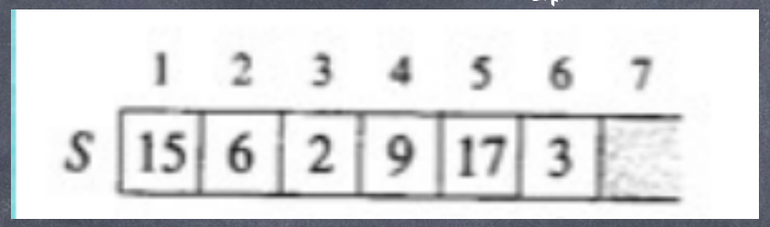
Container-Datenstrukturen

Große Übung 4

Aufgabenstellung

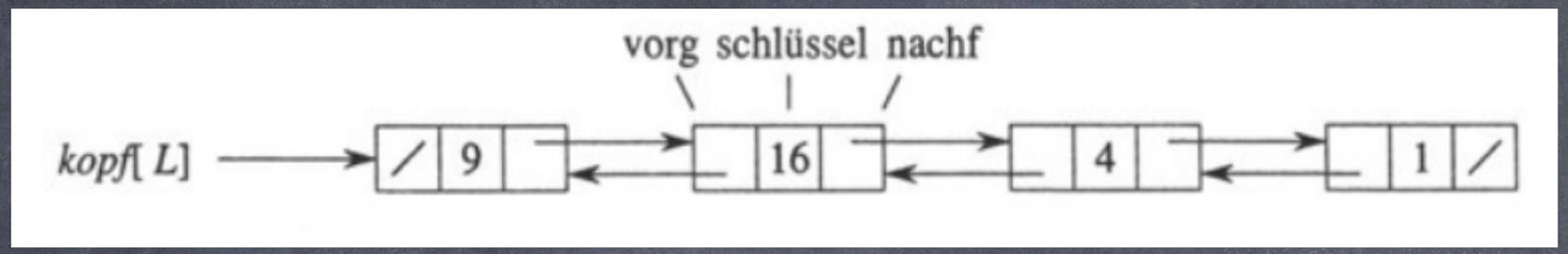
- Verwalte Kollektion S von n Objekten
- Grundaufgaben:
  - Iterieren/Auflistung
  - Suche nach Objekt x mit Wert/Schlüssel k
  - Füge ein Objekt x hinzu
  - Entferne ein Objekt x
  - Situationsabhängig weitere Operationen

Arrays



- Speicherblöcke, in denen die Elemente „nebeneinander" liegen
- Sehr schnell beim Auflisten und beim Zugriff auf das i-te Element S[i]
- <u>Unsortiert:</u>
  - Suche: Alle Elemente durchgehen
  - Löschen: Elemente rechts von x um 1 Position nach links schieben
  - Einfügen: In freien Platz „am Ende" einfügen; falls dieser voll ist: Neues Array anlegen und alle Elemente kopieren
- <u>Sortiert:</u>
  - Suche: Binäre Suche
  - Einfügen: Position suchen und einfügen, Elemente nach rechts schieben
  - Löschen wie bei unsortiertem Array
- Einfügen/Löschen verschiebt Elemente; Zeiger/Adressen bleiben nicht erhalten!

Verkettete Listen



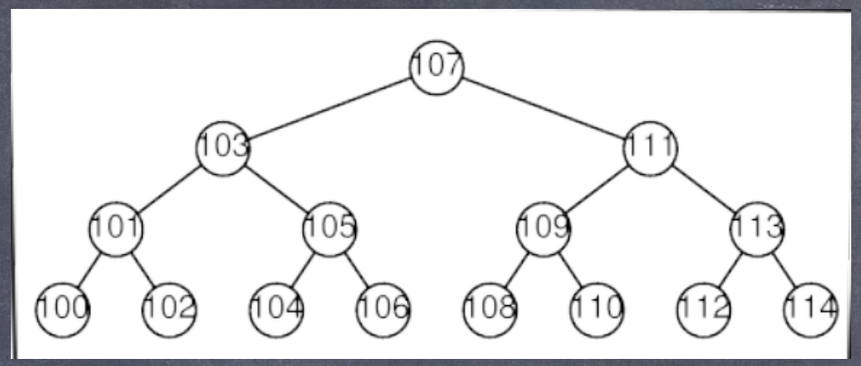
- Einfache Datenstruktur für Listen, wenn oft eingefügt oder gelöscht werden muss
- Gespeichert als Elemente, jeweils mit Zeiger auf Vorgänger und Nachfolger
- Suche: Alle Elemente durchgehen
- Einfügen/Löschen: Kein Verschieben nötig
- Zeiger/Adressen bleiben dabei erhalten!

