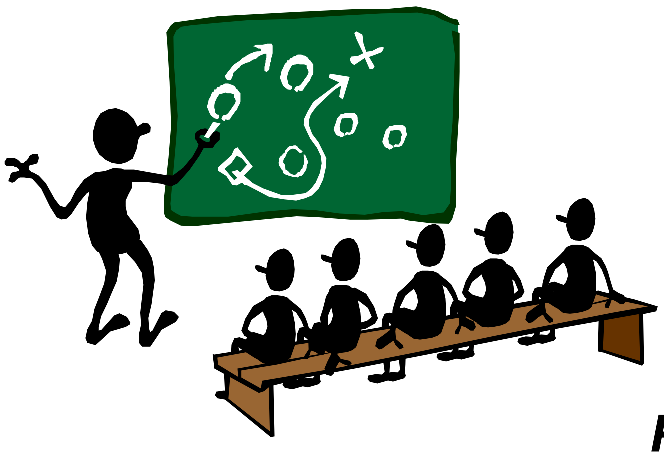
Algorithms – Chapter 6
Heap Sort



Juinn-Dar Huang

Professor

jdhuang@mail.nctu.edu.tw

August 2007

Rev. '08, '11, '12, '15, '16, '18, '19, '20, '21

Sorting Algorithms (1/2)

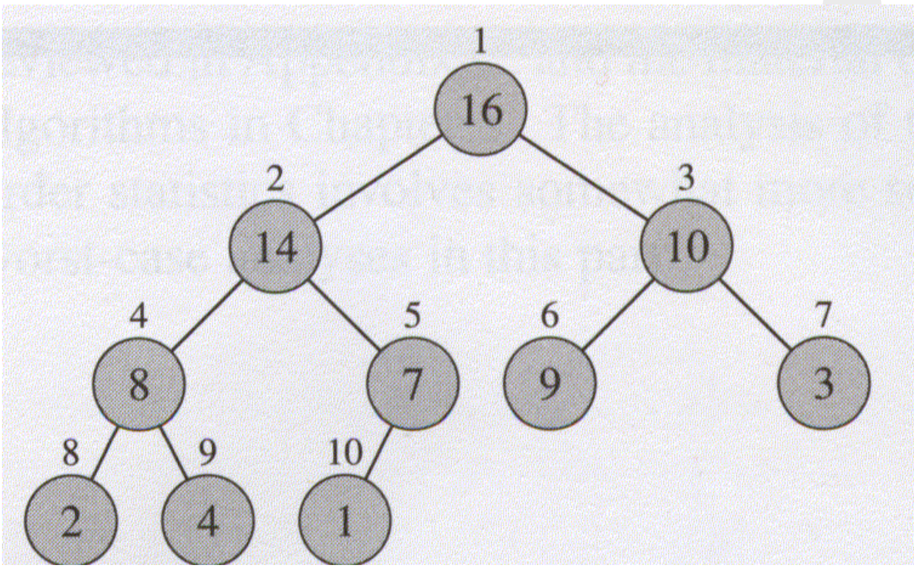
- Insertion sort
  - in place: only a constant number of elements of the input array are sorted outside the array ☺
  - worst case: $O(n^2)$ ☹
- Merge sort
  - worst case: $\Theta(n\lg n)$ ☺
  - not in place ☹
- Heap sort (Chap 6)
  - in place ☺
  - worst case: $O(n\lg n)$ ☺

Sorting Algorithms (2/2)

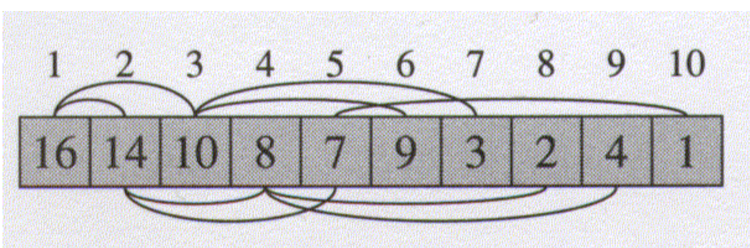
- Quick sort (Chap 7)
  - worst case: $\Theta(n^2)$ ☹
  - average case: $\Theta(n\lg n)$ ☺
  - in-place ☺
- Worst-case time complexity for all comparison sorts: $\Omega(n\lg n)$
- Counting / Radix / Bucket sort (Chap 8)
  - non-comparison sorts
  - linear-time algorithms
- Order statistics (Chap 9)
  - find i-th smallest element in $O(n)$ time

