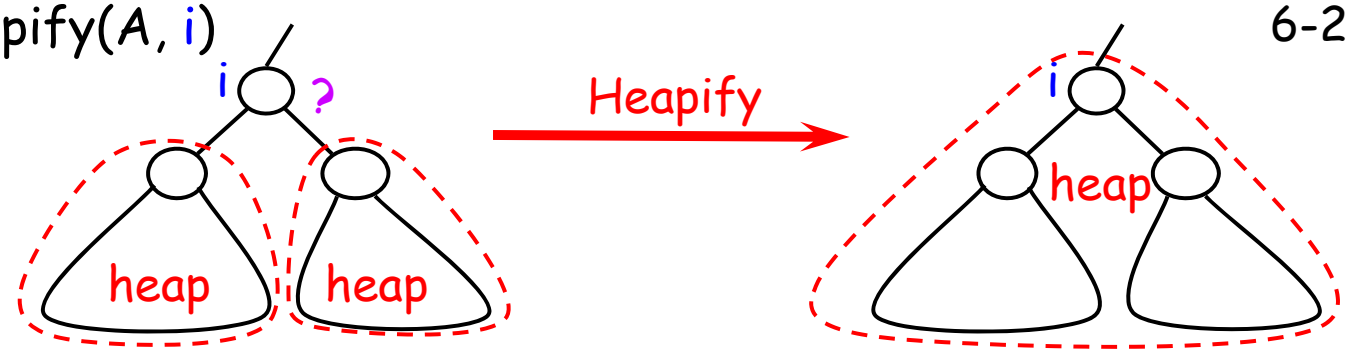
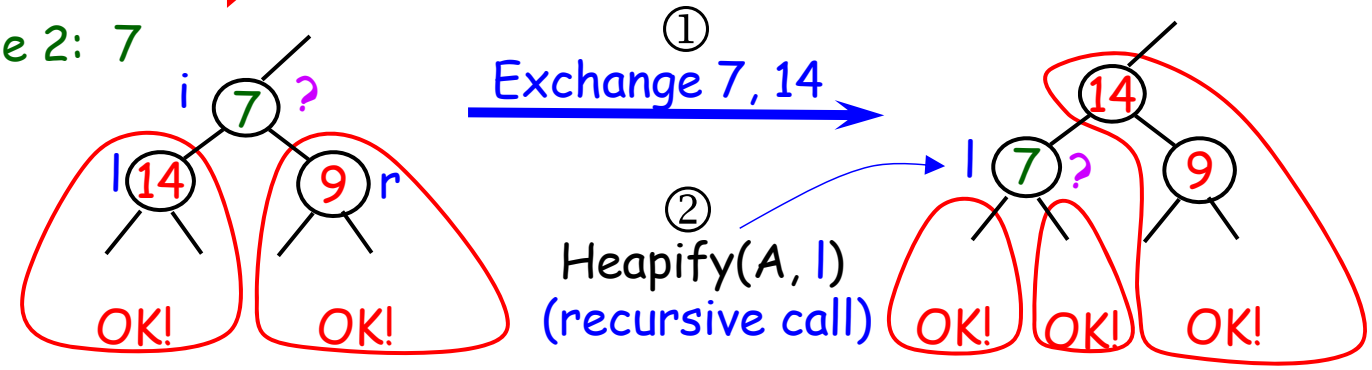
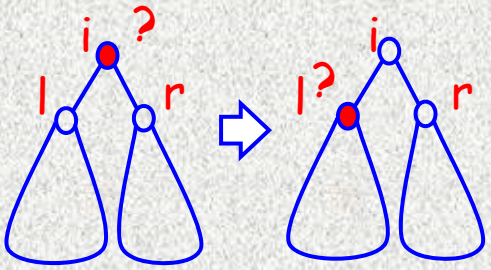
Heapify(A, i)



Case 1: 19 ⇨ Done!

