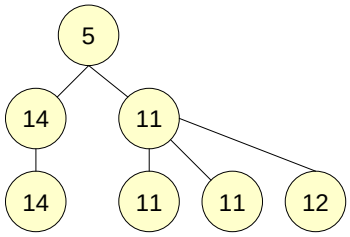
Heaps und Priority Queues

Oliver Kohlbacher

Heaps (Halden)

Def.: Ein **Heap** ist eine Datenstruktur in Form eines Baums mit der folgenden Eigenschaft: Jeder Knoten hat einen Schlüssel der extremer (kleiner/größer) oder gleich dem Schlüssel des Elterknoten ist (*Heapeigenschaft*).



Priority Queues (Vorrangwarteschlangen)

Def.: Eine Priority Queue ***PQ*** ist ein *abstrakter Datentyp*, der eine Menge von Paaren ***(key, inf)*** aus Schlüssel ***key*** und Information ***inf*** verwaltet und die folgenden Basisoperationen unterstützt:

- **`insert`** – Einfügen eines Elements
- **`find_min`** – Das kleinste Elements zurückgeben
- **`remove_min`** – Entfernen des kleinsten Elements

Häufig werden zusätzlich noch die folgenden Operationen benötigt:

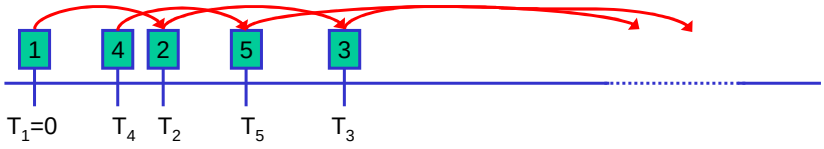
- **`merge`** – Verschmelzen zweier Warteschlangen
- **`decrease_key`** – Den Schlüssel eines Elements erniedrigen
- **`create`** – Konstruktion der PQ aus einem unsortierten Feld

Anwendungen von Warteschlangen

- Sortieren
  - Elemente einfügen
  - Elemente sortiert entnehmen
- Scheduling
  - Verteile priorisierte Prozesse auf Resource
  - Idee:
    - Wähle Prozess mit höchster Priorität, ausführen
    - „altere“ alle Prozesse in der Schlange
    - Füge den ausgeführten Prozess wieder mit Startpriorität ein
- Ereignisgesteuerte Simulationen

Ereignisgesteuerte Simulation

- Beispiel: Paketdienst



1 – Anruf beim Paketdienst ($T_1 = 0$) – erzeugt:
2 – Abholung ($T_2 = T_1 + 60$) – erzeugt:
3 – Ankunft im Sortierzentrum ($T_3 = T_2 + 120$)
4 – Noch ein Anruf ($T_4 = 40$) – erzeugt:
5 – Abholung ($T_5 = T_4 + 60$) – erzeugt:
….

Binary Heap: Interface

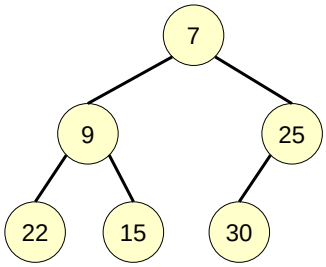
```
template <typename Key, typename Inf>
class BinaryHeap
{
  public:
  typedef std::pair<Key, Inf> Item;
  […]
  Item& operator [] (int i);     // Zugriff auf data
  void push_back(const Item& item);
  void push_down(int i);
  void push_up(int i);
  bool empty() const;            //  = (data.empty())
  size_t size() const;           //  = data.size()

  protected:
  std::vector<Item> data;     // Wurzel in data[0]!
};
```

Priority Queue: Interface

```
template <typename Key, typename Inf>
class PriorityQueue
{
  public:
  typedef std::pair<Key, Inf> Item;

  void insert(const Item& i);
  const Item& find_min() const;
  void delete_min();
  void merge(PriorityQueue& pq);
  void decrease_key(int index, const Key& key);
  void create(const vector<Item>& array);
  bool empty() const;

  BinaryHeap<Key, Inf> heap;
};
```

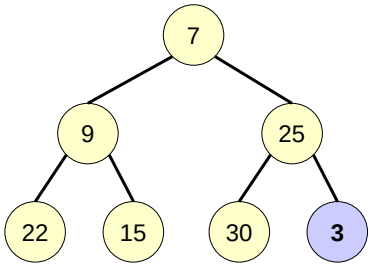

Implementierung von `insert`

```
void insert(const Item& item)
{
  // Speichere im nächsten leeren Baumelement
  heap.push_back(item);

  // Sortiere bis zur Wurzel neu
  heap.push_up(heap.size() - 1);
}
```



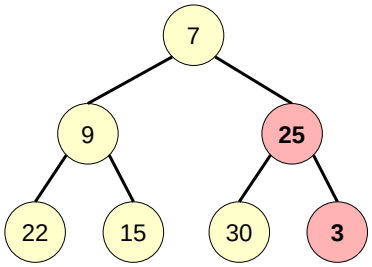
Implementierung von `insert`

```
void insert(const Item& item)
{
  // Speichere im nächsten leeren Baumelement
  heap.push_back(item);

  // Sortiere bis zur Wurzel neu
  heap.push_up(heap.size() - 1);
}
```



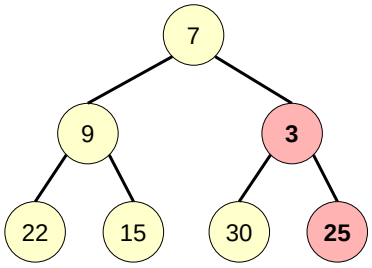
Implementierung von `insert`

```
void insert(const Item& item)
{
  // Speichere im nächsten leeren Baumelement
  heap.push_back(item);

  // Sortiere bis zur Wurzel neu
  heap.push_up(heap.size() - 1);
}
```



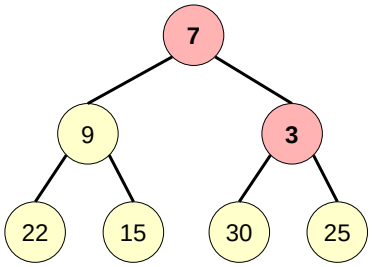
Implementierung von `insert`

```
void insert(const Item& item)
{
  // Speichere im nächsten leeren Baumelement
  heap.push_back(item);

  // Sortiere bis zur Wurzel neu
  heap.push_up(heap.size() - 1);
}
```



Implementierung von `insert`

```
void insert(const Item& item)
{
  // Speichere im nächsten leeren Baumelement
  heap.push_back(item);

  // Sortiere bis zur Wurzel neu
  heap.push_up(heap.size() - 1);
}
```



Implementierung von `insert`

```
void insert(const Item& item)
{
  // Speichere im nächsten leeren Baumelement
  heap.push_back(item);

  // Sortiere bis zur Wurzel neu
  heap.push_up(heap.size() - 1);
}
```

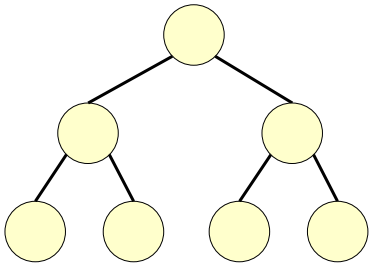
- Schlimmstenfalls müssen log N Austausch-operationen auf dem Pfad zur Wurzel durchgeführt werden (für einen Heap mit N Elementen)

  $\Rightarrow$ Laufzeit $\mathcal{O}(\log N)$
- Komplexität von **`BinaryHeap::push_back`**?

