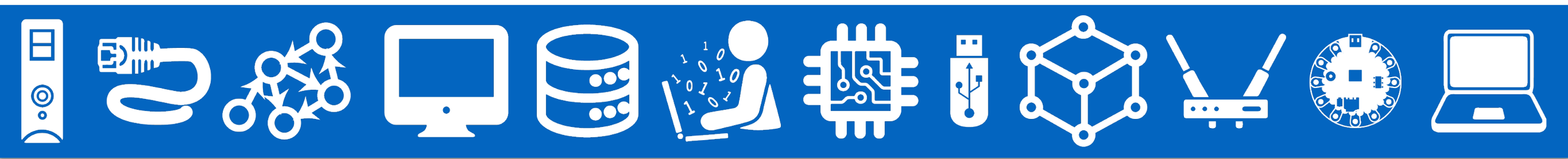
CS134 Lecture:
Linked Lists & Efficiency



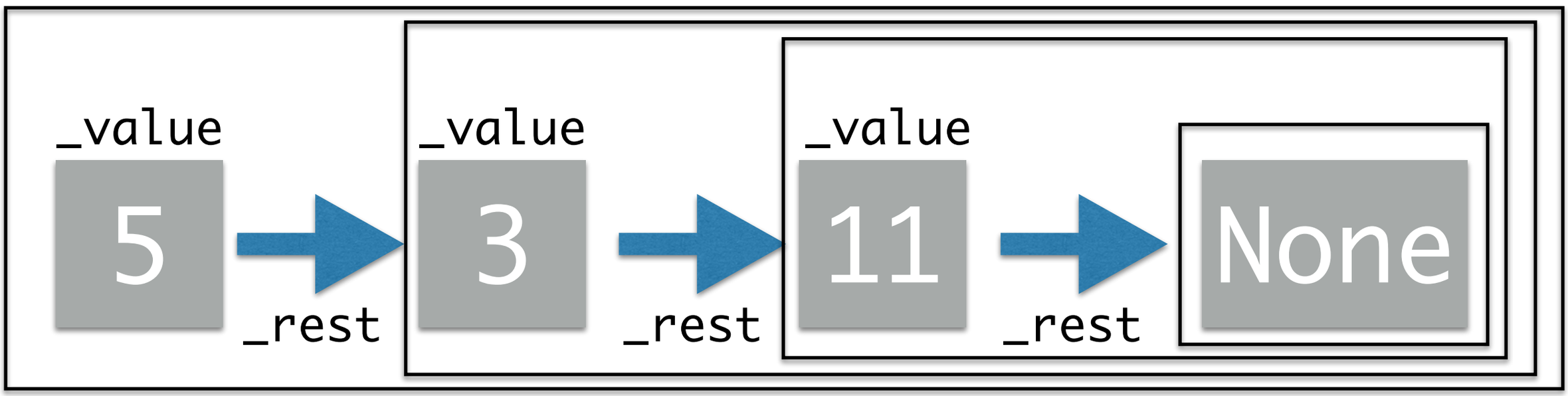
Announcements & Logistics

- **HW 10** due Monday @ 10 pm on Gradescope
- **Lab 9 Boggle**: two-week lab now in progress
  - **Part 1 -** you have automated feedback!
  - You should fix anything broken before turning in Part 2
  - **Part 2** due Wednesday/Thursday (handout posted today)
  - Part 2 also has a **prelab!**
    - Asks you to draw out the Boggle game logic
- **Final Exam**: Wednesday, December 11 at 9:30am in Wachenheim B11

Do You Have Any Questions?

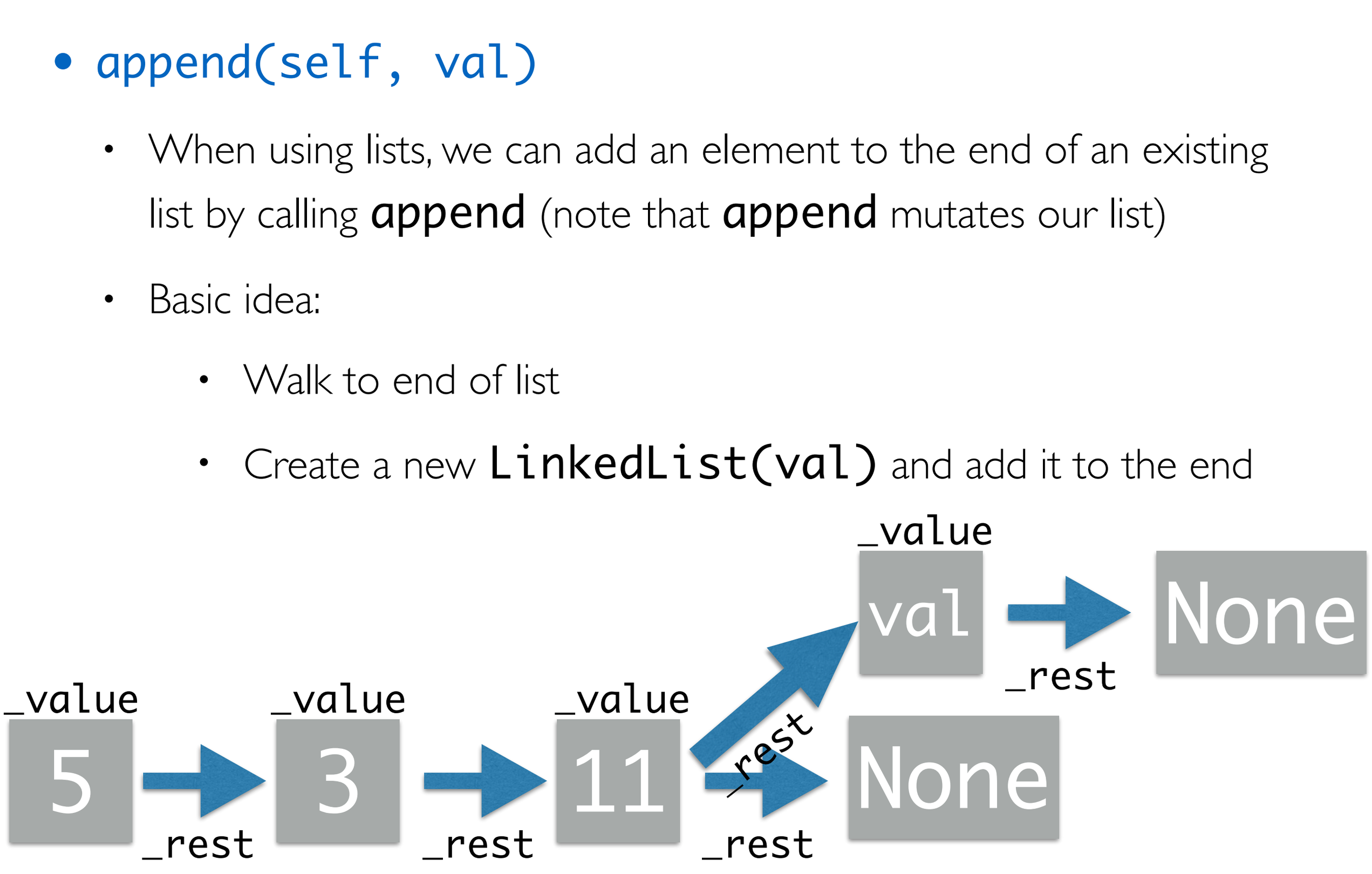
Last Time

- Started the implementation of our own linked list class
    - Why? Help us understand what's happening in Python's built-in classes
    - A glimpse of data structure design (precursor to CS136)
- Implemented several methods:
    - `__init__`, `__str__`, `__len__`, `__contains__` (in), `__eq__`, `__getitem__` ([ ]), `__setitem__` ...
    - append, prepend, insert



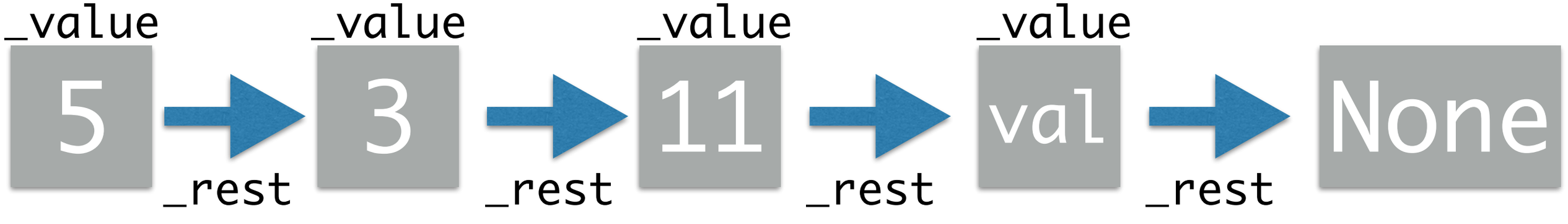
Last Time: `append`

- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling **`append`** (note that **`append`** mutates our list)
  - Basic idea:
    - Walk to end of list
    - Create a new `LinkedList(val)` and add it to the end



