Tree Traversal

Introduction

In this lab, we will practice tree traversal algorithms, and get to use trees for different applications, including expression trees.

Trees data types are ideal for representing hierarchical structure. Trees are composed of nodes and nodes have 0 or more children or child nodes. A node is called the parent of its children. Each node has (at most) one parent. There is a single special node called the root of the tree. The root is the only node that does not have a parent. The nodes that do not have any children are called leaves or leaf nodes. If the children are ordered in some way, then we have an ordered tree.

There are many examples of hierarchical (tree-like) structures:

1. Class Hierarchies (assuming single inheritance). The subclass is the child and the superclasses is the parent. The python class object is the root.
2. File folders or directories on a computer can be nested inside other directories. The parent-child relationship encode containment.
3. The tree of recursive function calls in the execution of a divide-and-conquer algorithm. The leaves are those calls where the base case executes.

The purpose of this assignment is to help you:

1. Gain familiarity of the tree data structure, for binary and general trees
2. Practice preorder, postorder, and inorder tree traversal algorithms.
3. Get familiar with generic visit function for trees.
4. Perform evaluation of expression trees

When we draw trees, we take an Australian convention of putting the root on the top and the children below. Although this is backwards with respect to the trees we see in nature, it does correspond to how we generally think of most other hierarchies.

Even though we use the family metaphor for many of the naming conventions, a family tree is not actually a tree. The reason is that these violate the one parent rule.

Part 1: In-lab

As part of this lab, you have been provided code to create a tree from a tree specification. This code makes no assumptions how many children a node may have, or whether the specification is a `tuple` or a `list`:

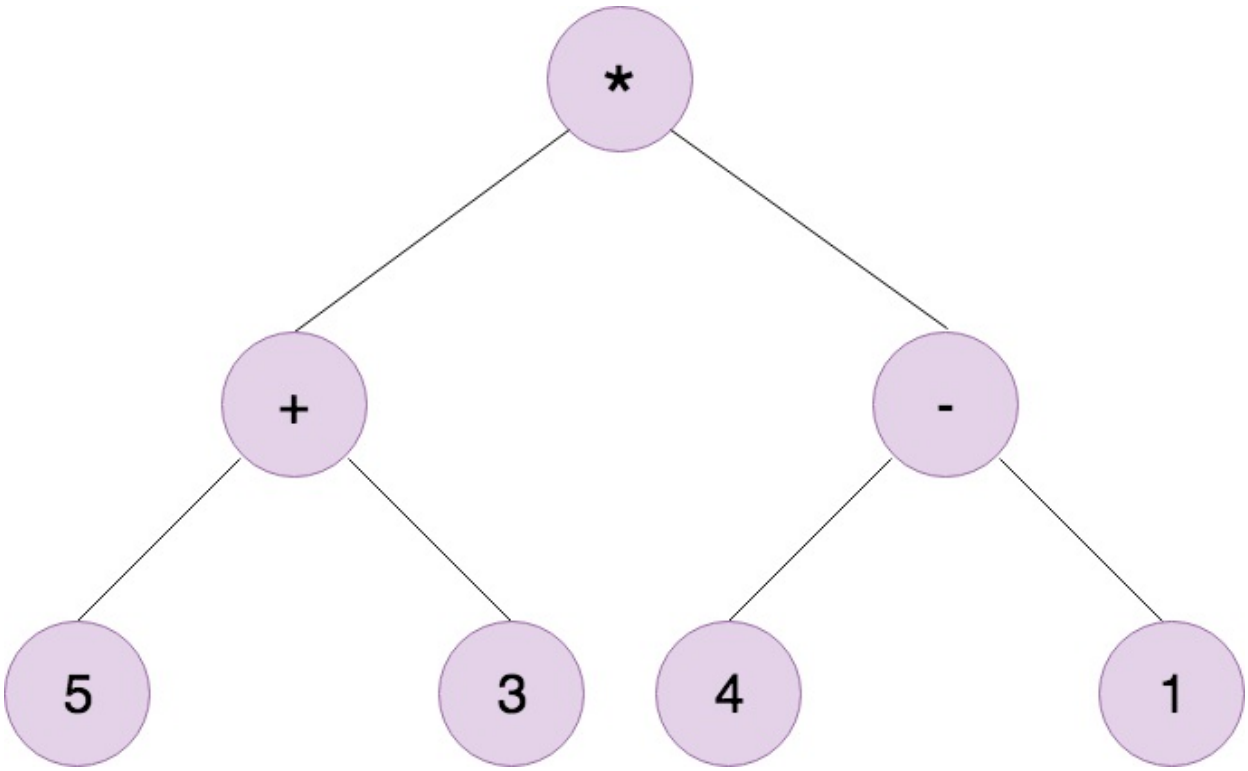
```
class Tree:
  def __init__(self, spec):
    if type(spec) is tuple or type(spec) is list:
      self.data = spec[0]
      self.children = [Tree(subSpec) for subSpec in spec[1:]]
    else:
      self.data = spec
      self.children = []

  def printpreorder(self):
    print(self.data)
    for child in self.children:
      child.printpreorder()
```

