Wörterbücher: Hashtabellen & Suchbäume

Christian Knauer

vc: md5:c4b9d23c5d534421 - (handout - built: June 2, 2020)

ADT Wörterbuch

- Der ADT Wörterbuch modelliert eine durchsuchbare Sammlung von Schlüssel-Wert Paaren
- Die primären Operationen auf einem Wörterbuch sind die Suche, das Einfügen und das Löschen von Elementen
- Es ist zulässig, dass mehrere Elemente in einem Wörterbuch den gleichen Schlüssel haben

ADT Wörterbuch (2)

- Methoden eines ADT Wörterbuch:
  1. *findElement*(k): wenn das Wörterbuch einen Eintrag mit Schlüssel k hat, wird das Element des Eintrags ausgegeben, anderenfalls wird ein Fehler gemeldet (NO_SUCH_KEY)
  2. *insertItem*(k, o): fügt den Eintrag (k, o) in das Wörterbuch ein
  3. *removeElement*(k): Wenn das Wörterbuch einen Eintrag mit Schlüssel k hat, wird dieses gelöscht und das in dem Eintrag gespeicherte Element ausgegeben, anderenfalls wird ein Fehler gemeldet (NO_SUCH_KEY)
  4. *size*$()$, *isEmpty*$()$
  5. *keys*$()$, *elements*$()$

Implementierung des ADT Wörterbuch

Hashfunktionen & Hashtabellen

- Eine Hashfunktion h bildet Schlüssel auf ganze Zahlen in einem vorgegebenen Intervall $[0, N-1]$ ab
- Beispiel:

$$h(x) = x \mod N$$

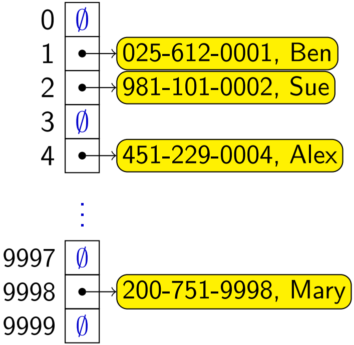
  ist eine Hashfunktion für ganzzahlige Schlüssel
- Die Zahl $h(x)$ wird der Hashwert des Schlüssels x genannt
- Eine Hashtabelle für eine gegeben Art Schlüssel besteht aus
  1. einer Hashfunktion h
  2. einem Array der Größe N
- Wenn ein Wörterbuch mittels einer Hashtabelle implementiert wird, ist das Ziel den Eintrag (k, o) an dem Index $h(k)$ im Array zu speichern

Hashfunktionen & Hashtabellen

Beispiel

- Wir haben eine Hashtabelle entworfen um Einträge der Art (SVN, Name) zu speichern, wobei die SVN (Sozialversicherungsnr.) eine neunstellige, positive Zahl ist.
- Unsere Hashtabelle verwendet ein Feld der Größe $N = 10.000$ und die Hashfunktion

  $h(x) :=$ letzte vier Stellen von x



- Hashfunktionen werden überlichweise als eine Verkettung von zwei Funktionen spezifiziert:
  Hashcodeabbildung: h_1 : Schlüssel $\rightarrow \mathbb{N}^+$
  Kompressionsabbildung: $h_2 : \mathbb{N}^+ \rightarrow [0, N-1]$

- Für die Hashfunktion wird erst die Hashcodeabbildung, dann die Kompressionsabbildung angewandt:
$$h(x) = h_2(h_1(x))$$

- Das Ziel einer Hashfunktion ist es, die Schlüssel scheinbar zufällig auf das Intervall $[0, N-1]$ zu verteilen.

Hashcodeabbildungen

Memory address & Integer cast

- Adresse des Objektes (Memory address):
  1. Die Adresse des Schüsselobjekts im Speicher wird als ganze Zahl interpretiert (Standard für alle Java-Objekte)
  2. Funktioniert im Allgemeinen gut, nur nicht bei Schlüsseln die Strings oder selbst Zahlen sind

- Ganzzahlige Konvertierung (Integer cast):
  1. Die Bits des Schlüsselobjektes werden als ganze Zahl interpretiert
  2. Passend für Schlüssel bei denen die Länge ihrer Repräsentation höchstens die Länge des `int` Datentyps ist (z.B., `byte`, `short`, `int` and `float` in Java)

Hashcodeabbildungen (2)

