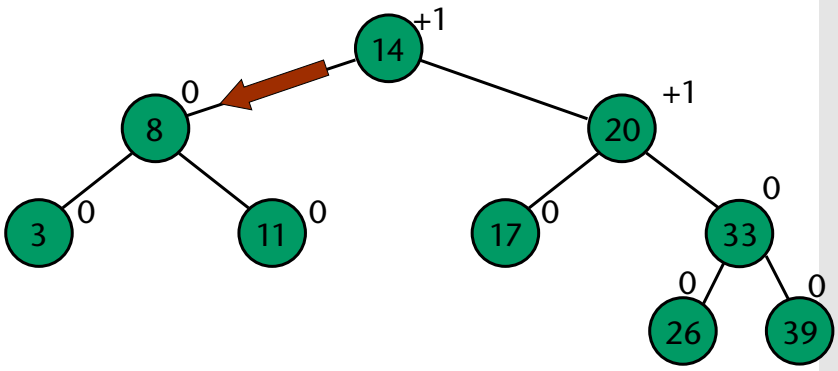
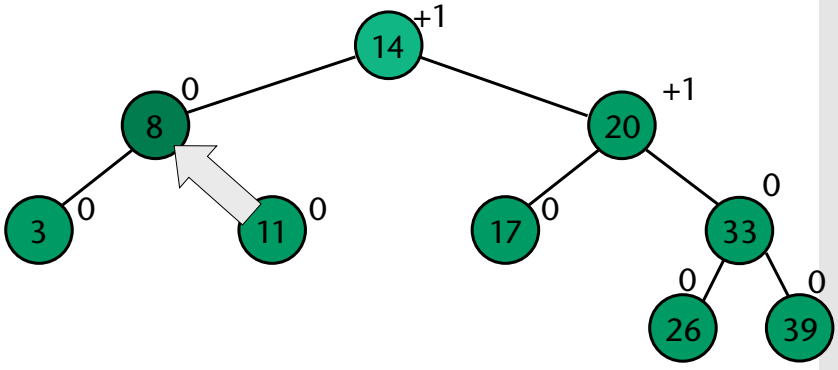
Löschen von Knoten

