

Composition

Announcements

Linked Lists

Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

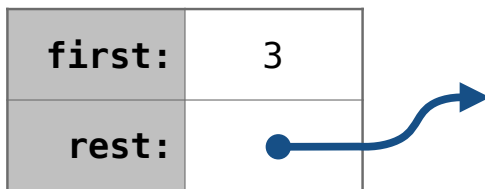
3 , 4 , 5

Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5

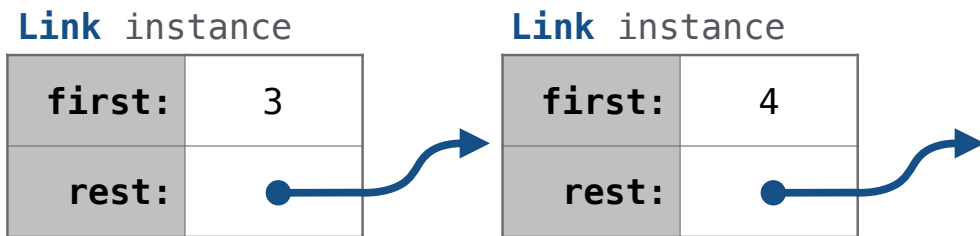
Link instance



Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

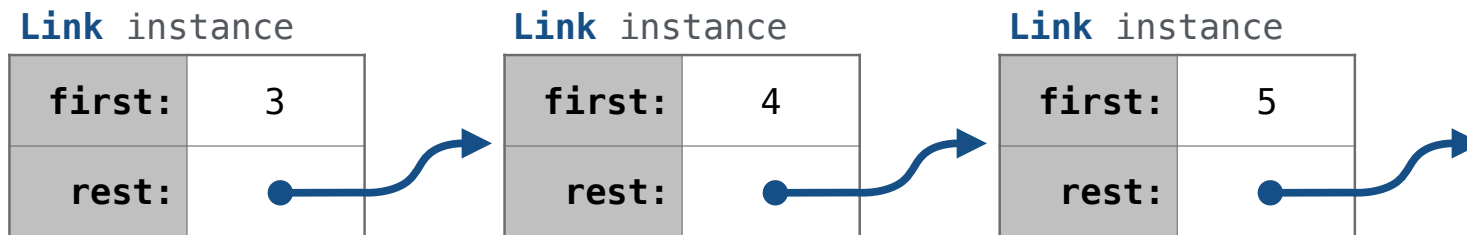
3 , 4 , 5



Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

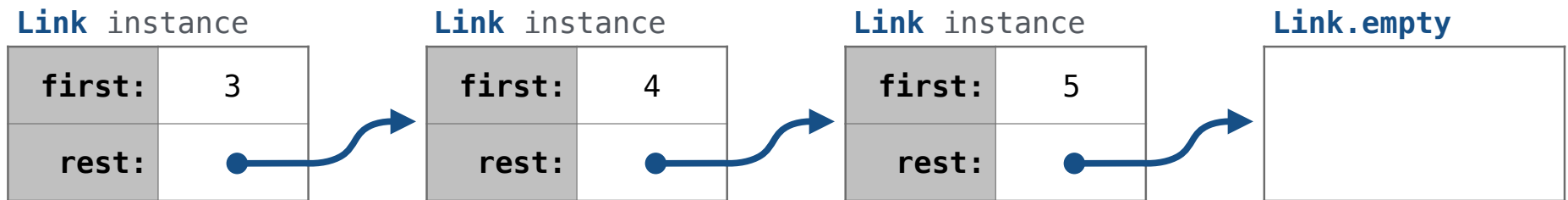
3 , 4 , 5



Linked List Structure

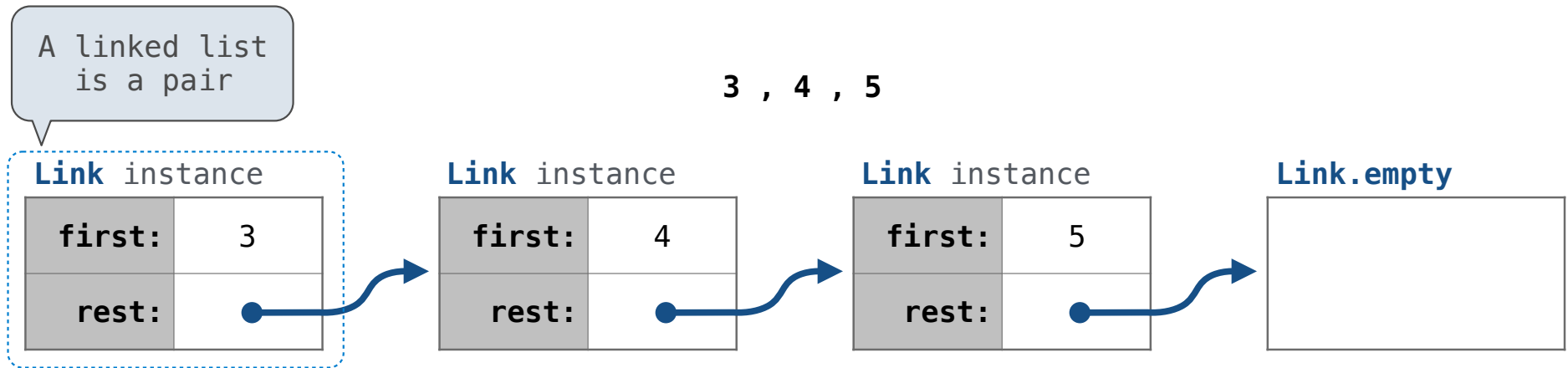
A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5



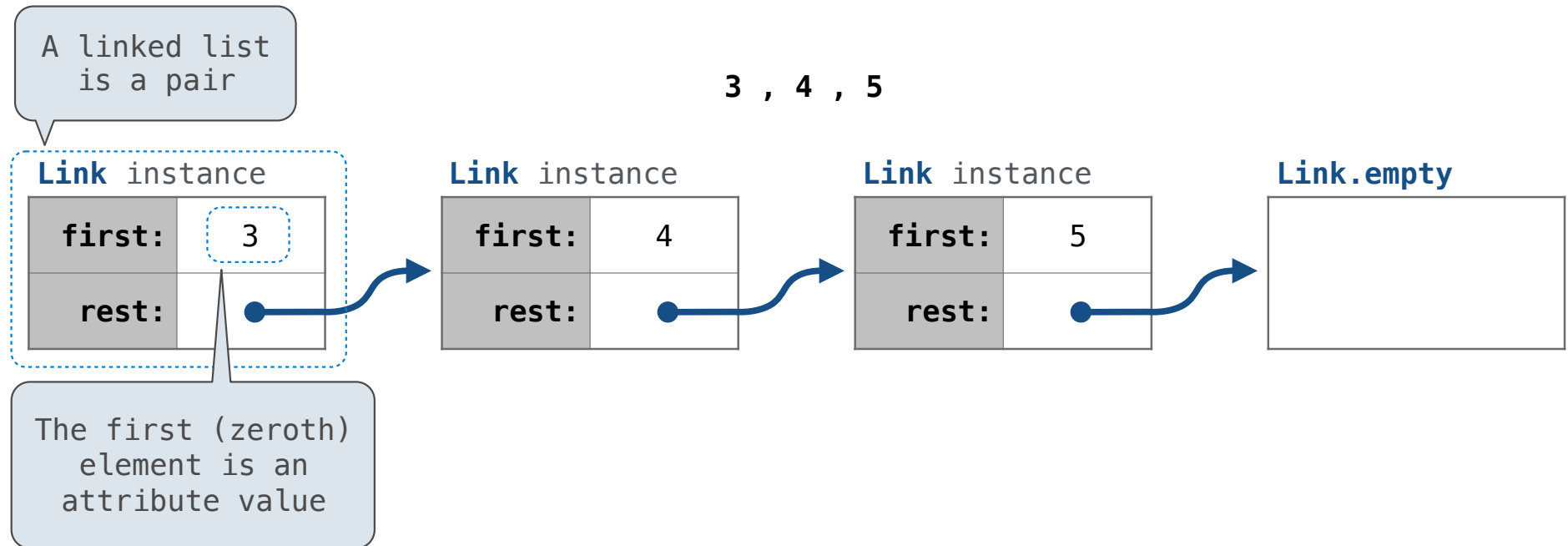
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



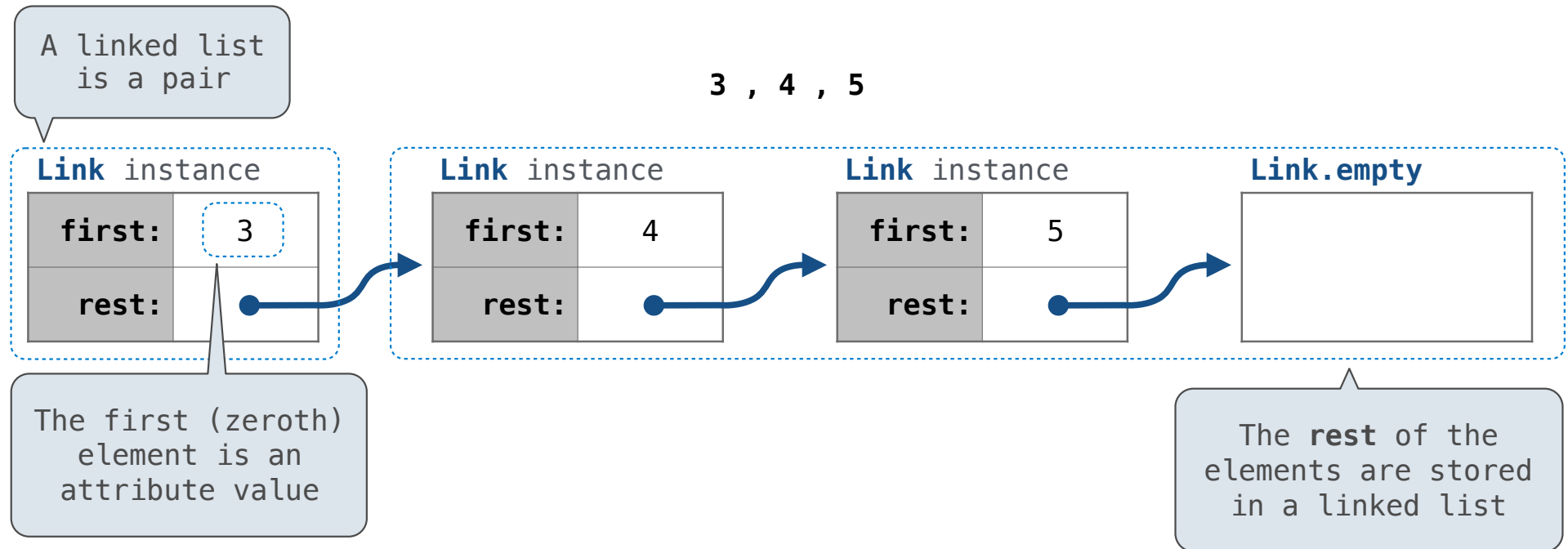
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



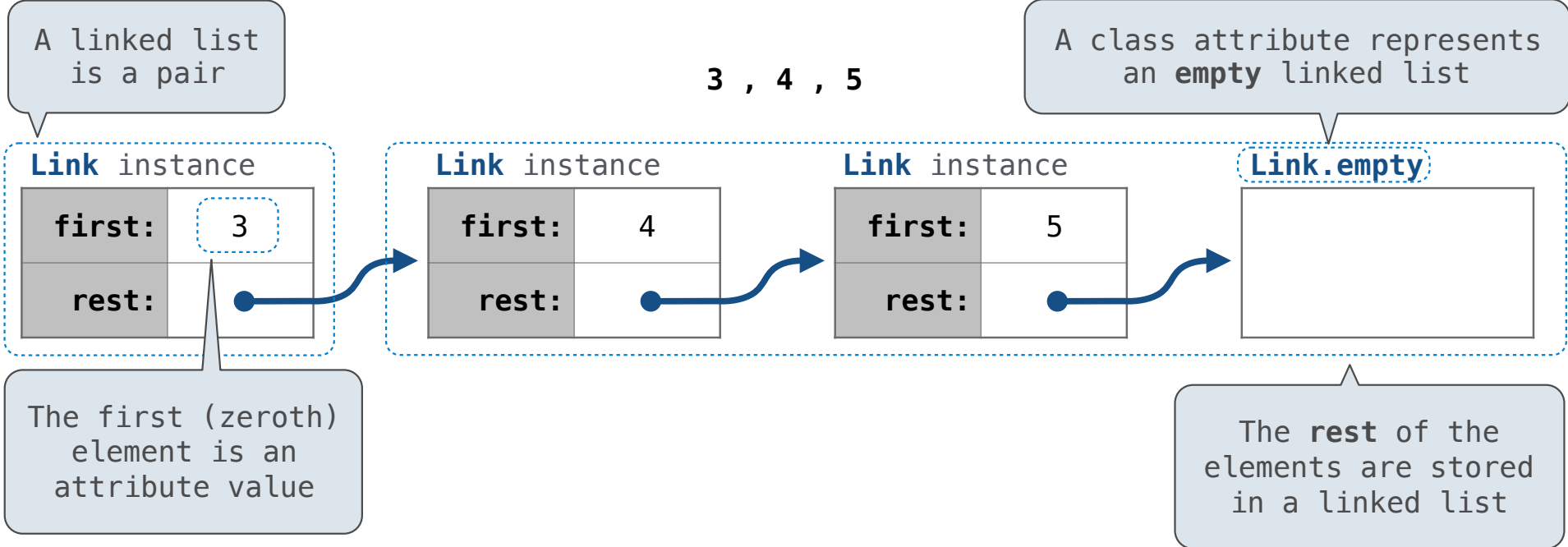
Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list



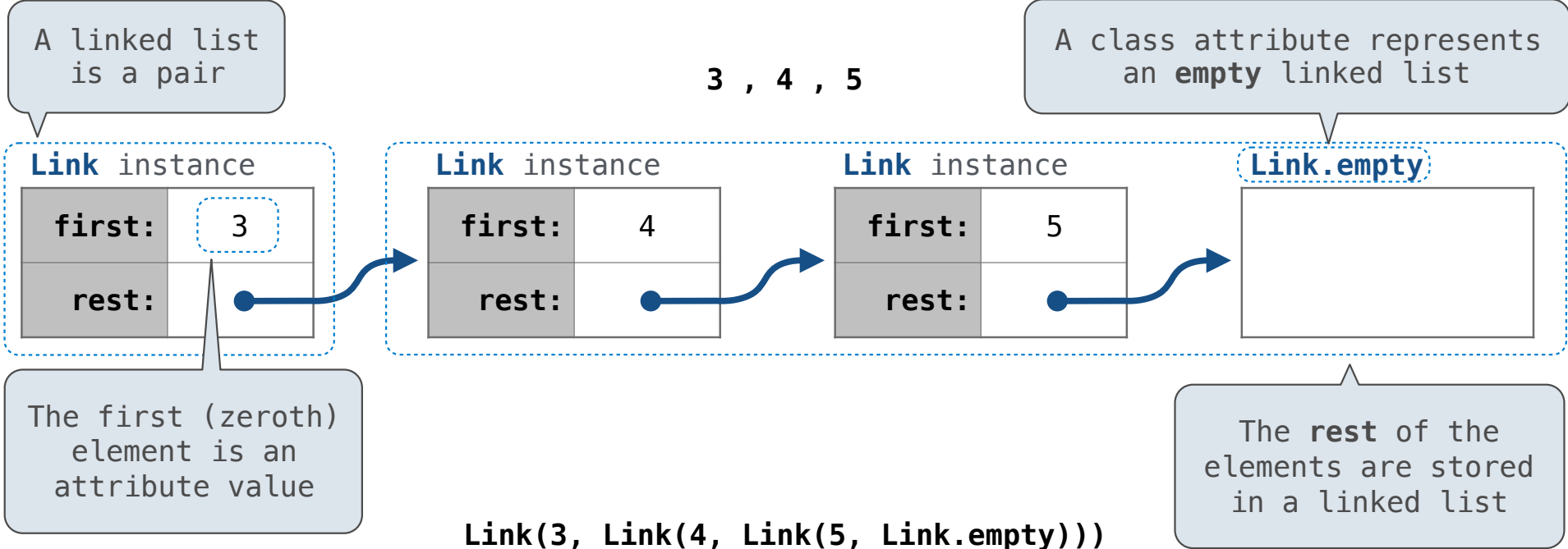
Linked List Structure

A linked list is either empty or a first value and the rest of the linked list



Linked List Structure

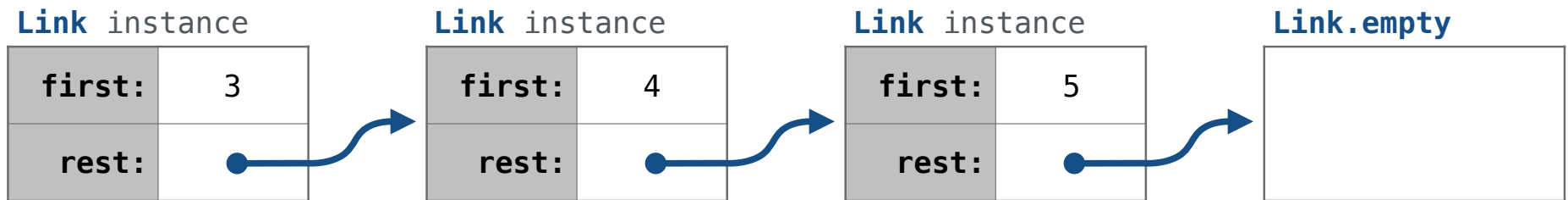
A linked list is either empty or a first value and the rest of the linked list



Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5

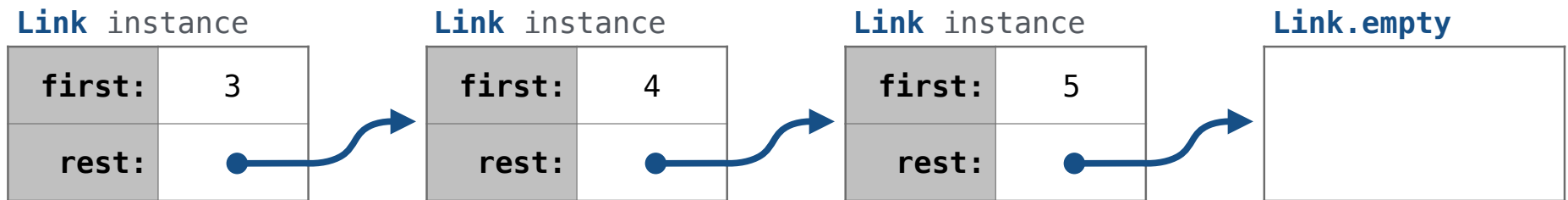


```
Link(3, Link(4, Link(5, Link.empty)))
```

Linked List Structure

A linked list is either empty **or** a first value and the rest of the linked list

3 , 4 , 5

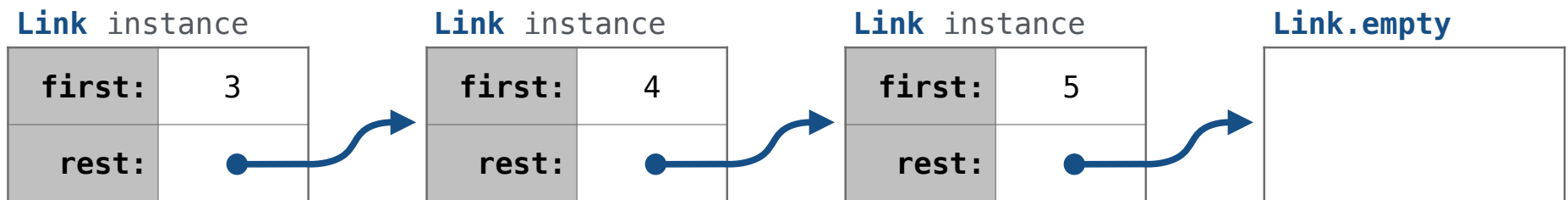


```
Link(3, Link(4, Link(5, Link.empty)))
```


Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

3 , 4 , 5

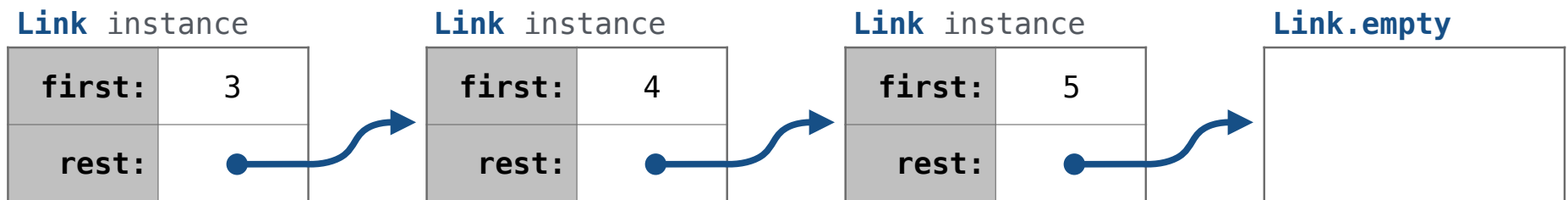


`Link(3, Link(4, Link(5, Link.empty)))`

Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

3 , 4 , 5

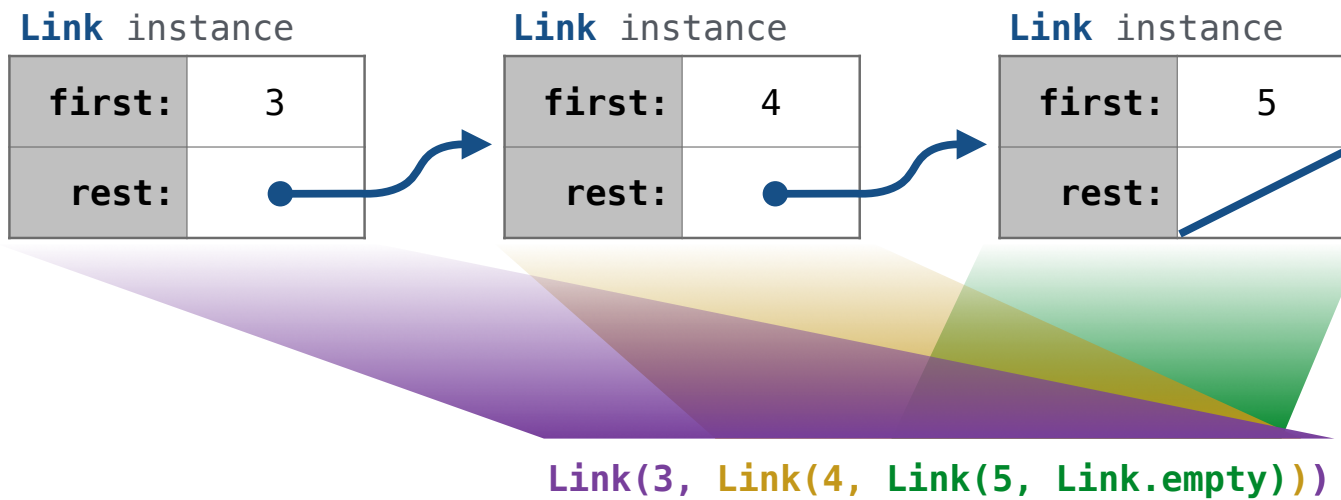


`Link(3, Link(4, Link(5, Link.empty)))`

Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

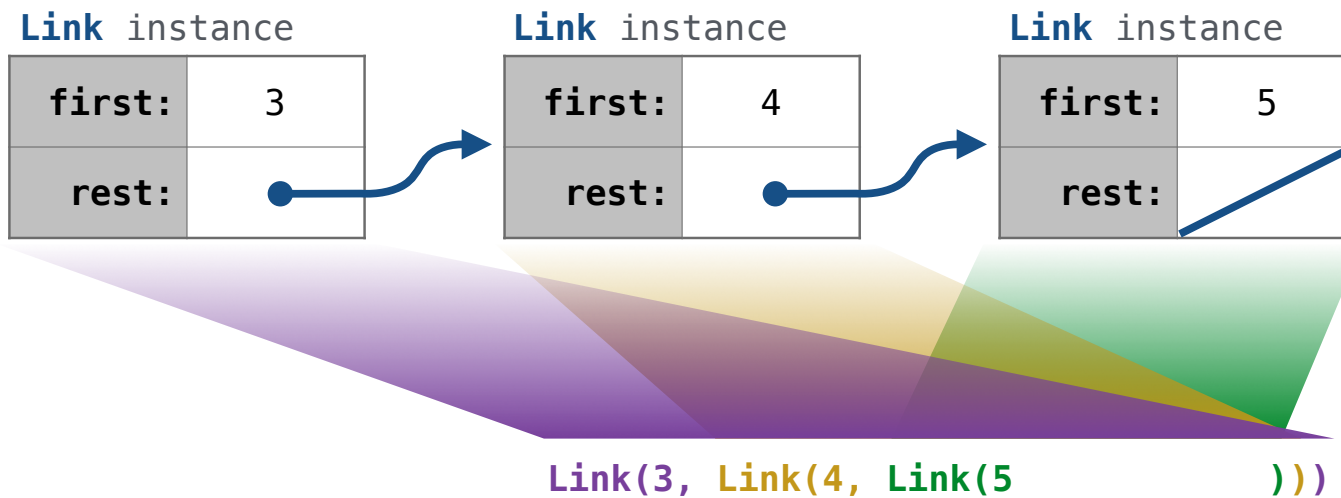
3 , 4 , 5



Linked List Structure

A linked list is either empty or a first value and the rest of the linked list

3 , 4 , 5



Linked List Class

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:
```

```
    Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:
```

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

```
        Link(3, Link(4, Link(5)))
```


Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:
```

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

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

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

    Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:
```

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

Returns whether
rest is a Link

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:
```

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

Returns whether
rest is a Link

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

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

Returns whether
rest is a Link

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

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

Some zero-length sequence

Returns whether rest is a Link

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

```
Link(3, Link(4, Link(5)))
```

Linked List Class

Linked list class: attributes are passed to `__init__`

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

Some zero-length sequence

Returns whether rest is a Link

`help(isinstance)`: Return whether an object is an instance of a class or of a subclass thereof.

```
Link(3, Link(4, Link(5)))
```

(Demo)

Property Methods

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
```

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
```

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
```

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
```

