Algorithmen und Datenstrukturen

Teil 3: Datenstrukturen

Martin Hofmann

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. April 2016

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 3: Datenstrukturen

Dynamische Mengen

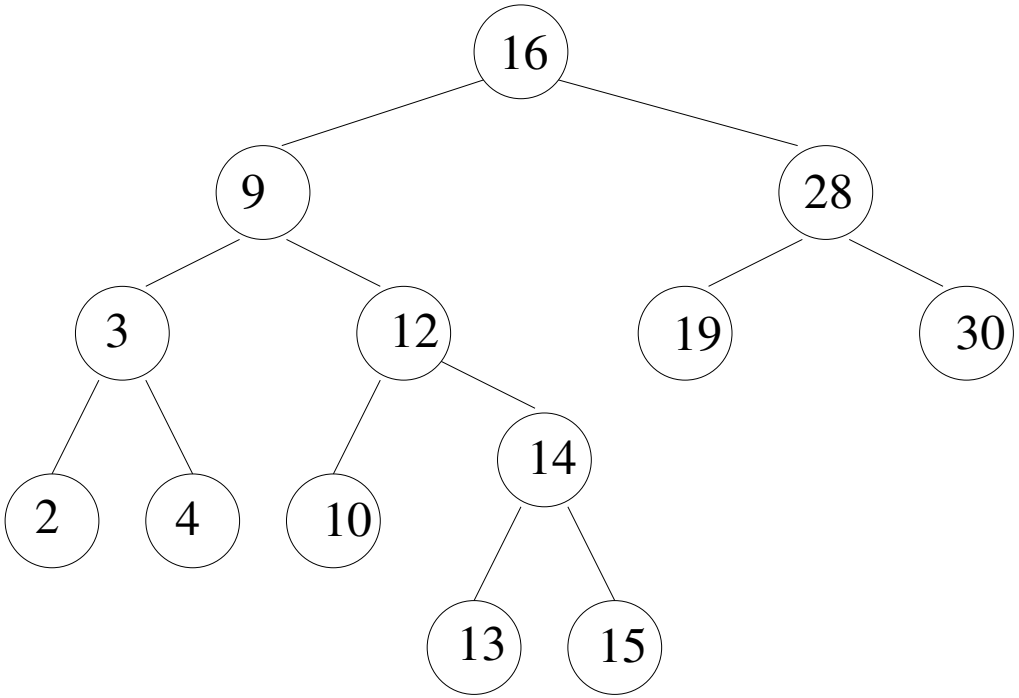
Eine **dynamische Menge** ist eine Datenstruktur, die Objekte verwaltet, welche einen Schlüssel tragen, und zumindest die folgenden Operationen unterstützt:

- Search(S, k): liefert (einen Zeiger auf) ein Element in S mit Schlüssel k, falls ein solches existiert; Nil sonst.
- Insert(S, x): fügt das Element (bezeichnet durch Zeiger) x in die Menge S ein.

Oft werden weitere Operationen unterstützt, wie etwa Löschen, Maximum, Minimum (bei geordneteten Schlüsseln).
Typische Anwendung: Symboltabelle in einem Compiler.
Schlüssel = Bezeichner, Objekte = (Typ, Adresse, Größe, ...)

Realisierungen durch Bäume

Dynamische Mengen können durch **binäre Suchbäume** (BST) realisiert werden.



Für jeden Knoten mit Eintrag x gilt:

- Die Einträge des **linken Unterbaumes** sind $\leq x$;
- Die Einträge des **rechten Unterbaumes** sind $\geq x$;

Notation für binäre Bäume

- **Wurzel** des Baumes T gespeichert in $root[T]$.
- Jeder Knoten x hat Zeiger auf seinen Eintrag $elem[x]$, die **Kinder** $left[x]$ und $right[x]$, sowie auf den **Elternknoten** $p[x]$ (ggf. nil).

Die Operationen

- Tree-Search(T, k)
- Tree-Insert(T, x)
- Tree-Delete(T, x)
- Tree-Minimum(T) und Tree-Maximum(T)
- Tree-Successor(T, x) und Tree-Predecessor(T, x)

haben sämtlich eine Laufzeit von $O(h)$, wobei h die Tiefe des Baumes T ist.

Wiederholung: Operationen auf binären Suchbäumen

Einfügen in BST

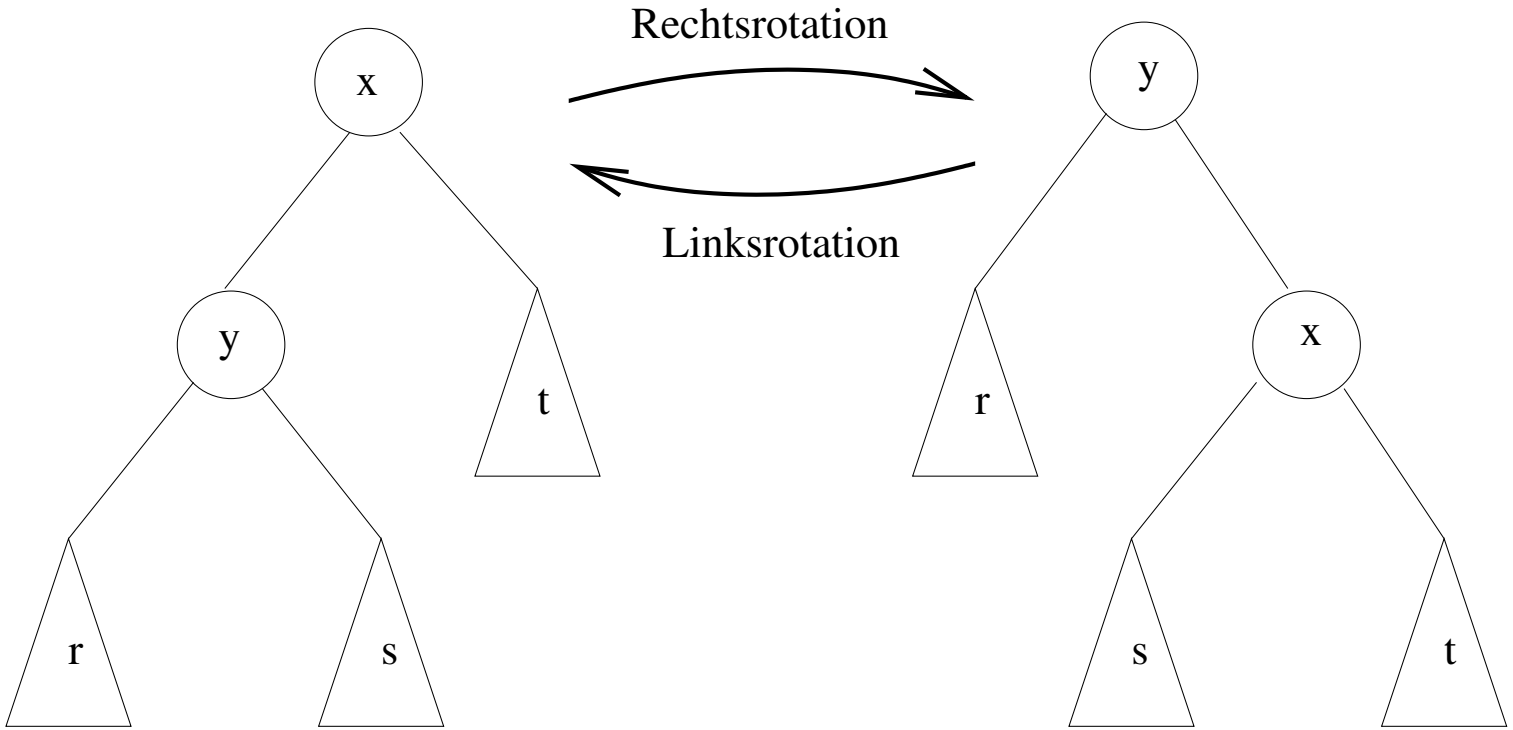
Man fügt einen neuen Eintrag als Blatt an passender Stelle ein. Im Beispiel würde die 17 als linkes Kind der 19 eingefügt.

Löschen aus BST

Blätter und Knoten mit nur einem Kind werden "einfach so" gelöscht.
Um einen Eintrag zu löschen, der sich in einem Knoten mit zwei Kindern befinden, ersetzt man den Eintrag durch den eines Knotens mit nur einem oder gar keinem Kind und löscht dann letzteren Knoten.
Im Beispiel löscht man die 12, indem man sie mit der 13 überschreibt und dann den Knoten mit der 13 entfernt.

Balancierte binäre Suchbäume

Damit die Operationen auf binären Suchbäumen effizient sind, sollten diese **balanciert** sein, d.h. die Höhe eines Baumes mit n Knoten ist $O(\log n)$. (Balancierung ist eine Eigenschaft einer Familie von Bäumen.)
Das Balancieren erfolgt durch **Rotationsoperationen:**



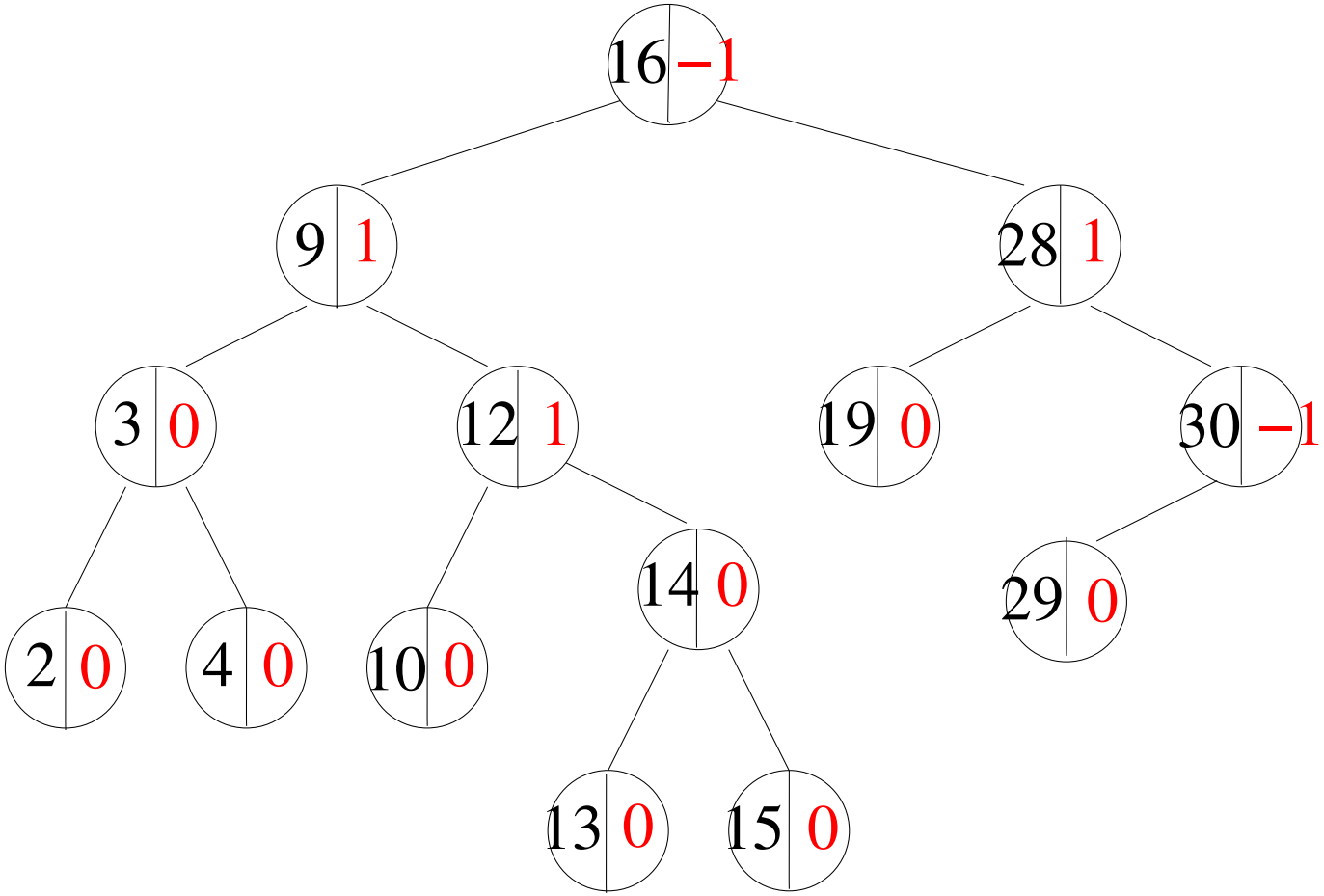
AVL-Bäume

AVL-Baum

Ein AVL-Baum ist ein binärer Suchbaum in dem für jeden Knoten gilt: die Höhen der beiden Unterbäume (links und rechts) unterscheiden sich um höchstens eins. Die Höhendifferenz ist also -1, 0, oder 1.

- In der Regel wird die Höhendifferenz $(-1, 0, 1)$ der Unterbäume in jedem Knoten gespeichert und bei Modifikationen entsprechend aktualisiert. Diese Zusatzinformationen heißen dann **Balancefaktoren** (BF).
- Oft bezeichnet man AVL-Bäume auch als balancierte Bäume.

Beispiel



Ein AVL-Baum. Die Balancefaktoren sind in Rot eingetragen.

Grösse der AVL-Bäume

Sei $N(h)$ die minimale Zahl von Knoten eines AVL-Baums der Höhe h.
Das Minimum erreicht man, wenn man nie BF=0 hat. Es gilt also:

$$N(0) = 1$$
$$N(1) = 2$$
$$N(h+2) = N(h+1) + N(h) + 1$$

Höhe der AVL-Bäume

Man sieht: $N(h) \geq F(h)$, wobei

$$F(0) = 1$$
$$F(1) = 1$$
$$F(h+2) = F(h+1) + F(h)$$

Es gilt aber $F(h) = \Omega(\phi^h)$ mit $\phi = (\sqrt{5}+1)/2 = 1,618\ldots$ (Fibonaccizahlen). Es folgt: $N(h) = \Omega(\phi^h)$.
Für die maximale Höhe $h(n)$ eines AVL-Baums mit n Knoten gilt dann $h(n) = \max\{h \mid N(h) \leq n\}$, also $h(n) = O(\log(n))$.

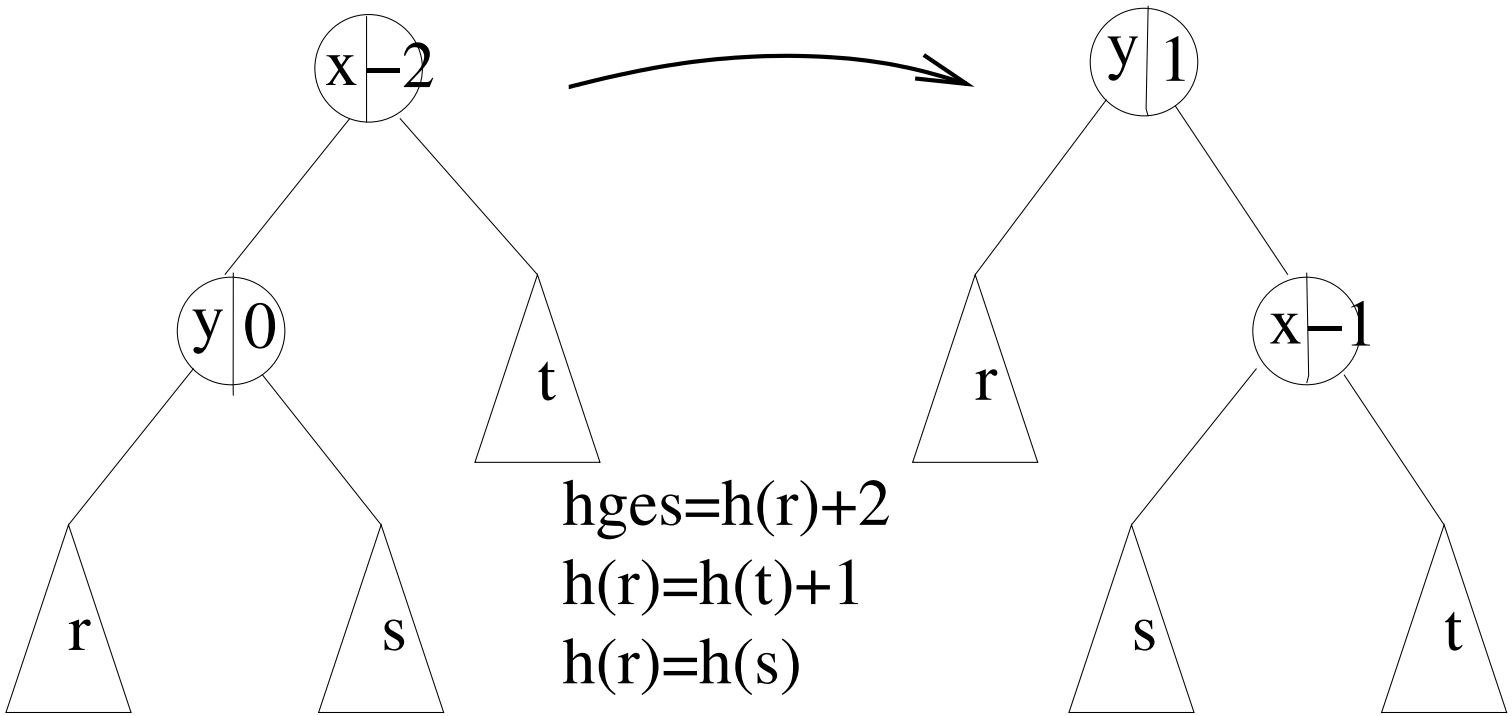
Satz

Die Höhe eines AVL-Baumes mit n Knoten ist $O(\log(n))$, d.h. AVL-Bäume sind balanciert.

