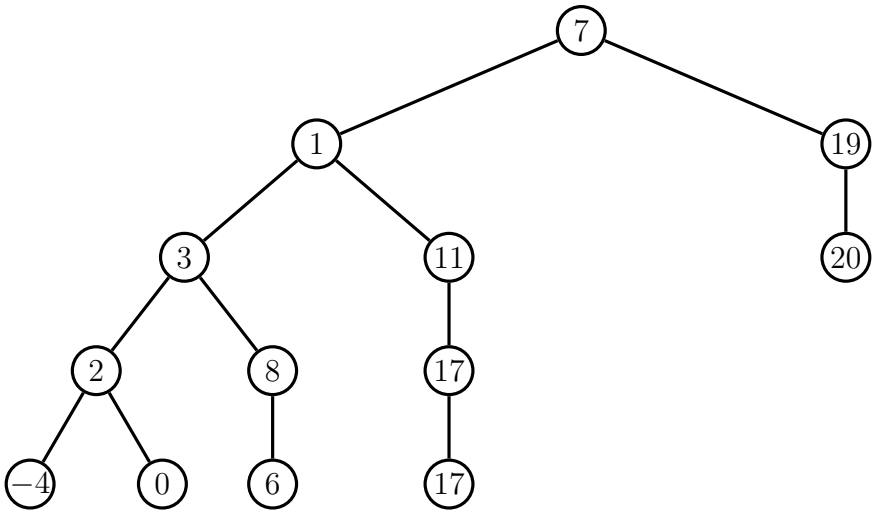
TREES AND ORDERS OF GROWTH 7

COMPUTER SCIENCE 61A

October 17, 2013

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.

Trees consist of three components: an entry, a left child, and a right child.

1. **Entry**: Each tree houses one item (entry). The data could be numbers, strings, tuples, etc.
2. **Left**: The left child of the current node. Note that the child is another tree.
3. **Right**: The right child of the current node. Note that the child is another tree.

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. There is only one root. Because every other node branches directly or indirectly from the root, it is possible to start from the root and reach any other node in the tree. The root is, of course, a parent — it is the only node that is not a child. For example, the node that contains the 7 at the top is the root.
- **Leaf**: Nodes that have no children. For example, the nodes that contain the bottom -4, 0, 6, 17, and 20 are leaves. The node that contains 19 is not a leaf, since it has one child.
- **Subtree**: Notice that each child of a parent is itself the root of a smaller tree (for 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, how many generations away from the root is the specific child node? In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. We define the root of a tree to have depth 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 3. Thus, the entire tree has height 3.

In computer science, there are many different types of trees – some vary in the number of children each node has, and others vary in the structure of the tree.

1.1 Our Implementation

In CS 61A, we will be working with a special type of tree: the binary tree, which has an extra condition that each node can have at most two children.

```
class Tree:

    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

    def __repr__(self):
        args = repr(self.entry)
        if self.left or self.right:
            args += ', {0}, {1}'.format(repr(self.left), repr(self.right))
```

```
        return 'Tree({0})'.format(args)
```

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"""
```

Solution:

```
    if t:
        t.entry = t.entry ** 2
        square_tree(t.left)
        square_tree(t.right)
```

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

   If it helps, there is a Python built-in function `max` that takes an arbitrarily long sequence of numbers and returns the maximum value in the sequence.

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

Solution:

```
    result = 0
    if t.left:
        result = max(height(t.left), result)
    if t.right:
        result = max(height(t.right), result)
    return 1 + result
```

3. Let's actually define `tree_size(t)`, which returns how many items the Tree t contains. Try to solve this without the given size or items property.

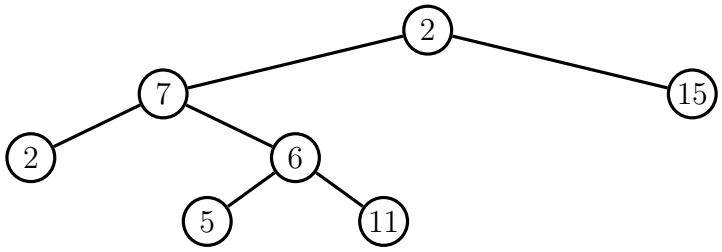
```
def tree_size(t):
    """Returns the number of items in a Tree t."""
```

Solution:

```
if not t:
    return 0
return 1 + tree_size(t.left) + tree_size(t.right)
```

4. Define the procedure `find_path` that, given an Tree `t` and an entry `entry`, returns a tuple 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`. Assume that the elements in `t` are unique.

   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:

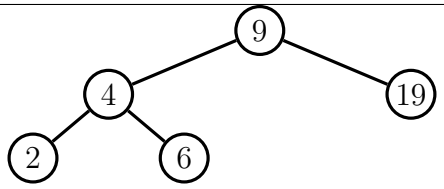
```
def find_path(t, entry):
    if not t:
        return False
    if t.entry == entry:
        return (entry,)
    for child in (t.left, t.right):
        path = find_path(child, entry)
        if path:
            return (t.entry,) + path
    return False
```

1.3 Binary Search Trees

Note that there is a special subtype of binary trees: the binary *search* tree, which must meet the following conditions:

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- The left and right subtrees must also be binary search trees.
- There must no duplicate nodes.

Here is an example of a binary search tree:



1.4 Questions

1. Write a function that prints out all of the values in a binary search tree in increasing order.

```
def print_inorder(t):
```

Solution:

```
    if t:
        print_inorder(t.left)
        print(t.entry)
        print_inorder(t.right)
```

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, as the size of the input grows, how do the speed of the procedure and the space its process occupies grow?

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`. More generally, we can say that `foo` is in some $\Theta(f(n))$ if there exist some constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \tag{1}$$

for $n > N$, where N is sufficiently large.

This is a mathematical definition of big Theta notation. To prove that `foo` is in $\Theta(f(n))$, we only need to find constants k_1 and k_2 where the above holds.

There is also another way to express orders of growth: big O notation. This denotes the worst case complexity of a procedure, whereas big Theta notation gives a rough approximation of the actual complexity. Still, big O notation can be useful when it is not possible

to find a big Theta. The mathematical definition of big O is, for some values k_1 and n,

$$R(n) \leq k_1 \times f(n) \tag{2}$$

for $n > N$, where N is sufficiently large.

For example, $O(n^2)$ states that a function's worst case run time would be in quadratic time. This does not mean the function will never be slower than quadratic time; in fact, it might very well run in linear or even constant time!

Fortunately, in CS 61A, we're not that concerned with rigorous mathematical proofs. (You'll get the painful details in CS 61B!) What we want you to develop in CS 61A is the intuition to guess the orders of growth for certain procedures.

2.1 Kinds of Growth

Here are some common orders of growth, ranked from best to worst:

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

2.2 Orders of Growth in Time

"Time," for us, basically refers to the number of recursive calls or the number of times the suite of a `while` loop executes. Intuitively, the more recursive calls we make, the more time it takes to execute the function.

- If the function contains only primitive procedures like $+$ or $*$, then it is constant time – $\Theta(1)$.
- If the function is recursive, you need to:
  - Count the number of recursive calls that will be made, given input n.
  - Count how much time it takes to process the input per recursive call.

  The answer is usually the product of the above two. For example, let's try to sum all of the items in an `rlist` recursively. Each addition takes us 3 seconds, and there are 10 items. Therefore, we would take $3 \times 10 = 30$ seconds to calculate the summation.
- If the function contains calls of helper functions that are not constant-time, then you need to take orders of growth of the helper functions into consideration as well. In general, how much time the helper function takes would be included.

- When we talk about orders of growth, we don't really care about constant factors. So if you get something like $\Theta(1000000n)$, this is really $\Theta(n)$. We can also usually ignore lower-order terms. For example, if we get something like $\Theta(n^3+n^2+4n+399)$, we can take it to be $\Theta(n^3)$. This is because the general shape of the plot for $n^3+n^2+4n+399$ is very similar to n^3 as we look towards where n approaches infinity. However, note that in practice, these extra terms and constant factors 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)
```

Solution: $\Theta(n^2)$

2.
```
def fibonacci(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Solution: $\Theta(\Phi^n)$, where Φ is the golden ratio.

3.
```
def fib_iter(n):
    prev, cur, i = 0, 1, 1
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return curr
```

Solution: $\Theta(n)$

4.
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Solution: $\Theta(1)$

5. Given:

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

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

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.

```
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 Tree(lst[mid], left, right)
```

2. Write a function, `find_path_bst`, that takes in a binary search tree and a value. It returns the path from the root to the value. If the value is not in the tree, return `False`. This function should run in $O(\log n)$ time, where n is the number of nodes in the tree.

```
def find_path_bst(t, val):
```

Solution:

```
if not t:
    return False
if t.entry == val:
    return (val,)
if t.entry > val:
    path = find_path_bst(t.left, val)
    if path:
        return (t.entry,) + path
else:
    path = find_path_bst(t.right, val)
    if path:
        return (t.entry,) + path
return False
```

3. Given the following function, `f`, give the running times of the functions `g` and `h` in terms of `n`.

```
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`: $\Theta(n)$