IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(17 – Fibonacci-Heaps)

Prof. Dr. Susanne Albers

Vorrangswarteschlangen

- Bekannte Datenstrukturen zum Einfügen und Entfernen von Elementen
  - Stapel (*Stack*): „last in – first out“
  - Schlange (*Queue*): „first in – first out“

- Neue Struktur: Vorrangswarteschlange (*Priority Queue*)
  - Elemente erhalten beim Einfügen einen Wert für ihre Priorität
  - Es soll immer das Element mit höchster Priorität entnommen werden
  - Veranschaulichung: „To-do“-Liste

Vorrangswarteschlangen: Elemente

Ein Element (Knoten) enthält die ihm zugeordnete Priorität sowie den eigentlichen Inhalt .

Die Priorität muss mit einem Datentyp realisiert werden, auf dem eine Ordnung besteht.

Wir speichern die Priorität als `int` (wobei eine kleinere Zahl für eine höhere Priorität steht).

```
class Node {
    int key;          // Prioritaet
    Object content;   // Inhalt
}
```

Vorrangswarteschlangen (Priority Queues)

Kernoperationen:

Ein neues Element wird hinzugefügt (zusammen mit einer Priorität).

Das Element mit der höchsten Priorität wird entfernt.

Weitere Operationen:

Rückgabe des Elements mit höchster Priorität (ohne Entfernen).

Ein gegebenes Element wird „wichtiger“, d.h. seine Priorität erhöht sich.

Ein gegebenes Element wird überflüssig, d.h. es wird entfernt.

Zwei Vorrangswarteschlangen werden zusammengefügt.

Überprüfen, ob eine Vorrangswarteschlange leer ist.

Vorrangswarteschlangen: Operationen

Operationen einer Vorrangswarteschlange (*Priority Queue*) *Q*:

`Q.insert(int k)`: Füge einen neuen Knoten mit Schlüssel (= Priorität) *k* ein.

`Q.accessmin()`: Gib den Knoten mit niedrigstem Schlüssel (= der höchsten Priorität) zurück.

`Q.deletemin()`: Entferne den Knoten mit dem niedrigsten Schlüssel.

`Q.decreasekey(Node N, int k)`: Setze den Schlüssel von Knoten *N* auf den Wert *k* herab.

`Q.delete(Node N)`: Entferne den Knoten *N*.

`Q.meld(PriorityQueue P)`: Vereinige *Q* mit der Vorrangswarteschlange *P*.

`Q.isEmpty()`: Gibt an, ob *Q* leer ist.

Bemerkung: Die effiziente Suche nach einem bestimmten Element oder Schlüssel wird in Vorrangswarteschlangen nicht unterstützt! Für *decreasekey* und *delete* muss man das entsprechende Element also bereits kennen bzw. Zugriff darauf haben.

Implementationsmöglichkeit 1: Liste

Idee: Doppelt verkettete zirkuläre Liste mit Zeiger auf das Minimum (= Knoten mit minimalem Schlüssel)

Operationen

insert: füge das neue Element ein und aktualisiere (falls nötig) den Minimum-Zeiger

accessmin: gib den Minimalknoten zurück

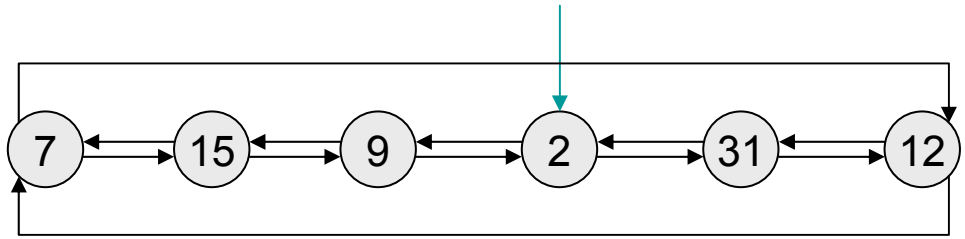
deletemin: entferne den Minimalknoten, aktualisiere den Minimum-Zeiger

decreasekey: setze den Schlüssel herab und aktualisiere den Minimum-Zeiger

delete: falls der zu entfernende Knoten der Minimalknoten ist, führe deletemin aus; ansonsten entferne den Knoten.

meld: hänge die beiden Listen aneinander

Implementation als lineare Liste



Implementationsmöglichkeit 2: (Min-)Heap

Idee: Speichere die Elemente in einem heapgeordneten Array (vgl. Heap Sort)

Operationen

insert: füge das neue Element an der letzten Stelle ein und stelle durch Vertauschungen die Heapordnung wieder her

accessmin: der Minimalknoten steht immer an der ersten Position

deletemin: ersetze das erste Element durch das letzte, dann versickere

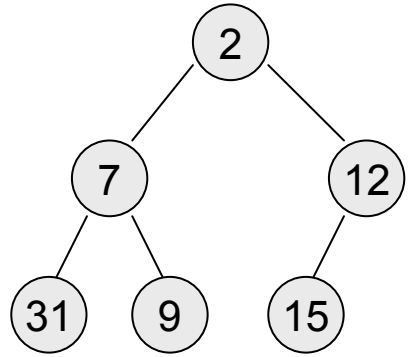
decreasekey: setze den Schlüssel herab und stelle durch Vertauschungen die Heapordnung wieder her

delete: ersetze das entfernte Element durch das letzte, dann versickere

meld: füge alle Elemente des kleineren Heaps nacheinander in den größeren ein

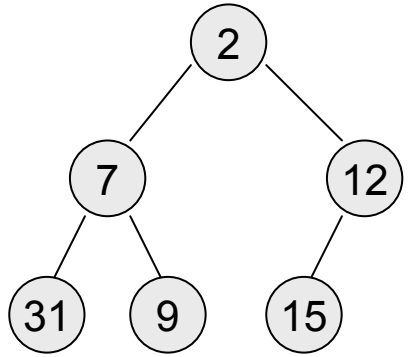
Implementation als Min-Heap

2	7	12	31	9	15



Implementation als Min-Heap

2	7	12	31	9	15



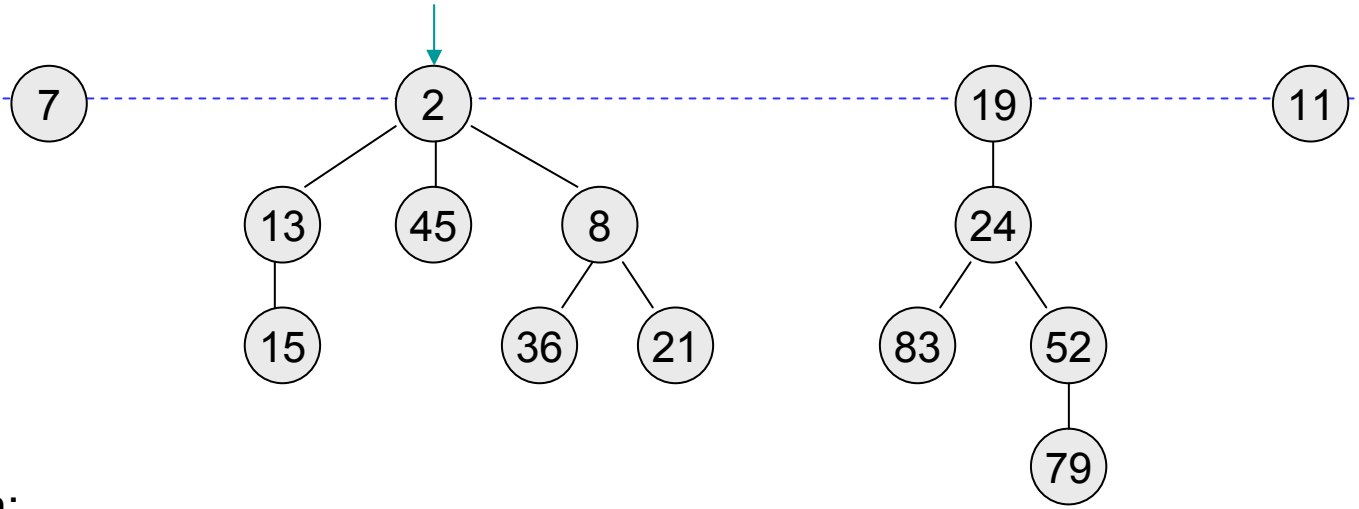
Implementationen: Vergleich

	Lineare Liste	(Min-)Heap	???
insert:	$O(1)$	$O(\log n)$	$O(1)$
accessmin:	$O(1)$	$O(1)$	$O(1)$
deletemin:	$O(n)$	$O(\log n)$	$O(\log n)$
decreasekey:	$O(1)$	$O(\log n)$	$O(1)$
delete:	$O(n)$	$O(\log n)$	$O(\log n)$
meld:	$O(1)$	$O(m \log(n+m))$	$O(1)$

Frage: Lassen sich die Vorteile von Listen und Heaps verbinden?

Fibonacci-Heaps: Idee

- Liste von Bäumen (beliebigen Verzweigungsgrades), die alle *heapgeordnet* sind.



Definition:
Ein Baum heißt (min-)heapgeordnet, wenn der Schlüssel jedes Knotens größer oder gleich dem Schlüssel seines Vaterknotens ist (sofern er einen Vater hat).

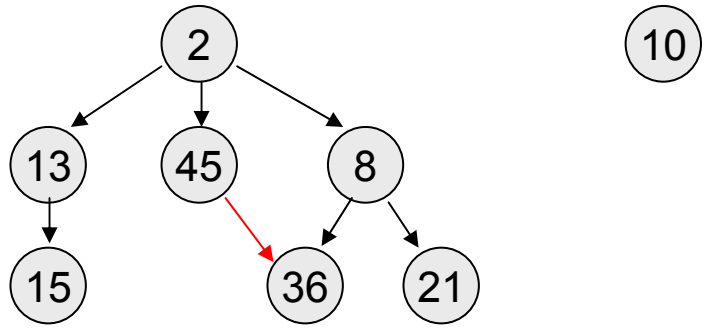
- Die Wurzeln der Bäume sind in einer *doppelt verketteten, zirkulären Liste* miteinander verbunden (*Wurzelliste*).
- Der Einstiegspunkt ist ein Zeiger auf den Knoten mit minimalem Schlüssel.

Exkurs: Bäume

Bäume lassen sich als Verallgemeinerung von Listen auffassen:

- Es gibt genau ein Anfangselement („Wurzel“).
- Jedes Element kann beliebig viele Nachfolger („Söhne“) haben.
- Jedes Element (außer der Wurzel) ist Nachfolger von genau einem Knoten.

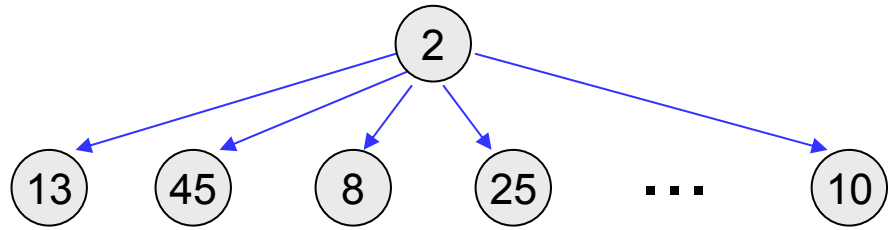
```
class Tree {
    TreeNode root;
}

class TreeNode {
    int key;
    Node[] children;
}
```



Repräsentation von Bäumen

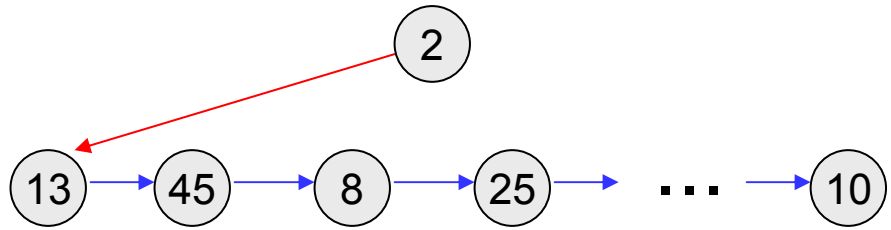
Bei Bäumen mit hohem Verzweigungsgrad ist es aus Speicherplatzgründen ungünstig, in jedem Knoten Zeiger auf alle Söhne zu speichern.

```
class TreeNode {
    int key;
    Node[] children;
}
```