Einfügen und Löschen in AVL-Bäumen

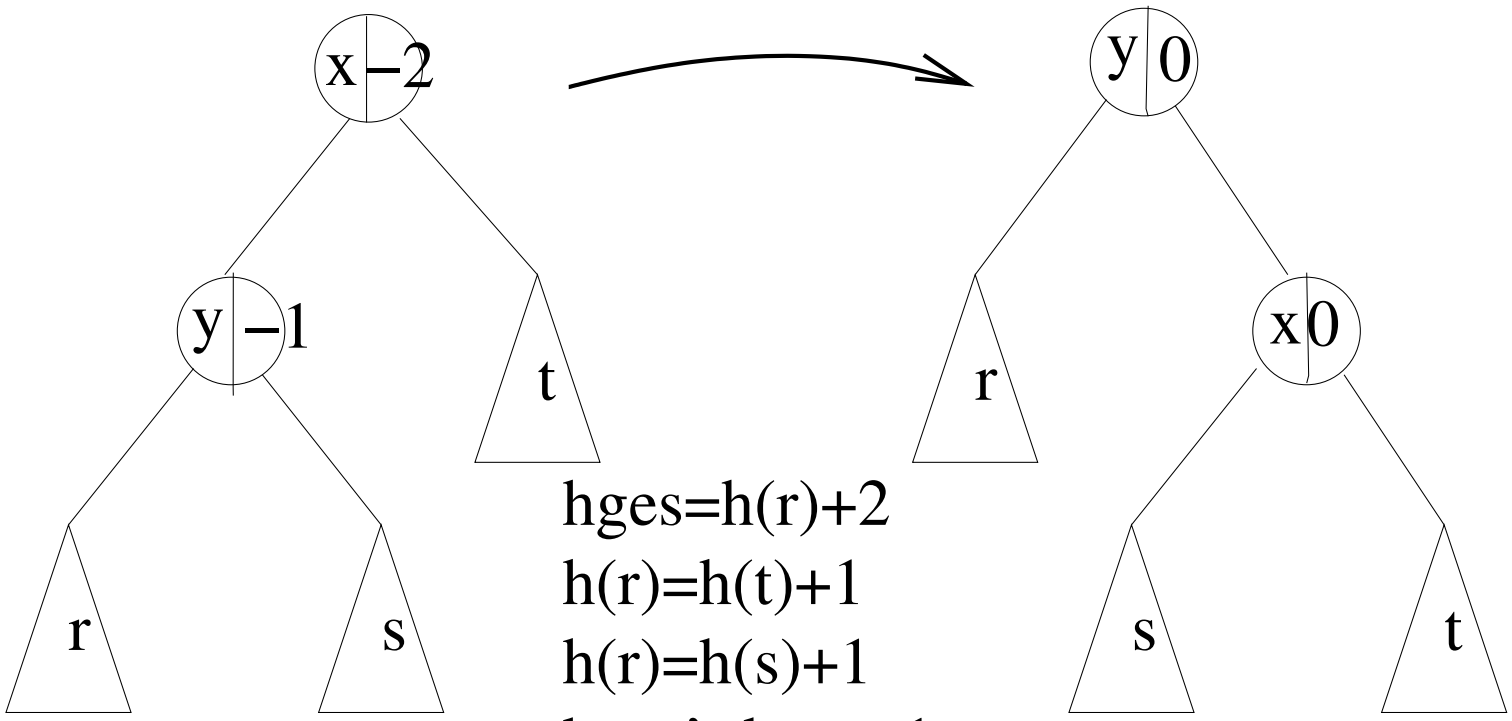
Einfügen und Löschen von Knoten bei AVL-Bäumen erfolgt zunächst wie bei binären Suchbäumen.
Sodann wird die Balance durch systematisches Rotieren wiederhergestellt.
Die nächsten drei Folien zeigen, wie ein Knoten mit BF -2 durch Rotation "repariert" werden kann (BF=2 ist symmetrisch).
Die Höhe des entsprechenden Teilbaumes kann sich dabei um eins ändern, was dann zu BF von -2 oder 2 weiter oben führt. Diese werden dann durch abermalige Rotationen nach demselben Muster repariert.

BF = -2. Erster Fall



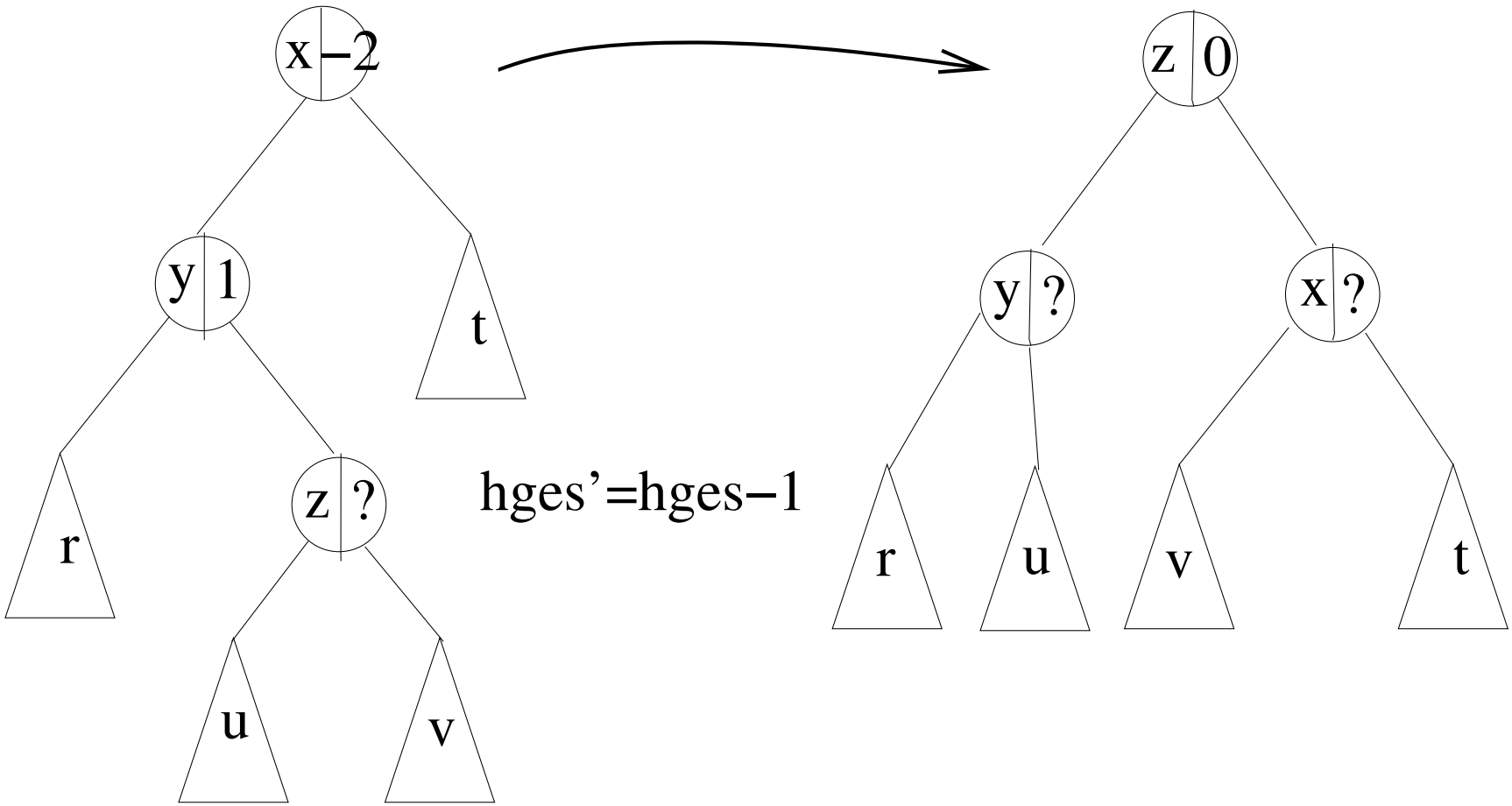
hges=h(r)+2
h(r)=h(t)+1
h(r)=h(s)
hges'=hges

BF = -2. Zweiter Fall



hges=h(r)+2
h(r)=h(t)+1
h(r)=h(s)+1
hges'=hges−1

BF = -2. Dritter Fall



Zusammenfassung

- AVL-Bäume speichern in jedem Knoten den Höhenunterschied (BF) der beiden Unterbäume. Dieser muss 0,1, oder -1 sein.
- Die Höhe eines AVL-Baums mit n Knoten ist $O(\log(n))$.
- Die Laufzeit aller Operationen ist proportional zur Höhe, also $O(\log(n))$.
- Einfügen erfordert maximal 3 Rotationen, Löschen kann $\Omega(\log(n))$ viele Rotationen benötigen (Ohne Begründung).

B-Bäume

B-Bäume verallgemeinern binäre Suchbäume dahingehend, dass in einem Knoten mehrere Schlüssel stehen und ein Knoten mehrere Kinder hat. (Typischerweise jeweils 500-1000).
Dadurch sinkt die Zahl der Knoten, die bei einem Suchvorgang besucht werden müssen, dafür ist aber jedes einzelne Besuchen aufwendiger.
Das ist sinnvoll, wenn die Knoten auf einem Massenspeichermedium abgelegt sind (Plattenstapel o.ä.) wo einzelne Zugriffe recht lange dauern, dafür aber gleich eine ganze *Seite* (z.B.: 1024Byte) auslesen, bzw. schreiben.

