CS100 Tutorial 11

Hands-on small projects

Task 1

Implement a calculus for expressions

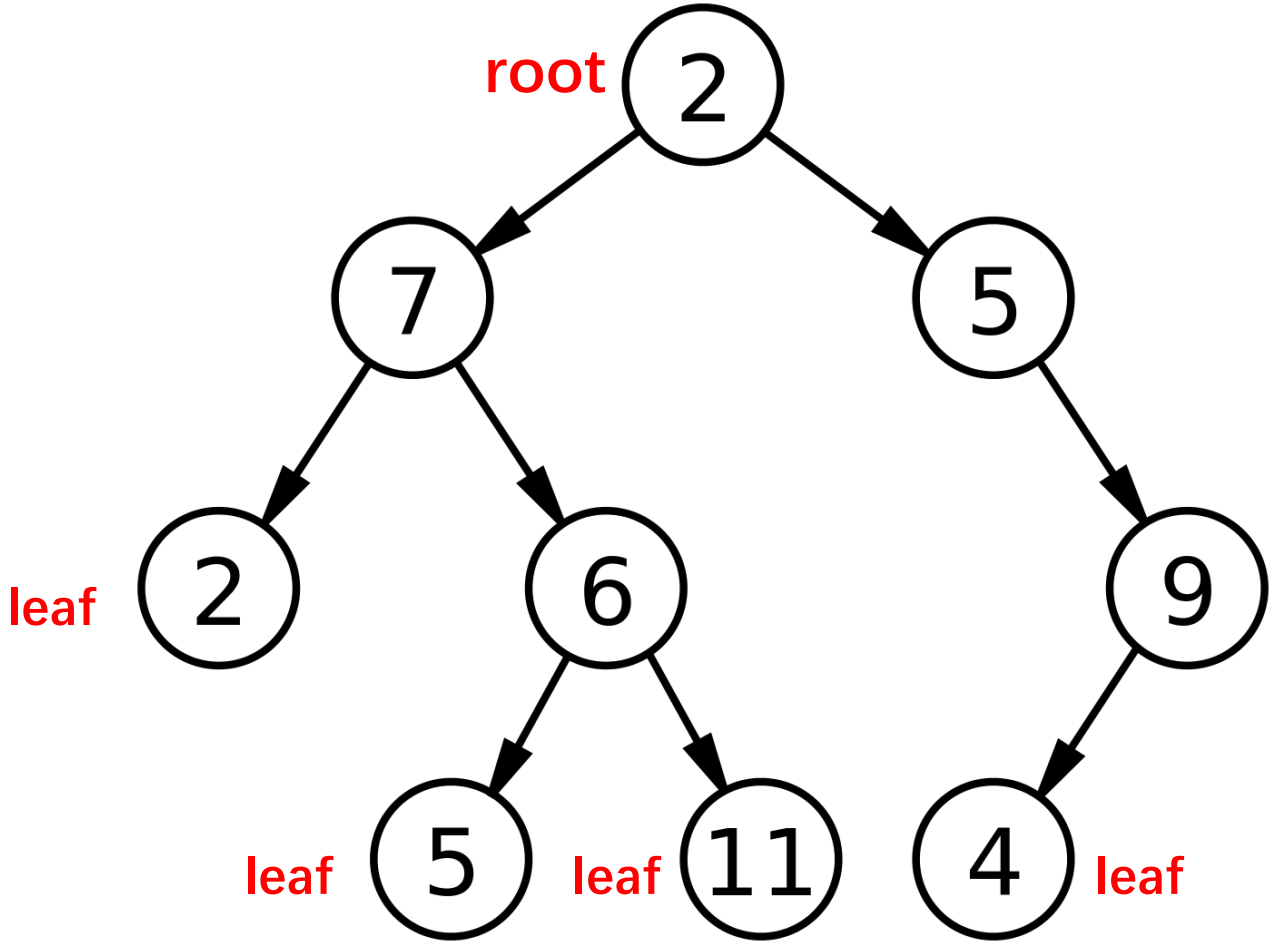
- Binary tree
- Stack
- Tokenizer
- Expressions:
  - Operators ‘+’, ’-’, ’*’, ’/’ , ‘(‘ and ‘)’
  - Constants: integer and float

Binary Tree

A (binary) tree T=(N,E) is a non-linear abstract :

- N: is a finite set of nodes
- E: is an edge function mapping each node to a list consisting of at most two nodes
  - the first node in the list E(n) is called left child of n (if it exists)
  - the second node in the list E(n) is called right child of n (if it exists)
- A node n is called root, if E(n') does not contain n for all nodes n' in N
- A node n is called leaf, if E(n) is an empty list

