# Algorithmen und Datenstrukturen

Tanya Braun

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Tanya Braun (Übungen) sowie viele Tutoren



## Danksagung

Die nachfolgenden Präsentationen wurden mit ausdrücklicher Erlaubnis des Autors übernommen und danach abgewandelt aus:

 "Effiziente Algorithmen und Datenstrukturen" (Kapitel 6: Verschiedenes) gehalten von Christian Scheideler an der TUM <a href="http://www14.in.tum.de/lehre/2008WS/ea/index.html.de">http://www14.in.tum.de/lehre/2008WS/ea/index.html.de</a>

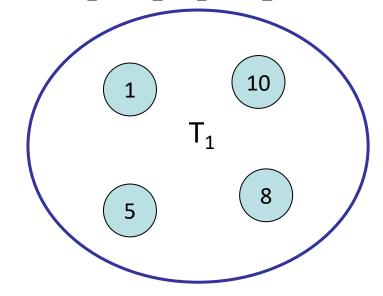
Der Inhalt zum Beweis zur amortisierten Analyse der Union-Find Datenstruktur basiert auf

 Hopcroft, J.E. and Ullman, J.D.; Set Merging Algorithms, SIAM Journal of Computing 2(4), S. 294-303, 1973.

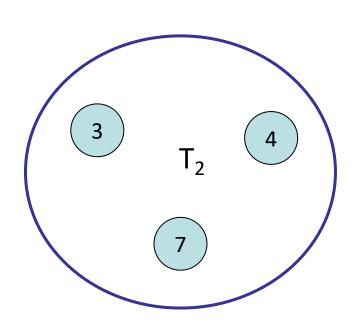


# Partitionen einer Menge

- Disjunkte Teilmengen, die zusammen die Ursprungsmenge ergeben
  - $-T = \{1, 5, 8, 10, 3, 4, 7\}$
  - Partitionen:  $T_1$  und  $T_2$
  - $-T = T_1 \cup T_2, T_1 \cap T_2 = \emptyset$



Identifizierung?





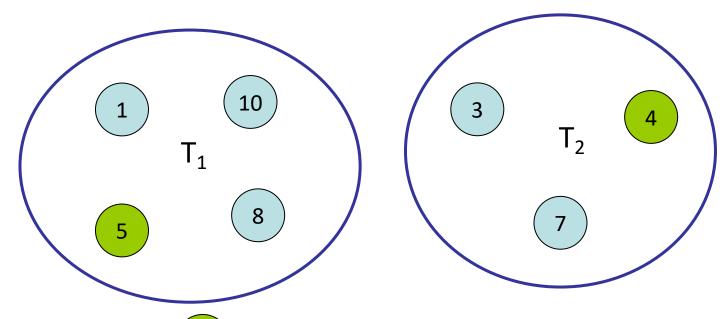
# Identifizierung einer Partition

Element aus Partition als Repräsentant

$$-T = \{1, 5, 8, 10, 3, 4, 7\}$$

-  $T_1$ : Repräsentant 5

- T<sub>2</sub>: Repräsentant 4







### Datenstruktur für Disjunkte Mengen

#### Wozu brauchen wir so eine Datenstruktur?

- Effiziente Implementierung von Graphalgorithmen
  - Ermittlung minimaler Spannbäume (Kruskal)
  - Ermittlung starker Zusammenhangskomponenten
     (Beides kommt in dem Vorlesungsteil zu Graphen vor)

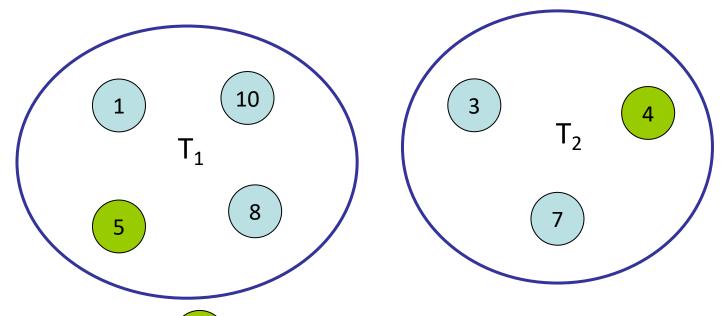
#### Was muss die Datenstruktur können?

- Testen, ob zwei Elemente zu einer Menge gehören
- Zwei Mengen vereinigen



### Test: Zugehörigkeit zur selben Partition?

- Gegeben: Partitionen mit Repräsentant, zwei Elemente
  - Partitionen:  $T_1, T_2$ , Elemente: 1 und 10, 1 und 3
- Test über Gleichheit der Repräsentanten
  - Anforderung: schnell auf Repräsentant kommen

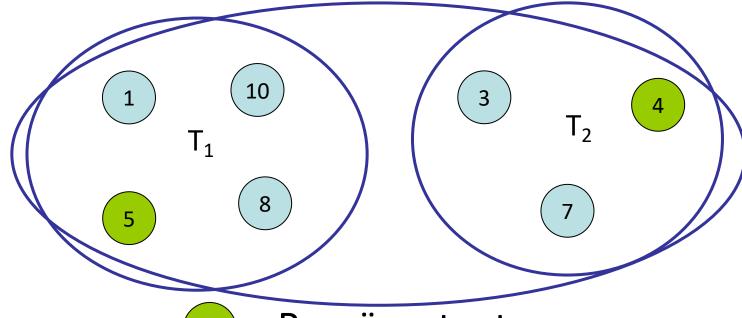




: Repräsentant

### Vereinigung zweier Partitionen

- Gegeben: Partitionen mit Repräsentant
  - Partitionen:  $T_1$ ,  $T_2$
- Elemente vereinigen, einen Repräsentanten behalten
  - Anforderung: schnell zwei Mengen verschmelzen





: Repräsentant

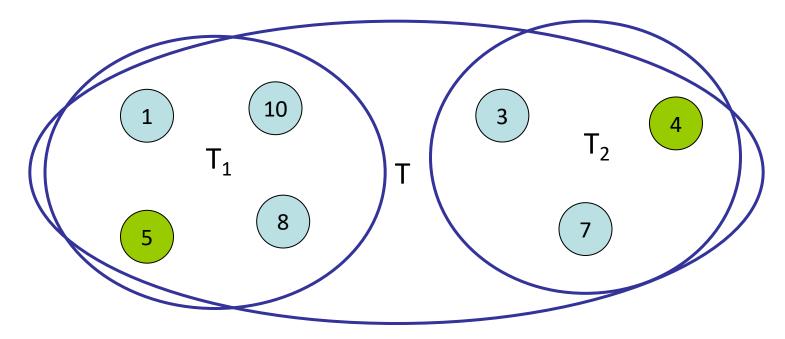
Gegeben: Menge von n Elementen.

#### Operationen:

- MakeSet(x): erzeugt für x eine (Teil)menge T mit x als Repräsentant (Initialisierung)
- Union(T<sub>1</sub>,T<sub>2</sub>): vereinigt Elemente in T<sub>1</sub> und T<sub>2</sub> zu
   T=T<sub>1</sub> U T<sub>2</sub>
- Find(x): gibt (eindeutigen) Repräsentanten der Teilmenge aus, zu der x gehört
  - Nimmt an, dass es einen direkten Zugriff auf x gibt



# Union $(T_1,T_2)$ :

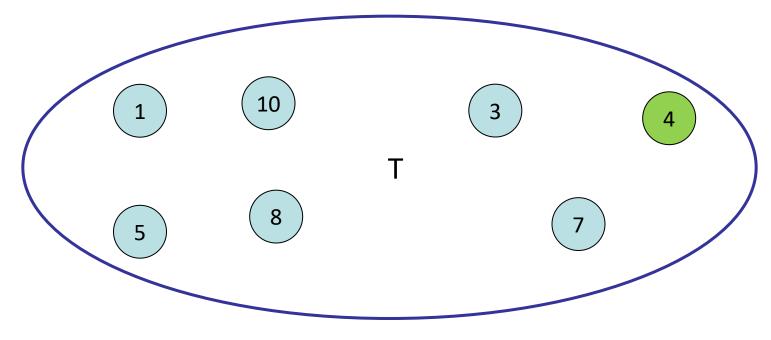




: Repräsentant



# Find(10) liefert 4





: Repräsentant



### Umsetzung als Datenstruktur

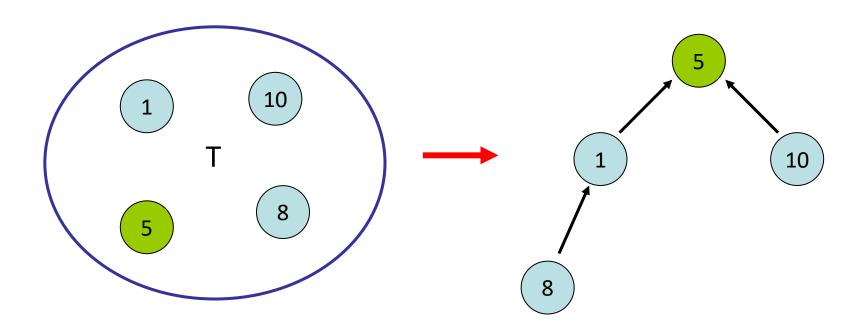
- Version 1: Liste mit Repräsentant als Listenende
  - Schnell bei Union $(T_1, T_2)$  (eine Liste an andere hängen) O(1)
  - Langsam bei Find(x) (durchlaufen bis zum Ende) O(n)
- Version 2: Baum mit Repräsentant als Wurzel, Elemente als Blattknoten unter Wurzel
  - Schnell bei Find(x) (sofort von Blatt an Wurzel) O(1)
  - Langsam bei Union(T<sub>1</sub>, T<sub>2</sub>) (für eine Partition alle Blattknoten und Wurzel umhängen) O(n)



#### Union-Find Datenstruktur: Gerichteter Baum

Idee: Repräsentiere jede Teilmenge T als gerichteten Baum mit Wurzel als Repräsentant

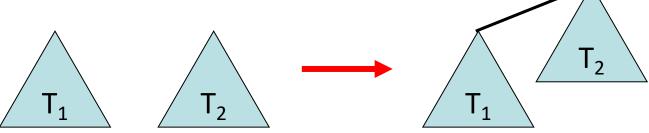
Wald von Bäumen für ganze Menge





### Realisierung der Operationen:

• Union $(T_1,T_2)$ :



• Find(x): Suche Wurzel des Baumes, in dem sich x befindet

#### Naïve Implementierung:

- Union(1,5), Union(8,5), Union(10,5), ...
- Union(1,5), Union(5,8), Union(8,10), ...

#### Beobachtung

Tiefe des Baums kann bis zu n sein (bei n Elementen)









#### Naïve Implementierung:

- Zeit für Find: O(n)
- Zeit für Union: O(1)
  - Annahme: T<sub>1</sub> und T<sub>2</sub> liegen durch Repräsentant vor

- Schluss?
  - Tiefe des Baums berücksichtigen



Gewichtete Union-Operation: Mache die Wurzel des flacheren Baums zum Kind der Wurzel des tieferen Baums.

### Beobachtung

- Unterschiedlich tiefe Bäumen?
  - Tiefe des neuen Baums ist gleich Tiefe des tieferen Baums
- Was ist bei gleicher Tiefe?
  - Tiefe nimmt um 1 zu

Worst Case bei n Elementen?

- 1
- 5
- 8
- 10
- 3
- 7
- 4



Gewichtete Union-Operation: Mache die Wurzel des flacheren Baums zum Kind der Wurzel des tieferen Baums.

Beh.: Die Tiefe eines Baums mit n Elementen ist höchstens O(log n)

- Die Tiefe von T=T<sub>1</sub> U T<sub>2</sub> erhöht sich nur dann, wenn Tiefe(T<sub>1</sub>)=Tiefe(T<sub>2</sub>) ist
- N(t): min. Anzahl Elemente in Baum der Tiefe t
- Es gilt  $N(t)=2\cdot N(t-1)=2^t$  mit N(0)=1
  - Beweis über Induktion
- Also ist  $N(\log n) = 2^{\log n} = n$



Gewichtete Union-Operation: Mache die Wurzel des flacheren Baums zum Kind der Wurzel des tieferen Baums.

#### Beobachtungen:

- Bei n Elementen ist die max. Tiefe eines Baums log n
- In einem Baum der Tiefe t sind min. 2<sup>t</sup> Elemente
- Bei n Elementen im Wald gibt es max. n/2<sup>t</sup> Knoten der Tiefe t



#### Mit gewichteter Union-Operation:

- Zeit für Find: O(log n)
- Zeit für Union: O(1)

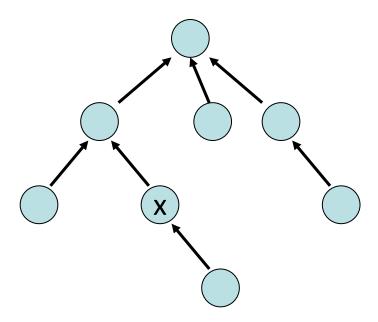
#### Geht das noch besser für Find?

- Best Case für Find: Nicht-Repräsentanten sind Blattknoten
  - Schnell beim Repräsentant
- Bei Find durchlaufen wir den Pfad vom Element zum Repräsentant
  - Elemente auf Pfad direkt auf Wurzel umleiten



#### Besser: gewichtetes Union mit Pfadkompression

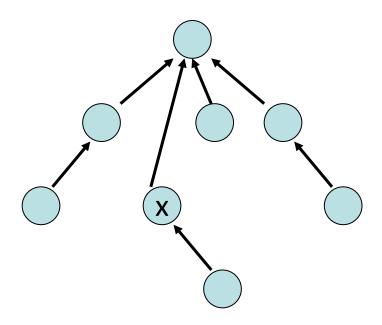
 Pfadkompression bei jedem Find(x): alle Knoten von x zur Wurzel zeigen direkt auf Wurzel





#### Besser: gewichtetes Union mit Pfadkompression

 Pfadkompression bei jedem Find(x): alle Knoten von x zur Wurzel zeigen direkt auf Wurzel





Theorem: Bei gewichtetem Union mit Pfadkompression ist die amortisierte Zeit für **Find** O(log\* n).

Was ist log\* n?



# Iterierter Logarithmus log\* n

Bemerkung: log\* n ist definiert als

```
log* n = 0 f \ddot{u} r n \le 1

log* n = min{i > 0 | log log ... log n \le 1} sonst

i-mal
```

#### Beipiele:

- log\* 2 = 1
- log\* 4 = 2
- $\log^* 16 = 3$
- $\log^* 2^{65536} = 5$

log\* n wächst sehr langsam



Theorem: Bei gewichtetem Union mit Pfadkompression ist die amortisierte Zeit für **Find** O(log\* n).

→ quasi konstant (log\* n ≤ 5 für sehr große Zahlen)

Ein paar Hilfsterme und Beobachtungen folgen...



## Union-Find Datenstruktur: rank(x)

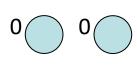
#### Ordne jedem Element x zu:

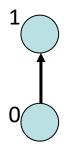
- rank(x) = Tiefe des Unterbaums von Wurzel x ohne Pfadkompression
  - MakeSet(x) setzt rank(x) = 0
  - Union( $T_1$ , $T_2$ ) by rank: Erhöht rank(x) um 1 für Wurzel der Vereinigung, wenn für die Repräsentanten  $x_1$ ,  $x_2$  von  $T_1$ ,  $T_2$  gilt: rank( $x_1$ ) = rank( $x_2$ )
  - Kann sich also nur unter Umständen für Repräsentanten ändern



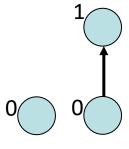
### Union-Find Datenstruktur: rank(x)

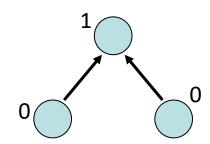
- Veränderung von rank(x) bei Union(T<sub>1</sub>,T<sub>2</sub>)
  - Gleiche Ränge





Unterschiedliche Ränge





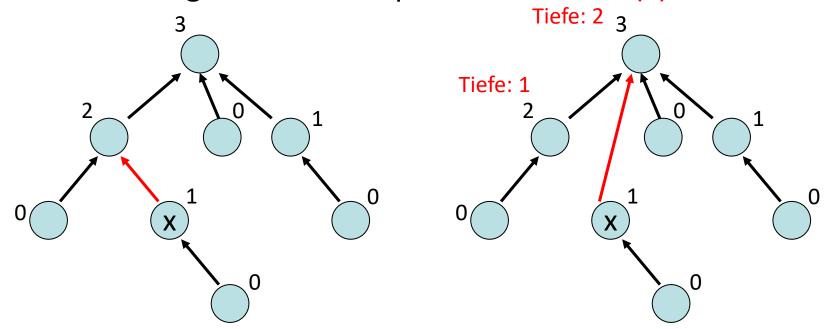
#### Beobachtungen:

 Wenn ein Repräsentant angehängt wird und damit kein Repräsentant mehr ist, ändert sich sein Rang nicht mehr



# Union-Find Datenstruktur: rank(x)

Auswirkung von Pfadkompression bei Find(x)



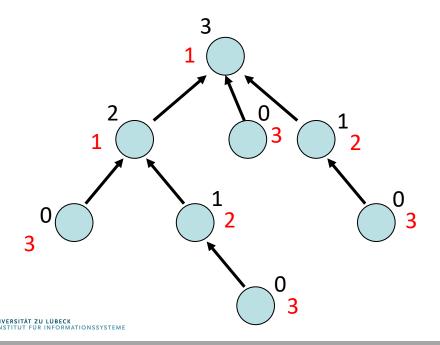
### Beobachtungen:

- Auf dem Weg zur Wurzel: rank(xi) aufsteigend
- Neuer Elternknoten hat h\u00f6heren Rang



### Jedes Element x gehört einer Gruppe gi an

- Gruppe  $g_i = \{v \mid log^{j+1} n < rank(v) \le log^j n\}, j > 0$ 
  - log<sup>j</sup> n wendet den Logarithmus j-mal auf n an

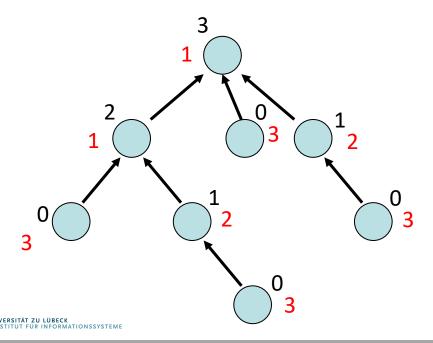


n = 8, 
$$\log^* 8 = 3$$
  
 $\log^1 8 = 3$   $g_1$   
 $\log^2 8 = 1.6$   $g_2$   
 $\log^3 8 = 0.7$   $g_3$   
 $\log^4 8 = -0.5$ 

$$rank(x)$$
  $j: x \in g_j$ 

### Beobachtungen:

- Knoten mit Rang log n (max. Rang) liegt in g<sub>1</sub>
  - Gruppe  $g_j = \{v \mid log^{j+1} n < rank(v) \le log^j n\}, j > 0$
  - $g_1 = \{v \mid \log^2 n < rank(v) \le \log^1 n\}$

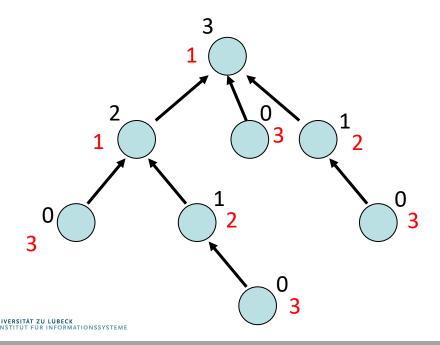


n = 8, 
$$\log^* 8 = 3$$
  
 $\log^1 8 = 3$  gs  
 $\log^2 8 = 1.6$  gs  
 $\log^3 8 = 0.7$  gs  
 $\log^4 8 = -0.5$ 

$$rank(x)$$
  $j: x \in g_j$ 

#### Beobachtungen:

- Knoten mit Rang 1 sind in g<sub>log\* n-1</sub> oder in g<sub>log\* n</sub>
- Knoten mit Rang 0 sind in g<sub>log\* n</sub> oder in g<sub>log\* n+1</sub>
  - Gruppe  $g_i = \{v \mid log^{j+1} n < rank(v) \le log^j n\}, j > 0$
- Max. (log\* n) + 1 Gruppen

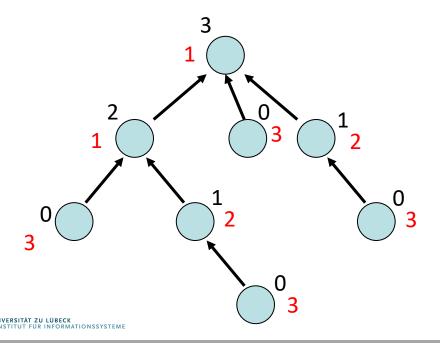


n = 8, 
$$\log^* 8 = 3$$
  
 $\log^1 8 = 3$  gs  
 $\log^2 8 = 1.6$  gs  
 $\log^3 8 = 0.7$  gs  
 $\log^4 8 = -0.5$ 

$$rank(x)$$
  $j: x \in g_j$ 

#### Beobachtungen:

- Es gibt maximal 2n/log<sup>j</sup> n Knoten in g<sub>i</sub>.
  - g<sub>i</sub> enthält Knoten mit Rang zwischen [log<sup>j-1</sup> n, log<sup>j</sup> n]
  - n/2<sup>i</sup> Knoten, die einen Rang i haben können (siehe N(t))
  - Summe mit Abschätzung nach oben



n = 8, 
$$\log^* 8 = 3$$
  
 $\log^1 8 = 3$  g  
 $\log^2 8 = 1.6$  g  
 $\log^3 8 = 0.7$  g  
 $\log^4 8 = -0.5$ 

$$rank(x)$$
  $j: x \in g_j$ 

IM FOCUS DAS LEBEN

Theorem: Bei gewichtetem Union mit Pfadkompression ist die amortisierte Zeit für **Find** O(log\* n).

Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Operationen O(m log\* n).

- Sequenz aus m Operationen besteht aus
  - Union: max. n-1
  - Find: min. m-(n-1)



Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Union und Find Operationen O(m log\* n).

- Max. n-1 Union Operationen
  - Aufwand pro Union von O(1) (konstant)
  - Also Aufwand in O(n)
- Max. m Find Operationen
  - Aufwand abhängig von Anzahl der Knoten, die durch Pfadkompression auf die Wurzel umgeleitet werden.



Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Union und Find Operationen O(m log\* n).

### Begründung:

Maximal m Find Operationen

- Find(x): Pfad von x zur Wurzel
  - Jeder Knoten auf dem Pfad wird umgeleitet
  - Kosten (1 Einheit) für das Umleiten aufteilen
    - Wenn group(x) > group(parent(x)): Find(x) zusortieren
    - 2. Wenn group(x) = group(parent(x)) : x zusortieren
- Zusortierte Kosten aufsummieren



Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Union und Find Operationen O(m log\* n).

### Begründung:

Kosten für das Umleiten aufteilen

- 1. Wenn group(x) > group(parent(x)) : Find(x) zusortieren
  - Oder wenn parent(x) Wurzel
  - Max. log\* n + 1 Gruppen
  - Wechsel von einer Gruppe zur nächsten also max. log\* n
  - Kosten, die Find(x) zusortiert werden: max. log\* n
- 2. Wenn group(x) = group(parent(x)) : x zusortieren



Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Union und Find Operationen O(m log\* n).

- 2. Wenn group(x) = group(parent(x)) : x zusortieren
  - Pro Gruppe max. 2n/log<sup>j</sup> n Knoten x<sub>j</sub>
  - Jeder Knoten x<sub>i</sub> kann max. log<sup>j</sup> n mal umgeleitet werden
    - Dann gehört Elternknoten der nächsten Gruppe an
    - Kosten gehen an Find ab nächstem Mal
  - Pro Gruppe erhalten die Knoten max. 2n Kosten
  - Bei max. log\* n + 1 Gruppen: max. 2n (log\* n + 1)



Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Union und Find Operationen O(m log\* n).

- Max. m Find Operationen
  - Pro Find(x): O(log\* n)
  - Bei m Find Operationen: O(m log\* n)
  - Für n Knoten über alle m Find Operationen:
    O(n (log\* n + 1)) = O(n (log log\* n)) = O(n (log\* log n)) =
    O(n log\*n)
  - O(m log\* n) + O(n log\* n) = O((m+n) log\* n)



Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Operationen O(m log\* n).

- Aufwand für Union und Find Operationen
  - Union: O(n)
  - Find: O( (m+n) log\* n)
  - Zusammen: O((m+n) log\* n)
- Wenn m >> n : O(m log\* n)
  - Quasi linear abhängig von m, da log \* n quasi konstant



Theorem: Bei gewichtetem Union mit Pfadkompression ist die amortisierte Zeit für **Find** O(log\* n).

Theorem: Bei gewichtetem Union mit Pfadkompression und gegeben n≥2 Elementen ist die amortisierte Zeit für m≥n Operationen O(m log\* n).

- Wenn m >> n sind fast alle m Operationen Finds
  - Da Anzahl an Unions max. n-1
  - Gleiches Argument für n MakeSet(x) Operationen
- Amortisierte Zeit für eine Find Operation O(log\* n)



## Zusammenfassung

- Find: O(log\* n) amort., Union: O(1)
- Können wir Find auf O(1) bringen?
  - Nur wenn Union nicht mehr in O(1)
  - Die Find-Abschätzung kann tatsächlich noch deutlich verbessert werden¹:  $O(\alpha(n))$  amort., wobei  $\alpha$  die Umkehrfunktion der Ackermannfunktion ist, also SEHR SEHR langsam wächst
- Man kann nicht gleichzeitig Find und Union auf O(1) bringen<sup>2</sup>



<sup>&</sup>lt;sup>1</sup> Tarjan, Robert E.; van Leeuwen, Worst-case analysis of set union algorithms, Journal of the ACM 31 (2), S. 245–281, **1984**