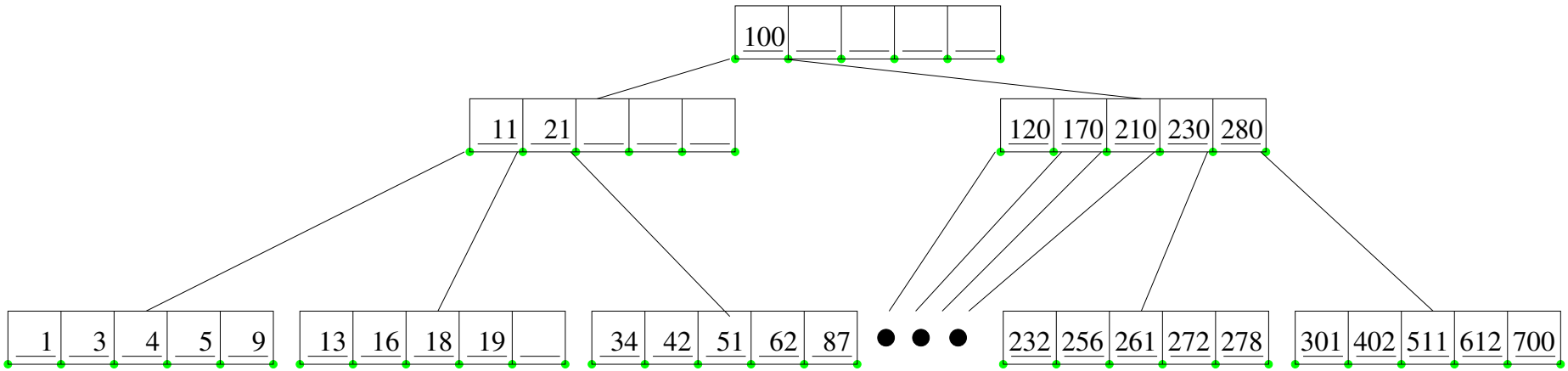
B-Bäume: Definition

Die Definition von B-Bäumen bezieht sich auf ein festes $t \in \mathbb{N}$, z.B.: $t = 512$.

Ein **B-Baum** ist ein Baum mit den folgenden Eigenschaften:

- Jeder Knoten x enthält die folgenden Einträge:
  - die Anzahl $n[x]$ der im Knoten gespeicherten Schlüssel, wobei $n[x] \leq 2t - 1$.
  - die in aufsteigender Reihenfolge gespeicherten Schlüssel: $key_1[x] \leq key_2[x], \ldots, key_{n[x]}[x]$
- Jeder Knoten x ist entweder ein Blatt oder ein innerer Knoten und hat dann gerade $n[x] + 1$ Kinder.
- Alle Blätter liegen auf derselben Tiefe.
- Für alle Knoten x außer der Wurzel gilt $n[x] \geq t - 1$.
- Alle Blätter liegen auf derselben Tiefe.
- Alle Schlüssel, die sich im oder unterhalb des i-ten Kindes eines Knotens x befinden, liegen bzgl. der Ordnung zwischen $key_{i-1}[x]$ und $key_i[x]$. Grenzfall $i = 1$: die Schlüssel sind kleiner als $key_1[x]$; Grenzfall $i = n[x]$: die Schlüssel sind größer

Beispiel mit $t = 3$



Die Wurzel hat $n[\,] = 1$.
Der erste Knoten der zweiten Schicht hat $n[\,] = 3$.
Der zweite Knoten der zweiten Schicht hat $n[\,] = 5$. Nur zwei seiner sechs Kinder sind dargestellt.

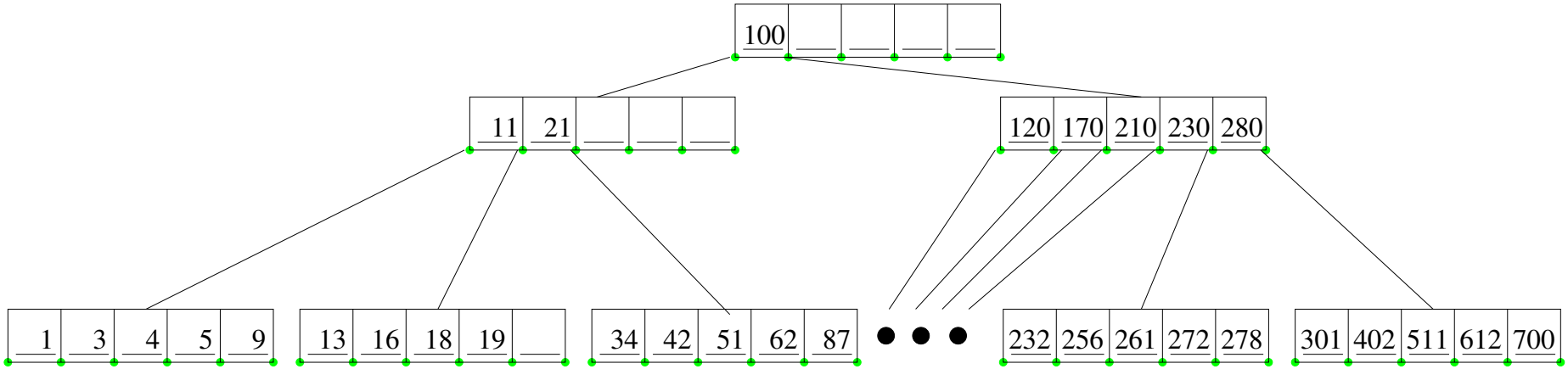
Suchen in B-Bäumen

Beim Suchen vergleicht man den gesuchten Schlüssel k mit den Einträgen der Wurzel. Entweder ist k unter den Wurzeleinträgen, oder man ermittelt durch Vergleich mit den Wurzeleinträgen denjenigen Unterbaum, in welchem k, wenn überhaupt, zu finden ist und sucht in diesem rekursiv weiter.
Natürliche Verallgemeinerung des Suchens in BST.

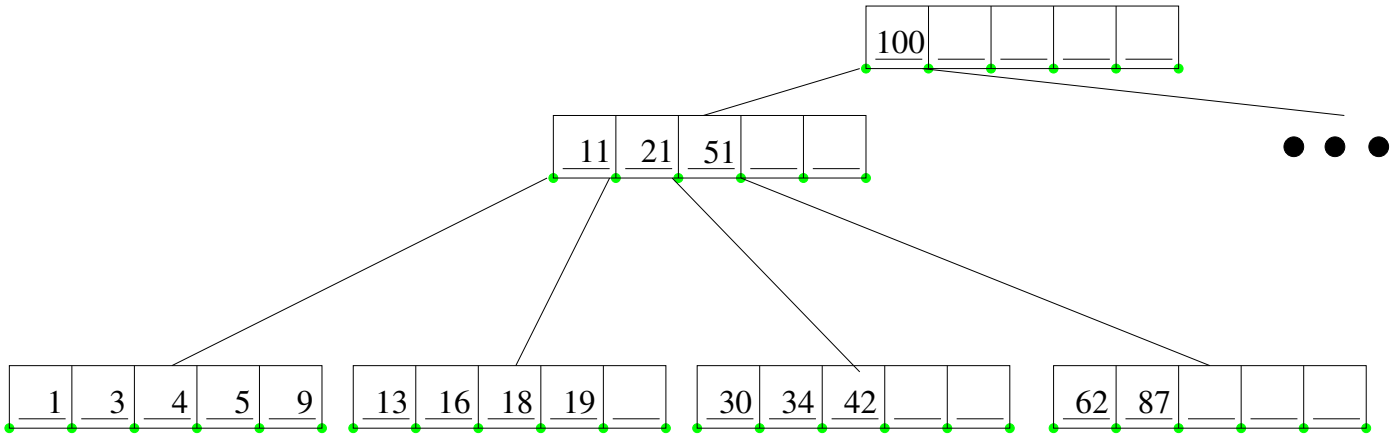
Einfügen in B-Bäumen

1. Bestimme durch Suche das Blatt x, in welches der neue Schlüssel k aufgrund seiner Ordnungsposition gehört.
2. Falls dort noch nicht zuviele Einträge (also $n[x] < 2t - 1$) vorhanden sind, so füge den neuen Schlüssel dort ein.
3. Andernfalls füge k trotzdem in das Blatt ein (es enthält dann $2t$ Schlüssel) und teile es in zwei Blätter der Größe t und $t - 1$ auf. Der übrige Schlüssel wird dem Elternknoten als Trenner der beiden neuen Kinder hinzugefügt.
4. Führt dies zum Überlauf im Elternknoten, so teile man diesen ebenso auf, etc. bis ggf. zur Wurzel.

Einfügen des Schlüssels 30



wird zu



Löschen aus *B*-Bäumen

1. Zunächst ersetzt man ähnlich wie bei BST den zu löschenden Schlüssel durch einen passenden Schlüssel aus einem Blatt. Dadurch kann man sich auf das Entfernen von Schlüsseln aus Blättern beschränken.
2. Kommt es beim Entfernen eines Schlüssels aus einem Blatt b zum Unterlauf (weniger als $t-1$ Schlüssel), hat aber der rechte Nachbar b' mehr als $t-1$ Schlüssel, so gibt man den Schlüssel aus dem Elternknoten, der b von b' trennt, zu b hinzu und nimmt den kleinsten Schlüssel in b' als neuen Trenner.
3. Hat der rechte Nachbar b' von b auch nur $t-1$ Schlüssel, so verschmilzt man b und b' samt dem Trenner im Elternknoten zu einem neuen Knoten mit nunmehr $2t-2$ Schlüsseln. Dem Elternknoten geht dadurch ein Schlüssel verloren, was wiederum zum Unterlauf führen kann. Ein solcher ist durch rekursive Behandlung nach demselben Schema zu beheben.

