SS 2005

Einführung in die Informatik IV

Ernst W. Mayr

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2005SS/info4/index.html.de

8. Juli 2005

4.3 Vorrangwarteschlangen (priority queues)

Definition 203

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. *Insert*
2. *ExtractMin* Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. *DecreaseKey* Verkleinern eines Schlüssels
4. *Union* Vereinigung zweier Priority Queues

4.3 Vorrangwarteschlangen (priority queues)

Definition 203

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. *Insert*
2. *ExtractMin* Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. *DecreaseKey* Verkleinern eines Schlüssels
4. *Union* Vereinigung zweier Priority Queues

4.3 Vorrangwarteschlangen (priority queues)

Definition 203

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. *Insert*
2. *ExtractMin* Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. *DecreaseKey* Verkleinern eines Schlüssels
4. *Union* Vereinigung zweier Priority Queues

4.3 Vorrangwarteschlangen (priority queues)

Definition 203

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. *Insert*
2. *ExtractMin* Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. *DecreaseKey* Verkleinern eines Schlüssels
4. *Union* Vereinigung zweier Priority Queues

4.3 Vorrangwarteschlangen (priority queues)

Definition 203

Eine Vorrangwarteschlange (priority queue) ist eine Datenstruktur, die die folgenden Operationen effizient unterstützt:

1. *Insert*
2. *ExtractMin* Extrahieren und Löschen des Elements mit dem kleinsten Schlüssel
3. *DecreaseKey* Verkleinern eines Schlüssels
4. *Union* Vereinigung zweier Priority Queues

Wir besprechen die Implementierung einer Vorrangwarteschlange als Binomial Heap. Diese Implementierung ist (relativ) einfach, aber asymptotisch bei weitem nicht so gut wie modernere Datenstrukturen für Priority Queues, z.B. Fibonacci Heaps (die in der weiterführenden Vorlesung EA1 besprochen werden).

Hier ist ein kurzer Vergleich der Zeitkomplexitäten:

	BinHeap	FibHeap
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$
Union	$O(\log n)$	$O(1)$
	worst case	amortisiert

Wir besprechen die Implementierung einer Vorrangwarteschlange als Binomial Heap. Diese Implementierung ist (relativ) einfach, aber asymptotisch bei weitem nicht so gut wie modernere Datenstrukturen für Priority Queues, z.B. Fibonacci Heaps (die in der weiterführenden Vorlesung EA1 besprochen werden).

Hier ist ein kurzer Vergleich der Zeitkomplexitäten:

	BinHeap	FibHeap
Insert	$O(\log n)$	$O(1)$
ExtractMin	$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(1)$
Union	$O(\log n)$	$O(1)$
	worst case	amortisiert

Definition 204

- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

B_0 B_1

Definition 204

- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

B_0 B_1

Definition 204

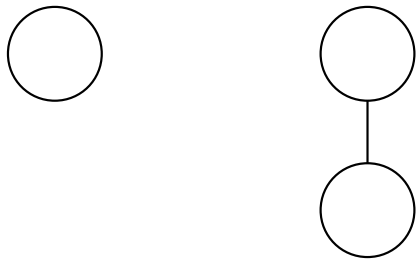
- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

B_0 B_1

Definition 204

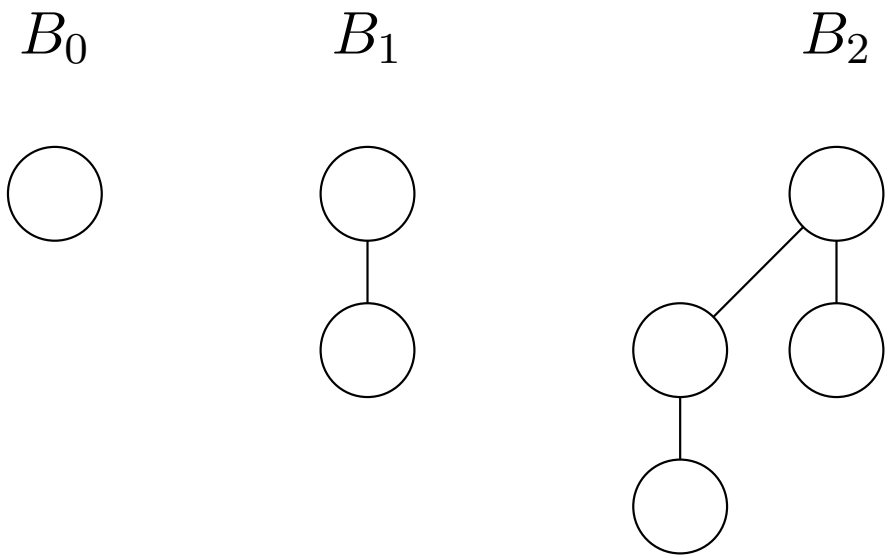
- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

