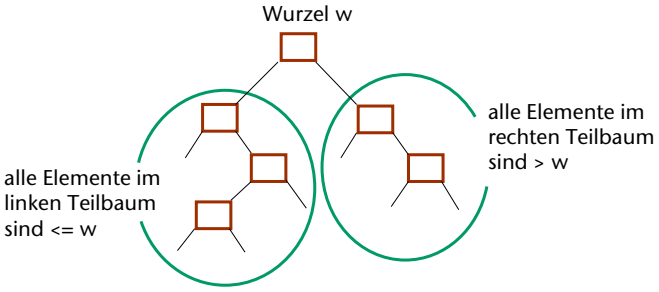
Informatik II
Suchbäume für effizientes *Information Retrieval*

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Binäre Suchbäume (binary search tree, BST)

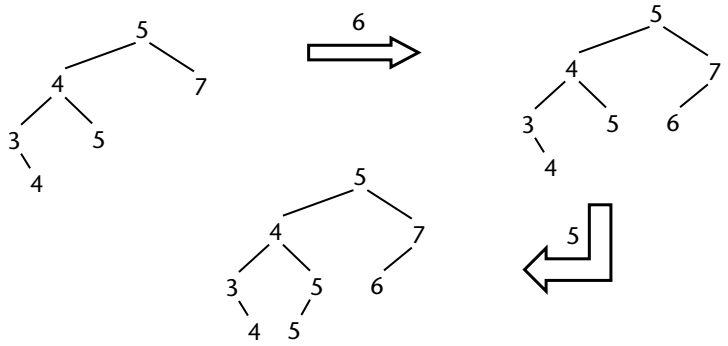
- Speichere wieder Daten als "Schlüssel + Nutzdaten"
- Sei eine Ordnungsrelation für die Schlüssel im Baum definiert
- Ziel: Binärbäume zur Speicherung von Mengen von Schlüsseln, so daß folgende Operationen effizient sind:
  - Suchen (find)
  - Einfügen (insert)
  - Entfernen (remove, delete)

- Definition:
  Ein binärer Baum mit Suchbaumeigenschaft ist von folgender Form

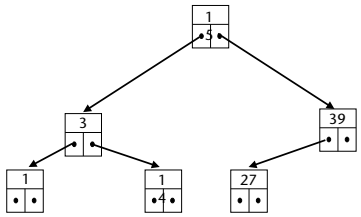


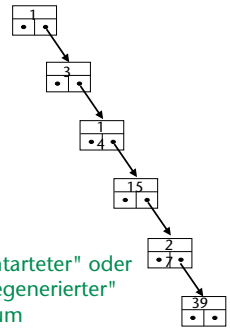
Einfügen

- Hinzufügen von x zum Baum B
  - wenn B leer ist, erzeuge Wurzel mit x
  - wenn x <= Wurzel, füge x rekursiv zum linken Teilbaum hinzu
  - wenn x > Wurzel, füge x rekursiv dem rechten Teilbaum hinzu
- Beispiel:



- Achtung:
  - Baum-Struktur hängt von Einfügereihenfolge im anfangs leeren Baum ab!
  - Dito für Höhe: Höhe kann, je nach Reihenfolge, zwischen n und $\lceil \log_2(n+1) \rceil$ liegen
- Beispiel: resultierende Suchbäume für die Reihenfolgen 15, 39, 3, 27, 1, 14 und 1, 3, 14, 15, 27, 39:

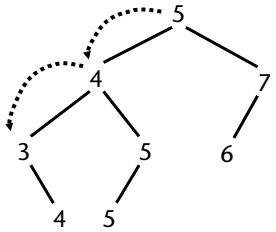




"entarteter" oder "degenerierter" Baum

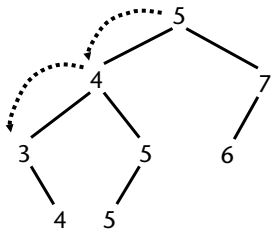
Suchen

- Aufgabe: Key x im BST B suchen:
  - wenn B leer ist, dann ist x nicht in B
  - wenn x = Wurzelelement gilt, haben wir x gefunden
  - wenn x < Wurzelelement, suche im linken Teilbaum
  - wenn x > Wurzelelement, suche im rechten Teilbaum
- Beispiel: suche 3 im Baum



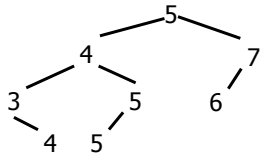
- Bemerkung: gibt es mehrere Knoten mit gleichem Wert, wird offenbar derjenige mit der geringsten Tiefe gefunden

- Aufgabe: suche kleinstes Element
  - Folge dem linken Teilbaum, bis Knoten gefunden wurde, dessen linker Teilbaum leer ist



- Analog: größtes Element finden

- Aufgabe: Nachfolger in Sortierreihenfolge suchen
  - Wenn der Knoten einen rechten Teilbaum hat, ist der Nachfolger das kleinste Element des rechten Teilbaumes
  - Ansonsten: Aufsteigen im Baum, bis ein Element gefunden wurde, das größer oder gleich dem aktuellen Knoten ist
    - Falls man die Wurzel erreicht hat, gibt es kein solches Element
  - Dazu sollte der Knoten eine Referenz auf seinen Vater beinhalten
- Beispiel:

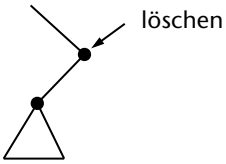


Löschen

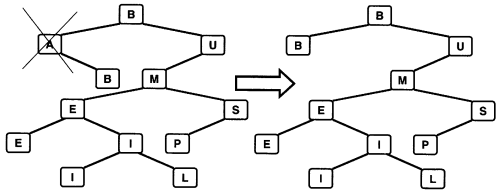
- Ist die aufwendigste Operation
  - Auch hier muß sichergestellt werden, daß der verbleibende Baum ein gültiger binärer Suchbaum ist

1. Fall: Knoten hat keinen Teilbaum (= Blatt)
   - Knoten einfach löschen

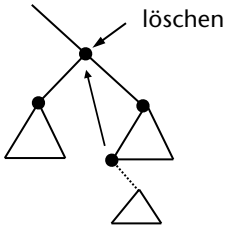
2. Fall: nur ein Teilbaum
   - Teilbaum eine Etage höher schieben



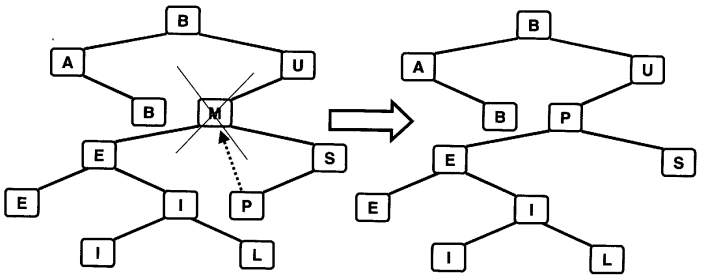
- Beispiel:



3. Fall: Knoten hat zwei Teilbäume
   - Suche kleinstes Element des rechten Teilbaumes
     - Dieses kleinste Element hat höchstens einen rechten Teilbaum
   - Kopiere Inhalt (Schlüssel + Daten) dieses kleinsten Knotens in den Inhalt des zu löschenden Knotens
   - Lösche den Knoten, der vorher dieses kleinste Element beinhaltete → Fall 2

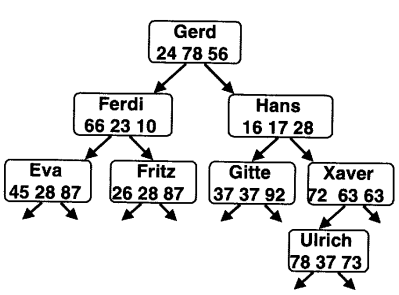


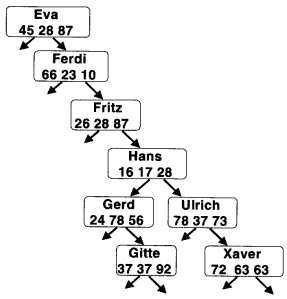
- Beispiel:



Balancierte Bäume

- Aufwand, ein Element zu finden, entspricht der Tiefe des gefundenen Knotens
  - Im worst case = Tiefe des Baumes
  - Diese liegt zwischen $\lfloor \log N \rfloor + 1$ und N
- Schlecht balancierte Bäume erhält man, wenn die Elemente in sortierter Reihenfolge angeliefert werden





- Definition für "balanciert":
  - Es gibt verschiedene Definitionen!
  - Allgemein: kein Blatt ist “wesentlich weiter“ von der Wurzel entfernt als irgendein anderes
  - Hier: Für alle Knoten unterscheidet sich Anzahl der Knoten in linkem und rechtem Teilbaum höchstens um 1
  - Folge: ein balancierter BST hat die Tiefe $\lfloor \log N \rfloor + 1$
- Der Aufwand, einen optimal balancierten Baum nach Einfüge- und Löschoperationen zu erzwingen, ist sehr groß

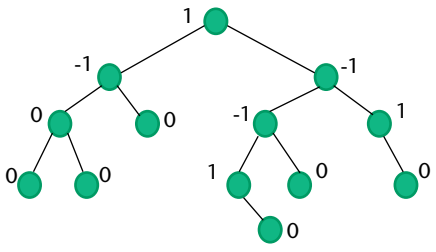
AVL-Bäume

- AVL-Baum
  - Von Adelson-Velskij und Landis eingeführt [Sowjetunion, 1962]
  - Schwächere Form eines balancierten Baumes
- Definition Balance-Faktor eines Knotens x:

  bal(x) := (Höhe des rechten Unterbaumes von x) - (Höhe des linken Unterbaumes von x)
- Definition AVL-Baum:

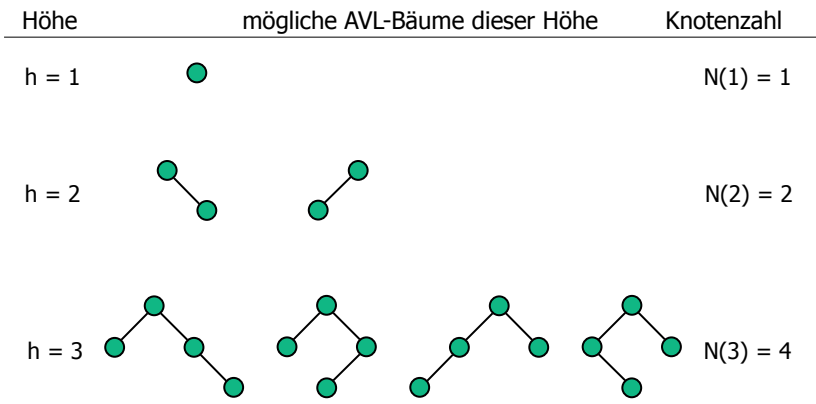
  Ein binärer Baum, wobei für jeden Knoten x gilt:

  $bal(x) \in \{-1,0,1\}$



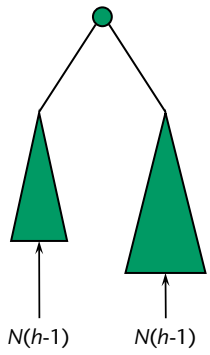
Minimale Knotenanzahl von AVL-Bäumen

- Bezeichne $N(h)$ die minimale Anzahl von Knoten eines AVL-Baumes der Höhe h



- Allgemeiner worst case Fall bei Höhe h:

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



Satz:

$$N(h) = F_{h+2} - 1$$

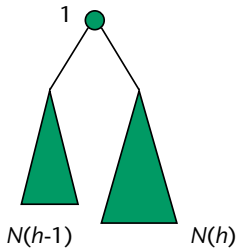
Beweis:

1. Induktionsanfang: $h = 1$

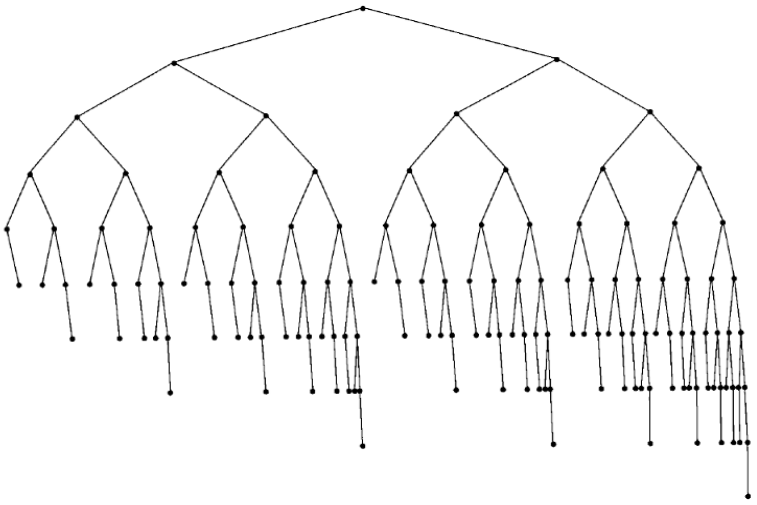
$$F_{1+2} - 1 = F_3 - 1 = 2 - 1 = 1$$

2. Induktionsschritt: $h \to h + 1$

$$\begin{aligned} N(h+1) &= 1 + N(h) + N(h-1) \\ &= 1 + F_{h+2} - 1 + F_{h+1} - 1 \\ &= F_{h+3} - 1 \\ &= F_{[h+1]+2} - 1 \end{aligned}$$



Beispiel: ein minimaler AVL-Baum der Höhe 10



Die maximale Höhe von AVL-Bäumen

- Erinnerung bzgl. Fibonacci-Zahlen :

$$F_n \approx \frac{1}{\sqrt{5}}\Phi^n \quad , \quad \Phi = \frac{1+\sqrt{5}}{2} \approx 1.61803398875\ldots$$

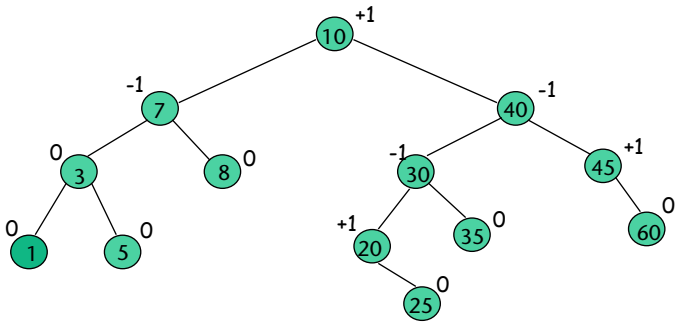
- Aus $N(h) = F_{h+2} - 1$ folgt nach Umformung und Abschätzung von F_n die …
- Wichtige Eigenschaft von AVL-Bäumen:
  Ein AVL-Baum mit N Knoten hat höchstens die Höhe

$$h \leq 1.44\ldots * \log(N) + \text{const}$$

- Erinnerung: Die Höhe jedes binären Baumes mit N Knoten beträgt mindestens $\log(N+1)$

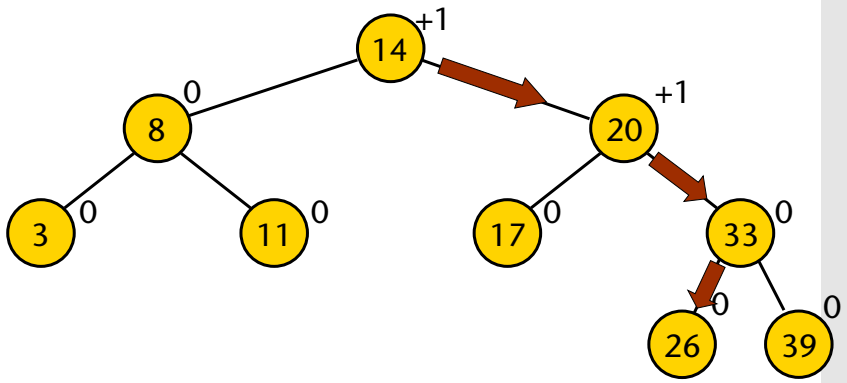
Der AVL Search Tree

- Problem: wir wollen einen BST, der auch über viele Insert- und Delete-Operationen halbwegs gut balanciert bleibt
- Idee: verwende BST, der zusätzlich die AVL-Eigenschaften hat
- Aufgabe: wie erhält man AVL-Eigenschaften bei Einfügen/ Löschen?



