Effiziente Algorithmen

Martin Hofmann und Jan Johannsen

Institut für Informatik

LMU München

Sommersemester 2002

Binäre Suchbäume

Repräsentation binärer Bäume:

Jeder Knoten x besteht aus:

$key[x]$	Schlüssel des gespeicherten Datums	
$left[x]$	Zeiger auf den linken Teilbaum	(NIL, falls x Blatt ist)
$right[x]$	rechten	
$p[x]$	Zeiger auf den Vorgänger	(NIL, falls x die Wurzel ist)

+ möglicherweise weitere Daten.

Baum T: Zeiger $root[T]$ auf die Wurzel (NIL, falls T leer ist.)

Eigenschaft binärer Suchbäume:

Für jeden Knoten x gilt:

- für alle y im linken Teilbaum von x ist $key[y] \leq key[x]$,
- für alle y im rechten Teilbaum von x ist $key[y] \geq key[x]$.

Suche in binären Suchbäumen

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2       then return x
3  if k < key[x]
4       then return TREE-SEARCH(left[x], k)
5       else return TREE-SEARCH(right[x], k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2       do if k < key[x]
3            then x ← left[x]
4            else x ← right[x]
5  return x
```

Vorgänger und Nachfolger

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2        then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y ; y ← p[y]
6  return y
```

Analog arbeiten TREE-MAXIMUM und TREE-PREDECESSOR.

Einfügen in binäre Suchbäume

Einfügen von Schlüssel k: Bilde z mit $key[z] = k$, $p[z] = left[z] = right[z] = \text{NIL}$

```
TREE-INSERT(T, z)
 1  y ← NIL ; x ← root[T]
 2  while x ≠ NIL
 3        do  y ← x
 4            if key[z] < key[x]
 5               then x ← left[x]
 6               else x ← right[x]
 7  p[z] ← y
 8  if y = NIL
 9        then root[T] ← z
10        else if key[z] < key[y]
11                  then left[y] ← z
12                  else right[y] ← z
```