B_0 B_1



Definition 204

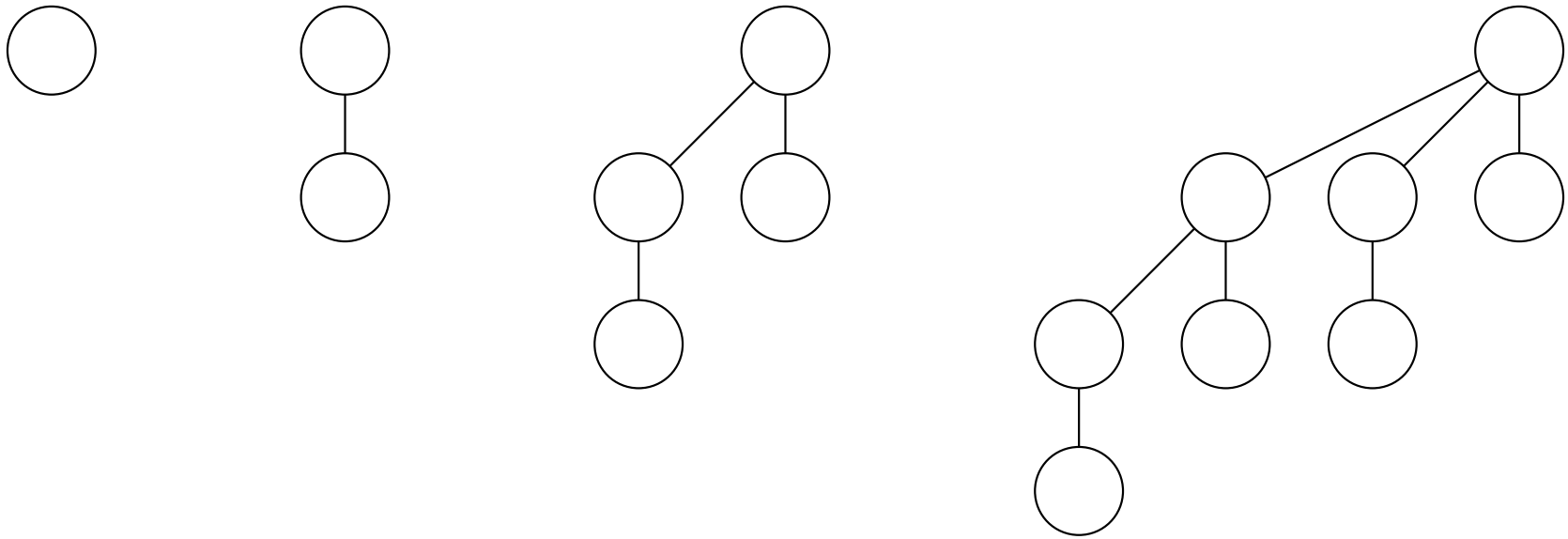
- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.



