61A Lecture 22

Wednesday, October 17

Closure Property of Data

Closure Property of Data

A tuple can contain another tuple as an element.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

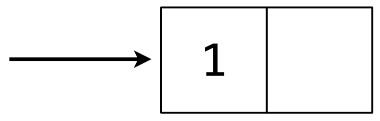
Recursive list representation of the sequence 1, 2, 3, 4:

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

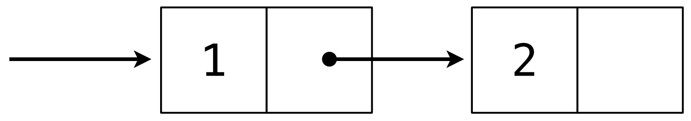


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

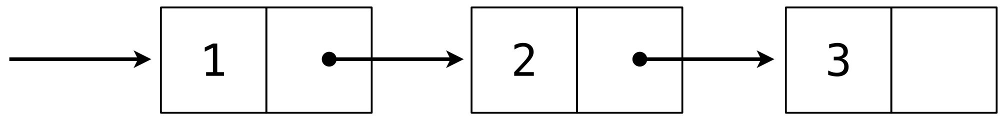


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

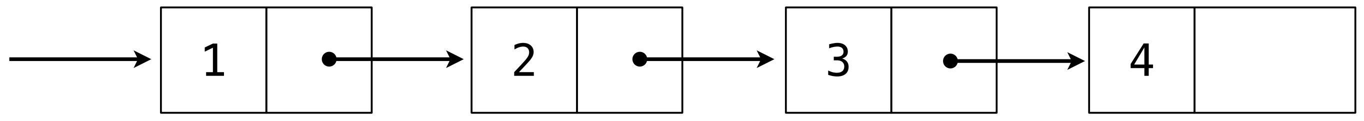


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

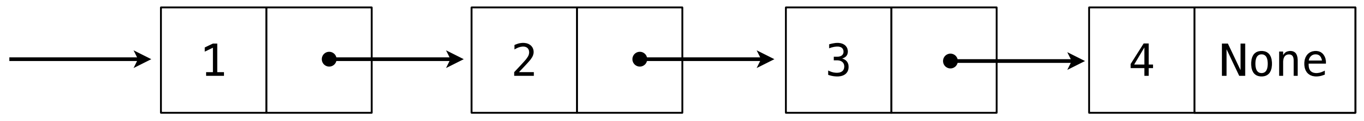


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

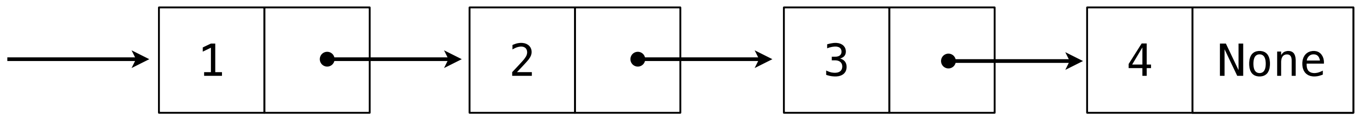


Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



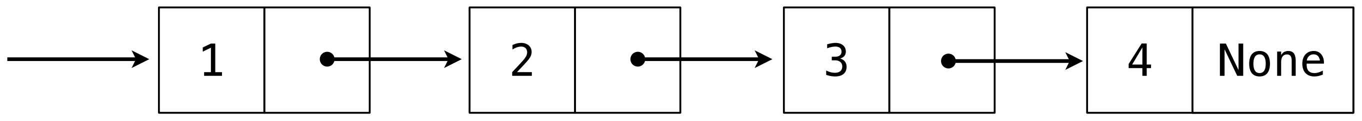
Recursive lists are recursive: the rest of the list is a list.

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



Recursive lists are recursive: the rest of the list is a list.

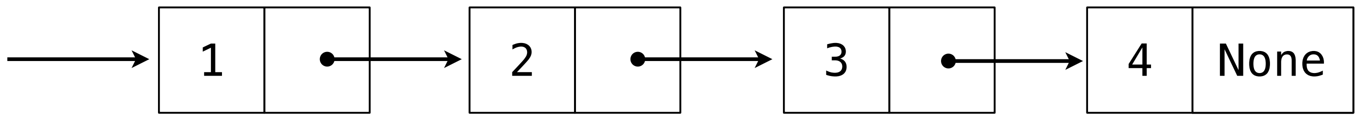
Nested Tuples (old):

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



Recursive lists are recursive: the rest of the list is a list.

