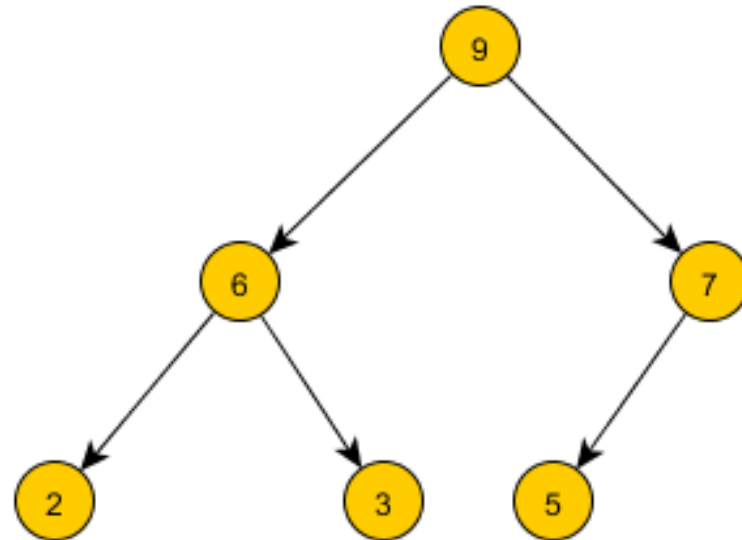
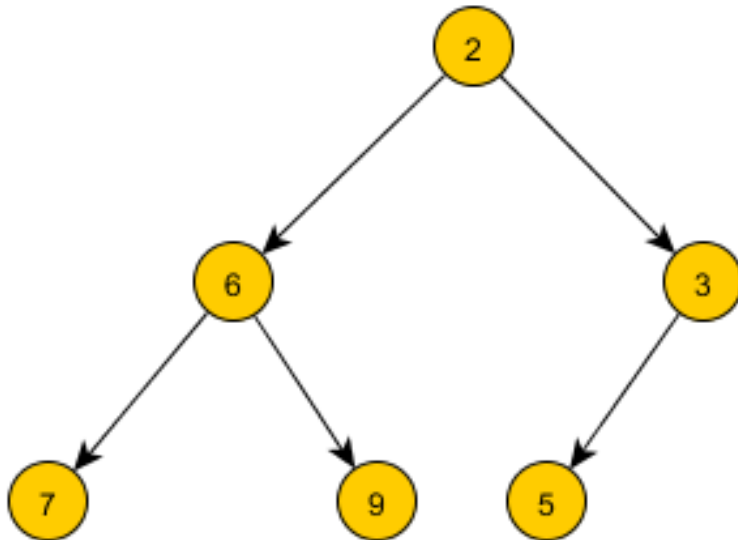


# Priority Queues - HeapSort

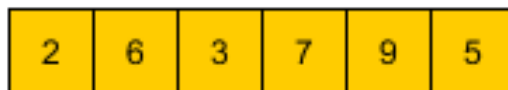
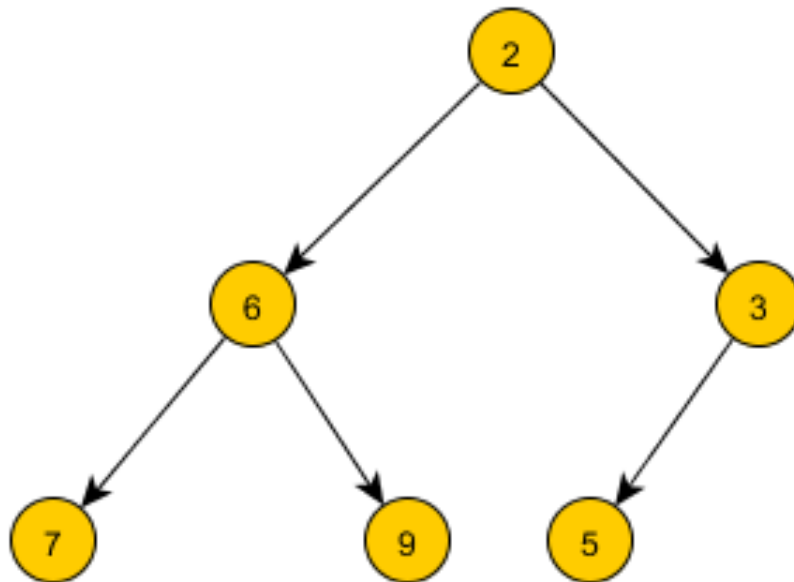
## Algorithmen und Datenstrukturen 2

- Grüne Farbe: Bitte im Script nachtragen

## Repetition Heaps



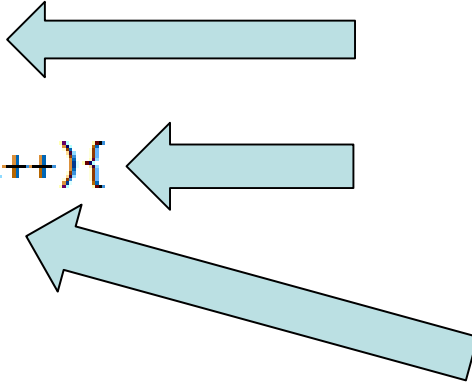
## 1. Ansatz: Heap-Sort basierend auf Min-Heap



```
public int[] sort(int[] values) {  
    int[] sorted = new int[values.length];  
  
    MinPQ minPQ = new MinPQ(values);  
  
    for (int i=0; i < values.length; i++){  
        sorted[i] = minPQ.removeMin();  
    }  
    return sorted;  
}
```

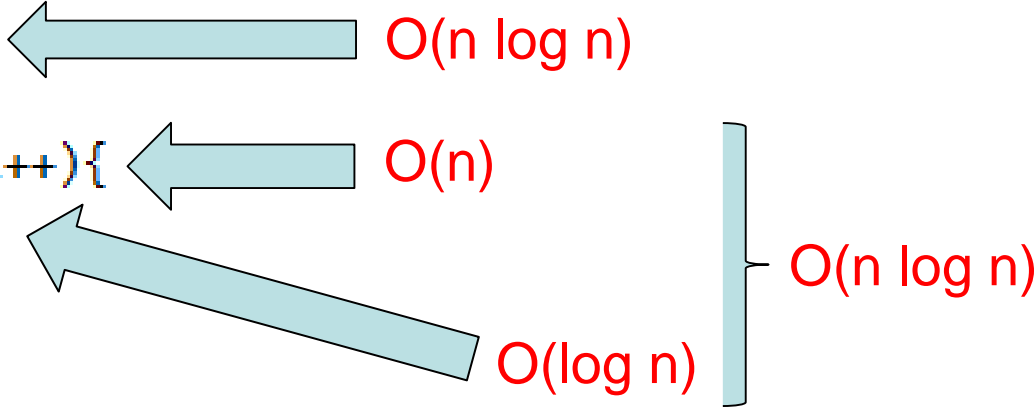
## 1. Ansatz: Komplexitätsanalyse: Heap-Sort basierend auf Min-Heap

```
public int[] sort(int[] values) {  
    int[] sorted = new int[values.length];  
  
    MinPQ minPQ = new MinPQ(values);  
  
    for (int i=0; i < values.length; i++){  
        sorted[i] = minPQ.removeMin();  
    }  
    return sorted;  
}
```



## 1. Ansatz: Komplexitätsanalyse: Heap-Sort basierend auf Min-Heap

```
public int[] sort(int[] values) {  
    int[] sorted = new int[values.length];  
  
    MinPQ minPQ = new MinPQ(values);  
  
    for (int i=0; i < values.length; i++){  
        sorted[i] = minPQ.removeMin();  
    }  
    return sorted;  
}
```



The diagram illustrates the complexity analysis of the provided code. It uses light blue arrows and red text to assign time complexities to different parts of the code:

- A horizontal arrow points from the `MinPQ minPQ = new MinPQ(values);` line to the complexity  $O(n \log n)$ .
- A horizontal arrow points from the `for` loop header `for (int i=0; i < values.length; i++){` to the complexity  $O(n)$ .
- A diagonal arrow points from the `sorted[i] = minPQ.removeMin();` line to the complexity  $O(\log n)$ .
- A vertical bracket on the right side groups the `removeMin()` calls within the loop, with a label  $O(n \log n)$  indicating the total complexity for this part.

**Insgesamt:**  $O(n \log n) + O(n \log n) = 2 (O(n \log n) \in O(n \log n))$  (Worst und Best Case)

## 1. Ansatz: Komplexitätsanalyse: Heap-Sort basierend auf Min-Heap

```
public int[] sort(int[] values) {  
    int[] sorted = new int[values.length];  
  
    MinPQ minPQ = new MinPQ(values);  
  
    for (int i=0; i < values.length; i++){  
        sorted[i] = minPQ.removeMin();  
    }  
    return sorted;  
}
```

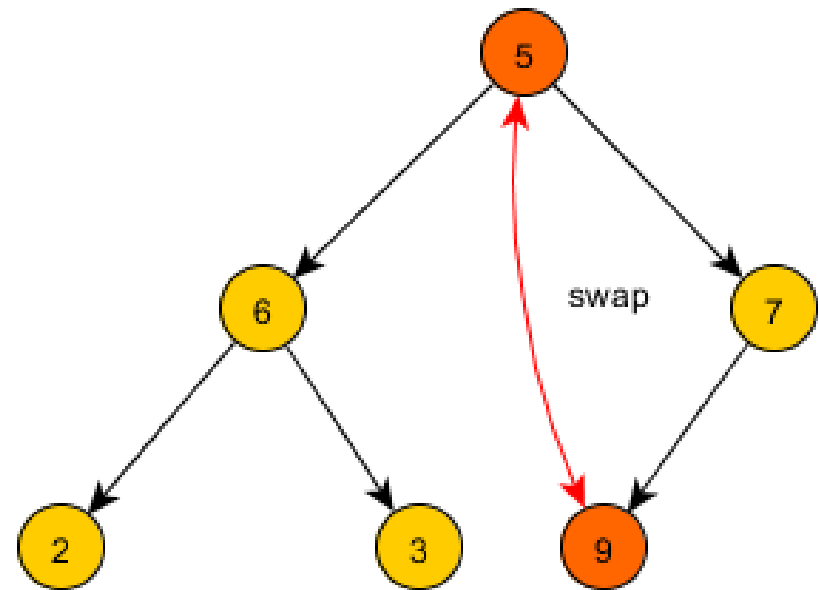
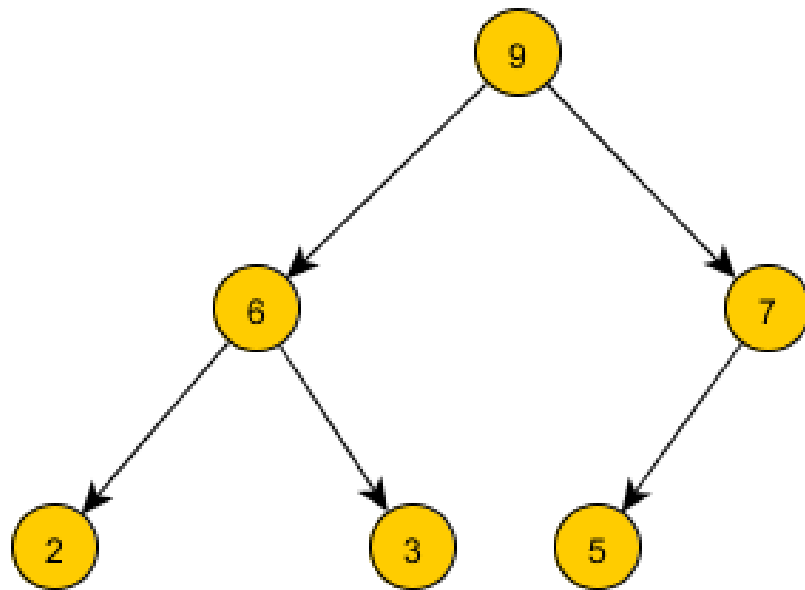
The diagram illustrates the complexity analysis of the provided Java code for Heap Sort. It uses blue arrows to point from complexity labels to specific code segments:

- A horizontal blue arrow points from  $O(n \log n)$  to the line `MinPQ minPQ = new MinPQ(values);`.
- A horizontal blue arrow points from  $O(n)$  to the loop header `for (int i=0; i < values.length; i++){`.
- A diagonal blue arrow points from  $O(\log n)$  to the `removeMin()` call inside the loop.
- A vertical blue bracket on the right side of the loop body groups the `sorted[i] = minPQ.removeMin();` line and the closing brace `}`, with a label  $O(n \log n)$  next to it.

**Insgesamt:**  $O(n \log n) + O(n \log n) = 2 (O(n \log n)) \in O(n \log n)$  (Worst und Best Case)

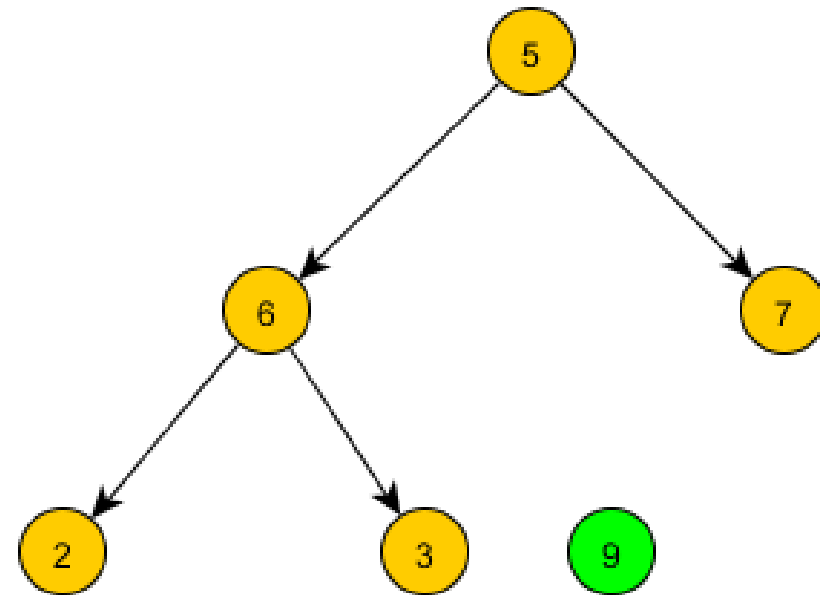
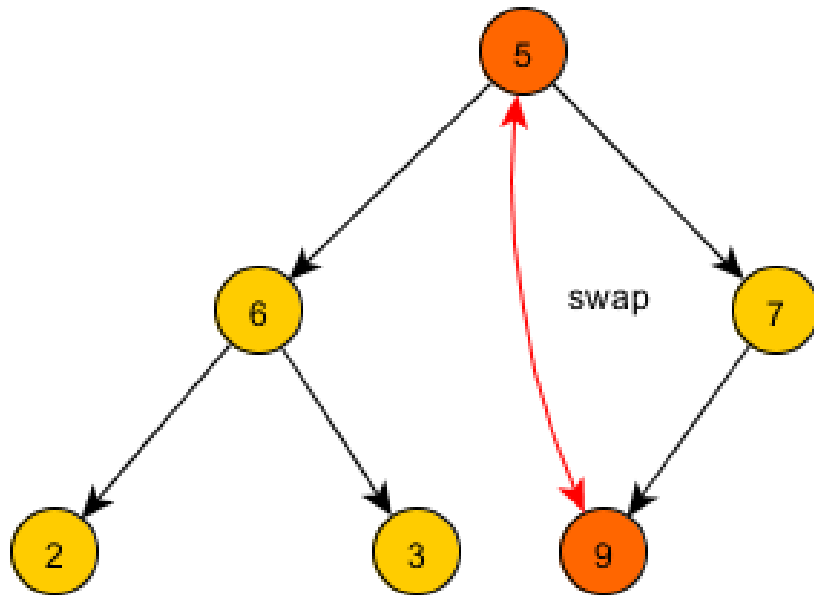
## Heap-Sort basierend auf Max-Heap

