Universität Bremen

# Praktische Informatik 2
# Manipulation von Mengen


Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

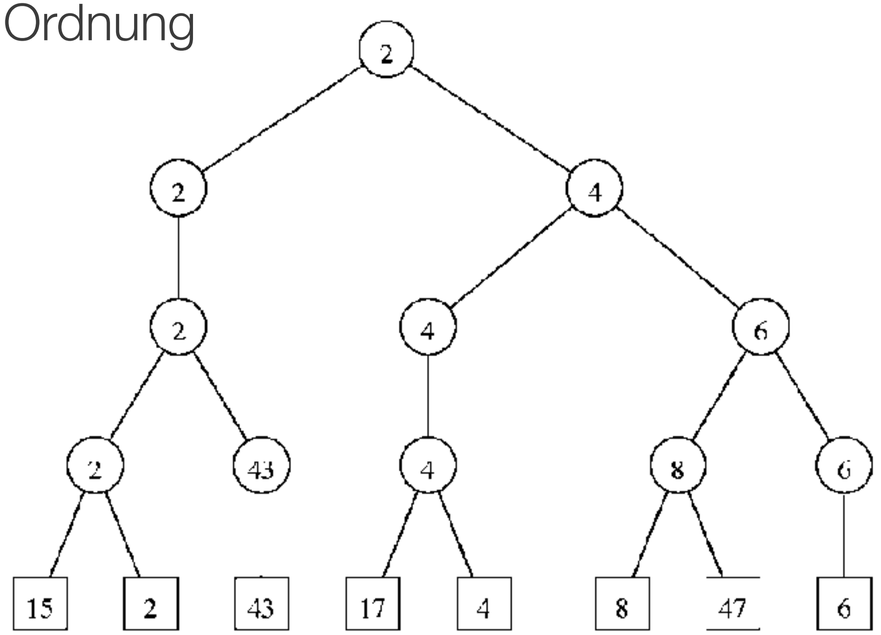
Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen



Bild: ©Michael Ihle

# Vorrangwarteschlange

- Als Vorrangwarteschlange (**priority queue**) wird eine Datenstruktur zur Speicherung einer Menge von Elementen bezeichnet, für die eine Ordnung definiert ist, so dass die folgenden Operationen ausführbar sind:
  - Initialisieren einer leeren Struktur
  - Einfügen eines Elements (**insert**)
  - Minimum suchen (**accessMin**)
  - Minimum entfernen (**deleteMin**)




- Können z.B. durch Listen oder balancierte Bäume implementiert werden
- Da Anforderungen geringer, können sie aber effizienter implementiert werden

Universität Bremen

# Vorrangwarteschlange

- Balancierte Bäume mit **n+1** Blättern und **n** inneren Knoten haben die Eigenschaft, dass jeder Pfad von der Wurzel zu einem Blatt eine Länge der Größenordnung **O(log n)** hat

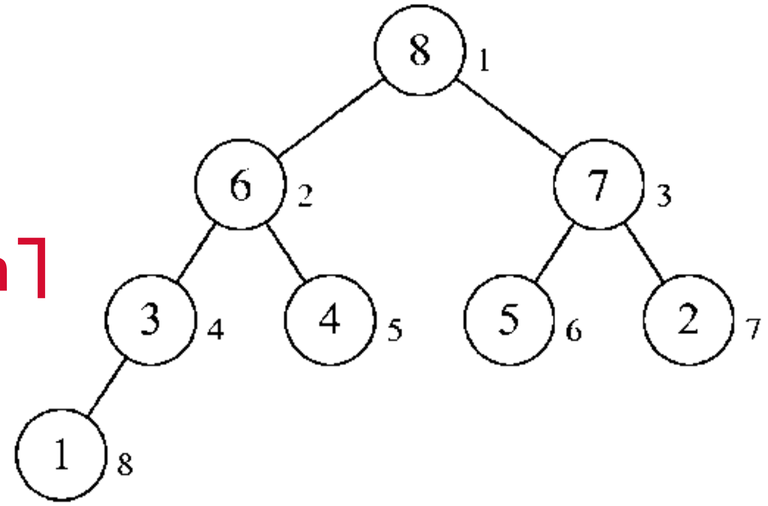
- Für eine Vorrangwarteschlange reicht eine wesentlich schwächere Forderung aus, um sicherzustellen, dass **accessMin** in **O(1)** und **insert**, **deleteMin** sowie das Verschmelzen zweier Schlangen in **O(log n)** ausführbar sind

- Anforderungen

  - Die Schlüsselwerte von Kindern müssen stets größer (oder gleich) sein als der ihres Elternknotens

  - Mindestens ein Blatt mit Tiefe **O(log n)**

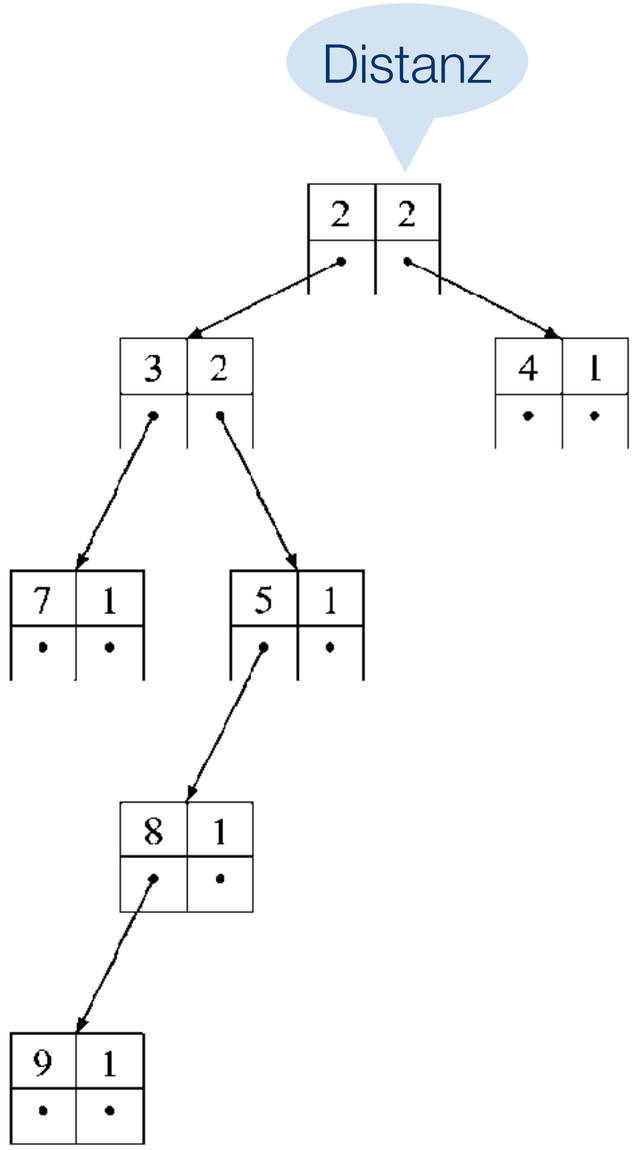
# Implementierungen



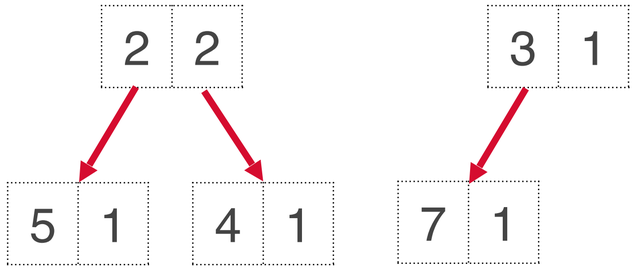

- **Heap**: In Array gespeicherter Binärbaum mit Höhe $\lceil \log_2 n \rceil$
  - Basisoperation: Versickern
  - Verschmelzen zweier Heaps in **O(log n)** geht aber nicht
- **Linksbaum**: Binärbaum, der eine größere Höhe haben kann
  - Wenn Einfüge- und Verschmelzungsoperationen entlang des kurzen Pfades durchgeführt werden, sind sie aber genauso effizient wie bei balancierten Bäumen
  - Basisoperation: Verschmelzen zweier Bäume

