SS09

Effiziente Algorithmen

3. Kapitel: Greedy-Algorithmen (2. Teil)

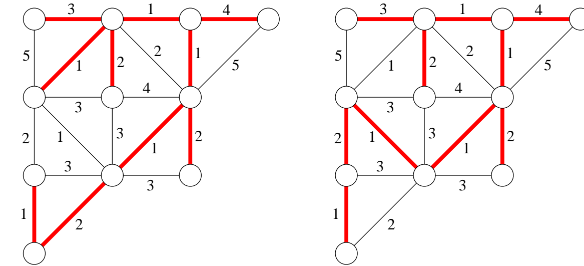
Martin Dietzfelbinger

Juni 2009

Ein Spannbaum $T \subseteq E$ für G heißt ein **minimaler Spannbaum (MST)**, wenn

$$c(T) = \min\{c(T') \mid T' \text{ Spannbaum von } G\},$$

d.h. wenn er minimale Kosten unter allen Spannbäumen hat.



Zwei minimale Spannbäume, jeweils mit Gesamtgewicht 18.

3.4 Algorithmus von Kruskal

Auch dieser Algorithmus löst das **MST-Problem**.

Anderer Ansatz als bei Jarník/Prim:

Probiere Kanten in **aufsteigender Reihenfolge** des Kantengewichts, und wähle eine Kante für den zu bauenden MST, wenn sie die Kreisfreiheitseigenschaft nicht verletzt.

Algorithmus von Kruskal

1. Schritt: Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c(e_1), \ldots, c(e_m)$ aufsteigend. – Also O.B.d.A.: $c(e_1) \le \ldots \le c(e_m)$.

2. Schritt: Setze $\mathtt{R} \leftarrow \emptyset$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

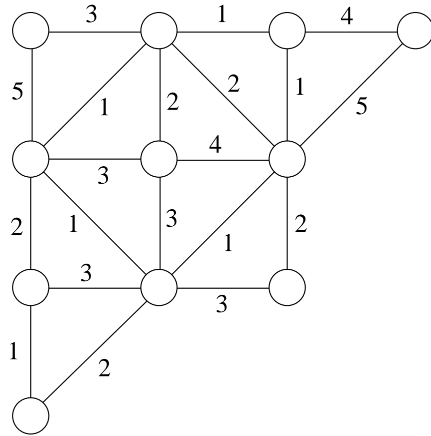
Falls $\mathtt{R} \cup \{e_i\}$ kreisfrei ist, setze $\mathtt{R} \leftarrow \mathtt{R} \cup \{e_i\}$

($*$ sonst, d.h. wenn e_i einen Kreis schließt, bleibt $\mathtt{R}$ unverändert $*$)

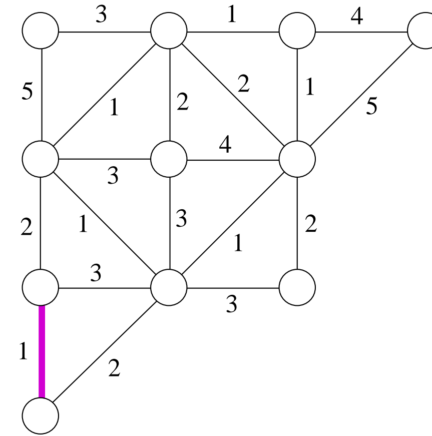
($*$ Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$. $*$)

4. Schritt: Die Ausgabe ist $\mathtt{R}$.

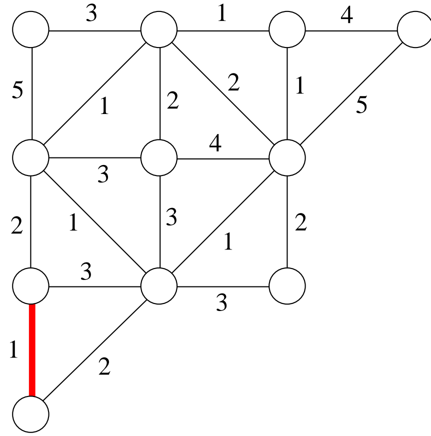
Beispiel (Kruskal):



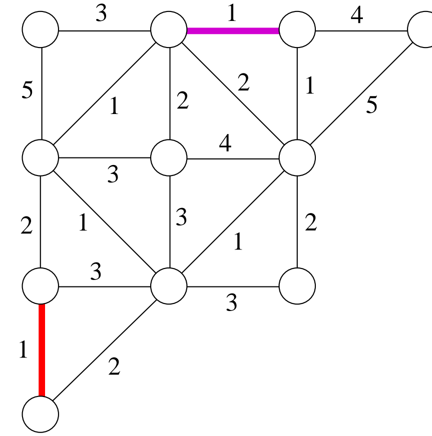
Beispiel (Kruskal):



Beispiel (Kruskal):



Beispiel (Kruskal):



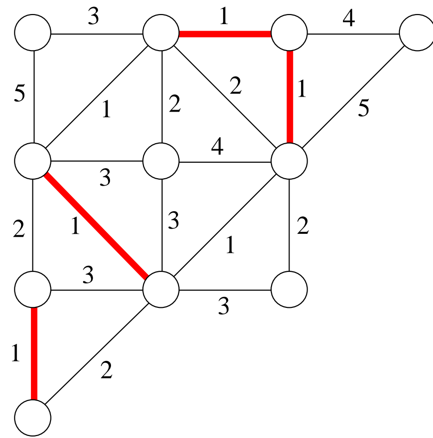
Beispiel (Kruskal):

Beispiel (Kruskal):

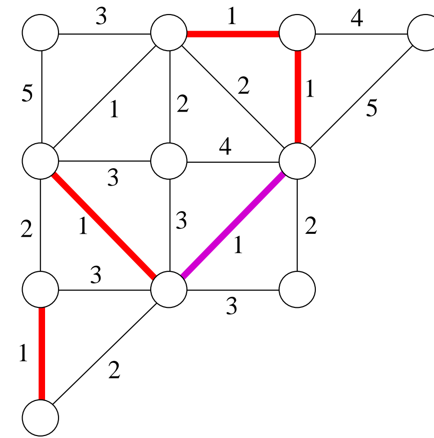
Beispiel (Kruskal):

Beispiel (Kruskal):

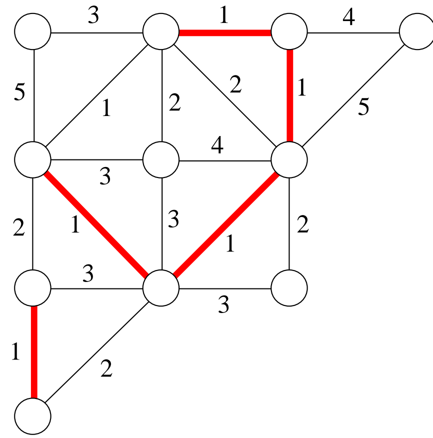
Beispiel (Kruskal):



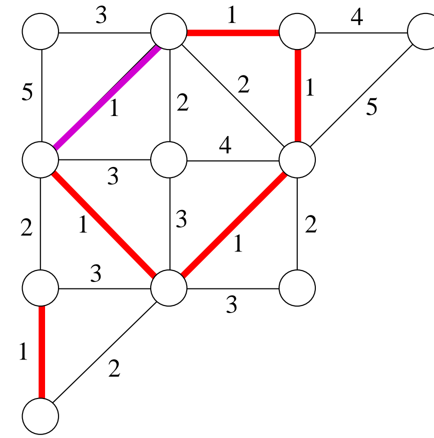
Beispiel (Kruskal):



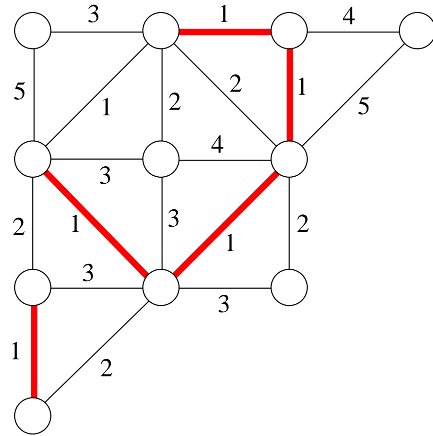
Beispiel (Kruskal):



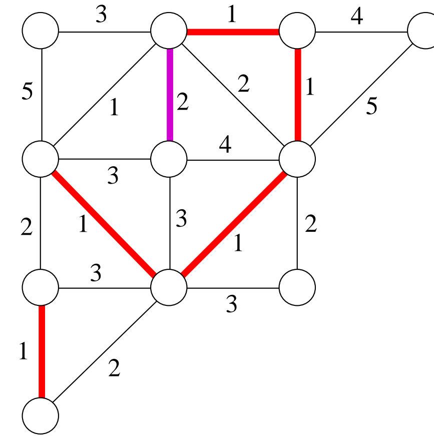
Beispiel (Kruskal):



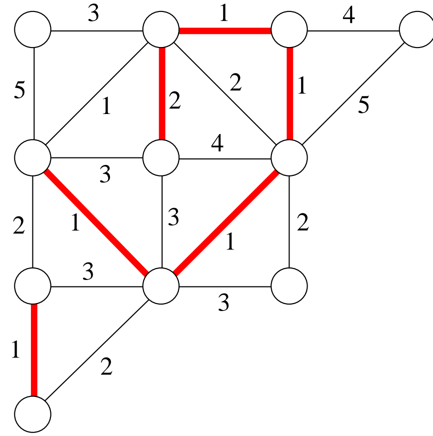
Beispiel (Kruskal):



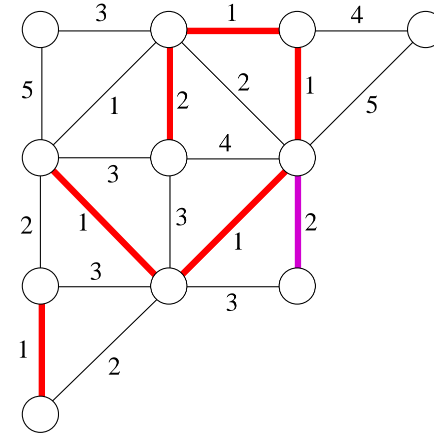
Beispiel (Kruskal):



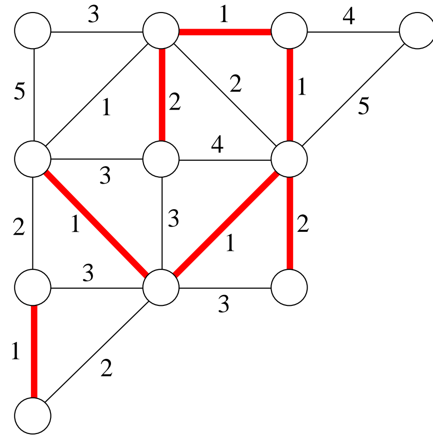
Beispiel (Kruskal):



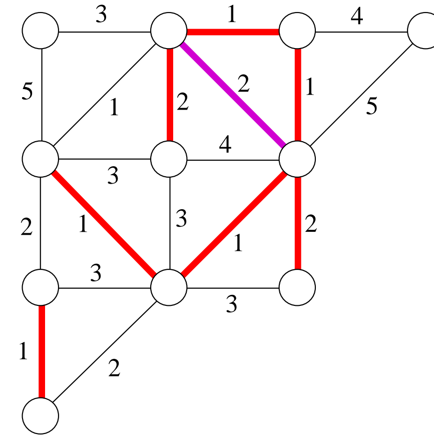
Beispiel (Kruskal):



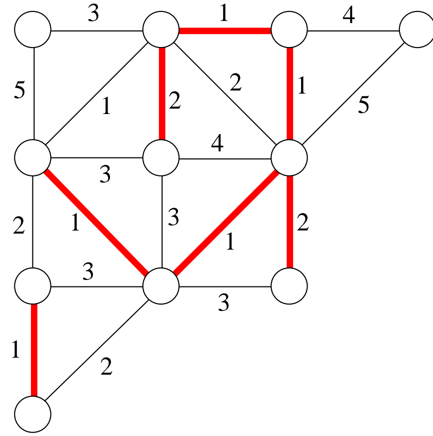
Beispiel (Kruskal):



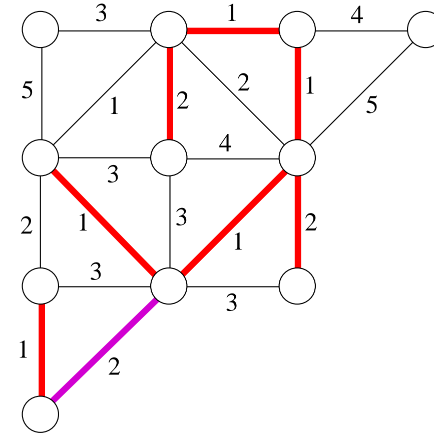
Beispiel (Kruskal):



