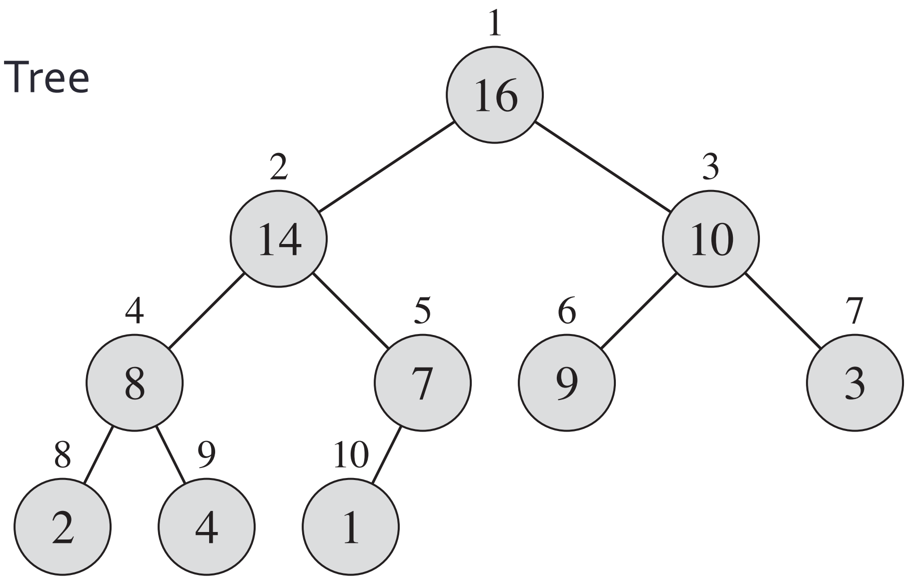
HEAP

Michael Tsai

2017/4/25

Array Representation of Tree

Tree

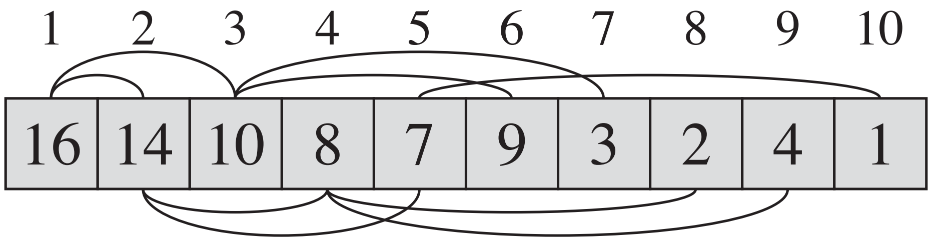


```
PARENT($i$)
    return $\lfloor i/2 \rfloor$

LEFT($i$)
    return $2i$

RIGHT($i$)
    return $2i + 1$
```

Array

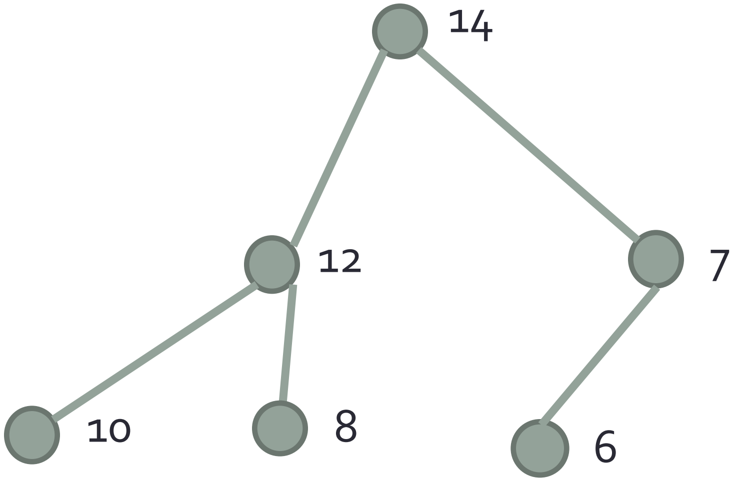


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Note that the index starts at 1 from the root node.

