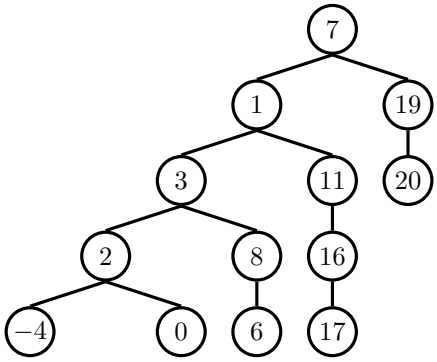
CS 61A
Summer 2018

# Trees and Orders of Growth

Discussion 5: July 05, 2018 Solutions

## 1 Trees




In computer science, **trees** are recursive data structures that are widely used in various settings. The diagram to the right is an example of a 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 branches. Parent nodes can have multiple branches.
- **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.
- **Label**: The value at a node. In our example, all of the integers are values.
- **Leaf**: A node that has no branches. In our example, the nodes that contain $-4$, 0, 6, 17, and 20 are leaves.
- **Branch**: A subtree of the root. Note that trees have branches, which are trees themselves: this is why trees are *recursive* data structures.
- **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 branches each node has; others vary in the structure of the tree.

# Implementation

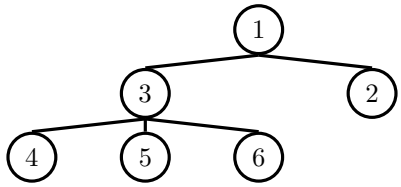
A tree has both a value for the root node and a sequence of branches, which are also trees. In our implementation, we represent the branches as a list of trees. Since a tree is an abstract data type, our choice to use lists is just an implementation detail.

- The arguments to the constructor `tree` are the value for the root node and a list of branches.
- The selectors for these are `label` and `branches`.

Note that `branches` returns a list of trees and not a tree directly. It's important to distinguish between working with a tree and working with a **list of** trees.

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

Let's try to create the tree below.




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

```
# Constructor
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

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

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

# For convenience
def is_leaf(tree):
    return not branches(tree)
```

# Questions

1.1 Consider a tree ADT `t` constructed by calling `tree(1, [tree(2), tree(4)])`. For each of the following expressions, answer these two questions:

- What does the expression evaluate to?
- Does the expression violate any abstraction barriers? If so, write an equivalent expression that does not violate abstraction barriers.

1. `label(t)`

   Evaluates to 1, the label of the entire tree. This is simply using a selector to get the label, which is not violating any abstraction barriers.

2. `t[0]`

   This expression evaluates to 1, the label of the entire tree. However, it makes use of the fact that trees are implemented using lists, and violates the abstraction barrier. An equivalent expression is `label(t)`.

3. `label(branches(t)[0])`

   This expression evaluates to the label of the first branch of `t`. It is not a violation to index into `branches(t)` because it is given in the description of the ADT that `branches(t)` returns a list of branches.

4. `is_leaf(t[1:][1])`

   This expression accesses the branches of `t` by slicing `t`. Although this works because this is technically what `branches(t)` returns, this is an abstraction violation because we cannot assume the implementation of `branches(t)`.

   It then accesses the second branch by indexing into the list of branches, which is *not* an abstraction violation because we are allowed to assume that branches is a list. This expression evaluates to `True` because the second branch of `t` is a leaf. An equivalent expression is `is_leaf(branches(t)[1])`.

5. `[label(b) for b in branches(t)]`

   This expression uses the `branches` selector to access the branches of `t` and then iterates through it to construct a new list containing the labels of the branches. The result list is `[2, 4]`. It does not violate any abstraction barriers.

6. **Challenge:** `branches(tree(5, [t, tree(3)]))[0][0]`

   This expression evaluates to the label of the tree `t`, which is 1. This is because the expression `tree(5, [t, tree(3)])` evaluates to a tree whose first branch is the tree `t` that we constructed above! Howvever, this expression violates the abstraction barrier by indexing into `t` to get its label. An equivalent expression would be `label(branches(tree(5, [t, tree(3)]))[0])`.

