Musterlösung zur Übung 5 — Abgabe 29.05.2019

Aufgabe 1 – Binärbäume — 16 Punkte

a) Der Code von der SearchTree-Klasse ist in dem File `searchtree.py` gegeben.

b) Ein Algorithmus zur Bestimmung der Tiefe des Baumes könnte sein:

- Test, ob aktueller Knoten ein leerer ist („None“)
- Falls ja, abrechen und 0 zurückgeben
- Ansonsten rekursives Traversieren zunächst des linken,
- dann des rechten Teilbaums
- Bestimmen des Maximums der bisherigen Tiefen der beiden Teilbäume
- Inkrementieren des Rückgabewertes (bisherige Tiefe)
- in jedem rekursiven Schritt

Der Code der Funktion `SearchTree.depth()` im `searchtree.py` spiegelt diesen Algorithmus wider.

c) Sortiert man die einzufügenden Daten in aufsteigender Reihenfolge vor, so hat der nach dem Einfügen aller Daten entstandene Binäre Suchbaum die geringste Tiefe, wenn man das Divide-and-Conquer-Prinzip anwendet. Das heißt, man wählt zunächst das mittlere Element als Wurzel und teilt die Liste der Daten an dieser Elementposition in zwei Teillisten (TL). Das mittlere Element der linken TL wird das linke Kind, das mittlere Element der rechten TL wird zum rechten Kind der Wurzel. Das Verfahren wiederholt man solange, bis alle Daten eingefügt sind.

Also:

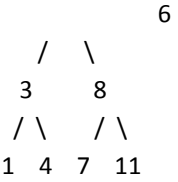
- Nehme mittleres Element der Vorsortierten Daten und füge es in den BST ein
- Betrachte linke Hälfte der Daten und wende Algorithmus rekursiv an
- Betrachte rechte Hälfte der Daten und wende Algorithmus rekursiv an

Beispiel:

Daten (sort.): 1 3 4 6 7 8 11

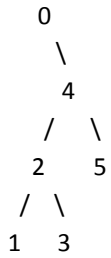
Einfügereihenfolge: 6 3 1 4 8 7 11

BST:



d) Die Aussage stimmt nicht, wenn man den Algorithmus TreeRemove() aus der Vorlesung zugrundelegt.

Gegenbeispiel:



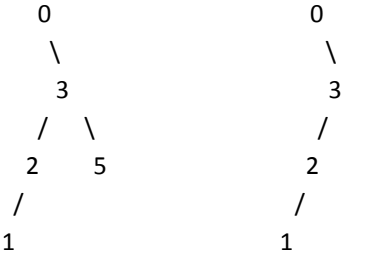
Lösche...

Fall 1:

... erst 5, dann 4.

Fall 2:

...oder erst 4, dann 5.



q.e.d.

Ändert man treeRemove() derart, dass in Fall 1 der Knoten (4) nicht durch Anhängen des linken Teilbaumes (2) entsteht, sondern durch seinen Vorgänger (3) ersetzt wird, so sind die beiden Ergebnisbäume zunächst gleich. Allerdings kann man dann andere Gegenbeispiele finden. Man sieht also, dass das Ergebnis maßgeblich von der Art der Implementierung des Löschalgorithmus abhängt.

