

Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 22 (20.7.2016)

Greedy Algorithmen - Datenkompression



**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

- Greedy Algorithmen sind eine Algorithmenmethode, um **Optimierungsprobleme** zu lösen.
 - wie auch z.B. dynamische Programmierung
- Greedy Algorithmen funktionieren oft nicht gut, aber wenn Sie funktionieren, dann sind sie besonders einfach und oft effizient
- Wir haben schon Beispiele gesehen und werden noch ein weiteres wichtiges Beispiel anschauen

- Vielen Dank an Christian Scheideler (U. Paderborn) für die Folien

“Gierige” Strategie für Optimierungsprobleme:

- Aufbau einer Lösung in „kleinen“ Schritten
- In jedem dieser Schritte wird entsprechend eines definierten Optimierungskriteriums eine irreversible Entscheidung getroffen

Frage 1

Wann kann eine solche Strategie zu einer optimalen Lösung führen?

Frage 2

Wie beweist man, dass ein Greedy Algorithmus eine optimale Lösung liefert?

- Wir werden das an einem Beispiel sehen:
zusätzliche Beispiele und verschiedene Beweisstrategien werden in der Algorithmentheorie besprochen

1. Greedy Algorithmen bestimmen Lösung durch sukzessives Erweitern von bereits gefundenen Teillösungen.
2. Erweitern geschieht durch lokal optimale Wahlen.
3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Greedy Algorithmen bestimmen Lösung durch sukzessives Erweitern von bereits gefundenen Teillösungen.
2. Erweitern geschieht durch lokal optimale Wahlen.
3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Algorithmus von Prim bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.
2. Prims Algorithmus wählt möglichst leichte Kante, die isolierten Knoten mit Teilbaum verbindet.
3. Analyse über Schnitte und die Definition von leichten Kanten über einen Schnitt

1. Greedy Algorithmen bestimmen Lösung durch sukzessives Erweitern von bereits gefundenen Teillösungen.
2. Erweitern geschieht durch lokal optimale Wahlen.
3. Analyse muss dann zeigen, dass lokal optimale Wahlen zu global optimalen Lösungen führt.

1. Algorithmus von Kruskal bestimmt minimalen Spannbaum durch sukzessives Hinzufügen von Kanten.
2. Kruskals Algorithmus wählt möglichst leichte Kante, die Zusammenhangskomp. verbindet.
3. Analyse über Schnitte und die Definition von leichten Kanten über einen Schnitt

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, ...)

Kodierung:

- Computer arbeiten auf Bits (Symbole 0 und 1), nutzen also das Alphabet $\{0,1\}$
- Menschen nutzen umfangreichere Alphabete (z.B. Alphabete von Sprachen)
- Darstellung auf Rechner erfordert Umwandlung in Bitfolgen

Beispiel:

- Alphabet $\Sigma = \{a, b, c, d, \dots, x, y, z, \dots, :, !, ?, \&\}$ (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

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ , so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma = \{0, 1, 2, \dots, 9\}$
- Text = 00125590004356789 (17 Zeichen)

Länge der Kodierung:
 $4 \cdot 17 = 68$ Bits

0	1	2	3	4	5	6	7	8	9
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Grundlegendes Problem:

- Eingabe: Text über dem Alphabet Σ
- Gesucht: Eine binäre Kodierung von Σ , so dass die Länge des Textes in dieser Kodierung minimiert wird

Beispiel:

- $\Sigma = \{0, 1, 2, \dots, 9\}$
- Text = 00125590004356789 (17 Zeichen)

0	1	2	3	4	5	6	7	8	9
0	11000	11001	11010	11011	10	11100	11101	11110	11111

MorseCode:

- Elektrische Pulse über Kabel
- Punkte (kurze Pulse)
- Striche(Lange Pulse)

Beispiele aus dem MorseCode:

- e ist 0 (ein einzelner Punkt)
- t ist 1 (ein einzelner Strich)
- a ist 01 (Punkt – Strich)

Problem:

- Ist 0101 eta, aa, etet, oder aet ?

Problem Mehrdeutigkeit:

- Ist die Kodierung eines Buchstabens ein Präfix der Kodierung eines anderen Buchstabens, dann kann es passieren, dass die Kodierung nicht eindeutig ist.

Beispiel:

- $e = 0$, $a = 01$
- **0** ist Präfix von **01**

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

Präfix-Code: Dekodierung

- Wie dekodiert man eine gegebenen mit einem Präfix-Code kodierte Zeichenkette?

Beispiel

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	1000	1001	1010	010	011	1011	1101	1110	1111

Kodierte Zeichenkette

0100010100110011110110001011011000100001000100101100

Präfix-Code als Binärbaum

- Wie dekodiert man eine gegebenen mit einem Präfix-Code kodierte Zeichenkette?

Beispiel

$x \in \Sigma$	0	1	2	3	4	5	6	7	8	9
$\gamma(x)$	00	1000	1001	1010	010	011	1011	1101	1110	1111

Definition (Frequenz)

- 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)
- $f[0] = 3/5$
- $f[1] = 1/5$
- $f[2] = 1/5$

Definition (Kodierungslänge)

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

$$\text{Kodierungslänge} = \sum_{x \in X} n \cdot f[x] \cdot |\gamma(x)|$$

Beispiel:

- $\Sigma = \{a,b,c,d\}$
- $\gamma(a) = 0$; $\gamma(b) = 101$; $\gamma(c) = 110$; $\gamma(d) = 111$
- 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

$$\text{ABL}(\gamma) = \sum_{x \in X} f[x] \cdot |\gamma(x)|$$

Beispiel:

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

Problem einer optimalen Präfix-Kodierung:

