Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 22 (16.7.2018)

Greedy Algorithmen II
(Datenkompression)

Fabian Kuhn

Algorithmen und Komplexität

UNI FREIBURG

Datenkompression

- Reduziert Größen von Files
- Viele Verfahren für unterschiedliche Anwendungen: MP3, MPEG, JPEG, …
- Wie funktioniert Datenkompression?

Zwei Typen von Kompression:

- Verlustbehaftete Kompression (Bilder, Musik, Filme,…)
- Verlustfreie Kompression (Programme, Texte, …)

UNI FREIBURG

Beispiel:

- Alphabet Σ={a,b,c,d,…,x,y,z, ,.,:,!,?,&} (32 Zeichen)
- 5 Bits pro Symbol: $2^5 = 32$ Möglichkeiten

a	b	…	z		.	:	!	?	&
00000	00001		11001	11010	11011	11100	11101	11110	11111

Fragen?

- Sind 4 Bits pro Symbol nicht genug ?
- Müssen wir im Durchschnitt 5 Bits für jedes Vorkommen eines Symbols in langen Texten verwenden?

Beobachtung:

- Nicht jeder Buchstabe kommt gleich häufig vor
- z. B. kommen x, y und z in der deutschen Sprache viel seltener vor als e, n oder r

Idee:

- Benutze kurze Bitstrings für Symbole die häufig vorkommen

Effekt:

- Gesamtlänge der Kodierung einer Symbolfolge (eines Textes) wird reduziert

Präfix-Kodierung:

Eine Präfix-Kodierung für ein Alphabet Σ ist eine Funktion γ, die jeden Buchstaben $x \in \Sigma$ auf eine endliche Sequenz von 0 und 1 abbildet, so dass für $x, y \in \Sigma$, $x \neq y$, die Sequenz $\gamma(x)$ *kein* Präfix der Sequenz $\gamma(y)$ ist.

Beispiel (Präfix-Kodierung):

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	0100	0110	0111	1001	1010	1011	1101	1110	1111

UNI FREIBURG

Definition (Frequenz)

relative Häufigkeit

- Die Frequenz $f[x]$ eines Buchstaben $x \in \Sigma$ bezeichnet den Bruchteil der Buchstaben im Text, die x sind.

Beispiel:

- $\Sigma = \{0,1,2\}$
- Text =„0010022001“ (10 Zeichen)
- $f[0] = 3/5$
- $f[1] = 1/5$
- $f[2] = 1/5$

UNI FREIBURG

Definition (Kodierungslänge)

Die Kodierungslänge eines Textes mit n Zeichen bzgl. einer Kodierung γ ist definert als

$$\text{Kodierungslänge} = \sum_{x \in X} \overbrace{n \cdot f[x]}^{\#\text{Vorkommen von } x} \cdot \overbrace{|\gamma(x)|}^{\text{Länge der Kodierung von } x}$$

Beispiel:

- Σ = {a,b,c,d}
- γ(a) = 0; γ(b) =101; γ(c)= 110; γ(d)=111
- Text = „aacdaabb“
- Kodierungslänge = 16

Definition (durchschn. Kodierungslänge)

Die durchschnittliche Kodierungslänge eines Buchstabens in einem Text mit n Zeichen und bzgl. einer Kodierung γ ist definiert als

$$\mathbf{ABL}(\boldsymbol{\gamma}) = \sum_{x \in \Sigma} \boldsymbol{f}[\boldsymbol{x}] \cdot |\boldsymbol{\gamma}(\boldsymbol{x})|$$

Average **B**its per **L**etter

Beispiel:

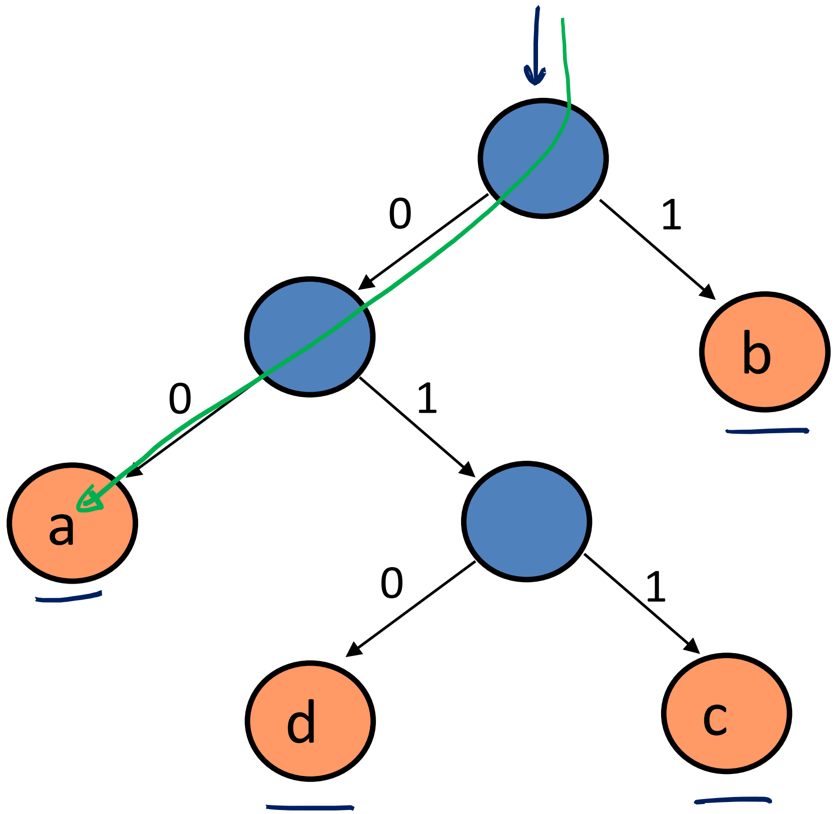
- Σ = {a,b,c,d}
- γ(a) = 0; γ(b) =101; γ(c)= 110; γ(d)=111
- Text = „aacdaabb“
- Durchschnittliche Kodierungslänge = 16/8 = 2

Problem einer optimalen Präfix-Kodierung:

- **Eingabe:**
  Alphabet Σ und für jedes $x \in \Sigma$ seine Frequenz $f[x]$

- **Ausgabe:**
  Eine Präfix-Kodierung γ, die $ABL(\gamma)$ minimiert

Binärbäume und Präfix-Kodierungen:

