Suchbäume

Algorithmen und Datenstrukturen 1
VU 186.813, 4h, 6 ECTS, SS 2016
Letzte Änderung: 27. April 2016

Wörterbuchproblem

Wörterbuch:

- Als Wörterbuch wird eine Menge von Elementen eines gegebenen Grundtyps bezeichnet, auf der man die Operationen Suchen, Einfügen und Entfernen ausführen kann.
- Für jedes Element wird ein Schlüssel k (key) und seine Nutzdaten gespeichert.
- Wir nehmen hier beispielhaft $k \in \mathbb{N}$ an.
- Alle Elemente sind über den Schlüssel identifizierbar.
- Die Operationen sind nur vom Schlüssel abhängig, sodass wir zur weiteren Vereinfachung annehmen, dass ein Wörterbuch aus einer Menge ganzzahliger Schlüssel besteht.

Wörterbuchproblem: Finde eine geeignete Datenstruktur zusammen mit möglichst effizienten Algorithmen zum Suchen, Einfügen und Entfernen von Schlüsseln.

Suchverfahren

Suchen:

- Ein wichtiges Thema der Algorithmentheorie.
- Das Suchen ist eine der grundlegendsten Operationen, die man immer wieder mit Computern ausführt:
  - Suchen von Datensätzen in Datenbanken
  - Suchen nach Wörtern in Wörterbüchern
  - Suchen von Stichwörtern im WWW

Array mit n Schlüsseln: In VU Programmkonstruktion besprochen.

- Lineare Suche in $O(n)$ Zeit.
- Binäre Suche in $O(\log n)$ Zeit.
- Einfügen in ein sortiertes Array in $O(n)$ Zeit.
- Entfernen aus einem sortierten Array in $O(n)$ Zeit.

Frage: Geht es effizienter?
Antwort: Ja, z.B. mit Suchbäumen.

Binäre Suchbäume

Suchbäume

Allgemein: Suchbäume existieren in unterschiedlichen Ausprägungen und bieten i.A. die Möglichkeit folgende Operationen effizient zu implementieren:

- Einfügen eines neuen Elements.
- Suche eines Elements.
- Entfernen eines Elements.
- Durchlaufen aller gespeicherten Elemente in geordneter Reihenfolge.
- Finden des kleinsten/größten Elements.
- Finden eines nächstkleineren/nächstgrößeren Elements.

Wurzelbäume

Baum:

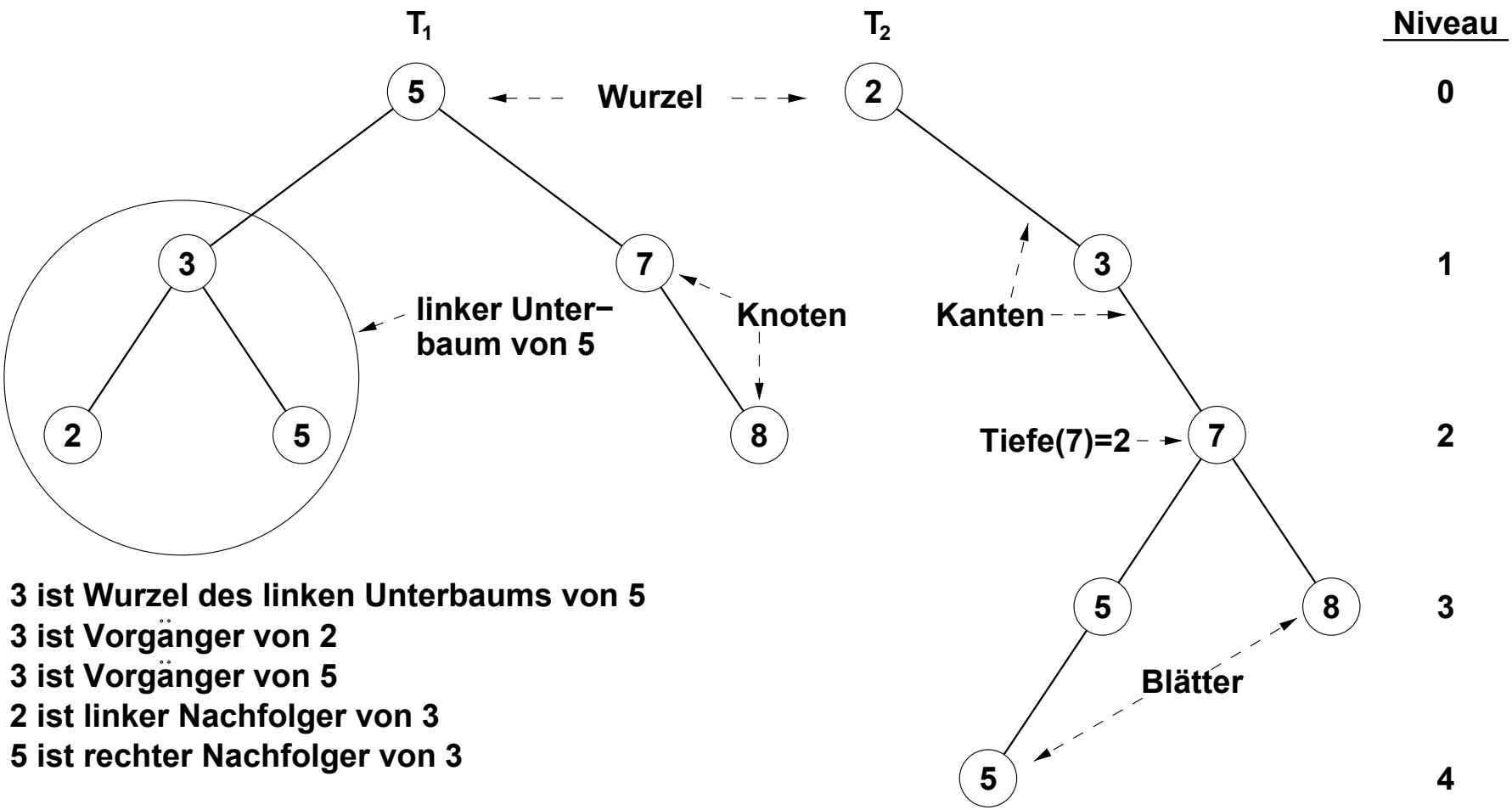
- Wir betrachten hier im Speziellen Wurzelbäume (siehe Kapitel über Graphen).
- Die Kinder (Nachfolger) eines jeden Knotens sind in einer bestimmten Reihenfolge gegeben.

Wurzel:

- Ist der einzige Knoten ohne Vorgänger.
- Alle anderen Knoten können genau über einen Pfad von der Wurzel erreicht werden.
- Jeder Knoten ist gleichzeitig die Wurzel eines Unterbaumes, der aus ihm selbst und seinen direkten und indirekten Nachfolgern besteht.

Binärer Baum: Jeder Knoten besitzt höchstens zwei Kinder, ein „linkes“ und ein „rechtes“ Kind.

Binäre Bäume



3 ist Wurzel des linken Unterbaums von 5
3 ist Vorgänger von 2
3 ist Vorgänger von 5
2 ist linker Nachfolger von 3
5 ist rechter Nachfolger von 3

Wurzelbäume

Tiefe (Niveau) eines Knotens: Tiefe

- Die Tiefe eines Knotens ist die Anzahl der Kanten auf dem Pfad von der Wurzel zu diesem Knoten.
- Die Tiefe ist eindeutig, da nur ein solcher Pfad existiert.
- Die Tiefe der Wurzel ist 0.
- Die Menge aller Knoten mit gleicher Tiefe im Baum wird auch als Ebene oder Niveau bezeichnet.

Wurzelbäume: Höhe

Höhe eines (Teil-)baumes:

- Die Höhe $h(T_r)$ eines (Teil-)Baumes T_r mit dem Wurzelknoten r ist die Länge eines längsten Pfades in dem Teilbaum von r zu einem beliebigen Knoten v in T_r.
- Ein Baum mit nur einem (Wurzel-)Knoten hat die Höhe 0.
- Die Höhe des leeren Baumes definieren wir mit -1.

Blatt:

- Ein Knoten heißt Blatt, wenn er keine Kinder besitzt.
- Alle anderen Knoten nennt man innere Knoten.

Implementierung von binären Bäumen

Knoten x:

$x.key$	Schlüssel von x
$x.left$	Verweis auf linkes Kind ($null$ wenn nicht vorhanden)
$x.right$	Verweis auf rechtes Kind ($null$ wenn nicht vorhanden)
$x.info$	Nutzdaten im Knoten (hier als optional betrachtet)
$x.parent$	Verweis auf Vorgänger von x (optional; $null$ wenn Wurzel)

- Der Zugriff auf den Baum erfolgt über einen Verweis auf den Wurzelknoten, z.B. $root$.
- Ist der Baum leer, so gilt $root = null$.

Durchmusterungen (Traversals): Inorder

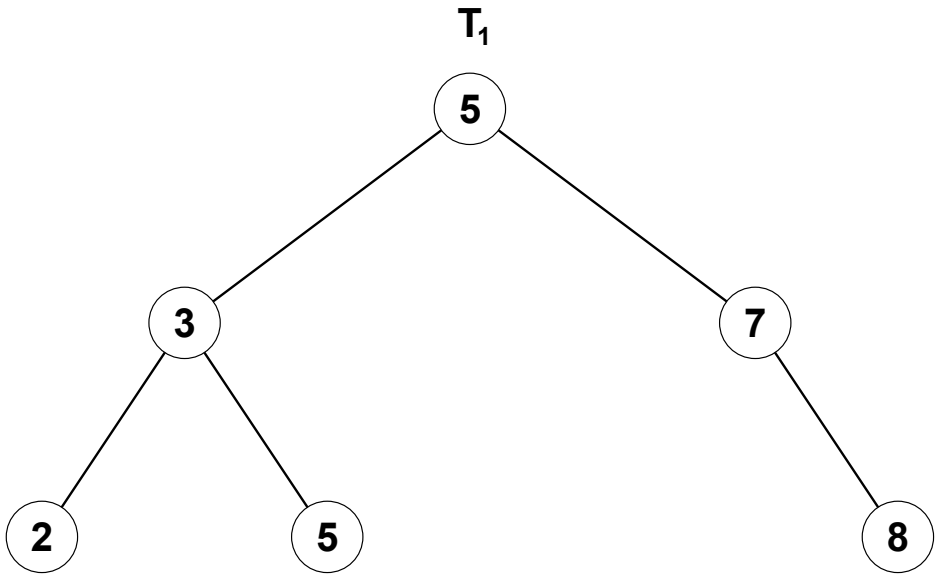
Inorder-Durchmusterung: Behandle rekursiv zunächst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum.

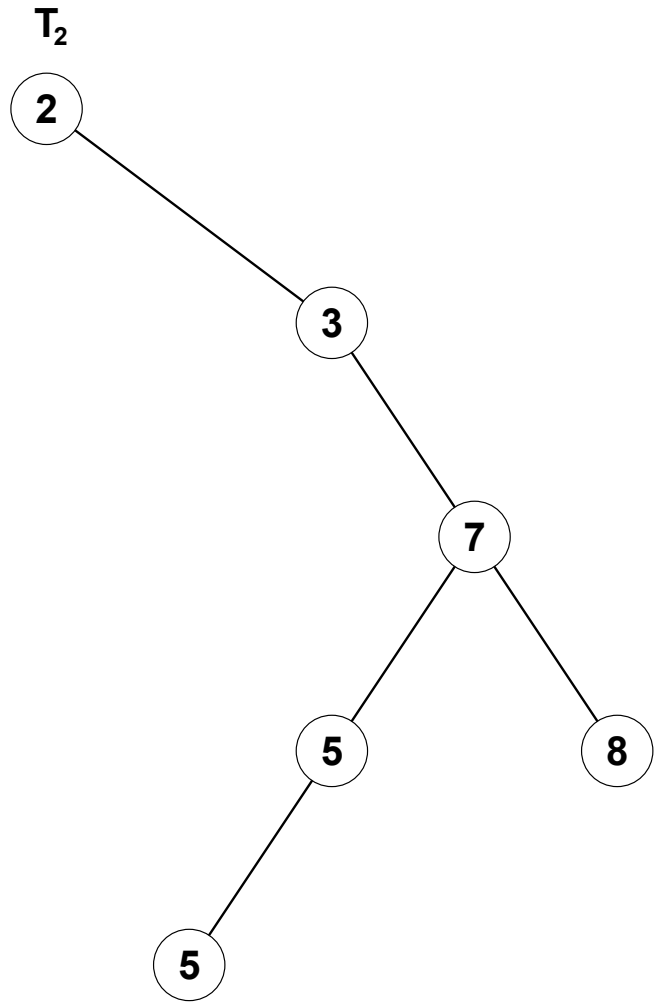
Aufruf: Inorder($root$).

```
Inorder(p)
if p ≠ null
    Inorder(p.left)
    Bearbeite p (z.B. Ausgabe)
    Inorder(p.right)
```

Laufzeit: Die Laufzeit liegt für $n \geq 1$ Knoten in $\Theta(n)$, da für jeden Knoten genau ein rekursiver Aufruf erfolgt, maximal $2n$ zusätzliche Aufrufe für leere Unterbäume hinzukommen, und jeder Aufruf ohne Folgeaufrufe eine Laufzeit von $\Theta(1)$ hat.

