

Übung 1 – Algorithmen II

Moritz Laupichler, Hans-Peter Lehmann – {moritz.laupichler, hans-peter.lehmann}@kit.edu
http://algo2.iti.kit.edu/AlgorithmenII_WS22.php

Institut für Theoretische Informatik - Algorithmik II

```
    result = current_weight;
    return true;
}

for( EdgeID eid = graph.edgeBegin( current ); eid != graph.edgeEnd( current ); ++eid ){
    const Edge & edge = graph.getEdge( eid );
    COUNTING( statistic_data.inc( DijkstraStatisticData::TOUCHED_EDGES ); )
    if( edge.forward ){
        COUNTING( statistic_data.inc( DijkstraStatisticData::RELAXED_EDGES ); )
        weight new_weight = edge.weight + current_weight;
        GUARANTEE( new_weight >= current_weight, std::runtime_error, "Weight overflow detected." );
        if( !priority_queue.isReached( edge.target ) ){
            COUNTING( statistic_data.inc( DijkstraStatisticData::SUCCESSFULLY_RELAXED_EDGES ); )
            COUNTING( statistic_data.inc( DijkstraStatisticData::REACHED_NODES ); )
            priority_queue.push( edge.target, new_weight );
        } else {
            if( priority_queue.getCurrentKey( edge.target ) > new_weight ){
                COUNTING( statistic_data.inc( DijkstraStatisticData::SUCCESSFULLY_RELAXED_NODES ); )
                priority_queue.decreaseKey( edge.target, new_weight );
            }
        }
    }
}
```

Vorlesungen:

Mo 09:45–11.15 Neue Chemie – Vorlesung

Di 15:45–16:30 Neue Chemie – Vorlesung

Übungen:

Di 16:30–17:15 Neue Chemie – Übung

75% Vertiefung der VL, 25% Besprechung der
Übungsblätter

Ilias Forum:

Fragen zum Vorlesungsinhalt

Vorlesungsaufzeichnung:

Vorlesungen der letzten Jahre auf Youtube

↪ Link: *Algorithmen II (WS20/21, Englisch) - Playlist*

↪ Link: *Algorithmen II (WS19/20, Deutsch) - Playlist*

Übungsblätter:

- Ausgabe: 14-tägig, jeweils Dienstag
- Musterlösung: 14 Tage nach Veröffentlichung
- Besprechung:
 - 14 Tage nach Veröffentlichung
 - pro Blatt 1-2 Aufgaben ← **Abstimmung auf Ilias**
 - Abstimmen bis Sonntag vor Besprechung
- 1. Blatt:
 - Ausgabe am 08.11.2022
 - Abstimmung bis 20.11.2022
 - Besprechung und Musterlösung am 22.11.2022

Sprechstunden:

Corona-bedingt: Ilias Forum oder E-Mail bevorzugt

Letzte Vorlesung:

14.02.2023

Klausur:

16.03.2023, 11:30 Uhr

- Fibonacci Heaps
 - Wiederholung der Operationen
 - Ammortisierte Analyse mit Potentialmethode

Fibonacci Heaps - Insert

minPtr



2

Fibonacci Heaps - Insert

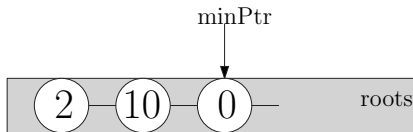


10

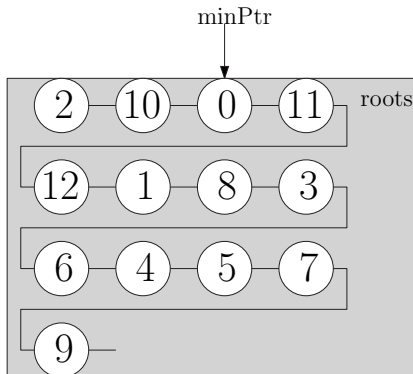
Fibonacci Heaps - Insert



Fibonacci Heaps - Insert



Fibonacci Heaps - Insert



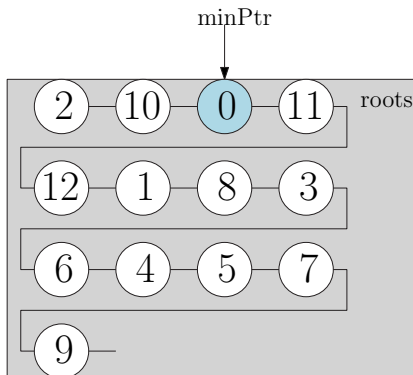
Fibonacci Heaps - Insert

- Insert in konstanter Zeit
- Erzeugt großen Wald einzelner Knoten
- Entspricht ohne **union** nur linearer Liste

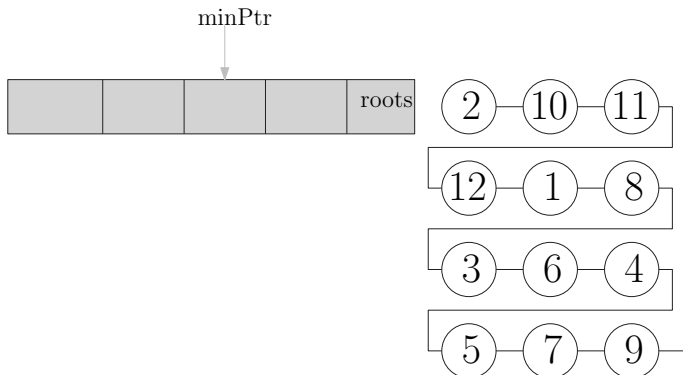
Delete Min:

- delete minimum m from forest
- for each child h of m :
 - make new tree with root h
- while $\exists a, b \in \text{forest with rank}(a) = \text{rank}(b)$ do:
 - union(a, b)
- return m