Heaps

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



A max heap



Parent(i)
return $\lfloor i/2 \rfloor$
Left(i)
return $2i$
Right(i)
return $2i+1$

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Heap Properties

- Max-heap
  - A[Parent(i)] $\geq$ A[i]
- Min-heap
  - A[Parent(i)] $\leq$ A[i]
- The height of a node in a heap (tree)
  - the number of edges on the longest simple downward path from the node to a leaf
- The height of a heap (tree)
  - the height of the root
- The height of a heap: $\Theta(\lg n)$

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Turn an Array into a Max-Heap (1/3)

```
Max-Heapify ( A, i )
      * assume binary trees rooted at Left(i) & Right(i)
        are both max-heaps already
      l ← Left (i)
      r ← Right(i)
      if l ≤ heap-size[A] and A[ l ] > A[ i ]
      * A[ i ] is smaller than its left child
            then largest ← l
            else largest ← i
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

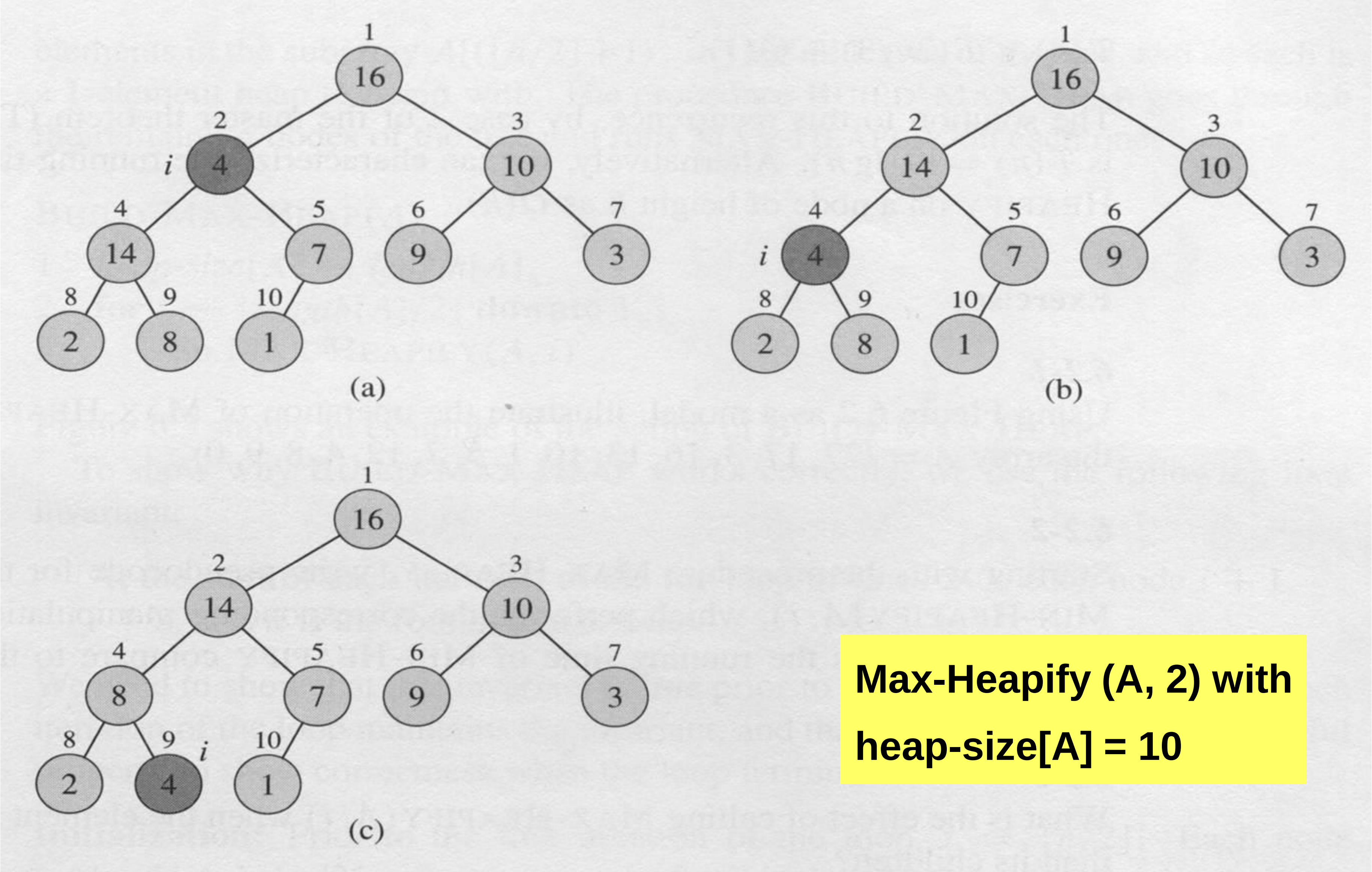
Turn an Array into a Max-Heap (2/3)

