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

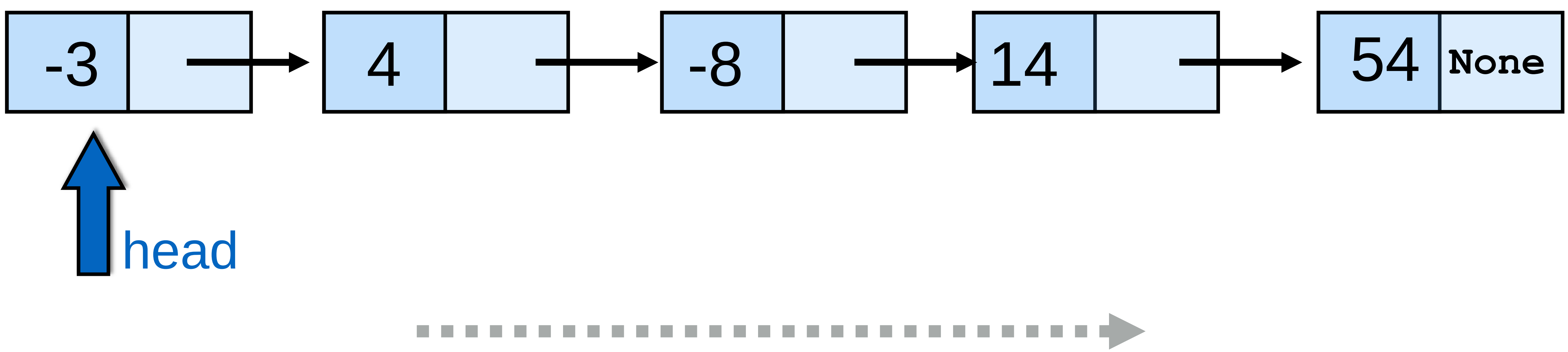
Lecture 28
Iteration vs Recursion

FIT 1008
Introduction to Computer Science

Operation	Class Method
str(obj)	__str__(self)
len(obj)	__len__(self)
item in obj	__contains__(self,item)
y = obj[ndx]	__getitem__(self,ndx)
obj[ndx] = value	__setitem__(self,ndx,value)
obj == rhs	__eq__(self,rhs)
obj < rhs	__lt__(self,rhs)
…	
obj + rhs	__add__(self,rhs)
...	

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

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

?



count from head to access elements

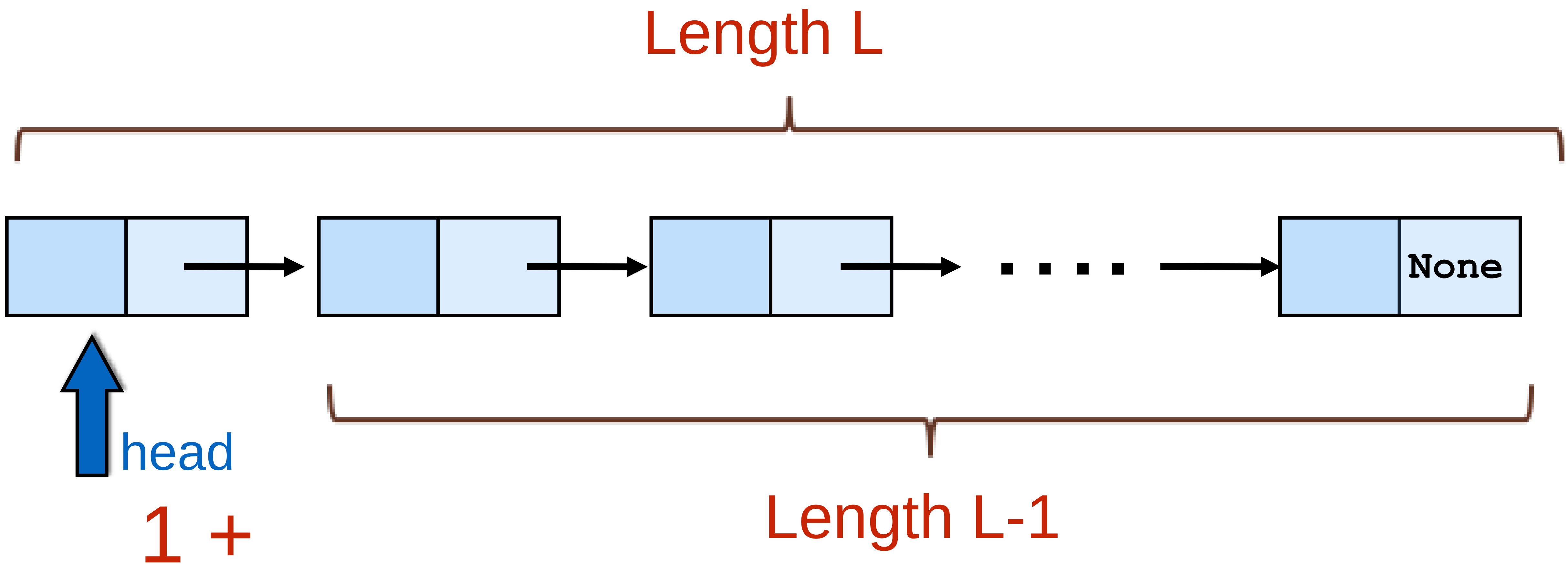
```python
class List:
    def __init__(self):
        self.head = None

    def __len__(self):
```

