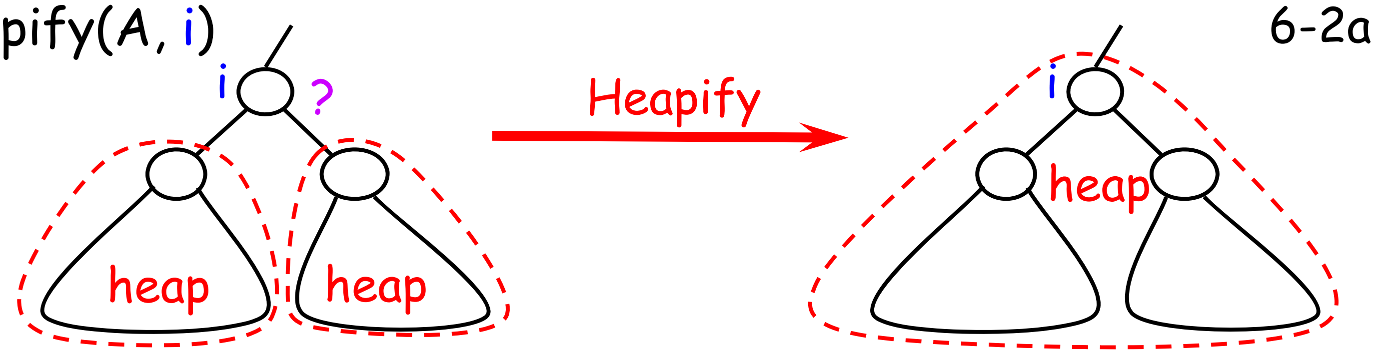
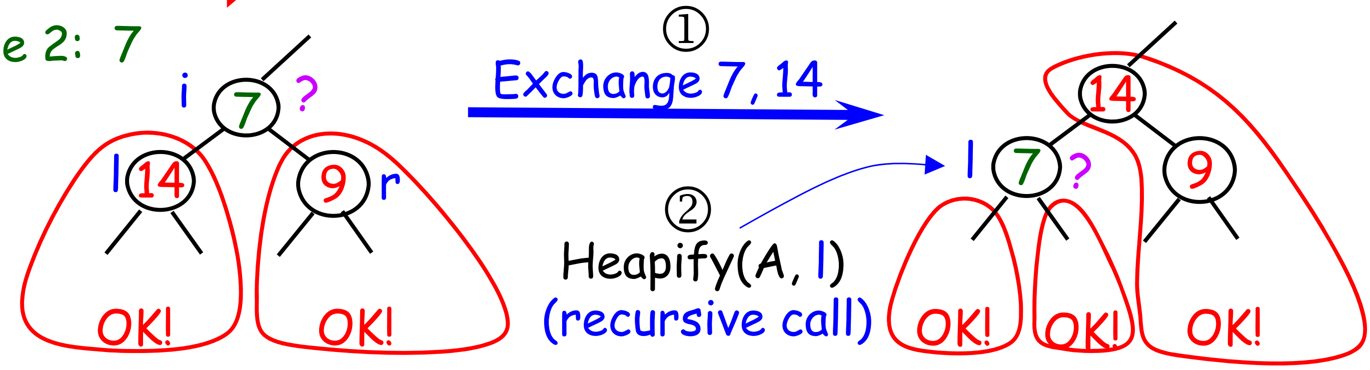
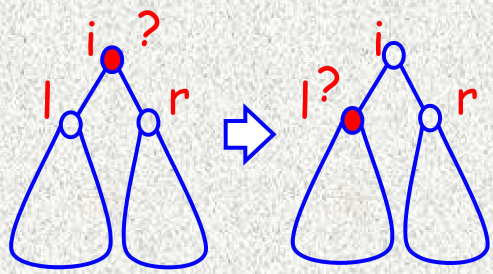
Heapify(A, i)



Case 1: 19 ⇨ Done!