```
if r ≤ heap-size[A] and A[r] > A[largest]
* right child is the largest among l, r, i
        then largest ← r
if largest ≠ i
        then exchange A[ i ] ↔ A[largest]
                Max-Heapify (A, largest)
                * recursive call
```

Runtime of Max-Heapify on a node of height h is **O(h)**

(Note that the maximum height = $\lfloor \lg n \rfloor$)

Example



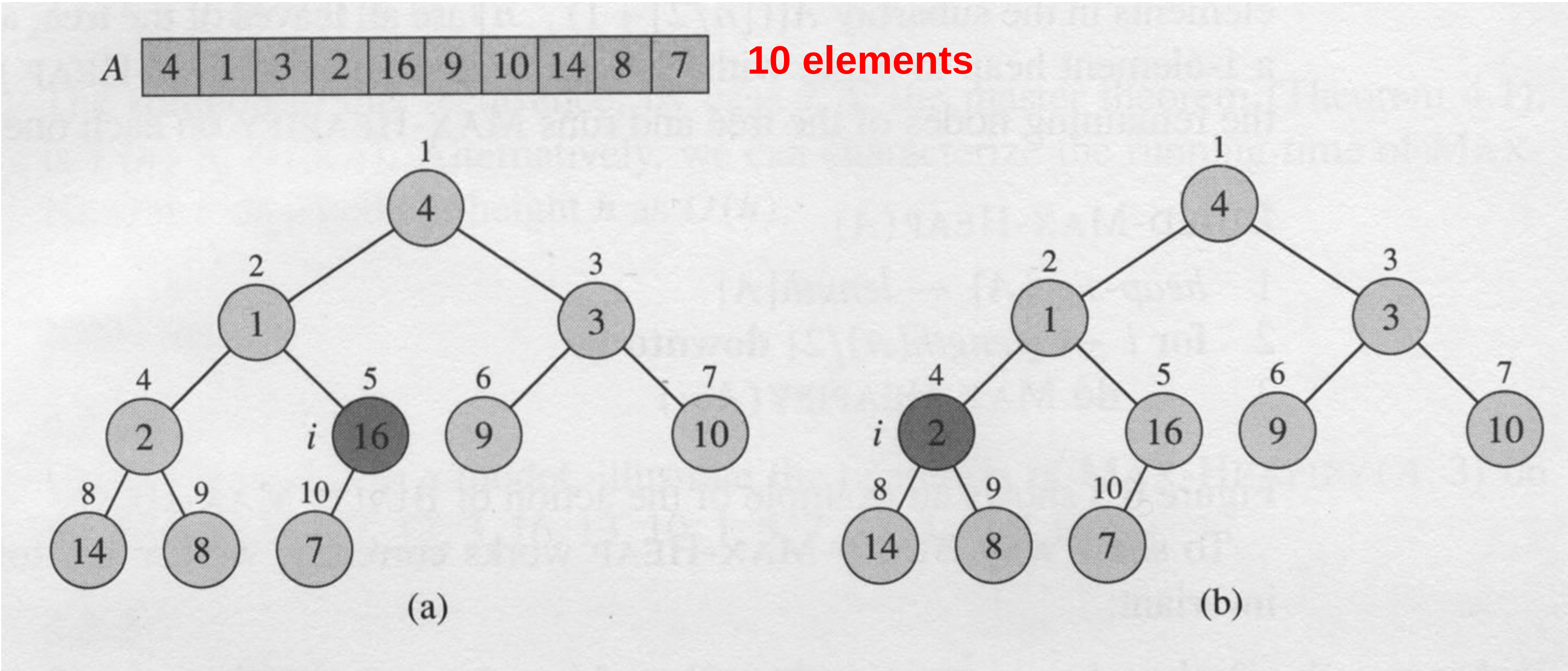
Max-Heapify (A, 2) with heap-size[A] = 10

Turn an Array into a Max-Heap (3/3)