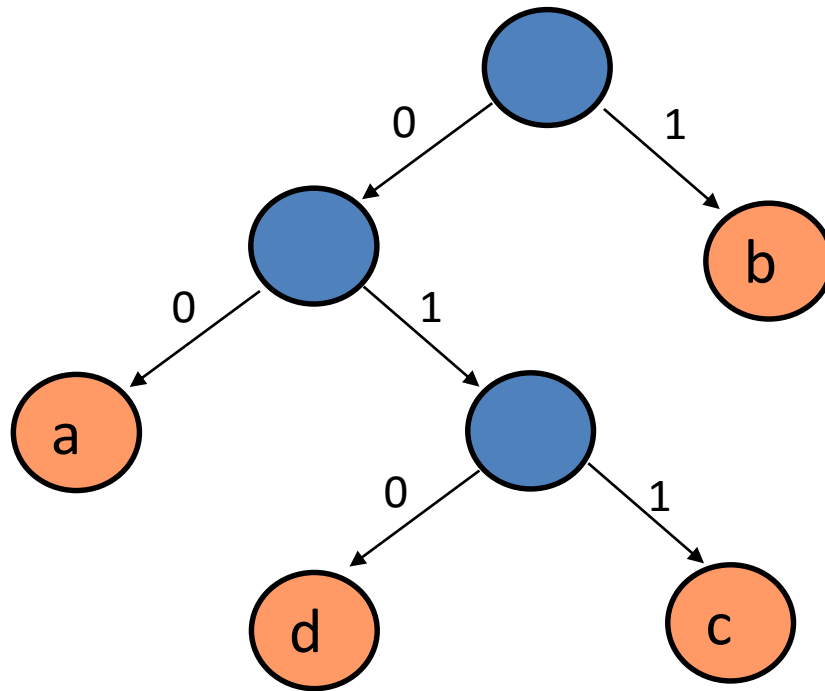
- **Eingabe:**

Alphabet Σ und für jedes $x \in \Sigma$ seine Frequenz $f[x]$

- **Ausgabe:**

Eine Kodierung γ , die $ABL(\gamma)$ minimiert

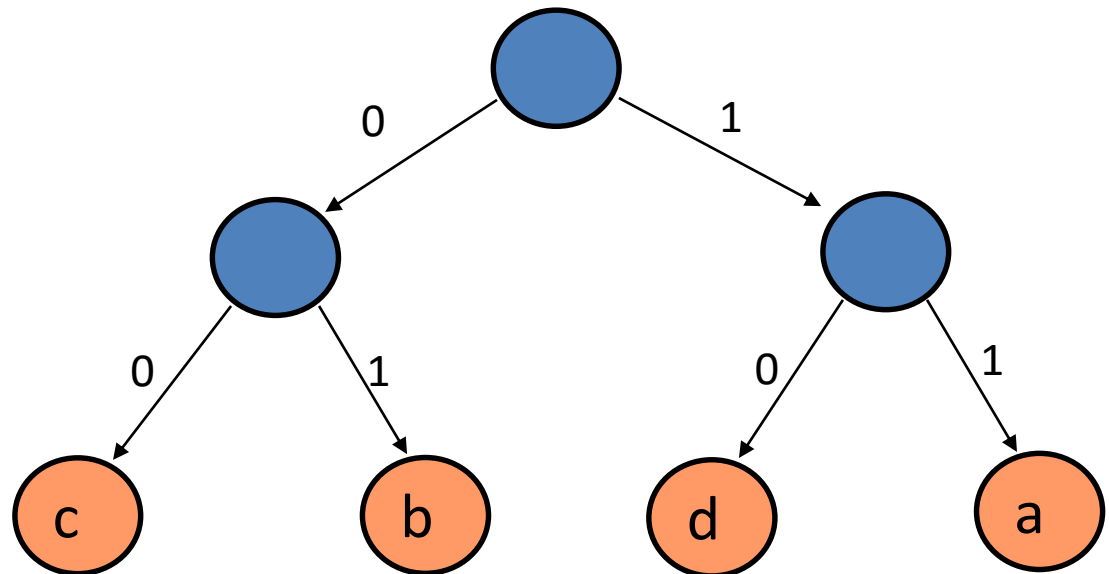
Binärbäume und Präfix-Kodierungen:



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

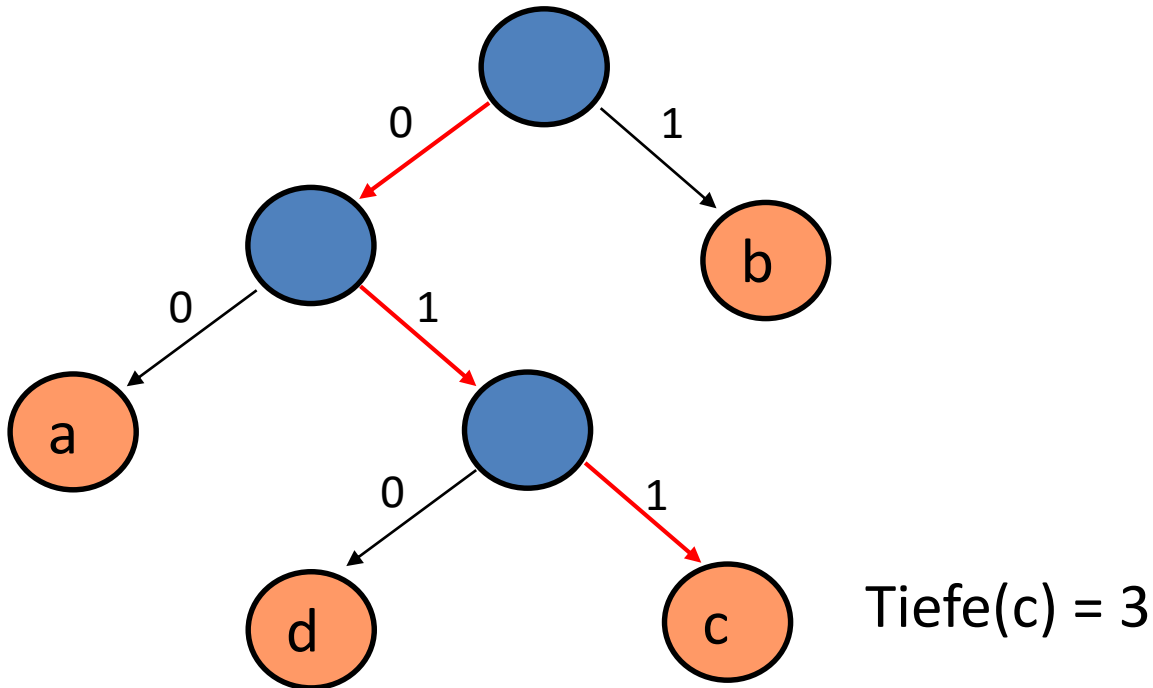
Präfix-Kodierungen und Binärbäume:

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



Definition:

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



Neue Problemformulierung:

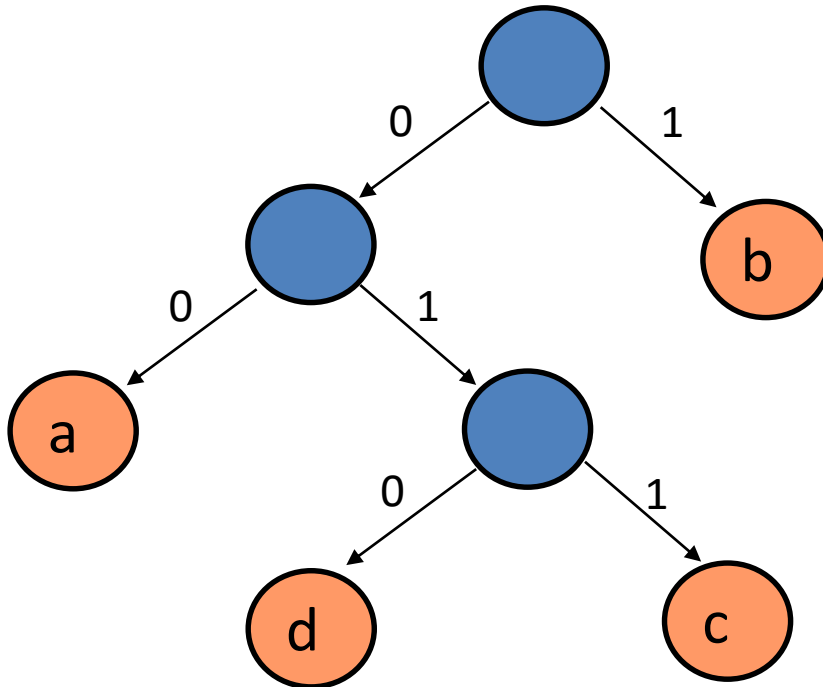
- Suche Binärbaum T , dessen Blätter die Symbole aus Σ sind und der

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

minimiert.

Definition:

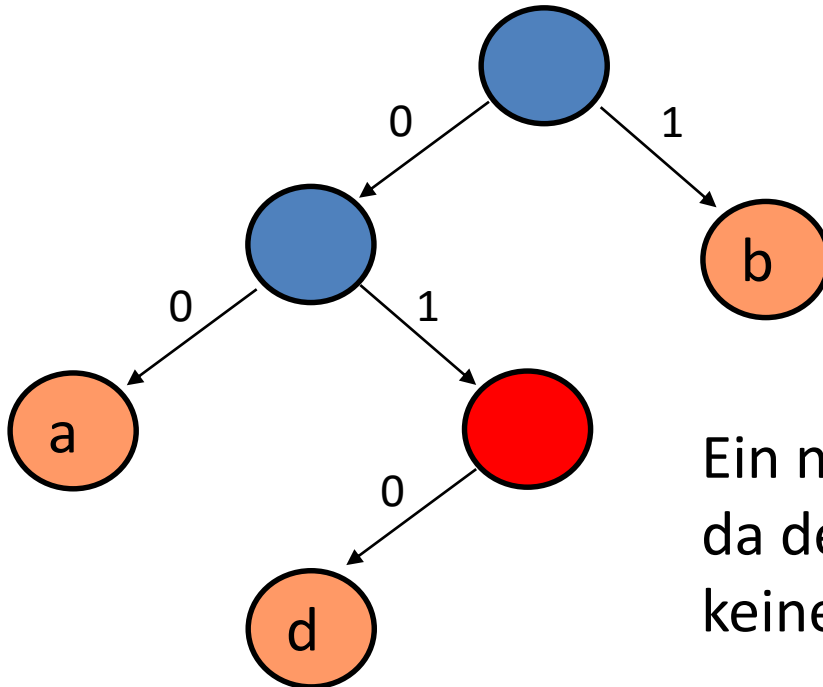
Ein Binärbaum heißt **voll**, wenn jeder innere Knoten genau zwei Kinder hat.



Ein voller Binärbaum

Definition:

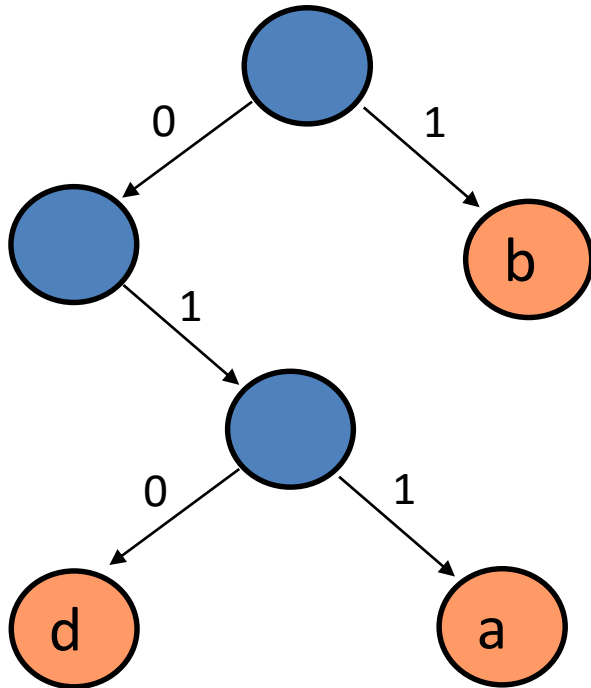
Ein Binärbaum heißt **voll**, wenn jeder innere Knoten genau zwei Kinder hat.