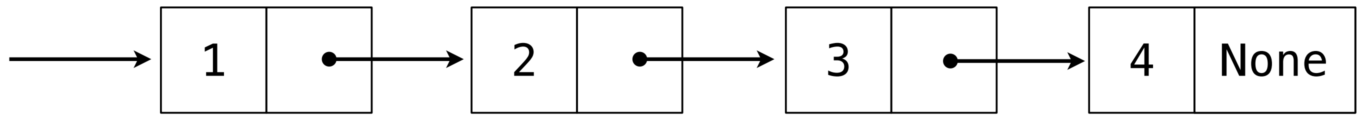
Nested Tuples (old):   (1, (2, (3, (4, None))))

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



Recursive lists are recursive: the rest of the list is a list.

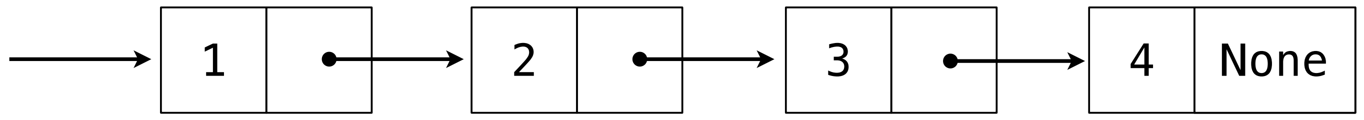
Nested Tuples (old): (1, (2, (3, (4, None)))) Demo

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



Recursive lists are recursive: the rest of the list is a list.

Nested Tuples (old): (1, (2, (3, (4, None)))) Demo

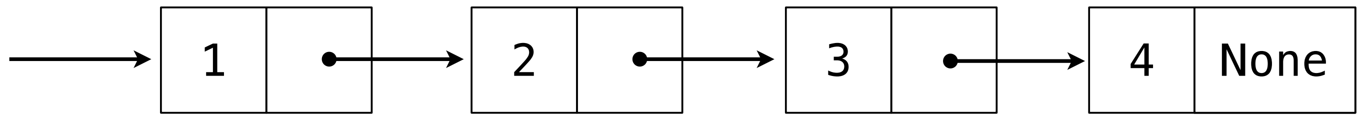
Rlist class (new):

Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:



Recursive lists are recursive: the rest of the list is a list.

Nested Tuples (old): `(1, (2, (3, (4, None))))` Demo

Rlist class (new): `Rlist(1, Rlist(2, Rlist(3, Rlist(4))))`

Recursive List Class

Recursive List Class

```
class Rlist(object):
```

Recursive List Class

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()
```

Recursive List Class

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

Recursive List Class

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

Recursive List Class

Methods can be recursive as well!

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

Recursive List Class

Methods can be recursive as well!

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

    def __len__(self):
        return 1 + len(self.rest)
```

Recursive List Class

Methods can be recursive as well!

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

```
    def __len__(self):
        return 1 + len(self.rest)
```

Yes, this call is recursive

Recursive List Class

Methods can be recursive as well!

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __len__(self):
        return 1 + len(self.rest)
```

There's the base case!

Yes, this call is recursive

Recursive List Class

Methods can be recursive as well!

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
```

There's the base case!

Yes, this call is recursive

Recursive List Class

Methods can be recursive as well!

This part was all in Homework 6

```
class Rlist(object):

    class EmptyList(object):
        def __len__(self):
            return 0

    empty = EmptyList()

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

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
```

There's the base case!

Yes, this call is recursive

Demo

Recursive Operations on Recursive Lists

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))


def extend_rlist(s1, s2):
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

```
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```

Demo

Map and Filter on Recursive Lists

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
        if fn(s.first):
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
        if fn(s.first):
            return Rlist(s.first, rest)
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
        if fn(s.first):
            return Rlist(s.first, rest)
        return rest
```

Map and Filter on Recursive Lists

We want operations on a whole list, not an element at a time.

```
>>> def map_rlist(s, fn):
        if s is Rlist.empty:
            return s
        return Rlist(fn(s.first), map_rlist(s.rest, fn))

>>> def filter_rlist(s, fn):
        if s is Rlist.empty:
            return s
        rest = filter_rlist(s.rest, fn)
        if fn(s.first):
            return Rlist(s.first, rest)
        return rest
```

Demo

Tree Structured Data

Tree Structured Data

Nested Sequences are Hierarchical Structures.