Inorder: Beispiel



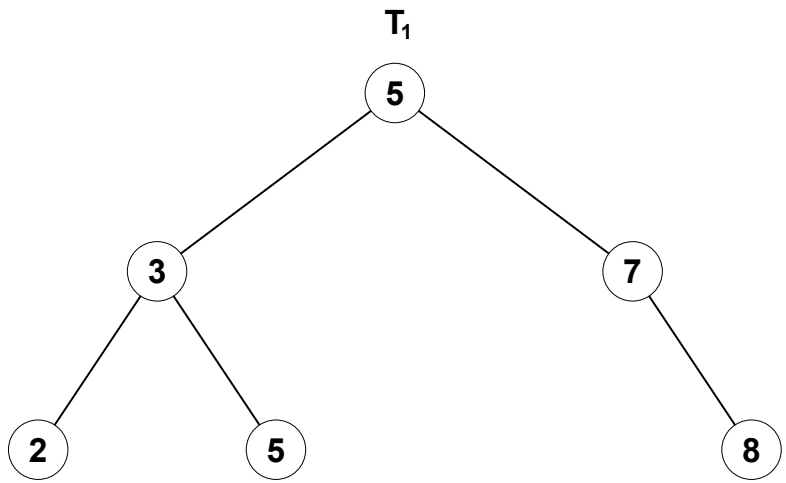


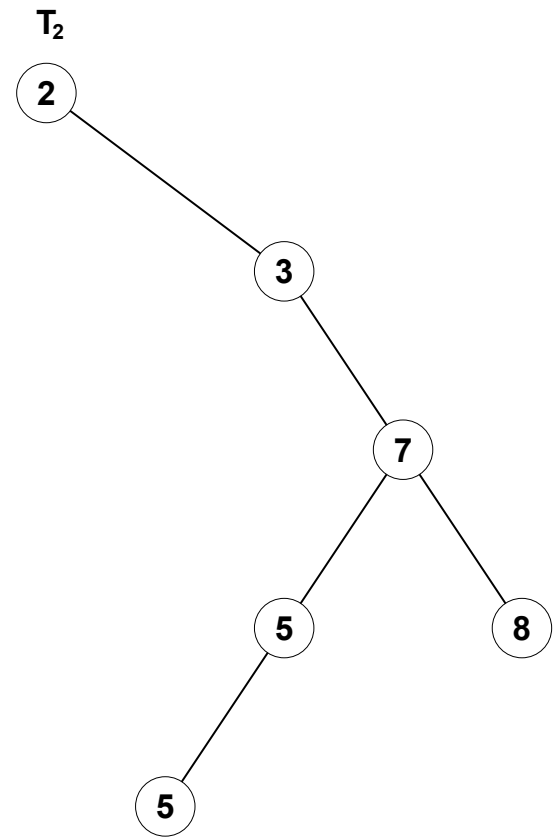
Für beide gezeigten Bäume ist die Inorder-Durchmusterungsreihenfolge 2, 3, 5, 5, 7, 8.

Durchmusterungen: Preorder

Preorder-Durchmusterung: Behandle rekursiv zunächst die Wurzel, dann den linken Unterbaum, danach den rechten Unterbaum.

Beispiel:



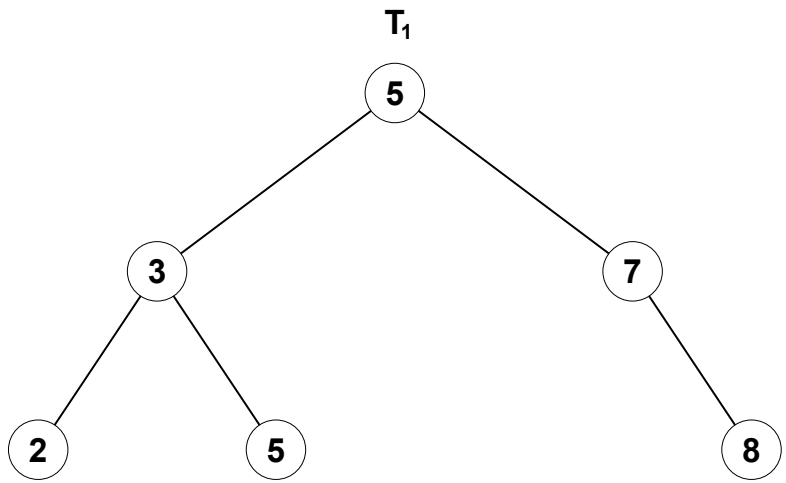


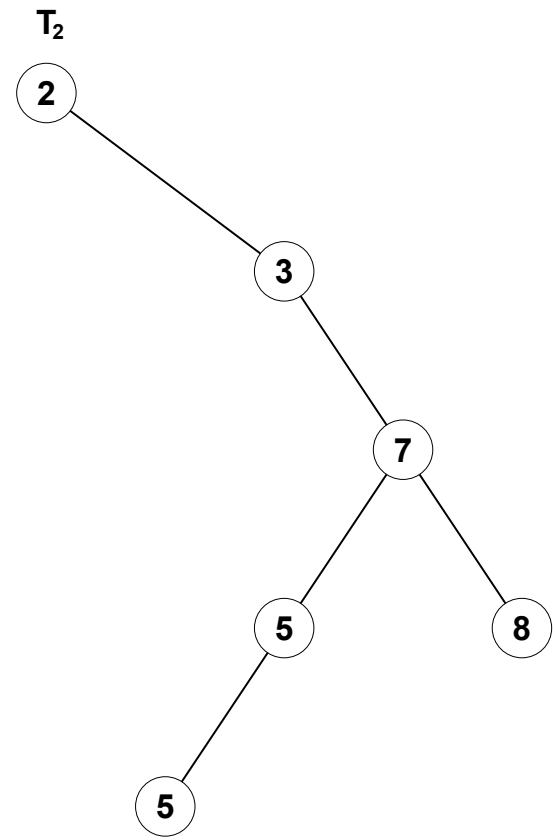
- Für T_1: 5, 3, 2, 5, 7, 8
- Für T_2: 2, 3, 7, 5, 5, 8

Durchmusterungen: Postorder

Postorder-Durchmusterung: Behandle rekursiv zunächst den linken Unterbaum, dann den rechten Unterbaum, danach die Wurzel.

Beispiel:





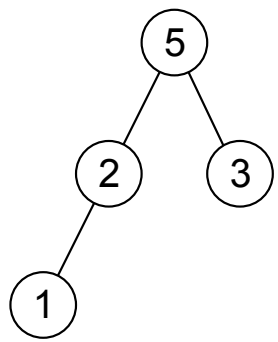
- Für T_1: 2, 5, 3, 8, 7, 5
- Für T_2: 5, 5, 8, 7, 3, 2

Durchmusterungen: Theorem

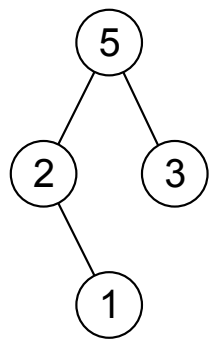
Theorem (ohne Beweis): Ein binärer Baum kann immer eindeutig aus der Inorder- und Preorder-Durchmusterungsfolge der Elemente rekonstruiert werden, wenn die Elemente paarweise unterschiedlich sind. Gleiches gilt für Inorder zusammen mit Postorder, nicht jedoch für Preorder und Postorder.

Beispiel:

- Inorder für (1): 1, 2, 5, 3
- Inorder für (2): 2, 1, 5, 3
- Preorder für beide: 5, 2, 1, 3
- Postorder für beide : 1, 2, 3, 5



(1)



(2)

Binäre Suchbäume

Ein binärer Suchbaum ist ein binärer Baum der in jedem Knoten x folgende binäre Suchbaumeigenschaft erfüllt:

- Ist y ein Knoten des linken Unterbaumes von x so gilt:

$$y.key \leq x.key.$$

- Ist z ein Knoten des rechten Unterbaumes von x so gilt:

$$z.key \geq x.key.$$

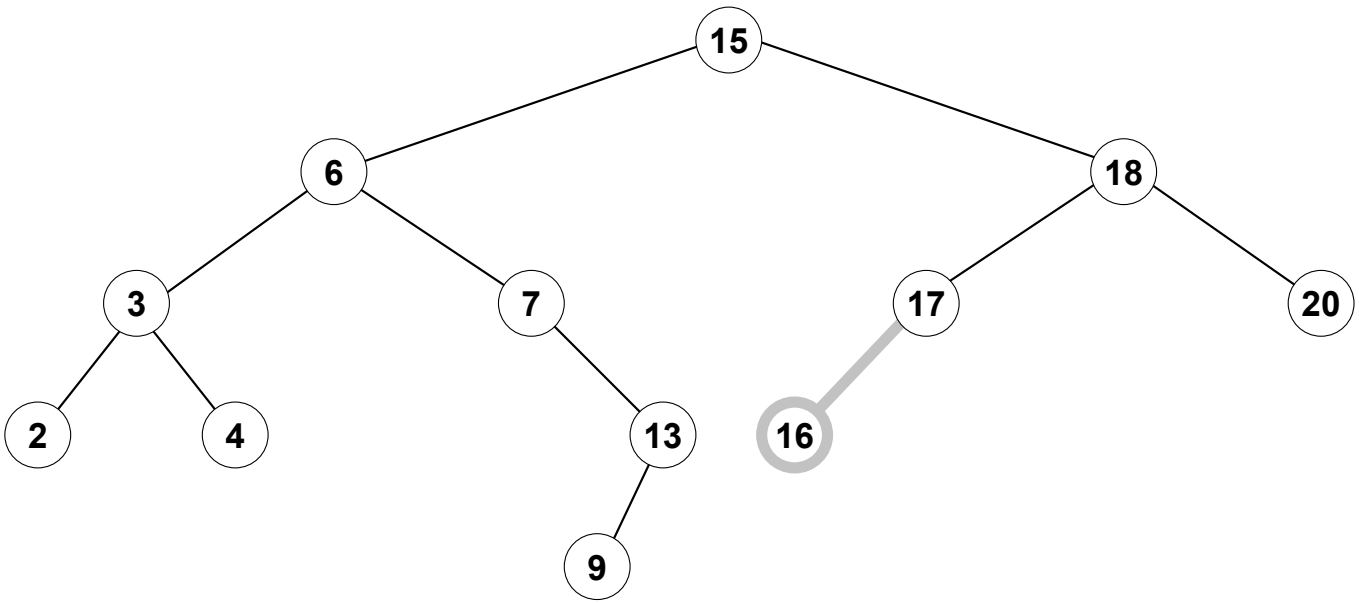
Hinweis: In einer konkreten Implementierung muss man sich entscheiden, ob man gleiche Schlüssel immer links oder immer rechts speichert.

Operationen auf binären Suchbäumen

Typische Operationen:

- Einfügen / Entfernen eines Elements
- Suchen nach einem Element
- Minimum / Maximum: kleinstes / größtes Element finden
- Predecessor / Successor:
  voriges / nächstes Element entsprechend der Sortierreihenfolge finden
- Durchmusterung

Operationen auf binären Suchbäumen



Minimum(7) = 7
Minimum(15) = 2

Maximum(15) = 20
Maximum(6) = 13

Predecessor(15) = 13
Predecessor(9) = 7

Successor(15) = 17
Successor(13) = 15

Insert(16)

Suchen

Eingabe: Baum mit Wurzel p und gesuchter Schlüssel s

Rückgabewert: Knoten mit Schlüssel s oder $null$, falls s nicht vorhanden ist.

```
Search(p,s):
while p ≠ null und p.key ≠ s
    if s < p.key
        p ← p.left
    else
        p ← p.right
return p
```

Minimum

Eingabe: Baum mit Wurzel p.

Rückgabewert: Knoten mit dem kleinsten Schlüssel.

```
Minimum(p):
if p = null
    return null
while p.left ≠ null
    p ← p.left
return p
```

Vorgehen: Solange beim linken Kind weitergehen, bis es keinen linken Nachfolger mehr gibt.

Maximum

Eingabe: Baum mit Wurzel p.

Rückgabewert: Knoten mit dem größten Schlüssel.

```
Maximum(p):
if p = null
    return null
while p.right ≠ null
    p ← p.right
return p
```

Vorgehen: Solange beim rechten Kind weitergehen, bis es keinen rechten Nachfolger mehr gibt.

Successor

Eingabe: Baum mit Wurzel p.

