Week 7 - Trees

Updated Oct 23, 8am

While lists are a really useful data structure, not all data has a natural linear order. One example from last week was the family tree, which has a branching structure that is difficult to model with lists.

This week, we'll introduce the `Tree`, another recursive data structure that you can view as a generalization of lists. The basic tree structure is the following:

- A tree has a *root* node which stores a single item. (Analogous to the *head* attribute of the linked list.)
- A tree has zero or more *subtrees*, each of which is a tree. (This is where trees and lists differ. You can think of a list as a tree that only ever has zero or one subtrees.)
- A tree can also be empty, and contain neither a root node nor any subtrees.

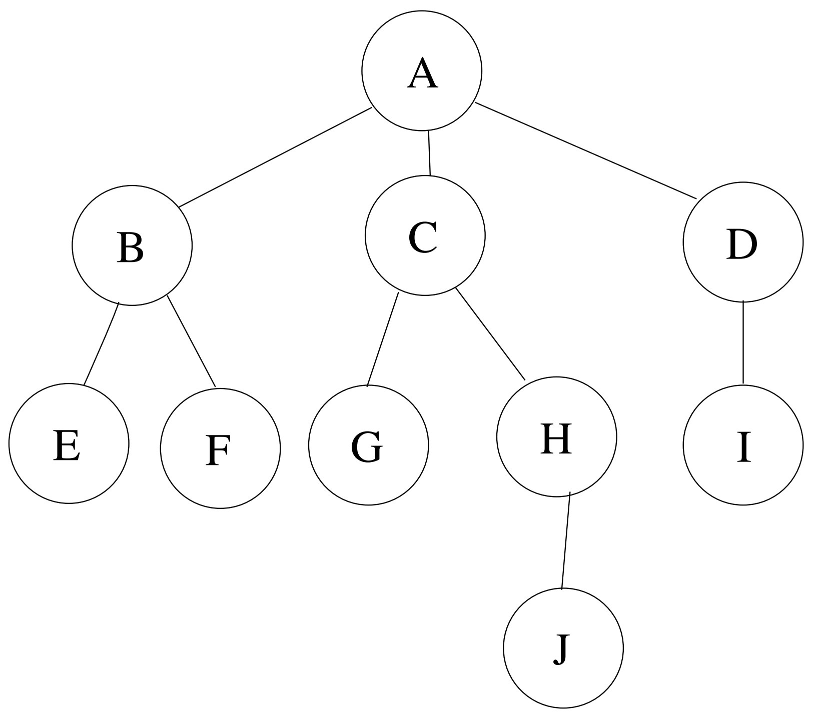
We can implement this in Python as follows, using the same `EmptyValue` class from last week to signify an empty tree.

```python
class EmptyValue:
    pass

class Tree:

    def __init__(self, root=EmptyValue):
        """ (Tree, object) -> NoneType """
        self.root = root
        self.subtrees = []

    def is_empty(self):
        """ (Tree) -> bool """
        return self.root is EmptyValue
```

Our constructor here always creates either an empty tree (when `root is EmptyValue`), or a tree with just an item at the root and no children. These two cases are generally the base cases when dealing with trees. When you write your recursive functions, your base cases will generally be one or both of these types of trees.

For our convenience in building trees, we've also included an `add_subtrees` method that adds trees to the list of children, which you can find in the source code.

Tree Terminology



A tree is either **empty** or **non-empty**. Every non-empty tree has a **root node** (which is generally drawn at the top), connected to zero or more **subtrees**. The root node of the above tree is labelled A. The **size** of a tree is the number of nodes in the tree. *What's the relationship between the size of a tree and the size of its subtrees?*

A **leaf** is a node with no subtrees. The leaves of the above tree are labelled E, F, G, I, and J. *What's the relationship between the number of leaves of a tree and the number of leaves of its subtrees?*

The **height** of a tree is the length of the *longest* path from its root to one of its leaves. The height of the above tree is 3. *What's the relationship between the height of a tree and the heights of its subtrees?*

The **children** of a node are all nodes directly connected underneath that node. The children of node A are nodes B, C, and D. Note that the number of children of a node is equal to the number of subtrees of a node, but that these two concepts are quite different. The **descendants** of a node are its children, the children of its children, etc. *What's the relationship between the descendants of a node and the descendants of its children?*

Similarly, the **parent** of a node is the one immediately above and connected to it; each node has one parent, except the root, which has no parent. The **ancestors** of a node are its parent, the parent of its parent, etc.

There's a reason I keep asking the same question: understanding the relationship between a tree and its subtrees is precisely understanding the recursive structure of the trees. Understand this and you'll be able to write extremely simple and elegant code for processing trees.

Here's a quick example: the size of a non-empty tree is the sum of the sizes of its subtrees, plus 1 for the root; the size of an empty tree is, of course, 0. This single observation immediately lets us write the following recursive function for computing the size of a tree:

```
def size(self):
    if self.is_empty():
        return 0
    else:
        size = 1
        for subtree in subtrees:
            size += subtree.size()
        return size
```

Traversing a Tree

Because lists have a natural order of their elements, they're pretty straightforward to traverse, meaning (among other things) that it's easy to print out all of the elements. How might we accomplish the same with a tree?

Here's an idea: print out the root, then recursively print out all of the subtrees. That's pretty easy to implement. Note that in our implementation, the base case is when the tree is empty, and in this case the method does nothing.

```
def print_tree(self):
    if not self.is_empty():
        print(self.root)
        for subtree in self.subtrees:
            subtree.print_tree()
```

Consider what happens when we run this on the following tree structure:

```
>>> t1 = Tree(1)
>>> t2 = Tree(2)
>>> t3 = Tree(3)
>>> t4 = Tree(4)
>>> t4.add_subtrees([t1, t2, t3])
>>> t5 = Tree(5)
>>> t5.add_subtrees([t4])
>>> t5.print_tree()
5
4
1
2
3
```

We know that 5 is the root of the tree, but it's ambiguous how many children it has. Let's try to use *indentation* to differentiate between the root node and all of its subtrees. But how do we do this? Looking at the previous code, what we want is for when `subtree.print_tree()` is called, that everything that's printed out in the recursive call is indented.

But remember, we *aren't* tracing the code, and in particular, we can't "reach in" to make the recursive calls act differently. However, this problem - we want some context from where a method is called to influence what happens inside the method - is **exactly** the problem that parameters are meant to solve.

So we'll pass in an extra parameter for the *depth* of the current tree, which will be reflected by adding a corresponding number of whitespace characters to printing.

```
def print_tree_indent(self, depth=0):
    if not self.is_empty():
        print(depth * '  ' + str(self.root))
        for subtree in self.subtrees:
            subtree.print_tree(depth + 1)
```

Side Note: Optional Parameters

In both the constructor and `print_tree_indent`, we used an *optional parameter* that could either be included or not included when the function is called.

So we can call `t.print_tree_indent(5)`, which sets its `depth` parameter to `5`, as we would expect. However, we can also call the method as `t.print_tree_indent()`, in which case it sets the `depth` parameter to `0`.

Optional parameters are a powerful Python feature that allows us to write more flexible functions and methods to be used in a variety of situations. Two important points to keep in mind, though:

- All optional parameters must appear after all of the required parameters in the function signature
- Do not use mutable values like lists for your optional parameters; instead, use immutable values like ints, strings, and `None`

Computer Science
UNIVERSITY OF TORONTO

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)