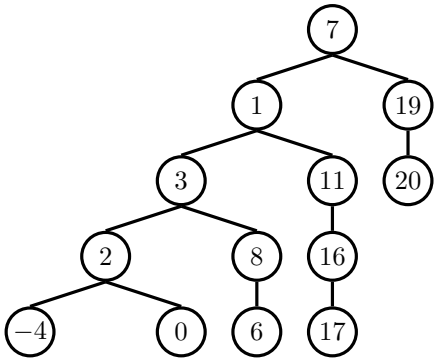
CS 61A Fall 2019

Trees, Mutability, and Nonlocal

Discussion 5: October 2, 2019

1 Trees



In computer science, **trees** are recursive data structures that are widely used in various settings. The diagram to the right is an example of a tree.

Notice that the tree branches downward. In computer science, the **root** of a tree starts at the top, and the **leaves** are at the bottom.

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.

Implementation

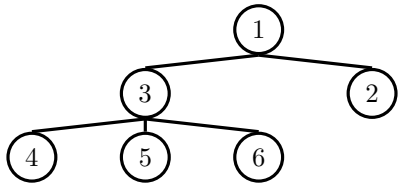
A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is an abstract data type, our choice to use lists is just an implementation detail.

- The arguments to the constructor `tree` are the value for the root node and a list of branches.
- The selectors for these are `label` and `branches`.

Note that `branches` returns a list of trees and not a tree directly. It's important to distinguish between working with a tree and working with a **list of** trees.

We have also provided a convenience function, `is_leaf`.

Let's try to create the tree below.




```
# Example tree construction
t = tree(1,
      [tree(3,
          [tree(4),
           tree(5),
           tree(6)]),
      tree(2)])
```

Questions

1.1 Write a function that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    """Return the height of a tree.

    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    """
```

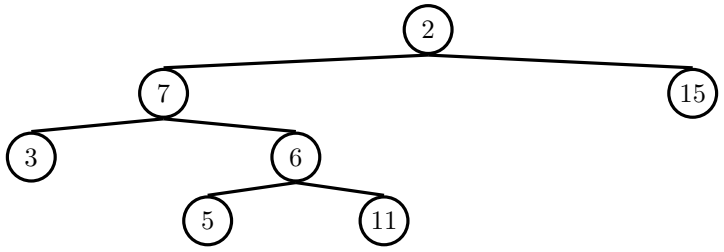
1.2 Write a function that takes in a tree and squares every value. It should return a new tree. You can assume that every item is a number.

```
def square_tree(t):
    """Return a tree with the square of every element in t
    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(square_tree(numbers))
    1
      4
        9
        16
      25
        36
          49
        64
    """
```

1.3 Write a function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`.

If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`




```
def find_path(tree, x):
    """
    >>> t = tree(2, [tree(7, [tree(3), tree(6, [tree(5), tree(11)])] ), tree(15)])
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """

    if _______________________________:

        return _______________________________

    _______________________________:

        path = _______________________________

        if _______________________________:

            return _______________________________
```

2 Mutation

Let's imagine you order a mushroom and cheese pizza from La Val's, and that they represent your order as a list:

```
>>> pizza = ['cheese', mushrooms']
```

A couple minutes later, you realize that you really want onions on the pizza. Based on what we know so far, La Val's would have to build an entirely new list to add onions:

```
>>> pizza = ['cheese', mushrooms']
>>> new_pizza = pizza + ['onions'] # creates a new python list
>>> new_pizza
['cheese', mushrooms', 'onions']
>>> pizza # the original list is unmodified
['cheese', 'mushrooms']
```

This is silly, considering that all La Val's had to do was add onions on top of `pizza` instead of making an entirely new pizza.

We can fix this issue with **list mutation**. In Python, some objects, such as lists and dictionaries, are **mutable**, meaning that their contents or state can be changed over the course of program execution. Other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created.

Therefore, instead of building a new pizza, we can just mutate `pizza` to add some onions!