1. Schritt: Anstatt removeMax() wird swap() ausgeführt:



## Heap-Sort basierend auf Max-Heap

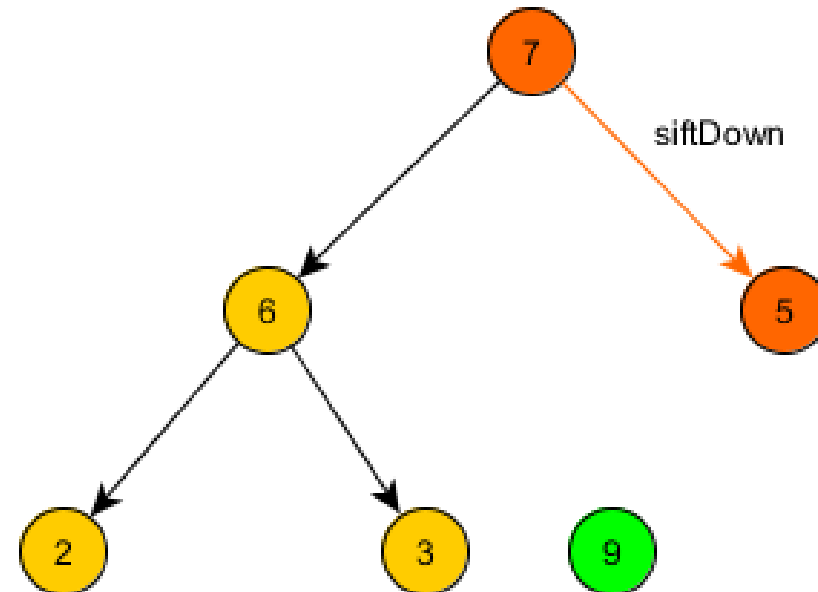
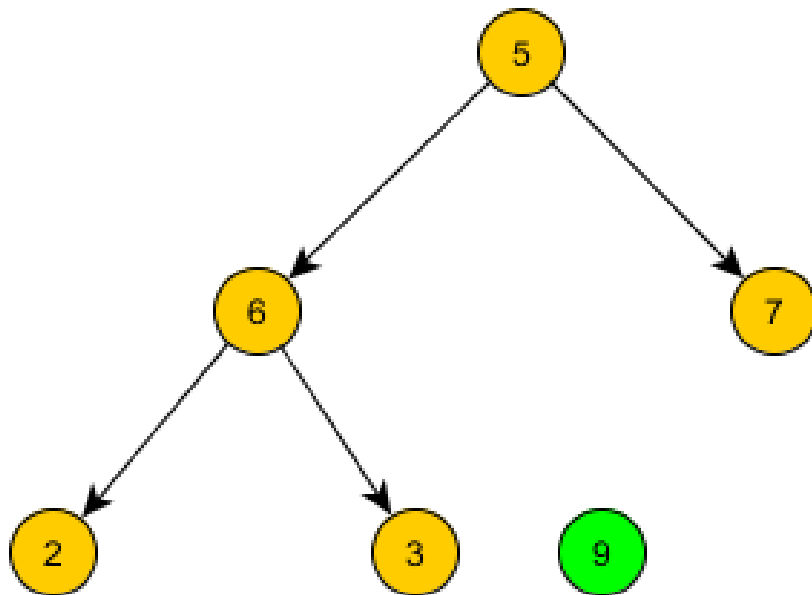
### 2. Schritt: Heap verkleinern





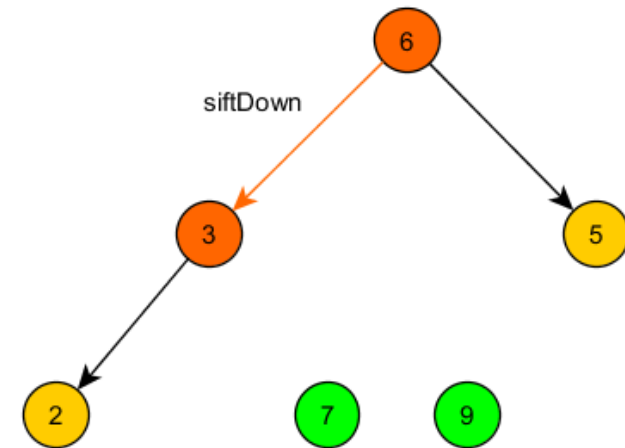
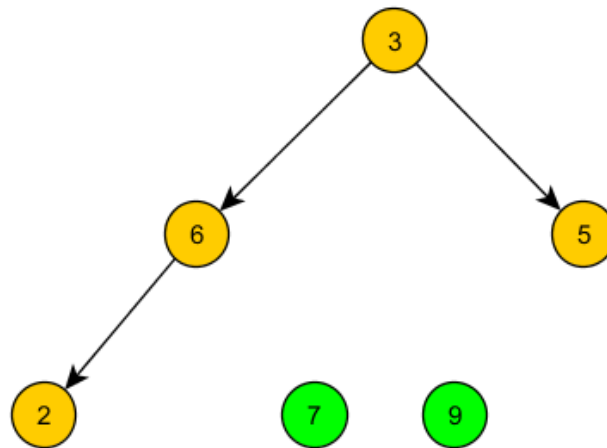
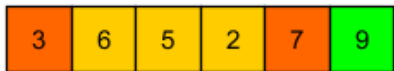
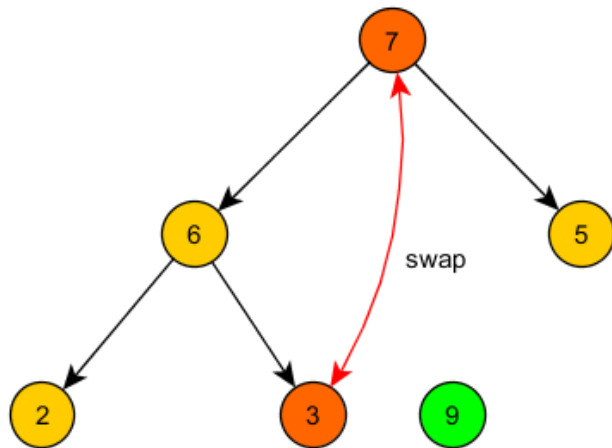
## Heap-Sort basierend auf Max-Heap

3. Schritt: Ordnungs-Relation wiederherstellen (siftDown)

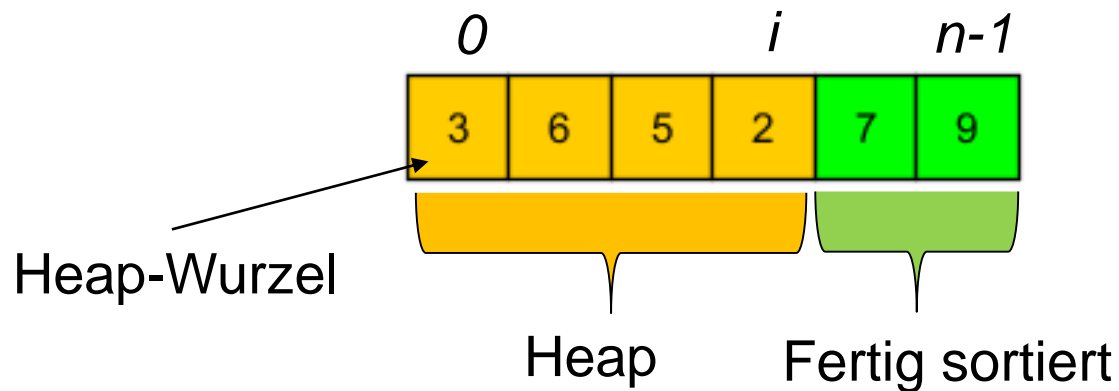


## Heap-Sort basierend auf Max-Heap

Und diese 3 Schritte für alle Elemente...



## Heap-Sort: Phase 1: Ergänzungen im Script (Kap. 5.8, Seite 6)

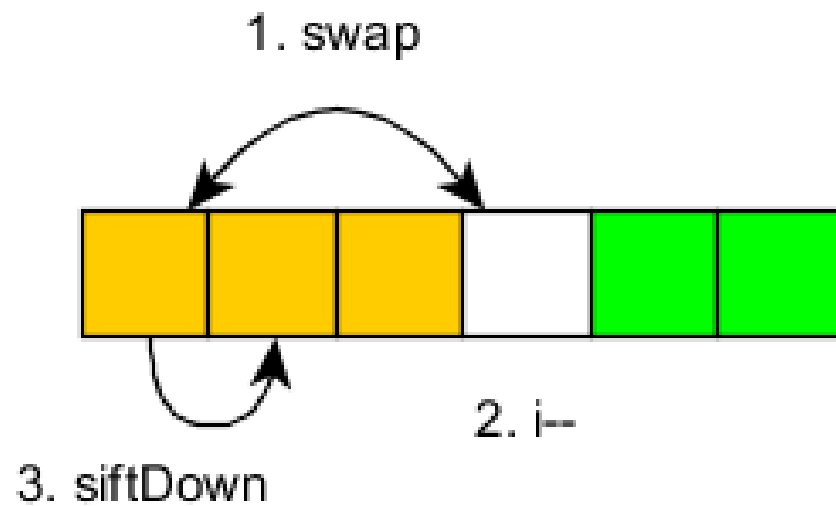


Heap-Wurzel liegt bei Index: 0

Heap-Bereich:  $[0 \dots i]$

An der Wurzel sollte das **grösste** Element zu finden sein, d.h es wird ein **Max-Heap** benötigt.

## Heap-Sort: Phase 2: Ergänzungen im Script (Kap. 5.8, Seite 7)



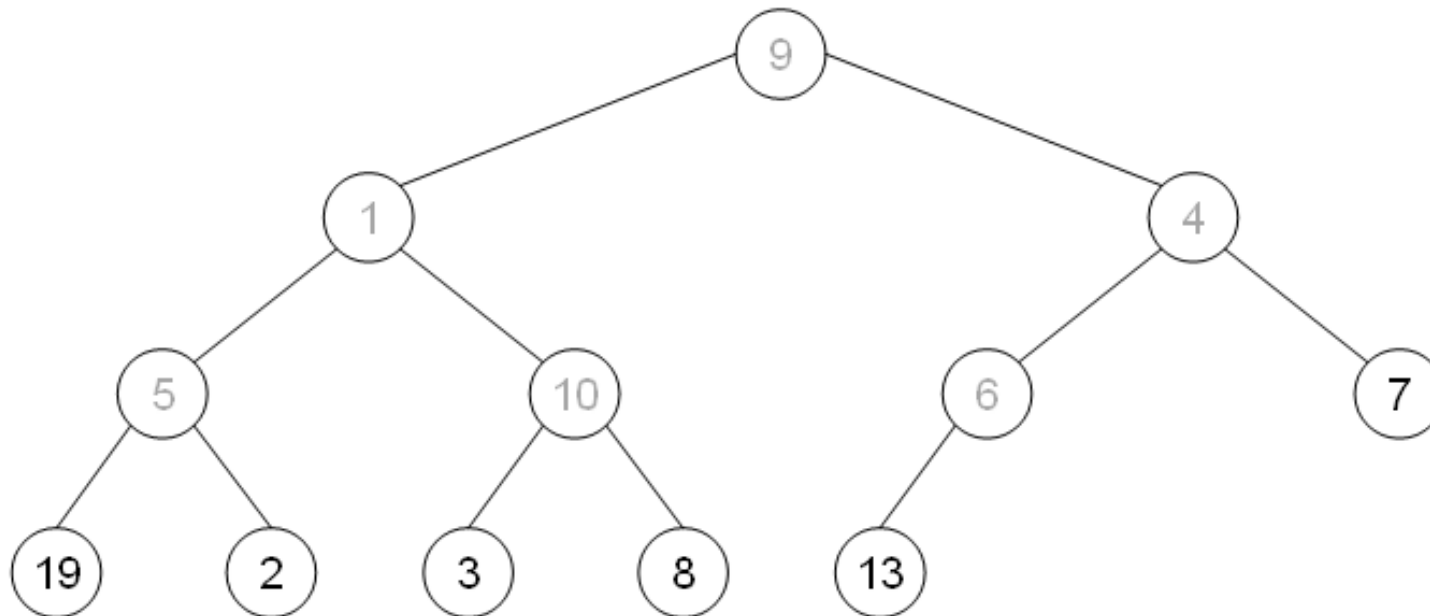
## Heap-Sort Zusammenfassung

- $O(n \log n)$  (optimal)
- In-Place
- Nicht stabil!
- Obwohl  $O(n \log n)$  in Praxis langsamer als Quicksort
  - Quicksort hat bessere Avg-Performance
  - HeapSort nutzt Caches schlecht
  - Quicksort Worst Case Wahrscheinlichkeit kann auf Minimum reduziert werden (median-of-3)
- Optional: Introsort (Hybrid-Ansatz Quicksort – HeapSort)

## Effizienter Heap-Aufbau (Phase 1) nach Floyd

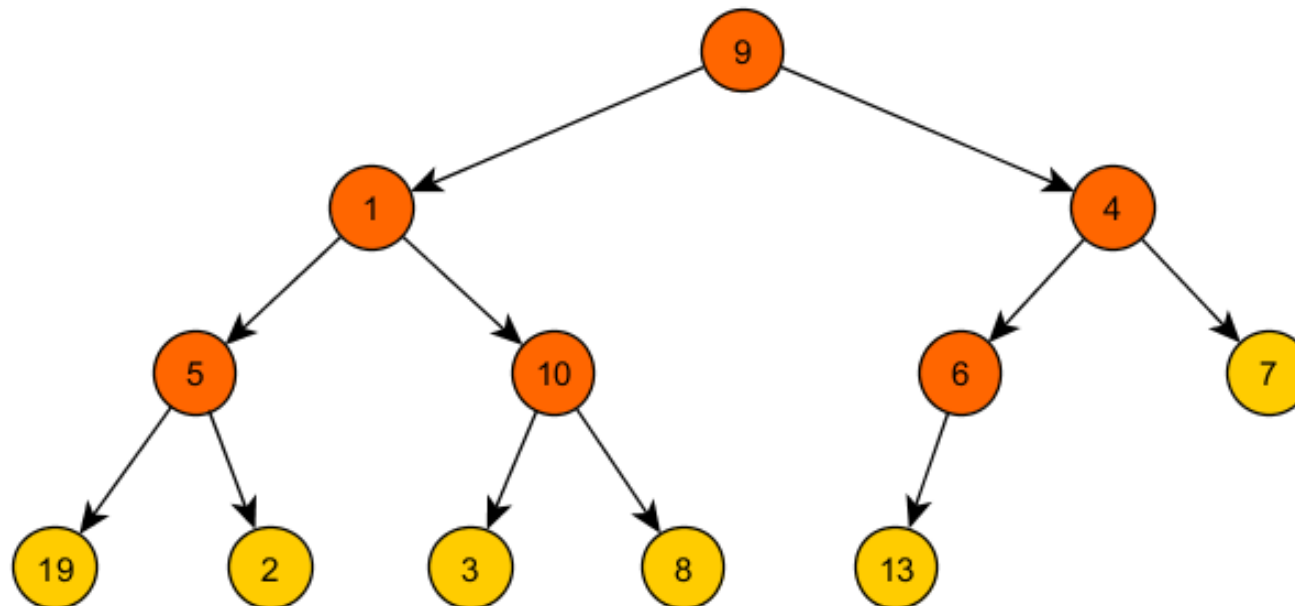
- Input-Array nehmen und direkt als **(Max-)Heap** abbilden
  - 2 Invarianten bereits erfüllt (Binärer Suchbaum, Struktur-Relation)
  - Verletzt: Ordnungsrelation