```
Build-Max-Heap(A)
    heap-size[A] ← length[A]
    for  i ← ⌊ length[A] / 2 ⌋ downto 1
        do   Max-Heapify( A, i )
    * for k > ⌊ length[A] / 2 ⌋, A [k] is a leaf
```

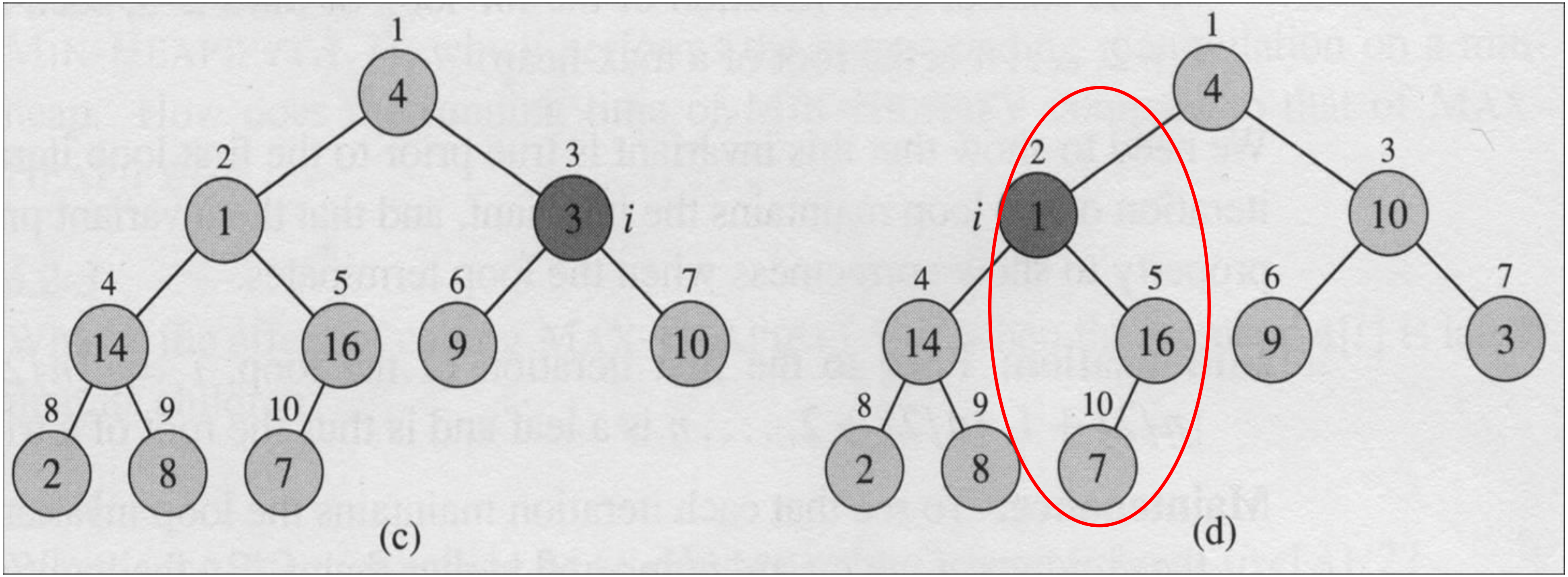
Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example (1/3)



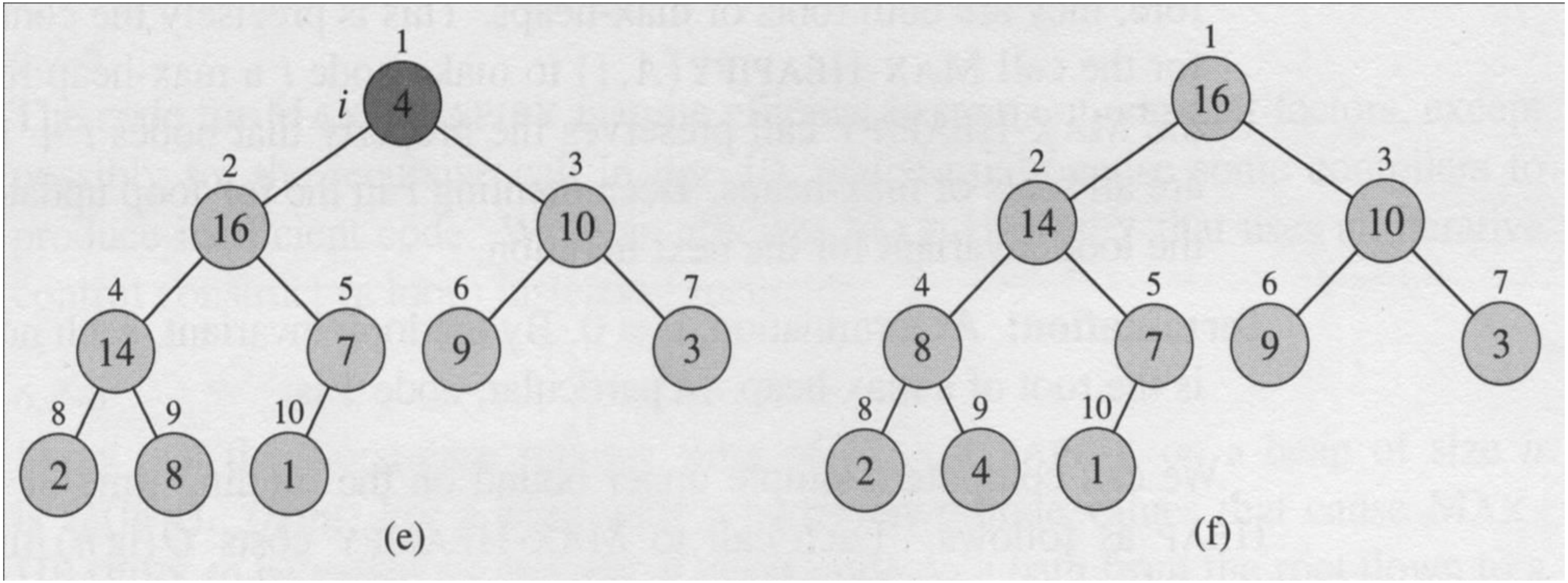
10 elements

Example (2/3)



Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example (3/3)



1
i
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1
(e)
1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
10
1
(f)

Time Complexity of Build-Max-Heap (1/2)

- An easy but **non-tight** bound
  - each call to Max-Heapify costs O(lgn)
  - there are O(n) such calls
  - ➔ O(nlgn)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Time Complexity of Build-Max-Heap (2/2)

Asymptotically tight bound

- Corollaries
  - an n-element heap has height $\lfloor \lg n \rfloor$
  - at most $\lceil n/2^{h+1} \rceil$ nodes of any height h

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

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \left(\because \sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}\right) \quad \text{(See Apendix A)}$$

$$\Rightarrow \quad O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = \boxed{O(n)}$$

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

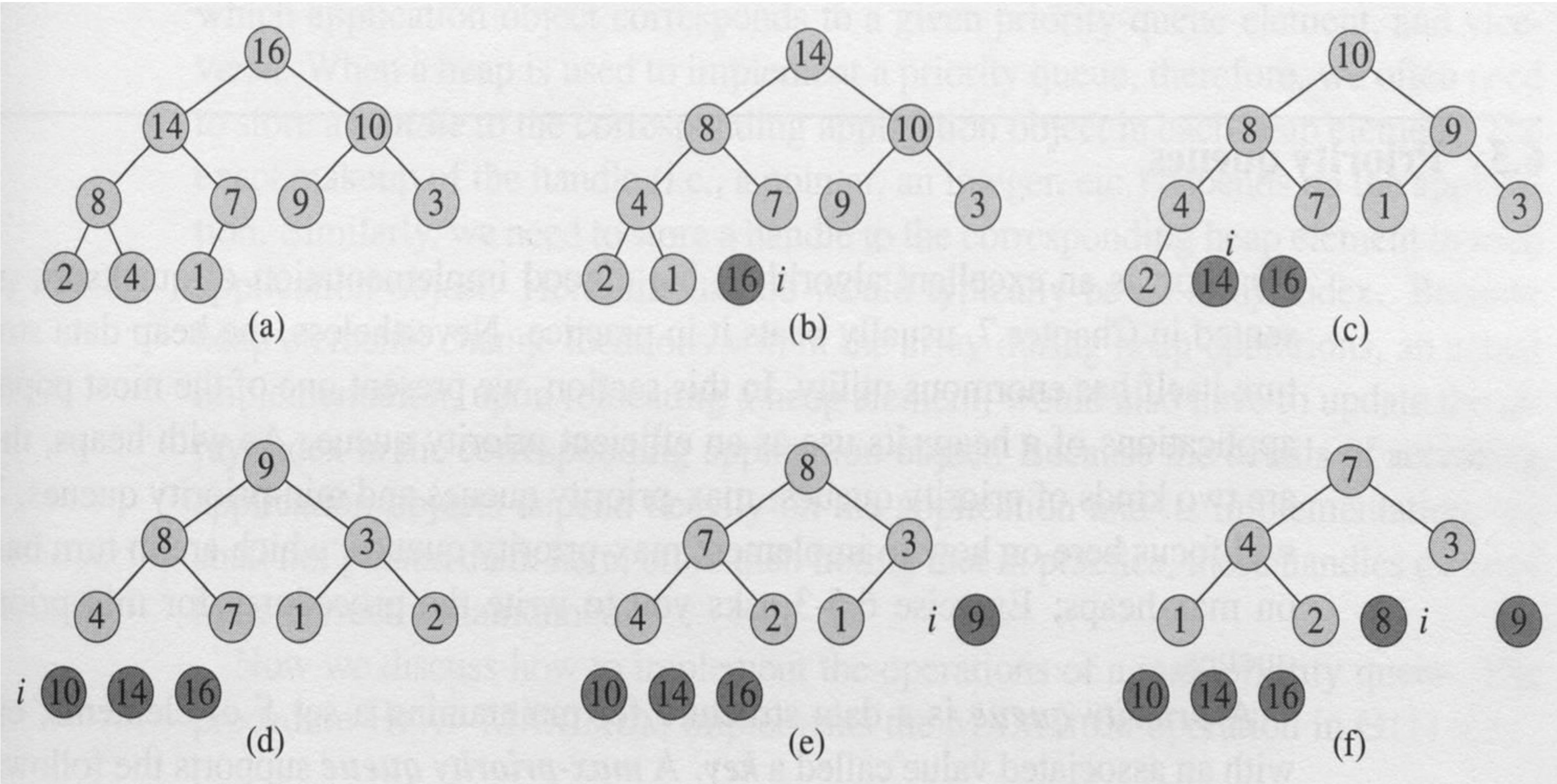
Heap Sort

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

Time complexity: **O(nlgn)**

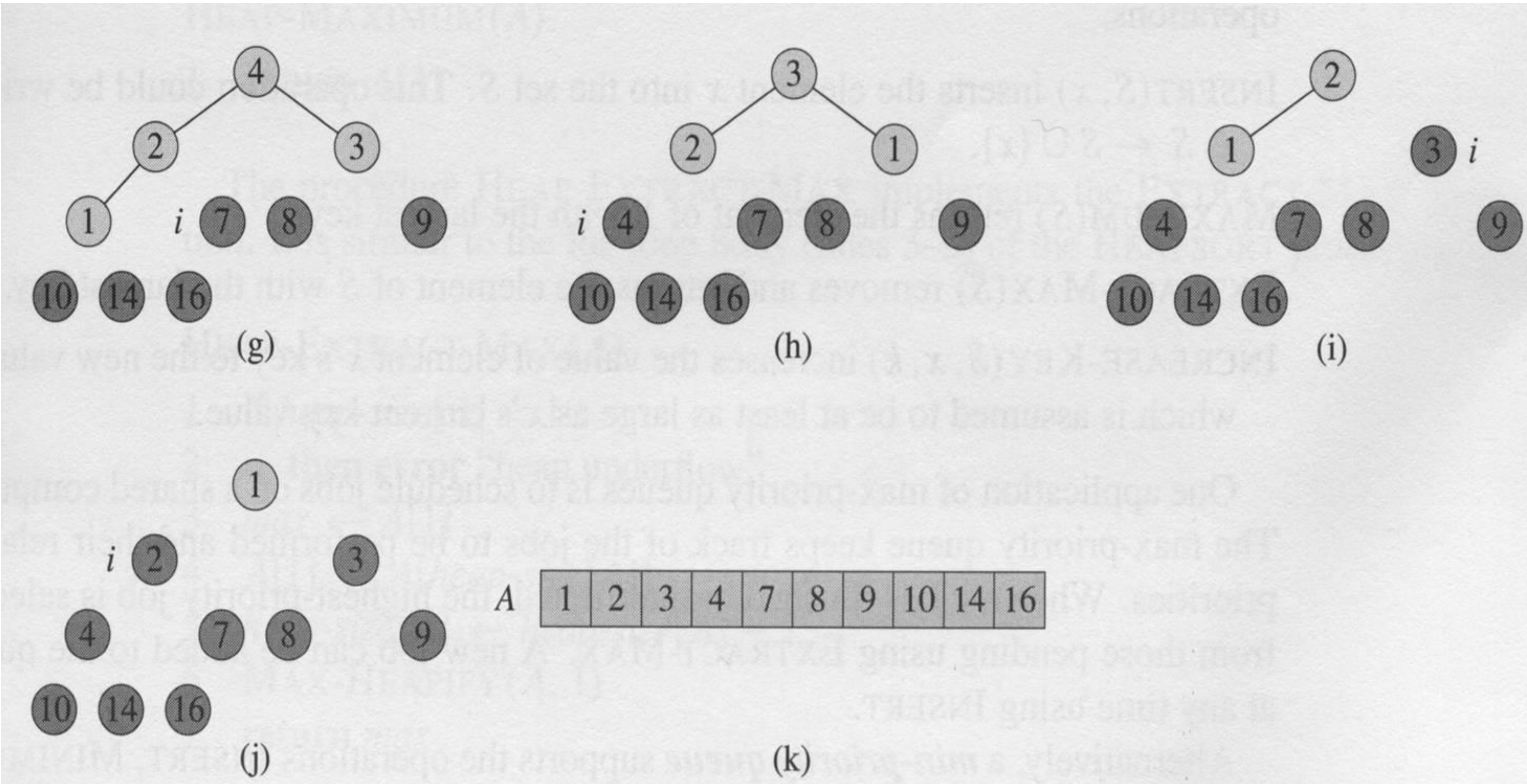
Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Example (1/2)


16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
i
2
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example (2/2)



Priority Queues

- A priority queue is a data structure that maintains a set S of elements, each with an associated value called a key
- A max-priority queue supports the following operations:
  - Insert(S, x) $O(\lg n)$
  - Maximum(S) $\Theta(1)$
  - Extract-Max(S) $O(\lg n)$
  - Increase-Key(S, x, k) $O(\lg n)$
- Of course, there is a min-priority queue
  - the dual of the max-priority queue

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Maximum and Extract-Max

