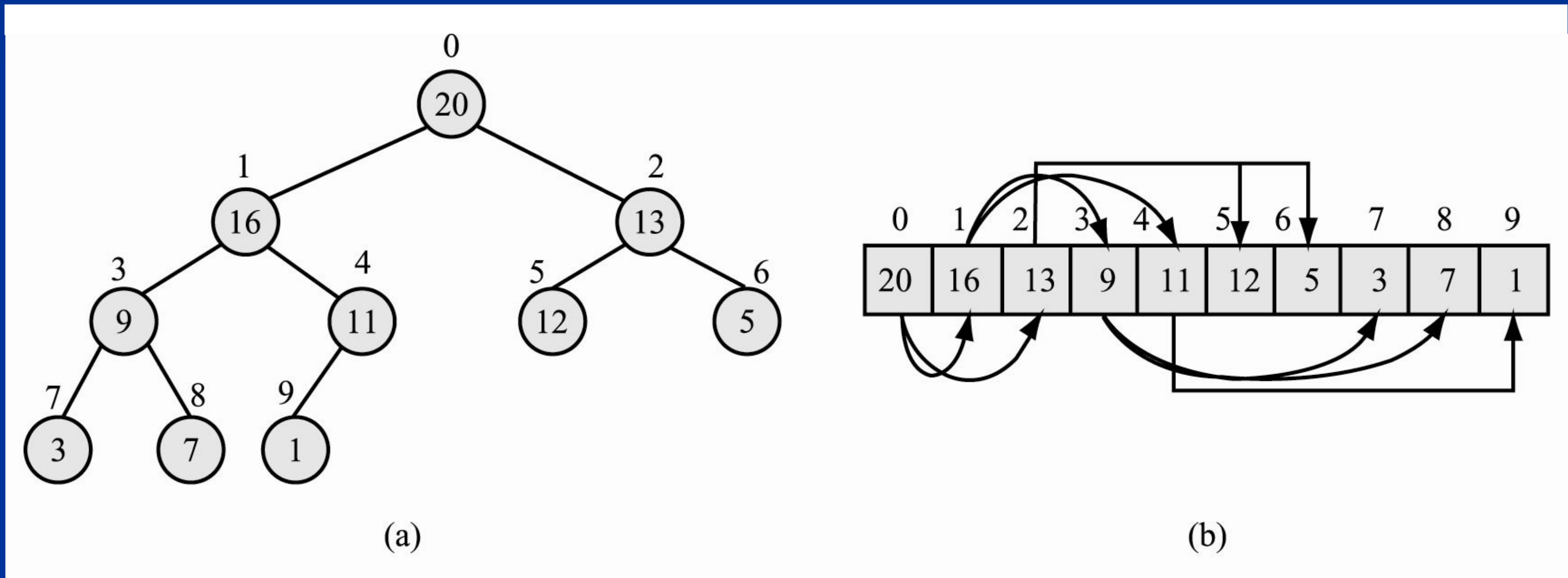
演算法 堆積排序



0
20
1
16
2
13
3
9
4
11
5
12
6
5
7
3
8
7
9
1
(a)
0 1 2 3 4 5 6 7 8 9
20 16 13 9 11 12 5 3 7 1
(b)

樹 (tree)

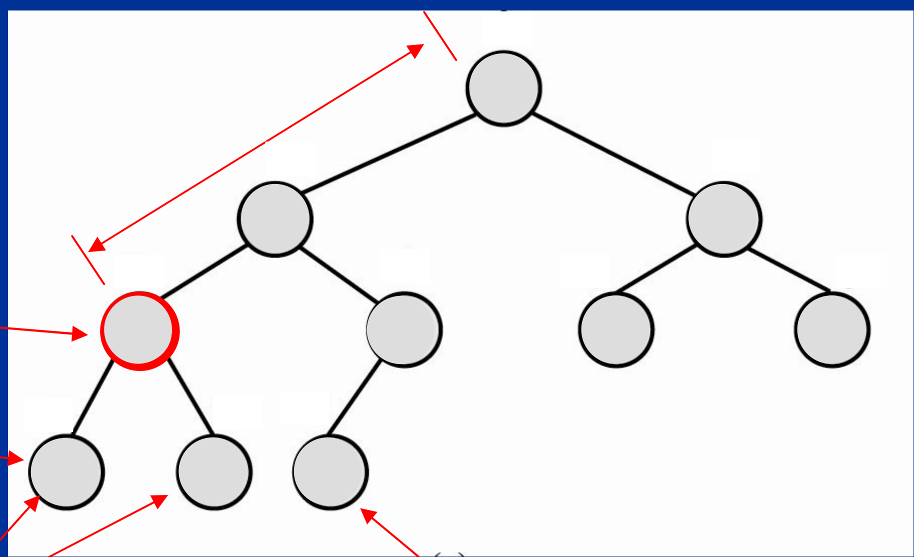
- 節點 (node) 間以邊線 (edge) 連結，節點間不循環的圖

根節點 (root)

路徑 (path)
ancestor
descendant

父節點 (parent)

子節點 (child)
子樹 (subtree)



兄弟節點 (sibling)

樹葉 (leaf node)

樹的性質

- n 元樹 (multi-way tree)
  每個節點最多可允許 n 個子節點

- 階層 (level)
  根節點的階層爲1，其子節點的階層爲2，...

- 節點的深度 (depth)
  根節點到此節點的路徑長度→經過多少個邊線

- 樹的高度 (height)
  根節點到最遠樹葉的路徑長度

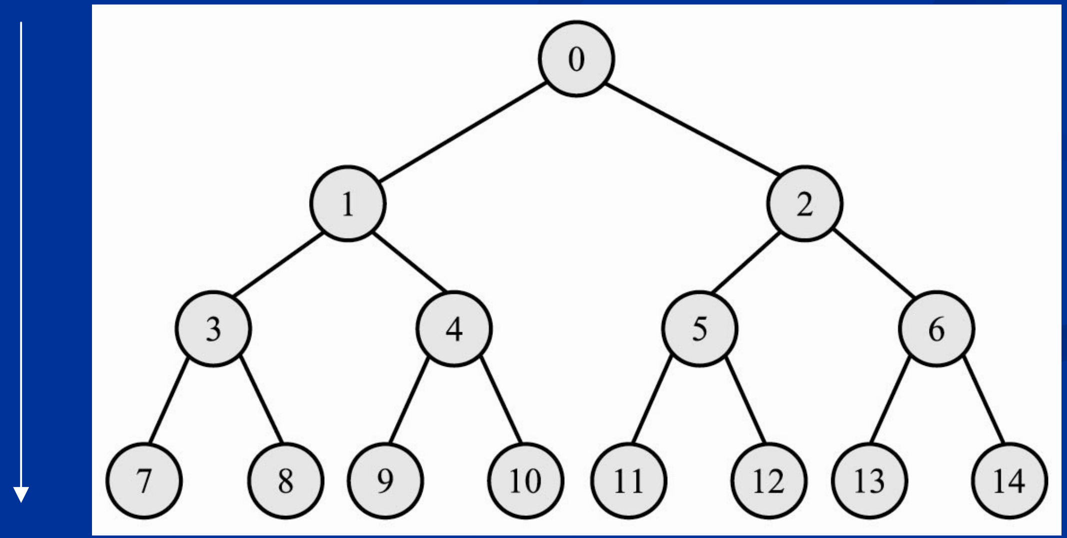
二元樹(binary tree)