Rückgabewert: Nächster Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Successor(p):
if p.right ≠ null
    return Minimum(p.right)
else
    q ← p.parent
    while q ≠ null und p = q.right
        p ← q
        q ← q.parent
    return q
```

Predecessor

Eingabe: Baum mit Wurzel p.

Rückgabewert: Vorhergehender Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Predecessor(p):
if p.left ≠ null
    return Maximum(p.left)
else
    q ← p.parent
    while q ≠ null und p = q.left
        p ← q
        q ← q.parent
    return q
```

Einfügen

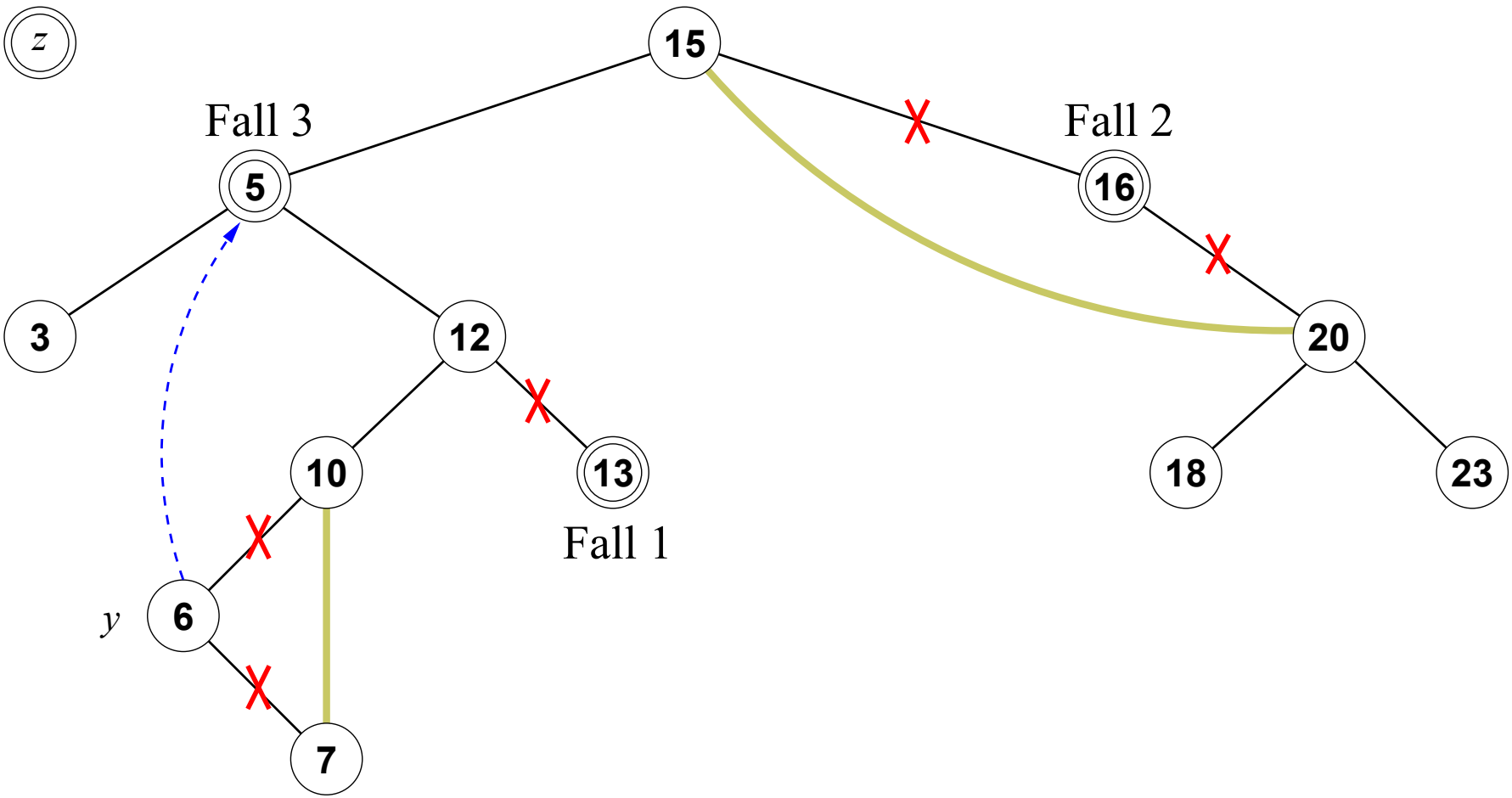
Eingabe: Baum mit Wurzel $root$ und ein neuer Knoten q.

Hinweis: $root$ wird bei leerem Baum verändert (Referenzparameter)

```
Insert(root, q):
r ← null, p ← root
while p ≠ null
    r ← p
    if q.key < p.key
        p ← p.left
    else
        p ← p.right
q.parent ← r, q.left ← null, q.right ← null
if r = null
    root ← q
else
    if q.key < r.key
        r.left ← q
    else
        r.right ← q
```

Entfernen eines Knotens

Entfernen: Knoten z soll entfernt werden. Dabei müssen drei Fälle unterschieden werden:



Entfernen eines Knotens

Fall 1: z hat keine Kinder (z.B. Knoten 13). In diesem Fall kann z einfach entfernt werden.

Fall 2: z hat ein Kind (z.B. Knoten 16). Dieser Fall entspricht dem Entfernen aus einer verketteten linearen Liste.

Fall 3: z hat zwei Kinder (z.B. Knoten 5). Wir bringen an die Stelle des zu löschenden Knotens einen Ersatzknoten und löschen den Ersatzknoten an seiner ursprünglichen Position.
Geeigneter Ersatzknoten für z:

- Der Knoten mit dem größten Schlüssel des linken Unterbaumes (Predecessor) oder
- der Knoten mit dem kleinsten Schlüssel des rechten Unterbaumes (Successor)

Hinweis: Der Ersatzknoten hat in seiner ursprünglichen Position immer maximal einen Nachfolger und wird schließlich gemäß Fall 1 oder 2 entfernt.

Entfernen eines Knotens

Eingabe: Baum mit Wurzel $root$ und ein Knoten q.

Hinweis: $root$ kann verändert werden (Referenzparameter)

```
Remove(root, q):
if q.left = null oder q.right = null
    r ← q
else
    r ← Successor(q), q.key ← r.key, q.info ← r.info
if r.left ≠ null
    p ← r.left
else
    p ← r.right
if p ≠ null
    p.parent ← r.parent
if r.parent = null
    root ← p
else
    if r = r.parent.left
        r.parent.left ← p
    else
        r.parent.right ← p
```

Laufzeiten

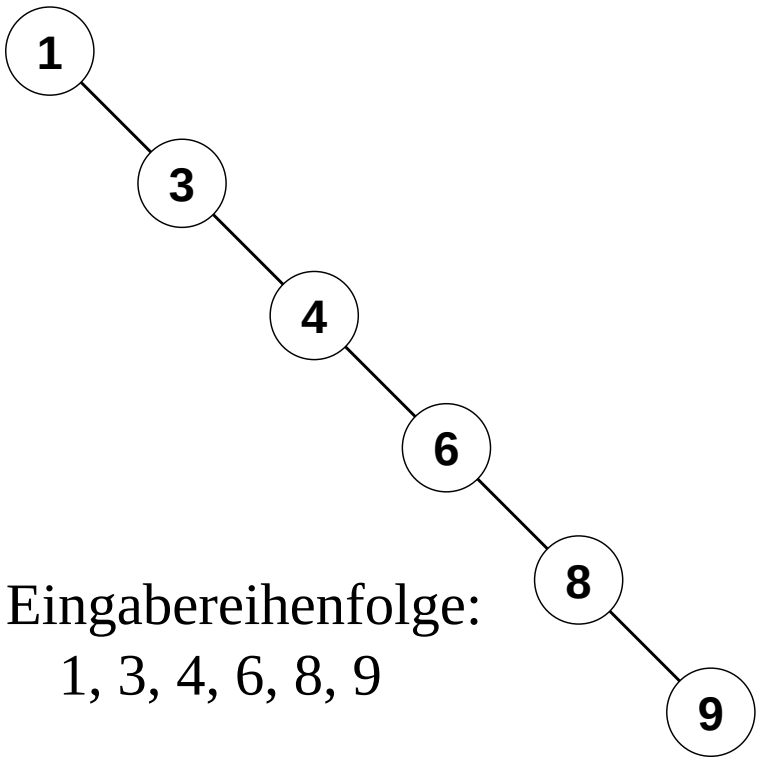
Worst-Case:

- Der im schlechtesten Fall erforderliche Zeitaufwand für Suchen, Einfügen, Entfernen, Minimum, Maximum, Predecessor und Successor in einem binären Suchbaum mit der Höhe h liegt daher in $O(h)$.
- D.h. der Aufwand für diese Operationen hängt unmittelbar von der Höhe ab. In jedem Fall muss man ungünstigstenfalls einem Pfad von der Wurzel zu einem tiefsten Blatt folgen um die Operation auszuführen.

Beispiel für Struktur

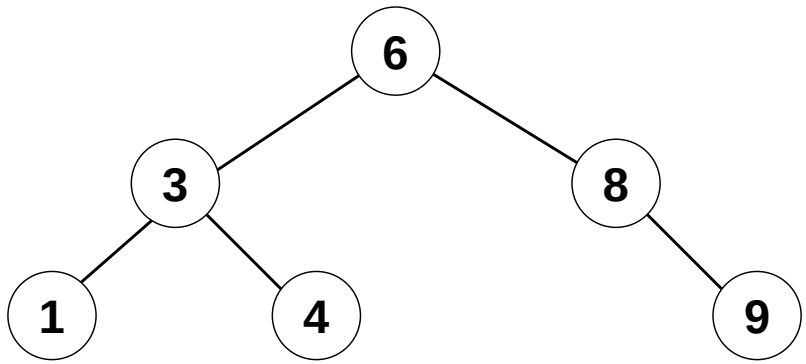
Die Struktur eines binären Suchbaums hängt von der Eingabereihenfolge ab.

T_1 (Höhe=5)



Eingabereihenfolge:
1, 3, 4, 6, 8, 9

T_2 (Höhe=2)



Eingabereihenfolge, z.B.:
6, 8, 3, 4, 1, 9

Einen binären Suchbaum aus einer sortierten Eingabereihenfolge aufzubauen führt zur Entartung in eine lineare Liste!

Laufzeiten

Vollständig balancierter Baum:

- Alle inneren Knoten bis auf jene in der vorletzten Ebene haben 2 Nachfolger.
- Hat die Höhe $h(T) = \lfloor \log_2 n \rfloor$ und daher benötigen alle Operationen bis auf das Durchmustern $O(\log n)$ Zeit.

Zu einer Liste entarteter Baum:

- Ist der Baum jedoch entartet und hat eine Höhe $h(T) = O(n)$, dann benötigen alle Operationen $O(n)$ Zeit!

Laufzeiten

Average-Case:

- Die erwartete durchschnittliche Suchpfadlänge (über alle mögliche binären Suchbäume für n unterschiedliche Elemente) ist für große n nur ca. 40% länger als im Idealfall, d.h in $O(\log n)$.

  □ *Für einen ausführlichen Beweis siehe Seite 277ff in: T. Ottmann und P. Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Spektrum Akademischer Verlag, 2012*

Hinreichend balancierter Baum: Für ein garantiertes logarithmisches Zeitverhalten genügen aber auch hinreichend balancierte Bäume, die wir im nächsten Abschnitt besprechen werden.

Die bisher besprochenen Suchbäume werden im Gegensatz zu den folgenden Bäumen, bei denen wir die Struktur speziell beeinflussen, auch konkreter natürliche Suchbäume genannt.

AVL-Bäume (Adelson-Velski/Landis-Bäume)

Balancierte Bäume

Idee: Suchbaum wird durch geeignete Randbedingungen und entsprechende Umordnungsoperationen effizient ausbalanciert, um zu garantieren, dass seine Höhe logarithmisch bleibt.

Möglichkeiten:

- Höhenbalancierte Bäume: Die Höhe der Unterbäume eines jeden Knotens unterscheidet sich jeweils um höchstens eine Konstante voneinander.
- Gewichtsbalancierte Bäume: Die Anzahl der Knoten in den Unterbäumen jedes Knotens unterscheidet sich höchstens um einen konstanten Faktor.
- (a,b)-Bäume $(2 \leq a \leq b)$: Jeder Knoten (außer der Wurzel) hat zwischen a und b Kinder und alle Blätter haben den gleichen Abstand zur Wurzel (z.B. $a = 2$, $b = 4$).

Wir betrachten in dieser Vorlesung ein Beispiel für höhenbalancierte Bäume (AVL-Bäume) und eines für (a,b)-Bäume (B-Bäume).

AVL-Bäume

Geschichte: Der historisch erste Vorschlag aus 1962 sind AVL-Bäume, die auf Adelson-Velski und Landis zurückgehen.

Generelle Idee: Durch eine Forderung an die Höhendifferenz der beiden Teilbäume eines jeden Knotens wird ein Degenerieren von Suchbäumen verhindert.

Kritische Operationen: Nur Einfügen und Löschen sind kritische Operationen und erfordern eine speziellere Behandlung. Alle anderen Operationen, die den Baum nicht verändern (Suchen, ...), funktionieren wie bisher.

