Heaps

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

Heaps (Min-heap)

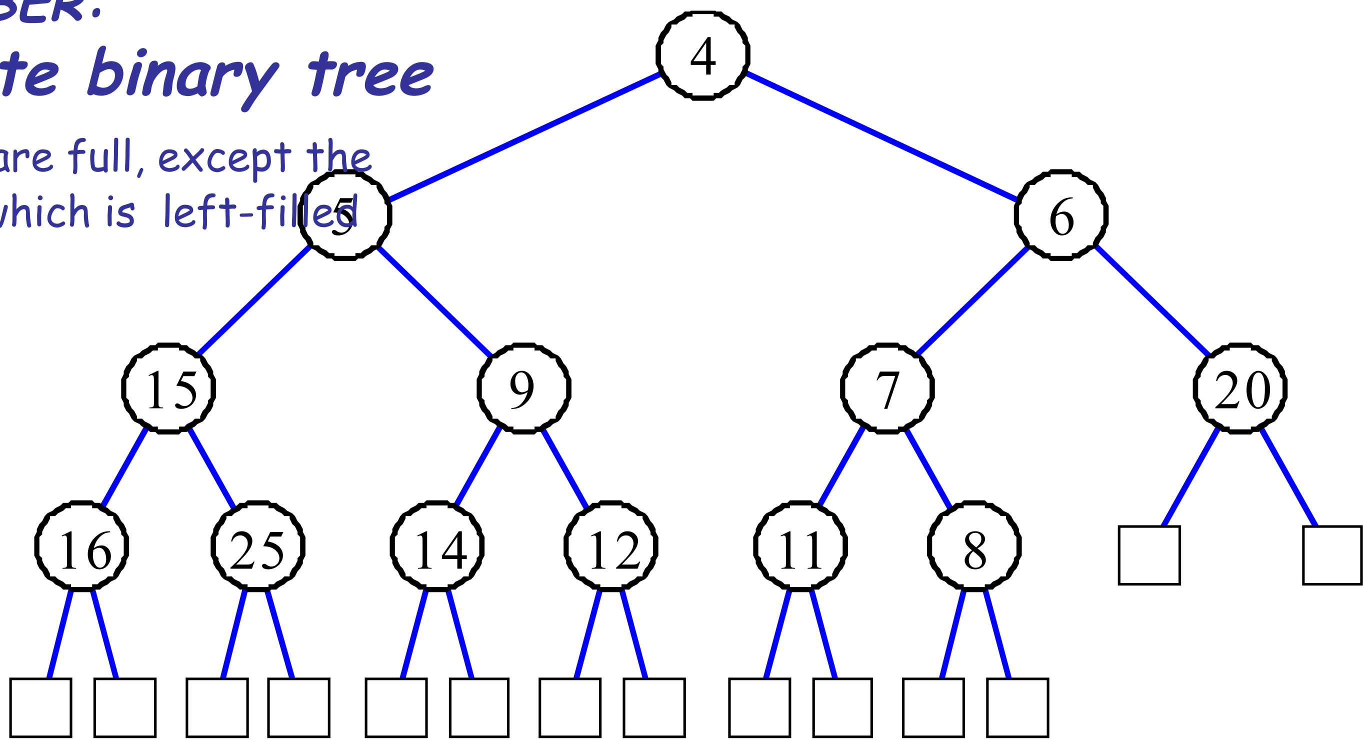
Complete binary tree that stores a collection of keys (or key-element pairs) at its internal nodes and that satisfies the additional property:

key(parent) ≤ key(child)

REMEMBER:

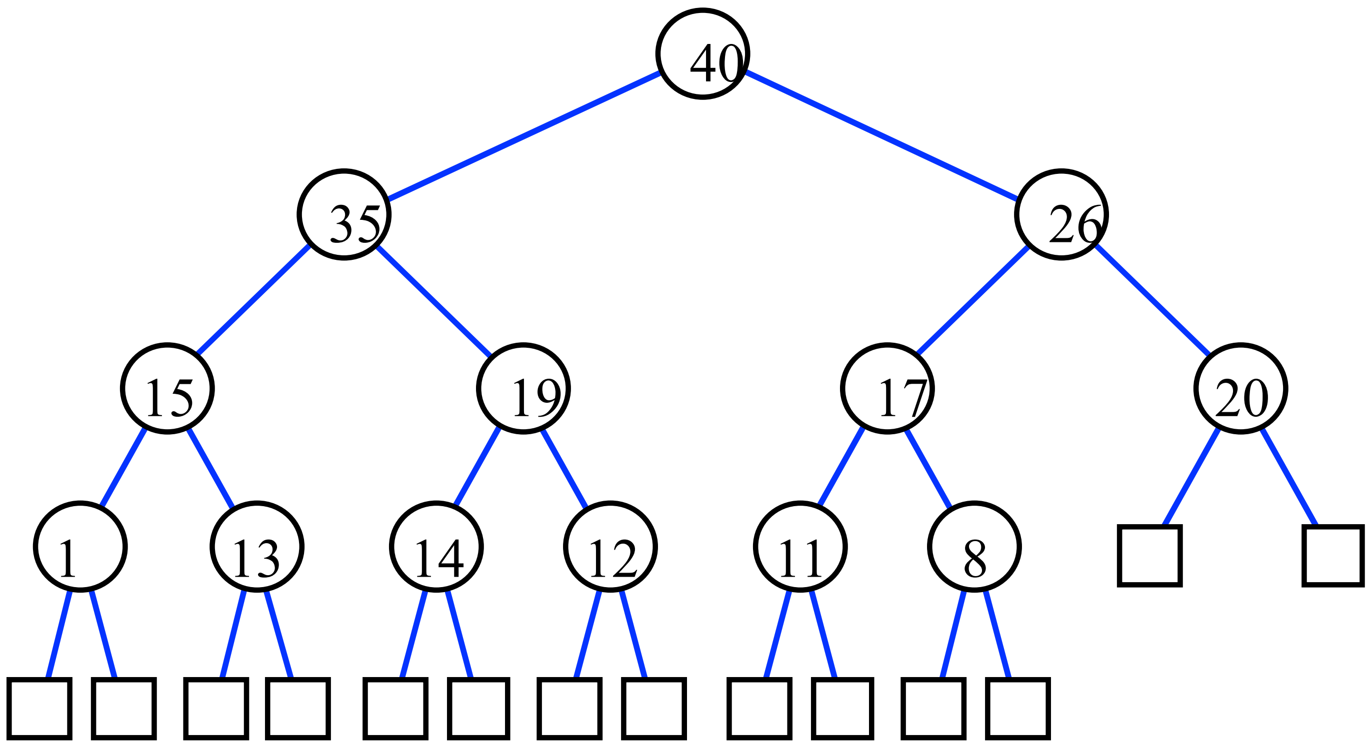
complete binary tree

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

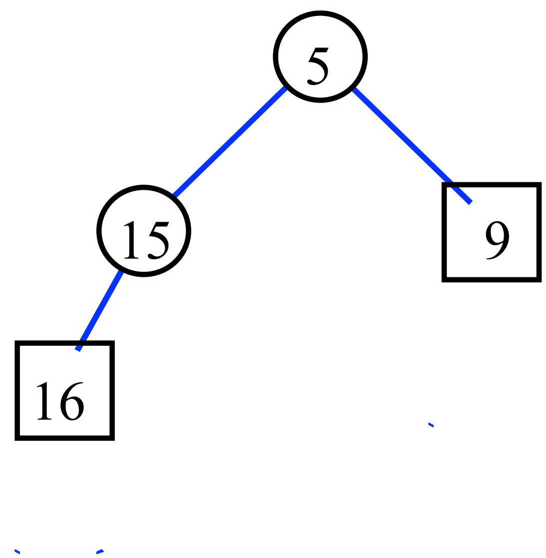


Max-heap

key(parent) $\geq$ key(child)



We do not need to add dummy leaves.

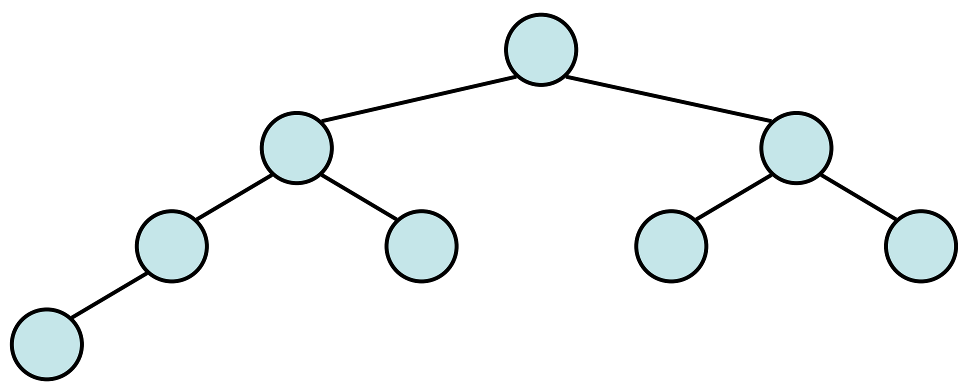


Height of a Heap

- 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 $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$

depth	keys
0	1
1	2
$h-1$	2^{h-1}
h	at least 1



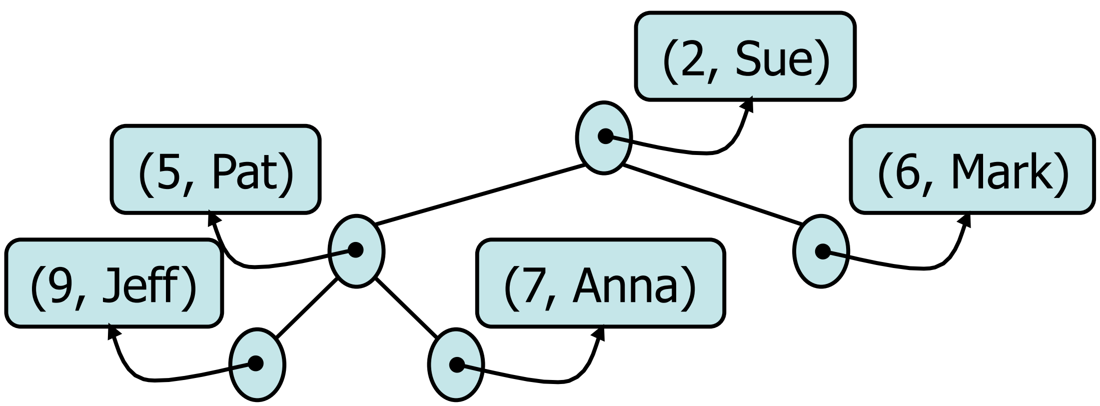
Notice that ….

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

removeMin():

→ Remove the root

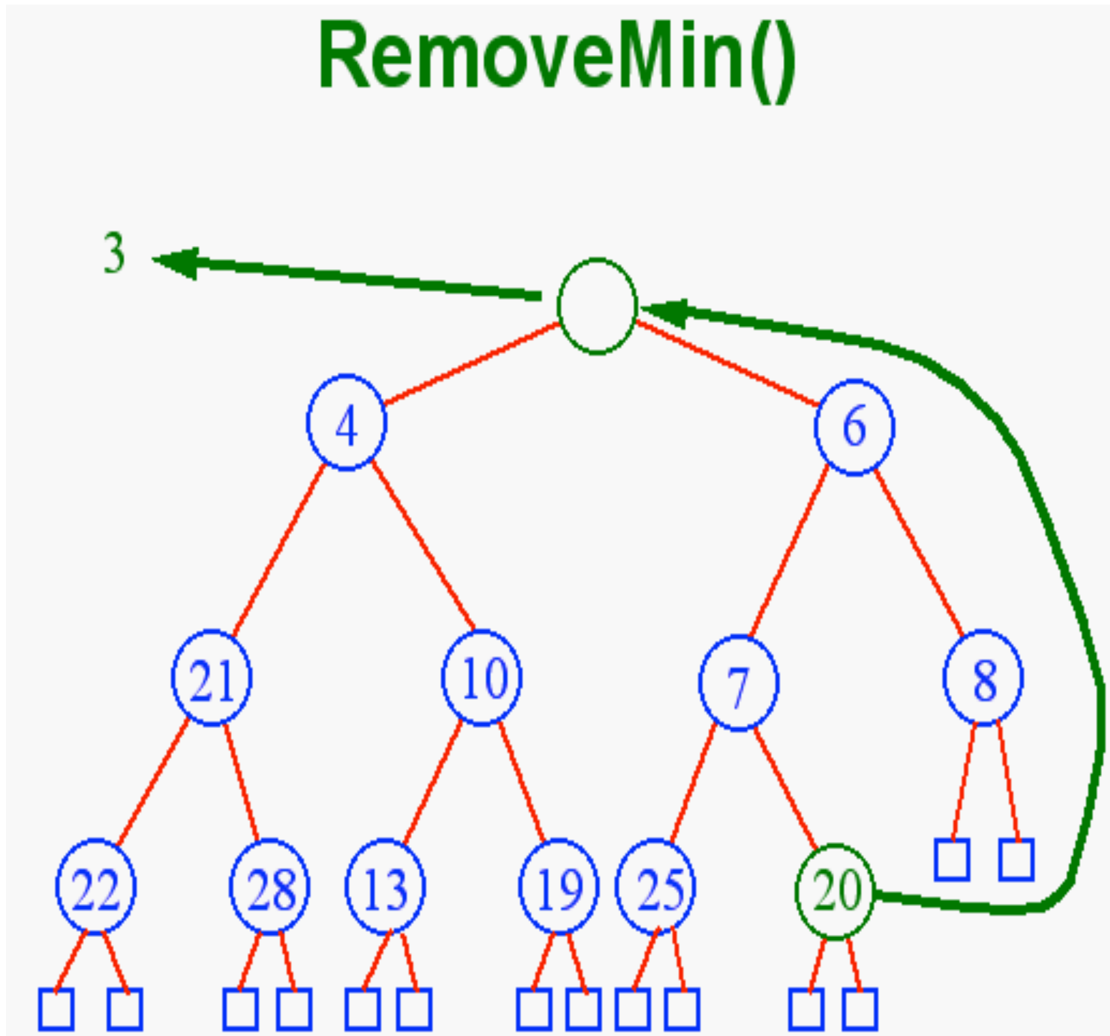
→ Re-arrange the heap!



