Algorithmen & Komplexität

Johannes Lengler
Institut für Theoretische Informatik

Kapitel 4:

Datenstrukturen

Datenstrukturen

Suchbäume:

- Operationen: *Find, Insert, Delete*

Mengen:

- Operationen: *Find, Insert, Union*

Wörterbücher:

- Operationen: *Find, Insert, Delete*

Vorrangwarteschlangen:

- Operationen: *Insert, ExtractMin, DecreaseKey*

Kapitel 4:

Datenstrukturen

Kapitel 4.2:

Mengen

Anwendungsbeispiel: Algorithmus von Kruskal

Algorithmus 2.9 Algorithmus von Kruskal

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$

Ausgabe: ...nnbaum $T = (V, F)$ in N

{ *Initial...* }

Sortiere ... $\ell(e_1) \leq \ldots \leq \ell(e_$

Setze $F := \emptyset$.

{ *Aufbau des Baums* }

for $i := 1$ **to** m **do**

  if $(V, F \cup \{e_i\})$ ist kreisfrei **then** $F := F \cup \{e_i\}$.

$\forall v \in V$:
MakeNewSet(v)

Let $e_i = \{x,y\}$:

Union(x, y)

Let $e_i = \{x,y\}$:

if Find(x) $\neq$ Find(y) then …

Union-Find-Strukturen

Gegeben:

Datensätze partitioniert in (paarweise disjunkte) Mengen, wobei jede Menge durch einen in ihr enthaltenen Datensatz repräsentiert werde.

Gesucht:

Datenstruktur, die folgende Operationen unterstützt.

MakeNewSet(x)	→	Füge neuen Datensatz x ein; dieser bildet eine neue (einelementige) Menge.
Find(x)	→	Gebe Repräsentanten derjenigen Menge aus, die x enthält.
Union(x,y)	→	Vereinige die beiden Mengen, zu denen x und y gehören.

Grundversion

Gegeben:

Datensätze partitioniert in (paarweise disjunkte) Mengen, wobei jede Menge durch einen in ihr enthaltenen Datensatz repräsentiert werde.

Idee:

Jede Menge wird durch einen zur Wurzel hin gerichteten Baum dargestellt (engl. *intree*). An jeder Wurzel ist gespeichert:

- Repräsentant der Menge
- Höhe des Baumes

MakeNewSet(x) → x bildet einen Baum, der nur aus der Wurzel besteht;

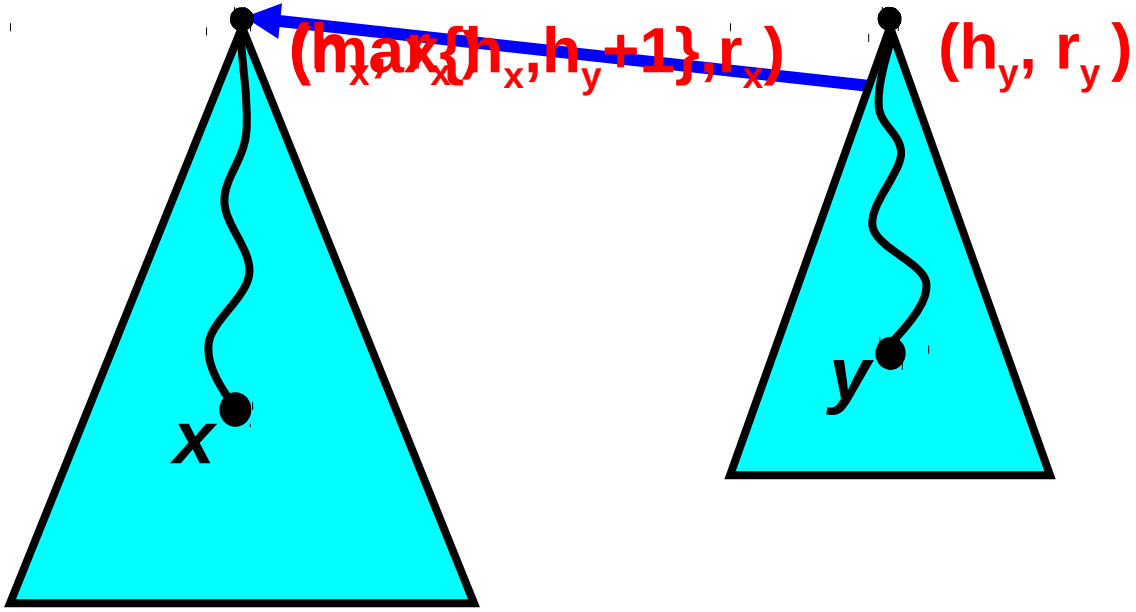
Höhe := 0, Repräsentant := x

Find(x) → Laufe von x zur Wurzel; gebe den an der Wurzel gespeicherten Repräsentanten aus,

Union(x,y) → Vereinige die Bäume …

Union(x,y)

Informationen an der Wurzel müssen angepasst werden.



$h_y \leq h_x$: Hänge Wurzel von y an die Wurzel von x an …

Eigenschaften

Lemma:
Für jeden in der Union-Find-Struktur enthaltenen Baum T gilt

$$\text{size}(T) \geq 2^{\text{height}(T)}$$

wobei $\text{size}(T)$ die Anzahl der Knoten in T und $\text{height}(T)$ die Höhe von T sei.

Korollar:
Für eine Union-Find Struktur mit n Elementen haben Find(x) und Union(x,y) Laufzeit **O(log(n)).**

Datenstrukturen

Suchbäume:

- Operationen: *Find, Insert, Delete*

Mengen:

- Operationen: *Find, Insert, Union*

Wörterbücher:

- Operationen: *Find, Insert, Delete*

Vorrangwarteschlangen:

- Operationen: *Insert, ExtractMin, DecreaseKey*

Kapitel 4.3:

Wörterbücher (dictionaries)

Operationen: Find, Insert, Delete

Ziel: alles in Zeit O(1) !!

Beispiel: ISBN-Nummern (13-stellige Nummern)

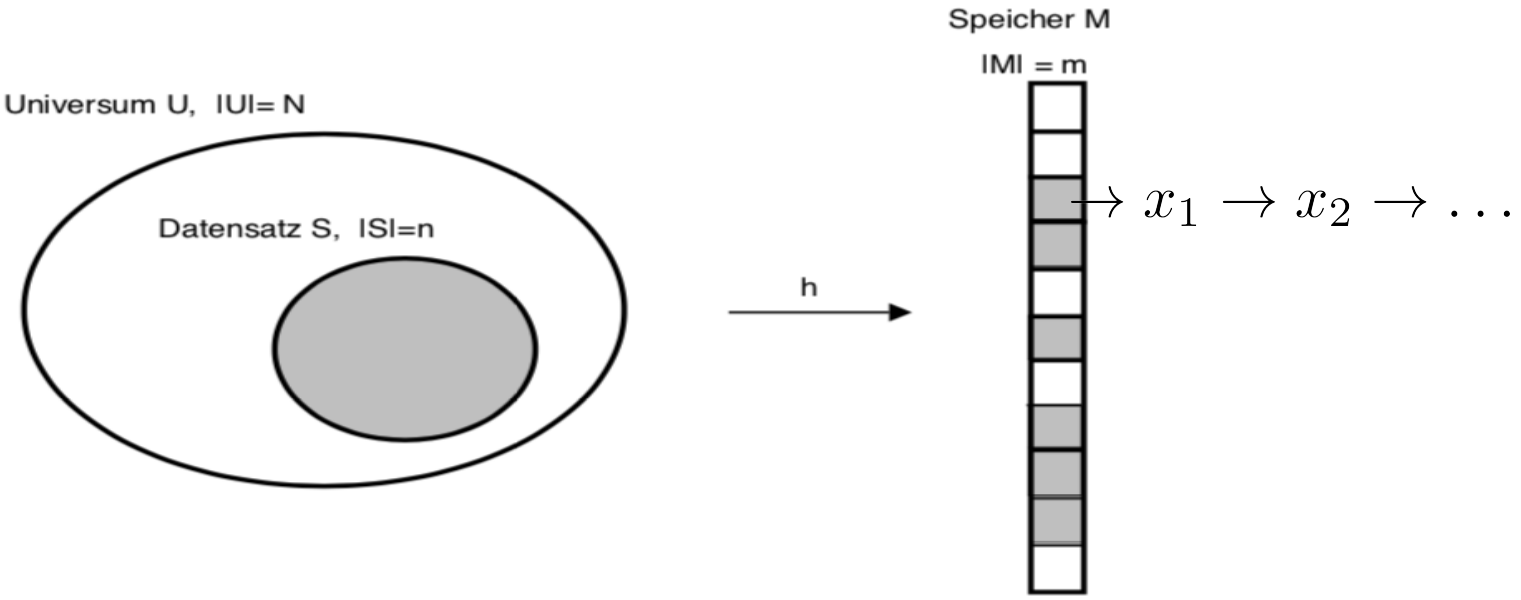
Lösung: Erzeuge Array der Länge 10^{13}. Speichere jedes Buch in der Zelle mit entsprechender Nummer.

Problem: zu viel Speicher

Ziel: Speicher nicht viel grösser als die Zahl der Einträge.

Hashing

Idee: Wir benutzen eine Funktion $h : U \to \{1, \ldots, m\}$.



Wir wollen die Zahl der Kollisionen minimieren, d.h. für alle $x \in S$ soll $\{y \in S \mid h(y) = h(x)\}$ klein sein.

Idee: Wähle h zufällig!

