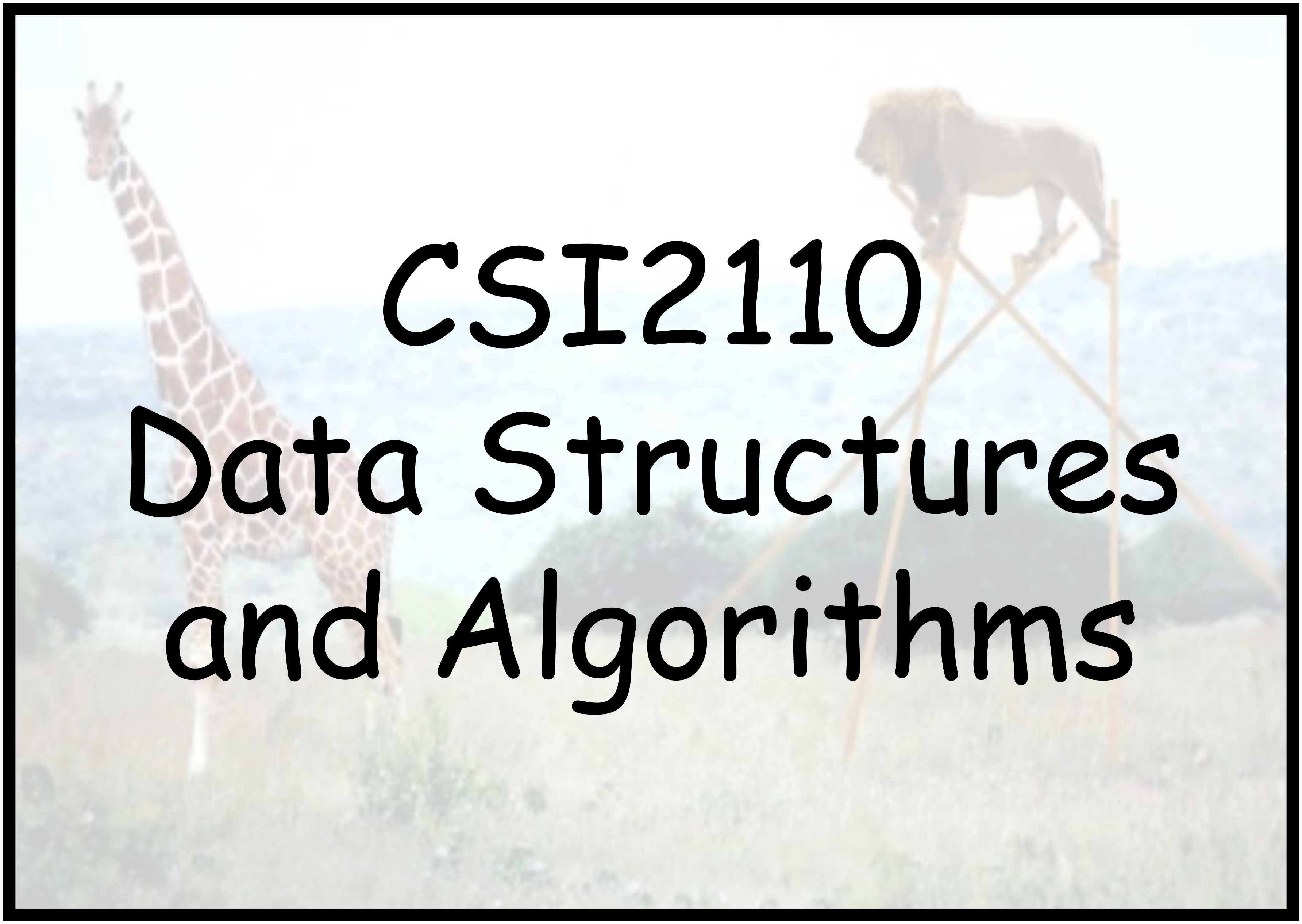


CSI2110 Data Structures and Algorithms

Heaps

- Heaps
- Properties
- Deletion, Insertion, Construction
- Implementation of the Heap
- Implementation of Priority Queue using a Heap
- An application: HeapSort

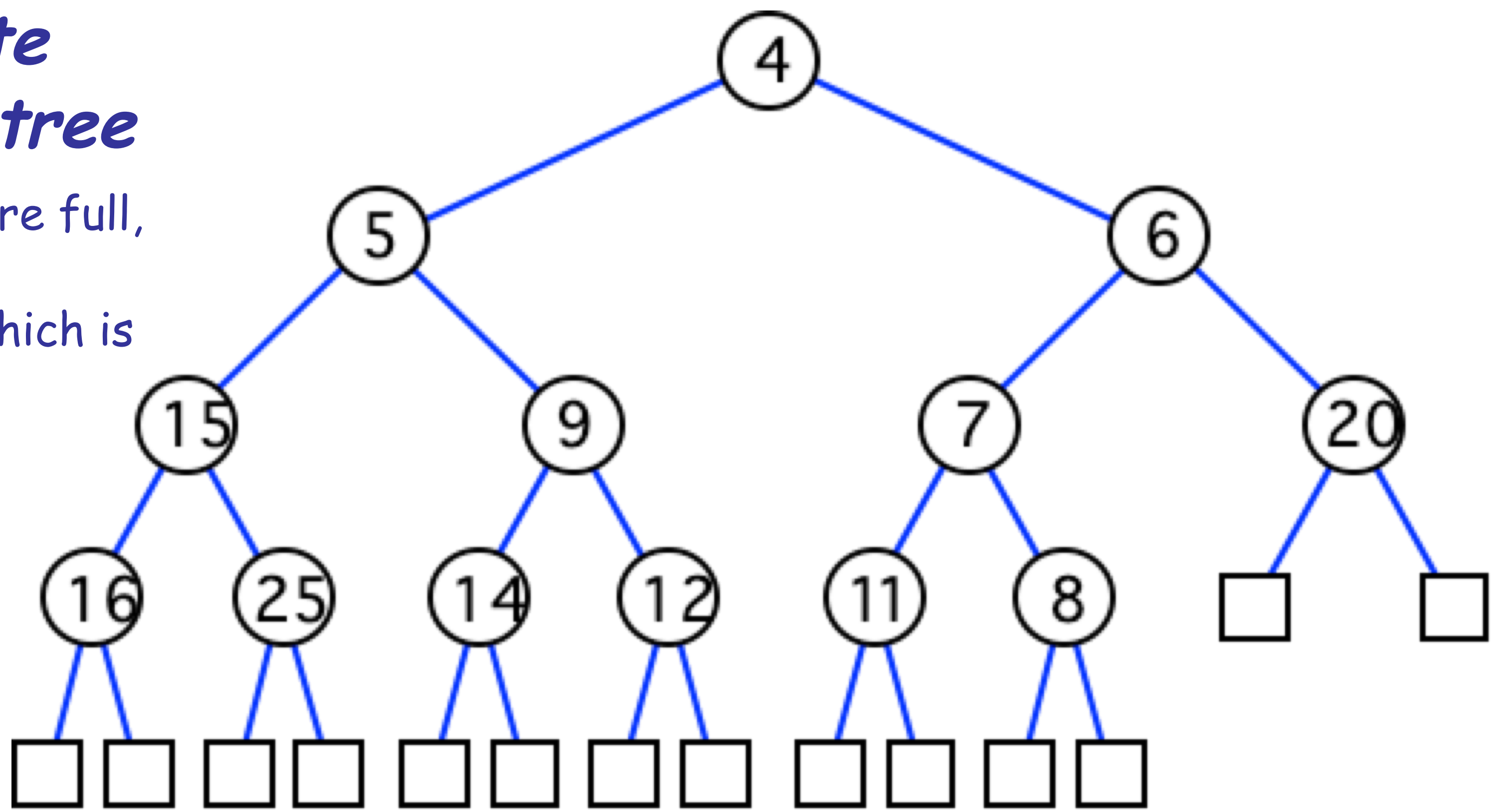
Heaps (Min-heap)

Complete binary tree that stores a collection of keys (or key-element pairs) at its *internal* nodes and that satisfies the additional property:

key(parent) ≤ key(child)

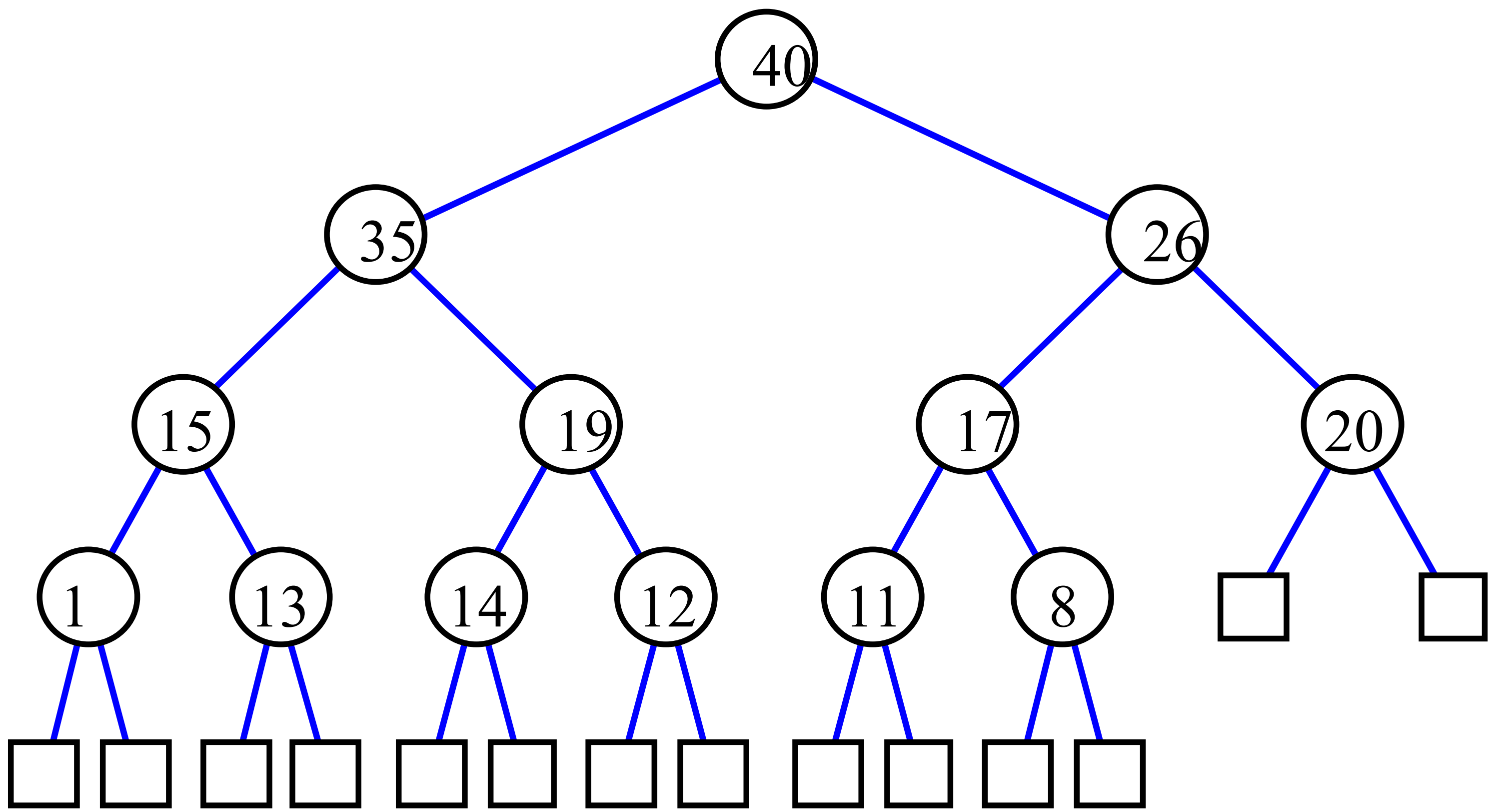
REMEMBER:
complete binary tree

all levels are full, except the last one, which is left-filled

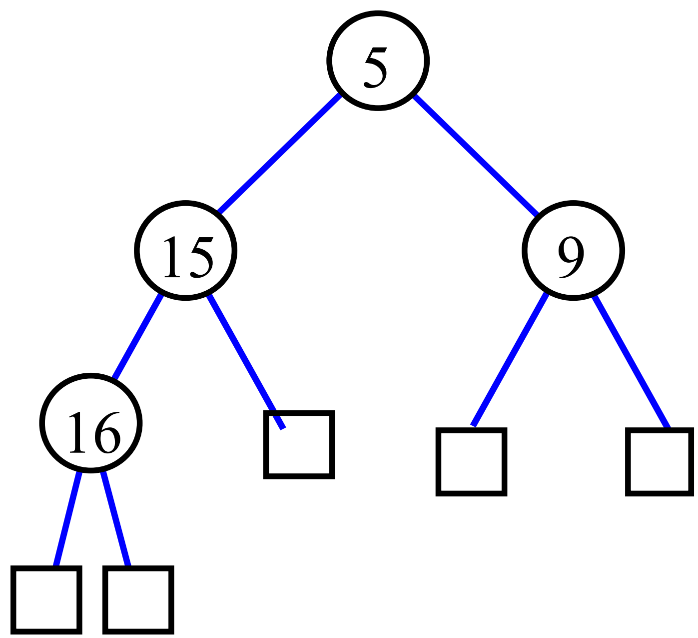


Max-heap

key(parent) ≥ key(child)



We store the keys in the internal nodes only

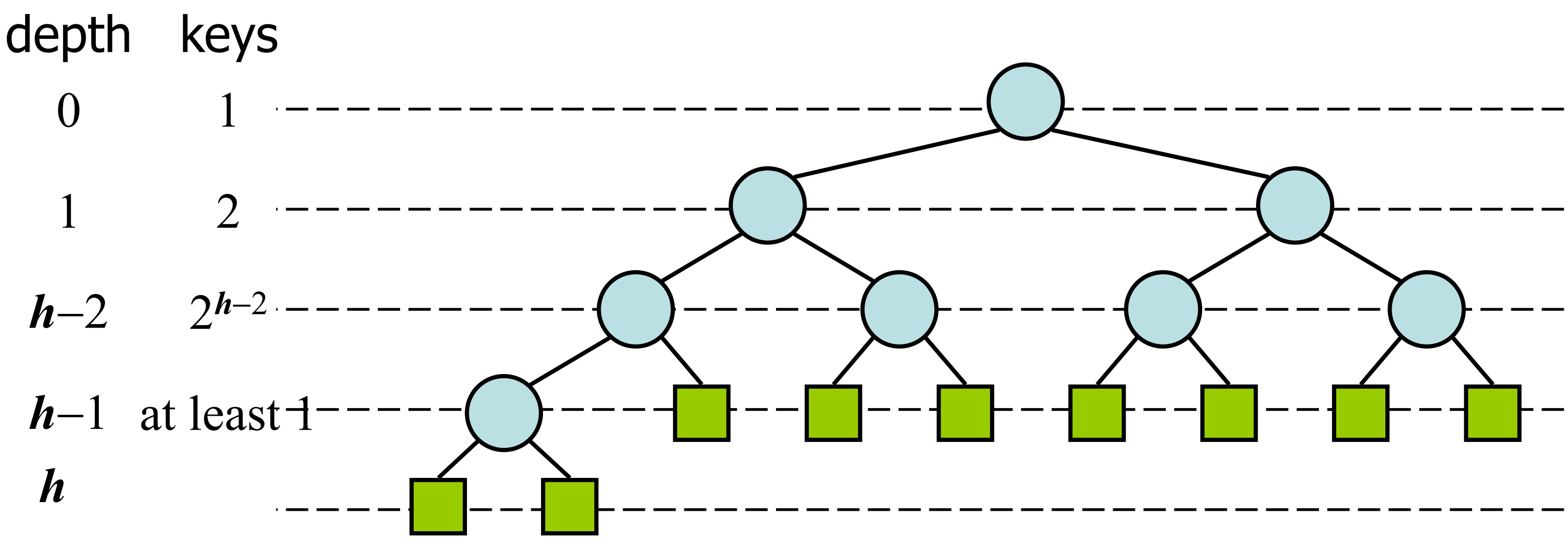


After adding the □ leaves the resulting tree is full

Height of a Heap

- Theorem: A heap storing n keys has height $O(\log n)$

  Proof:
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h - 2$ and at least one key at depth $h - 1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



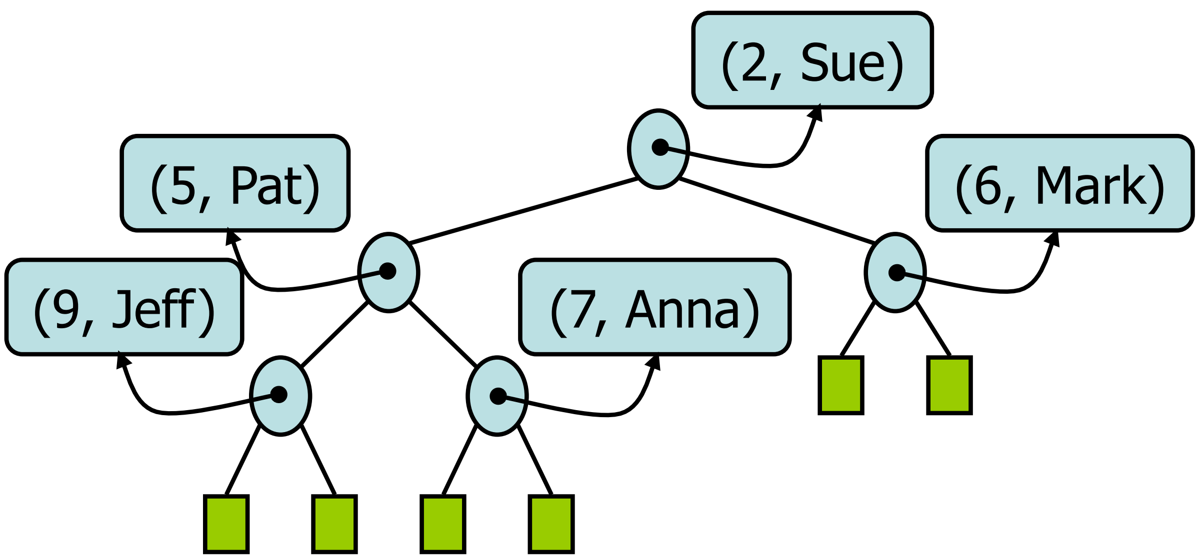
Notice that ....

- We could use a heap to implement a priority queue
- We store a (key, element) item at each internal node

removeMin():

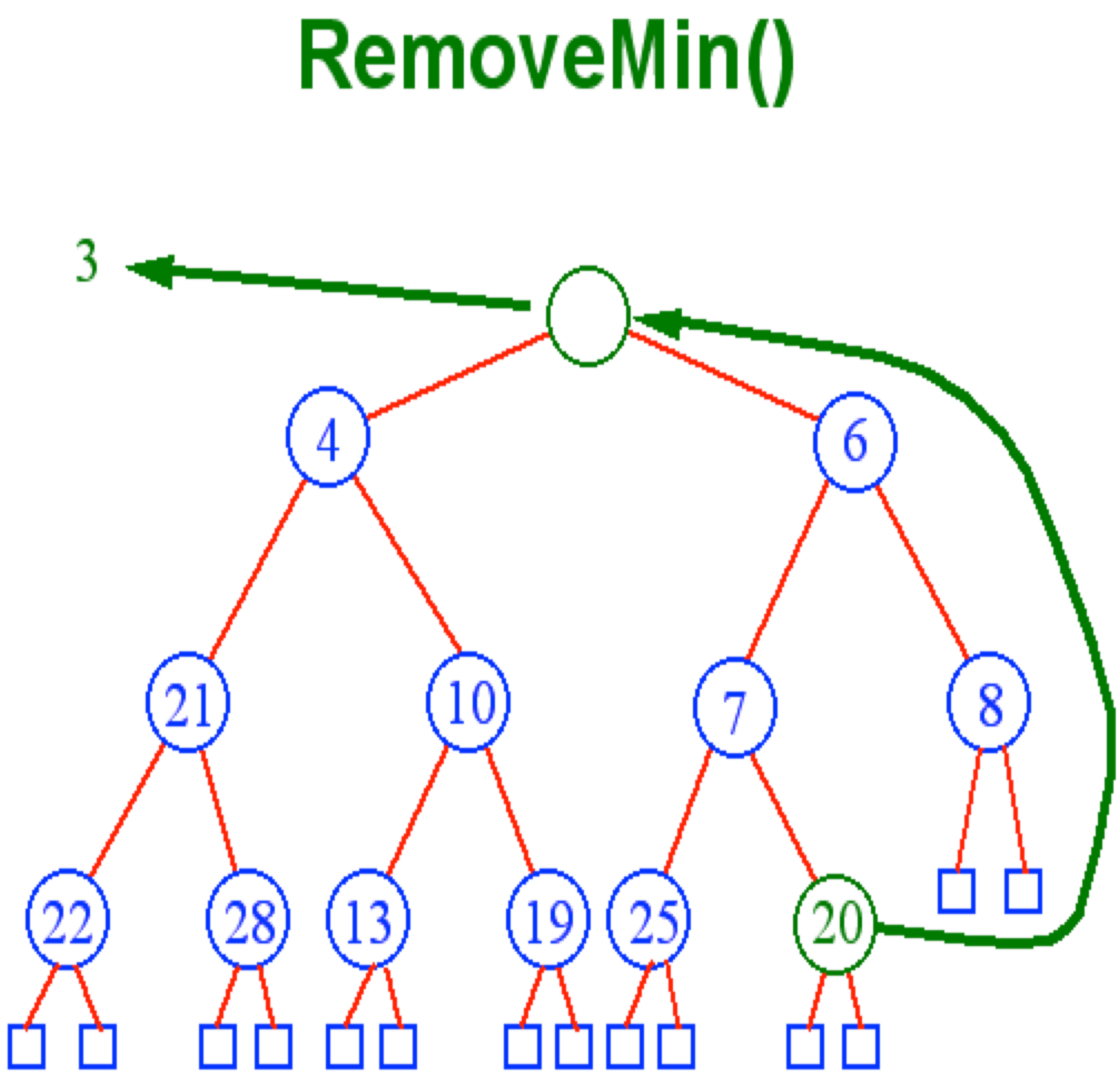
→ Remove the root

→ Re-arrange the heap!

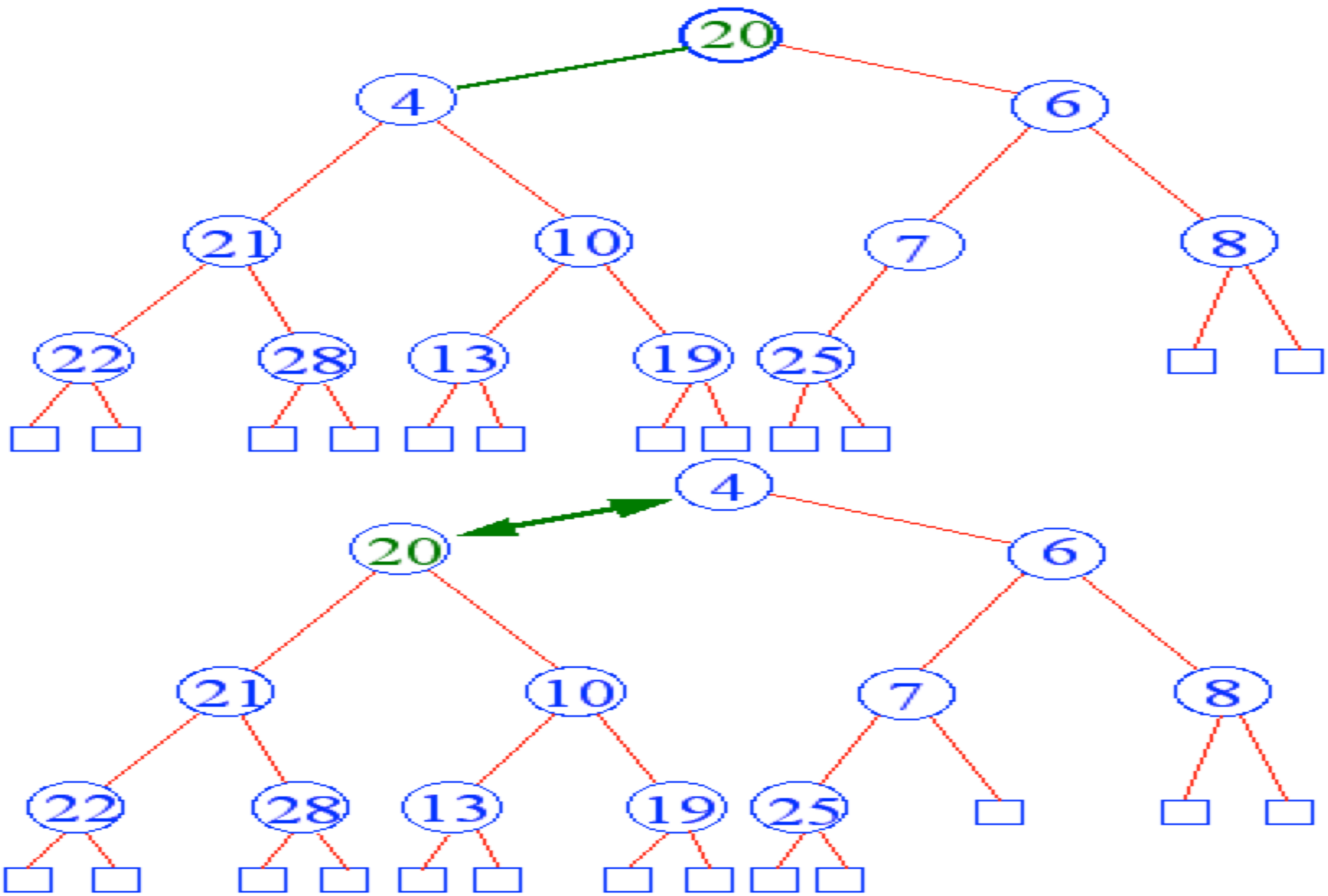


Removal From a Heap

- The removal of the top key leaves a hole
- We need to fix the heap
- First, replace the hole with the last key in the heap
- Then, begin Downheap …

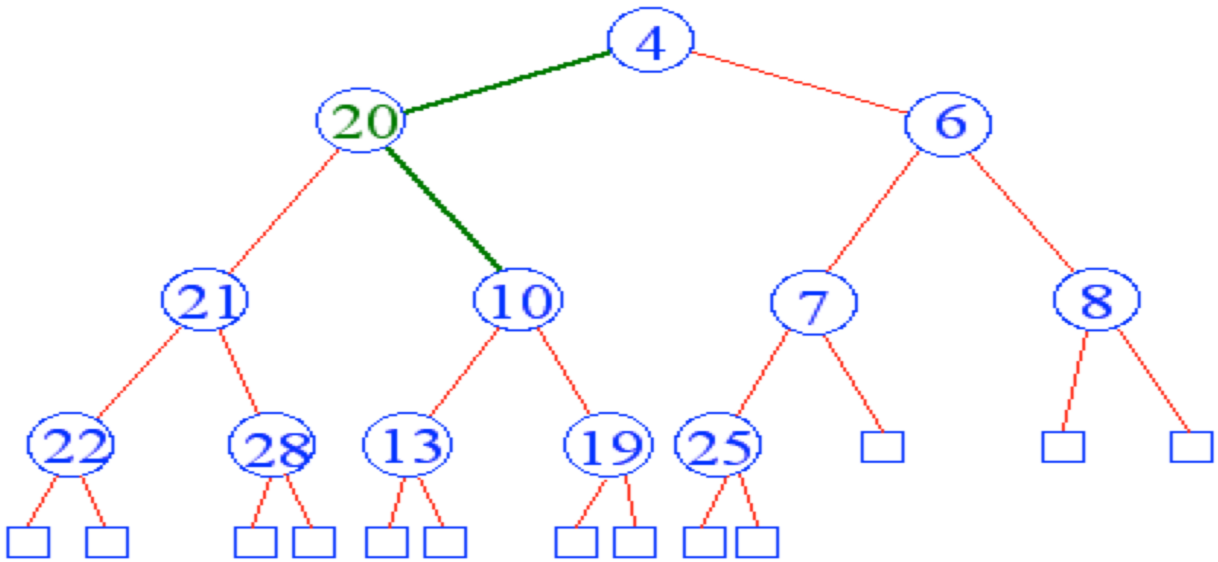


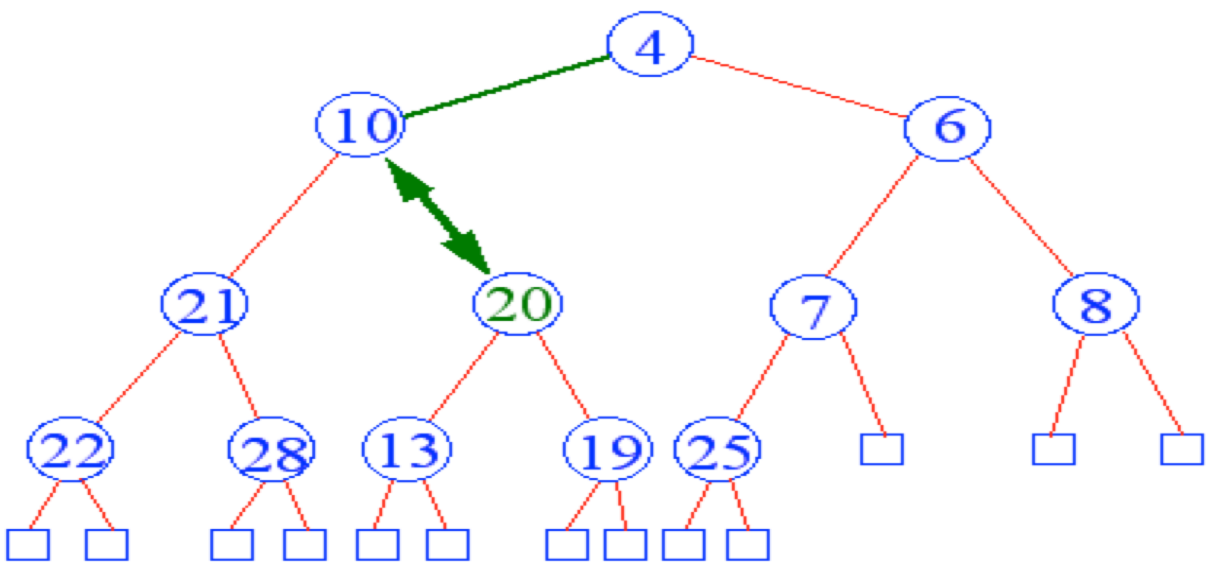
Downheap



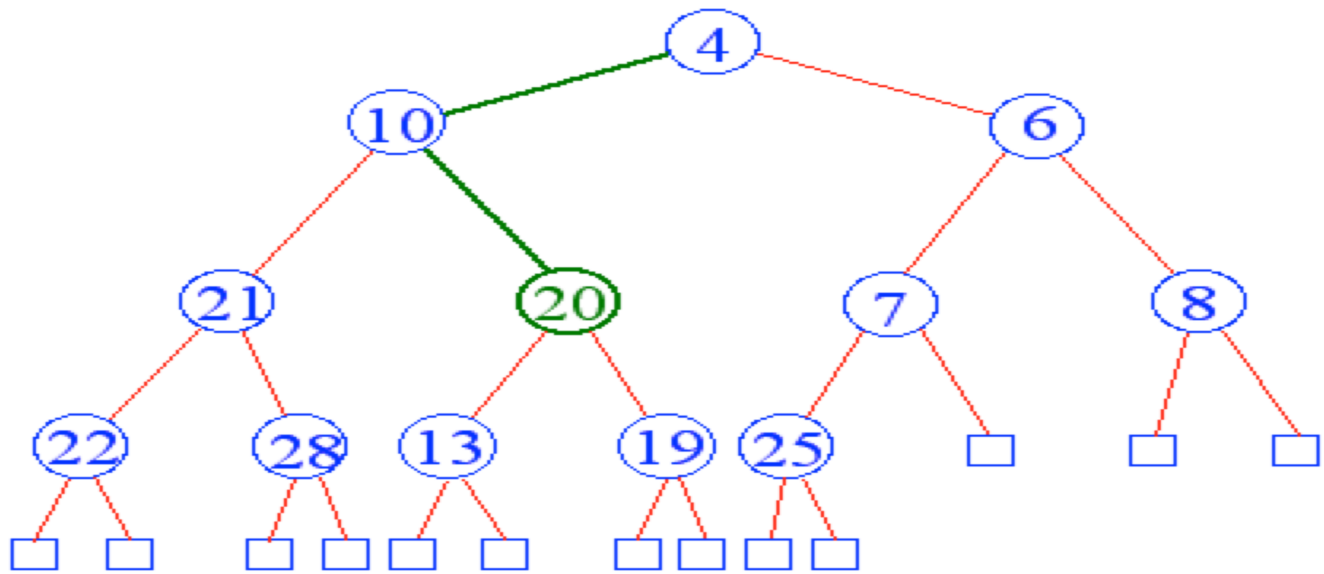
- Downheap compares the parent with the smallest child. If the child is smaller, it switches the two.

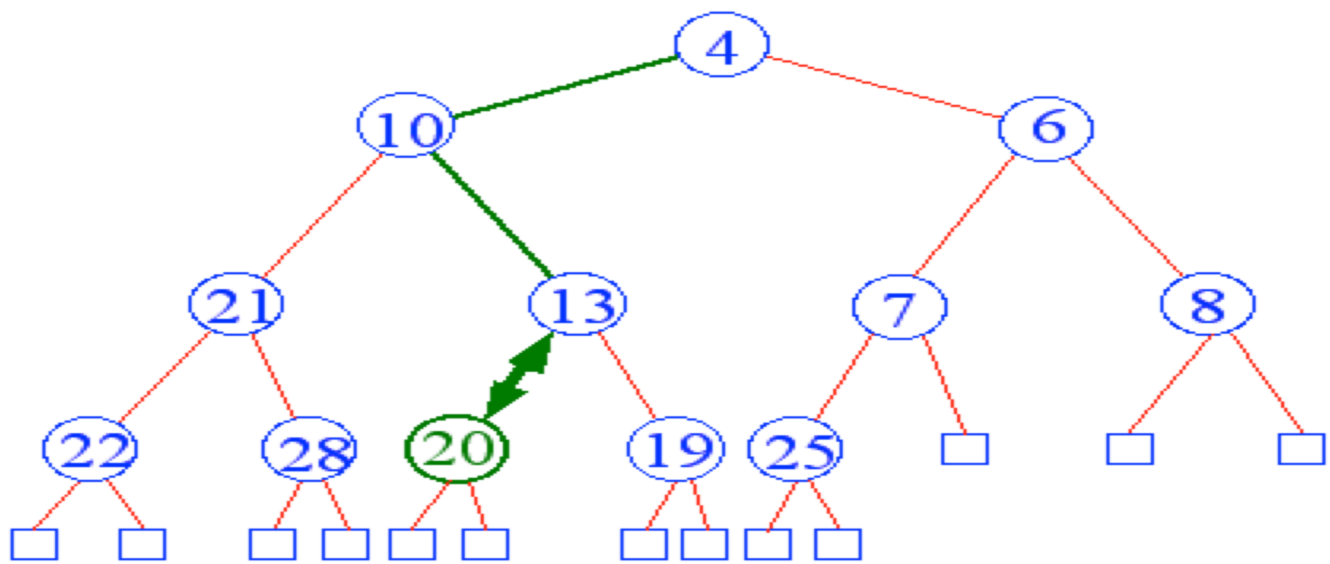
Downheap Continues



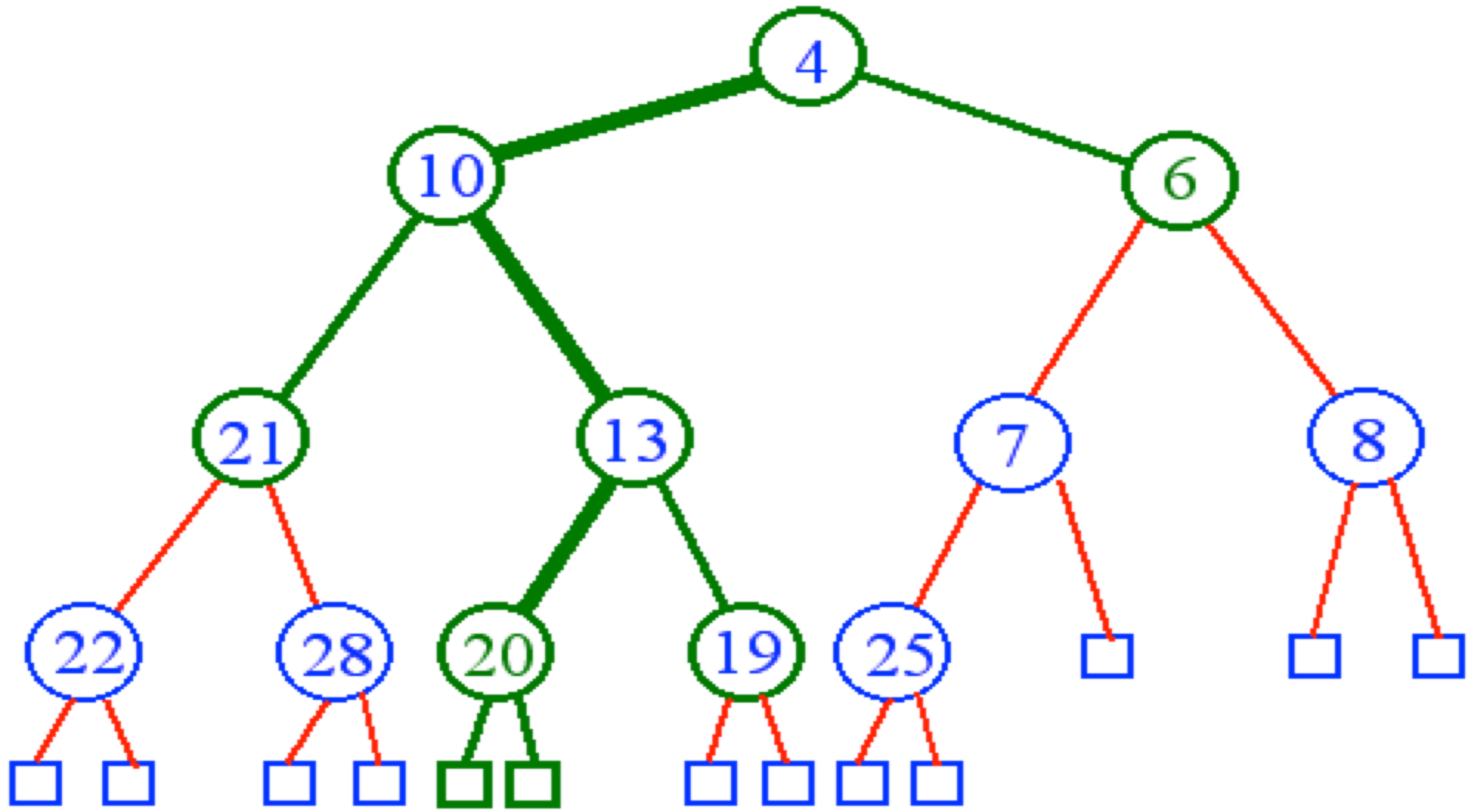


Downheap Continues





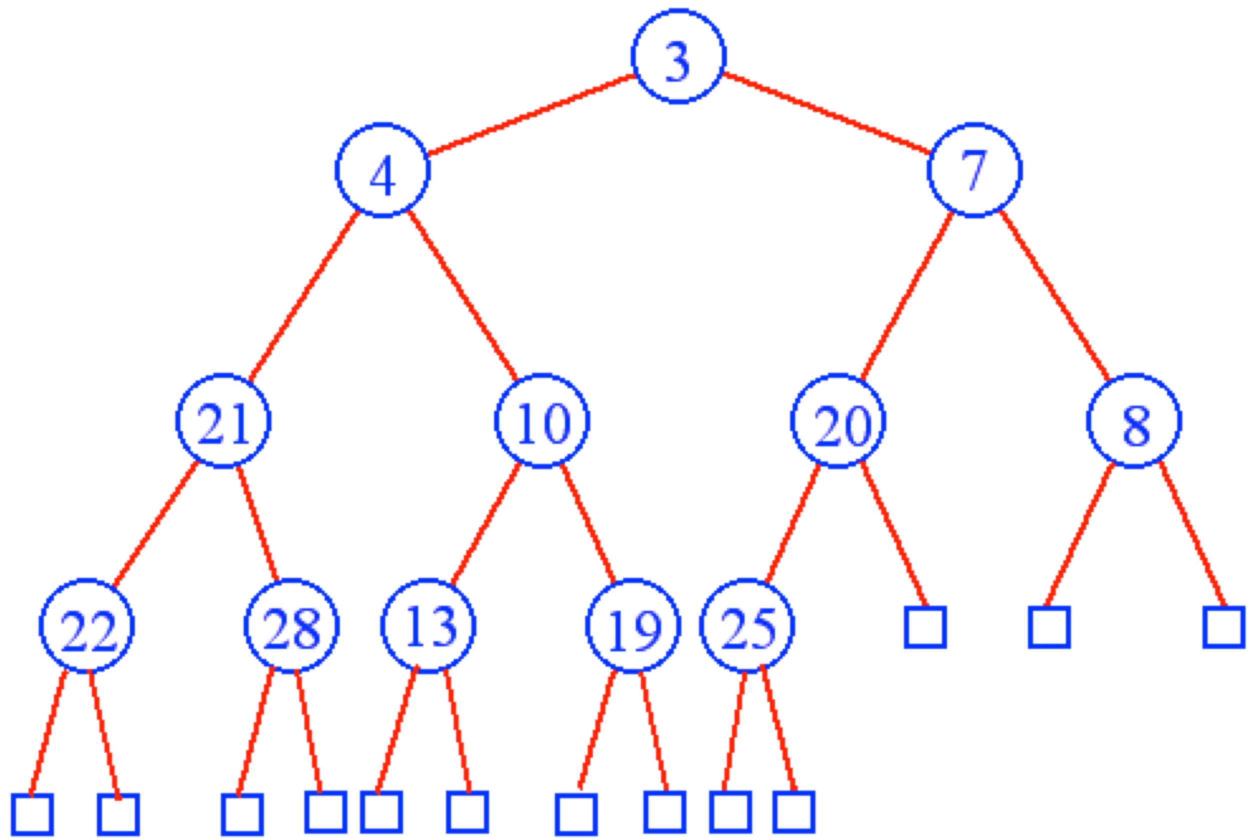
End of Downheap



- Downheap terminates when the key is greater than the keys of both its children or the bottom of the heap is reached.
  - (total #swaps) $\leq$ $(h - 1)$, which is $O(\log n)$

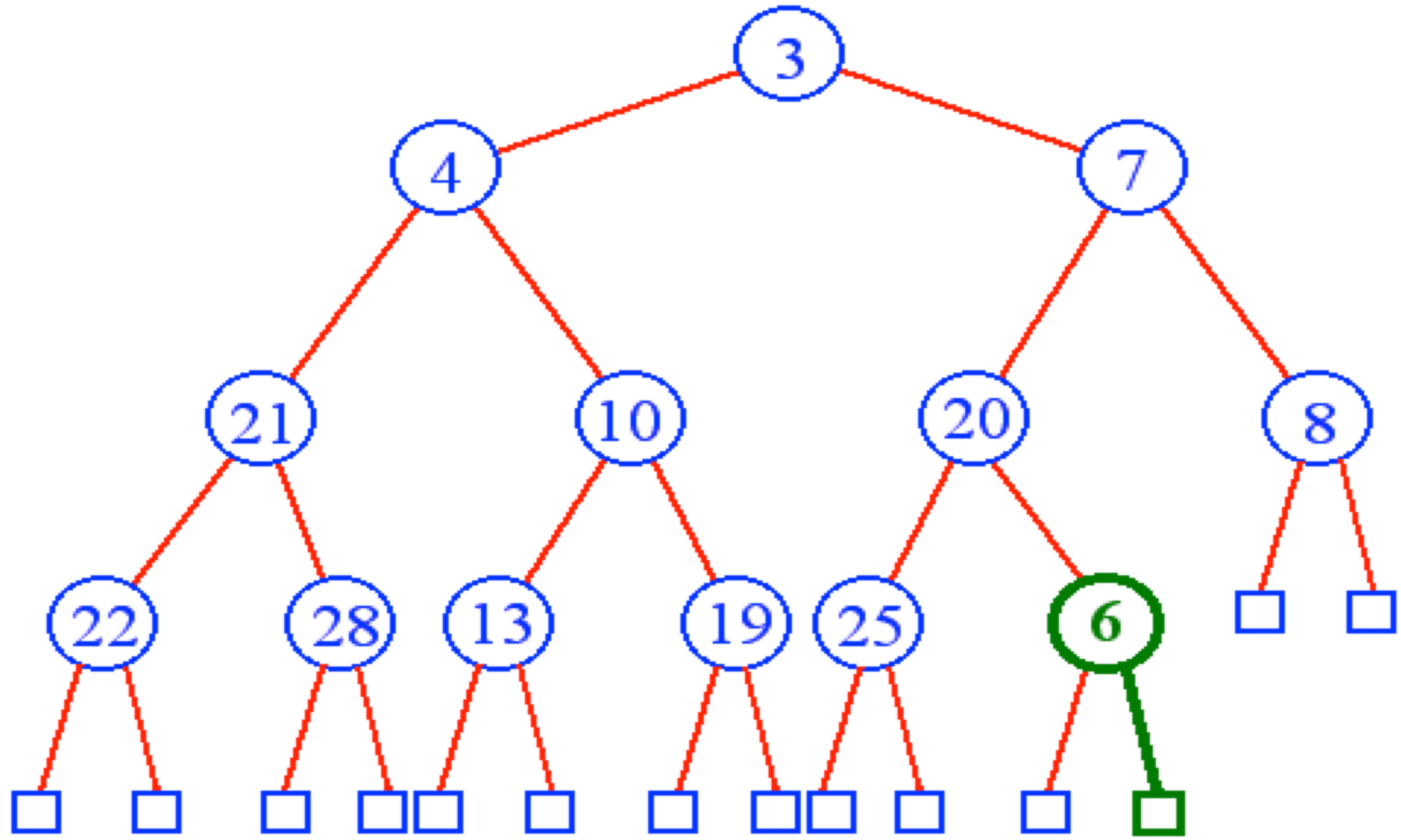
Heap Insertion

The key to insert is 6



Heap Insertion

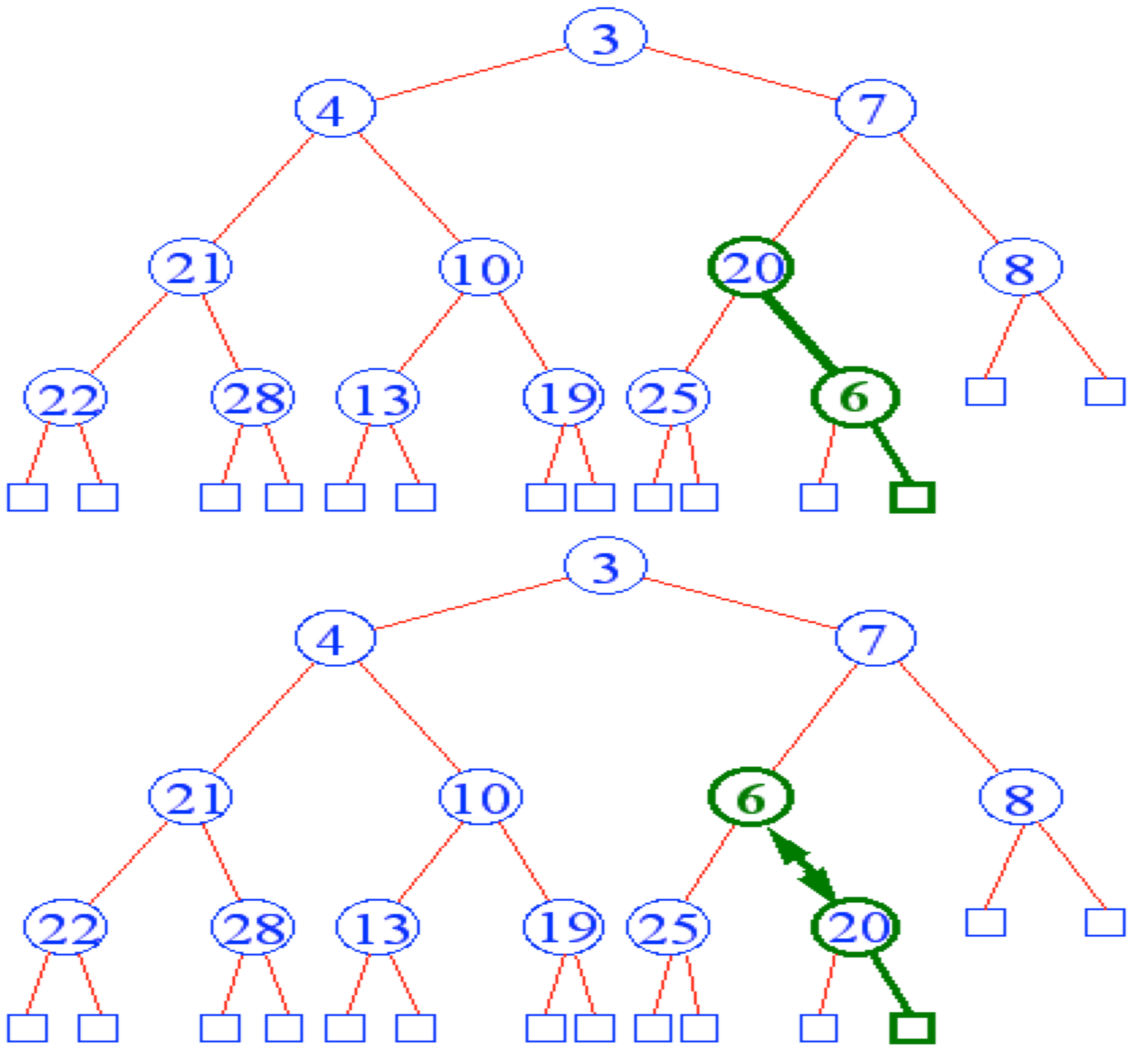
Add the key in the ***next available position*** in the heap.



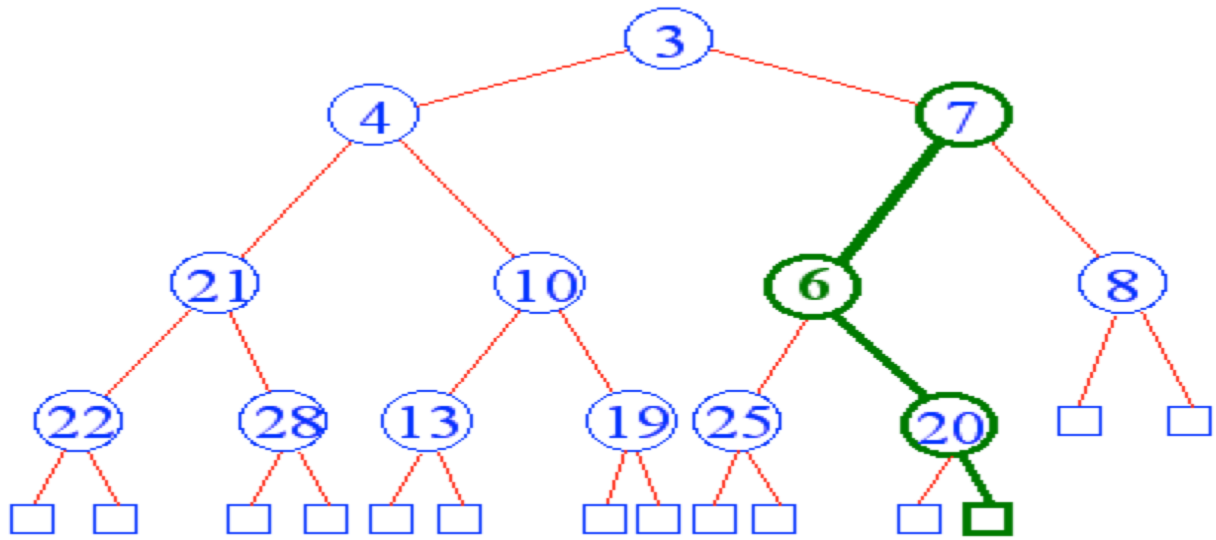
Now begin ***Upheap***.

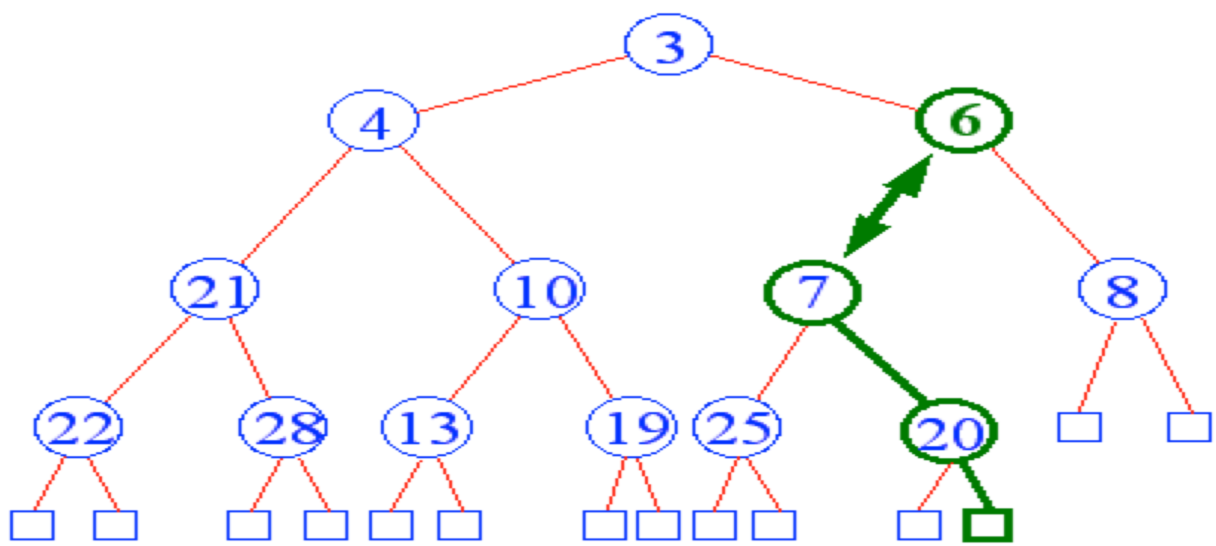
Upheap

- Swap parent-child keys out of order

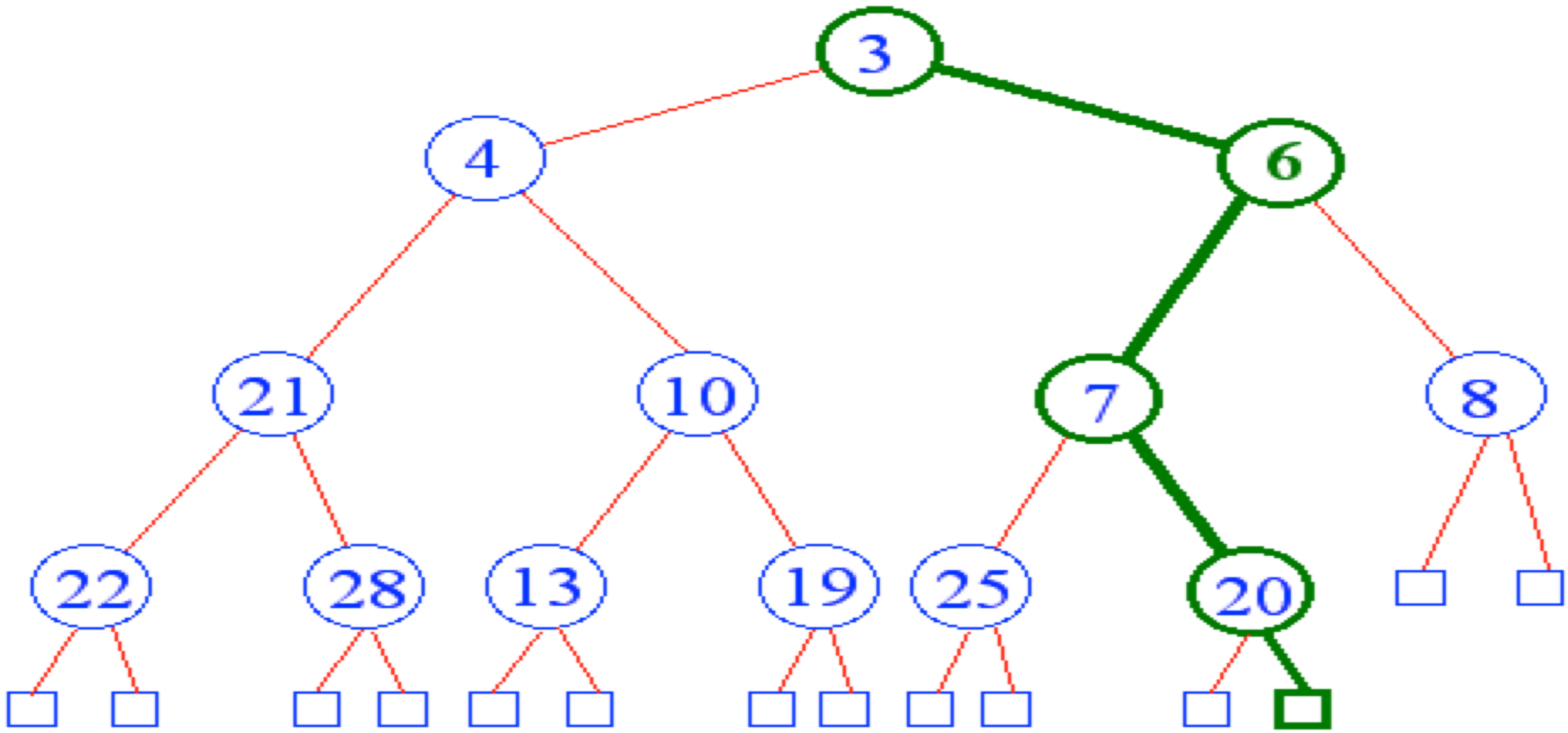


Upheap Continues





End of Upheap



- Upheap terminates when new key is greater than the key of its parent c≤ the top of the heap is reached
- (total #swaps)   (h - 1), which is O(log n)

Heap Construction

We could insert the Items one at the time with a sequence of Heap Insertions:

$$\sum_{k=1}^{n} \log k = O(n \log n)$$

But we can do better ....

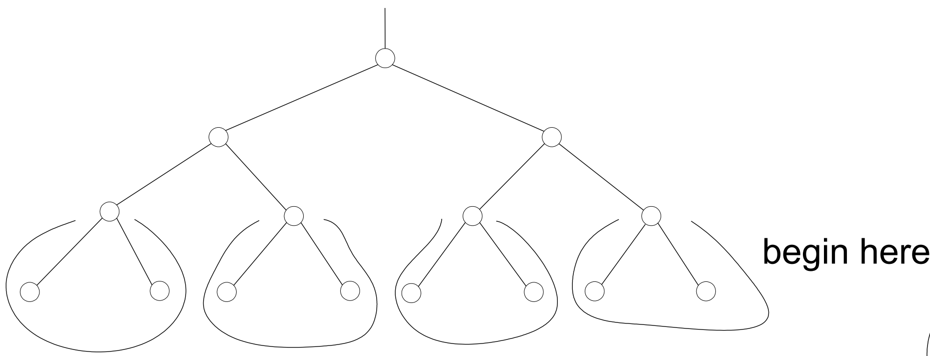
O(n) using Bottom-up Heap Construction

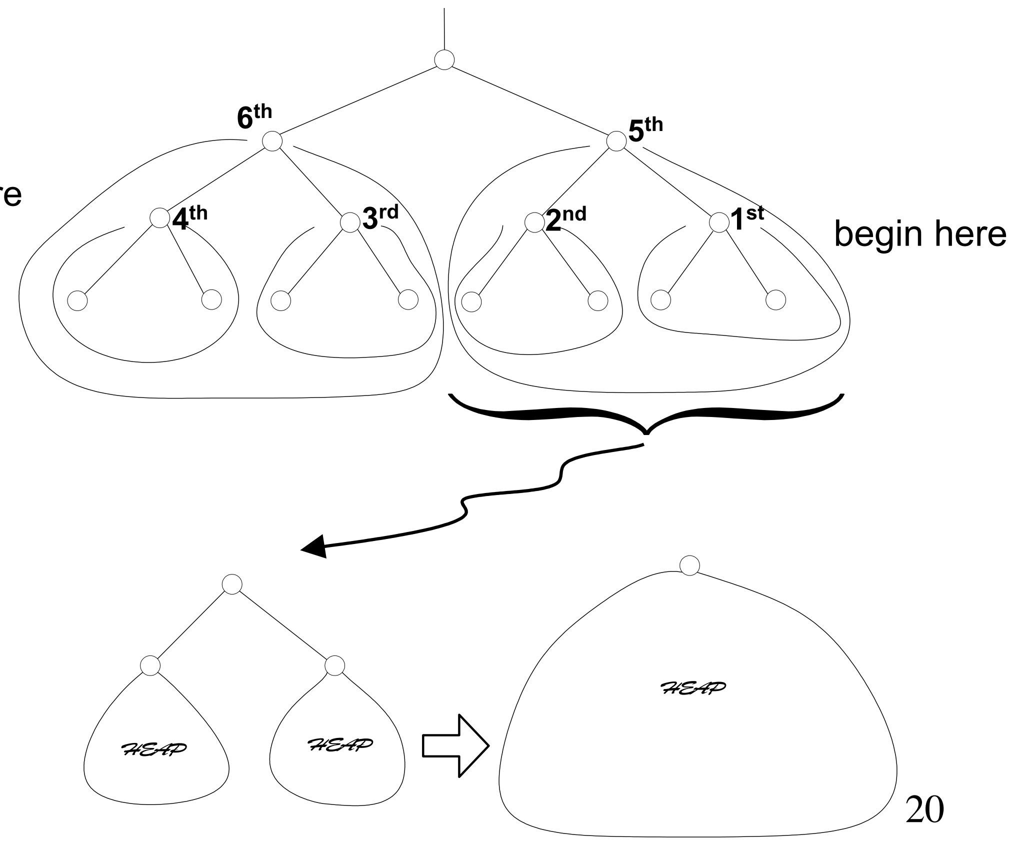
Bottom-up Heap Construction

- We can construct a heap storing ***n*** given keys using a bottom-up construction

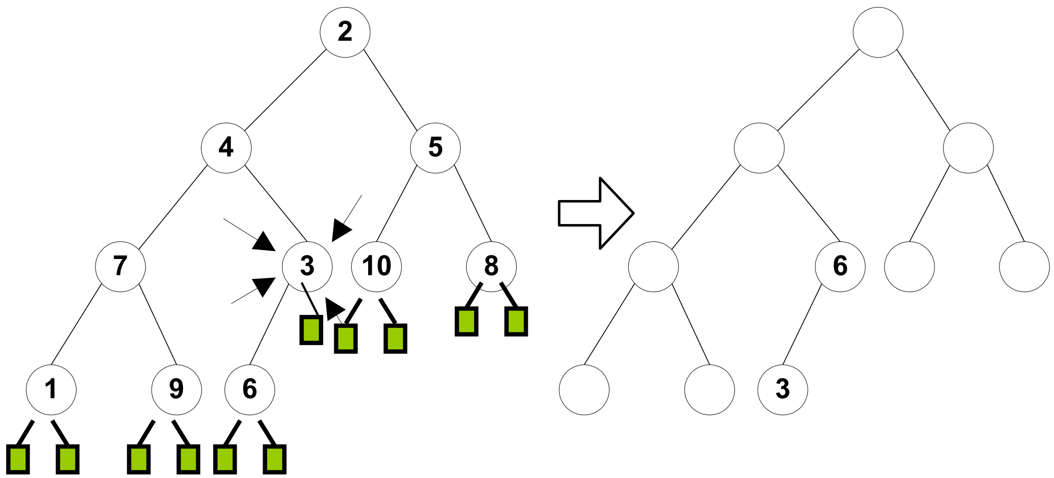
Construction of a Heap

Idea: Recursively re-arrange each sub-tree in the heap starting with the leaves





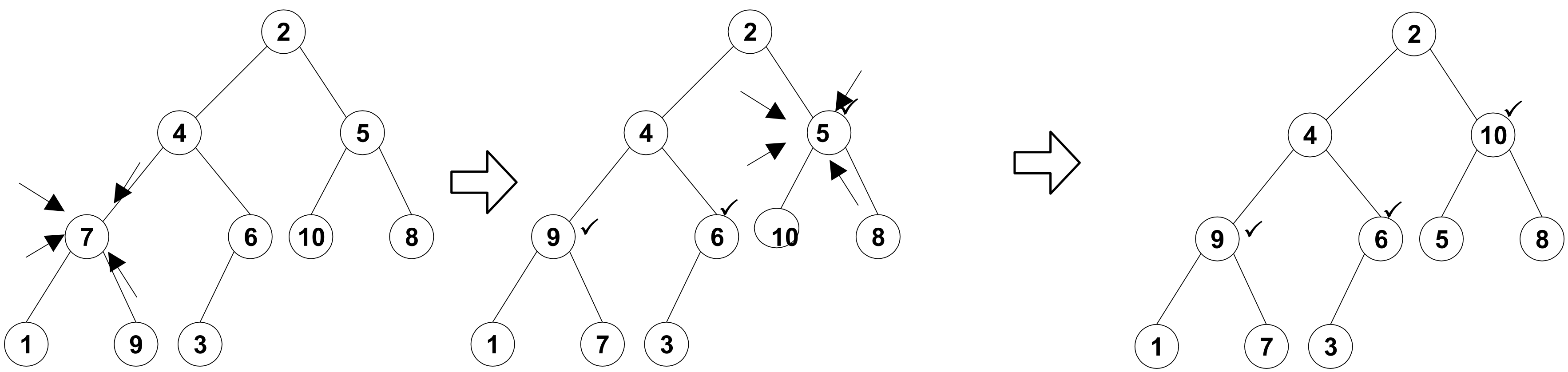
Example 1 (Max-Heap)



$\underline{3} \leftrightarrow 6$

--- keys already in the tree ---

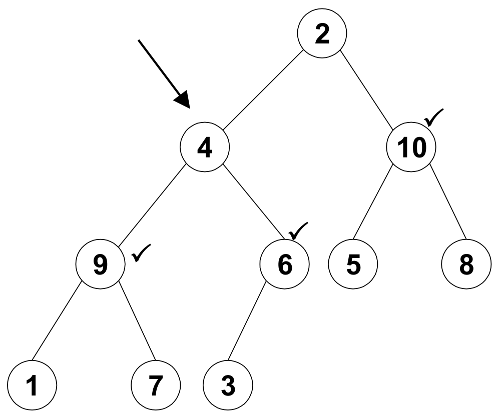
I am now drawing the leaves anymore here



$\underline{7} \leftrightarrow 9$

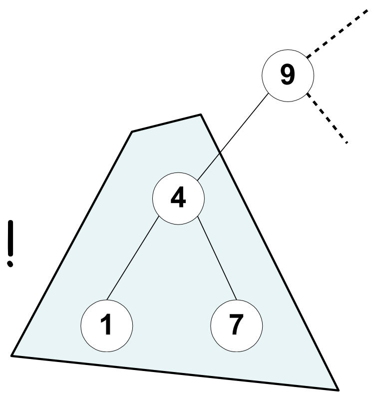
$\underline{5} \leftrightarrow 10$

Example 1

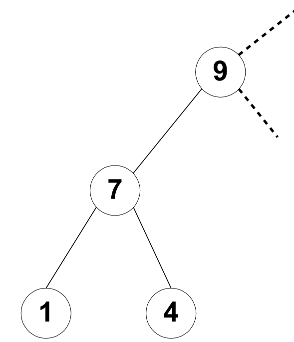


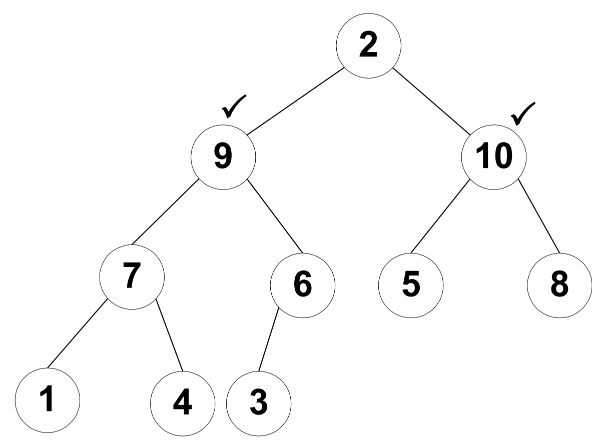
$\underline{4} \leftrightarrow 9$

This is not a heap !

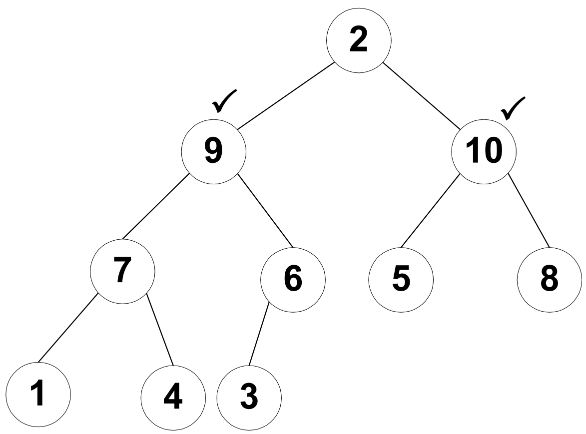


$\underline{4} \leftrightarrow 7$

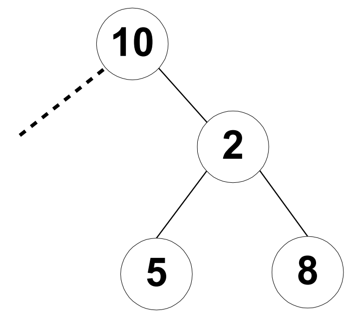


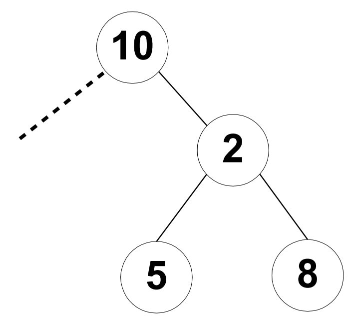


Example 1

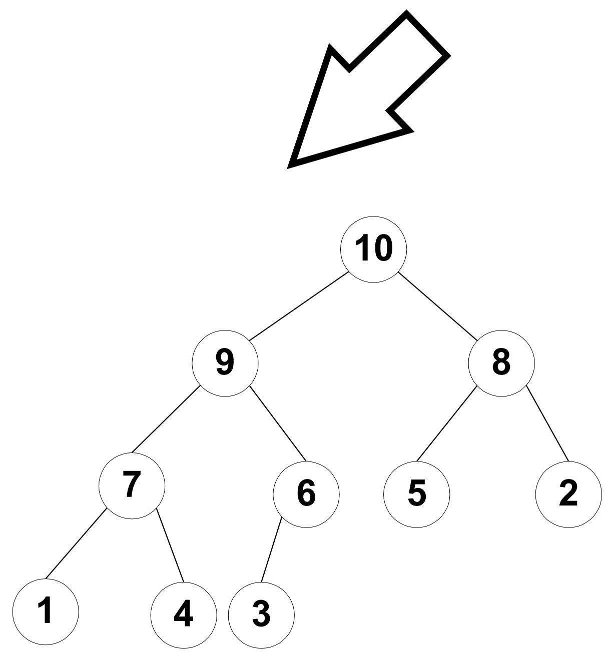


Finally: $\underline{2} \leftrightarrow 10$





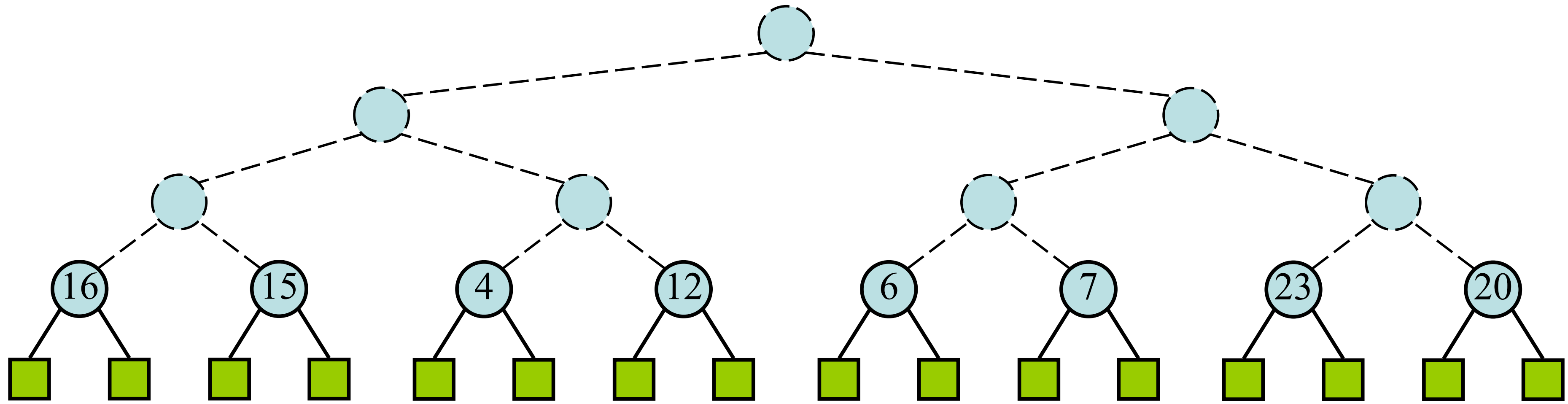
$\underline{2} \leftrightarrow 8$

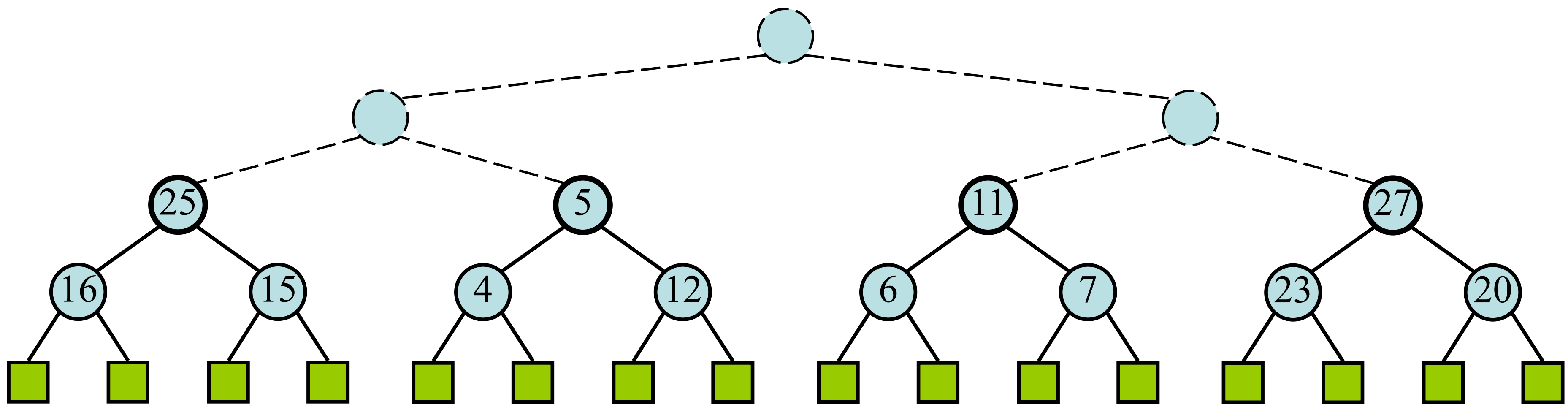


--- keys given one at a time ---

Example 2 (min-heap)

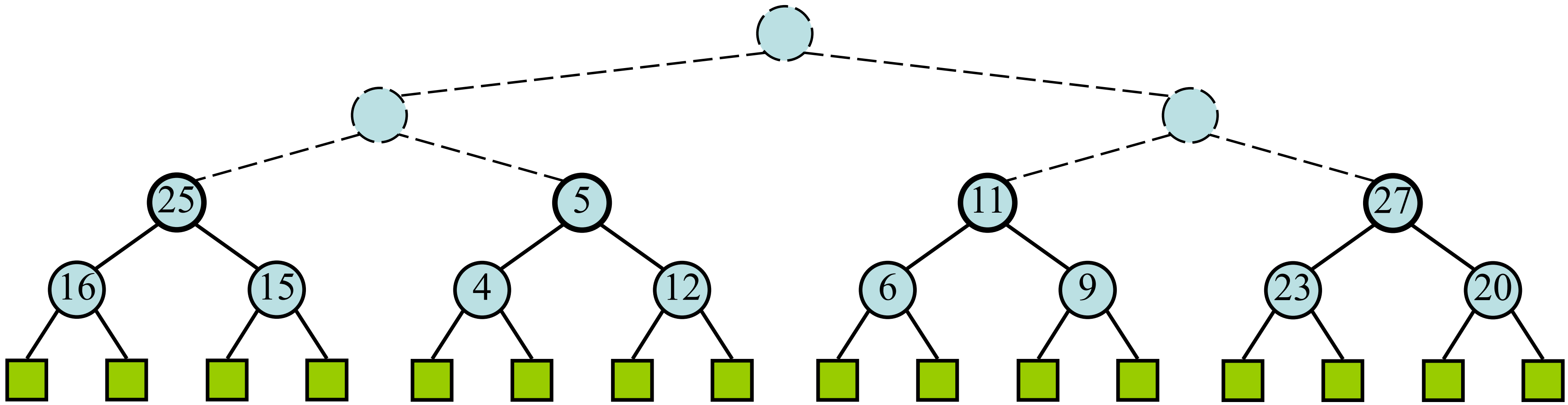
[20,23,7,6,12,4,15,16,27,11,5,25,8,7,10]

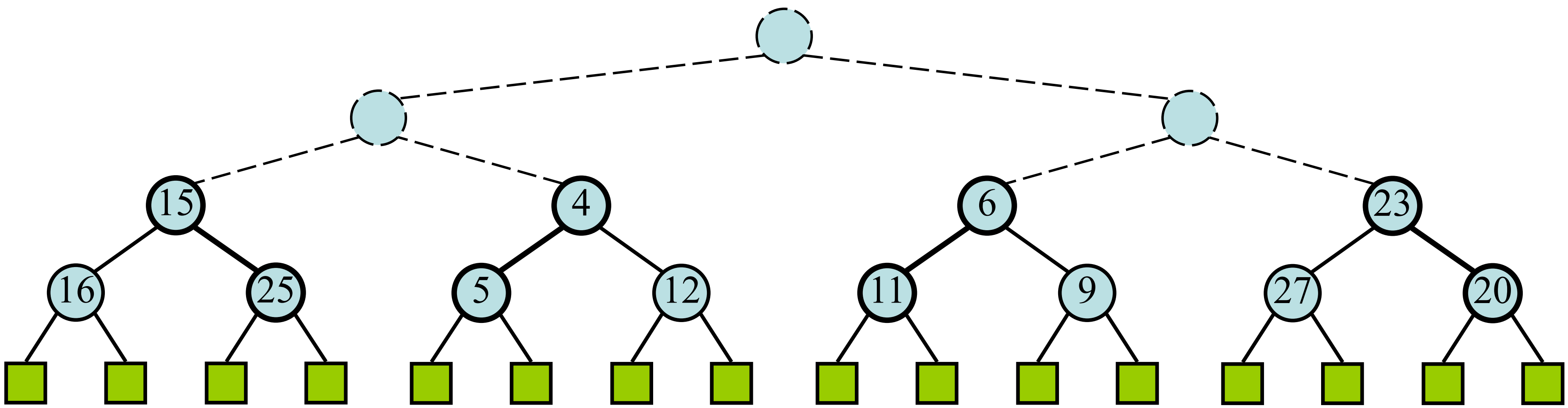




Example 2

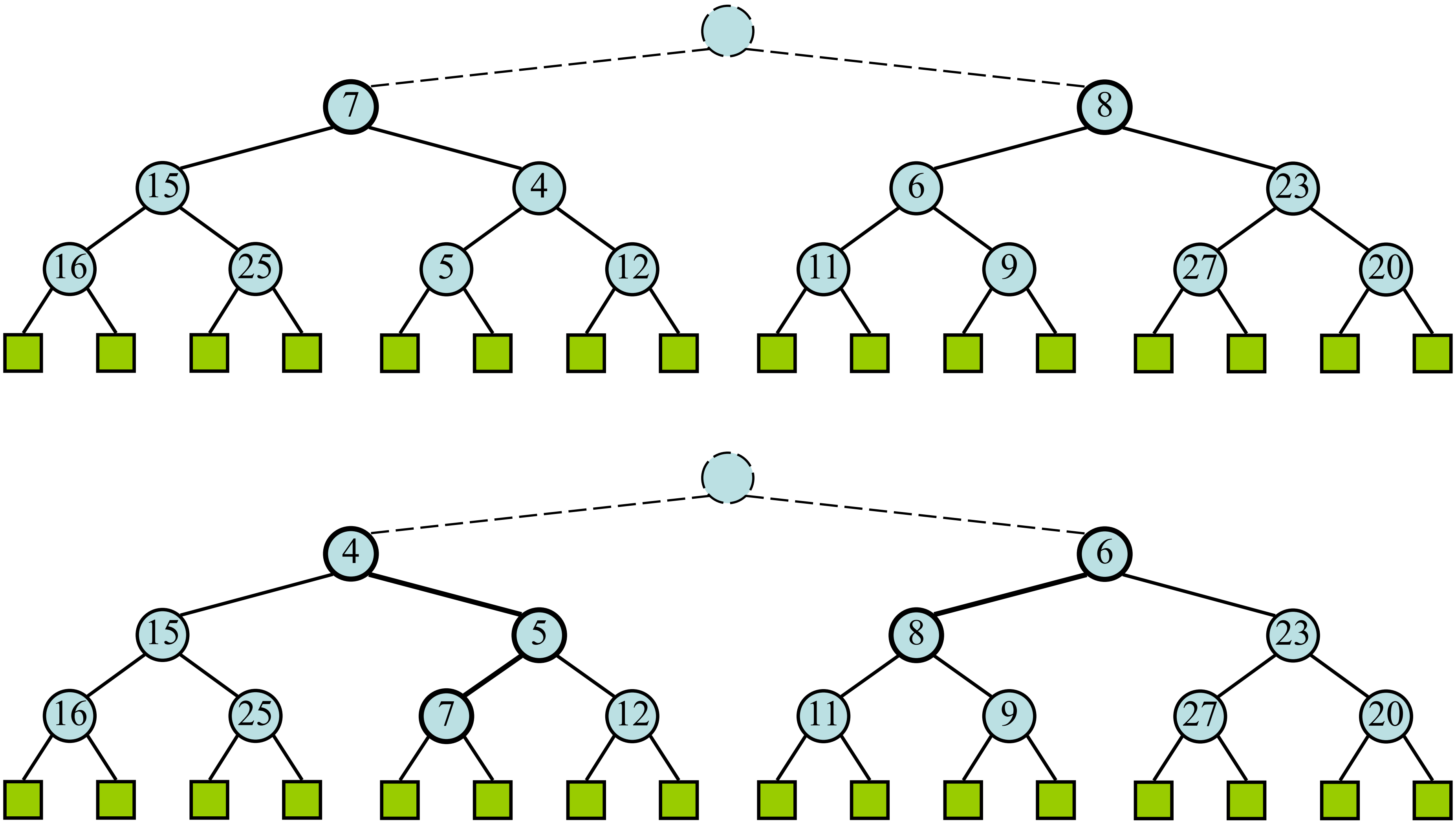
20,23,7,6,12,4,15,16,27,11,5,25,8,7,10]





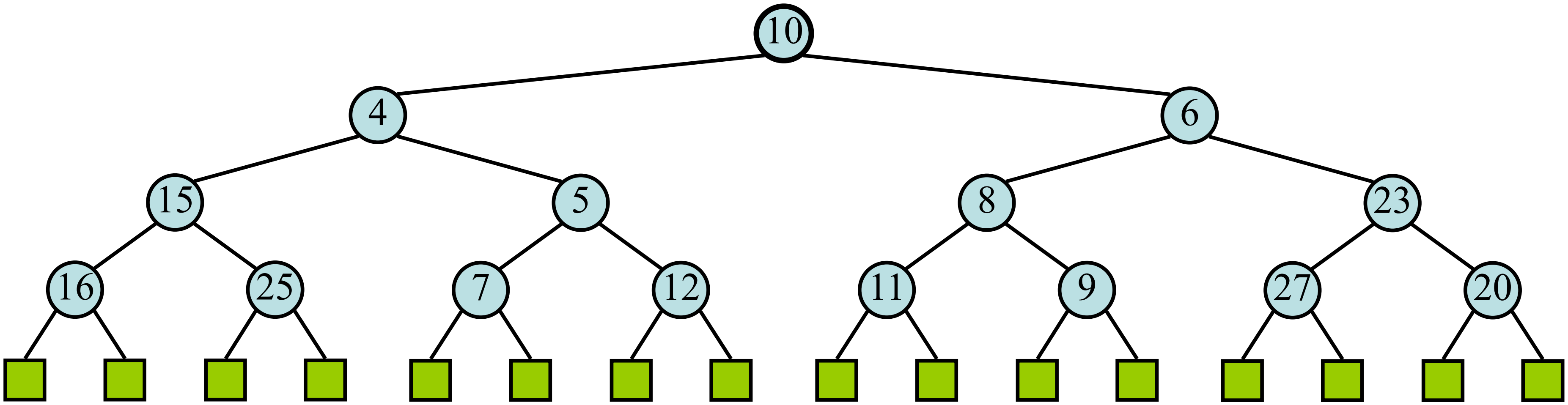
Example 2

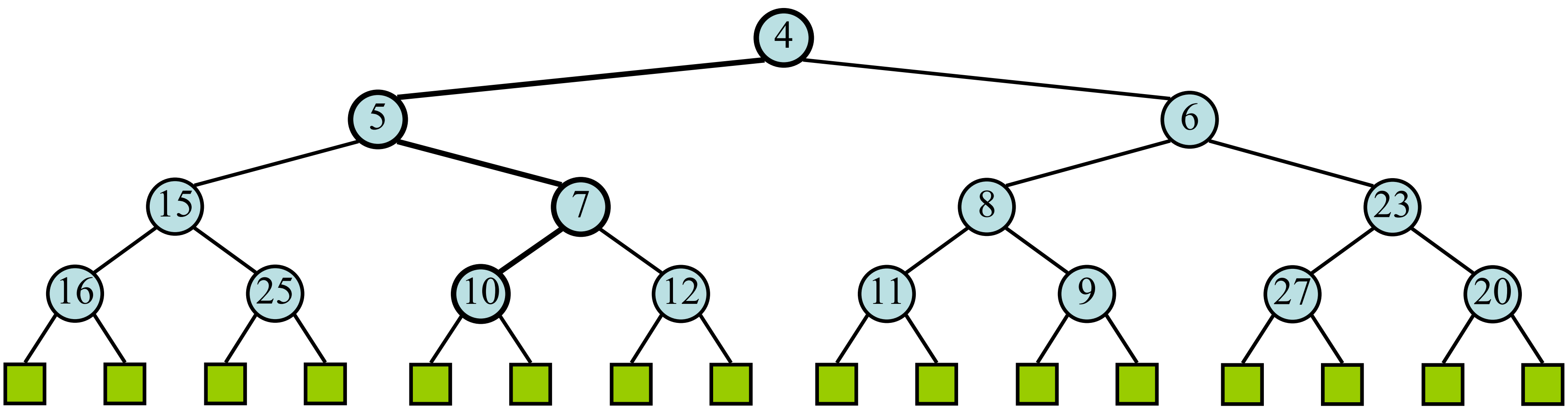
20,23,7,6,12,4,15,16,27,11,5,25,8,7,10]



Example 2

20,23,7,6,12,4,15,16,27,11,5,25,8,7,10]





Analysis of Heap Construction

Number of swaps

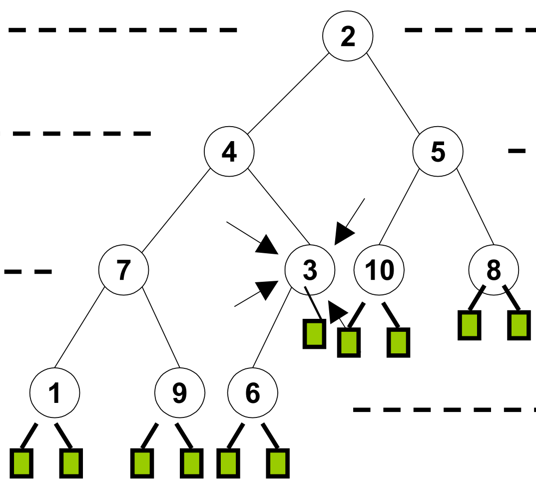
h = 4

3 swaps -------- level 0

2 swaps -------- level 1

1 swap ------- level 2

0 swaps ----- level 3

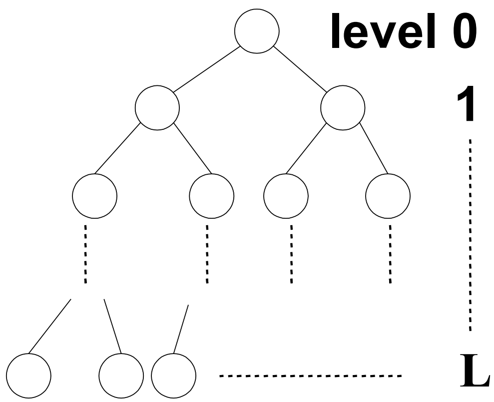


Let L be the max level (L = h-1)

level i -------- L - i swaps

Analysis of Heap Construction

Number of swaps



At level i the number of swaps is

$\leq$ $L - i$ for each node

At level i there are $\leq 2^i$ nodes

Total: $\leq \sum_{i=0}^{L} (L - i) \cdot 2^i$

Calculating $O(\Sigma(L - i)\cdot 2^i)$

Let $j = L-i$, then $i = L-j$ and

$$\sum_{i=0}^{L}(L - i)\cdot 2^i = \sum_{j=0}^{L} j\, 2^{L-j} = 2^L \sum_{j=0}^{L} j\, 2^{-j}$$

Consider $\Sigma j\cdot 2^{-j}$:

$$\Sigma\, j\cdot 2^{-j} = 1/2 + 2\ 1/4 + 3\ 1/8 + 4\ 1/16 + \cdots$$

$$\begin{array}{lllllll} = 1/2 + 1/4 & + & 1/8 + & 1/16 + \cdots & <= & 1 \\ + \quad\quad 1/4 & + & 1/8 + & 1/16 + \cdots & <= & 1/2 \\ + & + & 1/8 + & 1/16 + \cdots & <= & 1/4 \\ \hline \end{array}$$

$$\Sigma j\cdot 2^{-j} \quad <= \quad 2$$

So $2^L\, \Sigma j\, 2^{-j} <= 2\cdot 2^L = 2n$ where L is $O(\log n)$

$$2^L \sum_{j=1}^{L} j/2^j \quad \le \quad 2^{L+1} \quad = 2n$$

Where L is O(log n)

O(n)

So, the number of swaps is $\le$ O(n)

Review

- Geometric Sum : $f(i) = a^i$
- The geometric progressions have an exponential growth

$$S = \sum_{i=0}^{n} r^i = 1 + r + r^2 + \ldots + r^n$$

$$rS = \qquad r + r^2 + \ldots + r^n + r^{n+1}$$

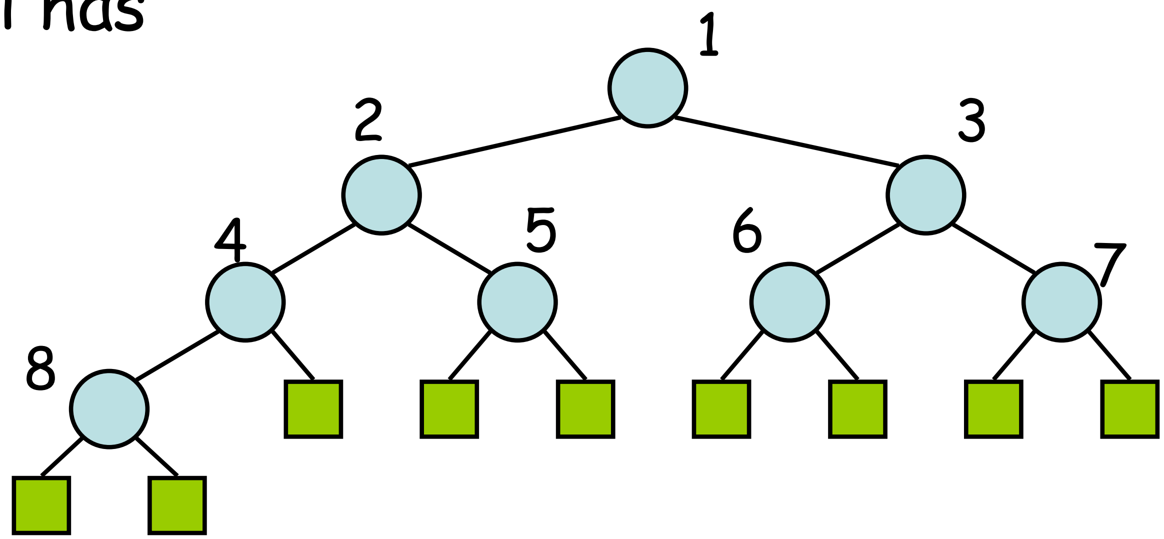
$$rS - S = (r-1)S = r^{n+1} - 1$$

$$S = (r^{n+1}-1)/(r-1)$$

$$\text{If } r=2,\ S = (2^{n+1}-1)$$

Implementing a Heap with an Array

A heap can be nicely represented by a vector (array-based), where the node at rank i has



- left child at rank 2i

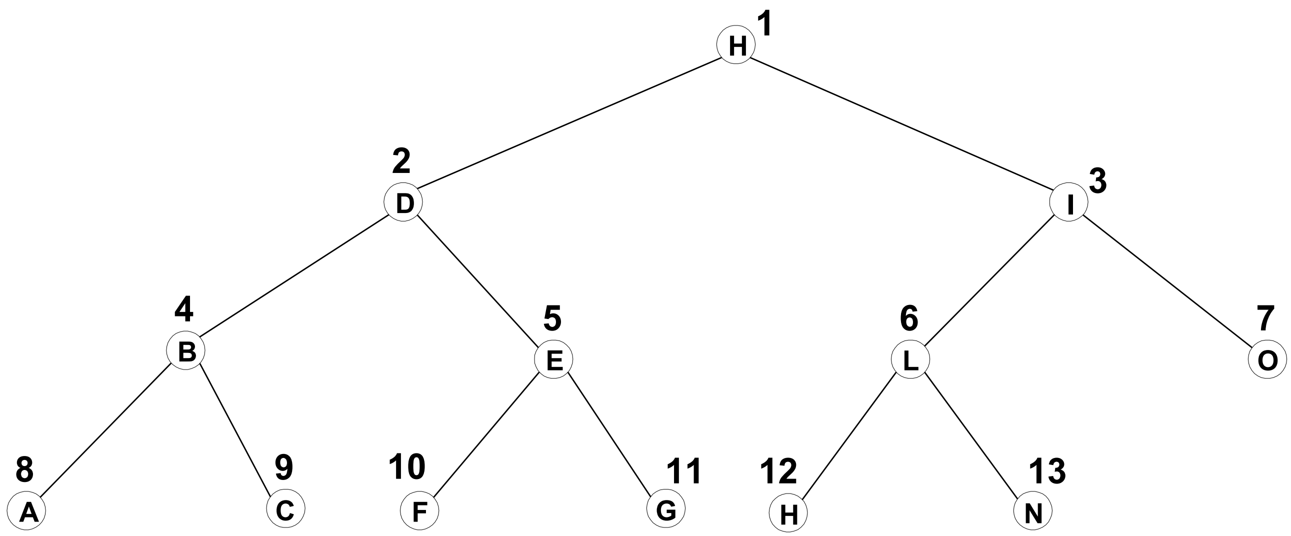
and

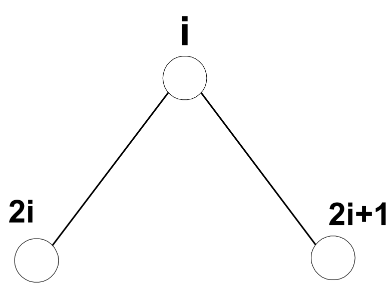
- right child at rank 2i + 1

1	2	3	4	5	6	7	8

The leaves do no need to be explicitly stored

Example

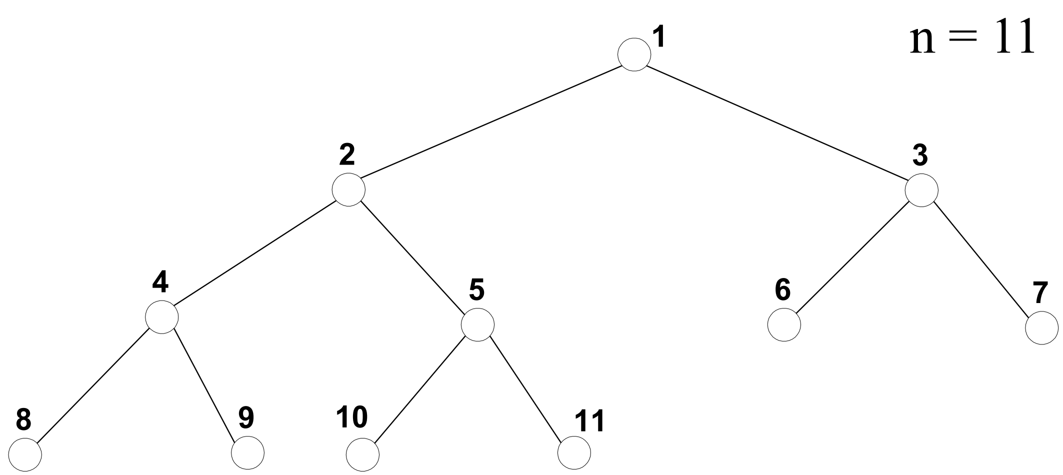




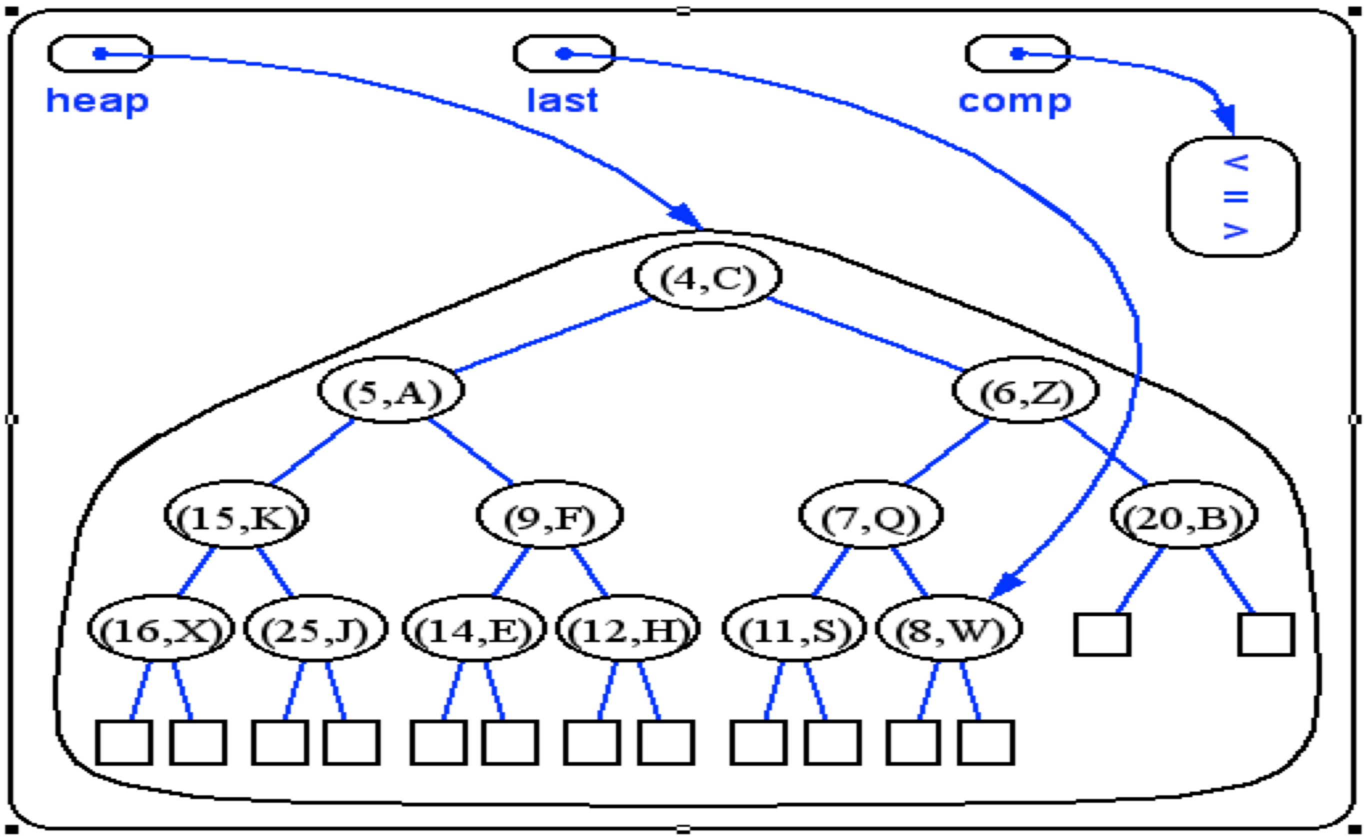
1	2	3	4	5	6	7	8	9	10	11	12	13
H	D	I	B	E	L	O	A	C	F	G	H	N

Reminder .....

Left child of T[i]	T[2i]	if	$2i \leq n$
Right child of T[i]	T[2i+1]	if	$2i + 1 \leq n$
Parent of T[i]	T[i div 2]	if	$i > 1$
The Root	T[1]	if	$T \neq 0$
Leaf? T[i]	TRUE	if	$2i > n$



Implementation of a Priority Queue with a Heap



(upheap)

insertItem	O(log n)
minKey, minElement	O(1)
removeMin	O(log n)

(remove root + downheap)

Application: Sorting
Heap Sort

Construct initial heap   O(n)

n times

remove root	O(1)
re-arrange	O(log n)
remove root	O(1)
re-arrange	O(log (n-1))
…	⋮
…	

When there are i nodes left in the PQ: $\lfloor \log i \rfloor$

$$\rightarrow \text{TOT} = \sum_{i=1}^{n} \lfloor \log i \rfloor$$

$$= (n + 1)q - 2^{q+1} + 2 \text{ where } q = \lfloor \log (n+1) \rfloor$$

$\Rightarrow$ $O(n \log n)$ $\Rightarrow$ The heap-sort algorithm sorts a sequence S of n elements in $O(n \log n)$ time

Remember it was the $O(n^2)$ running time of selection-sort and insertion-sort

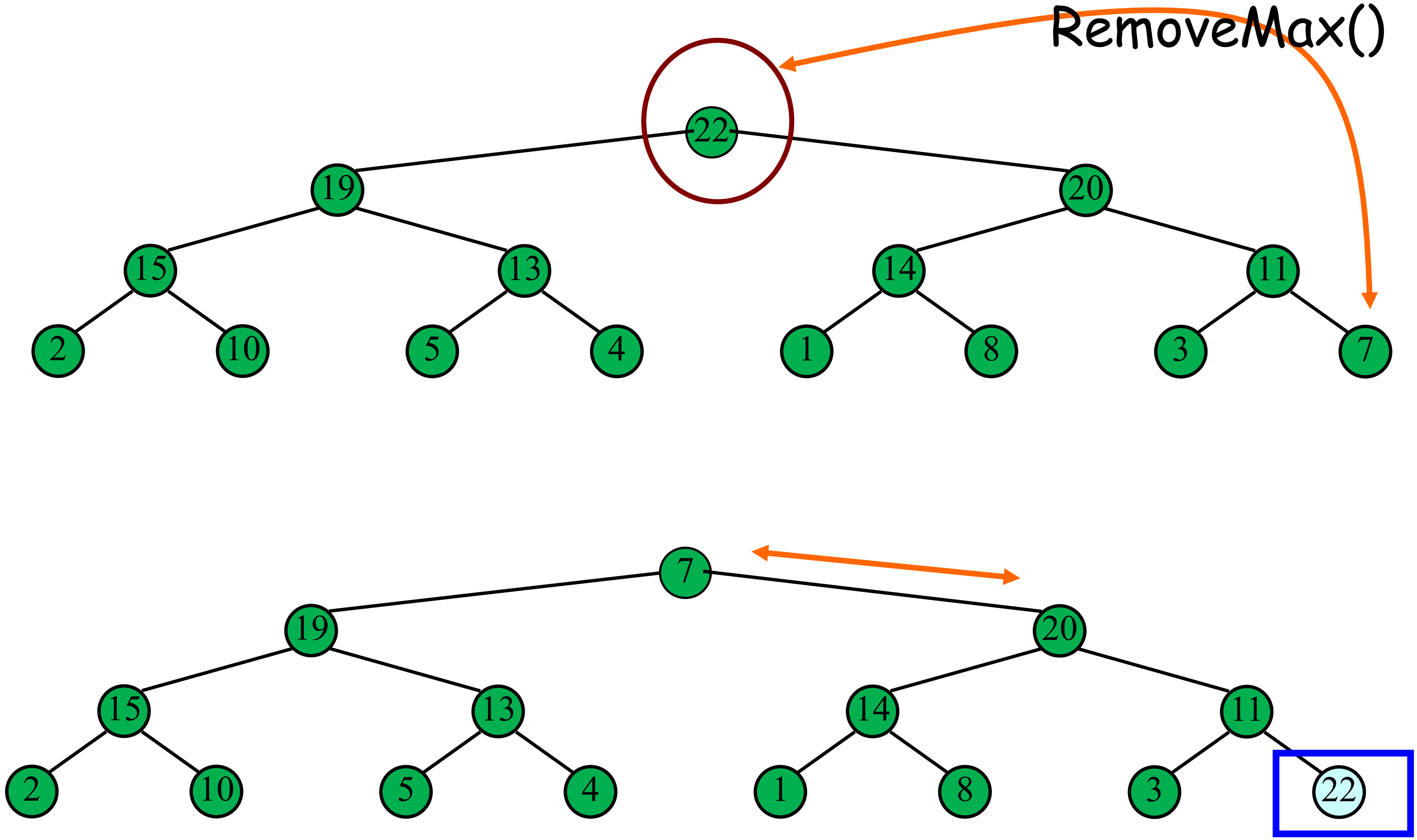
Heap Sort animation

- https://www.youtube.com/watch?v=MtQL_ll5KhQ

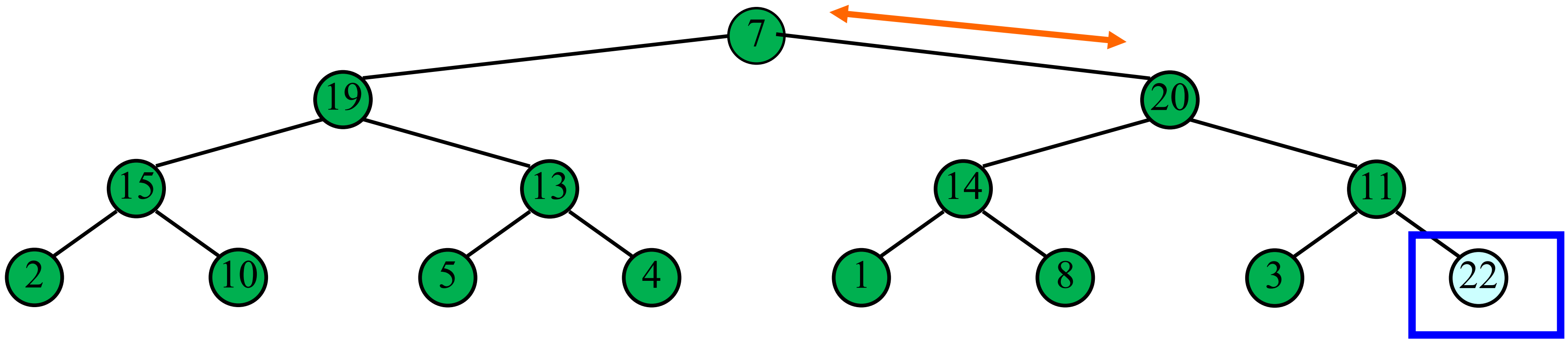
HeapSort in Place

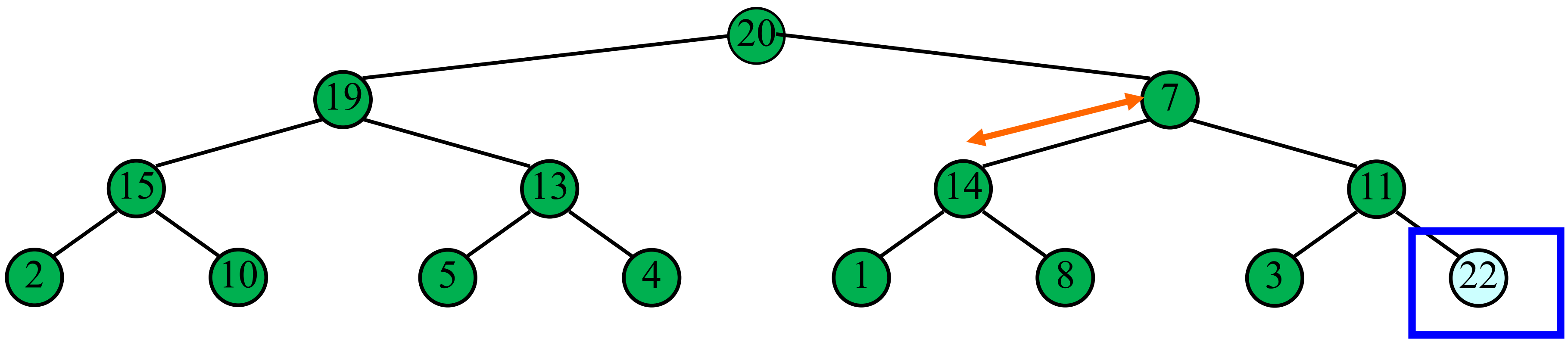
Instead of using a secondary data structure P to sort a sequence S,
We can execute heapsort « in place » by dividing S in two parts, one representing the heap, and the other representing the sequence.
The algorithm is executed in two phases:

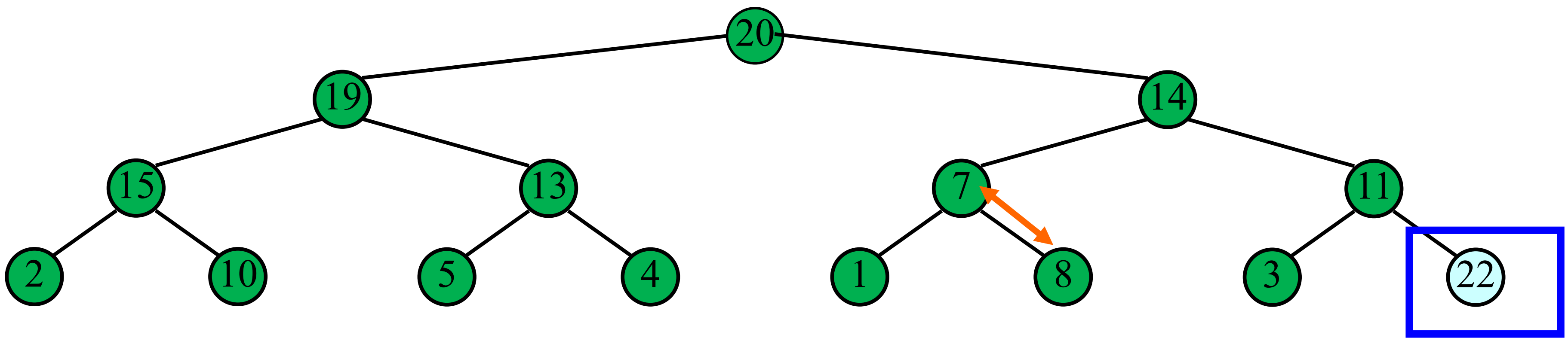
- ✓ Phase 1: We build a max-heap so to occupy the whole structure.
- ✓ Phase 2: We start with the part « sequence » empty and we grow it by removing at each step i (i=1..n) the max value from the heap and by adding it to the part « sequence », always maintaining the heap properties for the part « heap ».

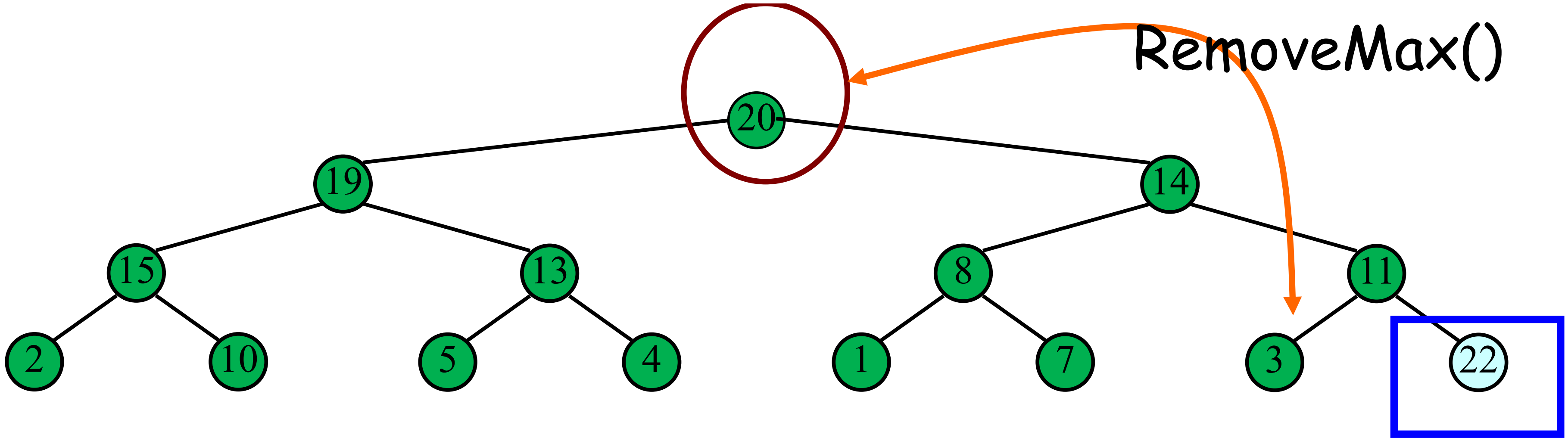


Not a heap anymore !


7
19
20
15
13
14
11
2
10
5
4
1
8
3
22

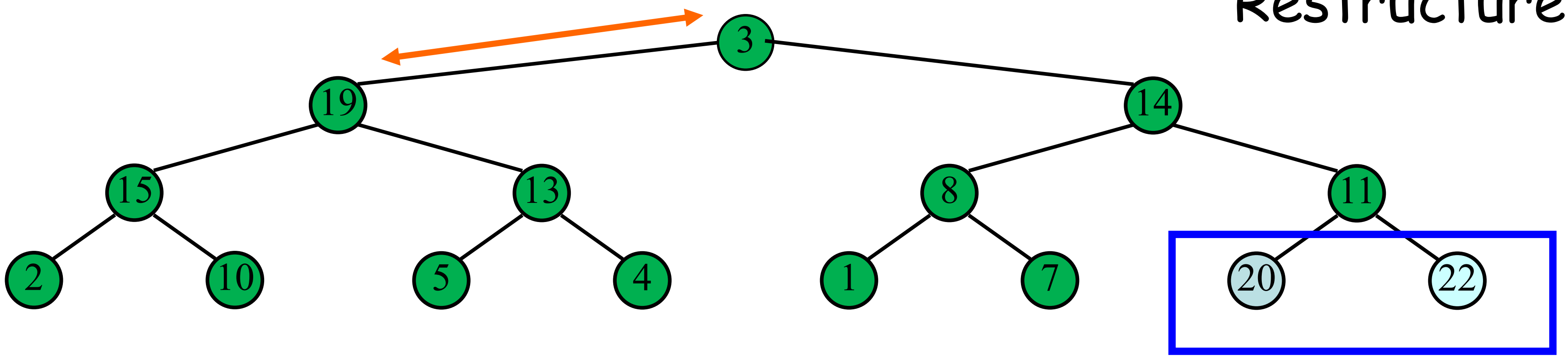

20
19
7
15
13
14
11
2
10
5
4
1
8
3
22


20
19
14
15
13
7
11
2
10
5
4
1
8
3
22



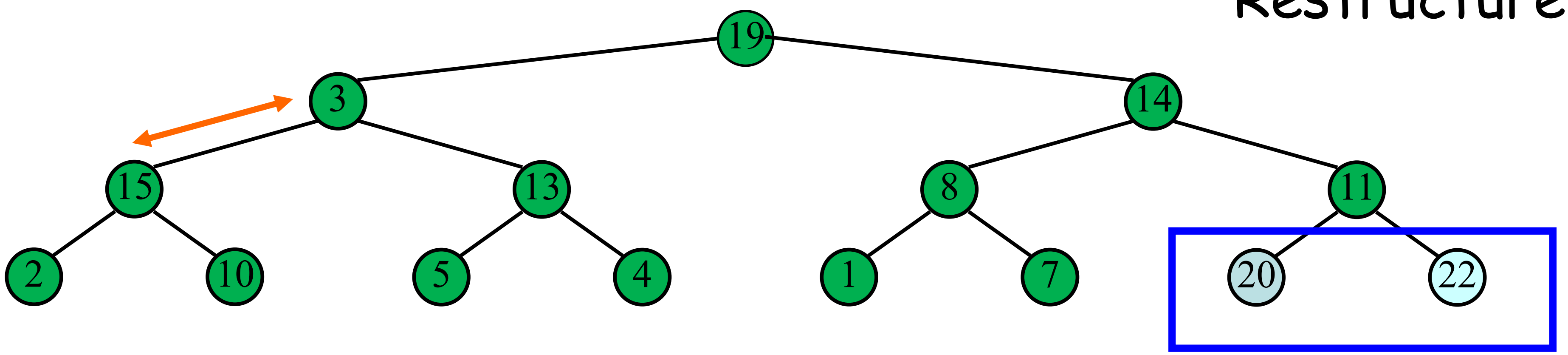
Now the part heap is smaller, the part Sequence contains a single element.

Restructure



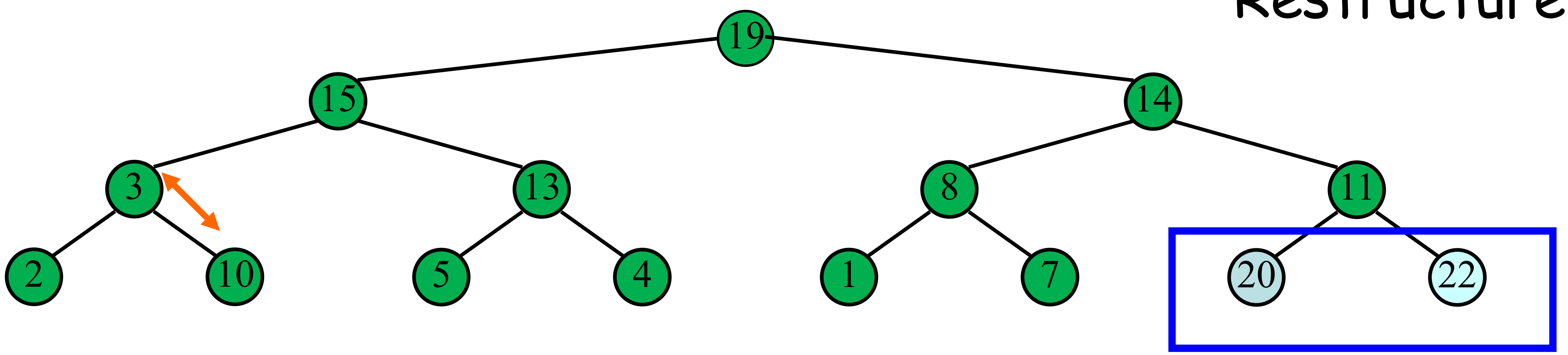
Not a heap anymore !

Restructure

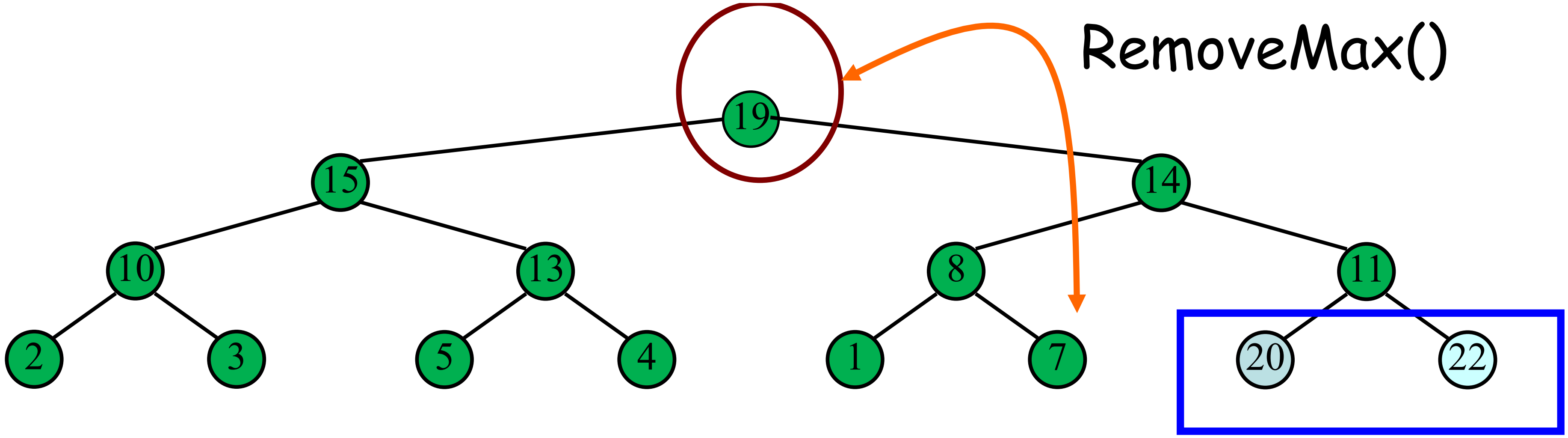


Not a heap anymore !

Restructure

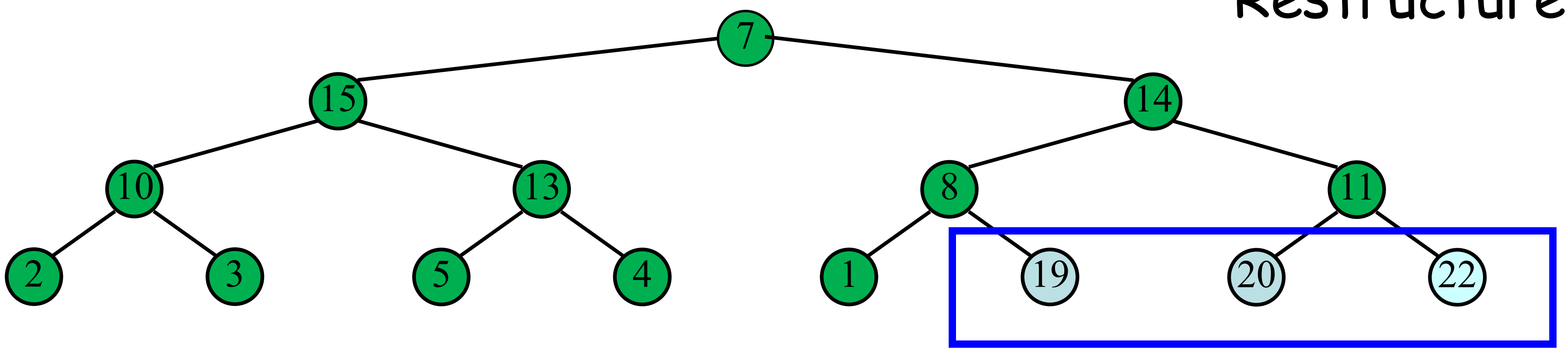


Not a heap anymore !

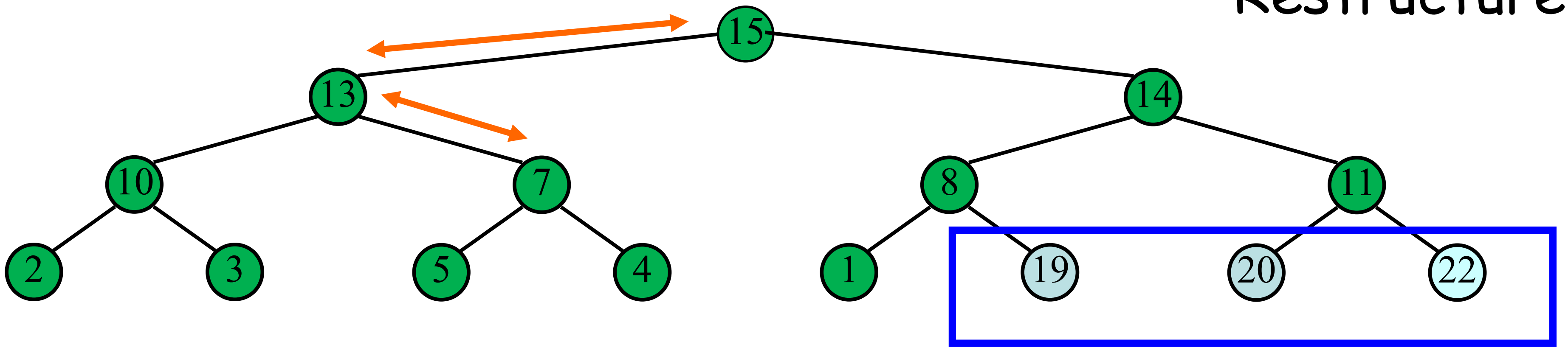


Now it is a heap again !

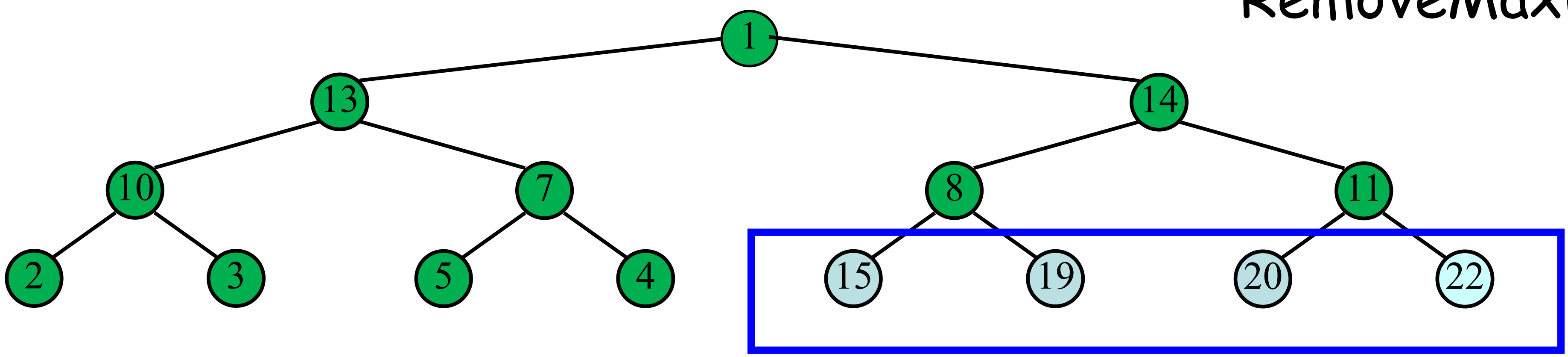
Restructure



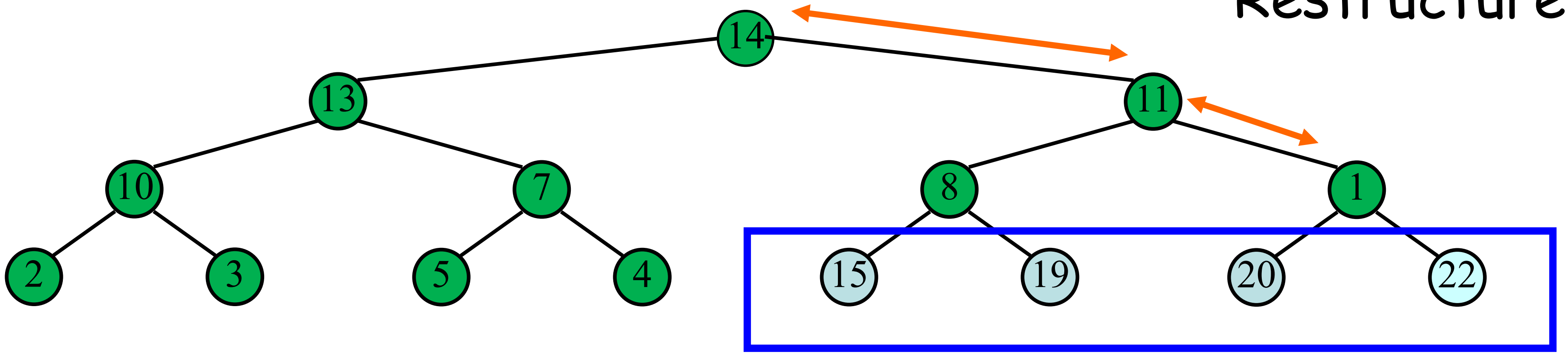
Restructure



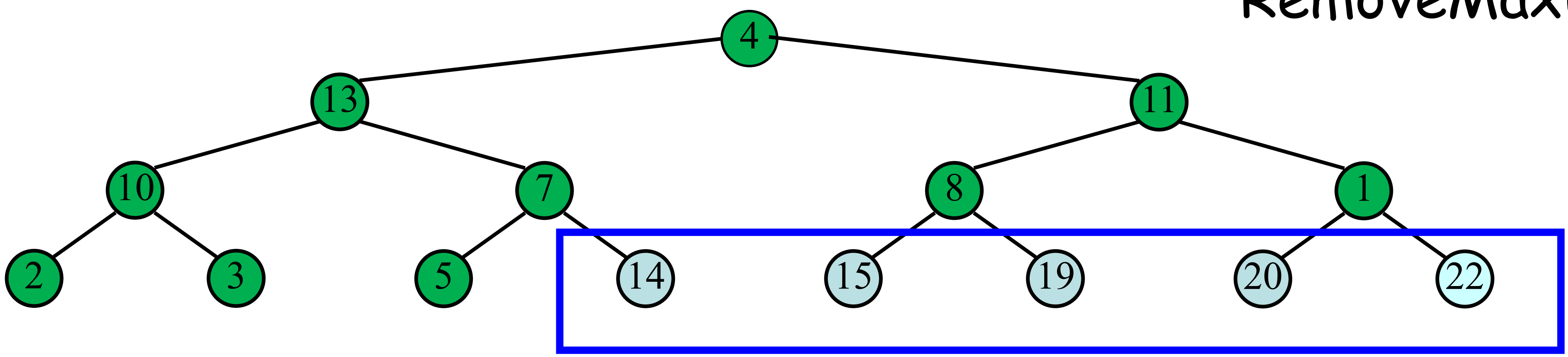
RemoveMax()



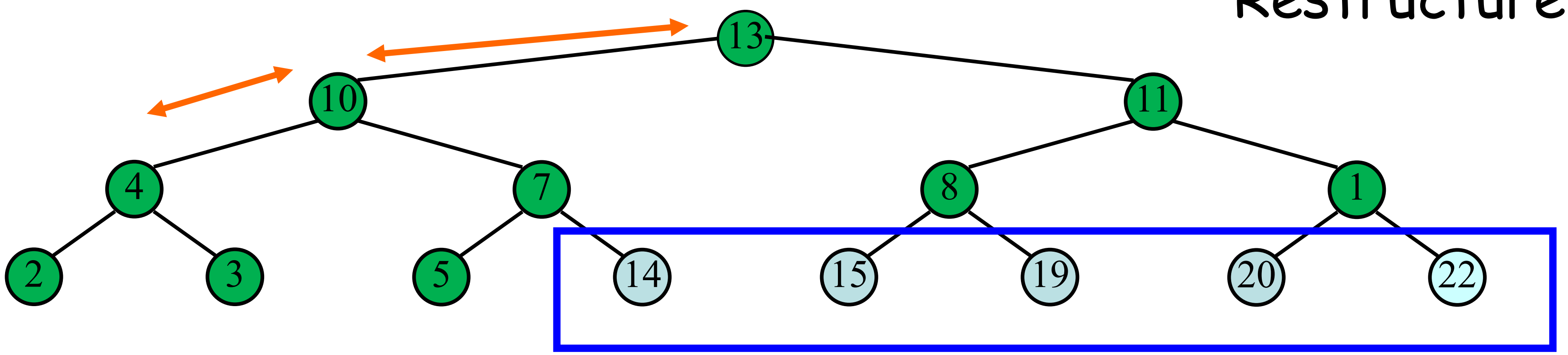
Restructure

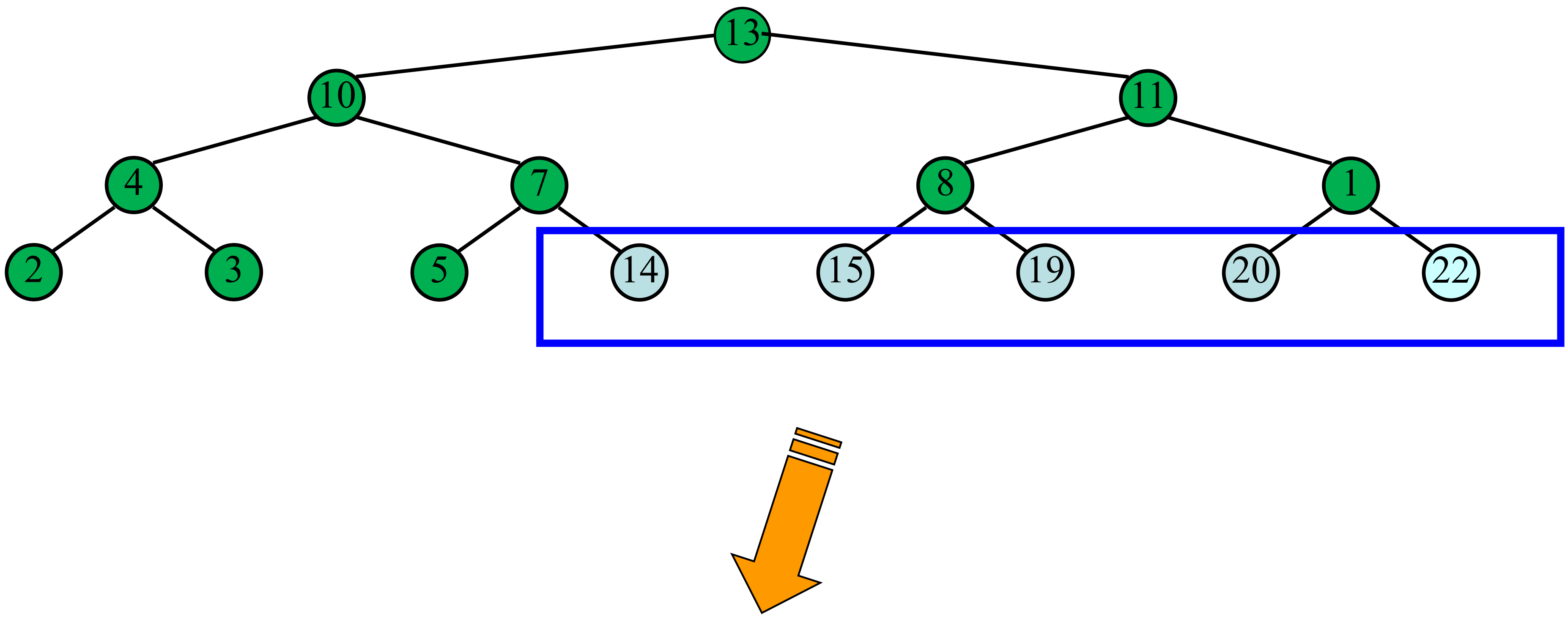


RemoveMax()



Restructure





Heap part: unsorted

Sequence part: sorted

	13	10	11	4	7	8	1	2	3	5	14	15	19	20	22

Pseudocode for in-place HEAPSORT
(based on wikipedia pseudocode)

```
procedure heapsort(A,n) {
  input: an unordered array A of length n

  heapify(A,n)  // in O(n) with bottom-up heap construction
                 // or in O(n log n) with n heap insertions

 // Loop Invariant: A[0:end] is a heap; A[end+1:n-1] is sorted
   end ← n - 1
   while end > 0 do
       swap(A[end], A[0])
       end ← end - 1
       downHeap(A, 0, end)
}
```

```
Procedure downHeap(A, start, end) {
    root ← start
    while root * 2 + 1 ≤ end do    (While the root has at least one child)
        child ← root * 2 + 1       (Left child)
        swap ← root                (Keeps track of child to swap with)
        if A[swap] < A[child]
            swap ← child
        (If there is a right child and that child is greater)
        if child+1 ≤ end and A[swap] < A[child+1]
            swap ← child + 1
        if swap = root
            (case in which we are done with downHeap)
            return
        else
            swap(A[root], A[swap])
            root ← swap  (repeat to continue downHeap the child now)
}
```

```
procedure heapify(A, n)
    (start is assigned the index in ‘A' of the last parent node)
    (the last element in a 0-based array is at index n-1;
    find the parent of that element)
    start ← floor ((n - 2) / 2)

    while start ≥ 0 do
        (downHeap the node at index 'start' to the proper place)
        downHeap(A, start, n - 1)
        (go to the next parent node)
        start ← start - 1

// after this loop array A is a heap
```