Amortisierte Analyse
und
Fibonacci Heaps

Amortisierte Analyse - Modell

- Worst-Case-Laufzeiten für Einzeloperationen für die Abschätzung der Gesamtlaufzeit häufig ungeeignet
- **Idee:** Statt Einzeloperationen, betrachte *Sequenz* von Operationen Op_1-Op_n

$$S_0 \xrightarrow{Op_1} S_1 \xrightarrow{Op_2} S_3 \xrightarrow{Op_3} \ldots \xrightarrow{Op_n} S_n$$

- Die Datenstruktur durchläuft dabei die Zustände $S_0 \ldots S_n$
- Die Laufzeiten für die Operationen sind $T_1 \ldots T_n$
- Die Gesamtzeit beträgt somit $T = \sum_i T_i$
- Die Abschätzung bezieht sich auf die *Sequenz*, gibt aber *durchschnittliche* Kosten für die *Einzeloperationen* an!

Amortisierte Analyse

- Begriff der Amortisierung stammt aus der BWL:

  Eine teure Investition zahlt sich über längere Sicht hin aus und wird über die gesamte Zeit der Wirksamkeit der Investition *amortisiert*.
- Auf Algorithmen zum ersten Mal Anfang der 80er Jahre angewandt [Brown, Huddleston, Mehlhorn, Sleator, Tarjan u.a.]
- Abgrenzung zur Average-Case-Analyse
  - Keine Wahrscheinlichkeiten notwendig!
  - Ergebnis: durchschnittliche Laufzeit jeder Operation im schlimmsten Fall! ⇒ Leistungsgarantie!

Beispiel: Binärzähler

- Sequenz: Zählen für k = 0…N
- Operation: Inkrementieren
- Kosten: Ändern einer Ziffer (0→1 oder 1→0)
- Betrachte
  - Kosten pro Operation T_{k+1} (# invertierte Bits für k → k + 1)
  - Gesamtkosten T

k	bin(k)
0	000000
1	000001
2	000010
3	000011
4	000100
5	000101
6	000110
7	000111
8	001000
9	001001
10	001010
11	001011
12	001100
13	001101
14	001110
15	001111
16	010000

Beispiel: Binärzähler

- Beobachtung
  $1 \leq T_k \leq 1 + \lceil \log k \rceil$
- Worst-Case-Abschätzung
  $T_k = 1 + \lceil \log k \rceil$
  $T = N \cdot (1 + \lceil \log N \rceil)$

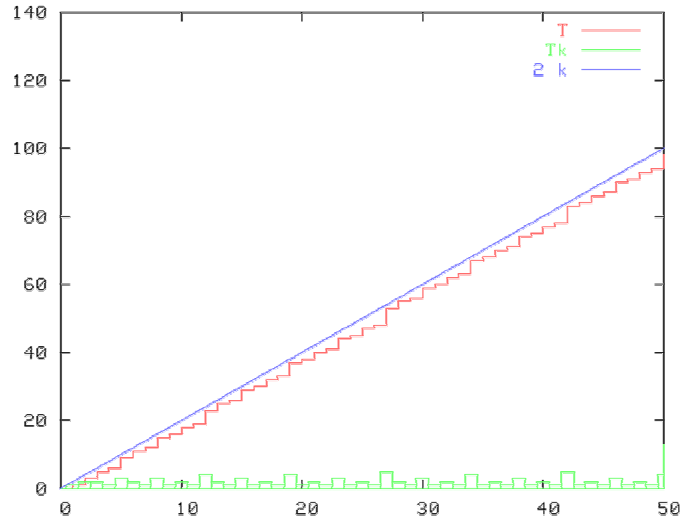
k	bin(k)	T_k
0	000000	0
1	000001	1
2	000010	2
3	000011	1
4	000100	3
5	000101	1
6	000110	1
7	000111	2
8	001000	4
9	001001	1
10	001010	2
11	001011	2
12	001100	3
13	001101	1
14	001110	2
15	001111	1
16	010000	5

Beispiel: Binärzähler

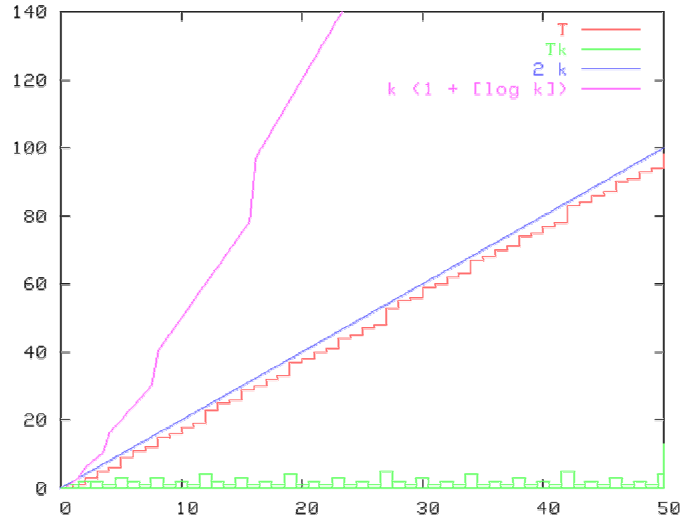
- Beobachtung
  $1 \leq T_k \leq 1 + \lceil \log k \rceil$
- Worst-Case-Abschätzung
  $T_k = 1 + \lceil \log k \rceil$
  $T = N \cdot (1 + \lceil \log N \rceil)$
- Weitere Beobachtung
  $\sum T_k \leq 2k$

k	bin(k)	T_k	$\sum T_k$
0	000000	0	0
1	000001	1	1
2	000010	2	3
3	000011	1	4
4	000100	3	7
5	000101	1	8
6	000110	1	9
7	000111	2	11
8	001000	4	15
9	001001	1	16
10	001010	2	18
11	001011	2	20
12	001100	3	23
13	001101	1	24
14	001110	2	26
15	001111	1	27
16	010000	5	32

Beispiel: Binärzähler



Beispiel: Binärzähler



Binärzähler: Analyse

k	bin(k)	T_k	ΣT_k
0	000000	0	0
1	000001	1	1
2	000010	2	3
3	000011	1	4
4	000100	3	7
5	000101	1	8
6	000110	1	9
7	000111	2	11
8	001000	4	15
9	001001	1	16
10	001010	2	18
11	001011	2	20
12	001100	3	23
13	001101	1	24
14	001110	2	26
15	001111	1	27
16	010000	5	32

Binärzähler: Analyse

- B[0] wird jeden Schritt invertiert

k	bin(k)	T_k	ΣT_k
0	000000	0	0
1	000001	1	1
2	000010	2	3
3	000011	1	4
4	000100	3	7
5	000101	1	8
6	000110	1	9
7	000111	2	11
8	001000	4	15
9	001001	1	16
10	001010	2	18
11	001011	2	20
12	001100	3	23
13	001101	1	24
14	001110	2	26
15	001111	1	27
16	010000	5	32

Binärzähler: Analyse

- B[0] wird jeden Schritt invertiert
- B[1] wird jeden zweiten Schritt invertiert
- B[k] wird alle 2^k Schritte invertiert

k	bin(k)	T_k	ΣT_k
0	000000	0	0
1	000001	1	1
2	000010	2	3
3	000011	1	4
4	000100	3	7
5	000101	1	8
6	000110	1	9
7	000111	2	11
8	001000	4	15
9	001001	1	16
10	001010	2	18
11	001011	2	20
12	001100	3	23
13	001101	1	24
14	001110	2	26
15	001111	1	27
16	010000	5	32

Binärzähler: Analyse

- Addition über die Zeilen der Tabelle lieferte uns

$$T = \sum_{i=0}^{N} T_i \leq N(1 + \lceil logN \rceil)$$

- Addition über die Spalten hingegen

$$T = \sum_{c=0}^{\lfloor logN \rfloor} \left\lfloor \frac{N}{2^c} \right\rfloor \leq \sum_{c=0}^{\lfloor logN \rfloor} \frac{N}{2^c} < N \sum_{c=0}^{\infty} \frac{1}{2^c} = 2N$$

- Laufzeit für das Zählen von k = 0…N: $\mathcal{O}$(N)
- Amortisierte Laufzeit für Inkrement: $\mathcal{O}$(N)/N = $\mathcal{O}$(1)

Aggregatmethode

- Berechne die Gesamtkosten für eine Sequenz von Operationen
- Division der Gesamtkosten durch die Zahl der Operationen liefert amortisierte Kosten pro Operation

Dieses Vorgehen heißt Aggregatmethode und ist die einfachste Technik der amortisierten Analyse. Die zweite wichtige Technik ist die Bankkonto-Methode (siehe 5. Übungsblatt!)

Potenzialmethode

- **Idee:** Operationen können potenzielle Energie erzeugen/verbrauchen
- Die Funktion Φ ordnet jedem Zustand S_i unserer Datenstruktur ein Potenzial $\Phi(S_i)$ zu
- Die amortisierten Kosten a_i der Operation Op_i hängen von den tatsächlichen Kosten c_i und dem Potenzial Φ ab

$$a_i = c_i + \Delta\Phi = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

- Die Gesamtkosten der N Operationen sind damit

$$\sum_{i=1}^{N} a_i = \sum_{i=1}^{N} (c_i + \Phi(S_i) - \Phi(S_{i-1}))$$

$$= \Phi(S_N) - \Phi(S_0) + \sum_{i=1}^{N} c_i$$

Potenzialfunktionen

- „Billige Operationen“ erhöhen das Potenzial über ihre eigenen Kosten hinaus:

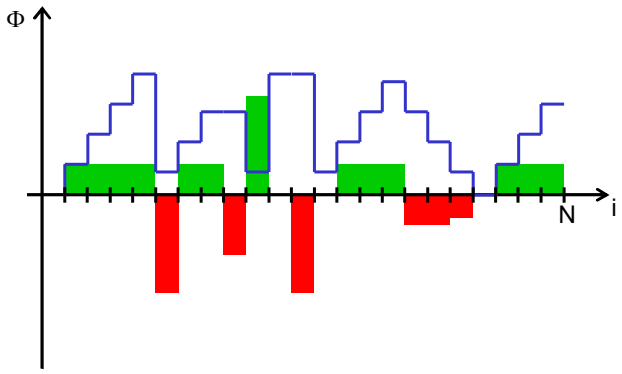
$$a_i > c_i$$

- „Teure Operationen“ verringern das Potenzial, bezahlen also einen Teil der Operation mit dem was die vorhergehenden billigen Operationen an „Potenzieller Energie“ angesammelt haben:

$$a_i < c_i$$

- Für $\Phi(S_N) \geq \Phi(S_0)$ sind die amortisierten Kosten der N Operationen eine obere Schranke für die tatsächlichen Kosten

Potenzialverlauf



Binärzähler mit der Potenzialmethode

Definiere das Potenzial als Anzahl der gesetzten Bits im derzeitigen Zählerstand:

$$\Phi(S_i) = \Phi_i = \#1 \in \text{bin}(i)$$

Der Zählerstand sei i und der Übergang zu i+1 soll t_i Bits zurücksetzen ($1 \rightarrow 0$). Ferner wird maximal ein weiteres Bit gesetzt

$$\Rightarrow c_i = t_i + 1$$

Für die Potenzialänderung gilt dann:

$$\Phi_i - \Phi_{i-1} \leq \Phi_{i-1} - t_i + 1 - \Phi_{i-1} = 1 - t_i$$

Damit gilt für die amortisierten Kosten:

$$a_i = c_i + \Phi_1 - \Phi_{i-1} \leq (t_i+1) + (1 - t_i) = 2 \Rightarrow \mathcal{O}(1)$$

Amortisierte Analyse von Heaps

Operation	Binärer Heap	Binomial-Heap
`insert`	$\mathcal{O}$(log N)	$\mathcal{O}$(log N)
`find_min`	$\mathcal{O}$(1)	$\mathcal{O}$(1)
`delete_min`	$\mathcal{O}$(log N)	$\mathcal{O}$(log N)
`decrease_key`	$\mathcal{O}$(log N)	$\mathcal{O}$(log N)
`create`	$\mathcal{O}$(N)	$\mathcal{O}$(N)
`merge`	$\mathcal{O}$(N)	$\mathcal{O}$(log N)

Amortisierte Analyse von Heaps

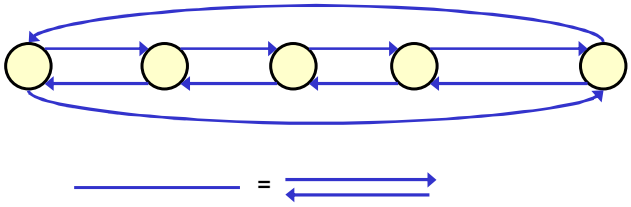
Operation	Binärer Heap	Binomial-Heap	Fibonacci-Heap
`insert`	$\mathcal{O}$(log N)	$\mathcal{O}$(log N)	?
`find_min`	$\mathcal{O}$(1)	$\mathcal{O}$(1)	?
`delete_min`	$\mathcal{O}$(log N)	$\mathcal{O}$(log N)	?
`decrease_key`	$\mathcal{O}$(log N)	$\mathcal{O}$(log N)	?
`create`	$\mathcal{O}$(N)	$\mathcal{O}$(N)	?
`merge`	$\mathcal{O}$(N)	$\mathcal{O}$(log N)	?

Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.

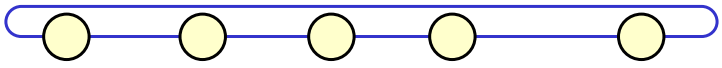
Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.



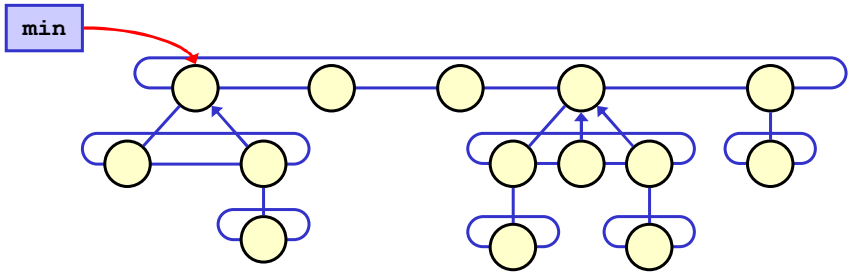
Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.



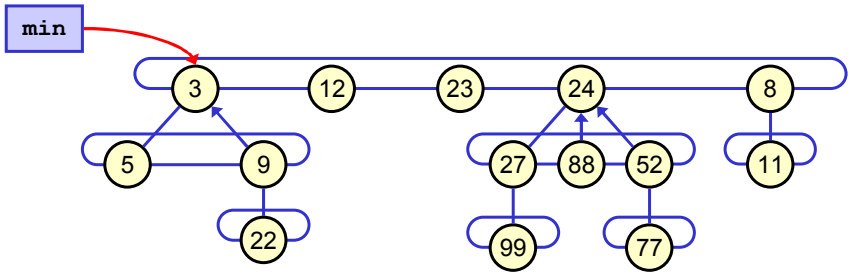
Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.



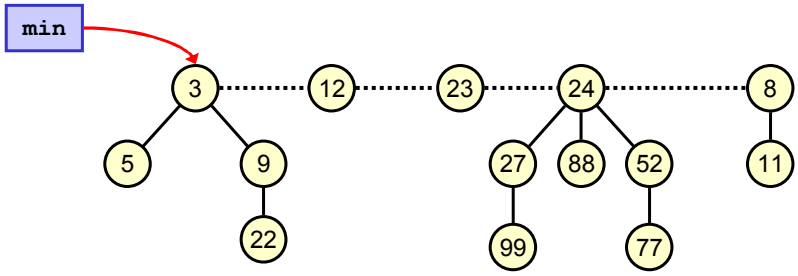
Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.



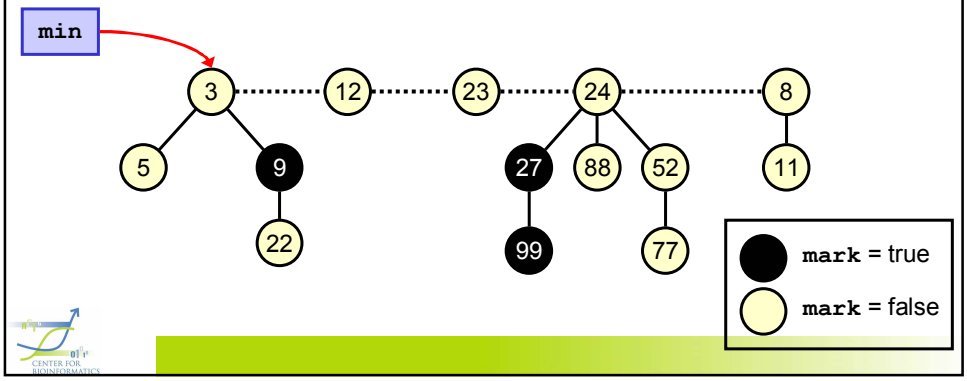
Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.



Fibonacci-Heaps

Def.: Ein *Fibonacci-Heap* ist eine Datenstruktur, die einen Wald von Bäumen mit Heapeigenschaft besitzt. Jeder der Knoten (außer den Wurzeln) trägt ein zusätzliches Markierungsbit **`mark`**. Das Element **`min`** zeigt auf die kleinste Wurzel des Walds. Die Wurzeln sind doppelt zyklisch verkettet, ebenso die Kinder eines jeden Knotens.



FibonacciHeap – Interface

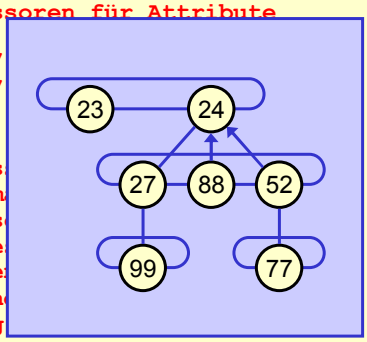
```
template <typename Key, typename Inf>
class FibonacciHeap
{
  public:
  typedef std::pair<Key, Inf> Item;
  typedef FibonacciNode* NodePtr;

  const Item& find_min() const;
  void delete_min();
  void insert(const Item& item);

  void create(const std::vector<Item>& array);
  void merge(FibonacciHeap& heap);
  void decrease_key(NodePtr item, const Key& key);
```

FibonacciHeap – Interface

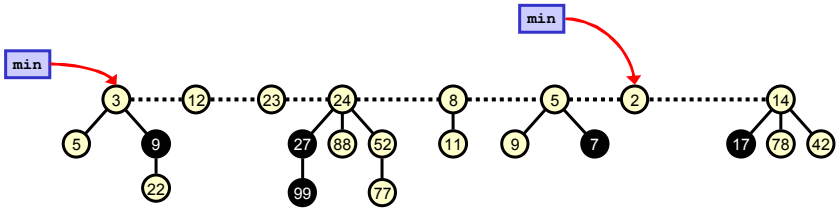
```
  protected:
  // Hilfsmethoden
  void consolidate();
  void cut(NodePtr node);
  void cascading_cut(NodePtr node);
  void link(NodePtr root, NodePtr child);
  void insert_into_rootlist(NodePtr node);
  void erase_from_rootlist(NodePtr node);
  unsigned int max_deg() const; // max. Grad der Wurzeln

  // Attribute
  int number_of_nodes;          // Gesamtzahl Knoten
  NodePtr min;                  // Minmale Wurzel
  // min ist auch der Anfang der Wurzelliste!
};
```

FibonacciNode – Interface

```
template <typename Key, typename Inf>
class FibonacciNode
{
  public:
  typedef FibonacciNode* NodePtr;

  const Key& key() const { return key; }
  Key& key() { return key; }  // Accessoren für Attribute
  […]
  unsigned int degree() const;       //
  void remove_child(NodePtr child); //

  protected:
  Key key;                  // Der Schlüs
  Inf inf;                  // Die Inform
  NodePtr left;             // Linker Ges
  NodePtr right;            // Rechter Ge
  NodePtr parent;           // Elterknote
  NodePtr first_child;      // Erster Kin
  bool mark;                // Markierung
};
```



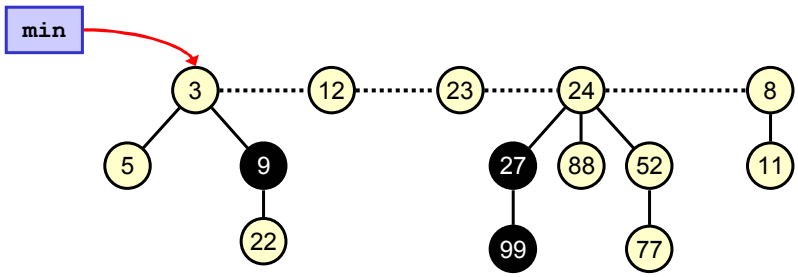
`find_min`, `merge` im Fibonacci-Heap



- **`merge`**
  - Vereinige die Wurzellisten
  - **`min`**(H) = min(**`min`**(H_1), **`min`**(H_2))

    $\Rightarrow \mathcal{O}(1)$
- **`find_min`**: $\mathcal{O}(1)$

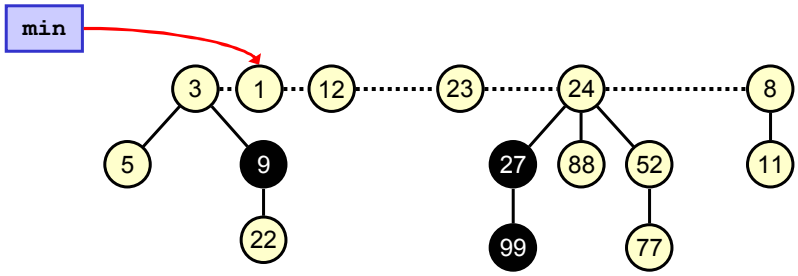
`insert` im Fibonacci-Heaps



Knoten wird neben dem Minimum in der Wurzelliste eingefügt, der Zeiger auf das Minimum gegebenenfalls angepasst

$\Rightarrow \mathcal{O}(1)$

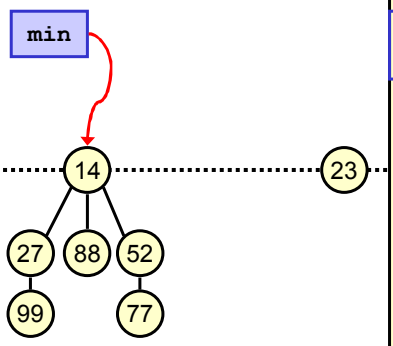
`insert` im Fibonacci-Heaps



Knoten wird neben dem Minimum in der Wurzelliste eingefügt, der Zeiger auf das Minimum gegebenenfalls angepasst

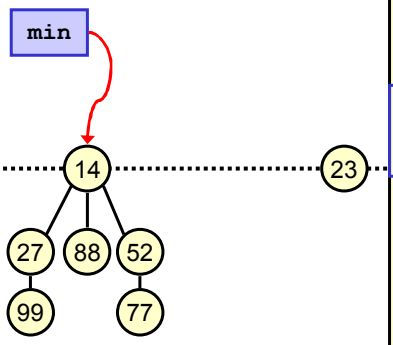
$\Rightarrow \mathcal{O}(1)$

delete_min im Fibonacci-Heap



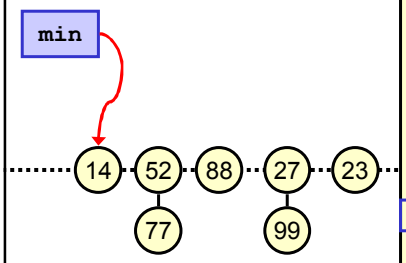

```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node:
    child.parent = 0;
    insert_into_rootlist(child);

  erase_from_rootlist(node);
  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();
  }
  delete node;
  number_of_nodes--;
}
```


delete_min im Fibonacci-Heap



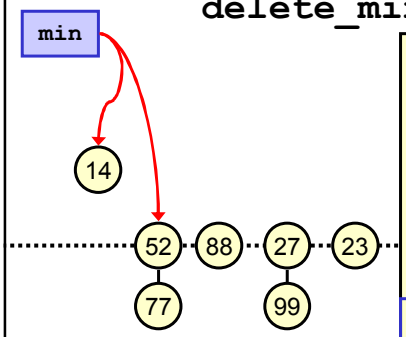

```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node:
    child.parent = 0;
    insert_into_rootlist(child);

  erase_from_rootlist(node);
  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();
  }
  delete node;
  number_of_nodes--;
}
```

delete_min im Fibonacci-Heap




```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node:
    child.parent = 0;
    insert_into_rootlist(child);

  erase_from_rootlist(node);
  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();
  }
  delete node;
  number_of_nodes--;
}
```


delete_min im Fibonacci-Heap

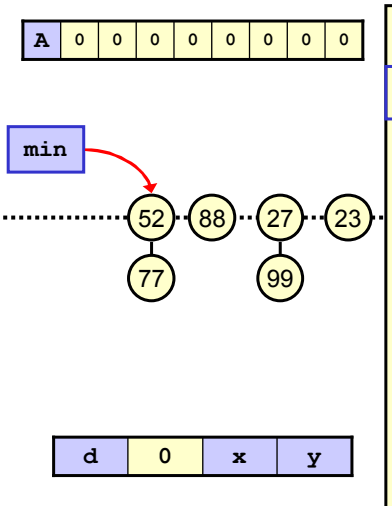



```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node:
    child.parent = 0;
    insert_into_rootlist(child);

  erase_from_rootlist(node);
  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();
  }
  delete node;
  number_of_nodes--;
}
```