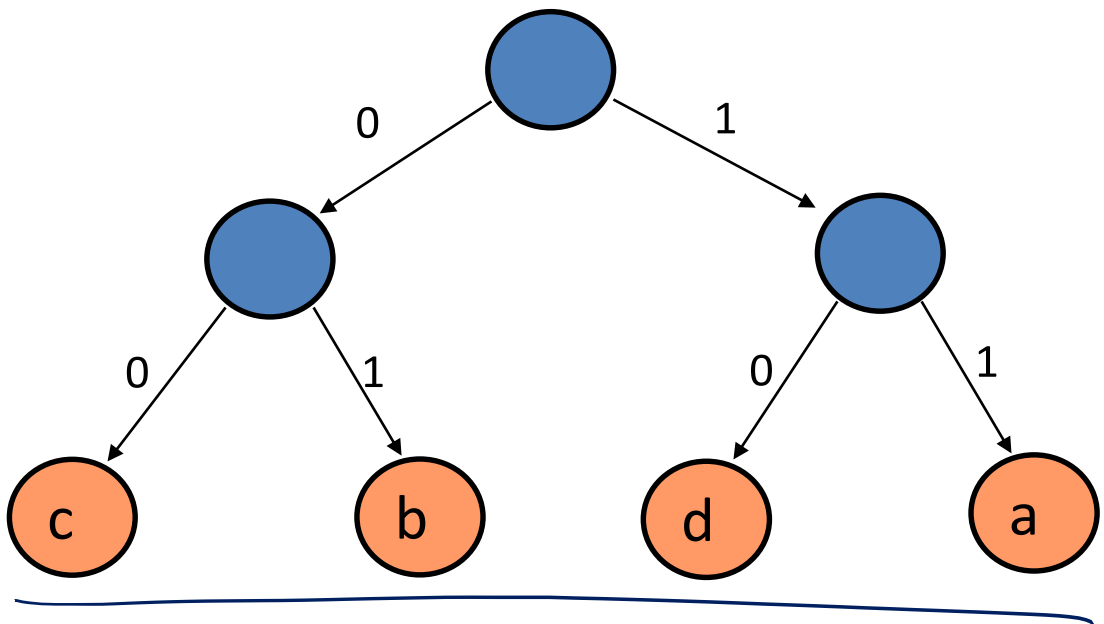


$x \in \Sigma$	$\gamma(x)$
a	00
b	1
c	011
d	010

10

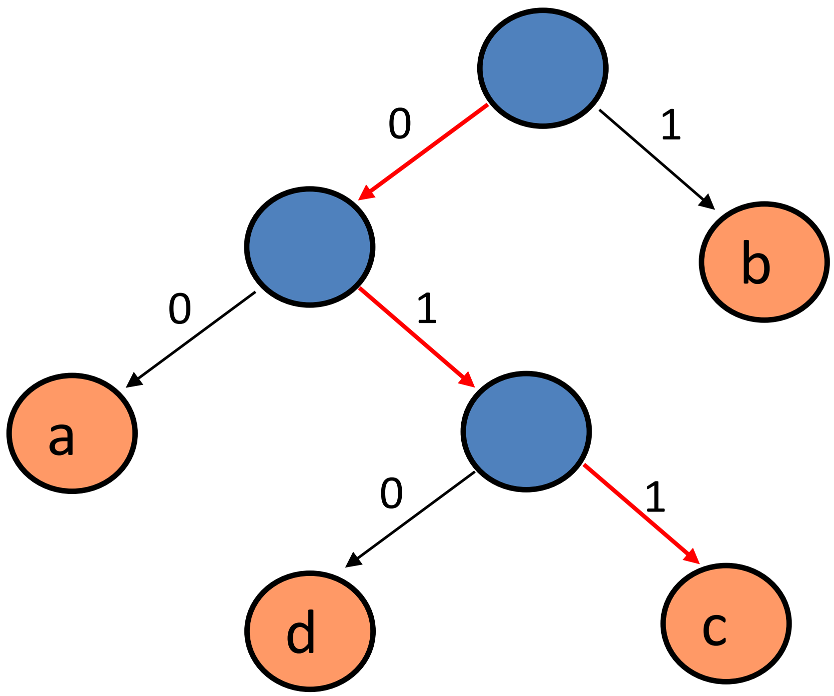
Präfix-Kodierungen und Binärbäume:

$x \in \Sigma$	$\gamma(x)$
a	11
b	01
c	00
d	10



Definition:

Die Tiefe eines Baumknotens ist die Länge seines Pfades zur Wurzel.



Tiefe = Code-Wort-Länge

Tiefe(c) = 3

UNI FREIBURG

Neue Problemformulierung:

- Suche Binärbaum T, dessen Blätter die Symbole aus Σ sind und der

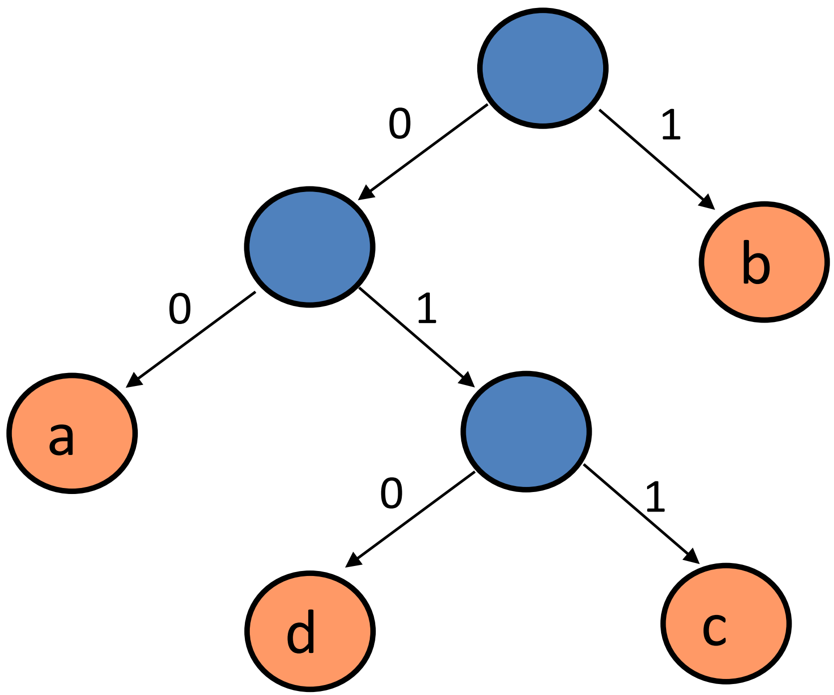
$$\mathrm{ABL}(T) = \sum_{x \in X} f[x] \cdot \underbrace{\mathrm{Tiefe}_T(x)}$$

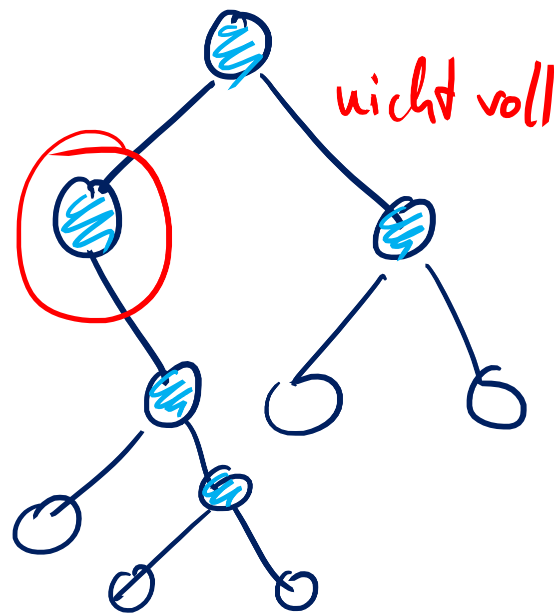
minimiert.

$\mathrm{Tiefe}_T(x) = |\gamma(x)|$

Definition:

Ein Binärbaum heißt voll, wenn jeder innere Knoten genau zwei Kinder hat.

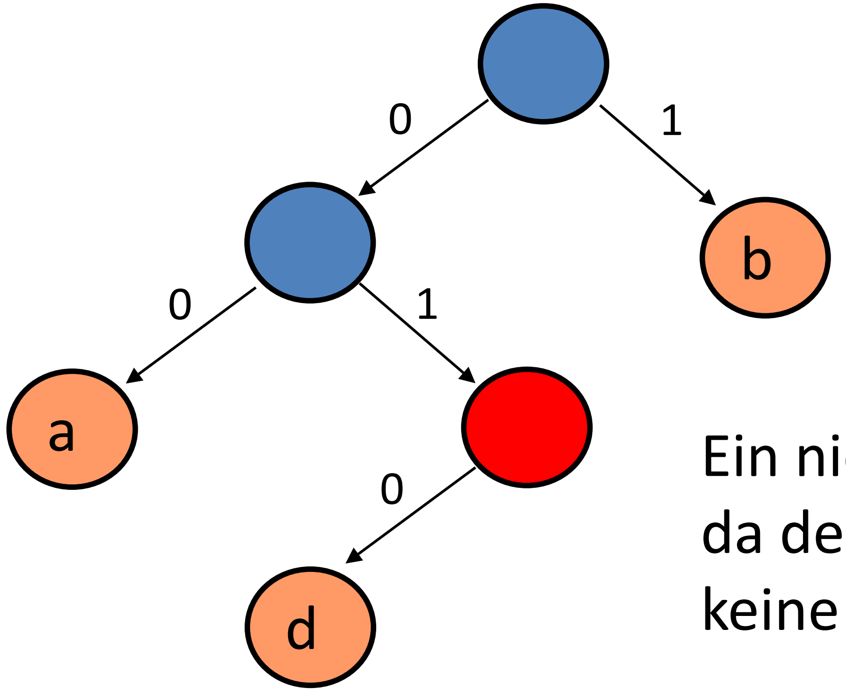




Ein voller Binärbaum

Definition:

Ein Binärbaum heißt voll, wenn jeder innere Knoten genau zwei Kinder hat.