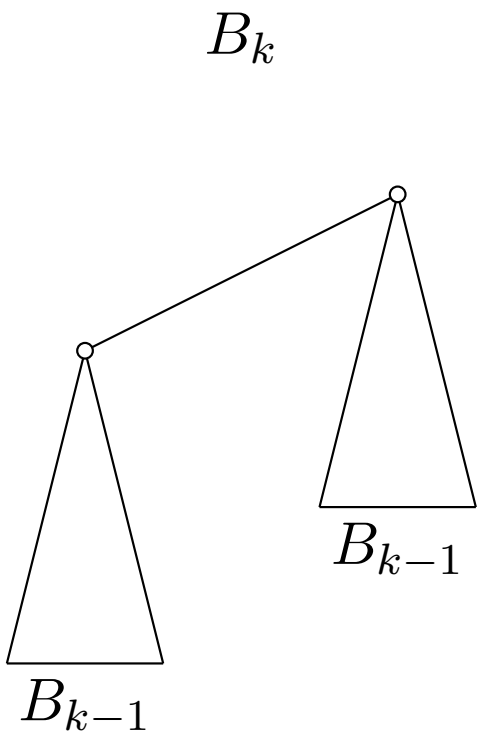
Definition 204

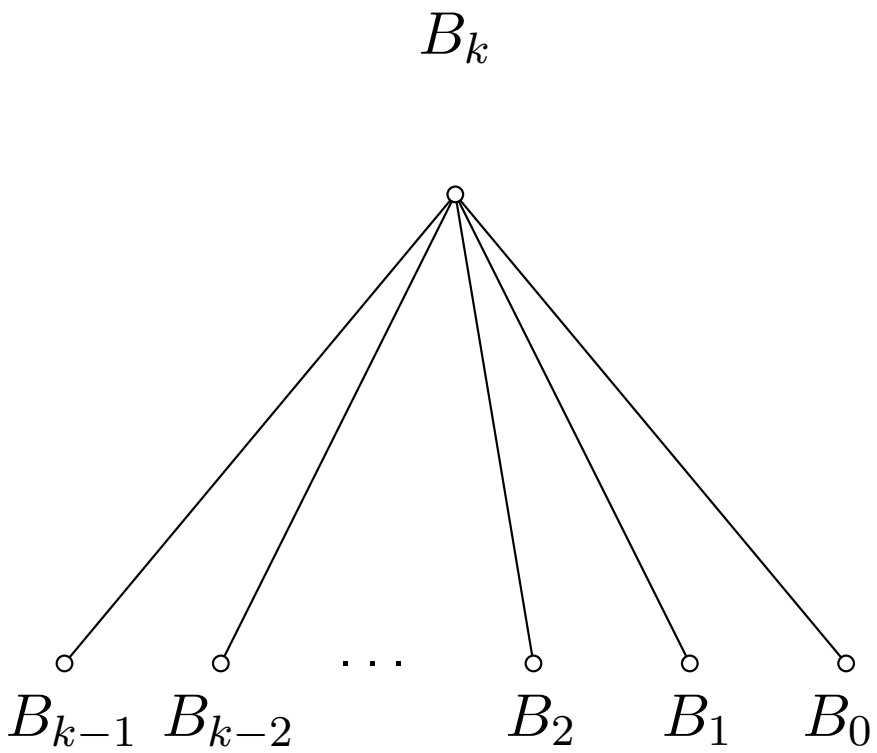
- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

B_0 B_1 B_2 B_3



Rekursiver Aufbau des Binomialbaums B_k





Satz 205

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Satz 205

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Satz 205

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Satz 205

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Satz 205

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Beweis:

Der Beweis ergibt sich sofort aus dem rekursiven Aufbau des B_k und der Rekursionsformel

$$\binom{n}{i} = \binom{n-1}{i} + \binom{n-1}{i-1}$$

für Binomialkoeffizienten. □

Definition 206

Ein Binomial Heap ist eine *Menge* $\mathcal{H}$ von Binomialbäumen, wobei jedem Knoten v ein Schlüssel $key(v)$ zugeordnet ist, so dass folgende Eigenschaften gelten:

1. jeder Binomialbaum $\in \mathcal{H}$ erfüllt die Heap-Bedingung und ist ein min-Heap:

$$(\forall \text{ Knoten } v, w)[v \text{ Vater von } w \Rightarrow key(v) \leq key(w)]$$

2. $\mathcal{H}$ enthält für jedes $k \in \mathbb{N}_0$ höchstens einen B_k

Definition 206

Ein Binomial Heap ist eine *Menge* $\mathcal{H}$ von Binomialbäumen, wobei jedem Knoten v ein Schlüssel $key(v)$ zugeordnet ist, so dass folgende Eigenschaften gelten:

1. jeder Binomialbaum $\in \mathcal{H}$ erfüllt die Heap-Bedingung und ist ein min-Heap:

$$(\forall \text{ Knoten } v, w)[v \text{ Vater von } w \Rightarrow key(v) \leq key(w)]$$

2. $\mathcal{H}$ enthält für jedes $k \in \mathbb{N}_0$ höchstens einen B_k

Definition 206

Ein Binomial Heap ist eine *Menge* $\mathcal{H}$ von Binomialbäumen, wobei jedem Knoten v ein Schlüssel $key(v)$ zugeordnet ist, so dass folgende Eigenschaften gelten:

1. jeder Binomialbaum $\in \mathcal{H}$ erfüllt die Heap-Bedingung und ist ein min-Heap:

$$(\forall \text{ Knoten } v, w)[v \text{ Vater von } w \Rightarrow key(v) \leq key(w)]$$

2. $\mathcal{H}$ enthält für jedes $k \in \mathbb{N}_0$ höchstens einen B_k

Für jeden Binomial Heap $\mathcal{H}$ gilt also

1. Enthält $\mathcal{H}$ n Schlüssel, $n \in \mathbb{N}$, so besteht $\mathcal{H}$ höchstens aus $\max\{1, \lceil \mathrm{ld}\, n \rceil\}$ Binomialbäumen.
2. In jedem von $\mathcal{H}$'s Binomialbäumen ist ein kleinster Schlüssel an der Wurzel gespeichert; verlinkt man daher die Wurzeln aller Binomialbäume von $\mathcal{H}$ in einer zirkulären Liste, kann ein minimaler Schlüssel in $\mathcal{H}$ durch einfaches Durchlaufen dieser Liste gefunden werden.

Für jeden Binomial Heap $\mathcal{H}$ gilt also

1. Enthält $\mathcal{H}$ n Schlüssel, $n \in \mathbb{N}$, so besteht $\mathcal{H}$ höchstens aus $\max\{1, \lceil \mathrm{ld}\, n \rceil\}$ Binomialbäumen.
2. In jedem von $\mathcal{H}$'s Binomialbäumen ist ein kleinster Schlüssel an der Wurzel gespeichert; verlinkt man daher die Wurzeln aller Binomialbäume von $\mathcal{H}$ in einer zirkulären Liste, kann ein minimaler Schlüssel in $\mathcal{H}$ durch einfaches Durchlaufen dieser Liste gefunden werden.

Für jeden Binomial Heap $\mathcal{H}$ gilt also

1. Enthält $\mathcal{H}$ n Schlüssel, $n \in \mathbb{N}$, so besteht $\mathcal{H}$ höchstens aus $\max\{1, \lceil \mathrm{ld}\, n \rceil\}$ Binomialbäumen.
2. In jedem von $\mathcal{H}$'s Binomialbäumen ist ein kleinster Schlüssel an der Wurzel gespeichert; verlinkt man daher die Wurzeln aller Binomialbäume von $\mathcal{H}$ in einer zirkulären Liste, kann ein minimaler Schlüssel in $\mathcal{H}$ durch einfaches Durchlaufen dieser Liste gefunden werden.

Für jeden Binomial Heap $\mathcal{H}$ gilt also

1. Enthält $\mathcal{H}$ n Schlüssel, $n \in \mathbb{N}$, so besteht $\mathcal{H}$ höchstens aus $\max\{1, \lceil \mathrm{ld}\, n \rceil\}$ Binomialbäumen.
2. In jedem von $\mathcal{H}$'s Binomialbäumen ist ein kleinster Schlüssel an der Wurzel gespeichert; verlinkt man daher die Wurzeln aller Binomialbäume von $\mathcal{H}$ in einer zirkulären Liste, kann ein minimaler Schlüssel in $\mathcal{H}$ durch einfaches Durchlaufen dieser Liste gefunden werden.

Korollar 207

In einem Binomial Heap mit n Schlüsseln benötigt FindMin Zeit $O(\log n)$.

Wir betrachten nun die Realisierung der *Union*-Operation. Hierbei wird vorausgesetzt, dass die Objektmengen, zu denen die Schlüssel in den beiden zu verschmelzenden Binomial Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$ gehören, disjunkt sind. Es ist jedoch durchaus möglich, dass der gleiche Schlüssel für mehrere Objekte vorkommt.

1. Fall $\mathcal{H}_1$ und $\mathcal{H}_2$ enthalten jeweils nur einen Binomialbaum B_k (mit dem gleichen Index k):

In diesem Fall fügen wir den B_k, dessen Wurzel den größeren Schlüssel hat, als neuen Unterbaum der Wurzel des anderen B_k ein, gemäß der rekursiven Struktur der Binomialbäume. Es entsteht ein B_{k+1}, für den per Konstruktion weiterhin die Heap-Bedingung erfüllt ist.

Wir betrachten nun die Realisierung der *Union*-Operation. Hierbei wird vorausgesetzt, dass die Objektmengen, zu denen die Schlüssel in den beiden zu verschmelzenden Binomial Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$ gehören, disjunkt sind. Es ist jedoch durchaus möglich, dass der gleiche Schlüssel für mehrere Objekte vorkommt.