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

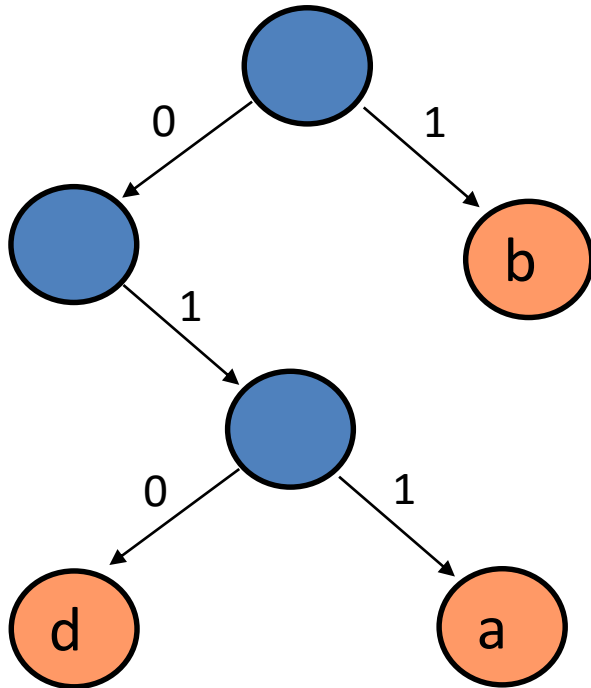
Lemma:

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



Lemma:

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

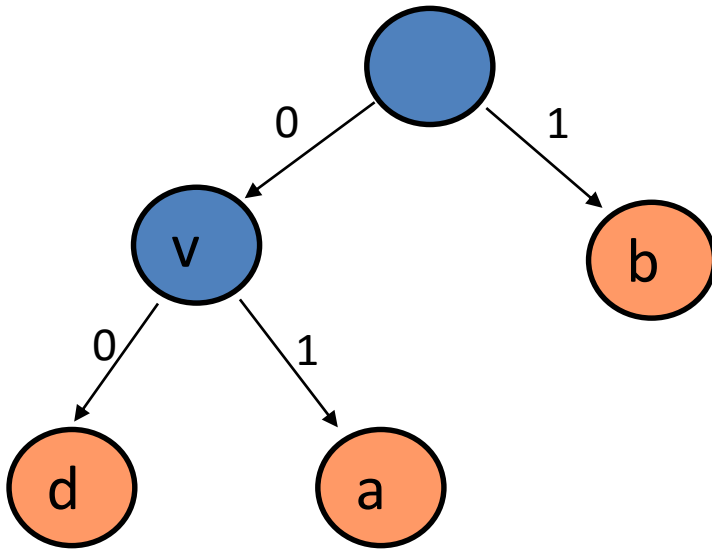


Beweis:

- 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

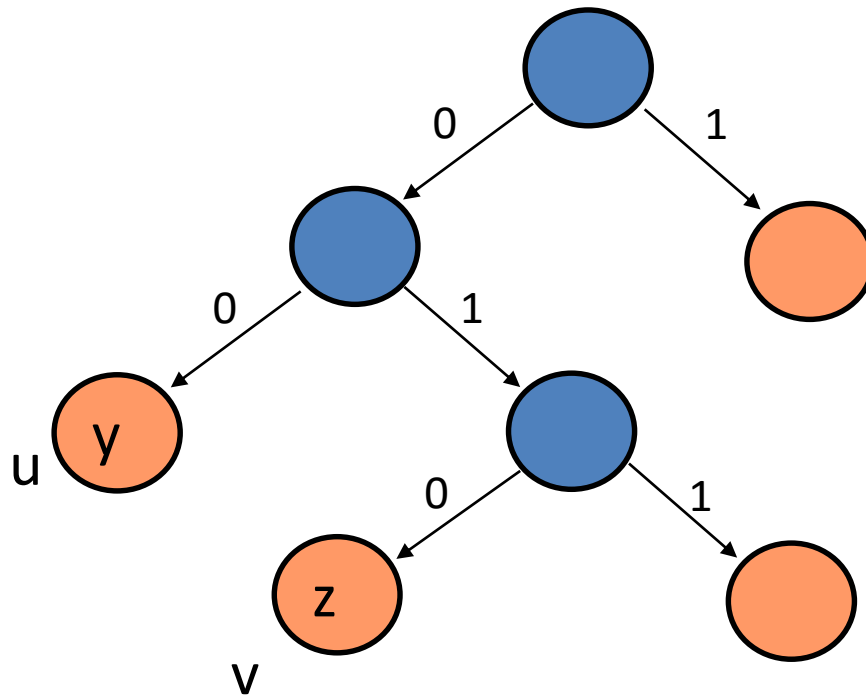
Ein Gedankenexperiment:

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

Lemma:

Seien u und v Blätter von T^* mit $\text{Tiefe}(u) < \text{Tiefe}(v)$.

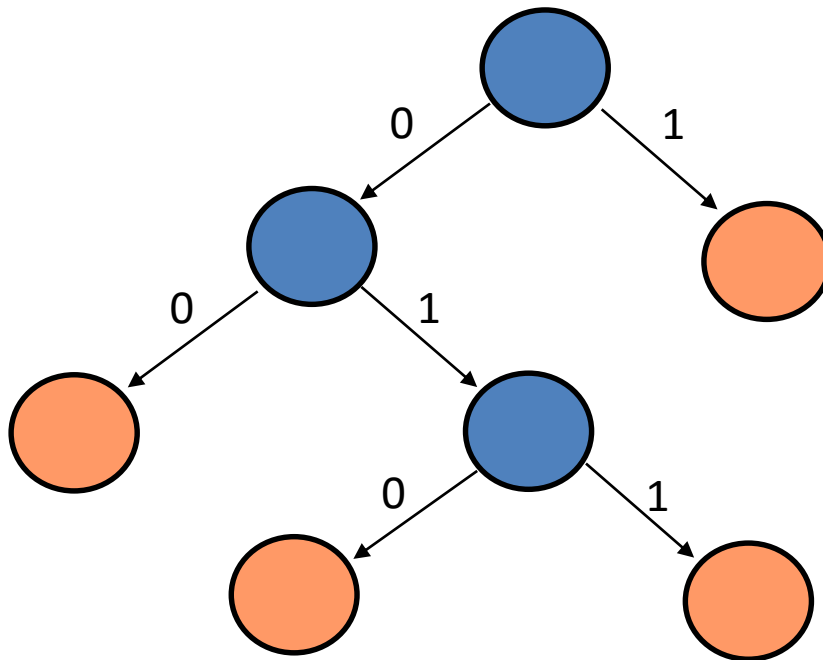
Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



Lemma:

Seien u und v Blätter von T^* mit $\text{Tiefe}(u) < \text{Tiefe}(v)$.

