Recursive Objects

(Revisit)
Lecture Quiz

__str__ and objects

Consider the following variable assignments. Which of the created variables can we call `__str__` on?

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name

lamb = Lamb("Fleecey")                            # 80%

grade = 79.5                                      # 80%

colors = ["red", "orange", "yellow"]              # 72%

translations = {"one": "uno", "two": "dos"} # 65%

is_fluffy = True                                  # 60%
```

__str__ and objects

Consider the following variable assignments. Which of the created variables can we call `__str__` on?

```
class Lamb:
    species_name = "Lamb"
    scientific_name = "Ovis aries"

    def __init__(self, name):
        self.name = name

lamb = Lamb("Fleecey")                        # 80%
print(Lamb.__str__(lamb))
grade = 79.5                                  # 80%
print(float.__str__(grade))
colors = ["red", "orange", "yellow"]          # 72%
print(list.__str__(colors))
translations = {"one": "uno", "two": "dos"} # 65%
print(dict.__str__(translations))
is_fluffy = True                              # 60%
print(bool.__str__(is_fluffy))
```

All of them! All of them inherit from `object` and thus have a `__str__` method defined. Run `type(var).__str__(var)` to prove it to yourself.

Attribute access

Imagine we have a class `Book` with a class variable `available_formats`, and an instance of that class named `bookywook`.

```
class Book:
    available_formats = ["Kindle", "paperback"]

    def __init__(self, title):
        self.title = title

bookywook = Book("Where's Boo?")
```

Which line of code would retrieve the value of that attribute?

```
Book.__getattribute__(bookywook, "available_formats")  # 61%

getattr(bookywook, "available_formats")                # 60%

bookywook.available_formats                            # 51%

Book.available_formats                                 # 20%

getattr(Book, "available_formats")                     # 19%
```

Attribute access

Imagine we have a class `Book` with a class variable `available_formats`, and an instance of that class named `bookywook`.

```
class Book:
    available_formats = ["Kindle", "paperback"]

    def __init__(self, title):
        self.title = title

bookywook = Book("Where's Boo?")
```

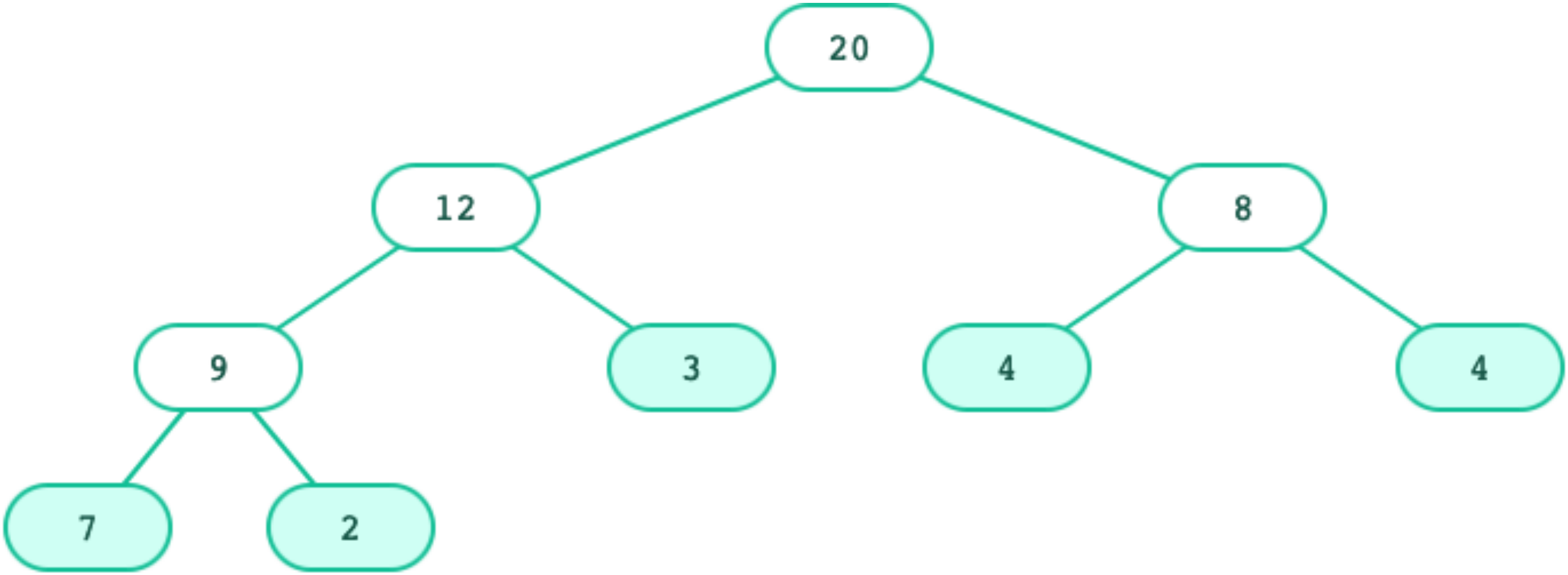
Which line of code would retrieve the value of that attribute?

