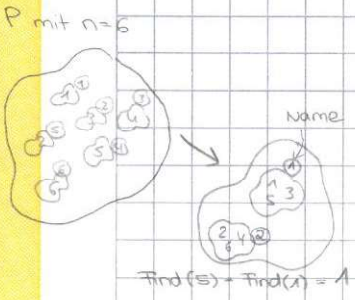
I) Datenstrukturen für Mengen

1. Verwaltung disjunkter Mengen

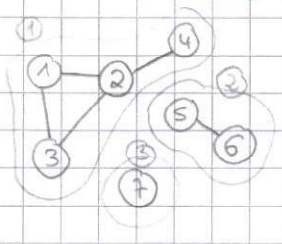
1.1. Das UNION-Find Problem

Problem Verwalte eine Partition (Menge von disj. Teilmengen) von Blöcken (Teilmengen) der Menge $U = \{1, \dots, n\}$ unter den Operationen

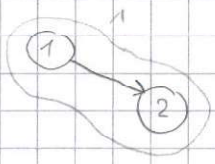


- Init(n): erzeugt der Partition $\{\{1\}, \{2\}, \dots, \{n\}\}$ Block $\{i\}$ erhält den Namen i
- Find(x): $x \in U$ liefert den Namen des Blocks, der x enthält
- Union(A,B,C): Vereinige die Blöcke mit Namen A und B zu einem Block C → A und B werden zerstört

1. Anwendung Zusammenhang ungerichteter Graphen $G = (V, E)$, $V = \{1, \dots, n\}$



UNION - FIND



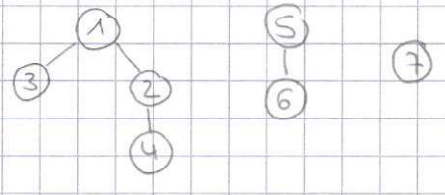

```
1. Init(n)
2. ∀(v,w) ∈ E do
3.       A ← Find(v);
4.       B ← Find(w);   ← besser: if A≠B then union(A,B,A)
5.       UNION(A,B,A)
6. od;
```

⇒ Funktion: SAME_COMPONENT(v,w)
return Find(v) = Find(w);

2. Spaning Tree → aufgespannter Baum (Wald)

berechne Teilmenge $E' \subseteq E$
$G'(V, E')$ ist azyklisch und hat die selben ZHK wie G

im Beispiel



Aufgabe: Berechne E' mit Union-Find

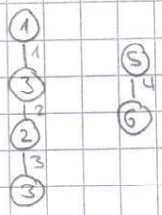
$E' \leftarrow E' \cup \{v, w\}$

3. Minimum Spanning Tree (MST)

Zusatz: Kantenkosten $\forall v,w \in E : cost(v,w)$

Berechne Spanning Tree mit minimalen Gesamtkosten
d.h. eine Menge E' (wie oben) mit $\sum_{e \in E'} cost(e)$ minimal

im Beispiel:



Kosten: 13

Lösungen für das Union-Find-Problem

1) Feld von Namen: array[1...n], das für jedes $i \in \{1,...,n\}$ den Namen des Blocks (Name[i]) angibt, das i enthält

Initialisierung: for i=1 to n do
Name[i] = i;
od
} $O(n)$

Find(x): return Name[x] } $O(1)$

Union(A,B,C): forall $x \in \{1,...,n\}$ do
if Name[x] $\in$ {A,B} then
Name[x] = C
} $O(n)$

2) Relabel the smaller half

zusätzliche Daten-(strukturen):
- ein Feld size: array[1,...,n] d.h. size[A] = |A|
- für jeden Block A eine lineare Liste L[A] realisiert durch ein Feld von Listenköpfen
L[A] enthält alle Elemente von A

Union(A,B,C):
- Gehe kürzere der beiden Listen A und B durch, etwa B, und ändere alle B's zu A's
- Konkateniere L[A] und L[B] (gehe in $O(1)$, falls Zeiger auf letztes Element
hier: neuer Block heißt nun A statt C
Ausweg: wir unterscheiden interne und externe Namen und benutzen 2 Felder HAPOUT und HAPIN zur Übersetzung interner Namen in externe Namen und umgekehrt.
HAPOUT: intern → extern
HAPIN: extern → intern

Initialisierung:
```
Name[i] = i ;
MAPIN[i] = i ;
MAPOUT[i] = i ;   } ∀ i ∈ {1, ..., n}      O(n)
size[i] = 1
L[i] = []
```

Find(x): return MAPOUT[Name[x]]; } O(1)

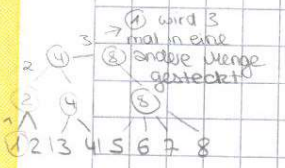
Union(A,B,C): if size(A) < size(B) then
```
        x ← MAPIN[A]; y ← MAPIN[B];
    else
        x ← MAPIN[B]; y ← MAPIN[A];
    fi
    // ändere X's in Y's um
    forall z ∈ L[x] do
        Name[z] ← y ;
    od
    L[y] ← L[x] conc L[y]    // hänge L[x] an L[y]

    MAPOUT[y] ← C ;
    MAPIN[C] ← y ;
    size[C] ← size[A] + size[B] ;
```

Laufzeit: $O(\min(|A|, |B|))$

Satz: Die Gesamtkosten einer beliebigen Folge von n-1 Unions und m Finds sind $O(m + n \log n)$

Beweis: Find hat Laufzeit $O(1)$ → m Finds haben Lz $O(m)$



Eine Operation Union(A,B,C) hat die Kosten c · Anzahl der Elemente $x \in \{1,\dots,n\}$ für die Name[x] geändert wird (c ist Konstante)

Bemerkung:

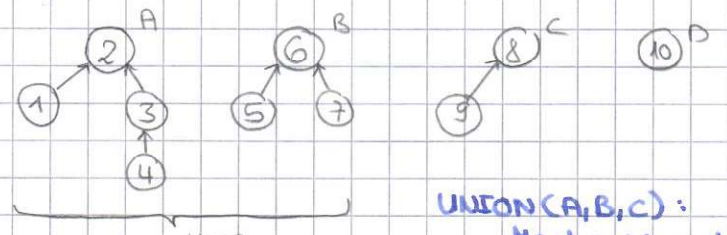
1. Falls für ein $x \in A$ Name[x] geändert wird, dann gehört x danach zu einem Block, der Größe $\geq 2 \cdot |A|$

2. Am Anfang gehört jedes x zu einem Block der Größe 1, am Schluss zu einem Block der Größe $\leq n$
   ⇒ Für jedes $x \in \{1,\dots,n\}$ gilt:
   Name[x] wird höchstens $\log n$ - mal geändert
   ⇒ Insgesamt passierten höchstens $n \log n$ Namensänderungen. Die Kosten aller UNIONs = $c \cdot n \log n = O(n \log n)$

Lösungen 1+2 begünstigen FINDs, die nächsten Lösungen werden Unions begünstigen.

3) Jeder Block wird durch einen Baum dargestellt, die Kinder repräsentieren die Elemente des Blocks, in der Wurzel steht der Name

Beispiel:



Find(x):
starte im Knoten des x darstellt und laufe bis zur Wurzel. Sie enthält den Namen des Blockes der x enthält
Kosten: $O(\text{Tiefe}(x) \text{ im Baum})$

Union(A,B,C):
Mache Wurzel von B zu Kind der Wurzel von A (oder umgekehrt)
Schreibe C in Wurzel (neuer Name)
⇒ Kosten: $O(1)$

Realisierung der Bäume durch Felder:

$$\text{Vater}[i] = \begin{cases} \text{Vater von } i \text{ im Baum} \\ 0, \text{ falls } i \text{ Wurzel} \end{cases}$$

Name[i] = Name des Blocks mit Wurzel i
(hat nur Bedeutung wenn i Wurzel ist)

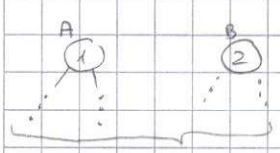
Wurzel[A] = Wurzel des Blocks mit Namen A

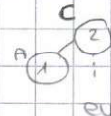
Initialisierung:

$$\left.\begin{array}{l} \text{Vater}[i] = 0 \\ \text{Name}[i] = i \\ \text{Wurzel}[i] = i \end{array}\right\} \forall i \in \{1, \dots, n\}$$

```
Find(x): while Vater[x] ≠ 0 do
             x ← Vater[x]
         od
         return Name[x]

Union(A,B,C): ω1 ← Wurzel[A];
              ω2 ← Wurzel[B];
              Vater(ω1) ← ω2;   // hier: A wird an B gehängt
              Wurzel[C] ← ω2;
              Name[ω2] ← C;
```





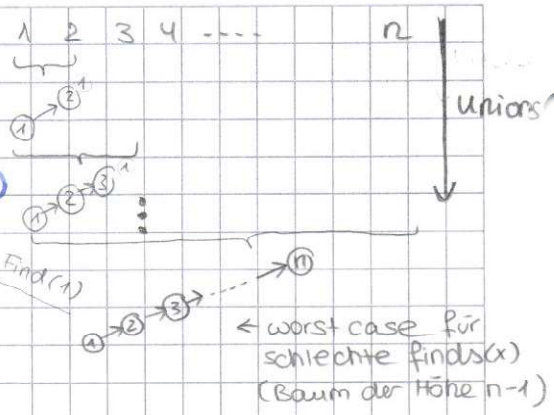
man muss bis zur Wurzel hochlaufen, da an anderen Stellen eventuell der falsche alte Name steht und nur die Wurzel den aktuellen Namen hat

Laufzeitanalyse:

Union: $O(1)$

Find(x): $O(\text{Höhe des Baumes})$

worst case: $O(n)$



4) Verbesserung zu 3):

Vermeide große Tiefen dadurch, dass immer Wurzel des kleineren Baumes zum Kind der Wurzel des größeren Baumes gemacht wird (weighted union rule)

zusätzliches Feld:

size[i] = # der Knoten im Baum mit Wurzel i

Initialisierung

```
size[i] = 1;
Rest analog zu 3)
```

```
UNION(A,B,C):
    w1 ← Wurzel[A];
    w2 ← Wurzel[B];

    if size[w1] < size[w2] then
        Vater[w1] ← w2;
        size[w2] ← size[w1] + size[w2];
        Wurzel[C] ← w2;
        Name[w2] ← C;
    else
        analog mit w1 und w2 vertauscht
```

Find(x): analog zu 3)

Satz Bei Union-Find mit weighted union rule hat eine Folge von n-1 unions und m Find's die Gesamtkosten

$$O(n + m \log n)$$

↑ amortisierte Laufzeit ↔ worst case

Lemma: Für alle Knoten x gilt:

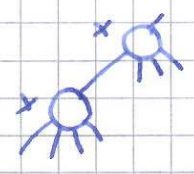
$$\text{gewicht}(x) \geq 2^{\text{Höhe}(x)}$$

Beweis
Induktion über Höhe(x)

Höhe(x) = 0 ⇒ x ist Blatt
⇒ Gewicht(x) = 1 = $2^{\text{Höhe}(x)}$

Sei nun Höhe(x) > 0 : Sei y ein Kind von x mit Höhe(y) = höhe(x) − 1, (siehe Def von Höhe(x))

I.A. $\Rightarrow$ gewicht(y) $\geq 2^{(\text{höhe}(y)}$



Betrachte die Union-Operation die y zum Kind von x gemacht hat.
Seien $\overline{\text{gewicht}(x)}$, $\overline{\text{gewicht}(y)}$ die Gewichte vor dieser Operation

Dann gilt

1. $\text{gewicht}(y) = \overline{\text{gewicht}(y)}$ nur für Wurzeln kann sich Gewicht ändern
2. $\overline{\text{gewicht}(x)} \geq \overline{\text{gewicht}(y)}$ weighted union
3. nach der Union Operation

$$\text{gewicht}(x) \geq \overline{\text{gewicht}(x)} + \overline{\text{gewicht}(y)}$$
$$\geq 2 \cdot \overline{\text{gewicht}(y)} \quad \text{(wegen 2)}$$
$$= 2 \cdot \text{gewicht}(y) \quad \text{(wegen 1)}$$
$$\text{I.A.} \quad \geq 2 \cdot 2^{\text{höhe}(y)} = 2^{\text{höhe}(y)+1} \quad \text{(weil von y)}$$
$$= 2^{\text{höhe}(x)}$$

Da ja das Gewicht(x) $\leq n \ \forall x$ (insgesamt: n Knoten), folgt

$$n \geq \text{gewicht}(x) \overset{\text{Lemma}}{\geq} 2^{\text{höhe}(x)}$$

$$\Rightarrow 2^{\text{höhe}(x)} \leq n \mid \log$$

$$\Rightarrow \text{höhe}(x) \leq \log n$$

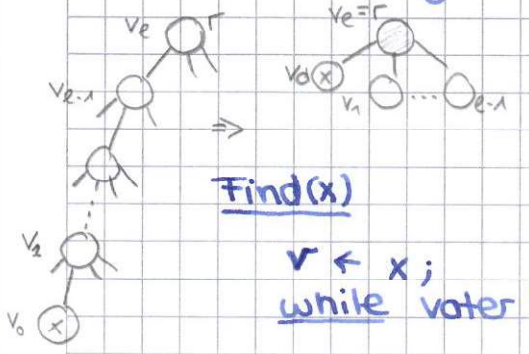
$\Rightarrow$ Kosten einer bel. FIND-Operation sind $O(\log n)$ (worst case)!
$\rightarrow$ Satz ■ m Finds Kosten $O(m \log n)$

5) Verbesserung 4)

weighted union + Pfad Komprimierung (path. compression)

→ Find(x) durchläuft einen Pfad $v_0, \dots, v_e$ von Knoten wobei $v_0 = x$, v_e = Wurzel r

Idee: Hänge $v_0, \dots, v_{e-2}$ direkt an die Wurzel r



Find(x)

```
r ← x;                                   // r ist Wurzel
while vater[r] ≠ 0 do
        r ← vater[r];
od
                    (≠ r)
while vater[x] ≠ 0 do
        p ← vater[x];
        vater[x] ← r;
        x ← p;
od
```

Beobachtung:

Pfadkompremierung:

- verteuert aktuelle Find aber nur um konstanten Faktor → $O(\log n)$
- macht spätere Finds erheblich billiger

Satz (Tarjan)

Bei Union-Find mit weighted Union und path compression hat eine beliebige Folge von $n-1$ Unions und $m \geq n$ Finds die Gesamtkosten $O(m \cdot \alpha(m,n))$

Eine Variante der Ackermann-Funktion und ihre Inverse

$A : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

$A(i,0) = 0$ für $i \geq 0$
$A(0,x) = 2x$ für $x \geq 0$
$A(i,1) = 2$ für $i \geq 0$

} Verankerung

Für $i > 0, x > 1$: $A(i,x) = A(i-1, A(i, x-1))$