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

    def __len__(self):
        current = self.head
        count = 0
        while current is not None:
            current = current.next
            count += 1
        return count
```

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

    def __len__(self):
        current = self.head
        count = 0
        while current is not None:
            current = current.next
            count += 1
        return count
```

Complexity: O(n) where n is the size of the list.

What about recursively?



Convergence: Call recursion with L-1. Use variable *current*.

Base case: Empty? Size of empty list is 0.

Combining solutions: Add up result of recursive call +

```python
def __len__(self):
```

```python
def __len__(self):
    if self.head is None:
        return 0
    else:
        return 1  + self.__len__(self.head.next)
```

https://c3.thejournal.ie/media/2013/11/awkward-moment-sign1-230x150.jpg


```
def __len__(self):
    if self.head is None:
        return 0
    else:
        return 1  + self.__len__(self.head.next)
```

```
TypeError: __len__() takes 1 positional argument but 2 were given
```

```python
def length(self, current):
```

```python
def _length(self, current):
    if self.current is None: # base case
        return 0
    else:
        return 1  + self._length(current.next)

def __len__(self):
    return self._length(self.head)
```

Base case

Convergence

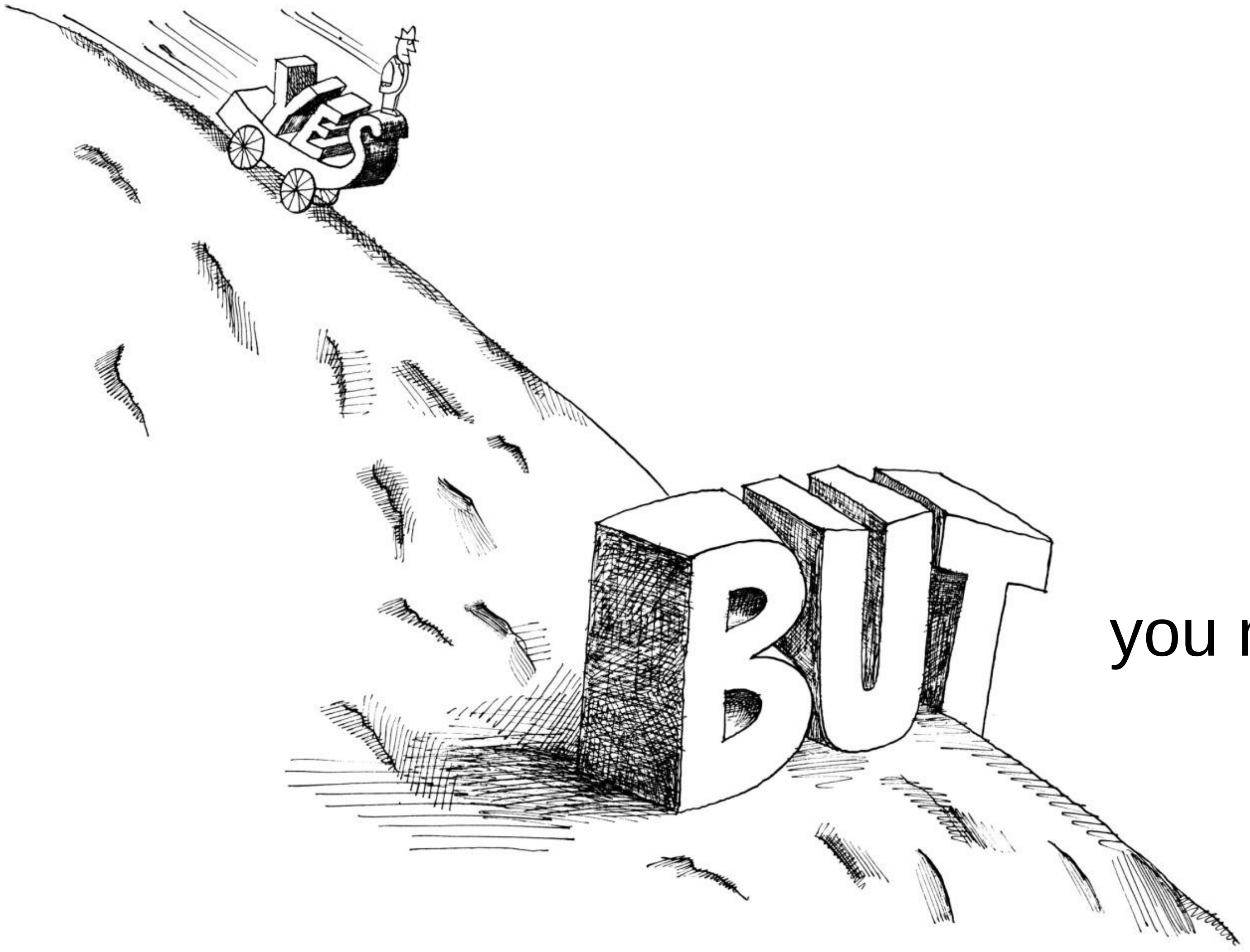
Combination

Auxiliary method sets up the initial parameters

Recursion vs Iteration

- Can every **iterative function** be implemented using **recursion**?
  Yes, it is straightforward.

- Can every **recursive function** be implemented using **iteration**?

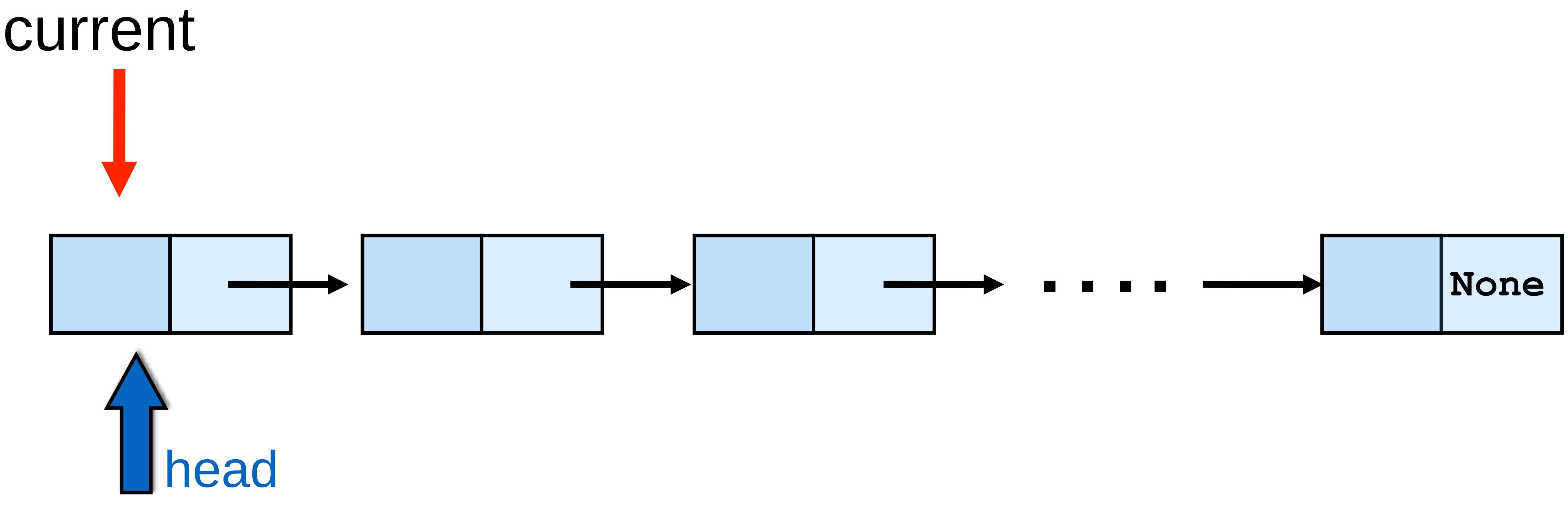


you might also need to store past results.

Lovingly stolen from:
https://66.media.tumblr.com/3f1bb6a13c96f90cb332b385a6363b2e/tumblr_nkaavk9owg1qz6f4bo2_1280.jpg

Operation	Class Method
str(obj)	__str__(self)
len(obj)	__len__(self)
item in obj	__contains__(self,item)
y = obj[ndx]	__getitem__(self,ndx)
obj[ndx] = value	__setitem__(self,ndx,value)
obj == rhs	__eq__(self,rhs)
obj < rhs	__lt__(self,rhs)
…	
obj + rhs	__add__(self,rhs)
...	

__contains__



__contains__

```python
def __contains__(self, item):
```

__contains__

```
def __contains__(self, item):
    current = self.head
    while current is not None:
        if current.item == item:
            return True
        current = current.next
    return False
```

Complexity: Worst case - O(n) where n is the size of the list.