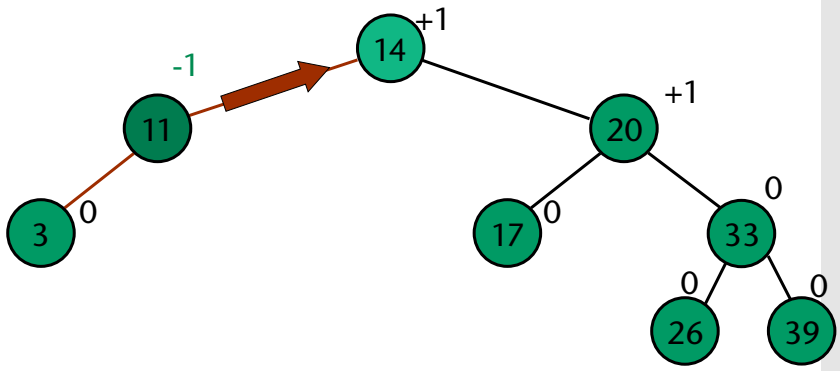
Beispiel: Löschen von $k = 8$



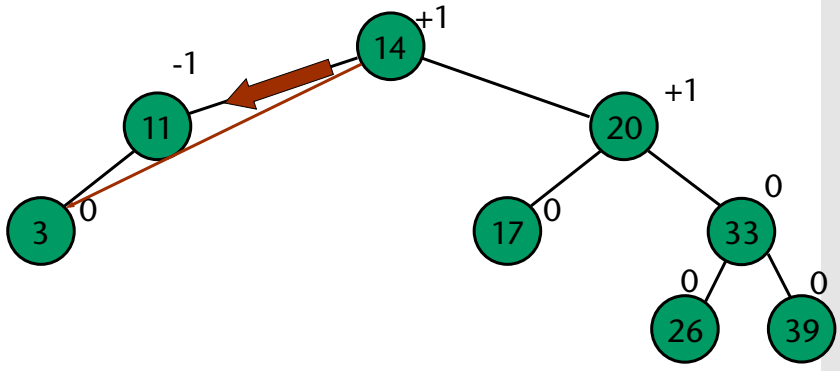
Beispiel: Löschen von $k = 8$



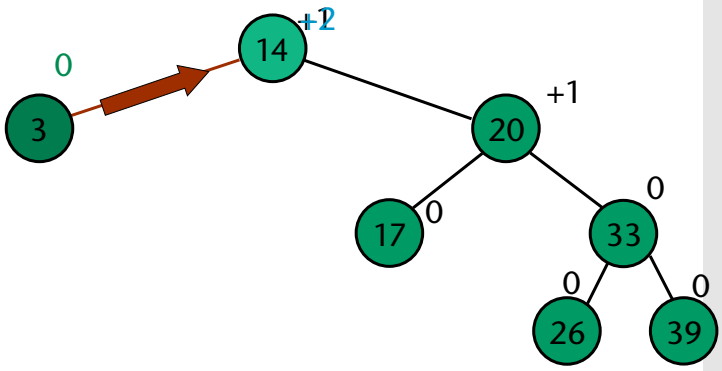
Beispiel: Löschen von $k = 8$



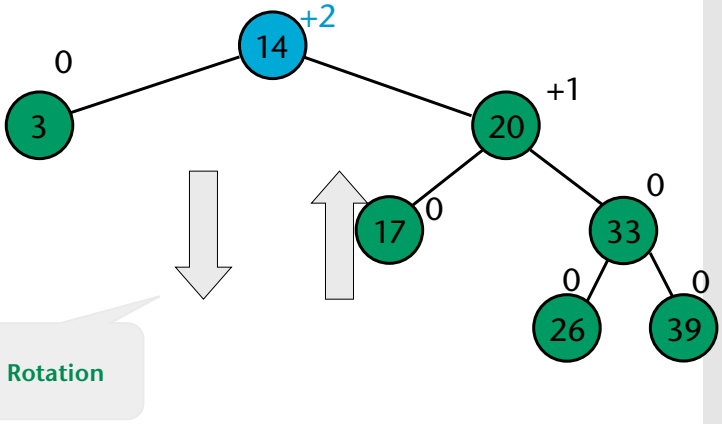
Beispiel: Löschen von $k = 11$



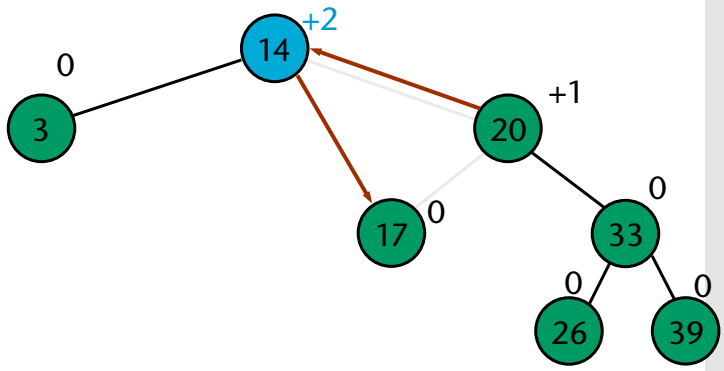
Beispiel: Löschen von $k = 11$



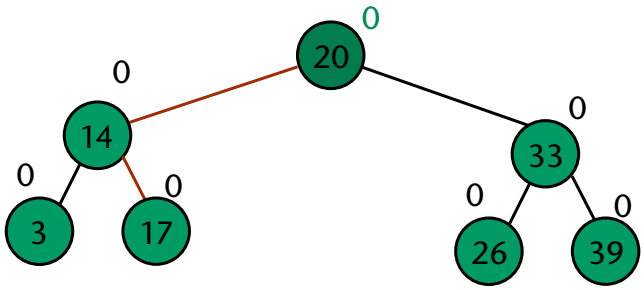
Beispiel: Löschen von $k = 11$



