

## Manipulation von Mengen

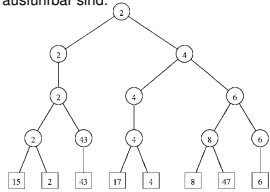
Thomas Röfer

- Vorrangwarteschlange
- Linksbaum
- Heap
- HeapSort
- Union-Find-Strukturen
- Allgemeiner Rahmen für Mengenmanipulationen

## Rückblick „Hashing“

## Vorrangwarteschlange

- Definition**
  - Als Vorrangwarteschlange (*priority queue*) bezeichnet man eine Datenstruktur zur Speicherung einer Menge von Elementen, 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*)
- Anmerkung**
  - Vorrangwarteschlangen können z.B. durch Listen oder balancierte Bäume implementiert werden.
  - Allerdings sind die Anforderungen geringer, daher können sie durch geeignete Strukturen effizienter implementiert werden.

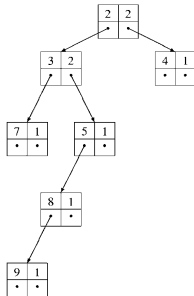


## Vorrangwarteschlange

- Motivation**
  - Balancierte Bäume haben die Eigenschaft, dass jeder Pfad von der Wurzel zu einem Blatt des Baumes mit  $n+1$  Blättern und  $n$  inneren Knoten eine Länge der Größenordnung  $O(\log n)$  hat
  - Für eine Vorrangwarteschlange reicht eine wesentlich schwächere Forderung aus, um zu sichern, dass *accessMin* in  $O(1)$  und *insert*, *deleteMin* sowie das Verschmelzen zweier Schlangen in  $O(\log n)$  ausführbar ist
- Anforderungen**
  - Die Schlüsselwerte von Kindern müssen stets größer (oder gleich) sein als der ihres Elternknotens
  - Es muss mindestens einen Pfad von der Wurzel zu einem Blatt mit  $O(\log n)$  Länge geben
  - Wenn Einfüge- und Verschmelzeoperationen entlang des kurzen Pfades durchgeführt werden, sind sie genauso effizient wie bei balancierten Bäumen

## Linksbaum

- Ein binärer Baum heißt Linksbaum, wenn gilt:**
  - Jeder innere Knoten enthält neben dem Schlüssel und den zwei Zeigern auf seine Kinder noch einen Distanzwert, der die Entfernung zum nächstgelegenen Blatt enthält
  - Blätter haben die Distanz 0
  - 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
    - $p.dist = 1 + \min(p.left.dist, p.right.dist)$
  - Die Distanz des linken Nachfolgers ist immer größer oder gleich der Distanz des rechten Nachfolgers
    - $p.left.dist \geq p.right.dist$



## Linksbaum

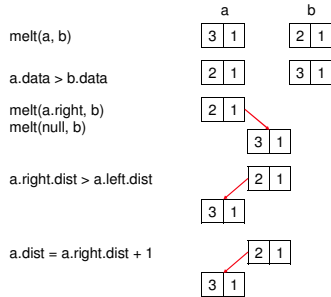
- Basisoperation**
  - Verschmelzen zweier Links bäume
- insert(x)**
  - Erzeuge einen neuen Linksbaum aus x
  - Verschmelze den bisherigen Linksbaum mit dem neuen
- deleteMin()**
  - Entferne die Wurzel
  - Verschmelze die beiden Teilbäume

```

class LeftTree
{
    Comparable data;
    int distance;
    LeftTree left, right;

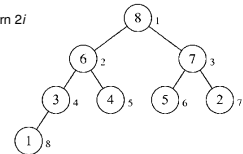
    static LeftTree melt(LeftTree a, LeftTree b)
    {
        if(a == null)
            return b;
        else if(b == null)
            return a;
        else
        {
            if(a.data.compareTo(b.data) > 0)
                // b ist kleiner -> vertausche a und b
                a.right = melt(a.right, b);
            if(getDistance(a.right) > getDistance(a.left))
                // Vertausche a.left und a.right
                a.distance = getDistance(a.right) + 1;
            return a;
        }
    }
}
    
```

### Linksbaum – Beispiel



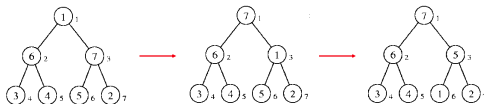
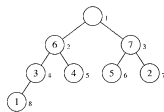
### Heap