It has a printpreorder method that prints the contents of the tree in preorder fashion.

Example 1: Expressions

The expression `(5 + 3) * (4 - 1)` is represented by the following expression tree:




```
expr1 = ('*', ('+', 5, 3), ('-', 4, 1))
```

```
exprTree = Tree(expr1)
exprTree.printpreorder()
```

The output is:

```
*
+
5
3
-
4
1
```

Examples 2: Book contents

```
book1 = [('Make Money Fast!', 0), [('Motivations', 2), [('Greed', 5)],
[('Avidity', 10)]], [('Methods', 15), [('Stock Fraud', 20)],
[('Ponzi Scheme', 25)], [('Bank Robbery', 30)]], [('References', 40)]]

bookTree = Tree(book1)
print(" Calling preorderprint method:")
bookTree.printpreorder()
```

The output is:

```
Calling preorderprint method:
('Make Money Fast!', 0)
('Motivations', 2)
('Greed', 5)
('Avidity', 10)
('Methods', 15),
('Stock Fraud', 20)
('Ponzi Scheme', 25)
('Bank Robbery', 30)
('References', 40)
```

Your task:

Now, define a class `BinaryTree` where the nodes have 3 attributes: `data`, `leftChild`, and `rightChild`.
It will have the same two functions, `__init__` and `printpreorder` as regular `Trees`.

The input to `__init__` should be a `tuple` or a `list`, of length between `1` and `3`. If it's `1`, we have no children (`leftChild` and `rightChild` are set to `None`), so it's a leaf. If it's `3`, then both `leftChild` and `rightChild` are trees (objects of type `BinaryTree`).

For any tree specification TS where no node happens to have more than `2` children, `BinaryTree.printpreorder` should produce the same output as `Tree.printpreorder`.

```
expr = ('*', ('+', 5, 3), ('-', 4, 1))
exprTree = BinaryTree(expr)
exprTree.printpreorder() # produces the same print-out as above!
```

Part 2: Printing Book Content

In the background section, we mentioned that trees are ideal for representing hierarchical structure. In our daily life, book, this object, follows a standard class hierarchy which can be represented in a tree manner, where a book contains chapters and a chapter contains different sections.

We already saw an example of book contents in Part 1 of the lab:

```
Tbook = [('Make Money Fast!', 0), [('Motivations', 2), [('Greed', 5)],
[('Avidity', 10)]], [('Methods', 15), [('Stock Fraud', 20)],
[('Ponzi Scheme', 25)], [('Bank Robbery', 30)]], [('References', 40)]]

treeBook = Tree(Tbook)
```

As we can see, the data contained in each node of the tree is a `tuple`. Each `tuple` contains the content name and the corresponding page.

An easy way to print the table of contents of this book is to call the `printpreorder` method , as we did in part 1 of the lab:

```
print(" Calling printpreorder method:")
treeBook.printpreorder()
```

The output is:

```
Calling printpreorder method:
```

```
('Make Money Fast!', 0)
('Motivations', 2)
('Greed', 5)
('Avidity', 10)
('Methods', 15),
('Stock Fraud', 20)
('Ponzi Scheme', 25)
('Bank Robbery', 30)
('References', 40)
```

This output shows all the book contents in order but not nicely. For example, we can't tell which are chapters and which are sections. So, in this part of the lab, we will implement a modified version of `printpreorder` method, called `printbookcontent` method, so that the table of contents can be printed nicely:

```
Calling printbookcontent method:
Book title: 'Make Money Fast!'
1. 'Motivations', page 2
1.1 'Greed', page 5
1.2 'Avidity', page 10
2 'Methods', page 15
2.1 'Stock Fraud', page 20
2.2 'Ponzi Scheme', page 25
2.3 'Bank Robbery', page 30
3 'References', page 40
```

Compared with the original `printpreorder` method, `printbookcontent` has the following changes:

1. We added a 'Book Title:' before the title of the book, and we skip printing the page information of the title.

Note: we cannot locate the node which contains book title by simply checking its page. For example, the page of the book title can be `0`, or `-1`, or `1` or `None` or even some special numbers that we not know. Instead, we must take advantage of the fact that book title node is always the root.

2. For all the following chapters and sections, we added 'page:' before each page number.

3. The title of each chapter or section is prefixed by its number. Note that the numbers can get arbitrarily long, if our book happens to have chapters, and subchapters, and subsubchapters, and sections and subsections and subsubsections… you get the idea.

Part 3: Space Computation of File System

Previously, all the collections we stored were either sequential (i.e., `list`, `tuple`, and `str`) or non-sequential (i.e., `dict` and `set`). The tree structure seems to lie somewhere between the two. There is some structure, but it's not linear.

For trees, the process of visiting all the nodes is called tree traversal. This is analogous to iteration through a sequential collection. However, unlike sequential collections, there is not a unique way to do this.

In general, there are two standard traversals, called preorder and postorder, both are naturally defined recursively. In a preorder traversal, we visit the node first followed by the traversal of its children. In a postorder traversal, we traverse all the children and then visit the node itself.

The visit refers to whatever computation we want to do with the nodes. The `printpreorder` method in Parts 1 and 2 of the Lab is a classic example of a preorder traversal, where the visit operation consists of printing the node's data.

Here is how the same tree printing would look with postorder traversal.

```
class Tree:
       # definitions of other methods skipped

    def printpostorder(self):
        for child in self.children:
            child.printpostorder()
        print(self.data)

expr1 = ('*',('+',5,3),('-',4,1))
exprTree1 = Tree(expr1)
exprTree1.printpostorder()
```

The output is:

```
5
3
+
4
1
-
*
```

In this part of the lab, we will still print trees in pre-order as before, but we will use post-order traversal to compute the space used by a file system.

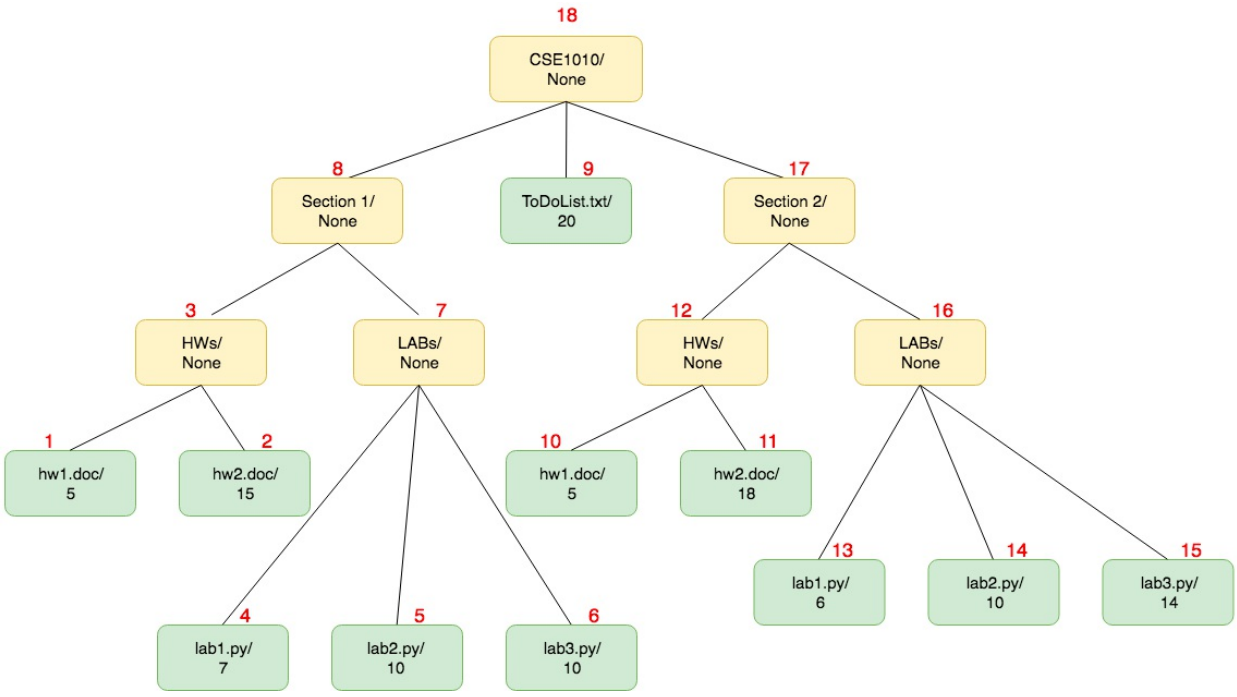
A computer file system can be represented in a tree manner, where file folders or directories on a computer can be nested inside other directories. And, in this subsection, we will implement one method which helps to compute the space used by files in a directory and its subdirectories in a tree-structured file system.

An example of a file system:

```
Tfile = [('CSE1010/', None), [('Section 1', None), [('HWs/', None),
[('hw1.doc', 5)], [('hw2.doc', 15)]], [('LABs/', None), [('lab1.py', 7)],
[('lab2.py', 10)], [('lab3.py', 10)]]], [('Section 2', None),
[('HWs/', None), [('hw1.doc', 5)], [('hw2.doc', 18)]], [('LABs/', None),
[('lab1.py', 6)], [('lab2.py', 10)], [('lab3.py', 14)]]],
[('ToDoList.txt', 20)]]

treeFile = Tree(Tfile)
```

We can visualize the file system as the following. Similar to the previous example, the data contained in each node of the tree is also a tuple. The tuple contains the folder or file name and the size of each file, while the space of each directory (folder) is temporally unknown.



So, here we will implement the `computespace` method. This method will help to compute the space used by files in the root directory and its subdirectories (CSE1010/, HWs/ and LABs/). The size of the files and/or the size of the subdirectories will be summed up to be the size of their parent directory. So the size of any directory equals the total space of all the files and folders in that directory. This method can be implemented by taking advantage of the postorder tree traversal algorithm. In the file system tree figure, the red numbers above each of

the node shows the order that each node should be visited.

A simple example is shown for your reference.

```
Tfile = [('CSE1010/', None), [('Section 1', None), [('HWs/', None),
[('hw1.doc', 5)], [('hw2.doc', 15)]], [('LABs/', None), [('lab1.py', 7)],
[('lab2.py', 10)], [('lab3.py', 10)]]], [('Section 2', None),
[('HWs/', None), [('hw1.doc', 5)], [('hw2.doc', 18)]], [('LABs/', None),
[('lab1.py', 6)], [('lab2.py', 10)], [('lab3.py', 14)]]],
[('ToDoList.txt', 20)]]

treeFile = Tree(Tfile)
print("Original File System Tree:")
treeFile.printpreorder()
print("\nComputing Space...\n")
treeFile.computespace()
print("File System Tree After Computing Space")
treeFile.printpreorder()
```

The output is as following. We can see, the original file system is initialized without the information of the space usage of directories. After calling the `computespace` method, the space used by files in the root directory and its subdirectories are updated.

```
Original File System Tree:
('CSE1010/', None)
('Section 1', None)
('HWs/', None)
('hw1.doc', 5)
('hw2.doc', 15)
('LABs/', None)
('lab1.py', 7)
('lab2.py', 10)
('lab3.py', 10)
('Section 2', None)
('HWs/', None)
('hw1.doc', 5)
('hw2.doc', 18)
('LABs/', None)
('lab1.py', 6)
('lab2.py', 10)
('lab3.py', 14)
('ToDoList.txt', 20)

Computing Space...

File System Tree After Computing Space
('CSE1010/', 120)
('Section 1', 47)
('HWs/', 20)
```

```
('hw1.doc', 5)
('hw2.doc', 15)
('LABs/', 27)
('lab1.py', 7)
('lab2.py', 10)
('lab3.py', 10)
('Section 2', 53)
('HWs/', 23)
('hw1.doc', 5)
('hw2.doc', 18)
('LABs/', 30)
('lab1.py', 6)
('lab2.py', 10)
('lab3.py', 14)
('ToDoList.txt', 20)
```

Part 4: Pretty-Printing and Evaluating Expressions

How to print expressions so they look "good", meaning familiar to us? Neither preorder or postorder makes sense. But there is also inorder traversal.

Here is how inorder print looks for binary trees:

```
def printinorder(self):
    leftChild.printinorder()
    print(self.data)
    rightChild.printinorder()
```

But this still puts each number or operation on a separate line! To avoid this, we can build one output string during the traversal, and then return it at the end:

```
def printToStr(self):
    sOut = ""
    sOut += leftChild.printToStr()
    sOut += self.data
    sOut += rightChild.printToStr()
    return sOut
```

We have provided a `printExpr` method for the `Tree` class, that pretty-prints an expression using inorder traversal:

```
  def printExpr(self):
    sOut = ""
```

```
    if len(self.children) > 0:
      sOut += '(' + self.children[0].printExpr() + ')'
    sOut += str(self.data)
    if len(self.children) > 1:
      sOut += '(' + self.children[1].printExpr() + ')'
    return sOut
```

Example:

```
expr1 = ('*',('+',5,3),('-',4,1))
exprTree1 = Tree(expr1)
s = exprTree1.printExpr()
print(s)  # prints ((5)+(3)*((4)-(1))
```

Make sure this example works for you. And with a little clean-up of extra parentheses, this looks like a real expression!

But can we evaluate these expressions? Yes, and this is also done recursively, using postorder traversal very similar to the one in Part 3 of the lab. Here is the Tree method to do it:

```
def computeValue(self):
    childValues = [x.computeValue() for x in self.children]
    return value(self.data, childvalues)
```

Here is how you would use it:

```
expr1 = ('+',('sum',5,3,9,4),17) # sum(5,3,9,4) + 17 = 38
exprTree1 = Tree(expr1)
s = exprTree1.printExpr()
x = exprTree1.computeValue()
print("The value of expression", s, "is", x)
```

At the heart of `computeValue` method is a `value` function which is not recursive. It takes two arguments: the node's `data`, and the `values` of all the child expressions.

We have written its skeleton for you, which currently works for expressions with `+` and `sum` only.

```
def value (data, values):
    if (data is a number and values is an empty list)  # this is a leaf
```

```
        return float(data)
    else if (data == '+')
        return values[0] + values[1]
    else if (data == 'sum')
        return sum(values)
```

Please complete the value function, by adding `-`, `*`, `/`, `min`, `max`, `avg`.

Hint: Consider using the `eval()` function in Python that takes in a string as an argument and treats it as a piece of code. This means you can pass a string to `eval()` and the string is evaluated to a function name.

Example:

```
str = 'sum'
val = [9, 4, 3]
x = eval(str)(val)
print(x)              #prints 16
```

Part 5: Generic Tree Computations

In part 4, we could have passed the `value` function as an argument to `computeValue`. Let's rewrite our code to do that. We get a new `compute` function that takes such a function as argument. Here is how it will look:

```
def compute(self, evalFunc):
    childValues = [x.compute(evalFunc) for x in self.children]
    return evalFunc(self.data, childvalues)
```

And here is how it will work:

```
expr1 = ('+',('sum',5,3,9,4),17) # sum(5,3,9,4) + 17 = 38
exprTree1 = Tree(expr1)
s = exprTree1.printExpr()
x = exprTree1.compute(value)  # same value function as in part 4
print("The value of expression", s, "is", x)
```

Compare this code with the corresponding code in part 4. Make sure it runs for you.

Passing in a function makes `compute` generic. This very same `compute` method can compute other interesting information about the tree, if we pass it a different function instead of value. Here are 3 possible suggested functions:

1. `height` = 0 for leaves, 1 + max(child heights) for internal nodes
2. `size` = 1 for leaves, sum(child sizes) for internal nodes
3. `spaceusage` = data[1] for leaves , sum(child spaceusage) for internal nodes

Note: The height of a node is the number of edges on the longest path from the node to a leaf. A leaf node will have a height of 0. Size of the tree is number of nodes in the tree.

Please write the last of these functions, `spaceusage`. Remember, it will have the same structure as the `value` function.

```
def spaceusage (data, values):
    if this is a leaf
             ...
            return ...
       else
           ....
           return ...
```

You can now use this function to compute the space usage of a directory tree:

```
fileDir = ..............  # same fileDir as in part 3
fileTree1 = Tree(fileDir)
    x = fileTree1.compute(spaceusage)  # x should be 67, same as in part 3
```

For any file directory, you should get the amount of space the directory uses -- exactly the same number as you would obtain in Part 3 of the lab!