Löschen aus binären Suchbäumen

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2      then y ← z
 3      else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5      then x ← left[y]
 6      else x ← right[y]
 7  if x ≠ NIL
 8      then p[x] ← p[y]
 9  if p[y] = NIL
10      then root[T] ← x
11      else if y = left[p[y]]
12              then left[p[y]] ← x
13              else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
```

Insgesamt: Die dynamischen Mengenoperationen SEARCH, INSERT und DELETE können für binäre Suchbäume mit $O(h)$ Operationen realisiert werden.

Hier bezeichnet h die Höhe des Baumes.

Durchschnittliche Höhe binärer Suchbäume I

Wir betrachten zufällig aufgebaute binäre Suchbäume:

Schlüssel $k_1, k_2, \ldots, k_n$ werden nacheinander mit TREE-INSERT eingefügt.

Jede der $n!$ möglichen Anordnungen der k_i soll gleich wahrscheinlich sein.

Lemma: k_i ist Vorfahre von k_j $(i < j)$, genau dann, wenn

$$k_i = \min\{k_\ell \ ; \ \ell \leq i \text{ und } k_\ell > k_j\} \tag{1}$$

oder

$$k_i = \max\{k_\ell \ ; \ \ell \leq i \text{ und } k_\ell < k_j\} \tag{2}$$

Definiere:

$$G_j := \{k_i \ ; \ i < j \text{ und } k_\ell > k_i > k_j \text{ für alle } \ell < i \text{ mit } k_\ell > k_j\}$$
$$L_j := \{k_i \ ; \ i < j \text{ und } k_\ell < k_i < k_j \text{ für alle } \ell < i \text{ mit } k_\ell < k_j\}$$

Dann sind $G_j \cup L_j$ die Vorfahren von k_j, also ist die Tiefe $d(k_j, T)$ von k_j in T

$$d(k_j, T) = |G_j| + |L_j| \ .$$

Chernoff-Ungleichung

Es sei

$$X = \sum_{i=1}^{n} X_i \text{ wobei } X_i = \begin{cases} 1 & \text{mit Wahrscheinlichkeit } p_i \\ 0 & \text{mit Wahrscheinlichkeit } 1 - p_i \end{cases}$$

und $\mu := E[X]$ der Erwartungswert von X. Dann gilt für $\delta > 0$:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

Anwendung: Sei $S := \{ k_i \; ; \; k_\ell > k_i \text{ für alle } \ell < i \}$

$\Pr[k_i \in S] = \frac{1}{i}$. Also $E[|S|] = \sum_{i=1}^{n} \frac{1}{i} =: H_n$. Bekannt: $H_n = \ln n + O(1)$

Die Chernoff-Ungleichung liefert:

$$\Pr[|S| \geq 4H_n] \leq \left(\frac{e^3}{256}\right)^{H_n} \leq \left(\frac{1}{e^2}\right)^{H_n} \leq \left(\frac{1}{e^2}\right)^{\ln n} = \frac{1}{n^2}$$

Durchschnittliche Höhe binärer Suchbäume II

Satz: Die durschnittliche Höhe eines zufällig aufgebauten binären Suchbaumes mit n Knoten ist $O(\log n)$.

Beweis:
$$\Pr\big[d(k_j,T) \geq t\big] \leq \Pr\big[|G_j| \geq t/2\big] + \Pr\big[|L_j| \geq t/2\big]$$

$$\Pr\big[|G_j| \geq t/2\big] \leq \Pr\big[|S| \geq t/2\big]$$

Symmetrisch:
$$\Pr\big[|L_j| \geq t/2\big] \leq \Pr\big[|S| \geq t/2\big]$$

Also:
$$\Pr\big[d(k_j,T) \geq t\big] \leq 2\Pr\big[|S| \geq t/2\big]$$

Für $t = 8H_n$:
$$\Pr\big[d(k_j,T) \geq 8H_n\big] \leq 2\Pr\big[|S| \geq 4H_n\big] \;\leq\; 2/n^2$$

Insgesamt:
$$\Pr\big[\mathit{depth}(T) \geq 8H_n\big] \leq \sum_{j=1}^{n} \Pr\big[d(k_j,T) \geq 8H_n\big] \leq n \cdot (2/n^2) = 2/n$$

also
$$E[\mathit{depth}(T)] \;\leq\; 8H_n \cdot (1-(2/n)) + n \cdot (2/n) \;=\; O(\log n)\ .$$

Rot-Schwarz-Bäume

Rot-Schwarz-Bäume: binäre Suchbäume mit zusätzlicher Struktur, die dafür sorgt, dass die Höhe im *worst-case* $O(\log n)$ ist.

Jeder Knoten wird als innerer Knoten aufgefasst, Blätter sind zusätzliche, leere Knoten (Wächter).

Knoten haben ein zusätzliches Feld $color[x]$, mit den Werten RED und BLACK. Wir sprechen von schwarzen und roten Knoten.

Rot-Schwarz-Eigenschaft:

1. Jedes Blatt ist schwarz.
2. Beide Söhne eines roten Knotens sind schwarz.
3. Für jeden Knoten x gilt: jeder Pfad von x zu einem Blatt enthält gleich viele schwarze Knoten.

Eigenschaften von Rot-Schwarz-Bäumen

Lemma:

Die Höhe eines Rot-Schwarz-Baums mit n inneren Knoten ist höchstens $2\log_2(n+1)$.

Daraus folgt:

Die Operationen Tree-Search, Tree-Minimum, Tree-Maximum, Tree-Successor und Tree-Predecessor benötigen für Rot-Schwarz-Bäume $O(\log n)$ Operationen.

Auch Tree-Insert und Tree-Delete laufen auf Rot-Schwarz-Bäumen in Zeit $O(\log n)$, erhalten aber nicht die Rot-Schwarz-Eigenschaft.

$\rightsquigarrow$ Für diese benötigen wir spezielle Einfüge- und Löschoperationen.

Rotation

LEFT-ROTATE(T, x)

```
 1  y ← right[x]
 2  right[x] ← left[y]
 3  if left[y] ≠ NIL
 4        then p[left[y]] ← x
 5  p[y] ← p[x]
 6  if p[x] = NIL
 7        then root[T] ← y
 8        else if x = left[p[x]]
 9                 then left[p[x]] ← y
10                 else right[p[x]] ← y
11  left[y] ← x
12  p[x] ← y
```

RIGHT-ROTATE(T, x)

Analog mit *left* und *right* vertauscht

LEFT-ROTATE und RIGHT-ROTATE haben konstante Laufzeit $O(1)$.

Einfügen in einen Rot-Schwarz-Baum

- Neuer Knoten x wird mit TREE-INSERT eingefügt und rot gefärbt.
- Problem: Vater $p[x]$ kann auch rot sein $\quad\leadsto\quad$ Eigenschaft 2 verletzt.
- In einer Schleife wird diese Inkonsistenz durch lokale Änderungen im Baum nach oben geschoben.
- Fall 1: $p[x] = \mathit{left}[p[p[x]]]$:
  - Betrachte x' Onkel $y = \mathit{right}[p[p[x]]]$.
  - Fall 1a: y ist rot. Dann färbe $p[x]$ und y schwarz, und $p[p[x]]$ rot.
    Aber: $p[p[p[x]]]$ kann rot sein $\quad\leadsto\quad$ Inkonsistenz weiter oben.
  - Fall 1b: y ist schwarz, und $x = \mathit{left}[p[x]]$. Dann wird $p[x]$ schwarz und $p[p[x]]$ rot gefärbt, und anschliessend um $p[p[x]]$ nach rechts rotiert.
    $\leadsto\quad$ keine Inkonsistenz mehr.
  - Fall 1c: y ist schwarz, und $x = \mathit{right}[p[x]]$. Wird durch eine Linksrotation um $p[x]$ auf Fall 1b zurückgeführt.
- Fall 2: $p[x] = \mathit{right}[p[p[x]]]$ ist symmetrisch.

Einfügen in einen Rot-Schwarz-Baum

