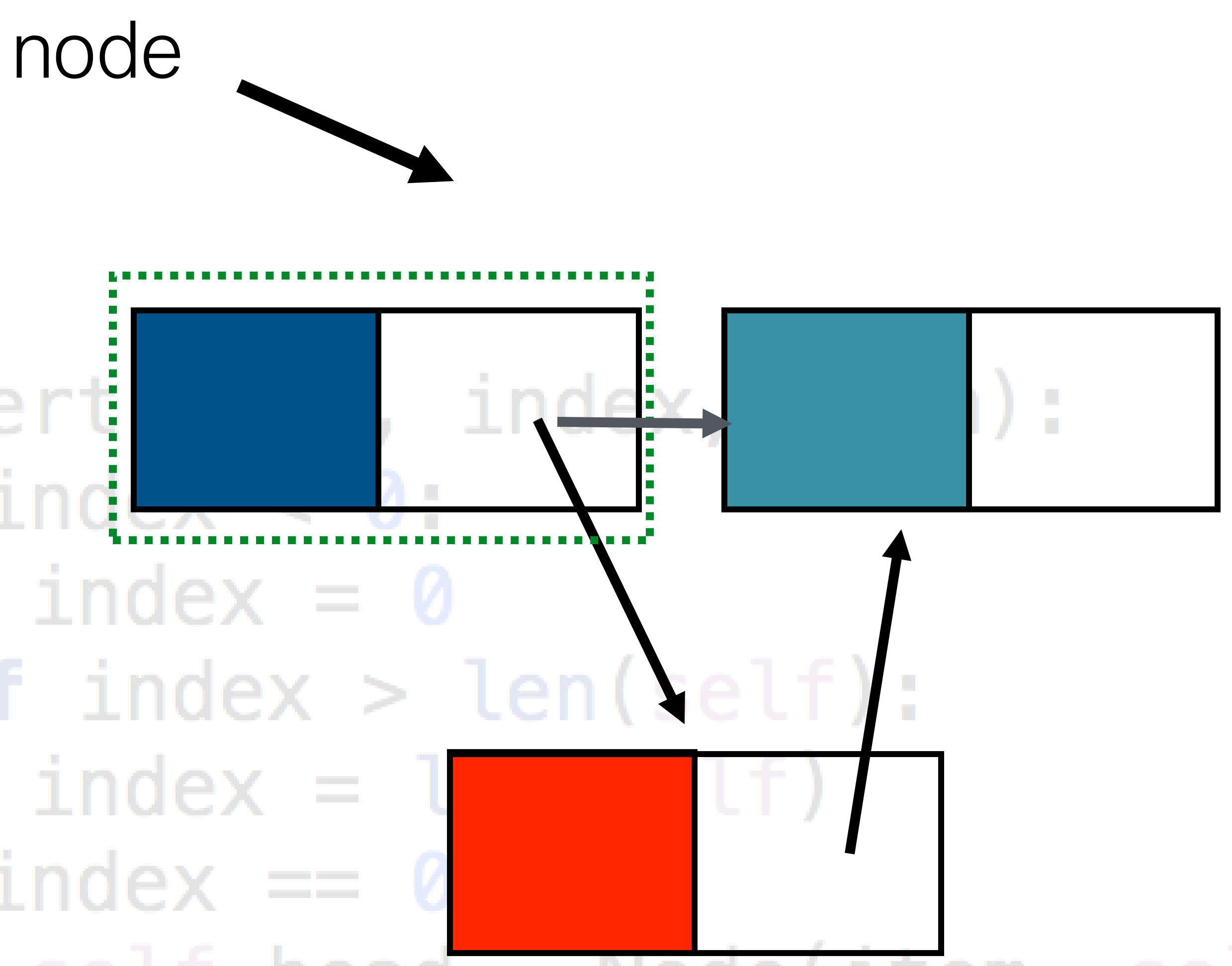
insert



```
def insert(self, index, item):
    if index < 0:
        index = 0
    elif index > len(self):
        index = len(self)
    if index == 0:
        self.head = Node(item, self.head)
    else:
        node = self._get_node(index-1)
        node.next = Node(item, node.next)
    self.count += 1
```

Adding before current head

Adding between two nodes



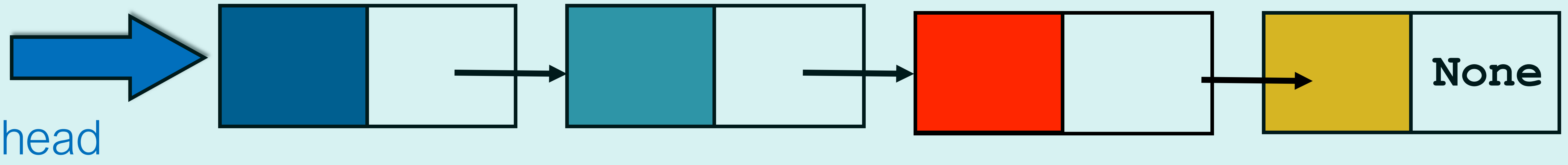
```
def insert(self, index, item):
    if index < 0:
        index = 0
    elif index > len(self):
        index = len(self)
    if index == 0:
        self.head = Node(item, self.head)
    else:
        node = self._get_node(index-1)
        node.next = Node(item, node.next)
    self.count += 1
```

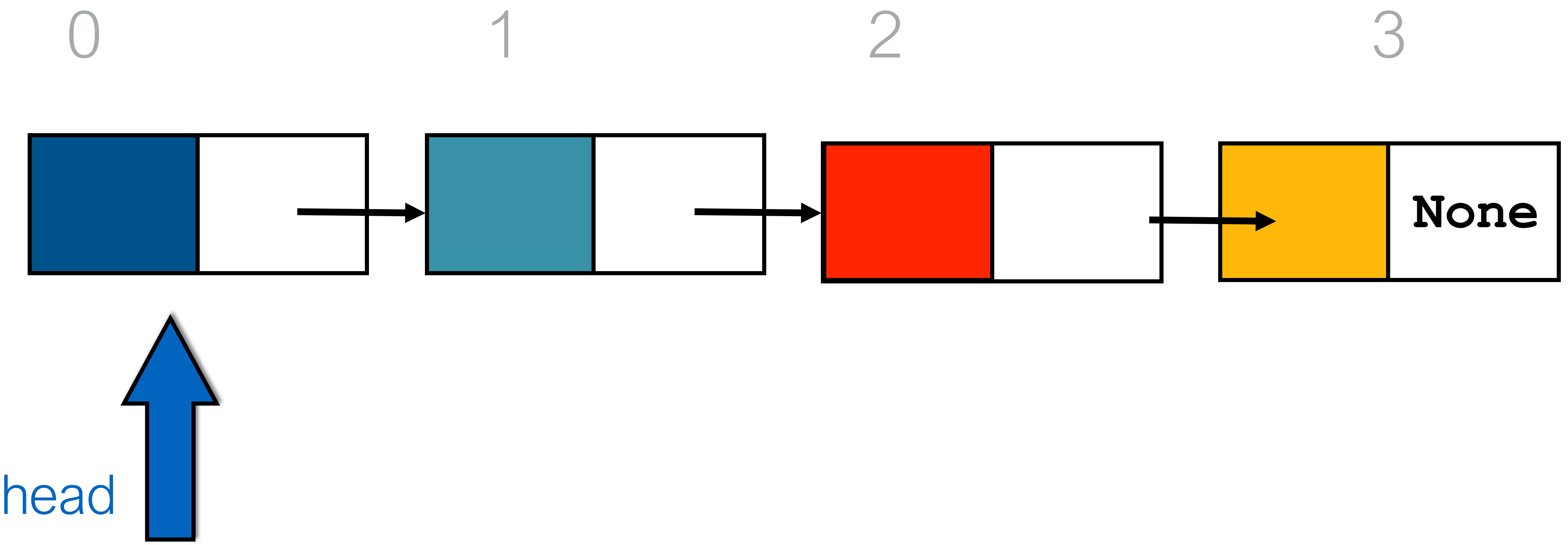
insert

```
def insert(self, index, item):
    if index < 0:
        index = 0
    elif index > len(self):
        index = len(self)
    if index == 0:
        self.head = Node(item, self.head)
    else:
        node = self._get_node(index-1)
        node.next = Node(item, node.next)
    self.count += 1
```

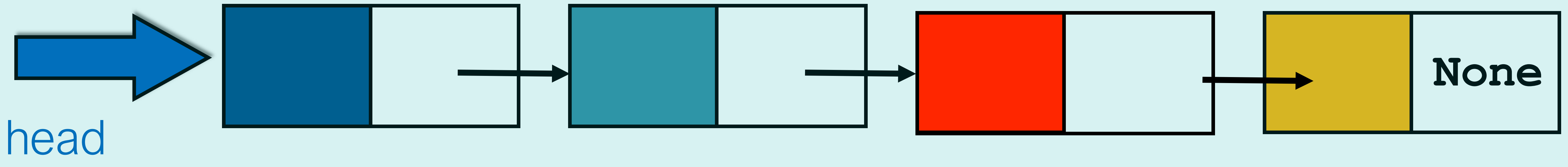
delete

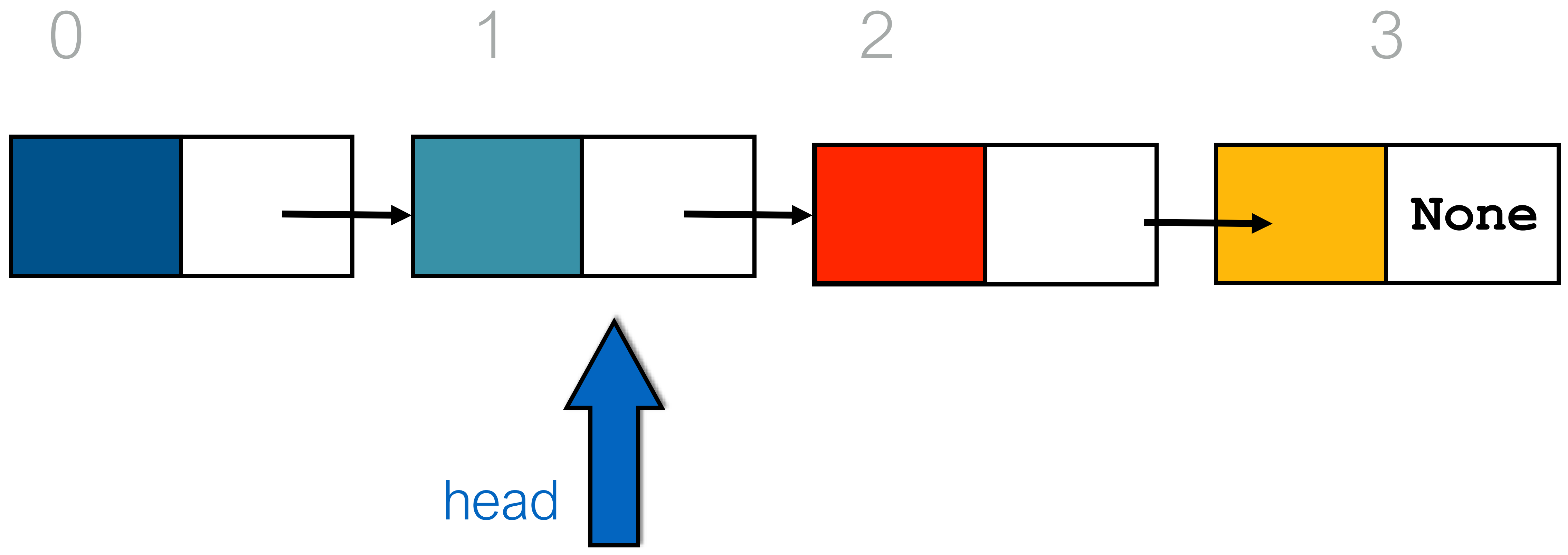
delete item in position 0



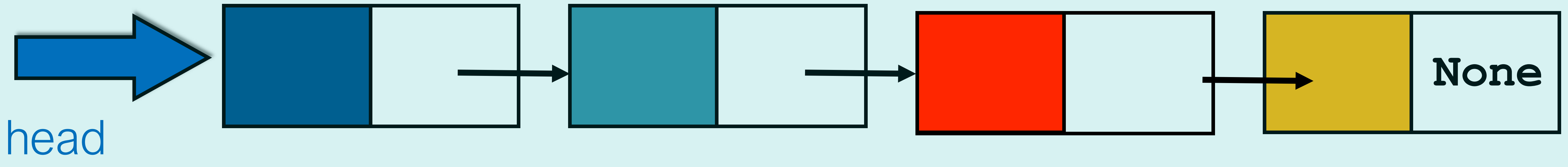


delete item in position 0

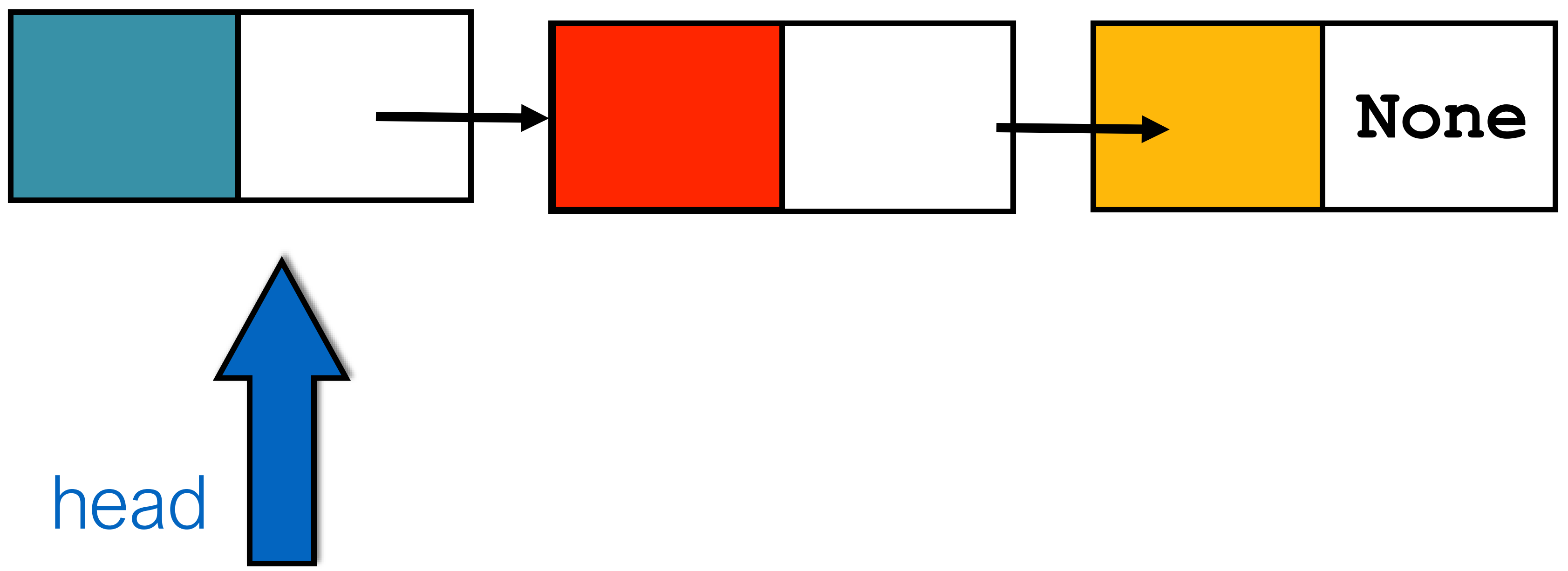




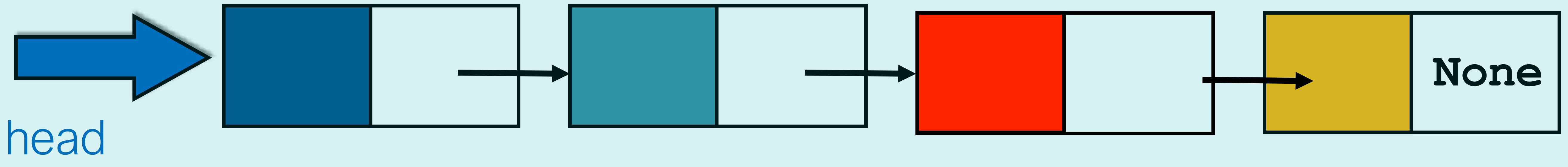
delete item in position 0

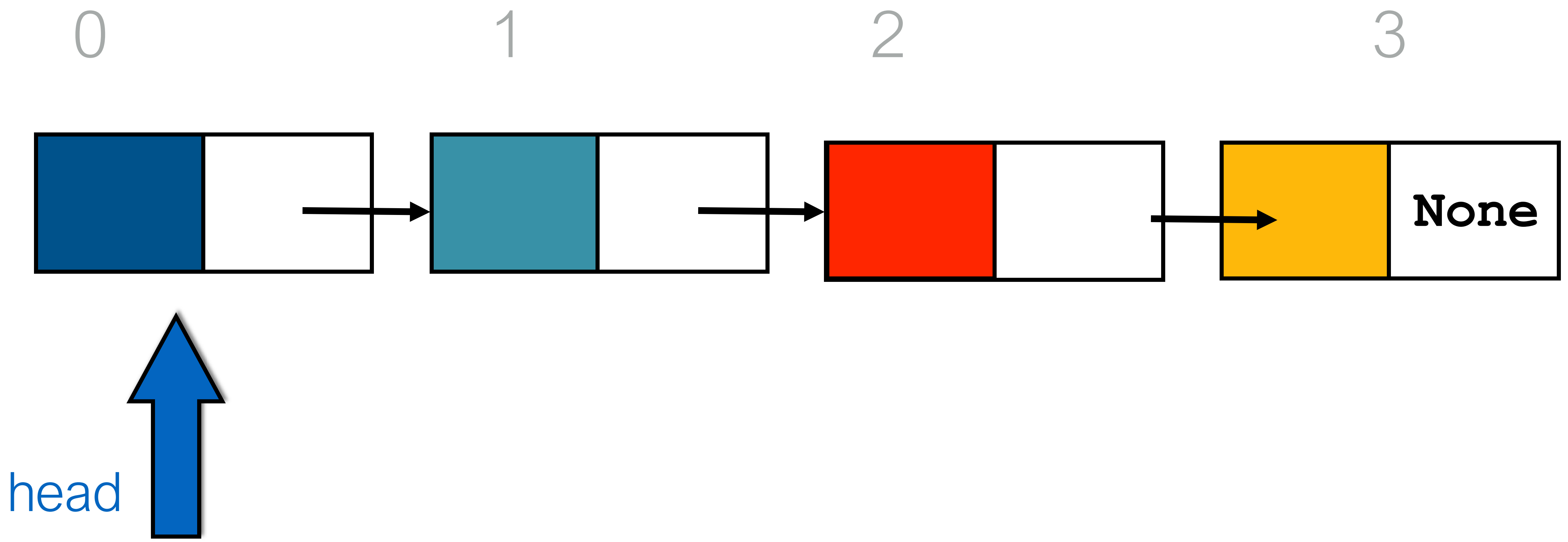


Just like pop.

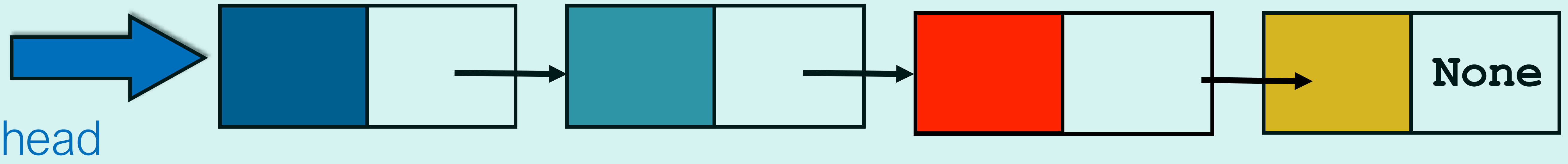


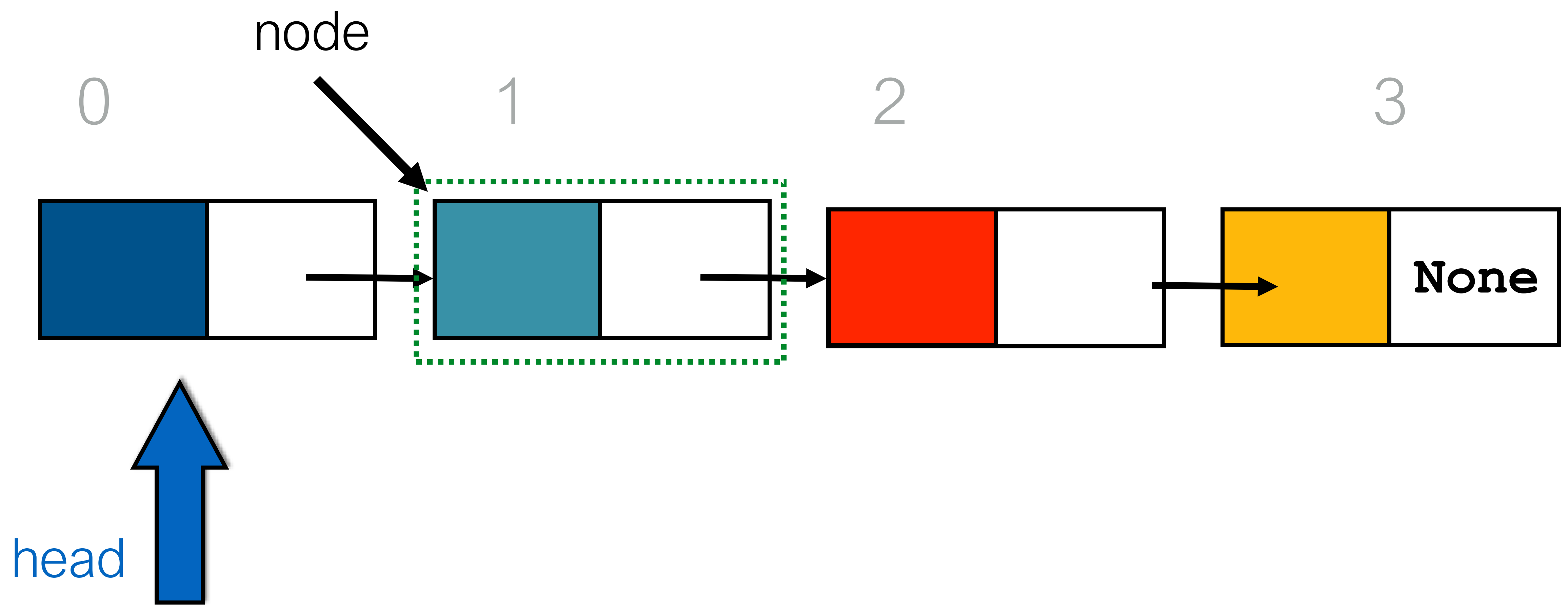
delete item in position 2



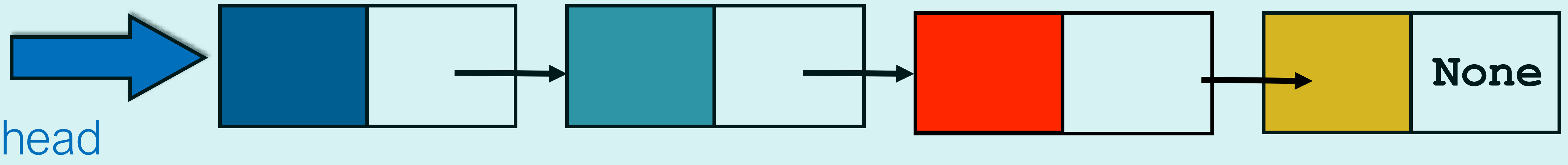


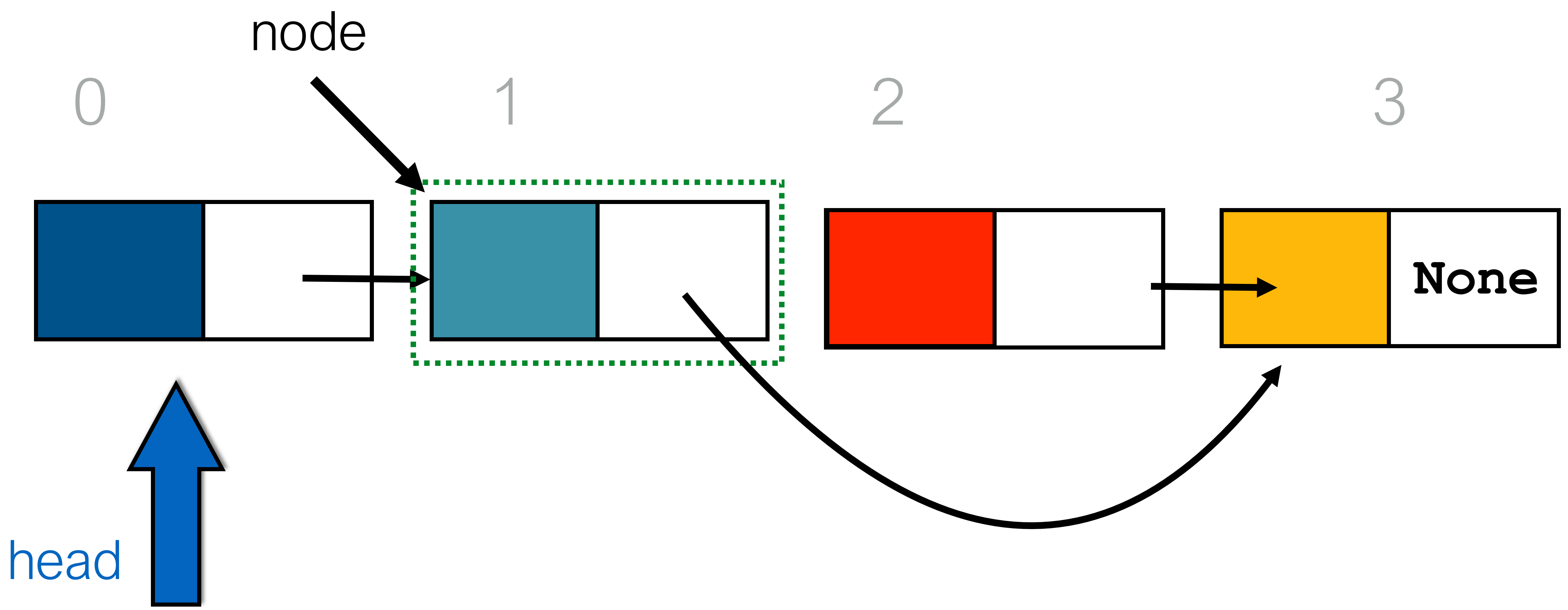
delete item in position 2



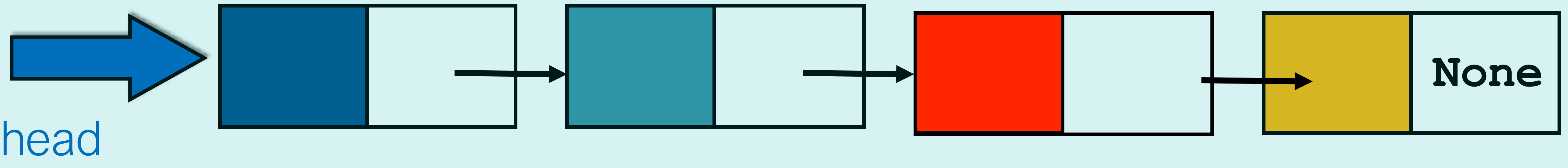


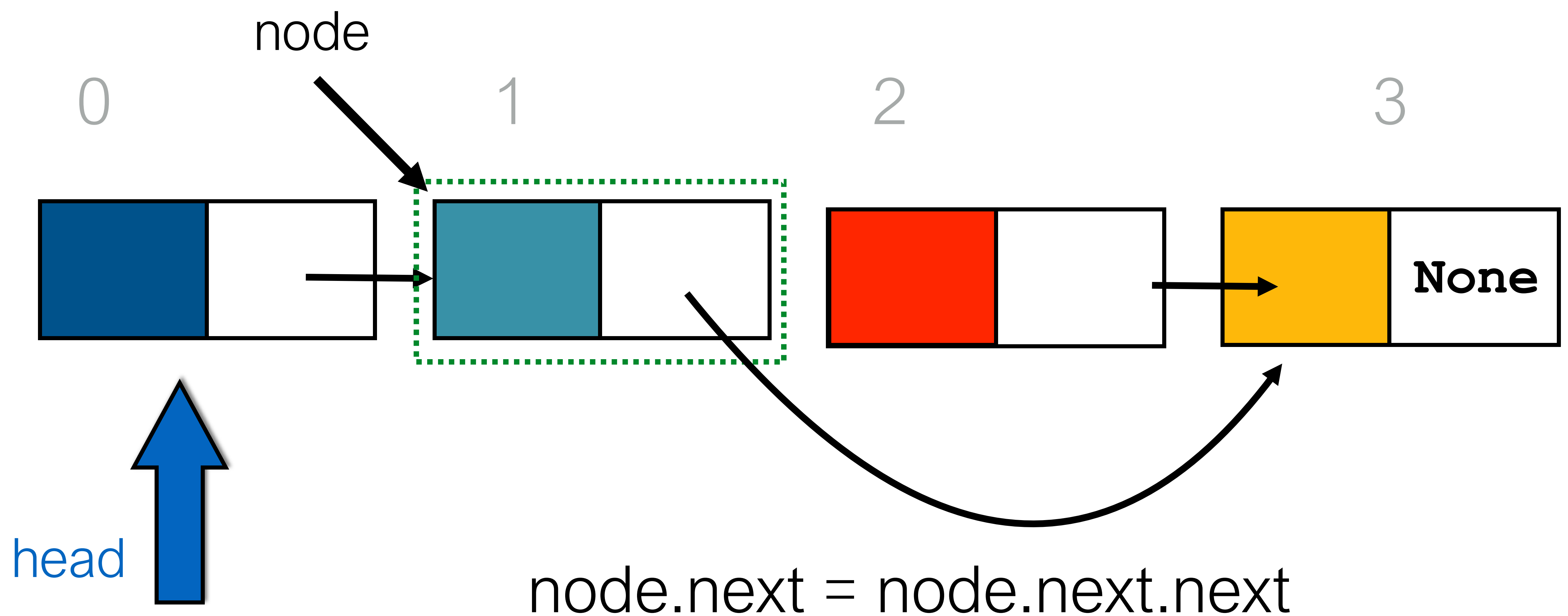
delete item in position 2





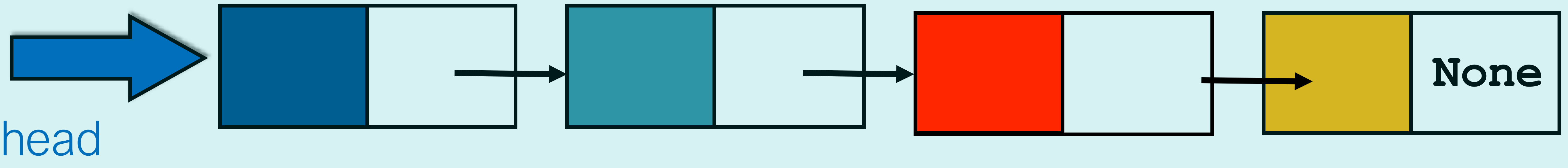