9	1	4	5	10	6	7	19	2	3	8	13
---	---	---	---	----	---	---	----	---	---	---	----

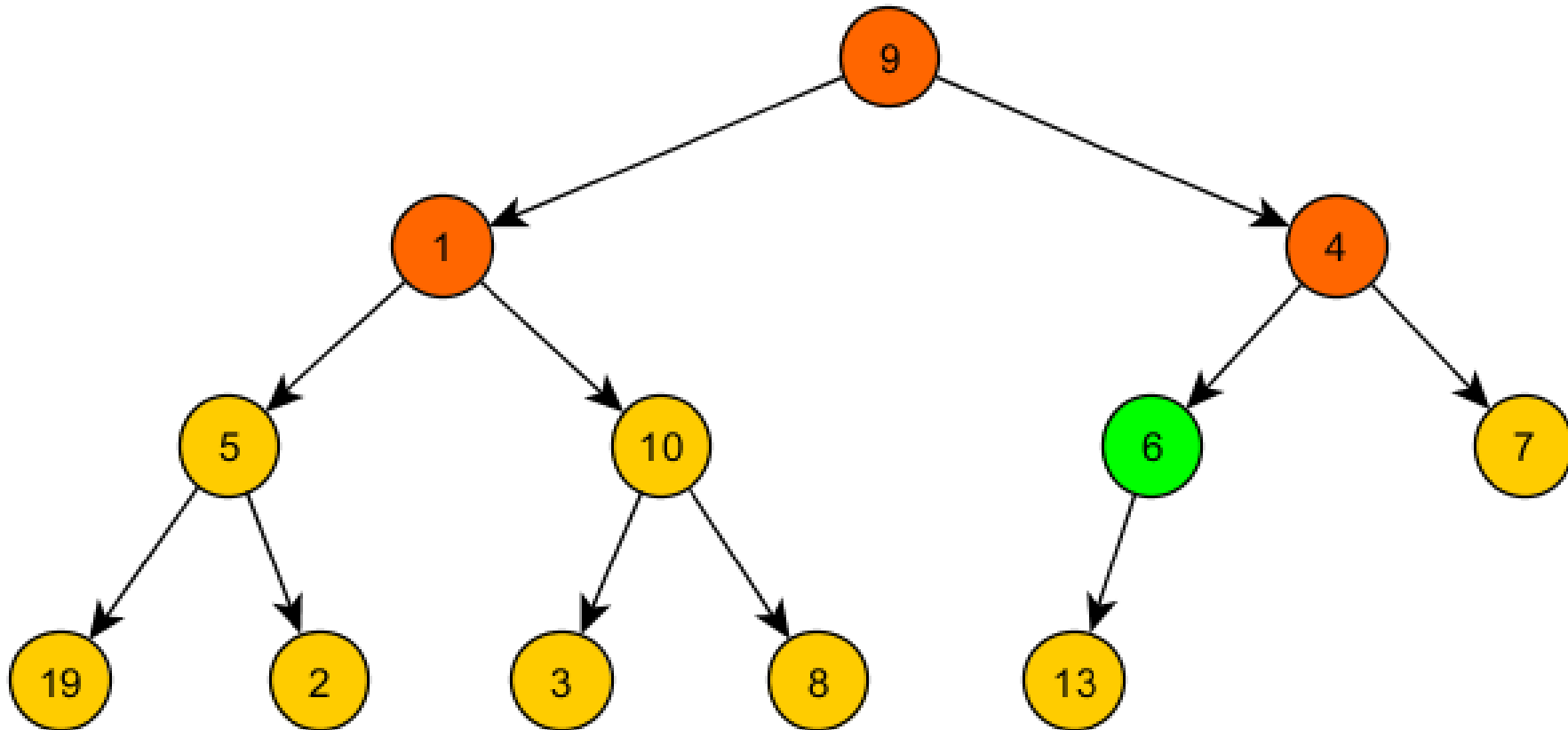


## Effizienter Heap-Aufbau (Phase 1) nach Floyd

- Ordnungsrelation korrigieren (**Max-Heap**)
- Blattknoten erfüllen bereits automatisch Bedingung
- Für alle inneren Knoten:
  - Rückwärts die inneren Knoten durchlaufen und siftDown ausführen

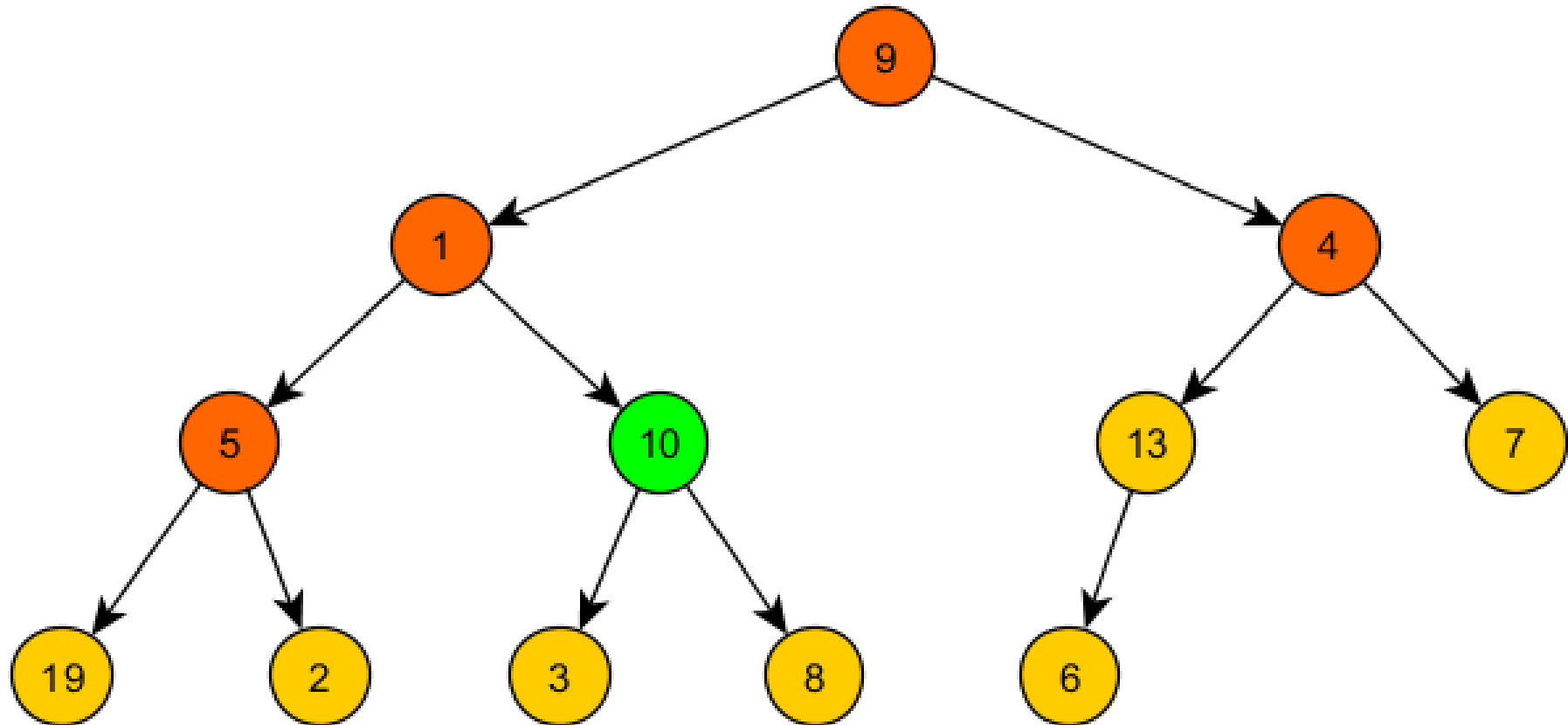


## Floyd Heap-Aufbau Beispiel (1)

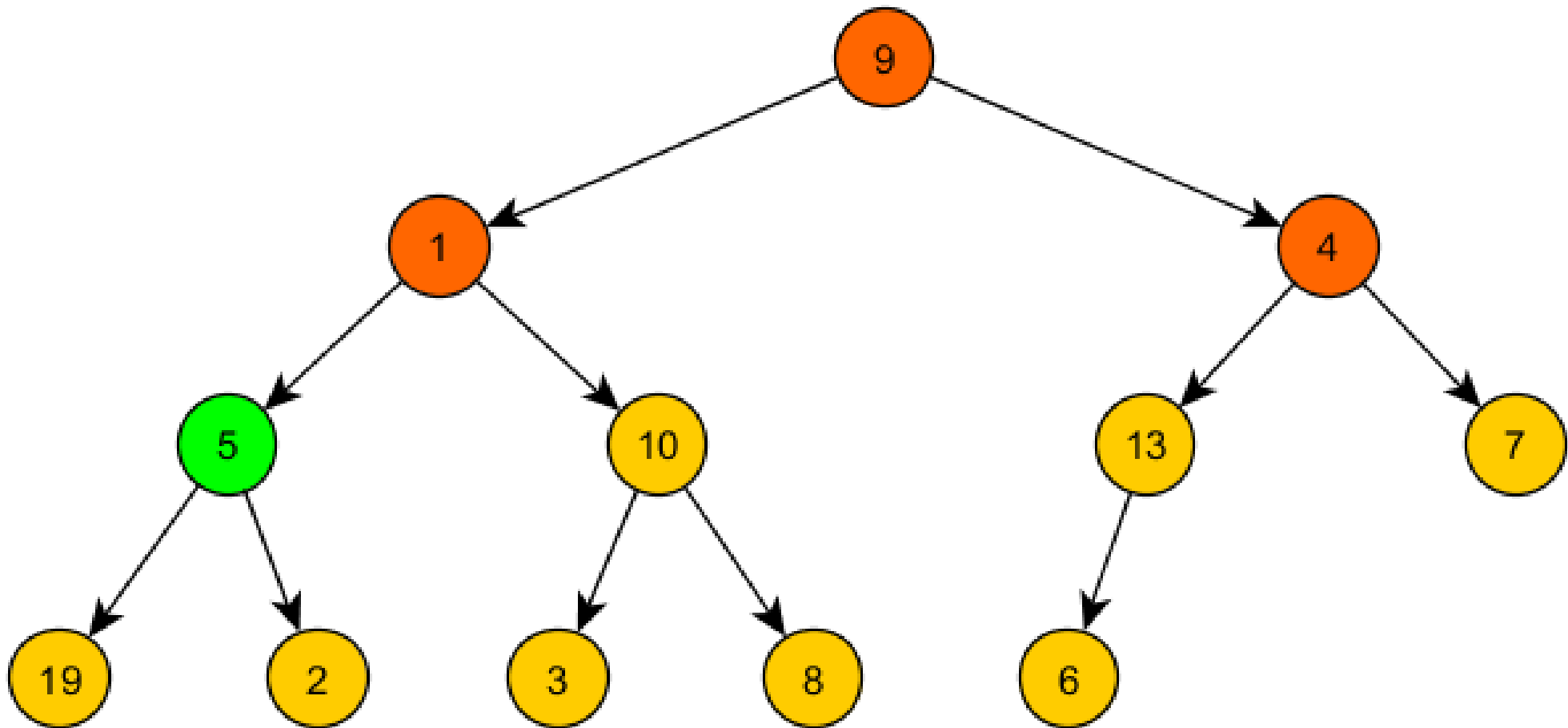




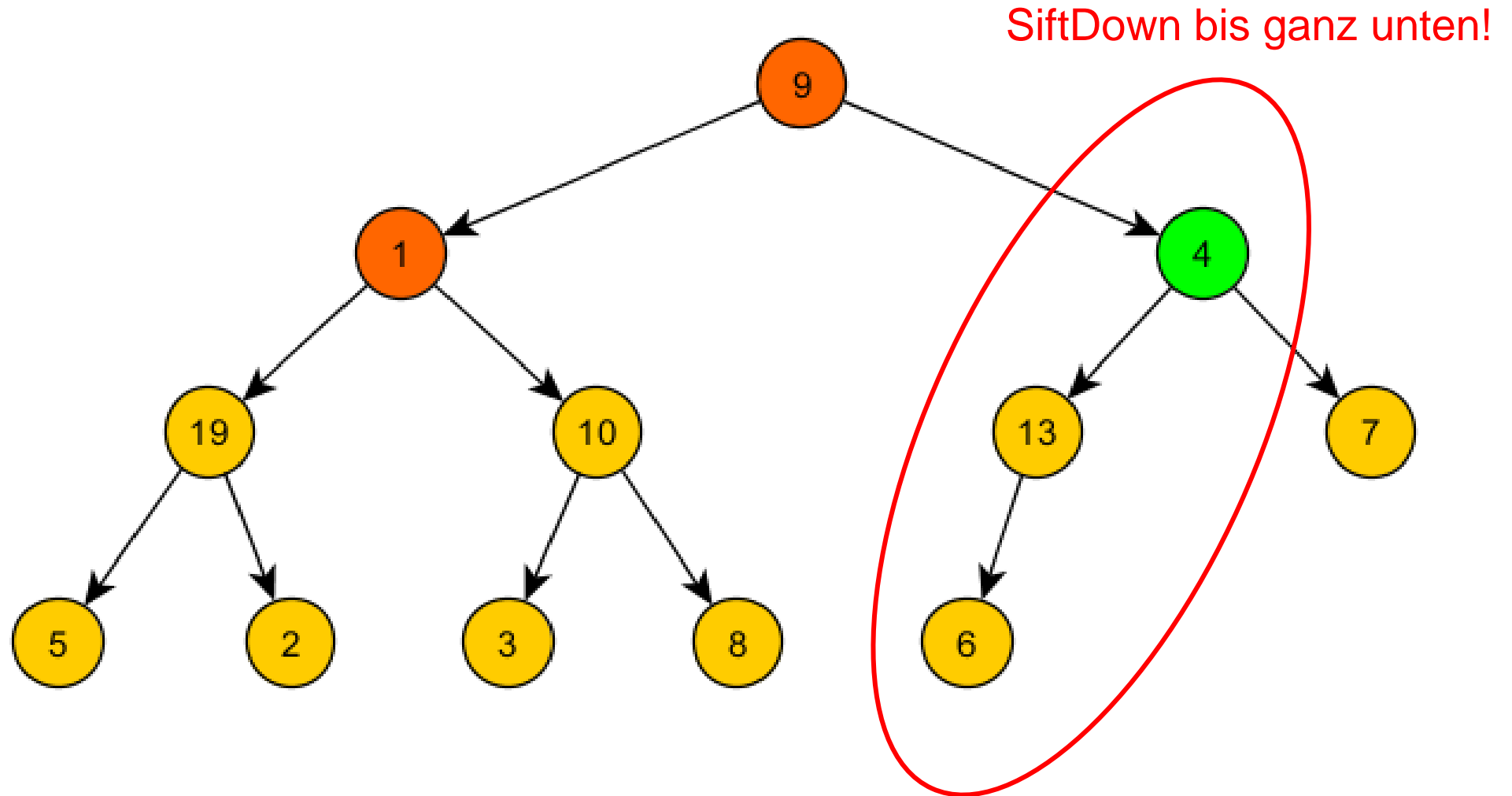
## Floyd Heap-Aufbau Beispiel (2)