Implementierung von `create`

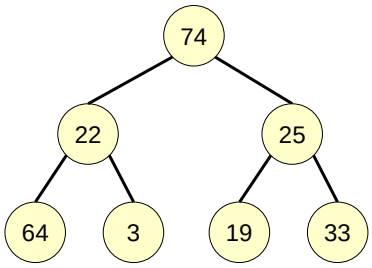
- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}(N \log N)$
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit $N = 2^h - 1$ Elementen

74	22	25	64	3	19	33



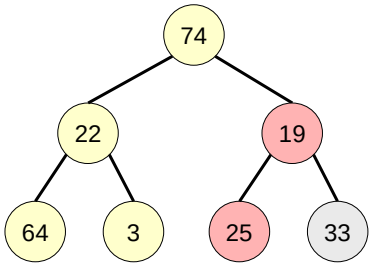
Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen



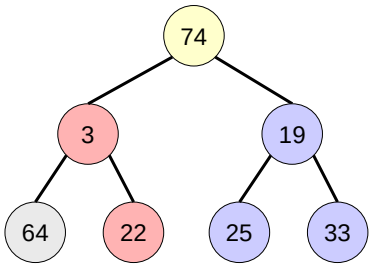
Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen



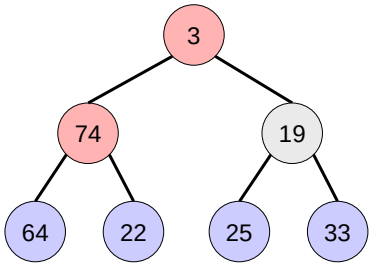
Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen



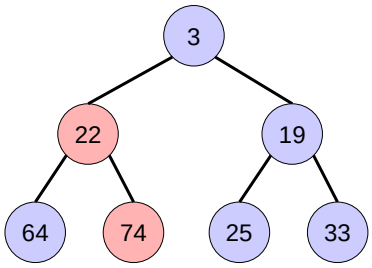
Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen



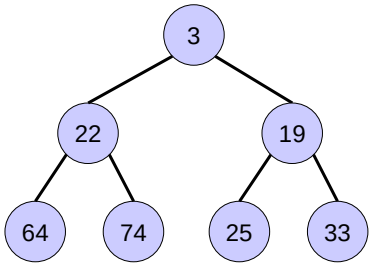
Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen



Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen



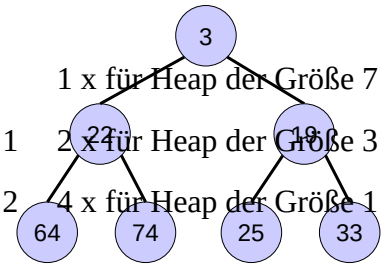
Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen
- Von unten nach oben für jeden Knoten **`push_down`**
- **`push_down`** für Höhe ***h***: max. ***2 (h – 1)*** Vergleiche

2^{h-3} x für Höhe h — 1 x für Heap der Größe 7

2^{h-2} x für Höhe h - 1 — 2 x für Heap der Größe 3

2^{h-1} x für Höhe h - 2 — 4 x für Heap der Größe 1



Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen
- Von unten nach oben für jeden Knoten **`push_down`**
- **`push_down`** für Höhe ***h***: max. ***2 (h – 1)*** Vergleiche

2^{h-3} x für Höhe h — 1 x für Heap der Größe 7

2^{h-2} x für Höhe h - 1 — 2 x für Heap der Größe 3

2^{h-1} x für Höhe h - 2 — 4 x für Heap der Größe 1

$\sum 2^{h-k}$

Implementierung von `create`

- Naiv: N * **`insert`** $\Rightarrow$ $\mathcal{O}$(N log N)
- Konstruktion ist in Linearzeit möglich!
- Konstruiere einen Heap mit N = 2^h -1 Elementen
- Von unten nach oben für jeden Knoten **`push_down`**
- **`push_down`** für Höhe ***h***: max. ***2 (h – 1)*** Vergleiche
- Es gibt 2^{h-k} Teilbäume der Höhe ***k***

Implementierung von `create`

- **`push_down`** für Höhe ***h***: max. ***2 (h – 1)*** Vergleiche
- Es gibt 2^{h-k} Teilbäume der Höhe ***k***

$$\begin{aligned}
\sum_{k=1}^{h} 2^{h-k}\,2(k-1) &= \sum_{k=1}^{h} 2^{h-k+1}(k-1) = \sum_{k=0}^{h-1} k\,2^{h-k} \\
&= 2^h \cdot 0 + 2^{h-1} \cdot 1 + \ldots + 2^1 (h-1) \\
&= \sum_{k=1}^{h} 2^k (h-k) \\
&= h \sum_{k=1}^{h} 2^k - \sum_{k=1}^{h} k 2^k \\
&= h\left(\frac{1-2^{h+1}}{1-2} - 1\right) - \left((h-1)2^{h+1} + 2\right) \\
&= \boxed{2^{h+1} - 2h - 2 \le 2N}
\end{aligned}$$

Implementierung von `create`

- Analog kann man eine Schranke für die Konstruktion von *unvollständigen* binären Heaps herleiten
- **`create`** ist in $\mathcal{O}(N)$ möglich

```
void create(const vector<Item>& array)
{
  // Implementierung: Übung!
  Kopiere Feldinhalt in Heap

  Für t := (h – 1),…, 1, 0:
    Für alle Knoten v in Tiefe t:
      push_down(v)
}
```


Implementierung von `find_min`

```
const Item& find_min() const
{
  // Konsistenzprüfung
  if (heap.empty()) throw “Heap is empty!”;

  // Das kleinste Element liegt in der Wurzel!
  return heap[0];
}
```

- Das kleinste Element liegt immer in der Wurzel des Heaps $\Rightarrow$ **`find_min`** benötigt $\mathcal{O}(1)$ Zeit
- Der Heap selbst wird nicht verändert und nur eine Referenz auf das Element zurückgegeben.

Einschub: Exceptions in C++

```
const Item& find_min() const
{
  if (heap.empty()) throw “Heap is empty!”;
  […]
}
[…]

try
{
  pq.find_min();
  […]
}
catch (const char* message)
{
  std::cerr << “Error: “ << message << std::endl;
  […]
}
```

Implementierung von `delete_min`

```
void delete_min()
{
  if (heap.empty()) throw “Heap is empty!”;

  // Letztes Element -> Wurzel, Heap verkleinern
  heap[0] = heap[heap.size() - 1];
  head.pop_back();

  // Heapeigenschaft wiederherstellen
  heap.push_down(0);
}
```

- Maximal $\log_2 N$ Austauschoperationen
- Laufzeit $\mathcal{O}(\log N)$

Implementierung von `decrease_key`

```
void decrease_key(int index, const Key& key)
{
  if (index >= heap.size()) throw “illegal index!”;
  if (heap[index].first <= key) throw “no decrease!”;

  // Letztes Element -> Wurzel, Heap verkleinern
  heap[index].first = key;

  // Heapeigenschaft wiederherstellen
  heap.push_up(index);
}
```

- Maximal log N Austauschoperationen
- Laufzeit $\mathcal{O}$(log N)

Implementierung von `merge`

```
void merge(const PriorityQueue& pq)
{
  std::vector<Item> tmp(heap.size() + pq.heap.size());

  std::copy(heap.data.begin(), heap.data.end(),
            tmp.begin());
  std::copy(pq.heap.data.begin(), pq.heap.data.end()),
            tmp.begin() + heap.data.size());

  create(tmp);
}
```

- Idee: konkateniere die beiden Heaps, dann wie **`create`**
- **`std::copy(iterator first, iterator last, iterator to)`** kopiert den Bereich **`[first, last)`** nach **`[to,…)`**
- Der Kopieralgorithmus benötigt Linearzeit, **`create`** ebenso
  $\Rightarrow$ Laufzeit $\mathcal{O}$(N)