- 二元樹
  - 每個節點最多只有二棵子樹
  - 子節點有順序(ordered)
    左子節點(left child)，右子節點(right child)
- 高度爲 k 的二元樹最多有 $2^{k+1} - 1$ 個節點，最少有 k+1 個節點
- 包含 n 個節點的二元樹，其高度最大爲 n-1，最小爲 $[\log_2 n]$

滿二元樹 (full binary tree)

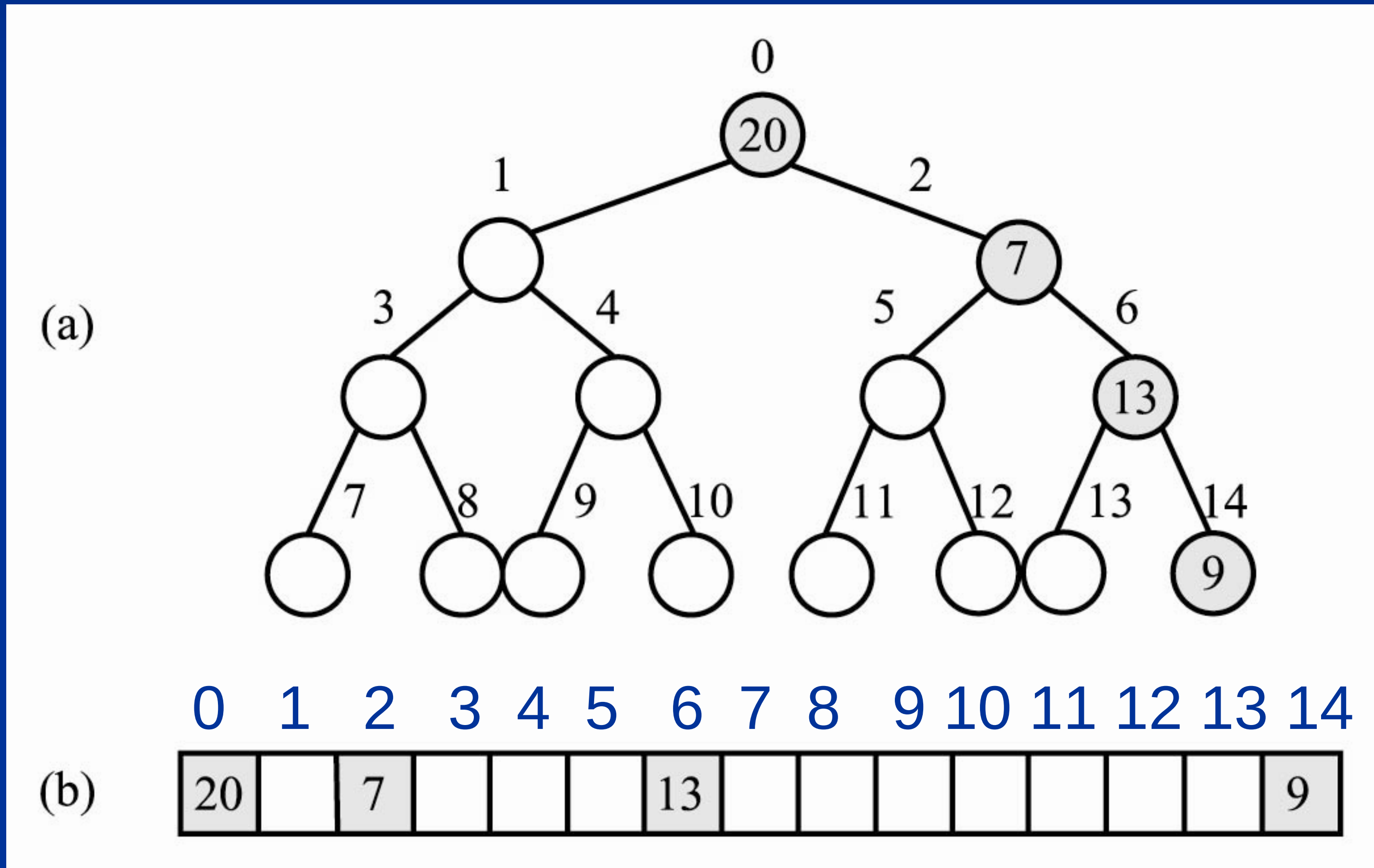
- 一棵二元樹的高度為 k 且有 $2^{k+1} - 1$個節點
- 最底層的節點都是葉子，其它層的節點都有 2 個子節點
- 每個節點依由上到下、由左到右的順序，從 0 到 $2^{k+1} - 2$ 進行編號

高度 k=3



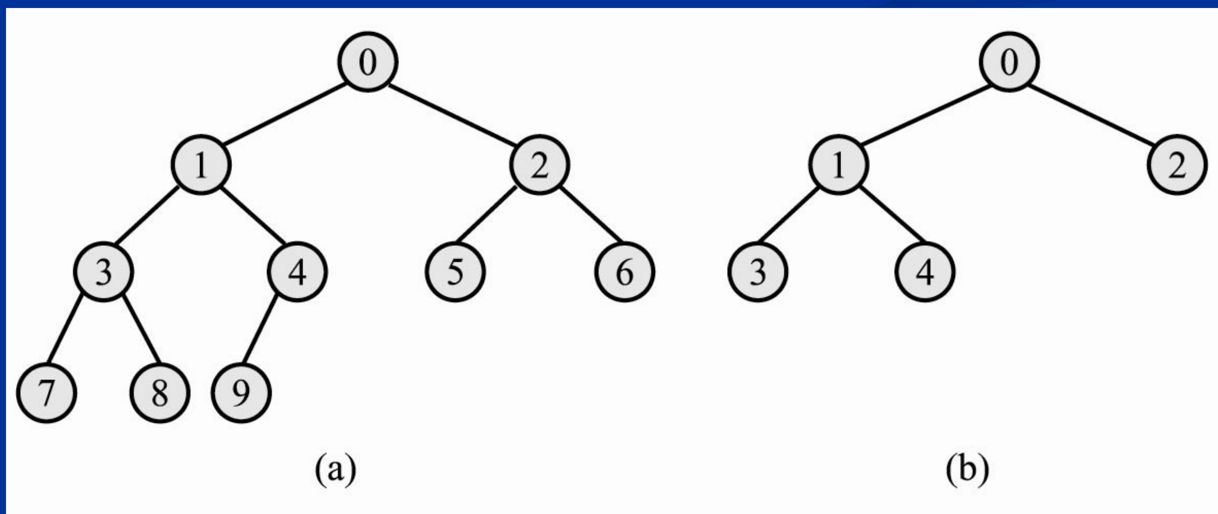
以陣列表示右傾斜二元樹

高度 3，須使用長度為 2^{3+1}-1=15 的陣列



完整二元樹 (complete binary tree)