# Linksbaum: Anforderungen

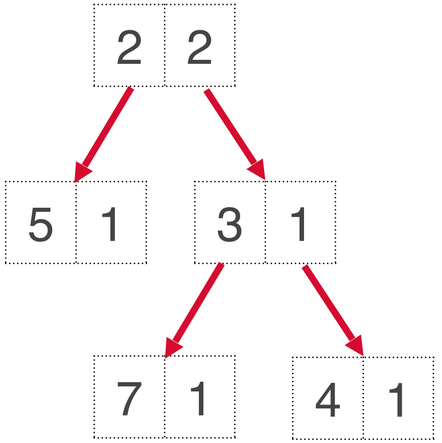
- Jeder Knoten speichert Distanz zum nächstgelegenen Blatt
- Der Schlüssel eines Knotens ist immer kleiner oder gleich den Schlüsseln seiner Kinder
- Die Distanz eines Knotens ist um 1 größer als das Minimum der Distanzen seiner beiden Kinder
  - **p.distance = 1 + min(p.left.distance, p.right.distance)**
- Distanz des linken Nachfolgers ist immer größer oder gleich der Distanz des rechten Nachfolgers
  - **p.left.distance ≥ p.right.distance**



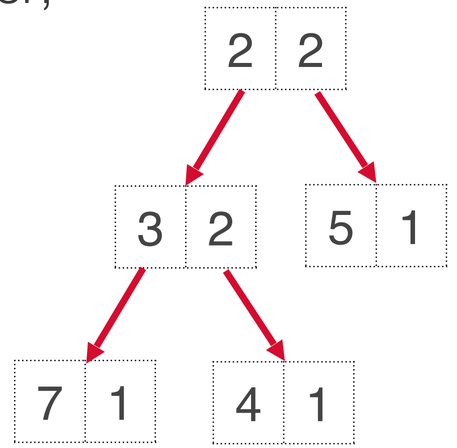
# Linksbaum




- **Basisoperation**: Verschmelzen zweier Linksbäume
  - Baum mit kleinerem Wert in Wurzel nach oben
  - **Rekursiver Abstieg**: Rechter Teilbaum dieser Wurzel wird mit anderem Teilbaum verschmolzen
  - **Rekursiver Aufstieg**: Teilbäume vertauschen, wenn Distanz rechts größer; Distanz aktualisieren

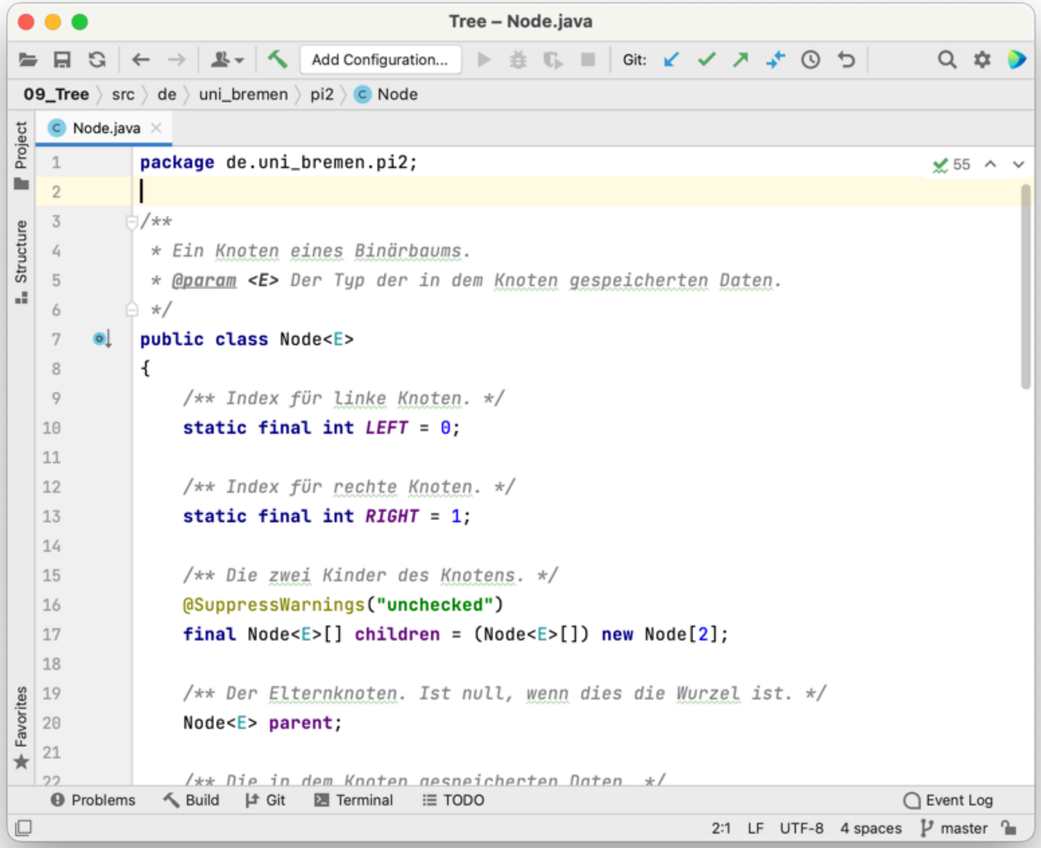



- **insert(x)**: Erzeuge einen neuen Linksbaum aus **x** und verschmelze den bisherigen Linksbaum mit dem neuen
- **deleteMin()**: Entferne Wurzel und verschmelze die beiden Teilbäume