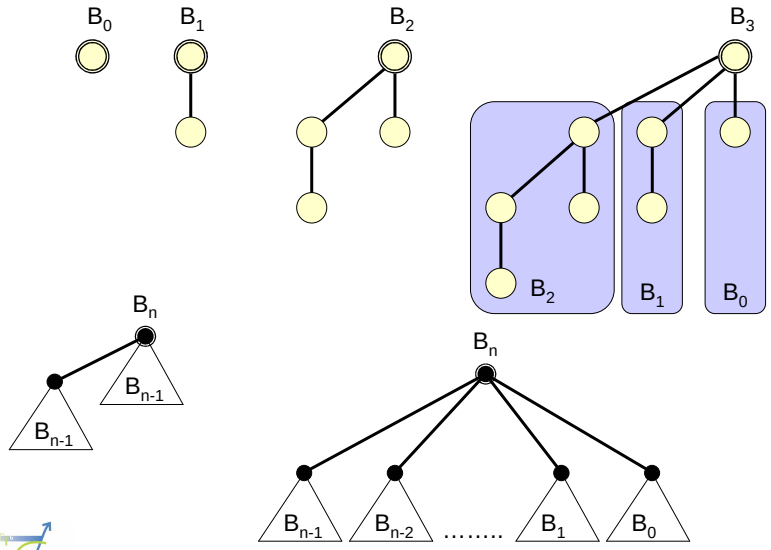
Übersicht PQ mit binärem Heap

Operation	Komplexität
`insert`	$\mathcal{O}(\log N)$
`find_min`	$\mathcal{O}(1)$
`delete_min`	$\mathcal{O}(\log N)$
`decrease_key`	$\mathcal{O}(\log N)$
`create`	$\mathcal{O}(N)$
`merge`	$\mathcal{O}(N)$

Übersicht PQ mit binärem Heap

Operation	Komplexität
`insert`	$\mathcal{O}(\log N)$
`find_min`	$\mathcal{O}(1)$
`delete_min`	$\mathcal{O}(\log N)$
`decrease_key`	$\mathcal{O}(\log N)$
`create`	$\mathcal{O}(N)$
`merge`	$\mathcal{O}(N)$

Binomialbäume



Eigenschaften von Binomialbäumen

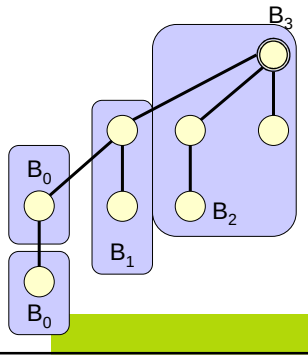
Satz: B_n hat

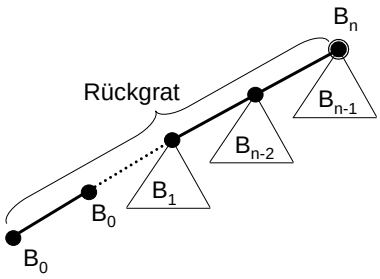
1. Höhe ***n***
2. 2^n Knoten
3. Grad ***n*** in der Wurzel
4. $\binom{n}{k}$ Knoten in Tiefe ***k***

Eigenschaften von Binomialbäumen

Satz: B_n hat

1. Höhe ***n***
2. 2^n Knoten
3. Grad ***n*** in der Wurzel
4. $\binom{n}{k}$ Knoten in Tiefe ***k***





Eigenschaften von Binomialbäumen

Satz: B_n hat

1. Höhe ***n***
2. 2^n Knoten
3. Grad ***n*** in der Wurzel
4. $\binom{n}{k}$ Knoten in Tiefe ***k***

Bew.: [2] durch Induktion

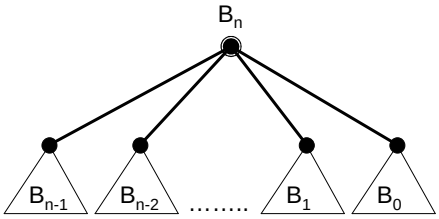
- B_0 hat einen Knoten
- $|B_n| = 2\,|B_{n-1}|$ (nach Definition der Binomialbäume)

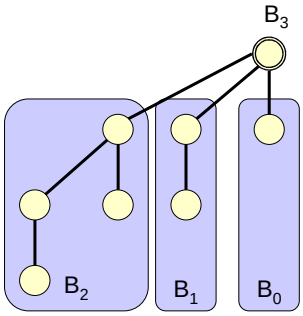
$\Rightarrow\ |B_n| = 2^n$ ■

Eigenschaften von Binomialbäumen

Satz: B_n hat

1. Höhe ***n***
2. 2^n Knoten
3. Grad ***n*** in der Wurzel
4. $\binom{n}{k}$ Knoten in Tiefe ***k***

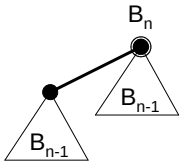




Eigenschaften von Binomialbäumen

Satz: B_n hat

1. Höhe ***n***
2. 2^n Knoten
3. Grad ***n*** in der Wurzel
4. $\binom{n}{k}$ Knoten in Tiefe ***k***



Bew.: [4] durch Induktion

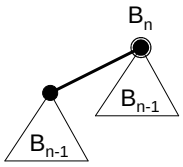
Ann.: B_n hat K_n^k Knoten in Tiefe k

n = 0: $K_0^0 = 1$, keine weiteren Knoten, also $K_0^k = 0 = \binom{0}{k}$ $\forall\, k > 0$

Eigenschaften von Binomialbäumen

Satz: B_n hat

1. Höhe ***n***
2. 2^n Knoten
3. Grad ***n*** in der Wurzel
4. $\binom{n}{k}$ Knoten in Tiefe ***k***



Bew.: [4] durch Induktion

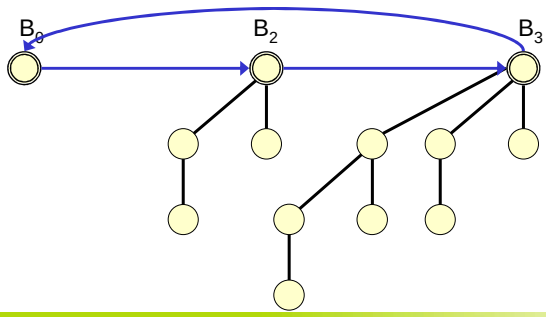
Ann.: B_n hat K_n^k Knoten in Tiefe k

n = 0: $K_0^0 = 1$, keine weiteren Knoten, also $K_0^k = 0 = \binom{0}{k}$ $\forall\, k > 0$

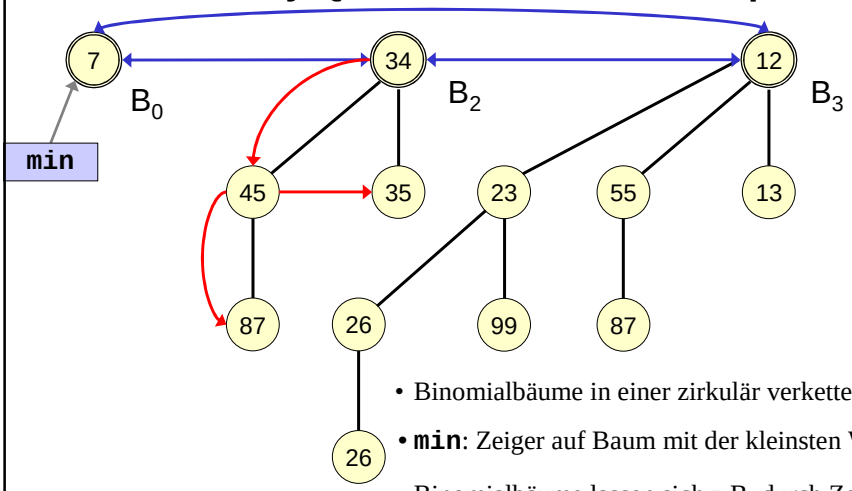
$$K_k^{n+1} = K_k^n + K_{k-1}^n = \binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k} \blacksquare$$

Binomial-Heap

Def.: Ein *Binomial-Heap* ist eine Datenstruktur die aus einem Wald von Binomialbäumen besteht. Jeder der Bäume besitzt die Heapeigenschaft und ist mit einem Index $k = 0 \ldots N$ versehen. Der Baum mit dem Index k enthält entweder 2^k Knoten oder ist leer.



Priority Queue mit Binomial-Heaps



- Binomialbäume in einer zirkulär verketteter Liste
- **`min`**: Zeiger auf Baum mit der kleinsten Wurzel
- Binomialbäume lassen sich z.B. durch Zeiger auf 1. Kind und nächstes Geschwister implementieren

Struktur von Binomial-Heaps

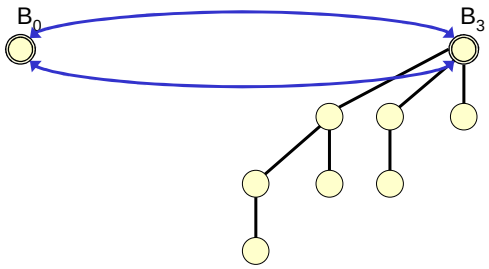
- **Idee:** Die N Elemente eines Heaps lassen sich auf eine Menge von Binomialbäumen verteilen, die der Binärdarstellung von N entspricht
- Einem gesetzten Bit *k* entspricht ein Binomialbaum B_k, der 2^k Knoten enthält
- Jeder Binomialbaum muß die Heapeigenschaft besitzen $\Rightarrow$ Wurzel ist das Minimum des Baums
- Ein solcher Heap besitzt höchstens $\log_2 N$ Bäume

Struktur von Binomial-Heaps

- **Idee:** Die N Elemente eines Heaps lassen sich auf eine Menge von Binomialbäumen verteilen, die der Binärdarstellung von N entspricht

Bsp.: $N = 9 = 1001_2$

Heap enthält zwei Bäume: B_0 (ein Knoten) und B_3 (acht Knoten)

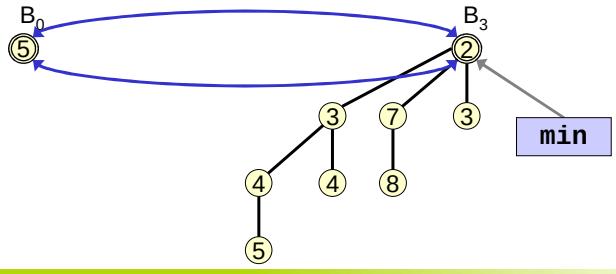


Struktur von Binomial-Heaps

- Jeder Binomialbaum muß die Heapeigenschaft besitzen
  ⇒ Wurzel ist das Minimum des Baums

Bsp.: $N = 9 = 1001_2$

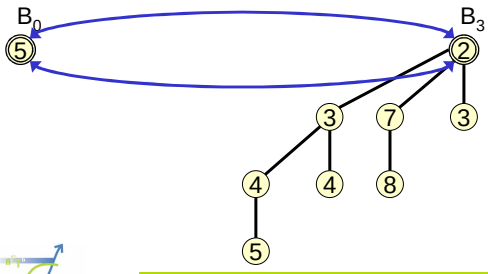
Heap enthält zwei Bäume: B_0 (ein Knoten) und B_3 (acht Knoten)

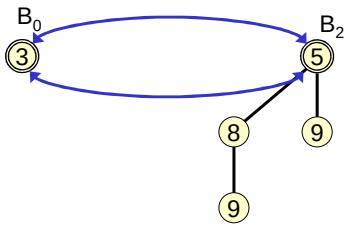


merge mit Binomial-Heaps

- Merge-Operation entspricht dem Addieren zweier Binärzahlen
- Bsp.: zwei Heaps H_1, H_2 mit $N_1 = 9 = 1001_2$, $N_2 = 5 = 101_2$

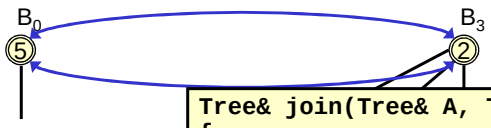
```
  1001
+  101
------
  1110
```

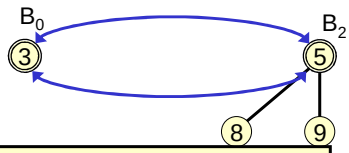




merge mit Binomial-Heaps

```
for (k = 0; k <= log2(N1 + N2); ++k){
  m = #Bk ∈ (H1 ∪ H2);
  if (m == 1)     {// behalten Bk }
  else if (m == 2){// vereinige beide Bk zu Bk+1}
  else if (m == 3){// vereinige zwei Bk zu Bk+1,
                   // behalte drittes Bk }
}
```

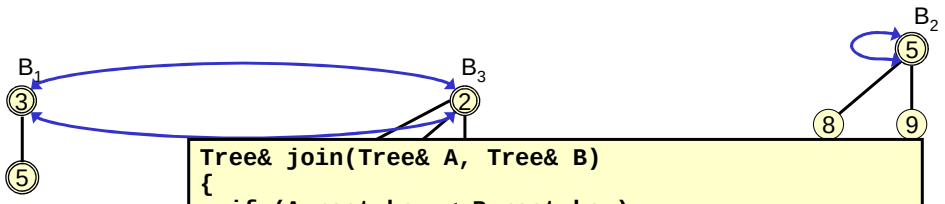





```
Tree& join(Tree& A, Tree& B)
{
  if (A.root.key < B.root.key)
    return A.root.insert_child(B.root);
  else
    return B.root.insert_child(A.root);
}
```

merge mit Binomial-Heaps

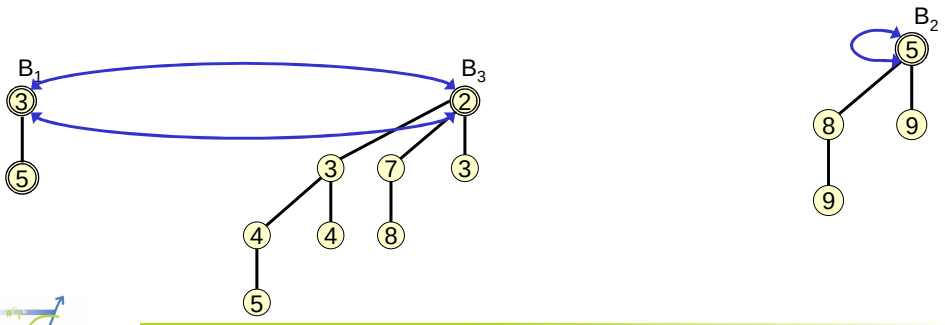
```
for (k = 0; k <= log2(N1 + N2); ++k){
  m = #Bk ∈ (H1 ∪ H2);
  if (m == 1)     {// behalten Bk }
  else if (m == 2){// vereinige beide Bk zu Bk+1}
  else if (m == 3){// vereinige zwei Bk zu Bk+1,
                   // behalte drittes Bk }
}
```




```
Tree& join(Tree& A, Tree& B)
{
  if (A.root.key < B.root.key)
    return A.root.insert_child(B.root);
  else
    return B.root.insert_child(A.root);
}
```

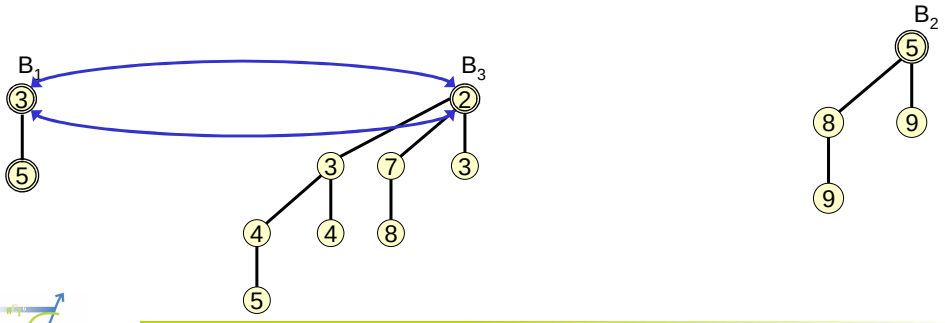

merge mit Binomial-Heaps

```
for (k = 0; k <= log2(N1 + N2); ++k){
  m = #Bk ∈ (H1 ∪ H2);
  if (m == 1)     {// behalten Bk }
  else if (m == 2){// vereinige beide Bk zu Bk+1}
  else if (m == 3){// vereinige zwei Bk zu Bk+1,
                   // behalte drittes Bk }
}
```