Universelles Hashing

Eine Menge $\mathcal{H}$ von Hashfunktionen $\mathcal{U} \to \{1, \ldots, m\}$ heisst *universell*, falls:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m} \quad \forall x, y \in \mathcal{U}, x \neq y$$

Für jede universelle Menge $\mathcal{H}$ von Hashfunktionen, jeden Datensatz $S \subseteq \mathcal{U}$ mit $|S| \leq m$, und jedes $x \in S$ gilt:

Die erwartete Zahl von Kollisionen mit x ist höchstens 1.

Kapitel 4:

Datenstrukturen

Kapitel 4.4:

Vorrangwarteschlangen

Algorithmus von Dijkstra

Algorithmus 2.5 Algorithmus von Dijkstra

Eingabe: Ein zusammenhängendes Netzwerk $N = (V, E, \ell)$ mit $\ell \geq 0$ und Knoten $s, t \in V$
Ausgabe: Ein kürzester s–t-Pfad in N

{ *Initialisierung* }
$W := \emptyset$; $\rho[s] := 0$; pred$[s] := nil$;
for all $v \in V \setminus \{s\}$ **do** $\rho[v] := \infty$; Insert(v,∞)

{ *Traversieren und Markieren* }
while $t \notin W$ **do**
{ *Traversieren* }
Finde ein $x_0 \in V \setminus W$ mit $\rho[x_0] = \min\{\rho[v] \mid v \in V \setminus W\}$; x_0 = ExtractMin
$W := W \cup \{x_0\}$;
{ *Markieren* }
for all $v \in \Gamma(x_0) \cap (V \setminus W)$ mit $\rho[v] > \rho[x_0] + \ell(x_0, v)$ **do**
$\rho[v] := \rho[x_0] + \ell(x_0, v)$; pred$[v] := x_0$; DecreaseKey(v, $\rho[x_0] + l(x_0,v)$)

Laufzeit:

O(n · Laufzeit von Insert + n · Laufzeit von ExtractMin +
m · Laufzeit von DecreaseKey)

Vorrangwarteschlangen - Realisierungen

	Feld oder Liste (sortiert)	Feld oder Liste (unsortiert)	Suchbaum

`Insert`
`Extract-Min`
`Decrease-Key`			

Fibonacci-Heaps:

Insert: O(1) *
DecreaseKey: O(1) *
ExtractMin: O(log n) *

*amortisierte Kosten

Fibonacci-Heaps: Eigenschaften

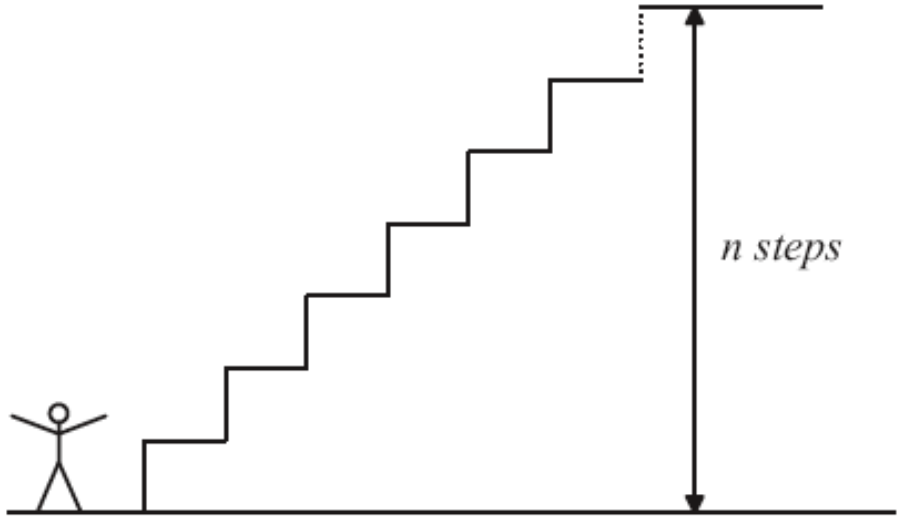
Satz:
Beginnend mit einem leeren Heap können k Operationen `Insert`, `Decrease-Key` und `Extract-Min` in Zeit $\mathcal{O}(k + \ell \cdot \log n)$ ausgeführt werden, wobei ℓ die Anzahl von `Extract-Min` Operationen ist und n die maximale Anzahl von Elementen bezeichnet, die zu irgend einem Zeitpunkt im Heap enthalten sind.

Für Algorithmus von Dijkstra/Prim heisst das:

- n Insert,
- n Extract-Min und
- m Decrease-Key

können in Zeit O(m+ n log n) ausgeführt werden

Amortisierte Analyse - Beispiel



Annahmen:
- pro Stufe eine Sekunde, unabh. von Richtung
- wir beginnen unten

Zwei Arten von Operationen:

Rauf: Steige eine Stufe hinauf, falls noch nicht ganz oben.

Runter: Gehe ganz nach unten

Frage:

Wie lange dauert eine Operation höchstens?

Wie lange dauert eine Folge von k Operationen höchstens?