Übung 5 Abgabe 15.6.2020

Aufgabe 0 – Debuggen von Übung 4

Studieren Sie die Musterlösung zu Übung 4 auf Moodle und vergleichen Sie mit Ihrer eigenen Lösung. Fügen Sie Kommentare in Ihre ursprüngliche Abgabe ein, welche Fehler Sie gemacht haben und wie die Lösung korrigiert werden muss. Kommentieren Sie auch Dinge, die bei Ihnen zwar korrekt, aber umständlich oder ineffizient gelöst sind. Je besser Ihre Kommentare sind, desto weniger Punkte werden abgezogen. Insbesondere werden keine Punkte für Folgefehler abgezogen, wenn Sie die Grundursache eines Problems identifiziert und korrigiert haben. In Quellcode-Dateien kennzeichnen Sie Ihre Kommentare mit

```
# comment: <Ihr Kommentar>
```

In PDF-Dateien verwenden Sie eine geeignete Farbe. Wichtig ist, dass die Tutoren die Kommentare leicht von der ursprünglichen Abgabe unterscheiden können.

Benennen Sie die Dateien mit kommentierten Lösungen in "`kommentiert_<original-filename>`" um und geben Sie sie im zip-File für Übung 5 ab.

Aufgabe 1 – Binäre Suchbäume 16 Punkte

a) In der Vorlesung haben wir Suchbäume behandelt, die eine effiziente Suche nach Schlüsseln erlauben. In dieser Übung sollen Sie das Gelernte in einer Datenstruktur `SearchTree` so implementieren, dass jeder Knoten `(key, value)`-Paare abspeichert, wie man es auch von Python's `dict`-Klasse kennt: *6 Punkte*

```
class Node:
    def __init__(self, key, value):
        self._key = key
        self._value = value
        self._left = self._right = None
```

Das Grundgerüst der Klasse `SearchTree` kapselt die Funktionen aus der Vorlesung wie folgt:

```
class SearchTree:
    def __init__(self):                 # implements 'tree = SearchTree()'
        self._root = None
        self._size = 0

    def __len__(self):                  # implements 'len(tree)'
        return self._size

    def __getitem__(self, key):         # implements 'value = tree[key]'
        ... # your code here

    def __setitem__(self, key, value):  # implements 'tree[key] = value'
        ... # your code here

    def __delitem__(self, key):         # implements 'del tree[key] '
        ... # your code here

    @staticmethod
    def _tree_find(node, key):          # internal implementation
        ... # your code here
```

```
    @staticmethod
    def _tree_insert(node, key, value):  # internal implementation
        ... # your code here

    @staticmethod
    def _tree_remove(node, key):         # internal implementation
        ... # your code here
```

Vervollständigen Sie das File `searchtree.py` auf Moodle und fügen Sie geeignete `pytest`-Tests hinzu. Wenn `tree` ein Objekt vom Typ `SearchTree` ist, soll die Semantik der Aufrufe wie folgt realisiert werden (das entspricht dem Verhalten von `dict`):

- `tree[key] = value`: Fügt den gegebenen Wert `value` unter dem Schlüssel `key` ein. Falls `key` schon vorhanden war, soll der zuvor gespeicherte Wert überschrieben werden. Andernfalls soll ein neuer `Node` so eingefügt werden, dass die Suchbaumbedingung erhalten bleibt. Beachten Sie, dass `_size` nur im zweiten Fall aktualisiert werden darf.
- `value = tree[key]`: Gibt den Wert zu `key` zurück oder löst einen `KeyError` aus, der den Schlüssel angibt, wenn `key` nicht vorhanden ist.
- `del tree[key]`: Löscht den `Node`, der den gegebenen Schlüssel enthält, wobei die Suchbaumbedingung erhalten bleibt. Falls der Schlüssel nicht vorhanden ist, soll eine `KeyError`-Exception ausgelöst werden, deren Fehlermeldung den Schlüssel angibt.