Achtung: In der Praxis meist schlecht!

Warteschlange, Stapel

- Haben zusätzlich kopf(Q), ende(Q), top(S)
- Warteschlange: Enqueue, Dequeue (LIFO)
- Stapel: Push, Pop (FIFO)
- Lassen sich auf Arrays und verketteten Listen implementieren
- Operationen (amortisiert) in O(1)

Binäre Suchbäume



- Struktur der Daten wie bei binärer Suche
- Ziel: Suche, Einfügen, Löschen in O(log n)
- Speichere Daten in gerichtetem Baum (1 Element/Knoten)
- Jeder Knoten hat 0–2 Kinder
- Jeder Knoten bis auf die Wurzel hat genau einen Vaterknoten
- Knoten ohne Kinder: Blätter
- An jedem Knoten mit Wert k:
  - Im linken Teilbaum: Schlüssel k' < k
  - Im rechten Teilbaum: Schlüssel k' >= k

Warum gerade 2?

- Siehe Blatt 3: Logarithmen zu verschiedenen Basen > 1 unterscheiden sich nur um konstanten Faktor
- Das rechtfertigt die Schreibweise O(log n) ohne Angabe der Basis!
- Mit Basis 2 muss die gesamte Kollektion in 2 Teile gespalten werden
- Für größere Basen wird das komplizierter, meist ohne entsprechend großen Nutzen
- In der Vorlesung werden wir noch (kurz) auf größere Basen eingehen
- Beispiel für ternäre (Basis 3) Suche: Wiegerätsel – gegeben k Kugeln, finde die eine, die etwas leichter ist als die anderen; jedes Wiegen teilt die Kugeln in 3 Gruppen

Binäre Bäume - Suche

- Gegeben Schlüssel k, finde x mit diesem Schlüssel (oder NIL, falls so ein x nicht existiert)
- Beginne bei der Wurzel
- Am aktuellen Knoten mit Schlüssel k':
  - Ist aktueller Knoten NIL: Element existiert nicht
  - Ist k < k': Fahre beim linken Kind fort
  - Ist k > k': Fahre beim rechten Kind fort
  - Ist k = k': Element gefunden
- Genauso auch bei AVL-Bäumen!
- Laufzeit proportional zur Höhe

