TREECHECK – PROTOKOLL

Überlegen Sie sich vor der Implementierung, wie die rekursiven Funktionen aufgebaut werden müssen (Abbruchbedingung, Parameter, Rückgabewert, …) und protokollieren Sie Ihre Überlegungen.
Schätzen Sie weiters den Aufwand der Funktionen mittels O-Notation in Abhängigkeit der Anzahl N der Integer-Werte in der Eingabedatei.

Rekursive Funktionen

Insert – Funktion:

```
def insert(self, val):
    if self.val is None:
        self.val = val
        return self
    elif self.val == val:
        return self
    elif self.val > val:
        if self.left is None:
            self.left = TreeNode(val)
            return self.left
        else:
            return self.left.insert(val)
    else:
        if self.right is None:
            self.right = TreeNode(val)
            return self.right
        else:
            return self.right.insert(val)
```

Die Funktion **insert()** verwendet Rekursion, um einen Wert in einen binären Suchbaum einzufügen. In dieser Funktion wird der Wert mit dem aktuellen Knotenwert verglichen und die Rekursion wird entweder auf dem linken oder rechten Kindknoten fortgesetzt. Rekursion ist eine natürliche Wahl für die Implementierung von Baumoperationen, da jeder Knoten als ein neuer Baum, mit linken und rechten Kindknoten, gesehen werden kann.

Abbruchbedingung: Die Rekursion endet, wenn ein leerer Knoten (None) erreicht wird, an dem der neue Wert eingefügt wird.

Parameter: Die Funktion nimmt einen Parameter **val** entgegen, der den einzufügenden Wert repräsentiert, und verwendet **self** als Referenz auf den aktuellen Knoten, um den entsprechenden Knoten im Baum zu erstellen.

Return-Werte: Die Rückgabewerte der Funktion sind entweder der neu eingefügte oder der bestehenden Knoten (keine Duplikate).

Aufwandsschätzung:

Die Aufwandschätzung für die **insert-Funktion** ist **O(H), wobei H die Höhe des Baums ist**. Dies liegt daran, dass die Funktion bis zum Blattknoten traversiert, um den neuen Knoten einzufügen, und die maximale Anzahl von Ebenen im Baum, die traversiert werden müssen, um den neuen Knoten einzufügen, der Höhe des Baums entspricht.

(Da ein **ausbalancierter AVL Baum** eine **maximale Höhe von O(log N)** hat, wobei N die Anzahl der Knoten im Baum ist, ist die Aufwandschätzung für die insert-Funktion bei Verwendung eines ausbalancierten Baums O(log N).)

Traversieren des Baumes (Präorder):

```
def traverse_and_check_avl(node, stats):
    if node is None:
        return True

    left_avl = traverse_and_check_avl(node.left, stats)
    right_avl = traverse_and_check_avl(node.right, stats)
    left_height = height(node.left)
    right_height = height(node.right)
    balance_factor = abs(left_height - right_height)
    is_avl = True

       if balance_factor > 1:
        print(bcolors.FAIL + f"bal({node.val}) = {balance_factor} (AVL violation!)"
+ bcolors.RESET)
        is_avl = False

    else:
        print(bcolors.GREEN + f"bal({node.val}) = {balance_factor}" +
bcolors.RESET)

    stats['min'] = min(stats['min'], node.val)
    stats['max'] = max(stats['max'], node.val)
    stats['sum'] += node.val
    stats['count'] += 1

    return is_avl and left_avl and right_avl
```

Die Funktion **traverse_and_check_avl**() verwendet Rekursion, um durch den Baum zu navigieren und die AVL-Eigenschaft für jeden Knoten zu überprüfen. Die Rekursion wird auf die linken und rechten Unterbäume angewendet, um sicherzustellen, dass der gesamte Baum die AVL-Bedingungen erfüllt. Während des rekursiven Durchlaufs werden auch Statistiken wie Min, Max, Sum und Count berechnet. Die rekursive Herangehensweise ist hier besonders geeignet, da sie es ermöglicht, den Baum auf natürliche Weise zu durchlaufen und die AVL-Bedingungen für jeden Knoten effizient zu prüfen.

Abbruchbedingung: Die Rekursion endet, wenn ein leerer Knoten (None) erreicht wird, und es wird überprüft, ob der aktuelle Knoten und seine Unterbäume die AVL-Eigenschaft erfüllen.

Parameter: Die Funktion nimmt zwei Parameter entgegen: **node**, der den aktuellen Knoten des Baumes repräsentiert, und **stats**, ein Dictionary, das die statistischen Informationen des Baumes speichert.

Return-Werte: Die bool'schen Rückgabewerte der rekursiven Funktion geben an, ob der aktuelle Knoten und seine Unterbäume die AVL-Eigenschaft erfüllen.

Aufwandsschätzung:

Die Laufzeit der **traverse_and_check_avl Funktion** beträgt **O(N), wobei N die Anzahl der Knoten im Baum ist.** Das liegt daran, dass die Funktion jeden Knoten im Baum mindestens einmal besucht, um die AVL-Eigenschaft zu überprüfen und die Statistiken zu aktualisieren. Zusätzlich wird in der Funktion die **height()-Funktion** aufgerufen, die die Höhe jedes Knotens prüft, daher ist **die Laufzeit ebenfalls O(N) und insgesamt $O(N^2)$.**

Simple Search:

```
def simple_search(node, key):
    if node is None:
        return False

    if node.val == key:
        return True

    if key < node.val:
        return simple_search(node.left, key)

    return simple_search(node.right, key)
```

Die Funktion **simple_search** verwendet Rekursion, um einen Schlüsselwert in einem binären Suchbaum zu suchen. Wenn der gesuchte Schlüsselwert kleiner als der Wert des aktuellen Knotens ist, wird die Suche rekursiv im linken Teilbaum fortgesetzt, andernfalls im rechten Teilbaum.

Abbruchbedingung: Die Rekursion terminiert, sobald der Schlüsselwert gefunden wird oder ein None-Knoten erreicht wird, was bedeutet, dass der Schlüsselwert nicht im Baum vorhanden ist.

Parameter: Es wird einerseits der Baum **node**, im dem der Knoten gesucht werde soll, übergeben. Und die zu suchende Zahl **key.**

Return-Werte: Die Funktion gibt **True** zurück, wenn der gesuchte Schlüsselwert im Baum gefunden wird, und **False**, wenn er nicht gefunden wird.

Aufwandsschätzung:

Die Laufzeit der **simple_search** Funktion beträgt **O(H), wobei H die Höhe des Baumes ist**. Das liegt daran, dass die Funktion rekursiv entlang des Baumes von der Wurzel bis zu einem Blattknoten oder bis zum gesuchten Schlüsselwert navigiert, was im schlimmsten Fall einer Tiefe von h entspricht.

Subtree – Search:

```
def check_subtree_structure(tree_node, subtree_node):
    if subtree_node is None:
        return True

    if tree_node is None:
        return False

    if tree_node.val == subtree_node.val:
        left_match = check_subtree_structure(tree_node.left,
subtree_node.left)
        right_match = check_subtree_structure(tree_node.right,
subtree_node.right)

        return left_match and right_match

    left_match = check_subtree_structure(tree_node.left, subtree_node)
    right_match = check_subtree_structure(tree_node.right, subtree_node)

    return left_match or right_match


def subtree_search(tree, subtree):
    if tree is None:
        return False
    if check_subtree_structure(tree, subtree):
        return True

    return subtree_search(tree.left, subtree) or subtree_search(tree.right,
subtree)
```

Die Funktionen **check_subtree_structure** und **subtree_search** verwenden beide Rekursion, um Teilbäume in einem Binärbaum zu durchsuchen. Die Funktion **check_subtree_structure** prüft, ob ein gegebener Teilbaum im Baum vorhanden ist, während **subtree_search** rekursiv die Funktion **check_subtree_structure** aufruft, um zu prüfen, ob der gegebene Teilbaum in einem der beiden Unterbäume des aktuellen Knotens enthalten ist. In beiden Funktionen wird der linke und rechte Teilbaum rekursiv untersucht, um den gewünschten Knoten oder Teilbaum zu finden.

Abbruchbedingung: In beiden Funktionen, **check_subtree_structure** und **subtree_search**, werden Abbruchbedingungen verwendet, um das Ende der Rekursion zu erreichen, wenn der zu durchsuchende Baum leer ist oder der gesuchte Teilbaum nicht gefunden wurde.

Parameter: Die Funktionen **check_subtree_structure** und **subtree_search** haben jeweils zwei Parameter. **tree** und **subtree**. **tree** ist der Baum, in dem nach einem Subtree gesucht werden soll, während **subtree** der Subtree ist, nach dem gesucht wird.s

Return-Werte: Beide Funktionen geben **bool'sche Werte** zurück, wenn der Subtree im Baum gefunden wurde

Aufwandsschätzung:

Für die Funktion **subtree_search** beträgt die Laufzeitschätzung **O(N * M), wobei N die Anzahl der Knoten im Baum ist und M die Anzahl der Knoten des Teilbaumes ist.** Im schlimmsten Fall sind beide Bäume ident, und somit beträgt die **Laufzeit $O(N^2)$.**