CS 61A
Fall 2017

# Trees and Sequences

Discussion 3: September 20, 2017

## 1 Sequences and Lists

A *sequence* is an ordered collection of values. It has two fundamental properties: length and element selection. In this discussion, we'll explore one of Python's data types, the *list*, which implements this abstraction.

In Python, we can have lists of whatever values we want, be it numbers, strings, functions, or even other lists! Furthermore, the types of the list's contents need not be the same. In other words, the list need not be homogenous.

Lists can be created using square braces. Their elements can be accessed (or *indexed*) with square braces. Lists are zero-indexed: to access the first element, we must index at 0; to access the *i*th element, we must index at $i - 1$.

zero index => max index = len(list) -1

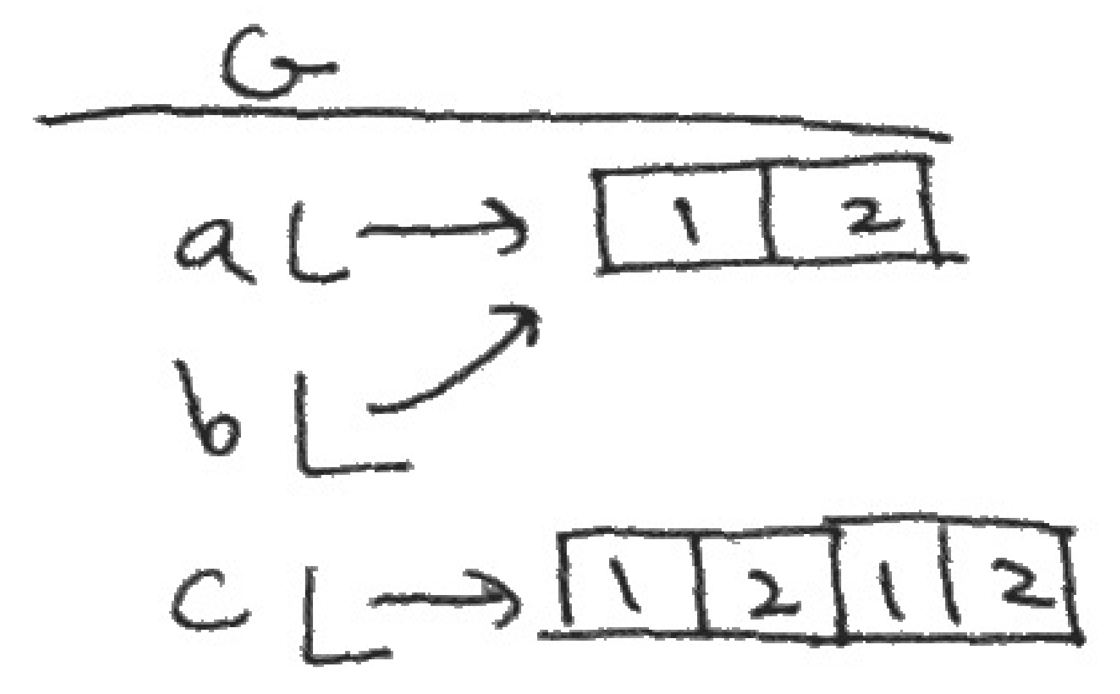
☆ We can also index with negative numbers. These begin indexing at the end of the list, so the index $-1$ is equivalent to the index len(list) - 1 and index $-2$ is the same as len(list) - 2.