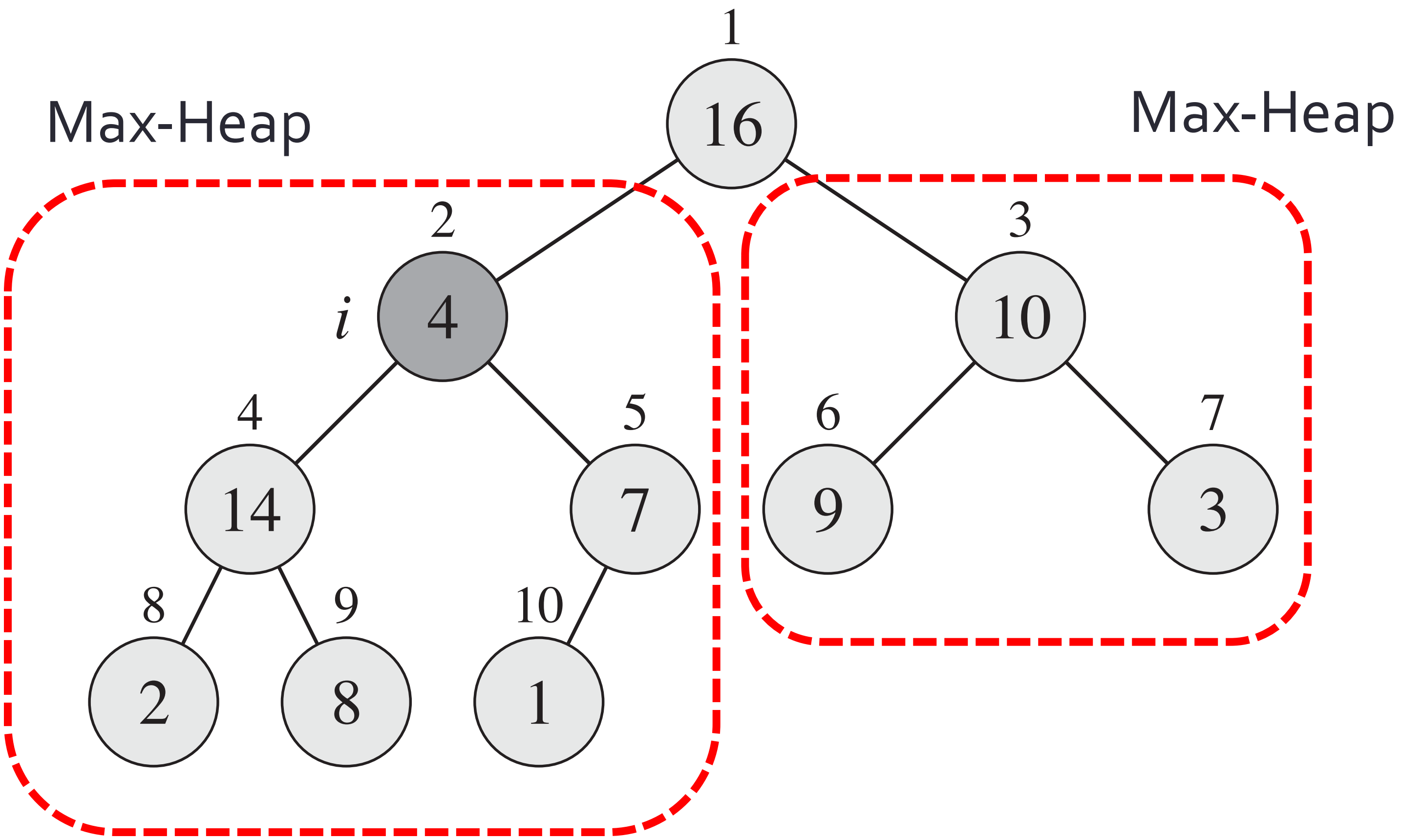
Heap

- Definition: A **max tree** is a tree in which the key value **in each node is no smaller (larger) than the key values in its children** (if any).
- Definition: A **max heap** is a **complete binary tree** that is also a **max tree**. A **min heap** is a **complete binary tree** that is also a **min tree**.



`Max-Heapify`

Assumption: the left and right subtrees are max-heaps, but the root node might violate the max tree property.