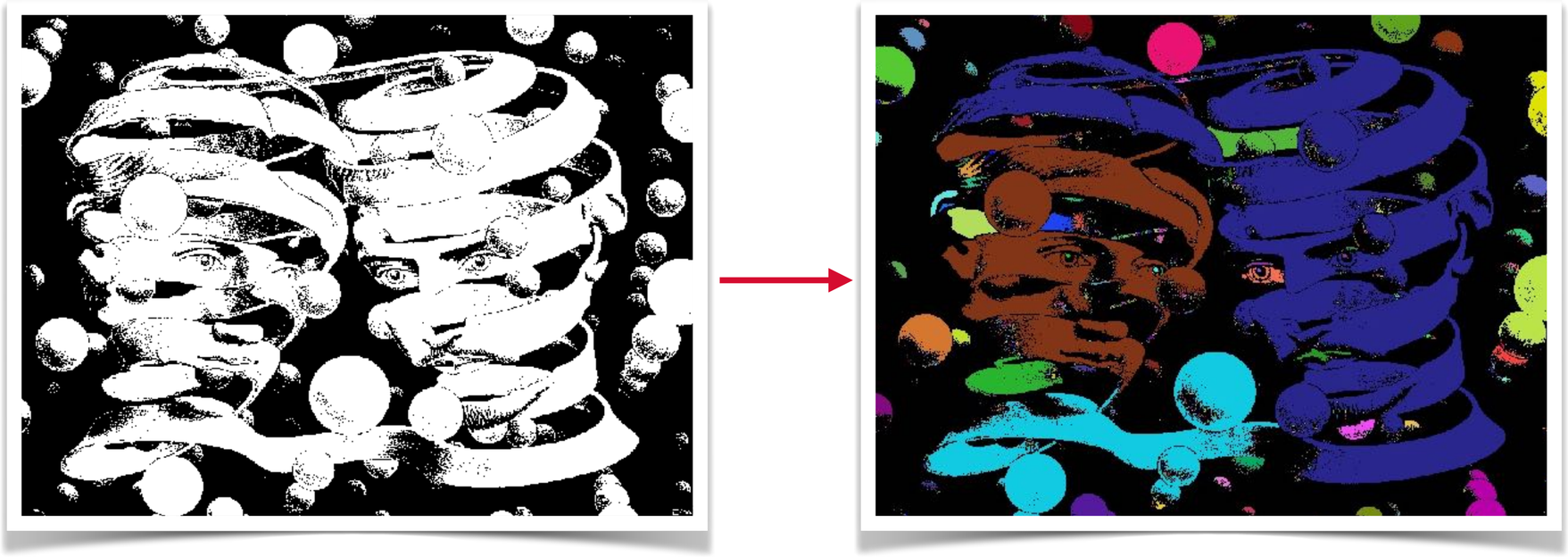


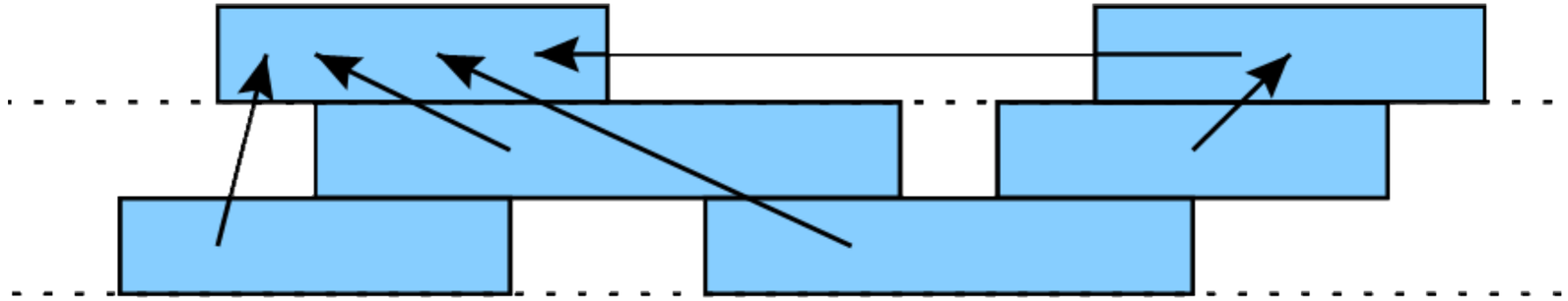
Universität Bremen

# Links-Baum: Demo



# Regionen vereinigen: Beispiel





Universität Bremen

# Union-Find-Strukturen

- In vielen Problemstellungen sollen Objekte (z.B. Knoten oder Kanten eines Graphen) in Äquivalenzklassen eingeteilt werden
- Dabei wird mit einer sehr feinen Einteilung begonnen, die durch sukzessive Vereinigung von Mengen vergröbert wird
- **makeSet(e, i)**: Erzeuge eine neue Menge mit Namen **i** mit dem einzigen Element **e**, das in keiner anderen Menge vorkommt
- **find(x)**: Liefert den Namen der Menge, die **x** enthält
- **union(i, j, k)**: Vereinigt zwei Mengen mit Namen **i** und **j** zu einer neuen Menge mit Namen **k** (Die Mengen mit Namen **i** und **j** werden gelöscht)

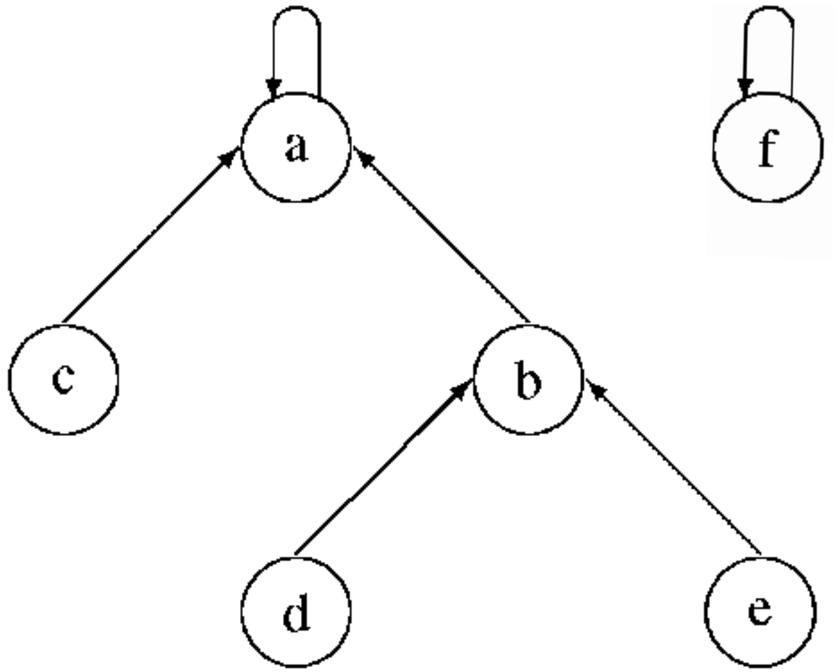
Universität Bremen

# Kanonisches Element

- Verschiedene Mengen enthalten nur verschiedene Elemente, weshalb jede Menge auch eindeutig durch eines ihrer Elemente repräsentiert werden kann
- Dieses **kanonische Element** kann in einer Menge frei gewählt werden und ihren Namen ersetzen
  - Kanonisches Element von **makeSet(e, i)** ist **e**, also reicht **makeSet(e)**
  - **find(x)** liefert kanonisches Element einer Menge
  - Vereinigung zweier Mengen kann durch kanonisches Element einer der beiden Mengen repräsentiert werden, also reicht **union(e, f)**
- **Union-Find-Problem**: Finden einer Datenstruktur zur Repräsentation von paarweise disjunkten Mengen und Algorithmen für **makeSet**, **find** und **union**

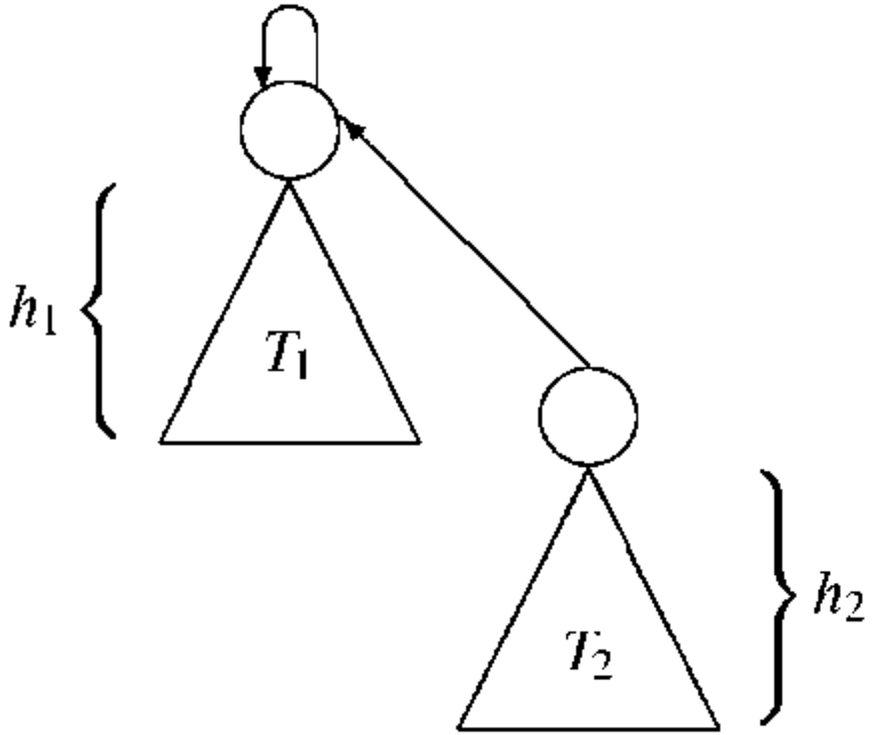
# Vereinigung von Mengen: Repräsentation

- Jede Menge der Kollektion wird durch einen unsortierten Baum repräsentiert
- Die Knoten des Baums sind die Elemente der Menge
- Die Wurzel des Baums enthält das kanonische Element
- Jeder Knoten enthält eine Referenz auf seinen Elternknoten, die Wurzel zeigt auf sich selbst

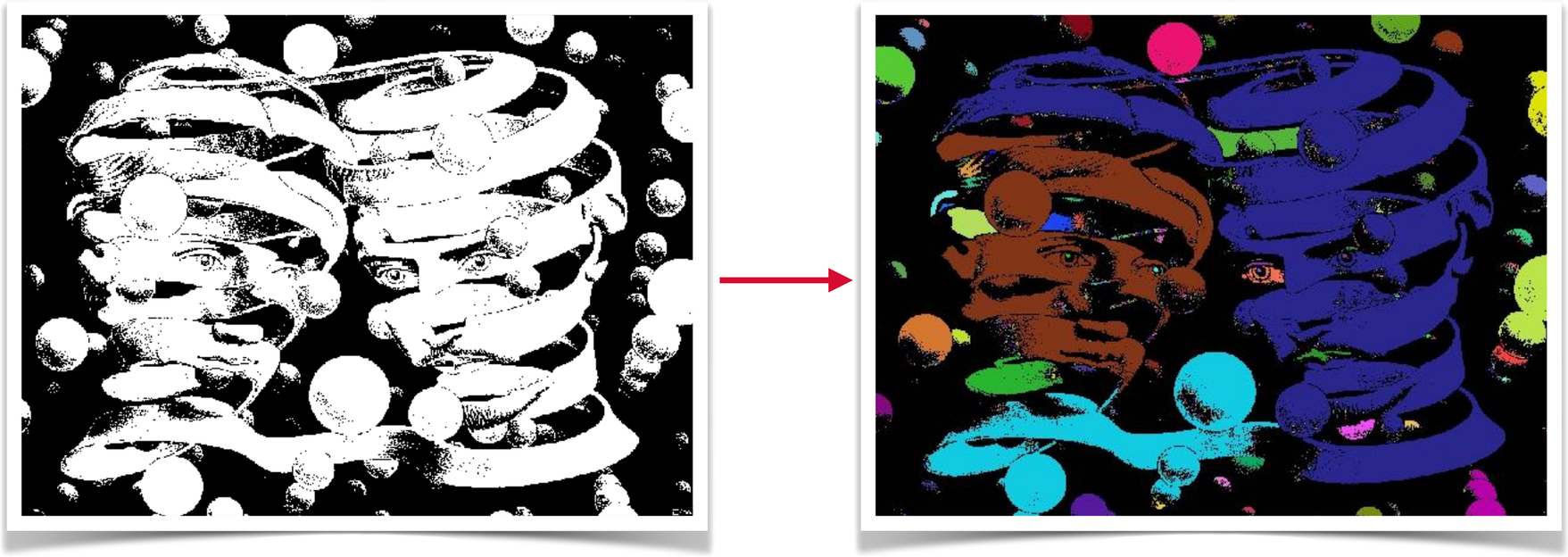


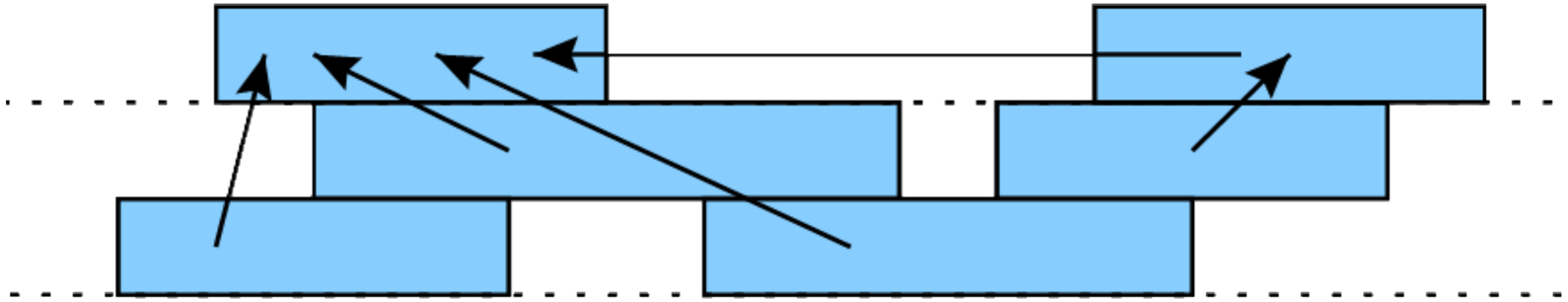
# Vereinigung von Mengen

- Hänge die Wurzel (kanonisches Element) des einen Baums an die Wurzel des anderen
- Vereinigung nach Größe bzw. Höhe
  - Vermeidung von degenerierten Bäumen
  - Hänge Baum mit weniger Knoten / geringerer Höhe an die Wurzel des größeren/höheren
- Andere Kriterien sind möglich



# Union-Find-Strukturen: Demo



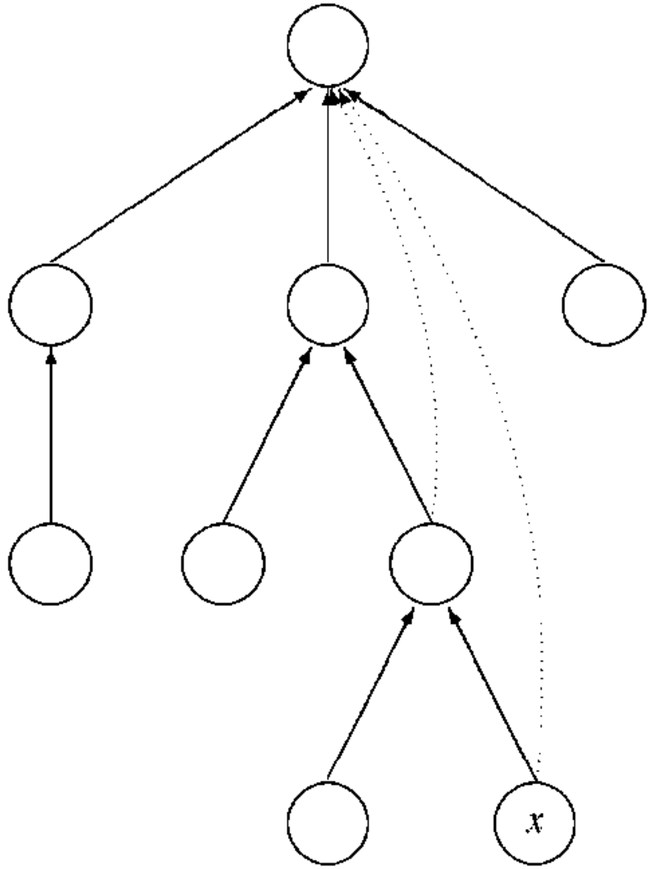


# Pfadverkürzung (Pfadkompression)

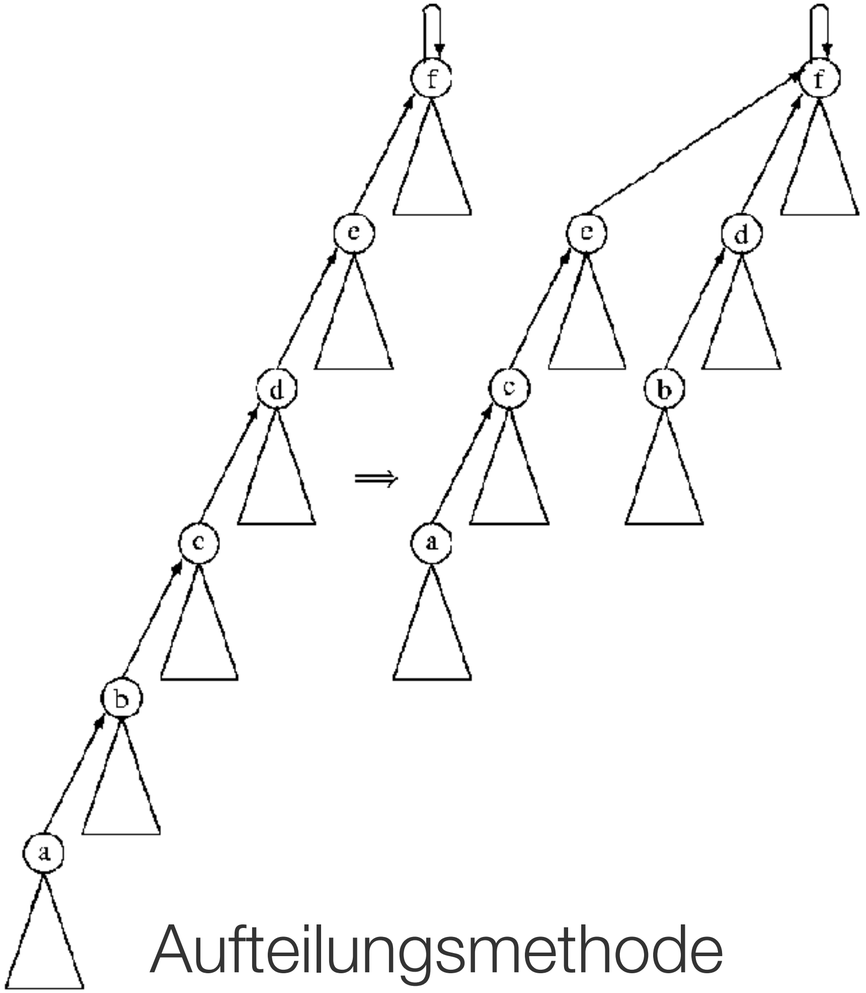
- Während der Suche wird der Pfad durch Umhängen von Kanten verkürzt, so dass bei der nächsten Suche weniger Kanten durchlaufen werden müssen

- **Aufteilungsmethode (Splitting)**: Verweise so umhängen, dass sie statt auf den Elternknoten auf den Großelternknoten zeigen

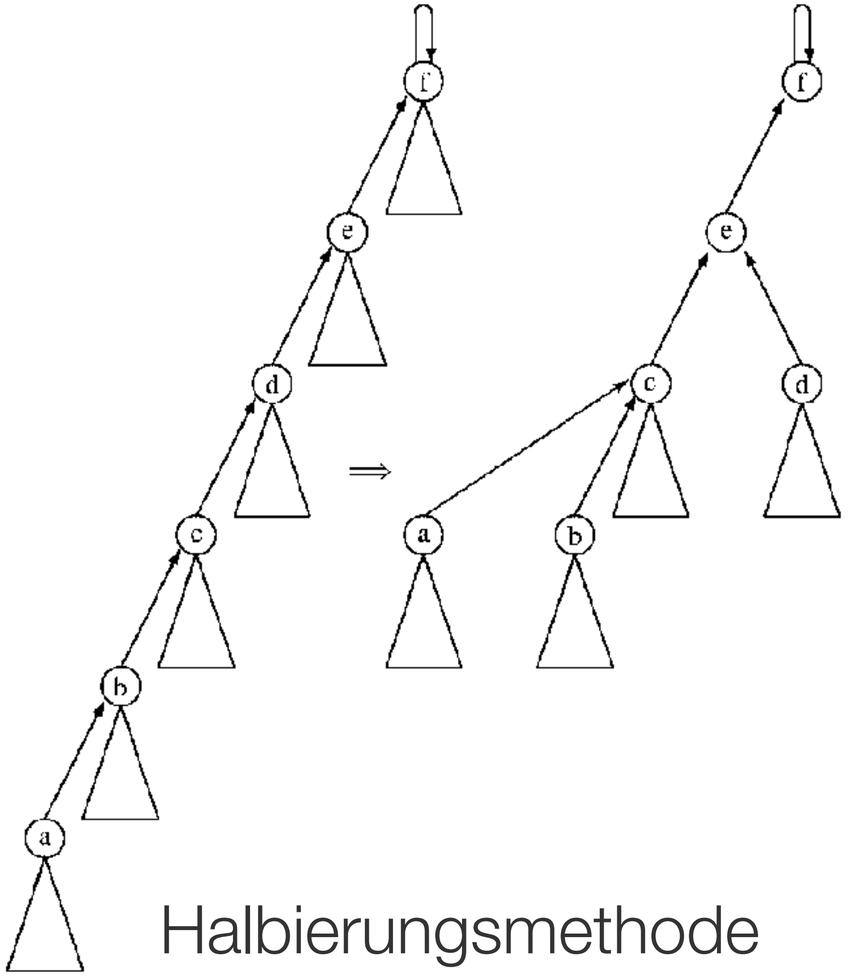
- **Halbierungsmethode**: Wie Aufteilungsmethode, aber nur für jeden zweiten Knoten