Case 2: 7





MAX-HEAPIFY(A, i)

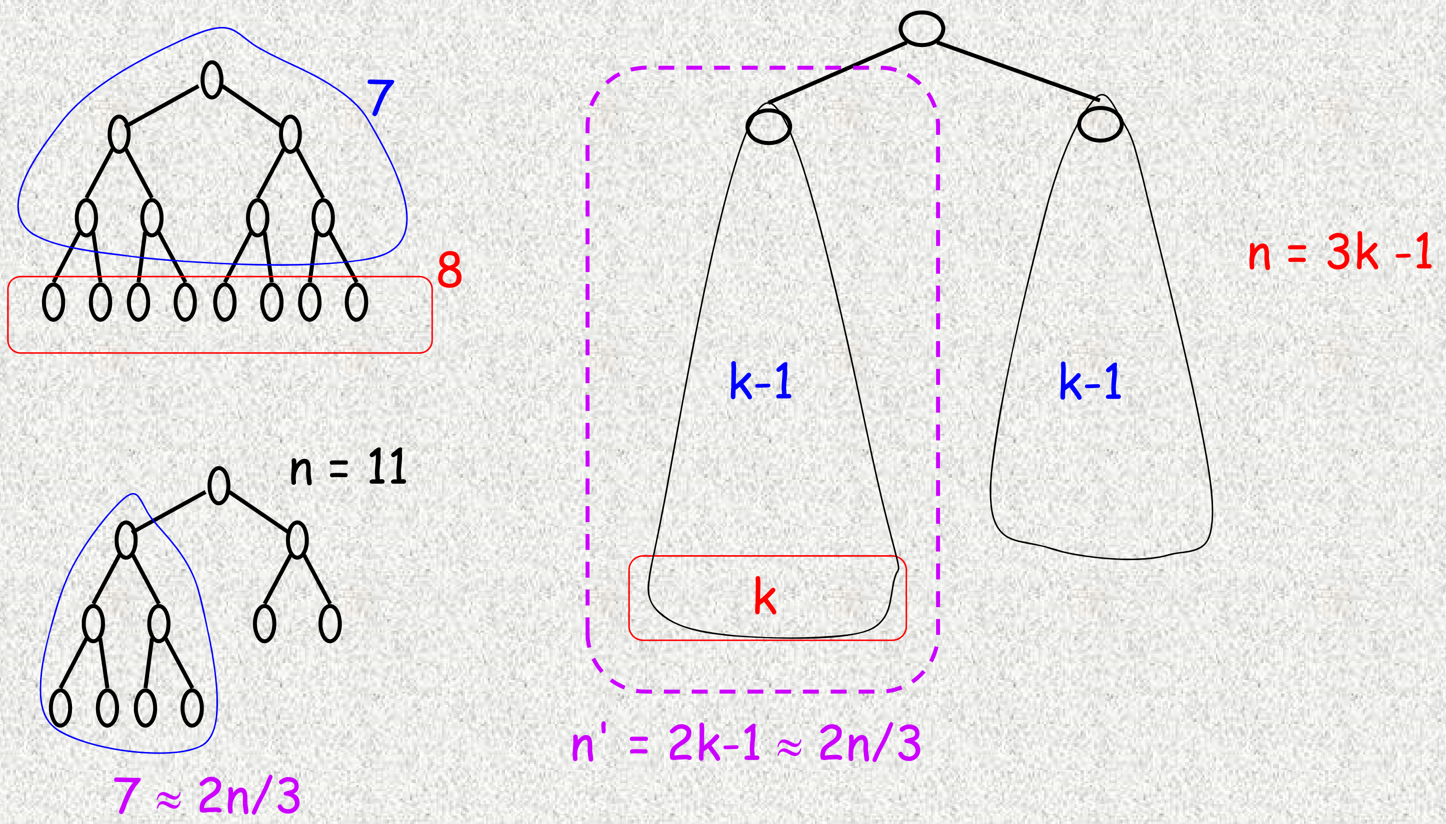
```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
         MAX-HEAPIFY(A, largest)
```

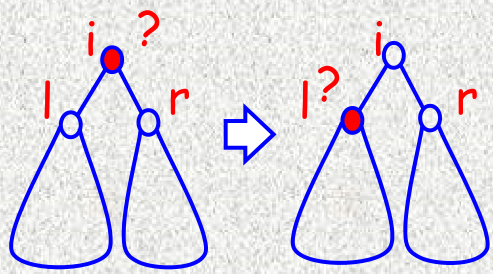
c operations
||
O(c) time
||
O(1) time
||
constant time
⇩
1 秒鐘
(1 單位時間)

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

~~else~~

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





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

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

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

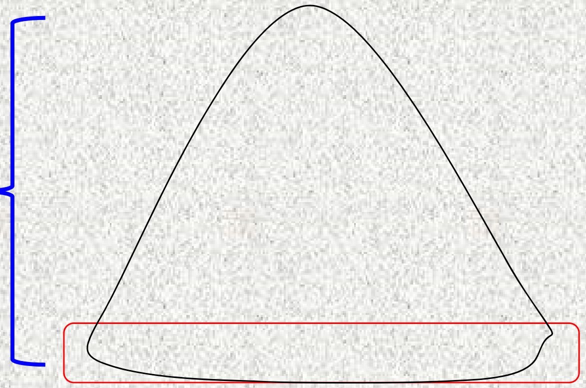
$$T(n) \le T(\frac{2}{3}n) + O(1)$$

Build_Heap: n/2 calls to heapify, each taking O(lg n)

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

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

lg n



$1 \times \lg n$

$2 \times \lg n - 1$

⋮

$n/4 \times 1$

$n/2 \times 0$

most nodes needs O(1) time

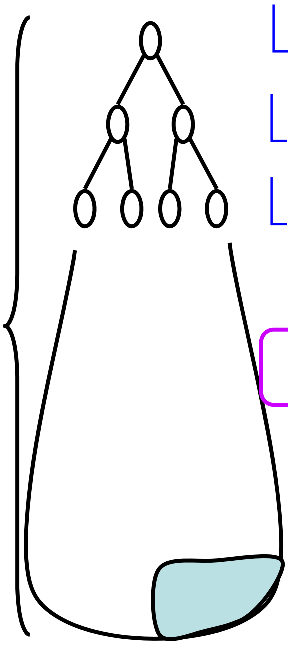
6-3z

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

$\lfloor \lg n \rfloor$



課本用$\left\lceil \frac{n}{2^{h+1}} \right\rceil$比較準，但沒好處

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

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

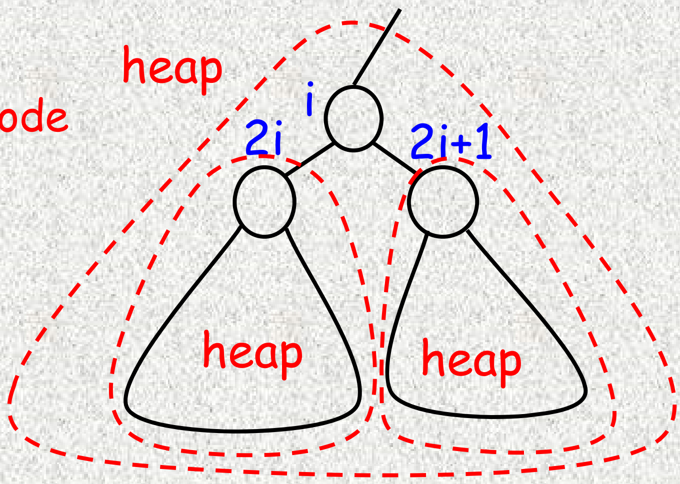
$$\le O\left(n \times \sum \frac{h}{2^h}\right)$$

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

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

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

```
Build_H(A, i)                 i isn't a tree node
    if i > n then return
    Build_H(A, 2i)
    Build_H(A, 2i+1)
    Heapify(A, i)
end
```

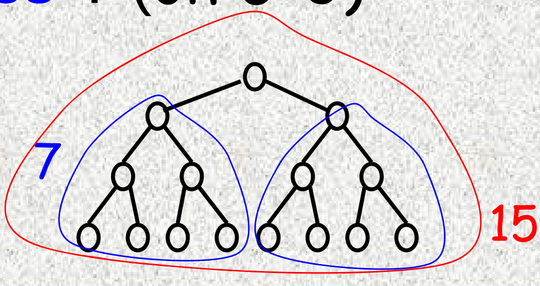


call Build_H(A, 1) to build the whole heap

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

OR: $T(n)=2T(n/2)+\lg n$ (Assume $n = 2^{h+1} - 1$.)
$\lfloor\ \rfloor$ Append dummy nodes

