6.Heapsort

中國文化大學
資訊工程系
副教授　張耀鴻
112學年度第2學期

1. Heap 錐型(樹)結構
2. Heap property 錐型樹特性
3. Heap construction 錐型樹之創建
4. Heapsort 錐型排序
5. Priority queue 優先佇列

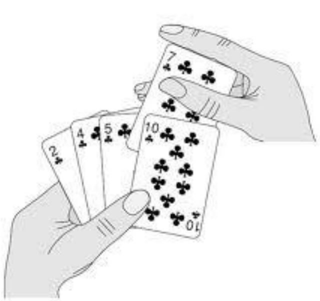
H ⇨ h_{Heap} << H

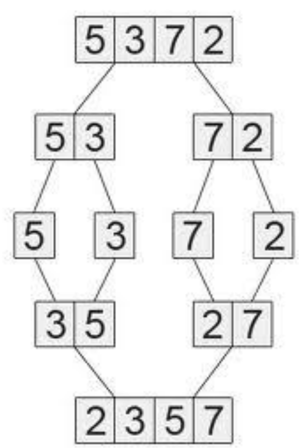
Why sorting 排序法的重要性

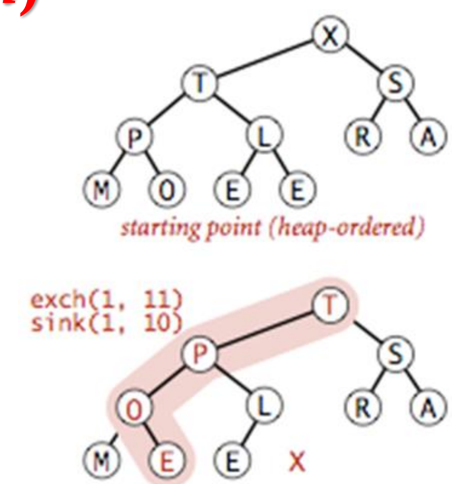
1. Sometimes the need to sort information is inherent in a application. 應用中不可或缺(固有)
2. Algorithms often use sorting as a key subroutine.
   演算法中關鍵副程式
3. There is a wide variety of sorting algorithms, and they use rich set of techniques.
   多種排序法可選擇, 應用到許多核心技術
4. Sorting problem has a nontrivial lower bound
   下界未知(理論仍有突破空間)
5. Many engineering issues come to fore when implementing sorting algorithms.
   實務應用時凸顯許多實作上的困難

Sorting algorithm

- **Insertion sort**： 無需(或只需常數)額外空間 $O(n^2)$
  - ✓ **In place**: only a constant number of elements of the input array are even sorted outside the array.
- **Merge sort**： 較快但需額外空間儲存中間值 $O(n \lg n)$
  - ✓ ***not* in place.**
- **Heap sort**：(Chapter 6) 無需額外空間且能在$O(n \lg n)$完成
  - ✓ Sorts n numbers ***in place*** in $O(n \lg n)$

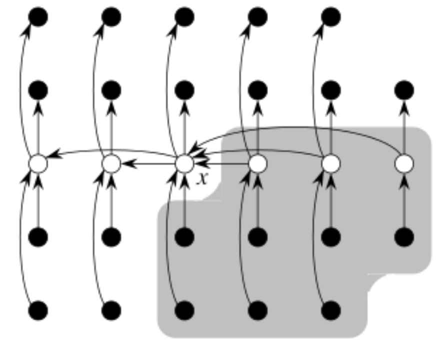






Sorting algorithm

- **Quick sort** : (chapter 7) 快速排序
  - ✓ worst time complexity $O(n^2)$
  - ✓ Average time complexity $O(n \lg n)$



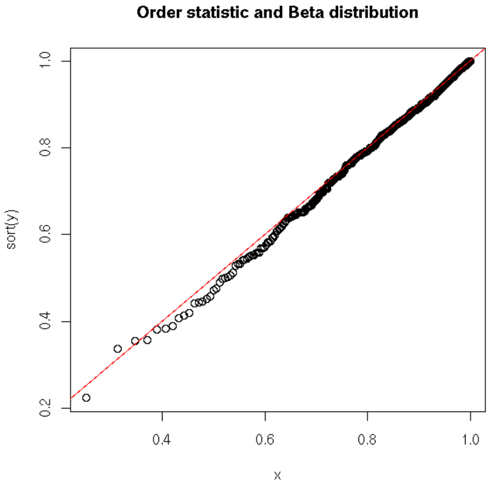
- **Decision tree model** : (chapter 8)
  - ✓ Lower bound $O(n \lg n)$ 決策樹模型
  - ✓ **Counting sort**
  - ✓ **Radix sort**

特殊狀況可打破模型限制（降至$O(n)$）

- **Order statistics** (chapter 8, 9)

有序統計

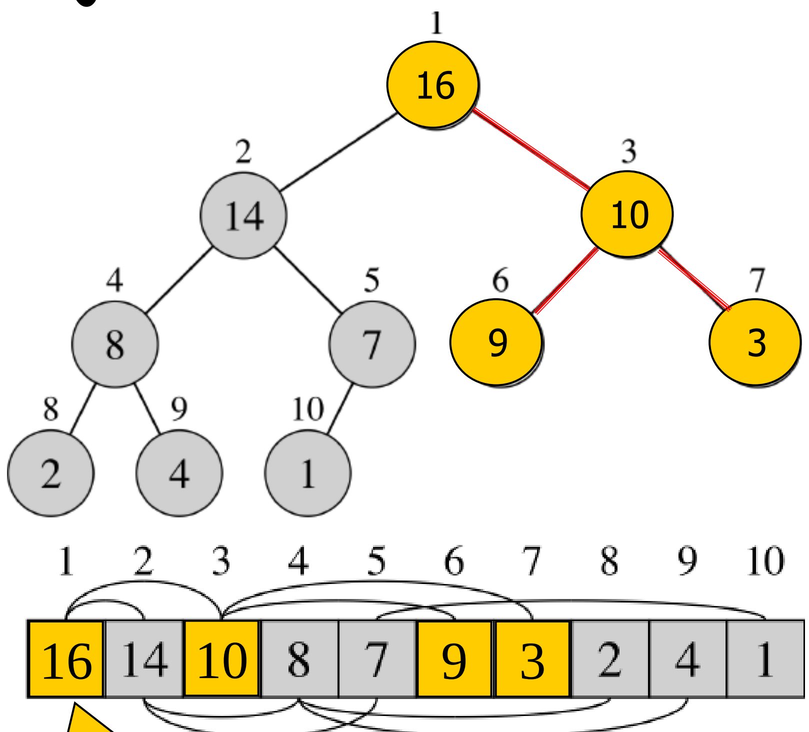
把qsort w.c. 降至$O(n \lg n)$



6.1 Heaps (Binary heap) 二元錐型結構

- The ***binary heap*** data structure is an **array** object that can be viewed as a complete tree.

