CS134 Lecture 29: Special Methods & Linked Lists

Announcements & Logistics

- **HW 9** due tonight @ 10 pm on GLOW
  - Short: 6 questions for practice on OOP concepts
- **Lab 9 Boggle**: two-week lab now in progress
  - **Part 2** due May 1/2 (handout posted)
  - Part 2 also has a **prelab!**
    - Asks you to draw out the Boggle game logic
    - Draw it on a sheet of paper and bring the diagram to lab
    - Make sure it is legible and clear!

Do You Have Any Questions?

Last Time

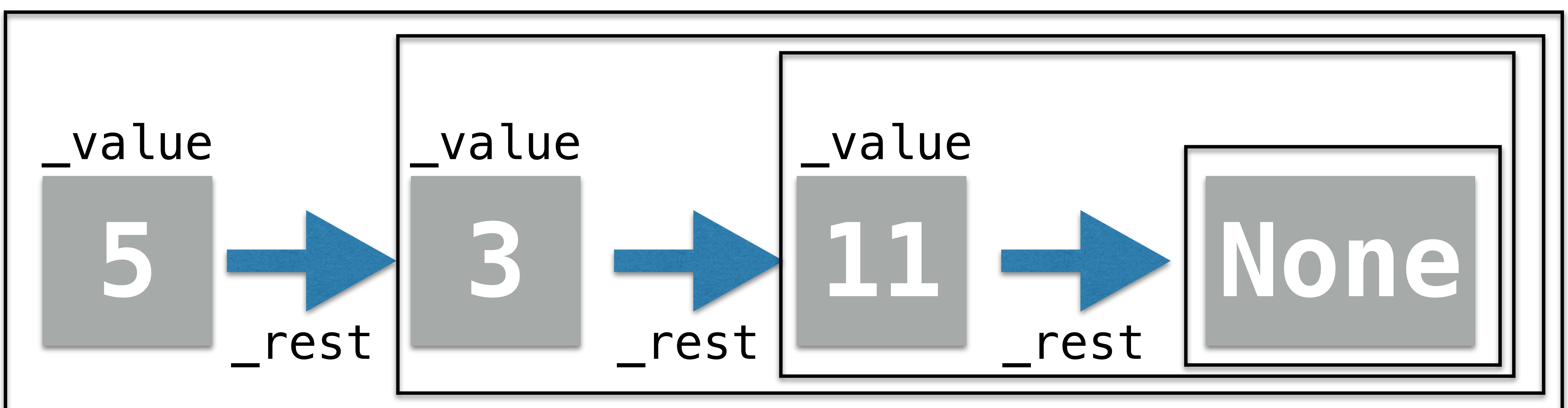
- Learn how to implement several **special methods** which let us utilize built-in operators in Python for user-defined types
- Discussed options to store an ordered mutable sequence:
  - **Arrays**: elements stored contiguously in memory
    - ***Upside***: fast accesses (constant # of steps)
    - ***Downside***: slow inserts (might have to shift everything!)
  - **Linked List**: elements stored (possibly non-contiguously) but remember the next item's location
    - ***Upside***: fast inserts at the front of list (may need to traverse whole list for updates in middle but requires no shifting)
    - ***Downside***: slow access (might have to traverse everything!)

Today's Plan

- Write our own implementation of LinkedList
- Implement functionality (write code) for special methods:
    - `__init__`
    - `__str__`
    - `__len__`
    - `__getitem__`
    - `__contains__`
- Discuss at high level (without code) other functionality we may want