```
RB-Insert(T, x)
 1  Tree-Insert(T, x)
 2  color[x] ← red
 3  while x ≠ root[T] and color[p[x]] = red
 4    do if p[x] = left[p[p[x]]]
 5          then y ← right[p[p[x]]]
 6               if color[y] = red
 7                  then color[p[x]] ← black
 8                       color[y] ← black
 9                       color[p[p[x]]] ← red
10                       x ← p[p[x]]
11          else if x = right[p[x]]
12                  then x ← p[x]
13                       Left-Rotate(T, x)
14               color[p[x]] ← black
15               color[p[p[x]]] ← red
16               Right-Rotate(T, p[p[x]])
17    else  ...
18  color[root[T]] ← black
```

Zeile 17: **else** zum **if** in Zeile 4, symmetrisch zum **then**-Teil (Zeilen 5-16) mit *left* und *right* vertauscht.

Löschen aus einem Rot-Schwarz-Baum

- Zu löschender Knoten z wird mit TREE-DELETE gelöscht.
- Problem: Wenn der entfernte Knoten y schwarz war, ist Eigenschaft 3 verletzt.
- Intuition: der (einzige) Sohn x von y erbt dessen schwarze Farbe, und ist jetzt "doppelt schwarz".
- Das zusätzliche Schwarz wird in einer Schleife durch lokale Änderungen im Baum nach oben geschoben.
- Ist x rot, so wird es schwarz gefärbt, und die Schleife bricht ab. Andernfalls unterscheide zwei Fälle:
- Fall 1: $x = \mathit{left}[p[x]]$.
  - Betrachte x' Bruder $w = \mathit{right}[p[x]] \neq$ NIL, und unterscheide 2 Fälle:
  - Fall 1.1: w ist rot. Dann muss $p[x]$ schwarz sein, also färbe $p[x]$ rot, w schwarz und rotiere links um $p[x]$
    $\rightsquigarrow$ reduziert auf Fall 1.2.

Löschen aus einem Rot-Schwarz-Baum

- Fall 1.2: w ist schwarz. Es gibt drei weitere Fälle:
  - Fall 1.2.1: Beide Kinder von w sind schwarz. Also können wir w rot färben, und das zusätzliche Schwarz von x zu $p[x]$ versetzen
    $\rightsquigarrow$ nächste Iteration.
  - Fall 1.2.2: $\mathit{left}[w]$ ist rot, $\mathit{right}[w]$ ist schwarz. Dann färbe w rot, $\mathit{left}[w]$ schwarz und rotiere rechts um w
    $\rightsquigarrow$ reduziert auf Fall 1.2.3.
  - Fall 1.2.3: $\mathit{right}[w]$ ist rot. Dann vertausche die Farben von w und $p[x]$, färbe $\mathit{right}[w]$ schwarz, und rotiere links um $p[x]$
    $\rightsquigarrow$ Zusätzliches Schwarz ist verbraucht.

- Fall 2: $x = \mathit{right}[p[x]]$ ist symmetrisch.

Zusammengefasst: die Operationen Search, Minimum, Maximum, Successor, Predecessor, Insert und Delete können für Rot-Schwarz-Bäume mit Laufzeit $O(\log n)$ realisiert werden.

B-Bäume

Definition: Ein B-Baum T ist ein Baum mit den Eigenschaften:

1. Jeder Knoten x hat die folgenden Felder:
   (a) die Zahl $n[x]$ der in x gespeicherten Schlüssel,
   (b) die $n[x]$ Schlüssel $key_1[x] \leq key_2[x] \leq \ldots \leq key_{n[x]}[x]$,
   (c) ein flag $leaf[x]$, dass den Wert TRUE hat wenn x ein Blatt ist.
2. Ein innerer Knoten x enthält $n[x]+1$ Zeiger $c_1[x], \ldots, c_{n[x]+1}[x]$ auf seine Söhne.
3. Sind die Schlüssel k_i für $1 \leq i \leq n[x]+1$ im Unterbaum mit der Wurzel $c_i[x]$ gespeichert, so gilt
$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq k_{n[x]} \leq key_{n[x]}[x] \leq k_{n[x]+1} \ .$$
4. Alle Blätter haben die gleiche Tiefe, die Höhe h des Baumes.
5. Es gibt eine Zahl t, den Minimalgrad, derart dass $n[x] \geq t-1$ für jeden Knoten $x \neq root[T]$, und $n[x] \leq 2t-1$. Ein Knoten mit $n[x] = 2t-1$ heißt voll.

Tiefe von B-Bäumen

Theorem: Für einen B-Baum mit $n \geq 1$ gespeicherten Schlüsseln, Höhe h und Minimalgrad t gilt:

$$h \leq \log_t \frac{n+1}{2}$$

Beweis: Die Anzahl der Schlüssel n ist minimal, falls $n[root[T]] = 1$ und $n[x] = t - 1$ für alle übrigen Knoten x.

$\rightsquigarrow$ Es gibt $2t^{i-1}$ Knoten der Tiefe i für $1 \leq i \leq h$. Also gilt:

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &= 2t^h - 1 \end{aligned}$$

Suchen in B-Bäumen