Let's try out some indexing:

```
>>> fantasy_team = ['aaron rodgers', 'desean jackson']
>>> print(fantasy_team)
['aaron rodgers', 'desean jackson']
>>> fantasy_team[0]
'aaron rodgers'
>>> fantasy_team[len(fantasy_team) - 1]
'desean jackson'
>>> fantasy_team[-1]
'desean jackson'
```

If we have two lists, we can use the + operator to create a new list with the values of the original two lists, concatenated together.

ex: a = [1,2]
b = a ← not a new list!
c = a + a



```
>>> fish_names = ['Dory', 'Flounder']
>>> rabbit_names = ['Bugs Bunny', 'Officer Hopps']
>>> animal_names = fish_names + rabbit_names
>>> animal_names
['Dory', 'Flounder', 'Bugs Bunny', 'Officer Hopps']
```

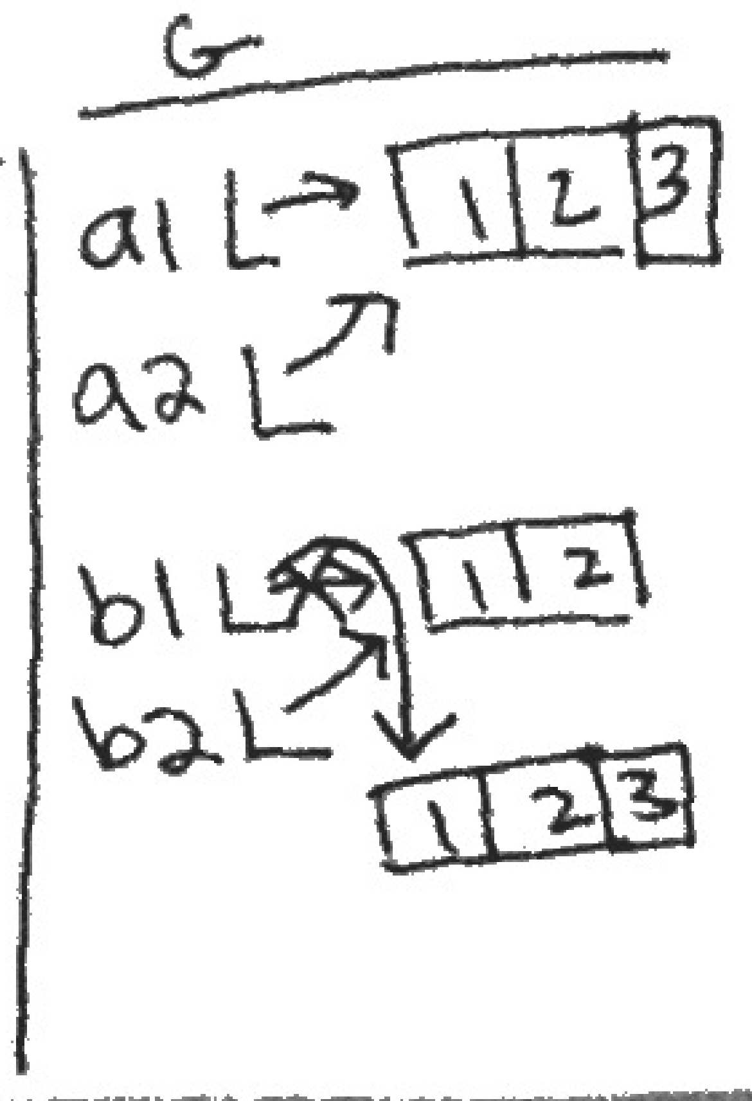
Sequences also have a notion of length, the number of items stored in the sequence. In Python, we can check how long a sequence is with the len built-in function.

We can also check if an item exists within a list with the in statement.

