
Algorithmen und Datenstrukturen

Partitionierung von Mengen, Disjunkte Mengen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren



DataStructures.jl

This package implements a variety of data structures, including

- Deque (implemented with an [unrolled linked list](#))
- CircularBuffer
- CircularDeque (based on a circular buffer)
- Stack
- Queue
- Priority Queue
- Fenwick Tree
- Accumulators and Counters (i.e. Multisets / Bags)
- Disjoint Sets
- Binary Heap
- Mutable Binary Heap
- Ordered Dicts and Sets
- RobinDict and OrderedRobinDict (implemented with [Robin Hood Hashing](#))
- SwissDict (inspired from [SwissTables](#))
- Dictionaries with Defaults
- Trie
- Linked List and Mutable Linked List
- Sorted Dict, Sorted Multi-Dict and Sorted Set
- DataStructures.IntSet
- SparseIntSet
- DiBitVector
- Red Black Tree
- AVL Tree
- Splay Tree

DataStructures.jl

Contents

- Deque
- CircularBuffer
- CircularDeque
- Stack and Queue
- Priority Queue
- Fenwick Tree
- Accumulators and Counters
- Disjoint-Set
- Heaps
- OrderedDicts and OrderedSets
- DefaultDict and DefaultOrderedDict
- RobinDict
- SwissDict
- Trie
- Linked List
- Mutable Linked List
- DataStructures.IntSet

Danksagung

Die nachfolgenden Präsentationen wurden mit ausdrücklicher Erlaubnis des Autors und einigen Änderungen übernommen aus:

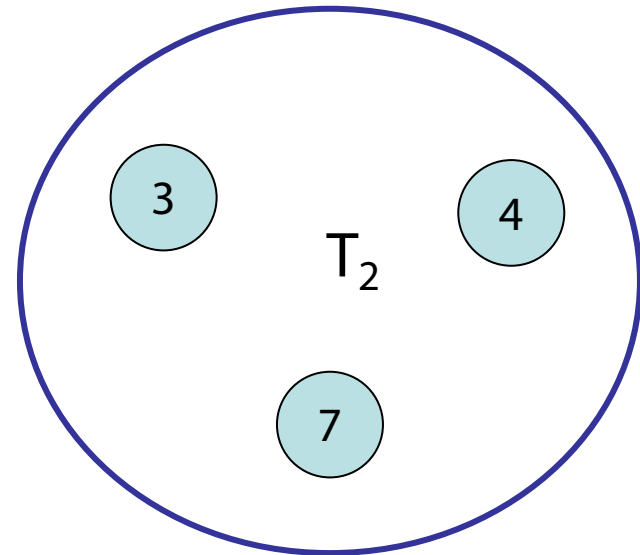
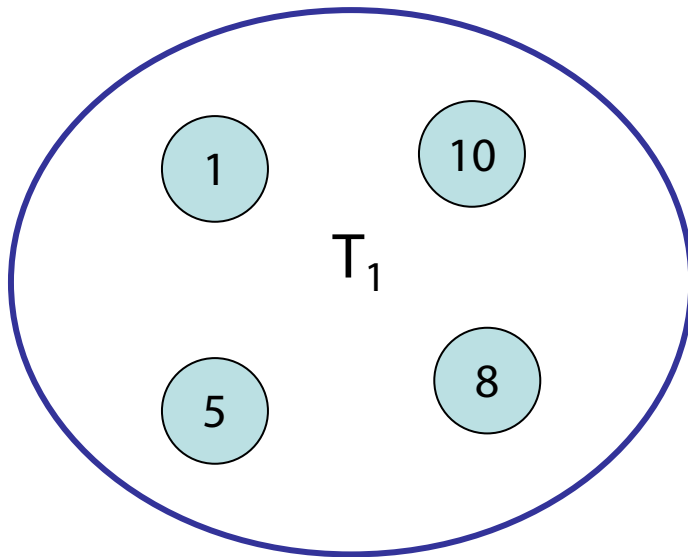
- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 6: Verschiedenes) gehalten von Christian Scheideler an der TUM
<http://www14.in.tum.de/lehre/2008WS/ea/index.html.de>



Partitionen einer Menge

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

Identifizierung
einer Partition?



Datenstruktur für Disjunkte Mengen

Wozu brauchen wir so eine Datenstruktur?

- Anwendungen im Bereich Data-Mining
 - Clusterbildung und Clusterverschmelzung
- Effiziente Implementierung von Algorithmen für Graphen (kommt demnächst)

Was muss die Datenstruktur möglichst in $O(1)$ können?

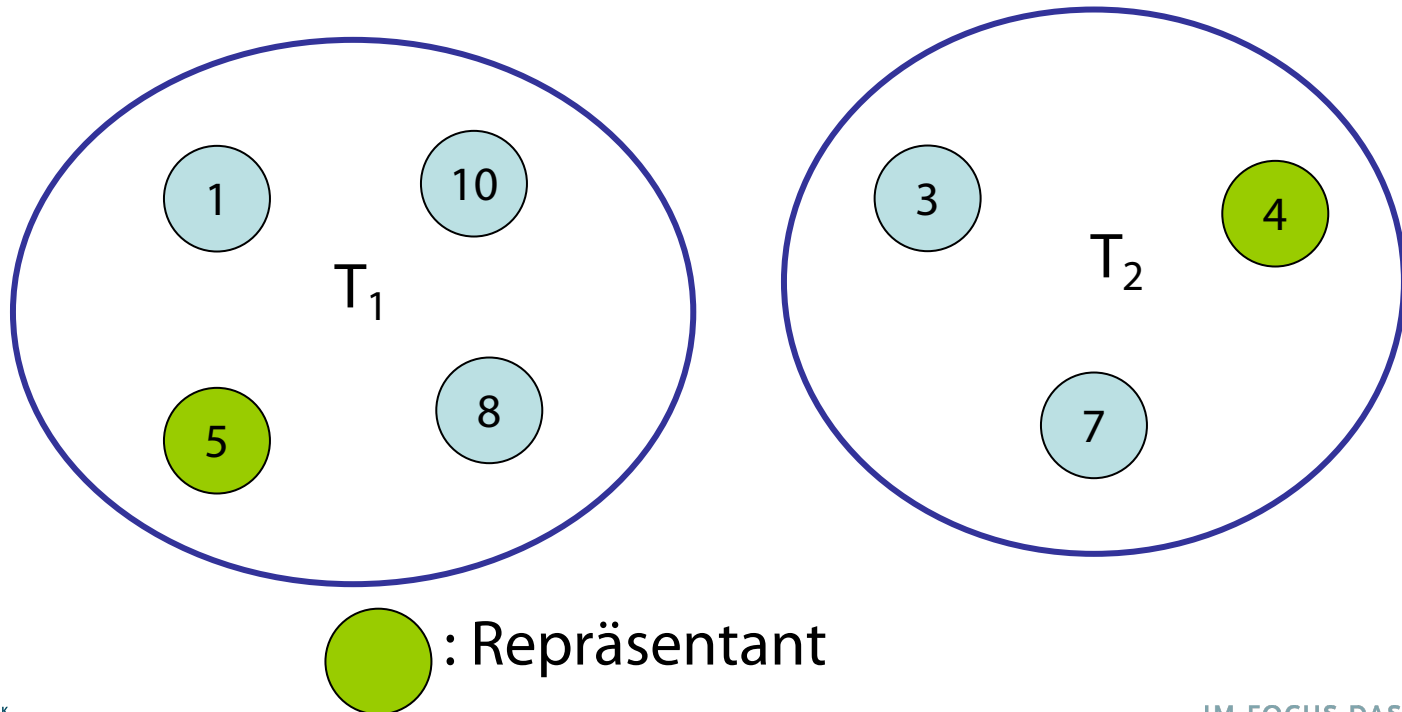
- Testen, ob zwei Elemente zu derselben Menge gehören
- Zwei Mengen vereinigen

Repräsentation von Disjunkten Mengen

- Häufiges Verwendungsmuster:
 - Initiale Bildung von einelementigen Mengen
 - Anschließende Vereinigung von Mengen (Partitionen werden immer größer, deren Anzahl nimmt ab)
- Repräsentation mit ADTs: Menge von Teilmengen?
 - **find**: Sind zwei Knoten in gleicher Menge?
 - **find** am **Anfang** in $O(n)$
 - n einelementige Mengen
 - **find** am **Ende** in $O(\log n)$
 - eine n -elementige Menge
 - **union** in $O(?)$
- **Bessere Realisierung** ist anzustreben!