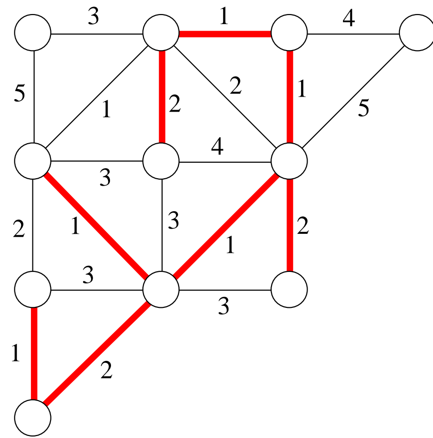
Beispiel (Kruskal):



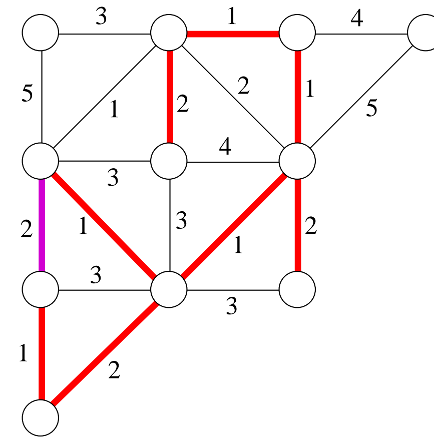
Beispiel (Kruskal):



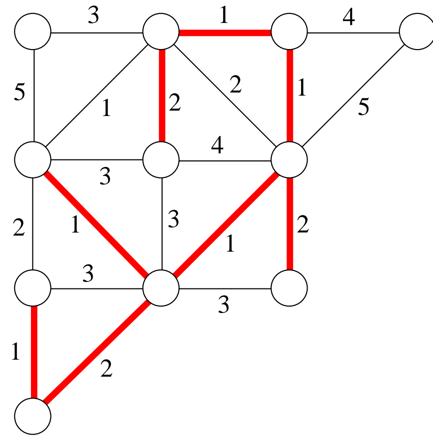
Beispiel (Kruskal):



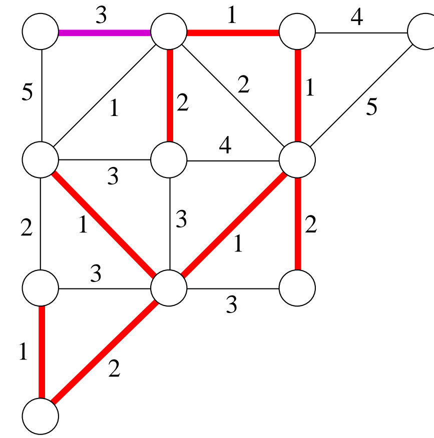
Beispiel (Kruskal):



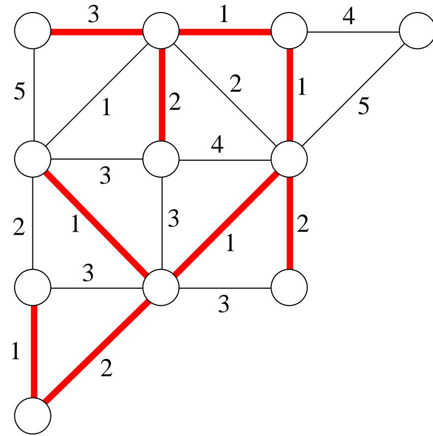
Beispiel (Kruskal):



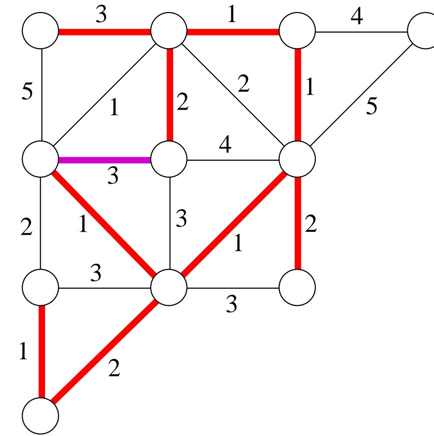
Beispiel (Kruskal):



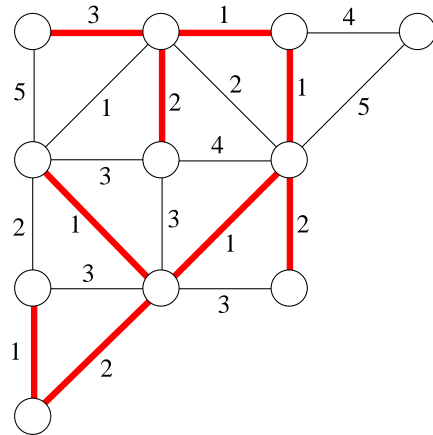
Beispiel (Kruskal):



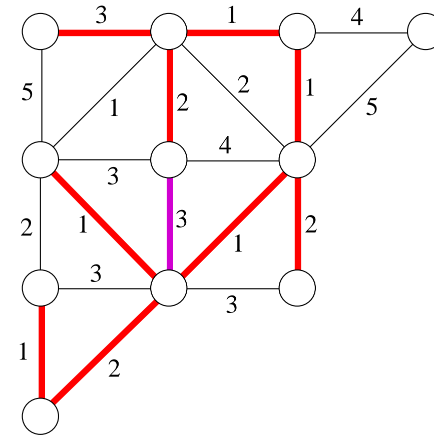
Beispiel (Kruskal):



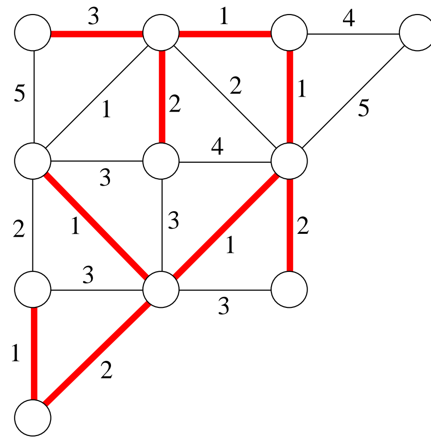
Beispiel (Kruskal):



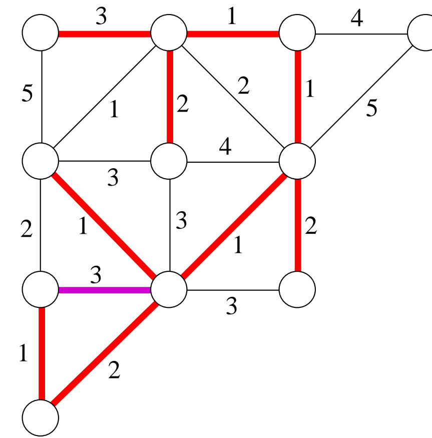
Beispiel (Kruskal):



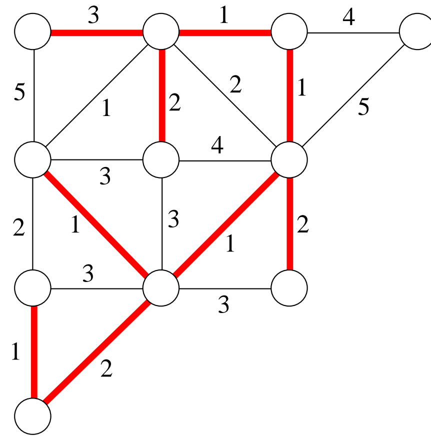
Beispiel (Kruskal):



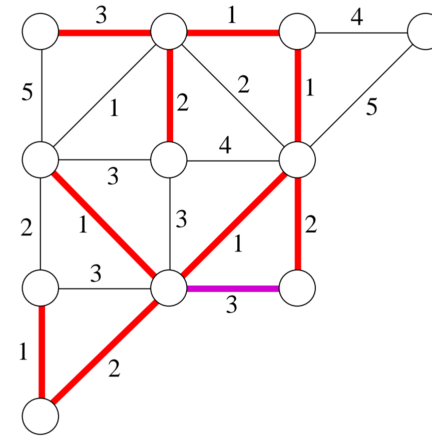
Beispiel (Kruskal):



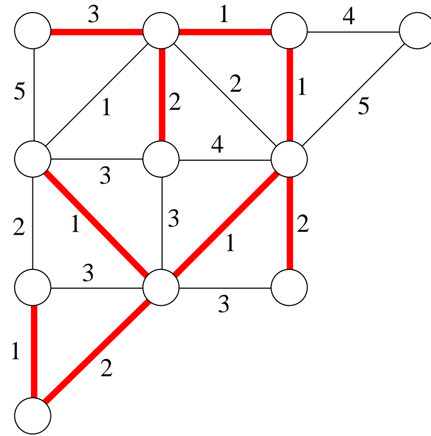
Beispiel (Kruskal):



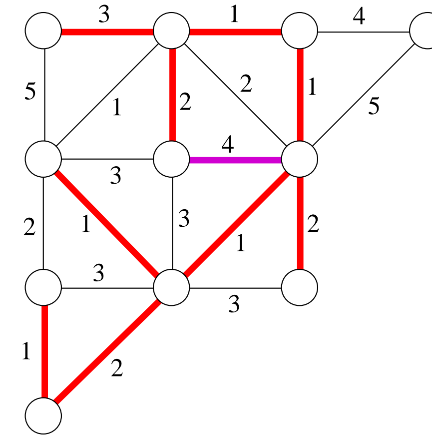
Beispiel (Kruskal):



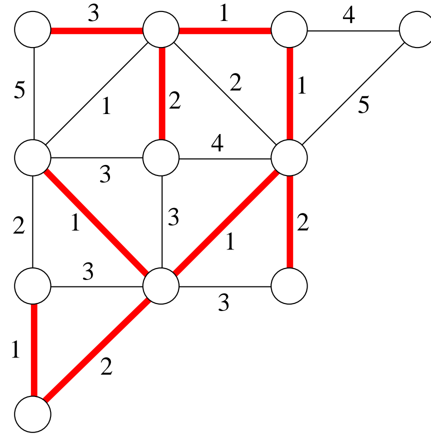
Beispiel (Kruskal):



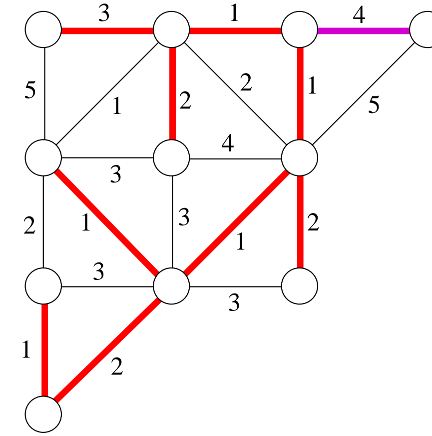
Beispiel (Kruskal):



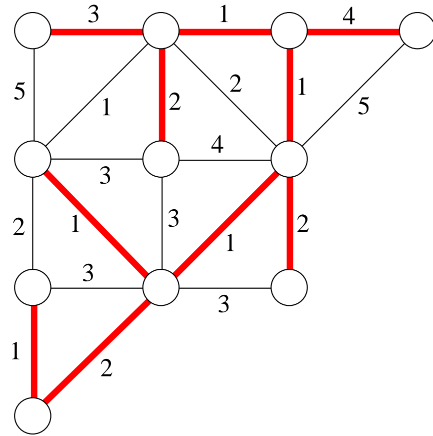
Beispiel (Kruskal):



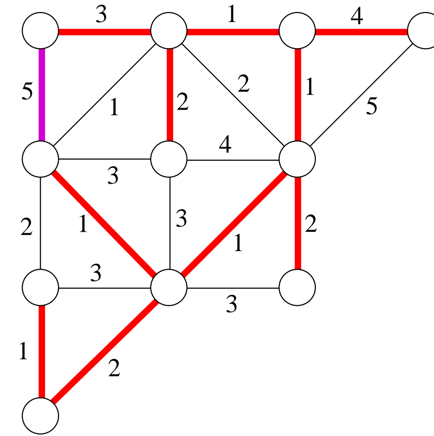
Beispiel (Kruskal):



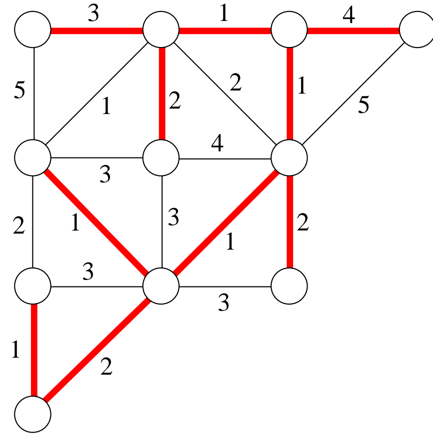
Beispiel (Kruskal):



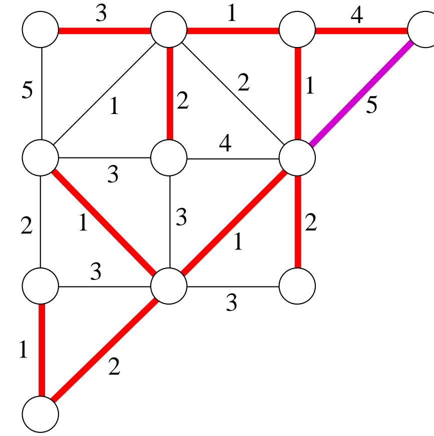
Beispiel (Kruskal):



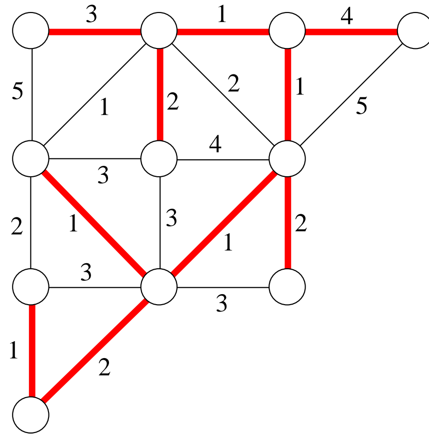
Beispiel (Kruskal):



Beispiel (Kruskal):



Beispiel (Kruskal):



Uns interessieren: 1) Korrektheit; 2) Laufzeit (später)

Korrektheit des Algorithmus von Kruskal:

R_i: Kantenmenge, die nach Runde i in R steht.

Weil dies im 3. Schritt getestet wird, ist sichergestellt, dass jedes R_i kreisfrei, also ein Wald ist.

Zu zeigen: R_m ist ein MST für G.

Erinnerung: $R \subseteq E$ heißt **erweiterbar**, wenn $R \subseteq T$ für einen MST T.

Induktionsbehauptung IB(i):
R_i ist erweiterbar.
(Beweis gleich, durch Induktion über $i = 0, 1, \ldots, m$.)

Dann besagt IB(m), dass $R_m \subseteq T$ ist für einen MST T.

Beh.: Es gilt auch $T \subseteq R_m$.
(Also gilt Gleichheit, und R_m ist ein MST.)

(*Beweis* der Beh.: Sei $e \in T$. Dann ist $e = e_i$ für ein i, und e_i wurde in Runde i getestet. Weil $R_{i-1} \subseteq R_m \subseteq T$ und $e_i \in T$, ist $R_{i-1} \cup \{e_i\} \subseteq T$, also ist $R_{i-1} \cup \{e_i\}$ kreisfrei, also fügt der Algorithmus die Kante e_i in Runde i zu R hinzu, also gilt $e_i \in R_m$.)

Induktionsbehauptung IB(i): R_i ist erweiterbar.

I.A.: $R_0 = \emptyset$ ist erweiterbar.

I.V.: $1 \leq i < m$ und R_{i-1} ist erweiterbar.

I.S.: Runde i wird mit Kante e_i ausgeführt.

1. Fall: $R_{i-1} \cup \{e_i\}$ enthält einen Kreis, dann ist $R_{i-1} = R_i$, also R_i erweiterbar.

2. Fall: $R_{i-1} \cup \{e_i\}$ ist kreisfrei.

Sei $e_i = (v, w)$. Wir definieren:

$S := \{u \in V \mid$ im Wald (V, R_{i-1}) sitzen u, v im selben Baum$\}$
Dann (Kreisfreiheit): $w \in V - S$; und (klar):
$(*)$ Es gibt keine Kante in R_{i-1}, die S mit $V - S$ verbindet.

Behauptung: $c(e_i)$ ist minimal unter allen $c((x, y))$, $x \in S$, $y \in V - S$.

Behauptung: $c(e_i)$ ist minimal unter allen $c((x,y))$, $x \in S$, $y \in V - S$.

Beweis indirekt: **Annahme:** $c(e_j) < c(e_i)$ und $e_j = (v_j, w_j)$ verbindet S mit $V - S$, etwa $v_j \in S$, $w_j \in V - S$.

$\Rightarrow$ $j < i$ (weil Kanten aufsteigend sortiert sind)

$\Rightarrow$ e_j ist in Runde $j < i$ schon untersucht worden.

Mit $(*)$ und $R_j \subseteq R_{i-1}$ folgt $e_j \notin R_j$.

$\overset{\text{Algo}}{\Longrightarrow}$ $R_{j-1} \cup \{e_j\}$ enthält einen Kreis.

D.h.: $e_j = (v_j, w_j)$ schließt einen Kreis, also gibt es in R_{j-1} einen Weg von $v_j \in S$ nach $w_j \in V - S$. Auf diesem Weg muss es eine Kante geben, die einen Knoten in S mit einem Knoten in $V - S$ verbindet, **Widerspruch** zu $(*)$.

Gesehen: $c(e_i)$ ist minimal unter allen $c((x,y))$, $x \in S$, $y \in V - S$.

Nach der Schnitteigenschaft folgt: $R_i = R_{i-1} \cup \{e_i\}$ ist erweiterbar,

und das ist die Induktionsbehauptung.

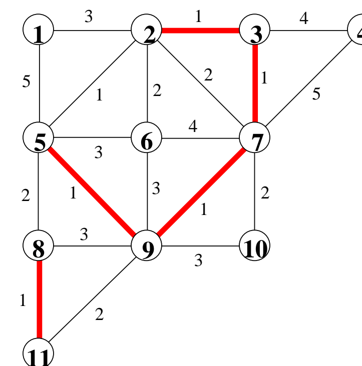
Ende des Korrektheitsbeweises für den Algorithmus von Kruskal.

3.5 Union-Find-Datenstrukturen

Union-Find-Datenstrukturen dienen als **Hilfsstruktur** für verschiedene Algorithmen, insbesondere für den Algorithmus von Kruskal.

Diese Verfahren haben zudem eine instruktive Analyse.

Eine Zwischenkonfiguration im Algorithmus von Kruskal besteht in einer Menge $F \subseteq E$ von Kanten, die einen Wald bilden, sowie einer Folge $e_i, \ldots, e_m$ von noch zu verarbeitenden Kanten.



Die im Algorithmus von Kruskal zu lösende Aufgabe: zu zwei Knoten i und j entscheide, ob es in (V, F) einen Weg von i nach j gibt.
Möglich, aber ungeschickt: Jedesmal Tiefensuche o.ä.

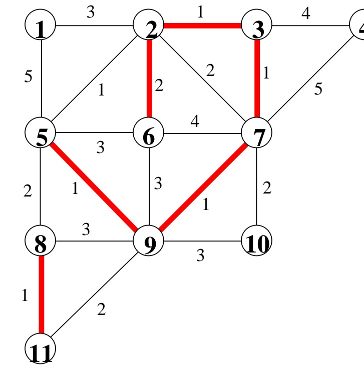
Ansatz: Repräsentiere die **Knotenmengen**, die den Zusammenhangskomponenten von (V, F) entsprechen.

Im Beispielbild:

$\{1\}, \{2,3,5,7,9\}, \{4\}, \{6\}, \{8,11\}, \{10\}$.

Es soll dann **schnell zu ermitteln** sein, ob zwei Knoten in **derselben Komponente (Menge)** liegen.

Wenn wir einen Schritt des Kruskal-Algorithmus ausführen, bei dem eine Kante akzeptiert, also in F aufgenommen wird, müssen wir zwei der disjunkten Mengen **vereinigen**.



Neue Mengen:

$\{1\}, \{2,3,5,6,7,9\}, \{4\}, \{8,11\}, \{10\}$.

Auch diese Operation sollte „schnell" durchführbar sein.

Abstrakte Aufgabe:

Verwalte eine **„dynamische Partition"** der Menge $\{1, 2, \ldots, n\}$ unter Operationen

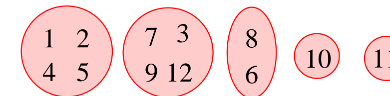
init (Anfangsinitialisierung)
union (Vereinigung von Klassen)
find („In welcher Klasse liegt i?").

Eine **Partition** ist dabei eine **Zerlegung** von $\{1, 2, \ldots, n\}$ in Mengen

$$\{1, 2, \ldots, n\} = S_1 \cup S_2 \cup \cdots \cup S_\ell,$$

wobei $S_1, S_2, \ldots, S_\ell$ **disjunkt** sind.

Beispiel: $n = 12$.



Eine **Klasse** in einer solchen Partition ist eine dieser Mengen $S_1, \ldots, S_\ell$.

Erinnerung: Eine Partition ist gleichbedeutend mit einer **Äquivalenzrelation** $\sim$ über $\{1, \ldots, n\}$, wobei $i \sim j$ einfach heißt, dass i, j in derselben Menge liegen. Die Mengen sind dann genau die Äquivalenzklassen zu $\sim$.

(Bitte die Wörter „Klasse" und „Partition" nicht verwechseln, auch wenn im Computerjargon bei Festplatten „Partition" im Sinn von „ein Teil von mehreren" verwendet wird.)

Wir betrachten **dynamische** Partitionen, die durch Operationen veränderbar sind.

In jeder Klasse K der Partition soll ein **Repräsentant** $r \in K$ ausgezeichnet sein. Dieser fungiert als **Name** von K. Wir schreiben K_r für die Klasse mit dem Repräsentanten r.

Beispiel: Für $n = 12$ bilden die folgenden 5 Klassen eine Partition mit Repräsentanten:

1 2 4 5 — K_4; 7 3 9 12 — K_7; 8 6 — K_6; 10 — K_{10}; 11 — K_{11}

Name	Klasse	Repräsentant
K_4	$\{1,2,4,5\}$	4
K_6	$\{6,8\}$	6
K_7	$\{3,7,9,12\}$	7
K_{10}	$\{10\}$	10
K_{11}	$\{11\}$	11

Eine solche Partition mit Repräsentanten ist mathematisch vollständig beschrieben durch eine Funktion

$$r : \{1,\dots,n\} \to \{1,\dots,n\}$$

mit folgender Eigenschaft:

$$r(r(i)) = r(i), \quad \text{für alle } i \in \{1,\dots,n\}.$$

Das Element i gehört zur Klasse $K_{r(i)} = \{j \mid r(i) = r(j)\}$; diese hat den gemeinsamen Wert $r(i)$ als Repräsentanten.

Im Beispiel sieht r folgendermaßen aus:

i	1	2	3	4	5	6	7	8	9	10	11	12
$r(i)$	4	4	7	4	4	6	7	6	7	10	11	7

Operationen:

init(n): Erzeugt zu $n \geq 1$ die „diskrete Partition“ mit den n einelementigen Klassen $\{1\},\{2\},\dots,\{n\}$, also $K_i = \{i\}$.

find(i): Gibt zu $i \in \{1,\dots,n\}$ den Namen $r(i)$ der Klasse $K_{r(i)}$ aus, in der sich i (gegenwärtig) befindet.

union(s,t): Die Argumente s und t müssen Repräsentanten **verschiedener Klassen** K_s bzw. K_t sein. Die Operation ersetzt in der Partition K_s und K_t durch $K_s \cup K_t$. Als Repräsentant von $K_s \cup K_t$ kann entweder s oder t verwendet werden.

Im obigen Beispiel werden durch **union**$(4,10)$ die Klassen $K_4 = \{1,2,4,5\}$ und $K_{10} = \{10\}$ entfernt und eine neue Klasse

$$K'_{10} = \{1,2,4,5,10\}$$

hinzugefügt.

Aus Sicht der r-Funktion entspricht diese Operation der Änderung der Funktionswerte auf der Klasse K_s:

$$r'(i) := \begin{cases} t & , \text{ falls } r(i) = s, \\ r(i) & , \text{ sonst.} \end{cases}$$

Im Beispiel: Die r-Werte in S_4 werden von 4 auf 10 geändert, die anderen bleiben gleich.

i	1	2	3	4	5	6	7	8	9	10	11	12
$r'(i)$	**10**	**10**	7	**10**	**10**	6	7	6	7	10	11	7

Nützliche abgeleitete Operationen:

same_class(i, j): Liefert *true*, wenn **find**(i) = **find**(j), *false* sonst.

test_and_union(i, j): Berechnet s = **find**(i) und t = **find**(j) und führt dann **union**(s, t) aus, falls $s \neq t$ ist.

Zwei Implementierungsmöglichkeiten:

1) **Arrays mit Listen** (erlaubt schnelle **find**s)
2) **Wurzelgerichteter Wald** (erlaubt schnelle **union**s)

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt: Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Sortierte Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

s ← **find**(v_i); t ← **find**(w_i);
if s ≠ t **then begin** R ← R ∪ $\{e_i\}$; **union**(s,t) **end**;
(∗ Optional: Beende Schleife, wenn $|\mathtt{R}| = n - 1$. ∗)

4. Schritt: Die Ausgabe ist R.

Satz 3.5.3

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Laufzeit des Algorithmus ist $O(m \log m) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

(c) Die Laufzeit des Algorithmus ist $O(m \log m) + O(m \log^* n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit einem **wurzelgerichteten Wald** mit **Pfadkompression** implementiert.

Bem.: In (b) und (c) steht der Summand $O(m \log m)$ für die Kosten des Sortierens.

Beweis: (a) Man zeigt durch Induktion über die Runden, dass nach i Runden die Klassen der Union-Find-Struktur genau die Zusammenhangskomponenten des Graphen $(V, \{e_1, \ldots, e_i\})$ sind, und dies sind die Knotenmengen der Bäume im Wald (V, R_i) (Inhalt von R nach i Durchläufen).

Daher testet „s ← **find**(v_i); t ← **find**(w_i); **if** s ≠ t ...“ korrekt die Kreiseigenschaft.

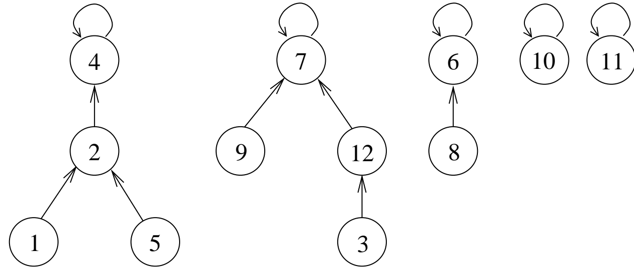
(b), (c): Der erste Term $O(m \log m)$ benennt die Kosten für das Sortieren. Danach sind $2m$ **find**-Operationen und $n - 1$ **union**-Operationen durchzuführen. Die Zeiten hierfür in beiden Implementierungen werden unten begründet. □

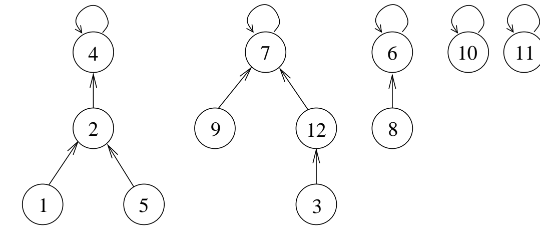
Bem.: Wenn die Kanten schon sortiert sind oder billig sortiert werden können ($O(m)$ Zeit, z. B. mittels Radixsort), und $G = (V, E)$ sehr wenige Kanten hat, ergibt Union-Find mit Pfadkompression eine sehr günstige Laufzeit $O(m \log^* n)$: fast linear.

Baumimplementierung von Union-Find

Eine Implementierung der Union-Find-Datenstruktur verwendet einen **wurzelgerichteten Wald**.

Beispiel: Partition $\{1,2,4,5\},\{3,7,9,12\},\{6,8\},\{10\},\{11\}$ wird dargestellt durch:





Für jede Klasse K_s gibt es genau einen Baum B_s.

Jedes Element $i \in K_s$ ist Knoten im Baum.

Es gibt nur Zeiger in Richtung auf die Wurzel zu: $p(i)$ ist der Vorgänger von i; die Wurzel ist der Repräsentant s; die Wurzel zeigt auf sich selbst als „Vorgänger“: $p(i) = i$ genau dann wenn i Repräsentant ist.

Leicht einzusehen :

Eine Funktion $p\colon \{1,\ldots,n\} \to \{1,\ldots,n\}$ stellt einen wurzelgerichteten Wald dar

genau dann wenn

für jedes i die Folge $i_0 = i, i_1 = p(i_0), \ldots, i_l = p(i_{l-1}), \ldots$ schließlich ein i_l mit $p(i_l) = i_l$ erreicht

(d. h.: die Vorgängerzeiger bilden keine Kreise der Länge > 1).

Eine kostengünstige Darstellung eines solchen wurzelgerichteten Waldes benutzt nur ein Array `p[1..n]: array of int`; für Knoten i gibt der Eintrag `p[i]` den Vorgängerknoten $p(i)$ an.

Das dem Wald im Beispiel entsprechende Array sieht so aus:

	1	2	3	4	5	6	7	8	9	10	11	12
p :	2	4	12	4	2	6	7	6	7	10	11	7

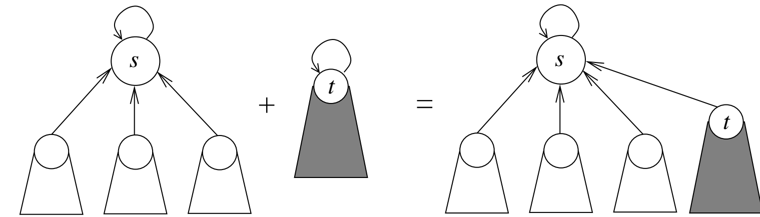
Implementierung von **find**(i):

Prozedur find(i)
(1) $\mathtt{j} \leftarrow i$;
(∗ verfolge Vorgängerzeiger bis zur Wurzel ∗)
(2) $\mathtt{jj} \leftarrow \mathtt{p[j]}$;
(3) **while** $\mathtt{jj} \neq \mathtt{j}$ **do**
(4) $\mathtt{j} \leftarrow \mathtt{jj}$;
(5) $\mathtt{jj} \leftarrow \mathtt{p[j]}$;
(6) **return** j.

Zeitaufwand: $\Theta(\mathit{depth}(i)) = \Theta(\text{Tiefe von } i \text{ in seinem Baum})$.

union(s,t): Gegeben: Verschiedene Repräsentanten s und t.

Man macht einen der Repräsentanten zum Sohn des anderen, setzt also $\mathtt{p}(s) := t$ **oder** $\mathtt{p}(t) := s$.



? Welche der beiden Optionen sollte man nehmen?

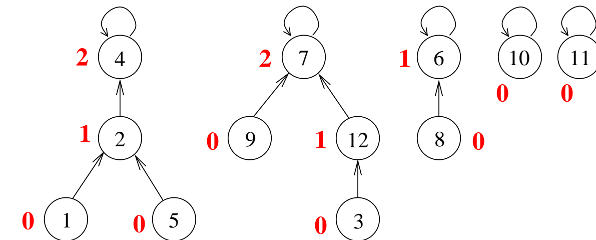
Ungeschickt: **union**$(i, i+1)$, $i = 1, \ldots, n-1$, dadurch ausführen, dass man nacheinander $\mathtt{p}(i) := i+1$ ausführt.

Dies führt zu dem Baum

find-Operationen sind nun teuer! Ein **find**(i) für jedes $i \in \{1, \ldots, n\}$ hat Gesamtkosten $\Theta(n^2)$!

Trick: Führe für jeden Knoten i eine Zahl *rank*(i) **(Rang)** mit, in einem Array $\mathtt{rank}[1..n]$: `array of int`, mit

$$\mathtt{rank}[i] = \text{Tiefe des Teilbaums mit Wurzel } i.$$



Bei **union**(s,t) machen wir die Wurzel desjenigen Baums zur Wurzel der Vereinigung, die den größeren Rang hat („*union by rank*“). Bei gleichen Rängen ist die Reihenfolge gleichgültig – **(genau) in diesem Fall** steigt der Rang des Wurzelknotens an.

Operationen:

Prozedur init(n) (∗ Initialisierung ∗)
(1) Erzeuge p, rank: Arrays der Länge n für int-Einträge
(2) **for** i **from** 1 **to** n **do**
(3) p[i] ← i; rank[i] ← 0;
(∗ n Bäume mit je einem (Wurzel-)Knoten vom Rang 0 ∗)

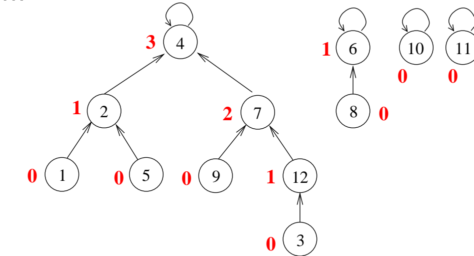
Zeitaufwand: $\Theta(n)$.

Prozedur union(s,t)
(∗ Ausgangspunkt: s,t sind verschiedene Repräsentanten von Klassen ∗)
(∗ D.h.: p[s] $= s$ und p[t] $= t$ ∗)
(1) **if** rank[s] > rank[t]
(2) **then** p[t] ← s
(3) **elseif** rank[t] > rank[s]
(4) **then** p[s] ← t
(5) **else** p[t] ← s; rank[s] ← rank[s] + 1.

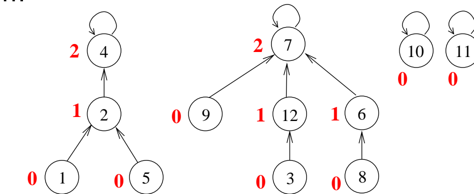
Zeitaufwand: $O(1)$.

Beispiele:

union$(4,7)$ würde liefern:



union$(6,7)$ würde liefern:



Satz 3.5.1

Die eben beschriebene Implementierung einer Union-Find-Struktur mit einem wurzelgerichteten Wald hat folgende Eigenschaften:

a) Sie ist korrekt (d. h. hat das vorgesehene Ein-/Ausgabeverhalten).

b) **init**(n) benötigt Zeit $\Theta(n)$;
find(i) benötigt Zeit $O(\log n)$;
union(s,t) benötigt Zeit $O(1)$.

Beweis:

(a) Betrachte eine Operationenfolge $Op_0 = init(n)$, Op_1, . . . , Op_k, wobei die letzten k Operationen **legale union**-Operationen sind.

Man beweist durch Induktion über ℓ, dass nach **init**(n) und nach jedem Op_ℓ gilt:

Das p-Array stellt diejenige Partition als wurzelgerichteten Wald dar, die von $Op_0 = init(n)$, Op_1, . . . , Op_ℓ erzeugt wird. Weiter ist size$[t] = |K_t|$ für jeden Repräsentanten t.

Aus der Behauptung folgt dann auch, dass die **find**-Operationen immer die korrekte Ausgabe liefern.

Beweis: (Fortsetzung)

(b) Wir beobachten:

Fakt 1: $i \neq p(i) \Rightarrow \mathit{rank}(i) < \mathit{rank}(p(i))$.

Beweis: Wenn eine Wurzel s Kind von t wird, dann ist entweder $\mathit{rank}(s) < \mathit{rank}(t)$ (und das bleibt so) oder es ist $\mathit{rank}(s) = \mathit{rank}(t)$, und der Rang von t erhöht sich nun um 1.

Bei Knoten, die keine Wurzeln mehr sind, ändern sich Vorgänger und Ränge nicht mehr.

Fakt 2: Wenn s Wurzel des Baums B_s ist und $h = \mathit{rank}(s)$ gilt, dann enthält B_s mindestens 2^h Knoten.

Beweis: Ein Knoten vom Rang h entsteht, wenn zwei Bäume vereinigt werden, deren Wurzeln beide Rang $h-1$ haben. Daraus folgt Fakt 2 leicht durch Induktion über $h = 0, 1, \ldots$.

Fakt 3: Bei n Knoten insgesamt gibt es höchstens $n/2^h$ Knoten von Rang h.

Beweis: Wegen Fakt 1 können Knoten mit Rang h nicht Nachfahren voneinander sein. Also sind alle Unterbäume, deren Wurzeln Rang h haben, disjunkt. Aus Fakt 2 folgt leicht, dass auch für Knoten i im Inneren von Bäumen mit $\mathit{rank}(i) = h$ gilt, dass ihr Unterbaum mindestens 2^h Knoten hat. Also kann es nicht mehr als $n/2^h$ solche Knoten geben.

Aus Fakt 3 folgt, dass es keine Knoten mit Rang größer als $\log n$ geben kann.

Also hat kein Baum Tiefe größer als $\log n$, also kosten **find**-Operationen maximal Zeit $O(\log n)$. $\square$

Pfadkompression

Eine interessante Variante der Union-Find-Datenstruktur, die mit einem wurzelgerichteten Wald implementiert ist, ist der Ansatz der „**Pfadkompression**“ (oder „**Pfadverkürzung**“).

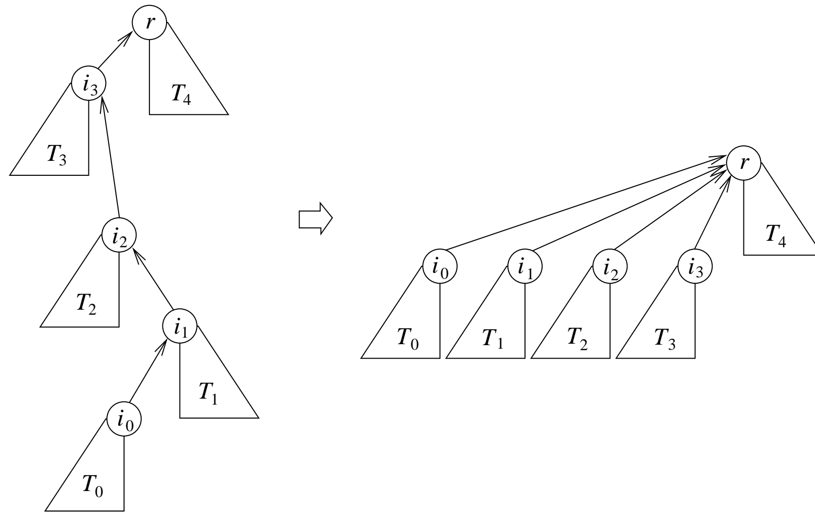
Bei einem **find**(i) muss man den ganzen Weg von Knoten i zu seiner Wurzel $r(i)$ ablaufen; Aufwand $O(\mathit{depth}(i))$.

Ein gutes Ziel ist also, diese Wege möglichst kurz zu halten.

Idee: Man investiert bei **find**(i) etwas mehr Arbeit, jedoch immer noch im Rahmen $O(\mathit{depth}(i))$, um dabei die **Wege zu verkürzen** und damit spätere **find**s billiger zu machen.

Jeder Knoten $i = i_0, i_1, \ldots, i_{d-1}$ auf dem Weg von i zur Wurzel $i_d = r(i)$ wird direkt als Kind an die Wurzel gehängt. Kosten pro Knoten/Ebene: $O(1)$, insgesamt also $O(\mathit{depth}(i))$.

Beispiel: $d = \mathit{depth}(i) = 4$.

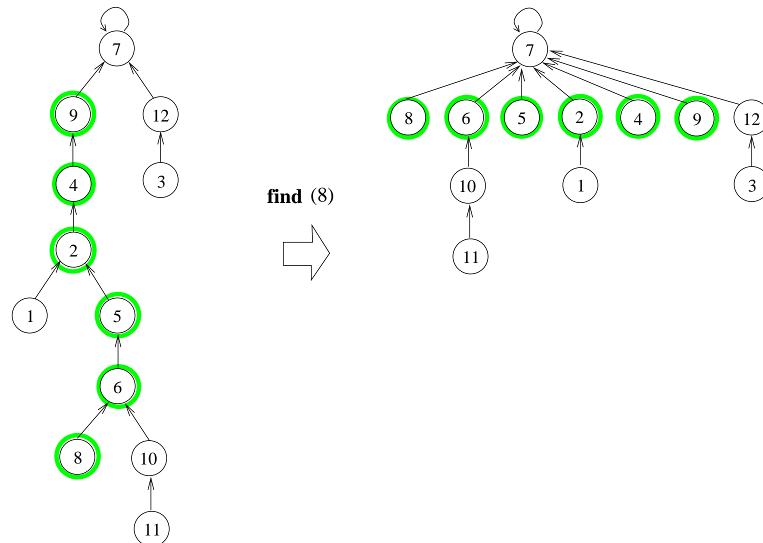


Die neue Version der **find**-Operation:

Prozedur find(i) (∗ Pfadkompressions-Version ∗)

```
(1)   j ← i;
      (∗ verfolge Vorgängerzeiger bis zur Wurzel r(i): ∗)
(2)   jj ← p[j];
(3)   while jj ≠ j do
(4)     j ← jj;
(5)     jj ← p[j] ;
(6)   r ← j; (∗ r enthält nun Wurzel r(i) ∗)
      (∗ Erneuter Lauf zur Wurzel, Vorgängerzeiger umhängen: ∗)
(7)   j ← i;
(8)   jj ← p[j];
(9)   while jj ≠ j do
(10)    p[j] ← r;
(11)    j ← jj;
(12)    jj ← p[j];
(13)  return r.
```


Beispiel:



Analyse: Interessant, neue Technik.

Die **union**-Operationen werden exakt wie bisher durchgeführt.

Beobachtungen: (1) Die Werte im `rank`-Array werden aktualisiert wie vorher. **Man kann sie aber nicht mehr als Baumtiefen interpretieren.**

(2) Nach (1) fällt bei einer **union**-Operationen die Entscheidung, welcher Knoten Kind eines anderen wird, exakt wie im Verfahren ohne Pfadkompression. Daher verändert die Pfadkompression die Knotenmengen der Bäume (also die Klassen) und die Repräsentanten nicht.

Aus (1) und (2) folgt, dass **Fakt 2** weiterhin gilt.

Feinheit: Nicht-Wurzeln vom Rang h können – weil bei der Pfadkompression Kindknoten abgehängt werden können – weniger als 2^h Nachfahren haben.

(3) Wenn ein Knoten i Kind eines anderen wird (also aufhört, Repräsentant zu sein), dann **ändert** sich sein **Rang**, wie in rank[i] gespeichert, **nie mehr**. Dies gilt in der Version ohne und in der Version mit Pfadkompression. **Diesen Wert werden wir als den „endgültigen" Rang von i ansehen.**

Solange ein Knoten Wurzel ist, ist sein Rang nicht endgültig und wird hier nicht berücksichtigt.

Die Rang-Werte sind damit in der Version mit und in der Version ohne Pfadkompression stets **identisch**.

Insbesondere: **Fakt 3** gilt weiterhin.

(4) Weder **union**- noch **find**-Operationen (mit Pfadkompression) ändern etwas an **Fakt 1**: Ränge wachsen strikt von unten nach oben entlang der Wege in den Bäumen.

(Beweis durch Induktion über ausgeführte Operationen, Übung.)

Definition: $F(0) := 1, \qquad F(i) := 2^{F(i-1)}$, für $i \geq 1$.

$F(i)$ ist die Zahl, die von einem „Zweierpotenzturm" der Höhe i berechnet wird:

$$F(i) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad i \text{ Zweien}$$

Beispielwerte:

i	0	1	2	3	4	5	6
$F(i)$	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$

Definition: $\log^* n := \min\{k \mid F(k) \geq n\}$.

Leicht zu sehen: $\log^* n = \min\{k \mid \underbrace{\log\log\ldots\log}_{k} n \leq 1\}$.

Nun teilen wir die Knoten j mit Rang > 0 (Nicht-Blätter, Nicht-Wurzeln) in **„Ranggruppen"** $G_0, G_1, G_2, \ldots$ ein:

Knoten j gehört zu Ranggruppe G_k,
wenn $F(k-1) < \mathit{rank}(j) \leq F(k)$ gilt.

G_0: Rang 1; G_1: Ränge $2, 3$; G_2: Ränge $4, \ldots, 16$;
G_3: Ränge $17, \ldots, 65536$; etc.

Wenn j in Ranggruppe G_k ist, folgt (mit Fakt 3):

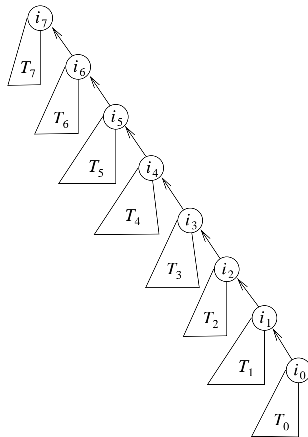
$F(k) = 2^{F(k-1)} < 2^{\mathit{rank}(j)} \leq 2^{\log n} = n$.

Also: $k < \log^* n$, d. h. es gibt maximal $\log^* n$ nichtleere Ranggruppen.

Beachte: $2^{65536} > 1000^{6553} > 10^{19500}$; Werte n mit $\log^* n > 5$ kommen in wirklichen algorithmischen Anwendungen nicht vor.
$\Rightarrow$ Man wird nie mehr als fünf nichtleere Ranggruppen sehen.

(Es gilt aber: $\lim_{n\to\infty} \log^* n = \infty$.)

Wir wollen nun die Kosten von m **find**-Operationen berechnen. Bei **find**(i) läuft man einen Weg $i = i_0, i_1 = p(i_0), i_2 = p(i_1), \ldots, i_d = p(i_{d-1})$ entlang, von i bis zur Wurzel $r(i) = i_d$. – *Beispiel*:



Wir veranschlagen Kosten $d + 1$ für diese Operation, also 1 für jeden Knoten j auf dem Weg.

Die **Rechenzeit** für alle m **find**s zusammen ist dann O(Summe aller „Kosten“.)

Für die Kostenanalyse benutzen wir die sogenannte **Bankkontomethode**, in einer anschaulichen Form.

Ein Knoten in Ranggruppe k bekommt $F(k)$ Euros Taschengeld, in dem Moment, in dem er aufhört, Repräsentant/Baumwurzel zu sein, sein Rang also endgültig feststeht.

In G_k sind Knoten mit Rängen $F(k-1)+1, \ldots, F(k)$. Nach Fakt 1 gilt:

$$|G_k| \le \frac{n}{2^{F(k-1)+1}} + \frac{n}{2^{F(k-1)+2}} + \cdots + \frac{n}{2^{F(k)}} < \frac{n}{2^{F(k-1)}} = \frac{n}{F(k)}.$$

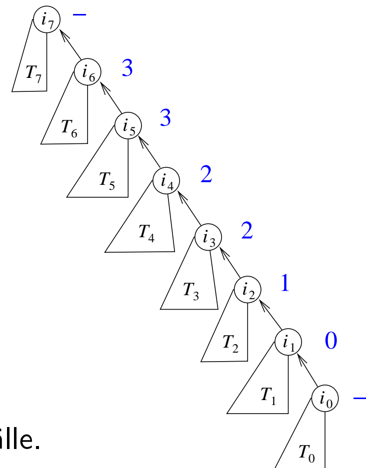
Also bekommen alle Knoten in G_k zusammen höchstens n Euros, und in allen höchstens $\log^* n$ Ranggruppen zusammen werden nicht mehr als $n \log^* n$ Euros als Taschengeld ausgezahlt.

Dies ist ein entscheidender Punkt in der Analyse: Die Ranggruppen sind sehr klein, daher kann man sich ein großzügiges Taschengeld leisten.

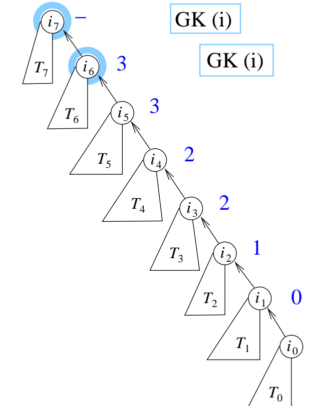
Weiter legen wir $m(2 + \log^* n)$ Euros in einer „Gemeinschaftskasse“ bereit.

Ziel: Wir wollen zeigen, dass das Taschengeld und das Geld in der Gemeinschaftskasse zusammen ausreichen, um die Kosten aller m **find**-Operationen zu bestreiten.

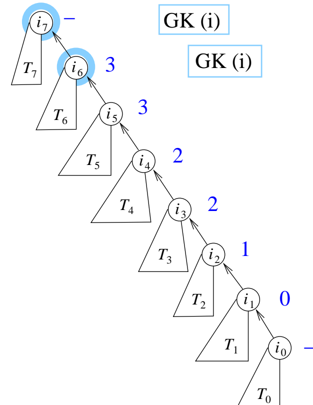
Wir betrachten eine **find**(i)-Operation. Jeder Knoten j auf dem Weg verursacht Kosten 1.



Es gibt drei Fälle.

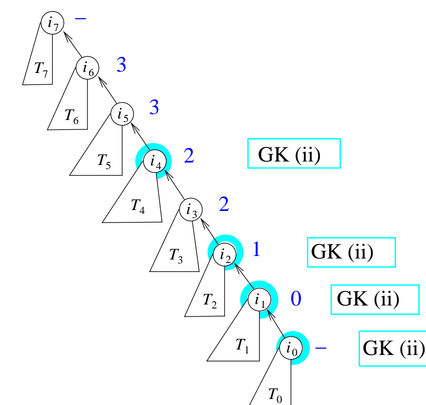


(i) j ist **Wurzel** oder ist der **vorletzte** Knoten auf dem Weg von i nach $r(i)$.
Die Kosten von j bestreiten wir aus der Gemeinschaftskasse.

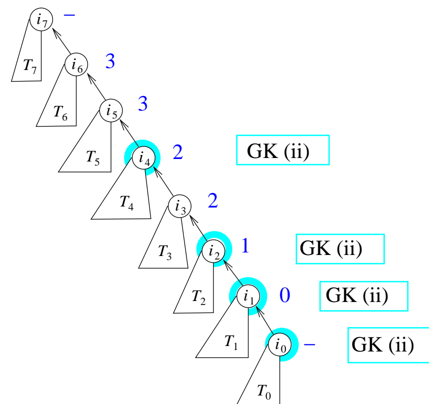


Dieser Fall kostet maximal 2 Euros für die **find**(i)-Operation.

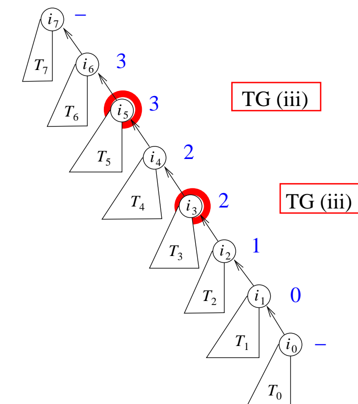
Ab hier: j ist nicht Wurzel oder vorletzter Knoten.



(ii) *rank*(j) $= 0$ oder $p(j)$ ist in einer höheren Ranggruppe als j.
Für die Kosten dieser j bezahlen wir mit (höchstens) $\log^* n$ Euros aus der Gemeinschaftskasse.



Von diesen j's kann es höchstens $\log^* n$ viele geben, weil die Ränge entlang des Weges strikt wachsen (Fakt 1) und es nur $\log^* n$ Ranggruppen gibt.



(iii) $p(j)$ ist in derselben Ranggruppe wie j.
In diesem Fall muss j mit 1 Euro aus seinem Taschengeld bezahlen.

Nun wird abgerechnet:

(A) Die Gesamtkosten aus Fällen (i) und (ii), summiert über alle m **union**s, betragen höchstens $m(2 + \log^* n)$; der Inhalt der Gemeinschaftskasse reicht.

(B) Wie steht es mit Fall (iii) und dem Taschengeld?

Wir betrachten nun **(Trick!)** einen festen Knoten j.

Immer wenn j bezahlen muss, nimmt er an einer Pfadverkürzung teil, d. h., sein neuer Vorgängerknoten $p'(j)$ wird seine aktuelle Wurzel (aber bleibt nicht gleich dem bisherigen Vorgänger $p(j)$). Wegen Fakt 1 hat der **neue Vorgänger** $p'(j)$ einen **höheren Rang** als der **alte**.

Sei G_k die Ranggruppe von j. In dieser Ranggruppe gibt es nicht mehr als $F(k)$ verfügbare Rang-Werte. Daher ändert sich der Vorgänger von j weniger als $F(k)$-mal, bevor ein Knoten aus einer höheren Ranggruppe Vorgänger von j wird (und alle späteren Vorgänger ebenfalls aus höheren Ranggruppen kommen).

Also tritt Situation (iii) für Knoten j weniger als $F(k)$-mal ein. Daher reicht das Taschengeld von Knoten j aus, um für diese Situationen zu bezahlen.

Die Gesamtkosten aus Fall (iii), summiert über alle j's, betragen also nicht mehr als das gesamte Taschengeld, also höchstens $n \log^* n$.

Satz 3.5.2

In der Implementierung der Union-Find-Struktur mit wurzelgerichteten Wäldern und Pfadkompression haben m **find**-Operationen insgesamt Kosten von maximal $(2 + m + n) \log^* n$, benötigen also Rechenzeit $O((m + n) \log^* n)$.
Jede einzelne **union**-Operation benötigt Zeit $O(1)$.

Bemerkung:
(a) Da für real vorkommende n die Zahl $\log^* n$ nicht größer als 5 ist, wird man in der Praxis eine Laufzeit beobachten, die linear in $n + m$ ist.
(b) Implementierungen von Union-Find-Strukturen als wurzelgerichteter Wald mit Pfadkompression verhalten sich in der Praxis sehr effizient.

Arrayimplementierung von Union-Find

Array `r`:$[1..n]$ enthält in `r`$[i]$ den Repräsentanten $r(i)$ der Klasse $K_{r(i)}$, in der i **gegenwärtig** liegt. In unserem Beispiel sähe `r` folgendermaßen aus:

	1	2	3	4	5	6	7	8	9	10	11	12
`r` :	4	4	7	4	4	6	7	6	7	10	11	7

Offensichtlich: **find**(i) in Zeit $O(1)$ ausführbar.

Naiver Ansatz für **union**(s, t): Durchlaufe das Array `r` und ändere dabei alle „s“ in „t“ (oder umgekehrt). Kosten: $\Theta(n)$.
– Dies kann man verbessern.

Arrayimplementierung von Union-Find

Trick 1:
Halte die Elemente jeder Klasse K_s (mit Repräsentant s) in einer linearen Liste L_s.

Dann muss man bei der Umbenennung von „s“ in „t“ nur L_s durchlaufen – die Kosten betragen $\Theta(|S_s|)$. Aus Listen L_s und L_t wird eine neue Liste L'_t.

Trick 2:
Bei der Vereinigung von Klassen sollte man den Repräsentanten der **größeren** Klasse übernehmen und den der **kleineren** ändern, da damit die Zahl der Änderungen kleiner gehalten wird. – Hierzu muss man die **Listenlängen/Klassengrößen** in einem zweiten Array `size`:$[1..n]$ halten.

Im Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
`r` :	4	4	7	4	4	6	7	6	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
`size`:	–	–	–	4	–	2	4	–	–	1	1	–

Nur die Einträge `size`$[s]$ mit $r(s) = s$, also s Repräsentant, sind relevant. Die anderen Einträge (mit „–“ gekennzeichnet) sind unwesentlich.

Listen: $L_4 = (4, 2, 1, 5)$, $L_7 = (7, 3, 12, 9)$, $L_6 = (6, 8)$, $L_{10} = (10)$, $L_{11} = (11)$.

Für eine besonders sparsame Implementierung dieser Listen ist es nützlich, wenn L_s mit dem Repräsentanten s beginnt.

Beispiel:

union(7, 6): Weil `size[6]` $<$ `size[7]`, werden die Einträge `r[`i`]` für i in L_6 auf **7** geändert und $L_6 = \mathbf{(6,8)}$ wird unmittelbar nach dem ersten Element **7** in L_7 eingefügt:

	1	2	3	4	5	6	7	8	9	10	11	12
r :	4	4	7	4	4	7	7	7	7	10	11	7

	1	2	3	4	5	6	7	8	9	10	11	12
size:	–	–	–	4	–	–	6	–	–	1	1	–

Aus $L_7 = (7,3,12,9)$ und $L_6 = (6,8)$ wird die neue Liste $L_7 = (7,\mathbf{6},\mathbf{8},3,12,9)$ gebildet.

Zeitaufwand: Θ(Länge der kleineren Liste) für die Umbenennungen im Array `r` und $O(1)$ für das Ändern des Eintrags `size[t]` sowie das Kombinieren der Listen.

Zur Effizienzverbesserung (um konstanten Faktor) und zur Platzersparnis können wir noch folgenden Trick benutzen. Offensichtlich haben alle Listen zusammen stets die Einträge $1, 2, \ldots, n$. Daher sind keine dynamisch erzeugten Listenelemente nötig, sondern wir können alle Listen kompakt in einem Array speichern:

`next[1..`n`]`: `integer` mit

$$\texttt{next}[i] = \begin{cases} j, & \text{falls } j \text{ Nachfolger von } i \text{ in einer der Listen } L_s, \\ 0, & \text{falls } j \textbf{ letzter} \text{ Eintrag in seiner Liste } L_{r(j)}. \end{cases}$$

Beispiel:

	1	2	3	4	5	6	7	8	9	10	11	12
next:	···	···	12	···	···	8	6	3	0	···	···	9

Darstellung von $L_7 = (7, 6, 8, 3, 12, 9)$.
(Hier sieht man, wieso L_s mit s beginnen muss.)

Implementierung der Operationen im Detail:

Prozedur init(n) (∗ Initialisierung einer Union-Find-Struktur ∗)

(1) Erzeuge `r`, `size`, `next`: Arrays der Länge n für `int`-Einträge
(2) **for** `i` **from** 1 **to** n **do**
(3) `r[i]` ← `i`;
(4) `size[i]` ← 1;
(5) `next[i]` ← 0.

Zeitaufwand: $\Theta(n)$.

Prozedur find(i)

(1) **return** `r[`i`]`.

Zeitaufwand: $O(1)$.

Prozedur union(s,t)

(∗ Ausgangspunkt: s,t sind **verschiedene** Repräsentanten ∗)

(1) **if** size$[s]$ > size$[t]$ **then** vertausche s,t;
(2) (∗ nun: size$[s] \le$ size$[t]$ ∗)
(3) `z` $\leftarrow s$;
(4) **repeat** size$[s]-1$ **times** (∗ Durchlauf L_s ∗)
(5) `r[z]` $\leftarrow t$;
(6) `z` $\leftarrow$ `next[z]`;
(7) (∗ nun: `z` enthält letztes Element von L_s; `next[z]` $=0$ ∗)
(8) `r[z]` $\leftarrow t$;
(9) (∗ L_s nach dem ersten Eintrag in L_t einhängen: ∗)
(10) `next[z]` $\leftarrow$ `next[`t`]`; `next[`t`]` $\leftarrow s$;
(11) `size[`t`]` $\leftarrow$ `size[`t`]` + `size[`s`]`.

Aufwand: O(Länge der kürzeren der Listen L_s, L_t).

Korrektheit:

Betrachte eine Operationenfolge $Op_0 = init(n)$, Op_1, ..., Op_k, wobei die letzten k Operationen **legale union**-Operationen sind.

Man beweist durch Induktion über ℓ, dass nach **init**(n) und nach jedem Op_ℓ, $\ell = 1,\ldots,k$ gilt:

Die Mengen $K_t = \{i \mid \mathtt{r}[i] = t\}$, für diejenigen $t \in \{1,\ldots,n\}$ mit $\mathtt{r}[t] = t$, bilden die Partition von $\{1,\ldots,n\}$, die von den bisher ausgeführten Operationen $Op_0 = init(n)$, Op_1, ..., Op_ℓ erzeugt wird. Dabei ist $\mathtt{size}[t] = |K_t|$ und K_t ist die Menge der i in $\{1,\ldots,n\}$, die von t aus durch Verfolgen von `next`-Zeigern erreichbar ist.

Aus der Behauptung folgt dann auch die Korrektheit von **find**-Operationen, die an jeder Stelle eingeschoben sein können.

Rechenaufwand: **init**(n) benötigt Zeit $\Theta(n)$ und **find**(i) benötigt Zeit $O(1)$.

Behauptung: $n-1$ **union**-Aufrufe (mehr kann es nicht geben!) benötigen Zeit $O(n \log n)$.

Wir behaupten **nicht**, dass jeder einzelne **union**-Aufruf Zeit $O(\log n)$ benötigt (hierfür kann man leicht Gegenbeispiele konstruieren) – aber in der Summe entfällt auf jeden solchen Aufruf ein Anteil von $O(\log n)$:

„Amortisierte Analyse“

Wir nummerieren die **union**-Aufrufe mit $p = 1,2,\ldots,n-1$ durch. Die beiden in Aufruf Nummer p vereinigten Klassen seien $K_{p,1}$ und $K_{p,2}$, wobei $|K_{p,1}| \le |K_{p,2}|$ gelten soll.

Für den p-ten **union**-Aufruf veranschlagen wir Kosten $|K_{p,1}|$.

Dann ist der Gesamt-Zeitaufwand für alle **union**s zusammen

$$O(n) + O\Big(\sum_{1\le p<n} |K_{p,1}|\Big).$$

Wir wollen $\sum_{1\le p<n} |K_{p,1}|$ abschätzen.

Trick: Ermittle den Beitrag zu dieser Summe aus Sicht der einzelnen Elemente i.

Für $1 \le i \le n$, $1 \le p < n$ definiere:

$$a_{ip} := \begin{cases} 1 & \text{falls } i \in K_{p,1}, \\ 0 & \text{sonst.} \end{cases}$$

Dann erhält man durch Umordnen der Summation:

$$\sum_{1 \le p < n} |K_{p,1}| = \sum_{1 \le p < n} \sum_{1 \le i \le n} a_{i,p} = \sum_{1 \le i \le n} \Big(\sum_{1 \le p < n} a_{i,p} \Big).$$

In der letzten inneren Summe $c_i = \sum_{1 \le p < n} a_{i,p}$ wird für jedes $i \in \{1, \ldots, n\}$ gezählt, wie oft es bei **union**-Operationen in der kleineren Menge $K_{p,1}$ enthalten ist (d. h. wie oft insgesamt der Repräsentant seiner Klasse gewechselt hat).

Bei jeder **union**-Operation, bei der i in der kleineren Klasse ist, wird die Größe der Klasse mindestens **verdoppelt**, startend mit der Klasse $\{i\}$. Da Klassen nicht größer als n werden können, gilt $2^{c_i} \le n$, also $c_i \le \log n$, also

$$\sum_{1 \le p < n} |K_{p,1}| = \sum_{1 \le i \le n} c_i \le n \log n.$$

Damit ist die Behauptung bewiesen. □

Algorithmus von Kruskal mit Union-Find

Input: Gewichteter zusammenhängender Graph $G = (V, E, c)$ mit $V = \{1, \ldots, n\}$.

1. Schritt: Sortiere die Kanten $e_1, \ldots, e_m$ nach den Gewichten $c_1 = c(e_1), \ldots, c_m = c(e_m)$ aufsteigend.

Resultat: Sortierte Kantenliste $(v_1, w_1, c_1), \ldots, (v_m, w_m, c_m)$.

2. Schritt: Initialisiere Union-Find-Struktur für $\{1, \ldots, n\}$.

3. Schritt: Für $i = 1, 2, \ldots, m$ tue folgendes:

  s ← **find**(v_i); t ← **find**(w_i);
  if s ≠ t **then begin** R ← R ∪ $\{e_i\}$; **union**(s, t) **end**;
  (∗ Optional: Beende Schleife, wenn |R| $= n - 1$. ∗)

4. Schritt: Die Ausgabe ist R.

Satz 3.5.3

(a) Der Algorithmus von Kruskal in der Implementierung mit Union-Find ist korrekt.

(b) Die Laufzeit des Algorithmus ist $O(m \log m) + O(m) + O(n \log n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit **Arrays** implementiert.

(c) Die Laufzeit des Algorithmus ist $O(m \log m) + O(m \log^* n)$, also $O(m \log n)$, wenn man die Union-Find-Struktur mit einem **wurzelgerichteten Wald** mit **Pfadkompression** implementiert.

Bem.: In (b) und (c) steht der Summand $O(m \log m)$ für die Kosten des Sortierens.

3.6 Huffman-Codes

Gegeben: **Alphabet Σ** und „Wahrscheinlichkeiten“ $p(a) \in [0,1]$ für jeden Buchstaben $a \in \Sigma$. Also: $\sum_{a \in \Sigma} p(a) = 1$.

Beispiel:

a	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09

Herkunft der Wahrscheinlichkeiten:

(1) Buchstabenhäufigkeit in natürlicher Sprache *oder*

(2) empirische relative Häufigkeiten in einem gegebenen Text $w = a_1 \ldots a_n$:

Anteil des Buchstabens a an w ist $p(a) \cdot 100$ %.

Gesucht: ein **„guter“ binärer Präfixcode** für Σ.

Definition Präfixcode:

Jedem $a \in \Sigma$ ist binärer „Code“ $c(a) \in \{0,1\}^+$ zugeordnet, mit Eigenschaft **Präfixfreiheit**:

Für $a, b \in \Sigma$, $a \neq b$ ist $c(a)$ kein Präfix von $c(b)$.

Beispiel:

A	B	C	D	E	F	G	H	I	K
1100	0110	000	111	10	0011	010	0010	0111	1101

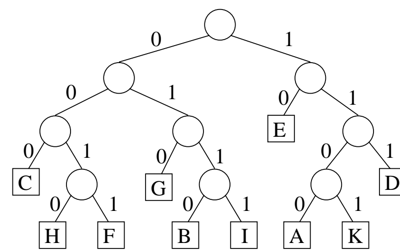
Codierung von Wörtern (Zeichenreihen):

$c(a_1 \ldots a_n) = c(a_1) \cdots c(a_n) \in \{0,1\}^*$.

Zur **Codierung** benutzt man (konzeptuell) direkt die Tabelle.

Beispiel: $c(\texttt{F E I G E}) = 0011\,10\,0111\,010\,10$.

Kompakte Repräsentation des Codes als Binärbaum:



Blätter sind mit Buchstaben markiert; Weg von der Wurzel zum Blatt gibt das Codewort wieder (links: 0, rechts: 1).

Decodierung: Laufe Weg im Baum, vom Codewort gesteuert, bis zum Blatt. Wiederhole mit dem Restwort, bis nichts mehr übrig ist. – Präfixeigenschaft $\Rightarrow$ keine Zwischenräume nötig.

Beispiel: 001111000000010 liefert „FACH“.

1. Idee: Mache alle Codewörter $c(a)$ gleich lang, am besten $\lceil \log_2 |\Sigma| \rceil$ Bits.

$\Rightarrow c(a_1 \ldots a_n)$ hat Länge $\lceil \log_2 |\Sigma| \rceil \cdot n$.

(Beispiele: 52 Groß- und Kleinbuchstaben plus Leerzeichen und Satzzeichen: $\log 64 = 6$ Bits pro Codewort. ASCII-Code: 8 Bits pro Codewort.)

2. Idee: Einsparmöglichkeit: Häufige Buchstaben mit kürzeren Codes codieren als seltenere Buchstaben.

Erster Ansatz zu **Datenkompression** (platzsparendes Speichern, zeitsparendes Übermitteln).

Hier: **„verlustfreie Kompression“** – die Information ist unverändert vorhanden.

Gegensatz: MP3: Informationsverlust bei der Kompression.

	A	B	C	D	E	F	G	H	I	K
$p(a)$	0,15	0,08	0,07	0,10	0,21	0,08	0,07	0,09	0,06	0,09
c_1	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
c_2	1100	0110	000	111	10	0011	010	0010	0111	1101

Wir codieren eine Datei T mit 100000 Buchstaben aus Σ, wobei die relative Häufigkeit von $a \in \Sigma$ durch $p(a)$ gegeben ist.

Mit c_1 (fixe Codewortlänge): 400000 Bits.
Mit c_2 (variable Codewortlänge):
$(4 \cdot (0,15 + 0{,}08 + 0{,}08 + 0{,}09 + 0{,}06 + 0{,}09)) + 3 \cdot (0{,}07 + 0{,}10 + 0{,}07) + 2 \cdot 0{,}21 \cdot 100000 \;= 334000$ Bits.

Bei langen Dateien und wenn die Übertragung teuer oder langsam ist, lohnt es sich, die Buchstaben abzuzählen, die relativen Häufigkeiten $p(a)$ zu bestimmen und einen guten Code mit unterschiedlichen Codewortlängen zu suchen.

Definition

Ein **Codierungsbaum** für das Alphabet Σ ist ein Binärbaum T, in dem

- die Kante in einem inneren Knoten zum linken/rechten Kind (implizit) mit 0 (1) markiert ist;
- jedem Buchstaben $a \in \Sigma$ ein Blatt (externer Knoten) von T exklusiv zugeordnet ist.

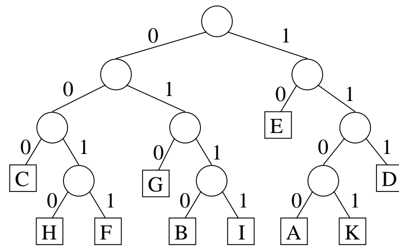
$c_T(a)$ ist die Kanteninschrift auf dem Weg von der Wurzel zum Blatt mit Inschrift a.

Die **Kosten** von T unter p sind definiert als:

$$\boldsymbol{B(T,p)} = \sum_{a\in\Sigma} p(a) \cdot d_T(a),$$

wobei $d_T(a)$ die Tiefe des a-Blatts in T ist.

Beispiel: Wenn T unser Beispielbaum ist und p die Beispielverteilung, dann ist $B(T,p) = 3{,}34$.



Leicht zu sehen: $B(T,p) = |c_T(a_1 \ldots a_n)|/n$, wenn die relative Häufigkeit von a in $w = a_1 \ldots a_n$ durch $p(a)$ gegeben ist,

oder $B(T,p) =$ die **erwartete durchschnittliche Bitzahl** pro Buchstabe, wenn die Buchstabenwahrscheinlichkeiten durch $p(a)$ gegeben ist.

Ziel: Minimierung von $B(T,p)$ zu gegebenem $p : \Sigma \to [0,1]$.

Beobachtung (siehe Lemma 3.6.1 unten):
Wenn es in T einen inneren Knoten v gibt, der einen leeren und einen nichtleeren Unterbaum hat, kann man diesen Knoten aus T entfernen, ohne die Kosten zu erhöhen. –
Weil es nur endlich viele Codierungsbäume T gibt, in denen jeder innere Knoten zwei Kinder hat, gibt es Codierungsbäume, die $B(T,p)$ minimal machen.

Solche optimalen Bäume heißen **Huffman-Bäume**, die entsprechenden Codes heißen **Huffman-Codes**.

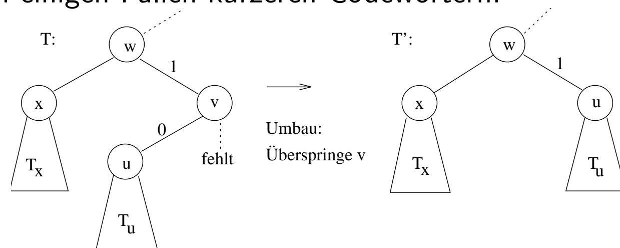
Huffman-Bäume sind i. A. nicht eindeutig; zu einem p gibt es verschiedene Huffman-Bäume.

Aufgabe: Gegeben p, finde einen Huffman-Baum zu p.
Methode: „Greedy“.

Lemma 3.6.1

In einem Huffman-Baum T hat jeder innere Knoten zwei Kinder.

Beweis: Sei T Huffman-Baum, v innerer Knoten mit einem leeren Unterbaum. Dann kann man v aus T herausschneiden. Resultat: T' für Σmit denselben markierten Blättern wie T, aber in einigen Fällen kürzeren Codewörtern:



Daher: $B(T,p) \geq B(T',p)$. □

Lemma 3.6.2

Es seien a, a' zwei Buchstaben mit $p(a)$, $p(a') \leq p(b)$ für alle $b \in \Sigma - \{a, a'\}$.

(a, a' sind die beiden **„seltensten“** Buchstaben.)

Dann gibt es einen Huffman-Baum, in dem die a- und a'-Blätter Kinder desselben inneren Knotens sind.

Beweis:

Starte mit beliebigem Huffman-Baum T.

O.B.d.A. (Le. 3.6.1): Alle inneren Knoten haben zwei Kinder.

a und a' sitzen in Blättern von T, Tiefen $d(a)$ und $d(a')$.

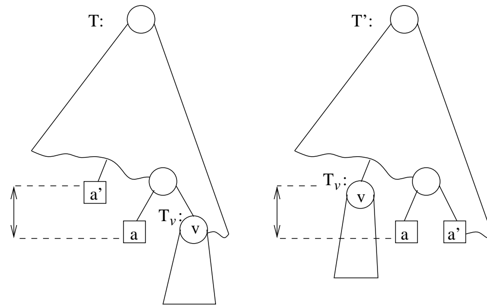
O.B.d.A.: $(*)$ $\quad d_T(a) \geq d_T(a')$ (sonst umbenennen).

Der a-Knoten hat einen Bruderknoten v (innerer Knoten oder Blatt).

1. Fall: Der a'-Knoten liegt im Unterbaum T_v mit Wurzel v. Wegen $(*)$ muss er gleich v sein, und a-Knoten und a'-Knoten sind Geschwisterknoten in T.

2. Fall: a'-Knoten nicht in T_v.
Klar: auch a-Knoten nicht in T_v.
Weil T_v mindestens ein Blatt hat und $p(b) \geq p(a')$ für alle $b \in \Sigma - \{a, a'\}$ gilt, haben wir

$$\sum_{b \text{ in } T_v} p(b) \geq p(a').$$



Wir vertauschen Blatt a' und T_v.

Liefert Codierungsbaum T', in dem a und a' Geschwister sind, und für den gilt:

$$B(T',p)-B(T,p) = (d_T(a)-d_T(a'))(p(a')-\sum_{b \text{ in } T_v} p(b)) \leq 0.$$

Das heißt: auch T' ist ein Huffman-Baum für p. □

Damit ist der erste Schritt zur Realisierung eines Greedy-Ansatzes getan!

Man beginnt den Algorithmus mit

„Mache die beiden seltensten Buchstaben zu Geschwistern“.

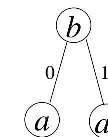
Dann ist man sicher, dass dies stets zu einer optimalen Lösung ausgebaut werden kann.

Diese optimale Lösung findet man **rekursiv** (konzeptuell) bzw. dann in der Realisierung **iterativ**.

Ansatz: Wir bauen einen Huffman-Baum **„bottom-up“** auf, per Rekursion.

Wenn $|\Sigma| = 1$: fertig, man benötigt nur einen Knoten, der auch Blatt ist.
Optimalität: Klar.

Sonst werden zwei „seltenste“ Buchstaben a, a' zu benachbarten Blättern gemacht.



Die Wurzel des so erzeugten Mini-Baums wird als ein „Kunstbuchstabe“ b aufgefasst mit $p(b) = p(a) + p(a')$.

Neues Alphabet: $\Sigma' := (\Sigma - \{a, a'\}) \cup \{b\}$; neue Verteilung:

$$p'(d) := \begin{cases} p(d) & \text{falls } d \neq b \\ p(a) + p(a') & \text{falls } d = b \end{cases}$$

Nun bauen wir durch **rekursive** Verwendung des Algorithmus einen Huffman-Baum T' für Σ' und p'.

In T' fügen wir an der Stelle des b-Knotens den a, a'-Baum ein.

Ergebnis: Ein **Codierungsbaum** T für Σ und p,
mit $B(T, p) = B(T', p) + (p(a) + p(a'))$. (Checken!)

Lemma 3.6.3: T ist **Huffman**-Baum für (Σ, p).

Beweis: Durch Induktion über die rekursiven Aufrufe.

Nach Lemma 3.6.1 gibt es einen **Huffman**-Baum T_1 für (Σ, p), in dem a- und a'-Knoten Geschwister sind. Daher: $B(T_1, p) \leq B(T, p)$.

Aus T_1 bilden wir T_1' durch Ersetzen des a, a'-Teilbaums durch den Kunstknoten b. Dann ist T_1' **Codierungs**baum für (Σ', p').

Nach I.V. ist T' Huffman-Baum, also $B(T', p') \leq B(T_1', p')$. Daher:

$$B(T, p) = B(T', p) + p(a) + p(a') \leq B(T_1', p') + p(a) + p(a') = B(T_1, p).$$

Weil T_1 Huffman-Baum für (Σ, p) ist: $B(T, p) = B(T_1, p)$, und auch T ist ein Huffman-Baum. □

Man könnte nach dem angegebenen Muster eine rekursive Prozedur programmieren.

Mit einer **Priority-Queue** findet man stets effizient die beiden Knoten mit geringstem Gewicht!

Anfangs in **PQ**: Buchstaben $a \in \Sigma$ mit Gewichten $p(a)$ als Schlüssel.
Das Ermitteln und Entfernen der beiden leichtesten Buchstaben a, a' erfolgt durch zwei Aufrufe **PQ.extractMin**;
das Einfügen des neuen Kunstbuchstabens b durch **PQ.insert**$(p(a) + p(a'))$.

Die Implementierung wird **effizienter**, wenn man **iterativ** vorgeht.

Eine spezielle Repräsentation des Baums (nur **Vorgänger**zeiger werden durch Indizes dargestellt) ermöglicht es, ganz ohne Zeiger auszukommen.

Als Datenstruktur genügt ein Array `A[1..2m-1]`, $m = |\Sigma|$.

Dabei repräsentieren die Positionen $1, \ldots, m$ die Buchstaben $a_1, \ldots, a_m$ in Σ, die Positionen $m + 1, \ldots, 2m - 1$ die $m - 1$ „Kunstbuchstaben“ und inneren Knoten des zu bauenden Baums.

p[1..$2m-1$] ist ein Array, das in den Positionen $1,\ldots,m$ die Gewichte $p(a_i)$ der Buchstaben $a_1,\ldots,a_m$ enthält.

Für den Algorithmus tut man so, als ob $1,\ldots,m$ die Buchstaben und $m+1,\ldots,2m-1$ die „Kunstbuchstaben“ wären.

Positionen p[$m+1$..$2m-1$]: Gewichte der $m-1$ „Kunstbuchstaben“.

Das Array predecessor[1..$2m-1$] speichert die Vorgänger der Knoten im Baum (Knoten $2m-1$ ist die Wurzel und hat keinen Vorgänger).

In einem Bitarray mark[1..$2m-1$] kann man mitführen, ob ein Knoten linkes („0“) oder rechtes („1“) Kind ist.

PQ: Priority Queue, Einträge: $a \in \{1,\ldots,2m-1\}$;
Schlüssel: die Gewichte p[a].

Algorithmus Huffman(p[1..m])

Eingabe: Gewichtsvektor p[1..m]
Ausgabe: Implizite Darstellung eines Huffman-Baums

```
(1)   for a from 1 to m do
(2)     PQ.insert(a);  (* Buchstabe a hat Priorität p[a] *)
(3)   b ← m + 1 (* letzter echter Buchstabe *)
(4)   repeat m − 1 times
(5)     a ← PQ.extractMin;
(6)     aa ← PQ.extractMin;
(7)     b ← b + 1 (* nächster Kunstbuchstabe *)
(8)     predecessor[a] ← b;
(9)     mark[a] ← 0;
(10)    predecessor[aa] ← b;
(11)    mark[aa] ← 1;
(12)    p[b] ← p[a] + p[aa];
(13)    PQ.insert(b);
(14)  Ausgabe: predecessor[1..2m − 1] und mark[1..2m − 1].
```


Aus predecessor[1..$2m-1$] und mark[1..$2m-1$] baut man den Huffman-Baum wie folgt:

Allokiere ein Array node[1..m] mit Blattknoten-Objekten und ein Array inner[$m+1$..$2m-1$] mit Objekten für innere Knoten.

```
(1)   for i from 1 to m do
(2)     leaf[i].letter ← Buchstabe a_i.
(3)     j ← predecessor[i]
(4)     if mark[i] = 0
(5)       then inner[j].left ← leaf[i]
(6)       else inner[j].right ← leaf[i]
(7)   for i from m + 1 to 2m − 2 do
(8)     j ← predecessor[i]
(9)     if mark[i] = 0
(10)      then inner[j].left ← inner[i]
(11)      else inner[j].right ← inner[i]
(12)  return inner[2m − 1] (* Wurzelknoten *)
```


Satz 3.6.5

Der Algorithmus **Huffman** ist korrekt und hat Laufzeit $O(m \log m)$, wenn m die Anzahl der Buchstaben des Alphabets Σ bezeichnet.

Beweis: Laufzeit: Aufbau der Priority Queue dauert $O(n \log n)$; mit Tricks könnte man auch mit Zeit $O(n)$ auskommen.

Die Schleife wird $(m-1)$-mal durchlaufen. In jedem Durchlauf gibt es maximal 3 PQ-Operationen, mit Kosten $O(\log n)$.

Korrektheit: Folgt aus der Korrektheit der rekursiven Version.

Kann man $B(T)$ für einen optimalen Baum einfach aus den Häufigkeiten $p(\sigma_1), \ldots, p(\sigma_n)$ **berechnen**, ohne T zu konstruieren?

Antwort: Ja, zumindest näherungsweise.

Definition Sind $p_1, \ldots, p_n \geq 0$ mit $\sum_{i=1}^n p_i = 1$, setzt man

$$H(p_1, \ldots, p_n) := \sum_{i=1}^n p_i \cdot \log(1/p_i).$$

$H(p_1, \ldots, p_n)$ heißt die **Entropie** der Verteilung $p_1, \ldots, p_n$. (Wenn $p_i = 0$ ist, setzt man $p_i \log(1/p_i) = 0$, was vernünftig ist, weil $\lim_{x \searrow 0} x \cdot \log(1/x) = 0$.)

Interessant: Zusammenhang zwischen **Entropie** $H(p_1, \ldots, p_n)$ und der **erwarteten Bitlänge** eines Textes, in dem $n = |\Sigma|$ Buchstaben mit Wahrscheinlichkeiten $p_1, \ldots, p_n$ auftreten.

Klassisches Resultat:

Lemma 3.6.6 (Lemma von Gibb)

Sind $q_1, \ldots, q_n > 0$ mit $\sum_{i=1}^n q_i \leq 1 = \sum_{i=1}^n p_i$, so gilt

$$\sum_{i=1}^n p_i \log(1/q_i) \geq \sum_{i=1}^n p_i \log(1/p_i) = H(p_1, \ldots, p_n).$$

Beweis:

Weil $\log_2 x = \ln x / \ln 2$ ist, darf man mit dem natürlichen Logarithmus rechnen.

$$\begin{aligned} &\sum_{i=1}^n p_i \ln\left(\frac{1}{p_i}\right) - \sum_{i=1}^n p_i \ln\left(\frac{1}{q_i}\right) \\ &= \sum_{i=1}^n p_i \ln\left(\frac{q_i}{p_i}\right) \leq \sum_{i=1}^n p_i \left(\frac{q_i}{p_i} - 1\right) \\ &= \sum_{i=1}^n (q_i - p_i) = \sum_{i=1}^n q_i - \sum_{i=1}^n p_i \leq 0. \end{aligned}$$

Es wird benutzt, dass $\ln(x) \leq x - 1$ gilt, für alle $x \in \mathbb{R}$. (Offenbar gilt $p_i \ln(q_i/p_i) \leq q_i - p_i$ auch für $p_i = 0$.)

Satz 3.6.7 (Kraft'sche Ungleichung)

Es seien $n \geq 1$, $l_1, \ldots, l_n \in \mathbb{N}$. Dann existiert ein Präfixcode mit Codewortlängen $l_1, \ldots, l_n$ genau dann wenn

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1.$$

Auch hier ist der Beweis sehr einfach: Statt „Existenz eines Präfixcodes" kann man auch „Existenz eines Binärbaums mit Blättern auf Tiefe $l_1, \ldots, l_n$" sagen, nach den Bemerkungen am Anfang des Abschnitts.

„$\Rightarrow$": Es seien $x_1, \ldots, x_n \in \{0,1\}^*$ die Codewörter eines präfixfreien Codes mit $|x_i| = l_i$, $1 \leq i \leq n$. Sei $L := \max\{l_i \mid 1 \leq i \leq n\}$. Klar:

$$B_i := \{x_i s \mid s \in \{0,1\}^{L-l_i}\}$$

hat 2^{L-l_i} Elemente und alle B_i sind disjunkt (wegen der Präfixfreiheit). Also:

$$2^L \geq \sum_{i=1}^{n} |B_i| = \sum_{i=1}^{n} 2^{L-l_i},$$

d.h.

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1.$$

„$\Leftarrow$": Nun seien $l_1, \ldots, l_n \geq 1$ gegeben, mit $\sum_{i=1}^{n} 2^{-l_i} \leq 1$. Wir benutzen Induktion über $n \geq 1$, um die behauptete Existenz eines passenden präfixfreien Codes zu zeigen.

$n = 1$: Wähle ein beliebiges Codewort x_1 aus $\{0,1\}^{l_1}$. (Achtung: Wenn $l_1 = 0$, ist $x_1 = \epsilon$, das leere Wort. Dies entspricht einem Codierungsbaum, der nur aus der Wurzel besteht.)

Nun sei $n \geq 2$. Wir ordnen (o.B.d.A.) die $l_1, \ldots, l_n$ so an, dass $l_1 \geq l_2 \geq \cdots \geq l_n$ gilt. Wenn $l_1 > l_2$, hat man $\sum_{i=2}^{n} 2^{l_n - l_i} < 2^{l_n}$, und die Summe ist durch $2^{l_n - l_2}$ teilbar, also gilt sogar $\sum_{i=2}^{n} 2^{l_n - l_i} \leq 2^{l_n} - 2^{l_n - l_2}$. D.h.: $2^{-(l_2-1)} + \sum_{i=3}^{n} 2^{-l_i} \leq 1$. Setze $l'_1 := l_2 - 1$ und finde (nach Induktionsvoraussetzung) einen Präfixcode $\{x'_1, x_3, x_4, \ldots, x_n\}$ für $l'_1, l_3, \ldots, l_n$. Nun bilde $x_2 := x'_1 1$ und $x_1 := x'_1 0 \ldots 0$ (mit $l_1 - l_2 + 1$ angehängten Nullen). Es ist leicht zu sehen, dass auch $\{x_1, \ldots, x_n\}$ präfixfrei ist.

Beispiel:

$(l_1, \ldots, l_6) = (1,4,5,3,6,3)$; sortiert: $(6,5,4,3,3,1)$.

Dies führt zu rekursiven Aufrufen für:

1) $(4,4,3,3,1)$

2) $(3,3,3,1)$

3) $(2,3,1)$, sortiert: $(3,2,1)$

4) $(1,1)$

5) (1).

Die Präfixcodes für diese Aufrufe:

5) $\{\epsilon\}$

4) $\{0,1\}$

3) $\{000,01,1\}$, also $\{01,00,1\}$

2) $\{010,001,000,1\}$

1) $\{0100,0101,011,000,1\}$

Gesamtlösung: $\{010000,01001,0101,011,000,1\}$

Satz 3.6.8 (Huffman versus Entropie)

Sind $p_1, \ldots, p_n \geq 0$ mit $\sum_{i=1}^{n} p_i = 1$ gegeben, so gilt für einen Huffman-Baum T zu Buchstaben $a_1, \ldots, a_n$ mit den Wahrscheinlichkeiten $p_1, \ldots, p_n$:

$$H(p_1, \ldots, p_n) \leq B(T) \leq H(p_1, \ldots, p_n) + 1.$$

(Informal: Die erwartete Zahl von Bits, die man braucht, um einen Text $t_1 \ldots t_N$ über $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ zu codieren, liegt zwischen $N \cdot H(p_1, \ldots, p_n)$ und $N \cdot (H(p_1, \ldots, p_n) + 1)$.)

Beweis: 1. Ungleichung: Es seien $l_1, \ldots, l_n$ die Tiefen der Blätter in T zu den Buchstaben $\sigma_1, \ldots, \sigma_n$. Dann gilt $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ nach Satz 7.4.10. Damit können wir Satz 7.4.9 mit $q_i = 2^{-l_i}$ anwenden und erhalten

$$B(T) = \sum_{i=1}^{n} p_i \cdot l_i = \sum_{i=1}^{n} p_i \cdot \log(1/2^{-l_i}) \geq H(p_1, \ldots, p_n).$$

Für die 2. Ungleichung genügt es zu zeigen, dass ein Codierungsbaum T' für $\sigma_1, \ldots, \sigma_n$ existiert, in dem $B(T') \le H(p_1, \ldots, p_n) + 1$ gilt. (Unser Huffman-Baum T erfüllt ja $B(T) \le B(T')$.) Wir setzen $l_i := \lceil \log(1/p_i) \rceil$, für $1 \le i \le n$, und beobachten:

$$\begin{aligned}\sum_{i=1}^{n} 2^{-l_i} &= \sum_{i=1}^{n} 2^{-\lceil \log(1/p_i) \rceil} \le \sum_{i=1}^{n} 2^{-\log(1/p_i)} \\ &= \sum_{i=1}^{n} p_i = 1.\end{aligned}$$

Nach Satz 3.6.9 existiert also ein Präfixcode mit Codewortlängen $(l_1, \ldots, l_n)$; im entsprechenden Codierungsbaum T' ordnen wir dem Blatt auf Tiefe l_i den Buchstaben σ_i zu. Dann ist

$$\begin{aligned}B(T') &= \sum_{i=1}^{n} p_i \cdot l_i \le \sum_{i=1}^{n} p_i \cdot (\log(1/p_i) + 1) \\ &= H(p_1, \ldots, p_n) + \sum_{i=1}^{n} p_i \\ &= H(p_1, \ldots, p_n) + 1.\end{aligned}$$

Bemerkung: Es gibt bessere Kodierungsverfahren als Huffman (vermeiden den Verlust von bis zu einem Bit pro Buchstabe: z.B. „arithmetische Kodierung“), aber Huffman-Kodierung ist ein guter Anfang . . .