- 一棵有 n 個節點的二元樹，按照滿二元樹的編號方式對它進行編號，樹中所有節點和滿二元樹 0 到 n-1 編號完全一致
- 有 n 個節點，樹的高度爲 $[\log_2 n]$

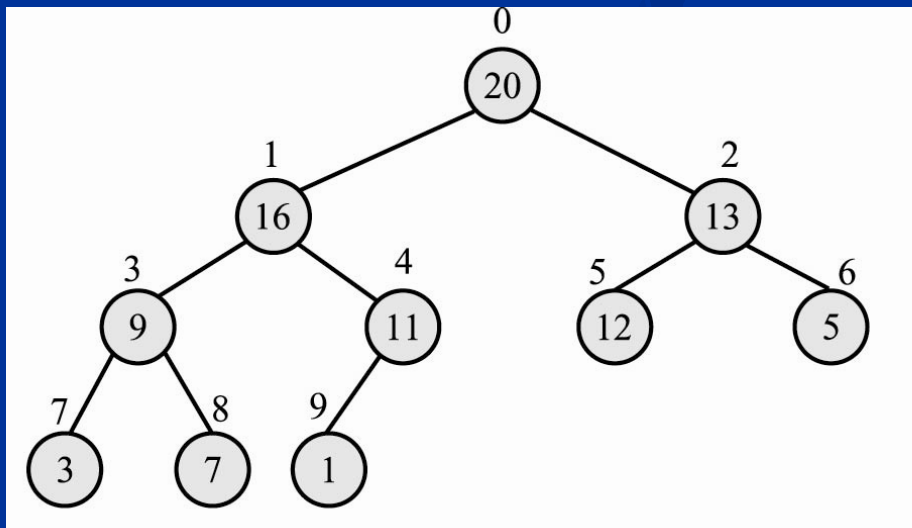


完整二元樹

- 對於編號為 i 的節點，0 ≤ i ≤ n-1，則以下關係成立
  - 若編號 i 的節點不是根節點，其父節點編號為 [(i-1)/2] (取整數)
  - 編號 i 的節點，其左子節點的編號為 2i+1
  - 編號 i 的節點，其右子節點的編號為 2i+2

資料結構 堆積 (heap)

- 節點具有鍵值 (key) 的完整二元樹
  - 依鍵值規則可分成兩種
  - 最大堆積
    每一個節點的值，都不小於其子節點的值
  - 最小堆積
    每一個節點的值，都不大於其子節點的值



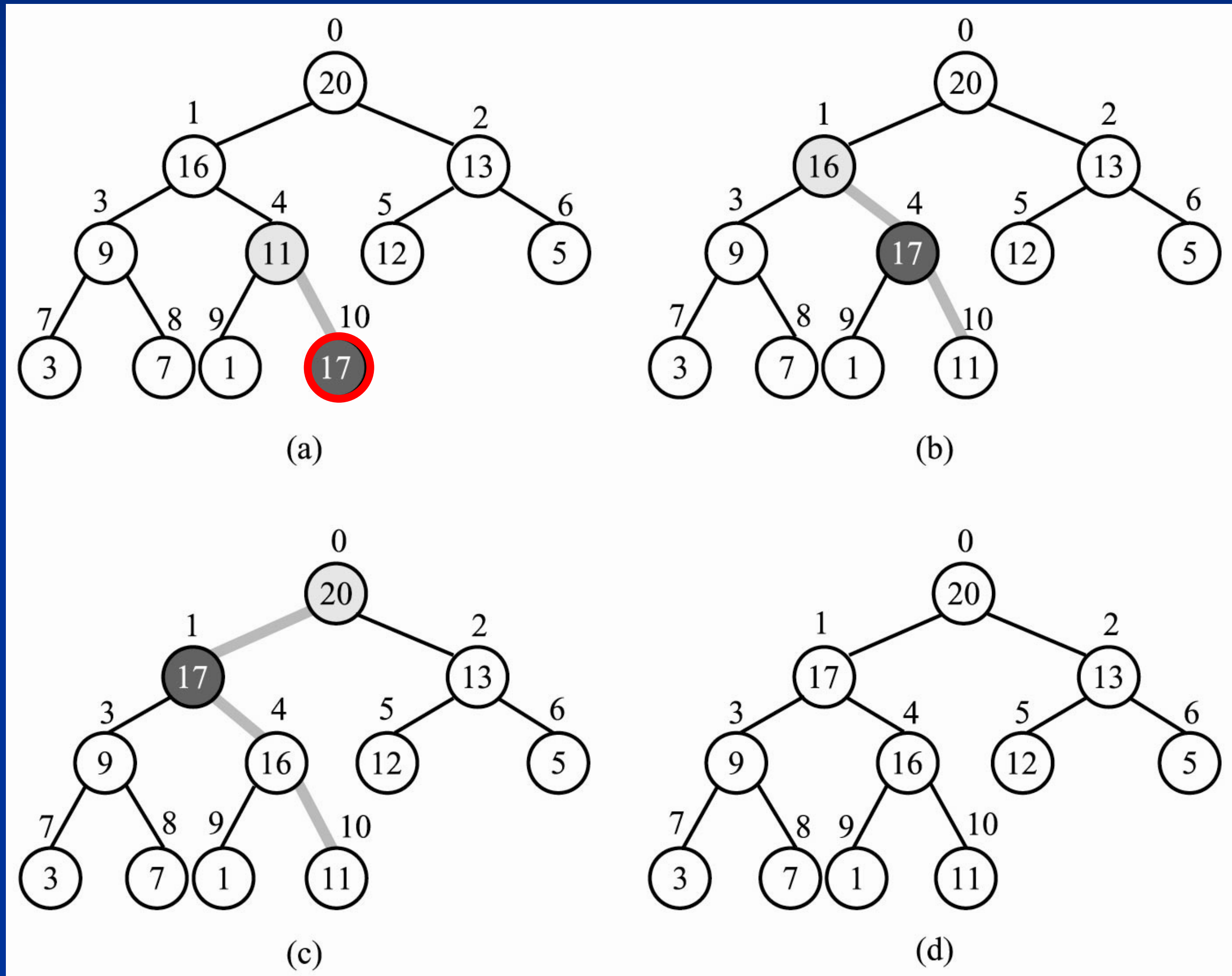
堆積

- 以陣列表示，索引值之計算
  - 若節點 i 不是根節點，其父節點為 [(i-1)/2]
  - 節點 i 的左子節點為 2i+1
  - 節點 i 的右子節點為 2i+2

```
#define ParentIndex(i)      i==0?0:(i-1)/2
#define LeftChildIndex(i)          (i)*2+1
#define RightChildIndex(i)        (i)*2+2
```