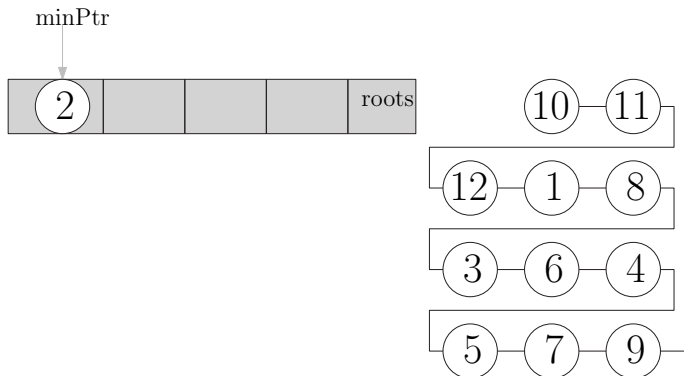
Fibonacci Heaps - Delete Min



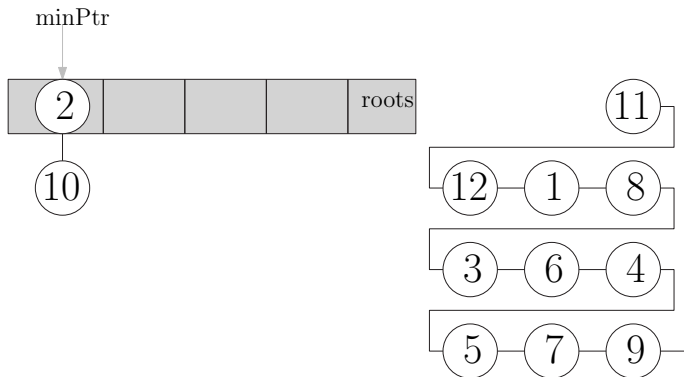
Fibonacci Heaps - Delete Min



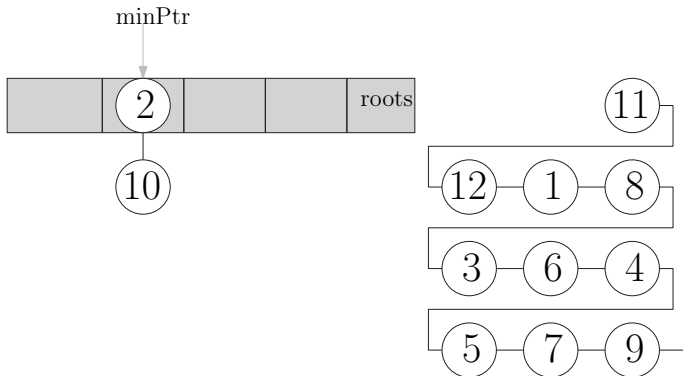
Fibonacci Heaps - Delete Min



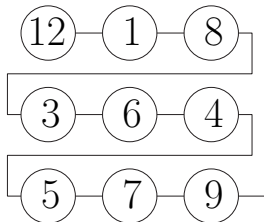
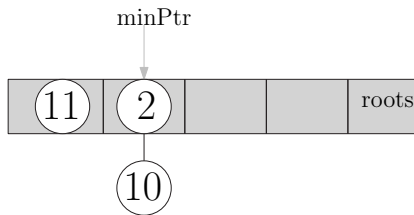
Fibonacci Heaps - Delete Min



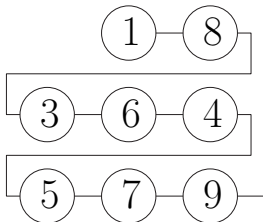
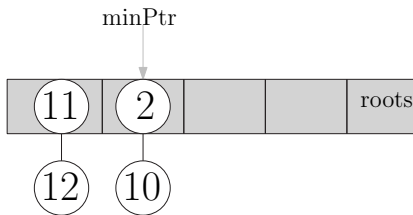
Fibonacci Heaps - Delete Min



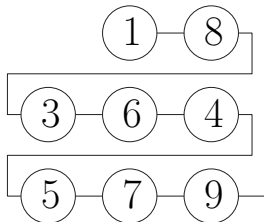
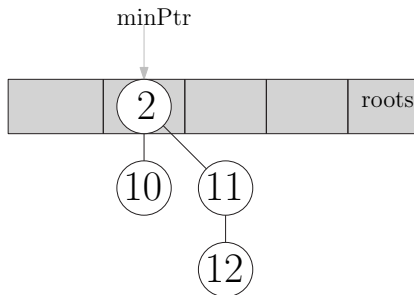
Fibonacci Heaps - Delete Min