Example: A binary tree



Tree Representation

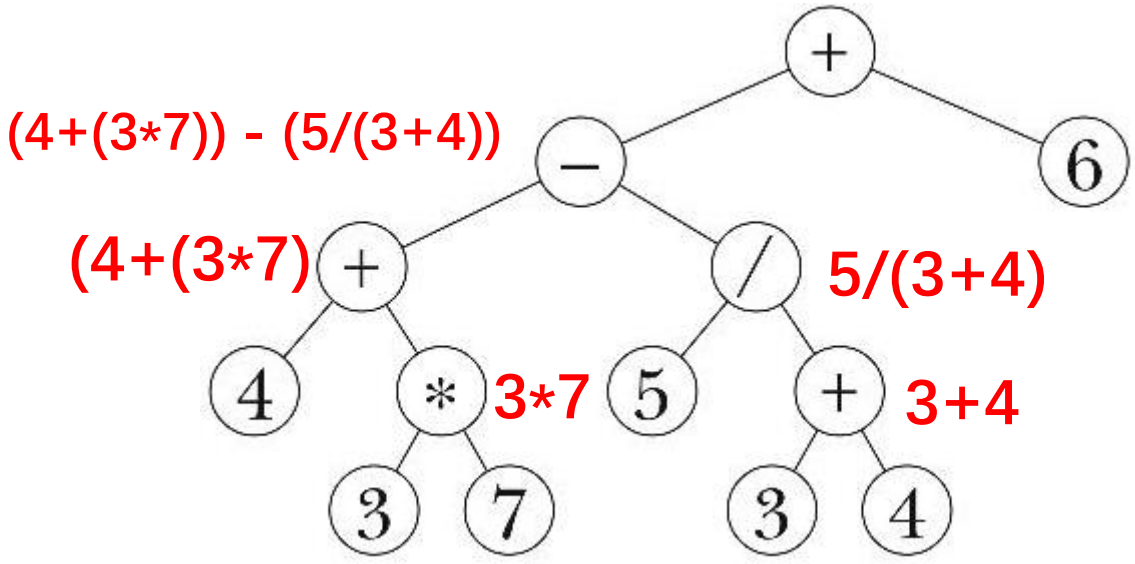
- Tree class
  - define a class called Tree
  - the class has an instance attribute data (i.e., list or tuple) for representing data associated to the node
  - the class has two instance attributes called left and right for storing the left child and right child (that are tree instance objects)

```
class Tree():
    def __init__(self, data, leftChild = None, rightChild =None):
        self._data=data
        self._leftChild = leftChild
        self._rightChild = rightChild
```

Representing an Expression as a Tree

An expression can be represented by a tree, where

- **leaves denote operands**
- **other nodes denote operators**
- **each subtree denotes an subexpression**



Tree for the expression: ((4+(3*7)) - (5/(3+4))) +6

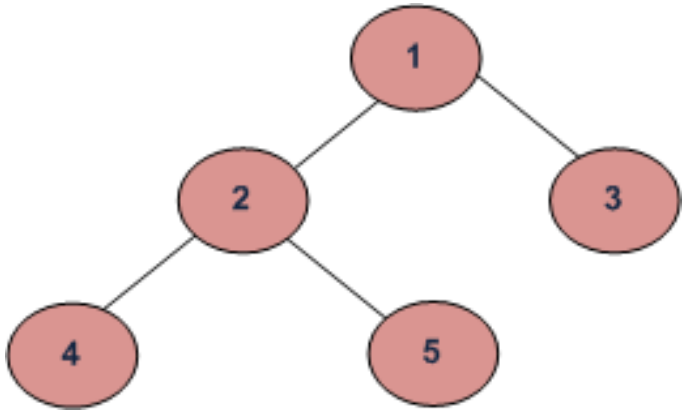
Representing an Expression as a Tree

```
class Tree():
    def __init__(self, data, leftChild = None, rightChild =None):
        self._data=data
        self._leftChild = leftChild
        self._rightChild = rightChild

# create a tree for 4 + (3 * 7)

t3 = Tree(3)
t7 = Tree(7)
t37 = Tree("*", t3, t7)
t4 = Tree(4)
t437 = Tree("+", t4, t37)
```

Tree Traversal



- Inorder: visit left substree, then root, later right subtree

  4 2 5 1 3
- Preorder: visit root, then, left substree, later right subtree

  1 2 4 5 3
- Postorder: visit left substree, then right subtree, later root

  4 5 2 3 1

Tree Traversal

```
class Tree():
    ……
    def inOrder(self):
        if self._leftChild!=None:
            self._leftChild.inOrder()
        print(self._data)
        if self._rightChild!=None:
            self._rightChild.inOrder()

    def preorder(self):
        print(self._data)
        if self._leftChild!=None:
            self._leftChild.inOrder()
        if self._rightChild!=None:
            self._rightChild.inOrder()
```

