Karlsruher Institut für Technologie

2. Zusatztutorium Datenstrukturen

Algorithmen I, Zusatztutorium

Henriette Färber, Tobias Knorr | 8. August 2022

FAKULTÄT FÜR INFORMATIK

Roadmap

Roadmap

Motivation

Was wollen wir?

Daten speichern!

Und was noch?

Motivation

Was wollen wir?

Daten speichern!

Und was noch?

- Einfügen (an beliebiger Stelle)
- Löschen (an beliebiger Stelle)
- Suchen
- Sortieren
- Abändern (an beliebiger Stelle)
- Wahlfreier Zugreifen
- ...

Operationen auf Datenstrukturen

Wie interagieren wir mit einer Datenstruktur?

- insert, pushBack, pushFront
- remove, popBack, popFront
- first, last
- size
- findNext
- concat, splice
- …

Listen

Listen bilden eine Aneinanderreihung einzelner Elemente ab.

Eigenschaften

Vorteile:

- Kapazität (prinzipiell) uneingeschränkt
- Zugriff auf Vorgänger- und Nachfolger-Element in $\Theta(1)$
- Erweiterung durch Einfügen am Ende in $\Theta(1)$

Nachteile:

- Zugriff auf ein beliebiges Listenelement in nur $\Theta(n)$
- zusätzlicher Speicheraufwand pro Element für Zeiger
- nicht cache-effizient, Elemente sind i.d.R. nicht benachbart

Doppelt verkettete Listen (DLLs)

Aufbau

Class Item **of** ListElement

e: ListElement

next: Handle

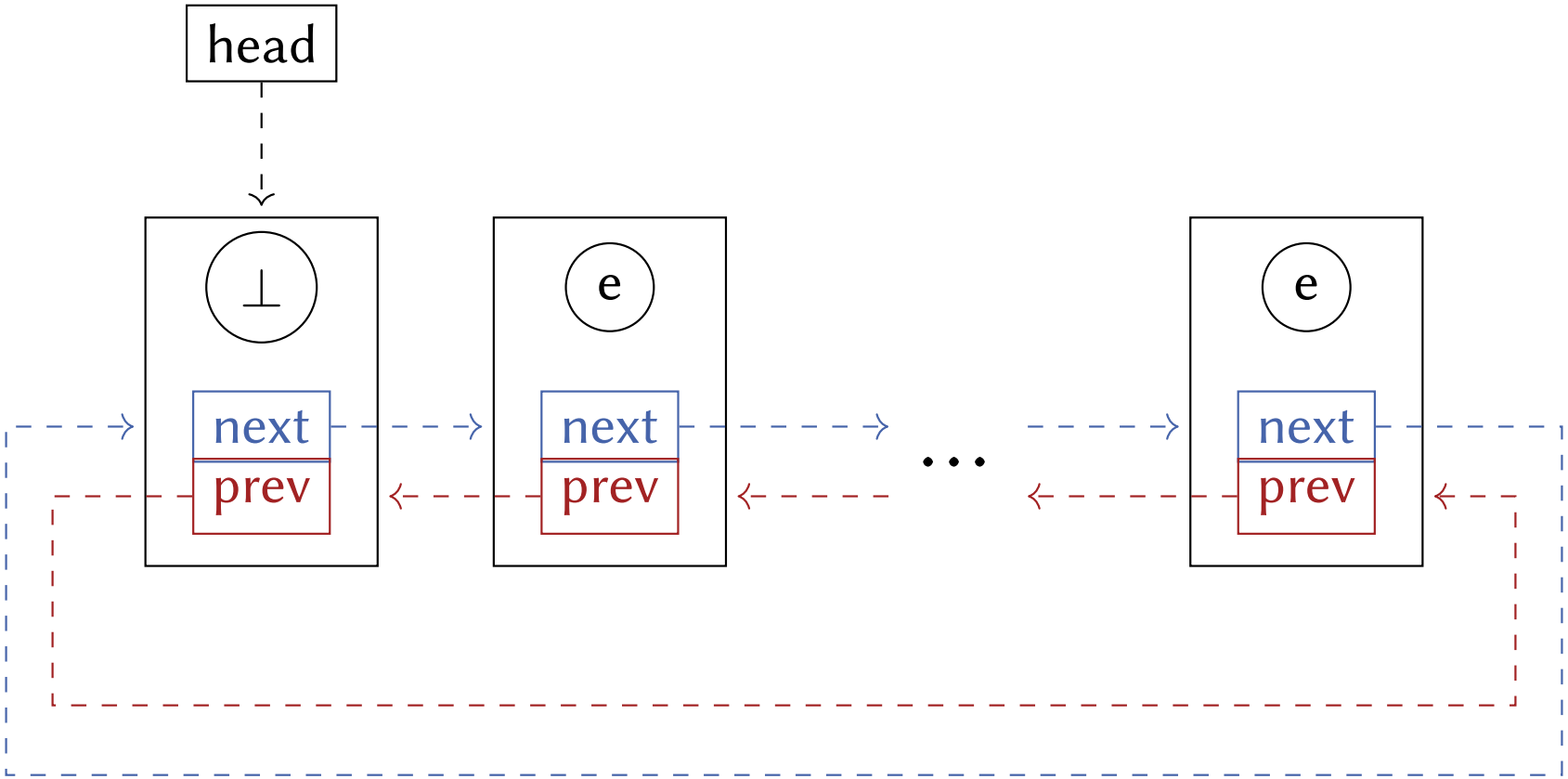
prev: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger
- Zeiger auf Vorgänger

Als «Einstiegspunkt» gibt es einen head-Knoten, oft auch als Dummy-Header oder Wächter-Element bezeichnet.

Doppelt verkettete Listen (DLLs)



Der Dummy-Header

Vor- und Nachteile Dummy-Header

+ Invarianten immer erfüllt
+ Vermeidung von Sonderfällen
+ deswegen auch lesbarer in der Programmierung
- verbraucht Speicherplatz

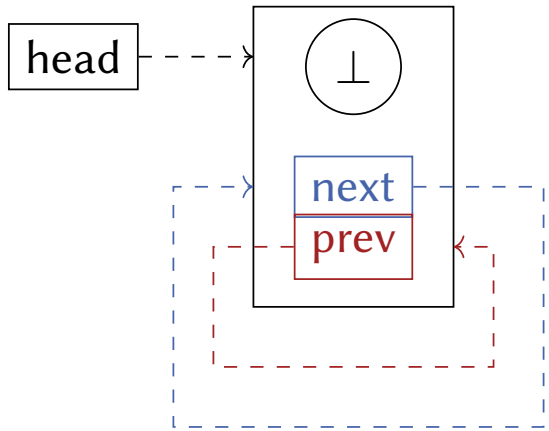
Der Dummy-Header

Vor- und Nachteile Dummy-Header

+ Invarianten immer erfüllt
+ Vermeidung von Sonderfällen
+ deswegen auch lesbarer in der Programmierung
- verbraucht Speicherplatz

Invariante für Listenelement e

e.next.prev = e.prev.next = e



Einfach verkettete Listen (SLLs)

Aufbau

Class Item **of** ListElement

e: ListElement

next: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger

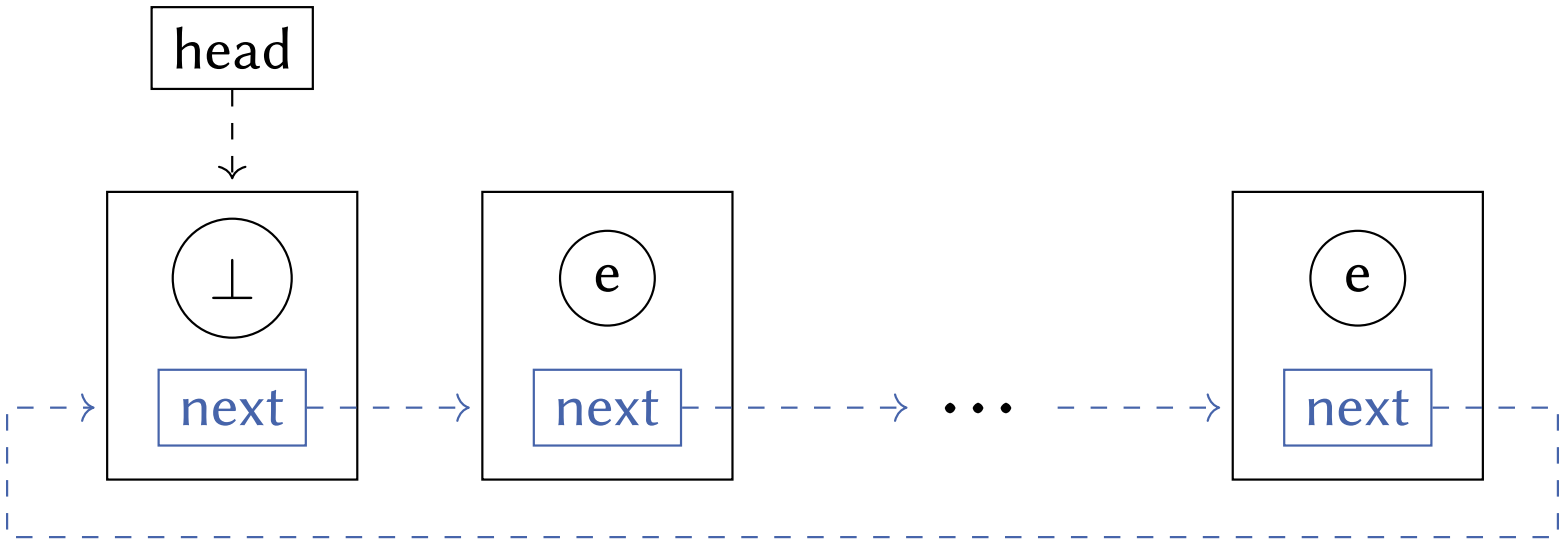
Einfach verkettete Listen (SLLs)

Aufbau

Class Item **of** ListElement
    e: ListElement
    next: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger



Ein zusätzlicher Zeiger auf das letzte Element erlaubt `pushBack`-Operationen in $\Theta(1)$.

Einfach verkettete Listen (SLLs)

Aufbau

Class Item **of** ListElement
    e: ListElement
    next: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger

Vor- und Nachteile

+ verbraucht weniger Speicher
+ Speichergewinn oftmals auch Zeitgewinn
- Operationen, die auf DLLs einfach sind, können hier kompliziert (und langsam) werden
- hat zusätzlich die Nachteile einer DLL

Beschränkte Felder (Arrays)

Arrays bilden Blöcke allozierten Speichers von fester Größe.

Eigenschaften

- Größe wird bei der Initialisierug festgelegt
- jedes Element ist eindeutig über Index adressierbar

Vorteile:

Beschränkte Felder (Arrays)

Arrays bilden Blöcke allozierten Speichers von fester Größe.

Eigenschaften

- Größe wird bei der Initialisierug festgelegt
- jedes Element ist eindeutig über Index adressierbar

Vorteile:

- effizienter Zugriff auf jedes Element in $\Theta(1)$
- cache-effizient, da zusammenhängender Speicherblock

Nachteile:

Beschränkte Felder (Arrays)

Arrays bilden Blöcke allozierten Speichers von fester Größe.

Eigenschaften

- Größe wird bei der Initialisierug festgelegt
- jedes Element ist eindeutig über Index adressierbar

Vorteile:

- effizienter Zugriff auf jedes Element in $\Theta(1)$
- cache-effizient, da zusammenhängender Speicherblock

Nachteile:

- beschränkte Kapazität
- kein effizientes Einfügen an einer beliebigen Position

Unbeschränkte Felder (Arrays)

Arrays, aber ohne die Nachteile von Arrays (?)

Eigenschaften

Gewünschte Operationen:

- `pushBack()`: fügt ein Element am Ende des Feldes hinzu
- `popBack()`: entfernt das letzte Element des Feldes

Umsetzung

Unbeschränkte Felder (Arrays)

Arrays, aber ohne die Nachteile von Arrays (?)

Eigenschaften

Gewünschte Operationen:

- `pushBack()`: fügt ein Element am Ende des Feldes hinzu
- `popBack()`: entfernt das letzte Element des Feldes

Umsetzung

- verwende beschränktes Feld, reallokiere ggf.
- Kein Platz?
  Kopiere alle Einträge in ein neues Feld um, welches um einen konstanten Faktor **größer** ist
- Zu viel ungenutzter Platz?
  Kopiere alle Einträge in ein neues Feld um, welches um einen konstanten Faktor **kleiner** ist

Unbeschränkte Felder (Arrays)

```
1: struct Array {
2:         capacity: ℕ_0
3:         size: ℕ_0
4:         data: [Element; capacity]
```

Unbeschränkte Felder (Arrays)

```
1:  struct Array {
2:      capacity: ℕ₀
3:      size: ℕ₀
4:      data: [Element; capacity]

5:  pushBack(e: Element)
6:      if size = capacity then
7:          new_data : [Element; 2 · capacity]
8:          for i ∈ {0, ..., size − 1} do
9:              new_data[i] := data[i]
10:         capacity := 2 · capacity
11:         data := new_data
12:     data[size] := e
13:     size := size + 1
14: }
```

Unbeschränkte Felder (Arrays)

```
1:  struct Array {
2:      capacity : ℕ_0
3:      size : ℕ_0
4:      data : [Element; capacity]

5:  popBack( )
6:      size := size − 1
7:      if size ≤ ⌊capacity/4⌋ then
8:          new_data : [Element; ⌊capacity/2⌋]
9:          for i ∈ {0, ..., size − 1} do
10:             new_data[i] := data[i]
11:         capacity := ⌊capacity/2⌋
12:         data := new_data
13: }
```

Unbeschränkte Felder (Arrays)

Was haben wir damit gewonnen?

Laufzeiten

Sowohl für `pushBack` als auch für `popBack` gilt:

- Liegt `size` im tolerierten Bereich[1]?
  $\Rightarrow$ konstante Laufzeit
- Ansonsten: lineare Laufzeit durch Umkopieren

Aber: Sowohl `pushBack` als auch `popBack` haben amortisiert konstante Laufzeit.

[1] in Abhängigkeit von `capacity` und den gewählten Faktoren

Vergleich: Listen vs Felder

Vorteile von Listen

- flexibel
- kein ungenutzter Speicherplatz
- effizientes Einfügen von Elementen an eine beliebige Position

Vergleich: Listen vs Felder

Vorteile von Listen

- flexibel
- kein ungenutzter Speicherplatz
- effizientes Einfügen von Elementen an eine beliebige Position

Vorteile von Feldern

- effizienter Zugriff auf beliebige Elemente
- kein Overhead für Zeiger
- Cache-effizientes iterieren
- simpel in Handhabung und Implementierung

Vergleich: Listen vs Felder

Nachteile von Listen

- Zeiger benötigen zusätzlichen Speicher
- kein effizienter beliebiger Zugriff
- komplizierter in Handhabung und Impelementierung

Vergleich: Listen vs Felder

Nachteile von Listen

- Zeiger benötigen zusätzlichen Speicher
- kein effizienter beliebiger Zugriff
- komplizierter in Handhabung und Impelementierung

Nachteile von Feldern

- kein effizientes Einfügen von Elemente an eine beliebiger Position
- ungenutzter Speicherplatz
- einzelne Operationen haben nur amortisiert gute Laufzeit
- nicht sehr flexibel

Kosten der Operationen

Operation	DLL	SLL	Array	Erklärung ($*$)
first	1	1	1	
last	1	1	1	
insert	1	1*	n	nur insertAfter
remove	1	1*	n	nur removeAfter
pushBack	1	1	1*	amortisiert
pushFront	1	1	n	
popBack	1	n	1*	amortisiert
popFront	1	1	n	
concat	1	1	n	
splice	1	1	n	
findNext	n	n	n	

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- stark limitiertem Speicherplatz?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- stark limitiertem Speicherplatz?
- Iteration über alle Elemente?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- stark limitiertem Speicherplatz?
- Iteration über alle Elemente?
- Vereinen zweier Folgen?

Alles klar soweit?

Welche der gerade behandelten Datenstrukturen sollte man wählen bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- stark limitiertem Speicherplatz?
- Iteration über alle Elemente?
- Vereinen zweier Folgen?
- direktem Zugriff auf einzelne Elemente?

Roadmap

Motivation

Was wollen wir? Daten speichern!

Und was noch?

- Elemente in Teilmengen unterteilen
- Teilmengen verwalten können

Motivation

Was wollen wir? Daten speichern!

Und was noch?

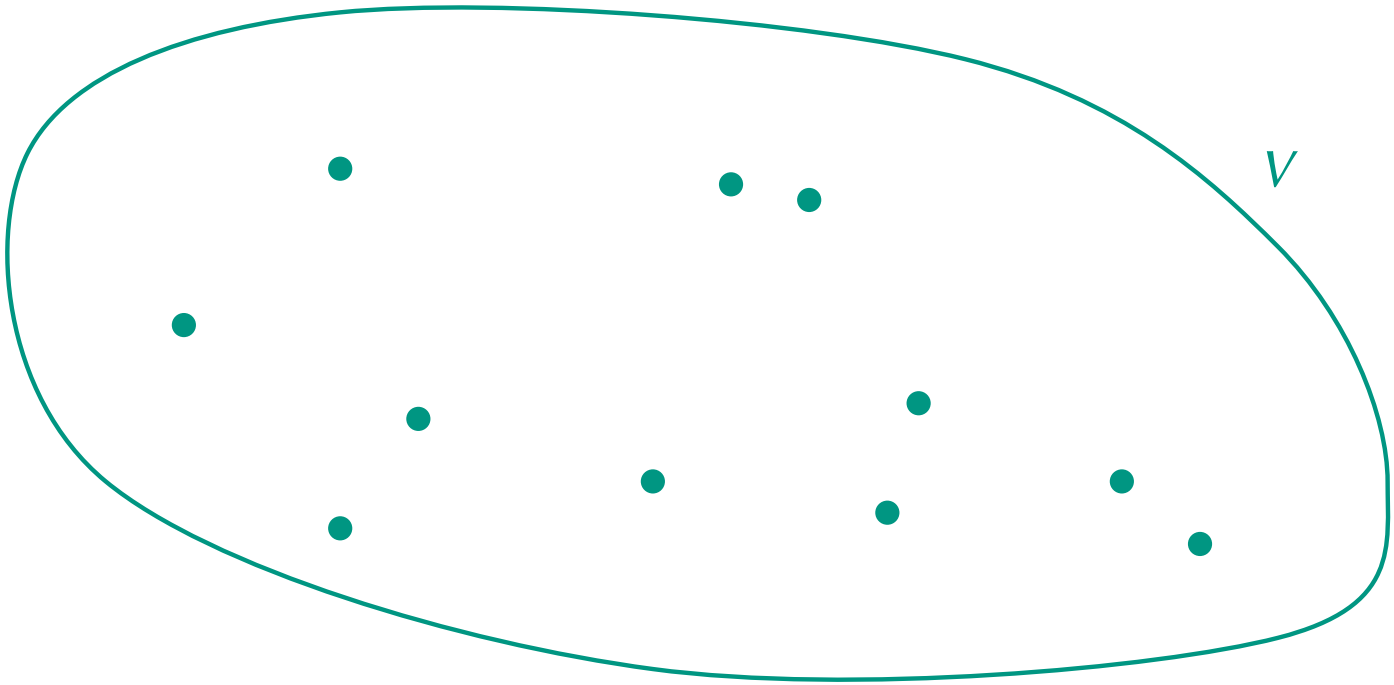
- Elemente in Teilmengen unterteilen
- Teilmengen verwalten können

Beispiel:
Algorithmus von Kruskal fügt Teilbäume eines Graphen zu MST zusammen

$\Rightarrow$ Wie müssen effizient herausfinden können, ob gegebene Kante verschiedene Teilbäume verbindet

Union-Find Datenstruktur

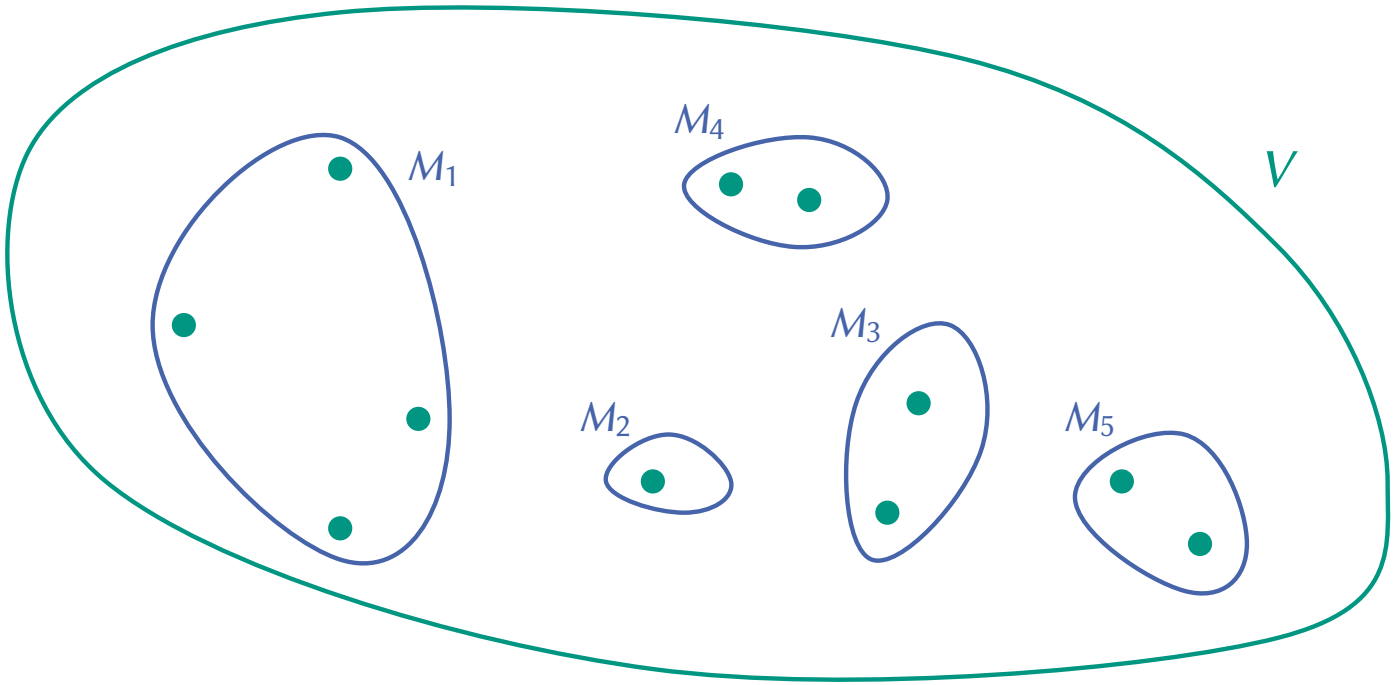
Eine endliche Menge $V = \{1 \dots n\}$



Union-Find Datenstruktur

Eine endliche Menge $V = \{1 \dots n\}$ sei in disjunkte Klassen M_i partitioniert:

$$V = M_1 \cup \cdots \cup M_k \qquad \forall i \neq j : M_i \cap M_j = \emptyset$$

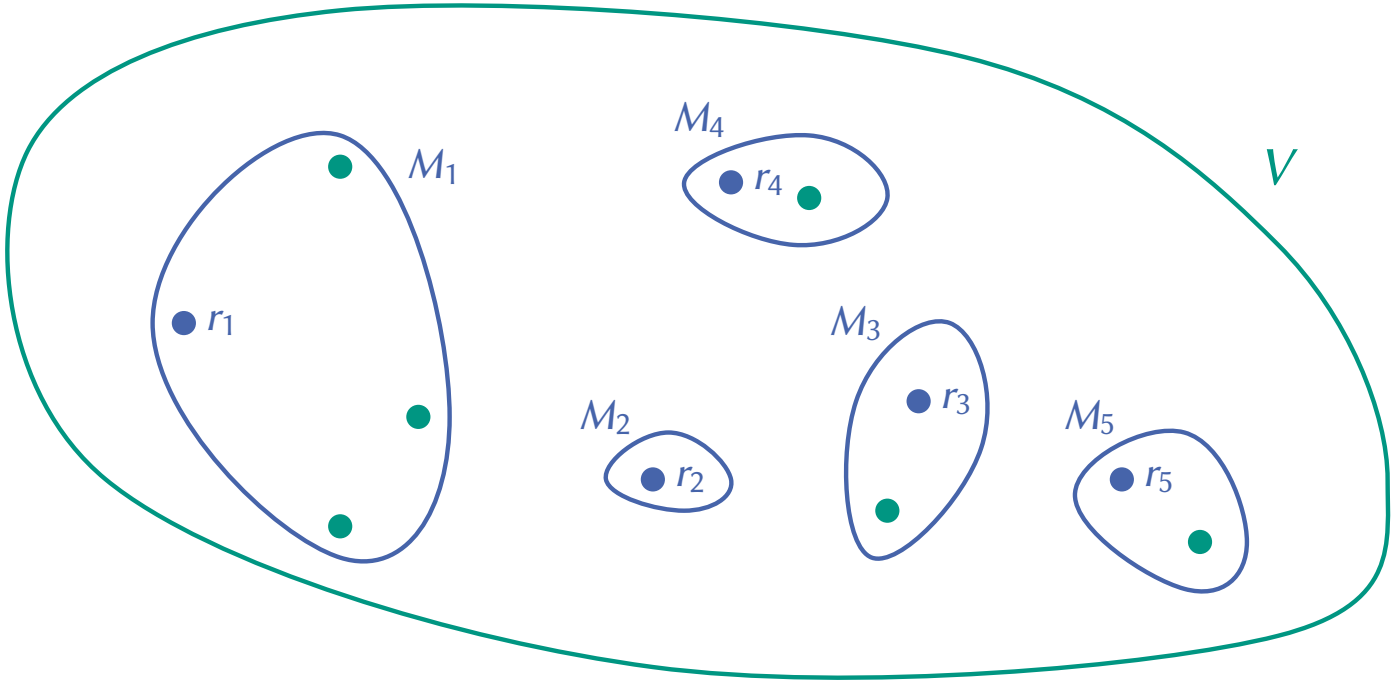


Union-Find Datenstruktur

Eine endliche Menge $V = \{1 \dots n\}$ sei in disjunkte Klassen M_i partitioniert:

$$V = M_1 \cup \cdots \cup M_k \qquad \forall i \neq j : M_i \cap M_j = \emptyset$$

Jede Klasse M_i hat eindeutigen Repräsentanten $r_i \in M_i$.



Union-Find Datenstruktur

Eine endliche Menge $V = \{1 \dots n\}$ sei in disjunkte Klassen M_i partitioniert:

$$V = M_1 \cup \dots \cup M_k \qquad \forall i \neq j : M_i \cap M_j = \emptyset$$

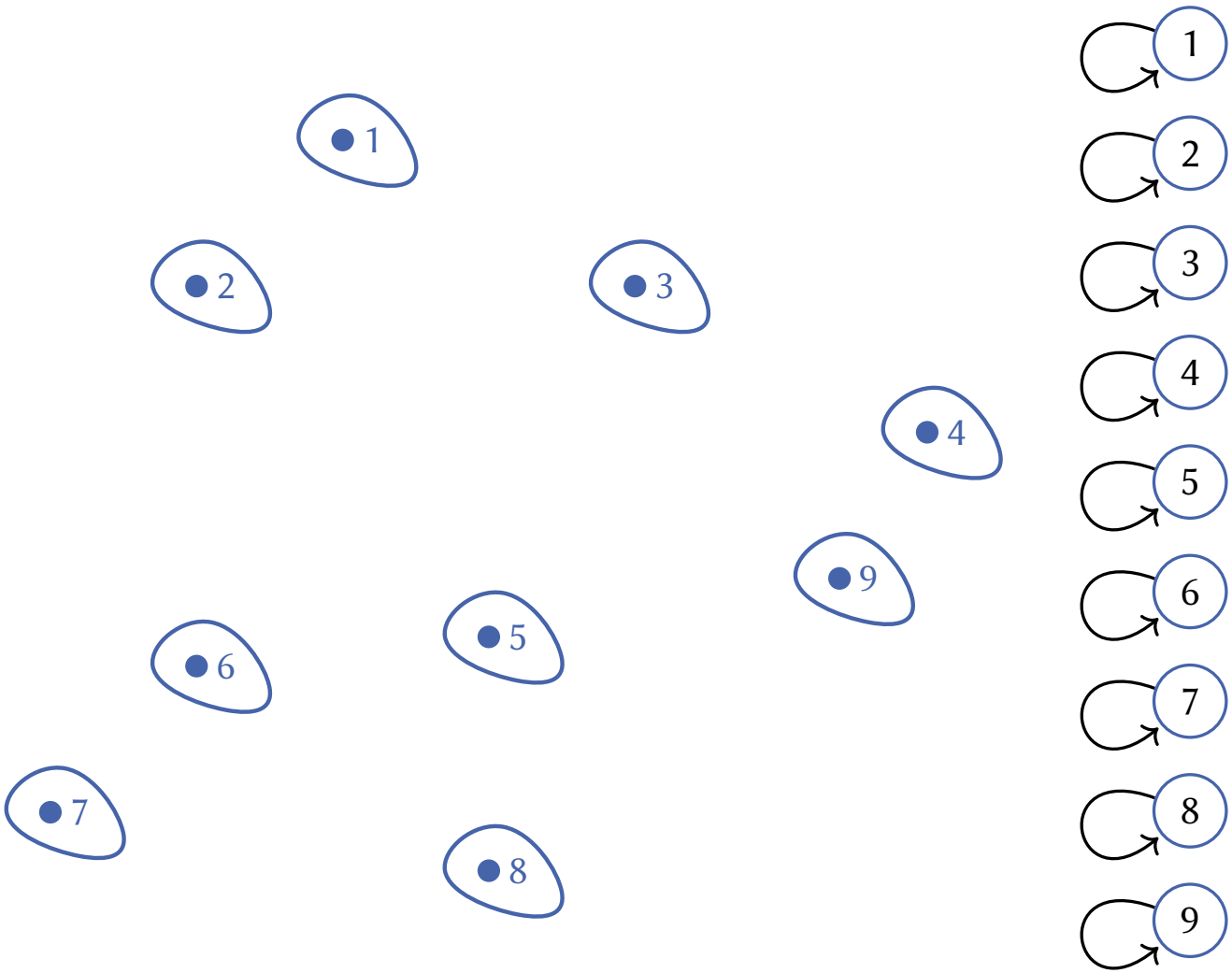
Jede Klasse M_i hat eindeutigen Repräsentanten $r_i \in M_i$.

Die Union-Find-Datenstruktur unterstützt

- `union`$(i, j : V)$
  - vereinigt die beiden Klassen, zu denen i und j gehören
- `find`$(i : V) : V$
  - bestimmt den Repräsentanten, zu dessen Klasse i gehört
- **Initialisierung**: jedes $v \in V$ bildet eigene Klasse mit v als Repräsentant

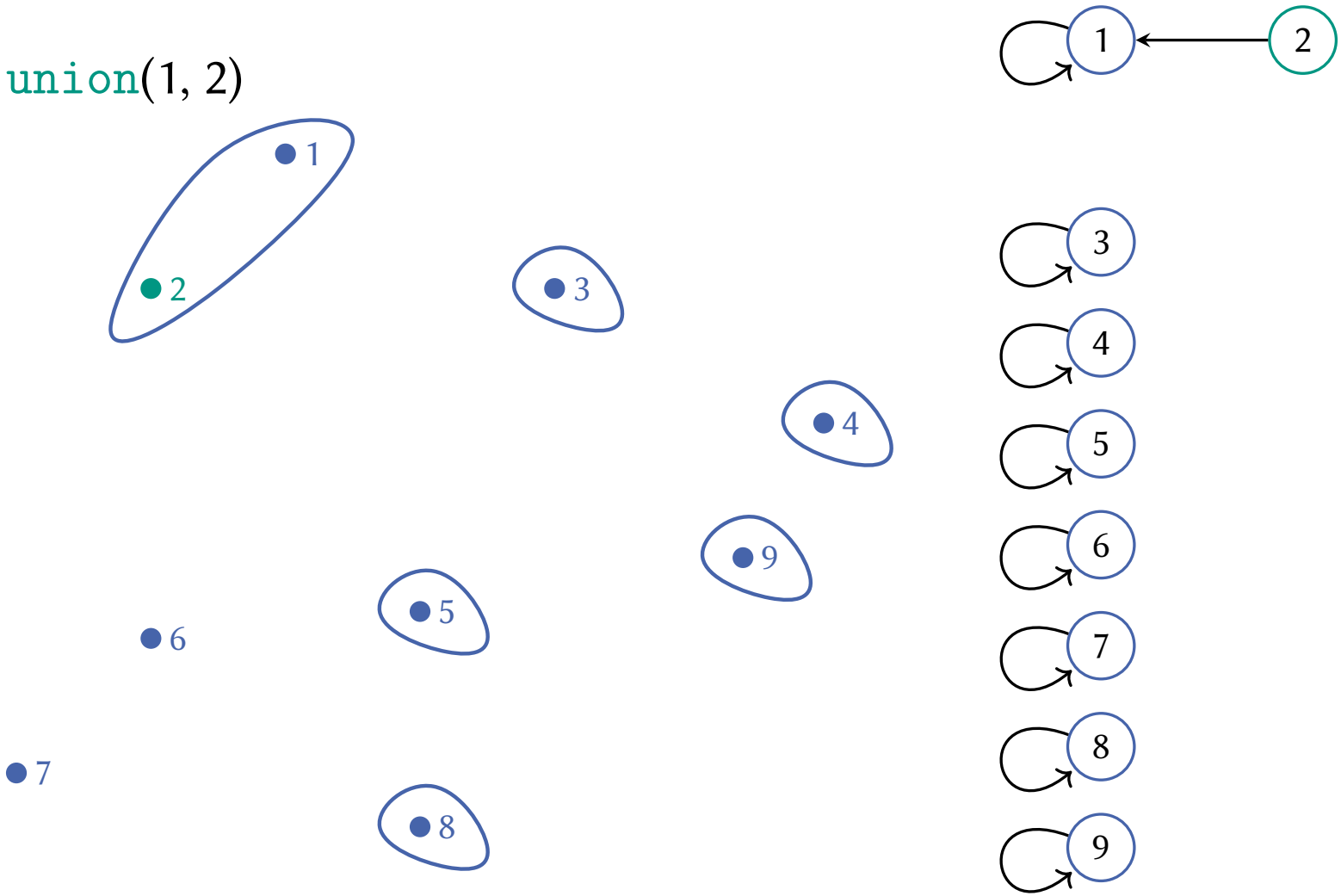
Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln



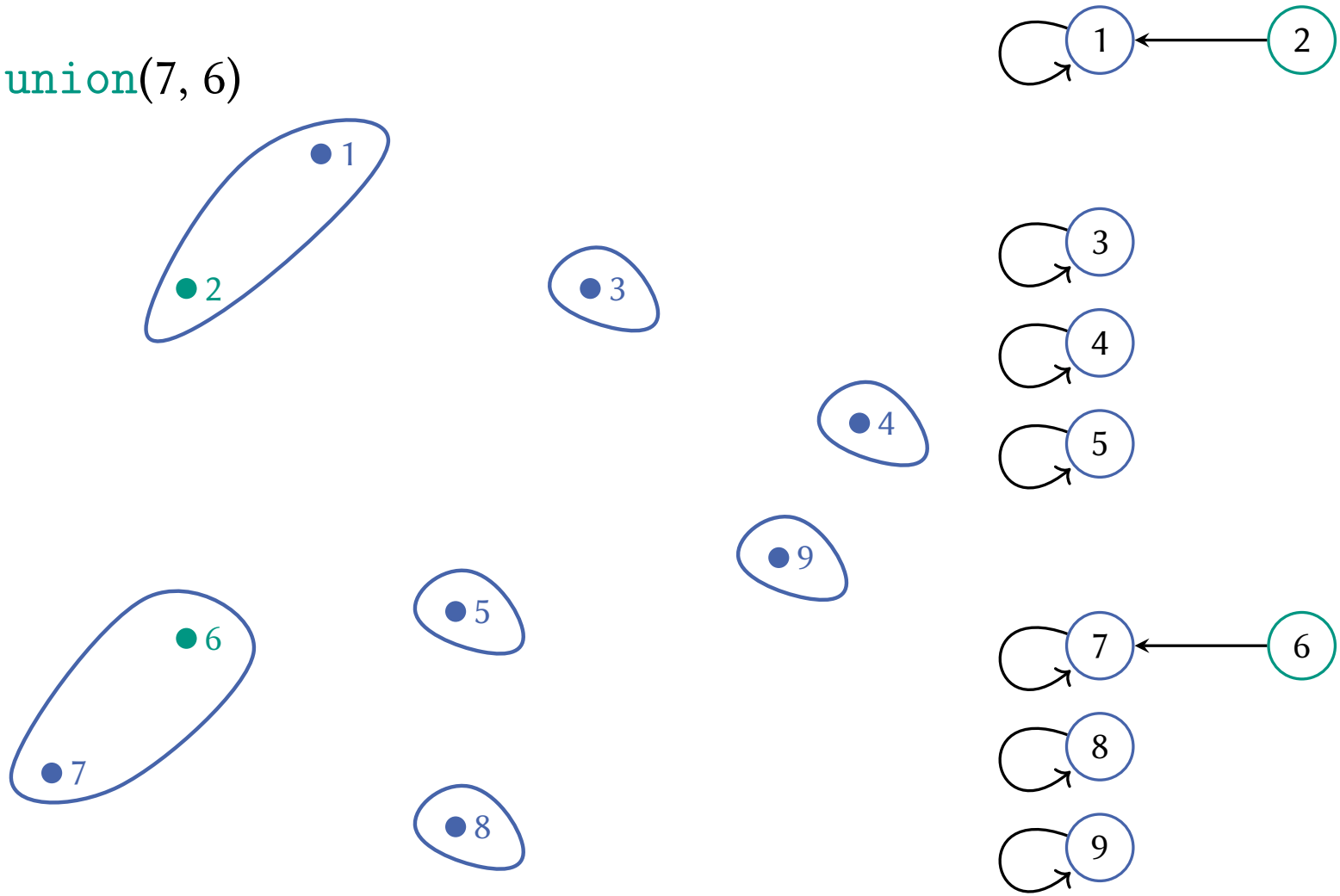
Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln



Union-Find - Erste Version

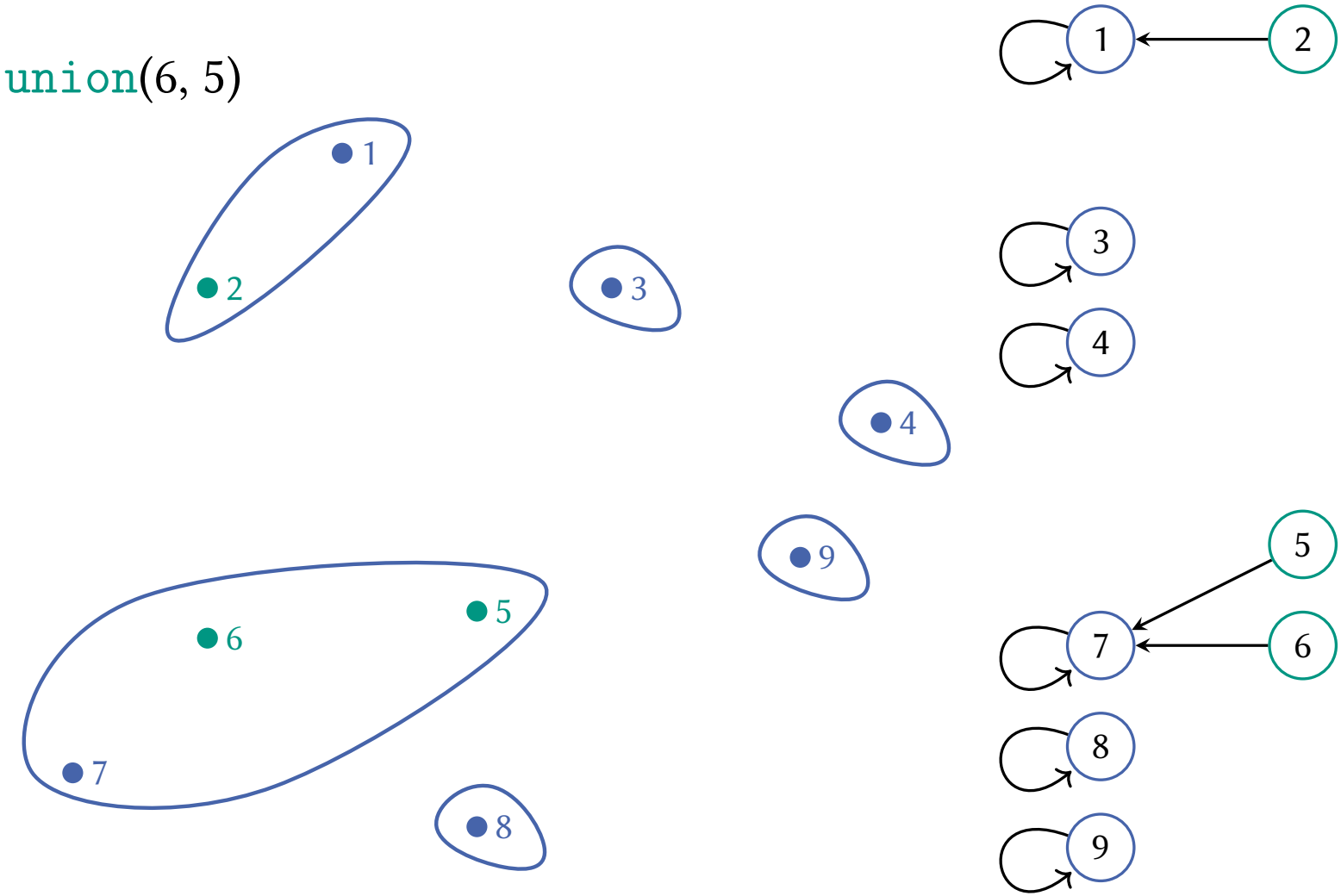
Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln



Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

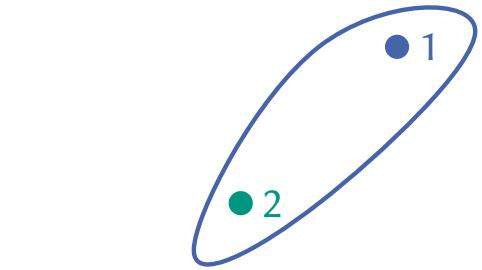
`union`(6, 5)

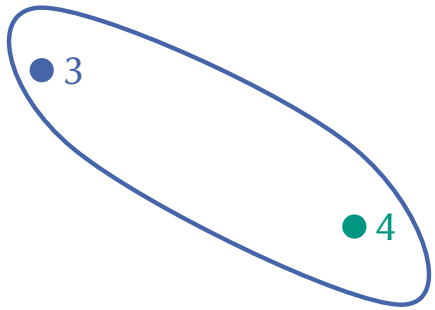


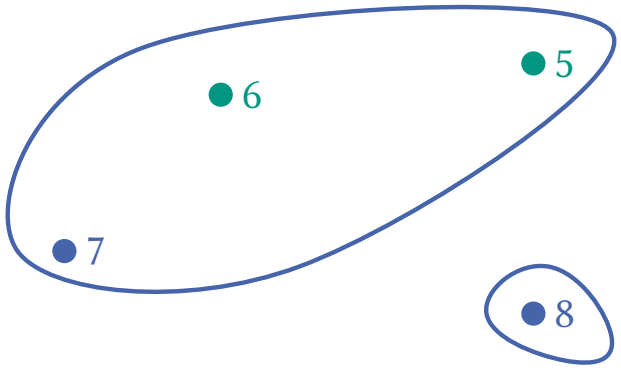
Union-Find - Erste Version

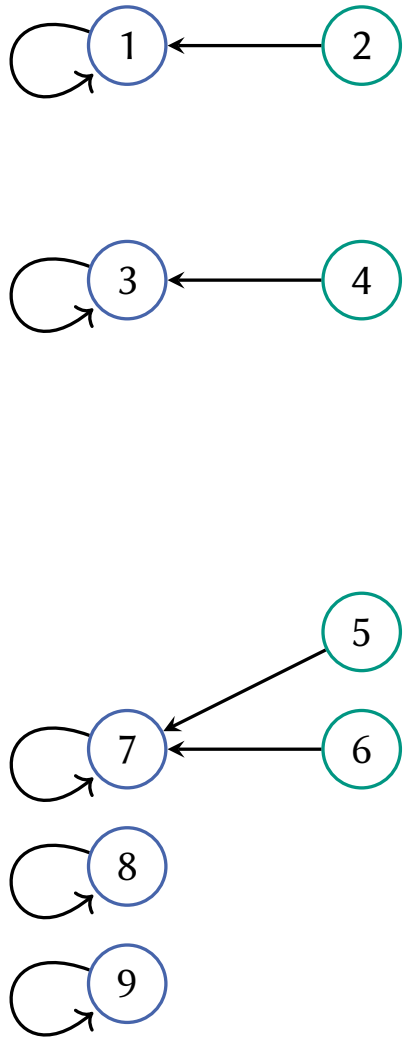
Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

`union`(3, 4)





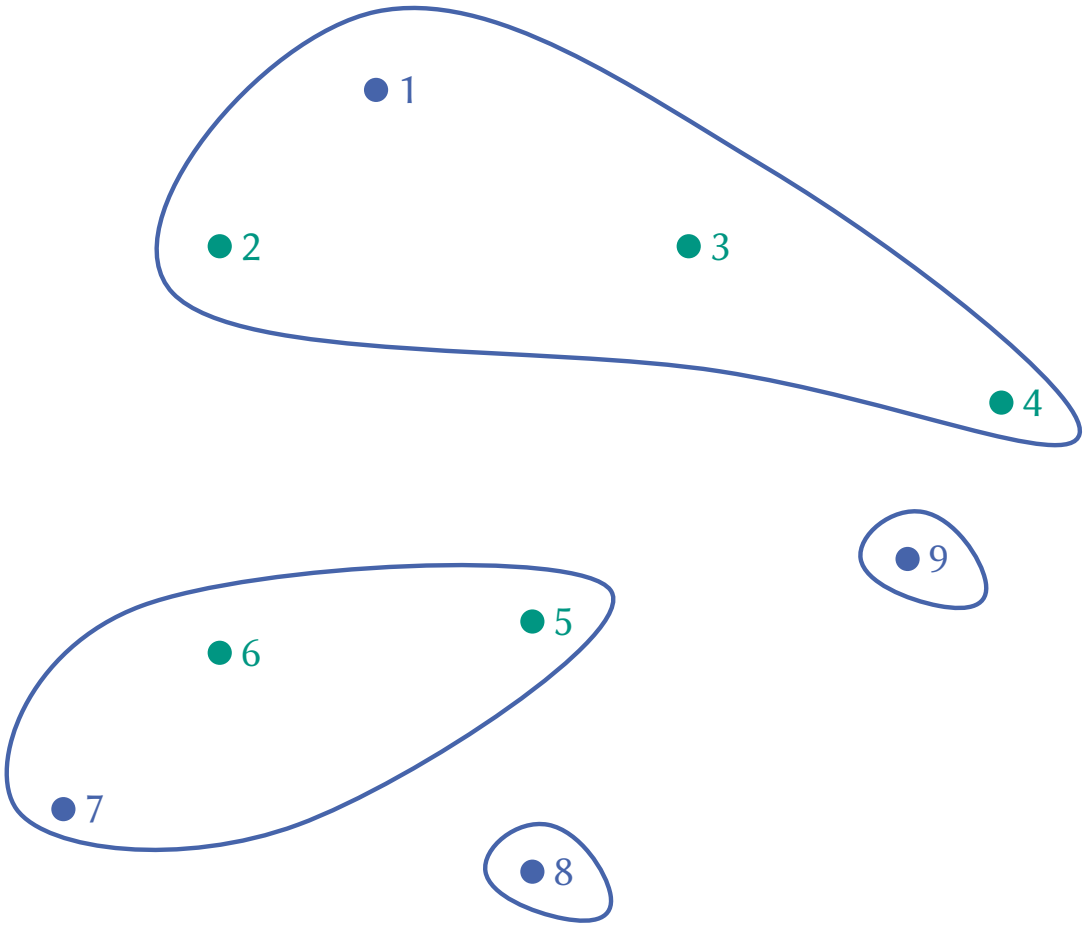


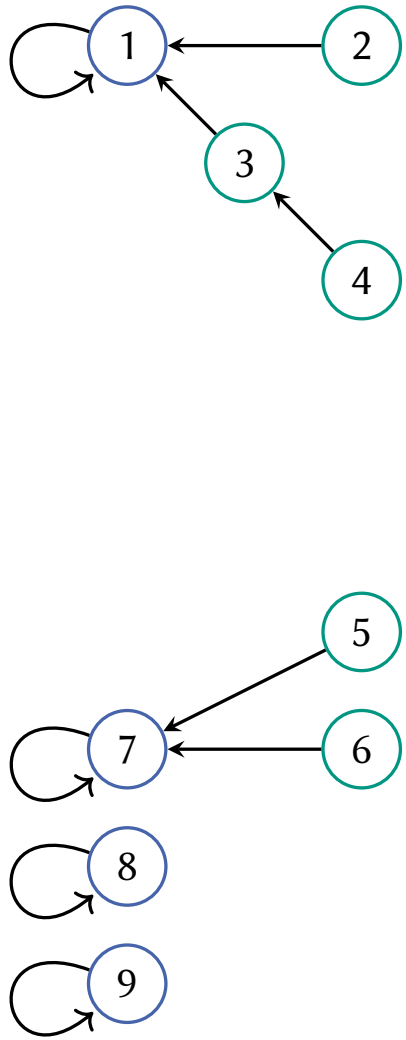


Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

`union`(1, 4)





Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

FIND($i : V$) : V

  if $i.\text{parent} == i$ **then return** i

  else return FIND($i.\text{parent}$)

Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

FIND($i : V$) : V

  if $i.\text{parent} == i$ **then return** i

  else return FIND($i.\text{parent}$)

UNION($i, j : V$)

  $r_i :=$FIND(i)

  $r_j :=$FIND(j)

  if $r_i \neq r_j$ **then** $r_i.\text{parent} := r_j$

Laufzeit:

Union-Find - Erste Version

Grundidee: Klassen sind Bäume, Repräsentanten sind Wurzeln

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

FIND($i : V$) : V

  if $i.\text{parent} == i$ **then return** i

  else return FIND($i.\text{parent}$)

UNION($i, j : V$)

  $r_i :=$FIND(i)

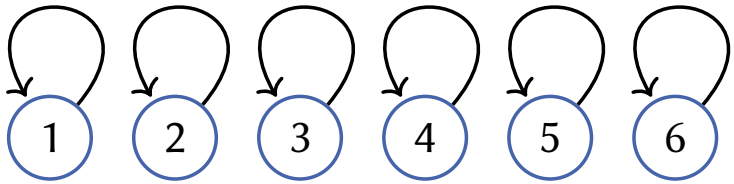
  $r_j :=$FIND(j)

  if $r_i \neq r_j$ **then** $r_i.\text{parent} := r_j$

Laufzeit: FIND im schlechtesten Fall in $\Theta(n)$, weil der Baum unbalanciert wird (siehe Binärbäume)

Union-Find - Union by Rank

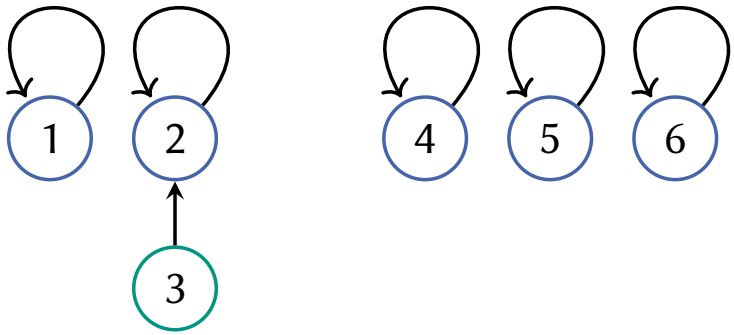
Was passiert im schlechtesten Fall?



1: $M = \{1, 2, 3, 4, 5\}$
2: $U :=$ UnionFind$\{M\}$
3: ...
4: U.union$(2, 3)$
5: U.union$(4, 2)$
6: U.union$(5, 3)$
7: U.union$(5, 6)$
8: U.union$(2, 1)$
9: ...

Union-Find - Union by Rank

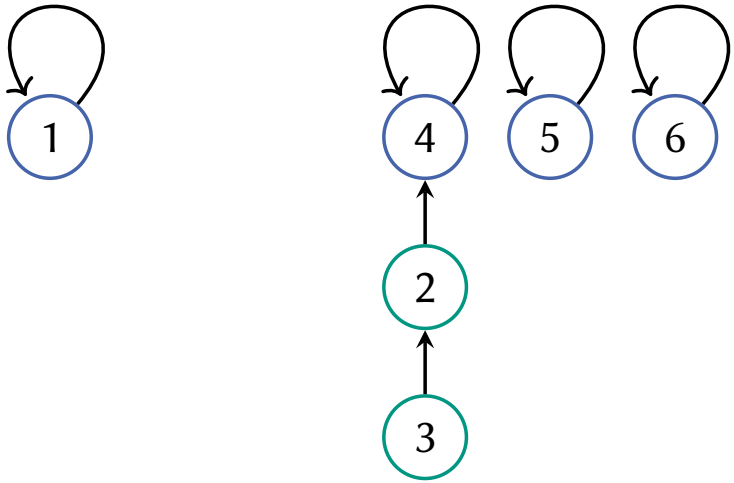
Was passiert im schlechtesten Fall?



1: $M = \{1, 2, 3, 4, 5\}$
2: $U :=$ UnionFind$\{M\}$
3: ...
4: U.union$(2, 3)$
5: U.union$(4, 2)$
6: U.union$(5, 3)$
7: U.union$(5, 6)$
8: U.union$(2, 1)$
9: ...

Union-Find - Union by Rank

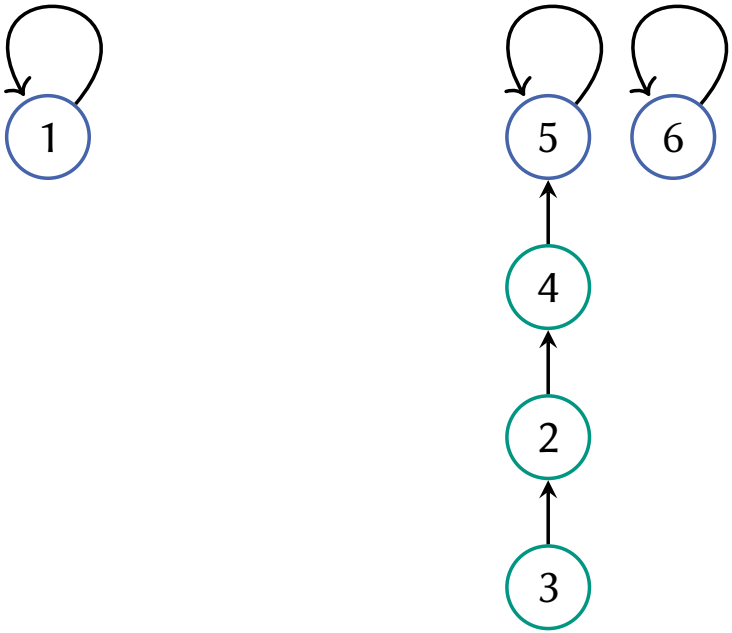
Was passiert im schlechtesten Fall?



1: $M = \{1, 2, 3, 4, 5\}$
2: $U := \text{UnionFind}\{M\}$
3: ...
4: $U.\texttt{union}(2, 3)$
5: $U.\texttt{union}(4, 2)$
6: $U.\texttt{union}(5, 3)$
7: $U.\texttt{union}(5, 6)$
8: $U.\texttt{union}(2, 1)$
9: ...

Union-Find - Union by Rank

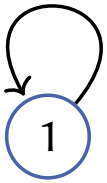
Was passiert im schlechtesten Fall?

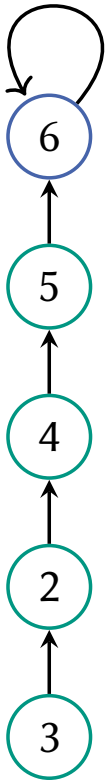


1: $M = \{1, 2, 3, 4, 5\}$
2: $U := \text{UnionFind}\{M\}$
3: ...
4: $U.\texttt{union}(2, 3)$
5: $U.\texttt{union}(4, 2)$
6: $U.\texttt{union}(5, 3)$
7: $U.\texttt{union}(5, 6)$
8: $U.\texttt{union}(2, 1)$
9: ...

Union-Find - Union by Rank

Was passiert im schlechtesten Fall?

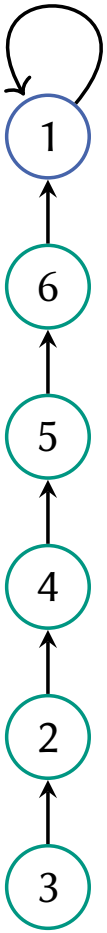




1: $M = \{1, 2, 3, 4, 5\}$
2: $U :=$ UnionFind$\{M\}$
3: ...
4: U.union$(2, 3)$
5: U.union$(4, 2)$
6: U.union$(5, 3)$
7: U.union$(5, 6)$
8: U.union$(2, 1)$
9: ...

Union-Find - Union by Rank

Was passiert im schlechtesten Fall?



1: $M = \{1, 2, 3, 4, 5\}$
2: $U :=$ UnionFind$\{M\}$
3: ...
4: U.union$(2, 3)$
5: U.union$(4, 2)$
6: U.union$(5, 3)$
7: U.union$(5, 6)$
8: U.union$(2, 1)$
9: ...

Union-Find - Union by Rank

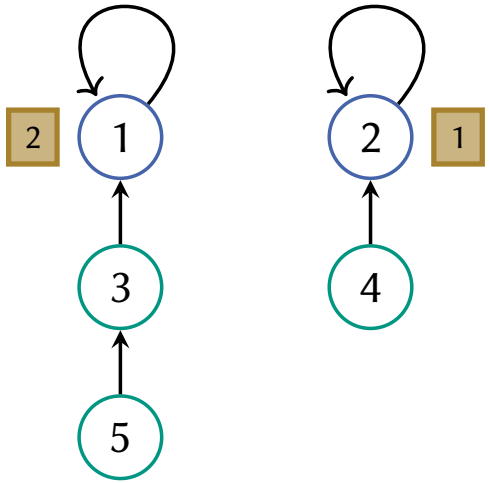
- **Ziel**: Höhe des Ergebnisses von `union` so weit wie möglich beschränken
- **Idee**: Bei der Operation `union`(i, j) wird der flachere unter den höheren Baum gehängt
- jeder Knoten erhält einen Rang
  - initalisiert zu 0
  - wird durch `union` um 1 erhöht

  $\Rightarrow$ entspricht Höhe des Baumes

Union-Find - Union by Rank

- **Ziel**: Höhe des Ergebnisses von `union` so weit wie möglich beschränken
- **Idee**: Bei der Operation `union`(i, j) wird der flachere unter den höheren Baum gehängt
- jeder Knoten erhält einen Rang
  - initalisiert zu 0
  - wird durch `union` um 1 erhöht

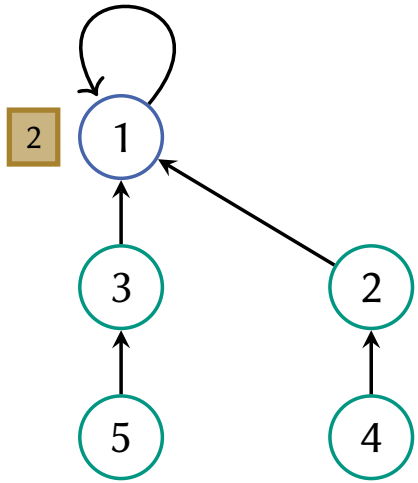
  $\Rightarrow$ entspricht Höhe des Baumes



Union-Find - Union by Rank

- **Ziel**: Höhe des Ergebnisses von `union` so weit wie möglich beschränken
- **Idee**: Bei der Operation `union`(i, j) wird der flachere unter den höheren Baum gehängt
- jeder Knoten erhält einen Rang
  - initalisiert zu 0
  - wird durch `union` um 1 erhöht

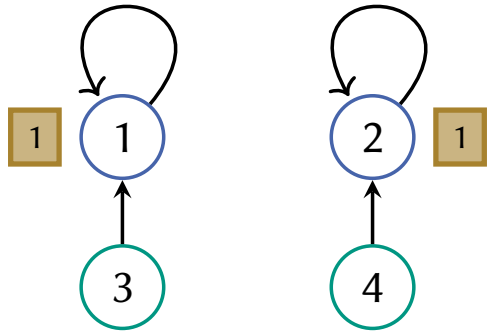
  $\Rightarrow$ entspricht Höhe des Baumes



Union-Find - Union by Rank

- **Ziel**: Höhe des Ergebnisses von `union` so weit wie möglich beschränken
- **Idee**: Bei der Operation `union`(i, j) wird der flachere unter den höheren Baum gehängt
- jeder Knoten erhält einen Rang
  - initalisiert zu 0
  - wird durch `union` um 1 erhöht

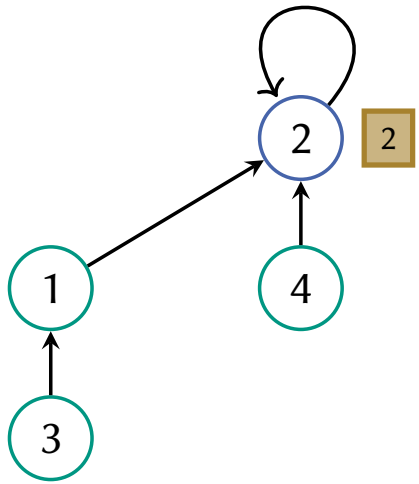
  $\Rightarrow$ entspricht Höhe des Baumes



Union-Find - Union by Rank

- **Ziel**: Höhe des Ergebnisses von `union` so weit wie möglich beschränken
- **Idee**: Bei der Operation `union`(i, j) wird der flachere unter den höheren Baum gehängt
- jeder Knoten erhält einen Rang
  - initalisiert zu 0
  - wird durch `union` um 1 erhöht

  $\Rightarrow$ entspricht Höhe des Baumes



Union-Find - Union by Rank

- **Ziel**: Höhe des Ergebnisses von `union` so weit wie möglich beschränken
- **Idee**: Bei der Operation `union`(i, j) wird der flachere unter den höheren Baum gehängt
- jeder Knoten erhält einen Rang
  - initalisiert zu 0
  - wird durch `union` um 1 erhöht

  $\Rightarrow$ entspricht Höhe des Baumes

Verbesserung durch Union By Rank

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

$\Rightarrow$ Bäume können nicht zu Listen entarten

Union-Find - Pfadkompresssion

- **Ziel**: Bäume mit jedem Aufruf von `find` stauchen
- **Idee**: parent jedes iterierten Elements auf Repräsentanten setzen

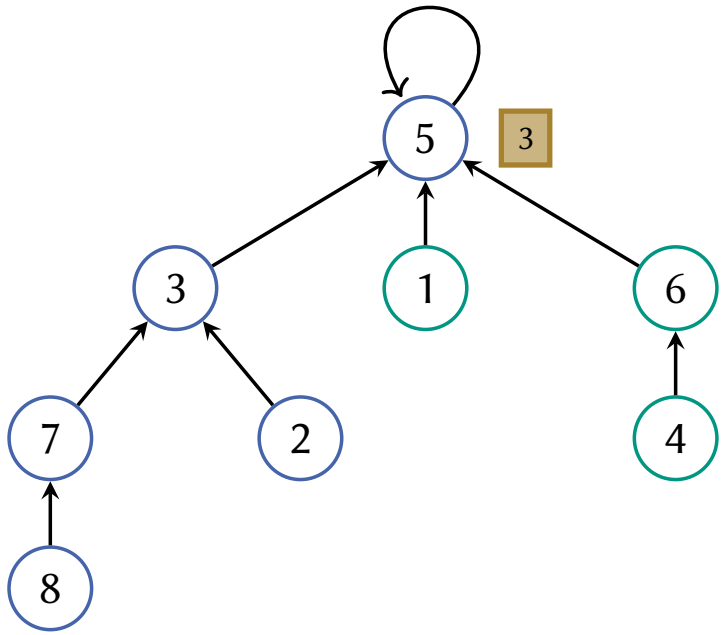
$\Rightarrow$ nächstes `find` für jedes dieser Elemente in $\Theta(1)$

Union-Find - Pfadkompresssion

- **Ziel**: Bäume mit jedem Aufruf von `find` stauchen
- **Idee**: parent jedes iterierten Elements auf Repräsentanten setzen

$\Rightarrow$ nächstes `find` für jedes dieser Elemente in $\Theta(1)$

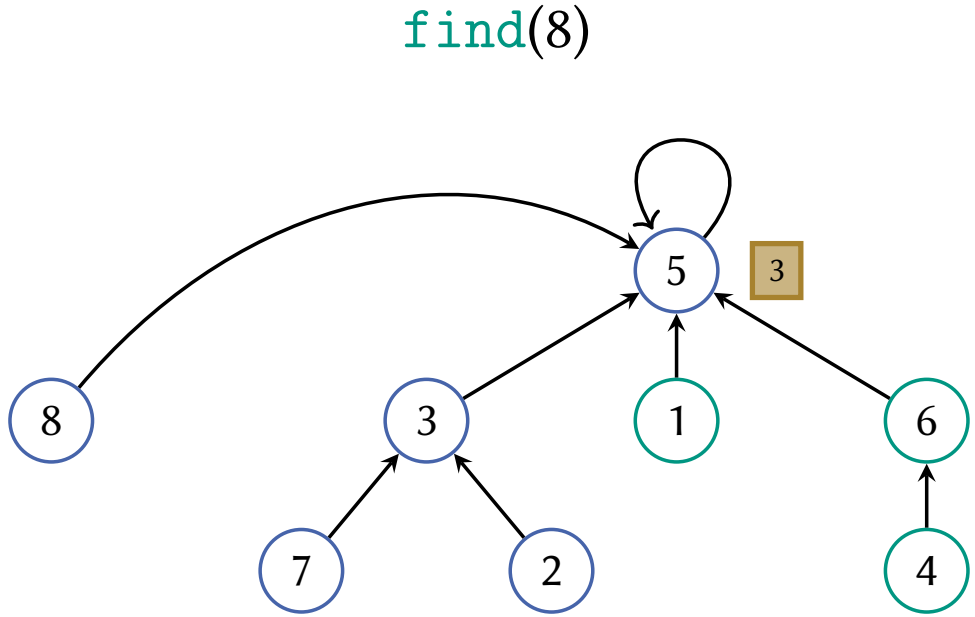
`find`(8)



Union-Find - Pfadkompresssion

- **Ziel**: Bäume mit jedem Aufruf von `find` stauchen
- **Idee**: parent jedes iterierten Elements auf Repräsentanten setzen

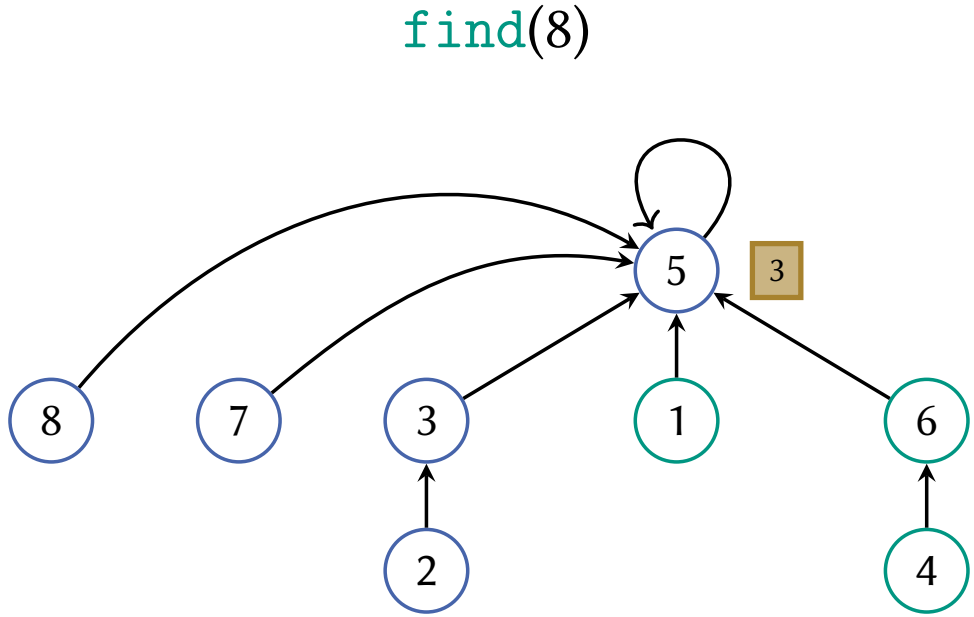
$\Rightarrow$ nächstes `find` für jedes dieser Elemente in $\Theta(1)$



Union-Find - Pfadkompresssion

- **Ziel**: Bäume mit jedem Aufruf von `find` stauchen
- **Idee**: parent jedes iterierten Elements auf Repräsentanten setzen

$\Rightarrow$ nächstes `find` für jedes dieser Elemente in $\Theta(1)$



Union-Find - Pfadkompresssion

- **Ziel**: Bäume mit jedem Aufruf von `find` stauchen
- **Idee**: parent jedes iterierten Elements auf Repräsentanten setzen

$\Rightarrow$ nächstes `find` für jedes dieser Elemente in $\Theta(1)$

```
    ⋮
FIND(i : V) : V
    if i.parent == i then return i
    else
        i' := FIND(i.parent)
        i.parent := i'
        return i'
    ⋮
```

Optimierte Laufzeiten - Fazit

- **Problem:** Pfadkompression verändert die Höhe eines Baumes, aber nicht den Rang
- Also unvereinbar mit Union by Rank?

Optimierte Laufzeiten - Fazit

- **Problem:** Pfadkompression verändert die Höhe eines Baumes, aber nicht den Rang
- Also unvereinbar mit Union by Rank? **Nein!**
- weiterhin: Baum mit Rang r hat mindestens 2^r Knoten
- außerdem: Höhe dieses Baumes ist höchstens r

Optimierte Laufzeiten - Fazit

- **Problem:** Pfadkompression verändert die Höhe eines Baumes, aber nicht den Rang
- Also unvereinbar mit Union by Rank? **Nein!**
- weiterhin: Baum mit Rang r hat mindestens 2^r Knoten
- außerdem: Höhe dieses Baumes ist höchstens r

Mit beiden Optimierungen bekommen wir erfreuliche Laufzeiten:

- $m \cdot \textsc{find} + n \cdot \textsc{union} \in \mathcal{O}((n+m) \cdot \log^*(n))$
- $\log^*$ ist hierbei der iterierte Logarithmus

Optimierte Laufzeiten - Fazit

- **Problem:** Pfadkompression verändert die Höhe eines Baumes, aber nicht den Rang
- Also unvereinbar mit Union by Rank? **Nein!**
- weiterhin: Baum mit Rang r hat mindestens 2^r Knoten
- außerdem: Höhe dieses Baumes ist höchstens r

Genauer, aber nicht in dieser Vorlesung gezeigt:

- $m \cdot \text{FIND} + n \cdot \text{UNION} \in \mathcal{O}((n+m) \cdot \alpha_T(m,n))$
- α_T ist hierbei die inverse Ackermann-Funktion und wächst unglaublich langsam

Alles klar soweit?

1: $V = \{0, \ldots, 15\}$
2: $U :=$ UnionFind$\{V\}$
3: $U.\texttt{union}(1, 3)$
4: $U.\texttt{union}(4, 14)$
5: $x := U.\texttt{find}(3)$
6: $U.\texttt{union}(3, 4)$
7: $y := U.\texttt{find}(14)$

Alles klar soweit?

1: $V = \{0, \ldots, 15\}$
2: $U :=$ UnionFind$\{V\}$
3: $U.\texttt{union}(1, 3)$
4: $U.\texttt{union}(4, 14)$
5: $x := U.\texttt{find}(3)$
6: $U.\texttt{union}(3, 4)$
7: $y := U.\texttt{find}(14)$

Es ist nun $x \circ y$ mit $\circ \in \{<, =, >\}$.
Was ist $\circ$?

Alles klar soweit?

1: $V = \{0, \ldots, 15\}$
2: $U :=$ UnionFind$\{V\}$
3: $U.\texttt{union}(1, 3)$
4: $U.\texttt{union}(4, 14)$
5: $x := U.\texttt{find}(3)$
6: $U.\texttt{union}(3, 4)$
7: $y := U.\texttt{find}(14)$
8: $U.\texttt{union}(3, 7)$
9: $z := U.\texttt{find}(7)$

Alles klar soweit?

1: $V = \{0, \ldots, 15\}$
2: $U :=$ UnionFind$\{V\}$
3: $U.\texttt{union}(1, 3)$
4: $U.\texttt{union}(4, 14)$
5: $x := U.\texttt{find}(3)$
6: $U.\texttt{union}(3, 4)$
7: $y := U.\texttt{find}(14)$
8: $U.\texttt{union}(3, 7)$
9: $z := U.\texttt{find}(7)$

Es ist nun $y \circ z$ mit $\circ \in \{<, =, >\}$.
Was ist $\circ$?

Roadmap

Was war nochmal Hashing?

Was war nochmal Hashing?

Und wozu war das nochmal gut?

Dr. Metas Problem

Der ebenso geniale wie misstrauische Superbösewicht Dr. Meta hat einen schweren Schlag erlitten: Nach dem letzten Coup, den er gegen seinen Erzfeind Theorie-Man startete, ging seine Flotte der superschnellen Meta-Mobile unter ungeklärten Umständen in Flammen auf. Da dies nicht das erste Mal war, beschließen sich einige seiner Investoren, ihm die Geldmittel zu streichen.
Dr. Meta muss sparen!

Er beschließt kurzerhand, Personalkosten zu sparen und den Teil seiner Mitarbeiter den Krokodilen vorzuwerfen, die seiner Einschätzung nach im letzten Jahr zu viel Urlaub nehmen.

Dr. Metas Problem

- GEG: Anzahl Mitarbeiter, Aufzählung von Urlaubstagen
  - Eintrag: (Mitarbeiter-ID, Tag)
  - IDs sind eindeutig
  - Tag: natürliche Zahl aus $\{0, \ldots, 364\}$
- GES: Mitarbeiter, die mehr als 2 Wochen Urlaub genommen haben
  - Ausgabe: IDs der Mitarbeiter
  - Anforderung: erwartet linearer Laufzeit

Dr. Metas Problem

- Geg: Anzahl Mitarbeiter, Aufzählung von Urlaubstagen
  - Eintrag: (Mitarbeiter-ID, Tag)
  - IDs sind eindeutig
  - Tag: natürliche Zahl aus $\{0, \ldots, 364\}$
- Ges: Mitarbeiter, die mehr als 2 Wochen Urlaub genommen haben
  - Ausgabe: IDs der Mitarbeiter
  - Anforderung: erwartet linearer Laufzeit

Aber wie?

Dr. Metas Problem

- GEG: Anzahl Mitarbeiter, Aufzählung von Urlaubstagen
  - Eintrag: (Mitarbeiter-ID, Tag)
  - IDs sind eindeutig
  - Tag: natürliche Zahl aus $\{0, \ldots, 364\}$
- GES: Mitarbeiter, die mehr als 2 Wochen Urlaub genommen haben
  - Ausgabe: IDs der Mitarbeiter
  - Anforderung: erwartet linearer Laufzeit

Aber wie?

Lösungsansätze

Was können wir nutzen?

Lösungsansätze

Was können wir nutzen?

- Daten pro Tag speichern?
  - nicht zielführend, wir wollen Daten pro Mitarbeiter
- Mitarbeiter-IDs nutzen?
  - **Problem**: IDs sind unabhängig von der Anzahl Mitarbeiter

$\Rightarrow$ maximale ID finden wir in Linearzeit, bringt uns nur leider nichts

Lösungsansätze

Was können wir nutzen?

- Daten pro Tag speichern?
  - nicht zielführend, wir wollen Daten pro Mitarbeiter
- Mitarbeiter-IDs nutzen?
  - **Problem**: IDs sind unabhängig von der Anzahl Mitarbeiter

  $\Rightarrow$ maximale ID finden wir in Linearzeit, bringt uns nur leider nichts
- Optimal wäre: ein Eintrag pro Mitarbeiter, Zugriff in $\Theta(1)$
  - i.d.R. nicht möglich, da wir zu wenig über IDs wissen
  - Aber wir können dem nahe kommen…

Motivation I

- «hashing» zu deutsch: «zerhacken»
- zerhackt wird die Grundmenge, über der unser Input gebildet wird
- jede Untermenge ist zu Hashwert assoziiert

Motivation I

- «hashing» zu deutsch: «zerhacken»
- zerhackt wird die Grundmenge, über der unser Input gebildet wird
- jede Untermenge ist zu Hashwert assoziiert

Wozu braucht man das?

Hashtabellen sind eine Möglichkeit, assoziative Arrays zu implementieren.

Intuition: Ein Array mit potentiell unendlicher / sehr großer Indexmenge, bei dem nur sehr wenige Indizes tatsächlich benutzt werden

(Zum Beispiel Mitarbeiter-IDs)

Etwas formaler …

Motivation II

Wir wollen eine Datenstruktur aufbauen, die ein assoziatives Array implementiert. Sie soll die Operationen

- GET(`key`)
- SET(`key`, `value`)
- DELETE(`key`)

in erwartet konstanter Zeit ausführen können.

Dabei machen wir uns Schlüssel zu nutze, um Elemente in der Datenetruktur zu finden.

Vorhang auf für …

Hashtabellen

Eine Hashtabelle ist ein Array von m Buckets zusammen mit einer Hashfunktion $h : U \to \{0, \ldots, m-1\}$.

Hashtabellen

Eine Hashtabelle ist ein Array von m Buckets zusammen mit einer Hashfunktion $h : U \to \{0, \ldots, m-1\}$.

Wollen wir ein Element e in eine Hashtabelle eintragen, müssen wir Folgendes leisten:

Hashtabellen

Eine Hashtabelle ist ein Array von m Buckets zusammen mit einer Hashfunktion $h : U \to \{0, \ldots, m-1\}$.

Wollen wir ein Element e in eine Hashtabelle eintragen, müssen wir Folgendes leisten:

1. den (eindeutigen) Schlüssel $\texttt{key}(e)$ bestimmen
2. den Hashwert $h(\texttt{key}(e))$ berechnen
3. e am Index $h(\texttt{key}(e))$ in A einfügen

Hashing

Def.: Hash-Funktion

Sei M eine Menge von beliebigen Elementen. Jedes Element $e \in M$ hat dabei einen eindeutigen Schlüssel $\texttt{key}(e)$, d.h. es gilt

$$\forall e_1, e_2 \in M : \texttt{key}(e_0) = \texttt{key}(e_1) \Rightarrow e_0 = e_1$$

Die Menge aller möglichen Schlüssel nennen wir das Universum U.
Eine Hash-Funktion **h** bildet Schlüssel auf natürliche Zahlen aus $\{0, \ldots, m-1\}$ für ein $m \in \mathbb{N}$ ab.

Elemente vs. Schlüssel vs. Hashwert

$$\text{Element} \xrightarrow{\texttt{key}} \text{Schlüssel} \xrightarrow{h} \text{Hashwert}$$

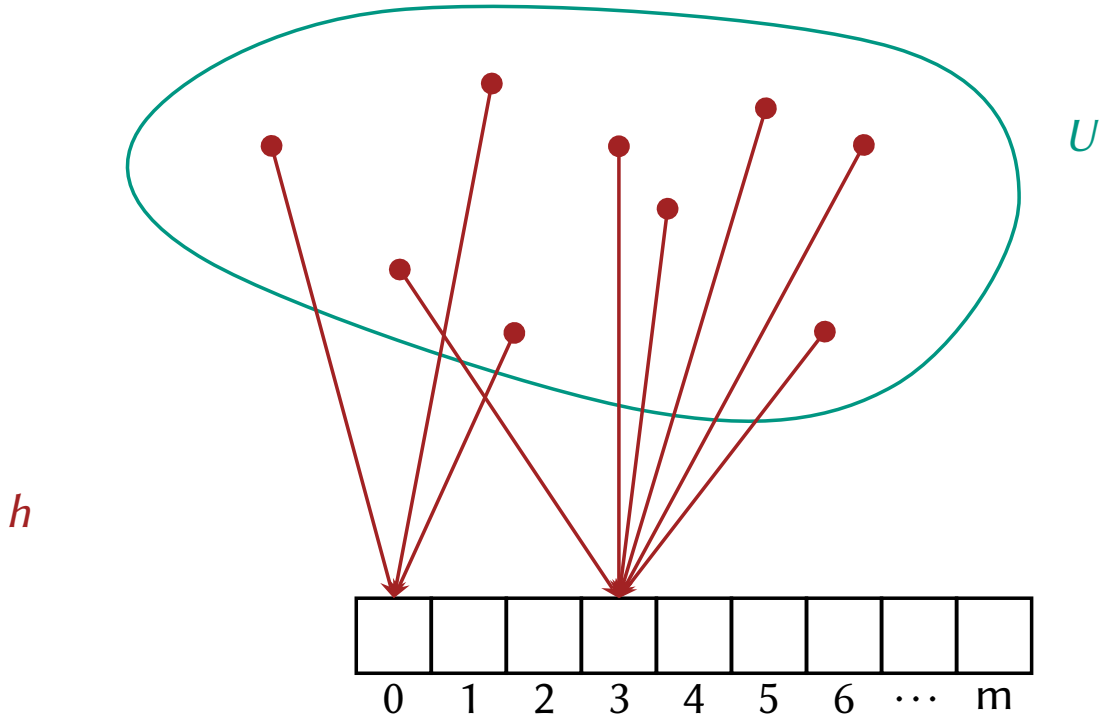
Aber: Elemente können ihr eigener Schlüssel sein (z.B. numerische Werte)

Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

Hashing

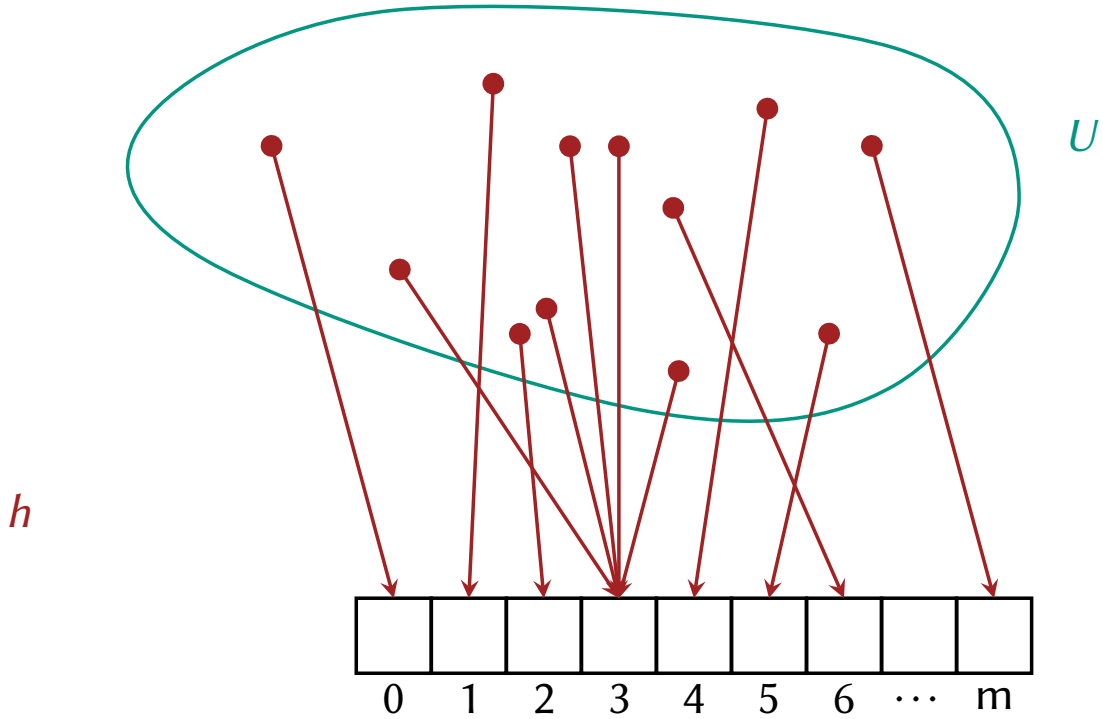
Welche Kriterien erfüllt eine «gute» Hashfunktion?



Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

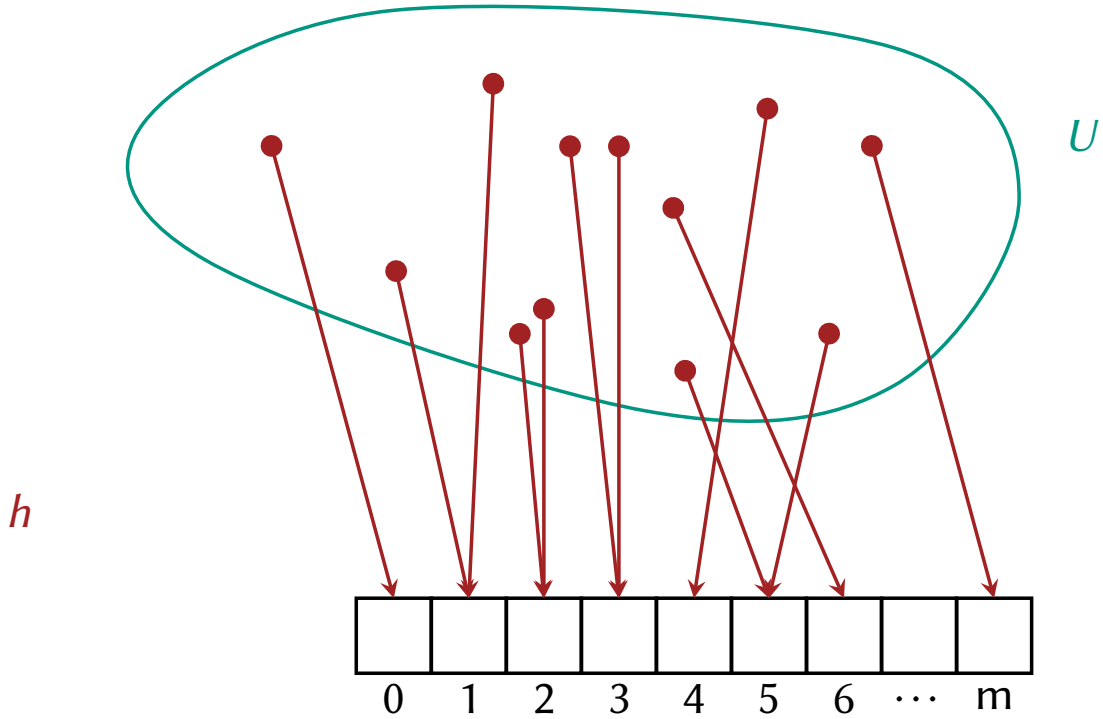
- **Surjektivität**: alle Hashwerte können genutzt werden



Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

- **Surjektivität**: alle Hashwerte können genutzt werden
- **Gleichverteilung**: alle Hashwerte werden mit der gleichen Wahrscheinlichkeit getroffen
- Kollisionen sind möglichst selten



Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

- **Surjektivität**: alle Hashwerte können genutzt werden
- **Gleichverteilung**: alle Hashwerte werden mit der gleichen Wahrscheinlichkeit getroffen
- Kollisionen sind möglichst selten

Was, wenn es beim Einfügen eines Elements zur Kollision kommt?

Kollisionsauflösung!

- eine Liste pro Bucket
- Elemente werden in die Liste am entsprechenden Index in der Hashtabelle eingefügt

Sind Kollisionen Anzeichen einer schlechten Hashfunktion?

Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

- **Surjektivität**: alle Hashwerte können genutzt werden
- **Gleichverteilung**: alle Hashwerte werden mit der gleichen Wahrscheinlichkeit getroffen
- Kollisionen sind möglichst selten

Was, wenn es beim Einfügen eines Elements zur Kollision kommt?

Kollisionsauflösung!

- eine Liste pro Bucket
- Elemente werden in die Liste am entsprechenden Index in der Hashtabelle eingefügt

Sind Kollisionen Anzeichen einer schlechten Hashfunktion?

Nein! Allein schon deswegen, weil wir die Anzahl der Elemente, die wir in eine Hashtabelle einfügen, nicht beschränken, sind Kollisionen unvermeidbar.

Auswahl einer geeigneten Hashfunktion

- **Problem:** i.d.R. kaum wissen über die Schlüsselmenge

$\Rightarrow$ Entscheiden wir uns für eine feste Hashfunktion, ist es allein abhängig von der Schlüsselmenge, wie (un-)gleichmäßig die Elemente verteilt werden.

- **Frage:** Wie können wir trotzdem eine möglichst günstige Hashfunktion wählen?

Idee der zufällig gewählten Hashfunktion

Lösung: Wir wählen unsere Hashfunktion aus einer Familie (geeigneter) Hashfunktionen **zufällig**.

Natürlich kann diese Funktion immer noch ungünstig für unsere späteren Eingaben sein. Der Vorteil ist jedoch, dass die Wahrscheinlichkeit dafür sinkt.

Universelles Hashing

Definition: Universelle Hashfamilie

Sei $\mathcal{H}$ eine Menge von Hashfunktionen, welche von U auf $\{0, .., \ m-1\}$ abbilden.
$\mathcal{H}$ heißt **universell**, wenn für ein zufällig gewähltes $h \in \mathcal{H}$ gilt:

$$\forall k, \ l \in U, \ k \neq l : \ \Pr[h(k) = h(l)] = \frac{1}{m}$$

Anders formuliert: Wählen wir eine Hashfunktion $h \in \mathcal{H}$ zufällig, ist die Kollisionswahrscheinlichkeit für zwei beliebige Schlüssel $\frac{1}{m}$.

Universelles Hashing

Definition: Universelle Hashfamilie

Sei $\mathcal{H}$ eine Menge von Hashfunktionen, welche von U auf $\{0, .., \ m-1\}$ abbilden.
$\mathcal{H}$ heißt **universell**, wenn für ein zufällig gewähltes $h \in \mathcal{H}$ gilt:

$$\forall k, \ l \in U, \ k \neq l : \ \Pr[h(k) = h(l)] = \frac{1}{m}$$

Anders formuliert: Wählen wir eine Hashfunktion $h \in \mathcal{H}$ zufällig, ist die Kollisionswahrscheinlichkeit für zwei beliebige Schlüssel $\frac{1}{m}$.

Beispiel einer universellen Hashfamilie

Wenn $U \subseteq \mathbb{N}$:
$h_{a,b}(k) = ((ak + b) \ mod \ p)$ mit $a, b \in \mathbb{N}_0, \ a \neq 0$ und p ist eine Primzahl.

Alles klar soweit?

Dr. Metas Lösung

Wie kann Dr. Metas Problem also gelöst werden?

Erinnerung:

- GEG:
  - Aufzählung der Form (Mitarbeiter-ID, Tag)
  - Anzahl Mitarbeiter
- GES: Mitarbeiter, die mindestens 14 Tage Urlaub genommen haben
- gewünscht: erwartet lineare Laufzeit

Idee:

- Ein Zähler pro Mitarbeiter
- Hashtabelle mit Kapazität in Θ(Anzahl Mitarbeiter)
- IDs als Schlüssel

Dr. Metas Lösung

```
1:  publicShaming(vacations : [(ℕ, {0, ..., 364}); n], count : ℕ) : List⟨ℕ⟩
2:      map := HashMap{ℕ → ℕ}
3:      doomed : List⟨ℕ⟩
4:      for (id, date) ∈ vacations do
5:          days : ℕ = map.get(id)
6:          if days = ⊥ then
7:              map.set(id, 1)
8:          else
9:              if days = 14 then
10:                 doomed.append(id)
11:             map.set(id, days + 1)
12:     return doomed
```

Was wäre, wenn ...

Angenommen, Dr. Meta will seine Mitarbeiter nicht beseitigen, sondern ihnen nur das Leben schwer machen. Er sucht nicht mehr Mitarbeiter, sondern die Tage, an denen zu viel Urlaub genommen wurde, um an genau diesen Tagen Urlaub zu verbieten.

Wie könnten wir die Tage bestimmen, an denen mehr als die Hälfte der Mitarbeiter Urlaub genommen hat?

Was wäre, wenn ...

Angenommen, Dr. Meta will seine Mitarbeiter nicht beseitigen, sondern ihnen nur das Leben schwer machen. Er sucht nicht mehr Mitarbeiter, sondern die Tage, an denen zu viel Urlaub genommen wurde, um an genau diesen Tagen Urlaub zu verbieten.

Wie könnten wir die Tage bestimmen, an denen mehr als die Hälfte der Mitarbeiter Urlaub genommen hat?

Wieder mit einer Hashmap!

Was wäre, wenn ...

Angenommen, Dr. Meta will seine Mitarbeiter nicht beseitigen, sondern ihnen nur das Leben schwer machen. Er sucht nicht mehr Mitarbeiter, sondern die Tage, an denen zu viel Urlaub genommen wurde, um an genau diesen Tagen Urlaub zu verbieten.

Wie könnten wir die Tage bestimmen, an denen mehr als die Hälfte der Mitarbeiter Urlaub genommen hat?

Wieder mit einer Hashmap?

Was wäre, wenn ...

Angenommen, Dr. Meta will seine Mitarbeiter nicht beseitigen, sondern ihnen nur das Leben schwer machen. Er sucht nicht mehr Mitarbeiter, sondern die Tage, an denen zu viel Urlaub genommen wurde, um an genau diesen Tagen Urlaub zu verbieten.

Wie könnten wir die Tage bestimmen, an denen mehr als die Hälfte der Mitarbeiter Urlaub genommen hat?

Wieder mit einer Hashmap? Nö.

Was wäre, wenn ...

Wichtiger Unterschied zwischen den Mitarbeiter-IDs und den Nummern der Tage:

- Menge der möglichen Tage ist begrenzt
- zusätzlich: jeder Tag kann ein Urlaubstag sein

Generell: Nachteile von Hashing beachten erwartet konstante Laufzeit ist nicht $\Theta(1)$

Fragen?