Binäre Suchbäume: Eine Einführung

Ein **Binärer Suchbaum (BST)** ist eine Datenstruktur, die Elemente in einer Baum-Struktur speichert und effizientes Suchen, Einfügen und Löschen ermöglicht. Sein Kernprinzip ist die Aufrechterhaltung einer sortierten Reihenfolge der Elemente, was ihn zu einem mächtigen Werkzeug für verschiedene Anwendungen macht.

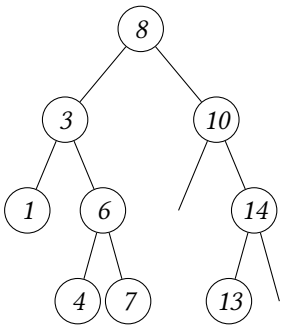
Definition

Ein Binärer Suchbaum ist eine baumbasierte Datenstruktur mit den folgenden Eigenschaften:

1. **Binäre Eigenschaft:** Jeder Knoten im Baum hat höchstens zwei Kinder, die als *linkes Kind* und *rechtes Kind* bezeichnet werden.
2. **Suchbaum-Eigenschaft:** Für jeden Knoten gilt:
   - Alle Werte in seinem *linken Unterbaum* sind kleiner als der Wert des Knotens selbst.
   - Alle Werte in seinem *rechten Unterbaum* sind größer als der Wert des Knotens selbst.
3. **Keine Duplikate (typischerweise):** Während einige Implementierungen Duplikate zulassen, tun dies die meisten Standard-BSTs nicht. Wenn Duplikate erlaubt sind, werden sie normalerweise entweder im linken oder rechten Unterbaum nach einer konsistenten Regel platziert (z.B. alle Duplikate gehen in den rechten Unterbaum).

Der oberste Knoten im Baum wird als **Wurzel** bezeichnet. Knoten ohne Kinder werden als **Blätter** bezeichnet.

Beispiel 1 (Beispiel eines BSTs). *Es folgt die Illustration eines Baumes. Falls ein Knoten nur einen einzigen Kindknoten hat, ist eine leere Kante eingezeichnet. Diese dient nur zur besseren Unterscheidung vom linken und rechten Kindknoten.*



In diesem Beispiel:

- *8 ist die Wurzel.*
- *3 ist das linke Kind von 8, und 10 ist das rechte Kind von 8.*
- *Alle Werte im linken Unterbaum von 8 (3, 1, 6, 4, 7) sind kleiner als 8.*
- *Alle Werte im rechten Unterbaum von 8 (10, 14, 13) sind größer als 8.*

Grundlegende Operationen

1. Einfügen

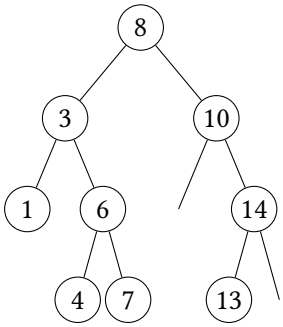
Das Einfügen eines neuen Wertes in einen BST beinhaltet das Durchlaufen des Baumes, um die richtige Position zu finden, an der der neue Knoten platziert werden soll, wobei die drei BST-Eigenschaften beibehalten werden.

Algorithmus:

1. Beginnen Sie an der Wurzel des Baumes.
2. Wenn der Baum leer ist, wird der neue Wert zur Wurzel.
3. Wenn der neue Wert kleiner als der Wert des aktuellen Knotens ist, gehen Sie zum linken Kind.
4. Wenn der neue Wert größer als der Wert des aktuellen Knotens ist, gehen Sie zum rechten Kind.
5. Wiederholen Sie die Schritte 3–4, bis Sie eine NULL-Position (leere Position) erreichen oder der neue Wert gleich dem Wert des aktuellen Knotens ist.
6. Falls Sie eine NULL-Position erreicht haben, fügen Sie den neuen Wert als neuen Knoten an dieser Position ein. Anderenfalls bleibt der BST unverändert und der Algorithmus endet.

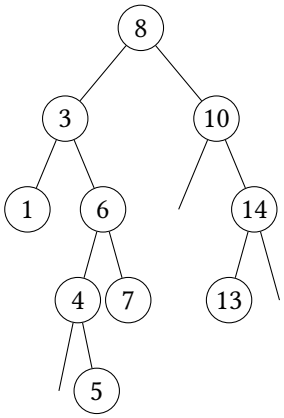
Beispiel: Einfügen von 5 in den BST

Verwenden wir den vorherigen BST:



1. Beginnen Sie bei 8. 5 < 8, gehen Sie nach links zu 3.
2. Bei 3. 5 > 3, gehen Sie nach rechts zu 6.
3. Bei 6. 5 < 6, gehen Sie nach links zu 4.
4. Bei 4. 5 > 4, gehen Sie nach rechts. Das rechte Kind von 4 ist NULL.
5. Fügen Sie 5 als rechtes Kind von 4 ein.