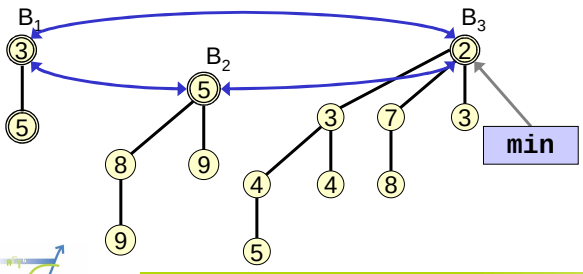
merge mit Binomial-Heaps

```
for (k = 0; k <= log2(N1 + N2); ++k){
  m = #Bk ∈ (H1 ∪ H2);
  if (m == 1)     {// behalten Bk }
  else if (m == 2){// vereinige beide Bk zu Bk+1}
  else if (m == 3){// vereinige zwei Bk zu Bk+1,
                   // behalte drittes Bk }
}
```



merge mit Binomial-Heaps

```
for (k = 0; k <= log2(N1 + N2); ++k){
  m = #Bk ∈ (H1 ∪ H2);
  if (m == 1)     {// behalten Bk }
  else if (m == 2){// vereinige beide Bk zu Bk+1}
  else if (m == 3){// vereinige zwei Bk zu Bk+1,
                   // behalte drittes Bk }
}
```



Ergebnis:

Heap mit $N_1 + N_2$ Knoten:

```
  1001
+  101
------
  1110
```

min = min(**min**(H_1), **min**(H_2))

merge mit Binomial-Heaps

```
for (k = 0; k <= log2(N1 + N2); ++k){
  m = #Bk ∈ (H1 ∪ H2);
  if (m == 1)     {// behalten Bk }
  else if (m == 2){// vereinige beide Bk zu Bk+1}
  else if (m == 3){// vereinige zwei Bk zu Bk+1,
                   // behalte drittes Bk }
}
```

Analyse:

- Join-Operation auf zwei *Bäumen* erfolgt in $\mathcal{O}(1)$
- Ergebnis-Heap hat $N = N_1 + N_2$ Knoten, max. $\log_2 N$ Bäume
- $\log_2 N$ Bäume entstanden durch max. $\log_2 N$ Join-Operationen

$\Rightarrow$ Komplexität: $\mathcal{O}(\log_2 N) = \mathcal{O}(\log_2(N_1 + N_2))$

Vergleich binärer Heap und Binomial-Heap

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

`insert` mit Binomial-Heaps

```
void insert(const Item& item)
{
  Tree trivial_tree(item);
  this->root = &merge(*this, trivial_tree);
}
```

Analyse:

- Implementierung erfolgt über **`merge`**
- **`Tree(const Item&)`** erzeugt B_0 mit **`item`** in der Wurzel

  $\Rightarrow$ Komplexität: $\mathcal{O}(\log N)$

Warum nicht $\mathcal{O}(1)$? „Übertrag“!

```
    1111111
  +       1
  ---------
   10000000
```

`create, find_min` mit Binomial-Heaps

- **`create`** naiv: schlimmstenfalls N * **`insert`** = $\mathcal{O}(N \log N)$
- Bessere Abschätzung
  - Einfügen von N Elementen entspricht binärem Zählen bis N

    $0_2 \xrightarrow{+B_0} 1_2 \xrightarrow{+B_0} 10_2 \xrightarrow{+B_0} 11_2 \xrightarrow{+B_0} 100_2 \ldots$

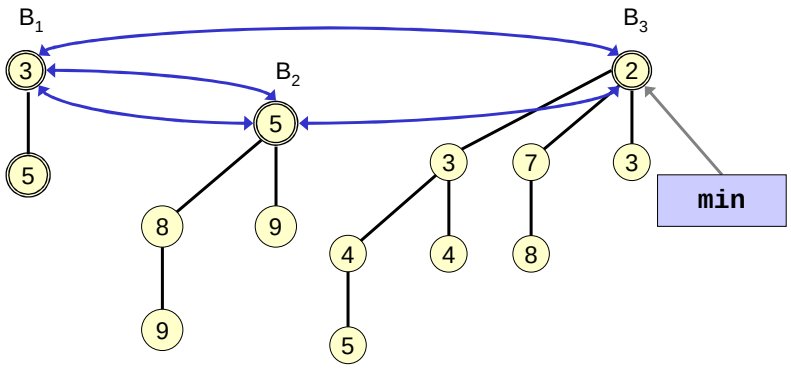
  - Übungsblatt 5: Amortisierte Analyse des Binärzählers
    - Erzeugen eines trivialen Baums (B_0): $\mathcal{O}(1)$
    - **`merge`** für zwei Binomialbäume: $\mathcal{O}(1)$
    - Bit setzen/löschen im Binärzähler $\Leftrightarrow$ elementare Merge-Operation

    $\Rightarrow$ Komplexität: $\mathcal{O}(N)$

- **`find_min`** ist trivial: wir haben den **`min`**-Pointer!

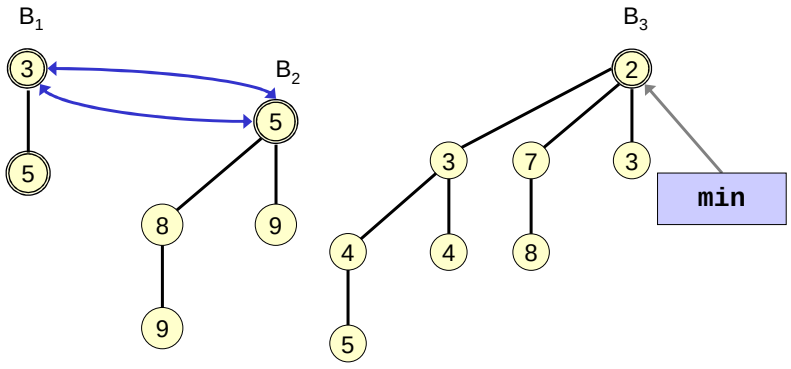
  $\Rightarrow$ Komplexität: $\mathcal{O}(1)$

`delete_min` mit Binomial-Heaps



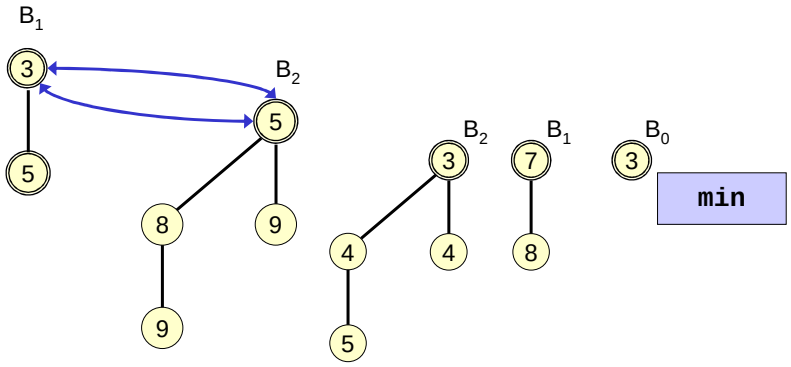
1. Entferne **`min`** aus der Wurzelliste

