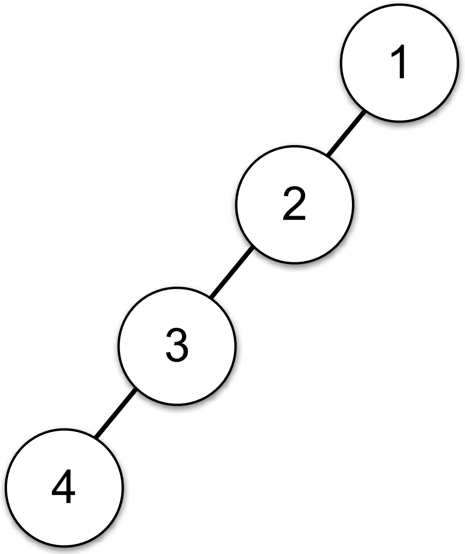
Algorithmen und Datenstrukturen

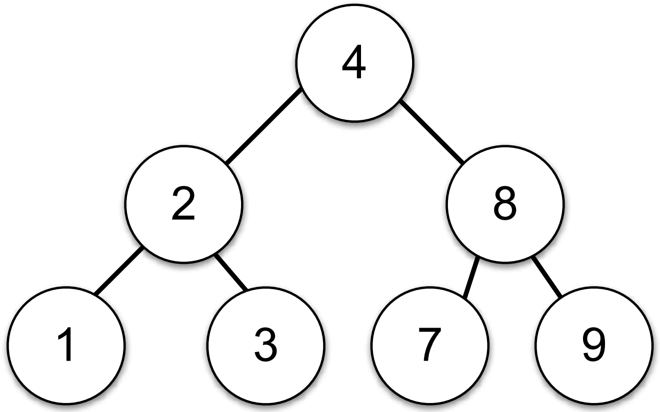
Prof. Dr. Thomas Wiemann - FB AI

Hochschule Fulda
University of Applied Sciences

Rekapitulation Suchbäume

- Binäre Suchbäume sind eine vielseitig verwendbare Datenstruktur zur Verwaltung von Elementen mit Schlüsselm
- Suchbaum-Eigenschaft: Links kleiner, rechts größer für alle Knoten
- Operationen hängen meist von der Höhe des BSTs ab
- Wie kann man Suchbäume *balanciert* halten?





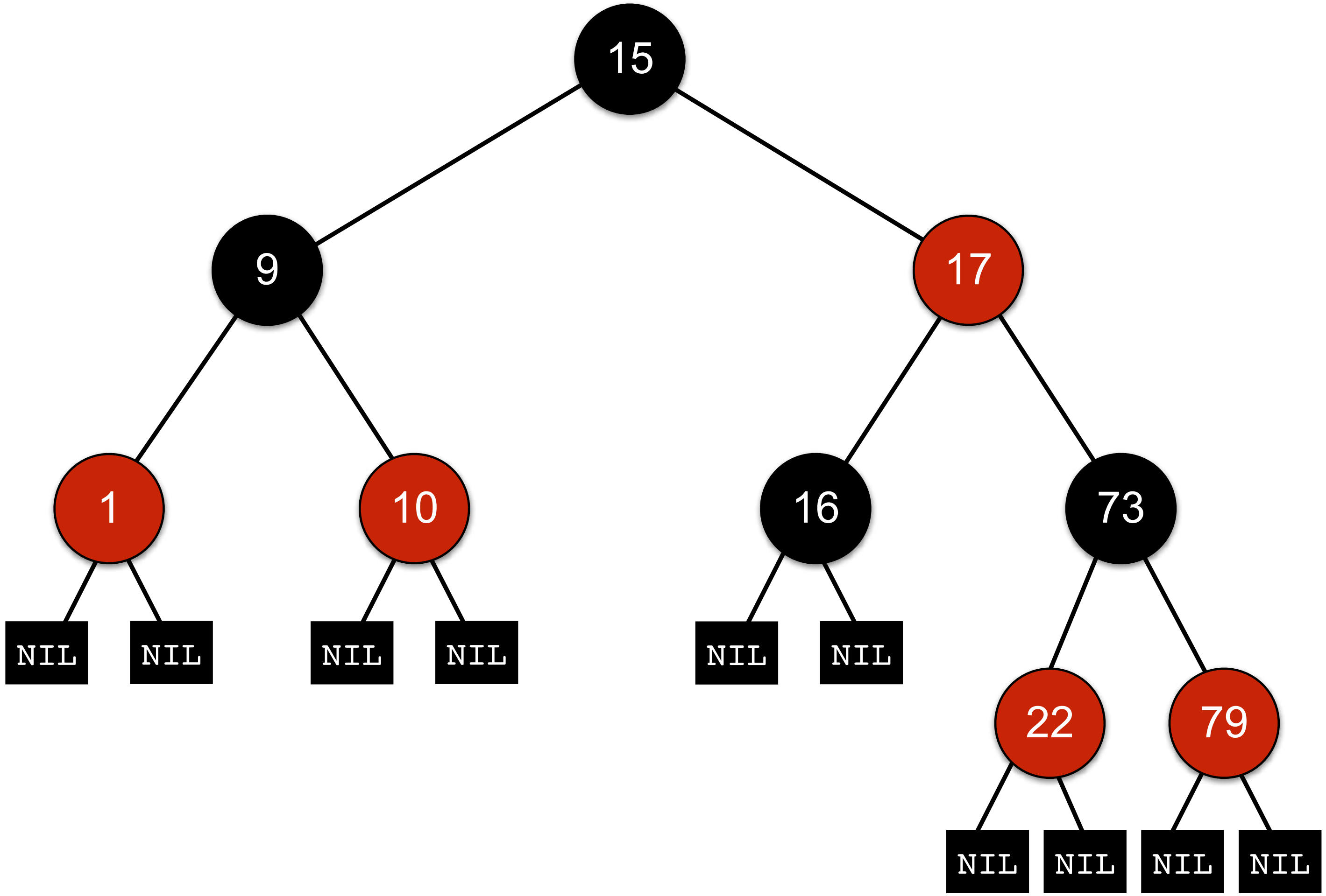
Rot-Schwarz-Bäume (1)

- Definiere eine Zusatzeigenschaft, die Balanciertheit garantiert
- Sorge beim Einfügen und Löschen dafür, dass diese Eigenschaft erhalten bleibt
- Blätter haben einen speziellen Wert `NIL`

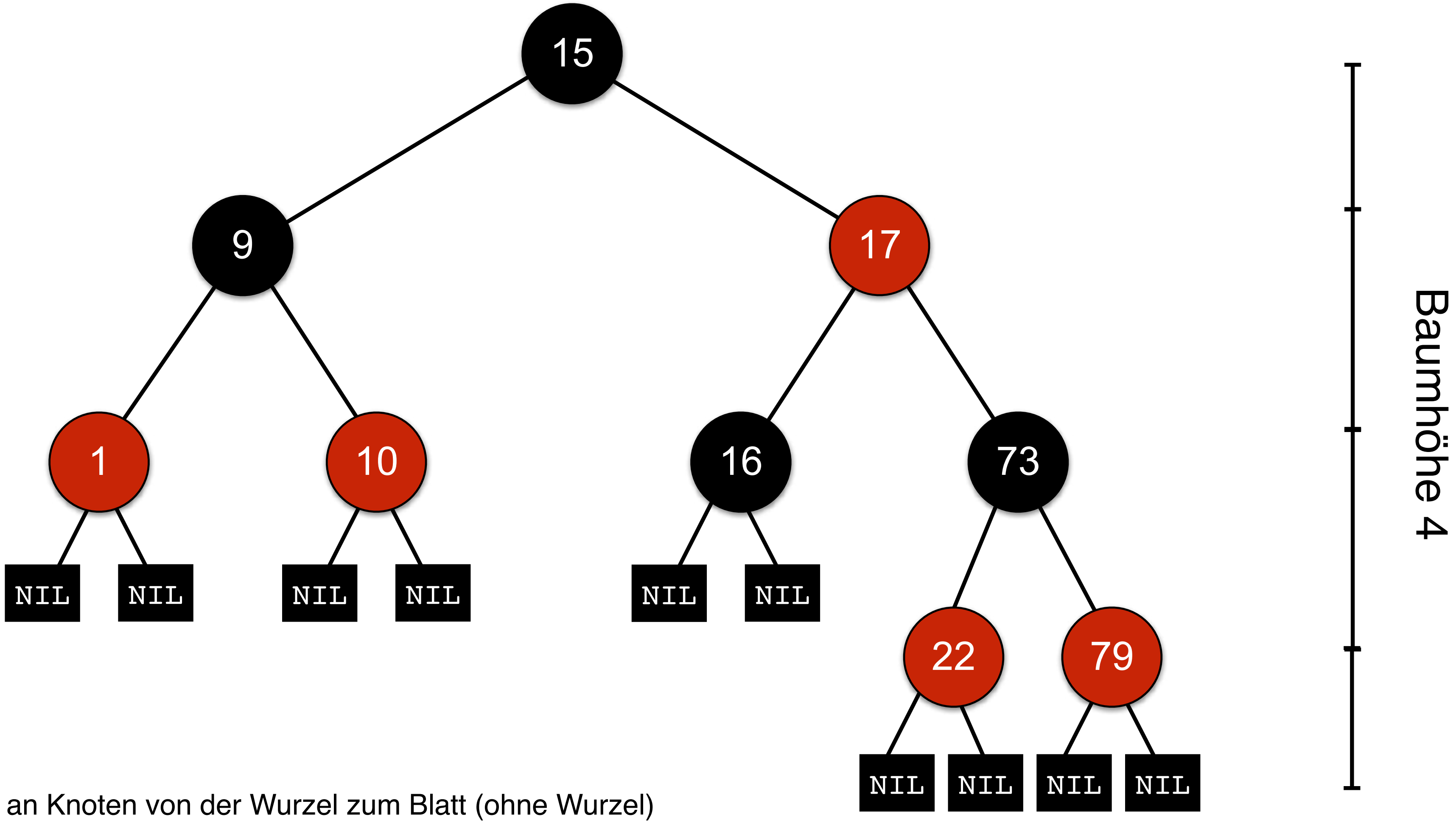
Regeln

1. Jeder Knoten ist entweder rot oder schwarz
2. Die Wurzel ist schwarz
3. Alle `NIL`-Blätter sind schwarz
4. Ein roter Knoten darf keine roten Kinder haben
5. Alle Pfade von einem Knoten zu den Blättern enthalten gleich viele schwarze Knoten

Red-Black Tree Beispiel



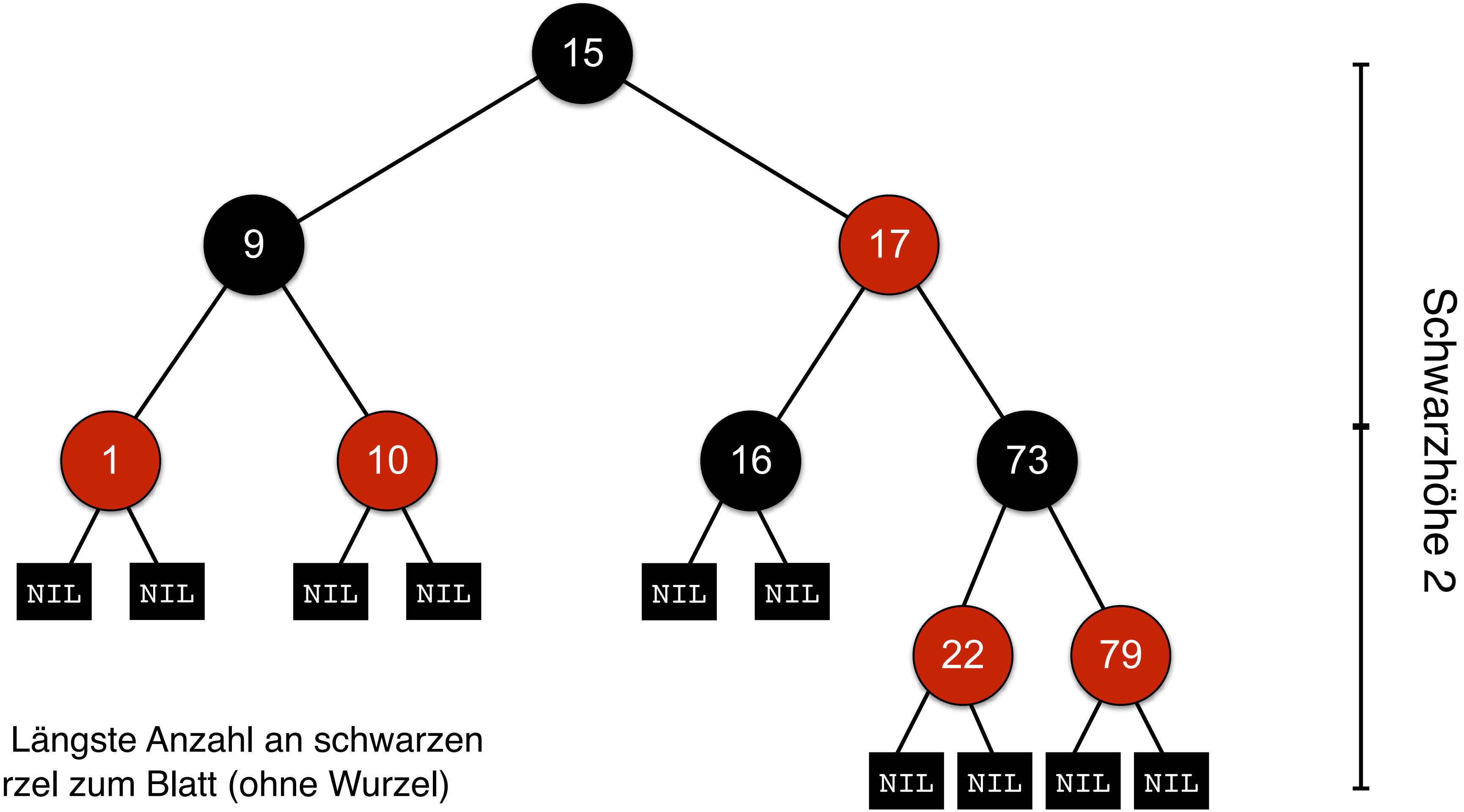
Red-Black Baum - Höhe



Höhe $h =$ Längste Anzahl an Knoten von der Wurzel zum Blatt (ohne Wurzel)

Hochschule Fulda
University of Applied Sciences

Red-Black Baum - Schwarzhöhe



Schwarzhöhe $bh =$ Längste Anzahl an schwarzen Knoten von der Wurzel zum Blatt (ohne Wurzel)

Red-Black Tree - Balanciertheit

Die Höhe eines RBT mit n inneren Knoten ist höchstens $2\lg(n+1)$

Lemma: Teilbaum mit Wurzel x hat mindestens $2^{bh(x)} - 1$ innere Knoten

Beweis durch **vollständige Induktion** über die Höhe $h(x)$

Induktionsanfang: $h(x) = 0$, dann ist x ein Blatt mit 0 inneren Knoten: $2^{bh(x)} - 1 = 2^0 - 1 = 0$.

Induktionsschritt: $h(x) > 0$. Dann hat x zwei Kinder, y und z mit $bh(y), bh(z) \geq bh(x)$. Nach Induktionsannahme ist die Anzahl innerer Knoten des Teilbaums mit Wurzel x mindestens

$$2^{bh(y)} - 1 + 2^{bh(z)} - 1 + 1 \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2 \cdot 2^{hb(x)-1} - 1 = 2^{bh(x)} - 1$$

Red-Black Tree - Balanciertheit

Die Höhe h eines RBT mit n inneren Knoten ist höchstens $2\lg(n+1)$

Lemma: Teilbaum mit Wurzel x hat mindestens $2^{bh(x)} - 1$ innere Knoten

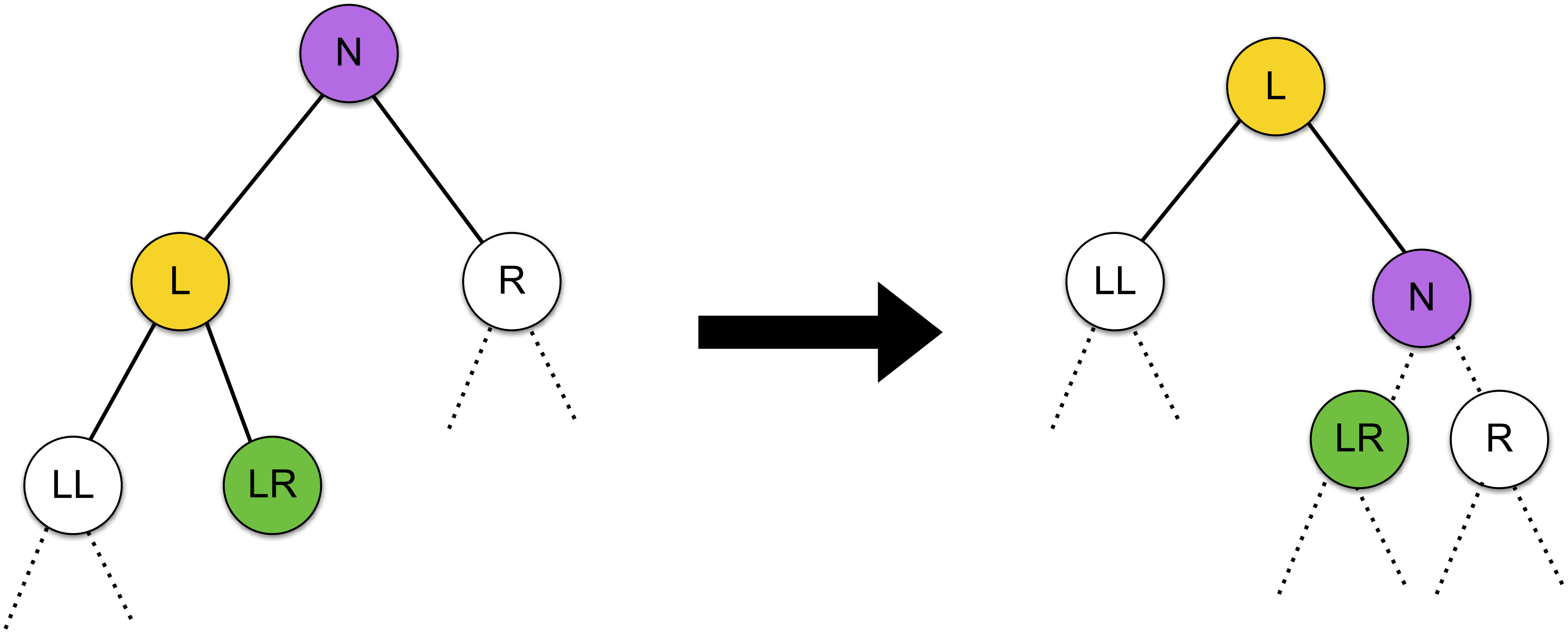
Lemma gilt für jeden Teilbaum, also insbesondere die Wurzel:

$$n \geq 2^{\frac{h}{2}} - 1 \Leftrightarrow n + 1 \geq 2^{\frac{h}{2}} \Leftrightarrow \lg(n+1) \geq \frac{h}{2} \Leftrightarrow h \leq 2\lg(n+1)$$