1.2 Write a function that returns the number of branches of a tree.

```
def num_branches(t)
    """Return the number of branches of a tree.
```

```
    >>> t = tree(1, [tree(2), tree(3)])
    >>> num_branches(t)
    2
    """

    return len(branches(t))
```

1.3 Write a function that returns the largest number in a tree.

```
def tree_max(t):
    """Return the maximum label in a tree.

    >>> t = tree(4, [tree(2, [tree(1)]), tree(10)])
    >>> tree_max(t)
    10
    """

    return max([label(t)] + [tree_max(branch) for branch in branches(t)])
```

Video walkthrough

1.4 Write a function 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.

    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    """


    if is_leaf(t):
        return 0
    return 1 + max([height(branch) for branch in branches(t)])
```

Video walkthrough

1.5 Define the function tree_map, which takes in a tree and a one-argument function as arguments and returns a new tree which is the result of mapping the function over the entries of the input tree.

```
def tree_map(fn, t):
    """Maps the function fn over the entries of tree and returns the
    result in a new tree.

    >>> numbers = tree(1,
    ...                [tree(2,
    ...                      [tree(3),
    ...                       tree(4)]),
    ...                 tree(5,
    ...                      [tree(6,
    ...                            [tree(7)]),
    ...                       tree(8)])])
    >>> print_tree(tree_map(lambda x: 2**x, numbers))
    2
      4
        8
        16
      32
        64
          128
        256
    """


    if is_leaf(t):
        return tree(fn(label(t)))
    mapped_subtrees = []
    for b in branches(t):
        mapped_subtrees += [tree_map(fn, b)]
```

```
    return tree(fn(label(t)), mapped_subtrees)

# Alternate solution
    return tree(fn(label(t)), [tree_map(fn, t) for t in branches(t)])
```

1.6 An **expression tree** is a tree that contains a function for each non-leaf node, which can be either `'+'` or `'*'`. All leaves are numbers. Implement `eval_tree`, which evaluates an expression tree to its value. You may want to use the functions `sum` and `prod`, which take a list of numbers and compute the sum and product respectively.

```
def eval_tree(tree):
    """Evaluates an expression tree with functions the root.
    >>> eval_tree(tree(1))
    1
    >>> expr = tree('*', [tree(2), tree(3)])
    >>> eval_tree(expr)
    6
    >>> eval_tree(tree('+', [expr, tree(4), tree(5)]))
    15
    """


    if is_leaf(tree):
        return label(tree)
    args = [eval_tree(branch) for branch in branches(tree)]
    if label(tree) == '+':
        return sum(args)
    else: # label(tree) == '*'
        return prod(args)
```

Leaf values are guaranteed to be a number, so we can just return their label.

Otherwise, we have to evaulate each of the branches and then combine their result using whichever operater we have in our current root.

If you want to try this out yourself, note that `prod` isn't actually a built-in operator in Python. You can write it yourself using something like the following:

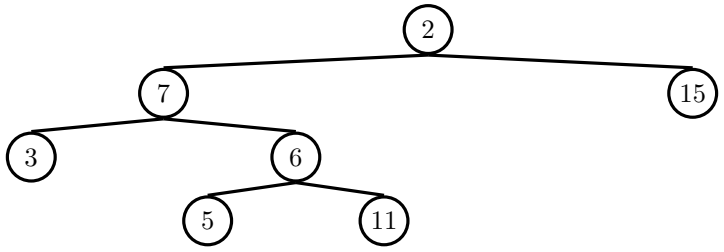
```
def prod(iterable):
    from functools import reduce
    from operator import mul
    return reduce(mul, iterable, 1)
```

Video walkthrough

1.7 Write a function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`.

If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

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



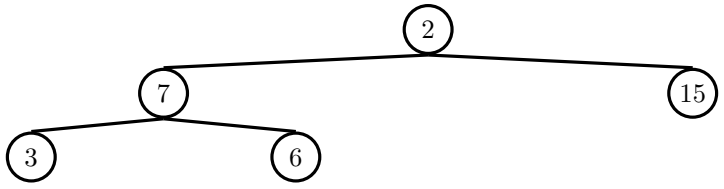