`consolidate` im Fibonacci-Heap

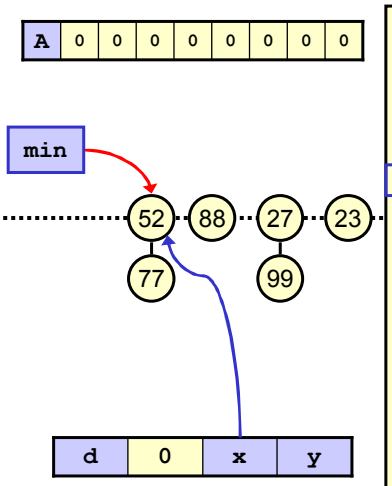
- **`insert`** und **`delete_min`** erhöhen die Anzahl der Bäume
- Gelegentlich muss die Zahl der Bäume reduziert werden ⇒ **`consolidate`**
- **Idee:** Verschmelze Bäume gleichen Grads in der Wurzel bis nur noch Bäume mit unterschiedlichem Grad übrig bleiben. Erhalte dabei die Heapeigenschaft.

`consolidate` im Fibonacci-Heap



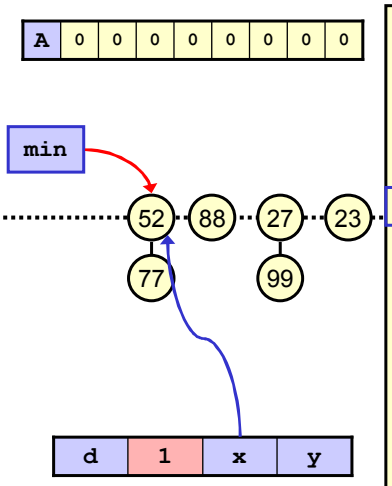

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0; // Löscht Wurzelliste!
```

consolidate im Fibonacci-Heap



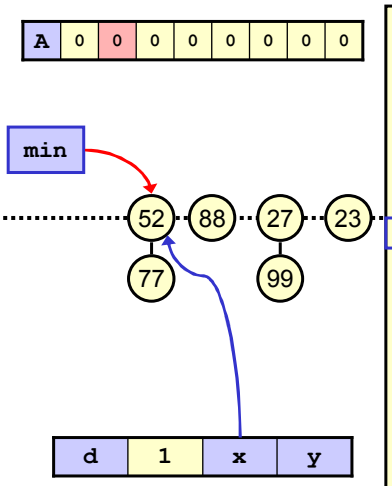

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



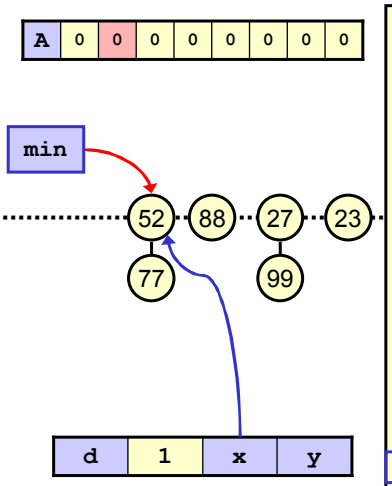

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



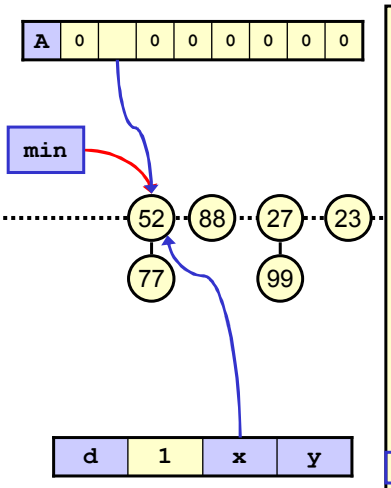

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



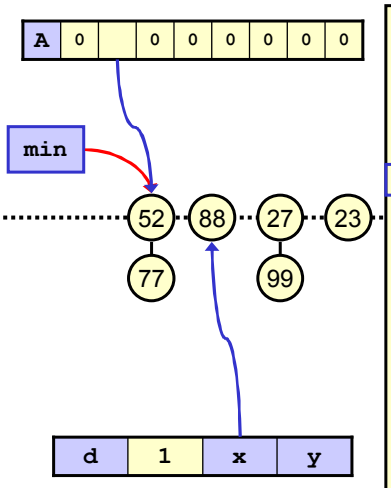

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



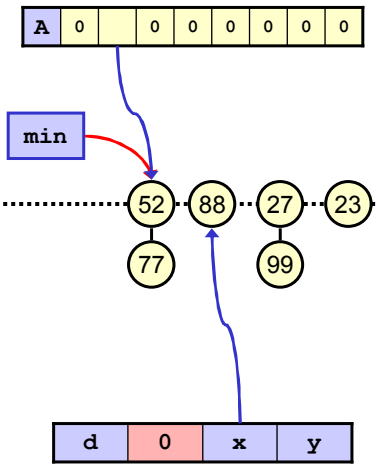

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



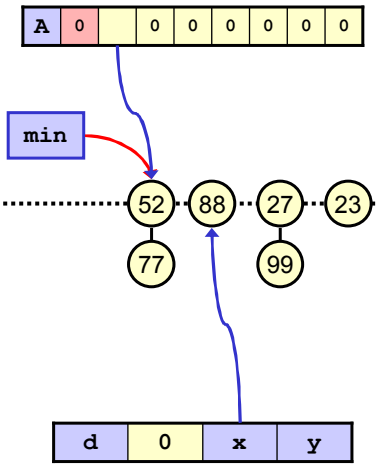

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



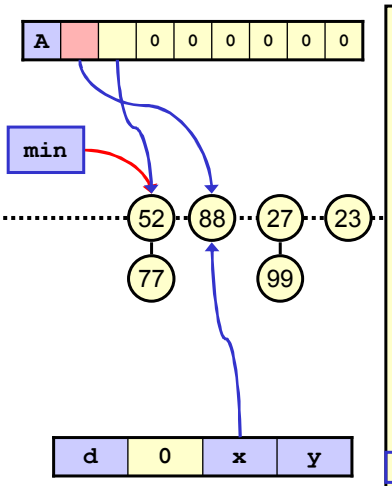

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



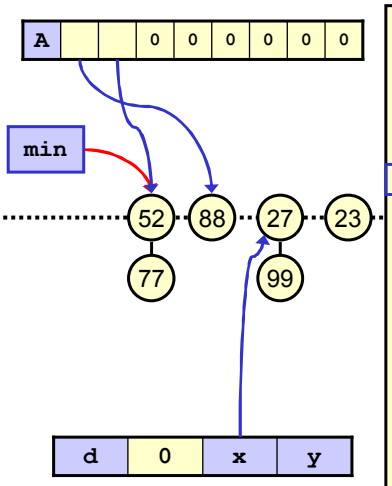

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



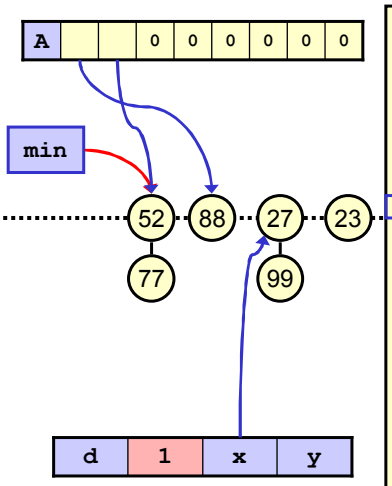

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



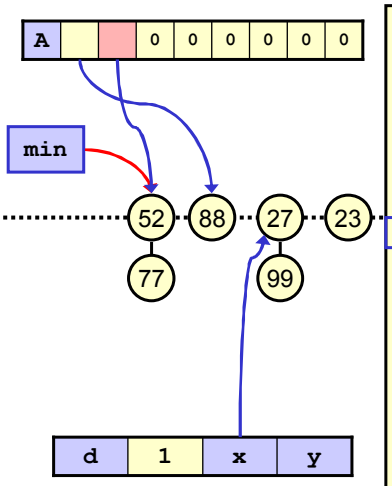

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



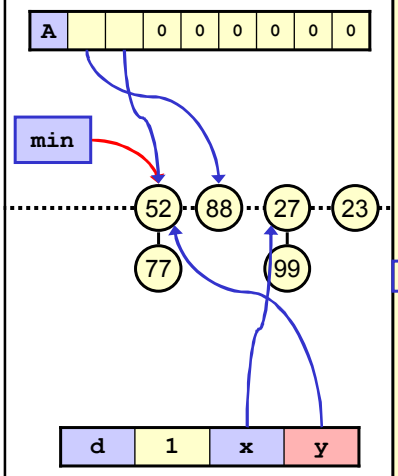

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



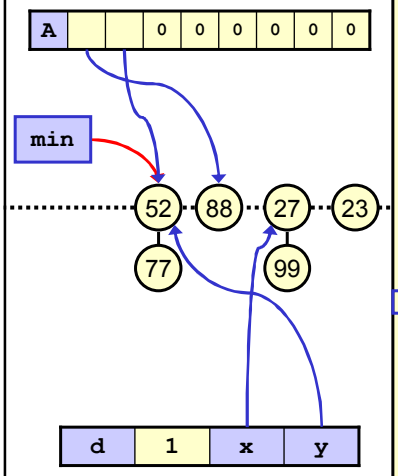

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



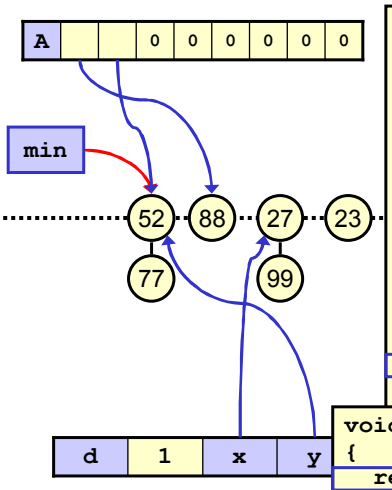

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap




```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

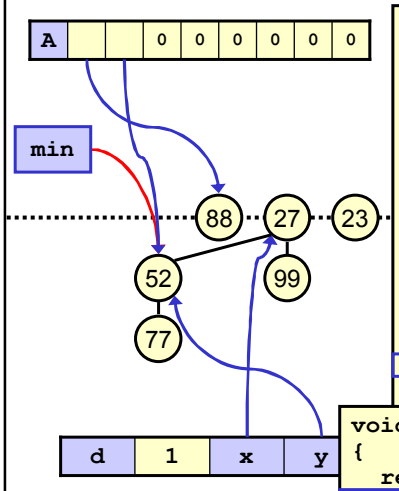
consolidate im Fibonacci-Heap




```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
```

```
void link(NodePtr root, NodePtr child)
{
  remove from root list(child);
  root.insert_child(child);
  child->mark() = false;
}
```

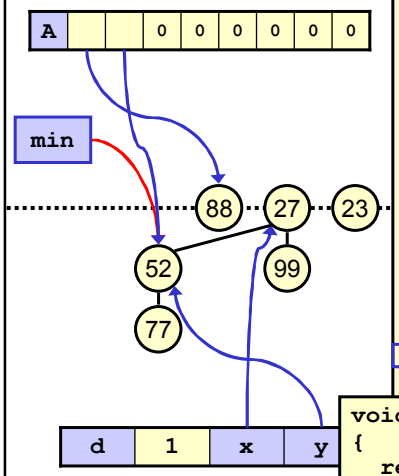

consolidate im Fibonacci-Heap




```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
```

```
void link(NodePtr root, NodePtr child)
{
  remove from rootlist(child);
  root.insert_child(child);
  child->mark() = false;
}
```

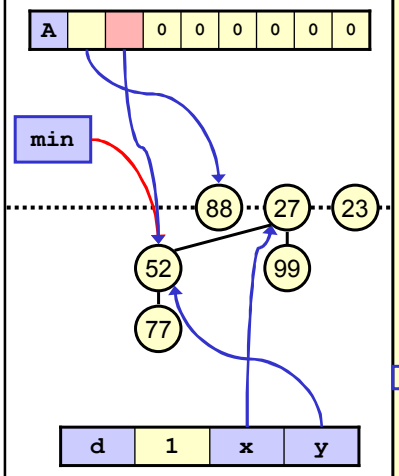
consolidate im Fibonacci-Heap




```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
```

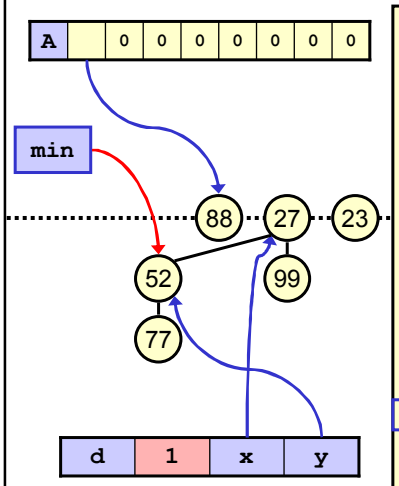
```
void link(NodePtr root, NodePtr child)
{
  remove_from_rootlist(child);
  root.insert_child(child);
  child->mark() = false;
}
```


consolidate im Fibonacci-Heap



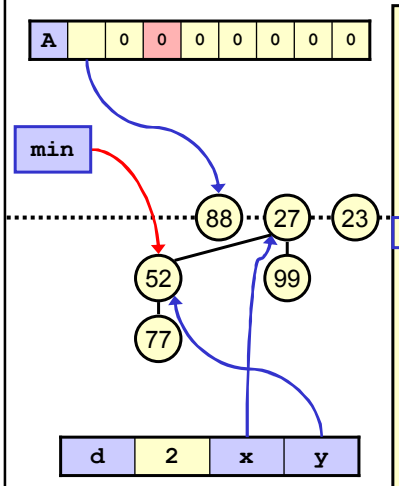

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



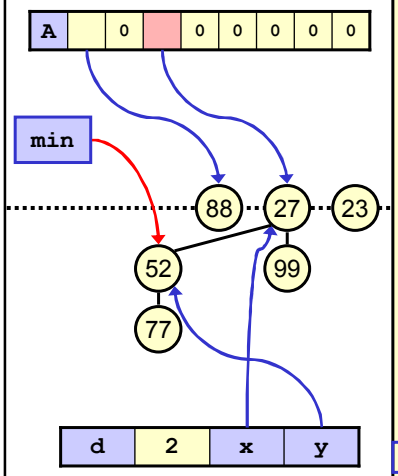

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



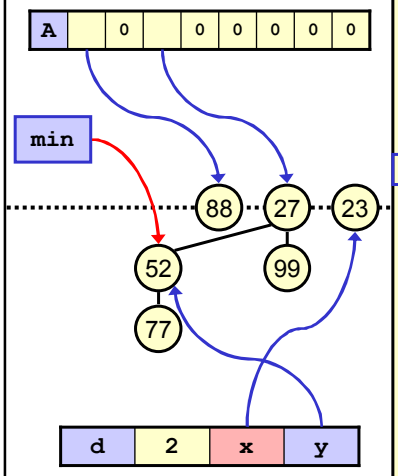

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



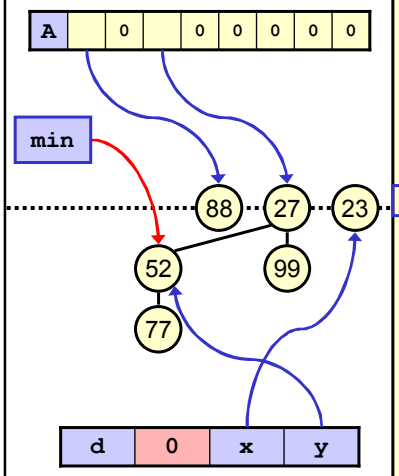

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



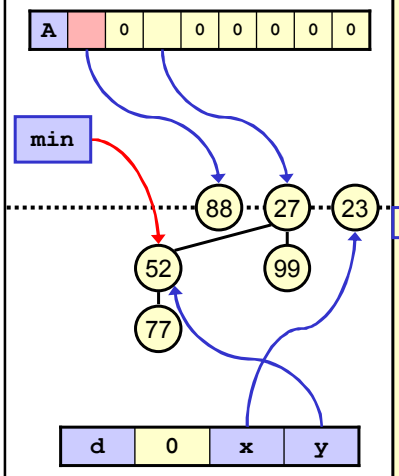

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



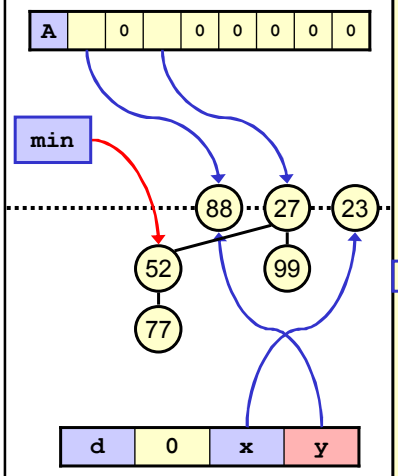

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



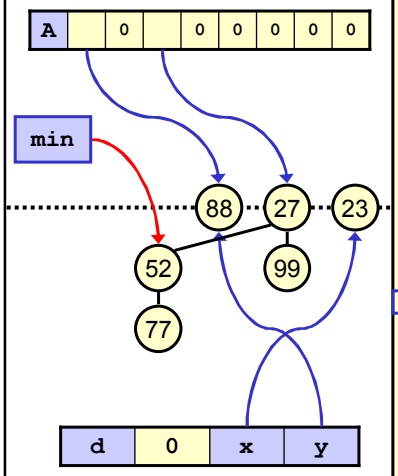

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



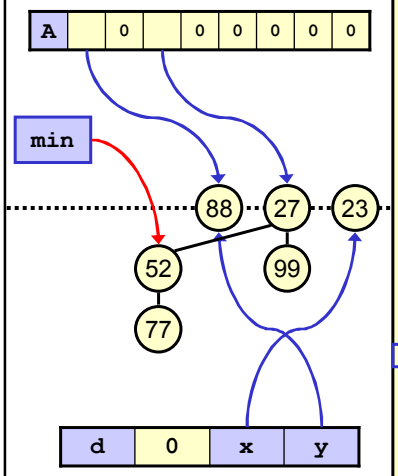

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



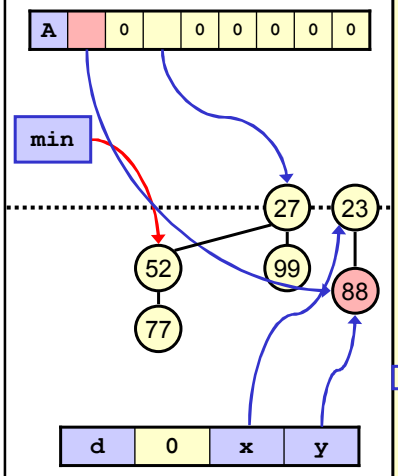

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



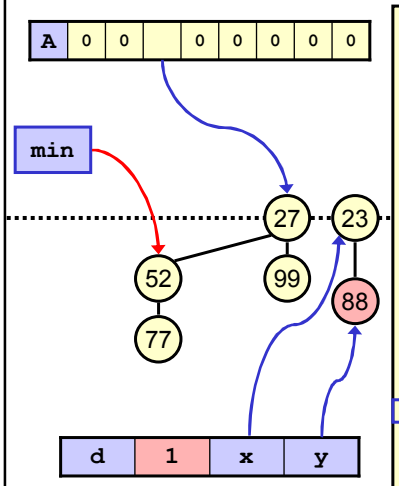

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



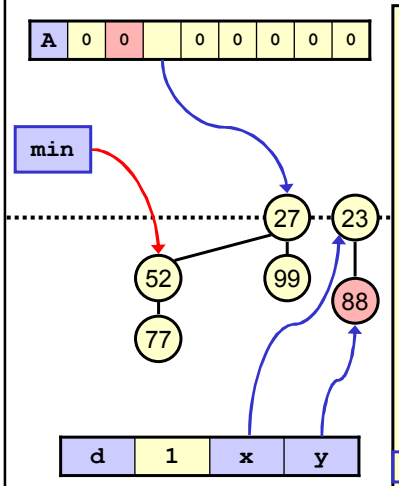

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



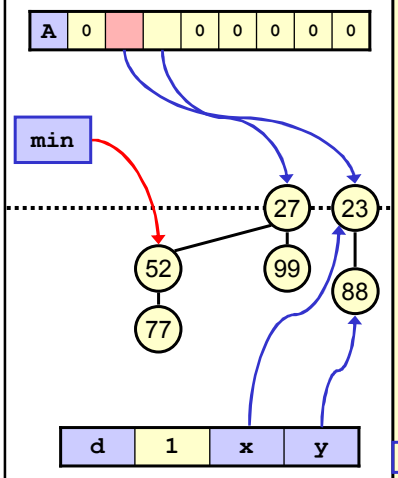

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



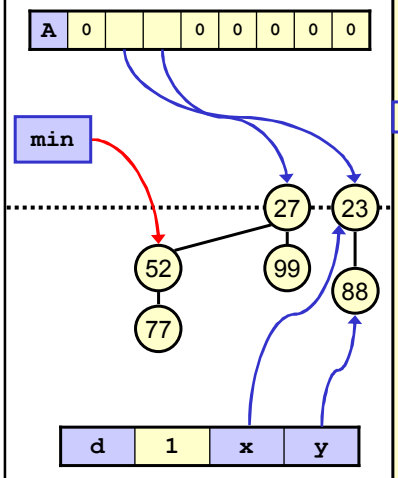

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



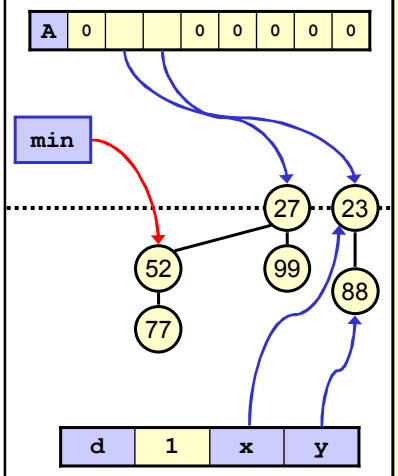

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap



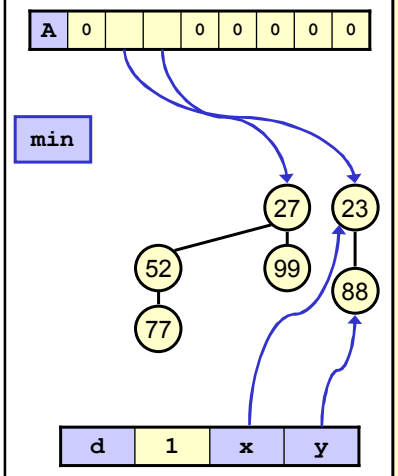

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

consolidate im Fibonacci-Heap



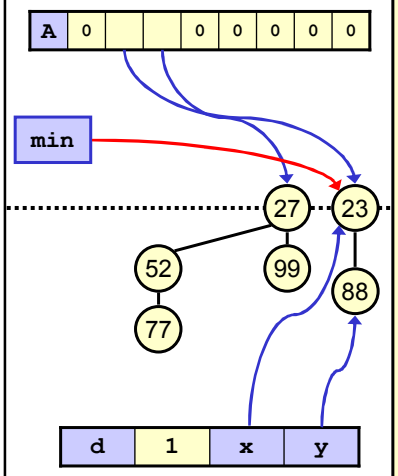

```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```


consolidate im Fibonacci-Heap




```
void consolidate()
{
  vector<NodePtr> A(max_deg() + 1);
  NodePtr v, x, y;
  Für alle Knoten v aus root_list:
  {
    x = v;
    int d = x->degree();
    while (A[d] != 0)
    {
      y = A[d];
      if (x->key() > y->key())
        std::swap(x, y)
      link(x, y);
      A[d] = 0;
      d++;
    }
    A[d] = x;
  }
  min() = 0;
```

`consolidate` im Fibonacci-Heap




```
void consolidate()
{
  […]

  for (i = 0; i < A.size(); i++)
  {
    if (A[i] != 0)
    {
      insert_into_rootlist(A[i]);
      if (min() == 0
        || A[i]->key() < min()->key())
      {
        min() = A[i];
      }
    }
  }
```


Potenzialfunktion zur amortisierten Analyse

- Der Fibonacci-Heap ***H*** enthalte ***N*** Knoten
- ***M*** sei die Anzahl markierter Knoten (`mark` = true)
- ***T*** sei die Anzahl der Bäume in der Wurzelliste
- Es existiere ein maximaler Grad ***D*** aller Knoten
- Wir wählen die Potenzialfunktion als

$$\Phi(H) = T(H) + 2\,M(H)$$

- Φ hat folgende Eigenschaften
  - Für einen leeren Heap ist $\Phi = 0$
  - Für zwei Heaps H_1 und H_2 gilt

$$\Phi(H_1 \cup H_2) = \Phi(H_1) + \Phi(H_2)$$

Amortisierte Analyse von `insert`

`insert`: neuer Knoten wird neben dem Minimum in der Wurzelliste eingefügt, der Zeiger auf das Minimum gegebenenfalls angepasst

Analyse:

- # Bäume erhöht sich: $T_{i+1} = T_i + 1$
- # Markierungen konstant: $M_{i+1} = M_i$

$\Rightarrow \Delta\Phi = \Phi_{i+1} - \Phi_i = (T_i + 1 + 2\,M_i) - (T_i - 2\,M_i) = 1$

Worst-Case-Laufzeit und amortisierte Laufzeit $\mathcal{O}(1)$

Analyse von `delete_min`

```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node: // max. O(D) Kinder!
    child.parent = 0;
    insert_into_rootlist(child);
  erase_from_rootlist(node);      // T_{i+i} = T_i + D - 1

  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();
  }
  delete node;
  number_of_nodes--;
}
```

Analyse von `delete_min`

```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node: // max. O(D) Kinder!
    child.parent = 0;
    insert_into_rootlist(child);
  erase_from_rootlist(node);      // T_{i+i} = T_i + D - 1

  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();                // Analyse von consolidate?
  }
  delete node;
  number_of_nodes--;
}
```


Analyse von `consolidate`

```
vector<NodePtr> A(max_deg() + 1);     // O(D_i)
Für alle Knoten v aus root_list:      // O(T_i)
  […]
  while (A[d] != 0)
  {
    […]
    d++;
  }
  A[d] = x;

min() = 0;                            // T_{i+1} = 0?
for (i = 0; i < A.size(); i++)        // O(D_i)
{
  if (A[i] != 0)
  {
    insert_into_rootlist(A[i]);       // T_{i+1} ≤ D_i + 1
    […]
  }
}
```

Analyse von `delete_min`

```
void delete_min()
{
  NodePtr node = min();
  if (node == 0) throw "Empty heap!";

  Für jedes Kind child von node: // O(D_i)
    child.parent = 0;
    insert_into_rootlist(child);
  erase_from_rootlist(node);     // T_{i+i} = T_i + D_i - 1

  if (node->right == node)
  {
    min() = 0;
  } else {
    min() = node->right;
    consolidate();               // O(T_i + D_i)
  }
  delete node;
  number_of_nodes--;
}
```

Analyse von `delete_min`

Die tatsächlichen Kosten c_i für **`delete_min`** sind

$$c_i = \mathcal{O}(D_i + T_i)$$

Für das Potenzial vor der Ausführung von delete_min gilt:

$$\Phi_i = T_i + 2\,M_i$$

Nach der Ausführung bleiben maximal $T_{i+1} = D_i + 1$ Wurzeln übrig. Ferner werden keine Knoten neu markiert, d.h. $M_{i+1} = M_i$. Daher gilt für Φ_{i+1}:

$$\Phi_{i+1} = T_{i+1} + 2\,M_{i+1} = D_i + 1 + 2\,M_i$$

Die amortisierten Kosten a_i betragen somit

$$\begin{aligned} a_i &= c_i + \Phi_{i+1} - \Phi_i \\ &= \mathcal{O}(D_i + T_i) + D_i + 1 + 2\,M_i - T_i - 2\,M_i \\ &= \mathcal{O}(D_i + T_i) + D_i + 1 - T_i = \mathcal{O}(D_i) + \mathcal{O}(T_i) - T_i \\ &= \mathcal{O}(D_i) \end{aligned}$$

Der letzte Schritt ist möglich, da sich Φ stets so skalieren lässt, dass die in $\mathcal{O}(T_i)$ versteckten Konstanten kompensiert werden und $\mathcal{O}(T_i) - T_i$ verschwindet.

Bestimmung des maximalen Grades D

||x|| sei die Anzahl der Knoten im Unterbaum von Knoten x inklusive x.

grad(x) sei der Grad (Anzahl der Kinder) von Knoten x.

Lemma 1:

Sei x irgendein Knoten in einem Fibonacci-Heap mit Grad k. Seien y_1, y_2, …, y_k die Kinder von x in der Reihenfolge in der sie hinzugefügt wurden. Dann gilt:

$$grad(y_1) \geq 0 \text{ und } grad(y_i) \geq i - 2 \quad \forall\, i = 2, 3, \ldots, k$$

Beweis:

$grad(y_1) \geq 0$ ist trivial. Als y_i ein Kind von x wurde, waren y_1, …, y_{i-1} bereits Kinder, d.h. grad(x) war (i – 1). `consolidate` wird y_i nur zu einem Kind von x machen, wenn $grad(x) = grad(y_i)$. Da y_i seit dem einfügen höchstens ein Kind verloren, nicht aber dazugewonnen, haben kann, gilt also: $grad(y_i) \geq i - 2$.

Fibonacci

Ein Problem im „Liber Abaci“ des Leonardo von Pisa (genannt Fibonacci) lautet:

„Jemand setzt ein Paar Kaninchen in einen Garten, der auf allen Seiten von einer Mauer umgeben ist, um herauszufinden, wie viele Kaninchen innerhalb eines Jahre geboren werden. Wenn angenommen wird, dass jeden Monat jedes Paar ein weiteres Paar erzeugt, und dass Kaninchen zwei Monate nach ihrer Geburt geschlechtsreif sind, wie viele Paare Kaninchen werden dann jedes Jahr geboren?“

Die Fibonacci-Zahlen

$$F_k = \begin{cases} 0 & \text{für } k = 0 \\ 1 & \text{für } k = 1 \\ F_{k-1} + F_{k-2} & \text{für } k \geq 2 \end{cases}$$

Lemma 2:

Für alle k ∈ ℕ gilt $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Beweis:

Induktion über k.

k = 0: 1 + F_0 = 1 + 0 = F_2

Mit der Behauptung $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$ erhält man für F_{k+2}

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i. \end{aligned}$$

■

Bestimmung des maximalen Grades D

Lemma 3:

Sei x irgendein Knoten in einem Fibonacci-Heap und grad(x) = k. Dann gilt:

$$\|x\| \geq F_{k+2} \geq \phi^k \quad \text{mit } \phi = \frac{1+\sqrt{5}}{2}$$

Beweis:

Sei s_k der minimale Wert ||z|| eines Knotens z mit grad(z) = k. Offensichtlich gilt $s_0 = 1$, $s_1 = 2$, $s_2 = 3$. Ebenso ist s_k nicht größer als ||x|| und die s_k wachsen monoton mit k.

Ferner seien $y_1, y_2, \ldots y_k$ die Kinder von x in der Reihenfolge in der sie eingefügt wurden. Um eine untere Schranke für ||x|| zu berechnen, zählen wir eine Eins für x selbst und für y_1 und wenden dann Lemma 1 an:

$$\|x\| \geq s_k = 2 + \sum_{i=2}^{k} s_{grad(y_i)} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

Lemma 1:

$grad(y_1) \geq 0$ und $grad(y_i) \geq i - 2 \quad \forall\, i = 2, 3, \ldots, k$

Bestimmung des maximalen Grades D

Mit vollständiger Induktion über k lässt sich zeigen, dass $s_k \geq F_{k+2}$ für alle k:

k = 0: $s_0 = 1 \geq 0 = F_0$

k = 1: $s_1 = 2 \geq 1 = F_1$

Mit der Induktionsbehauptung gilt:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

Ferner glauben wir (ohne Beweis), dass $F_{k+2} \geq \phi^n$. ■

Korollar:

Der maximale Grad D(N) eines jeden Knotens in einem Fibonacci-Heap mit N Knoten ist $\mathcal{O}(\log N)$.

Beweis:

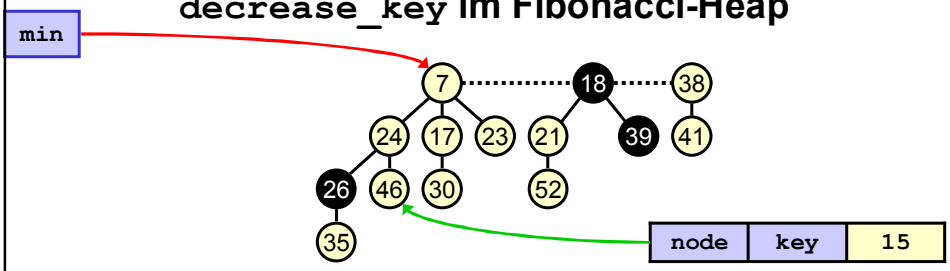
Sie x ein Knoten des Heaps und grad(x) = k. Nach Lemma 3 gilt:

$N \geq \|x\| \geq \phi^k$

Durch Logarithmieren erhält man:

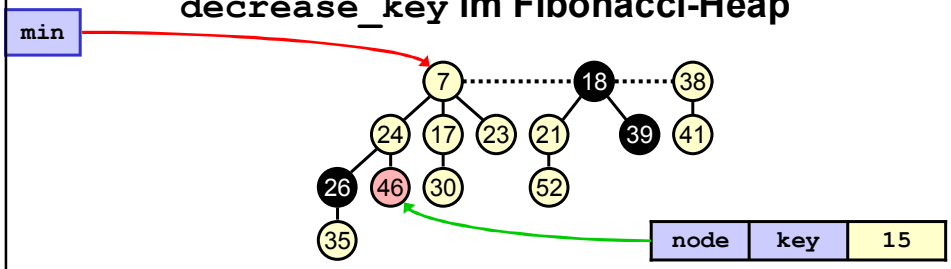
$\log_\phi N \geq k$ ■

decrease_key im Fibonacci-Heap



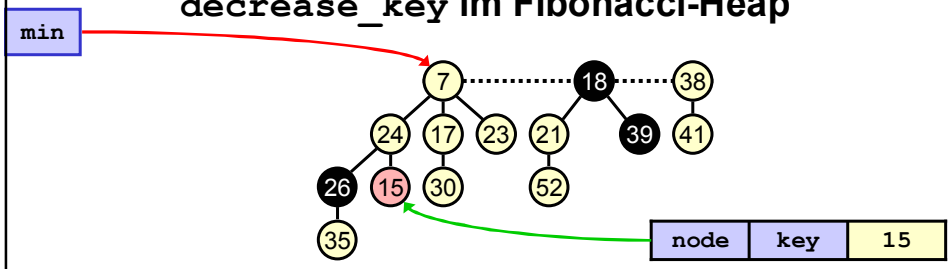

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";      // Konsistenzcheck
  node.key() = key;                                // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                   // Minimum anpassen
}
```

decrease_key im Fibonacci-Heap



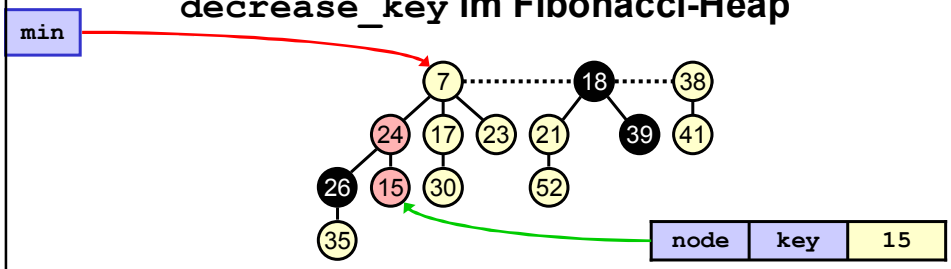

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```


decrease_key im Fibonacci-Heap



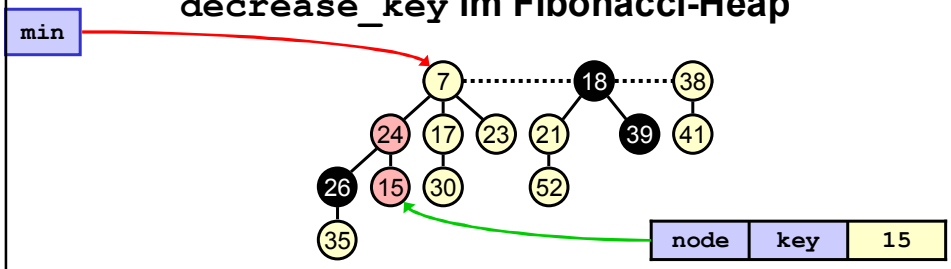

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```

decrease_key im Fibonacci-Heap



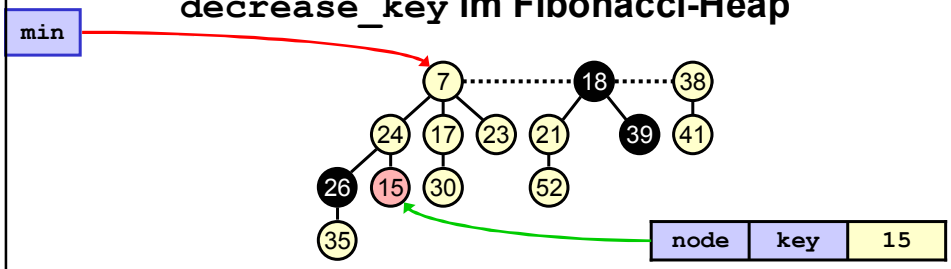

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```


decrease_key im Fibonacci-Heap



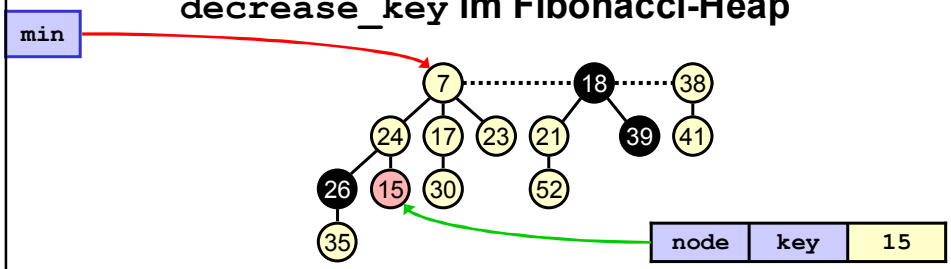

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```

decrease_key im Fibonacci-Heap



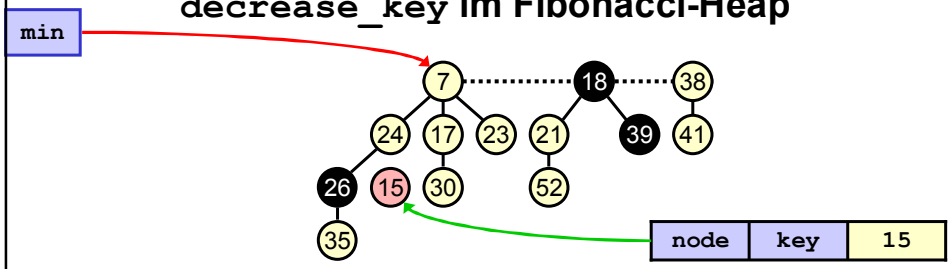

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```


decrease_key im Fibonacci-Heap



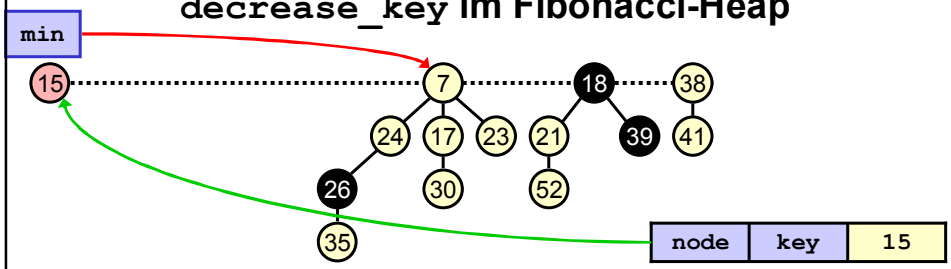

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```

decrease_key im Fibonacci-Heap



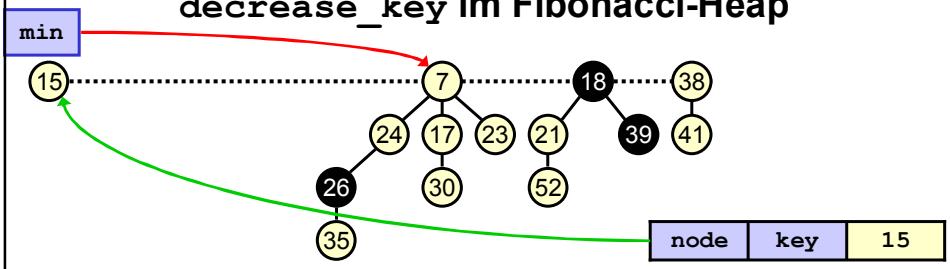

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```


decrease_key im Fibonacci-Heap



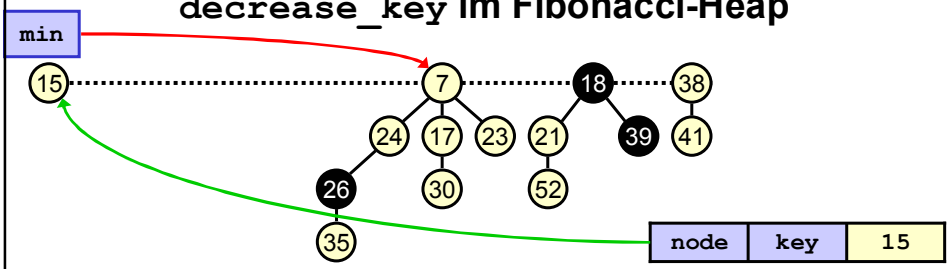

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```

decrease_key im Fibonacci-Heap

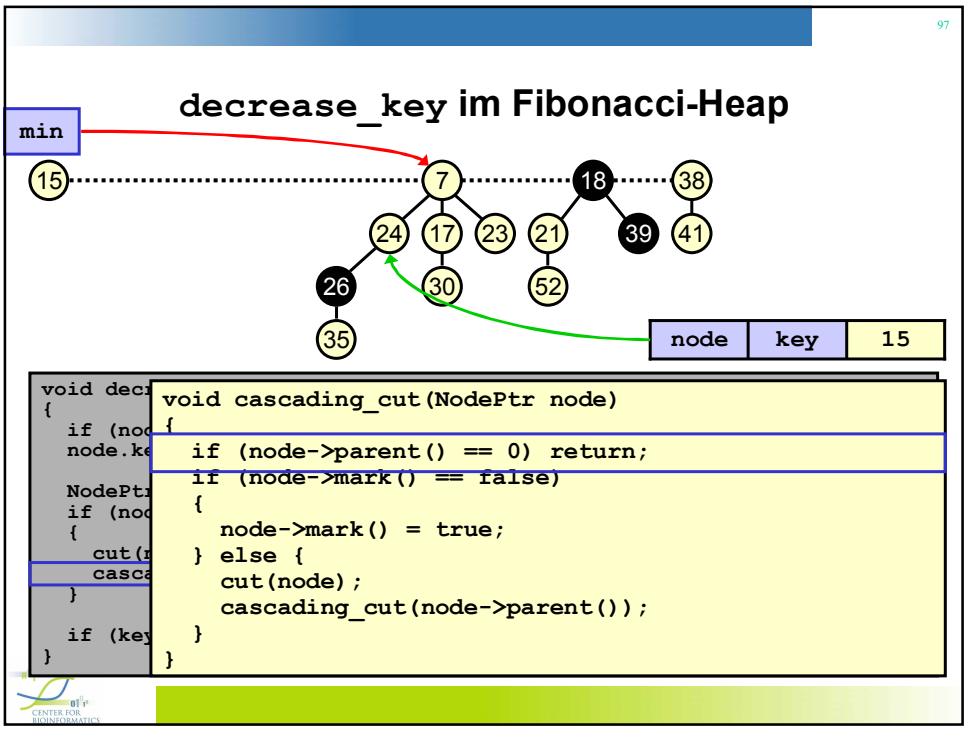


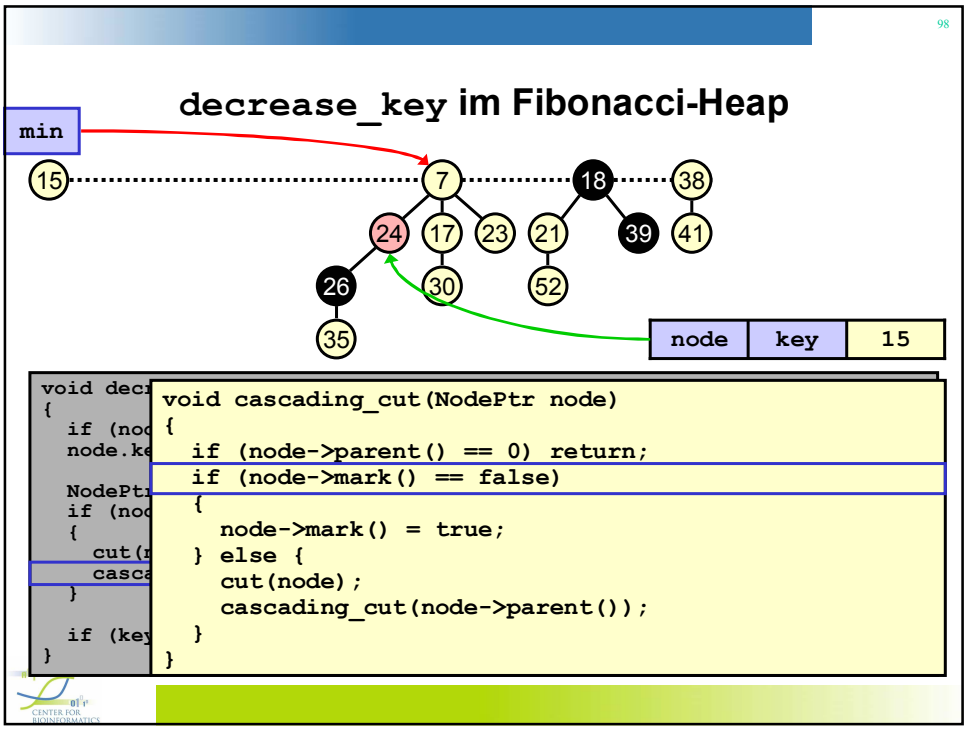

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```


decrease_key im Fibonacci-Heap

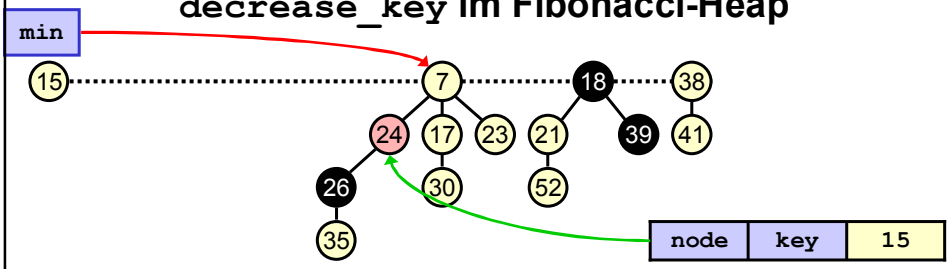



```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```


97
decrease_key im Fibonacci-Heap
min
15
7
18
38
24
17
23
21
39
41
26
30
52
35
node
key
15
void cascading_cut(NodePtr node)
{
if (node->parent() == 0) return;
if (node->mark() == false)
{
node->mark() = true;
} else {
cut(node);
cascading_cut(node->parent());
}
}

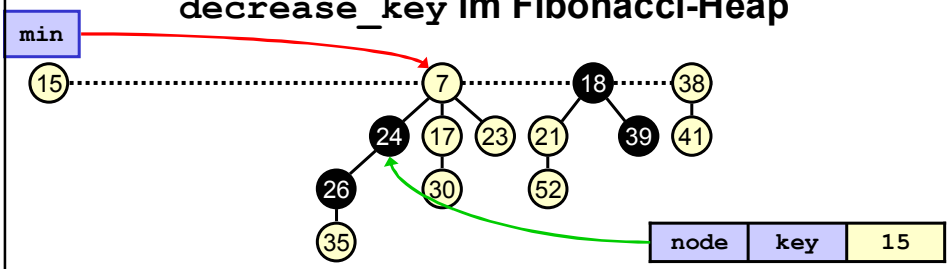

98
decrease_key im Fibonacci-Heap
min
15
7
18
38
24
17
23
21
39
41
26
30
52
35
node
key
15