```
B-Tree-Search(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3     do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else Disk-Read(c_i[x])
9          return B-Tree-Search(c_i[x], k)
```

B-Tree-Search(x, k) braucht Zeit $O(th) = O(t \log_t n)$ und $O(h) = O(\log_t n)$ Plattenzugriffe.

Aufspalten eines Knoten in einem B-Baum

B-Tree-Split-Child nimmt einen nicht-vollen inneren Knoten x und einen vollen Sohn y, die bereits im Hauptspeicher gehalten werden, sowie i mit $c_i[x] = y$, und spaltet y auf.

B-Tree-Split-Child(x, i, y)

1 Erzeuge neuen Knoten z ; $leaf[z] \leftarrow leaf[y]$

2 kopiere $key_{t+1}[y] \ldots key_{2t-1}[y]$ nach $key_1[z] \ldots key_{t-1}[z]$

3 **if not** $leaf[y]$ **then** kopiere $c_{t+1}[y] \ldots c_{2t}[y]$ nach $c_1[z] \ldots c_t[z]$

4 $n[y] \leftarrow t - 1$; $n[z] \leftarrow t - 1$

5 schiebe $c_{i+1}[x] \ldots c_{n[x]+1}$ nach $c_{i+2}[x] \ldots c_{n[x]+2}[x]$

6 $c_{i+1}[x] \leftarrow z$

7 schiebe $key_{i+1}[x] \ldots key_{n[x]}$ nach $key_{i+2}[x] \ldots key_{n[x]+1}[x]$

8 $key_i[x] \leftarrow key_t[y]$

9 $n[x] \leftarrow n[x] + 1$

10 schreibe x, y, z auf die Platte zurück

Einfügen in einen B-Baum

B-Tree-Insert(x, k) fügt Schlüssel k in Teilbaum mit Wurzel x ein.
Erwartet, dass x nicht voll ist, und sorgt auch beim rekursiven Aufruf dafür.

```
B-Tree-Insert(x, k)
 1  finde das größte i ≤ n[x] mit k ≥ key_i[x]
 2  if leaf[x]
 3     then schiebe key_{i+1}[y] ... key_{n[x]}[y] nach key_{i+1}[x] ... key_{n[x]+1}[x]
 4          n[x] ← n[x] + 1
 5          Disk-Write(x)
 6     else i ← i + 1 ; Disk-Read(c_i[x])
 7          if n[c_i[x]] = 2t − 1
 8             then B-Tree-Split-Child(x, i, c_i[x])
 9                  if k > key_i[x] then i ← i + 1
10          B-Tree-Insert(c_i[x], k)
```

Ist beim ersten Aufruf die Wurzel r voll, so erzeuge neue Wurzel r', hänge r als einzigen Sohn an: $c_1[r'] \leftarrow r$, und rufe B-Tree-Split-Child$(r', 1, r)$ auf.

Löschen aus einem B-Baum

B-Tree-Delete(x, k) löscht Schlüssel k aus dem Teilbaum mit Wurzel x. Erwartet, dass $n[x] \geq t$ ist, und sorgt auch beim rekursiven Aufruf dafür.

1. Ist x ein Blatt, und k in x gespeichert, so lösche k aus x.
2. Ist x innerer Knoten, und $k = key_i[x]$, so gibt es drei Fälle:
   (a) Sei $y := c_i[x]$. Ist $n[y] \geq t$, so finde in y den Vorgänger k' von k, lösche diesen rekursiv, und ersetze k in x durch k'.
   (b) Sei $z := c_{i+1}[x]$. Ist $n[z] \geq t$, so finde in z den Nachfolger k' von k, lösche diesen rekursiv, und ersetze k in x durch k'.
   (c) Andernfalls bilde aus y, k und z einen neuen Knoten y', setze $c_i[x] \leftarrow y'$ und entferne k und z aus x. Lösche dann rekursiv k aus y'.
3. Ist x innerer Knoten, und k ist nicht in x, so finde i mit $key_i[x] \leq k \leq key_i[x]$, und setze $y = c_i[x]$. Falls $n[y] = t - 1$ ist, so führe (a) oder (b) aus. Lösche dann k rekursiv aus y.
   (a) Gibt es einen Bruder z von y mit $n[z] \geq t$, so schiebe einen Schlüssel von x in y, einen von z in x, und den entsprechenden Sohn von z nach y.
   (b) Haben beide Brüder von y nur $t - 1$ Schlüssel, so verschmelze einen davon mit y, analog zu Fall 2(c).

Dynamische Programmierung

Optimierungsprobleme:

- Zu jeder Eingabe x mehrere mögliche Lösungen $y \in S(x)$.
- Lösungen versehen mit einer Wertfunktion $v(y) \in \mathbb{R}$.
- Gesucht optimale Lösung $y \in S(x)$ mit:

  $v(y) = \max\{v(z)\ ;\ z \in S(x)\}$ (v Nutzen) bzw.
  $v(y) = \max\{v(z)\ ;\ z \in S(x)\}$ (v Kosten)

Dynamische Programmierung: Entwurfsprinzip, (hauptsächlich) für Algorithmen für Optimierungsprobleme.

- Anwendbar, falls optimale Lösungen rekursiv aus optimalen Lösungen kleinerer Teilprobleme zusammengesetzt sind.
- Vorteilhaft, wenn Teilprobleme nicht unabhängig sind, sondern ihrerseits überlappende Teilprobleme besitzen.

$\rightsquigarrow$ Bottom-Up-Berechnung der optimalen Lösungen größerer Probleme aus denen kleinerer, die dabei in einer Tabelle gespeichert werden.

Vorgehensweise bei Dynamischer Programmierung

1. Charakterisierung der möglichen Lösungen und der Struktur optimaler Lösungen.

2. Rekursive Definition, wie sich der Wert einer optimalen Lösung aus den Werten kleinerer Teillösungen zusammensetzt.

3. Bottom-up Berechnung des Wertes einer optimalen Lösung.

4. Speicherung zusätzlicher Information zum tatsächlichen Auffinden einer optimalen Lösung.

Einfaches Beispiel: Matrizen-Kettenmultiplikation

Problem: n Matrizen $M_1, \ldots, M_n$ sollen multipliziert werden, wobei M_i eine $n_i \times n_{i+1}$-Matrix ist. Aufwand hängt von der Klammerung ab.

Bezeichne $M_{i..j}$ das Produkt $M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$, und $m[i,j]$ die minimale Zahl von Multiplikationen zum Berechnen von $M_{i..j}$.

1. Optimale Klammerung von $M_{1..n}$ ist von der Form $M_{1..k} \cdot M_{k+1..n}$, für optimale Klammerungen von $M_{1..k}$ und $M_{k+1..n}$.
2. Deshalb gilt:

$$(*) \qquad m[i,j] = \begin{cases} 0 & \text{falls } i = j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + n_i n_{k+1} n_{j+1} & \text{sonst.} \end{cases}$$

3. Fülle eine Tabelle, in der unten die $m[i,i] = 0$ stehen, und der h-ten Ebene die Werte $m[i,j]$ mit $j - i = h$. An der Spitze steht schließlich $m[1,n]$.
4. Speichere in jedem Schritt auch einen Wert $s[i,j]$, nämlich dasjenige k, für das in $(*)$ das Minimum angenommen wird.

Längste gemeinsame Teilfolge (lgT)

Definition: $Z = \langle z_1, z_2, \ldots, z_m \rangle$ ist Teilfolge von $X = \langle x_1, x_2, \ldots, x_n \rangle$,
falls es Indizes $i_1 < i_2 < \ldots < i_m$ gibt mit $z_j = x_{i_j}$ für $j \leq m$.

Problem: Gegeben $X = \langle x_1, x_2, \ldots, x_m \rangle$ und $Y = \langle y_1, y_2, \ldots, y_n \rangle$

Ziel: finde $Z = \langle z_1, z_2, \ldots, z_k \rangle$ maximaler Länge k, das Teilfolge von X und Y ist.

Abkürzung: $X_i := \langle x_1, x_2, \ldots, x_i \rangle$ ist das i-te Präfix von X.

Lösung mit dynamischer Programmierung:

1. Sei Z lgT von X und Y. Dann gilt:
   (a) Ist $x_m = y_n$, so ist $z_k = x_m = y_n$, und Z_{k-1} ist LgT von X_{m-1} und Y_{n-1}.
   (b) Ist $x_m \neq y_n$, so gilt:
   - Ist $z_k \neq x_m$, dann ist Z LgT von X_{m-1} und Y.
   - Ist $z_k \neq y_n$, dann ist Z LgT von X und Y_{n-1}.

Längste gemeinsame Teilfolge (lgT)

2. Sei $c[i,j]$ die Länge einer lgT von X_i und Y_j. Es gilt:

$$c[i,j] = \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0, \\ c[i-1,j-1]+1 & \text{falls } i,j > 0 \text{ und } x_i = y_j \\ \max(c[i-1,j], c[i,j-1]) & \text{falls } i,j > 0 \text{ und } x_i \neq y_j \end{cases}$$

3. Fülle die Tabelle der $c[i,j]$ zeilenweise, jede Zeile von links nach rechts. Am Ende erhält man $c[m,n]$, die Länge einer lgT von X und Y.

4. Für $i,j \geq 1$, speichere einen Wert $b[i,j] \in \{\uparrow, \nwarrow, \leftarrow\}$, der anzeigt, wie $c[i,j]$ berechnet wurde:

$$\begin{aligned} \uparrow &: c[i,j] = c[i-1,j] \\ \nwarrow &: c[i,j] = c[i-1,j-1]+1 \\ \leftarrow &: c[i,j] = c[i,j-1] \end{aligned}$$

Damit läßt sich ein lgT aus der Tabelle ablesen.

Greedy-Algorithmen

Entwurfsprinzip für Algorithmen für Optimierungsprobleme.

- Lösung wird sukzessive aufgebaut.
- Bei jeder anstehenden Entscheidung wird die lokal optimale Wahl getroffen (engl. *greedy* = gierig, „nimm immer das größte Stück“).
- Folgende Bedingungen müssen erfüllt sein, damit dadurch eine optimale Lösung gefunden wird:
  1. Greedy Choice-Prinzip:

     Es gibt stets eine optimale Lösung, die mit der lokal optimalen Wahl konsistent ist.
  2. Optimale Teillösungen:

     Optimale Lösung enthält eine optimale Lösung des nach der lokal optimalen Wahl verbleibenden Teilproblems.

Einfaches Beispiel: ein Auswahlproblem

Gegeben: Menge $A = \{(s_1, e_1), \ldots, (s_n, e_n)\}$ mit $s_i \leq e_i \in \mathbb{R}^+$ für alle i.

Ziel: Finde $B \subseteq \{1, \ldots, n\}$ maximaler Größe mit $s_i \geq e_j$ oder $s_j \geq e_i$ für alle $i \neq j \in B$.

Annahme: A nach den e_i sortiert: $e_1 \leq e_2 \leq \cdots \leq e_n$.

Greedy-Algorithmus:

$B \leftarrow \{1\}$
$j \leftarrow 1$
for $i \leftarrow 2$ **to** n
  do if $s_i \geq e_j$
    then $B \leftarrow B \cup \{i\}$
      $j \leftarrow i$
return B

Satz: Der Greedy-Algorithmus findet stets eine optimale Lösung für das Auswahlproblem.

Huffman-Codes

Definition:

Für Wörter $v, w \in \{0,1\}^*$ heißt v ein Präfix von w, falls $w = vu$ für ein $u \in \{0,1\}^*$.

Ein Präfixcode ist eine Menge von Wörtern $C \subseteq \{0,1\}^*$ mit: für alle $v, w \in C$ ist v nicht Präfix von w.

Das Huffman-Code-Problem:

Gegeben: Alphabet $A = \{a_1, \ldots, a_n\}$ mit Häufigkeiten $h(a_i) \in \mathbb{R}_+$

Ziel: Finde Präfixcode $C = \{c_1, \ldots, c_n\}$ derart dass $\sum_{i=1}^{n} h(a_i) \cdot |c(a_i)|$ minimal wird.

Ein solches C heißt optimaler Präfixcode, oder Huffman-Code; diese werden bei der Datenkompression verwendet.

Der Huffman-Algorithmus

- Erzeuge Knoten $z_1, \ldots, z_n$ mit Feldern $B[z_i] \leftarrow a_i$ und $H[z_i] \leftarrow h(a_i)$.
- Speichere diese in einer Priority Queue Q mit Operationen INSERT und EXTRACT-MIN (beides realisierbar mit Laufzeit $O(\log n)$, z.B. mit Heap).
- Baue sukzessive einen binären Baum auf: Blätter sind die Knoten z_i, innere Knoten x haben nur ein Feld $H[x]$.
- Aufbau des Baumes nach dem folgenden Algorithmus:

  for $i \leftarrow 1$ **to** $n-1$
  do erzeuge neuen Knoten x
  $\mathit{left}[x] \leftarrow$ EXTRACT-MIN(Q)
  $\mathit{right}[x] \leftarrow$ EXTRACT-MIN(Q)
  $H[x] \leftarrow H[\mathit{left}[x]] + H[\mathit{right}[x]]$
  INSERT(Q, x)
  return EXTRACT-MIN(Q)

- Codewort c_i entspricht Pfad im Baum zu Knoten z_i.
- Laufzeit: $3(n-1) \cdot O(\log n) = O(n \log n)$.

Korrektheit des Huffman-Algorithmus

Satz: Der Huffman-Algorithmus findet stets einen optimalen Präfixcode.

Beweis: Identifiziere Präfixcodes mit Codebäumen.

1. Greedy Choice-Prinzip:

   Seien $a, b \in A$ mit $h(a), h(b) \leq h(c)$ für alle $c \in A \setminus \{a, b\}$.

   Dann gibt es einen optimalen Codebaum, in dem a und b in benachbarten Blättern maximaler Tiefe liegen.

2. Optimale Teillösungen:

   Sei T ein optimaler Codebaum, und seien a, b benachbarte Blätter maximaler Tiefe, und z deren Vater. Konstruiere Baum T', indem z durch ein Blatt mit neuem Buchstaben c ersetzt wird.

   Dann ist T' ein optimaler Codebaum für $A \setminus \{a, b\} \cup \{c\}$, mit $h(c) = h(a) + h(b)$.

Amortisierungs-Analyse

Prinzip zur Analyse von Operationen auf Datenstrukturen.

- Betrachte nicht einzelne Operationen, sondern Folgen von Operationen.
- Statt der (*worst-case*-) Kosten jeder einzelnen Operation werden die mittleren Kosten der Operationen in der Folge bestimmt.

  $\rightsquigarrow$ teure Operationen können sich durch nachfolgende billigere amortisieren.
- Kosten einiger Operationen werden zu hoch, andere zu niedrig veranschlagt.
- Drei Methoden:
  1. Aggregat-Methode:
     Berechne worst-case Kosten $T(n)$ einer Folge von n Operationen.
     Amortisierte Kosten: $T(n)/n$.
  2. Konto-Methode:
     Elemente der Datenstruktur haben ein Konto, auf dem zuviel berechnete Kosten gutgeschrieben werden. Zu niedrig veranschlagte Operationen werden aus dem Konto bezahlt.
  3. Potenzial-Methode:
     Ähnlich der Konto-Methode, aber gesamte zuviel berechnete Kosten werden als *potenzielle Energie* der Datenstruktur aufgefasst.

Einfaches Beispiel: Binärzähler

Datenstruktur: Array A von $length[A] = k$ Bits.

Einzige Operation:

INCREMENT(A)

