TREES AND ORDERS OF GROWTH 8

COMPUTER SCIENCE 61A

October 23, 2014

1 Trees in OOP

1.1 Our Implementation

Previously, we have seen trees defined as an abstract data type using lists. This time, we will use OOP syntax to define trees. One advantage of using OOP is that specialized tree types such as binary trees will now be more easily specified via inheritance.

```
class Tree:
    """A tree with entry as its root value."""
    def __init__(self, entry, branches=()):
        self.entry = entry
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

1.2 Questions

1. Define a function `square_tree(t)` that squares every item in `t`. You can assume that every item is a number.

```
def square_tree(t):
    """Mutates a Tree t by squaring all its elements"""
```

2. Define a function `make_even` which takes in a `tree` of integers, and mutates the tree such that all the odd numbers are increased by 1 and all the even numbers remain the same. Then write this function so that it returns a new tree instead.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t # Assuming __repr__ is defined
    Tree(2, [Tree(2, [Tree(4)]), Tree(4), Tree(6)])
    """
```

3. Assuming that every item in `t` is a number, let's define `average(t)`, which returns the average of all the items in `t`.

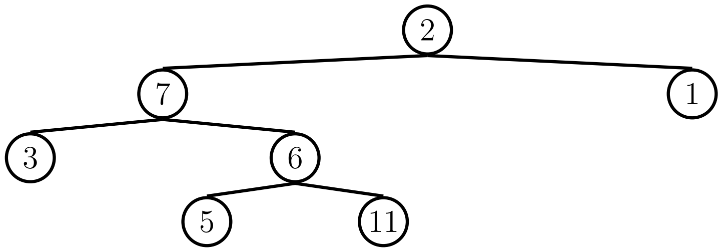
```
def average(t):
```

1.3 Extra Questions

4. Define the procedure `find_path` that, given an Tree `t` and an entry `entry`, returns a list containing the nodes along the path required to get from the root of `t` to `entry`. If `entry` is not present in `t`, return `False`.

a. Assume that the elements in t are unique. Find the path to an element.

For instance, for the following tree, find_path should return:

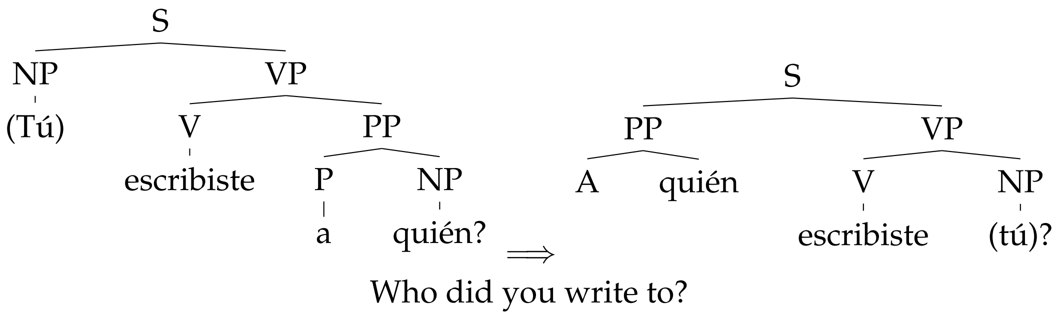


```
>>> find_path(tree_ex, 5)
[2, 7, 6, 5]

def find_path(t, entry):
```

b. Now assume that the elements of the tree might not be unique. How would you change your answer from part a to find the shortest path? Try to implement the function find_shortest, which has the same parameters as find_path.

5. In syntax, a way to model question formation is move the question word in the syntax tree.



a. How would we modify the Tree class so that each node remembers its parent?

b. Now write a **method** `first_to_last` for the `Tree` class that swaps a tree's own first child with the last child of `other`.

```
def first_to_last(self, other):
```

1.4 Binary Trees

Sometimes, it is more practical to assume that there will only be at most two branches per tree. Since a binary tree is just a specialization of a regular tree, we can use inheritance to help us with the implementation.

```
class BinaryTree(Tree):
    empty = Tree(None)
    empty.is_empty = True

    def __init__(self, entry, left=empty, right=empty):
        for branch in (left, right):
            assert isinstance(branch, BinaryTree) or branch.
               is_empty
        Tree.__init__(self, entry, (left, right))
        self.is_empty = False

    @property
    def left(self):
        return self.branches[0]

    @property
    def right(self):
        return self.branches[1]
```

1.5 Conceptual Questions

1. a. What is the purpose of the assert statement in the second line of `__init__`? Why must we write this line explicitly instead of relying on `Tree.__init__`'s assert?

   b. Summarize the process of creating a new BinaryTree. How does `Tree.__init__` contribute?

   c. Why do we use `@property` instead of writing `self.left = self.branches[0]` in `__init__`?

2 Orders of Growth

When we talk about the efficiency of a procedure (at least for now), we are often interested in how much more expensive it is to run the procedure with a larger input. That is, how do the time a process takes and the space it occupies grow as the size of the input grows?

For expressing all of these, we use what is called the big Theta notation. For example, if we say the running time of a procedure `foo` is in $\Theta(n^2)$, we mean that the running time of the process, `R(n)`, will grow proportionally to the square of the size of the input `n` as `n` increases to infinity.

Fortunately, in CS 61A, we're not that concerned with rigorous mathematical proofs. (You'll get more details, including big O and big Omega notation, in CS 61B!) What we want you to develop in CS 61A is the intuition to reason out the orders of growth for certain procedures.

2.1 Kinds of Growth

Here are some common orders of growth, ranked from no growth to fastest growth:

- $\Theta(1)$ — constant time takes the same amount of time regardless of input size
- $\Theta(\log n)$ — logarithmic time
- $\Theta(n)$ — linear time
- $\Theta(n^2)$, $\Theta(n^3)$, etc. — polynomial time
- $\Theta(2^n)$ — exponential time ("intractable"; these are really, really horrible)

2.2 Orders of Growth in Time

"Time" refers to the number of primitive operations completed, such as $+$, $*$, and assignment. The time it takes for these operations is $\Theta(1)$. Consider the following functions:

```
def double(n):
    return n * 2
```

```
def double_list(lst):
    return [double(elem) for elem in lst
    ]
```

- `double` is runs in $\Theta(1)$, or constant time because it does the same number of primitive procedures no matter what `n` is.
- `double_list` runs in $\Theta(n)$, where `n = len(lst)`, because a $\Theta(1)$ procedure (`double`) is repeated n times. As `lst` becomes larger, so does the runtime for `double_list`.

Here are some general guidelines for orders of growth:

- If the function is recursive or iterative, you can subdivide the problem as seen above:
  - Count the number of recursive calls/iterations that will be made, given input n.
  - Count how much time it takes to process the input per recursive call/iteration.

  The answer is usually the product of the above two, but pay attention to control flow!
- If the function calls helper functions that are not constant-time, you need to take the orders of growth of the helper functions into consideration.
- We can ignore constant factors. For example, $\Theta(1000000n) = \Theta(n)$.
- We can also ignore lower-order terms. For example, $\Theta(n^3 + n^2 + 4n + 399) = \Theta(n^3)$. This is because the plot for $n^3 + n^2 + 4n + 399$ is similar to n^3 as n approaches infinity.

Note: in practice, constant factors and extra terms are important. If a program `f` takes $1000000n$ time whereas another, `g`, takes n time, clearly `g` is faster (and better).

2.3 Questions

What is the order of growth in time for the following functions?

1.
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

2.
```
def fib_iter(n):
    prev, cur, i = 0, 1, 1
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return curr
```

3.
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

4.
```
def bar(n):
    if n % 2 == 1:
        return n + 1
    return n

def foo(n):
    if n < 1:
        return 2
    if n % 2 == 0:
        return foo(n - 1) + foo(n - 2)
    else:
        return 1 + foo(n - 2)
```

What is the order of growth of `foo(bar(n))`?

5.
```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

3 Extra Questions

1. Write a function that creates a balanced binary search tree from a given sorted list. Its runtime should be in $\Theta(n)$, where n is the number of nodes in the tree. Binary search trees have an additional invariant (property) that each element in the right branch must be larger than the entry and each element in the left branch must be smaller than the entry.

```
def list_to_bst(lst):
```

2. Give the running times of the functions `g` and `h` in terms of `n`. Then, for a bigger challenge, give the runtime of `f` in terms of `x` and `y`.

```
def f(x, y):
    if x == 0 or y == 0:
        return 1
    if x < y:
        return f(x, y-1)
    if x > y:
        return f(x-1, y)
    else:
        return f(x-1, y) + f(x, y-1)

def g(n):
    return f(n, n)

def h(n):
    return f(n, 1)
```