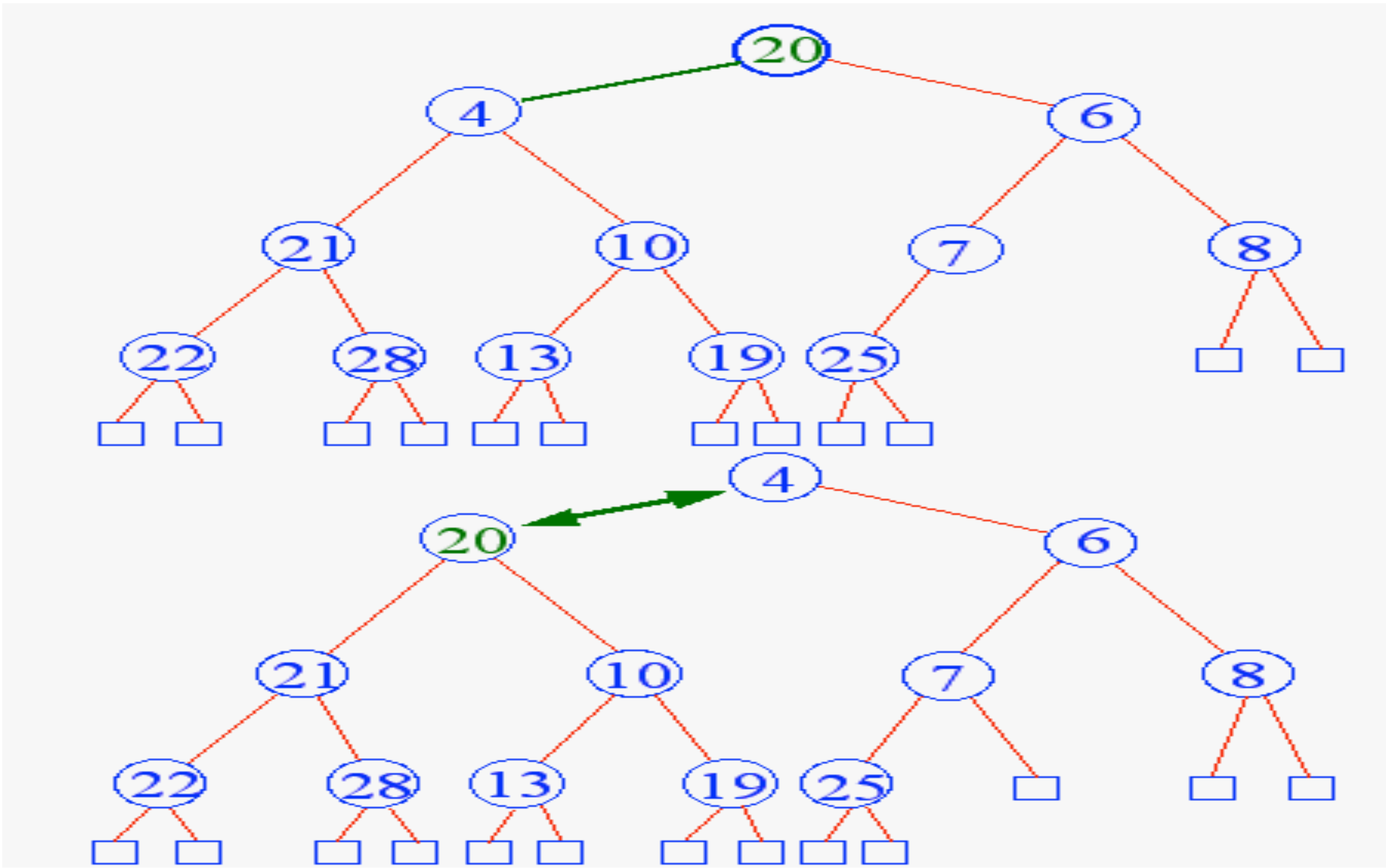
Removal From a Heap

- 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 …

Note example uses dummy leaves (this is optional)

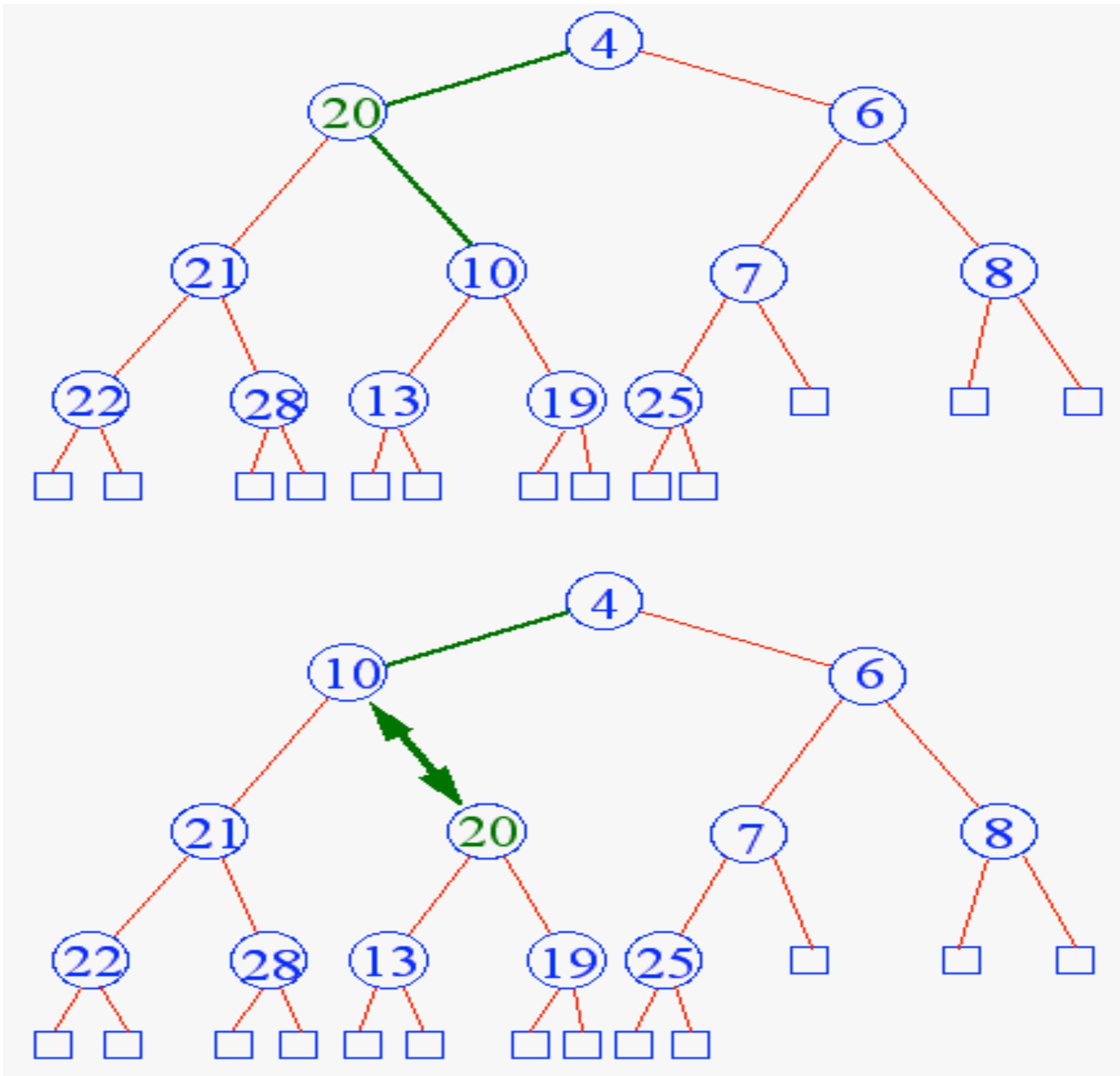


Downheap

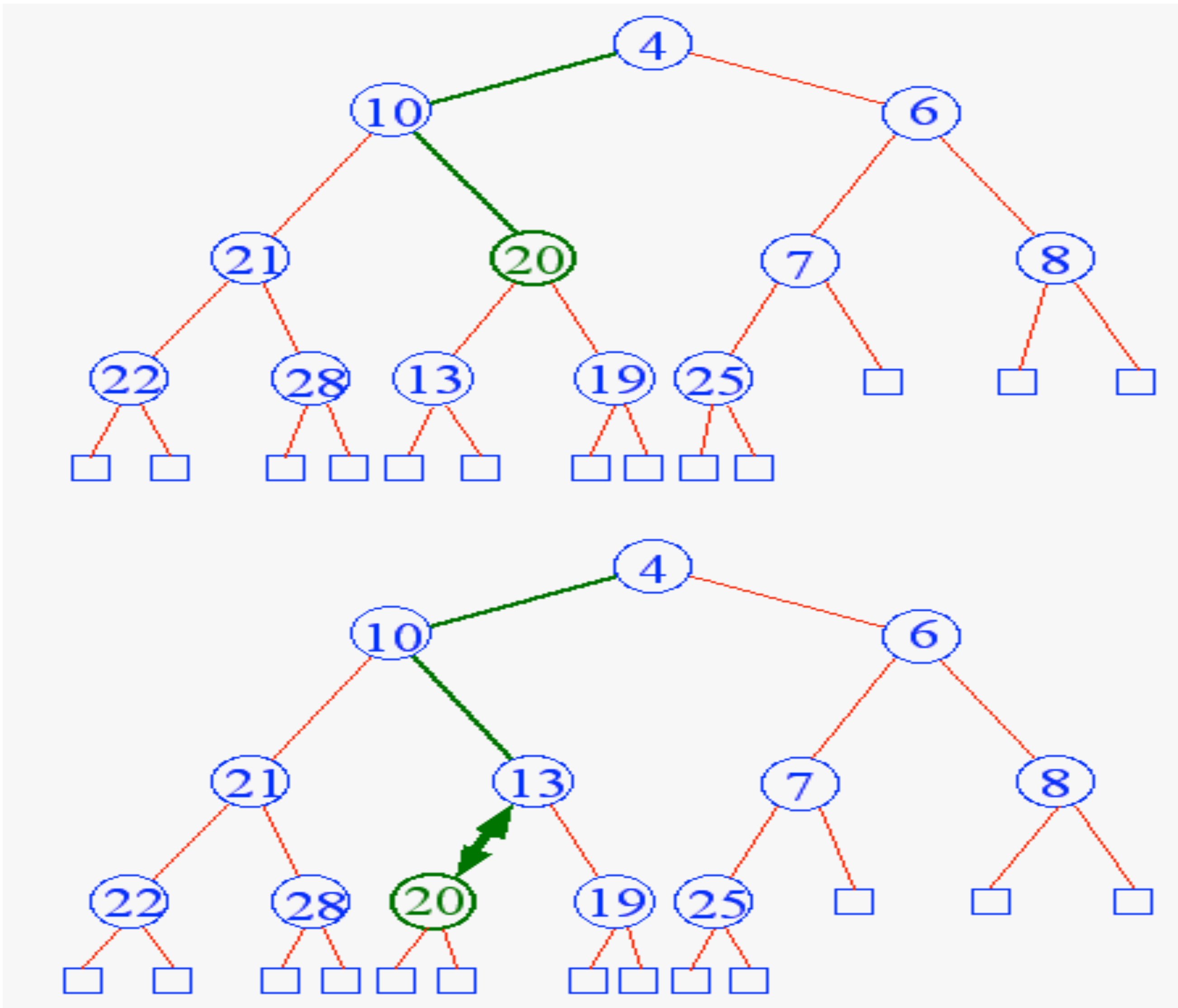


- Downheap compares the parent with the smallest child. If the child is smaller, it switches the two.

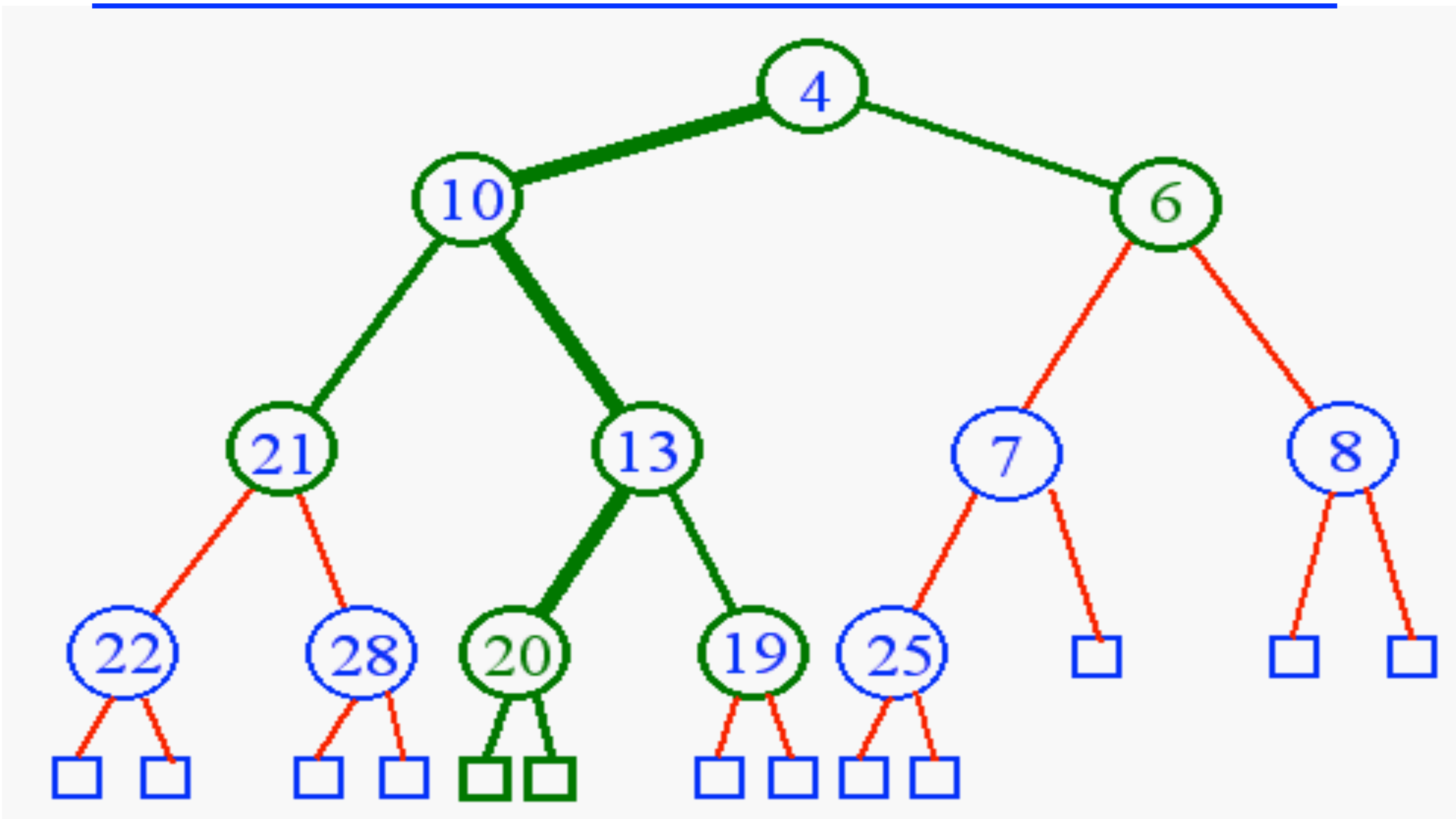
Downheap Continues



Downheap Continues



End of Downheap



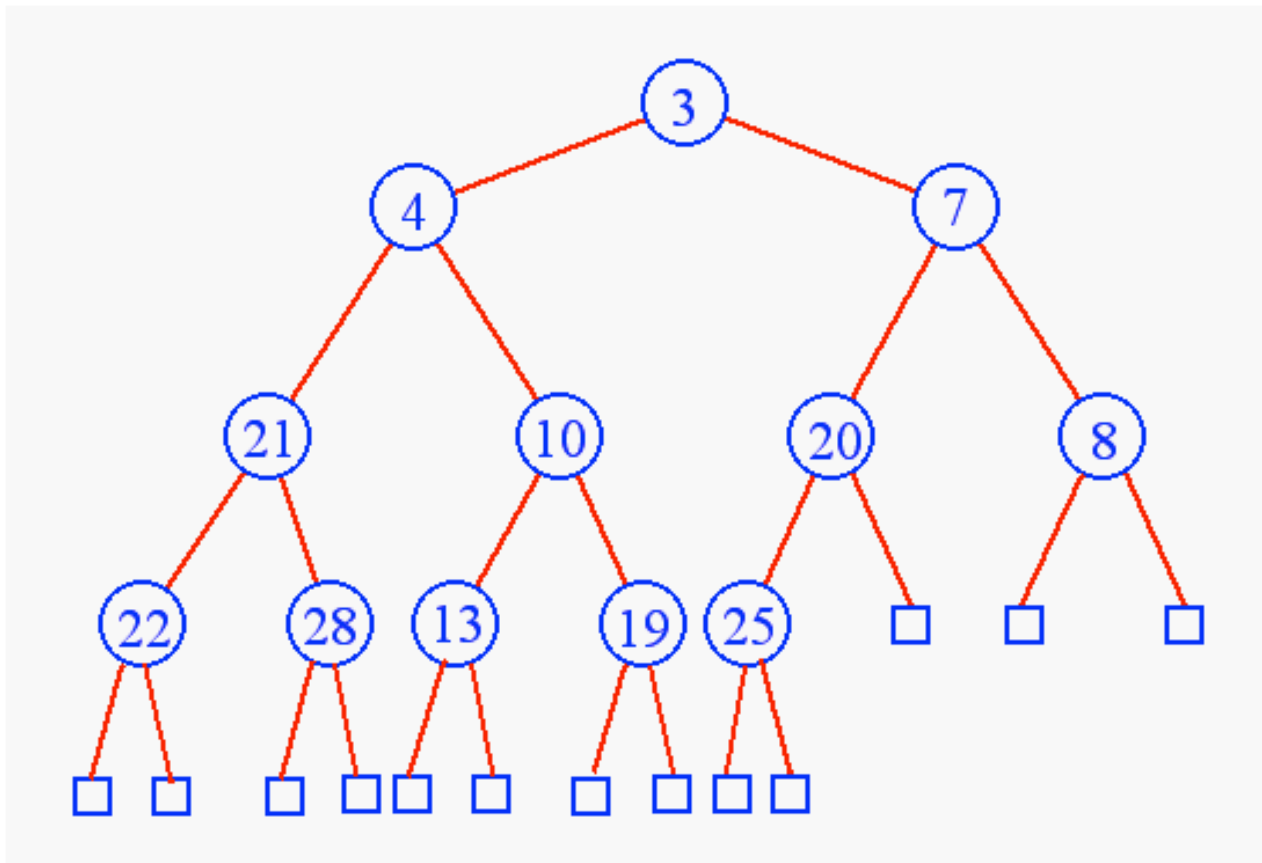
- Downheap terminates when the key is greater than the keys of both its children or the bottom of the heap is reached.

(total #swaps) $\leq$ (h - 1), which is **O(log n)**

(note: h swaps if not using dummy leaves)

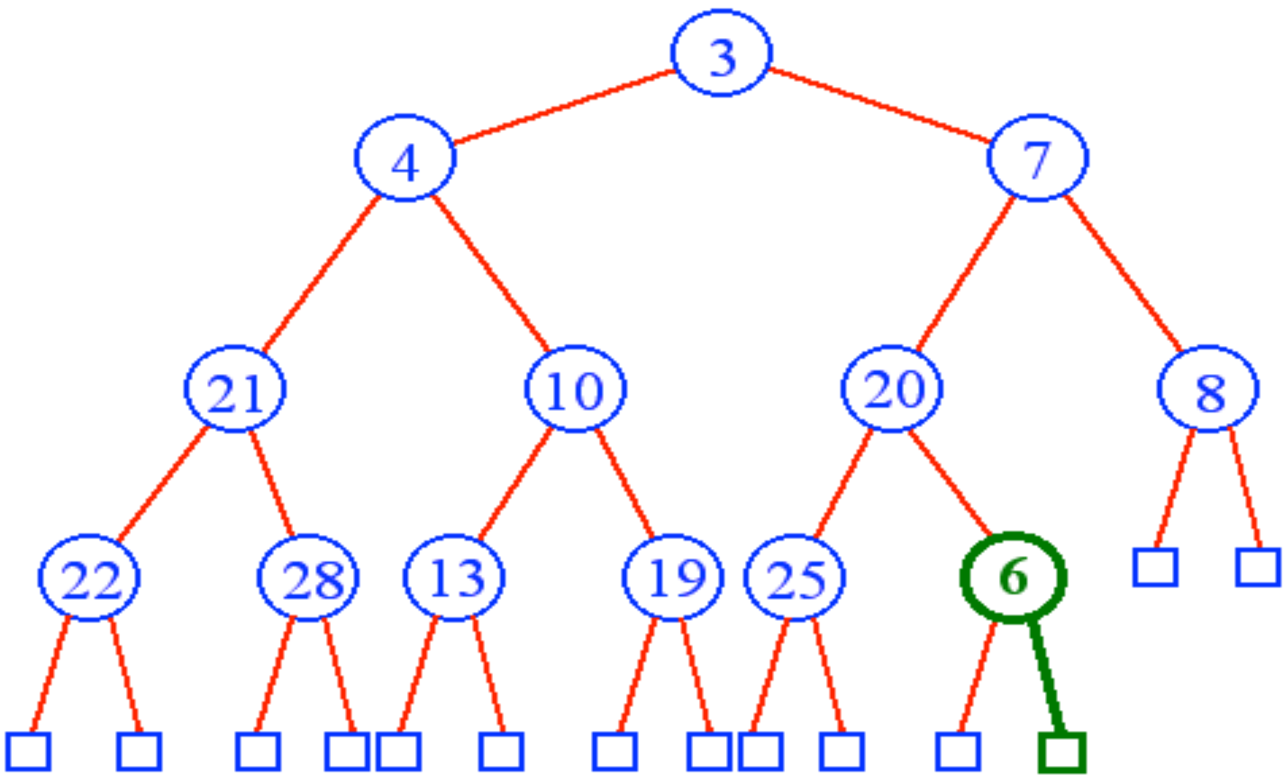
Heap Insertion

The key to insert is 6



Heap Insertion

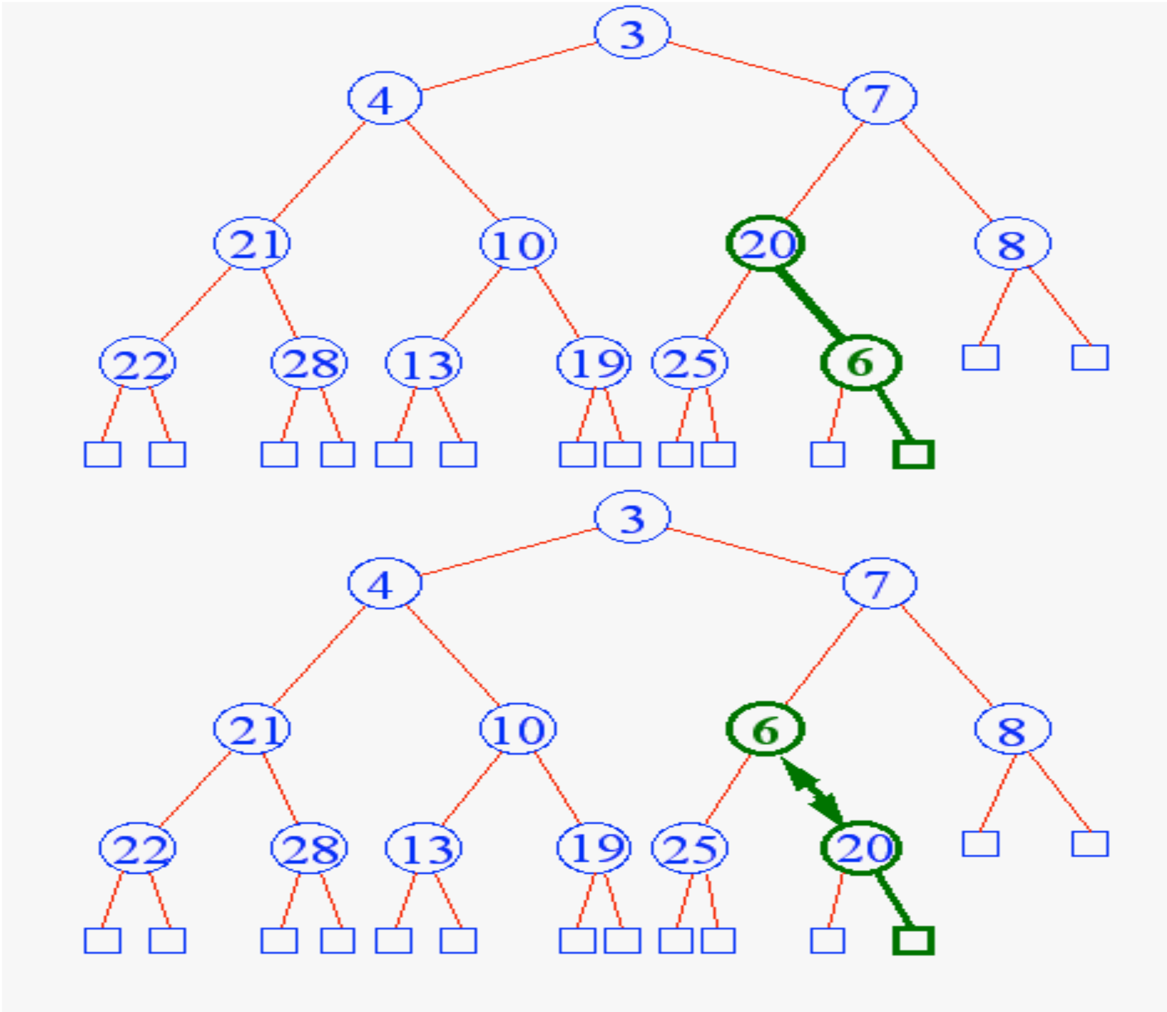
Add the key in the ***next available position*** in the heap.



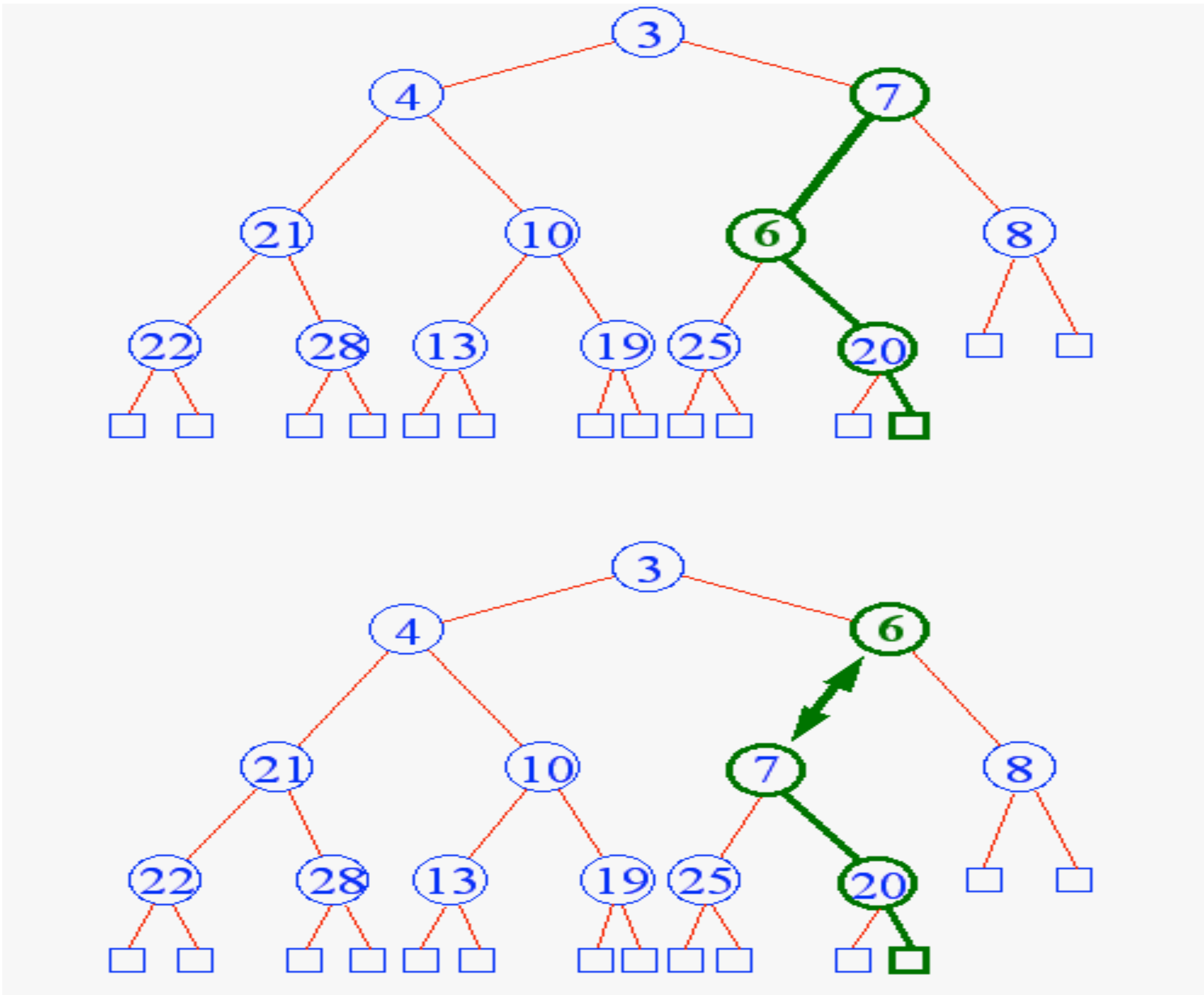
Now begin ***Upheap***.

Upheap

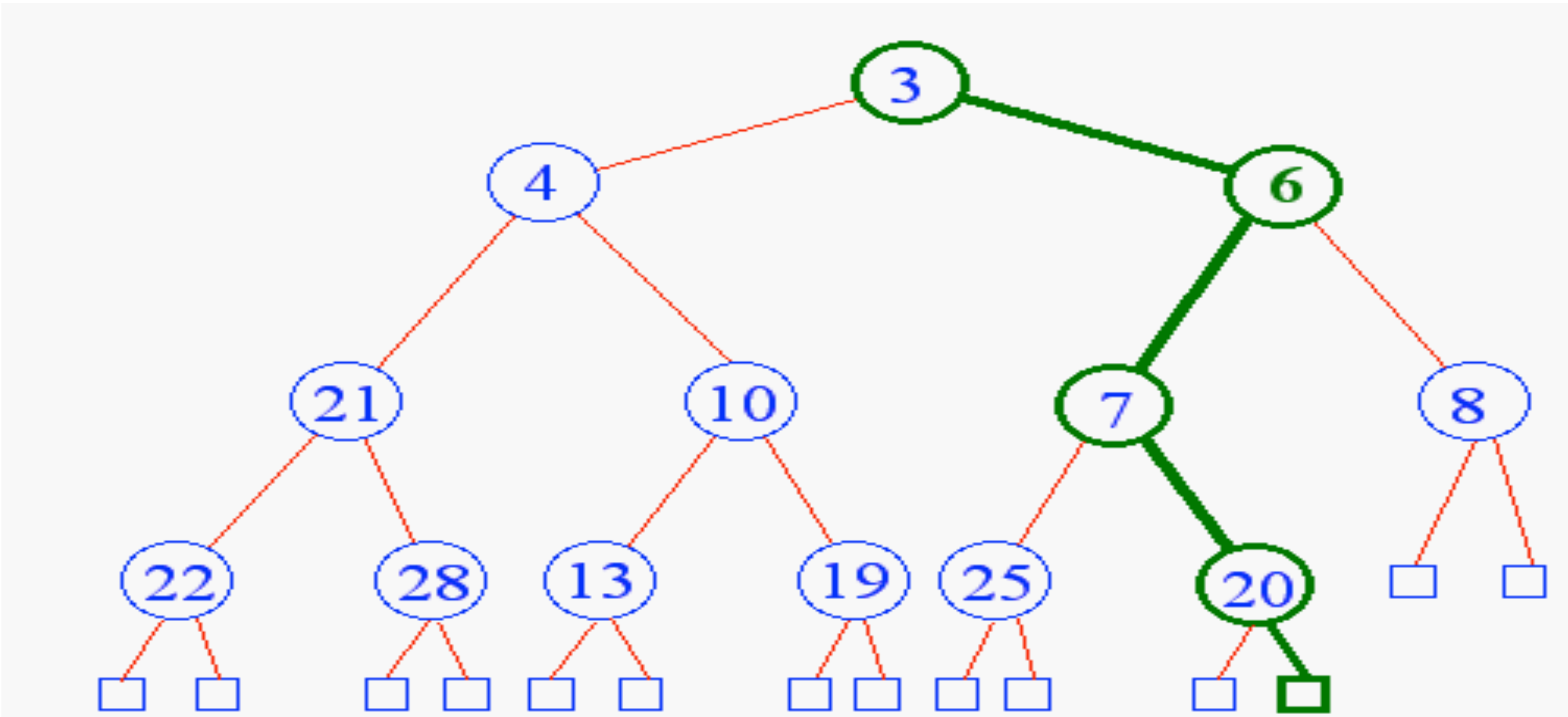
- Swap parent-child keys out of order



Upheap Continues



End of Upheap



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

removeMin()

```
public Entry<K,V> removeMin( ) {
  if (heap.isEmpty( )) return null;
  Entry<K,V> answer = heap.get(0);
  swap(0, heap.size( ) − 1);
  heap.remove(heap.size( ) − 1);
  downheap(0);
  return answer;
}
```

```
protected void downheap(int j) {
  while (hasLeft(j)) {                          // continue to bottom (or brea
    int leftIndex = left(j);
    int smallChildIndex = leftIndex;                 // although right ma
    if (hasRight(j)) {
        int rightIndex = right(j);
        if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
          smallChildIndex = rightIndex;              // right child is small
    }
    if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
      break;                                         // heap property has
    swap(j, smallChildIndex);
    j = smallChildIndex;                             // continue at positic
  }
}
```


Excerpt from the texbook Java code pages 378-379.

Insert(key, value)

```java
/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
  checkKey(key);        // auxiliary key-checking method (could throw exception)
  Entry<K,V> newest = new PQEntry<>(key, value);
  heap.add(newest);                                   // add to the end of the list
  upheap(heap.size() - 1);                            // upheap newly added entry
  return newest;
}
```

```java
/** Moves the entry at index j higher, if necessary, to restore the heap property. */
protected void upheap(int j) {
  while (j > 0) {                 // continue until reaching root (or break statement)
    int p = parent(j);
    if (compare(heap.get(j), heap.get(p)) >= 0) break;   // heap property verified
    swap(j, p);
    j = p;                                        // continue from the parent's location
  }
}
```

Heap Construction

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

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

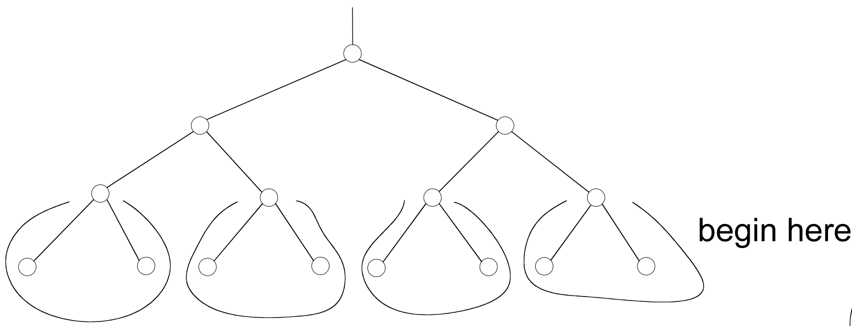
But we can do better ....

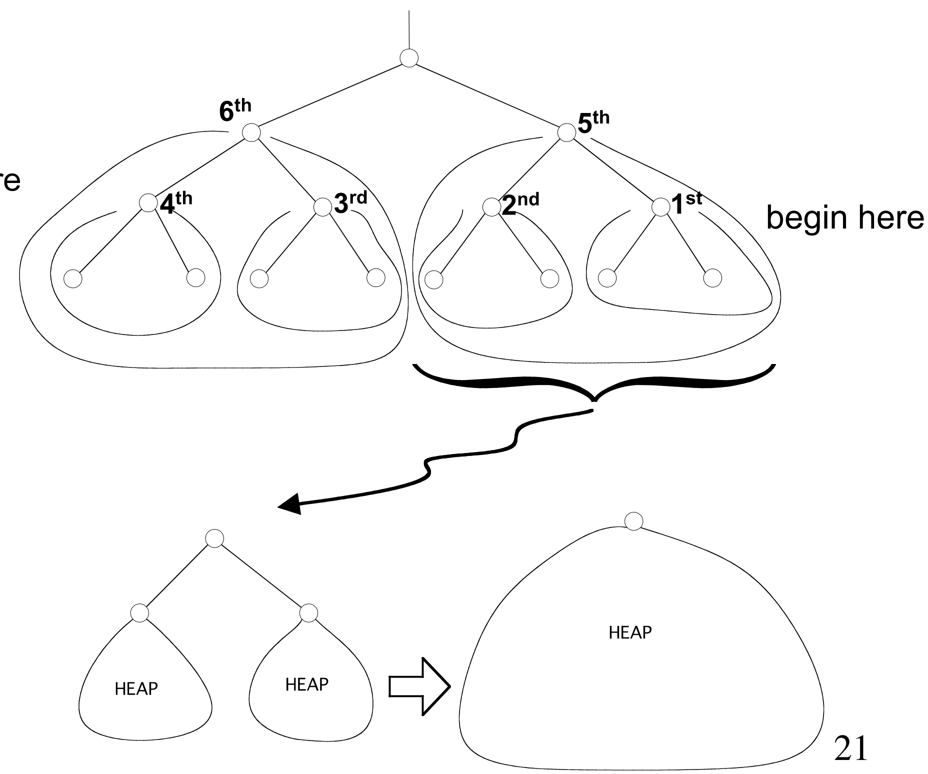
Bottom-up Heap Construction

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

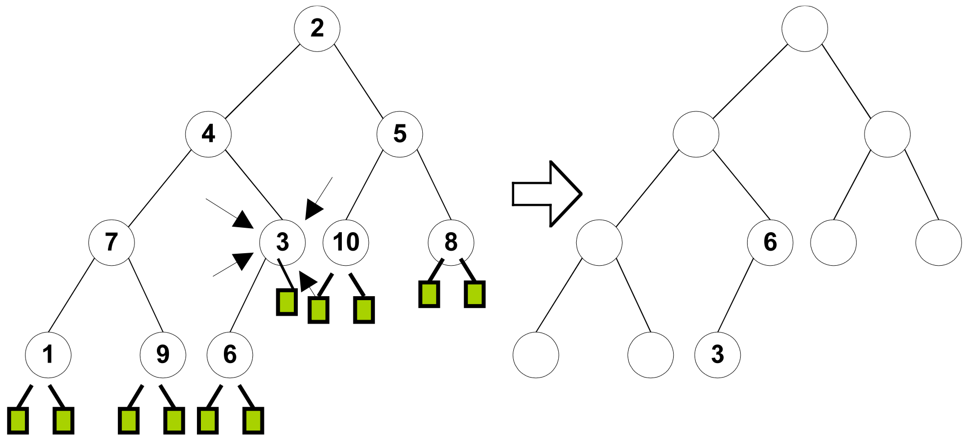
Construction of a Heap

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





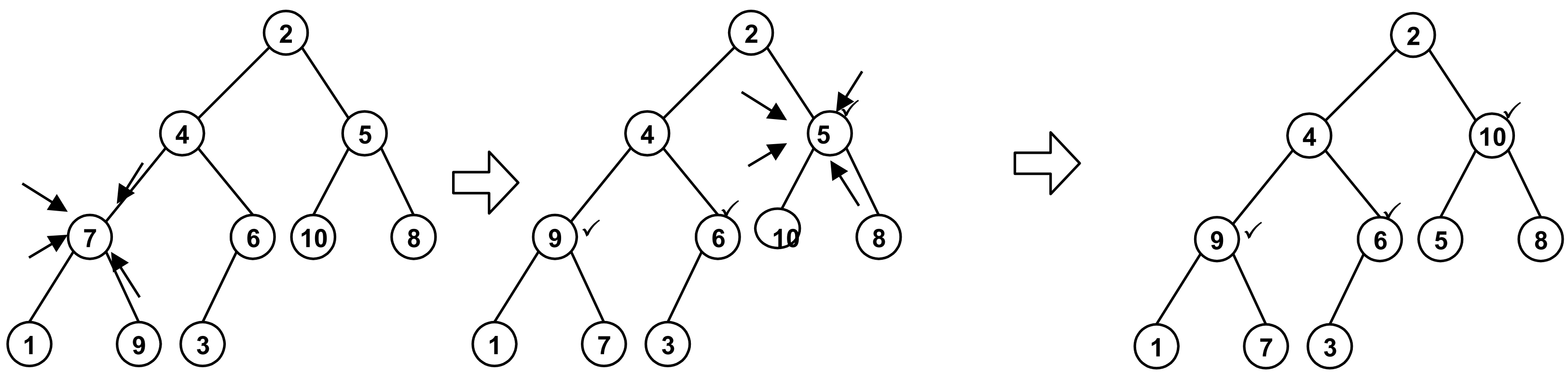
Example 1 (Max-Heap)



$\underline{3} \leftrightarrow 6$

--- keys already in the tree ---

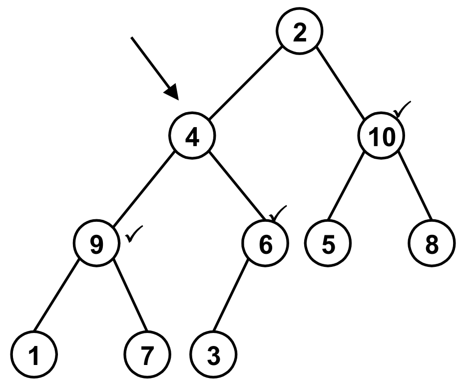
I am not drawing the leaves anymore here



$\underline{7} \leftrightarrow 9$

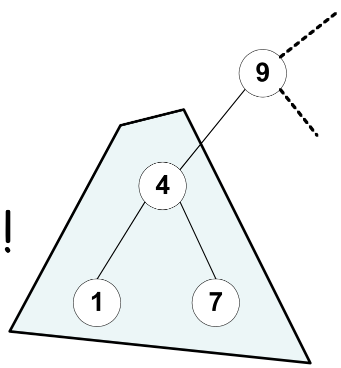
$\underline{5} \leftrightarrow 10$

Example 1

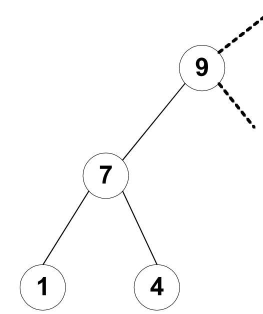


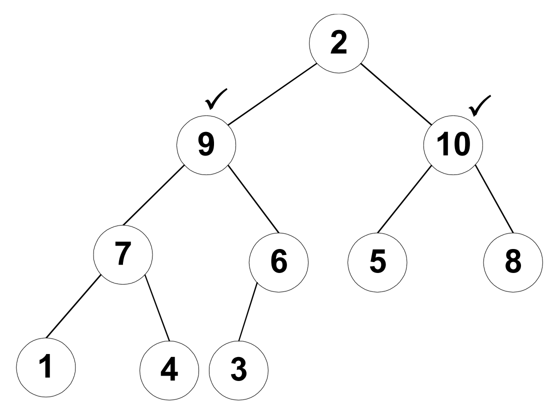
$\underline{4} \leftrightarrow 9$

This is not a heap !

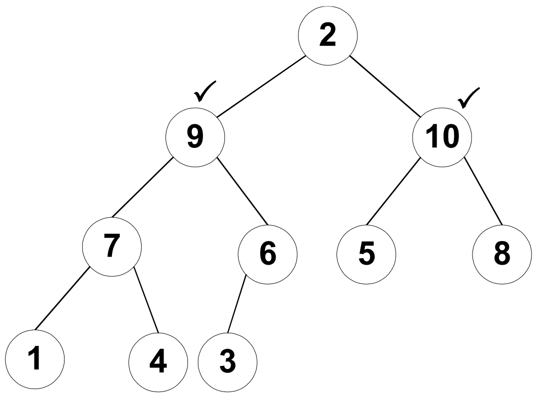


$\underline{4} \leftrightarrow 7$



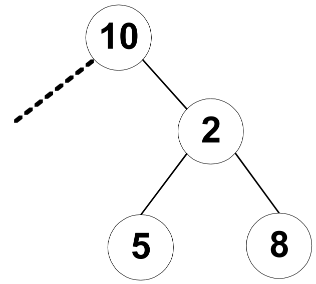


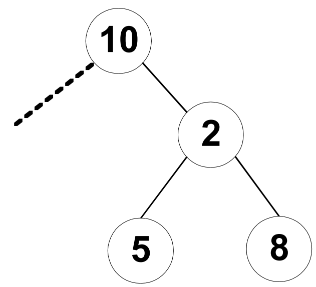
Example 1



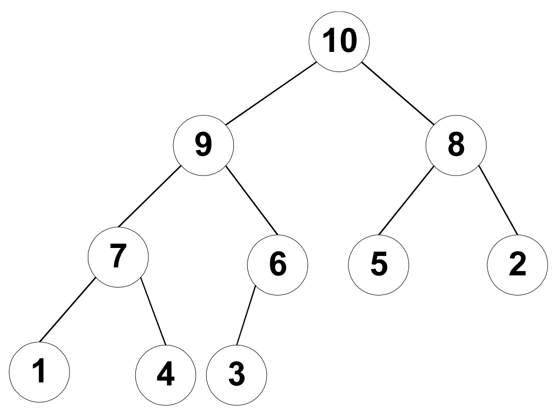
Finally:
10

$\underline{2} \leftrightarrow$



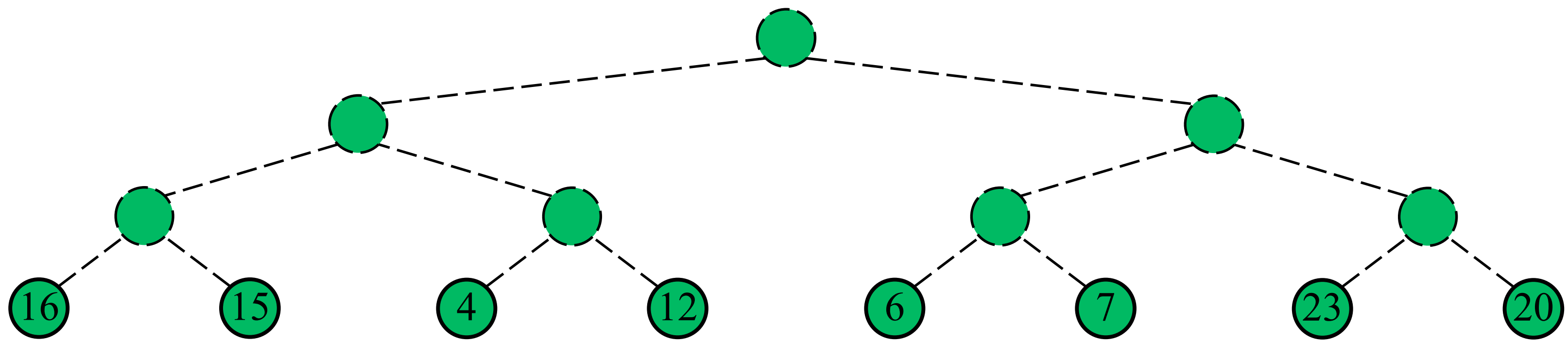


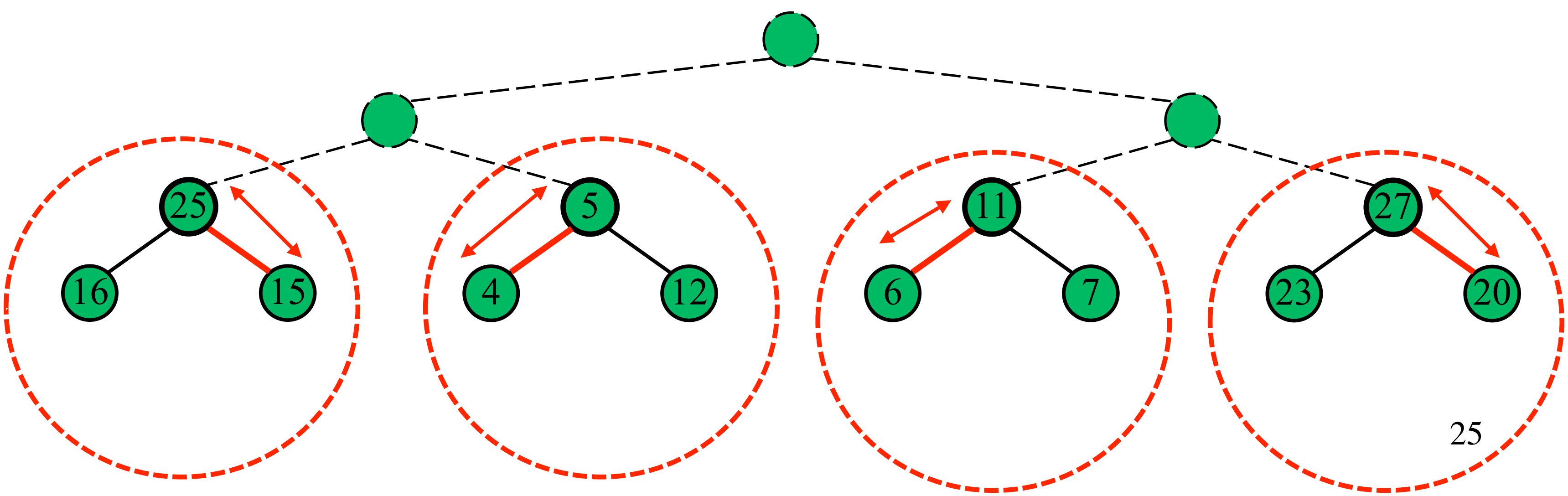
$\underline{2} \leftrightarrow 8$



Keys given one at a time:

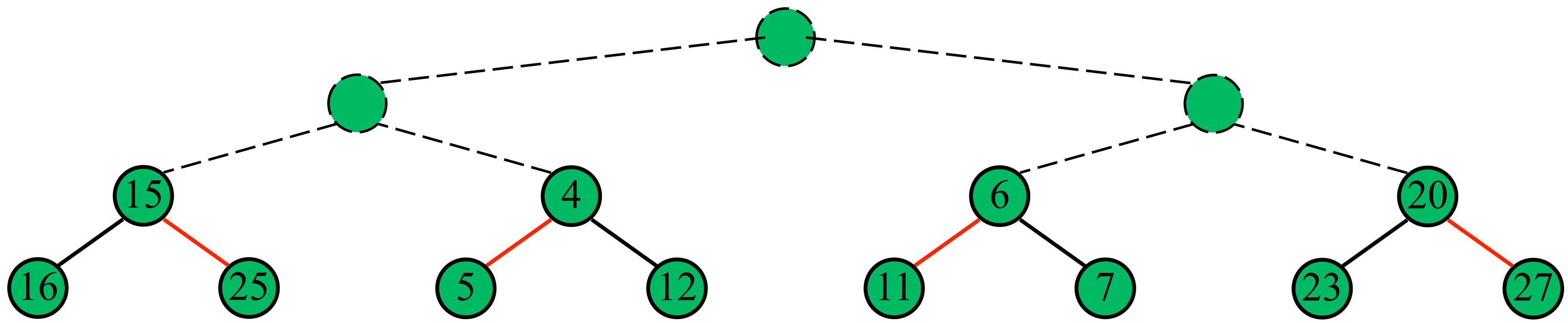
Example 2 Min-Heap
{14,9,8,25,5,11,27,16,15,4,12,6,7,23,20}

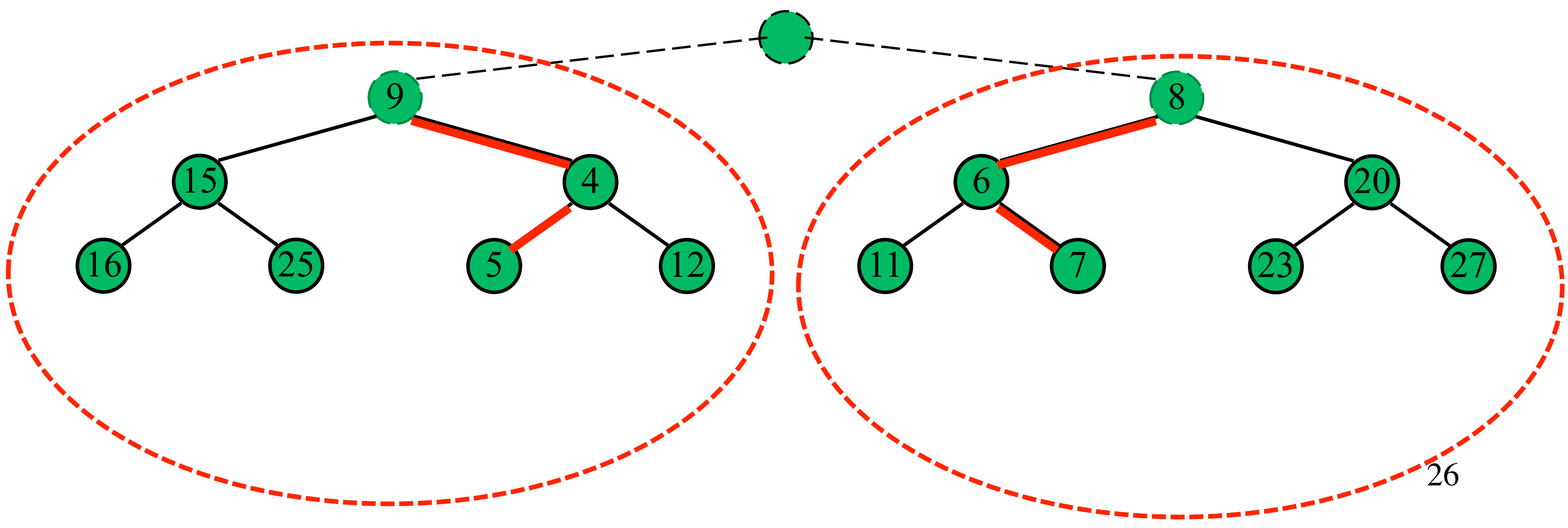




Keys given one at a time:

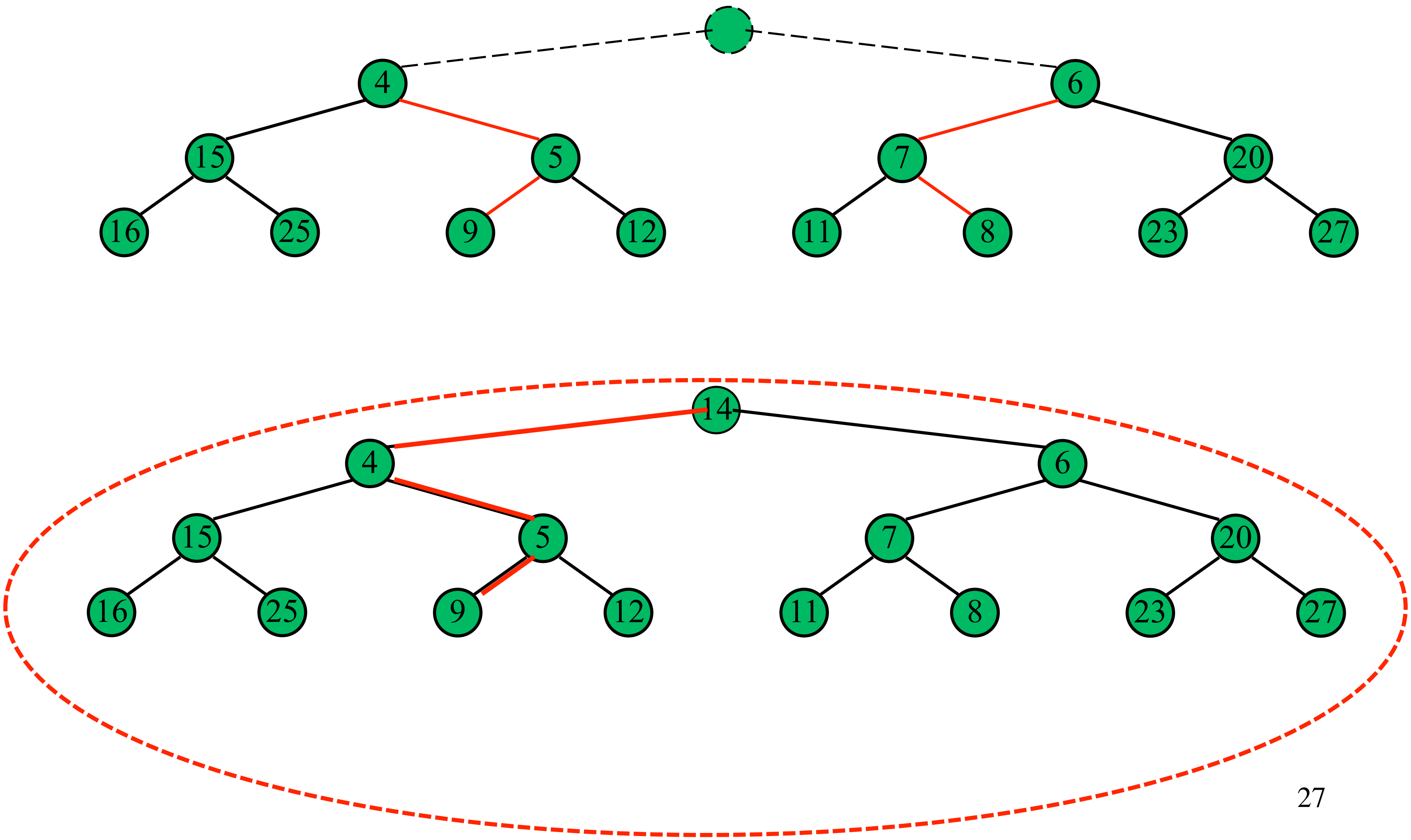
Example 2 Min-Heap
{14,9,8,25,5,11,27,16,15,4,12,6,7,23,20}





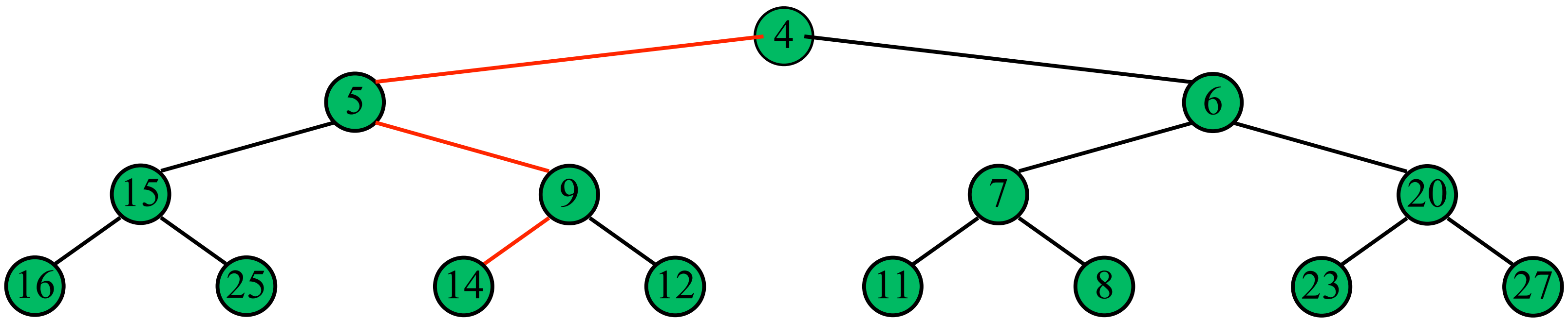
Keys given one at a time:

Example 2 Min-Heap
{14,9,8,25,5,11,27,16,15,4,12,6,7,23,20}



Keys given one at a time:

Example 2 Min-Heap
{14,9,8,25,5,11,27,16,15,4,12,6,7,23,20}

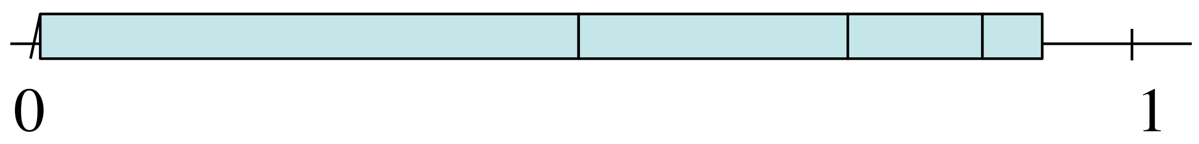


Analysis of Heap Construction

Before we start ....

?

$$\sum 2^{-j} = 1/2 + 1/4 + 1/8 + 1/16 + \cdots \leq 1$$



Analysis of Heap Construction

(let us not consider the dummy leaves)

Number of swaps

h = 4

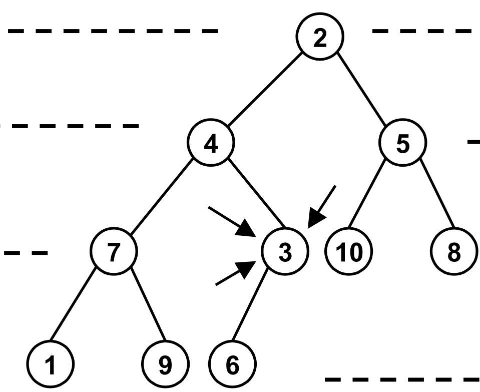
at most

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

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

1 swap ------- level 2

0 swaps ----- level 3

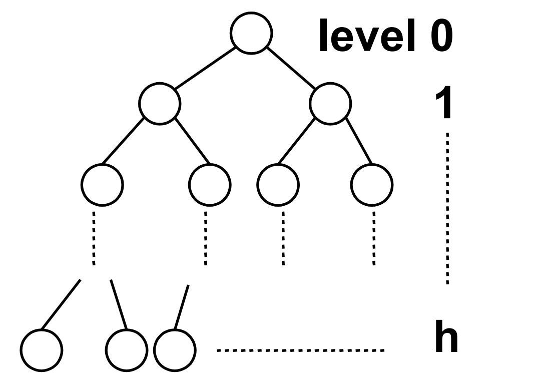


h is the max level

level i -------- h - i swaps

Analysis of Heap Construction

Number of swaps



At level i the number of swaps is

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

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

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

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

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

Consider $\sum j\, 2^{-j}$:

$$\begin{aligned}
\sum j\, 2^{-j} &= 1/2 + 2\ 1/4 + 3\ 1/8 + 4\ 1/16 + \cdots \\
&= 1/2 + 1/4 + 1/8 + 1/16 + \cdots <= 1 \\
&+ \quad\quad 1/4 + 1/8 + 1/16 + \cdots <= 1/2 \\
&+ \quad\quad\quad\quad 1/8 + 1/16 + \cdots <= 1/4 \\
\hline
\sum j\, 2^{-j} & \quad\quad\quad\quad\quad\quad\quad\quad <= 2
\end{aligned}$$

So $2^h \sum j\, 2^{-j} <= 2 \cdot 2^h = 2\, n \quad O(n)$

$$2^h \sum_{j=1}^{h} j/2^j \leq 2^{h+1}$$

Where **h** is **O(log n)**

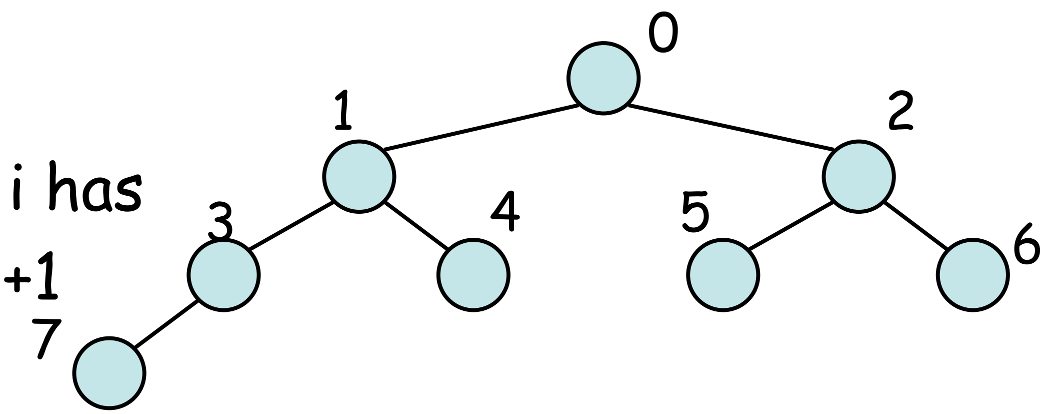
So, the number of swaps is $\leq$ **O(n)**

Implementing a Heap with an Array

A heap can be nicely represented by an array list
(array-based),

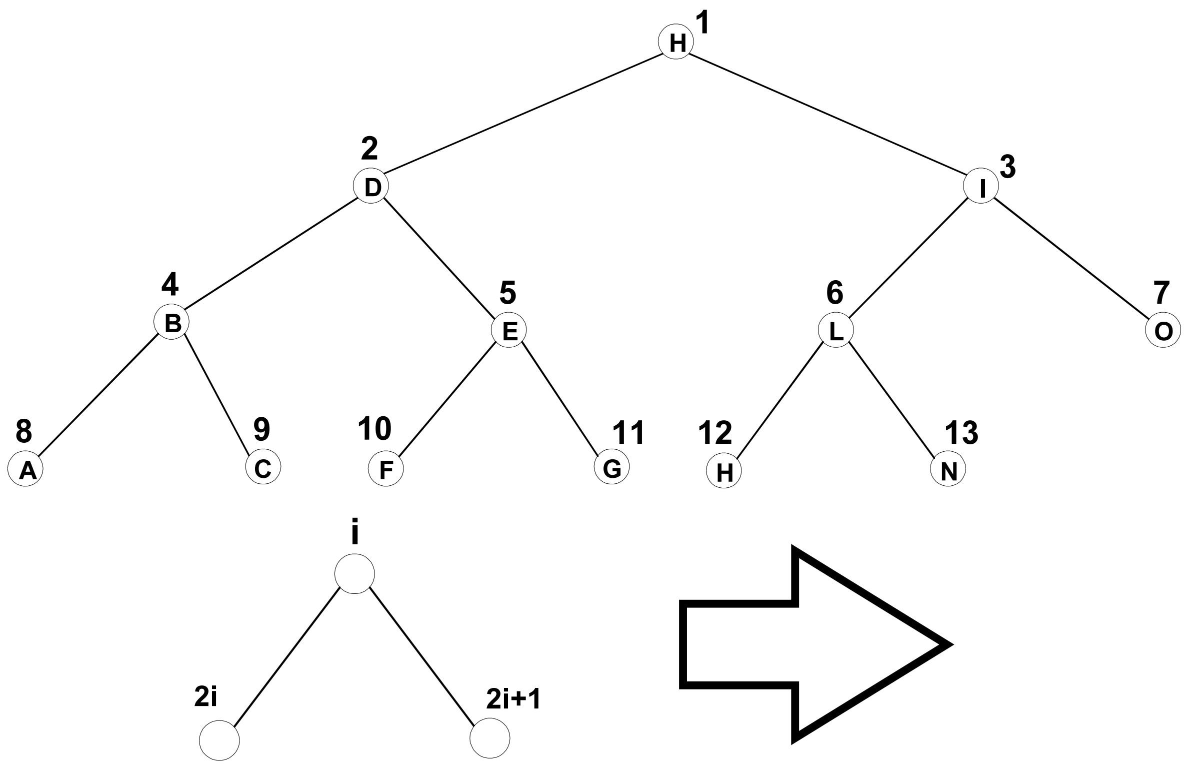
where the node at rank i has
- left child at rank 2i+1

and
- right child at rank 2i + 2



0	1	2	3	4	5	6	7

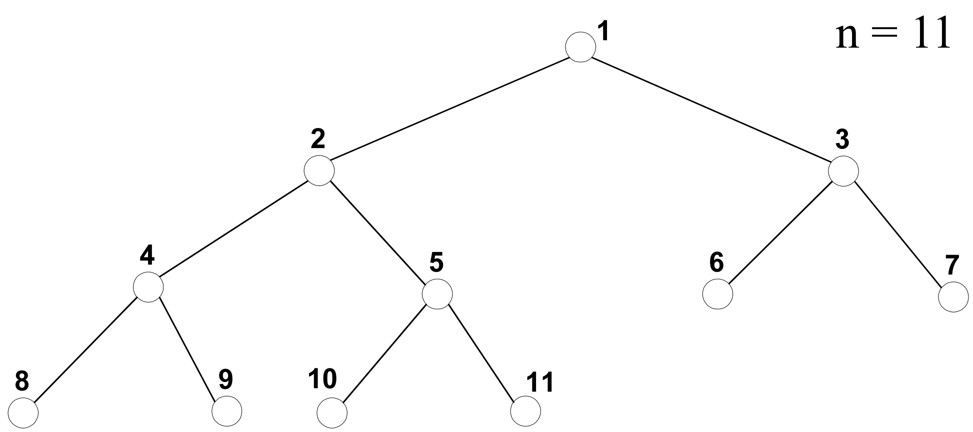
Example (with indexes 1..N)



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

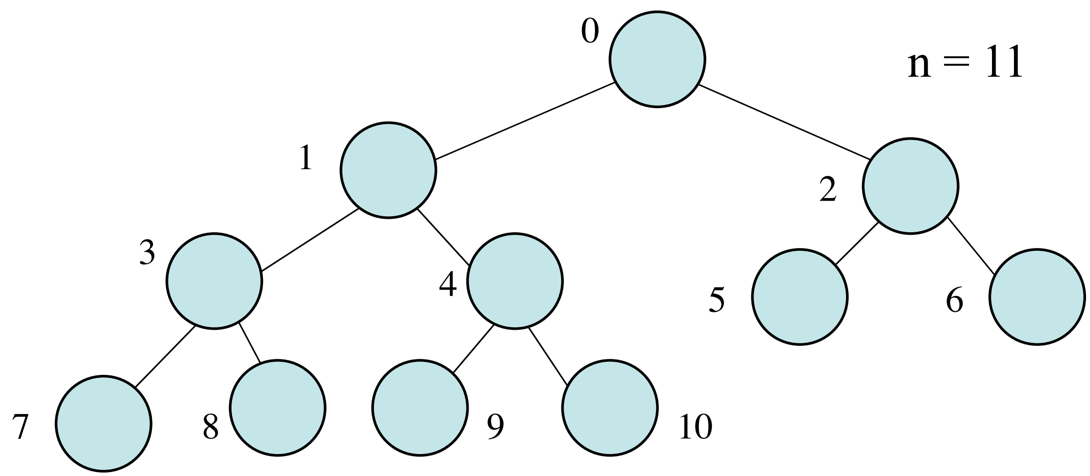
Reminder if using indices 1 to n:

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

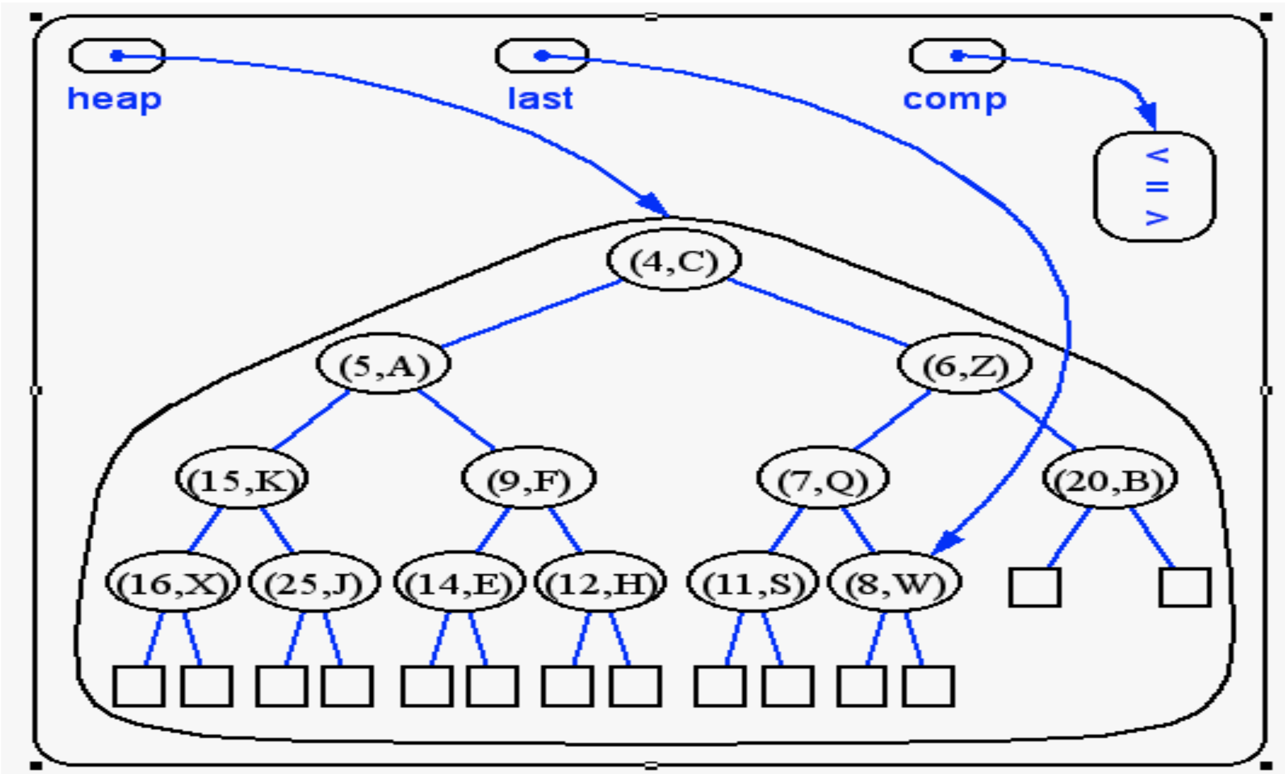


Reminder if using indices 0 to n-1:

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



Implementation of a Priority Queue with a Heap



(upheap)

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

(remove root + downheap)

Application: Sorting Heap Sort

PriorityQueueSort where the PQ is implemented with a HEAP

```
Algorithm PriorityQueueSort(S, P):
    Input: A sequence S storing n elements, on which a
           total order relation is defined, and a Priority
         Queue P that compares keys with the same relation
    Output: The Sequence S sorted by the total
                 order relation
    while ¬ S.isEmpty() do
           e ← S.removeFirst()          } Build Heap
           P.insertItem(e, e)
    while ¬ P.isEmpty() do
           e ← P.removeMin()            } Remove from heap
           S.insertLast(e)
```

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	$O(\log (n-1))$
	…	$\vdots$
	…	

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

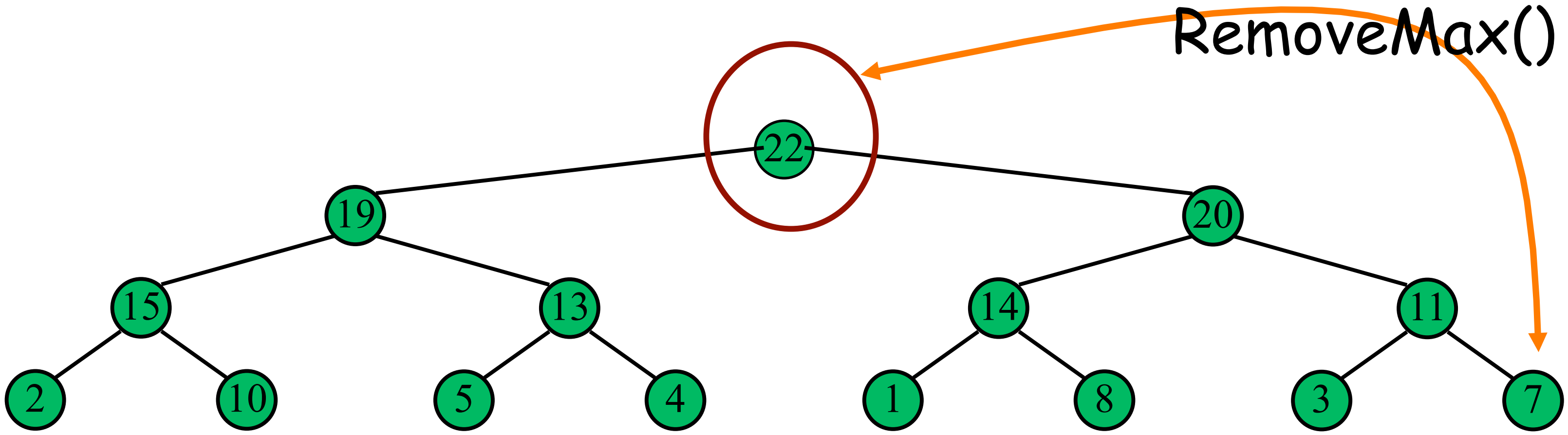
$$\rightarrow \text{TOT} = \sum_{i=1}^{n} \lfloor \log i \rfloor$$

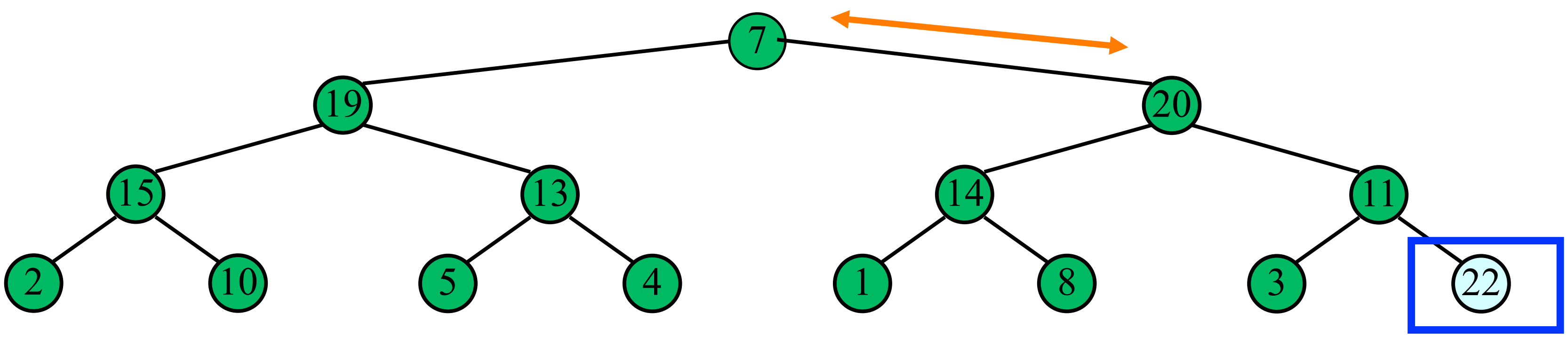
$$= O(n \log n)$$

HeapSort in Place

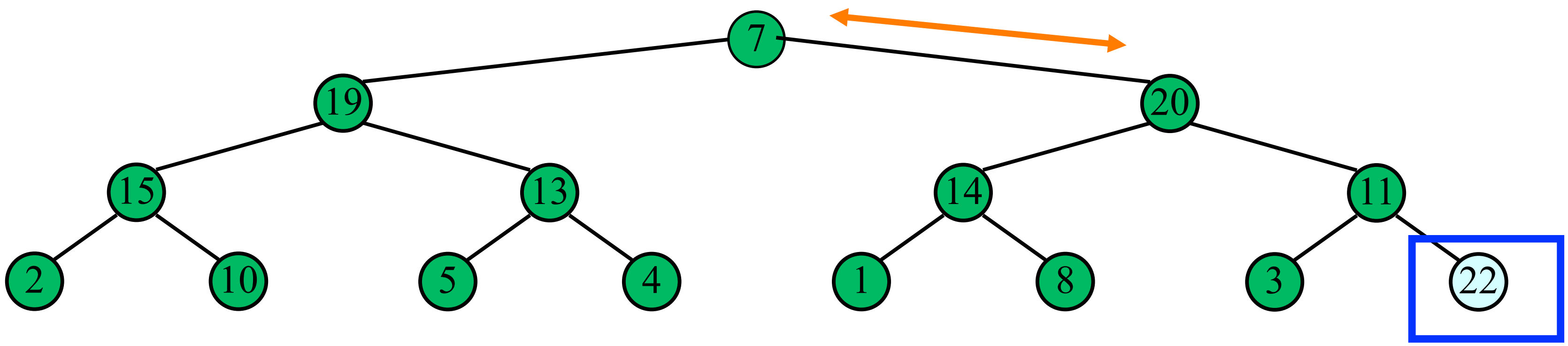
Instead of using a secondary data structure P to sort a sequence S, We can execute heapsort « in place » by dividing S in two parts, one representing the heap, and the other representing the sequence.
The algorithm is executed in two phases:

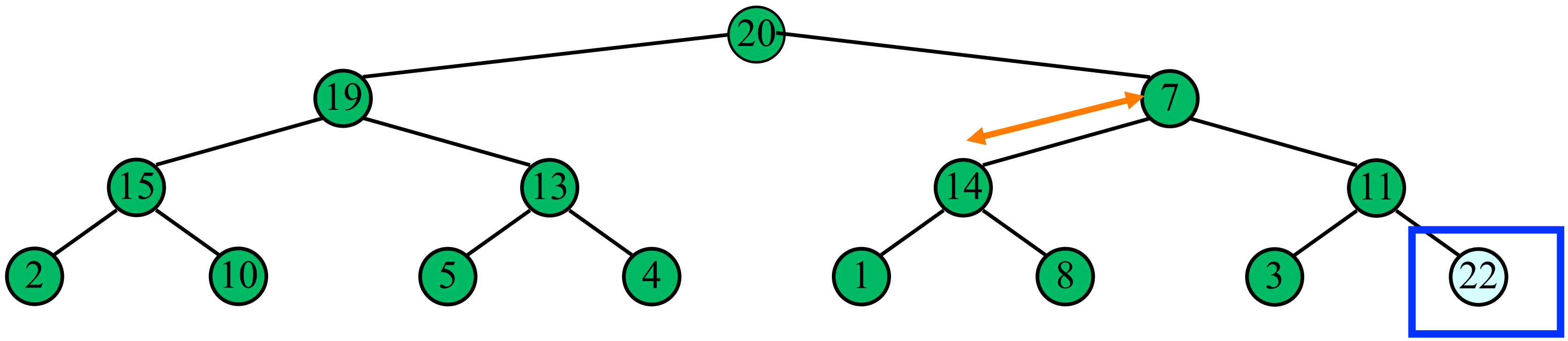
- ✓ Phase 1: We build a max-heap so to occupy the whole structure.
- ✓ Phase 2: We start with the part « sequence » empty and we grow it by removing at each step i (i=1..n) the max value from the heap and by adding it to the part « sequence », always maintaining the heap properties for the part « heap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

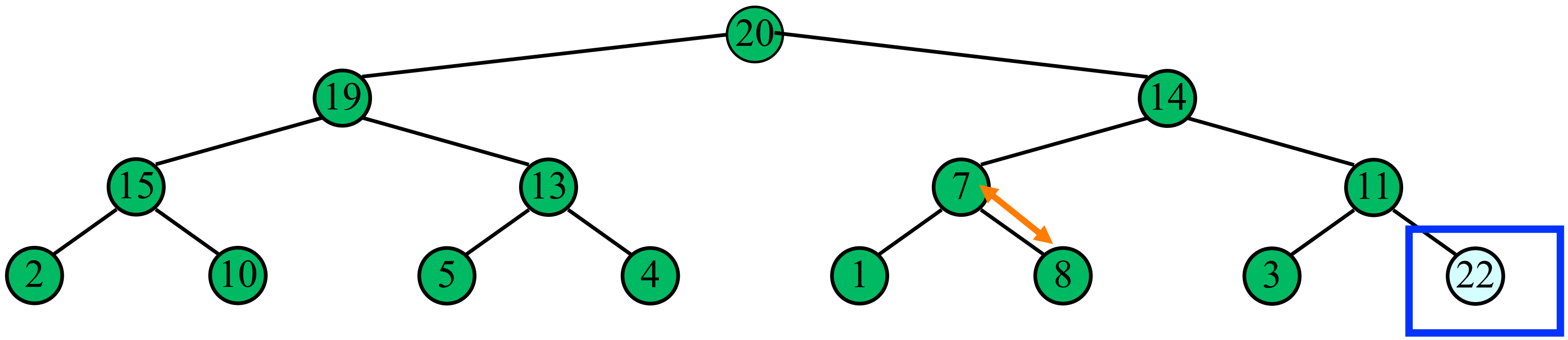


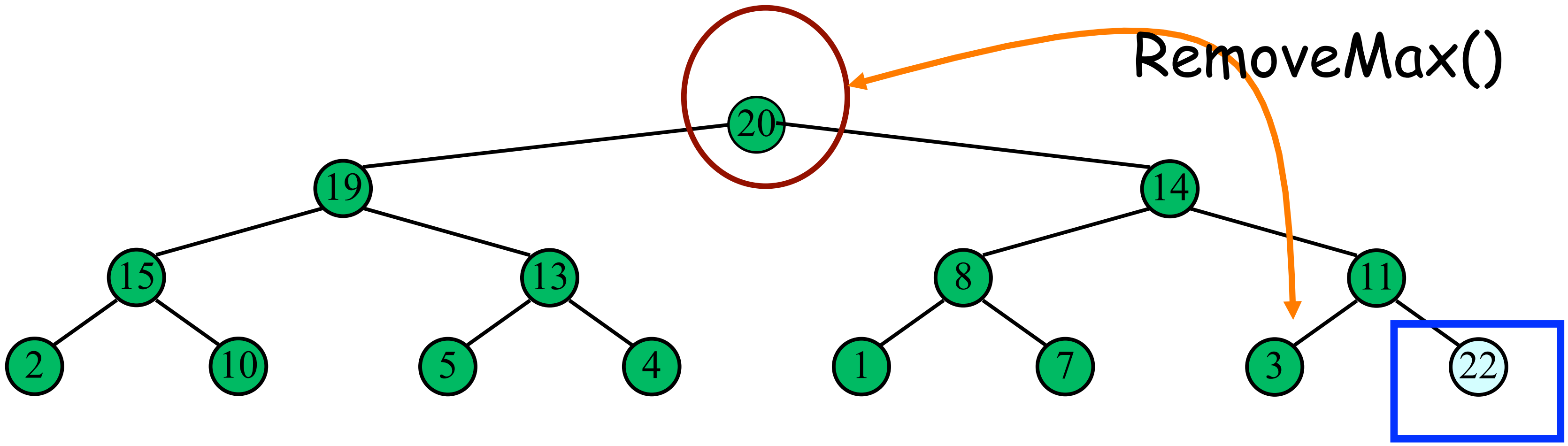


Not a heap anymore !


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

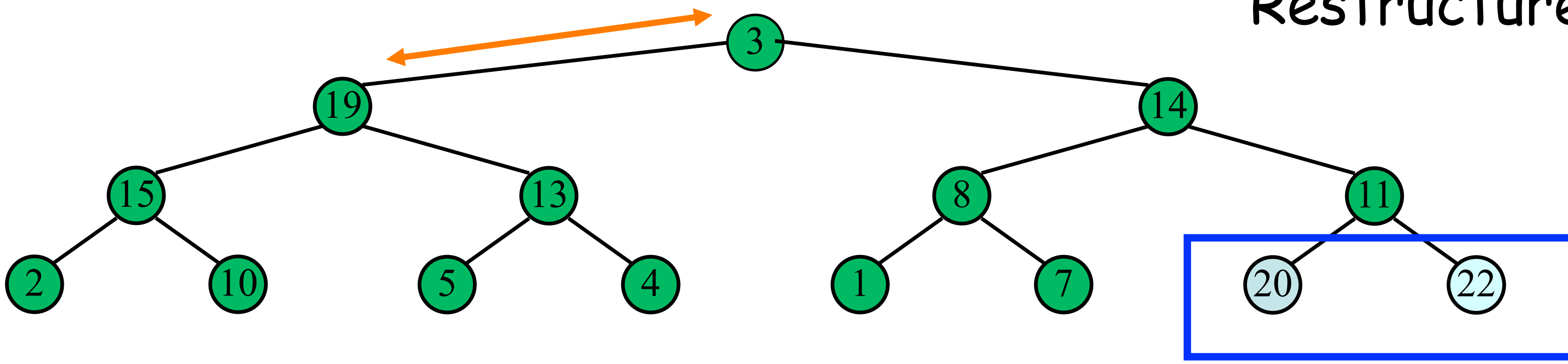

20
19
7
15
13
14
11
2
10
5
4
1
8
3
22


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



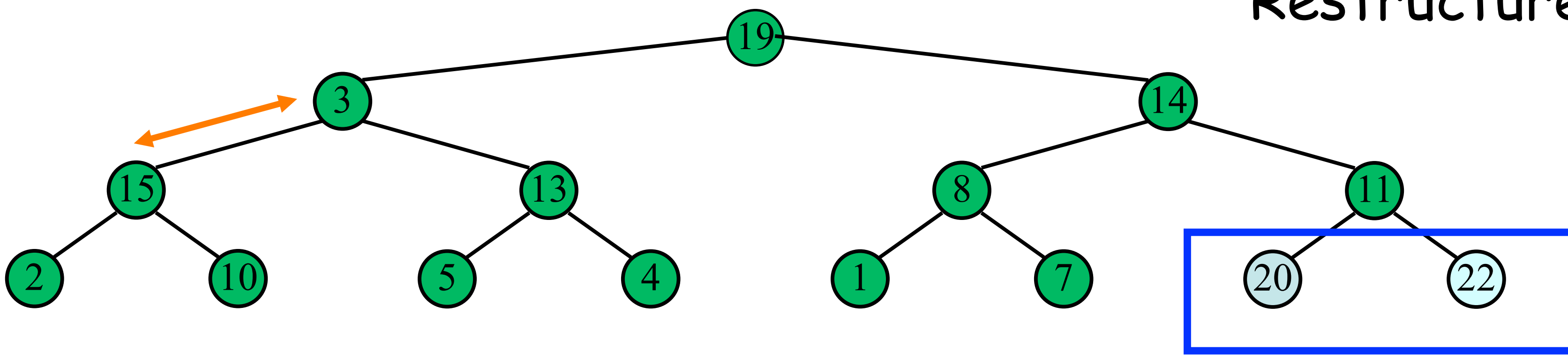
Now the part heap is smaller, the part Sequence contains a single element.

Restructure



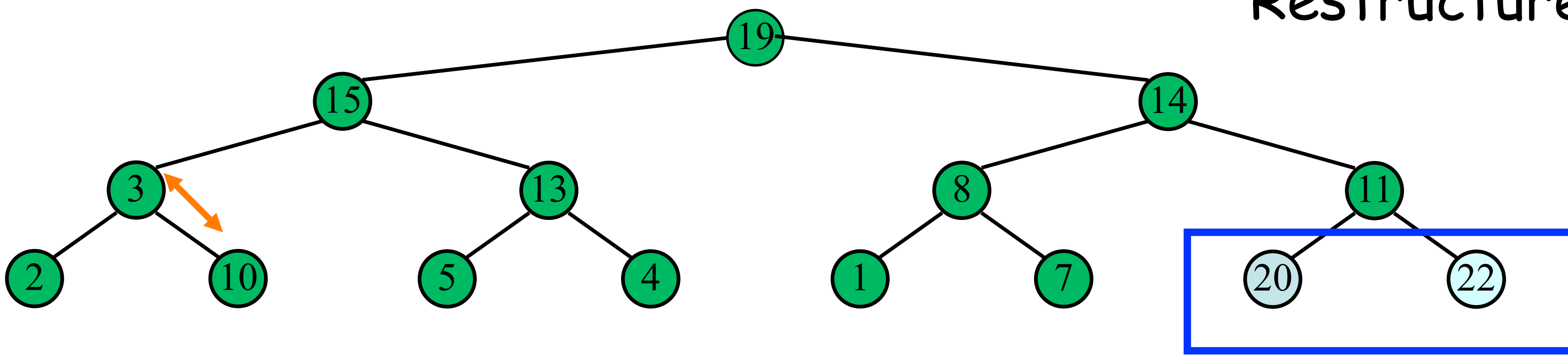
Not a heap anymore !