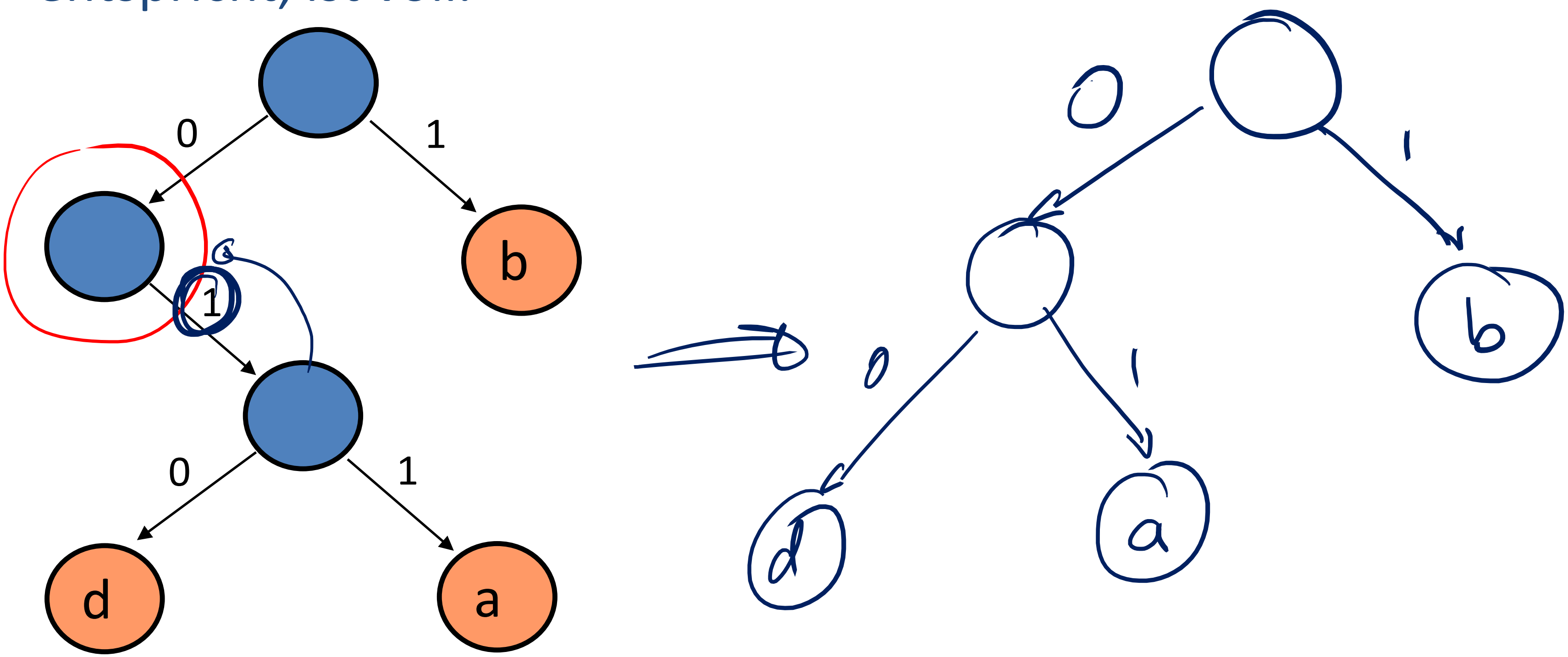


Ein nicht voller Binärbaum, da der rote innere Knoten keine zwei Kinder hat

Greedy Algorithmen – Datenkompression

Lemma:

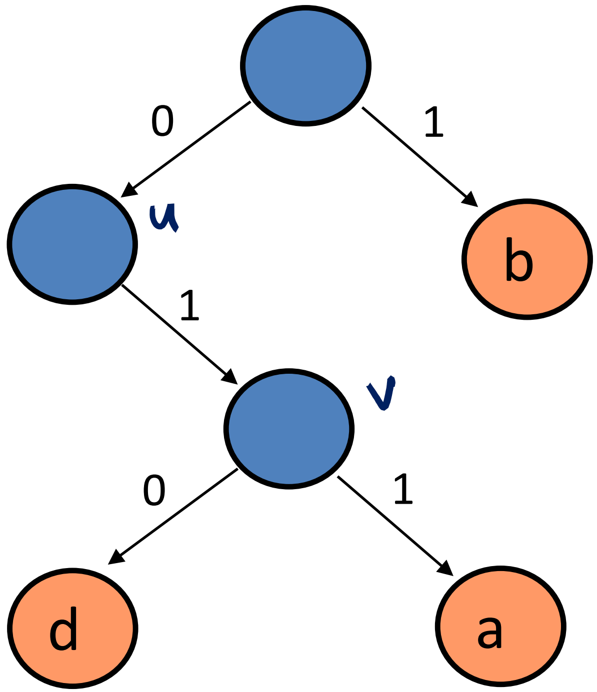
Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.



Greedy Algorithmen – Datenkompression

Lemma:

Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.

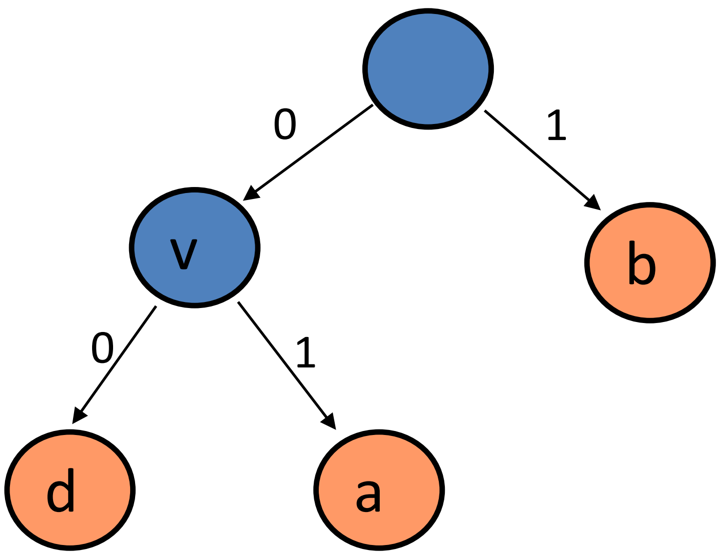


Beweis:

- Annahme: T ist optimal und hat inneren Knoten u mit einem Kind v
- Ersetze u durch v
- Dies verkürzt die Tiefe einiger Knoten, erhöht aber keine Tiefe
- Damit verbessert man die Kodierung

Lemma:

Der Binärbaum, der einer optimalen Präfix-Kodierung entspricht, ist voll.

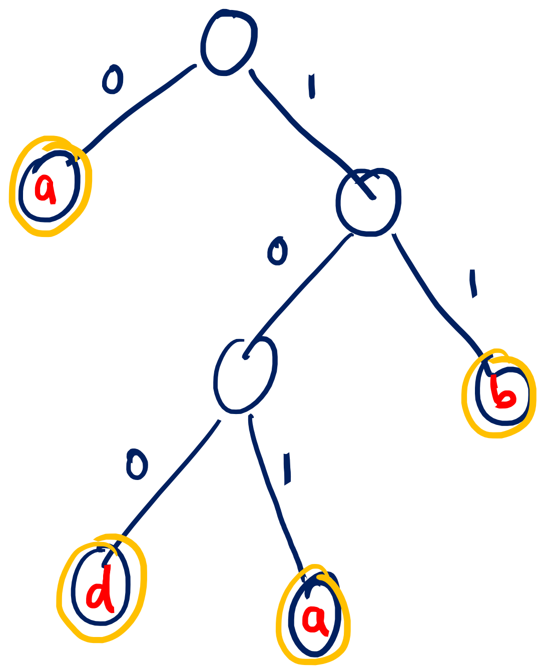


Beweis:

- Annahme: T ist optimal und hat inneren Knoten u mit einem Kind v
- Ersetze u durch v
- Dies verkürzt die Tiefe einiger Knoten, erhöht aber keine Tiefe
- Damit verbessert man die Kodierung

Ein Gedankenexperiment:

- Angenommen, jemand gibt uns den optimalen Baum T*, aber nicht die Bezeichnung der Blätter
- Wie schwierig ist es, die Bezeichnungen zu finden?