```
1  i ← 0
2  while i < length[A] and A[i] = 1
3        do A[i] ← 0
4           i ← i + 1
5  if i < length[A]
6    then A[i] ← 1
```

Betrachte Folge von n INCREMENT-Operationen (Annahme: $k \geq \lceil \log_2(n+1) \rceil$).

Worst-case-Kosten: $\lceil \log_2(n+1) \rceil$ Wertzuweisungen.

Aggregat-Methode

Berechne Kosten $T(n)$ der ganzen Folge von n INCREMENT-Operationen:

- Beobachtung:

  $A[0]$ wird bei jedem INCREMENT verändert,
  $A[1]$ nur bei jedem zweiten,
  $A[2]$ nur bei jedem vierten,
  etc.

- Gesamtzahl der Wertzuweisungen bei n INCREMENT-Operationen:

$$T(n) \leq \sum_{i=0}^{\lceil \log_2(n+1) \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = 2n$$

- Also: $T(n) = O(n)$, somit amortisierte Kosten jedes INCREMENT:

$$O(n)/n = O(1)$$

Konto-Methode

Für die i-te Operation:

tatsächliche Kosten c_i amortisierte Kosten $\hat{c}_i$.

- Elemente der Datenstruktur haben Konten.
- Ist $\hat{c}_i > c_i$, so wird die Differenz $\hat{c}_i - c_i$ auf ein Konto gutgeschrieben.
- Ist $\hat{c}_i < c_i$, so muss die Differenz $c_i - \hat{c}_i$ aus einem Konto bezahlt werden.

  $\rightsquigarrow$ Die $\hat{c}_i$ müssen so gewählt sein, dass genug Guthaben auf den Konten ist.

Im Beispiel:

- Kosten einer Wertzuweisung seien 1 €.
- Jede Array-Position $j \leq k$ hat ein Konto.
- Veranschlage amortisierte Kosten eines INCREMENT: 2 €.
- Jedes INCREMENT setzt nur ein Bit $A[i] \leftarrow 1$ (Zeile 6):
  Zahle dafür 1 €, und schreibe 1 € auf dessen Konto gut.

  $\rightsquigarrow$ Jedes j mit $A[j] = 1$ hat ein Guthaben von 1 €.
  Zahle damit die Wertzuweisungen $A[i] \leftarrow 0$ in Zeile 3.

Potenzial-Methode

Sei D_i die Datenstruktur nach der i-ten Operation.

- Definiere eine Potenzialfunktion Φ, die jedem D_i ein $\Phi(D_i) \in \mathbb{R}$ zuordnet.
- Amortisierte Kosten $\hat{c}_i$ definiert durch $\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$.
- Gesamte amortisierte Kosten:

$$\sum_{i=1}^{n} \hat{c}_i \;=\; \sum_{i=1}^{n} \big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big) \;=\; \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- Ist $\Phi(D_n) \geq \Phi(D_0)$, so sind tatsächliche Gesamtkosten höchstens die amortisierten Gesamtkosten.
- Meist ist n variabel, und man setzt $\Phi(D_0) = 0$, und $\Phi(D_i) \geq 0$ für $i > 0$.

Potenzial-Methode

Im Beispiel:

- Definiere $\Phi(D_i) = b_i :=$ Anzahl der 1 im Zähler nach i INCREMENT-Operationen .
- Damit ist $\Phi(D_0) = 0$, und $\Phi(D_i) > 0$ für alle $i > 0$.
- Sei t_i die Zahl der Durchläufe der **while**-Schleife beim i-ten INCREMENT.
- Damit ist $c_i \leq t_i + 1$: Es werden t_i bits auf 0 gesetzt, und höchstens eines auf 1.
- Ausserdem ist $b_i \leq b_{i-1} - t_i + 1$.
- Daher ist die Potenzialdifferenz:

$$\Phi(D_i) - \Phi(D_{i-1}) \;=\; b_i - b_{i-1} \;\leq\; (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i \;,$$

  also sind die amortisierten Kosten:

$$\hat{c}_i \;=\; c_i + \Phi(D_i) - \Phi(D_{i-1}) \;\leq\; t_i + 1 + (1 - t_i) \;=\; 2$$

Union-Find: Datenstruktur für disjunkte Mengen

Es soll eine Familie von disjunkten dynamischen Mengen

$$M_1, \ldots, M_k \quad \text{mit} \quad M_i \cap M_j = \emptyset \text{ für } i \neq j$$

verwaltet werden.

Dabei sollen folgende Operationen zur Verfügung stehen:

- MAKE-SET(x) fügt die Menge $\{x\}$ zur Familie hinzu. Es ist Aufgabe des Benutzers, sicherzustellen dass x vorher in keiner der M_i vorkommt.
- FIND-SET(x) liefert ein kanonisches Element der Menge M_i mit $x \in M_i$, also FIND-SET(x) = FIND-SET(y), falls x und y in der gleichen Menge M_i liegen.
- UNION(x, y) ersetzt die Mengen M_i und M_j mit $x \in M_i$, $y \in M_j$ durch Ihre Vereinigung $M_i \cup M_j$.

Typische Anwendungen: Äquivalenzklassen einer Äquivalenzrelation, Zusammenhangskomponenten eines Graphen , ...

Union-Find: Realisierung als Wälder

Jede Menge wird durch einen Baum dargestellt. Knoten x haben Zeiger $p[x]$ auf ihren Vater. Kanonische Elemente sind an der Wurzel und haben $p[x] = x$.

Optimierungen:

- Rangfunktion: Jedes Element hat Rang $\approx \log\Big(\big|\text{Teilbaum ab } x\big|\Big)$
- Pfadkompression: Hänge jeden besuchten Knoten direkt an die Wurzel.

MAKE-SET(x)

