CS 61A
Summer 2026

Mutable Trees

Discussion 7: July 21, 2026

Mutable Trees

A `Tree` instance has two instance attributes:

- `label` is the value stored at the root of the tree.
- `branches` is a list of `Tree` instances that hold the labels in the rest of the tree.

The `Tree` class (with its `__repr__` and `__str__` methods omitted) is defined as:

```
class Tree:
    """A tree has a label and a list of branches.

    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """
    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
```

To construct a `Tree` instance from a label `x` (any value) and a list of branches `bs` (a list of `Tree` instances) and give it the name `t`, write `t = Tree(x, bs)`.

For a tree `t`:

- Its root label can be any value, and `t.label` evaluates to it.
- Its branches are always `Tree` instances, and `t.branches` evaluates to the **list** of its branches.
- `t.is_leaf()` returns `True` if `t.branches` is empty and `False` otherwise.
- To construct a leaf with label `x`, write `Tree(x)`.

Displaying a tree `t`:

- `repr(t)` returns a Python expression that evaluates to an equivalent tree.
- `str(t)` returns one line for each label indented once more than its parent with children below their parents.

```
>>> t = Tree(3, [Tree(1, [Tree(4), Tree(1)]), Tree(5, [Tree(9)])])

>>> t         # displays the contents of repr(t)
Tree(3, [Tree(1, [Tree(4), Tree(1)]), Tree(5, [Tree(9)])])

>>> print(t)  # displays the contents of str(t)
3
  1
    4
    1
  5
    9
```

Changing (also known as mutating) a tree `t`:

- `t.label = y` changes the root label of `t` to `y` (any value).
- `t.branches = ns` changes the branches of `t` to `ns` (a list of `Tree` instances).
- Mutation of `t.branches` will change `t`. For example, `t.branches.append(Tree(y))` will add a leaf labeled `y` as the right-most branch.
- Mutation of any branch in `t` will change `t`. For example, `t.branches[0].label = y` will change the root label of the left-most branch to `y`.

```
>>> t.label = 3.0
>>> t.branches[1].label = 5.0
>>> t.branches.append(Tree(2, [Tree(6)]))
>>> print(t)
3.0
  1
    4
    1
  5.0
    9
  2
    6
```

Here is a summary of the differences between the tree data abstraction implemented as a functional abstraction vs. implemented as a class:

-	Tree constructor and selector functions	Tree class
Constructing a tree	To construct a tree given a `label` and a list of `branches`, we call `tree(label, branches)`	To construct a tree object given a `label` and a list of `branches`, we call `Tree(label, branches)` (which calls the `Tree.__init__` method).
Label and branches	To get the label or branches of a tree `t`, we call `label(t)` or `branches(t)` respectively	To get the label or branches of a tree `t`, we access the instance attributes `t.label` or `t.branches` respectively.
Mutability	The functional tree data abstraction is immutable (without violating its abstraction barrier) because we cannot assign values to call expressions	The `label` and `branches` attributes of a `Tree` instance can be reassigned, mutating the tree.
Checking if a tree is a leaf	To check whether a tree `t` is a leaf, we call the function `is_leaf(t)`	To check whether a tree `t` is a leaf, we call the method `t.is_leaf()`. This method can only be called on `Tree` objects.

Q1: WWPD: Trees

What would Python display?

```
>>> t = Tree(1, Tree(2))
```

Error

```
>>> t = Tree(1, [Tree(2)])
>>> t.label
```

1

```
>>> t.label *= 4
>>> t
```

Tree(4, [Tree(2)])

```
>>> t.branches[0]
```

Tree(2)

```
>>> t.branches[0].label
```

2

```
>>> t.label = t.branches[0].label
>>> t
```

Tree(2, [Tree(2)])

```
>>> t.branches.append(Tree(4, [Tree(8)]))
>>> len(t.branches)
```

2

```
>>> t.branches[0]
```

Tree(2)

```
>>> t.branches[1]
```

Tree(4, [Tree(8)])

Working with the Tree class

Q2: Widest Level

Write a function that takes a `Tree` object and returns the elements at the depth with the most elements. In the case of a tie between two depths with the same number of elements, pick the lower depth.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    labels = []
    curr_level = [t]
    while curr_level:
        labels.append([t.label for t in curr_level])
        curr_level = sum([t.branches for t in curr_level], [])
    return max(labels, key=len)
```

Q3: Long Paths

Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a list of node labels that starts with the root and ends at a leaf. Each subsequent element must be from a label of a branch of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are ordered in the output list from left to right in the tree. See the doctests for some examples.