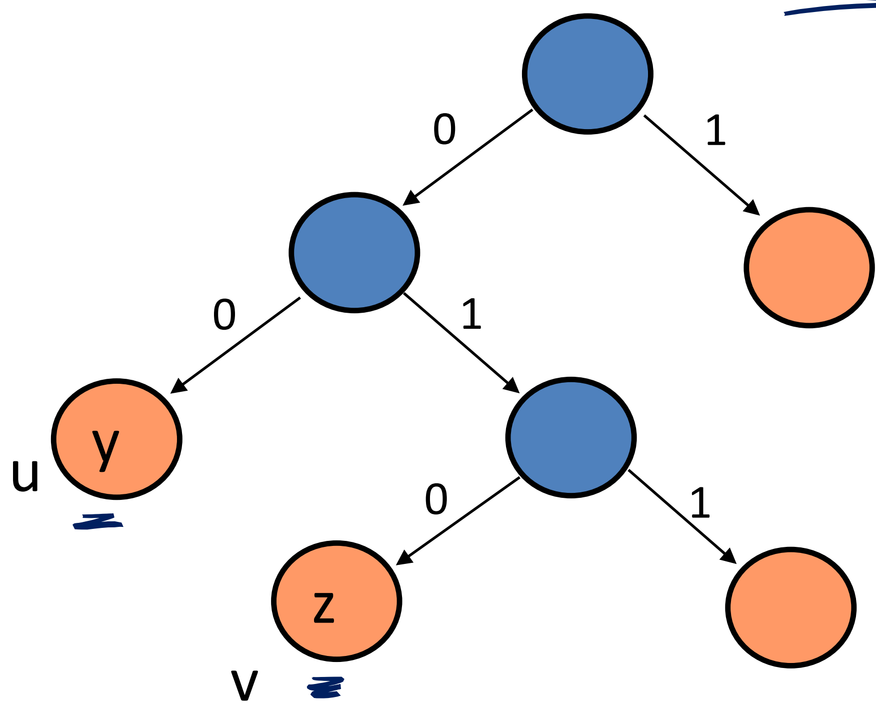


a : 0.15
b : 0.3
c : 0.5
d : 0.05

Lemma:

Seien u und v Blätter von T* mit Tiefe(u) < Tiefe(v).

Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



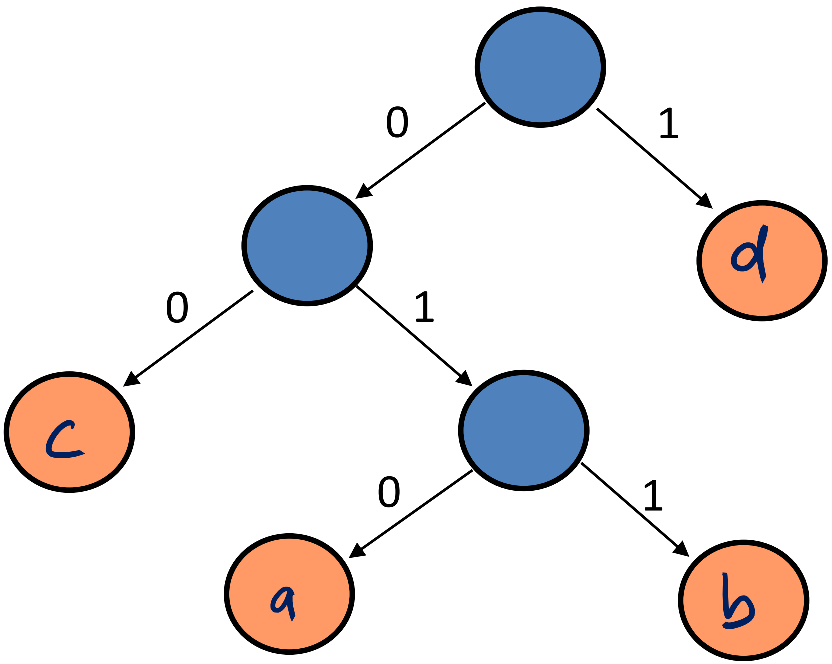
Vertauschen der Buchst. würde sonst die Kodierung verbessern

Greedy Algorithmen – Datenkompression

Lemma:

Seien u und v Blätter von T* mit Tiefe(u) < Tiefe(v).

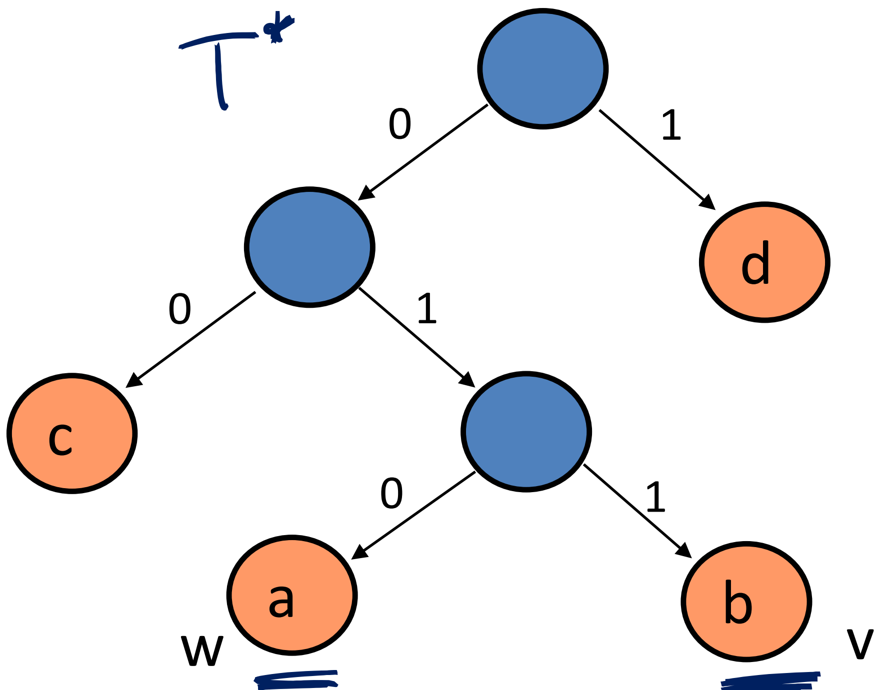
Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Beobachtung

Sei v der tiefste Blattknoten in T*. Dann hat v einen Geschwisterknoten und dieser ist ebenfalls ein Blattknoten.



(Geschwisterknoten von v)

w ist Blatt

↳ sonst wäre v nicht tiefster Blattknoten

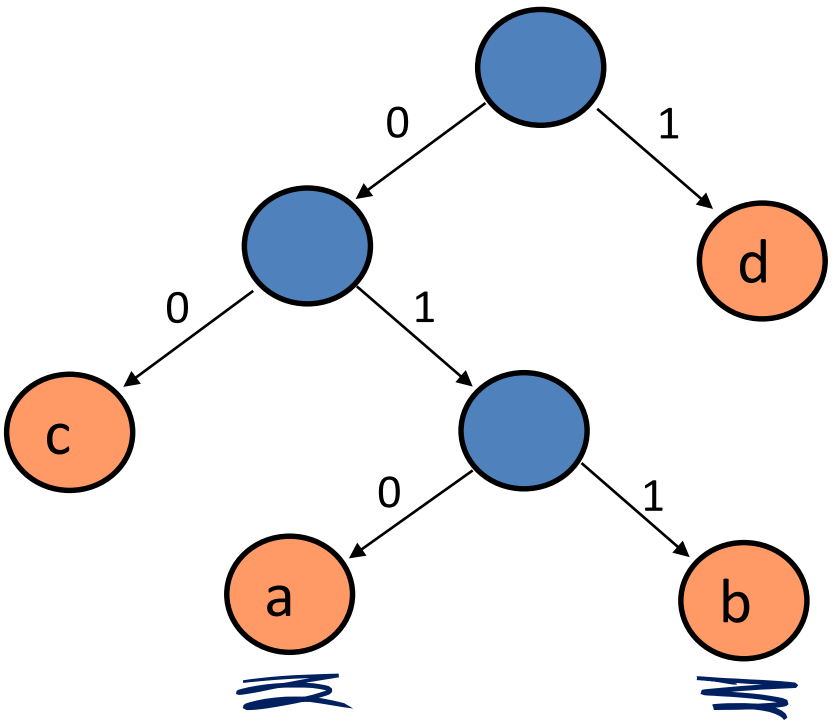
Wir wissen: T* ist voll

deshalb muss v einen Geschw.-knoten w haben

Zusammenfassende Behauptung

- Es gibt eine optimale Präfix-Kodierung mit zugehörigem Baum T*, so dass die beiden Blattknoten, denen die Symbole mit den kleinsten Frequenzen zugewiesen wurden, Geschwisterknoten in T* sind.

