1. **Priority Queues**

**Allgemeines**

**Heaps**

**HeapSort**

**Binomial Heaps**

**1.1Prioritätswarteschlangen**

M: Menge von Elementen

key(e): Priorität von Element e

Operationen:

• M.build({e1, . . . , en}): M = {e1, . . . , en}

• M.insert(Element e): M = M ∪ e

• Element M.min(): gib Element e mit minimaler Priorität key(e) zurück

• Element M.deleteMin():

entferne Element e mit minimaler Priorität key(e) und gib es zurück

**1.1.2Adressierbare Prioritätswarteschlangen**

Zusätzliche Operationen für adressierbare Priority Queues:

• Handle insert(Element e): wie zuvor, gibt aber ein Handle (Referenz / Zeiger) auf

das eingefügte Element zurück

• remove(Handle h): lösche Element spezifiziert durch Handle h

• decreaseKey(Handle h, int k):

reduziere Priorität des Elements mit Handle h auf Wert k

(je nach Implementation evtl. auch um Differenz k)

• M.merge(Q): M = M ∪ Q; Q = ∅;

Priority Queue mittels unsortierter Liste:

• build({e1, . . . , en}): Zeit **O(n)**

• insert(Element e): Zeit **O(1)**

• min(), deleteMin(): Zeit **O(n)**

Priority Queue mittels sortierter Liste:

• build({e1, . . . , en}): Zeit **O(n log n)**

• insert(Element e): Zeit **O(n)**

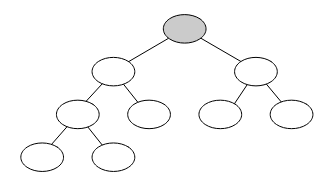
• min(), deleteMin(): Zeit **O(1)**

⇒ bessere Struktur als eine Liste notwendig!

**1.2 Haufen:**

**WDL:BinBaum**

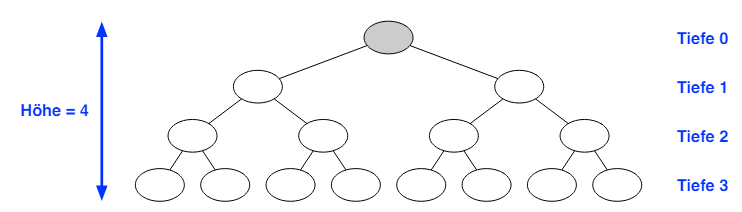
• ein Binärbaum ist ein spezieller Baum, bei dem jeder Knoten maximal zwei direkte Nachfolger (“Kinder”) hat

• ein nicht-leerer Binärbaum hat eine Wurzel, deren linke und rechte Nachfolger ebenfalls wieder Binärbäume sind

• die Tiefe eines Knotens ist die Anzahl der Kanten bis zur Wurzel

• die Höhe des Binarbaums ist die maximale Tiefe eines Knotens plus eins

**WDL: Vollständige Binärbaum**

• ein Binärbaum ist vollständig , wenn jeder innere Knoten zwei Kinder hat und alle Blatter die gleiche Tiefe haben

• ein vollständiger Binärbaum der Höhe h ≥ 1 hat 2^h − 1 Knoten, davon 2^(h−1) Blätter

• ein vollständiger Binärbaum mit n Knoten hat Höhe ⌊log2(n)⌋ + 1

**1.2.2 Bin Heap**

Sei G = (V,E) ein Binärbaum mit Wurzel w ∈ V und |V| = n. Jeder Knoten in v ∈ V sei mit einem Wert key(v) verknüpft, die Werte seien durch ≤, ≥ geordnet.

G heißt binärer Heap, falls folgende zwei Eigenschaften erfüllt sind:

1) G ist fast vollständig (Form-Invariante), d.h. alle Ebenen sind vollständig, außer der untersten Ebene, die von links her nur bis zu einem bestimmten Punkt gefüllt sein muss.

