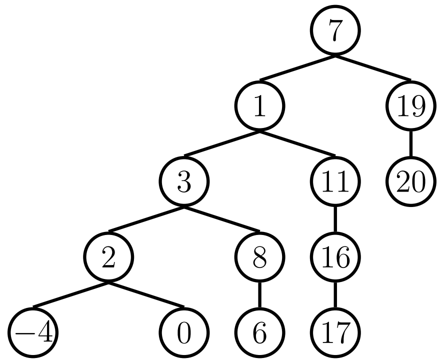
TREES AND MUTATION 5

COMPUTER SCIENCE 61A

February 25, 2016

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

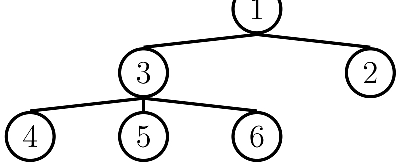
- The arguments to the constructor, `tree`, as a value for the label and a list of children.
- The selectors are `label` and `children`.

```
# Constructor
def tree(label, children=[]):
    return [label] + list(children)

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

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

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

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

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)])
```

1.1 Questions

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.

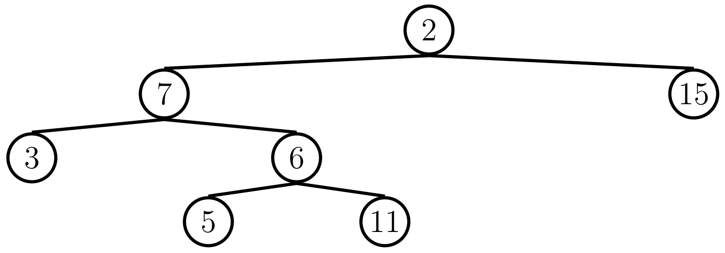
```
def square_tree(t):
    """Return a tree with the square of every element in t"""
```

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

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

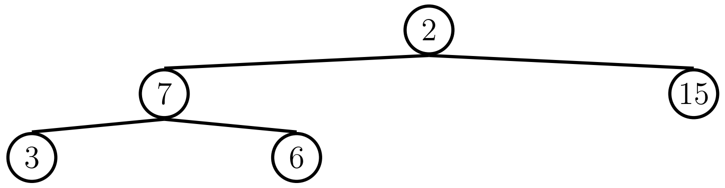
3. Define the procedure `find_path(tree, x)` that, given a tree `tree` and a value `x`, returns a list containing the nodes along the path required to get from the root of `tree` to a node `x`. If `x` is not present in `tree`, return `None`. Assume that the labels of `tree` are unique.

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



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

4. 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



```
def prune(t, k):
```

2 Mutable Lists

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

```
>>> pizza1 = ['cheese', 'mushrooms']
```

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

```
>>> pizza2 = pizza1 + ['onions'] # creates a new python list
>>> pizza2
['cheese', mushrooms', 'onions']
>>> pizza1 # the original list is unmodified
['cheese', 'mushrooms']
```

But this is silly, considering that all Domino's had to do was add onions on top of `pizza1` instead of making an entirely new `pizza2`.

Python actually allows you to *mutate* some objects, includings lists and dictionaries. Mutability means that the object's contents can be changed. So instead of building a new `pizza2`, we can use `pizza1.append('onions')` to mutate `pizza1`.

```
>>> pizza1.append('onions')
>>> pizza1
['cheese', 'mushrooms', 'onions']
```

Although lists and dictionaries are mutable, many other objects, such as numeric types, tuples, and strings, are *immutable*, meaning they cannot be changed once they are created. We can use the familiar indexing operator to mutate a single element in a list. For instance `lst[4]='hello'` would change the fifth element in `lst` to be the string `'hello'`. In addition to the indexing operator, lists have many mutating methods. List *methods* are functions that are bound to a specific list. Some useful list methods are listed here:

1. `append(el)` adds `el` to the end of the list
2. `insert(i, el)` insert `el` at index `i` (does not replace element but adds a new one)
3. `remove(el)` removes the first occurrence of `el` in list, otherwise errors
4. `pop(i)` removes and returns the element at index `i`

List methods are called via *dot notation*, as in:

```
>>> colts = ['andrew luck', 'reggie wayne']
>>> colts.append('trent richardson')
>>> colts.pop(1)
'reggie wayne'
>>> colts
['andrew luck', 'trent richardson']
```

2.1 Questions

1. Consider the following definitions and assignments and determine what Python would output for each of the calls below *if they were evaluated in order*. It may be helpful to draw the box and pointers diagrams to the right in order to keep track of the state.

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [1, 2, 3]
>>> lst1 == lst2 #compares each value


>>> lst1 is lst2 #compares references


>>> lst2 = lst1
>>> lst2 is lst1


>>> lst1.append(4)
>>> lst1


>>> lst2


>>> lst2[1] = 42
>>> lst2


>>> lst1 = lst1 + [5]
>>> lst1 == lst2


>>> lst1


>>> lst2


>>> lst2 is lst1
```

2. Write a function that removes all instances of an element from a list.

```
def remove_all(el, lst):
    """
    >>> x = [3, 1, 2, 1, 5, 1, 1, 7]
    >>> remove_all(1, x)
    >>> x
    [3, 2, 5, 7]
    """
```

3. Write a function that takes in two values `x` and `el`, and a list, and adds as many `el`'s to the end of the list as there are `x`'s.

```
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]
    """
```

3 Dictionaries

Dictionaries are data structures which map keys to values. Dictionaries in Python are unordered, unlike real-world dictionaries — in other words, key-value pairs are not arranged in the dictionary in any particular order. Let's look at an example:

```
>>> pokemon = {'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['pikachu']
25
>>> pokemon['jolteon'] = 135
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148, 'mew': 151}
>>> pokemon['ditto'] = 25
>>> pokemon
{'jolteon': 135, 'pikachu': 25, 'dragonair': 148,
'ditto': 25, 'mew': 151}
```

The *keys* of a dictionary can be any *immutable* value, such as numbers, strings, and tuples. Dictionaries themselves are mutable; we can add, remove, and change entries after creation. There is only one value per key, however — if we assign a new value to the same key, it overrides any previous value which might have existed.

To access the value of `dictionary` at `key`, use the syntax `dictionary[key]`.

Element selection and reassignment work similarly to sequences, except the square brackets contain the key, not an index.

- To add `val` corresponding to `key` *or* to replace the current value of `key` with `val`:
  ```
  dictionary[key] = val
  ```
- To iterate over a dictionary's keys:
  ```
  for key in dictionary: #OR for key in dictionary.keys()
      do_stuff()
  ```
- To iterate over a dictionary's values:
  ```
  for value in dictionary.values():
      do_stuff()
  ```
- To iterate over a dictionary's keys and values:
  ```
  for key, value in dictionary.items():
      do_stuff()
  ```
- To remove an entry in a dictionary:
  ```
  del dictionary[key]
  ```
- To get the value corresponding to `key` and remove the entry:
  ```
  dictionary.pop(key)
  ```

3.1 Questions

1. What would Python output given the following inputs?

```
>>> 'mewtwo'in pokemon
```

```
>>> len(pokemon)
```

```
>>> pokemon['ditto'] = pokemon['jolteon']
>>> pokemon[('diglett', 'diglett', 'diglett')] = 51
>>> pokemon[25] = 'pikachu'
>>> pokemon
```

```
>>> pokemon['mewtwo'] = pokemon['mew'] * 2
>>> pokemon
```

```
>>> pokemon[['firetype', 'flying']] = 146
```

Note that the last example demonstrates that dictionaries cannot use other mutable data structures as keys. However, dictionaries can be arbitrarily deep, meaning the *values* of a dictionary can be themselves dictionaries.

2. Given a (non-nested) dictionary d, write a function which deletes all occurrences of x as a value. You cannot delete items in a dictionary as you are iterating through it.

```
def remove_all(d, x):
    """
    >>> d = {1:2, 2:3, 3:2, 4:3}
    >>> remove_all(d, 2)
    >>> d
    {2: 3, 4: 3}
    """
```

3. Write a function `group_by` that takes in a sequence `s` and a function `fn` and returns a dictionary. The function `fn` will take in an element of the sequence and return some key. The returned dictionary groups all of the elements in `s` by the key returned from `fn`.

```
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]}
    """
```

4. Given an arbitrarily deep dictionary `d`, replace all occurences of `x` *as a value (not a key)* with `y`. Hint: You will need to combine iteration and recursion.

```
def replace_all_deep(d, x, y):
    """
    >>> d = {1: {2: 3, 3: 4}, 2: {4: 4, 5: 3}}
    >>> replace_all_deep(d, 3, 1)
    >>> d
    {1: {2: 1, 3: 4}, 2: {4: 4, 5: 1}}
    """
```