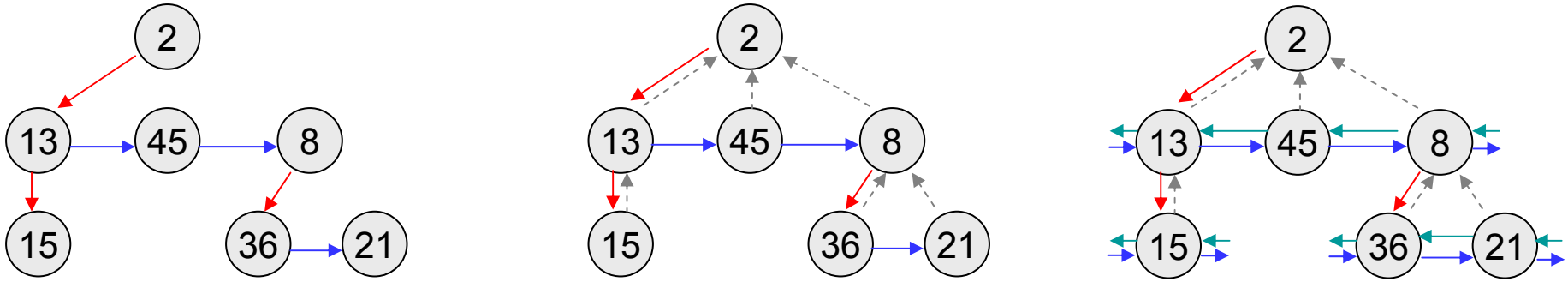


Eine platzsparende Alternative ist die *Child-Sibling*-Darstellung: hier sind alle Söhne in einer Liste untereinander verkettet. Dadurch genügt es, im Vaterknoten *einen* Zeiger auf den ersten Sohn zu speichern.

```
class ChildSiblingNode {
    int key;
    Node child;
    Node sibling;
}
```



Child-Sibling-Repräsentation



- Um sich im Baum auch aufwärts bewegen zu können, fügt man einen Zeiger auf den Vaterknoten hinzu.
- Um das Entfernen von Söhnen (und das Aneinanderhängen von Sohn-Listen) in O(1) zu realisieren, verwendet man doppelt verkettete zirkuläre Listen.
- Also hat jeder Knoten 4 Zeiger: `child`, `parent`, `left`, `right`

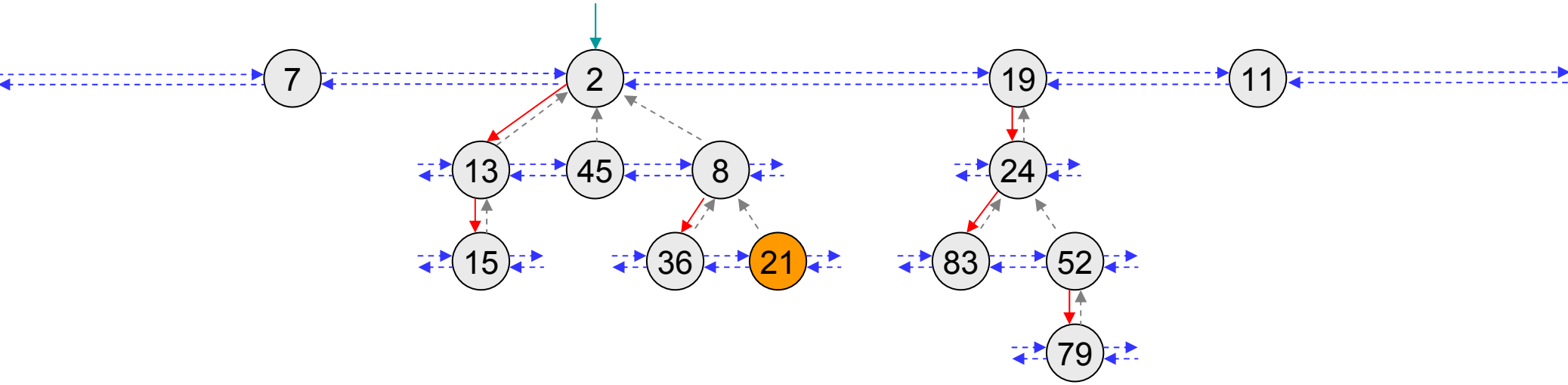
Fibonacci-Heaps: Knotenformat

```
class FibNode {
    int key;                    // Schlüssel (Priorität)
    Object content;             // der eigentliche Inhalt
    FibNode parent, child;      // Zeiger auf Vater und einen Sohn
    FibNode left, right;        // Zeiger auf linken und rechten Nachbarn

    int rank;                   // Anzahl der Söhne dieses Knotens
    boolean mark;               // Markierung
}
```

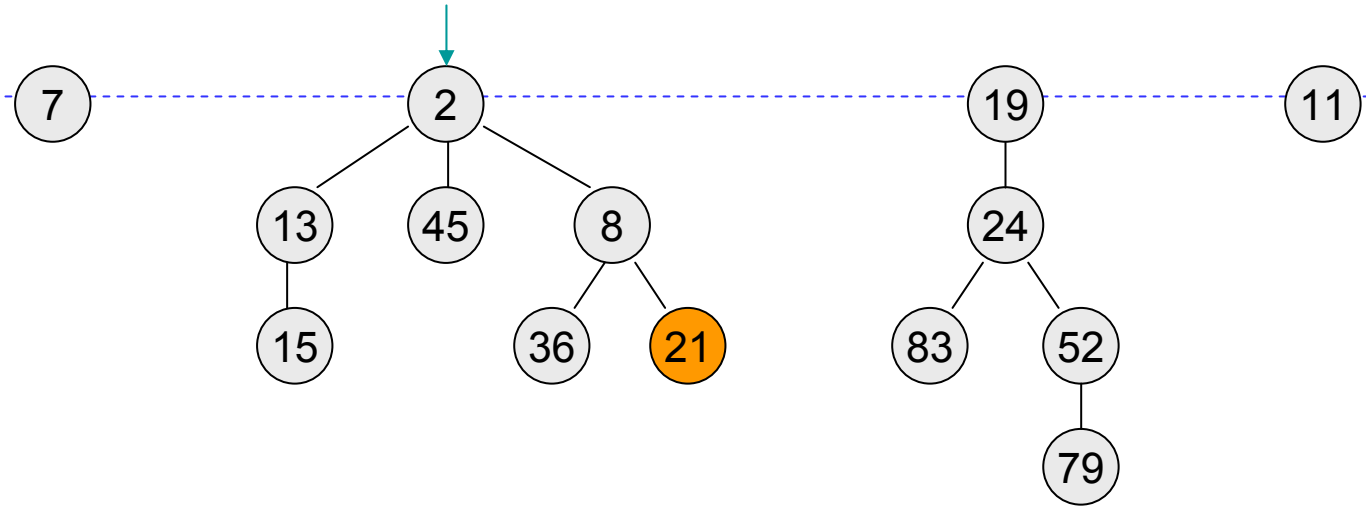
Die Zahl **rank** gibt an, wie viele Söhne der Knoten hat (= der *Rang* des Knotens)

Die Bedeutung der Markierung **mark** wird später deutlich. Diese Markierung gibt an, ob der Knoten bereits einmal einen seiner Söhne verloren hat, seitdem er selbst zuletzt Sohn eines anderen Knotens geworden ist.

Fibonacci-Heap: Detaillierte Darstellung



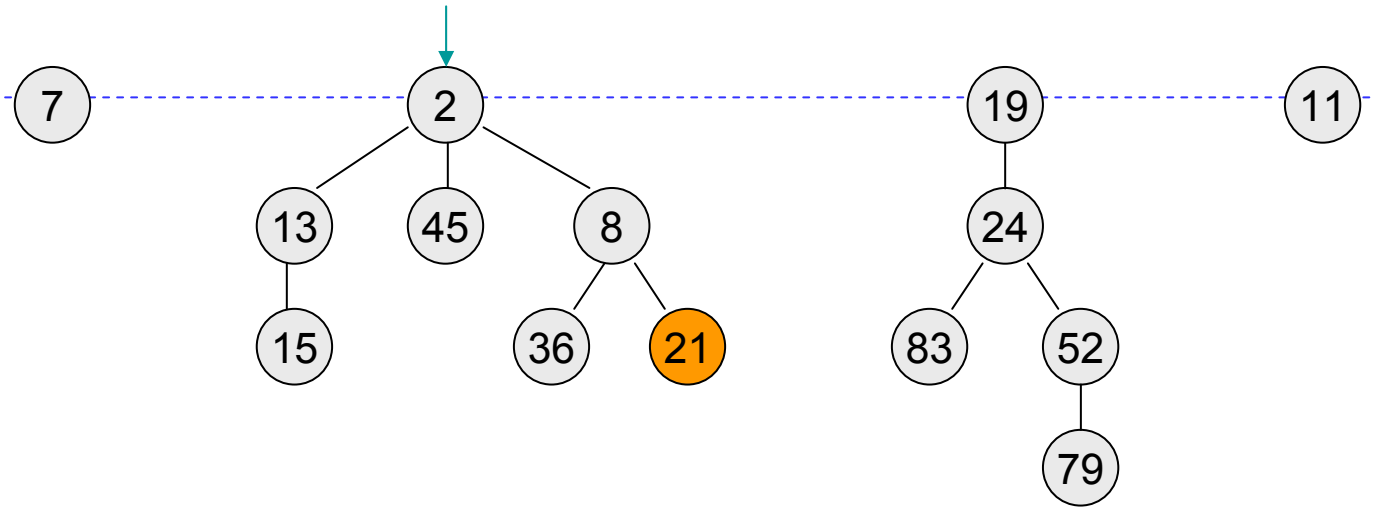
Fibonacci-Heap: Vereinfachte Darstellung



Fibonacci-Heaps: Operationen

- **`Q.accessmin()`**: Gib den Knoten `Q.min` zurück (bzw. `null`, wenn `Q` leer ist).

- **`Q.insert(int k)`**: Erzeuge einen neuen Knoten `N` mit Schlüssel `k` und füge ihn in die Wurzelliste von `Q` ein. Falls `k < Q.min.key`, aktualisiere den Minimum-Zeiger (setze `Q.min = N`). Gib den neu erzeugten Knoten zurück.



Manipulation von Bäumen in Fibonacci-Heaps

Drei Basis-Methoden zur Manipulation von Bäumen in Fibonacci-Heaps:

link: „Wachstum“ von Bäumen.

Zwei Bäume werden zu einem neuen verbunden.

cut: „Beschneiden“ von Bäumen im Inneren.

Ein Teilbaum wird aus einem Baum herausgetrennt und als neuer Baum in die Wurzelliste eingefügt.

remove: „Spalten“ von Bäumen an der Wurzel.

Entfernt die Wurzel eines Baums und fügt die Söhne der Wurzel als neue Bäume in die Wurzelliste ein.

Baummanipulation: *link*

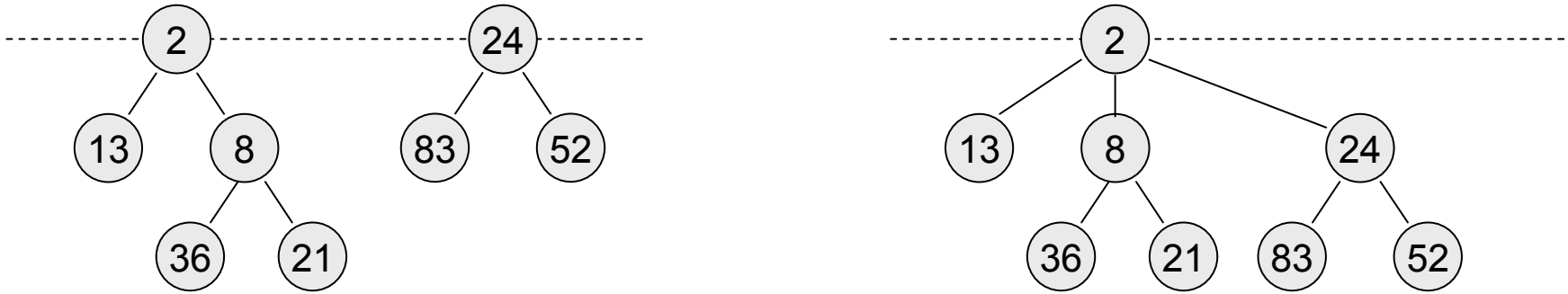
link:

Input: 2 Knoten mit demselben Rang *k* in der Wurzelliste

Methode: vereinigt zwei Bäume mit gleichem Rang, indem die Wurzel mit größerem Schlüssel zu einem neuen Sohn der Wurzel mit kleinerem Schlüssel gemacht wird. Die Gesamtzahl der Bäume verringert sich um 1, die Knotenzahl ändert sich nicht.

Output: 1 Knoten mit Rang *k*+1

Laufzeit:



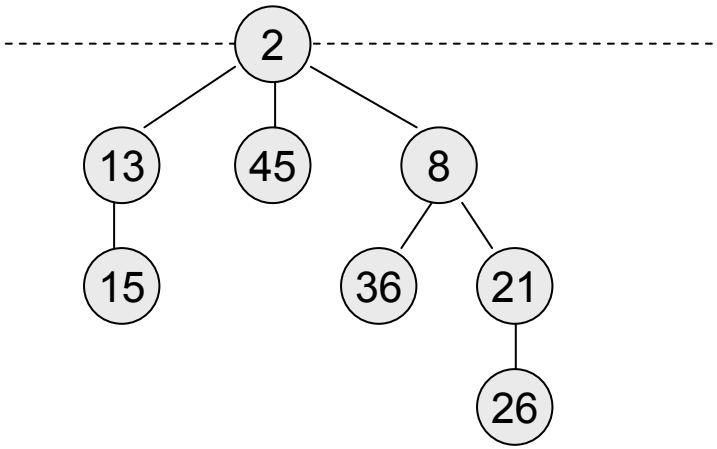
Baummanipulation: *cut*

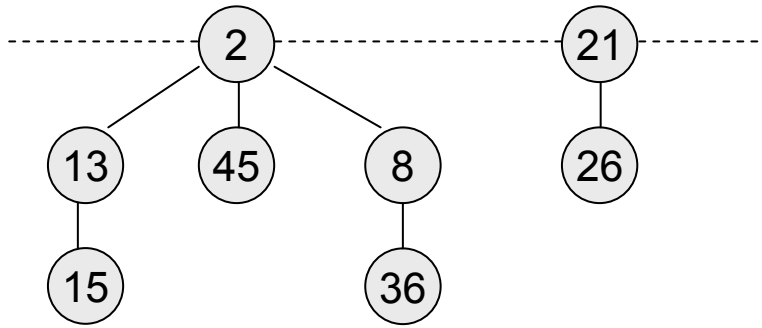
cut:

Input: 1 Knoten, der nicht in der Wurzelliste ist

Methode: trennt den Knoten (samt dem Teilbaum, dessen Wurzel er ist) von seinem Vater ab und fügt ihn als neuen Baum in die Wurzelliste ein.
Die Gesamtzahl der Bäume erhöht sich um 1, die Knotenzahl ändert sich nicht.

Laufzeit:





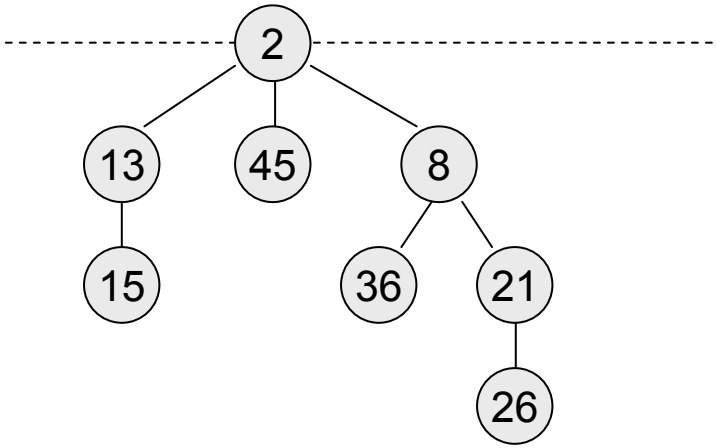
Baummanipulation: *remove*

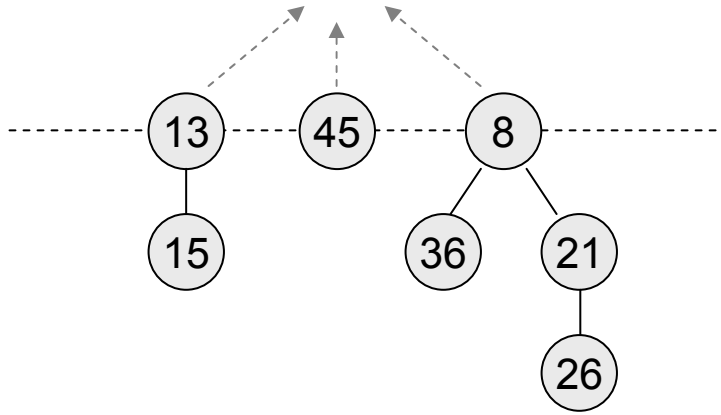
remove:

Input: 1 Knoten mit Rank *k* aus der Wurzelliste

Methode: entfernt die Wurzel eines Baums und fügt stattdessen seine *k* Söhne in die Wurzelliste ein. Die Zahl der Bäume erhöht sich um *k*-1, die Gesamtzahl der Knoten verringert sich um 1.

Laufzeit: [sofern die Vaterzeiger der Söhne nicht gelöscht werden!]





Weitere Operationen

- Mit Hilfe der drei Manipulationsmethoden
  - *link*
  - *cut*
  - *remove*

  lassen sich die noch fehlenden Operationen
  - *deletemin*
  - *decreasekey*
  - *delete*

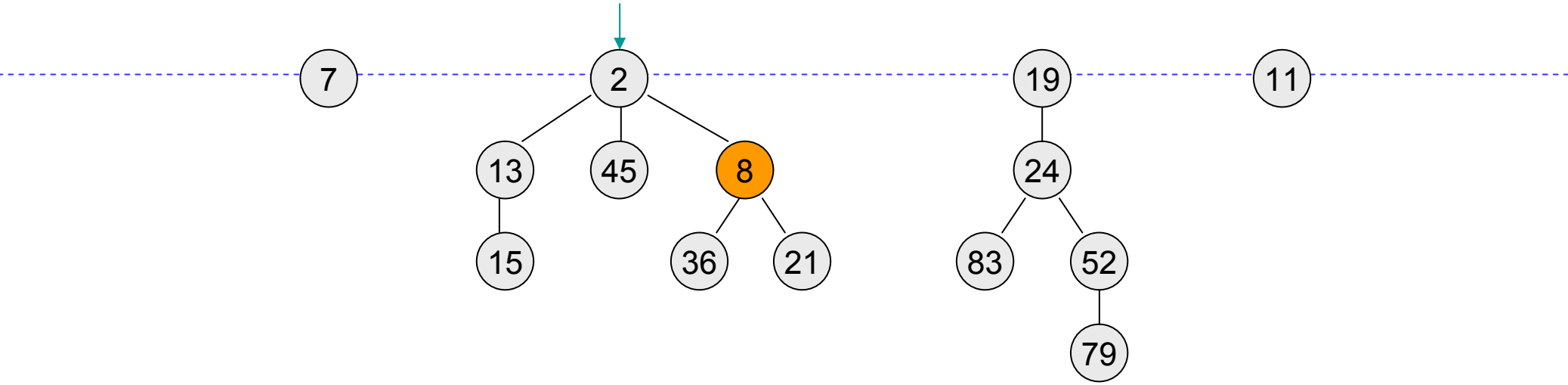
  beschreiben.

Entfernen des Minimums

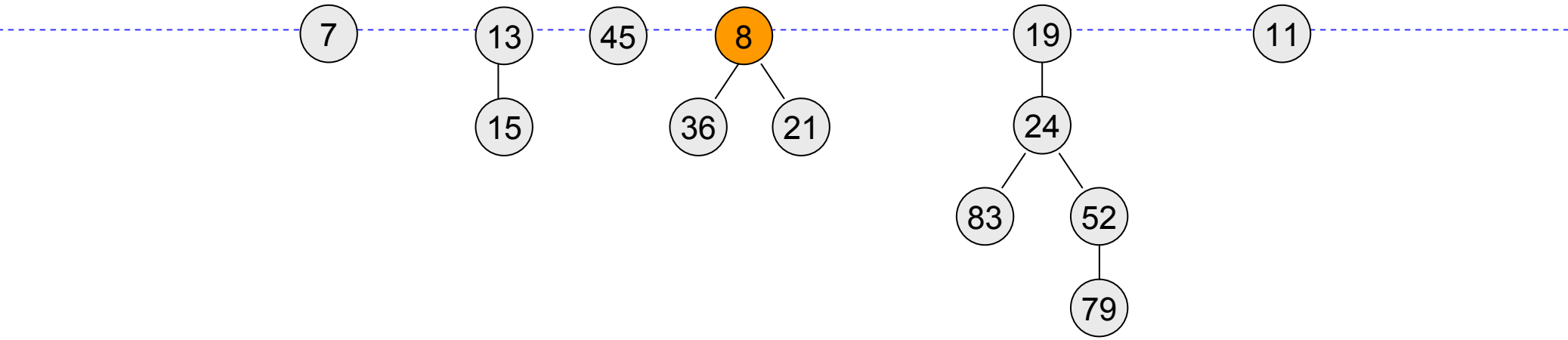
`Q.deletemin()`:

- Wenn *Q* leer ist, gib *null* zurück.
- Andernfalls:
  - Entferne den Minimalknoten (mit *remove*).
  - „Konsolidiere“ die Wurzelliste:
    Verbinde (mit *link*) je zwei Wurzelknoten mit demselben Rang, und zwar solange, bis nur noch Knoten mit unterschiedlichem Rang in der Wurzelliste vorkommen.
    Lösche dabei die Markierungen der Knoten, die zum Sohn eines anderen werden, und entferne außerdem evtl. vorhandene Vaterzeiger der Wurzelknoten.
  - Finde unter den Wurzelknoten das neue Minimum.
  - Gib den entfernten Knoten zurück.

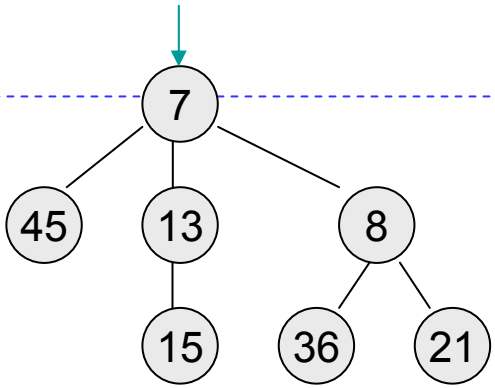
deletemin: Beispiel

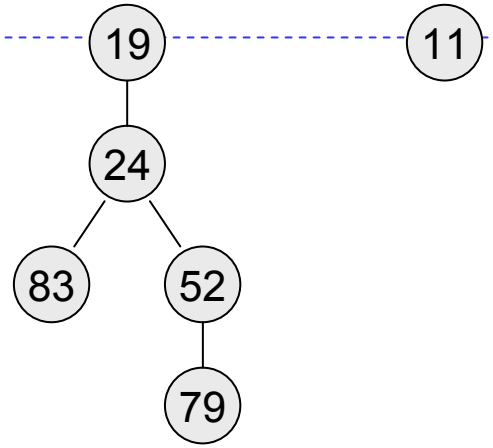


deletemin: Beispiel



deletemin: Beispiel



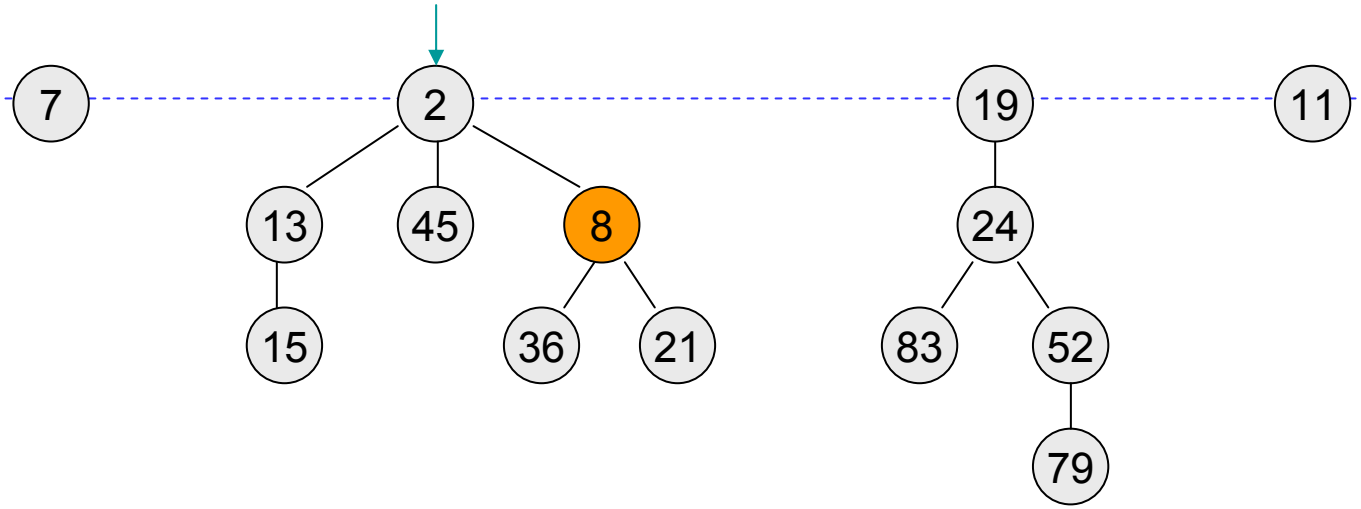


Herabsetzen eines Schlüssels

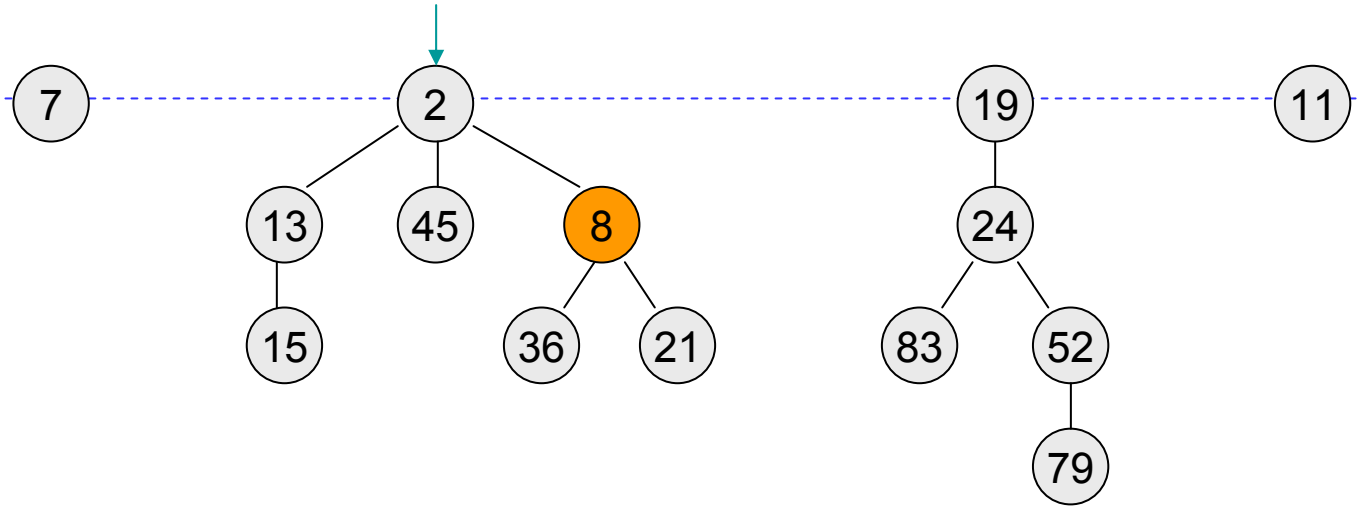
`Q.decreasekey(FibNode N, int k)`:

- Setze den Schlüsselwert von **`N`** auf **`k`** herab.
- Wenn die Heap-Bedingung nicht mehr erfüllt ist (**`k < N.parent.key`**):
  - Trenne **`N`** von seinem Vater ab (mit *cut*)
  - Falls der Vater markiert ist (**`N.parent.mark == true`**), trenne auch ihn von seinem Vater ab; wenn auch dessen Vater markiert ist, trenne auch diesen ab, usw. (*„cascading cuts“*)
  - Markiere den Knoten, dessen Sohn zuletzt abgetrennt wurde (sofern dieser kein Wurzelknoten ist).
  - Aktualisiere den Minimum-Zeiger (falls **`k < min.key`**).

decreasekey: Beispiel 1



decreasekey: Beispiel 2



Weitere Operationen

- **`Q.meld(FibHeap P)`**: Hänge die Wurzelliste von **`P`** an die Wurzelliste von **`Q`** an. Aktualisiere den Minimum-Zeiger von **`Q`**: falls **`P.min.key < Q.min.key`**, setze **`Q.min = P.min`**.

- **`Q.isEmpty()`**: Falls **`Q.size == 0`**, gib *true* zurück; ansonsten *false*.

Laufzeiten

	Lineare Liste	(Min-)Heap	Fibonacci-Heap
insert:	$O(1)$	$O(\log n)$	$O(1)$
accessmin:	$O(1)$	$O(1)$	$O(1)$
deletemin:	$O(n)$	$O(\log n)$	?
decreasekey:	$O(1)$	$O(\log n)$	?
delete:	$O(n)$	$O(\log n)$	?
meld:	$O(1)$	$O(m \log(n+m))$	$O(1)$

Laufzeitanalyse

Laufzeit von *deletemin()*:

remove:	$O(1)$
consolidate:	?
updatemin:	O(#Wurzelknoten **nach** consolidate) =

Nach dem Konsolidieren gibt es von jedem Rang nur noch höchstens einen Wurzelknoten.

Definiere *maxRank*(*n*) als den höchstmöglichen Rang, den ein Wurzelknoten in einem Fibonacci-Heap der Größe *n* haben kann. (Berechnung von *maxRank*(*n*) später.)

Nun müssen wir die Komplexität von *consolidate* bestimmen.

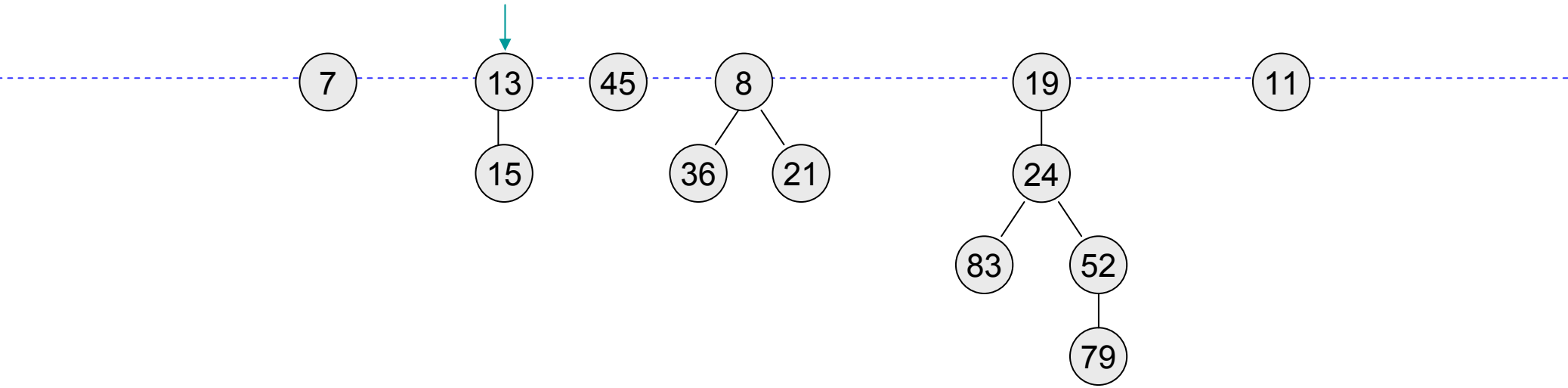
deletemin: Konsolidieren der Wurzelliste