Beispiel: Löschen von $k = 11$



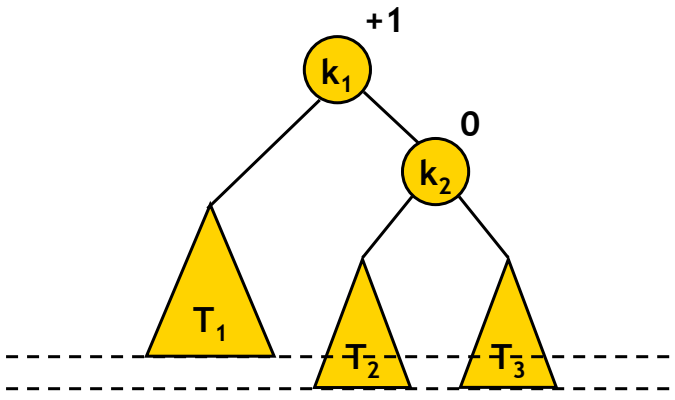
Beispiel: Löschen von $k = 11$

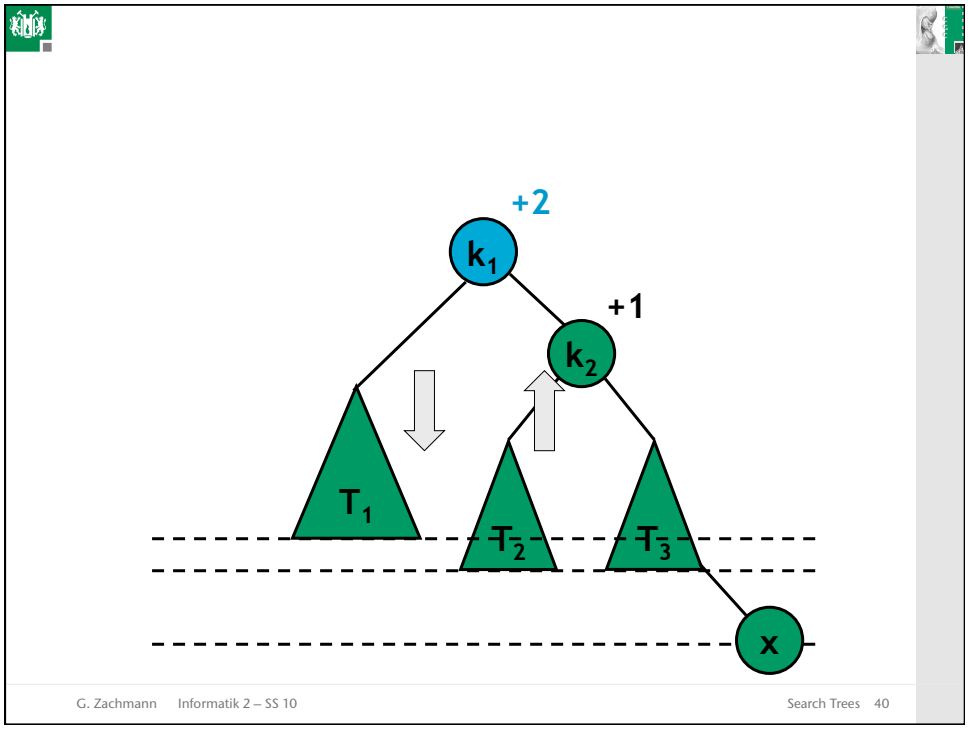


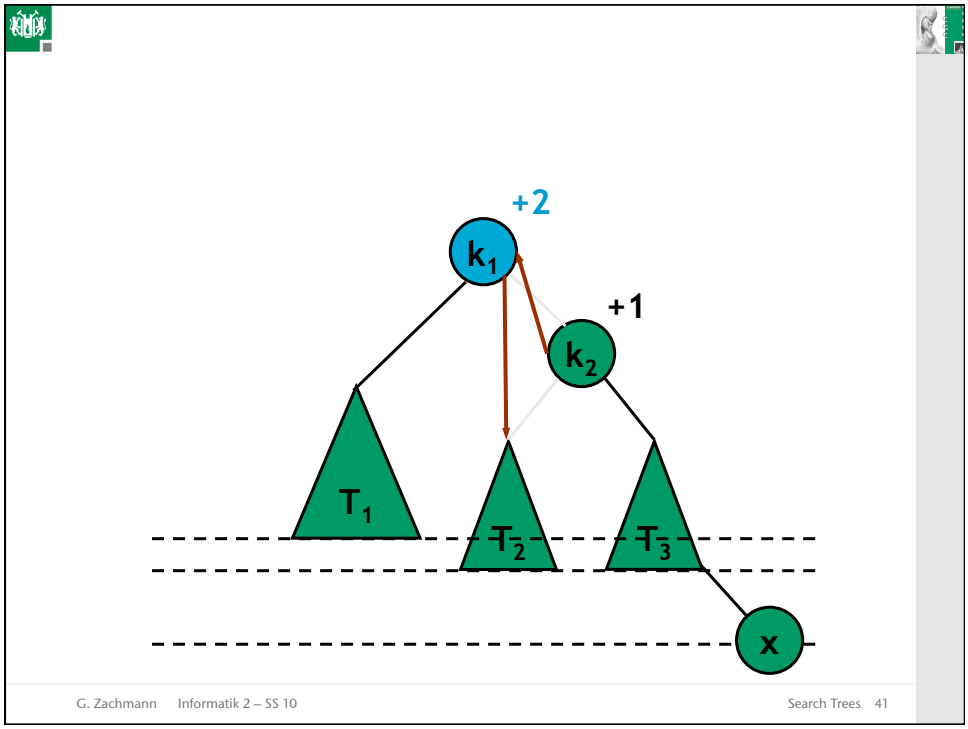
AVL-Rotationen

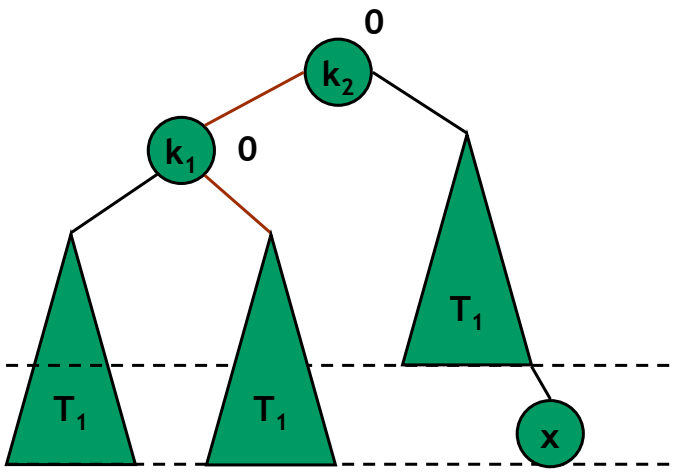
- Operationen auf AVL-Bäumen zur Erhaltung der AVL-Eigenschaft
- Bestehen ausschließlich aus “Umhängen” von Zeigern
- Es gibt 2 verschiedene Arten von Rotationen
  - Double Rotation:
    - RL = neuer Knoten ist im linken Unterbaum des rechten Unterbaumes
    - M.a.W.: vom Knoten mit dem "schlechten" 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



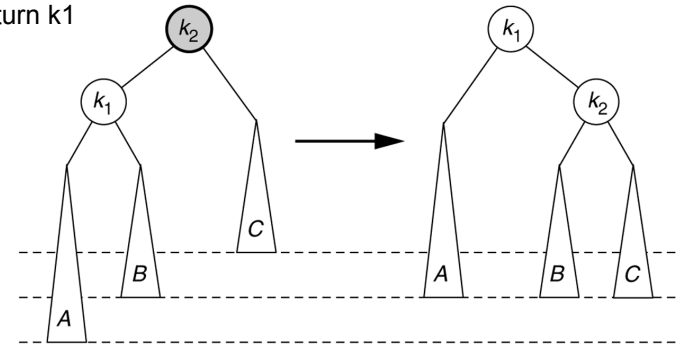

+2
k1
+1
k2
T1
T2
T3
x


+2
k1
+1
k2
T1
T2
T3
x



LL Rotation Algorithm

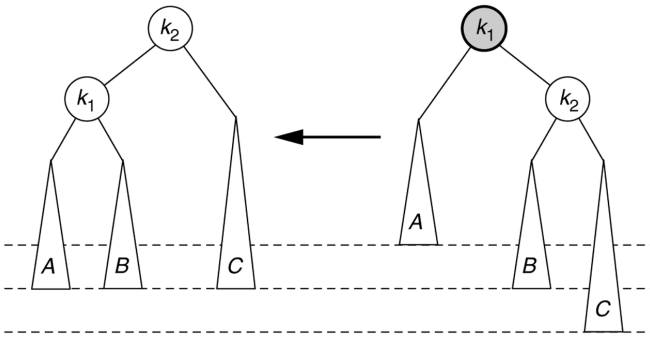
- def LL_Rotate (k2):
  k1 = k2.left
  k2.left = k1.right
  k1.right = k2
  return k1



(a) Before rotation (b) After rotation

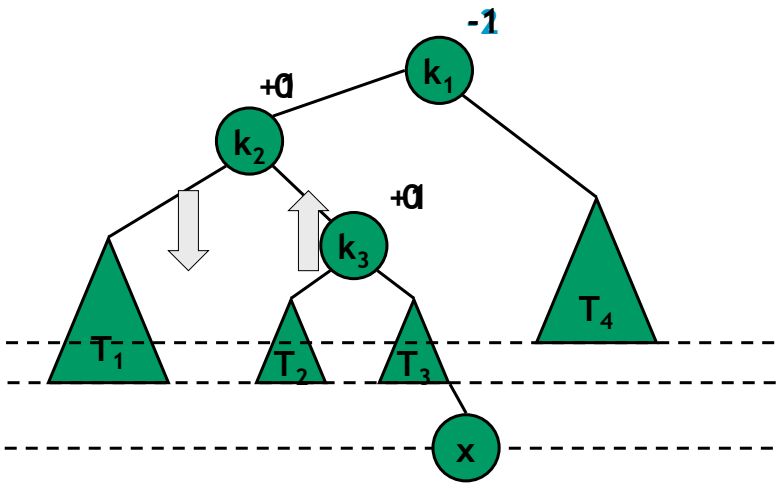
RR Rotation Algorithm

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

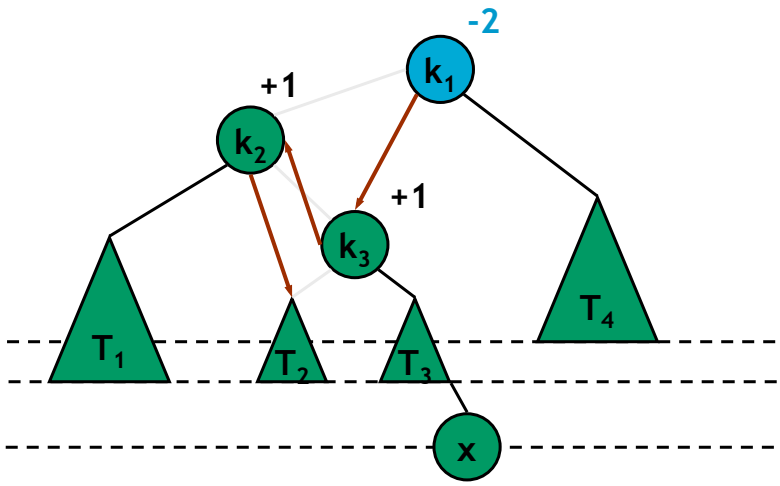


(a) After rotation (b) Before rotation

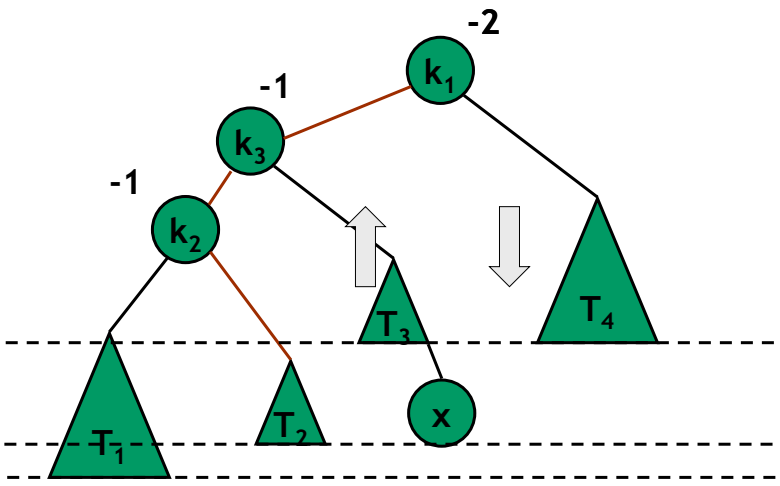
LR-Rotation



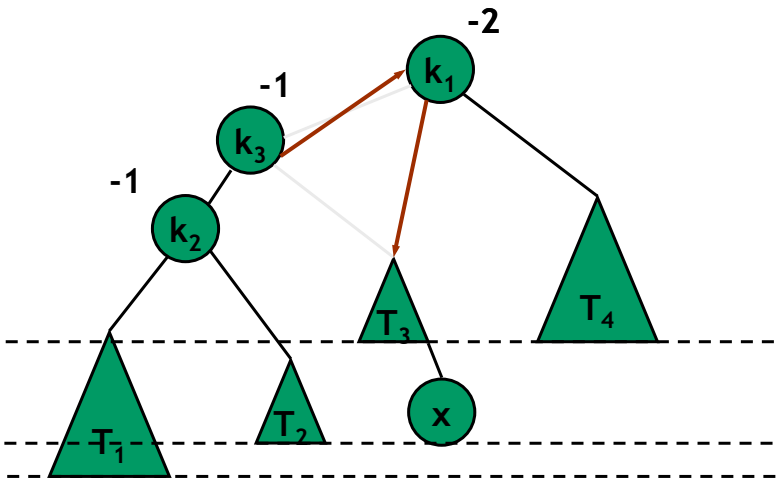
LR-Rotation



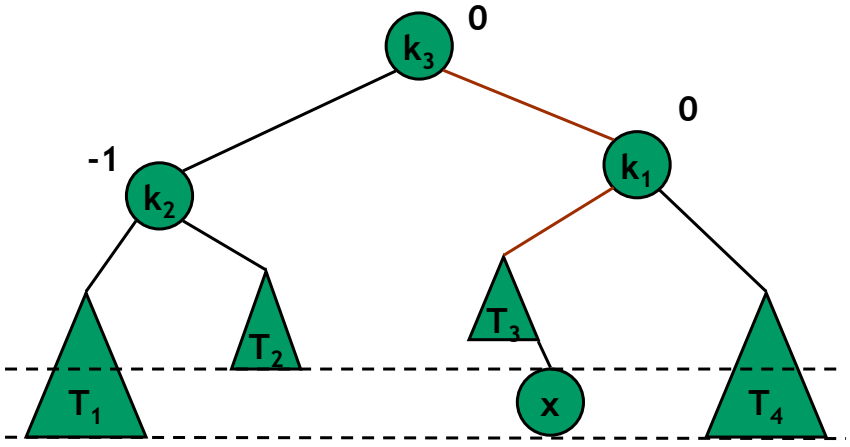
LR-Rotation



LR-Rotation

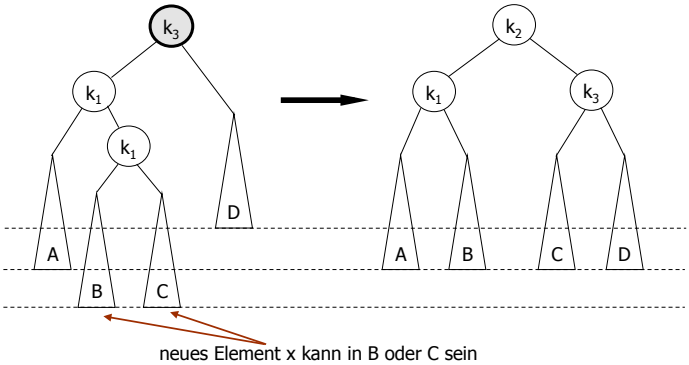


LR-Rotation



Code

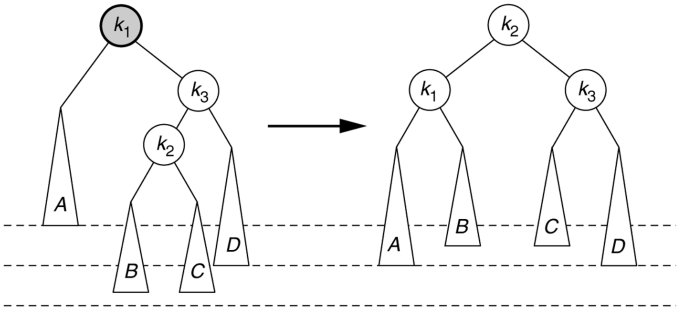
```
def LR_doubleRotate( k3 ):
  k3.left = RR_Rotate( k3.left )
  return LL_Rotate( k3 )
```



neues Element x kann in B oder C sein

(a) Before rotation (b) After rotation

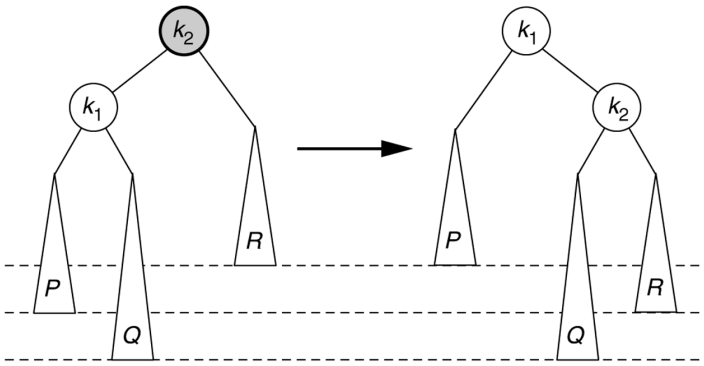

```
def RL_doubleRotate( k1 ):
  k1.right = LL_Rotate( k1.right )
return RR_Rotate( k1 )
```



(a) Before rotation (b) After rotation

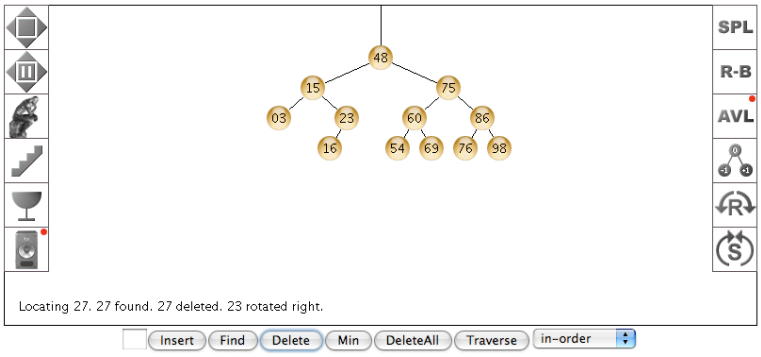
Warum Double Rotation?

- 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

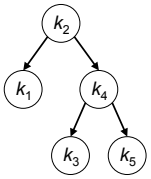
Problemstellung

- Gegeben sei Folge $k_1 < k_2 < ... < k_n$ von n sortieren 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 $d_T(\textit{Wurzel}) = 0$
- Also: erwartete (= mittlere) Kosten für die Suche (eines Keys) ist

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

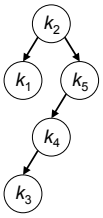
Beispiel

- Gegeben seien 5 Keys mit den Suchwahrscheinlichkeiten:
  $p_1 = 0.25,\ p_2 = 0.2,\ p_3 = 0.05,\ p_4 = 0.2,\ p_5 = 0.3$



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 Baum für diese Key-Menge)

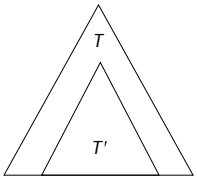
- Beobachtungen:
  - Der optimale BST muß nicht die kleinste Höhe haben
  - Der optimale BST muß nicht die größte Wahrscheinlichkeit an der Wurzel haben
- Erstellen durch erschöpfendes Testen? (*exhaustive enumeration*)
  - Erstelle alle möglichen BSTs mit n Knoten
  - Für jeden BST: verteile die Schlüssel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

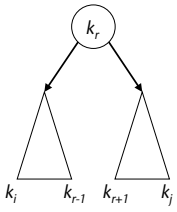
Die optimale Unterstruktur

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!

1. 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 $k_{r+1}, ..., k_j$



2. 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

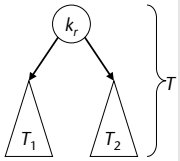
$$\sum_{l=i}^{j} p_l \neq 1$$

  - 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 $1 \le i \le n$
     - $w[i,j] = w[i,j-1] + p_j = p_i + w[i+1,j]$ für $1 \le i \le j \le n$

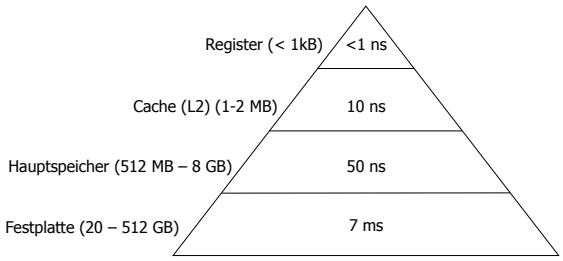

```
def optimal_bst( p,q,n ):
    for i in range( 1,n ):
        e[i,i] = w[i,i] = p[i]
    for l in range( 2,n ):# calc all opt trees w/ l keys
        for i in range( 1, n-l ):          # n - ell
            j = i+l-1
            e[i,j] = ∝                     # z.B. 2^31-1
            w[i,j] = w[i,j-1]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

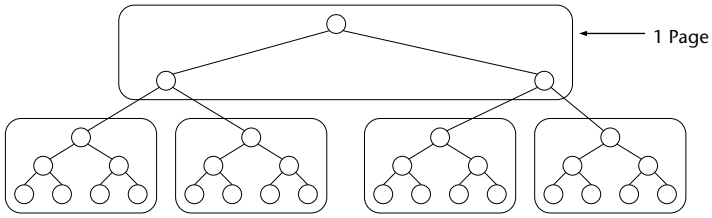
- D.h., Zugriff auf (externe) Knoten ist seeeehr teuer
- Bsp. :
  - File-Tree des File-Systems
  - Datenbanken (DB2 (IBM), Sybase, Oracle, …)
- Man hätte gerne einen Baum, der noch geringere Tiefe hat als

$$\log_2(n)$$

- 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

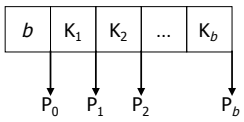
- Ein m-Wege-Suchbaum ist eine Verallgemeinerung eines binären Suchbaumes (d. h., ein binärer Suchbaum ist ein 2-Wege-Suchbaum)



Anstatt 5 nur noch 2 Zugriffe auf die Platte / CD

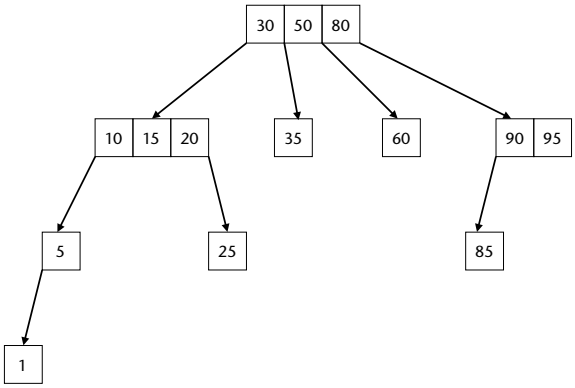
Definition

- 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:
    - K_i sind die Schlüssel, $1 \le i \le b$
    - *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:
    - $K_1 \le K_2 \le \ldots \le K_b$
  - Alle Schlüssel im Unterbaum P_i sind kleiner als K_{i+1} , für $0 \le i < b$
  - Alle Schlüssel im Unterbaum P_b sind größer als K_b
  - Die Unterbäume P_i , für $0 \le i \le b$, sind ebenfalls *m*-Wege-Bäume

Beispiel



Bemerkungen

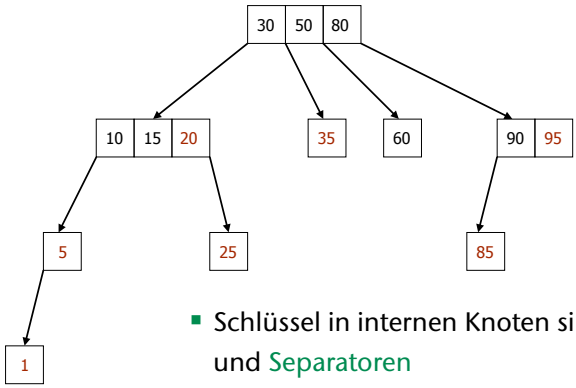
- Probleme:
  - Der Baum ist nicht ausgeglichen (balanciert)
  - Die Blätter sind auf verschiedenen Stufen
  - Bei Veränderungen gibt es keinen Ausgleichsalgorithmus (à la AVL)
  - Kann also zu verketteten Listen degenerieren
- Anzahl der Knoten im vollständigen *m*-Wege Baum mit Höhe *h*:

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

- 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

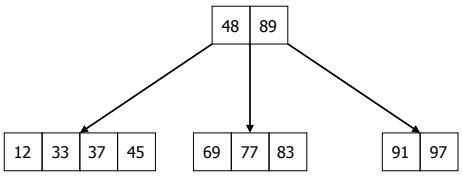
- Einfügen von 20, 35, 5, 95, 1, 25, 85



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

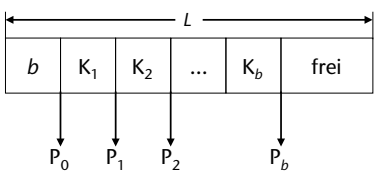
B-Bäume

- 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

- Jeder Weg von der Wurzel zu einem Blatt ist gleich lang
- Jeder Knoten (außer Wurzel) enthält mindestens k und höchstens 2k Schlüssel
- Jeder Knoten (außer der Wurzel) hat zwischen k+1 und 2k+1 Kinder (Unterbäume)
- Die Wurzel ist entweder ein Blatt oder hat mindesten 2 Kinder
- Jeder Knoten hat die Struktur:
  - b ist die Anzahl der Keys im Knoten, k ≤ b ≤ 2k
  - Die Keys in Knoten sind aufsteigend sortiert ($K_1 < K_2 < \ldots < K_b$)
  - (L = Größe einer Seite)



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 2*t*-1 Schlüssel, also höchsten 2*t* Kinder
  - Formeln müssen „angepasst“ werden, damit sie stimmen
  - die Prüfung läuft gemäß den Folien, bei Unsicherheiten die Definition angeben