幾乎是完全樹



陣列的第1個元素為樹的根

第i個節點的父親在i/2位置

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

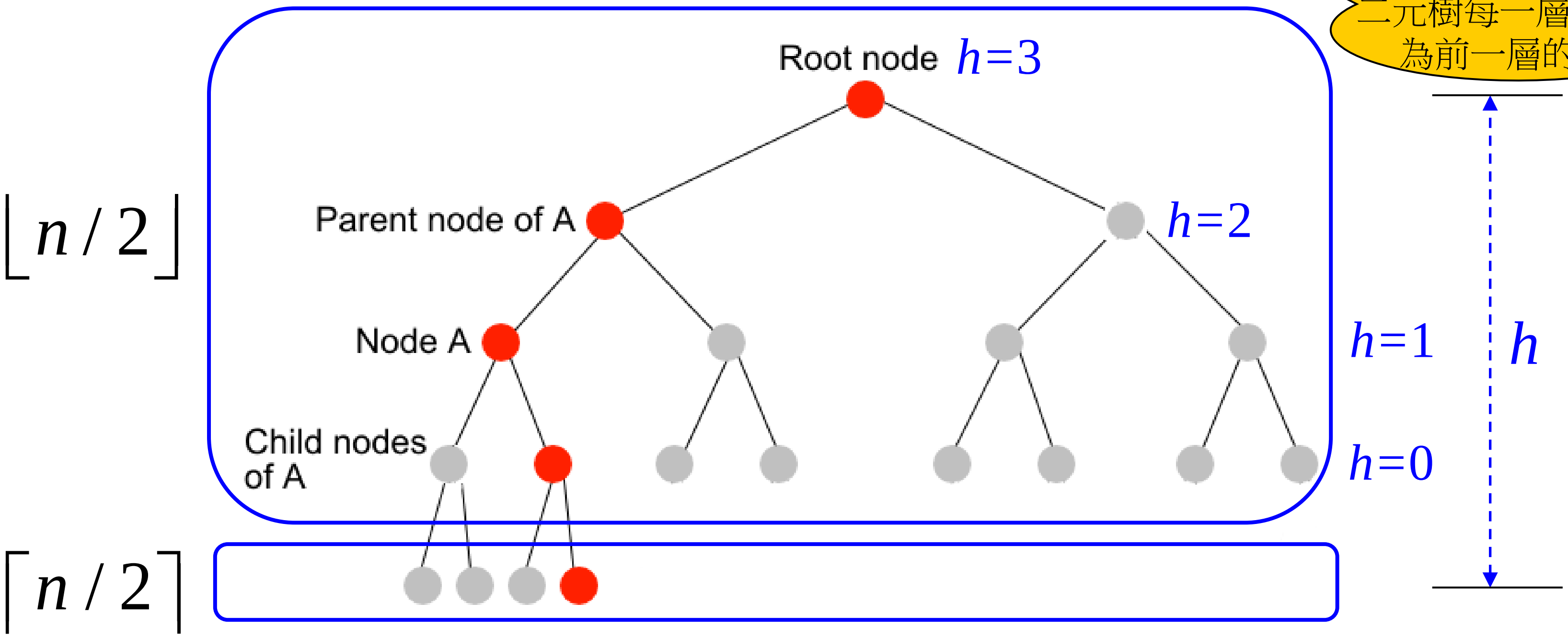
LEFT(i) i的左邊小孩在2i位置
 return $2i$

RIGHT(i) i的右邊小孩在2i+1
 return $2i+1$

完全二元樹的特性

1. 任何不在從最後一個leaf到root的唯一簡單路徑之節點(node)，為完全二元樹之根。
2. 一個高度為 h 之完全二元樹有2^h 個leaves.
3. 若一顆樹有n個節點，則最多有$\lceil n/2 \rceil$ 個leaves

二元樹每一層節點數為前一層的2倍

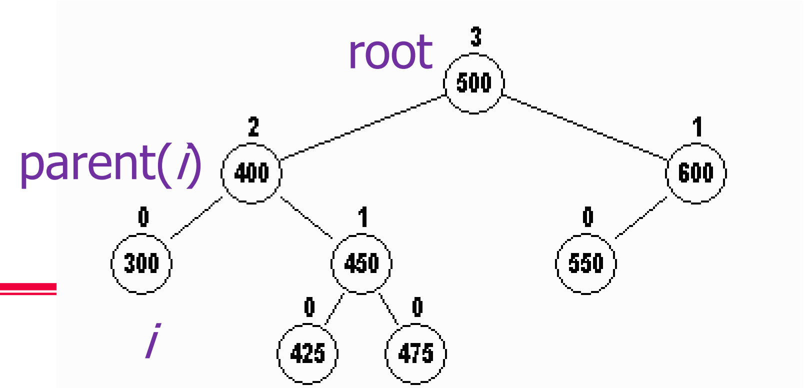


Exercise 0

寫出完全二元樹的特性。

Heap特性

Heap property



- **Max-heap** : $A[\text{parent}(i)] \geq A[i]$ —— 爸爸永遠比兒子大
- **Min-heap** : $A[\text{parent}(i)] \leq A[i]$
- The ***height of a node*** in a tree: the number of edges on the longest simple downward path from the node to a leaf.
- The ***height of a tree***: the height of the root
- ***The height of a heap***: **$O(\lg n)$**.

節點高度 = #從該節點至最遠的leaf的邊

樹的高度 = root的高度

Basic procedures on heap Heap基本運算

- Max-Heapify ➔ $O(\lg n)$ 錐型化
- Build-Max-Heap ➔ $O(n)$ 建立錐型
- Heapsort ➔ $O(n \lg n)$ 錐型排序
- Max-Heap-Insert
- Heap-Extract-Max
- Heap-Increase-Key

插入，刪除最大，鍵值升級 $O(\lg n)$

- Heap-Maximum ➔ $O(1)$

找最大

Exercise 1~7

寫出以下演算法的執行時間複雜度:

1. Max-Heapify
2. Build-Max-Heap
3. Heapsort
4. Max-Heap-Insert
5. Heap-Extract-Max
6. Heap-Increase-Key
7. Heap-Maximum

Exercise 8

Is the array with values <23, 17, 14, 6, 13, 10, 1, 5, 7, 12> a max-heap?

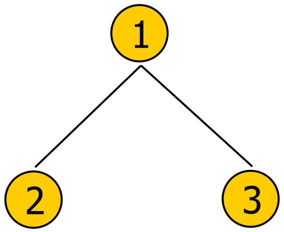
PAUL TAYLOR DANCE COMPANY
PROMETHEAN FIRE
photo: Lois Greenfield

維持Heap結構

6.2 Maintaining the heap property

- Heapify is an important subroutine for manipulating heaps.
  鎚型化為heap之重要操弄動作
- Its inputs are an array *A* and an index *i* in the array.
  輸入一陣列A, 及索引值 i
- When Heapify is called, it is assume that 何時需鎚型化?
  - ✓ the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are heaps,
  - ✓ but that *A*[*i*] may be smaller than its children,
  - ✓ thus violating the heap property.

節點 i 的左右子樹皆為 heap,
但 i 的鍵值比兒子小



Max-Heapify (*A*, *i*)

```
1  l ← Left (i)
2  r ← Right(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       then largest ← l
5       else largest  ← i
6  if r ≤ A.heap-size 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)
```

(Line 3-7)爸爸跟兩個兒子比大小，誰比較大誰就是爸爸。

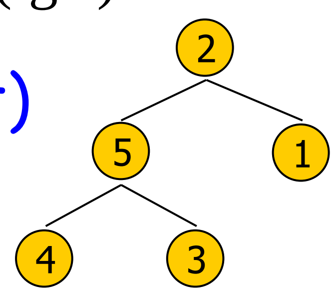
如果節點i不是最大

i往下掉

遞迴呼叫**Heapify**

$$*T(n) \le T(\frac{2n}{3}) + \Theta(1) \Rightarrow T(n) = O(\lg n)$$

Alternatively O(h)(h: height)