Mögliche Optimierung

Cormen und auch die Wikipedia empfehlen, beim Einfügen bereits beim Aufsuchen der entsprechenden Blattposition solche Knoten, die das Höchstmaß $2t - 1$ erreicht haben, vorsorglich zu teilen, damit nachher ein Überlauf lokal behandelt werden kann und nicht wieder rekursiv nach oben gereicht werden muss.
Analog würde man beim Löschen Knoten der Mindestgröße $t - 1$ durch Ausgleichen vom Nachbarn oder, falls das nicht möglich, Verschmelzen, vorsorglich verdicken.
Man nimmt hierdurch mehr Spaltungen / Verschmelzungen vor, als unbedingt nötig, spart sich dafür aber Knotenzugriffe. Was im Einzelfall besser ist, ist unklar.

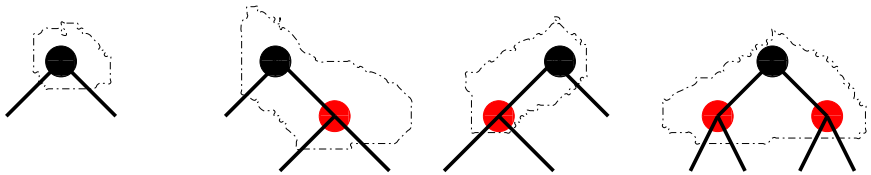
Rot-Schwarz-Bäume

- Ein **Rot-Schwarz-Baum** ist ein binärer Suchbaum, dessen Knoten ein zusätzliches Feld $color[x]$ haben, mit den Werten red und black. Wir sprechen von **roten** und **schwarzen** Knoten.
- Man stellt sich alle Knoten eines Rot-Schwarz als innere Knoten vor, deren Kinder ggf. gedachte **schwarze** Blätter sind.
- Die Wurzel (und alle Blätter) eines Rot-Schwarz Baumes sind schwarz.
- Beide Kinder eines roten Knotens sind schwarz.
- Für jeden Knoten x gilt:

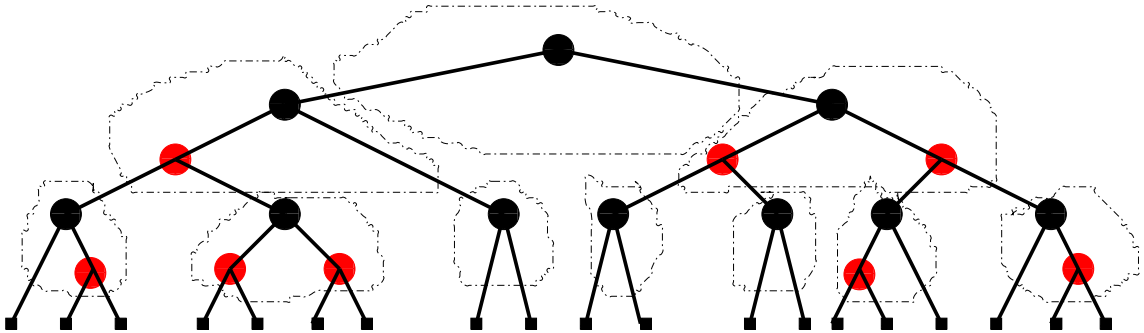
  jeder Pfad von x zu einem Blatt enthält gleich viele schwarze Knoten.

Rot-Schwarz-Bäume als 2-3-4-Bäume

Man kann sich einen Rot-Schwarz-Baum als B-Baum mit $t = 2$ vorstellen, dessen Knoten ("Makro-Knoten") als System aus einem schwarzen und 0,1 oder 2 roten Knoten repräsentiert werden. Im Falle eines roten Knotens gibt es dann zwei Varianten:



Die vier Typen von "Makro–Knoten"



Ein Rot–Schwarz–Baum mit 18 Eintraegen, 10 Makro–Knoten und 19 (schwarzen) NIL–Blaettern.

Eigenschaften von Rot-Schwarz-Bäumen

Satz

Die Höhe eines Rot-Schwarz-Baums mit n inneren Knoten ist höchstens $2\log(n+1)$.

Wir zeigen, dass Rot-Schwarz-Bäume balanciert sind:

Schwarz-Höhe

Die Schwarz-Höhe eines Knotens ist die Zahl der schwarzen Knoten auf einem beliebigen Pfad zu einem Blatt.

Die Schwarz-Höhe ist gleich der Höhe eines Knotens im entsprechenden 2-3-4 Baum. Ein Baum mit Schwarz-Höhe s enthält also mindestens $2^s - 1$ Knoten. Es folgt $s \leq \log(n+1)$. Für die Höhe h selbst gilt $h \leq 2s$ und damit der Satz.

Operationen auf Rot-Schwarz-Bäumen

Folgerung

Die Operationen Tree-Search, Tree-Minimum, Tree-Maximum, Tree-Successor und Tree-Predecessor benötigen für Rot-Schwarz-Bäume $O(\log n)$ Operationen.

Auch Tree-Insert und Tree-Delete laufen auf Rot-Schwarz-Bäumen in Zeit $O(\log n)$, erhalten aber nicht die Rot-Schwarz-Eigenschaft.

Diese wird dann durch Rotations- und Umfärbeoperationen wiederhergestellt.
Diese Operationen können aus der Deutung der Rot-Schwarz- als 2-3-4-Bäume hergeleitet werden.

Einfügen in einen Rot-Schwarz-Baum

Neuer Knoten x wird mit Tree-Insert eingefügt und **rot** gefärbt.

Ist $p[x]$ schwarz, so liegt $p[x]$ entweder in 2er oder richtig orientiertem 3er Makroknoten. Es ist nichts weiter zu tun.

Ist $p[x]$ rot, so liegt $p[x]$ entweder in 4er oder falsch orientiertem 3er Makroknoten.

Wir betrachten aus Symmetriegründen nur den Fall, dass $p[x] = \mathit{left}[p[p[x]]]$.

Wenn der Onkel $y = \mathit{right}[p[p[x]]]$ schwarz ist, so liegt ein falsch orientierter 3-er Makroknoten vor: Gemeinsam mit x zu einem 4-er Makroknoten zusammenfassen.

Einfügen in Rot-Schwarz-Baum

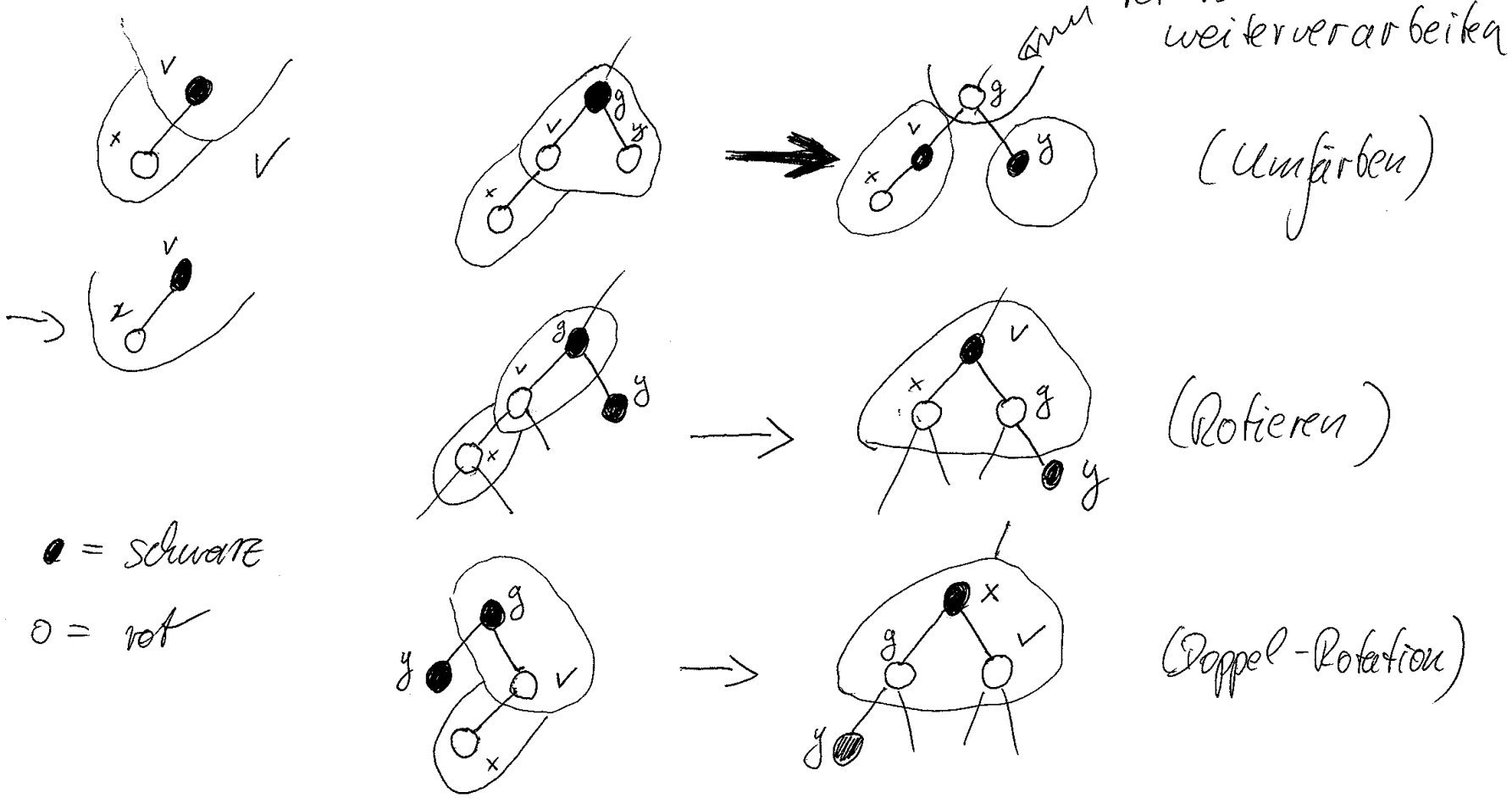
Wenn der Onkel $y = \mathit{right}[p[p[x]]]$ rot ist, so liegt ein 4-er Makroknoten vor. Umfärben von $x, p[x]p[p[x]], y$ spaltet diesen in 3er und 2er Knoten auf. Großvater $p[p[x]]$ jetzt rot. Evtl. Inkonsistenz weiter oben.