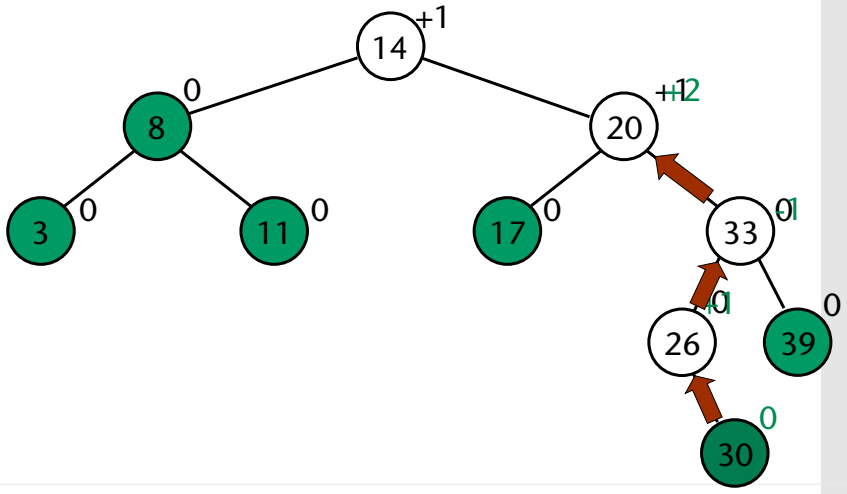
Einfügen von Knoten

Beispiel: Einfügen von $k = 30$

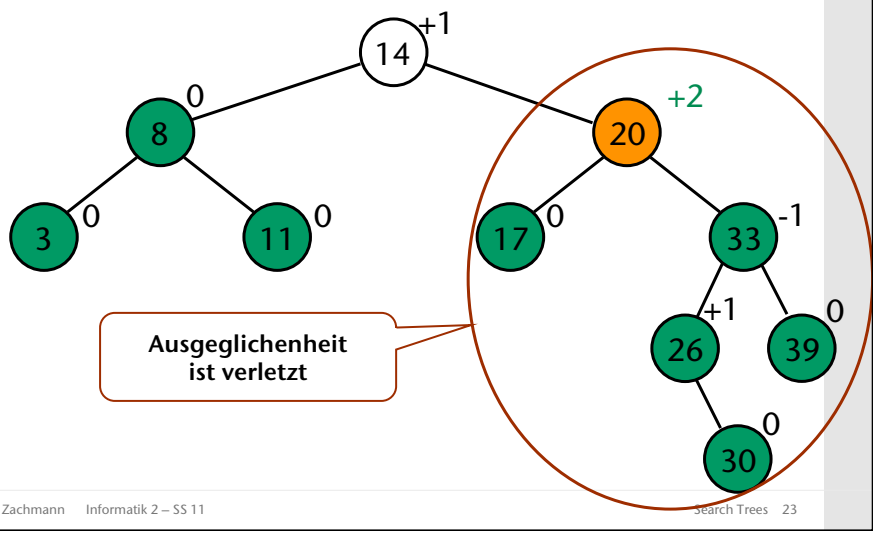


Einfügen von Knoten

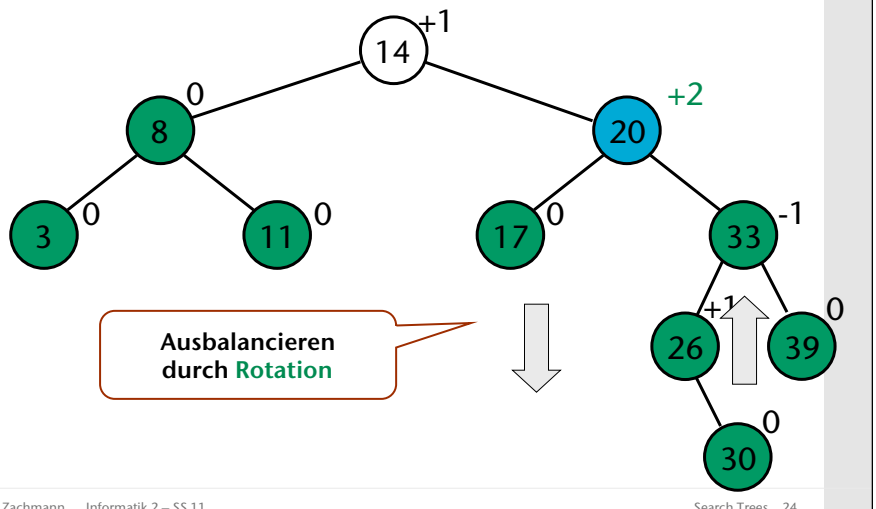
Beispiel: Einfügen von $k = 30$



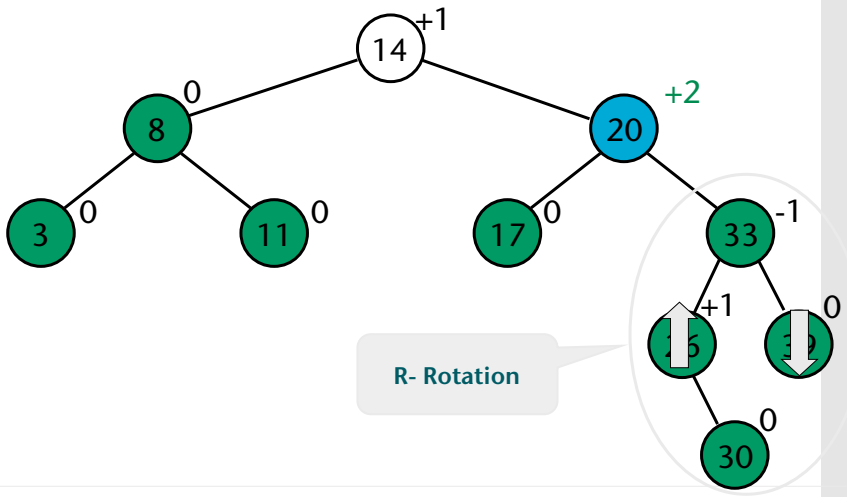
Beispiel: Einfügen von $k = 30$



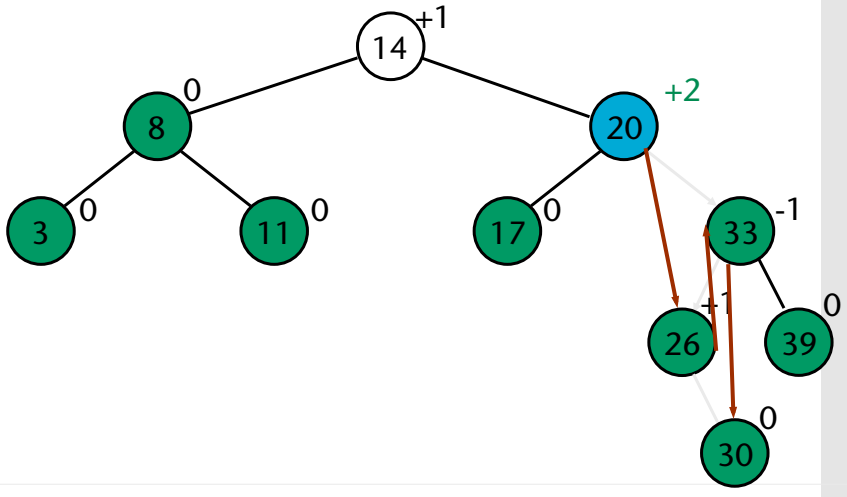
Beispiel: Einfügen von $k = 30$



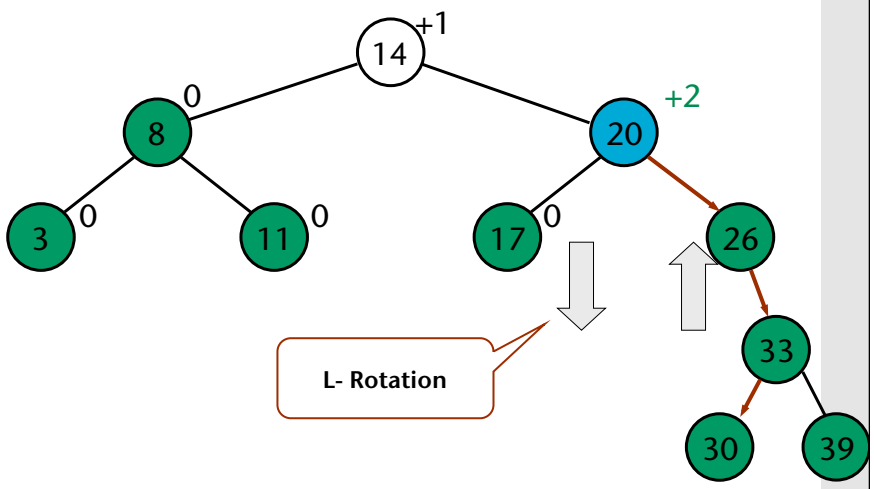
Beispiel: Einfügen von $k = 30$



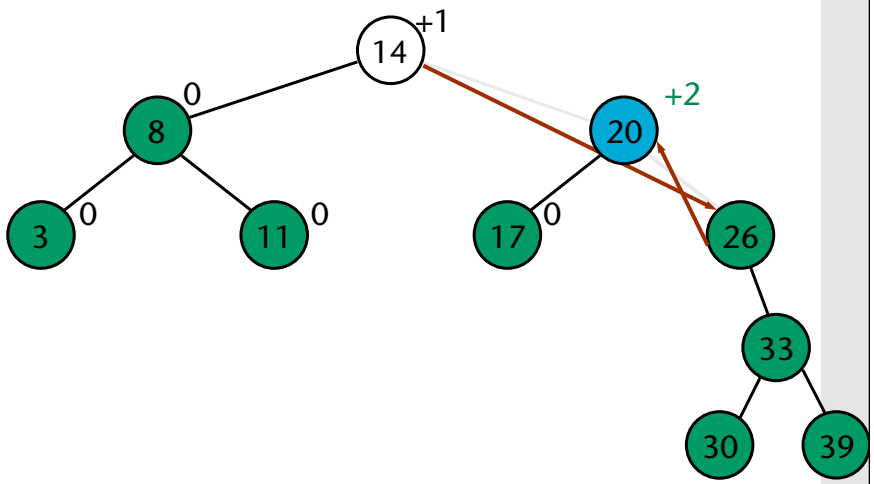
Beispiel: Einfügen von $k = 30$



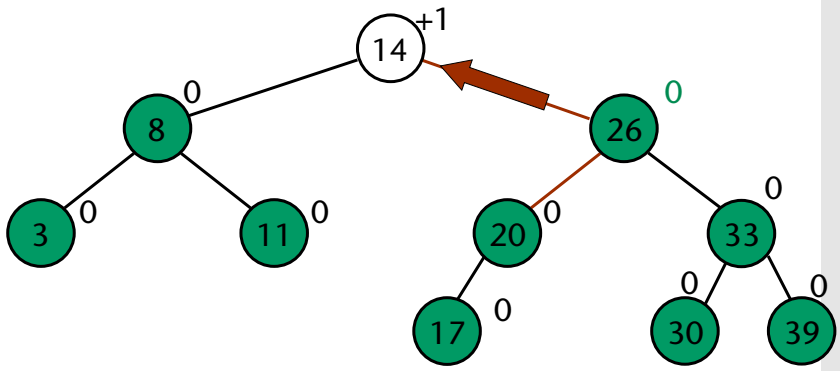
Beispiel: Einfügen von $k = 30$



Beispiel: Einfügen von $k = 30$

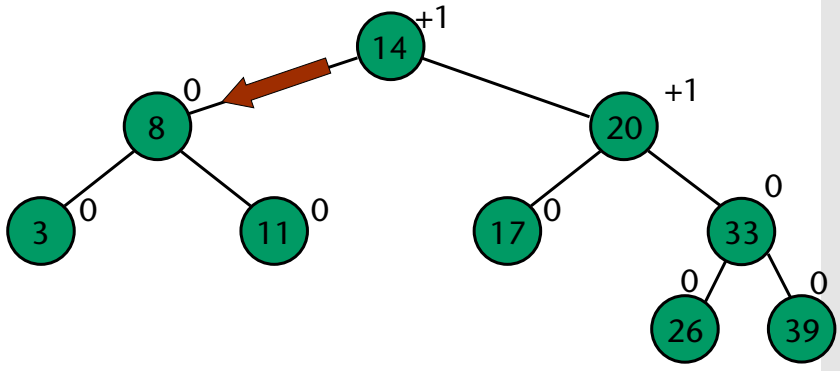


Beispiel: Einfügen von $k = 30$

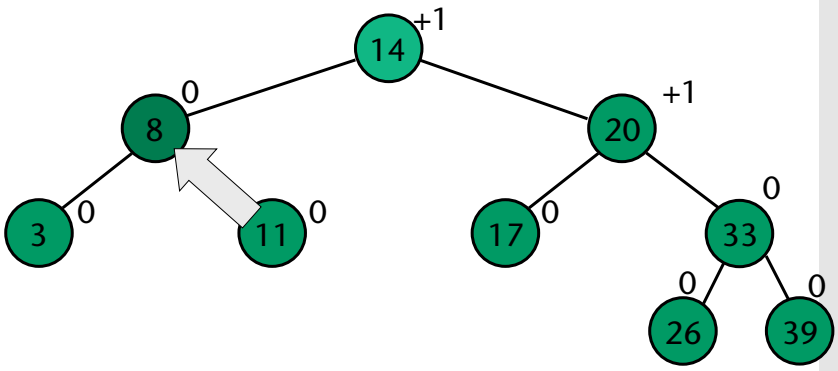


Löschen von Knoten

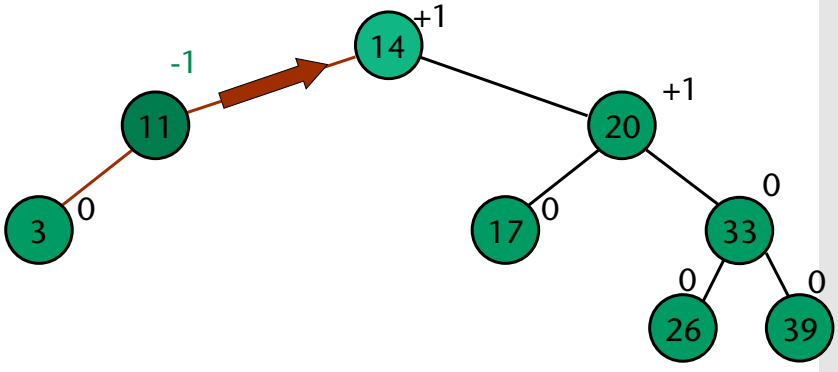
Beispiel: Löschen von $k = 8$



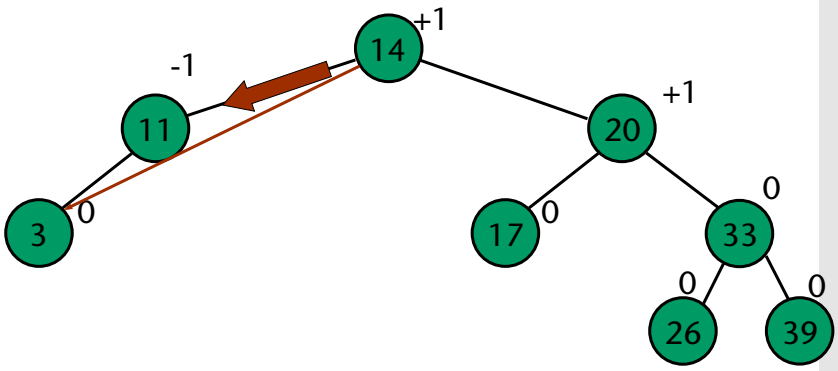
Beispiel: Löschen von $k = 8$



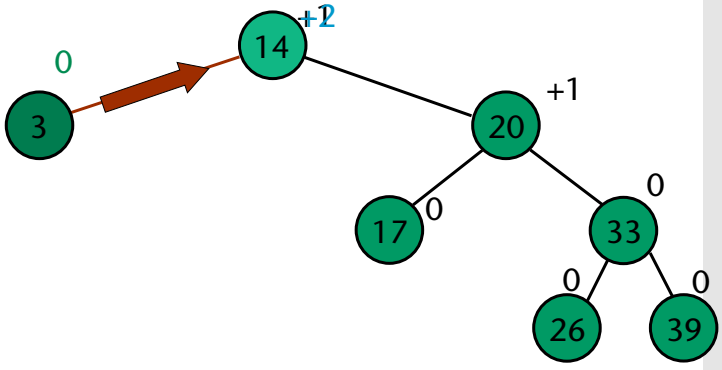
Beispiel: Löschen von $k = 8$



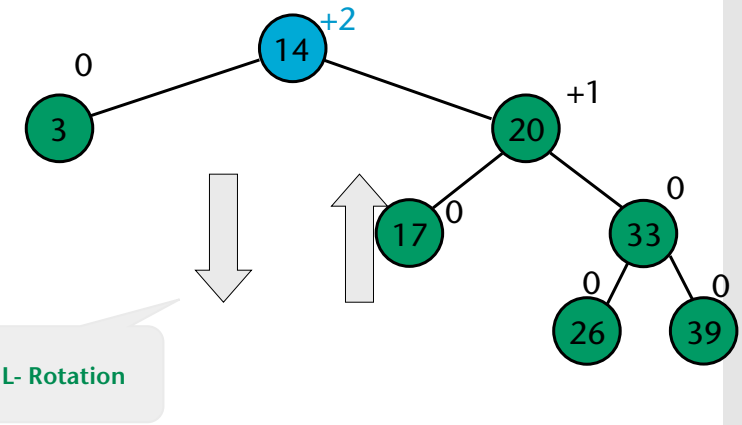
Beispiel: Löschen von $k = 11$



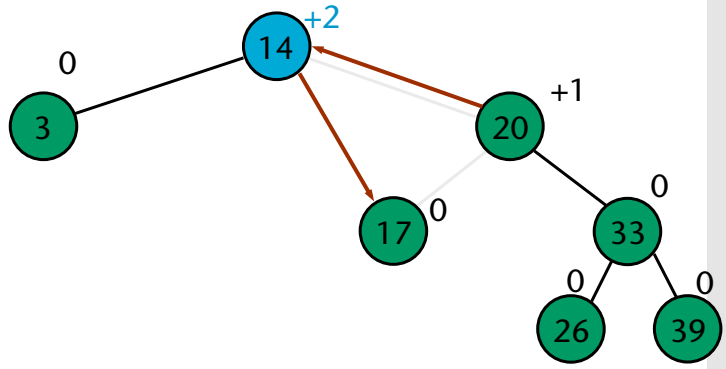
Beispiel: Löschen von $k = 11$



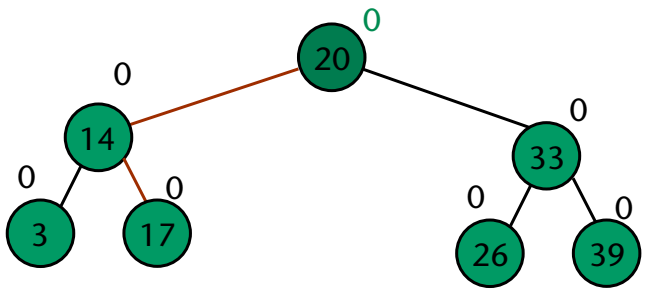
Beispiel: Löschen von $k = 11$



Beispiel: Löschen von $k = 11$



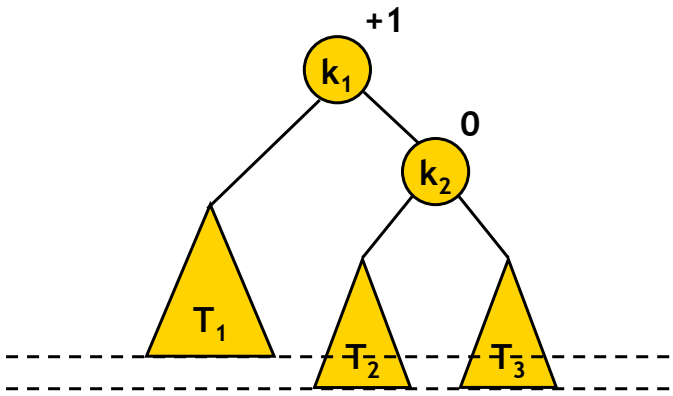
