SORTING

Algorithms and Data Structures 1
Exercise – 2023S

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Bernhard Anzengruber (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
bernhard.anzengruber-tanase@jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

HEAP REVIEW

Heap-based sorting

- insert keys to be sorted in **PQ**
  - insert into lowest level (leftmost) (→ *structure property*)
  - **upHeap** (→ *order property*)
- iteratively remove the smallest element
  - remove **root** (smallest element)
  - fill the gap with element from the lowest level (→ *structure property*)
  - **downHeap** (→ *order property*)

Complexity

- additional copy of elements to be sorted added to PQ
- N insert operations to build a heap is inefficient (top-down)
- **O(N log N)**

BOTTOM-UP HEAP CONSTRUCTION

Bottom-Up heap construction **in-place** in **O(N)**

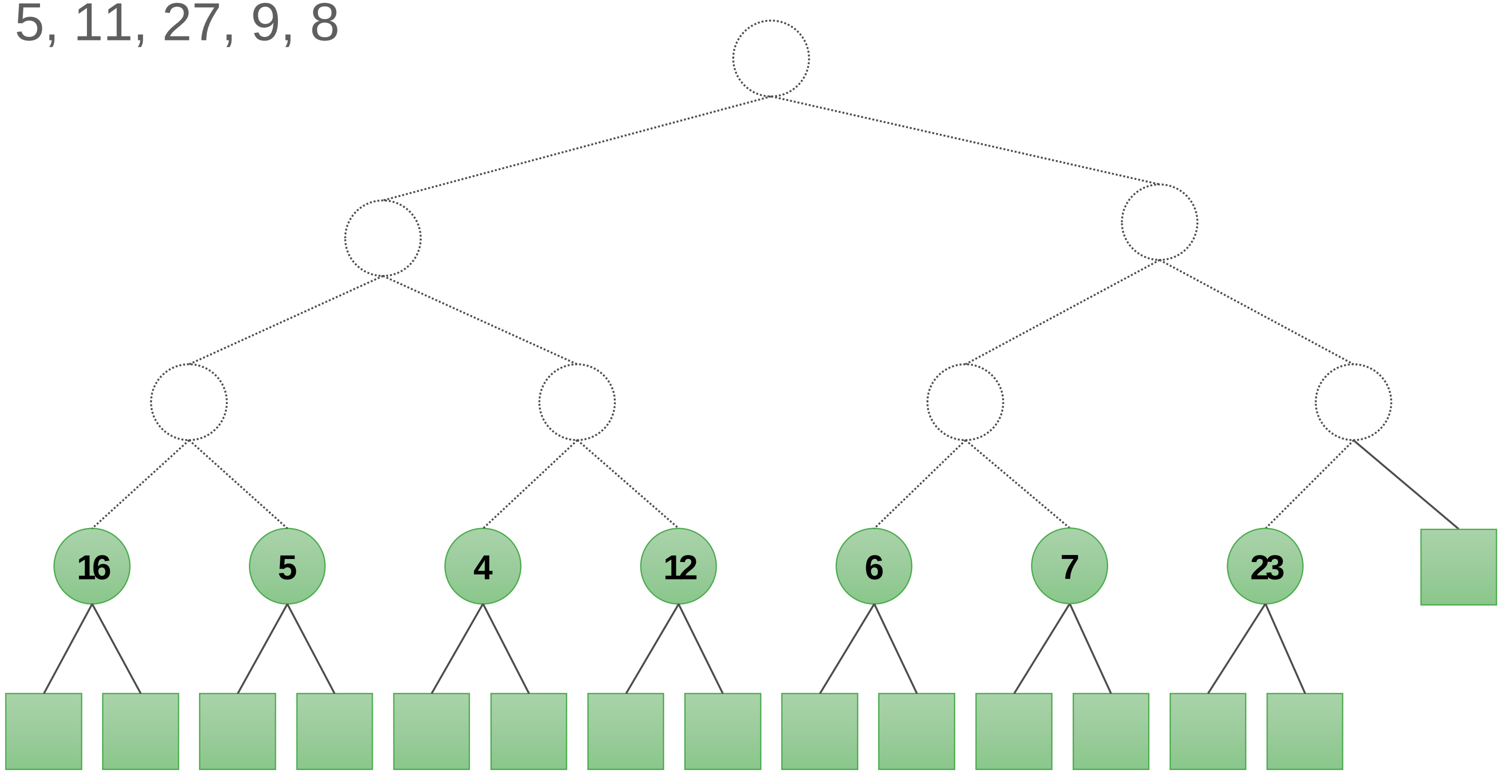
- avoid iterative insertions
- small partial heaps are created from the lowest level upwards (starting halfway, bottom-up)
- each position in the array is seen as the root of a small partial heap
- if node successors are heaps, calling **downHeap** on this node makes its subtree also a heap

BOTTOM-UP HEAP CONSTRUCTION

Create heap from, e.g., 16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

Step 1: Create 1-element heaps out of the first **(n+1)/2 elements** (trivial)

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

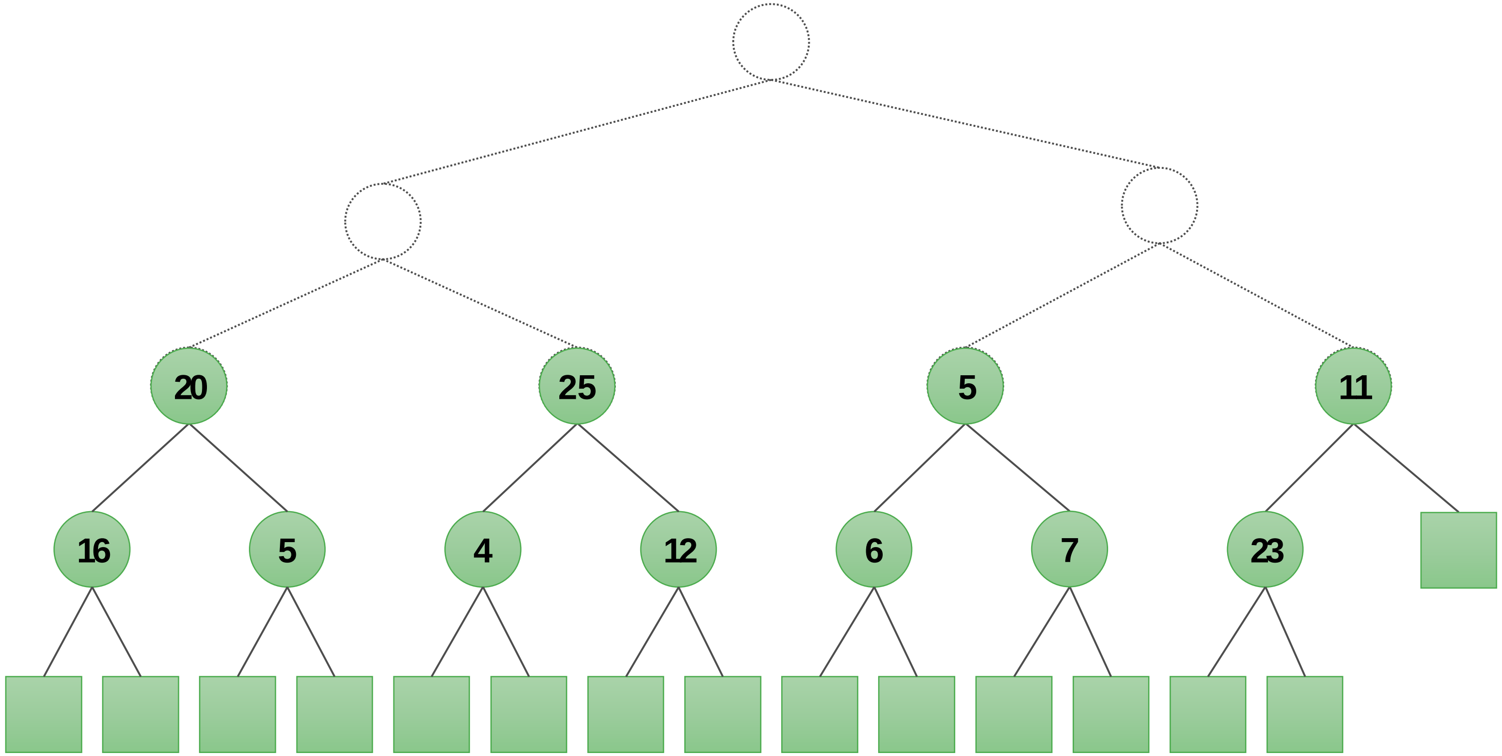


Institute of Pervasive Computing

BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

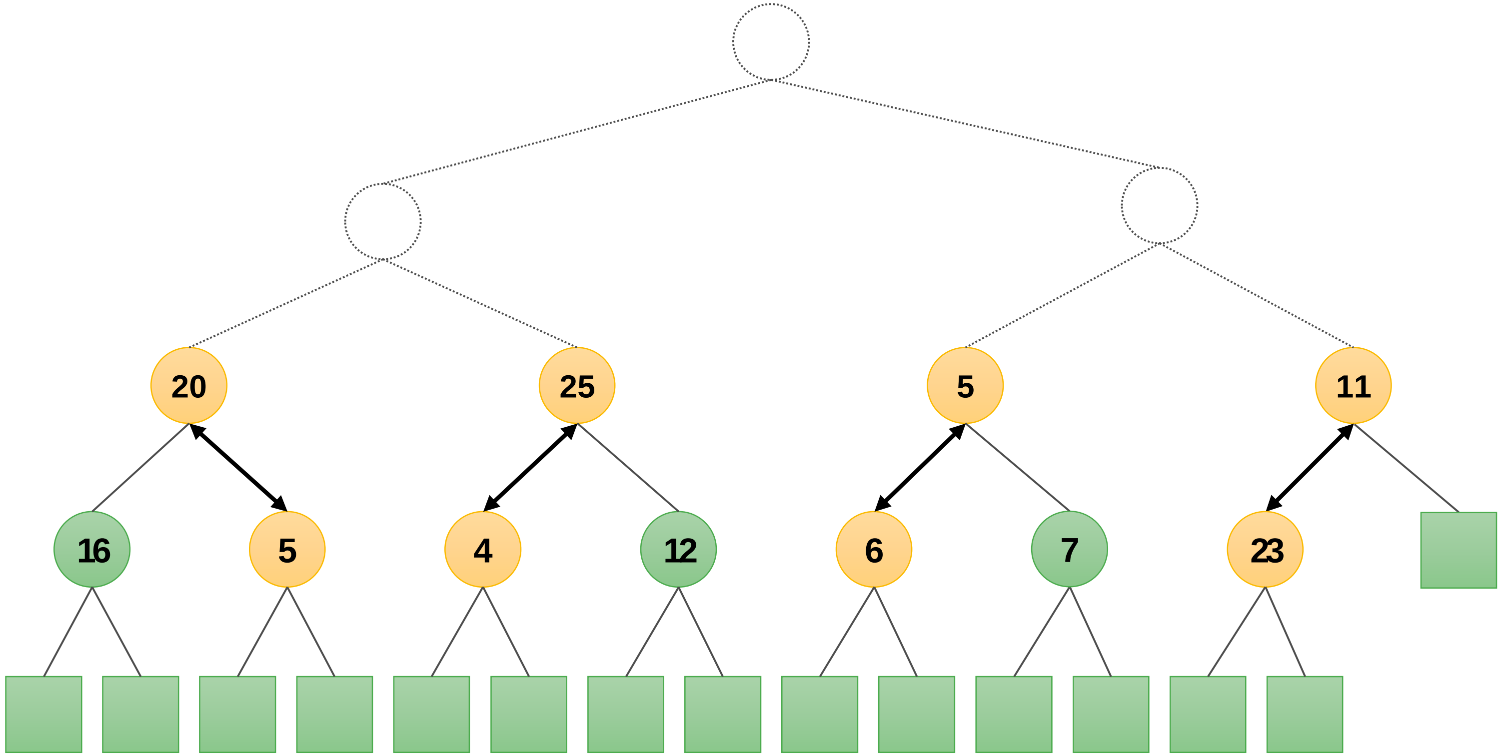
Step 2: Create 3-element heaps from trivial heaps



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

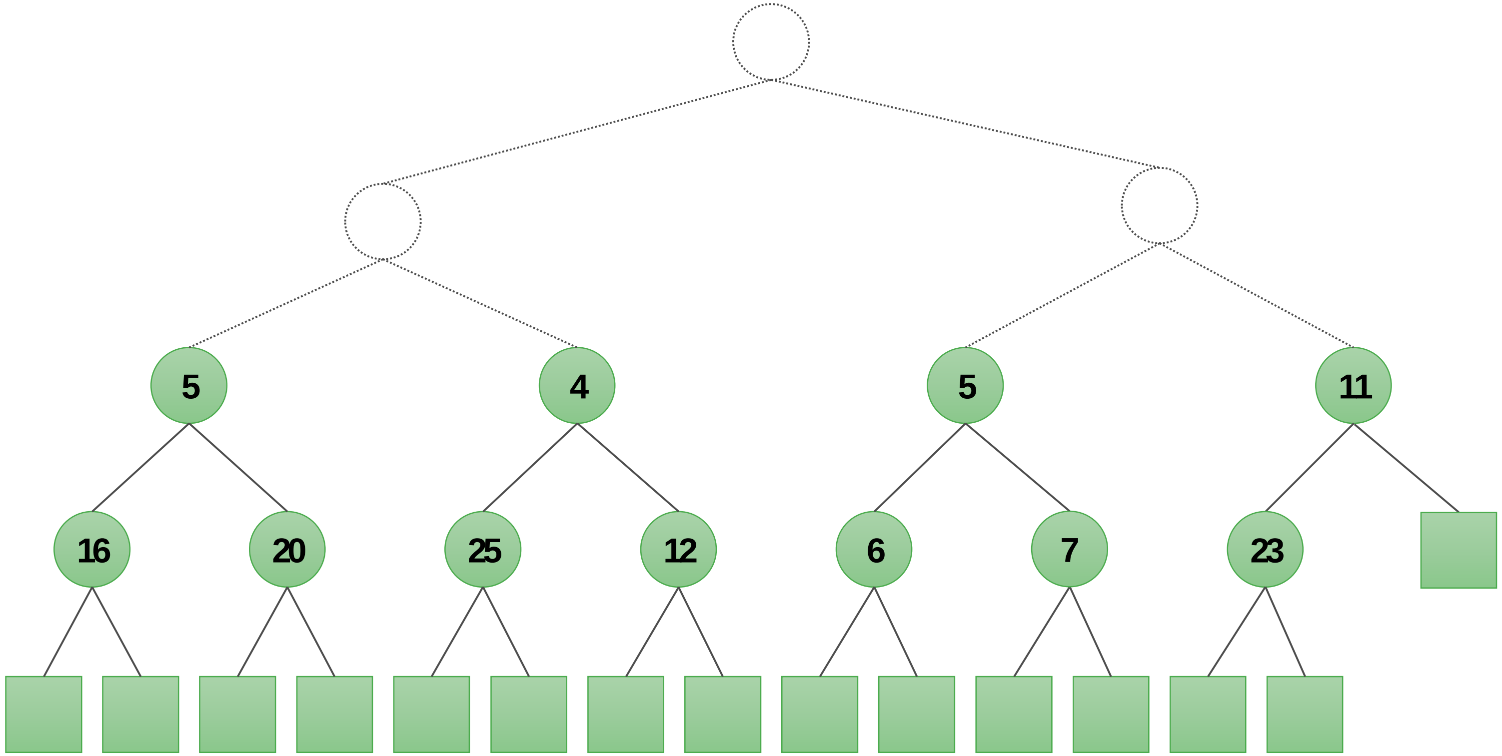
Step 3: Use downHeap to restore the order property



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

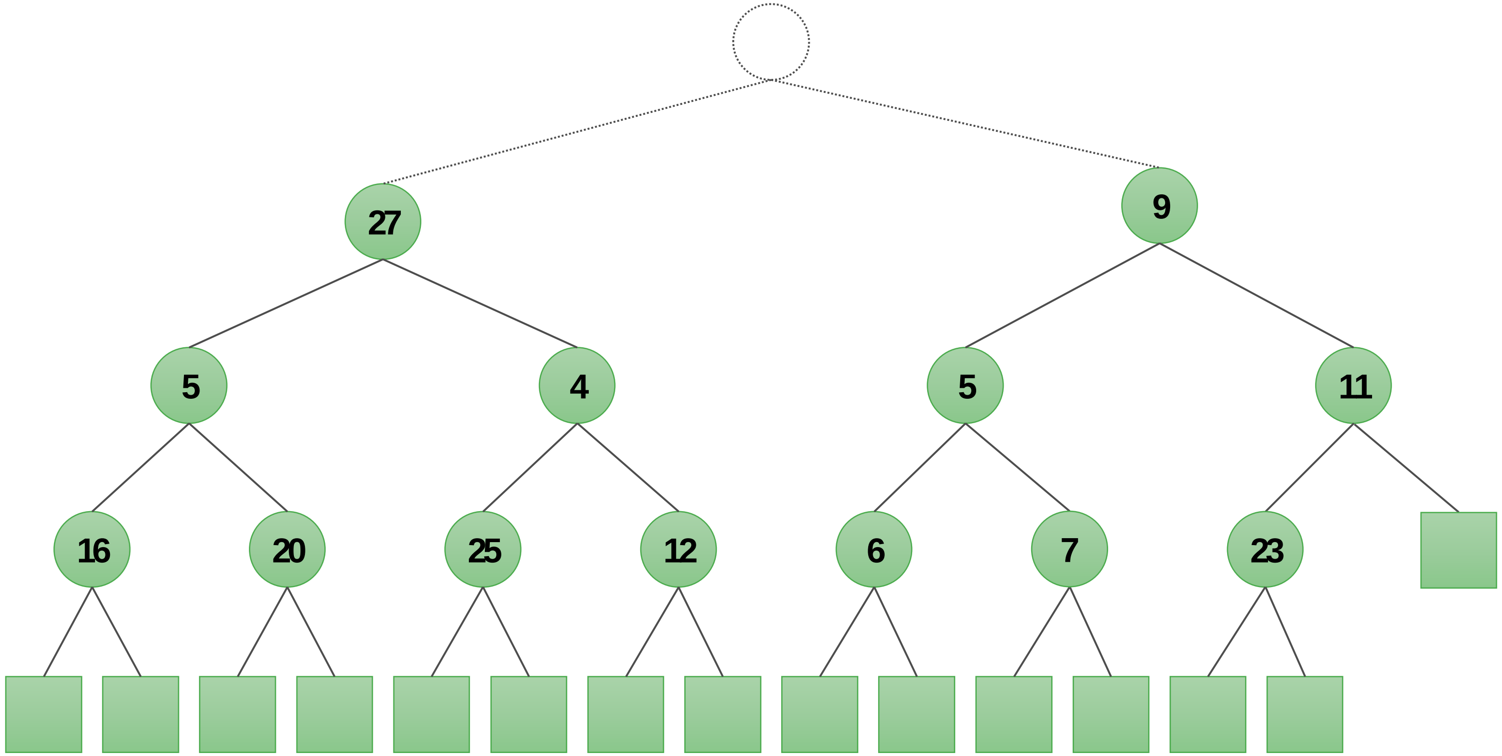
Step 3: Use downHeap to restore the order property



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

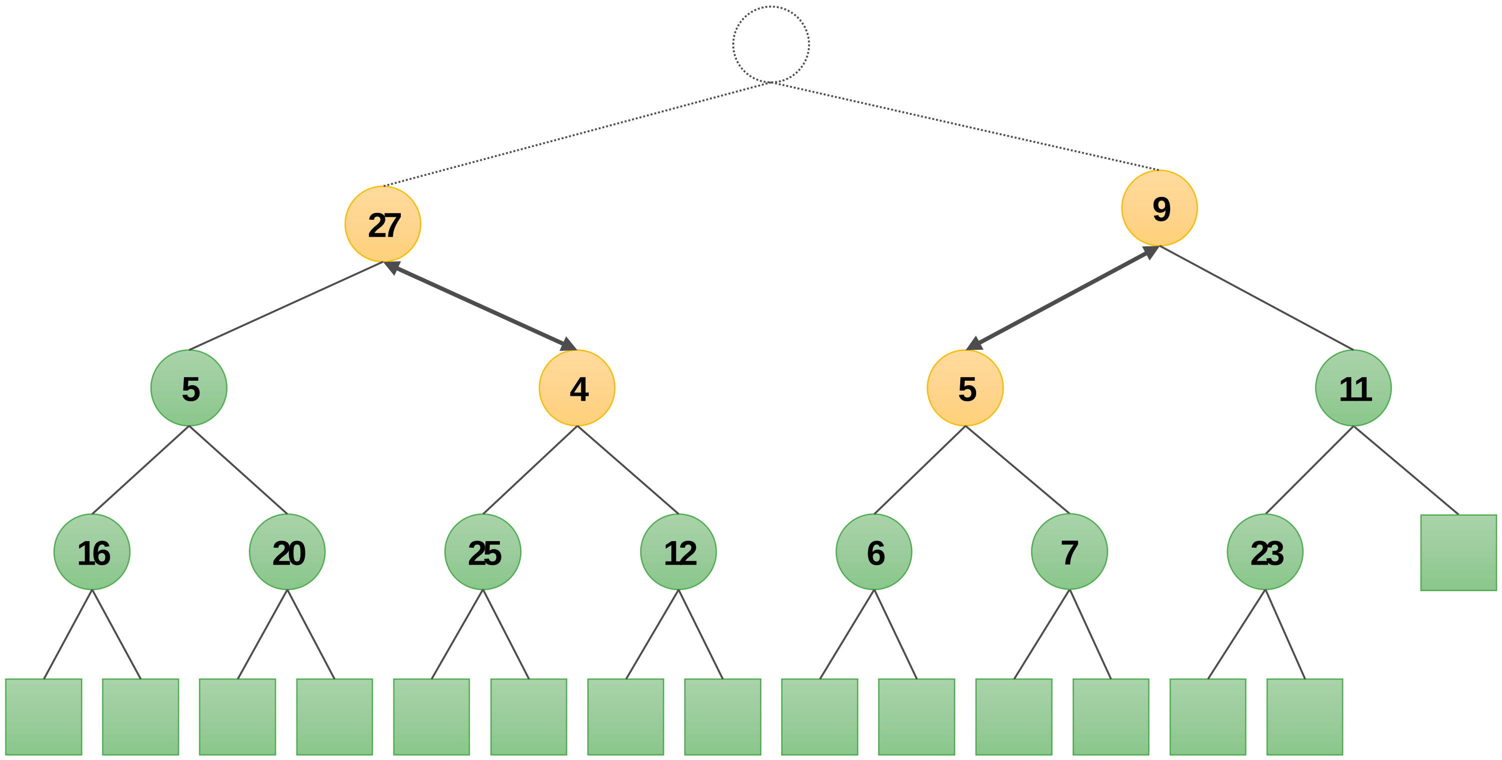
Next Step: Create 7-element heaps



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

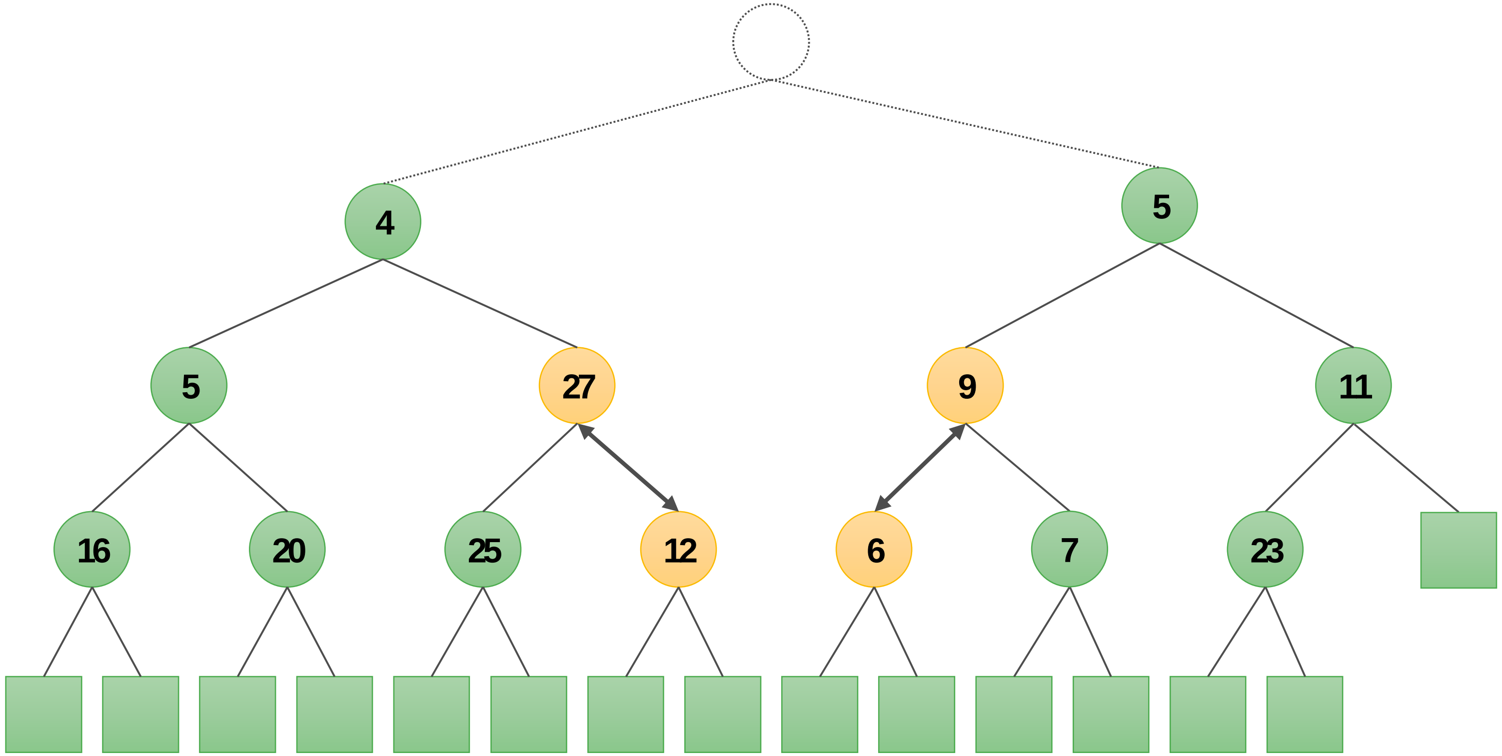
downHeap



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

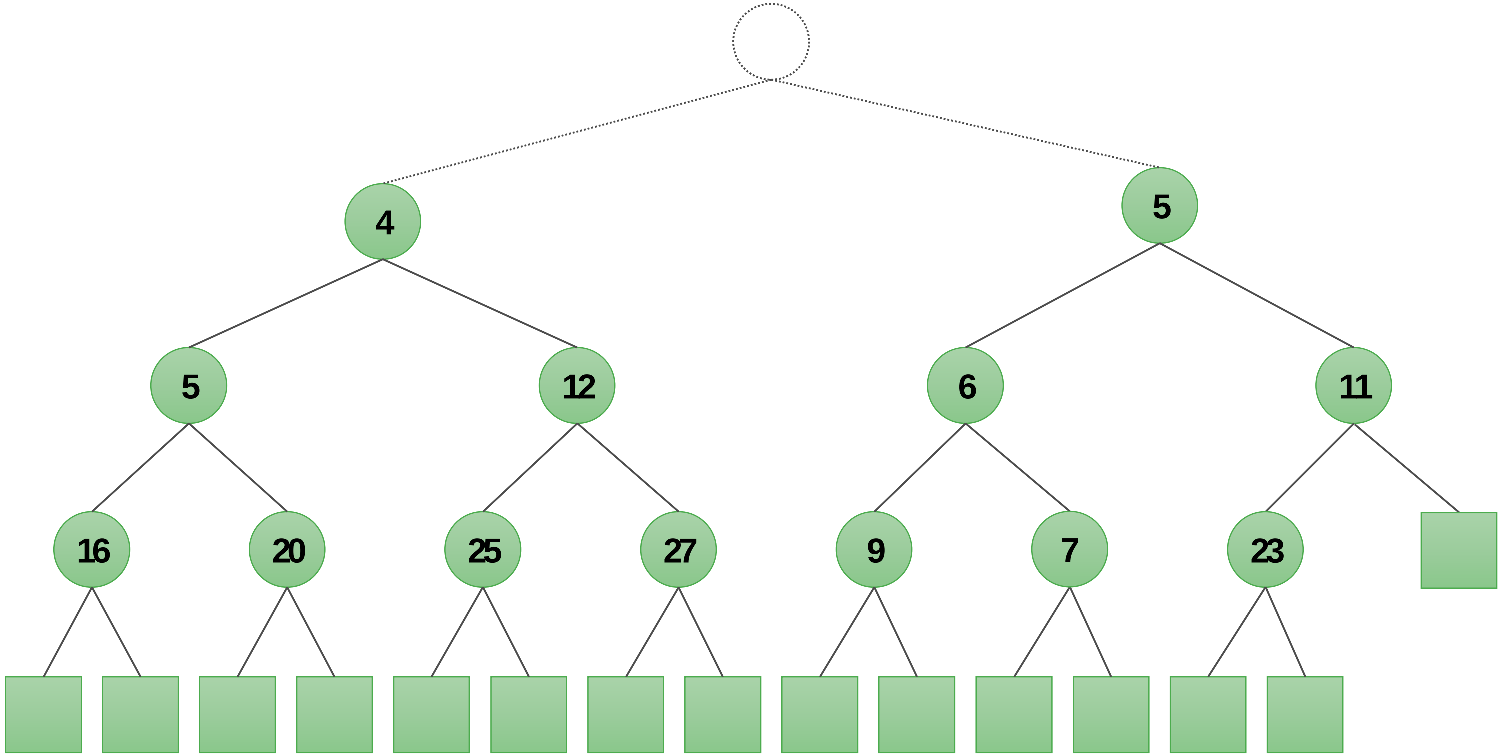
downHeap



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

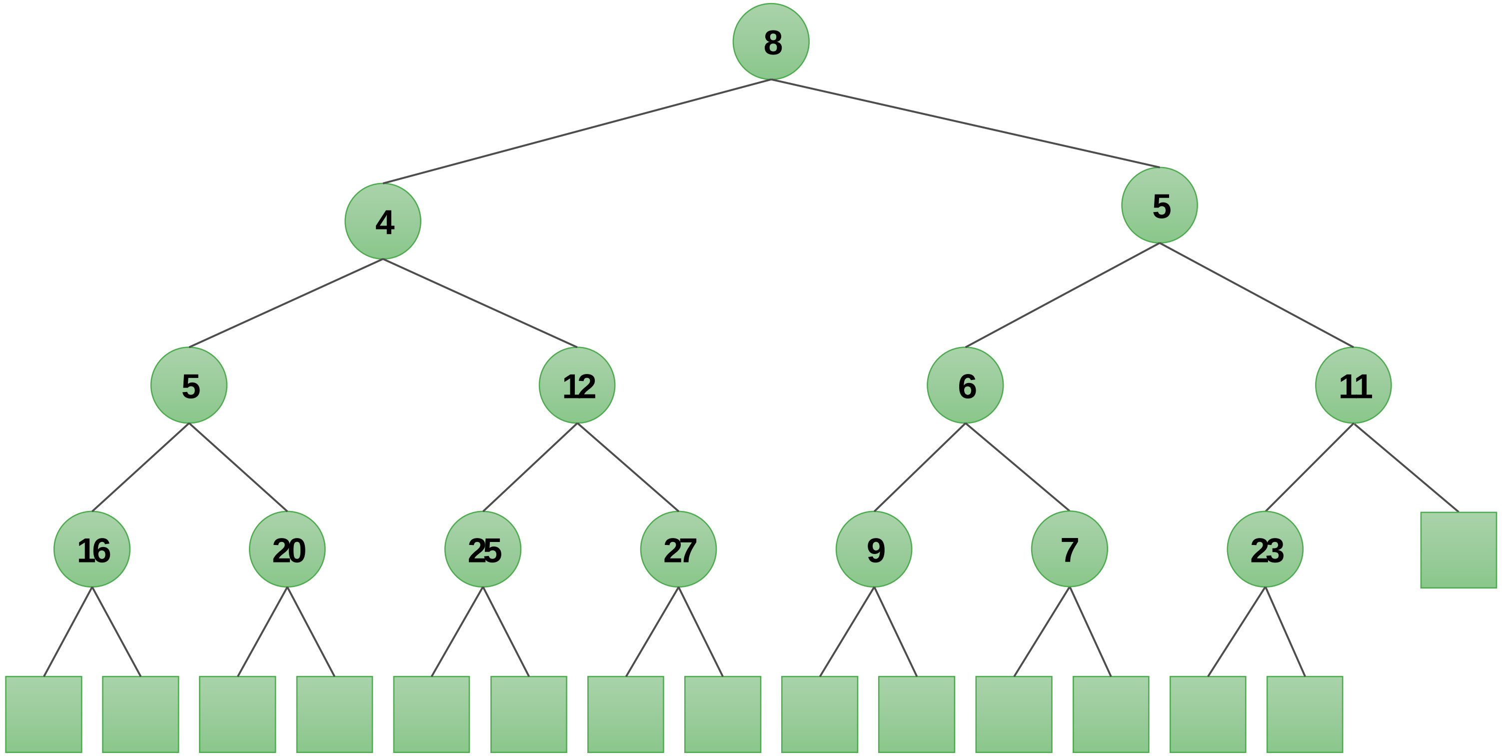
downHeap



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

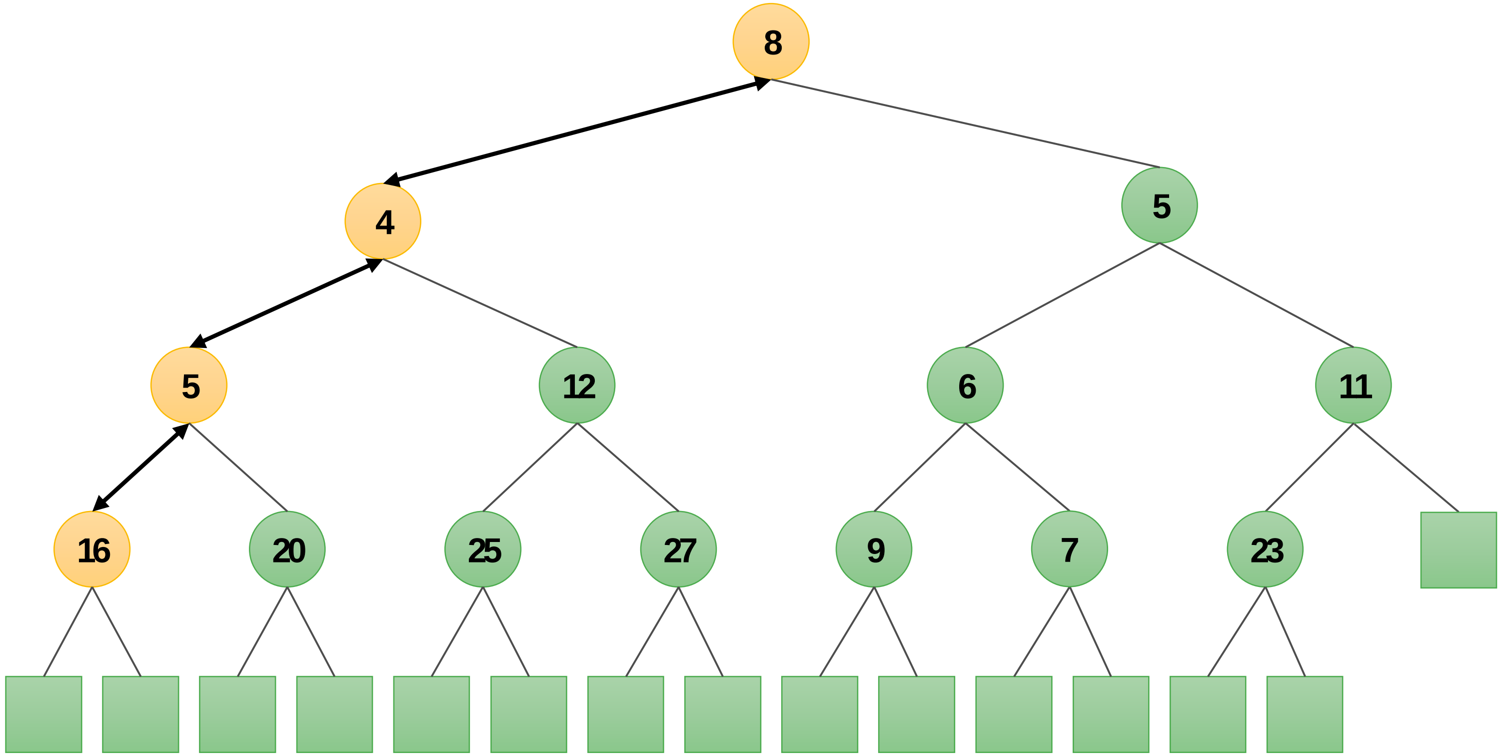
Last Step: Create n-element heap



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

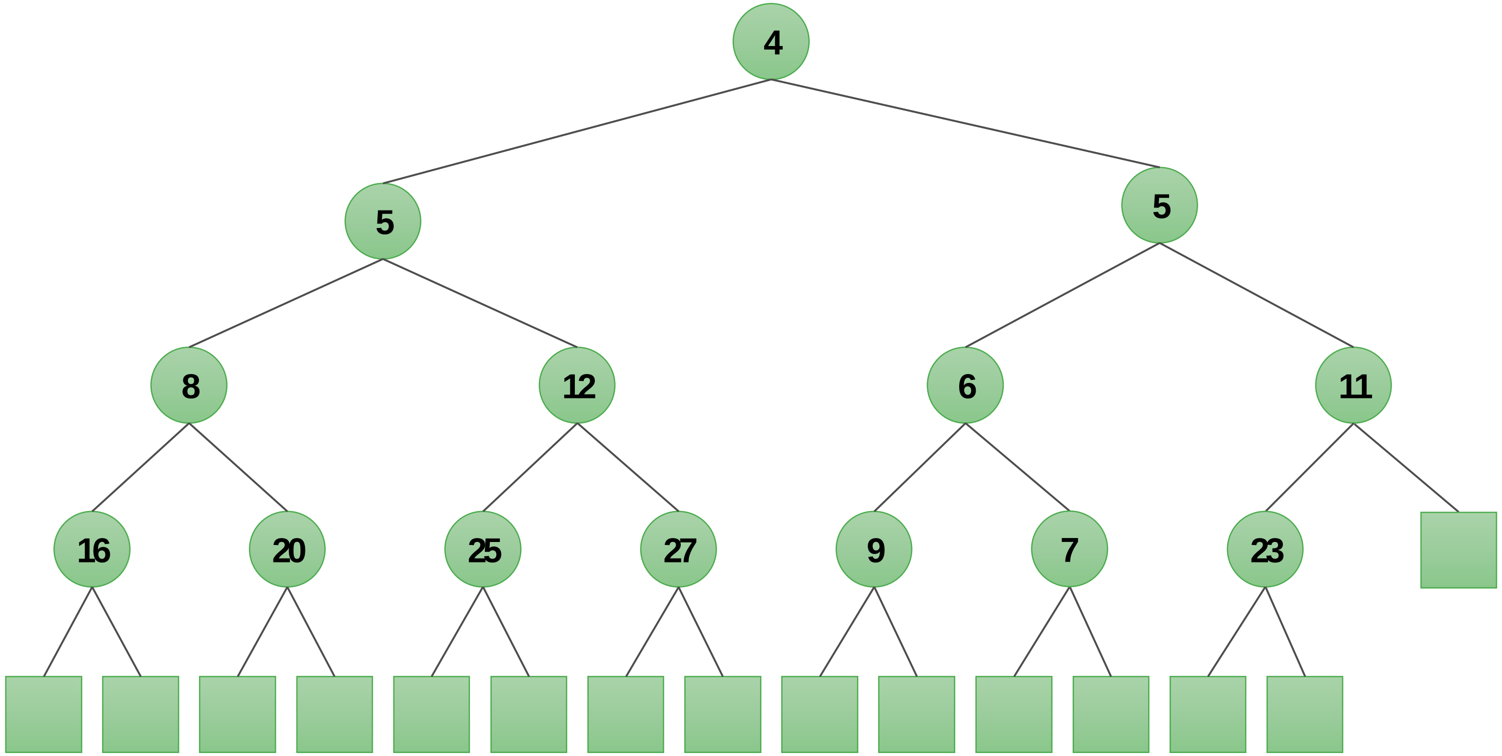
downHeap



BOTTOM-UP HEAP CONSTRUCTION

16, 5, 4, 12, 6, 7, 23, 20, 25, 5, 11, 27, 9, 8

Done, algorithm terminates!



BOTTOM-UP HEAP CONSTRUCTION

Observation

- most processed partial heaps are very small
  - in a heap with 127 elements we have to process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63 and 1 heap of size 127
  - requires 120 downHeap operations in the worst case (32*1 + 16*2 + 8*3 + 4*4 + 2*5 + 1*6)
  - general computation:

$$\sum_{i=1}^{\log n} \frac{n}{2^{i+1}} * i < \frac{n}{2} \sum_{i=1}^{\infty} \frac{1}{2^i} * i = \frac{n}{2} \sum_{i=1}^{\infty} i * \frac{1}{2}^i = \frac{n}{2} * 2 = n$$

$$\sum_{i=1}^{\infty} i * \frac{1}{2}^i = \sum_{i=1}^{\infty} \sum_{j=i}^{\infty} \frac{1}{2}^j = \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) + \left(\frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots\right) + \left(\frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \cdots\right) + \cdots = \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) = 2$$

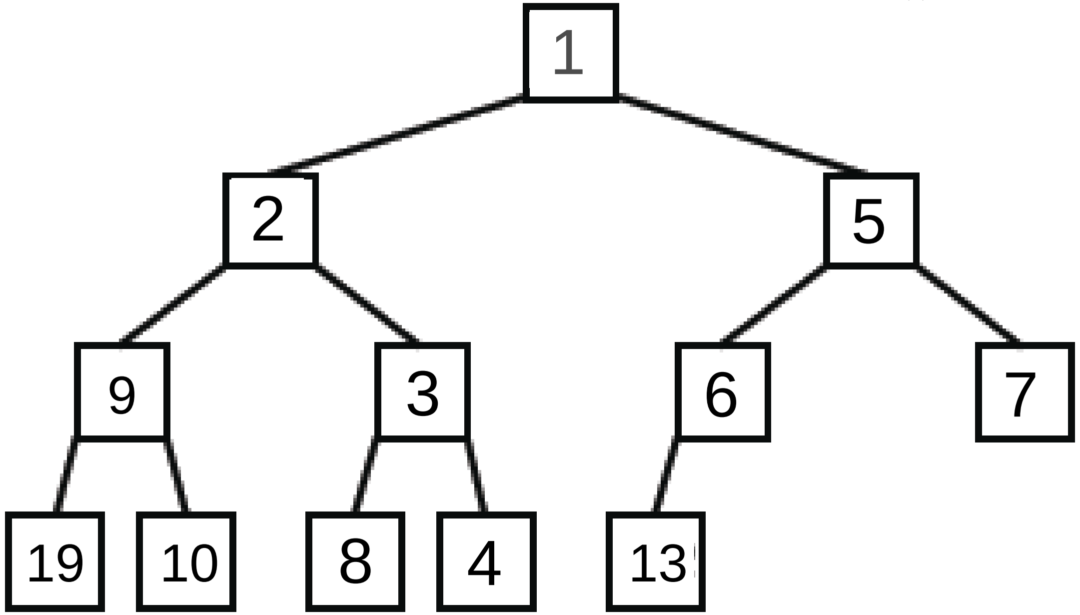
→ linear complexity O(n)

HEAPSORT

In-place sorting of heaps in **O(N log N)**

- iterative root removal places the smallest element at the position cleared by shrinking the heap

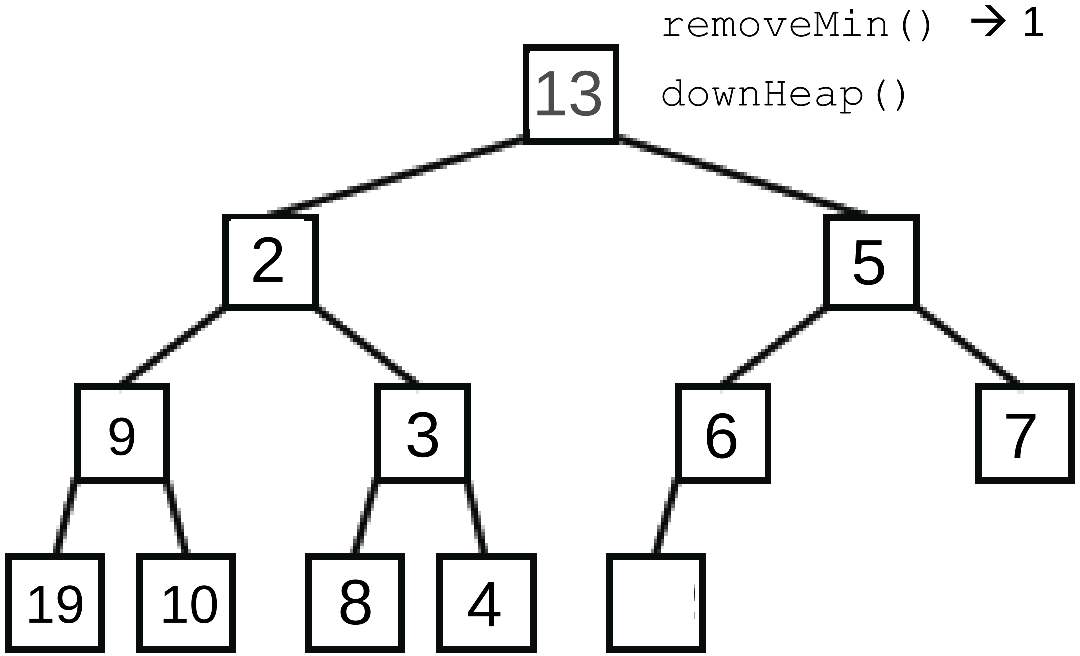
`removeMin()` → 1



HEAPSORT

In-place sorting of heaps in **O(N log N)**

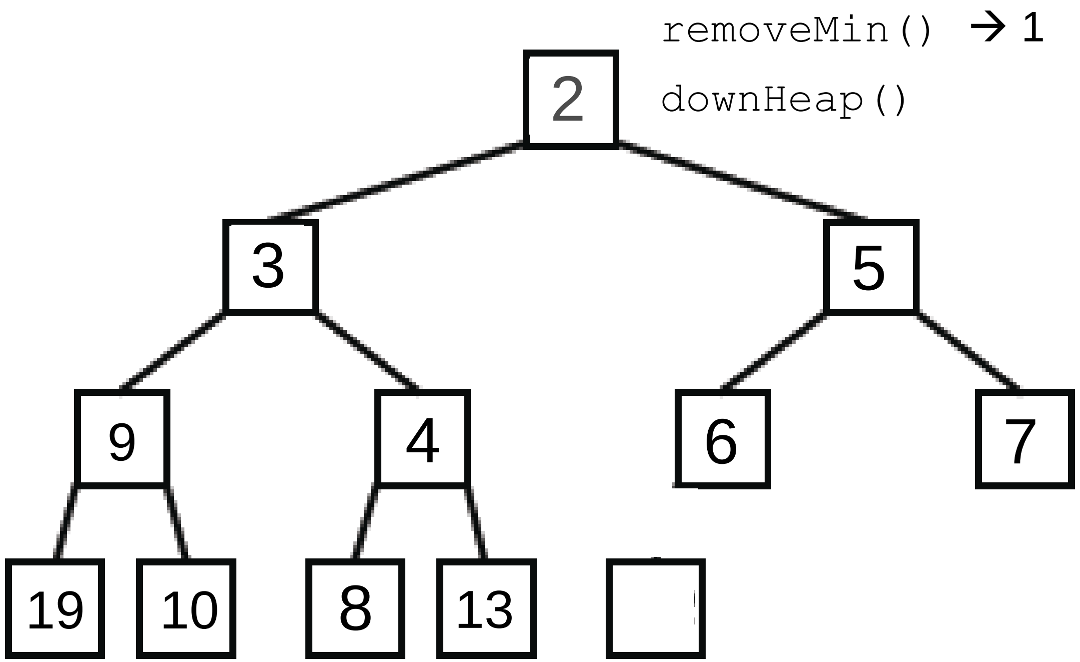
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

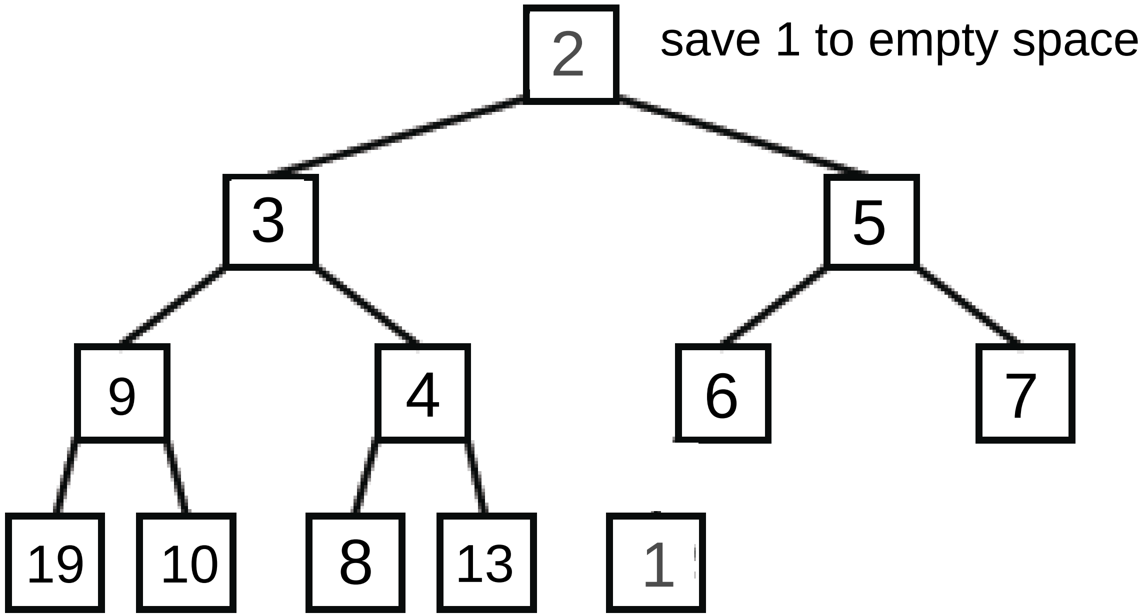
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

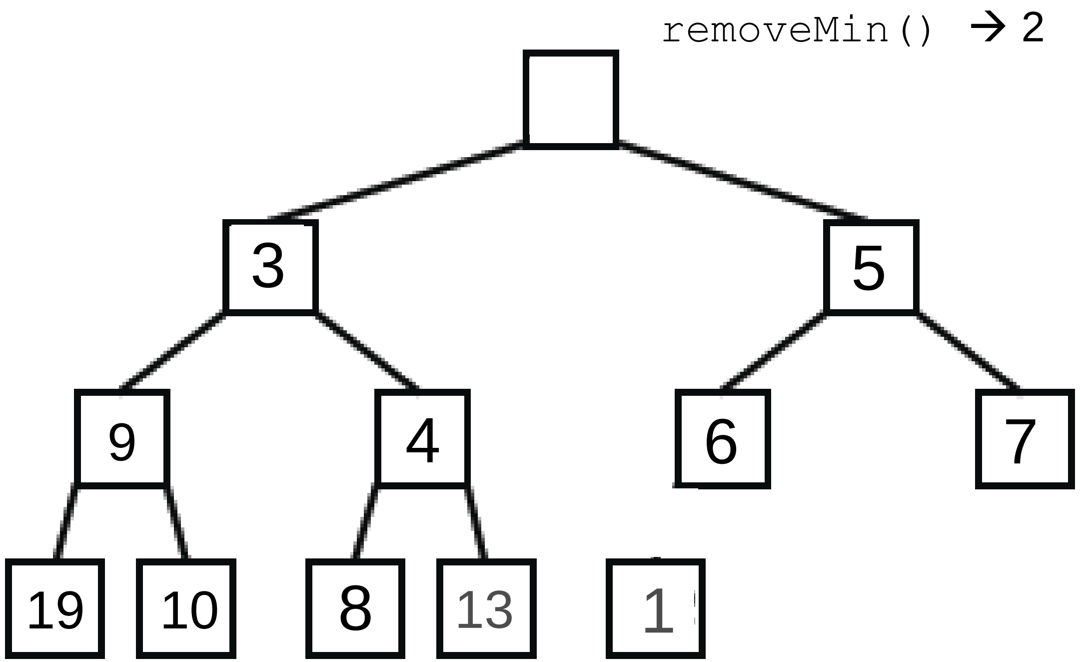
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

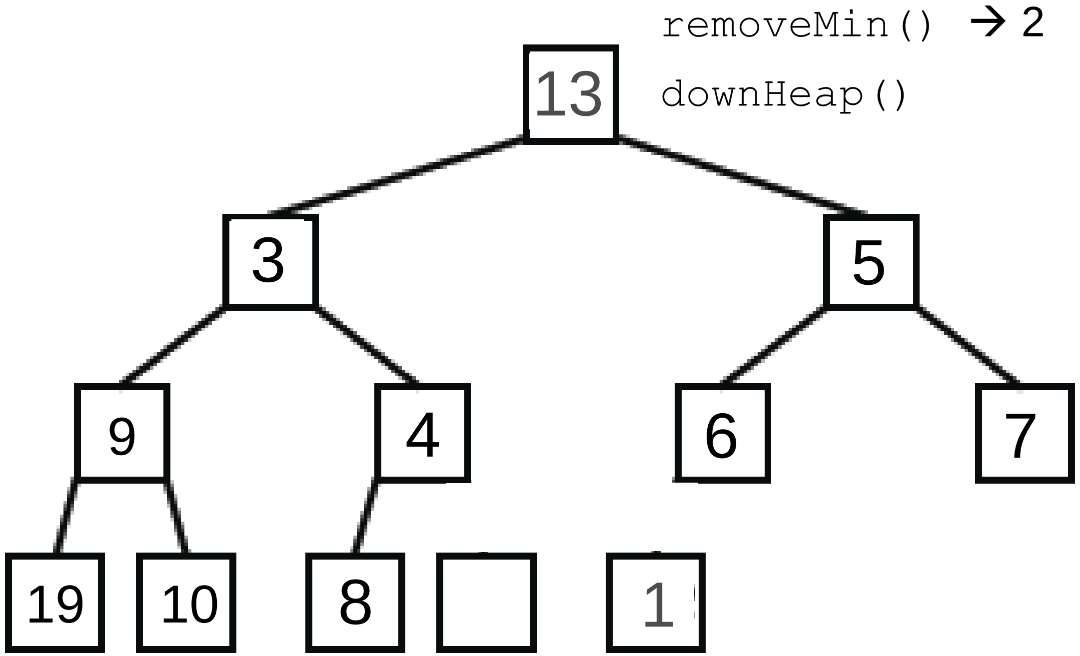
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

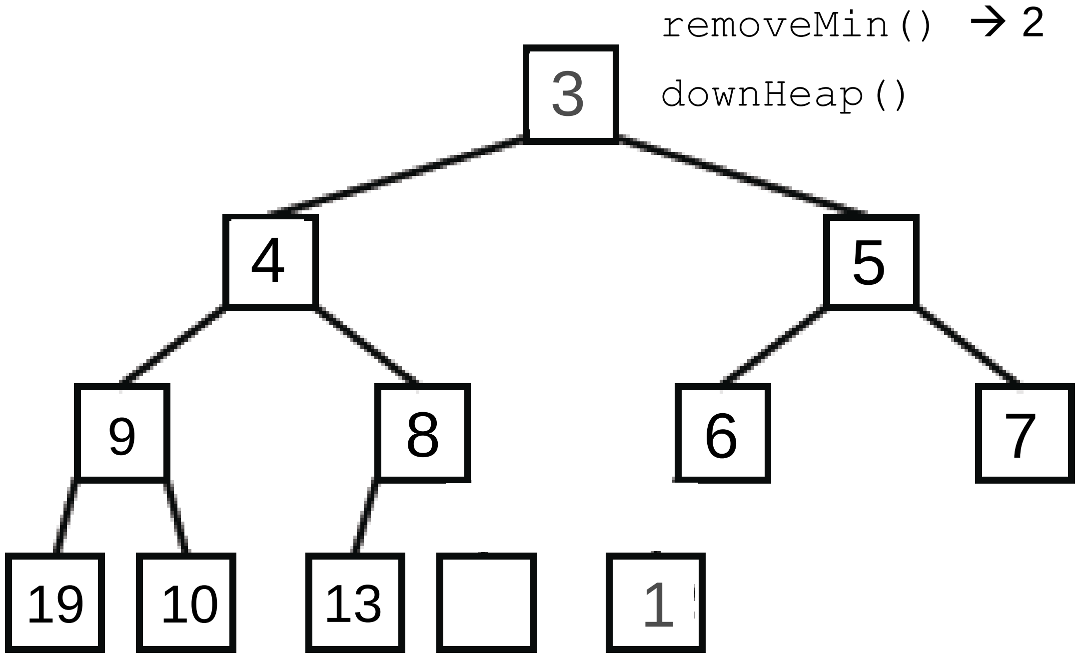
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

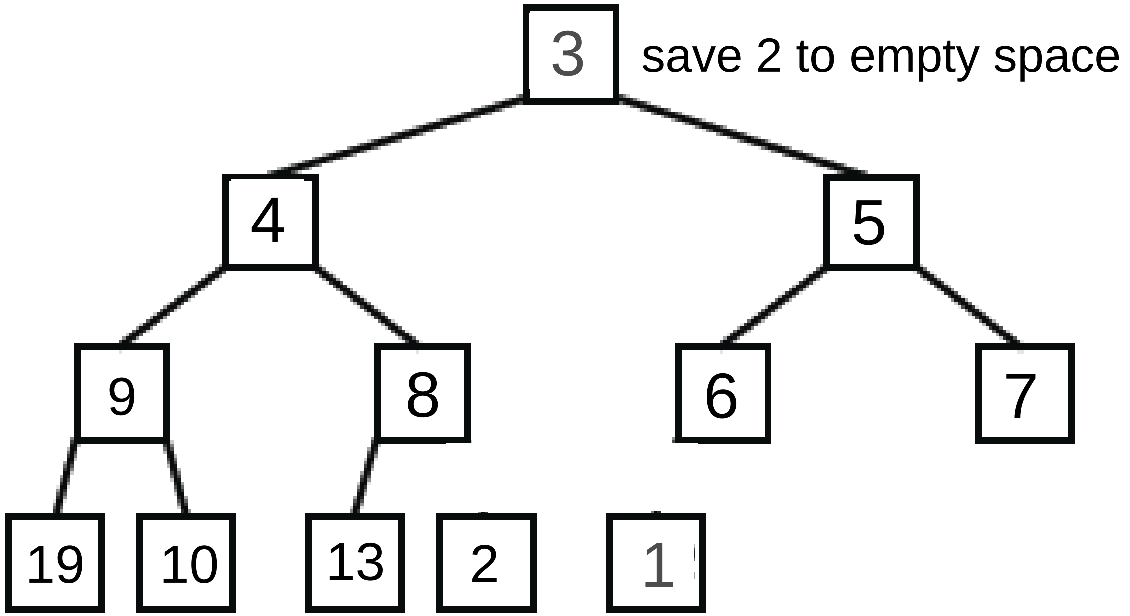
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

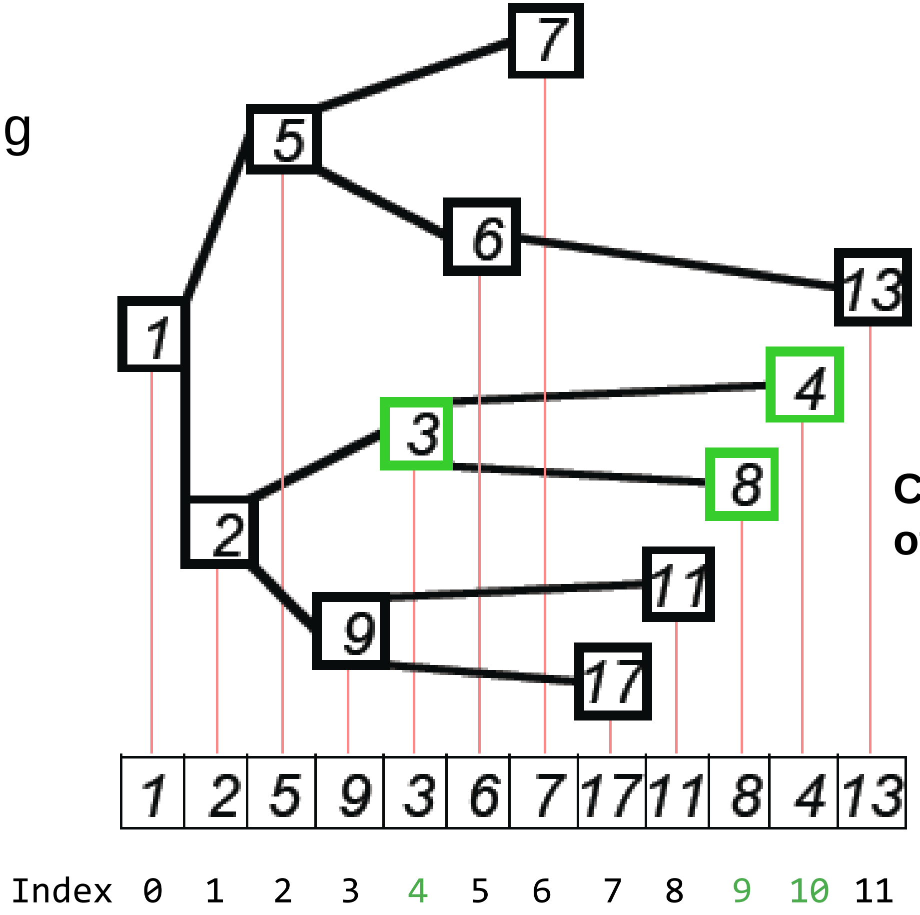
- iterative root removal places the smallest element at the position cleared by shrinking the heap



HEAPSORT

In-place sorting of heaps in **O(N log N)**

- iterative root removal places the smallest element at the position cleared by shrinking the heap



Counter for number of elements in heap!

SEARCH IN HEAPS

Search procedure

- starting at the root search the heap recursively until
  - current node is smaller than the searched node (in case of a MaxHeap) or
  - lowest level in the tree is reached

Efficiency

- make use of the heap properties
- do not search sequentially (linearly)

Complexity

- still O(n), since in the worst case all nodes have to be searched

RADIXSORT

Break keys into a sequence of fixed-size components

- binary numbers are bit sequences
- strings are characters sequences
- decimal numbers are sequences of digits

RadixSort methods - sorting methods that process numbers piece by piece

- R(adix) is referred to as base
- typically, **R=2** or a power of 2

General **principle** (w = word length)

```
for k in range(w):
  # sort the array in a stable way, looking only at the k-th digit
```

Stable sorting methods keep the relative order of elements with equal keys → **BucketSort**

RADIX-EXCHANGE SORT

Binary radix-exchange sort

- sort a[1]…a[N] based on **binary** keys
- divide array into **two** parts depending on the **leading bit**
- elements with leading 0 into the upper/left part, elements with leading 1 into lower/right part
- division by swapping **in-situ** as in QuickSort
- sort parts **recursively** alike, whereby the next bit from the left is used as leading bit

Complexity

- maximum recursion depth equals to the key length **b**
- processing (e.g., distribution) per recursion level in linear time **N**
- total: **O(b N)**

DIRECT RADIXSORT USING BUCKETS

Stable sorting method

- given **n** numbers
- for each digit **d** (of each number), **d** ∈ {1, 2, 3, ..., **m**}
- **m** = number of buckets

Algorithm

- choose number of buckets **m** (based on the data to be sorted)
- for all digits **d**
  - store number in a bucket corresponding to the digit **d** (keep relative order) - **O(n)**
  - combine numbers from all buckets into a new list (keep relative order) - **O(m)**

DIRECT RADIXSORT USING BUCKETSORT

- Sort (radix 3): 10**1**, 2**0**, 201**2**, 1**2**, 201**0**, 12**0**, 20**2**, 222**1**, **0**, 1**1**

0	20	2010	120	0
1	101	2221	11	
2	2012	12	202	

0				
1				
2				

0				
1				
2				

0				
1				
2				

DIRECT RADIXSORT USING BUCKETSORT

- Sort (radix 3): 10**1**, 2**0**, 201**2**, 1**2**, 201**0**, 12**0**, 20**2**, 222**1**, **0**, 1**1**

0	20	2010	120	0
1	101	2221	11	
2	2012	12	202	

- Merge: **2**0, 20**1**0, 1**2**0, 0, 1**0**1, 22**2**1, **1**1, 20**1**2, **1**2, 2**0**2

0	0	101	202	
1	2010	11	2012	12
2	20	120	2221	

0				
1				
2				

0				
1				
2				

DIRECT RADIXSORT USING BUCKETSORT

- Sort (radix 3): 10**1**, 2**0**, 201**2**, 1**2**, 201**0**, 12**0**, 20**2**, 222**1**, **0**, 1**1**

0	20	2010	120	0
1	101	2221	11	
2	2012	12	202	

- Merge: **2**0, 20**1**0, 1**2**0, 0, 1**0**1, 22**2**1, **1**1, 20**1**2, **1**2, 2**0**2

0	0	101	202	
1	2010	11	2012	12
2	20	120	2221	

- Merge: 0, **1**01, **2**02, 2**0**10,11, 2**0**12, 12, 20, **1**20, 2**2**21

0	0	2010	11	2012	12	20
1	101	120				
2	202	2221				

0						
1						
2						

DIRECT RADIXSORT USING BUCKETSORT

- Sort (radix 3): 10**1**, 2**0**, 201**2**, 1**2**, 201**0**, 12**0**, 20**2**, 222**1**, **0**, 1**1**

0	20	2010	120	0
1	101	2221	11	
2	2012	12	202	

- Merge: **2**0, 20**1**0, 1**2**0, 0, 1**0**1, 22**2**1, **1**1, 20**1**2, **1**2, 2**0**2

0	0	101	202	
1	2010	11	2012	12
2	20	120	2221	

- Merge: 0, **1**01, **2**02, 2**0**10,11, 2**0**12, 12, 20, **1**20, 2**2**21

0	0	2010	11	2012	12	20
1	101	120				
2	202	2221				

- Merge: 0, **2**010, 11, **2**012, 12, 20, 101, 120, 202, **2**221

0	0	11	12	20	101	120	202
1							
2	2010	2012	2221				

Result: 0,11, 12, 20, 101, 120, 202, 2010, 2012, 2221

SORTING

Algorithms and Data Structures 1
Exercise – 2023S

Markus Jäger (Computer Science)
Florian Beck (Artificial Intelligence)
Bernhard Anzengruber (Artificial Intelligence)

Institute of Pervasive Computing
Johannes Kepler University Linz

markus.jaeger@jku.at
florian.beck@jku.at
bernhard.anzengruber-tanase@jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at