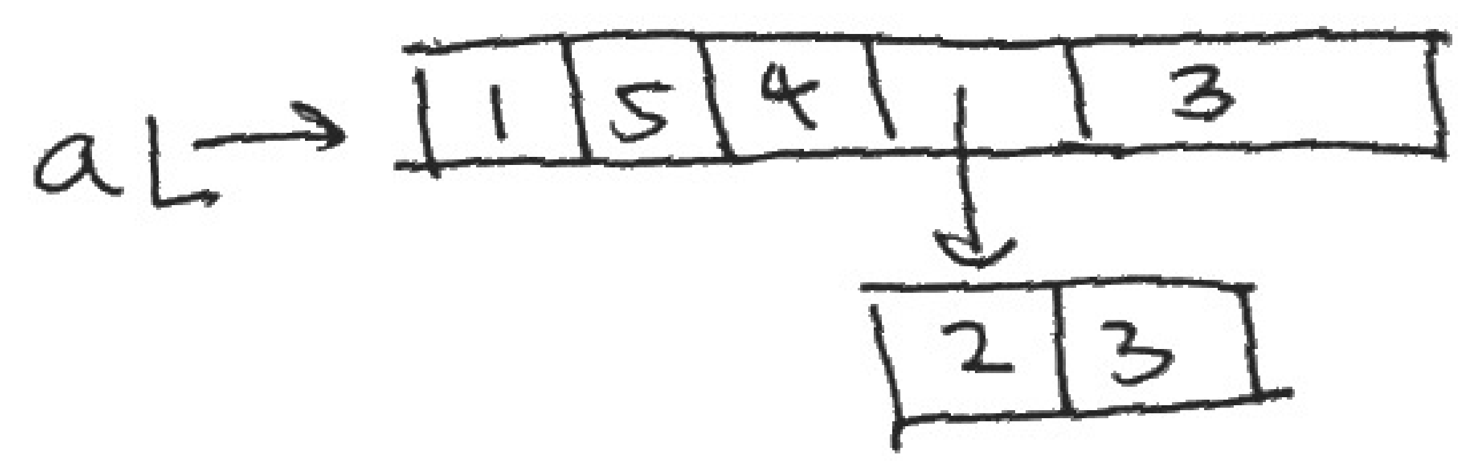
```
>>> poke_team = ['Meowth', 'Mewtwo']
>>> len(poke_team)
2
```

☆ '+=' vs a +

a1 = a2 = [1,2]
b1 = b2 = [1,2]
a1 += [3]
b1 = b1 + [3]



```
>>> 'Meowth' in poke_team
True
>>> 'Pikachu' in poke_team
False
```



## Questions

1.1 What would Python print?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
```
↑ index: -1

1 3

```
>>> len(a)
```

5 * [2,3] is one element of a

```
>>> 2 in a
```

False * the value 2 is not in a; another list containing 2 ([2,3]) is.

```
>>> 4 in a
```

True

```
>>> a[3][0]
```

2 * a[3] = [2,3]
↑
0 index

tested after!

## ☆ Slicing

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index.

• [<start>: <end>]
↑ inclusive ↑ exclusive
* in math notation: [<start>, <end>)

We can also specify a step size, which tells Python how to collect values for us. For example, if we set step size to 2, the returned list will include every **other** value, from the starting index until the ending index. A negative step size indicates that we are stepping backwards through a list when collecting values.

• [<start>: <end>: <step>]

You can also choose not to specify any/all of the slice arguments. Python will perform some default behaviour if this is the case:

- If the step size is left out, the default step size is 1.
- If the start index is left out, the default start index is the beginning of the list.
- If the end index is left out, the default end index is the end of the list.
- ☆ If the step size is negative, the default start index becomes the end of the list, and the default end index becomes the beginning the of the list.

} list[:] ⇒ creates copy of the entire list

list[::-1] will reverse the entire list

Thus, lst[:] creates a list that is identical to lst (a copy of lst). lst[::-1] creates a list that has the same elements of lst, but reversed. Those rules still apply if more than just the step size is specified e.g. lst[3::-1].

```
>>> directors = ['jenkins', 'spielberg', 'bigelow', 'kubrick']
>>> directors[:2]
['jenkins', 'spielberg']
>>> directors[1:3]
['spielberg', 'bigelow']
>>> directors[1:]
['spielberg', 'bigelow', 'kubrick']
>>> directors[0:4:2]
['jenkins', 'bigelow']
>>> directors[::-1]
['kubrick', 'bigelow', 'spielberg', 'jenkins']
```

☆ "out of bounds" slicing doesn't error → it just creates a empty list

```
ex: >>> a = [1, 2]
    >>> b = [2:]
    >>> b
    []
```

## Questions

1.2 What would Python print?