堆積的新增

先加到最後面，再往上比較，直到滿足堆積規則



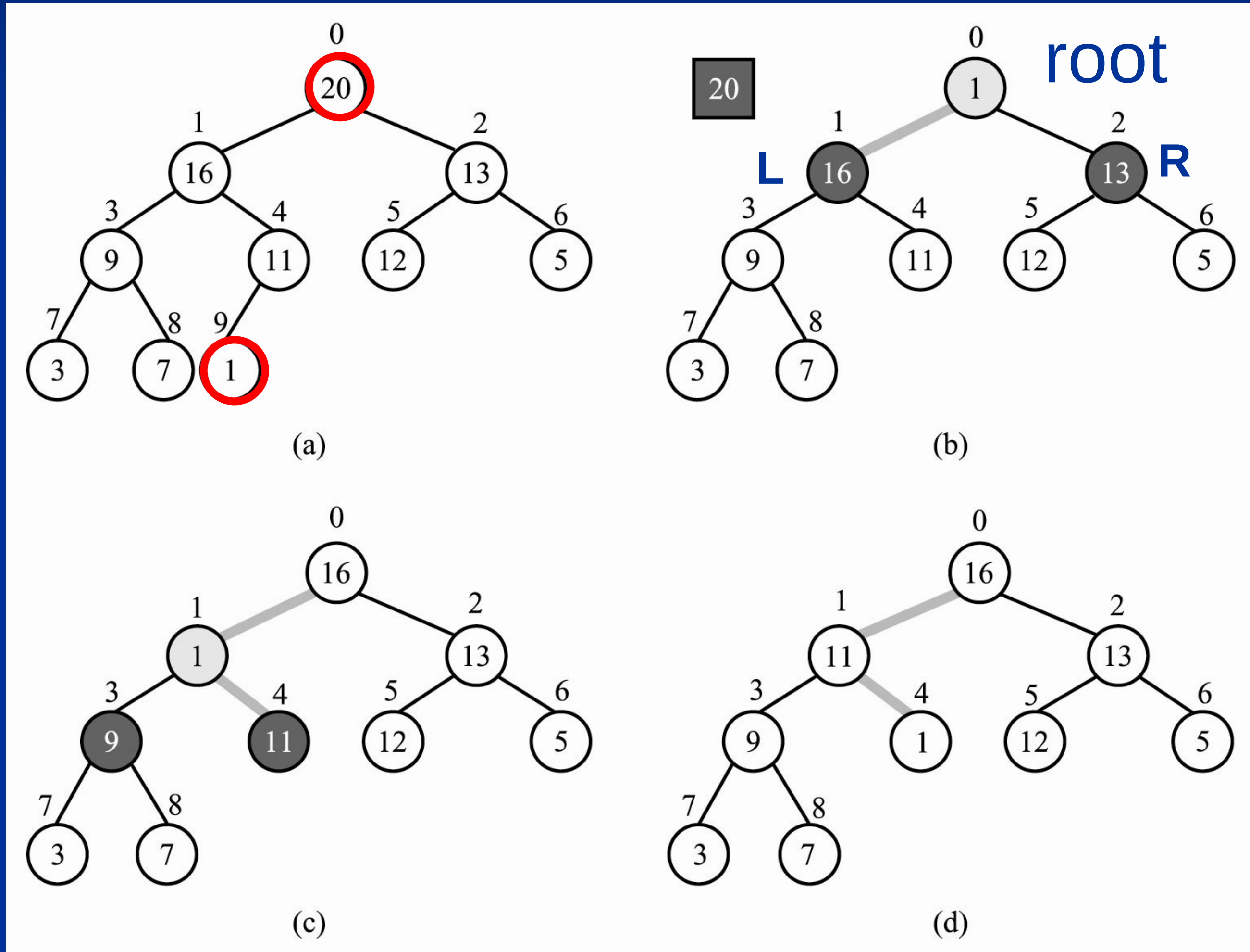
新增至最大堆積

```
void MaxHeapAdd(int A[], int n){
        int i, j;
        i=n;
        do {
                j=ParentIndex(i);
                if (A[i]>A[j]) swap(A, i, j);
                else return;
                i=j;
        } while (i!=0);
}
```

樹的高度為 $[\log_2 n]$，時間複雜度 $O(\mathit{log}\ n)$

堆積的刪除

取出根節點後，由最後的葉子取代，再重整堆積



堆積的刪除

```
int MaxHeapDelete(int A[], int n){
        swap(A, 0, n-1);
        MaxHeapBalance(A, n-1, 0);
        return A[n-1];
}
```

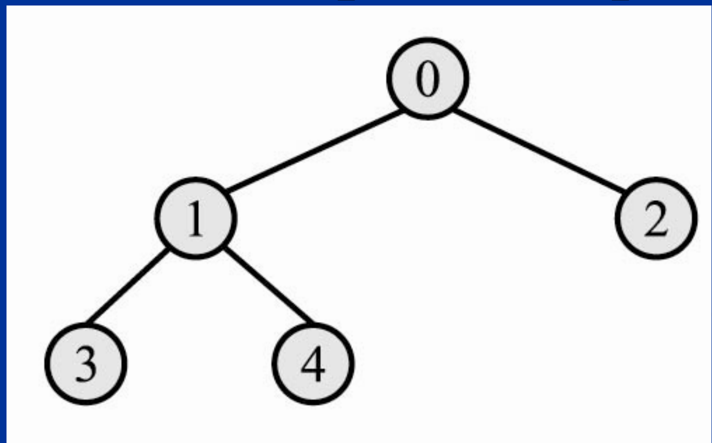
```
void MaxHeapBalance(int A[], int n, int root){
    int i, L, R,  bigger;
    bigger=root;
    while(1){
        i=bigger;
        L=LeftChildIndex(i); R=RightChildIndex(i);
        if( (R<n) && (A[R]>A[L]) ) bigger=R;
                else if (L<n) bigger=L;
                else bigger=root;
        if( (bigger>root) && (A[i]<A[bigger]) )
                swap(A, i, bigger);
        else return;
    }
}
```