NB: Muss rotiert werden, so wird keine Inkonsistenz nach oben propagiert. Insgesamt also höchstens 2 Rotationen nötig.

Einfügen in RS-Baum

Einfügen in RS-Baum hat Laufzeit $O(\log(n))$ und erfordert $O(1)$ Rotationen.

Einfügen in RS-Baum



Löschen aus einem Rot-Schwarz-Baum

Zu löschender Knoten z wird mit Tree-Delete(T, z) gelöscht.

Erinnerung: Tree-Delete(T, z) entfernt einen Knoten y, der höchstens einen (inneren) Sohn hat.

- Hat z höchstens einen Sohn, so ist $y = z$.
- Andernfalls $y \leftarrow$ Tree-Successor(T, z).

Problem: Wenn der entfernte Knoten y schwarz war, ist Eigenschaft 3 verletzt.

Intuition: der (einzige) Sohn x von y erbt dessen schwarze Farbe, und ist jetzt "doppelt schwarz".

Das zusätzliche Schwarz wird in einer Schleife durch lokale Änderungen im Baum nach oben geschoben.

Ist x rot, so wird es schwarz gefärbt, und die Schleife bricht ab. Andernfalls nehmen wir o.B.d.A. an, dass $x = \mathit{left}[p[x]]$ und betrachten x' Bruder $w = \mathit{right}[p[x]] \neq \text{nil}$, und unterscheiden 2 Fälle:

- Fall 1: w ist rot. Dann muss $p[x]$ schwarz sein, also färbe $p[x]$ rot,
  w schwarz und rotiere links um $p[x]$
  $\rightsquigarrow$ reduziert auf Fall 2.
- Fall 2: w ist schwarz. Es gibt drei weitere Fälle:

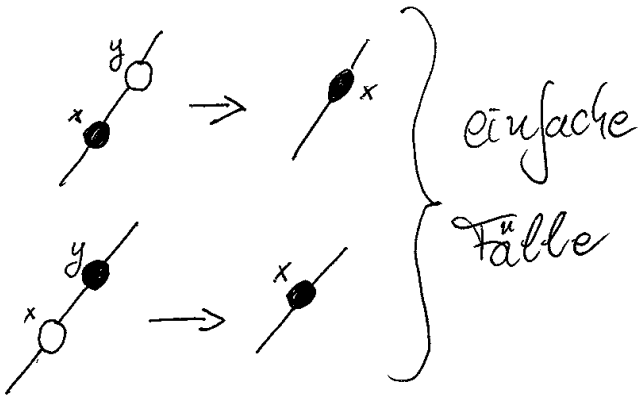
Drei Unterfälle

- Fall 2.1: Beide Kinder von w sind schwarz. Also können wir w rot färben, und das zusätzliche Schwarz von x zu $p[x]$ versetzen
  $\rightsquigarrow$ nächste Iteration.
- Fall 2.2: $left[w]$ ist rot, $right[w]$ ist schwarz. Dann färbe w rot, $left[w]$ schwarz und rotiere rechts um w
  $\rightsquigarrow$ reduziert auf Fall 2.3.
- Fall 2.3: $right[w]$ ist rot. Dann vertausche die Farben von w und $p[x]$, färbe $right[w]$ schwarz, und rotiere links um $p[x]$
  $\rightsquigarrow$ Zusätzliches Schwarz ist verbraucht.

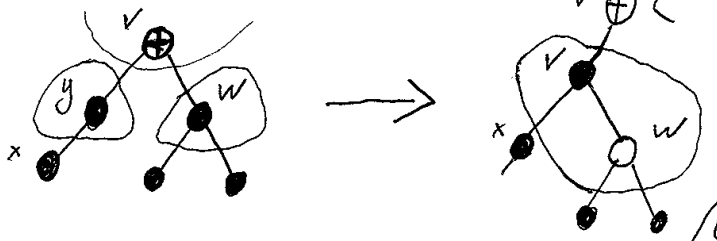
Löschen aus RS-Baum

● schwarz
○ rot
⊕ egal

Zu löschender Knoten y hat höchstens ein Kind x, das kein Blatt ist.

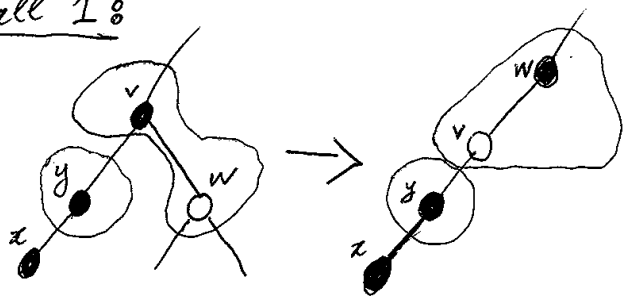


Fall 2.1

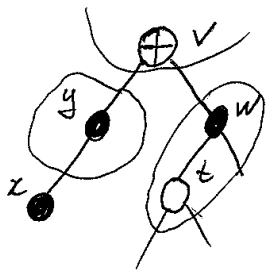


In Wirklichkeit wird v nicht verdoppelt

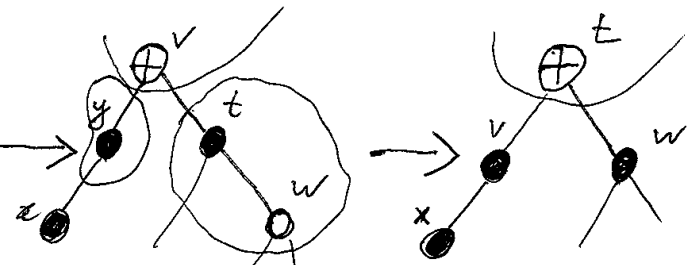
Fall 1:



Fall 2.2



Fall 2.3



Auch beim Löschen wird am Ende die Wurzel $root[T]$ schwarz gefärbt.

Bemerke:

- In den Fällen 2.2 und 2.3 bricht die Schleife sofort ab.
- Schleife wird nur im Fall 2.1 wiederholt, dann ist die Tiefe von x kleiner geworden.
- Im Fall 1 wird die Tiefe von x größer, aber die Schleife bricht danach im Fall 2 sofort ab.

$\rightsquigarrow$ Laufzeit ist $O(\log n)$, es werden höchstens 3 Rotationen durchgeführt.

ZUSAMMENGEFASST: die Operationen **Search, Minimum, Maximum, Successor, Predecessor, Insert** und **Delete** können für Rot-Schwarz-Bäume mit Laufzeit $O(\log n)$ realisiert werden.

Dynamische Mengen als Hashtabelle

- Direkte Adressierung
- Hashing und Hashfunktionen
- Kollisionsauflösung durch Verkettung
- Offene Adressierung
- Analyse der erwarteten Laufzeiten

In diesem Abschnitt werden Arrayfächer beginnend mit 0 nummeriert.

Direkte Adressierung

- Sind die Schlüssel ganze Zahlen im Bereich $0 \ldots N-1$, so kann eine dynamische Menge durch ein **Array A der Größe N** implementiert werden.
- Der Eintrag $A[k]$ ist x falls ein Element x mit Schlüssel k eingetragen wurde.
- Der Eintrag $A[k]$ ist **Nil**, falls die dynamische Menge kein Element mit Schlüssel k enthält.
- Die Operationen Search, Insert, Delete werden unterstützt und haben Laufzeit $\Theta(1)$.
- Nachteile: *Enormer Speicherplatz* bei großem N. Nicht möglich, falls keine obere Schranke an Schlüssel vorliegt.

Hash-Tabelle

- Sei U die Menge der Schlüssel, z.B.: $U = \mathbb{N}$.
- Gegeben eine Funktion $h : U \to \{0, 1, 2, 3, \ldots, m-1\}$, die „Hashfunktion“.
- Die dynamische Menge wird implementiert durch ein Array der Größe m.
- Das Element mit Schlüssel k wird an der Stelle $A[h(k)]$ abgespeichert.

Search(A, k)	Insert(A, x)	Delete(A, k)
1 **return** $A[h(k)]$	1 $A[h(key[x])] \leftarrow x$	1 $A[h(k)] \leftarrow$ Nil

- Zum Beispiel: $U = \mathbb{N}$, $h(k) = k \bmod m$.
- Problem: $h(k_1) = h(k_2)$ obwohl $k_1 \neq k_2$ (**Kollision**). Kommt es zu Kollisionen, so ist dieses Verfahren **inkorrekt** (also **gar kein Verfahren**!).

Häufigkeit von Kollisionen

- Alle m Hashwerte seien gleichwahrscheinlich.
- Die Wahrscheinlichkeit, dass k zufällig gewählte Schlüssel *paarweise verschiedene Hashwerte* haben ist dann:

$$\begin{aligned} & 1 \cdot \tfrac{m-1}{m} \cdot \tfrac{m-2}{m} \cdots \tfrac{m-k+1}{m} = \textstyle\prod_{i=0}^{k-1}(1-\tfrac{i}{m}) \\ \leq & \quad \textstyle\prod_{i=0}^{k-1} e^{-i/m} \\ = & \quad e^{-\sum_{i=0}^{k-1} i/m} = e^{-k(k-1)/2m} \end{aligned}$$

- Diese Wahrscheinlichkeit wird kleiner als 50% wenn $k \geq 1 + \frac{1}{2}\sqrt{1 + 8m \ln 2}$.
- Beispiel $m = 365$, $h(k) =$„Geburtstag von k“. Bei mehr als 23 Menschen ist es wahrscheinlicher, dass zwei *am selben Tag Geburtstag* haben, als umgekehrt.
- Kollisionen sind *häufiger als man denkt*.

Kollisionsauflösung durch Verkettung

Um Kollisionen zu begegnen, hält man in jeder Arrayposition eine **verkettete Liste von Objekten**.

- **Suchen** geschieht durch Suchen in der jeweiligen Liste,
- **Einfügen** geschieht durch Anhängen an die jeweilige Liste,
- **Löschen** geschieht durch Entfernen aus der jeweiligen Liste.

Search(A, k)
1 Suche in der Liste $A[h(k)]$ nach Element mit Schlüssel k

Insert(A, x)
1 Hänge x am Anfang der Liste $A[h(k)]$ ein.

Delete(A, k)
1 Entferne das Objekt mit Schlüssel k aus der Liste $A[h(k)]$.

Leider ist die Laufzeit jetzt *nicht mehr* $\Theta(1)$.

Lastfaktor

Die Hashtabelle habe m Plätze und enthalte n Einträge.
Der Quotient $\alpha := n/m$ heißt **Lastfaktor**.
Beachte: $\alpha > 1$ ist möglich.
Der Lastfaktor heißt auch **Belegungsfaktor**
Eine Hashtabelle heißt auch **Streuspeichertabelle**.

Analyse von Hashing mit Verkettung

Die Hashwerte seien wiederum uniform verteilt.
Dann werden die Listen im Mittel Länge α besitzen.
Die Operationen Search, Insert, Delete haben also jeweils *erwartete* (=mittlere) Laufzeit

$$T \leq c(1 + \alpha)$$

für geeignete Konstante $c > 0$.
Der Summand „1“ bezeichnet den Aufwand für das Berechnen der Hashfunktion und die Indizierung.
Der Summand α bezeichnet die lineare Laufzeit des Durchsuchens einer verketteten Liste.

Hashfunktionen

Seien die einzutragenden Objekte x irgendwie zufällig verteilt.
Die Hashfunktion sollte so beschaffen sein, dass die Zufallsvariable $h(key[x])$ **uniform verteilt** ist (da ja sonst manche Fächer leer bleiben, während andere überfüllt sind.)
Sind z.B. die Schlüssel in $\{0, \ldots, N-1\}$ uniform verteilt, so ist $h(k) = k \bmod m$ eine gute Hashfunktion.
Sind z.B. die Schlüssel in $[0, 1[$ uniform verteilt, so ist $h(k) = \lfloor mk \rfloor$ eine gute Hashfunktion.
m wie immer die Größe der Hashtabelle.
Die Schlüssel sind meist **nicht uniform** verteilt:
Bezeichner in einer Programmiersprache: `count`, `i`, `max_zahl` häufiger als `zu6fgp98qq`. Wenn `kli` dann oft auch `kli1`, `kli2`, etc.

Nichtnumerische Schlüssel

. . . müssen vor Anwendung einer „Lehrbuch-Hashfunktion“ zunächst in Zahlen konvertiert werden.

Zeichenketten etwa unter Verwendung der Basis 256:

`'p'` $= 112$, `'q'` $= 116$, also `"pq"` $= 112 \cdot 256 + 116 = 28788$.

Verwendet man eine Hashfunktion der Form $h(x) = x \pmod m$, so kann man schon während der Konversion immer wieder mod m reduzieren.

Divisionsmethode

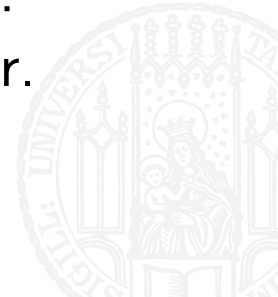
Wie immer: Schlüssel: $0 \ldots N-1$, Hashwerte: $0 \ldots m-1$.
Hashfunktion: $h(k) = k \bmod m$.

- m sollte keine Zweierpotenz sein, da sonst $h(k)$ nicht von allen Bits (von k) abhängt.
- Ist k eine Kodierung eines Strings im 256er System, so bildet h bei $m = 2^p - 1$ zwei Strings, die sich nur durch eine Transposition unterscheiden, auf denselben Wert ab.
- Eine gute Wahl für m ist eine Primzahl, die nicht nahe bei einer Zweierpotenz liegt. Z.B. $n = 2000$, vorauss. Lastfaktor $\alpha = 3$: Tabellengröße $m = 701$ bietet sich an.
- Es empfiehlt sich ein Test mit „realen Daten“.

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m(k \cdot A \bmod 1) \rfloor$ für $A \in]0,1[$.
Hier $x \bmod 1 =$„gebrochener Teil von x“, z.B.:
$\pi \bmod 1 = 0,14159\ldots$.
Rationale Zahlen A mit kleinem Nenner führen zu Ungleichverteilungen, daher empfiehlt sich die Wahl $A = (\sqrt{5}-1)/2$ („Goldener Schnitt“)
Vorteile der Multiplikationsmethode:

- Arithmetische Progressionen von Schlüsseln $k = k_0, k_0 + d, k_0 + 2d, k_0 + 3d, \ldots$ werden ebenmäßig verstreut.
- Leicht zu implementieren, wenn $m = 2^p$ (hier unproblematisch) und $N \leq 2^w$, wobei w die Wortlänge ist: Multipliziere k mit $\lfloor A \cdot 2^w \rfloor$. Dies ergibt zwei w-bit Wörter. Vom Niederwertigen der beiden Wörter bilden die p höchstwertigen Bits den Hashwert $h(k)$.



Weiterführendes

- **Universelles Hashing**: Zufällige Wahl der Hashfunktion bei Initialisierung der Tabelle, dadurch Vermeidung systematischer Kollisionen, z.B. Provokation schlechter Laufzeit durch bösartig konstruierte Benchmarks.
- Gute Hashfunktionen können zur Authentizierung verwendet werden, z.B., MD5 *message digest*.

Offene Adressierung

Man kann auf verkettete Listen verzichten, indem man bei Auftreten einer Kollision eine andere Arrayposition benutzt.
Dazu braucht man eine zweistellige Hashfunktion
$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$.

```
Insert(T, x)
1  i ← 0
2  while  i ≤ m and T[h(key[x], i)] ≠ Nil do
3         i ← i + 1
4  if  i ≤ m
5         then  T[h(key[x], i)] ← x
6         else  error  "hash table overflow"
```

Für jeden Schlüssel k sollte die **Sondierungsfolge** (*probe sequence*)

$$h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)$$

eine Permutation von $0, 1, 2, \ldots, m-1$ sein, damit jede Position irgendwann sondiert wird.

```
Search(T, k)
1  i ← 0
2  repeat
3        j ← h(k, i); i ← i + 1
4  until  i = m or T[j] = Nil or key[T[j]] = k
5  if  i < m and key[T[j]] = k
6        then return T[j]
7        else return Nil
```

NB: Tabelleneinträge sind Objekte **zuzüglich** des speziellen Wertes Nil.

Z.B. Zeiger auf Objekte oder **Nullzeiger**.

EINSCHRÄNKUNG: Bei offener Adressierung ist Löschen etwas umständlich (etwa durch explizites Markieren von Einträgen als "gelöscht").

Siehe auch

`http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html`

Hashfunktionen für offene Adressierung

- **Lineares Sondieren** (*linear probing*):

$$h(k,i) = (h'(k) + i) \bmod m$$

  Problem: Lange zusammenhängende Blöcke besetzter Plätze entstehen (***primary clustering***), dadurch oft lange Sondierdauer.
- **Quadratisches Sondieren** (*quadratic probing*):

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

  für passende c_1, c_2 sodass $h(k,\cdot)$ Permutation ist.