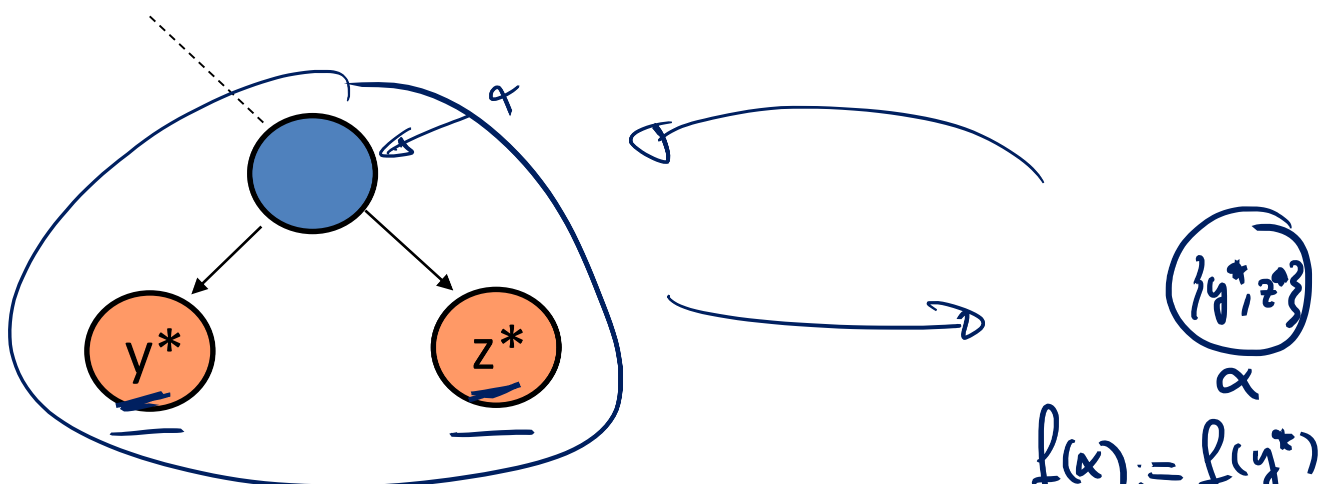
müssen nicht eindeutig sein



$x \in \Sigma$	f[x]
a	10%
b	12%
c	18%
d	60%

Idee des Algorithmus:

- Die beiden Symbole y* und z* mit den niedrigsten Frequenzen sind Geschwisterknoten
- Fasse y* und z* zu einem neuen Symbol zusammen
- Löse das Problem für die übrigen n-1 Symbole (z.B. rekursiv)



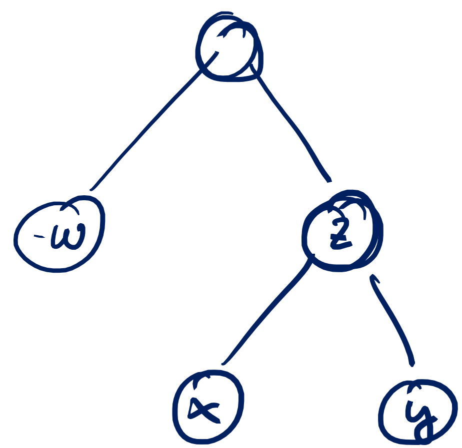
$$f(\alpha) := f(y^*) + f(z^*)$$

Greedy Algorithmen – Datenkompression

$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1\. $n \leftarrow |\ \Sigma\ |$
2\. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3\. **for** $i \leftarrow 1$ **to** n-1 **do**
4\. $\quad x \leftarrow$ deleteMin(Q)
5\. $\quad y \leftarrow$ deleteMin(Q)
6\. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7\. $\quad f[z] \leftarrow f[x] + f[y]$
8\. $\quad Q \leftarrow Q \cup \{z\}$
9\. **return** deleteMin(Q)



$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** $n-1$ **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

Q: 10% 12% 23% 55%

Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** $n-1$ **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=1

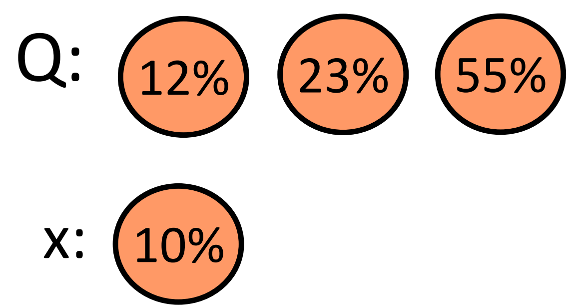
Q: 10% 12% 23% 55%

Greedy Algorithmen – Datenkompression

$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\ \Sigma\ |$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** $n-1$ **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=1



Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\ \Sigma\ |$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=1

Q: 23% 55%

x: 10%

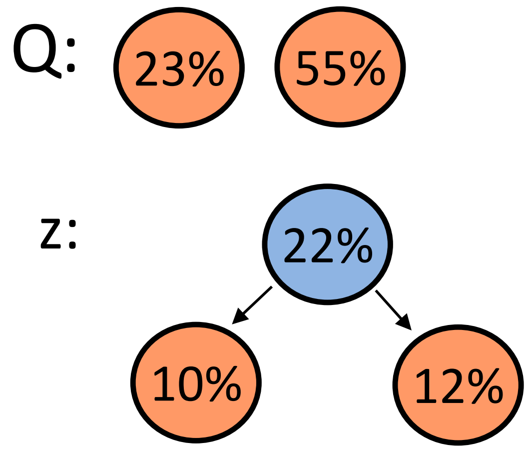
y: 12%

Greedy Algorithmen – Datenkompression

$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

```
Huffman(Σ)
n ← | Σ |
Q ← Σ      /* Priority Queue bzgl. f[x] */
for i ← 1 to n-1 do
    x ← deleteMin(Q)
    y ← deleteMin(Q)
    z←new BinTree(x, f[x]+f[y],y)
    f[z]← f[x] + f[y]
    Q ← Q ∪{z}
return deleteMin(Q)
```

i=1



Greedy Algorithmen – Datenkompression

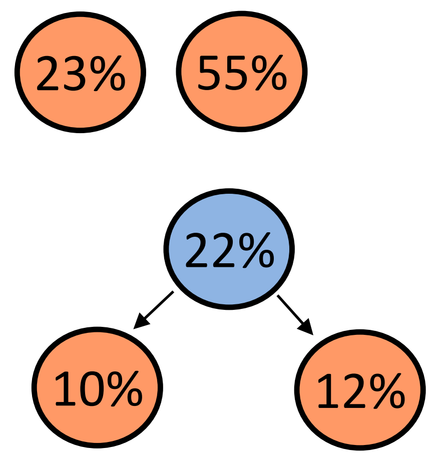
$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

```
Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ      /* Priority Queue bzgl. f[x] */
3. for i ← 1 to n-1 do
4.     x ← deleteMin(Q)
5.     y ← deleteMin(Q)
6.     z←new BinTree(x, f[x]+f[y],y)
7.     f[z]← f[x] + f[y]
8.     Q ← Q ∪{z}
9. return deleteMin(Q)
```

i=1

Q:

z:



Greedy Algorithmen – Datenkompression

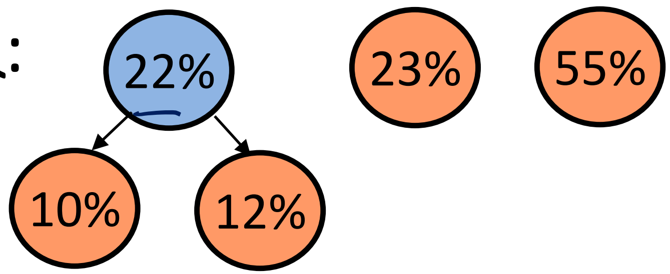
$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\,\Sigma\,|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=1

Q:



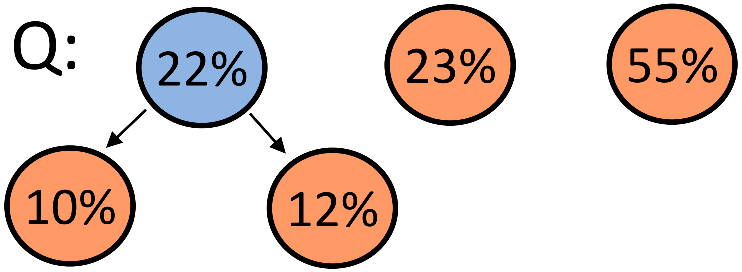
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=2

Q:



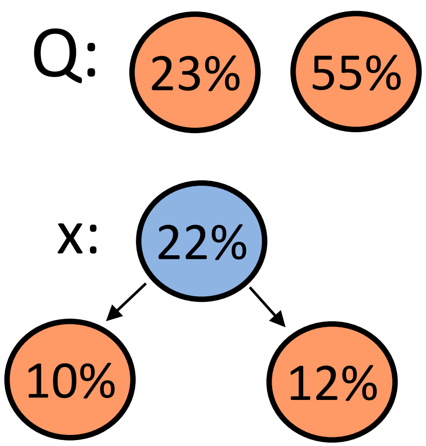
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=2



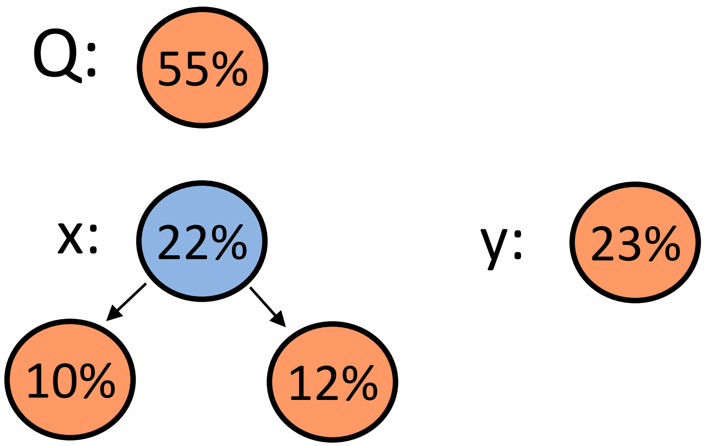
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\ \Sigma\ |$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=2

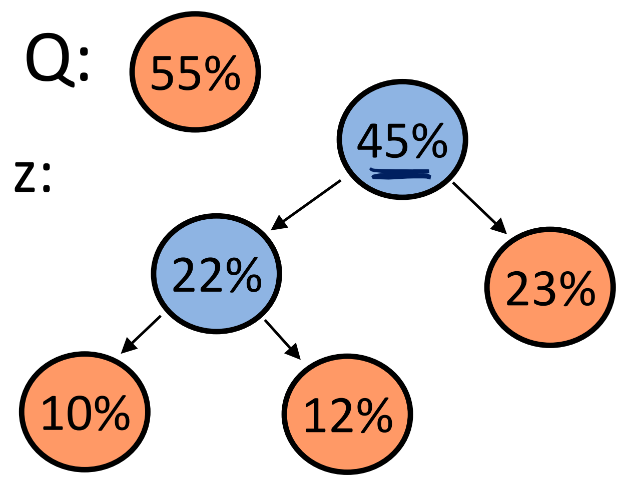


Greedy Algorithmen – Datenkompression

$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

```
Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ      /* Priority Queue bzgl. f[x] */
3. for i ← 1 to n-1 do
4.     x ← deleteMin(Q)
5.     y ← deleteMin(Q)
6.     z←new BinTree(x, f[x]+f[y],y)
7.     f[z]← f[x] + f[y]
8.     Q ← Q ∪{z}
9. return deleteMin(Q)
```

i=2

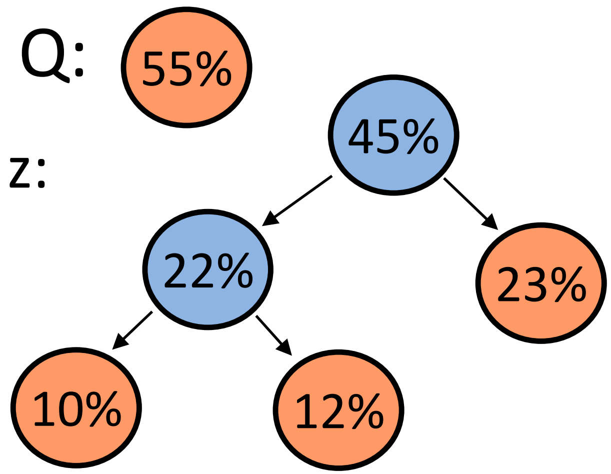


Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\,\Sigma\,|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=2



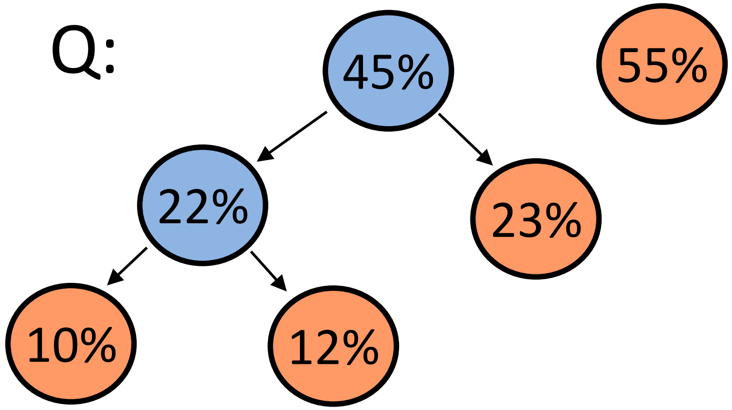
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\ \Sigma\ |$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=2

Q:



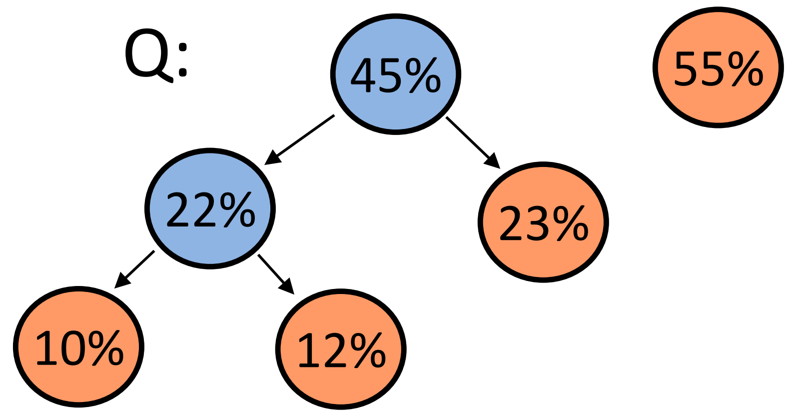
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

```
Huffman(Σ)
1. n ← | Σ |
2. Q ← Σ      /* Priority Queue bzgl. f[x] */
3. for i ← 1 to n-1 do
4.     x ← deleteMin(Q)
5.     y ← deleteMin(Q)
6.     z←new BinTree(x, f[x]+f[y],y)
7.     f[z]← f[x] + f[y]
8.     Q ← Q ∪{z}
9. return deleteMin(Q)
```

i=3

Q:



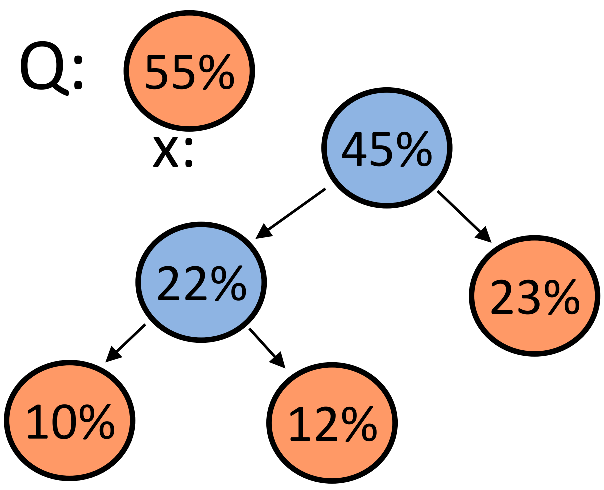
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)

1. $n \leftarrow |\ \Sigma\ |$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=3

