TREES AND EFFICIENCY AND EXCEPTIONS 9

DATA C88C

October 31, 2023

1 Trees

1.1 Introduction

In computer science, **trees** are recursive data structures that are widely used in various settings. Contrary to our ideas of a tree, in computer science, a tree branches downward. The **root** of a tree starts at the top, and the **leaves** are at the bottom. A tree is considered a *recursive* data structure because every branch from a node is also a tree.

1.2 Implementation

Recall that we have defined a tree as having a value and a list of branches. Below is the most basic implementation of a Tree class that we will be using.

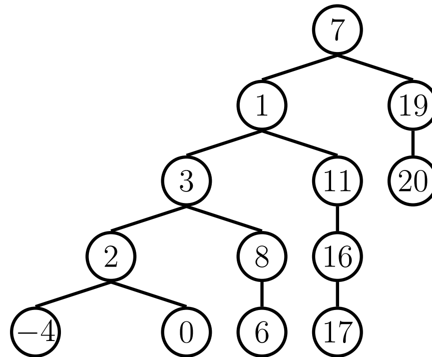
```
class Tree:
    def __init__(self, value, branches=()):
        for b in branches:
            assert isinstance(b, Tree)
        self.value = value
        self.branches = list(branches)
    def is_leaf(self):
        return not self.branches
    def __repr__(self):
        if self.branches:
            branches_str = ', ' + repr(self.branches)
        else:
            branches_str = ''
        return 'Tree({0}{1})'.format(self.value, branches_str)
```

Notice that with this implementation we can mutate a tree using attribute assignment.

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.value = 5
>>> t
Tree(5, [Tree(4), Tree(5)])
```

1.3 Definitions

Here is an example tree:



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.

2 Questions

1. What would Python display? If you believe an expression evaluates to a `Tree` object, write *Tree*.

```
>>> t0 = Tree(0)
>>> t0.value
```

Solution: `0`

```
>>> t0.branches
```

Solution: `[]`

```
>>> t1 = Tree(0, [1, 2])#Is this a valid tree?
```

Solution: `AssertionError #As the branches must be Tree objects`

```
>>> t2 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
>>> t2.branches[0]
```

Solution: `Tree(1)`

```
>>> t2.branches[1].branches[0].value
```

Solution: `3`

2. Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 but all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.value
    2
    >>> t.branches[0].branches[0].value
    4
    >>> t
    Tree(2, [Tree(2, [Tree(4)]), Tree(4), Tree(6)])
    """
```

Solution:

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

3. Write a function that combines the values of two trees `t1` and `t2` together with the `combiner` function. Assume that `t1` and `t2` have identical structure. This function should return a new tree.

```
def combine_tree(t1, t2, combiner):
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = combine_tree(a, b, mul)
    >>> combined.value
    4
    >>> combined.branches[0].value
    10
    >>> combined
    Tree(4, [Tree(10, [Tree(18)])])
    """
```

Solution:

```
    combined = [combine_tree(b1, b2, combiner) for b1, b2
                in zip(t1.branches, t2.branches)]
    return Tree(combiner(t1.value, t2.value), combined)
```

Alternate solution without using `zip`:

```
    combined = []
    for i in range(len(t1.branches)):
        combined.append(combine_tree(t1.branches[i], t2.
           branches[i], combiner))
    return Tree(combiner(t1.value, t2.value), combined)
```

3 Efficiency

3.1 Introduction

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
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	square(100)	$100 \cdot 100$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
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
$\vdots$	$\vdots$	$\vdots$	$\vdots$
100	factorial(100)	$100 \cdot 99 \cdots 1 \cdot 1$	100
$\vdots$	$\vdots$	$\vdots$	$\vdots$
n	factorial(n)	$n \cdot (n-1) \cdots 1 \cdot 1$	n

3.2 Questions

1. Previously, we looked at the `is_prime` function. Here's the code for it:

```
def is_prime(n):
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True
```

What is the order of growth of `is_prime`?

Solution: $O(n)$

How can we change `is_prime` so that it runs in $O(\sqrt{n})$?

```
def is_prime(n):
```

Solution: If `n` is not prime, it can be factored into at least two factors. If both of the factors are greater than the $\sqrt{n}$, then their product is greater than `n`. Therefore, one of the factors must be less than or equal to $\sqrt{n}$.

```
    if n == 1:
        return False
    k = 2
    while k * k <= n:
        if n % k == 0:
            return False
        k += 1
    return True
```

2.
```
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: $\Theta(n^2)$

4 Exceptions

4.1 Introduction

Exceptions are used to signify when something goes wrong in your program. For interpreters, they're often used to categorize a case when the user inputs something that doesn't make sense (just try typing in `Hi Soumya` in your Python interpreter and see what happens!)

There are two major things that you do with exceptions: `raise` and **handle** them.

Generally, to raise an exception you use the statement `raise <expression>`.

To handle an exception, you use a `try-except` block. The syntax is as follows:

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

You can have multiple `except` suites for different types of exceptions that might occur in the try suite.

4.2 Questions

1. How do we raise exceptions in Python?

Solution: An exception is a object instance with a class that inherits, either directly or indirectly, from the BaseException class. The assert statement introduced in Chapter 1 raises an exception with the class AssertionError. In general, any exception instance can be raised with the raise statement. The general form of raise statements are described in the Python docs. The most common use of raise constructs an exception instance and raises it.

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

2. How do we handle raised exceptions? And why would we need to do so?

Solution: An exception can be handled by an enclosing try statement. A try statement consists of multiple clauses; the first begins with try and the rest begin with

except:

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

The ¡try suite¿ is always executed immediately when the try statement is executed. Suites of the except clauses are only executed when an exception is raised during the course of executing the try suite. Each except clause specifies the particular class of exception to handle.
We want to handle exceptions if we don't want our program to crash immediately when it encounters an error, and if we can anticipate the errors that would occur/have pre-defined ways of handling them.