Q: Why "Append dummy nodes"? (on 6-3)



* When $n = 2^{h+1} - 1$, we can always take $\lfloor n/2 \rfloor$, making the recurrence simpler

* Why can we make such assumption?

n=13

7	1	4	3	7	5	9	2	6	11	8	12	4	$-\infty$	$-\infty$

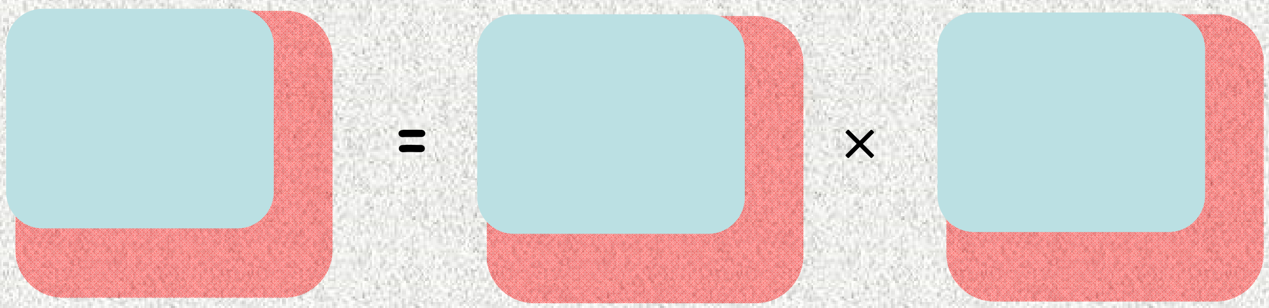
(for max-heap)

$$\frac{n}{2}\times\frac{n}{2} \quad \overset{C}{\begin{pmatrix} r & s \\ t & u \end{pmatrix}} = \overset{A}{\begin{pmatrix} a & b \\ c & d \end{pmatrix}} \overset{B}{\begin{pmatrix} e & g \\ f & h \end{pmatrix}} \quad \cdots\cdots \text{(EQ-1)}$$

Q: Why can we have the following ? (on 4-7)

