

# Algorithmen 1

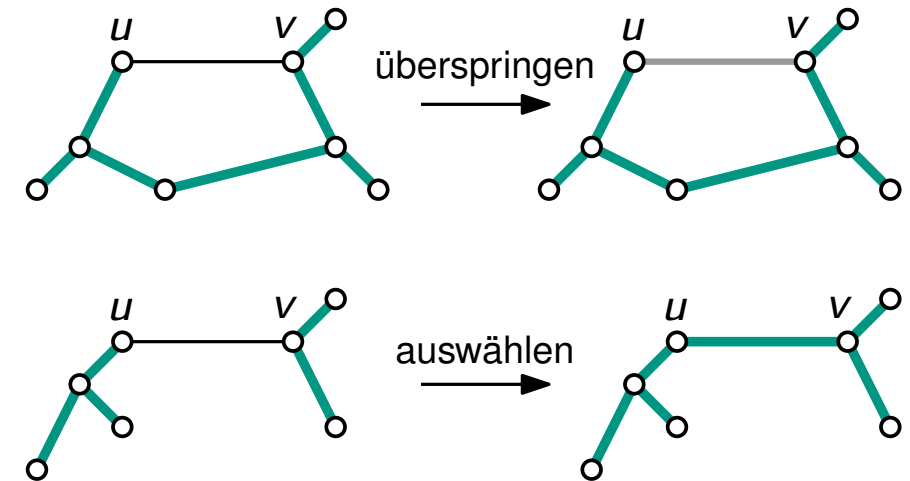
## Union-Find



# Motivation: Union-Find

## Letzte Vorlesung: Algorithmus von Kruskal

- wähle in jedem Schritt minimale Kante
- unter allen Kanten, die keinen Kreis mit schon gewählten Kanten bilden



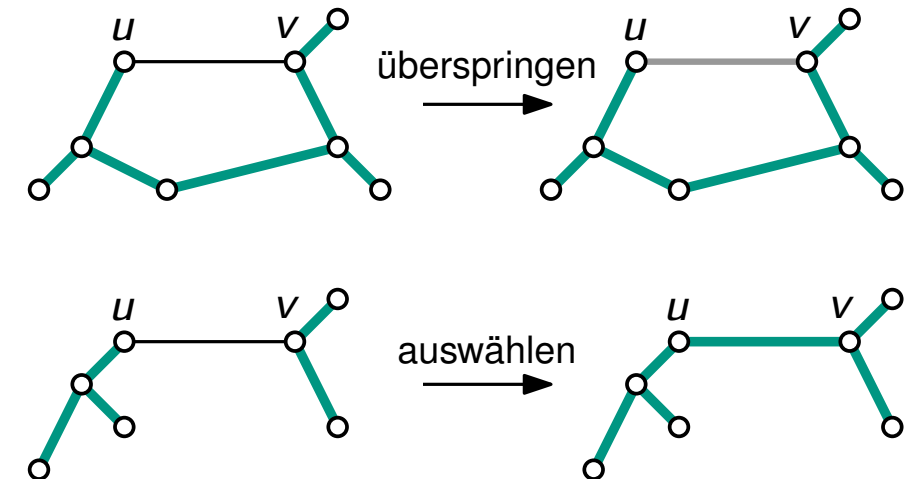
# Motivation: Union-Find

## Letzte Vorlesung: Algorithmus von Kruskal

- wähle in jedem Schritt minimale Kante
- unter allen Kanten, die keinen Kreis mit schon gewählten Kanten bilden

### Was wollen wir erreichen?

- schnell testen ob  $\{u, v\}$  Kreis schließt
- äquivalent:  $u$  und  $v$  liegen in der selben Komponente



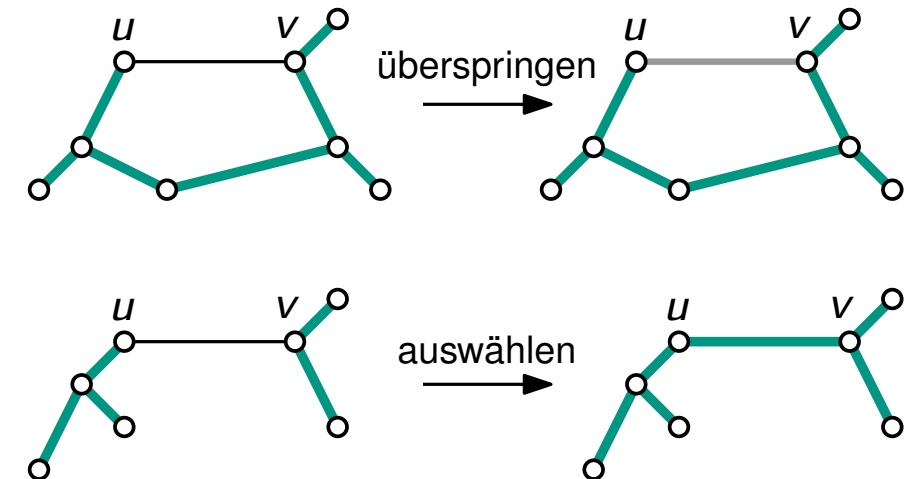
# Motivation: Union–Find

## Letzte Vorlesung: Algorithmus von Kruskal

- wähle in jedem Schritt minimale Kante
- unter allen Kanten, die keinen Kreis mit schon gewählten Kanten bilden

### Was wollen wir erreichen?

- schnell testen ob  $\{u, v\}$  Kreis schließt
- äquivalent:  $u$  und  $v$  liegen in der selben Komponente
- Einfügen einer Kante vereinigt zwei Komponenten



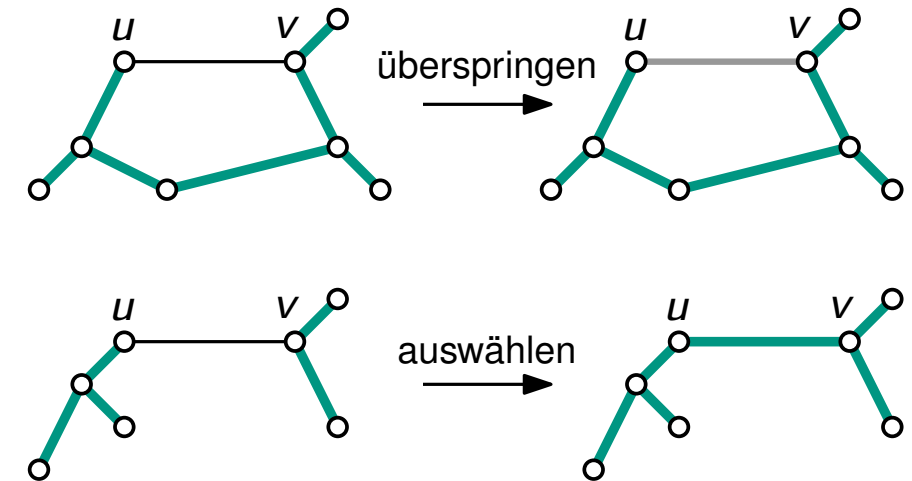
# Motivation: Union–Find

## Letzte Vorlesung: Algorithmus von Kruskal

- wähle in jedem Schritt minimale Kante
- unter allen Kanten, die keinen Kreis mit schon gewählten Kanten bilden

### Was wollen wir erreichen?

- schnell testen ob  $\{u, v\}$  Kreis schließt
- äquivalent:  $u$  und  $v$  liegen in der selben Komponente
- Einfügen einer Kante vereinigt zwei Komponenten
- gewünschte Operationen
  - **union** $(u, v)$ : vereinige Komponenten von  $u$  und  $v$
  - **find** $(v)$ : Komponente\* von  $v$
- Test ob  $u$  und  $v$  in der selben Komponente liegen: **find** $(u) = \text{find}(v)$



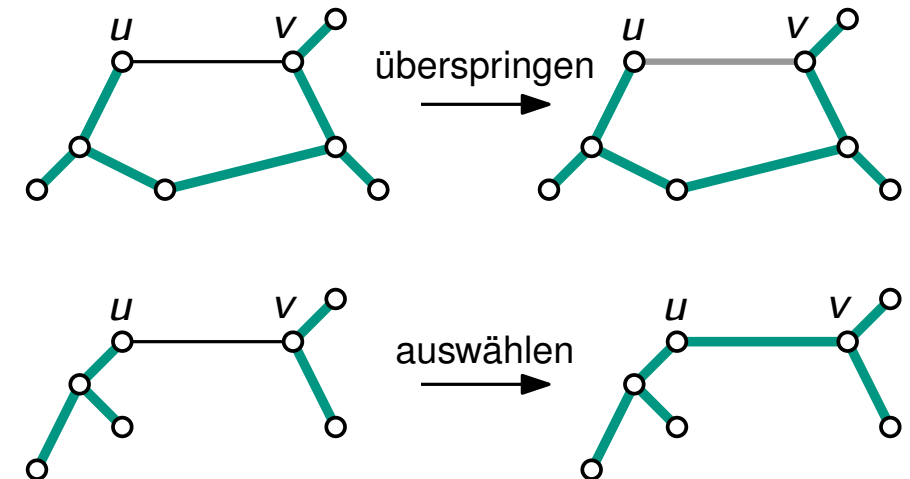
# Motivation: Union-Find

## Letzte Vorlesung: Algorithmus von Kruskal

- wähle in jedem Schritt minimale Kante
- unter allen Kanten, die keinen Kreis mit schon gewählten Kanten bilden

### Was wollen wir erreichen?

- schnell testen ob  $\{u, v\}$  Kreis schließt
- äquivalent:  $u$  und  $v$  liegen in der selben Komponente
- Einfügen einer Kante vereinigt zwei Komponenten
- gewünschte Operationen
  - **union** $(u, v)$ : vereinige Komponenten von  $u$  und  $v$
  - **find** $(v)$ : Komponente\* von  $v$
- Test ob  $u$  und  $v$  in der selben Komponente liegen: **find** $(u) = \text{find}(v)$



\*Umsetzung: bestimme für jede Komponente einen eindeutigen Vertreter und gib diesen zurück

# Union–Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

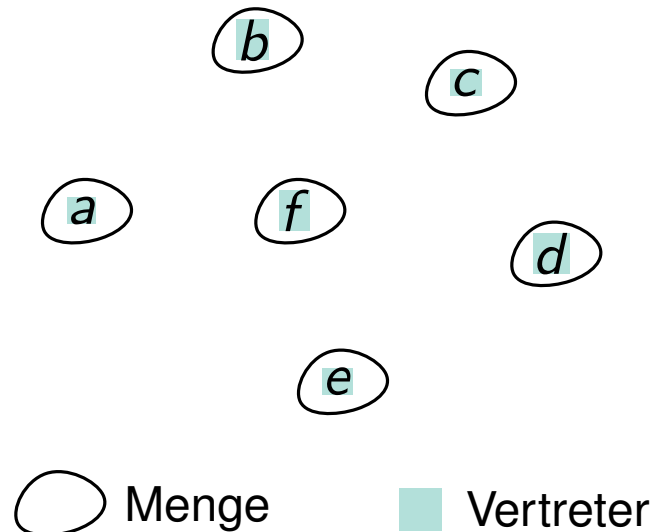
- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )

**find**( $f$ )

**find**( $c$ )

**union**( $b, d$ )

**find**( $c$ )

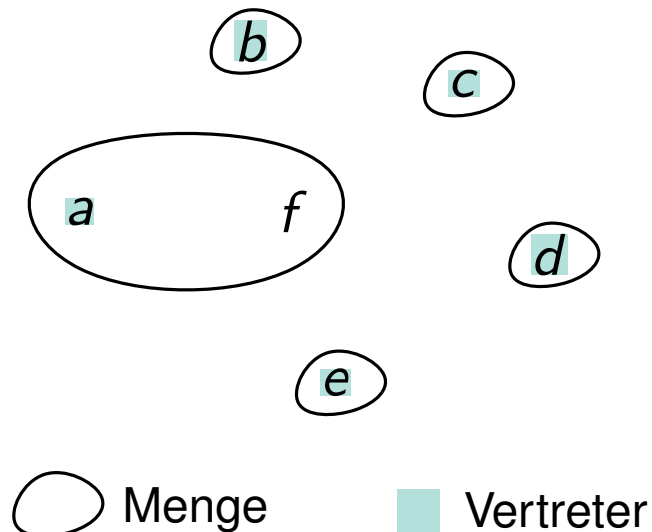


# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )

**find**( $f$ )

**find**( $c$ )

**union**( $b, d$ )

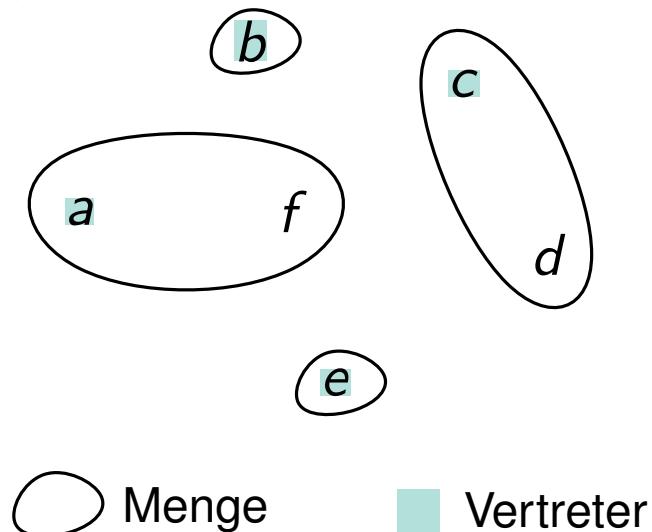
**find**( $c$ )

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )

**find**( $f$ )

**find**( $c$ )

**union**( $b, d$ )

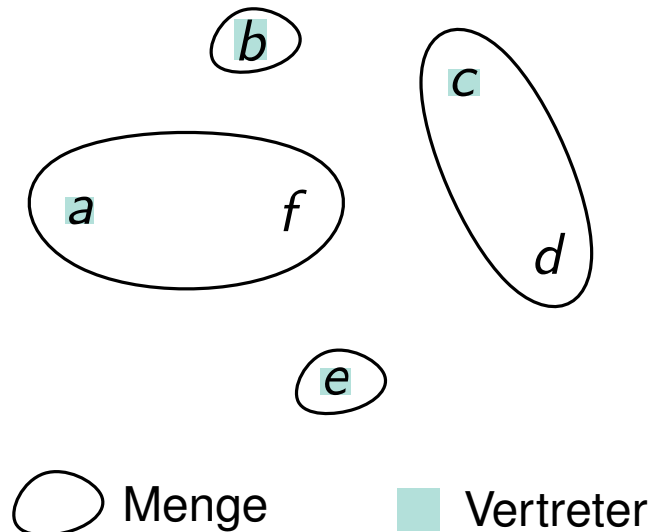
**find**( $c$ )

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )  $\rightarrow a$

**find**( $f$ )

**find**( $c$ )

**union**( $b, d$ )

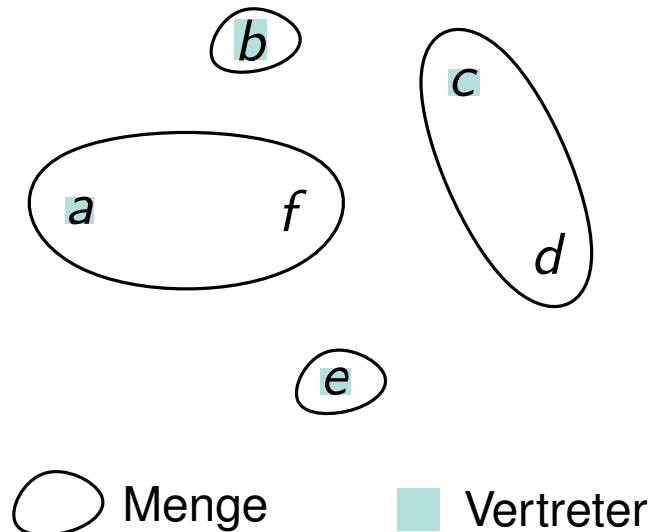
**find**( $c$ )

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )  $\rightarrow a$

**find**( $f$ )  $\rightarrow a$

**find**( $c$ )

**union**( $b, d$ )

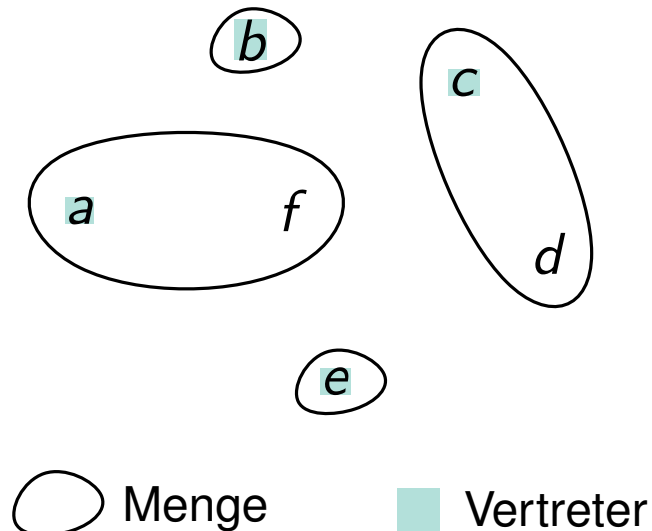
**find**( $c$ )

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )  $\rightarrow a$

**find**( $f$ )  $\rightarrow a$

**find**( $c$ )  $\rightarrow c$

**union**( $b, d$ )

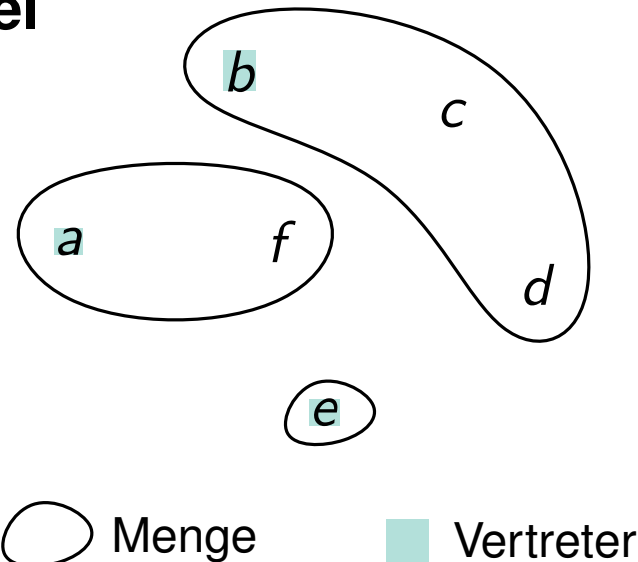
**find**( $c$ )

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )  $\rightarrow a$

**find**( $f$ )  $\rightarrow a$

**find**( $c$ )  $\rightarrow c$

**union**( $b, d$ )

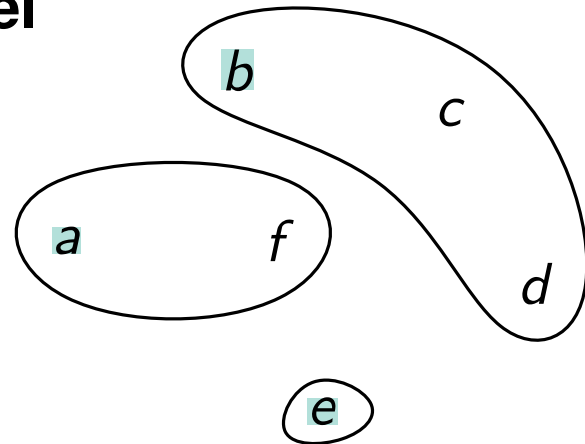
**find**( $c$ )

# Union-Find

## Etwas formaler (und ohne die Anwendung im Hinterkopf)

- Ausgangssituation: endliche Menge von disjunkten einelementigen Mengen
- Beispiel:  $\{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}\}$
- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

## Beispiel



○ Menge      ■ Vertreter

**union**( $a, f$ )

**union**( $c, d$ )

**find**( $a$ )  $\rightarrow a$

**find**( $f$ )  $\rightarrow a$

**find**( $c$ )  $\rightarrow c$

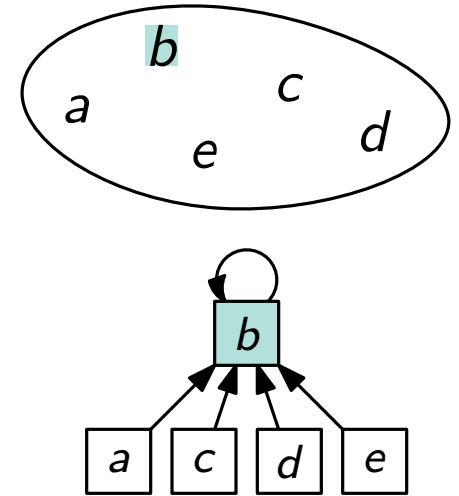
**union**( $b, d$ )

**find**( $c$ )  $\rightarrow b$

# Vorüberlegung: maximale Ordnung

## Fokus auf **find**: Was hätten wir gerne?

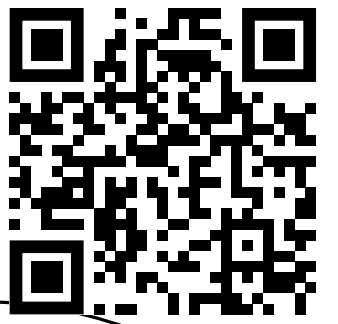
- optimal: jedes Element kennt den Vertreter der eigenen Menge
- Verzeigerte Struktur
  - jedes Element ist ein Knoten
  - jeder Knoten hat einen Zeiger zum Vertreter der Menge
- **find**: nur einen Zeiger verfolgen  $\rightarrow \Theta(1)$



### Erinnerung: Operationen

- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

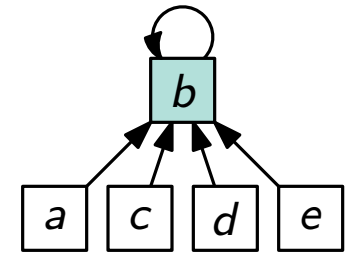
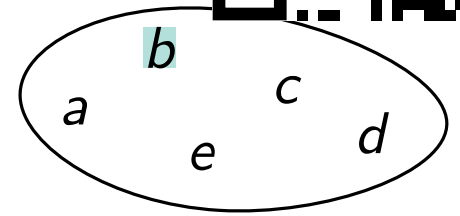




# Vorüberlegung: maximale Ordnung

## Fokus auf **find**: Was hätten wir gerne?

- optimal: jedes Element kennt den Vertreter der eigenen Menge
- Verzeigerte Struktur
  - jedes Element ist ein Knoten
  - jeder Knoten hat einen Zeiger zum Vertreter der Menge
- **find**: nur einen Zeiger verfolgen  $\rightarrow \Theta(1)$



Wie lange dauert **union**, wenn zwei Mengen der Größe  $n$  vereinigt werden?