```
Book.__getattribute__(bookywook, "available_formats")  # 61%

getattr(bookywook, "available_formats")                # 60%

bookywook.available_formats                            # 51%

Book.available_formats                                 # 20%

getattr(Book, "available_formats")                     # 19%
```

All of them retrieve the value!

Trees

Tree concepts



- A tree has a **root label** and a list of **branches**
- Each **branch** is itself a **tree**
- A tree with **zero branches** is called a **leaf**

Trees: Data abstraction

This is what we've been using:

`tree(label, children)`	Returns a tree with given LABEL at its root, whose children are CHILDREN
`label(tree)`	Returns the label of root node of TREE
`branches(tree)`	Returns the branches of TREE (each a tree).
`is_leaf(tree)`	Returns true if TREE is a leaf node.

Trees: Data abstraction

Using an implementation like this:

```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)
```

🤔 How could we represent trees as a Python class?

A Tree class

```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

🤔 What's different? What's the same?

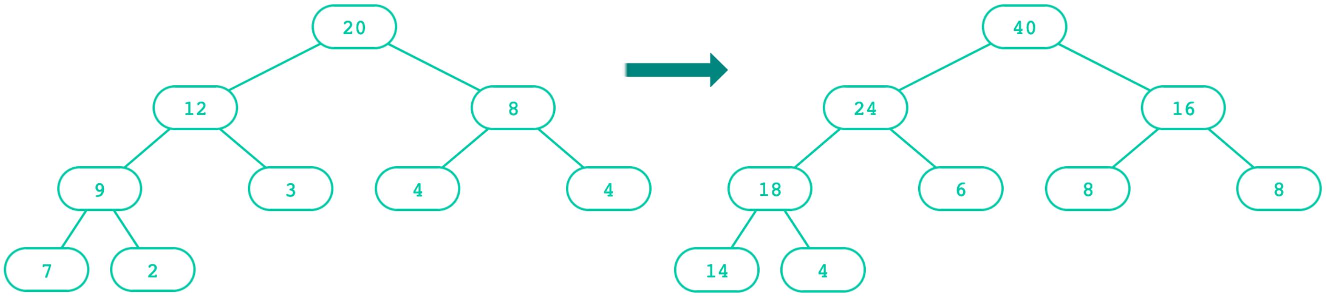
tree versus Tree

tree	Tree
`t = tree(label, branches=[])`	`t = Tree(label, branches=[])`
`branches(t)`	`t.branches`
`label(t)`	`t.label`
`is_leaf(t)`	`t.is_leaf()`

```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n - 2)
        right = fib_tree(n - 1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```

Doubling a Tree

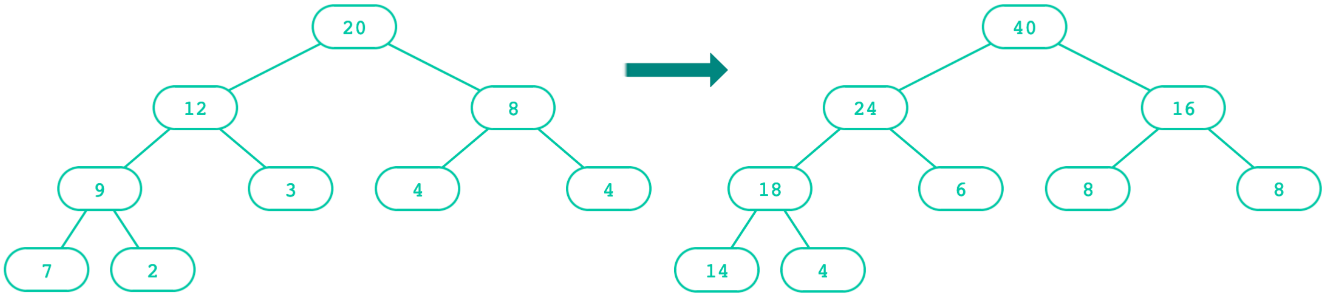