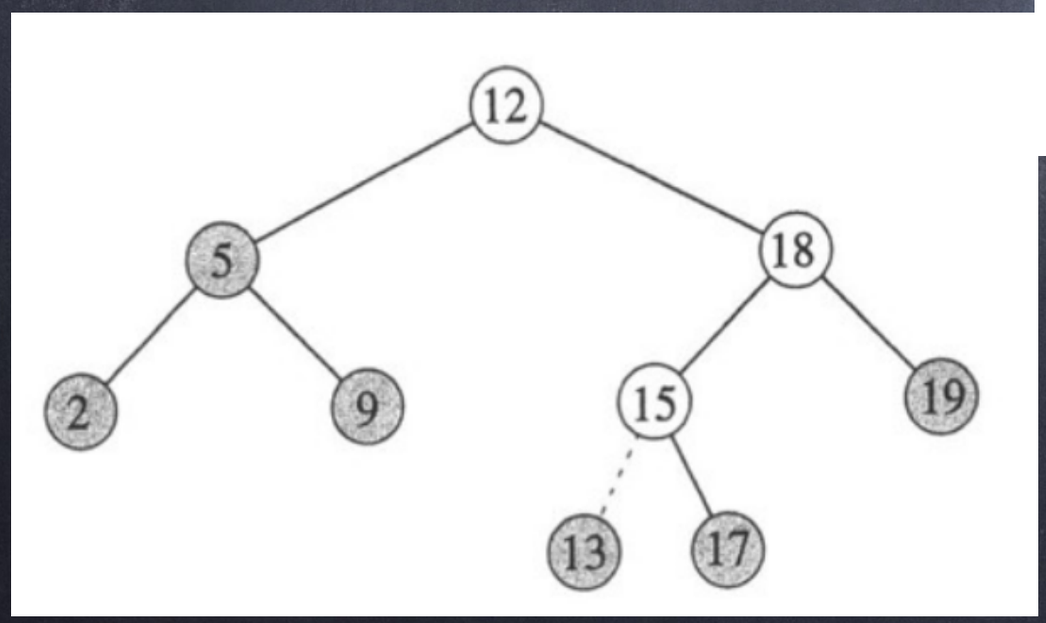
Binäre Bäume – Einfügen

Wie bei Suche (bis NIL)

Setze neuen Vater

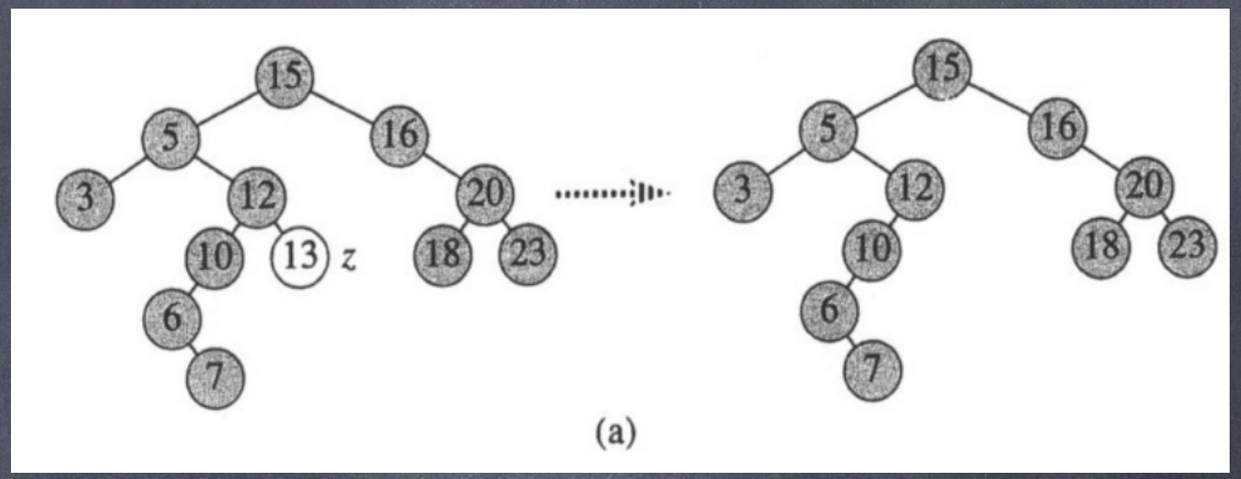
```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← wurzel[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if schlüssel[z] < schlüssel[x]
 6              then x ← links[x]
 7              else x ← rechts[x]
 8  p[z] ← y
 9  if y = NIL
10     then wurzel[T] ← z                    ▷ Baum T war leer
11     else if schlüssel[z] < schlüssel[y]
12             then links[y] ← z
13             else rechts[y] ← z
```