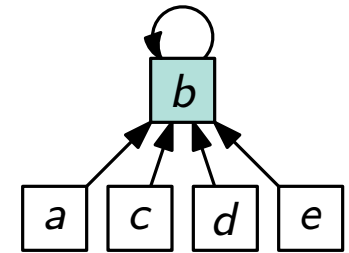
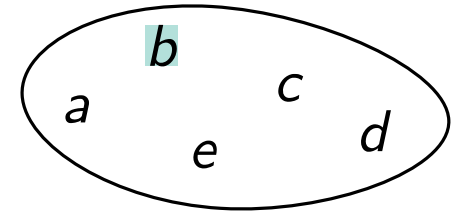
### Erinnerung: Operationen

- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

# Vorüberlegung: maximale Ordnung

## Fokus auf **find**: Was hätten wir gerne?

- optimal: jedes Element kennt den Vertreter der eigenen Menge
- Verzeigerte Struktur
  - jedes Element ist ein Knoten
  - jeder Knoten hat einen Zeiger zum Vertreter der Menge
- **find**: nur einen Zeiger verfolgen  $\rightarrow \Theta(1)$



## Problem für **union**

- alle Elemente aus einer der beiden Mengen müssen umgehängt werden  $\rightarrow \Theta(n)$
- maximale Ordnung aufrecht zu erhalten ist teuer

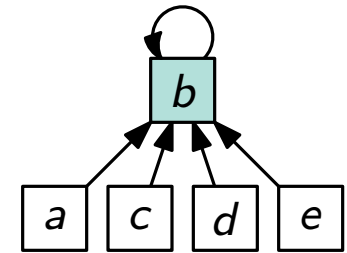
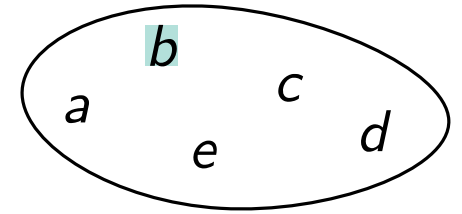
### Erinnerung: Operationen

- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

# Vorüberlegung: maximale Ordnung

## Fokus auf **find**: Was hätten wir gerne?

- optimal: jedes Element kennt den Vertreter der eigenen Menge
- Verzeigerte Struktur
  - jedes Element ist ein Knoten
  - jeder Knoten hat einen Zeiger zum Vertreter der Menge
- **find**: nur einen Zeiger verfolgen  $\rightarrow \Theta(1)$



## Problem für **union**

- alle Elemente aus einer der beiden Mengen müssen umgehängt werden  $\rightarrow \Theta(n)$
- maximale Ordnung aufrecht zu erhalten ist teuer

## Lösung: erlaube etwas Unordnung

- weniger Zeiger umhängen  $\rightarrow$  schnelles **union**
- kein direkter Zeiger zum Vertreter  $\rightarrow$  langsames **find**
- Ziel: genug Ordnung, dass **find** weiterhin schnell ist

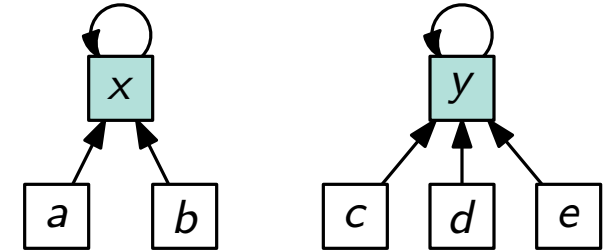
### Erinnerung: Operationen

- **union**( $x, y$ ): vereinige die Mengen, die  $x$  und  $y$  enthalten
- **find**( $x$ ): liefere einen eindeutigen Vertreter der Menge, die  $x$  enthält

# Schnelleres Union: erlaube etwas Unordnung

## Plan für schnelles union

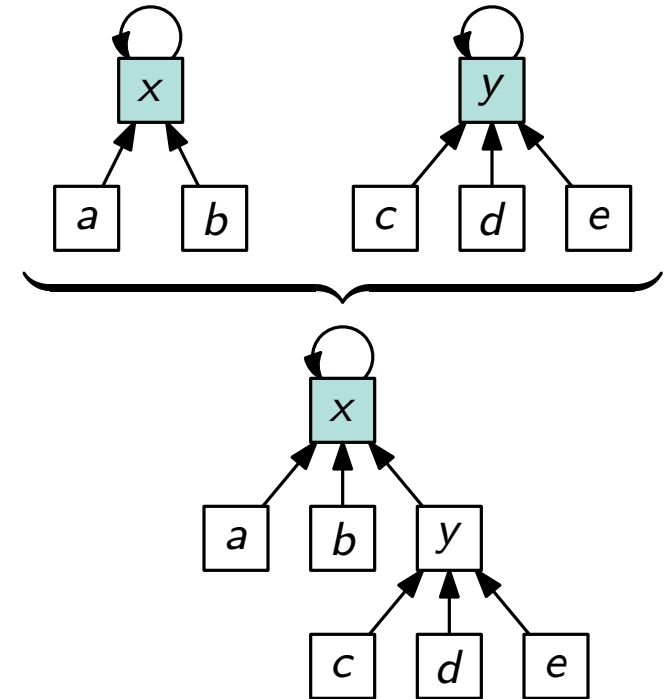
- Situation: vereinige  $X$  und  $Y$  mit Vertretern  $x \in X$  und  $y \in Y$



# Schnelleres Union: erlaube etwas Unordnung

## Plan für schnelles union

- Situation: vereinige  $X$  und  $Y$  mit Vertretern  $x \in X$  und  $y \in Y$
- wähle  $x$  als neuen Vertreter von  $X \cup Y$
- neuer Zeiger:  $y \rightarrow x$
- für alle anderen  $y' \in Y$ : behalte Zeiger zu  $y$



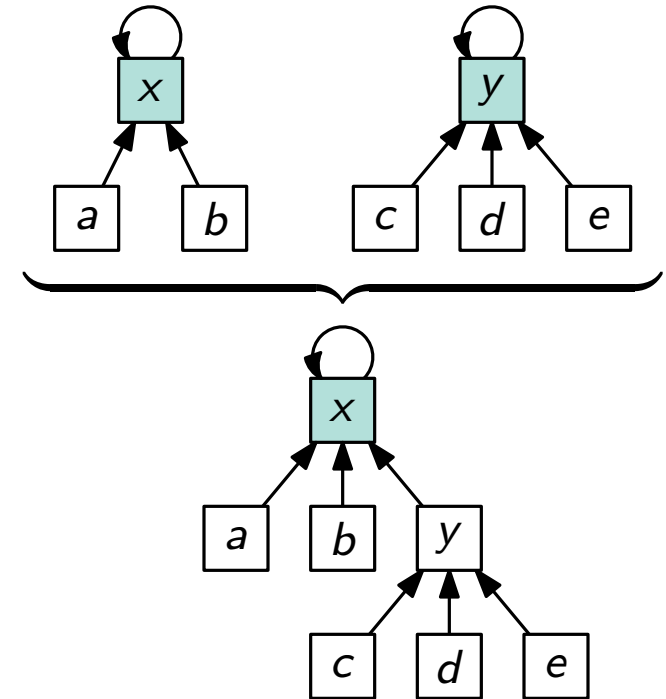
# Schnelleres Union: erlaube etwas Unordnung

## Plan für schnelles union

- Situation: vereinige  $X$  und  $Y$  mit Vertretern  $x \in X$  und  $y \in Y$
- wähle  $x$  als neuen Vertreter von  $X \cup Y$
- neuer Zeiger:  $y \rightarrow x$
- für alle anderen  $y' \in Y$ : behalte Zeiger zu  $y$

## Mehr Aufwand für find

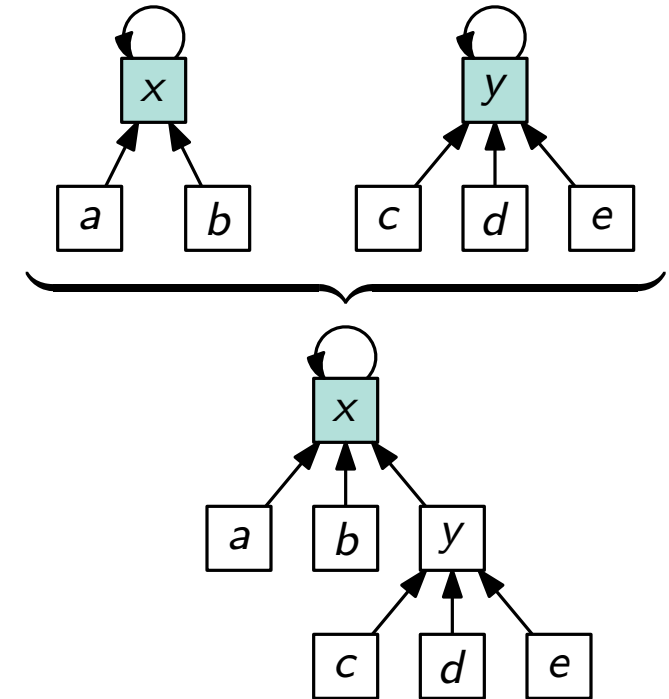
- verfolge ggf. mehrere Zeiger, um den Vertreter zu finden
- Hoffnung: die Pfade von Zeigern werden nicht so lang



# Schnelleres Union: erlaube etwas Unordnung

## Plan für schnelles union

- Situation: vereinige  $X$  und  $Y$  mit Vertretern  $x \in X$  und  $y \in Y$
- wähle  $x$  als neuen Vertreter von  $X \cup Y$
- neuer Zeiger:  $y \rightarrow x$
- für alle anderen  $y' \in Y$ : behalte Zeiger zu  $y$



## Mehr Aufwand für find

- verfolge ggf. mehrere Zeiger, um den Vertreter zu finden
- Hoffnung: die Pfade von Zeigern werden nicht so lang

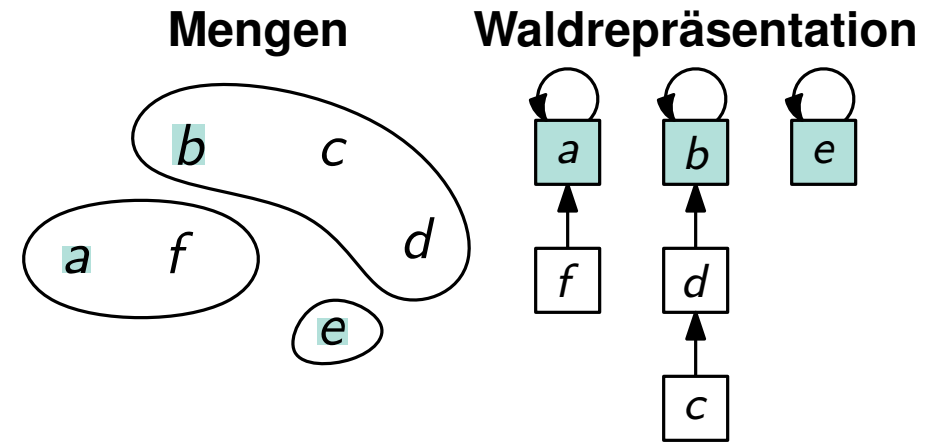
## Worst-Case Kosten und Baumhöhe

- resultierender Baum hat Höhe  $h \rightarrow$  Kosten  $O(h)$  pro Operation
- Plan: wähle neuen Vertreter, sodass Höhe möglichst klein

# Was genau machen wir jetzt?

## Repräsentation der Menge als gewurzelten Wald

- jeder Baum gehört zu einer Menge
- Wurzel jedes Baums ist der Vertreter der Menge





# Was genau machen wir jetzt?

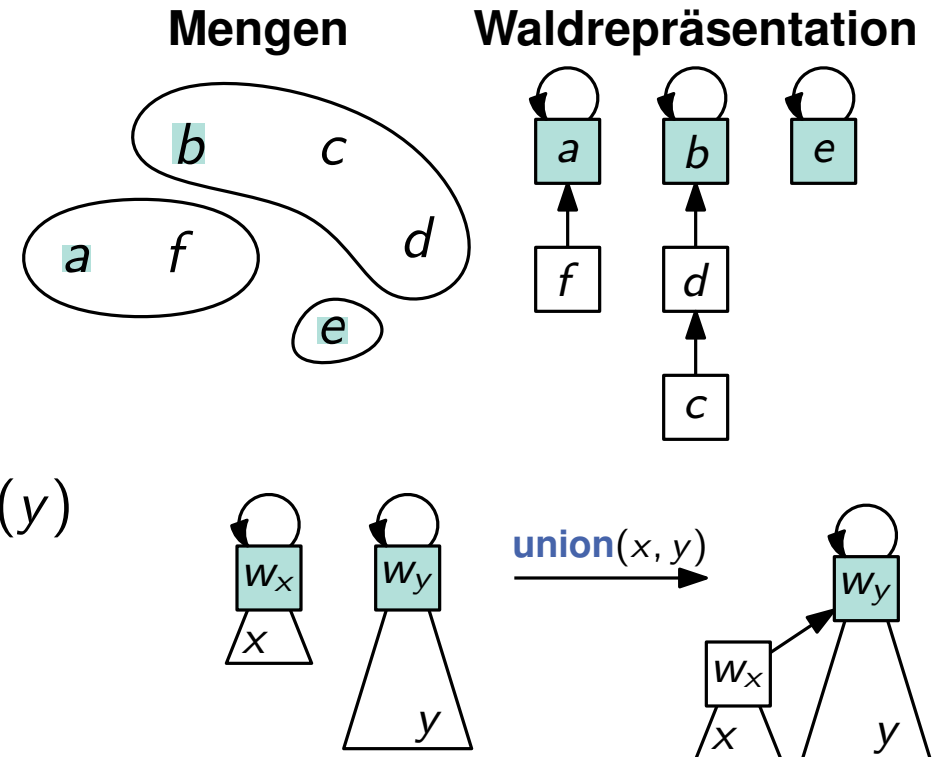
## Repräsentation der Menge als gewurzelten Wald

- jeder Baum gehört zu einer Menge
- Wurzel jedes Baums ist der Vertreter der Menge

**find(x)**: verfolge Zeiger zum Elter bis zur Wurzel

## union(x, y)

- finde zunächst Wurzeln:  $w_x = \text{find}(x)$  und  $w_y = \text{find}(y)$
- seien  $h_x$  und  $h_y$  die Höhen der Bäume
- o.B.d.A.  $h_x \leq h_y \rightarrow$  füge  $w_x$  als Kind von  $w_y$  ein



# Was genau machen wir jetzt?

## Repräsentation der Menge als gewurzelten Wald

- jeder Baum gehört zu einer Menge
- Wurzel jedes Baums ist der Vertreter der Menge

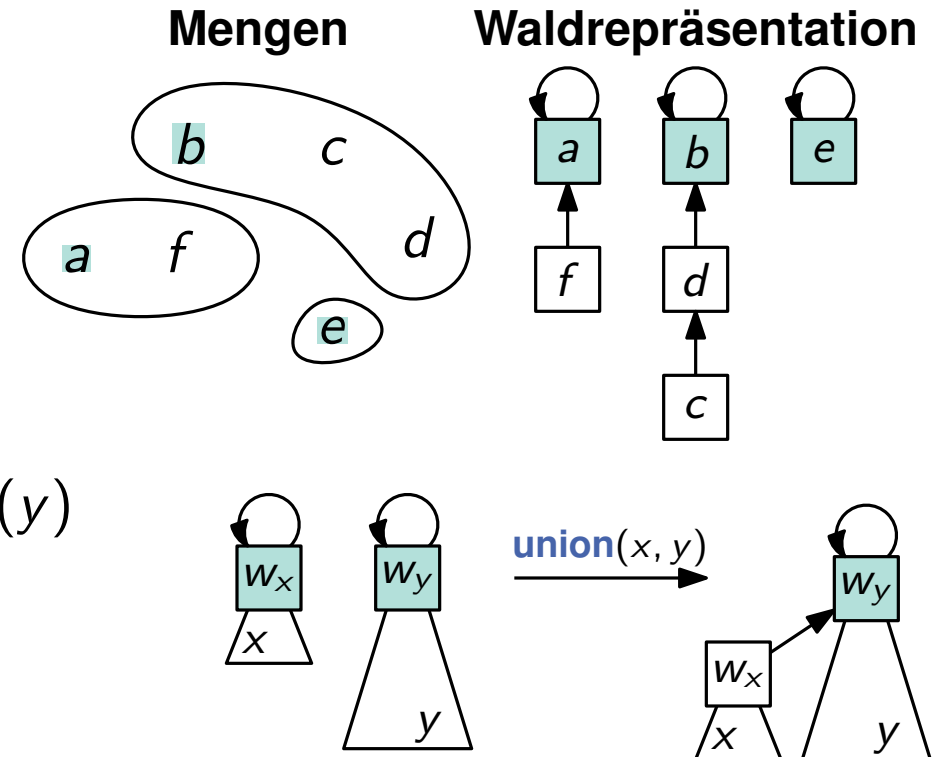
**find**( $x$ ): verfolge Zeiger zum Elter bis zur Wurzel

## **union**( $x, y$ )

- finde zunächst Wurzeln:  $w_x = \mathbf{find}(x)$  und  $w_y = \mathbf{find}(y)$
- seien  $h_x$  und  $h_y$  die Höhen der Bäume
- o.B.d.A.  $h_x \leq h_y \rightarrow$  füge  $w_x$  als Kind von  $w_y$  ein

## Anmerkungen

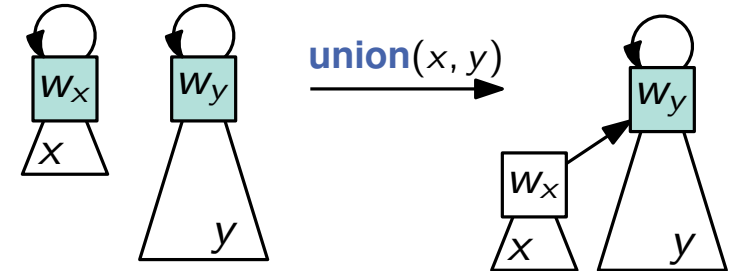
- speichere Höhe des Baums an der Wurzel  $\rightarrow$  ggf. update bei **union**
- Laufzeit: dominiert durch die Höhe bei **find**



# Höhe des Baumes

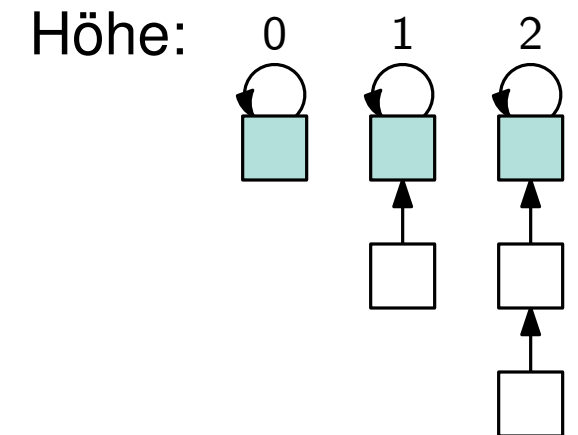
## union(x, y)

- finde zunächst Wurzeln:  $w_x = \text{find}(x)$  und  $w_y = \text{find}(y)$
- seien  $h_x$  und  $h_y$  die Höhen der Bäume
- o.B.d.A.  $h_x \leq h_y \rightarrow$  füge  $w_x$  als Kind von  $w_y$  ein



### Lemma

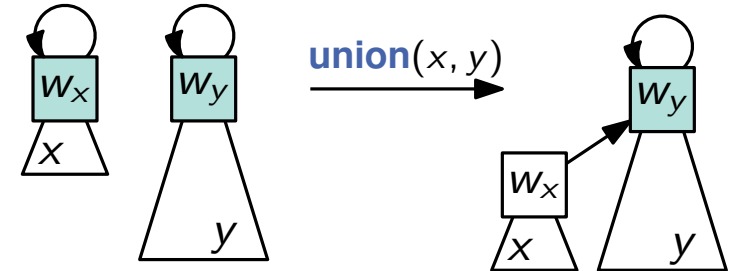
Ein solcher Baum der Höhe  $h$  enthält mindestens  $2^h$  Knoten. Damit gilt  $h \in O(\log n)$ .



# Höhe des Baumes

## union(x, y)

- finde zunächst Wurzeln:  $w_x = \text{find}(x)$  und  $w_y = \text{find}(y)$
- seien  $h_x$  und  $h_y$  die Höhen der Bäume
- o.B.d.A.  $h_x \leq h_y \rightarrow$  füge  $w_x$  als Kind von  $w_y$  ein

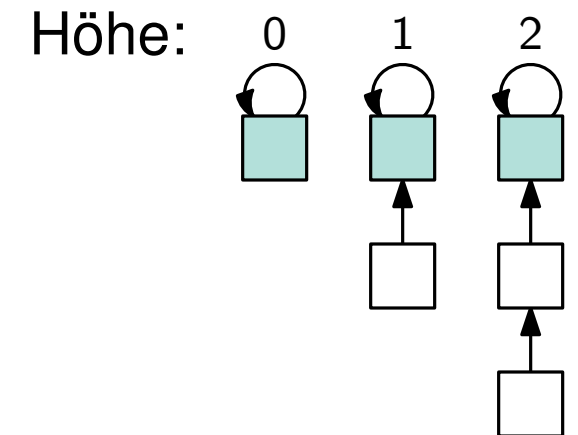


### Lemma

Ein solcher Baum der Höhe  $h$  enthält mindestens  $2^h$  Knoten. Damit gilt  $h \in O(\log n)$ .

### Beweis: Induktion über $h$

- Anfang:  $h = 0 \Rightarrow 1 = 2^0$  Knoten
- Höhe  $h + 1$  entsteht nur durch **union** zweier Bäume der Höhe  $h$
- nach Induktionsvoraussetzung: mindestens  $2 \cdot 2^h = 2^{h+1}$  Knoten



# Geht es besser?

## Theorem

Es gibt eine Union–Find Datenstruktur, bei der **union** und **find** je  $O(\log n)$  Zeit benötigen.

# Geht es besser?

## Theorem

Es gibt eine Union–Find Datenstruktur, bei der **union** und **find** je  $O(\log n)$  Zeit benötigen.

## Idee

- Höhe verringern durch gelegentliches Aufräumen
- optimal: alle Knoten sind direkte Kinder der Wurzel
- Aufräumen  $\equiv$  Knoten direkt an die Wurzel hängen

# Geht es besser?

## Theorem

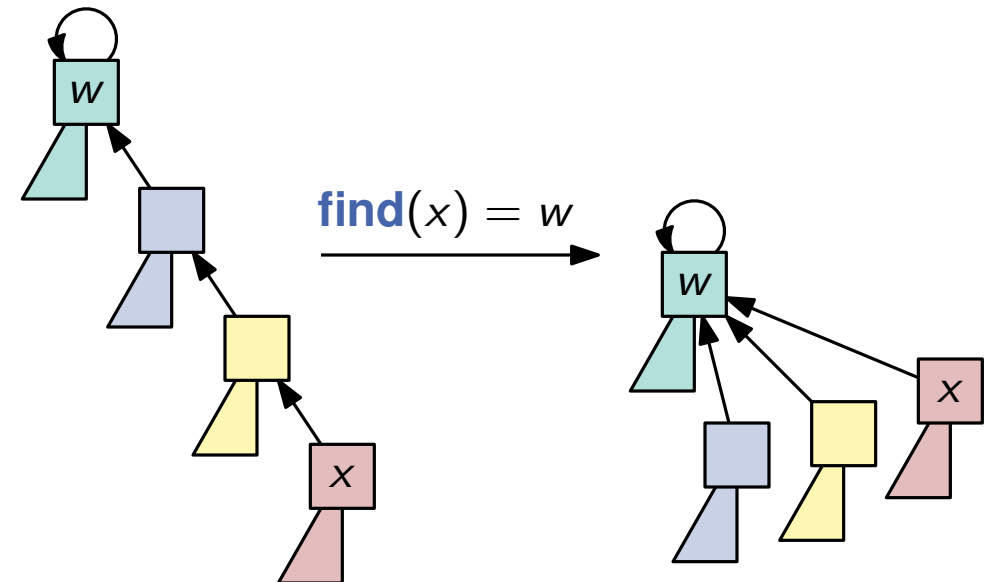
Es gibt eine Union–Find Datenstruktur, bei der **union** und **find** je  $O(\log n)$  Zeit benötigen.

## Idee

- Höhe verringern durch gelegentliches Aufräumen
- optimal: alle Knoten sind direkte Kinder der Wurzel
- Aufräumen  $\equiv$  Knoten direkt an die Wurzel hängen

## **find**( $x$ ) mit Pfadkompression

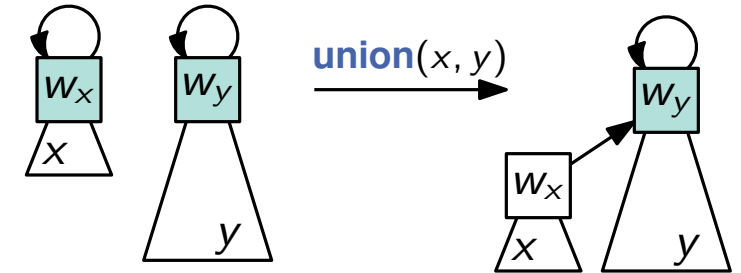
- sei  $\pi$  der Pfad von  $x$  zur Wurzel  $w$
- hänge jeden Knoten auf  $\pi$  direkt an  $w$
- asymptotisch keine zusätzlichen Kosten für **find**( $x$ )
- aber: spätere Aufrufe von **find** ggf. schneller



# Union by Rank

## Problem

- Pfadkompression verringert ggf. die Höhe des Baumes, ohne dass wir es merken
- **union** nutzt die Höhe um zu entscheiden, welcher Baum unter welchen gehängt wird

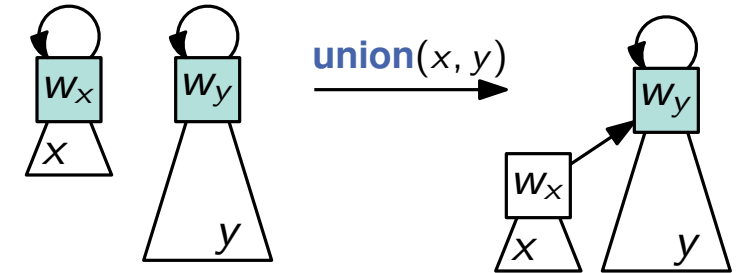




# Union by Rank

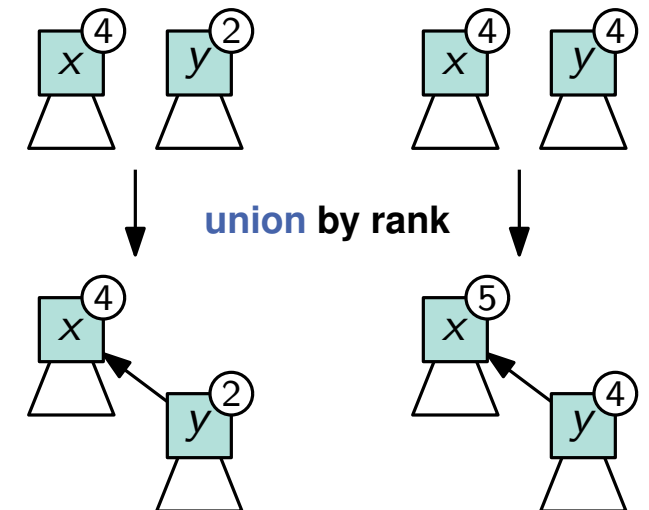
## Problem

- Pfadkompression verringert ggf. die Höhe des Baumes, ohne dass wir es merken
- **union** nutzt die Höhe um zu entscheiden, welcher Baum unter welchen gehängt wird



## Gar nicht schlimm

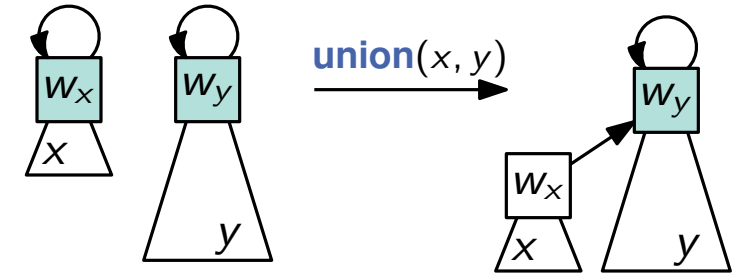
- nenne die gespeicherte Zahl nicht Höhe, sondern **Rang**  $r$
- alles wie bisher bei der Höhe:
  - initial: Rang  $r = 0$  für Mengen der Größe 1
  - **union**: kleiner Rang unter großen Rang
  - bei gleichem Rang: erhöhe einen um 1



# Union by Rank

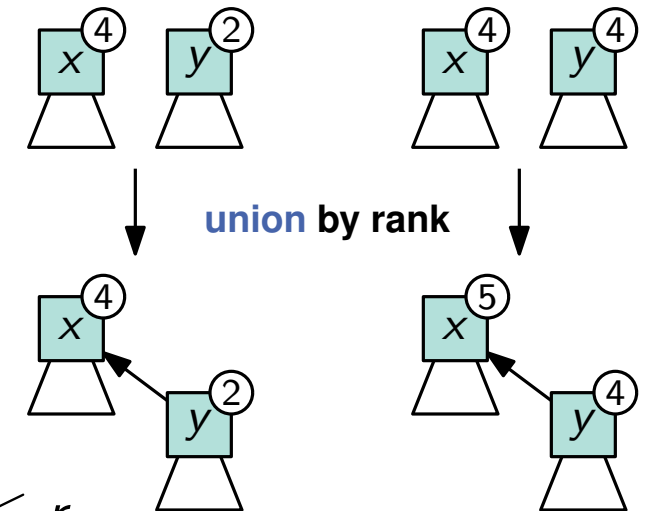
## Problem

- Pfadkompression verringert ggf. die Höhe des Baumes, ohne dass wir es merken
- **union** nutzt die Höhe um zu entscheiden, welcher Baum unter welchen gehängt wird



## Gar nicht schlimm

- nenne die gespeicherte Zahl nicht Höhe, sondern **Rang**  $r$
- alles wie bisher bei der Höhe:
  - initial: Rang  $r = 0$  für Mengen der Größe 1
  - **union**: kleiner Rang unter großen Rang
  - bei gleichem Rang: erhöhe einen um 1
- weiterhin: Teilbaum unter Wurzel mit Rang  $r$  hat  $\geq 2^r$  Knoten
- außerdem: Teilbaum unter einem Knoten mit Rang  $r$  hat Höhe  $\leq r$



# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

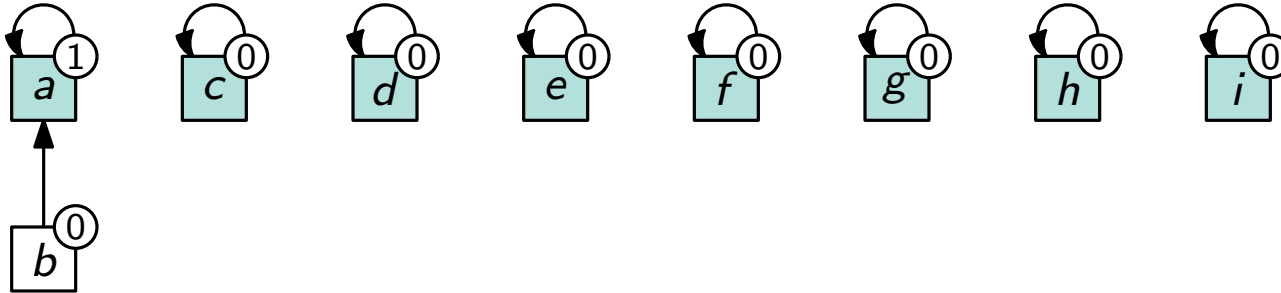
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

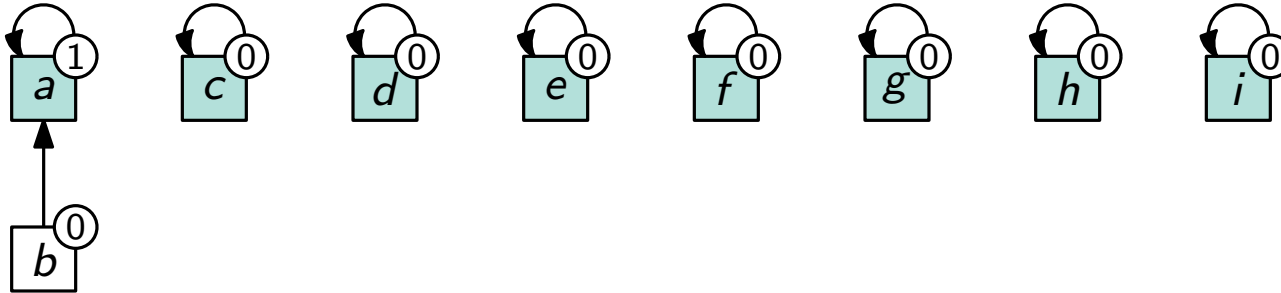
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

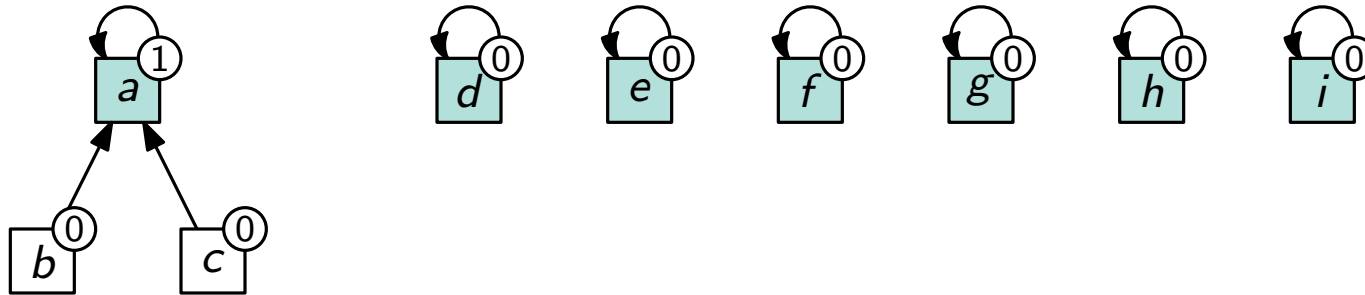
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

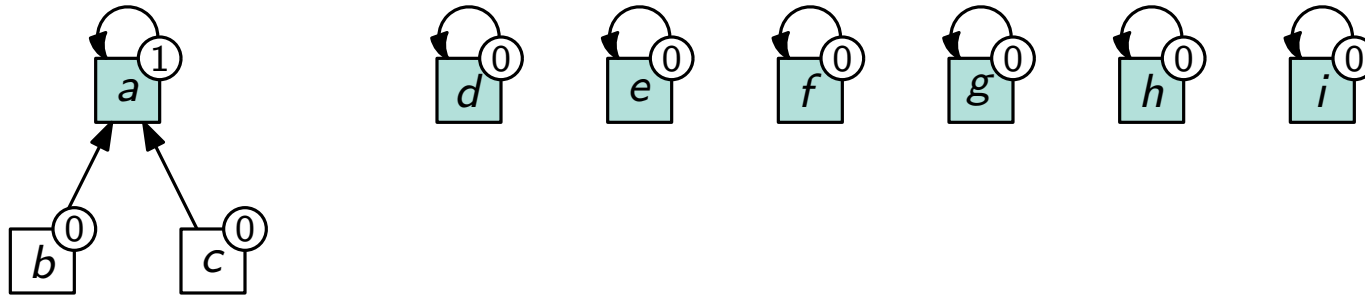
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

**union**( $a, f$ )

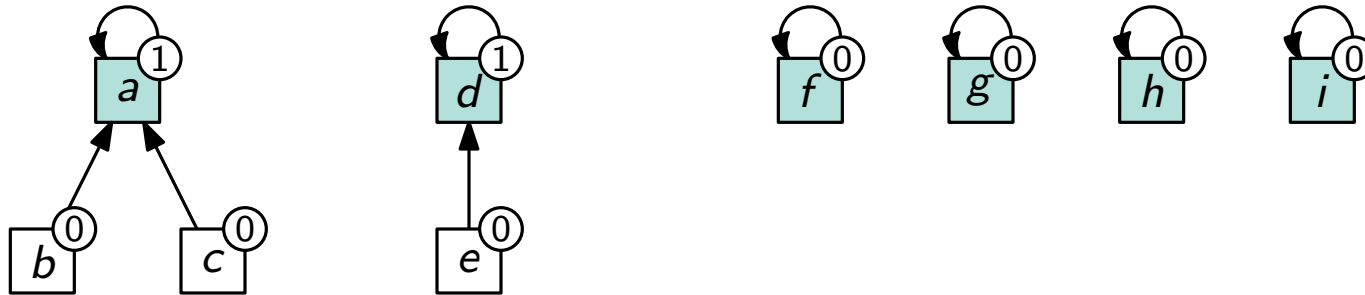
**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel



# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

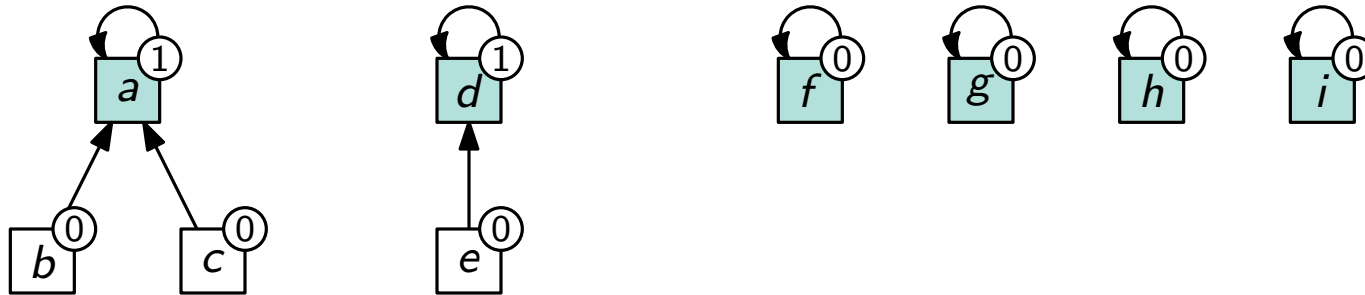
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

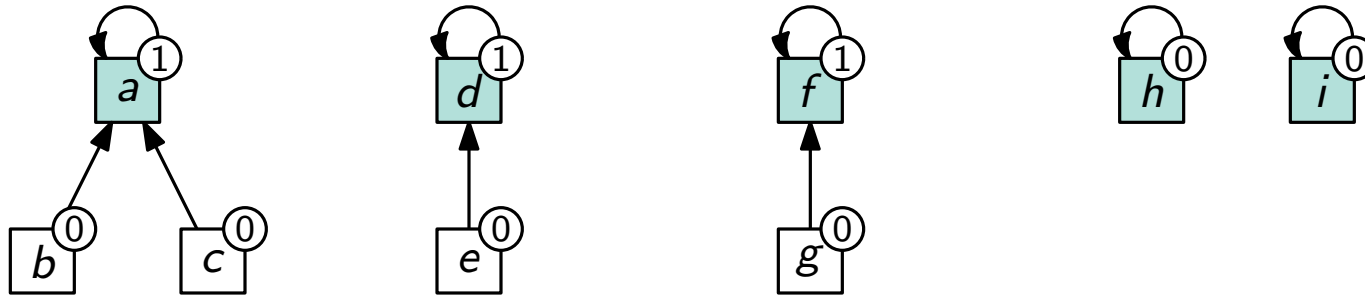
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

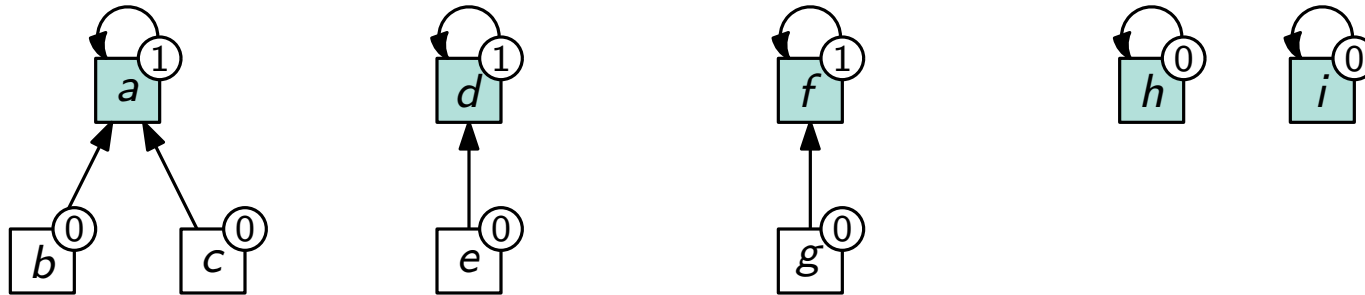
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**(a, b)

**union**(b, c)

**union**(d, e)

**union**(f, g)

**union**(h, i)

**union**(a, d)

**union**(f, h)

**find**(e)

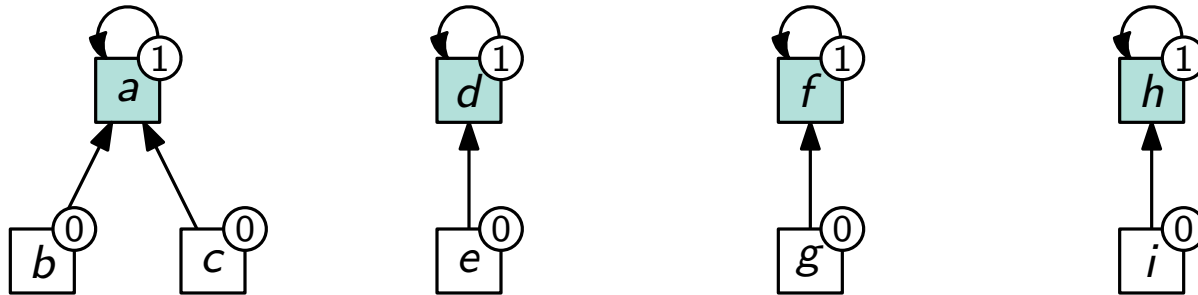
**union**(a, f)

**find**(h)

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

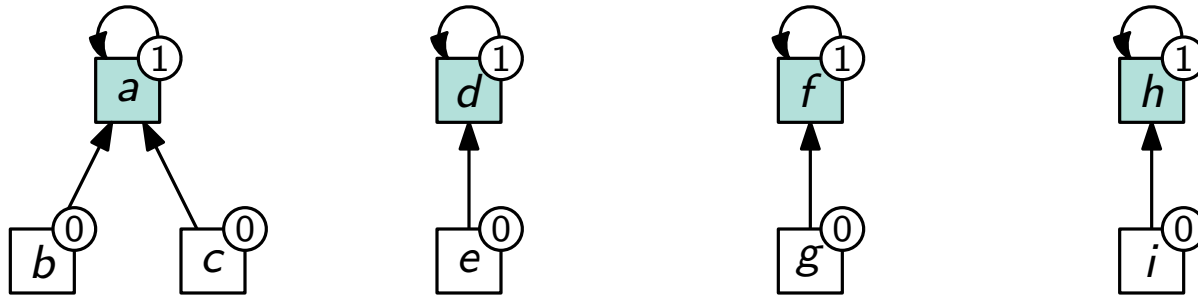
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

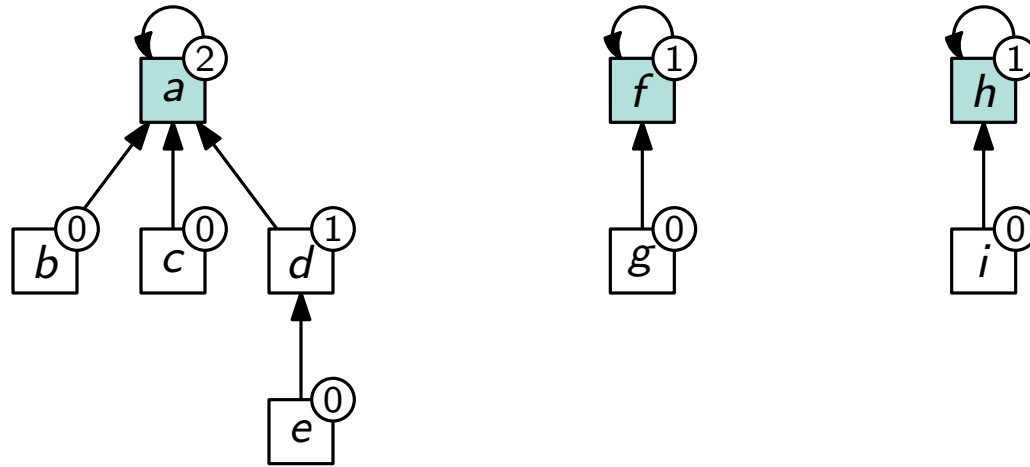
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

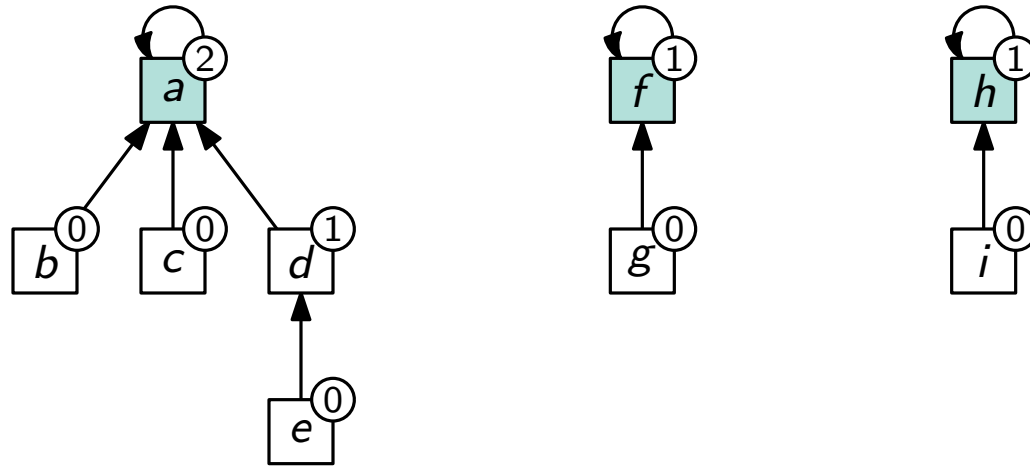
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

**union**( $a, f$ )

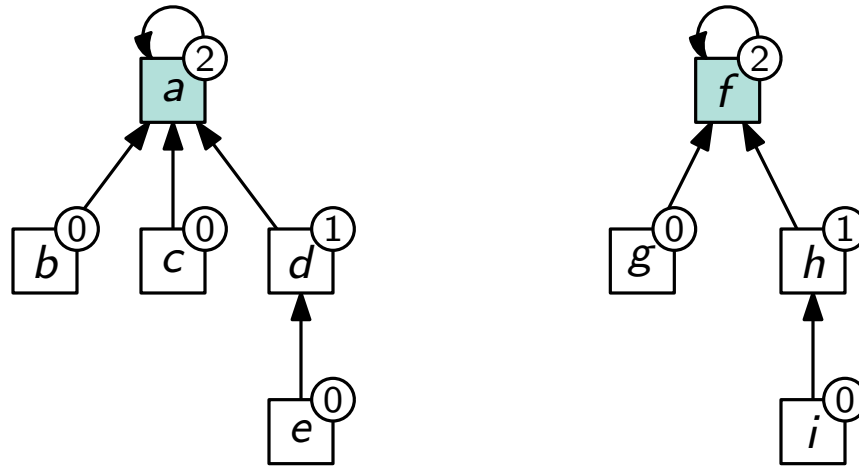
**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel



# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

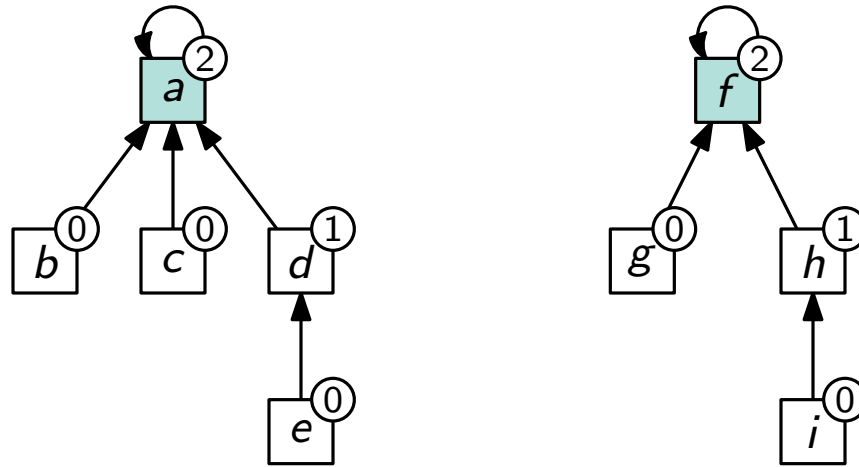
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

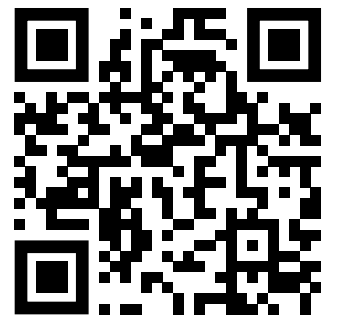
**find**( $e$ )

**union**( $a, f$ )

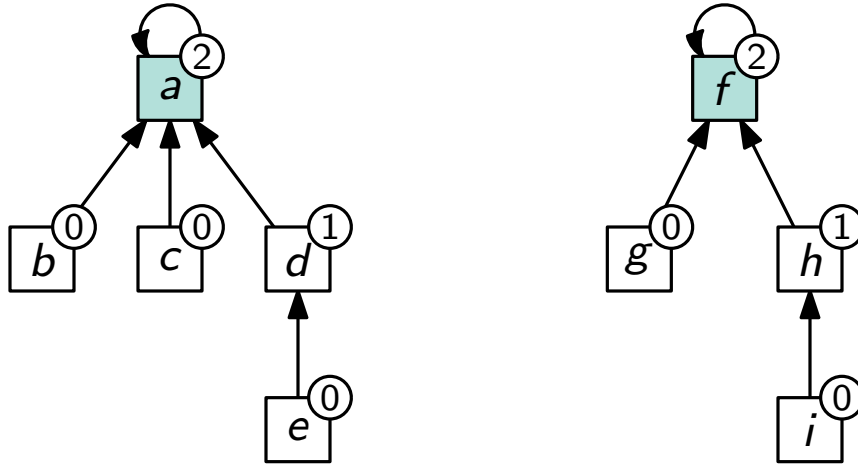
**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel



# Beispiel



Welchen Zustand hat die Datenstruktur nach Anwendung der drei übrigen Operationen?

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

## Operationen

**union**(a, b)

**union**(b, c)

**union**(d, e)

**union**(f, g)

**union**(h, i)

**union**(a, d)

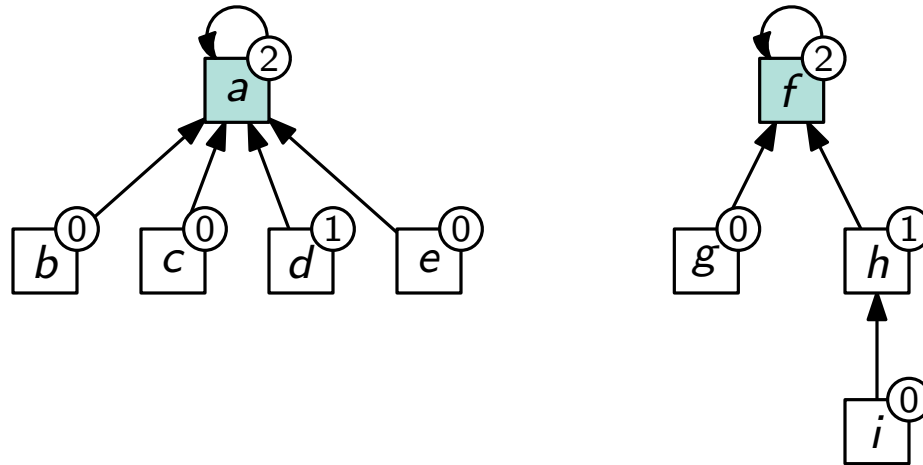
**union**(f, h)

**find**(e)

**union**(a, f)

**find**(h)

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

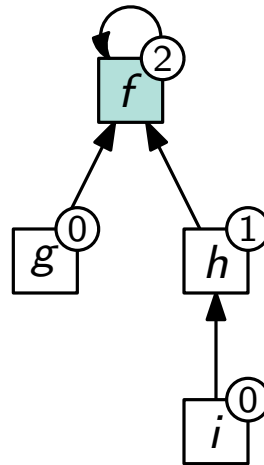
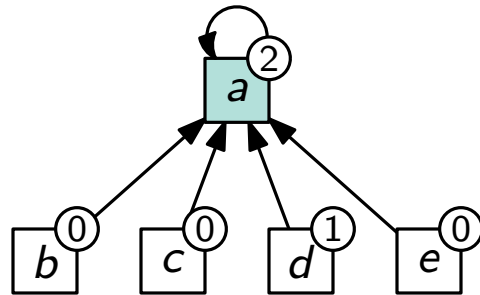
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**(a, b)

**union**(b, c)

**union**(d, e)

**union**(f, g)

**union**(h, i)

**union**(a, d)

**union**(f, h)

**find**(e)

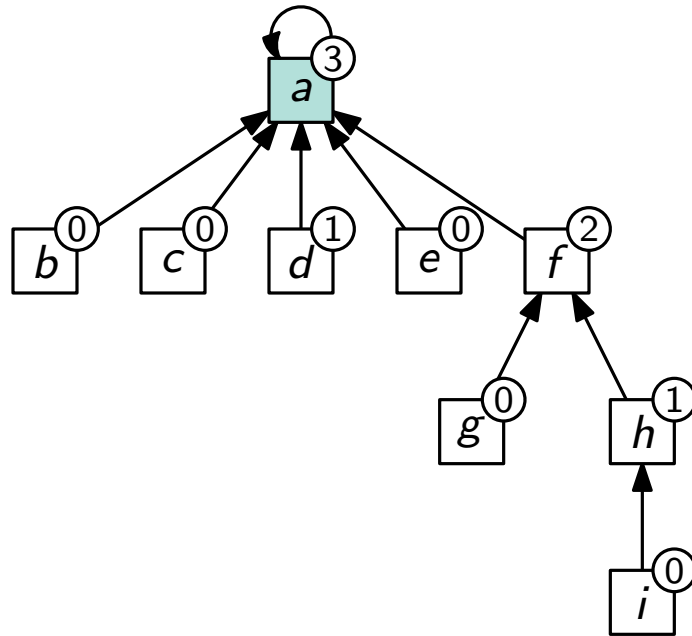
**union**(a, f)

**find**(h)

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

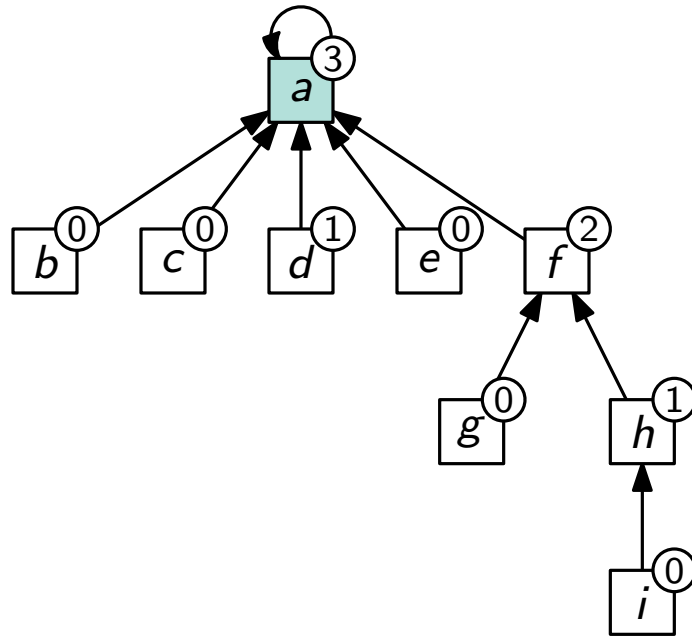
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**( $a, b$ )

**union**( $b, c$ )

**union**( $d, e$ )

**union**( $f, g$ )

**union**( $h, i$ )

**union**( $a, d$ )

**union**( $f, h$ )

**find**( $e$ )

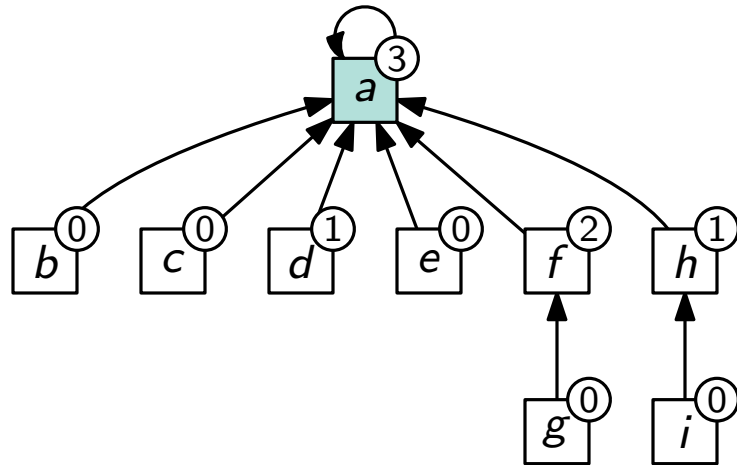
**union**( $a, f$ )

**find**( $h$ )

## Tiebreaker

- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel

# Beispiel



## Operationen

**union**(a, b)

**union**(b, c)

**union**(d, e)

**union**(f, g)

**union**(h, i)

**union**(a, d)

**union**(f, h)

**find**(e)

**union**(a, f)

**find**(h)

## Tiebreaker

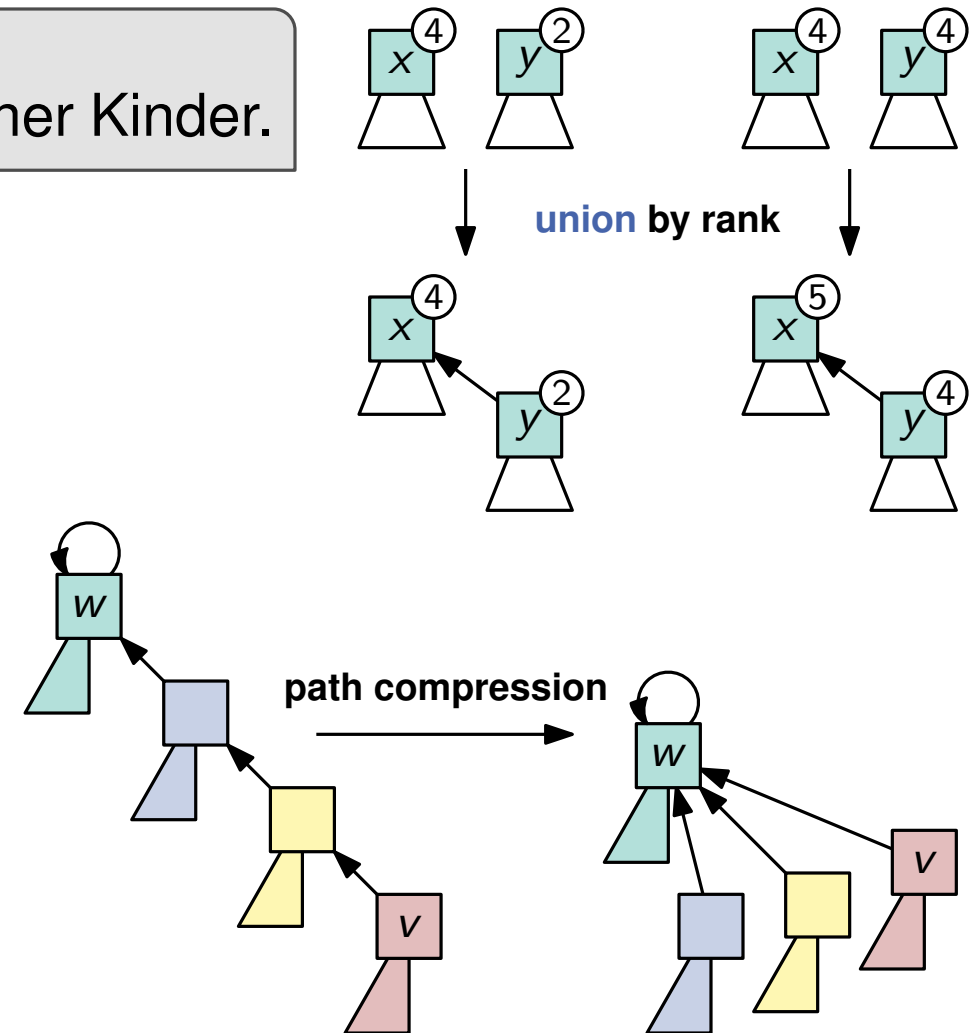
- **union** zwei Wurzeln mit gleichem Rang
- wähle alphabetisch kleineren Knoten als Wurzel



# Laufzeitanalyse: Grundlegende Invarianten

## Lemma

Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.



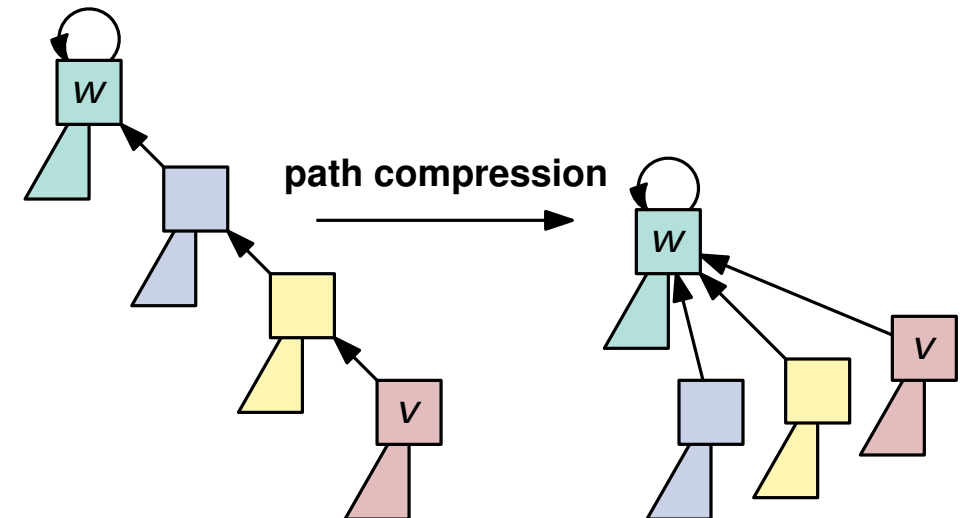
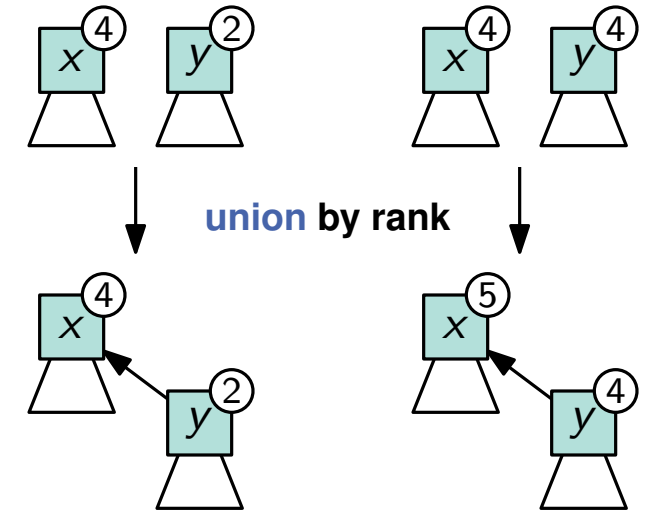
# Laufzeitanalyse: Grundlegende Invarianten

## Lemma

Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

## Beweis

- neue Eltern–Kind Beziehungen entstehen nur durch **union**
- der Rang des Elter ist dann immer größer als der des Kindes
- Pfadkompression: Rang des neuen Elters größer als des alten



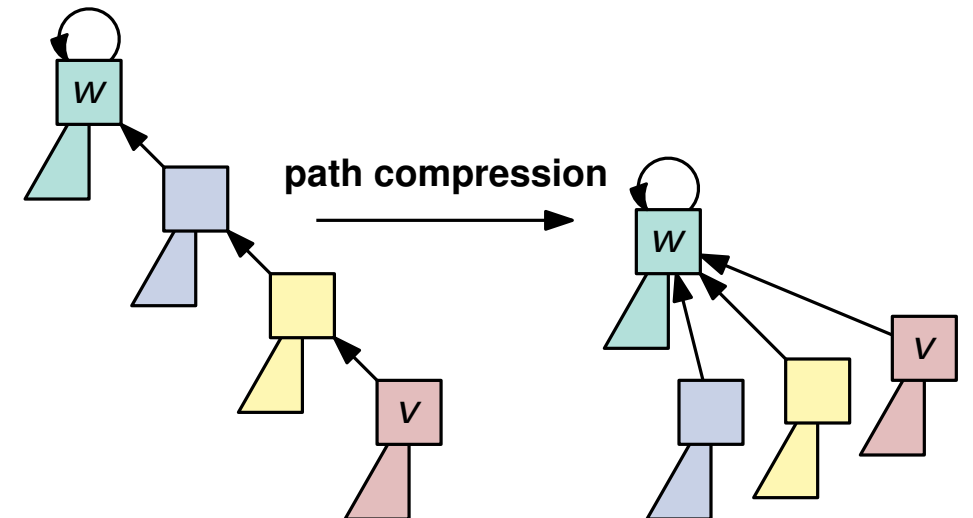
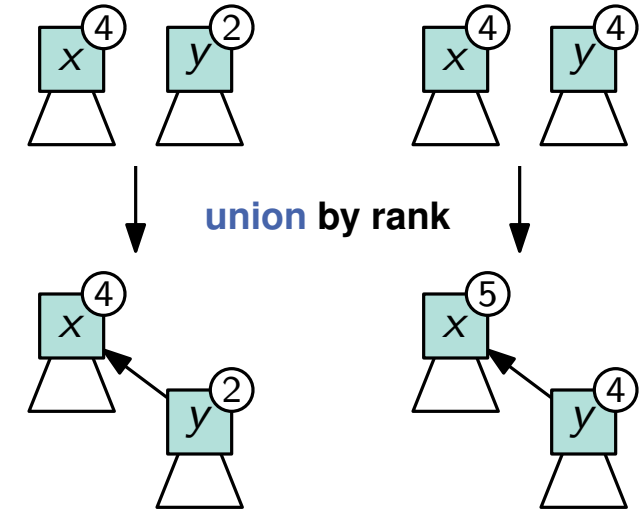
# Laufzeitanalyse: Grundlegende Invarianten

## Lemma

Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

## Lemma

Es gibt höchstens  $n/2^r$  Knoten mit Rang gleich  $r$ .



# Laufzeitanalyse: Grundlegende Invarianten

## Lemma

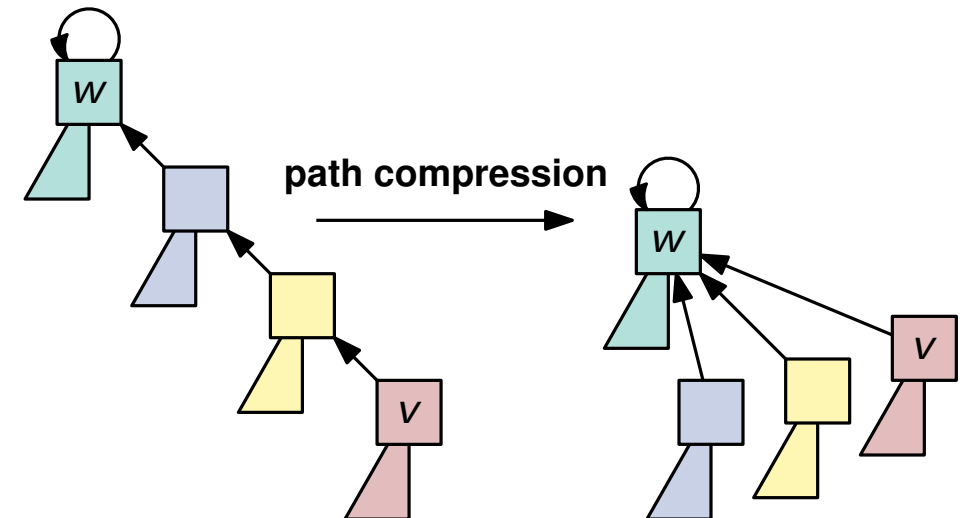
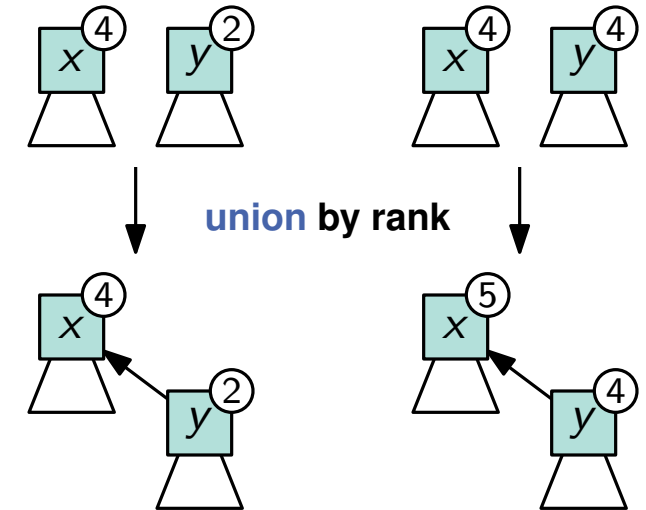
Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

## Lemma

Es gibt höchstens  $n/2^r$  Knoten mit Rang gleich  $r$ .

## Beweis

- beachte: Ränge ändern sich nur durch **union**
- zeige mittels Induktion über  $r$ : Anzahl Knoten unter Baum mit Rang  $r$  ist mindestens  $2^r$   
(geht ganz genau so, wie vorhin bei der Höhe)
- außerdem: Teilbäume unter Knoten mit dem selben Rang  $r$  sind disjunkt



# Laufzeitanalyse: Grundlegende Invarianten

## Lemma

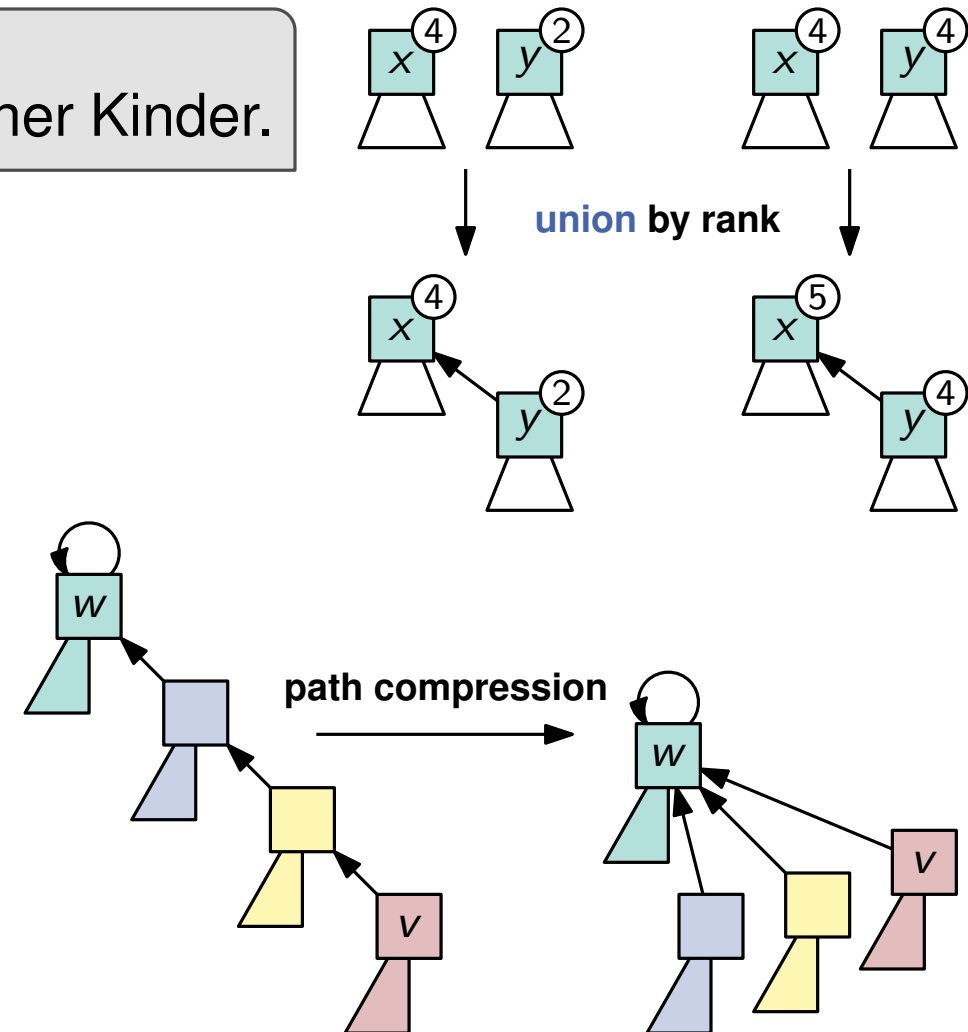
Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

## Lemma

Es gibt höchstens  $n/2^r$  Knoten mit Rang gleich  $r$ .

## Lemma

Es gibt höchstens  $n/2^r$  Knoten mit Rang größer  $r$ .



# Laufzeitanalyse: Grundlegende Invarianten

## Lemma

Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

## Lemma

Es gibt höchstens  $n/2^r$  Knoten mit Rang gleich  $r$ .

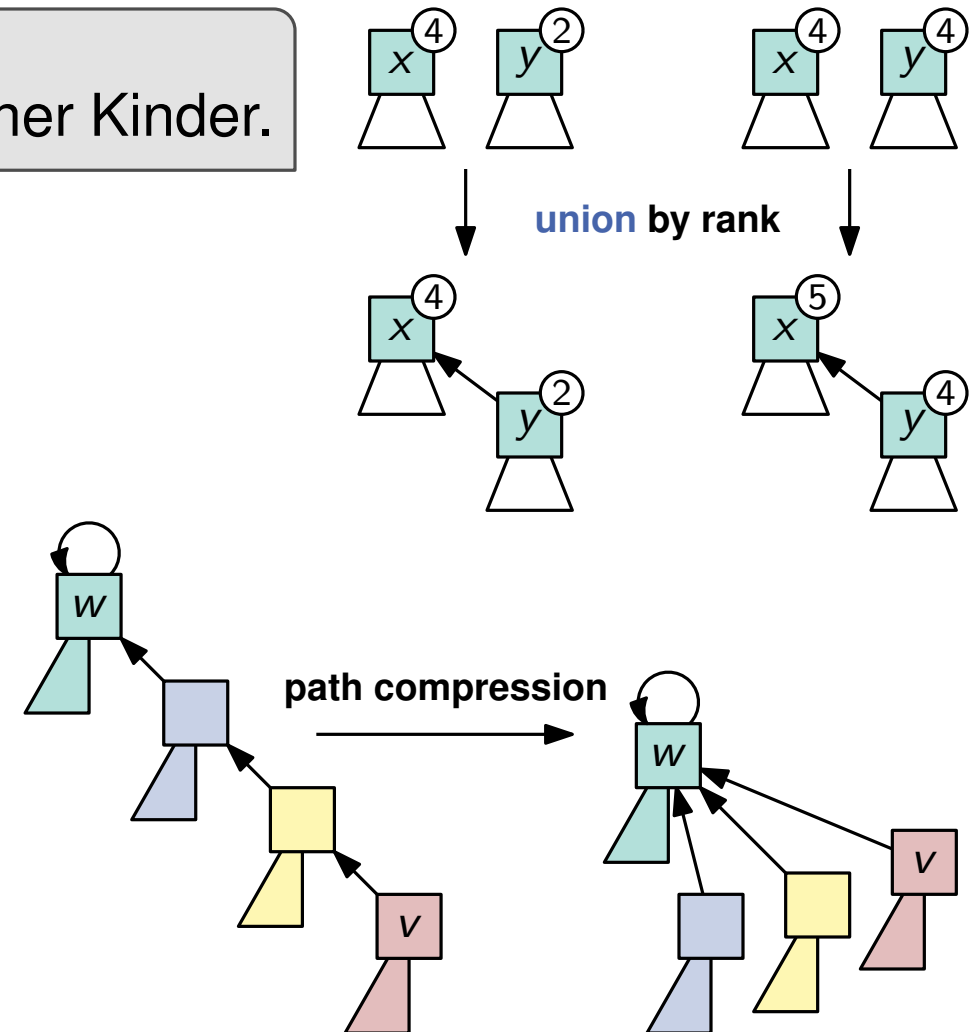
## Lemma

Es gibt höchstens  $n/2^r$  Knoten mit Rang größer  $r$ .

## Beweis

- bilde Summe über alle größeren Ränge:

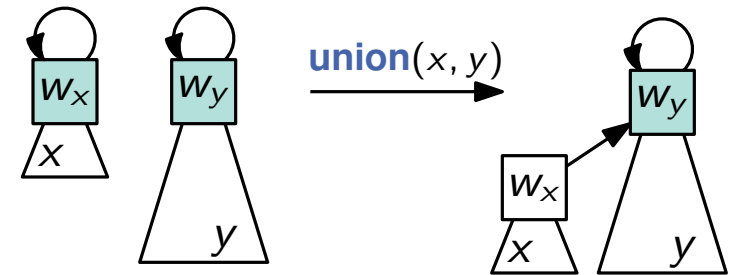
$$\sum_{i>r} \frac{n}{2^i} \leq 2 \cdot \frac{n}{2^{r+1}} = \frac{n}{2^r}$$



# Laufzeitanalyse: neue Operation **link**

## Die drei Schritte von **union**( $x, y$ )

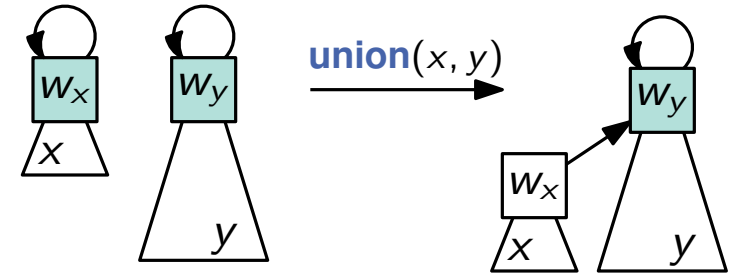
- $w_x = \mathbf{find}(x)$
- $w_y = \mathbf{find}(y)$
- hänge  $w_x$  unter  $w_y$  oder umgekehrt



# Laufzeitanalyse: neue Operation **link**

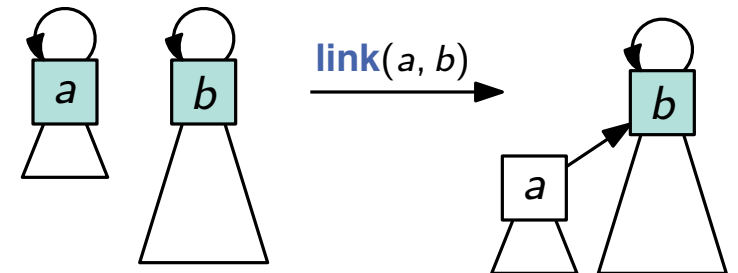
## Die drei Schritte von **union**( $x, y$ )

- $w_x = \mathbf{find}(x)$
- $w_y = \mathbf{find}(y)$
- hänge  $w_x$  unter  $w_y$  oder umgekehrt



## Neue Operation: **link**

- genauso wie **union**, aber nur für Wurzeln
- die beiden **find** Operationen fallen dann weg

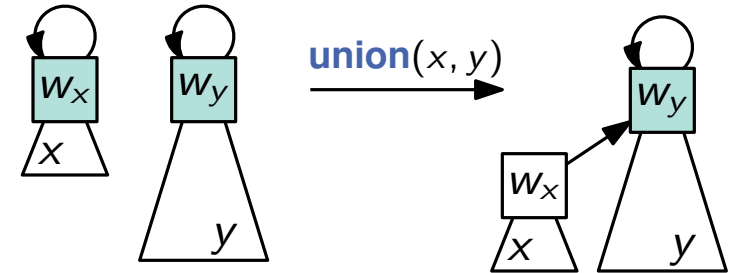




# Laufzeitanalyse: neue Operation **link**

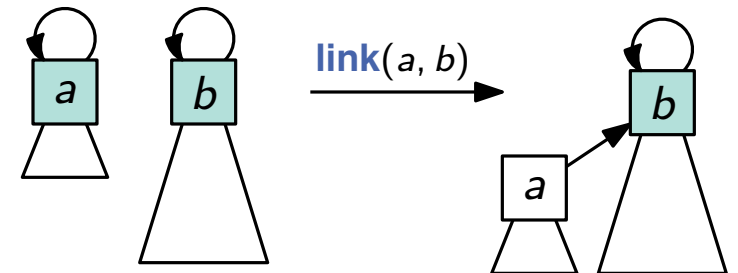
## Die drei Schritte von **union**( $x, y$ )

- $w_x = \mathbf{find}(x)$
- $w_y = \mathbf{find}(y)$
- hänge  $w_x$  unter  $w_y$  oder umgekehrt



## Neue Operation: **link**

- genauso wie **union**, aber nur für Wurzeln
- die beiden **find** Operationen fallen dann weg



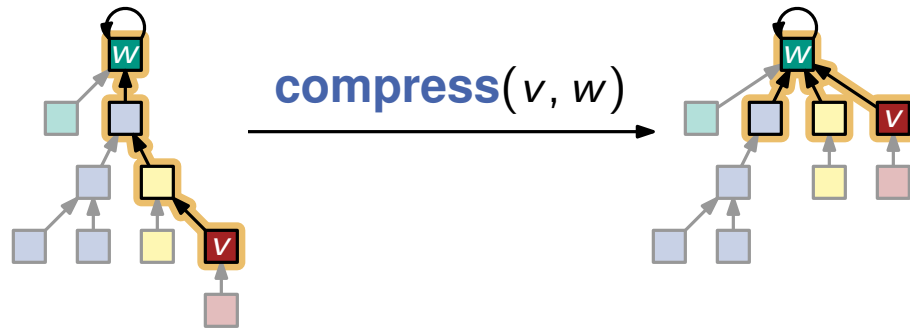
## Warum machen wir das?

- Plan: analysiere Kosten für **union** und **find** getrennt voneinander
- Problem: lässt sich kaum trennen, wenn jedes **union** zwei Mal **find** ausführt
- Lösung: ersetze Sequenz von  $n$  **union** und  $m$  **find** durch  $n$  **link** und  $2n + m$  **find**

# Laufzeitanalyse: neue Operation **compress**

## Die zwei Schritte von **find**( $v$ )

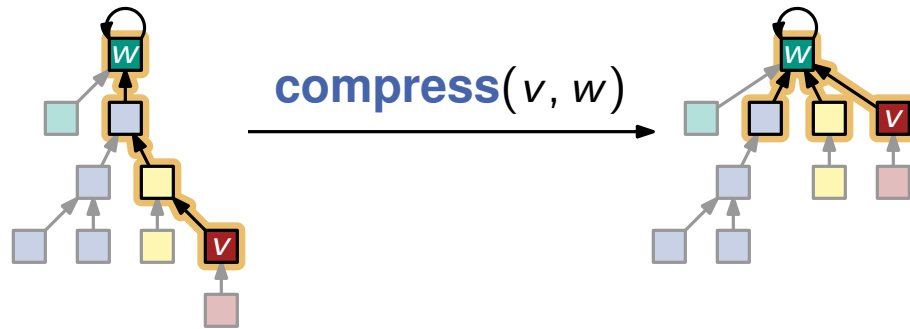
- Finden der Wurzel  $w$
- Kompression des Pfades von  $v$  nach  $w$   $\rightarrow$  bezeichne diesen Schritt mit **compress**( $v, w$ )



# Laufzeitanalyse: neue Operation **compress**

## Die zwei Schritte von **find**( $v$ )

- Finden der Wurzel  $w$
- Kompression des Pfades von  $v$  nach  $w$   $\rightarrow$  bezeichne diesen Schritt mit **compress**( $v, w$ )



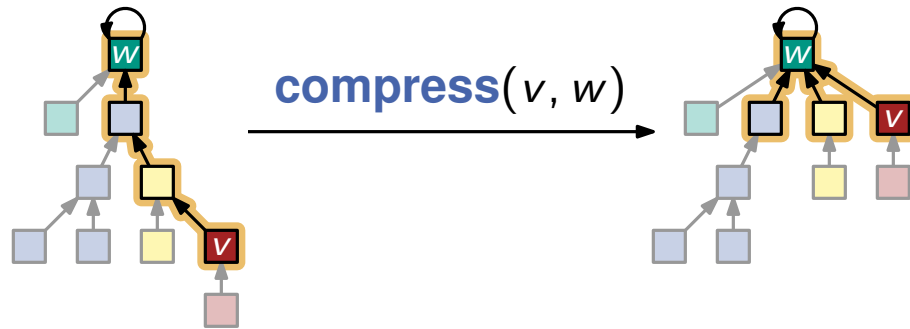
## Betrachte **compress** statt **find**

- ersetze in Folge von **link** und **find** Operationen jedes **find** durch passendes **compress**
- beachte: resultierende Folge liefert zu jedem Zeitpunkt die selbe Datenstruktur
- außerdem: asymptotische Laufzeit bleibt gleich

# Laufzeitanalyse: neue Operation **compress**

## Die zwei Schritte von **find**( $v$ )

- Finden der Wurzel  $w$
- Kompression des Pfades von  $v$  nach  $w$   $\rightarrow$  bezeichne diesen Schritt mit **compress**( $v, w$ )



### Grundsätzliches Vorgehen

- Ziel: Analysiere Laufzeit von Algo  $A$
- analysiere stattdessen einen anderen Algo  $B$
- zeige: Laufzeit von  $A$  ist nicht größer als die von  $B$
- damit: Laufzeit von  $B$  gibt Schranke an Laufzeit von  $A$

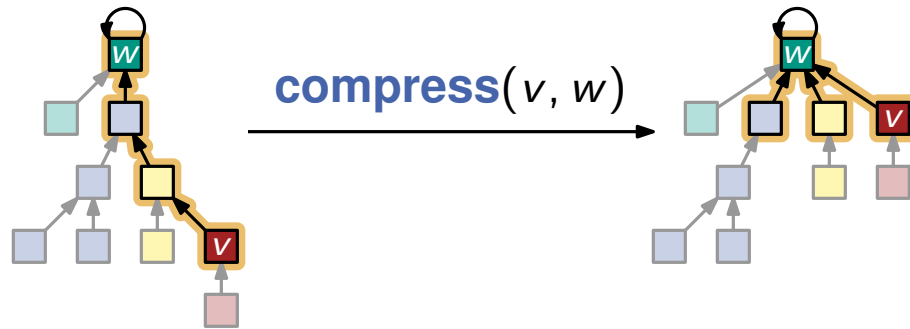
## Betrachte **compress** statt **find**

- ersetze in Folge von **link** und **find** Operationen jedes **find** durch passendes **compress**
- beachte: resultierende Folge liefert zu jedem Zeitpunkt die selbe Datenstruktur
- außerdem: asymptotische Laufzeit bleibt gleich

# Laufzeitanalyse: neue Operation **compress**

## Die zwei Schritte von **find**( $v$ )

- Finden der Wurzel  $w$
- Kompression des Pfades von  $v$  nach  $w$   $\rightarrow$  bezeichne diesen Schritt mit **compress**( $v, w$ )



### Grundsätzliches Vorgehen

- Ziel: Analysiere Laufzeit von Algo  $A$
- analysiere stattdessen einen anderen Algo  $B$
- zeige: Laufzeit von  $A$  ist nicht größer als die von  $B$
- damit: Laufzeit von  $B$  gibt Schranke an Laufzeit von  $A$

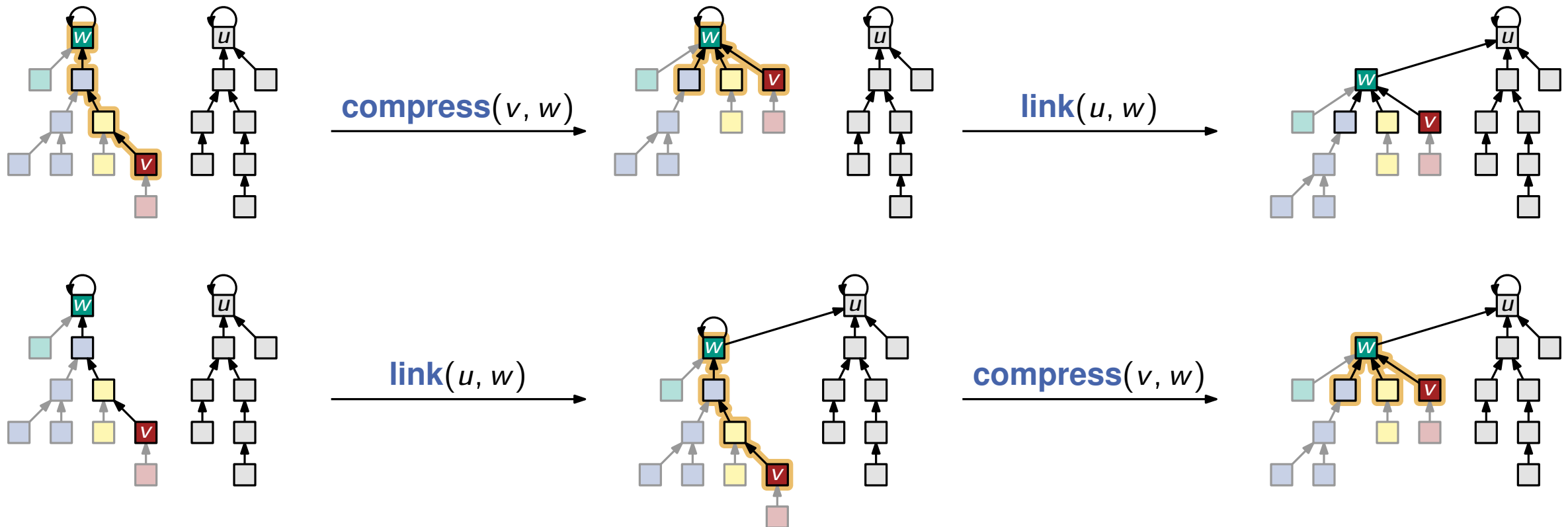
## Betrachte **compress** statt **find**

- ersetze in Folge von **link** und **find** Operationen jedes **find** durch passendes **compress**
- beachte: resultierende Folge liefert zu jedem Zeitpunkt die selbe Datenstruktur
- außerdem: asymptotische Laufzeit bleibt gleich
- gleich: sortiere **compress** nach hinten

# Laufzeitanalyse: Umsortieren der Operationen

## Tausche **compress** nach hinten

- Ergebnis und Laufzeit werden durch den Tausch nicht geändert
- wir können annehmen: zunächst alle **link** und dann alle **compress** Operationen



# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

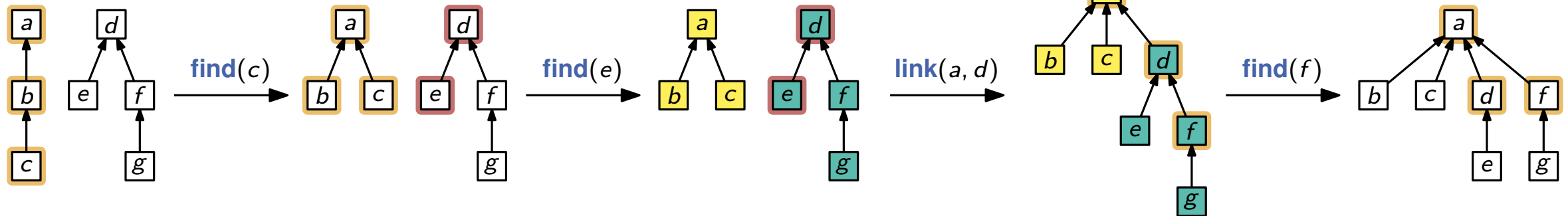
- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
} aber: selbe asymptotische Laufzeit

# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
 aber: selbe asymptotische Laufzeit

eigentliche Folge: **union**( $c, e$ ), **find**( $f$ )



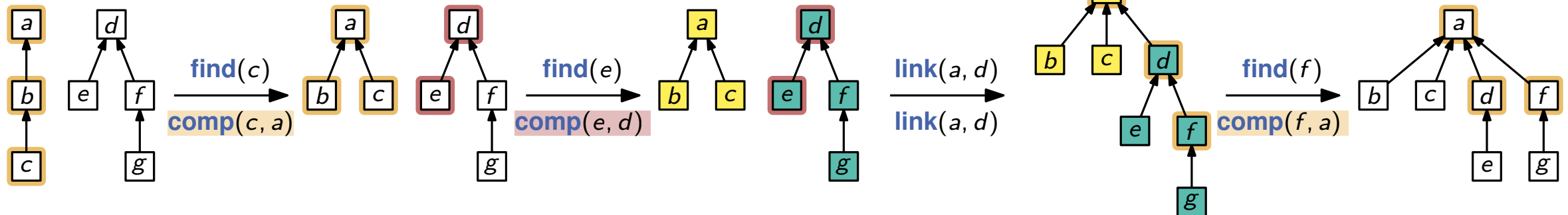


# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
aber: selbe asymptotische Laufzeit

eigentliche Folge: **union**( $c, e$ ), **find**( $f$ )

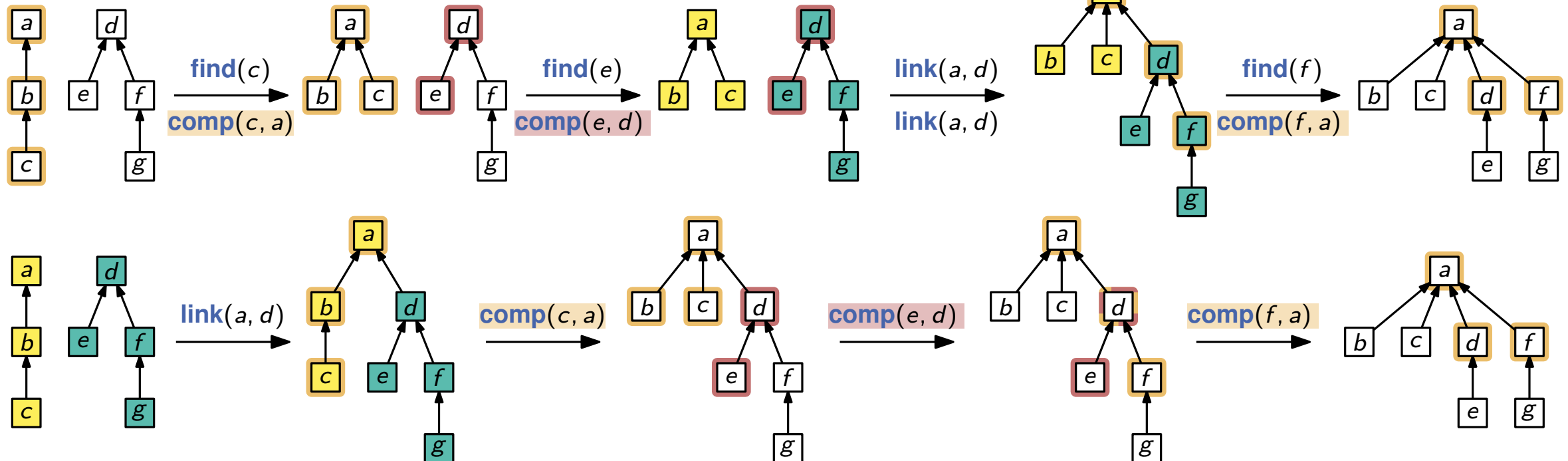


# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
aber: selbe asymptotische Laufzeit

eigentliche Folge: **union**( $c, e$ ), **find**( $f$ )



# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
} aber: selbe asymptotische Laufzeit

## Die $n$ **link** Operationen

- **link** läuft in Konstanter Zeit  $\rightarrow \Theta(n)$
- Ergebnis: Wald mit höchstens  $N = 2n$  nicht-isolierten Knoten

# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

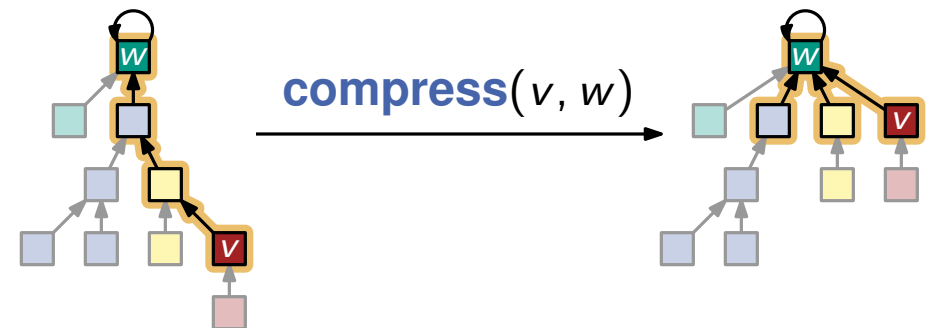
- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
 aber: selbe asymptotische Laufzeit

## Die $n$ **link** Operationen

- **link** läuft in Konstanter Zeit  $\rightarrow \Theta(n)$
- Ergebnis: Wald mit höchstens  $N = 2n$  nicht-isolierten Knoten

## Was wir noch tun müssen

- analysiere  $M = 2n + m$  **compress** Operationen auf einem Wald mit  $N$  Knoten



# Laufzeitanalyse: Zwischenstand

## Umbau der Folge von Operationen

- eigentlich:  $n$  **union** und  $m$  **find** in beliebiger Reihenfolge
  - stattdessen:  $n$  **link** gefolgt von  $2n + m$  **compress**
- } machen sehr unterschiedliche Dinge,  
 aber: selbe asymptotische Laufzeit

## Die $n$ **link** Operationen

- **link** läuft in Konstanter Zeit  $\rightarrow \Theta(n)$
- Ergebnis: Wald mit höchstens  $N = 2n$  nicht-isolierten Knoten

## Was wir noch tun müssen

- analysiere  $M = 2n + m$  **compress** Operationen auf einem Wald mit  $N$  Knoten

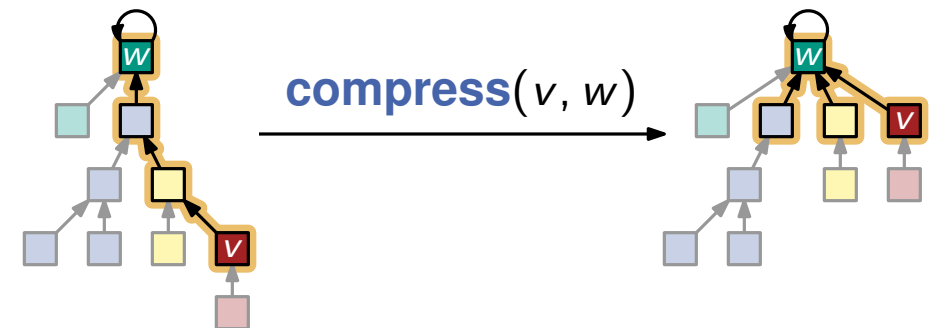
## Invarianten für die Ränge der Knoten

### Lemma

Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

### Lemma

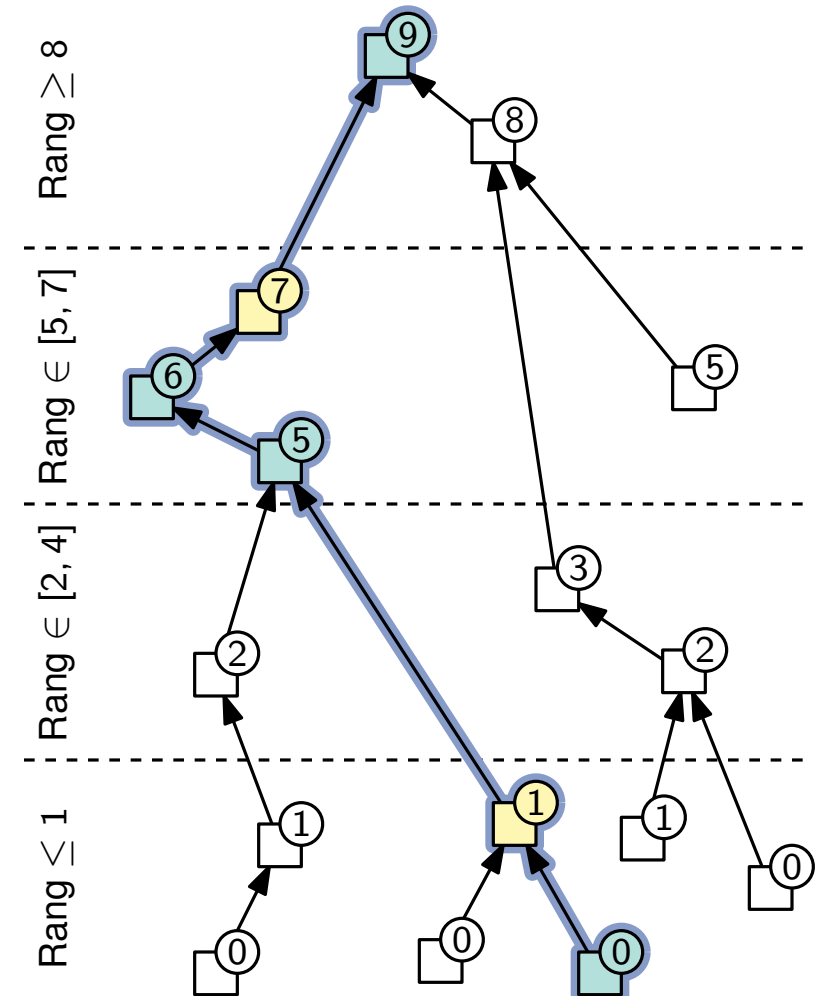
Es gibt höchstens  $N/2^r$  Knoten mit Rang größer  $r$ .



# Laufzeitanalyse: Einteilung in Ranggruppen

## Gruppen von Rängen

- gruppierere Knoten nach ihrem Rang in  $k$  Gruppen
- zwei Typen von Knoten bei einem **compress**-Pfad
  - Typ 1: Elter ist in der selben Gruppe
  - Typ 2: Elter ist in einer höheren Gruppe



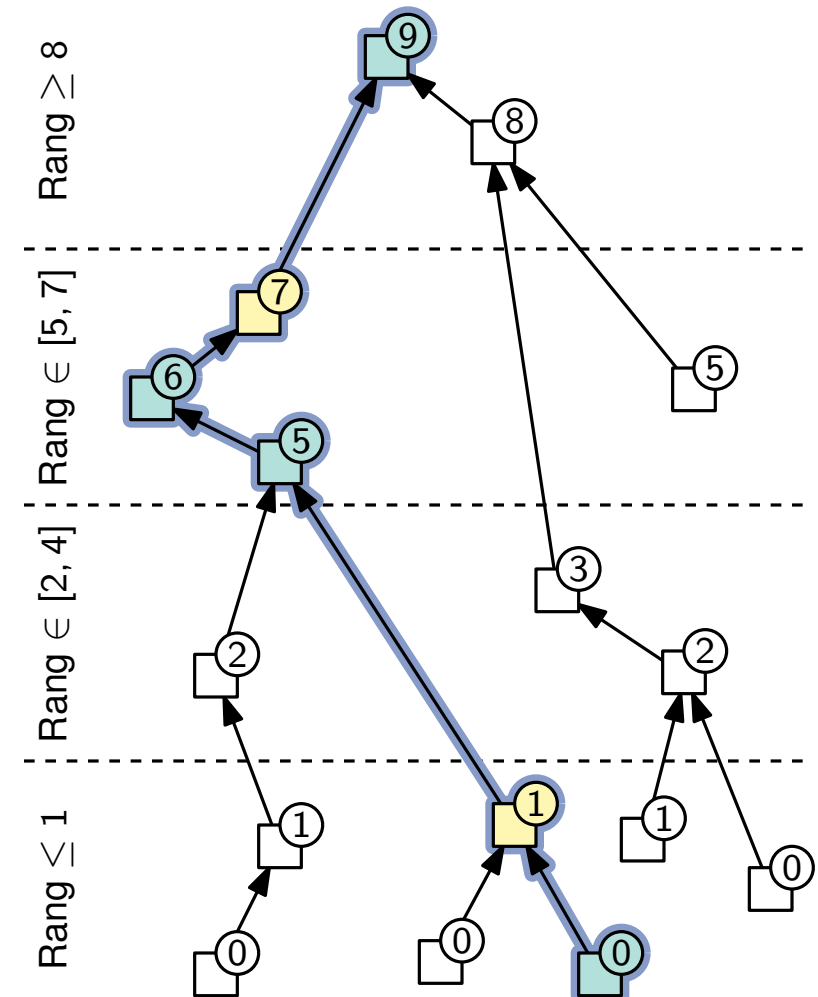
# Laufzeitanalyse: Einteilung in Ranggruppen

## Gruppen von Rängen

- gruppierere Knoten nach ihrem Rang in  $k$  Gruppen
- zwei Typen von Knoten bei einem **compress**-Pfad
  - Typ 1: Elter ist in der selben Gruppe
  - Typ 2: Elter ist in einer höheren Gruppe

## Wie oft hängen wir Typ 2 Knoten um?

- höchstens  $k$  Typ 2 Knoten pro **compress**
- $M$  mal **compress**  $\rightarrow$  insgesamt  $O(kM)$  Zeit



# Laufzeitanalyse: Einteilung in Ranggruppen

## Gruppen von Rängen

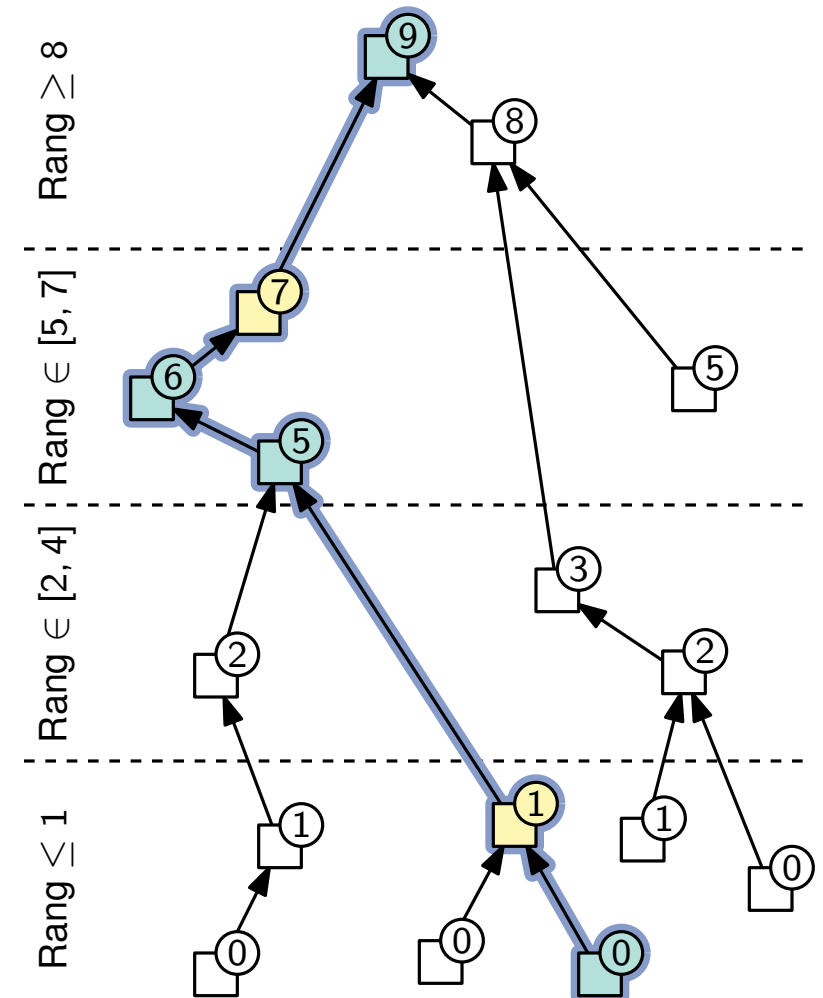
- gruppierere Knoten nach ihrem Rang in  $k$  Gruppen
- zwei Typen von Knoten bei einem **compress**-Pfad
  - Typ 1: Elter ist in der selben Gruppe
  - Typ 2: Elter ist in einer höheren Gruppe

## Wie oft hängen wir Typ 2 Knoten um?

- höchstens  $k$  Typ 2 Knoten pro **compress**
- $M$  mal **compress**  $\rightarrow$  insgesamt  $O(kM)$  Zeit

## Wie oft hängen wir Typ 1 Knoten um?

- betrachte beliebigen Knoten  $v$
- $v$  wird umgehängt  $\rightarrow$  neuer Elter hat größeren Rang
- kleiner Bereich von Rängen pro Gruppe
  - $\rightarrow v$  kann nicht oft Typ 1 sein bevor  $v$  Typ 2 wird

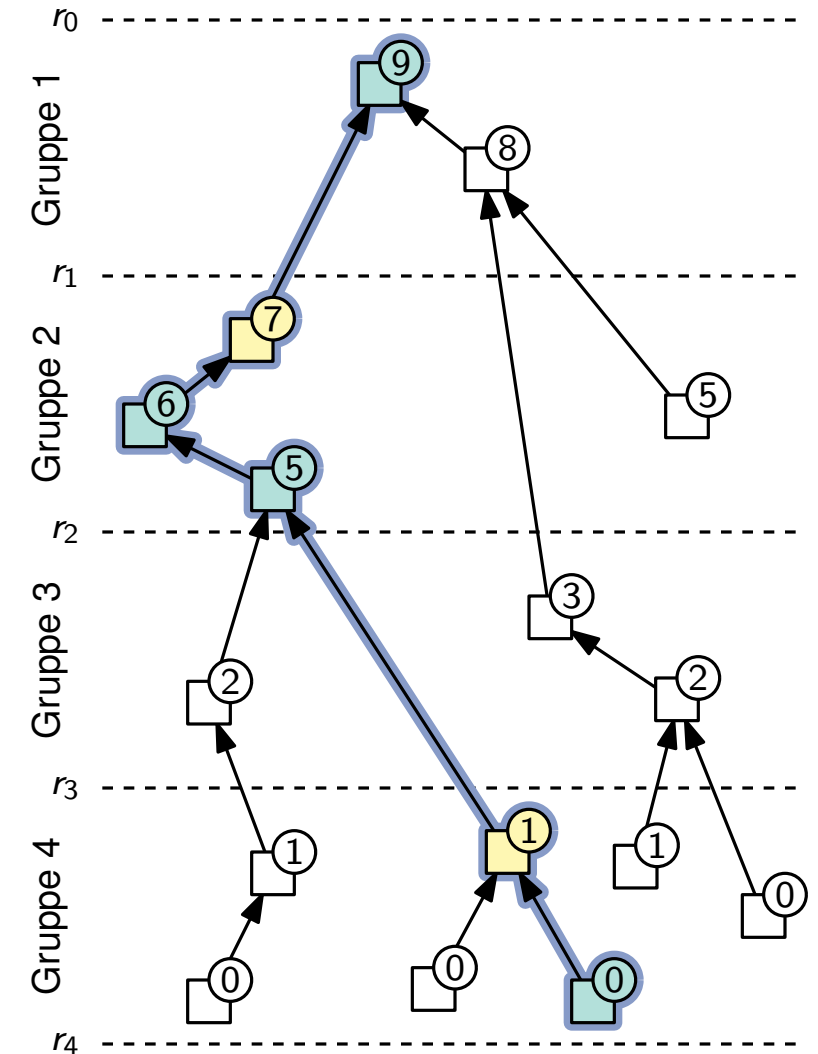




# Laufzeitanalyse: Type 1 Operationen

## Lemma

Sei Gruppe  $i$  die Menge aller Knoten mit Rang in  $(r_i, r_{i-1}]$  für  $r_0 > r_1 > \dots > r_k$ . In jeder Folge von **compress** Operationen werden insgesamt höchstens ??? Knoten aus Gruppe  $i$  von Type 1 umgehängt.



# Laufzeitanalyse: Type 1 Operationen

## Lemma

Sei Gruppe  $i$  die Menge aller Knoten mit Rang in  $(r_i, r_{i-1}]$  für  $r_0 > r_1 > \dots > r_k$ . In jeder Folge von **compress** Operationen werden insgesamt höchstens ??? Knoten aus Gruppe  $i$  von Typ 1 umgehängt.

## Beweis

### Lemma

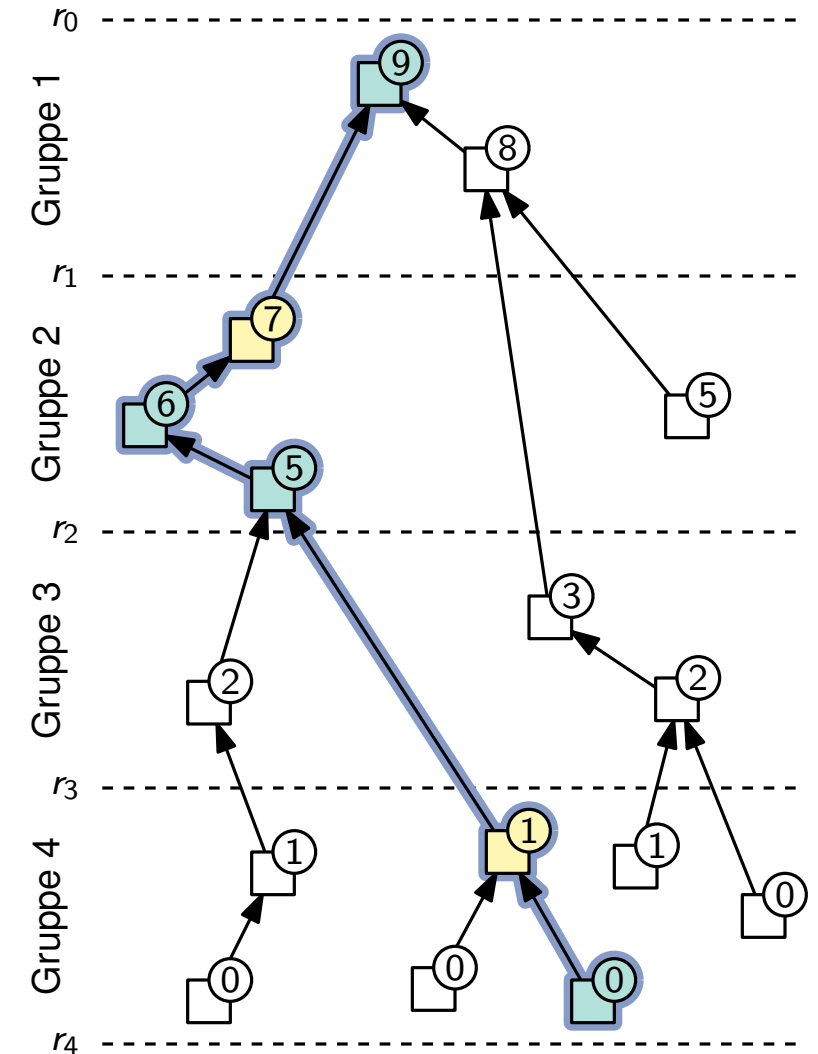
Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

Wie oft kann ein Knoten aus Gruppe  $i$  als Typ 1 Knoten umgehängt werden?

### Lemma

Es gibt höchstens  $N/2^r$  Knoten mit Rang größer  $r$ .

Wie viele Knoten enthält Gruppe  $i$  höchstens?



# Laufzeitanalyse: Type 1 Operationen

## Lemma

Sei Gruppe  $i$  die Menge aller Knoten mit Rang in  $(r_i, r_{i-1}]$  für  $r_0 > r_1 > \dots > r_k$ . In jeder Folge von **compress** Operationen werden insgesamt höchstens ??? Knoten aus Gruppe  $i$  von Typ 1 umgehängt.

## Beweis

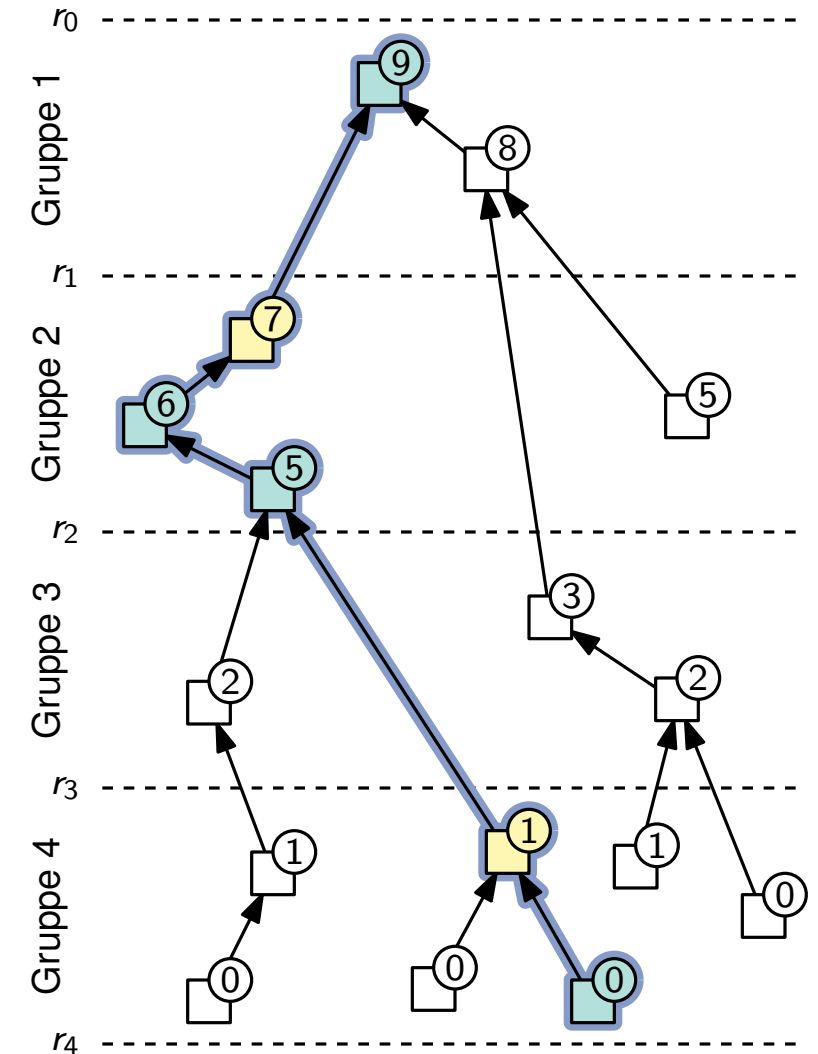
### Lemma

Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

- $v$  wird umgehängt  $\rightarrow$  neuer Elter hat größeren Rang
- jeder Knoten in Gruppe  $i$  ist höchstens  $r_{i-1}$  mal Typ 1

### Lemma

Es gibt höchstens  $N/2^r$  Knoten mit Rang größer  $r$ .



# Laufzeitanalyse: Type 1 Operationen

## Lemma

Sei Gruppe  $i$  die Menge aller Knoten mit Rang in  $(r_i, r_{i-1}]$  für  $r_0 > r_1 > \dots > r_k$ . In jeder Folge von **compress** Operationen werden insgesamt höchstens ??? Knoten aus Gruppe  $i$  von Typ 1 umgehängt.

## Beweis

### Lemma

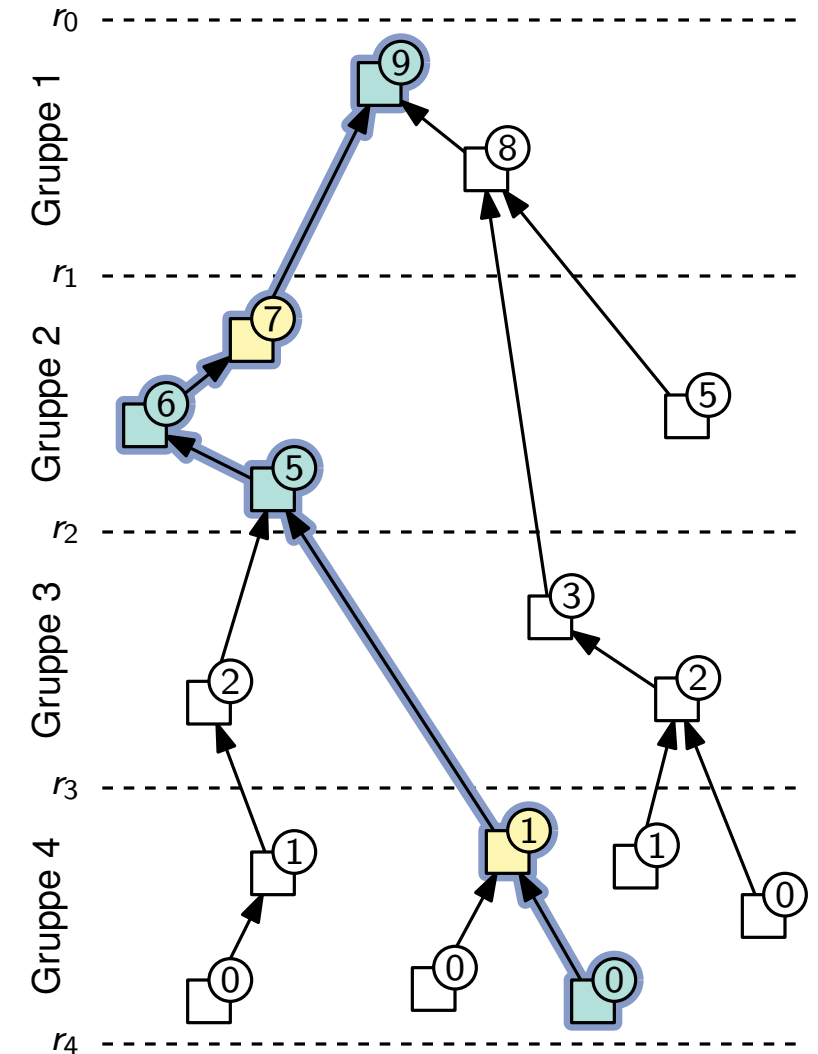
Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

- $v$  wird umgehängt  $\rightarrow$  neuer Elter hat größeren Rang
- jeder Knoten in Gruppe  $i$  ist höchstens  $r_{i-1}$  mal Typ 1

### Lemma

Es gibt höchstens  $N/2^r$  Knoten mit Rang größer  $r$ .

- jeder Knoten in Gruppe  $i$  hat Rang größer  $r_i$
- davon gibt es höchstens  $N/2^{r_i}$  viele



# Laufzeitanalyse: Type 1 Operationen

## Lemma

Sei Gruppe  $i$  die Menge aller Knoten mit Rang in  $(r_i, r_{i-1}]$  für  $r_0 > r_1 > \dots > r_k$ . In jeder Folge von **compress** Operationen werden insgesamt höchstens  $N \cdot \frac{r_{i-1}}{2^{r_i}}$  Knoten aus Gruppe  $i$  von Typ 1 umgehängt.

## Beweis

### Lemma

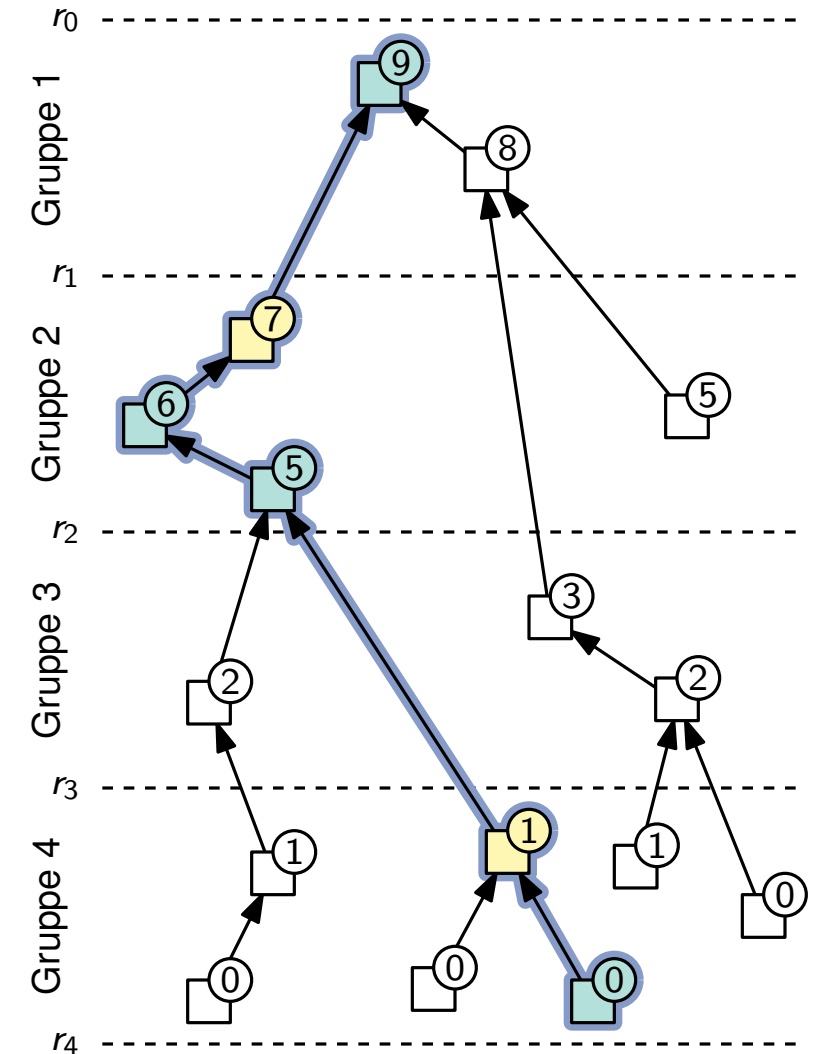
Jeder Knoten hat einen größeren Rang als jedes seiner Kinder.

- $v$  wird umgehängt  $\rightarrow$  neuer Elter hat größeren Rang
- jeder Knoten in Gruppe  $i$  ist höchstens  $r_{i-1}$  mal Typ 1

### Lemma

Es gibt höchstens  $N/2^r$  Knoten mit Rang größer  $r$ .

- jeder Knoten in Gruppe  $i$  hat Rang größer  $r_i$
- davon gibt es höchstens  $N/2^{r_i}$  viele



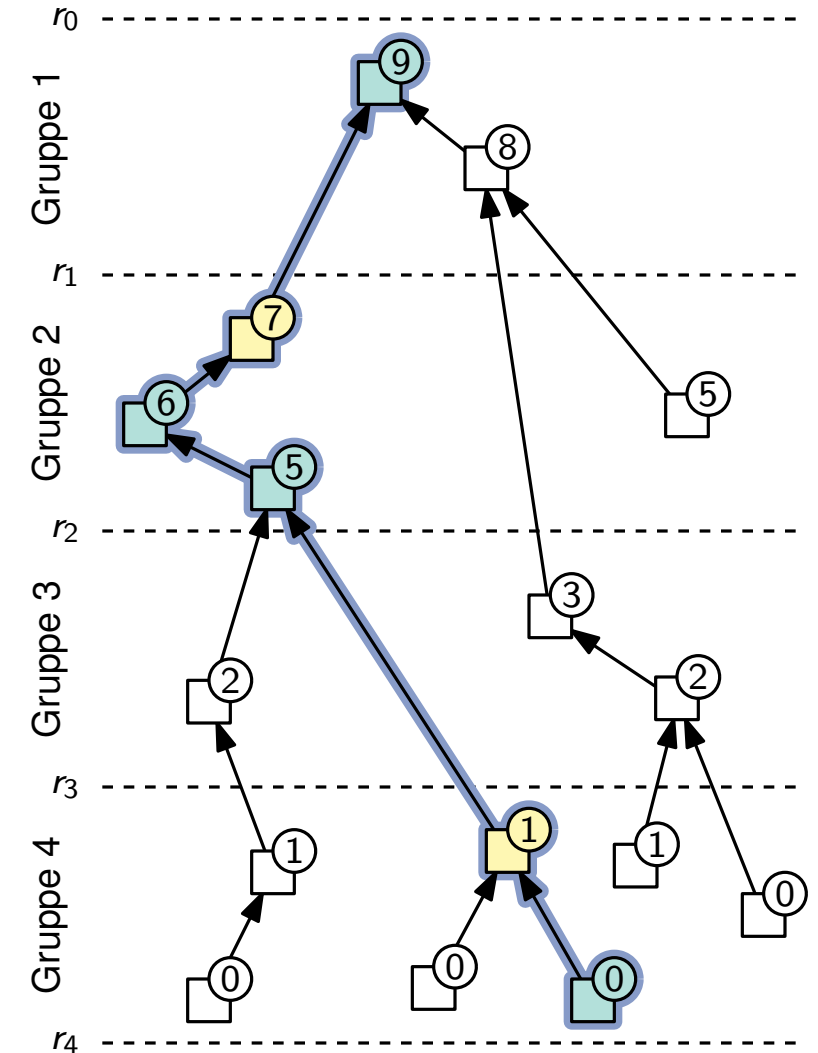
# Laufzeitanalyse: Wahl der Gruppen

## Lemma

Es werden insgesamt höchstens  $N \cdot \frac{r_i - 1}{2^{r_i}}$  Knoten aus Gruppe  $i$  von Typ 1 umgehängt.  
 ( $N$ : Anzahl Knoten)

## Lemma

Es werden insgesamt nur  $O(kM)$  Knoten von Typ 2 umgehängt.  
 ( $M$ : Anzahl **compress** Operationen,  $k$ : Anzahl Gruppen)



# Laufzeitanalyse: Wahl der Gruppen

## Lemma

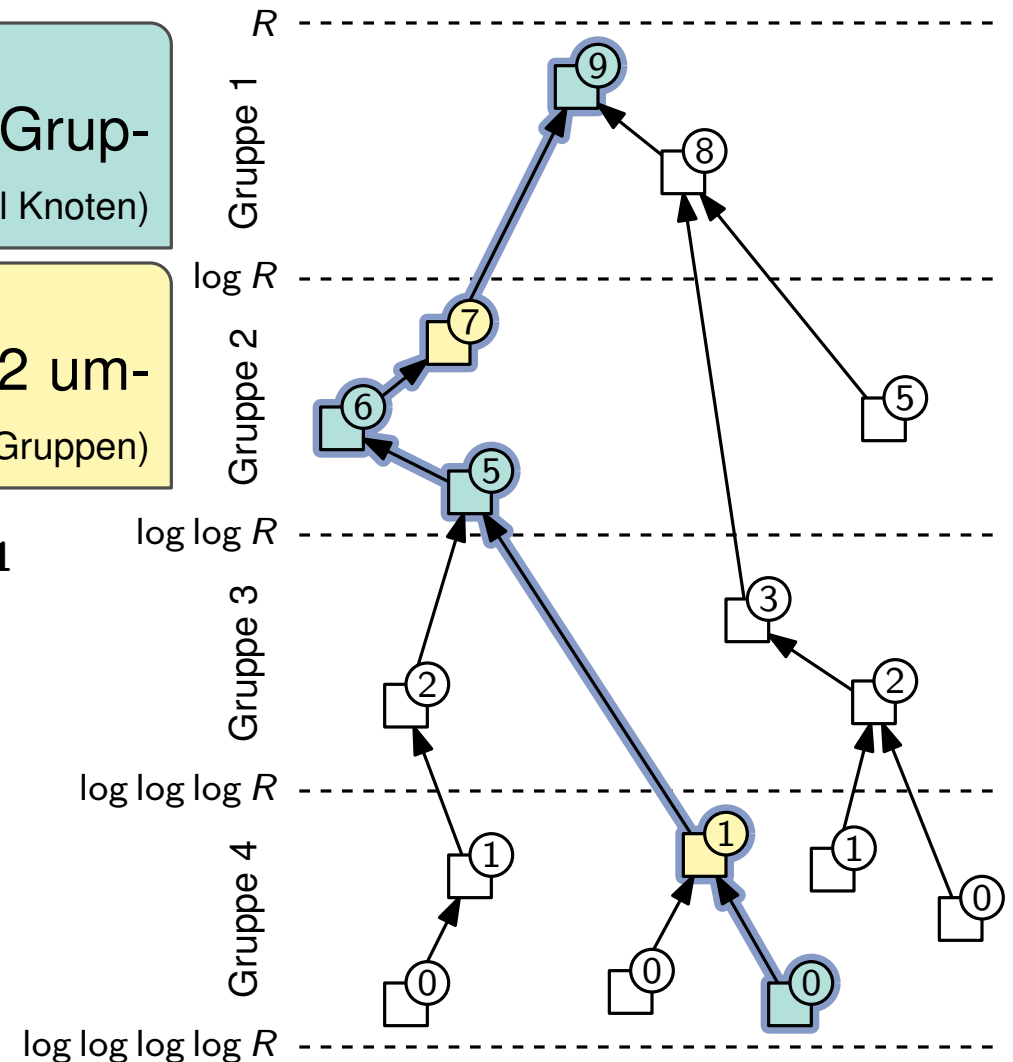
Es werden insgesamt höchstens  $N \cdot \frac{r_{i-1}}{2^{r_i}}$  Knoten aus Gruppe  $i$  von Typ 1 umgehängt. ( $N$ : Anzahl Knoten)

## Lemma

Es werden insgesamt nur  $O(kM)$  Knoten von Typ 2 umgehängt. ( $M$ : Anzahl **compress** Operationen,  $k$ : Anzahl Gruppen)

**Wähle:**  $r_0 = R$  (maximaler Rang) und  $r_i = \log r_{i-1}$

- dann: Kosten für Typ 1 in Gruppe  $i$  in  $O(N)$
- Typ 1 insgesamt:  $O(kN)$



# Laufzeitanalyse: Wahl der Gruppen

## Lemma

Es werden insgesamt höchstens  $N \cdot \frac{r_{i-1}}{2^{r_i}}$  Knoten aus Gruppe  $i$  von Typ 1 umgehängt. ( $N$ : Anzahl Knoten)

## Lemma

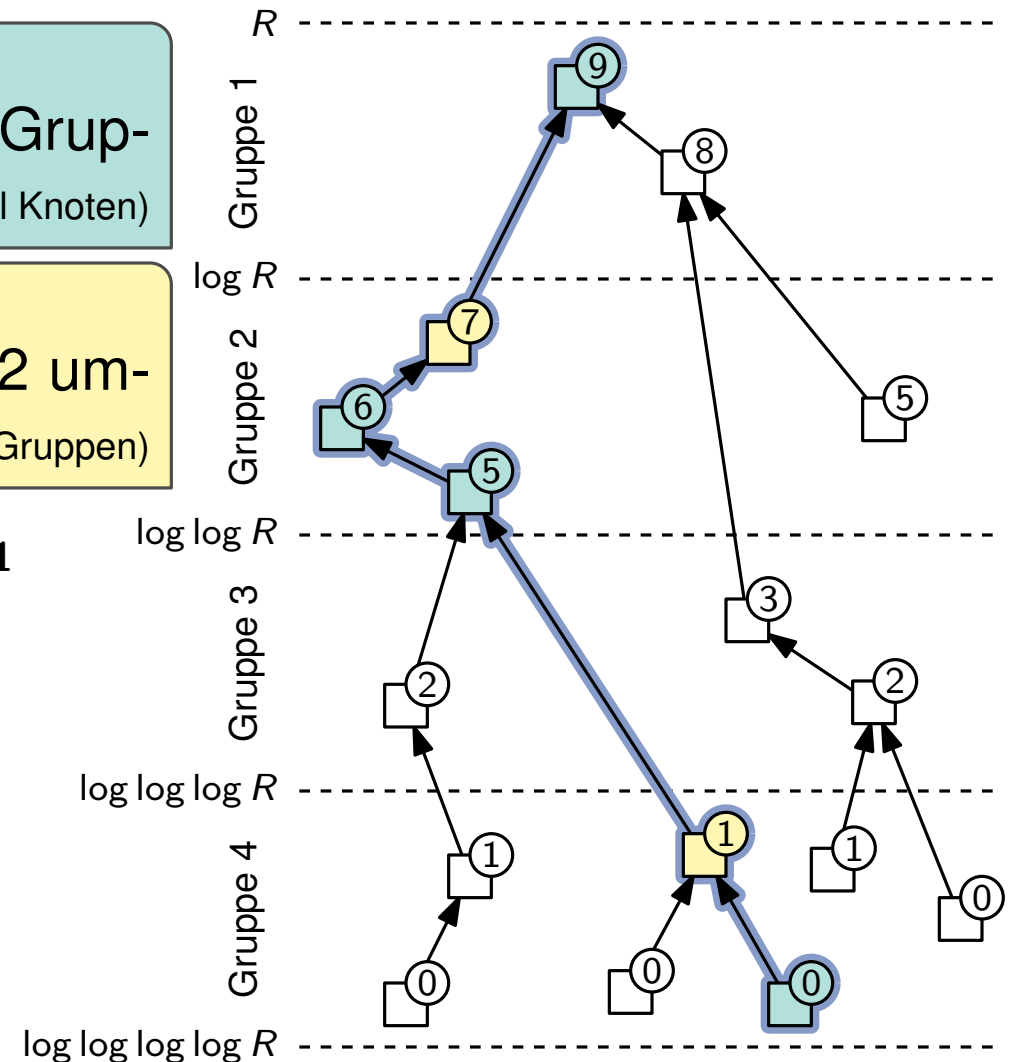
Es werden insgesamt nur  $O(kM)$  Knoten von Typ 2 umgehängt. ( $M$ : Anzahl **compress** Operationen,  $k$ : Anzahl Gruppen)

**Wähle:**  $r_0 = R$  (maximaler Rang) und  $r_i = \log r_{i-1}$

- dann: Kosten für Typ 1 in Gruppe  $i$  in  $O(N)$
- Typ 1 insgesamt:  $O(kN)$

**Wie viele Gruppen bekommen wir dann?**

- Nach wie vielen  $\log$  ist das Ergebnis  $\leq 1$ ?
- diese Anzahl heißt **iterierten Logarithmus**  $\log^* R$
- beachte:  $R \in O(\log N)$





# Laufzeitanalyse

## Folgerung

Eine Folge von  $M$  **compress** Operationen in einem Wald mit  $N$  Knoten benötigt nur  $O((N + M) \log^* N)$  Umhängeoperationen.

# Laufzeitanalyse

## Folgerung

Eine Folge von  $M$  **compress** Operationen in einem Wald mit  $N$  Knoten benötigt nur  $O((N + M) \log^* N)$  Umhängeoperationen.

## Zurück zu **union** und **find**

- was wir analysieren wollten:  $n$  **union** und  $m$  **find** Operationen
- was wir analysiert haben:
  - $n$  **link**  $\rightarrow$  liefert Wald mit höchstens  $N = 2n$  Knoten in  $\Theta(n)$  Zeit
  - $M = 2n + m$  **compress** auf Wald mit  $N$  Knoten in  $O((N + M) \log^* N)$
- wir wissen: **union**, **find**-Folge hat gleiche Laufzeit wie **link**, **compress**-Folge

# Laufzeitanalyse

## Folgerung

Eine Folge von  $M$  **compress** Operationen in einem Wald mit  $N$  Knoten benötigt nur  $O((N + M) \log^* N)$  Umhängeoperationen.

## Zurück zu **union** und **find**

- was wir analysieren wollten:  $n$  **union** und  $m$  **find** Operationen
- was wir analysiert haben:
  - $n$  **link**  $\rightarrow$  liefert Wald mit höchstens  $N = 2n$  Knoten in  $\Theta(n)$  Zeit
  - $M = 2n + m$  **compress** auf Wald mit  $N$  Knoten in  $O((N + M) \log^* N)$
- wir wissen: **union**, **find**-Folge hat gleiche Laufzeit wie **link**, **compress**-Folge

## Theorem

Eine beliebige Folge von  $n$  **union** und  $m$  **find** Operationen mit Union-by-Rank und Pfadkompression benötigt  $O((n + m) \log^* n)$  Zeit. Also amortisiert  $O(\log^* n)$  pro Operation.

# Wie cool ist das denn?

## Theorem

Eine beliebige Folge von  $n$  **union** und  $m$  **find** Operationen mit Union-by-Rank und Pfadkompression benötigt  $O((n + m) \log^* n)$  Zeit. Also amortisiert  $O(\log^* n)$  pro Operation.

# Wie cool ist das denn?

## Theorem

Eine beliebige Folge von  $n$  **union** und  $m$  **find** Operationen mit Union-by-Rank und Pfadkompression benötigt  $O((n + m) \log^* n)$  Zeit. Also amortisiert  $O(\log^* n)$  pro Operation.

## Wie schnell wächst $\log^*(n)$ ?

- betrachte Potenzturm der Höhe  $k$ :  $n = 2^{2^{\dots^2}}$
- dann gilt:  $\log^* n = k$
- mit größer werdendem  $k$  wächst  $n$  extrem schnell  
→ mit größer werdendem  $n$  wächst  $\log^* n$  extrem langsam

# Wie cool ist das denn?

## Theorem

Eine beliebige Folge von  $n$  **union** und  $m$  **find** Operationen mit Union-by-Rank und Pfadkompression benötigt  $O((n + m) \log^* n)$  Zeit. Also amortisiert  $O(\log^* n)$  pro Operation.

## Wie schnell wächst $\log^*(n)$ ?

- betrachte Potenzturm der Höhe  $k$ :  $n = 2^{2^{\dots^2}}$
- dann gilt:  $\log^* n = k$
- mit größer werdendem  $k$  wächst  $n$  extrem schnell  
→ mit größer werdendem  $n$  wächst  $\log^* n$  extrem langsam

## Zum Vergleich

- für  $n \in (65\,536, 2^{65\,536}]$  gilt  $\log^* n = 5$  und  $\log n \in (16, 65\,536]$

# Wie cool ist das denn?

## Theorem

Eine beliebige Folge von  $n$  **union** und  $m$  **find** Operationen mit Union-by-Rank und Pfadkompression benötigt  $O((n + m) \log^* n)$  Zeit. Also amortisiert  $O(\log^* n)$  pro Operation.

## Wie schnell wächst $\log^*(n)$ ?

- betrachte Potenzturm der Höhe  $k$ :  $n = 2^{2^{\dots^2}}$
- dann gilt:  $\log^* n = k$
- mit größer werdendem  $k$  wächst  $n$  extrem schnell  
 → mit größer werdendem  $n$  wächst  $\log^* n$  extrem langsam

## Zum Vergleich

- für  $n \in (65\,536, 2^{65\,536}]$  gilt  $\log^* n = 5$  und  $\log n \in (16, 65\,536]$
- $2^{65\,536} > 10^{19\,728}$
- das beobachtbare Universum hat zwischen  $10^{78}$  und  $10^{82}$  Atome

# Zusammenfassung

## Union–Find

- sehr nützliche Datenstruktur
- in vielen Programmiersprachen (z.B. C++, Java) nicht Teil des Standards
- aber: nicht so schwer selbst zu implementieren (Analyse kompliziert, Algo selbst leicht)



# Zusammenfassung

## Union-Find

- sehr nützliche Datenstruktur
- in vielen Programmiersprachen (z.B. C++, Java) nicht Teil des Standards
- aber: nicht so schwer selbst zu implementieren (Analyse kompliziert, Algo selbst leicht)

## Laufzeitanalyse

- spannende Technik: baue den Algorithmus zum Zweck der Analyse um
- der analysierte Algo macht ggf. gar nicht das gewünschte oder ist nicht umsetzbar
- aber: die Analyse des anderen Algos liefert trotzdem Schranke für den eigentlichen Algo

# Zusammenfassung

## Union-Find

- sehr nützliche Datenstruktur
- in vielen Programmiersprachen (z.B. C++, Java) nicht Teil des Standards
- aber: nicht so schwer selbst zu implementieren (Analyse kompliziert, Algo selbst leicht)

## Laufzeitanalyse

- spannende Technik: baue den Algorithmus zum Zweck der Analyse um
- der analysierte Algo macht ggf. gar nicht das gewünschte oder ist nicht umsetzbar
- aber: die Analyse des anderen Algos liefert trotzdem Schranke für den eigentlichen Algo

## Ausblick: Da geht noch was

- kann verbessert werden auf amortisierte Laufzeit  $\Theta(\alpha(n))$  pro Operation (das ist tight)
- $\alpha(n)$ : inverse Ackermannfunktion
- $\alpha(n)$  wächst noch langsamer als  $\log^* n$