FHV

Algorithmen und Objektorientierte Programmierung II

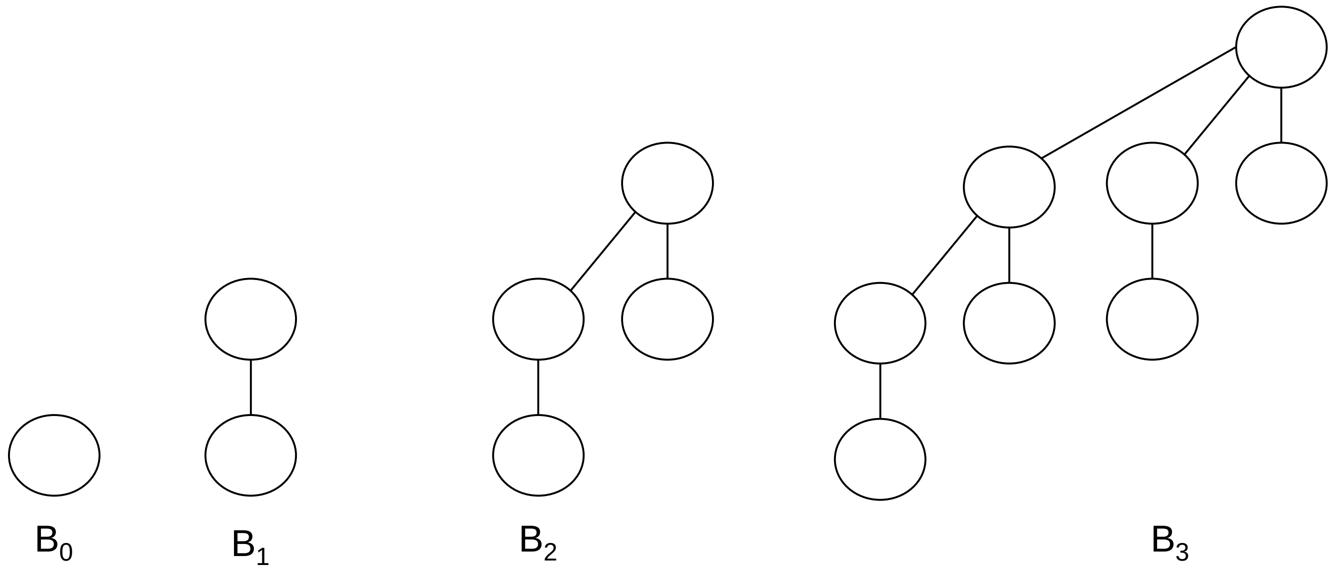
Binomialer Heap und Fibonacci Heap

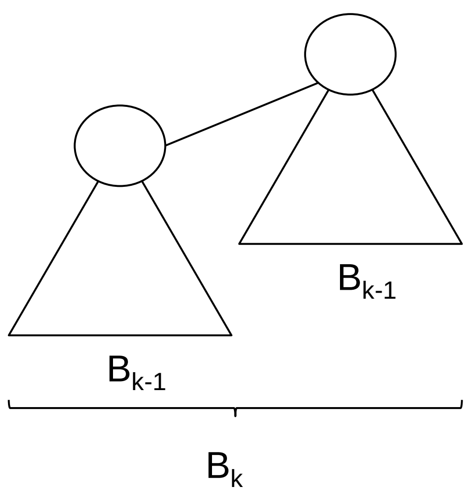
Motivation

Operation	Binärer Heap	Binomialer Heap	Fibonacci-Heap
Make-Heap	O(1)	O(1)	O(1)
Insert	O(log n)	O(log n)	O(1)
Minimum	O(1)	O(log n)	O(1)
Extract-Min	O(log n)	O(log n)	O(log n)*
Decrease-Key/Increase-Key	O(log n)	O(log n)	O(1)*
Delete	O(log n)	O(log n)	O(log n)*
Union	O(n)	O(log n)	O(1)

(*)Amortisierte Kosten ([1])Bei bekannter Position, sonst O(log n)*

Binomialer Baum





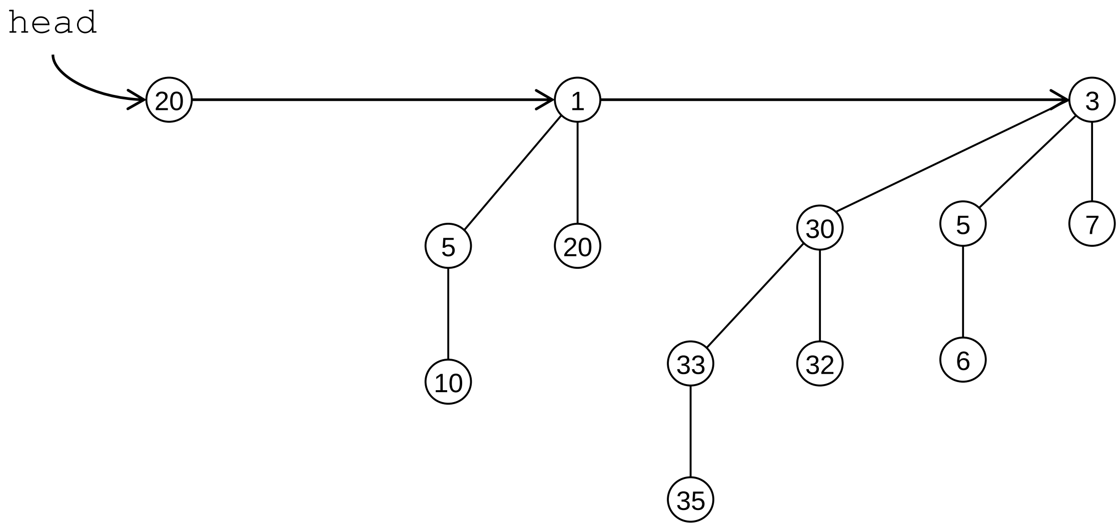
Eigenschaften B_k:

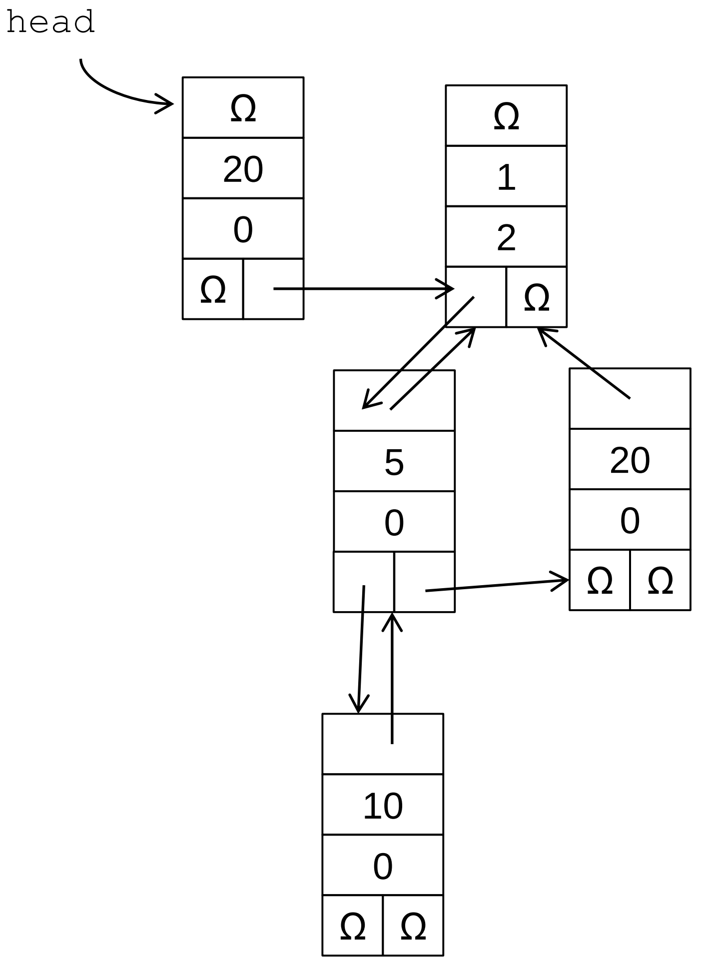
1. Ein binomialer Baum ist ein geordneter Baum
2. Besteht aus 2^k Knoten
3. Die Höhe des Baumes ist k
4. Wurzel hat Grad k
    1. Kinder der Wurzel haben von links nach rechts den Grad $k-1$, $k-2$, ... 0
5. 4 gilt rekursiv

Ein Binomialer Heap

- ist eine Menge von binomialen Bäumen
- jeder Binomiale Baum erfüllt die Heap-Bedingung
- Für `k >= 0` gibt es höchstens einen binomialen Baum, dessen Wurzel den Grad `k` besitzt

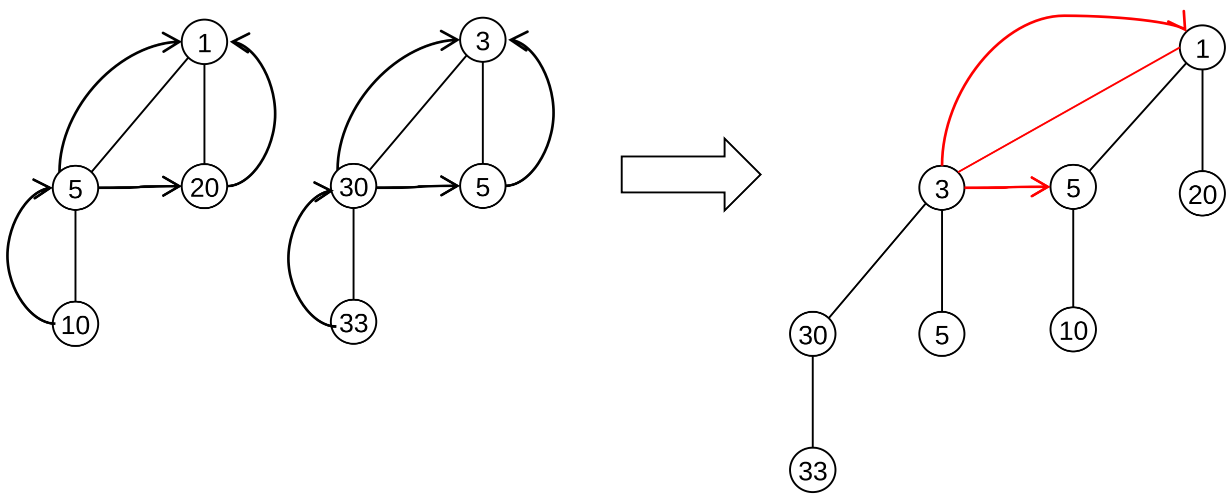
=> Heap mit `n` Knoten besteht aus maximal $\lfloor \log n \rfloor + 1$ binomiale Bäume





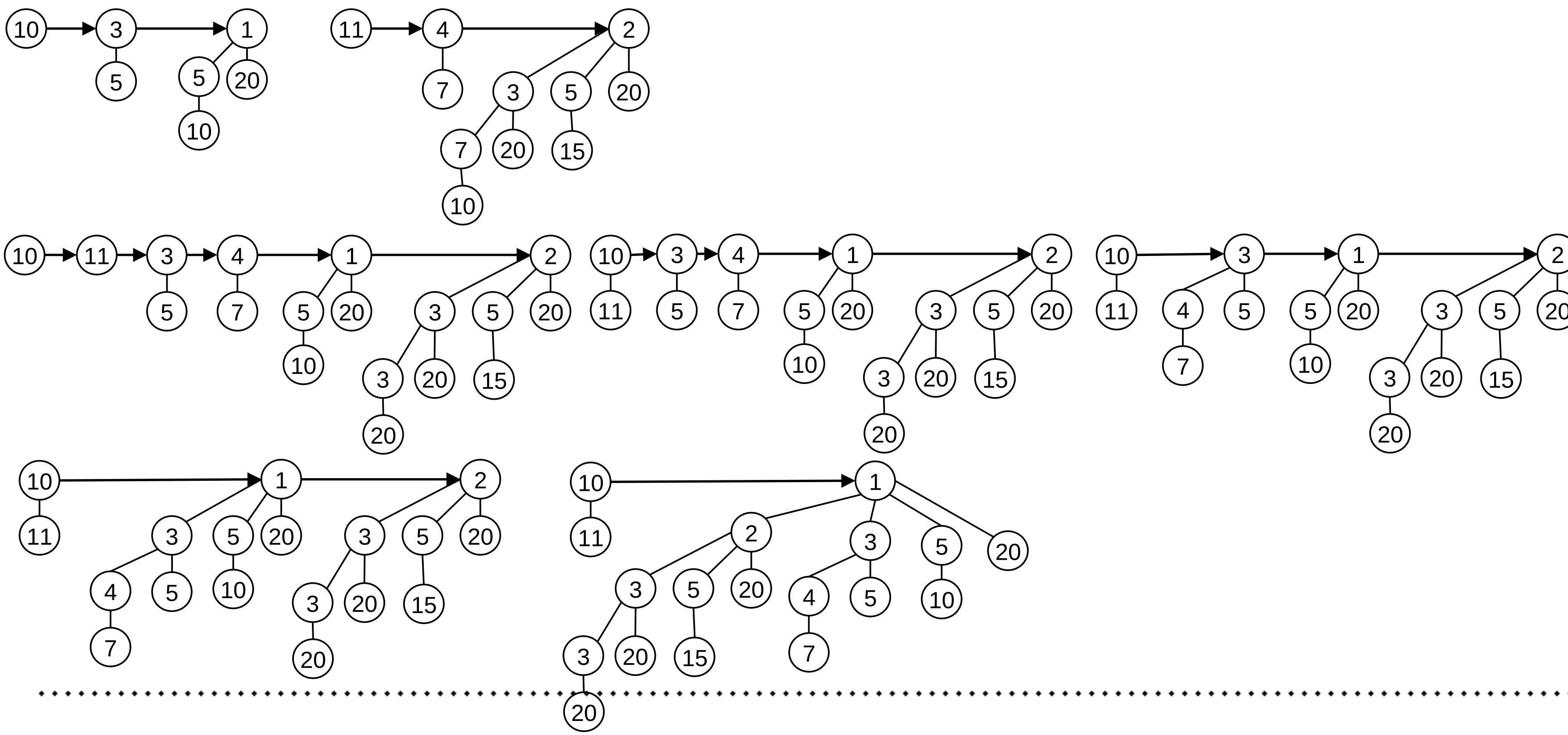

```
class Node<T> {
      T value;
      int degree;
      Node<T> parent;
      Node<T> child;
      Node<T> sibling;
}
```

```
Binomial-Link(y, z) {
      ASSERT(y.degree == z.degree)
      y.parent = z
      y.sibling = z.child
      z.child = y
      z.degree = z.degree  + 1
}
```



Basisoperationen

```
Binomial-Union(H1, H2) {
      H = Binomial-Heap-Merge(H1, H2) //Link roots and sort by increasing degree
      while (|{Bi}| > 1, i ε k) {
          Binomial-Link(Bm, Bn) // m == n
      }
      return H
}
```

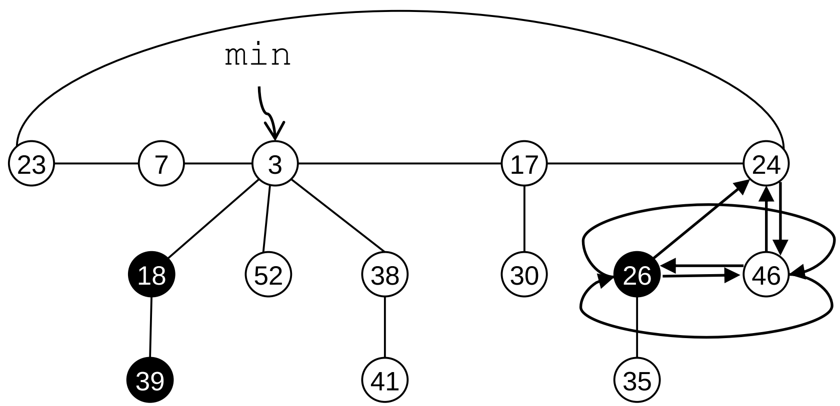


Operationen

```
Binomial-Heap-Insert(H, x) {
     Make-Binomial-Heap(H1)
     Insert(H1,x)
     Binominal-Heap-Union(H, H1)
}
Binomial-Heap-ExtractMin(H) {
     Search-Min min and remove
     Make-Binomial-Heap(H1)
     for all x in min.children in reverse order
          Insert(H1, x)
     Binominal-Heap-Union(H, H1)
     return min
}
Binomial-Heap-Decrease-Key(H, x, k) {
     ASSERT(k < x.key)
     //Vertausche in Richtung zur Wurzel
}
Binomial-Heap-Delete(H, x) {
      Binomial-Heap-Decrease-Key(H, x, -∞)
      Binomial-Heap-ExtractMin(H)
}
```

Ein Binomialer Heap
- ist eine Menge von ungeordneten Bäumen
- jeder Baum erfüllt die Heap-Bedingung

=> In einem Heap mit `n` Knoten ist der maximale Grad `D(n)` <= $\lfloor \log n \rfloor$