Restructure

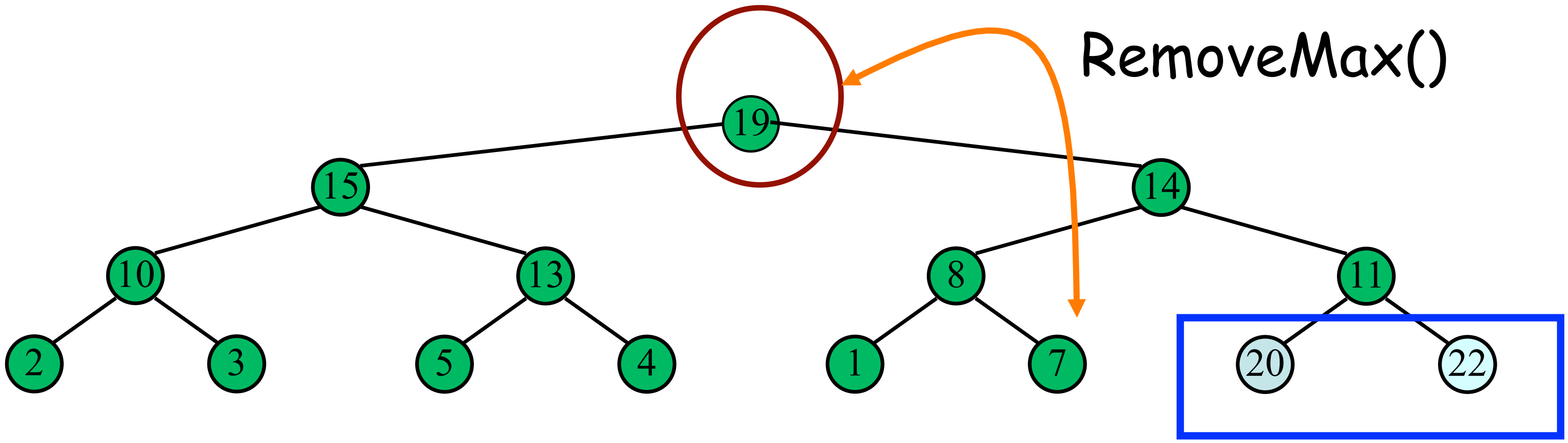


Not a heap anymore !

Restructure

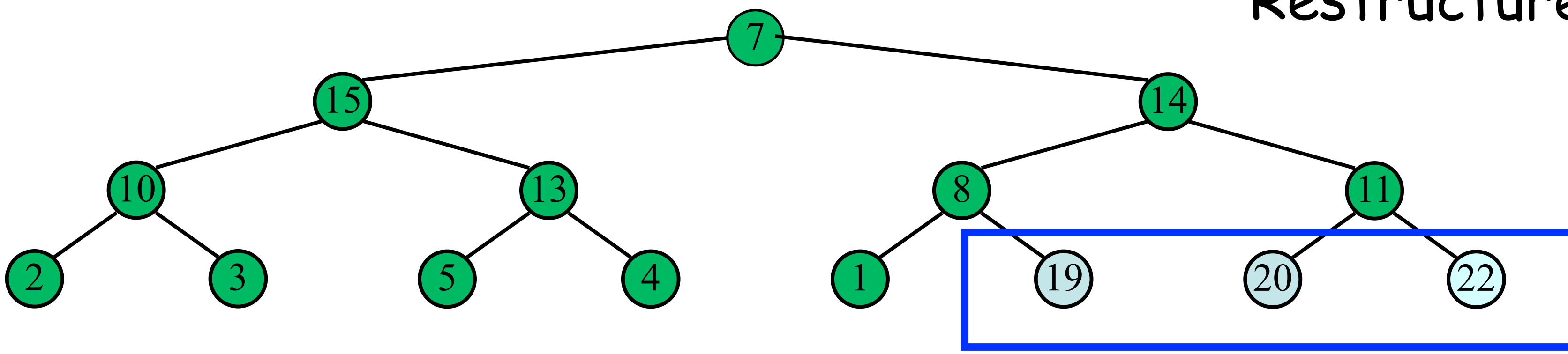


Not a heap anymore !

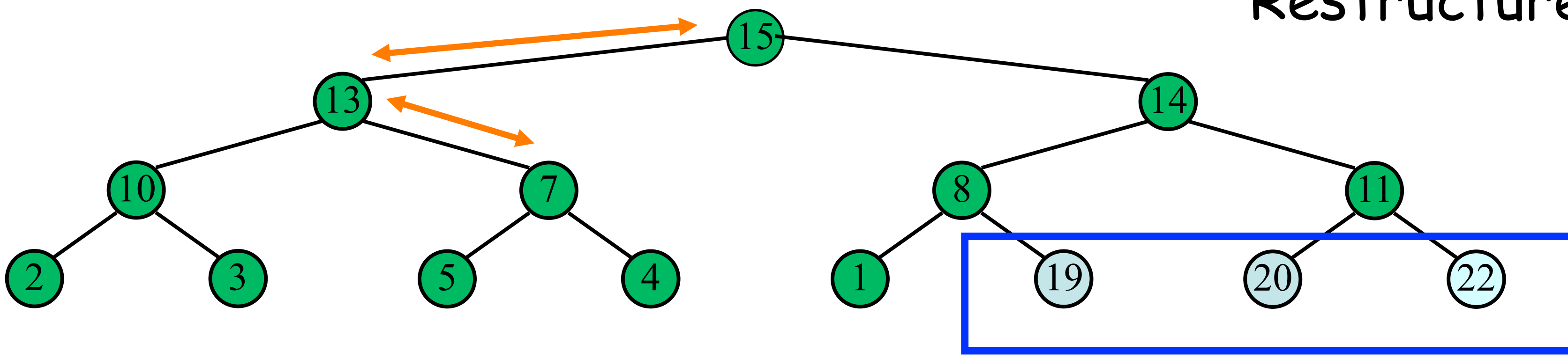


Now it is a heap again !

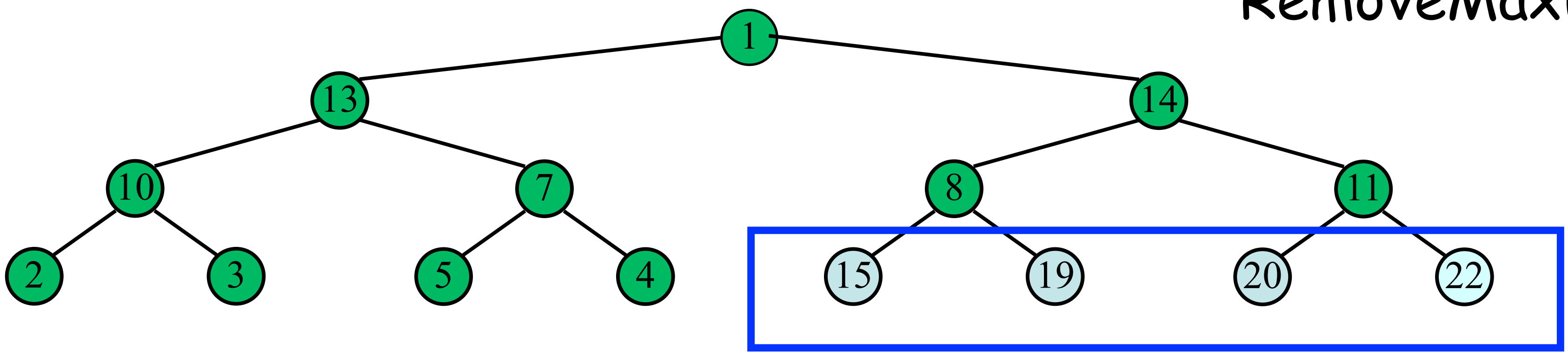
Restructure



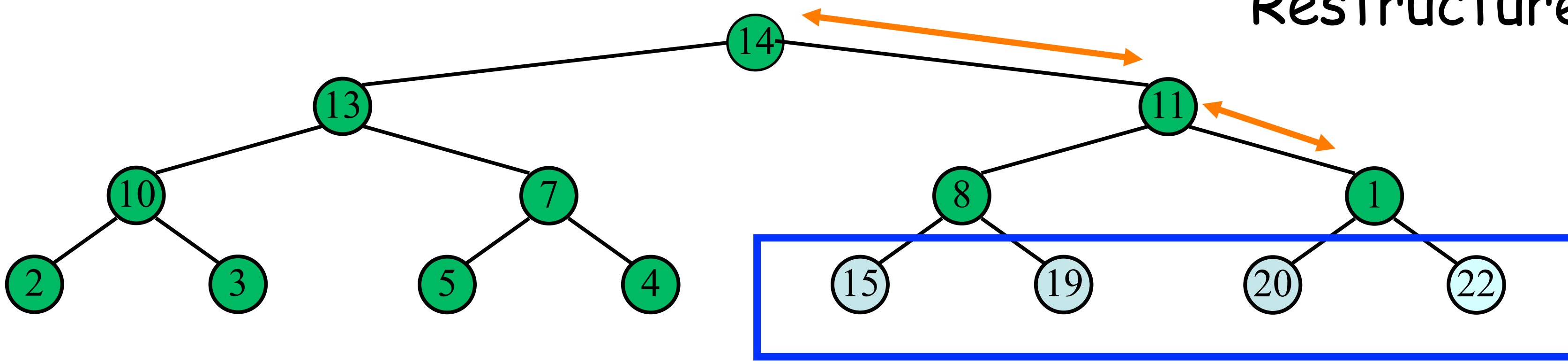
Restructure



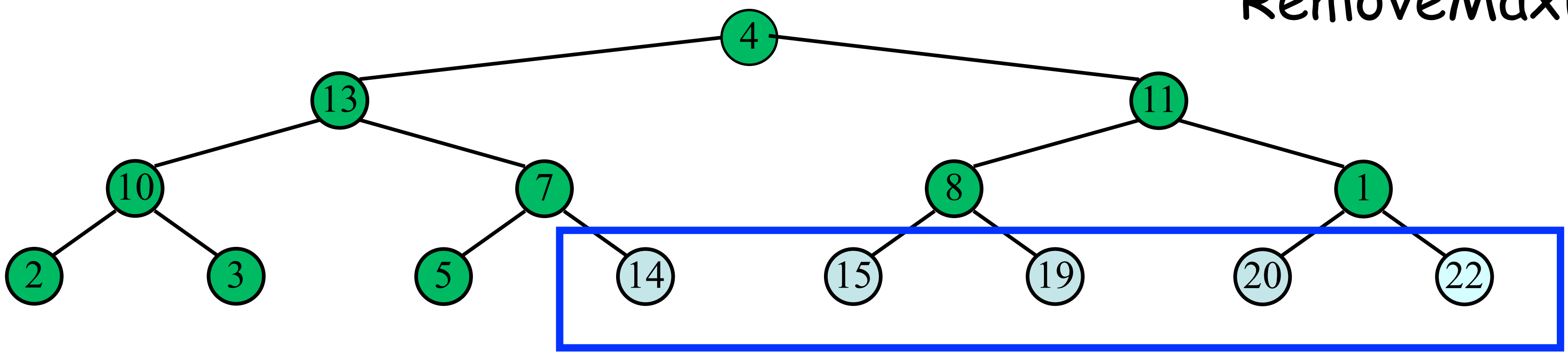
RemoveMax()



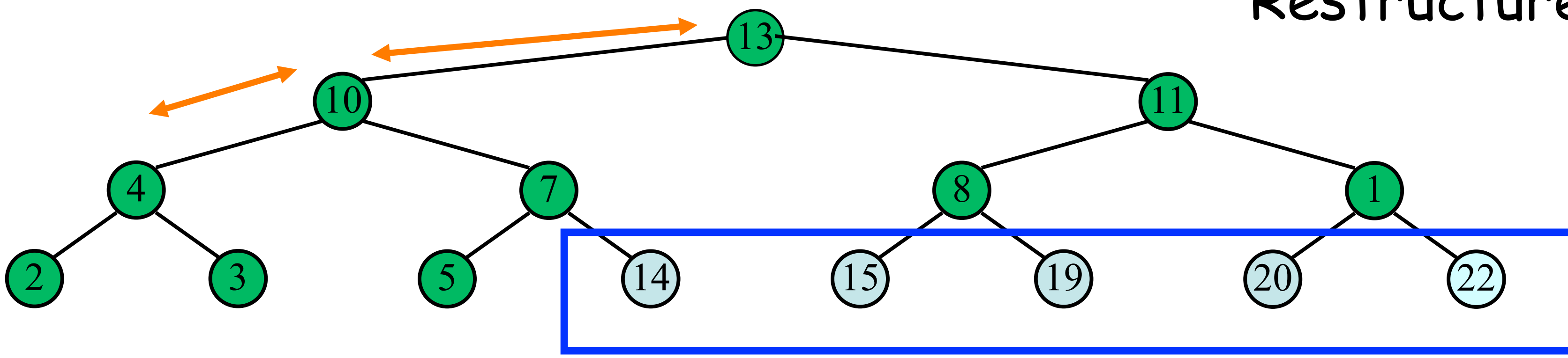
Restructure

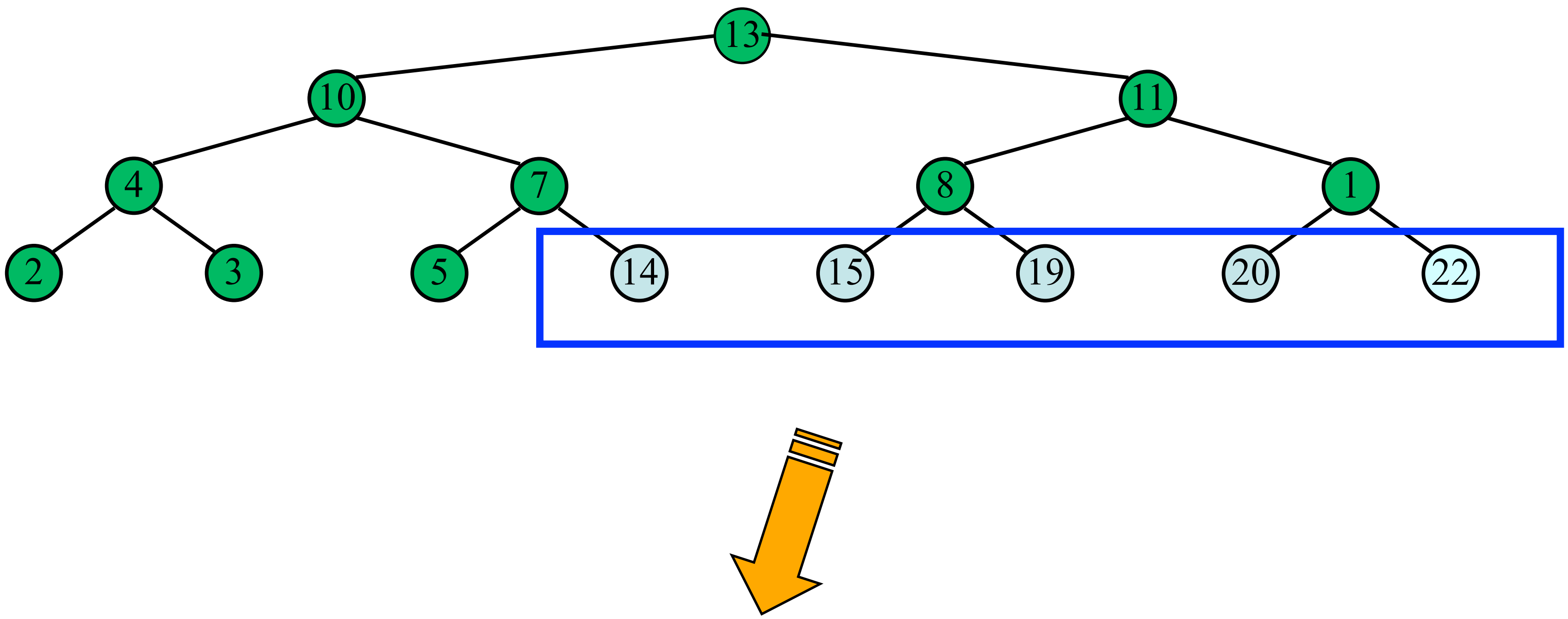


RemoveMax()



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
```

HeapSort in Place

Continue example on the board.