# Pfadverkürzung (Pfadkompression)




Aufteilungsmethode




Halbierungsmethode

# Mengenoperationen

- **Motivation**: Wörterbücher, Vorrangwarteschlangen und Union-Find-Strukturen können als Spezialfälle eines Mengenmanipulationsproblems aufgefasst werden

- Kollektion **K** von paarweise disjunkten Mengen

$$K = \left\{ S_{n_1}, \ldots, S_{n_t} \right\}, S_{n_i} \cap S_{n_j} = \varnothing \text{ für } i \neq j$$

- Elemente der Mengen gehören zu einem Universum **U**

$$U \supseteq \bigcup K = \left\{ x \in S \mid S \in K \right\}$$

- Die Namen der Mengen gehören zu einer Menge **N**

$$N \supseteq \left\{ n_i \mid S_{n_i} \in K \right\}$$

Universität Bremen

# Mengenoperationen

- **makeSet(x, n)**: Bilde eine Menge mit einzigem Element **x** und gib ihr den Namen **n** (**x** und **n** sind neu)
- **search(x, n)**: Suche **x** in der Menge mit Namen **n**
- **insert(x, n)**: Füge **x** in die Menge mit Namen **n** ein (**x** ist neu)
- **delete(x, n)**: Entferne **x** aus der Menge mit Namen **n**
- **find(x)**: Bestimme den Namen der Menge, die **x** enthält
- **union(i, j, k)**: Vereinige die Mengen mit Namen **i** und **j** zu einer Menge mit Namen **k**

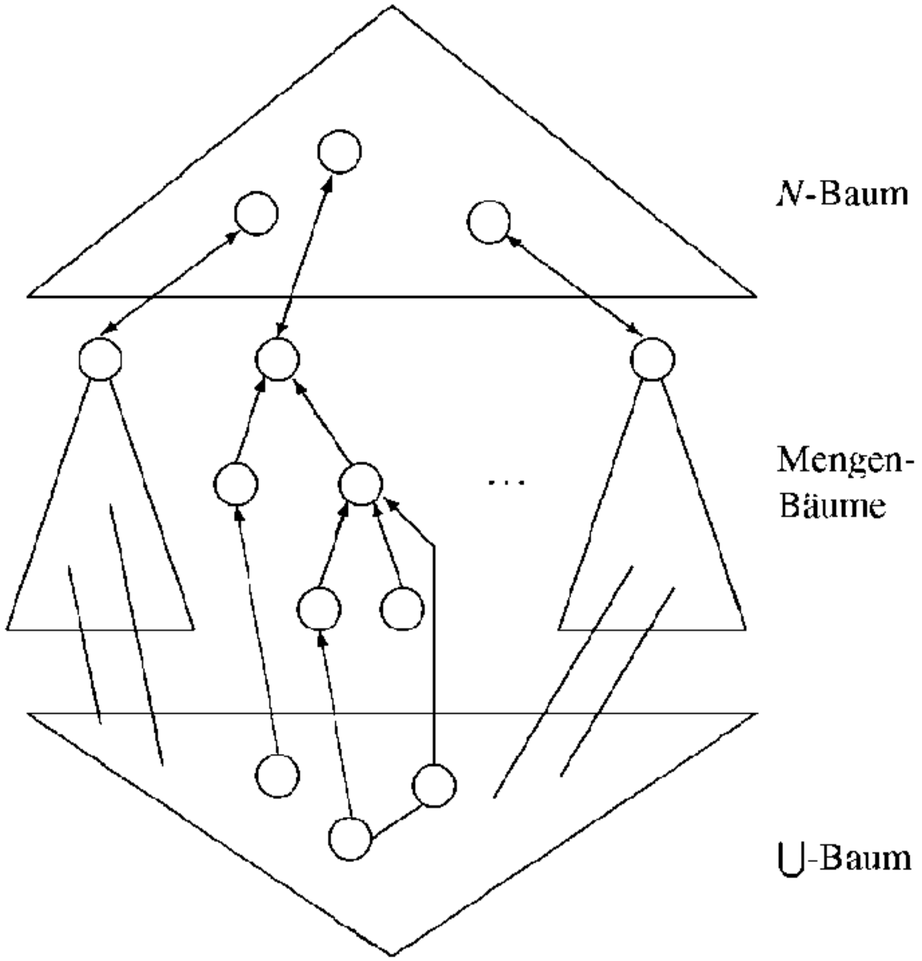
Universität Bremen

# Mengenoperationen

- **accessMin(n)**: Bestimme Minimum in der Menge mit Namen **n**
- **deleteMin(n)**: Entferne Minimum aus der Menge mit Namen **n**
- **successor(x, n)**: Bestimme das zu **x** nächst größere Element in der Menge mit Namen **n**
- **predecessor(x, n)**: Bestimme das zu **x** nächst kleinere Element in der Menge mit Namen **n**
- **kthElement(k)**: Bestimme das **k**-größte Element in $\cup$**K**

# Modellierung: N-Baum

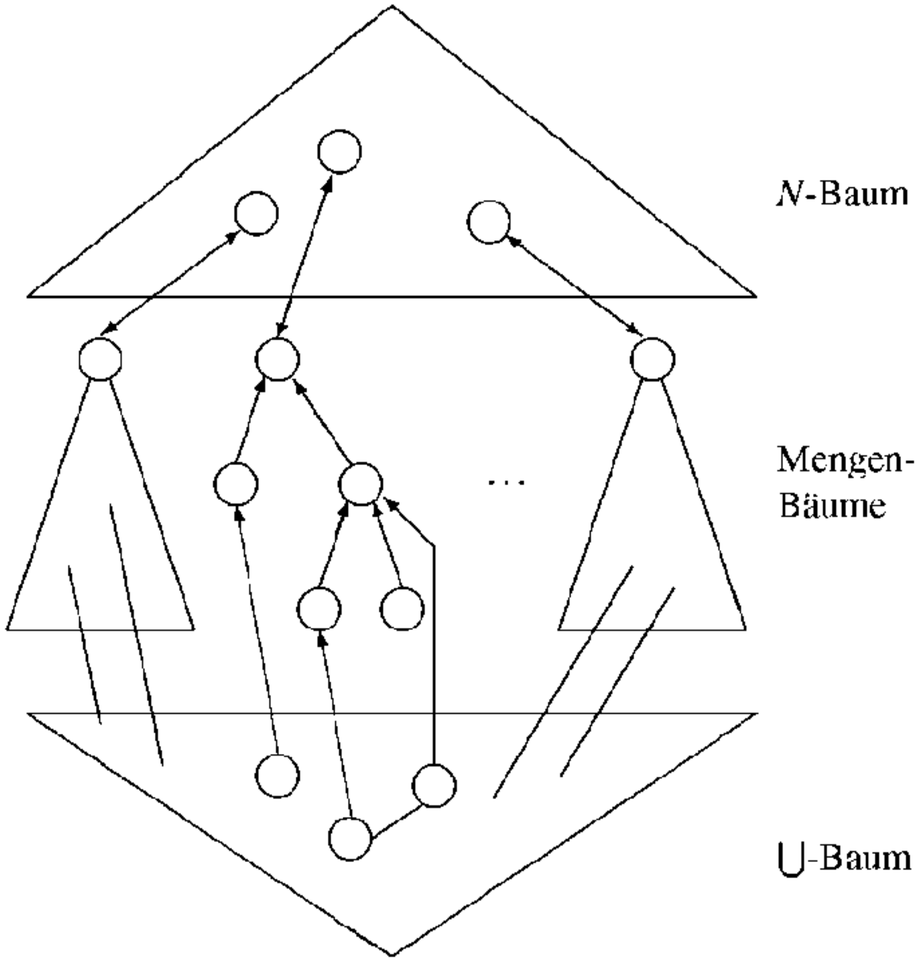
- Die Menge aller Namen von Mengenbäumen ist in einem balancierten, sortierten Baum gespeichert
- Die Wurzel jedes Mengenbaums ist durch Referenzen in beide Richtungen mit seinem Namen im N-Baum verbunden