Case 2: 7





MAX-HEAPIFY(*A*, *i*)

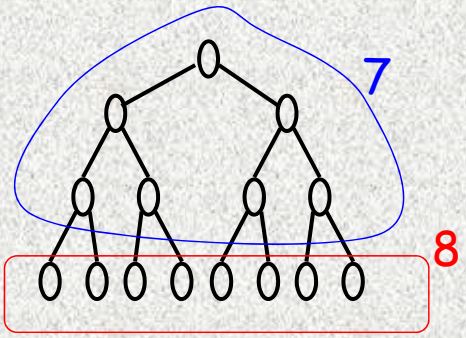
```
1   l ← LEFT(i)
2   r ← RIGHT(i)
3   if l ≤ heap-size[A] and A[l] > A[i]
4         then largest ← l
5         else largest ← i
6   if r ≤ heap-size[A] and A[r] > A[largest]
7         then largest ← r
8   if largest ≠ i
9         then exchange A[i] ↔ A[largest]
10        MAX-HEAPIFY(A, largest)
```

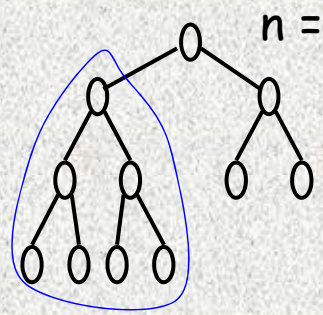
else ✗

c operations
||
O(c) time
||
O(1) time
||
constant time
⇩
1 秒 鐘
(1 單 位 時 間)

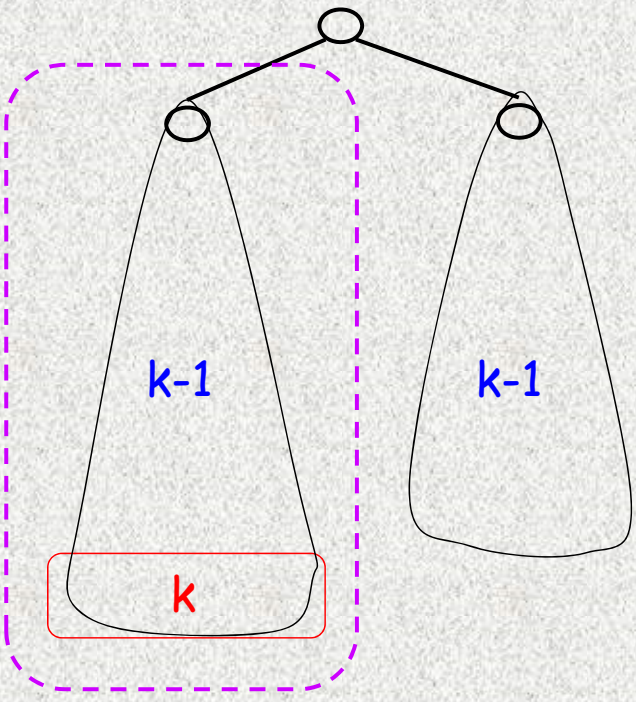
$T(n) \le T(\frac{2}{3}n)$

$T(n) \le T(\frac{2}{3}n) + O(1)$





7 ≈ 2n/3



n = 3k -1

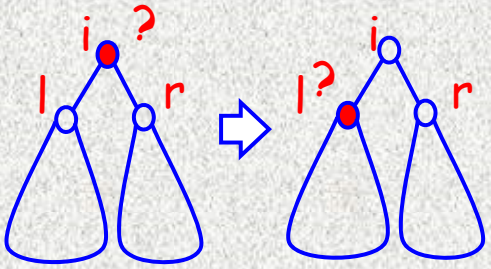
n' = 2k-1 ≈ 2n/3

6-3y

MAX-HEAPIFY(A, i)

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ heap-size[A] and A[l] > A[i]
4        then largest ← l
5        else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7        then largest ← r
8  if largest ≠ i
9        then exchange A[i] ↔ A[largest]
10             MAX-HEAPIFY(A, largest)
```

c operations
||
O(c) time
||
O(1) time
||
constant time
⇩
1 秒 鐘



$T(n) \le T(\frac{2}{3}n)$

$T(n) \le T(\frac{2}{3}n) + O(1)$

6-3x

Build_Heap: n/2 calls to heapify

Roughly: T(n) = O(n lg n) (true, but overestimated)

Carefully: T(n) = O(n)

1 x lg n

2 x lg n -1

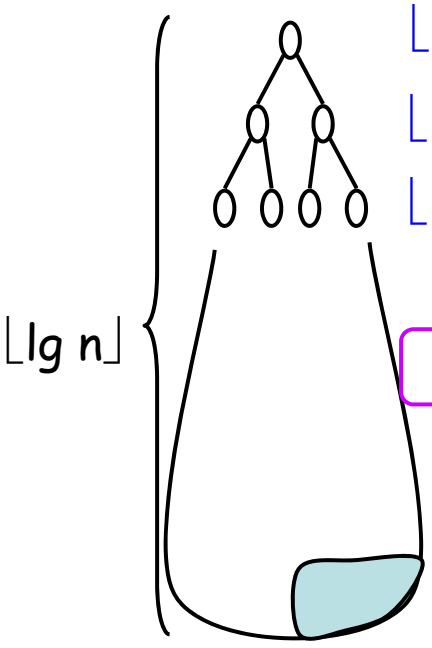
⋮

n/4 x 1

n/2 x 0

most nodes needs O(1) time

6-3a



height	nodes
$\lfloor \lg n \rfloor$	1
$\lfloor \lg n \rfloor - 1$	2
$\lfloor \lg n \rfloor - 2$	2^2
⋮	⋮
h	$2^{\lfloor \lg n \rfloor - h}$
⋮	⋮
1	$2^{\lfloor \lg n \rfloor - 1}$
0	$2^{\lfloor \lg n \rfloor}$

at most

at most

課 李 用 $\lceil \frac{n}{2^{h+1}} \rceil$

比 較 準，但 沒 好 處

$$\lfloor \lg n \rfloor \sum_{h=0} 2^{\lfloor \lg n \rfloor - h} O(h)$$

$$\leq \sum \frac{n}{2^h} O(h)$$

$$\leq O(n \times \sum \frac{h}{2^h})$$

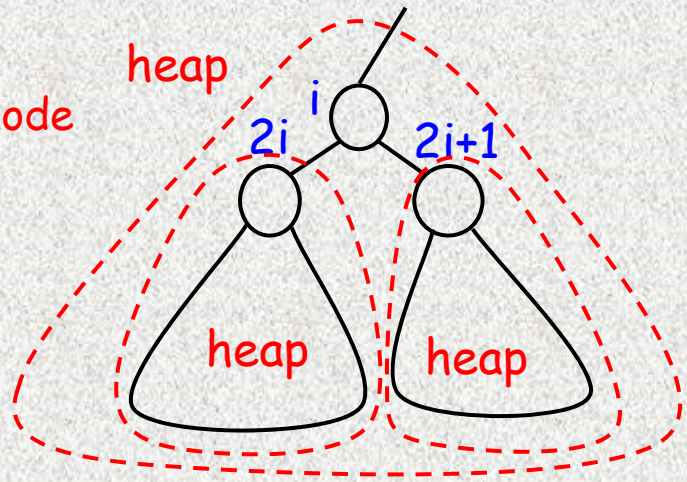
$$\leq O(n \times 2)$$

$$\sum kx^k = \frac{x}{(1-x)^2}$$

Building a heap: a top-down viewpoint (D&C)