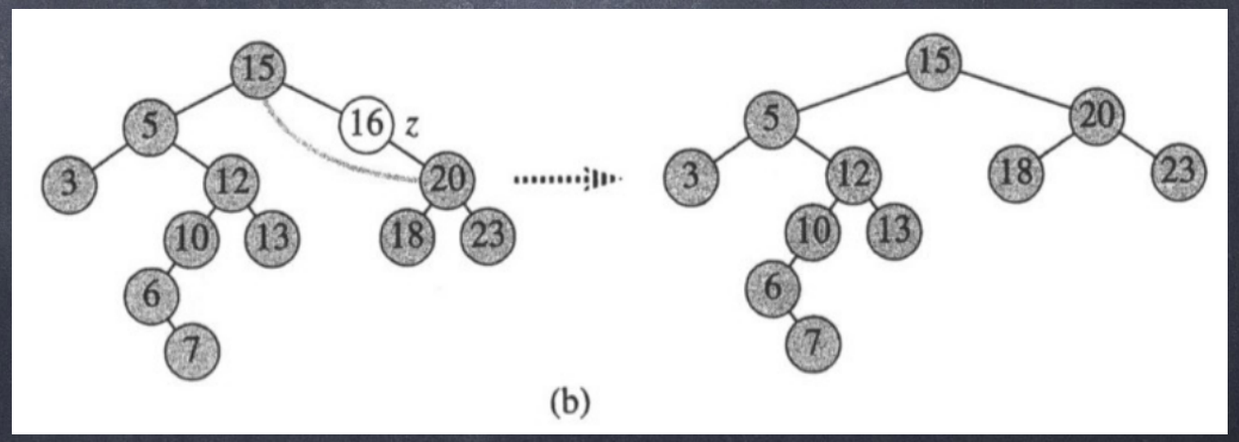
Füge Knoten an Blatt y an



Binäre Suchbäume – Löschen

- Entferne gegebenes Element z aus dem Baum
- Drei Fälle: z hat...
  - a) kein Kind
  - b) ein Kind
  - c) zwei Kinder
- a) und b) sind sehr einfach:
  - a) Knoten z entfernen
  - b) Wie bei verketteter Liste:
    - „Überbrücke“ z
  - c) ist komplizierter!





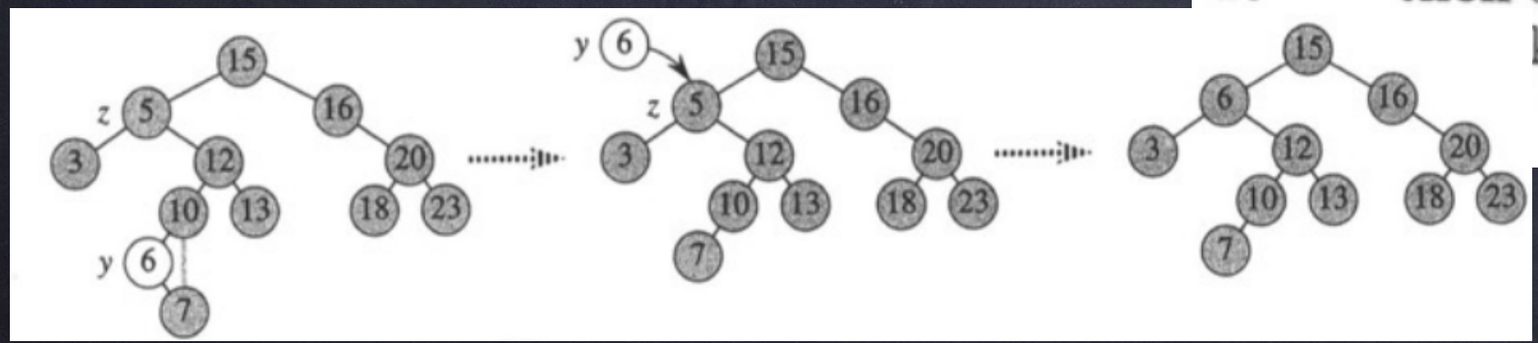
Binäre Suchbäume – Löschen

Knoten mit 2 Kindern: Suche „Ersatzknoten" y mit höchstens 1 Kind

Lösche den Ersatzknoten y

Kopiere den Inhalt von y nach z

```
TREE-DELETE(T, z)
if links[z] = NIL oder rechts[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if links[y] ≠ NIL
    then x ← links[y]
    else x ← rechts[y]
if x ≠ NIL
    then p[x] ← p[y]
if p[y] = NIL
    then wurzel[T] ← x
    else if y = links[p[y]]
            then links[p[y]] ← x
            else rechts[p[y]] ← x
if y ≠ z
    then schlüssel[z] ← schlüssel[y]
         kopiere die Satellitendaten von y in z
```



Binäre Bäume – Laufzeit

- Alle Operationen in O(h)
- h: Höhe des Suchbaums
- Also: Anzahl Knoten auf dem längsten Weg von der Wurzel zu einem Blatt
- Erreichen wir unser Ziel log(n)?