# Modellierung: U-Baum

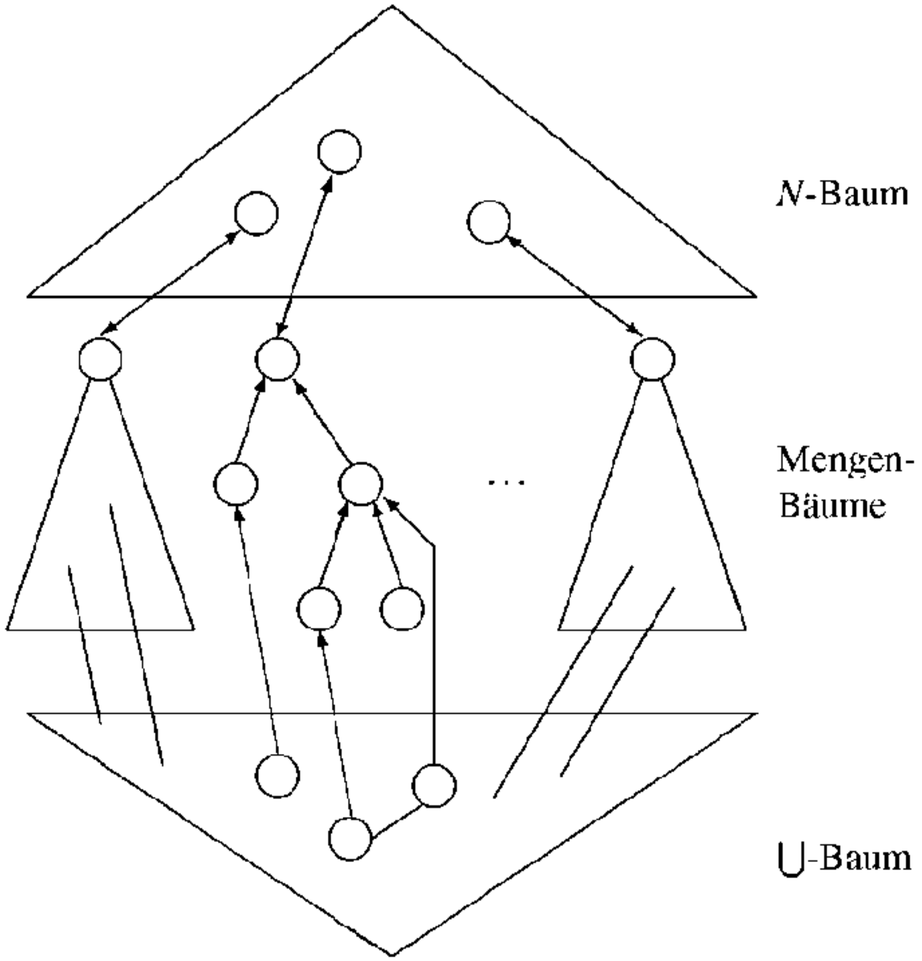
- $\cup$**K** $\subseteq$ **U** durch balancierten, sortierten Binärbaum repräsentieren

- Soll die Operation **k**-tes Element unterstützt werden, enthält jeder Knoten **p** im Baum einen Zähler **z(p)** darüber, wie viele Schlüssel im Teilbaum mit Wurzel **p** enthalten sind



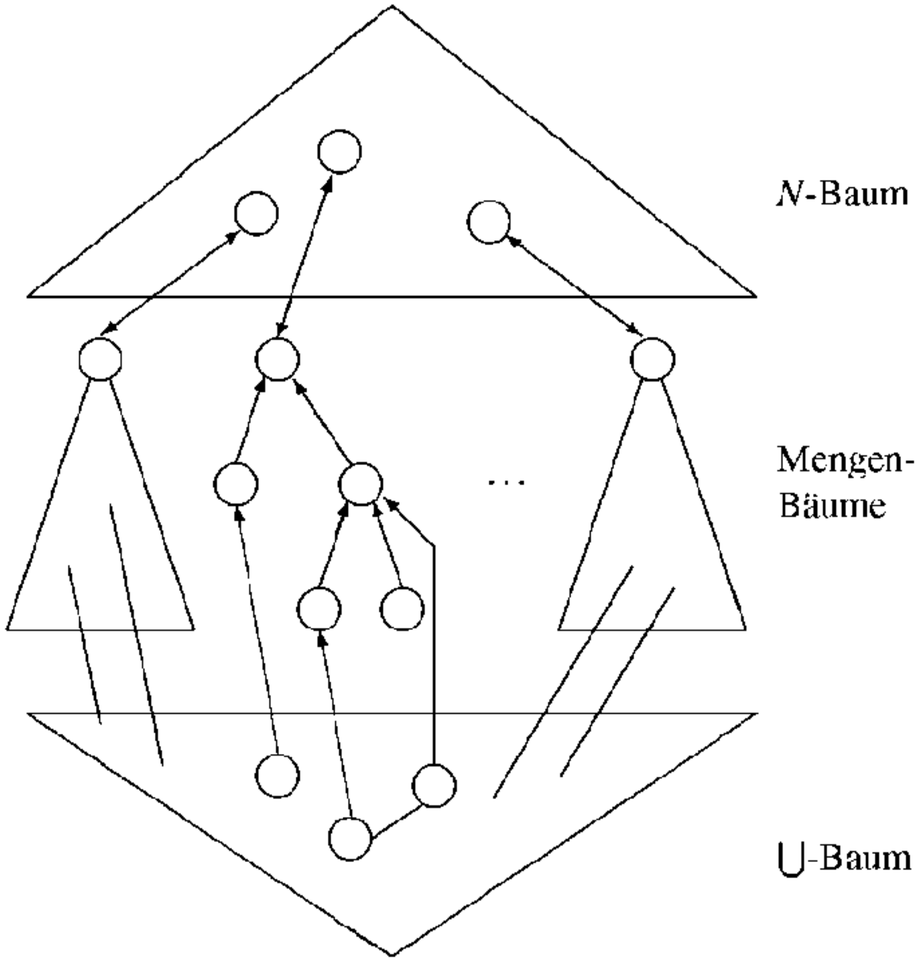
# Modellierung: Mengenbäume

- Jede Menge **$S_i$** der Kollektion **K** durch einen nicht-sortierten Mengenbaum darstellen
  - **Heap-geordnet**, falls **accessMin** unterstützt wird
- Der Knoten **x** im U-Baum ist durch eine Referenz mit dem Knoten **x** im Mengenbaum **$S_i$** verbunden, wenn **$x \in S_i$**



