TREES, MUTABLE STRUCTURES, AND GROWTH

COMPUTER SCIENCE MENTORS 61A

September 26 to September 30, 2016

1 Trees

Things to remember

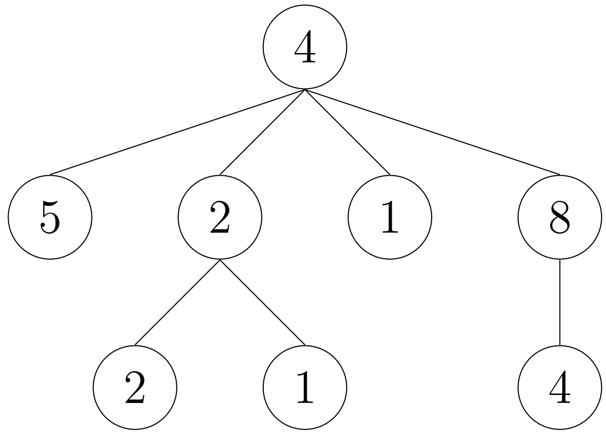
```
def tree(root, branches=[]):
    return [root] + list(branches)

def root(t):
    return t[0]

def branches(t): # Always returns a list of trees
    return t[1:]
```

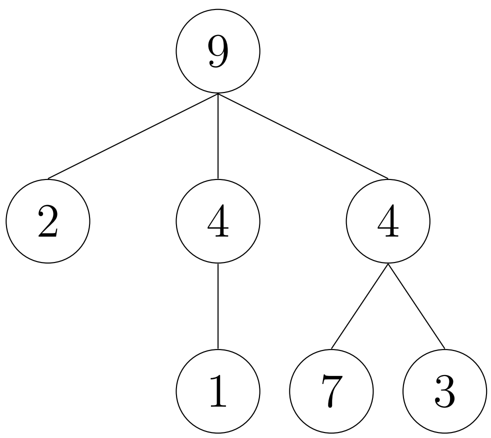
1. Draw the tree that is created by the following statement:

```
tree(4,
     [tree(5, []),
      tree(2,
          [tree(2, []),
           tree(1, [])]),
      tree(1, []),
      tree(8,
          [tree(4, [])])])
```



Solution:

2. Construct the following tree and save it to the variable `t`.



Solution:

```
t = tree(9, [tree(2, []),
             tree(4, [tree(1, [])]),
             tree(4, [tree(7, []),
                      tree(3, [])])])
```

3. What would this output?

```
>>> root(t)
```

Solution: 9

```
>>> branches(t)[2]
```

Solution:
```
tree(4, [tree(7, []), tree(3, [])])
```

```
>>> branches(branches(t)[2])[0]
```

Solution:
```
tree(7, [])
```

4. Write the Python expression to get the integer 2 from t.

Solution:
```
root(branches(t)[0])
```

5. Write the function `sum_of_nodes` which takes in a tree and outputs the sum of all the elements in the tree.

```
def sum_of_nodes(t):
    """
    >>> t = Tree(...) # Tree from question 2.
    >>> sum_of_nodes(t) # 9 + 2 + 4 + 4 + 1 + 7 + 3 = 30
    30
    """
```

Solution:
```
    total = root(t)
    for branch in branches(t):
        total += sum_of_nodes(branch)
    return total

    Alternative solution:
    return root(t) +\
           sum([sum_of_nodes(b) for b in branches(t)])
```

2 Mutation

1. **What Would Python Display?**

```
>>> a = [1, 2]
>>> a.append([3, 4])
>>> a
```

Solution:
```
[1, 2, [3, 4]]
```

```
>>> b = list(a)
>>> a[0] = 5
>>> a[2][0] = 6
>>> b
```

Solution:
```
[1, 2, [6, 4]]
```

```
>>> a.extend([7])
>>> a += [8]
>>> a += 9
```

Solution:
```
TypeError: 'int' object is not iterable
```

```
>>> a
```

Solution:
```
[5, 2, [6, 4], 7, 8]
```

Challenge problem:

```
>>> b[2][1] = a[2:]
>>> a[2][1][0][0]
```

Solution:
```
6
```

2. Given a list of lists `lst_of_lsts` and some element `elem`, append elem to every list in `lst_of_lsts`.

```
def append_to_all(lst_of_lsts, elem):
    """
    >>> l = [[1, 0, 5], [2, 6, 4], [8, 3]]
    >>> append_to_all(l, 7)
    >>> l
    [[1, 0, 5, 7], [2, 6, 4, 7], [8, 3, 7]]
    """
```

Solution:

```
    for lst in lst_of_lsts:
        lst.append(elem)
```

3. Given some list `lst`, possibly a deep list, mutate lst to have the accumulated sum of all elements so far in the list. If there is a nested list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Return the total sum of lst.

   Note: You may find it useful to use the `isinstance` function, which returns True for `isinstance(l, list)` if `l` is a list and False otherwise.

```
def accumulate(lst):
    """
    >>> l = [1, 5, 13, 4]
    >>> accumulate(l)
    23
    >>> l
    [1, 6, 19, 23]
    >>> deep_l = [3, 7, [2, 5, 6], 9]
    32
    >>> deep_l
    [3, 10, [2, 7, 13], 32]
    """
```

Solution:

```
    sum_so_far = 0
    for i in range(len(lst)):
        item = lst[i]
        if isinstance(item, list):
            inside = accumulate(item)
            sum_so_far += inside
        else:
            sum_so_far += item
            lst[i] = sum_so_far
    return sum_so_far

    Alternate solution:
    if isinstance(lst[0], list):
        lst[0] = accumulate(lst[0])
    for i in range(1, len(lst)):
        if isinstance(lst[i], list):
            lst[i] = accumulate(lst[i)i]) + lst[i-1]
        else:
            lst[i] = lst[i] + lst[i-1]
    return lst[-1]
```

3 Growth

1. In big-O notation, what is the runtime for `foo`?

   (a)
   ```
   def foo(n):
       for i in range(n):
           print('hello')
   ```

   Solution: $O(n)$. This is simple loop that will run n times.

   (b) What's the runtime of `foo` if we change `range(n)`:

   i. To `range(n / 2)`?

   Solution: $O(n)$. The loop runs $n/2$ times, but we ignore constant factors.

   ii. To `range(10)`?

   Solution: $O(1)$. No matter the size of n, we will run the loop the same number of times.

   iii. To `range(10000000)`?

   Solution: $O(1)$. No matter the size of n, we will run the loop the same number of times.

2. **Orders of Growth and Trees:** Assume we are using the non-mutable tree implementation introduced earlier. Consider the following function:

   ```
   def word_finder(t, n, word):
       if root(t) == word:
           n -= 1
           if n == 0:
               return True
       for branch in branches(t):
           if word_finder(branch, n, word):
               return True
       return False
   ```

   (a) What does this function do?

Solution: This function take a Tree `t`, an integer `n`, and a string `word` in as input.

Then, `word_finder` returns True if any paths from the root towards the leaves have at least `n` occurrences of the word and False otherwise.

(b) If a tree has n total nodes, what is the total runtime for all searches in big-O notation?

Solution: $O(n)$. At worst, we must visit every node of the tree.

3. What is the order of growth in time for the following functions? Use big-O notation.

(a)
```
def strange_add(n):
    if n == 0:
        return 1
    else:
        return strange_add(n - 1) + strange_add(n - 1)
```

Solution: $O(2^n)$. To see this, try drawing out the call tree. Each level will create two new calls to `strange_add`, and there are n levels. Therefore, 2^n calls.

(b)
```
def stranger_add(n):
    if n < 3:
        return n
    elif n % 3 ==  0:
        return stranger_add(n - 1) + stranger_add(n - 2) +
           stranger_add(n - 3)
    else:
        return n
```

Solution: $O(n)$ is n is a multiple of 3, otherwise $O(1)$.
The case where n is not a multiple of 3 is fairly obvious – we step into the else clause and immediately return.
If n is a multiple of 3, then neither $n-1$ nor $n-2$ are multiples of 3 so those calls will take constant time. Therefore, we just run `stranger_add`, decrementing the argument by 3 each time.

(c)
```
def waffle(n):
    i = 0
    sum = 0
    while i < n:
        for j in range(50 * n):
            sum += 1
        i += 1
    return sum
```

Solution: $O(n^2)$. Ignore the constant term in $50 * n$, and it because just two for loops.

(d)
```
def belgian_waffle(n):
    i = 0
    sum = 0
    while i < n:
        for j in range(n ** 2):
            sum += 1
        i += 1
    return sum
```

Solution: $O(n^3)$. Inner loop runs n^2 times, and the outer loop runs n times. To get the total, multiply those together.

(e)
```
def pancake(n):
    if n == 0 or n == 1:
        return n
    # Flip will always perform three operations and return
      -n.
    return flip(n) + pancake(n - 1) + pancake(n - 2)
```

Solution: $O(2^n)$. Flip will run in constant time. Therefore, this call tree looks very similar to fib! (which is 2^n)

(f)
```
def toast(n):
    i = 0
    j = 0
    stack = 0
    while i < n:
        stack += pancake(n)
        i += 1
    while j < n:
        stack += 1
        j += 1
    return stack
```

Solution: $O(n2^n)$. There are two loops: the first runs n times for 2^n calls each time (due to pancake), for a total of $n2^n$. The second loop runs n times. When calculating orders of growth however, we focus on the dominating term – in this case, $n2^n$.