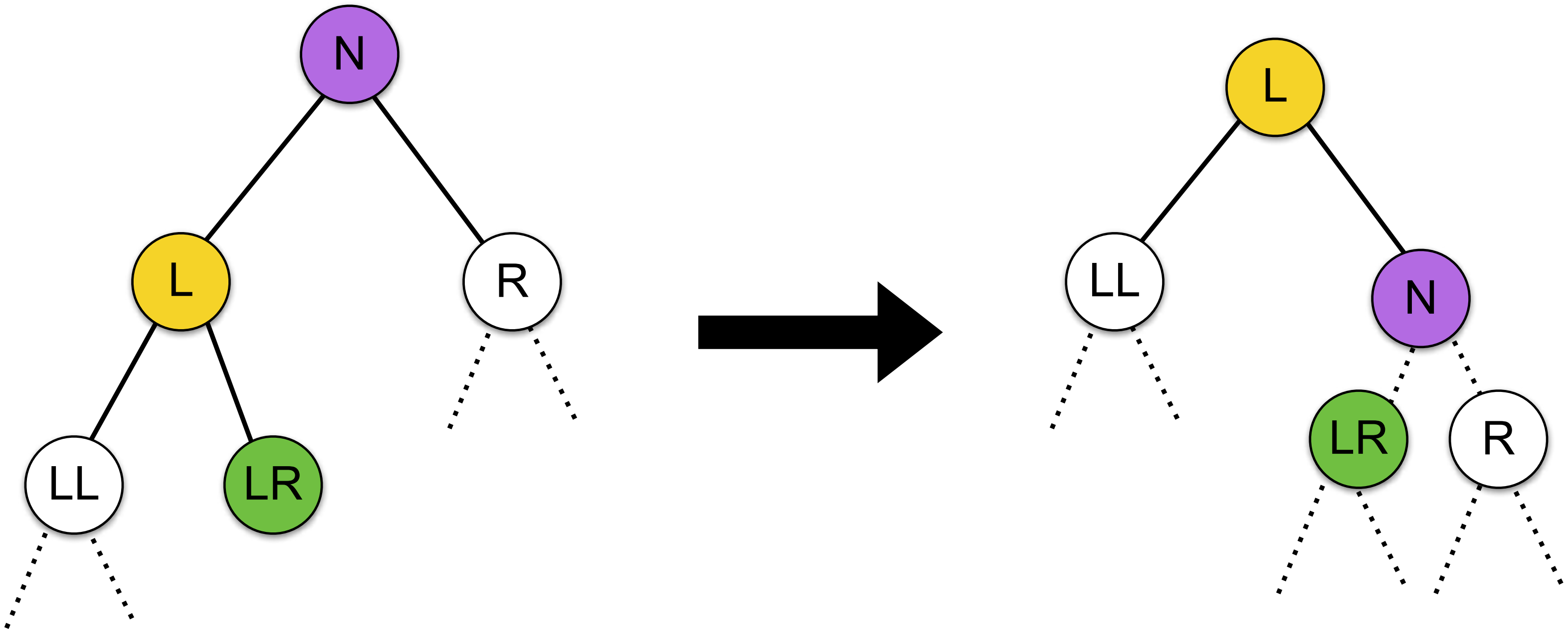
Red Black Tree - Repräsentation in Java

```java
public class Node {
  Comparable data;

  Node left;
  Node right;
  Node parent;

  boolean color;

  public Node(Comparable data) {
    this.data = data;
    left = right = parent = null;
  }
}
```

Rechts-Rotation



Rechts-Rotation - Eigenschaften

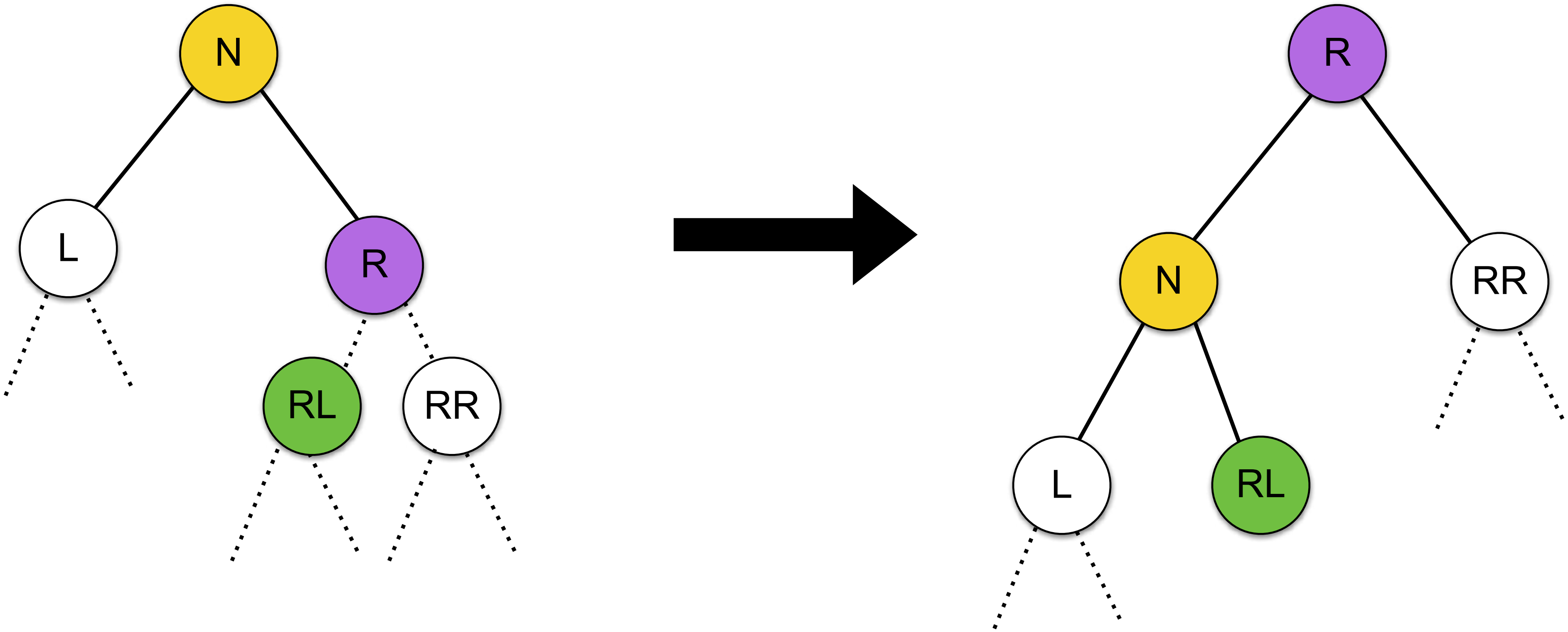


- Der linke Sohn L wird zur neuen Wurzel
- Die Wurzel wird zum rechten Kind
- Der rechte Teilbaum LR wird zum linken Sohn von N nach der Rotation
- LL und R behalten ihre relative Position bei

Rechts-Rotation in Pseudo-Java

```java
private void rotateRight(Node n) {

      Node a1 = n.left;
      Node a2 = n.right;

      n.left = a1.left;
      n.right = a1;

      a1.left = a1.right;
      a1.right = a2;

      Object tmp = a1.data;
      a1.data = n.data;
      n.data = tmp;

}
```

Analog: Linksrotation



Hochschule Fulda
University of Applied Sciences

Einfügen im RBT

- Analog zum Einfügen im Suchbaum:
  - Suche die Einfügeposition von Wurzel abwärts
  - Füge den neuen Knoten ein und repariere die RBT-Eigenschaften

Einfügen im RBT

```java
public void insertNode(int key) {
  Node node = root; Node parent = null;

  // Traverse the tree to the left or right depending on the key
  while (node != null) {
    parent = node;
    if (key < node.data) {
      node = node.left;
    } else if (key > node.data) {
      node = node.right;
    } else {
      throw new IllegalArgumentException("BST already contains a node with key " + key);
    }
  }

  // Insert new node
  Node newNode = new Node(key);
  newNode.color = RED;
  if (parent == null) {
    root = newNode;
  } else if (key < parent.data) {
    parent.left = newNode;
  } else {
    parent.right = newNode;
  }
  newNode.parent = parent;

  fixRedBlackPropertiesAfterInsert(newNode);
}
```

Einfügen im RBT

Regeln

1. Jeder Knoten ist entweder rot oder schwarz
2. Die Wurzel ist schwarz
3. Alle `NIL`-Blätter sind schwarz
4. Ein roter Knoten darf keine roten Kinder haben
5. Alle Pfade von einem Knoten zu den Blättern enthalten gleich viele schwarze Knoten

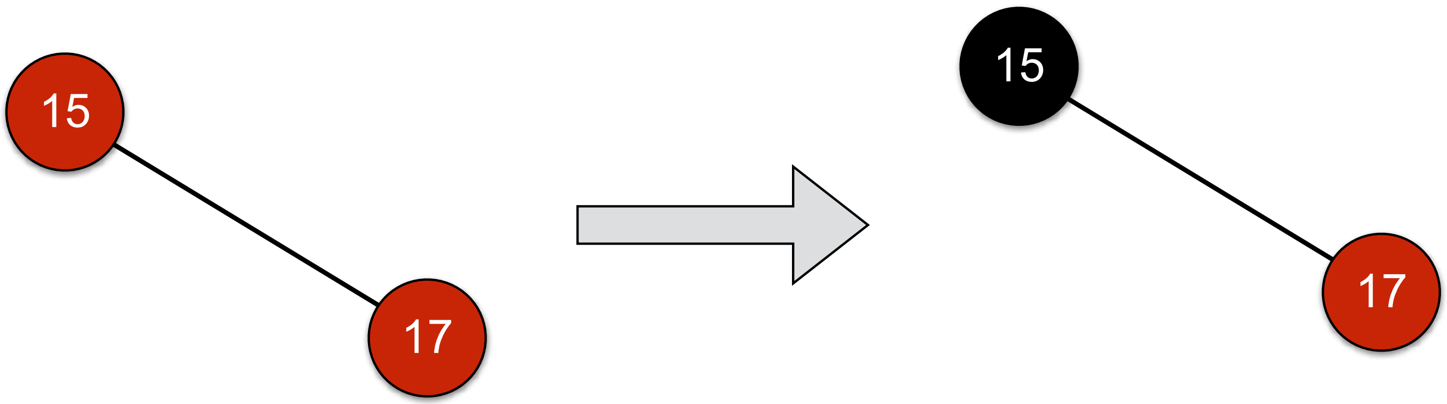
- Wir haben den neuen Knoten rot eingefärbt, um Regel 5 einzuhalten
- Aber falls der Vater des neuen Knotens ebenfalls rot ist, haben wir Regel 4 verletzt
- Dann müssen wir durch Rotationen und Umfärben den Baum reparieren, so dass alle Regeln wieder erfüllt sind

Einfügen im RBT

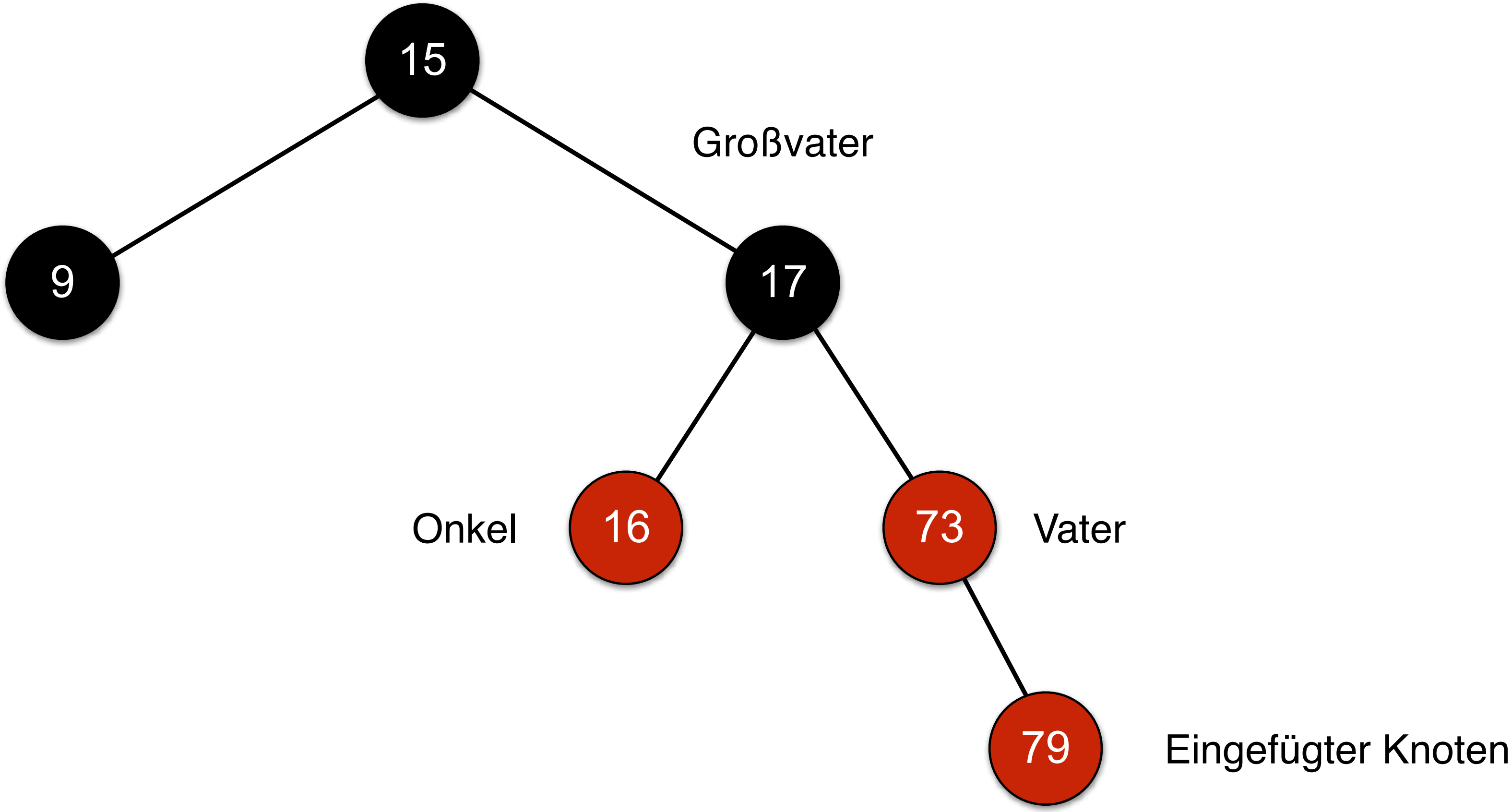
- Während der Reparatur müssen 5 Fälle unterschieden werden:
  - Fall 1: Der neue Knoten ist die Wurzel
  - Fall 2: Der Vater ist die Wurzel und rot
  - Fall 3: Vater und Onkelknoten sind rot
  - Fall 4: Vater ist rot, Onkel ist schwarz, Knoten ist “innerer Enkel”
  - Fall 5: Vater ist rot, Onkel ist schwarz, Knoten ist “äußerer Enkel”

Fall 1 und 2: Der Knoten ist die neue Wurzel

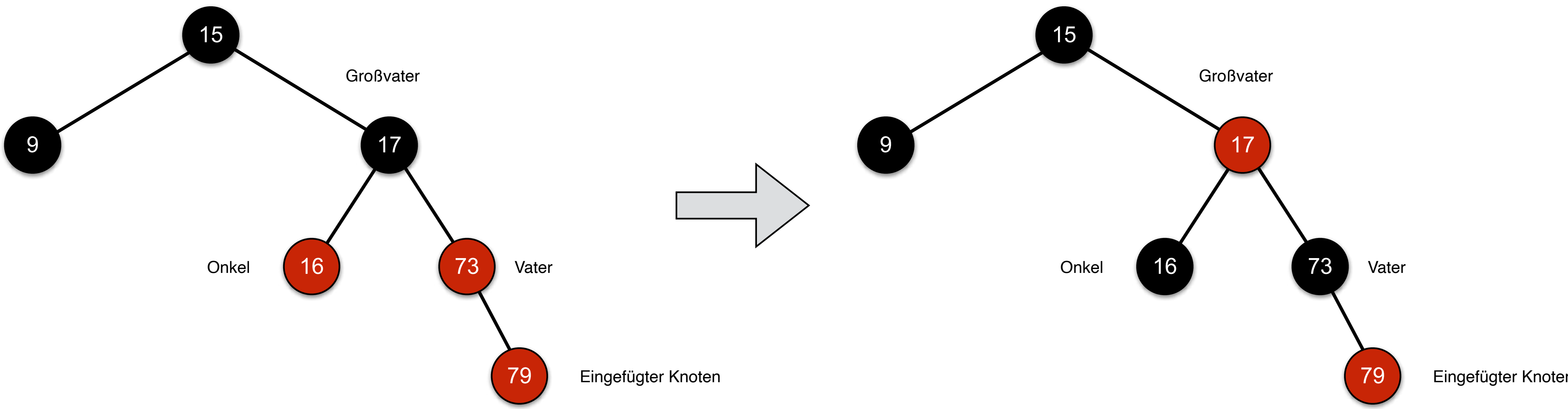
- Falls die alte Wurzel rot ist muss umgefärbt werden. Fertig



Fall 3: Vater und Onkel sind rot

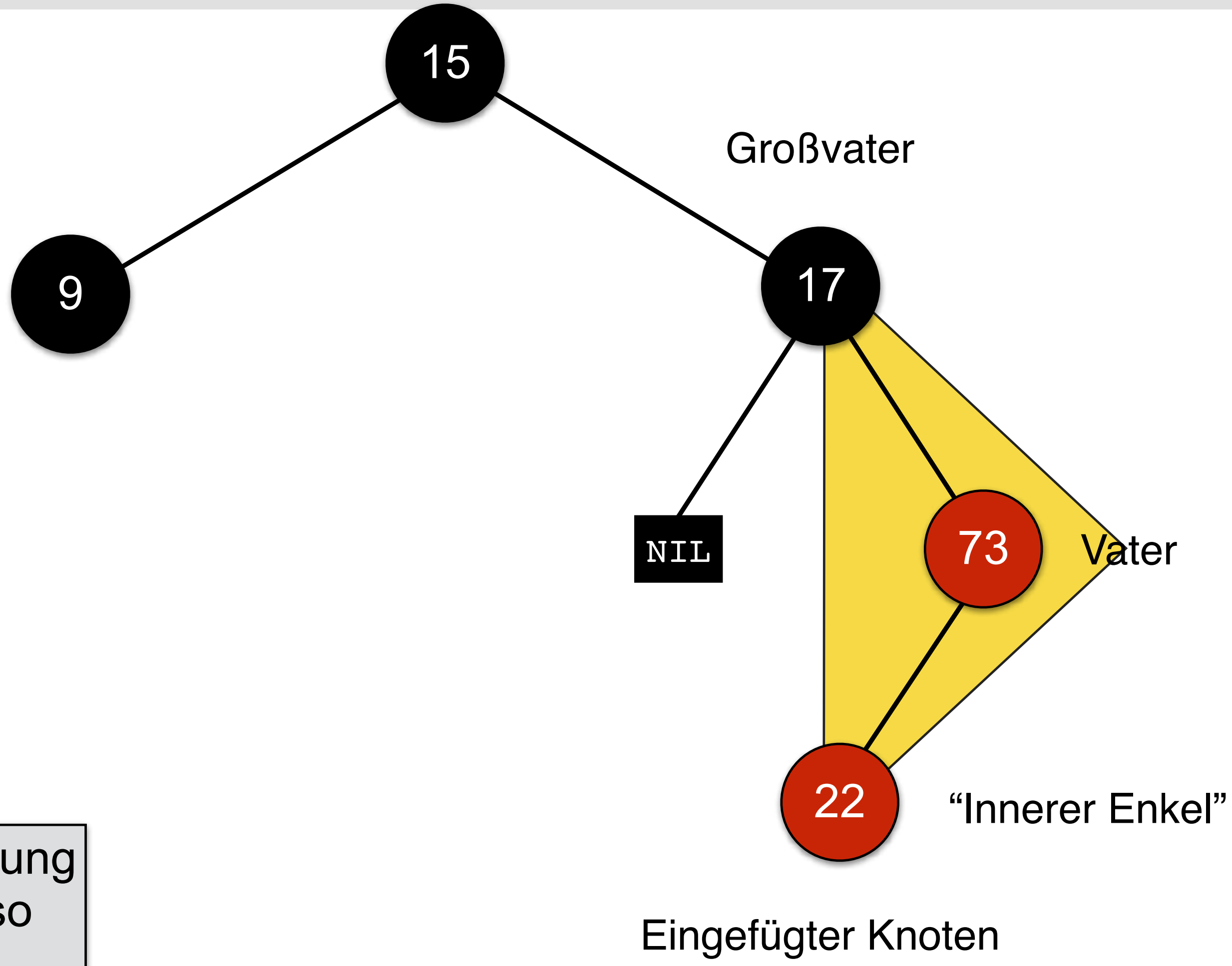


Fall 3: Vater und Onkel sind rot



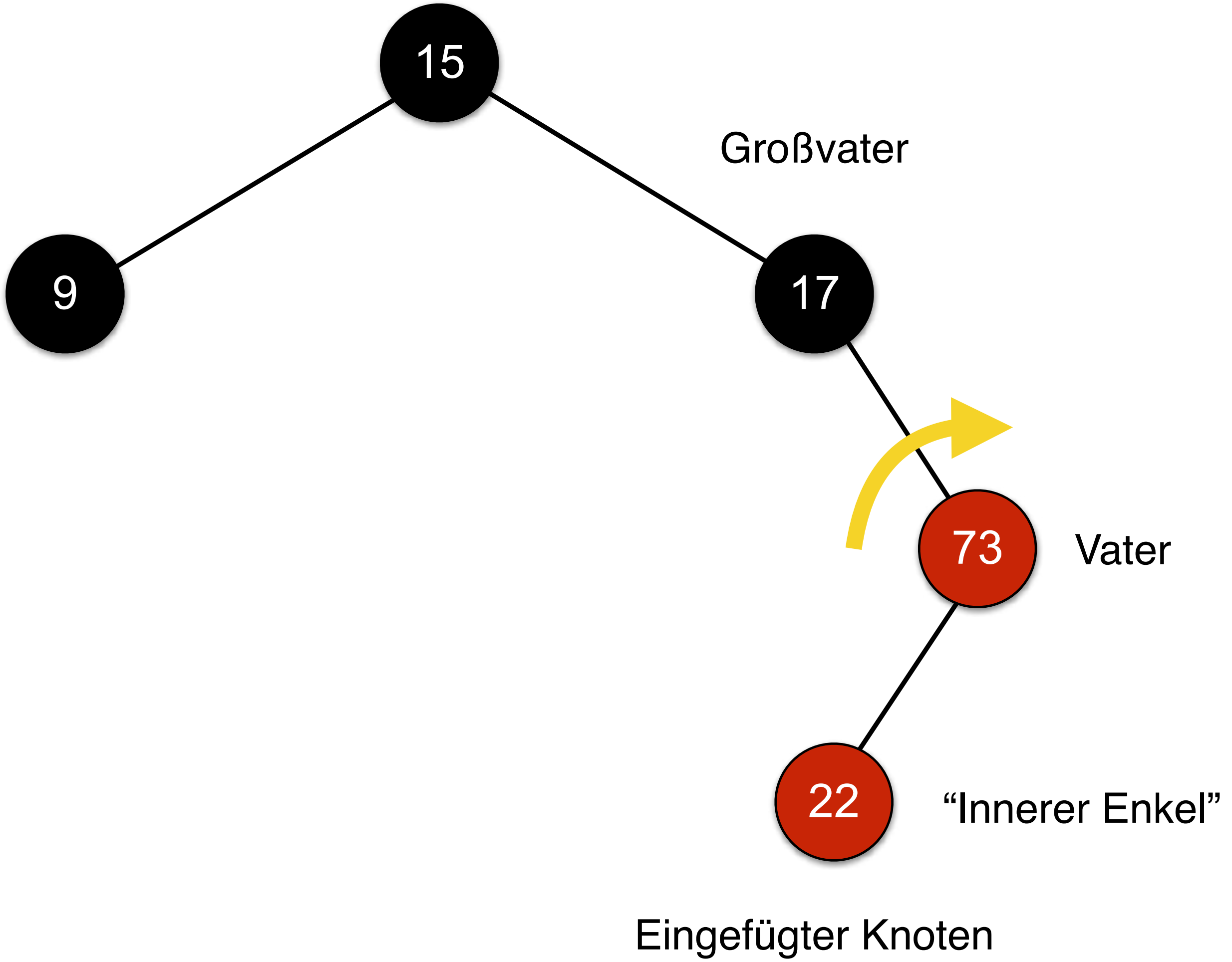
- Vater und Onkel werden schwarz gefärbt und der Großvater rot
- Falls der Urgroßvater auch rot war, repariere rekursiv

