

3 Suchen in Mengen

3.1 Problemstellung

Gegeben:

Eine Menge von Records (Elementen), von denen jeder aus einer Schlüssel-Komponente und weiteren Komponenten besteht. In der Regel werden Duplikate ausgeschlossen, wobei sich der Begriff Duplikat beziehen kann:

- auf den Schlüssel;
- auf den vollen Record (exakt: keine Menge mehr).

Typische Aufgabe:

- Finde zu einem vorgegebenen Schlüsselwert den Record und führe gegebenenfalls eine Operation aus.

Damit sind (u.U.) alle Operationen auf Mengen wünschenswert.

Die Darstellung als Menge und deren Verarbeitung kommt oft vor (Datenbanken!).

Im folgenden betrachten wir primär das *Dictionary-Problem* (Wörterbuch-Problem).

Notation: etwa wie [Mehlhorn]

Universum: $U :=$ Menge aller möglichen Schlüssel

Menge: $S \subseteq U$

Wörterbuch-Operationen:

Search(x, S): Falls $x \in S$, liefere den vollen zu x gehörigen Record (oder Information oder Adresse),
sonst Meldung: „ $x \notin S$ “.

Insert(x, S): Füge Element x zur Menge S hinzu: $S := S \cup \{x\}$
(Fehlermeldung falls $x \in S$).

Delete(x, S): Entferne Element x aus der Menge S : $S := S \setminus \{x\}$
(Fehlermeldung falls $x \notin S$).

Hinweis: Die Terminologie ist nicht standardisiert:

z.B.: Search = Member = Contains = Access

| |
|----------------------|
| Weitere Operationen: |
|----------------------|

| | |
|--------------------|--|
| Order(k, S): | Finde das k -te Element in der geordneten Menge S . |
| ListOrder(S): | Produziere eine geordnete Liste der Elemente der Menge S . |
| Enumerate(S): | Zähle alle Elemente der Menge S auf. |
| FindMin(S): | $:=$ Order(1, S) |
| Initialize(S): | Initialisiere die Darstellung, d.h. $S := \emptyset$. |

| |
|--|
| Weitere Operationen auf Mengen, die prinzipiell in Frage kommen: |
|--|

Seien S, A Teilmengen von U :

| | |
|---------------|----------------------|
| Union (Join): | $S := S \cup A$ |
| Intersection: | $S := S \cap A$ |
| Difference: | $S := S \setminus A$ |

Seien A_k Teilmengen von U , $k = 1, \dots, K$:

Find(x, A_k): Finde die Menge A_k , zu der das Element x gehört.

Bemerkung: Bisher wird nicht berücksichtigt, ob das Universum groß oder klein ist.

Beispiele [Mehlhorn, Seite 97]:

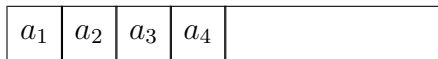
- Symboltabelle (Compiler): 6 Character (z.B. Buchstaben + Ziffern):
 $|U| = (26 + 10)^6 = 2.2 * 10^9$
- Autorenverzeichnis einer Bibliothek in lexikograph./alphabetischer Ordnung
 $|U| =$ ähnliche Größenordnung
- Konten einer Bank (6-stellige Konto-Nummer, 50% davon tatsächlich genutzt)
 $|U| = 10^6$. Hier sind die Größe des Universums und die Menge der benutzten Schlüssel gleich groß.

3.2 Einfache Implementierungen

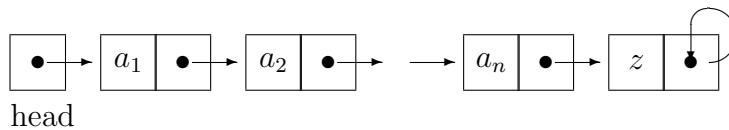
3.2.1 Ungeordnete Arrays und Listen

Darstellung: vgl. Abschnitt 1.3.2

- Liste im Array



- verkettete Liste



3.2.2 Vergleichsbasierte Methoden

Voraussetzung: Existenz einer Vergleichsoperation. Die Annahme einer Ordnung ist keine wirkliche Einschränkung: Es gibt immer eine Ordnung auf der internen Darstellung der Elemente von U [Mehlhorn, Seite 139].

Wir betrachten Suchverfahren für Daten im geordneten Array S mit $S[i] < S[i + 1]$ für $1 \leq i \leq n - 1$.

Wir unterscheiden:

- sequentielle oder lineare Suche (auch ohne Ordnung möglich);
- Binärsuche;
- Interpolationssuche.

Allgemeines Programmschema:

```
S : array [0..n+1] of element;
S[0] =  $-\infty$ ;
S[n+1] =  $+\infty$ ;
```

```
procedure Search(a,S);
  var low, high: element;
begin
  low := 1; high := n;
```

```

next := an integer ∈ [low..high]
while (a ≠ S[next]) and (low < high) do
  begin
    if a < S[next]
    then high := next - 1
    else low := next + 1;
    next := an integer ∈ [low..high]
  end
if a = S[next]
then output "a wurde an Position " next " gefunden.";
else output "a wurde nicht gefunden!";
return
end;

```

Varianten der Anweisung $next := an\ integer \in [low..high]$:

1. **Lineare Suche** (sequentiell): $next := low$
2. **Binärsuche** (sukzessives Halbieren):

$$next := \left\lceil \frac{high + low}{2} \right\rceil$$

3. **Interpolationssuche** (lineare Interpolation):

$$next := (low - 1) + \left\lceil (high - low + 1) \cdot \frac{a - S[low - 1]}{S[high + 1] - S[low - 1]} \right\rceil$$

Komplexität (Zahl der Schlüsselvergleiche $T(n)$):

1. **Lineare Suche:** $O(n)$
zwischen 1 und n Operationen:
 - a) ungeordneter Liste/Array:
 - $n/2$: erfolgreich
 - n : erfolglos (jedes Element muß geprüft werden)
 - b) geordnete Liste/Array:
 - $n/2$: erfolgreich
 - $n/2$: erfolglos

2. **Binärsuche:** $O(\lg n)$

Rekursionsgleichung:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$$\Rightarrow T(n) = \lg n + 1$$

immer weniger als $\lg n + 1$ Vergleiche3. **Interpolationssuche** (ohne Beweis und Erläuterung):

- average-case: $\lg(\lg n) + 1$
- worst-case: $O(n)$ (Entartung in lineare Suche)

Korrektheit des ProgrammsDas folgende Prädikat P ist eine Invariante der **while**-Schleife:

$$P \equiv (a \in S \Rightarrow a \in S[\text{low}..\text{high}]) \wedge (\text{low} \leq \text{high} \Rightarrow \text{low} \leq \text{next} \leq \text{high})$$

Der Beweis der Korrektheit des Programms erfolgt in drei Schritten:

- A) Wir zeigen, daß das Prädikat P vor Eintritt in die Schleife gilt und auch im Rumpf der Schleife gültig bleibt.
- B) Wir zeigen, daß die Schleife terminiert.
- C) Also gilt P auch bei verlassen der Schleife. Aus der Gültigkeit von P beim Verlassen der Schleife wird dann die Korrektheit des Programms nachgewiesen.

Beweisschritte im einzelnen:

- A) vor der Schleife: P ist erfüllt
in der Schleife:
es gilt $a \neq S[\text{next}]$ und also
 - a) entweder $a < S[\text{next}]$
 $\Rightarrow a \notin S[\text{next}..\text{high}]$ (wegen Ordnung!)
und somit: $a \in S \Rightarrow a \in S[\text{low}..\text{next} - 1]$
 - b) oder $a > S[\text{next}]$: analoge Behandlung

Falls $low \leq high$ gilt, folgt:
 $low \leq next \leq high$

B) Schleife terminiert:

In jedem Durchlauf wird $high - low$ mindestens um 1 verringert.

C) Falls also die Schleife terminiert, gilt P und

a) entweder $a = S[next]$: Suche erfolgreich

b) oder $low \geq high$

Sei nun $a \neq S[next]$. Weil P gilt, folgt aus $a \in S[1..n]$, daß $a \in S[low..high]$.

Nun $low \geq high$:

i. Falls $high < low$: dann ist $a \notin S[1..n]$

ii. Falls $high = low$: dann ist $next = high$ wegen P

(insbesondere wegen $a \neq S[next]$) und somit $a \notin S[1..n]$

in beiden Fällen: Suche erfolglos

3.2.3 Bitvektordarstellung (Kleines Universum)

Annahme:

$N = |U|$ = vorgegebene maximale Anzahl von Elementen

$S \subset U = \{0, 1, \dots, N - 1\}$

Methode: Schlüssel = Index in Array

Bitvektor (auch: charakteristische Funktion; Array of Bits):

- $Bit[i] = \text{false} : i \notin S$
- $Bit[i] = \text{true} : i \in S$

Vergleich der Komplexitäten (O-Komplexität):

(falls möglich: Binärsuche)

$N = |U|$ = die Kardinalität des Universums U , und

$n = |S|$ die Kardinalität der Teilmenge $S \subseteq U$ der tatsächlich vorhandenen Elemente.

| Methode | Zeitkomplexität | | | Platzkomplexität |
|--------------------|------------------|--------|--------|------------------|
| | Search | Insert | Delete | |
| ungeordnete Liste | n | 1^* | n | n |
| ungeordnetes Array | n | 1^* | n | N |
| geordnete Liste | n | n | n | n |
| geordnetes Array | $\text{ld } n^*$ | n | n | N |
| Bitvektor | 1 | 1 | 1 | N |

Anmerkungen:

- 1^* heißt: Erfordert Laufzeit $O(n)$ mit Duplikateneliminierung.
 $\text{ld } n^*$ heißt: Search ist mit Binärsuche implementiert.
- Bitvektor-Methode:
 - Operationen $O(1)$ gut, aber:
 - Initialize = $O(N)$
 - Platz = $O(N)$
- Ideal wäre: 3 Operationen mit Zeit $O(1)$ und Platz $O(n)$

3.2.4 Spezielle Array-Implementierung

[Mehlhorn, Seite 270]

Prinzip: Bitvektor-Darstellung mit zwei Hilfsarrays ohne $O(N)$ -Initialisierung

andere Namen:

- lazy initialization
- invertierte Liste
- array/stack-Methode

Deklarationen:

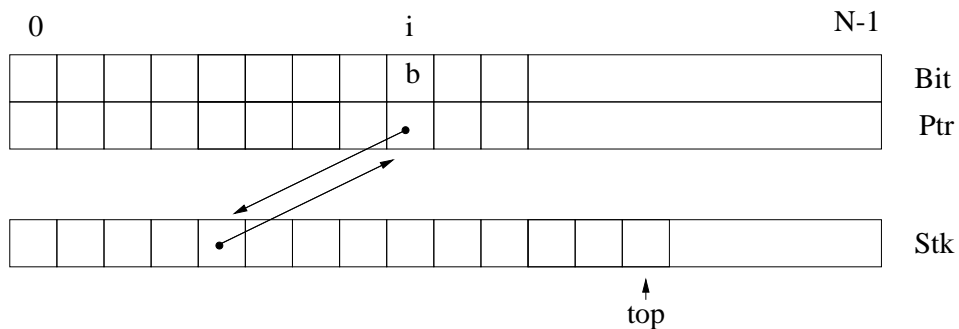
| | |
|---------------------------------------|---------------|
| Bit[0..N-1] : array of boolean | Bitvektor |
| Ptr[0..N-1] : array of integer | Pointer-Array |
| Stk[0..N-1] : array of integer | Stack-Array |

Das Konzept beruht auf der Invarianten: $i \in S$ genau dann, wenn

1. $Bit[i] = \text{true}$ und
2. $0 \leq Ptr[i] \leq top$ und
3. $Stk[Ptr[i]] = i$.

wobei anfangs: $top = -1$

Durch den Zähler top und den Backpointer in $Stk[.]$ wird die Initialisierung ersetzt.



Implementierung der Operationen

```
Globale Variablen: top : integer;
                   Bit[0..N-1] : array of boolean;
                   Ptr[0..N-1] : array of integer;
                   Stk[0..N-1] : array of integer;
```

```
procedure Initialize(S);
begin
  top := -1;
end;
```

```
function Search(i, S): boolean;
begin
  if (0 ≤ Ptr[i] ≤ top) and (Stk[Ptr[i]] = i)
  then return Bit[i];
  else return false ; (*i ∉ S never initialized.*)
end;
```

Anmerkung: Beim logischen **and** in der **if**-Anweisung sei nochmals auf die Problematik der Auswertung solcher Ausdrücke hingewiesen („Lazy Evaluation“).

Insert(i, S) und Delete(i, S) werden realisiert durch:

```

procedure Insert( $i, S$ );
begin
  if ( $0 \leq \text{Ptr}[i] \leq \text{top}$ ) and ( $\text{Stk}[\text{Ptr}[i]] = i$ )
    then  $\text{Bit}[i] := \text{true}$ ;
  else begin
     $\text{top} := \text{top} + 1$ ;
     $\text{Ptr}[i] := \text{top}$ ;
     $\text{Stk}[\text{top}] := i$ ;
     $\text{Bit}[i] := \text{true}$ ;
  end;
end;

procedure Delete( $i, S$ );
begin
   $\text{Bit}[i] := \text{false}$ ; (* no error check *)
end;

```

Anmerkungen:

- Die Variable top gibt die maximale Anzahl der jemals angesprochenen Elemente von $S \subseteq U$ an. Bei der Delete-Operation wird nichts zurückgesetzt außer $\text{Bit}[i]$.
- Die Information des Bitvektors $\text{Bit}[0..N - 1]$ könnte auch direkt als zusätzliche Information von jeweils einem Bit im Pointer-Array $\text{Ptr}[0..N - 1]$ gespeichert werden.
- Die Methode funktioniert auch für das Initialisieren großer Real-/Integer-Arrays in numerischen Rechnungen.

