Trees and Sequences 3

Computer Science 61A

September 24, 2015

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 ith element, we must index at $i - 1$.

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 = ['frank gore', 'calvin johnson']
>>> print(fantasy_team)
['frank gore', 'calvin johnson']
>>> fantasy_team[0]
'frank gore'
>>> fantasy_team[len(fantasy_team) - 1]
'calvin johnson'
>>> fantasy_team[-1]
'calvin johnson'
```

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.

```
>>> mouse_names = ['Picasso', 'Fred']
>>> dog_names = ['Spot', 'Rusty']
>>> pet_names = mouse_names + dog_names
>>> pet_names
['Picasso', 'Fred', 'Spot', 'Rusty']
```

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_list = ['Meowth', 'Mewtwo']
>>> len(poke_list)
2
>>> 'Meowth' in poke_list
True
>>> 'Pheobe' in poke_list
False
```

1. What would Python print?

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

>>> len(a)

>>> 2 in a

>>> 4 in a

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

1.1 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.

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.

If the step size is left out, the default step size is 1. If either the start or end indices are left out, the slice starts at the beginning and ends at the end of the list. When the step size is negative, the slice starts at the end and ends at the beginning of the 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]`.

```
>>> pet_list = ['Mochi', 'Picasso', 'Rusty', 'Pheobe']
>>> pet_list[:2]
['Mochi', 'Picasso']
>>> pet_list[1:3]
['Picasso', 'Rusty']
>>> pet_list[1:]
['Picasso', 'Rusty', 'Pheobe']
>>> pet_list[0:4:2]
['Mochi', 'Rusty']
>>> pet_list[::-1]
['Pheobe', 'Rusty', 'Picasso', 'Mochi']
```

1. What would Python print?

```
>>> a = [3, 1, 4, 2, 5, 3]
>>> a[1::2]

>>> a[:]

>>> a[4:2]

>>> a[1:-2]

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

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>]
```

Let's break down an example:

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

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.

1. What would Python print?

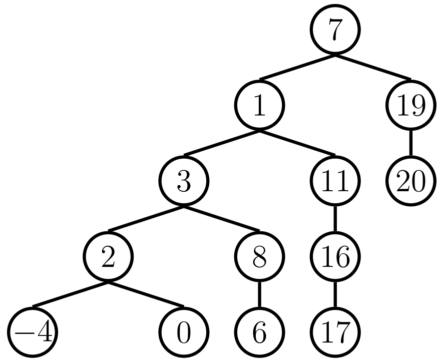
```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
```

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

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

3 Trees

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

Some terminology regarding trees:

- **Parent node**: A node that has children. Parent nodes can have multiple children.
- **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.
- **Leaf**: A node that has no children. In our example, the nodes that contain -4, 0, 6, 17, and 20 are leaves.
- **Subtree**: Notice that each child 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 are made up of subtrees, 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 children each node has; others vary in the structure of the tree.

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

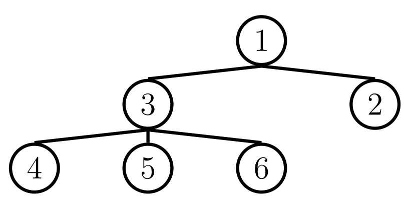
- The arguments to the constructor, `tree`, as a value for the root and a list of branches.
- The selectors are `root` and `branches`.

```
# Constructor
def tree(value, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [value] + list(branches)

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

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

We have also provided two convenience functions, `is_leaf` and `is_tree`:

```
def is_leaf(tree):
    return not branches(tree)

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```

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




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

1. Define a function `square_tree(t)` that squares every item in the tree `t`. 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"""
```

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"""
```

3. Define a function `tree_size(t)` that returns the number of nodes in a tree.

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

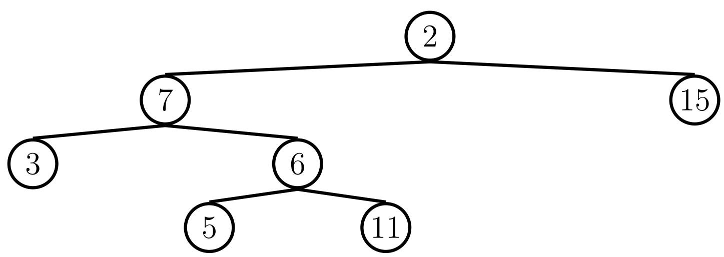
4. Define a function `tree_max(t)` that returns the largest number in a tree.

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

3.1 Extra Questions!

1. Define the procedure `find_path(tree, x)` that, given a rooted 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 elements in `tree` are unique.

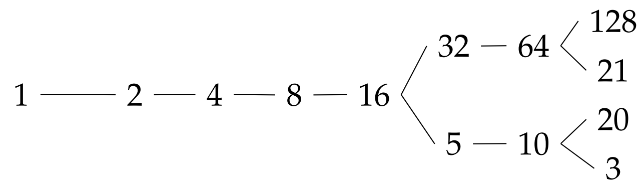
   For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`




```
def find_path(tree, x):
    """
    >>> t = <See Above>
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)
    """
```

2. We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number n, continuing to $n/2$ if n is even or $3n + 1$ if n is odd, ending with 1. Write a function `hailstone_tree(n, h)` which generates a tree of height `h`, containing hailstone numbers that will reach `n`.

   Hint: A node of a hailstone tree will always have at least one, and at most two branches (which are also hailstone trees). Under what conditions do you add the second branch?




```python
def hailstone_tree(n, h):
    """Generates a tree of hailstone numbers that will
       reach N, with height H.
    >>> hailstone_tree(1, 0)
    [1]
    >>> hailstone_tree(1, 4)
    [1, [2, [4, [8, [16]]]]]
    >>> hailstone_tree(8, 3)
    [8, [16, [32, [64]], [5, [10]]]]
    """
```