Trees

Friday, January 6, 2023 12:11 AM

Trees

Tree-Structured Data

constructor

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

def label(t):
    return t[0]

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

selectors

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

Helper function tells you whether or not a tree is a leaf

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

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

Selectors are needed because we just have the attributes

```
A tree can contains other trees:

[5, [6, 7], 8, [[9], 10]]

(+ 5 (- 6 7) 8 (* (- 9) 10))

(S
  (NP (JJ Short) (NNS cuts))
  (VP (VBP make)
      (NP (JJ long) (NNS delays)))
  (. .))

<ul>
  <li>Midterm <b>1</b></li>
  <li>Midterm <b>2</b></li>
</ul>

Tree processing often involves
recursive calls on subtrees
```

Tree Processing

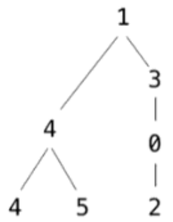
Friday, January 6, 2023 12:23 AM

Some methods of solving

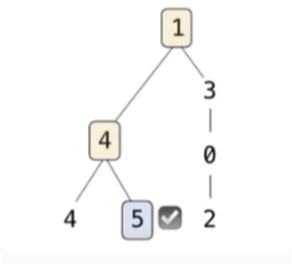
Solving Tree Problems

Implement bigs, which takes a Tree instance t containing integer labels.
It returns the number of nodes in t whose labels are larger than any labels of their ancestor nodes.

```
Def bigs(t):
    "return the number of nodes in t that are larger that are larger than all their ancestors."
    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
```



Try drawing the tree, a diagram is always helpful



The root label is always larger than all of its ancestors cuz it doesn't have any ancestor to be smaller than

```
if t.is_leaf():
    return ___
else:
    return ___([___ for b in t.branches])
```

Somehow increment the total count

Somehow track a list of ancestors

```
if node.label > max(ancestors):
```

Somehow track the largest ancestor

```
if node.label > max_ancestors:
```

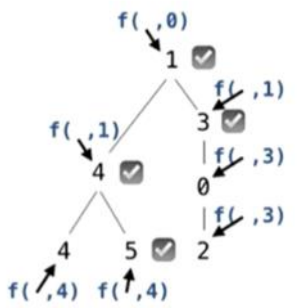
It's better to track the largest ancestor node

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.
```

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors.

    >>> a = Tree(1, [Tree(4, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(2)])])])
    >>> bigs(a)
    4
    """
    def f(a, x):        Somehow track the
                        largest ancestor
        if ________________________________________:

            return 1 + ____________________________

        else:

            return ________________________________

    return ________________________________________
```

The X is the largest one tracked
If a.label > x:
    Return 1+ sum([f(b, a.label) for b in a.branches])
Else:
    Return sum([f(b, x) for b in a.branches])
Return f(t, table - 1)

```
    >>> bigs(a)
    4
    """
    def f(a, x):        Somehow track the
                        largest ancestor
A node in t   max_ancestor
        if a.label > x          node.label > max_ancestors   :

            return 1 + sum([f(b, a.label) for b in a.branches])

        else:           Somehow increment the total count

            return  sum([f(b, x) for b in a.branches])

    return f(t, t.label - 1)    Root label is always larger than its ancestors

                   Some initial value for the largest ancestor so far...
```



Using diagram in order to go back and check your program will do for checking your work

But figuring out why the function is not doing what it's supposed to be doing almost always involves you manually tracing through what it's doing by understanding exactly how expressions and statements are evaluated and executed in a programming language

And checking your work is a critical step in solving problems

Recursive Accumulation

Friday, January 6, 2023 10:52 PM

Return the accumulation so far and so it's the return value of the recursive function that's
Keeping track of what you want to return in the end

Another option is to initialize some accumulation to zero or an empty list and then populate it as you go

```
def bigs(t):
    """Return the number of nodes in t that are larger than all their ancestors."""
    n = 0
                   Somehow track the
    def f(a, x):   largest ancestor
        nonlocal n
        if a.label > x:        node.label > max_ancestors
            n += 1
                               Somehow increment
        for b in a.branches:   the total count
            f(b, max(a.label, x))
    f(t, t.label - 1)   Root label is always larger than its ancestors
    return n
```

Designing Functions

Friday, January 6, 2023 11:17 PM

A book of on how to design programs

How to design programs

From Problem Analysis to Data Definitions
Identify the information that must be represented and how it is represented in the chosen programming language. Formulate data definitions and illustrate them with examples.

Signature, Purpose Statement, Header
State what kind of data the desired function consumes and produces. Formulate a concise answer to the question *what* the function computes. Define a stub that lives up to the signature.

Functional Examples
Work through examples that illustrate the function's purpose.

Function Template
Translate the data definitions into an outline of the function.

What would be defined of function within the function
And if, else if, some stuff things

wrote down the key expressions that were going to be used somewhere in our implementation

Function Definition
Fill in the gaps in the function template. Exploit the purpose statement and the examples.

Finally the function definition
Fill in the gaps in the function template

Testing
Articulate the examples as tests and ensure that the function passes all. Doing so discovers mistakes. Tests also supplement examples in that they help others read and understand the definition when the need arises—and it will arise for any serious program.

Don't think about it generally but instead focusing on its example that you walk through in order to illustrate the functions purpose

Once you know the how to make them and think about the examples

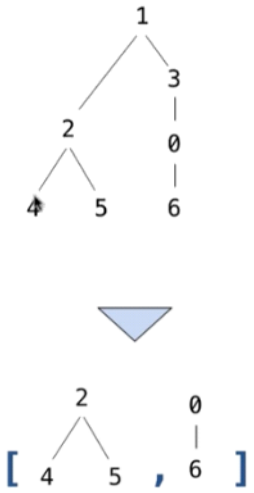
Applying the design Process

Friday, January 6, 2023 11:52 PM

```
Implement smalls, which takes a Tree instance t containing integer labels. It returns the
non-leaf nodes in t whose labels are smaller than any labels of their descendant nodes.
```

```
def smalls(t):
    """Return the non-leaf nodes in t that are smaller than all their descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []
    def process(t):



    process(t)
    return result
```

Then write diagram



Dev a template

```
def smalls(t):     Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []            Signature: Tree -> number
    def process(t):        "Find smallest label in t & maybe add t to result"
```

Look through the process

About the template

```
def process(t):          "Fin
    if t.is_leaf():
        return t.label
    else:


        return min(...)
process(t)
return result
```

To

```
def process(t):          "Find smallest label in t & maybe a
    if t.is_leaf():
        return ______________________________________
    else:
        smallest = ______________________________
        if ______________________________________:
            ______________________________________
        return min(smallest, t.label)
process(t)
return result
```

```
def smalls(t):          Signature: Tree -> List of Trees
    """Return the non-leaf nodes in t that are smaller than all their descendants.

    >>> a = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(0, [Tree(6)])])])
    >>> sorted([t.label for t in smalls(a)])
    [0, 2]
    """
    result = []          Signature: Tree -> number
    def process(t):      "Find smallest label in t & maybe add t to result"
        if t.is_leaf():
            return ______________t.label______________
        else:
            smallest = __min([process(b) for b in t.branches])__
            if ______t.label < smallest______:
                ______result.append( t )______
            return min(smallest, t.label)
    process(t)
    return result
```

smallest label in a branch of t