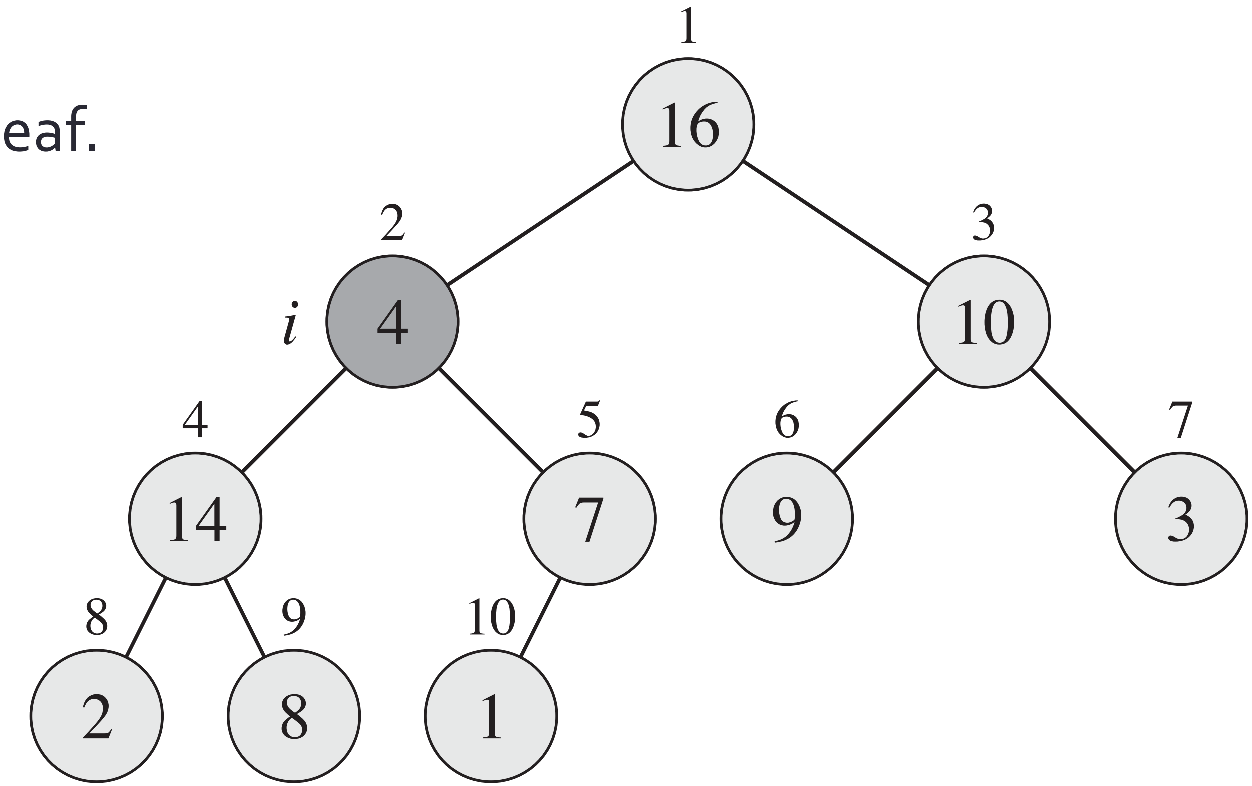
See Fig. 6.2 on p.155 of Cormen

Max-Heapify

Assumption: the left and right subtrees are max-heaps, but the root node might violate the max tree property.

Worst case:
need to go
all the way to the leaf.

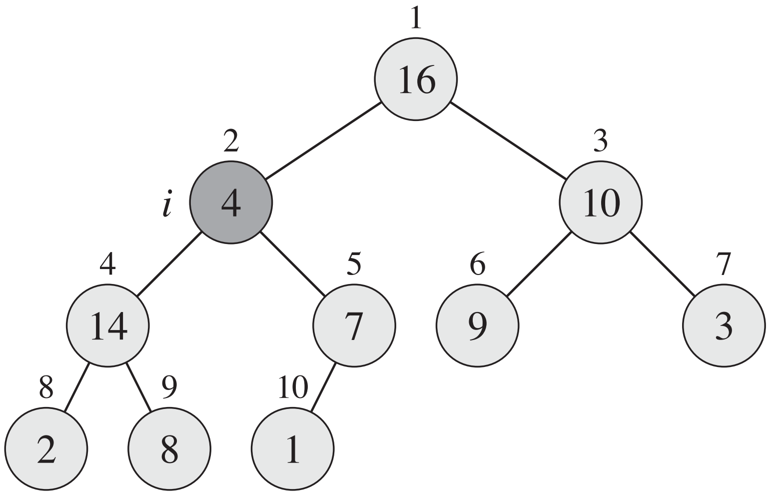
O(h)=O(log n)