```
def double(t):
    """Doubles every label in T, mutating T.
    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(6, [Tree(10)]), Tree(14)])
    """
    t.label = t.label * 2
    if not t.is_leaf():
        for b in t.branches:
            double(b)
```

Is the `Tree` object mutable or immutable?

Is `double(t)` destructive or non-destructive?

Doubling a Tree

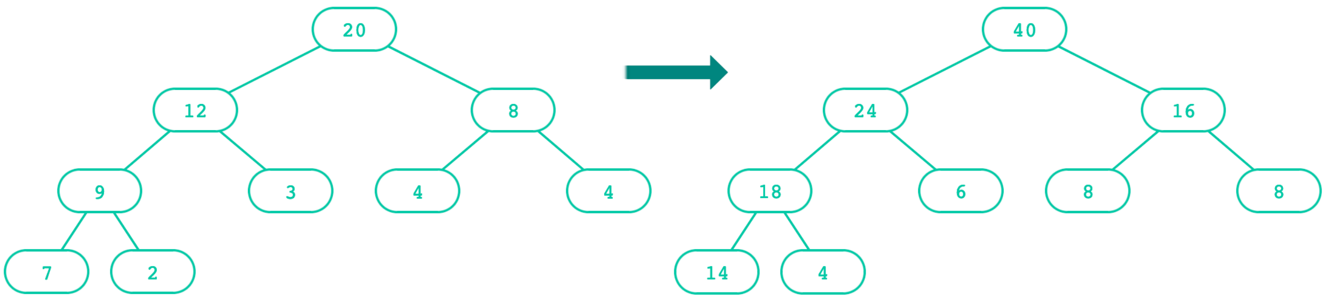



```python
def double(t):
    """Doubles every label in T, mutating T.
    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(6, [Tree(10)]), Tree(14)])
    """
    t.label = t.label * 2
    if not t.is_leaf():
        for b in t.branches:
            double(b)
```

Is the `Tree` object mutable or immutable? Mutable!

Is `double(t)` destructive or non-destructive?

Doubling a Tree




```
def double(t):
    """Doubles every label in T, mutating T.
    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> double(t)
    >>> t
    Tree(2, [Tree(6, [Tree(10)]), Tree(14)])
    """
    t.label = t.label * 2
    if not t.is_leaf():
        for b in t.branches:
            double(b)
```

Is the `Tree` object mutable or immutable? Mutable!

Is `double(t)` destructive or non-destructive? Destructive!

A fancier Tree

This is what assignments actually use:

```
class Tree:

    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append('  ' + line)
        return [str(self.label)] + lines
```

It's built in to code.cs61a.org, and remember, you can `draw()` any tree/Tree.

Linked lists

Why do we need a new list?

Python lists are implemented as a "dynamic array", which isn't optimal for all use cases.

😭 Inserting an element is slow, especially near front of list:

"A"	"B"	"C"	"D"	"E"	"F"
0	1	2	3	4	5
3300	3301	3302	3303	3304	3305

What should we insert?

value: Z @ index: 3 Insert

Why do we need a new list?

Python lists are implemented as a "dynamic array", which isn't optimal for all use cases.

😭 Inserting an element is slow, especially near front of list:

"A"	"B"	"C"	"D"	"E"	"F"
0	1	2	3	4	5
3300	3301	3302	3303	3304	3305

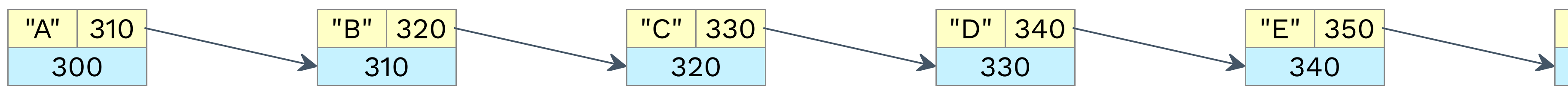
What should we insert?

value: Z @ index: 3 Insert

😭 Plus inserting too many elements can require re-creating the entire list in memory, if it exceeds the pre-allocated memory.

Linked lists

A linked list is a chain of objects where each object holds a **value** and a **reference to the next link**. The list ends when the final reference is empty.

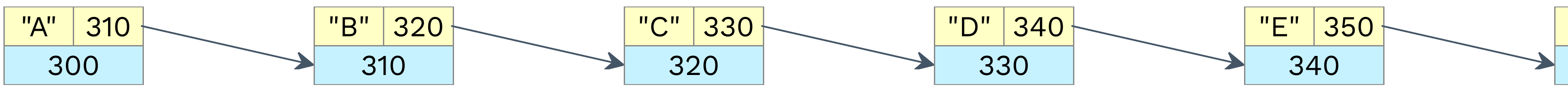


What should we insert?

value: Z @ index: 5 Insert

Linked lists

A linked list is a chain of objects where each object holds a **value** and a **reference to the next link**. The list ends when the final reference is empty.



What should we insert?

value: Z @ index: 5 Insert

Linked lists require more space but provide faster insertion.

A Link class

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

How would we use that?

A Link class

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
```