delete item in position 2

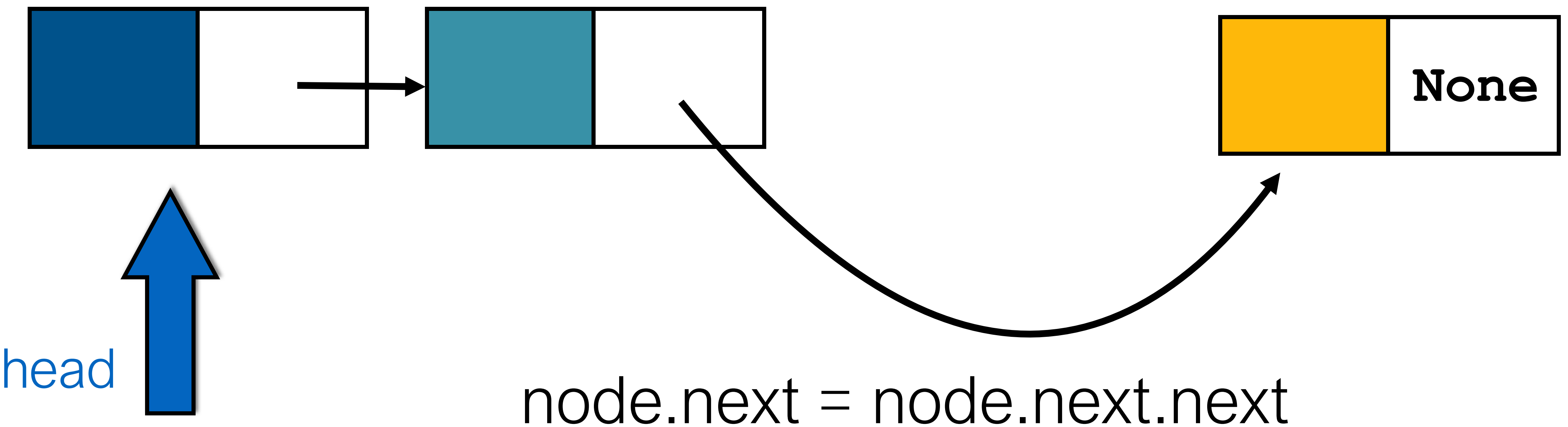




node.next = node.next.next

delete item in position 2



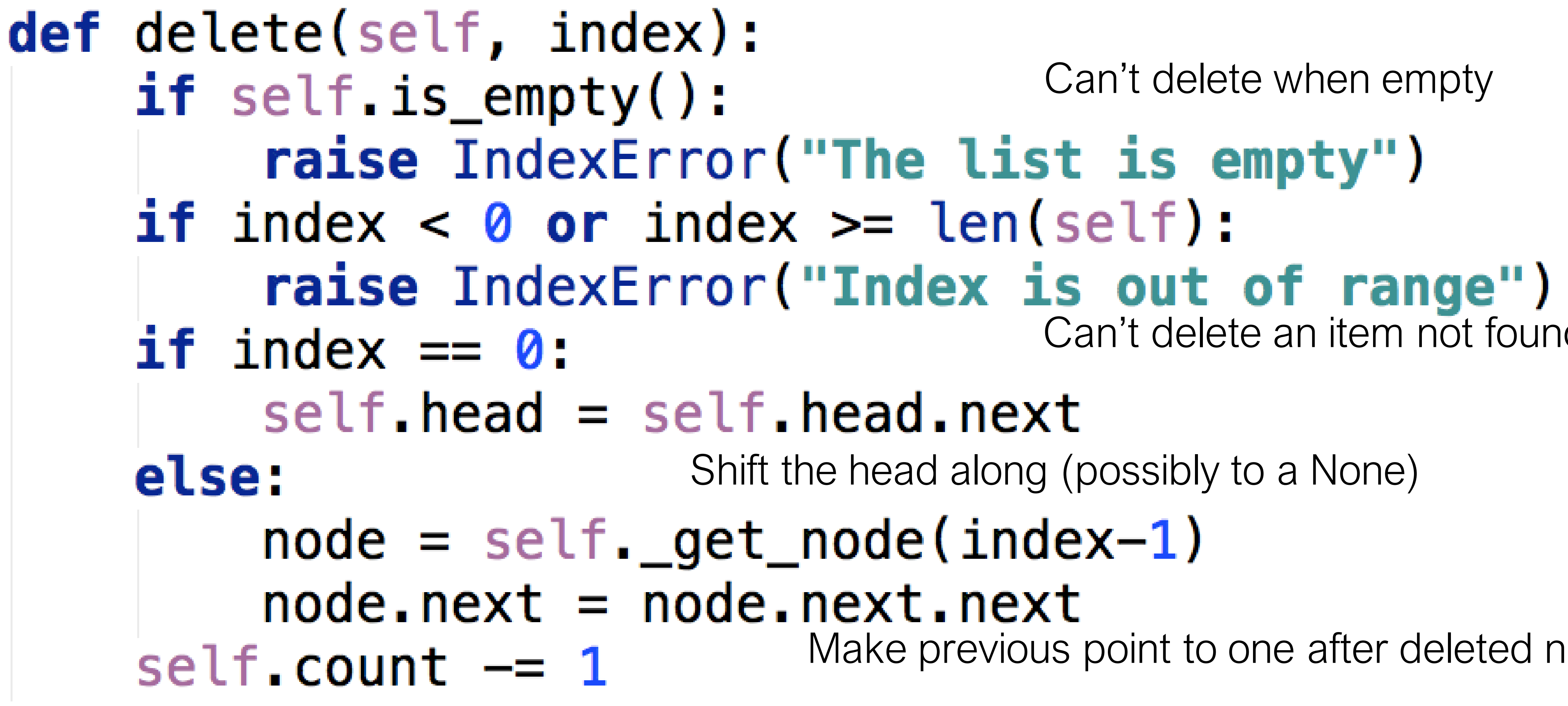


node.next = node.next.next

Boundary cases?

Empty List or Index out of Bounds

```python
def delete(self, index):
```



```python
def delete(self, index):
    if self.is_empty():
        raise IndexError("The list is empty")
    if index < 0 or index >= len(self):
        raise IndexError("Index is out of range")
    if index == 0:
        self.head = self.head.next
    else:
        node = self._get_node(index-1)
        node.next = node.next.next
    self.count -= 1
```

Can’t delete when empty

Can’t delete an item not found

Shift the head along (possibly to a None)

