Algorithms
Chapter 6 Heapsort

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

Outline

- **Heaps**
- Maintaining the heap property 維護性質
- Building a heap 建立 heap
- The heapsort algorithm
- Priority queues 建立優先權佇列

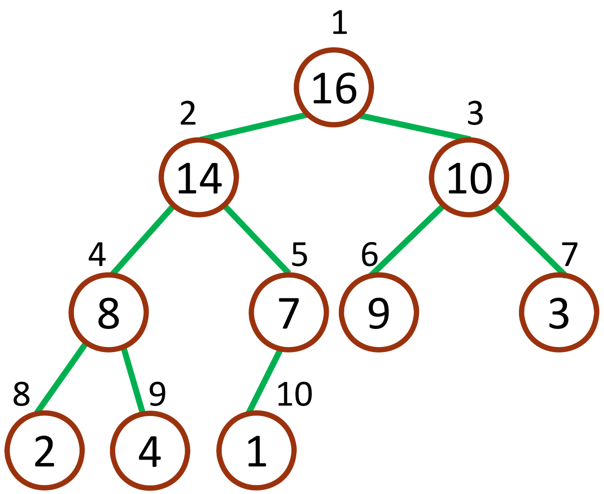
The purpose of this chapter

- In this chapter, we introduce the **heapsort** algorithm.
  - with worst case running time $O(n \lg n)$ 最差情形
  - an **in-place** sorting algorithm: only a constant number of array elements are stored outside the input array at any time.
    只有少數資料會放在 input array 以外的地方(在任何時間)
  - thus, require at most $O(1)$ additional memory
    所以,只需要 O(1) 的額外空間
- We also introduce the **heap** data structure.
  - an useful data structure for heapsort
  - makes an efficient priority queue

2個用處 用在 heap sort
用在 priority queue

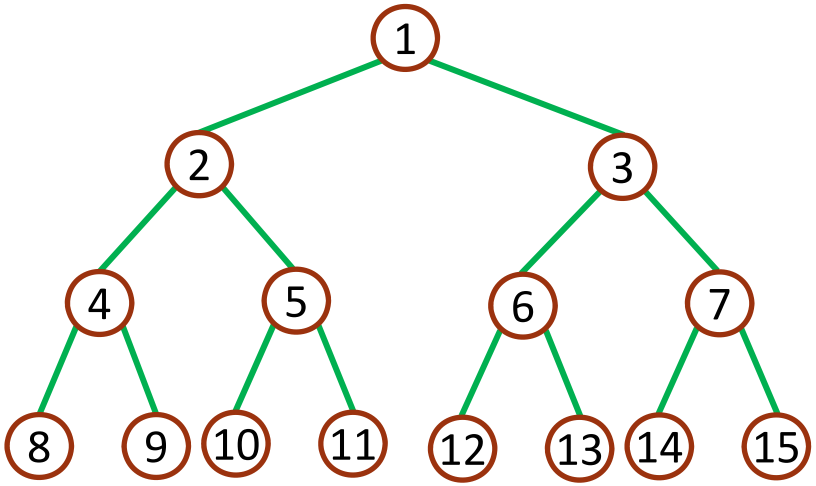
Heaps 事實上是個array，但我們將它看成complete binary tree

- The (**Binary**) **heap** data structure is an **array** object that can be viewed as a nearly complete binary tree.
  - A binary tree with *n* nodes and depth *k* is **complete** iff its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*. n個點，深度為k的二元樹為complete ⇔這n個點與深度為k的full binary tree的點相對應



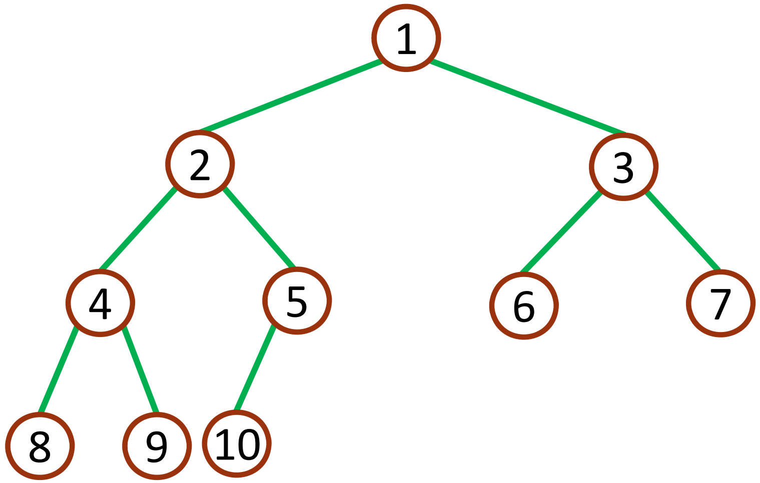
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Binary tree representations



A full binary tree of height 3.

root → leaf 經過的edge數

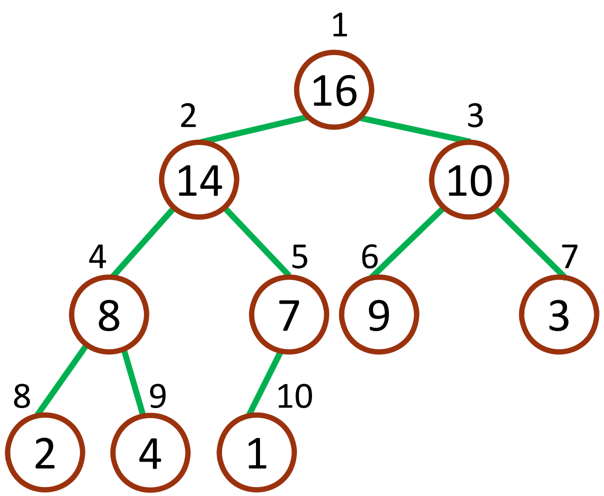


A complete binary tree with 10 nodes and height 3.

array表示heap時需要2個變數 (I) length[A]
(II) heap-size[A]

Attributes of a Heap

- An array A that presents a heap with two attributes:
  - **length[*A*]:** the number of elements in the array.
  - **heap-size[*A*]:** the number of elements in the heap stored with array *A*.
  - **length[*A*] ≥ heap-size[*A*]** array 的大小要比 heap-size 大

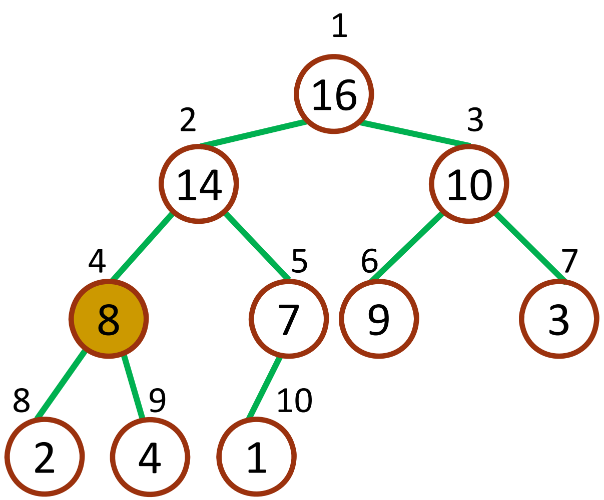


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

length[A] = heapsize[A] = 10

Basic procedures$_{1/2}$ 對於 complete binary tree有此性質

- If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have
  - A[1] is the **root** of the tree
  - the parent **PARENT(i)** is at $\lfloor i/2 \rfloor$ if $i \neq 1$ 父節點的 index = ⌊子節點/2⌋
  - the left child **LEFT(i)** is at $2i$ 左子節點 index = 父節點 index × 2
  - the right child **RIGHT(i)** is at $2i+1$ 右子節點 index = 父節點 index × 2 + 1



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

$\lfloor \frac{i}{2} \rfloor$ 0000 0010 2
i 0000 0100 4
$2i$ 0000 1000 8
$2i+1$ 0000 1001 9

Basic procedures$_{2/2}$

- The **LEFT** procedure can compute 2*i* in one instruction by simply shifting the binary representation of *i* left one bit position.
  算左子節点時，往左移一個bit Ex: 0100 → 1000
- Similarly, the **RIGHT** procedure can quickly compute 2*i*+1 by shifting the binary representation of *i* left one bit position and adding in a 1 as the low-order bit.
  算右子節点時，往左移一個bit再加1 Ex: 0100 → 1001
- The **PARENT** procedure can compute $\lfloor i/2 \rfloor$ by shifting *i* right one bit position. 算父節点時往右移1 bit Ex: 0100 → 0010

Heap properties

heap 的種類 { Max heap / min heap }

- There are two kind of binary heaps: max-heaps and min-heaps.
  - In a **max-heap**, the **max-heap property** is that for every node *i* other than the root, Max-heap property: 父節点 ≥ 子節点

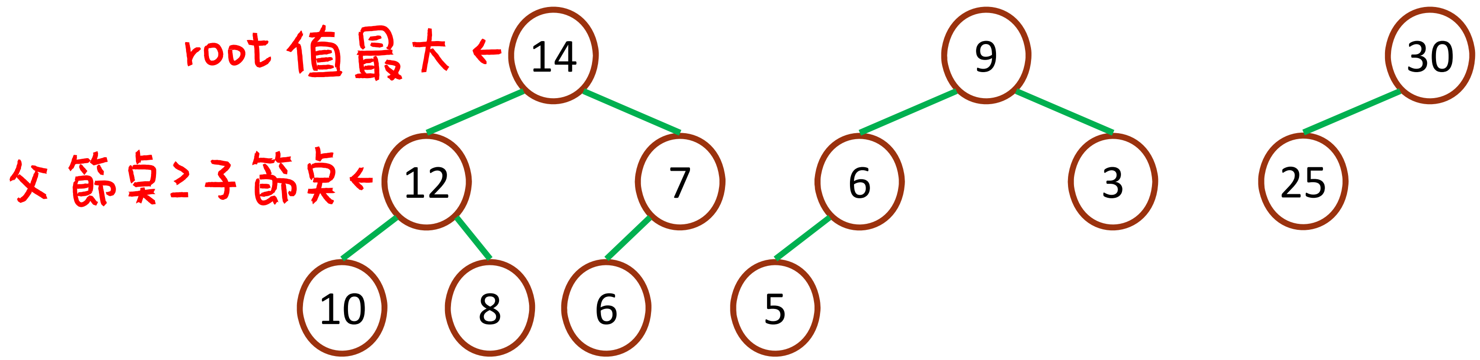
    $$A[\textsc{Parent}(i)] \geq A[i] .$$

    - the largest element in a max-heap is stored at the root 在 Max-heap 中, root 最大
    - the subtree rooted at a node contains values no larger than that contained at the node itself 對於每個子樹, 子樹的 root ≥ 子樹中其他点
  - In a **min-heap**, the **min-heap property** is that for every node *i* other than the root, min-heap property: 父節点 ≤ 子節点

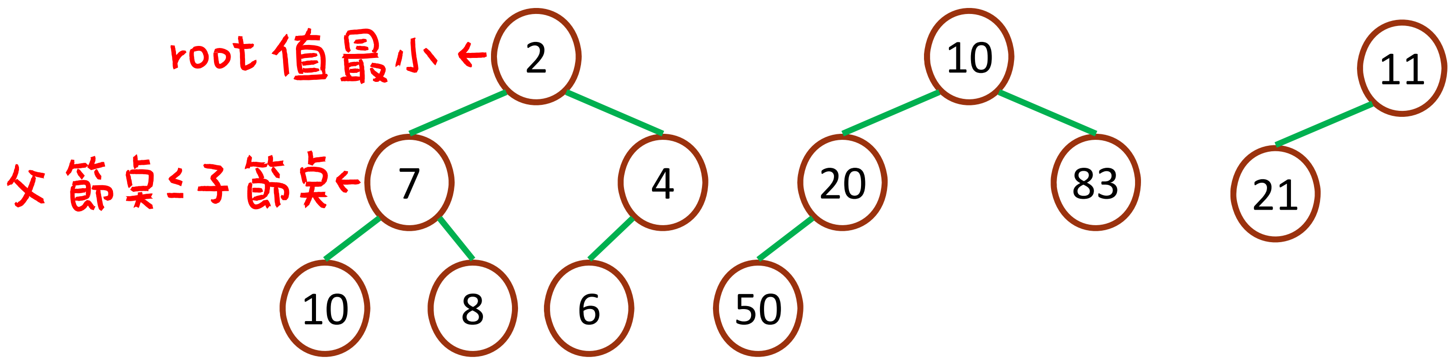
    $$A[\textsc{Parent}(i)] \leq A[i] .$$

    - the smallest element in a min-heap is at the root 在 min-heap 中, root 最小
    - the subtree rooted at a node contains values no smaller than that contained at the node itself 對於每一個子樹, 子樹的 root ≤ 子樹中其他点

Max and min heaps



Max Heaps 父節點必大於子節點



Min Heaps 父節點必小於子節點

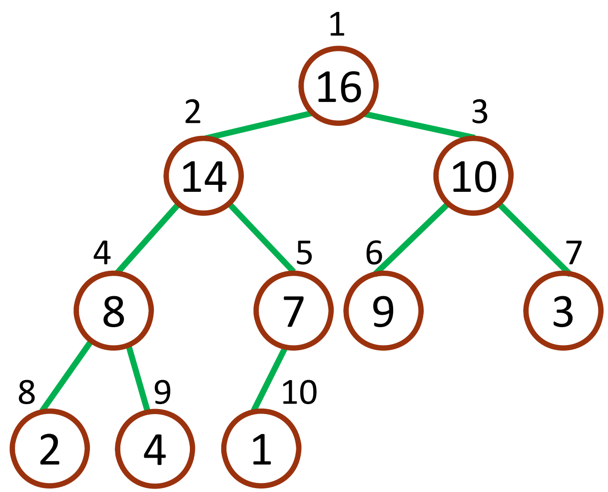
The height of a heap

node的高度為到leaf經過的最多edge數
heap的高度就是root的高度

- The **height** of a node in a heap is the number of edges on the longest simple downward path from the node to a leaf, and the height of the heap to be the height of the root, that is $\Theta(\lg n)$.

n個點的heap，高度為 $\lfloor \lg n \rfloor = \theta(\lg n)$
Ex: $\lfloor \lg 10 \rfloor = \lfloor 3.\sim \rfloor = 3$，heap的高度為3

- For example:
  - the height of node 2 is 2
  - the height of the heap is 3



↑ 高度為 3 的 heap

The remainder of this chapter

- We shall presents some basic procedures in the remainder of this chapter.
  - The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property. 維護性質
  - The **BUILD-MAX-HEAP** procedure, which runs in $O(n)$ time, produces a max-heap from an unordered input array. 建立 heap
  - The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
  - The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to be used as a priority queue.

Outline

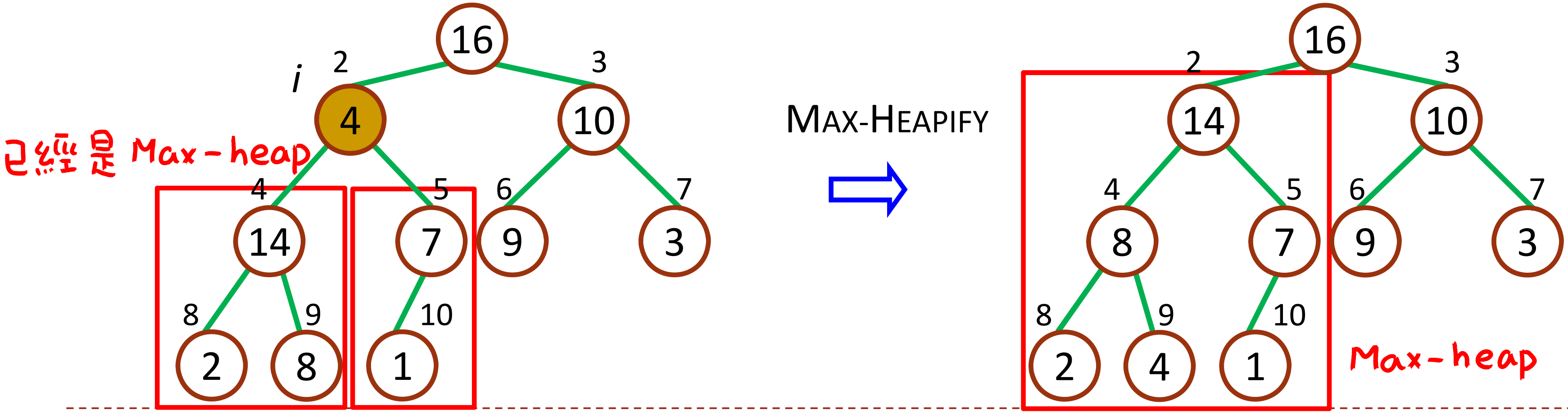
- Heaps
- **Maintaining the heap property** 維護heap的性質
- Building a heap
- The heapsort algorithm
- Priority queues

The MAX-HEAPIFY procedure$_{1/2}$

- MAX-HEAPIFY is an important subroutine for manipulating max heaps.
  - **Input**: an array *A* and an index *i*
  - **Output**: the subtree rooted at index *i* becomes a max heap

    以i為 root 的 subtree 為 max-heap
  - **Assume**: the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max-heaps, but *A*[i] may be smaller than its children
  - **Method**: let the value at *A*[i] "float down" in the max-heap

    向下滑動法



The MAX-HEAPIFY procedure$_{2/2}$

MAX-HEAPIFY(A, *i*)

$\ell \leftarrow$ LEFT(*i*) 左子

$r \leftarrow$ RIGHT(*i*) 右子

if $\ell \le$ *heap-size*[A] and A[ℓ] > A[*i*]

  then *largest* $\leftarrow \ell$

  else *largest* $\leftarrow i$

largest = max {ℓ, i}
跟左子相比

if $r \le$ *heap-size*[A] and a[*r*] > A[*largest*]

  then *largest* $\leftarrow r$

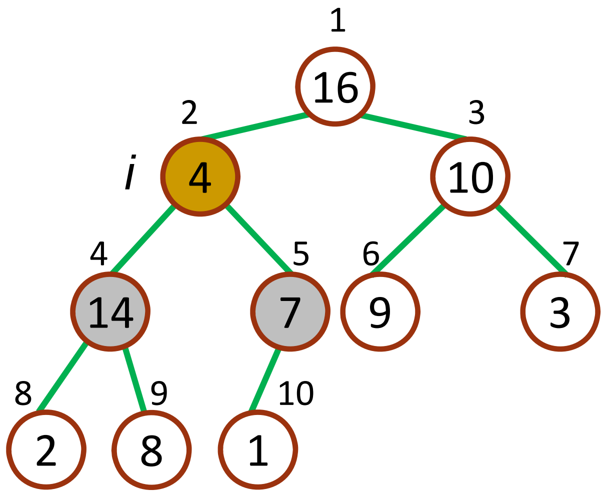
largest = max {largest, r}
跟右子相比

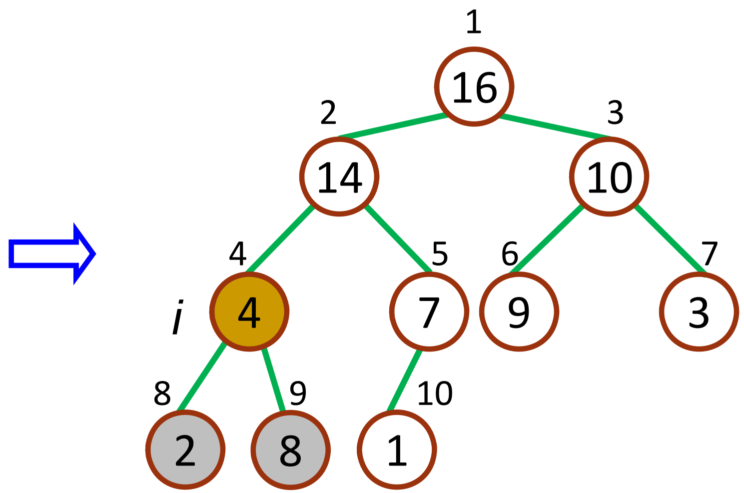
if *largest* $\ne i$ 如果子節点較大

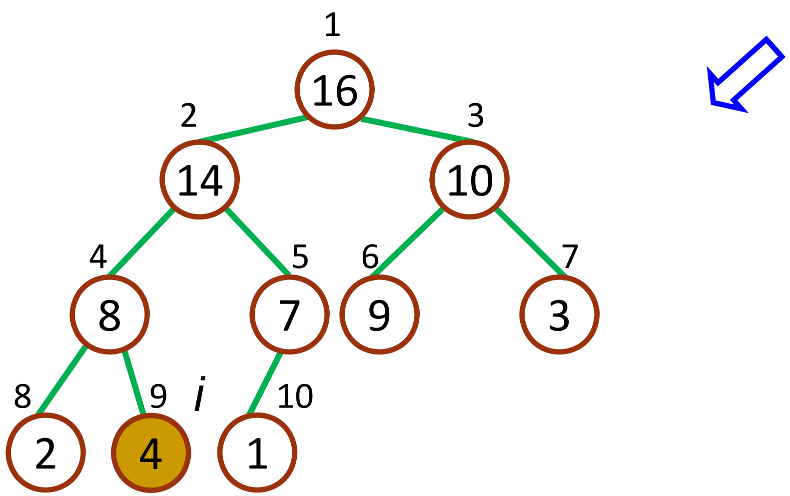
  then exchange A[*i*] $\leftrightarrow$ A[*largest*]

    MAX-HEAPIFY (A, *largest*)

An example of MAX-HEAPIFY procedure







i與左子右子比大小
與大者交換
向下滑動，把大的往上放

The time complexity

- It takes $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\textsc{Left}(i)]$, and $A[\textsc{Right}(i)]$.
  跟子節點比大小，做置換，花θ(1)時間
- We can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$, that is $O(\lg n)$. 以高度來看O(h)

Outline

- Heaps
- Maintaining the heap property
- **Building a heap** 建立 heap
- The heapsort algorithm
- Priority queues

Building a Heap

- We can use the MAX-HEAPIFY procedure to convert an array A=[1..n] into a max-heap in a **bottom-up** manner.

用Max-heapify由下往上建

- The elements in the subarray A[(⌊n/2⌋+1)…n] are all **leaves** of the tree, and so each is a 1-element heap.

$\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2$ … n 皆為leaf，不用比

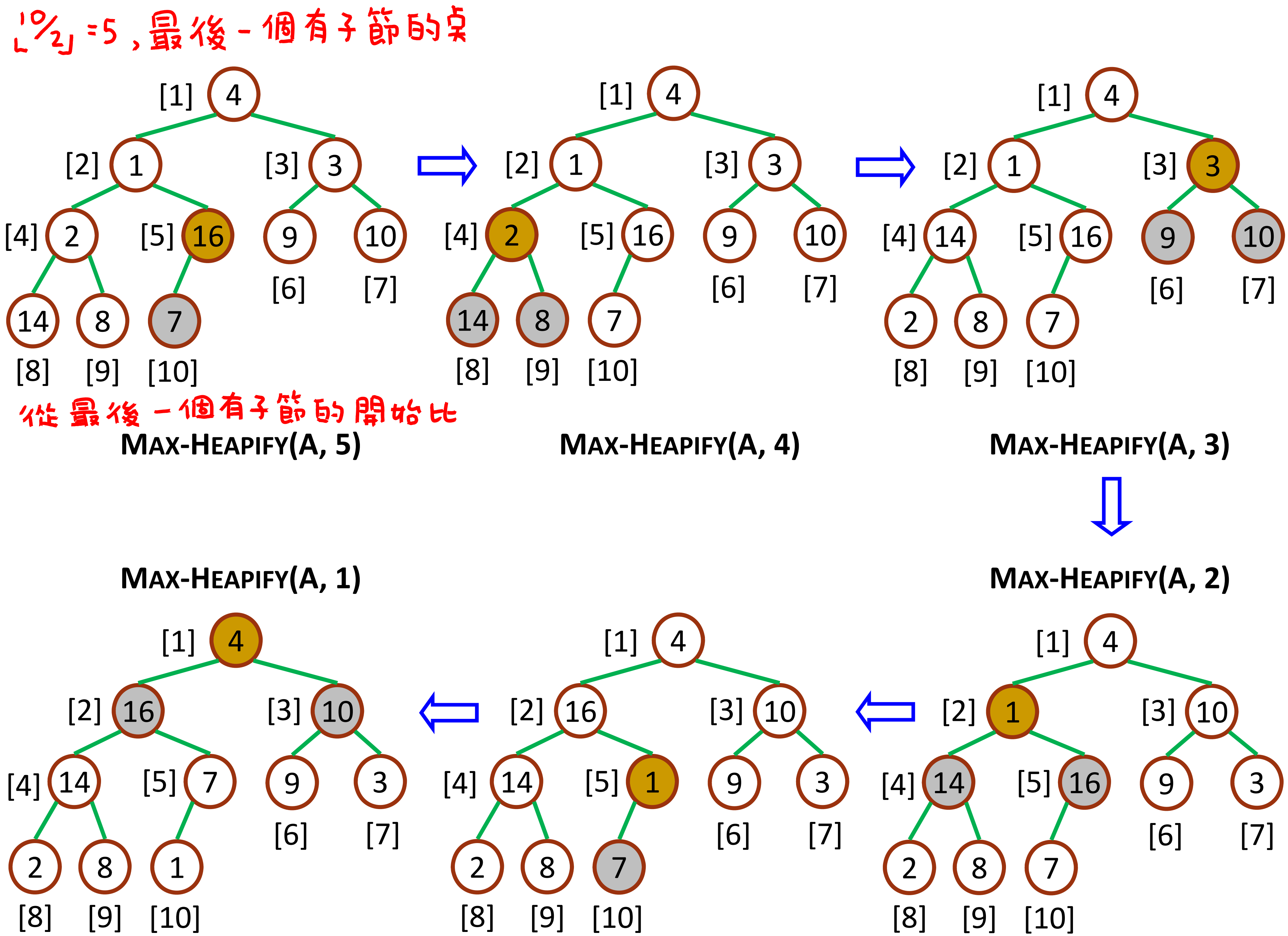
- The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

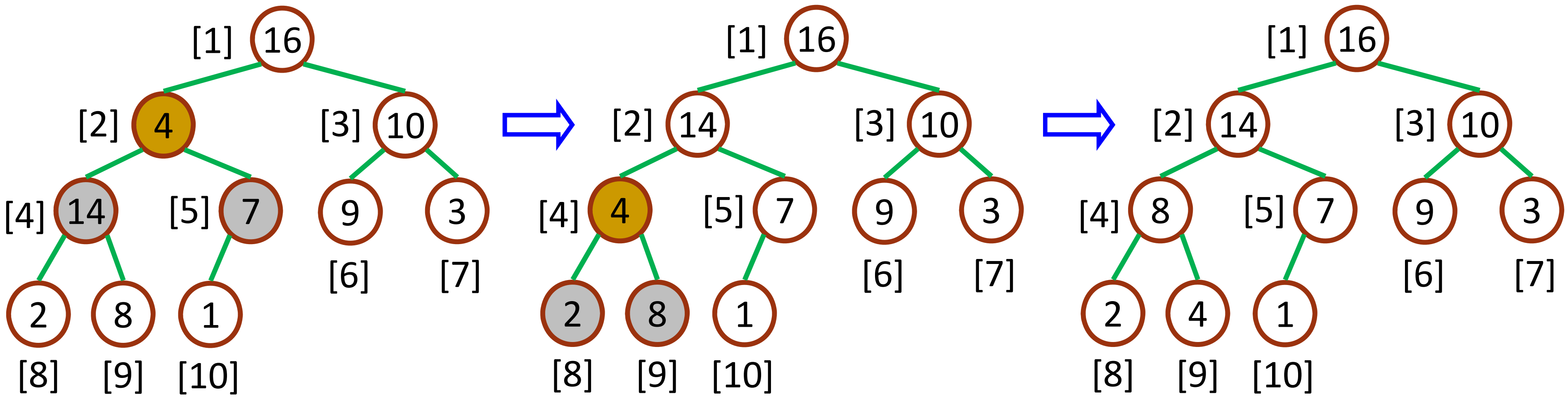
$\lfloor n/2 \rfloor$是最後一個有子節的點，從$\lfloor n/2 \rfloor$一直往前比到root

BUILD-MAX-HEAP(A)

1. *heap-size*[A] ← *length*[A]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 = root
3. **do** MAX-HEAPIFY(A,*i*)

需由後往前做MAX_HEAPIFY
如此，在做節點2時，左子樹
和右子樹都是MAX HEAP
⇒滿足MAX_HEAPIFY(A,2)的要求


⌊10/2⌋=5，最後一個有子節的点
從最後一個有子節的開始比
MAX-HEAPIFY(A, 5)
MAX-HEAPIFY(A, 4)
MAX-HEAPIFY(A, 3)
MAX-HEAPIFY(A, 2)
MAX-HEAPIFY(A, 1)


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

max-heap

Correctness$_{1/2}$

- To show why BUILD-MAX-HEAP work correctly, we use the following **loop invariant:** 先有一個敘述
  - At the start of each iteration of the for loop of lines 2-3, each node *i*+1, *i*+2, …, *n* is the root of a max-heap.

BUILD-MAX-HEAP(A)

1. *heap-size*[A] ← *length*[A]
2. **for** *i* ← ⌊*length[A]*/2⌋ **downto** 1
3.   **do** MAX-HEAPIFY(A,*i*)

做第i步時以i+1，i+2，… n 為root的subtree都是Max-heap

Ex：i=5，以6，7，8，9，10為root 的subtree都是Max-heap

- We need to show that
  - this invariant is true prior to the first loop iteration 在第一個loop前正確
  - each iteration of the loop maintains the invariant 在每個loop後仍正確
  - the invariant provides a useful property to show correctness when the loop terminates. 利用敘述來證明正確性

Correctness$_{2/2}$

- **Initialization**: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. $\lfloor n/2 \rfloor+1, \ldots n$ is a leaf and is thus the root of a trivial max-heap.

因為從第一個有子節的桌開始做，$i=\lfloor n/2 \rfloor$，所以 $\lfloor n/2 \rfloor+1$，$\lfloor n/2 \rfloor+2 \ldots n$ 都是 leaf，自然是 Max-heap

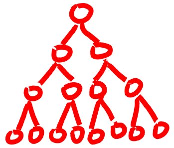
- **Maintenance**: By the loop invariant, the children of node i are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \ldots, n$ are all roots of max-heaps.

因為 max-heapify 的關係，所以 $i, i+1, \ldots, n$ 為 root 的 subtree 都是 max-heap

- **Termination**: At termination, i=0. By the loop invariant, each node 1, 2, …, n is the root of a max-heap. In particular, node 1 is.

在程式結束時 $i=0$，所以以 $1,2,3 \ldots n$ 為 root 的 subtree 都是 max-heap
因為以 1 為 root 就是整棵樹，故得證

Time complexity$_{1/2}$

Ex: 

$n=15$

$\lfloor \lg 15 \rfloor = \lfloor 3.\cdots \rfloor = 3$

h高度	第h層 node 數
0	8
1	4
2	2
3	1

$\left\lceil \frac{15}{2^{0+1}} \right\rceil = \lceil 7.5 \rceil = 8$

- **Analysis 1**: 粗略的分析
  - Each call to MAX-HEAPIFY costs $O(\lg n)$, and there are $O(n)$ such calls. 每次呼叫 max-heapify 花費 $O(\lg n)$，最多呼叫 $O(n)$ 次
  - Thus, the running time is $O(n \lg n)$. This upper bound, through correct, is **not asymptotically tight**.

花費時間 $\le c \cdot n \lg n$，但分析不緊密，不夠好

- **Analysis 2**:
  - For an n-element heap, height is $\lfloor \lg n \rfloor$ and at most $\lceil n / 2^{h+1} \rceil$ nodes of any height h. n 個 node 的高度為 $\lfloor \lg n \rfloor$，第 h 層 node 數最多為 $\lceil n/2^{h+1} \rceil$ 個 nodes
  - The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$.
  - The total cost is $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$.

將每層加起來

Time complexity$_{2/2}$

$\boxed{\text{法2}}$

$S = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots$

$-\frac{1}{2}S = \frac{1}{4} + \frac{2}{8} + \cdots$

$\frac{1}{2}S = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots$

$= \frac{\frac{1}{2}}{1-\frac{1}{2}} = 1 \Rightarrow S = 2$

- The last summation yields

$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$, for $|x|<1$

$x + 2x^2 + 3x^3 + \cdots$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

公式 $\boxed{\text{法1}}$

- Thus, the running time of BUILD-MAX-HEAP can be bounded as

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

- We can build a max-heap from an unordered array in **linear time**.

原 $O(n \lg n)$

今 $O(n)$

⇒上界推算出往下了，更加緊密

Outline

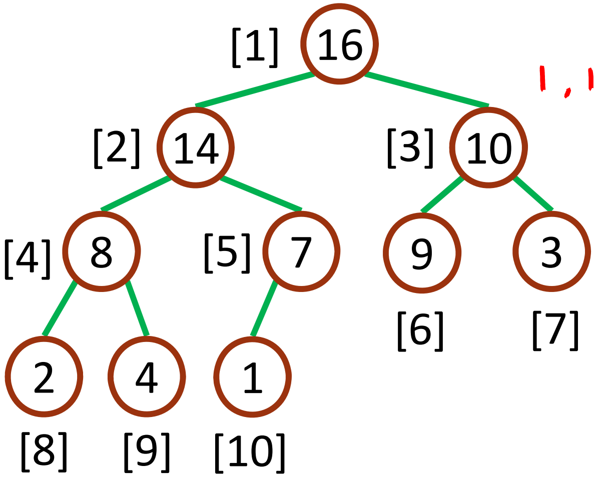
- Heaps
- Maintaining the heap property
- Building a heap
- **The heapsort algorithm**
- Priority queues

