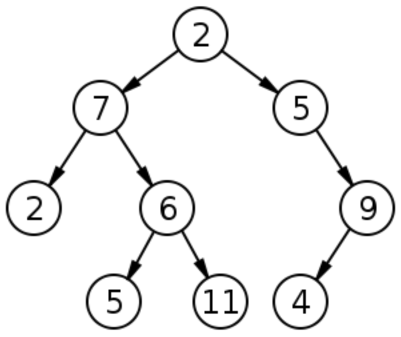
Trees, Orders of Growth, and Exceptions 5a

Computer Science 61A

July 23, 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 two components: an entry, and children.

1. **Entry**: Each tree houses one item (entry). The data could be numbers, strings, tuples, etc.
2. **Children**: This is a sequence containing all of its *children* (each of which are trees).

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 2 at the top is the root.
- **Leaf**: Nodes that have no children. For example, the nodes that contain the bottom 2, 5, 11, and 4 are leaves. The node that contains 9 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 7 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 7 has depth 1; the node containing 6 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 5, 11, and 4 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

For a given Tree `t`, here are a few of the attributes implemented inside tree.py:

- **t.entry**: returns the entry housed inside the root
- **t.children**: returns a sequence of children trees from the root
- **t.size**: returns how many items are inside the tree
- **t.is_leaf**: checks whether the given tree has no children
- **t.items**: returns a tuple of all the items in the tree
- **t.map(func)**: mutates t based on the given function
- **t.copy()**: returns a copy of t
- **str(t)**: prints out a visual representation of t

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.

   Hint: Use one (or more) of the Tree methods provided above to solve this problem!

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

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.

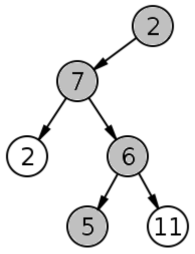
```
def height(t):
    """Returns the height of the Tree t."""
```

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."""
```

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):
```

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-Oh notation. This denotes the worst case complexity of a procedure, whereas big-Theta notation gives a rough approximation of the actual complexity. Still, big-Oh notation can be useful when it is not possible to find a big-Theta. The mathematical definition of big-Oh 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 CS61A, we're not that concerned with rigorous mathematical proofs (you'll get the painful details in CS61B!). What we want you to develop in CS61A 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, given a fruit basket with 10 apples, how long does it take for me to process the whole basket? Well, I'll recursively call my `eat` procedure, which eats one apple at a time (so I'll call the procedure 10 times). Each time I eat an apple, it takes me 30 minutes. So the total amount of time is just $30 \times 10 = 300$ minutes!
- 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 $\Theta(n^3)$.

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

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

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

6.
```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

3 Exceptions

Up to this point in the semester, we have assumed that the input to our functions are always correct, and thus have not done any error handling. However, functions can often have large domains, and we want our functions to handle erroneous input gracefully. This is where exceptions come in.

Exceptions provide a general mechanism for adding error-handling logic to programs. **Raising an exception** is a technique for interrupting the normal flow of execution in a program, signaling that some exceptional circumstance has arisen.

An **exception** is an object instance of a class that inherits, either directly or indirectly, from the `BaseException` class. The following is an example of how to raise an exception:

```
>>> raise Exception('An error occurred')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: An error occurred
```

Notice how the string 'An error occurred' is an argument to the `Exception` object being created, and the string is part of what Python prints out in response to the exception being raised.

If the exception is raised while with a `try` statement, then the interpreter will immediately look for an `except` statement that handles the type of exception being raised. `try` and `except` statements allow programs to respond to unexpected arguments and other errors gracefully, rather than terminating entirely.

Here's how to structure `try` and `except` statements:

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
except <exception class> as <name>:
    <except suite>
```

3.1 Questions

1. Fill in all the blanks to produce the desired output:

```
>>> try:
...     x = ______
... except _______________ as ___:
...     print('handling a', type(e))
...     x = _____
handling a <class 'ZeroDivisionError'>
>>> x
9001
```

2. Write the function `safe_square` that uses exceptions to print 'Incorrect argument type' when anything other than an `int` or `float` class is given as an argument. Otherwise, `safe_square` should multiply the argument by itself. A useful fact is that a `TypeError` is raised when * is given incorrect arguments.

```
def safe_square(x):
```

3. Predict the output of each of the following lines, assuming `safe_square` is implemented as described in the previous question.

```
>>> safe_square('hello')

>>> safe_square('hello * 5)

>>> safe_square('hello' * 'hello')

>>> safe_square(1 * 2.5)

>>> safe_square(1/ 0)
```