7.1 Heaps

a) Min./max. Zahl. Elemente in Heap der Höhe h?

Höhe 0: min.: ① → 1 $= 2^0 = 2^h$

max.: ① → 1 $= 2^1 - 1 = 2^{h+1} - 1$

(+1)

Höhe 1: min.: ① ② → 2 $= 2^1 = 2^h$

max.: ① ② ③ → 3 $= 2^2 - 1 = 2^{h+1} - 1$

(+1)

Höhe 2: min.: ① ② ③ ④ → 4 $= 2^2 = 2^h$

max.: ① ② ③ ④ ⑤ ⑥ ⑦ → 7 $= 2^3 - 1 = 2^{h+1} - 1$

(+1)

Höhe 3: min.: ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ → 8 $= 2^3 = 2^h$

max.: ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫ ⑬ ⑭ ⑮ → 15 $= 2^4 - 1 = 2^{h+1} - 1$

⇒ min.: 2^h

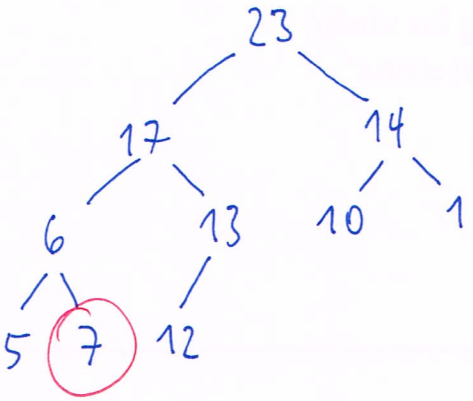
max.: $2^{h+1} - 1$

b) Ist jedes sortierte Array ein Min-Heap?

Ja!

Die Funktion Build-Min-Heap, angewendet auf sort. Array, ändert nichts: Alle Aufrufe von Min-Heapify stellen fest, dass Min-Heap-Eigenschaft schon erfüllt ist.

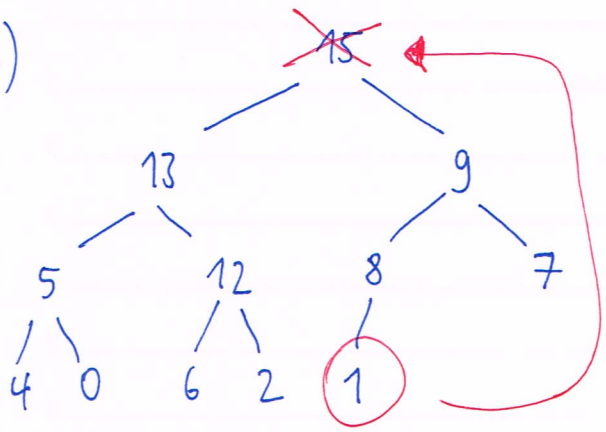
c) $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ Max-Heap?



Nein!

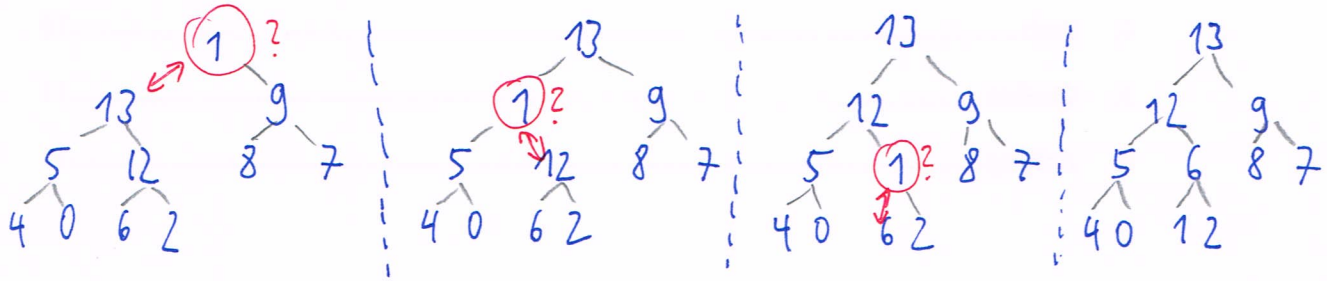
$7 > 6$ ↯

d)

