

Teil 4: Verwaltung von Mengen  
Prioritäts-Warteschlangen

# Fibonacci Heaps

# Fibonacci Heaps

**Binary Heaps** und **Binomial Queues** haben die Eigenschaft, dass ihre Struktur durch die Anzahl der in ihnen enthaltenen Schlüsseln eindeutig bestimmt ist.

**Fibonacci Heaps (F-Heaps)** überwinden diese Beschränkung. Sie sind lediglich als ein **Wald heapgeordneter Bäume mit disjunkten Schlüsselmenge**n definiert. Ihre Struktur ergibt sich implizit über die Operationen, die auf ihnen definiert sind:

- Aufbau eines F-Heaps
- `min`
- `delete_min`
- `decrease_key`
- `delete`
- `union`

## Fibonacci Heaps - Marken und Ränge

Ein F-Heap wird als Liste von (heapgeordneten) Bäumen implementiert. Dabei wird jeder Baum durch seine Wurzel repräsentiert.

Zusätzlich halten wir einen Zeiger auf die Wurzel mit dem kleinsten Schlüssel, dem so genannten **Minimalknoten** des F-Heaps.

Außerdem benötigen wir für jeden Schlüssel im F-Heap eine **Marke**, die wir über  $\text{mark}(x)$  abfragen und setzen können. Mögliche Werte sind **0 (nicht gesetzt)** und **1 (gesetzt)**.

### Definition (Rang)

Sei  $\mathcal{F}$  ein F-Heap,  $x \in \mathcal{F}$  ein Schlüssel in  $\mathcal{F}$  und  $B \in \mathcal{F}$  ein Baum in  $\mathcal{F}$ . Dann definieren wir die **Ränge von  $x$  und  $B$**  als

- $\text{rank}(x) := \text{Anzahl der Kinder von } x \text{ in seinem Baum}$
- $\text{rank}(B) := \text{rank}(\text{root}(x))$

Wir gehen davon aus, dass jeder Knoten seinen Rang kennt.

## Fibonacci Heaps - Initialisierung, union und insert

Die **Initialisierung** eines F-Heaps erzeugt einen leeren F-Heap **nil** ohne Markierungen. Dies verursacht  $O(1)$  Kosten.

Da wir keine besonderen Anforderungen an die Struktur von F-Heaps gestellt haben, gestalten sich **union** und **insert** als besonders einfach.

Um zwei F-Heaps  $\mathcal{F}_1$  und  $\mathcal{F}_2$  mittels **union**( $\mathcal{F}_1, \mathcal{F}_2$ ) zu vereinigen, werden einfach ihre Baumlisten aneinander gehängt. Dies verursacht  $O(1)$  Kosten.

Um ein neues Element  $x$  mittels **insert**( $x$ ) einzufügen, wird ein Baum mit genau einem Schlüssel,  $x$ , erzeugt und an die Liste des F-Heaps gehängt. Auch hier ergeben sich Kosten von  $O(1)$ .

## Fibonacci Heaps - delete\_min

$\mathcal{F}$  sei ein F-Heap mit  $T$  Bäumen und  $n$  Schlüsseln.

$r_{\max}(n)$  sei der maximale Rang, der in einem F-Heap mit  $n$  Elementen auftreten kann.

Offensichtlich gilt  $r_{\max}(n) \leq n - 1$ . Allerdings werden wir später noch sehen, dass  $r_{\max}(n) \in O(\log n)$ .

delete\_min löscht den Minimalknoten und hängt die so entstehenden Bäume an den F-Heap an. Anschließend wird der F-Heap konsolidiert, d.h. man sorgt durch Vereinigung von Bäumen dafür, dass am Ende maximal ein Baum eines Ranges  $r \in \{0, \dots, r_{\max}(n - 1)\}$  vorliegt.

# Fibonacci Heaps - delete\_min

## Prozedur (delete\_min)

**Eingabe:** Ein  $F$ -Heap  $\mathcal{F}$  mit  $n$  Schlüsseln und  $T$  Bäumen

$x$  sei der Minimalknoten und  $r := \text{rank}(x) \leq r_{\max}(n)$ ; //  $O(1)$

Trenne  $x$  ab und füge die entstehenden  $r$  Teilbäume an die Liste an.