`delete_min` mit Binomial-Heaps



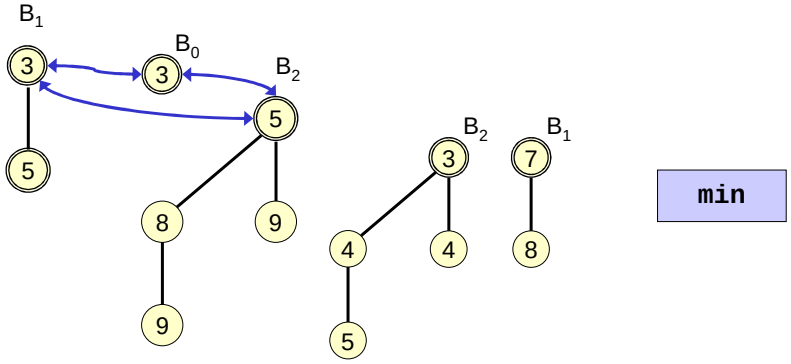
2. Lösche **`min`**

`delete_min` mit Binomial-Heaps



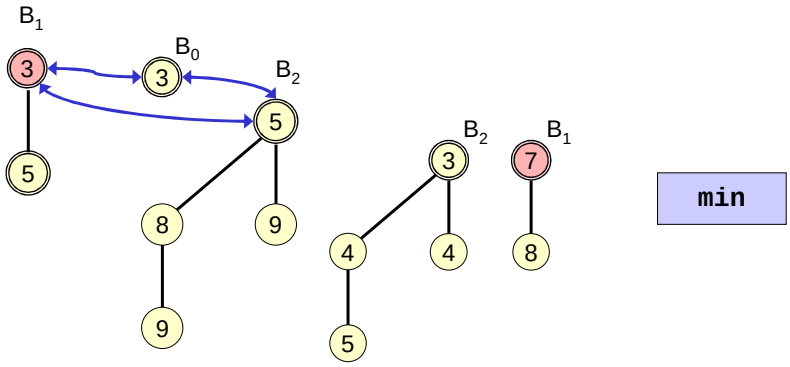
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



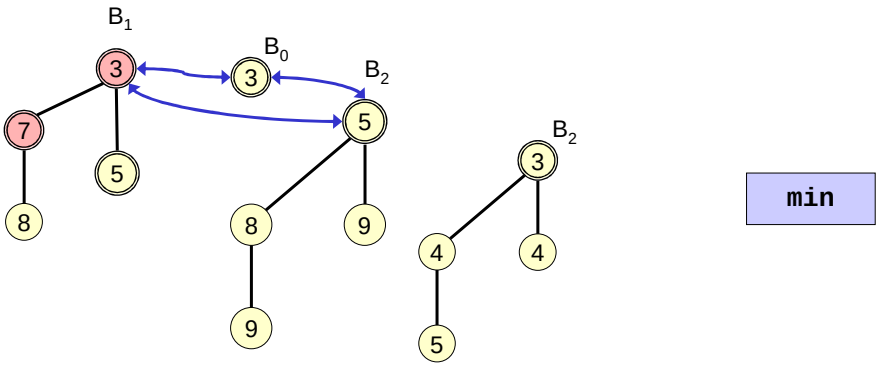
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



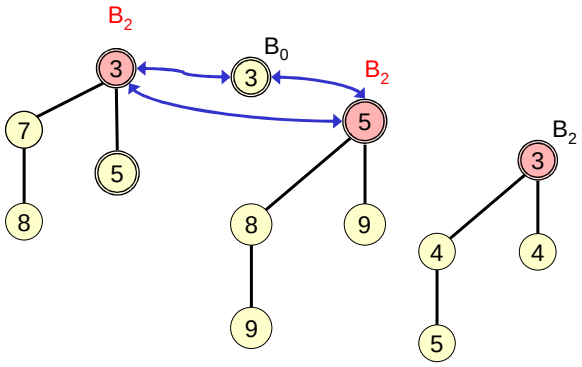
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



3. Füge Kinder von **`min`** ein

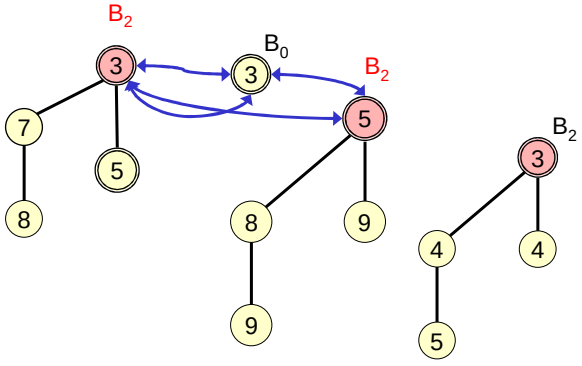
`delete_min` mit Binomial-Heaps



`min`

3. Füge Kinder von **`min`** ein

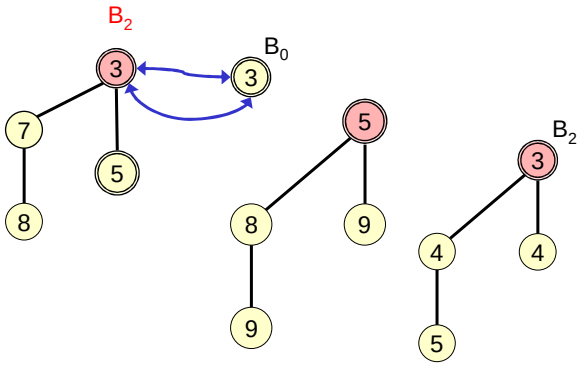
`delete_min` mit Binomial-Heaps



`min`

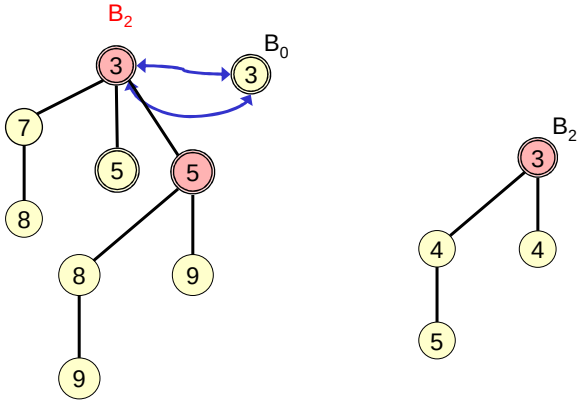
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



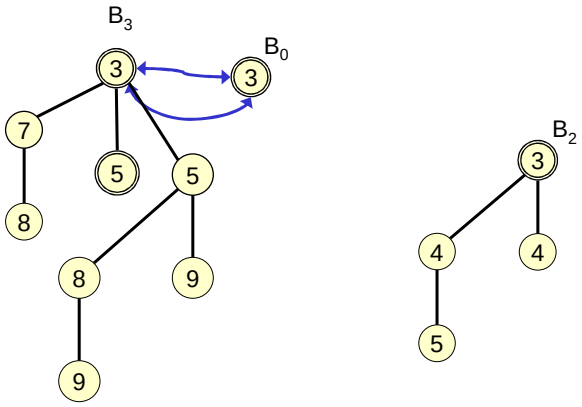
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



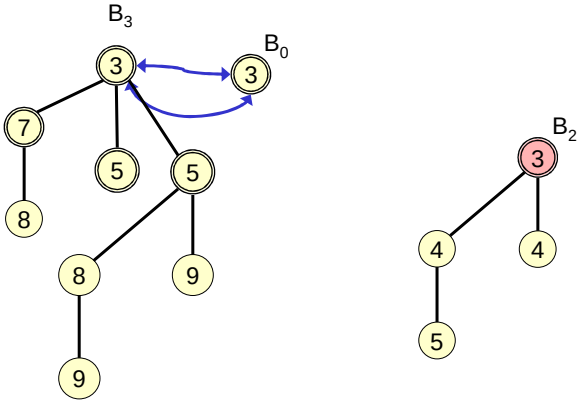
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



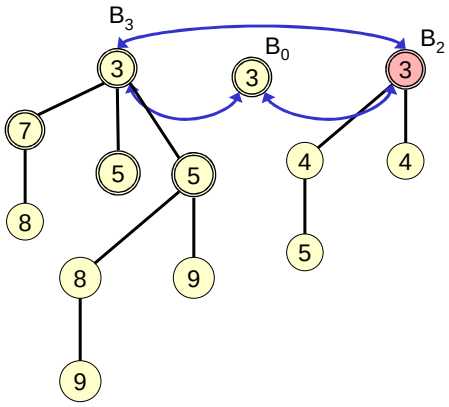
3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



3. Füge Kinder von **`min`** ein

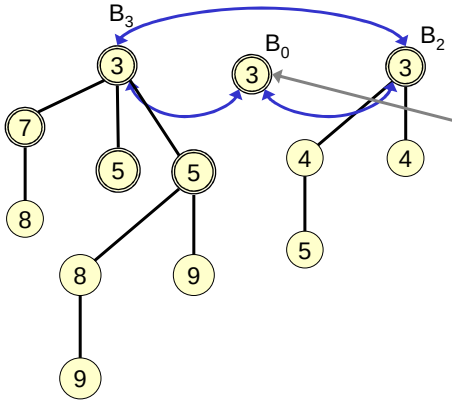
`delete_min` mit Binomial-Heaps



`min`

3. Füge Kinder von **`min`** ein

`delete_min` mit Binomial-Heaps



`min`

4. Neues **`min`** Element suchen

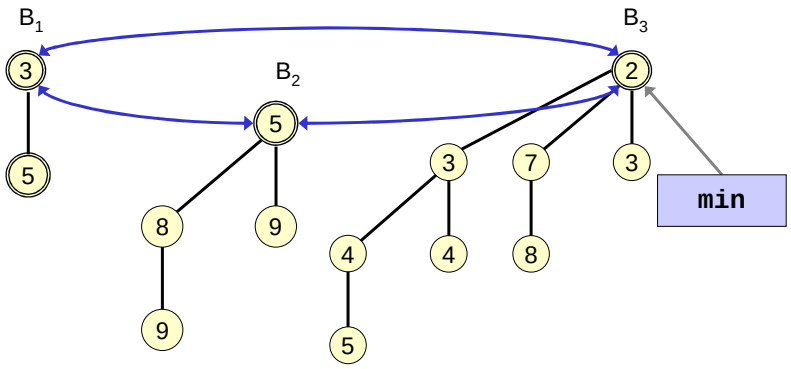
delete_min mit Binomial-Heaps

```
void delete_min()
{
  // 1. Entferne min aus Wurzelliste
  root_list.erase(min);

  // 2. Lösche min
  Tree* child = min->first_child;
  delete min;

  // 3. Füge Kinder ein
  while (child != 0)
  {
    Tree* next_child = child->next;
    merge(*child);
    child = next_child;
  }

  // 4. Finde neues Minimum
  min = minimum(); // Geht über Wurzelliste, findet minimum
}
```


delete_min mit Binomial-Heaps

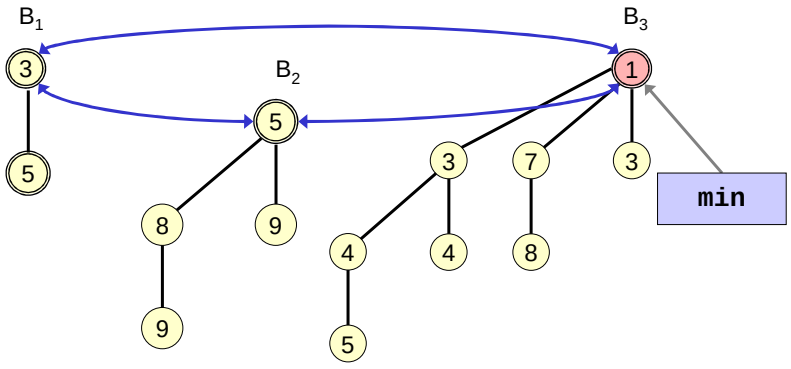
```
void delete_min()
{
  // 1. Entferne min aus Wurzelliste   O(1)
  root_list.erase(min);

  // 2. Lösche min                     O(1)
  Tree* child = min->first_child;
´ delete min;

  // 3. Füge Kinder ein                O(log N)
  while (child != 0)
  {
    Tree* next_child = child->next;
    merge(*child);
    child = next_child;
  }

  // 4. Finde neues Minimum            O(log N)
  min = minimum(); // Geht über Wurzelliste, findet minimum
}
```

decrease_key mit Binomial-Heaps



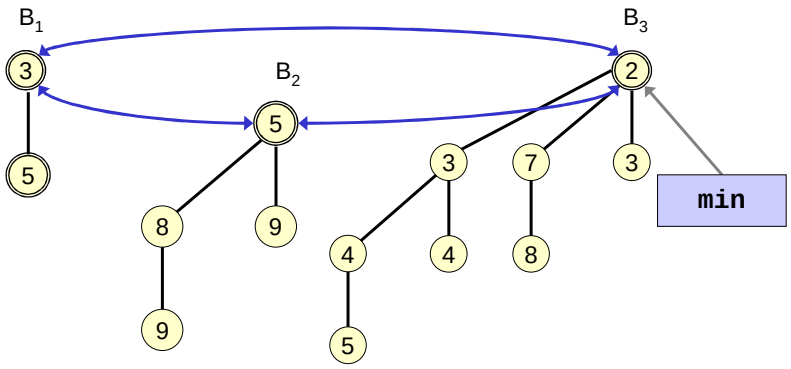
1. Fall: **decrease_key** auf **min**

decrease_key mit Binomial-Heaps



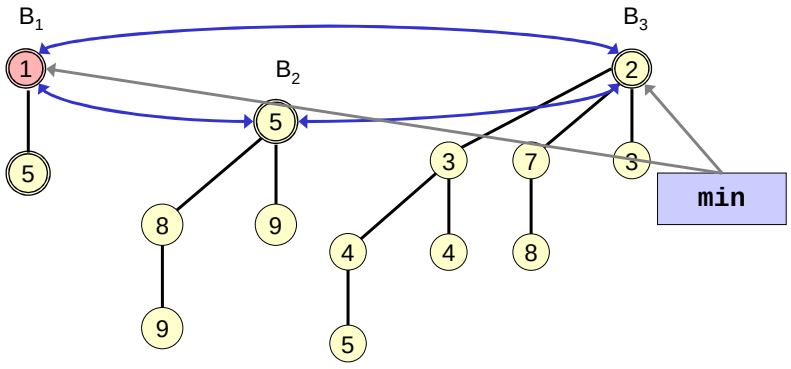
1. Fall: **decrease_key** auf **min - $\mathcal{O}(1)$**

decrease_key mit Binomial-Heaps



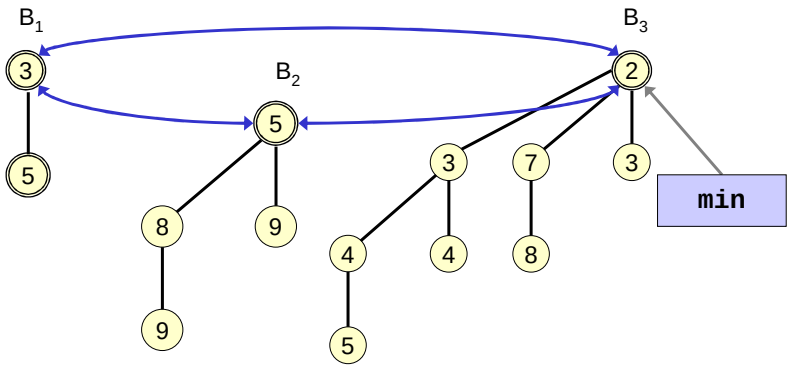
2. Fall: **decrease_key** auf bel. Wurzel

decrease_key mit Binomial-Heaps



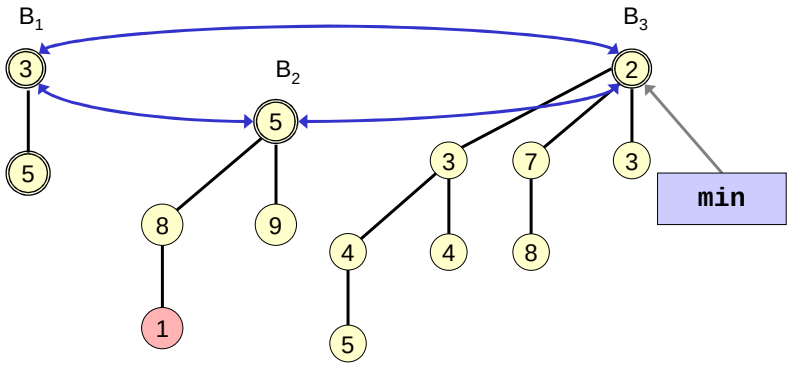
2. Fall: **decrease_key** auf bel. Wurzel - $\mathcal{O}(1)$

decrease_key mit Binomial-Heaps



3. Fall: **decrease_key** auf bel. Element

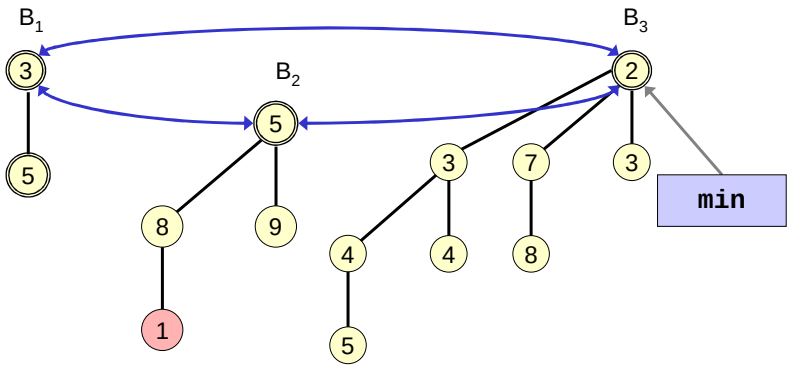
decrease_key mit Binomial-Heaps



3. Fall: **decrease_key** auf bel. Element

77

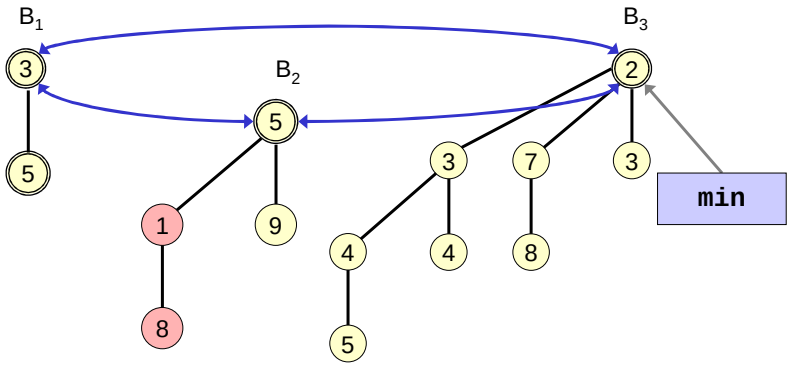
decrease_key mit Binomial-Heaps



3. Fall: **decrease_key** auf bel. Element

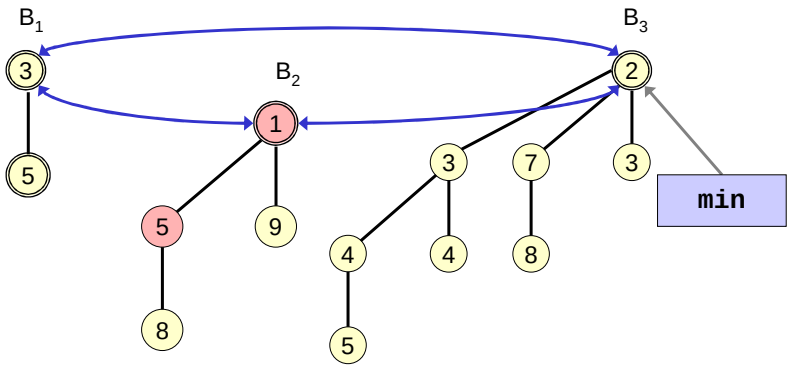
78

decrease_key mit Binomial-Heaps



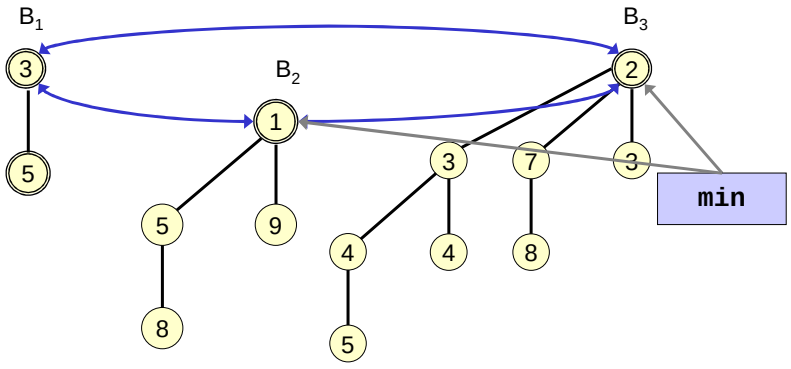
3. Fall: **decrease_key** auf bel. Element

decrease_key mit Binomial-Heaps



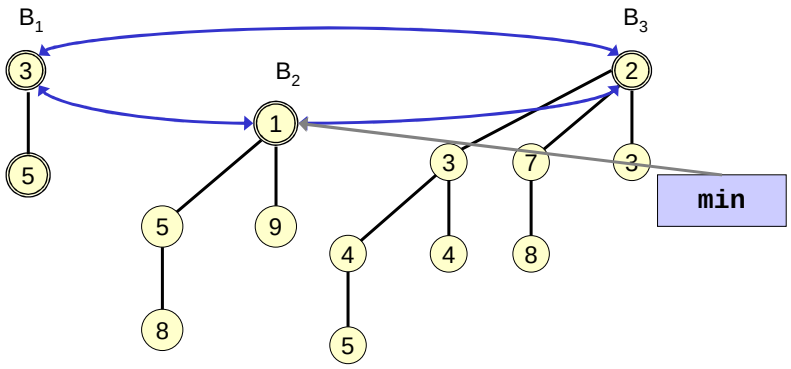
3. Fall: **decrease_key** auf bel. Element

decrease_key mit Binomial-Heaps



3. Fall: **decrease_key** auf bel. Element

decrease_key mit Binomial-Heaps



3. Fall: **`decrease_key`** auf bel. Element - $\mathcal{O}(\log N)$

Vergleich binärer Heap und Binomial-Heap

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`	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$

Die Heap-Algorithmen der STL

```
vector<int> my_heap(20);
[…]
std::make_heap(my_heap.begin(), my_heap.end());
[…]
// Hinzufügen
my_heap.push_back(item);
std::push_heap(my_heap.begin(), my_heap.end());
```

- Die STL definiert die Standardoperationen auf binären Heaps im Header **<algorithm>**
- Die Algorithmen arbeiten auf beliebigen Containern mit wahlfreiemn Zugriff – sie benötigen nur entsprechende Iteratoren

Die STL-Klasse `priority_queue`

```
// Aus STL header <queue>:
template <typename T, typename Container,
          typename Compare>
class priority_queue
{
  public:
  […]
  void push(const value_type& item); // insert
  const_reference top() const;       // find_min
  void pop();                        // insert
};
```

- **priority_queue** ist im Header **<queue>** definiert
- Die Klasse ist ein Adapter, d.h sie setzt auf einen bereits implementierten Container auf (**vector<T>**)
- Die Implementierung basiert auf binären Heaps
- Durch Wahl von **Compare** wird die Ordnung bestimmt