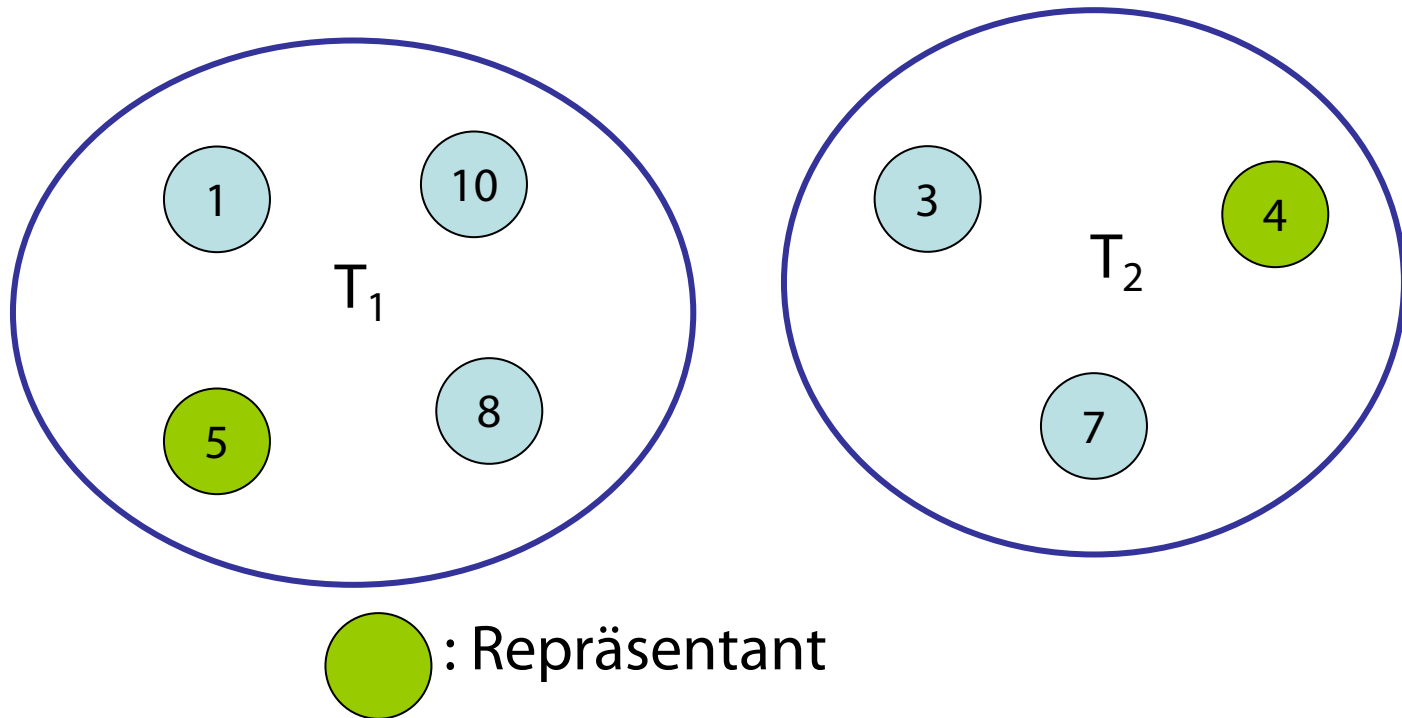
Identifizierung einer Partition

- Element aus Partition als Repräsentant
 - $T = \{1, 5, 8, 10, 3, 4, 7\}$
 - T_1 : Repräsentant 5
 - T_2 : Repräsentant 4



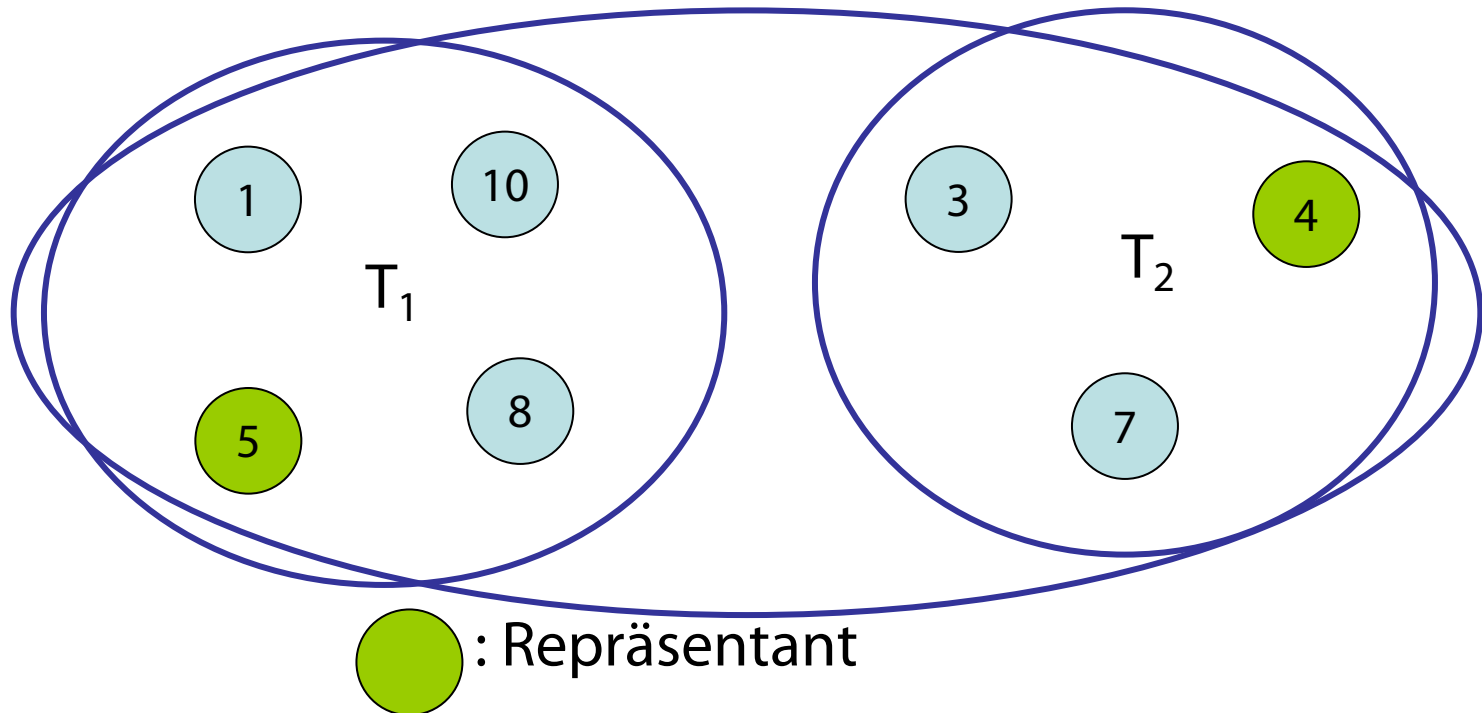
Test: Zugehörigkeit zur selben Partition?

- Gegeben: Partitionierung, Elemente: 1 und 10, 1 und 3
- Test über Gleichheit der Repräsentanten
 - Anforderung: schnell auf den Repräsentanten kommen



Vereinigung zweier Partitionen

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



Union-Find Datenstruktur: Partitionierung

Operationen:

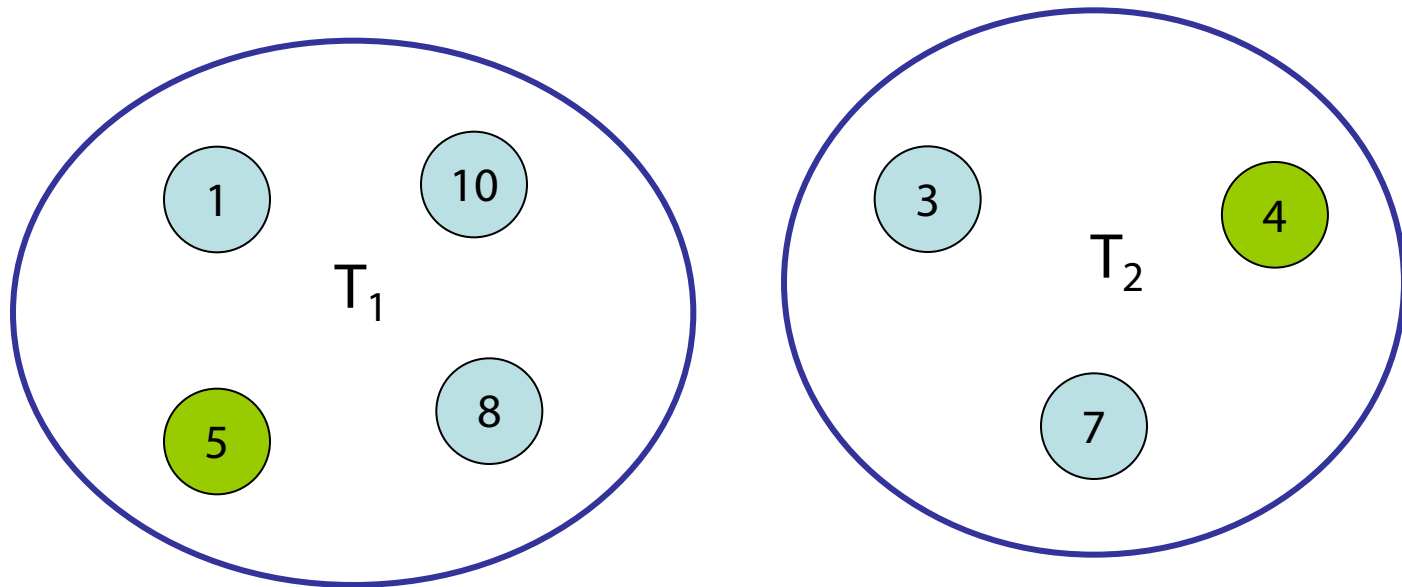
- **make_pset(elems)**: erzeugt eine partitionierte Menge mit Einer-Partitionen der Elemente **elems**
- **union($T_1, T_2, pset$)**: vereinigt Elemente in T_1 und T_2 zu $T_1 \cup T_2$ in **pset**, die Mengen T_1 und T_2 werden über den Repräsentanten eindeutig referenziert. Ggf. wird **pset** verändert, wenn die Repräsentanten gleich sind wird **pset** allerdings nicht verändert
- **find(x, pset)**: gibt (eindeutigen) Repräsentanten der Teilmenge aus **pset** zurück, zu der **x** gehört

Union-Find Datenstruktur

Sei $P = T_1 \cup T_2$

Find(10, P) liefert 5

Find(7, P) liefert 4

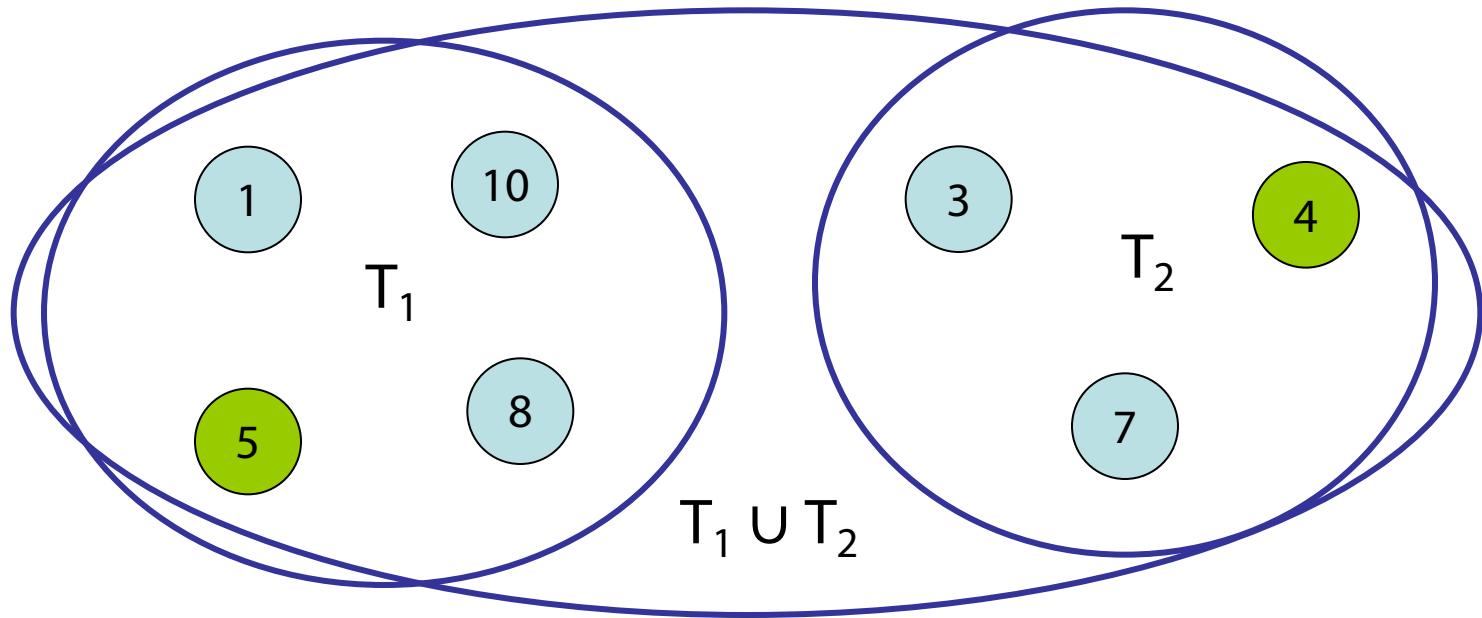


 : Repräsentant

Union-Find Datenstruktur

Sei $P = T_1 \cup T_2$

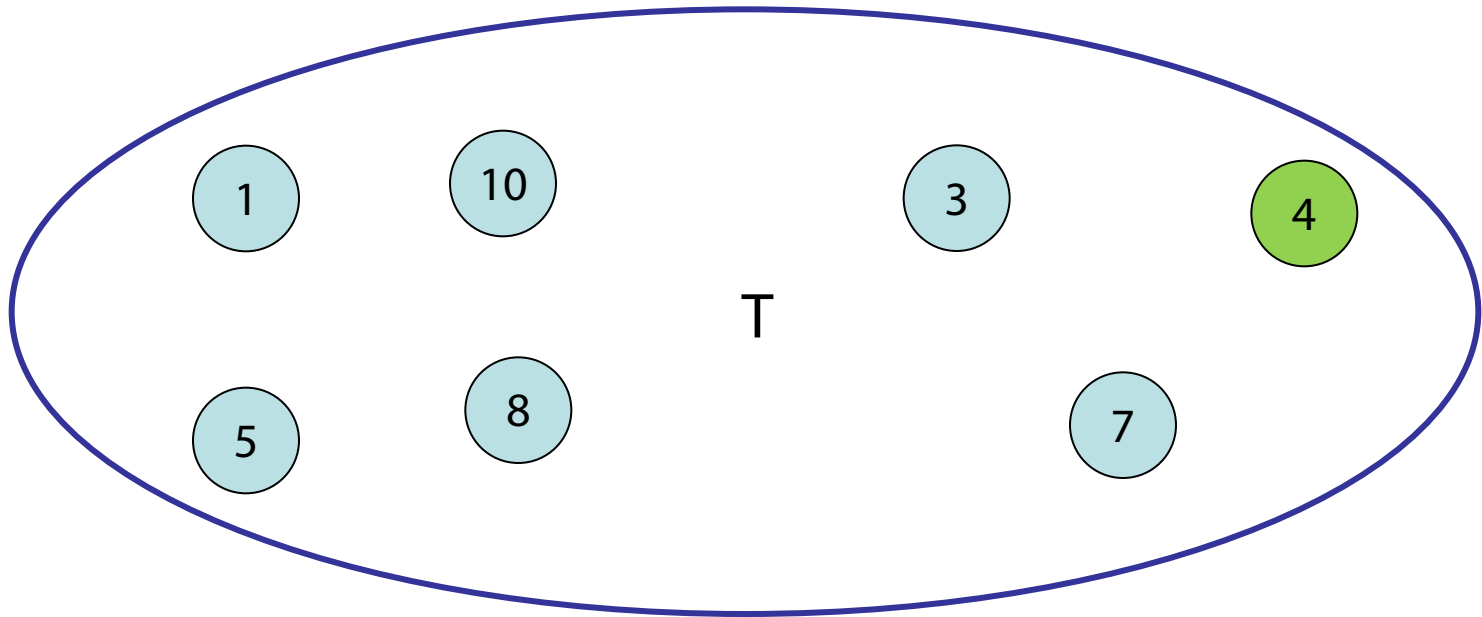
Union(T_1, T_2, P):



 : Repräsentant

Union-Find Datenstruktur

Find(10, P) liefert 4



 : Repräsentant

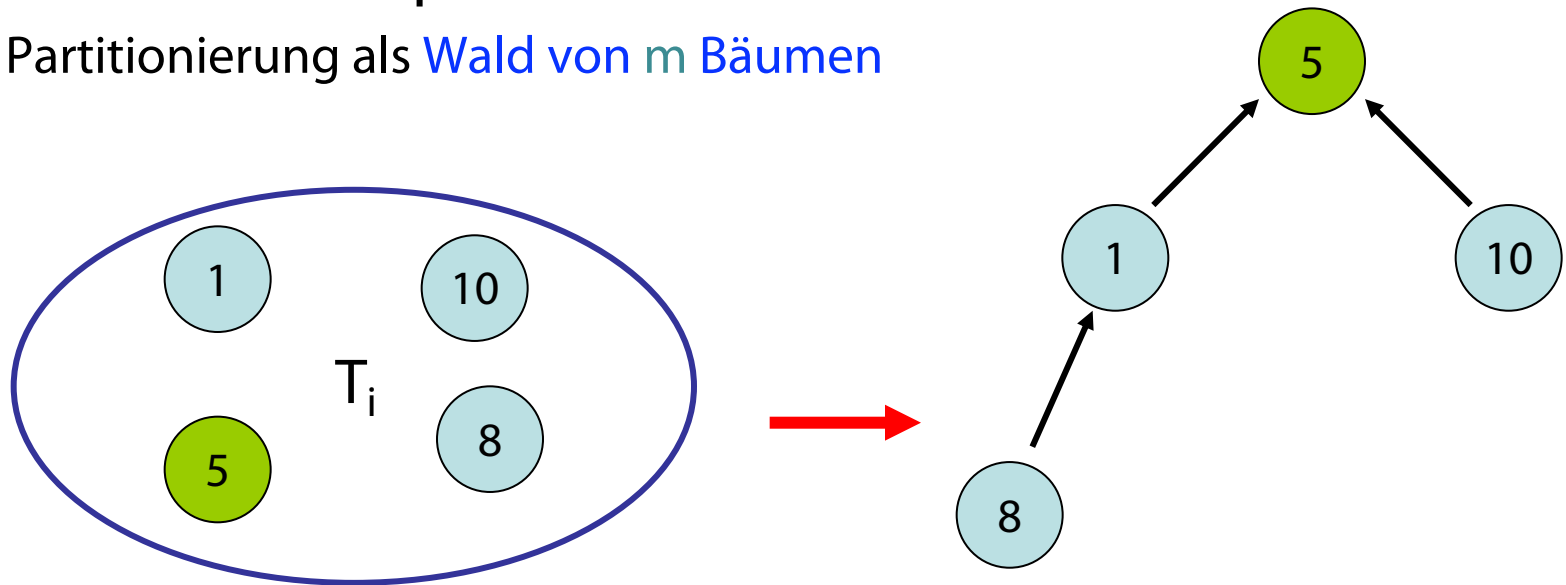
Repräsentation einer Partition?

1. **Verkettete Liste** mit Repräsentant als Listenende und Endelementzeiger (wie bei Schlangen)
 - **Schnell** bei **union** (eine Liste an andere hängen) $O(1)$
 - **Langsam** bei **find** (durchlaufen bis zum Ende) $O(n)$
2. **Zweischichtiger Baum** mit Repräsentant als Wurzel, Elemente als Blattknoten unter Wurzel mit Zeigern auf Wurzel
 - **Schnell** bei **find** (sofort von Blatt an Wurzel) $O(1)$
 - **Langsam** bei **union** (für eine Partition alle Blattknoten und Wurzel umhängen) $O(n)$

Union-Find Datenstruktur: Gerichteter Baum

Idee: Repräsentiere jede der m Teilmengen T_i einer Partitionierung P als **mehrschichtigen** gerichteten Baum mit Wurzel als Repräsentant

- Partitionierung als **Wald von m Bäumen**

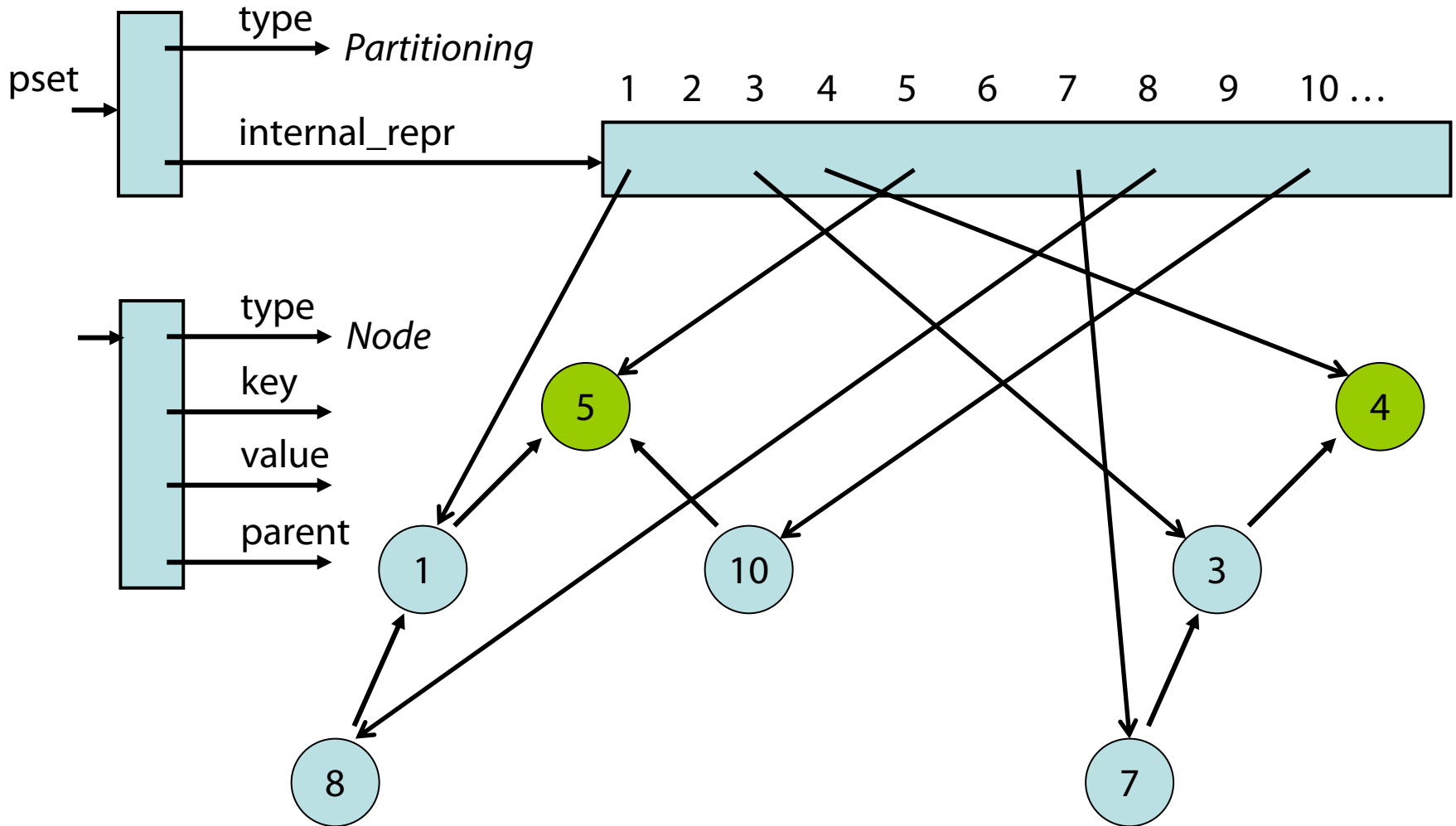


Union-Find-Datenstruktur als Black-Box-ADT

- Abbildung von Schlüsseln auf (interne) Knoten in $O(1)$ notwendig
 - Realisierbar bei **ganzen Zahlen als Schlüssel** z.B. über **Array**
 - Werte der Arrayelemente sind Zeiger auf (interne) Knoten
 - Schlüssel werden als Indexe verwendet
 - Assoziation Schlüssel zu Knoten in $O(1)$
 - Array gespeichert als interne Repräsentation in **pset**
- Weitere Techniken zur Assoziation von Schlüsseln zu Knoten in $O(1)$ lernen wir später kennen
 - Schlüssel könnten dann auch komplexe Objekte sein

Beispielimplementierung
in Julia vorhanden.

Internal View

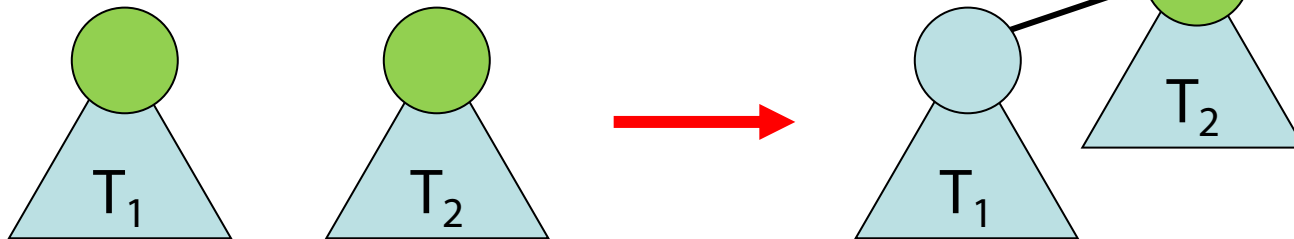


$$\text{Find}(10, p) = \text{Find}(7, p)$$

Union-Find Datenstruktur

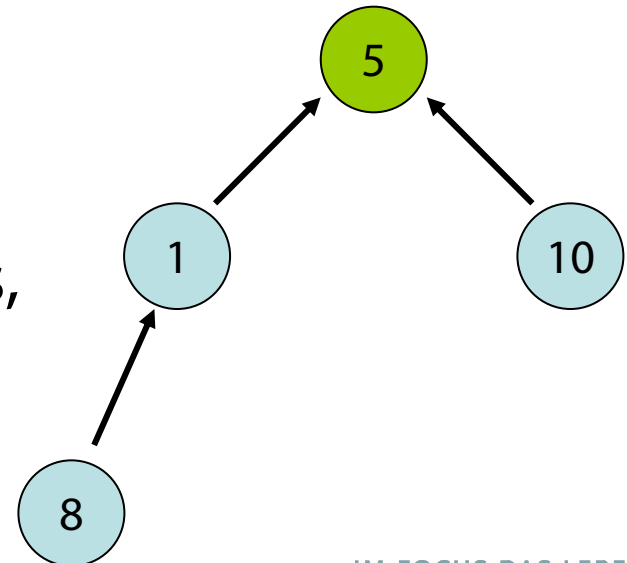
Realisierung der Operationen:

- **union**(T_1, T_2):



- T_1 und T_2 werden über den **Repräsentanten** referenziert

- **find**(x): Suche Wurzel des Baumes, in dem sich x befindet (Also: suche Repräsentanten)



Union-Find Implementierung

Naïve Implementierung:

- Rechtes Argument von `union` kommt nach oben
 - `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



Analyse der Komplexität

Naïve Implementierung:

- Zeit für `find(x)`: $O(n)$
- Zeit für `union(T1, T2)`: $O(1)$
 - Annahme: T_1 und T_2 liegen durch Repräsentanten vor

- Was können wir verbessern?
 - **Tiefe** des Baums **reduzieren** beim Erzeugen mit `union`

Union-Find Implementierung

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

Wir brauchen als Knotenkomponente zusätzlich zu **parent** noch die Tiefe, **rank** genannt

Ordne jedem Element **x** zu:

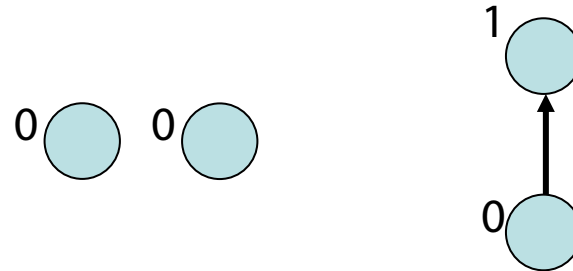
rank(x) = Tiefe des Unterbaums von Knoten **x**
(eigentlich inverser Baum: Kinder von **x** zeigen auf **x**)

- Initialer Wert: **x.rank = 0**
- **union(T₁, T₂):** Erhöht **rank(x)** um 1 für Wurzel **x** der Vereinigung, wenn für die Repräsentanten **T₁, T₂** gilt:
rank(T₁) == rank(T₂)

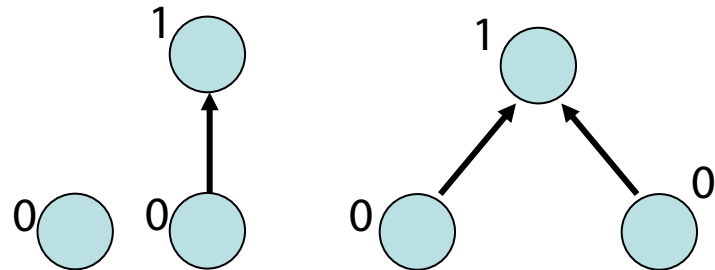
Union-Find Implementierung : rank(x)

- Veränderung von rank(x) bei $\text{union}(T_1, T_2)$

- Gleiche Ränge



- Unterschiedliche Ränge



Worst Case für rank(x) bei n Elementen?

Union-Find Implementierung

Beh.: Tiefe des Baums mit n Elementen ist in $O(\log n)$

Begründung:

- Die Tiefe von $T = T_1 \cup T_2$ erhöht sich nur dann, wenn $\text{Tiefe}(T_1) = \text{Tiefe}(T_2)$ ist
- Sei $N(t)$ die 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 Substitutionsmethode (oder Induktion)
- Für $t = \log n$ sind alle Knoten unterbracht, denn $N(\log n) = 2^{\log n} = n$
- Tiefe t in $O(\log n)$

Union-Find Implementierung

Beobachtungen (*):

- Bei n Elementen ist die max. Tiefe eines Baums $\log n$
- In einem Baum der Tiefe t sind min. 2^t Elemente
- Bei n Elementen im gesamten Wald gibt es max. $n/2^t$ Knoten der Tiefe t

Gewichtetes Union: Analyse der Komplexität

Mit gewichteter Union-Operation:

- Zeit für **find**: $O(\log n)$
- Zeit für **union**: $O(1)$

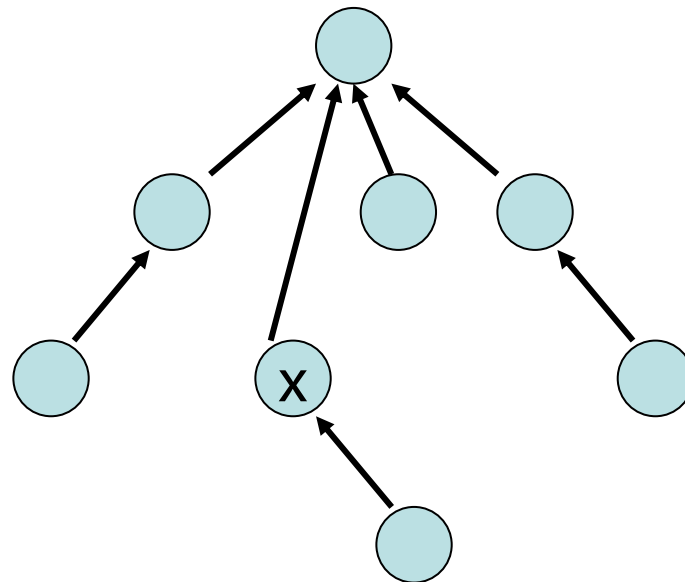
Geht das noch besser für **find**?

- **Best Case für Find-Operation**: Eingabe ist Repräsentant bzw. führt immer sofort zum Repräsentanten: $O(1)$
- Bei **find(x)** durchlaufen wir sowieso den Pfad vom Element **x** zum Repräsentanten
 - **Idee: Elemente auf Pfad direkt auf Wurzel (Repräsentant) umleiten (Pfadkompression)** (jeweils konstanter Aufwand)

Union-Find: Verbesserung

Besser: gewichtetes Union mit Pfadkompression

- Pfadkompression bei **find**: **alle** Knoten auf dem Pfad von **x** zur Wurzel zeigen hinterher direkt auf Wurzel



Jeder zugriffene Knoten auf dem Pfad wird umgeleitet

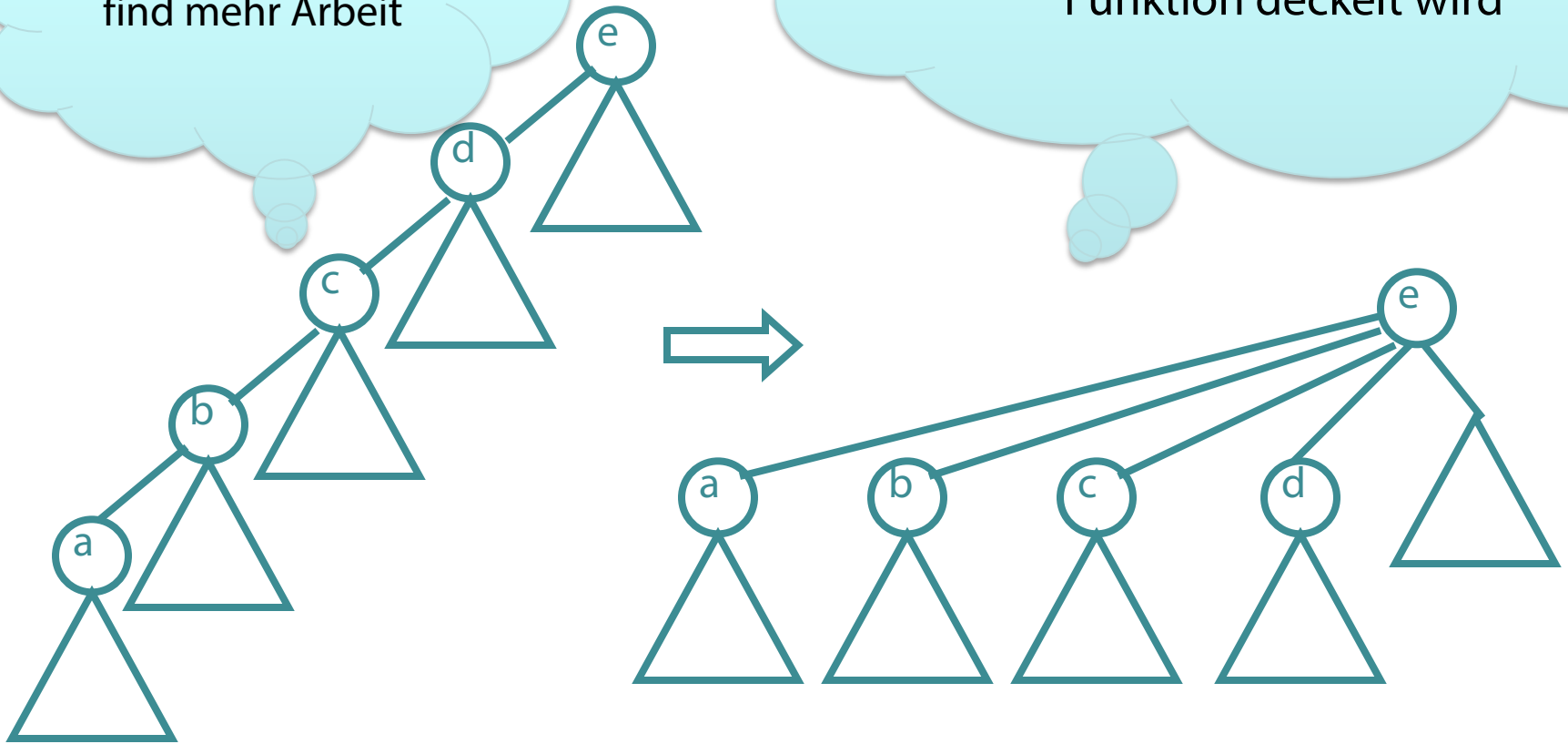
Hopcroft, J.E. and Ullman, J.D.; Set Merging Algorithms, SIAM Journal of Computing 2(4), S. 294-303, **1973**.

¹ Robert E. Tarjan, Jan van Leeuwen. Worst-case analysis of set union algorithms, Journal of the ACM 31 (2), S. 245-281, **1984**

find(a)

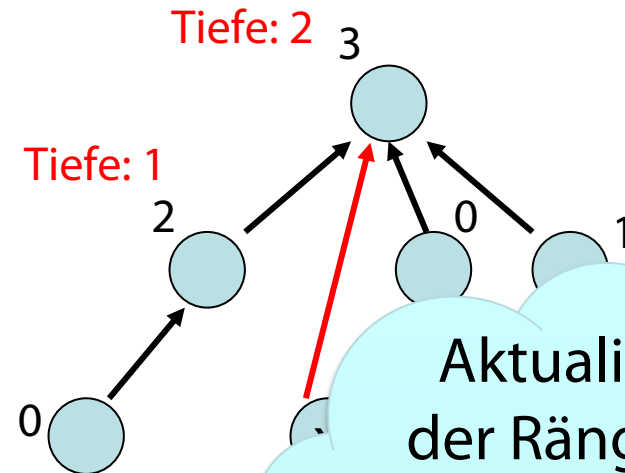
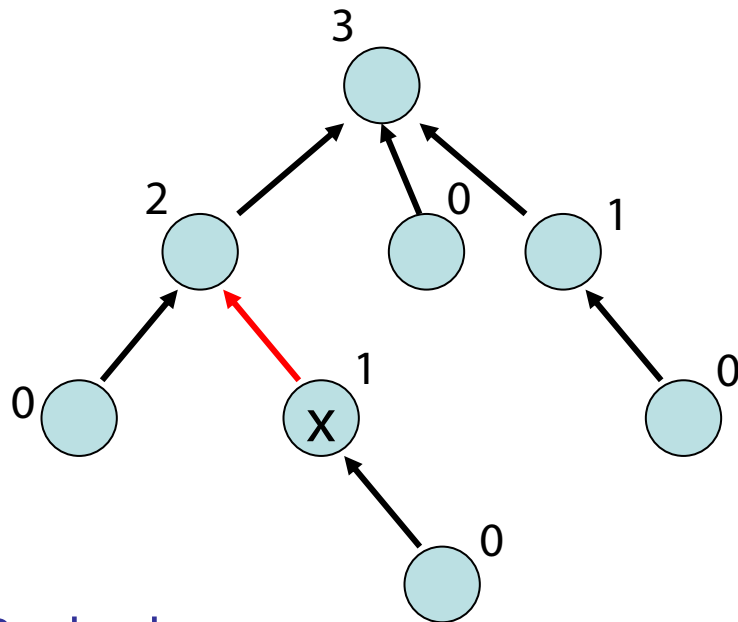
Union erzeugt für find mehr Arbeit

Amortisierte Analyse: Arbeit von Find so auf Union umverteilen, dass T_{union} in $O(1)$ bleibt und T_{find} von einer möglichst kleinen Funktion deckelt wird



Union-Find Implementierung : rank(x)

- Auswirkung von Pfadkompression bei **find**



Aktualisierung der Ränge kostet zu viel Zeit (man müsste alle Kinder anschauen)!

Beobachtungen:

- Auf dem Weg zur Wurzel: $\text{rank}(x_i)$ aufsteigend
- Neuer Elternknoten hat höheren Rang
- Ränge werden **nicht** angepasst und sind nur noch eine obere Schranke

Amortisierte Analyse

Theorem: Bei gewichtetem union und Pfadkompression ist die amortisierte Zeitfunktion für find in $O(\log^* n)$.

Was ist $\log^* n$?

Iterierter Logarithmus $\log^* n$

Bemerkung: $\log^* n$ ist definiert als

$$\log^* n = 0 \text{ für } n \leq 1$$

$$\log^* n = \min\{i > 0 \mid \underbrace{\log \log \dots \log n}_{i\text{-mal}} \leq 1\} \text{ sonst}$$

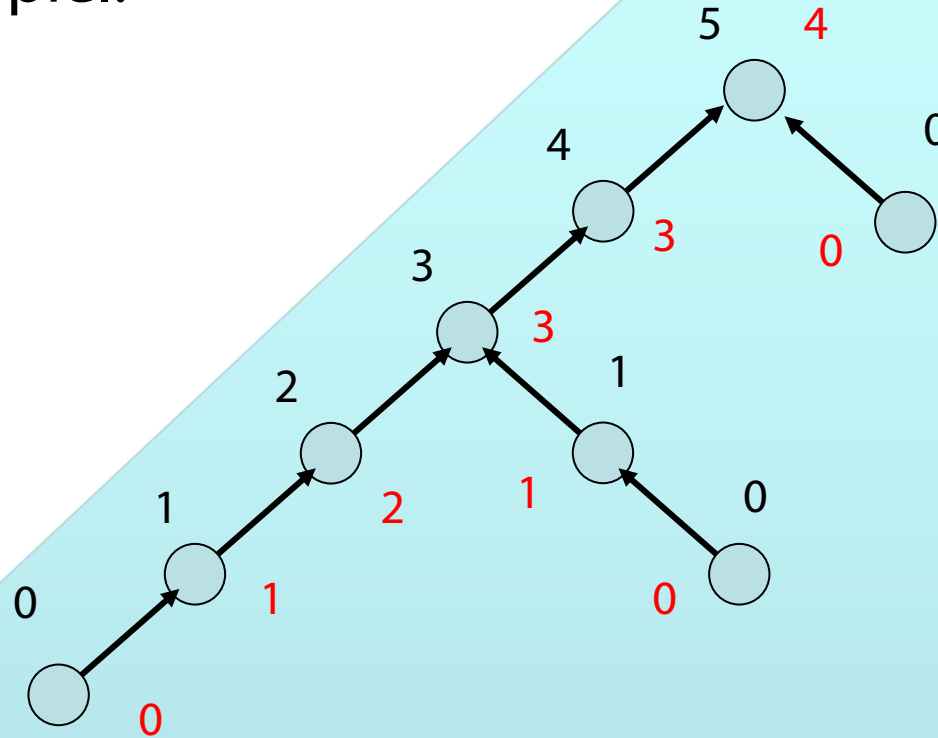
Beispiele:

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

$\log^* n$ wächst sehr langsam

Union-Find Datenstruktur

Beispiel:



x: rank

x: class

$i = \text{class}$

-1 0 1 2 3

4

5

a_i

-1, 0, 1, 2, 4,

16,

65536

rank

0 1 2 3,4 5,6,7,...16

...



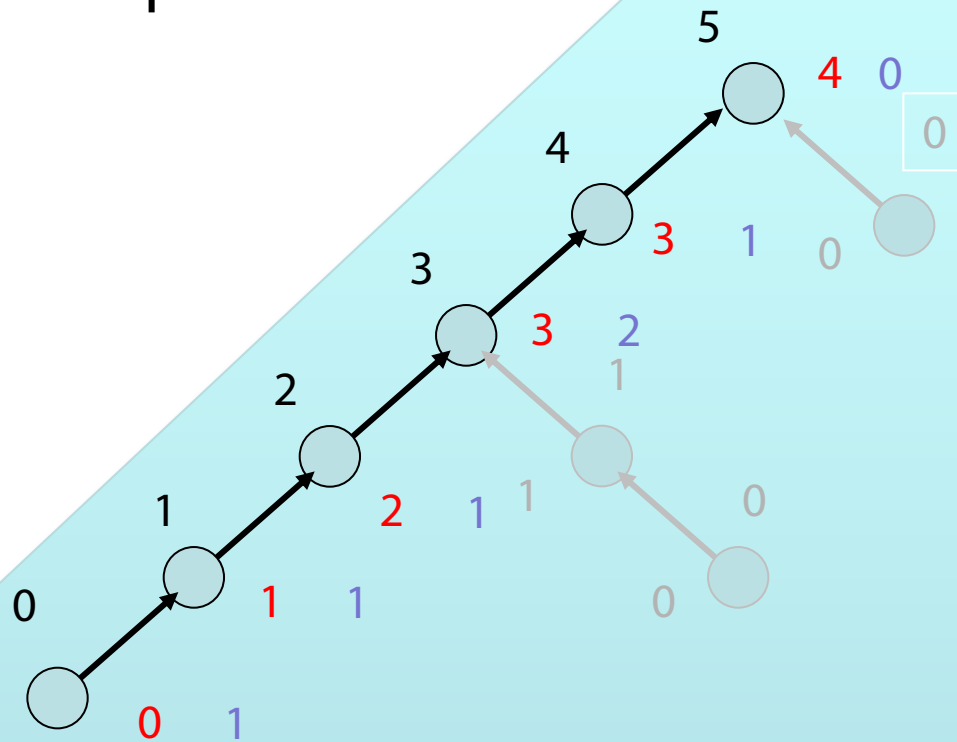
Amortisierte Analyse: Potentialmethode

Vorbereitung der Potentialdefinition:

- Sei $\text{dist}(x)$ die Anzahl von Schritten "nach oben" bis zum Klassenwechsel
- Sei $\text{dist}(x)$ die minimale Distanz von x zu einem Vorfahr y im tatsächlichen Union-Find-Baum T (mit Pfadkompression), so dass $\text{class}(y) > \text{class}(x)$ ist oder y die Wurzel ist
- dist für Wurzeln = 0

Union-Find Datenstruktur

Dist-Beispiel für T':



x: rank

x: class

x: dist

Amortisierte Analyse: Potentiale

Beweis (Fortsetzung):

- Bei **union** soll Potential zunehmen
 - Für **find** wird mehr Arbeit produziert
 - Rang der neuen Wurzel kann steigen
 - Werte $\text{dist}(x)$ von Knoten x unter der Wurzel steigen evtl. auch (wenn kein Klassenwechsel),
- Bei **find** für Knoten x soll Potential abgebaut werden
 - Nachfolgende Aufrufe von **find** für den gleichen Knoten in $O(1)$
 - Nach Umhängen von Knoten x an die Wurzel: $\text{dist}(x) = 1$
 - Wir wollen die Arbeit von **find** auf die Aufrufe von **union** umverteilen, so dass die asymptotische Komplexität von **union** sich nicht ändert (auf jeden Union-Aufruf kommt konstanter Anteil)
 - Wie viele Einzeloperation in **find** kann man auf **union** umlegen, damit das gewährleistet bleibt?

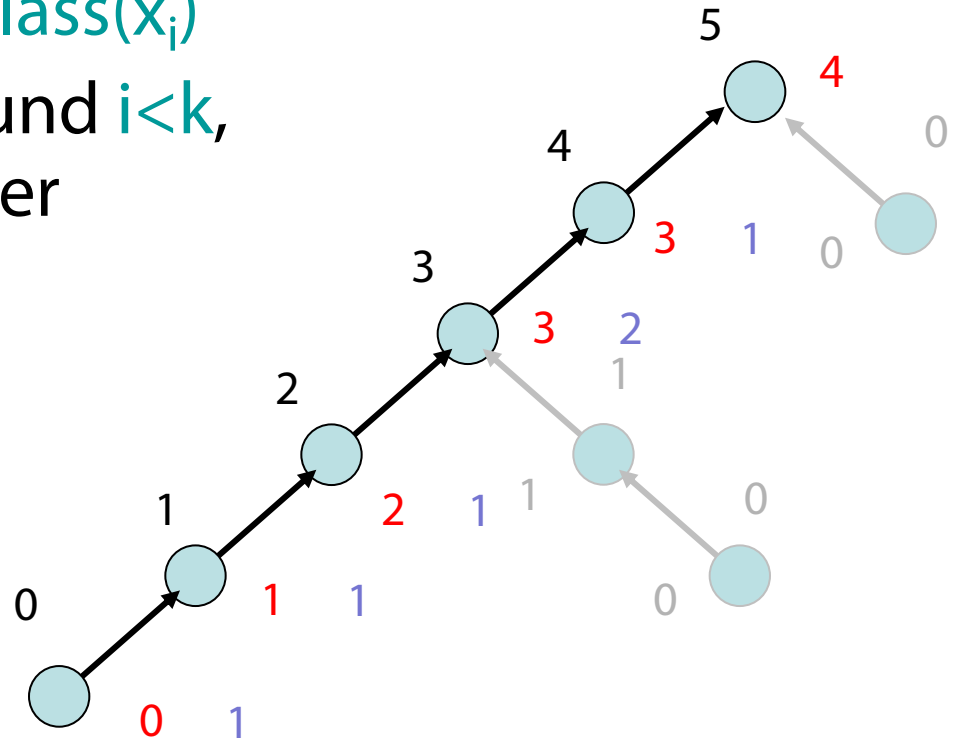
Definition der Potentialfunktion: Sei T ein Union-Find-Baum

$$\Phi(T) := c \sum_{x \in T} \text{dist}(x) \quad \text{für eine geeignete Konstante } c > 0$$

Union-Find: Amortisierte Analyse

Amortisierte Kosten von **find**:

- $x_0 \rightarrow x_1 \rightarrow x_2 \dots x_k$: Pfad von x_0 zur Wurzel in T'
- Es gibt auf dem Pfad höchstens $\log^* n$ Kanten (x_{i-1}, x_i) mit $\text{class}(x_{i-1}) < \text{class}(x_i)$
- Ist $\text{class}(x_{i-1}) = \text{class}(x_i)$ und $i < k$, dann ist $\text{dist}(x_{i-1})$ vor der Find-Operation ≥ 2 und nachher $= 1$



Union-Find: Amortisierte Analyse

Amortisierte Kosten von **find**:

- Damit können die Kosten für die Umlenkung aller Kanten (x_{i-1}, x_i) mit $\text{class}(x_{i-1}) = \text{class}(x_i)$ aus der Potentialverringerung “bezahlt” werden, denn die dist-Werte von x_{i-1} werden kleiner, wenn $\text{class}(x_{i-1}) = \text{class}(x_i)$, also $\Phi(s') - \Phi(s) < 0$
- Nur beim Klassenübergang gewinnt man nichts
 - $O(\log^* n)$ viele Klassenübergänge
- Amortisierte Kosten von **find** also $O(\log^* n)$

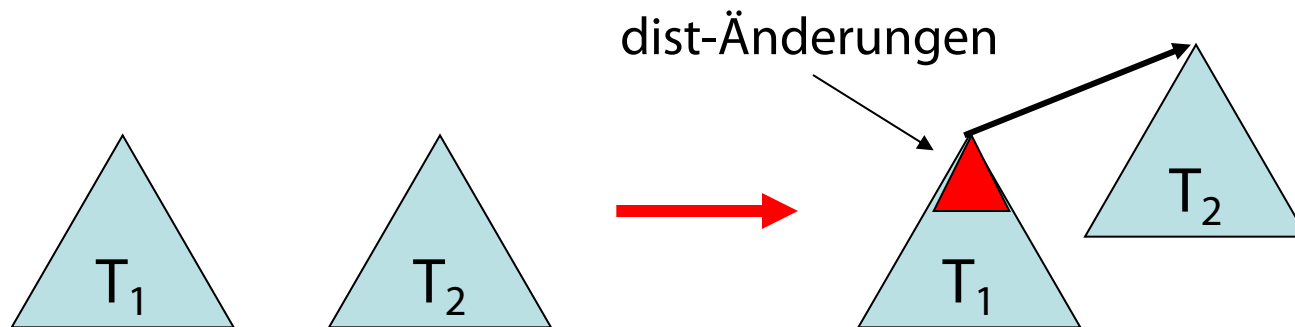
Überlegung

- Die Anzahl der Umhängevorgänge geht auf **find** und hängt von **n** ab.
- Wir würden eigentlich gern die Arbeit für *jeden* Umhängevorgang auf **union** umverteilen (“ aus Potentialverringerung bei **find** bezahlen”)
- Warum können wir nicht alle Schritte von **find** ausgehend von **x** auf die Union-Aufrufe umverteilen?
- Das wäre ja für **n** Elemente bei konstantem Umlenkungsaufwand maximal **n** Schritte
- Dann wäre **find** doch amortisiert in $O(1)$, oder?
- Haben wir eine Obergrenze für das Potential?

Union-Find: Amortisierte Analyse

Maximales Potential, das durch alle Union-Operationen produziert wird:

- **dist**-Änderungen über alle Unions bzgl. T' ist gleich $\Phi(T')$ (Reihenfolge der Unions egal)



- Potential von Baum T' mit n Knoten:

$$\Phi(T') \leq c \sum_{i=0}^{\log^* n} \sum_{x: \text{rank}(x) \in [a_{i-1}+1, a_i]} \text{dist}(x)$$

$$\text{class}(T_1) = \text{class}(T_2)$$

Union-Find: Amortisierte Analyse

$$\Phi(T') \leq c \sum_{i=0}^{\log^* n} \sum_{x: \text{rank}(x) \in [a_{i-1}+1, a_i]} \text{dist}(x)$$

Alle Unterbäume, deren Wurzel x den Rang $\text{rank}(x)=j$ haben, sind disjunkt und enthalten jeweils mind. 2^j Knoten: Es gibt $n/2^j$ Bäume der Tiefe j (siehe Beobachtung (*))

$$\Phi(T') \leq c \sum_{i=0}^{\log^* n} (n/2^j) a_i \text{ mit } j = a_{i-1} + 1, a_i \geq \text{dist}(x)$$

$$\leq c' \cdot n \sum_{i=0}^{\log^* n} a_i / 2^{a_i-1}$$

$$\leq c' \cdot n \sum_{i=0}^{\log^* n} 1$$

$$\in O(n \log^* n)$$

i=class	-1	0	1	2	3	4	5
a_i	-1	0	1	2	4	16	65536
rank	0	1	2	3,4	5,6,7,...16	...	

Nach n Umlenkungen bei **find**: $\Phi(T) \in O(n \log^* n) / n \cdot O(1) = O(\log^* n)$

NB: $\log^* n$ ist nicht asymptotisch eng (ist nur eine "lose" Abschätzung)

Zusammenfassung: Disjunkte Mengen

- $\text{find} \in O(\log^* n)$ amort., $\text{union} \in O(1)$
 - Die find -Abschätzung kann tatsächlich noch deutlich verbessert werden¹: $O(\alpha(n))$ amort., wobei α die Umkehrfunktion der Ackermannfunktion ist, also SEHR SEHR langsam wächst
- Können wir find auf $O(1)$ bringen?
 - Nur wenn union nicht mehr in $O(1)$
 - Man kann nicht gleichzeitig find und union in $O(1)$ bringen²

¹ Robert E. Tarjan, Jan van Leeuwen. Worst-case analysis of set union algorithms, *Journal of the ACM* 31 (2), S. 245–281, 1984

² M. Fredman, M. Saks. The cell probe complexity of dynamic data structures, In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing.*, S. 345–354, 1989