Beispiel: Löschen von $k = 11$

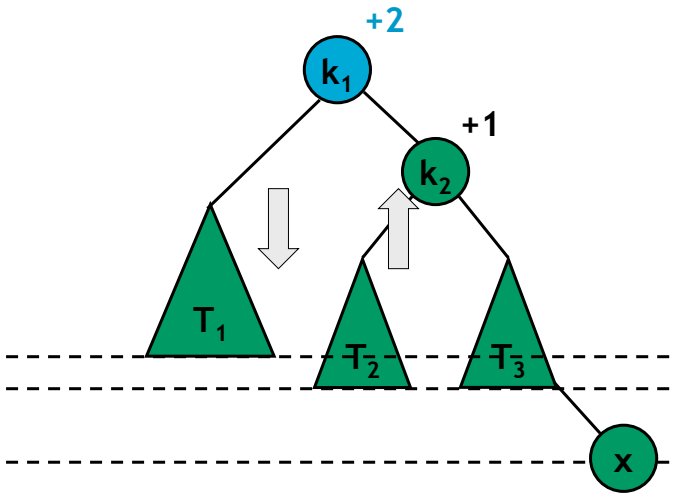


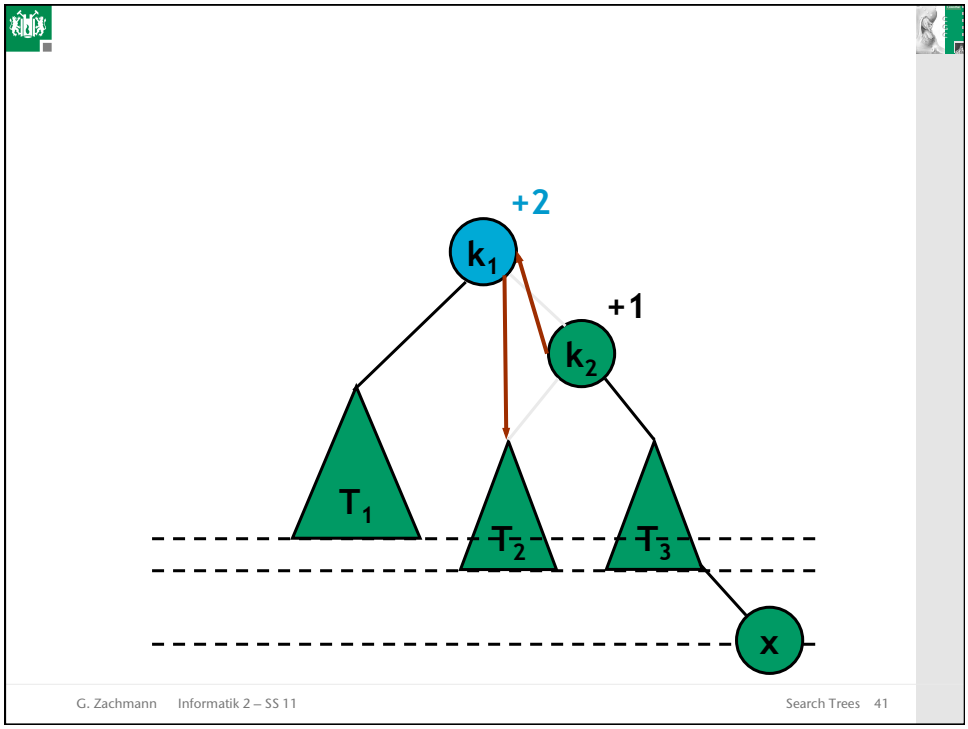
AVL-Rotationen

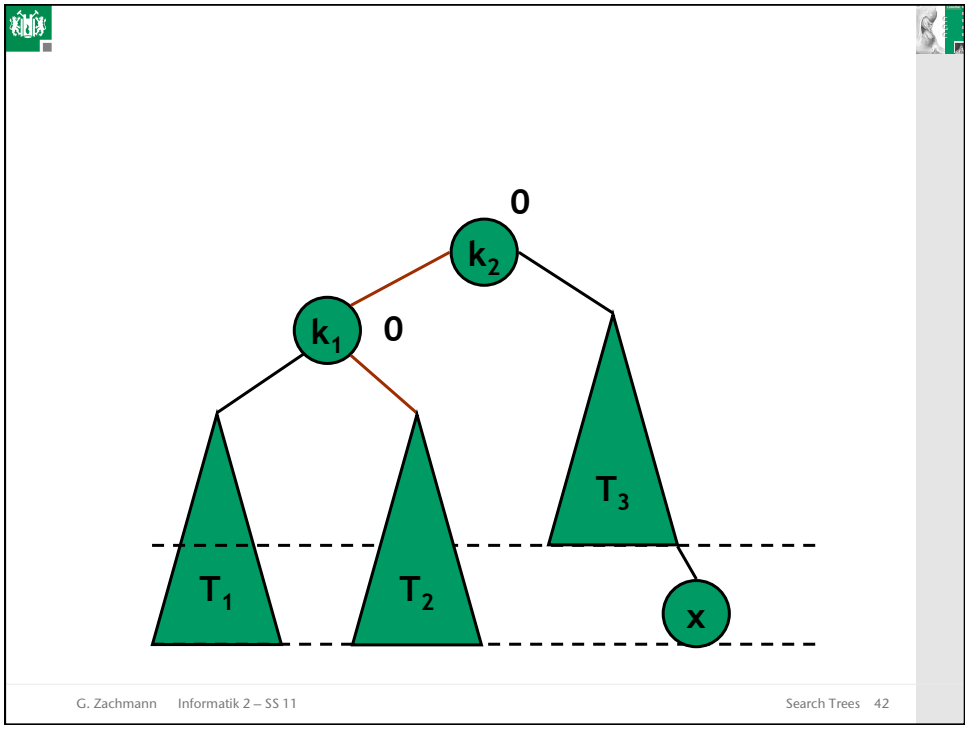
- Operationen auf AVL-Bäumen zur Erhaltung der AVL-Eigenschaft
- Bestehen ausschließlich aus “Umhängen” von Zeigern
- Es gibt 2 verschiedene Arten von Rotationen
  - *Double Rotation*:
    - RL = neuer Knoten ist im linken Unterbaum des rechten Unterbaumes
    - M.a.W.: vom Knoten mit dem "schlechten" Balancefaktor muß man in den rechten Teilbaum gehen, dann von da aus in den linken Teilbaum, dann kommt man zu dem neu eingefügten Knoten
    - LR = analog
  - *Single Rotation*: RR und LL
    - RR = der neue Knoten befindet sich im rechten Teilbaum des rechten Teilbaums vom (jetzt) unbalancierten Knoten aus
    - LL = analog
    - Wird manchmal auch einfach nur R- bzw. L-Rotation genannt

RR-Rotation



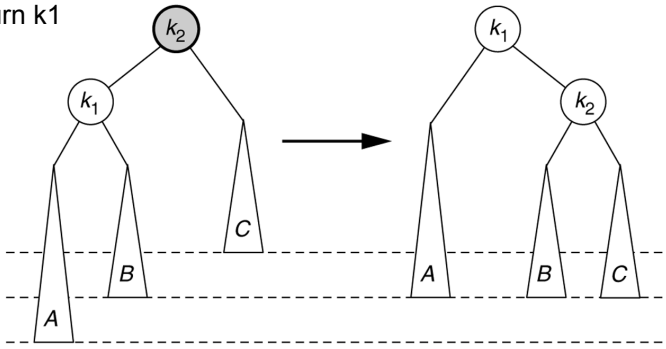



+2
k1
+1
k2
T1
T2
T3
x


0
k2
k1
0
T3
T1
T2
x

LL Rotation Algorithm

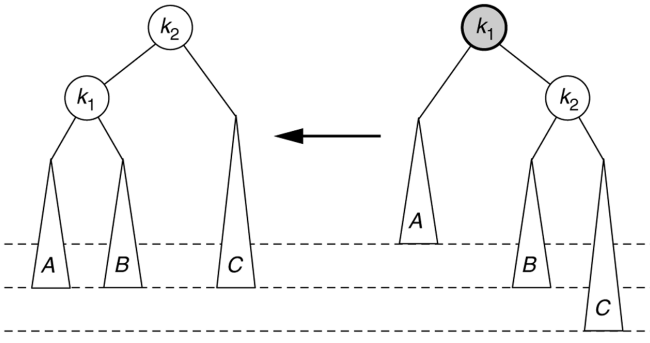
- def LL_Rotate (k2):
  k1 = k2.left
  k2.left = k1.right
  k1.right = k2
  return k1



(a) Before rotation (b) After rotation

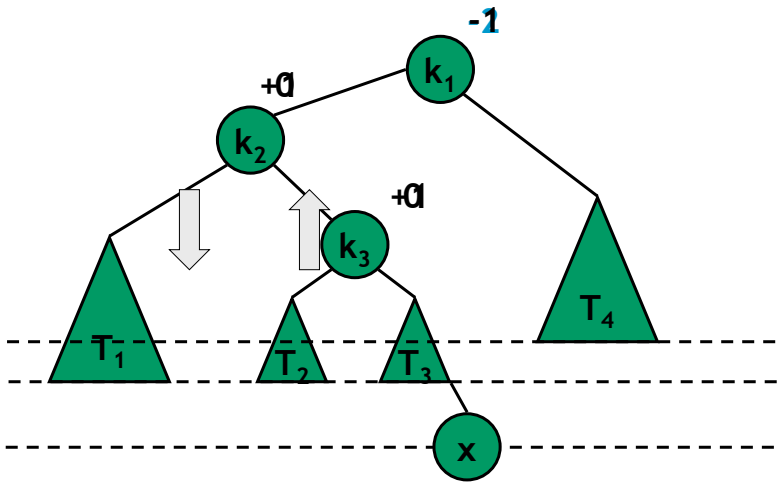
RR Rotation Algorithm

```
def RR_Rotate( k1 ):
k2 = k1.left
k1.right = k2.left
k2.left = k2
return k2
```