AVL-Bäume

Balance: Balance eines Knotens v: $bal(v) = h_2 - h_1$

- h_1 ... Höhe des linken Unterbaumes von v
- h_2 ... Höhe des rechten Unterbaumes von v

Balancierter Knoten: Ein Knoten v heißt balanciert, wenn $bal(v) \in \{-1, 0, 1\}$.

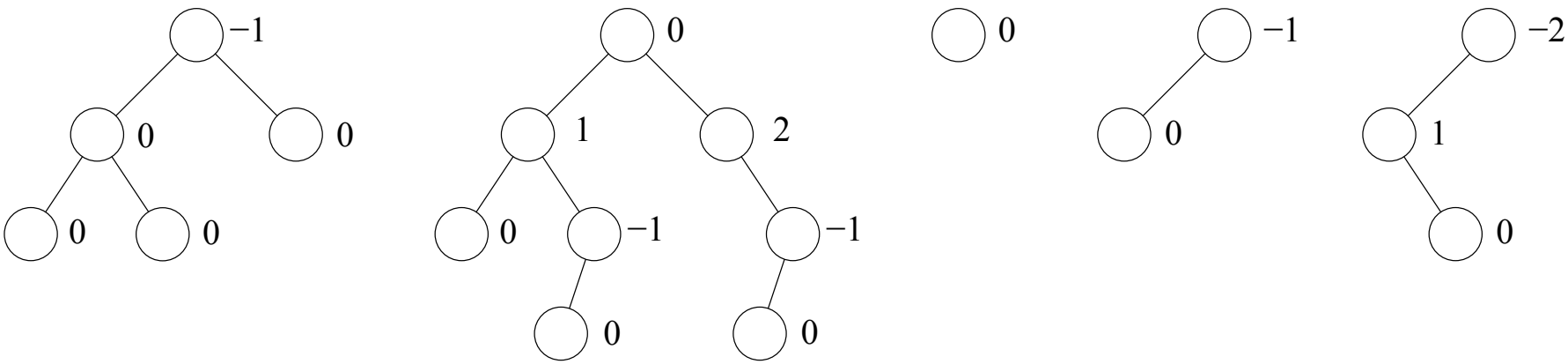
AVL-Bedingung:

- Ein AVL-Baum ist ein binärer Suchbaum, in dem alle Knoten balanciert sind.
- Für jeden Knoten v gilt: Die Höhe des linken Teilbaums unterscheidet sich von der Höhe des rechten Teilbaums um höchstens 1.

Hinweis: Die Höhe eines leeren Baumes haben wir mit -1 definiert.

Balance von Knoten

Balance: Beispiele für die Balance von Knoten in binären Bäumen.



AVL-Bäume: Grundlegende Idee

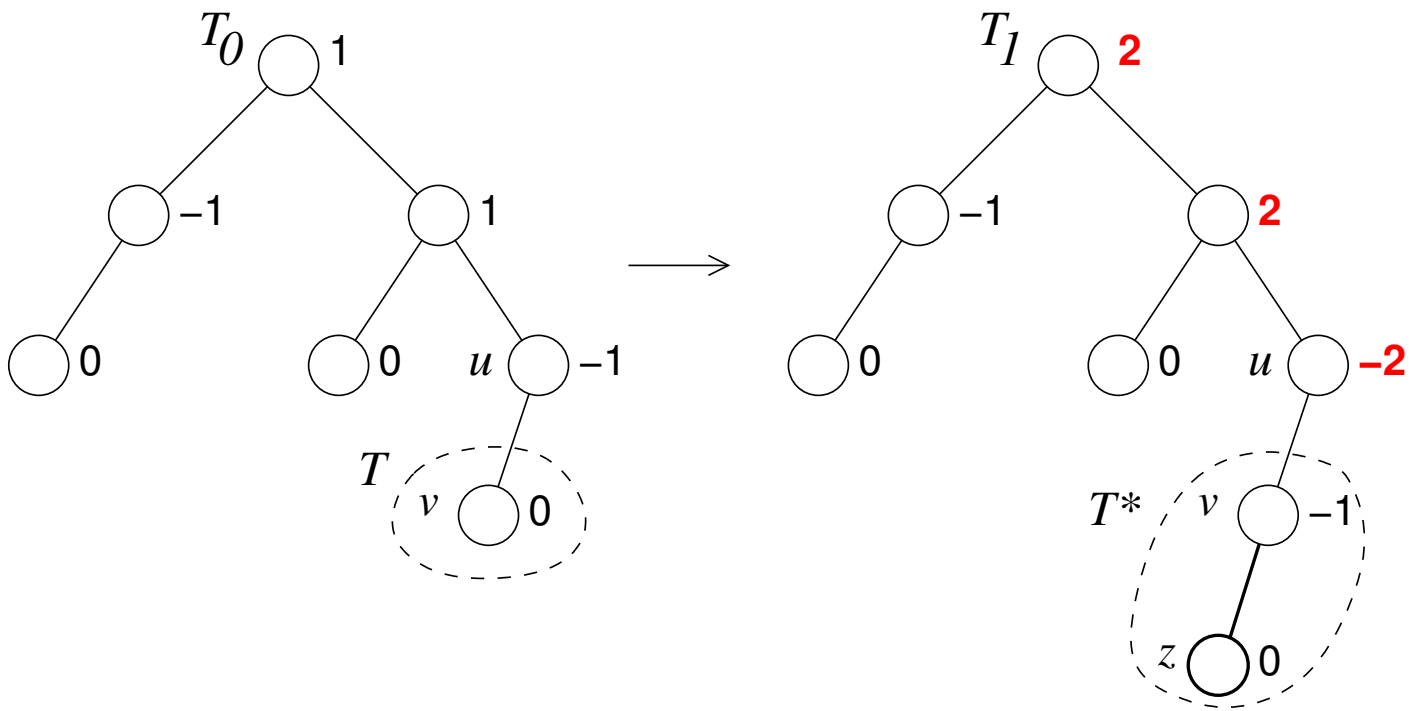
Grundlegende Idee zum Einfügen und Entfernen:

- Führe das Einfügen und Entfernen wie bisher aus.
- Überprüfe danach die Balance in möglicherweise betroffenen Knoten und führe gegebenenfalls eine Rebalancierung (lokale Umordnung) durch, um die Balance in allen Knoten wieder herzustellen.

AVL-Ersetzung

Definition: Eine AVL-Ersetzung

- ist eine Operation (z.B. Einfügen, Löschen),
- die einen Unterbaum T eines Knotens
- durch einen modifizierten (gültigen) AVL-Baum T^* ersetzt,
- dessen Höhe um höchstens 1 von der Höhe von T abweicht.

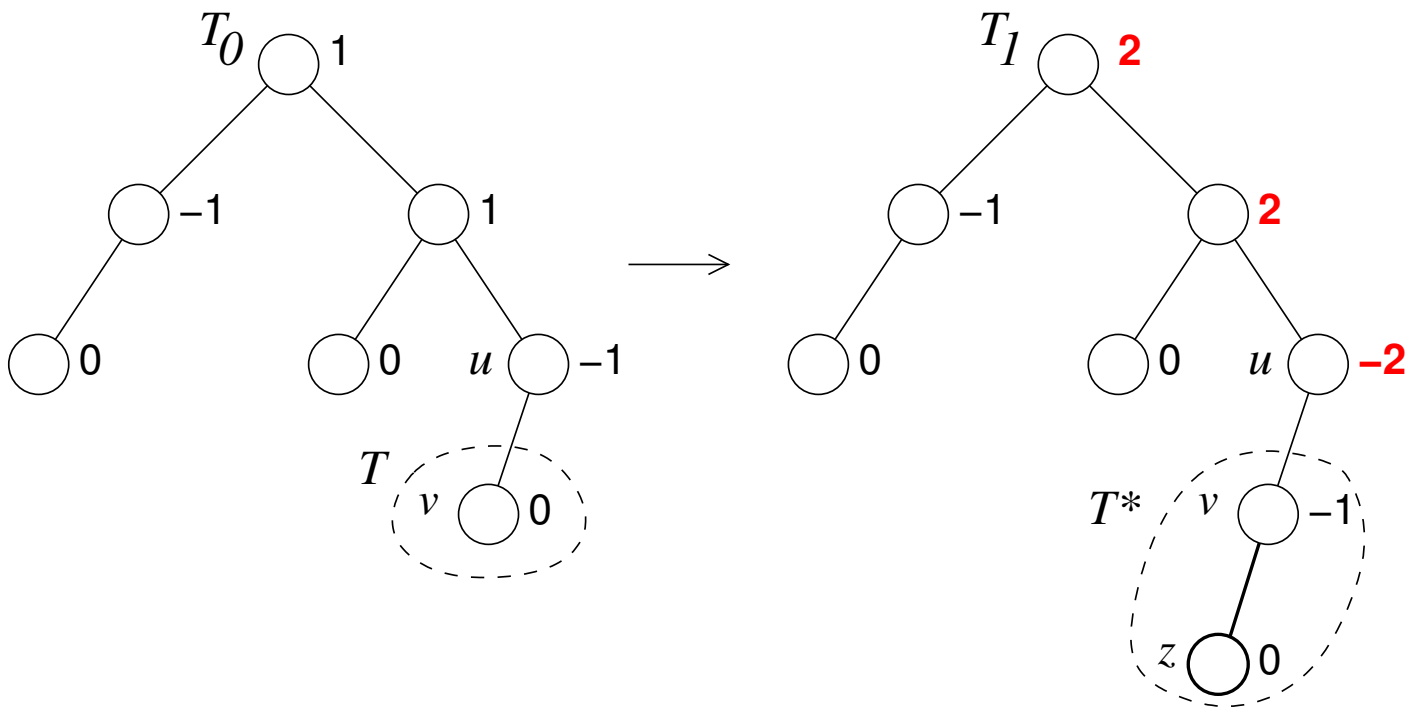


Das Beispiel zeigt, dass es in darüberliegenden Knoten zur Verletzung der Balance kommen kann.

Rebalancierung: Grundlagen

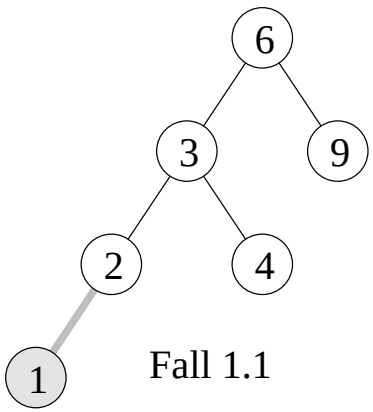
Definitionen:

- Sei T_0 ein gültiger AVL-Baum vor der AVL-Ersetzung und T_1 der unbalancierte Baum hinterher.
- Sei u der unbalancierte Knoten ($bal(u) \in \{-2, +2\}$) maximaler Tiefe. An diesem wird mit der Rebalancierung gestartet.

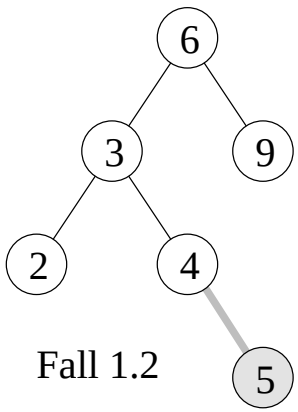


Rebalancierung: Überblick

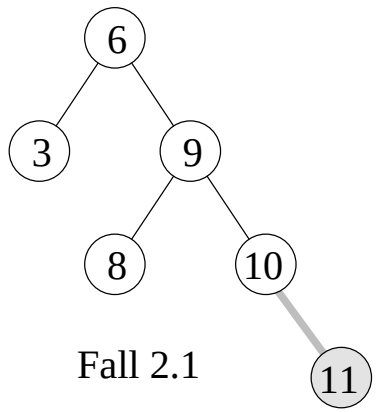
Balancierung: Vier Beispiele für AVL-Bäume, die durch das Einfügen eines Knotens aus der Balance geraten sind.