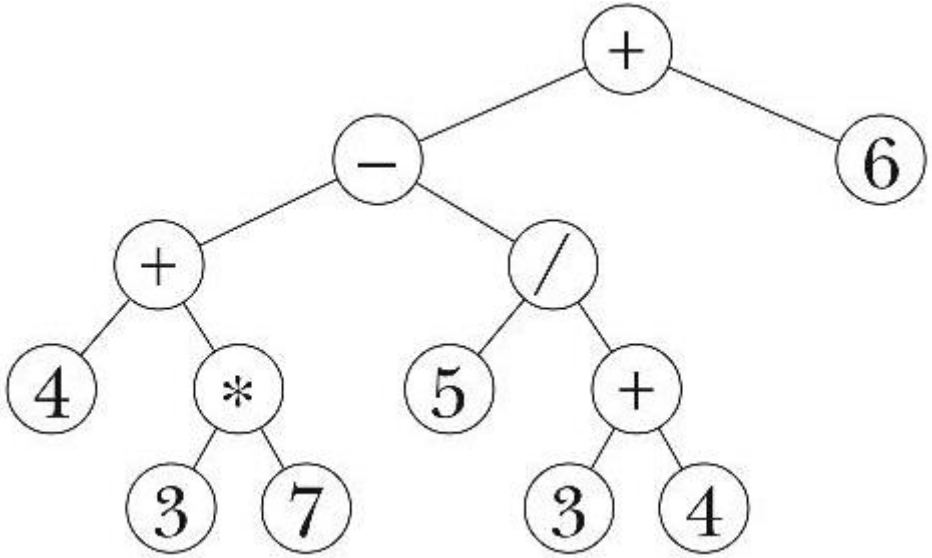
Tree Traversal

```
class Tree():
    ……
    def preorder(self):
        print(self._data)
        if self._leftChild!=None:
            self._leftChild.inOrder()
        if self._rightChild!=None:
            self._rightChild.inOrder()

    def postorder(self):
        if self._leftChild!=None:
            self._leftChild.inOrder()
        if self._rightChild!=None:
            self._rightChild.inOrder()
        print(self._data)
```

Computing an expression via Postorder Traversal

1. Compute 4
2. Compute 3
3. Compute 7
4. Compute 3*7
5. Compute 4+(3*7)
6. Compute 5
7. Compute 3
8. Compute 4
9. Compute 3+4
10. Compute 5/(3+4)
11. Compute (4+(3*7)) – (5/(3+4))
12. Compute 6
13. Compute ((4+(3*7)) – (5/(3+4)))+6



Stack

- Stack: is an abstract data structure, served as a collection of elements
  - a list-like object with two ends called bottom and top
  - Last in first out (LIFO)
  - basic operations: push, pop, peek on top of stack
- Stack can be implemented using list type s
  - index 0 is the bottom, and index -1 is the top
  - empty stack: [ ]
  - push: s.append(e)
  - pop: s.pop()
  - top: s[-1]

Tokenize

tokenize module provides a lexical scanner for Python source code, implemented in Python

- import tokenize first
- tokenize.tokenize(readline): readline, which must be a callable object which provides the same interface as the io.IOBase.readline() method of file objects. It can be obtained from a string via

  BytesIO(s.encode('utf-8')).readline

- return a 5-tuples:
  - the token type;
  - the token string;
  - a 2-tuple (srow, scol) of the row and column where the token begins;
  - a 2-tuple (erow, ecol) of ints where the token ends in the source;
  - the line on which the token was found

```
import tokenize
from io import BytesIO
import token

s = "(4.2+3)*2"
tokens = tokenize.tokenize(BytesIO(s.encode('utf-8')).readline)
for tokType, tokVal, _, _, _ in tokens:
    if tokType == token.OP: print("OP: ",tokVal)
    if tokType ==token.NUMBER: print("NUMBER: ",tokVal)
```

Output:

```
OP:  (
NUMBER:  4.2
OP:  +
NUMBER:  3
OP:  )
OP:  *
NUMBER:  2
```

Expression to Tree

Input: an expression create two empty stacks: opStack and exprStack

1. Create a new string s = "("+input +")", then tokenize s
2. For each valid token t of an expression from left to right:
   - 2.1 If t is "(", push it onto the opStack.
   - 2.2 If t is ")":
     - 2.2.1 while opStack.top != "(":
       - a) op = opStack.pop()
       - b) right = exprStack .pop()
       - c) left = exprStack.pop()
       - d) expr = Tree(op, left, right)
       - e) push expr onto exprStack
     - 2.2.2 Remove ")" from opStack.
   - 2.3 If t is an operand, push Expr(Tree(t)) onto exprStack
   - 2.4 If t is an operator:
     - 2.4.1 While opStack.top has equal or higher precedence than t:
       Do the same as in a)—e)
     - 2.4.2 Push t onto opStack
3. opStack.top is the tree of the input expression

Requirement

Using binary tree, stack, tokenizer to implement a module calc.py for computing values of expressions.

Testcases:

python calc.py “3+4”

7

python calc.py “(3+4)/2”

3.5