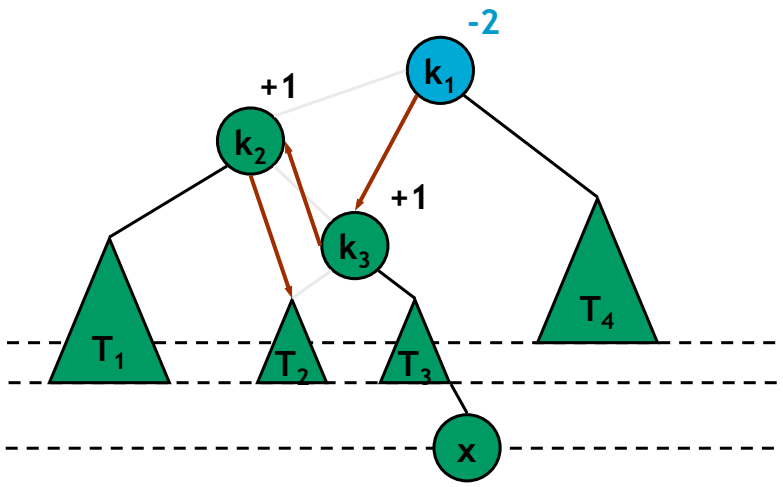


(a) After rotation (b) Before rotation

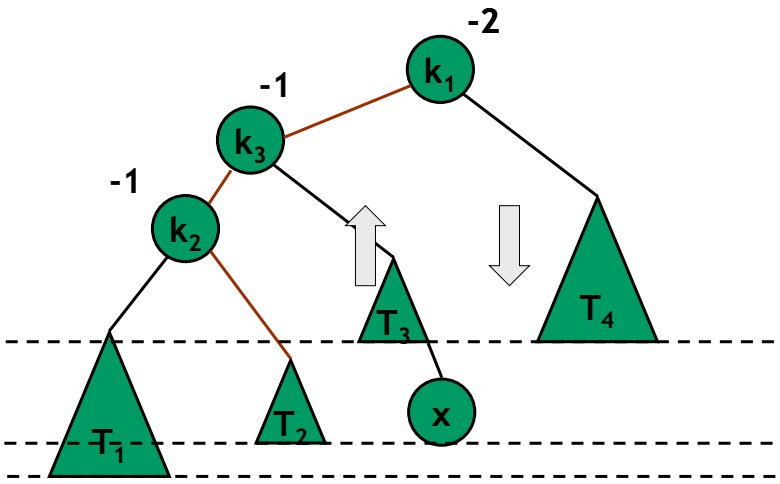
LR-Rotation



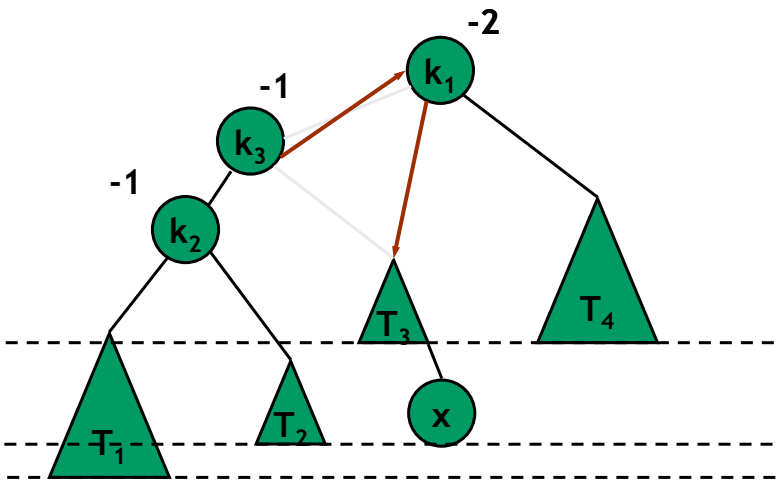
LR-Rotation



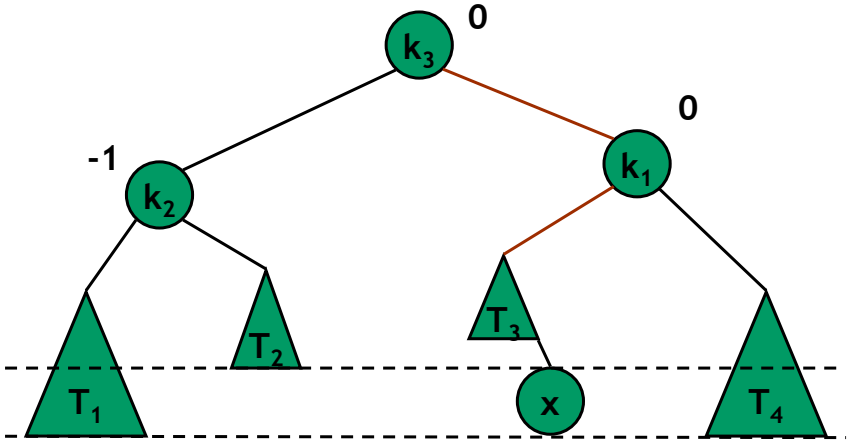
LR-Rotation



LR-Rotation

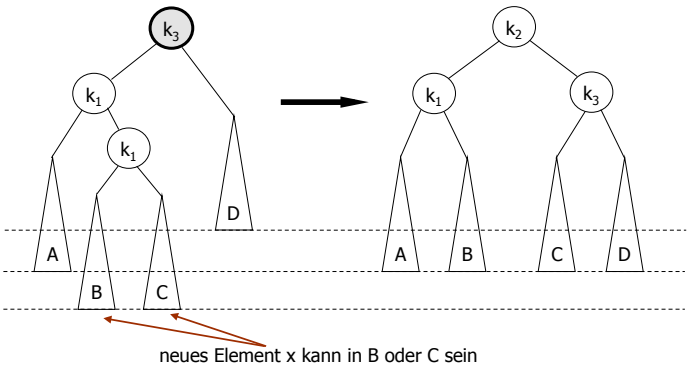


LR-Rotation



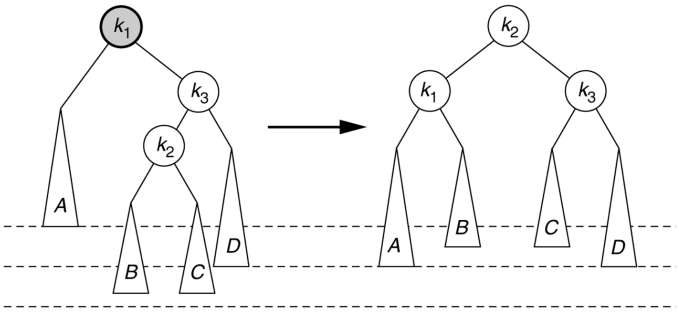
Pseudo-Code

```
def LR_doubleRotate( k3 ):
  k3.left = RR_Rotate( k3.left )
  return LL_Rotate( k3 )
```



(a) Before rotation (b) After rotation

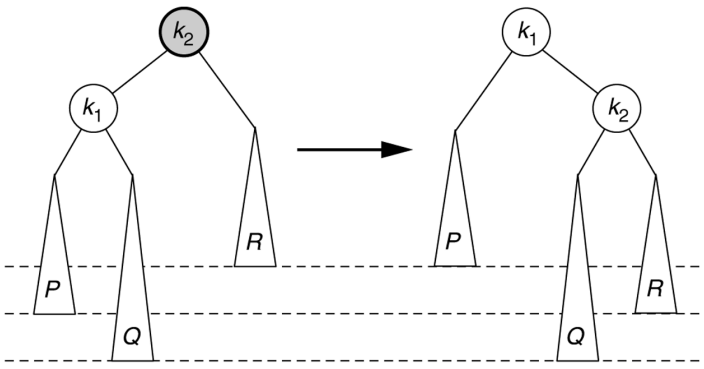
```
def RL_doubleRotate( k1 ):
  k1.right = LL_Rotate( k1.right )
return RR_Rotate( k1 )
```



(a) Before rotation (b) After rotation

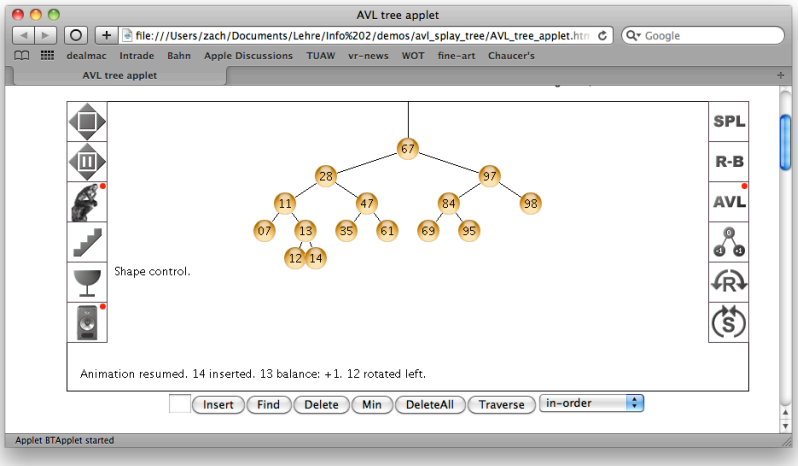
Warum Double Rotation?

- Single Rotation kann LR oder RL nicht lösen:



(a) Before rotation (b) After rotation

Algo-Animation



http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm

Optimale Suchbäume

- Beispiel: Wörterbuch Englisch → Französisch
- Mit AVL-Bäumen oder perfekt balancierten Bäumen bekommt man $O(\log n)$ worst-case Lookup-Zeit
- Typische Anwendung: Übersetzung eines englischen Textes
- Folge: manche Wörter werden wesentlich häufiger als andere nachgesehen (Erinnerung: Huffman-Codierung)
- Ziel: Gesamtzeit zum Übersetzen eines Textes soll möglichst klein sein, d.h., Gesamtzeit für Lookup aller Wörter des Textes soll klein sein
  - M.a.W.: die durchschnittliche Lookup-Zeit pro Wort soll klein sein
- Fazit: häufige Wörter müssen "eher weiter oben" an der Wurzel stehen

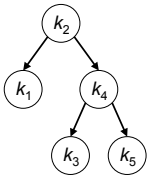
Problemstellung

- Gegeben sei Folge $k_1 < k_2 < ... < k_n$ von n sortierten Keys, mit einer Suchwahrscheinlichkeit p_i für jeden Key k_i
- Es soll ein binärer Suchbaum (BST) mit minimalen zu erwartenden Suchkosten erstellt werden
- Kosten eines Lookup = Anzahl der besuchten Knoten im Baum
- Für jeden Key k_i im Baum T sind die Kosten = $d_T(k_i)$, mit $d_T(k_i)$ = Tiefe von k_i im BST T , wobei $d_T($ *Wurzel* $) = 0$
- Also: erwartete (= mittlere) Kosten für die Suche (eines Keys) ist

$$E[\text{Suchkosten in } T] = \sum_{i=1}^{n} d_T(k_i) p_i$$

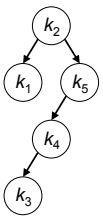
Beispiel

- Gegeben seien 5 Keys $k_1 < k_2 < k_3 < k_4 < k_5$ mit den Such-wahrscheinlichkeiten: $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$



i	$d_T(k_i)$	$d_T(k_i)\cdot p_i$
1	2	0.5
2	1	0.2
3	3	0.15
4	2	0.4
5	3	0.9
		2.15

→ E[Suchkosten] = 2.15

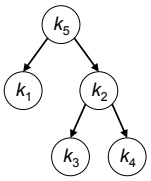


i	$d_T(k_i)$	$d_T(k_i)\cdot p_i$
1	2	0.5
2	1	0.2
3	4	0.2
4	3	0.6
5	2	0.6
		2.1

→ E[Suchkosten] = 2.1

(Dies ist übrigens der optimale BST für diese Key-Menge)

- Gegeben seien 5 Keys $k_1 < k_2 < k_3 < k_4 < k_5$ mit den Such-wahrscheinlichkeiten: $p_1 = 0.25,\ p_2 = 0.2,\ p_3 = 0.05,\ p_4 = 0.2,\ p_5 = 0.3$
- Ist folgender Baum nicht noch viel besser?!



i	$d_T(k_i)$	$d_T(k_i)\cdot p_i$
1	2	0.5
2	2	0.4
3	3	0.15
4	3	0.6
5	1	0.3
		1.95

Ist noch viel besser!?

... ODER ??!!

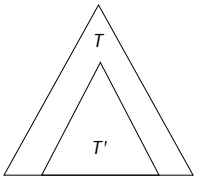
- Beobachtungen:
  - Der optimale BST muß nicht die kleinste Höhe haben
  - Der optimale BST muß nicht die größte Wahrscheinlichkeit an der Wurzel haben
- Erstellen durch erschöpfendes Testen? (*exhaustive enumeration*)
  - Erstelle alle möglichen BSTs mit n Knoten
  - Für jeden BST: verteile die Schlüssel und berechne die zu erwartenden Suchkosten
  - Erinnerung: Es gibt aber

$$C_n \in \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

verschiedene BSTs mit n Knoten

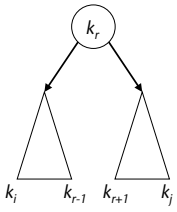
Die optimale Unterstruktur

1. Jeder Unterbaum eines BST beinhaltet Schlüssel in einem zusammenhängenden Bereich $k_i, ..., k_j$ für $1 \le i \le j \le n$
2. Wenn T ein optimaler BST ist und den Unterbaum T' enthält, dann ist auch T' ein optimaler BST für die Keys $k_i, ..., k_j$



- Beweis: "Cut and paste", d.h., Beweis durch Widerspruch:
  - Annahme: T' ist nicht optimal für seine $k_i, ..., k_j$
  - Dann existiert ein T'' für die $k_i, ..., k_j$, der besser als T' ist
  - Sei $\hat{T}$ der BST, der aus T entsteht, indem T' durch T'' ersetzt wird
  - $\hat{T}$ ist korrekter BST und hat außerdem kleinere mittlere Suchkosten als T $\Rightarrow$ W!

3. Einer der Schlüssel $k_i, ..., k_j$, z.B. k_r mit $i \le r \le j$, muß die Wurzel eines optimalen Unterbaumes für diese Schlüssel sein
   - Der linke Unterbaum von k_r enthält $k_i, ..., k_{r-1}$
   - Der rechte Unterbaum von k_r enthält $k_{r+1}, ..., k_j$

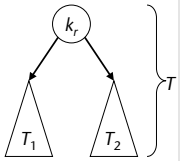


4. Zum Finden eines optimalen BST:
   - Betrachte alle Knoten k_r ($i \le r \le j$), die als Wurzel in Frage kommen
   - Bestimme die optimalen BSTs für $k_i, ..., k_{r-1}$ und $k_{r+1}, ..., k_j$

Eine rekursive Lösung

- Definiere $e_{i,j}$:= erwartete Suchkosten des optimalen BST für $k_i, \ldots, k_j$
- Nützliche Verallgemeinerung:
  - Es wird nicht mehr verlangt, daß $p_i,\ldots, p_j$ eine Wahrscheinlichkeitsverteilung bilden; es darf also gelten
$$\sum_{l=i}^{j} p_l \neq 1$$
  - Gesucht ist aber weiterhin ein BST für $k_i, \ldots, k_j$, so daß die Summe
$$e_{ij} = \sum_{l=i}^{j} d_T(k_l) p_l$$
  minimiert wird
- Diese Summe wird weiterhin erwarteter Suchaufwand genannt

- Wenn k_r die Wurzel eines optimalen BST für $k_i, \ldots, k_j$:



$$e[i,j] = \sum_{l=i}^{j} d_T(k_l) p_l$$

$$= \sum_{l=i}^{r-1} (d_{T_1}(k_l) + 1) p_l + p_r + \sum_{l=r+1}^{j} (d_{T_2}(k_l) + 1) p_l$$

$$= \underbrace{\sum_{l=i}^{r-1} d_{T_1}(k_l) p_l}_{e[i,r-1]} + \underbrace{\sum_{l=r+1}^{j} d_{T_2}(k_l) p_l}_{e[r+1,j]} + \sum_{l=i}^{j} p_l$$

$$= e[i, r-1] + e[i, r+1] + w(i,j)$$

- Aber k_r ist nicht bekannt, daher gilt
$$e[i,j] = \min_{i \leq r \leq j} \{ e[i, r-1] + e[i, r+1] + w(i,j) \}$$

Bottom-up-Berechnung einer optimalen Lösung

- Erstelle iterativ folgende 3 Tabellen
- Speichere für jedes Unterproblem (i,j) mit $1 \le i \le j \le n$:
  1. e[i,j] = zu erwartende Suchkosten
  2. $r[i,j]$ = Wurzel des Teilbaums mit den Schlüsseln $k_i, \ldots, k_j$, $i \le r[i,j] \le j$
  3. $w[i,j] \in [0,1]$ = Summe der Wahrscheinlichkeiten
     - $w[i,i] = p_i$ für $1 \le i \le n$
     - $w[i,j] = w[i,j\text{-}1] + p_j = p_i + w[i\text{+}1,j]$ für $1 \le i \le j \le n$

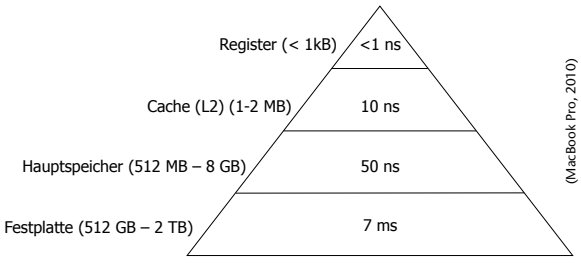

```
def optimal_bst( p,q,n ):
    for i in range( 1,n ):
        e[i,i] = w[i,i] = p[i]
    for l in range( 2,n ):# calc all opt trees w/ l keys
        for i in range( 1, n-l ):          # n - ell
            j = i+l-1
            e[i,j] = ∝                     # z.B. 2^31-1
            w[i,j] = w[i,j-1] + p[j]
            for r in range( i, j+1 ):
                t = e[i,r-1] + e[r+1,j] + w[i,j]
                if t < e[i,j]:
                    e[i,j] = t
                    r[i,j] = r
    return e,r
```

- Laufzeit ist offensichtlich $O(n^3)$

- Satz: Ein optimaler Suchbaum für n Keys mit gegebenen Zugriffshäufigkeiten kann in Zeit $O(n^3)$ konstruiert werden.