```
>>> a = [3, 1, 4, 2, 5, 3]
>>> a[1::2]
[1, 2, 3]
```
* start from index 1, stepping by 2

```
>>> a[:]
[3, 1, 4, 2, 5, 3]
```
* get entire list

```
>>> a[4:2]
[]
```
* <start> > <end> so it gets nothing; ⇒ empty list

index: 0 1 2 3 4 5

* [3, 1, 4, 2, 5, 3]

$\uparrow$ $\uparrow$

i: -2 index: -1

```
>>> a[1:-2]
```

[1, 4, 2]

$\therefore$ a[1: -2] $\equiv$ a[1: 4]

```
>>> a[::-1]
```

[3, 5, 2, 4, 1, 3] * reverses the list

## 2 List Comprehensions

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

newList = []

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

$\Rightarrow$ for x in [1,2,3,4,5]:
  if x % 2 == 1:
    newList += [redacted] [x*x-3]

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (1, 3, and 5). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

*Note*: The if clause in a list comprehension is optional.

syntax changes if you want [redacted] if/else:
[ <expr> if <cond> else <expr2> for x in list]

### Questions

2.1 What would Python print?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
```

[3, 5] * if i is even, add to list [redacted] the element i+1

```
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]
```

[20, 6, 6] * if i > 2, add to list the element (i*i)-i

☆ 
```
>>> [[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```

nested list comprehension!

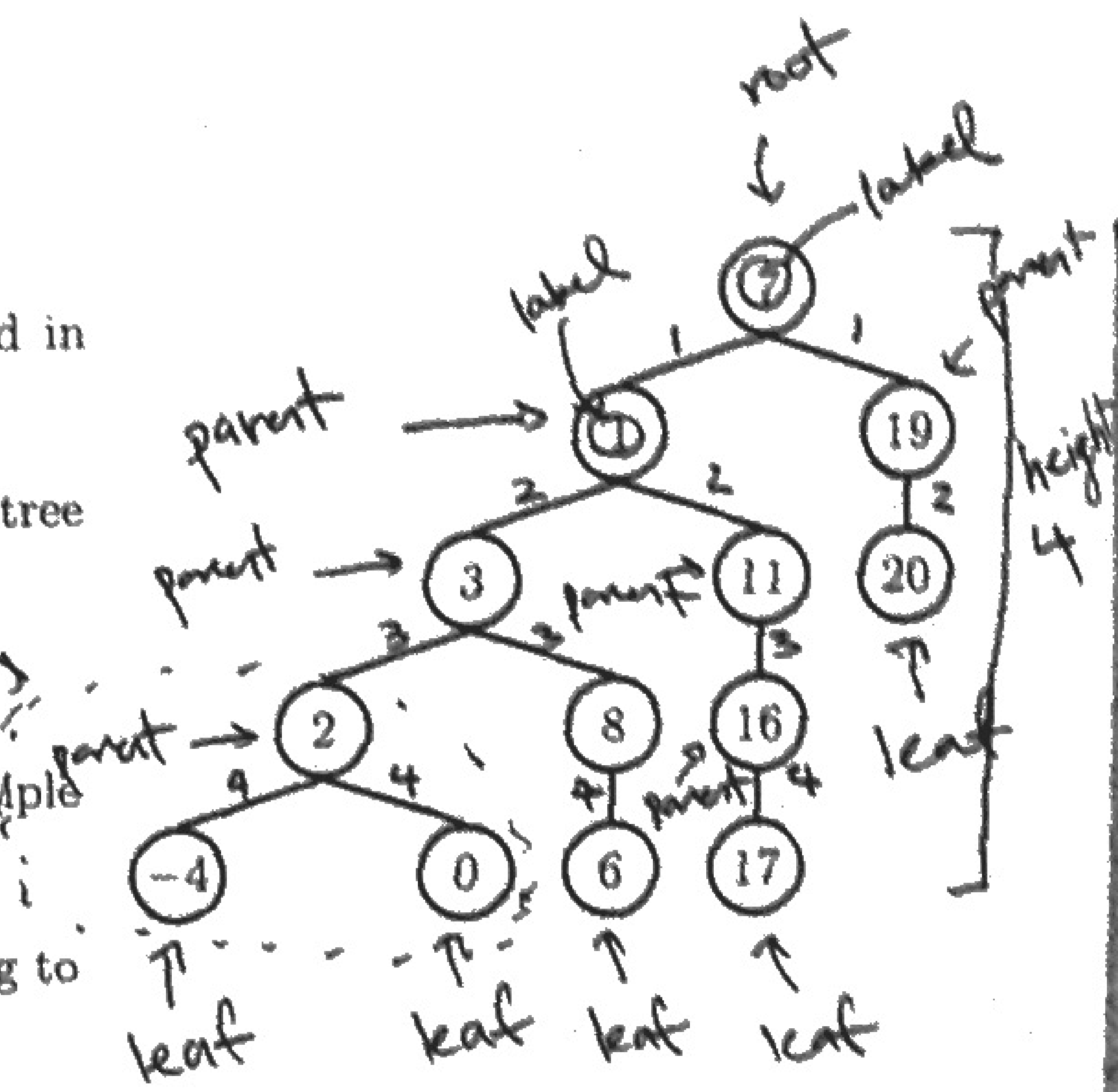
[[2,4], [4,6], [6,8], [8,10]]

list = []
$\Rightarrow$ for x in [1, 2, 3, 4]:
  list1 = []
  for y in [x, x+1]:
    list1 += [y*2]
  list += [list1]

* I tend to avoid nested list comprehensions for the sake of readability, but it's a personal choice!

## 3 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 simple 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.

subtree where $\boxed{2}$ is the root

Some terminology regarding trees:

- **Parent node**: A node that has branches. Parent nodes can have multiple branches.
- **Branch node**: A node that has a parent. A branch 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**: Notice that each branch of a parent is itself the root of a smaller tree. In our example, the node containing 1 is the root of another tree. This is why trees are *recursive* data structures: trees have branches, which are trees themselves.
- **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 simply an implementation detail.

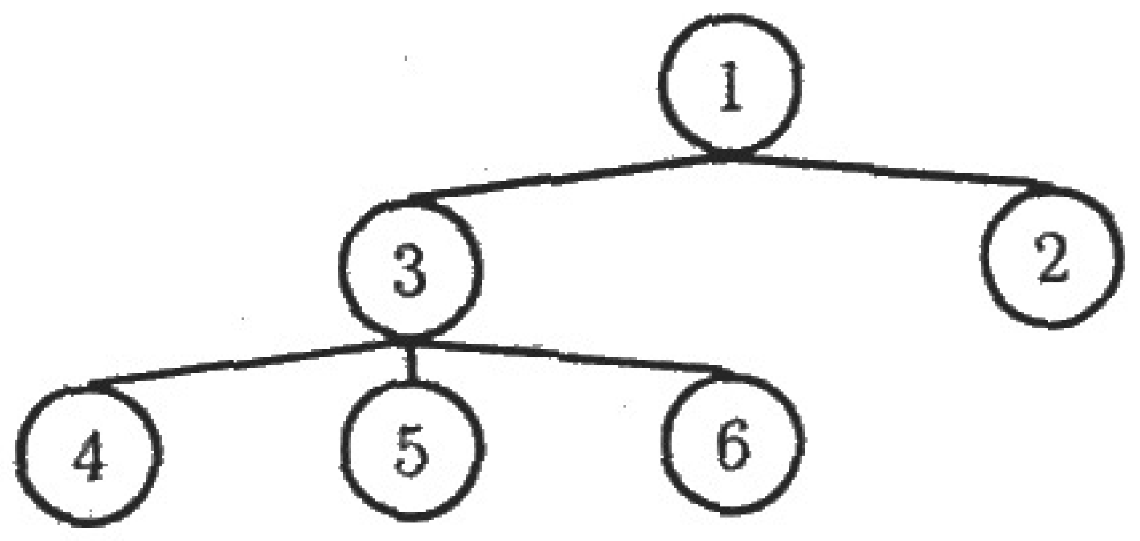
- The arguments to the constructor `tree` are the value for the root node and a list of branches.
- The selectors for these are `label` and `branches`.

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`.

It's simple to construct a tree. Let's try to create the tree below.

```
# Constructor
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

# Selectors
def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

# For convenience
def is_leaf(tree):
    return not branches(tree)
```



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

☆branches are <u>lists</u> of <u>trees</u>!

→ a node is a <u>leaf</u> if it has no branches

↳ a leaf node is still a tree

☆ abstraction:
call label(tree) to get the value,
call branches(tree) to get branches

## Questions

3.1 Define a function tree_max(t) that returns the largest number in a tree.

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

* recursively pull out all the labels in the tree, then use the max(...) function to retrieve the largest

```
return max([label(t)] + [tree_max(branch) for branch in branches(t)])
```

3.2 Define a function height(t) 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"""
    if is_leaf(t):
        return 0

    return 1 + max([height(branch) for branch in branches(t)])
```

← we want the height of the longest branch; leap of faith ⇒ assume height(t) works: call that on each branch, take the tallest w/ max(...), then add 1 to account for the top node

3.3 Define a function square_tree(t) that squares every value in the tree t. It should return a new tree. You can assume that every item is a number.

→ * we need to call the tree(...) constructor!

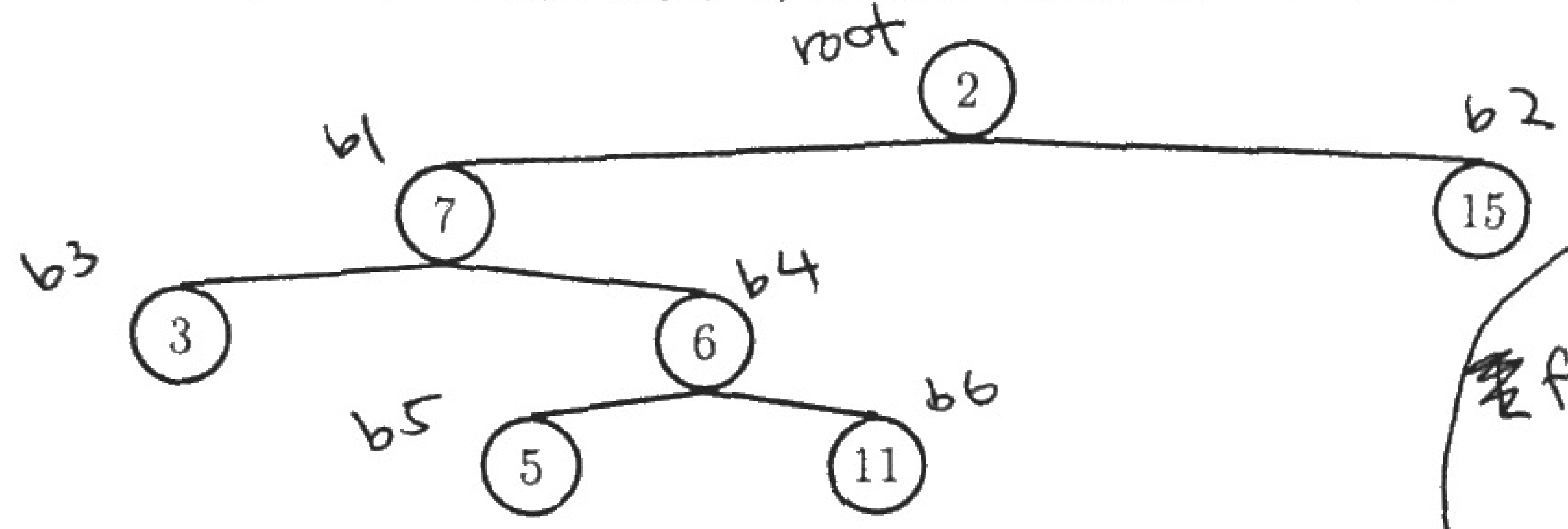
```
def square_tree(t):
    """Return a tree with the square of every element in t"""
    ~~if is_leaf(t)~~:
        ~~return tree(label(t)**2)~~

    return tree(label(t)**2, [square_tree(branch) for branch in
                              branches(t)]
```

} don't actually need this b/c ~~the~~ list comprehensim will be a empty list if no branches

3.4 Define the procedure find_path(tree, x) that, given a tree tree and a value x, returns a list containing the nodes along the path required to get from the root of tree to a node x. If x is not present in tree, return None. Assume that the entries of tree are unique.

[2, 7, 6, 5]

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



find_path(~~root~~, 5) [7, 6, 5]
↳ [find_path(~~b1~~, 5),
find_path(b2, 5)]

find_path(b1, 5)
↳ [find_path(~~b3~~, 5) None
find_path(~~b4~~, 5)] [6, 5]

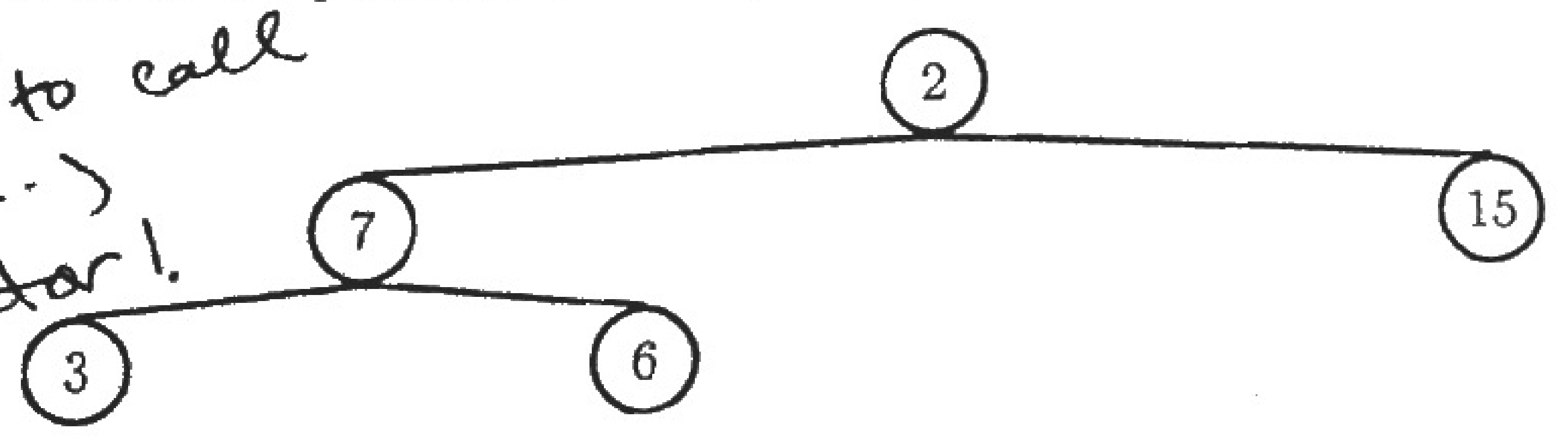
find_path(b2, 5)
↳ ~~[find_path(15~~ None

[5]
[find_path(~~b5~~, 5),
find_path(~~b6~~, 5)]
None

```
def find_path(tree, x):
    """
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """
    if label(tree)==x:
        return [label(tree)]
    node, branches = label(tree), branches(tree)
    for path in [find_path(t,x) for t in trees]:
        if path:
            return [node] + path
```

3.5 Implement a prune function which takes in a tree t and a depth k, and should return a new tree that is a copy of only the first k levels of t. For example, if t is the tree shown in the previous question, then prune(t, 2) should return the tree

need to call tree(...) constructor!



k decrements because we already "used up" one level to include the current node

```
def prune(t, k):
    if k==0:
        return tree(label(t), [])
    else:
        return tree(label(t), [prune(branch, k-1) for branch
                 in branches(t)])
```