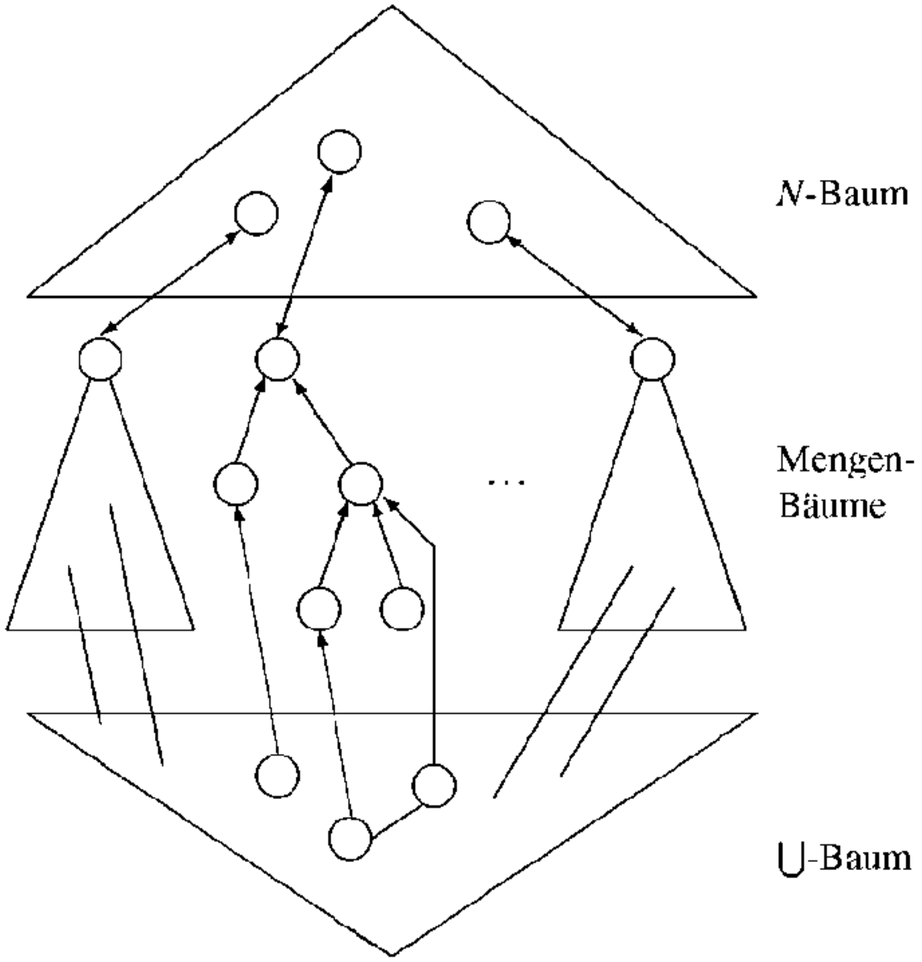
# Umsetzung der Operationen

- **find(x)**: Jeder Knoten eines Mengenbaums zeigt auf seinen Elternknoten
  - **x** im U-Baum finden, von dort in den Mengenbaum wechseln, der **x** enthält, dann hoch zur Wurzel
- **accessMin(n)**, **deleteMin(n)**: Mengenbäume sind Heap-geordnet
- **union(i, j, k)**: Falls Vereinigung nach Größe oder Höhe, dann muss diese Größe in der Wurzel der Mengenbäume mitgeführt werden



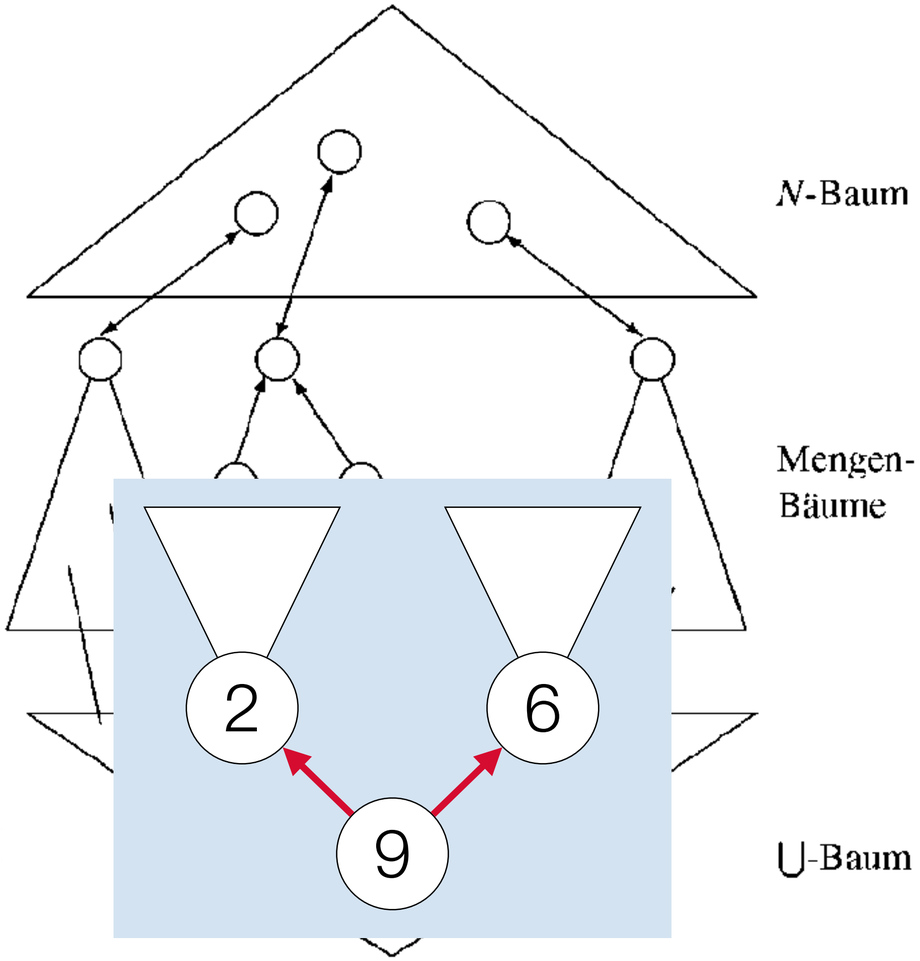
# Umsetzung der Operationen

- **insert(x, i)**: Füge **x** in U-Baum ein
  - Suche **i** im N-Baum
  - Folge Referenz zur Wurzel des $S_i$-Baums
  - Füge **x** in $S_i$-Baum ein
- **delete(x, i)**: **find(x)**
  - Wenn der gefundene Mengenbaum tatsächlich **i** heißt, dann entferne **x** aus $S_i$ und dem U-Baum



# Umsetzung der Operationen: **kthElement(k)**

- Beginne bei Wurzel **p** des U-Baums
- Falls **z(p) < k**, dann gibt es kein **k**-tes Element im U-Baum
- Falls **k = z(left(p)) + 1**, dann enthält **p** das **k**-te Element
- Falls **k ≤ z(left(p))**, dann suche in **links(p)** weiter
- Ansonsten suche das **(k - z(left(p)) - 1)**-te Element in **right(p)**



Universität Bremen

# Zusammenfassung der Konzepte

- **Linksbaum**
- **Union-Find-Strukturen**
  - **Pfadverkürzung**
- **Mengenoperationen**