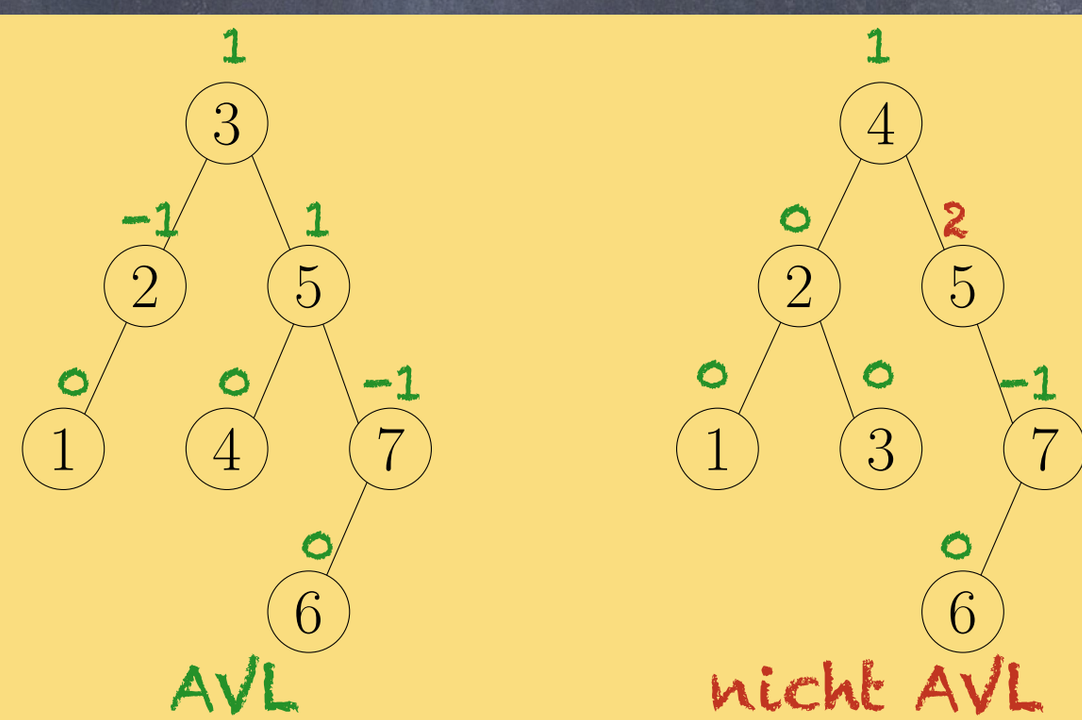
Binäre Suche – Laufzeit

- Nicht unbedingt: Einfügen 1,2,3,4,5,...
- Im Wesentlichen verkettete Liste
- Operationen brauchen O(n)!
- Wir sind also noch nicht am Ziel – Bäume können entarten, d.h. unbalanciert werden.

AVL-Bäume

- AVL-Bäume erfüllen ein bestimmtes Balance-Kriterium
- Dadurch bleibt die Höhe logarithmisch in n
- In einem AVL-Baum gilt für jeden Knoten:
  - Die Höhe seines rechten Unterbaums weicht von der Höhe seines linken Unterbaums höchstens um 1 ab
  - D.h. Höhe rechts - Höhe links ist -1, 0 oder 1

Beispiel

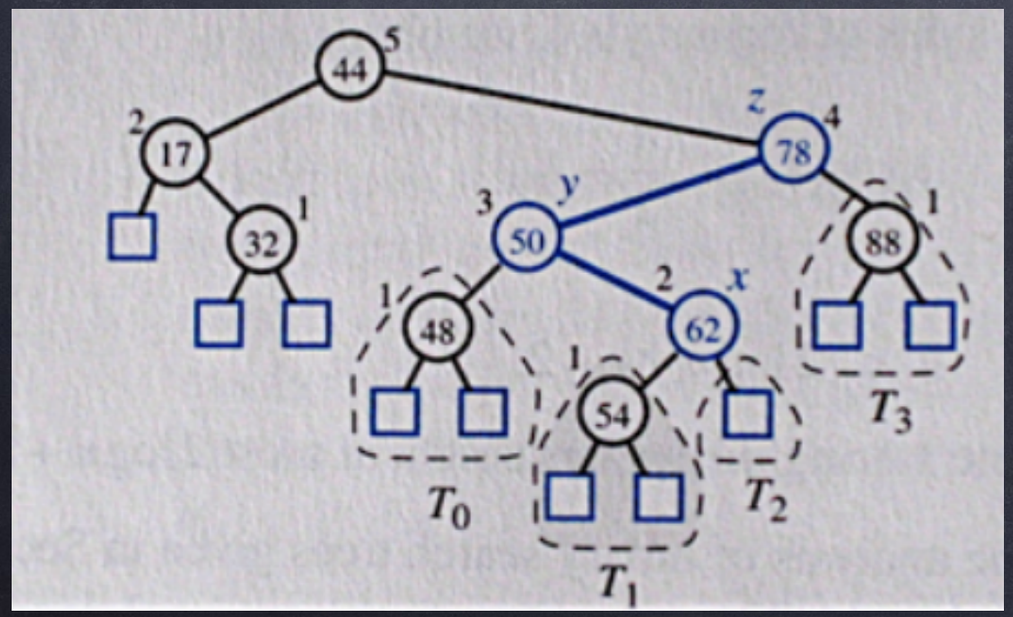


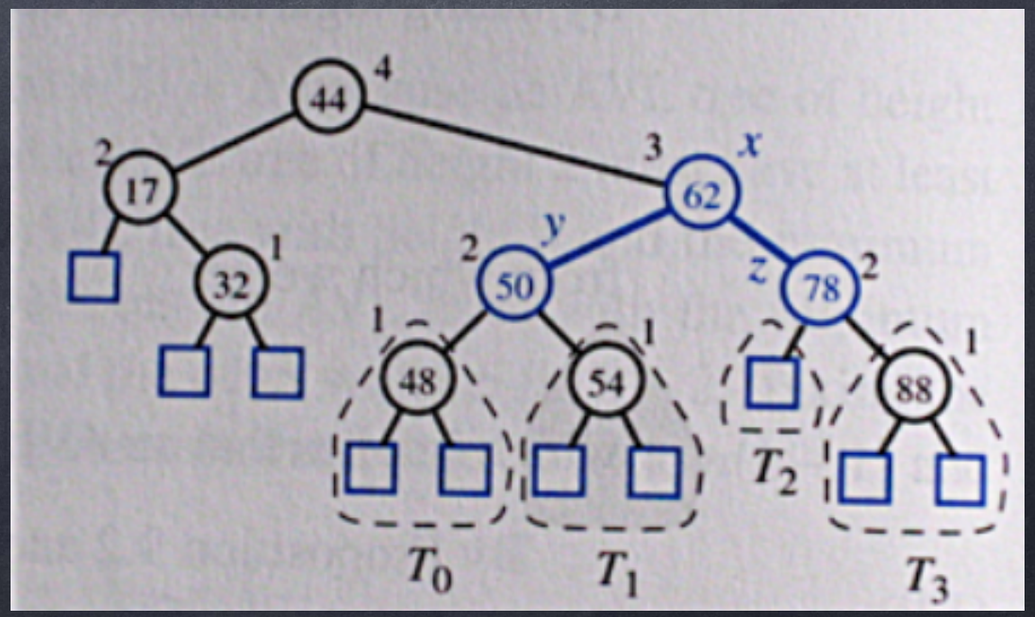
AVL-Bäume

- Ein leerer Baum ist ein AVL-Baum!
- Wir müssen es schaffen, nach Einfügen und Löschen die AVL-Eigenschaft zu erhalten
- Wenn wir das in O(log n) schaffen: Suche, Einfügen und Löschen in O(log n)
- Wie das geht: Benutzung von Restructure auf geeignete Weise

AVL - Restructure

- Gegeben: Knoten x, Vater y, Großvater z
- Bei z ist die AVL-Bedingung verletzt
- Restructure:
  - Sortiere (x,y,z) nach ihrem Schlüssel, erhalte (a,b,c)
  - Sortiere die 4 Teilbäume darunter, erhalte (T_0,T_1,T_2,T_3)
  - Ersetze Teilbaum unter z durch Teilbaum mit Wurzel b
  - Linkes Kind von b wird a, rechtes Kind wird c
  - a erhält T_0 und T_1, c erhält T_2 und T_3 als Kinder





AVL - Einfügen

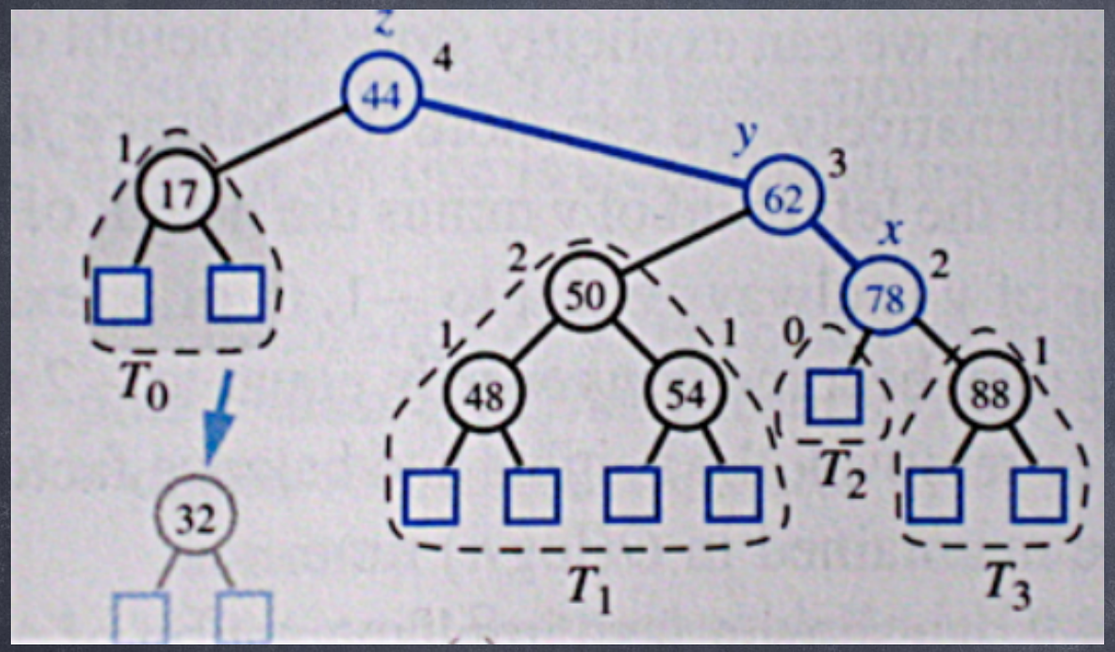
- Damit läuft Einfügen von v in einen AVL-Baum so ab:
  - Füge zunächst den Knoten v hinzu wie bei normalem binärem Baum
  - Suche dann auf dem Weg zur Wurzel nach z mit verletzter AVL-Bedingung
  - Führe Restructure auf z, y, x aus
  - y,x sind die (eindeutigen!) Nachfolger von z auf dem Weg zu v
  - Fertig - ein Restructure-Schritt reicht!

