Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
M. Fuchs, G. Schmid

Algorithmen und Datenstrukturen
Sommersemester 2026
Musterlösung Übungsblatt 7

Abgabe: Dienstag, 16. Juni, 2026, 10:00 Uhr

Aufgabe 1: AVL-Eigenschaft und Rotationen *(3 Punkte)*

Ein AVL-Baum ist ein binärer Suchbaum, in dem für jeden Knoten gilt, dass sich die Höhen des linken und rechten Teilbaums um höchstens eins unterscheiden.

a) Fügen Sie die Schlüssel

$$10, 20, 30, 40, 50, 25$$

der Reihe nach in einen zunächst leeren AVL-Baum ein. Zeichnen Sie den Baum nach jeder Einfügung, bei der eine Rotation durchgeführt wird. *(2 Punkte)*

b) Geben Sie für jede Rotation aus Teilaufgabe a) an, ob es sich um eine Linksrotation, Rechtsrotation, Links-Rechts-Rotation oder Rechts-Links-Rotation handelt. *(1 Punkt)*

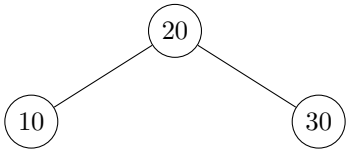
Musterlösung

Einfügen von 10 **und** 20**.** Nach dem Einfügen von 10 bzw. 20 ist der Baum noch balanciert. Es ist keine Rotation notwendig.

Einfügen von 30**.** Nach dem normalen BST-Einfügen erhält man lokal die Kette

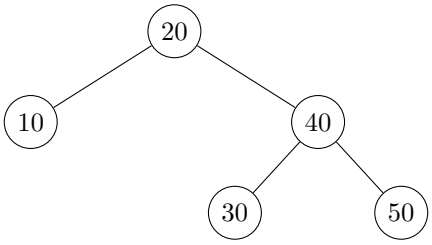
$$10 \to 20 \to 30.$$

Der Knoten 10 ist rechtslastig. Es liegt ein Rechts-Rechts-Fall vor, also wird bei 10 eine Linksrotation durchgeführt. Danach ist der Baum:

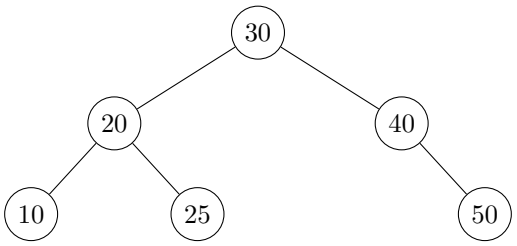


Einfügen von 40**.** Nach dem Einfügen von 40 ist der Baum weiterhin AVL-balanciert. Es ist keine Rotation notwendig.

Einfügen von 50**.** Der Knoten 30 wird rechtslastig. Es liegt wieder ein Rechts-Rechts-Fall vor. Daher wird bei 30 eine Linksrotation durchgeführt. Danach ist der Baum:



Einfügen von 25. Der Schlüssel 25 wird als linkes Kind von 30 eingefügt. Dadurch wird der Knoten 20 rechtslastig, wobei das rechte Kind 40 linkslastig ist. Es liegt also ein Rechts-Links-Fall vor. Man rotiert zunächst bei 40 nach rechts und anschließend bei 20 nach links. Danach ist der Baum:



Damit ergeben sich die folgenden Rotationen:

a) Nach dem Einfügen von 30: Linksrotation bei 10.

b) Nach dem Einfügen von 50: Linksrotation bei 30.

c) Nach dem Einfügen von 25: Rechts-Links-Rotation, bestehend aus einer Rechtsrotation bei 40 und einer Linksrotation bei 20.

Aufgabe 2: Programmieraufgabe - AVL-Baum in Python *(10 Punkte)*

In dieser Aufgabe implementieren Sie vier zentrale Operationen auf AVL-Bäumen:

- *rebalance(node)*
- *rotate_right(node)*
- *insert(key, value=None)*
- *successor(node)*

Die Hilfsfunktionen für Höhen, Balance-Faktoren und Links-Rotationen sind bereits vorgegeben. Ihre Aufgabe ist es, ausschließlich die vier mit `TODO` markierten Funktionen zu vervollständigen.

Anforderungen

Ihre Implementierung muss folgende Eigenschaften erfüllen.

i) Nach `rebalance` ist die AVL-Eigenschaft wieder hergestellt, falls der balance Faktor falsch war.

ii) Nach jeder Funktion haben die Strukturvariablen des Baums die korrekten Werte (z.B. die `height` Werte der Knoten werden angepasst).

iii) Nach einer Rechts-Rotation ist der Baum immernoch ein Binärer Suchbaum.

iv) Nach jedem Aufruf von `insert` ist der Baum weiterhin ein binärer Suchbaum.

v) Nach jedem Aufruf von `insert` erfüllt der Baum die AVL-Eigenschaft.

vi) Parent-Pointer müssen korrekt gesetzt sein.

vii) Wird ein bereits vorhandener Schlüssel eingefügt, soll kein neuer Knoten erzeugt werden. Stattdessen wird nur der Wert aktualisiert.

viii) `successor(node)` gibt den Knoten mit dem nächstgrößeren Schlüssel zurück oder `None`, falls es keinen solchen Knoten gibt.

ix) `insert` soll in $O(\log n)$ Zeit laufen.

x) `successor` soll in $O(\log n)$ Zeit laufen.

xi) `right_rotate` soll in $O(1)$ Zeit laufen.

