Composition

Linked Lists

Linked List Structure

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

\[
\text{Link(3, Link(4, Link(5, Link.empty)))}
\]

The first (zeroth) element is an attribute value

The rest of the elements are stored in a linked list

Linked List Class

A class attribute represents an empty linked list

\[
\text{Link.empty}
\]

Linked list class: attributes are passed to \texttt{__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

    @property
def second(self):
        return self.rest.first

    @second.setter
    def second(self, value):
        self.rest.first = value
```

Property Methods

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

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

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.
**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):**
- 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 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 = self.fib_tree(n-2)
            right = self.fib_tree(n-1)
            fib_n = left.label + right.label
            return Tree(fib_n, [left, right])
```

**Example: Pruning Trees**

Removing subtrees from a tree is called pruning.

Prune branches before recursive processing.

```
def prune(t, n):
    print("****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)
```

**Tree Mutation**

Example: Pruning Trees

Removing subtrees from a tree is called pruning.

Prune branches before recursive processing.

```
def prune(t, n):
    if n == 0 or n == 1:
        return t
    else:
        left = prune(t.left, n)
        right = prune(t.right, n)
        return Tree(t.label, [left, right])
```

**Memoization:**
- Returned by fib
- Found in cache
- Skipped