Quadratisches Sondieren ist besser als lineares Sondieren, hat aber immer noch **folgenden Nachteil**: Wenn $h(k_1,0) = h(k_2,0)$, dann $h(k_1,i) = h(k_2,i)$, d.h., kollidierende Schlüssel haben dieselbe Sondierungsfolge. Insgesamt gibt es nur m verschiedene Sondierungsfolgen (von $m!$ Möglichen!); das führt auch zu Clusterbildung (*secondary clustering*).

- **Double hashing**

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

Jede Sondierungsfolge ist eine **arithmetische Progression**, Startwert und Schrittweite sind durch Hashfunktionen bestimmt.
Damit alle Positionen sondiert werden, muss natürlich $\mathrm{ggT}(h_2(k), m) = 1$ sein. Z.B. m Zweierpotenz und $h_2(k)$ immer ungerade.
Es gibt dann $\Theta(m^2)$ Sondierungsfolgen.

Analyse der Offenen Adressierung

Vereinfachende Annahme: Die Schlüssel seien so verteilt, dass jede der $m!$ Sondierungsfolgen *gleichwahrscheinlich* ist.
Diese Annahme wird durch **double hashing** approximiert, aber nicht erreicht.
Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartende Länge einer erfolglosen Suche beschränkt durch $1/(1-\alpha)$.

Beispiel: Lastfaktor $\alpha = 0,9$ (Tabelle zu neunzig Prozent gefüllt): Eine erfolglose Suche erfordert im Mittel weniger als 10 Versuche (unabhängig von m, n).

Bemerkung: Dies ist auch die erwartete Laufzeit (Zahl der Sondierungen) für eine Insertion.

Beweis des Satzes

Sei X eine Zufallsvariable mit Werten aus $\mathbb{N}$.
Dann ist

$$E[X] := \sum_{i=0}^{\infty} i\Pr\{X = i\} = \sum_{i=1}^{\infty} \Pr\{X \geq i\} = \sum_{i=0}^{\infty} \Pr\{X > i\}$$

Dies deshalb, weil $\Pr\{X \geq i\} = \sum_{j=i}^{\infty} \Pr\{X = j\}$.
Daher ergibt sich für die erwartete Suchdauer D:

$$D = \sum_{i=0}^{\infty} \Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\}$$

$$\Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \alpha^i$$

Also, $D \leq \sum_{i=0}^{\infty} \alpha^i = 1/(1-\alpha)$.

Analyse der Offenen Adressierung

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartetende Dauer einer erfolgreichen Suche beschränkt durch $1/\alpha \ln(\frac{1}{1-\alpha})$).
Beispiel: Lastfaktor $\alpha = 0,9$: Eine erfolgreiche Suche erfordert im Mittel weniger als 2,6 Versuche (unabhängig von m, n).
Lastfaktor $\alpha = 0,5$: mittlere Suchdauer ≤ 1.39.
Achtung: All das gilt natürlich nur unter der *idealisierenden* Annahme von **uniform hashing**.

Bemerkung: in der ursprünglichen Version der Folien (und einer älteren Version von Cormen) stand die ebenso gültige, aber schlechtere, Abschätzung $(1 - \ln(1 - \alpha))/\alpha$.

Beweis

Die beim Aufsuchen des Schlüssels durchlaufene Sondierungsfolge ist dieselbe wie die beim Einfügen durchlaufene.
Die Länge dieser Folge für den als $i+1$-ter eingefügten Schlüssel ist im Mittel beschränkt durch $1/(1-i/m) = m/(m-i)$. (Wg. vorherigen Satzes!)
Gemittelt über alle Schlüssel, die eingefügt wurden, erhält man also

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{i=m-n+1}^{m}\frac{1}{i} \leq$$
$$\frac{1}{\alpha}\int_{i=m-n}^{m}\frac{1}{i} = \frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$$

als obere Schranke für den Erwartungswert der Suchdauer.
Die Ungleichung ist eine Anwendung der Integralabschätzung, die schon beim randomisierten Quicksort vorkam.

Mittlere Suchdauer bei Linearem Sondieren

Ohne Beweis geben wir noch die folgenden Schranken für das lineare Sondieren (Knuth 1962).

Für die mittlere Suchdauer bei erfolgloser Suche und linearem Sondieren gilt die obere Schranke:

$$\frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$$

Für die mittlere Suchdauer bei erfolgreicher Suche und linearem Sondieren gilt die obere Schranke:

$$\frac{1}{2}(1 + (\frac{1}{1-\alpha}))$$

Es empfiehlt sich $\alpha < 0.6$.

Weiterführendes: Bloom-Filter

Der Bloom Filter bietet die Möglichkeit eine Datenmenge approximativ zu verwalten.

Elemente werden eingefügt, aber nicht gelöscht. Suchanfragen (hier nur Test auf Enthaltensein!) werden approximativ beantwortet: wurde das gesuchte Element vorher eingefügt, so wird mit Sicherheit "ja" geantwortet. Falls nein, so wird mit Wahrscheinlichkeit $\geq 1 - \epsilon$ "nein" geantwortet. Es gibt aber einen Anteil (höchstens) ϵ von "false positives".

Mögliche Anwendung: Register von als "gefährlich" erkannten Webseiten.

Eine Möglichkeit: verwende Hashtabelle, ohne die Einträge explizit zu speichern, setze nur ein Bit, falls "vorhanden":
Boole'sches Array T der Größe m. Um x einzutragen, setze $T[h(x)] \leftarrow 1$. Beim Suchen von x liefert man "ja" zurück, gdw. $T[h(x)] = 1$.

Fehleranalyse

Fehlerwahrscheinlichkeit bei n eingetragenen Elementen und m Slots unter uniform Hashing und Unabhängigkeitsannahme:

$$\epsilon = 1 - (1 - \frac{1}{m})^n \approx 1 - e^{-\alpha}$$

mit $\alpha = n/m$. Die Näherung ist i.O. für $\frac{1}{m}$ sehr klein.

Falls $\alpha = 0.1$: Fehlerw.: $\approx 10\%$.

Verbesserung mit mehreren Hashfunktionen

Verwende k Hashfunktionen $h_1, \ldots, h_k$. Beim Eintrag von x setze k Bits $T[h_1(x)], \ldots, T[h_k(x)]$.

Beim Test, ob x eingetragen ist, liefere "ja" nur, wenn alle Bits $T[h_1(x)], \ldots, T[h_k(x)]$ gesetzt sind. Fehlt auch nur eines, so antworte "nein".

Hilft das wirklich?

Analyse der Fehlerwahrscheinlichkeit

Fehlerwahrscheinlichkeit bei n eingetragenen Elementen und m Slots und k Hashfunktionen unter den üblichen Annahmen:

$$\epsilon = (1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{-\alpha k})^k$$

mit $\alpha = n/m$. Die Näherung ist i.O. für $\frac{1}{m}$ sehr klein.

Dieser Ausdruck nimmt sein Minimum an für $k = \ln(2)/\alpha$ (ist Nullstelle der Ableitung, Bestätigung durch Einsetzen) und hat dann den Wert 2^{-k}.
Für $\alpha = 0.1$ hat man $k = 8$ und die Fehlerw. ist $\approx 2^{-8} < 1\%$.

Further Reading:
`https://en.wikipedia.org/wiki/Bloom_filter`

Weiterführendes: Cuckoo-Hashing

2001 von Pagh und Rodler vorgeschlagen. Seitdem intensiv theoretisch untersucht, möglicherweise bald auch praktisch verwendet.

Ähnlich wie offene Adressierung, aber verwendet nur zwei Hashfunktionen: $h_1, h_2 : U \to \{0, \ldots, m-1\}$.
Ein Element x wird entweder in $T[h_1(x)]$ oder $T[h_2(x)]$ gespeichert. Suchen (Search) geht daher in konstanter Zeit.

Will man ein neues Element x eintragen und sind beide Plätze $T[h_1(x)]$ und $T[h_2(x)]$ schon belegt, dann wird eines der beiden hinausgeworfen und muss dann an seinen Alternativplatz. Ggf. muss dazu wieder ein "Ei" hinausgeworfen werden, etc.

Wurde nach $\log(m)$ Schritten kein Platz gefunden, so wird die gesamte Tabelle vergrößert und neu aufgebaut.

Man kann zeigen (kompliziert!), dass auch das Einfügen in erwartet konstanter Zeit abläuft.

Zusammenfassung

- Hashing = Speichern von Objekten an Arraypositionen, die aus ihrem Schlüssel *berechnet* werden.
- Die Zahl der Arraypositionen ist i.a. *wesentlich kleiner* als die der Schlüssel.
- Kollisionsauflösung durch Verkettung: Jede Arrayposition enthält eine verkettete Liste.
- Offene Adressierung: Bei Kollision wird eine andere Arrayposition sondiert.
- Hashfunktionen für einfaches Hashing: Multiplikationsmethode, Divisionsmethode.
- Hashfunktionen für **offene Adressierung**: Lineares, quadratisches Sondieren. Double Hashing.
- Analyse unter Uniformitätsannahme, Komplexität jeweils als Funktion des **Lastfaktors** α (Auslastungsgrad).