堆積的重整
(heapify)

時間正比於樹的高度 $\lfloor \log_2 n \rfloor$，時間複雜度 O(*log* n)

建立堆積 由上而下

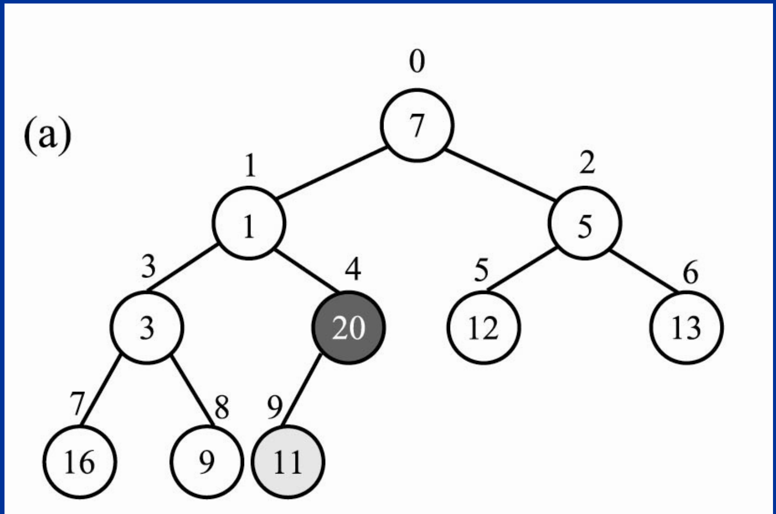
- 新增 A[0] 開始
- 使用 MaxHeapAdd , 依序加入 A[1]…A[n-1]
- 最後就可以得到 A[0…n-1] 的堆積



- 時間複雜度: O(n*log*n)
  - MaxHeapAdd 需時 O(*log*n), 共執行n-1次

建立堆積 由下而上

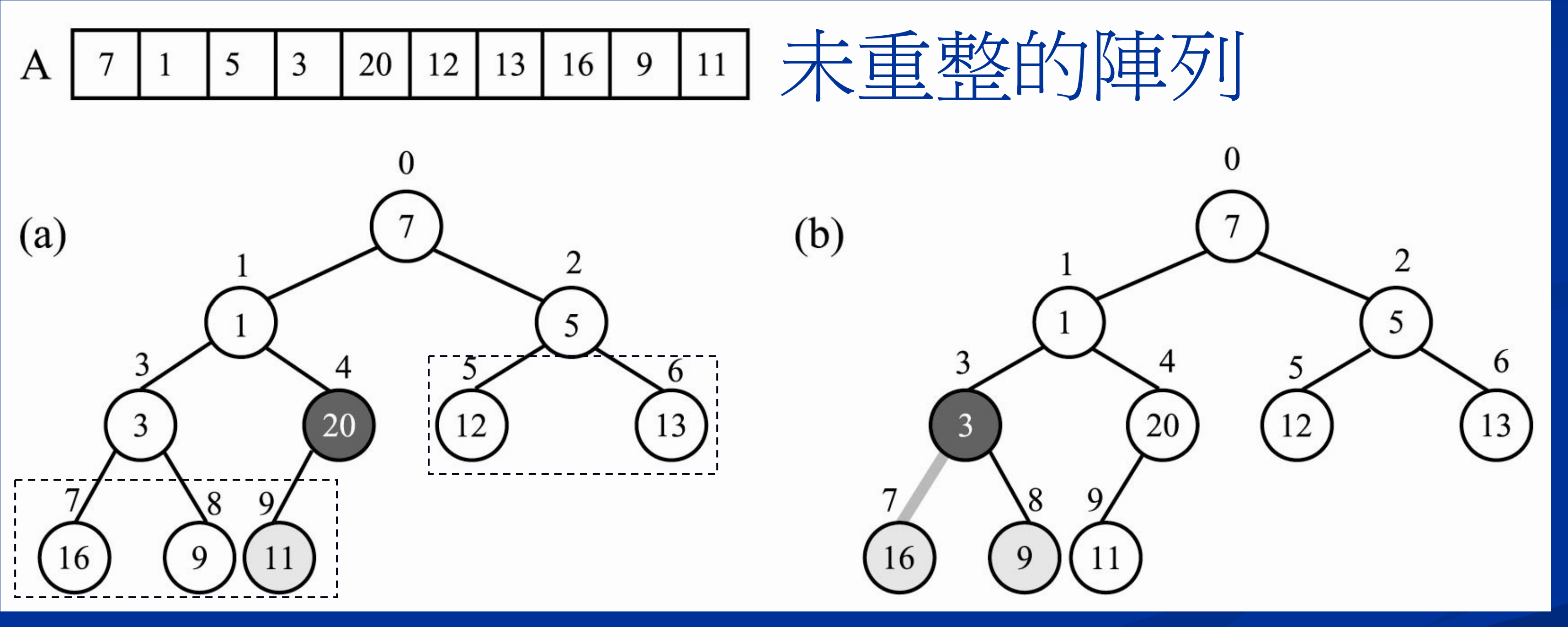
- 非葉子共 [n/2] 個，葉子共有 n-[n/2] 個
- 葉子不須重整，其他節點 A[[n/2]-1]…A[0] 使用 MaxHeapBalance 依序重整，即完成