void cascading_cut(NodePtr node)
{
if (node->parent() == 0) return;
if (node->mark() == false)
{
node->mark() = true;
} else {
cut(node);
cascading_cut(node->parent());
}
}

decrease_key im Fibonacci-Heap



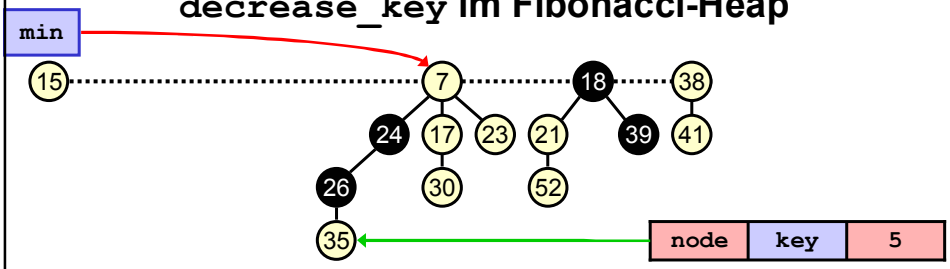

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap



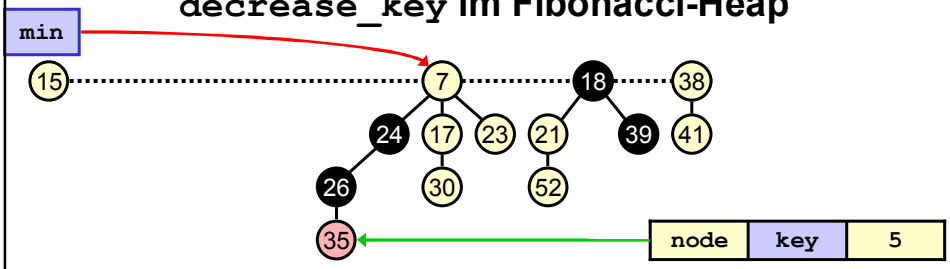

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```

decrease_key im Fibonacci-Heap



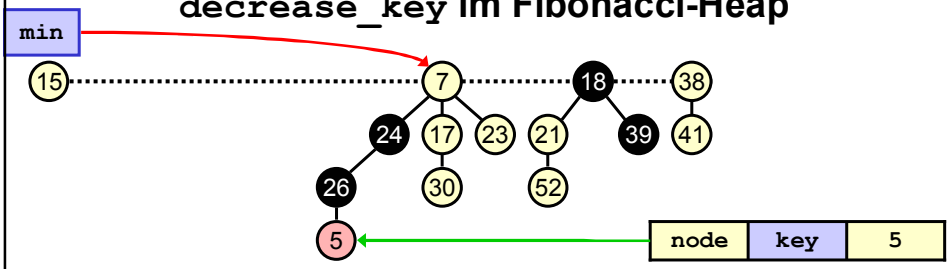

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```


decrease_key im Fibonacci-Heap



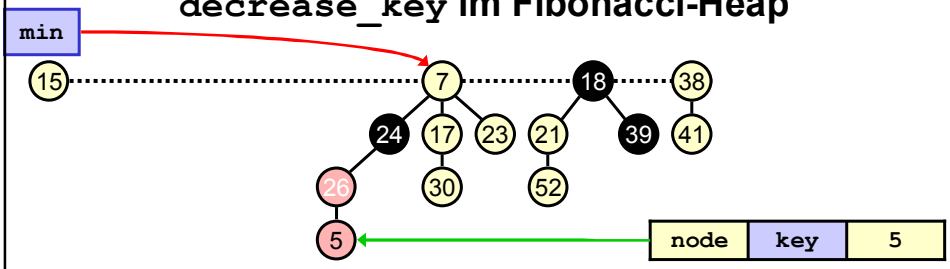

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```

decrease_key im Fibonacci-Heap



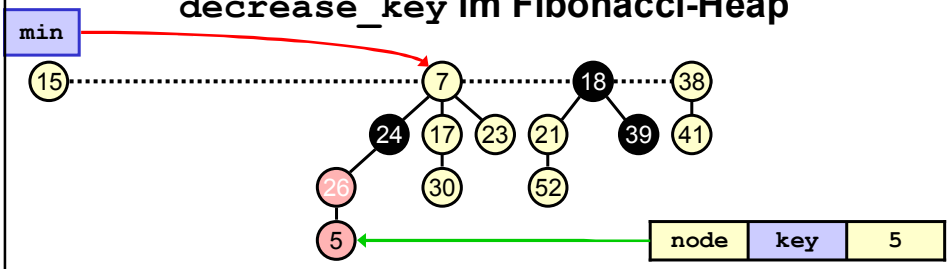

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```


decrease_key im Fibonacci-Heap



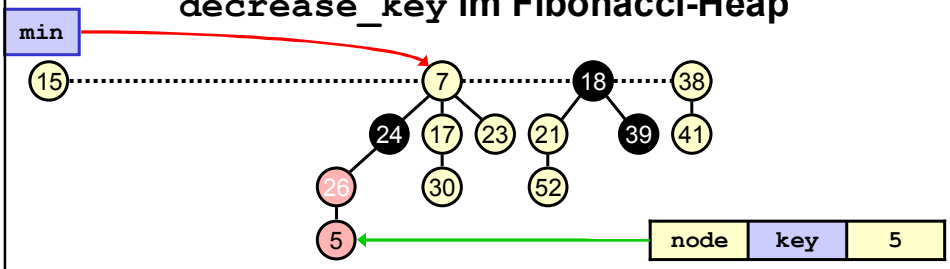

```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";       // Konsistenzcheck
  node.key() = key;                                 // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                 // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                    // Minimum anpassen
}
```

decrease_key im Fibonacci-Heap



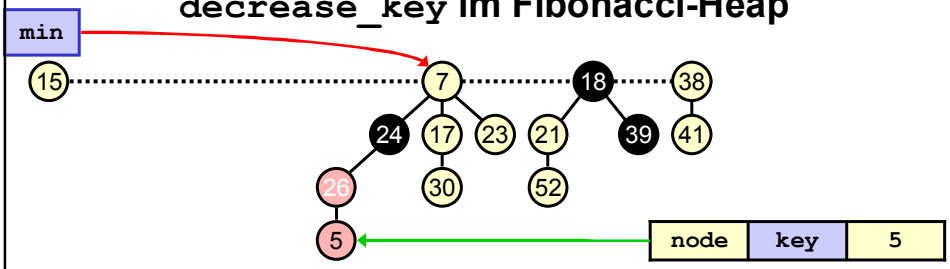

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```


decrease_key im Fibonacci-Heap



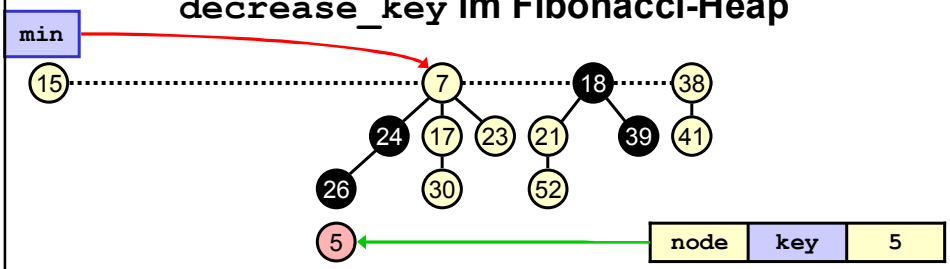

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```

decrease_key im Fibonacci-Heap



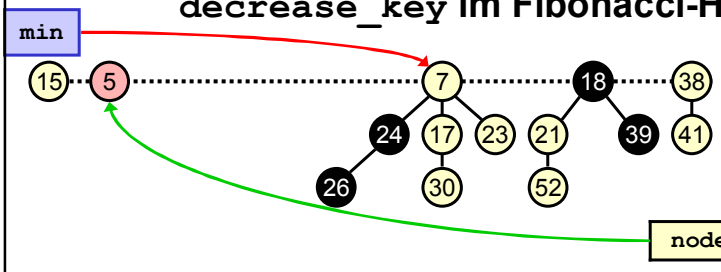

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```


decrease_key im Fibonacci-Heap



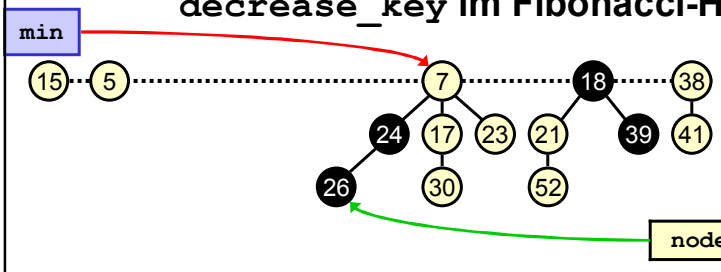

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```

decrease_key im Fibonacci-Heap



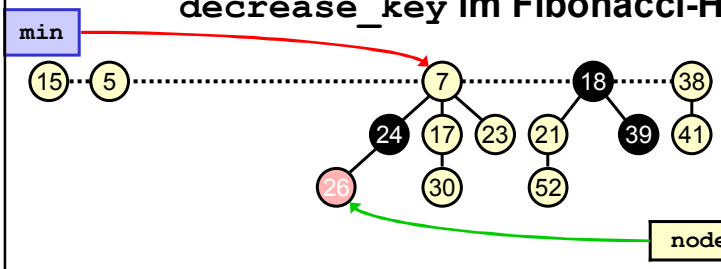