Tree Structured Data

Nested Sequences are Hierarchical Structures.

```
((1, 2), (3, 4), 5)
```

Tree Structured Data

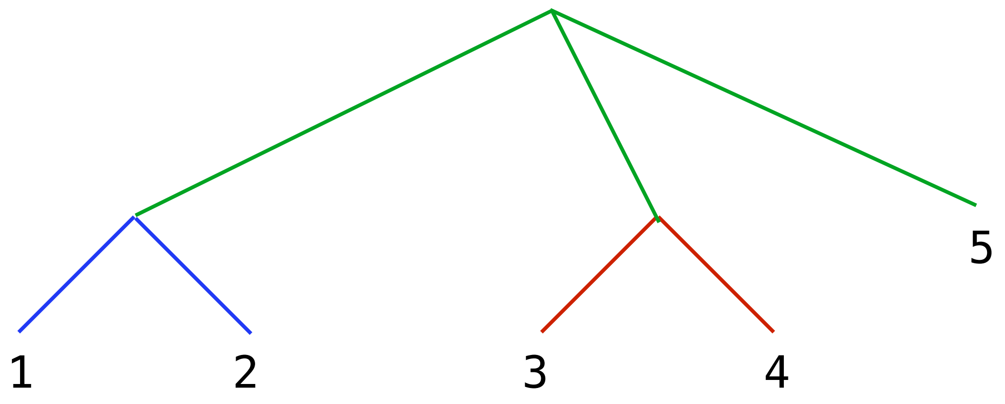
```
Nested Sequences are Hierarchical Structures.
```

```
((1, 2), (3, 4), 5)
```

Tree Structured Data

Nested Sequences are Hierarchical Structures.

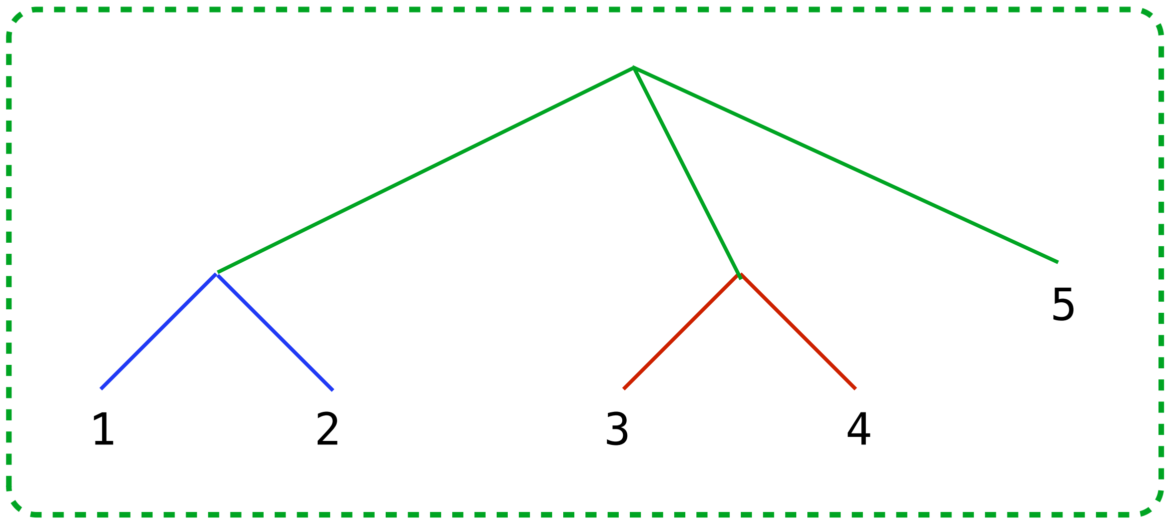
((1, 2), (3, 4), 5)



Tree Structured Data

Nested Sequences are Hierarchical Structures.