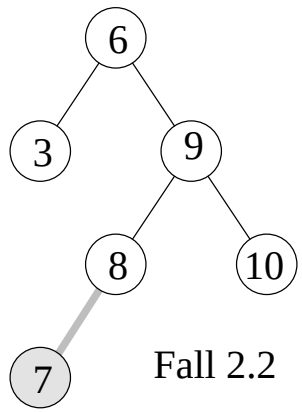
Fall 1.1



Fall 1.2



Fall 2.1



Fall 2.2

Wenn $bal(u) = -2$ und v ist das linke Kind von u und

- $bal(v) \in \{-1, 0\} \rightarrow$ Fall 1.1
- $bal(v) = 1 \rightarrow$ Fall 1.2

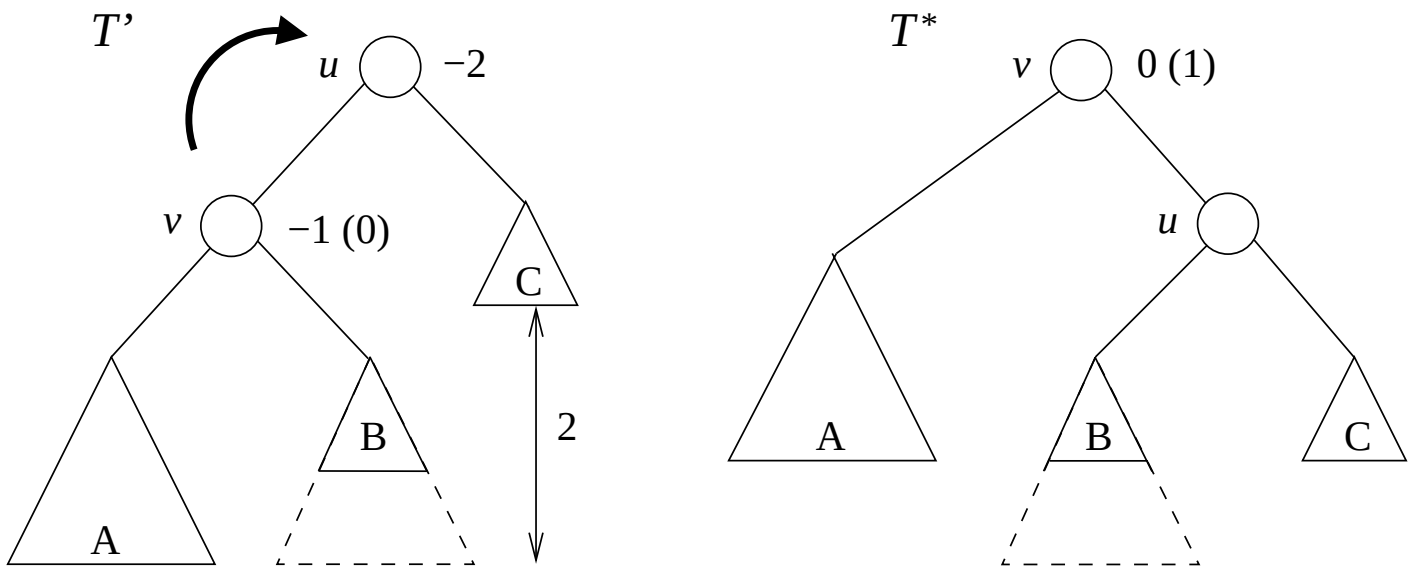
Wenn $bal(u) = 2$ und v ist das rechte Kind von u und

- $bal(v) \in \{0, 1\} \rightarrow$ Fall 2.1
- $bal(v) = -1 \rightarrow$ Fall 2.2

Rebalancierung: Fall 1.1

Fall 1.1: Sei $bal(u) = -2$. Sei v das linke Kind von u und $bal(v) \in \{-1, 0\}$.

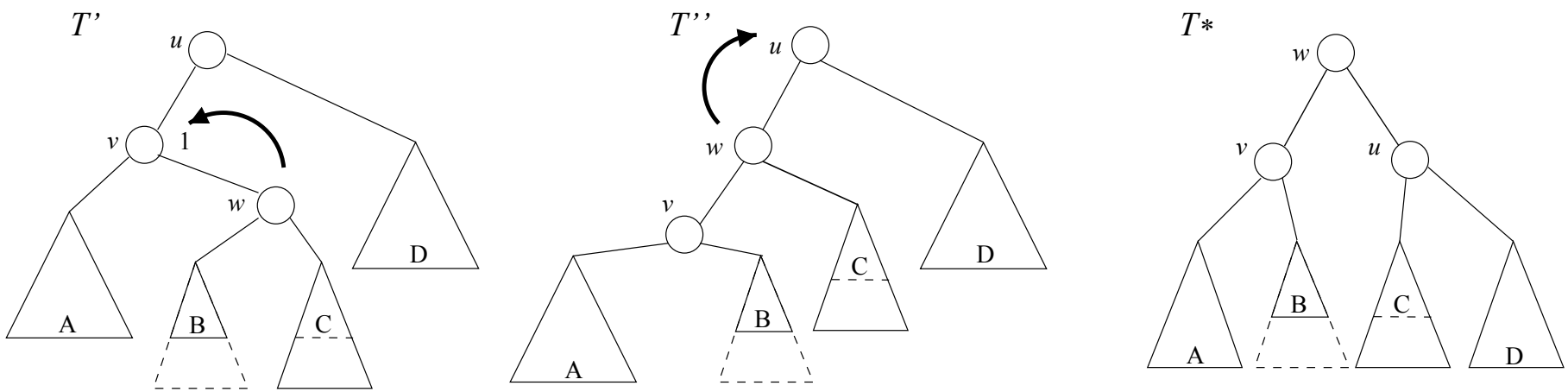
- Der linke Unterbaum des linken Kindes von u ist höher als oder gleich hoch wie der rechte.
- Rebalancierung von u durch einfache Rotation nach rechts an u, d.h. u wird rechtes Kind von v.
- Das rechte Kind von v wird als linkes Kind an u abgegeben.



Rebalancierung: Fall 1.2

Fall 1.2: Sei $bal(u) = -2$. Sei v das linke Kind von u und $bal(v) = 1$.

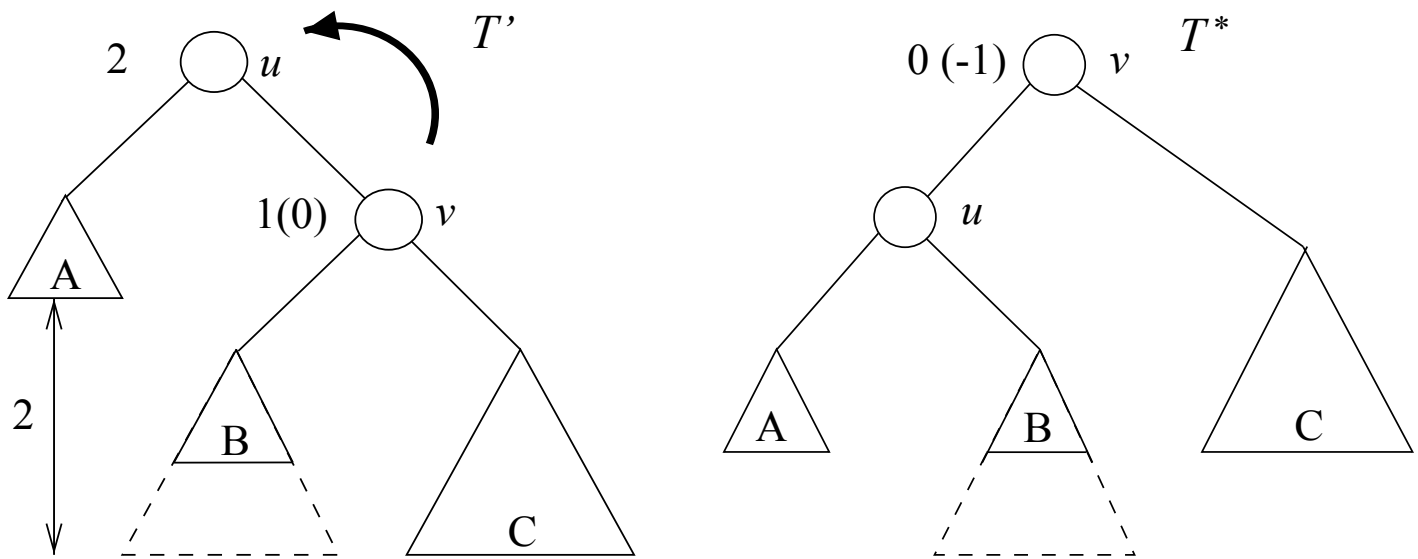
- Der rechte Unterbaum des linken Kindes von u ist höher als der linke.
- Dann existiert das rechte Kind w von v.
- Rebalancierung durch eine Rotation nach links an v und eine anschließende Rotation nach rechts an u (Doppelrotation links-rechts).



Rebalancierung: Fall 2.1

Fall 2.1: Sei $bal(u) = 2$. Sei v das rechte Kind von u und $bal(v) \in \{0, 1\}$.

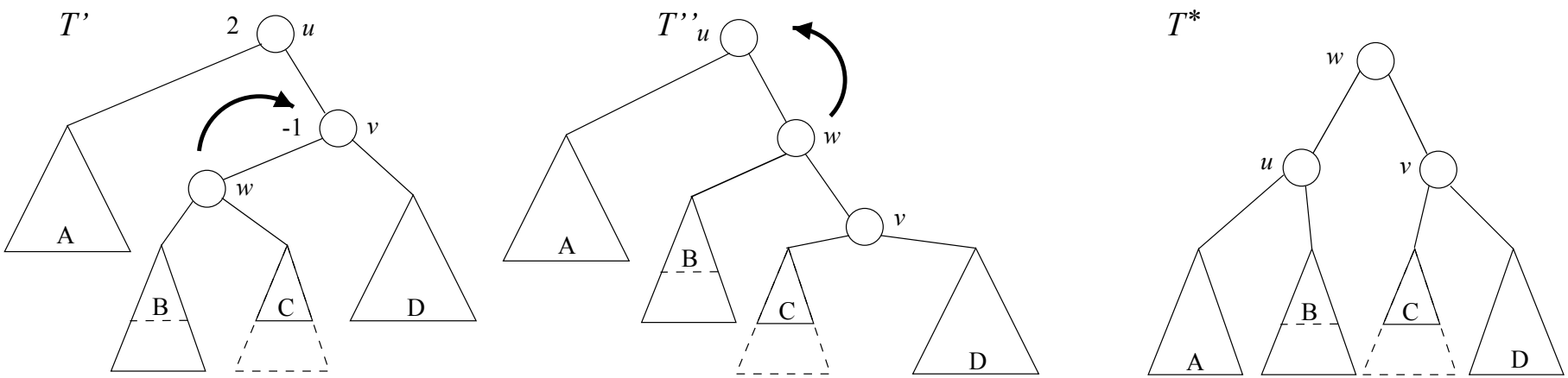
- Der rechte Unterbaum des rechten Kindes von u ist höher als oder gleich hoch wie der linke.
- Rebalancierung von u durch einfache Rotation nach links an u, d.h. u wird linkes Kind von v.
- Das linke Kind von v wird als rechtes Kind an u abgegeben.



Rebalancierung: Fall 2.2

Fall 2.2: Sei $bal(u) = 2$. Sei v das rechte Kind von u und $bal(v) = -1$.

- Der linke Unterbaum des rechten Kindes von u ist höher als der rechte.
- Dann existiert das linke Kind w von v.
- Rebalancierung durch eine Rotation nach rechts an v und eine anschließende Rotation nach links an u (Doppelrotation rechts-links).



Höhe

Implementierung: Wir ergänzen alle Knoten u um ein Attribut $u.height$, das jeweils die Höhe des (Unter-)baumes mit der Wurzel u angibt.

Hilfsfunktion: Wir definieren folgende Hilfsfunktion zur Ermittlung der Höhe eines Baumes:

- Eingabe: Teilbaum mit Wurzel u.
- Rückgabewert: Höhe des Teilbaums.

```
Height(u):
if u = null
    return −1
else
    return u.height
```

Einfache Rotation nach rechts

Eingabe: Teilbaum mit Wurzel u.

Rückgabewert: Neue Wurzel v des rebalancierten Teilbaums.

```
RotateToRight(u):
v ← u.left
u.left ← v.right
v.right ← u
u.height ← max(Height(u.left), Height(u.right)) + 1
v.height ← max(Height(v.left), Height(u)) + 1;
return v
```

Doppelrotation links-rechts

Eingabe: Teilbaum mit Wurzel u.

Rückgabewert: Neue Wurzel des rebalancierten Teilbaums.

```
DoubleRotateLeftRight(u):
u.left ← RotateToLeft(u.left)
return RotateToRight(u)
```

Einfügen in einen AVL-Baum

Eingabe: Wurzel p (wird mögl. geändert), neuer Knoten q.
Rückgabewert: Höhe des Teilbaums.

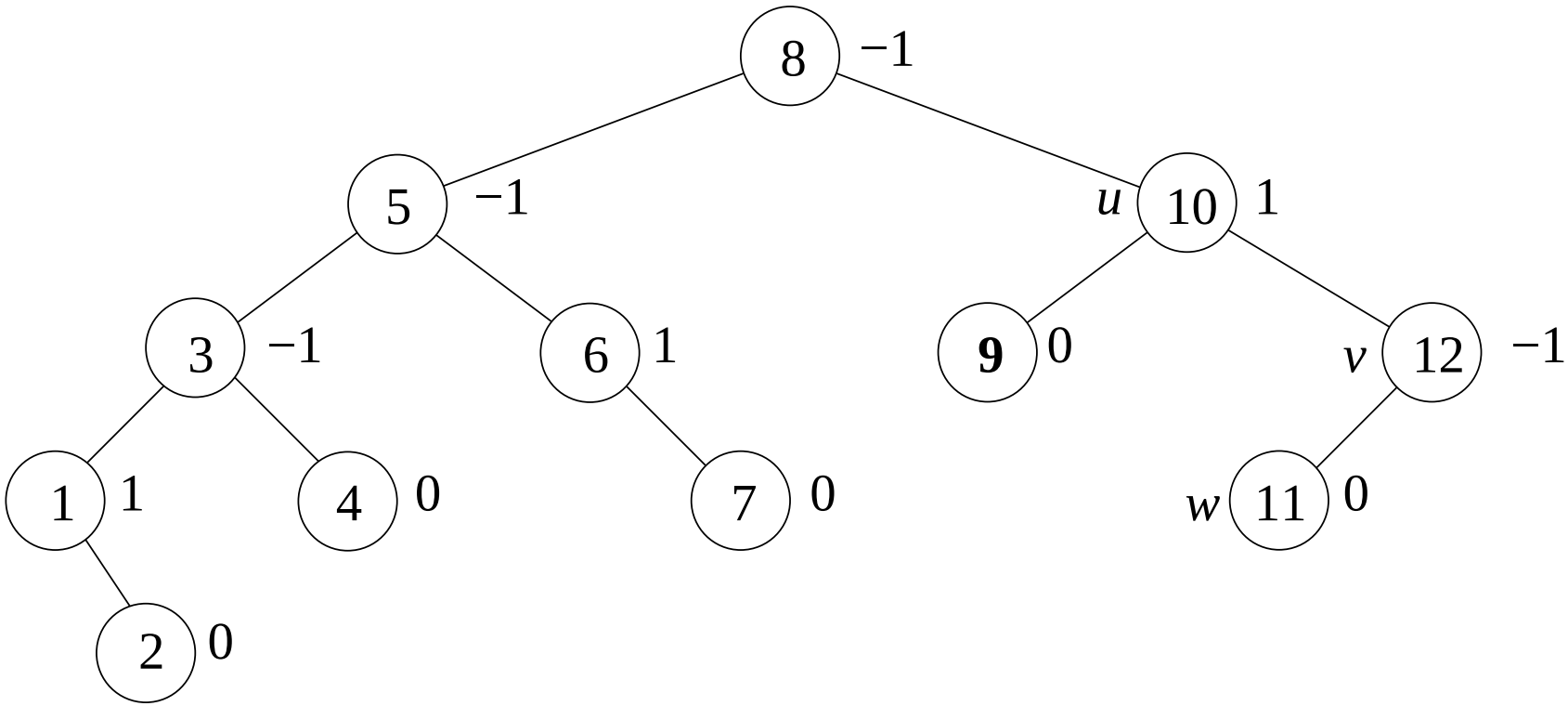
```
Insert(p, q):
if p = null
    p ← q, q.left ← q.right ← null, q.height ← 0
else
    if q.key < p.key
        Insert(p.left, q)
        if Height(p.right) − Height(p.left) = −2
            if Height(p.left.left) ≥ Height(p.left.right)
                p ← RotateToRight(p)
            else
                p ← DoubleRotateLeftRight(p)
    elseif q.key > p.key
        ....
    else
        Knoten q schon vorhanden
p.height ← max(Height(p.left),Height(p.right)) + 1
```

Einfügen in einen AVL-Baum

```
Insert(p, q):
if p = null
    p ← q, q.left ← q.right ← null, q.height ← 0
else
    if q.key < p.key
        ...
    elseif q.key > p.key
        Insert(p.right, q)
        if Height(p.right) − Height(p.left) = 2
            if Height(p.right.right) ≥ Height(p.right.left)
                p ← RotateToLeft(p)
            else
                p ← DoubleRotateRightLeft(p)
    else
        Knoten q schon vorhanden
p.height ← max(Height(p.left),Height(p.right)) + 1
```

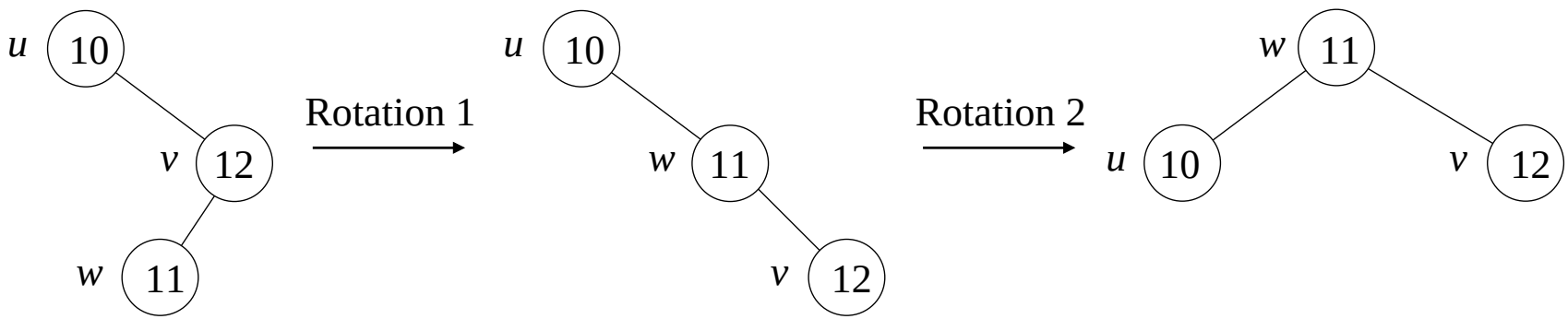
Beispiel: Entfernen eines Elementes aus AVL-Baum

Ausgangssituation: 9 soll entfernt werden.



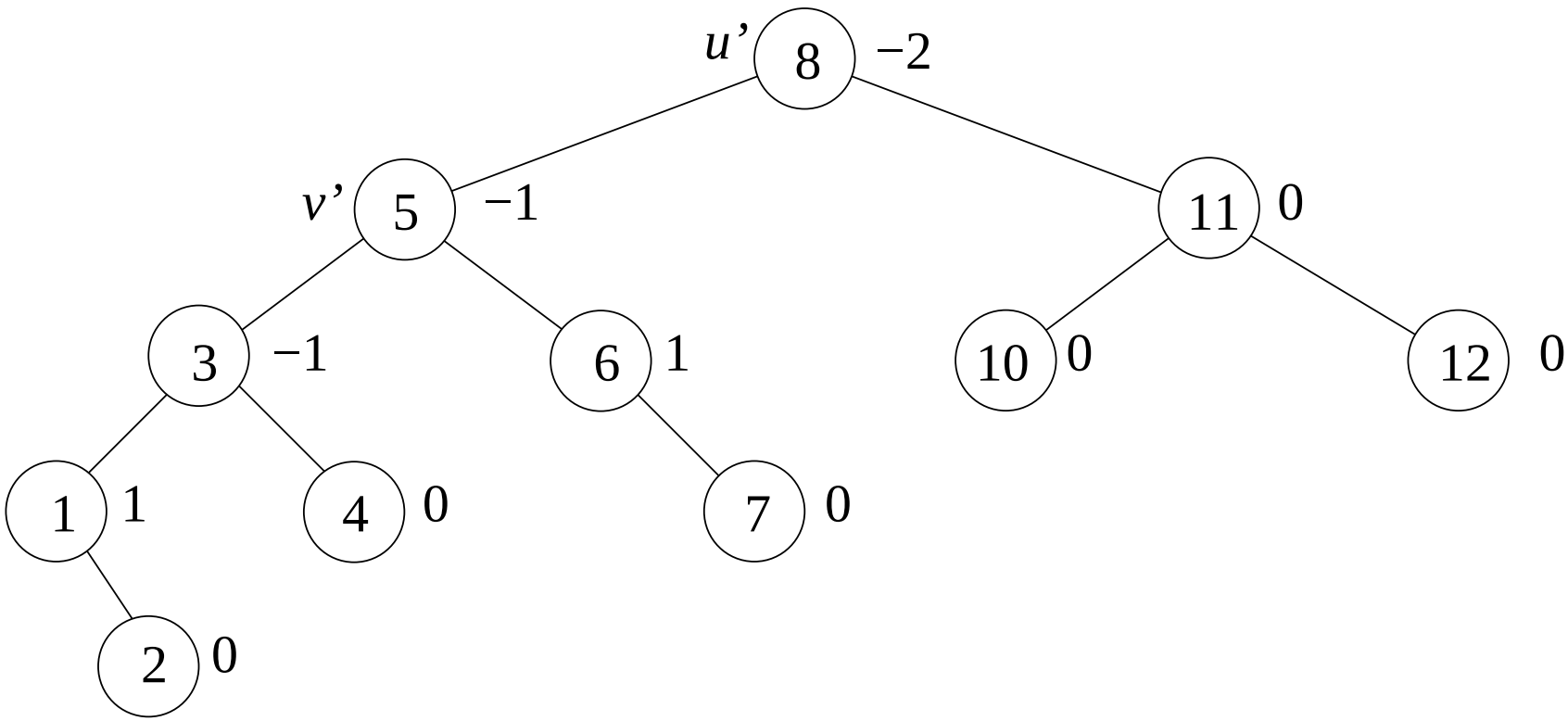
Beispiel: Entfernen eines Elementes aus AVL-Baum

Entfernen: Erfordert Doppelrotation rechts-links.



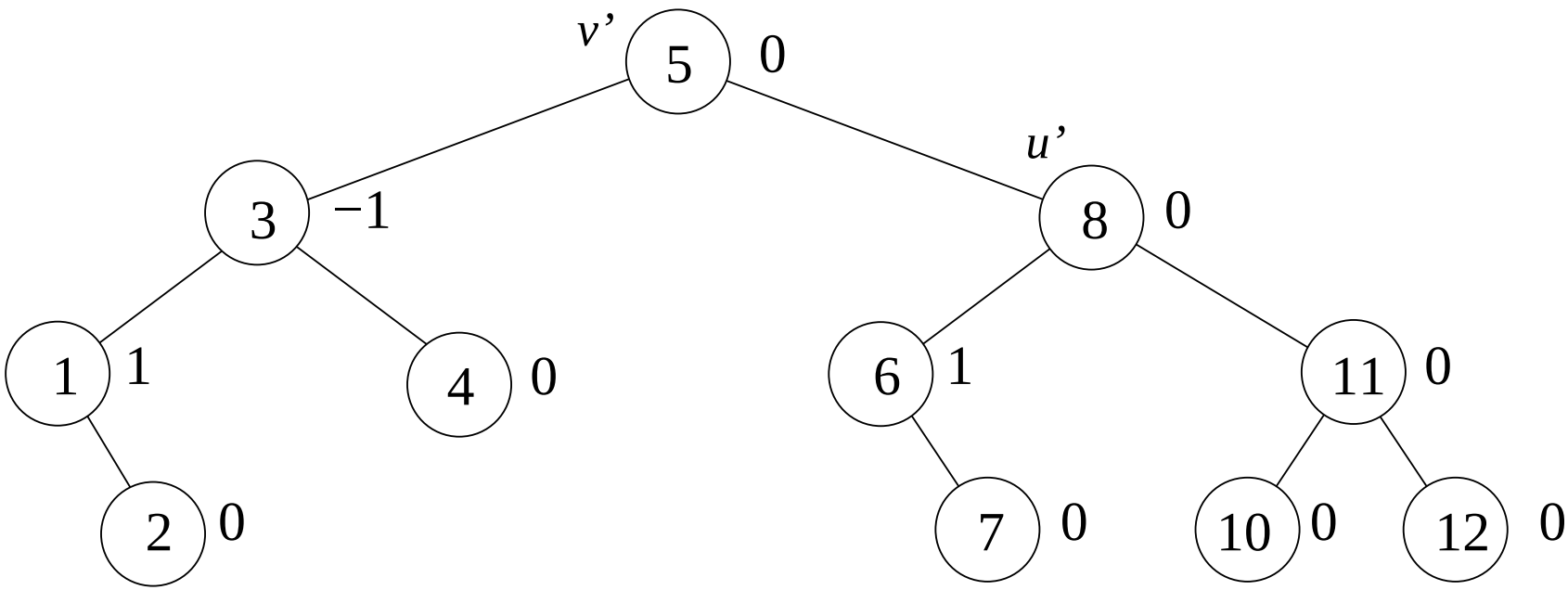
Beispiel: Entfernen eines Elementes aus AVL-Baum

Ergebnis: 9 wurde entfernt, Unterbaum rotiert, Höhe der Wurzel hat sich verändert.



Entfernen eines Elementes aus einem AVL-Baum

Abschluss: Einfache Rotation nach rechts über die Wurzel.



Analyse von AVL-Bäumen

Rotieren: Der Zeitaufwand zum Ausführen einer einzelnen Rotation oder Doppelrotation ist konstant.

Höhe eines Baumes: Die Höhe eine AVL-Baumes mit $n \geq 1$ Schlüsseln ist durch $O(\log n)$ beschränkt.

Worst-Case:

- Beim Einfügen gibt es maximal eine (Doppel-)Rotation.
- Beim Entfernen werden im Worst-Case auf dem Suchpfad von der betroffenen Stelle weg bis zur Wurzel Rotationen bzw. Doppelrotationen durchgeführt.

Aufwand: Die Laufzeit der Operationen Einfügen und Entfernen liegt daher immer in $O(\log n)$.