```
Build_H(A, i)
    if i > n then return
    Build_H(A, 2i)
    Build_H(A, 2i+1)
    Heapify(A, i)
end
```

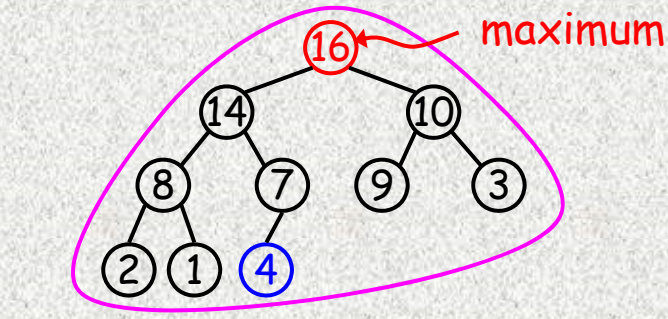
i isn't a tree node



call Build_H(A, 1) to build the whole heap

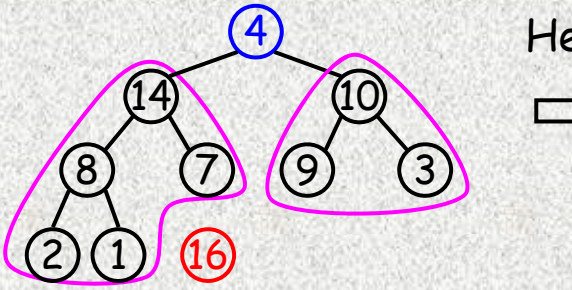
$T(n) = 2T(n/2) + O(\lg n) = O(n)$ (by Master Thm.)

6-3w



16	14	10	8	7	9	3	2	1	4

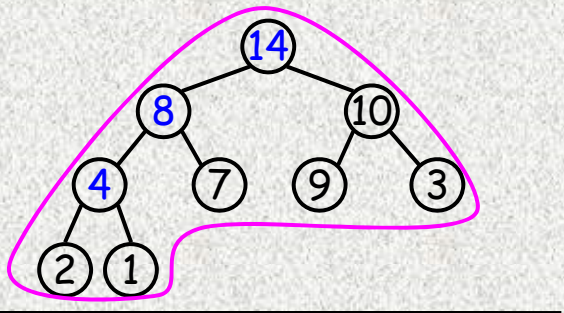
n = 10 (a)



4	14	10	8	7	9	3	2	1	16

n = 9 (b)

Heapify(A, 1)



14	8	10	4	7	9	3	2	1	16

n = 9 (c)

6-6x

Build_Heap: n/2 calls to heapify