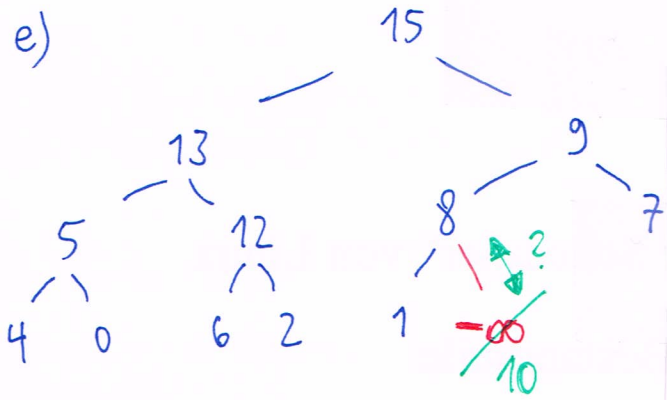



```
HEAP-EXTRACT-MAX(A)
1  if A.heap-größe < 1
2        error "Heap-Unterlauf"
3  max = A[1]
4  A[1] = A[A.heap-größe]
5  A.heap-größe = A.heap-größe − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

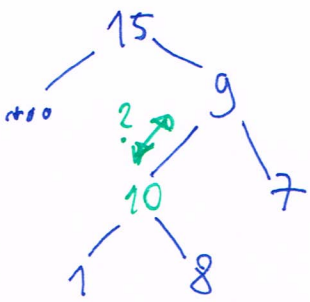
Heap-Extract-Max()

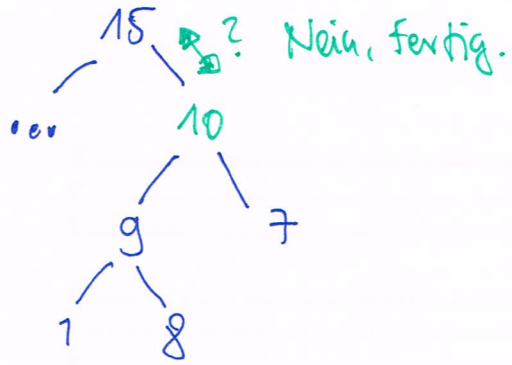


e)



Max-Heap-Insert (A, 10)





MAX-HEAP-INSERT(A, *schlüssel*)

```
1  A.heap-größe = A.heap-größe + 1
2  A[A.heap-größe] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-größe, schlüssel)
```

HEAP-INCREASE-KEY(A, i, *schlüssel*)

```
1  if schlüssel < A[i]
2      error "neuer Schlüssel ist kleiner als der aktuell
3  A[i] = schlüssel                                        Schlüssel"
4  while i > 1 und A[PARENT(i)] < A[i]
5      vertausche A[i] mit A[PARENT(i)]
6      i = PARENT(i)
```

4/9

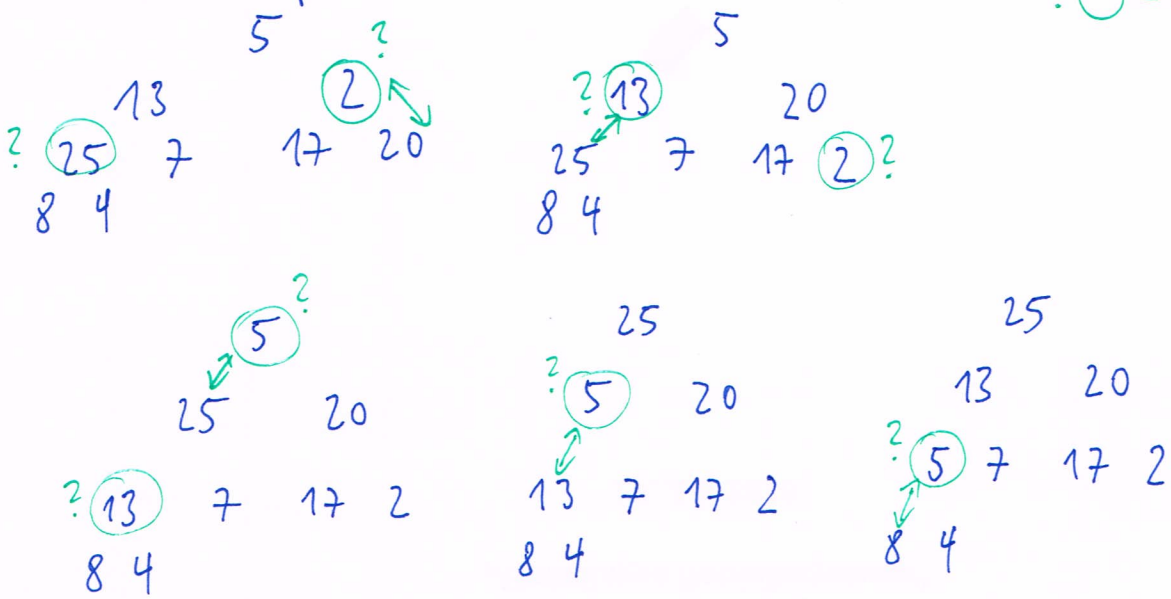
```
# SKA 7.2 a), Heapsort in Python