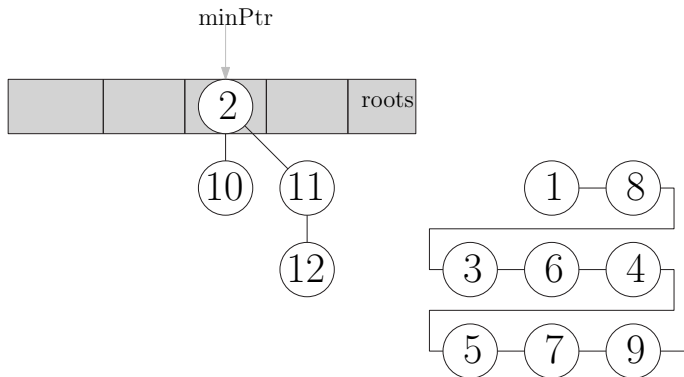
Fibonacci Heaps - Delete Min



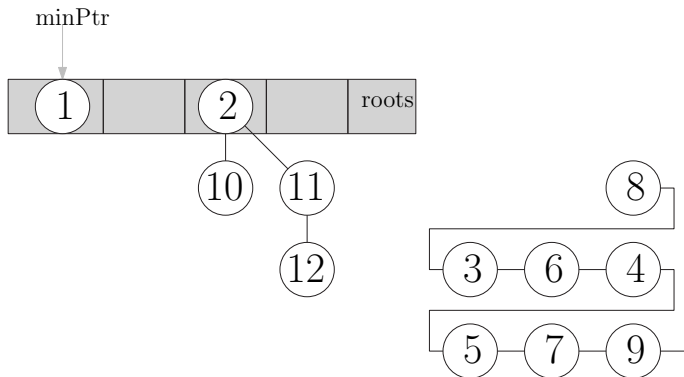
Fibonacci Heaps - Delete Min



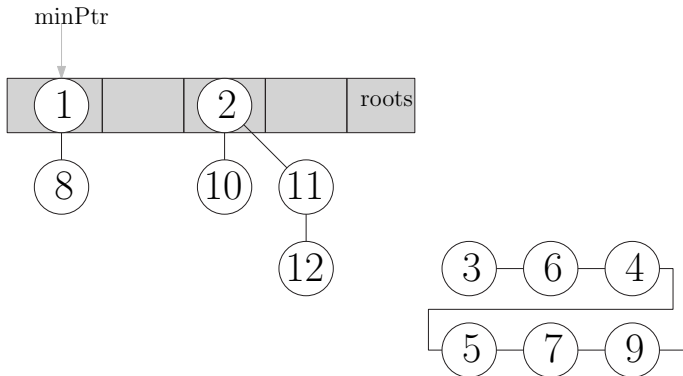
Fibonacci Heaps - Delete Min



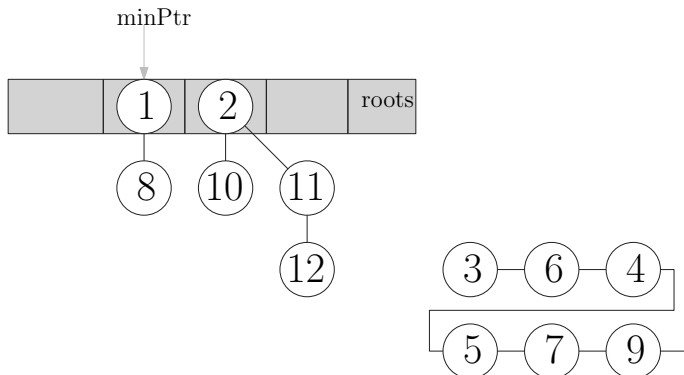
Fibonacci Heaps - Delete Min



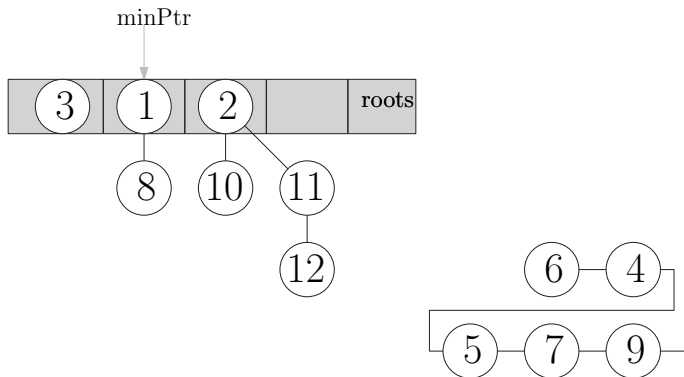
Fibonacci Heaps - Delete Min



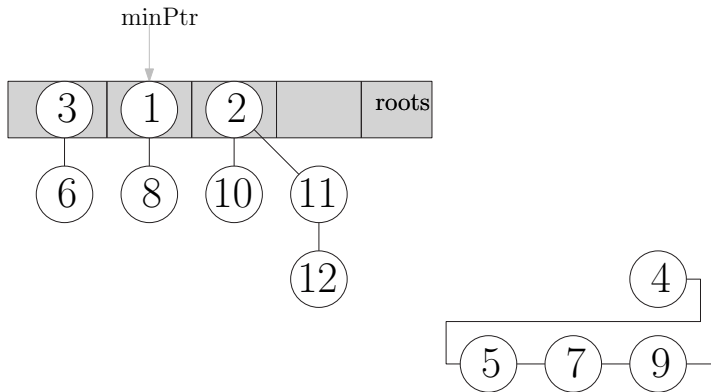
Fibonacci Heaps - Delete Min



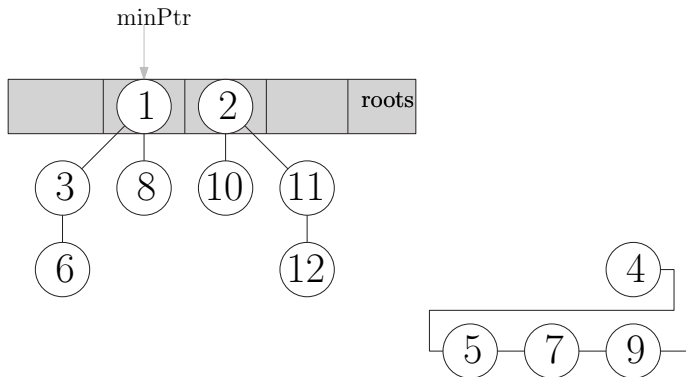
Fibonacci Heaps - Delete Min



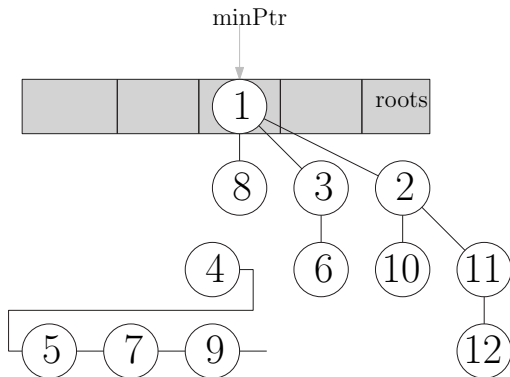
Fibonacci Heaps - Delete Min



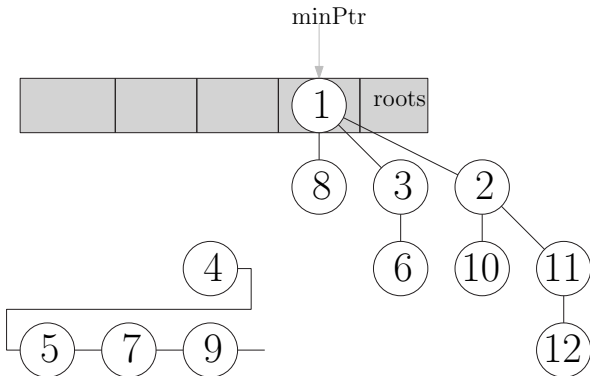
Fibonacci Heaps - Delete Min



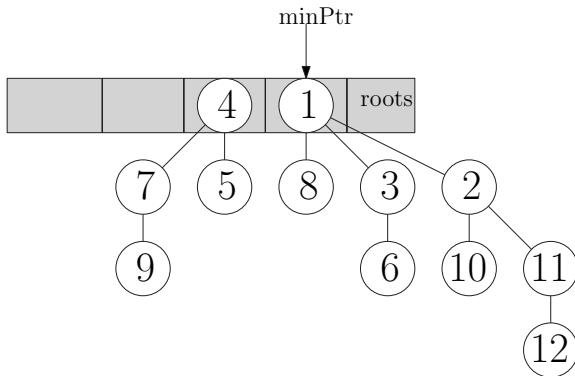
Fibonacci Heaps - Delete Min



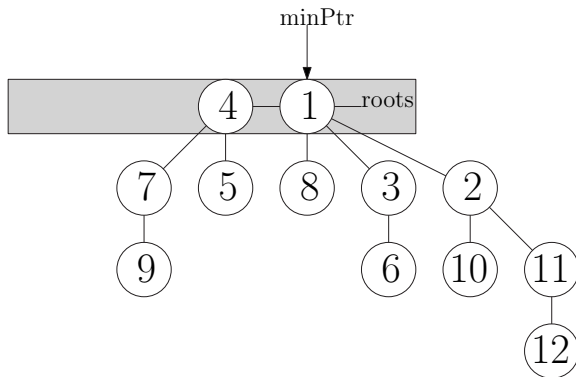
Fibonacci Heaps - Delete Min



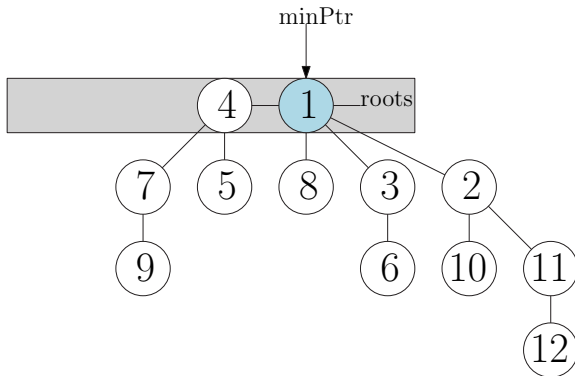
Fibonacci Heaps - Delete Min



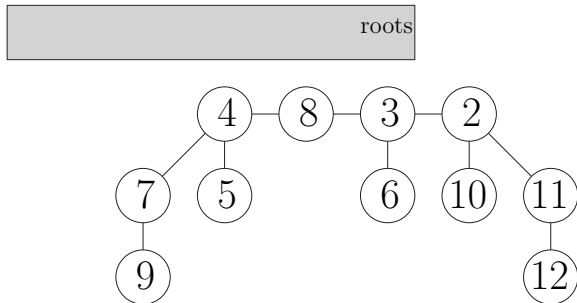