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

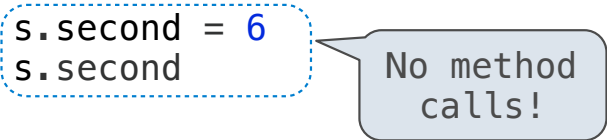
```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```



No method calls!

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```



No method calls!

The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```



No method calls!

The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance

A `@<attribute>.setter` decorator on a method designates that it will be called whenever that attribute is assigned. `<attribute>` must be an existing property method.

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```



No method calls!

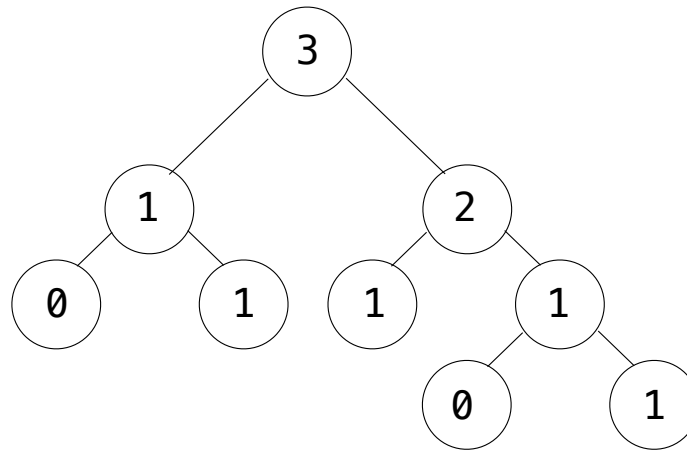
The `@property` decorator on a method designates that it will be called whenever it is looked up on an instance

A `@<attribute>.setter` decorator on a method designates that it will be called whenever that attribute is assigned. `<attribute>` must be an existing property method.

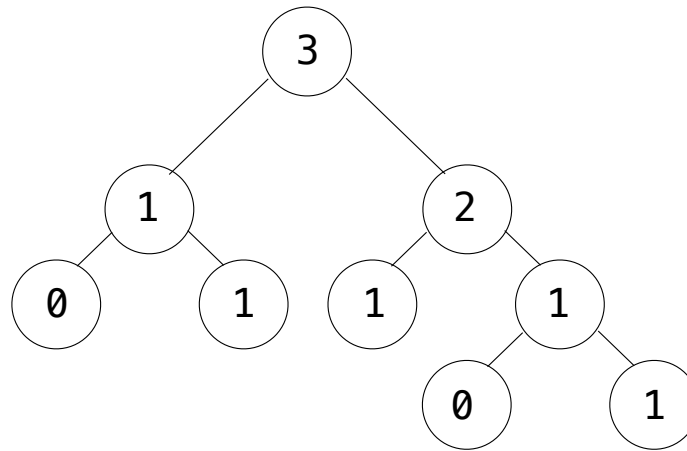
(Demo)

Tree Class

Tree Abstraction (Review)



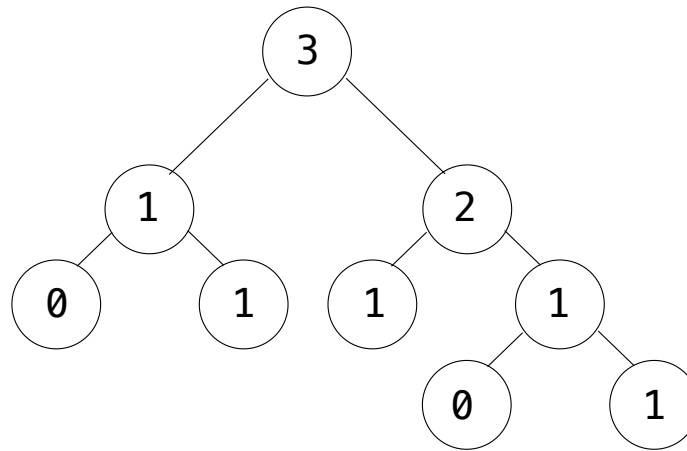
Tree Abstraction (Review)



Recursive description (wooden trees):

Relative description (family trees):

Tree Abstraction (Review)

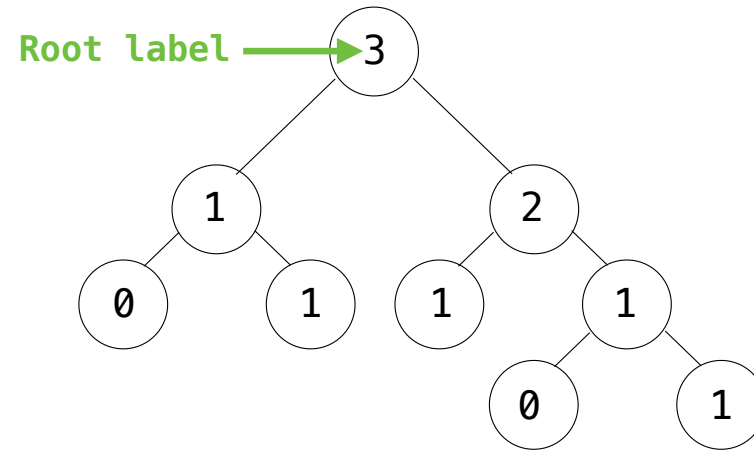


Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Relative description (family trees):

Tree Abstraction (Review)

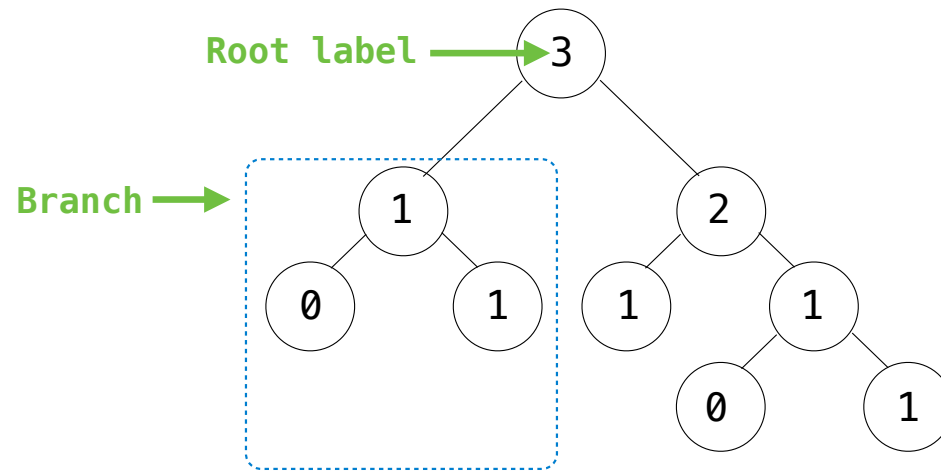


Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Relative description (family trees):

Tree Abstraction (Review)

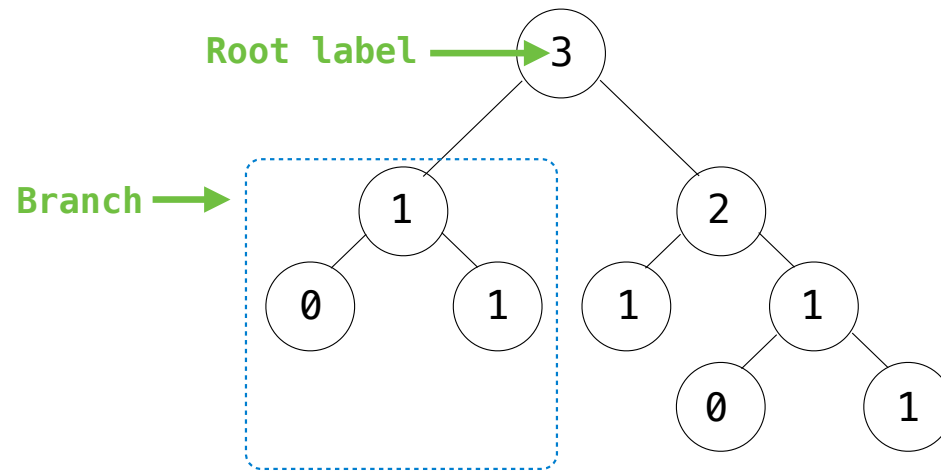


Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Relative description (family trees):

Tree Abstraction (Review)



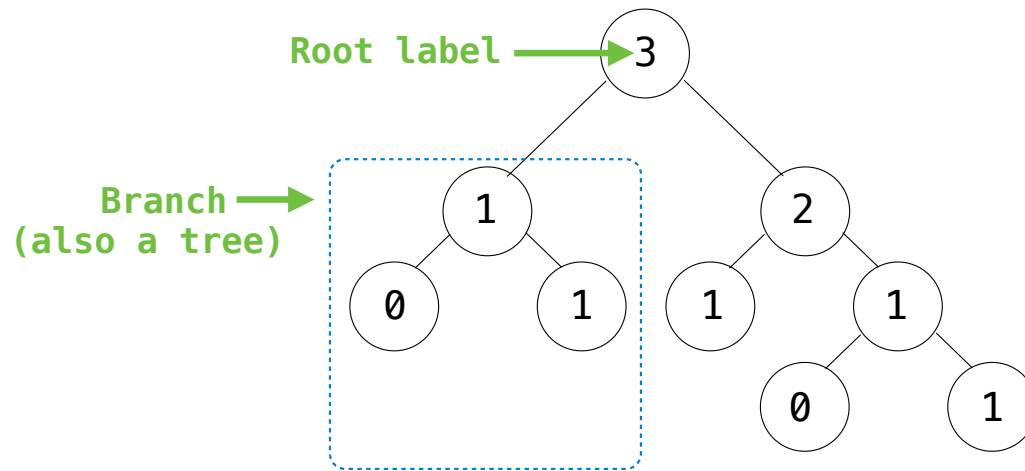
Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

Relative description (family trees):

Tree Abstraction (Review)



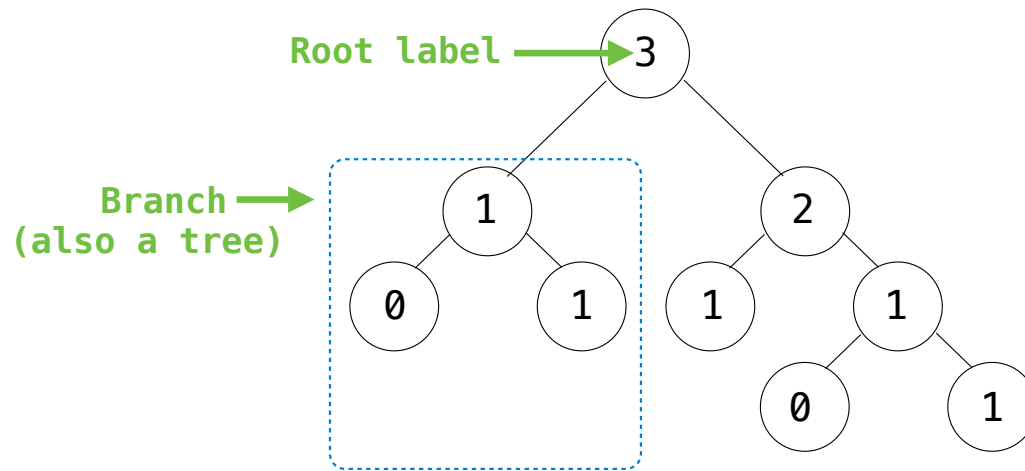
Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

Relative description (family trees):

Tree Abstraction (Review)



Recursive description (wooden trees):

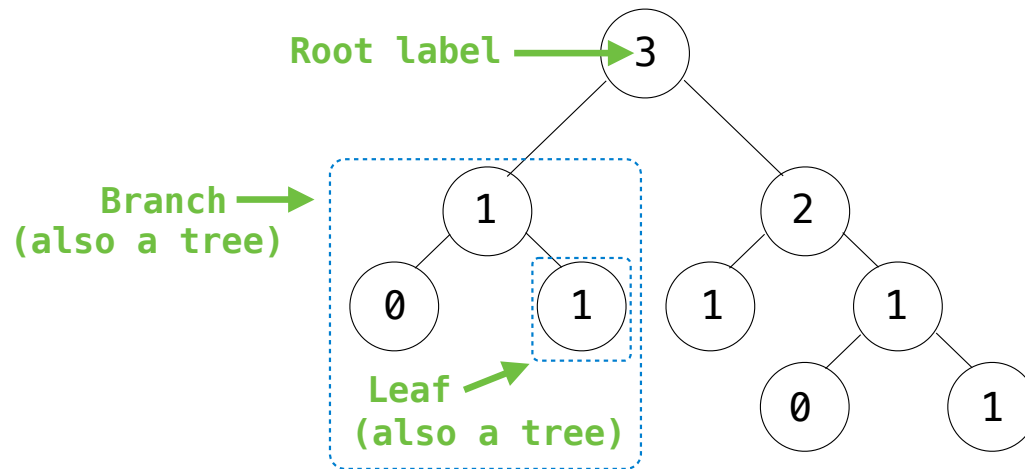
A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

Relative description (family trees):

Tree Abstraction (Review)



Recursive description (wooden trees):

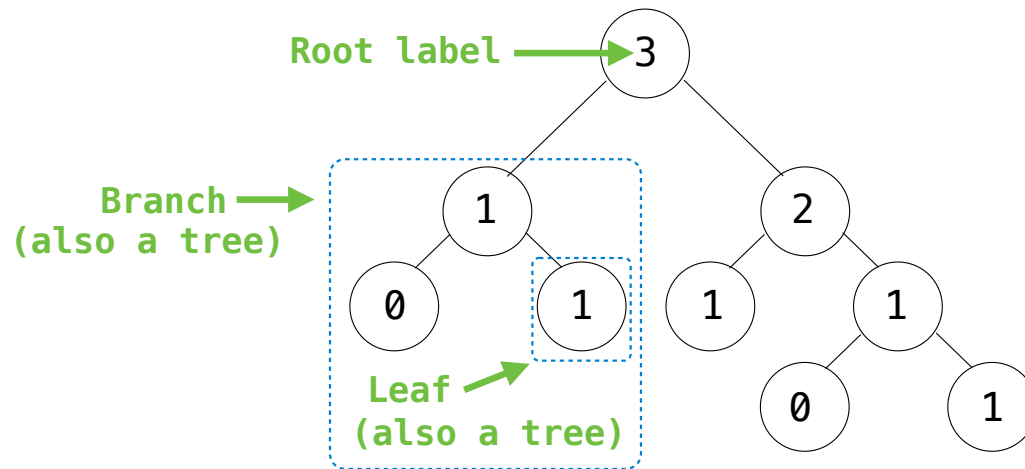
A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

Relative description (family trees):

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

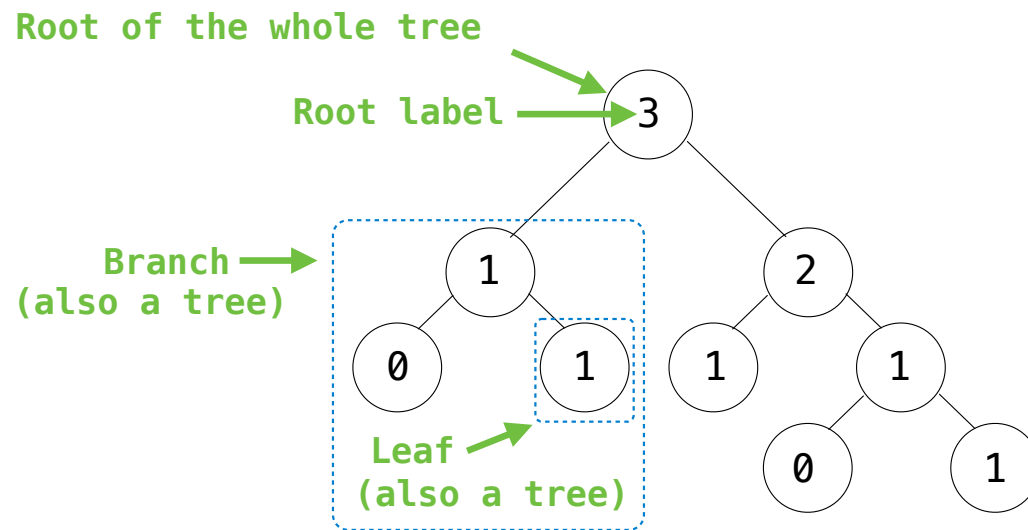
Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

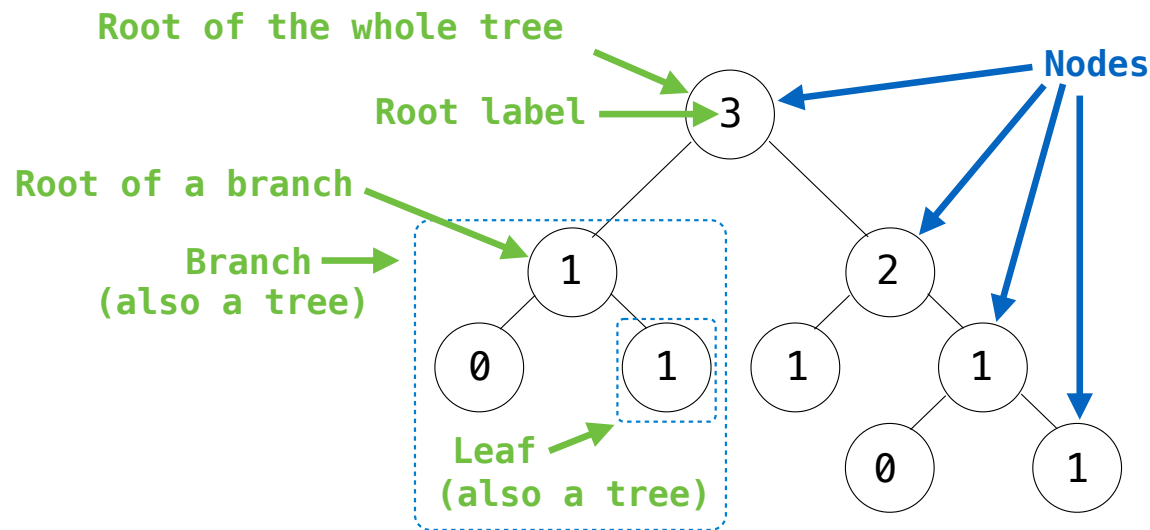
Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

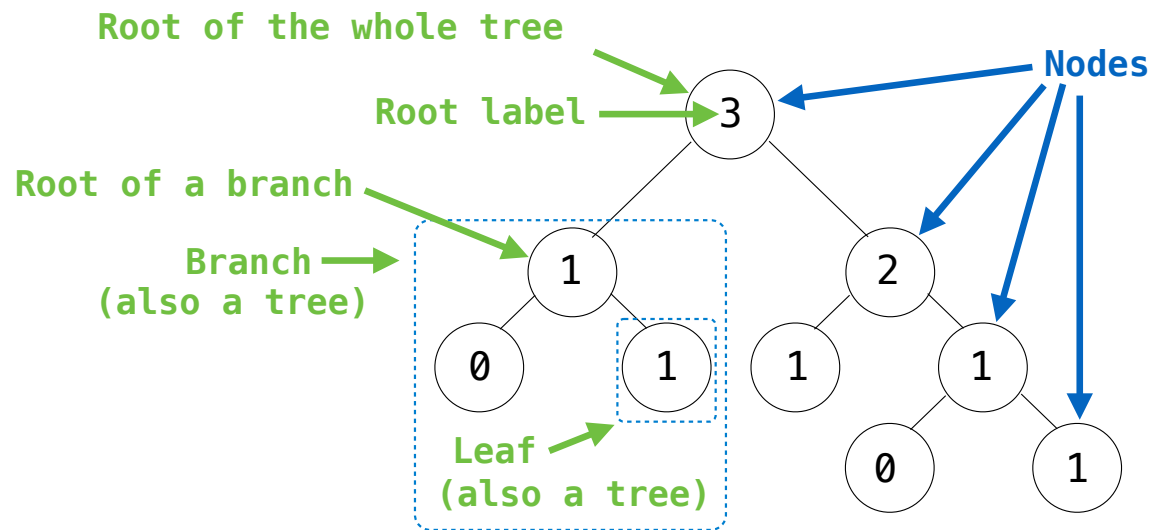
A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

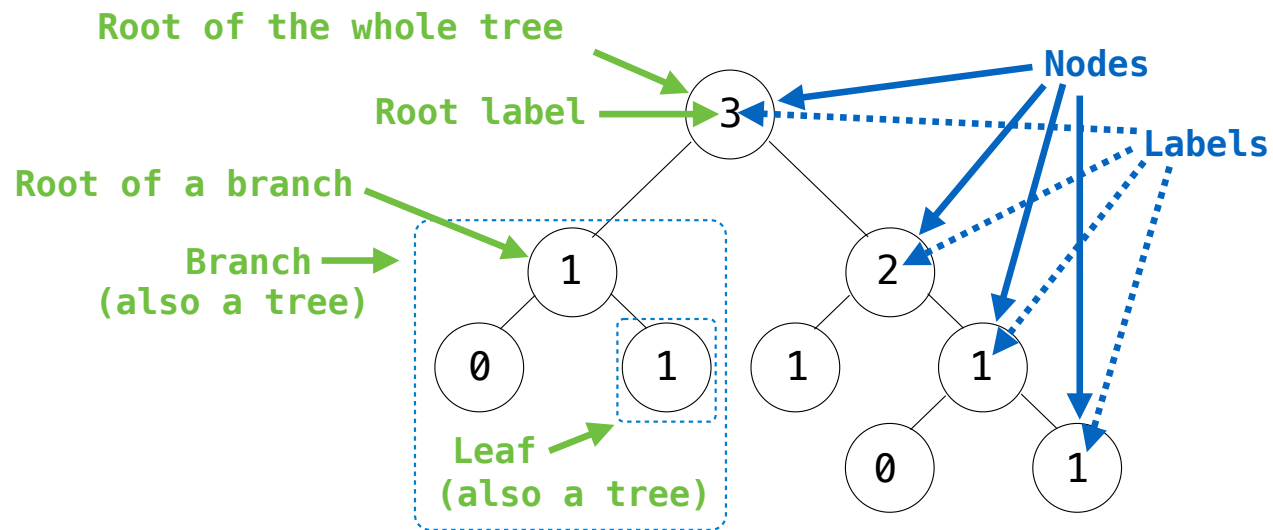
A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

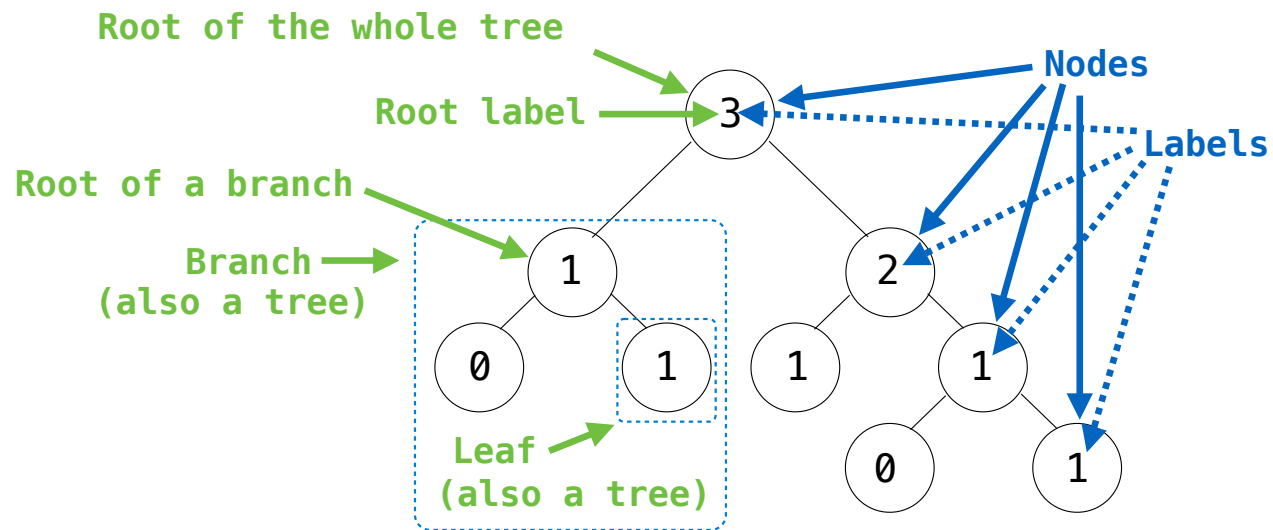
A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

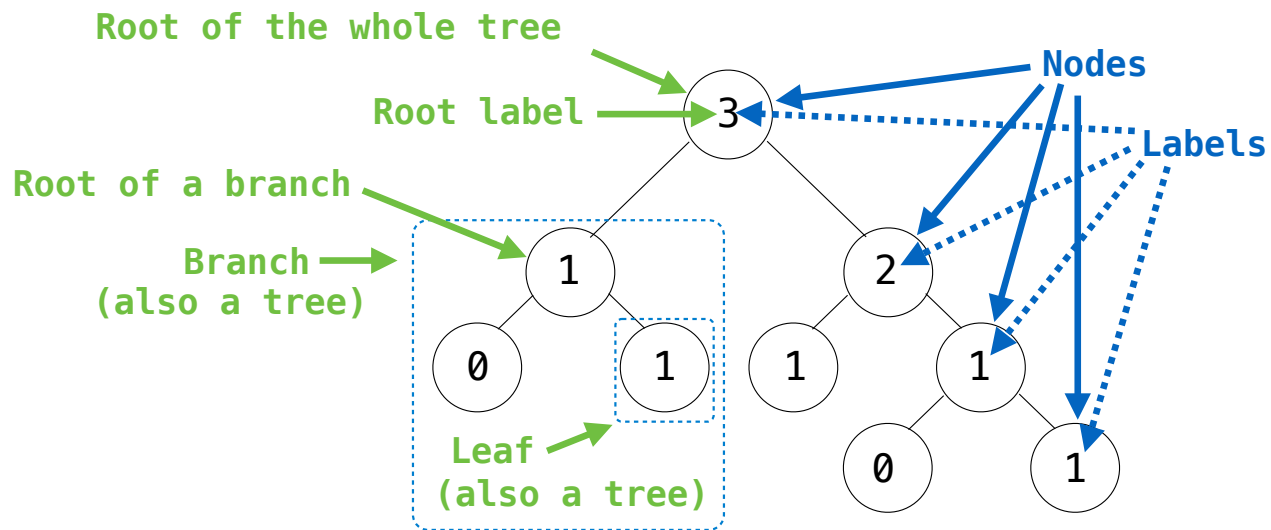
Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

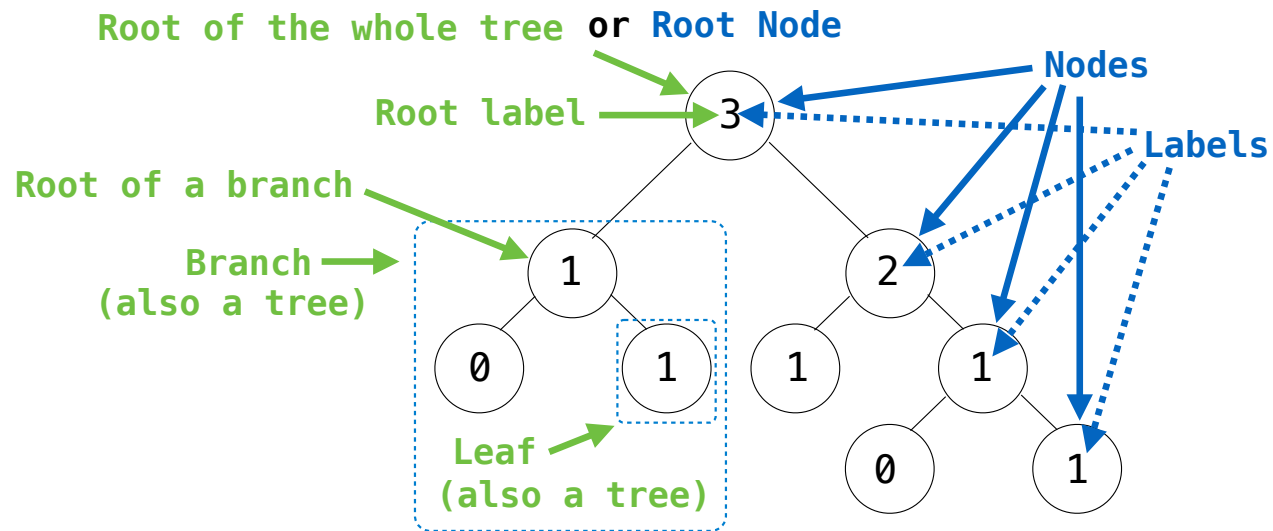
Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

The top node is the **root node**

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

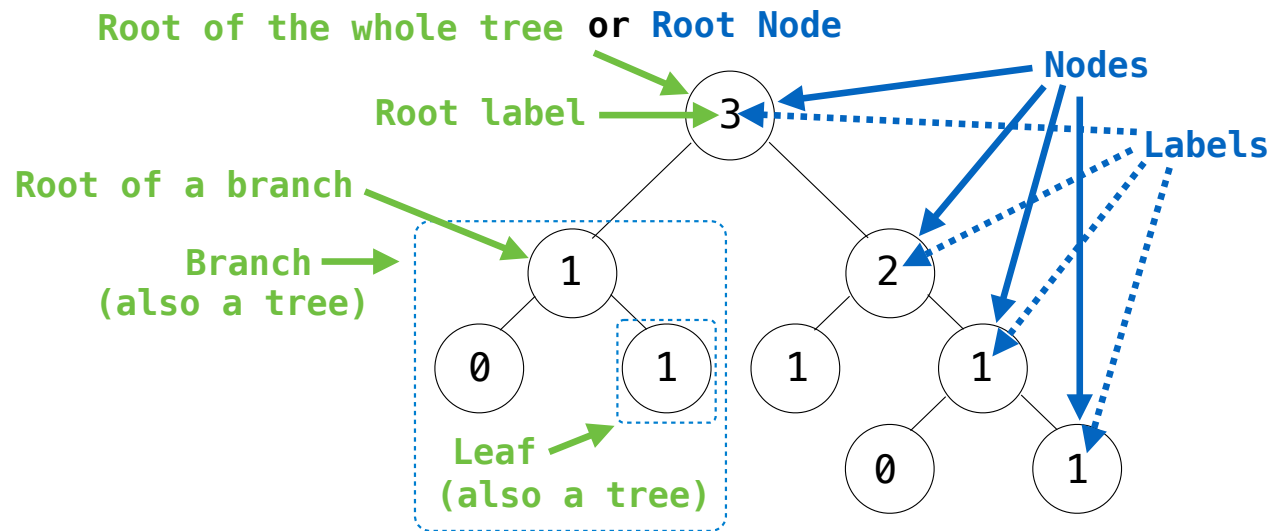
Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

The top node is the **root node**

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

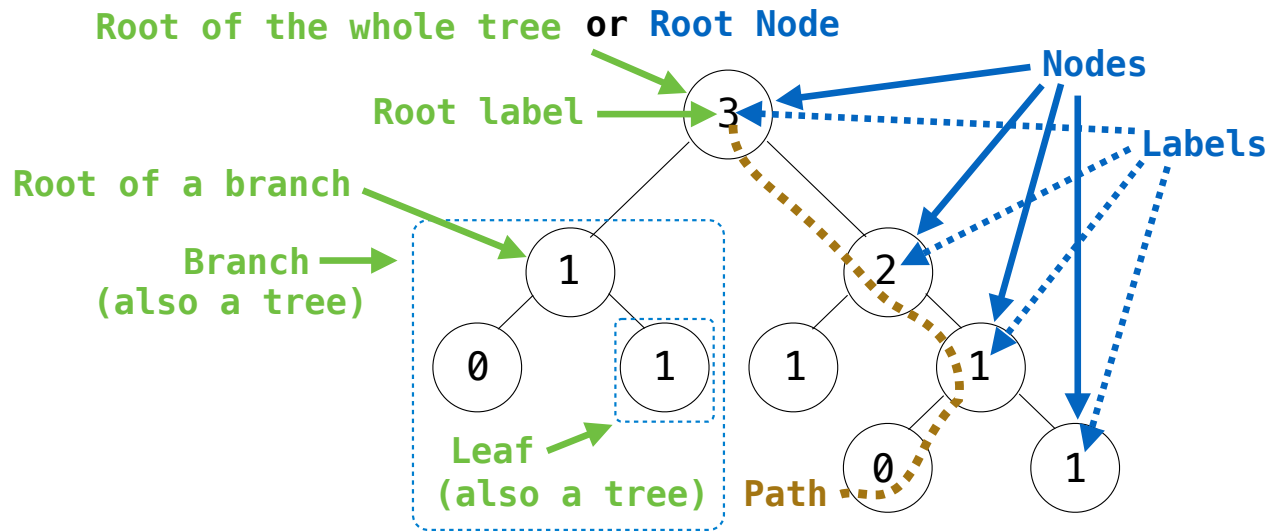
Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

The top node is the **root node**

People often refer to labels by their locations: "each parent is the sum of its children"

Tree Abstraction (Review)



Recursive description (wooden trees):

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description (family trees):

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent/child** of another

The top node is the **root node**

People often refer to labels by their locations: "each parent is the sum of its children"

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

```
class Tree:
```

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

    def tree(label, branches=[]):
        for branch in branches:
            assert is_tree(branch)
        return [label] + list(branches)

    def label(tree):
        return tree[0]

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

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

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])

def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

```
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])
```

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

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])
```


Tree Class

A Tree has a label and a list of branches; each branch is a Tree

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

```
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])
```

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

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])
```

(Demo)

Tree Mutation

Example: Pruning Trees

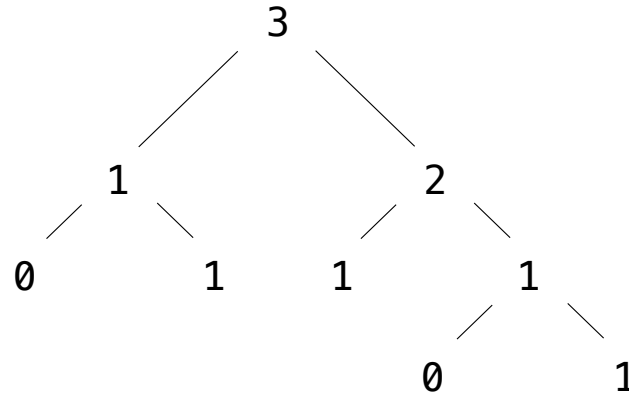
Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

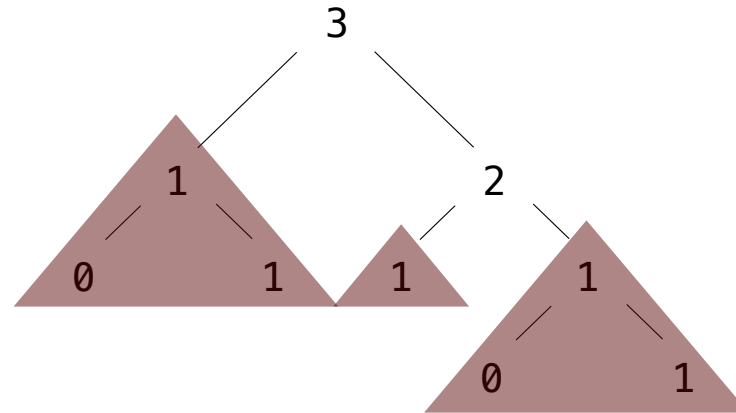
Prune branches before recursive processing



Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

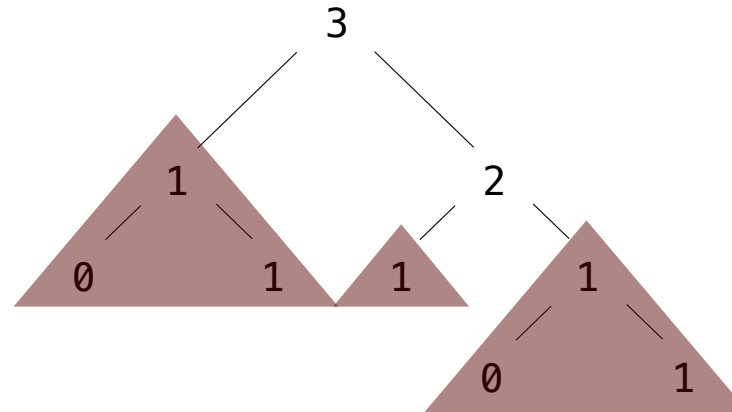
Prune branches before recursive processing



Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

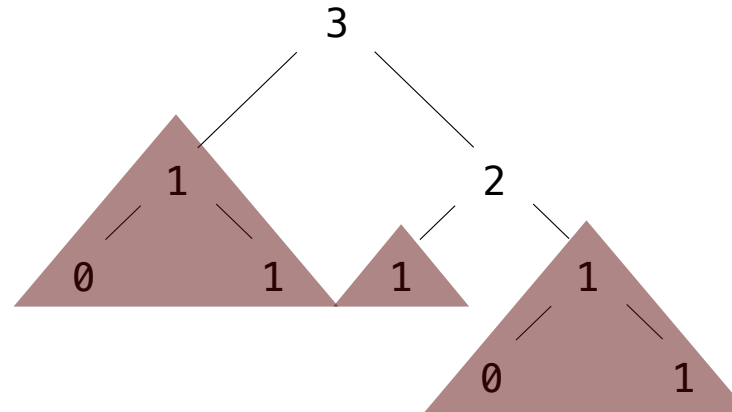


```
def prune(t, n):  
    """Prune sub-trees whose label value is n."""  
    t.branches = [_____ for b in t.branches if _____]  
    for b in t.branches:  
        prune(_____, _____)
```

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

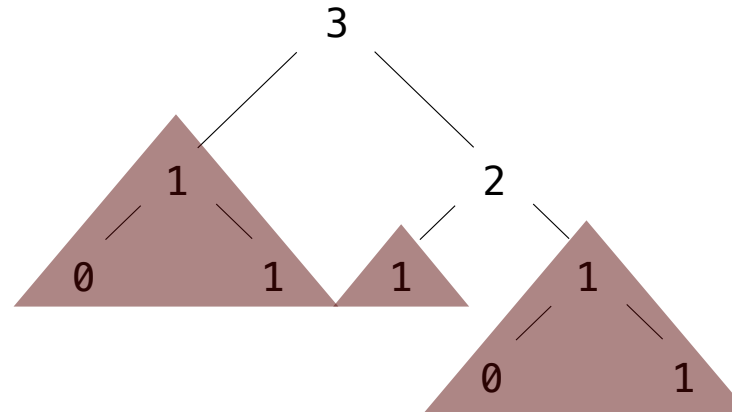


```
def prune(t, n):  
    """Prune sub-trees whose label value is n."""  
    t.branches = [_____ b _____ for b in t.branches if _____ b.label != n _____]  
    for b in t.branches:  
        prune(_____, _____)
```

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

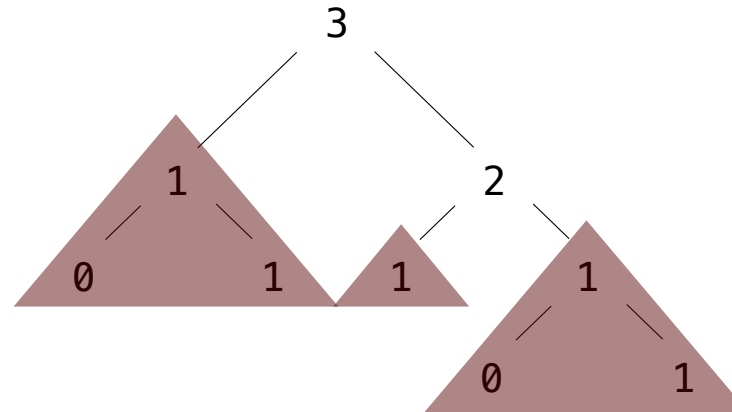


```
def prune(t, n):  
    """Prune sub-trees whose label value is n."""  
    t.branches = [_____ b _____ for b in t.branches if _____ b.label != n _____]  
    for b in t.branches:  
        prune(_____ b _____, _____ n _____)
```


Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing



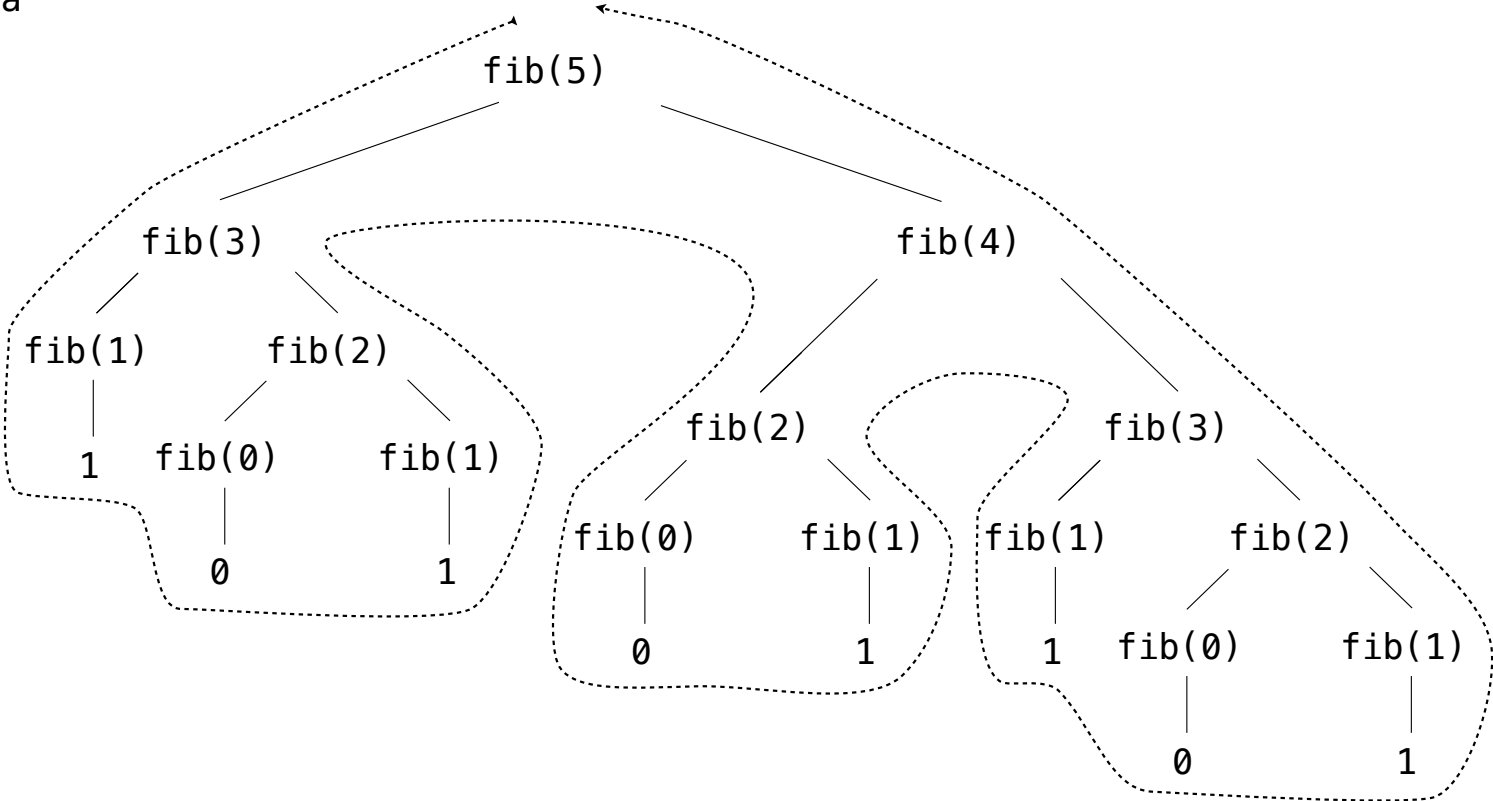
```
def prune(t, n):  
    """Prune sub-trees whose label value is n."""  
    t.branches = [_____ b _____ for b in t.branches if _____ b.label != n _____]  
    for b in t.branches:  
        prune(_____ b _____, _____ n _____)
```

(Demo)

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

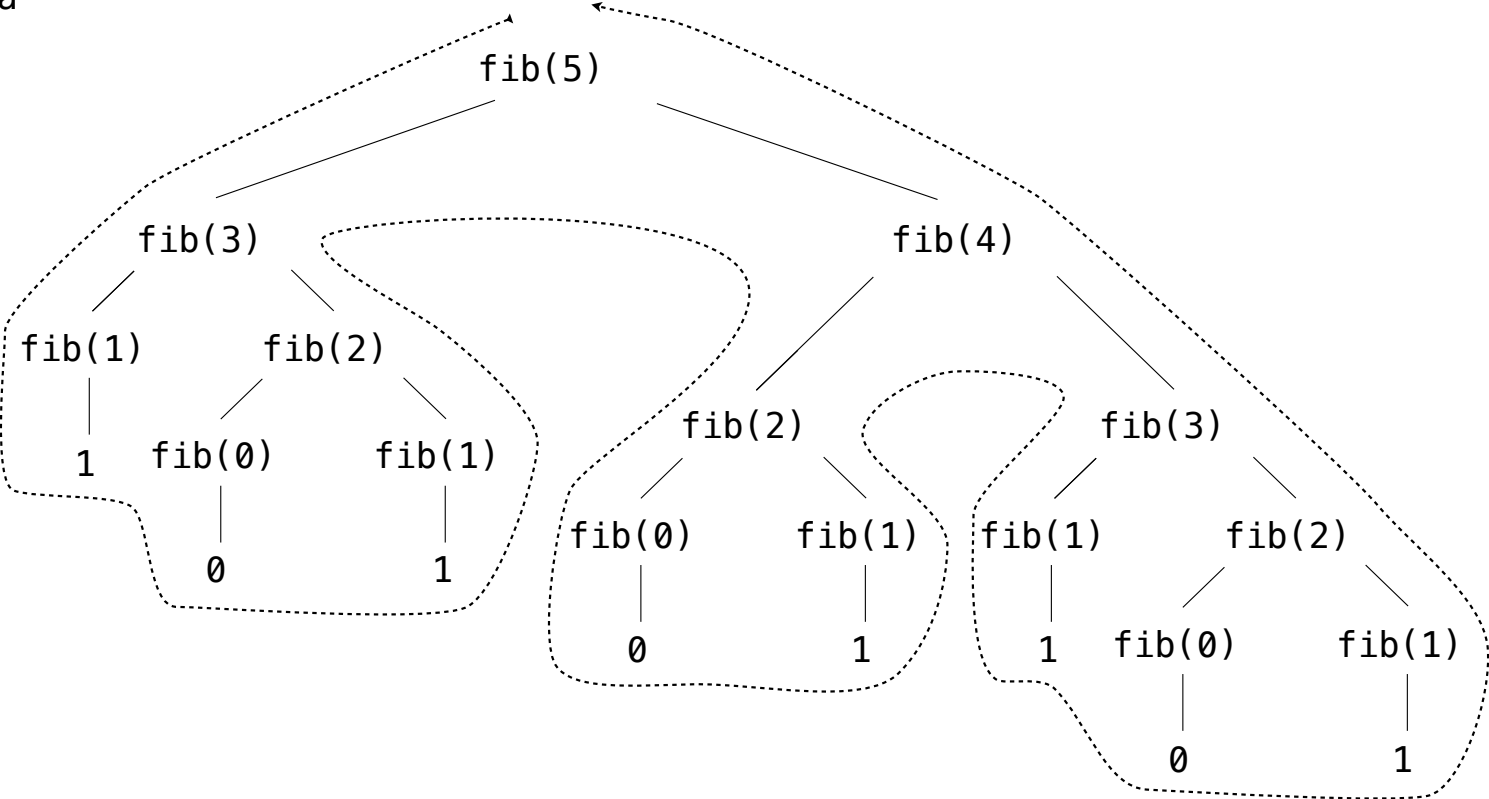


Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:



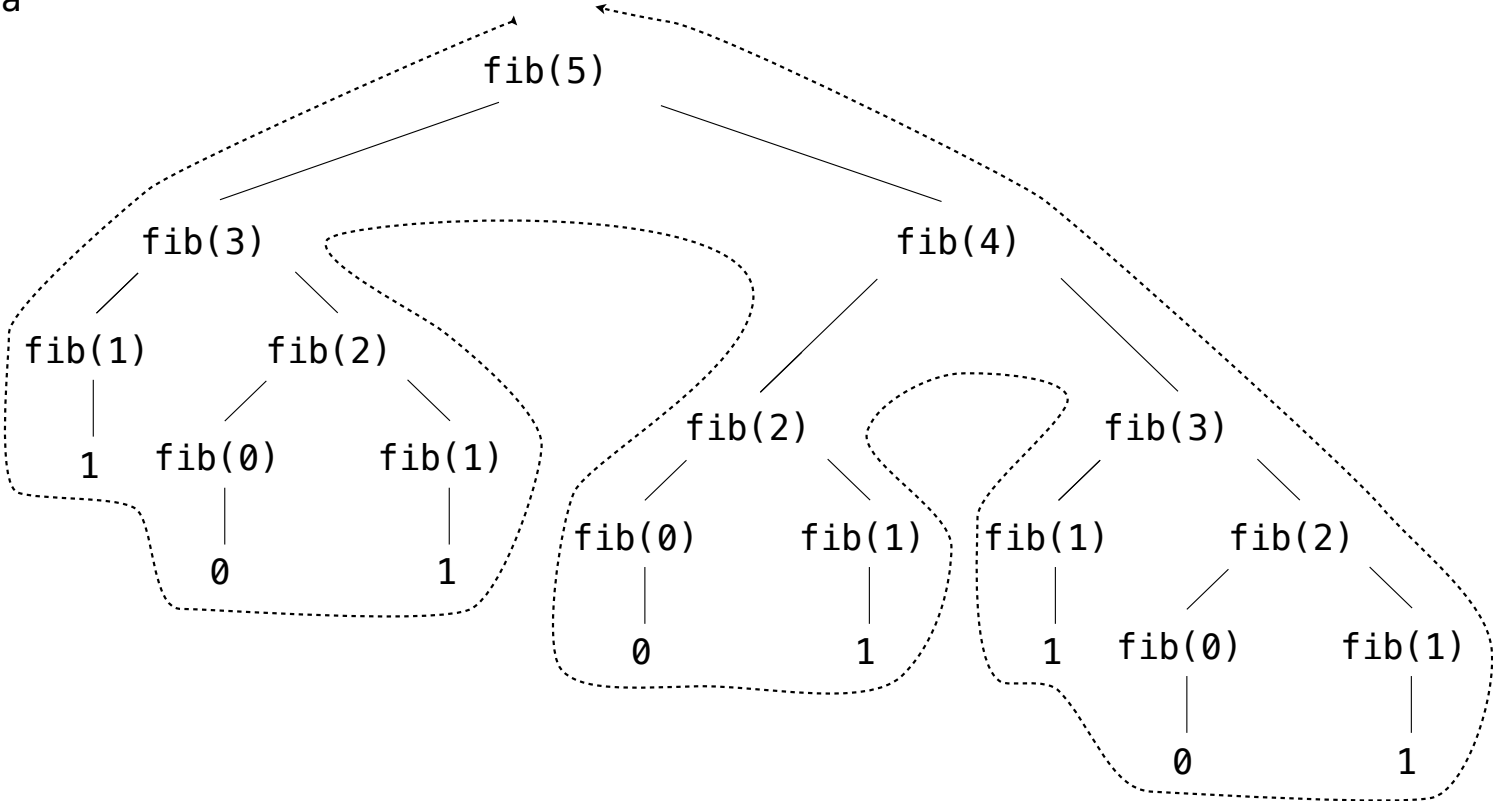
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib



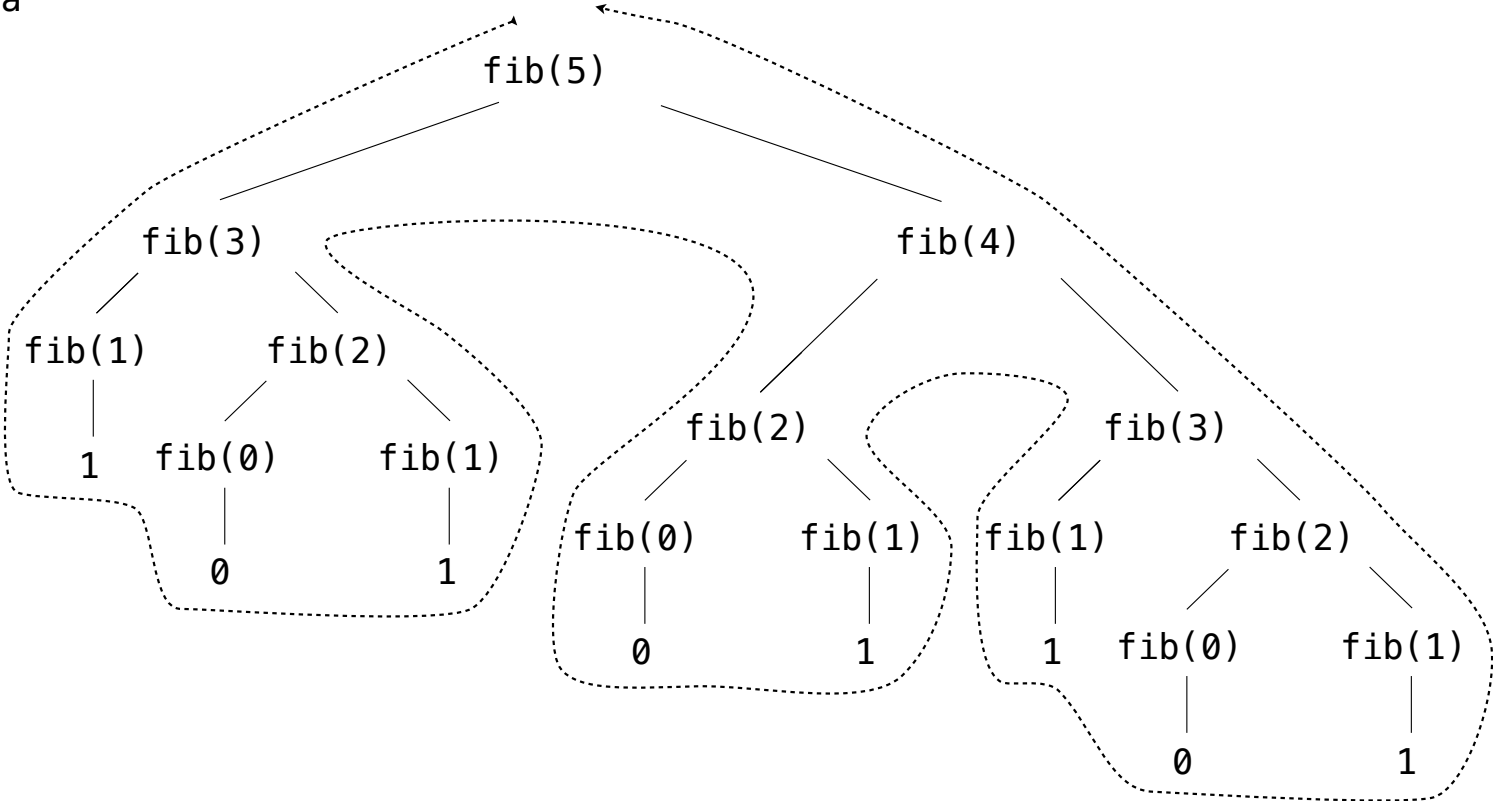
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache



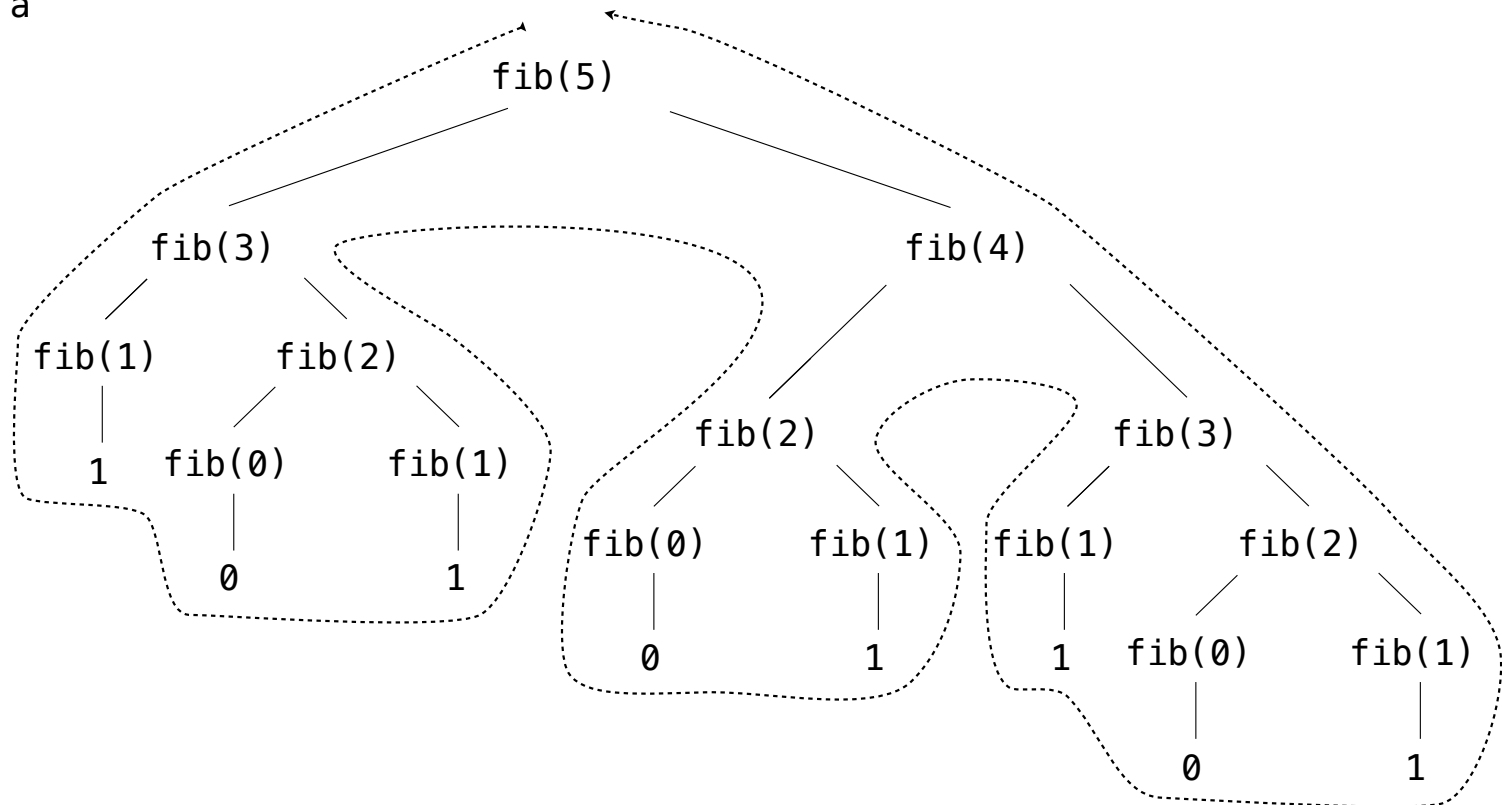
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



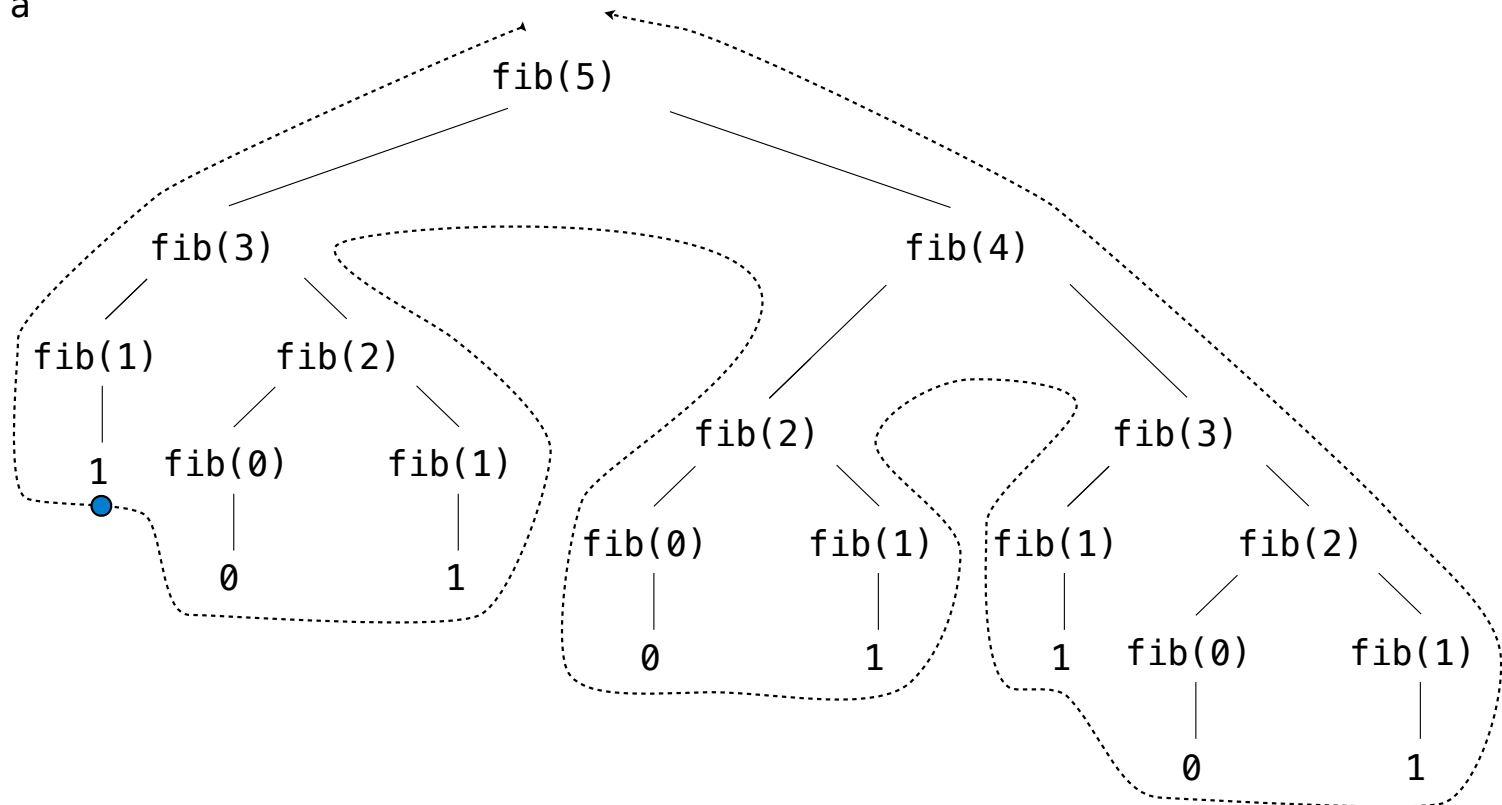
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



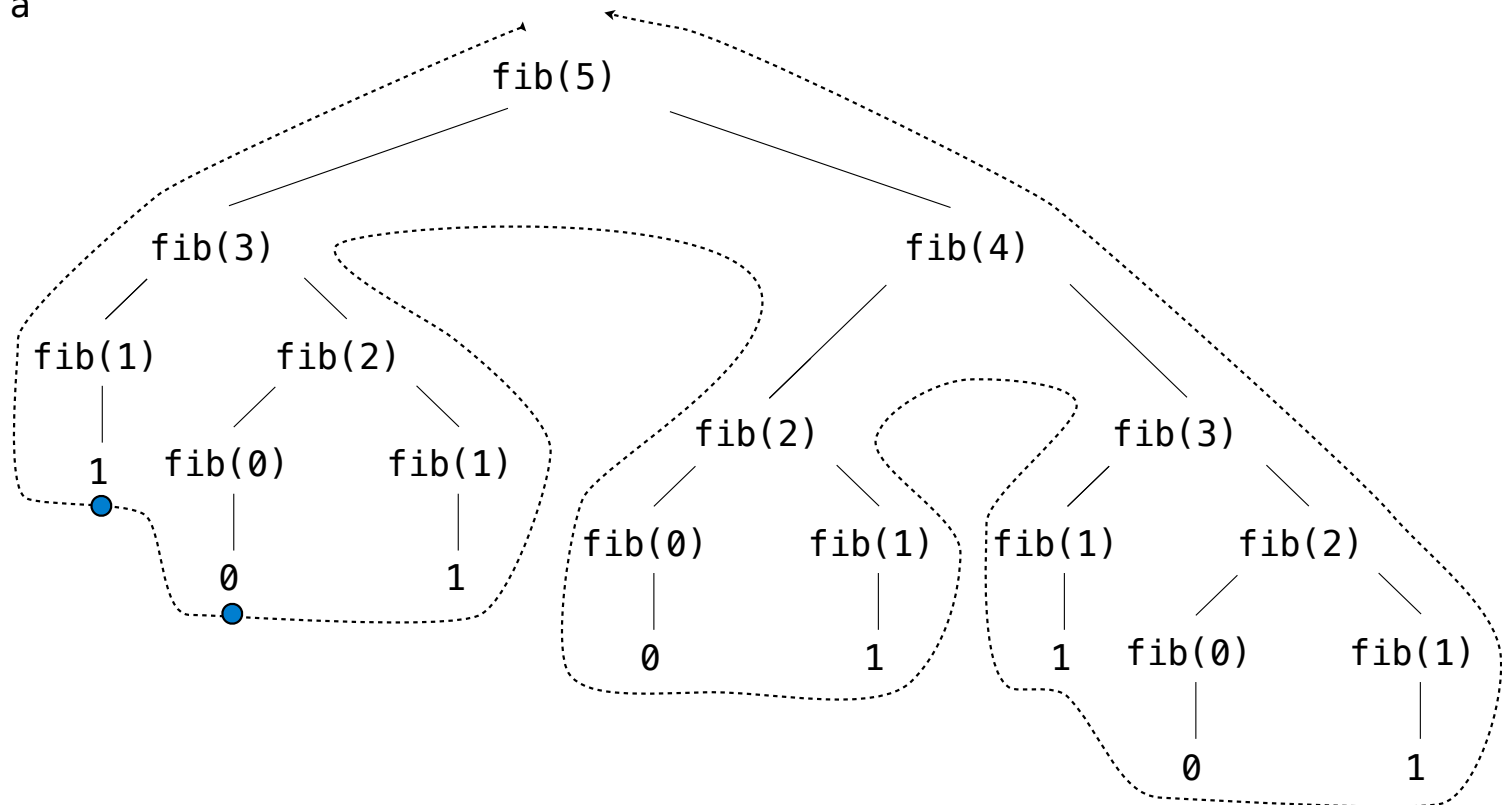
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



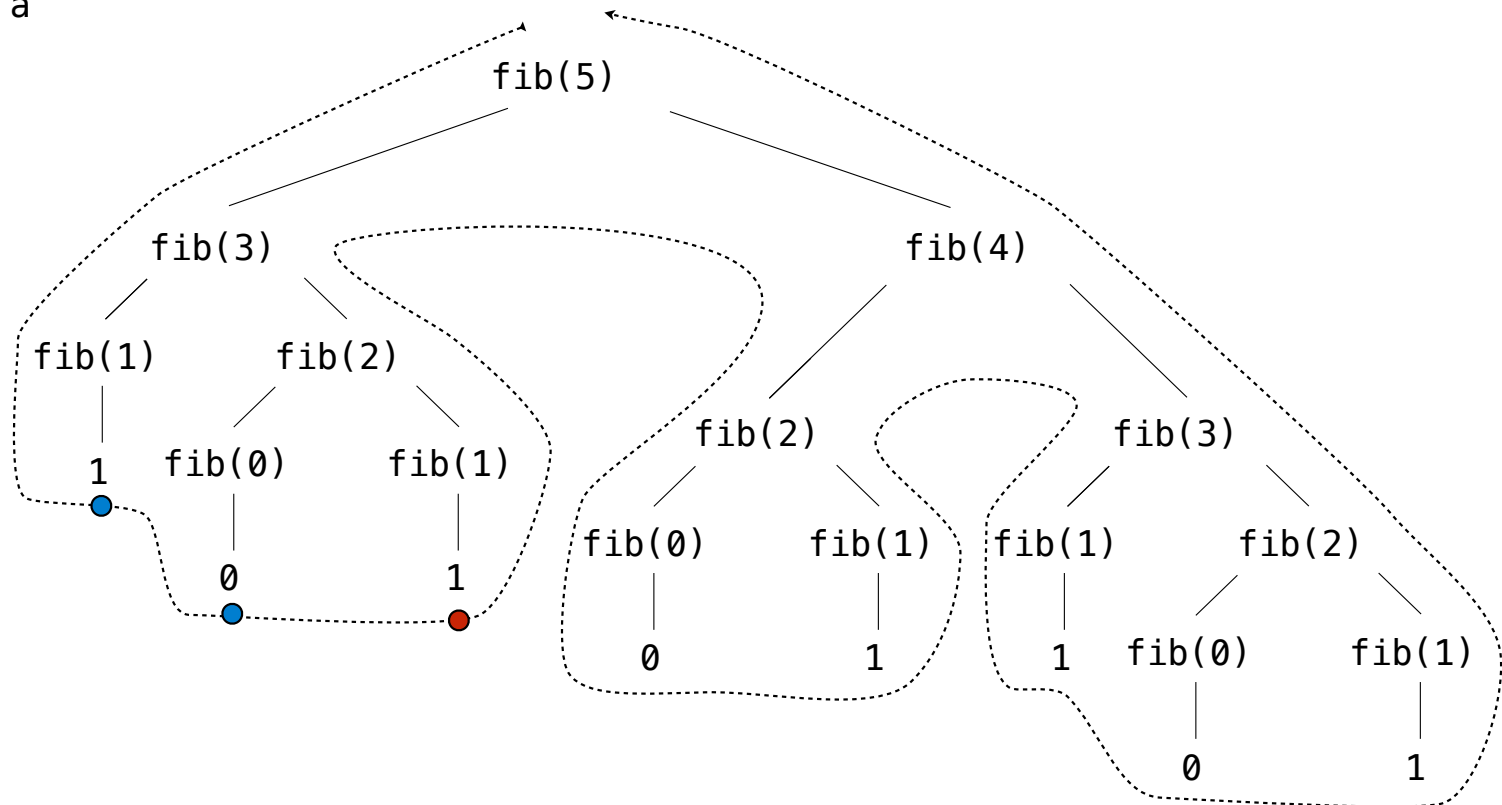
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



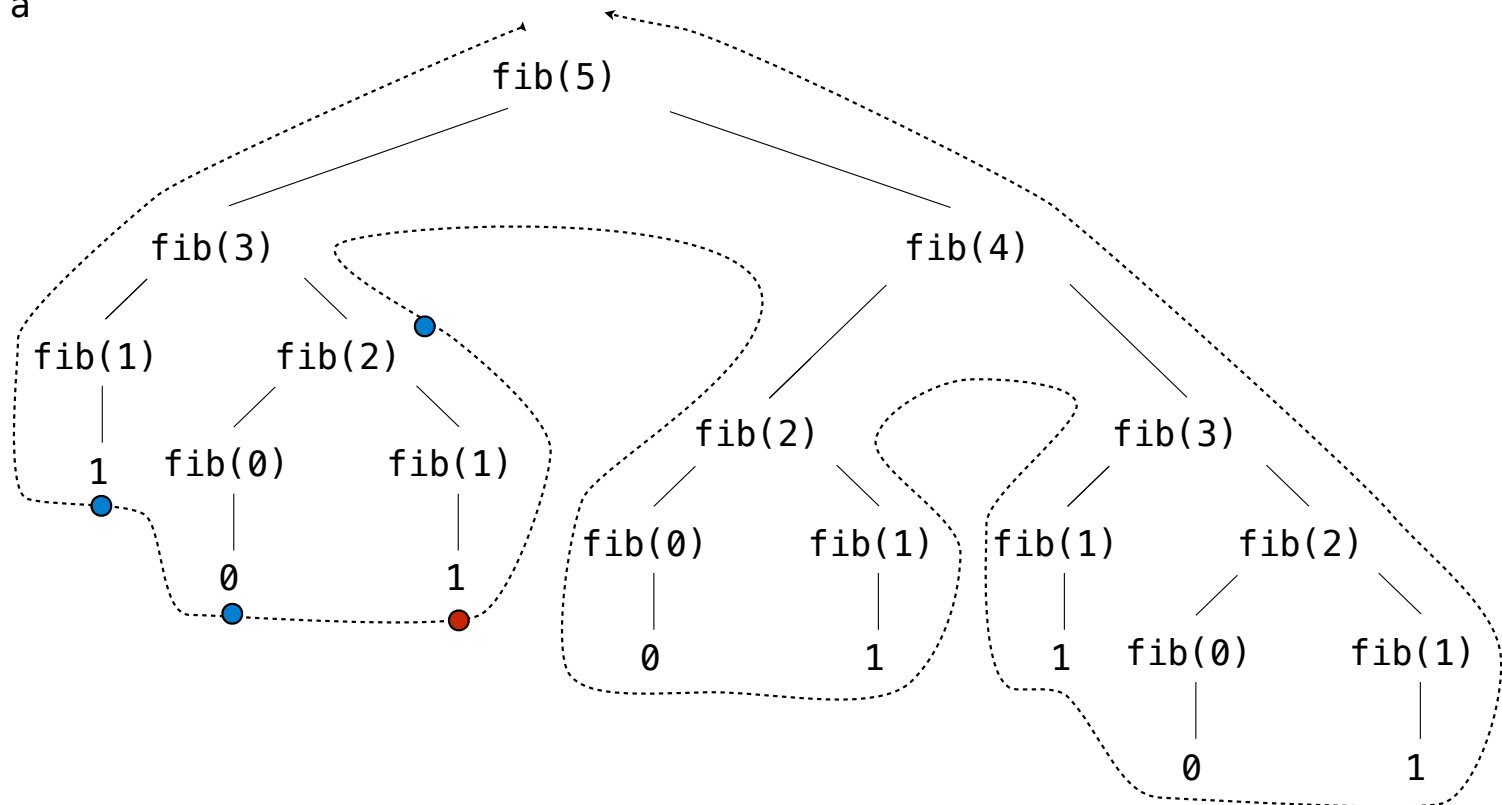
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



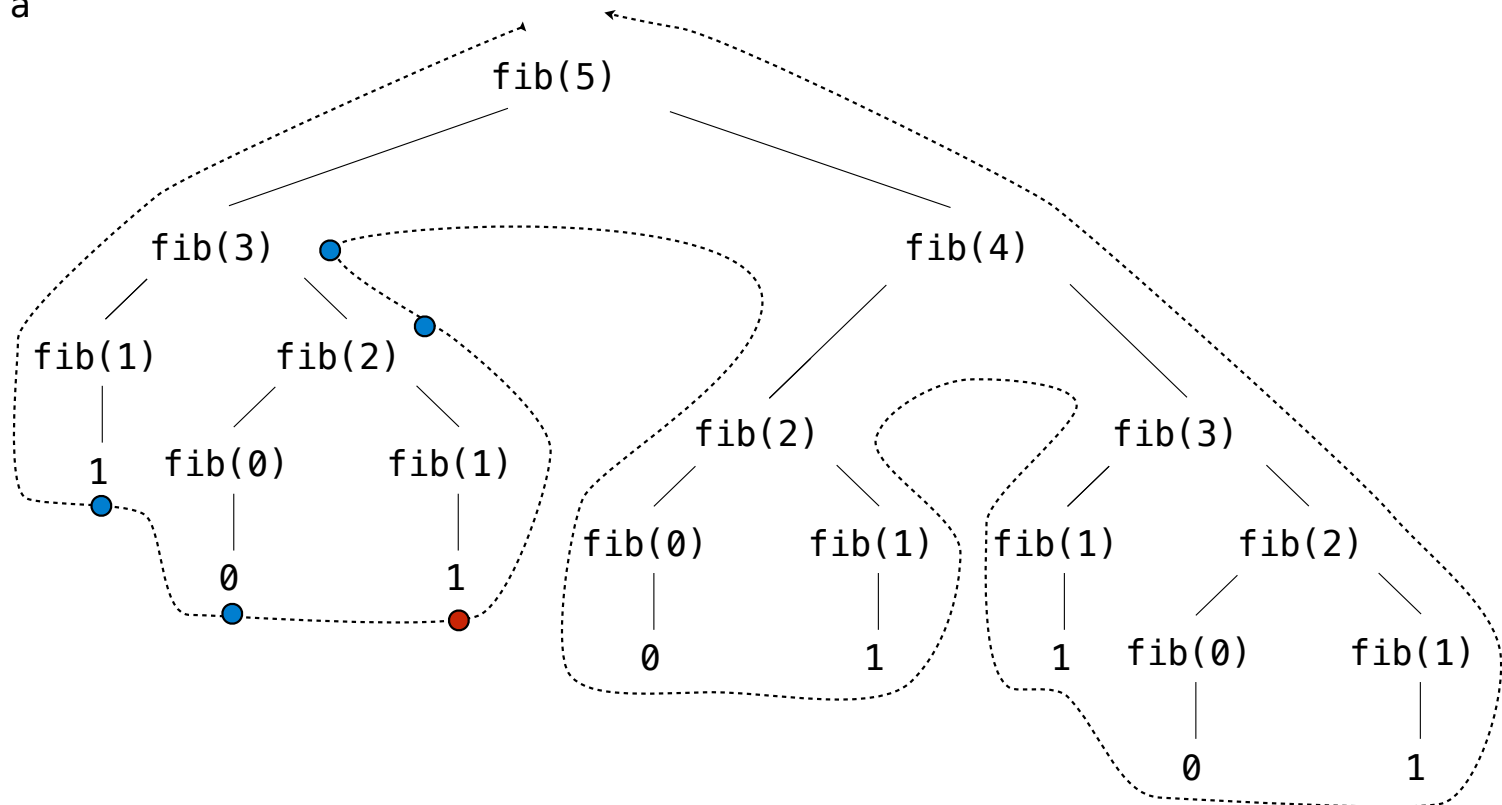
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



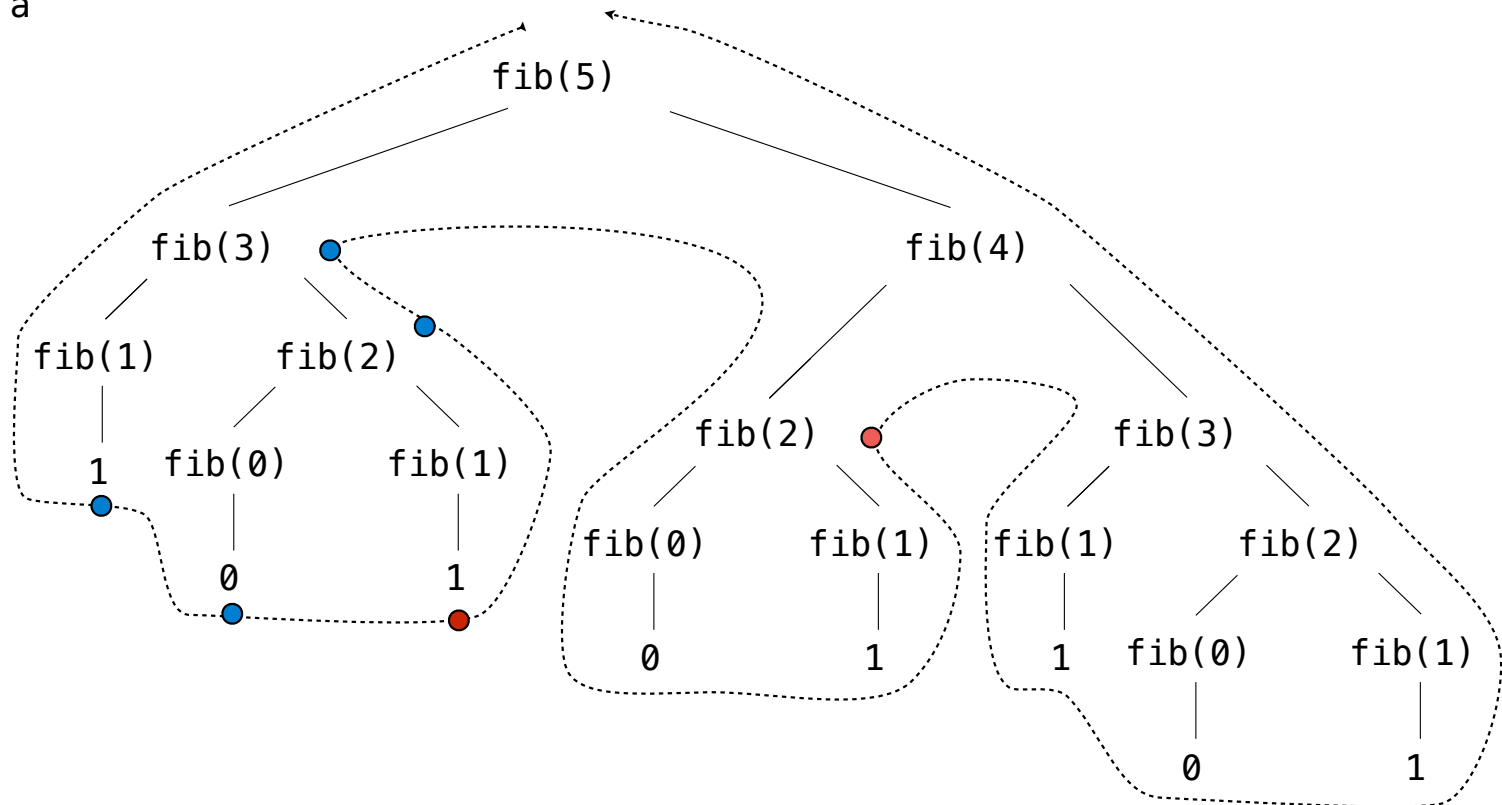
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



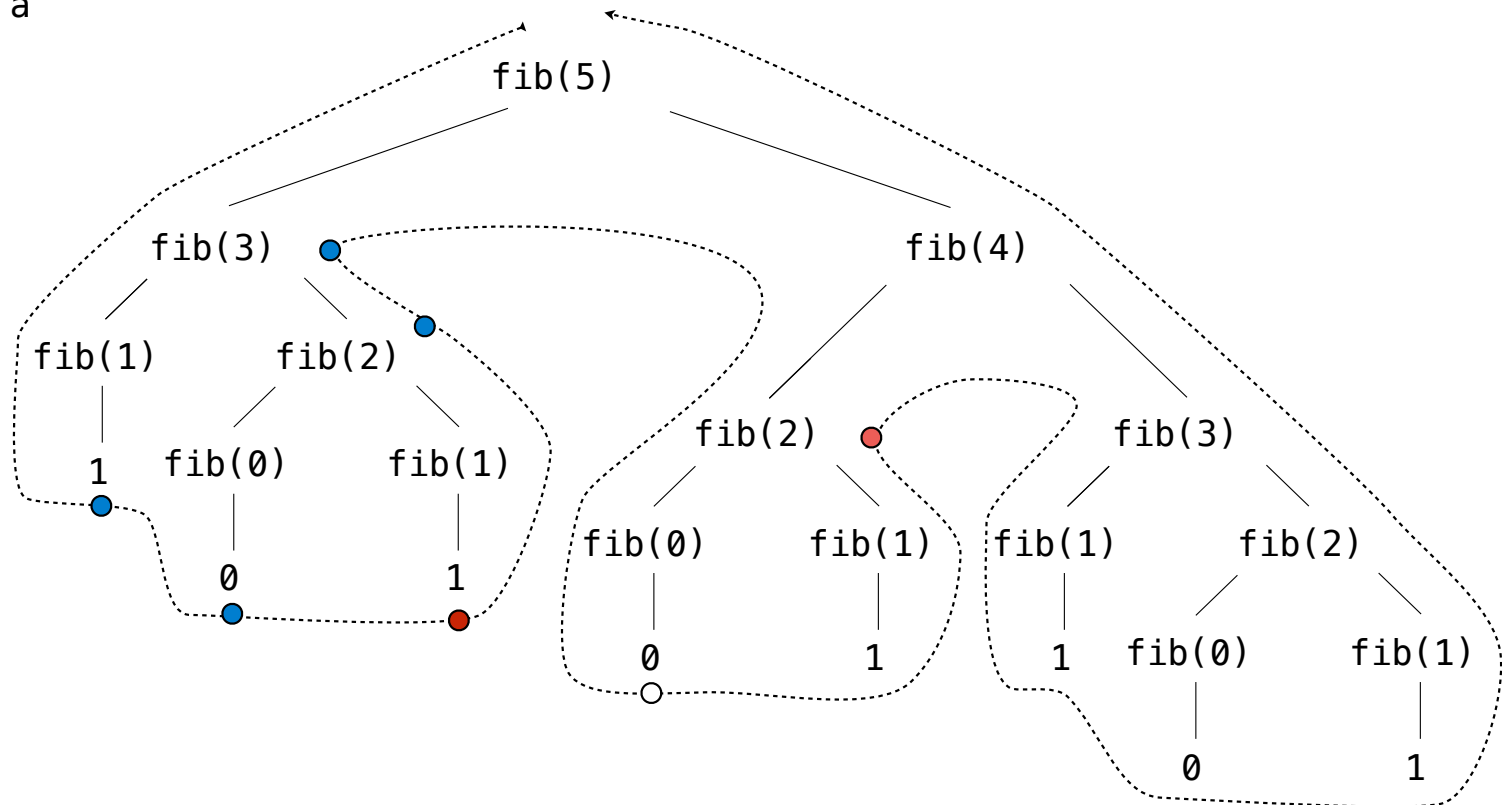
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



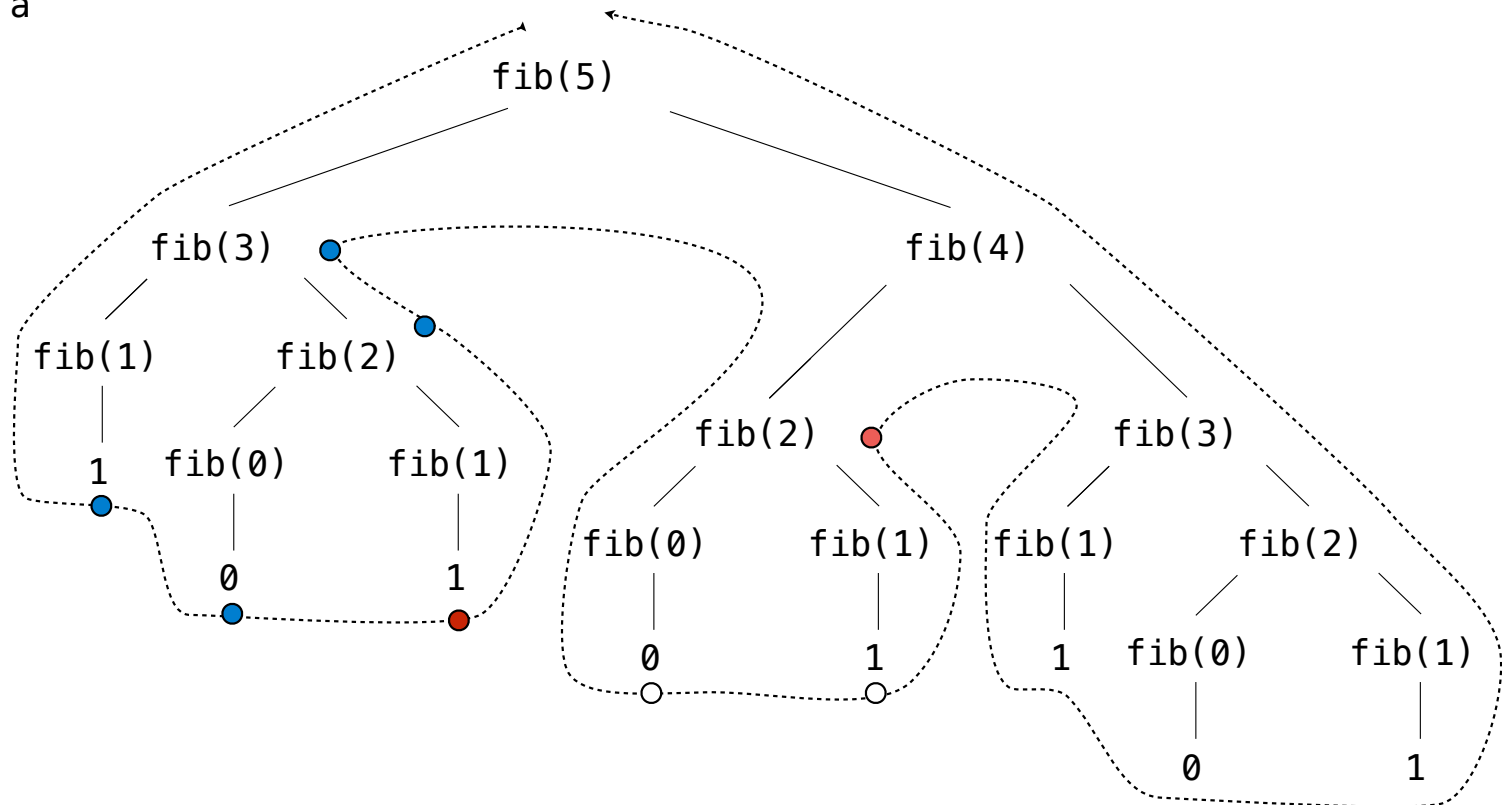
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



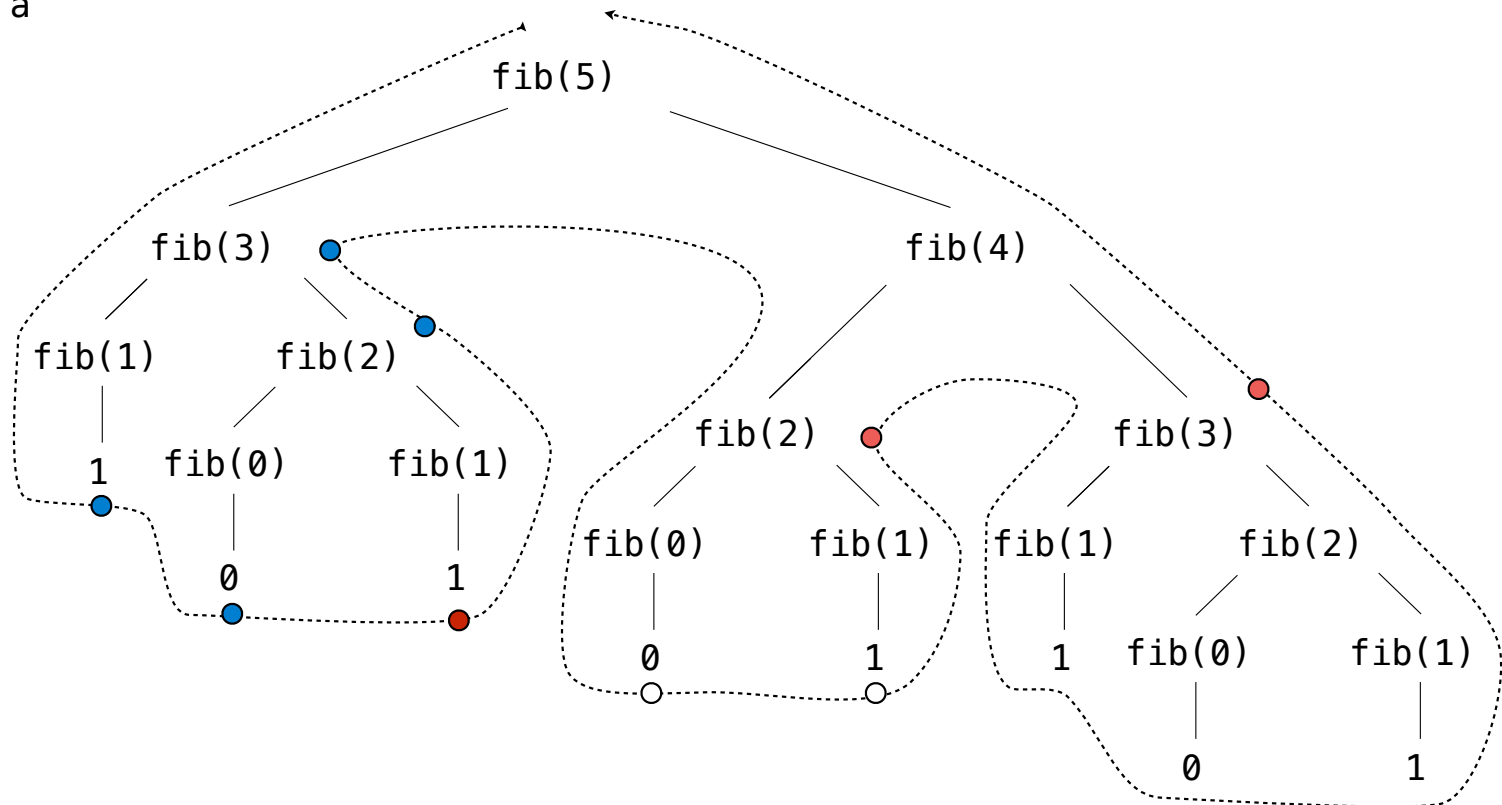
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



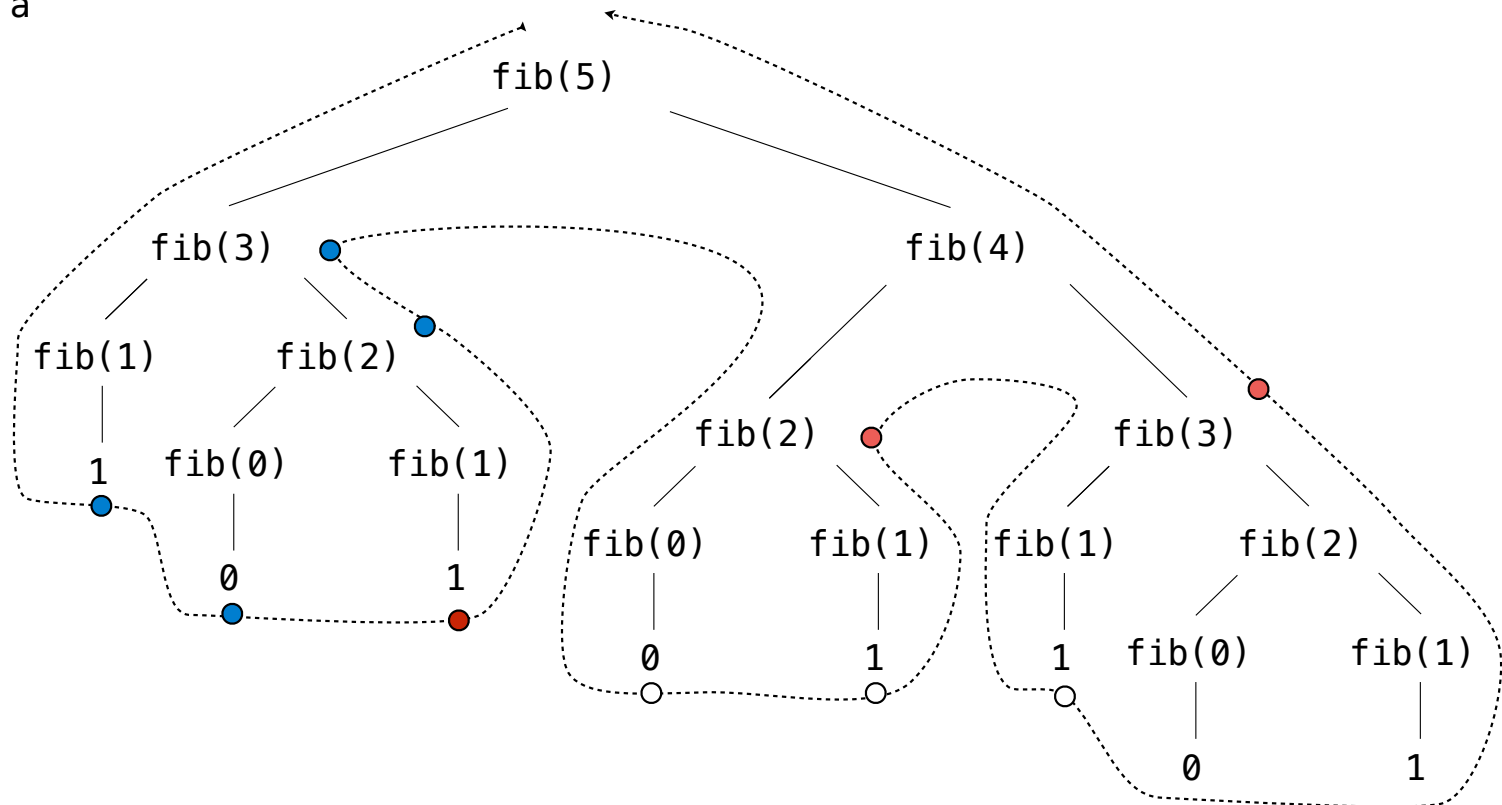
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



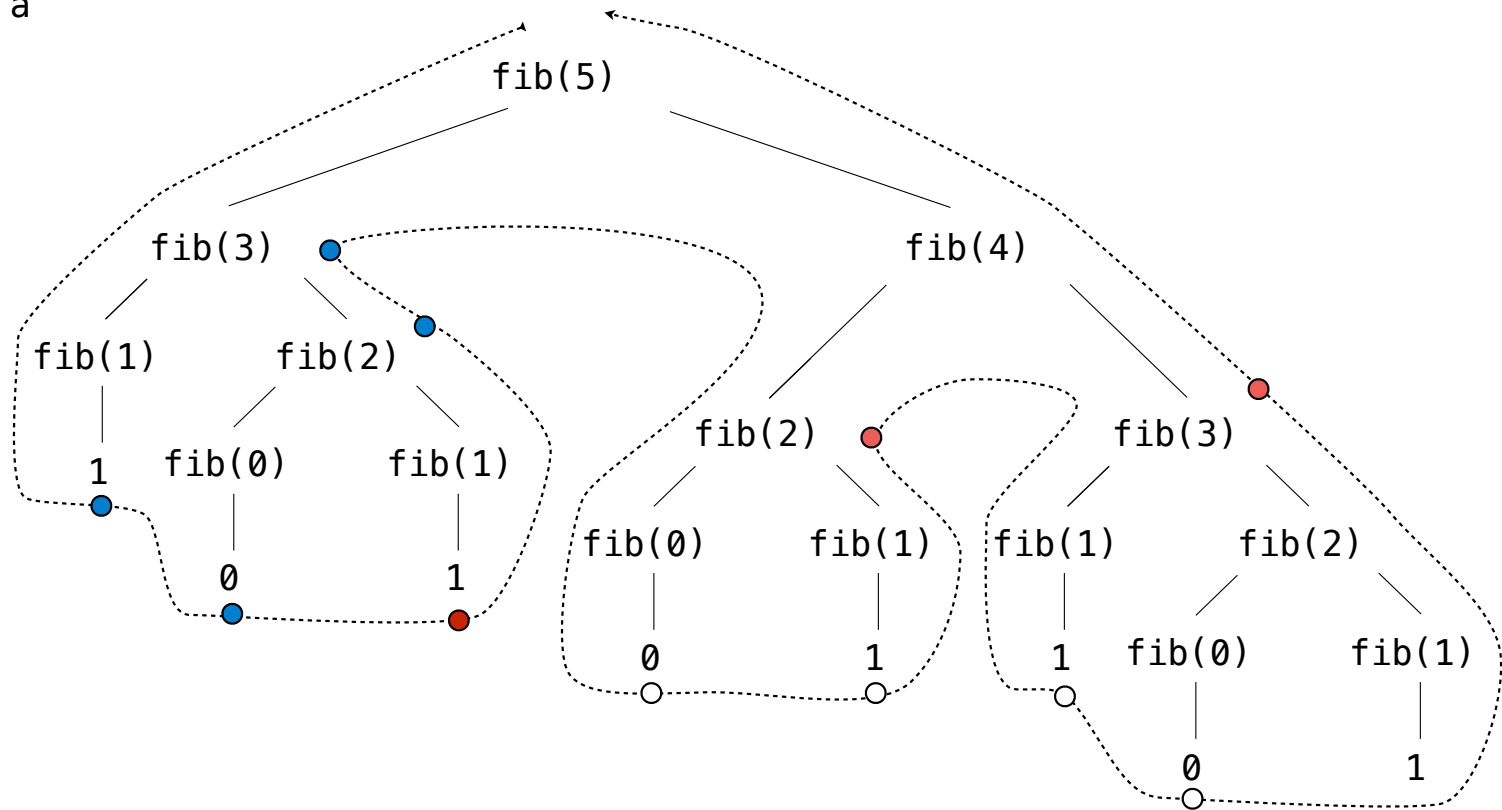
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



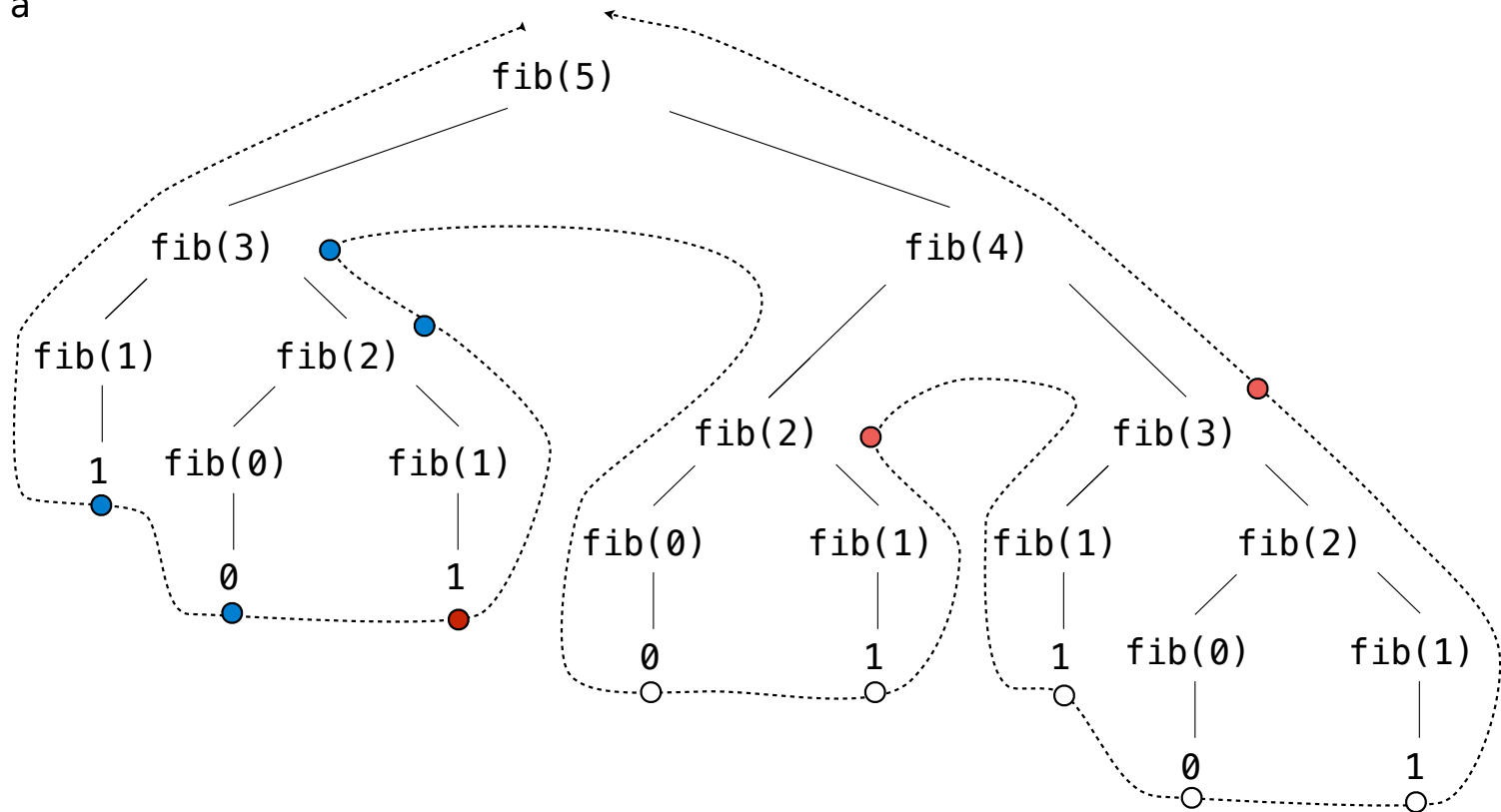
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



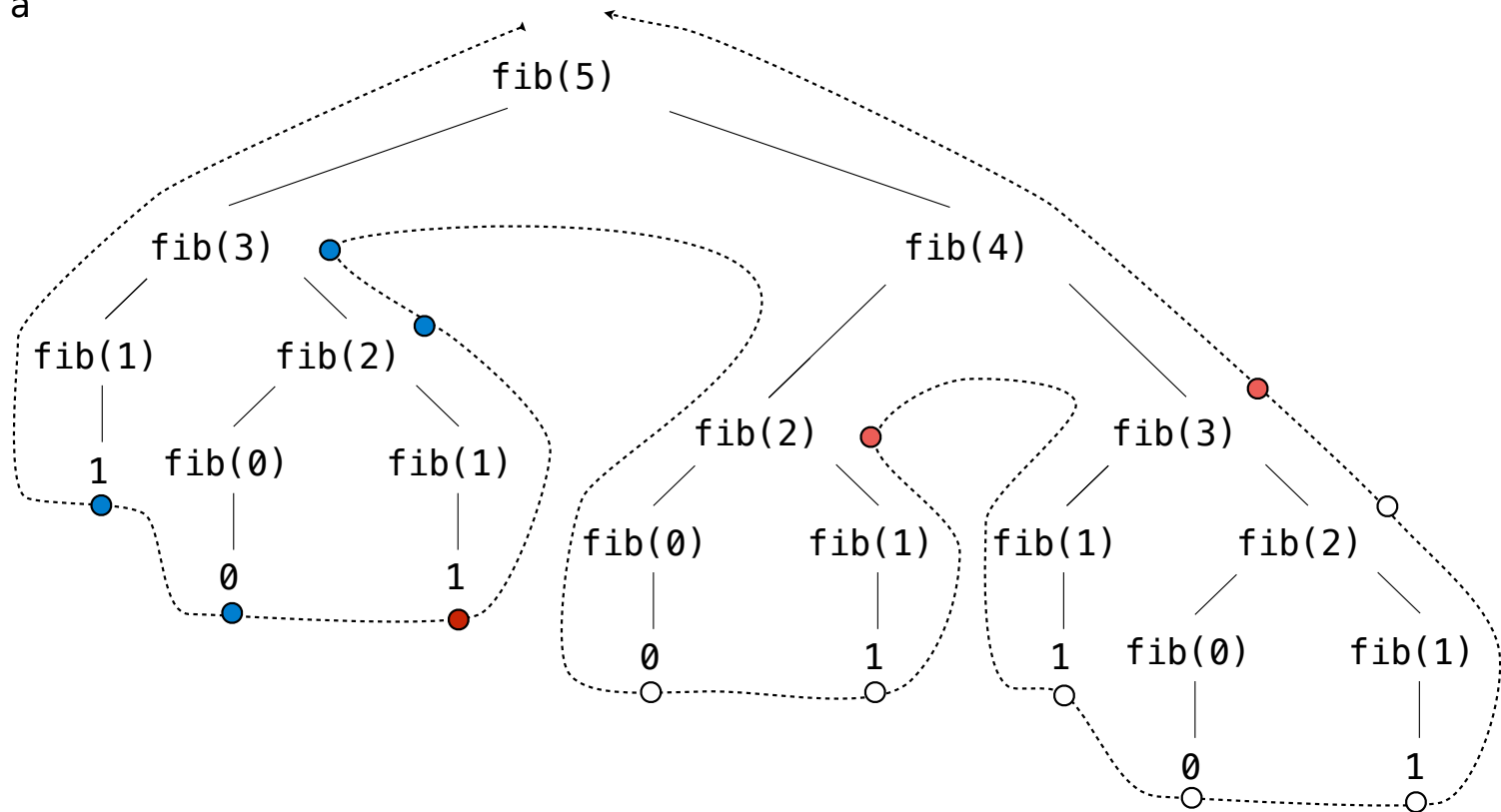
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



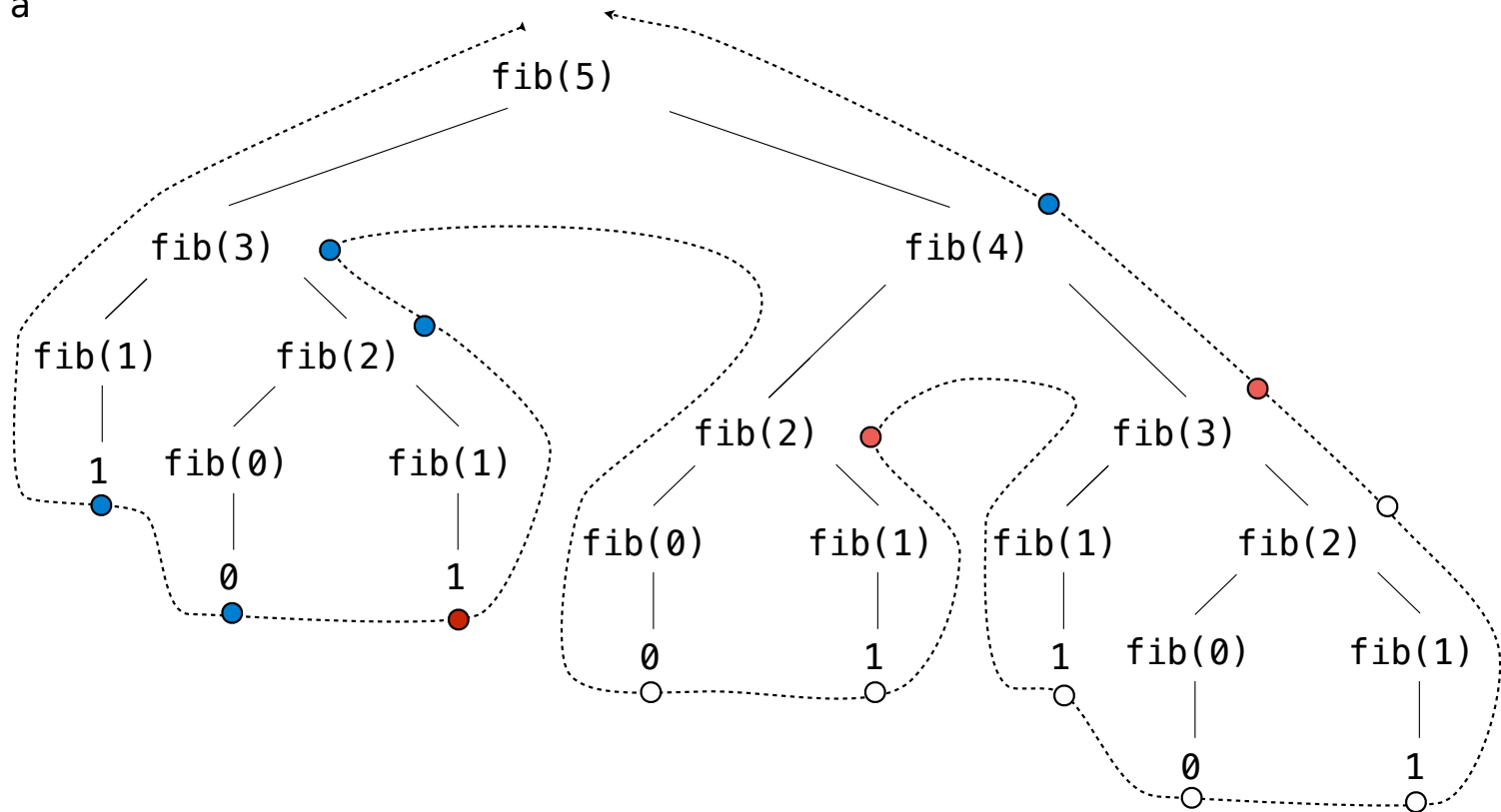
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



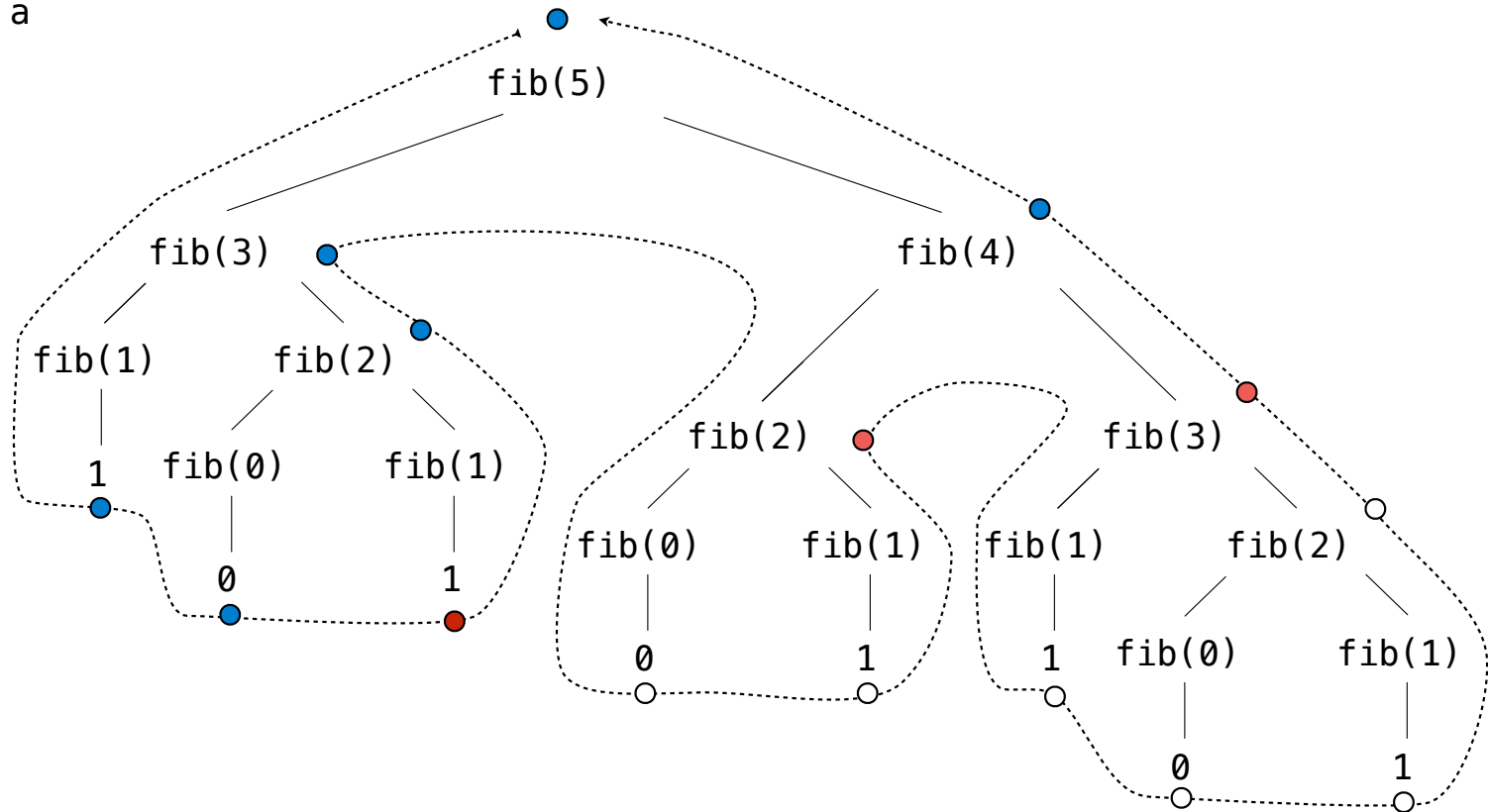
Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped



Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing

Memoization:

- Returned by fib
- Found in cache
- Skipped