Component sum

- Blockweise Summe (Component sum):
  1. Die Bits des Schlüssels werden in Blöcke fester Länge zerlegt (z.B. 32 Bit) und aufsummiert (wobei mögliche Überläufe ignoriert werden)
  2. Passend für Schlüssel deren Repräsentationslänge die des `int` Datentyps überschreiten (z.B., `long` und `double` in Java)

Hashcodeabbildungen (3)

Polynomielle Akkumulation

- Polynomielle Akkumulation:
  1. die Bits des Schlüssels werden in eine Folge von Blöcken fester Länge zerlegt (z.B. 32 Bit)
     $$a_0 a_1 \dots a_{n-1}$$
  2. Daraufhin wird das Polynom
     $$p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$$
     für ein festes z ausgewertet (Überläufe werden ignoriert).
  3. Besonders geeignet für Strings: Z.B. haben für $z = 33$ auf einer repräsentativen Menge von 50000 (englischen) Wörtern höchstens 6 den gleichen Hashwert.

Kompressionsabbildungen

Division & MAD

- Division:
  1. $h_2(y) = y \mod N$
  2. N ist üblicherweise eine Primzahl

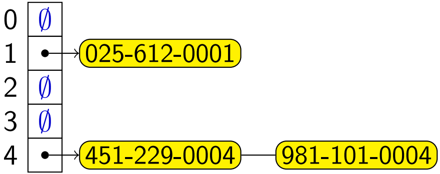
- Multiplizieren, Addieren und Dividieren (MAD):
  1. $h_2(y) = (ay + b) \mod N$
  2. a und b sind nicht negative Zahlen, so dass

$$a \mod N \neq 0$$

Kollisionen

- Kollisionen treten auf, wenn unterschiedliche Einträge auf dieselbe Zelle der Hashtabelle abgebildet werden, also den selben Hashwert haben:

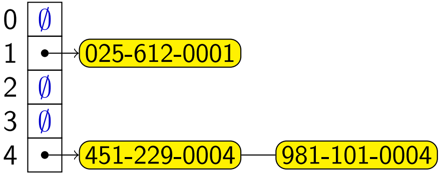
$$x \text{ und } y \neq x \text{ kollidieren (unter } h\text{) gwd. } h(x) = h(y)$$



Kollisionsbehandlung

Verkettung (Chaining)

- Jede Zelle der Hashtabelle zeigt auf eine verkettete Liste in welcher die Einträge mit dem Hashwert des Index dieser Zelle gespeichert werden



- Einfach umzusetzen, bedarf aber zusätzlichen Speichers

Kollisionsbehandlung

Offene Adressierung

Die kollidierenden Einträge werden in unterschiedlichen Zellen gespeichert:

- Lineares Probieren (engl. *linear probing*) löst Kollisionen auf, indem die (zyklisch) nächste freie Zelle verwendet wird
- Doppeltes Hashing (engl. *double hashing*) verwendet eine sekundäre Hashfunktion $d(k)$; Kollisionen werden aufgelöst, indem die erste freie Zelle der Folge

  $$h(k) + j \cdot d(k) \mod N, \text{ für } j = 0, 1, \ldots, N-1$$

  verwendet wird

Offene Adressierung

Lineares Probieren

- Löst Kollisionen auf, indem die (zyklisch) nächste freie Zelle verwendet wird
- Beispiel:
  1. $h(x) = x \mod 13$
  2. Die Schlüssel $18, 41, 22, 44, 59, 32, 31, 73$ werden in dieser Reigenfolge eingefügt

0	1	2	3	4	5	6	7	8	9	10	11	12

$\Downarrow$

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

- Nachteil: Kollidierende Einträge bilden zusammenhängende Blöcke, was zukünftige Suchsequenzen ggf. verlängert

Offene Adressierung

Suchen beim linearen Probieren

- Sei A eine Hashtabelle, die lineares Probieren verwendet
- *findElement*(k)
  1. beginne bei Zelle $h(k)$
  2. teste aufeinanderfolgende Zellen bis folgendes eintritt:
     - 2.1 ein Eintrag mit Schlüssel k wurde gefunden, oder
     - 2.2 eine leere Zelle wurde gefunden, oder
     - 2.3 N Zellen wurden erfolglos getestet

```
Algorithmus: findElement(k)
begin
    i ← h(k);
    p ← 0;
    repeat
        c ← A[i];
        if c = ∅ then
            return NO_SUCH_KEY;
        else if c.key() = k then
            return c.element();
        else
            i ← (i + 1) mod N;
            p ← p + 1;
    until p = N;
```

Offene Adressierung

Aktualisierungen beim linearen Probieren

- Um das Einfügen und Löschen zu ermöglichen wird ein spezielles Objekt namens **AVAILABLE** eingeführt das gelöschte Einträge ersetzt
- *removeElement*(k)
  1. Suche den Eintrag mit Schlüssel k
  2. Wenn (k, o) gefunden wird, ersetzte es durch das Objekt **AVAILABLE** und liefere o
  3. Anderenfalls melden einen NO_SUCH_KEY Fehler
- *insertItem*(k, o)
  1. Melde einen Fehler wenn die Tabelle voll ist
  2. Beginne bei $h(k)$
  3. Teste solange aufeinanderfolgende Zellen bis eine Zelle gefunden wird die leer ist oder das Objekt **AVAILABLE** speichert
  4. Speichere den Eintrag (k, o) in dieser Zelle

Offene Adressierung

Doppeltes Hashing

- Es wird eine sekundäre Hashfunktion $d(k)$ verwendet; Kollisionen werden aufgelöst, indem die erste freie Zelle der Folge
$$h(k) + j \cdot d(k) \mod N, \text{ für } j = 0, 1, \ldots, N-1$$
verwendet wird
- Die sekundäre Hashfunktion darf keine Nullstellen haben
- Die Größe N der Hashtabelle muss eine Primzahl sein, sodass alle Zellen betrachtet werden
- Eine übliche sekundäre Hashfunktion ist:
$$d_2(k) = 1 + (q - k \mod q),$$
wobei
    1. $q < N$
    2. q eine Primzahl ist
    3. die möglichen Werte für $d_2(k)$ sind $1, 2, \ldots, q$

Offene Adressierung

Beispiel für doppeltes Hashing

- Wir betrachten das Einfügen von ganzzahligen Schlüsseln in eine Hashtabelle unter Verwendung von doppeltem Hashing, wobei
  1. $N = 13$
  2. $h(k) = k \mod 13$
  3. $d(k) = 7 - k \mod 7$
- Wir fügen die Schlüssel $18, 41, 22, 44, 59, 32, 31, 73$ in dieser Reihenfolge ein

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Effizienz von Hashing
(ohne Beweise)

- Im schlechtesten Fall erfordern die Operationen Suchen, Einfügen und Löschen in eine Hashtabelle $O(n)$ Zeit

  Der schlechteste Fall tritt ein, wenn alle Schlüssel eine Kollision erzeugen

- Der Auslastungsfaktor (engl. *load factor*) $\alpha = \frac{n}{N}$ beeinflusst die Effizienz der Hashtabelle: Unter der Annahme, dass sich die Hashwerte wie Zufallszahlen verhalten, kann gezeigt werden, dass die *erwartete* Länge einer Testsequenz mittels offener Adressierung

  $$O(\frac{1}{1-\alpha})$$

  beträgt (für $n < N$)

- Die erwartete Laufzeit der Operationen Suchen, Einfügen und Löschen in eine Hashtabelle ist $O(1)$

ADT geordnetes Wörterbuch

- Es gibt eine totale Ordnung auf den Schlüsseln

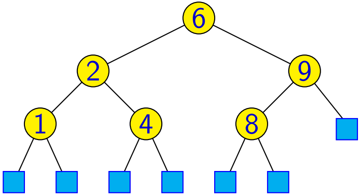
- Neue Methoden:
  1. $closestKeyBefore(k)$
  2. $closestElemBefore(k)$
  3. $closestKeyAfter(k)$
  4. $closestElemAfter(k)$

Binäre Suchbäume

- Ein binärer Suchbaum ist ein Binärbaum der Schlüssel (Schlüssel-Element Paare) in den internen Knoten speichert, mit der Eingeschaft:
  1. Seien u, v und w drei Knoten, so dass u im linken Teilbaum von v und w im rechten Teilbaum von v liegt. Dann ist

$$key(u) \leq key(v) \leq key(w)$$

- Externe Knoten speichern keine Werte



- Konsequenz: Eine inorder-Traversierung gibt die Schlüssel in aufsteigender Reihenfolge aus

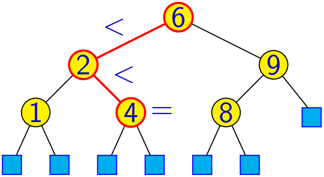
Suche in binären Suchbäumen

- Um einen Schlüssel k zu suchen, wird ein Pfad beginnend bei der Wurzel verfolgt
- Für jeden Knoten der auf dem Pfad besucht wird, wird der gespeicherte Schlüssel mit k verglichen und entsprechend in den linken oder rechten Teilbaum abgestiegen (wenn der Schlüssel ungleich k)
- Sollte ein Blatt erreicht werden, wird eine NO_SUCH_KEY Fehler gemeldet

Suche in binären Suchbäumen

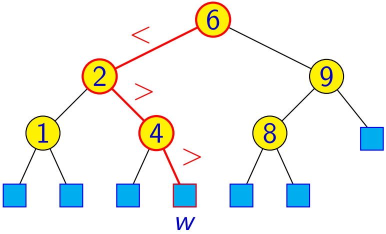
Pseudocode & Beispiel

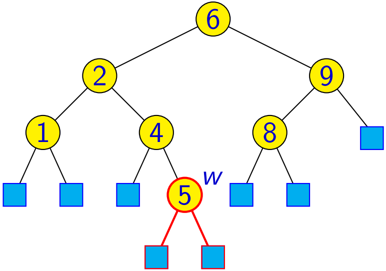
```
Algorithmus: findElement(k, v)
begin
    if T.isExternal(v) then
        return NO_SUCH_KEY;
    if k < key(v) then
        return
          findElement(k, T.leftChild(v));
    else if k = key(v) then
        return element(v);
    else if k > key(v) then
        return
          findElement(k, T.rightChild(v));
```



Einfügen

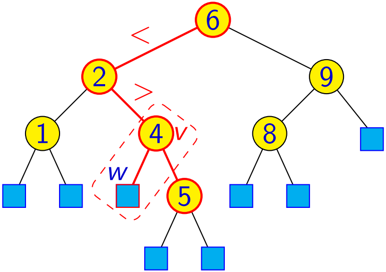
- Um eine Einfügeoperation *insertItem*(k, o) auszuführen wird zunächst der Schlüssel k gesucht
- Nehmen wir an, k sei nicht bereits im Baum gespeichert und sei w das Blatt, welches bei der Suche gefunden wurde
- k wird in w gespeichert und der Knoten w zu einem internen Knoten erweitert
- Beispiel: Einfügen von 5



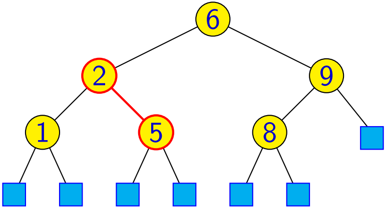


Löschen - Teil 1

- Um die Operation *removeElement*(k) auszuführen wird zunächst k gesucht
- Nehmen wir an, k sei im Baum enthalten und v sei der Knoten der k speichert

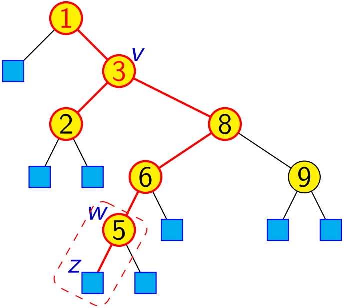


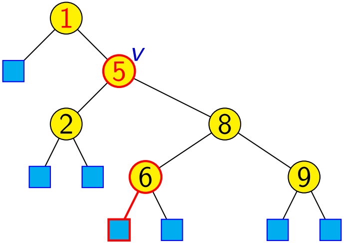
- Falls v als Kind ein Blatt w hat, so entfernen wir v und w aus dem Baum mittels der Methode *removeAboveExternal*(w)
- Beispiel: Löschen von 4



Löschen - Teil 2

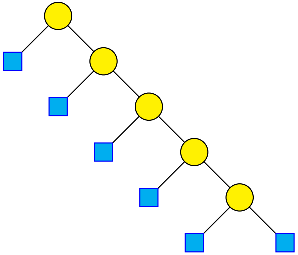
- Falls die Kinder von v beide intern sind
    1. suchen wir den Knoten w der auf v in inorder-Traversierungsordnung folgt
    2. kopieren wir $key(w)$ (& das in w gespeicherte Element) nach v
    3. löschen wir w und dessen linkes Kind z (welches ein Blatt sein muss) mittels der Operation *removeAboveExternal*(z)
- Beispiel: Löschen von 3





Effizienz

- Für ein Wörterbuch in dem n Einträge gespeichert sind und das mittels eines binären Suchbaums der Höhe h implementiert wurde
  1. ist der Platzbedarf $O(n)$
  2. benötigen die Methoden *findElement*, *insertItem* und *removeElement* je $O(h)$ Zeit
- Die Höhe h ist im schlechtesten Fall $\Omega(n)$ und im besten Fall $O(\log n)$



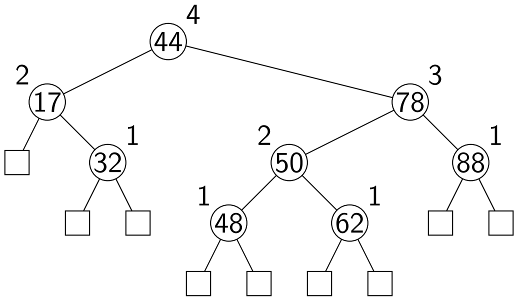
AVL-Baum

Definition

- Ein AVL-Baum ist ein binärer Suchbaum mit der Eigenschaft:

  Für jeden inneren Knoten v unterscheiden sich die Höhen der (Teilbäume unter den) Kindknoten von v um maximal 1.

- Beispiel (die Höhen sind neben den Knoten angegeben):



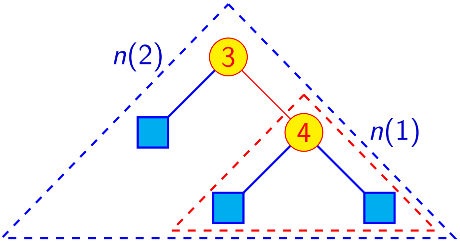
AVL-Baum

Höhe eines AVL-Baumes (1)

Satz: Die Höhe eines AVL-Baumes der n Schlüssel speichert ist $O(\log n)$.

Beweis:

- Sei $n(h)$ die minimale Anzahl interner Knoten eines AVL-Baumes mit Höhe h.
- Es ist $n(1) = 1$ und $n(2) = 2$.
- Für $n > 2$ enthält ein AVL-Baum von Höhe h den Wurzelknoten, einen AVL-Teilbaum mit Höhe $h - 1$ und einen mit Höhe $h - 2$.



AVL-Baum

Höhe eines AVL-Baumes (2)

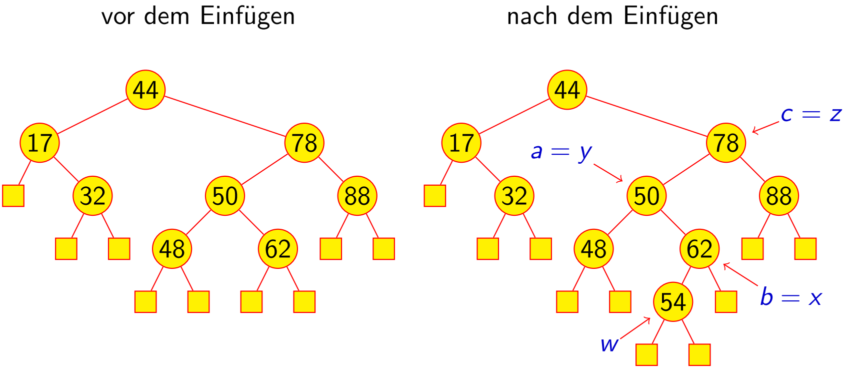
- Es gilt also $n(h) = 1 + n(h-1) + n(h-2)$.
- Wir wissen, dass $n(h-1) > n(h-2)$, also $n(h) > 2n(h-2)$.
- Mit Induktion folgt $n(h) > 2^i n(h-2i)$.
- Für $i = h/2 - 1$ erhält man $n(h) > 2^{\frac{h}{2}-1}$.
- Damit folgt $h < 2\log n(h) + 2$.

- Also ist die Höhe eines AVL-Baumes der n Schlüssel speichert $O(\log n)$.

Einfügen in einen AVL-Baum

1 - Erweitern eines äußeren Knotens

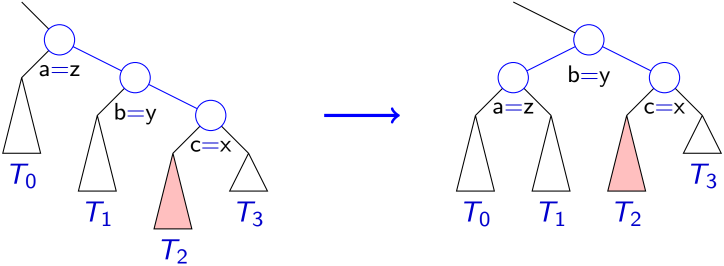
- Einfügen funktioniert (zunächst) wie beim binären Suchbaum durch Erweitern eines äußeren Knotens.
- *Beispiel*: Einfügen von 54

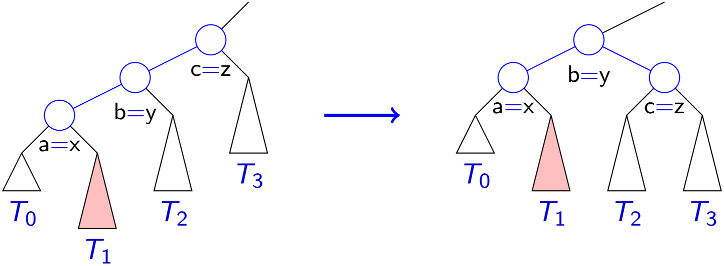


Sei z der erste Knoten auf dem Pfad P von w zur Wurzel der nicht balanciert ist, y das Kind von z auf P und x das Kind von y auf P.

Lokale Restrukturierung von Suchbäumen

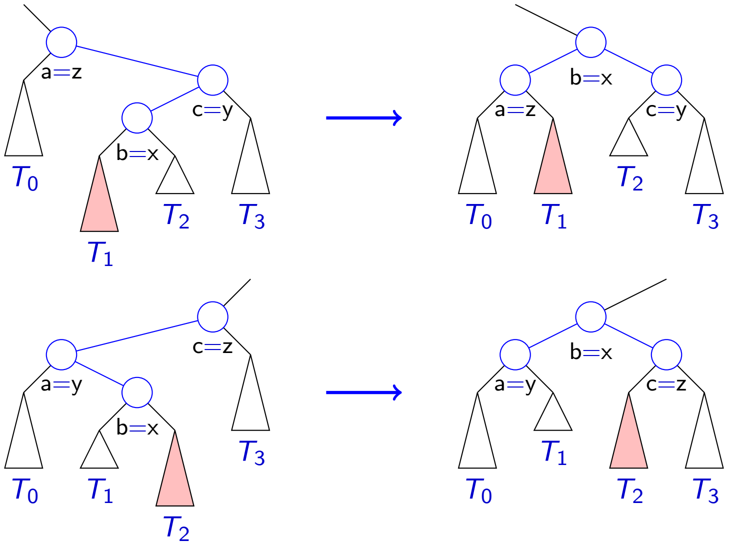
Einfache Rotation





Lokale Restrukturierung von Suchbäumen

Doppelte Rotation

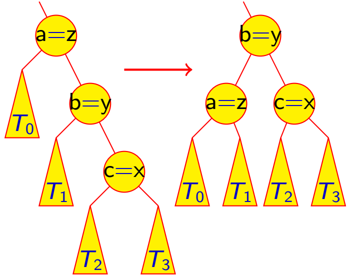


Einfügen in einen AVL-Baum

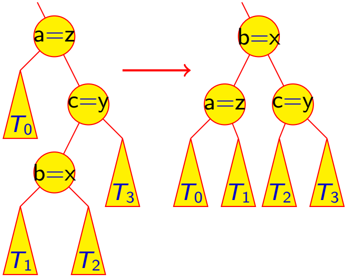
2 - Restrukturierung

- Sei (a, b, c) die Inordner-Reihenfolge von x, y, z.
- Führe die Rotationen aus, die benötigt werden, um b zum obersten Knoten der drei zu machen.

Fall 1: einfache Rotation



Fall 2: doppelte Rotation

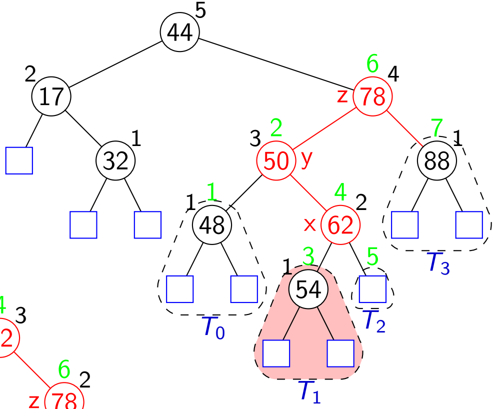


(Die übrigen beiden Fälle sind symmetrisch)

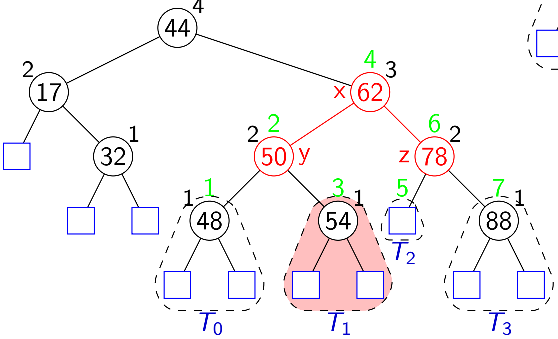
Einfügen in einen AVL-Baum

Beispiel, Fortsetzung

unbalanciert...



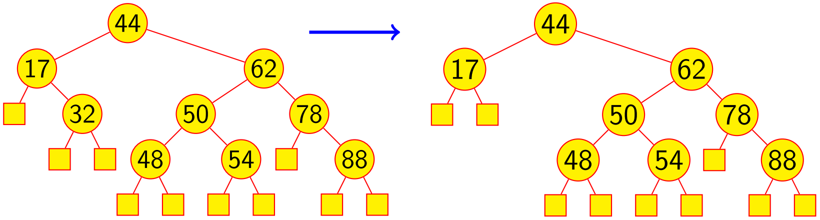
...balanciert



Entfernen aus einem AVL-Baum

1 - Löschen eines Knotens

- Das Entfernen beginnt wie in einem binären Suchbaum: Der gelöschte Knoten wird zu einem leeren, äußeren Knoten.
- Sein Elternknoten w kann ein Ungleichgewicht verursachen.
- *Beispiel*: Entfernen von 32

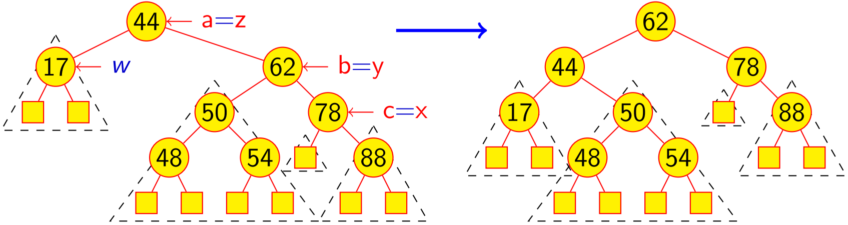


vor dem Entfernen | nach dem Entfernen

Entfernen aus einem AVL-Baum

2 - Rebalancierung nach einer Löschung

- Sei z der erste unbalancierte Knoten, der beim Durchlaufen des Baumes von w nach oben getroffen wird. Desweiteren sei y das Kind von z mit der größeren Höhe und x das Kind von y mit der größeren Höhe.
- Wir führen *restructure*(x) durch um die Balance bei z wiederherzustellen.
- Da diese Restrukturierung die Balance an anderen, näher an der Wurzel liegenden Knoten stören kann, müssen wir ggf. mit der Rebalancierung fortfahren bis die Wurzel erreicht ist.



Laufzeitanalyse AVL-Bäume

- Eine einfache Restrukturierung erfordert $O(1)$ Zeit
  1. falls eine verzeigerte Datenstruktur verwendet wird
- Das Finden eines Schlüssels erfordert $O(\log n)$ Zeit
  1. Die Höhe des Baumes ist $O(\log n)$; es sind keine Restukturierungen notwendig
- Das Einfügen eines Schlüssels erfordert $O(\log n)$ Zeit
  1. Das Finden der Einfügeposition erfordert $O(\log n)$ Zeit
  2. Die Restrukturierung des Baumes nach oben erfordert $O(\log n)$ Zeit
- Das Entfernen eines Schlüssels erfordert $O(\log n)$ Zeit
  1. Das Finden der Löschposition erfordert $O(\log n)$ Zeit
  2. Die Restrukturierung des Baumes nach oben erfordert $O(\log n)$ Zeit