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***

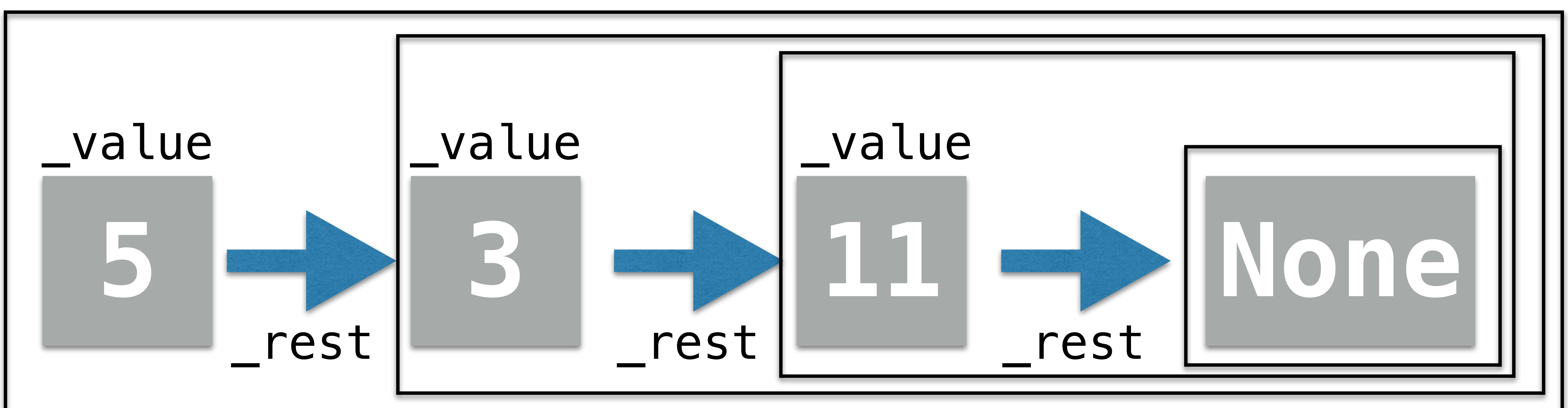


Initializing Our LinkedList

```python
class LinkedList:
    """Implements our own recursive list data
structure"""

    def __init__(self, value=None, rest=None):
        self._value = value
        self._rest = rest
```

How do we create an **empty** list?

`rest` is another instance of our LinkedList class



Recursive Implementation: `__str__`

- Let's think about how to implement a string representation of our list
- What is the base case?
  - What if our list has only one item
  - Just return `str` (value) (so if value is int, this return `str(5)` e.g.)
- How do we check if list only has one item in it?
  - `_rest` is `None`

Recursive Implementation: `__str__`

```python
# str() function calls __str__() method
def __str__(self):
    if self._rest is None:
        return str(self._value)
```

Notice the use of `is` and not `==`

Python: "`is None`" vs "`== None`":

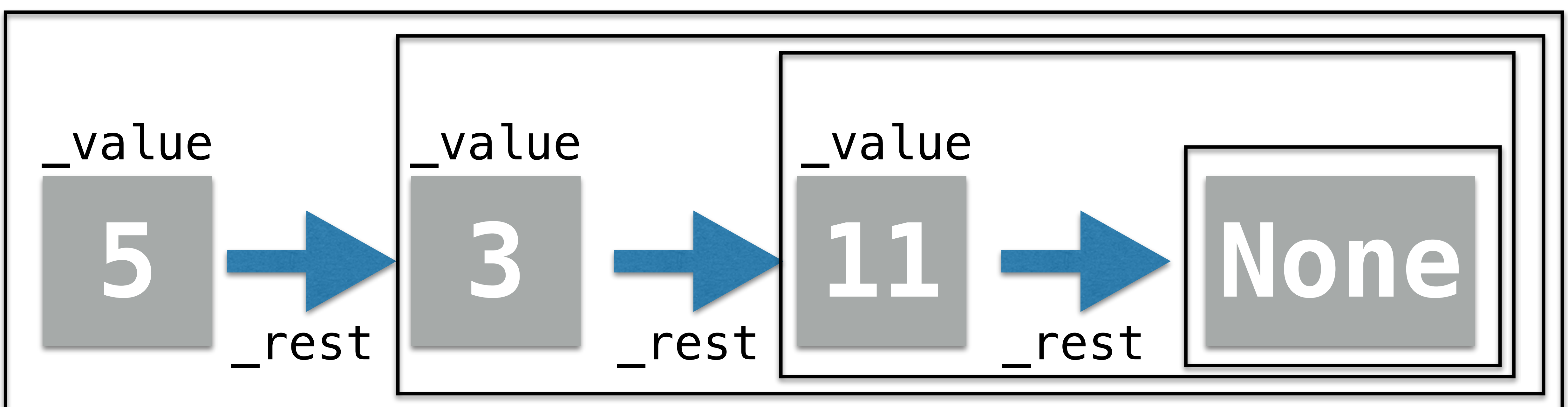
PEP 8 (Style Guide for Python Code) says:

"Comparisons to singletons like None should always be done with 'is' or 'is not', never the equality operators."