How would we use that?

```
ll = Link("A", Link("B", Link("C")))
```

Try in PythonTutor

A fancier LinkedList

```
class Link:
    """A linked list."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

It's built-in to code.cs61a.org and you can `draw()` any Link.

Creating linked lists

Creating a range

Similar to `[x for x in range(3, 6)]`

```
def range_link(start, end):
    """Return a Link containing consecutive integers
    from START to END, not including END.
    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
```

Try in PythonTutor

Creating a range

Similar to `[x for x in range(3, 6)]`

```
def range_link(start, end):
    """Return a Link containing consecutive integers
    from START to END, not including END.
    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    if start >= end:
        return Link.empty
    return Link(start, range_link(start + 1, end))
```

Try in PythonTutor

Mapping a linked list

Similar to `[f(x) for x in lst]`

```
def map_link(f, ll):
    """Return a Link that contains f(x) for each x in Link LL.
    >>> square = lambda x: x * x
    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """
```

Try in PythonTutor

Mapping a linked list

Similar to `[f(x) for x in lst]`

```
def map_link(f, ll):
    """Return a Link that contains f(x) for each x in Link LL.
    >>> square = lambda x: x * x
    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """
    if ll is Link.empty:
        return Link.empty
    return Link(f(ll.first), map_link(f, ll.rest))
```

Try in PythonTutor

Filtering a linked list

Similar to `[x for x in lst if f(x)]`

```
def filter_link(f, ll):
    """Return a Link that contains only the elements x of Link LL
    for which f(x) is a true value.
    >>> is_odd = lambda x: x % 2 == 1
    >>> filter_link(is_odd, range_link(3, 6))
    Link(3, Link(5))
    """
```

Try in PythonTutor

Filtering a linked list

Similar to `[x for x in lst if f(x)]`

```
def filter_link(f, ll):
    """Return a Link that contains only the elements x of Link LL
    for which f(x) is a true value.
    >>> is_odd = lambda x: x % 2 == 1
    >>> filter_link(is_odd, range_link(3, 6))
    Link(3, Link(5))
    """
    if ll is Link.empty:
        return Link.empty
    elif f(ll.first):
        return Link(ll.first, filter_link(f, ll.rest))
    return filter_link(f, ll.rest)
```

Try in PythonTutor

Mutating linked lists

Linked lists can change

Attribute assignments can change `first` and `rest` attributes of a `Link`.

```
s = Link("A", Link("B", Link("C")))
s.first = "Hi"
s.rest.first = "Hola"
s.rest.rest.first = "Oi"
```

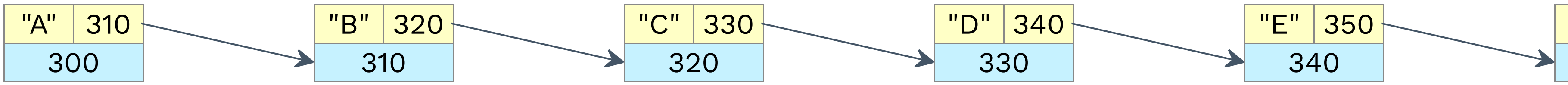
Try in PythonTutor

Beware infinite lists

The rest of a linked list can contain the linked list as a sub-list.