Mehrwegbäume — Motivation

- Wir haben gute Strukturen (AVL-Bäume) kennen gelernt, die die Anzahl der Operationen begrenzen
- Was ist, wenn der Baum zu groß für den Hauptspeicher ist?
- Externe Datenspeicherung → Speicherhierarchie:



- Verhältnis zwischen Zugriffszeit auf Hauptspeicher und Zugriffszeit auf Festplatte ≈ 10^5

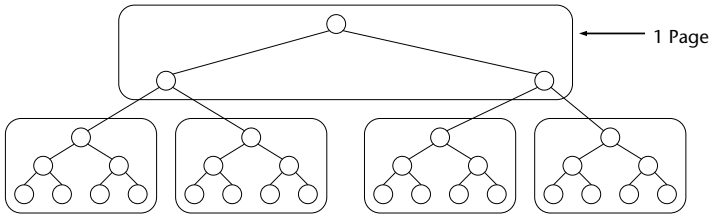
- D.h., Zugriff auf (externe) Knoten ist seeeehr teuer
- Bsp. :
  - File-Tree des File-Systems
  - Datenbanken (DB2 (IBM), Sybase, Oracle, …)
- Man hätte gerne einen Baum, der noch geringere Tiefe hat als

$$\log_2(n)$$

- Idee:
  - "Erhöhe die Basis des Logarithmus"
  - Oder: speichere mehrere "aufeinanderfolgende" Baum-Knoten (= ein "Bäumchen") auf derselben Seite (*Page / Sector*) auf der Platte
  - Reduziere damit die Anzahl der Seitenzugriffe

m-Wege-Bäume

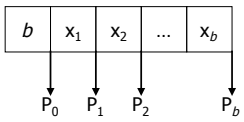
- Ein m-Wege-Suchbaum ist eine Verallgemeinerung eines binären Suchbaumes (d. h., ein binärer Suchbaum ist ein 2-Wege-Suchbaum)



Anstatt 5 nur noch 2 Zugriffe auf die Platte / CD

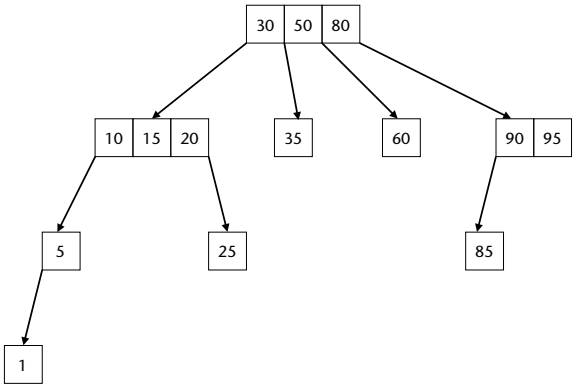
Definition

- In einem *m*-Wege-Baum haben alle Knoten den Grad ≤ *m*
- Der Baum ist entweder leer oder besitzt folgende Eigenschaften:
  - Jeder Knoten hat folgende Struktur:
    - x_i sind die Keys, $1 \le i \le b$
    - *b* ist die Anzahl der Schlüssel im Knoten
    - P_i sind die Zeiger zu den Unterbäumen des Knotens



  - Die Schlüssel innerhalb eines Knotens sind aufsteigend geordnet:
    - $x_1 \le x_2 \le \ldots \le x_b$
  - Alle Schlüssel im Unterbaum P_i sind kleiner als x_{i+1} , für $0 \le i < b$
  - Alle Schlüssel im Unterbaum P_b sind größer als x_b
  - Die Unterbäume P_i , für $0 \le i \le b$, sind ebenfalls *m*-Wege-Bäume

Beispiel



Bemerkungen

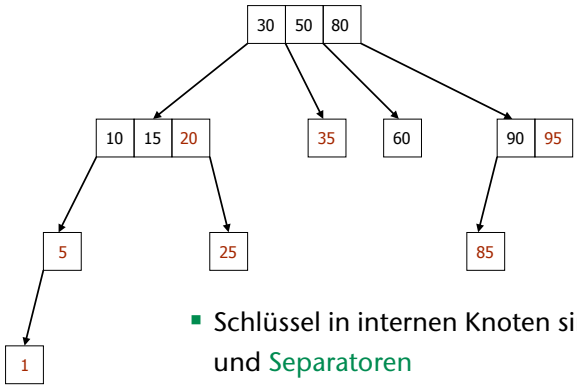
- Probleme:
  - Der Baum ist nicht ausgeglichen (balanciert)
  - Die Blätter sind auf verschiedenen Stufen
  - Bei Veränderungen gibt es keinen Ausgleichsalgorithmus (à la AVL)
  - Kann also zu verketteten Listen degenerieren
- Anzahl der Knoten im vollständigen *m*-Wege Baum mit Höhe *h*:

$$N = \sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m-1}$$

- Maximale Anzahl *n* von Keys:
  - Pro Knoten im *m*-Wege Baum hat man maximal *m*-1 Keys
  - Also: $n \leq (m-1)N = m^h - 1$
- Im völlig degenerierten Baum ist $N = h$, also $n = (m-1)h$
- Schranken für *h* im *m*-Wege-Baumes: $\log_m(n+1) \leq h \leq \frac{n}{m-1}$

Beispiel: Einfügen in einen 4-Wege-Baum

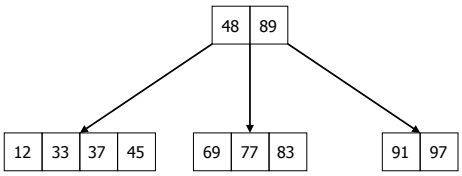
- Einfügen von 20, 35, 5, 95, 1, 25, 85



- Schlüssel in internen Knoten sind Schlüssel und Separatoren
- Innerhalb eines Knotens: sequentielle Suche, auch binäre Suche möglich

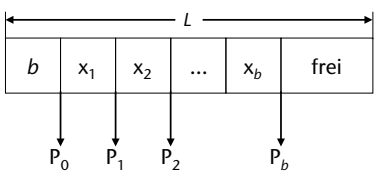
B-Bäume

- Ist vielleicht die Datenstruktur, die in der Praxis am meisten benutzt wird, nämlich in Filesystemen
- B-Bäume sind ausgeglichene m-Wege-Bäume
- Da ein Knoten die Größe einer Übertragungseinheit hat (eine Page der Festplatte, typ. 4-16 kBytes), sind 100-200 Keys pro Knoten üblich
  - Ergibt sehr flache Bäume = kurzer Weg von der Wurzel zu den Blättern

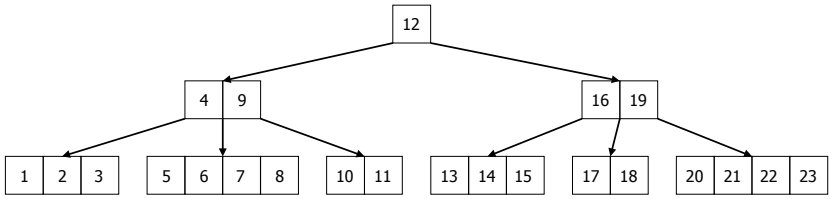


Definition von B-Bäumen

1. Jeder Weg von der Wurzel zu einem Blatt ist gleich lang
2. Jeder Knoten (außer Wurzel) enthält mindestens k und höchstens 2k Keys
3. Jeder Knoten (außer der Wurzel) hat zwischen k+1 und 2k+1 Kinder (Unterbäume)
4. Die Wurzel ist entweder ein Blatt oder hat mindesten 2 Kinder
5. Jeder Knoten hat die Struktur:
   - b ist die Anzahl der Keys im Knoten, k ≤ b ≤ 2k
   - Die Keys im Knoten sind aufsteigend sortiert ($x_1 < x_2 < \ldots < x_b$)
   - (L = Größe einer Page)



- Die Klasse $B(k,h)$ bezeichnet alle B-Bäume mit dem Parameter k und der Höhe h ($h \geq 0$ und $k>0$)
  - Höhe der Wurzel = 1
- Beispiel: B-Baum der Klasse $B(2,3)$



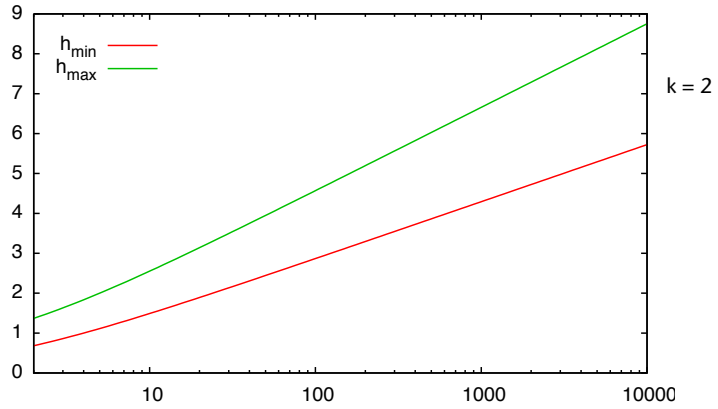
Bemerkungen

- CLRS benutzt eine etwas andere Definition:
  - Für einen B-Baum vom Grad t gilt:
    - jeder Knoten, außer der Wurzel, hat mindestens t-1 Schlüssel, also mindestens t Kinder
    - jeder Knoten besitzt höchstens $2t$-1 Schlüssel, also höchsten $2t$ Kinder
  - Formeln müssen „angepasst“ werden, damit sie stimmen
  - die Prüfung läuft gemäß den Folien, bei Unsicherheiten die Definition angeben

Höhe eines B-Baumes

- Für die Höhe eines B-Baumes mit *n* Schlüsseln gilt:

$$\log_{2k+1}(n+1) \leq h \leq \log_{k+1}\left(\frac{n+1}{2}\right) + 1$$



Erklärung

- In einem minimalen B-Baum hat jeder Knoten die kleinstmögliche Anzahl an Kindern (= k+1); also gilt für die min. Anzahl Knoten

$$N_{\min}(k,h) = 1 + 2 + 2\left((k+1) + (k+1)^2 + \cdots + (k+1)^{h-2}\right)$$
$$= 1 + 2\sum_{i=0}^{h-2}(k+1)^i = 1 + 2\frac{(k+1)^{h-1}-1}{k}$$

- In einem maximalen B-Baum hat jeder Knoten die größtmögliche Anzahl an Kindern (= 2k+1); also gilt für die max. Anzahl Knoten

$$N_{\max}(k,h) = 1 + (2k+1) + (2k+1)^2 + \cdots + (2k+1)^{h-1}$$
$$= \sum_{i=0}^{h-1}(2k+1)^i = \frac{(2k+1)^h - 1}{2k}$$

- Die Höhe definiert eine obere und untere Schranke für die Anzahl der Knoten $N(B)$ eines beliebigen B-Baumes der Klasse $\tau(k,h)$:

$$1 + 2\frac{(k+1)^{h-1}-1}{k} \leq N(B) \leq \frac{(2k+1)^h-1}{2k} \quad \text{für } h \geq 1$$
$$N(B) = 0 \quad \text{für } h = 0$$

- Von $N_{min}(B)$ und $N_{max}(B)$ lässt sich ableiten:

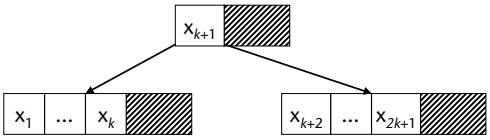
$$n \geq n_{min} = 1 + 2\frac{(k+1)^{h-1}-1}{k}k = 2(k+1)^{h-1} - 1$$
$$n \leq n_{max} = \frac{(2k+1)^h-1}{2k}2k = (2k+1)^h - 1$$

- Somit gilt für die Anzahl Keys n in einem B-Baum:

$$2(k+1)^{h-1} - 1 \leq n \leq (2k+1)^h - 1$$

Algorithmus zum Einfügen in einen B-Baum

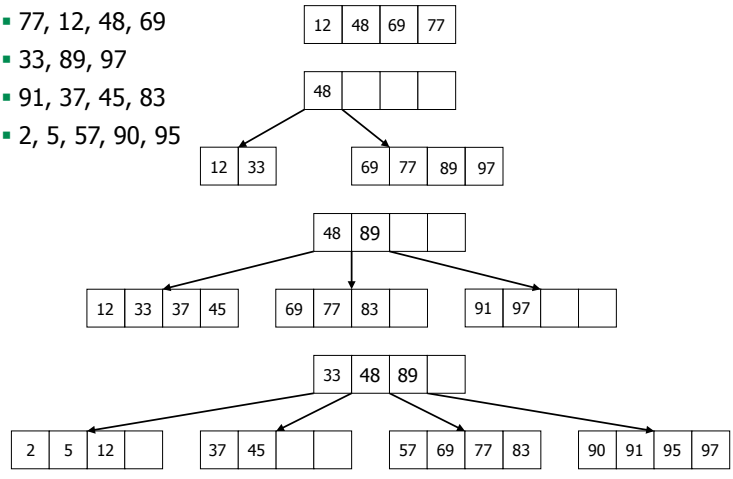
- Füge anfangs in ein leeres Feld der Wurzel ein
- Die ersten 2*k* Schlüssel werden sortiert in die Wurzel eingefügt
- Der nächste, der (2*k*+1)-te, Schlüssel passt nicht mehr in die Wurzel und erzeugt einen sog. "Überlauf" (*overflow*)
- Die Wurzel wird geteilt (ein sog. "Split"):
  - Jeder der beiden Kinder bekommt *k* Keys
    - Die ersten *k* Schüssel kommen in den linken Unterbaum
    - Die letzen *k* Schüssel kommen in den rechten Unterbaum
  - Die Wurzel bekommt 1 Key
    - Der Median der (2*k*+1) Schlüssel wird zum neuen Separator

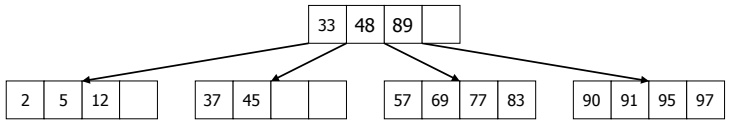


- Neue Schlüssel werden in den Blättern sortiert gespeichert
- Läuft ein Blatt über ($2k$+1 Keys), wird ein Split durchgeführt:
  - Das Blatt wird geteilt, ergibt 2 neue Blätter à k Keys;
  - diese werden im Vaterknoten anstelle des ursprünglichen Blattes verlinkt;
  - der Median-Key wandert in den Vaterknoten.
- Ist auch der Vaterknoten voll, wird das Splitten fortgesetzt
  - Im worst-case bis zur Wurzel

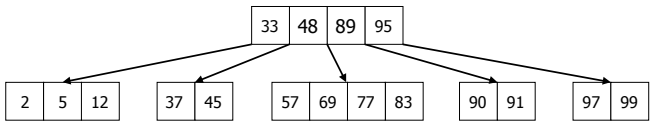
Beispiel

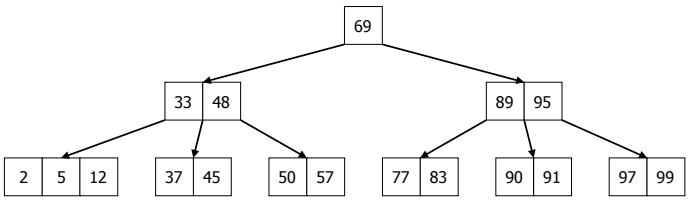
- k = 2 → 2 ≤ #Schlüssel ≤ 4 (außer Wurzel)
- Einzufügende Schlüssel:
  - 77, 12, 48, 69
  - 33, 89, 97
  - 91, 37, 45, 83
  - 2, 5, 57, 90, 95