See Fig. 6.2 on p.155 of Cormen

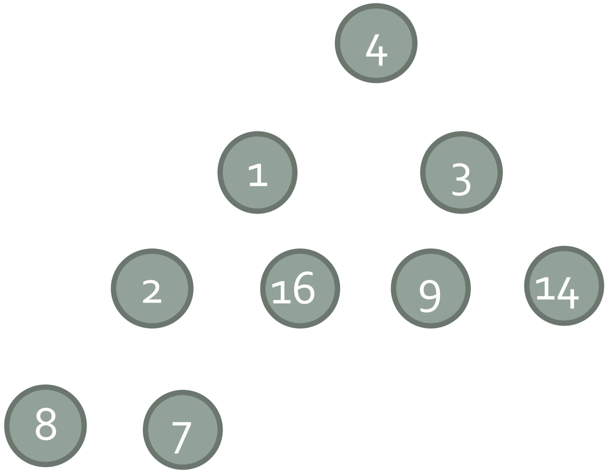
Max-Heapify

```
MAX-HEAPIFY(A, i)
    $l = \text{LEFT}(i)$
    $r = \text{RIGHT}(i)$
    if $l \le A.heap\text{-}size$ and $A[l] > A[i]$
        $largest = l$
    else $largest = i$
    if $r \le A.heap\text{-}size$ and $A[r] > A[largest]$
        $largest = r$
    if $largest \ne i$
        exchange $A[i]$ with $A[largest]$
        MAX-HEAPIFY$(A, largest)$
```



How to “heapify” an array?

Method:
Starting from the last node, each time build a small max-heap with that node as the root.



- Skip the leaves
  (already a “one-node heap”)
- Find the first non-leaf node
- Run max-heapify on that node, and afterwards the subtree with the node as the root will be a heap
- When we are done with the root node (of the entire tree), it will become a heap.

Build-Max-Heap

```
BUILD-MAX-HEAP(A)
    $A.heap\text{-}size = A.length$
    for $i = \boxed{\lfloor A.length/2 \rfloor \textbf{ downto } 1}$
        MAX-HEAPIFY$(A, i)$
```

Skipping the leaves

It can be shown that, with the array representation for storing an *n*-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.
(problem 6.1-7 on Cormen p.154)

Time complexity

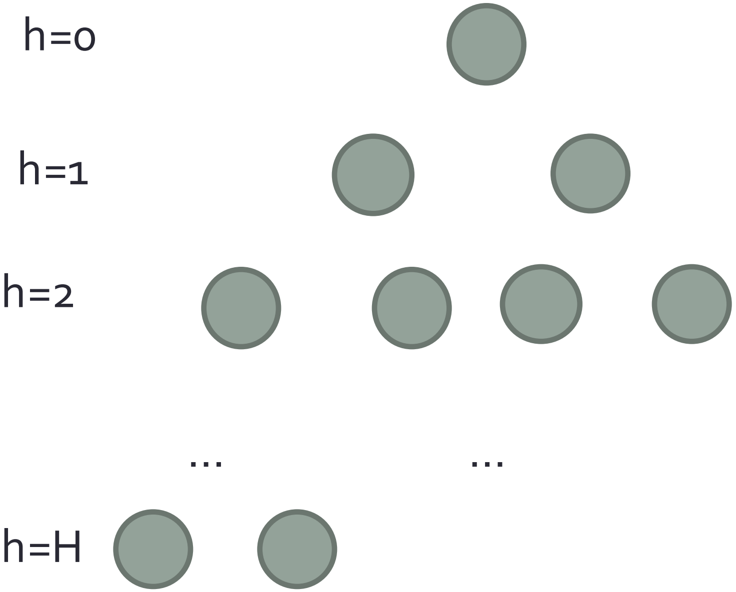
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

O(n) iterations

O(log n)

Therefore the time complexity is O(n log n).
This bound is correct, but not asymptotically tight.