Aufgabe 2 – Taschenrechner 24 Punkte

a) Einen Algorithmus, der einen arithmetischen Ausdruck in einen (Binär-)Baum umwandelt, bezeichnet man als *Parser*, und der resultierende Baum ist der *Parse-Baum*. Wir definieren zuerst die Knotenklassen für den Parse-Baum:

```
# Wir definieren zuerst die beiden Knotenklassen fuer den Baum:
# Die Blattknoten des Parse-Baums repraesentieren Zahlenwerte.
class Number:
    def __init__(self, n):
        self.number = n

        # evaluate() wertet den Ausdruck aus, den der Knoten repraesentiert.
        # Bei Blattknoten wird einfach der String in die entsprechende Zahl
        # umgewandelt.
    def evaluate(self):
        return float(self.number)

        # zum Ausdrucken als String
    def __repr__(self):
        return self.number

        # zum Ausdrucken als Baum
    def printTree(self, indent=''):
        return '(' + self.number + ')\n'

    # Die inneren Knoten des Parse-Baumes repaesentieren Operatoren
    # mit ihren linken und rechten Operanden.
class Operator:
    def __init__(self, left, operator, right):
        if not operator in '+-*/':
            raise SyntaxError('unknown operator')
        self.left = left
        self.operator = operator
        self.right = right

        # Bei inneren Knoten wertet evaluate() zuerst den linken und rechten
        # Operanden rekursiv aus, dann werden die Ergebnisse mit dem
        # gespeicherten Operator verknuepft.
    def evaluate(self):
        left, right = self.left.evaluate(), self.right.evaluate()
        if self.operator == '+':
            return left + right
        if self.operator == '-':
            return left - right
        if self.operator == '*':
            return left * right
        if self.operator == '/':
            return left / right

        # zum Ausdrucken als String
    def __repr__(self):
        return '(' + repr(self.left) + self.operator + repr(self.right) + ')'

        # zum Ausdrucken als Baum
    def printTree(self, indent=''):
       return '(' + self.operator + ') -- ' + self.right.printTree(indent + ' |
') + \
              indent + ' |\n' + \
              indent + self.left.printTree(indent)
```

Die Funktion `parse(t)` erzeugt den Parse-Baum und wird immer wieder rekursiv für Teilausdrücke aufgerufen. Am Ende gibt die Funktion die Wurzel des resultierenden Gesamtbaums zurück. **Lösung 1** implementiert dies explizit, ohne Verwendung einer Grammatik oder weiterer Werkzeuge. Die Lösung beruht auf folgender Beobachtung: Operationen, die tiefer im Baum sind, werden zuerst ausgeführt. Deshalb müssen sich Operationen mit höherer Priorität tiefer im Baum befinden. Bei Operationen gleicher Priorität müssen die vorderen tiefer als die hinteren im Baum sein (Links-Assoziativität). Durch die Rekursion landet das, was man als erstes verarbeitet, jedoch oben im Baum => wir müssen den String von hinten nach vorn parsen und Strichrechnung vor Punktrechnung verarbeiten (also genau gegen die Intuition).

```
    # An Position i ist eine oeffnende Klammer, suche die zugehoerige
    # schliessende Klammer durch Zaehlen von oeffnenden und
    # schliessenden Paaren.
def findClosingBracket(t, i):
    count = 1
    for k in range(i+1, len(t)):
```

```
        if t[k] == '(':
            count += 1
        elif t[k] == ')':
            count -= 1
        if count == 0:
            return k
    # Wenn wir hier landen, gab es keine passende schliessende Klammer.
    raise SyntaxError('invalid expression')

    # Teste ob rechts von Position i ein hoeherwertiger Operator steht
    # als links.
def rightPriorityIsHigher(t, i):
    if i == len(t)-1:
        # rechts ist nichts => die Bedingung ist nie erfuellt
        return False

    if i == 0:
        # links ist nichts => die Bedingung ist immer erfuellt
        return True

    # Die Bedingung ist erfuellt, wenn links Strichrechnung und rechts
    # Punktrechnung verwendet wird.
    return t[i-1] in '+-' and t[i+1] in '*/'

    # Wandle den Ausdruck t in einen Parse-Baum um, gegebenenfalls rekursiv.
    # Die Funktion gibt die Wurzel des resultierenden Baumes zurueck.
def parse(t):
    # Vorverarbeitung: Erzeugen einer Token-Liste
    if type(t) is str:
        # Entferne eventuelle Leerzeichen
        t = t.replace(' ', '')

        # Wandle den String in ein Array von 'Tokens' um.
        # Die Uebungsaufgabe ist so gestellt, dass die Tokens gerade den
        # einzelnen Zeichen des Strings entsprechen.
        # (Bei realen Programmiersprachen muss man zusammengesetzte
        # Zahlen (z.B. 324), Operatoren aus mehreren Zeichen (z.B. **) etc.
        # jeweils zu einem Token zusammenfassen. Dies die Aufgabe des 'Lexers',
        # der stets dem eigentlichen Parser vorgeschaltet ist.)
        t = [k for k in t]

    k = 0
    while True:
        # Wenn k ueber das Ende von t hinauslaeuft, kann t kein gueltiger
        # Ausdruck sein.
        if k >= len(t):
            raise SyntaxError('invalid expression')

        # Wenn t[k] noch nicht verarbeitet wurde ...
        if type(t[k]) is str:
            # ... generiere einen Blattknoten, wenn t[k] eine Zahl ist, ...
            if t[k] in '0123456789':
                t[k] = Number(t[k])
            # ... oder parse den Klammerausdruck, wenn t[k] eine oeffnende Klammer
ist ...
            elif t[k] == '(':
                # Finde das Ende des Klammerausdrucks.
                end = findClosingBracket(t, k)
                # Parse den Unterausdruck zwischen den Klammern rekursiv.
                # sub ist die Wurzel des resultierenden Teilbaums.
                sub = parse(t[k+1:end])
                # Ersetze den Klammerausdruck in t durch das Ergebnis des Parsings.
                t[k:end+1] = [ sub ]
            # ... oder signalisiere einen Syntaxfehler. (Da k immer gerade
            # ist, enthaelt t[k] entweder eine Zahl oder die Wurzel eines
            # bereits fertigen Teilbaumes oder es ist der Beginn eines
            # Klammerausdrucks. Alles andere ist deshalb ein Fehler.)
            else:
                raise SyntaxError('invalid expression')

        # Wenn die Prioritaet des rechten Operators hoeher ist, kuemmern wir uns
        # zuerst um den rechten Teilausdruck ...
        if rightPriorityIsHigher(t, k):
            k += 2
```

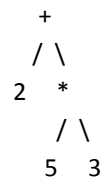
```
                # ... andernfalls, wenn es einen linken Teilausdruck gibt, erzeugen wir einen
                # Operatorknoten im Baum und ersetzen den linken Teilausdruck durch diesen
                # Knoten. Die Verarbeitungsreihenfolge garantiert, dass t[k-2] und t[k]
                # bereits geparsed wurden, also fertige Teilbaeume enthalten.
                elif k >= 2:
                    t[k-2:k+1] = [ Operator(t[k-2], t[k-1], t[k]) ]
                    k -= 2

                # Wenn t keine Operatoren und Klammern mehr enthaelt, sind wir fertig.
                # => t[0] ist jetzt die Wurzel des Parse-Baumes.
                if len(t) == 1:
                    return t[0]
```

Für Fortgeschrittene: **Lösung 2** nutzt das pyparsing Modul (http://pyparsing.wikispaces.com/), um eine formale Grammatik zu definieren. Dieses Vorgehen lohnt sich spätestens dann, wenn der Taschenrechner mehr Funktionalität unterstützen soll als in dieser Aufgabe:

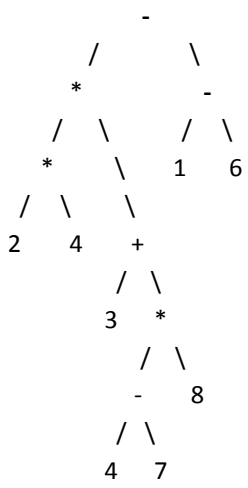
```
class PythonParser:
    def __init__(self):

        # Define helper functions to transform tokens into parse tree nodes.
        def createNumber(s, k, tokens):
            # Create a number object from the string in tokens[0].
            return Number(tokens[0])

        def createOperators(s, k, tokens):
            # Create a subtree of operator nodes.
            #
            # Even indices in 'tokens' contain operands, odd ones are operator
            # symbols. Due to the design of the grammar below, all operators
            # in 'tokens' have the same precedence, so we evaluate them
            # left-to-right.
            res = tokens[0]
            for k in range(1, len(tokens), 2):
                res = Operator(res, tokens[k], tokens[k+1])
            return res

        def removeBrackets(s, k, tokens):
            # Drop tokens[0] and tokens[2] (the opening and closing brackets).
            # This is possible because brackets control the order of
            # expression evaluation by enforcing a particular order of
            # parsing. At this point, construction of the parse tree for
            # the sub-expression between the brackets has been completed,
            # and the brackets are therefore no longer needed.
            return tokens[1]

        # Import the parsing library.
        from pyparsing import Forward, Word, ZeroOrMore

        # Define basic entities: numbers and operator symbols.
        number    = Word('0123456789', max=1).setParseAction(createNumber)
        muldiv    = Word('*') | Word('/')
        plusminus = Word('+') | Word('-')

        # Define the grammar rules and the associated parsing actions to be
        # executed when a rule matches some substring of the input.
        #
        # 'expression' is the toplevel entity. The hierarchical definition of
        # bracket_expr, factor, term, and expression ensures the correct
        # precedence: brackets have higher binding strength than * and /,
        # which in turn have higher binding strength than + and -.
        expression   =  Forward()
        bracket_expr =  ( '(' + expression + ')' ).setParseAction(removeBrackets)
        factor       =  bracket_expr | number
        term         =  (factor + ZeroOrMore(muldiv +
factor)).setParseAction(createOperators)
        expression   << (term + ZeroOrMore(plusminus + term)).setParseAction(createOperators)

        self.parser = expression

    # Make the parser class callable, so that it can be used like a function.
    def __call__(self, s):
        # Parse the input (res[0] is the root node of the resulting parse tree).
        res = self.parser.parseString(s)
        return res[0]
```

b) Parse-Bäume...

für: 2+5*3



2*4*(3+(4-7)*8)-(1-6)



Dasselbe, ausgegeben mit der `printTree()`-Methode der Klassen `Number` und `Operator` (siehe a):

```
>>> print parse('2+5*3').printTree()

(+) -- (*) -- (3)
 |      |
 |     (5)
 |
(2)

>>> print parse('2*4*(3+(4-7)*8)-(1-6)').printTree()

(-) -- (-) -- (6)
 |      |
 |     (1)
 |
(*) -- (+) -- (*) -- (8)
 |      |      |
 |      |     (-) -- (7)
 |      |      |
 |      |     (4)
 |      |
 |     (3)
 |
(*) -- (4)
 |
(2)
```

c) Die Funktion `evaluateTree()` ruft einfach die Funktion `evaluate()` in den Klassen `Operator` bzw. `Number` auf (siehe Teilaufgabe a):

```
def evaluateTree(root):
    return root.evaluate()
```

d) Der folgende unittest ist für die Lösungsvariante 1 geschrieben. Ersetzt man `parse()` durch `pyparse()`, ist er jedoch auch für die Lösungsvariante 2 gültig.

```
import unittest

def evaluateTree(root):
        return root.evaluate()


class testCalc(unittest.TestCase):

    """
    A test class for the calc(Calculator) module.
    """
    def setUp(self):
        """
        set up data used in the tests.
        """
        self.mathStrings = [
            ("1+1", 2),
            ("1-1", 0),
            ("1+1+1", 3),
            ("1+1*1", 2),
            ("2+2*2/2", 4),
            ("(2*3)+(4*5)", 26),
            ("2*9/4*(1+(4*5)/3-(2+0))-1+5", 29.5),
            ("1/5", 0.2),
            ("0", 0),
            ("2*4*(3+(4-7)*8)-(1-6)", -163),
            ("2*3+4*5", 26),
            ("(2-5)", -3),
            ("(((1+5*3+5)+7*8)*(1+3*2)+9+(1*2))+5", 555)]

    def test_calc(self):
        """
        test calculator using the expressions defined in the setUp function
        """
        for test in self.mathStrings:
            tree = parse(test[0])
            res = evaluateTree(tree)
            self.assertEqual(res, test[1],
                "Error when evaluating %s, expected %f but got %f" % \
                                        (test[0],test[1], res))

if __name__ == '__main__':
    unittest.main()
```