Recursive Implementation: `__str__`

```python
# str() function calls __str__() method
def __str__(self):
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ', ' + str(self._rest)
```

This is recursion since `str` calls `__str__`. The base case is when `self._rest` is `None`



Recursive Implementation: `__str__`

- What if we want to enclose the elements in square brackets `[ ]`?
- **Idea:** Use a helper method that does the same thing as `__str__()` on the previous slide, and then enclose its return in `'[ ]'`

```
def __get_string(self):
    '''Helper method for str of contents'''
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ', ' + self._rest.__get_string()

def __str__(self):
    return "[" + self.__get_string() + "]"
```

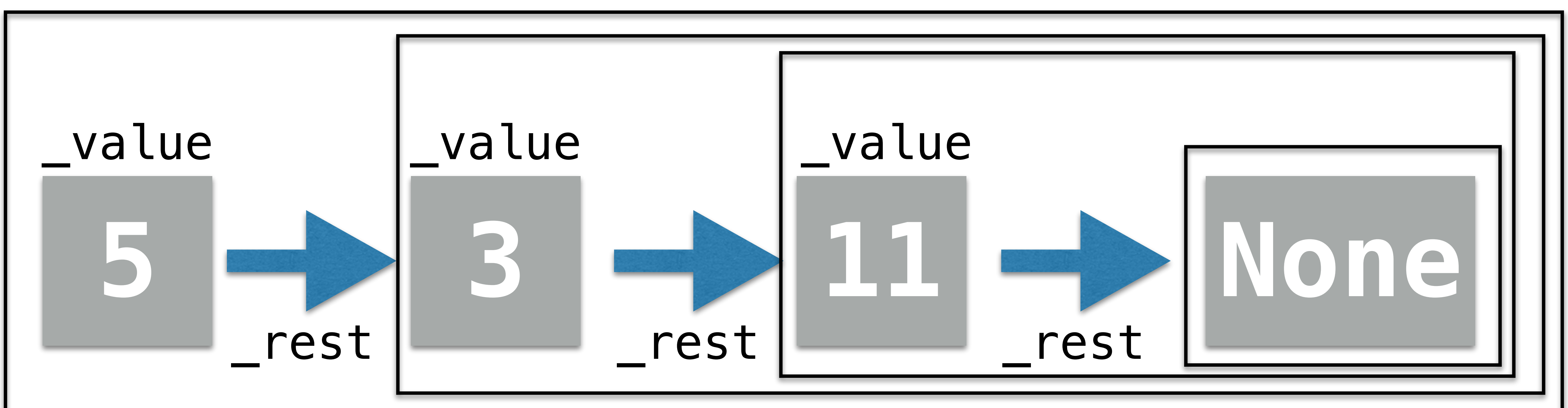
Empty Lists?

- What happens when we call print on an empty LinkedList?
- Do we want a different behavior? How do we change our code?

```
def __get_string(self):

    # handle empty list
    if self._value is None and self._rest is None:
        return '' # empty list notation

    elif self._rest is None:  # value is not None
        return str(self._value)

    else:  # neither is None
        return str(self._value) + ', ' + self._rest.__get_string()

def __str__(self):
    return "[" + self.__get_string() + "]"
```

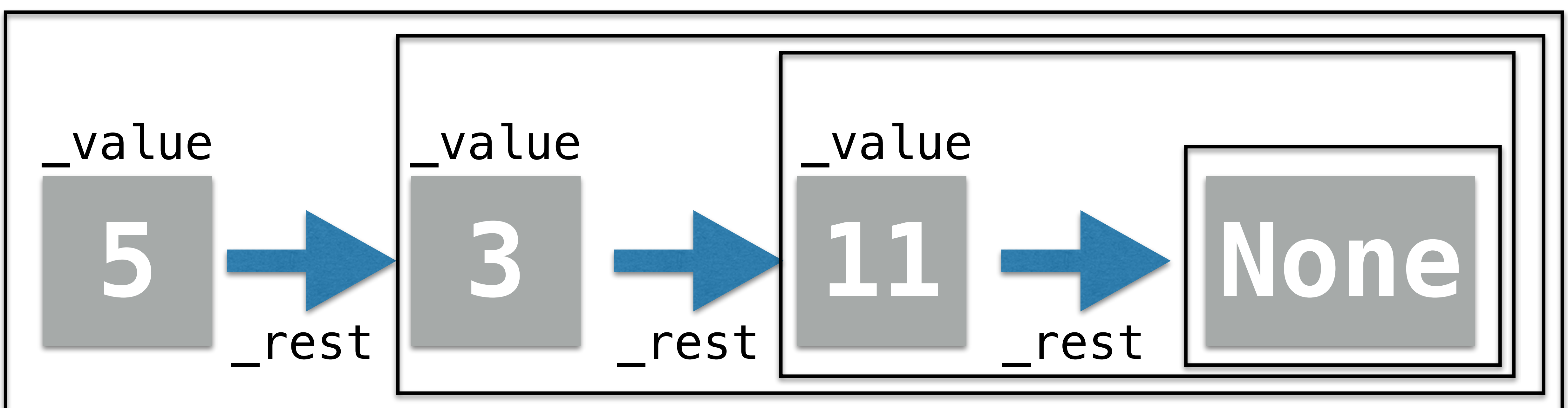
Special Method: `__len__`

- **`__len__(self)`**
  - Called when we use the built-in function `len()` in Python on an object `obj` of the class: `len(obj)`
  - We can call `len()` function on any object whose class has the `__len__()` special method implemented
- We want to implement this special method so it tells us the number of elements in our linked list, e.g. 3 elements in the list below



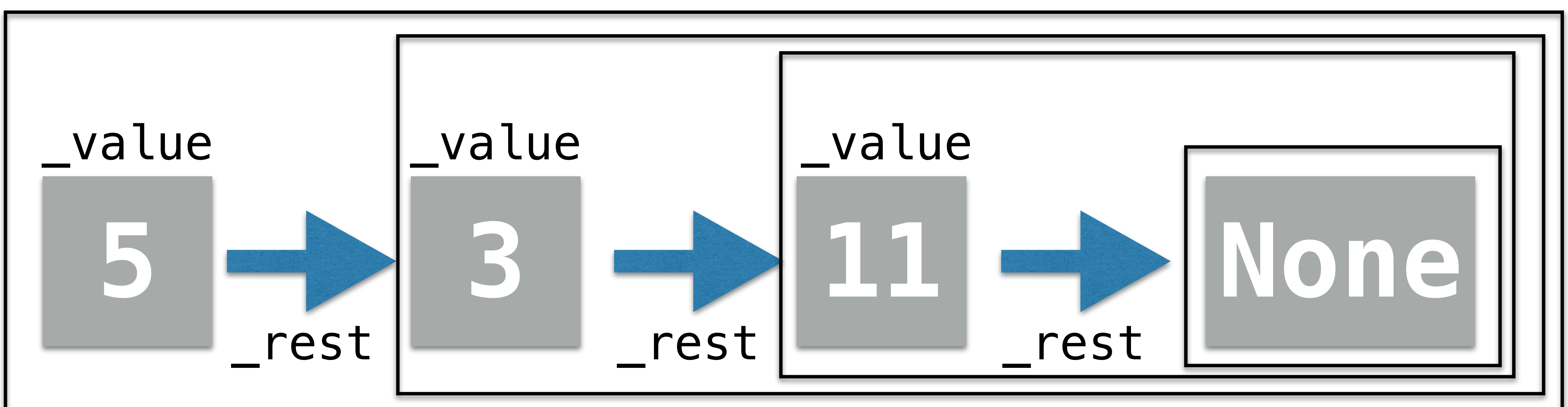
Implementing Recursively

- As our `LinkedList` class is defined recursively, let's implement the `__len__` method recursively
  - Method will return an int (num of elements)
- What is the base case(s)?
- What about the recursive case?
  - Count self (so, +1), and then call `len()` on the rest of the list!



Recursive Implementation: `__len__`

```
# len() function calls __len__() method
def __len__(self):
    # base case: handle empty list first
    if self._value is None and self._rest is None:
        return 0

    # list of length 1
    elif self._rest is None:
        return 1

    #recursive case (larger than 1)
    else:
        # same as return 1 + self._rest.__len__()
        return 1 + len(self._rest)
```



Other Special Methods

`in` Operator: `__contains__`

- `__contains__(self, val)`
  - When we say `elem in seq` in Python:
    - Python calls the `__contains__` special method on `seq`
    - That is, `seq.__contains__(elem)`
- If we want the `in` operator to work for the objects of our class, we can do so by implementing the `__contains__` special method
- Basic idea:
    - "Walk" along list checking values
    - If we find the value we're looking for, return True
    - If we make it to the end of the list without finding it, return False
    - We'll do this recursively!

`in` Operator: `__contains__`

- `__contains__(self, val)`
  - When we say `if elem in seq` in Python:
    - Python calls the `__contains__` special method on `seq`
    - That is, `seq.__contains__(elem)`
- If we want the `in` operator to work for the objects of our class, we can do so by implementing the `__contains__` special method

```
# in operator calls __contains__() method
def __contains__(self, val):
    if self._value == val:
        return True
    elif self._rest is None:
        return False
    else:
        # same as calling self.__contains__(val)
        return val in self._rest
```

Indexing [] Operator: `__getitem__`

- To support the `[]` operator to access the item at a given index in our `LinkedList`, we need to implement `__getitem__`
- Basic idea:
    - Walk out to the element at `index`
    - Get or set value at that index accordingly
    - Recursive!

Indexing [] Operator: __getitem__

- To support the [] operator to access the item at a given index in our LinkedList, we need to implement __getitem__

```
# [] list index notation calls __getitem__() method
def __getitem__(self, index):
    # if index is 0, we found the item we need to return
    if index == 0:
        return self._value
    # if reached end but index is not zero, index error
    elif index != 0 and self._rest == None:
        return 'IndexError!'
    else:
        # else we recurse until index reaches 0
        # remember that this implicitly calls __getitem__
        return self._rest[index - 1]
```

[Extra] Special Methods:

`__add__ (+), == (eq)`

+ Operator: `__add__`

- **`__add__(self, other)`**
  - When using lists, we can concatenate two lists together into one list using the + operator (this always returns a new list)
  - To support the + operator in our `LinkedList` class, we need to implement `__add__` special method
  - Make the end of our first list point to the beginning of the other
  - Basic idea:
    - Walk along first list until we reach the end
    - Set _rest to be the beginning of second list
    - More recursion!

+ Operator: `__add__`

- **`__add__(self, other)`**
  - When using lists, we can concatenate two lists together into one list using the **+** operator (this always returns a new list)
  - To support the **+** operator in our `LinkedList` class, we need to implement `__add__` special method
  - Make the end of our first list point to the beginning of the other

```
# + operator calls __add__() method
# + operator returns a new instance of LinkedList
def __add__(self, other):
    # other is another instance of LinkedList
    # if we are the last item in the list
    if self._rest is None:
        # set _rest to other
        self._rest = other
    else:
        # else, recurse until we reach the last item
        self._rest.__add__(other)
    return self
```

Note: Technically this does not return a **new** list. This is more like `extend`. Let's not worry about this for now!

`self` is the "head" or beginning of the list. Note that it didn't change!

`==` Operator: `__eq__`

- **`__eq__(self, other)`**
  - When using lists, we can compare their values using the `==` operator
  - To support the `==` operator in our `LinkedList` class, we need to implement `__eq__`
  - We want to walk the lists and check the values
  - Make sure the sizes of lists match, too

== Operator: `__eq__`

- **`__eq__(self, other)`**
  - When using lists, we can compare their values using the `==` operator
  - To support the `==` operator in our `LinkedList` class, we need to implement `__eq__`

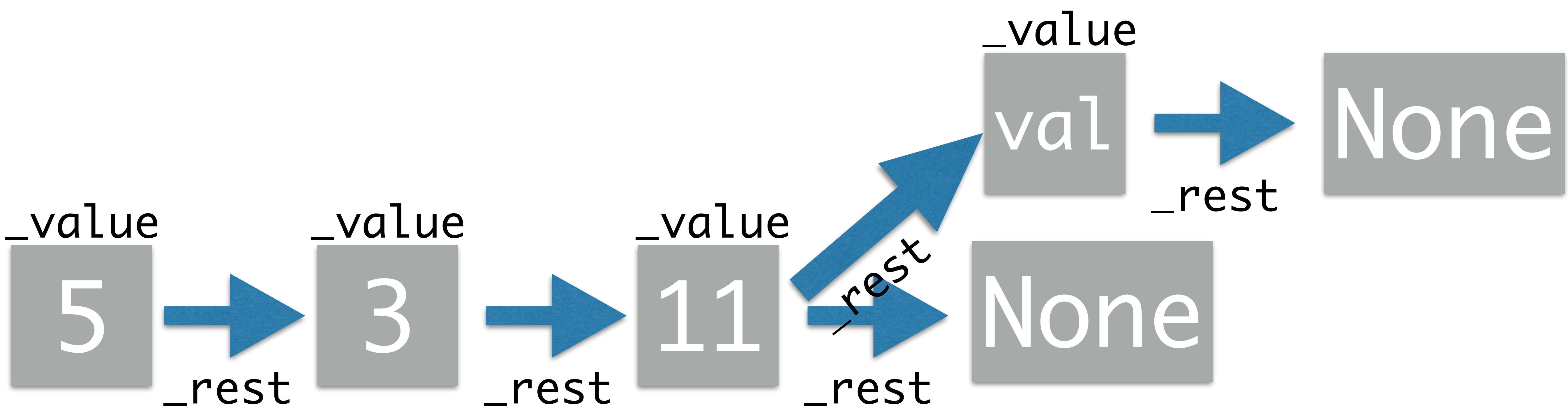
```
# == operator calls __eq__() method
def __eq__(self, other):
    # If both lists are empty
    if self._rest is None and other.get_rest() is None:
        return True

    # If both are empty, value of current list elems match
    elif self._rest is not None and other.get_rest() is not None :
        same_val = self._value == other.get_value()
        same_rest = self._rest == other.get_rest()
        return same_val and same_rest

    return False
```

Useful list methods:

`.append(), .prepend(), .insert()`

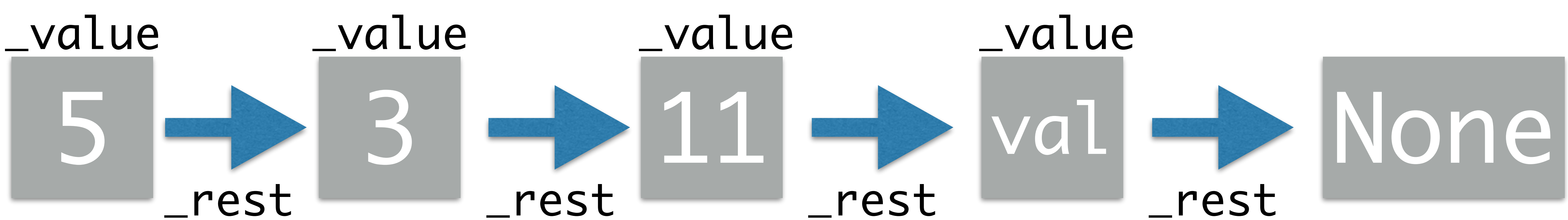
Useful List Method: **append**

- `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



Useful List Method: **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



Useful List Method: **append**

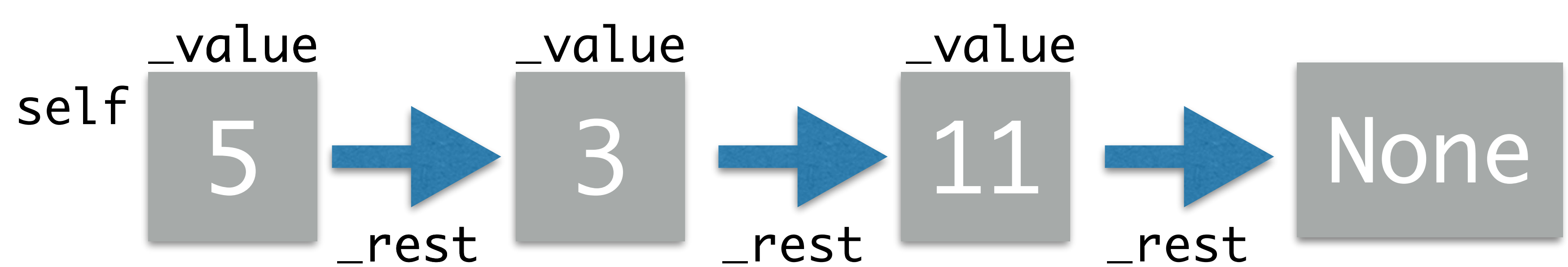
- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling **append** (note that **append** mutates our list)
  - This entails setting the `_rest` attribute of the last element to be a ***new*** LinkedList with the given value.

```
def append(self, val):
    # if am at the end of the list
    if self._rest is None:
        # add a new LinkedList to the end
        self._rest = LinkedList(val)
    else:
        # else recurse until we find the end
        self._rest.append(val)
```

Useful List Method: **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.

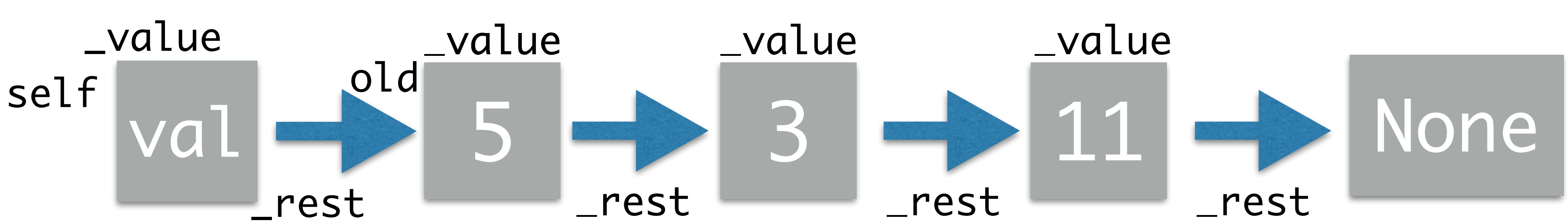
```
    def prepend(self, val):
        old_val = self._value
        old_rest = self._rest
        self._value = val
        self._rest = LinkedList(old_val, old_rest)
```



