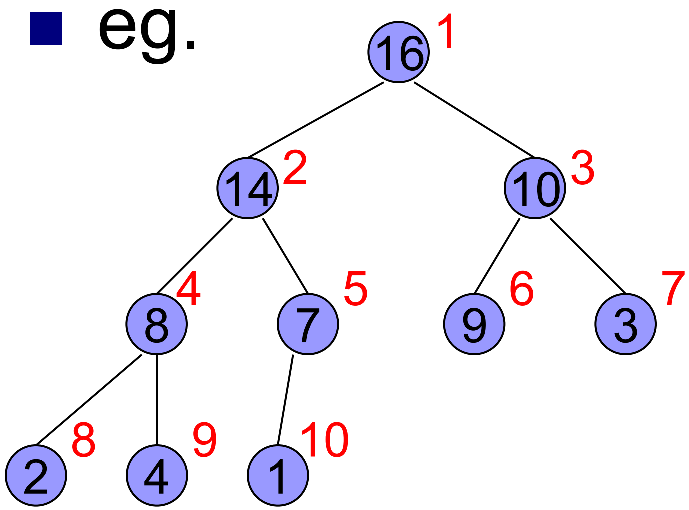
Heap and Heapsort

Heaps

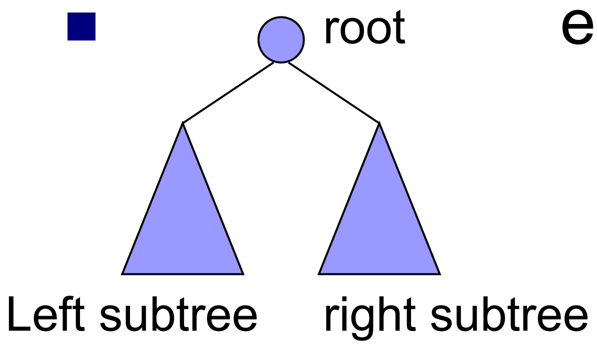
- A data structure with
  - Nearly complete binary tree
  - Heap property: A[parent(i)] ≥ A[i]
- eg.



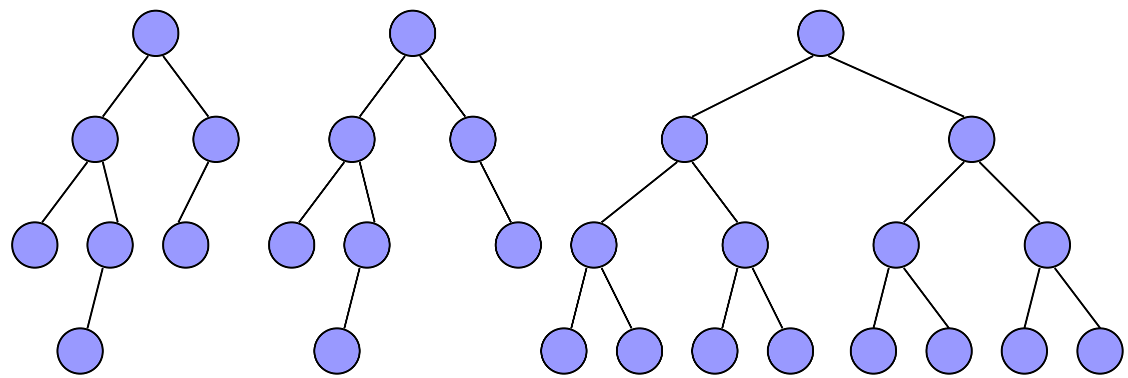
Parent(i) { return $\left\lfloor \frac{i}{2} \right\rfloor$ }
Left(i) { return 2i}
Right(i) { return 2i+1}