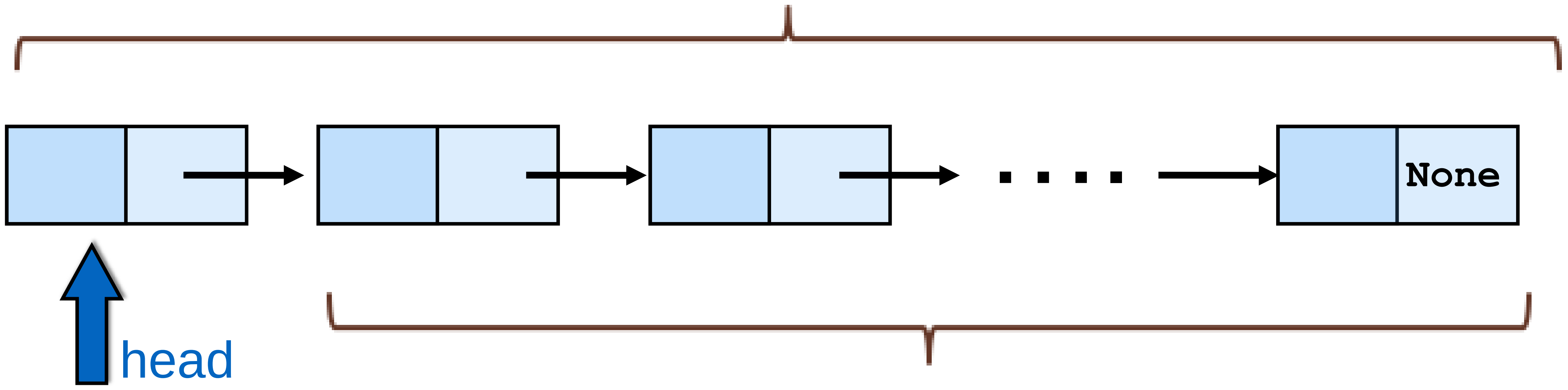
__contains__

```
def __contains__(self, item):
    current = self.head
    while current is not None:
        if current.item == item:
            return True
        current = current.next
    return False
```

- **Best case** complexity when found first: **O(1)***CompEq where CompEq is the complexity of == (or __eq__)
- **Worst case** when not found: **O(n)***CompEq where n is the length of the list.

What about recursively?

__contains__(6) in L



6 ==head.item **or** __contains__(6) in L-1

Convergence: Call recursion with L-1. Use variable *current*.

Base case: Empty or Element Found. We need both.

Combining solutions: it's in the head **or** in the remaining list.

```python
def __contains__(self, item):
```

Auxiliary method sets up the initial parameters

```
def __contains__(self, item):
    return self._contains_aux(self.head, item)

def _contains_aux(self, current, item):
    if current is None:
        return False
    return current.head == item or self._contains_aux(current.next, item)
```

Convergence

Base case

Combination

Alternative coding

```
def __contains__(self, item):
    return self._contains_aux(self.head, item)
```

The Simpsons… but I stole the picture from here: https://i.imgur.com/KQnegLL.jpg

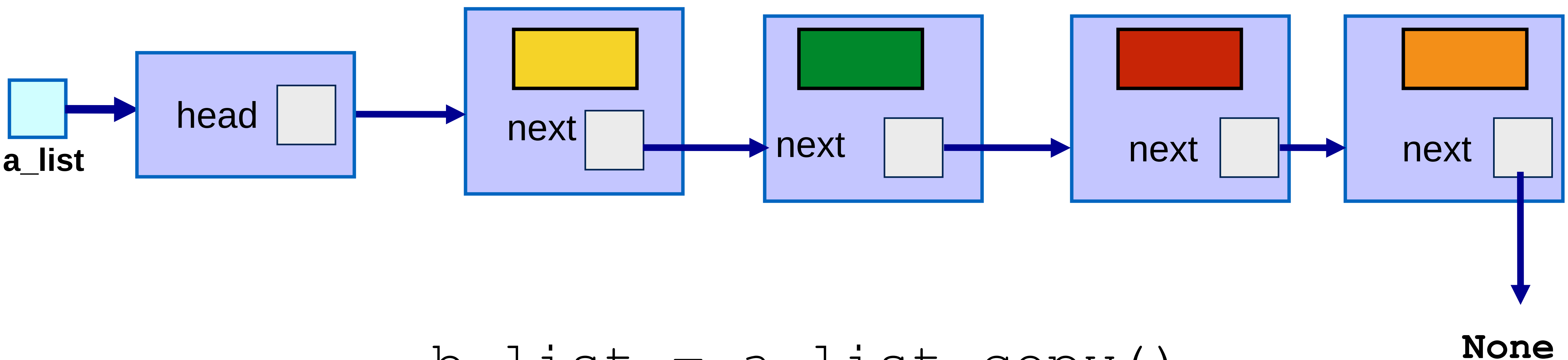
Alternative coding

```
def __contains__(self, item):
    return self._contains_aux(self.head, item)

def _contains_aux(self, current, item):
    if current is None:
        return False
    elif current.head == item:
        return True
    else:
        return self._contains_aux(current.next, item)
```