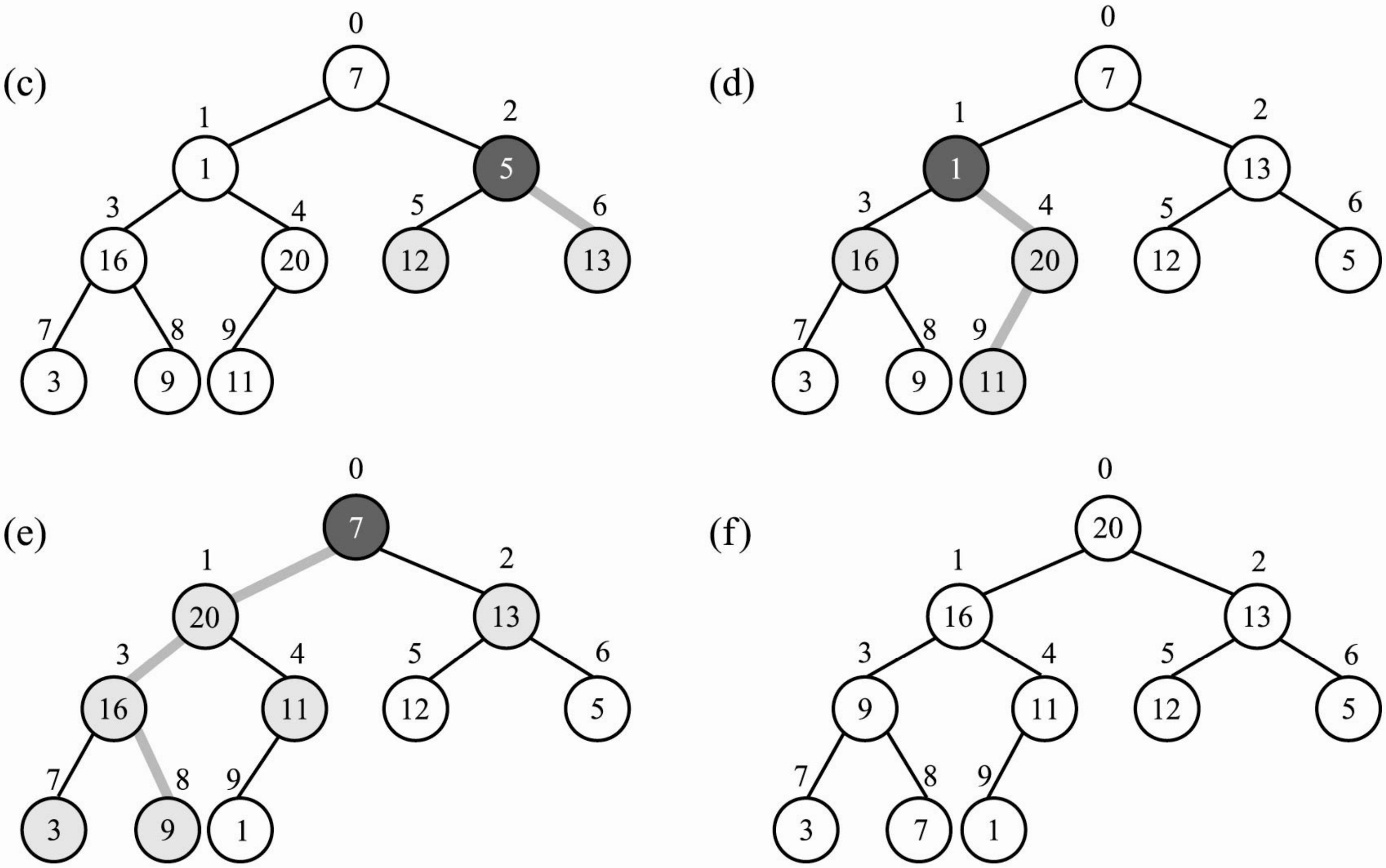
有 n=10 個節點
葉子有 5 個
非葉子 A[4]...A[0] 有 5 個

- 時間複雜度: $\Theta(n)$

由下而上
線性時間建立堆積



葉子不須重整



重整後的堆積

A	20	16	13	9	11	12	5	3	7	1

練習 建立堆積

```
void BuildMaxHeap( int A[], int n){
   int root;
   for(root = n/2-1; root>=0; root--)
        MaxHeapBalance(A, n, root);
}
```

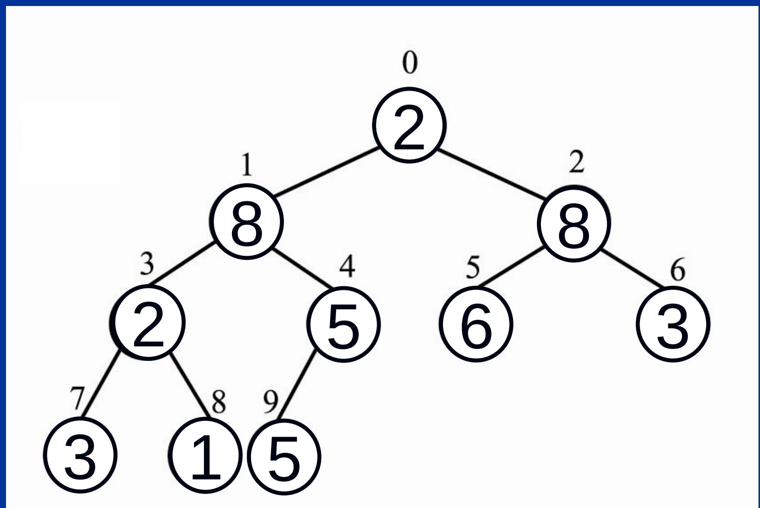
- 範例 6-1 將任意隨機數列整理為最大堆積
- 範例 6-2 將任意隨機數列整理為最小堆積