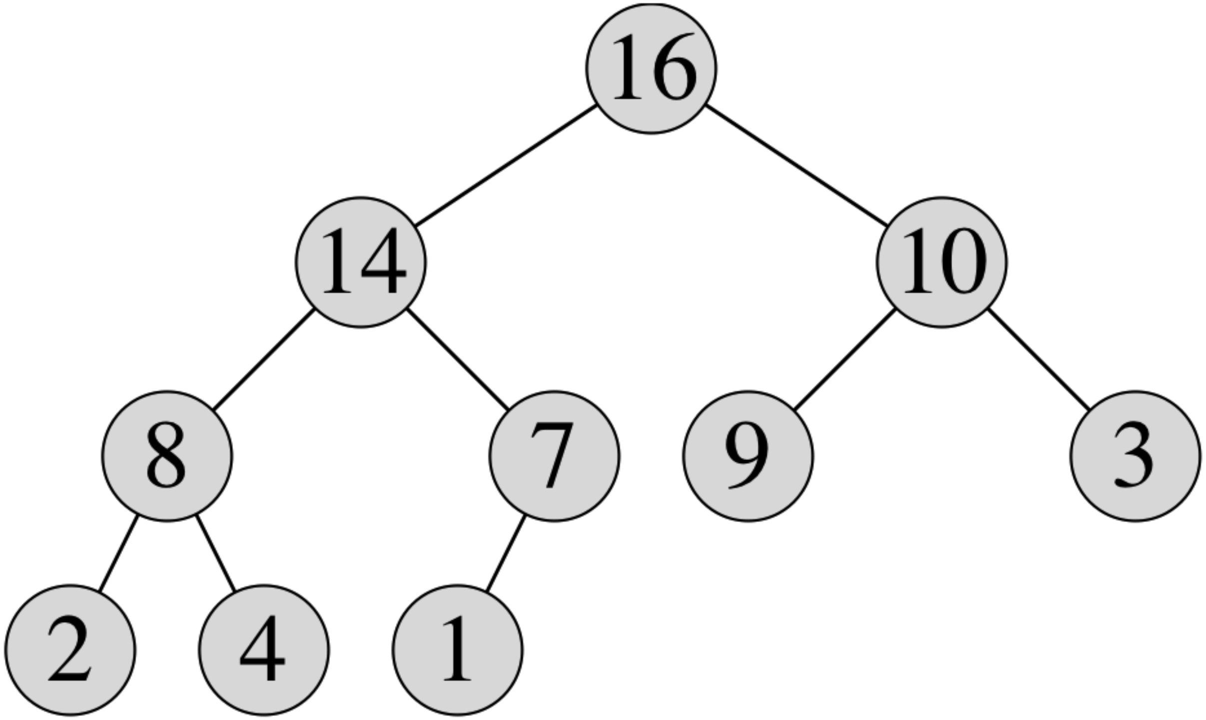
Time Complexity: Trial 2



- 所花的時間為:
- $h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1} \cdot 1$
- $= \sum_{i=0}^{h} 2^i \, (h-i)$
- Let $S = \sum_{i=0}^{h} 2^i \, (h-i)$.
- $2S = 2h + 4(h-1) + \cdots + 2^h$
- $2S - S = -h + 2 + 4 + \cdots + 2^h$
- $S = 2^{h+1} - h - 2$
- 又$h = O(\log n)$
- $2^{\lceil \log_2 n \rceil} - O(\log n) - 2$
- $\leq 2n - O(\log n)$
- $= O(n)$

You can also see a different proof on p.159 of Cormen.

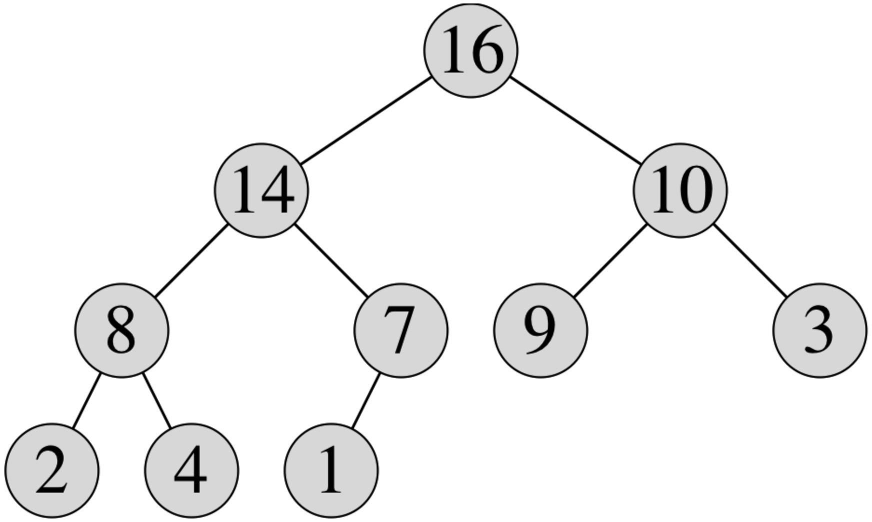
Heapsort: use a heap to sort



Heapsort: use a heap to sort

- How?

1. Use Build-Max-Heap to build a max-heap. $\boxed{O(n)}$
2. Exchange the root node (maximum element) with the last node.
3. Heapify again to maintain the max-heap property.
4. Repeat the above until the heap is empty. $\boxed{O(h) = O(\log n)}$



$\boxed{\text{Total: } O(n \log n)}$

See Cormen p.161 Figure 6.4)

比較四大金剛

	Worst	Average	Additional Space?
Insertion sort	$O(n^2)$	$O(n^2)$	O(1)
Merge sort	$O(n \log n)$	$O(n \log n)$	O(n)
Quick sort	$O(n^2)$	$O(n \log n)$	O(1)
Heap sort	$O(n \log n)$	$-$	O(1)

Covered today!

- Insertion sort: quick with small input size n. (small constant)
- Quick sort: Best average performance (fairly small constant)
- Merge sort: Best worst-case performance
- Heap sort: Good worst-case performance, no additional space needed!
- Real-world strategy: **a hybrid of insertion sort** + others. Use **input size** *n* to determine the algorithm to use.

Priority queue

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.
- It supports:
- Insert(S,x) inserts the element x into the set S.
- Maximum(S): returns the element of S with the largest key.
- Extract-Max(S): removes and returns the element of S with the largest key
- Increase-Key(S,x,k) increases the value of element x's key to the new value k, which is at least as large as x's current key value.

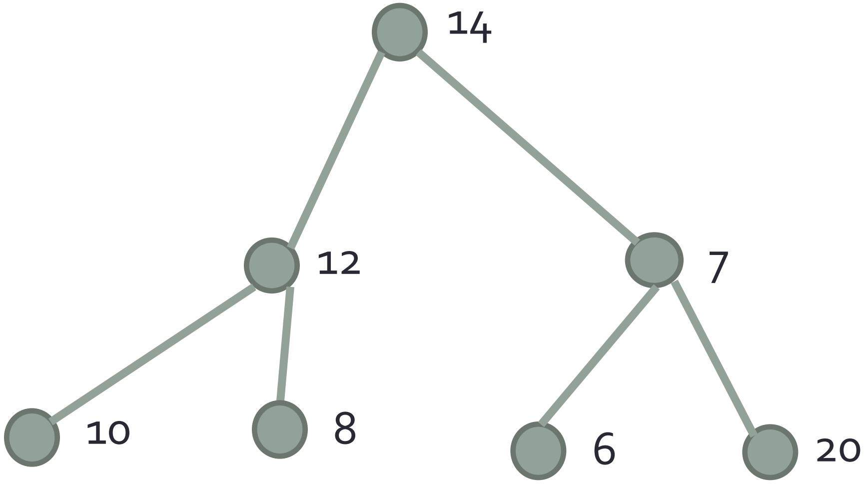
We already know how to do these!

See p. 164 on Cormen for the implementation of Increase-Key(). (very similar to insert)

Inserting a new element

- 1. Add the new element after the last leaf (it is always a complete binary tree)
- 2. Compare the value of the element with its parent's value. If it violates the max-heap property, then exchange the two then repeat 2. again.
- Time complexity?

$\boxed{O(\log n)}$



Need to compare with 6? No! Because 6's parent is already larger. Therefore 20 will be even larger.

Summary – Priority Queue using Heap

Operation	Running time
Maximum	O(1)
Extract Maximum	O(log n)
Insert	O(log n)
Increase-Key	O(log n)

All operations can be completed in O(log n) time!

Related Reading

- Cormen chapter 6