## Floyd Heap-Aufbau Beispiel (3)

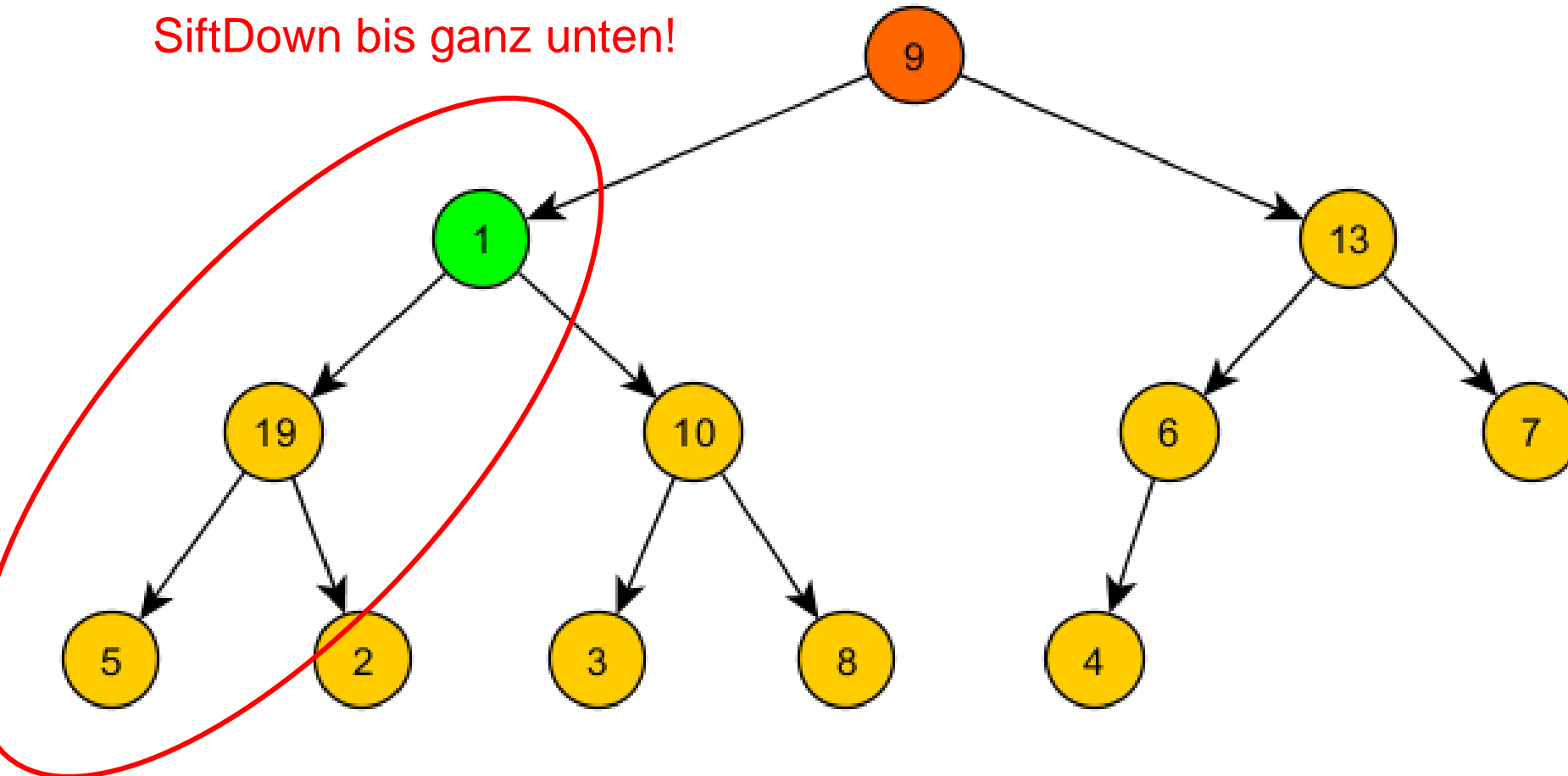


## Floyd Heap-Aufbau Beispiel (4)

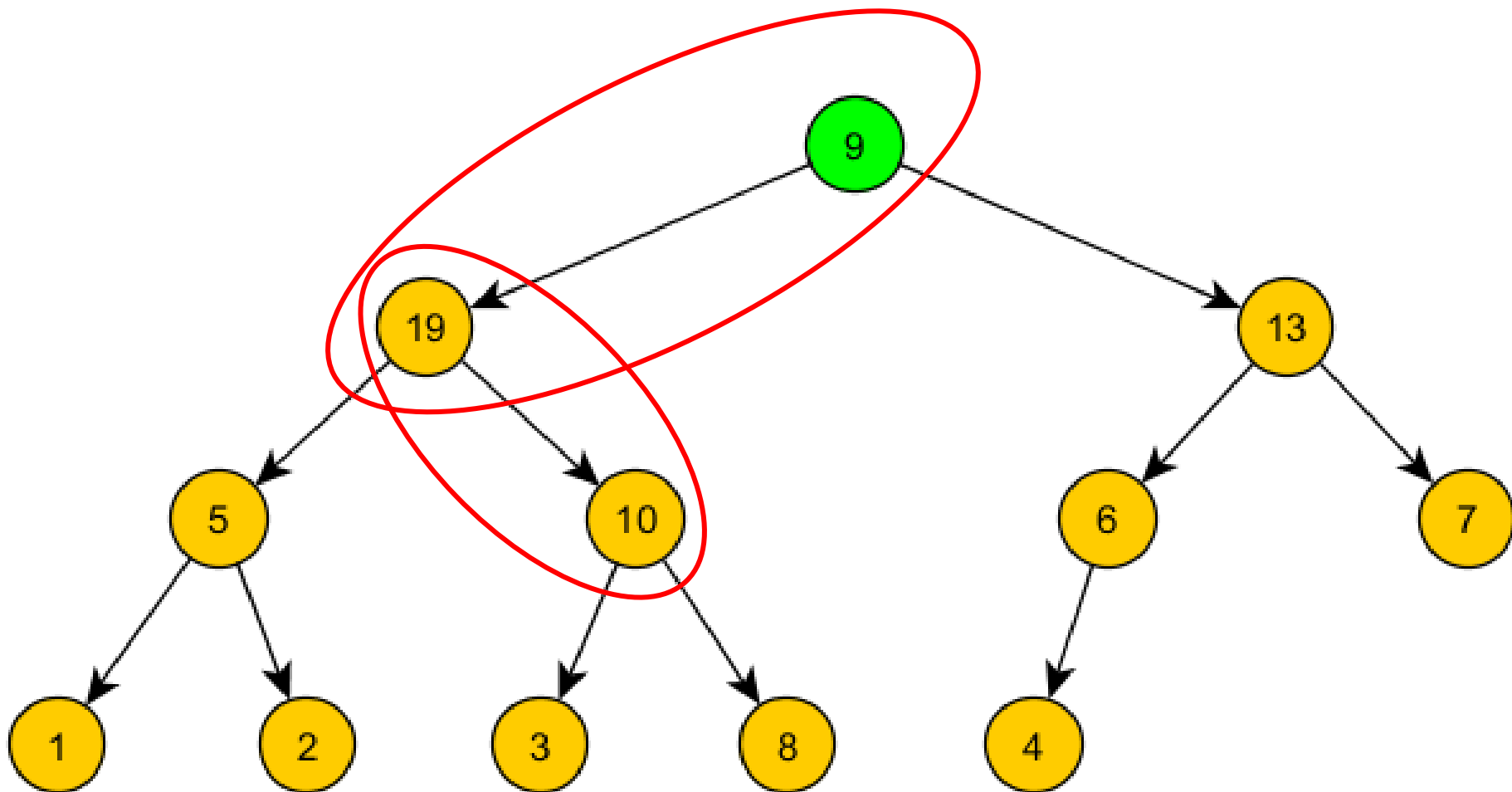


## Floyd Heap-Aufbau Beispiel (5)

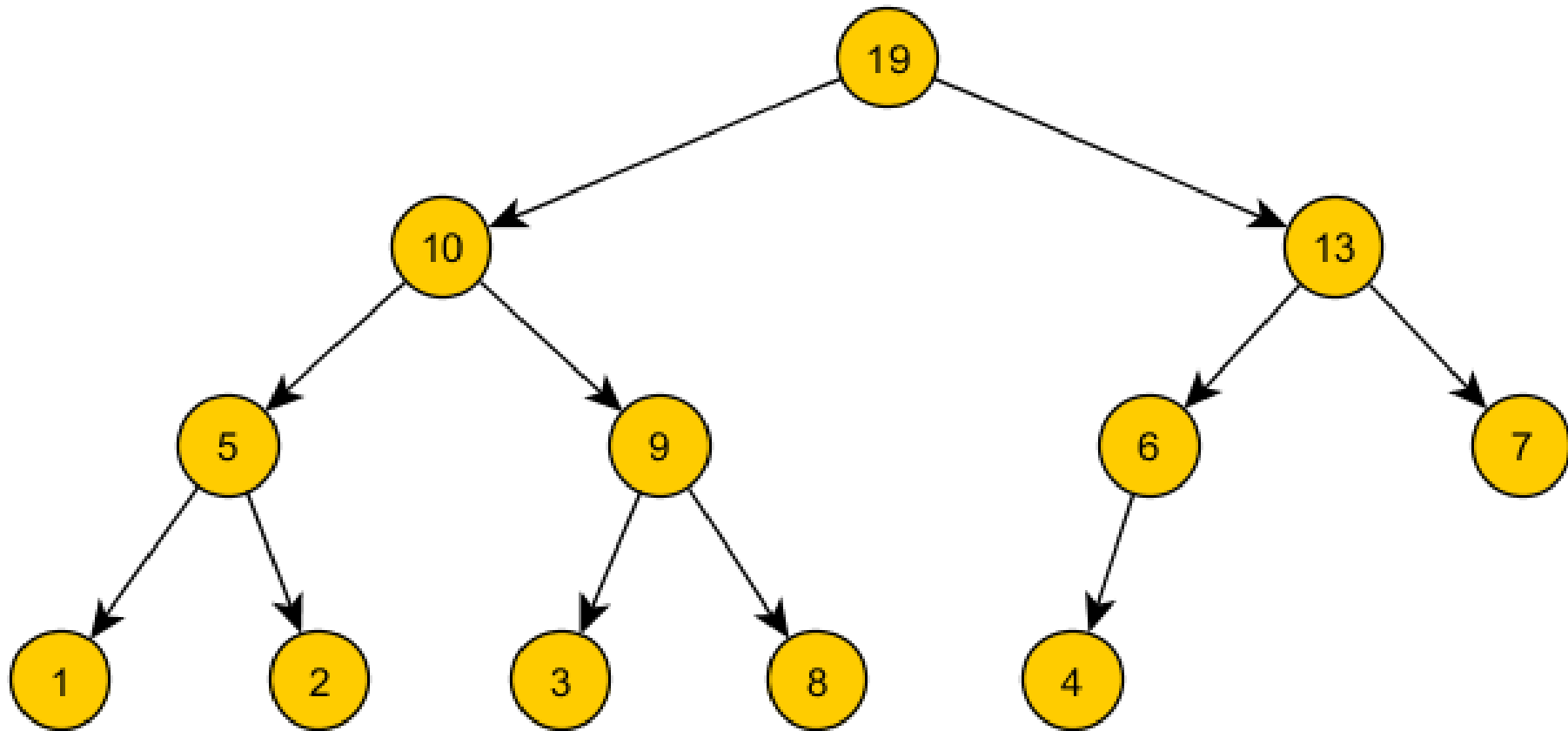
SiftDown bis ganz unten!



## Floyd Heap-Aufbau Beispiel (6)



## Floyd Heap-Aufbau Beispiel (7) - Schlussbild



## Selbststudium

1. Gegeben sei ein Array mit den folgenden Zahlen:

[12, 8, 3, 17, 22, 10, 3, 33, 1, 21]

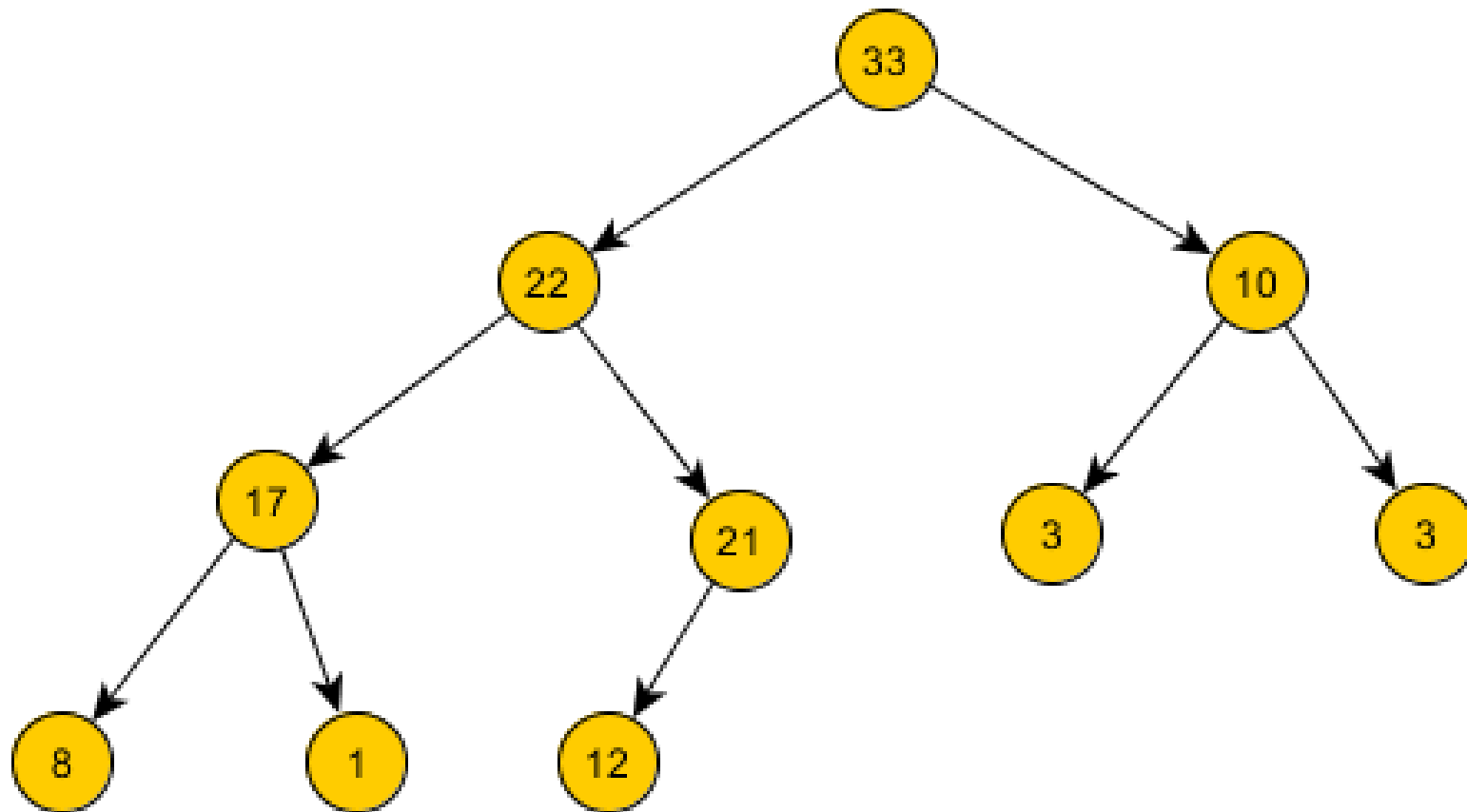
1. Führen Sie den Heap-Aufbau nach Floyd aus, um aus den Daten ein Max-Heap zu erzeugen
2. Führen Sie auf dem erzeugten Heap einen HeapSort aus

2. Vervollständigen Sie im Script auf der letzten Seite die beiden Abschnitte:

1. Array-Indizes der Blätter (abhängig von n) ergänzen
2. Programmcode am Ende des Scripts unter «Nimmt man nun schrittweise die...» Achtung, max. 3 Zeilen Code (siftUp, siftDown bereits vorhanden)

3. Machen Sie Programmierübung 2 – HeapSort (Beschreiben in «Anleitung für Selbststudium»)

## Lösung Aufgabe 1:





## Lösung Aufgabe 2:

### Wurzelement an Index: 0

In einem Array mit  $n$  Elementen haben die Blätter die Indizes

von  $\underbrace{(n-1-1)/2 + 1}_{n-1} = n/2$

bis  $n-1$  *Vorgänger des letzten Elementes ist letzter innerer Knoten*

Nimmt man nun schrittweise rückwärts die restlichen Knoten dazu, muss die Ordnungs-Eigenschaft ggf. mittels eines *siftDown* des jeweils neu betrachteten Knoten etabliert werden.

*for ( i = n/2 - 1; i >= 0; i-- ) siftDown( i );*

### Wurzelement an Index: 1

Blätter von:  $(n/2) + 1$

Blätter bis:  $n$

Schleife: *for ( i = n/2; i >= 1; i-- ) siftDown( i );*