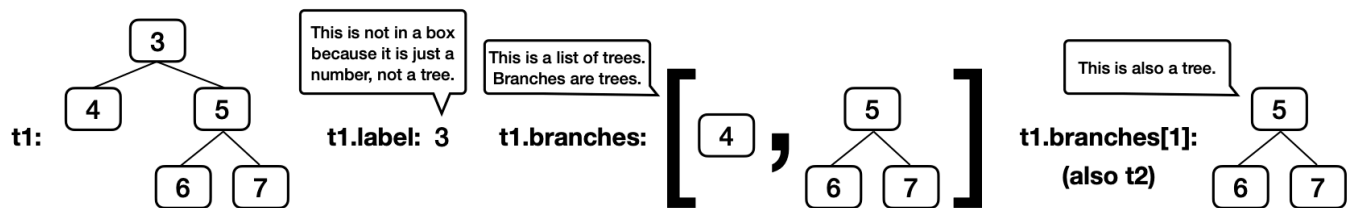


## Trees

For a `Tree` instance `t`: - Its root label can be any value, and `t.label` evaluates to it. - Its branches are all `Tree` instances, and `t.branches` evaluates to a list of branches, which is a list of `Tree` instances. - It is called a leaf if it has no branches, and `t.is_leaf()` returns whether `t` is a leaf. - A new `Tree` with the same root label and branches can be constructed with `Tree(t.label, t.branches)`.

Here's an example tree `t1`, for which its branch `t1.branches[1]` is `t2`.

```
t2 = Tree(5, [Tree(6), Tree(7)])  
t1 = Tree(3, [Tree(4), t2])
```



### Example Tree

A path is a sequence of nodes in which each is the parent of the next.

You don't need to know how the `Tree` class is implemented in order to use it correctly, but here is the implementation from lecture.

```

class Tree:
    """A tree is a label and a list of branches."""
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    # The rest of the class just determines how trees are displayed.

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(repr(self.label), branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self):
        lines = []
        for b in self.branches:
            for line in b.indented():
                lines.append('  ' + line)
        return [str(self.label)] + lines

```

**Q1: Min Tree**

What value is bound to `result`?

```

get_label = lambda t: t.label
result = min(max([t1, t2], key=get_label).branches, key=get_label).label

```

**Solution**

6: `max([t1, t2], key=get_label)` evaluates to the `t2` tree because its label 5 is larger than `t1`'s label 3. Among `t2`'s branches (which are leaves), the left one labeled 6 has a smaller label.

Here's a quick refresher on how key functions work with `max` and `min`,

`max(s, key=f)` returns the item `x` in `s` for which `f(x)` is largest.

```
>>> s = [-3, -5, -4, -1, -2]
>>> max(s)
-1
>>> max(s, key=abs)
-5
>>> max([abs(x) for x in s])
5
```

Therefore, `max([t1, t2], key=get_label)` returns the tree with the largest label, in this case `t2`.

## Q2: Has Path

Implement `has_path`, which takes a `Tree` instance `t` and a list `p`. It returns whether there is a path from the root of `t` with labels `p`. For example, `t1` has a path from its root with labels `[3, 5, 6]` but not `[3, 4, 6]` or `[5, 6]`.

**Important:** Before trying to implement this function, discuss these questions from lecture about the recursive call of a tree processing function: - What recursive calls will you make? - What type of values do they return? - What do the possible return values mean? - How can you use those return values to complete your implementation?

If you get stuck, you can view our answers to these questions by clicking the hint button below.

### What recursive calls will you make?

As you usual, you will call `has_path` on each branch `b`. You'll make this call after comparing `p[0]` to `t.label`, and so the second argument to `has_path` will be the rest of `p`: `has_path(b, p[1:])`.

### What type of values do they return?

`has_path` always returns a `bool` value: `True` or `False`.

### What do the possible return values mean?

If `has_path(b, p[1:])` returns `True`, then there is a path through branch `b` for which `p[1:]` are the node labels.

### How can you use those return values to complete your implementation?

If you have already checked that `t.label` is equal to `p[0]`, then a `True` return value means there is a path through `t` with labels `p` using that branch `b`. A `False` value means there is no path through that branch, but there might be path through a different branch.

```
def has_path(t, p):
    """Return whether Tree t has a path from the root with labels p.

    >>> t2 = Tree(5, [Tree(6), Tree(7)])
    >>> t1 = Tree(3, [Tree(4), t2])
    >>> has_path(t1, [5, 6])          # This path is not from the root of t1
    False
    >>> has_path(t2, [5, 6])          # This path is from the root of t2
    True
    >>> has_path(t1, [3, 5])          # This path does not go to a leaf, but that's ok
    True
    >>> has_path(t1, [3, 5, 6])       # This path goes to a leaf
    True
    >>> has_path(t1, [3, 4, 5, 6])    # There is no path with these labels
    False
    """
    if p == [t.label]:
        return True
    elif t.label != p[0]:
        return False
    else:
        for b in t.branches:
            if has_path(b, p[1:]):
                return True
        return False
```

# Efficiency

Tips for finding the order of growth of a function's runtime:

- If the function is recursive, determine the number of recursive calls and the runtime of each recursive call.
- If the function is iterative, determine the number of inner loops and the runtime of each loop.
- Ignore coefficients. A function that performs  $n$  operations and a function that performs  $100 * n$  operations are both linear.
- Choose the largest order of growth. If the first part of a function has a linear runtime and the second part has a quadratic runtime, the overall function has a quadratic runtime.
- In this course, we only consider constant, logarithmic, linear, quadratic, and exponential runtimes.

### Q3: The First Order...of Growth

What is the efficiency of `rey`?

```
def rey(finn):  
    poe = 0  
    while finn >= 2:  
        poe += finn  
        finn = finn / 2  
    return
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

**Solution:** Logarithmic, because our while loop iterates at most  $\log(\text{finn})$  times, due to `finn` being halved in every iteration. This is commonly known as  $\Theta(\log(\text{finn}))$  runtime. Another way of looking at this if you duplicate the input, we only add a single iteration to the time, which also indicates logarithmic.

What is the efficiency of `mod_7`?

```
def mod_7(n):  
    if n % 7 == 0:  
        return 0  
    else:  
        return 1 + mod_7(n - 1)
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

**Solution:** Constant, since in the worst case scenario our function `mod_7` will require 6 recursive calls to reach the base case. Consider the worst case where we have an input `n` such that our first call to `mod_7` evaluates `n % 7` as 6. Each recursive call will decrement `n` by 1, allowing us to eventually reach the base case of returning 0 in 6 recursive calls (`n` will range from 0 to 6). Since the growth of the computation is independent of the input, we say this is constant, which is commonly known as a  $\Theta(1)$  runtime.