- Definition**
  - Eine Folge  $F = k_1, k_2, \dots, k_N$  von Schlüsselwerten nennen wir einen **Heap (die Halde)**, wenn  $k_i \leq k_{\lfloor i/2 \rfloor}$  für  $2 \leq i \leq N$  gilt
  - Anders ausgedrückt:  $k_i \geq k_{2i}$  und  $k_i \geq k_{2i+1}$ , sofern  $2i \leq N$  bzw.  $2i+1 \leq N$
  - Hat nichts mit dem Java-Heap zu tun
- Beispiel**
  - $F = 8, 6, 7, 3, 4, 5, 2, 1$  genügt der Heap-Bedingung, weil  $8 \geq 6, 8 \geq 7, 6 \geq 3, 6 \geq 4, 7 \geq 5, 7 \geq 2, 3 \geq 1$
- Ausgabe in absteigender Reihenfolge**
  - Solange Heap nicht leer ist
    - gib  $k_i$  aus
    - entferne  $k_i$  aus dem Heap
    - stelle Heap-Bedingung für restliche Schlüssel wieder her, so dass die neue Wurzel an Position 1 steht



### Wiederherstellen der Heap-Bedingung

- Heap-Bedingung wieder herstellen**
  - Nach Entfernen der Wurzel sind zwei Teil-Heaps übrig
  - Vereinige beide, indem der Schlüssel mit dem höchsten Index als Wurzel eingesetzt wird
  - Dann lassen wir  $k_i$  **versickern (sift down)**, indem er so lange mit dem jeweils größeren Nachfolger vertauscht wird, bis beide Nachfolger kleiner sind oder der Schlüssel unten angekommen ist

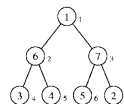


### Implementierung des Versickerns

- Vorteile**
  - Heap kann in einem Array implementiert werden
  - Aufwand für das Versickern ist  $O(\log N)$

```

static void siftDown(Comparable a[],
                    int i, int m)
{
    while (2 * i < m)
    {
        int j = 2 * i + 1;
        if (j < m && a[j].compareTo(a[j + 1]) < 0)
            ++j;
        if (a[i].compareTo(a[j]) < 0)
        {
            swap(a, i, j);
            i = j;
        }
        else
            break;
    }
}
    
```



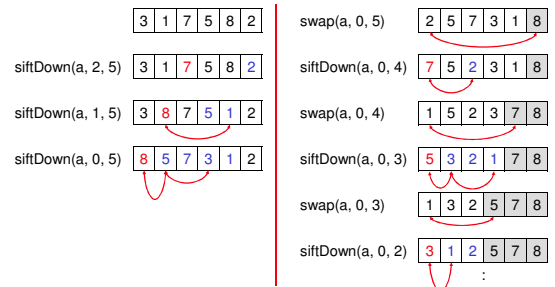
### HeapSort

- Ansatz**
  - Zuerst wird ein Heap aufgebaut, indem der Reihe nach vorne ein Element vorangestellt wird und dieses dann versickert
  - Dann wird der Heap schrittweise geleert, wodurch jeweils das größte Element entnommen wird
- Anmerkungen**
  - Das Voranstellen passiert nicht wirklich, stattdessen wächst der Heap einfach vom Ende des Arrays her
  - Nachdem Entfernen des jeweils größten Elements ist der Heap am Ende um ein Element kürzer. Dort kann das soeben entnommene Element eingetragen werden
- Aufwand**
  - $N/2$  mal Versickern für das Aufbauen des Heaps:  $O(N/2 \log N)$
  - $N$  mal Versickern für das Auslesen des Heaps:  $O(N \log N)$
  - Insgesamt:  $O(N \log N)$
  - Vorteil: Speicherkomplexität ist  $O(N)$

```

static void heapSort(Comparable a[])
{
    for (int i = a.length / 2 - 1; i >= 0; --i)
        siftDown(a, i, a.length - 1);
    for (int i = a.length - 1; i > 0; --i)
    {
        swap(a, 0, i);
        siftDown(a, 0, i - 1);
    }
}
    
```