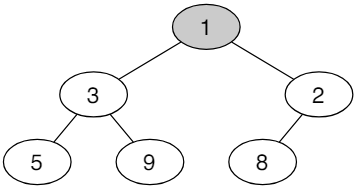
2) G erfüllt die Min-Heap-Eigenschaft bzw. die Max-Heap-Eigenschaft

(Heap-Invariante), d.h. für alle Knoten v ∈ V, v, w gilt

• Min-Heap: key(v.parent) ≤ key(v)

• Max-Heap: key(v.parent) ≥ key(v)

Entsprechend der Heap-Eigenschaft heißt G dann Min-Heap bzw. Max-Heap

Bsp.:

• key(v) sind hier natürliche Zahlen

• der Binärbaum ist fast vollständig (Form-Invariante)

• der Baum erfüllt die Min-Heap-Eigenschaft: (Heap-Invariante)

• für alle Knoten v (außer Wurzel) gilt key(v.parent) ≤ key(v)

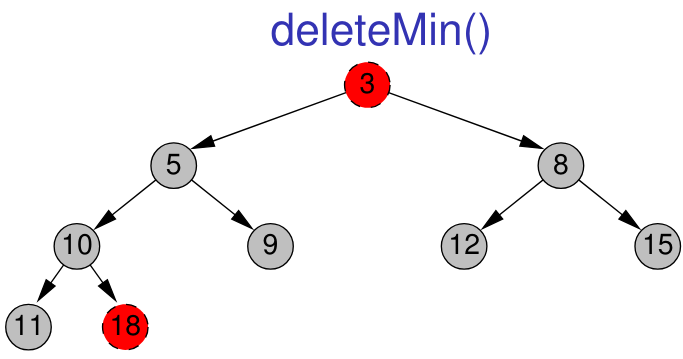
**1.2.3 Priority Queue mit binärem Min-Heap**

**1.2.3.1min ()**• In einem binären Min-Heap steht das kleinste Element in der Wurzel

→ Operation min() ist trivial in **O(1)** zu implementieren

**1.2.3.2 deleteMin()**

• Bei deleteMin() müssen wir nach Entfernen der Wurzel die Form-Invariante und die

Heap-Invariante wieder herstellen

Element deleteMin(Heap<Element> H) {

Element min = Wurzel von H;

ersetze Wurzel von H mit letztem Knoten in H; // Form-Invariante

siftDown(H, Wurzel von H); // Heap-Invariante

return min;

}

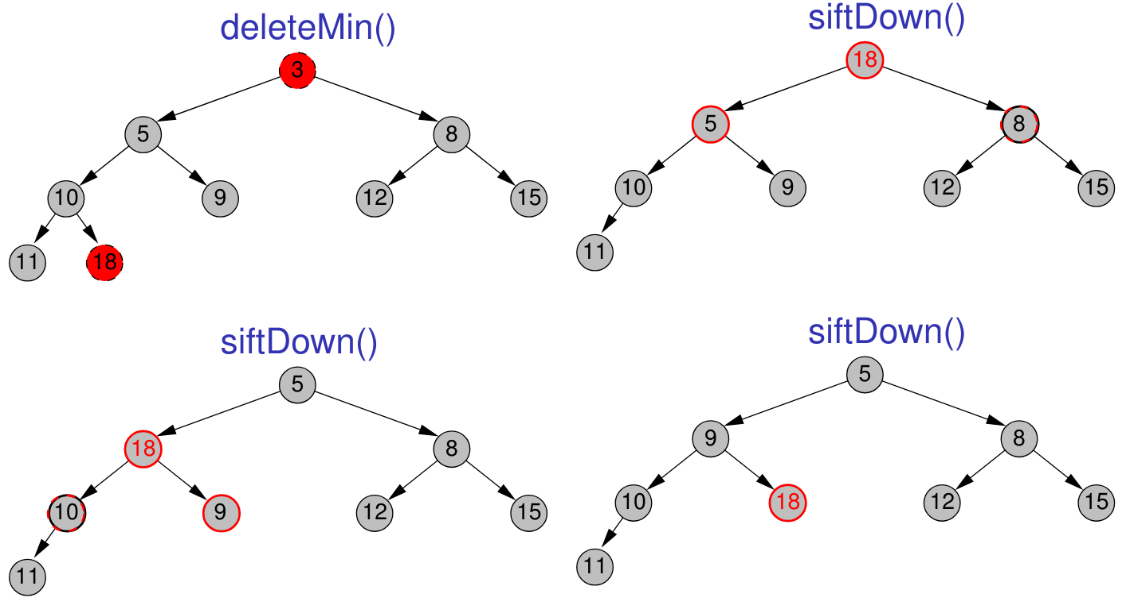
• siftDown(H, v) lässt den Knoten v im Heap H absinken, bis die Heap-Invariante wieder hergestellt ist

• Laufzeit deleteMin(): **O(1) + Laufzeit von siftDown()**

**1.2.3.3 shiftDown()**

• siftDown(H, v) stellt Heap-Invariante wieder her

• Voraussetzung: nur Knoten v verletzt die Heap-Invariante

siftDown(Heap<Element> H, Knoten v) {

if (v ist Blatt) return;

Knoten m;

if (key(v.left) < key(v.right))

m = v.left;

else m = v.right;

if (key(m) < key(v)) {

tausche Inhalt von m und v;

siftDown(H, m);

}}

• Laufzeit siftDown(): **O(log n)** (da jede Ebene im Heap maximal einmal besucht wird)

**1.2.3.4 insert()**

• insert() fügt Element am Ende des Heaps ein (erhält Form-Invariante), die Heap-Invariante wird durch siftUp() wieder hergestellt

insert(Heap<Element> H, Element e) {

Knoten v = füge e am Ende von Heap H ein;

siftUp(H, v);}

• siftUp(H, v) lässt Knoten v im Heap H aufsteigen, bis die Heap-Invariante wieder hergestellt ist

• Laufzeit insert**(): O(1) + Laufzeit von siftUp()**

**1.2.3.5 shiftUp()**

• siftUp(H, v) stellt Heap-Invariante wieder her

• Voraussetzung: nur Knoten v verletzt die Heap-Invariante

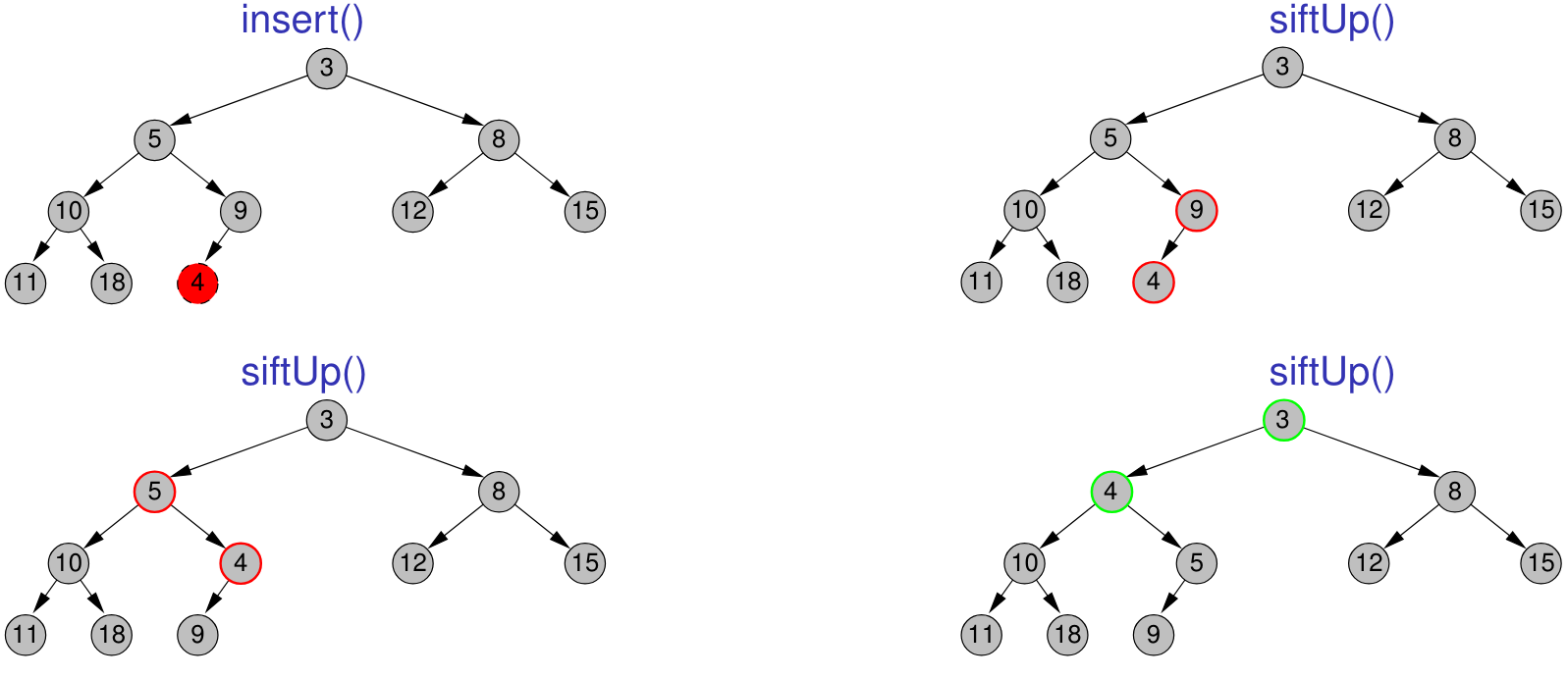
siftUp(Heap<Element> H, Knoten v) {

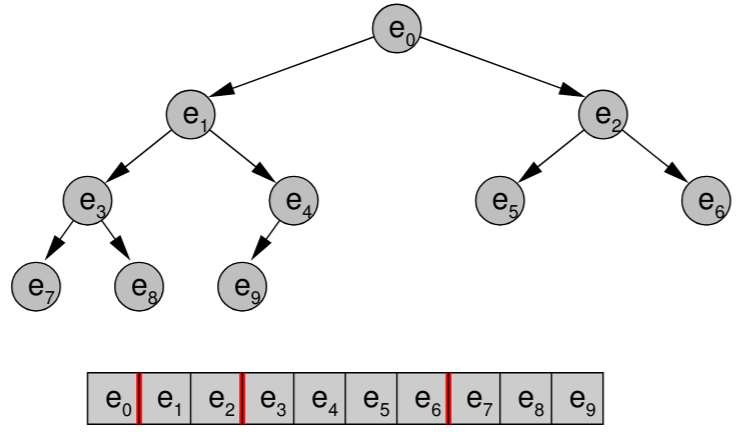
while (v nicht Wurzel ∧ key(v.parent) > key(v) ) {

tausche Inhalt von v mit v.parent;

v = v.parent;

}}

• Laufzeit siftUp(): **O(log n)** (da jede Ebene im Heap **maximal einmal** besucht wird)

**1.2.4 Bin Heap als Felder**

• Kinder von Knoten H[i] sind in H[2i + 1] und H[2i + 2]

• Vater von Knoten H[i] ist H[⌊(i − 1)/2⌋]

• Form-Invariante: H[0], . . . , H[n − 1] besetzt

• Heap-Invariante: H[i] ≤ min {H[2i+1], H[22+2]}

**1.2.4.2: insert(e)**

• Form-Invariante: H[n] = e; siftUp(n); n++;

• Heap-Invariante:

vertausche e mit seinem Vater bis

key(H[⌊(k − 1)/2⌋]) ≤ key(e) für e in H[k] (oder e in H[0])

siftUp(i) {

while (i > 0 ∧ key(H[⌊(i − 1)/2⌋]) > key(H[i])) {

swap(H, i,⌊(i − 1)/2⌋);

i = ⌊(i − 1)/2⌋;

}}

• Laufzeit: **O(log n)**

**1.2.4.3 Build():**

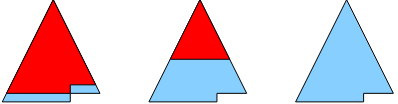
• naiver Ansatz um Heap aufzubauen: n-mal insert() ausführen

→ Laufzeit: **O(n log n)**

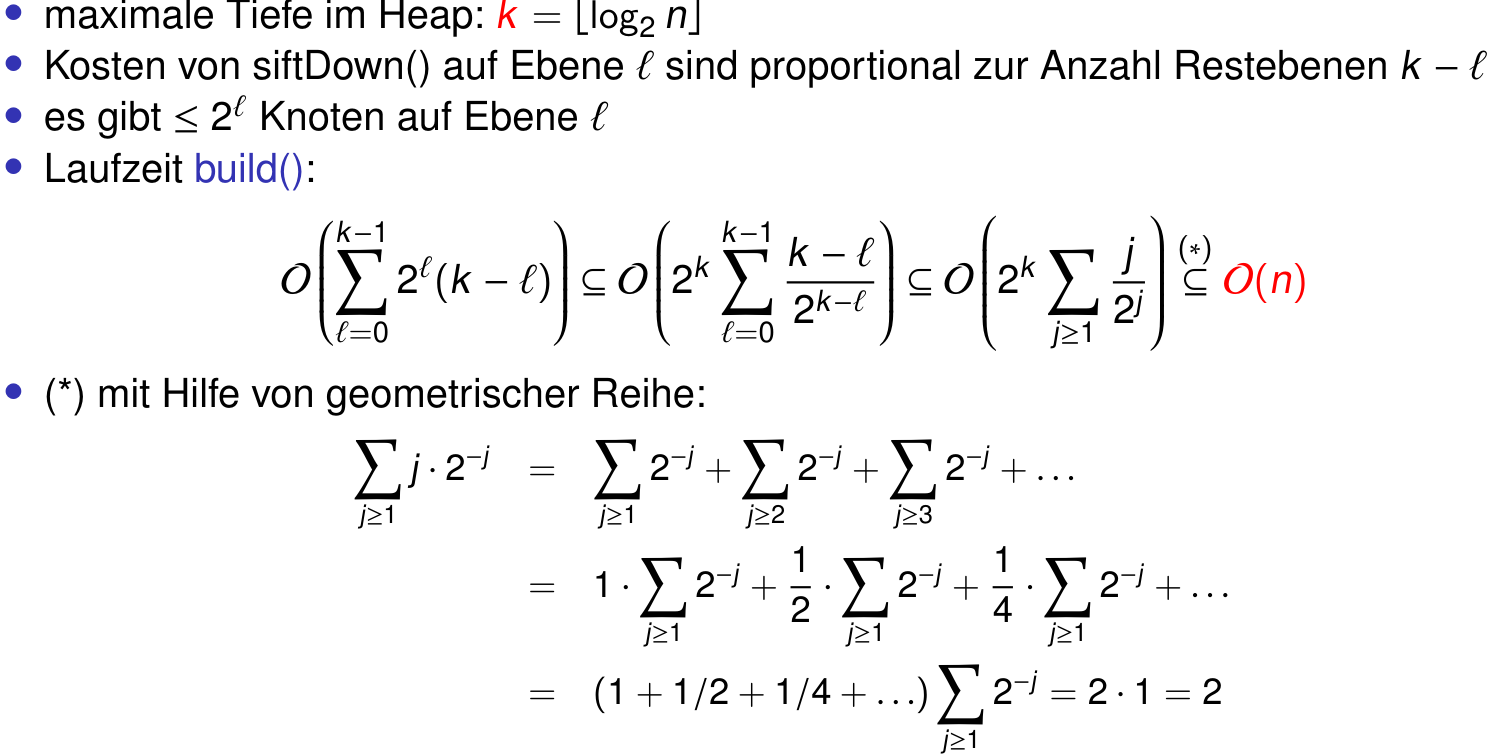
• effizienter Ansatz:

• die n Elemente unsortiert in den Binärbaum schreiben

• für jede Ebene von unten nach oben: siftDown() für jeden Knoten in der Ebene

 • Blätter müssen nicht behandelt werden

→ Laufzeit: **O(n) + O(n log n), oder besser abgeschätzt: O(n)**, beweis folgt:



**1.2.4.4 decreaseKey()**

• Reduziere Priorität eines bestimmten Elements und stelle Heap-Invariante wieder her mittels siftUp()

decreaseKey(Heap<Element> H, Knoten v, int k) {

if (k > key(v))

error(”neuer key größer als alter!”);

key(v) = k;

siftUp(H, v);}

• Laufzeit: **O(log n)**

**1.2.5 Prioritätswarteschlangen: Übersicht:**

**1.3 HeapSort**

**1.3.1 Idee:** verbessere SelectionSort indem eine Priority Queue (mittels binärem Min-Heap) verwendet wird

→ **HeapSort**

**1.3.2 Algorithmus:**

• verwende binären Min-Heap als Array

• initialisiere Min-Heap mit build() → Laufzeit: **O(n)**

• wiederhole deleteMin() bis Heap leer → Laufzeit: **O(n log n)**

• Elemente von deleteMin() landen im Array hinten (was nicht mehr zum Heap gehört)

**1.3.3 Pseudocode:**

HeapSort(Element[] H, int n ) {

build( H [ 0 ], . . . , H [ n − 1 ]);

for ( i = n − 1; i ≥ 1; i −− ) {

swap( H, 0, i);

H.length−− ;

siftDown( H, 0); } }

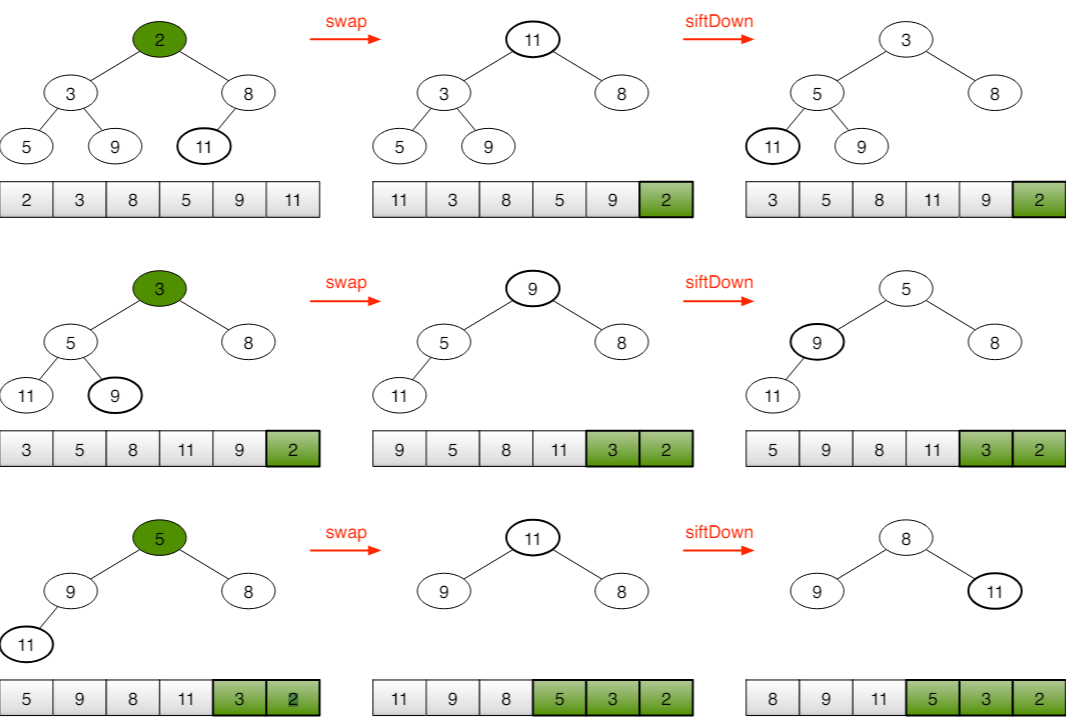
• HeapSort sortiert in-place, aber nicht stabil

• Sortiertes Array entsteht von hinten, also absteigend sortiert

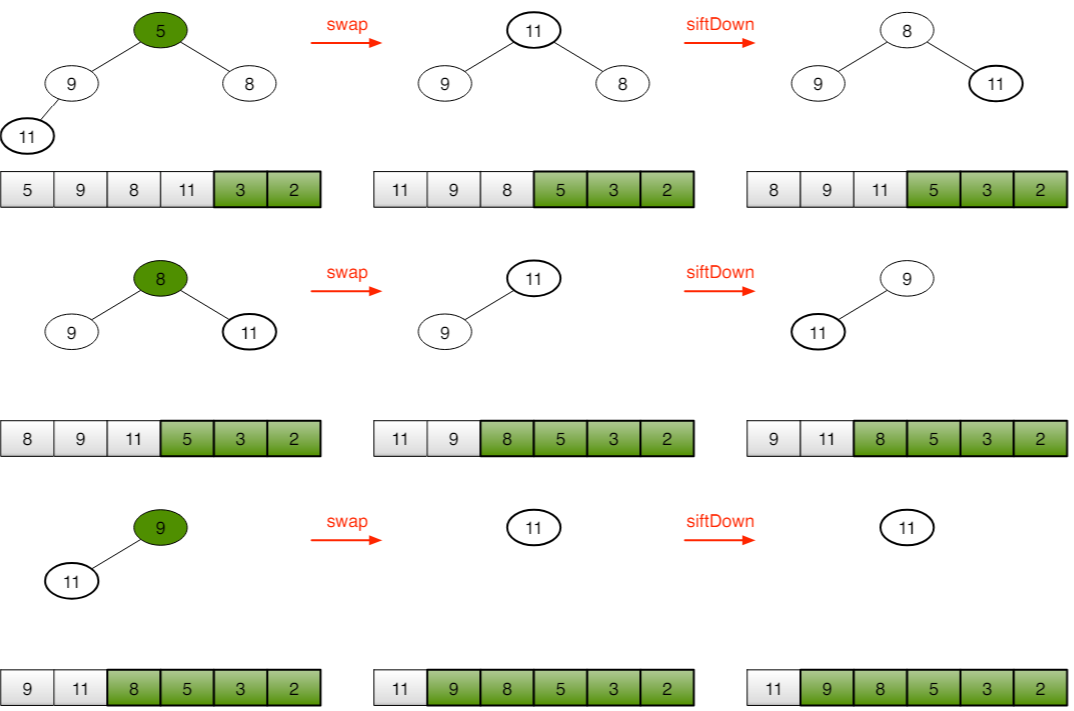
• aufsteigende Sortierung kann mit Max-Heap erzeugt werden

• Laufzeit: **O(n log n)**

Bsp.:



**---------------------------------------------------------------------------**



**SORT\_ALGORITHMUS\_LAUFZEIT\_ZF:  
• SelectionSort**

**• in-place, nicht stabil, Laufzeit O(n^2)**

**• InsertionSort**

**• in-place, stabil, Laufzeit O(n^2), best-case: O(n)**

**• QuickSort**

**• in-place, nicht stabil, Laufzeit worst-case O(n^2), average-case: O(n log n)**

**• MergeSort**

**• benötigt O(n) zusätzlichen Speicher, stabil, Laufzeit O(n log n)**

**• HeapSort**

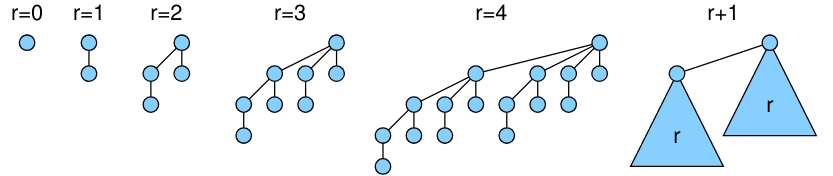
**• in-place, nicht stabil, Laufzeit O(n log n)**

**1.4 Binomial Heap**

**1.4.1 Begriff:** Binomial-Bäume sind rekursiv definiert:

• ein Binomial-Baum vom Rang 0 ist ein einzelner Knoten

• ein Binomial-Baum vom Rang r hat einen Wurzelknoten, dessen Kinder Wurzeln von

Binomial-Bäumen vom Rang r − 1, r − 2, . . ., 2, 1, 0 sind (in dieser Reihenfolge)

**1.4.2 Eigenschaft:**

Ein Binomial-Baum vom Rang r hat folgende Eigenschaften:

