Trees 10

Computer Science 88

April 11, 2022

1 Trees

In computer science, **trees** are recursive data structures that are widely used in various settings.

Contrary to our ideas of a tree, in computer science, a tree branches downward. The **root** of a tree starts at the top, and the **leaves** are at the bottom.

A tree is considered a recursive data structure because every branch from a node is also a tree.

Some terminology regarding trees:

- **Parent node**: A node that has branches. Parent nodes can have multiple branches.
- **Child node**: A node that has a parent. A child node can only belong to one parent.
- **Root**: The top node of the tree. In our example, the node that contains 7 is the root.
- **Label**: The value at a node. In our example, all of the integers are values.
- **Leaf**: A node that has no branches. In our example, the nodes that contain -4, 0, 6, 17, and 20 are leaves.
- **Branch**: A subtree of the root. Note that trees have branches, which are trees themselves: this is why trees are *recursive* data structures.
- **Depth**: How far away a node is from the root. In other words, the number of edges between the root of the tree to the node. In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. Since there are no edges between the root of the tree and itself, the depth of the root is 0.

- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing -4, 0, 6, and 17 are all the "lowest leaves," and they have depth 4. Thus, the entire tree has height 4.

In computer science, there are many different types of trees. Some vary in the number of branches each node has; others vary in the structure of the tree. Recall the tree abstract data type: a tree is defined as having a label and some branches. Trees can be implemented as an ADT, but we will only be focusing on the Tree class this semester. Below is the most basic implementation of a Tree class that we will be using.

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

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

Notice that with this implementation we can mutate a tree using attribute assignment, which wouldn't be possible in the implementation using lists in an ADT.

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.value = 5
>>> t.value
5
```

1.1 Questions

1. What would Python display? If you believe an expression valueates to a *Tree* object, write *Tree*.

```
>>> t0 = Tree(0)
>>> t0.value


>>> t0.branches


>>> t1 = Tree(0, [1, 2])#Is this a valid tree?

>>> t2 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
>>> t2.branches[0]

>>> t2.branches[1].branches[0].value
```

2. Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.value
    2
    >>> t.branches[0].branches[0].value
    4
    """
```

3. Write a function that combines the values of two trees `t1` and `t2` together with the `combiner` function. Assume that `t1` and `t2` have identical structure. This function should return a new tree.

```
def combine_tree(t1, t2, combiner):
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = combine_tree(a, b, mul)
    >>> combined.value
    4
    >>> combined.branches[0].value
    10
    """
```