# TREES AND ORDERS OF GROWTH 7

COMPUTER SCIENCE 61A

March 12, 2015

## 1 Trees in OOP

### 1.1 Our Implementation

Previously, we have seen trees defined as an abstract data type using lists. Let's look at another implementation using OOP syntax. With this implementation, we can easily specify specialized tree types such as binary trees through 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 = branches
```

### 1.2 Questions

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

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

**Solution:**

```
    t.entry = t.entry ** 2
    for branch in t.branches:
        square_tree(branch)
```

2. Define a function `make_even` which takes in a tree `t` whose entries are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers 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)])
    """
```

**Solution:**

```
    if t.entry % 2 != 0:
        t.entry += 1
    for branch in t.branches:
        make_even(branch)
```

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

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

**Solution:** It would help to write a helper function, because we cannot just add recursive calls of average together directly.

```
    def sum_helper(t):
        sum_entries, count = t.entry, 1
        for branch in t.branches:
            branch_sum, branch_count = sum_helper(branch)
            sum_entries += branch_sum
            count += branch_count
        return sum_entries, count
    sum_entries, count = sum_helper(t)
    return sum_entries / count
```

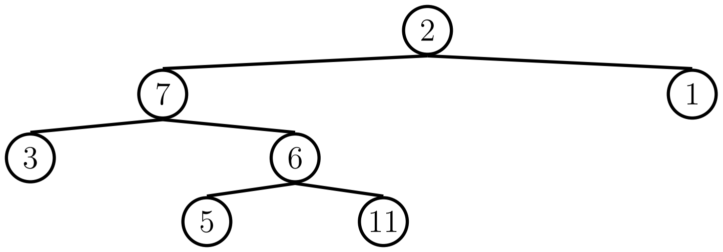
☐

## 1.3 Extra Questions

4. Define the procedure `find_path` that, given a Tree `t` and an `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):
```

**Solution:**

```
    if t.entry == entry:
        return [entry]
    for branch in t.branches:
        path = find_path(branch, entry)
        if path:
            return [t.entry] + path
    return False
```

   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`.

**Solution:** We want to check every child's entry before moving on to that child's children. It's easier to do this using an iterative implementation rather than a recursive implementation. In these two functions, we are actually running depth-first search and breadth-first search respectively, both of which are well known graph-traversal algorithms. For CS61A, you just have to recognize that a function call does not return until it is fully evaluated, so the `find_path` solution might not give the shortest path.

```
def find_shortest(t, entry):
    tree_queue = [(t, [])]
    while len(tree_queue) > 0:
        curr_tree, path = tree_queue.pop(0)
        if curr_tree.entry == entry:
            return path + [entry]
        for branch in curr_tree.branches:
            tree_queue.append((branch, path + [
               curr_tree.entry]))
    return False
```

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

**Solution:**

```
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)
            branch.parent = self
        self.branches = list(branches)
```

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`. Don't forget to make sure the parents are still correct after the swap!

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

**Solution:**

```
    assert len(self.branches) > 0 and len(other.branches) >
        0, "Must have children to swap."
    self.branches[0], other.branches[-1] = other.branches
       [-1], self.branches[0]
    self.branches[0].parent, other.branches[-1].parent =
       self, other
```

The important part here is that the parent pointers must be updated as well.

## 1.4 Binary Trees

Sometimes, it is more convenient to work with trees that have at most two branches per node. 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__`?

**Solution:**

a. We want to enforce the closure property for the BinaryTree, that each subtree of a BinaryTree is a BinaryTree instance. `Tree.__init__`'s assert only enforces that each subtree is a Tree instance. Therefore, we need to make a stronger assertion in the BinaryTree class.

b. First, an empty instance of BinaryTree is created. Then, after the assert, we will *fill in* the empty instance by calling `Tree.__init__` with particular arguments so that our tree will be a binary tree. Note that `Tree.__init__` does not create a new object, nor is it used as a bound method.

c. `self.left = self.branches[0]` will only copy the current value of `self.branches[(` not whatever that list element contains. Since the value of the list element can change anytime, we should instead re-compute the value of `self.left` every time we need the attribute. The `@property` construct helps us implement that easily.

## 1.6 Questions

1. Define a function `height(t)` that returns the height of a BinaryTree. The height is defined as the length of the *longest* path from the root node down to a leaf node. If a BinaryTree just consists of a root with no children, its height is 0.

```
def height(t):
    """Returns the height of the Tree t."""
```

**Solution:**

```
    result = 0
    if not t.left.is_empty:
        result = 1 + height(t.left)
    if not t.right.is_empty:
        result = max(1 + height(t.right), result)
    return result
```

# 2 Orders of Growth

When we talk about the efficiency of a function, we are often interested in how much more expensive it is to run the function with a larger input. That is, how does the time your code takes to run 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 function `foo` is in $\Theta(n^2)$, we mean that the running time of the process will grow proportionally to the square of the size of the input as it 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 functions.

## 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 (considered "intractable"; these are really, really horrible)

## 2.2 Orders of Growth in Time

"Time" here 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):          def double_list(lst):
    return n * 2            return [double(elem) for elem in lst]
```

- `double` runs in $\Theta(1)$, or constant time, because it does the same number of primitive operations no matter what `n` is.
- `double_list` runs in $\Theta(n)$, where `n = len(lst)`, because a $\Theta(1)$ function (`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 $n^3$ term dominates as $n$ gets larger.

## 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)
   ```

   **Solution:** factorial: $\Theta(n)$
   sum_of_factorial: $\Theta(n^2)$

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

   **Solution:** $\Theta(n)$

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

**Solution:** $\Theta(1)$

4.
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

**Solution:** $\Theta(\log(n))$

5.
```
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))`?

**Solution:** $\Theta(n^2)$

6.
```
def steven(x, y):
    # x and y are lists
    sum = 0
    for elem1 in x:
        for elem2 in y:
            sum += elem1 * elem2
    return sum
```

**Solution:** $\Theta(\text{len}(x) * \text{len}(y))$

## 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):
```

**Solution:**

```
    if lst:
        mid = len(lst) // 2
        left = list_to_bst(lst[:mid])
        right = list_to_bst(lst[mid+1:])
        return BinaryTree(lst[mid], left, right)
```

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)
```

**Solution:** Runtime of `g`: $\Theta(2^n)$ Runtime of `h`: $\Theta(n)$ Runtime of `f`: $\Theta(2^{\min(x,y)} + |x - y|)$