```
s = Link("A", Link("B", Link("C")))
t = s.rest
t.rest = s

s.first
s.rest.rest.rest.rest.rest.first
```

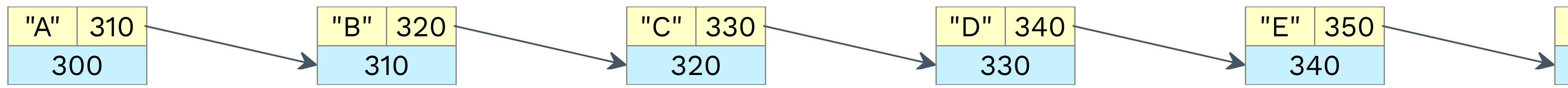
Adding to front of linked list



Insert

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,
    returning new linked list.

    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
```

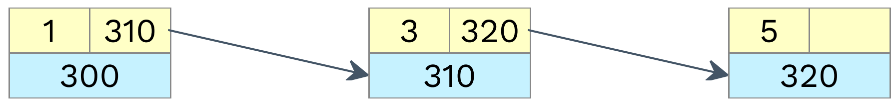
Adding to front of linked list



Insert

```
def insert_front(linked_list, new_val):
    """Inserts NEW_VAL in front of LINKED_LIST,
    returning new linked list.

    >>> ll = Link(1, Link(3, Link(5)))
    >>> insert_front(ll, 0)
    Link(0, Link(1, Link(3, Link(5))))
    """
    return Link(new_val, linked_list)
```

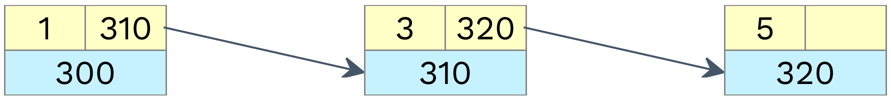
Adding to an ordered linked list



Insert value: 0 @ index: 0 Insert

```
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:



    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:

    elif new_val > ordered_list.first:

    return ordered_list
```

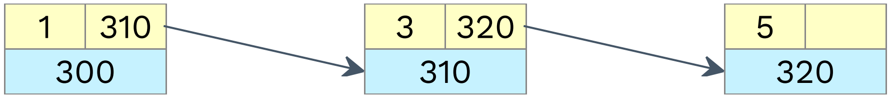
Adding to an ordered linked list



Insert value: 0 @ index: 0 Insert

```
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:

    elif new_val > ordered_list.first:

    return ordered_list
```

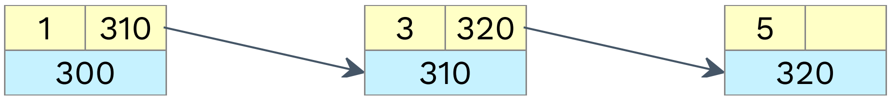
Adding to an ordered linked list



Insert value: 0 @ index: 0 Insert

```
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        ordered_list.rest = Link(new_val)
    elif new_val > ordered_list.first:

    return ordered_list
```

Adding to an ordered linked list



Insert value: 0 @ index: 0 Insert

```
def add(ordered_list, new_val):
    """Add NEW_VAL to ORDERED_LIST, returning modified ORDERED_LIST.
    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))))))
    """
    if new_val < ordered_list.first:
        original_first = ordered_list.first
        ordered_list.first = new_val
        ordered_list.rest = Link(original_first, ordered_list.rest)
    elif new_val > ordered_list.first and ordered_list.rest is Link.empty:
        ordered_list.rest = Link(new_val)
    elif new_val > ordered_list.first:
        add(ordered_list.rest, new_val)
    return ordered_list
```

Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

Version	10,000 runs	100,000 runs
Python list		
Link		

Try it yourself on your local machine (Legit Python!):
warandpeace.py

Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

Version	10,000 runs	100,000 runs
Python list	2.6 seconds	37 seconds
Link		

Try it yourself on your local machine (Legit Python!):
warandpeace.py

Showdown: Python list vs. Link

The challenge:

- Store all the half-a-million words in "War and Peace"
- Insert a word at the beginning.

Version	**10,000 runs**	**100,000 runs**
Python list	2.6 seconds	37 seconds
Link	0.01 seconds	0.1

Try it yourself on your local machine (Legit Python!): warandpeace.py

Recursive objects

Why are `Tree` and `Link` considered recursive objects?

Recursive objects

Why are `Tree` and `Link` considered recursive objects?

Each type of object contains references to the same type of object.

- An instance of `Tree` can contain additional instances of `Tree`, in the `branches` variable.
- An instance of `Link` can contain an additional instance of `Link`, in the `rest` variable.

Both classes lend themselves to recursive algorithms. Generally:

- For `Tree`: The base case is when `is_leaf()` is true; the recursive call is on the `branches`.
- For `Link`: The base case is when the rest is `empty`; the recursive call is on the `rest`.