Musterlösung

Die Lösung finden Sie in der Lösungsdatei `avl_tree.py` auf der Website.

Aufgabe 3: Tests für Ihre AVL-Implementierung *(3 Punkte)*

Schreiben Sie zusätzlich eine Datei `test_avl_tree.py`. Sie dürfen dafür entweder einfache `assert`-Anweisungen oder `pytest` verwenden.
Testen Sie mindestens die folgenden Fälle.

a) Einfügen in einen leeren Baum. *(0.5 Punkte)*

b) Einfügefolgen, die jeweils eine Linksrotation, Rechtsrotation, Links-Rechts-Rotation und Rechts-Links-Rotation erzwingen. *(0.5 Punkte)*

c) Eine längere Einfügefolge, nach der `inorder_keys()` die sortierte Reihenfolge der eingefügten Schlüssel liefert. *(0.5 Punkte)*

d) Mehrere Aufrufe von `successor`, darunter für den kleinsten Schlüssel, einen inneren Schlüssel und den größten Schlüssel. *(0.5 Punkte)*

e) Mehrfaches Einfügen desselben Schlüssels. Überprüfen Sie, dass sich die Größe des Baums nicht erhöht und der Wert aktualisiert wird. *(0.5 Punkte)*

f) Passen Sie ihr Makefile an, so dass alle Tests automatisch auf dem Buildserver laufen. *(0.5 Punkte)*

Musterlösung

Die Lösung finden Sie in der Lösungsdatei `test_avl_tree.py` auf der Website.

Aufgabe 4: Anwendungsfall - Nächste Termine effizient finden *(4 Punkte)*

Ein Kalender speichert Termine nach ihrer Startzeit. Jeder Termin hat einen eindeutigen ganzzahligen Schlüssel, zum Beispiel die Startzeit als Unix-Zeitstempel. Die Termine werden in einem AVL-Baum gespeichert.
Für einen gegebenen Termin v sollen die nächsten k späteren Termine in chronologischer Reihenfolge ausgegeben werden. Dabei soll nicht der ganze Baum traversiert werden.

a) Beschreiben Sie einen Algorithmus `next_k_events(v, k)`, der die `successor`-Funktion benutzt, um die nächsten k Termine nach v auszugeben. *(1 Punkt)*

b) Geben Sie Pseudocode für Ihren Algorithmus an. *(1 Punkt)*

c) Analysieren Sie die Laufzeit Ihres Algorithmus in Abhängigkeit von n und k. Nutzen Sie dabei aus, dass der Baum ein AVL-Baum ist. *(1 Punkt)*

d) Erklären Sie, warum eine vollständige Inorder-Traversierung des Baums für diesen Anwendungsfall unnötig ineffizient wäre, wenn $k \ll n$ gilt. *(1 Punkt)*

Musterlösung

a) Algorithmus

Man startet beim gegebenen Knoten v. Der erste spätere Termin ist der Successor von v. Danach wird wiederholt der Successor des zuletzt gefundenen Knotens berechnet. Jeder Successor liefert genau den nächsten Termin in aufsteigender Schlüsselreihenfolge. Der Algorithmus endet, sobald k Termine ausgegeben wurden oder kein weiterer Successor existiert.

b) Pseudocode

```
next_k_events(v, k):
    result = empty list
    current = successor(v)

    while current != None and length(result) < k:
        append current to result
        current = successor(current)

    return result
```

Falls statt der Knoten selbst nur die Termine ausgegeben werden sollen, wird in der Schleife jeweils der zu `current` gespeicherte Wert ausgegeben bzw. in `result` eingefügt.

c) Laufzeit

Ein AVL-Baum mit n Knoten hat Höhe $O(\log n)$. Ein Aufruf von `successor` benötigt daher im Worst Case $O(\log n)$ Zeit: Entweder steigt man einmal in den rechten Teilbaum ab, oder man läuft über Parent-Pointer nach oben.
Der Algorithmus ruft `successor` höchstens $k+1$-mal auf. Damit ergibt sich die Worst-Case-Laufzeit

$$O(k \log n).$$

Genauer ist die Laufzeit $O(\min\{k, m\} \log n)$, wobei m die Anzahl der tatsächlich nach v existierenden späteren Termine ist. Für die Aufgabenstellung genügt die Schranke $O(k \log n)$.

d) Warum keine vollständige Inorder-Traversierung?

Eine vollständige Inorder-Traversierung gibt zwar alle Termine in chronologischer Reihenfolge aus, kostet aber immer $\Theta(n)$ Zeit. Das ist unnötig, wenn nur die nächsten k Termine nach einem gegebenen Termin gesucht werden und $k \ll n$ gilt. Die Successor-Methode besucht nur Knoten entlang weniger Suchpfade und die tatsächlich ausgegebenen Nachfolger. Deshalb ist sie für kleine k asymptotisch und praktisch deutlich effizienter als eine Traversierung des gesamten Baums.

Abgabe

Geben Sie folgende Dateien ab:

- `loesung.pdf` mit Ihren Lösungen zu den theoretischen Aufgaben,
- `avl_tree.py`,
- `test_avl_tree.py`.
- `Makefile` die mit dem Test-Befehl die Tests in `test_avl_tree.py` aufruft.