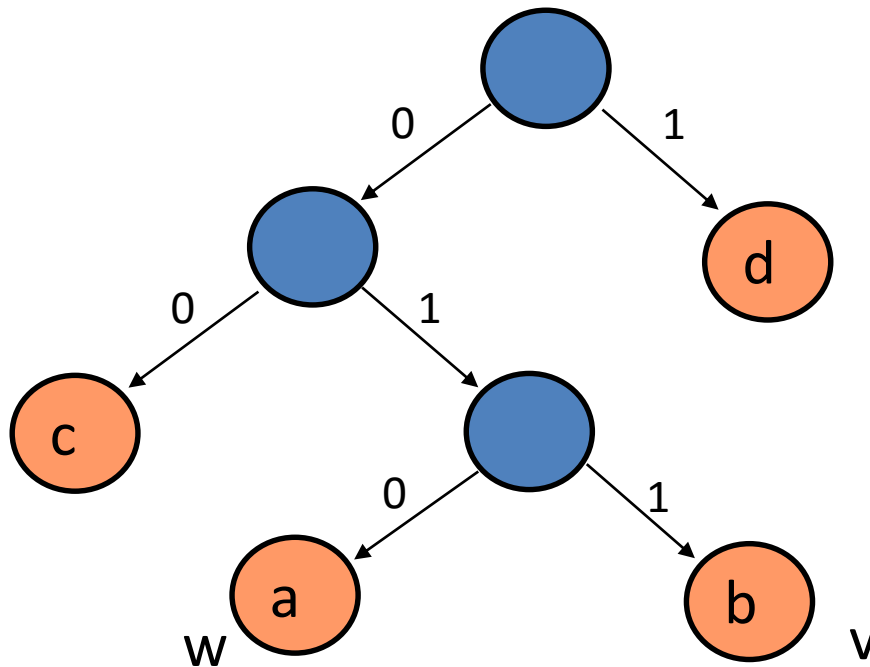
Seien u bzw. v in einer optimalen Kodierung mit $y \in \Sigma$ bzw. $z \in \Sigma$ bezeichnet. Dann gilt $f[y] \geq f[z]$.



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

Beobachtung¹

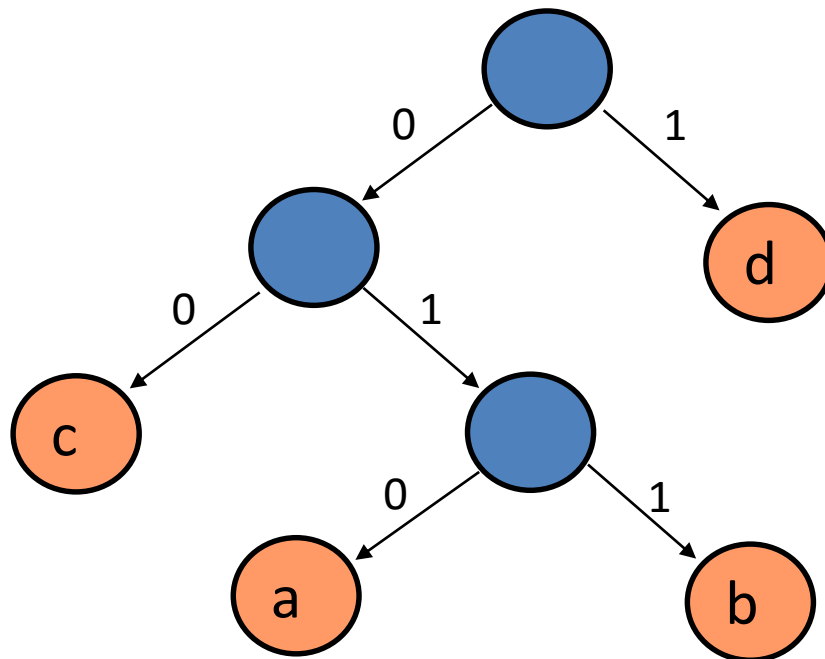
Sei v der tiefste Blattknoten in T^* . Dann hat v einen Geschwisterknoten und dieser ist ebenfalls ein Blattknoten.



(Geschwisterknoten von v)

Zusammenfassende Behauptung

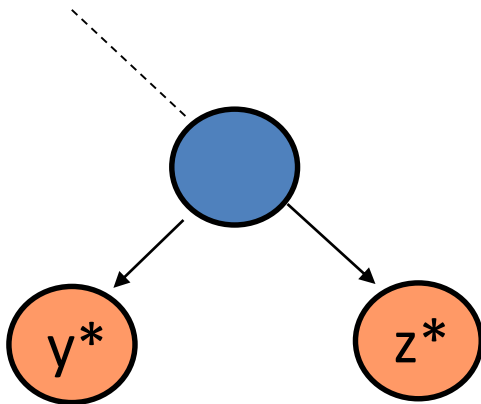
- Es gibt eine optimale Präfix-Kodierung mit zugehörigem Baum T^* , so dass die beiden Blattknoten, denen die Symbole mit den kleinsten Frequenzen zugewiesen wurden, Geschwisterknoten in T^* sind.



$x \in \Sigma$	$f[x]$
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)



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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

$x \in \Sigma$	$f[x]$
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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

$x \in \Sigma$	$f[x]$
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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

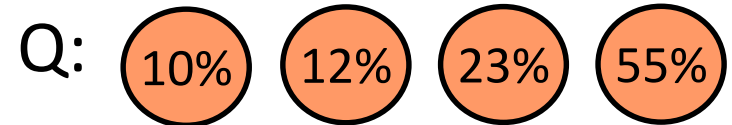


Huffman(Σ)

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

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

$i=1$

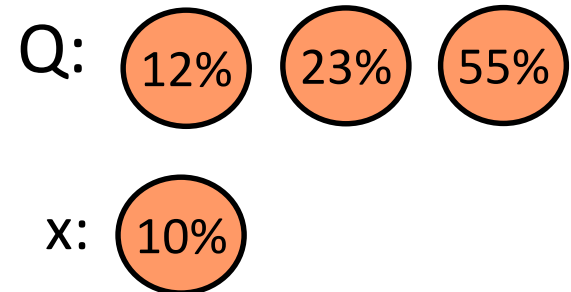


Huffman(Σ)