Fall 4: Vater rot, innerer Enkel

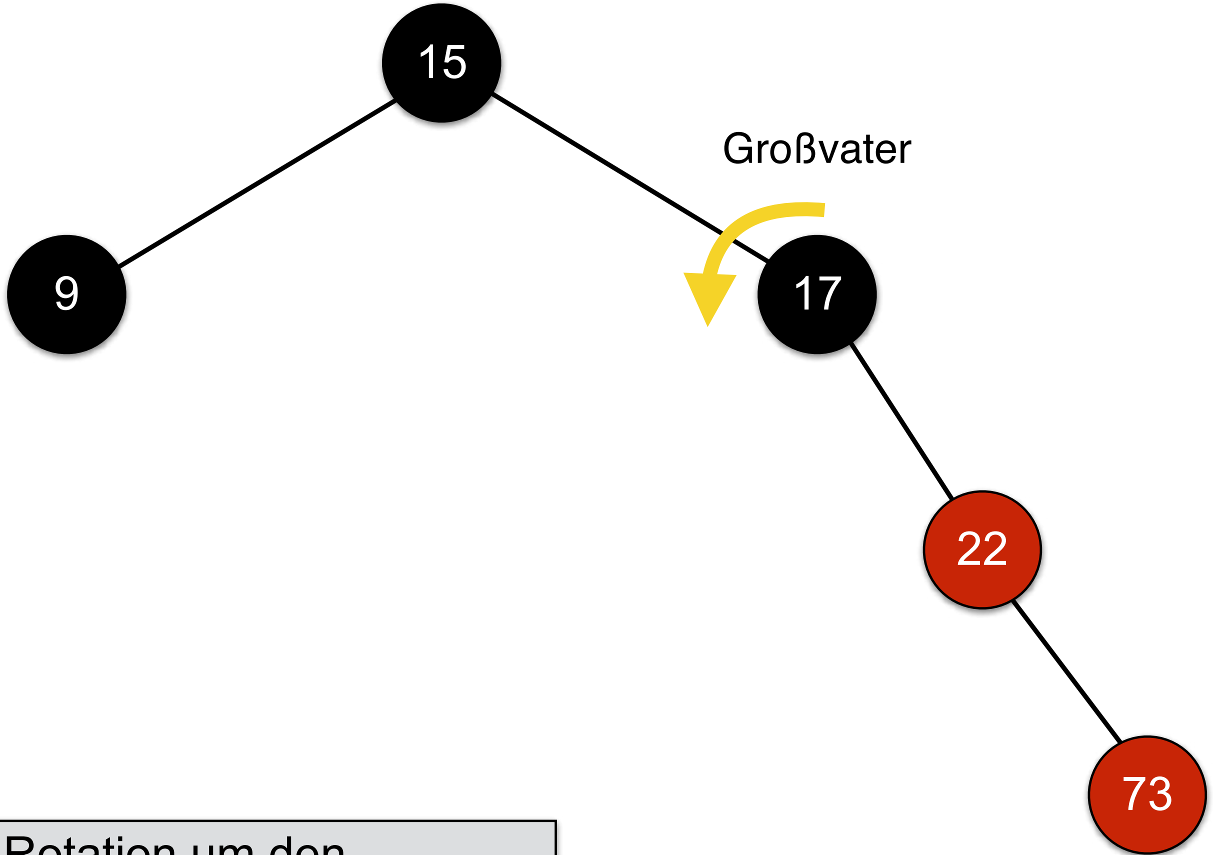


1. Schritt: Rotation in Gegenrichtung des eingefügten Knotens. Hier also nach rechts weil linker Knoten!

Fall 4: Vater rot, innerer Enkel

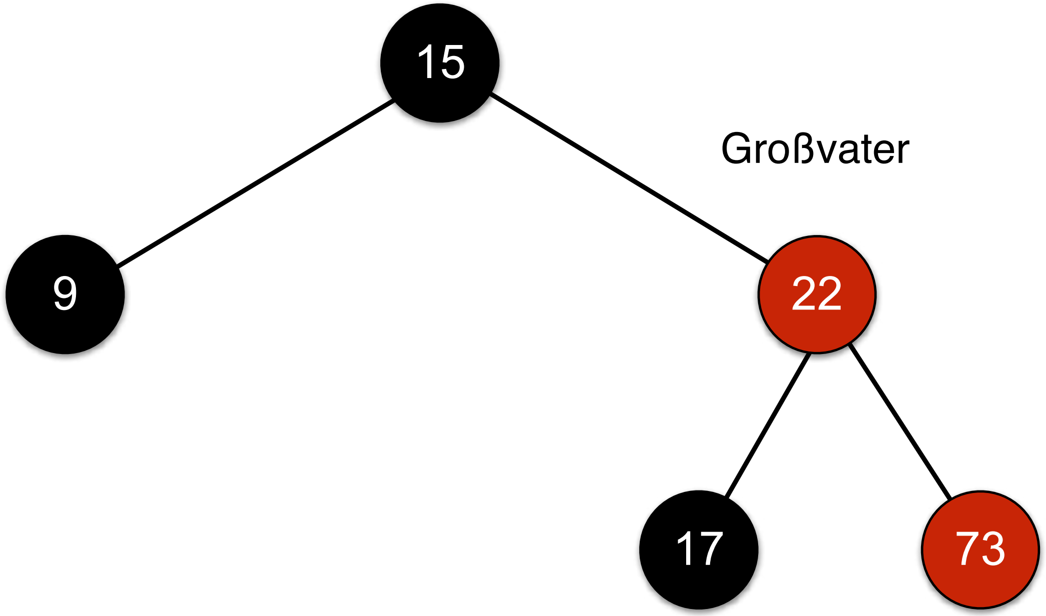


Fall 4: Vater rot, innerer Enkel



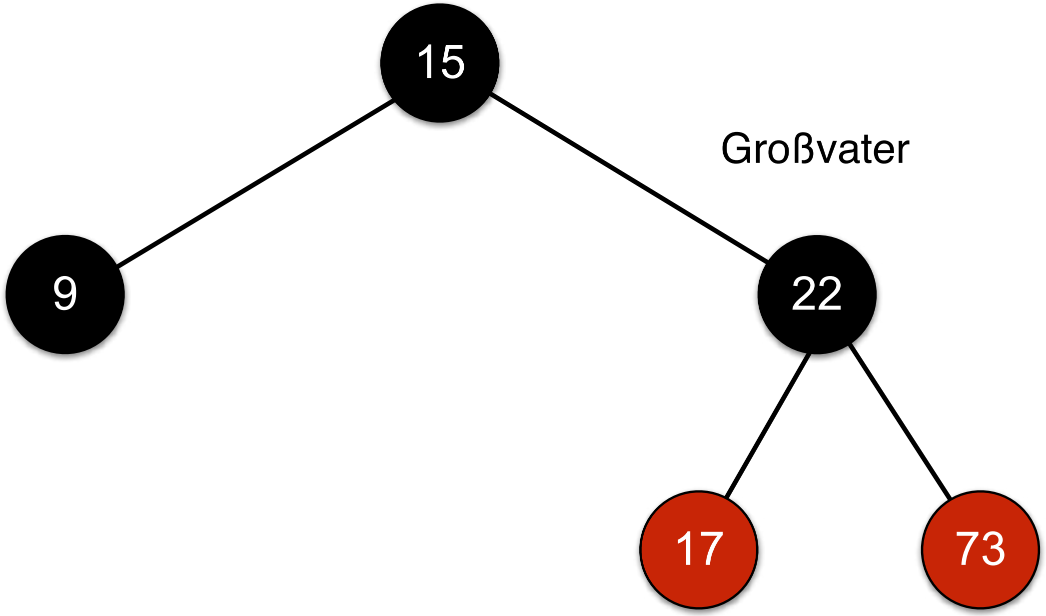
2. Schritt: Rotation um den Großvater in Gegenrichtung

Fall 4: Vater rot, innerer Enkel



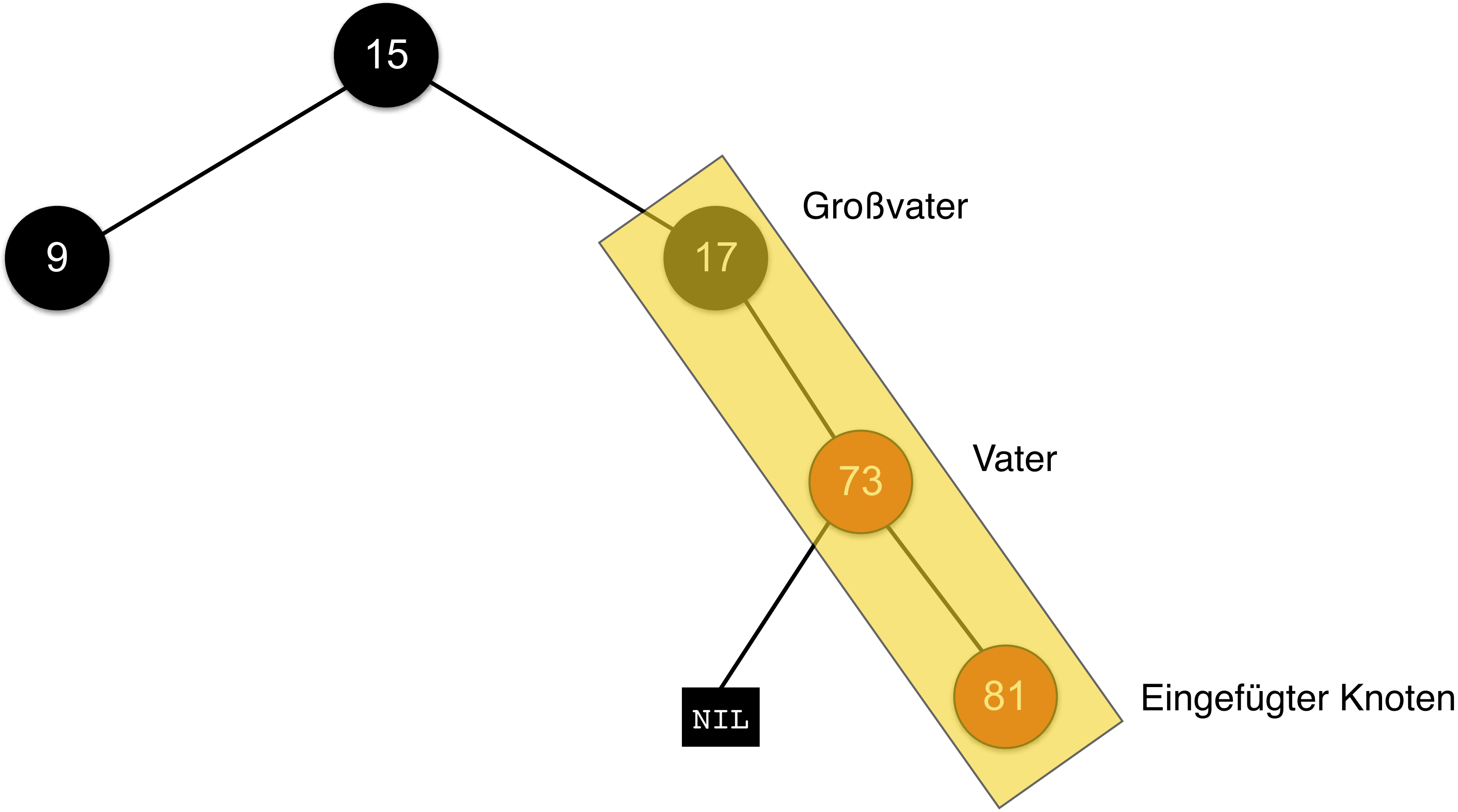
3. Schritt: Neuer Knoten wird rot und alter Großvater schwarz

Fall 4: Vater rot, innerer Enkel

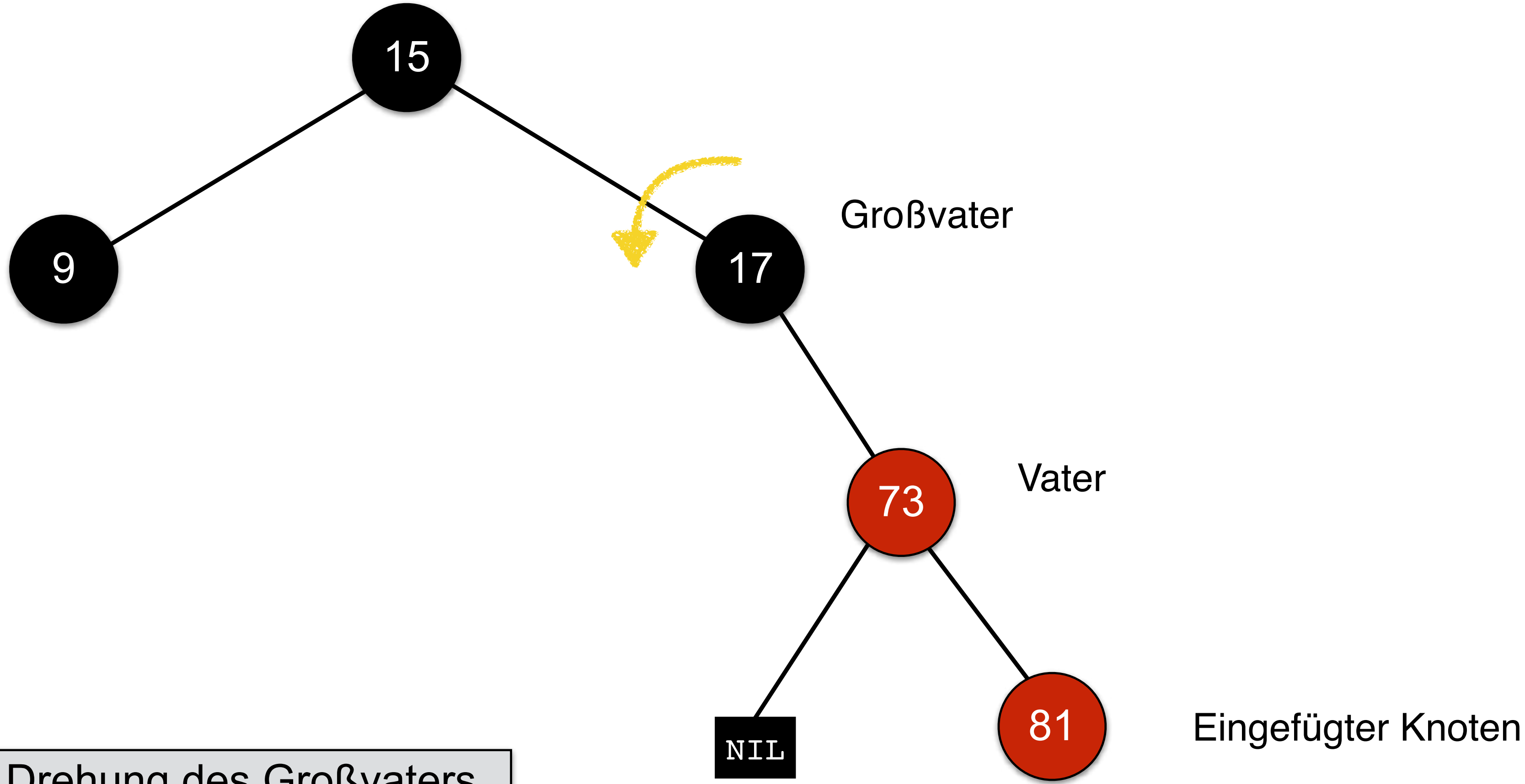


- Weil jetzt ein schwarzer Knoten die Wurzel des zuletzt rotierten Teilbaums ist, kann Regel 4 (kein rot-rot) nicht verletzt sein
- Ebenso kann das Umfärben des Großvaters Regel 4 nicht verletzen
- Die Reparatur ist abgeschlossen, keine Rekursion erforderlich

Fall 5: Vater rot, Onkel schwarz, äußerer Enkel

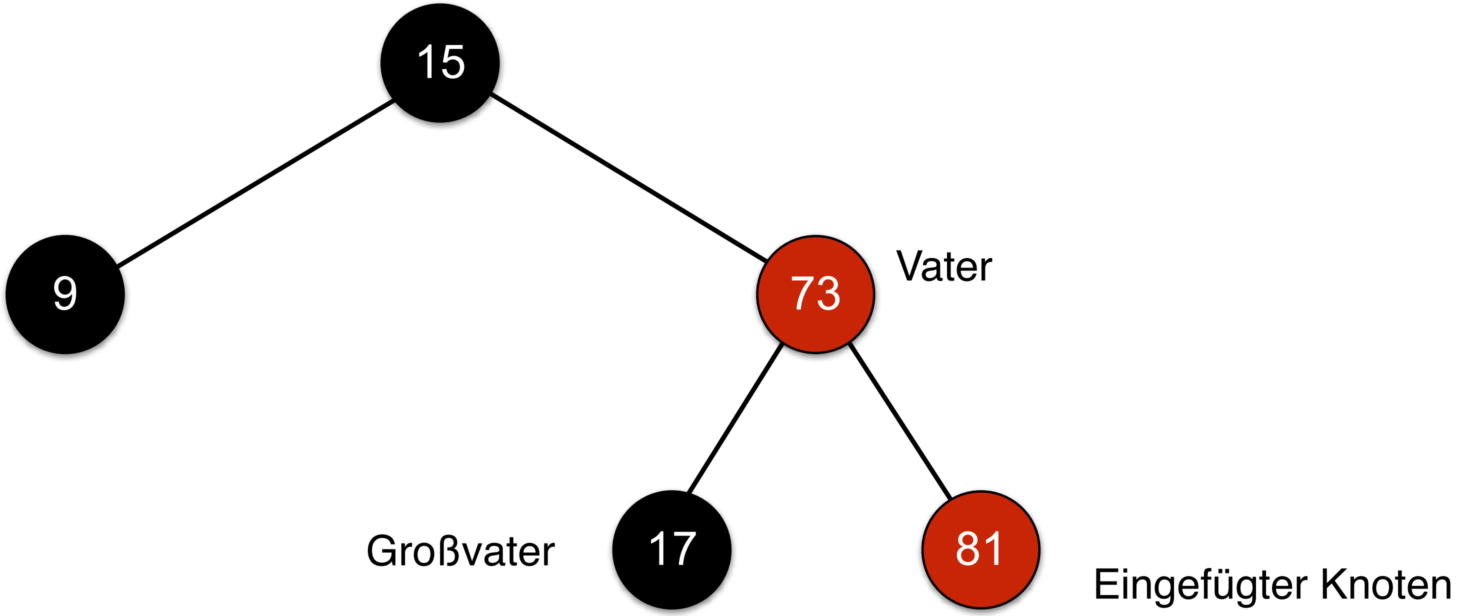


Fall 5: Vater rot, Onkel schwarz, äußerer Enkel

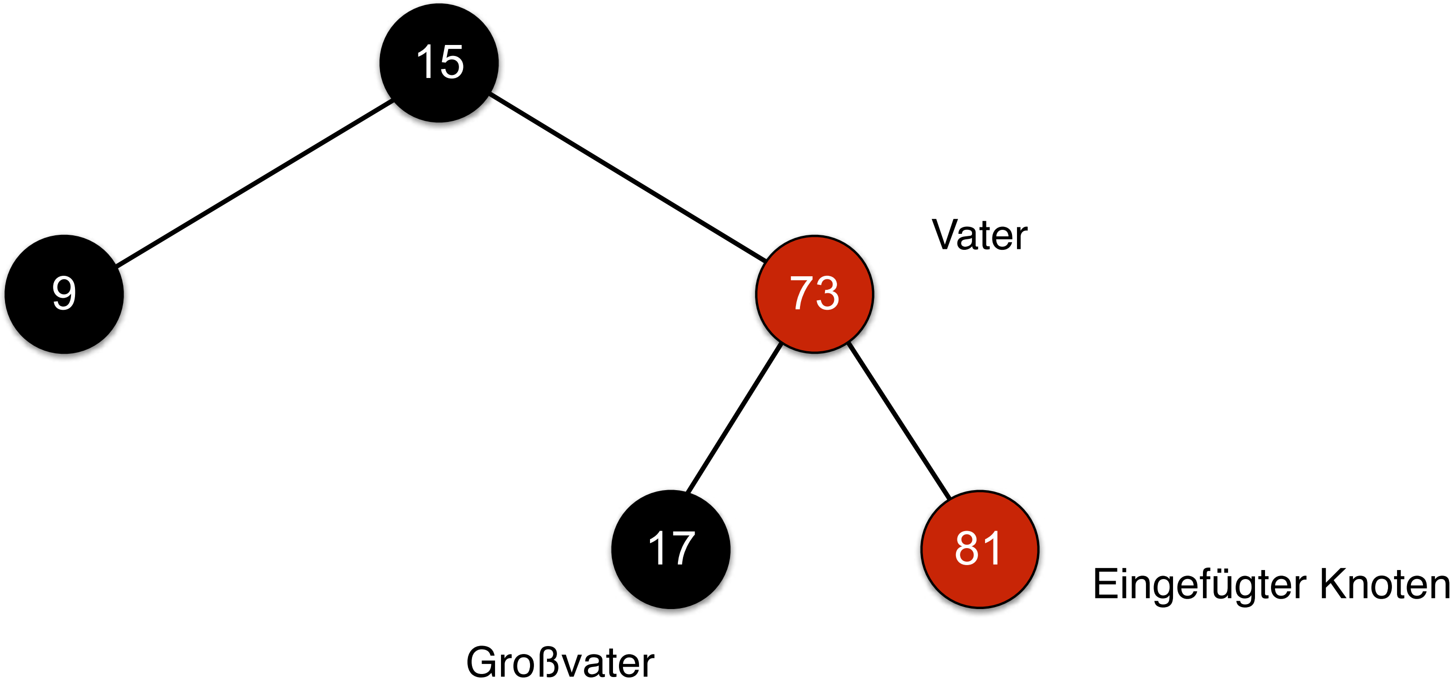


1. Schritt: Drehung des Großvaters in Gegenrichtung (links)

Fall 5: Vater rot, Onkel schwarz, äußerer Enkel

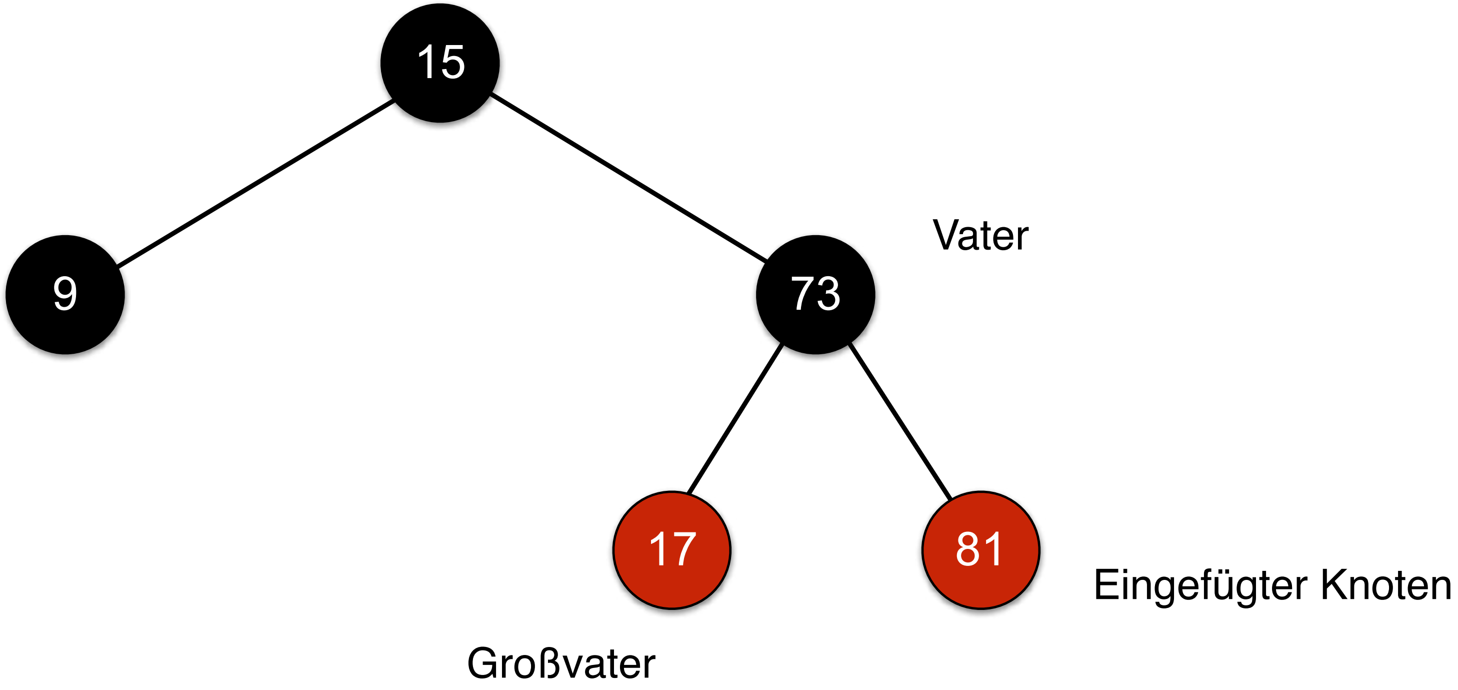


Fall 5: Vater rot, Onkel schwarz, äußerer Enkel



2. **Schritt:** Vater wird schwarz, Großvater rot

Fall 5: Vater rot, Onkel schwarz, äußerer Enkel



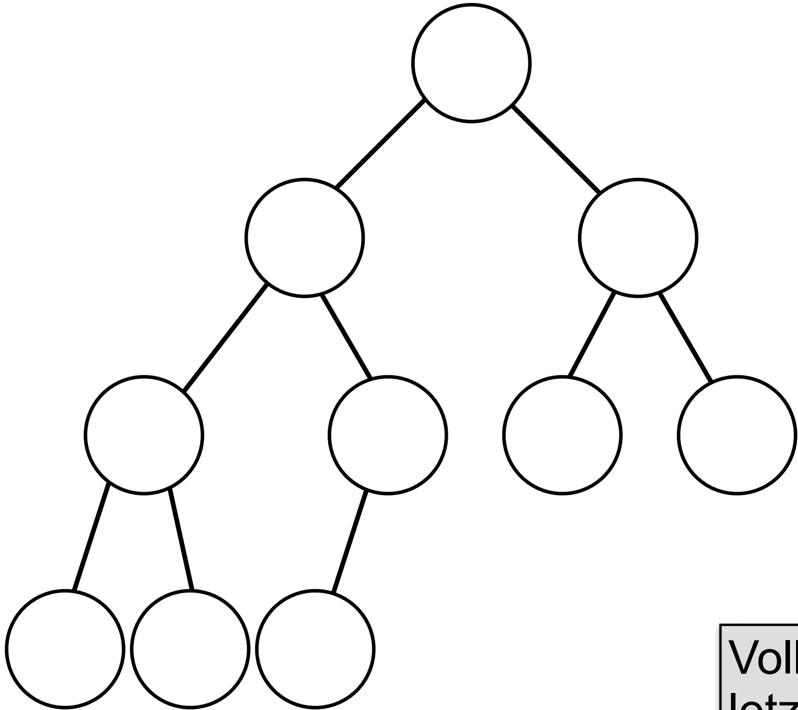
- Am Ende selbe Situation wie in Fall 4
- In der Implementierung muss man nur die erste Rotation ausführen und kann dann in den Code von Fall 4 springen

Animation RBT

- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Vollständiger Binärbaum

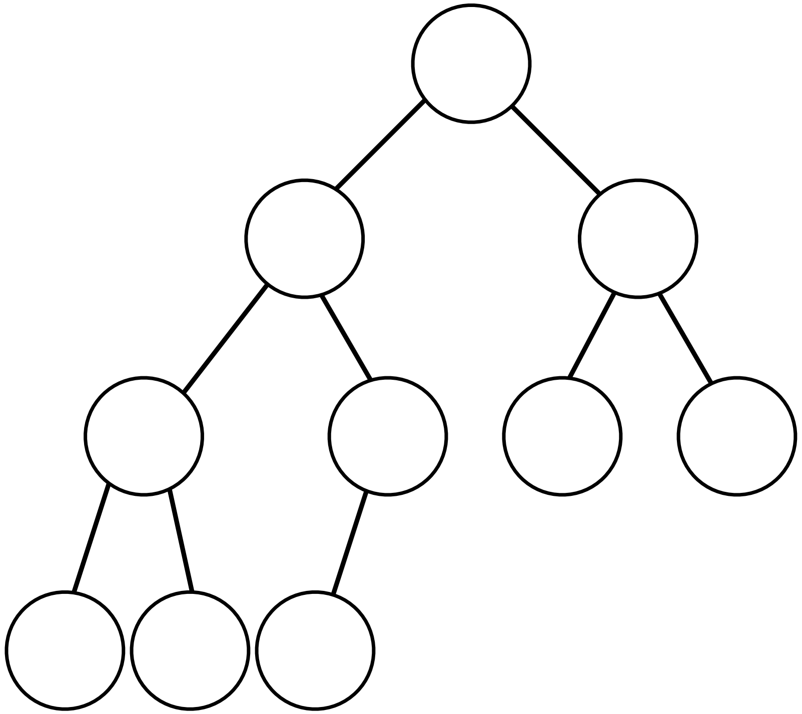
- Jeden Knoten sind maximal zwei binäre Unterbäume zugeordnet



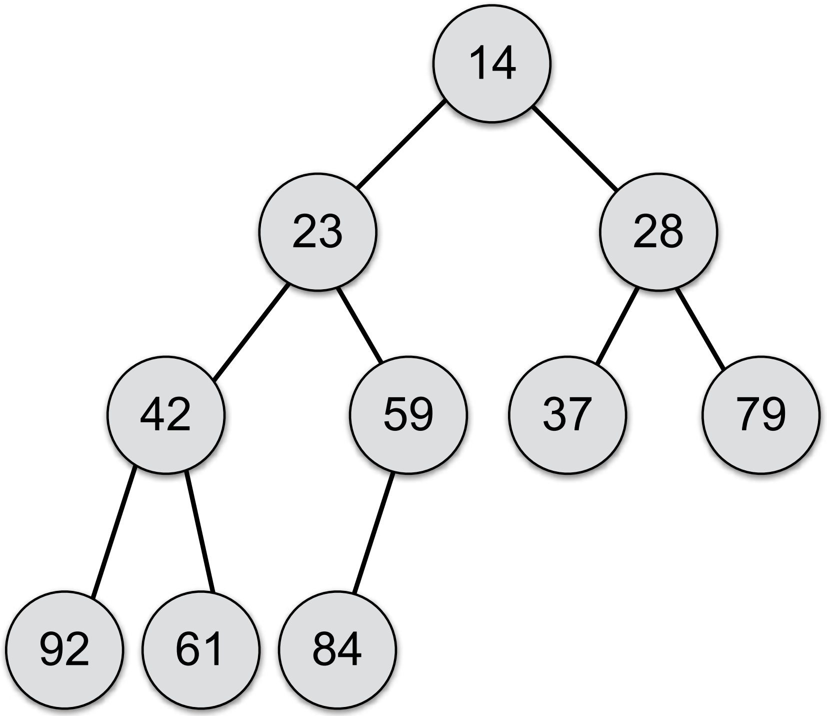
Vollständigkeit: Alle Ebenen bis auf die letzte müssen komplett gefüllt sein. Die letzte muss von links beginnend durchgehend gefüllt.

Heap

- Vollständiger Binärbaum
- Alle Knoten enthalten Schlüssel
- Schlüssel im Vater $\leq$ Schlüssel in den Söhnen (Min-Heap), $\geq$ für Max-Heap

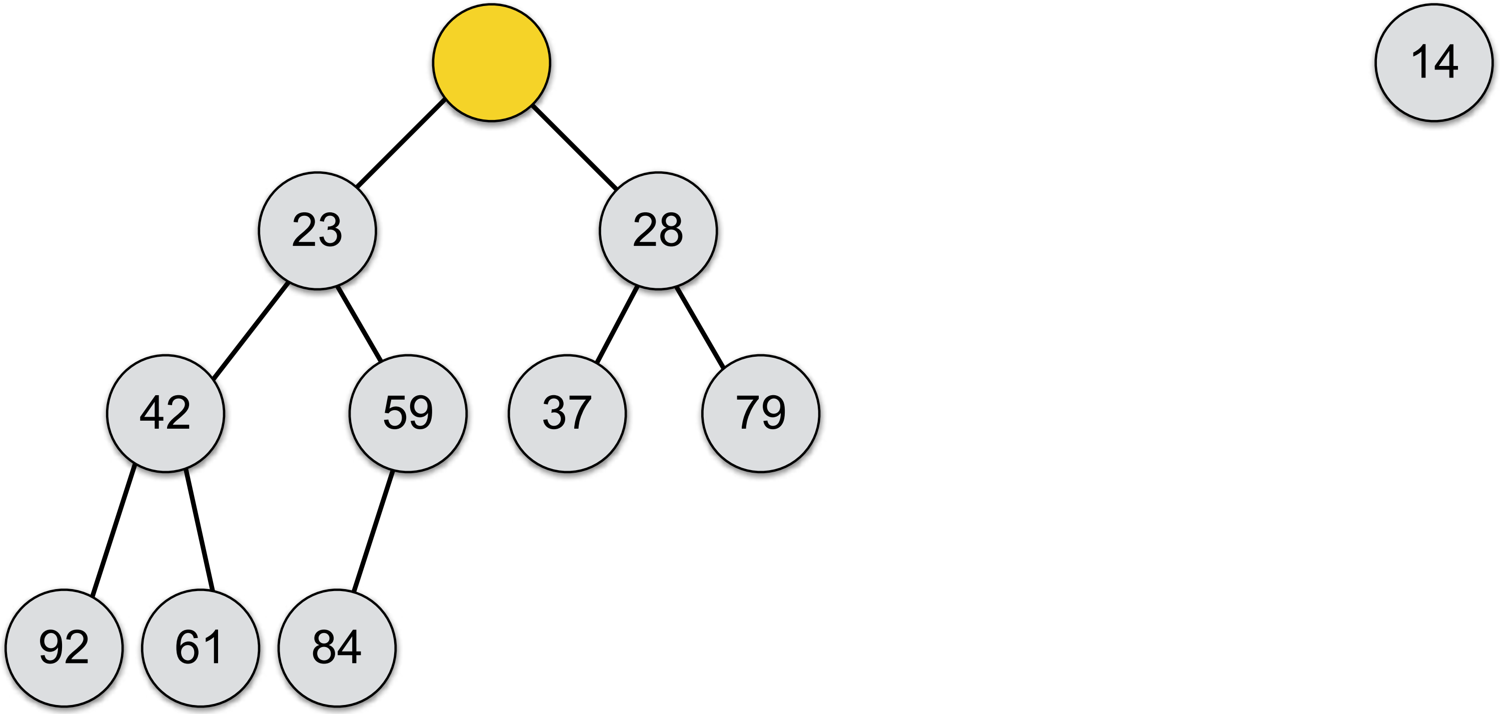


Beispiel für einen Heap



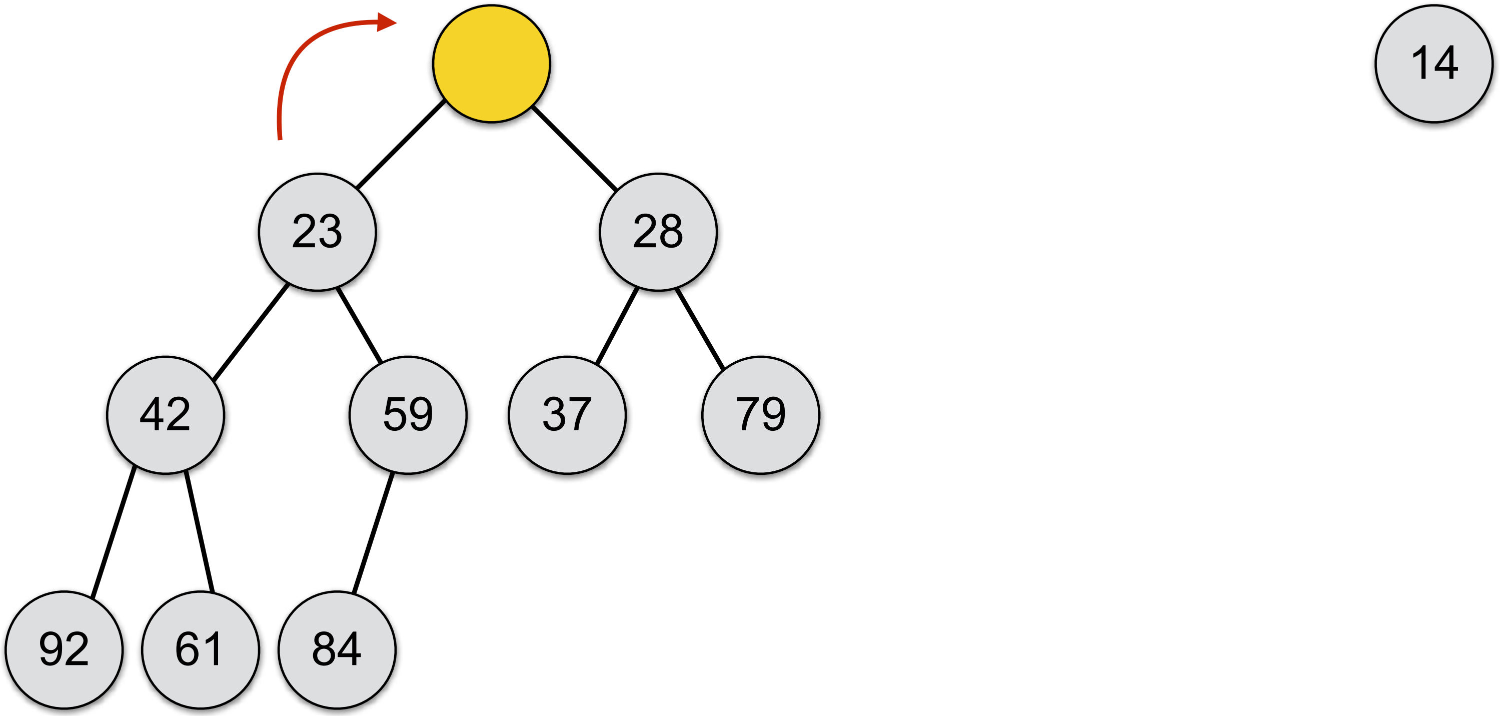
Heap zum sortieren, wie?

Heapsort



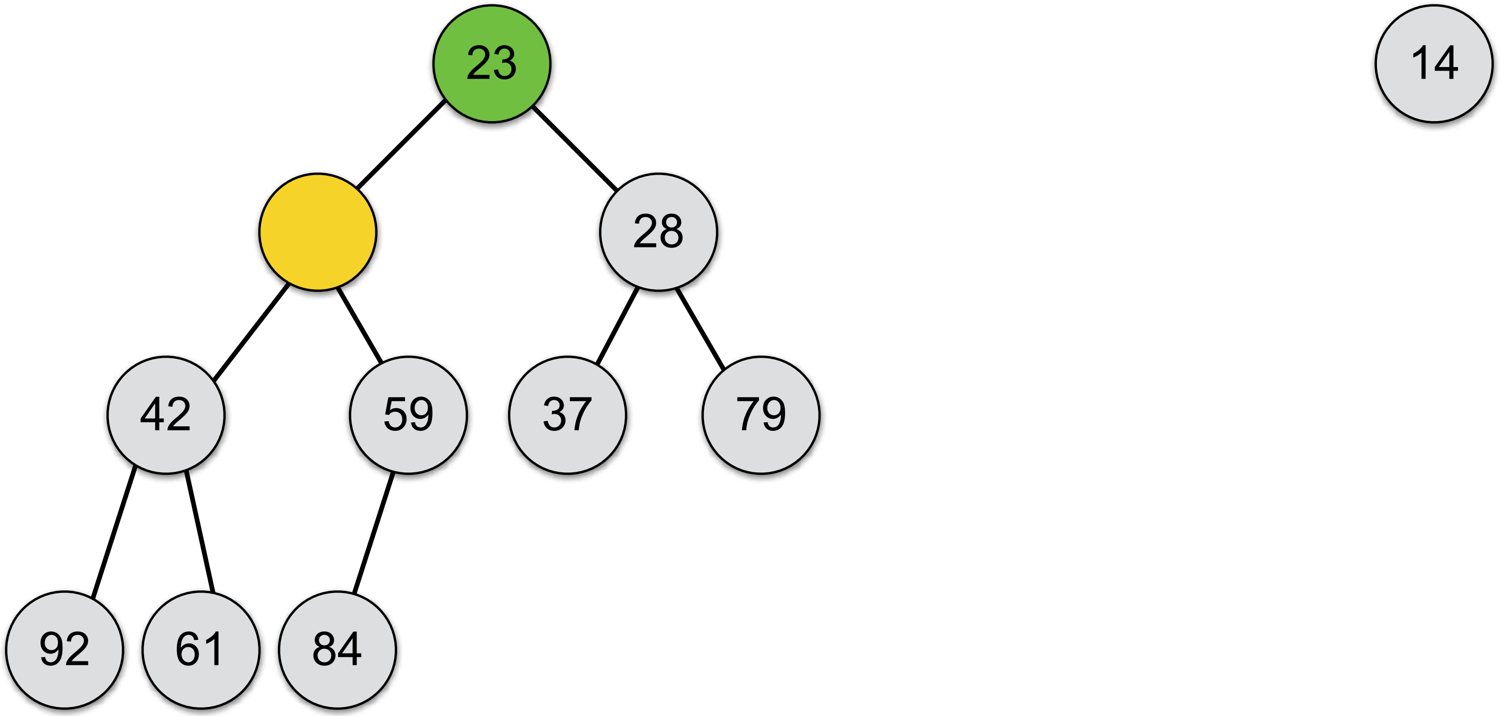
Idee zum reparieren?

Heapsort

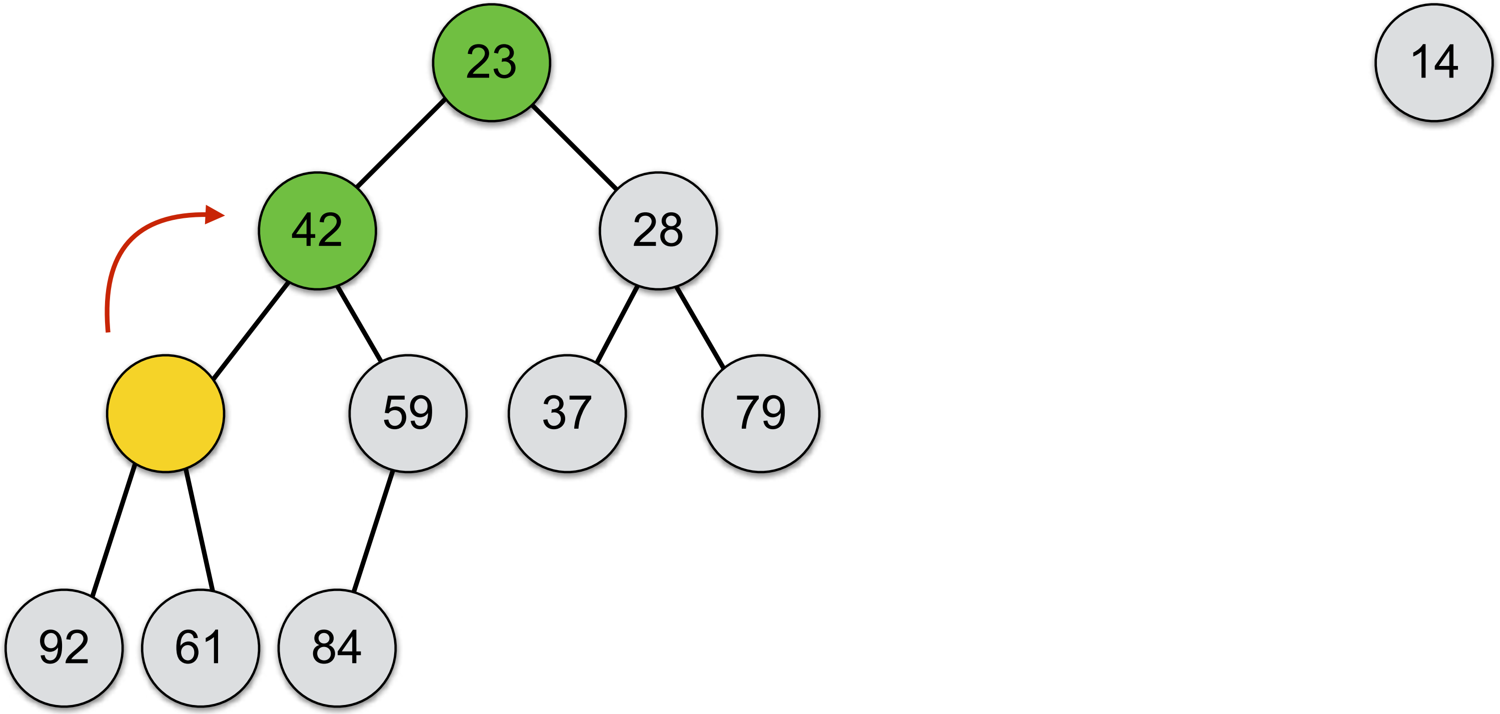


Idee: Ziehe den kleineren Sohn nach oben

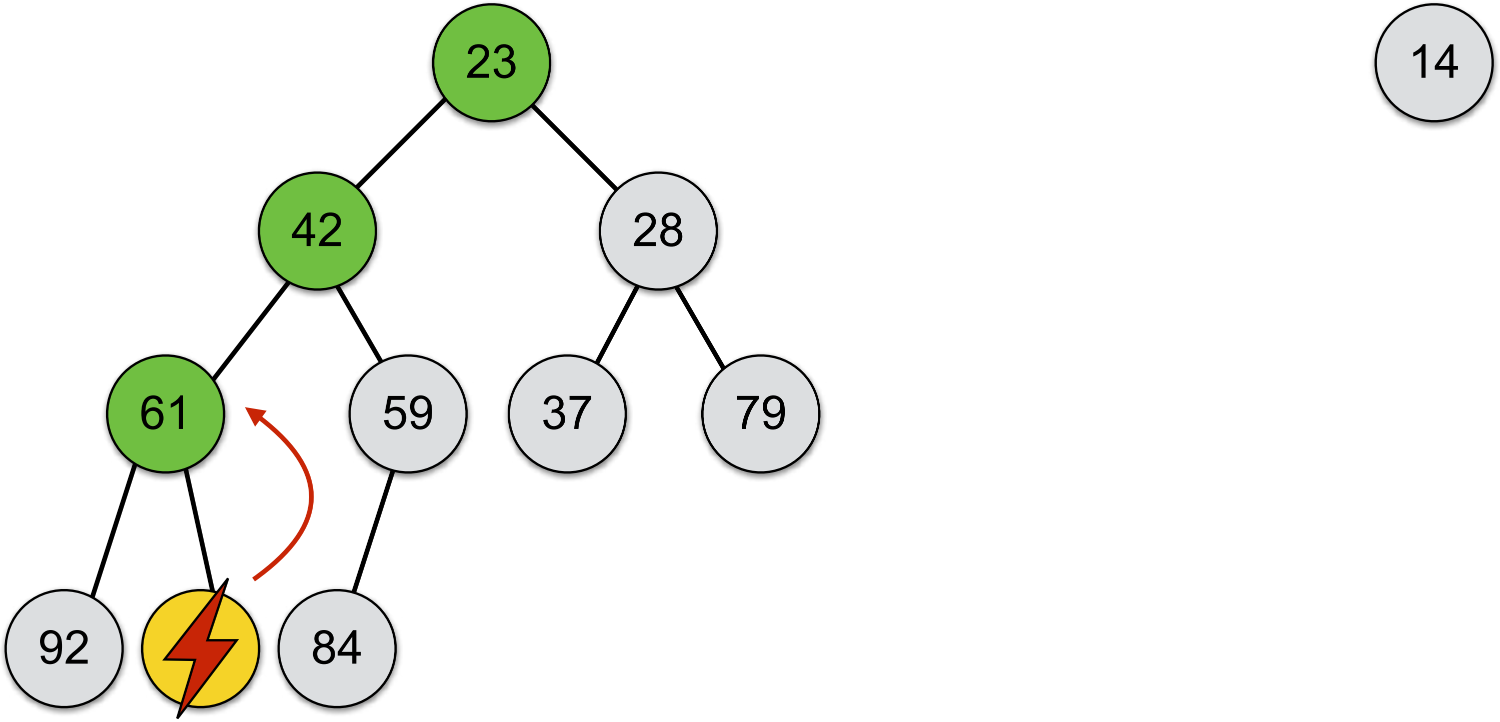
Heapsort



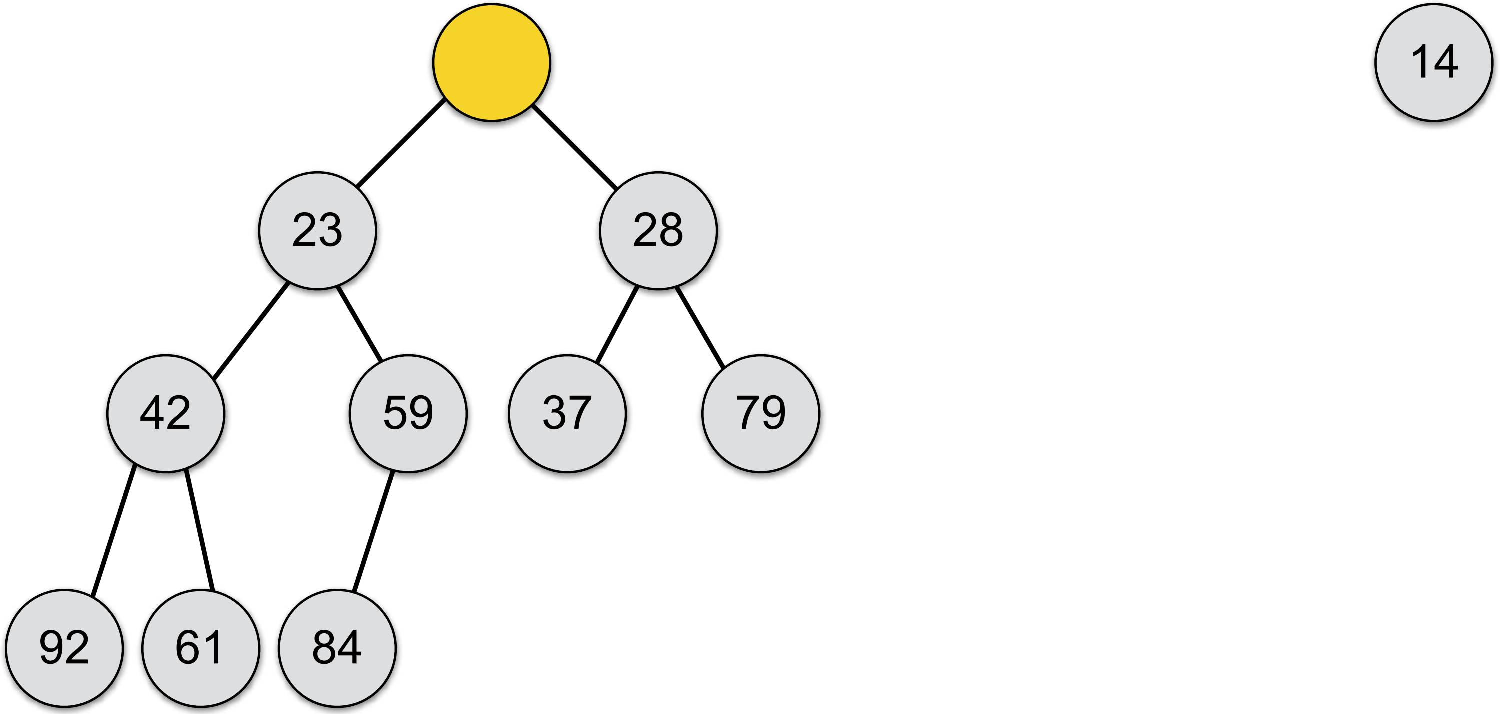
Heapsort



Heapsort

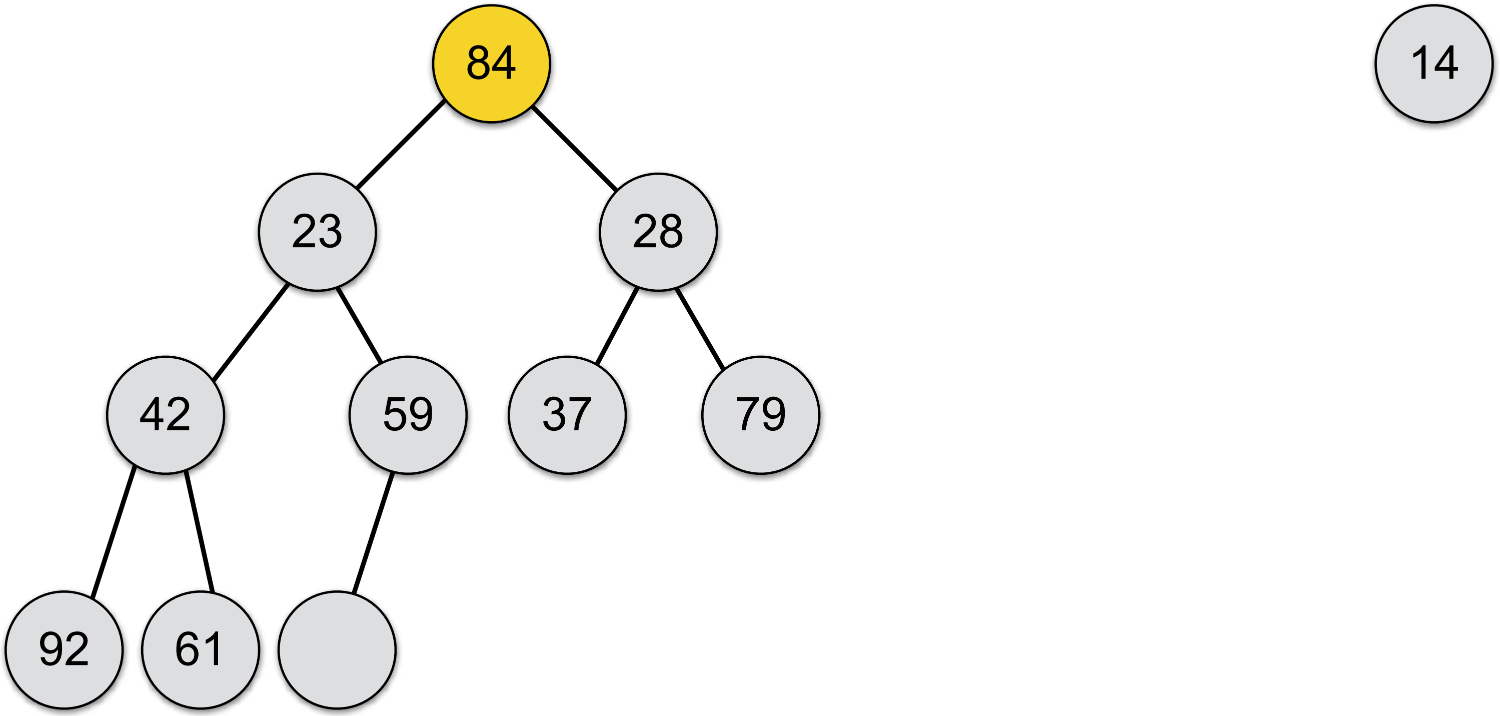


Heapsort



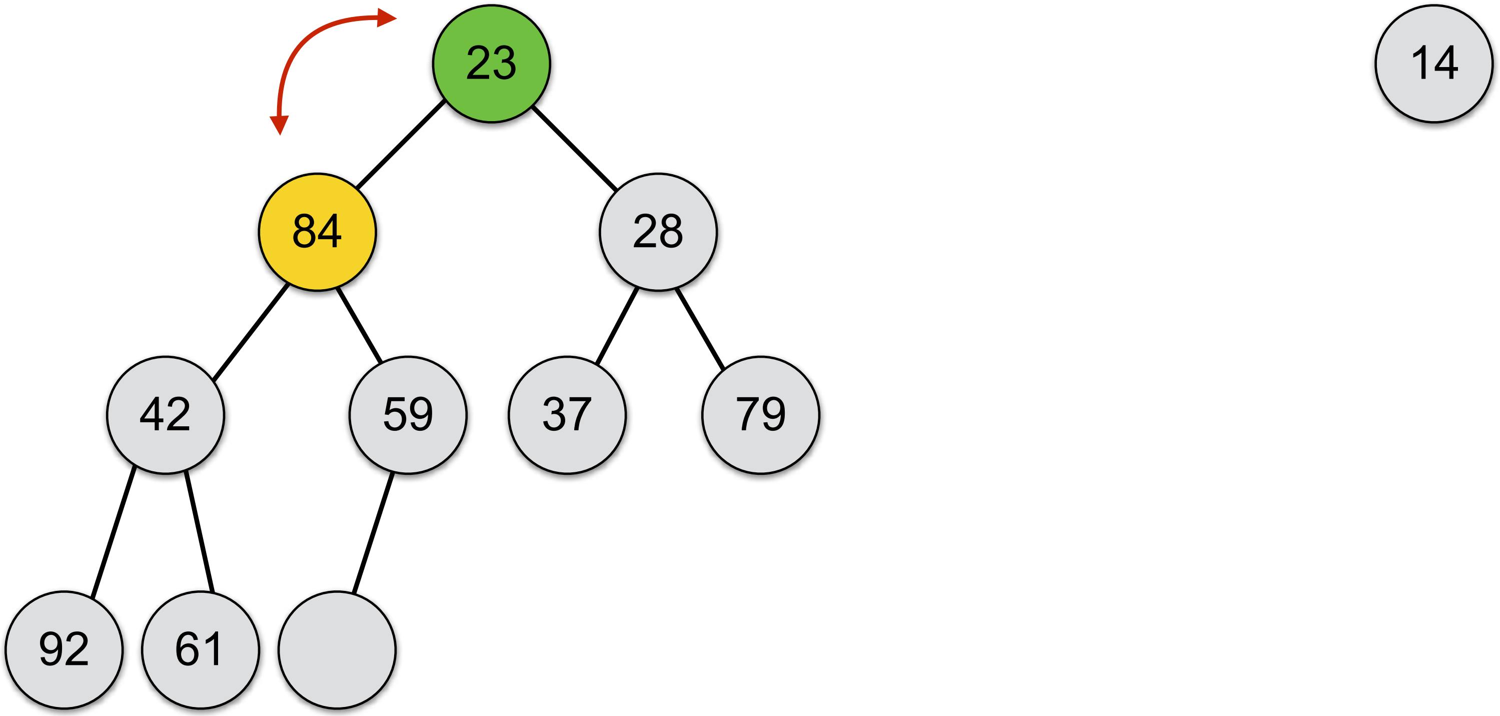
Idee zum reparieren?

Heapsort

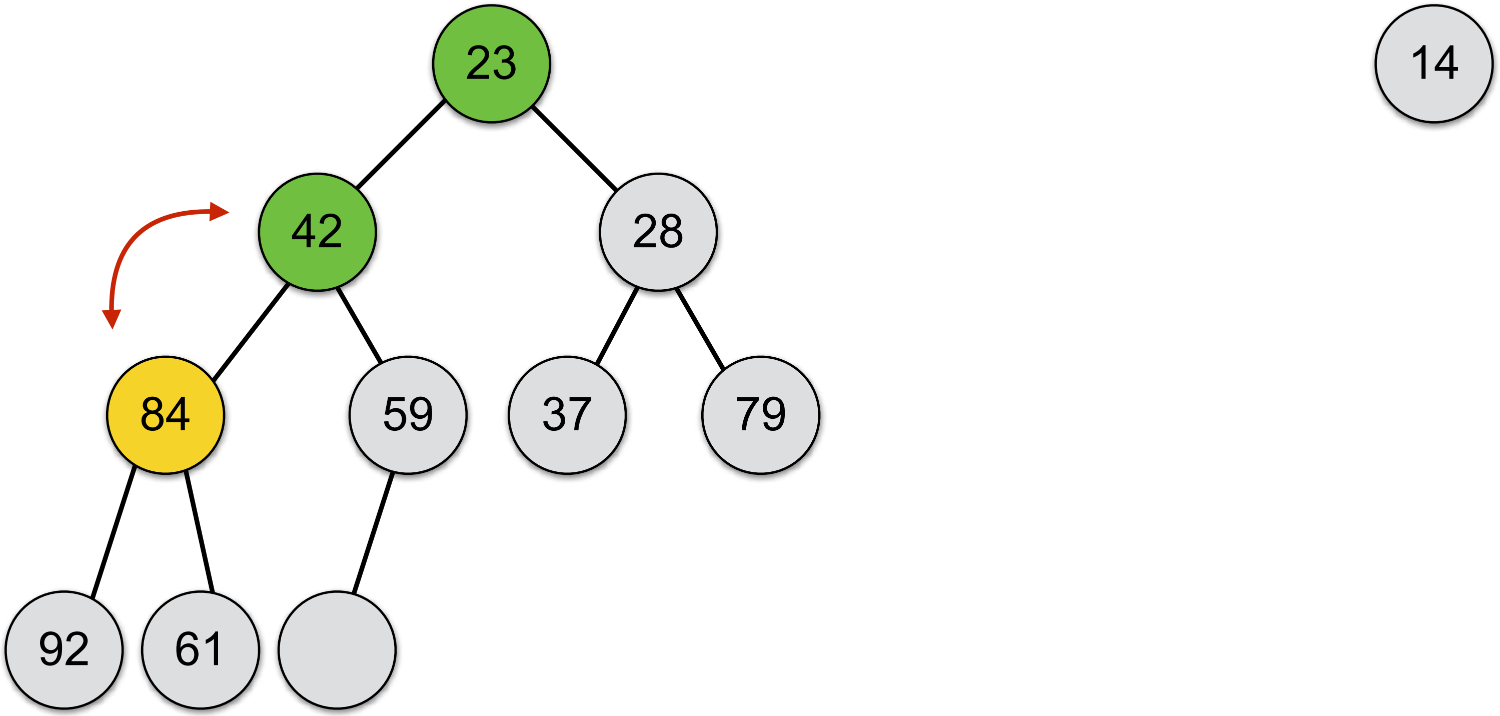


Hole das letzte Element der unteren Ebene nach oben

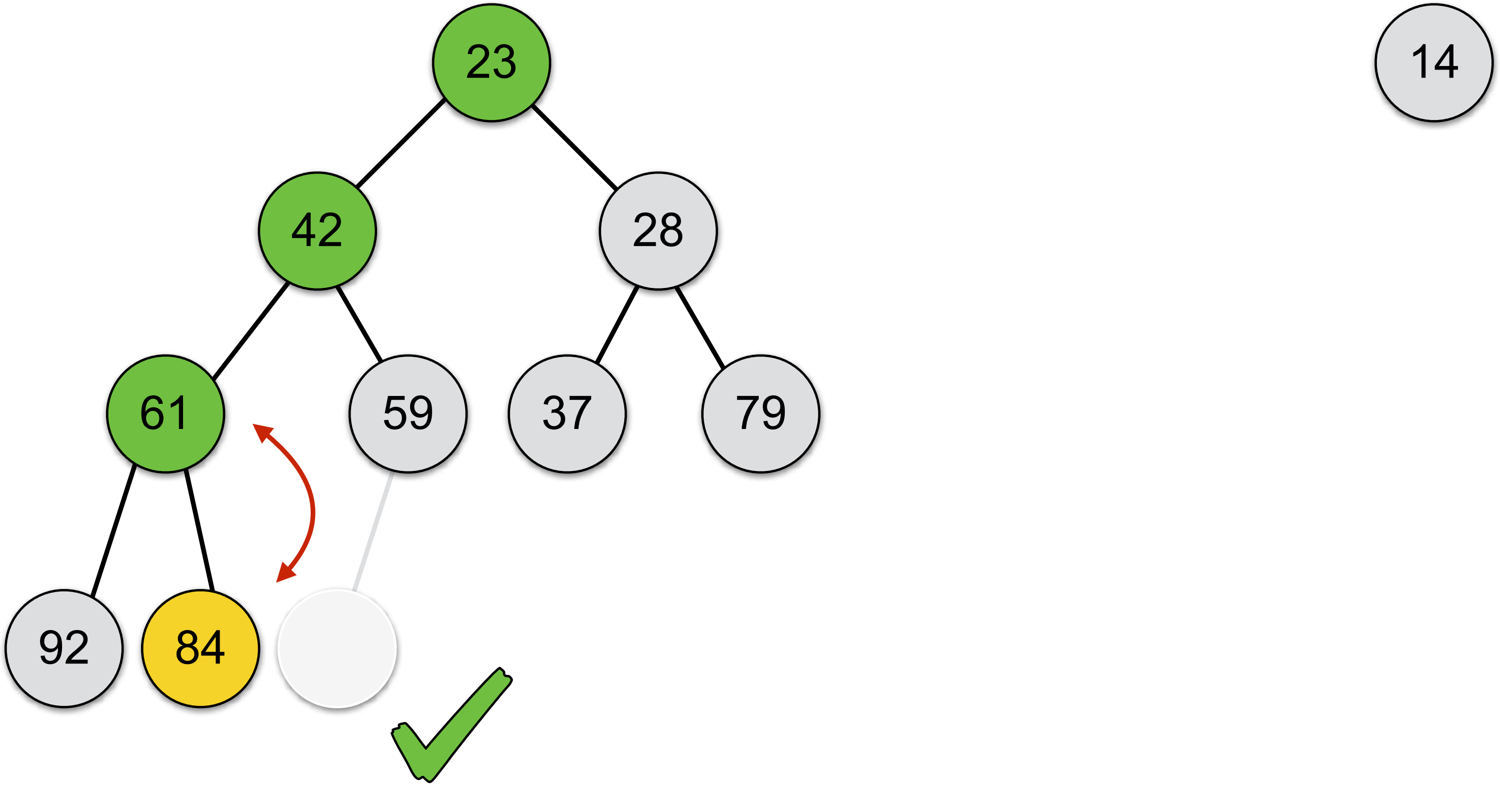
Heapsort



Heapsort

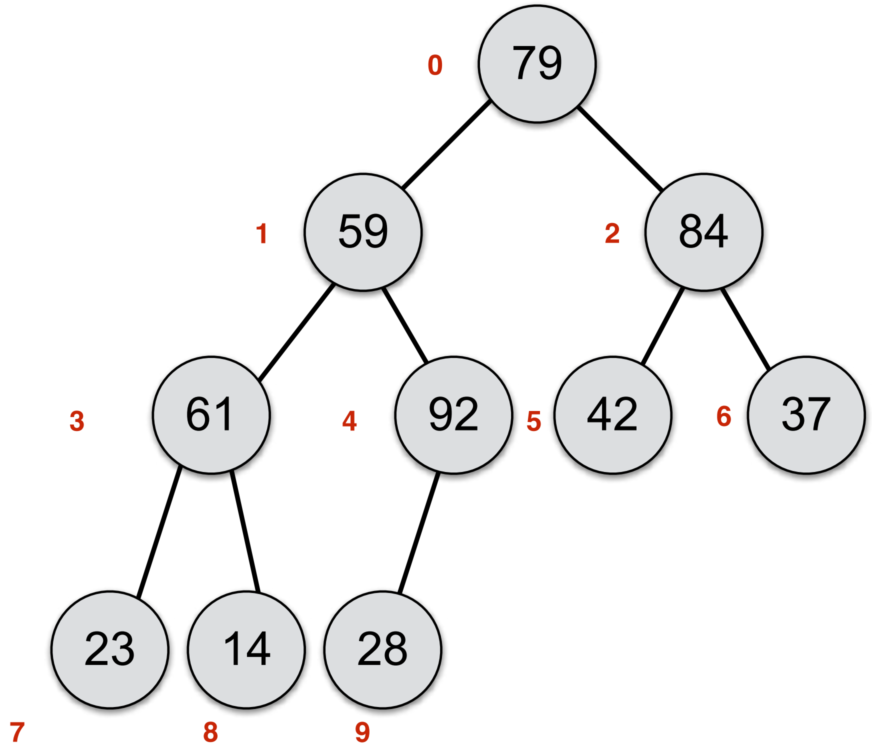


Heapsort



Rest an der Tafel

Heap als Array - Indizierung

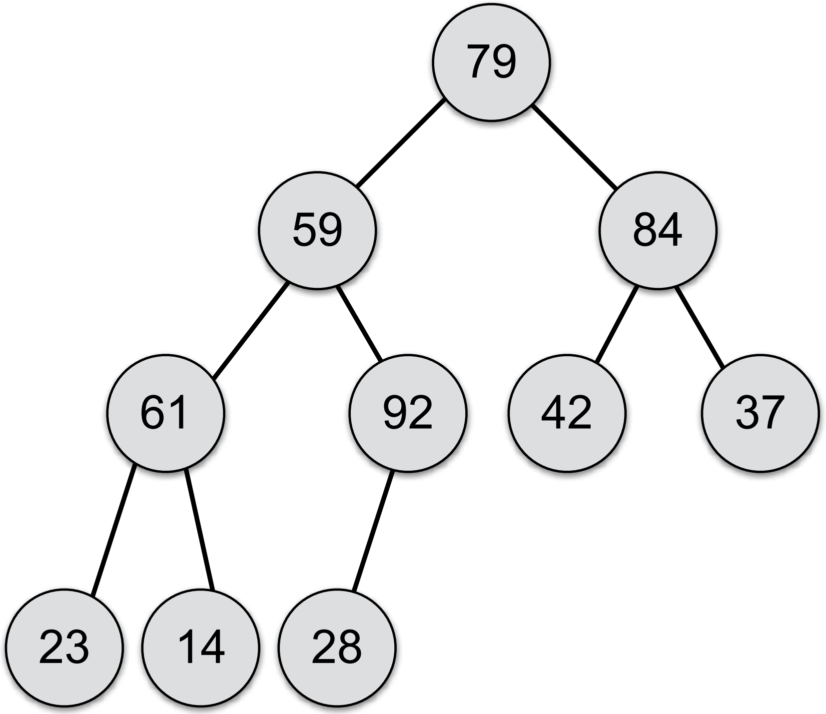


Knoten i hat die Söhne $2 * i + 1$ und $2 * i + 2$

Knoten i hat Vater $(i - 1)/2$

0	1	2	3	4	5	6	7	8	9
79	59	84	61	92	42	37	23	14	28

Reparatur eines Heaps (Heapify)



Knoten i hat die Söhne $2 * i + 1$ und $2 * i + 2$

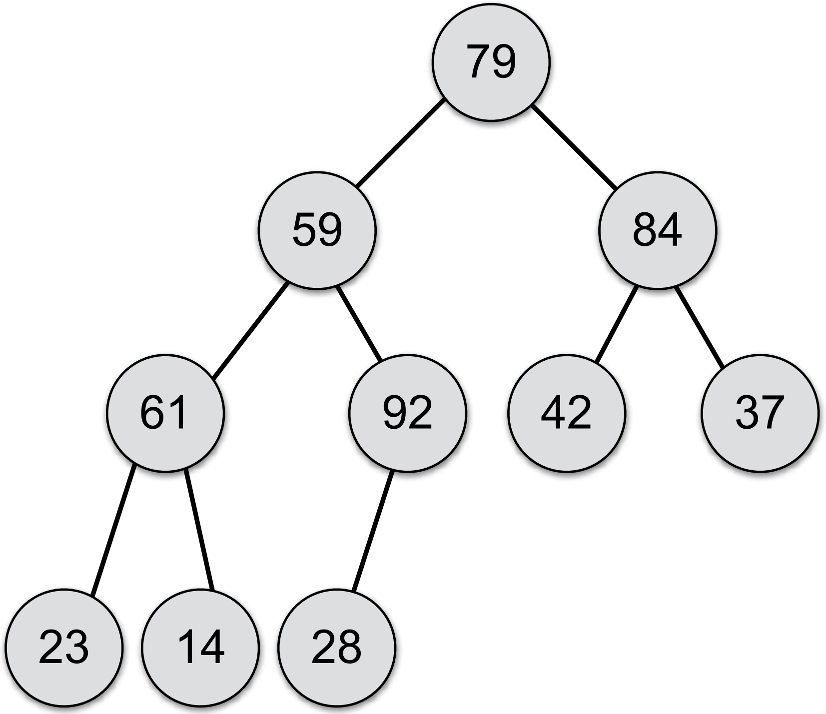
Knoten i hat Vater $(i - 1)/2$

Wie stelle ich die Heap-Eigenschaft wieder her ?

0	1	2	3	4	5	6	7	8	9
79	59	84	61	92	42	37	23	14	28

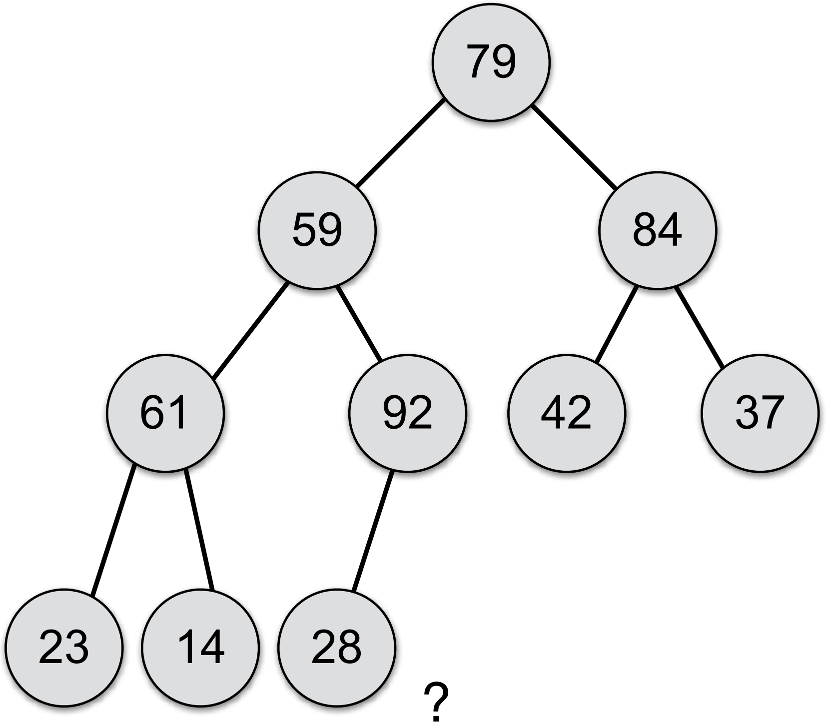
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



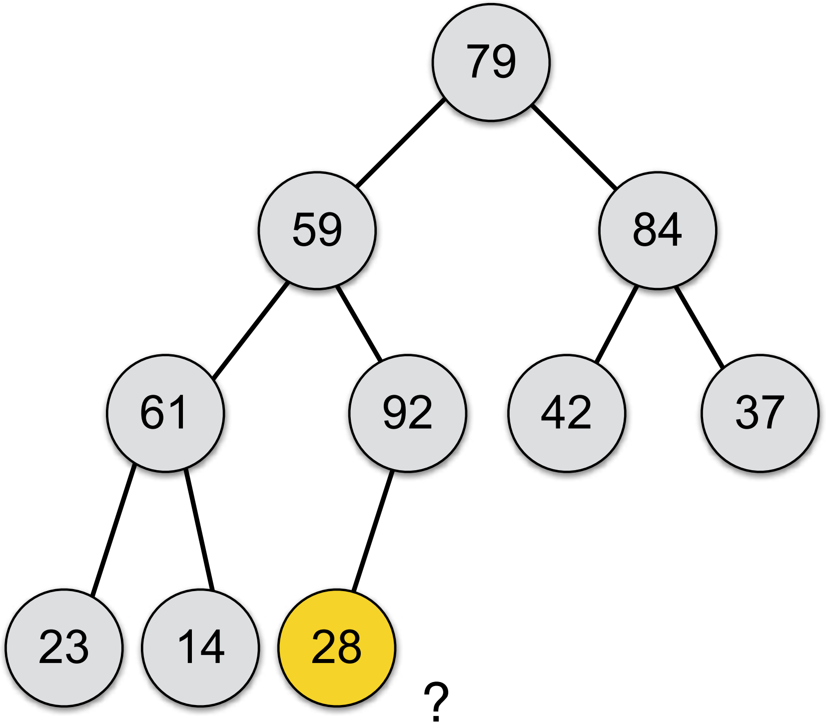
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



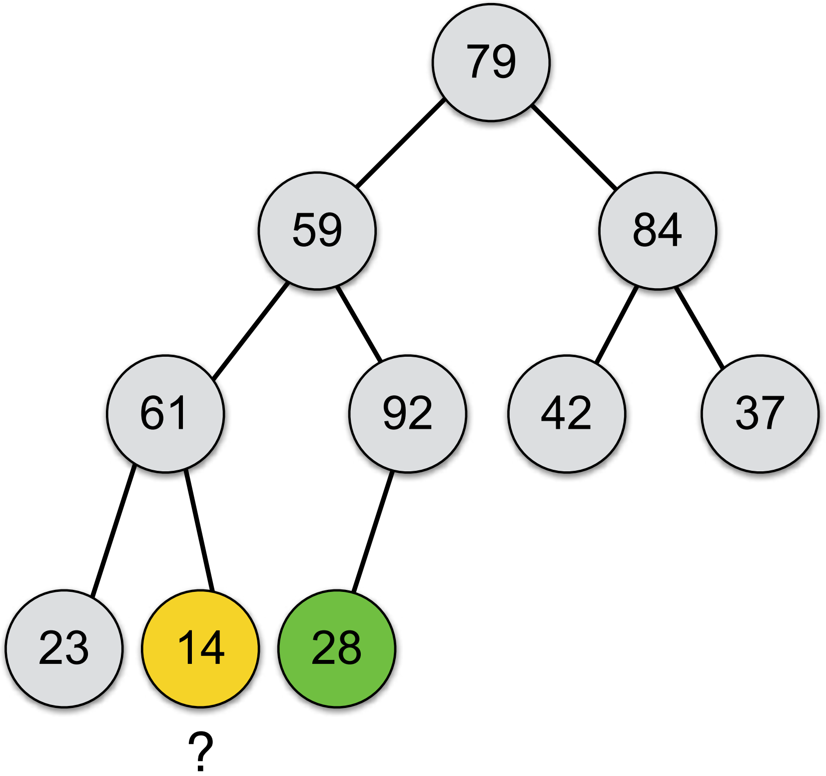
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



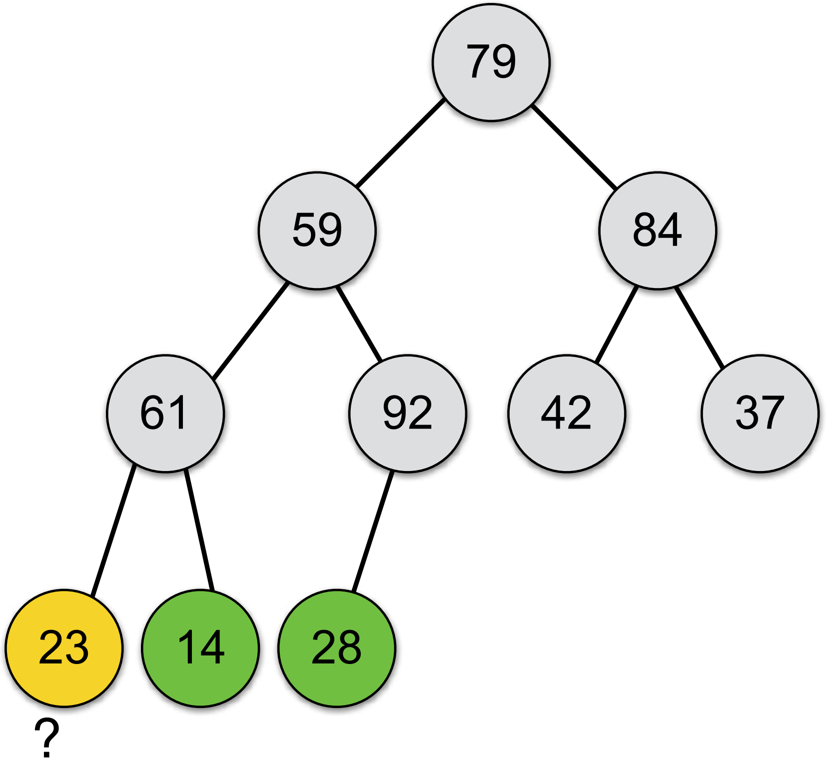
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



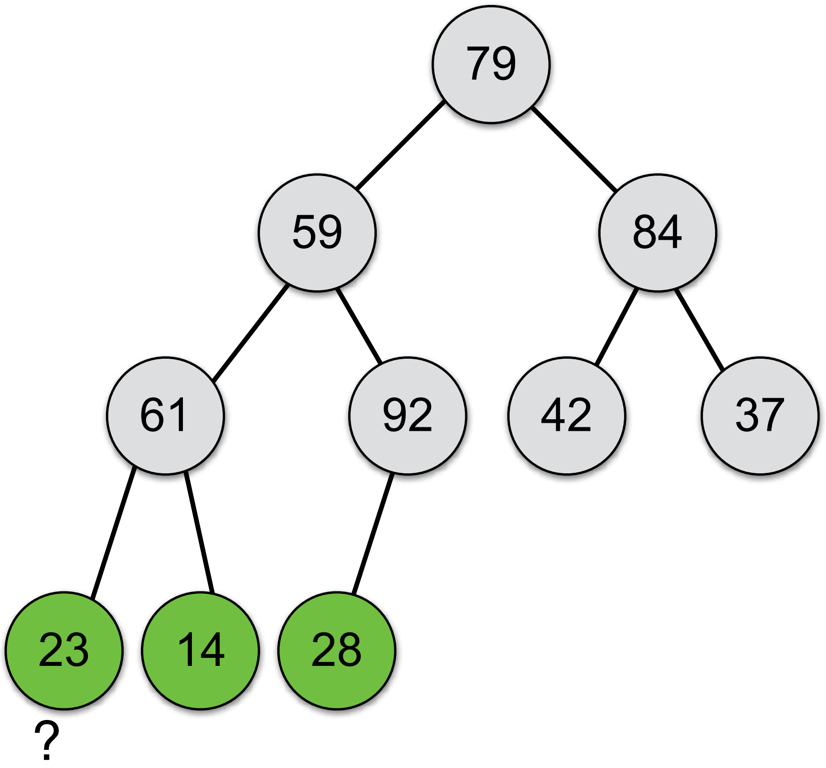
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



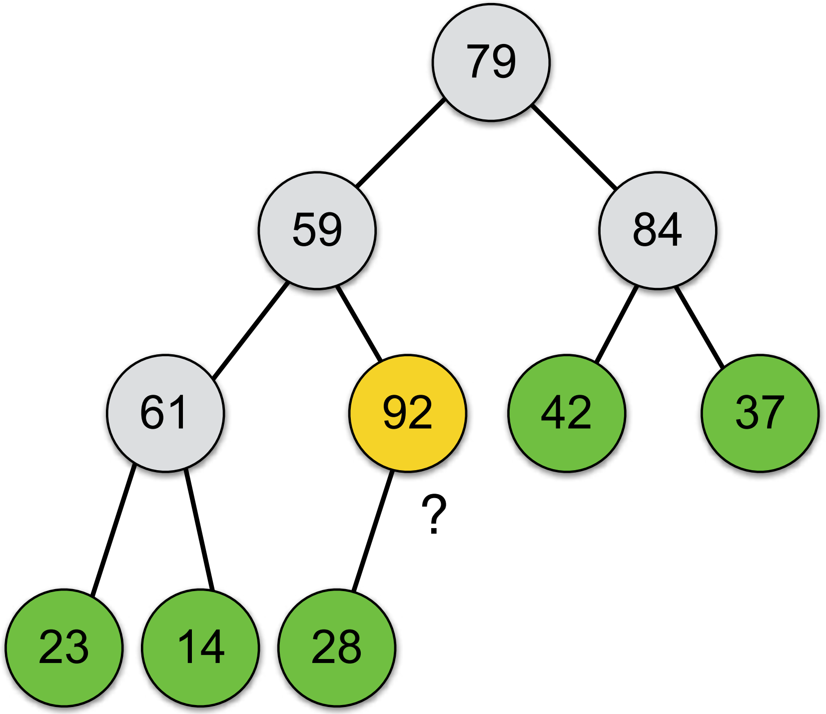
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



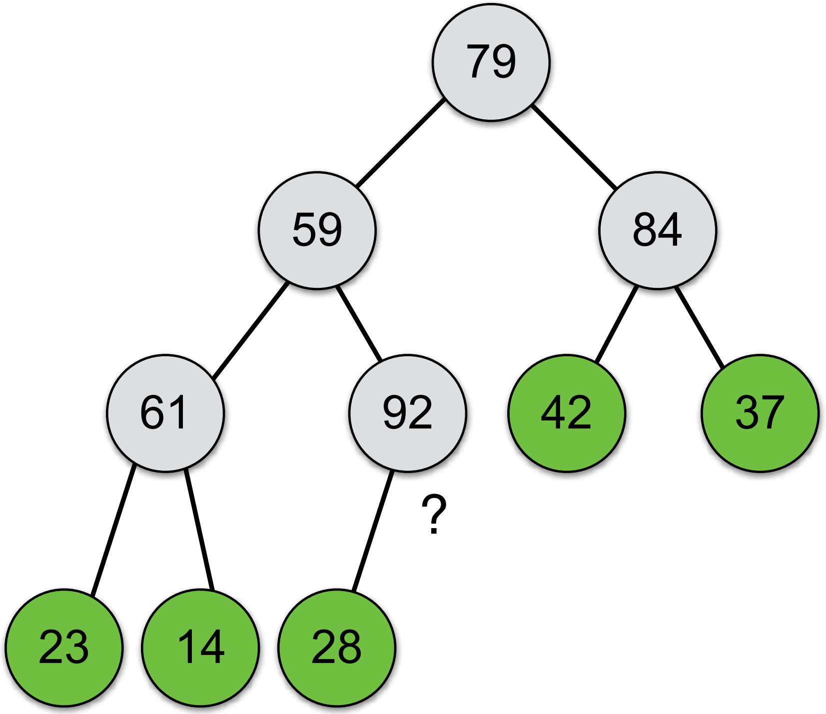
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



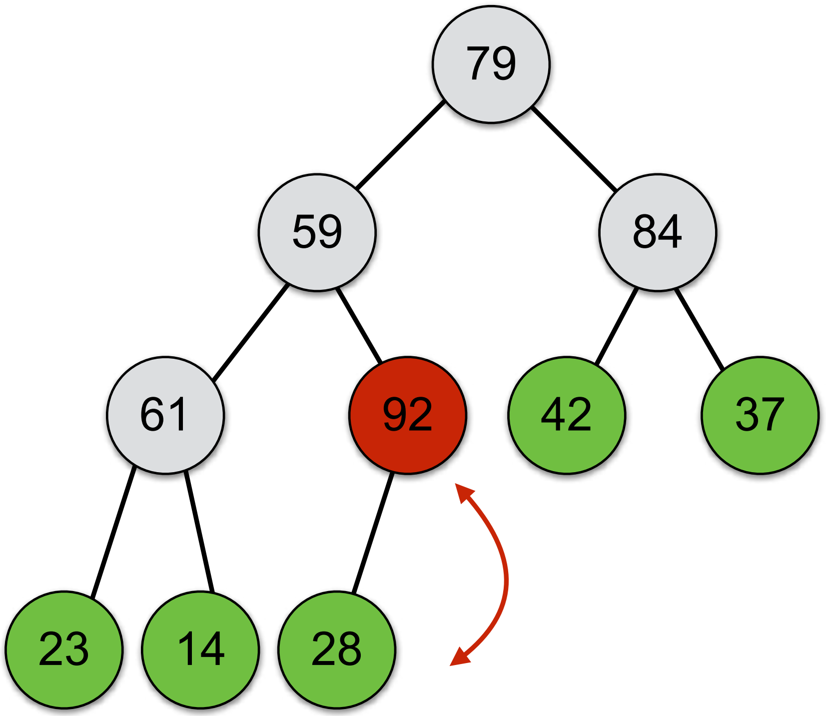
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



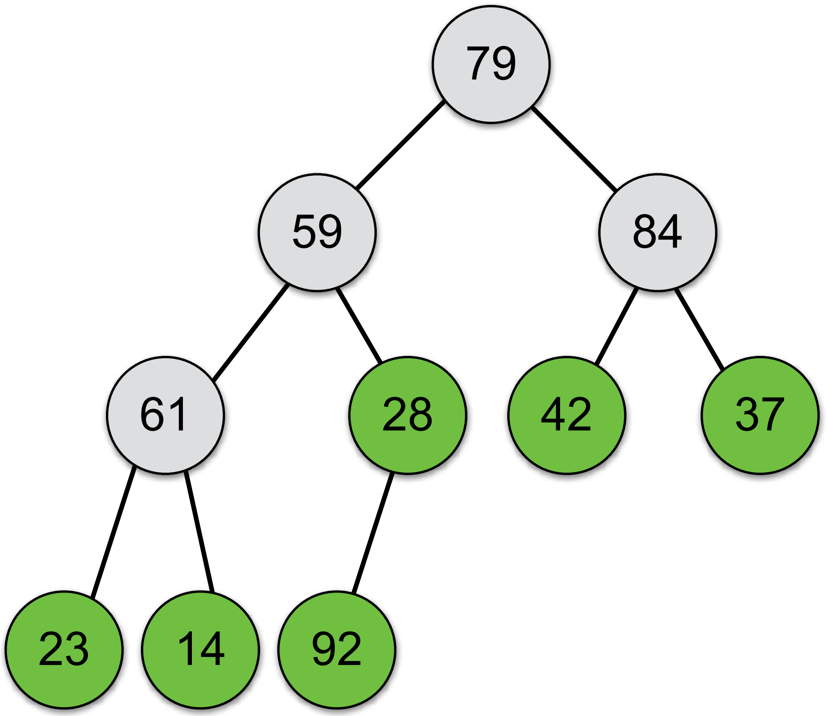
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



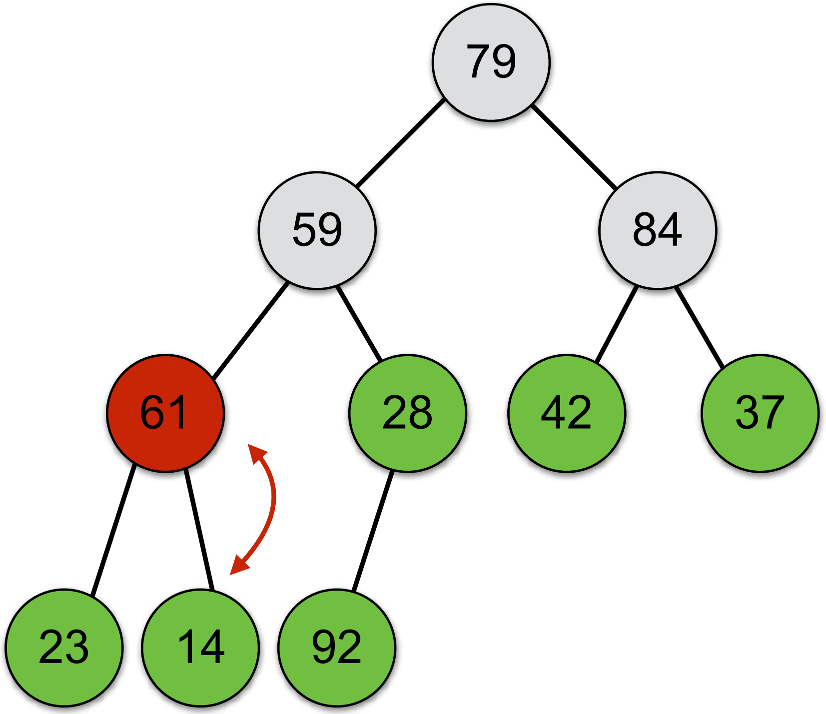
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



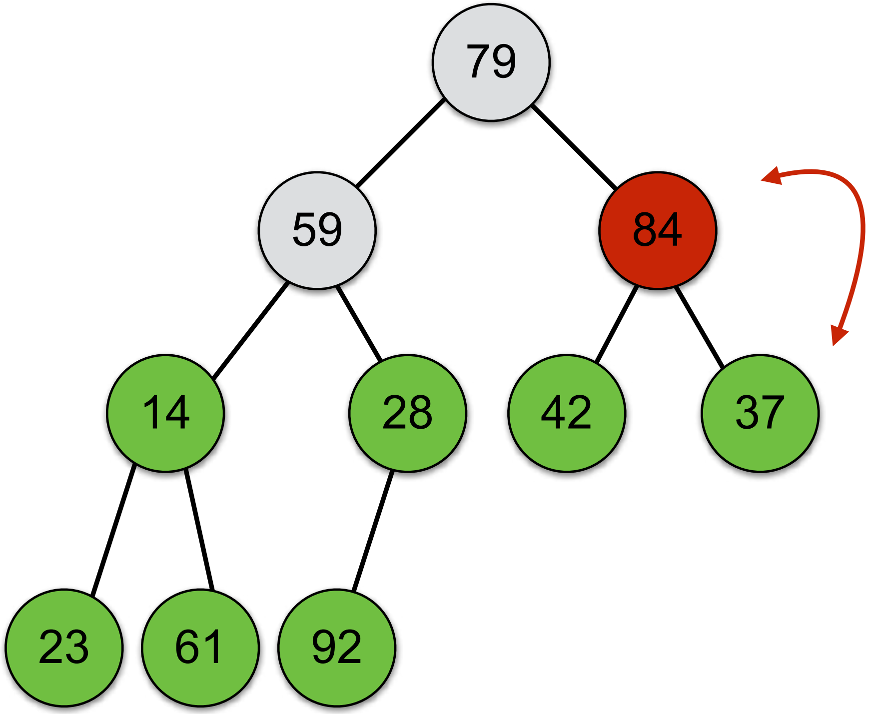
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



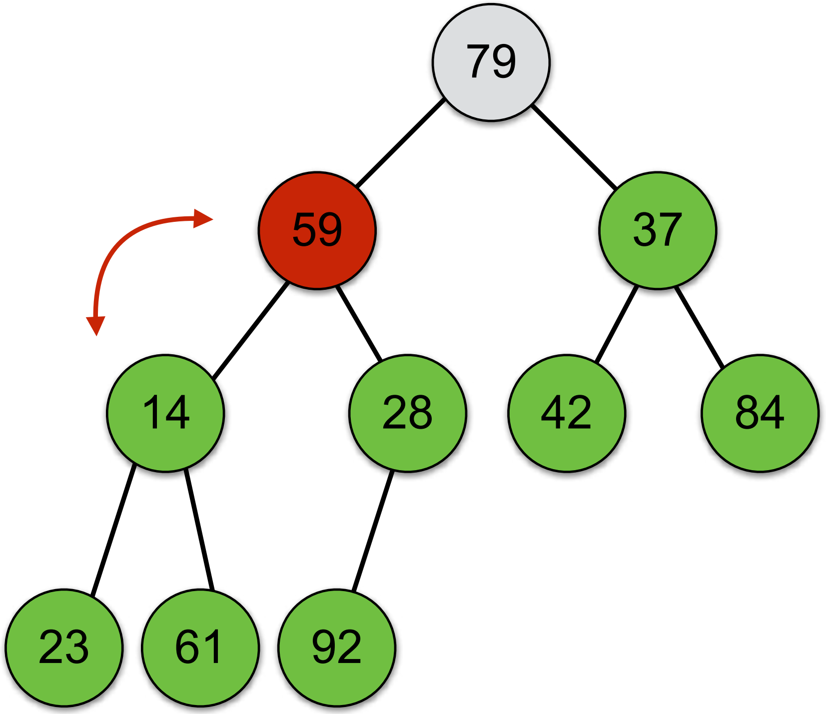
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



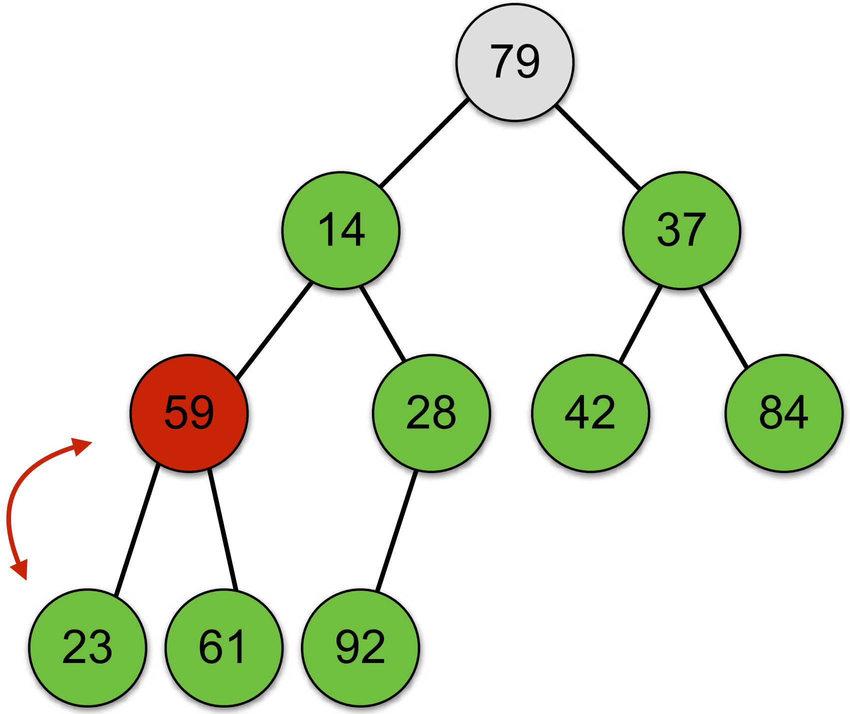
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



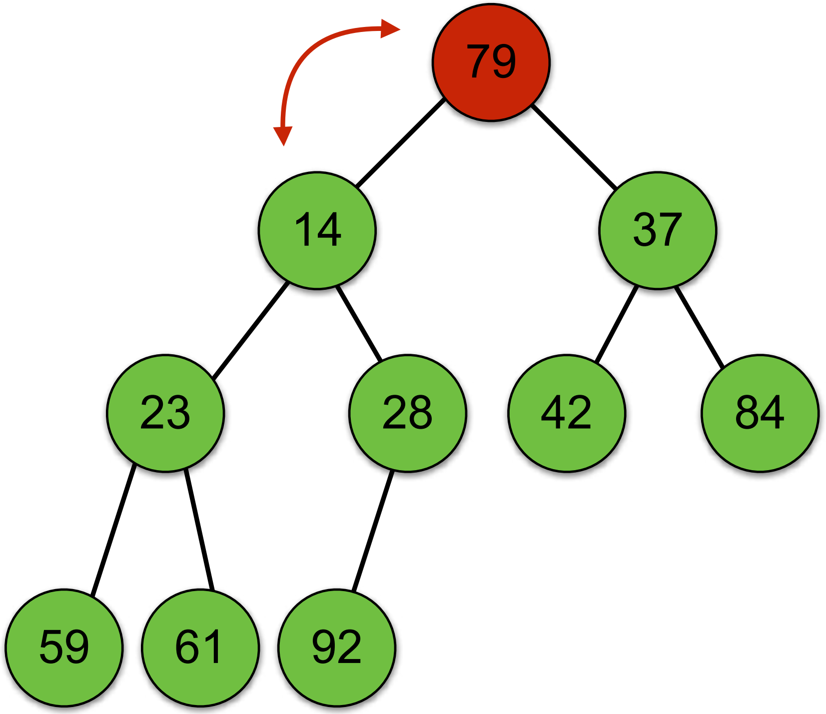
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



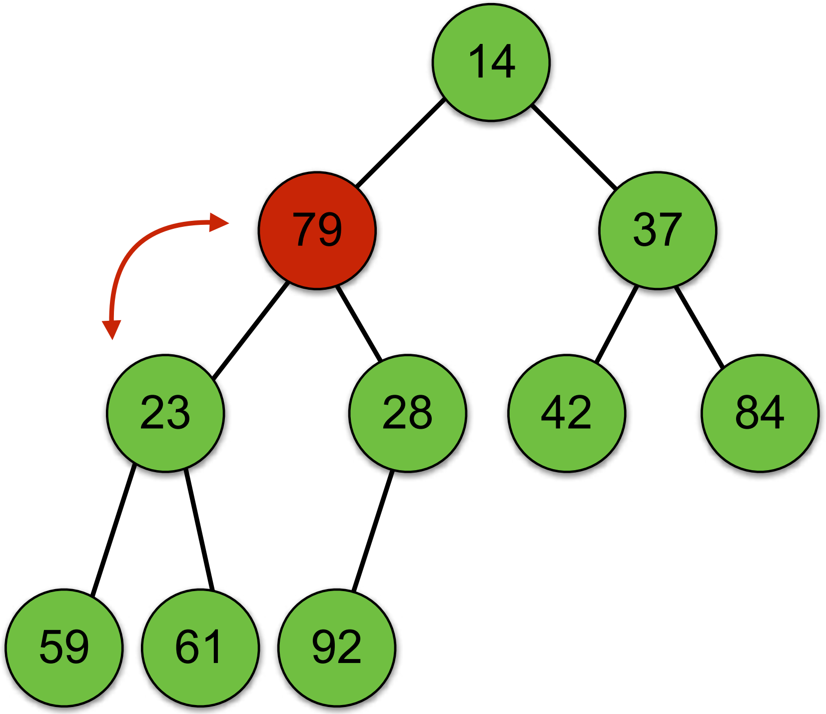
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



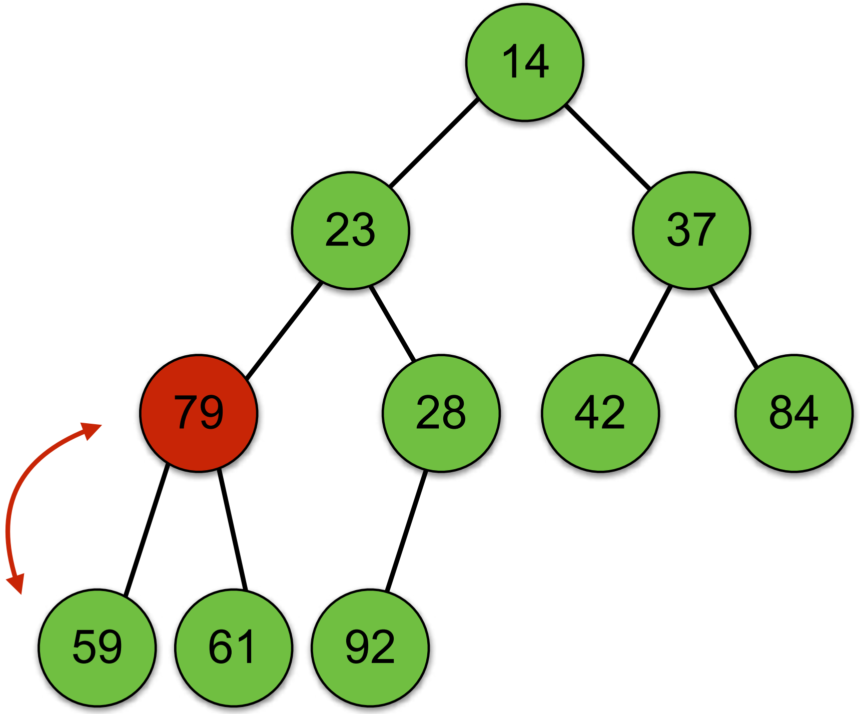
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



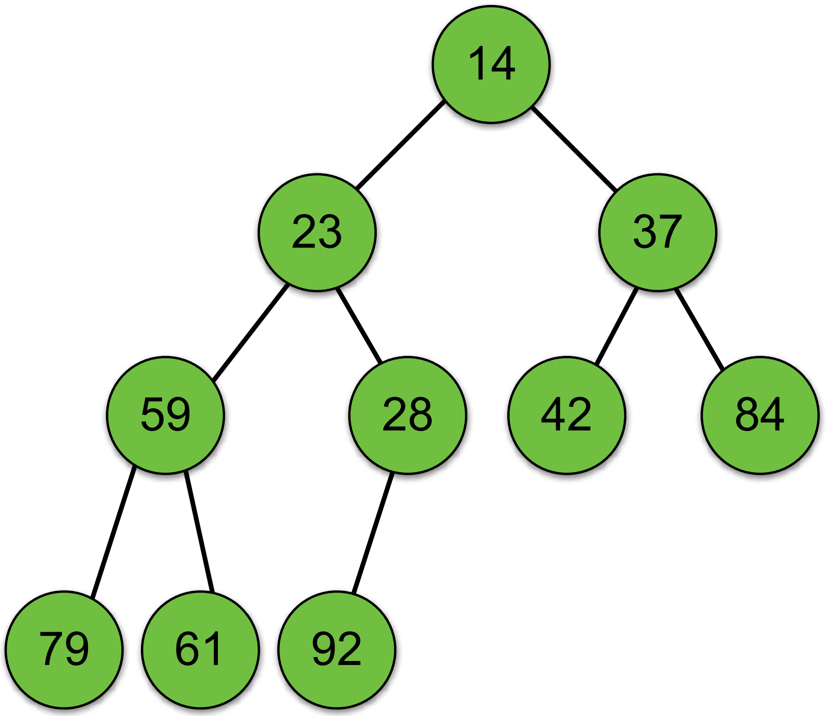
Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



Reparatur eines Heaps (Heapify)

- Annahme: Unter jedem betrachteten Knoten befindet sich bereits ein Heap
- Starte mit letztem Knoten



Hochschule Fulda
University of Applied Sciences

Erinnerung: Priority Queue

```java
public class PriorityQueueExample {
  public static void main(String[] args) {
    Queue<Integer> queue = new PriorityQueue<>();

    // Enqueue random numbers
    for (int i = 0; i < 8; i++) {
      int element = ThreadLocalRandom.current().nextInt(100);
      queue.offer(element);
      System.out.printf("queue.offer(%2d)    -->  queue = %s%n", element, queue);
    }

    // Dequeue all elements
    while (!queue.isEmpty()) {
      Integer element = queue.poll();
      System.out.printf("queue.poll() = %2d  -->  queue = %s%n", element, queue);
    }
  }
}
```

https://www.happycoders.eu/de/algorithmen/priorityqueue-java/

Erinnerung: Priority Queue - Build Heap

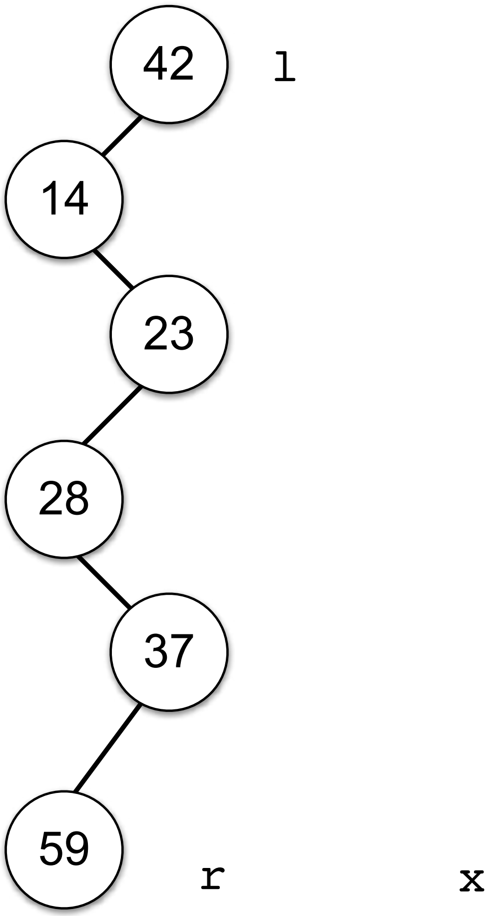
- Kleinstes Element immer vorne!
- Nicht zwangsläufig komplett sortiert
- Heap-Repräsentation des Arrays ➠Siehe Abschnitt 5

```
queue.offer(80)     -->  queue = [80]
queue.offer(14)     -->  queue = [14, 80]
queue.offer(10)     -->  queue = [10, 80, 14]
queue.offer(50)     -->  queue = [10, 50, 14, 80]
queue.offer( 9)     -->  queue = [9, 10, 14, 80, 50]
queue.offer(58)     -->  queue = [9, 10, 14, 80, 50, 58]
queue.offer(41)     -->  queue = [9, 10, 14, 80, 50, 58, 41]
queue.offer( 1)     -->  queue = [1, 9, 14, 10, 50, 58, 41, 80]
queue.poll() =  1   -->  queue = [9, 10, 14, 80, 50, 58, 41]
queue.poll() =  9   -->  queue = [10, 41, 14, 80, 50, 58]
queue.poll() = 10   -->  queue = [14, 41, 58, 80, 50]
queue.poll() = 14   -->  queue = [41, 50, 58, 80]
queue.poll() = 41   -->  queue = [50, 80, 58]
queue.poll() = 50   -->  queue = [58, 80]
queue.poll() = 58   -->  queue = [80]
queue.poll() = 80   -->  queue = []
```

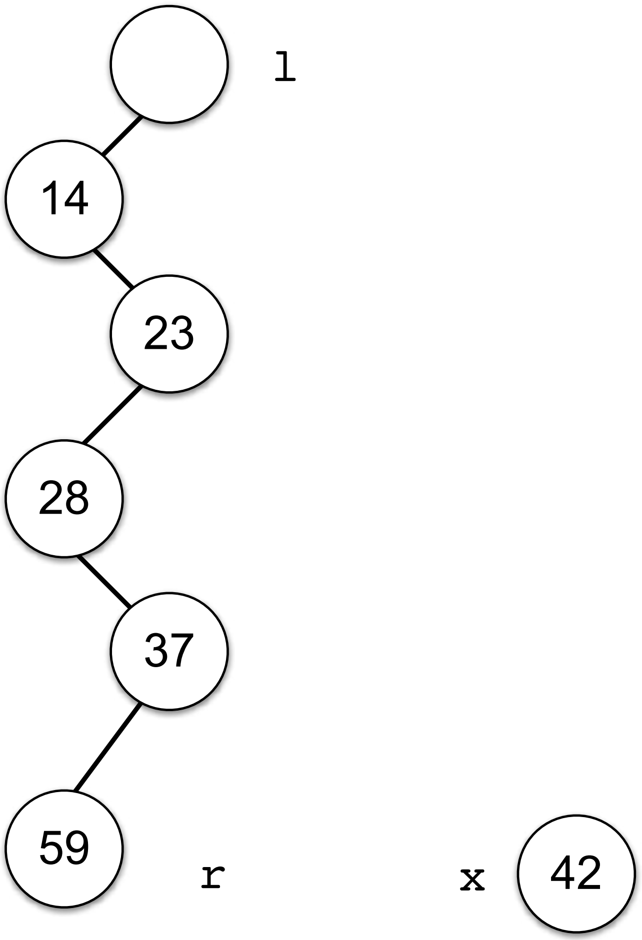
Heapsort

- Algorithmus:
  - Gegeben ein Array von Daten
  - Überführe in einen Heap
  - Entferne das kleinste Element
  - Reorganisiere Heap
  - Solange bis Heap leer

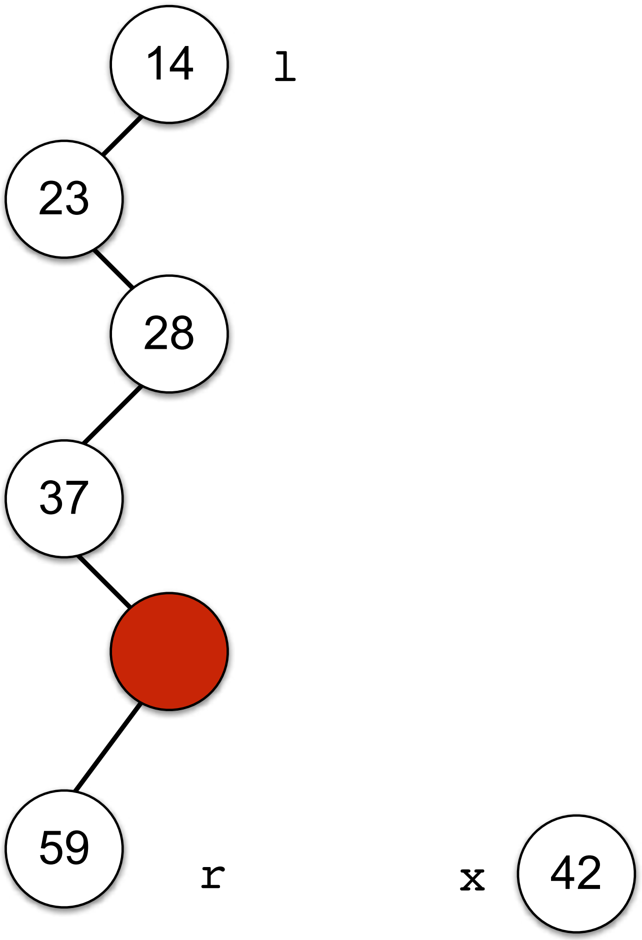
Operation Sift



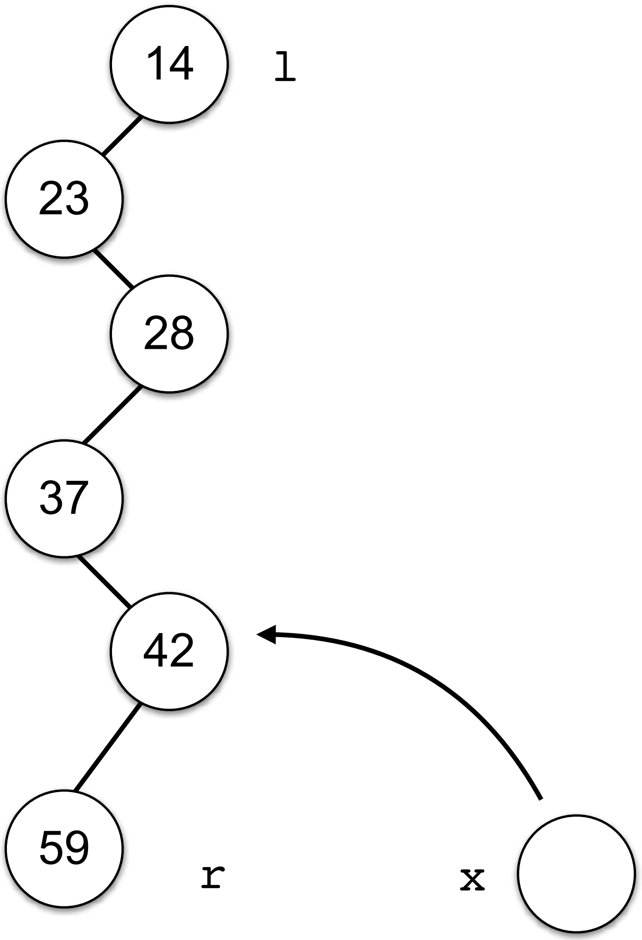
Operation Sift



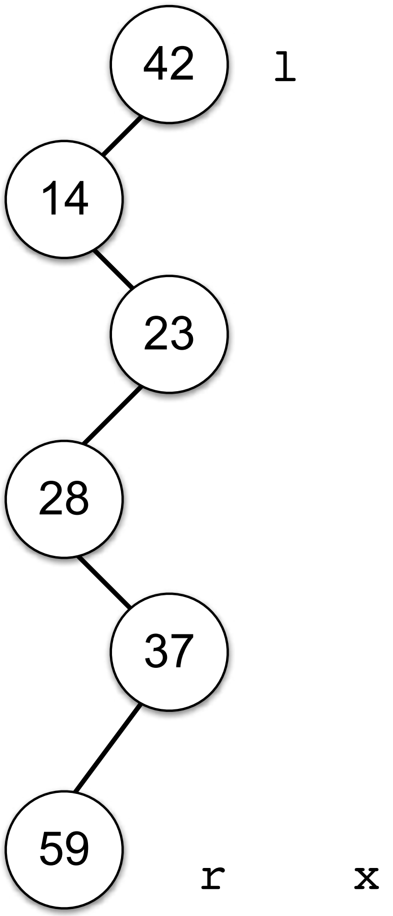
Operation Sift



Operation Sift

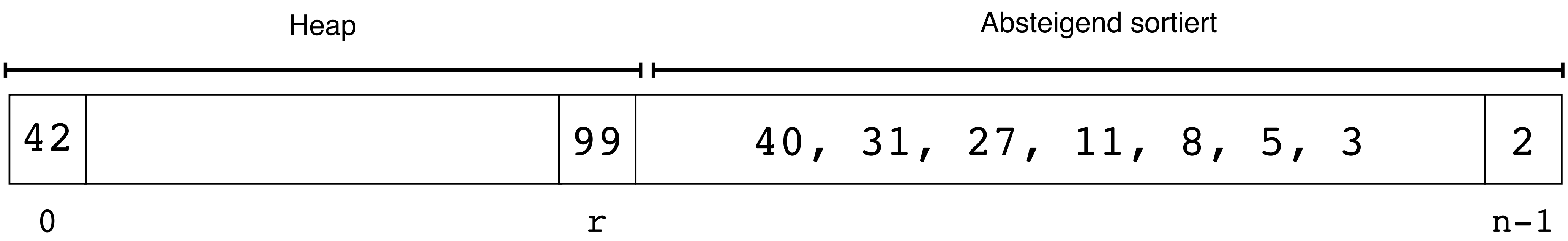


Operation Sift



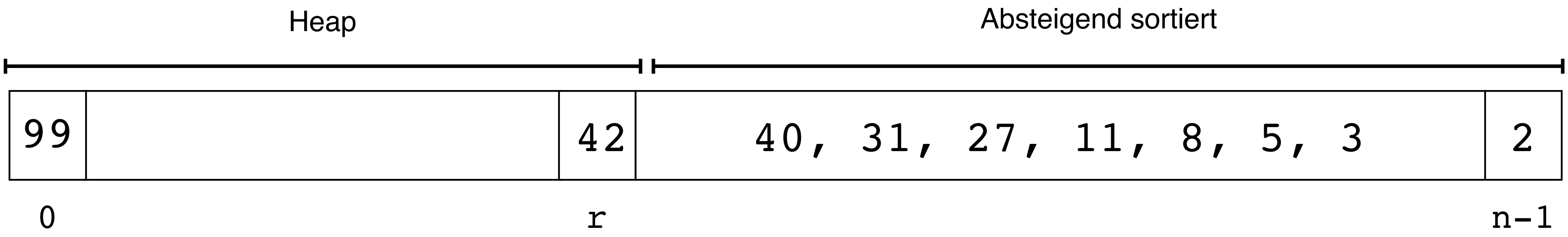

```java
public static void sift(int[] a, int l, int r) {
    int i, j, x; i = l; x = a[l];

    j = 2 * i + 1;
    if(j < r && a[j+1] < a[j]) j++;
    // j ist der kleinere Sohn
    while(j <= r && a[j] < x) {
        a[i] = a[j];
        i = j;
        j = 2 * i + 1;
        if(j < r && a[j+1] < a[j]) j++;
    }
    a[i] = x;
}
```

Heapsort



Heapsort

```
public static void sort(int[] a)
{
    int j, r, n, l, tmp;
    n = a.length;
    for(l = (n - 2) / 2; l >= 0; l--) {
        sift(a, l, n - 1);
    }
    for(r = n-1; r > 0; r--){
        tmp = a[0];
        a[0] = a[r];
        a[r] = tmp;
        sift(a, 0, r-1);
    }
}
```



Hochschule Fulda
University of Applied Sciences

Laufzeiten

- Vermutung: Sift: $\mathcal{O}(\log(n))$
- Heapsort: $\mathcal{O}(n \log(n))$, grob abgeschätzt
- Sift genauer betrachtet:

Ebene	Sickertiefe	Anzahl Knoten
0	h	1
1	h - 1	2
...	...	...
h - 3	3	n / 8
h - 2	2	n / 4
h - 1	1	n / 2

$$\sum_{i=1}^{h} \frac{n}{2^i} i \cdot c = c \cdot n \sum_{i=1}^{h} \frac{i}{2^i} \leq c \cdot n \cdot 2$$

Sift ist in $\mathcal{O}(n)$

Laufzeiten

- Heap:

`get_min` $\mathcal{O}(1)$
`delete_min` $\mathcal{O}(\log n)$
`build_heap` $\mathcal{O}(n)$
`insert` $\mathcal{O}(\log n)$

Frohe Weihnachten!