Lösche dabei alle Marken an den neu entstandenen Wurzeln. ; //  $O(T + r)$

Leg ein Feld  $L[0 \dots r_{\max}(n)]$  von Baumlisten an. ; //  $O(r_{\max}(n))$

**für**  $B \in \mathcal{F}$  **tue**  $L[\text{rank}(B)] := (L[\text{rank}(B)], B)$ ; //  $O(T + r)$

**für**  $i = 0 \dots r_{\max}(n) - 1$  **tue**

**solange**  $|L[i]| \geq 2$  **tue**

    Entnehme zwei Bäume aus  $L[i]$  und  $\mathcal{F}$

    Hänge den Baum mit der größeren Wurzel an die Wurzel des anderen Baumes.

    Füge den neu entstandenen Baum an  $L[i + 1]$  und  $\mathcal{F}$  an.

**Ende**

**Ende**

Bestimme den neuen Minimalknoten. ; //  $O(T + r_{\max}(n))$

## Fibonacci Heaps - delete\_min

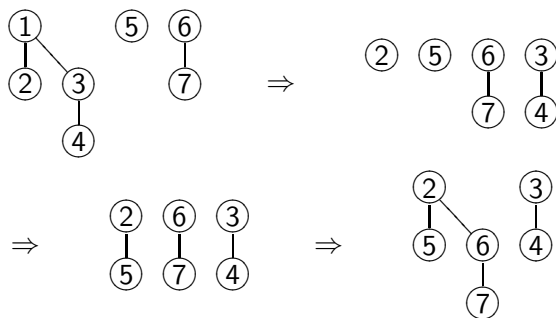
Die Kosten für die Schleife zur Konsolidierung belaufen sich auf  $O(T + r_{\max}(n))$  (Übungsaufgabe), wodurch sich eine Gesamtlaufzeit für delete\_min von  $O(T + r_{\max}(n))$  ergibt.

### Bemerkung

- *Nach der Ausführung von delete\_min gibt es pro möglichen Rang höchstens einen Baum. Insbesondere ist die Zahl der Bäume danach durch  $r_{\max}(n)$  beschränkt.*
- *Hat man bislang nur insert durchgeführt, erhält man nach Ausführen von delete\_min eine **Binomial Queue**.*

## Fibonacci Heaps - delete\_min - Ein Beispiel

Wir führen `delete_min` auf dem folgenden F-Heap aus:



# Fibonacci Heaps - decrease\_key

Um einen Schlüssel zu senken gehen wir folgendermaßen vor:

- 1 Falls  $x$  keine Wurzel ist:
  - 1 Sei  $x = y_0, y_1, \dots, y_l$  der Pfad von  $x$  zur Wurzel.
  - 2  $y_{k+1}, k \geq 0$  sei der erste Knoten (außer  $x$ ), der nicht markiert ist (existiert, da die Wurzel auf jeden Fall unmarkiert ist).
  - 3 Für alle  $j$  mit  $0 \leq j \leq k$ , trenne  $y_j$  von seinem Vorgänger ab und mache ihn zur Wurzel eines neuen Baumes und lösche seine Markierung.
  - 4 Falls  $y_{k+1}$  keine Wurzel ist, markiere ihn.
- 2 Senke den Schlüssel von  $x$ .
- 3 Falls der neue Schlüssel kleiner als das bisherige Minimum ist, merke ihn dir als Minimalknoten.

## Prozedur (decrease\_key)

**Eingabe:** Ein  $F$ -Heap  $\mathcal{F}$ , ein Element  $x$  in  $\mathcal{F}$  und ein Schlüsselwert  $k$

**Ausgabe:** Ein  $F$ -Heap mit auf  $k$  gesenkten Schlüssel  $\text{key}(x)$  (falls möglich)

```
wenn  $k < \text{key}(x)$  dann // Ändert sich der Schlüssel?
  wenn  $x \neq \text{pred}(x)$  dann // Wenn  $x$  keine Wurzel ist ...
    // Durchlaufe den Weg von  $x$  zur Wurzel bis zum ersten
    // unmarkierten Knoten.
    wiederhole
      Mache  $x$  zur Wurzel eines neuen Baumes
       $\text{mark}(x) := 0$  ; // Lösche Marke der neuen Wurzel
       $x := \text{pred}(x)$  ; // Gehe zum Vorgänger
    bis  $\text{mark}(x) = 0$ 
    wenn  $\text{pred}(x) \neq x$  dann  $\text{mark}(x) := 1$ 
  Ende
   $\text{key}(x) := k$  ; // Jetzt ist  $x$  eine Wurzel
  wenn  $\text{key}(x) < \text{key}(\text{Minimalknoten})$  dann  $\text{Minimalknoten} := x$ 
Ende
```