### HeapSort – Beispiel



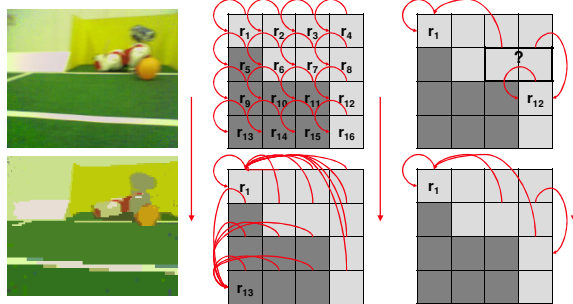
## Union-Find-Strukturen

- Motivation**
  - In vielen Problemstellungen möchte man Objekte (z.B. Knoten oder Kanten eines Graphen) in Äquivalenzklassen einteilen
  - Dabei beginnt man mit einer sehr feinen Einteilung, die durch sukzessive Vereinigung von Mengen vergrößert wird
- Basisoperationen**
  - `makeSet(e, i)`
    - Erzeuge eine neue Menge mit Namen  $i$  mit einzigem Element  $e$
    - $e$  ist neu, kommt also in keiner anderen Menge vor
  - `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

## Kanonisches Element

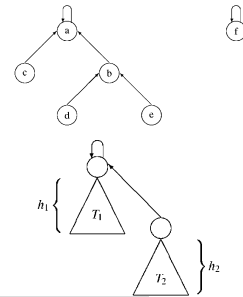
- Ansatz**
  - Verschiedene Mengen enthalten auch nur verschiedene Elemente
  - Daher kann jede Menge auch eindeutig durch eines ihrer Elemente repräsentiert werden, d.h. die Namen der Mengen werden nicht benötigt
  - Dieses „kanonische Element“ kann in einer Menge frei gewählt werden
- Verzicht auf Namen**
  - Das kanonische Element einer mit `makeSet(e, i)` erzeugten Menge ist  $e$ , also reicht `makeSet(e)`
  - `find(x)` liefert das kanonische Element einer Menge
  - Vereinigt man zwei Mengen, kann das Ergebnis wieder durch das kanonische Element einer der beiden Mengen repräsentiert werden, also reicht `union(e, i)`
- Union-Find-Problem**
  - Finden einer Datenstruktur zu Repräsentation einer Kollektion von paarweise disjunkten Mengen sowie Algorithmen für die Ausführung von `makeSet`, `find` und `union`

## Regionen vereinen



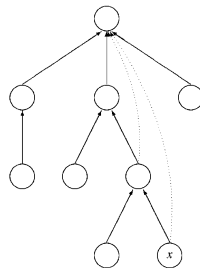
## Vereinigung von Mengen

- Implementierung**
  - 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 einen Zeiger auf seinen Elternknoten, die Wurzel zeigt auf sich selbst
- Vereinigung**
  - Hänge die Wurzel (kanonisches Element) des einen Baums an die Wurzel des anderen
- Vereinigung nach Größe bzw. Höhe**
  - Motivation: Vermeidung von degenerierten Bäumen
  - Hänge den Baum mit weniger Knoten bzw. geringerer Höhe an die Wurzel des größeren bzw. höheren

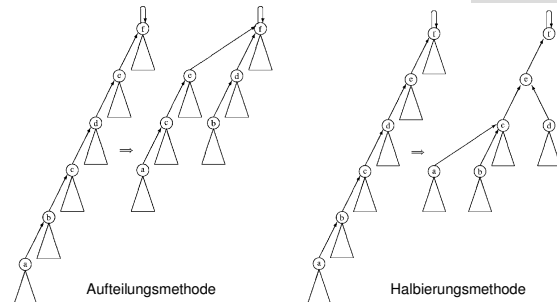


## Pfadverkürzung (Pfadkompression)

- Ziel**
  - Während der Suche wird der Pfad durch Umhängen von Kanten verkürzt, so dass bei der nächsten Suche weniger Kanten durchlaufen werden müssen
- Aufteilungsmethode (Splitting)**
  - Während der Suche werden Verweise so umgehängt, dass sie statt auf den Elternknoten auf den Großelternknoten zeigen
- Halbierungsmethode**
  - Während der Suche werden die Verweise jedes zweiten Knoten so umgehängt, dass sie statt auf den Elternknoten auf den Großelternknoten zeigen



## Pfadverkürzung (Pfadkompression)



## Allgemeiner Rahmen

- Motivation**
  - Wörterbücher, Vorrangwarteschlangen und Union-Find-Strukturen kann man als Spezialfälle eines Mengenmanipulationsproblems auffassen
- Gegeben**
  - Kollektion  $K$  von paarweise disjunkten Mengen
 
$$K = \{S_{n_1}, \dots, S_{n_t}\}, S_{n_i} \cap S_{n_j} = \emptyset, \text{ für } i \neq j$$
  - Elemente der Mengen gehören zu einem Universum  $U$ 

$$U \supseteq \bigcup K = \{x \in S \mid S \in K\}$$
  - Die Namen der Mengen gehören zu einer Menge  $N$ 

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

## Mengenoperationen

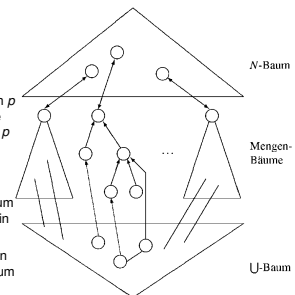
- makeSet(x, n)**
  - Bilde eine Menge mit einzigem Element  $x$  und gebe ihr den Namen  $n$ 
    - Voraussetzung:  $x$  und  $n$  sind neu
- search(x, n)**
  - Suche  $x$  in der Menge mit Namen  $n$
- insert(x, n)**
  - Füge  $x$  in die Menge mit Namen  $n$  ein
    - Voraussetzung:  $x$  ist neu
- 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  $k$

## Mengenoperationen

- accessMin(n)**
  - Bestimme das Minimum in der Menge mit Namen  $n$
- deleteMin(n)**
  - Entferne das Minimum aus der Menge mit Namen  $n$
- successor(x, n)**
  - Bestimme das zu  $x$  nächstgrößere Element in der Menge mit Namen  $n$
- predecessor(x, n)**
  - Bestimme das zu  $x$  nächstkleinere Element in der Menge mit Namen  $n$
- kthElement(k)**
  - Bestimme das  $k$ -größte Element in  $UK$

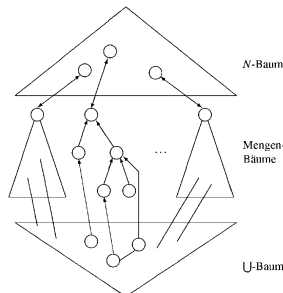
## Modellierung

- U-Baum**
  - Repräsentiere  $UK \subseteq U$  durch einen balancierten, sortierten Binärbaum
  - 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
- Mengenbäume**
  - Stelle jede Menge  $S_i$  der Kollektion  $K$  durch einen nicht-sortierten Mengenbaum dar (oder heap-geordnet, falls **accessMin** unterstützt wird).
  - Der Knoten  $x$  im U-Baum ist durch einen Zeiger mit dem Knoten  $x$  im Mengenbaum  $S_i$  verbunden, wenn  $x \in S_i$



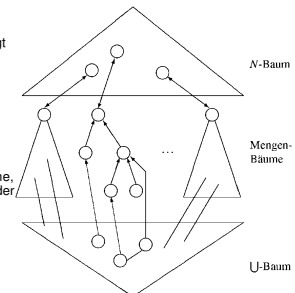
## Modellierung

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



## Umsetzung der Operationen

- find(x)**
  - Jeder Knoten eines Mengenbaums zeigt auf seinen Elternknoten
  - $x$  im U-Baum finden, von dort in den Mengenbaum, der  $x$  enthält, wechseln, 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
- insert(x, i)**
  - Füge  $x$  in U-Baum ein
  - Suche  $i$  im N-Baum
  - Folge Zeiger zur Wurzel des  $S_i$ -Baums
  - Füge  $x$  in  $S_i$ -Baum ein



## Umsetzung der Operationen

- ▶ **delete(x, l)**
  - ▶ find(x)
  - ▶ Wenn der gefundene Mengenbaum tatsächlich  $l$  heißt, dann entferne  $x$  aus  $S$  und dem U-Baum
- ▶ **kthElement(k)**
  - ▶ Beginne bei Wurzel  $p$  des U-Baums
  - ▶ Falls  $z(p) < k$ , dann gibt es kein  $k$ -tes Element im U-Baum
  - ▶ Falls  $k = z(\text{left}(p)) + 1$ , dann enthält  $p$  das  $k$ -te Element
  - ▶ Falls  $k \leq z(\text{left}(p))$ , dann suche in  $\text{left}(p)$  weiter
  - ▶ Ansonsten suche das  $(k - z(\text{left}(p)) - 1)$ -te Element in  $\text{right}(p)$