- Wie kann man das Konsolidieren effizient realisieren?
- Beobachtungen:
  - Jeder Wurzelknoten muss mindestens einmal betrachtet werden
  - Am Ende darf es für jeden möglichen Rang höchstens einen Knoten geben
- Idee: Trage die Wurzelknoten in ein temporär geschaffenes Array ein. Jeder Knoten wird an der Arrayposition eingetragen, die seinem Rang entspricht. Ist eine Position schon besetzt, so weiß man, dass es einen weiteren Knoten mit demselben Rang gibt, kann diese beiden mit *link* verschmelzen und den neuen Baum an der nächsthöheren Position im Array eintragen.

consolidate: Beispiel

Rang-Array:

0	1	2	3	4	5



Analyse von consolidate

```
rankArray = new FibNode[maxRank(n)+1];     // Erstelle das Array
```

```
for „each FibNode N in rootlist“ {
    while (rankArray[N.rank] != null) {    // Position besetzt
        N = link(N, rankArray[N.rank]);    // Verbinde Bäume
        rankArray[N.rank-1] = null;        // Lösche alte Pos.
    }
    rankArray[N.rank] = N;                 // Eintragen in Array
}
```

Gesamtkosten von *deletemin*

remove:	$O(1)$
link-Operationen:	#links $\cdot$ $O(1)$
Update Minimum-Zeiger:	$O(maxRank(n))$

Gesamtkosten:	$O(\#\text{links}) + O(maxRank(n))$

Laufzeitanalyse

Laufzeit von *decreasekey()*:

Schlüssel neu setzen:	$O(1)$
cut:	$O(1)$
Cascading cuts:	$\#\text{cuts} \cdot O(1)$
Markieren:	$O(1)$

Gesamtkosten:	$O(\#\text{cuts})$

Berechnung von *maxRank*(*n*)

Erinnerung: Fibonacci-Zahlen

$F_0 = 0$

$F_1 = 1$

$F_{k+2} = F_{k+1} + F_k$ für $k \geq 0$

Die Folge der Fibonacci-Zahlen wächst exponentiell mit $F_{k+2} \geq 1.618^k$

Es gilt außerdem: $F_{k+2} = 1 + \sum_{i=1}^{k} F_i$

(Beweis durch vollständige Induktion über k.)

Berechnung von *maxRank*(*n*)

Lemma 1:

Sei N ein Knoten in einem Fibonacci-Heap und $k = N.\text{rank}$. Betrachte die Söhne $C_1, ..., C_k$ von N in der Reihenfolge, in der sie (mit *link*) zu N hinzugefügt wurden. Dann gilt:

(1) $C_1.\text{rank} \geq 0$

(2) $C_i.\text{rank} \geq i - 2$ für $i = 2, ..., k$

Beweis: (1) klar

(2) Als S_i zum Sohn von N wurde, waren $C_1, ..., C_{i-1}$ schon Söhne von N, d.h. es war $N.\text{rank} \geq i-1$. Da durch *link* immer Knoten mit gleichem Rang verbunden werden, war beim Einfügen also auch $C_i.\text{rank} \geq i-1$. Seither kann C_i höchstens einen Sohn verloren haben (wegen *cascading cuts*), daher muss gelten: $C_i.\text{rank} \geq i - 2$

Berechnung von *maxRank*(*n*)

Lemma 2:

Sei N ein Knoten in einem Fibonacci-Heap und $k = N.\text{rank}$.
Sei $size(N)$ = die Zahl der Knoten im Teilbaum mit Wurzel N.

Dann gilt: $size(N) \geq F_{k+2} \geq 1.618^k$

D.h. ein Knoten mit k Söhnen hat mind. F_{k+2} Nachkommen (inkl. sich selbst).

Beweis: Sei $S_k = \min \{size(N) \mid N \text{ mit } N.\text{rank} = k\}$, d.h. die kleinstmögliche Größe eines Baums mit Wurzelrang k. (Klar: $S_0 = 1$ und $S_1 = 2$.)
Seien wieder $C_1, ..., C_k$ die Söhne von N in der Reihenfolge, in der sie zu N hinzugefügt wurden.

Es ist $size(N) \geq S_k \geq$

Berechnung von *maxRank*(*n*)

Satz:

Der maximale Rang *maxRank*(*n*) eines beliebigen Knotens in einem Fibonacci-Heap mit *n* Knoten ist beschränkt durch $2 \log n$.

Beweis: Sei N ein Knoten eines Fibonacci-Heaps mit n Knoten und sei $k = N.\text{rank}$.

Es ist $n \geq size(N) \geq 1.618^k$ (nach Lemma 2)

Daher ist $k \leq \log_{1.618}(n) \leq 2 \log n$

Amortisierte Analyse

Potentialmethode zur Analyse der Kosten der Operationen der Fibonacci-Heaps

Potentialfunktion Φ für Fibonacci-Heap Q:

$$\Phi = r + 2\,m$$

r = Anzahl der Knoten in Wurzelliste von Q

m = Anzahl der markierten Knoten in Q, die sich nicht in der Wurzelliste befinden.

Amortisierte Kosten a_i der i-ten Operation:

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

$$= t_i + (r_i - r_{i-1}) + 2(m_i - m_{i-1})$$

Amortisierte Kosten von *insert*

$t_i = 1$

$r_i - r_{i-1} = 1$

$m_i - m_{i-1} = 0$

$a_i = 1 + 1 + 0 = O(1)$

Amortisierte Kosten von *deletemin*

$t_i = r_{i-1} + 2 \log n$

$r_i - r_{i-1} \leq \quad 2 \log n - r_{i-1}$

$m_i - m_{i-1} \leq 0$

$a_i \leq \; r_{i-1} + 2 \log n + 2 \log n - r_{i-1} + 0 \qquad = \qquad O(\log n)$

Amortisierte Kosten von *decreasekey*

c = # cuts

$t_i = c + 2$

$r_i - r_{i-1} = c + 1$

$m_i - m_{i-1} \leq -c + 1$

$a_i \leq c + 2 + c + 1 + 2 (-c + 1)$

$= O(1)$

Zusammenfassung

	Lineare Liste	(Min-)Heap	Fibonacci-Heap
insert:	$O(1)$	$O(\log n)$	$O(1)$
accessmin:	$O(1)$	$O(1)$	$O(1)$
deletemin:	$O(n)$	$O(\log n)$	$O(\log n)$*
decreasekey:	$O(1)$	$O(\log n)$	$O(1)$*
delete:	$O(n)$	$O(\log n)$	$O(\log n)$*
meld:	$O(1)$	$O(m \log(n+m))$	$O(1)$

*Amortisierte Kosten