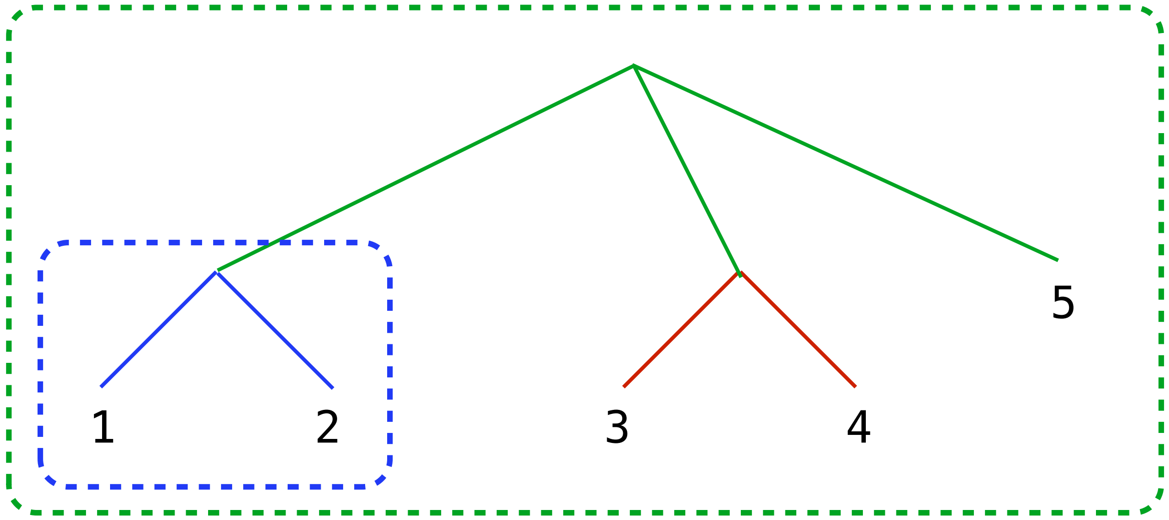
((1, 2), (3, 4), 5)



Tree Structured Data

Nested Sequences are Hierarchical Structures.

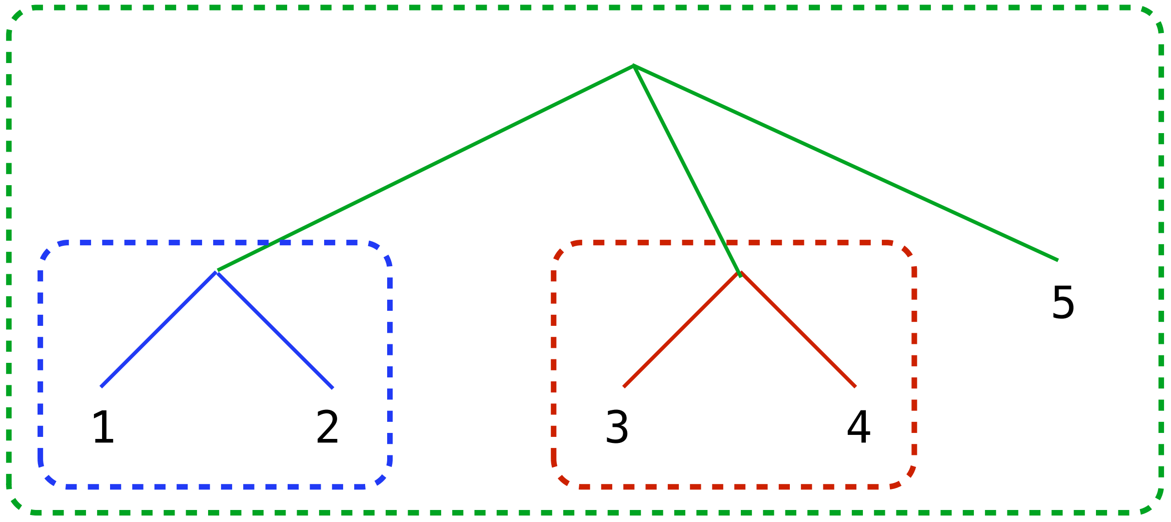
```
((1, 2), (3, 4), 5)
```



Tree Structured Data

`Nested Sequences are Hierarchical Structures.`

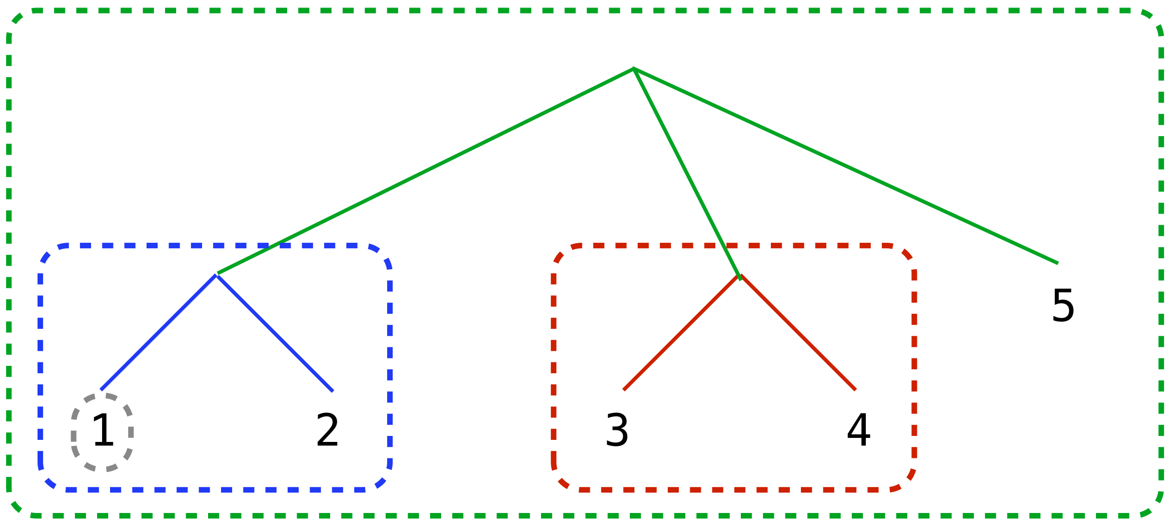
`((1, 2), (3, 4), 5)`



Tree Structured Data

Nested Sequences are Hierarchical Structures.

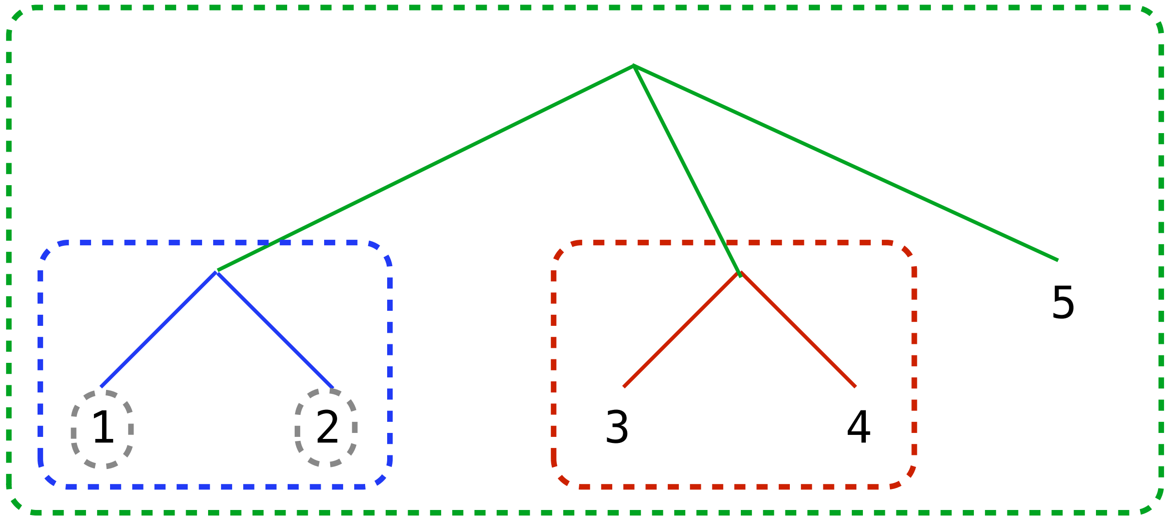
((1, 2), (3, 4), 5)



Tree Structured Data

Nested Sequences are Hierarchical Structures.

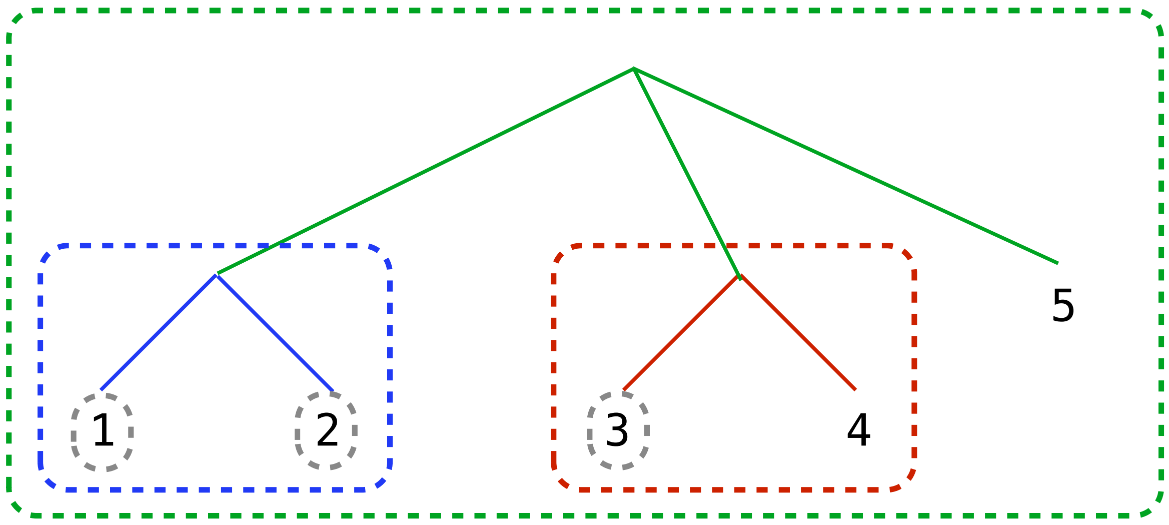
((1, 2), (3, 4), 5)



Tree Structured Data

Nested Sequences are Hierarchical Structures.

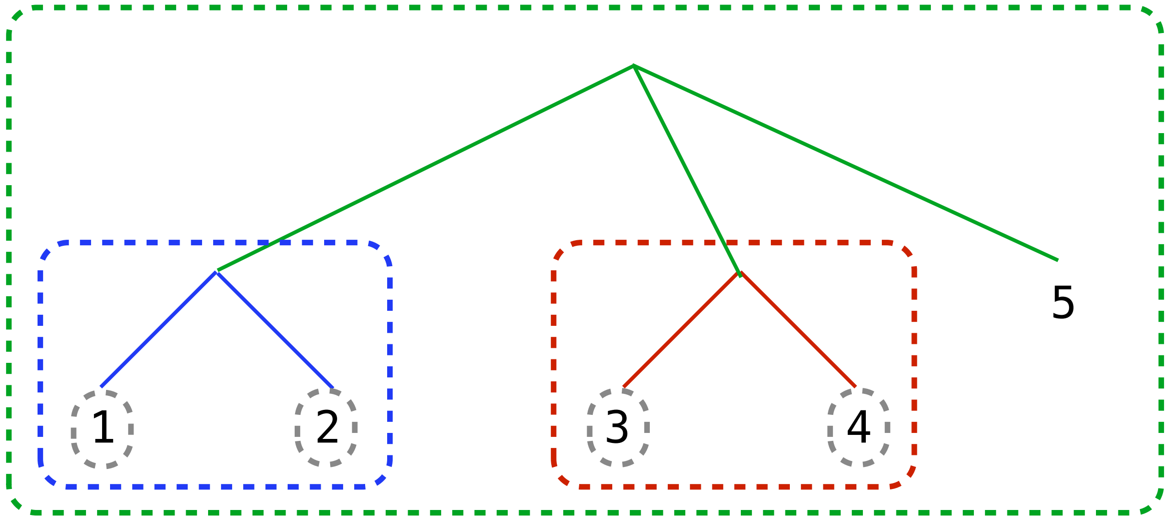
```
((1, 2), (3, 4), 5)
```



Tree Structured Data

Nested Sequences are Hierarchical Structures.

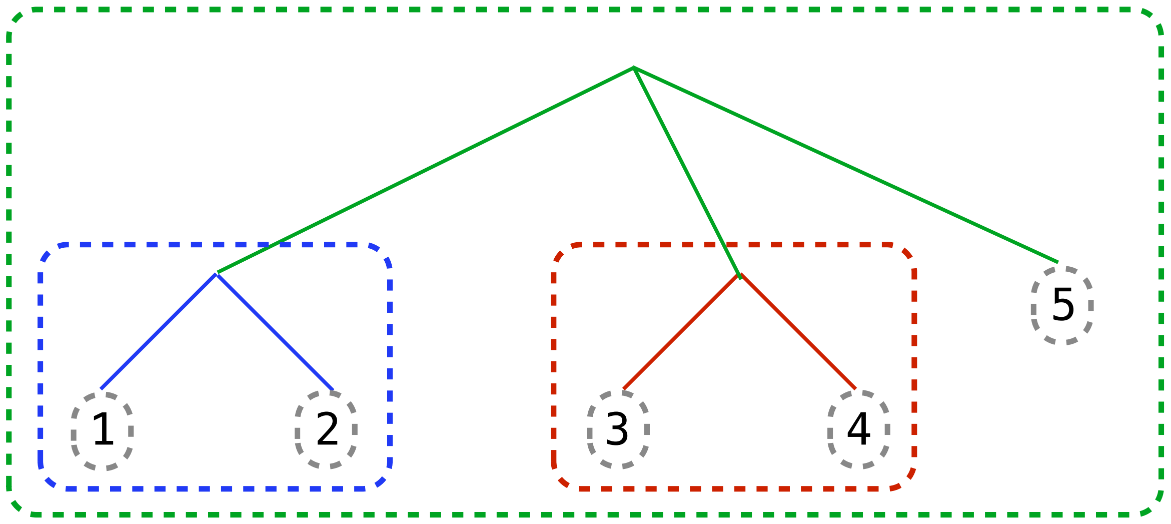
```
((1, 2), (3, 4), 5)
```



Tree Structured Data

Nested Sequences are Hierarchical Structures.

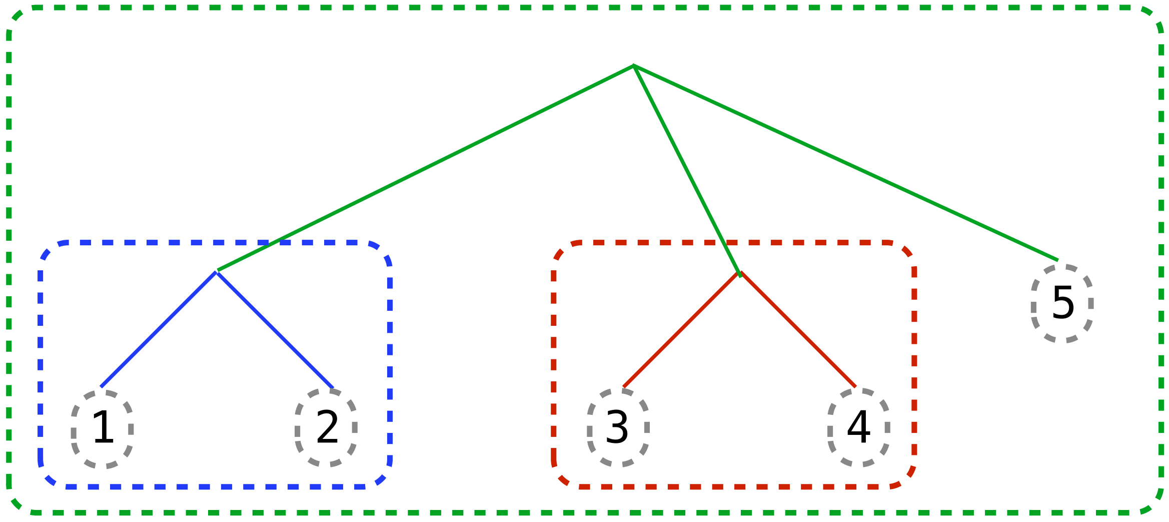
```
((1, 2), (3, 4), 5)
```



Tree Structured Data

Nested Sequences are Hierarchical Structures.

((1, 2), (3, 4), 5)



In every tree, a vast forest

Recursive Tree Processing

Recursive Tree Processing

Tree operations typically make recursive calls on branches

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```

```
def map_tree(tree, fn):
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```

```
def map_tree(tree, fn):
    if type(tree) != tuple:
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```

```
def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))


def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn) for branch in tree)
```

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```

```
def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn) for branch in tree)
```