```
>>> pizza.append('onions')
>>> pizza
['cheese', 'mushrooms', 'onions']
```

`append` is what's known as a method, or a function that belongs to an object, so we have to call it using dot notation. We'll talk more about methods later in the course, but for now, here's a list of useful list mutation methods:

1. `append(el)`: Adds `el` to the end of the list
2. `extend(lst)`: Extends the list by concatenating it with `lst`
3. `insert(i, el)`: Insert `el` at index `i` (does not replace element but adds a new one)
4. `remove(el)`: Removes the first occurrence of `el` in list, otherwise errors
5. `pop(i)`: Removes and returns the element at index `i`

We can also use the familiar indexing operator with an assignment statement to change an existing element in a list. For example, we can change the element at index 1 and to `'tomatoes'` like so:

```
>>> pizza[1] = 'tomatoes'
>>> pizza
['cheese', 'tomatoes', 'onions']
```

Questions

2.1 What would Python display? In addition to giving the output, draw the box and pointer diagrams for each list to the right.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = lst1
>>> lst1 is lst2


>>> lst2.extend([5, 6])
>>> lst1[4]


>>> lst1.append([-1, 0, 1])
>>> -1 in lst2


>>> lst2[5]


>>> lst3 = lst2[:]
>>> lst3.insert(3, lst2.pop(3))
>>> len(lst1)


>>> lst1[4] is lst3[6]


>>> lst3[lst2[4][1]]


>>> lst1[:3] is lst2[:3]


>>> lst1[:3] == lst3[:3]
```

2.2 Write a function that takes in a value `x`, a value `el`, and a list and adds as many `el`'s to the end of the list as there are `x`'s. **Make sure to modify the original list using list mutation techniques.**

```
def add_this_many(x, el, lst):
    """ Adds el to the end of lst the number of times x occurs
    in lst.
    >>> lst = [1, 2, 4, 2, 1]
    >>> add_this_many(1, 5, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5]
    >>> add_this_many(2, 2, lst)
    >>> lst
    [1, 2, 4, 2, 1, 5, 5, 2, 2]
    """
```

2.3 Write a function that takes in a sequence `s` and a function `fn` and returns a dictionary.

The values of the dictionary are lists of elements from `s`. Each element `e` in a list should be constructed such that `fn(e)` is the same for all elements in that list. Finally, the key for each value should be `fn(e)`.

```
def group_by(s, fn):
    """
    >>> group_by([12, 23, 14, 45], lambda p: p // 10)
    {1: [12, 14], 2: [23], 4: [45]}
    >>> group_by(range(-3, 4), lambda x: x * x)
    {0: [0], 1: [-1, 1], 4: [-2, 2], 9: [-3, 3]}
    """
```

3 Nonlocal

Until now, you've been able to access names in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a binding in a parent frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal name
        num = num + 1 # modifies num in the stepper frame
        return num
    return step

>>> step1 = stepper(10)
>>> step1()                # Modifies and returns num
11
>>> step1()                # num is maintained across separate calls to step
12
>>> step2 = stepper(10)    # Each returned step function keeps its own state
>>> step2()
11
```

As illustrated in this example, `nonlocal` is useful for maintaining state across different calls to the same function.

However, there are two important caveats with `nonlocal` names:

- **Global names** cannot be modified using the `nonlocal` keyword.
- **Names in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal binding with the same name in a single frame.

Because `nonlocal` lets you modify bindings in parent frames, we call functions that use it **mutable functions**.

Questions

3.1 Given the definition of `make_shopkeeper` below, draw the environment diagram.

```
def make_shopkeeper(total_gold):
    def buy(cost):
        nonlocal total_gold
        if total_gold < cost:
            return 'Go farm some more champions'
        total_gold = total_gold - cost
        return total_gold
    return buy

infinity_edge, zeal, gold = 3800, 1100, 3800
shopkeeper = make_shopkeeper(gold - 1000)
shopkeeper(zeal)
shopkeeper(infinity_edge)
```

3.2 Write a function that takes in a number `n` and returns a one-argument function. The returned function takes in a function that is used to update `n`. It should return the updated `n`.

```
def memory(n):
    """
    >>> f = memory(10)
    >>> f(lambda x: x * 2)
    20
    >>> f(lambda x: x - 7)
    13
    >>> f(lambda x: x > 5)
    True
    """
```