Fibonacci Heaps - Delete Min



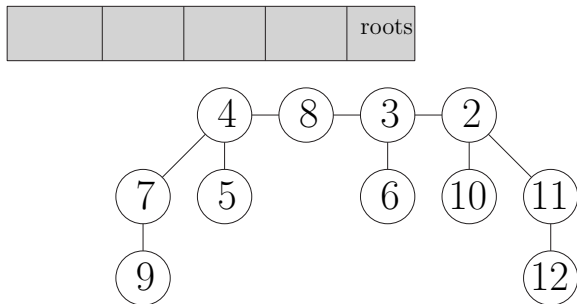
Fibonacci Heaps - Delete Min



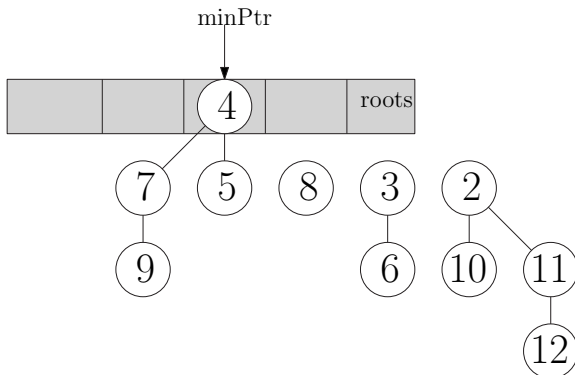
Fibonacci Heaps - Delete Min



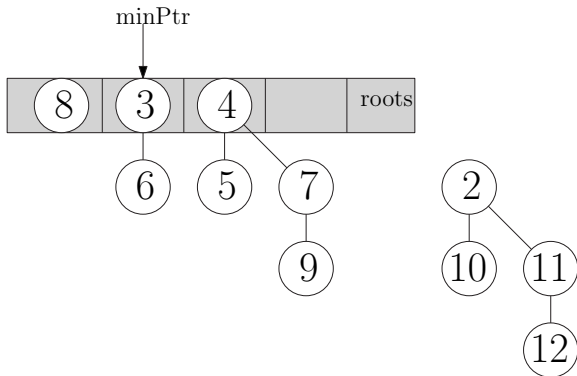
Fibonacci Heaps - Delete Min



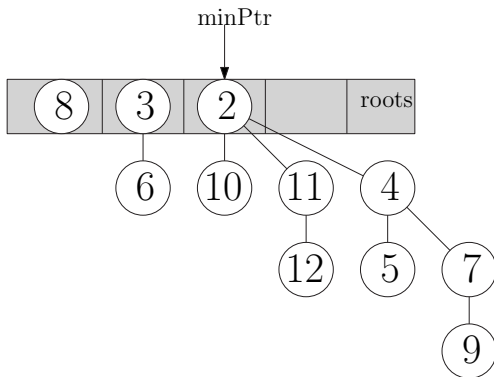
Fibonacci Heaps - Delete Min



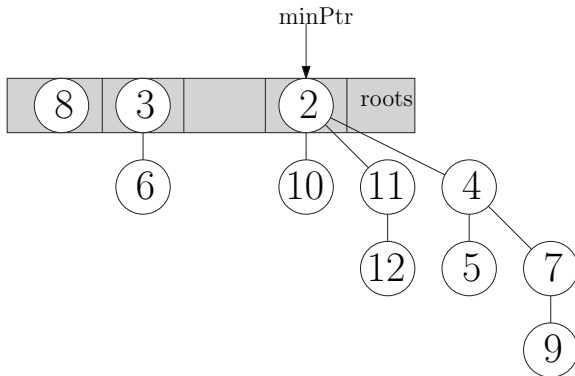
Fibonacci Heaps - Delete Min



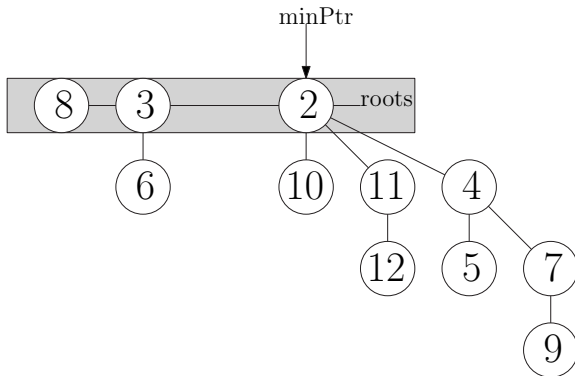
Fibonacci Heaps - Delete Min



Fibonacci Heaps - Delete Min



Fibonacci Heaps - Delete Min



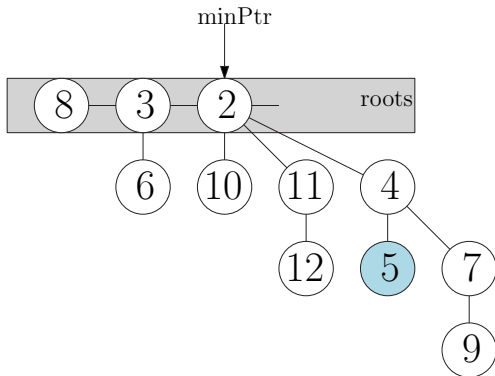
Decrease Key(h : Handle, k : Key):

- $\text{key}(h) \leftarrow k$
- $\text{cascadingCut}(h)$:

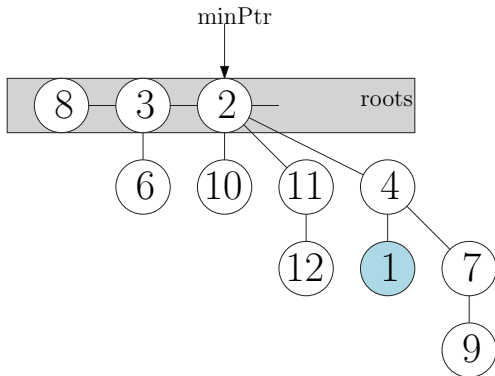
cascadingCut(h : Handle)

- if h is not a root:
 - $p \leftarrow \text{parent}(h)$
 - unmark h
 - $\text{cut}(h)$
 - if p is marked:
 - $\text{cascadingCut}(p)$
 - else:
 - mark p