Make previous point to one after deleted node

Comparison

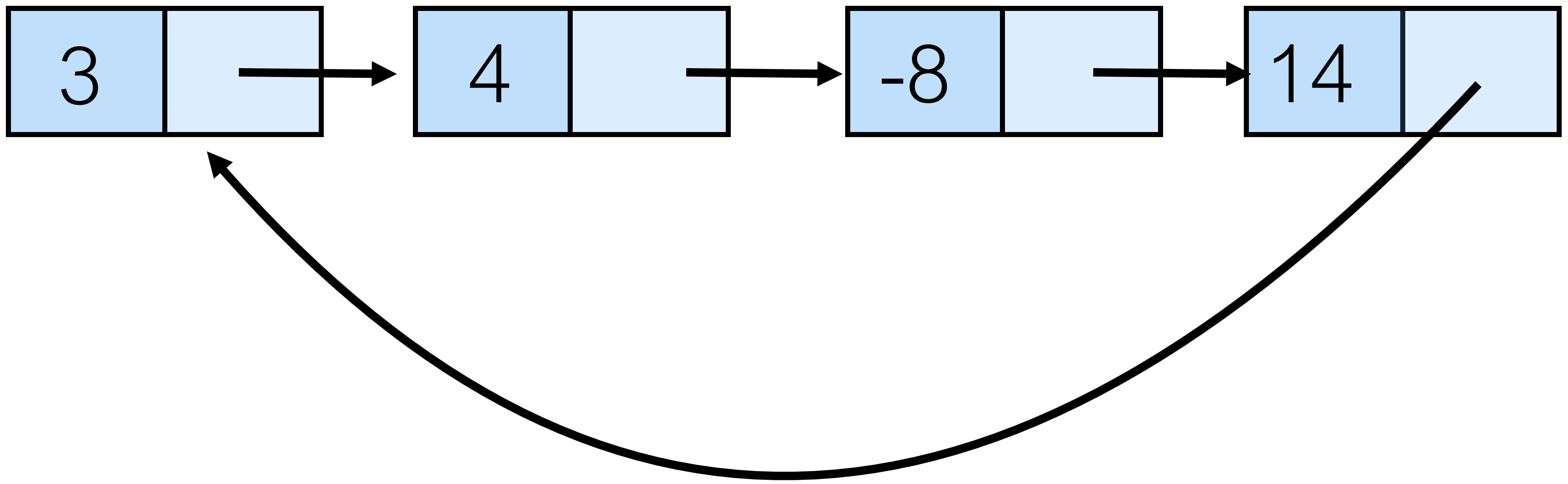
Linked Storage

- Unknown list size.
- Flexibility is needed: lots of insertions and deletions.

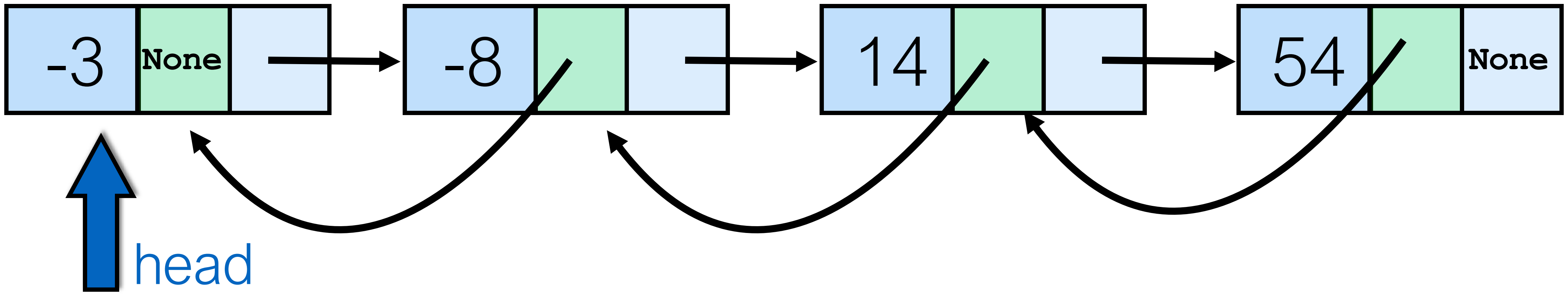
Contiguous Storage

- Known list size.
- Few insertions and deletions.
- <u>Random access</u>

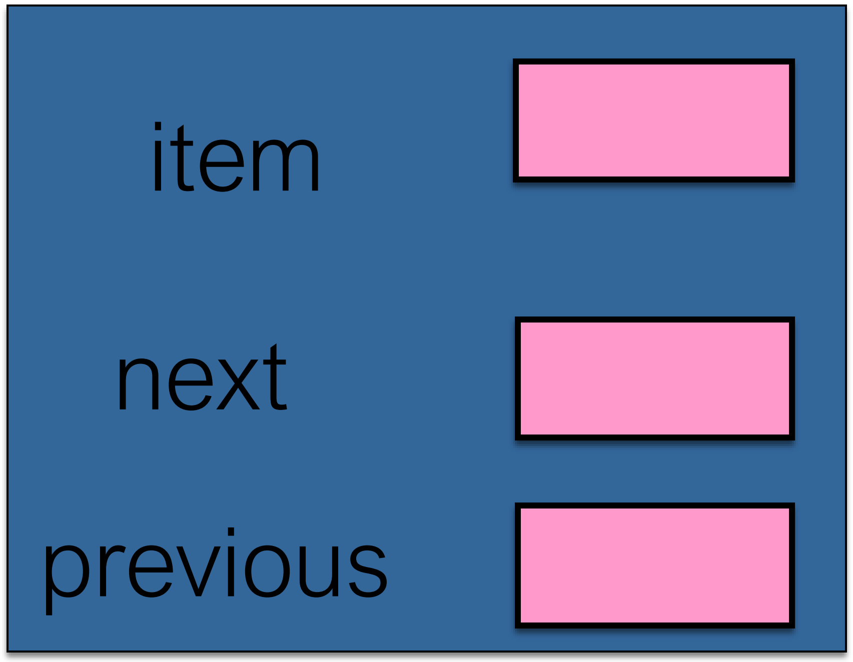
Circular linked list



Double linked list



Node



Summary

- Seen how to implement a Linked List
- In particular
    - Inserting an item
    - Deleting an item