## Fibonacci Heaps - decrease\_key

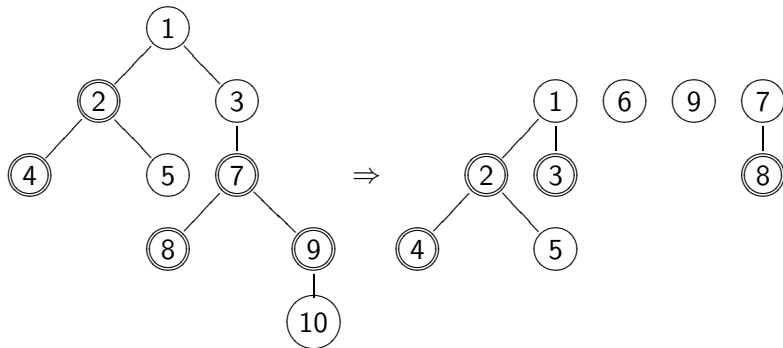
Die Laufzeit von `decrease_key(x, k)` ist  $O(i + 2)$ , wobei  $i$  die Anzahl der umzuhängenden Knoten außer  $x$  ist.

### Beobachtung

- *Nur in `decrease_key` werden Markierungen gesetzt.*
- *Markierungen werden gelöscht, wenn der Knoten zur Wurzel eines Baumes wird.*
- *Insbesondere sind die Wurzeln aller Bäume stets unmarkiert.*
- *Die Zahl der Marken hat sich um mindestens  $k - 1$  mit  $k \geq 0$  verändert. Für  $k = 0$  ist sie unter Umständen um 1 gestiegen.*
- *Die Zahl der Bäume nach der Ausführung von `decrease_key`, ist durch  $T + k + 1$  begrenzt.*

## Fibonacci Heaps - decrease\_key - Ein Beispiel

Wir führen `decrease_key(10, 6)` auf dem folgenden F-Heap aus:



# Fibonacci Heaps - min und delete

**min** ist offensichtlich nur der Zugriff auf den Minimalknoten und verursacht somit  $O(1)$  Kosten.

Ein beliebiges Element wird mittels **delete** auf die folgende Art entfernt:

- 1 `decrease_key(x,  $-\infty$ )`, d.h. mache  $x$  zum Minimalknoten
- 2 `delete_min()`

Damit verursacht `delete` die gleichen Kosten wie `decrease_key` und `delete_min`.

## Lemma

*Sei  $x$  Knoten eines F-Heaps und für  $i \geq 2$  sei  $c_i$  das  $i$ -t älteste Kind von  $x$ . Dann hat  $c_i$  mindestens Rang  $i - 2$ .*

## Beweis

*Seien  $c_1, c_2, \dots, c_{i-1}, c_i$  die nach dem Alter geordneten  $i$  ältesten Kinder von  $x$ .*

*Lediglich während der Konsolidierung in `delete_min` werden neue Kinder an Wurzeln gehängt. D.h. zu dem Zeitpunkt, zu dem  $c_i$  an den Knoten  $x$  gehängt wurde, sind  $c_1, \dots, c_{i-1}$  bereits Kinder von  $x$  und  $x$  ist eine Wurzel.*

*Der Rang von  $x$  war daher mindestens  $r \geq i - 1$  (es können in der Zwischenzeit Kinder entfernt worden sein). Da während der Konsolidierung nur Bäume gleichen Ranges miteinander vereinigt werden, mußte  $c_i$  zu diesem Zeitpunkt denselben Rang  $r \geq i - 1$  gehabt haben.*

## Beweis (Fortsetzung)

*Danach kann der Knoten  $c_i$  durch `decrease_key` ein Kind verloren haben, wobei er aber markiert worden wäre.*

*Hätte er ein weiteres Kind verloren, wäre er abgetrennt und zu einer neuen Wurzel geworden.*

*Damit hat  $c_i$  mindestens den Rang  $i - 2$ .* □

Damit ist auch der Sinn der Marken klar. Die Marken verhindern, dass ein Knoten der keine Wurzel ist, mehr als ein Kind verliert.