```
Heap-Maximum(A)
    return A[1]
```

Θ(1)

```
Heap-Extract-Max(A)
    if  heap-size[A] < 1
        then error "heap underflow"
    max ← A[1]
    A[1] ← A[ heap-size[A] ]
    heap-size[A] ← heap-size[A] – 1
    Max-Heapify( A, 1 )
    return max
```

O(lgn)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

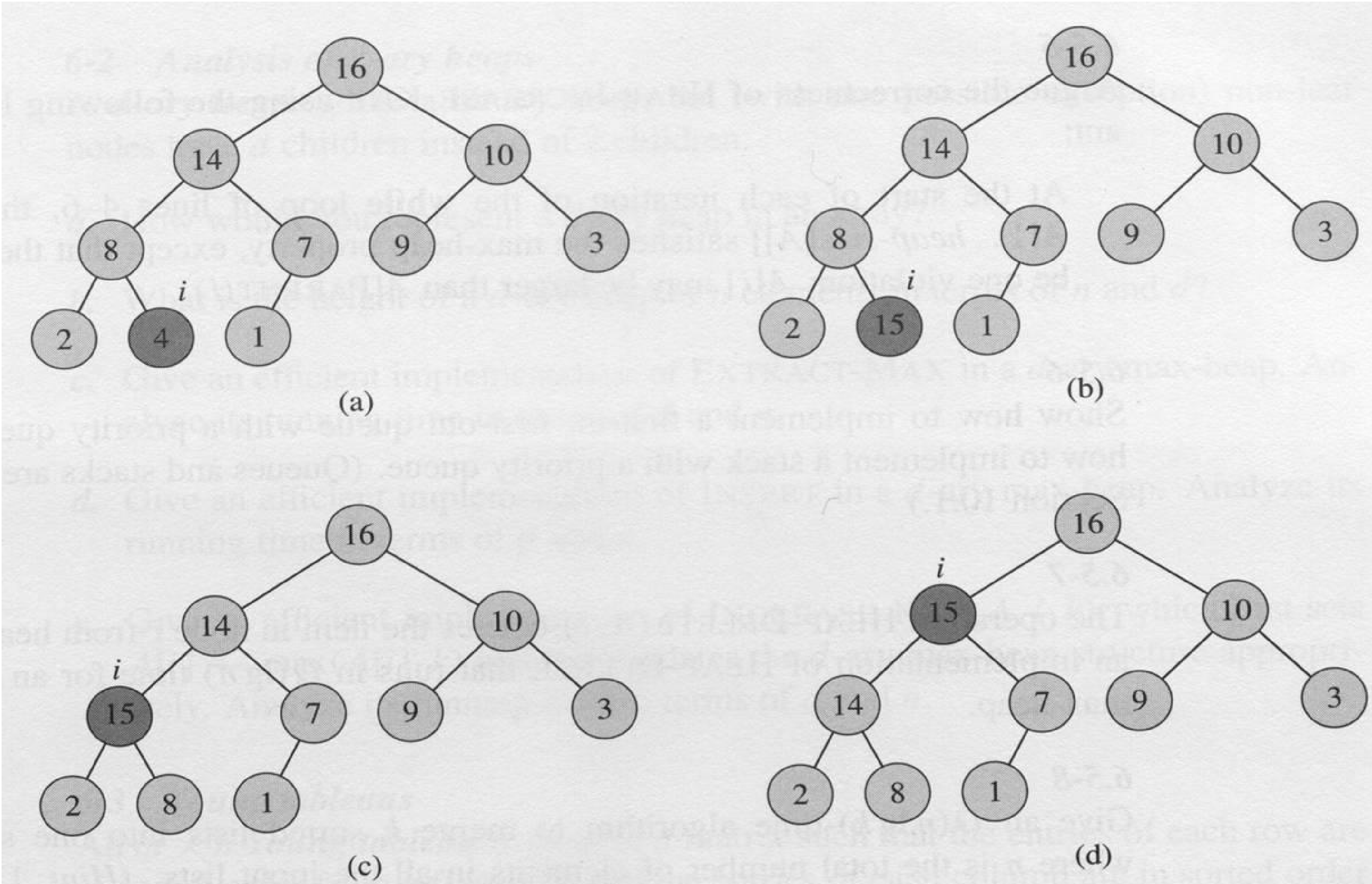
Heap-Increase-Key

Heap-Increase-Key(A, i, key) **O(lgn)**

```
if key < A[ i ]
    then error "new key < current key"
A[ i ] ← key
while i > 1 and A[Parent(i)] < A[ i ]
        do exchange A[ i ] ↔ A[Parent(i)]
            i ← Parent(i)
```

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Example


16
14
10
8
7
9
3
i
2
4
1
(a)
16
14
10
8
7
9
3
i
2
15
1
(b)
16
14
10
i
15
7
9
3
2
8
1
(c)
16
i
15
10
14
7
9
3
2
8
1
(d)

Max-Heap-Insert

Max-Heap-Insert(A, key) **O(lgn)**

1 heap-size[A] ← heap-size[A] + 1

2 A[heap-size[A]] ← –∞

3 Heap-Increase-Key(A, heap-size[A], key)

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw