

Teil 4

# Verwaltung von Mengen

In vielen Algorithmen ist die **Verwaltung und Manipulation von Mengen** ein wesentlicher Bestandteil. Daher ist es sinnvoll sich, unabhängig von konkreten Anwendungen, mit den grundlegenden Datenstrukturen und Algorithmen auf Mengen zu beschäftigen.

Um sinnvoll über die Repräsentation und den Umgang mit Mengen sprechen zu können, müssen wir zuerst die grundlegenden Operationen festlegen und beschreiben.

Anschließend können wir versuchen sie als Programm zu realisieren.

# Grundlegende Mengenoperationen

**Variablen:** Mengen  $S, S_1, S_2$  und ein Element  $a$ .

$$\text{member}(a, S) = \begin{cases} 1 & \text{falls } a \in S \\ 0 & \text{sonst} \end{cases}$$

$$\text{insert}(a, S): S = S \cup \{a\}$$

$$\text{delete}(a, S): S = S \setminus \{a\} = S - \{a\}$$

$$\text{union}(S_1, S_2): S_1 = S_1 \cup S_2$$

$\text{find}(a)$ : Gibt **eine** Menge zurück, die  $a$  enthält.

Für geordnete Mengen haben wir zusätzlich:

$\text{split}(a, S)$ : Gibt zwei Mengen  $S_1$  und  $S_2$  zurück, mit  
 $S_1 := \{b \in S \mid b \leq a\}$  und  $S_2 := \{b \in S \mid b > a\}$ .

$\text{min}(S)$ : Gibt das kleinste Element von  $S$  bzgl.  $\leq$  zurück.

Die Operation `split` wurde z.B. bereits in QUICKSORT, FIND und SELECT benutzt.

# Grundlegende Mengenoperationen

## Bemerkung

Häufig referenziert man eine Menge  $S$  nicht über ein Mengen-Objekt, sondern über **Repräsentanten**, d.h. ein Element  $a \in S$ .

Einen solchen Repräsentanten für eine Menge  $S$  erhält man in der Regel, in dem man `find(a)` für irgendein Element  $a \in S$  aufruft.

Dies macht natürlich nur Sinn, wenn die Menge eindeutig durch den Repräsentanten festgelegt ist.

Dies ist insbesondere dann der Fall, wenn wir Familien von **disjunkten** Mengen betrachten.

Natürlich gibt es noch mehr Operationen auf Mengen. Aber diese lassen sich durch Kombinationen der oben angegebenen Operationen darstellen.

# Grundlegende Mengenoperationen

In der Regel versucht man nicht eine Datenstruktur für alle Operationen zu optimieren. Sondern man bestimmt, welche Operationen für die vorliegende Anwendung besonders kritisch/häufig sind, und versucht diese zu optimieren.

Dadurch ergeben sich verschiedene Typen von Datenstrukturen:

Wörterbuch (Dictionary): insert, delete, member

Mergeable Heap: insert, delete, member, union

Union-Find-Datenstruktur: union, find

Prioritäts-Warteschlange (Priority-Queue): min, insert, delete

Wir beschäftigen uns im Folgenden mit den letzten beiden Datenstrukturen.

Teil 4: Verwaltung von Mengen

# Union-Find-Datenstrukturen

# Partitionen

## Definition (Partition)

$M$  sei eine (endliche) Menge. Eine **Partition**  $\mathcal{P}$  ist eine Menge von Teilmengen von  $M$ , so dass:

- 1 Für alle  $X \in \mathcal{P}$  gilt  $X \neq \emptyset$ .
- 2  $\bigcup_{X \in \mathcal{P}} X = M$
- 3 Für alle  $X, Y \in \mathcal{P}$  mit  $X \neq Y$  gilt:  $X \cap Y = \emptyset$ .

Eine Union-Find-Datenstruktur implementiert die folgenden Funktionen auf einer Partition  $\mathcal{P}$ :

$\text{make\_set}(\mathcal{P}, x) := \mathcal{P} \cup \{\{x\}\}$  wobei  $x \notin \mathcal{P}$

$\text{find}(\mathcal{P}, x) := X \in \mathcal{P}$  mit  $x \in X$  wobei  $x \in \mathcal{P}$

$\text{union}(\mathcal{P}, X, Y) := (\mathcal{P} \setminus \{X, Y\}) \cup \{X \cup Y\}$  wobei  $X, Y \in \mathcal{P}$

Ist eindeutig, auf welche Partition sich die Operationen beziehen, lassen wir das Argument  $\mathcal{P}$  in der Regel weg.

Teil 4: Verwaltung von Mengen  
Union-Find-Datenstrukturen

Anwendung:  
Zusammenhangskomponenten

## Union-Find Anwendung: Zusammenhangskomponenten

Sei  $G = (V, E)$  ein ungerichteter, einfacher Graph, d.h.  $E \subseteq V^2$  (keine Kantengewichte, keine mehrfachen Kanten).

Ein **Weg von  $u$  nach  $v$  in  $G$**  ist eine Folge  $(v_1, \dots, v_l)$  von Knoten von  $G$ , so dass

- 1  $u = v_0$  und  $v = v_l$
- 2 Für jedes  $1 \leq i \leq l$  ist  $(v_{i-1}, v_i) \in E$ .

Wir schreiben  $u \rightsquigarrow v$ , wenn ein Weg von  $u$  nach  $v$  existiert.

Da  $G$  ungerichtet ist, ist  $u \rightsquigarrow v$  eine Äquivalenzrelation (reflexiv, transitiv), sofern wir die leeren Wege zulassen (symmetrisch).

Eine **Zusammenhangskomponente** von  $G$  ist eine Äquivalenzklasse bzgl.  $\rightsquigarrow$ , d.h. es ist eine **maximale** Knotenmenge  $X \subseteq V$ , so dass für  $u, v \in X$  ein Weg von  $u$  nach  $v$  existiert. **Maximal** bedeutet dabei, dass keine Knotenmenge  $Y$  existiert, die ebenfalls diese Eigenschaft besitzt und  $X \subsetneq Y$ .

# Union-Find Anwendung: Zusammenhangskomponenten

## Problem (CONNECTED COMPONENTS)

**Gegeben:** Ein ungerichteter, einfacher Graph  $G = (V, E)$

*Gesucht ist eine Liste aller Zusammenhangskomponenten und ihrer Mitglieder.*

## Algorithmus (Zusammenhangskomponenten)

**Eingabe:** Ein ungerichteter, einfacher Graph  $G = (V, E)$

**Ausgabe:** Die Zusammenhangskomponenten von  $G$

**Daten:** Eine Union-Find-Datenstruktur  $\mathcal{P}$

**für**  $v \in V$  **tu**  $\text{make\_set}(v)$

**für**  $(u, v) \in E$  **tu**  $\text{union}(\text{find}(u), \text{find}(v))$

*Die Mengen von  $\mathcal{P}$  sind die Zusammenhangskomponenten.*

# Union-Find Anwendung: Zusammenhangskomponenten

## Lemma

*Der vorgestellte Algorithmus zur Berechnung der Zusammenhangskomponenten eines einfachen, ungerichteten Graphen ist korrekt.*

## Beweis

*Wir beweisen die Korrektheit per Induktion über die Anzahl  $m$  der Kanten. Für  $m = 0$  gilt die Behauptung offensichtlich, da jede Komponente nur aus einem Knoten besteht.*

*Für  $m \geq 1$  Seien  $e_1, \dots, e_m$  die Kanten in der Reihenfolge, in der sie vom Algorithmus durchlaufen werden.*

*Nach Induktionsvoraussetzung enthält die Partition  $\mathcal{P}$  nach der Behandlung der vorletzten Kante die Komponenten des Graphen  $G' := (V, \{e_1, \dots, e_{m-1}\})$ .*

# Union-Find Anwendung: Zusammenhangskomponenten

## Beweis (Fortsetzung)

*$G$  entsteht aus  $G'$  durch Hinzufügen der letzten Kante  $e_m = (u, v)$ . Liegen  $u$  und  $v$  bereits in der gleichen Komponente, so gilt*

$$\text{find}(u) = \text{find}(v)$$

*und die Vereinigung bewirkt keine Veränderung.*

*Liegen  $u$  und  $v$  in zwei verschiedenen Zusammenhangskomponenten  $U$  und  $V$ , werden diese offensichtlich zu einer neuen vereinigt.* □

Teil 4: Verwaltung von Mengen  
Union-Find-Datenstrukturen

# Realisierungen

## Union-Find mit Feldern

Nehmen wir an, dass wir eine Bijektion von  $M$  auf die Menge  $\{1, \dots, |M|\}$  haben, d.h.  $M$  hat die Form  $M = \{x_1, \dots, x_m\}$  mit  $m = |M|$ .

Dann können wir eine Partition  $\mathcal{P} = \{M_1, \dots, M_n\}$  von  $M$  in  $n$  Mengen, mittels eines Feldes  $P[1 \dots m]$  und  $n$  paarweise verschiedenen Labels  $\{S_1, \dots, S_n\}$  darstellen, indem wir

$$P[i] = S_j \text{ setzen, falls } x_i \in M_j.$$

Die Partition

$$\mathcal{P} = \{\{1, 5, 9\}, \{2, 6, 8\}, \{3, 4, 7, 10\}\}$$

von  $M = \{1, \dots, 10\}$  hätte somit die Form

1	2	3	4	5	6	7	8	9	10
$S_1$	$S_2$	$S_3$	$S_3$	$S_1$	$S_2$	$S_3$	$S_2$	$S_1$	$S_3$

$\text{union}(\mathcal{P}, S_3, S_1)$  ergäbe dann

1	2	3	4	5	6	7	8	9	10
$S_3$	$S_2$	$S_3$	$S_3$	$S_3$	$S_2$	$S_3$	$S_2$	$S_3$	$S_3$

## Union-Find mit Feldern

Für `union` würde man also die Labels der zweiten Menge durch die der ersten Menge ersetzen. Dazu muss das gesamte Feld durchlaufen werden, was  $O(n)$  benötigt.

`find` ließe sich in konstanter Zeit realisieren, da lediglich das Label aus dem Feld gelesen werden müsste.

### Union-Find mit Feldern

Die Operationen der Union-Find-Struktur mit Feldern auf  $n$  Elementen benötigen:

`union`:  $O(n)$

`find`:  $O(1)$

## Union-Find mit Listen

Statt jeder Menge ein Label zuzuordnen und sich diese in einem Feld, das über die Elemente in  $M$  indiziert ist, zu merken, kann man sich jede Teilmenge als Liste der in ihr enthaltenen Elemente vorstellen.

`union` wäre dann lediglich das Aneinanderhängen von Listen und für `find` müssten alle Listen durchsucht werden.

### Union-Find mit Listen

Die Operationen der Union-Find-Struktur mit Listen auf  $n$  Elementen benötigen:

`union`:  $O(1)$

`find`:  $O(n)$

## Frage

*Wie kann man erreichen, dass beide Operationen, union und find, relativ billig sind?*

## Antwort

*Mittels Bäumen.*

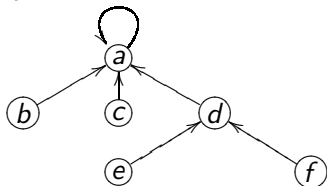
Eine Menge der Partition wird durch einen Baum dargestellt, dessen Knoten mit den Elementen der Menge beschriftet sind.

Eine Partition einer Menge  $M$  ist somit ein **Wald**, d.h. eine Menge von Bäumen.

Die genaue Form der Bäume ergibt sich dabei über die Folge der durchgeführten Operationen. D.h. wir werden im Folgenden beschreiben, wie die Operationen realisiert werden und wieviel Zeit sie in Anspruch nehmen.

# Die Bäume

Die einzige Anforderung an die Bäume ist, dass jeder Knoten  $x$  seinen Vorgänger  $\text{pred}(x)$  kennt.



Um die Wurzel eines Baumes erkennen zu können, setzen wir ihren Vorgänger auf sich selbst, d.h.

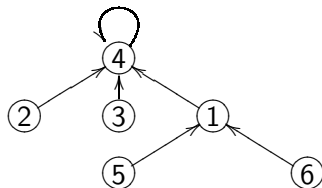
$$x \text{ ist eine Wurzel} \Leftrightarrow \text{pred}(x) = x$$

Einen Baum benennen wir mit seiner Wurzel.

## Realisierung der Bäume

Wir gehen davon aus, dass die  $m$  Elemente von  $M$  eindeutig geordnet sind, d.h. wir können davon ausgehen, dass  $M = \{1, \dots, m\}$ .

Dann können wir die gesamte Partition durch ein Feld mit  $m$  Einträgen darstellen.

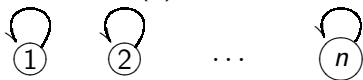


Dieser Baum wird durch das folgende Feld repräsentiert:

$x$	1	2	3	4	5	6
$\text{pred}(x)$	4	4	4	4	1	1

Wird ein neues Element in die Partition eingefügt, wird es zu einem Baum mit einem Knoten.

D.h. nach der Sequenz  $\text{make\_set}(1), \dots, \text{make\_set}(n)$  hat die Partition die Form



### Prozedur ( $\text{make\_set}$ )

**Eingabe:** Eine Partition  $\mathcal{P}$  und ein Element  $x \notin \mathcal{P}$

**Ausgabe:**  $\mathcal{P} \cup \{\{x\}\}$

$\text{pred}(x) := x$

# Find

`find(x)` traversiert den Baum, beginnend bei  $x$ , bis die Wurzel erreicht ist und gibt diese zurück.

## Prozedur (`find`)

**Eingabe:** Eine Partition  $\mathcal{P}$  und ein Element  $x$

**Ausgabe:** Die Menge  $X \in \mathcal{P}$  mit  $x \in X$

$y := x$

**solange**  $\text{pred}(y) \neq y$  **tue**  $y := \text{pred}(y)$

Gebe  $y$  zurück

Die Laufzeit von `find` entspricht der Tiefe von  $x$  in seinem Baum.

# Union

$\text{union}(x, y)$  hängt einfach die Wurzel des Baumes von  $y$  an die Wurzel des Baumes von  $x$ .

## Prozedur ( $\text{union}$ )

**Eingabe:** Eine Partition  $\mathcal{P}$  und zwei Elemente  $x, y$

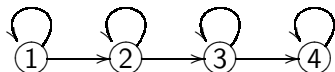
**Ausgabe:**  $(\mathcal{P} \setminus \{X, Y\}) \cup \{X \cup Y\}$  mit  $X = \text{find}(x)$  und  $Y = \text{find}(y)$

$a := \text{find}(x), b := \text{find}(y)$

$\text{pred}(b) := a$

Diese Implementation kann dazu führen, dass die Bäume **entarten**.

Durch die Sequenz  **$\text{union}(2, 1)$** ,  **$\text{union}(3, 2)$** ,  **$\text{union}(4, 3)$**  würde die Partition auf  $\{1, 2, 3, 4\}$  sich folgendermaßen entwickeln.



## Ränge und die gewichtete Vereinigungsregel

Günstiger, insbesondere in Anbetracht der Tatsache, dass die Laufzeit von `find` von der Tiefe der Bäume abhängt, ist es eine **gewichtete Vereinigungsregel** zu verwenden.

Dazu ordnen wir jedem Element  $x$  in der Union-Find-Datenstruktur einen **Rang**  $\text{rank}(x)$  zu, der stets eine obere Schranke für die Höhe des Elementes, d.h. seinem maximalen Abstand zu einem Blatt, ist.

Diesen Rang setzen wir nach dem Einfügen mittels `make_set(x)` auf 0 und aktualisieren ihn während der Ausführung von `union`. Die Prozedur `find` bleibt (vorerst) unverändert

### Prozedur (`make_set` mit Rängen)

**Eingabe:** Eine Partition  $\mathcal{P}$  und ein Element  $x$

**Ausgabe:**  $\mathcal{P} \cup \{\{x\}\}$

$\text{pred}(x) := x, \text{rank}(x) := 0$

## Ränge und die gewichtete Vereinigungsregel

`union` wird so abgeändert, dass der Baum mit kleinerem Rang an den mit größerem Rang gehängt wird.

### Prozedur (`union` mit Rängen)

**Eingabe:** Eine Partition  $\mathcal{P}$  und zwei Elemente  $x, y$

**Ausgabe:**  $(\mathcal{P} \setminus \{X, Y\}) \cup \{X \cup Y\}$  mit  $X = \text{find}(x)$  und  $Y = \text{find}(y)$

$a := \text{find}(x), b := \text{find}(y)$

**wenn**  $\text{rank}(a) < \text{rank}(b)$  **dann**

$\text{pred}(a) := b$

**sonst**

$\text{pred}(b) := a$

**wenn**  $\text{rank}(a) = \text{rank}(b)$  **dann**  $\text{rank}(a) := \text{rank}(a) + 1$

**Ende**

## Die gewichtete Vereinigungsregel

Wie `make_set` benötigt `union` konstante Zeit.

Um die Laufzeit von `find` abschätzen zu können müssen wir die möglichen Tiefen der Bäume betrachten.

### Lemma

*Das Verfahren nach der gewichteten Vereinigungsregel erzeugt, beginnend mit der diskreten Partition, nur Bäume, so dass ein Baum der Höhe  $h$  mindestens  $2^h$  Knoten besitzt.*

### Beweis

*Seien  $T_1$  und  $T_2$  Bäume mit den Größen  $\text{size}(T_1) \geq \text{size}(T_2)$  und den Höhen  $h_1$  und  $h_2$*

*Der entstehende Baum  $T := T_1 \cup T_2$  hat die nebenstehende Form.*

## Beweis (Fortsetzung)

Die Höhe von  $T$  ist somit  $h := \max(h_1, h_2 + 1)$ .

Falls  $h = h_1 \geq h_2 + 1$ , ergibt sich für die Anzahl der Knoten:

$$\text{size}(T) = \text{size}(T_1) + \text{size}(T_2) \geq 2^{h_1} + 2^{h_2} \geq 2^{h_1} = 2^h.$$

Falls  $h = h_2 + 1 \geq h_1$ , ergibt sich für  $\text{size}(T_1) \geq \text{size}(T_2) \geq 2^{h_2}$  und somit:

$$\text{size}(T) = \text{size}(T_1) + \text{size}(T_2) \geq 2^{h_2} + 2^{h_2} = 2^{h_2+1} = 2^h.$$



## Konsequenz

Wenn die Partition zu Beginn  $n$  diskrete Elemente enthält, benötigt eine spätere Find-Operation höchstens  $O(\log n)$  Schritte, da die entstehenden Bäumen höchstens die Höhe  $\lfloor \log n \rfloor$  haben.

## Pfadkompression

Da die Kosten der Find-Operation von der Höhe der Bäume abhängen, wäre es günstig alle Knoten direkt an die Wurzel anzuhängen. Das würde die Kosten der Union-Operation aber drastisch erhöhen.

Stattdessen versucht man nun während der Find-Operation die Pfadlängen zu verkürzen. Diesen Vorgang nennt man **Pfadkompression**.

Eine Methode dies zu erreichen besteht darin, zuerst die Wurzel zu suchen und anschließend alle Knoten auf diesem Weg direkt an die Wurzel zu hängen.

### Prozedur (find mit Pfadkompression)

**Eingabe:** Eine Partition  $\mathcal{P}$  und ein Element  $x$

**Ausgabe:** Die Menge  $X \in \mathcal{P}$  mit  $x \in X$

$y := x, z := x$

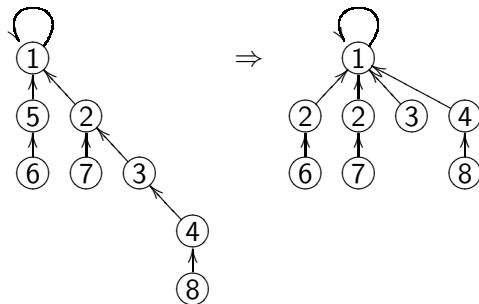
**solange**  $\text{pred}(y) \neq y$  **tue**  $y = \text{pred}(y)$

**solange**  $\text{pred}(z) \neq y$  **tue**  $t := z, z := \text{pred}(z), \text{pred}(t) := y$

Gebe  $y$  zurück

# Pfadkompression

`find(4)` würde damit z.B. zu folgender Veränderung führen:



## find mit Pfadkompression

Für nur eine Find-Operation erreichen wir damit keinen Gewinn. Allerdings bringt die Pfadkompression starke Einsparungen für spätere Operationen.

### Frage

*Wieviel bringt die Pfadkompression und die gewichtete Vereinigungsregel bei einer Folge von union- und find-Operationen, bzw. wieviel Zeit benötigt eine Folge von solchen Operationen?*

Um diese Frage beantworten zu können, führen wir eine neue Art der Analyse ein, die **Amortisierten Analyse**.

Teil 4: Verwaltung von Mengen  
Union-Find-Datenstrukturen

## Amortisierte Analyse

# Amortisierte Analyse

Ziel der **Amortisierten Analyse** ist es, die Laufzeit  $T(n)$  einer Folge von  $n$  Operationen zu ermitteln. Die **amortisierten Kosten** einer Operation ergeben sich dann als  $T(n)/n$ , d.h. als durchschnittliche Kosten einer Operation.

In vielen Fällen sind die so erhaltenen Schranken besser als die Schranken für den schlechtesten Fall, da dieser unter Umständen nicht sehr häufig auftritt.

Um eine amortisierte Analyse durchzuführen gibt es verschiedene Möglichkeiten. Wir werden für die Analyse der Union-Find-Datenstruktur mit Pfadkompression, Rängen und gewichteter Vereinigung, die so genannte **Potentialmethode** verwenden.

# Die Potentialmethode

Bei der **Potentialmethode** ordnen wir der betrachteten Datenstruktur  $D$  (in unserem Fall eine Union-Find-Datenstruktur) ein **Potential**  $\Phi(D)$  zu, das sich mit jeder Operation ändern kann.

Sei  $D_i$  die Datenstruktur nach der  $i$ -ten Operation. Die **amortisierten Kosten**  $\hat{c}_i$  der  $i$ -ten Operation definieren wir als:

$$\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$$

wobei  $c_i$  die tatsächlichen Kosten der Operation sind und  $\Phi(D_i) - \Phi(D_{i-1})$  die **Potentialänderung**.

Die gesamten amortisierten Kosten ergeben sich somit als

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0).$$

# Die Potentialmethode

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$$

Falls  $\Phi(D_n) \geq \Phi(D_0)$  gilt, sind die amortisierten Kosten somit eine obere Schranke für die tatsächlichen Kosten.

Damit man Aussagen über alle  $n$  erhält, fordert man häufig  $\Phi(D_i) \geq \Phi(D_0)$  für alle  $i$ . In der Regel setzt man außerdem  $\Phi(D_0) = 0$ .

Der Trick bei der amortisierten Analyse besteht nun darin, in einigen Schritten das Potential zu verringern und in anderen zu erhöhen. Damit **teure** Operationen aus dem Potential-Gewinn von **billigen** Operationen bezahlt werden können.

## Ein einfaches Beispiel: Stacks mit `multipop`

Wir betrachten einen erweiterten **Stack**  $\mathcal{S}$ , auf dem die üblichen Operationen `push` und `pop` definiert sind. Zusätzlich haben wir eine Funktion `multipop`, die eine natürliche Zahl als Argument erhält.

`multipop(k)` entfernt die obersten  $k$  Elemente vom Stack. Wenn der Stack weniger als  $k$  Elemente enthält, wird er komplett geleert.

Offensichtlich gilt `pop = multipop(1)`.

`push` benötigt offensichtlich  $O(1)$ , während `multipop(k)` im schlechtesten Fall  $O(k)$  benötigt.

Damit ergibt sich für eine Sequenz von  $n$  `push`-Operationen und  $l$  `multipop`-Operationen mit Argumenten  $k_i$  im schlechtesten Fall eine Laufzeit von  $O(n + \sum_{i=1}^l k_i)$ .

Dabei bleibt völlig unberücksichtigt, ob tatsächlich  $k_i$  Elemente auf dem Stack waren, oder nicht.

## Ein einfaches Beispiel: Stacks mit `multipop`

Eine amortisierte Analyse liefert ein genaueres Bild.

Das Potential  $\Phi(\mathcal{S}) := |\mathcal{S}|$  ist die **Anzahl der Elemente** auf dem Stack.

Damit hat eine `push`-Operation die amortisierten Kosten:

$$\hat{c} = 1 + (|\mathcal{S}| + 1) - |\mathcal{S}| = 1 + 1 = 2.$$

`multipop(k)` hat die amortisierten Kosten:

$$\begin{aligned}\hat{c} &= \min(k, |\mathcal{S}|) + \max(0, |\mathcal{S}| - k) - \mathcal{S} \\ &= \min(k, |\mathcal{S}|) + \max(-|\mathcal{S}|, -k) \\ &= \min(k, |\mathcal{S}|) - \min(|\mathcal{S}|, k) = 0\end{aligned}$$

Damit benötigt eine Sequenz von  $m$  `push`-Operationen und beliebigen `multipop`-Operationen höchstens  $O(2n) = O(n)$  Zeit.

Teil 4: Verwaltung von Mengen  
Union-Find-Datenstrukturen

# Analyse

## Lemma (Monotonie der Ränge)

Für jeden Knoten  $x$ , der keine Wurzel ist, gilt  $\text{rank}(x) < \text{rank}(\text{pred}(x))$ .

### Beweis

Wir führen eine strukturelle Induktion über die verschiedenen Operationen durch.

Für die leere Union-Find-Datenstruktur gilt die Behauptung offensichtlich.

Wir nehmen also an, dass die Behauptung für eine aus einer leeren Union-Find-Datenstruktur entstandenen Instanz  $D$  gilt.

Wird als nächstes `make_set(x)` ausgeführt, wird ein neuer Baum mit  $x$  als einzigem Knoten erzeugt. D.h. für alle Knoten außer  $x$  bleiben die Ränge unverändert. Für  $x$  hingegen gilt  $\text{rank}(x) = 0 = \text{rank}(\text{pred}(x))$  und  $x = \text{pred}(x)$ .

## Beweis (Monotonie: Fortsetzung)

Wird als nächstes  $\text{union}(a, b)$  ausgeführt, können wir der Einfachheit halber annehmen, dass  $a$  und  $b$  bereits die Wurzeln ihrer Bäume sind, und  $a$  an  $b$  angehängt wird. D.h. insbesondere  $\text{rank}(a) \leq \text{rank}(b)$ .

Falls  $\text{rank}(a) < \text{rank}(b)$ , verändert sich kein Rang und nur  $a$  wechselt den Vorgänger. Es gilt  $\text{rank}(a) < \text{rank}(b) = \text{rank}(\text{pred}(a))$ .

Falls  $\text{rank}(a) = \text{rank}(b)$ , wird nur der Rang von  $b$  um eins erhöht und nur  $a$  wechselt den Vorgänger und anschließend gilt

$\text{rank}(a) < \text{rank}(a) + 1 = \text{rank}(b) = \text{rank}(\text{pred}(a))$ .

Wird als nächstes  $\text{find}(a)$  ausgeführt, werden die Ränge nicht verändert. Aber die Knoten auf dem Weg von  $a$  zu seiner Wurzel  $b$  werden direkt an  $b$  gehängt. Dadurch gilt für jeden dieser Knoten  $z$ :

$$\text{rank}(a) \leq \text{rank}(z) < \text{rank}(b) = \text{rank}(\text{pred}(z)). \quad \square$$

## Lemma (Monotonie2)

Für die Ränge gilt weiterhin:

- 1 Die Ränge von Elementen auf dem Weg von  $x$  zu seiner Wurzel, steigen streng monoton.
- 2 Der Rang eines Knotens kann während einer Operation höchstens wachsen.
- 3  $\text{rank}(x) \leq n - 1$ .

## Beweis.

Die erste Behauptung folgt direkt aus dem vorhergegangenen Lemma. die zweite Behauptung ist eine Konsequenz aus der Tatsache, dass nur `union` den Rang verändert, d.h. erhöht.

Da jeder Knoten mit Rang 0 beginnt und maximal  $n - 1$  `union`-Operationen durchgeführt werden können (jede verringert die Anzahl der Bäume um 1), folgt eine obere Schranke von  $n - 1$  für jeden Knoten. □

# Eine schnell wachsende Funktion

Bevor wir das Potential definieren können, müssen wir eine sehr schnell wachsende Funktion einführen.

## Definition

Für  $k \geq 0$  und  $j \geq 1$  definieren wir:

$$A_k(j) := \begin{cases} j + 1 & \text{falls } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{falls } k \geq 1 \end{cases}$$

wobei für eine beliebige Funktion  $f$  gilt:

$$f^{(0)}(x) = x \text{ und } f^{(j)}(x) = f\left(f^{(j-1)}(x)\right),$$

d.h.  $f^{(j)}(x)$  ist die  $j$ -fache Anwendung von  $f$  auf  $x$ .

## Lemma (Monotonie der $A_k(j)$ )

- 1 Für jedes  $k \geq 0$  ist  $A_k(j)$  streng monoton steigend in  $j$ .
- 2 Für jedes  $j \geq 0$  ist  $A_k(j)$  streng monoton steigend in  $k$ .

### Beweis.

Für die erste Behauptung führen wir eine vollständige Induktion über  $k \geq 0$  durch.

Für  $k = 0$  gilt  $A_0(j + 1) = j + 2 = A_0(j) + 1$ .

Für  $k > 0$  gilt

$$A_k(j + 1) = A_{k-1}^{(j+2)}(j + 1) > A_{k-1}^{(j+1)}(j + 1) > A_{k-1}^{(j+1)}(j) = A_k(j).$$

Die zweite Behauptung können wir jetzt direkt beweisen:

$$A_{k+1}(j) = A_k^{(j+1)}(j) > A_k(j).$$



## Eine langsam wachsende Funktion

Diese Familie von Funktionen wächst sehr schnell. Ihr „Inverses“

$$\alpha(n) := \min \{k \mid A_k(1) \geq n\}$$

wächst sehr langsam. Durch Einsetzen erhält man z.B.

$$\alpha(n) = \begin{cases} 0 & \text{für } 0 \leq n \leq 22 \\ 1 & \text{für } n = 3 \\ 2 & \text{für } 4 \leq n \leq 7 \\ 3 & \text{für } 8 \leq n \leq 2047 \\ 4 & \text{für } 2048 \leq n \leq A_4(1) \end{cases}$$

Dabei ist  $A_4(1) \gg 2^{2048} \gg 10^{80}$ , d.h. für alle „vernünftigen“ Werte können wir  $\alpha(n) \leq 10$  annehmen.

„Inverse“ weil

$$\alpha(A_n(1)) = \min\{k \mid A_k(1) \geq A_n(1)\} = n.$$

## level und iter

Neben  $\alpha$  benötigen wir zwei weitere Funktionen, die knotenweise definiert sind.

$$\text{level}(x) := \max \{k \mid \text{rank}(\text{pred}(x)) \geq A_k(\text{rank}(x))\}$$

$$\text{iter}(x) := \max \left\{ i \mid \text{rank}(\text{pred}(x)) \geq A_{\text{level}(x)}^{(i)}(\text{rank}(x)) \right\}$$

$\text{level}(x)$  ist ein Maß für den Rangunterschied zwischen  $x$  und  $\text{pred}(x)$ . Dabei wird das Spektrum der Ränge oberhalb von  $\text{rank}(x)$  in Gruppen wachsender Größe eingeteilt.  $\text{level}(x)$  gibt an, in welcher Gruppe  $\text{rank}(\text{pred}(x))$ , relativ zu  $\text{rank}(x)$  liegt.

Wie wir sehen werden, unterteilt  $\text{iter}(x)$  die Gruppen in  $\text{rank}(x)$  kleinere Abschnitte und verfeinert so die Unterscheidung.

## level

$$\text{level}(x) := \max \{k \mid \text{rank}(\text{pred}(x)) \geq A_k(\text{rank}(x))\}$$

### Lemma

Ist  $x$  keine Wurzel und  $\text{rank}(x) \geq 1$ , so gilt

$$0 \leq \text{level}(x) < \alpha(n)$$

### Beweis

$\text{rank}(\text{pred}(x)) \geq \text{rank}(x) + 1$     *Der Rang des Vorgängers ist echt größer*  
 $= A_0(\text{rank}(x))$     *Definition von  $A_0(j)$*

was  $\text{level}(x) \geq 0$  beweist. Weiter haben wir

$$\begin{aligned} A_{\alpha(n)}(\text{rank}(x)) &\geq A_{\alpha(n)}(1) && A_k(j) \text{ ist streng monoton steigend} \\ &\geq n && \text{Definition von } \alpha(n) \\ &> \text{rank}(p(x)) && n - 1 \geq \text{rank}(x) \end{aligned}$$

## iter

$$\text{iter}(x) := \max \left\{ i \mid \text{rank}(\text{pred}(x)) \geq A_{\text{level}(x)}^{(i)}(\text{rank}(x)) \right\}$$

### Lemma

*Ist  $x$  keine Wurzel und  $\text{rank}(x) \geq 1$ , so gilt*

$$1 \leq \text{iter}(x) \leq \text{rank}(x).$$

### Beweis

*Die Definition von  $\text{level}(x)$  impliziert*

$$\text{rank}(\text{pred}(x)) \geq A_{\text{level}(x)}(\text{rank}(x))$$

*was  $\text{iter}(x) \geq 1$  beweist. Weiter haben wir*

$$\begin{aligned} A_{\text{level}(x)+1}^{(\text{rank}(x)+1)}(\text{rank}(x)) &= A_{\text{level}(x)+1}(\text{rank}(x)) && \text{Definition von } A_k(j) \\ &> \text{rank}(\text{pred}(x)) && \text{Definition von } \text{level}(x) \end{aligned}$$

# Das Potential

Jetzt können wir das Potential definieren.

Das Gesamtpotential  $\Phi(D)$  ist die Summe von **Elementpotentialen**, d.h.

$$\Phi(D) := \sum_{x \in D} \Phi(x)$$

mit

$$\Phi(x) := \begin{cases} \alpha(n) \text{rank}(x) & \text{falls } x \text{ eine Wurzel ist} \\ & \text{oder } \text{rank}(x) = 0 \\ (\alpha(n) - \text{level}(x)) \text{rank}(x) - \text{iter}(x) & \text{sonst} \end{cases}$$

Um den zeitlichen Verlauf beschreiben zu können, versehen wir alle Bezeichnungen mit einem Index  $q$ , um die Situation nach  $q$  Operationen zu beschreiben.

# Das Potential

## Lemma (Potentialschranken)

Für jede Datenstruktur  $D$  und jedes Element  $x$  gilt

$$0 \leq \Phi(x) \leq \alpha(n)\text{rank}(x)$$

## Beweis

Falls  $x$  eine Wurzel ist oder  $\text{rank}(x) = 0$  gilt, ist die Behauptung per Definition korrekt.

Nehmen wir nun an, dass  $x$  keine Wurzel ist und  $\text{rank}(x) \geq 1$ . Dann erhalten wir eine untere Schranke für  $\Phi(x)$ , indem wir  $\text{level}(x)$  und  $\text{iter}(x)$  maximieren.

Aus  $\text{level}(x) \leq \alpha(n) - 1$  und  $\text{iter}(x) \leq \text{rank}(x)$  folgt

$$\Phi(x) \geq (\alpha(n) - (\alpha(n) - 1)) \text{rank}(x) - \text{rank}(x) = \text{rank}(x) - \text{rank}(x) = 0$$

# Das Potential

## Beweis (Potentialschranken: Fortsetzung)

*Auf ähnliche Weise erhalten wir eine obere Schranke, indem wir  $\text{level}(x)$  und  $\text{iter}(x)$  minimieren. Mit  $\text{level}(x) \geq 0$  und  $\text{iter}(x) \geq 1$  gilt somit*

$$\Phi(x) \leq (\alpha(n) - 0) \text{rank}(x) - 1 = \alpha(n) \text{rank}(x) - 1$$



Jetzt können wir die Veränderung des Potentials bei verschiedenen Operationen untersuchen.

Dazu betrachten wir die Potentiale  $\Phi_q(x)$  eines einzelnen Knoten nach  $q$  Operationen.

## Lemma (Amortisierte Kosten von `make_set`)

*Die amortisierten Kosten einer `make_set`-Operation sind  $O(1)$ .*

### Beweis.

Wir nehmen an, dass die  $q$ -te Operation `make_set(x)` ist.

Damit wird ein Knoten mit Rang 0 erzeugt, so dass  $\Phi_q(x) = 0$  gilt.

Alle anderen Ränge und Potentiale verändern sich nicht, so dass  $\Phi_q(D) = \Phi_{q-1}(D)$  gilt.

Die tatsächlichen Kosten von `make_set(x)` sind  $O(1)$ , und somit

$$\hat{c}_q = O(1) + \Phi_q(D) - \Phi_{q-1}(D) = O(1).$$



## Lemma (Sinkendes Potential)

*x* sei keine Wurzel und die *q*-te Operation sei entweder union oder find. Dann gilt

$$\Phi_q(x) \leq \Phi_{q-1}(x).$$

Falls  $\text{rank}(x) \geq 1$  und entweder  $\text{level}(x)$  oder  $\text{iter}(x)$  durch die *q*-te Operation geändert wird, gilt sogar

$$\Phi_q(x) \leq \Phi_{q-1}(x) - 1.$$

## Beweis

Da *x* keine Wurzel ist, wird  $\text{rank}(x)$  weder durch eine union noch eine find Operation geändert.

Falls  $\text{rank}(x) = 0$ , bleibt das Potential gleich.

Falls  $\text{rank}(x) \geq 1$ , kann  $\text{level}(x)$  höchstens steigen, denn die Ränge steigen.