```
1  p[x] ← x
2  rank[x] ← 0
```

FIND-SET(x)

```
1  if x ≠ p[x]
2     then p[x] ← FIND-SET(p[x])
3  return p[x]
```

UNION(x, y)

```
1  LINK(FIND-SET(x), FIND-SET(y))
```

LINK(x, y)

```
1  if rank[x] > rank[y]
2     then p[y] ← x
3     else  p[x] ← y
4           if rank[x] = rank[y]
5              then rank[y] ← rank[y] + 1
```

Union-Find: Amortisierte Komplexität

Satz: Jede Folge von m Make-Set-, Union- und Find-Set-Operationen, von denen n Make-Set sind, hat für Wälder mit Rangfunktion eine Laufzeit von $O(m \log n)$.

Beweis in der Übung.

$$\log^{(0)} n := n$$

$$\log^{(i+1)} n := \begin{cases} \log \log^{(i)} n & \text{falls } \log^{(i)} n > 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

$$\log^* n := \min\{i \,;\, \log^{(i)} n \leq 1\}$$

Satz: Jede Folge von m Make-Set-, Union- und Find-Set-Operationen, von denen n Make-Set sind, hat für Wälder mit Rangfunktion und Pfadkompression eine Laufzeit von $O(m \log^* n)$.