4. Suchen

4.1 Elementare Methoden auf flachen Datenstrukturen

4.2 Binärbäume als Suchbäume

4.3 Mehrwegbäume als Suchbäume

4.4 Digitale Suchbäume (Radix-Suche)

4.5 Hashing

4.1 Elementare Methoden auf flachen Datenstrukturen

Die zu verwaltende Datenmenge besteht aus Datensätzen, die Suchschlüssel enthalten.

Grundlegende Operationen auf der Datenmenge

- Suchen eines bestimmten Datensatzes
- Einfügen eines Datensatzes
- Löschen eines Datensatzes

Behandlung von Datensätzen mit gleichem Schlüssel, zwei Möglichkeiten:

(1) Verboten in der primären Datenstruktur.
   - Nur jeweils ein Datensatz wird von den Algorithmen verarbeitet
   - Weitere Elemente mit gleichem Schlüssel werden als verkettete Liste an das erste Element angehängt.
   - Es gibt eine zusätzliche Operation "Finde Duplikate"

(2) Mehrere Elemente mit gleichem Schlüssel in der primären Datenstrukur erlaubt.
   - Die Algorithmen für die primäre Datenstruktur werden etwas komplexer, dafür entfällt die separate Behandlung der verketteten Liste.

4.1.1 Sequentielle Suche

Zwei Implementierungsmöglichkeiten

- **(a)** Speicherung der Datensätze in ungeordneter Folge
- **(b)** Speicherung sortiert

Implementierung (a), ungeordnete Folge

Algorithmus

Die Daten werden in einem Feld (array) gespeichert, Einfügen erfolgt am Ende, Suchen erfolgt von Anfang an, durch Vergleich mit jedem Schlüssel.

Programm

```
/* v is the key, info is the data field */
static struct node
  { int key; int info; };
static struct node a[maxN+1];
static int N;

seqinitialize()
  { N = 0; }
int seqsearch (int v)
  {
    int x = N+1;
    a[0].key = v; a[0].info = -1;
    while (v != a[--x].key) ;
    return a[x].info;
  }
seqinsert(int v, int info)
  { a[++N].key = v; a[N].info = info; }
```

Komplexität

Zahl der Vergleiche:
im Mittel N/2, falls erfolgreich
N, falls erfolglos

Implementierung (b), sortierte Folge

Algorithmus

Die Datensätze werden als verkettete Liste gespeichert. Einfügen erfolgt an der richtigen Stelle gemäß der Sortierordnung. Suchen erfolgt von Anfang an, bis zu der richtigen Stelle, durch Vergleich mit jedem Element.

Programm

```
static struct node
  { int key, info; struct node *next; };
static struct node *head, *z;

listinitialize ()
  {
    head = (struct node *) malloc(sizeof *head);
    z = (struct node *) malloc(sizeof *z);
    head->next = z; z->next = z; z->info = -1;
  }
```

```
int listsearch(int v)
  {
    struct node *t = head;
    z->key = v;
    while (v > t->key) t = t->next;
    if (v != t->key) return z->info;
    return t->info;
  }


listinsert (int v, int info)
  {
    struct node *x, *t = head;
    z->key = v;
    while (v > t->next->key) t = t->next;
    x = (struct node *) malloc(sizeof *x);
    x->next = t->next; t->next = x;
    x->key = v; x->info = info;
  }
```

Komplexität

- Suche kann abgebrochen werden, sobald ein Datensatz mit Schlüssel > Suchschlüssel gefunden wird.
- Zahl der Vergleiche:

im Mittel	N/2,	falls erfolgreich
im Mittel	N/2,	falls erfolglos

4.1.2 Binäre Suche

Voraussetzung: Speicherung in Sortierordnung

Algorithmus:

Rekursives Unterteilen des Suchbereichs in zwei gleiche Hälften, von denen nur eine weiter untersucht wird.

Programm

Die Datensätze seien in einem Feld (array) a der Dimension N sortiert gespeichert. v ist der Suchschlüssel.

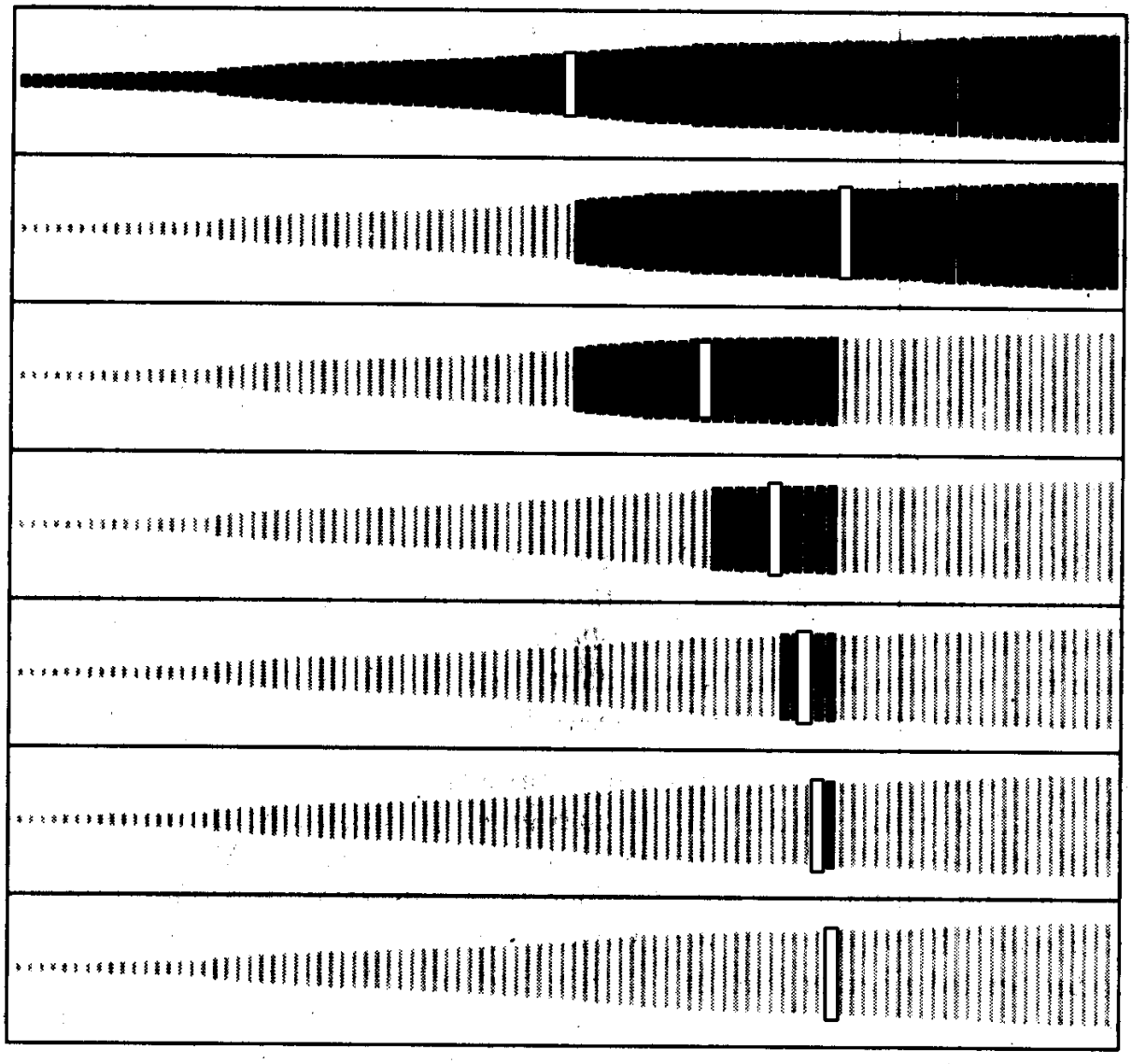
```
int binsearch(int v)
  {
    int 1 = 1; int r = N; int x;
    while (r >= 1)
      {
        x = (1+r)/2;
        if (v < a[x].key) r = x-1; else 1 = x+1;
        if (v == a[x].key) return a[x].info;
      }
    return -1;
  }
```

Binäre Suche, Beispiel

Suchschlüssel: M

A	A	A	C	E	E	E	G	H	I	L	M	N	P	R	S	X
									I	L	M	N	P	R	S	X
									I	L	M					
											M					

Visualisierung der binären Suche



Komplexität

- Zahl der Vergleiche: $\leq \log_2 N+1$ für erfolgreiche und erfolglose Suche
- Einfügen eines Datensatzes dauert relativ lange:
  Im Mittel müssen N/2 Elemente verschoben werden. Die Datensätze müssen in einem Feld gespeichert sein; eine verkettete Liste würde den Direktzugriff nicht erlauben.

Kommentar

Ein in der Praxis sehr gut einsetzbares Verfahren, wenn die Anzahl von Veränderungen gering ist.
Ein klassisches Beispiel für einen Algorithmus vom Typ Teile-und-Herrsche (Divide-and-Conquer).

4.1.3 Interpolationssuche

Idee: Verbesserung der binären Suche derart, daß der Teilungsindex abhängig vom gesuchten Schlüssel und den Schlüsselwerten an den Intervallgrenzen gewählt wird. Entspricht der "Suche im Telefonbuch".

Annahme: Alle Schlüsselwerte etwa gleichverteilt.

Es sei a das Feld mit den Schlüsselwerten, aufsteigend sortiert
l, r die Indizes an der linken und rechten Intervallgrenze
v der gesuchte Schlüssel

Bei der binären Suche hatten wir den Teilungsindex x wie folgt berechnet:

$$x = l + \frac{1}{2} * (r - l)$$

Bei der Interpolationssuche wird der Teilungsindex x nunmehr wie folgt berechnet:

$$x = l + \frac{(v - a[l].key)}{a[r].key - a[l].key} * (r - l)$$

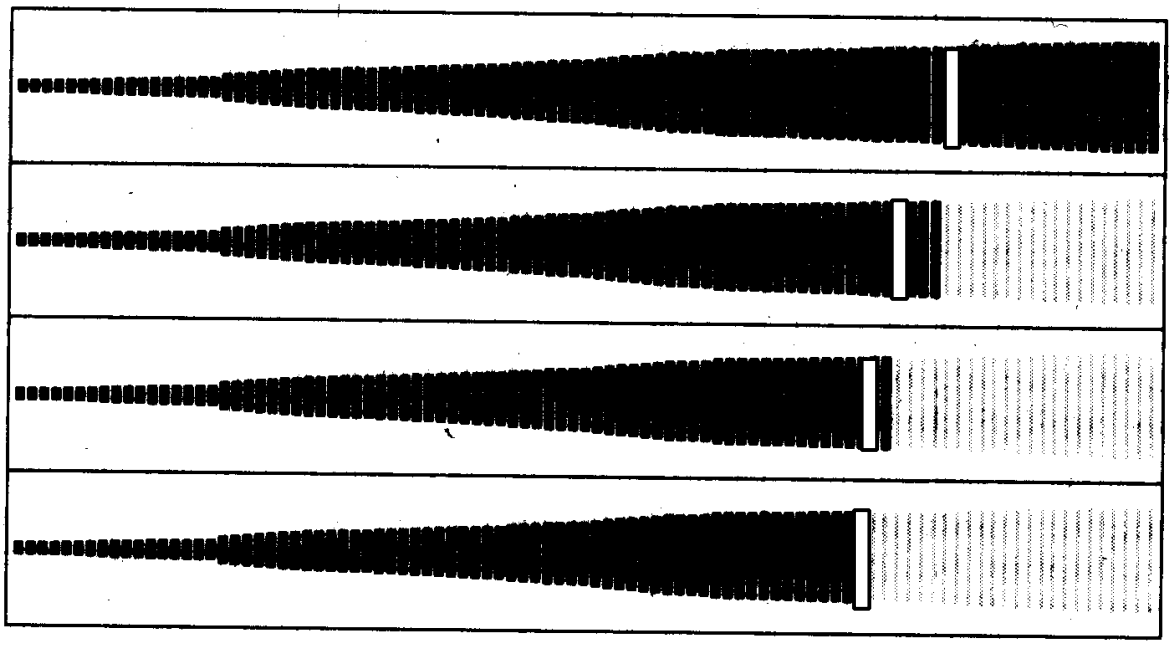
Interpolationssuche, Beispiel

Suchschlüssel: M

Erster Schritt: x = 1 + (13-1) * (17-1) / (24-1) = 9,3

A	A	A	C	E	E	E	G	H	I	L	M	N	P	R	S	X
									I	L	M	N	P	R	S	X
											M	N	P	R	S	X

Visualisierung der Interpolationssuche



Komplexität

- Falls die Schlüssel zufällig verteilt sind, benötigt die Interpolationssuche

  $\log_2 \log_2 N + 1$

  Vergleiche (Beweis schwierig). Diese Funktion wächst sehr langsam mit N.

  Beispiel: $\log_2 \log_2 10^9 < 5$.

Kommentar

- Die Interpolationssuche ist für sehr große, sortierte Felder mit annähernd gleichverteilten Schlüsselwerten und geringer Änderungshäufigkeit hervorragend geeignet.
- Nachteil: Die Berechnung von x ist aufwendiger als bei der binären Suche.

4.1.4 Selbstanordnende Listen

Wenn die Zugriffshäufigkeiten auf die Datensätze deutlich von der Gleichverteilung abweichen, kann es sich lohnen, die am häufigsten gesuchten Datensätze in einer verketteten Liste weiter vorne anzuordnen und dann jeweils linear in der Liste zu suchen. Die Reorganisation der Liste erfolgt dynamisch.

Drei Verfahren sind dazu gebräuchlich:

1. *MF-Regel* (Move-to-front):
Mache ein Element zum ersten Element der Liste, nachdem auf das Element (als Ergebnis einer erfolgreichen Suche) zugegriffen wurde. Die relative Anordnung der übrigen Elemente bleibt unverändert.

2. *T-Regel* (Transpose):
Vertausche ein Element mit dem unmittelbar vorangehenden, nachdem auf das Element zugegriffen wurde.

3. *FC-Regel* (Frequency Count):
Ordne jedem Element einen Häufigkeitszähler zu, der anfangs 0 ist und die Anzahl der Zugriffe auf das Element speichert. Nach jedem Zugriff auf ein Element wird dessen Häufigkeitszähler um 1 erhöht. Ferner wird die Liste nach jedem Zugriff neu geordnet, und zwar so, daß die Häufigkeitszähler der Elemente in absteigender Reihenfolge angeordnet sind.

Anmerkung: bei großen Datensätzen mit kleinen Schlüsselfeldern ist es sinnvoll, nur jeweils den Schlüssel und einen Zeiger auf den Inhalt als Element der Liste zu verwenden.

Selbstanordnende Liste, Beispiel

In einem Internet-Name-Server wird eine Tabelle benötigt, die logische Hostnamen auf IP-Adressen abbildet. Auf dem lokalen Host wird ein großer Cache mit häufig angefragten Hostnamen gehalten, um die Zugriffszeit und die Netzbelastung zu reduzieren.

Die Tabelleneinträge haben die Form

Hostname	IP-Adresse
pi4.informatik.uni-mannheim.de	156.47.98.13
i5.informatik.uni-mannheim.de	156.47.97.79
fourier.math.uni-mannheim.de	156.47.96.48
.	.
.	.
.	.

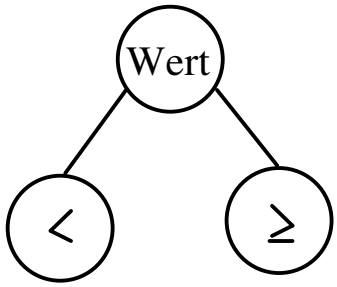
Es ist sinnvoll, diese Tabelle selbstanordnend zu verwalten.

4.2 Binärbäume als Suchbäume

Idee: Verwendung einer Datenstruktur, in der das Suchen, Einfügen und Löschen eines beliebigen Datensatzes relativ schnell geht (schneller als O(N)).

Lösung: Binärbaum als Suchbaum

Ordnungsregel:



Jeder Knoten im Baum entspricht einem Datensatz.

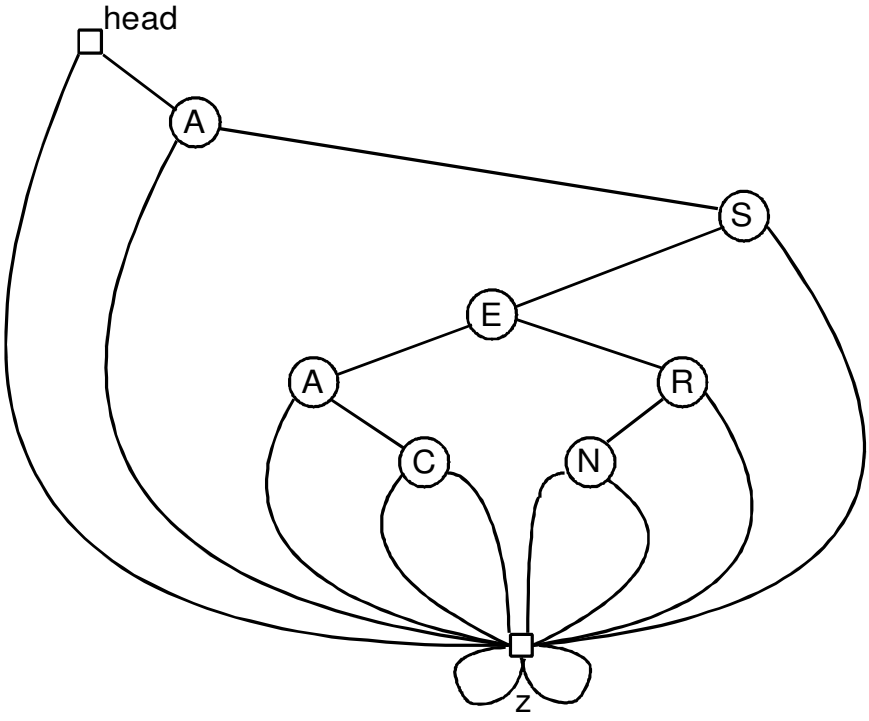
Algorithmus Treesearch:

Suche im Binärbaum nach einem Knoten, dessen Schlüssel gleich dem Suchschlüssel ist.

- Beginne mit der Wurzel
- Falls Suchschlüssel
  - < Schlüssel des aktuellen Knotens:
    Suche im linken Teilbaum weiter
  - = Schlüssel des aktuellen Knotens:
    fertig
  - > Schlüssel des aktuellen Knotens:
    Suche im rechten Teilbaum weiter

Programm:

```
/* head ist ein Hilfsknoten für den Anker; z ist ein
Hilfsknoten, der zum Erkennen von Blattkonten dient */
```



```
static struct node
  { int key, info; struct node *l, *r; };
static struct node *t, *head, *z;

int treesearch(int v)
  {
    struct node *x = head->r;
    z->key = v;
    while (v != x->key)
      x = (v < x->key) ? x->l : x->r;
    return x->info;
  }
```

Initialisieren des Binärbaums

Programm:

```
treeinitialize()
  {
    z = (struct node *) malloc (sizeof *z);
    z->l = z; z->r = z; z->info = -1;
    head = (struct node *) malloc(sizeof *head);
    head->r = z; head->key =0; head->l = z;
  }
```

Einfügen eines neuen Knotens in einen binären Suchbaum

- Suche den Schlüssel des neuen Knotens (erfolglos) im Baum. Das bestimmt die Einfügestelle, d.h. den werdenden Vater.

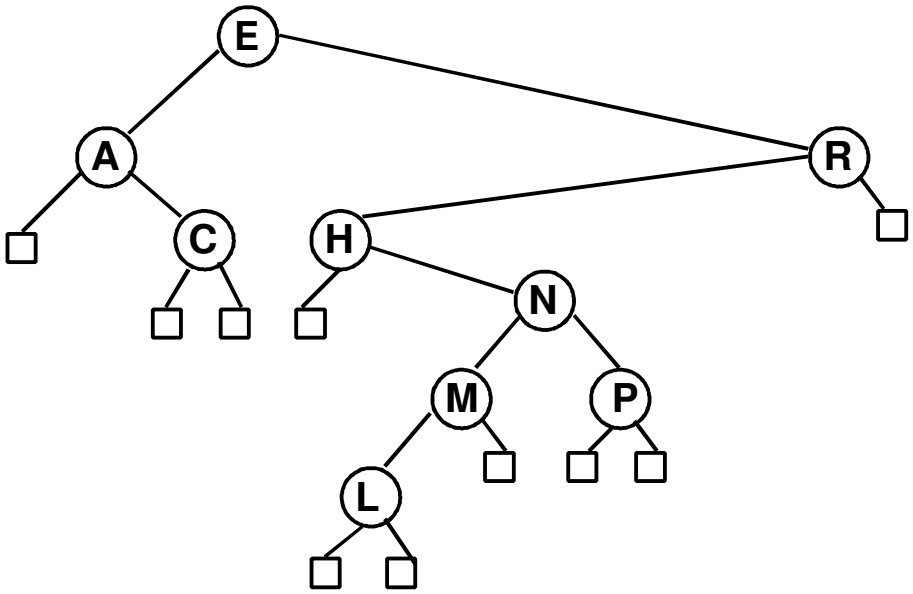
- Setze den Zeiger des werdenden Vaters auf den neuen Knoten. Dieser wird stets ein neues Blatt sein.

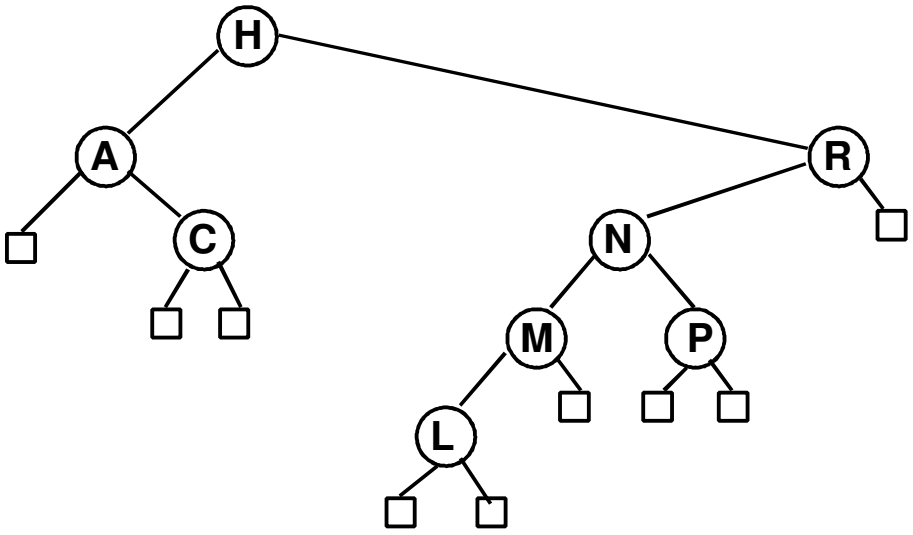
Programm

```
treeinsert (int v, int info)
  {
    struct node *p, *x;
    p = head; x= head->r;
    while (x !=z)
      { p = x; x= (v <x->key) ? x->l : x->r }
    x=(struct node *) malloc(sizeof *x);
    x->key = v; x->info = info;
    x->l = z; x->r = z;
    if (v<p->key) p->l = x; else p->r =x;
  }
```

Löschen im Binärbaum: leichte Fälle, schwere Fälle

Löschen des Knotens E





Löschen eines Knotens in einem binären Suchbaum

- Knoten hat keine Kinder ⇒ ersetze ihn durch z
- Knoten hat nur ein Kind ⇒ ersetze ihn durch sein Kind
- Knoten hat zwei Kinder, von denen eines selbst keine Kinder hat
  ⇒ ersetze ihn durch dieses Kind
- Knoten hat zwei Kinder, die selbst beide ebenfalls Kinder haben
  ⇒ ersetze ihn durch den Knoten mit dem nächsthöheren Schlüssel

Programm

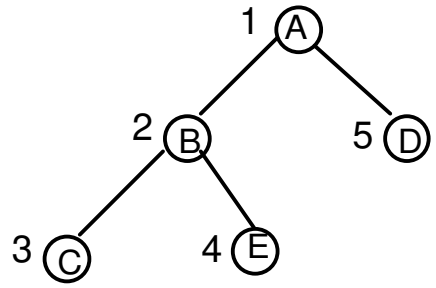
```
/* Annahme: der zu löschende Knoten existiert */
treedelete (int v)
  {
    struct node *c, *p, *x;
    z->key = v; p=head; x=head->r;
    while (v!=x->key)
      { p=x; x=(v < x->key) ? x->l : x->r; }
    t=x;
    if (t->r == z) x= x->l;
    else if (t->r->l == z) { x=x->r; x->l = t->l;  }
    else
      {
        c=x->r; while (c->l->l !=z)  c=c->l;
        x=c->l; c->l = x->r;
        x->l = t->l; x->r = t->r;
      }
    free (t);
    if (v < p->key) p->l = x; else p->r =x;
  }
```

Durchlaufen von Binärbäumen (Tree Traversal)

Gebräuchlich sind vier Verfahren zum Durchlaufen von Binärbäumen derart, daß jeder Knoten genau einmal verarbeitet wird:

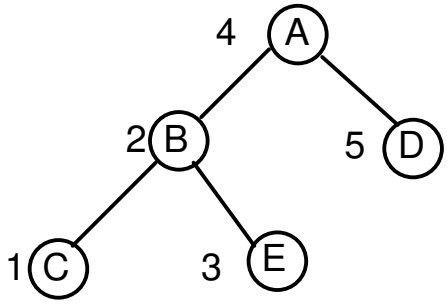
Preorder

Regel:
1. Wurzel
2. Linker Unterbaum
3. Rechter Unterbaum



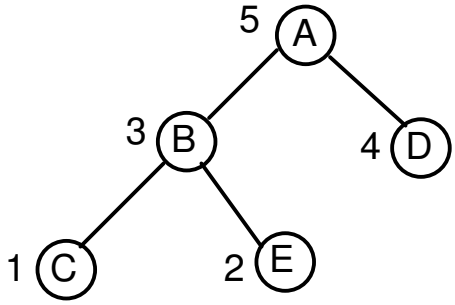
Inorder

Regel:
1. Linker Unterbaum
2. Wurzel
3. Rechter Unterbaum



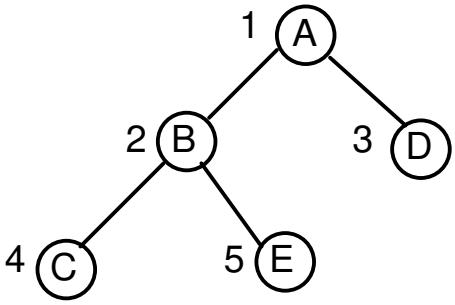
Postorder

Regel: 1. Linker Unterbaum
2. Rechter Unterbaum
3. Wurzel



Level - Order

Regel: 1. Alle Knoten einer Ebene werden von links nach rechts besucht.
2. Alle Ebenen werden von oben nach unten behandelt.



"Gute" und "schlechte" Binärbäume

Für jeden Knoten im Baum ist die Anzahl der Vergleiche, die zum Finden benötigt wird, gleich seinem Abstand von der Wurzel. Die Summe dieser Abstände heißt die *innere Pfadlänge* des Baumes. Die *mittlere Pfadlänge* in einem Baum ergibt sich aus 1/N*(innere Pfadlänge). Sie kann rekursiv berechnet werden als

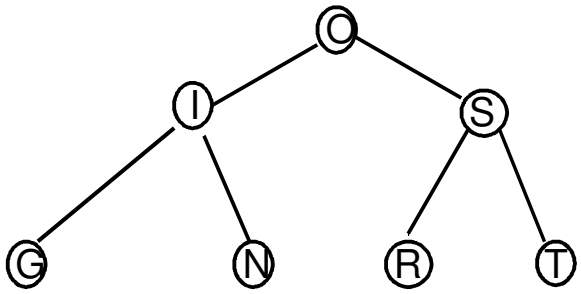
$$C_N = N + \frac{1}{N} \sum_{1 \le K \le N} (C_{k-1} + C_{N-k})$$

mit $C_1 = 1$.

Beispiele zur Gestalt von Binärbäumen

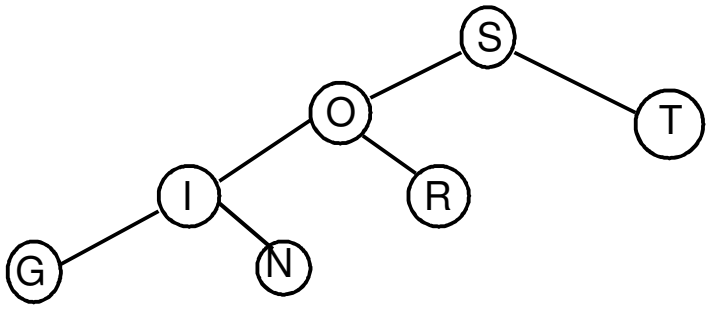
Schlüsselmenge: SORTING

1. Vollständiger Binärbaum



Mittlere Anzahl von Vergleichen: 17/7 = 2,428 im Mittel

2. Ein natürlich gewachsener Baum



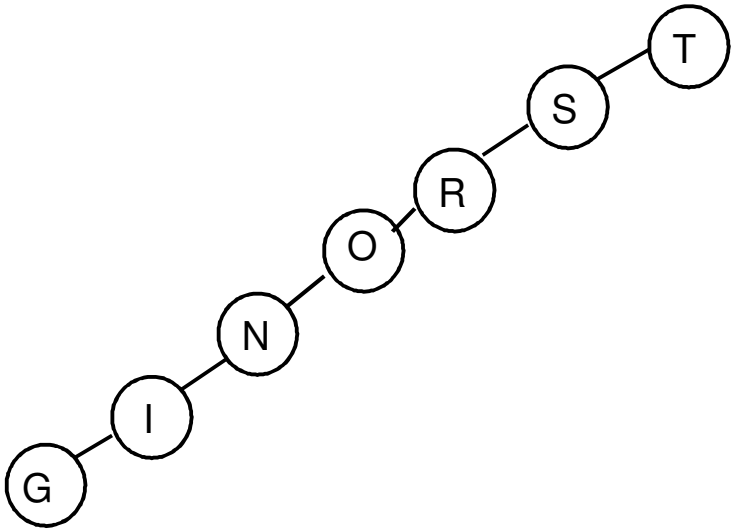
1x 1 Vergleich

2x 2 Vergleiche 19/7 = 2,714 im Mittel

2x 3 Vergleiche

2x 4 Vergleiche

3. Der ungünstigste Fall



1x 1 Vergleich

1x 2 Vergleiche

1x 3 Vergleiche

1x 4 Vergleiche

1x 5 Vergleiche

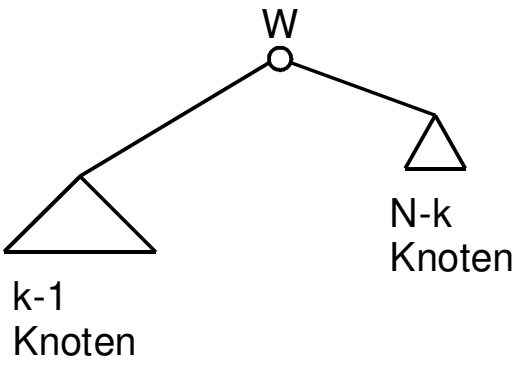
1x 6 Vergleiche

1x 7 Vergleiche

28/7 = 4 im Mittel

Herleitung der mittleren Pfadlänge

Der betrachtete Baum aus N Knoten besteht aus der Wurzel, deren Schlüssel zuerst eingefügt wurde und der k-größte des Baumes ist, sowie dem linken und dem rechten Unterbaum der Wurzel, die k-1 bzw. N-k Knoten haben:



Die Wurzel fügt zu jeder anderen Pfadlänge 1 hinzu, für N-1 Restknoten also N-1. Bei zufällig eingefügten Schlüsseln ist die Wahrscheinlichkeit für jedes k gleich 1/N. Dazu kommt noch ein einziger Vergleich, falls die Wurzel selbst gesucht wird.

Geschlossene Berechnung des rekursive Ausdrucks

$$C_N = 1+N-1+\frac{1}{N}\sum_{k=1}^{N} (C_{k-1}+C_{N-k})$$

$$= N+\frac{1}{N}\sum_{k=1}^{N} C_{k-1}+\frac{1}{N}\sum_{k=1}^{N} C_{N-k}$$

Nun ist $C_0+C_1 \ldots +C_{N-1} = C_{N-1}+ C_{N-2} \ldots + C_0$:

$$C_N = N+\frac{2}{N}\sum_{k=1}^{N} C_{k-1} \text{ ; } N*C_N \text{ ist}$$

$$N*C_N = N^2 +2\sum_{k=1}^{N} C_{k-1}$$

Wir bilden nun die Differenz $N*C_N-(N-1)\ C_{N-1}$:

$$N*C_N-(N-1)\ C_{N-1} = N^2+2\sum_{k=1}^{N} C_{k-1}-(N-1)^2-2\sum_{k=1}^{N-1} C_{k-1}$$

$$= N^2+2C_{N-1}-(N-1)^2$$

$$= 2C_{N-1}+2N+1$$

$$N*C_N = (N+1)\ 2C_{N-1}+2N+1$$

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N}+\frac{2N+1}{N(N+1)}$$

Eine Abschätzung (keine genaue Lösung) über die N-te harmonische Zahl oder über ein Integral ergibt nunmehr

$$C_N \approx 2 \ln N$$
$$\approx 1{,}386 \cdot \log_2 N$$

Die mittlerer Pfadlänge in einem durch zufälliges Einfügen gewachsenen Binärbaum ist also

$$C_{Navg} \approx 1{,}386 \log_2 N.$$

Mittlere Suchpfadlänge im vollständigen Binärbaum

Ein vollständiger Binärbaum der Höhe *h* hat $N = 2^h-1$ Knoten. Die mittlere Suchpfadlänge ist wieder die innere Pfadlänge dividiert durch N. Es kommen im Baum vor:

1x 1 Vergleich
2x 2 Vergleiche
4x 3 Vergleiche
8x 4 Vergleiche usw.

Insgesamt ist die mittlere Pfadlänge also:

$$C_{Nmin} = \frac{1}{N} \sum_{i=0}^{h-1} [(i+1)] * 2^i \quad ; \; N = 2^h-1:$$

$$= \frac{1}{2^h - 1} [(h-1) * 2^{h+1}]$$

Wegen $h = \log_2 (N+1)$ ist also

$$C_{Nmin} (N) = \frac{1}{2^{h-1}} * (h-1) (2^h-1)+h]$$

$$= \log_2 (N+1) + \frac{\log_2(N+1)}{N} + 1$$

$$C_{Nmin} = O(\log_2 (N))$$

Vergleich "natürlicher" Binärbaum und vollständiger Binärbaum

Die mittlere Suchpfadlänge im vollständigen Binärbaum ist um ca. 38% kürzer als im zufällig gewachsenen Binärbaum.

"Entartete" Binärbäume können noch wesentlich längere Suchpfade haben. Es sind dann O(N) Vergleiche erforderlich.

Problem 1: Es ist nicht immer $N = 2^i - 1$, daher kann nicht immer ein vollständiger Binärbaum aufgebaut werden.

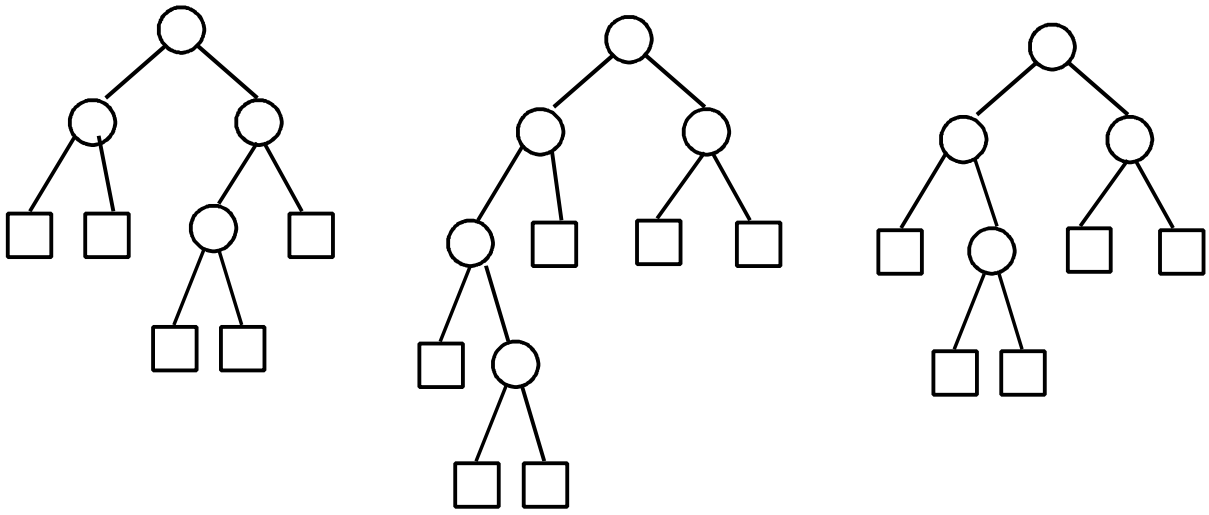
Problem 2: Vollständige Balancierung erfordert **erheblichen** Aufwand bei Änderungsoperationen.

AVL-Bäume (höhenbalancierte Binärbäume)

Vorgeschlagen von Adelson-Velskii und Landis (1962).

Definition

Ein binärer Suchbaum ist *AVL-ausgeglichen* oder *höhenbalanciert* kurz: ein AVL-Baum, wenn für jeden Knoten *p* des Baumes gilt, daß sich die Höhe des linken Teilbaumes von der Höhe des rechten Teilbaumes von *p* höchstens um 1 unterscheidet.

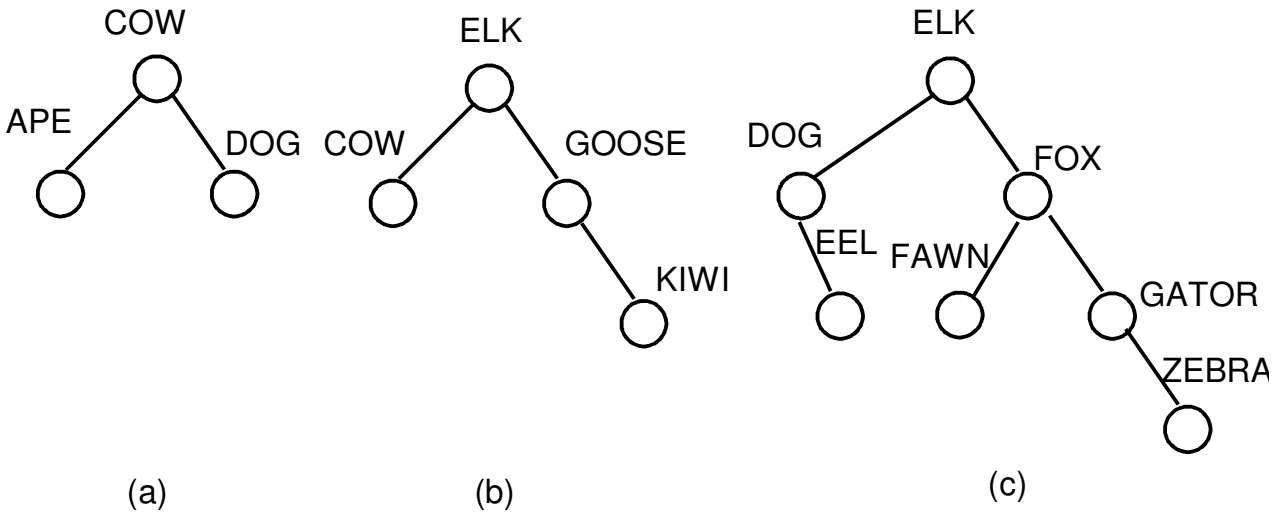


(a) (b) (c)

(a) und (c) sind höhenbalanciert, (b) nicht.

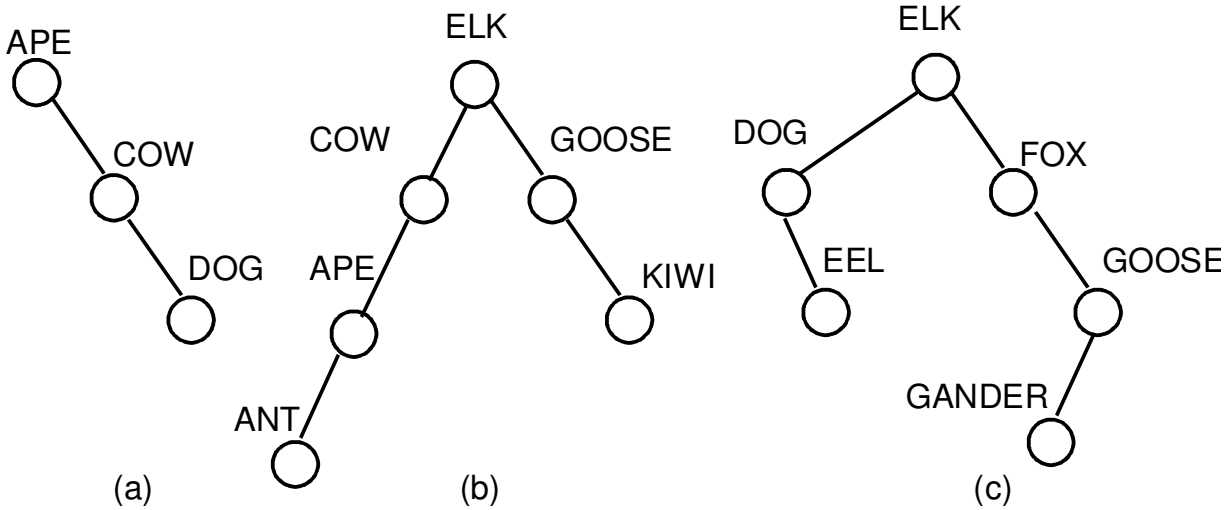
Man kann formal beweisen, daß AVL-Bäume mittlere Suchpfadlängen $O(\log_2 N)$ haben, also nicht entartet sein können.

Weitere Beispiele für höhenbalancierte und nicht-höhenbalancierte Binärbäume



(a) (b) (c)

Bounded-Balanced (BB) Trees



(a) (b) (c)

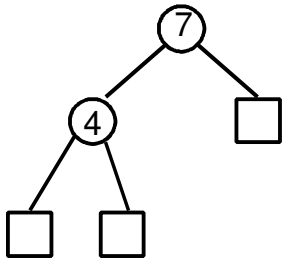
Suchen in AVL-Bäumen

Das Suchen in einem AVL-Baum erfolgt wie in jedem anderen binären Suchbaum. Der Aufwand ist $O(\log_2 N)$.

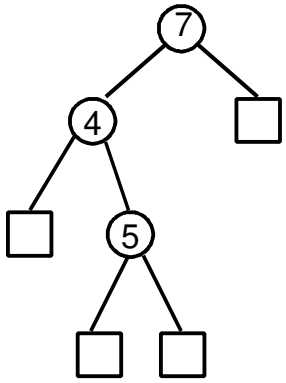
Einfügen in AVL-Bäumen

Das Einfügen eines neuen Schlüssels erfolgt zunächst wie in jedem anderen Binärbaum, also als Blatt an der Stelle, wo der Suchpfad endet. Aber: Möglicherweise ist der Baum dann nicht mehr höhenbalanciert.

Beispiel



Einfügen des Schlüssels 5.



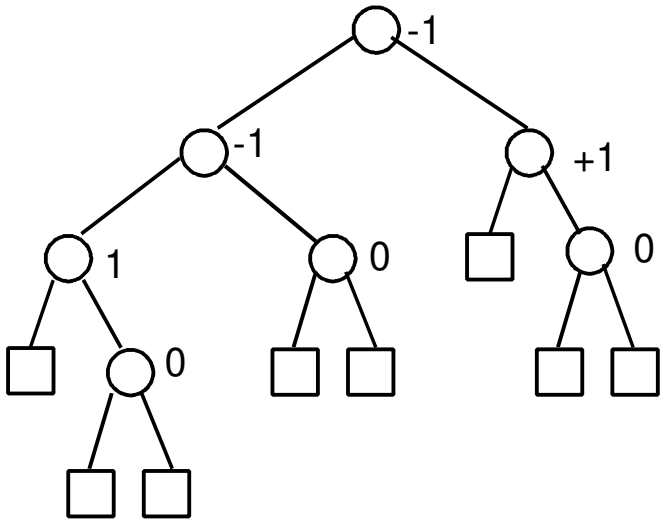
Kein AVL-Baum mehr!

Der Balancefaktor

Man könnte vermuten, daß man zur Prüfung der Höhenbedingung an einem Knoten im Baum die Höhen der Teilbäume des Knotens kennen muß. Das ist jedoch glücklicherweise nicht der Fall. Es genügt, an jedem inneren Knoten *p* den sogenannten *Balancefaktor bal(p)* mitzuführen, der wie folgt definiert ist:

bal(p) = Höhe des rechten Teilbaumes von *p*
- Höhe des linken Teilbaumes von *p*.

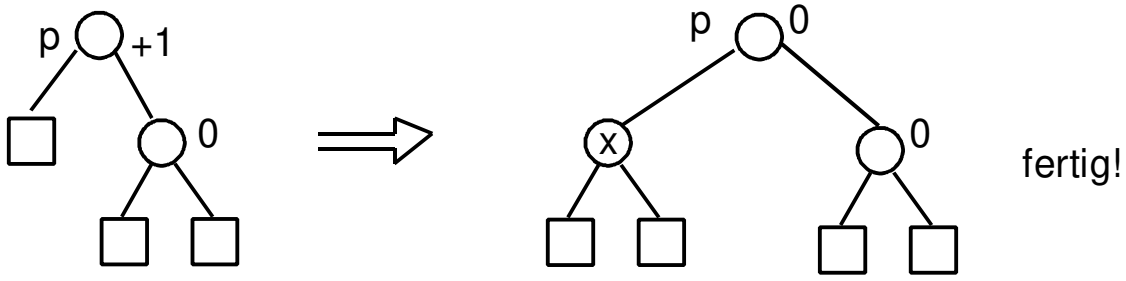
AVL-Bäume sind offenbar gerade dadurch charakterisiert, daß für jeden inneren Knoten *p* gilt: *bal(p)* $\in$ {-1,0,+1}. Die folgende Abbildung zeigt einen AVL-Baum mit rechts an die Knoten geschriebenen Balancefaktoren.



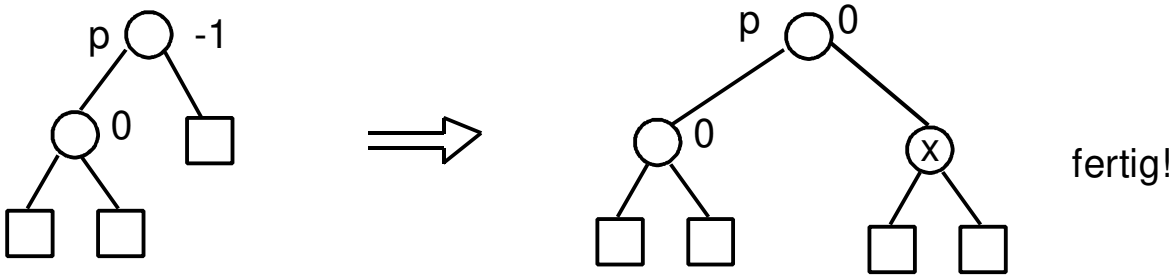
Balancierung von AVL-Bäumen durch Rotation

Sonst sei *p* der Vater des Blattes, bei dem die Suche endet. Drei Fälle sind möglich:

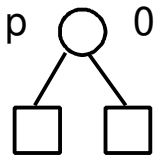
Fall 1 [*bal(p)* = +1]



Fall 2 [*bal(p)* = -1]



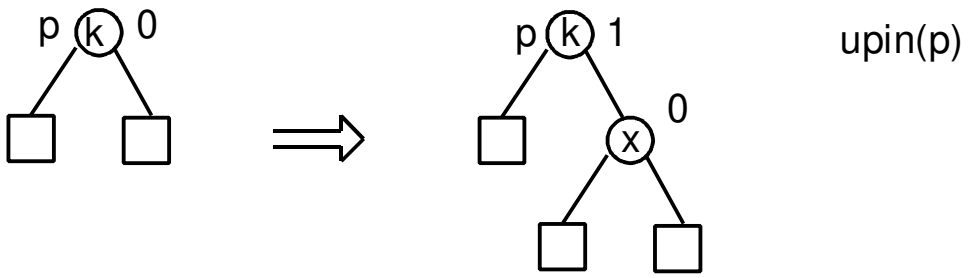
Fall 3 [*bal(p)* =0]



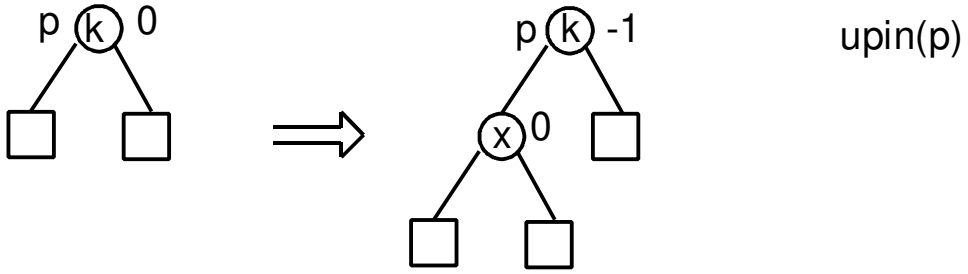
Durch Einfügen eines neuen Knotens als rechten oder linken Sohn von *p* wird *p* ein Knoten mit Balancefaktor -1 oder +1, und die Höhe des Teilbaumes mit Wurzel *p* wächst um 1. Wir rufen daher eine Prozedur *upin (p)* für den Knoten

p auf, die den Suchpfad zurückläuft, die Balancefaktoren prüft, gegebenenfalls adjustiert und Umstrukturierungen (sogenannte Rotationen oder Doppelrotationen) vornimmt, die sicherstellen, daß für alle Knoten die Höhendifferenzen der jeweils zugehörigen Teilbäume wieder höchstens 1 sind. Also:

Fall 3.1 [*bal(p)* = 0 und einzufügender Schlüssel *x* > Schlüssel *k* von *p*]



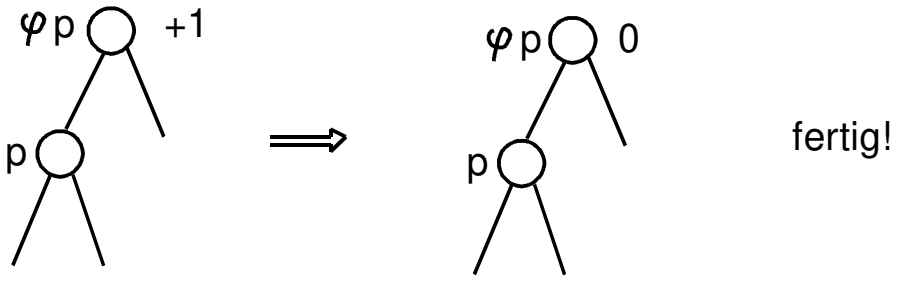
Fall 3.2 [*bal(p)* = 0 und einzufügender Schlüssel *x* < Schlüssel *k* von *p*]



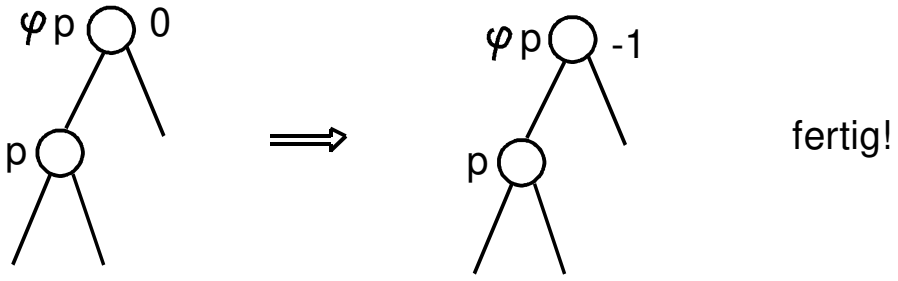
Wir erklären jetzt die Prozedur *upin*. Wenn *upin(p)* aufgerufen wird, so ist *bal(p)* ∈ {+1, -1} und die Höhe des Teilbaumes mit Wurzel *p* ist um 1 gewachsen. Wir müssen darauf achten, daß diese Invariante vor jedem rekursiven Aufruf von *upin* gilt; *upin(p)* bricht ab, falls *p* keinen Vater hat, d.h. wenn *p* die Wurzel des Baumes ist. Wir unterscheiden zwei Fälle, je nachdem ob φ*p*

Fall 1 [*p* ist linker Sohn seines Vaters φ*p*]

Fall 1.1 [*bal*(φ*p*) =+1]

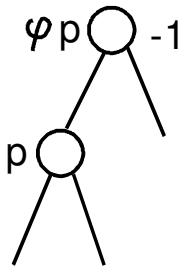


Fall 1.2 [*bal(φp)* = 0]



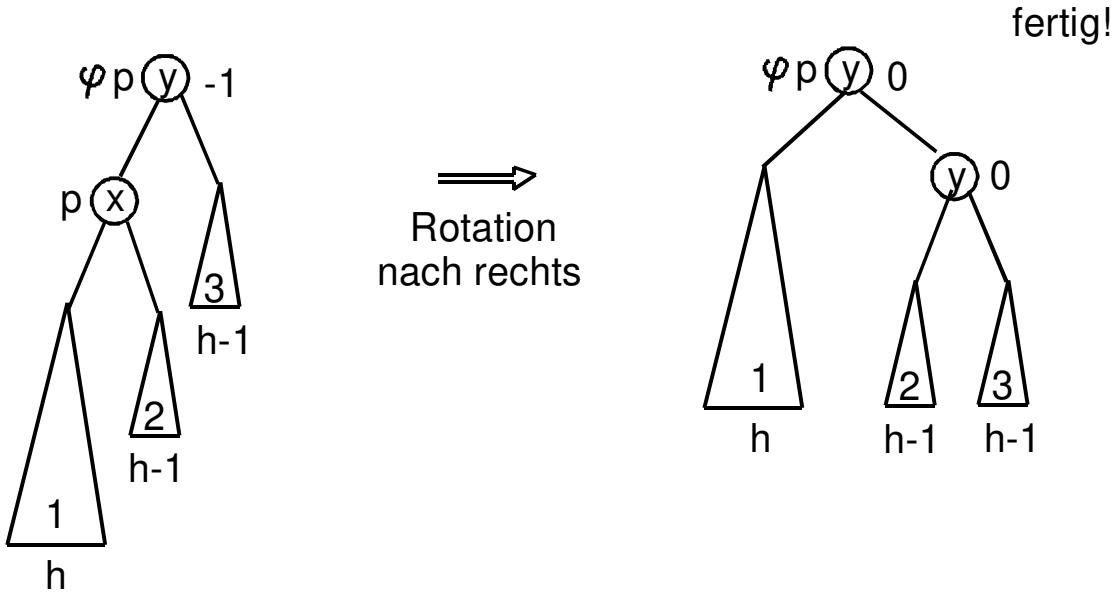
Man beachte, daß vor dem rekursiven Aufruf von *upin* die Invariante gilt.

Fall 1.3 [*bal(φp)* = -1]



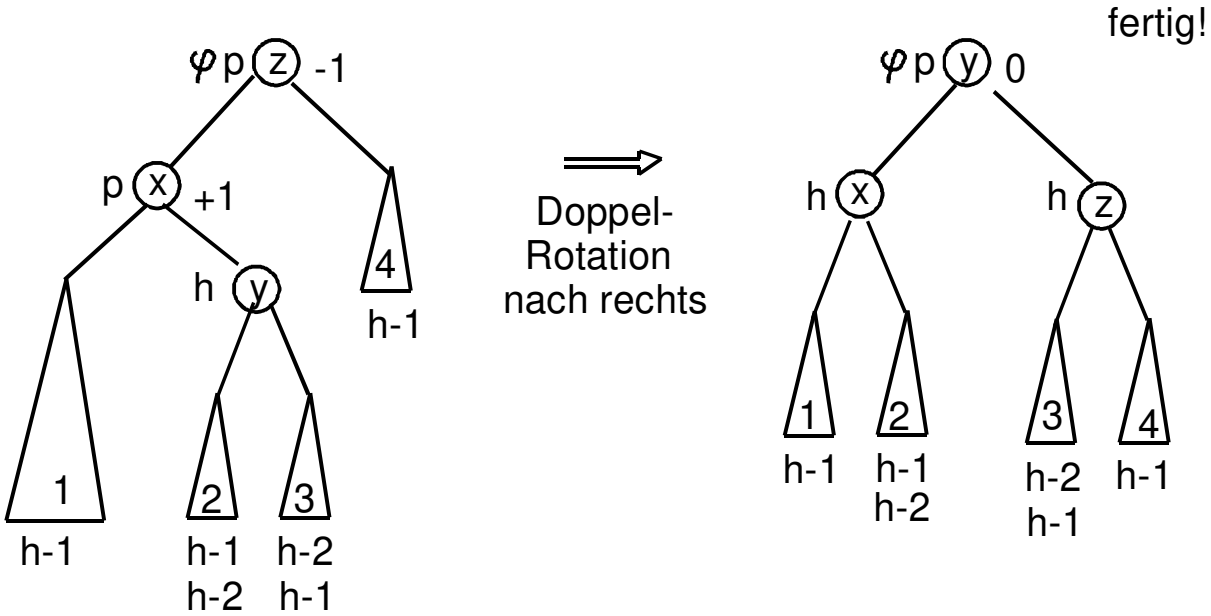
Wir müssen also umstrukturieren und unterscheiden dazu zwei Fälle, je nachdem, ob *bal(p)* = +1 oder *bal(p)* = -1 ist. (Wegen der Invariante ist *bal(p)* =0 nicht möglich!)

Fall 1.3.1 [*bal(p)* = -1]



Man beachte: Nach Voraussetzung ist die Höhe des Teilbaumes mit Wurzel *p* um 1 gewachsen und der linke Teilbaum von *p* um 1 höher als der rechte. Eine *Rotation nach rechts* bringt den Baum bei φ*p* wieder in die Balance.

Fall 1.3.2[*bal(p)* = +1]



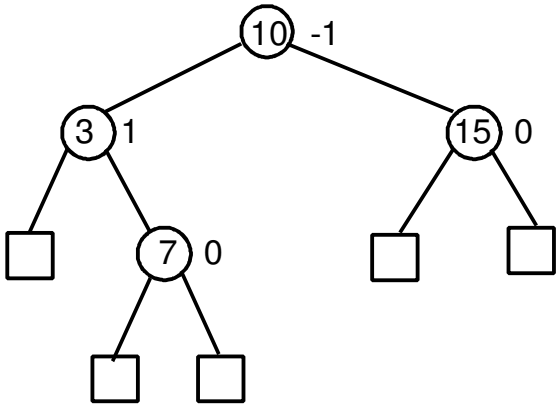
Man beachte: Die einzig möglichen Höhenkombinationen für die Teilbäume 2 und 3 sind (h - 1, h - 2) und (h - 2,h - 1). Sie können nicht gleiche Höhe haben. Denn auf Grund der Invarianten ist der Teilbaum mit Wurzel p in der Höhe um 1 gewachsen und wegen der Annahme von Fall 1.3.2 ist der rechte Teilbaum von p um 1 höher als sein linker. Eine *Doppelrotation*, d.h. zunächst eine Rotation nach links bei p und dann eine Rotation nach rechts bei φp, stellt die AVL-Ausgeglichenheit bei φp wieder her. Eine weitere Umstrukturierung ist nicht nötig, da der Teilbaum mit Wurzel φp in der Höhe nicht wächst.

Fall 2 [p ist rechter Sohn seines Vaters φp]

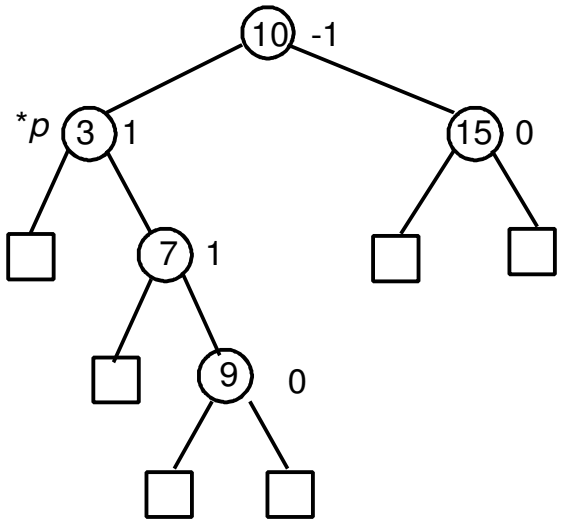
In diesem Fall geht man völlig analog vor und gleicht den Baum, wenn nötig, durch eine Rotation nach links bzw. eine Doppelrotation rechts-links bei φp wieder aus. Zur Veranschaulichung der Rotation nach links liest man die im Fall 1.3.1 gezeigt Abbildung von rechts nach links. Die Doppelrotation rechts-links erhält man aus der im Fall 1.3.2 gezeigten Figur durch Vertauschen der linken und rechten Teilbäume von p und φp.

AVL-Bäume, Beispiel für eine Rotation

Gegeben:

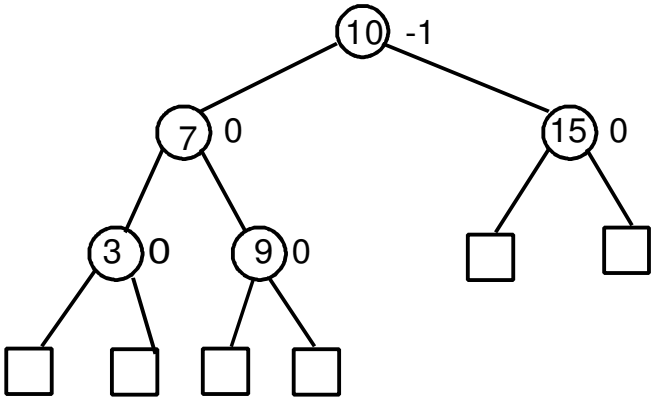


infügen des Schlüssels 9:

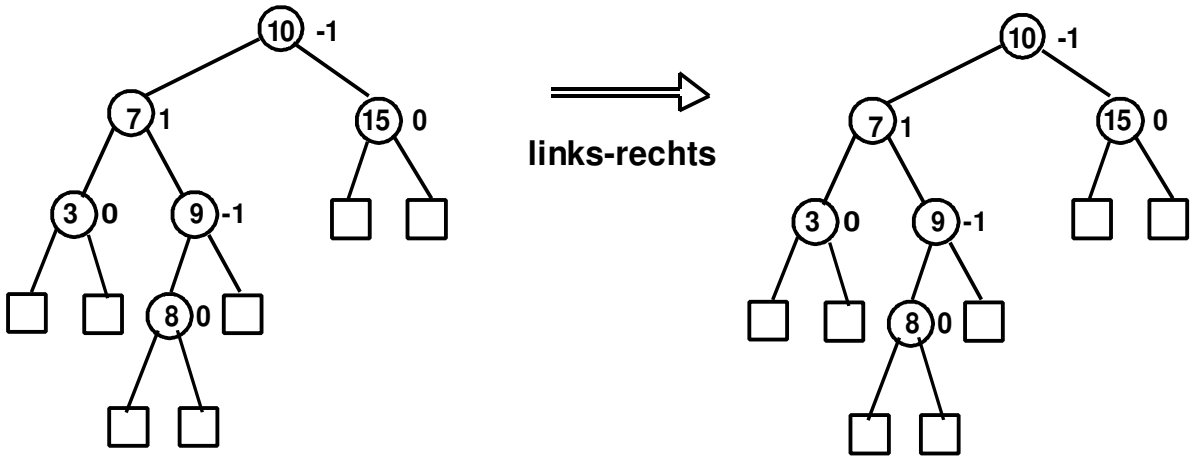


(alte Balancefactoren!)

Der Teilbaum mit Wurzel *p* ist nicht mehr höhenbalanciert. Rotation nach links bei *p*:



Einfügen des Schlüssels 8 erfordert eine Doppelrotation links-rechts:



Ein Aufruf der Prozedur *upin* kann schlimmstenfalls für alle Knoten auf dem Suchpfad von der Einfügestelle zurück zur Wurzel erforderlich sein. In jedem Fall wird aber höchstens eine Rotation oder Doppelrotation durchgeführt.

Die Umstrukturierung einschließlich der Adjustierung der Balancefaktoren ist also beendet und die AVL-Ausgeglichenheit wiederhergestellt. Damit ist klar, daß das Einfügen eines neuen Schlüssels in einen AVL-Baum mit N Schlüsseln in O(logN) Schritten ausführbar ist.

Löschen eines Schlüssels aus einem AVL-Baum

- Suchen des Schlüssels, wie üblich
- Entfernen des Knotens gemäß Algorithmus für alle Binärbäume
- Re-Balancierung analog zum Verfahren beim Einfügen

Gewichtsbalancierte Bäume

Gewicht eines Baumes: Anzahl der Knoten

Definition

Ein Baum heißt *gewichtsbalanciert*, wenn für jeden inneren Knoten gilt, daß die Gewichte des linken Teilbaums und des rechten Teilbaums sich um höchstens einen bestimmten Faktor unterscheiden.

Genauer:

Ist T ein Baum mit W(T) Blättern, dessen linker Teilbaum T_l W (T_l) Blätter hat, so nennt man den Quotienten

$$\rho(T) = \frac{W(T_l)}{W(T)}$$

die *Wurzelbalance* von T. Man fordert nun, daß die Wurzelbalance für jeden Teilbaum innerhalb bestimmter Grenzen liegen muß. Ist α eine Zahl mit $0 \leq \alpha \leq ½$, so heißt ein binärer Suchbaum T *von beschränkter Balance* α oder *gewichtsbalanciert mit Balance* α oder kurz ein BB[α]-Baum, wenn für jeden Teilbaum T' von T gilt:

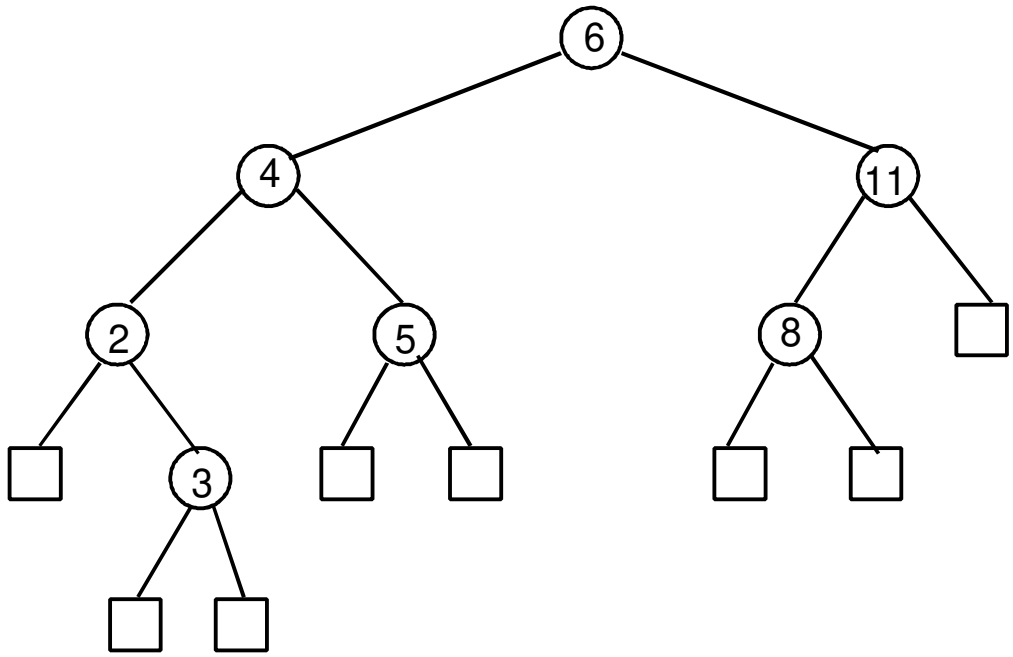
$$\alpha \leq \rho(T') \leq (1-\alpha) \qquad (*)$$

Anmerkung: Anzahl der Blätter = Anzahl der binären inneren Knoten +1

Also: $\alpha = ½$ bedeutet, daß genau die Hälfte aller Blätter jeweils im linken Teilbaum sein muß.

Gewichtsbalancierte Bäume, Beispiel

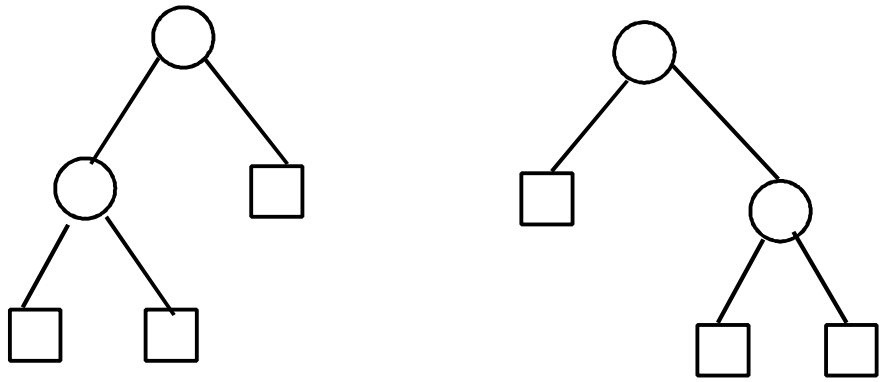
α = ¼, alle Wurzelbalancen zwischen ¼ und ¾.



Wurzelbalancen	
Knoten mit Schlüssel	Balance
6	5/8
4	3/5
11	2/3
2	1/3
5	1/2
3	1/2
8	1/2

Der Parameter α

Je näher α bei ½ liegt, desto ausgeglichener ist der Baum. Aber: $\alpha = ½$ ist nicht sinnvoll, da sich für viele Knotenzahlen N dann überhaupt kein gewichtsbalancierter Baum mehr finden läßt.
So gibt es beispielsweise nur zwei Suchbäume mit zwei inneren Knoten, wie in der Abbildung dargestellt wird. Die Wurzelbalance des linken Baumes ist 2/3 und die des rechten ist 1/3. Beide Bäume liegen in BB[1/3], aber BB[1/2] enthält keinen Baum mit 2 inneren Knoten.



Man kann zeigen: Wählt man

$$\alpha \in \left[\frac{1}{4}\ , 1 - \frac{\sqrt{2}}{2}\right],$$

so gibt es für jedes N einen Baum.

Suchen, Einfügen und Entfernen von Schlüsseln

Analog zu den Operationen in AVL-Bäumen:

Suchen wie in jedem Binärbaum

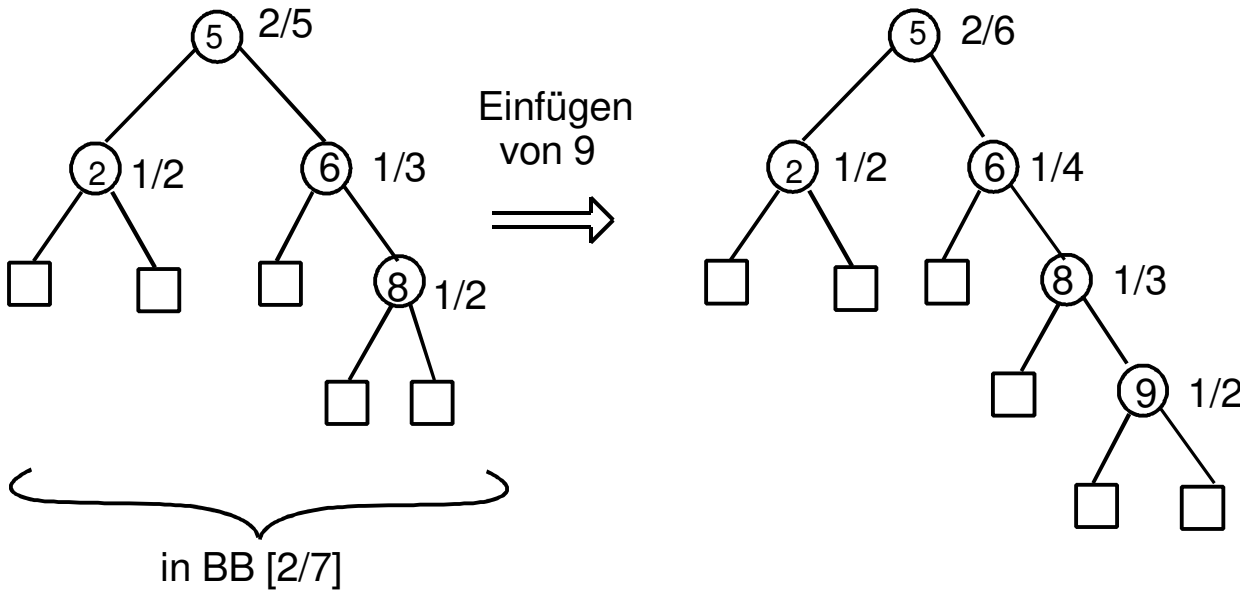
Einfügen durch Suchen des neuen Schlüssels im Baum; Suche endet bei einem Blatt; dort einfügen. Die Gewichtsbalance aller Knoten auf dem Suchpfad hat sich geändert; Zurücklaufen auf dem Suchpfad; wo die Gewichtsbalance $\notin [\alpha, 1-\alpha]$, Rotation oder Doppelrotation zur Wiederherstellung der Balance.

Löschen durch Suchen nach dem Schlüssel; Entfernen aus dem Baum wie beim natürlichen Binärbaum; Wiederherstellen der Balance durch Rotation bzw. Doppelrotation.

Einfügen in einen gewichtsbalancierten Baum, Beispiel

Es sei $\alpha = 2/7$, also

$$2/7 \leq \rho(T_i) \leq 5/7$$



Beim Knoten p: Balance verletzt Rotation nach links bei p.

Gewichtsbalancierte Binärbäume, Komplexität

Behauptung: Die Höhe h eines gewichtsbalancierten Baumes mit Parameter α ist logarithmisch in der Anzahl der Blätter (oder Knoten) beschränkt.

Beweisidee:

Gewichtsbalancierte Bäume sind dadurch charakterisiert, daß man beim Hinabsteigen von einem Knoten p zu einem seiner Söhne stets einen Mindestsbruchteil der Blätter verliert, der durch den Balancefaktor α bestimmt ist.

Beweis:
Genauer: Ist p ein Knoten mit linkem Sohn p_l und rechtem Sohn p_r , so folgt aus (*) und (**)):

(i) $W(p_l) \leq (1-\alpha)W(p)$
(ii) $W(p_r) \leq (1-\alpha)W(p)$

Sei nun ein gewichtsbalancierter Baum T aus BB[α] mit Höhe h gegeben. Wir betrachten einen Pfad maximaler Länge von der Wurzel zu einem Blatt. Seien $p_1, p_2, ...p_h$ die (inneren) Knoten auf diesem Pfad. Der Knoten p_1 ist also die Wurzel und p_h ist ein Knoten, dessen beide söhne Blätter sind. Daher ist

$$W(T) = W(p_1) \text{ und } W(p_h) = 2.$$

Wegen (i) und (ii) gilt

$$W(p_2) \leq (1-\alpha)W(p_1)$$
$$W(p_3) \leq (1-\alpha)W(p_2)$$

$$W(p_h) \leq (1-\alpha)W(p_{h-1})$$

Also

$$2 \leq (1-\alpha)^{h-1} \cdot W(p_1) = (1-\alpha)^{h-1} \cdot N,$$

wenn $N = W(p_1)$ die Anzahl der Blätter des Baumes T bezeichnet. Durch Logarithmieren dieser Ungleichung erhält man

$$1 \leq (h-1)\log_2(1-\alpha) + \log_2 N,$$

also

$$h-1 \leq \frac{\log_2 N - 1}{-\log_2(1-\alpha)} = O(\log N).$$

Gewichtsbalancierte Bäume, Kommentare

Alle Operationen O(logN); im Verhalten ähnlich wie höhenbalancierte Bäume.

Geschickte Wahl des Parameters α:
Je näher an 1/2, desto balancierter ist der Baum, desto kürzer der mittlere Suchpfad, aber desto höher der Aufwand beim Einfügen und Löschen.

In der Praxis kann man häufig davon ausgehen, daß die Schlüssel zufällig eingefügt/gelöscht werden. Dann erübrigen sich jegliche Balancierungsmaßnahmen. Bei kleinem N ebenfalls.