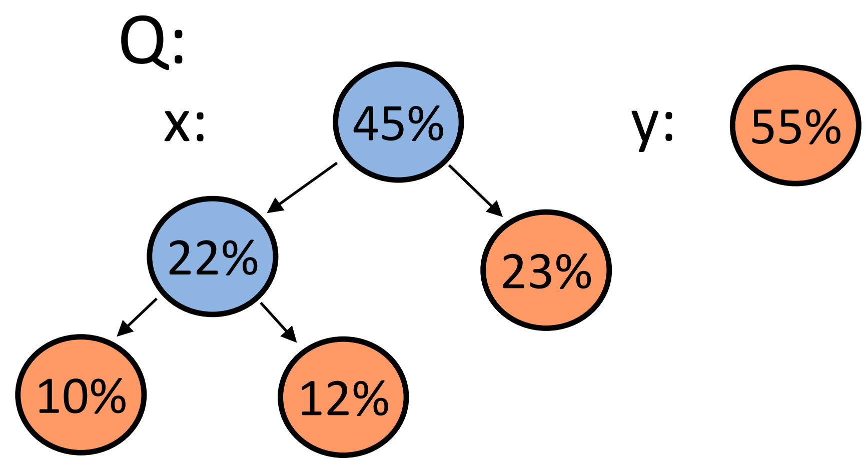


Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup${z}
9. **return** deleteMin(Q)

i=3



Greedy Algorithmen – Datenkompression

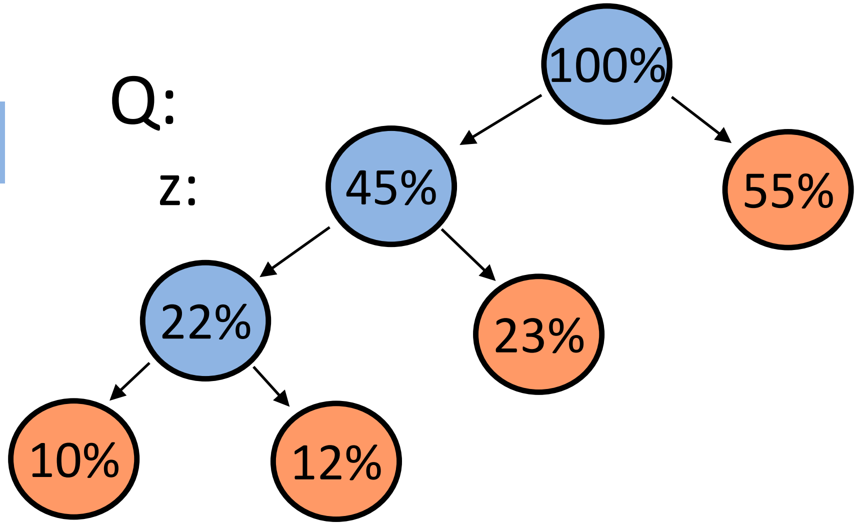
$x \in \Sigma$	$f[x]$
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. $n \leftarrow |\ \Sigma\ |$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $\quad x \leftarrow$ deleteMin(Q)
5. $\quad y \leftarrow$ deleteMin(Q)
6. $\quad z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $\quad f[z] \leftarrow f[x] + f[y]$
8. $\quad Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

i=3

Q:

z:



Greedy Algorithmen – Datenkompression

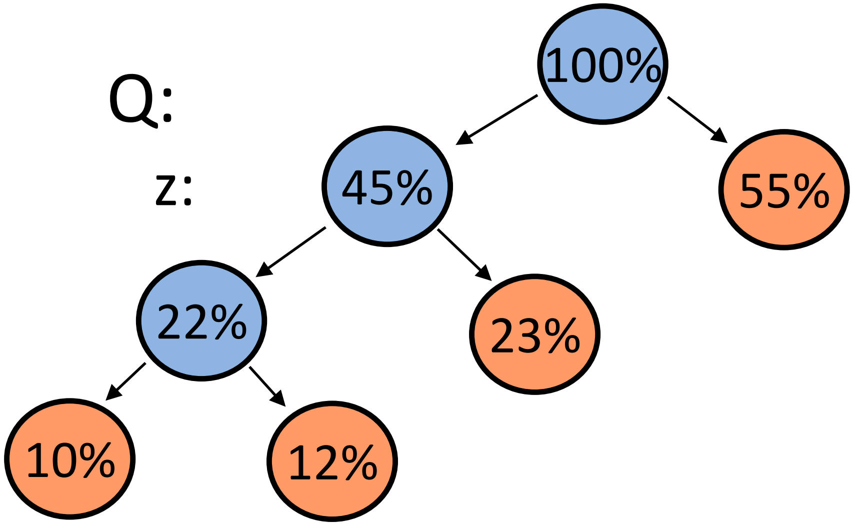
Huffman(Σ)
1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** n-1 **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

i=3

Q:

z:



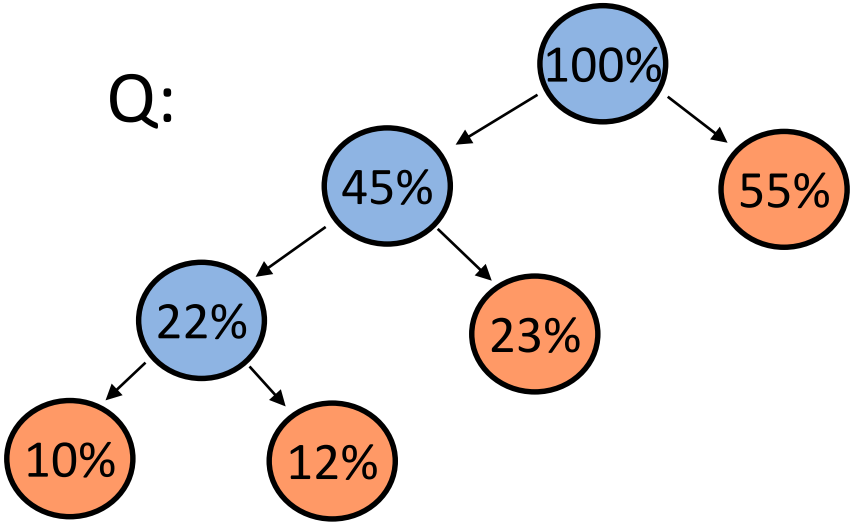
Greedy Algorithmen – Datenkompression

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

Huffman(Σ)
1. n $\leftarrow$ | Σ |
2. Q $\leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** i $\leftarrow$ 1 **to** n-1 **do**
4. x $\leftarrow$ deleteMin(Q)
5. y $\leftarrow$ deleteMin(Q)
6. z$\leftarrow$**new** BinTree(x, f[x]+f[y],y)
7. f[z]$\leftarrow$ f[x] + f[y]
8. Q $\leftarrow$ Q $\cup${z}
9. **return** deleteMin(Q)

i=3

Q:



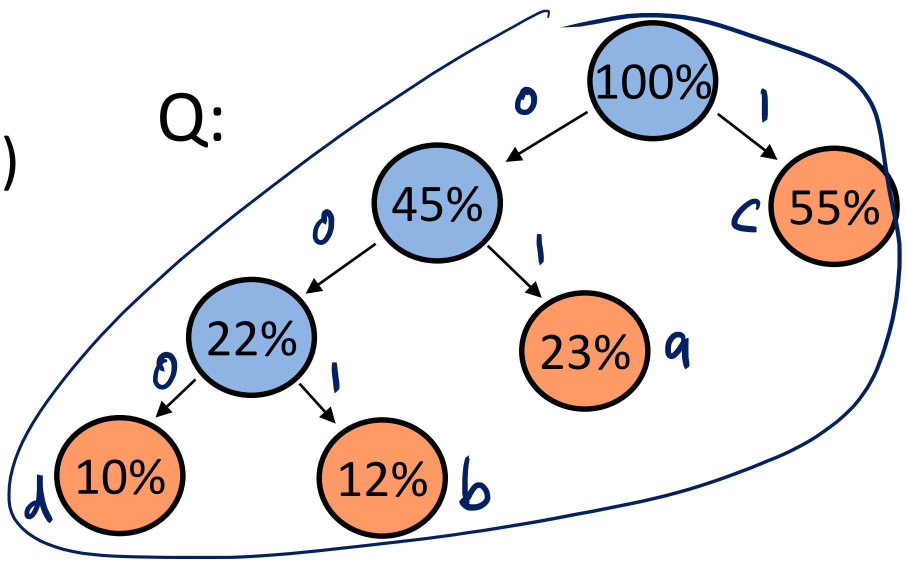
Greedy Algorithmen – Datenkompression

Huffman(Σ)
1. $n \leftarrow |\Sigma|$
2. $Q \leftarrow \Sigma$ /* Priority Queue bzgl. f[x] */
3. **for** $i \leftarrow 1$ **to** $n-1$ **do**
4. $x \leftarrow$ deleteMin(Q)
5. $y \leftarrow$ deleteMin(Q)
6. $z \leftarrow$ **new** BinTree(x, f[x]+f[y],y)
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** deleteMin(Q)

$x \in \Sigma$	f[x]
a	23%
b	12%
c	55%
d	10%

i=3

Q:



$f(z) \cdot \text{Tiefe}_{T'}(z)$

$(f(x)+f(y))(\text{Tiefe}_{T'}(z)+1)$

Theorem

Der Huffman(Σ)-Alg. berechnet eine optimale Präfix-Kodierung.