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Binary tree

- Contains no node, or
- 

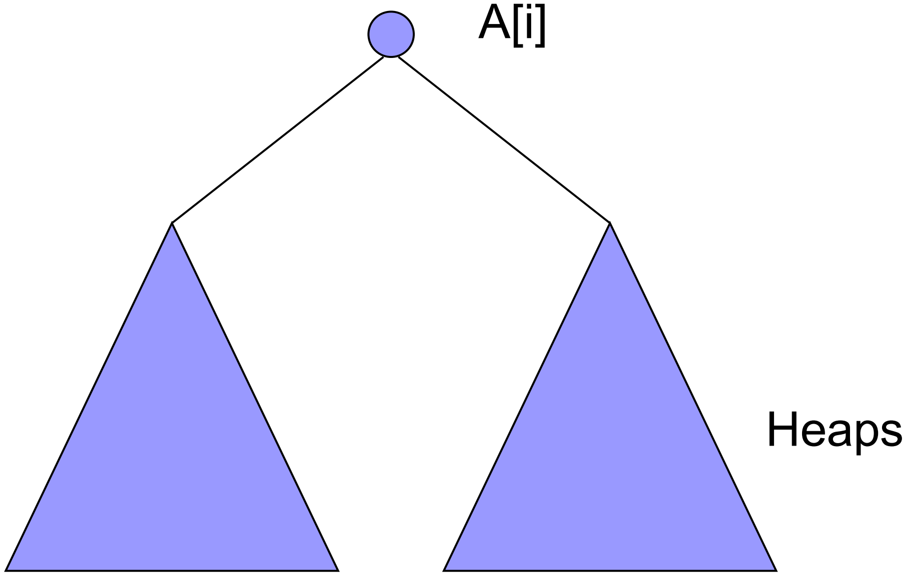
eg.



- A node without subtree is called a leaf.
- In a full binary tree, each node has 2 or NO children.
- A complete binary tree has all leaves with the same depth and all internal nodes have 2 children.

Maintaining the heap property

- Condition:
  A[i] may be smaller than its children.

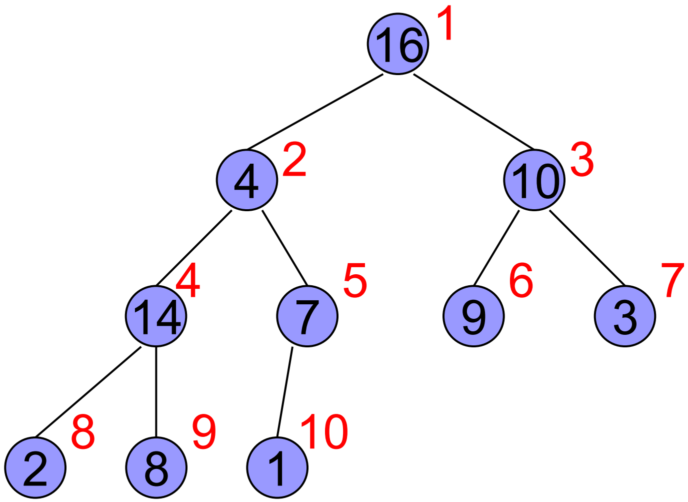


Pseudocode Heapify(A,i)

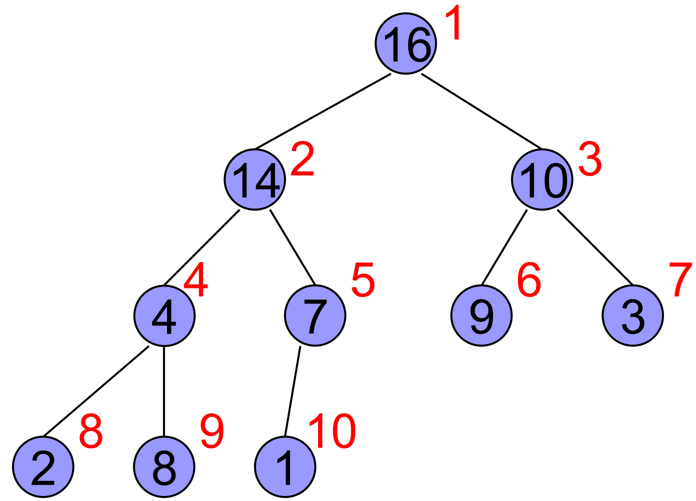
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