Komplexität

- Platzkomplexität: Drei Arrays der Länge N , also $O(N)$.
- Zeitkomplexität für die Standard-Operationen:
 - Initialize(S): $O(1)$
 - Search(i, S): $O(1)$
 - Insert(i, S): $O(1)$

- Delete(i, S): $O(1)$
- Enumerate(S): $O(top)$ mit $n \leq top \leq N, n := |S|, N := |U|$

3.3 Hashing

3.3.1 Begriffe und Unterscheidung

- offenes/geschlossenes Hashing
- Kollisionsstrategien:
 - lineares Sondieren
 - quadratisches Sondieren
 - Doppelhashing
- Hashfunktionen
- erweiterbares Hashing in Verbindung mit Hintergrundspeicher

Ausgangspunkt:

Bei BucketSort wurde aus dem Schlüssel direkt die Speicheradresse berechnet ebenso wie bei der Bitvektordarstellung einer Menge.

Hashing kann man als Erweiterung dieser Methode interpretieren, indem die Speicheradresse nicht mehr eindeutig umkehrbar (auf den Schlüssel) sein muß, und somit Mehrfachbelegungen der Speicheradresse zulässig sind ('Kollisionen').

Prinzip

Zur Verfügung stehen m Speicherplätze in einer *Hashtabelle* T :

var T: **array** [0.. $m - 1$] **of** element .

Dann wird ein Schlüssel $x \in \text{Universum } U = \{0, \dots, N - 1\}$ auf einen Speicherplatz in der Hashtabelle abgebildet. Dazu dient die *Hashfunktion* $h(x)$ (deutsch: Schlüsseltransformation, Streuspeicherung):

$$\begin{aligned} h : U &\rightarrow \{0, 1, \dots, m - 1\} \\ x &\rightarrow h(x) \end{aligned}$$

Nach dieser Adressberechnung kann das Element mit dem Schlüssel x an der Stelle $T[h(x)]$ gespeichert werden, falls dieser Platz noch frei ist.

Da in der Regel ($m \cong n$) $\ll N$ ist, kann es zu *Kollisionen* kommen, d.h.

$$h(x) = h(y) \text{ , für } x \neq y \text{ .}$$

x wird also nicht notwendigerweise in $T[h(x)]$ selbst gespeichert. Entweder enthält $T[h(x)]$ dann einen Verweis auf eine andere Adresse (offene Hashverfahren), oder mittels einer *Sondierfunktion* muß ein anderer Speicherplatz berechnet werden (geschlossene Hashverfahren).

Dementsprechend hat die Operation *Search* (x, S) dann zwei Teile. Zunächst muß $h(x)$ berechnet werden, dann ist x in $T[h(x)]$ zu suchen.

Beispiel: Symboltabelle für Compiler

Ein Anwendungsgebiet für Hashing sind Symboltabellen für Compiler. Das Universum U , nämlich die Menge aller Zeichenketten mit der (maximalen) Länge 20 (z.B. Namen) ist hier sehr groß: Selbst wenn man nur Buchstaben und Ziffern zuläßt, erhält man

$$|U| = (26 + 10)^{20} = 1.3 \cdot 10^{31}.$$

Somit ist keine umkehrbare Speicherfunktion realistisch.

3.3.2 Hashfunktionen

An die Hashfunktion $h(x)$ werden folgende Anforderungen gestellt:

- Die ganze Hashtabelle sollte abgedeckt werden, d.h. $h(x)$ ist surjektiv.
- $h(x)$ soll die Schlüssel x möglichst gleichmäßig über die Hashtabelle verteilen.
- Die Berechnung soll effizient, also nicht zu rechenaufwendig sein.

Nun wollen wir einige typische Hashfunktionen vorstellen [Güting, S. 109]. Es wird hierbei davon ausgegangen, daß für die Schlüssel x gilt: $x \in \mathbb{N}_0$. Fast alle in der Praxis vorkommenden Schlüssel lassen sich entsprechend umwandeln, z.B. durch Umsetzung von Buchstaben in Zahlen.

Divisions-Rest-Methode:

Sei m die Größe der Hashtabelle. Dann definiert man $h(x), x \in \mathbb{N}$, wie folgt:

$$h(x) = x \mod m.$$

Bewährt hat sich folgende Wahl für m :

- m Primzahl
- m teilt nicht $2^i \pm j$, wobei i, j kleine Zahlen $\in \mathbb{N}_0$ sind.

Diese Wahl gewährleistet eine surjektive und gleichmäßige Verteilung über die ganze Hashtabelle [Ottmann].

Die Divisions-Rest-Methode hat den Vorteil der einfachen Berechenbarkeit. Der Nachteil ist die Tendenz, daß aufeinanderfolgende Schlüssel auf aufeinanderfolgende Speicherplätze abgebildet werden. Das führt zu unerwünschtem Clustering, welches die Effizienz des Sondierens reduziert (s. Abschnitt 3.3.5).

Beispiel: Verteilen von Namen über eine Tabelle mit m Elementen [Gütting, Seite 97].

Die Zeichenketten $c_1 \dots c_k$ werden auf \mathbb{N}_0 abgebildet und auf die Hashtabelle verteilt:

$$h(c_1 \dots c_k) := \sum_{i=1}^k N(c_i) \pmod{m}$$

mit $N(A) = 1, N(B) = 2, \dots, N(Z) = 26$.

Zur Vereinfachung werden nur die ersten drei Buchstaben betrachtet:

$$h(c_1 c_2 c_3) := [N(c_1) + N(c_2) + N(c_3)] \pmod{m}.$$

Wir wählen $m = 17$; $S =$ deutsche Monatsnamen (ohne Umlaute).

| | |
|-----|-----------------|
| 0: | November |
| 1: | April, Dezember |
| 2: | Maerz |
| 3: | |
| 4: | |
| 5: | |
| 6: | Mai, September |
| 7: | |
| 8: | Januar |
| 9: | Juli |
| 10: | |
| 11: | Juni |
| 12: | August, Oktober |
| 13: | Februar |
| 14: | |
| 15: | |
| 16: | |

Beachte: Es gibt drei Kollisionen.

Mittel-Quadrat-Methode:

Die Mittel-Quadrat-Methode zielt darauf ab, auch nahe beieinanderliegende

Schlüssel auf die ganze Hashtabelle zu verteilen, um ein Clustering (siehe Abschnitt 3.3.5) aufeinanderfolgender Zahlen zu verhindern.

$$h(x) = \text{mittlerer Block von Ziffern von } x^2, x \in \{0, \dots, N-1\}.$$

Der mittlere Block hängt von allen Ziffern von x ab, und deswegen wird eine bessere Streuung erreicht.

Für $m = 100$ ergibt sich:

| x | $x \bmod 100$ | x^2 | $h(x)$ |
|-----|---------------|-------|--------|
| 127 | 27 | 16129 | 12 |
| 128 | 28 | 16384 | 38 |
| 129 | 29 | 16641 | 64 |

Es gibt noch weitere gebräuchliche Methoden, wie z.B. die *Multiplikative Methode*, auf die wir hier nicht näher eingehen wollen.

3.3.3 Wahrscheinlichkeit von Kollisionen

Das Hauptproblem beim Hashing besteht in der Behandlung von Kollisionen. Bevor wir verschiedene Methoden der Kollisionsbehandlung erörtern, wollen wir berechnen, wie häufig Kollisionen auftreten.

In der Mathematik gibt es ein analoges, unter dem Namen *Geburtstags-Paradoxon* bekanntes Problem:

Wie groß ist die Wahrscheinlichkeit, daß mindestens 2 von n Personen am gleichen Tag Geburtstag haben ($m = 365$) ?

Analogie zum Hashing:

$m = 365$ Tage \equiv Größe der Hashtabelle;

n Personen \equiv Anzahl der Elemente von $S \subset U$.

Annahme: Die Hashfunktion sei ideal, d.h. die Verteilung über die Hashtabelle sei gleichförmig.

Für die Wahrscheinlichkeit mindestens einer Kollision bei n Schlüsseln und einer m -elementigen Hashtabelle $Pr(Kol|n, m)$ gilt:

$$Pr(Kol|n, m) = 1 - Pr(NoKol|n, m).$$

Sei $p(i; m)$ die Wahrscheinlichkeit, daß der i -te Schlüssel ($i = 1, \dots, n$) auf einen freien

Platz abgebildet wird, wie alle Schlüssel zuvor auch. Es gilt

$$\begin{aligned}
 p(1; m) &= \frac{m-0}{m} = 1 - \frac{0}{m} && , \text{ weil } 0 \text{ Plätze belegt und } m-0 \text{ Plätze frei sind} \\
 p(2; m) &= \frac{m-1}{m} = 1 - \frac{1}{m} && , \text{ weil } 1 \text{ Platz belegt und } m-1 \text{ Plätze frei sind} \\
 &\vdots \\
 p(i; m) &= \frac{m-i+1}{m} = 1 - \frac{i-1}{m} && , \text{ weil } i-1 \text{ Plätze belegt und } m-i+1 \text{ Plätze frei sind}
 \end{aligned}$$

$Pr(NoKol|n, m)$ ist dann das Produkt der Wahrscheinlichkeiten $p(1; m), \dots, p(n; m)$.

$$\begin{aligned}
 Pr(NoKol|n, m) &= \prod_{i=1}^n p(i; m) \\
 &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right)
 \end{aligned}$$

Tabelle zum Geburtstagsproblem ($m = 365$):

| n | $Pr(Kol n, m)$ |
|----------|----------------|
| 10 | 0,11695 |
| 20 | 0,41144 |
| \vdots | \vdots |
| 22 | 0,47570 |
| 23 | 0,50730 |
| 24 | 0,53835 |
| \vdots | \vdots |
| 30 | 0,70632 |
| 40 | 0,89123 |
| 50 | 0,97037 |

Die Wahrscheinlichkeit, daß an mindestens einem Tag mindestens zwei Personen aus einer Gruppe von 23 Personen Geburtstag haben, beträgt also 50.7%.

Approximation

Frage: Wie muß m mit n wachsen, damit $Pr(Kol|n, m)$ und also auch $Pr(NoKol|n, m)$ konstant bleibt. Die obige Formel gibt auf diese Frage nur eine indirekte Antwort. Darum

die folgende Vereinfachung:

$$\begin{aligned} Pr(NoKol|n, m) &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \\ &= \exp \left[\sum_{i=0}^{n-1} \ln \left(1 - \frac{i}{m}\right) \right] \end{aligned}$$

Nun verwendet man: $\ln(1 + \varepsilon) \cong \varepsilon$ für $\varepsilon \ll 1$, d.h. $\frac{i}{m} < \frac{n}{m} \ll 1$ (siehe Abbildung 3.1).

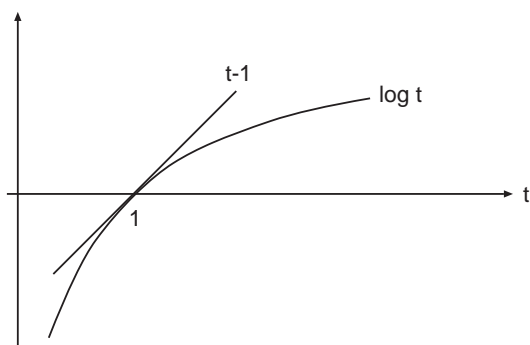


Abbildung 3.1: Eigenschaft des Logarithmus

$$\begin{aligned} Pr(NoKol|n, m) &\cong \exp \left[- \sum_{i=0}^{n-1} \frac{i}{m} \right] \\ &= \exp \left[- \frac{n(n-1)}{2m} \right] \\ &\cong \exp \left[- \frac{n^2}{2m} \right] \end{aligned}$$

Ergebnis: $Pr(NoKol|n, m)$ bleibt etwa konstant, wenn die Größe m der Hashtabelle quadratisch mit der Zahl der Elemente n wächst.

Übungsaufgaben:

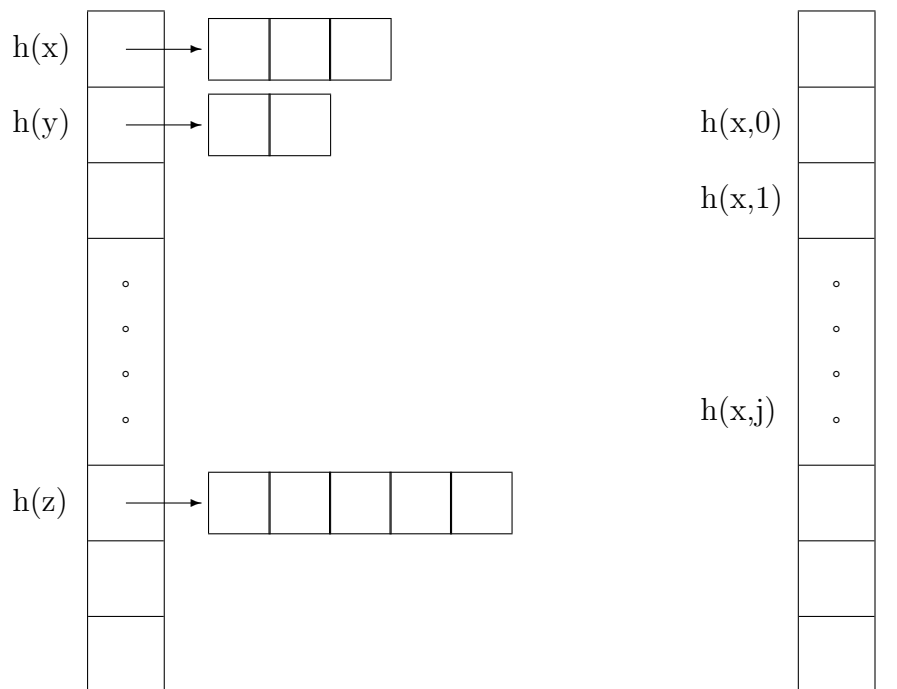
1. Die obige Approximation ist eine obere Schranke. Finden Sie eine enge untere Schranke.
2. Geben Sie eine gute Approximation an, in der nur „paarweise“ Kollisionen betrachtet werden, d.h. daß *genau* zwei Elemente auf eine Zelle abgebildet werden.