```
def find_path(tree, x):
    """
    >>> find_path(t, 5)
    [2, 7, 6, 5]
    >>> find_path(t, 10)  # returns None
    """


    if label(tree) == x:
        return [label(tree)]
    for b in branches(tree):
        path = find_path(b, x)
        if path:
            return [label(tree)] + path
```

Video walkthrough

1.8 Write a function that takes in a tree and a depth `k` and returns a new tree that contains only the first `k` levels of the original tree.

For example, if `t` is the tree shown in the previous question, then `prune(t, 2)` should return the following tree.




```
def prune(t, k):


    if k == 0:
        return tree(label(t), [])
    else:
        return tree(label(t), [prune(branch, k - 1) for branch in branches(t)])
```

Video walkthrough

# 2 Orders of Growth

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by "runtime"?

- `square(1)` requires one primitive operation: `*` (multiplication). `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes one operation.

| input | function call | return value | number of operations |
|---|---|---|---|
| 1 | square(1) | $1 \cdot 1$ | 1 |
| 2 | square(2) | $2 \cdot 2$ | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 100 | square(100) | $100 \cdot 100$ | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $n$ | square($n$) | $n \cdot n$ | 1 |

- `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases linearly proportional to the input.

| input | function call | return value | number of operations |
|---|---|---|---|
| 1 | factorial(1) | $1 \cdot 1$ | 1 |
| 2 | factorial(2) | $2 \cdot 1 \cdot 1$ | 2 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 100 | factorial(100) | $100 \cdot 99 \cdots 1 \cdot 1$ | 100 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $n$ | factorial($n$) | $n \cdot (n-1) \cdots 1 \cdot 1$ | $n$ |

Here are some general guidelines for finding the order of growth for the runtime of a function:

- 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 in terms of input size $n$.
  - Find how much work is done per recursive call or iteration in terms of input size $n$.

  The answer is usually the product of the above two, but be sure to pay attention to control flow!
- If the function calls helper functions that are not constant-time, you need to take the runtime 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.

# Big-Theta Notation

For expressing complexity, we use what is called big $\Theta$ (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 with the square of the size of the input as it becomes very large.

- **Ignore lower order terms:** If a function requires $n^3 + 3n^2 + 5n + 10$ operations with a given input $n$, then the runtime of this function is in $\Theta(n^3)$. As $n$ gets larger, the lower order terms (10, $5n$, and $3n^2$) all become insignificant compared to $n^3$.
- **Ignore constants:** If a function requires $5n$ operations with a given input $n$, then the runtime of this function is in $\Theta(n)$. We are only concerned with how the runtime grows asymptotically with the input, and since 5n is still asymptotically linear; the constant factor does not make a difference in runtime analysis.

# 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
- $\Theta(n \log n)$ — linearithmic time
- $\Theta(n^2)$, $\Theta(n^3)$, etc. — polynomial time
- $\Theta(2^n)$, $\Theta(3^n)$, etc. — exponential time (considered "intractable"; these are really, really horrible)

In addition, some programs will never terminate if they get stuck in an infinite loop.

# Questions

What is the order of growth for the following functions?

2.1
```
def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

$\Theta(n^2)$, we will call factorial $n$ times with arguments $n, n-1, n-2, ..., 0$. The sum from 0 to $n$ is approximately $n^2$.
Video walkthrough

2.2
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

$\Theta(\log(n))$, because our while loop iterates at most $\log(n)$ times, due to $n$ being halved in every iteration.
Video walkthrough

2.3
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

$\Theta(1)$, since at worst it will require 6 recursive calls to reach the base case. So this is $\Theta(6)$, which can be reduced to $\Theta(1)$.

# Extra Questions

2.4
```
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))`?

$\Theta(n^2)$