The heapsort algorithm

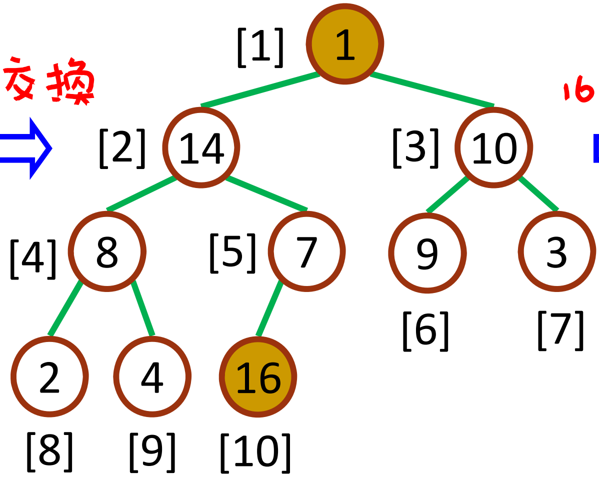
3 step : (I) 交換 exchange
(II) 忽略 discard
(III) 重新調整 readjust

- Since the maximum element of the array is stored at the root, A[1] we can **exchange** it with $A[n]$.
- If we now "**discard**" $A[n]$, we observe that $A[1...(n-1)]$ can easily be made into a max-heap.
- The children of the root $A[1]$ remain max-heaps, but the new root $A[1]$ element may violate the max-heap property, so we need to **readjust** the max-heap. That is to call MAX-HEAPIFY(A, 1).

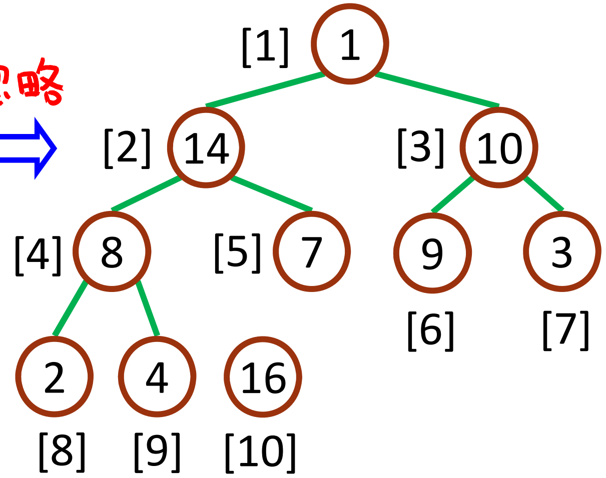
```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← length[A] downto 2        從n往前做到2
        do exchange A[1] ↔ A[i]       交換
           heap-size[A] ← heap-size[A] –1   忽略 將 heap-size - 1
           MAX-HEAPIFY(A, 1)          重新調整
```



1,16交換



16忽略



Initial heap

Exchange
Heap size = 10
Sorted=[16]

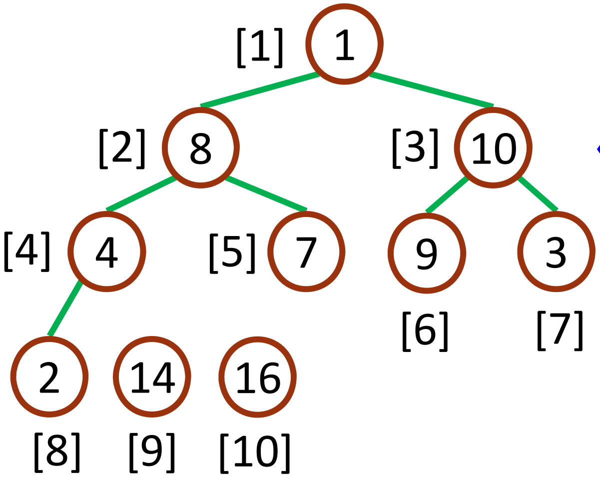
Discard
Heap size = 9
Sorted=[16]

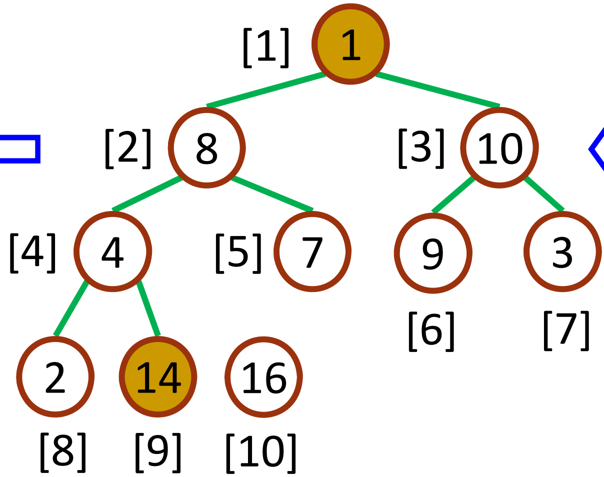
重新調整

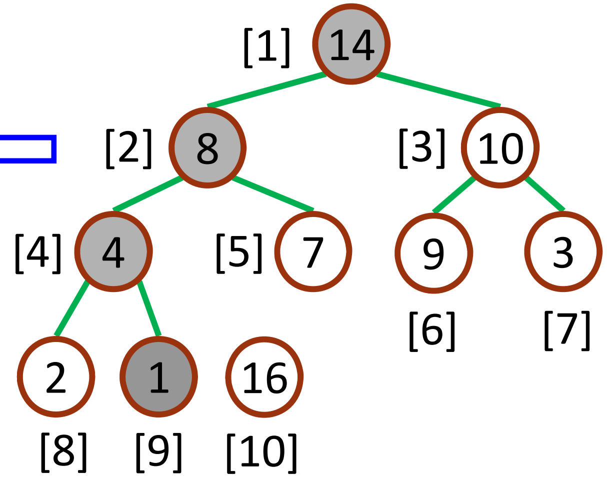
Readjust
Heap size = 9
Sorted=[16]

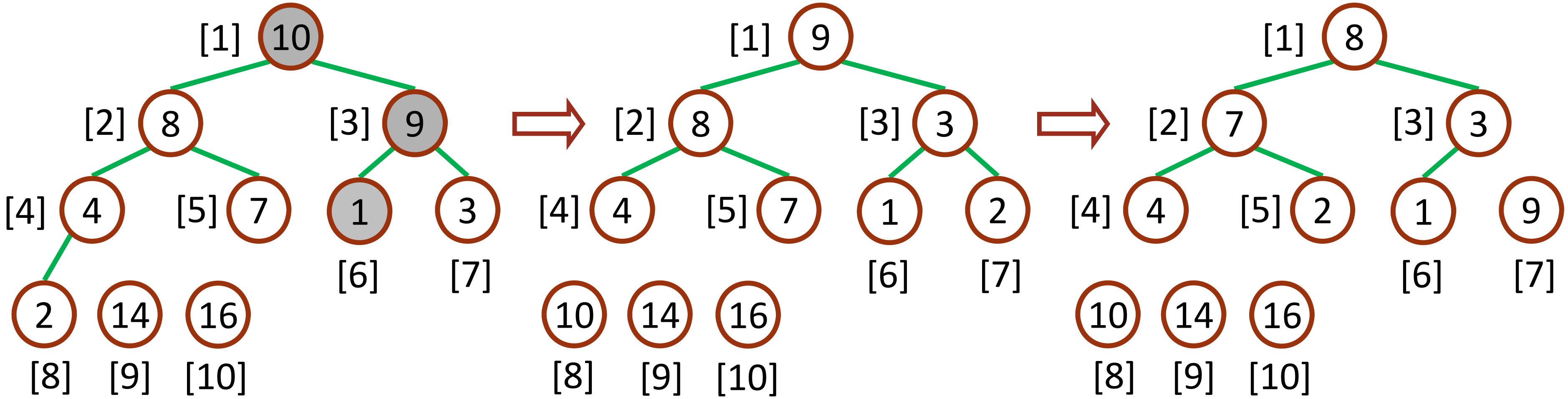
Exchange
Heap size = 9
Sorted=[14,16]

Discard
Heap size = 8
Sorted=[14,16]









Readjust
Heap size = 8
Sorted=[14,16]

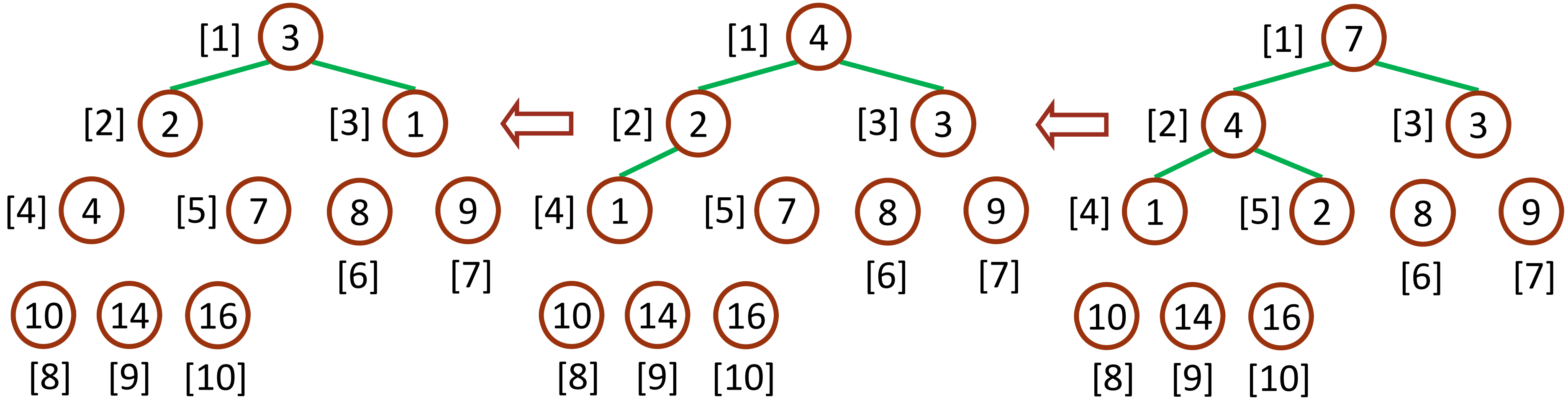
Heap size = 7
Sorted=[10,14,16]

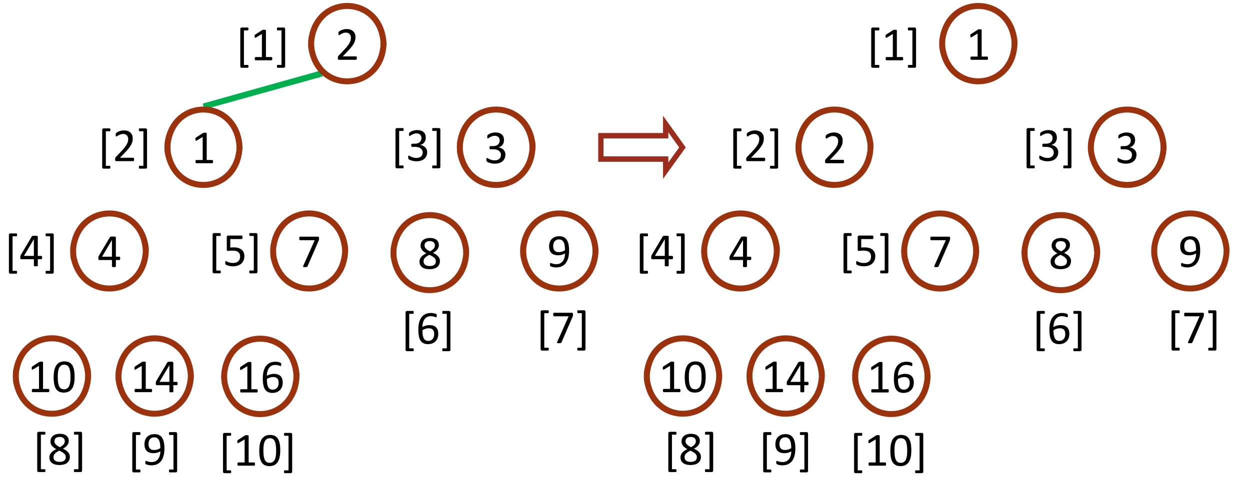
Heap size = 6
Sorted=[9,10,14,16]

Heap size = 3
Sorted=[4,7,8,9,10,14,16]

Heap size = 4
Sorted=[7,8,9,10,14,16]

Heap size = 5
Sorted=[8,9,10,14,16]





Heap size = 2
Sorted=[3,4,7,8,9,10,14,16]

Heap size = 1
Sorted=[2,3,4,7,8,9,10,14,16]

Time complexity $O(n) + (n-1) \cdot O(\lg n) = O(n \lg n)$

- The HEAPSORT procedure takes ***O*(*n* lg *n*)** time
  - the call to BUILD-MAX-HEAP takes $O(n)$ time 建立 heap 的時間
  - each of the $n-1$ calls to MAX-HEAPIFY takes $O(\lg n)$ time
  呼叫 max-heapify $n-1$ 次，每次 $O(\lg n)$

Outline

- Heaps
- Maintaining the heap property
- Building a heap
- The heapsort algorithm
- **Priority queues** 優先佇列

Heap implementation of priority queues

- Heaps efficiently implement priority queues.
- There are two kinds of priority queues: max-priority queues and min-priority queues.
- We will focus here on how to implement max-priority queues, which are in turn based on max-heaps.
- A **priority queue** is a data structure for maintaining a set *S* of elements, each with an associated value called a **key**.

priority queue 是一個資料結構，用來維護一群元素，
每一個元素都有一個鍵值 (key)

priority queue 有 2 種 Max priority queue
min priority queue

Priority queues

- A **max-priority queue** supports the following operations.
  - INSERT(S, x): inserts the element x into the set S.
    加入元素 x 到 set S 中
  - MAXIMUM(S): returns the element of S with the largest key.
    告訴我誰是最大者
  - EXTRACT-MAX(S): removes and returns the element of S with the largest key. 告訴我誰是最大者，並且將他除外
  - INCREASE-KEY(S, x, k): increases value of element x's key to the new value k. Assume $k \geq x$'s current key value.
    將 x 的 key 值增加為 k [將鍵值加大]

Finding the maximum element

- MAXIMUM(S): returns the element of S with the largest key.
- Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)

1. return A[1] 是max-heap，最大者為root

- The running time of HEAP-MAXIMUM is $\Theta(1)$.

Extracting max element

- EXTRACT-MAX(*S*): removes and returns the element of *S* with the largest key.

HEAP-EXTRACT-MAX(A)

O(1)

if *heap-size*[A] < 1
    then error "heap underflow" 檢查heap是否為空
max ← A[1] 將最大值存起來
A[1] ← A[*heap-size*[A]] 將最後一個放到root
heap-size[A] ← *heap-size*[A]−1 將heap-size減1
O(lgn) MAX-HEAPIFY(A, 1)做調整
O(1) **return** *max*

- **Analysis**: constant time assignments + time for MAX-HEAPIFY.
- The running time of HEAP-EXTRACT-MAX is $O(\lg n)$.

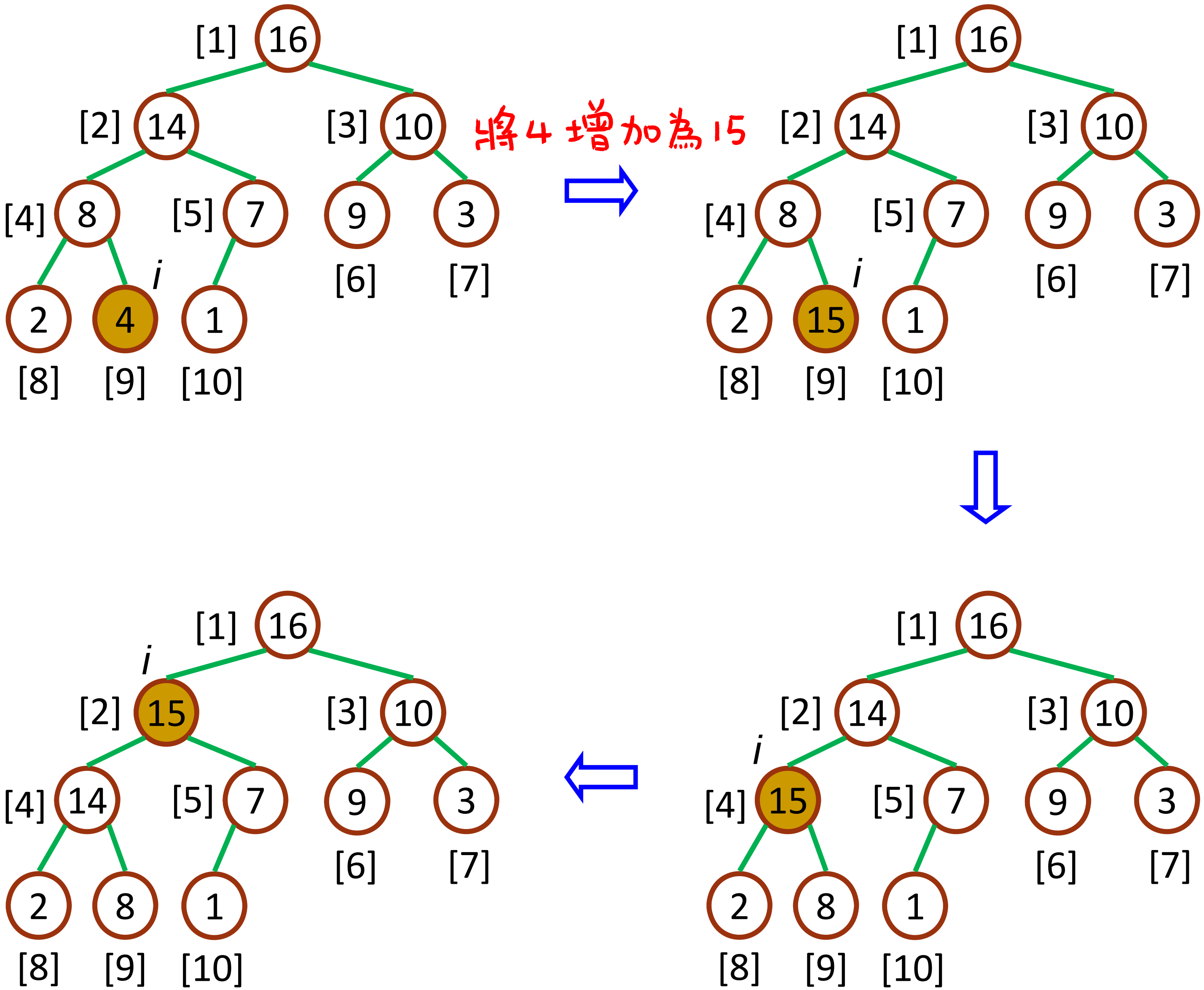
$O(1) + O(\lg n) = O(\lg n)$

Increasing key value 增加鍵值

- INCREASE-KEY(S, x, k): increases value of element x's key to k.
  Assume $k \geq x$'s current key value.

```
HEAP-INCREASE-KEY (A, i, key)
    if key < A[i]    檢查key值真的有增加
        then error "new key is smaller than current key"
    A[i] ← key
    While i > 1 and A[PARENT(i)] < A[i]    如果不是root，而且key值比父節点大，與父節点做交換
        do exchange A[i] ↔ A[PARENT(i)]
            i ← PARENT(i)
```

- **Analysis**: the path traced from the node updated to the root has length $O(\lg n)$. 最多是到root，所以最多為 $O(\lg n)$ 次 [tree的高度]
- The running time is $O(\lg n)$.


將4增加為15

Inserting into the heap

- INSERT(S, x): inserts the element x into the set S. 增加一個 element

MAX-HEAP-INSERT(A, *key*)

O(1) 1. *heap-size*[A] ← *heap-size*[A]+1 將 heap-size 加 1

2. A[*heap-size*[A]] ← −∞ 將最後一個放 −∞

O(lg n) 3. HEAP-INCREASE-KEY(A, *heap-size*[A], *key*)

將最後一個的 key 值從 −∞ 增加為 key 值

- **Analysis**: constant time assignments + time for HEAP-INCREASE-KEY.
- The running time is $O(\lg n)$.