Roughly: $T(n) = O(n \lg n)$ (true, but overestimated)

Carefully: $T(n) = O(n)$

$1 \times \lg n$

$2 \times \lg n - 1$

$n/4 \times 1$

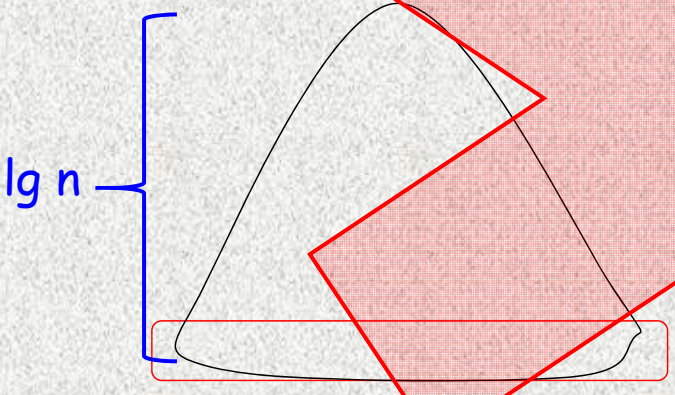
$n/2 \times 0$

most nodes needs $O(1)$ time

Stage 2 of Heapsort: n-1 calls to heapify

Roughly: $T(n) = O(n \lg n)$

Carefully: $T(n) = O(n \lg n)$



1×0

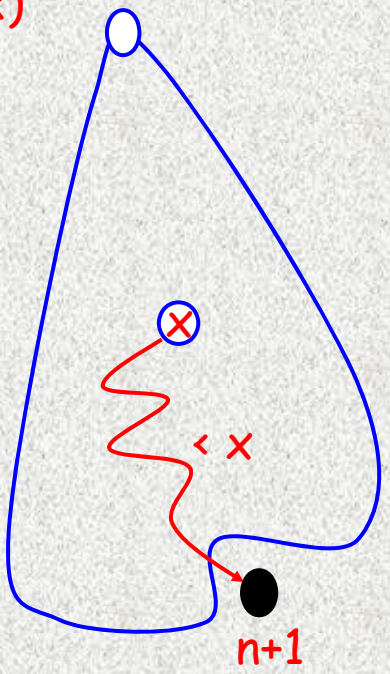
2×1

$n/4 \times \lg n - 1$

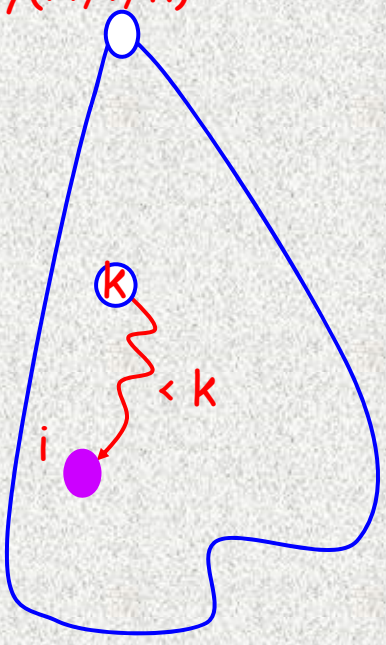
$n/2 \times \lg n$

most nodes needs $O(1g\ n)$ time

Insert(A, x)

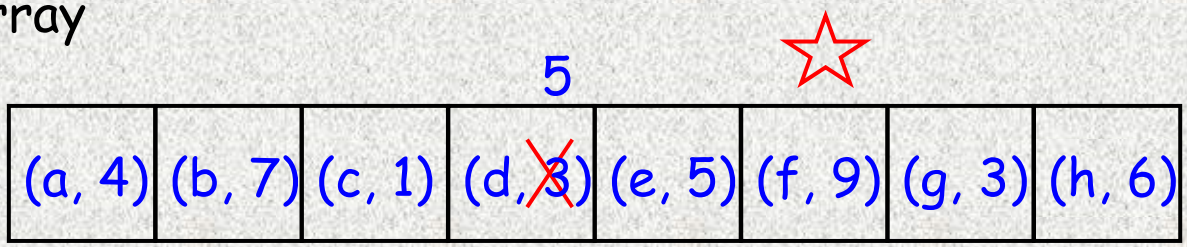


Increase-Key(A, i, k)