Verliert ein innerer Knoten ein zweites Kind, wird er selbst zur Wurzel gemacht.

Dadurch wird verhindert, dass die Bäume zu stark ausdünnen.

# Fiboaci Heaps - Der Ursprung des Namens

## Definition (Fibonacci Zahlen)

Für  $i \geq 0$  sind die **Fibonacci Zahlen** definiert als

$$F_0 = 0, F_1 = 1 \text{ und } F_{i+1} = F_i + F_{i-1}$$

Zum Exkurs **Fibonacci Zahlen**

## Lemma

*Hat ein Knoten  $x$  eines  $F$ -Heaps  $\mathcal{F}$  den Rang  $k$ , so enthält der Teilbaum mit Wurzel  $x$  mindestens  $F_{k+1}$  Knoten.*

## Beweis

*Sei  $S_k$  die Mindestanzahl von Knoten in einem Teilbaum vom Rang  $k \geq 0$ .*

*Wir zeigen mittels vollständiger Induktion, dass  $S_k \geq F_{k+2}$  gilt.*

*Für  $k = 0$  und  $k = 1$  gilt  $S_0 = 1 = F_2$  und  $S_1 \geq 2 = F_3$ .*

## Beweis (Fortsetzung)

Sei nun  $k + 1$  der Rang von  $x$  und die Behauptung gelte für alle Ränge  $0 \leq i \leq k$ .

Wir betrachten die Kinder  $c_1, \dots, c_{k+1}$  von  $x$ , geordnet nach Alter.

Nach dem vorhergehenden Lemma hat für  $i > 2$  das  $i$ -t älteste Kind mindestens den Rang  $i - 2$  und  $c_1$  und  $c_2$  haben mindestens den Rang 0.

Nach der Induktionsannahme und der Definition von  $S_i$  enthält der Teilbaum mit Wurzel  $c_i$  für  $i \geq 2$ , somit mindestens  $S_{i-2} \geq F_i$  Knoten.

Daher gilt

$$S_{k+1} \geq 2 + \sum_{i=2}^{k+1} S_{i-2} \geq 2 + \sum_{i=2}^{k+1} F_i = F_{k+3}. \quad \square$$

# Der maximale Rang

## Lemma

Für den maximalen Rang eines Baumes in einem  $F$ -Heap  $\mathcal{F}$  mit  $n$  Elementen gilt:

$$r_{\max}(n) \in O(\log n).$$

## Beweis.

Sei  $x$  ein beliebiger Knoten in  $\mathcal{F}$  in dessen Teilbaum  $S$  Knoten enthält.

Dann gilt:  $n \geq S \geq F_{\text{rank}(x)+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^{\text{rank}(x)+2}$ . Dies führt zu

$$\log n \geq (\text{rank}(x) + 2) \log \left( \frac{1 + \sqrt{5}}{2} \right),$$

was wiederum  $\text{rank}_{\max}(n) \in O(\log n)$  impliziert. □

## Die Laufzeit

Damit ergeben sich die folgenden Kosten für die Operationen auf F-Heaps:

<b>Operation</b>	<b>Fibonacci Heaps</b>
min	$O(1)$
union	$O(1)$
insert	$O(1)$
delete_min	$O(T + \log n)$
decrease_key	$O(1) + O(\max(0, k))$
delete	$O(T + \log n + k)$

Dabei ist  $T$  die Anzahl der Bäume im F-Heap und  $k$  die Anzahl der Marken, um die die Gesamtzahl der Marken in dem F-Heaps verringert wird mit  $k \geq -1$ .

# Amortisierte Kosten

Im Folgenden werden wir mit Hilfe der **amortisierten Analyse** zeigen, dass die tatsächlichen Kosten für eine Sequenz von Operationen auf einem Fibonacci Heap deutlich niedriger sind, als die Worst-Case Kosten nahelegen.

Insbesondere werden wir die amortisierten Kosten nur in Abhängigkeit von der Anzahl der Elemente im F-Heap beschreiben.

Als erstes entledigen wir uns der  $O$ -Notation. Durch geeignete Wahl der Konstanten können wir von den folgenden Kosten ausgehen:

- `union` und `insert` benötigen einen Schritt.
- `delete_min` benötigt höchstens  $T + \log n$  Schritte.
- `decrease_key` benötigt höchstens  $k + 2$  Schritte.

# Das Potential

Wir verwenden wieder die **Potentialmethode**.

Die kritischen Maße sind die Zahl  $T$  der Bäume und die Anzahl  $k$  der Markierungen. `decrease_key` verringert die Anzahl der Markierungen, was spätere Aufrufe von `decrease_key` verbilligt, allerdings mehr Bäume erzeugt. Dadurch wird `delete_min` teurer, garantiert aber im Gegenzug, dass die Anzahl der Bäume anschließend wieder auf maximal  $r_{\max}(n)$  sinkt.

Das **Potential**  $\Phi(\mathcal{F})$  eines F-Heaps definieren wir als

$$\Phi(\mathcal{F}) := T + 2M,$$

wobei  $T$  die Anzahl der Bäume und  $M$  die Anzahl der markierten Knoten ist.

# Das Potential

$$\Phi(\mathcal{F}) := T + 2M$$

Offensichtlich hat das Potential die folgenden Eigenschaften:

- $\Phi(\mathcal{F}) \geq 0$
- $\Phi(\mathcal{F}) \in O(n)$
- $\Phi(\emptyset) = 0$

Insbesondere wegen der ersten Eigenschaft können wir die tatsächlichen Kosten für eine beliebige Sequenz von Operationen auf  $\mathcal{F}$  über die amortisierten Kosten abschätzen.

# Die amortisierten Kosten

Für **union** gilt:

$$\hat{c}_{\text{union}} = c_{\text{union}} + 0 = 1$$

denn das Potential ändert sich nicht, da die Anzahl der Bäume und Markierungen gleich bleibt.

Für **insert** ergibt sich:

$$\hat{c}_{\text{insert}} = c_{\text{insert}} + 1 = 2$$

denn das Potential wird um eins erhöht, da die Anzahl der Bäume um eins wächst.

## Die amortisierten Kosten

Nach der Ausführung von `delete_min` ist die Anzahl der Bäume maximal  $r_{\max}(n)$ . Die Anzahl der Markierungen sinkt hingegen, da die Marken der neuen Wurzeln entfernt werden. Damit gilt

$$\begin{aligned}\hat{c}_{\text{delete\_min}} &\leq c_{\text{delete\_min}} + (r_{\max}(n) + 2M) - (T + 2M) \\ &= c_{\text{delete\_min}} + r_{\max}(n) - T \\ &= T + r_{\max}(n) - T \\ &= r_{\max}(n) \in O(\log n)\end{aligned}$$

## Die amortisierten Kosten

Im Laufe von `decrease_key` „verliert“  $\mathcal{F}$  insgesamt  $k \geq -1$  Marken und erhält höchstens  $k + 1$  neue Bäume (einen für jede gelöschte Marke und das veränderte Element).

Damit ergeben sich die amortisierten Kosten als:

$$\begin{aligned}\hat{c}_{\text{decrease\_key}} &\leq c_{\text{decrease\_key}} + (T + k + 1 + 2(M - k)) - (T + 2M) \\ &= k + 2 + T + k + 1 + 2M - 2k - T - 2M \\ &= 3\end{aligned}$$

### Satz

*Für einen Fibonacci Heap mit  $n$  Elementen gelten die folgenden amortisierten Kosten:*

- $\hat{c}_{\text{union}} \in O(1)$
- $\hat{c}_{\text{delete\_min}} \in O(\log n)$
- $\hat{c}_{\text{insert}} \in O(1)$
- $\hat{c}_{\text{decrease\_key}} \in O(1)$

# Priority Queues in der Übersicht

Die Kosten der Operationen auf Prioritätswarteschlangen			
Operation	Binary Heaps	Binomial Queues	F-Heaps*
min	$O(1)$	$O(\log N)$	$O(1)$
union	$O(N_1 \log(N_1 + N_2))$	$O(\log N_1 + \log N_2)$	$O(1)$
insert	$O(\log N)$	$O(\log N)$	$O(1)$
delete_min	$O(\log N)$	$O(\log N)$	$O(\log n)$
decrease_key	$O(1)$	$O(\log N)$	$O(1)$
delete	$O(\log N)$	$O(\log N)$	$O(\log n)$

\* Bei den Fibonacci Heaps handelt es sich um **amortisierte Kosten**.