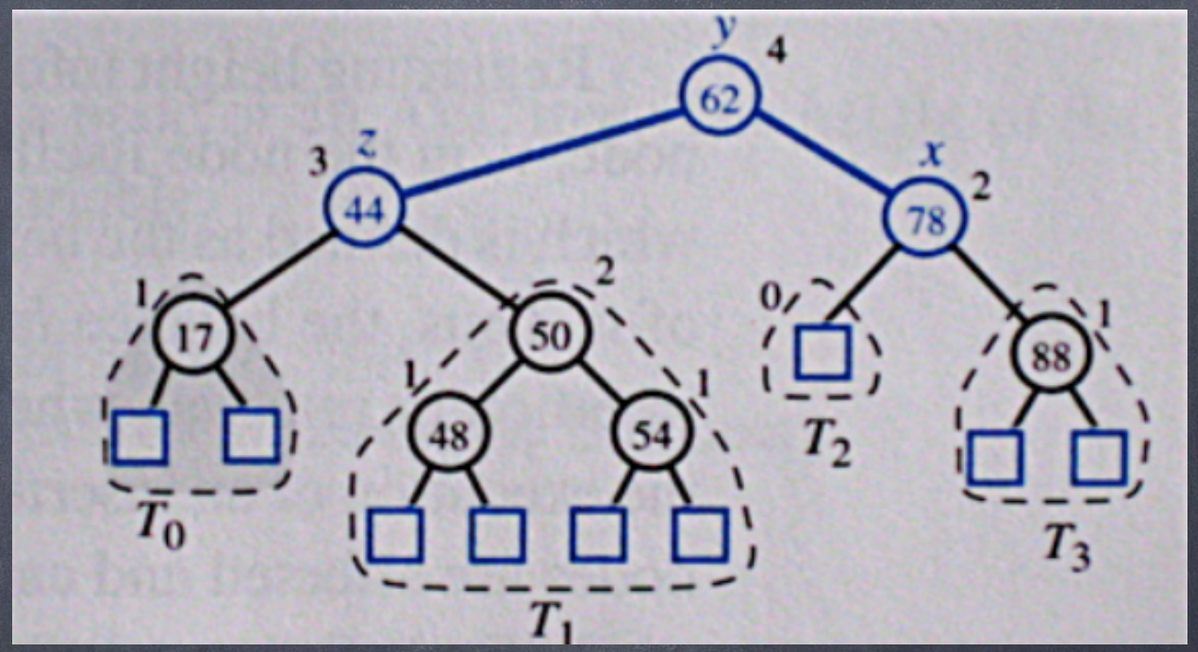
AVL – Löschen

- Löschen von v ist ein wenig komplizierter:
  - Lösche erst den Knoten v wie bei normalem binären Baum – d.h. im Zweifelsfall wieder den Nachfolger als Ersatzknoten bestimmen usw.
  - Vom tatsächlich gelöschten Knoten aus nach oben:
    Suche niedrigsten Knoten z mit verletzter AVL-Bedingung
  - Das höhere Kind von z ist y; dessen höheres Kind ist x.
    Bei Gleichstand verwende das rechte Kind.
  - Wende Restructure auf (z,y,x) an.
  - Dies produziert einen neuen Teilbaum z' an der alten Position von z
  - Falls dieser eine geringere Höhe hat als die von z vorher kann dadurch die AVL-Bedingung weiter oben im Baum verletzt werden. Suche also ggf. weiter, bis die Wurzel erreicht wurde.
- Bei logarithmischer Höhe geht das alles mit O(log n) Restructure-Schritten, d.h. in Zeit O(log n).

AVL - Löschen

Hier reicht ein Restructure-Schritt:





Allerdings: Hier gibt es Gleichstand!

Es kämen sowohl 50 als auch 78 als x in Frage!

Kann Gleichstand auch für y auftreten?

Beispiele!

siehe Tafel

Beispielaufgaben Klausur

- Gegeben folgender AVL-Baum, füge erst 2, dann 21, dann 3 und dann 5 ein. Gib nach jeder Einfüge- und Restructure-Operation den entstehenden Baum an.
- Lösche aus dem entstehenden AVL-Baum 7 und 21.
- Reicht bei einer Einfügeoperation in einen AVL-Baum immer ein Aufruf von Restructure aus? Wie ist das bei Löschoperationen?
- Wie viele Knoten n(h) hat ein AVL-Baum der Höhe h mindestens? Gib einen AVL-Baum der Höhe 5 mit höchstens 12 Knoten an.