Resultierender BST nach dem Einfügen von 5:



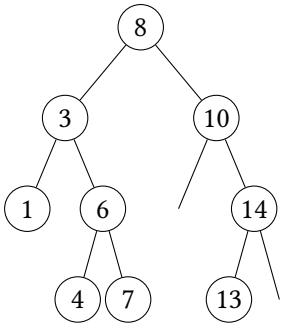
2. Suchen

Das Suchen nach einem bestimmten Wert in einem BST ist aufgrund seiner sortierten Struktur hoch effizient.

Algorithmus:

1. Beginnen Sie an der Wurzel des Baumes.
2. Wenn der Baum leer ist, wurde der Wert nicht gefunden.
3. Wenn der Wert des aktuellen Knotens dem Zielwert entspricht, ist die Suche erfolgreich.
4. Wenn der Zielwert kleiner als der Wert des aktuellen Knotens ist, gehen Sie zum linken Kind.
5. Wenn der Zielwert größer als der Wert des aktuellen Knotens ist, gehen Sie zum rechten Kind.
6. Wiederholen Sie die Schritte 3, 4 und 5, bis der Wert gefunden wird oder Sie eine NULL-Position (leere Position) erreichen (was bedeutet, dass der Wert nicht im Baum ist).

Beispiel: Suchen nach 7 im BST
Verwenden wir den BST vor dem Einfügen von 5:



1. Beginnen Sie bei 8. 7 < 8, gehen Sie nach links zu 3.
2. Bei 3. 7 > 3, gehen Sie nach rechts zu 6.
3. Bei 6. 7 > 6, gehen Sie nach rechts zu 7.
4. Bei 7. Der Wert stimmt mit dem Ziel überein. Suche erfolgreich!

Beispiel: Suchen nach 9 im BST (gleicher Baum wie oben)

1. Beginnen Sie bei 8. 9 > 8, gehen Sie nach rechts zu 10.
2. Bei 10. 9 < 10, gehen Sie nach links. Das linke Kind von 10 ist NULL.
3. Da wir NULL erreicht haben, ohne 9 zu finden, ist die Suche erfolglos.

3. Löschen

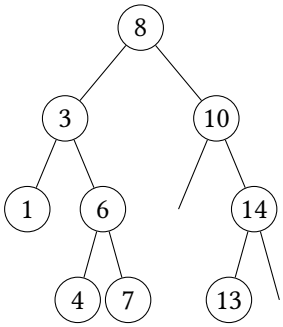
Das Löschen eines Knotens aus einem BST ist die komplexeste der grundlegenden Operationen, da sie die Wahrung der drei BST-Eigenschaften erfordert. Drei Hauptfälle müssen berücksichtigt werden:

Fall 1: Der zu löschende Knoten ist ein Blatt (hat keine Kinder).

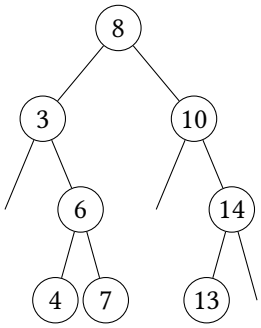
Algorithmus:

1. Finden Sie den Knoten, der gelöscht werden soll.
2. Entfernen Sie den Knoten einfach aus dem Baum.

Beispiel: Löschen von `1` aus dem BST .
Verwenden wir den BST vor dem Einfügen von `5`:



Der Knoten `1` ist ein Blatt. Er wird einfach entfernt. Resultierender BST:

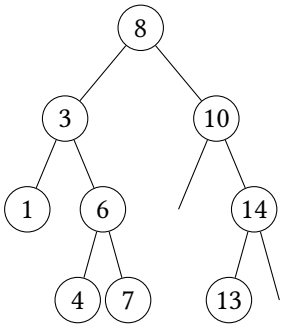


Fall 2: Der zu löschende Knoten hat ein Kind (entweder links oder rechts).

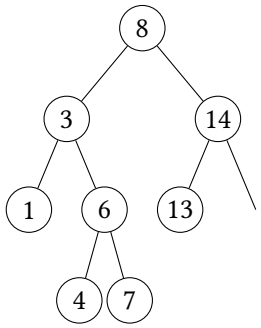
Algorithmus:

1. Finden Sie den Knoten, der gelöscht werden soll.
2. Ersetzen Sie den Knoten durch sein einziges Kind.
3. Löschen Sie den ursprünglichen Knoten.