W.C: 最底層恰好是半滿時

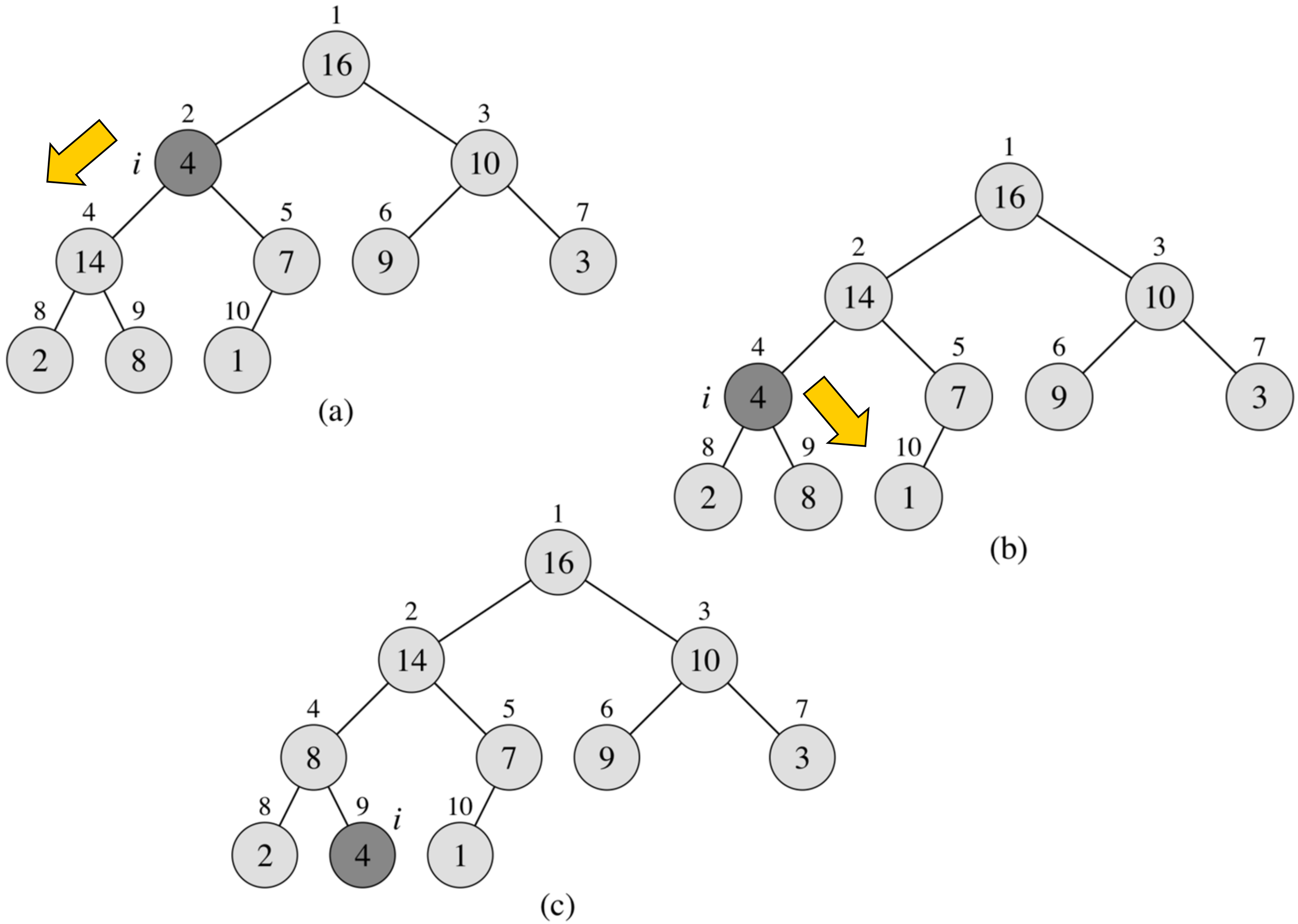
Exercise 9

用自己的話寫出Max_Heapify執行步驟。

Exercise 10

寫出Max_Heapify演算法的遞迴式，並求算將T(n)以Big-O表示的結果。

Max-Heapify(A,2)
A.heap-size = 10



Exercise 11

- 試畫出以Max-Heapify(A, 3)應用於以下陣列的過程。

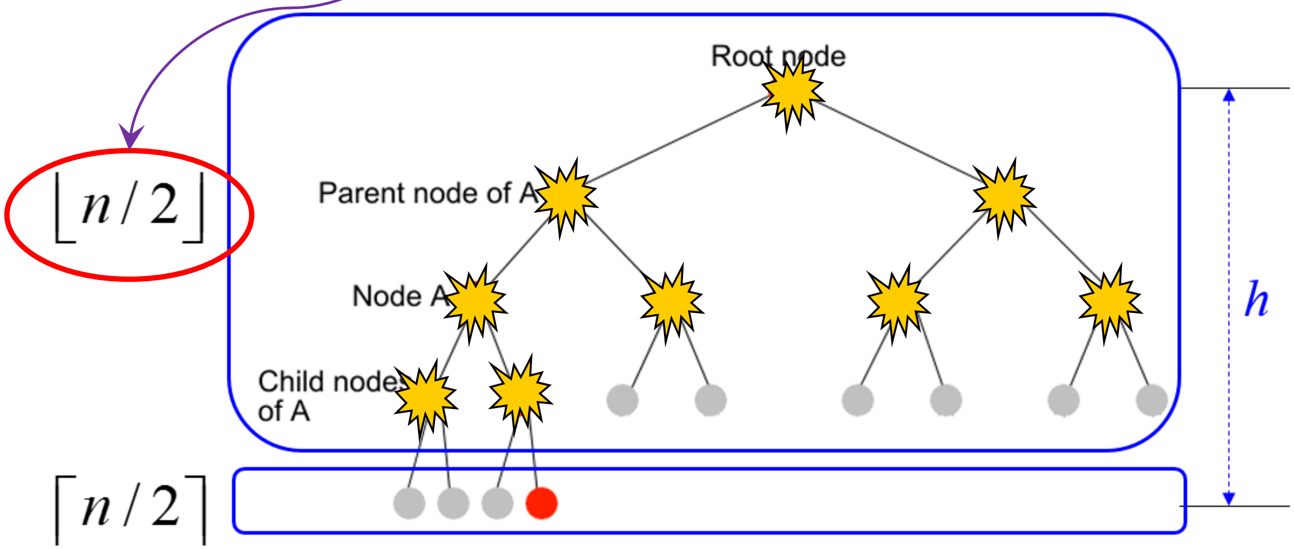
A = <27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0>

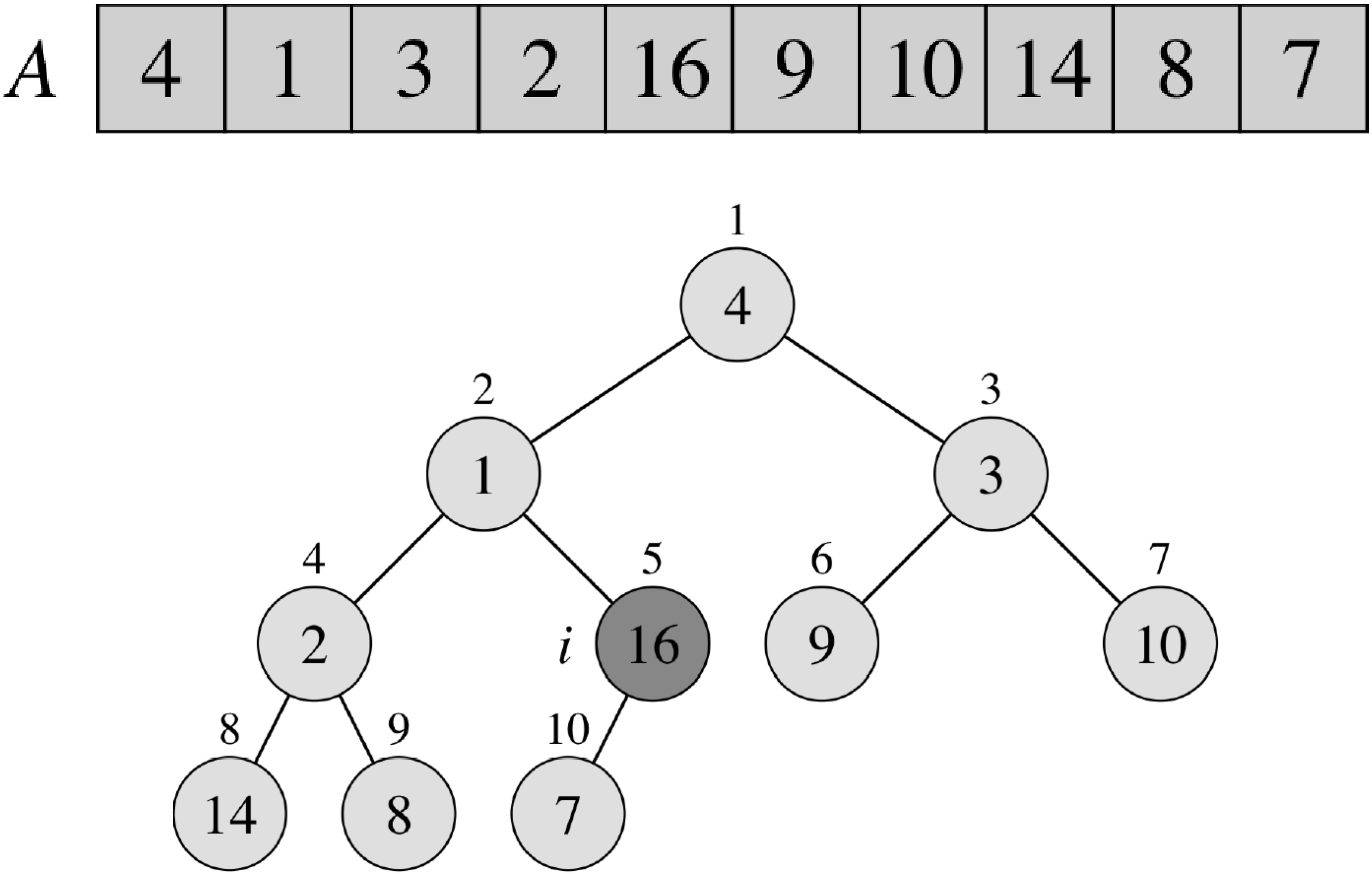
```
MAX-HEAPIFY(A, i, n)
l = LEFT(i)
r = RIGHT(i)
if l ≤ n and A[l] > A[i]
    largest = l
else largest = i
if r ≤ n and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest, n)
```