	binary heap	array
Build	O(n)	O(n)
Insert	O(1g n)	O(1)
Maximum	O(1)	O(n)
Increase-Key	O(lg n)	O(1)
Extract-Max	O(lg n)	O(n)

An array



Increase-Key (d, 3) -> (d, 5)

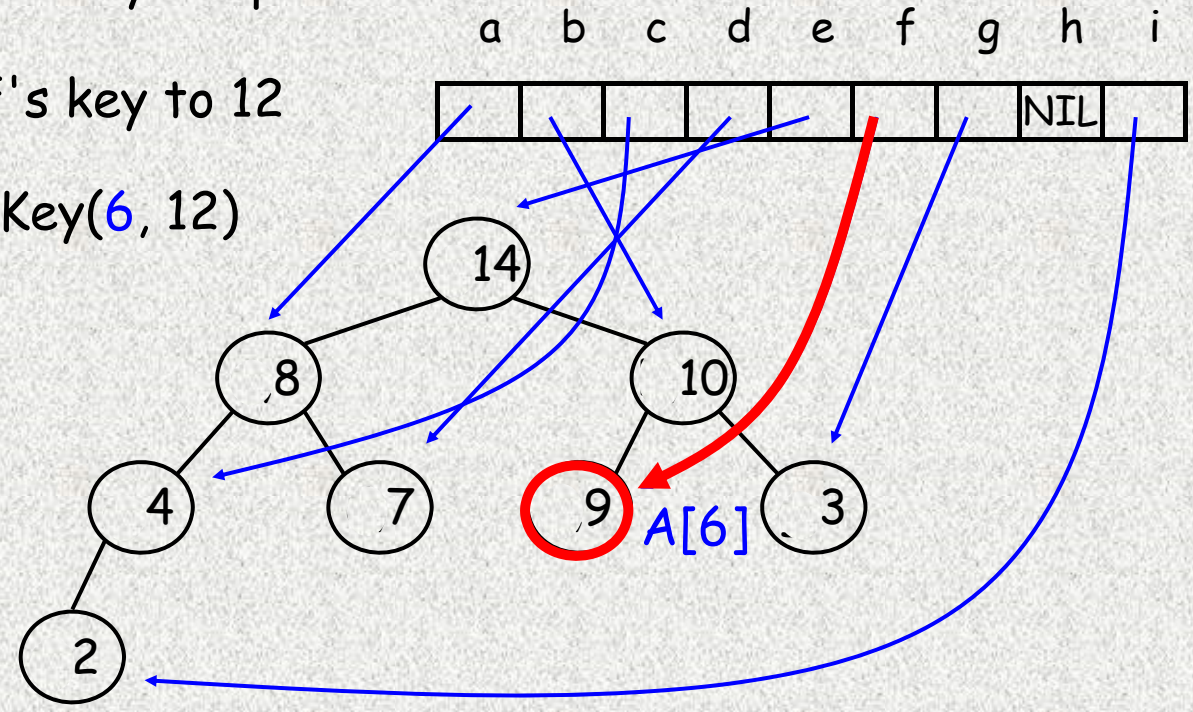
Extract-Max

handles: pointers to the objects in a data structure

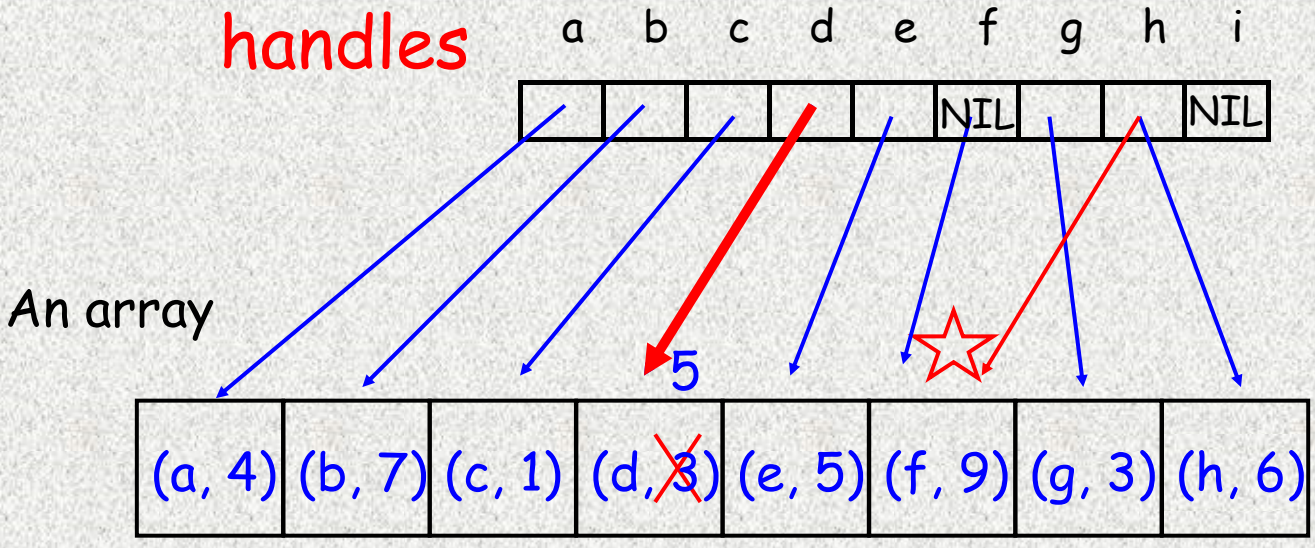
Example: a binary heap

increase f's key to 12

Increase-Key(6, 12)



handles



Increase-Key (d, 3) -> (d, 5)

Extract-Max

Usually, we omit the maintenance of handles, since it is simple and the cost is insignificant.