Demo

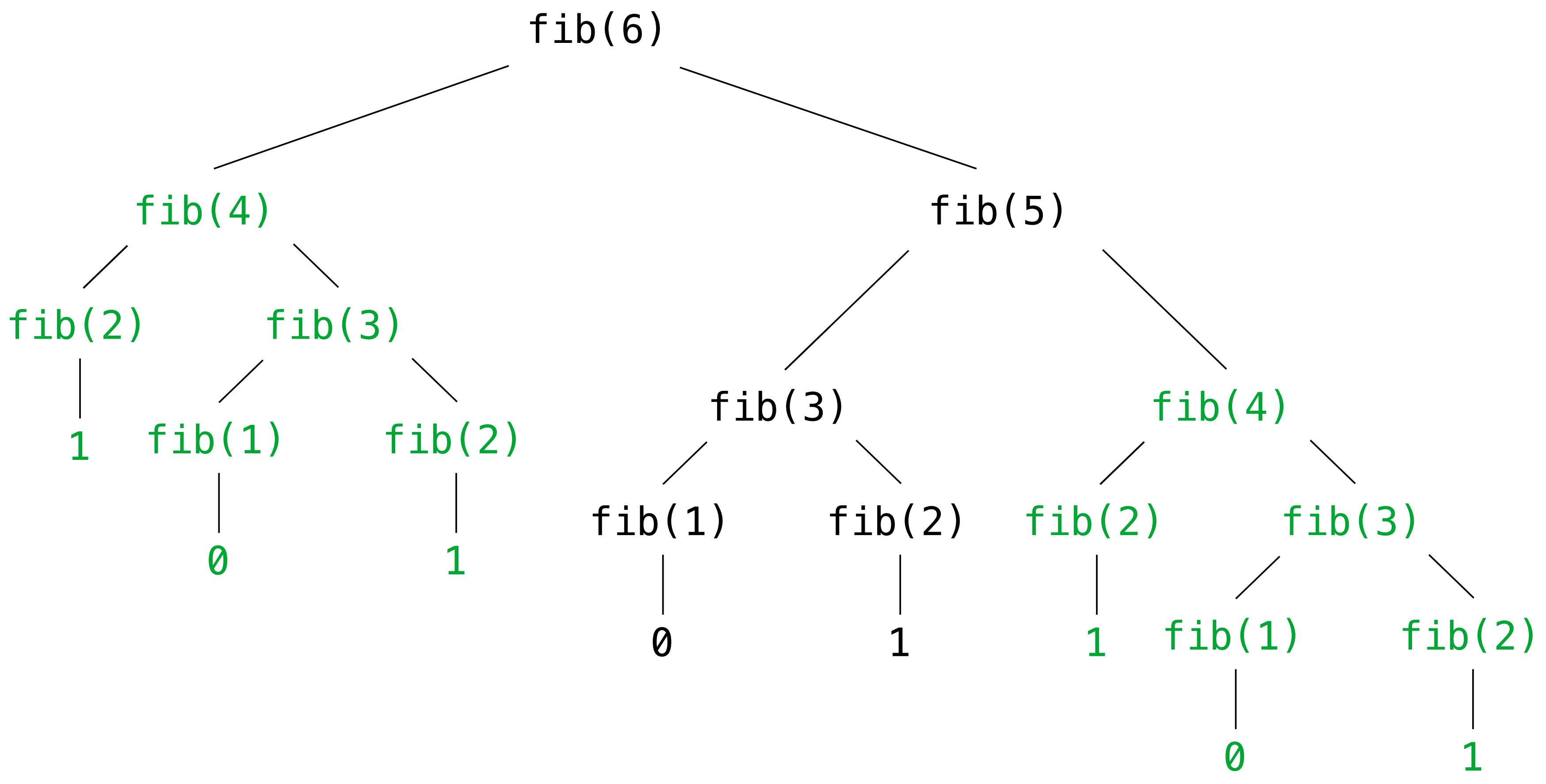
Trees with Internal Node Values

Trees with Internal Node Values

Trees can have values at their roots as well as their leaves.

Trees with Internal Node Values

Trees can have values at their roots as well as their leaves.



Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
    class Tree(object):
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
    return Tree(left.entry + right.entry, left, right)
```

Trees with Internal Node Values (Entries)

Trees need not only have values at their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right
```

Demo

```
def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
    return Tree(left.entry + right.entry, left, right)
```

Sets

Sets

One more built-in Python container type

Sets

One more built-in Python container type

- Set literals are enclosed in braces

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
```

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}


>>> 3 in s
True
```

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}


>>> 3 in s
True
>>> len(s)
4
```

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}


>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
```

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}


>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```

Sets

One more built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}


>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```

Demo