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

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

$i=1$



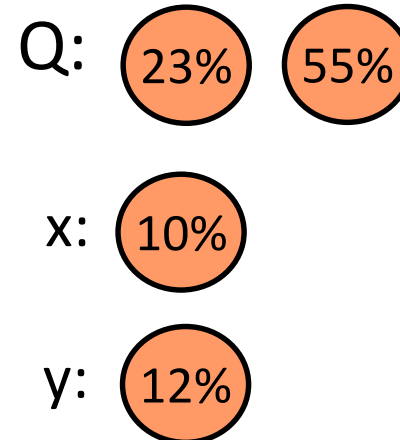
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=1$



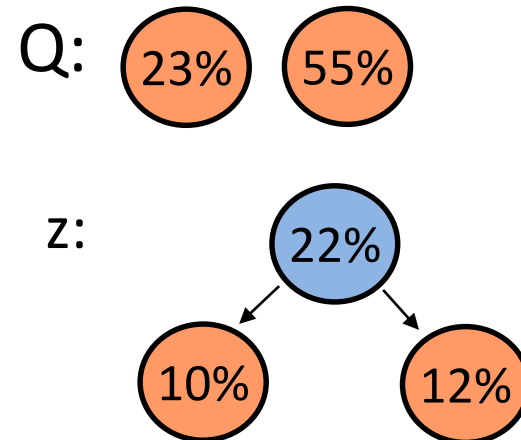
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=1$



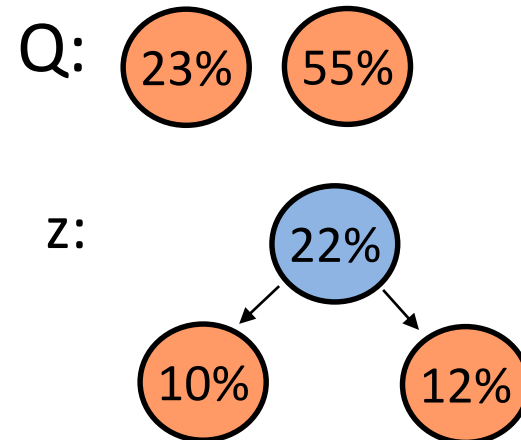
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=1$

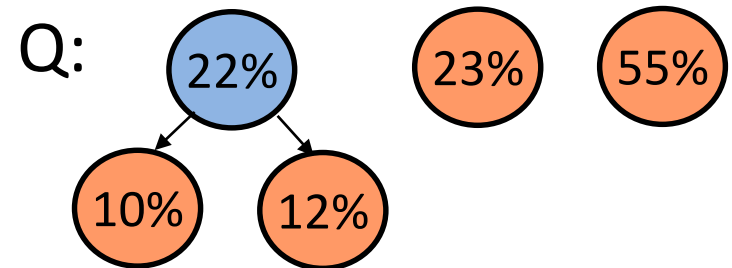


Huffman(Σ)

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

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

$i=1$



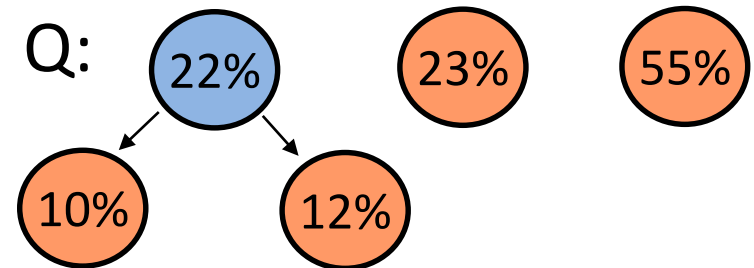
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=2$



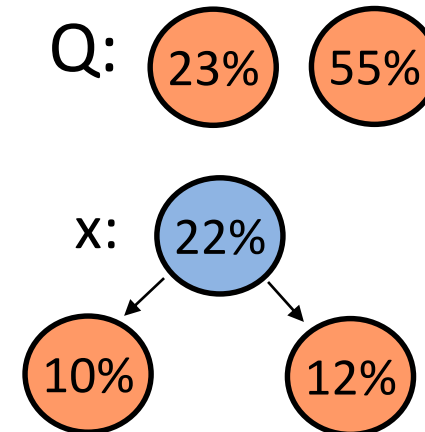
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=2$



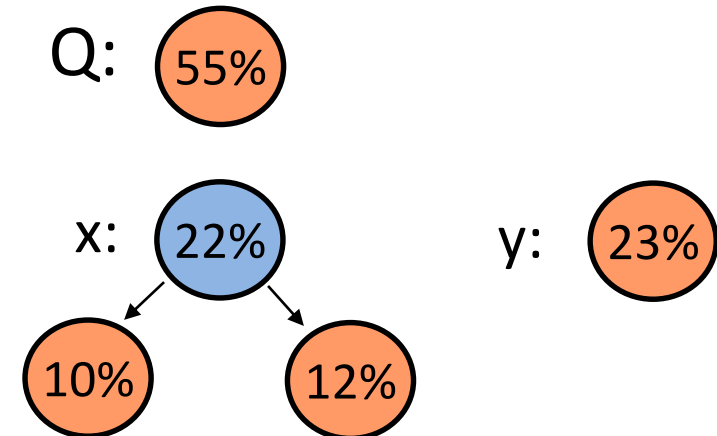
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=2$



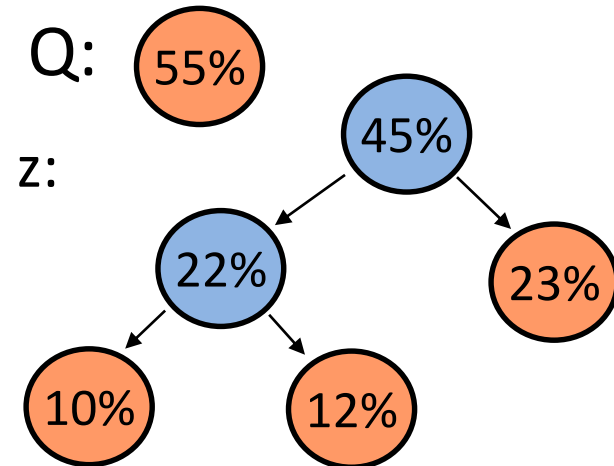
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=2$



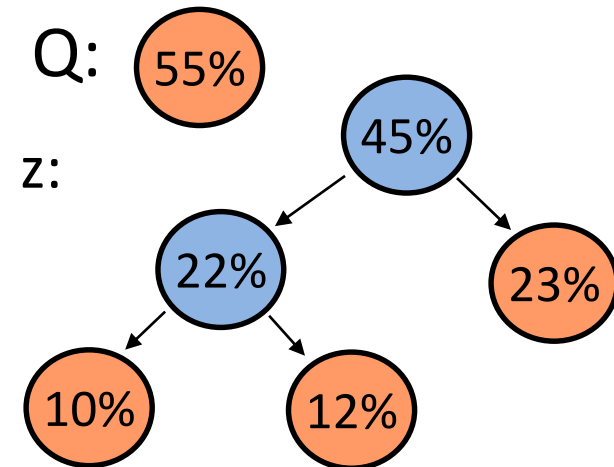
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=2$



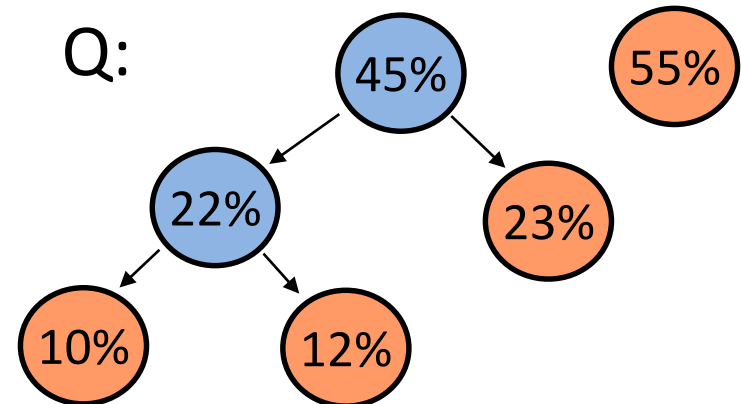
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=2$



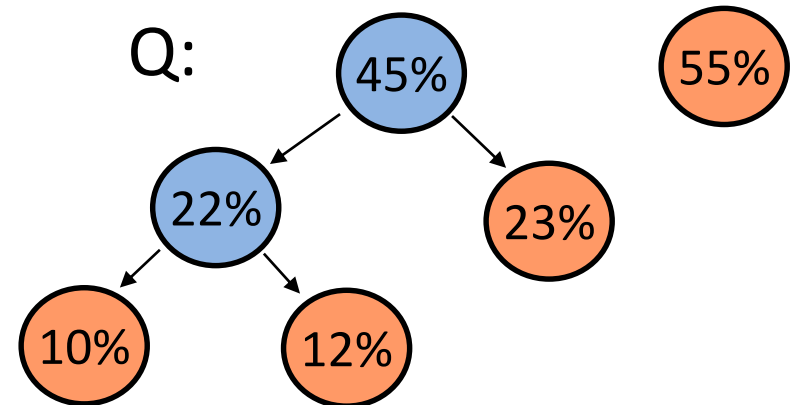
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=3$



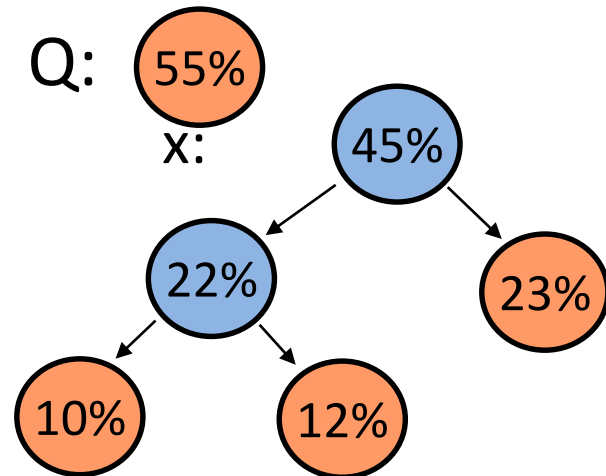
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=3$



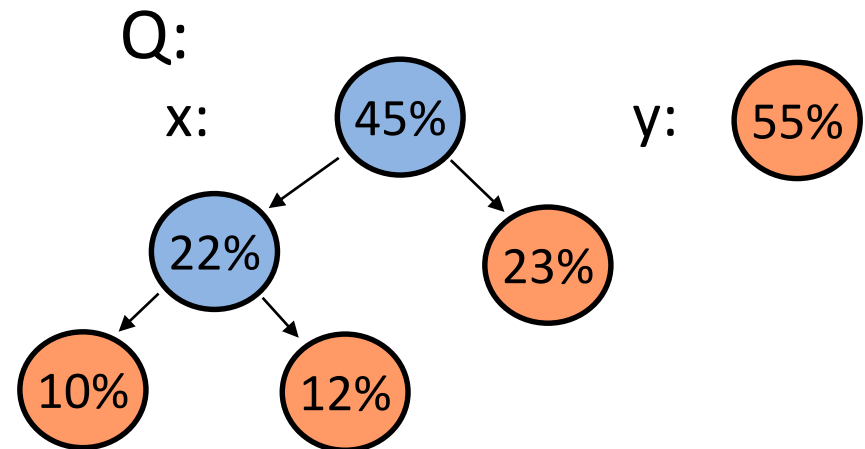
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

$i=3$

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



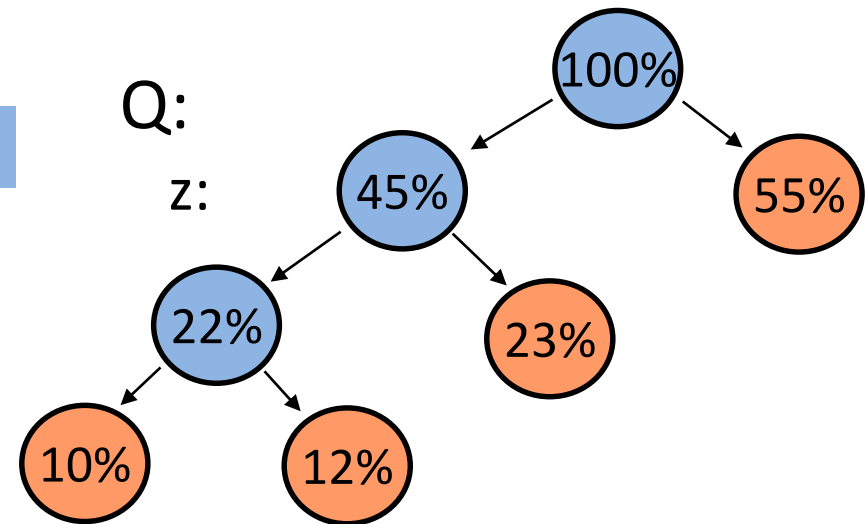
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=3$



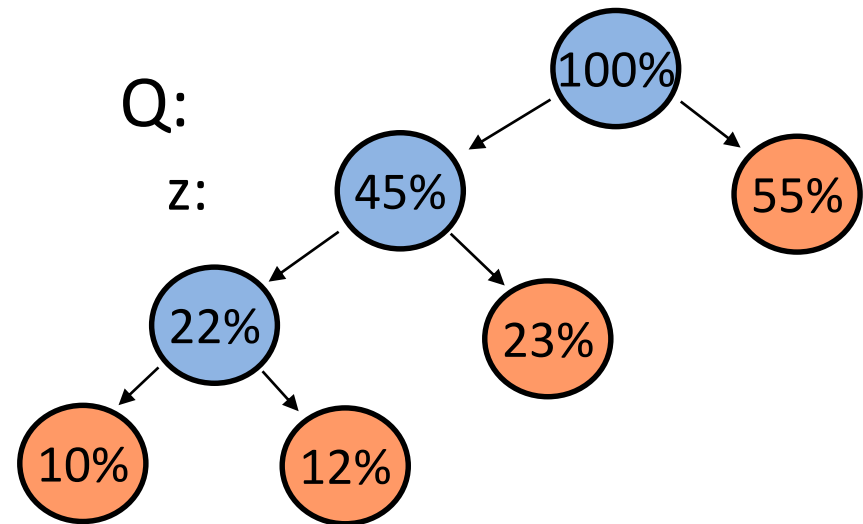
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

$i=3$

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



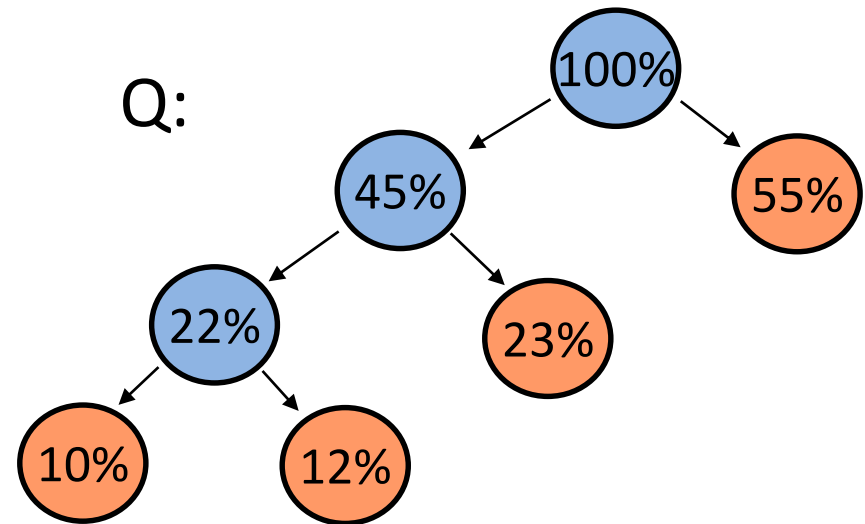
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=3$



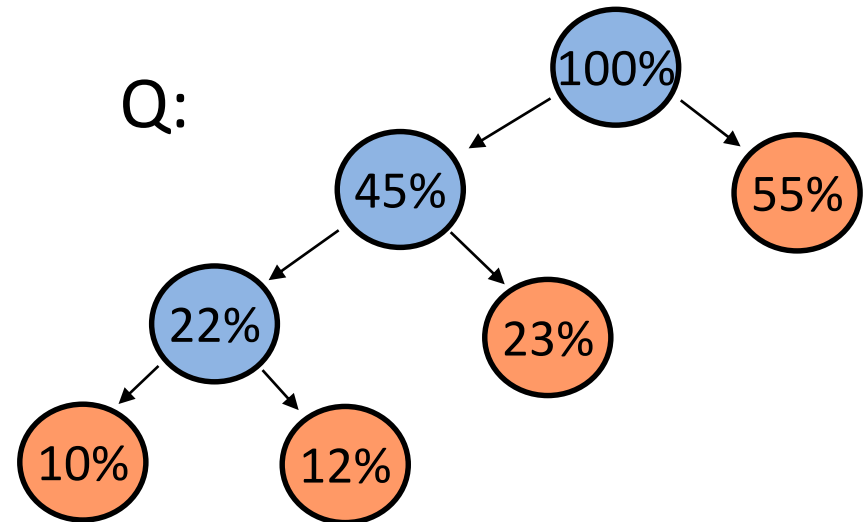
Greedy Algorithmen – Datenkompression

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. $x \leftarrow \text{deleteMin}(Q)$
5. $y \leftarrow \text{deleteMin}(Q)$
6. $z \leftarrow \text{new BinTree}(x, f[x]+f[y], y)$
7. $f[z] \leftarrow f[x] + f[y]$
8. $Q \leftarrow Q \cup \{z\}$
9. **return** $\text{deleteMin}(Q)$

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

$i=3$



Theorem

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

Theorem

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

- 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 zwei Zeichen mit Frequenz $1/2^{k_{\min}}$.

Optimale durchschn. Codewortlänge

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

Durchschnittliche Codewortlänge eines Huffman-Codes

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

Häufigkeitsverteilung / Wahrscheinlichkeitsvert. $p(x)$

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

Entropie $H(X)$

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

- Untere Schranke für die optimale durchschn. Codewortlänge
- Im Grenzwert (genug lange Zeichenketten) kann man mit durchschn. 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 😊