- Time: $O(\lg n)$, $T(n) \le T(2n/3) + \Theta(1)$

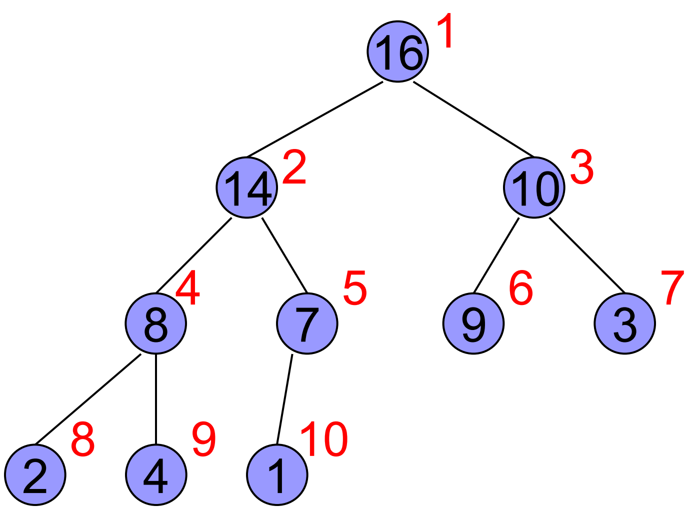
Heapify(A,2):



Heapify(A,4):