B-Bäume und B*-Bäume

B-Baum: Motivation

Bisherige Suchbäume: Gut geeignet, wenn sich die Daten im Hauptspeicher befinden (internes Suchen).

Große Datenmengen:

- Bei sehr großen Datenmengen (z.B. Datenbanken) werden die Daten auf Festplatten oder anderen externen Speichern abgelegt.
- Bei Bedarf werden Teile davon in den Hauptspeicher geladen.
- Ein Zugriff auf den externen Speicher benötigt deutlich mehr Zeit als ein Zugriff auf den Hauptspeicher.

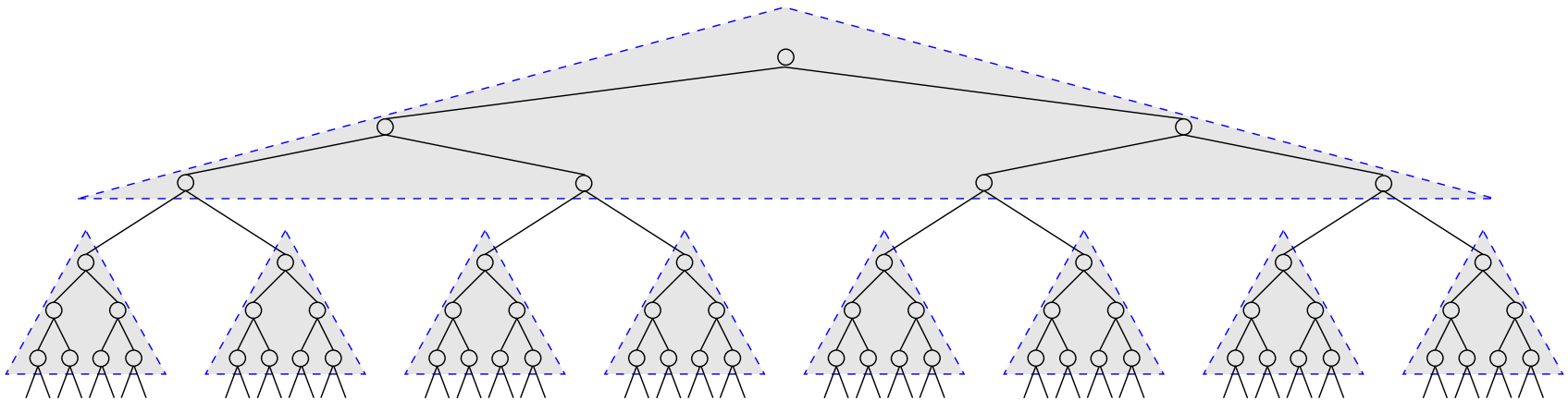
Beispiel: Binärer Baum mit N Datensätzen, extern gespeichert.

- Verweise auf die linken und rechten Unterbäume sind jeweils Adressen auf einem externen Speichermedium.
- Suche nach einem Schlüssel benötigt daher ungefähr $\log_2 n$ externe Zugriffe.
- Bei $N = 10^6$ sind das ca. 20 externe Speicherzugriffe.

B-Baum: Motivation

Annahme: Externe Speichermedien weisen i.A. eine blockorientierte Struktur auf. Bei einem externen Speicherzugriff wird immer eine gesamte Speicherseite (*page, block*) gelesen oder geschrieben.

Grundsätzliche Idee: „Zusammenfassen mehrerer Knoten“, sodass jeweils eine Speicherseite bestmöglich genutzt wird.



Vorteil: Weniger Zugriffe auf Speicherseiten notwendig.

Beispiel:

- Baum mit $N = 10^6$ Schlüssel, 128 Schlüssel pro Speicherseite.
- Man benötigt nur 3 externe Speicherzugriffe.

B-Baum: Definition

B-Bäume: Siesind eine Verallgemeinerung von binären Suchbäumen und setzen die Idee der Gruppierung in Speicherseiten in die Praxis um.

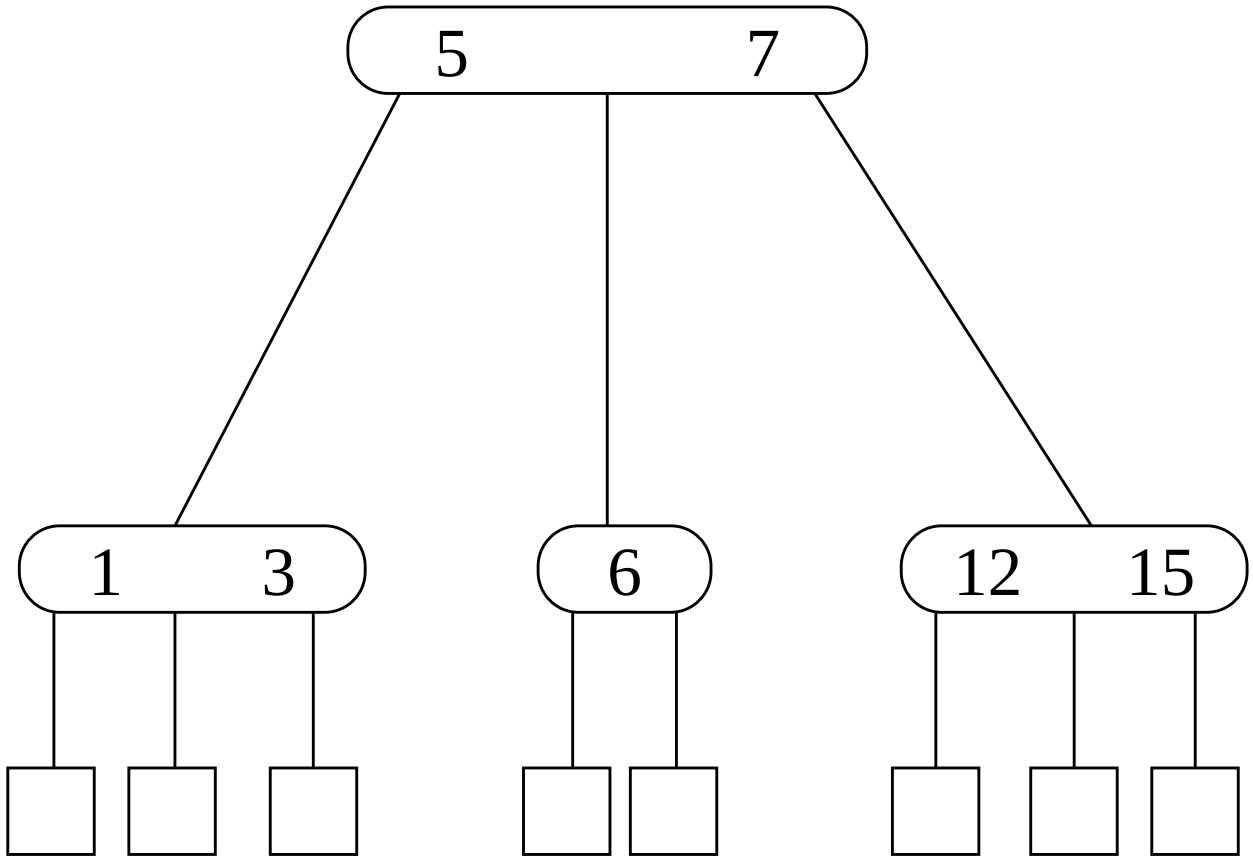
Definition: B-Baum der Ordnung m (Bayer und McCreight, 1972)

1. Alle Blätter haben gleiche Tiefe und sind leere Knoten.
2. Jeder innere Knoten außer der Wurzel hat mindestens $\lceil \frac{m}{2} \rceil$ Kinder. Die Wurzel hat mindestens 2 Kinder.
3. Jeder Knoten hat höchstens m Kinder.
4. Jeder Knoten mit l Schlüssel hat $l + 1$ Kindern.
5. Für jeden Knoten mit Schlüsseln $s_1, \ldots, s_l$ und Kindern $v_0, \ldots, v_l$ gilt: $\forall i = 1, \ldots, l$:
   - Alle Schlüssel in $T_{v_{i-1}}$ sind kleiner gleich s_i, und
   - s_i ist kleiner gleich allen Schlüsseln in T_{v_i}.

   (T_{v_i} bezeichnet den Teilbaum mit Wurzel v_i.)

B-Baum: Beispiel

Beispiel: B-Baum der Ordnung $m = 3$ (2-3 Baum) mit 7 Schlüsseln und 8 Blättern.



Implementierung eines B-Baum-Knotens

Implementierung:

- $p.l$ – Anzahl der Schlüssel
- $p.key[1], \ldots, p.key[l]$ – Schlüssel $s_1, \ldots s_l$
- $p.info[1], \ldots, p.info[l]$ – Datenfelder zu Schlüsseln $s_1, \ldots, s_l$
- $p.child[0], \ldots, p.child[l]$ – Verweis auf Kinderknoten $v_0, \ldots, v_l$

Blätter: Diese markieren wir hier durch $p.l = 0$.

Anmerkung: In einer realen Implementierung brauchen die leeren Blätter nicht explizit gespeichert zu werden. Uns erleichtern sie hier aber die Definitionen und Überlegungen.

Suche

Eingabe: B-Baum mit Wurzel p und ein Schlüssel x.

Rückgabewert: Knoten mit Schlüssel x oder $null$, falls x nicht vorhanden ist.

```
Search(p, x):
i ← 1
while i ≤ p.l und x > p.key[i]
    i ← i + 1
if i ≤ p.l und x = p.key[i]
    return (p, i)
if p.l = 0
    return null
else
    return Search(p.child[i − 1], x)
```

Einfügen

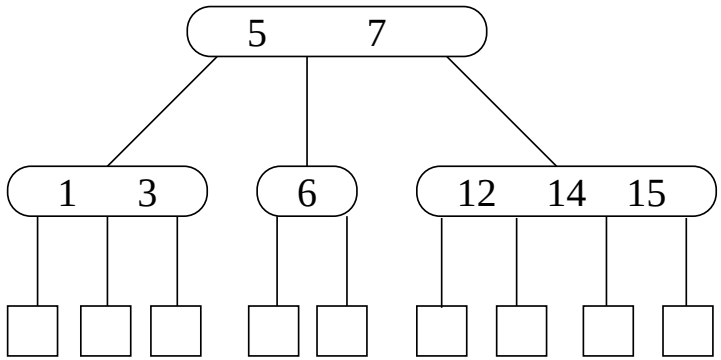
Einfügen im B-Baum:

- Schlüssel suchen $\rightarrow$ endet in Blatt p_0
- Sei p Vorgänger von p_0 und $p.child[i]$ zeigt auf p_0
- Schlüssel zwischen s_i und s_{i+1} in p einfügen, neues Blatt erzeugen
- Wenn $p.l < m$: fertig
  sonst: p splitten
  - p' : $s_1, \ldots, s_{\lceil m/2 \rceil - 1}$ $\quad$ p'' : $s_{\lceil m/2 \rceil + 1}, \ldots, s_m$
  - Schlüssel $s_{\lceil m/2 \rceil}$ in Vorgänger von p einfügen

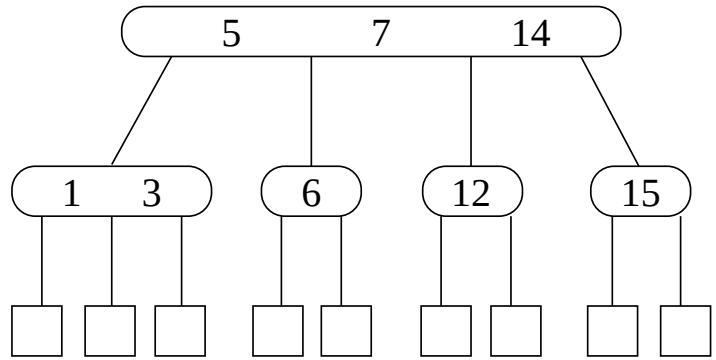
Anmerkung: B-Bäume wachsen immer nur an der Wurzel!

Einfügen: Beispiel

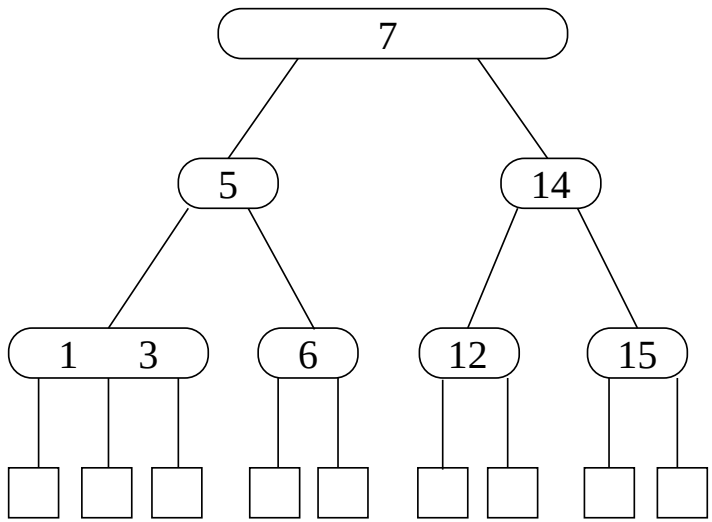
Beispiel: Einfügen von Schlüssel 14.