Terminologie

In den folgenden Abschnitten wollen wir uns mit den Vorgehensweisen bei Kollisionen beschäftigen. Da die Terminologie auf diesem Gebiet in der Literatur aber sehr uneinheitlich ist, wollen wir kurz definieren, welche wir benutzen.

Als *offene Hashverfahren* (*open hashing*) bezeichnen wir Verfahren, die mit dynamischem Speicher (verketteten Listen) arbeiten, und daher beliebig viele Schlüssel unter einem Hashtabellen-Eintrag unterbringen können.

Geschlossene Hashverfahren (*closed hashing*) hingegen können nur eine begrenzte Zahl von Schlüsseln (meistens nur einen) unter einem Eintrag unterbringen. Sie arbeiten mit einem Array und werden wegen des abgeschlossenen Speicherangebots als geschlossen bezeichnet. Sie müssen bei Kollisionen mittels Sondierungsfunktionen neue Adressen suchen.

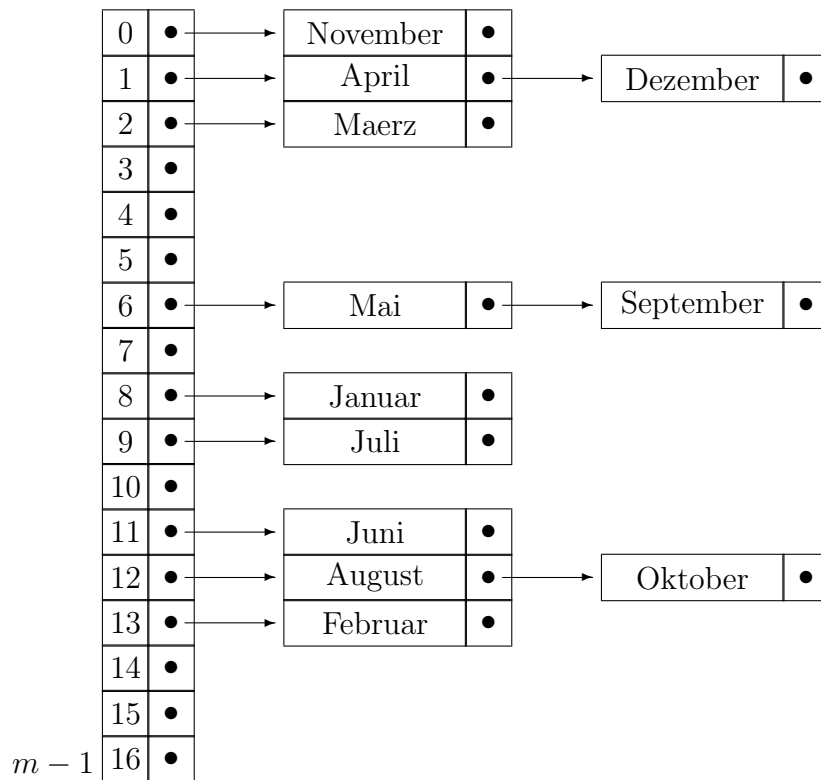


3.3.4 Open Hashing (Hashing mit Verkettung)

Jeder Behälter wird durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt. Ein Array von Zeigern verwaltet die Behälter.

```
var HashTable : array [0..m-1] of ^ListElem.
```

Für obiges Beispiel mit den Monatsnamen ergibt sich:



Kostenabschätzung: Jede der drei Operationen $Insert(x, S)$, $Delete(x, S)$, $Search(x, S)$ setzt sich dann aus zwei Schritten zusammen:

1. Berechnung der Adresse und Aufsuchen des Behälters.
2. Durchlaufen der unter der Adresse gespeicherten Einträge.

Der *Belegungsfaktor* $\alpha = \frac{n}{m}$ gibt gleichzeitig die durchschnittliche Listenlänge an.

Daraus ergibt sich für die *Zeit* folgende Kostenabschätzung:

- Adresse berechnen: $O(1)$
- Behälter aufsuchen: $O(1)$
- Liste durchsuchen:
 - Im Average Case, also bei erfolgreicher Suche: $O(1 + \frac{\alpha}{2})$
 - Im Worst Case, also bei erfolgloser Suche: $O(\alpha) = O(n)$

Die *Platzkomplexität* beträgt $O(m + n)$.

Verhalten:

- $\alpha \ll 1$: wie Array-Addressierung
- $\alpha \cong 1$: Übergangsbereich
- $\alpha \gg 1$: wie verkettete Liste

3.3.5 Closed Hashing (Hashing mit offener Adressierung)

[Güting, S.100] [Mehlhorn, S.117]

Diese Verfahren arbeiten mit einem statischen Array

```
var HashTable : array [0..m-1] of Element;
```

und müssen daher Kollisionen mit einer neuerlichen Adressberechnung behandeln („Offene Adressierung“). Dazu dient eine *Sondierungsfunktion* oder *Rehashing-Funktion*, die eine Permutation aller Hashtabellen-Plätze und damit die Reihenfolge der zu sondierenden Plätze angibt:

- jedes $x \in U$ definiert eine Folge $h(x, j)$, $j = 0, 1, 2, \dots$ von Positionen in der Hashtabelle.
- diese Folge muß bei jeder Operation durchsucht werden.
- Definition der zusätzlichen Hashfunktionen $h(x, j)$, $j = 0, 1, 2, \dots$ erforderlich

Achtung: Die Operation „Delete(x, S)“ erfordert eine Sonderbehandlung, weil die Elemente, die an x mittels Rehashing vorbeigeleitet wurden, auch nach Löschen von x noch gefunden werden müssen. Dazu führt man zusätzlich zu den Werten aus U die Werte „empty“ und „deleted“ ein, wobei „deleted“ kennzeichnet, daß nach diesem Feld noch weitergesucht werden muß.

Zwei Arten von Kollisionen können unterschieden werden für $x \neq y$:

1. Primärkollisionen: $h(x, 0) = h(y, 0)$
2. Sekundärkollisionen: $h(x, j) = h(y, 0)$ für $j = 0, 1, 2, \dots$

Lineares Sondieren

Die (Re)Hashfunktionen $h(x, j)$, $j = 0, 1, 2, \dots$ sollen so gewählt werden, daß für jedes x der Reihe nach sämtliche m Zellen der Hashtabelle inspiziert werden. Dabei muss jedoch beachtet werden, daß für zwei Werte x und y mit gleichen Hashwerten

$h(x, 0) = h(y, 0)$ auch deren Sondierbahnen gleich sind. Auf diese Weise können lange Sondierketten entstehen.

Einfachster Ansatz: Lineares Sondieren

$$h(x, j) = (h(x) + j) \pmod{m} \quad 0 \leq j \leq m - 1$$

Ein Problem beim linearen Sondieren ist die *Clusterbildung*: Es besteht die Tendenz, daß immer längere zusammenhängende, belegte Abschnitte in der Hashtabelle entstehen, so genannte Cluster. Diese Cluster verschlechtern deutlich die mittlere Suchzeit im Vergleich zu einer Hashtabelle mit demselben Belegungsfaktor, wo aber die Elemente zufällig gestreut sind.

Beispiel: $\text{Insert}(x, S)$ mit Monatsnamen in natürlicher Reihenfolge

| | | |
|-----|-----------|-------------|
| 0: | November | |
| 1: | April | (Dezember) |
| 2: | Maerz | |
| 3: | Dezember | |
| 4: | | ← |
| 5: | | |
| 6: | Mai | (September) |
| 7: | September | ← |
| 8: | Januar | |
| 9: | Juli | |
| 10: | | |
| 11: | Juni | |
| 12: | August | (Oktober) |
| 13: | Februar | ← |
| 14: | Oktober | |
| 15: | | |
| 16: | | |

Verallgemeinerung:

$$h(x, j) = (h(x) + c \cdot j) \pmod{m} \quad \text{für } 0 \leq j \leq m - 1$$

c ist eine Konstante, wobei c und m teilerfremd sein sollten.

Keine Verbesserung: Es bilden sich Ketten mit Abstand c .

Quadratisches Sondieren

Vorstufe: Um die oben angesprochene Häufung zu vermeiden, kann man mit folgender (Re-)Hashfunktion quadratisch sondieren:

$$h(x, j) = (h(x) + j^2) \pmod{m} \quad \text{für } 0 \leq j \leq m - 1$$

Warum j^2 ? Wenn m eine Primzahl ist, dann sind die Zahlen $j^2 \pmod{m}$ alle verschieden für $j = 0, 1, \dots, \left\lfloor \frac{m}{2} \right\rfloor$.

Verfeinerung: $1 \leq j \leq \frac{m-1}{2}$

$$\begin{aligned} h(x, 0) &= h(x) \pmod{m} \\ h(x, 2j-1) &= (h(x) + j^2) \pmod{m} \\ h(x, 2j) &= (h(x) - j^2) \pmod{m} \end{aligned}$$

Wähle m als Primzahl mit $m \pmod{4} = 3$ (Basis: Zahlentheorie).

Quadratisches Sondieren ergibt keine Verbesserung für *Primärkollisionen* ($h_0(x) = h_0(y)$), aber es vermeidet Clusterbildung bei *Sekundärkollisionen* ($h_0(x) = h_k(x)$ für $k > 0$), das heißt die Wahrscheinlichkeit für die Bildung längerer Ketten wird herabgesetzt [Güting, S. 127].

Doppel-Hashing

Beim Doppel-Hashing wählt man zwei Hashfunktionen $h(x)$ und $h'(x)$ und nimmt an:

$$\begin{aligned} Pr(h(x) = h(y)) &= \frac{1}{m} \quad x \neq y \\ Pr(h'(x) = h'(y)) &= \frac{1}{m} . \end{aligned}$$

Falls Kollisionswahrscheinlichkeiten für h und h' unabhängig sind, gilt

$$Pr(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2} .$$

Wir definieren

$$h(x, i) = (h(x) + j^2 \cdot h'(x)) \pmod{m} \quad \text{mit } 0 \leq j \leq m - 1 .$$

Eigenschaften:

- mit Abstand die beste der hier vorgestellten Strategien zur Kollisionsbehandlung
- kaum unterscheidbar von idealem Hashing.

Komplexität des geschlossenen Hashings

Wir wollen nun die Kosten für geschlossene Hashverfahren analysieren ([Mehlhorn, 118f.] und [Aho et al. 83, S. 131]). Dabei interessieren uns die Kosten der drei Standard-Operationen *Search*, *Insert* und *Delete*. Um jedoch Aussagen darüber machen zu können, benötigen wir Aussagen über die Kollisionswahrscheinlichkeit.

Gegeben sei eine *ideale Hashfunktion*, d.h. alle Plätze sind gleichwahrscheinlich und es gibt keine Kettenbildung.

$q(i; n, m) :=$ Wahrscheinlichkeit, daß mindestens i Kollisionen auftreten, wenn m die Größe der Hashtabelle ist und bereits n Elemente eingetragen sind.
 $=$ Wahrscheinlichkeit, daß die ersten $i - 1$ Positionen $h(x, 0), h(x, 1), \dots, h(x, i - 1)$ durch die Rehashing-Strategie besetzt sind (für n und m).

Dann gilt für

$i = 1:$

$$q(1; n, m) = \frac{n}{m}$$

$i = 2:$ Nach der ersten Kollision wird beim Rehashing eine von $m - 1$ Zellen geprüft, von denen $n - 1$ besetzt sind

$$q(2; n, m) = \frac{n}{m} \cdot \frac{n - 1}{m - 1}$$

$i:$ Nach $i - 1$ Kollisionen testet die Sondierungsfunktion noch eine von $m - (i - 1)$ Zellen, von denen $n - (i - 1)$ besetzt sind. Mittels Rekursion ergibt sich so

$$\begin{aligned} q(i; n, m) &= q(i - 1; n, m) \cdot \frac{n - i + 1}{m - i + 1} \\ &\vdots \\ &= \frac{n}{m} \cdot \frac{n - 1}{m - 1} \cdot \dots \cdot \frac{n - i + 1}{m - i + 1} \\ &= \prod_{j=0}^{i-1} \frac{n - j}{m - j} \end{aligned}$$

- $C_{Ins}(n, m)$ seien die mittleren Kosten von $\text{Insert}(x, S)$, also der Eintragung des $(n + 1)$ -ten Elementes in eine m -elementige Hashtabelle. Dann gilt

konkrete Operation $\text{Insert}(x)$

$$\text{Zahl der Sondierungsschritte} = 1 + \min\{i > 0 : T[h(x, i)] \text{ ist frei}\}$$

$$\begin{aligned}
C_{Ins}(n, m) &= \frac{\text{mittlere Kosten von Insert}}{n} \\
&= 1 + \sum_{i=1}^n i \cdot Pr(\text{exakt } i \text{ Sondierungsschritte} \mid n, m) \\
&= 1 + \sum_{i=1}^n i \cdot p(i \mid n, m) \quad p(i \mid n, m) \equiv 0 \text{ für } i > n \\
&= \dots \\
&= 1 + \sum_{i=1}^n Pr(\text{mindestens } i \text{ Sondierungsschritte} \mid n, m) \\
&= 1 + \sum_{i=1}^n q(i; n, m)
\end{aligned}$$

$$\begin{aligned}
\sum_{i=1}^n i \cdot p(i \mid n, m) &= \\
\sum_{i=1}^n i \cdot p_i &= p_1 + 2p_2 + 3p_3 + \dots + np_n \\
&= p_1 + p_2 + p_3 + \dots + p_n \equiv q_1 \\
&\quad + p_2 + p_3 + \dots + p_n \equiv q_2 \\
&\quad + p_3 + \dots + p_n \equiv q_3 \\
&\quad \dots \\
&\quad + p_n \equiv q_n \\
&= \sum_{i=1}^n q_i
\end{aligned}$$

$$\begin{aligned}
C_{Ins}(n, m) &= 1 + \sum_{i=1}^n q(i; n, m) \\
&= \frac{m+1}{m+1-n} \\
&= \frac{1}{1 - \frac{n}{m+1}} \\
&\cong \frac{1}{1-\alpha} \text{ mit } \alpha := \frac{n}{m}
\end{aligned}$$

Der Beweis kann mittels vollständiger Induktion über m und n geführt werden.

- $C_{Sea}^-(n, m)$ seien die mittleren Kosten von $\text{Search}(x, S)$ bei erfolgloser Suche. Dazu werden die Positionen $h(x, 0), h(x, 1), \dots$ getestet. Bei der ersten freien Zelle wird abgebrochen. Also

$$C_{Sea}^-(n, m) = C_{Ins}(n, m)$$

- $C_{Sea}^+(n, m)$ seien die mittleren Kosten von $\text{Search}(x, S)$ bei erfolgreicher Suche. Dabei werden alle Positionen $h(x, 0), h(x, 1), \dots$ durchlaufen, bis wir ein i finden mit $T[h(x, i)] = x$. Dieses i hat auch die Kosten für die Operation $\text{Insert}(x, S)$ bestimmt, und so erhält man durch Mittelung über alle Positionen $j = 0, \dots, n-1$ aus $C_{Ins}(j, m)$ folgende Kosten

$$\begin{aligned} C_{Sea}^+(n, m) &= \frac{1}{n} \cdot \sum_{j=0}^{n-1} C_{Ins}(j, m) \\ &= \frac{m+1}{n} \cdot \sum_{j=0}^{n-1} \frac{1}{m+1-j} \\ &= \frac{m+1}{n} \cdot \left[\sum_{j=1}^{m+1} \frac{1}{j} - \sum_{j=1}^{m+1-n} \frac{1}{j} \right] \\ &\dots \quad (\text{harmonische Zahlen}) \\ &\cong \frac{m+1}{n} \cdot \ln \frac{m+1}{m+1-n} \\ &\cong \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha} \end{aligned}$$

- $C_{Del}(n, m)$ seien die mittleren Kosten von $\text{Delete}(x, S)$. Dann werden alle Zellen $h(x, 0), h(x, 1), \dots$ durchlaufen bis $T[h(x, i)] = x$. Also entsprechen die Kosten denen der erfolgreichen Suche:

$$C_{Del}(n, m) := C_{Sea}^+(n, m)$$

Näherung für die Kostenanalyse: Falls $n, m \gg i$:

$$q(i; n, m) \cong \left(\frac{n}{m}\right)^i$$

Die Näherung ist gut, falls $i \leq n \ll m$ erfüllt ist.

Damit können wir die Formeln der geometrischen Reihe verwenden und einfacher

rechnen. Es genügt, die beiden Arten von Kosten zu betrachten:

$$\begin{aligned}
 C_{Sea}^-(n, m) = C_{Ins}(n, m) &\cong \sum_{i=0}^{\infty} \left(\frac{n}{m}\right)^i \\
 &= \frac{1}{1 - \alpha} \\
 C_{Sea}^+(n, m) = C_{Del}(n, m) &\cong \frac{1}{n} \int_0^{n-1} \frac{m}{m-x} dx \\
 &= \frac{m}{n} \ln \frac{m}{m+1-n} \\
 &\cong \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
 \end{aligned}$$

Nichtideales Hashing: Clustering (Kettenbildung) mit linearem Sondieren:

Formeln von Knuth (1973)

Wie bei idealem Hashing genügen zwei Größen:

$$\begin{aligned}
 C_{Sea}^+(n, m) &= \frac{1}{2} \left[1 + \frac{1}{1-\alpha} \right] \\
 C_{Sea}^-(n, m) &= \frac{1}{2} \left[1 + \frac{1}{1-\alpha^2} \right]
 \end{aligned}$$

3.3.6 Zusammenfassung der Hashverfahren

Im Average Case zeigen die Hashverfahren allgemein ein effizientes Verhalten mit $O(1)$, erst im Worst Case liegen die Operationen bei $O(n)$.

Ein Nachteil ist, daß alle Funktionen, die auf einer Ordnung basieren, z.B. $ListOrder(S)$ nicht unterstützt werden.

Anwendungen liegen dort, wo $|U| \gg |S|$ und dennoch ein effizienter Zugriff auf die Elemente wünschenswert ist. Als Beispiel haben wir die Symboltabellen von Compilern kennengelernt.

3.4 Binäre Suchbäume

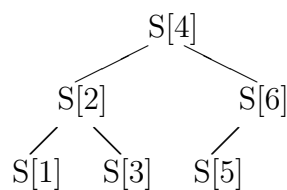
3.4.1 Allgemeine binäre Suchbäume

[Mehlhorn, Seite 140]

Ausgangspunkt: Binäre Suche

$$next := \left\lceil \frac{high + low}{2} \right\rceil$$

Veranschaulichung durch binären Baum für Array $S[1..6]$:



Zeitkomplexität für die Array-Darstellung:

- Search:
 - Der Wert von $[high - low + 1]$ wird bei jedem Durchgang der While-Schleife halbiert.
 - Zeitkomplexität höchstens $O(\lg n)$.
- Insert/Delete-Operationen: problematisch
z.B. erfordert Insert das Verschieben eines Teils des Arrays. Daher: Zeitkomplexität $O(n)$.
Ausweg: Wird der obige Suchbaum durch **Zeiger** statt durch ein Array realisiert, laufen auch die Operationen *Delete* und *Insert* effizient ab mit

$$T(n) \in O(\lg n) .$$

Definition 3.4.1 (binärer Suchbaum) Ein binärer Suchbaum für die n -elementige Menge $S = \{x_1 < x_2 < \dots < x_n\}$ ist ein binärer Baum mit n Knoten $\{v_1 \dots v_n\}$.

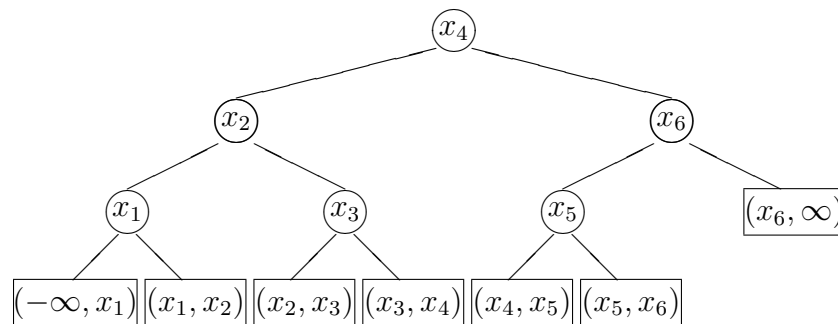
Die Knoten sind mit den Elementen von S beschriftet, d.h. es gibt eine injektive Abbildung $Inhalt : \{v_1 \dots v_n\} \rightarrow S$. Die Beschriftung bewahrt die Ordnung, d.h. wenn v_i im linken Unterbaum ist, v_j im rechten Unterbaum, v_k Wurzel des Baumes, so gilt:

$$Inhalt [v_i] < Inhalt [v_k] < Inhalt [v_j] .$$

Äquivalente Definition: Ein binärer Suchbaum ist ein Binärbaum, dessen *Inorder-Durchlauf* die Ordnung auf S ergibt.

In der Regel identifizieren wir Knoten mit ihrer Beschriftung: Den Knoten v mit Beschriftung x bezeichnen wir also auch mit „Knoten x “.

Beispiel



Die $n + 1$ Blätter des Baumes stellen die Intervalle von U dar, die kein Element von S enthalten, also enden erfolglose Suchoperationen entsprechend immer in Blättern.

Somit steht

$$(x_1, x_2) \quad \text{für} \quad \{y \mid y \in U : x_1 < y < x_2\}$$

Da die Blätter sich aus S ergeben, müssen sie nicht explizit abgespeichert werden.

Beispiel: Deutsche Monatsnamen

Dargestellt ist ein binärer Suchbaum in lexikographischer Ordnung entstanden durch Einfügen der Monatsnamen in kalendarischer Reihenfolge.

Die Inorder-Traversierung bzw. die Projektion auf die x-Achse erzeugt die alphabetische Reihenfolge.

Übung: Erstellen Sie den Baum für die umgekehrte Reihenfolge.

Programm: Binäre Suchbäume [Mehlhorn, Seite 141]

```

procedure Search(a,S);      (* mit expliziten Blättern *)
begin
  v := Root of T;
  while (v is node) and (a ≠ Inhalt[v]) do
    if (a < Inhalt[v])
      then v = LeftSon[v]
      else v = RightSon[v]
end;

```

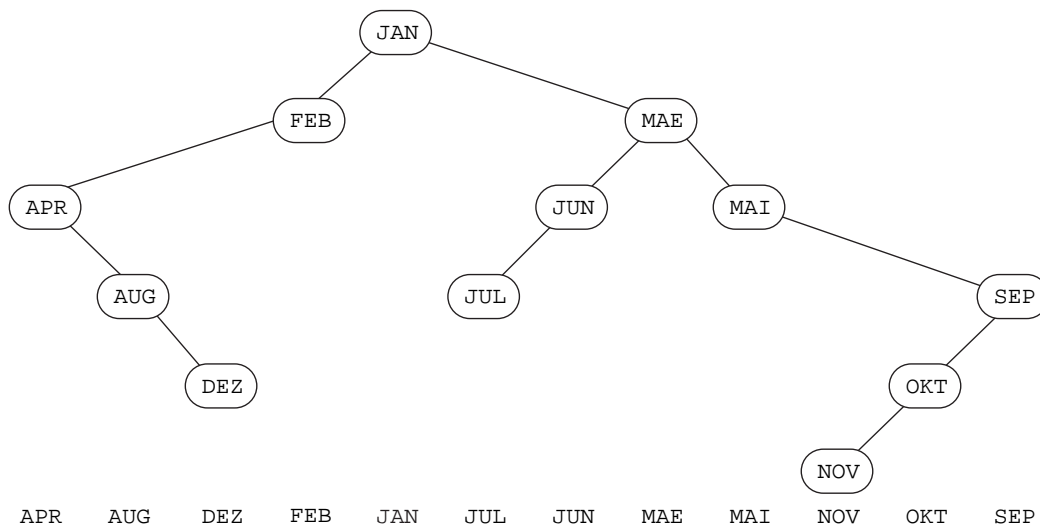


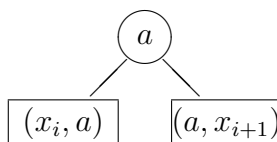
Abbildung 3.2: Binärer Suchbaum für deutsche Monatsnamen

Das obige Programm Search endet

- in einem Knoten v , falls $a = \text{Inhalt}[v]$;
- in einem Blatt (x_i, x_{i+1}) mit $x_i < a < x_{i+1}$, falls $a \notin S$.

Dann haben wir:

- **Insert**(a, S): ersetze das Blatt (x_i, x_{i+1}) mit dem Intervall, in dem a liegt durch den folgenden Teilbaum



- **Delete**(a, S): Diese Operation gestaltet sich etwas komplizierter, denn falls der Knoten a einen oder mehrere Söhne hat, kann er nicht einfach entfernt werden, sondern einer seiner Söhne muß dann an seine Stelle treten. Blätter, die ja nicht explizit gespeichert werden, wollen wir nicht als Söhne betrachten. Man geht dann folgendermaßen vor:

Die Suche endet in einem Knoten v mit $\text{Inhalt}[v] = a$. Endet sie in einem Blatt, erhält man eine Fehlermeldung.

Fallunterscheidung:

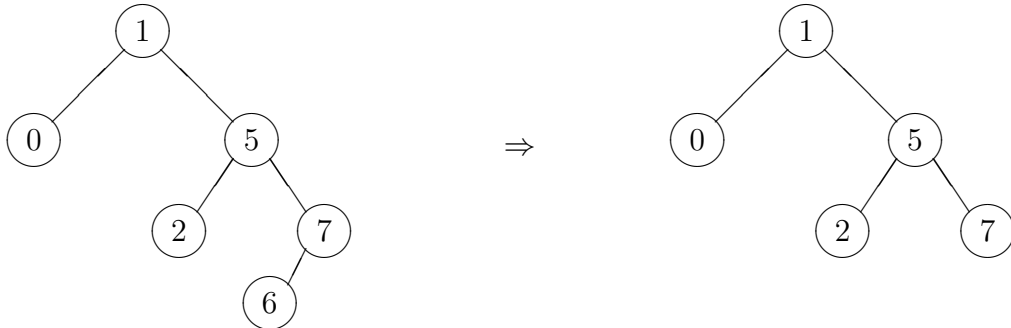
- Der Knoten v hat mindestens ein Blatt als Sohn:

- ersetze v durch den Sohn und
 - streiche v und das Blatt aus dem Baum.
- b) Der Knoten v hat zwei Söhne die innere Knoten sind:
- sei w der rechteste Unterknoten im linken Unterbaum von v
Dieser wird gefunden durch:
 - * $LeftSon[v]$
 - * rekursiv $RightSon[v]$, bis wir auf ein Blatt treffen.
 - $Inhalt[v] = Inhalt[w]$
 - um w zu entfernen, fahre fort wie in Fall a).

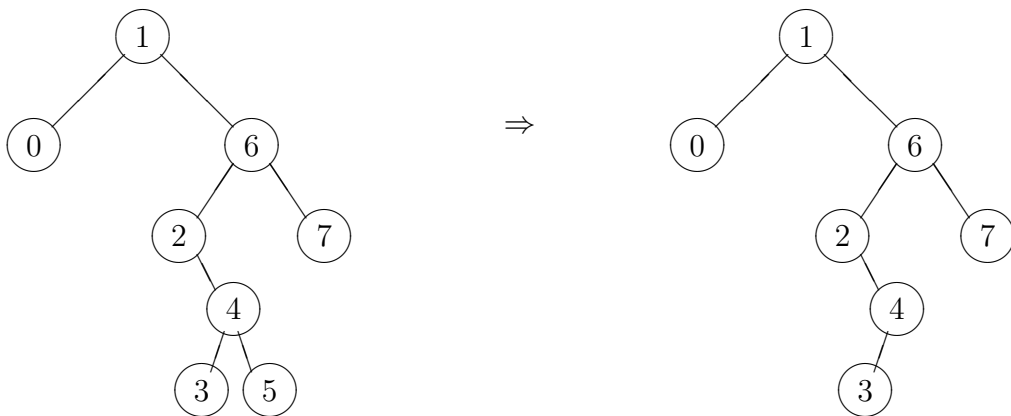
Dabei bleibt die Suchbaum-Eigenschaft erhalten!

Beispiel: Delete(6,S) (ähnlich [Mehlhorn, Seite 143])

Fall a): Der Knoten 6 hat mindestens ein Blatt als Sohn.



Fall b): Der Knoten 6 hat zwei Söhne.



Zeitkomplexitäten

Standard-Operationen *Search*, *Insert* und *Delete*:

- im wesentlichen: Baum-Traversieren und einige lokale Änderungen, die $O(1)$ Zeitkomplexität haben.

- deswegen: insgesamt $O(h(T))$, wobei $h(T)$ die Höhe des Suchbaums T ist.

Operation $ListOrder(S)$:

- Traversieren in symmetrischer Ordnung
- $O(|S|) = O(n)$

Operation $Order(k, S)$ (Ausgabe des k kleinsten oder größten Elementes):

- In jedem Knoten wird zusätzlich die Anzahl der Knoten seines linken Unterbaumes gespeichert. Mit Aufwand $O(1)$ kann dieser Wert bei Delete- und Insert- Operationen aktualisiert werden.
- Damit kann man die Funktion $Order$ mit einer Komplexität $O(h(T))$ implementieren (Übungsaufgabe).

Implementierung eines Binären Suchbaums

[Sedgewick 88, Seite 208].

Zusätzlich zu den drei Standard-Operationen entwerfen wir die Routinen $Initialize(S)$ und $ListOrder(S)$ zum Initialisieren bzw. zur geordneten Ausgabe eines binären Suchbaums.

Eigenheiten der Implementierung (siehe Abbildung 3.3):

- das *Head-Element* **head** (Tree Header Node, Anker, Anchor), also der Listenanfang (nicht zu verwechseln mit der Wurzel), und
- die *Darstellung der Blätter*, z.B. als Dummy Node **z**.

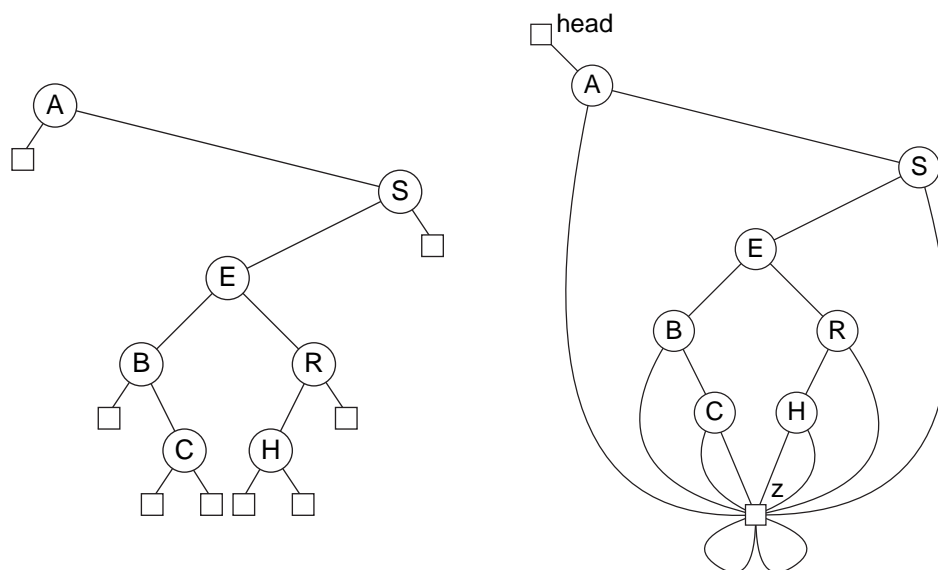


Abbildung 3.3: Binärer Suchbaum und seine Implementation mit Head-Element und Dummy-Knoten z

Implementierung:

```

type link = REF node;
node = record
    key, info : integer;
    l, r : link
end;
var head, z: link;

procedure TreeInitialize;
begin
    new(z); z↑.l := z; z↑.r := z;
    new(head); head↑.key := 0; head↑.r := z;
end;

```

Anmerkung: Falls auch negative Elemente zulässig sind, muß $\text{head}\uparrow.\text{key} = -\infty$ sein.

```

function TreeSearch (v : integer; x: link) : link;
begin
    z↑.key := v;

```

```

repeat
  if v < x↑.key
    then x := x↑.l
    else x := x↑.r
until v = x↑.key;
return x;
end;

function TreeInsert (v : integer; x: link) : link;
var p: link;
begin
  repeat
    p := x;      (* p ist Vorgaenger von x *)
    if v < x↑.key
      then x := x↑.l
      else x := x↑.r
    until x = z;
  new (x); x↑.key := v; x↑.l := z; x↑.r := z; (* x wird ueberschrieben *)
  if v < p↑.key
    then p↑.l := x (* x wird linker Nachfolger von p *)
    else p↑.r := x; (* x wird rechter Nachfolger von p *)
  return x;
end;

procedure TreePrint (x: link); (* inorder *)
begin
  if x <> z then begin
    TreePrint(x↑.l);
    printnode(x);
    TreePrint(x↑.r)
  end
end;
end;

```

Delete(T, E) (siehe Abbildung 3.4)

1. Suche den kleinsten Schlüssel im rechten Teilbaum von E . Dazu geht man einmal nach rechts und danach immer nach links, bis man auf einen Knoten trifft, der keinen linken Sohn mehr hat, das ist H .
2. Dann macht man, falls vorhanden den rechten Sohn dieses Knotens (also N) zum linken Sohn des Vaterknotens (also R). Dazu wird einfach die Adresse von H im l -Zeiger von R überschrieben mit der von N . Dabei wird *ein Zeiger* geändert.

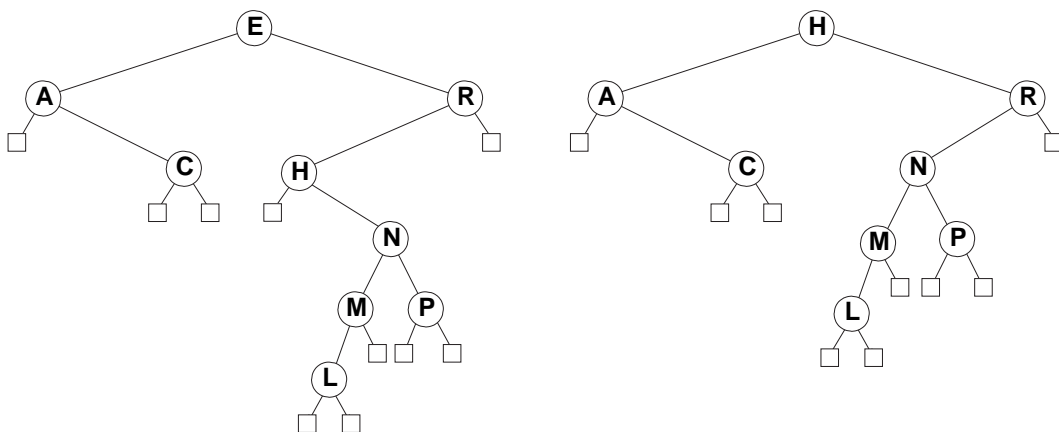


Abbildung 3.4: Beispiel eines Baumes T vor und nach $\text{Delete}(T, E)$

3. Jetzt muß H noch die Position von E einnehmen, dazu gehört:

- a) Die Nachfolger-Zeiger von H müssen auf die Söhne von E zeigen; also Link l auf A und Link r auf R . Dabei werden *zwei* Zeiger umdirigiert.
- b) Falls notwendig muß das entsprechende Link (l,r) des Vaters von E überschrieben werden durch einen Verweis auf H . Dazu wird *ein* Zeiger umdirigiert.

Insgesamt werden also nur vier Zeiger umgesetzt.

```

procedure TreeDelete (t, x : link);
  var p, c : link;
  (* p=parent, c=child *)
begin

  (* Suchen des Knotens t im von x induzierten Teilbaum *)
  repeat
    p := x; (* p ist Vorgänger von x *)
    if t↑.key < x↑.key
      then x := x↑.l
      else x := x↑.r
  until x = t; (* Knoten t ist gefunden *)

  (* t hat keinen rechten Sohn *)
  if t↑.r = z
    then x := t↑.l (* t wird durch seinen linken Sohn ersetzt *)

  (* t hat rechten Sohn, der keinen linken hat *)
  else if t↑.r↑.l = z;
    then begin
      x := t↑.r (* t wird durch seinen rechten Sohn ersetzt *)
      x↑.l := t↑.l (* der rechte Sohn übernimmt den linken Sohn von t*)
    end

  (* sonst *)
  else begin
    c := t↑.r; (* c speichert den Nachfolger von t *)
    while c↑.l↑.l <> z do c := c↑.l;
      (* c↑.l ist nun der linkeste Sohn und c dessen Vater *)
    x := c↑.l; (* t wird durch linkesten Sohn ersetzt *)
    c↑.l := x↑.r; (* c uebernimmt den rechten Sohn von x *)
    x↑.l := t↑.l;
    x↑.r := t↑.r
  end;

  (* Je nachdem ob t linker oder rechter Sohn von p war, *)
  (* muß der entsprechende Zeiger auf x verweisen *)
  if t↑.key < p↑.key
    then p↑.l := x
    else p↑.r := x;
end;

```

Anmerkungen zu TreeDelete(t,x):

- unsymmetrisch: hier Analyse des rechten Teilbaums
- Variable (Zielwerte):
 - p = parent of x (tree to be searched)
 - c = child of t (to be deleted)
- Ergebnis: x = child of p
- Ablauf:
 1. Suchen des Knotens t im Teilbaum x
 2. Fallunterscheidung für t
 - a) Kein rechter Sohn (C, L, M, P, R)
 - b) Rechter Sohn ohne linken Sohn (A, N)
 - c) sonst: (E, H)
 - suche kleinsten Schlüssel im rechten Teilbaum von t (Name: c)
(beachte: c hat keinen linken Sohn)
 - setze Zeiger um.
 3. setze Zeiger für Knoten x

Mit diesen Prozeduren und Funktionen lassen sich die fünf Operationen folgendermaßen realisieren:

```

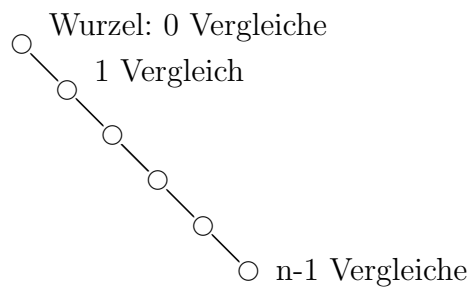
Initialize(S): TreeInitialize()
Search(v,S): y := TreeSearch(v,head)
Insert(v,S): y := TreeInsert(v,head)
Delete(v,S): TreeDelete(TreeSearch(v,head),head)
ListOrder(S): TreePrint(head↑.r)
  
```

Komplexitätsanalyse

Annahme: Konstruktion des binären Suchbaums durch n Insert-Operationen.

$C(n)$ = Zahl der Vergleiche.

1. Im *Worst Case* handelt es sich um einen Suchbaum, der zu einer linearen Liste entartet, da die Elemente in aufsteigender Reihenfolge eingetragen werden.



Also gilt für die Zahl der Vergleiche

$$C(n) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

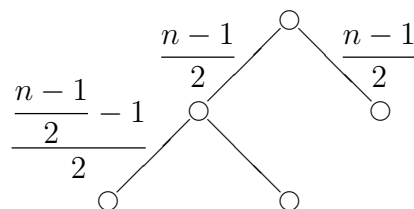
und für den *Aufwand*

$$\frac{C(n)}{n} = \frac{n-1}{2}.$$

2. Im *Best Case* handelt es sich um einen Binärbaum mit minimaler Höhe $h = \text{ld}(n+1)$, in dem man maximal $n = 2^h - 1$ Knoten unterbringen kann (s. Absch. 1.3.5).

Auf der Wurzelebene (0. Ebene) kann man ohne Vergleich einen Knoten plazieren, auf der ersten Ebene mit jeweils einem Vergleich zwei Knoten, auf der zweiten mit jeweils zwei Vergleichen vier Knoten usw.

Folgende Darstellung veranschaulicht die Verteilung der n Datensätze auf die Zweige des Binärbaumes.

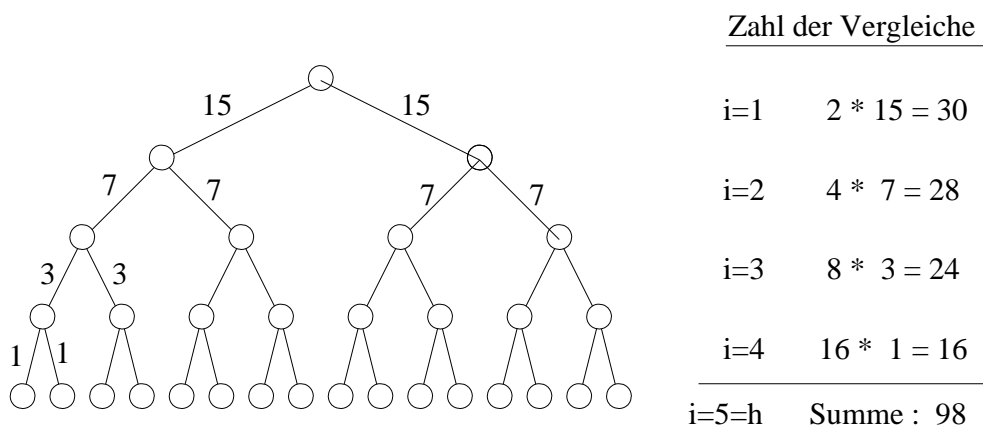


Über die Anzahl der Vergleiche mit der Wurzel ($2 \cdot \frac{n-1}{2}$) kommt man zu folgender Rekursionsgleichung:

$$C(n) = n - 1 + 2 \cdot C\left(\frac{n-1}{2}\right), \quad C(1) = 0$$

Beispiel: Die folgende Darstellung veranschaulicht die Herleitung der Formel über Gesamtzahl der Vergleiche, die direkt abhängt von der Knotenanzahl im Baum.

Sei $h = 5$ und somit $n = 2^h - 1 = 31$.



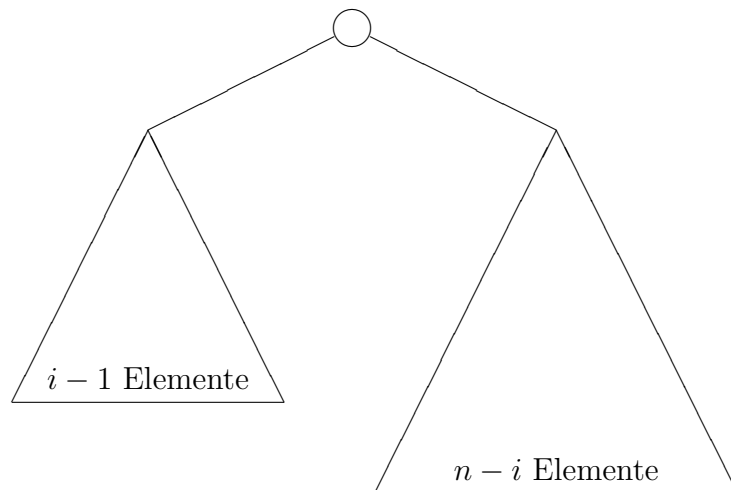
Summiert man nun über die Anzahl *aller* Vergleiche, so erhält man

$$\begin{aligned}
 C(n) &= \sum_{i=1}^{h-1} \underbrace{2^i}_{\text{Äste}} \cdot \underbrace{(2^{h-i} - 1)}_{\text{Vergleiche pro Ast}} \\
 &= \sum_{i=1}^{h-1} (2^h - 2^i) \\
 &= (h-1) \cdot 2^h - \left[\sum_{i=0}^{h-1} 2^i - 1 \right] \\
 &= (h-1) \cdot 2^h - \frac{2^h - 1}{2 - 1} - 1 \\
 &= 2^h \cdot (h-2) + 2 \\
 &= (n+1) \cdot \text{ld}(n+1) - 2n
 \end{aligned}$$

und für den Aufwand

$$\frac{C(n)}{n} \cong \text{ld } n .$$

3. Im *Average Case* werden von n Elementen $i - 1$ im linken Teilbaum eingetragen und der Rest abgesehen von der Wurzel im rechten.



Dabei ist jede Aufteilung $i = 1, \dots, n$ gleichwahrscheinlich. Für die Anzahl der benötigten Vergleich gilt dann:

$$\begin{aligned}
 C(n) &= n - 1 \text{ Vergleiche, da alle Elemente an der Wurzel vorbei müssen} \\
 &\quad + C(i - 1) \text{ Vergleiche im linken Teilbaum} \\
 &\quad + C(n - i) \text{ Vergleiche im rechten Teilbaum}
 \end{aligned}$$

Und somit ergibt sich folgende Anzahl von Vergleichen:

$$\begin{aligned}
 C(n) &= \frac{1}{n} \sum_{i=1}^n [n - 1 + C(i - 1) + C(n - i)] \\
 &= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C(i) \quad (\text{wie Quicksort}) \\
 &= 2n \cdot \ln n + \dots \\
 &= 2n \cdot (\ln 2) \text{ld } n + \dots \\
 &= 1.386n \text{ld } n + \dots
 \end{aligned}$$

und entsprechend für den Aufwand:

$$\frac{C(n)}{n} \cong 1.386 \text{ld } n .$$

3.4.2 Der optimale binäre Suchbaum

[Mehlhorn, Seite 144]

Diese Bäume sind gewichtet mit der Zugriffsverteilung, die die Zugriffshäufigkeit (oder Wichtigkeit) der Elemente von S widerspiegelt.

Wir beschränken uns auf die Betrachtung von $\text{Search}(a, S)$.

Definition *Zugriffsverteilung*: Gegeben sei eine Menge $S = \{x_1 < x_2 < \dots < x_n\}$ und zwei Sentinels x_0 und x_{n+1} mit $x_0 < x_i < x_{n+1}$ für $i = 1, \dots, n$.

Die Zugriffsverteilung ist ein $(2n + 1)$ -Tupel $(\alpha_0, \beta_1, \alpha_1, \beta_2, \alpha_2, \dots, \alpha_n, \beta_n)$ von Wahrscheinlichkeiten, für das gilt

- $\beta_i \geq 0$ ist die Wahrscheinlichkeit, daß eine $\text{Search}(a, S)$ -Operation *erfolgreich* im Knoten $x_i = a$ endet, und
- $\alpha_j \geq 0$ ist die Wahrscheinlichkeit, daß eine $\text{Search}(a, S)$ -Operation *erfolglos* mit $a \in (x_j, x_{j+1})$ endet.

Gilt für die Werte α_j und β_i

$$\sum_{i=1}^n \beta_i + \sum_{j=0}^n \alpha_j = 1$$

so spricht man von einer *normierten Verteilung*. Die Normierung ist zwar sinnvoll, wird aber im folgenden nicht benötigt.

Beispiel [Aigner, Seite 187]

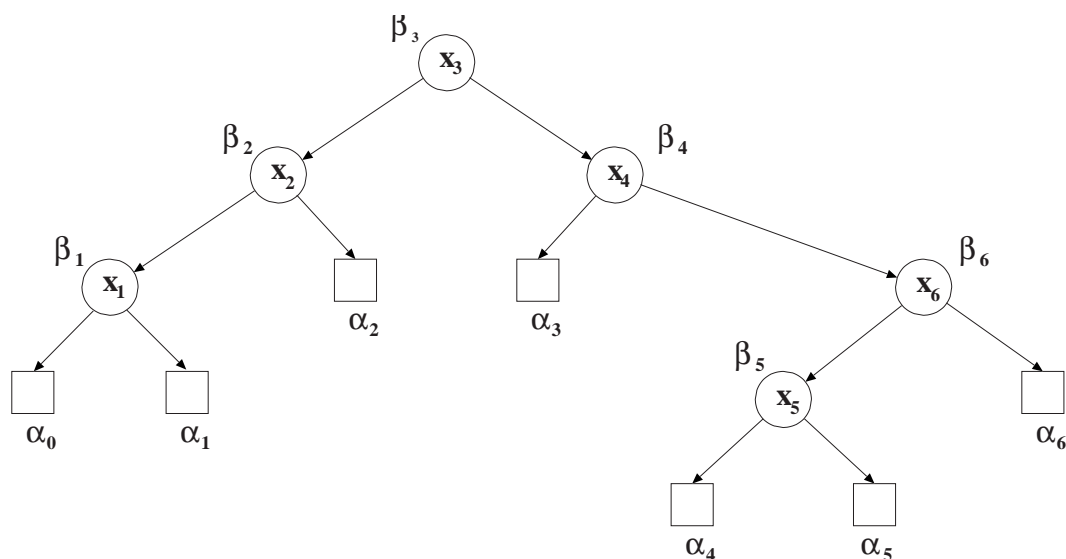


Abbildung 3.5: Baumdarstellung der Zugriffsverteilung

Komplexität von $\text{Search}(a, S)$

Wir betrachten nur den *Average Case*, also die mittlere Zahl von Vergleichen. Die Suche endet in einem inneren Knoten oder einem Blatt. Die Höhe eines Knotens oder eines

Blatts gibt direkt die Zahl der Vergleiche.
Sei ein Suchbaum gegeben mit:

b_i : Tiefe des Knotens x_i

a_j : Tiefe des Blattes (x_j, x_{j+1})

Sei P („Pfadlänge“) die mittlere Anzahl von Vergleichen, (auch gewichtete Pfadlänge):

$$P = \sum_{i=1}^n (\beta_i \cdot (1 + b_i)) + \sum_{j=0}^n (\alpha_j \cdot a_j)$$

Beachte: Ein Blatt erfordert einen Vergleich weniger als ein Knoten.

Bellman-Knuth-Algorithmus

[Ottmann, Seite 397]

Ziel: Konstruktion des optimalen Suchbaums, der bei gegebener Zugriffsverteilung die Pfadlänge minimiert.

Lösung: Dynamische Programmierung. Der Bellman-Knuth-Algorithmus fußt auf folgender Überlegung. Angenommen $S = \{x_i, \dots, x_j\}$ sei ein optimaler Suchbaum und $x_k \in S$. Dann definiert der Teilbaum mit Wurzel x_k eine Zerlegung von S in zwei Teilmengen

$$\{x_i, \dots, x_{k-1}\} \text{ und } \{x_k, \dots, x_j\}$$

Aufgrund der Additivität der Pfadlänge muß auch jeder der beiden Teilbäume optimal bezüglich der (Teil-)Pfadlänge sein.

Um die dynamische Programmierung anzuwenden, definieren wir die Hilfsgröße $c(i, j)$ mit $i < j$:

$$c(i, j) := \text{optimale Pfadlänge für den Teilbaum } \{x_i, \dots, x_j\}$$

Dann kann man $\{x_i, \dots, x_j\}$ aufgrund der Definition von $c(i, j)$ in zwei Teilbäume $\{x_i, \dots, x_{k-1}\}$ und $\{x_k, \dots, x_j\}$ zerlegen, die ihrerseits wiederum optimal sein müssen (siehe Abbildung 3.6).

Daher gilt

$$\begin{aligned} c(i, i) &= 0 && \text{für alle } i \\ c(i, j) &= w(i, j) + \min_{i < k \leq j} [c(i, k-1) + c(k, j)] && \text{für } i < j \end{aligned}$$

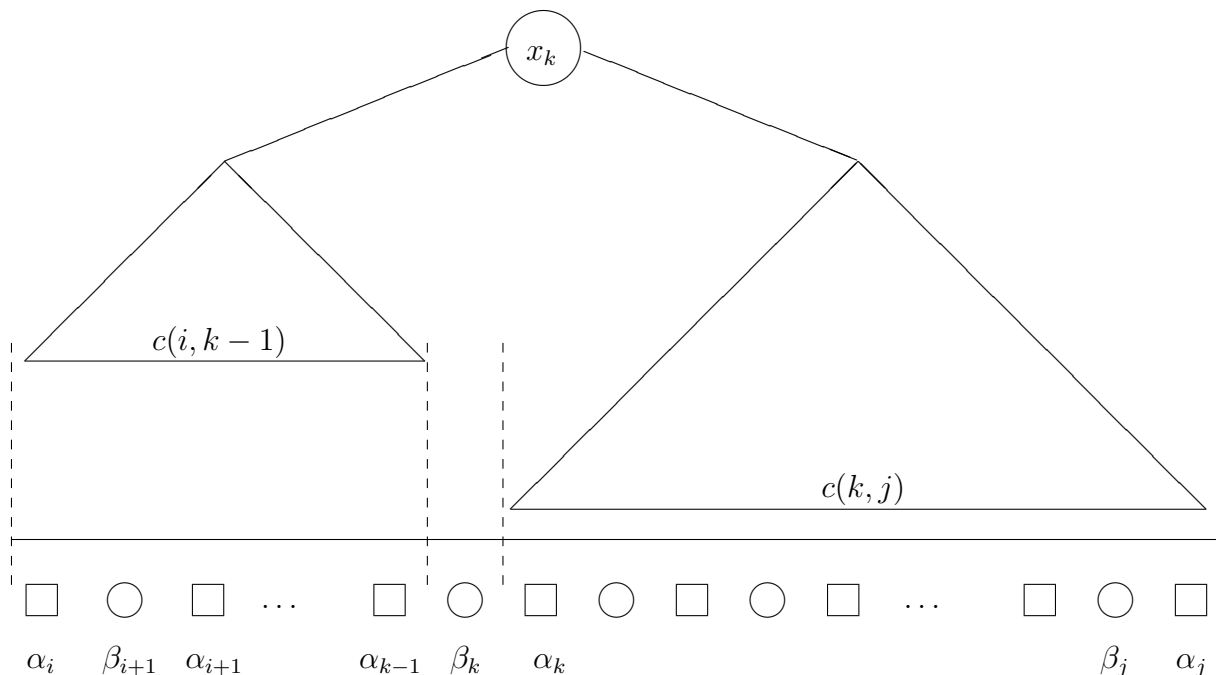


Abbildung 3.6: Schematische Darstellung der Teilbäume.

mit $w(i, j) = \sum_{l=i}^j \alpha_l + \sum_{l=i+1}^j \beta_l$.

Diese Gleichung ist iterativ, nicht rekursiv zu lösen. (DP-Gleichung; Gleichung der dynamischen Programmierung; Bellmansche Optimalitätsgleichung):
 gesuchtes Ergebnis:

$$P = c(0, n) .$$

Programm-Entwurf zur iterativen Lösung der DP-Gleichung

Die Lösung ist ähnlich der für das Matrix-Produkt (s. Abschnitt 1.4.2). Wir benötigen zwei zweidimensionale Arrays.

c: **array** [0..n][0..n] **of real** speichert die Pfadlängen, und

r: **array** [0..n][0..n] **of** [0..n] dient der Rekonstruktion des optimalen Suchbaums.

Dann geht man wie folgt vor:

1. berechne $w(i, j)$ vorab;
 initialisiere $c(i, i) = 0$.
2. Berechnung eines optimalen Suchbaumes mit zunehmend längeren Pfaden der Länge l .

```

for  $l=1$  to  $n$  do
  for  $i=0$  to  $n-l$  do
     $j := i + l$ 
     $c(i, j) = w(i, j) + \min_{i < k \leq j} [c(i, k-1) + c(k, j)]$ 
     $r(i, j) = \arg \min_{i < k \leq j} [c(i, k-1) + c(k, j)]$ 

```

3. Rekonstruktion des optimalen Suchbaums mittels des Index-Arrays $r(i, j)$

Dem Entwurf entnehmen wir die Komplexität der Konstruktion eines optimalen Suchbaumes:

- die *Zeitkomplexität* $O(n^3)$ und
- die *Platzkomplexität* $O(n^2)$.

Hinweise:

- Vergleiche:
 - Die naive Implementierung mittels Rekursion hat *exponentielle* Komplexität.
 - Dynamische Programmierung:
 - * Speichern der Zwischenwerte in einer Tabelle;
 - * Geschickter Aufbau der Tabelle.
- Verbesserung der Komplexität:
 - Matrix $r(i, j)$ (optimaler Index) ist monoton in jeder Spalte und Zeile für $0 \leq i < j \leq n$; d.h.:

$$r(i, j-1) \leq r(i, j) \leq r(i+1, j) \quad \forall i, j \in 0, \dots, n$$
 - Reduktion der Komplexität von $O(n^3)$ auf $O(n^2)$ (Beweis [Mehlhorn, Seite 151] mittels „Vierecksungleichung“ für $w(i, j)$).

Beispiel [Aigner, Seite 187]

| | | | | | | | | |
|------------|-----------|------------|-----------|------------|-----------|------------|-----------|------------|
| α_0 | β_1 | α_1 | β_2 | α_2 | β_3 | α_3 | β_4 | α_4 |
| 12 | 5 | 6 | 15 | 16 | 8 | 16 | 12 | 10 |

D.h. bei hundert Suchoperationen wird 12mal erfolglos nach einem Schlüssel kleiner x_1 gesucht, 5mal erfolgreich nach x_1 , 6mal erfolglos nach einem Schlüssel zwischen x_1 und x_2 usw.

Gesucht ist jetzt nach einer Anordnung, bei der diese hundert Suchoperationen mit minimalen Vergleichskosten p durchgeführt werden können, bzw. die minimalen Kosten selbst für die hundert Suchoperationen bei optimaler Anordnung.

| | | | | | |
|-----|-----|----------|----|-----|---|
| | | $w(i,j)$ | | | |
| | j | 1 | 2 | 3 | 4 |
| i | | | | | |
| 0 | 23 | 54 | 78 | 100 | |
| 1 | | 37 | 61 | 83 | |
| 2 | | | 40 | 62 | |
| 3 | | | | 38 | |

| | |
|---------|---|
| $l = 0$ | $c(i, i) = 0$ |
| $l = 1$ | $c(0, 1) = w(0, 1) + c(0, 0) + c(1, 1) = 23$ $c(1, 2) = w(1, 2) + c(1, 1) + c(2, 2) = 37$ $c(2, 3) = w(2, 3) + c(2, 2) + c(3, 3) = 40$ $c(3, 4) = w(3, 4) + c(3, 3) + c(4, 4) = 38$ |
| $l = 2$ | $c(0, 2) = w(0, 2) + \min[c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)]$ $= 54 + \min[37, 23] = 77$ $c(1, 3) = w(1, 3) + \min[c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)]$ $= 61 + \min[40, 37] = 98$ $c(2, 4) = w(2, 4) + \min[c(2, 2) + c(3, 4), c(2, 3) + c(4, 4)]$ $= 62 + \min[38, 40] = 100$ |
| $l = 3$ | $c(0, 3) = w(0, 3) + \min[c(0, 0) + c(1, 3), c(0, 1) + c(2, 3), c(0, 2) + c(3, 3)]$ $= 78 + \min[98, 63, 77] = 141$ $c(1, 4) = w(1, 4) + \min[c(1, 1) + c(2, 4), c(1, 2) + c(3, 4), c(1, 3) + c(4, 4)]$ $= 83 + \min[100, 75, 98] = 158$ |
| $l = 4$ | $c(0, 4) = w(0, 4) + \min[c(0, 0) + c(1, 4), c(0, 1) + c(2, 4), c(0, 2) + c(3, 4), c(0, 3) + c(4, 4)]$ $= 100 + \min[158, 123, 115, 141] = 215$ |

| | | r(i,j) | | | |
|---|---|--------|---|---|---|
| | | j | 1 | 2 | 3 |
| i | 0 | 1 | 2 | 2 | 3 |
| | 1 | | 2 | 3 | 3 |
| | 2 | | | 3 | 3 |
| | 3 | | | | 4 |

| | | c(i,j) | | | | |
|---|---|--------|----|----|-----|-----|
| | | j | 0 | 1 | 2 | 3 |
| i | 0 | 0 | 23 | 77 | 141 | 215 |
| | 1 | | 0 | 37 | 98 | 158 |
| | 2 | | | 0 | 40 | 100 |
| | 3 | | | | 0 | 38 |
| | 4 | | | | | 0 |

3.5 Balancierte Bäume

Die natürlichen binären Suchbäume haben einen gravierenden Nachteil. Im Worst Case haben sie eine Komplexität $O(n)$. Dieser Fall tritt ein, wenn sie aus der *Balance* geraten, d.h. in Strukturen entarten, die einer linearen Liste ähnlich sind. Dies passiert entweder aufgrund der Reihenfolge der Elemente bei $\text{Insert}(x, S)$ -Operationen oder auch wegen $\text{Delete}(x, S)$ -Operationen.

Abhilfe: Balancierte Bäume, die *garantierte* Komplexitäten von $O(\log n)$ haben.

Man unterscheidet (mindestens) zwei Balance-Kriterien:

1. *Gewichtsbalancierte Bäume*, auch BB(a)-Bäume (**B**ounded **B**alance) genannt: Die Anzahl der Blätter in den Unterbäumen wird möglichst gleich gehalten, wobei a die beschränkte Balance heißt (beschränkt den max. relativen Unterschied in der Zahl der Blätter zwischen den beiden Teilbäumen jedes Knotens) [Mehlhorn, Seite 176].
2. *Höhenbalancierte Bäume*: Bei diesen Bäumen ist das Balance-Kriterium die Höhe der Unterbäume, deren Unterschied möglichst gering gehalten wird. Je nach Ausprägung kann man hier viele Untertypen unterscheiden; die wichtigsten sind die

- *AVL-Bäume* und die
- (a, b) -Bäume, z.B. ein $(2,4)$ -Baum.

Wenn sie das Kriterium $b = 2a - 1$ erfüllen, werden sie als *B-Baum* bezeichnet. Das 'B' steht hier für „balanced“. Ein bekannter B-Baum ist der $(2,3)$ -Baum [Mehlhorn, Seite 186].

3.5.1 AVL-Bäume

[Güting, Seite 120]

Adelson-Velskij und Landis 1962:

- historisch erste Variante eines balancierten Baums;
- vielleicht nicht die effizienteste Methode, aber gut zur Darstellung des Prinzips geeignet.

Definition: Ein *AVL-Baum* ist ein binärer Suchbaum mit einer Struktur-Invarianten: Für jeden Knoten gilt, daß sich die Höhen seiner beiden Teilbäume höchstens um eins unterscheiden.

Um diese Invariante auch nach Update-Operationen (Delete, Insert) noch zu gewährleisten benötigt man *Rebalancierungs-Operationen*. Je nach vorangegangener Operation und Position des veränderten Elements können diese sehr aufwendig sein.

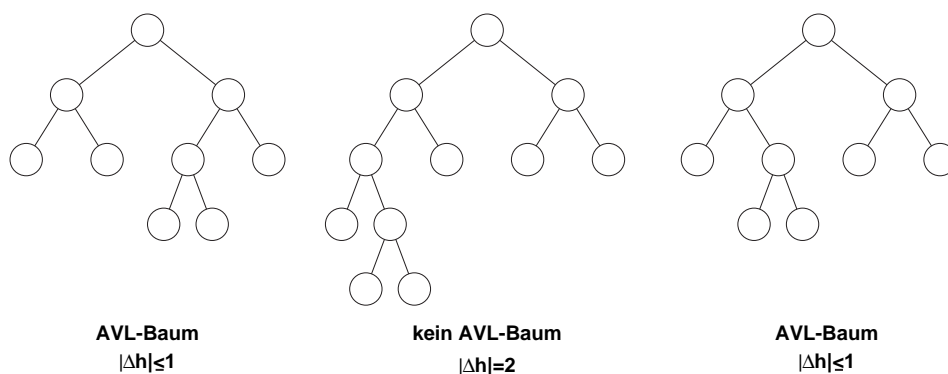


Abbildung 3.7: Beispiele für AVL-Bäume

Höhe eines AVL-Baumes

Wie hoch kann ein AVL-Baum bei vorgegebener Knotenzahl maximal, also im Worst Case sein? Oder: Wie stark kann ein AVL-Baum entarten?

Sei $N(h)$ die minimale Anzahl der Knoten in einem AVL-Baum der Höhe h .

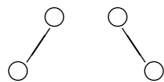
Ein AVL-Baum läßt sich folgendermaßen rekursiv definieren:

- $h = 0$: Der Baum besteht nur aus der Wurzel und es gilt

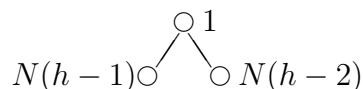
$$N(0) = 1$$

- $h = 1$: Da wir minimal gefüllte Bäume betrachten enthält die nächste Ebene nur einen Knoten, und es gilt

$$N(1) = 2$$



- $h \geq 2$: Methode: kombiniere 2 Bäume der Höhe $h - 1$ und $h - 2$ zu einem neuen, minimal gefüllten Baum mit neuer Wurzel:



Rekursionsgleichung für $N(h)$:

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

wie Fibonacci (exakt nachrechnen), wobei $Fib(n)$ die Fibonacci-Zahlen sind (siehe Abschnitt 1.2.5):

$$N(h) = Fib(h + 3) - 1$$

In Worten: Ein AVL-Baum der Höhe h hat mindestens $Fib(h + 3) - 1$ Knoten. Also gilt für die Höhe h eines AVL-Baums mit n Knoten:

$$N(h) \leq n < N(h + 1)$$

Beachte: Damit ist h exakt festgelegt.

Um von der Anzahl der Knoten n auf die *maximale Höhe* des AVL-Baumes zu schließen, formen wir die Gleichung noch nach h um.

Mit $\Phi = \frac{1 + \sqrt{5}}{2}$ und $\phi = \frac{1 - \sqrt{5}}{2}$ gilt:

$$Fib(i) = \frac{1}{\sqrt{5}} (\Phi^i - \phi^i) .$$

Einsetzen:

$$\begin{aligned} Fib(h + 3) &= \frac{1}{\sqrt{5}} [\Phi^{h+3} - \phi^{h+3}] \\ &\geq \frac{1}{\sqrt{5}} \Phi^{h+3} - \frac{1}{2} \end{aligned}$$

Und mit $Fib(h+3) \leq n+1$ folgt dann

$$\begin{aligned} \frac{1}{\sqrt{5}}\Phi^{h+3} &\leq n + \frac{3}{2} \\ \log_{\Phi} \left(\frac{1}{\sqrt{5}} \right) + h + 3 &\leq \log_{\Phi} \left[n + \frac{3}{2} \right] \\ h &\leq \log_{\Phi} n + \text{const} \\ &= (\ln 2 / \ln \Phi) \text{ld } n + \text{const} \\ &= 1.4404 \text{ld } n + \text{const} \end{aligned}$$

Also ist ein AVL-Baum maximal um 44% höher als ein vollständig ausgeglichener binärer Suchbaum.

Operationen auf AVL-Bäumen und ihre Komplexität

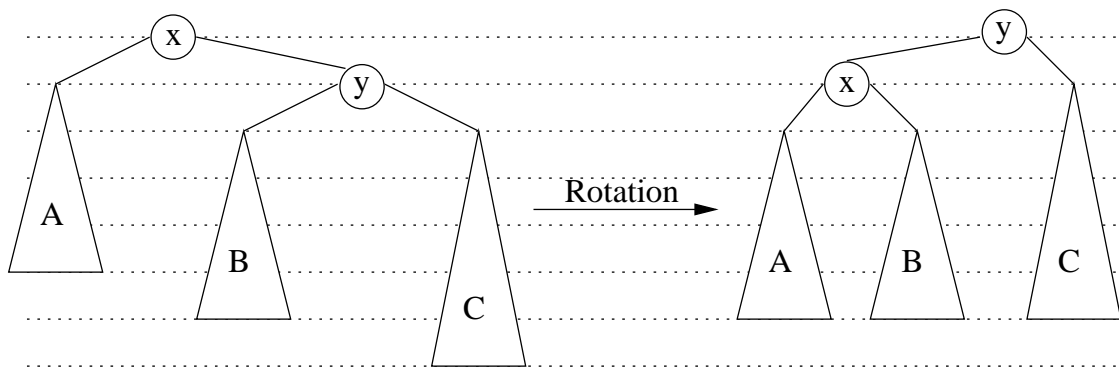
Nun wollen wir uns den Operationen auf AVL-Bäumen zuwenden. Dabei können zunächst die normalen Operationen wie gehabt ausgeführt werden, jedoch muß im Anschluß die Balance überprüft, und gegebenenfalls wiederhergestellt werden.

Die Balance-Überprüfung muß *rückwärts* auf dem ganzen Pfad vom veränderten Knoten bis zur Wurzel durchgeführt werden. Wir haben aber gesehen, daß dieser Pfad maximal eine Länge $\cong 1.44 \cdot \text{ld } n$ haben kann. Die Überprüfung hat also eine Komplexität $O(\text{ld } n)$.

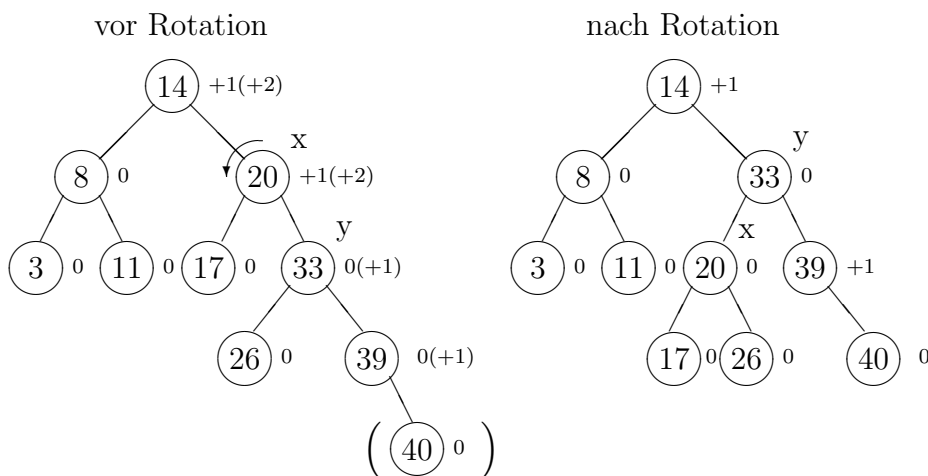
Ist die Balance an einer Stelle gestört, so kann sie in konstanter Zeit ($O(1)$) mittels einer *Rotation* oder *Doppel-Rotation* wiederhergestellt werden.

Ein AVL-Baum ist in einem Knoten aus der Balance, falls die beiden Teilbäume dieses Knotens einen Höhenunterschied größer eins aufweisen.

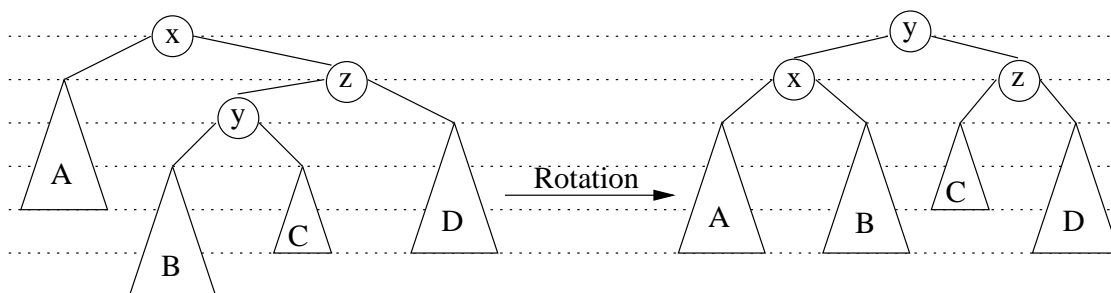
- **Rotation:** Ist ein Knoten betroffen (x), so betrachtet man nur den von ihm induzierten Teilbaum. Eine einfache Rotation muß durchgeführt werden, wenn der betroffene (also höhere) Teilbaum (C) *außen* liegt. Dann rotiert der betroffene Knoten zum kürzeren Teilbaum hinunter und übernimmt den inneren Sohn (B) des heraufrotierenden Knotens (y) als inneren Sohn.



Zur Veranschaulichung ein Beispiel: In den folgenden AVL-Baum sei das Element **40** eingefügt worden, wodurch die AVL-Bedingung im Element **20** verletzt wurde.

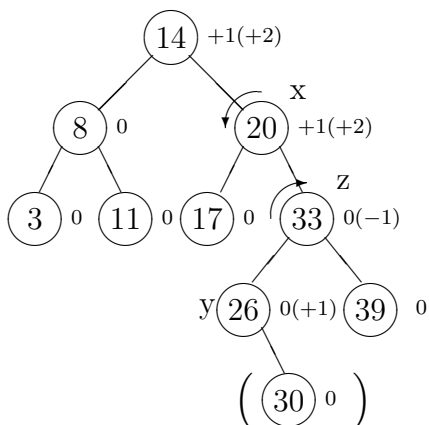


- Doppel-Rotation:** Eine Doppel-Rotation muß durchgeführt werden, falls der betroffene Teilbaum (mit Wurzel y) *innen* liegt, denn dann kann eine einfache Rotation das Ungleichgewicht auch nicht auflösen. Es wird zunächst eine Außen-Rotation im Vaterknoten der Wurzel des betroffenen Teilbaumes (z) durchgeführt, und anschließend eine Rotation in entgegengesetzter Richtung im Vaterknoten dieses Knotens (x).

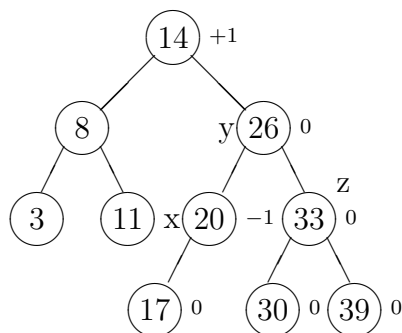


Zur Veranschaulichung wieder ein Beispiel: In den obigen AVL-Baum sei diesmal das Element **30** (in den Teilbaum B_2) eingefügt worden, wodurch die AVL-Bedingung wieder im Element **20** (x) verletzt wurde, jedoch ist diesmal der innere Teilbaum betroffen.

vor der Doppel-Rotation



nach Doppel-Rotation



Anmerkung: Es müssen insgesamt nur vier Pointer umdirigiert werden, um eine Rotation bzw. Doppel-Rotation durchzuführen. Dabei bleibt die Suchbaum-Ordnung erhalten.

Für die Komplexitäten haben wir also folgendes gesehen.

- Zeitkomplexität $O(\lg n)$ für die drei Standard-Operationen, und
- Platzkomplexität $O(n)$.

3.5.2 (a,b) -Bäume

Prinzip eines (a,b) -Baums: „Bei einem (a,b) -Baum haben alle Blätter gleiche Tiefe; die Zahl der Söhne eines Knotens liegt zwischen a und b .“ Achtung: Definitionen variieren leicht!

Definition: [Mehlhorn]: Seien $a, b \in \mathbb{N}$ mit $a \geq 2$ und $2a - 1 \leq b$. $\rho(v)$ sei die Anzahl der Söhne des Knotens v . Ein Baum heißt (a,b) -Baum, wenn

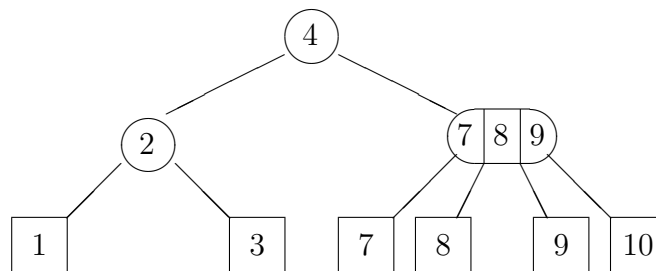
1. alle Blätter die gleiche Tiefe haben,
2. für jeden Knoten v gilt $\rho(v) \leq b$,
3. für alle Knoten v außer der Wurzel gilt $a \leq \rho(v)$, und
4. die Wurzel mindestens zwei Söhne hat.

Gilt $b = 2a - 1$, dann spricht man von einem B -Baum der Ordnung b .

Bei der Umsetzung gibt es noch zwei verschiedene Spielarten, die *blattorientierte Speicherung*, bei der alle Elemente nur in Blättern eingetragen werden und die inneren Knoten nur als Wegweiser dienen, und die herkömmliche Speicherung, bei der auch in den Knoten selbst Elemente abgelegt werden.

Wir wollen hier nur die blattorientierte Speicherung betrachten.

Beispiel: $(2,4)$ -Baum für $S = \{1, 3, 7, 8, 9, 10\} \subseteq \mathbb{N}$



Speicherung einer Menge als (a,b) -Baum

[Mehlhorn, Seiten 189ff.]

Sei $S = \{x_1 < \dots < x_n\} \subseteq U$ eine geordnete Menge und T ein vorgegebener leerer (a,b) -Baum mit n Blättern. Dann wird S in T wie folgt gespeichert:

1. Die Elemente von S werden in den Blättern w von T entsprechend ihrer Ordnung von links nach rechts gespeichert
2. Jedem Knoten v werden $\rho(v) - 1$ Elemente $k_1(v) < k_2(v) < \dots < k_{\rho(v)-1}(v) \in U$ so zugeordnet, daß für alle Blätter w im i -ten Unterbaum von v mit $1 < i < \rho(v)$ gilt

$$k_{i-1}(v) < \text{Inhalt}[w] \leq k_i(v)$$

Entsprechend gilt für die Randblätter w (also $i = 1$ bzw. $i = \rho(v)$) eines jeden Knotens:

- Befindet sich w im ersten Unterbaum eines Knoten, d.h. $i = 1$, so gilt

$$\text{Inhalt}[w] \leq k_1(v)$$

- befindet sich w im letzten Unterbaum eines Knoten, d.h. $i = \rho(v)$, gilt

$$k_{\rho(v)-1}(v) < \text{Inhalt}[w]$$

Für einen (a, b) -Baum mit n Blättern und Höhe h läßt sich folgendes ableiten:

$$\begin{aligned} 2a^{h-1} &\leq n \leq b^h \\ \log_b n &\leq h \leq 1 + \log_a \frac{n}{2} \end{aligned}$$

Speicherausnutzung: Wie man schon dem Beispiel entnehmen kann, muß jeder Knoten $(2b - 1)$ Speicherzellen besitzen, die sich unterteilen in b Zeiger zu den Söhnen und $b - 1$ Schlüssel (Elemente).

Die Speicherausnutzung beträgt bei einem (a, b) -Baum entsprechend mindestens $(2a - 1)/(2b - 1)$, z.B. für einen $(2, 4)$ -Baum $3/7$.

Standard-Operationen:

- $\text{Search}(y, S)$: Pfad von der Wurzel zum Blatt auswählen über die Elemente $k_1(v), \dots, k_{\rho(v)-1}(v)$ an jedem Knoten v („Wegweiser“).
- Nach einer $\text{Insert}(y, S)$ -Operation ist eventuell wiederholtes Spalten von Knoten notwendig, falls diese zu voll geworden sind.
- Nach einer $\text{Delete}(y, S)$ -Operation muß man u.U. Knoten *verschmelzen* oder *stehlen* um den Baum zu rebalancieren.

Insgesamt liegt die Zeitkomplexität für alle drei Operationen sowohl für den Worst case als auch für den Average case bei $O(\log n)$ mit $n = |S|$.

Typische (a,b) -Bäume

Aus der Optimierung der Zugriffszeit ergeben sich grob zwei Varianten, die oft anzutreffen sind, und in Abhängigkeit der Speicherung der Daten auf Hintergrundmedien (Festplatte, Band, CD-ROM) oder im Hauptspeicher ihre Vorzüge haben.

- Bei der *internen Speicherung*: $(2,4)$ -Bäume und $(2,3)$ -Bäume .
- Bei der *externen Speicherung*, bei der man von einer langen Zugriffszeit ausgeht, verwendet man vorwiegend B -Bäume hoher Ordnung (zwischen 100 und 1000), die man mit der Zugriffszeit steigen läßt (lange Zugriffszeit \rightarrow hohe Ordnung). Denn es gilt, daß die Anzahl der Zugriffe proportional zur Höhe des Baumes wächst.

Beispiel: Sei $n = 10^6$ die Anzahl der Blätter, dann gilt für die Höhe und entsprechend die Anzahl der Zugriffe auf einen binären Suchbaum

$$h = \text{ld } n \cong 20$$

und für einen B-Baum mit $a = 100$

$$h \leq 1 + \lceil \log_a(n/2) \rceil = 1 + 3 = 4 .$$

Das entspricht also einer Verringerung der Plattenzugriffe von 20 auf 4.

3.6 Priority Queue und Heap

Priority Queues sind Warteschlangen, bei denen die Elemente nach Priorität geordnet sind. Im Gegensatz dazu gilt bei normalen Queues das FIFO-Prinzip. Bei Priority Queues können Elemente nur am Kopf entfernt und am Schwanz eingefügt werden. Jedoch verlangt die Ordnung, daß am Kopf immer die Elemente mit höchster Priorität stehen. Also muß es eine Funktion geben, die es neu eingefügten Elementen hoher Priorität erlaubt, in der Warteschlange weiter nach vorne zu wandern.

Priority Queues findet man in Betriebssystemen beim *process scheduling*. Verschiedene Prozesse haben für das Betriebssystem unterschiedliche Prioritäten und sollen deshalb auch in der entsprechenden Reihenfolge abgearbeitet werden.

Beschreibung:

- nach Priorität geordnete Menge S
- *primäre Operationen*:
 - $\text{Insert}(x, S)$
 - $\text{DeleteMin}(x, S)$: Entfernen des Elements mit der höchsten Priorität.

- weitere Operationen:
 - Initialize(S)
 - ReplaceMin(x, S)
 - und unter Umständen Delete(x, S), Search(x, S)

Datenstruktur:

- AVL-Baum oder
- Heap (wie bei HeapSort): „partiell geordneter, links-vollständiger Baum“:
Partiell geordneter Binärbaum, bei dem in jedem Knoten das Minimum (Maximum) des jeweiligen Teilbaumes steht.

Zwei Standardoperationen (vgl. HeapSort):

- **Insert**(y, S): Fügt Element y an der letzten Stelle im Heap-Array ein und bewegt es aufwärts (überholen), bis es die seiner Priorität entsprechende Position erreicht hat (Prozedur *Upheap*).
- **DeleteMin**(S): Entfernt das Element in der Wurzel und setzt dort das letzte Element des Arrays ein, um es dann mittels *DownHeap* (wie bei HeapSort) an die richtige Position zurückzuführen.

Von HeapSort wissen wir, daß die **Komplexität** für beide Operationen bei maximal $2 \cdot \lg n$ Vergleichen liegt.

Übung: Warum wird die Heap-Datenstruktur nicht zur Mengendarstellung mit den drei Standard-Operationen *Search*, *Insert* und *Delete* verwendet?

Antwort: *Search* ist problematisch.