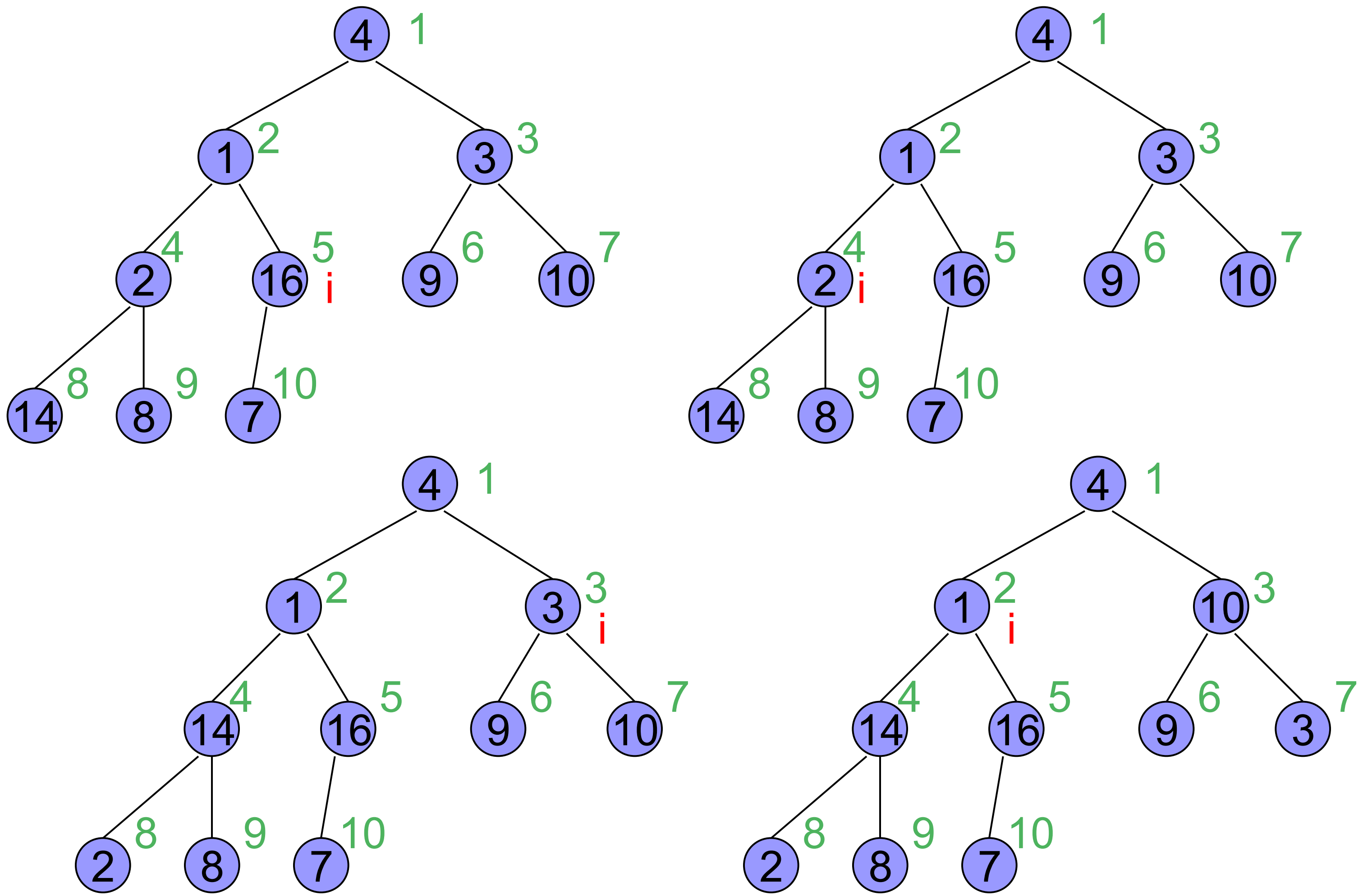


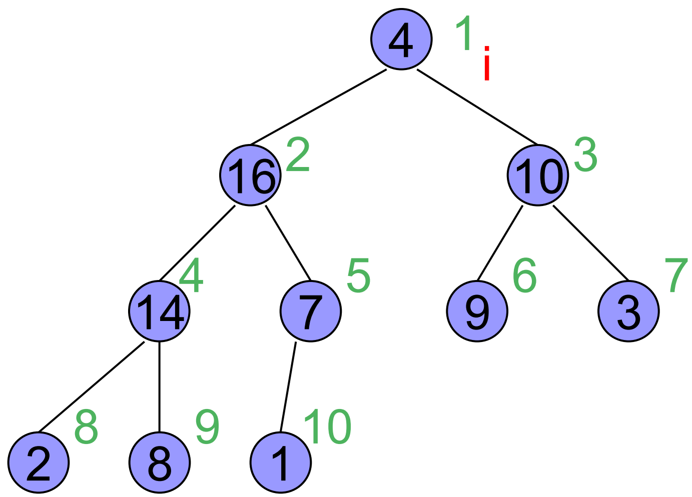
Heapify(A,9):

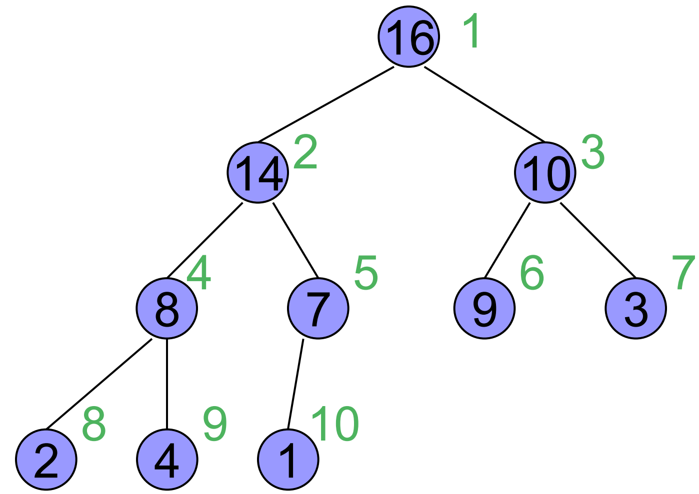


Build Heap

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

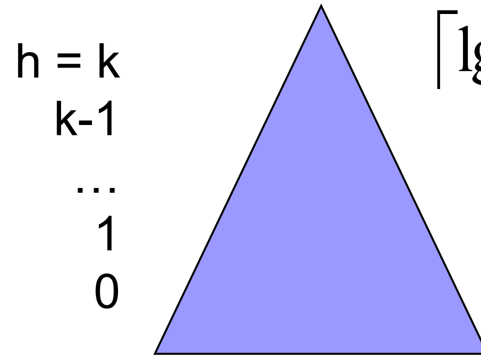







```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Analysis



$$\lceil \lg n \rceil = k$$

- By intuition:
  - Each call of Heapify cost Θ(lg n). There are O(n) calls. Thus, Build-Heap takes O(n lg n).
- Tighter analysis: O(n)
  - Assume n = 2^k-1, a complete binary tree. The time required by Heapify when called on a node of height h is O(h).
  - Total cost = $$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O(n)$$

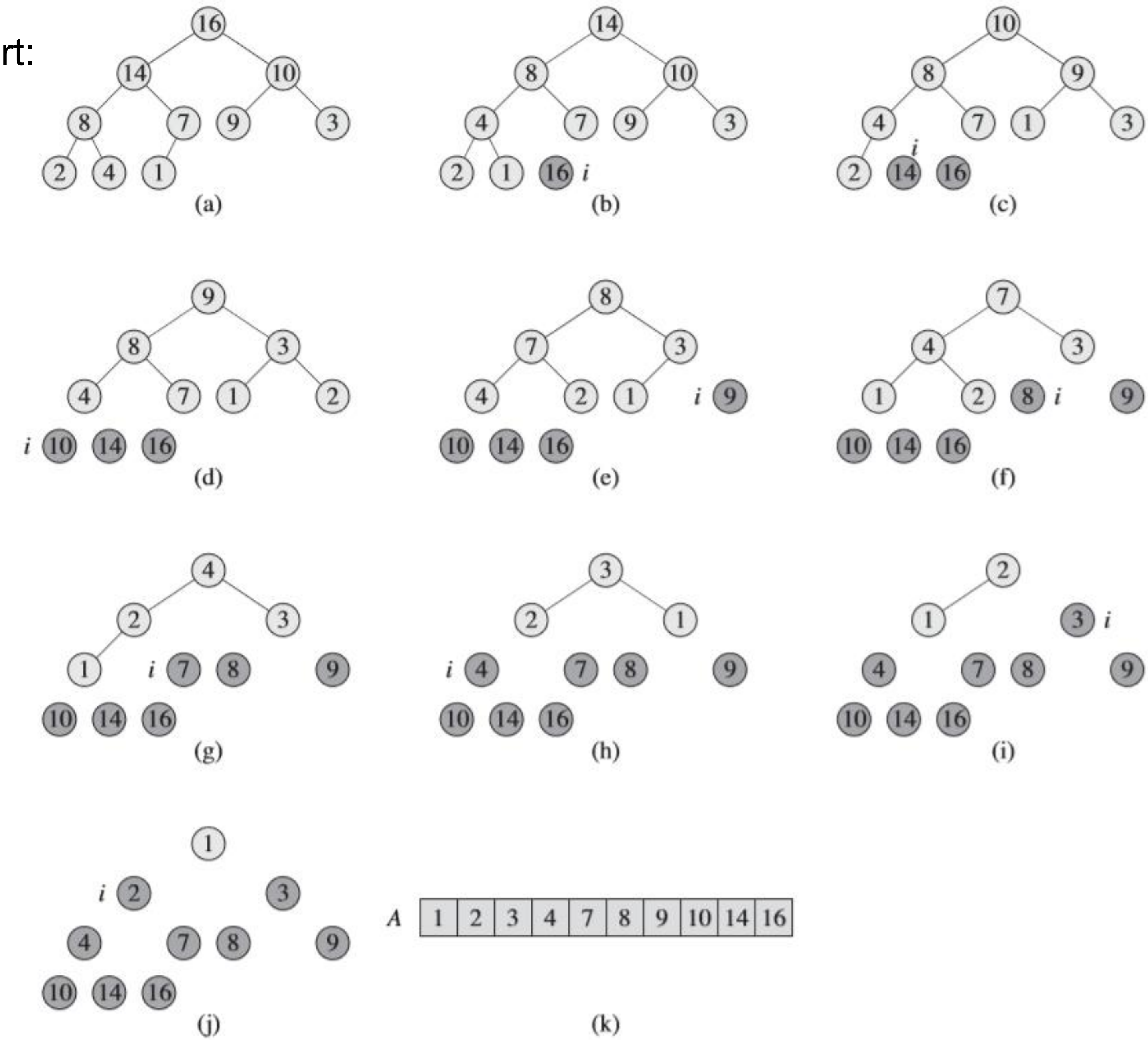
    by exercise: $$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Heapsort algorithm

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)                          ----O(n)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)                      ----O(lg n)
```

Time cost = O(n lg n)

Heap Sort:



Priority queue

- A data structure for maintaining a set S of elements, each with an associated value called a key.
- Application:
  - Job scheduling
  - Simulation
- Operations of a priority queue:
  - Insert(S,x)
  - Maximum(S)
  - Extract-Max(S)

Implement with a heap.

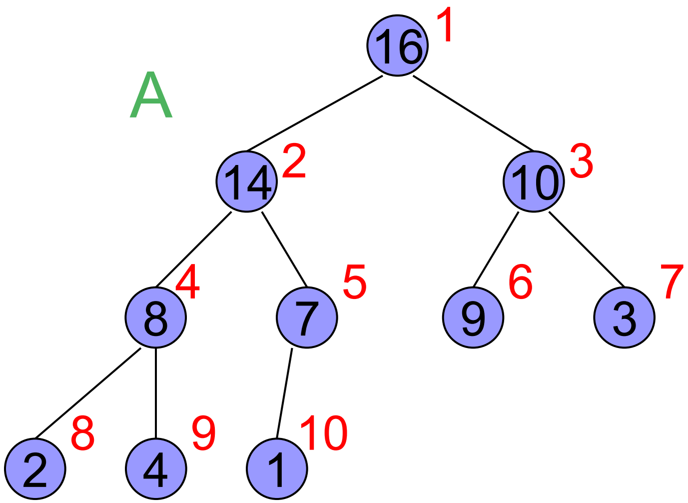
```
HEAP-MAXIMUM(A)
    return A[1]
```

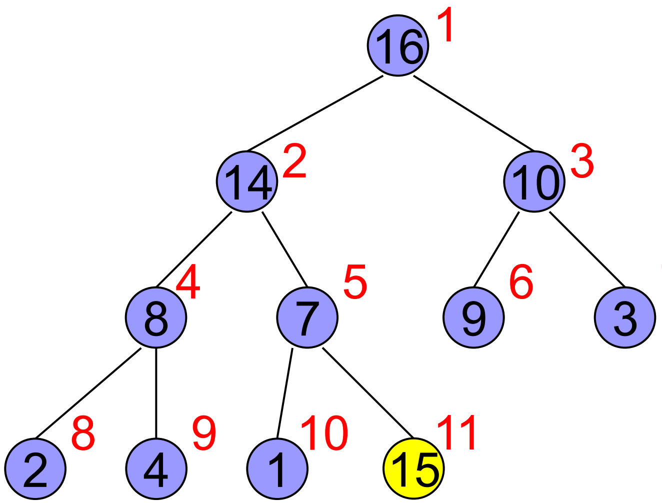
```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
    return max
```

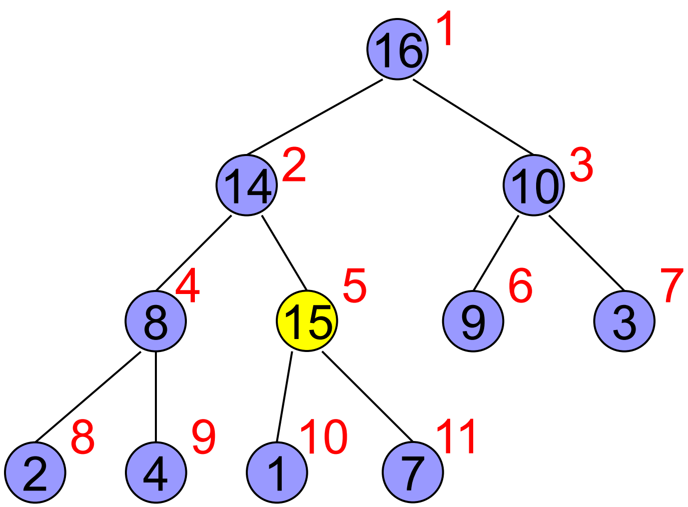
```
HEAP-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

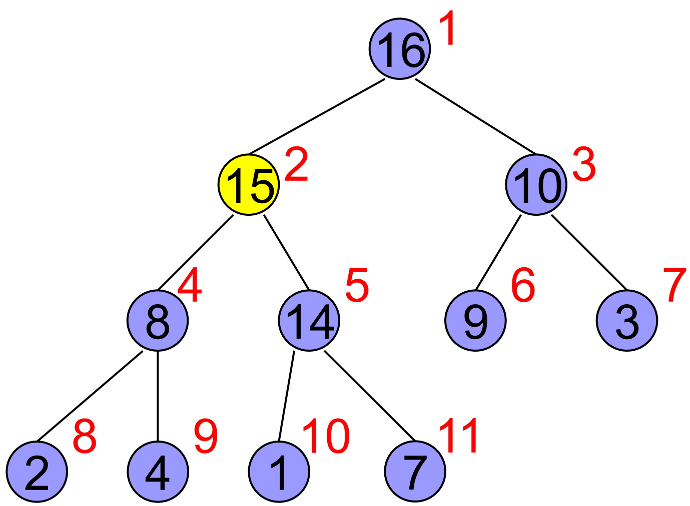
```
MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = -∞
    HEAP-INCREASE-KEY(A, A.heap-size, key)
```

key = 15, HeapInsert(A,key):









HEAP-INCREASE-KEY: