Treecheck Protokoll

Lorenz Rentenberger, Samuel Hammerschmidt

April 2024

Einfügefunktion

Die Methoden `insert(self, key)` & `_insert(self, node, key)` fügen neue Schlüssel in den Baum ein. `insert(self, key)` nimmt den neuen Schlüssel entgegen. `_insert(self, node, key)` ist die rekursive Funktion, die den neuen Schlüssel in den Baum einfügt. Hierbei wird der Schlüssel mit dem Schlüssel des aktuellen Knotens verglichen und folgendermaßen behandelt:

- Wenn der betrachtete Schlüssel leer ist, wird der neue Schlüssel an dieser Stelle eingefügt.
- Wenn der neue Schlüssel kleiner ist als der Schlüssel des aktuellen Knotens, wird der linke Nachfolger betrachtet.
- Wenn der neue Schlüssel größer ist als der Schlüssel des aktuellen Knotens, wird der rechte Nachfolger betrachtet.
- Nach erfolgreichem Einfügen wird der Baum anhand der Funktion `def balance(self, node)` balanciert.

Balance-Faktor & Höhe

Der Balance-Faktor eines Knotens wird durch die Methoden `height(self, node)` & `balance_factor(self, node)` berechnet. `height(self, node)` gibt die Höhe des Knotens zurück, sollte der Knoten leer sein, wird die Höhe -1 bestimmt. Andernfalls berechnet sich die Höhe aus der maximalen Höhe der beiden Kinder des Knotens plus 1. Der Balance-Faktor wird durch die Differenz der Höhen der beiden Kinder des Knotens berechnet, dies geschieht in `balance_factor(self, node)`.

Balancierung & Rotation

Die Methode `insert(self, key)` überprüft den Balance-Faktor des angegebenen Knotens. Ist der Balance-Faktor größer als 1, bedeutet dies, dass der Baum nach rechts geneigt ist und eine Rotation nach links wird ausgeführt. Genau umgekehrt wird eine Rotation nach rechts ausgeführt, wenn der Balance-Faktor kleiner als -1 ist. Wenn man keine Balancierung möchte kann man bei der Funktion `_insert(self,key)` anstatt `return self.balance(node) return node` verwenden.

- `rotate_right(self, node)` führt eine Rechtsrotation aus, dass linke Kind des Knotens wird der neue Wurzelknoten, während der ursprüngliche Knoten zum rechten Kind des neuen Wurzelknotens wird.
- `rotate_left(self, node)` führt eine Linksrotation aus, dass rechte Kind des Knotens wird der neue Wurzelknoten, während der ursprüngliche Knoten zum linken Kind des neuen Wurzelknotens wird.

AVL-Überprüfung

`is_avl(self)` überprüft zuerst, ob der Baum existiert (also die Wurzel nicht `None` ist). Wenn der Baum existiert, wird die Methode `_is_avl(self, node)` aufgerufen, die rekursiv die AVL-Eigenschaften des Baumes überprüft. Es wird also kontrolliert, ob ein Knoten im Baum aus dem Gleichgewicht ist (Ausgleichsfaktor größer als 1). Wenn dies der Fall ist, wird `False` zurückgegeben, andernfalls wird `True` zurückgegeben.

Durchschnittswertermittlung

Die Methode `_avg_key(self, node)` berechnet den Durchschnittswert der Schlüssel im Baum. Hierbei wird der Baum rekursiv durchlaufen und die Summe der Schlüsselwerte und die Anzahl der Schlüssel ermittelt. Diese Werte werden an die Funktion `avg_key(self)` übergeben, die den Durchschnittswert berechnet und zurückgibt.

Min- & Max-Berechnung

Die Methoden `_min_key(self, node)` & `_max_key(self, node)` funktionieren ähnlich. `_min_key(self, node)` durchläuft rekursive die linken Kindknoten des aktuellen Knotens, während `_max_key(self, node)` rekursiv die rechten Kindknoten durchläuft. Beide Methoden brechen ab, wenn sie keinen weiteren Knoten auf der linken bzw. rechten Seite finden - der Knoten an der aktuellen Stelle ist der Knoten mit dem größten/kleinsten Wert.

Einfache Suche

Die Funktion `simpleSearch(mainTreeRoot, subTreeRoot, nodes=[])` wird aufgerufen, wenn der Subtree `subtree.txt` eine Länge von 1 hat. Die Funktion durchläuft den Hauptbaum rekursiv, beginnend von der Wurzel `mainTreeRoot` und sucht nach dem einzigen Knoten des Subtrees `subTreeRoot`. Sollte der Knoten gefunden werden, wird der Pfad von der Wurzel des Hauptbaumes zum gesuchten Knoten ausgegeben. Andernfalls wird die Suche im linken oder im rechten Teilbaum vom Hauptbaum fortgesetzt, je nachdem ob der gesuchte Schlüssel größer oder kleiner als der Schlüssel des aktuellen Knotens vom Hauptbaum ist. In `nodes=[]` wird der Pfad von der Wurzel des Hauptbaumes zum gesuchten Knoten gespeichert. Nach erfolgreicher oder erfolgloser Suche wird die Liste wieder zurückgesetzt.

Subtree-Suche

`isSubtree(mainTreeRoot, subTreeRoot)` wird aufgerufen, wenn der Subtree `subtree.txt` eine Länge von mehr als 1 hat. Die Funktion überprüft zunächst, ob `mainTreeRoot` und `subTreeRoot` `None` sind, in diesem Fall wird `True` zurückgegeben. Sollte nur `mainTreeRoot` `None` sein, wird automatisch `False` zurückgegeben. Andernfalls wird die Funktion `isIdentical(mainTreeRoot, subTreeRoot)` aufgerufen, um die beiden Bäume zu vergleichen. `isIdentical(mainTreeRoot, subTreeRoot)` vergleicht zwei Bäume anhand ihrer Werte und Struktur.

- `if subTreeRoot is None` überprüft, ob der Subtree leer ist, in diesem Fall wird `True` zurückgegeben.
- `if mainTreeRoot is None` überprüft, ob der Hauptbaum leer ist, in diesem Fall wird `False` zurückgegeben.
- `if mainTreeRoot.key != subTreeRoot.key` überprüft, ob die Knoten der beiden Bäume ungleich sind, in diesem Fall wird die Funktion rekursiv aufgerufen, um zu überprüfen, ob der Subtree ein Teil des linken oder rechten Teilbaums des Hauptbaum ist. Die `(isIdentical(mainTreeRoot.left, subTreeRoot) or isIdentical(mainTreeRoot.right, subTreeRoot))` Bedingung wird verwendet, um zu überprüfen, ob der Subtree in einem der Teilbäume des Hauptbaums vorhanden ist.
- `if mainTreeRoot.key == subTreeRoot.key` überprüft, ob die Knoten der beiden Bäume gleich sind, in diesem Fall wird die Funktion rekursiv aufgerufen, um die Knoten auf der linken und rechten Seite der beiden Bäume zu vergleichen.

Sollte `isIdentical(mainTreeRoot, subTreeRoot)` `False` zurückgeben, wird `isSubtree(mainTreeRoot, subTreeRoot)` rekursiv aufgerufen, um zu überprüfen, ob der Teilbaum ein Unterbaum des linken oder rechten Teilbaums des Hauptbaums ist.

Aufwandsabschätzung

Operation	**Best Case**	**Worst Case**	**Average Case**
Suchfunktion (simpleSearch)	$O(1)$	$O(n)$	$O(\log n)$
Suchfunktion (isSubtree)	$O(1)$	$O(m * n)$	$O(m * n)$
Einfügefunktion (_insert)	$O(1)$	$O(n)$	$O(\log n)$
Höhenberechnung (height)	$O(\log n)$	$O(n)$	$O(n)$
Balance-Faktor-Berechnung (balance_Factor)	$O(\log n)$	$O(n)$	$O(\log n)$
Balancierung & Rotation (balance)	$O(\log n)$	$O(n)$	$O(n)$
Durchschnittswertermittlung (_avg_key)	$O(n)$	$O(n)$	$O(n)$
Min- & Maxermittlung (_min_key, _max_key)	$O(\log n)$	$O(n)$	$O(\log n)$
AVL-Überprüfung (_is_avl)	$O(\log n)$	$O(n^2)$	$O(n \log n)$

Tabelle 1: Aufwandsabschätzung für verschiedene Operationen