def laenge(liste):
    return len(liste) - 1
    # Index 0 ignorieren

def heap_groesse(heap):
    return heap[0]
    # Heapgroesse in Pos. 0 !

def setze_heap_groesse(heap,n):
    heap[0] = n

def links(i): return i*2
def rechts(i): return i*2 + 1

def vertausche_in (heap,x,y):
    heap[x],heap[y] = heap[y],heap[x]

def MaxHeapify (A,i):
    l = links(i)
    r = rechts(i)
    if (l <= heap_groesse (A)) and (A[l] > A[i]):
        maximum = l
    else:
        maximum = i
    if (r <= heap_groesse (A)) and (A[r] > A[maximum]):
        maximum = r
    if maximum != i:
        # A[i],A[maximum] = A[maximum],A[i]
        vertausche_in (A,i,maximum)
        MaxHeapify (A,maximum)

def BuildMaxHeap (A):
    setze_heap_groesse (A, laenge(A))
    for i in range(laenge(A)//2, 0, -1):  # downto 1
        MaxHeapify (A,i)

def HeapSort (A):
    BuildMaxHeap (A)
    for i in range(heap_groesse(A), 1, -1):  # downto 2
        vertausche_in (A,1,i)
        setze_heap_groesse (A,heap_groesse(A)-1)
        MaxHeapify(A,1)

print ("Cormen, Abb. 6.2, S. 157, MaxHeapify(A,2)")
A = [10,  # Heap-Groesse!
     16,4,10,14,7,9,3,2,8,1]
print (A[1:])
MaxHeapify (A,2)
print (A[1:],"\n")

print ("Cormen, Abb. 6.3, S. 159, BuildMaxHeap(A)")
A = [0, # ohne Bedeutung
     4,1,3,2,16,9,10,14,8,7]
print (A[1:])
BuildMaxHeap (A)
print (A[1:],"\n")

print ("Cormen, Abb. 6.4, S. 163, BuildMaxHeap(A)")
A = [0, # ohne Bedeutung
     16,14,10,8,7,9,3,2,4,1]
print (A[1:])
HeapSort (A)
print (A[1:],"\n")
```

a) → Python Code

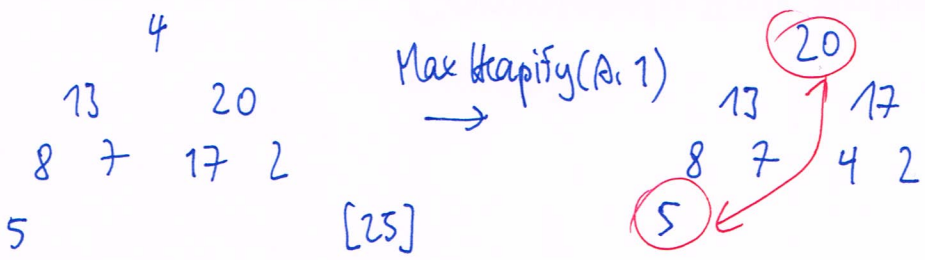
b) $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$

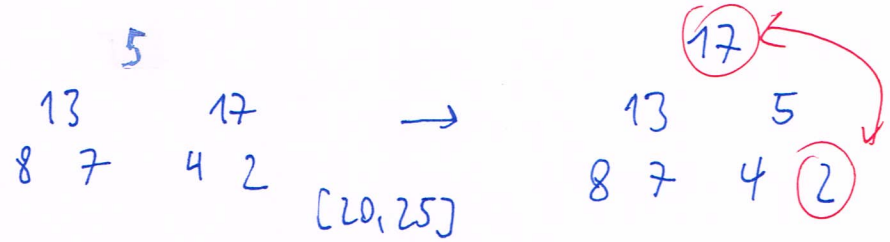
Build Max Heap (A)

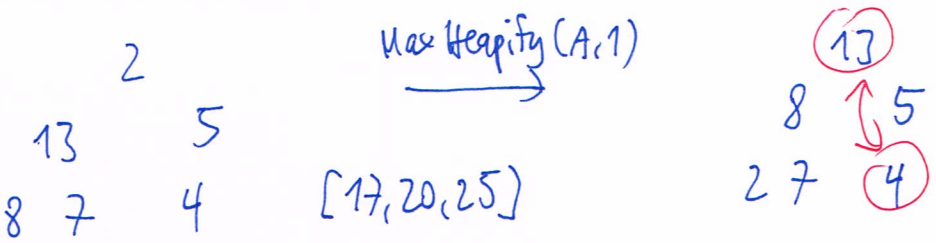
? ◯ = Max Heapify aufrufen

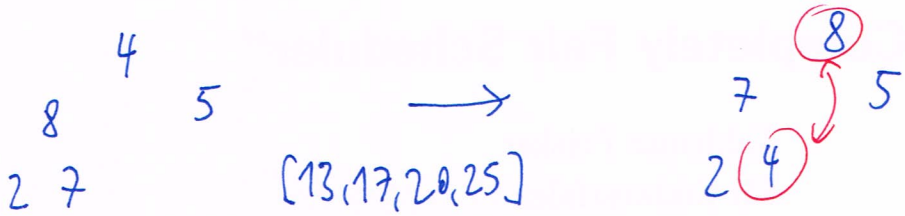


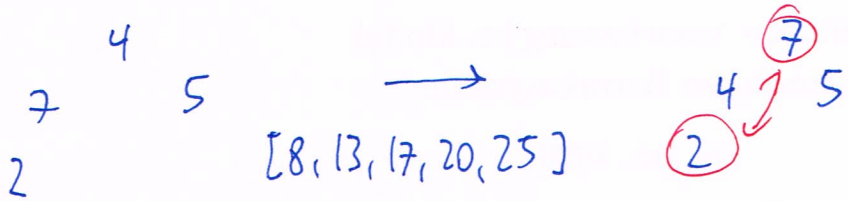
Schleife in Heap Sort: Maximum rausnehmen / tauschen, Heap verkleinern

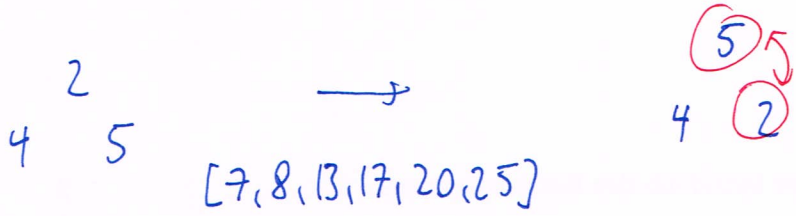


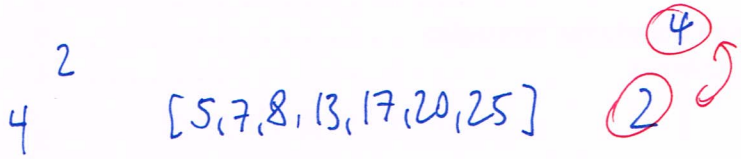










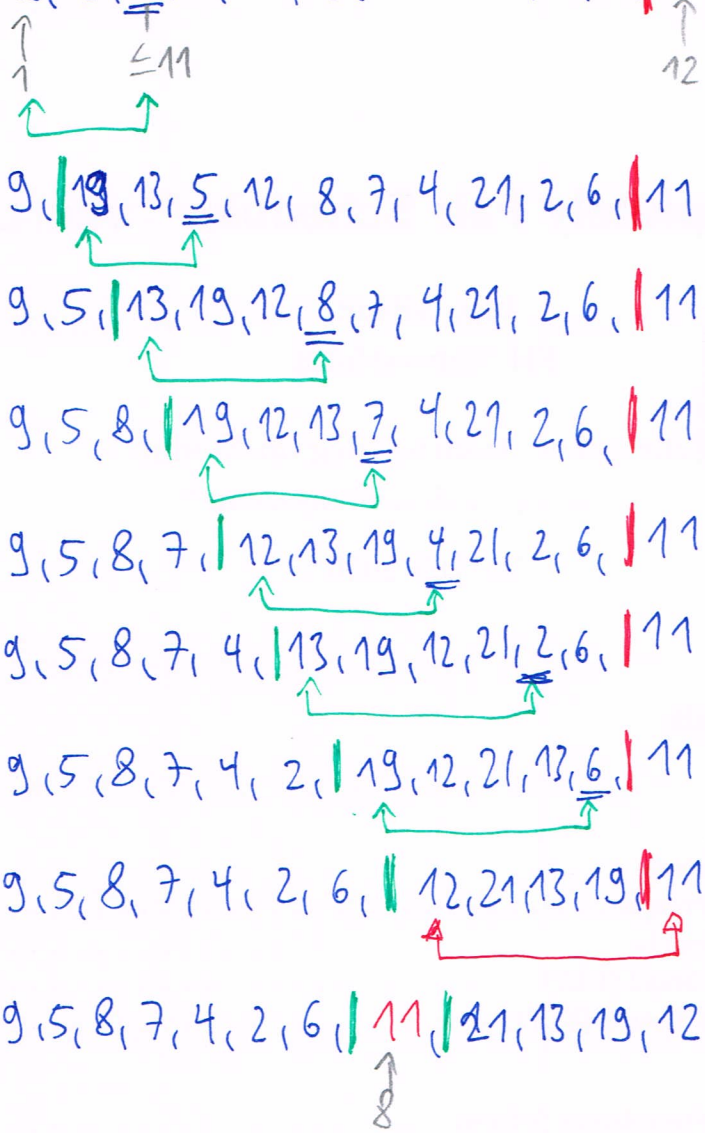


2, [4, 5, 7, 8, 13, 17, 20, 25]

als Liste: fertig!

7.3 Quicksort

a) $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$



(return 8)

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          vertausche
           A[i] mit A[j]
7  vertausche
   A[i + 1] mit A[r]
8  return i + 1
```

c) Quicksort anpassen, so dass nicht-aufsteigend sortiert wird?

↪ In Partition(), Zeile 4: if $A[j] \geq x$

b) $B = \langle 1,3,2,5,6,8,9,10,7,4 \rangle$

8/9

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          vertausche A[i] mit A[j]
7  vertausche A[i + 1] mit A[r]
8  return i + 1
```

QuickSort (B, 1, 10)

q = Partition (B, 1, 10)

1, 3, 2, 5, 6, 8, 9, 10, 7, 4

(1, 3, 2 tauschen mit sich selbst ...)

1, 3, 2, 4, 6, 8, 9, 10, 7, 5

q = 4

Quicksort (B, 1, 3)

q = Partition (B, 1, 3)

1, 3, | 2 (1 tauscht mit sich)

1, 2, 3 q = 2

Zwischenstand: [1, 2, 3, 4, 6, 8, 9, 10, 7, 5]

Quicksort auf 1-elem. Mengen: sofort Ende.

Quicksort (B, 5, 10)

q = partition (B, 5, 10)

6, 8, 9, 10, 7, | 5 (alle > 5 ...)

5, 8, 9, 10, 7, 6 q = 5 [1, 2, 3, 4, 5, 8, 9, 10, 7, 6]

[]

Quicksort (B, 6, 10)
q = Partition (B, 6, 10)

8, 9, 10, 7, | 6 (alle > 6)

6, 9, 10, 7, 8 q = 6 [1, 2, 3, 4, 5, 6, 9, 10, 7, 8]

Quicksort (B, 7, 10)
q = Partition (B, 7, 10)

9, 10, 7, | 8

7, | 10, 9, | 8

7, 8, 9, 10 q = 8 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

↑ fertig

Quicksort (B, 9, 10)
q = Partition (B, 9, 10)

9, | 10

Ende.