b) Entwickeln Sie einen Algorithmus, der die Tiefe des Baumes (den maximalen Abstand von der Wurzel zu einem Blatt) bestimmt und implementieren Sie ihn als Methode, so dass `depth=tree.depth()` die Tiefe zurückgibt. Fügen Sie Tests für diese Funktion hinzu. *4 Punkte*

c) Angenommen, Sie können die Schlüssel in einer selbstgewählten Reihenfolge einfügen. Welche Reihenfolge wählen Sie, damit der Baum nach dem Einfügen eine möglichst geringe Tiefe hat? *3 Punkte*

d) Beweisen oder widerlegen Sie folgende Aussage: Wenn aus einem Suchbaum erst der Schlüssel X und danach der Schlüssel Y entfernt wird, entsteht der gleiche Baum wie bei umgekehrter Reihenfolge. *3 Punkte*

Aufgabe 2 – Taschenrechner 24 Punkte

Binärbäume eignen sich auch, um mathematische Ausdrücke auszuwerten, die als Zeichenketten der Form "2+5*3" oder "2*4*(3+(4-7)*8)-(1-6)" gegeben sind. Man bezeichnet solche Bäume als *Syntaxbäume* (https://de.wikipedia.org/wiki/Abstrakter_Syntaxbaum).
Die Ausdrücke können die arithmetischen Operationen +, -, *, / mit den üblichen Rechenregeln enthalten (Klammern haben die höchste Priorität, Punktrechnung geht vor Strichrechnung). Zahlen sollen der Einfachheit halber immer einstellig und positiv sein. Variablen und Funktionen kommen nicht vor. Operationen mit gleicher Priorität werden von links nach rechts ausgewertet (sogenannte *Links-Assoziativität*), damit wie gewohnt 5-2+3 = (5-2)+3 = 6 gilt (und nicht 5-(2+3) = 0, was bei Auswertung von rechts herauskäme). Steht jedoch rechts von einer Zahl oder einem Klammerausdruck eine Operation mit höherer Priorität als links, wird der rechte Operator zuerst ausgewertet. Trifft man auf eine öffnende Klammer, muss man den Substring bis zur zugehörigen schließenden Klammer suchen und die Auswertung rekursiv auf diesen Substring anwenden. Dadurch ergibt sich ein Binärbaum.

a) Entwickeln Sie einen Algorithmus, der den zu einem Ausdruck korrespondierenden Binärbaum aufbaut, wobei jeder innere Knoten einen Operator (+, -, *, /) repräsentiert, jeder Unterbaum einen linken bzw. rechten Operanden, und jedes Blatt eine Zahl. Der Baum soll dann durch eine Funktion `parse(s)` erstellt werden, an die der Ausdruck als String übergeben wird und die den Wurzelknoten des Baums zurückgibt (die Verwendung der Python-Funktionen `eval()` bzw. `exec` ist dabei *nicht* erlaubt). Implementieren Sie diese Funktion im File `calculator.py` und erklären Sie in Kommentaren, wie der Algorithmus vorgeht. *14 Punkte*

Hinweis: Implementieren Sie zunächst zwei Klassen `Number` und `Operator`, die als Blattknoten bzw. innere Knoten des Syntaxbaums dienen. `Number`-Objekte speichern also eine Zahl, und `Operator`-Objekte ein Operatorsymbol sowie den linken und rechten Operanden (Unterbaum).

b) Skizzieren Sie die Bäume, die sich für die Ausdrücke "2+5*3" und "2*4*(3+(4-7)*8)-(1-6)" ergeben. *3 Punkte*

c) Implementieren Sie eine Funktion `evaluate_tree(root)`, um einen Ausdruck mit Hilfe des in a) erstellten Baums auszurechnen und geben Sie die Implementation im File `calculator.py` ab. *3 Punkte*

d) Schreiben Sie Tests für Ihr Verfahren (ebenfalls in `calculator.py`). Beachten Sie dabei die Hinweise zum Erstellen guter Tests im Kapitel "Korrektheit" des Skripts. Beispielsweise müssen Sie verschiedene Varianten der Operator-Präzedenz und Klammerung testen. *4 Punkte*

Bitte laden Sie Ihre Lösung bis zum 15.6.2020 um 11:00 Uhr auf Moodle hoch.