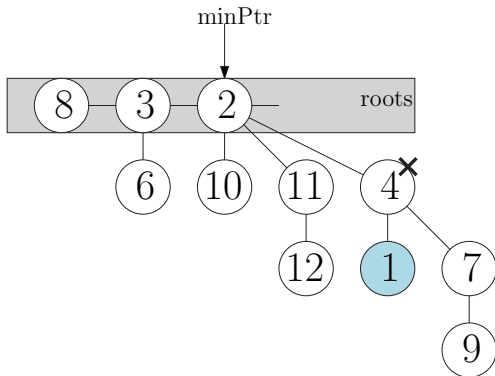
Fibonacci Heaps - Decrease Key



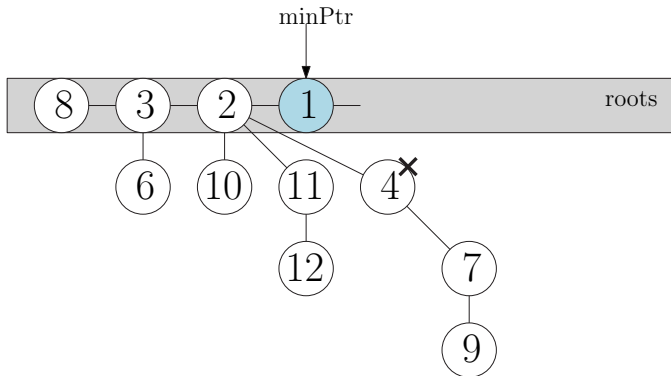
Fibonacci Heaps - Decrease Key



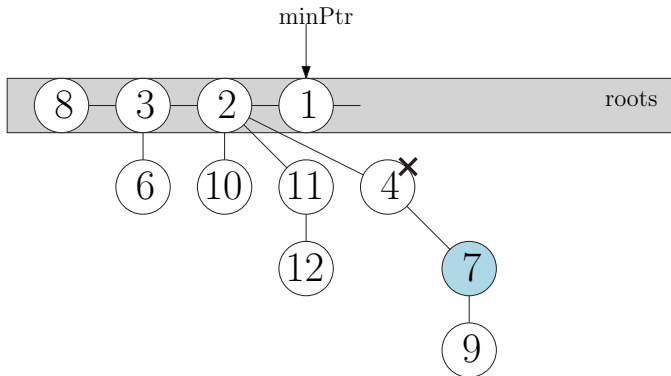
Fibonacci Heaps - Decrease Key



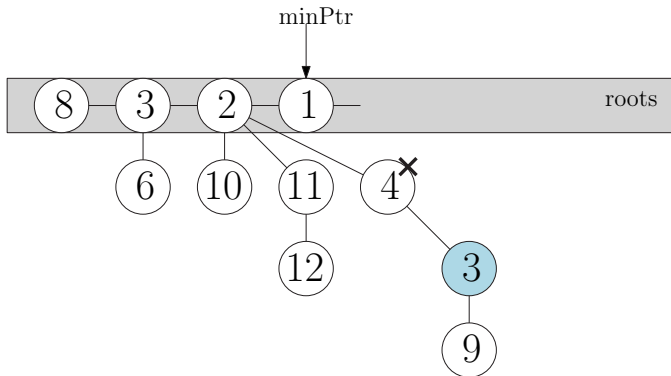
Fibonacci Heaps - Decrease Key



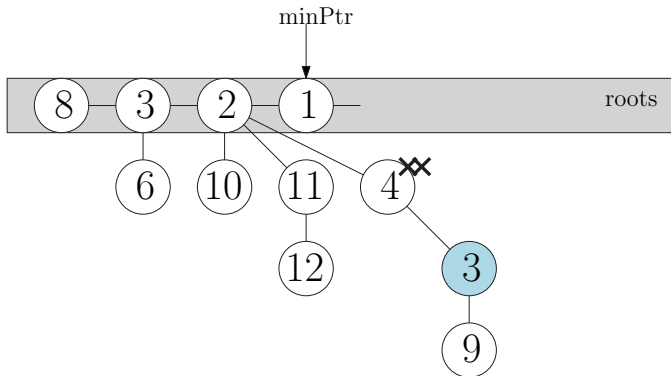
Fibonacci Heaps - Decrease Key



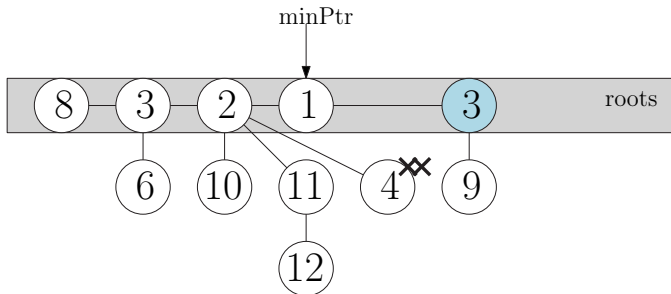
Fibonacci Heaps - Decrease Key



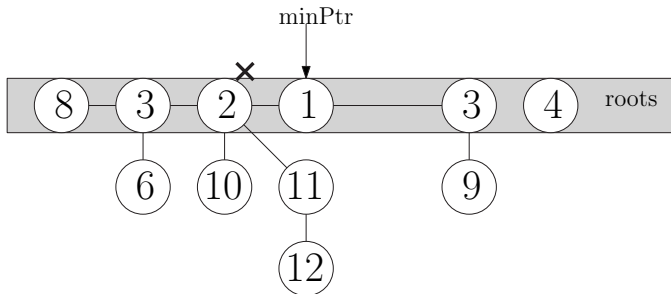
Fibonacci Heaps - Decrease Key



Fibonacci Heaps - Decrease Key



Fibonacci Heaps - Decrease Key



- Operationen $Op = \langle Op_1, \dots, Op_n \rangle$ angewandt auf Datenstruktur D
- Op_i überführt D von Zustand s_{i-1} in s_i
- Kosten/Laufzeit $T_{Op_i}(s_{i-1})$ (T_{Op_i} hängt von Zustand s_{i-1} ab)

$$T(Op) = \sum_{i=1}^n T_{Op_i}(s_{i-1})$$

- **Gesucht:** Amortisierte Laufzeit $A_{Op_i}(s_{i-1})$ für jede Operation, sodass

$$T(Op) \leq A(Op) := c + \sum_{i=1}^n A_{Op_i}(s_{i-1}) \text{ für beliebige Konstante } c$$

- Operationen $Op = \langle Op_1, \dots, Op_n \rangle$ angewandt auf Datenstruktur D
- Op_i überführt D von Zustand s_{i-1} in s_i
- Kosten/Laufzeit $T_{Op_i}(s_{i-1})$ (T_{Op_i} hängt von Zustand s_{i-1} ab)

$$T(Op) = \sum_{i=1}^n T_{Op_i}(s_{i-1})$$

- **Gesucht:** Amortisierte Laufzeit $A_{Op_i}(s_{i-1})$ für jede Operation, sodass

$$T(Op) \leq A(Op) := c + \sum_{i=1}^n A_{Op_i}(s_{i-1}) \text{ für beliebige Konstante } c$$

- Potentialfunktion $\Phi : S \rightarrow \mathbb{R}_{\geq}$ ($S = \text{Zustandsraum}$)
- $\Delta\Phi(s_{i-1}) := \Phi(s_i) - \Phi(s_{i-1})$

$$A_{Op_i}(s_{i-1}) := T_{Op_i}(s_{i-1}) + \Delta\Phi(s_{i-1})$$
$$\Rightarrow T(OP) \leq A(OP) + \Phi(s_0) \text{ (ohne Beweis)}$$

- **Anmerkung:** Φ kann beliebig gewählt werden (z.B. $\Phi(s) = 42$ für alle $s \in S$), jedoch führen nicht alle Φ zu sinnvollen amortisierten Schranken.