1. Fall $\mathcal{H}_1$ und $\mathcal{H}_2$ enthalten jeweils nur einen Binomialbaum B_k (mit dem gleichen Index k):

In diesem Fall fügen wir den B_k, dessen Wurzel den größeren Schlüssel hat, als neuen Unterbaum der Wurzel des anderen B_k ein, gemäß der rekursiven Struktur der Binomialbäume. Es entsteht ein B_{k+1}, für den per Konstruktion weiterhin die Heap-Bedingung erfüllt ist.

Wir betrachten nun die Realisierung der *Union*-Operation. Hierbei wird vorausgesetzt, dass die Objektmengen, zu denen die Schlüssel in den beiden zu verschmelzenden Binomial Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$ gehören, disjunkt sind. Es ist jedoch durchaus möglich, dass der gleiche Schlüssel für mehrere Objekte vorkommt.

1. Fall $\mathcal{H}_1$ und $\mathcal{H}_2$ enthalten jeweils nur einen Binomialbaum B_k (mit dem gleichen Index k):

In diesem Fall fügen wir den B_k, dessen Wurzel den größeren Schlüssel hat, als neuen Unterbaum der Wurzel des anderen B_k ein, gemäß der rekursiven Struktur der Binomialbäume. Es entsteht ein B_{k+1}, für den per Konstruktion weiterhin die Heap-Bedingung erfüllt ist.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls *kein* Binomialbaum mit Index k vorhanden ist: **noop**
2. falls *genau ein* Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls *genau zwei* Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls *genau drei* Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen, die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls *kein* Binomialbaum mit Index k vorhanden ist: **noop**
2. falls *genau ein* Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls *genau zwei* Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls *genau drei* Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls *kein* Binomialbaum mit Index k vorhanden ist: **noop**
2. falls *genau ein* Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls *genau zwei* Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls *genau drei* Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls *kein* Binomialbaum mit Index k vorhanden ist: **noop**
2. falls *genau ein* Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls *genau zwei* Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls *genau drei* Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls *kein* Binomialbaum mit Index k vorhanden ist: **noop**
2. falls *genau ein* Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls *genau zwei* Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls *genau drei* Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls *kein* Binomialbaum mit Index k vorhanden ist: **noop**
2. falls *genau ein* Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls *genau zwei* Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls *genau drei* Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

Man beachte, dass nie mehr als 3 Binomialbäume mit gleichem Index auftreten können!

Bemerkung:
Es besteht eine einfache Analogie zur Addition zweier Binärzahlen, wobei das Verschmelzen zweier gleich großer Binomialbäume dem Auftreten eines Übertrags entspricht!

Lemma 208
Die Union-Operation zweier Binomial Heaps mit zusammen n Schlüsseln kann in Zeit $O(\log n)$ durchgeführt werden.

Man beachte, dass nie mehr als 3 Binomialbäume mit gleichem Index auftreten können!

Bemerkung:
Es besteht eine einfache Analogie zur Addition zweier Binärzahlen, wobei das Verschmelzen zweier gleich großer Binomialbäume dem Auftreten eines Übertrags entspricht!

Lemma 208
Die Union-Operation zweier Binomial Heaps mit zusammen n Schlüsseln kann in Zeit $O(\log n)$ durchgeführt werden.

Man beachte, dass nie mehr als 3 Binomialbäume mit gleichem Index auftreten können!

Bemerkung:
Es besteht eine einfache Analogie zur Addition zweier Binärzahlen, wobei das Verschmelzen zweier gleich großer Binomialbäume dem Auftreten eines Übertrags entspricht!

Lemma 208
Die Union-Operation zweier Binomial Heaps mit zusammen n Schlüsseln kann in Zeit $O(\log n)$ durchgeführt werden.

Die Operation *Insert*$(\mathcal{H}, k)$:

1. erzeuge einen neuen Binomial Heap $\mathcal{H}' = \{B_0\}$ mit k als einzigem Schlüssel
2. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *Insert*$(\mathcal{H}, k)$:

1. erzeuge einen neuen Binomial Heap $\mathcal{H}' = \{B_0\}$ mit k als einzigem Schlüssel
2. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *Insert*$(\mathcal{H}, k)$:

1. erzeuge einen neuen Binomial Heap $\mathcal{H}' = \{B_0\}$ mit k als einzigem Schlüssel
2. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *Insert*$(\mathcal{H}, k)$:

1. erzeuge einen neuen Binomial Heap $\mathcal{H}' = \{B_0\}$ mit k als einzigem Schlüssel
2. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der *Insert*-Operation offensichtlich $O(\log n)$.

Die Operation *ExtractMin*($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen dieser Wurzel
4. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *ExtractMin*($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen dieser Wurzel
4. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *ExtractMin*($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen dieser Wurzel
4. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *ExtractMin*($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen dieser Wurzel
4. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *ExtractMin*($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen dieser Wurzel
4. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Die Operation *ExtractMin*($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen dieser Wurzel
4. $\mathcal{H} := \mathit{Union}(\mathcal{H}, \mathcal{H}')$

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der *ExtractMin*-Operation offensichtlich $O(\log n)$.

Die Operation *DecreaseKey*$(\mathcal{H}, v, k)$ (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Die Operation *DecreaseKey*$(\mathcal{H}, v, k)$ (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Die Operation *DecreaseKey*$(\mathcal{H}, v, k)$ (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Die Operation *DecreaseKey*$(\mathcal{H}, v, k)$ (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Die Operation *DecreaseKey*$(\mathcal{H}, v, k)$ (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der *DecreaseKey*-Operation offensichtlich $O(\log n)$.

5. Mengendarstellungen — Union-Find-Strukturen

Gegeben ist eine (endliche) Menge S von Objekten, die in Klassen X_i partitioniert ist:

$$S = X_1 \uplus X_2 \uplus \ldots \uplus X_l$$

Für jede Klasse X_i gibt es dabei einen Repräsentanten $r_i \in X_i$.

Es wird eine Datenstruktur gesucht, die die folgenden Operationen effizient unterstützt:

1. *Init*(S): jedes Element $\in S$ bildet eine eigene Klasse mit sich selbst als Repräsentant
2. *Union*(r, s): vereinige die beiden Klassen mit r bzw. s als Repräsentanten, wähle r als Repräsentant der neuen Klasse (die beiden alten Klassen verschwinden)
3. *Find*(x): bestimme zu $x \in S$ den Repräsentanten der eindeutigen aktuellen Klasse, die x enthält

5. Mengendarstellungen — Union-Find-Strukturen

Gegeben ist eine (endliche) Menge S von Objekten, die in Klassen X_i partitioniert ist:

$$S = X_1 \uplus X_2 \uplus \ldots \uplus X_l$$

Für jede Klasse X_i gibt es dabei einen Repräsentanten $r_i \in X_i$.

Es wird eine Datenstruktur gesucht, die die folgenden Operationen effizient unterstützt:

1. *Init*(S): jedes Element $\in S$ bildet eine eigene Klasse mit sich selbst als Repräsentant
2. *Union*(r, s): vereinige die beiden Klassen mit r bzw. s als Repräsentanten, wähle r als Repräsentant der neuen Klasse (die beiden alten Klassen verschwinden)
3. *Find*(x): bestimme zu $x \in S$ den Repräsentanten der eindeutigen aktuellen Klasse, die x enthält

5. Mengendarstellungen — Union-Find-Strukturen

Gegeben ist eine (endliche) Menge S von Objekten, die in Klassen X_i partitioniert ist:

$$S = X_1 \uplus X_2 \uplus \ldots \uplus X_l$$

Für jede Klasse X_i gibt es dabei einen Repräsentanten $r_i \in X_i$.

Es wird eine Datenstruktur gesucht, die die folgenden Operationen effizient unterstützt:

1. *Init*(S): jedes Element $\in S$ bildet eine eigene Klasse mit sich selbst als Repräsentant
2. *Union*(r, s): vereinige die beiden Klassen mit r bzw. s als Repräsentanten, wähle r als Repräsentant der neuen Klasse (die beiden alten Klassen verschwinden)
3. *Find*(x): bestimme zu $x \in S$ den Repräsentanten der eindeutigen aktuellen Klasse, die x enthält

5. Mengendarstellungen — Union-Find-Strukturen

Gegeben ist eine (endliche) Menge S von Objekten, die in Klassen X_i partitioniert ist:

$$S = X_1 \uplus X_2 \uplus \ldots \uplus X_l$$

Für jede Klasse X_i gibt es dabei einen Repräsentanten $r_i \in X_i$.

Es wird eine Datenstruktur gesucht, die die folgenden Operationen effizient unterstützt:

1. *Init*(S): jedes Element $\in S$ bildet eine eigene Klasse mit sich selbst als Repräsentant
2. *Union*(r, s): vereinige die beiden Klassen mit r bzw. s als Repräsentanten, wähle r als Repräsentant der neuen Klasse (die beiden alten Klassen verschwinden)
3. *Find*(x): bestimme zu $x \in S$ den Repräsentanten der eindeutigen aktuellen Klasse, die x enthält

5. Mengendarstellungen — Union-Find-Strukturen

Gegeben ist eine (endliche) Menge S von Objekten, die in Klassen X_i partitioniert ist:

$$S = X_1 \uplus X_2 \uplus \ldots \uplus X_l$$

Für jede Klasse X_i gibt es dabei einen Repräsentanten $r_i \in X_i$.

Es wird eine Datenstruktur gesucht, die die folgenden Operationen effizient unterstützt:

1. *Init*(S): jedes Element $\in S$ bildet eine eigene Klasse mit sich selbst als Repräsentant
2. *Union*(r, s): vereinige die beiden Klassen mit r bzw. s als Repräsentanten, wähle r als Repräsentant der neuen Klasse (die beiden alten Klassen verschwinden)
3. *Find*(x): bestimme zu $x \in S$ den Repräsentanten der eindeutigen aktuellen Klasse, die x enthält

Eine entsprechende Datenstruktur nennt man auch Union-Find-Struktur.

Eine triviale Implementierung, in der zu jedem Element aus S sein Repräsentant (bzw. ein Verweis darauf) gespeichert wird, benötigt im worst case Laufzeit $\Omega(|S|)$ für die *Union*-Operation.

Um logarithmische Laufzeit zu erreichen, realisieren wir die Datenstruktur als Vereinigung von zur Wurzel hin gerichteten Bäumen. Dabei bilden die Repräsentanten die Wurzeln und die restlichen Elemente der Klasse die restlichen Knoten.

Eine entsprechende Datenstruktur nennt man auch Union-Find-Struktur.

Eine triviale Implementierung, in der zu jedem Element aus S sein Repräsentant (bzw. ein Verweis darauf) gespeichert wird, benötigt im worst case Laufzeit $\Omega(|S|)$ für die *Union*-Operation.

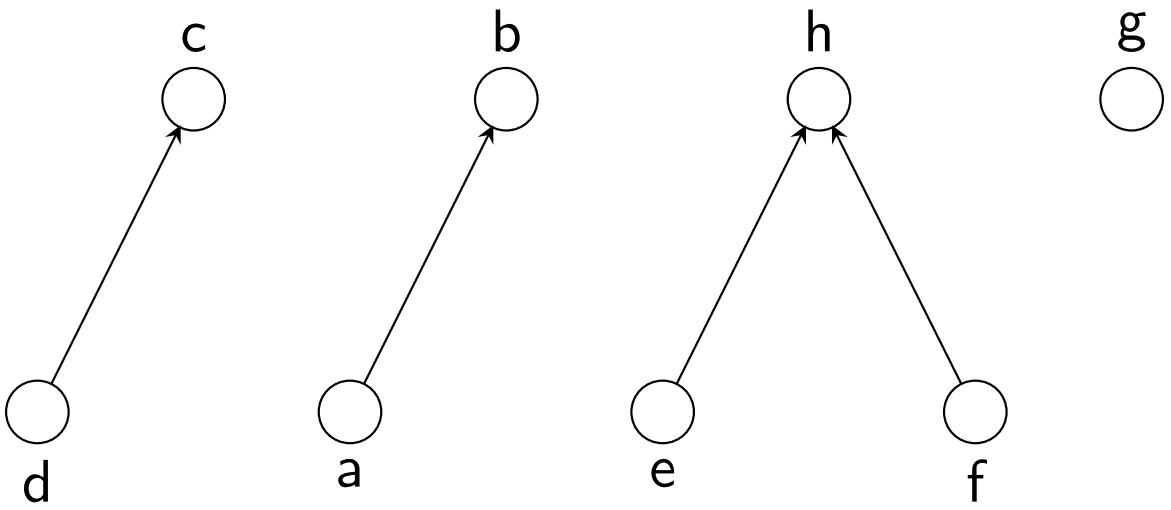
Um logarithmische Laufzeit zu erreichen, realisieren wir die Datenstruktur als Vereinigung von zur Wurzel hin gerichteten Bäumen. Dabei bilden die Repräsentanten die Wurzeln und die restlichen Elemente der Klasse die restlichen Knoten.

Eine entsprechende Datenstruktur nennt man auch Union-Find-Struktur.

Eine triviale Implementierung, in der zu jedem Element aus S sein Repräsentant (bzw. ein Verweis darauf) gespeichert wird, benötigt im worst case Laufzeit $\Omega(|S|)$ für die *Union*-Operation.

Um logarithmische Laufzeit zu erreichen, realisieren wir die Datenstruktur als Vereinigung von zur Wurzel hin gerichteten Bäumen. Dabei bilden die Repräsentanten die Wurzeln und die restlichen Elemente der Klasse die restlichen Knoten.

Beispiel 209



Die Operationen *Find* und *Union* werden wie folgt implementiert:

Find(x) verfolge von x aus den Pfad zur Wurzel und gib diese aus; die worst-case Laufzeit ist proportional zur maximalen Höhe eines Baums in der Union-Find-Struktur.

Union(r, s) hänge die Wurzel des niedrigeren Baums als neues Kind der Wurzel des anderen Baums ein; vertausche notfalls r und s, damit r an der Wurzel des neuen Baums zu liegen kommt.

Die Operationen *Find* und *Union* werden wie folgt implementiert:

Find(x) verfolge von x aus den Pfad zur Wurzel und gib diese aus; die worst-case Laufzeit ist proportional zur maximalen Höhe eines Baums in der Union-Find-Struktur.

Union(r, s) hänge die Wurzel des niedrigeren Baums als neues Kind der Wurzel des anderen Baums ein; vertausche notfalls r und s, damit r an der Wurzel des neuen Baums zu liegen kommt.

Die Operationen *Find* und *Union* werden wie folgt implementiert:

Find(x) verfolge von x aus den Pfad zur Wurzel und gib diese aus; die worst-case Laufzeit ist proportional zur maximalen Höhe eines Baums in der Union-Find-Struktur.

Union(r, s) hänge die Wurzel des niedrigeren Baums als neues Kind der Wurzel des anderen Baums ein; vertausche notfalls r und s, damit r an der Wurzel des neuen Baums zu liegen kommt.

Die Operationen *Find* und *Union* werden wie folgt implementiert:

Find(x) verfolge von x aus den Pfad zur Wurzel und gib diese aus; die worst-case Laufzeit ist proportional zur maximalen Höhe eines Baums in der Union-Find-Struktur.

Union(r, s) hänge die Wurzel des niedrigeren Baums als neues Kind der Wurzel des anderen Baums ein; vertausche notfalls r und s, damit r an der Wurzel des neuen Baums zu liegen kommt.

Wir nennen die bei der *Union*-Operation verfolgte Strategie gewichtete Vereinigung. Um sie effizient implementieren zu können, wird für jeden Baum der Union-Find-Struktur seine Höhe mitgeführt.

Lemma 210

Sei T ein Baum einer Union-Find-Struktur mit gewichteter Vereinigung. Dann gilt für seine Höhe $h(T)$

$$h(T) \leq \mathrm{ld}(|T|)\,,$$

wobei $|T|$ die Anzahl der Knoten in T bezeichnet.

Beweis:

Lemma 210

Sei T ein Baum einer Union-Find-Struktur mit gewichteter Vereinigung. Dann gilt für seine Höhe $h(T)$

$$h(T) \leq \mathrm{ld}(|T|)\,,$$

wobei $|T|$ die Anzahl der Knoten in T bezeichnet.

Beweis:

Beweis durch Induktion über die Höhe.

Induktionsanfang: Für Bäume der Höhe 0 oder 1 stimmt die Behauptung.

Lemma 210

Sei T ein Baum einer Union-Find-Struktur mit gewichteter Vereinigung. Dann gilt für seine Höhe $h(T)$

$$h(T) \leq \mathrm{ld}(|T|)\,,$$

wobei $|T|$ die Anzahl der Knoten in T bezeichnet.

Beweis:

Induktionsschluss: Sei o.B.d.A. $h(T_r) \geq h(T_s)$. Ist $h(T_r) > h(T_s)$, so ist die neue Höhe $= h(T_r)$ und die Behauptung erfüllt.

Ist $h(T_r) = h(T_s)$, so ist die neue Höhe $= h(T_r) + 1$, und es gilt:

$$\begin{aligned}
\mathrm{ld}(|T_r| + |T_s|) &\geq \mathrm{ld}(2\min\{|T_r|, |T_s|\}) \\
&= 1 + \mathrm{ld}(\min\{|T_r|, |T_s|\}) \\
&\geq 1 + h(T_s) \qquad\qquad \text{gemäß I.A.}
\end{aligned}$$

□