```
def long_paths(t, n):
    """Return a list of all paths in t with length at least n.

    >>> long_paths(Tree(1), 0)
    [[1]]
    >>> long_paths(Tree(1), 1)
    []
    >>> t = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(5), Tree(6, [Tree(7, [Tree(8)])])
    ])
    >>> print(t)
    1
      2
        3
        4
      5
      6
        7
          8
    >>> for path in long_paths(t, 2):
    ...     print(path)
    ...
    [1, 2, 3]
    [1, 2, 4]
    [1, 6, 7, 8]
    >>> long_paths(t, 3)
    [[1, 6, 7, 8]]
    """
    if n <= 0 and t.is_leaf():
      return [[t.label]]
    paths = []
    for b in t.branches:
      for path in long_paths(b, n - 1):
          paths.append([t.label] + path)
    return paths
```

Mutating Trees

Q4: Level Mutation

As a reminder, the depth of a node is how far away the node is from the root. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0.

Given a tree `t` and a list of one-argument functions `funcs`, write a function that will mutate the labels of `t` using the function from `funcs` at the corresponding depth. For example, the label at the root node (with a depth of 0) will be mutated using the function at index 0, or `funcs[0]`. Assume all of the functions in `funcs` will be able to take in a label value and return a valid label value.

If `t` is a leaf and there are more than 1 functions in `funcs`, all of the remaining functions should be applied in order to the label of `t`. (See the doctests for an example.) If `funcs` is empty, the tree should remain unmodified.

```
def level_mutation(t, funcs):
    """Mutates t using the functions in the list funcs.

    >>> t = Tree(1, [Tree(2, [Tree(3)])])
    >>> funcs = [lambda x: x + 1, lambda y: y * 5, lambda z: z ** 2]
    >>> level_mutation(t, funcs)
    >>> t                                # funcs[0] was applied to the label 1, funcs[1]
    to the label 2, etc.
    Tree(2, [Tree(10, [Tree(9)])])
    >>> t2 = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
    >>> level_mutation(t2, funcs)
    >>> t2                               # (2 * 5) ** 2 = 100
    Tree(2, [Tree(100), Tree(15, [Tree(16)])])
    >>> t3 = Tree(1, [Tree(2)])
    >>> level_mutation(t3, funcs)
    >>> t3
    Tree(2, [Tree(100)])
    """
    if not funcs:
        return
    t.label = funcs[0](t.label)
    remaining = funcs[1:]
    if t.is_leaf() and remaining:
        for f in remaining:
            t.label = f(t.label)
    for b in t.branches:
        level_mutation(b, remaining)
```

Q5: Delete

Implement `delete`, which takes a Tree `t` and removes all non-root nodes labeled `x`. The parent of each remaining node is its nearest ancestor that was not removed. In other words, when a non-leaf node is deleted, the deleted node's children become children of its parent. The root node is never removed, even if its label is `x`.

```
def delete(t, x):
    """Remove all nodes labeled x below the root within Tree t. When a non-leaf
    node is deleted, the deleted node's children become children of its parent.

    The root node will never be removed.

    >>> t = Tree(3, [Tree(2, [Tree(2), Tree(2)]), Tree(2), Tree(2, [Tree(2, [Tree(2),
    Tree(2)])])])
    >>> delete(t, 2)
    >>> t
    Tree(3)
    >>> t = Tree(1, [Tree(2, [Tree(4, [Tree(2)]), Tree(5)]), Tree(3, [Tree(6), Tree(2)]),
     Tree(4)])
    >>> delete(t, 2)
    >>> t
    Tree(1, [Tree(4), Tree(5), Tree(3, [Tree(6)]), Tree(4)])
    >>> t = Tree(1, [Tree(2, [Tree(4), Tree(5)]), Tree(3, [Tree(6), Tree(2)]), Tree(2, [
    Tree(6),  Tree(2), Tree(7), Tree(8)]), Tree(4)])
    >>> delete(t, 2)
    >>> t
    Tree(1, [Tree(4), Tree(5), Tree(3, [Tree(6)]), Tree(6), Tree(7), Tree(8), Tree(4)])
    """
    new_branches = []
    for b in t.branches:
        delete(b, x)
        if b.label == x:
            new_branches.extend(b.branches)
        else:
            new_branches.append(b)
    t.branches = new_branches
```