Beweis:

Durch Induktion über die Anzahl Symbole in Σ

Verankerung: $|\Sigma| \leq 2$ Alg. offensichtlich optimal

Schritt: $n-1 \longrightarrow n \quad (|\Sigma| = n)$

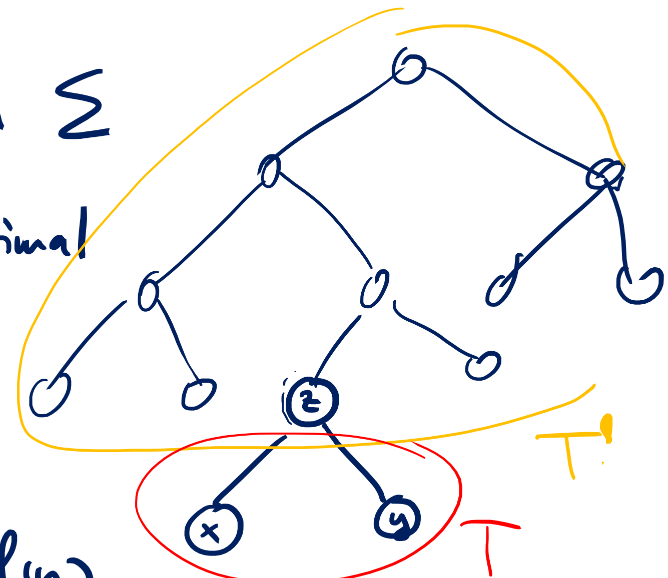
x, y: Symbole mit kleinsten Frequenzen in Σ

$\Sigma' = \Sigma \setminus \{x, y\} \cup \{z\}, \quad f(z) = f(x) + f(y)$

Alg. berechnet opt. Baum für $\Sigma' \Longrightarrow T'$

Induktionsvoraussetzung

Baum T des Alg. (für Σ): $ABL(T) = \sum_x f(x) \cdot \text{Tiefe}_T(x) = ABL(T') + f(x) + f(y)$



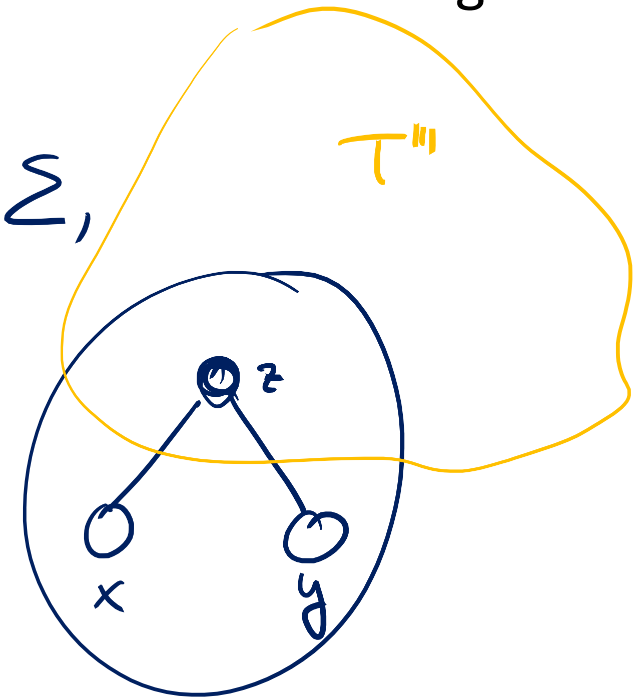
Greedy Algorithmen – Datenkompression

Theorem

Der Huffman(Σ)-Alg. berechnet eine optimale Präfix-Kodierung.

Annahme T ist nicht optimal

Es gibt einen besseren Baum T'' für Σ, so dass :



T''' ist ein Baum für Σ'

$\mathrm{ABL}(T'') = \mathrm{ABL}(T''') + f(x) + f(y)$

$\geq \mathrm{ABL}(T') + f(x) + f(y) = \mathrm{ABL}(T)$

Ind.-voraussetzung

Optimale durchschn. Codewortlänge

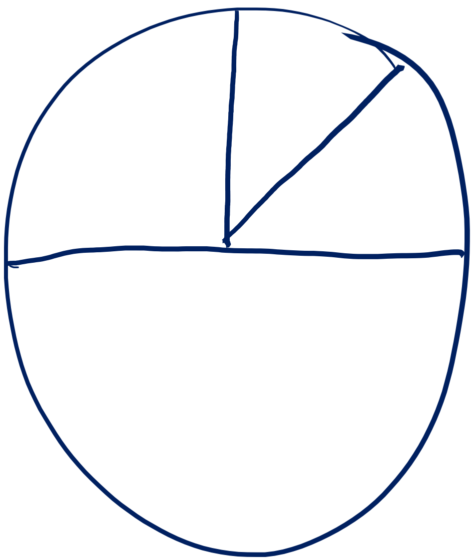
- Was ist die durchschnittliche Codewortlänge der Huffman-Codierung?

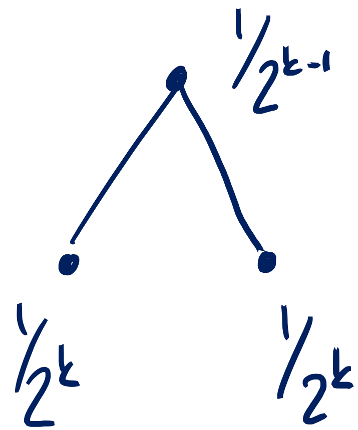
- Zur Einfachheit nehmen wir an, dass alle Frequenzen (rel. Häufigkeiten) von der Form $1/2^k$ sind

Optimale durchschn. Codewortlänge

Beobachtung: Falls alle Frequenzen von der Form $1/2^k$ sind und $1/2^{k_{\min}}$ die kleinste Frequenz ist, dann hat es mindestens zwei Zeichen mit Frequenz $1/2^{k_{\min}}$.

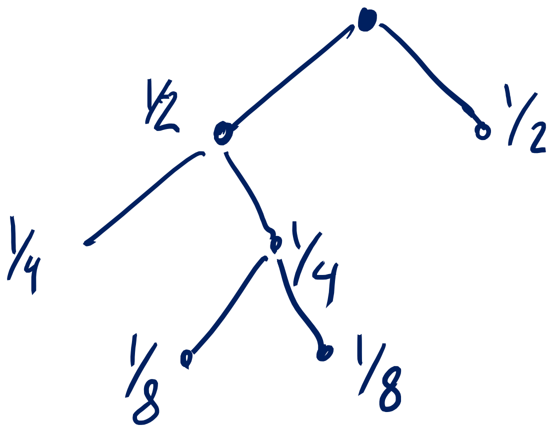
0.00001





Optimale durchschn. Codewortlänge

Lemma: Falls alle Frequenzen von der Form $1/2^k$ sind, dann hat ein Zeichen mit Frequenz $1/2^k$ ein Codewort der Länge genau k.



In Tiefe k sind Symbole mit Freq. genau 2^{-k}

Codewortlänge von x $= \log_2 \frac{1}{f(x)} = -\log_2 f(x)$

Optimale durchschn. Codewortlänge

Durchschnittliche Codewortlänge eines Huffman-Codes

- Annahme: Alle Frequenzen von der Form $1/2^k$

$$ABL(T) = \underbrace{-\sum_x f(x) \cdot \log_2 f(x)} = \sum_x f(x) \cdot \log_2 \frac{1}{f(x)}$$

Entropie

Häufigkeitsverteilung / Wahrscheinlichkeitsvert. $\boldsymbol{p(x)}$

- Elemente X, Element $x \in X$ hat Frequenz $p(x)$

aaaabaabab

Entropie $\boldsymbol{H(X)}$

Informationstheorie

$$H(X) := -\sum_{x \in X} p(x) \log_2 p(x)$$

Shannon

- Untere Schranke für die optimale durchschn. Codewortlänge
- Im Grenzwert (genug lange Zeichenketten) kann man mit durschn. Codewortlänge $H(X)$ codieren
- **Idee:**
  - Für genug grosses (konstantes) k, bestimme Codewort für jedes k-Tupel von Zeichen aus X
  - Verwende Huffman-Codierung

The End ☺