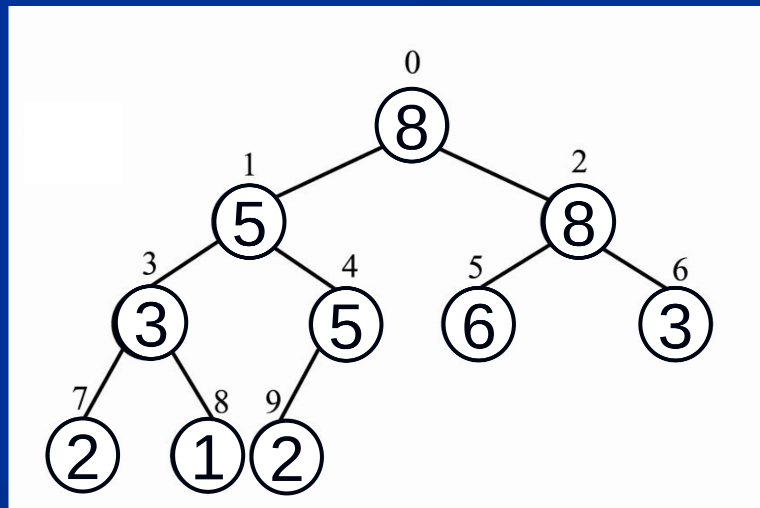
範例 6-1

- 輸出結果

  n=10

  2 8 8 2 5 6 3 3 1 5　　　　8 5 8 3 5 6 3 2 1 2





由下而上建立堆積的時間

- $n=2^{h+1}-1$, 樹高 $h=[\log_2 n]$, 每層有 2^i 個
- 每層須與 h-i 個下層比較, 每個元素比較 2 次

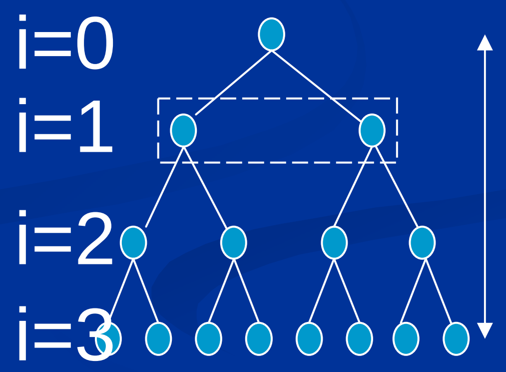
最糟情況要比較

$$C(n)=\sum_{i=0}^{h-1}2^i(h-i)2=2h\sum_{i=0}^{h-1}2^i-2\sum_{i=0}^{h-1}2^i i$$

$$=2\left[h\left(\frac{2^h-1}{2-1}\right)-\left(h2^h-2^{h+1}+2\right)\right]$$

$$=2(2^{h+1}-h-2)=2(n-\log_2 n-1)$$

$$T(n)=T_0C(n)\in\Theta(n)$$



n=15
h=3

堆積排序

1) 將輸入陣列建立成最大堆積的結構

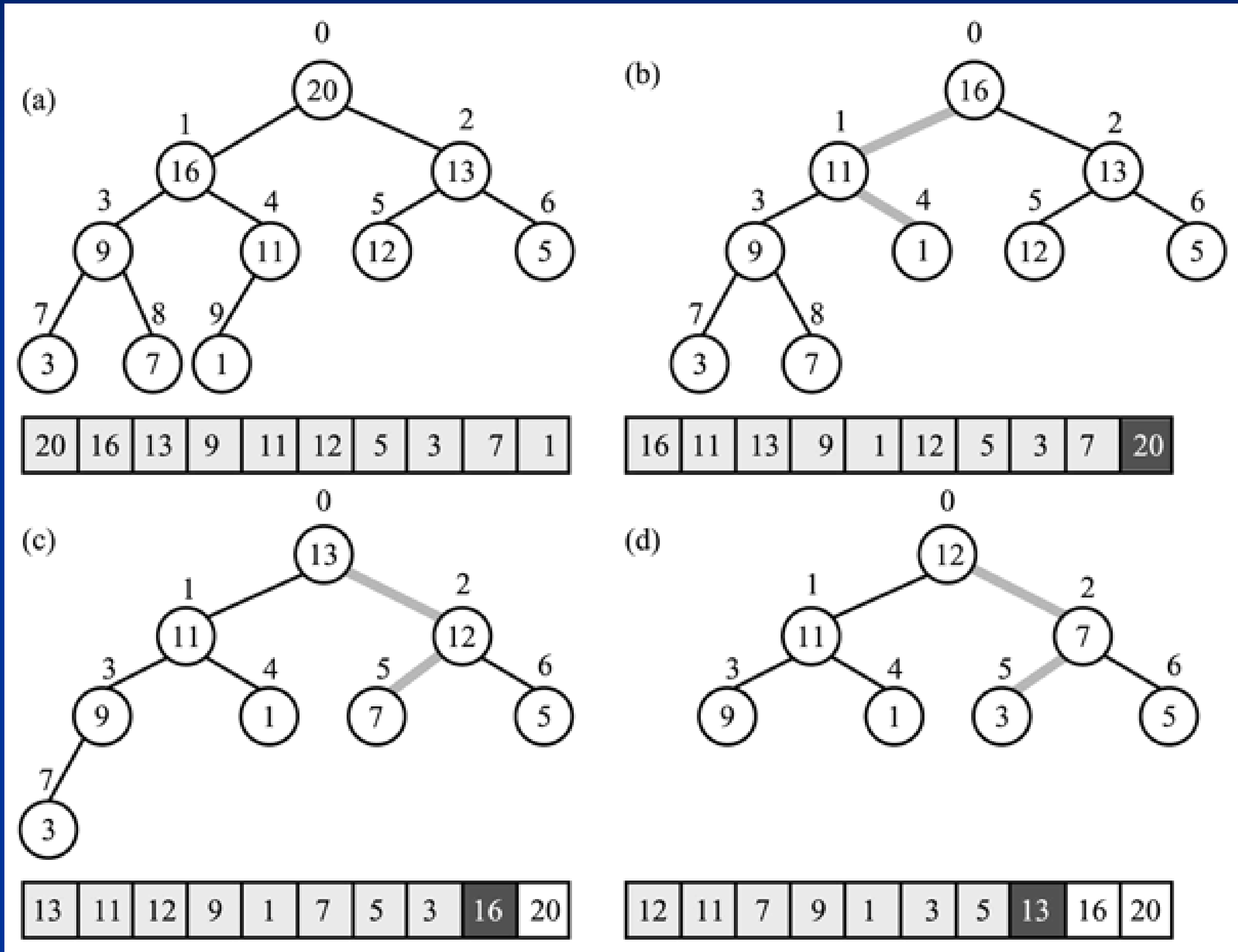
2) 重複取出最大值 (根節點) 並放到堆積陣列後面，直到堆積陣列長度為1

即完成「由小到大」的排序

- 步驟 1) 需時 Θ (n)
- 步驟 2) 每次「取出」需時 O (*log* n)，共執行n-1次，即需時 O (n *log* n)