6.3 Building a heap

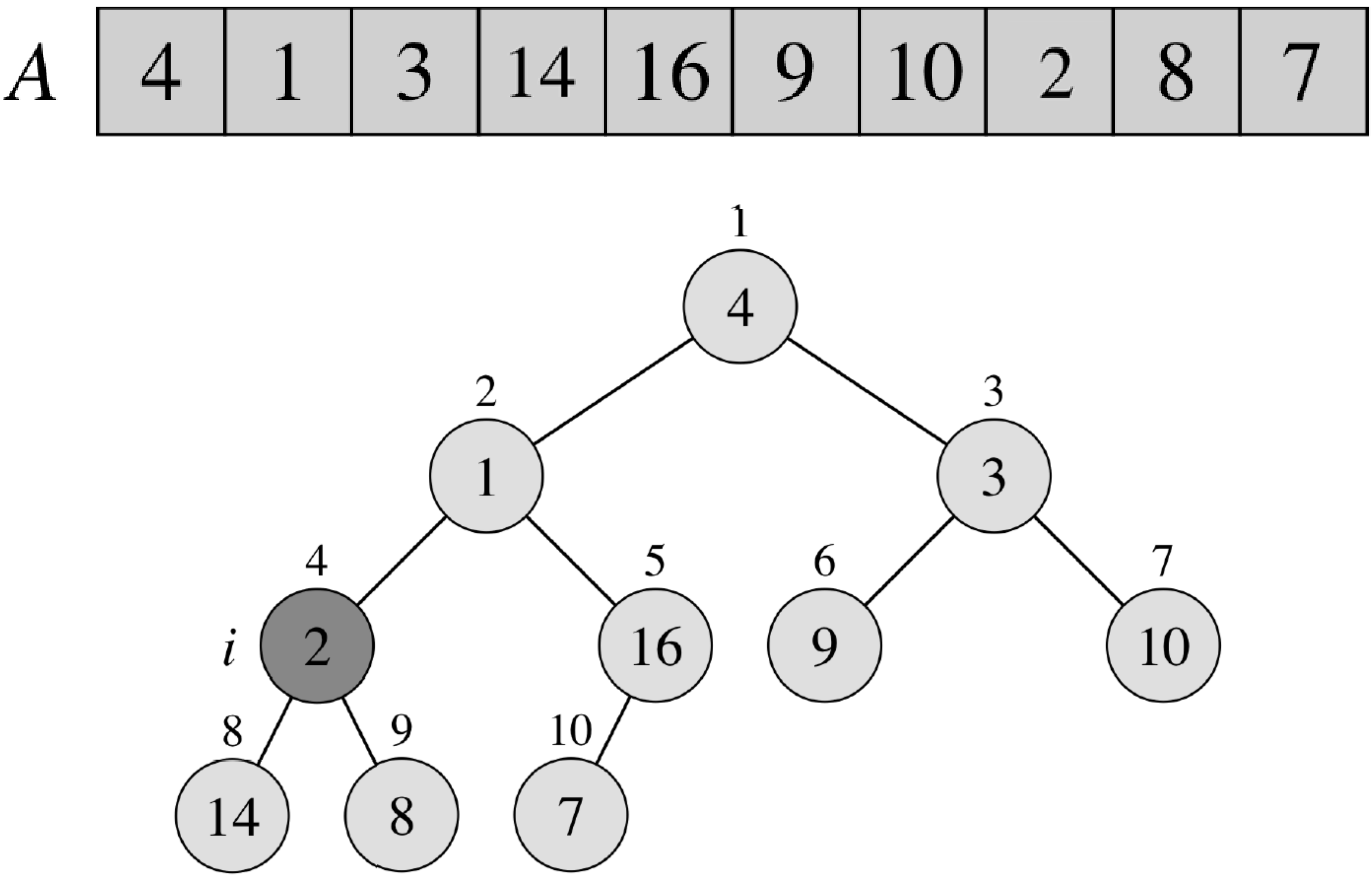
從最後一個 ***Non-Leaf*** 開始做 Heapify。

```
Build-Max-Heap(A)
A.heap_size ← A.length
  for i ← ⌊A.length/2⌋ downto 1
      do Max-Heapify(A, i)
```



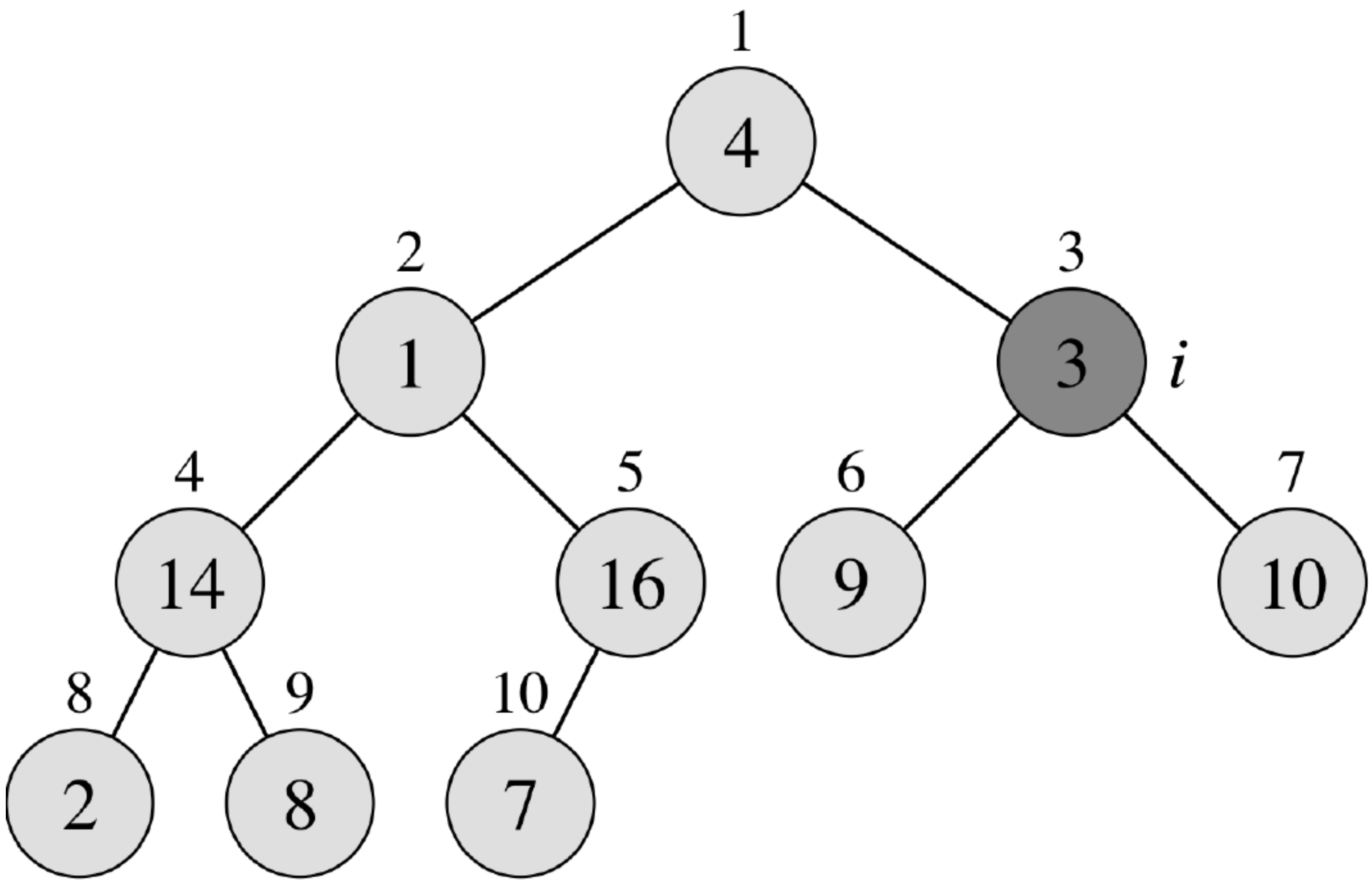


(a)