## Beweis (Sinkende Potential: Fortsetzung)

Falls *sich weder  $\text{level}(x)$  noch  $\text{iter}(x)$  ändern*, gilt  $\phi_q(x) = \Phi_{q-1}(x)$ .

Falls *sich nur  $\text{iter}(x)$  verändert*, muss es um mindestens 1 steigen, denn  $\text{level}(x)$  bleibt gleich, aber der Rang des Vorgängers steigt. Die Monotonie von  $A_{\text{level}(x)}$  erzwingt dann  $\text{iter}_q(x) > \text{iter}_{q-1}(x)$  und somit  $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$ .

Wenn  $\text{level}(x)$  wächst, sinkt der Term  $(\alpha(n) - \text{level}(x))\text{rank}(x)$  um mindestens  $\text{rank}(x)$ .

Gleichzeitig kann  $\text{iter}(x)$  sinken, allerdings um höchstens  $\text{rank}(x) - 1$ . Damit ist die Erhöhung des Potentials durch  $\text{iter}(x)$  auf jeden Fall echt kleiner als die Senkung durch  $\text{level}(x)$  und somit

$$\Phi_q(x) \leq \Phi_{q-1}(x) - 1.$$



## Lemma (Amortisierte Kosten von union)

*Die amortisierten Kosten einer union-Operation sind  $O(\alpha(n))$ , sofern die Argumente direkt die Wurzeln der jeweiligen Bäume sind.*

### Beweis

*Wir nehmen an, dass  $\text{union}(x, y)$  die  $q$ -te Operation ist, wobei  $x$  und  $y$  Wurzeln sind (d.h. die find-Operationen in union benötigen  $O(1)$ ).*

*Weiter nehmen wir an, dass  $x$  an  $y$  angehängt wird.*

*Offensichtlich ändern sich nur die Potentiale von Knoten in den beiden Bäumen von  $x$  und  $y$ .*

*Das Lemma [Sinkendes Potential] zeigt, dass das Potential jedes Knoten, der keine Wurzel ist, nicht steigt.*

*Da  $x$  vor der  $q$ -ten Operation eine Wurzel war, galt*

$$\Phi_{q-1}(x) = \alpha(n)\text{rank}(x).$$

## Beweis (Amortisierte Kosten von union: Fortsetzung)

Falls nach der Operation  $\text{rank}(x) = 0$  gilt, ergibt sich

$\Phi_q(x) = \Phi_{q-1}(x) = 0$ . Sonst gilt

$$\Phi_q(x) = (\alpha(n) - \text{level}(x)) \text{rank}(x) - \text{iter}(x) < \alpha(n) \text{rank}(n).$$

Da  $y$  eine Wurzel war, gilt  $\Phi_{q-1}(y) = \alpha(n) \text{rank}(y)$ . Da  $y$  eine Wurzel bleibt, erhöht die union-Operation  $\text{rank}(y)$  um eins oder ändert sie nicht. Dadurch gilt  $\Phi_q(y) = \Phi_{q-1}(y)$  oder  $\Phi_q(y) = \Phi_{q-1}(y) + \alpha(n)$

Insgesamt folgt somit, dass lediglich das Potential von  $y$  erhöht wird. Das Potential aller anderen Knoten sinkt höchstens.

Damit erhöht sich das Potential höchstens um  $\alpha(n)$ , der maximalen Erhöhung für  $y$ .

Die amortisierten Kosten einer union-Operation *mit Wurzeln als Argument* sind somit  $O(1) + \alpha(n) = O(\alpha(n))$ . □

## Lemma (Amortisierte Kosten von find)

Die amortisierten Kosten einer find-Operation sind  $O(\alpha(n))$ .

### Beweis

Wir nehmen an, dass die  $q$ -te Operation eine find-Operation ist und dass der Pfad  $s$  Knoten enthält. Damit sind die tatsächlichen Kosten der Operation  $O(s)$ .

Wir werden zeigen, dass das Potential keines Knotens steigt und das bei mindestens  $\max(0, s - (\alpha(n) + 2))$  Knoten auf dem Pfad das Potential sinkt.

Offensichtlich ändern sich nur die Potentiale im betroffenen Baum.

Wie oben gesehen steigt höchstens das Potential der Wurzel. Dieses ist aber  $\alpha(n)\text{rank}(x)$  und ändert sich somit nicht.

## Beweis (Amortisierte Kosten von find: Fortsetzung)

Sei  $x$  ein Knoten auf dem Pfad, so dass  $\text{rank}(x) > 0$  und es existiert ein  $y$  „oberhalb“ von  $x$  auf dem Pfad der keine Wurzel ist und mit  $\text{level}(x) = \text{level}(y)$  vor der find-Operation.

Die Knoten auf dem Weg, die diese Bedingung **nicht** erfüllen, sind

- 1 die Wurzel
- 2 der erste Knoten auf dem Weg (falls er Rang 0 hat)
- 3 und der jeweils letzte Knoten  $w$  auf dem Pfad mit  $\text{level}(w) = k$  für jedes  $k = 0, 1, \dots, \alpha(n) - 1$ .

D.h. wir haben insgesamt höchstens  $\alpha(n) + 2$  Knoten, die die Bedingung nicht erfüllen. Damit bleiben mindestens  $\max(0, s - \alpha(n) - 2)$  Knoten, die sie erfüllen.

## Beweis (Amortisierte Kosten von find: Fortsetzung)

Sei nun  $x$  ein solcher Knoten und  $y$  der geforderte Knoten mit  $\text{level}(x) = \text{level}(y) = k$  und  $i := \text{iter}(x)$ .

Direkt vor der Pfadkompression haben wir:

$$\text{rank}(\text{pred}(x)) \geq A_k^{(i)}(\text{rank}(x)) \quad \text{per Definition von } i = \text{iter}(x)$$

$$\text{rank}(\text{pred}(y)) \geq A_k(\text{rank}(y)) \quad \text{per Definition von level}(y)$$

$$\text{rank}(y) \geq \text{rank}(\text{pred}(x)) \quad \text{da die Ränge monoton steigen}$$

Damit ergibt sich

$$\begin{aligned} \text{rank}(\text{pred}(y)) &\geq A_k(\text{rank}(y)) \geq A_k(\text{rank}(\text{pred}(x))) \\ &\geq A_k(A_k^{(i)}(\text{rank}(x))) = A_k^{(i+1)}(\text{rank}(x)) \end{aligned}$$

## Beweis (Amortisierte Kosten von find: Fortsetzung)

*Nach der Pfadkompression haben  $x$  und  $y$  den gleichen Vorgänger und somit*

$$\text{rank}(\text{pred}(x)) = \text{rank}(\text{pred}(y)).$$

*Gleichzeitig senkt die Pfadkompression  $\text{rank}(\text{pred}(y))$  nicht.*

*Da sich  $\text{rank}(x)$  nicht ändert, haben wir nach der Kompression*

$$\text{rank}(\text{pred}(x)) \geq A_k^{(i+1)}(\text{rank}(x)).$$

*Damit muss entweder  $\text{level}(x)$  oder  $i = \text{iter}(x)$  erhöht werden und nach Lemma [Sinkendes Potential] ergibt sich:*

$$\Phi_q(x) \leq \Phi_{q-1}(x) - 1$$

*Damit sinkt das Gesamtpotential um mindestens  $\max(0, s - \alpha(n) - 2)$ .*

## Beweis (Amortisierte Kosten von find: Fortsetzung)

Damit ergeben sich die amortisierten Kosten der find-Operation als:

$$\begin{aligned} O(s) + \Phi(D_q) - \Phi(D_{q-1}) &\leq O(s) - \max(0, s - \alpha(n) - 2) \\ &\leq O(s) - s + \alpha(n) + 2 \\ &= O(\alpha(n)). \end{aligned}$$

Letzteres ergibt sich dadurch, dass das Potential einfach mit der in  $O(s)$  versteckten Konstante multipliziert werden kann. □

## Satz (Die Kosten einer Sequenz von Union-Find-Operationen)

*Eine Sequenz von  $m$  `make_set`, `union` und `find` Operationen mit insgesamt  $n$  `make_set`-Operationen, benötigt höchstens Zeit  $O(m\alpha(n))$ .*

### Beweis.

Wir haben gesehen, dass die amortisierten Kosten jeder Operation  $O(\alpha(n))$  sind. Dabei ist aber zu beachten, dass wir für `union` gefordert haben, dass die Argumente bereits Wurzeln sind. Ist dies nicht der Fall, wird eine `union`-Operation durch insgesamt 3 Operationen mit jeweils amortisierten Kosten  $O(\alpha(n))$  ersetzt. Dadurch ändert sich die Summe um höchstens eine Konstante. □

Teil 4: Verwaltung von Mengen  
Union-Find-Datenstrukturen

# Varianten der Pfadverkürzung

## Varianten der Pfadverkürzung

Bei der vorgestellten Methode zur Verkürzung der Pfade während der `find`-Operation, wird der Weg insgesamt zweimal durchlaufen. Einmal zum Finden der Wurzel und ein zweites Mal zum Umhängen der Knoten auf dem Weg.

Alternative Verfahren versuchen dieses zweimalige Durchlaufen zu verhindern.

Im Folgenden stellen wir zwei von ihnen vor.

# Die Aufteilungsmethode

Während der Ausführung der `find`-Operation teilen wir den Suchpfad in zwei Pfade ungefähr gleicher Länge auf. Dazu hängen wir jeden Knoten, außer dem letzten und vorletzten, an seinen **Großvater** um.

## Prozedur (`find` mit Aufteilungsmethode)

**Eingabe:** *Eine Partition  $\mathcal{P}$  und ein Element  $x$*

**Ausgabe:** *Die Menge  $X \in \mathcal{P}$  mit  $x \in X$*

$y := x$

**solange**  $\text{pred}(\text{pred}(y)) \neq \text{pred}(y)$  **tue**

$t := y$

$y := \text{pred}(y)$

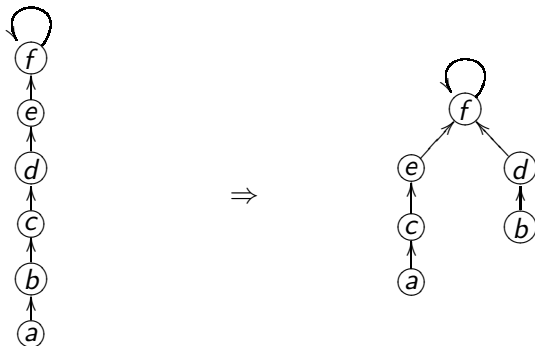
$\text{pred}(t) := \text{pred}(y)$

**Ende**

*Gebe  $y$  zurück*

# Die Aufteilungsmethode

`find(a)` führt zu folgender Veränderung:



# Die Halbierungsmethode

Bei der Halbierungsmethode geht man etwas anders vor.

Es wird nicht jeder Knoten an seinen Großvater gehängt, sondern nur jeder zweite, beginnend mit dem gesuchten Knoten.

## Prozedur (find mit Halbierungsmethode)

**Eingabe:** Eine Partition  $\mathcal{P}$  und ein Element  $x$

**Ausgabe:** Die Menge  $X \in \mathcal{P}$  mit  $x \in X$

$y := x$

**solange**  $\text{pred}(\text{pred}(y)) \neq \text{pred}(y)$  **tue**

$t := \text{pred}(\text{pred}(y))$

$\text{pred}(y) := t$

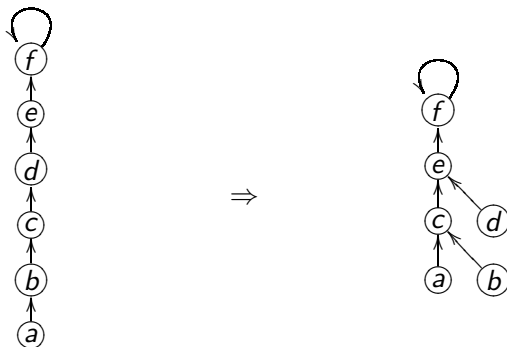
$y := t$

**Ende**

Gebe  $y$  zurück

# Die Halbierungsmethode

`find(a)` führt zu folgender Veränderung:



## Untere Schranken

Es bleibt die Frage, ob durch geschickte Verkürzungsstrategien lineare amortisierte Laufzeit erreicht werden kann?

Tarjan konnte zeigen, dass es für eine große Klasse von Techniken nicht möglich ist amortisierte lineare Kosten zu erreichen.

Tarjan konnte sogar zeigen, dass  $m\hat{\alpha}(m, n)$  eine **optimale untere Schranke** für die amortisierten Kosten von  $m$  find- und  $n$  union-Operationen ist. Dabei ist  $\hat{\alpha}$  die Inverse der **Ackermann-Funktion**, einer sehr schnell wachsenden, nicht primitiv rekursiven Funktion.