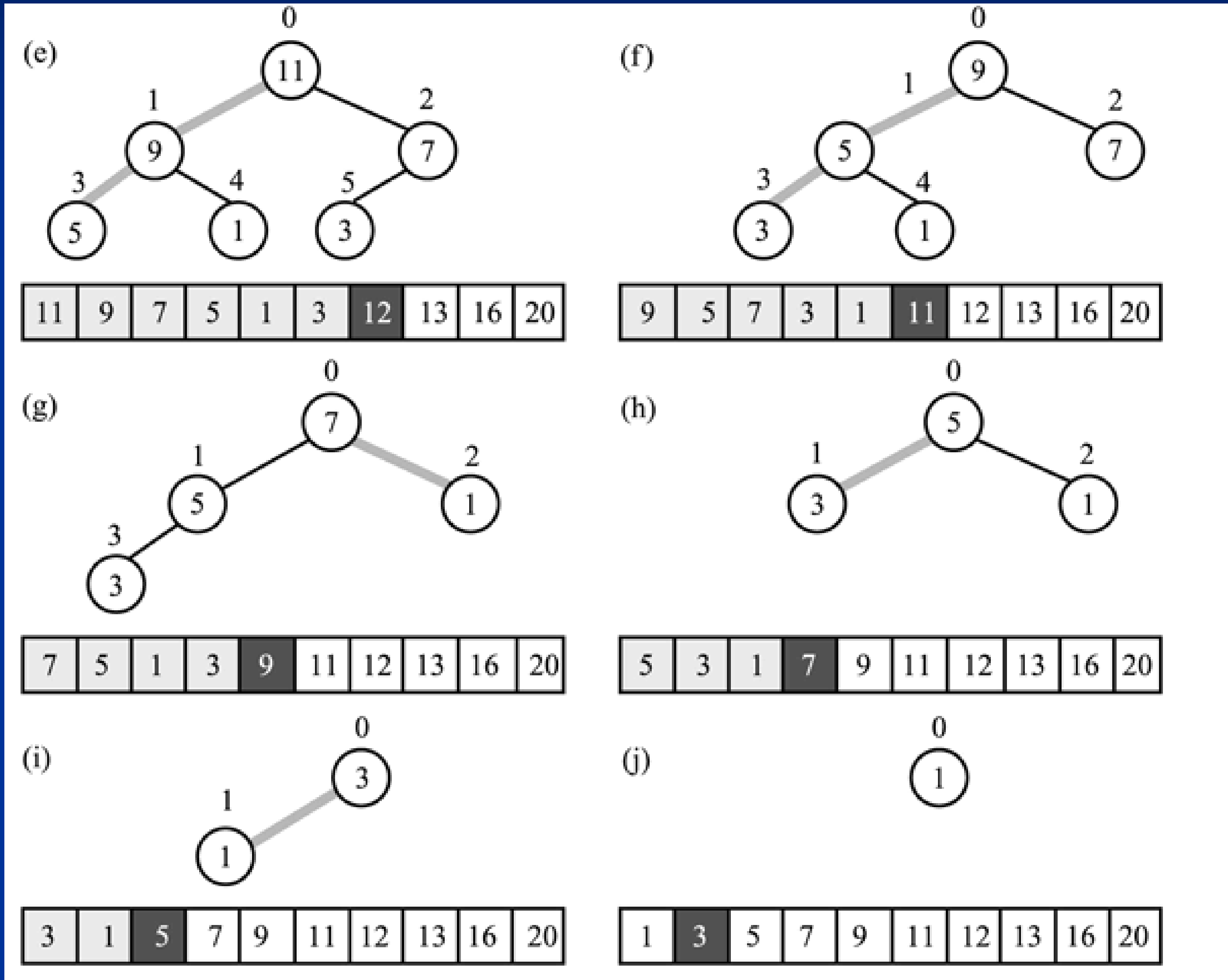
- 最糟情況、平均時間複雜度: O (n *log* n)

堆積排序過程之範例



(a)
0
20
1
16
2
13
3
9
4
11
5
12
6
5
7
3
8
7
9
1
20 16 13 9 11 12 5 3 7 1
(b)
0
16
1
11
2
13
3
9
4
1
5
12
6
5
7
3
8
7
16 11 13 9 1 12 5 3 7 20
(c)
0
13
1
11
2
12
3
9
4
1
5
7
6
5
7
3
13 11 12 9 1 7 5 3 16 20
(d)
0
12
1
11
2
7
3
9
4
1
5
3
6
5
12 11 7 9 1 3 5 13 16 20

堆積排序過程之範例



(e)
11 9 7 5 1 3 12 13 16 20
(f)
9 5 7 3 1 11 12 13 16 20
(g)
7 5 1 3 9 11 12 13 16 20
(h)
5 3 1 7 9 11 12 13 16 20
(i)
3 1 5 7 9 11 12 13 16 20
(j)
1 3 5 7 9 11 12 13 16 20

範例 6-3

堆積排序

```
void HeapSort(int A[], int n){
   int i;
   BuildMaxHeap(A, n);
   for(i=n; i>1; i--)
        MaxHeapDelete(A, i);
}
```

不須配置其他記憶體 → 在地排序

空間複雜度 $\Theta(1)$

堆積排序

- 測試結果

n=5 4 2 1 3 4

1 2 3 4 4

n=10 5 6 8 2 0 3 6 5 1 7

0 1 2 3 5 5 6 6 7 8

n=15 5 7 10 3 4 0 10 4 14 7 1 2 2 9 3

0 1 2 2 3 3 4 4 5 7 7 9 10 10 14

效能測試

- 設定較大的 n 值來測試演算法

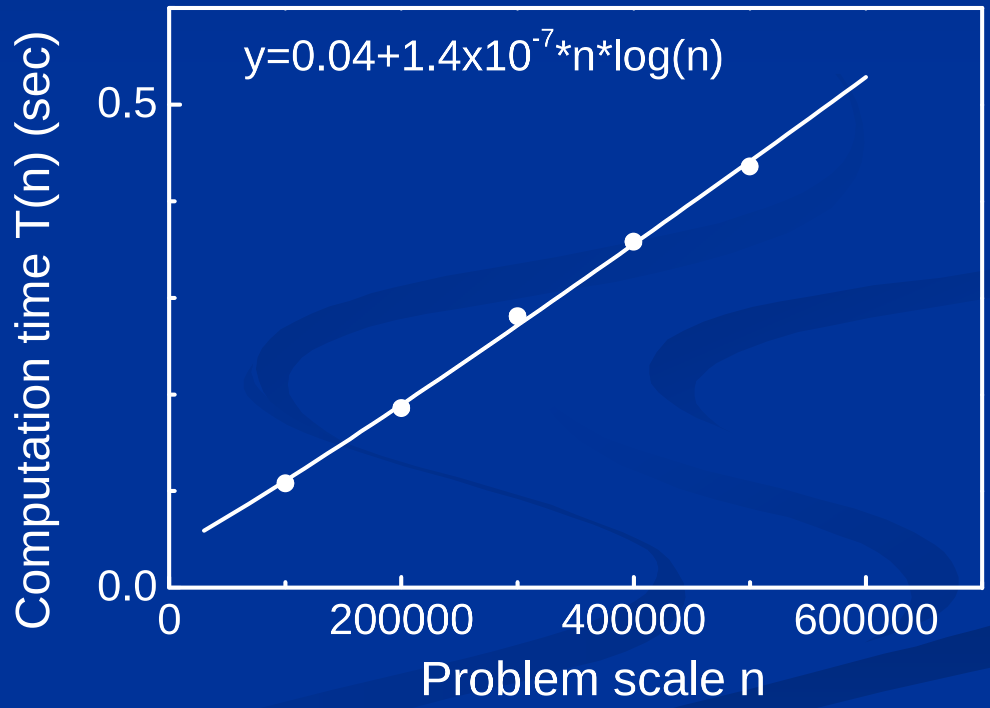
n=100000 0.108s

n=200000 0.186s

n=300000 0.281s

n=400000 0.358s

n=500000 0.436s



優先權佇列

20	16	13	9	11	12	5	3	7	1

- 佇列
  - 一個具有「先進先出」特性的資料結構
  - 最大優先權佇列
    每次取出的是佇列中數值最大的元素
  - 最小優先權佇列
    每次取出的是佇列中數值最小的元素

最大優先權佇列的操作

- 將一元素加入佇列
  對應到「堆積的新增」
  時間複雜度為 O(*log* n)

- 將數值最大的元素從佇列中取出
  對應到「堆積的刪除」
  時間複雜度為 O (*log* n)