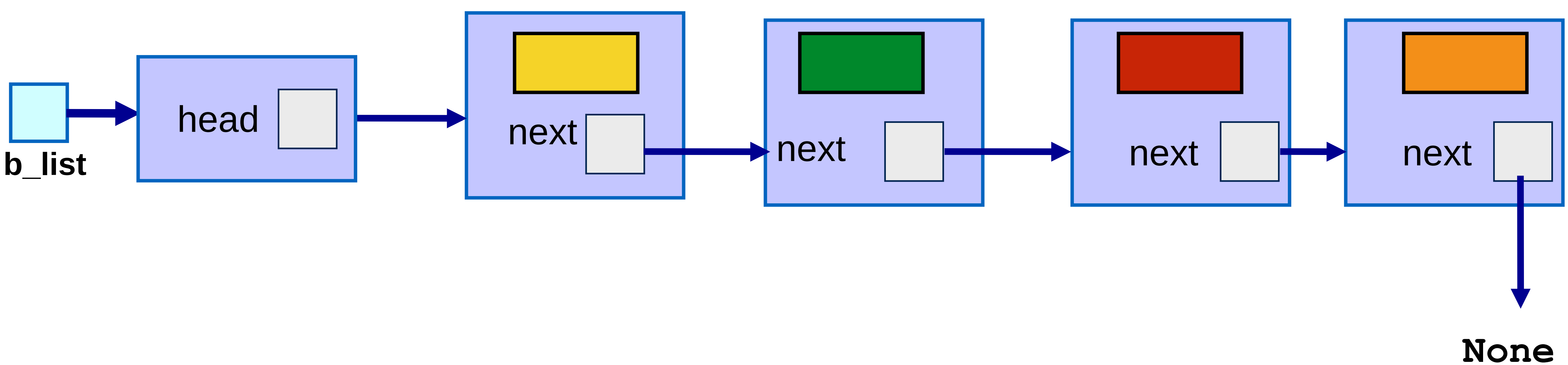
or is implicit

If complexity is the same, why bother with recursive implementations?

Copy Linked Lists




```
b_list = a_list.copy()
```

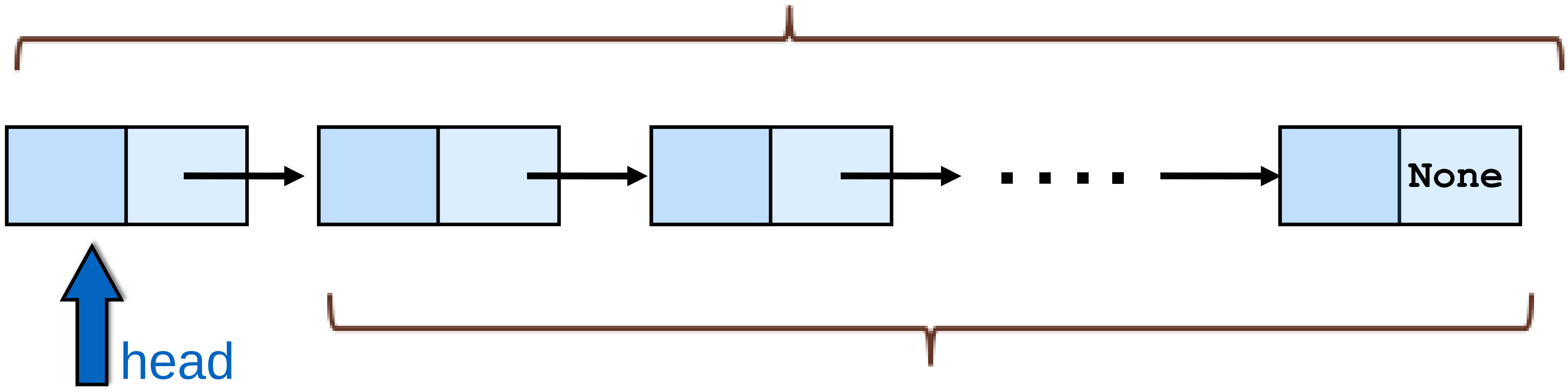


```python
def copy(self):
```

```python
def copy(self):
    new_list = List()
    for item in self:
        new_list.insert(len(new_list), item)
    return new_list
```

What about recursively?

_copy(self.head, new_list)



_copy(self.head.next, new_list)

new_list.insert(0, head.item)

copy

```
def copy(self):
```

copy

Auxiliary method se
up the initial paramete

```
def copy(self):
    new_list = List()
    self._copy_aux(self.head, new_list)
    return new_list
```

Convergence

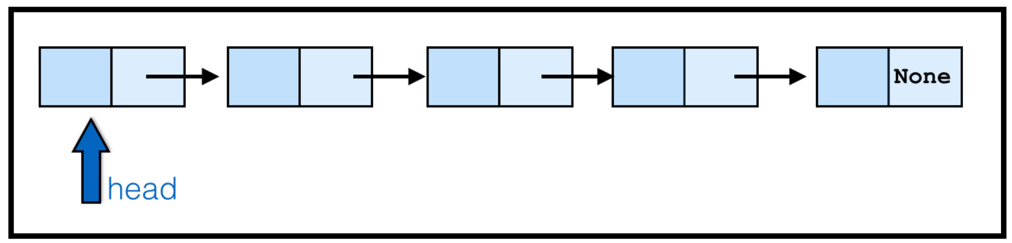
```
def _copy_aux(self, node, new_list):
    if node is not None:
        self._copy_aux(node.next, new_list)
        new_list.insert(0, node.item)
```

Combination

Base case

```python
def copy(self):
    new_list = List()
    self._copy_aux(self.head, new_list)
    return new_list

def _copy_aux(self, node, new_list):
    if node is not None:
        self._copy_aux(node.next, new_list)
        new_list.insert(0, node.item)
```

O(n)

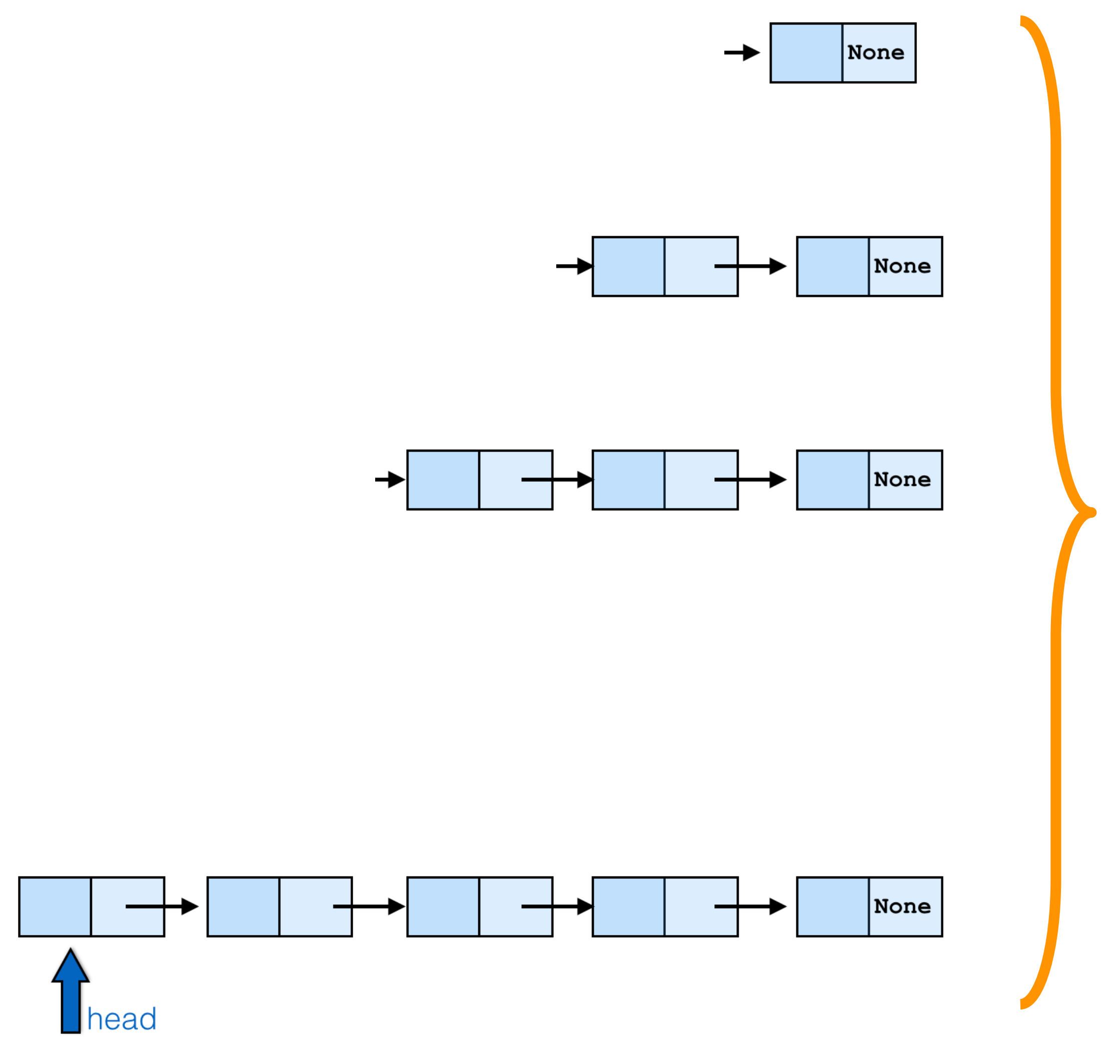


_copy_aux O(1)

_copy_aux O(1)

_copy_aux O(1)

_copy_aux O(1)



n times

Using iterators…

```
def copy(a_list):
        new_list = List()
        copy_aux(iter(a_list), new_list)
        return new_list

def copy_aux(iter, new_list):
    try:
        item = next(iter)
        copy_aux(iter, new_list)
        new_list.insert(0, item)
    except StopIteration:
        pass
```

copy

```
def copy(self):
    new_list = List()
    for item in self:
        new_list.insert(len(new_list), item)
    return new_list
```

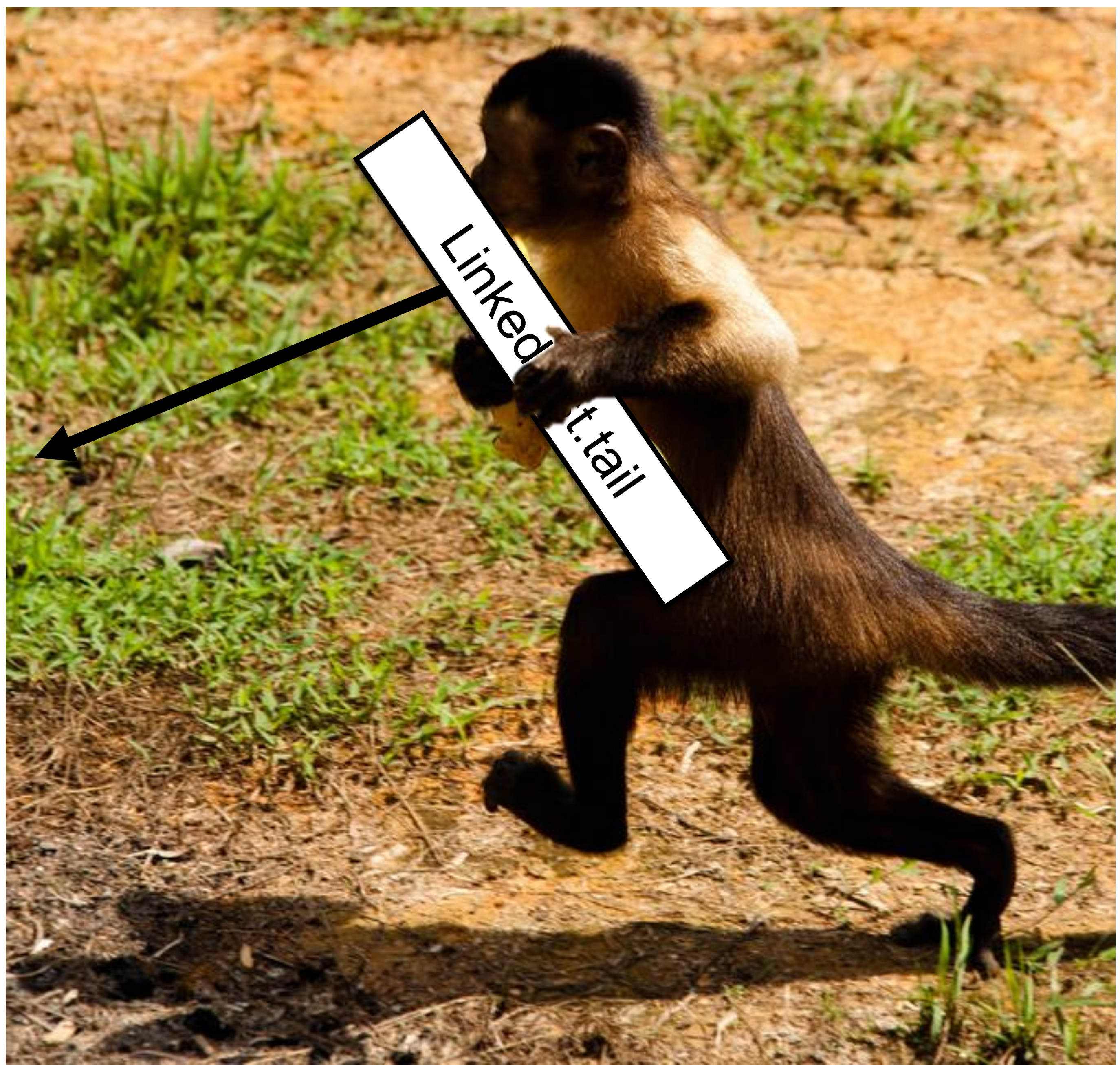
$O(n^2)$

```
def copy(self):
    new_list = List()
    self._copy_aux(self.head, new_list)
    return new_list

def _copy_aux(self, node, new_list):
    if node is not None:
        self._copy_aux(node.next, new_list)
        new_list.insert(0, node.item)
```

$O(n)$

Or….

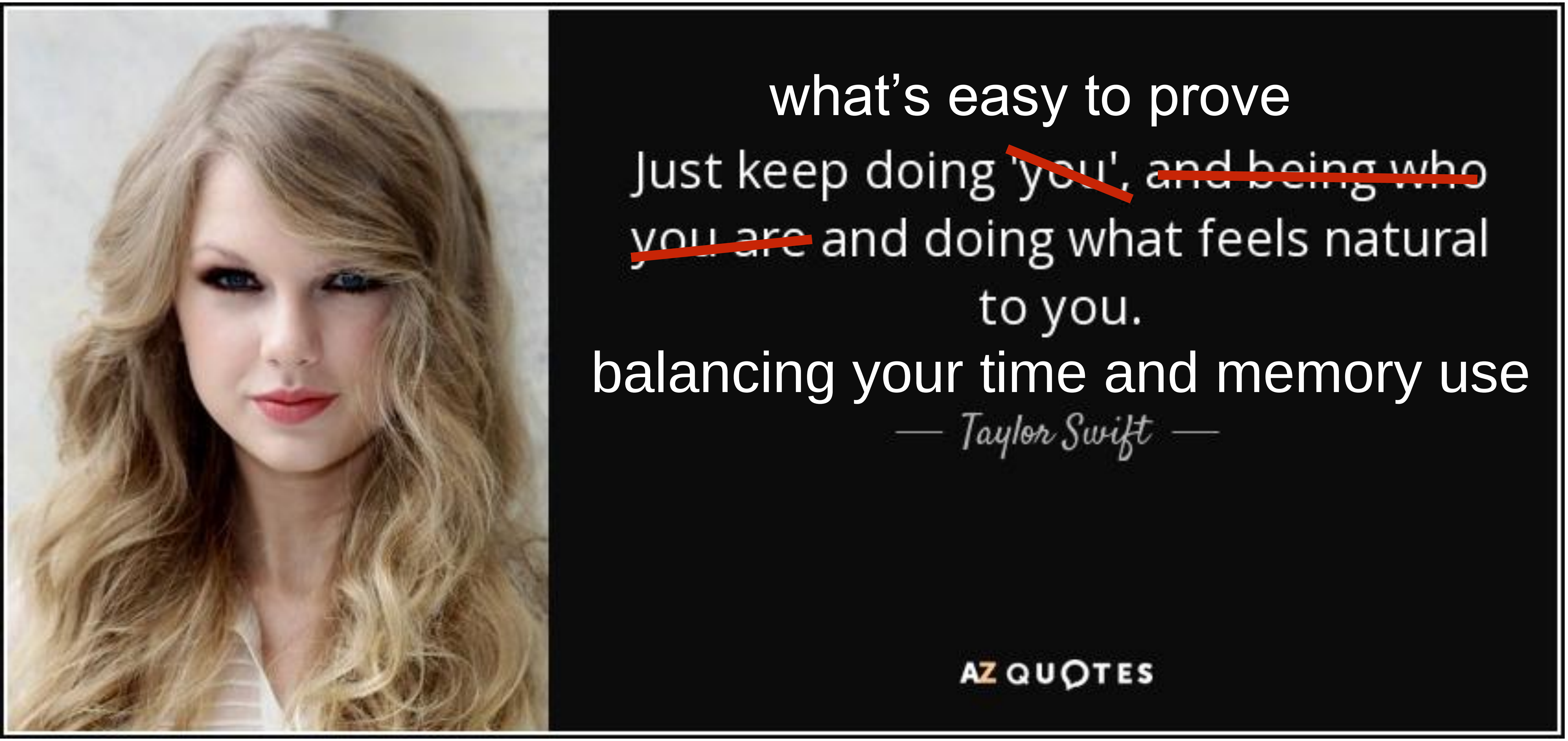
Have a tail attribute and you can have O(1) append and hence O(n) copy



Advantages/Disadvantages of Recursion

- Advantages:
  - More natural
  - Easier to prove correct
  - Easier to analyse

- Disadvantages:
  - Run-time overhead depending on the quality of the compiler
  - Memory overhead (fewer local variables versus stack space for function call)



I stole this quote from TayTay from here: http://www.azquotes.com/quote/863077

Summary

- Recursive algorithms are characterised by:
  1. Existence of base cases
  2. Decomposition into simpler subproblems
  3. Combination of solutions to subproblems

- Recursive methods require:
  1. One or more base cases
  2. One or more recursive calls
  3. Convergence in the recursive calls
  4. Combination of sub-solutions