Last Time: `append`

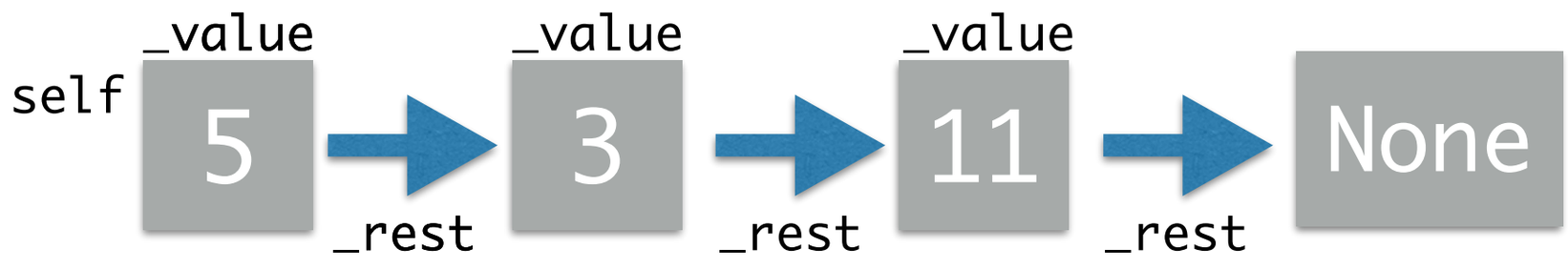
- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling **`append`** (note that **`append`** mutates our list)
  - Basic idea:
    - Walk to end of list
    - Create a new `LinkedList(val)` and add it to the end



Last Time: `prepend`

- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to **`append`**)
  - The **`prepend`** operation is really efficient, we don't need to walk through the list at all — just do some variable reassignments.

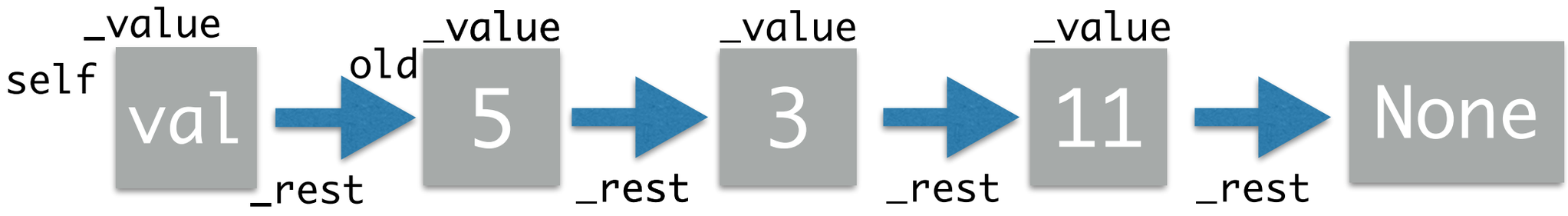
```
    def prepend(self, val):
        old_val = self._value
        old_rest = self._rest
        self._value = val
        self._rest = LinkedList(old_val, old_rest)
```



Last Time: `prepend`

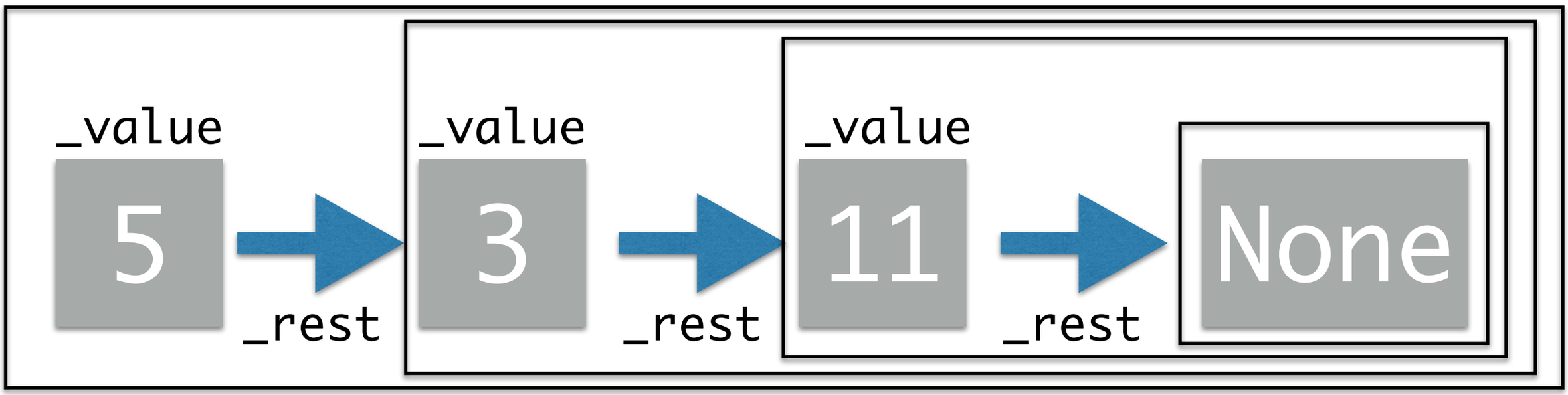
- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to **`append`**)
  - The **`prepend`** operation is really efficient, we don't need to walk through the list at all — just do some variable reassignments.

```
    def prepend(self, val):
        old_val = self._value
        old_rest = self._rest
        self._value = val
        self._rest = LinkedList(old_val, old_rest)
```

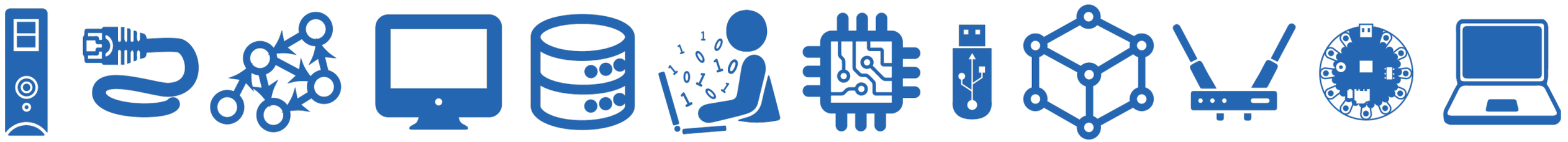


Today's Plan

- Discuss how we can turn our LinkedList into an "**iterable**" object
    - This will allow us to iterate over our lists in a for loop
    - Implement more special methods: `__iter__` and `__next__`
- Compare LinkedLists to contiguous arrays



Iterating Over Our List



Iterating Over Our List

- We can iterate over a Python list in a `for loop`
- It would be nice if we could **iterate** over our `LinkedList` in a for loop
- This won't quite work right now

```
for item in myList:
    print(item)
```

```
5
3
11
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-108-4bf86db75685> in <module>
----> 1 for item in myList:
      2     print(item)

<ipython-input-104-8a5ab5d1919c> in __getitem__(self, index)
     68             # else we recurse until index reaches 0
     69             # remember that this implicitly calls __getitem__
---> 70             return self._rest[index - 1]
     71

TypeError: 'NoneType' object is not subscriptable
```

Iterating Over Our List

- Currently, we can only indirectly **iterate** over our LinkedList using a loop and a `range` object.
- We'd really like to **iterate** directly over the elements of the list (without using a range)
- *An aside:* Given our LinkedList implementation, this loop is very inefficient! Each call to `new_lst[i]` walks the list out to index `i` each time.

```
new_lst = LinkedList(5)
new_lst.append(10)
new_lst.append(42)

for i in range(len(new_lst)):
    print(new_lst[i])
```

```
5
10
42
```

Making our List Iterable

- What do we need to directly **iterate** over our linked list?
    - We need to make our class **iterable**
    - We need to implement the special methods `__iter__` and `__next__`

- First, let's start with a few definitions

Making our List Iterable

- A Python object is considered **iterable** if it supports the `iter()` function: that is, the special method `__iter__` is defined
    - All ***sequences*** in Python are **iterable,** e.g., strings, lists, ranges, tuples, even files
    - We can **iterate** over an **iterable** object directly in a for loop
        - When an **iterable** object is passed to the `iter()` function, it creates an **iterator**
- An **iterator** object can generate values from the sequence **on demand**
    - This is accomplished using the `next()` function (and `__next__` method) which simply provides the "next" value in the sequence
- Note: **iterable** is an adjective, **iterator** is a noun, **iterate** is a verb

Python's Built-in Iterable Types

- We can create **iterators** for lists/strings/tuples by passing them to `iter()`
- Benefit? We can generate values from the sequence *on demand* (one at a time)
- An **iterator** maintains "state" between calls to `next()` (it remembers where we are)
- Once all values in the sequence have been iterated over, the **iterator** "runs dry" (and becomes empty)
- We can only iterate over values once (unless we create another **iterator**)

```
>>> char_lst = list("rain")
>>> print(char_lst)
['r', 'a', 'i', 'n']
>>> char_iterator = iter(char_lst)
>>> next(char_iterator)
'r'
>>> next(char_iterator)
'a'
>>> next(char_iterator)
'i'
>>> next(char_iterator)
'n'
>>> next(char_iterator)
Traceback:
  File "<stdin>", line 1
StopIteration
```

This means there are no elements left!

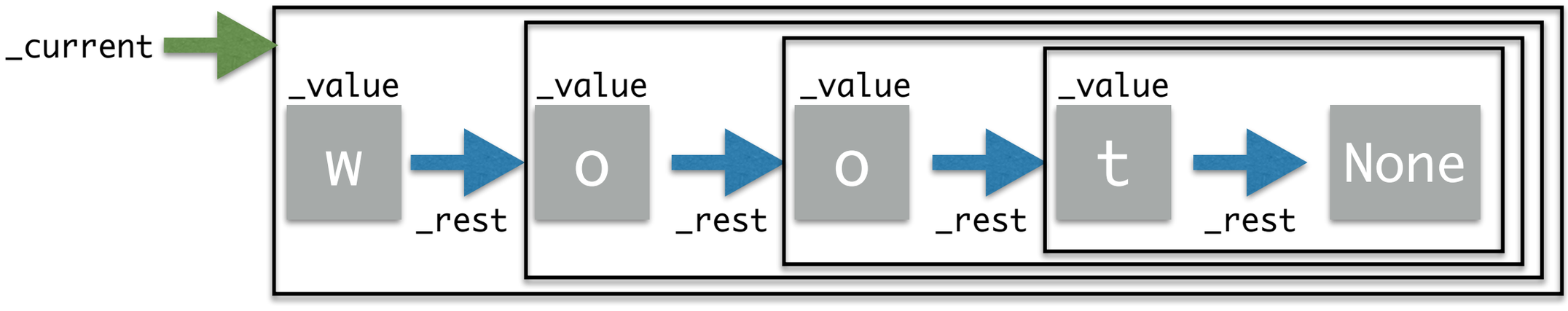
Creating an Iterator

- To create an **iterator** for our class we need to implement two methods:
  - `__iter__()` which is called to creates the iterator
  - `__next__()` which is called to advance to the next value
- The key aspect of creating iterators: maintaining state to keep track of ***where you are currently in the sequence*** (and what is the **next** value that should be returned)
- Thus, `__iter__()` should always "reset" the current state to the beginning of our list, and `__next__()` should update this state (i.e., move to the next element) each time its called
- Python for loops automatically (and implicitly) create an iterator and call `next()` until the `StopIteration` exception is reached (see leftover slides at the end for more info!)

Creating an Iterator for LinkedList

- First we add a new attribute, `'_current'`
    - `_current` keeps track of where we are in the iterator

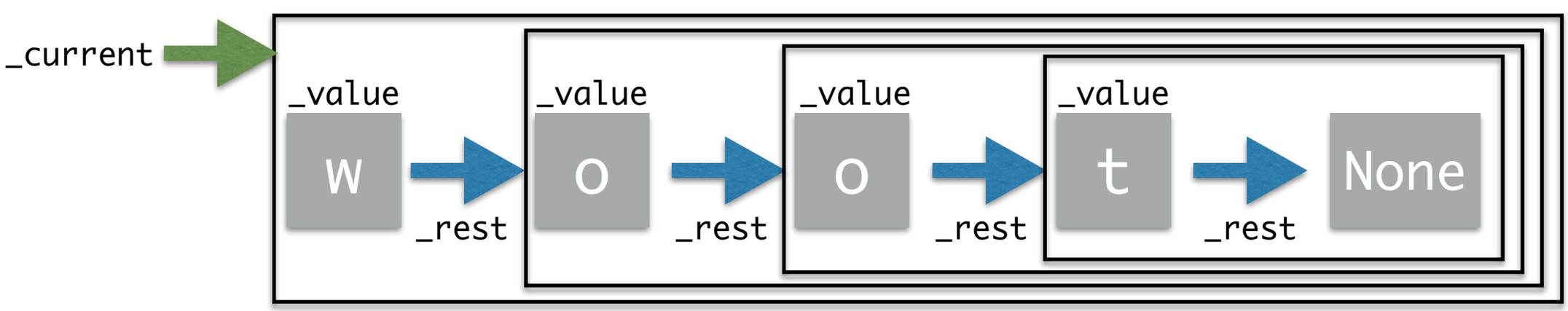
```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self
```



Creating an Iterator for LinkedList

- Then we need to implement **`__next__`** so that **`_current`** will be updated to the next LinkedList in the sequence

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to next element in list
        val = self._current._value
        self._current = self._current._rest
        return val
```



Creating an Iterator for LinkedList

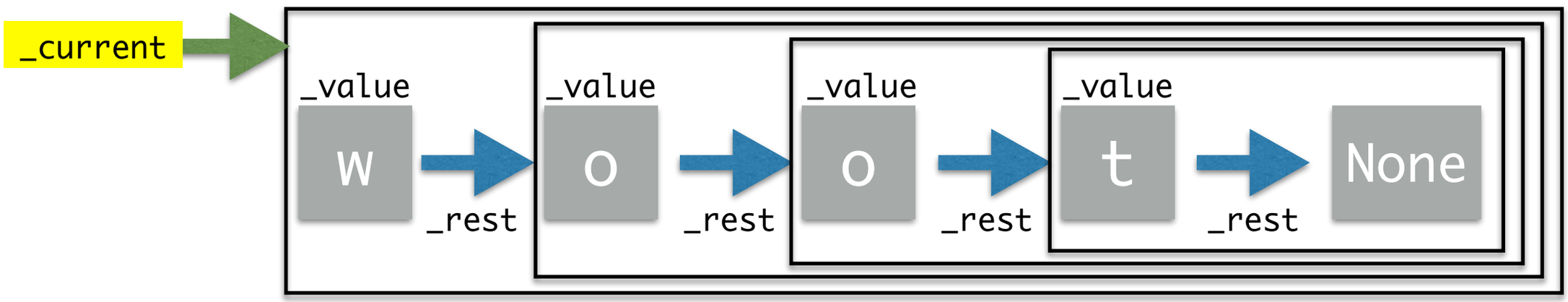
- Now let's make a sample LinkedList to walk-through

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to next element in list
        val = self._current._value
        self._current = self._current._rest
        return val
```

```
>>> test_lst = LinkedList()
>>> test_lst.append("w")
>>> test_lst.append("o")
>>> test_lst.append("o")
>>> test_lst.append("t")
>>> for char in test_lst:
...     print(char)
```

0th time

```
w
```



Creating an Iterator for LinkedList

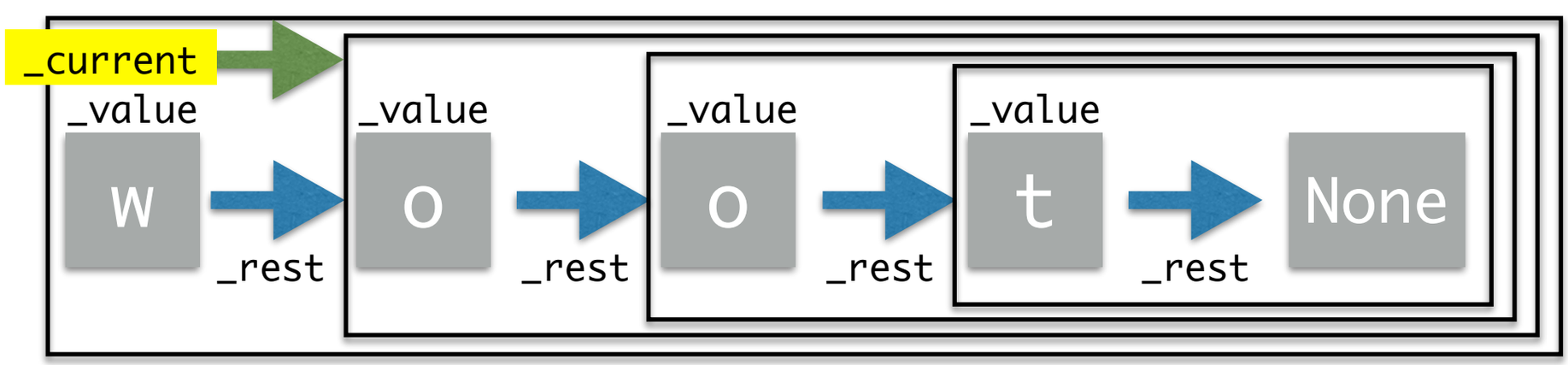
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to next element in list
        val = self._current._value
        self._current = self._current._rest
        return val
```

```
>>> test_lst = LinkedList()
>>> test_lst.append("w")
>>> test_lst.append("o")
>>> test_lst.append("o")
>>> test_lst.append("t")
>>> for char in test_lst:
...     print(char)
```

1th time

```
w
o
```



Creating an Iterator for LinkedList

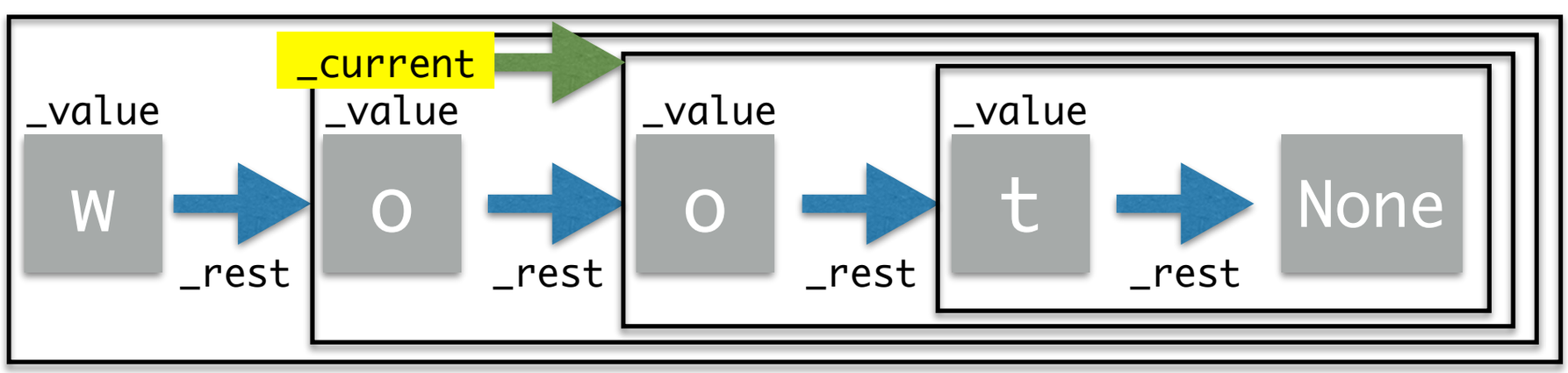
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to next element in list
        val = self._current._value
        self._current = self._current._rest
        return val
```

```
>>> test_lst = LinkedList()
>>> test_lst.append("w")
>>> test_lst.append("o")
>>> test_lst.append("o")
>>> test_lst.append("t")
>>> for char in test_lst:
...     print(char)
```

2th time

```
w
o
o
```



Creating an Iterator for LinkedList

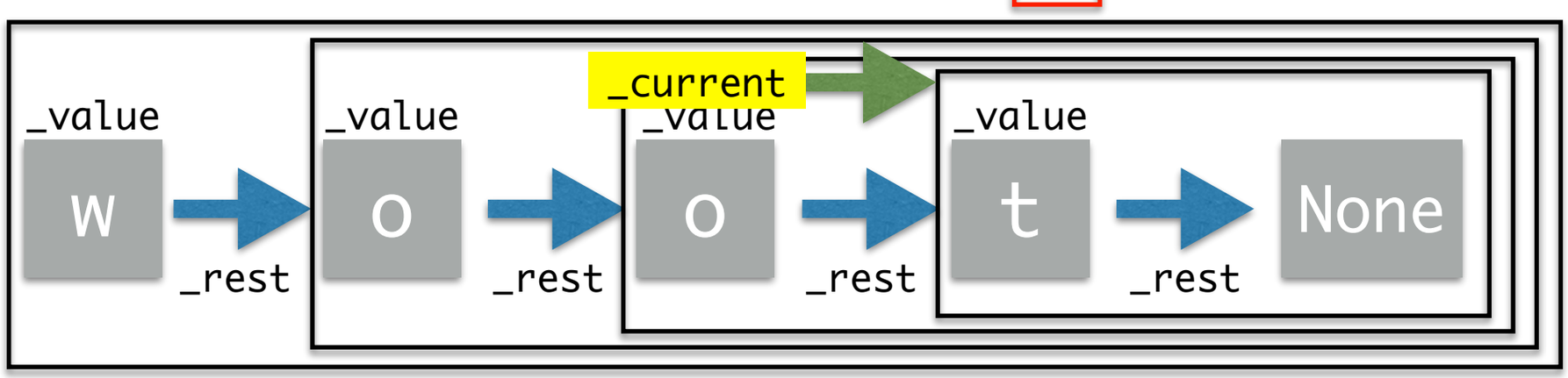
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to next element in list
        val = self._current._value
        self._current = self._current._rest
        return val
```

```
>>> test_lst = LinkedList()
>>> test_lst.append("w")
>>> test_lst.append("o")
>>> test_lst.append("o")
>>> test_lst.append("t")
>>> for char in test_lst:
...     print(char)
```

3th time

```
w
o
o
t
```



Creating an Iterator for LinkedList

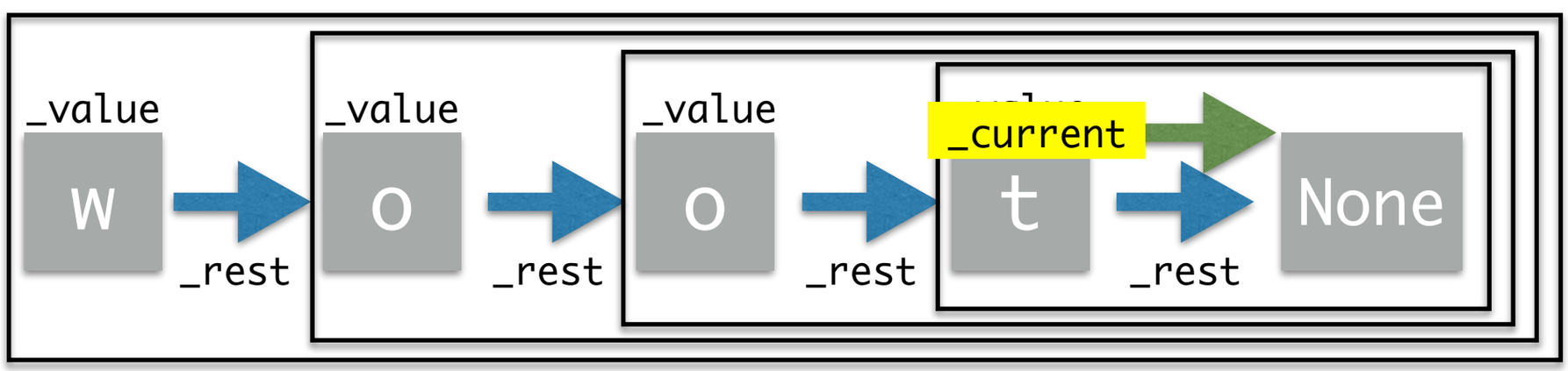
- First we add a new attribute `'_current'` to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to next element in list
        val = self._current._value
        self._current = self._current._rest
        return val
```

This means there are no elements left!

```
>>> test_lst = LinkedList()
>>> test_lst.append("w")
>>> test_lst.append("o")
>>> test_lst.append("o")
>>> test_lst.append("t")
>>> for char in test_lst:
...     print(char)
```

```
w
o
o
t
```



Using our New Iterable LinkedList

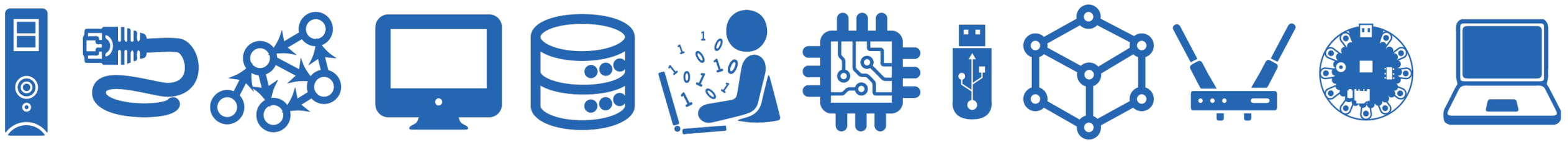
```
test_lst = LinkedList("w")
test_lst.append("o")
test_lst.append("o")
test_lst.append("t")
print("test_lst: ",test_lst)

# for loops automatically use iterators
for char in test_lst:
    print(char)
```

```
test_lst: [w, o, o, t]
w
o
o
t
```

```
list_iterator = iter(test_lst)
print(next(list_iterator))
print(next(list_iterator))
print(next(list_iterator))
print(next(list_iterator))
```

```
w
o
o
t
```

Why Build a LinkedList?

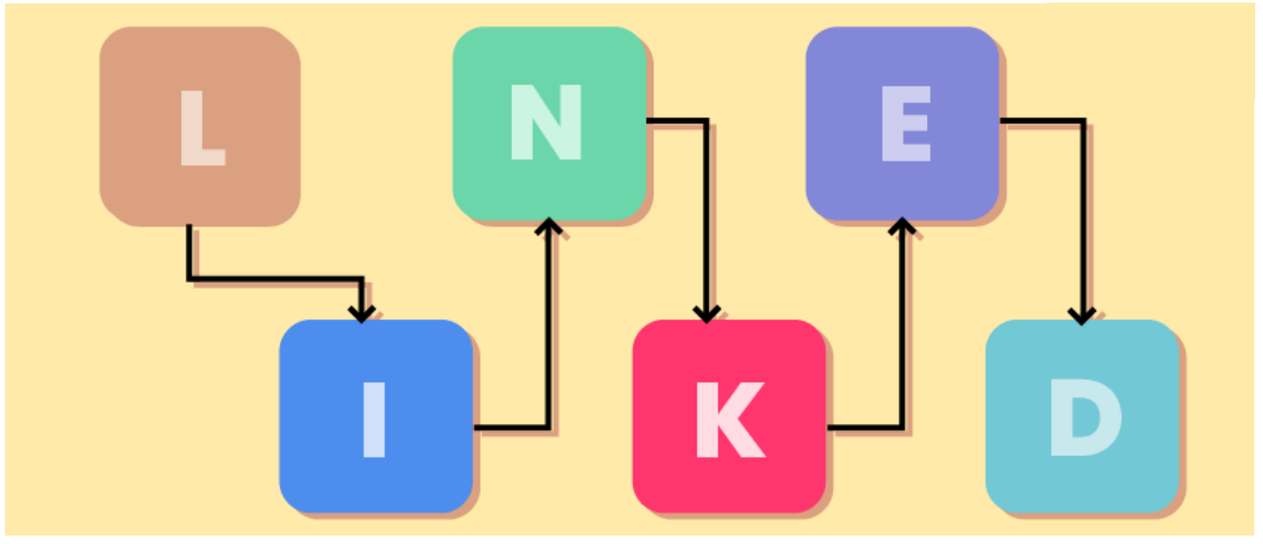


How to Store a Sequence

- A sequence is just an **ordered collection** of values
    - Can query for the 0th, 1st, 2nd item and so on..
- A sequence may be mutable or immutable
- Let's think about how we can design such a sequence
    - How to store these ordered values?
- Let's look at two options

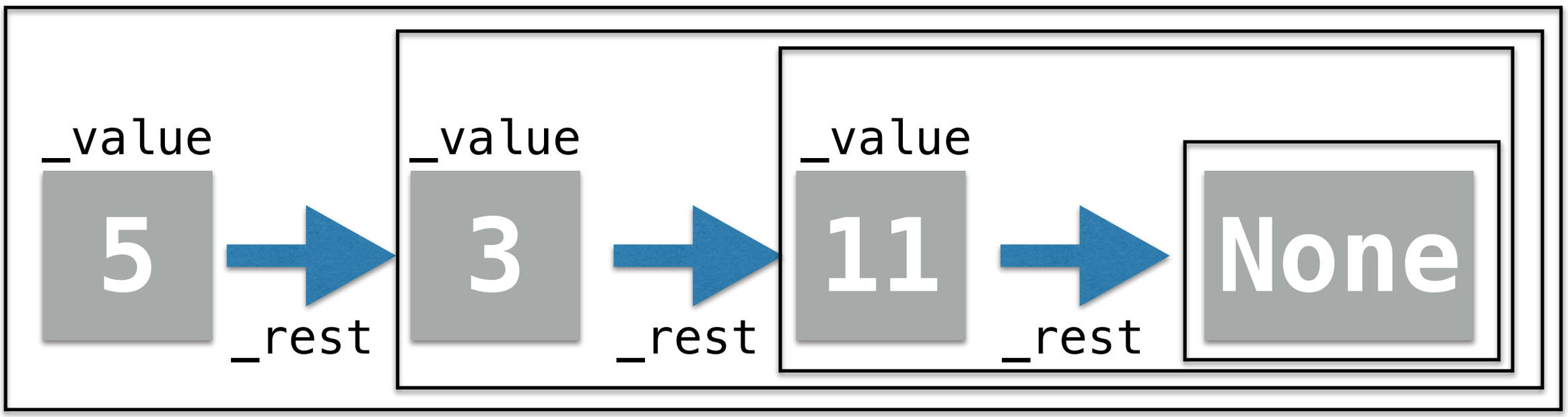
Linked List

- An ordered mutable sequence:
    - A nested *chain* of values, or a **linked list**
    - Each value has something after it: the rest of the sequence
- Must have a last item for a finite sequence
    - To signify last item it's next value should be **nothing**
    - What is a good type to represent nothing in Python?



Our Own Class `LinkedList`

- Attributes:
  - `_value, _rest`
- **Recursive class**:
  - `_rest` points to another instance of the ***same class***
  - Any instance of a class that is created by using another instance of the class is a ***recursive class***

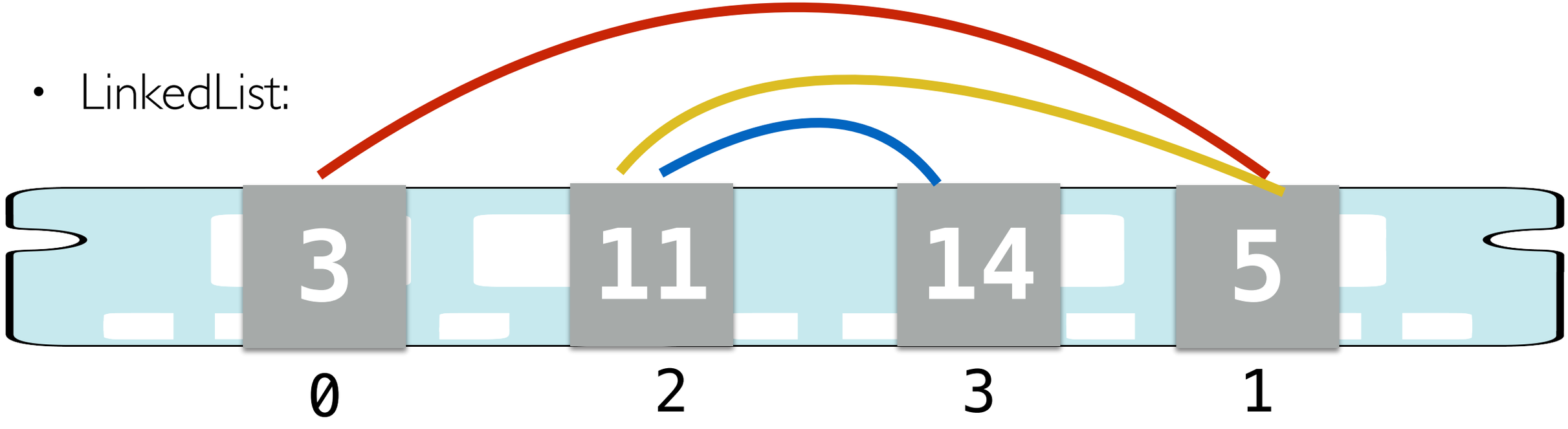


Linked List Efficiency

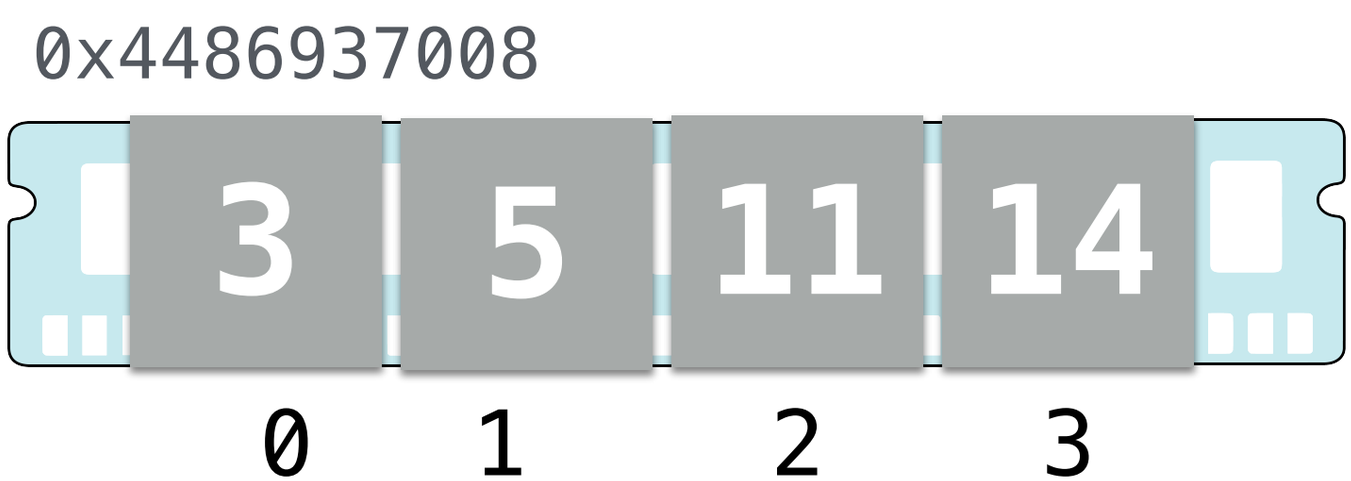
- How can we compare the efficiency of the following LinkedList operations?
    - append an item at the end of a LinkedList
    - prepend an item to the beginning of a LinkedList
- Any thoughts on which is "faster"
    - append needs to traverse the entire list to find last item
        - "number of steps" proportional to number of items
    - prepend just needs to change `self._rest` of newly inserted item
        - this is independent of how many items are in the LinkedList
- This is intuitively why prepend is more efficient than append
- For more formal discussion: need to figure out what we want to measure

Array: Contiguous Sequence

- LinkedList:

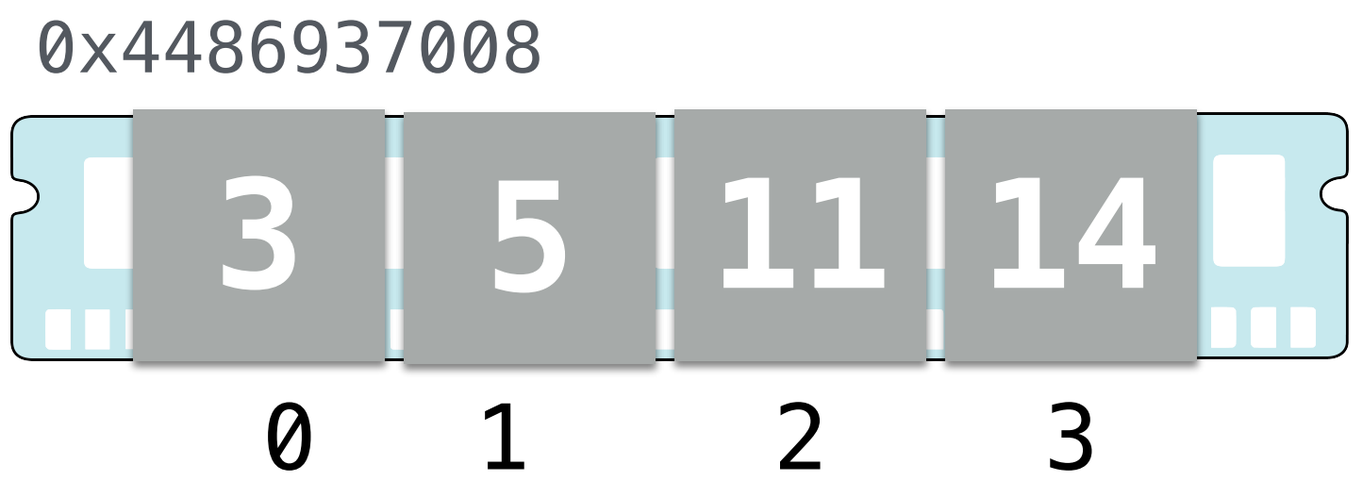


- Contiguous Array: Just store the items contiguously in memory
  - ...typically only have items of one **data type**



Array: Contiguous Sequence

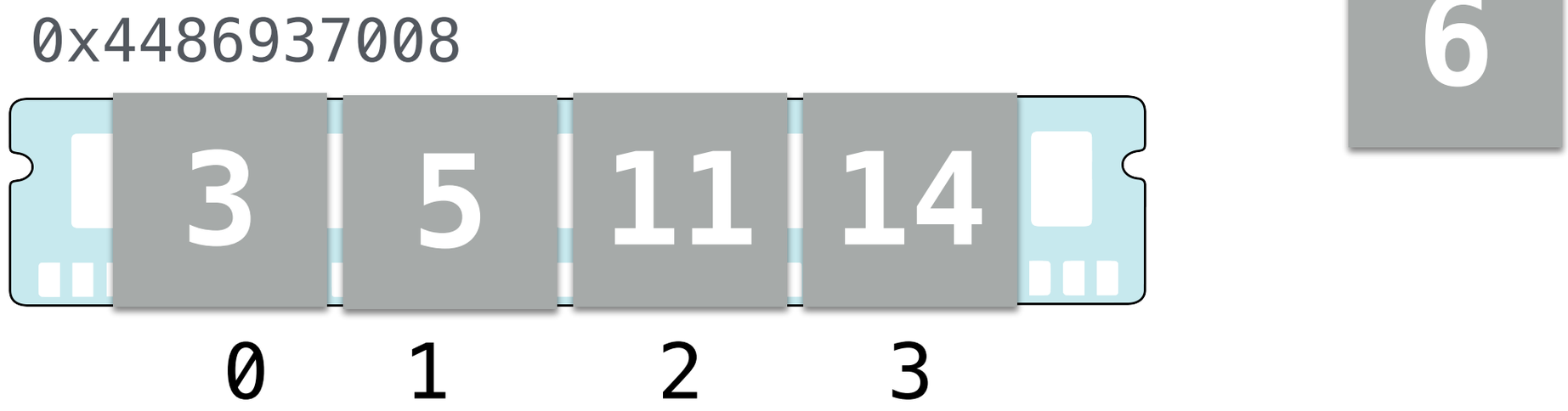
- Another ordered mutable sequence: Just store the items contiguously in memory
  - ...typically only have items of one **data type**
- Such a sequence is called **an array** in computer science
- To access a `item`, just need to know where the array starts and the **index** of item in array
- Great for static sequences! (i.e., sequences we don't change)



Array: Contiguous Sequence

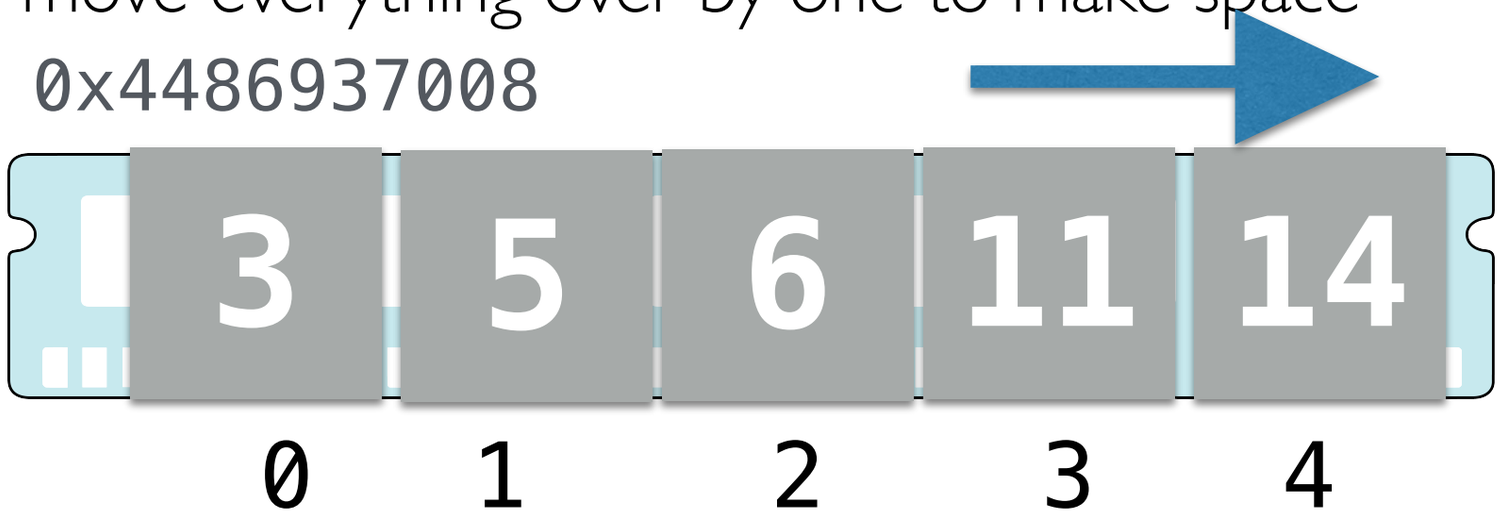
- Another ordered mutable sequence: Just store the items contiguously in memory
- Such a sequence is called **an array** in computer science
- To access a `item`, just need to know where the array starts and the **index** of item in array

- Suppose we want to create a dynamic sequence
  - Want to be able to insert: e.g. want to insert 6 between 5 and 11



Array: Contiguous Sequence

- Another ordered mutable sequence: Just store the items contiguously in memory
- Such a sequence is called **an array** in computer science
- To access a `item`, just need to know where the array starts and the **index** of item in array
- Suppose we want to create a dynamic sequence
    - Want to be able to insert: e.g. want to insert 6 between 5 and 11
    - Need to move everything over by one to make space

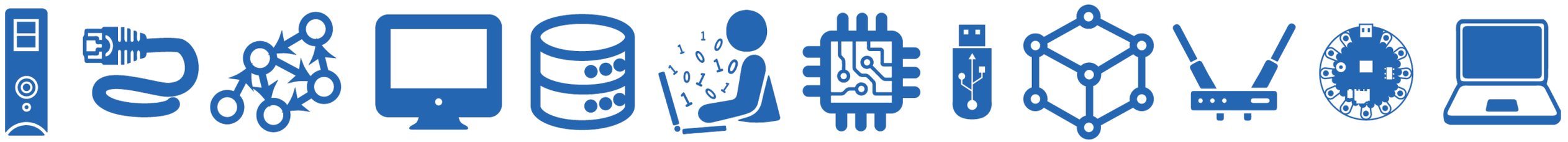


Insert Efficiently?

- If our array is made up of millions of items and we need to insert a lot:
    - Expensive to maintain ordering in an array
    - Maybe we need a different way to store items
- All we care about is that items are in order:
    - Each item has a item before it or after it in the sequence
    - Knowing this is sufficient to recover the total ordering, why?

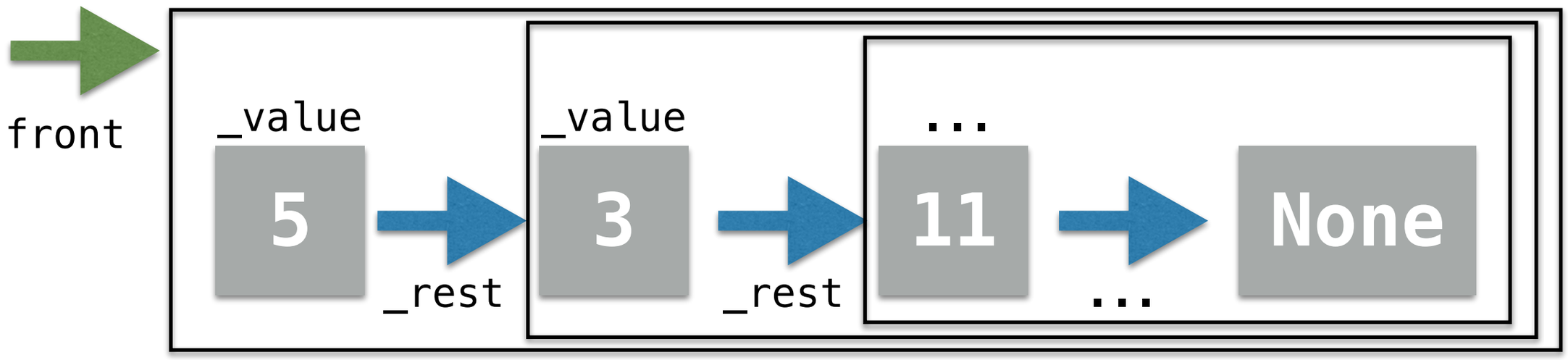
Lists (Arrays) vs. Linked Lists

Efficiency Trade Offs

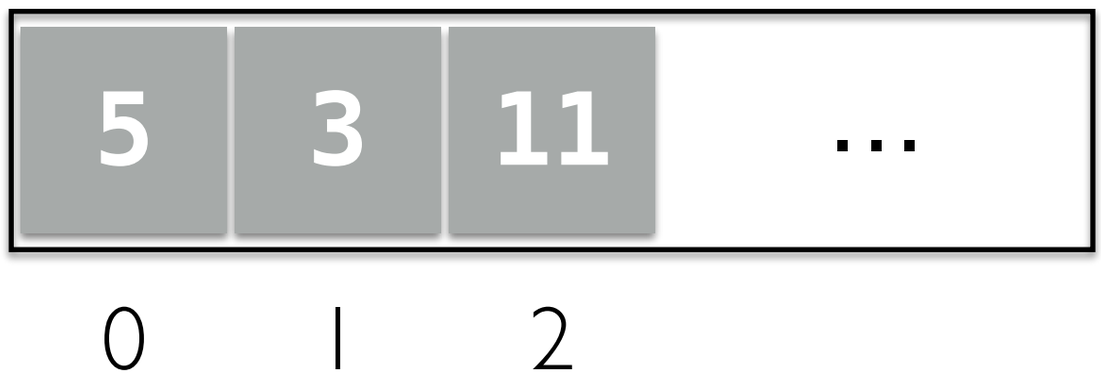


Lists vs Linked Lists

- **Linked Lists**: "pointer-based" data structure, items need not be contiguous in memory

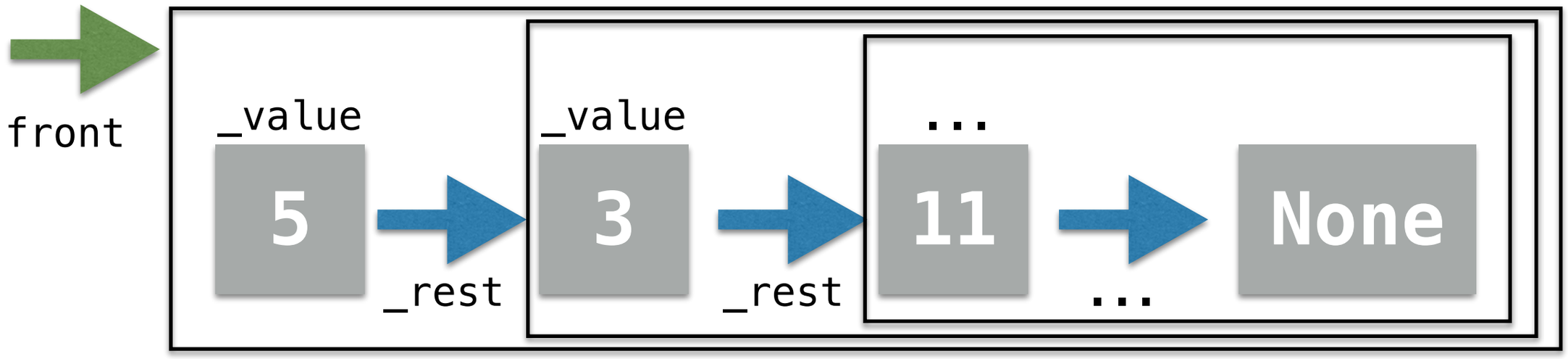


- **Arrays:** index-based data structure items are always stored contiguously in memory (think of a Python built-in list as an array)

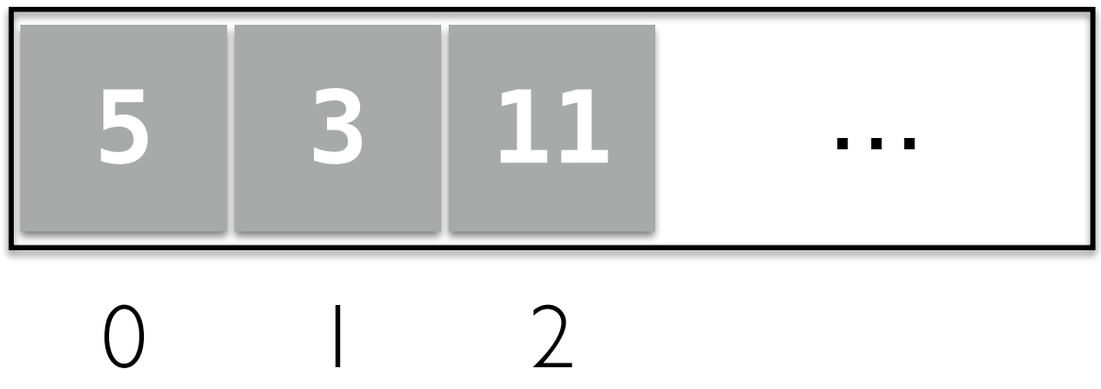


Lists vs Linked Lists

- **Linked Lists**: Can grow and shrink on the fly: do not need to know size at the time of creation (therefore no wasted space!)

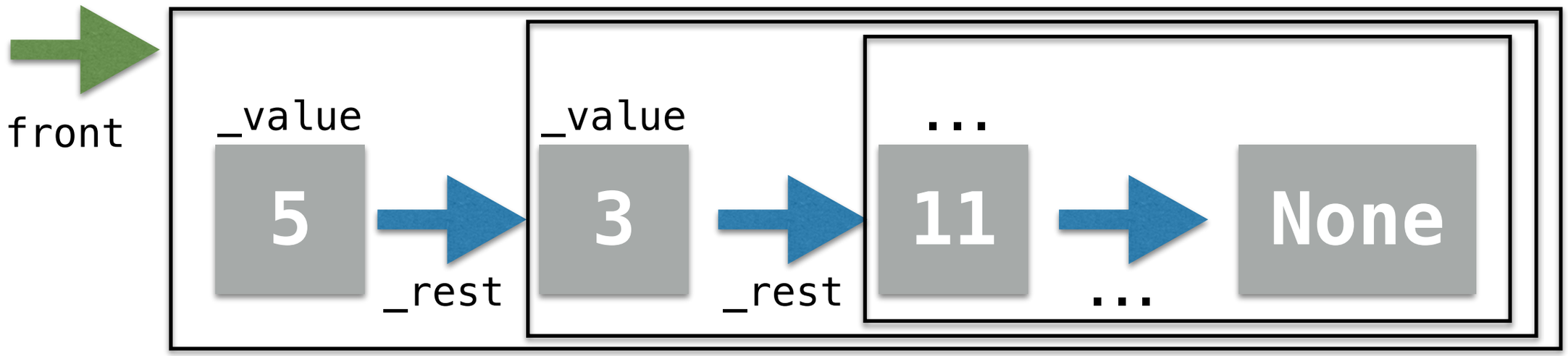


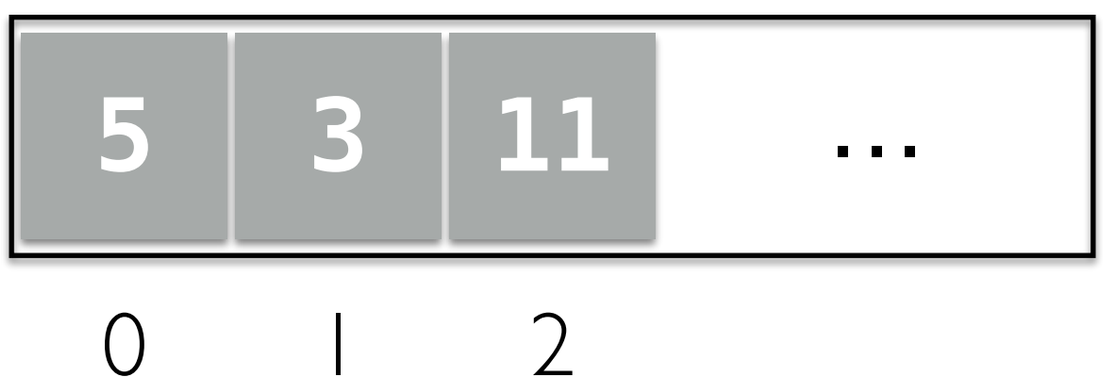
- **Arrays:** index-based data structure items are always stored contiguously in memory (think of a Python built-in list as an array)



Array vs Linked Lists

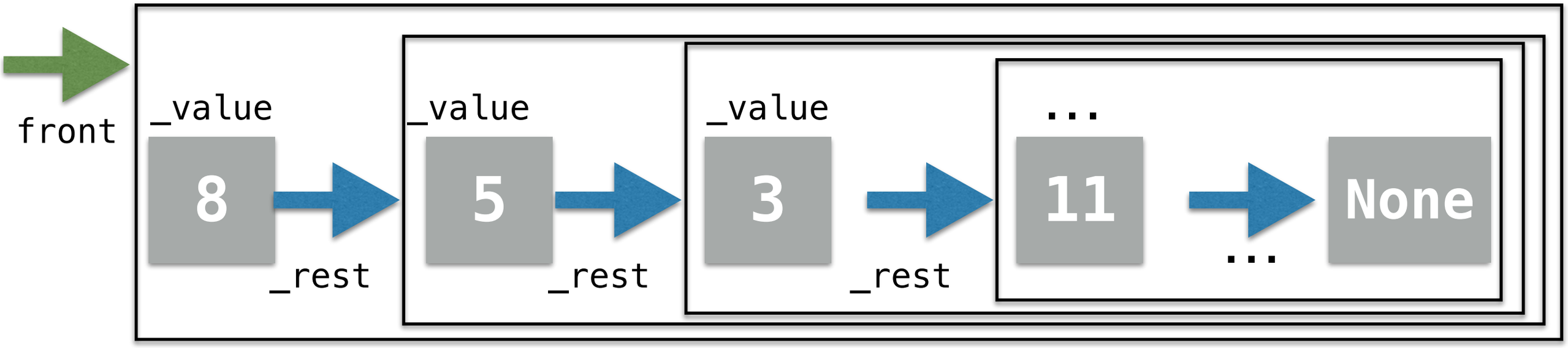
- Inserts at the beginning: which one is better?





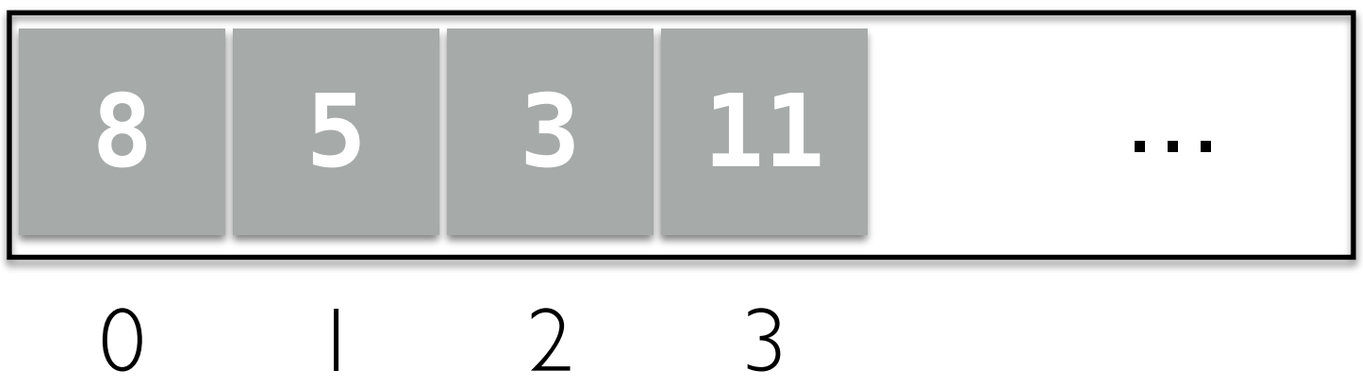
Array vs Linked Lists

- Linked list steps:
  - Point front to new element
  - Point rest of new element to old list
  - These steps don't depend on size of list
  - Therefore, run-time is **constant**, that is, $O(1)$ time



Array vs Linked Lists

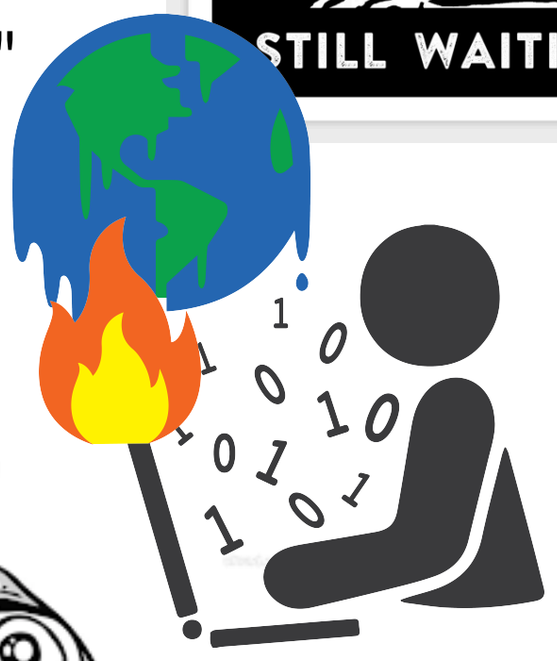
- Now consider an array-based list
- To insert at index 0, we need to shift every element over by one spot
    - This takes time proportional to the size: linear time or $O(n)$
- So when are arrays more efficient?
    - When **indexing** elements: they give **direct access** $O(1)$
    - Linked list: we need to traverse the list to get to the element $O(n)$



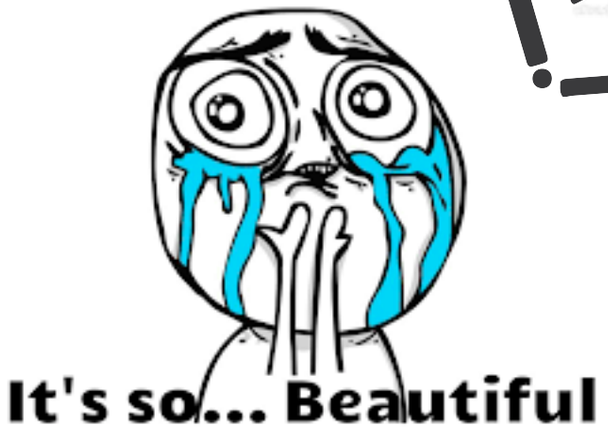
Why does algorithmic efficiency matter?

- We want programs that run "**fast**"

- We want programs that run "**efficiently**"



- We want programs that run "**optimally**"

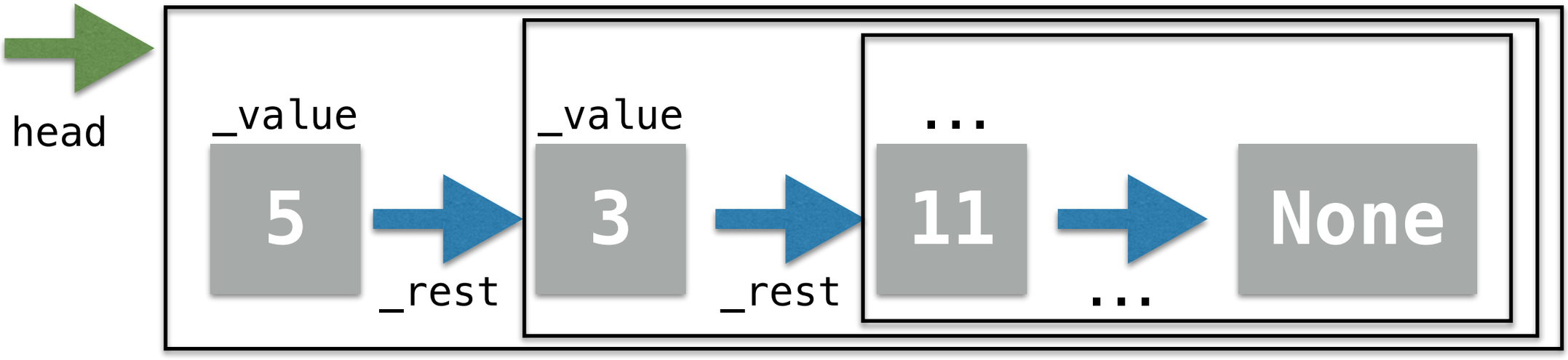


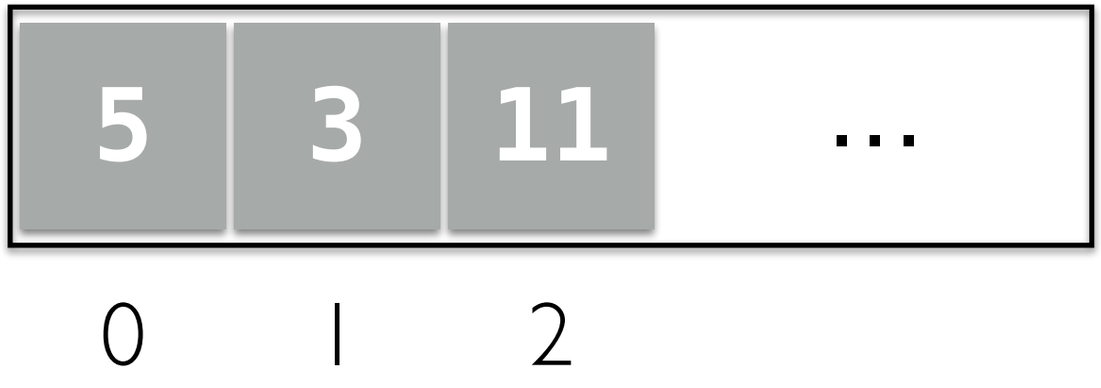
So Which is Better?

- It depends!
- Think about what operations are a priority in your program!
    - We should choose the way to represent/organize our data according to how we plan to use the data
- Think: prepend vs. append with LinkedLists vs. Arrays

An Aside: What exactly is Python's list?

- It's complicated: Python's list implementation is a hybrid





The end!