Useful List Method: `prepend`

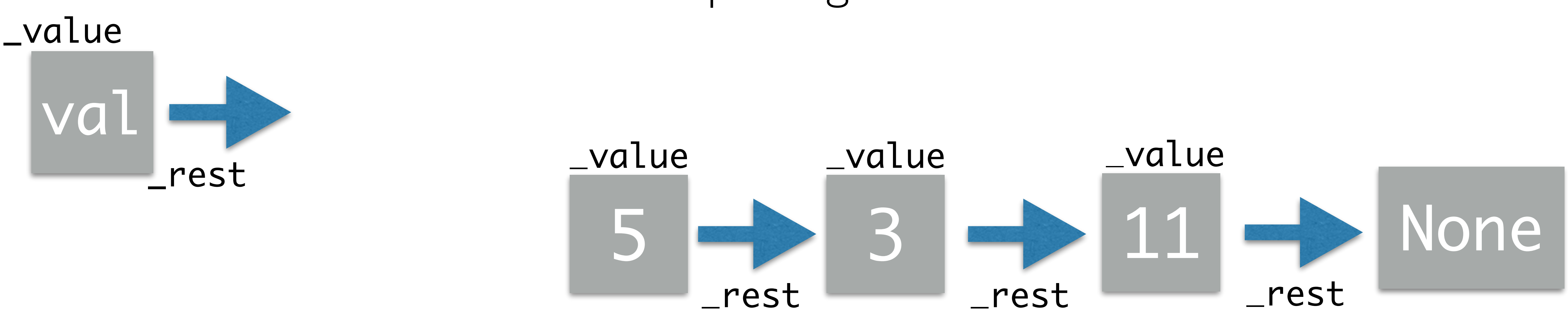
- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to **`append`**)
  - The **`prepend`** operation is really efficient, we don't need to walk through the list at all — just do some variable reassignments.

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



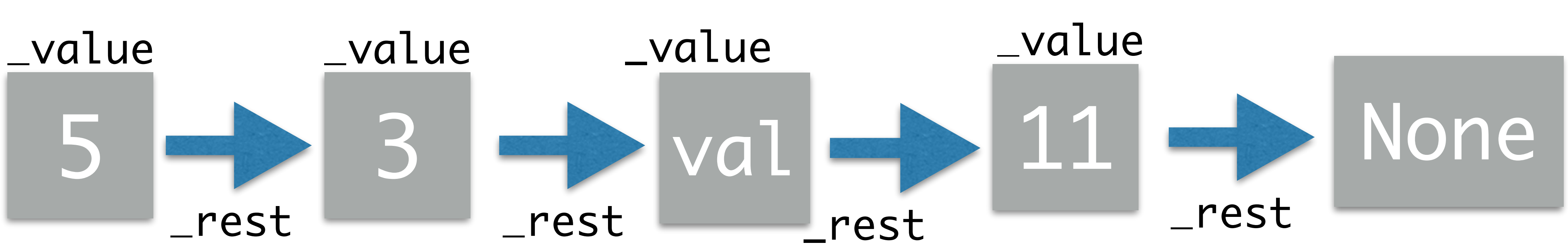
Useful List Method: `insert`

- `insert(self, val, index)`
  - Finally, we want to allow for insertions at a specific index.
  - Basic idea:
    - If the specified index is 0, we can just add to the beginning (easy!)
    - Otherwise, we walk to the appropriate index in the list, and reassign the `_rest` attribute at that location to point to a new LinkedList with the given value, and whose `_rest` attribute points to the linked list it is displacing.



Useful List Method: `insert`

- `insert(self, val, index)`
  - Finally, we want to allow for insertions at a specific index.
  - Basic idea:
    - If the specified index is 0, we can just add to the beginning (easy!)
    - Otherwise, we walk to the appropriate index in the list, and reassign the `_rest` attribute at that location to point to a new LinkedList with the given value, and whose `_rest` attribute points to the linked list it is displacing.



Useful List Method: `insert`

- `insert(self, val, index)`
  - If the specified index is 0, we can just use the `prepend` method.
  - Otherwise, check to see if we're at end of the list
  - Otherwise, we walk to the appropriate index in the list, and perform the insertion

```
def insert(self, val, index):
   # if index is 0, we found the item we need to return
    if index == 0:
        self.prepend(val)
    # elif we have reached the end, so just append
    elif self._rest is None:
        self._rest = LinkedList(val)
    # else we recurse until index reaches 0
    else:
        self._rest.insert(val, index - 1)
```

Takeaway

- Our first example of a **data structure**
  - A data structure is a specific way to organize/layout your data
  - Each data structure supports some abstract operations/methods, e.g.
    - Search for item/ membership query
    - Insert item at location
    - Delete item at location
- Different data structure may be efficient at different operations
    - E.g., among Python built-in data structures, sets are much more efficient at inserts/queries than ordered sequences
- Next time: Discuss what does **efficiency** means in Computer Science