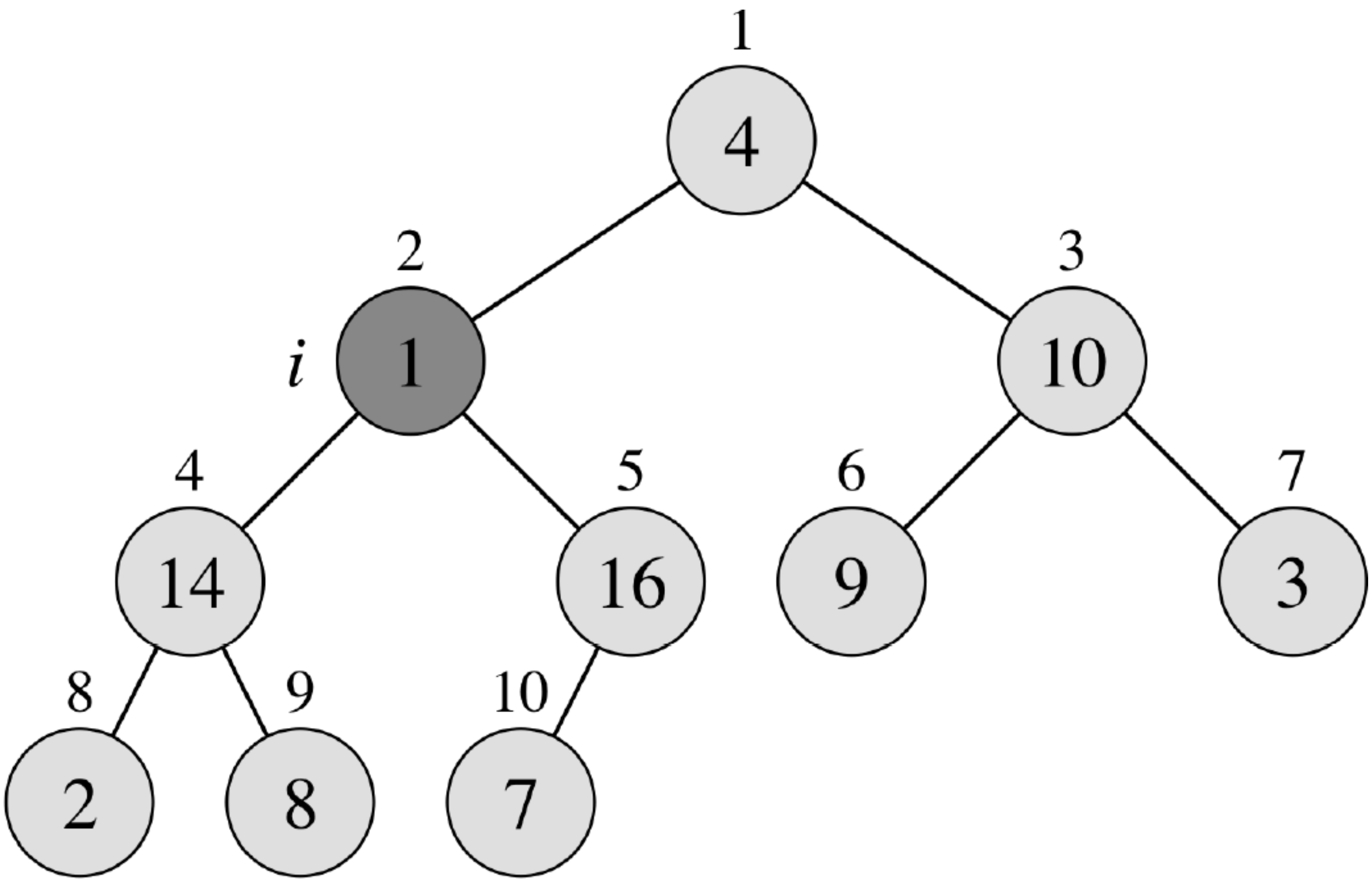
(b)

A									
4	1	10	14	16	9	3	2	8	7



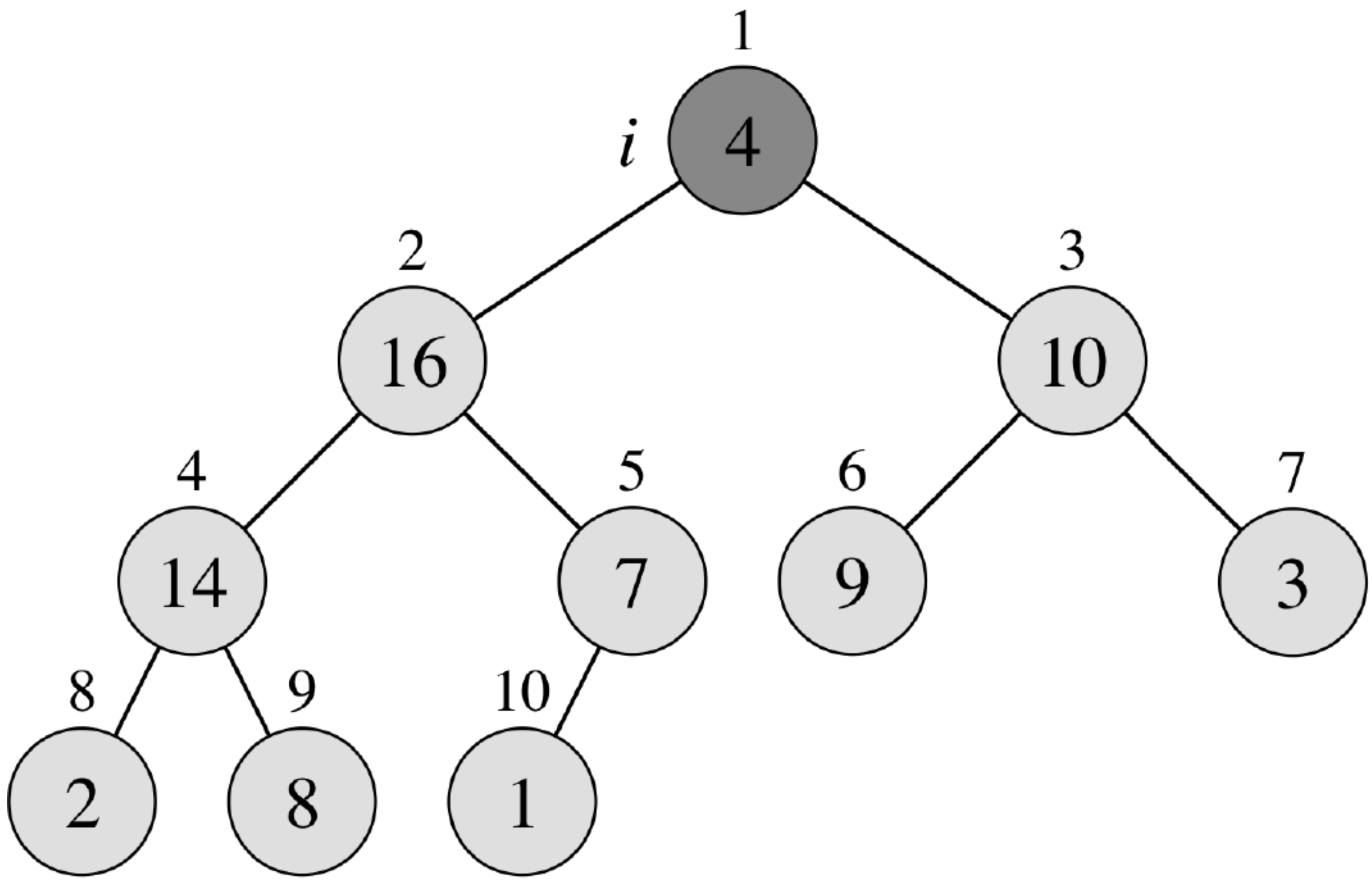
(c)

A	4	16	10	14	7	9	3	2	8	1



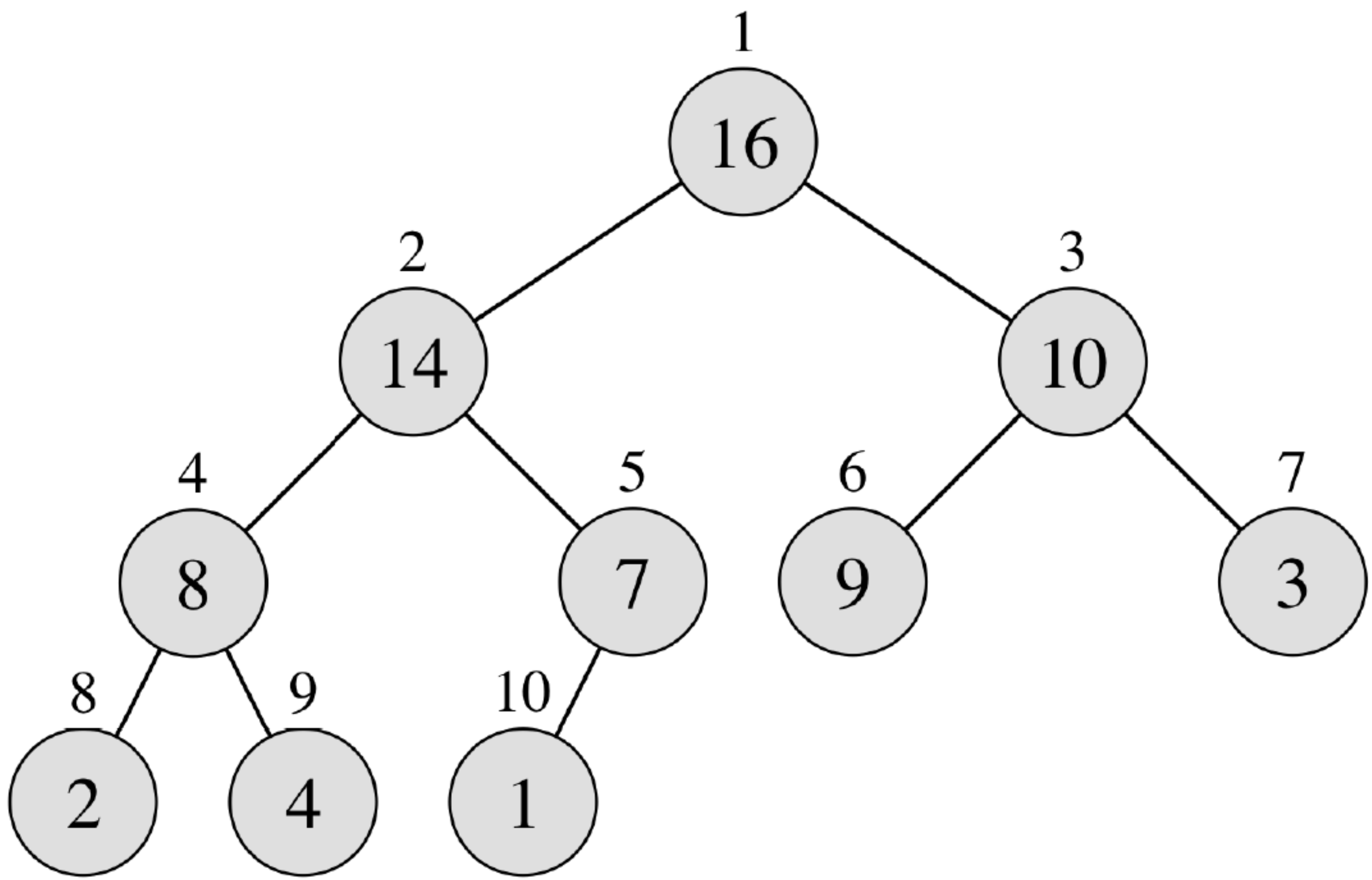
(d)

A	4	16	10	14	7	9	3	2	8	1



(e)

A	16	14	10	8	7	9	3	2	4	1

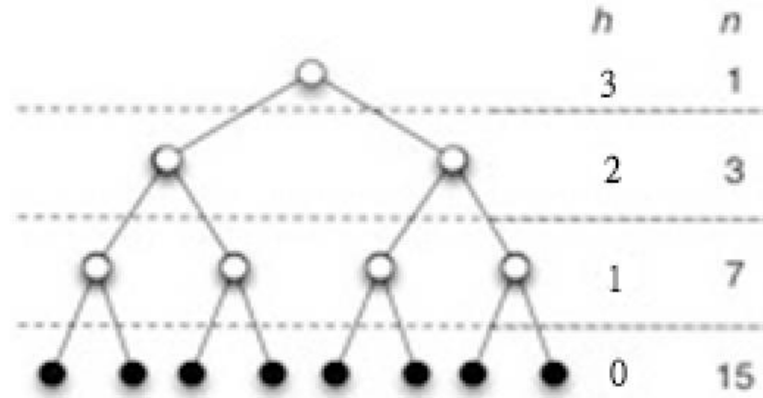


(f)

Tight Bound 注意太棒!

- By observation 觀察法

$$O(n\lg n)?$$



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

At most $\left\lceil n/2^{h+1} \right\rceil$ nodes at height h

高度為h的節點最多有幾個

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \qquad (\because \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2})$$

(A.8)

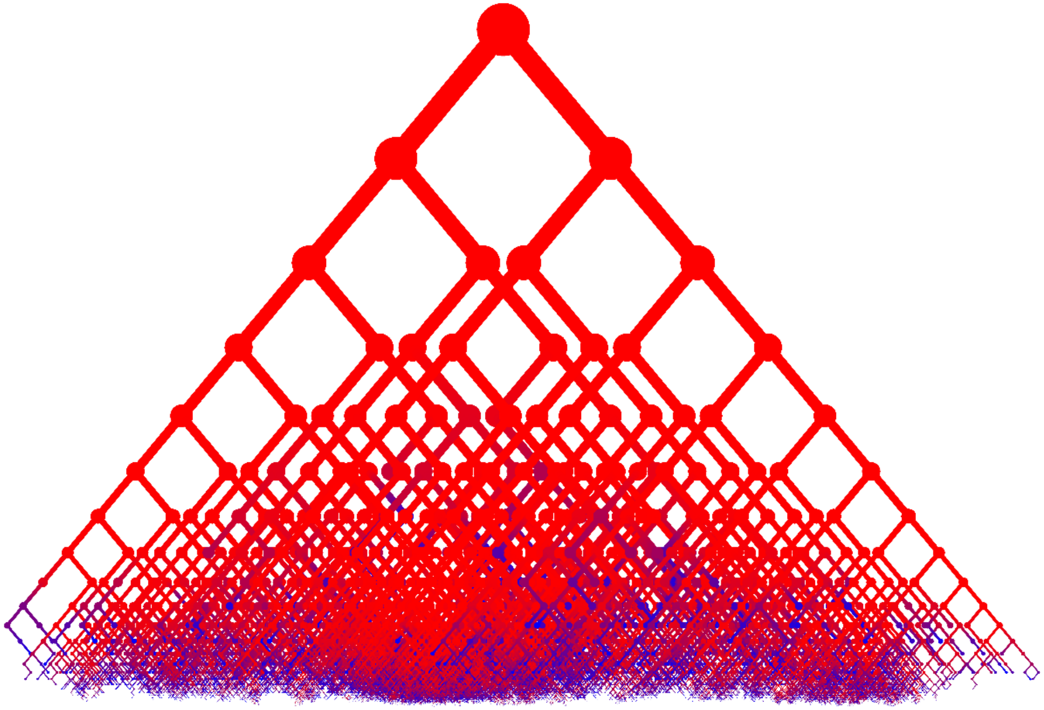
$$O(n\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n\sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

BuildHeap
只需線性時間

Exercise 12

- 試畫出以Build-Max-Heap 演算法為以下陣列建立Heap的過程。

  A = <5, 3, 17, 10, 84, 19, 6, 22, 9>

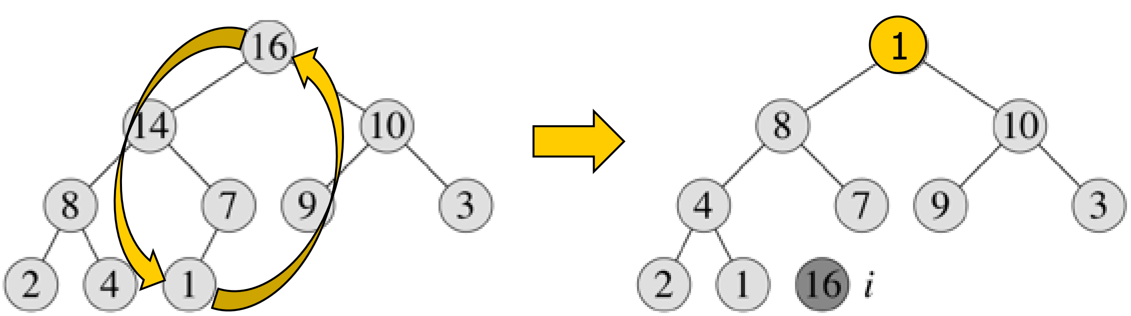


6.4 The Heapsort algorithm

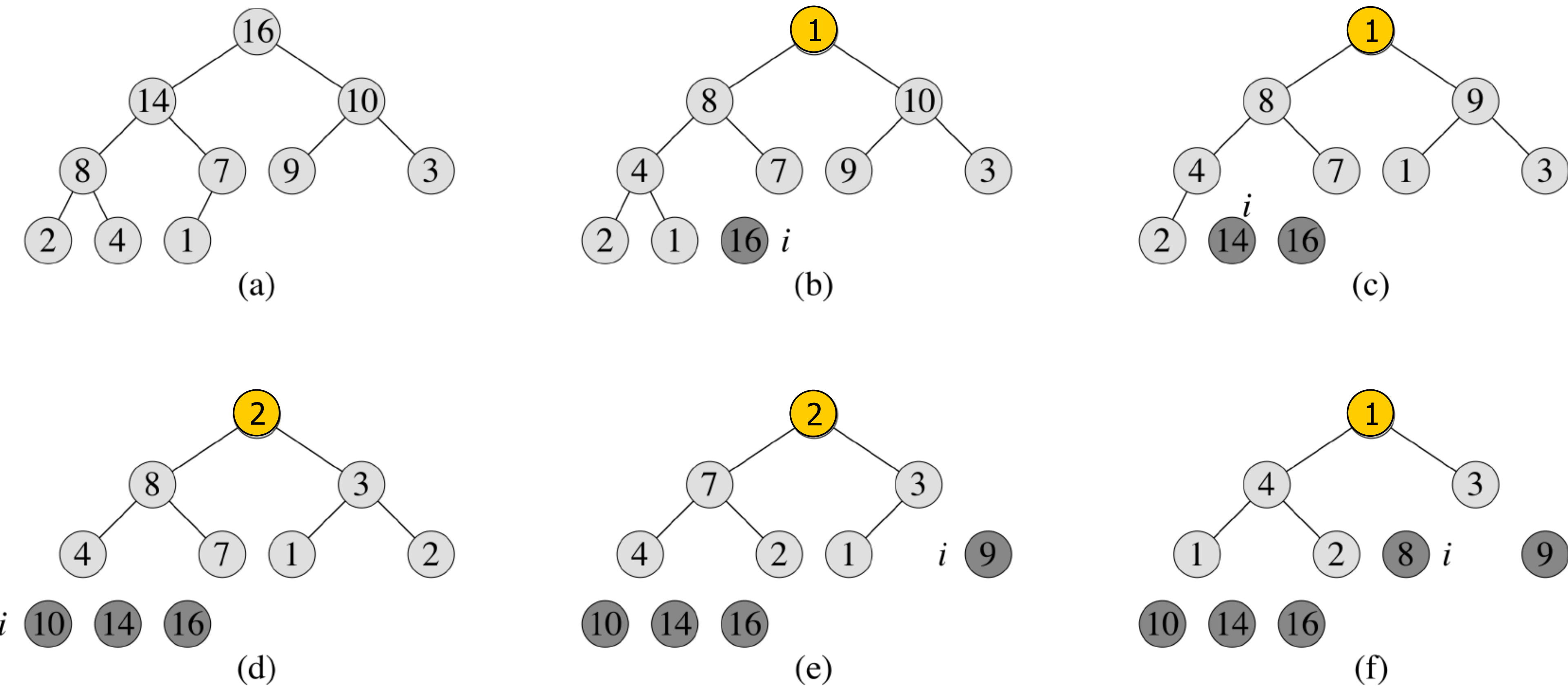
```
Heapsort(A)
Build-Max-Heap(A)
for i ← A.length down to 2
    do exchange A[1]←→A[i]
        A.heap-size ← A.heap-size -1
        Max-Heapify(A, 1)
```

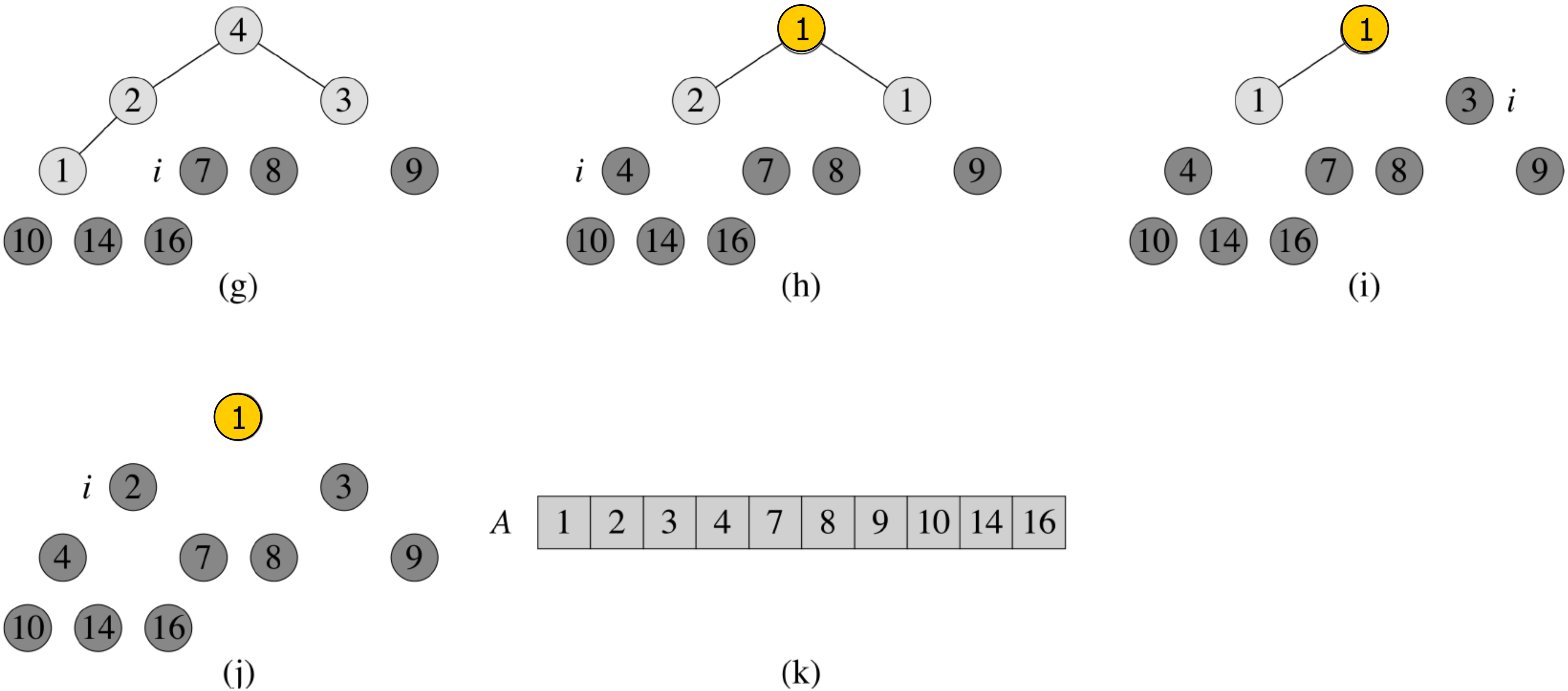
先Build-Heap➔O(n)

再*從最後一個* ***Node*** *開始*，
把每個 Node 跟 Root 交換，
再對 Root 做 Heapify
➔O(lg n)做 n 次。



The operation of Heapsort





Analysis: $\mathbf{O(\mathit{n}\ lg\ \mathit{n})}$

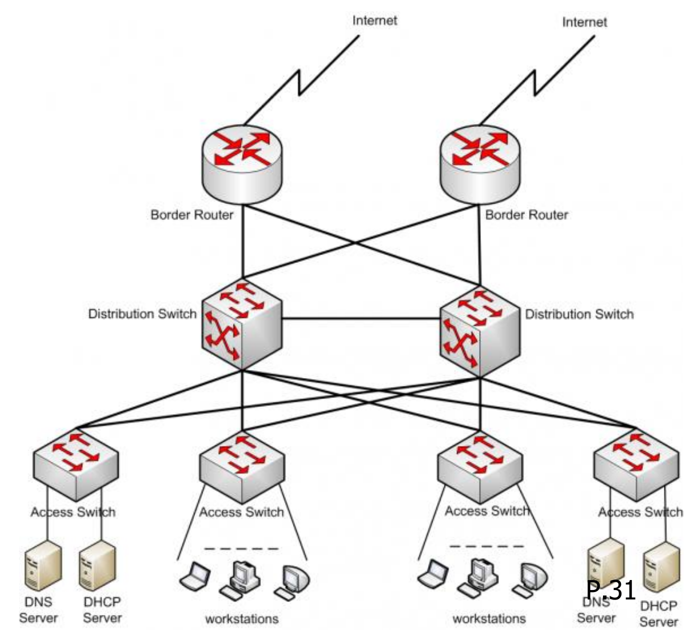
Analysis of HeapSort

1. Build-Max-Heap: **O(*n*)**
2. for loop: n-1 times **O(*n*)**
3. exchange elements: **O(1)**
4. Max-Heapify: **O(lg *n*)**

O(lg *n*)

O(*n*)

➢ Total time: **O(*n* lg *n*)**



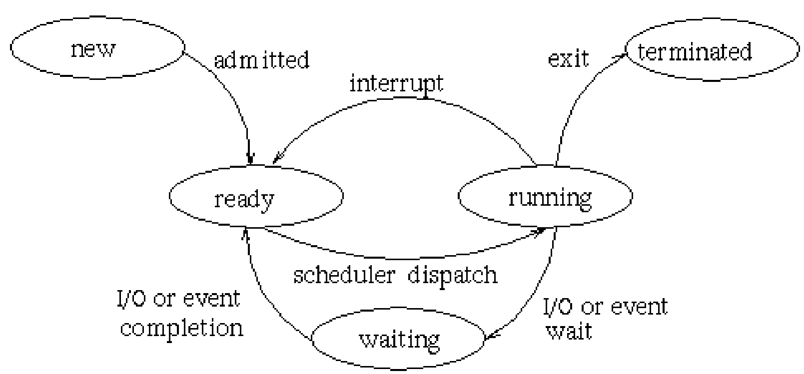
Exercise 13

- 寫出Heapsort演算法並逐行分析其執行時間複雜度。

6.5 Priority queues 優先佇列(Heap之應用)

A **priority queue** is a data structure that maintain a **set** S of elements, each with an associated value call a **key**. A **max-priority queue** support the following operations: 優先佇列: PQ

- INSERT (S, x) $O(\lg n)$ 插入
- MAXIMUM (S) $O(1)$ 找最大
- EXTRACT-MAX (S) $O(\lg n)$ 刪最大
- INCREASE-KEY (S, x, k) $O(\lg n)$ 鍵(優先權)升級



Exercise 14

- 寫出priority queue的4個基本運算及其執行時間複雜度。

HEAP-EXTRACT-MAX(*A*)

```
if heap_size[A] < 1
   then 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
```

(1) 先確認 heap 裡有東西.