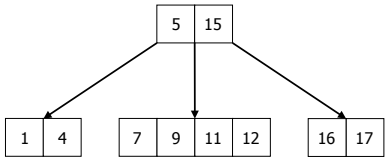
- Einzufügende Schlüssel:
  - 99
  - 50

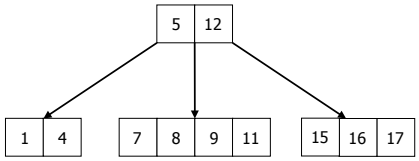


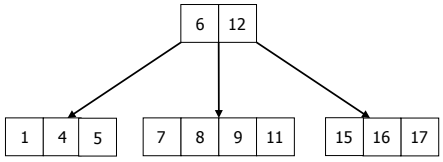


Der B*-Baum

- Variante des B-Baumes:
  - Selbe Datenstruktur
  - Algo zum Einfügen ist variiert
- Idee: zuerst Ausgleich mit Nachbarknoten statt Split
- Beispiel:
  - Füge ein:
  - 8
  - 6

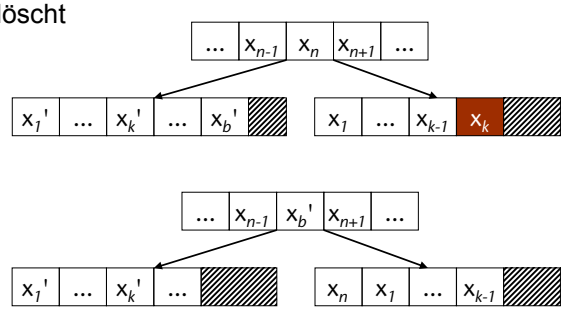




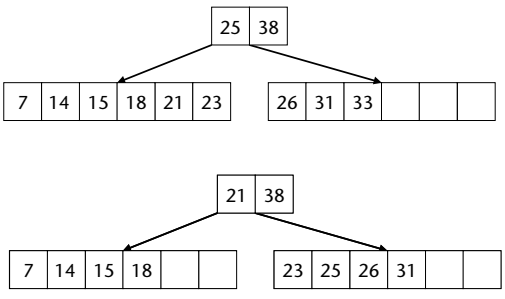


Löschen in einem B-Baum

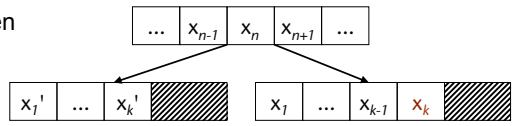
- Bedingung: minimale Belegung eines Knotens erhalten
  - Bei einem sog. *Underflow* muß ein Ausgleich geschaffen werden
- Methoden:
  - Rotation = Ausgleich mit Nachbarknoten
  - Merge = Gegenteil von Split
- Beispiel: x_k wird gelöscht
  - Wende Rotation an
  - Key x_b' wird in den Vaterknoten verschoben, Key x_n kommt in den rechten Knoten (mit k-1 Knoten)



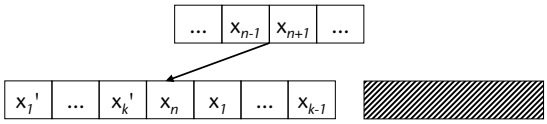
- Um wiederholte Rotationen (durch neue Löschoperationen) zu vermeiden, kann man mehr als einen Key rotieren
  - Beide Knoten haben danach ungefähr die gleiche Key-Anzahl
- Beispiel: 33 soll entfernt werden (k = 3)



- Zweite Art der Underflow-Behandlung: Merge
  - Wird benutzt, wenn es einen Underflow gibt und beide Nachbarknoten genau *k* Keys haben



- Beispiel: x_k löschen



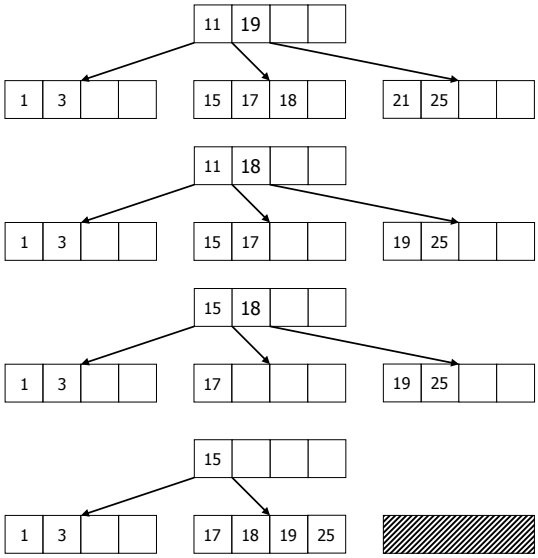
  - Die beiden Nachbarblätter und der Separator werden ge-merge-t; ergibt genau $k + 1 + (k-1) = 2k$ Keys

Allgemeiner Lösch-Algorithmus

- x = zu löschendes Element
- Suche x; unterscheide 2 Fälle:
  1. x ist in einem Blatt:
     - # Keys ≥ *k* (nach dem Löschen) → OK
     - # Keys = *k*-1 und # Keys in einem (direkten) Nachbarblatt > *k* → Rotation
     - Sonst → Merge
  2. x ist in einem internen Knoten:
     a) Ersetze den Separator x durch x_b oder x_1' des linken bzw. rechten Kind-Knotens (= nächstgößerer oder -kleiner Schlüssel)
     b) Lösche diesen Seperator (x_b oder x_1') aus dem entsprechenden Kind-Knoten (Rekursion)

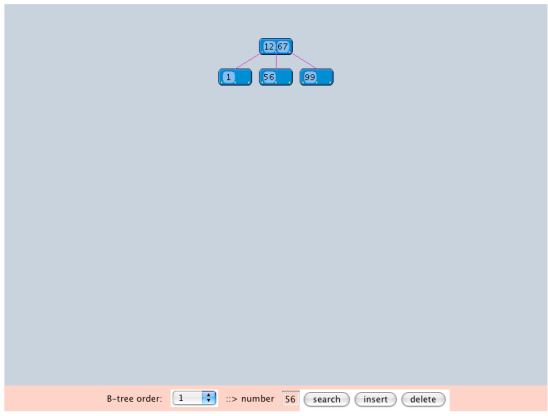
Beispiel (k = 2)

- Lösche:
- 21
- 11 → Underflow
- → Merge



Demo

B-Tree animation applet:



http://slady.net/java/bt/

Komplexitätsanalyse für B-Bäume

- Anzahl der Key-Zugriffe:
  - Z.B. Suche ≈ Pfad von Wurzel zu Knoten/Blatt
  - Falls lineare Suche im Knoten → $O(k)$ pro Knoten
  - Insgesamt $O(k \log_k n)$ Zugriffe im Worst-Case
- Aber: wirklich teuer sind read/write-Operationen auf die Platte
- Annahmen: jeder Knoten benötigt nur 1 Zugriff auf den externe Speicher und jeder modifizierte Knoten wird nur 1x geschrieben
- Schranken für die Anzahl Lesezugriffe f beim Suchen (f = „fetch“):

$f_{\text{min}} = 1$	Key ist in der Wurzel
$f_{\text{max}} = h \in O\left(\log_k(n)\right)$	Key ist in einem Blatt
$h - \frac{1}{k} \leq f_{\text{avg}} \leq h - \frac{1}{2k}$	ohne Beweis

Kosten für Insert

- f = Anzahl "Fetches", w = Anzahl "Writes"
- Unterscheide 2 extremale Fälle:
  - Kein Split: f_{min} = h, w_{min} = 1
  - Wurzel-Split: f_{max} = h, w_{max} = 2h+1
- Abschätzung der Split-Wahrscheinlichkeit:
  - Wahrscheinlichkeit, daß ein Knoten gesplittet werden muß = Wahrscheinlichkeit, daß der Knoten exakt $2k$ viele Keys enthält
  - Sei X = Anzahl Keys in einem Knoten
  - Es gilt: $X \in \{k, ..., 2k\}$
  - Für die Split-Wahrscheinlichkeit gilt also:

$$p_{\text{split}} = \frac{\text{Anzahl günstige}}{\text{Anzahl mögliche}} = \frac{1}{k}$$

- Eine Insert-Operation erfordert das Schreiben einer Seite, eine Split-Operation erfordert ungefähr 2 Schreib-Operationen
- Durchschnittlicher Aufwand für Insert:

$$
\begin{aligned}
w_{avg} &= 1 + p_{split} + \ p_{split}(1 + p_{split}) + \ p_{split}^2(\ldots) + \ \ldots \\
&= (1 + p_{split}) \sum_{i=0}^{h} p_{split}^i \\
&\leq (1 + p_{split}) \sum_{i=0}^{\infty} p_{split}^i \\
&= (1 + p_{split}) \frac{1}{1 - p_{split}} \\
&= \frac{k+1}{k-1}
\end{aligned}
$$

Kosten für Löschen

- Mehrere Fälle:
  1. Best-Case: kein „Underflow“
     f_{min} = h w_{min} = 1
  2. "Underflow" wird durch Rotation beseitigt (keine Fortpflanzung):
     f_{rot} = h+1 w_{rot} = 3
  3. Merge ohne Fortpflanzung:
     f_{mix} = h+1 w_{mix} = 2
  4. Worst-Case: Merge bis hinauf zum Wurzel-Kind, Rotation beim Wurzel-Kind:
     f_{max} = 2h-1 w_{max} = h+1

Durchschnittliche Kosten für Löschen

- Obere Schranke unter der Annahme, daß alle Schlüssel nacheinander gelöscht werden
- Kein Underflow: f_1 = h, $w_1 \leq 2$
- Zusätzliche Kosten für die Rotation (höchstens ein Underflow pro gelöschtem Schlüssel): $f_2 = 1$, $w_2 \leq 2$
- Zusätzliche Kosten für das Merge:
  - 1 Mix-Operation = Elemente eines Knotens in Nachbarknoten mergen und Knoten löschen, d.h., Mix mit Propagation = bis zu h Mix-Op.
  - Maximale Anzahl von Mix-Operationen: $N(n) - 1 = \frac{n-1}{k}$
  - Kosten pro Mix-Operation: 1 „read“ und 1 „write“
  - zusätzliche Kosten pro Schlüssel:

$$f_3 = w_3 = \frac{\#\text{Mix-Op.}}{\#\text{Schlüssel}} = \frac{N(n)-1}{n} = \frac{(n-1)/k}{n} < \frac{1}{k}$$

- Addition ergibt:

$$f_{\text{avg}} \leq f_1 + f_2 + f_3 < h + 1 + \frac{1}{k}$$
$$w_{\text{avg}} \leq w_1 + w_2 + w_3 < 2 + 2 + \frac{1}{k} = 4 + \frac{1}{k}$$

Digitale Suchbäume & Binäre Tries

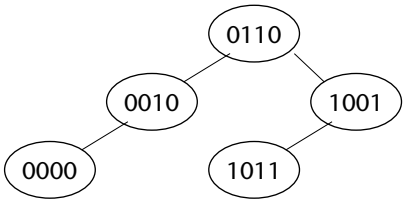
- Bisher: Traversierung durch Baum wurde gelenkt durch Vergleich zwischen gesuchtem Key und Key im Knoten
- Idee: lenke die Traversierung durch Bits/Ziffern/Zeichen im Key
  - Vergleiche die 2 großen Kategorien der Sortieralgorithmen
- Ab jetzt: Schlüssel sind Bitketten / Zeichenfolgen
  - Bei fester Länge → 0110, 0010, 1010, 1011
  - Bei variabler Länge → 01$, 00$, 101$, 1011$
- Anwendungen: z.B. IP-Routing, Paket-Klassifizierung, Firewalls
  - IPv4 – 32 bit IP-Adresse
  - IPv6 – 128 bit IP-Adresse

Digitale Suchbäume (DST = digital search trees)

- Annahme: feste Anzahl an Bits
- Konstruktion (= rekursive Definition):
  - Die Wurzel enthält irgendeinen Key
  - Alle Keys, die mit 0 beginnen, sind im linken Unterbaum
  - Alle Keys, die mit 1 beginnen, sind im rechten Unterbaum
  - Linker und rechter Unterbaum sind jeweils Digitale Suchbäume ihrer Keys

Beispiel

- Starte mit leerem Suchbaum:



  - füge den Schlüssel 0000 ein

- Bemerkungen:
  - Aussehen des Baumes hängt von der Reihenfolge des Einfügens ab!
  - Es gilt nicht: Keys(linker Teilb.) < Key(Wurzel) < Keys(rechter Teilb.) !
  - Es gilt aber: Keys(linker Teilbaum) < Keys(rechter Teilbaum)

- Algorithmus zum Suchen:

```
vergleiche gesuchten Key mit Key im Knoten
falls gleich:
    fertig
sonst:
    vergleiche auf der i-ten Stufe das i-te Bit
    1 → gehe in rechten Unterbaum
    0 → gehe in linken Unterbaum
```

- Anzahl der Schlüsselvergleiche = O(Höhe) = O(# Bits pro Key)
- Komplexität aller Operationen (Suchen, Einfügen, Löschen) :
  $O((\#\text{ Bits pro Schlüssel})^2)$
- Ergibt hohe Komplexität wenn die Schlüssellänge sehr groß ist

Python-Implementierung

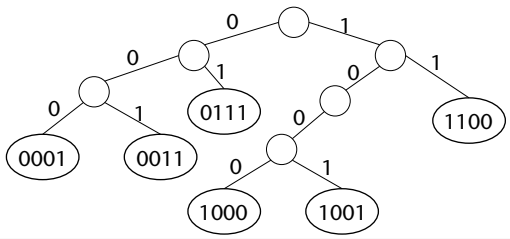
```
def digit(value, bitpos):
    return (value >> bitpos) & 0x01
```

```
def searchR(node, key, d):
    if node == None:
        return None
    if key == node.item.key:
        return node.item
    if digit(key, d) == 0:
        return searchR(node.left, key, d+1)
    else:
        return searchR(node.right, key, d+1)
```

```
class DigitalSearchTree:
    . . .
    def search(self, key):
        return searchR(self.root, key, 0)
```

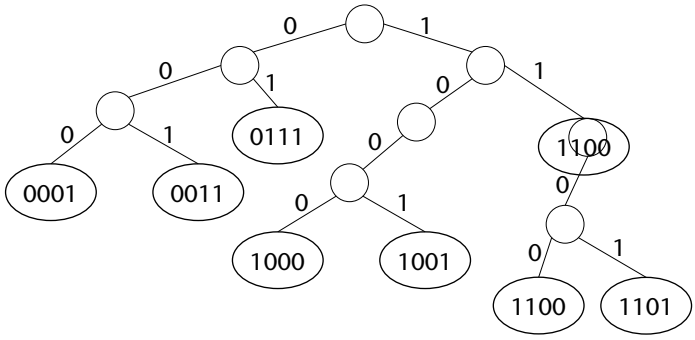

Binäre Tries

- Ziel: höchstens 1 kompletter Key-Vergleich pro Operation
- Zum Begriff:
  - "Trie" kommt von Information Re***trie***val
  - Aussprache wie "try"
- Es gibt 2 Arten von Knoten:
  - Verzweigungsknoten: hat nur linke und rechte Kindknoten, keine Keys
  - Elementknoten: keine Kindknoten, Datenfeld mit Schlüssel



Beispiel: Einfügen

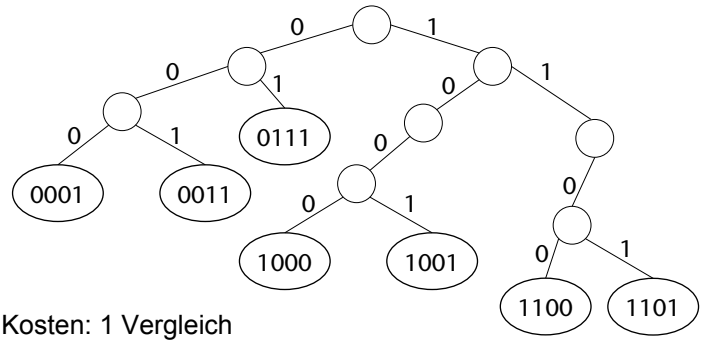
- Füge Schlüssel 1101 ein:



- Kosten: 1 Vergleich

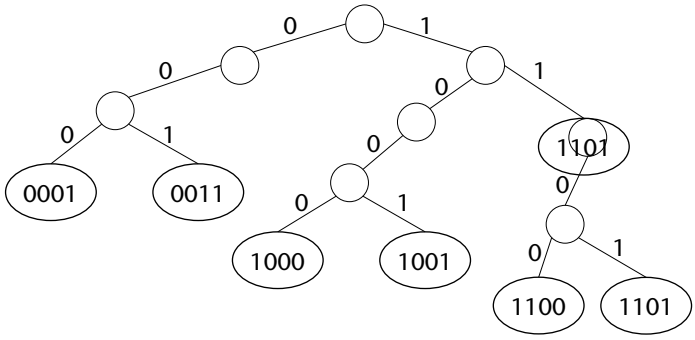
Beispiel: Entfernen

- Entferne Schlüssel 0111:



- Kosten: 1 Vergleich

- Entferne Schlüssel 1100:



- Kosten: 1 Vergleich

Implementierung

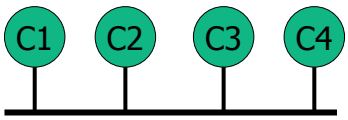
```
class Trie:
    . . .
    def insert(self,item):
        self.root = insertR(self.root, item, 0)
```

```
def insertR(node, item, d):
    if node == None:
        return TrieNode(item)
    if (node.left == None) and (node.right == None):
        return split( TrieNode(item), node, d )
    if digit(item.key, d) == 0:
        node.left = insertR(node.left, item, d+1)
    else:
        node.right = insertR(node.right, item, d+1)
    return node
```

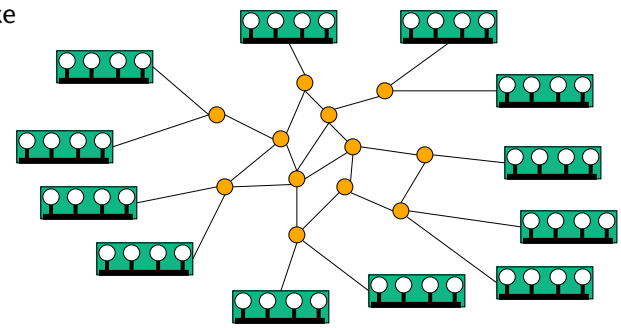
```
def split(nodeP, nodeQ, d):
    nodeN = TrieNode(None)
    splitcase = 2 * digit(nodeP.item.key, d)
                  + digit(nodeQ.item.key, d)
    if   splitcase == 0: # 00
        nodeN.left  = split(nodeP, nodeQ, d+1)
    elif splitcase == 3: # 11
        nodeN.right = split(nodeP, nodeQ, d+1)
    elif splitcase == 1: # 01
        nodeN.left  = nodeP
        nodeN.right = nodeQ
    elif splitcase == 2: # 10
        nodeN.left  = nodeQ
        nodeN.right = nodeP
    else:
        print "Can't happen!"
return nodeN
```


Anwendung: Routing in Netzwerken

- Netzwerk: miteinander verbundene Computer



- Internet = miteinander verbundene Netzwerke



Routing und Forwarding

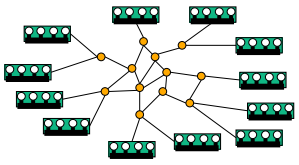
- Wie gelangt eine Nachricht von Computer A zu Computer B?
  - Beispiel Brief-Adresse: Straße/Hausnummer, Postleitzahl/Ort
    - 2 Schritte: zuerst die richtige Postleitzahl, dann das richtige Haus
  - Bei Computernetzwerken:
    - Zuerst muß das richtige Netzwerk ermittelt werden, dann der richtige Computer

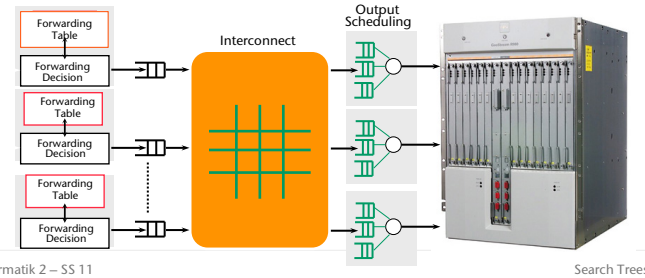
IP-Adressen

- Adresse eines Computers im Internet:
  - 32 Bits lang, meist in Punktnotation, z. B. 139.174.2.5
  - Besteht aus zwei Teilen: Netzwerkadresse und Host-Adresse
  - Beispiel: 139.174.2.5 =
    10001011101011100000001000000101
    Netzwerkadresse Host-Adresse
  - Anzahl der Bits für jeden Teil ist nicht festgelegt!
  - Um IP-Adressen richtig interpretieren zu können, muß man die Anzahl der Bits, mit denen das Netzwerk kodiert wird, kennen!
  - Wie man den Netzwerkanteil bestimmt, kommt demnächst
  - Länge der Netzwerkadresse bzw. die Netzwerkadresse selbst wird im folgenden Präfix heißen

Weiterleitung (Forwarding)

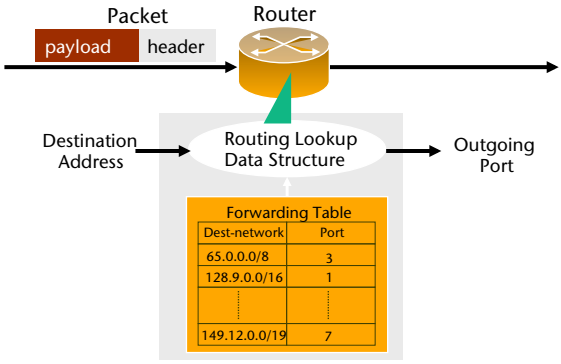
- Weiterleitung wird erledigt durch sog. *Router*
- Ein Router hat viele sog. *Ports*, die mit Computern oder anderen Routern verbunden sind
- Für eingehende Nachricht muß der Router entschieden, zu welchem Ausgangsport sie geschickt wird → hop-by-hop Weiterleitung
- Aufbau:





Der Router

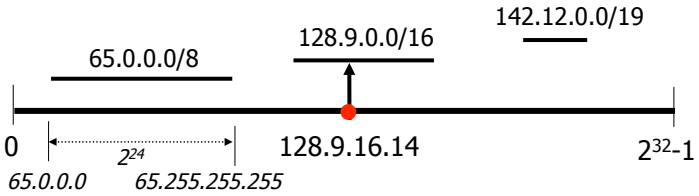
- Erstelle Routing-Tabelle mit "allen" Netzwerkadressen
- Routing-Tabelle verknüpft Netzwerkadressen mit Ausgangsports
- (Die Routing-Tabelle zu erstellen ist ein anderes Problem)



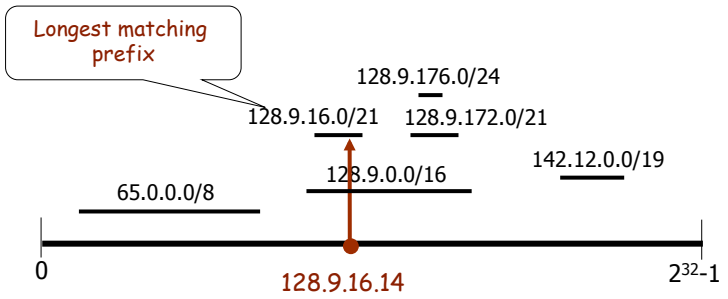
Beispiel einer Routing-Tabelle

Destination IP Prefix	Outgoing Port
65.0.0.0/8	3
128.9.0.0/16	1
142.12.0.0/19	7

Präfix-Länge



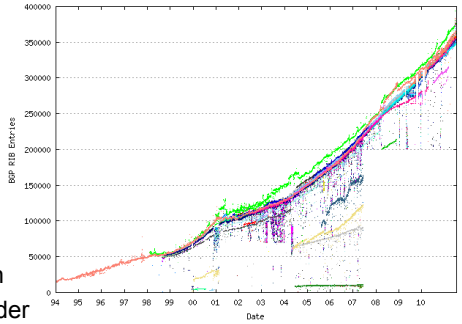
- Problem: Präfixe können überlappen!



- Eindeutigkeit wird durch Longest-Prefix-Regel gewährleistet:
  1. Vergleiche Adresse mit allen Präfixen
  2. Gibt es mehrer Präfixe, die matchen (bis zu ihrer Länge), wähle denjenigen Präfix, der am längsten ist (d.h., am "genauesten")

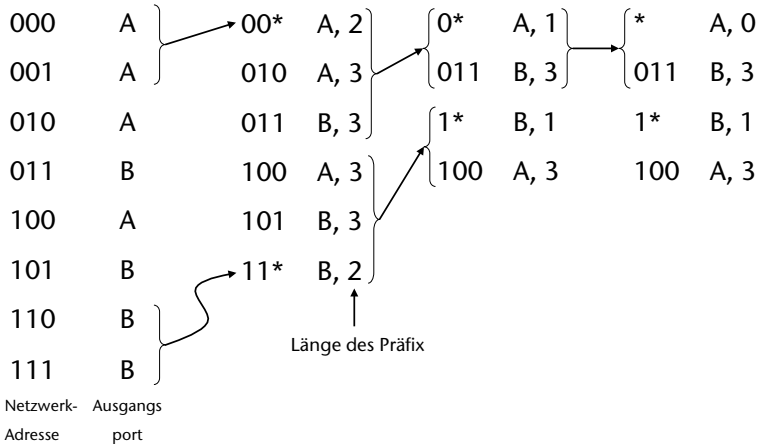
Anforderungen

- Geschwindigkeit:
  - Muß ca. 500,000…15 Mio Pakete pro Sekunde routen/forwarden
  - Update der Routing-Tabelle:
    - Bursty: einige 100 Routes auf einen Schlag hinzufügen/löschen → effiziente Insert/Delete-Operationen
    - Frequenz: im Mittel ca. 100 Updates / Sekunde
- Speicherbedarf:
  - Anzahl der Netzwerkadressen ist groß
  - Hätte man für jede mögliche Netzwerkadresse eine Zeile in der Tabelle, dann bräuchte jeder Router eine große Menge an Speicher
  - Das wiederum hätte Auswirkungen auf die Geschwindigkeit



http://bgp.potaroo.net/

Beispiel einer Routing-Tabelle



Problem

- Gegeben:
  - Menge von *n* Präfixen plus Länge jedes Präfix',
  - Bit-String zum Vergleich
- Ziel: effiziente Algorithmen zur
  - Bestimmung des *Longest Matching Prefix*
  - Einfügen von Präfixen in die Tabelle
  - Löschen von Präfixen in der Tabelle

1. Ansatz: Lineare Suche (= Brute-Force)

- Jeden Eintrag prüfen
- Sich den longest match merken
- Zeitkomplexität Einfügen: O(1)
- Zeitkomplexität Löschen: O(n)
- Average-Case lookup: O(n/2) = O(n)
- Worst-Case lookup: O(n)
- Speicherkomplexität: O(n)

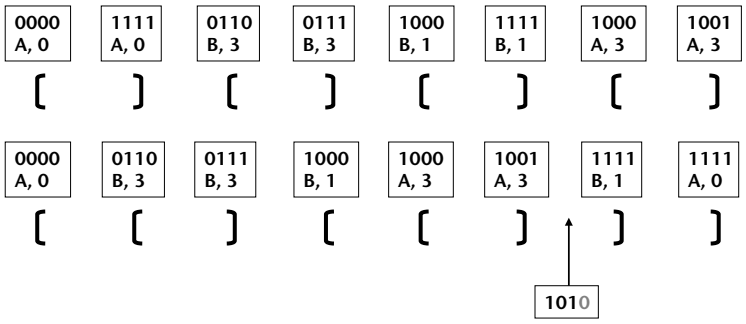
2. Ansatz: Sortierte Bereiche

- Erstelle von jedem Tabelleneintrag zwei "Marker":
  - Jeder Marker ist 1 Bit länger als der längste Präfix
  - Linker Marker ([) = Tabelleneintrag, aufgefüllt mit 0-en
  - Rechter Marker (]) = Tabelleneintrag, aufgefüllt mit 1-en
  - Beide Marker zusammen definieren den Bereich, den der Präfix abdeckt (abzüglich eventueller Intervalle in dessen Innerem, die von längeren Präfixen belegt werden)
  - Assoziiere Präfixlänge und Präfixe mit den Markern
  - Sortiere Marker

Beispiel

*	A, 0
011	B, 3
1*	B, 1
100	A, 3

Marker sind 4 Bits lang



Komplexität

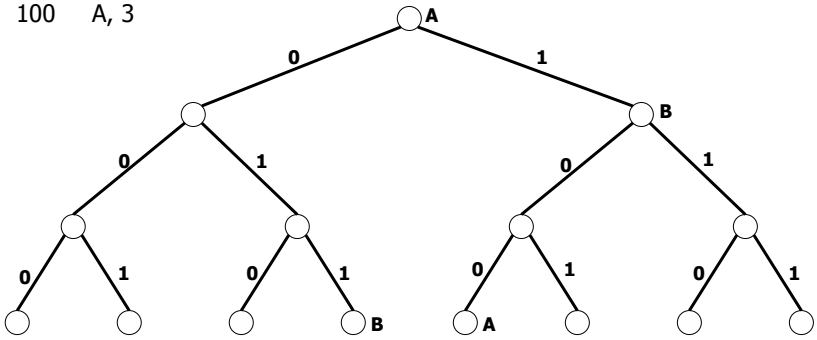
- Zeitkomplexität Einfügen: O(n log(n))
- Zeitkomplexität Löschen: O(n log(n))
- Zeitkomplexität Lookup: O(log(n))
- Speicherkomplexität: O(2n)

3. Ansatz: Lösung mit DST / Trie

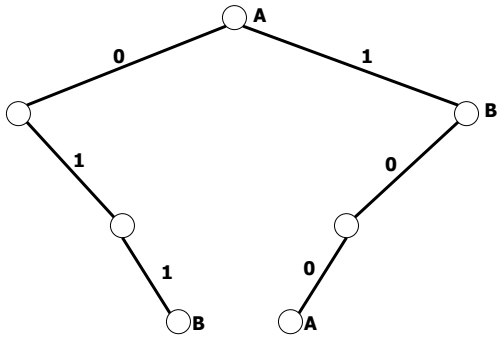
- Wird in aktuellen Routern verwendet
- Erstelle einen Binärbaum
- Jede Stufe des Baumes wird jeweils mit dem nächsten Bit indiziert
- Der Baum wird nur so weit aufgebaut, wie nötig
- Bezeichne jeden Knoten im Baum mit dem Port, der dem Präfix zugeordnet ist

Beispiel

*	A, 0
011	B, 3
1*	B, 1
100	A, 3



*	A, 0
011	B, 3
1*	B, 1
100	A, 3



Komplexität

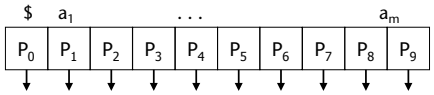
- b = maximale Anzahl Bits der Einträge
- Zeitkomplexität Lookup: O(b)
- Zeitkomplexität Einfügen: O(b)
- Zeitkomplexität Löschen: O(b)

Allgemeine Tries

- Verwaltung von Keys verschiedener Länge:
  - Füge spezielles Zeichen (z.B. $) zum Alphabet hinzu (Terminierungszeichen, Terminator)
  - Füge dieses Zeichen am *Ende* jedes Keys hinzu
  - Effekt: die Menge der Keys wird präfixfrei, d.h., kein Key ist Präfix eines anderen
  - (Das Ganze ist nur ein "Denkhilfsmittel"!)
- Keys werden als Zeichenfolgen eines Alphabets $\$,a_1,\ldots,a_m$ ausgedrückt:
  - Zahlen: m=10+1
  - Buchstaben: m=26+1
  - alpha-numerische Zeichen: m=36+1