Beispiel: Löschen von `10` aus dem BST .
Verwenden wir den BST vor dem Einfügen von `5`:



Der Knoten `10` hat nur ein rechtes Kind (`14`). Er wird durch `14` ersetzt. Resultierender BST:

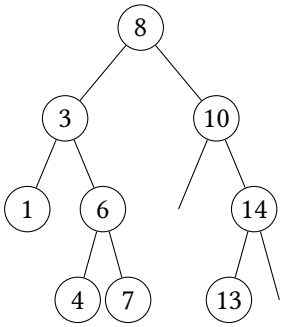


Fall 3: Der zu löschende Knoten hat zwei Kinder.

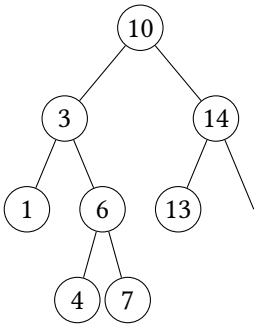
Algorithmus:

1. Finden Sie den zu löschenden Knoten.
2. Finden Sie den **Nachfolger im Baum** (oder **Inorder-Nachfolger**) des Knotens. Der Inorder-Nachfolger ist der kleinste Wert im rechten Unterbaum des Knotens. (Alternativ könnte auch der größte Wert im linken Unterbaum, der **Inorder-Vorgänger**, verwendet werden.)
3. Ersetzen Sie den Wert des zu löschenden Knotens durch den Wert seines Inorder-Nachfolgers.
4. Löschen Sie den Inorder-Nachfolger. Beachten Sie, dass der Inorder-Nachfolger entweder keine oder nur ein rechtes Kind hat, was die Löschung vereinfacht (reduziert auf Fall 1 oder Fall 2).

Beispiel: Löschen von 8 aus dem BST .
Verwenden wir den BST vor dem Einfügen von 5:



Der Knoten 8 hat zwei Kinder. Sein Inorder-Nachfolger ist 10 (der kleinste Wert im rechten Unterbaum). Wir ersetzen 8 durch 10. Dann löschen wir den ursprünglichen Knoten 10.
Resultierender BST:



Fazit

Binäre Suchbäume bieten eine effiziente Möglichkeit, Daten zu speichern und abzurufen. *Wenn der BST balanciert ist,* sogar mit einer logarithmischen Zeitkomplexität für grundlegende Operationen ($O(\log n)$). Allerdings, im schlimmsten Fall (die Rede ist von der sog. worst-case complexity) kann ein BST jedoch zu einer verketteten Liste degenerieren, was zu einer linearen Zeitkomplexität ($O(n)$) führt. Balancierte BSTs (wie etwa AVL-Bäume die im nächsten Abschnitt behandelt werden) beheben dieses Problem, indem sie sich automatisch neu ausbalancieren, um eine optimale Leistung aufrechtzuerhalten.

AVL-Bäume: Eine Einführung in selbstbalancierende BSTs

AVL-Bäume sind selbstbalancierende Binäre Suchbäume. Der "AVL" im Namen steht für die Initialen ihrer Erfinder: Adelson-Velsky und Landis, die sie 1962 veröffentlichten. Der Hauptvorteil eines AVL-Baums gegenüber einem regulären BST ist, dass er garantiert, dass die Höhe des Baumes logarithmisch zur Anzahl der Elemente ist. Dies stellt sicher, dass Such-, Einfüge- und Löschoperationen immer eine Zeitkomplexität von $O(\log n)$ haben, selbst im schlimmsten Fall.

Definition und Balancierungsfaktor

Ein AVL-Baum ist ein Binärer Suchbaum, der die zusätzliche Eigenschaft besitzt, dass für jeden Knoten im Baum die Differenz der Höhen seines linken und rechten Unterbaums höchstens 1 beträgt. Diese Differenz wird als **Balancierungsfaktor** bezeichnet.

$$\text{Balancierungsfaktor}(\text{Knoten}) = \text{Höhe}(\text{linkes Kind}) - \text{Höhe}(\text{rechtes Kind})$$

Ein Knoten ist ausbalanciert, wenn sein Balancierungsfaktor -1, 0 oder 1 ist. Wenn der absolute Wert des Balancierungsfaktors größer als 1 ist, ist der Knoten unausgeglichen, und es müssen Rotationen durchgeführt werden, um den Baum neu zu balancieren.