(2) 備份(複製) root.

(3) 把最後一個節點搬到 root.

(4) Heapify, with one fewer node.

(5) Return the max element.

O(lg n)

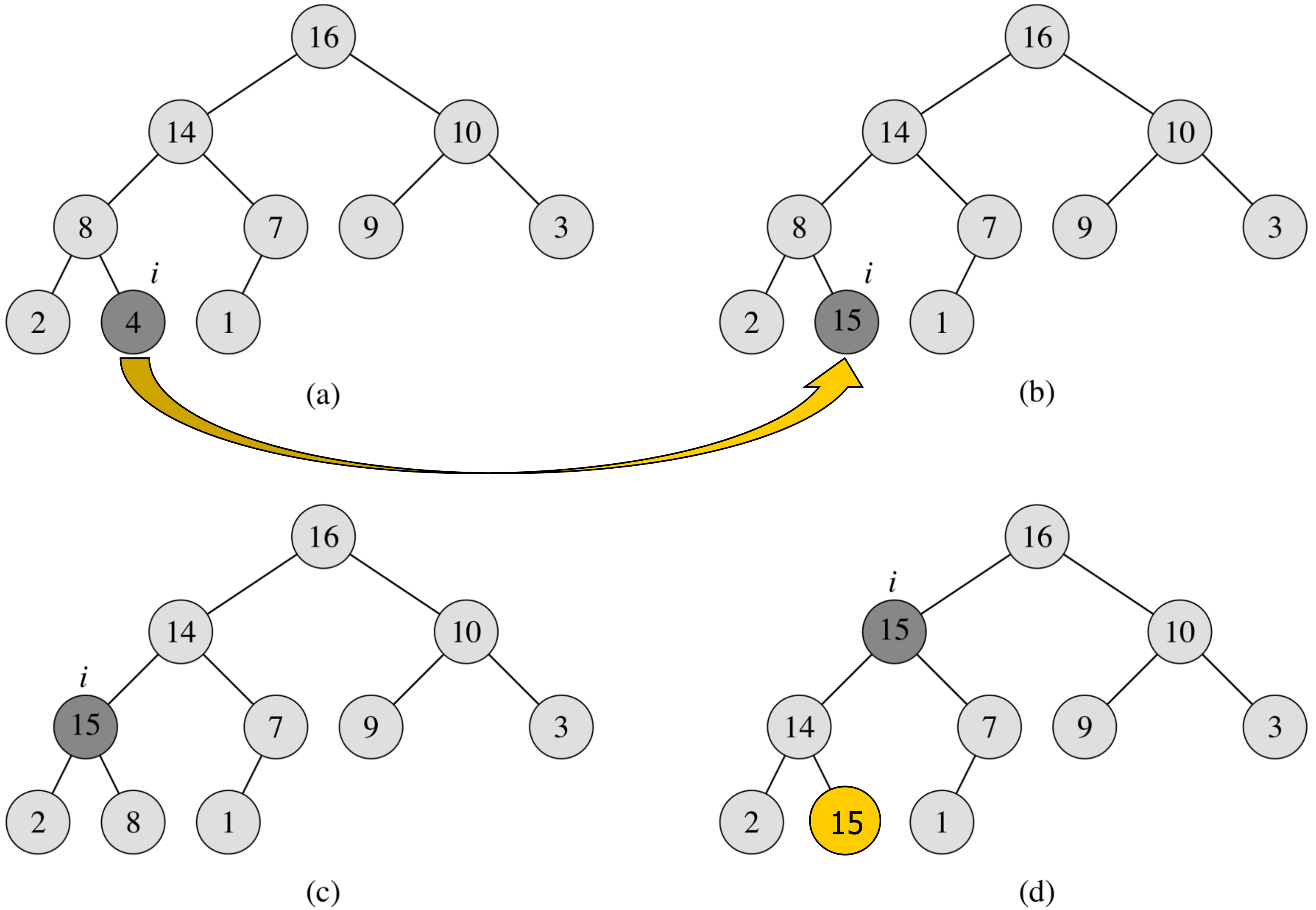
HEAP-INCREASE-KEY (*A*, *i*, *key*)

if *key* < *A*[*i*]
  then error “new key is smaller than current key”
A[*i*] ← *key*
while *i* > 1 and *A*[PARENT(*i*)] < *A* [*i*]
  do exchange *A* [*i*] ←→ *A* [PARENT(*i*)]
    i ← PARENT (*i*)

如果 Node i 的鍵值比爸爸大就往上跑
最多可能從最底層到root

Total time: **O(lg *n*)**

Heap-Increase-Key



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

1 heap_size[A] ← heap_size[A] + 1

2 A [heap_size[A]] ← -∞

3 HEAP-INCREASE-KEY (A, heap_size[A], *key*)

先插到最後一個 Node
再做 Increase-key

補充：C++ std heap_sort implementation

```cpp
void heap_sort(vector<int> & A)
{
    unsigned heapSize = A.size() - 1;
    build_heap(A, heapSize);
    for (unsigned i=heapSize; i > 0; --i) {
        swap(A[0], A[i]);
        heapSize -= 1;
        heapify(A, 0, heapSize);
    } // end of if
} // end of heap_sort
```

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

```cpp
void std_heap_sort(vector<int> & A)
{
    make_heap(A.begin(), A.end() );
    sort_heap(A.begin(), A.end() );
} // end of std_heap_sort
```

補充：C++ std priority_queue implementation

```cpp
void std_priority_queue(vector<int> & A)
{
    priority_queue<int, vector<int>, greater<int> > q;
    for (const auto &i : A)
        q.push(i);
    for (vector<int>::iterator it=A.begin(); it<A.end(); ++it, q.pop() )
        *it = q.top();
} // end of std_priority_queue
```

```
Before:
1222 6187 4111 3304 3875 1767 1360 729 587 131
After:
131 587 729 1222 1360 1767 3304 3875 4111 6187
10,9.964e-006
```

補充：Summary of std::priority_queue Operations

Operation	Notes
priority_queue <T, [ctr_type<T>], [cmp_type]>([cmp], [ctr])	Constructs a priority_queue of Ts using ctr as its internal container and srt as its comparator object. If no container is provided, constructs an empty deque. Uses std::less as default sorter.
pq.empty()	Returns true if container is empty.
pq.size()	Returns number of elements in container.
pq.top()	Returns a reference to the greatest element in the container.
pq.push(t)	Puts a copy of t onto the end of the container.
pq.emplace(...)	Constructs a T in place by forwarding ... to the appropriate constructor.
pq.pop()	Removes the element at the end of the container.
pq1.swap(pq2) swap(pq1, pq2)	Exchanges the contents of s2 with s1.

Exercise (Lab Homework)

- 用C++實作以下4個演算法來排序10,000到1,000,000個隨機亂數，並將其執行時間與**O(*n* lg *n*)**比較:
  1. C++ STL內建heap_sort
  2. CLRS的Heapsort
  3. CLRS的Merge_Sort
  4. C++ STL內建的priority_queue

Summary

- Heap
- Heap property
- Height of node, height of heap
- Basic procedures on heap
  - MAX-HEAPIFY, BUILD-MAX-HEAP, HEAPSORT
  - MAX-HEAP-INSERT, HEAP-EXTRACT-MAX
  - HEAP-INCREASE-KEY, HEAP-MAXIMUM
- Priority queue
  - INSERT, MAXIMUM, EXTRACT-MAX, INCREASE-KEY