* Assume that n is an exact power of 2

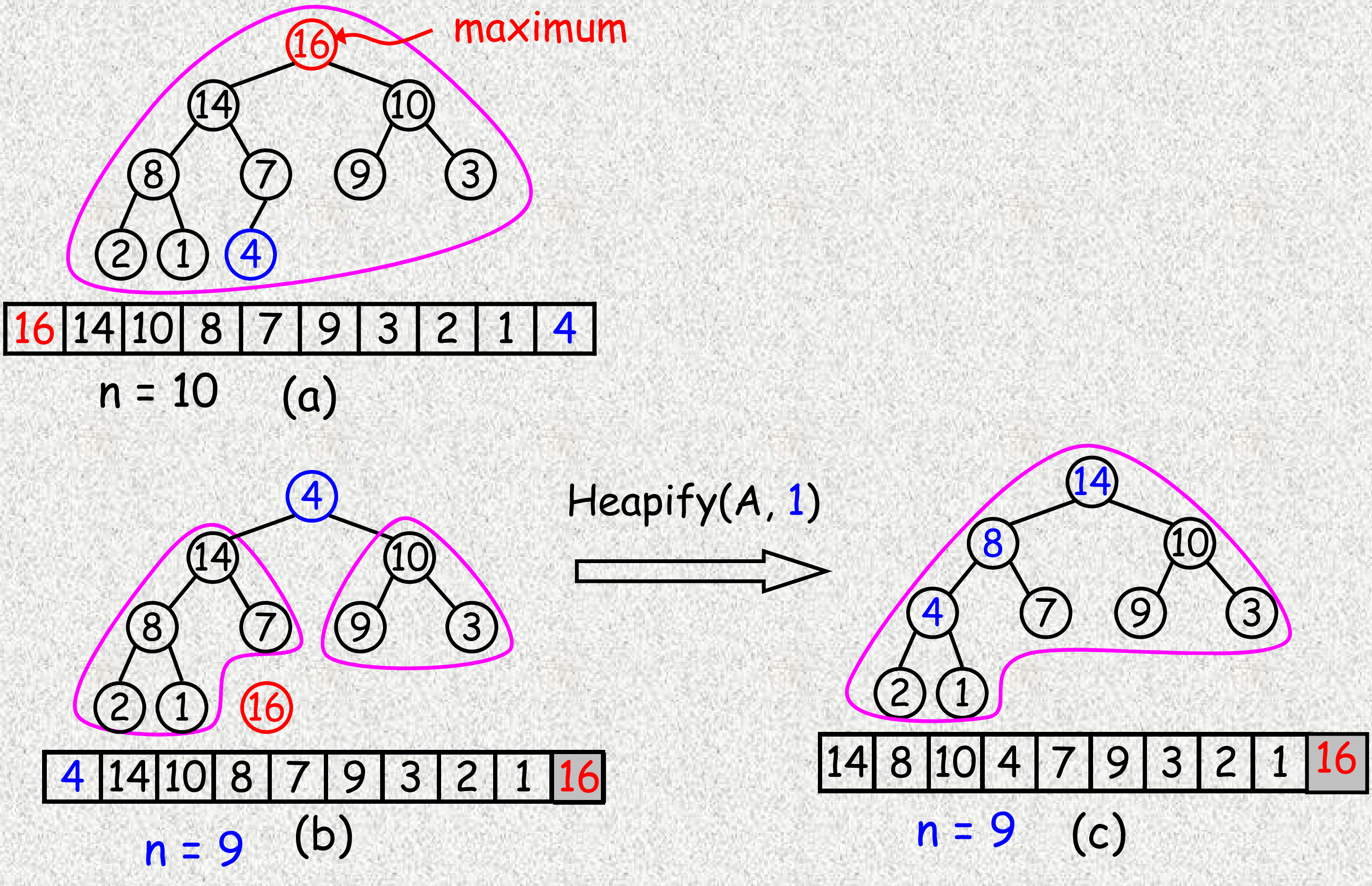
*fill ??? in dummy entries



To make the analysis of an algorithm easier

* we can assume that n is of a specific form

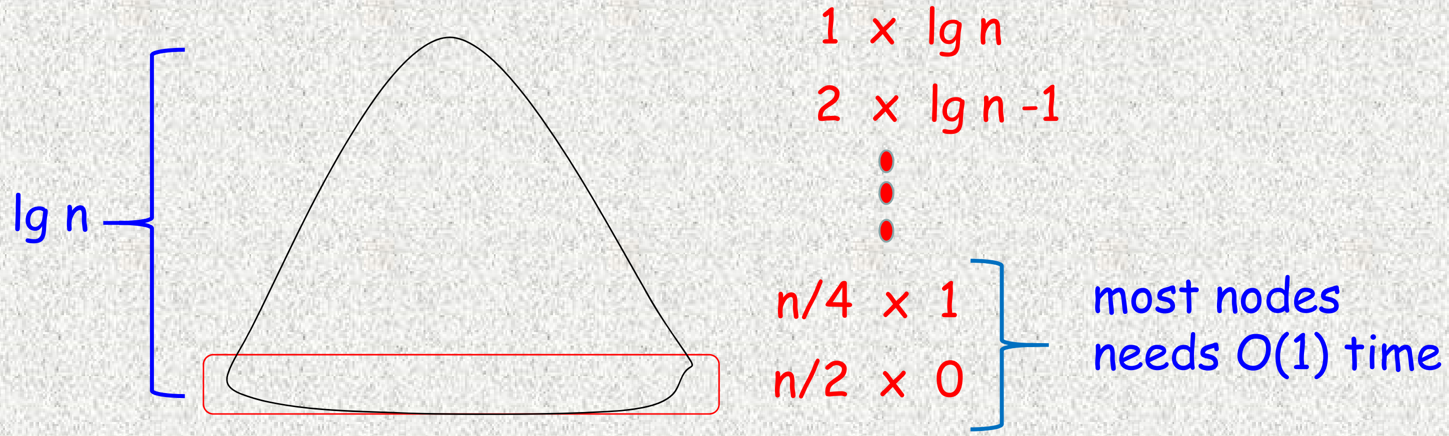
* however, you must explain why the assumption is reasonable



Build_Heap: n/2 calls to heapify

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

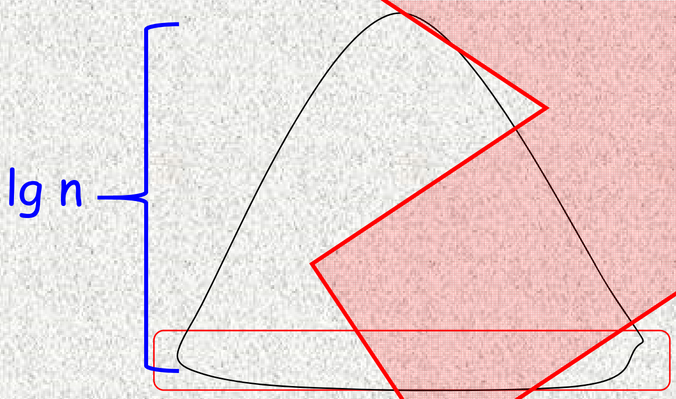
Carefully: T(n) = O(n)



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

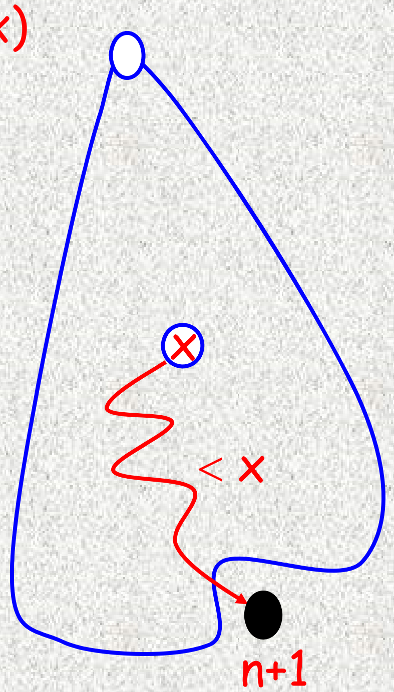
⋮

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

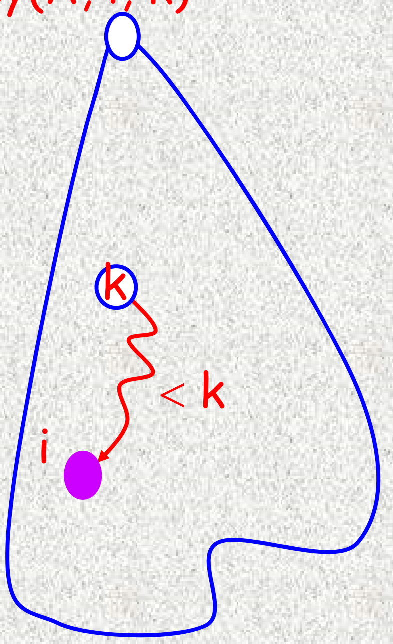
$n/2 \times \lg n$

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

			5				
(a, 4)	(b, 7)	(c, 1)	(d, ~~3~~)	(e, 5)	(f, 9)	(g, 3)	(h, 6)

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

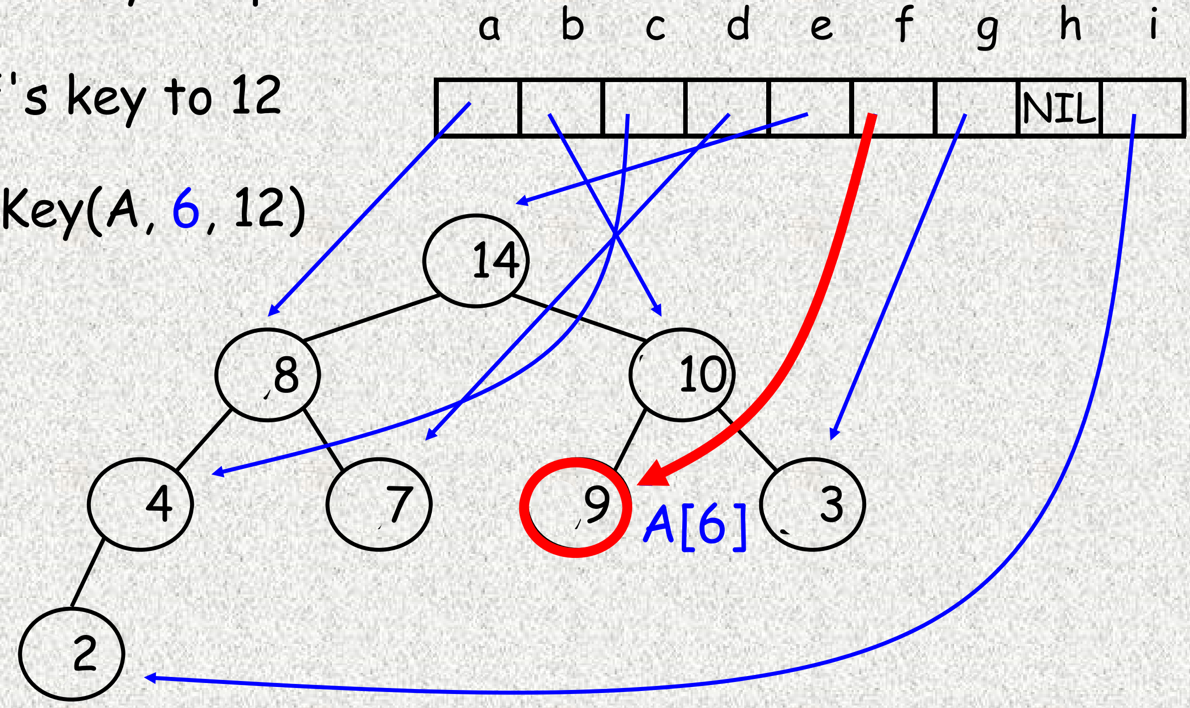
Extract-Max

handles: pointers to the objects in a data structure

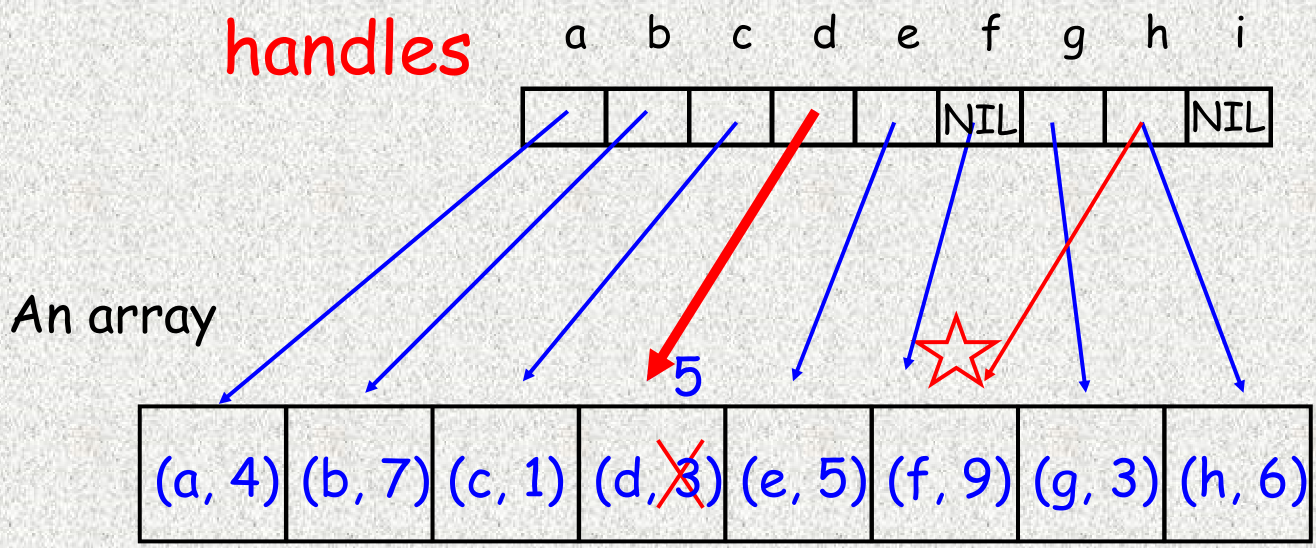
Example: a binary heap

increase f's key to 12

Increase-Key(A, 6, 12)



handles



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

Extract-Max

Usually, we omit the maintenance of handles, which may be implemented by table-lookup, hashing, or a search tree.