Heap Priority Queue and Heapsort

CS 4231, Fall 2020 Mihalis Yannakakis

Priority Queue

- Max-Priority Queue: Data structure for a set S of items, each with a key (its "priority")
- Basic Operations:
  – Insert: insert item x ($S := S \cup \{x\}$)
  – Max: returns an item with maximum key
  – Extract-Max: returns and deletes a max-key item from S
- Other operations: Increase key, Delete
- Min-Priority Queue

Many Applications

- Scheduling jobs in computer systems
- Event-driven simulation: priority = event times
- Graph algorithms: shortest paths, min spanning tree …
- Data compression: Huffman code
- Artificial intelligence: A* search
- ….

Sorting with a Priority Queue

- Sorting A[1..n] with a Min-Priority Queue S

  S = $\varnothing$

  for i=1 **to** n **do** Insert(S,A[i])

  for i=1 **to** n **do** A[i] = Extract-Min(S)

- Sorting with a Max-Priority Queue

  S = $\varnothing$

  for i=1 **to** n **do** Insert(S,A[i])

  for i=n **down to** 1 **do** A[i] = Extract-Max(S)

Simple Approaches

	Insert	Extract-Max
• Unsorted List	$\Theta(1)$	$\Theta(n)$
• Sorted List	$\Theta(n)$	$\Theta(1)$
Would like	$O(\log n)$	$O(\log n)$

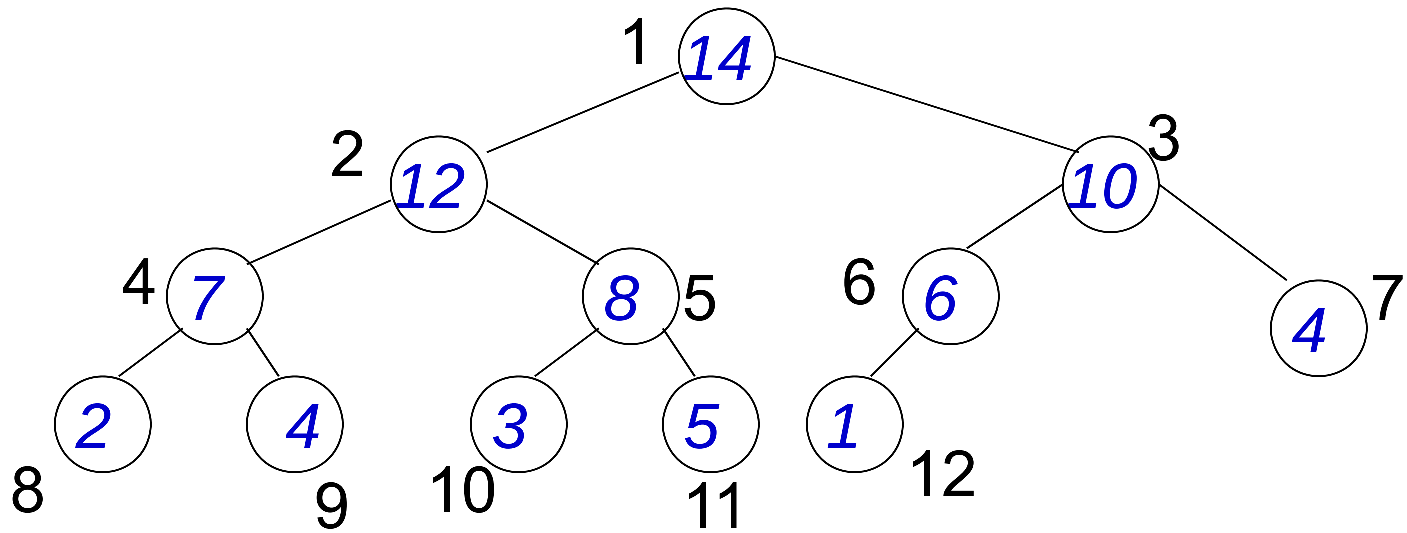
Heap

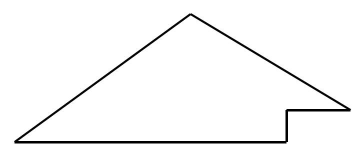
- Binary tree, implemented via an array

Two properties:

- Shape property
- Order property

Shape property

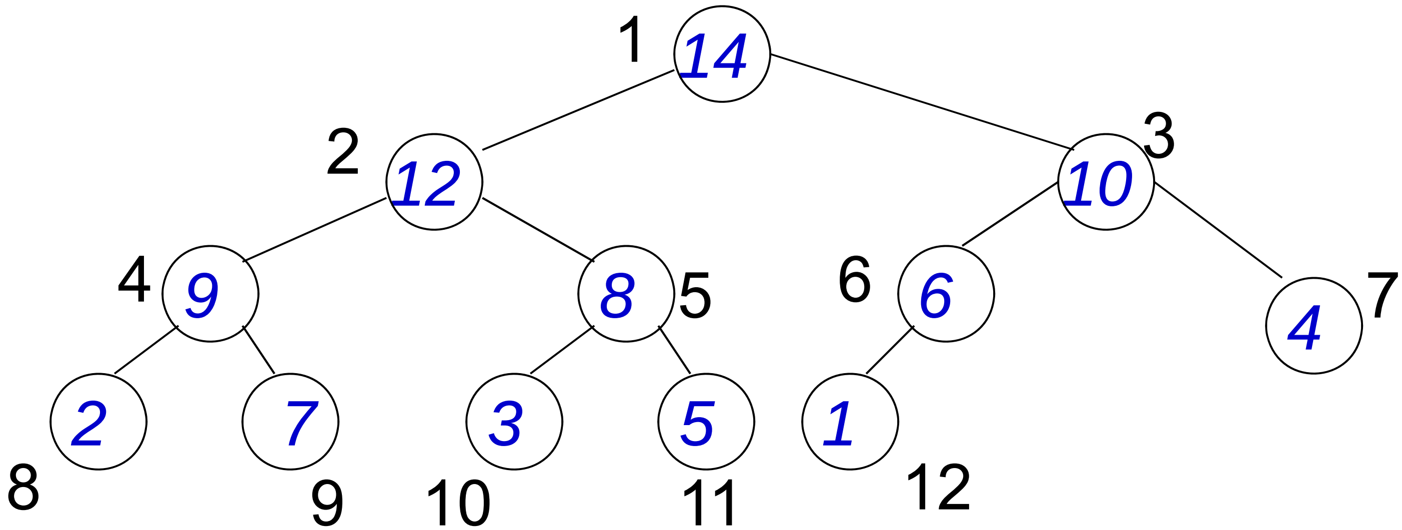


Nearly complete binary tree: 

All levels full, except possibly last level, which is filled partially from left

$\#\text{levels} = \lceil \log(n+1) \rceil$

Shape property ⇒ Array representation



$\text{parent}(i)=\lfloor i/2 \rfloor$ $\text{children}(i)=\{2i, 2i+1\}$

14	*12*	*10*	*9*	*8*	*6*	*4*	*2*	*7*	*3*	*5*	*1*
1	2	3	4	5	6	7	8	9	10	11	12

Order Property

- Max-Heap: key(i) ≤ key(parent(i))
- Hence, keys of ancestors at least as great
- Hence, maximum key at the root

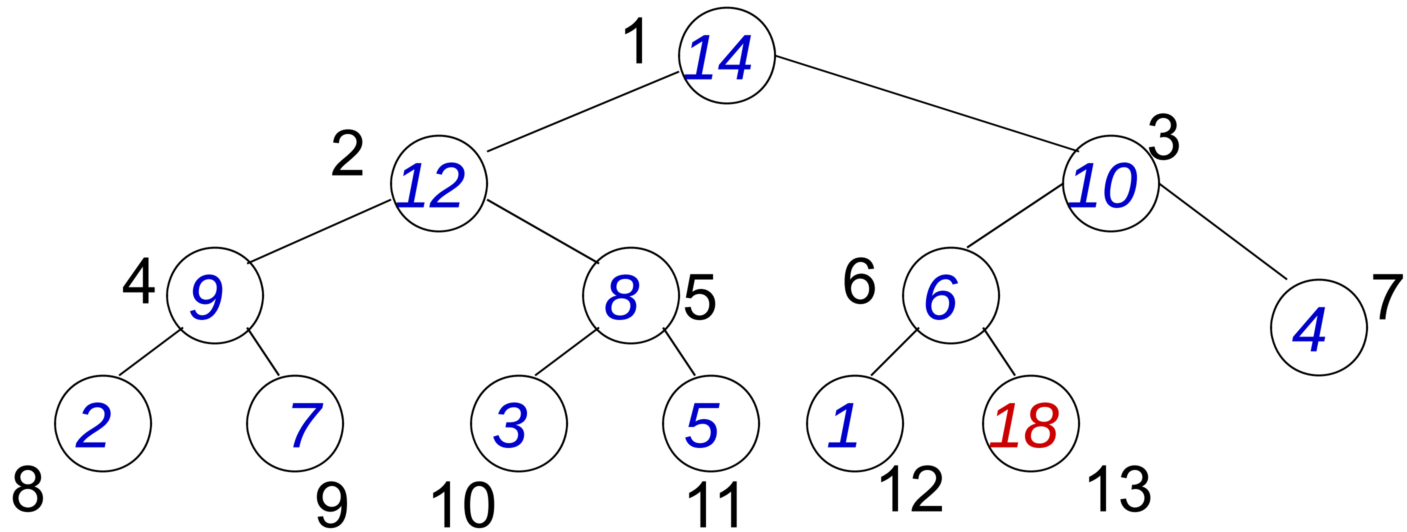
Symmetrically:

- Min-Heap: key(i) ≥ key(parent(i))
- Hence, minimum key at the root

Restrict to max-heaps from now on;
min-heaps symmetric

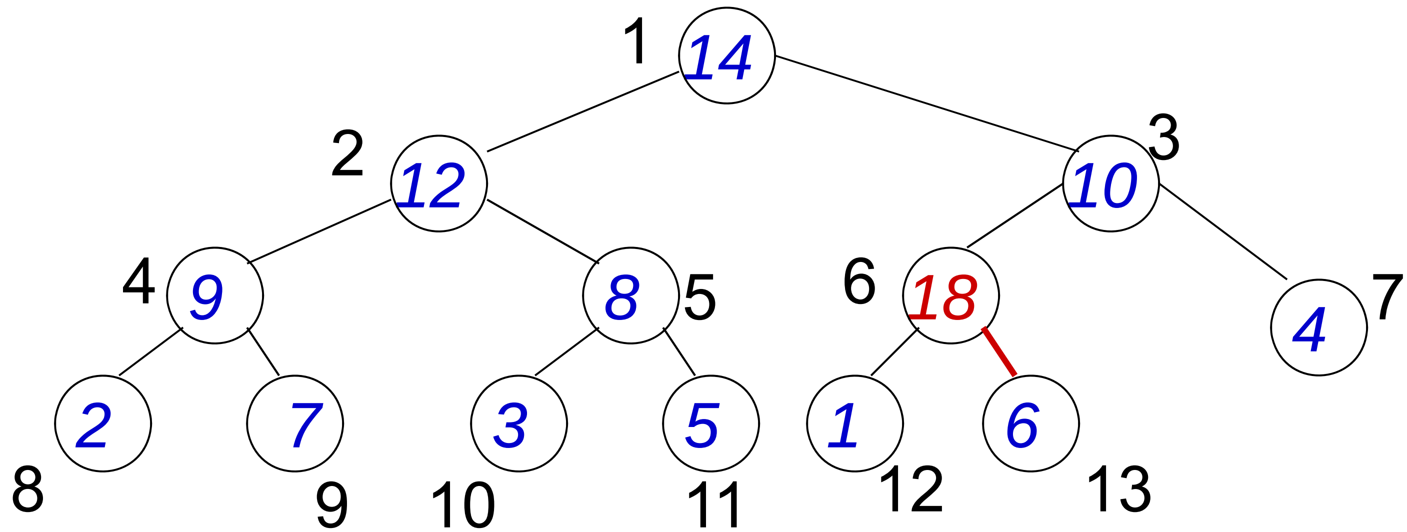
Insert

Add new leaf n+1 with new element

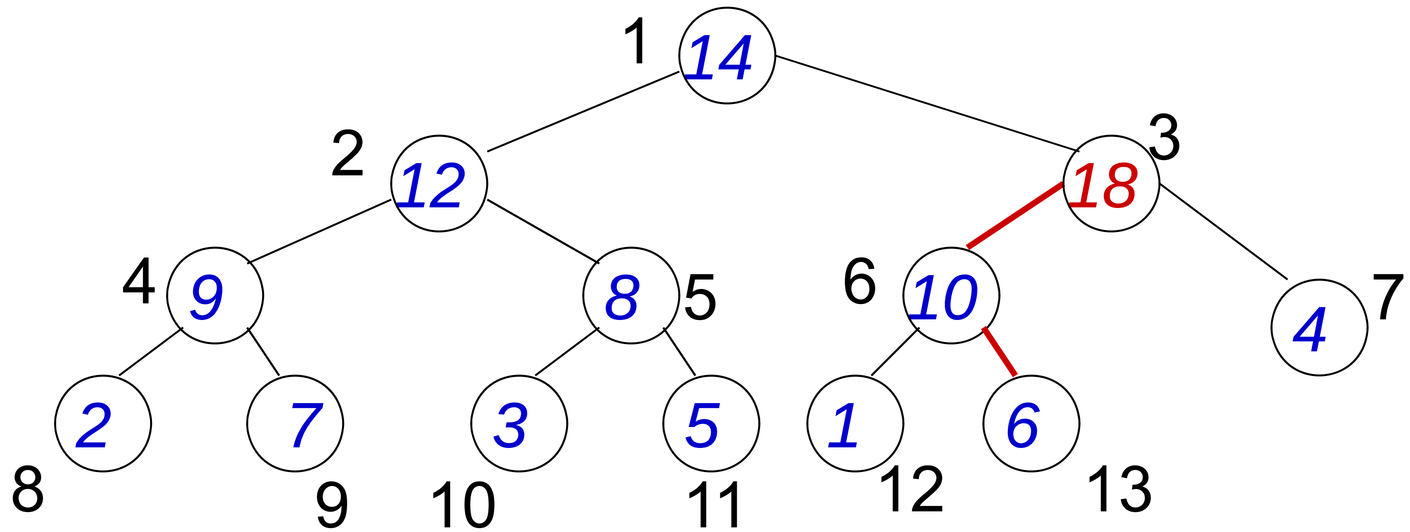


- If key(n+1) > key(parent(n+1)), then move new element up the tree exchanging with parent, till it satisfies the order property

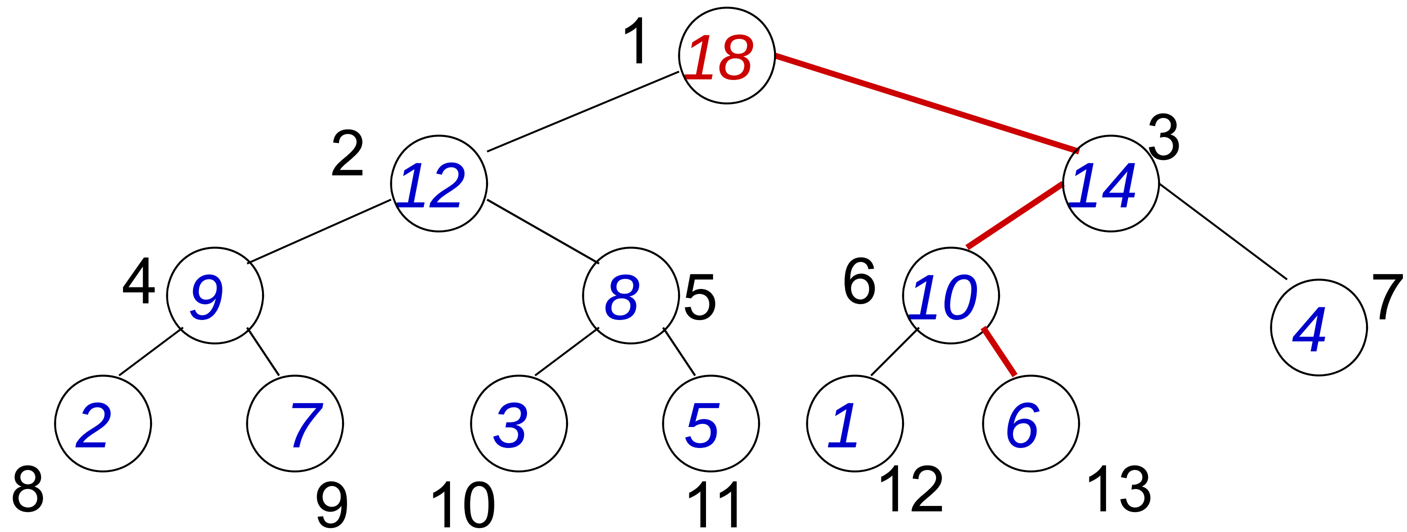
Moving up the new key



Moving up the new key



Moving up the new key



- Order property still holds at all other nodes
- Complexity = O(logn)

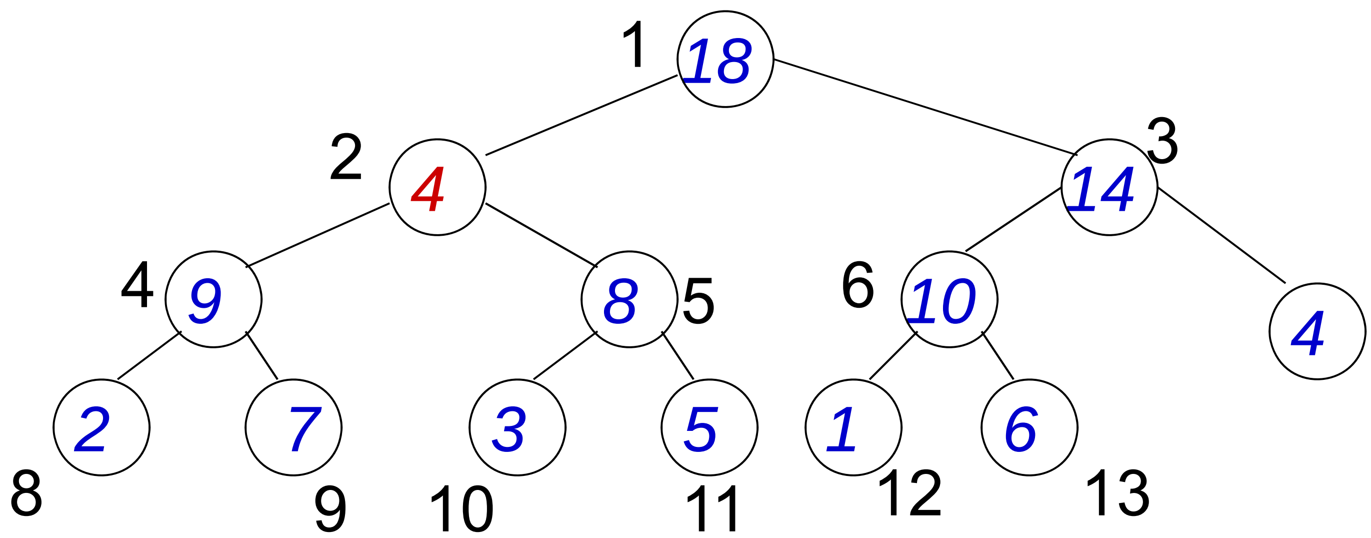
Insert (A,x)

- Input: Array representation A of heap, new key x to be inserted

```
heap-size(A)=heap-size(A)+1
i=heap-size(A)
A[i]=x
while i >1 and A[i] > A[⌊i/2⌋]
   {  exchange A[i] and A[⌊i/2⌋]
      i = ⌊i/2⌋
   }
```

Fixing a violation of the order property

- Suppose change the value of a key in a heap
- If increase key(i) then can fix the violation by moving key up exchanging with parent
- If decrease key(i) then can fix the violation by moving key(i) down the tree, exchanging with the child with maximum key: HEAPIFY(A,i)



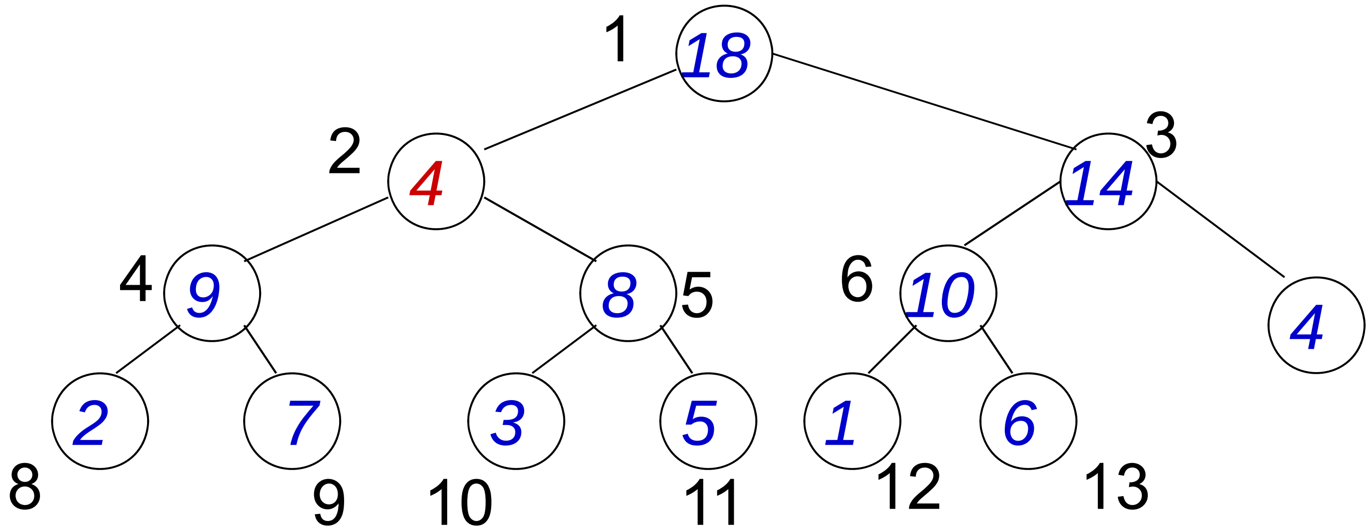
HEAPIFY(A,i)

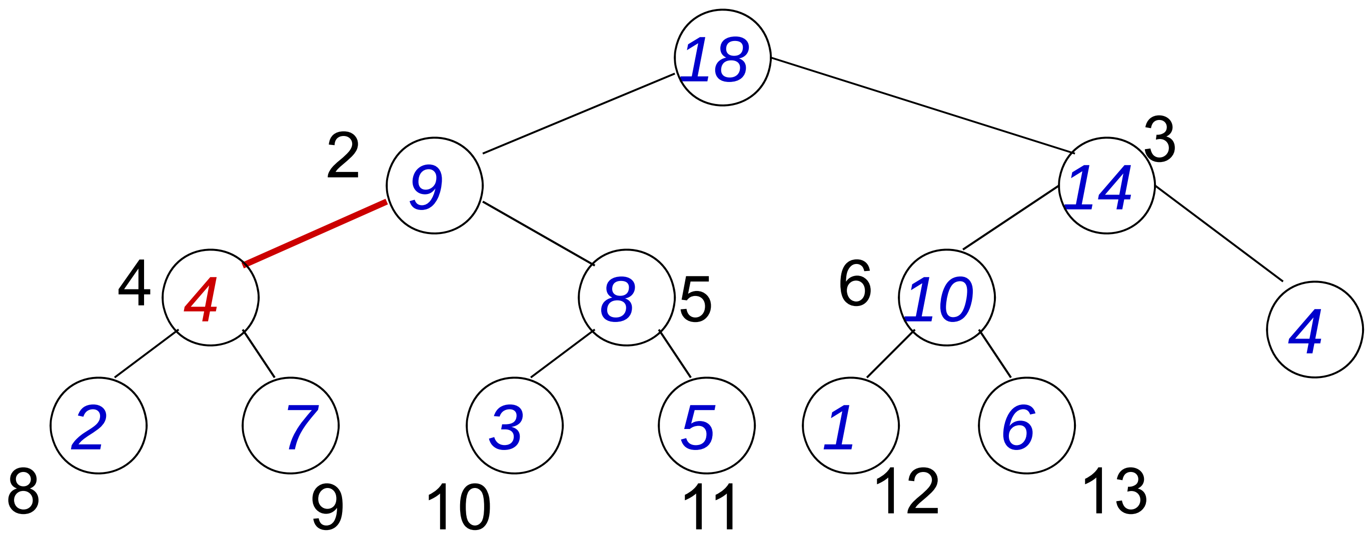
- Fixes a possible violation of order property between node i and a child
- If key(i) < key(child(i)), then move key(i) down the tree exchanging with child with maximum key
- HEAPIFY(A,i)

```
HEAPIFY(A,i)
if i is not a leaf then
    {  j = child of i with maximum key
       if A[i] < A[j] then { exchange A[i] and A[j]
                              HEAPIFY(A,j)  }
    }
```

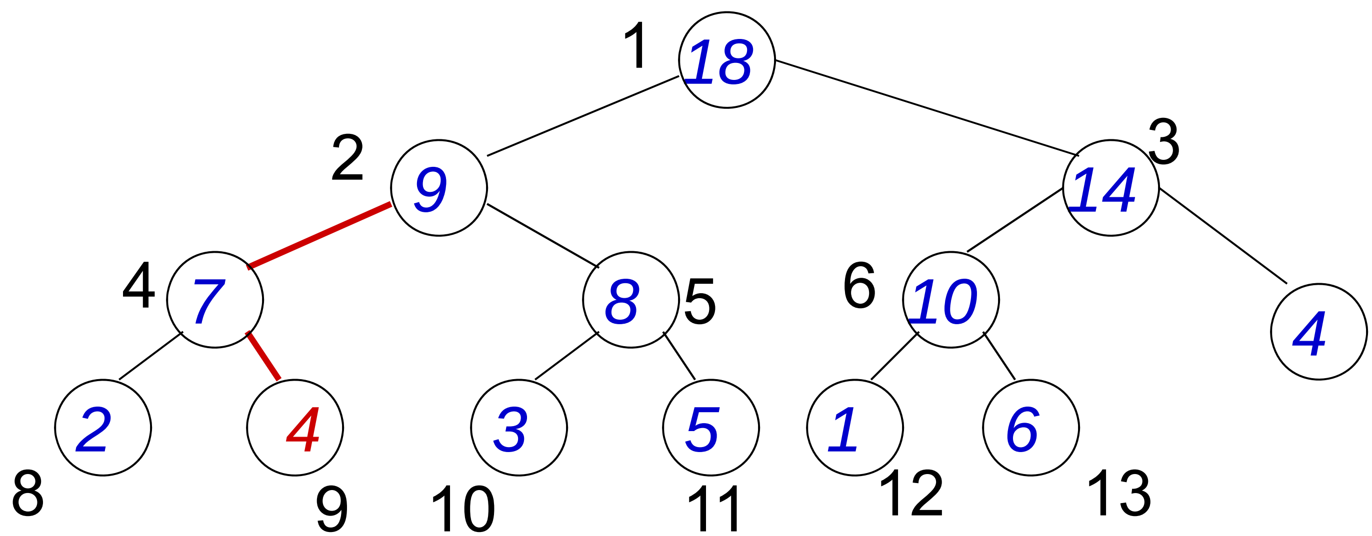
- Complexity O(height(i))

HEAPIFY(A,i) : Moving down a key





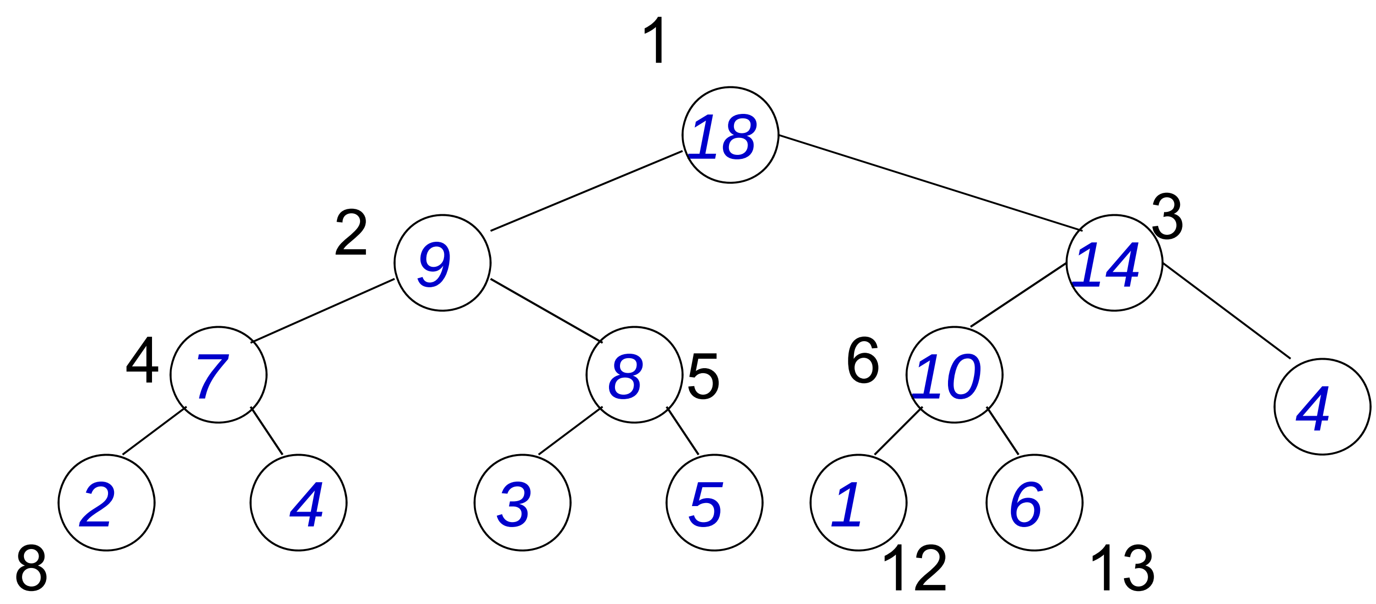
HEAPIFY(A,i) : Moving down a key



- For every other node j≠i, if it satisfied the order property key(j)≤key(parent(j)) before, then it still satisfies it.

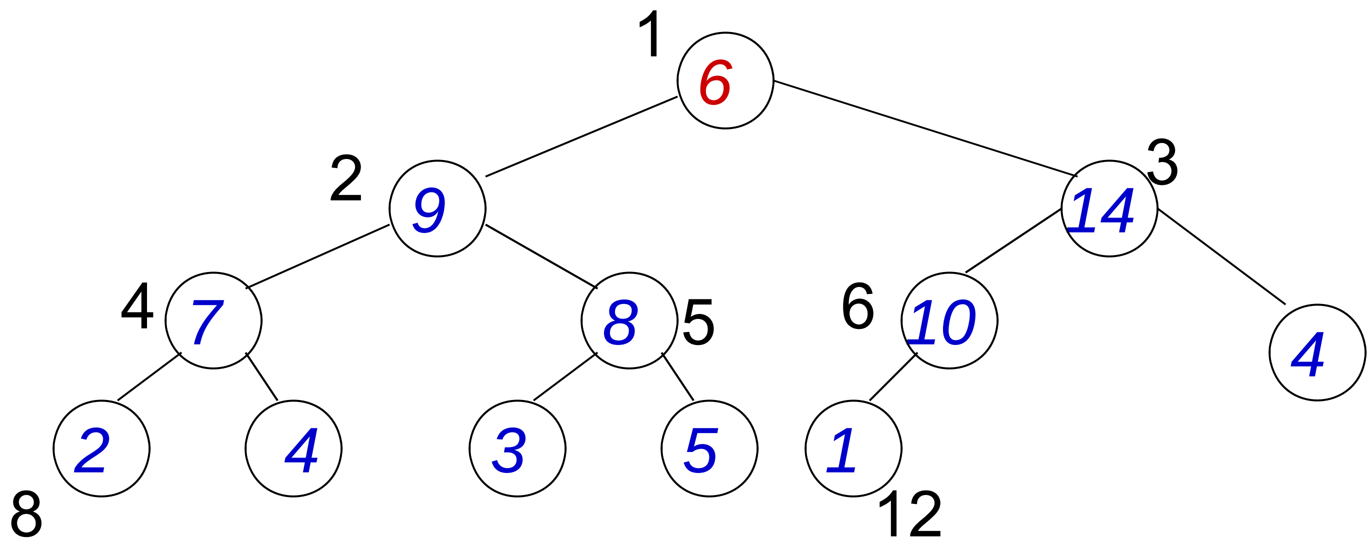
- Complexity O(height(i))

Extract-Max



- Max key at root = 18
- Delete last leaf to satisfy shape property, place its key at root

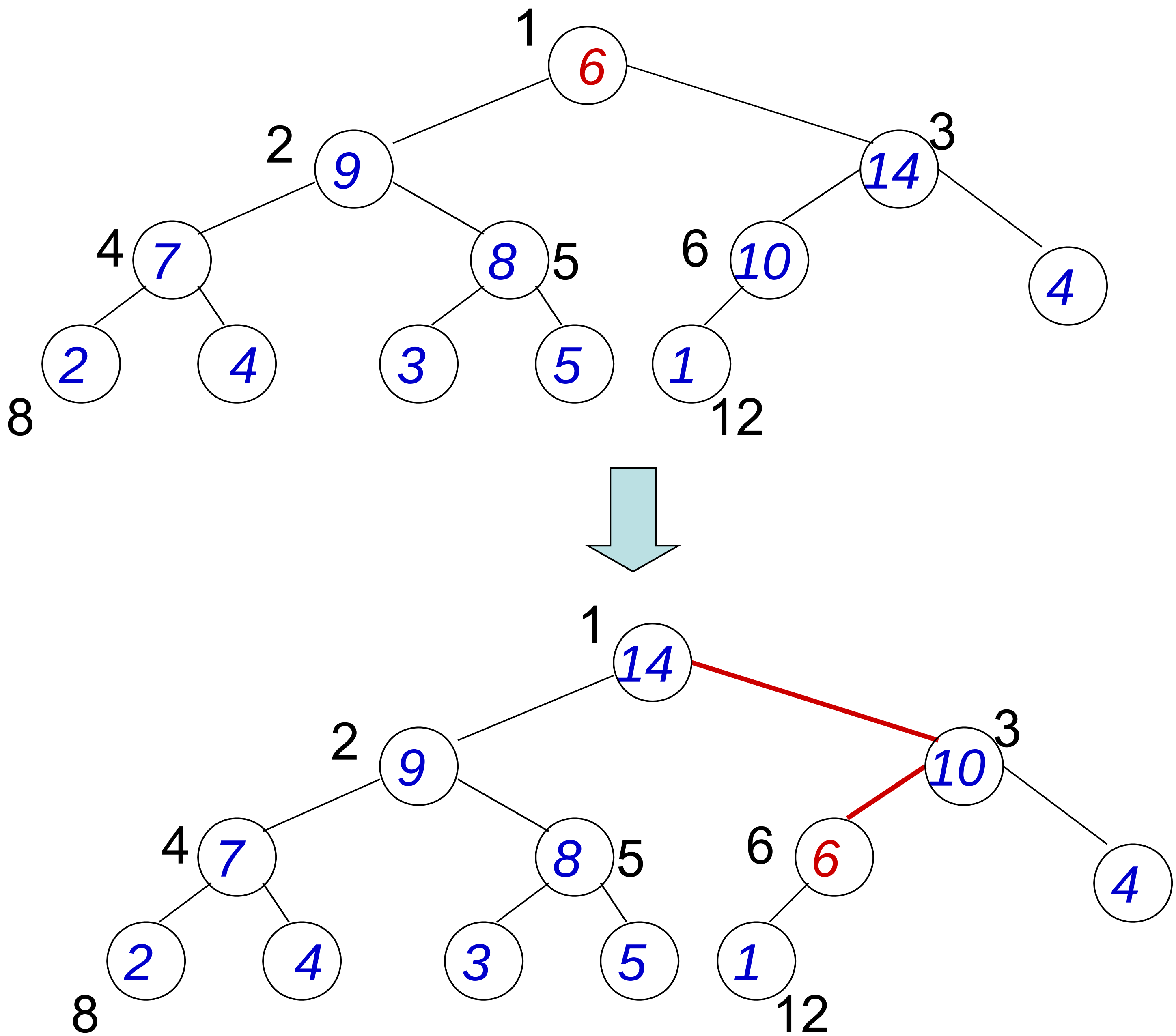
Extract-Max



- All nodes satisfy the order property except the root

- HEAPIFY(A,1) will restore the order property of heap at all nodes

- Complexity : O(logn)



1
6
2
9
3
14
4
7
8
5
6
10
4
2
4
3
5
1
8
12
1
14
2
9
3
10
4
7
8
5
6
6
4
2
4
3
5
1
8
12

Building a heap initially

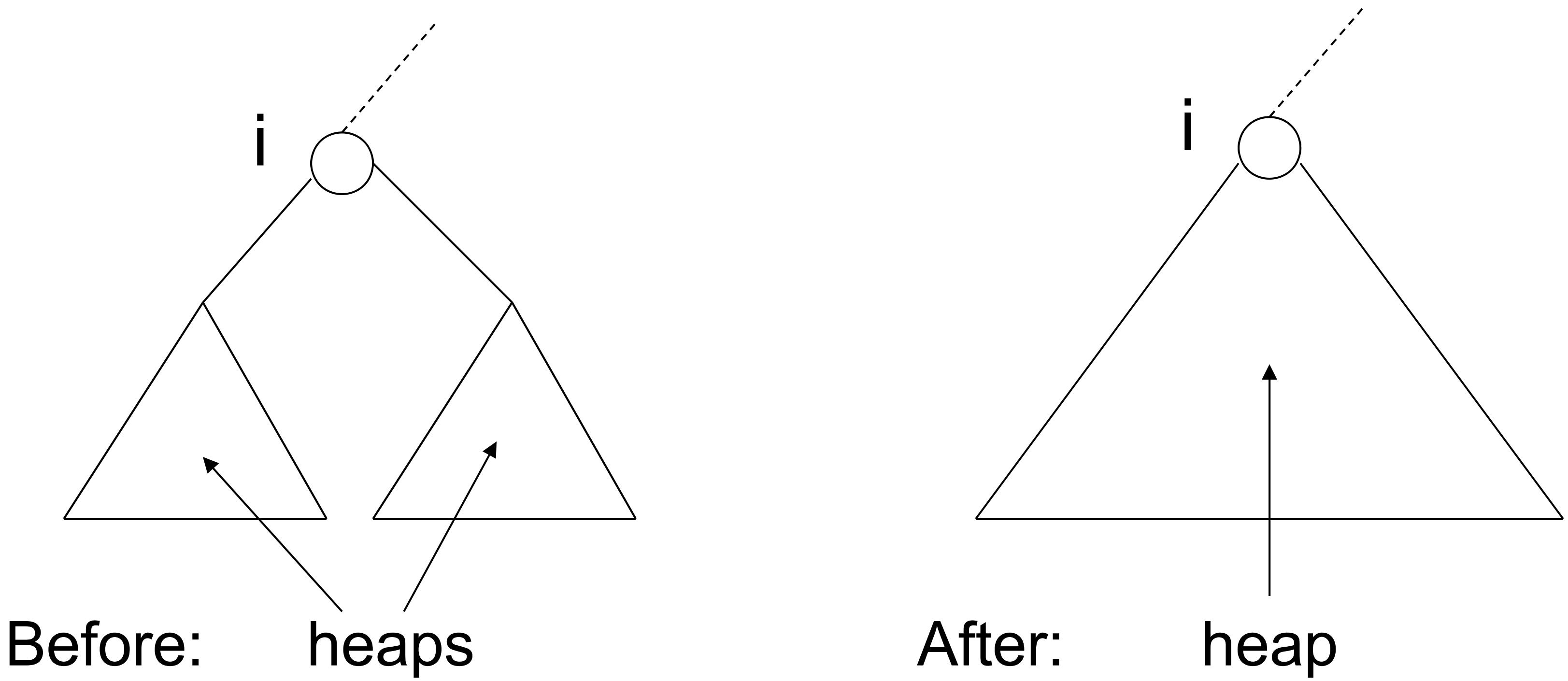
BUILD-HEAP(A)

- Input: Array A[1..n]
- Output: Heap A[1..n] with same elements

for i = [n/2] **down to** 1 **do** HEAPIFY(A,i)

Correctness of Build-Heap

- By induction: After HEAPIFY(A,i) the subtree rooted at i is a heap



Complexity of Build-Heap: O(n)

$$Time = O(\sum_{i=1}^{n} \text{height}(i))$$

height 1: 2^{h-1} nodes
height 2: 2^{h-2} nodes
....
height h: 1 node (the root)

Tree height $h = \#\text{levels-1}$
$= \lceil \log(n+1) \rceil - 1$

$$Time \simeq 1 \cdot 2^{h-1} + 2 \cdot 2^{h-2} + 3 \cdot 2^{h-3} \cdots + h \cdot 2^0$$
$$= \sum_{j=1}^{h} j \cdot 2^{h-j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=1}^{\infty} \frac{j}{2^j} = 2^{h+1} = O(n)$$

Heapsort

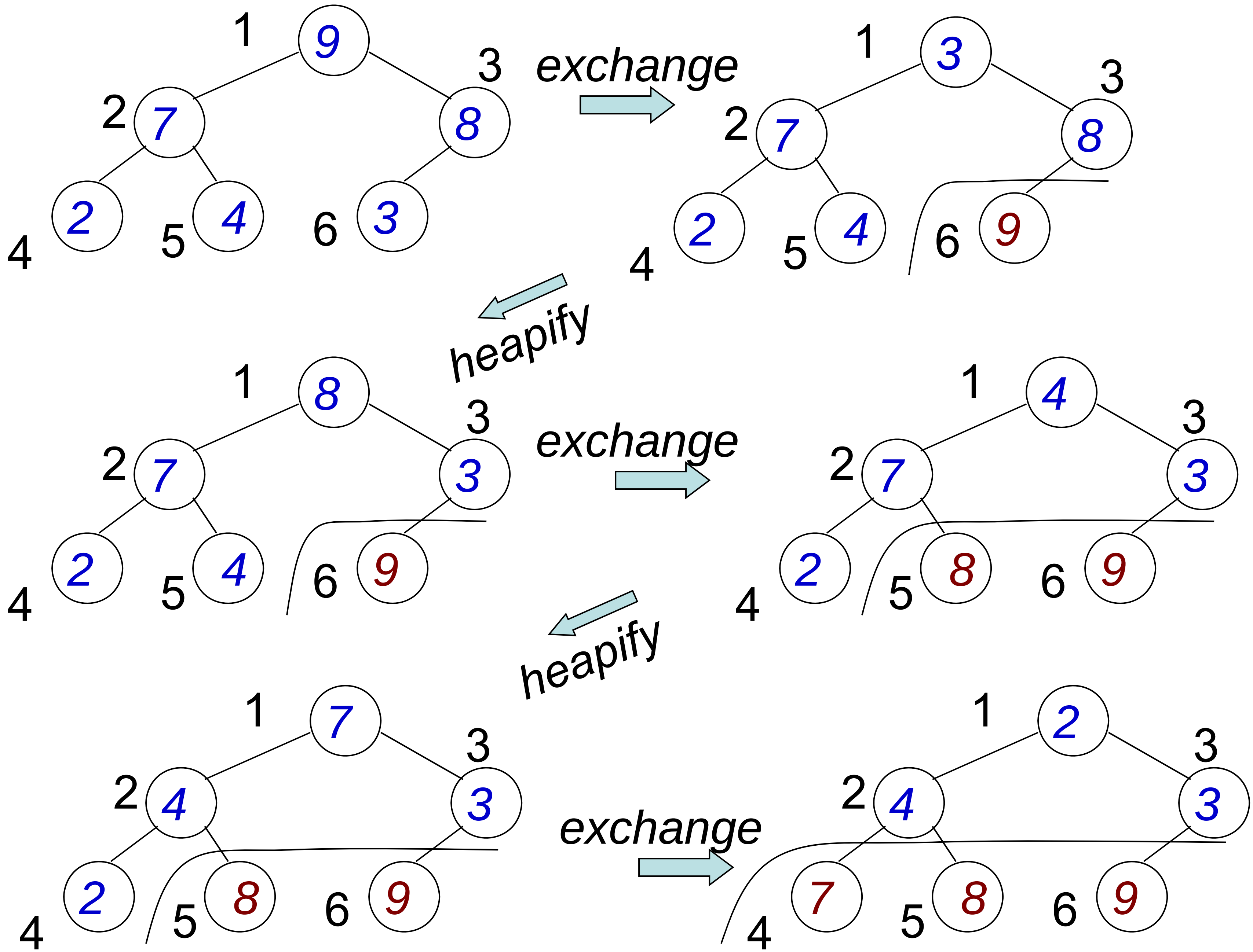
- HEAPSORT(A)
- Input: Array A[1..n]
- Output: Sorted array A[1..n]

BUILD-HEAP(A)

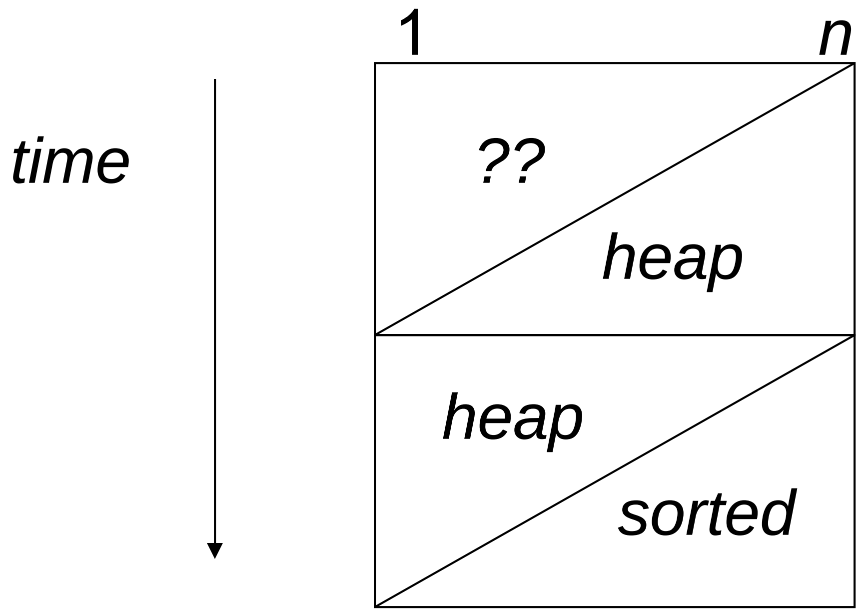
for i = n **down to** 2

$$\left.\begin{array}{l}\{\ \ \text{exchange A[1] and A[i]} \\ \quad\ \text{heap-size(A) = i-1} \\ \quad\ \text{HEAPIFY(A,1)}\end{array}\right\}\ \textit{Extract-Max}$$

}


exchange
heapify
exchange
heapify
exchange

Progress of Heapsort



Phase 1: Build-heap
O(n)

Phase 2: n Extract-max operations: O($n$log$n$)

Time Complexity: Θ($n$log$n$)

Top k (k largest) elements

- Compute *k* largest elements in sorted order in time O(n+$k$log$n$)
- Run phase 2 of Heapsort only for k passes

- Find k largest in *online stream* of n elements, where n>>k using space O(k) in O(nlogk) time
- Keep k largest elements seen so far in a min-heap
- If |heap|=k, compare a new element with the min and if new > min, then extract-min and insert(new)

  otherwise (if < k elements so far), insert(new)

Other Operations

- Delete an element
- Change (Increase/Decrease) a key

O(logn) per operation

- Join (=Merge, Union)
- Heaps do not support fast join
- Other priority queues can – see chapter 19