Grundlegende Operationen und Rotationen

Beim Einfügen oder Löschen eines Knotens in einem AVL-Baum kann die Balancierungsbedingung verletzt werden. Um dies zu beheben, werden eine oder mehrere Baumrotationen durchgeführt. Es gibt vier Haupttypen von Rotationen:

1. **Links-Rotation (RR-Fall):** Wenn ein Knoten im rechten Unterbaum des rechten Kindes eingefügt wird (oder gelöscht wird, was zu einer unausgeglichenen rechten Seite führt).
2. **Rechts-Rotation (LL-Fall):** Wenn ein Knoten im linken Unterbaum des linken Kindes eingefügt wird (oder gelöscht wird, was zu einer unausgeglichenen linken Seite führt).
3. **Links-Rechts-Rotation (LR-Fall):** Eine Links-Rotation auf dem linken Kind, gefolgt von einer Rechts-Rotation auf dem aktuellen Knoten. Dies tritt auf, wenn ein Knoten im rechten Unterbaum des linken Kindes eingefügt wird.
4. **Rechts-Links-Rotation (RL-Fall):** Eine Rechts-Rotation auf dem rechten Kind, gefolgt von einer Links-Rotation auf dem aktuellen Knoten. Dies tritt auf, wenn ein Knoten im linken Unterbaum des rechten Kindes eingefügt wird.

Die Rotationen sind Operationen, die die Struktur des Baumes neu anordnen, um die Balancierungsbedingung wiederherzustellen, ohne eine der drei BST-Eigenschaften zu verletzen.

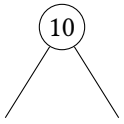
1. Einfügen

Das Einfügen in einen AVL-Baum ähnelt dem Einfügen in einen normalen BST, mit dem zusätzlichen Schritt der Balancierungsprüfung und -korrektur nach dem Einfügen.

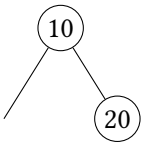
Algorithmus:

1. Fügen Sie den neuen Knoten wie in einem normalen Binären Suchbaum ein.
2. Gehen Sie den Pfad vom neu eingefügten Knoten zurück zur Wurzel und aktualisieren Sie die Höhen der betroffenen Knoten.
3. Überprüfen Sie an jedem Knoten auf diesem Pfad den *Balancierungsfaktor*, wie oben definiert.
4. Wenn ein Knoten unausgeglichen ist (Balancierungsfaktor > 1 oder < -1), führen Sie die entsprechende Rotation (oder Rotationskombination) durch, um den Baum an dieser Stelle zu balancieren.

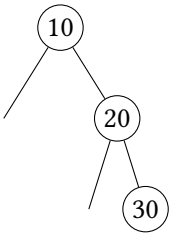
Beispiel: Einfügen von `10, 20, 30` in einen leeren AVL-Baum
1. Einfügen von `10`:



(Balanciert, BF=0)
2. Einfügen von `20`:



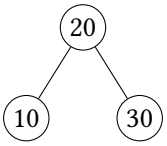
(Balanciert, BF=-1) (links 0 - rechts 1).
2. Einfügen von `30`



Betrachten Sie die Knoten von unten nach oben:

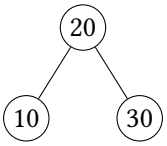
- `30`: BF = 0.
- `20`: BF = -1. (Höhe links 0 - Höhe rechts 1 = -1). Ausbalanciert.
- `10`: BF = -2. (Höhe links 0 - Höhe rechts 2 = -2). Unausgeglichen! Dies ist ein RR-Fall (rechts-rechts).

Führen Sie eine **Links-Rotation** um Knoten `10` durch:

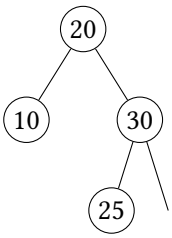


Der Baum ist nun balanciert.

Beispiel: Einfügen von `25` in den oben erzeugten Baum (`20, 10, 30`)
Startbaum:



Einfügen von `25`. Es wird als linkes Kind von `30` eingefügt:



Betrachten Sie die Knoten von unten nach oben:

- `25`: BF = 0.
- `30`: BF = 1 (Höhe links 1 - Höhe rechts 0 = 1). Ausbalanciert.
- `10`: BF = 0. Ausbalanciert.
- `20`: BF = -1 (Höhe links 1 - Höhe rechts 2 = -1). Ausbalanciert.

2. Suchen

Die Suchoperation in einem AVL-Baum ist identisch mit der Suche in einem Binären Suchbaum. Da AVL-Bäume immer balanciert sind, ist die maximale Höhe des Baumes $O(\log n)$, was bedeutet, dass die Suche immer in $O(\log n)$ Zeit durchgeführt wird. Da der Suchalgorithmus der gleiche ist wie bei einem BST, verweisen wir hier einfach auf das Beispiel aus dem BST-Abschnitt.

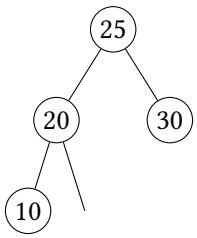
3. Löschen

Das Löschen eines Knotens aus einem AVL-Baum folgt den gleichen drei Fällen wie beim Löschen aus einem Binären Suchbaum. Nach dem Löschen und der möglichen Ersetzung des Knotens muss jedoch der Pfad von der Position des gelöschten Knotens (oder seines Nachfolgers, falls zutreffend) zurück zur Wurzel des Baumes überprüft werden, um die Balancierungsbedingung aufrechtzuerhalten und bei Bedarf Rotationen durchzuführen.

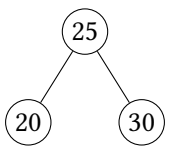
Algorithmus:

1. Löschen Sie den Knoten wie in einem normalen Binären Suchbaum (unter Berücksichtigung der drei Fälle: Blatt, ein Kind, zwei Kinder). Wenn der Knoten zwei Kinder hatte, löschen Sie seinen Inorder-Nachfolger und verwenden Sie dessen Wert zur Ersetzung.
2. Gehen Sie den Pfad von der Position des tatsächlich gelöschten Knotens (oder seines Nachfolgers) zurück zur Wurzel und aktualisieren Sie die Höhen der betroffenen Knoten.
3. Überprüfen Sie an jedem Knoten auf diesem Pfad den Balancierungsfaktor.
4. Wenn ein Knoten unausgeglichen ist, führen Sie die entsprechenden Rotationen durch, um den Baum an dieser Stelle zu balancieren. Beachten Sie, dass mehrere Rotationen auf dem Pfad zur Wurzel erforderlich sein können.

Beispiel: Löschen von `10` aus dem Baum (`25, 20, 30, 10`)



1. Löschen von `10`: `10` ist ein Blatt und wird direkt entfernt.

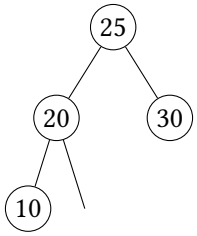


2. Balancierungsprüfung:

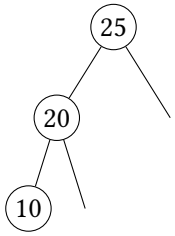
- `20`: BF = 0 (Höhe links 0 - Höhe rechts 0 = 0). Ausbalanciert.
- `30`: BF = 0. Ausbalanciert.
- `25`: BF = 0 (Höhe links 1 - Höhe rechts 1 = 0). Ausbalanciert.

Der Baum bleibt nach dem Löschen von 10 balanciert.

Beispiel: Löschen von 30 **aus dem Baum** $(25, 20, 30, 10)$.
(Wir kehren zum Baum vor dem Löschen von 10 zurück, um ein anderes Szenario zu zeigen.)



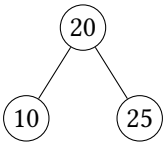
1. Löschen von 30**:** 30 ist ein Blatt und wird direkt entfernt.



2. Balancierungsprüfung:

- 10: BF = 0.
- 20: BF = 1 (Höhe links 1 - Höhe rechts 0 = 1). Ausbalanciert.
- 25: BF = 2 (Höhe links 2 - Höhe rechts 0 = 2). Unausgeglichen! Dies ist ein LL-Fall (Links-Links).

Führen Sie eine **Rechts-Rotation** um Knoten 25 durch:



Der Baum ist nun balanciert.

Fazit

AVL-Bäume lösen die Balancierungsprobleme von Binären Suchbäumen, indem sie nach jeder Einfüge- oder Löschoperation Rotationen durchführen, um sicherzustellen, dass der Balancierungsfaktor jedes Knotens im Bereich [-1, 1] bleibt. Dies garantiert, dass die Höhe des Baumes immer logarithmisch ist ($O(\log n)$), wodurch Such-, Einfüge- und Löschoperationen eine amortisierte Zeitkomplexität von $O(\log n)$. Diese Komplexität gilt auch im schlimmsten Fall. Obwohl die Balancierung zusätzliche Komplexität mit sich bringt, ist daher die garantierte Laufzeit von grundlegenen Operationen in Anwendungen, die häufige Modifikationen erfordern, von großem Vorteil.