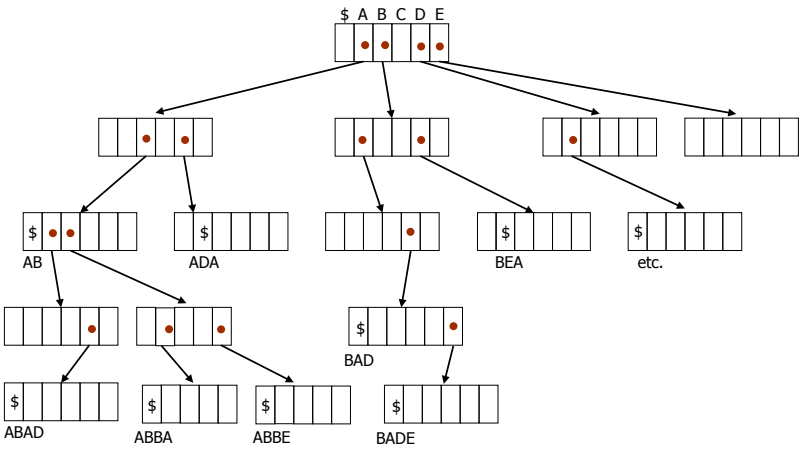
m-stufiger Trie

- Ein Knoten eines Trie über Alphabet der Mächtigkeit *m* ist ein Vektor mit *m*+1 Zeigern:
  - Jedes Vektorelement repräsentiert ein Zeichen des Alphabets
  - Für ein Zeichen a_k an der i-ten Stelle in einem Key gibt es einen Zeiger an der Stelle k in einem Vektor auf der i-ten Stufe des Baumes
  - Dieser Zeiger
    - zeigt auf einen Unterbaum für die "normalen" Zeichen ($P_1 \ldots P_m$), oder
    - es ist nur ein, von NULL verschiedener, Dummy-Wert (Fall P_0; zeigt also an, daß es einen Key gibt, der hier endet)



Beispiel

- Ein Trie für die Schlüssel AB, ABAD, ABBA, ADA, BAD, BADE, DA, DADA, EDA, EDDA, EDE über dem Alphabet {A,…,E} und $ als Terminator



Bemerkungen

- Grundlegende Struktur eines Tries (mit fester Schlüssellänge) ist einem B-Baum ähnlich:

m-Wege-Baum	m-way Trie
Key-Menge wird durch Separatoren in die Unterbäume aufgeteilt	Key-Menge wird durch "Selektoren" in die Unterbäume aufgeteilt
Blätter zeigen auf Nutzdaten	Knoten mit gesetztem Terminator zeigen auf Daten

- Belegung der Knoten nimmt zu den Blättern hin ab (schlechte Speicherausnutzung)
- Analog gibt es auch einen m-Wege-DST

Suchen im Multi-Way Trie

- Schema: verfolge den für das aktuelle Zeichen im Key "zuständigen" Zeiger

```
k = Input-Key
Index i = 0
starte bei x = Wurzel
while i < Länge(k):
    teste Zeiger x.sel[ k[i] ]
    if Zeiger == None:
        Key k ist nicht im Trie
    else:
        x = x.sel[ k[i] ]
    i += 1
teste, ob Flag in x["$"] gesetzt
```

- Anzahl der durchsuchten Knoten = Länge des Schlüssels + 1
- Vorteil: Such-Komplexität ist unabhängig von der Anzahl der gespeicherten Schlüssel

Einfügen in einen Trie

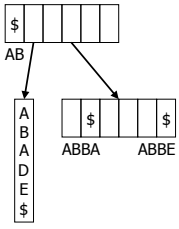
- Analog zum Suchen: verfolge Zeiger
- Falls Key zu Ende: teste Terminator-Flag ($-Feld)
  - Falls schon gesetzt → Key war schon im Trie
  - Sonst: setzen, Daten dort speichern
- Falls Zeiger nicht vorhanden (Key noch nicht zu Ende): erzeuge neue Knoten
- Spezialbehandlung, falls man die reinen "Terminierungsknoten" eingespart hat

Löschen aus einem Trie

- Baum durchsuchen, bis der Terminator ($) gefunden ist
- Terminator $ löschen
- alle Zeiger des Knotens überprüfen, ob sie alle auf NULL sind
  - nein → Lösch-Operation ist beendet
  - ja (alle sind NULL) → lösche den Knoten und überprüfe den Vaterknoten, falls der jetzt auch leer, dann wiederhole

Bemerkungen

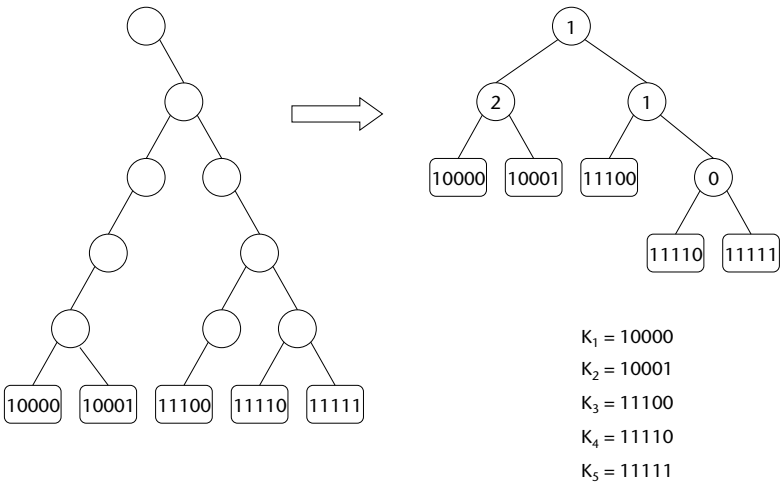
- Die Struktur eines Trie's hängt nur von den vorliegenden Keys ab, nicht von der Reihenfolge der Einfügungen!
- Keine optimale Speichernutzung, weil die Knoten eine feste Länge haben, auch bei minimaler Belegung
- Häufig ist One-Way-Branching (z.B. BEA und ABADE)
  - Lösung: zeigt ein Zeiger auf einen Unterbaum, der nur einen Key enthält, wird der Key im Knoten gespeichert



PATRICIA Trees

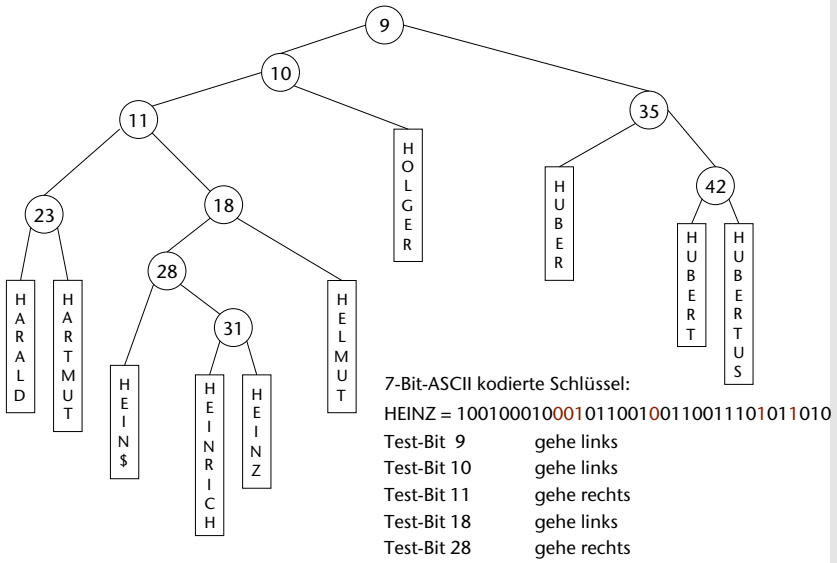
- Sind eine Variante der binären Tries:
  - Vermeidet Pfade im Baum ohne Gabelung ("Einweg-Pfade")
  - Hat nur einen Typ Knoten
- PATRICIA = "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [Morrison, 1968]
- Ideen:
  1. Speichere an inneren Knoten die Anzahl Bits, die übersprungen werden können und nicht getestet werden brauchen (weil es sowieso keine Gabelung auf dem Pfad darunter gibt, d.h., alle Keys in diesem Teilbaum bis dahin den gleichen Präfix haben)
  2. Speichere Keys/Daten an ("irgendeinem") inneren Knoten

Einweg-Pfade vermeiden



$K_1 = 10000$
$K_2 = 10001$
$K_3 = 11100$
$K_4 = 11110$
$K_5 = 11111$

Beispiel zur Suche in PATRICIA



7-Bit-ASCII kodierte Schlüssel:

HEINZ = 1001000100010110010011001110101101 0

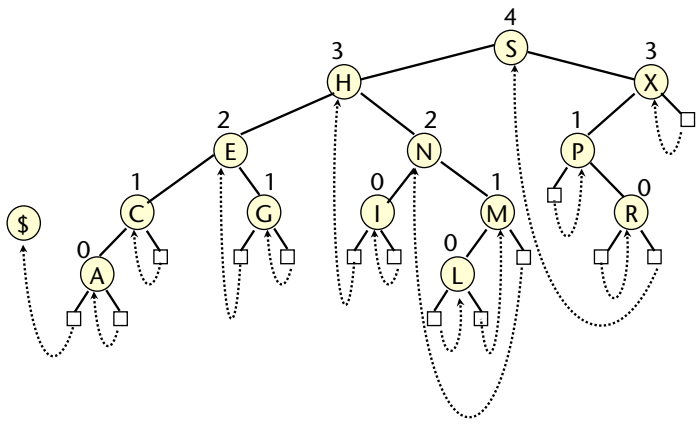
Test-Bit 9	gehe links
Test-Bit 10	gehe links
Test-Bit 11	gehe rechts
Test-Bit 18	gehe links
Test-Bit 28	gehe rechts
Test-Bit 31	gehe rechts, Blatt, vergleiche, OK

Eine praktischere Variante der PATRICIA Trees

- Nummeriere Bits der Keys von rechts nach links (0 = rechtes Bit)
- Speichere im Knoten die Nummer desjenigen Bits, das an diesem Knoten getestet werden muß
- Daher manchmal auch der Name "*crit bit tree*" für "*critical bit tree*"
- Konsequenz: auf jedem Pfad durch den Baum von oben nach unten nehmen diese Nummern ab

- Lösung für das Problem der zwei verschiedenen Knoten-Arten:
  - Speichere Keys (bislang in Blättern) in ("irgend einem") inneren Knoten
  - Geht gut, weil #Blätter = #innere Knoten + 1
    - Man braucht nur einen "Extra-Knoten"
- Knoten haben jetzt 2 verschiedene Funktionen zu verschiedenen Zeiten:
  - Knoten = Selektor während der Traversierung
  - Knoten = Daten-Container am Ende der Traversierung
- Es gibt keine "richtigen" Blätter mehr

Beispiel



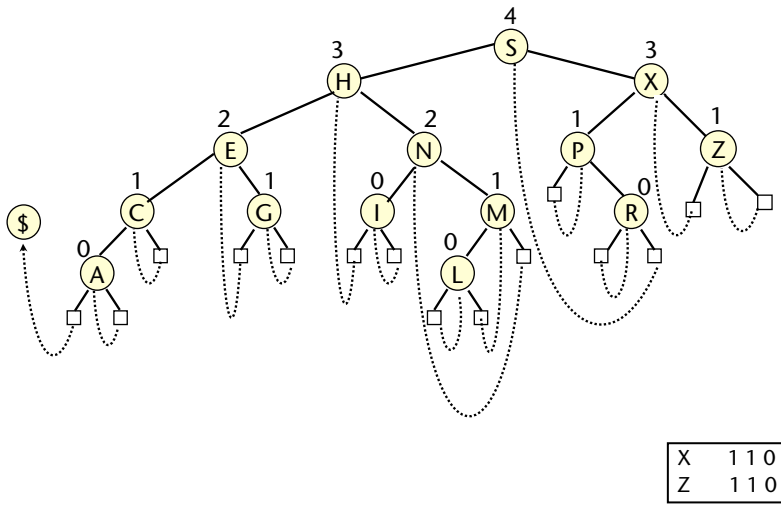
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

Die Suche in PATRICIAs

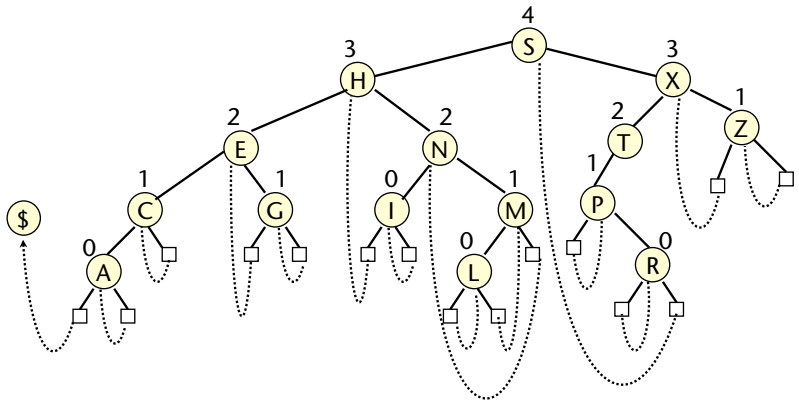
- Laufe im Baum abwärts, wobei man den Bitindex in jedem Knoten benutzt, um festzustellen, welches Bit im Key zu testen ist
- Die Keys in den Knoten werden auf dem Weg im Baum abwärts überhaupt nicht beachtet!
- Schließlich wird ein aufwärts zeigender Pointer auf einen inneren Knoten vorgefunden → führe vollständigen Vergleich zwischen gesuchtem Key und Key in jenem inneren Knoten durch.
- Es ist leicht zu testen, ob ein Zeiger nach oben zeigt, da die Bitindizes in den Knoten (per Definition) kleiner werden, wenn man sich im Baum abwärts bewegt.

Einfügen in PATRICIAs

- 2 Fälle
- Zunächst: äußeres Einfügen in einen Patricia-Baum
- Beispiel: Z = 11010 einfügen:
  - rechten Zeiger von X (der ja nach oben zeigt) ersetzen durch einen Zeiger, der auf einen (neuen) "Testknoten" zeigt;
  - neuen Testknoten erzeugen, der X und Z unterscheiden kann (Bit 1 testen, da X = 11000);
  - von dort aus linken Zeiger nach X hoch, rechten Zeiger auf Z hoch



- Zweiter Fall (etwas komplizierter): Inneres Einfügen
  - Wenn der Key auf dem Weg im Inneren eingefügt werden muß
  - D.h., das Bit, das diesen Key von den anderen unterscheidet, wurde bei der Suche übersprungen
- Beispiel: T = 10100 einfügen
  - Suche endet bei P = 10000.
  - T und P unterscheiden sich in Bit 2, einer Position, die während der Suche übersprungen wurde. Die Forderung, daß die Bitindizes fallen müssen, wenn man sich im Baum abwärts bewegt, macht es notwendig, T zwischen X und P einzufügen, mit einem nach oben auf T selbst gerichteten Zeiger, der seinem eigenen Bit 2 entspricht.
  - Beachte: die Tatsache, daß Bit 2 vor dem Einfügen von T übersprungen wurde, impliziert, daß P und R den gleichen Wert von Bit 2 besitzen.



Eigenschaften

- Sedgewick: "Patricia stellt die Quintessenz der digitalen Suchmethoden dar"
- Knuth: "Patricia is a little tricky, and she requires careful scrutiny before all of her beauties are revealed."

- Haupttrick: Patricia identifiziert diejenigen Bits, die die Suchschlüssel von anderen unterscheiden, und baut sie in eine Datenstruktur (ohne überflüssige Knoten) ein, so daß man schnell von einem beliebigen Suchschlüssel zu dem einzigen Schlüssel in der Datenstruktur kommt, der gleich sein könnte.

- Satz:
  Ein Patricia-Trie, der aus *N* zufälligen Keys mit *b* Bits erzeugt wurde, hat *N* Knoten und erfordert für eine durchschnittliche Suche log(*N*) Bitvergleiche.
- Bemerkung:
  - Die Bit-Länge der Schlüssel spielt keine Rolle!
  - Die o.g. Komplexität ist eine Bitkomplexität.
  - Bei allen anderen Suchmethoden ist die Länge der Keys in irgendeiner Weise in die Suchprozedur "eingebaut" (z.B. beim Vergleich von kompletten Keys)