```
void cut(NodePtr node)
{
  if (node->parent() == 0) throw "Cannot cut root!";
  // Entferne Knoten aus Elterknoten
  node->parent()->remove_child(node);
  node->parent() = 0;
  // Füge in Wurzelliste ein
  insert_into_rootlist(node);
  // Entferne mögliche Markierung
  node->mark() = false;
}
```


decrease_key im Fibonacci-Heap



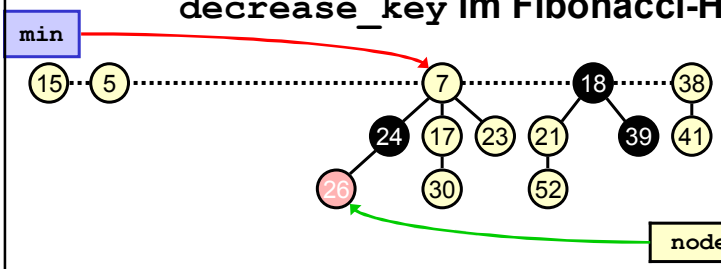

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap



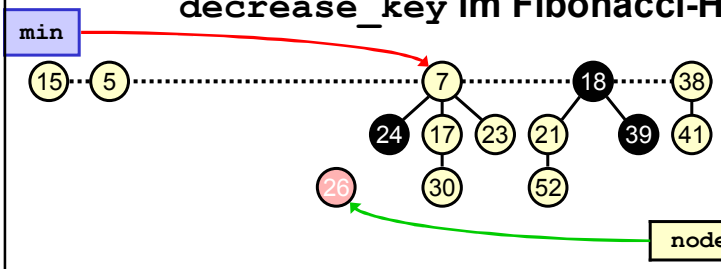

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap



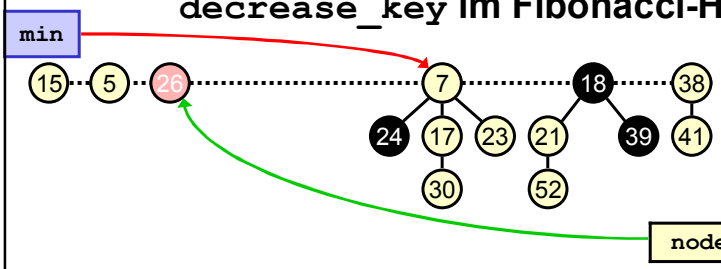

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap



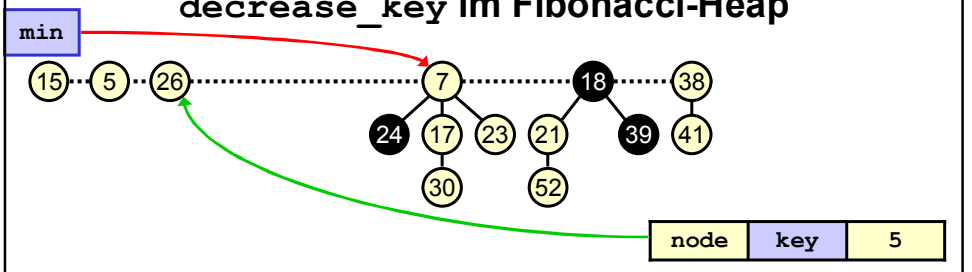

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap



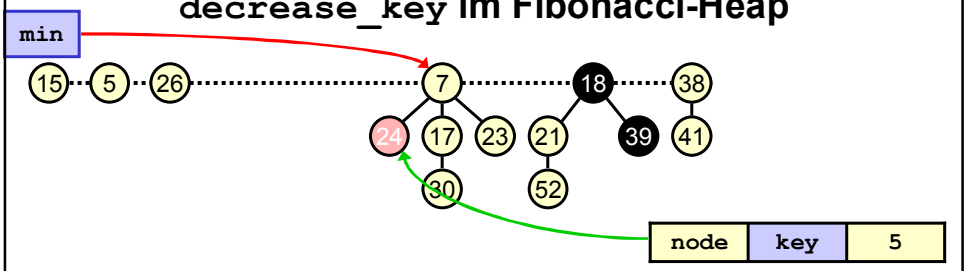

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap



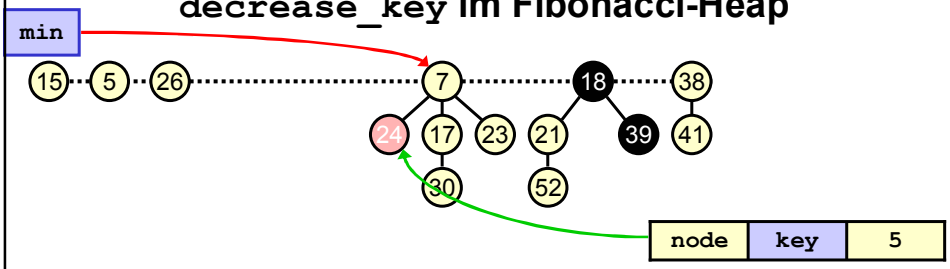

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap



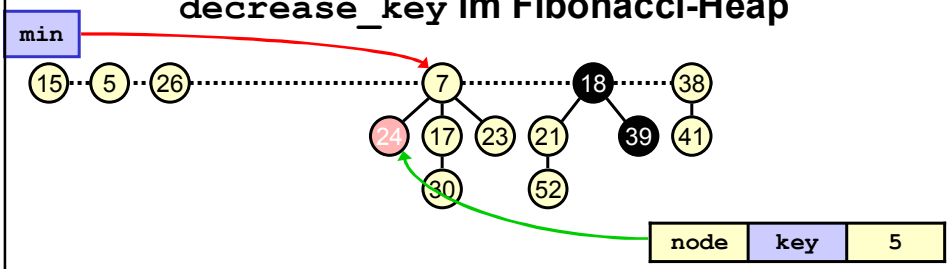

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap



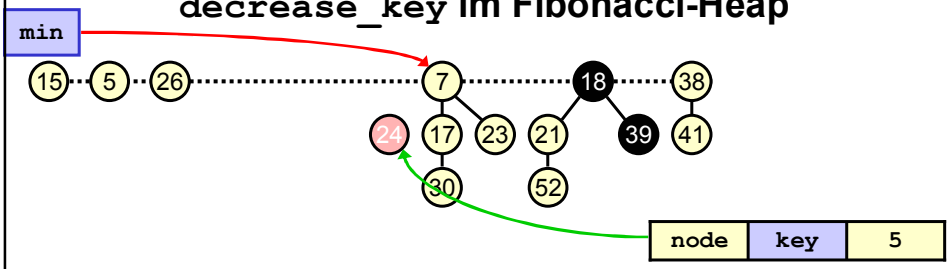

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap



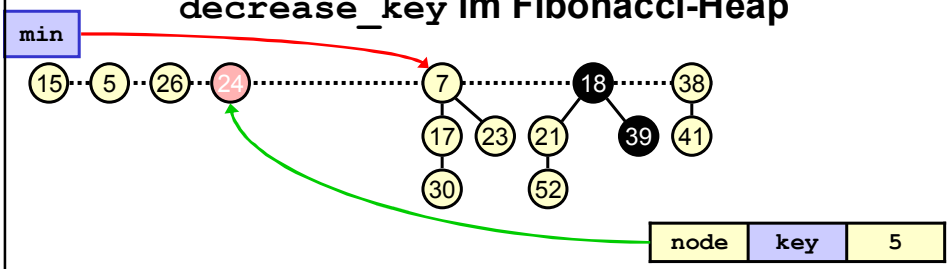

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap



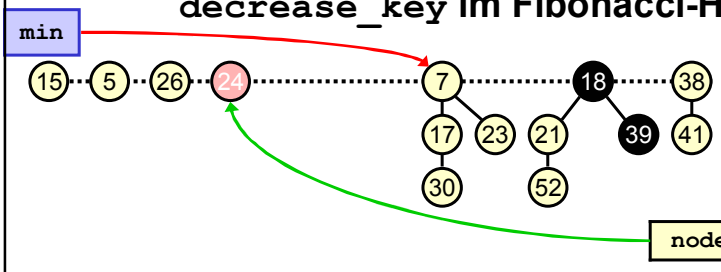

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap



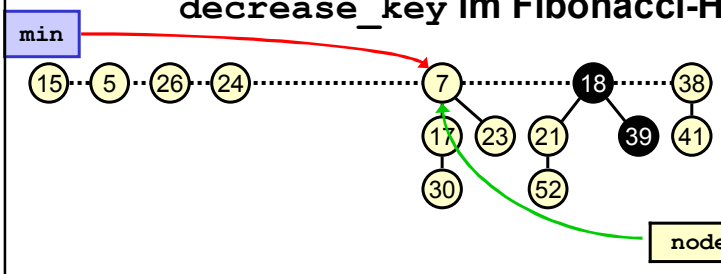

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap



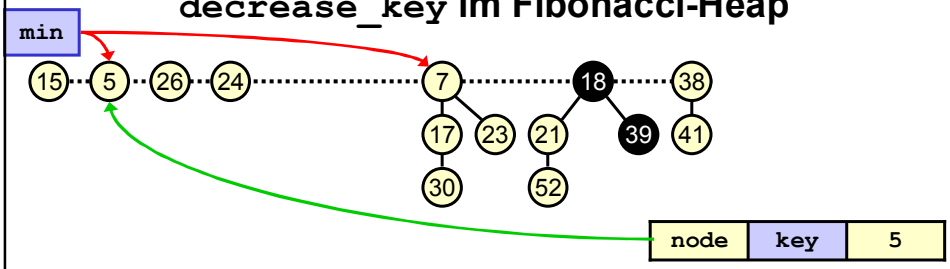

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```


decrease_key im Fibonacci-Heap




```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;
  } else {
    cut(node);
    cascading_cut(node->parent());
  }
}
```

decrease_key im Fibonacci-Heap




```
void decrease_key(NodePtr node, const Key& key)
{
  if (node.key() < key) throw "No decrease!";      // Konsistenzcheck
  node.key() = key;                                // Neuen Schlüssel zuweisen

  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {                                                // Heapeigenschaft verletzt!
    cut(node);
    cascading_cut(p);
  }

  if (key < min()) min() = node;                   // Minimum anpassen
}
```


Analyse von decrease_key

```
void decrease_key(NodePtr node, const Key& key)
{
  […]
  NodePtr p = node->parent();
  if (node->parent() != 0 && node->parent->key() > key)
  {
    cut(node);                                     // O(1)
    cascading_cut(p);                              // ???
  }
  if (key < min()) min() = node;
}

void cut(NodePtr node)
{
  […]
  node->parent()->remove_child(node); // O(1)
  […]
  insert_into_rootlist(node);                      // O(1), T erhöht
  node->mark() = false;                            // evtl. M erniedrigt
}
```

Analyse von `cascading_cut`

```
void cascading_cut(NodePtr node)
{
  if (node->parent() == 0) return;
  if (node->mark() == false)
  {
    node->mark() = true;              // M um eins erhöht!
  } else {
    cut(node);                        // O(1), T erhöht
                                      // evtl. M erniedrigt
    cascading_cut(node->parent());    // Rekursion!
  }
}
```

Unter der Annahme von r rekursiven Aufrufen von **`cascading_cut`** ist die tatsächliche Arbeit in **`cascading_cut`** $\mathcal{O}(r)$.
Jeder Aufruf von **`cascading_cut`**, mit Ausnahme des letzten, entfernt einen markierten Knoten und fügt ihn in die Wurzelliste ein. Dabei werden die Markierungen von $(r-1)$ Knoten entfernt und eventuell ein weiterer Knoten markiert.

Analyse von `decrease_key`

- Tatsächliche Arbeit in **`decrease_key`**
  $\mathcal{O}(1) + \mathcal{O}(r)$
- Potenzialänderung
  - r Bäume erzeugt: $T_{i+1} = T_i + r$
  - max. $(2 - r)$ Markierungen erzeugt: $M_{i+1} = M_i - r + 2$
    $\Rightarrow \Delta\Phi = r + 2\,(2 - r) = 4 + r$
- Amortisierte Kosten:
  $a_i = c_i + \Delta\Phi = \mathcal{O}(r) + 4 - r = \mathcal{O}(1)$
- Die letzte Umformung folgt wieder aus der Skalierbarkeit des Potenzials

Amortisierte Analyse von Heaps

Operation	Binärer Heap	Binomial-Heap	Fibonacci-Heap (amortisiert)
`insert`	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$
`find_min`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`delete_min`	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
`Decrease_key`	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$
`create`	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
`merge`	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$