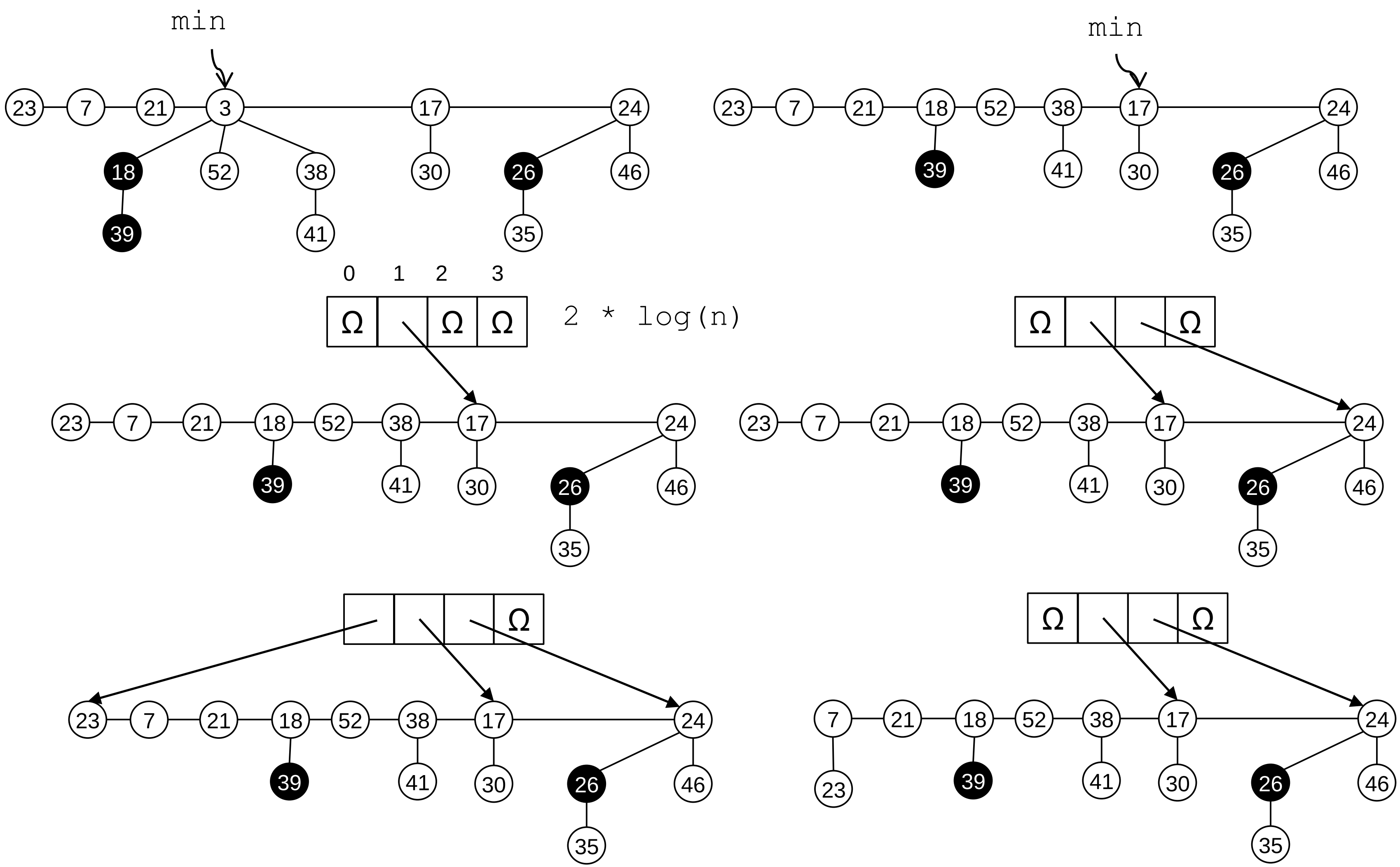

```
class Node<T> {
      T value;
      int degree;
      boolean mark;
      Node<T> parent;
      Node<T> child;
      Node<T> leftSibling;
      Node<T> rightSibling;
}
```

Operationen

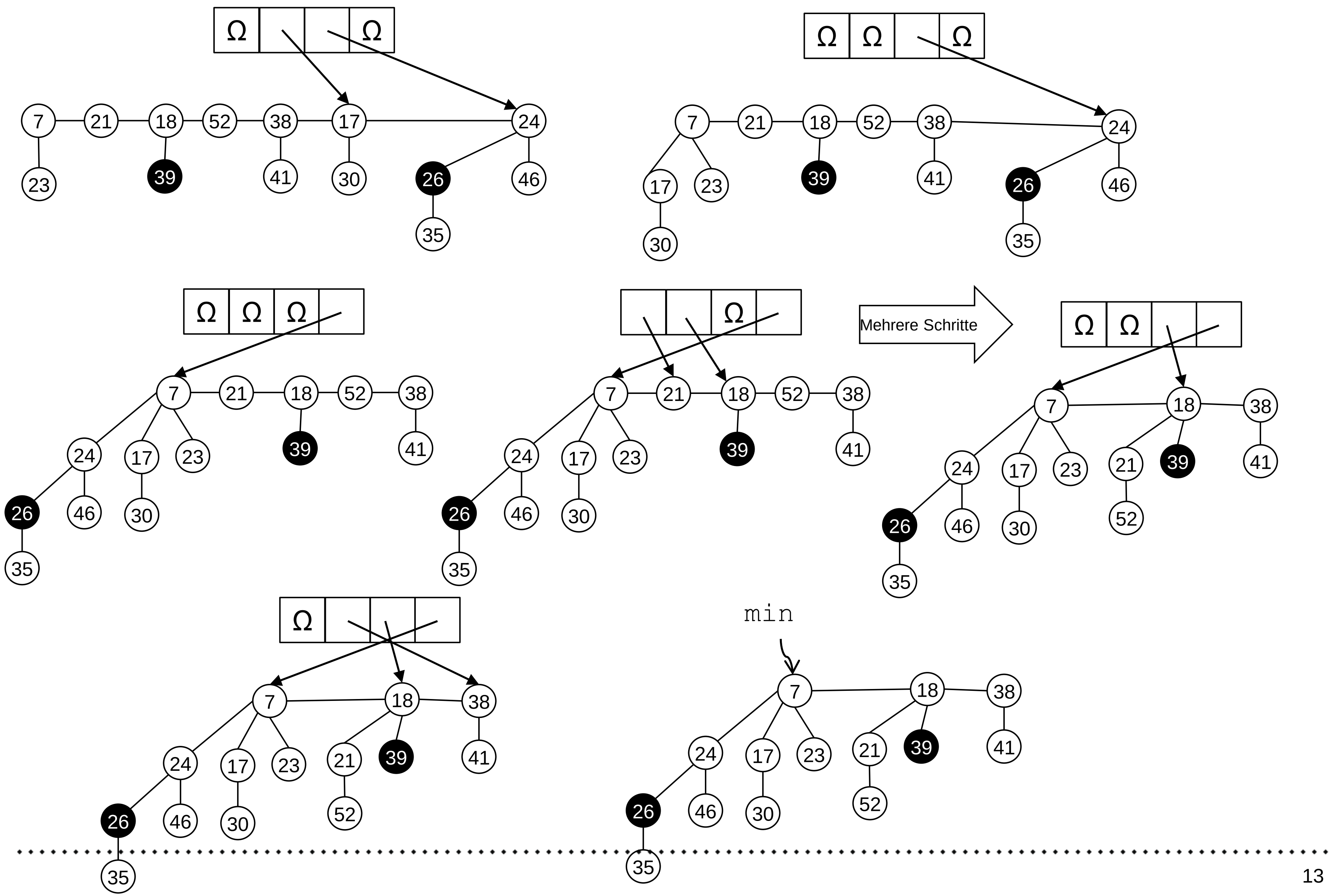
```
FIB-Heap-Insert(H, x) {
      x.degree=0 ... x.mark = false //init x
      Insert(H, x)

      if (H.min == null or x.key < h.min.key)
           H.min = x

      H.n = H.n + 1
}

FIB-Heap-Union(H1, H2) {
      FIB-Heap-Make(H)
      H.min = H1.min //Shallow copy
      Concat(H, H2) //link root lists

      if ((H1.min == null) || (H2.min != null and H2.min < H1.min))
           H.min = H2.min

      H.n = H1.n + H2.n
      return H
}
```

```
FIB-Heap-Extract-Min(H) {
     z = H.min
     if (z != null) {
          for each x of z.children
             FIB-Heap-Insert(H,x)

          Remove(H, z)
          if (z == z.rightSibling)
             H.min = null
          else {
             H.min = z.rightSibling
             FIB-Heap-Consolidate(H)
          }

          H.n = H.n - 1
     }
     return z
}
```

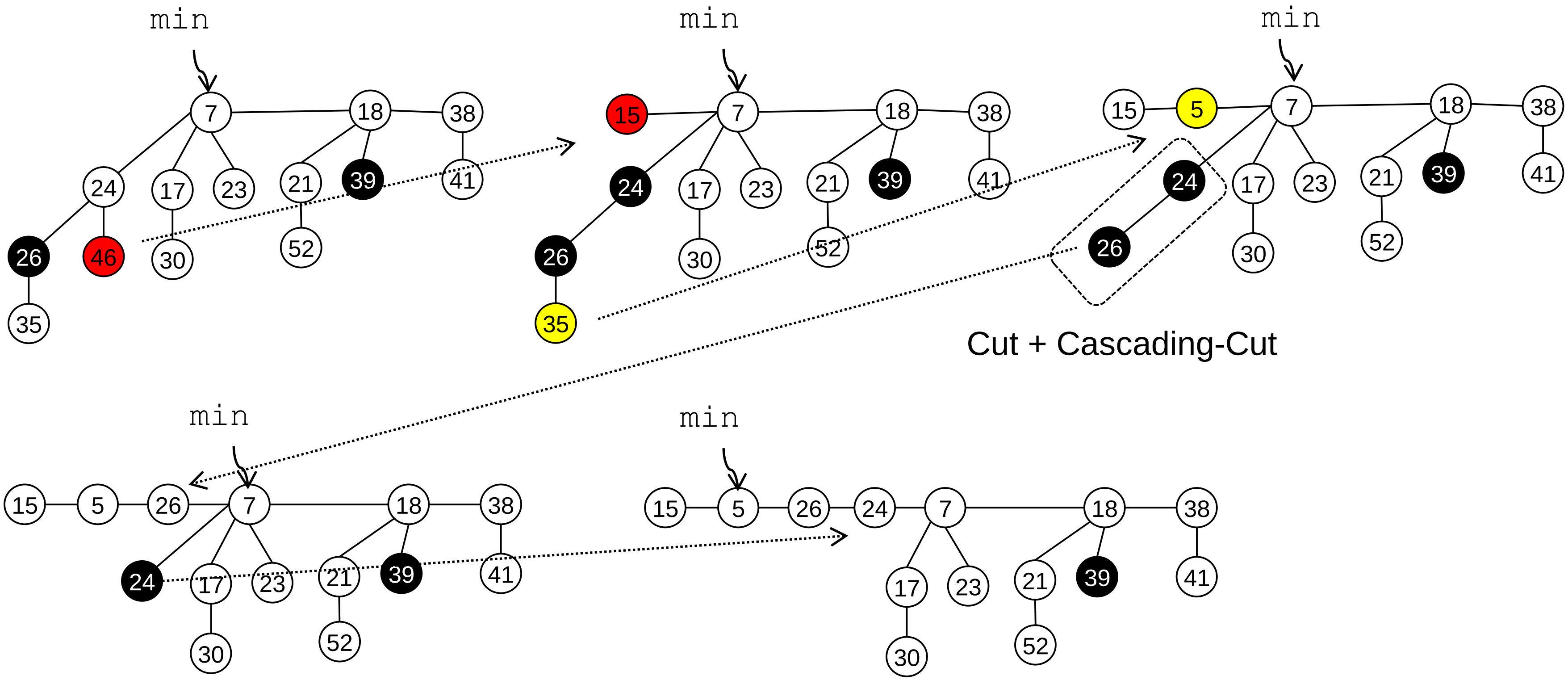
Operationen (FIB-Heap-Consolidate 1/2)



Operationen (FIB-Heap-Consolidate 2/2)



Operation (FIB-Heap-Decrease-Key)



Operation (FIB-Heap-Decrease-Key und FIB-Heap-Delete)

```
FIB-Heap-Decrease-Key(H, x, k) {
     ASSERT(k < x.key)
     x.key = k
     y = x.parent
     if (y != null and x.key < y.key) {
         FIB-Cut(H, x, y)
         FIB-Cascading-Cut(H, y)
     }

     if (x.key < H.min.key)
         H.min = x
}
```

```
FIB-Cut(H, x, y) {  //move x to rootList
     Remove x from y.children
     y.degree = y.degree - 1
     x.parent = null
     FIB-Heap-Insert(H, x)
}
```

```
FIB-Cascading-Cut(H, y) {
    z = y.parent
    if (z != null) {
         if (y.mark == false)
             y.mark = true
         else {
             FIB-Cut(H, y, z)
             FIB-Cascading-Cut(H, z)
         }
    }
}
```

```
FIB-Heap-Delete(H, x) {
    FIB-Heap-Decrease-Key(H, x, -∞)
    FIB-Heap-ExtractMin(H)
}
```