(a)



(b)



(c)

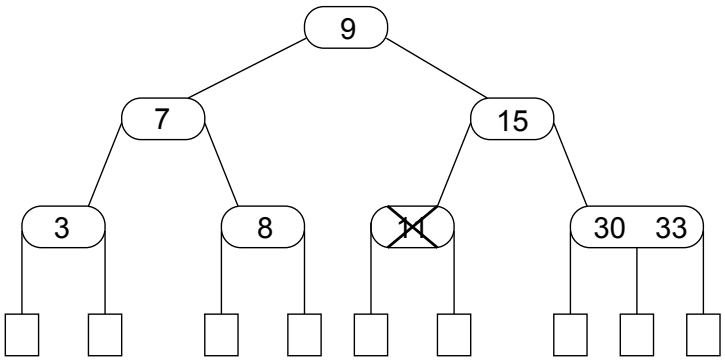
Entfernen

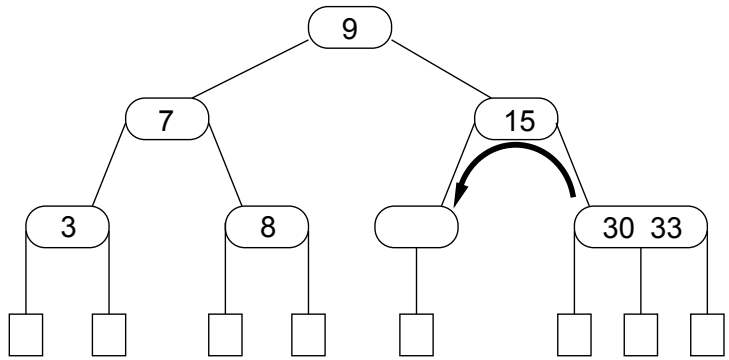
Entfernen aus einem B-Baum:

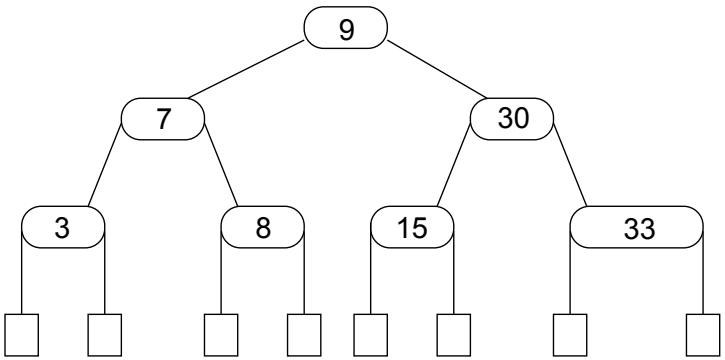
- Schlüssel suchen, mit Blatt entfernen
  - Falls Schlüssel nicht in unterster Ebene:
    mit Ersatzschlüssel aus unterster Ebene tauschen
    (kleinster im rechten Unterbaum oder größter im linken)
- Falls nun der Knoten zu wenige Schlüssel hat:
  - Übernehme Schlüssel von Geschwisterknoten
  - oder verschmelze mit Geschwisterknoten

Entfernen: Beispiel

Beispiel: 11 aus B-Baum (Ordnung 3) entfernen.

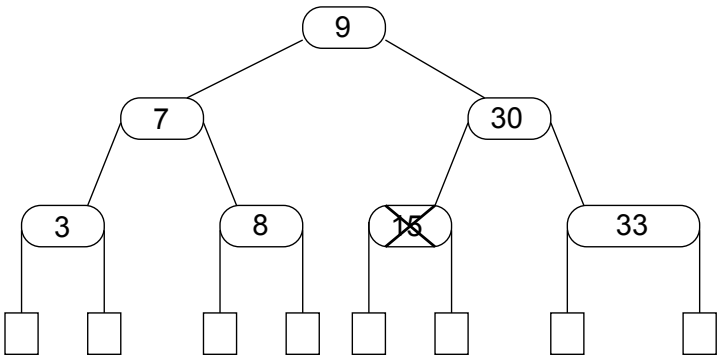


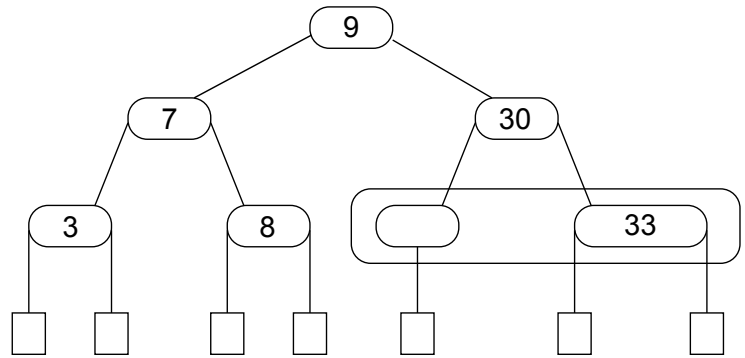


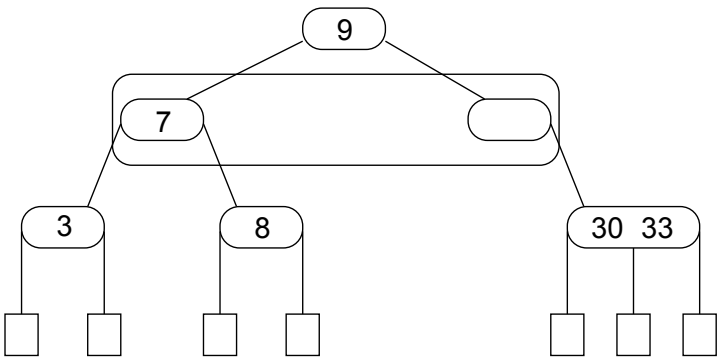


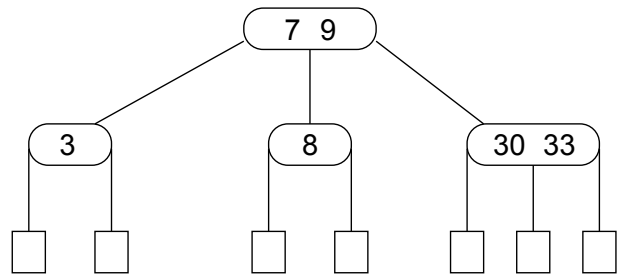
Entfernen: Beispiel

Beispiel: 15 aus B-Baum (Ordnung 3) entfernen.









Eigenschaften von B-Bäumen

Lemma: Die Anzahl der Blätter in einem B-Baum T ist immer um 1 größer als die Anzahl der Schlüssel.

Beweis: mit Induktion über die Höhe h

- Gilt für $h = 1$: Wurzel mit k Blättern, $k - 1$ Schlüssel, $2 \leq k \leq m$
- Annahme: Theorem gilt für Höhe h.
- Wir zeigen, dass es auch für Höhe $h + 1$ gilt:
  In einem B-Baum mit Höhe $h + 1$ gibt es
  k Unterbäume $T_1, \ldots, T_k$ mit Höhe h mit $n_1, \ldots, n_k$ Blättern.
  Diese haben $(n_1 - 1), \ldots, (n_k - 1)$ Schlüssel.
  Wurzel: $k - 1$ Schlüssel
  $\rightarrow$ Insgesamt gibt es $\sum_{i=1}^{n} n_i$ Blätter und
  $\sum_{i=1}^{n}(n_i - 1) + (k - 1) = \sum_{i=1}^{k} n_i - 1$ Schlüssel.

Eigenschaften von B-Bäumen

- Die minimale Anzahl der Blätter $N_{\min}$ wird erreicht, wenn die Wurzel nur 2 und jeder andere innere Knoten nur $\lceil m/2 \rceil$ Nachfolger hat, d.h. $N_{\min} = 2\lceil m/2 \rceil^{h-1}$.

- Die maximale Anzahl der Blätter $N_{\max}$ wird erreicht, wenn jeder innere Knoten die maximal mögliche Anzahl m von Nachfolger hat, d.h. $N_{\max} = m^h$.

Eigenschaften von B-Bäumen

Korollar: Für die Höhe h eines B-Baumes mit N Schlüsseln und $(N+1)$ Blättern muss gelten:

$$N_{\min} = 2\lceil m/2 \rceil^{h-1} \quad \leq \quad (N+1) \quad \leq \quad m^h = N_{\max}$$

und somit auch

$$\log_m(N+1) \quad \leq \quad h \quad \leq \quad 1 + \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)$$

D.h., wir haben gezeigt, dass B-Bäume immer eine Höhe $h = \Theta(\log N)$ haben, und somit sind die Operationen Suchen, Einfügen und Entfernen auch wieder effizient in $\Theta(\log N)$ Zeit durchführbar.

B*-Bäume

Variante von B-Bäumen

Definition:

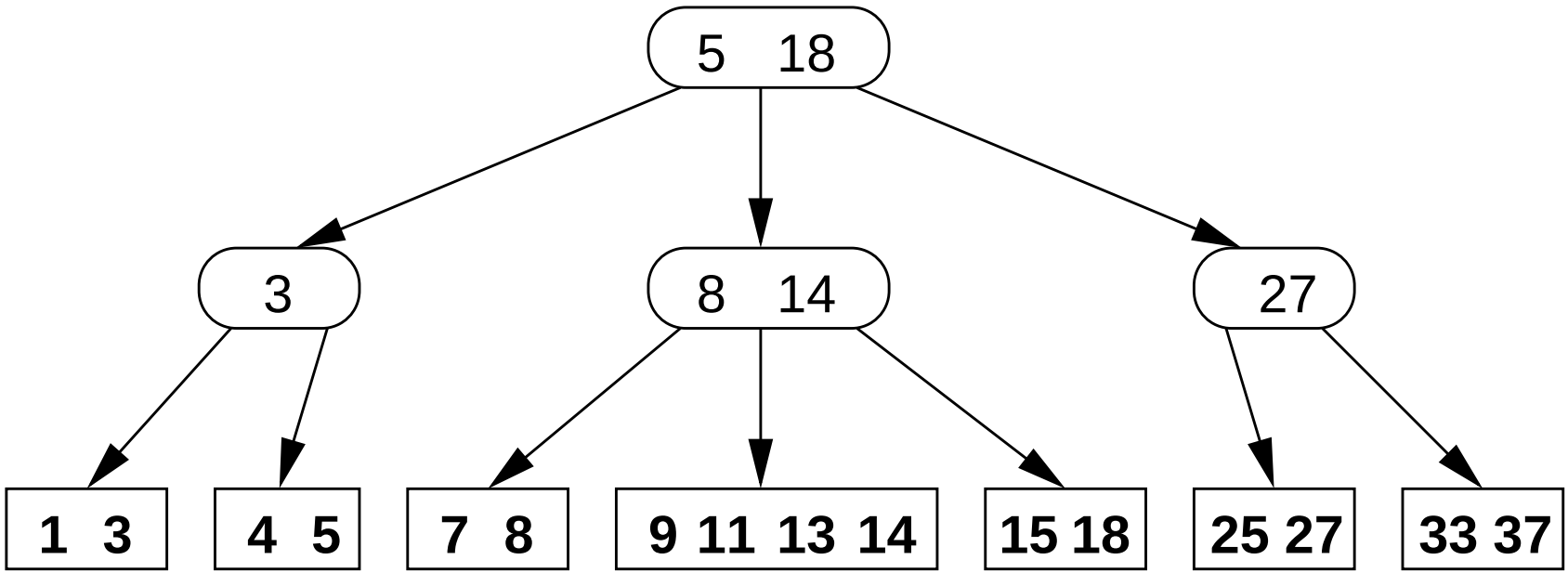
- Komplette Datensätze nur in Blättern:
  k^* bis $2k^*$ Datensätze, keine Verweise
  (k^* ist ein von m unabhängiger Parameter)

- In Zwischenknoten nur Schlüssel und Verweise
  Schlüssel: immer der des größten Elements im vorangehenden Unterbaum

Motivation und Konsequenz: m kann größer gewählt werden,
T hat geringere Höhe!

Beispiel zu B*-Bäumen

Beispiel: Ein B*-Baum mit $m = 3$ und $k^* = 2$



B*-Bäume

- Suchen: Man muss im Unterschied zum B-Baum in jedem Fall bis zu einem Blatt gehen. Das erhöht die mittlere Anzahl von Schritten jedoch kaum, zumal der B*-Baum i.A. ja auch geringere Höhe hat.
- Einfügen: Das Prinzip ist das gleiche wie beim B-Baum. Kleine Unterschiede ergeben sich dadurch, dass es nur einen trennenden Schlüssel (keine trennenden Datensätze) gibt.
- Entfernen: Der zu entfernende Datensatz liegt immer in einem Blatt. Verweise in darüberliegenden Zwischenknoten müssen ggfs. aktualisiert werden.