- Potentialfunktion $\Phi : S \rightarrow \mathbb{R}_>$ ($S = \text{Zustandsraum}$)
- $\Delta\Phi(s_{i-1}) := \Phi(s_i) - \Phi(s_{i-1})$

$$A_{Op_i}(s_{i-1}) := T_{Op_i}(s_{i-1}) + \Delta\Phi(s_{i-1})$$
$$\Rightarrow T(OP) \leq A(OP) + \Phi(s_0) \text{ (ohne Beweis)}$$

- **Anmerkung:** Φ kann beliebig gewählt werden (z.B. $\Phi(s) = 42$ für alle $s \in S$), jedoch führen nicht alle Φ zu sinnvollen amortisierten Schranken.

- Potentialfunktion $\Phi : S \rightarrow \mathbb{R}_{>}$ ($S = \text{Zustandsraum}$)
- $\Delta\Phi(s_{i-1}) := \Phi(s_i) - \Phi(s_{i-1})$

$$\begin{aligned} A_{Op_i}(s_{i-1}) &:= T_{Op_i}(s_{i-1}) + \Delta\Phi(s_{i-1}) \\ \Rightarrow T(OP) &\leq A(OP) + \Phi(s_0) \text{ (ohne Beweis)} \end{aligned}$$

- **Anmerkung:** Φ kann beliebig gewählt werden (z.B. $\Phi(s) = 42$ für alle $s \in S$), jedoch führen nicht alle Φ zu sinnvollen amortisierten Schranken.

- Potentialfunktion $\Phi : S \rightarrow \mathbb{R}_{>}$ ($S = \text{Zustandsraum}$)
- $\Delta\Phi(s_{i-1}) := \Phi(s_i) - \Phi(s_{i-1})$

$$\begin{aligned} A_{Op_i}(s_{i-1}) &:= T_{Op_i}(s_{i-1}) + \Delta\Phi(s_{i-1}) \\ \Rightarrow T(OP) &\leq A(OP) + \Phi(s_0) \text{ (ohne Beweis)} \end{aligned}$$

- **Anmerkung:** Φ kann beliebig gewählt werden (z.B. $\Phi(s) = 42$ für alle $s \in S$), jedoch führen nicht alle Φ zu sinnvollen amortisierten Schranken.

- Dynamisches Array A , das nur wächst
 - $size(s)$: Anzahl Elemente im Array in Zustand s
 - $cap(s)$: Kapazität in Zustand s (= Anzahl Elemente, für die Speicher allokiert ist)
- $push_back(s, e : \text{Element}) : s'$
 - if $size(s) = cap(s)$ then
 - allocate memory for $cap(s') := 2 \cdot cap(s)$ elements
 - copy contents of A to allocated memory
 - $A[size(s)] := e$
 - $size(s') := size(s) + 1$
- $T_{push_back}(s) = \mathcal{O}(1 + copy(s))$
 - $copy(s) := \# \text{kopierte Elemente für } push_back \text{ (0 oder } cap(s))$

- Dynamisches Array A , das nur wächst
 - $size(s)$: Anzahl Elemente im Array in Zustand s
 - $cap(s)$: Kapazität in Zustand s (= Anzahl Elemente, für die Speicher allokiert ist)
- $push_back(s, e : \text{Element}) : s'$
 - if $size(s) = cap(s)$ then
 - allocate memory for $cap(s') := 2 \cdot cap(s)$ elements
 - copy contents of A to allocated memory
 - $A[size(s)] := e$
 - $size(s') := size(s) + 1$
- $T_{push_back}(s) = \mathcal{O}(1 + copy(s))$
 - $copy(s) := \# \text{kopierte Elemente für } push_back \text{ (0 oder } cap(s))$

- Dynamisches Array A , das nur wächst
 - $size(s)$: Anzahl Elemente im Array in Zustand s
 - $cap(s)$: Kapazität in Zustand s (= Anzahl Elemente, für die Speicher allokiert ist)
- $push_back(s, e : \text{Element}) : s'$
 - if $size(s) = cap(s)$ then
 - allocate memory for $cap(s') := 2 \cdot cap(s)$ elements
 - copy contents of A to allocated memory
 - $A[size(s)] := e$
 - $size(s') := size(s) + 1$
- $T_{push_back}(s) = \mathcal{O}(1 + copy(s))$
 - $copy(s) := \# \text{kopierte Elemente für } push_back \text{ (0 oder } cap(s))$

Einfaches Beispiel: Wachsendes Array

Konto-Methode

- nach “Umkopieren”: $cap/2$ Plätze frei
- `push_back`-Operationen für diese Plätze zahlen nächstes Umkopieren
- Jede `push_back`-Operation zahlt 2 Token ($\mathcal{O}(1)$)
- Bei nächstem Umkopieren genau cap Token angespart
- $\Rightarrow$ amortisierte Laufzeit $A_{\text{push_back}}(s) = \mathcal{O}(1)$

Einfaches Beispiel: Wachsendes Array

Potential-Methode

$$\Phi(s) = 2size(s) - cap(s) + 1$$

- `push_back` überführt Array von Zustand s nach s'
 - Erinnerung: $T_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s))$
 - Es gilt $copy(s) = cap(s') - cap(s)$
 - Amortisierte Kosten: $A_{\text{push_back}}(s) = T_{\text{push_back}}(s) + \Delta\Phi(s)$
 - $\Delta\Phi(s) = 2\Delta size(s) - \Delta cap(s) = 2 - copy(s)$
 - $size(s') = size(s) + 1 \Rightarrow \Delta size(s) = 1$
 - $cap(s') = cap(s) + copy(s) \Rightarrow \Delta cap(s) = copy(s)$
- $\Rightarrow A_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s)) + 2 - copy(s) = \mathcal{O}(1)$

$$\Phi(s) = 2\text{size}(s) - \text{cap}(s) + 1$$

- `push_back` überführt Array von Zustand s nach s'
 - Erinnerung: $T_{\text{push_back}}(s) = \mathcal{O}(1 + \text{copy}(s))$
 - Es gilt $\text{copy}(s) = \text{cap}(s') - \text{cap}(s)$
 - Amortisierte Kosten: $A_{\text{push_back}}(s) = T_{\text{push_back}}(s) + \Delta\Phi(s)$
 - $\Delta\Phi(s) = 2\Delta\text{size}(s) - \Delta\text{cap}(s) = 2 - \text{copy}(s)$
 - $\text{size}(s') = \text{size}(s) + 1 \Rightarrow \Delta\text{size}(s) = 1$
 - $\text{cap}(s') = \text{cap}(s) + \text{copy}(s) \Rightarrow \Delta\text{cap}(s) = \text{copy}(s)$
- $\Rightarrow A_{\text{push_back}}(s) = \mathcal{O}(1 + \text{copy}(s)) + 2 - \text{copy}(s) = \mathcal{O}(1)$

$$\Phi(s) = 2size(s) - cap(s) + 1$$

- `push_back` überführt Array von Zustand s nach s'
 - Erinnerung: $T_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s))$
 - Es gilt $copy(s) = cap(s') - cap(s)$
 - Amortisierte Kosten: $A_{\text{push_back}}(s) = T_{\text{push_back}}(s) + \Delta\Phi(s)$
 - $\Delta\Phi(s) = 2\Delta size(s) - \Delta cap(s) = 2 - copy(s)$
 - $size(s') = size(s) + 1 \Rightarrow \Delta size(s) = 1$
 - $cap(s') = cap(s) + copy(s) \Rightarrow \Delta cap(s) = copy(s)$
- $\Rightarrow A_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s)) + 2 - copy(s) = \mathcal{O}(1)$

$$\Phi(s) = 2size(s) - cap(s) + 1$$

- `push_back` überführt Array von Zustand s nach s'
- Erinnerung: $T_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s))$
- Es gilt $copy(s) = cap(s') - cap(s)$
- Amortisierte Kosten: $A_{\text{push_back}}(s) = T_{\text{push_back}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = 2\Delta size(s) - \Delta cap(s) = 2 - copy(s)$
 - $size(s') = size(s) + 1 \Rightarrow \Delta size(s) = 1$
 - $cap(s') = cap(s) + copy(s) \Rightarrow \Delta cap(s) = copy(s)$

$$\Rightarrow A_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s)) + 2 - copy(s) = \mathcal{O}(1)$$

$$\Phi(s) = 2size(s) - cap(s) + 1$$

- `push_back` überführt Array von Zustand s nach s'
- Erinnerung: $T_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s))$
- Es gilt $copy(s) = cap(s') - cap(s)$
- Amortisierte Kosten: $A_{\text{push_back}}(s) = T_{\text{push_back}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = 2\Delta size(s) - \Delta cap(s) = 2 - copy(s)$
 - $size(s') = size(s) + 1 \Rightarrow \Delta size(s) = 1$
 - $cap(s') = cap(s) + copy(s) \Rightarrow \Delta cap(s) = copy(s)$

$$\Rightarrow A_{\text{push_back}}(s) = \mathcal{O}(1 + copy(s)) + 2 - copy(s) = \mathcal{O}(1)$$

$$\Phi(s) := \text{roots}(s) + 2 \cdot \text{marks}(s)$$

- $\text{roots}(s)$ = Anzahl an Wurzeln im Zustand s
- $\text{marks}(s)$ = Anzahl markierter Knoten im Zustand s

Fibonacci Heaps - Delete Min

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DeleteMin überführt Heap von Zustand s nach s'
- Laufzeit DeleteMin: $T_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s))$
 - $roots(s)$ = Anzahl an **merge** Operationen
 - $maxRank(s)$ = Maximale Anzahl an Kindern eines Knoten im Zustand s
- Amortisierte Kosten: $A_{DeleteMin}(s) = T_{DeleteMin}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq maxRank(s) + 1 - roots(s)$
 - $marks(s') \leq marks(s) \Rightarrow \Delta marks(s) \leq 0$
 - $roots(s') \leq maxRank(s) + 1 \Rightarrow \Delta roots(s) \leq maxRank(s) + 1 - roots(s)$

$$\Rightarrow A_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s)) + maxRank(s) + 1 - roots(s) = \mathcal{O}(maxRank(s))$$

Fibonacci Heaps - Delete Min

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DeleteMin überführt Heap von Zustand s nach s'
- Laufzeit DeleteMin: $T_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s))$
 - $roots(s)$ = Anzahl an **merge** Operationen
 - $maxRank(s)$ = Maximale Anzahl an Kindern eines Knoten im Zustand s
- Amortisierte Kosten: $A_{DeleteMin}(s) = T_{DeleteMin}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq maxRank(s) + 1 - roots(s)$
 - $marks(s') \leq marks(s) \Rightarrow \Delta marks(s) \leq 0$
 - $roots(s') \leq maxRank(s) + 1 \Rightarrow \Delta roots(s) \leq maxRank(s) + 1 - roots(s)$

$$\Rightarrow A_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s)) + maxRank(s) + 1 - roots(s) = \mathcal{O}(maxRank(s))$$

Fibonacci Heaps - Delete Min

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DeleteMin überführt Heap von Zustand s nach s'
- Laufzeit DeleteMin: $T_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s))$
 - $roots(s)$ = Anzahl an **merge** Operationen
 - $maxRank(s)$ = Maximale Anzahl an Kindern eines Knoten im Zustand s
- Amortisierte Kosten: $A_{DeleteMin}(s) = T_{DeleteMin}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq maxRank(s) + 1 - roots(s)$
 - $marks(s') \leq marks(s) \Rightarrow \Delta marks(s) \leq 0$
 - $roots(s') \leq maxRank(s) + 1 \Rightarrow \Delta roots(s) \leq maxRank(s) + 1 - roots(s)$

$$\Rightarrow A_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s)) + maxRank(s) + 1 - roots(s) = \mathcal{O}(maxRank(s))$$

Fibonacci Heaps - Delete Min

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DeleteMin überführt Heap von Zustand s nach s'
- Laufzeit DeleteMin: $T_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s))$
 - $roots(s)$ = Anzahl an **merge** Operationen
 - $maxRank(s)$ = Maximale Anzahl an Kindern eines Knoten im Zustand s
- Amortisierte Kosten: $A_{DeleteMin}(s) = T_{DeleteMin}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq maxRank(s) + 1 - roots(s)$
 - $marks(s') \leq marks(s) \Rightarrow \Delta marks(s) \leq 0$
 - $roots(s') \leq maxRank(s) + 1 \Rightarrow \Delta roots(s) \leq maxRank(s) + 1 - roots(s)$

$$\Rightarrow A_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s)) + maxRank(s) + 1 - roots(s) = \mathcal{O}(maxRank(s))$$

Fibonacci Heaps - Delete Min

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DeleteMin überführt Heap von Zustand s nach s'
- Laufzeit DeleteMin: $T_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s))$
 - $roots(s)$ = Anzahl an **merge** Operationen
 - $maxRank(s)$ = Maximale Anzahl an Kindern eines Knoten im Zustand s
- Amortisierte Kosten: $A_{DeleteMin}(s) = T_{DeleteMin}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq maxRank(s) + 1 - roots(s)$
 - $marks(s') \leq marks(s) \Rightarrow \Delta marks(s) \leq 0$
 - $roots(s') \leq maxRank(s) + 1 \Rightarrow \Delta roots(s) \leq maxRank(s) + 1 - roots(s)$

$$\Rightarrow A_{DeleteMin}(s) = \mathcal{O}(roots(s) + maxRank(s)) + maxRank(s) + 1 - roots(s) = \mathcal{O}(maxRank(s))$$

Fibonacci Heaps - Decrease Key

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DecreaseKey(h) überführt Heap von Zustand s nach s'
- Laufzeit DecreaseKey: $T_{\text{DecreaseKey}}(s) = \mathcal{O}(cuts(h, s) + 1)$
 - $cuts(h, s)$ = Anzahl nötige Schnitte (= 1 + Anzahl an markierten *direkten* Parents von Handle h im Zustand s (Cascading Cuts))
- Amortisierte Kosten: $A_{\text{DecreaseKey}}(s) = T_{\text{DecreaseKey}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq 4 - cuts(h, s)$
 - $roots(s') = roots(s) + cuts(h, s) \Rightarrow \Delta roots(s) = cuts(h, s)$
 - $marks(s') \leq marks(s) - (cuts(h, s) - 1) + 1$
 $\Rightarrow \Delta marks(s') \leq 2 - cuts(h, s)$

$$\Rightarrow A_{\text{DeleteMin}}(s) = \mathcal{O}(cuts(h, s) + 1) + 4 - cuts(h, s) = \mathcal{O}(1)$$

Fibonacci Heaps - Decrease Key

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DecreaseKey(h) überführt Heap von Zustand s nach s'
- Laufzeit DecreaseKey: $T_{\text{DecreaseKey}}(s) = \mathcal{O}(cuts(h, s) + 1)$
 - $cuts(h, s)$ = Anzahl nötige Schnitte (= 1 + Anzahl an markierten *direkten* Parents von Handle h im Zustand s (Cascading Cuts))
- Amortisierte Kosten: $A_{\text{DecreaseKey}}(s) = T_{\text{DecreaseKey}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq 4 - cuts(h, s)$
 - $roots(s') = roots(s) + cuts(h, s) \Rightarrow \Delta roots(s) = cuts(h, s)$
 - $marks(s') \leq marks(s) - (cuts(h, s) - 1) + 1$
 $\Rightarrow \Delta marks(s') \leq 2 - cuts(h, s)$

$$\Rightarrow A_{\text{DeleteMin}}(s) = \mathcal{O}(cuts(h, s) + 1) + 4 - cuts(h, s) = \mathcal{O}(1)$$

Fibonacci Heaps - Decrease Key

Amortisierte Analyse

Wiederholung: $\Phi(s) := \text{roots}(s) + 2 \cdot \text{marks}(s)$

- DecreaseKey(h) überführt Heap von Zustand s nach s'
- Laufzeit DecreaseKey: $T_{\text{DecreaseKey}}(s) = \mathcal{O}(\text{cuts}(h, s) + 1)$
 - $\text{cuts}(h, s)$ = Anzahl nötige Schnitte (= 1 + Anzahl an markierten *direkten* Parents von Handle h im Zustand s (Cascading Cuts))
- Amortisierte Kosten: $A_{\text{DecreaseKey}}(s) = T_{\text{DecreaseKey}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta\text{roots}(s) + 2 \cdot \Delta\text{marks}(s) \leq 4 - \text{cuts}(h, s)$
 - $\text{roots}(s') = \text{roots}(s) + \text{cuts}(h, s) \Rightarrow \Delta\text{roots}(s) = \text{cuts}(h, s)$
 - $\text{marks}(s') \leq \text{marks}(s) - (\text{cuts}(h, s) - 1) + 1$
 $\Rightarrow \Delta\text{marks}(s') \leq 2 - \text{cuts}(h, s)$

$$\Rightarrow A_{\text{DeleteMin}}(s) = \mathcal{O}(\text{cuts}(h, s) + 1) + 4 - \text{cuts}(h, s) = \mathcal{O}(1)$$

Fibonacci Heaps - Decrease Key

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DecreaseKey(h) überführt Heap von Zustand s nach s'
- Laufzeit DecreaseKey: $T_{\text{DecreaseKey}}(s) = \mathcal{O}(cuts(h, s) + 1)$
 - $cuts(h, s)$ = Anzahl nötige Schnitte (= 1 + Anzahl an markierten *direkten* Parents von Handle h im Zustand s (Cascading Cuts))
- Amortisierte Kosten: $A_{\text{DecreaseKey}}(s) = T_{\text{DecreaseKey}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq 4 - cuts(h, s)$
 - $roots(s') = roots(s) + cuts(h, s) \Rightarrow \Delta roots(s) = cuts(h, s)$
 - $marks(s') \leq marks(s) - (cuts(h, s) - 1) + 1$
 $\Rightarrow \Delta marks(s') \leq 2 - cuts(h, s)$

$$\Rightarrow A_{\text{DeleteMin}}(s) = \mathcal{O}(cuts(h, s) + 1) + 4 - cuts(h, s) = \mathcal{O}(1)$$

Fibonacci Heaps - Decrease Key

Amortisierte Analyse

Wiederholung: $\Phi(s) := roots(s) + 2 \cdot marks(s)$

- DecreaseKey(h) überführt Heap von Zustand s nach s'
- Laufzeit DecreaseKey: $T_{\text{DecreaseKey}}(s) = \mathcal{O}(cuts(h, s) + 1)$
 - $cuts(h, s)$ = Anzahl nötige Schnitte (= 1 + Anzahl an markierten *direkten* Parents von Handle h im Zustand s (Cascading Cuts))
- Amortisierte Kosten: $A_{\text{DecreaseKey}}(s) = T_{\text{DecreaseKey}}(s) + \Delta\Phi(s)$
- $\Delta\Phi(s) = \Delta roots(s) + 2 \cdot \Delta marks(s) \leq 4 - cuts(h, s)$
 - $roots(s') = roots(s) + cuts(h, s) \Rightarrow \Delta roots(s) = cuts(h, s)$
 - $marks(s') \leq marks(s) - (cuts(h, s) - 1) + 1$
 $\Rightarrow \Delta marks(s') \leq 2 - cuts(h, s)$

$$\Rightarrow A_{\text{DeleteMin}}(s) = \mathcal{O}(cuts(h, s) + 1) + 4 - cuts(h, s) = \mathcal{O}(1)$$

- Amortisierte Laufzeiten sind **optimal** (vergleichsbasiert)
- **Einzelne Operationen** Kosten aber deutlich **länger**

Ende!



Feierabend!