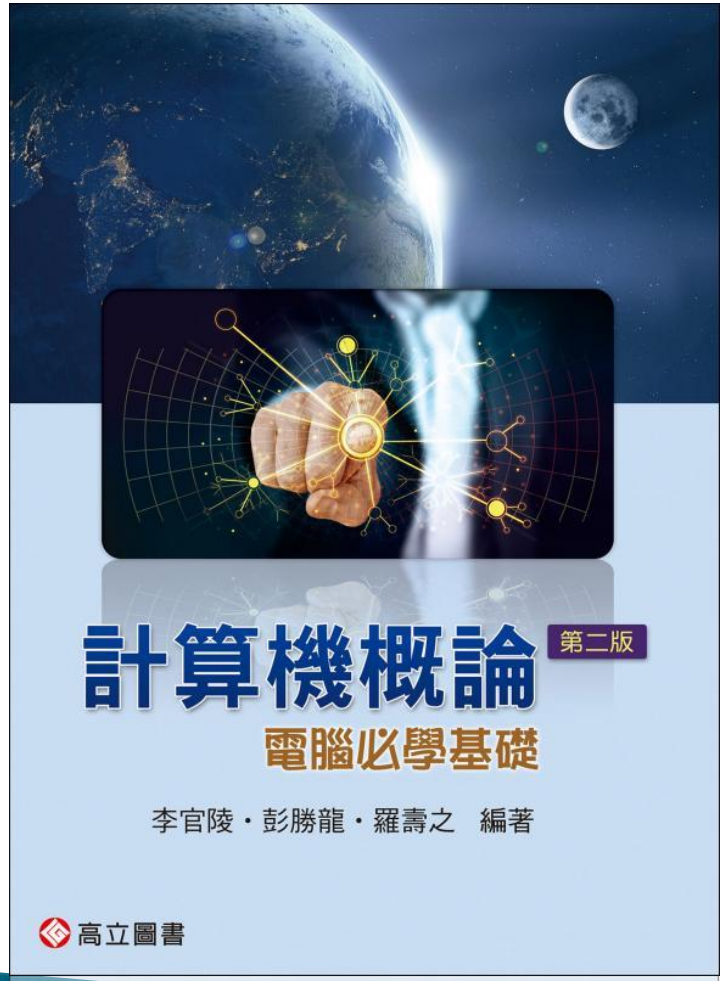


Ch08
資料結構

李官陵 彭勝龍 羅壽之

大綱

- 基本資料結構
  - 陣列 (array)
  - 連結串列 (linked list)
- 樹狀資料結構
  - 二元搜尋樹 (binary search tree)
  - 堆積 (heap)
- 抽象資料結構
  - 堆疊 (stack)
  - 佇列 (queue)

基本資料結構

- 陣列：就像是一個表格 (如圖 8.1)，存放在記憶體上的一塊連續位置，每一格是一個陣列元素，我們可以藉由其編號直接存取到它的資料。

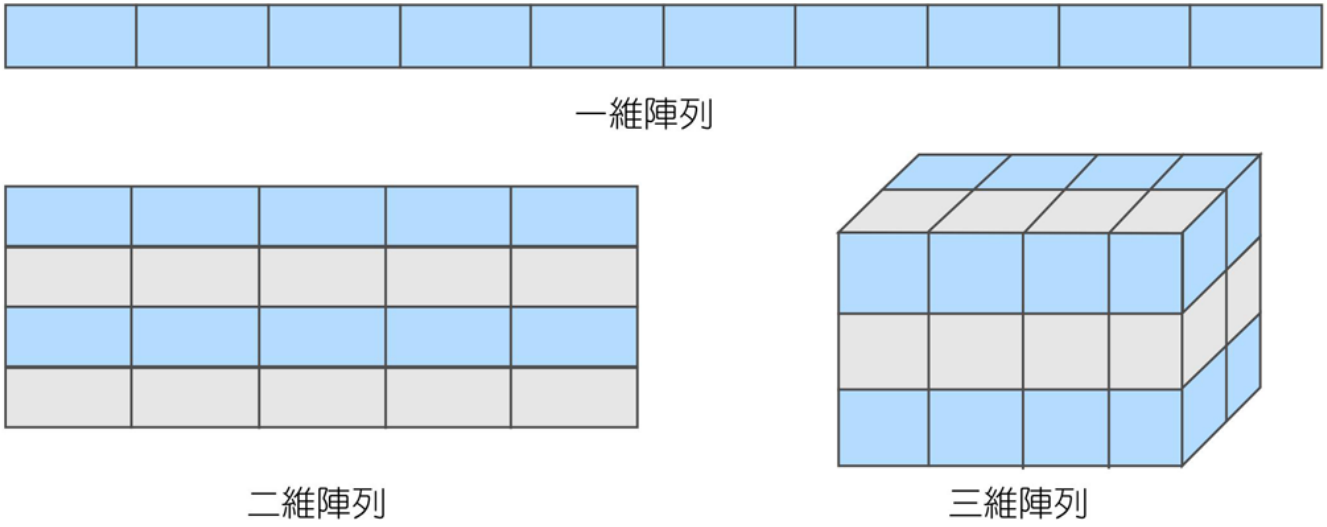
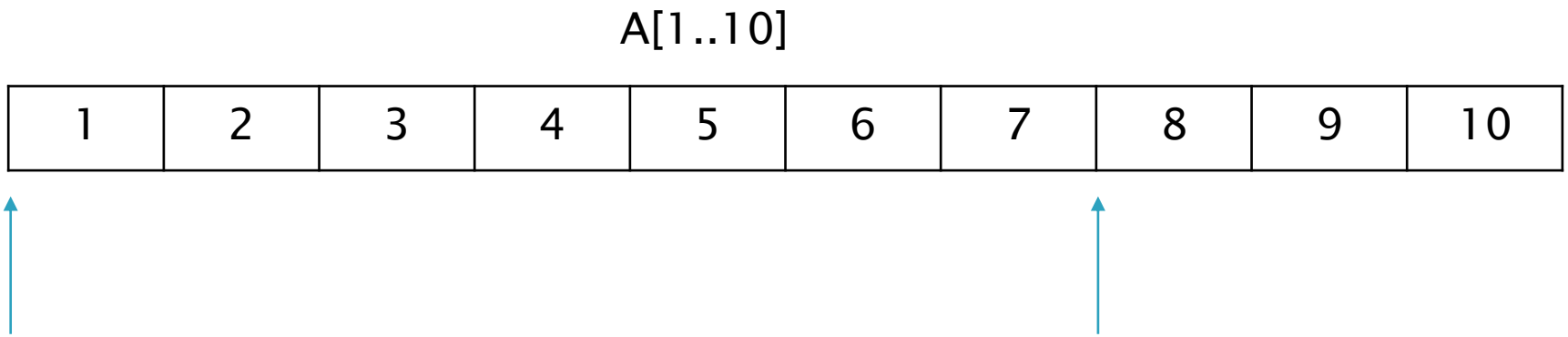


圖 8.1　一維、二維和三維陣列的樣子

基本資料結構

- 陣列能夠直接存取的原因在於每一格大小一樣，位置又是連續，所以只要知道起始位置，就能算出欲存取那一格的位置。



目前位置=起始位置+(目前指標-起始指標)*每個元素佔用空間

基本資料結構

隨堂練習

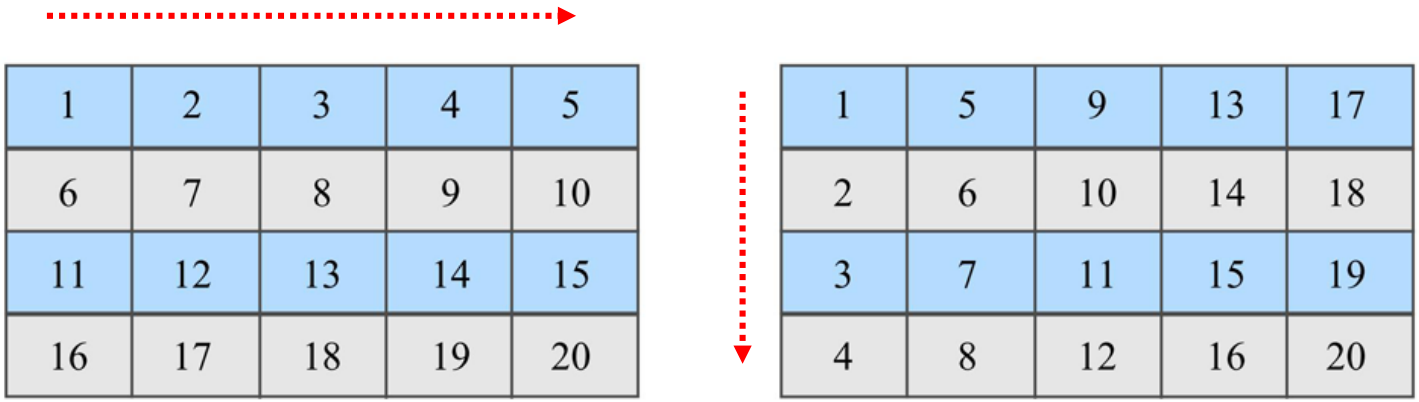
- 給定一個一維陣列 *a*[5..15]，若起始位置在 0100，每個元素佔四個位元組，請問 *a*[8] 的位置為何？

  解答：

  0100 + (8–5) * 4 = 0112

基本資料結構

- 二維陣列：通常有二種常用的定址方式，就是以列為主 (row major) 和以行為主 (column major)。



1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

以列為主的次序

1	5	9	13	17
2	6	10	14	18
3	7	11	15	19
4	8	12	16	20

以行為主的次序

圖 8.2　二種方式排列資料的次序

基本資料結構

- 二維陣列：以列為主 (row major)

A[1..4, 1..5]

1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	2,5
3,1	3,2	3,3	3,4	3,5
4,1	4,2	4,3	4,4	4,5

起始位置：0100
每個元素佔3個位元組

A[3,4]的位置為何？
Ans: 0100+((3−1)*5+4−1)*3=0139

A[r,c]位置=起始位置+((r−列起始指標)*行數+c−1)*每個元素佔用空間

基本資料結構

- 二維陣列：以行為主 (column major)

A[1..4, 1..5]

1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	2,5
3,1	3,2	3,3	3,4	3,5
4,1	4,2	4,3	4,4	4,5

起始位置：0100
每個元素佔3個位元組

A[3,4]的位置為何？
Ans: 0100+((4−1)*4+3−1)*3=0142

A[r,c]位置=起始位置+((c−行起始指標)*列數+r−1)*每個元素佔用空間

基本資料結構

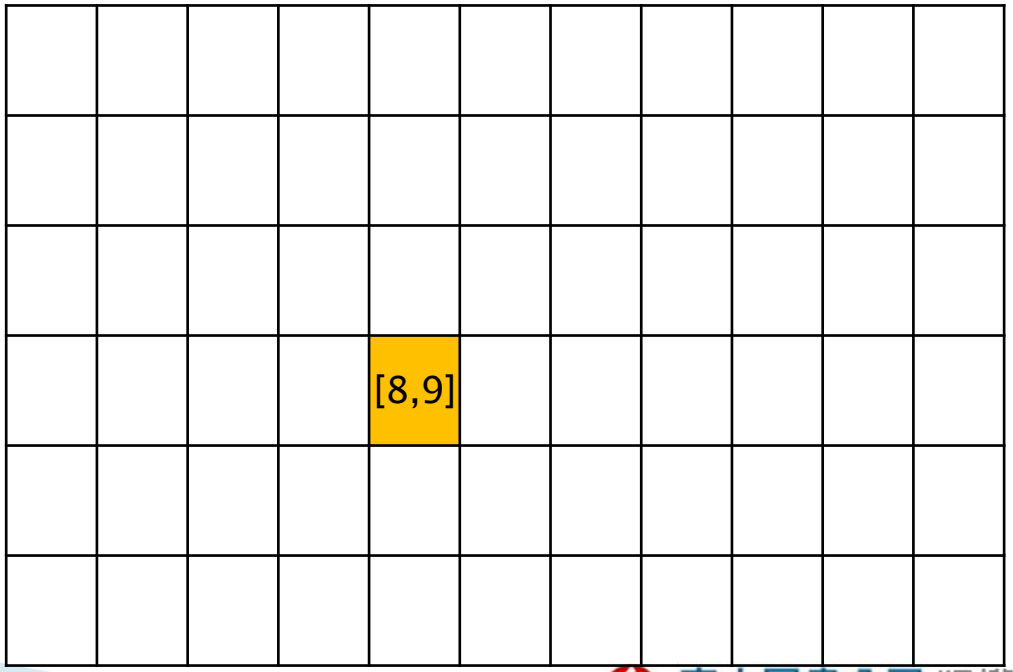
隨堂練習

- 給定一個二維陣列 *a*[5..10, 5..15]，請算出 *a*[8, 9] 在以列為主和以行為主時，它是此陣列的第幾個元素？

  解答：

  以列為主：(8–5)*11 + (9–5+1) = 38

  以行為主：(9–5)*6 + (8–5+1) = 28



基本資料結構

- 通常連結串列有一個特殊的指標指著此串列的頭 (head)，一般常用二種連結串列，稱為單連結串列 (singly linked list) 和雙連結串列 (doubly linked list)。

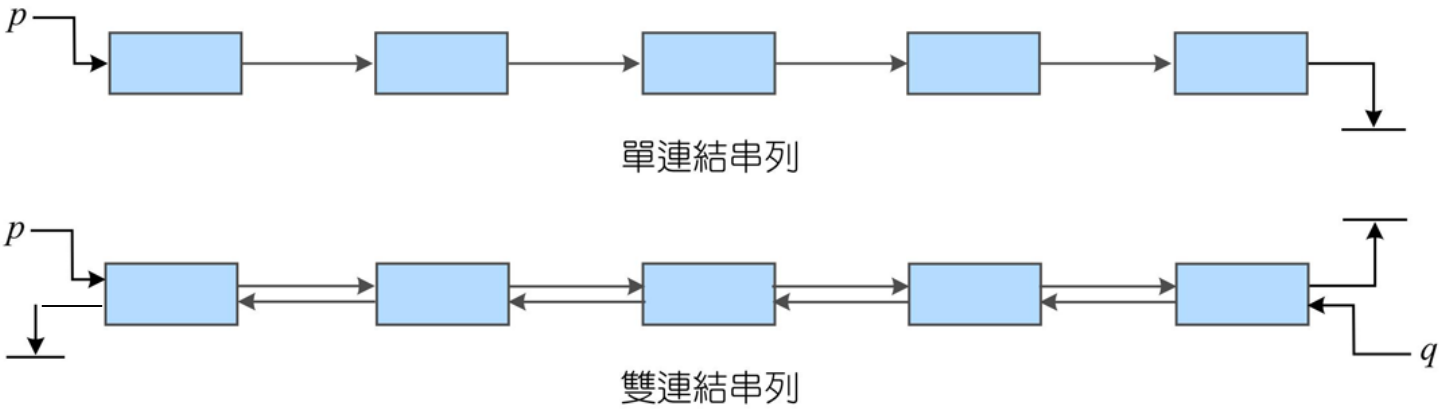


圖 8.3　單連結串列和雙連結串列示意圖

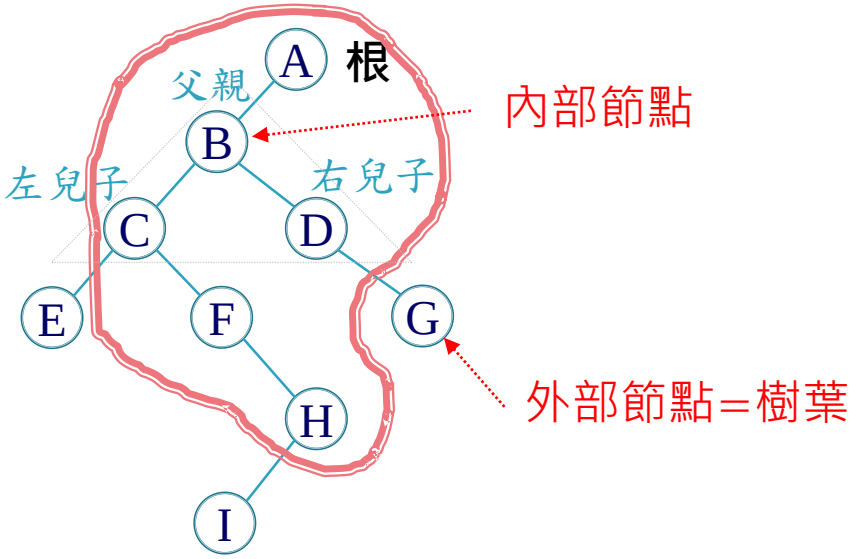
基本資料結構

- 一般稱陣列為**直接存取**資料結構
- 連結串列必須循著指標才能到達目的地，我們稱它為**循序存取**資料結構

雖然連結串列的存取較陣列麻煩，但它不用預留一塊連續的記憶體，是它的好處之一。

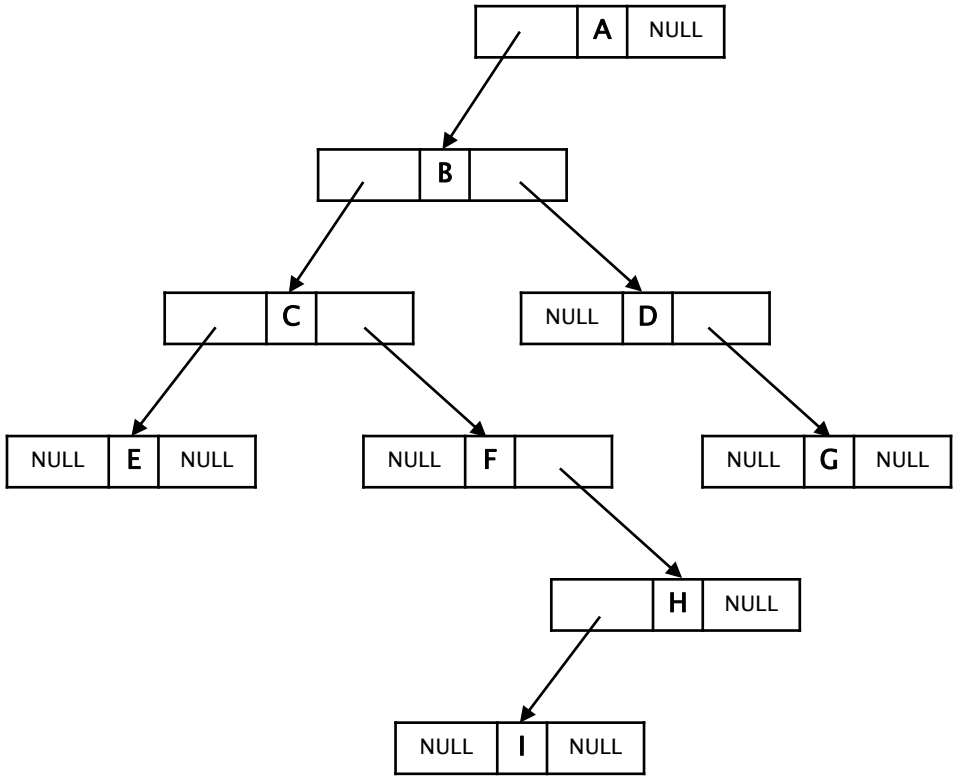
樹狀資料結構

- 樹有一個特殊的節點稱為根 (root)，類似連結串列的頭，每個節點都有多個指標指著它的兒子，為了方便起見，一般都有相同數目的指標，指著它們的兒子，這個數目稱為分支度 (degree)。二元樹 (binary tree)，也就是每個節點有二個指標，稱為左指標和右指標。沒有半個兒子的節點稱為樹葉 (leaf)，其他非樹葉的節點稱為內部節點。



樹狀資料結構

- 在一個樹狀結構裡，樹葉的指標都是指到空 (nil)，所以一個葉節點含越多指標，就越浪費空間。



樹狀資料結構

- 將根到某一個最遠樹葉節點的距離定義為該樹的高度 (high)，以圖 8.4 為例，它的高度為 2。另外每個節點 (含內部節點和葉節點) 都可以定義一個階層數 (level)，樹根的階層為 0，其它節點的階層被定義為它到根的距離。

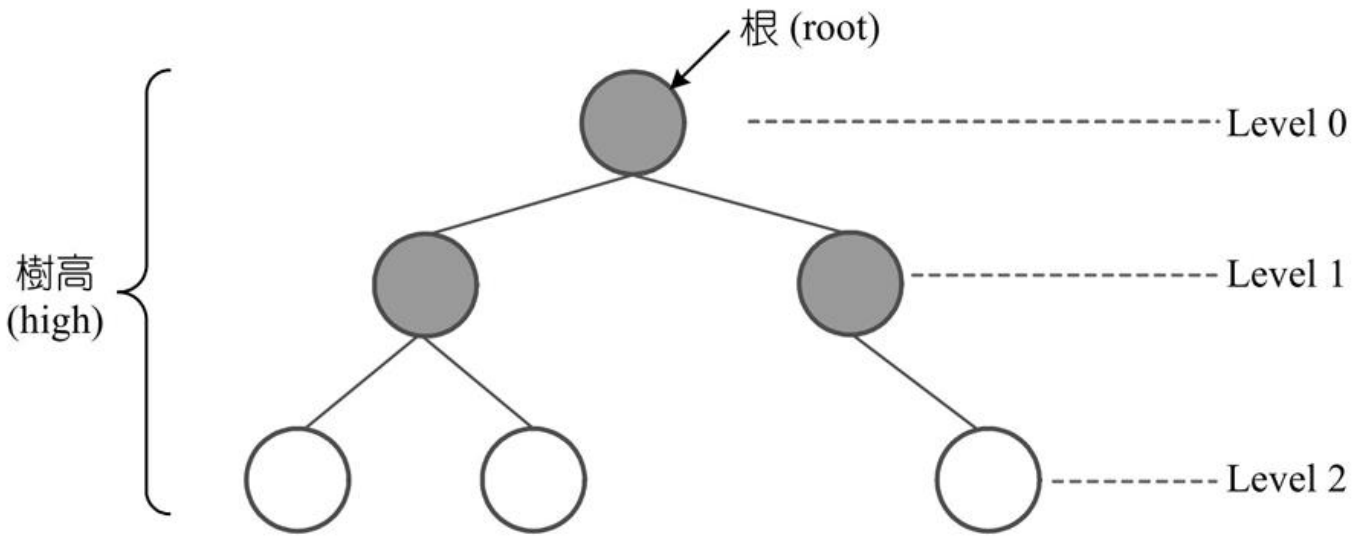
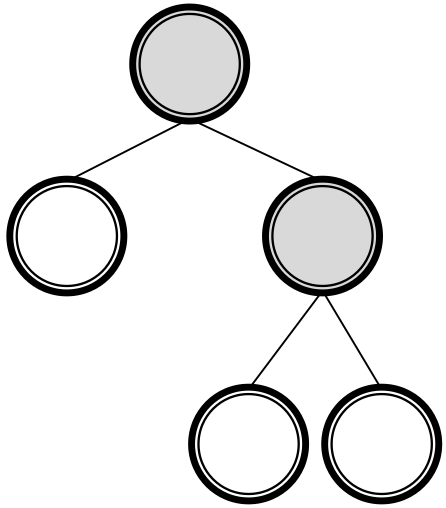


圖 8.4 一個二元樹的例子，實心節點為內部節點，空心節點為外部節點或葉節點。

樹狀資料結構

完滿二元樹 (full binary tree)

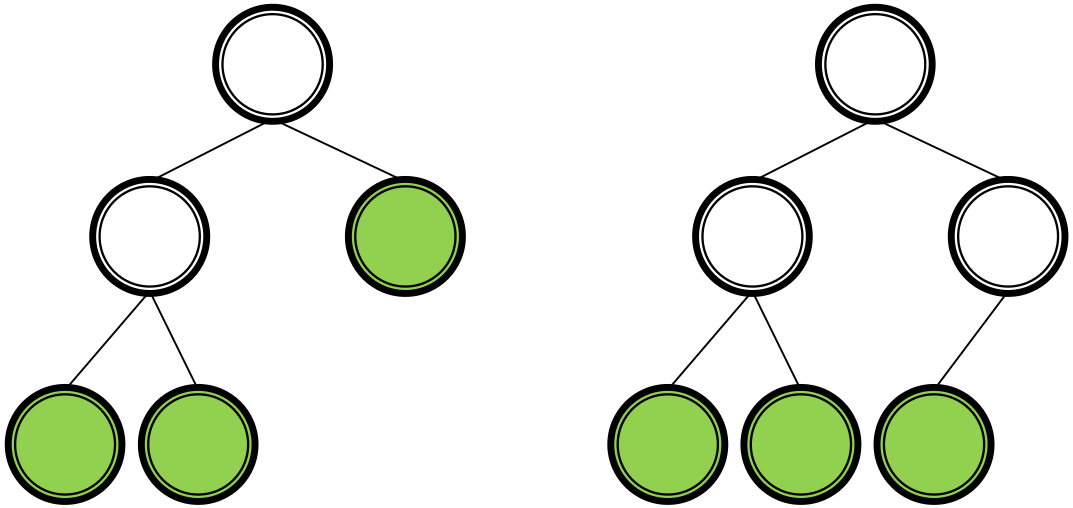
- 一個二元樹裡，如果每個內部節點都有二個兒子，則這種二元樹稱為完滿二元樹。



樹狀資料結構

完全二元樹 (complete binary tree)

- 一個二元樹的所有葉子都在最下二層，倒數第二層葉子的父親都有二個兒子，而且最後一層的葉子都集中在左邊，則此二元樹又稱為完全二元樹。

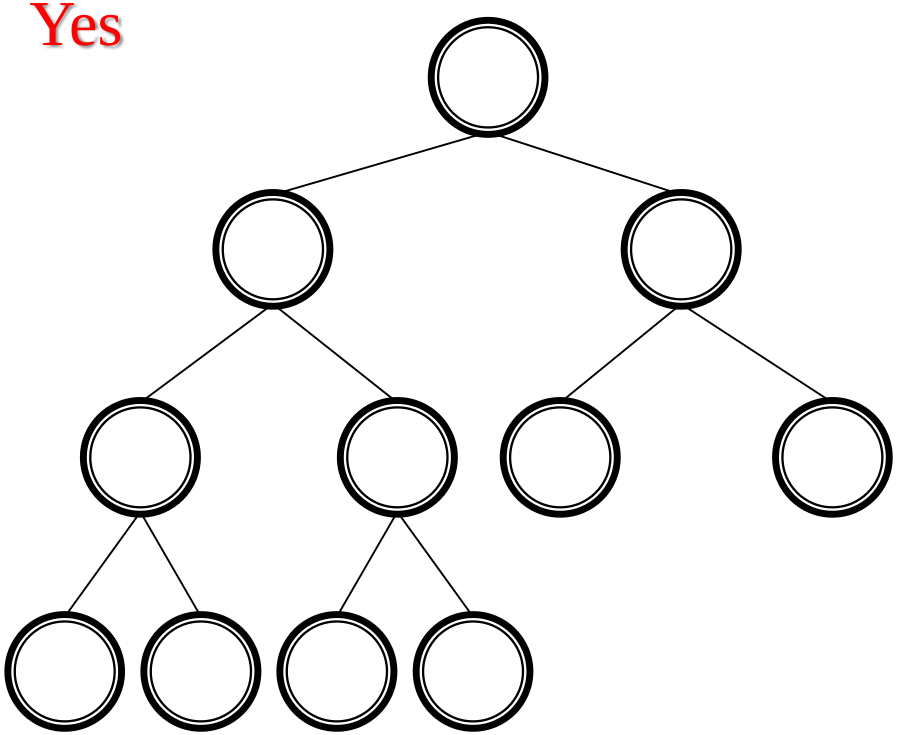


樹狀資料結構

隨堂練習

- 請問下面是否為一棵完全二元樹？

  解答：Yes

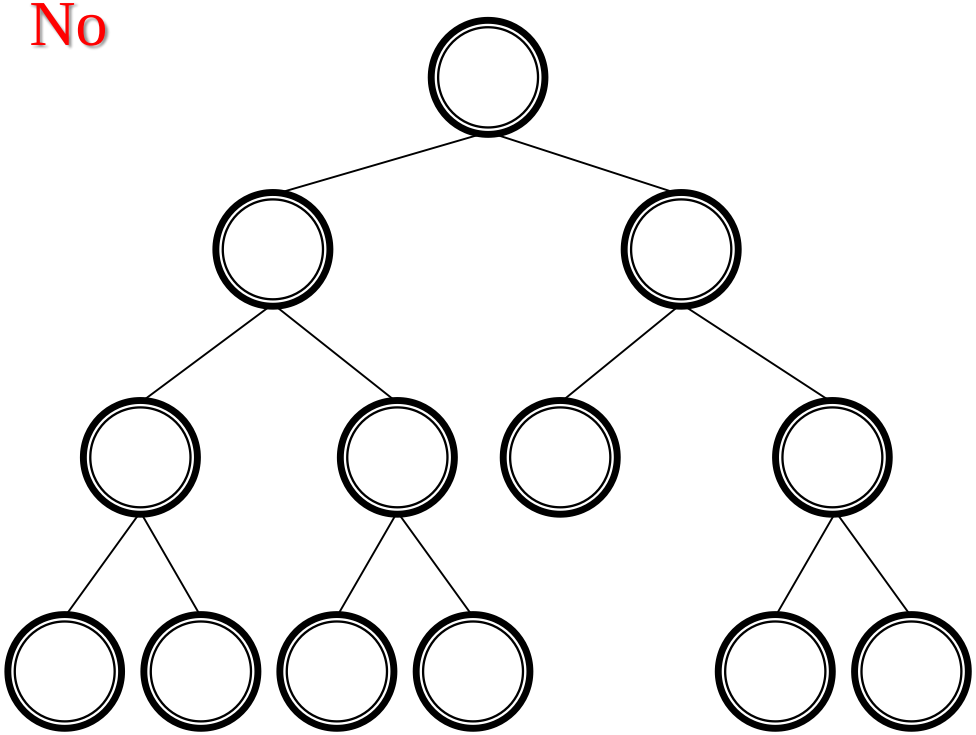


樹狀資料結構

隨堂練習

- 請問下面是否為一棵完全二元樹？

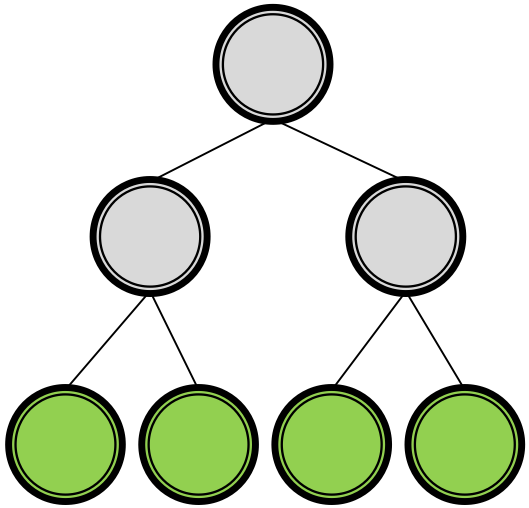
  解答：No



樹狀資料結構

完美二元樹 (perfect binary tree)

- 如果一個完全二元樹的葉子都在最後一層，且每個葉子的父親都有二個兒子，那麼這個完全二元樹又被稱為完美二元樹 。



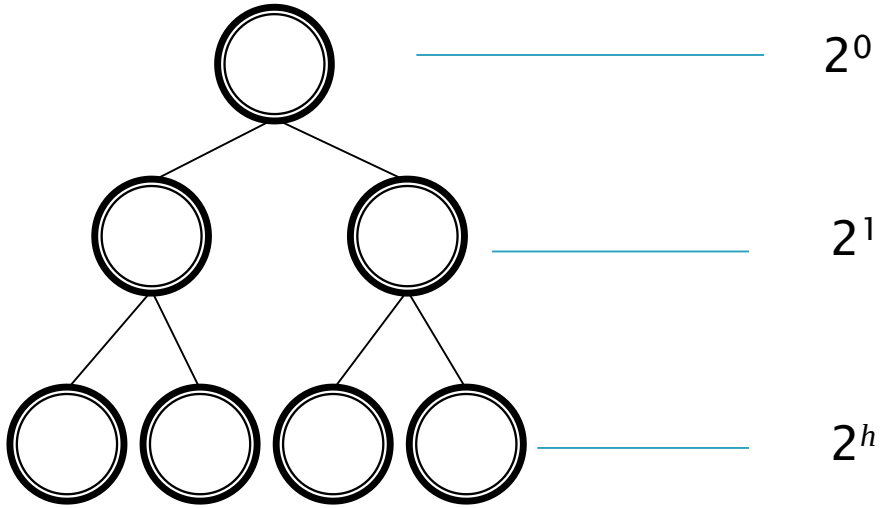
樹狀資料結構

隨堂練習

- 請問一個樹高為 h 的完美二元樹共有多少個節點？

  解答：

$$2^0 + 2^1 + \cdots 2^h = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$



樹狀資料結構

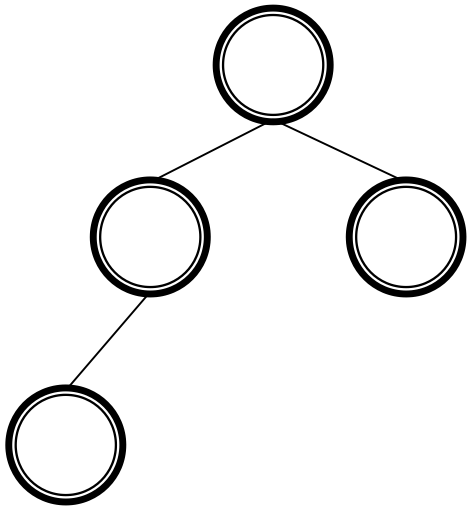
隨堂練習

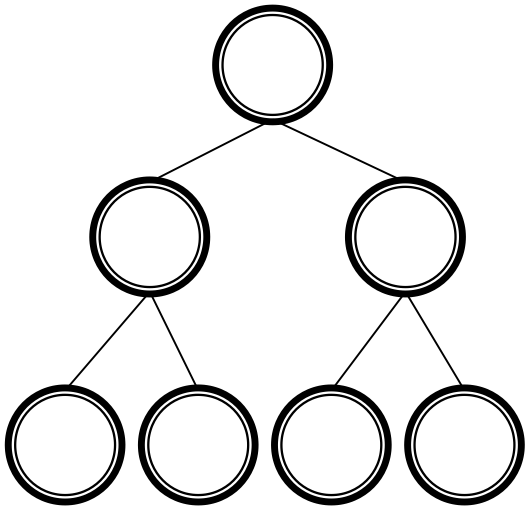
- 請問一個樹高為 h 的完全二元樹最少有多少個節點？最多有多少個節點？

  解答：

  最少 $2^0 + 2^1 + \cdots + 2^{h-1} + 1 = 2^h$

  最多 $2^0 + 2^1 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$





樹狀資料結構

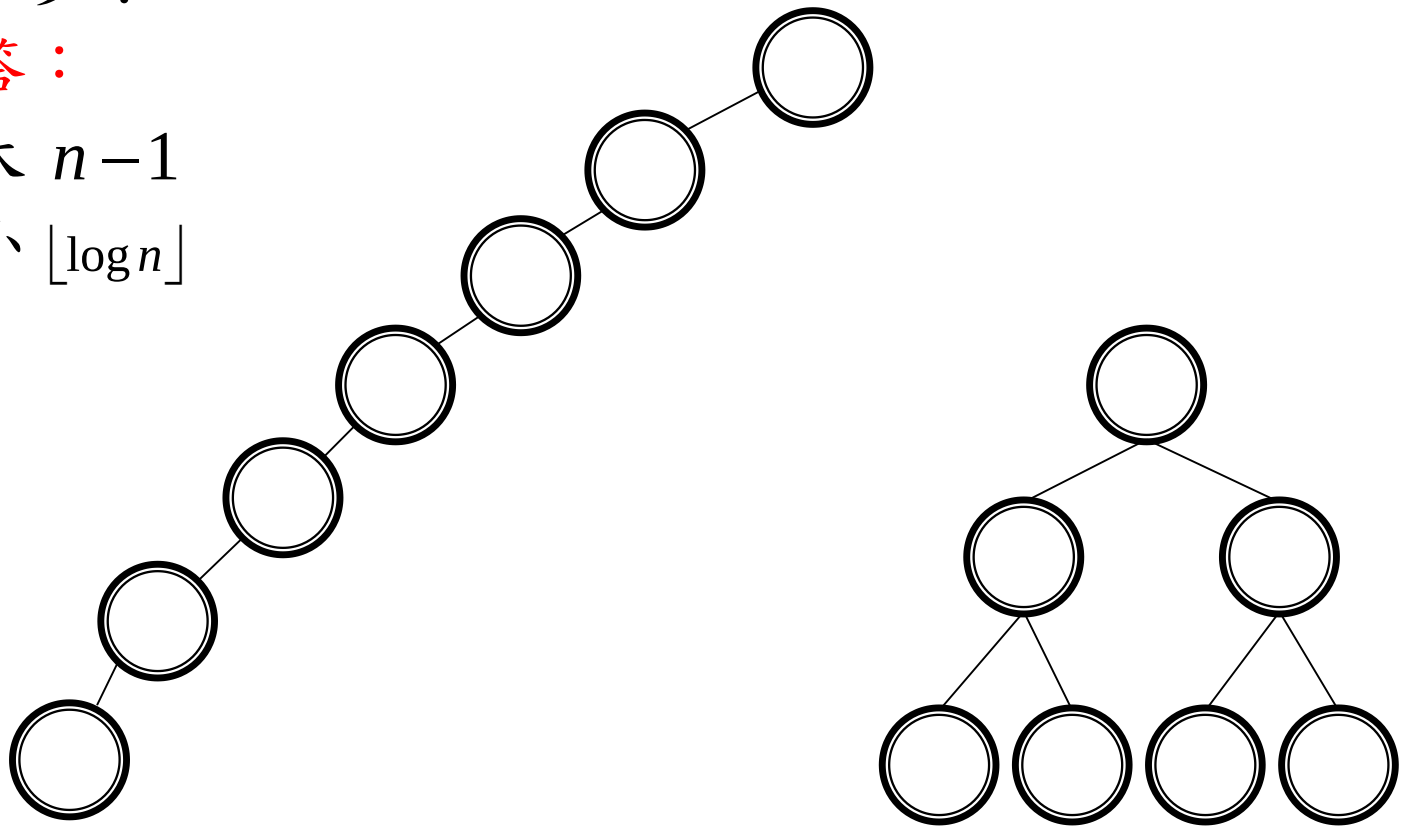
隨堂練習

- 請問一個含 n 個節點的二元樹，它的樹高最大是多少？最小是多少？

  解答：

  最大 $n-1$

  最小 $\lfloor \log n \rfloor$

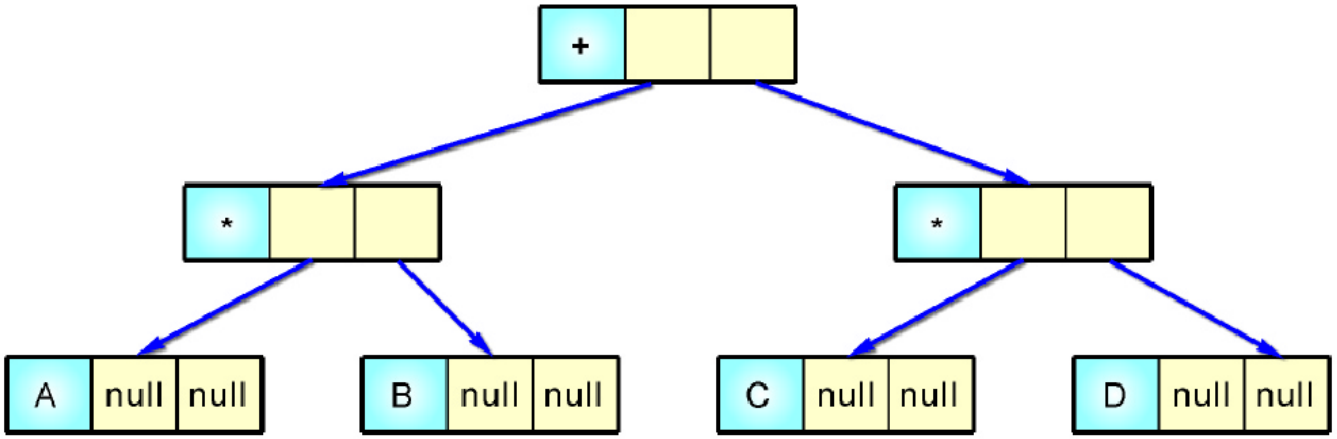


實做二元樹

- 定義樹中每一個節點的資料型態

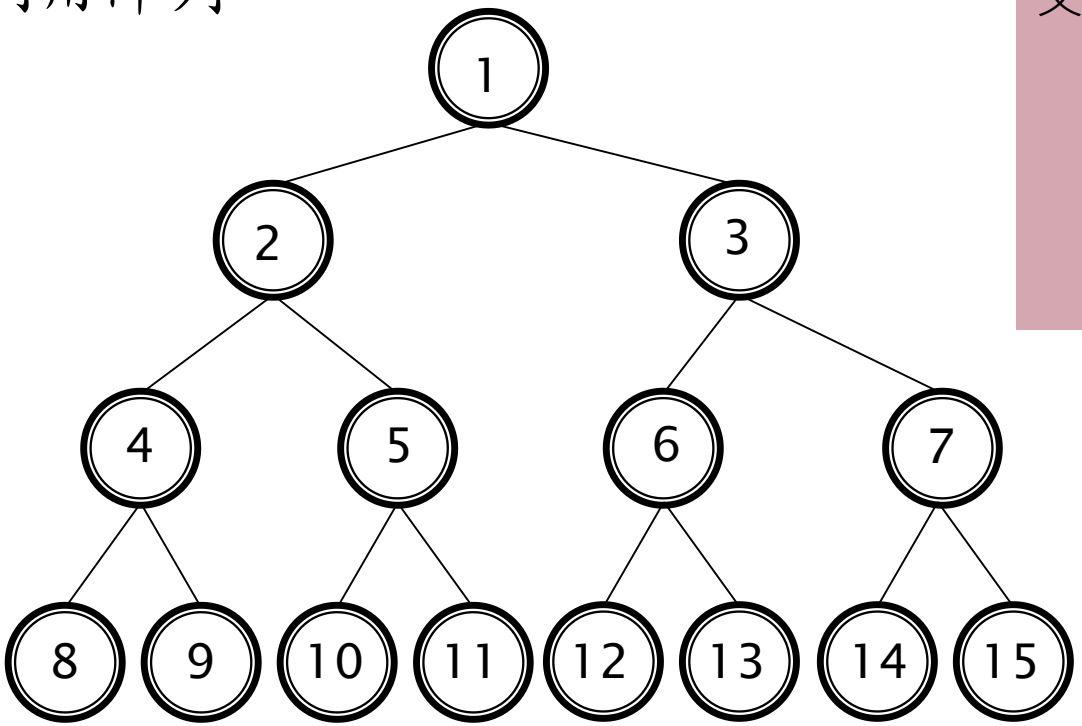
```
struct node
{
    char data;
    struct node *left;
    struct node *right;
};
```

- 將左子節點（或左子樹）以指標 “left”表示，而將右子節點（或右子樹）以指標 “right”表示，示意圖如下：



實做二元樹

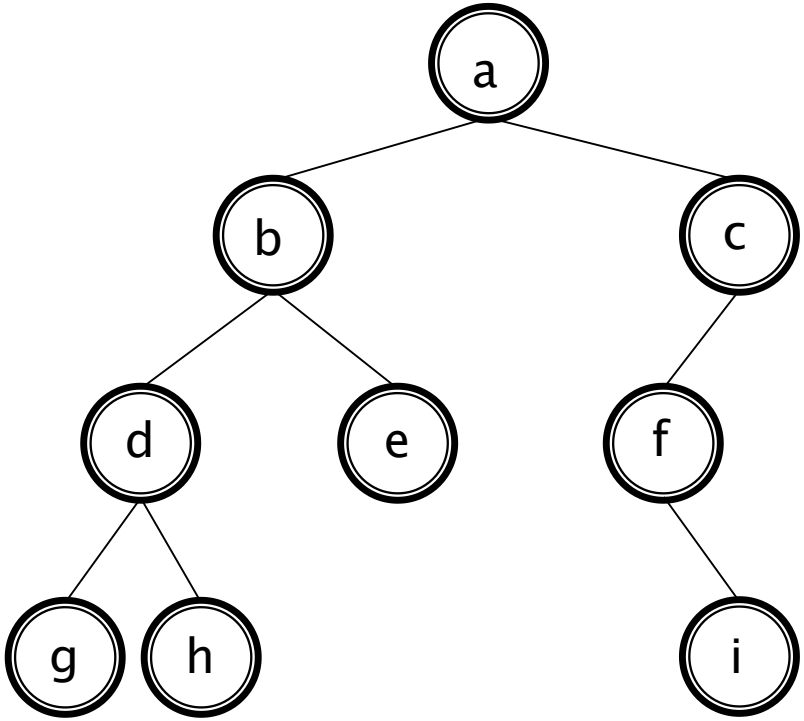
- 利用陣列



父與子的關係
- 父節點
- 子位置除2取整數部位
- 子節點
- 父位置乘2 (左子)
- 父位置乘2+1 (右子)

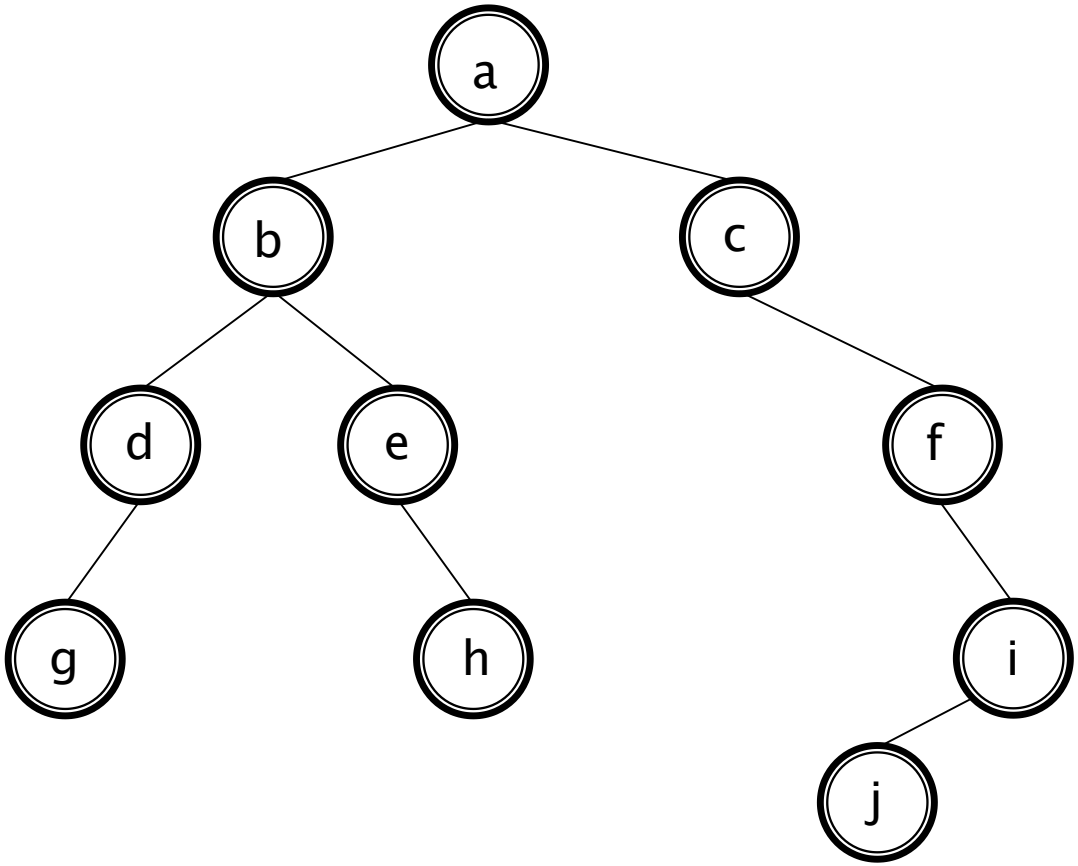
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

範例



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	c	d	e	f		g	h				i		

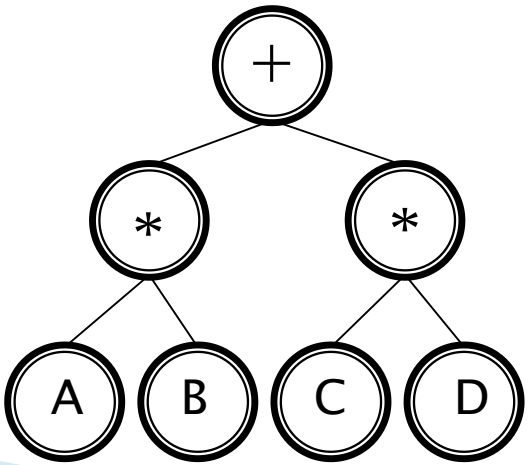
實作練習



a	b	c	d	e		f	g			h				i													j	

二元樹的三種探訪法

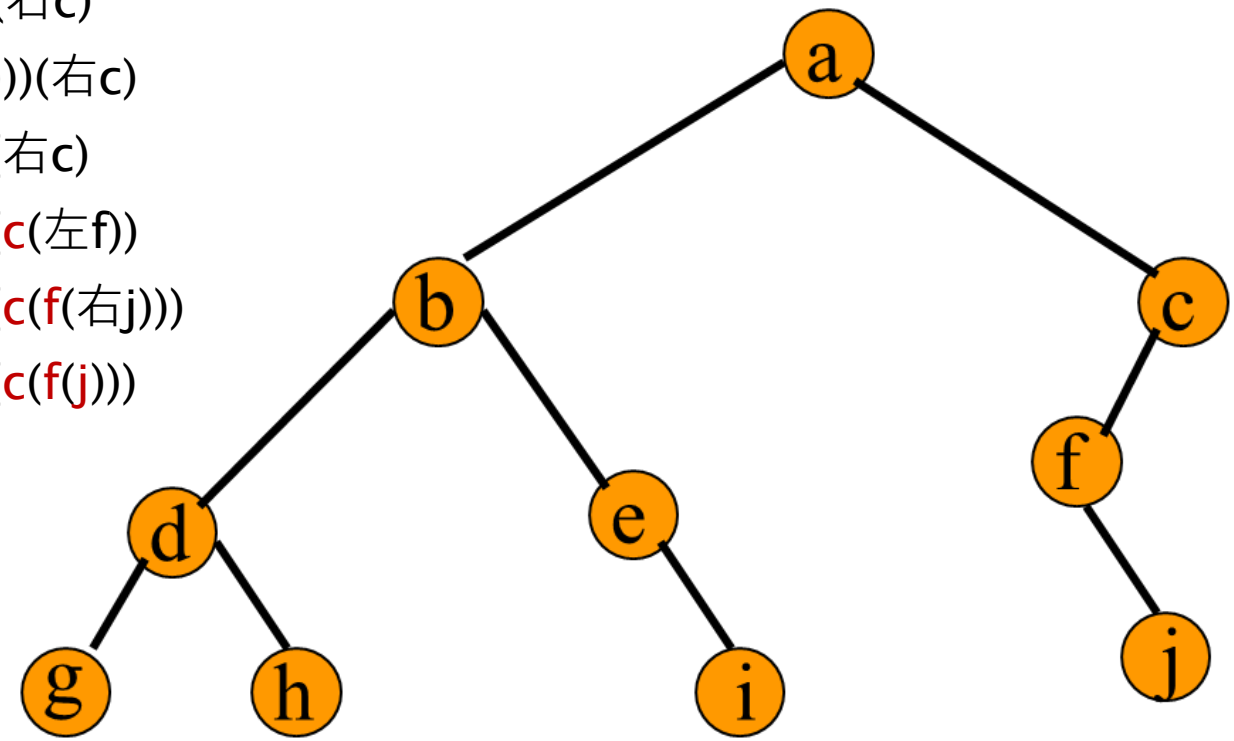
- 前序法（Preorder）：
  - 先探訪父節點、再探訪左子節點、最後探訪右子節點
  - 對應到運算式的前序法，如：+*AB*CD
- 中序法（Inorder）：
  - 先探訪左子節點、再探訪父節點、最後探訪右子節點
  - 對應到運算式的中序法，如：A*B+C*D
- 後序法（Postorder）：
  - 先探訪左子節點、再探訪右子節點、最後探訪父節點
  - 對應到運算式的後序法，如：AB*CD*+



前序法範例

a(左b)(右c)
a(b(左d)(右e))(右c)
a(b(d(左g)(右h))(右e))(右c)
a(b(d(g)(h))(右e))(右c)
a(b(d(g)(h))(e(右i)))(右c)
a(b(d(g)(h))(e(i)))(右c)
a(b(d(g)(h))(e(i)))(c(左f))
a(b(d(g)(h))(e(i)))(c(f(右j)))
a(b(d(g)(h))(e(i)))(c(f(j)))

a b d g h e i c f j



中序法範例

(左b)a(右c)

((左d)b(右e))a(右c)

(((左g)d(右h))b(右e))a(右c)

(((g)d(h))b(右e))a(右c)

(((g)d(h))b(e(右i)))a(右c)

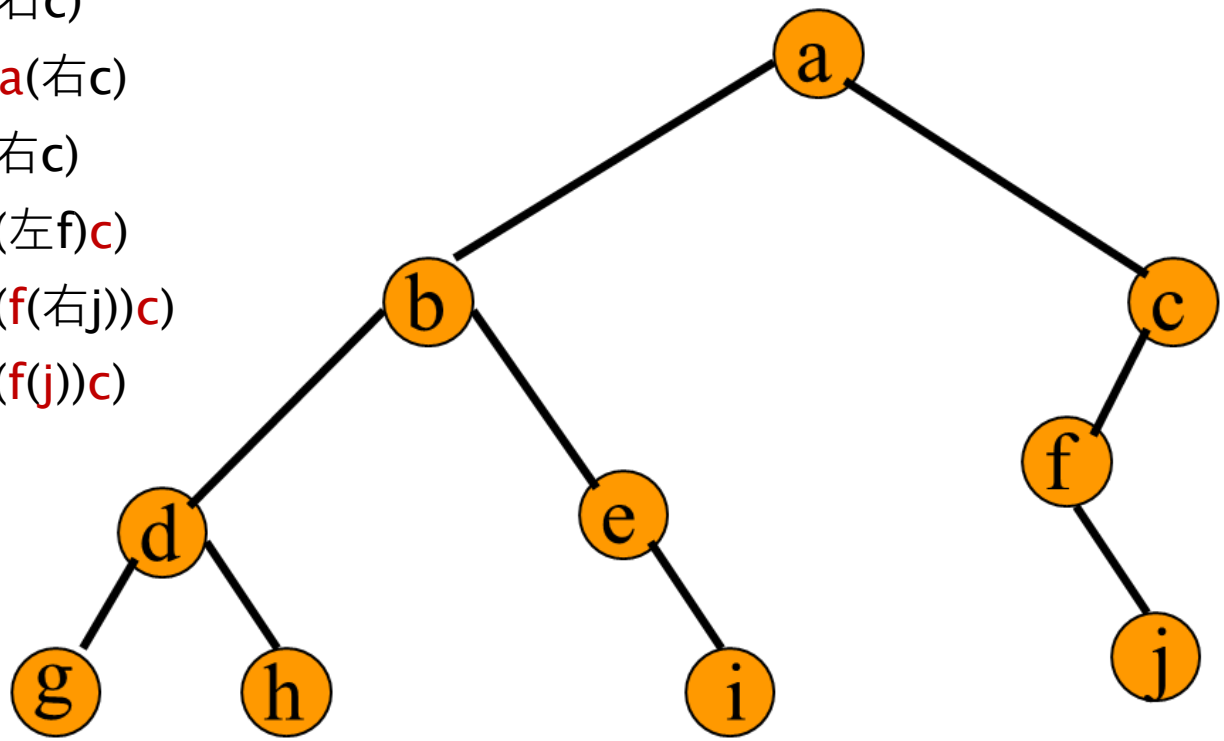
(((g)d(h))b(e(i)))a(右c)

(((g)d(h))b(e(i)))a((左f)c)

(((g)d(h))b(e(i)))a((f(右j))c)

(((g)d(h))b(e(i)))a((f(j))c)

g d h b e i a f j c



後序法範例

(左b)(右c)a

((左d)(右e)b)(右c)a

(((左g)(右h)d)(右e)b)(右c)a

(((g)(h)d)(右e)b)(右c)a

(((g)(h)d)((右i)e)b)(右c)a

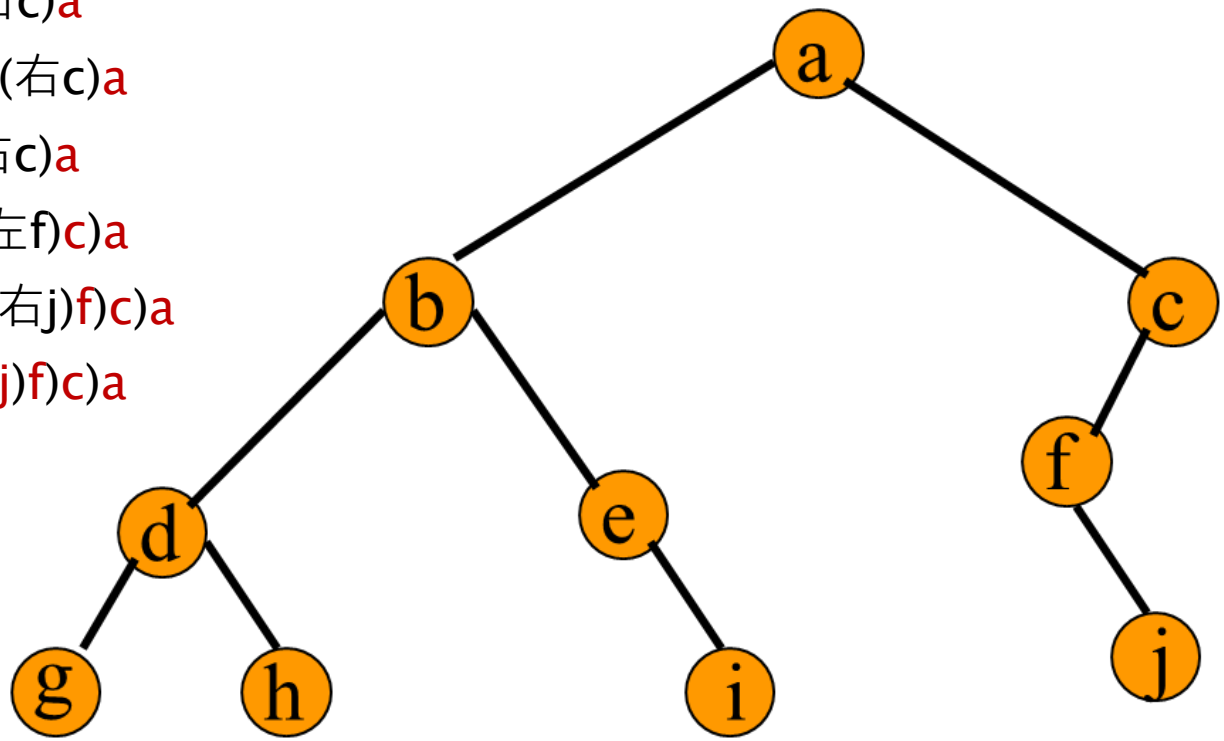
(((g)(h)d)((i)e)b)(右c)a

(((g)(h)d)((i)e)b)((左f)c)a

(((g)(h)d)((i)e)b)(((右j)f)c)a

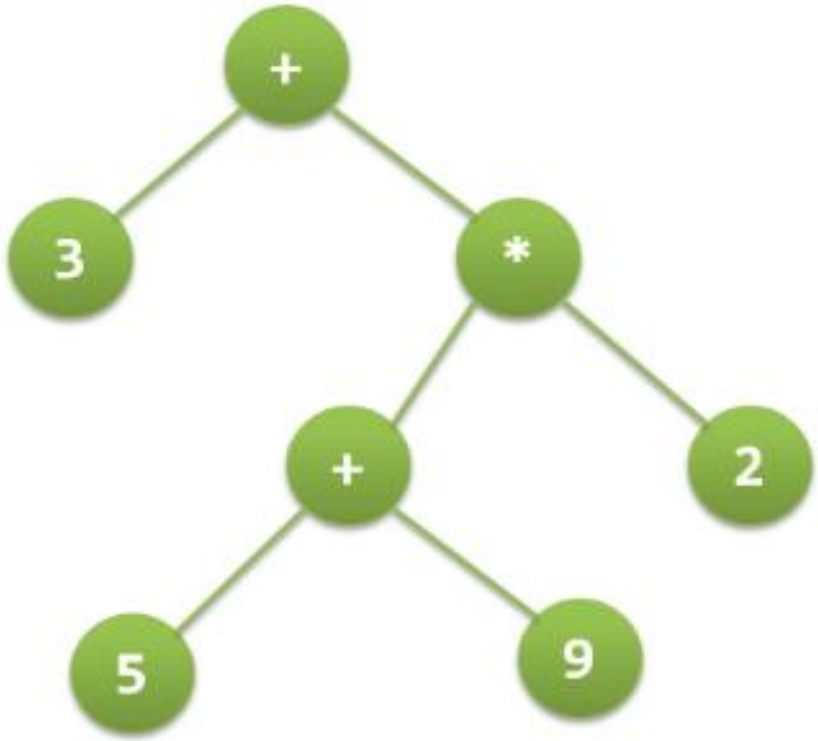
(((g)(h)d)((i)e)b)(((j)f)c)a

g h d i e b j f c a



運算樹(Expression tree)

3+(5+9)*2



前序prefix
+3*+592

後序postfix
359+2*+

樹狀資料結構

- 二元搜尋樹(Binary Search Tree)：假設每一個節點都儲存一個數值，如果對每一個節點都滿足：「它的值大於等於左兒子的值，但小於右兒子的值」，那麼這棵樹就被稱作「二元搜尋樹」。

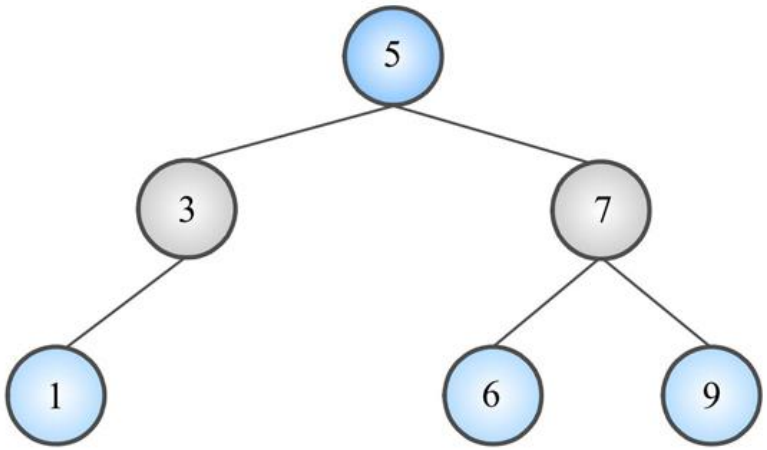
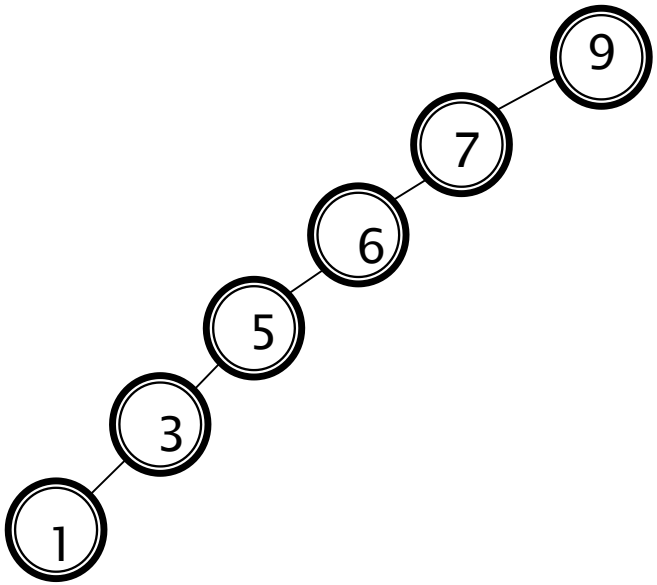
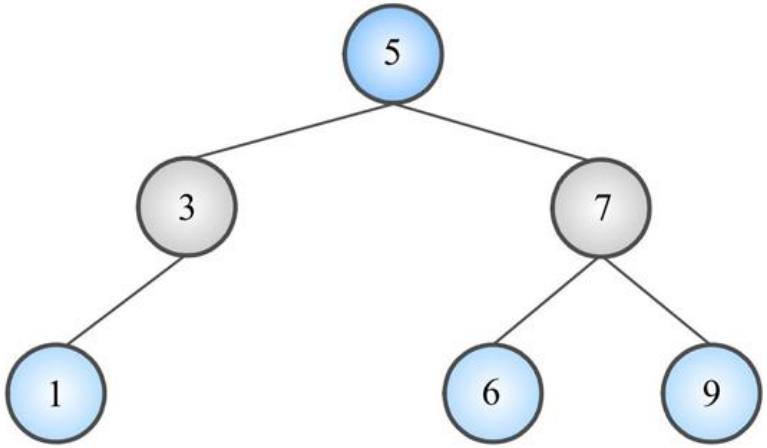


圖 8.6　一棵二元搜尋樹



樹狀資料結構

- 假設我們有 n 個數值，如果存在陣列或連結串列裡，那麼搜尋一個數值最糟的情況下，可能要拜訪這 n 個資料。
- 若儲存在二元搜尋樹裡，要比較的資料數會跟它的樹高有關，而非所有資料數。搜尋的方法是先比較根節點的資料，若是值等於根節點的值，表示搜尋成功；若是小於根節點的值，就往它的左子樹走，反之就往它的右子樹走。



搜尋值	比較次數
1	3
2	3
3	2
4	2
5	1
6	3
7	2

樹狀資料結構

- 堆積 (heap) 是一個使用陣列來實做的樹狀資料結構。它必須是一個完全二元樹，樹根就是陣列裡的第一個元素。然後假設我們在任一個節點 i 上，如果它不是葉節點，那麼它的左兒子就是 $2i$，而右兒子就是 $2i+1$ (如果有的話)；如果節點 i 不是根節點，那麼它的父親就是 $\left\lfloor \frac{i}{2} \right\rfloor$ (也就是 i 除以 2 取整數)。

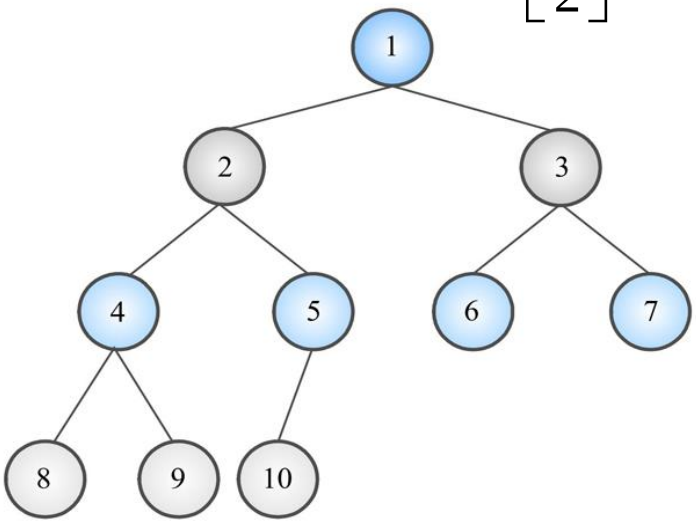
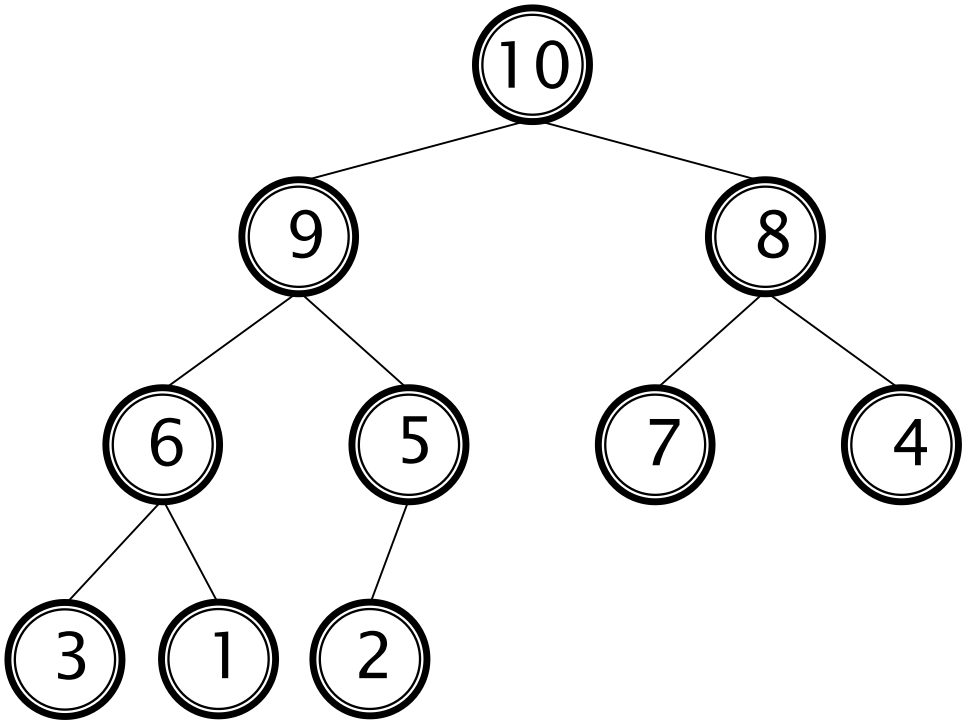


圖 8.7 考慮 $i=3$，則它的父親就是 $\left\lfloor \frac{3}{2} \right\rfloor = 1$，它的左兒子就是 2*3=6，而它的右兒子就是 2*3+1=7。

樹狀資料結構

堆積 (heap)

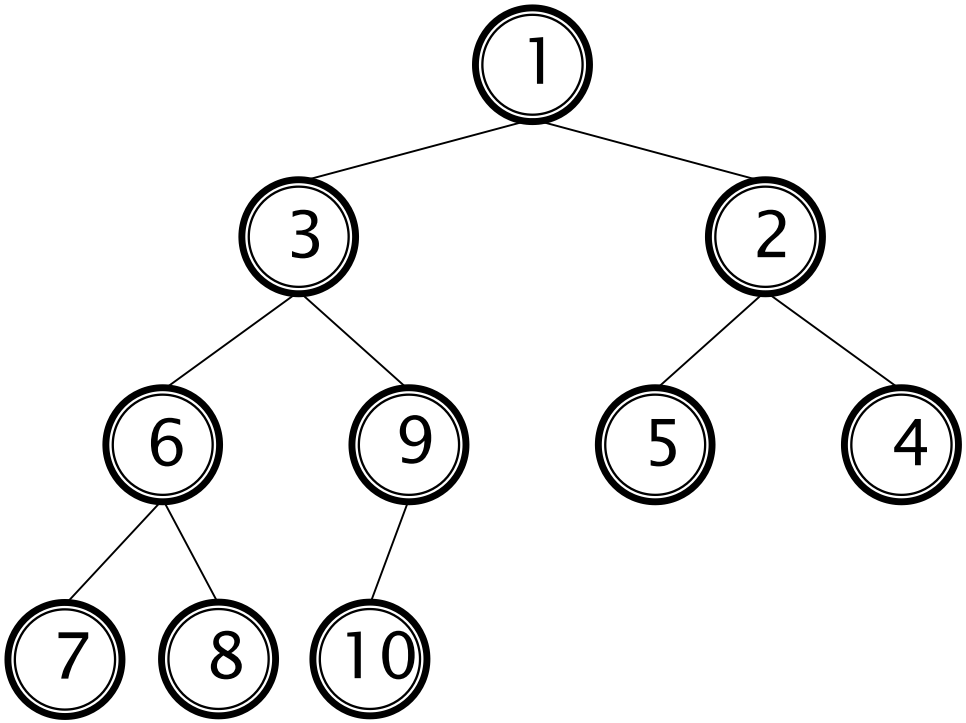
- 一個最大堆積 (max heap) 是一個完全二元樹並且滿足對任一個節點，它的值要比兒子們都大。



樹狀資料結構

堆積 (heap)

- 一個最小堆積 (min heap) 是一個完全二元樹並且滿足對任一個節點，它的值要比兒子們都小。



樹狀資料結構

- 堆積是一個很好的資料結構，通常用來儲存優先權，如最大或最小優先權。
- 堆積的建立有二種方式，分別是由上而下法和由下而上法。

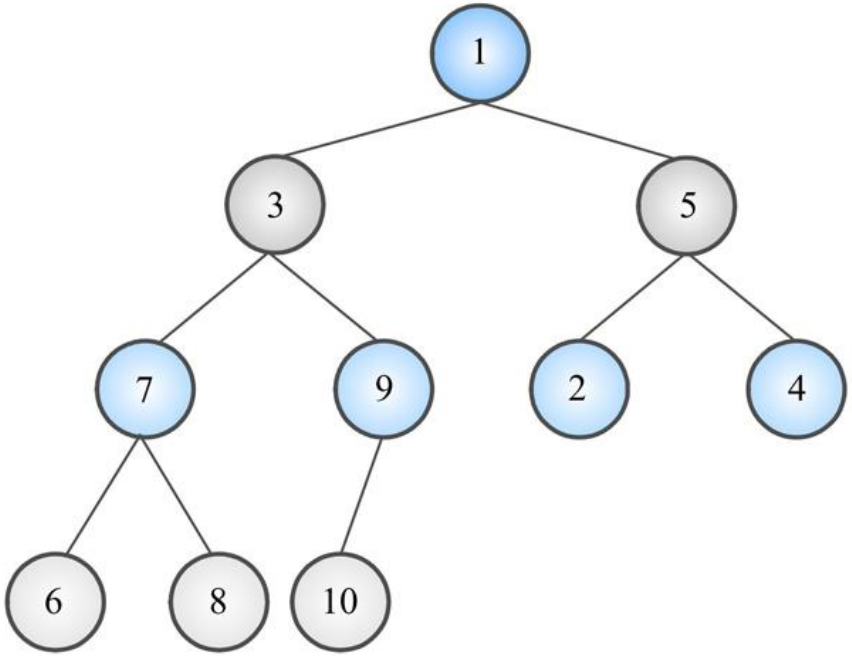
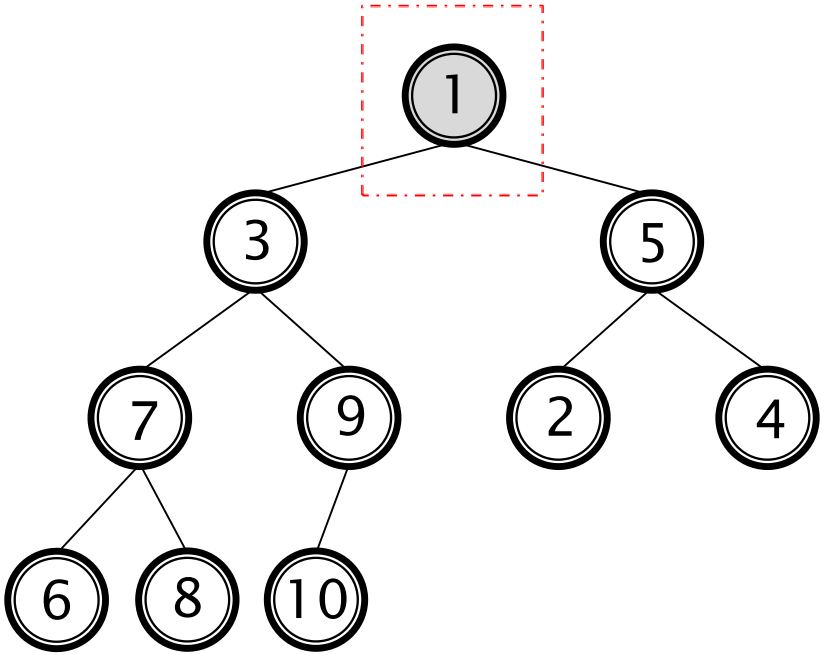


圖 8.8 樹狀結構

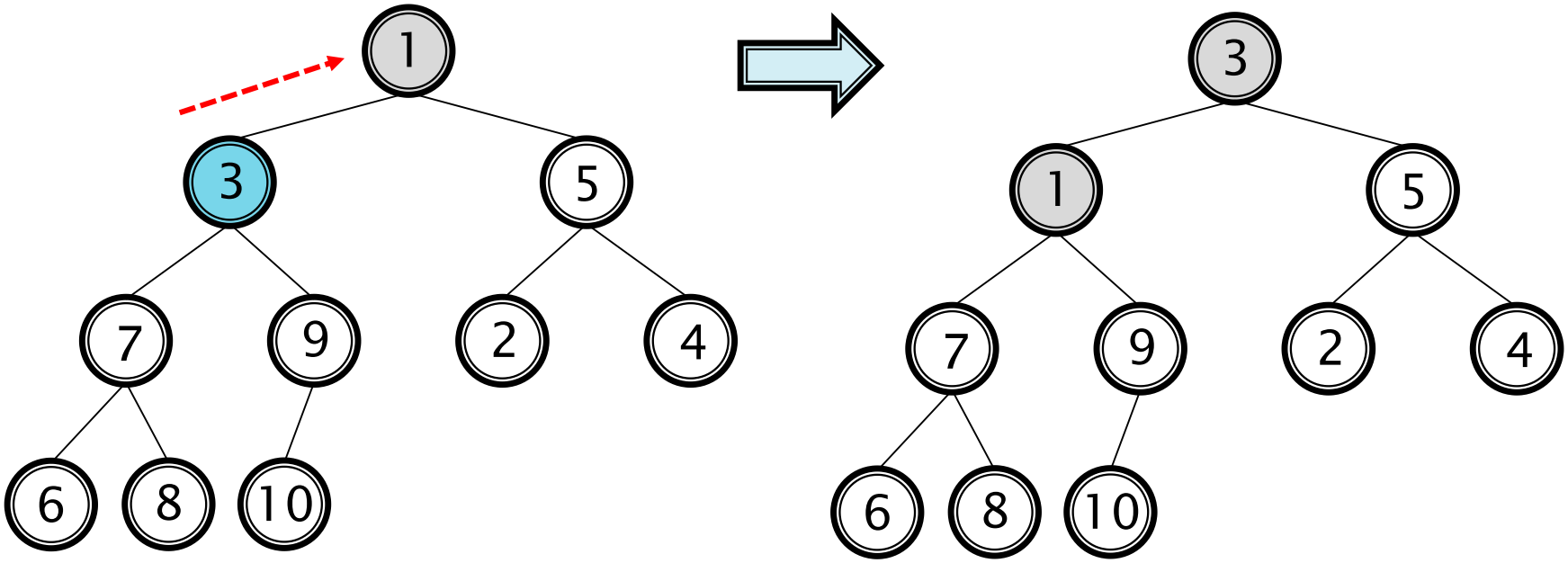
樹狀資料結構

- 由上而下建立最大堆積



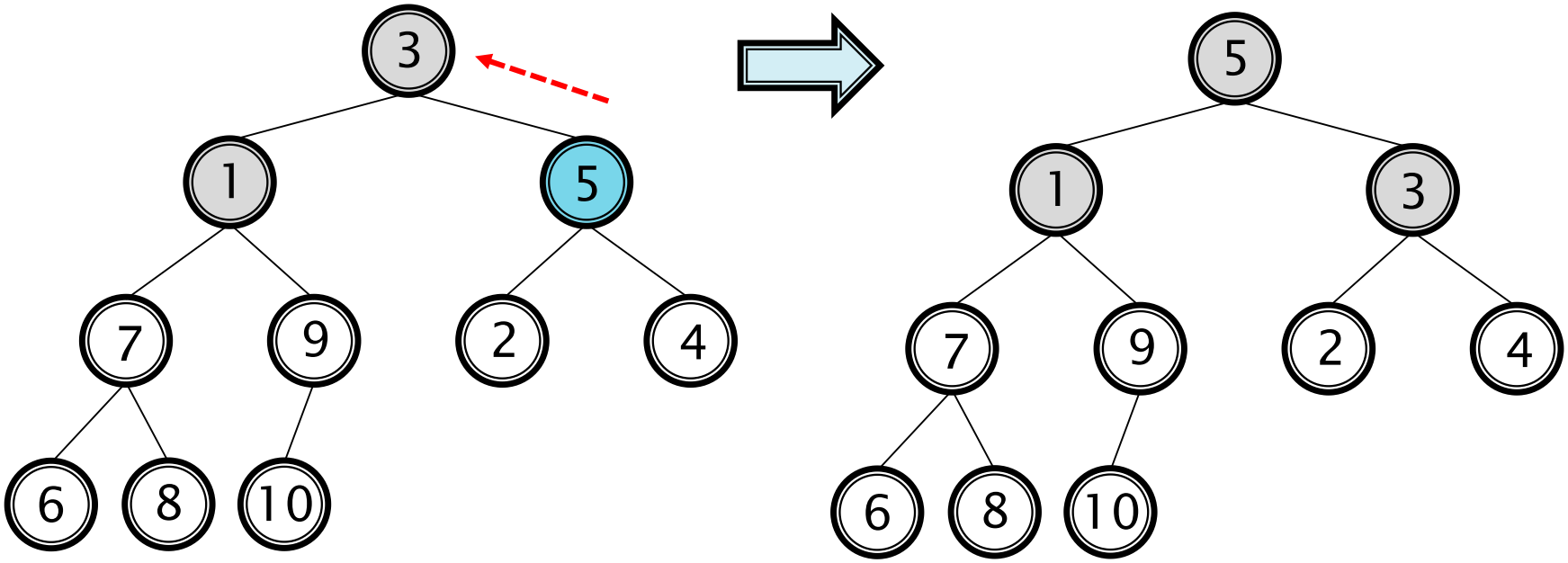
樹狀資料結構

- 由上而下建立最大堆積



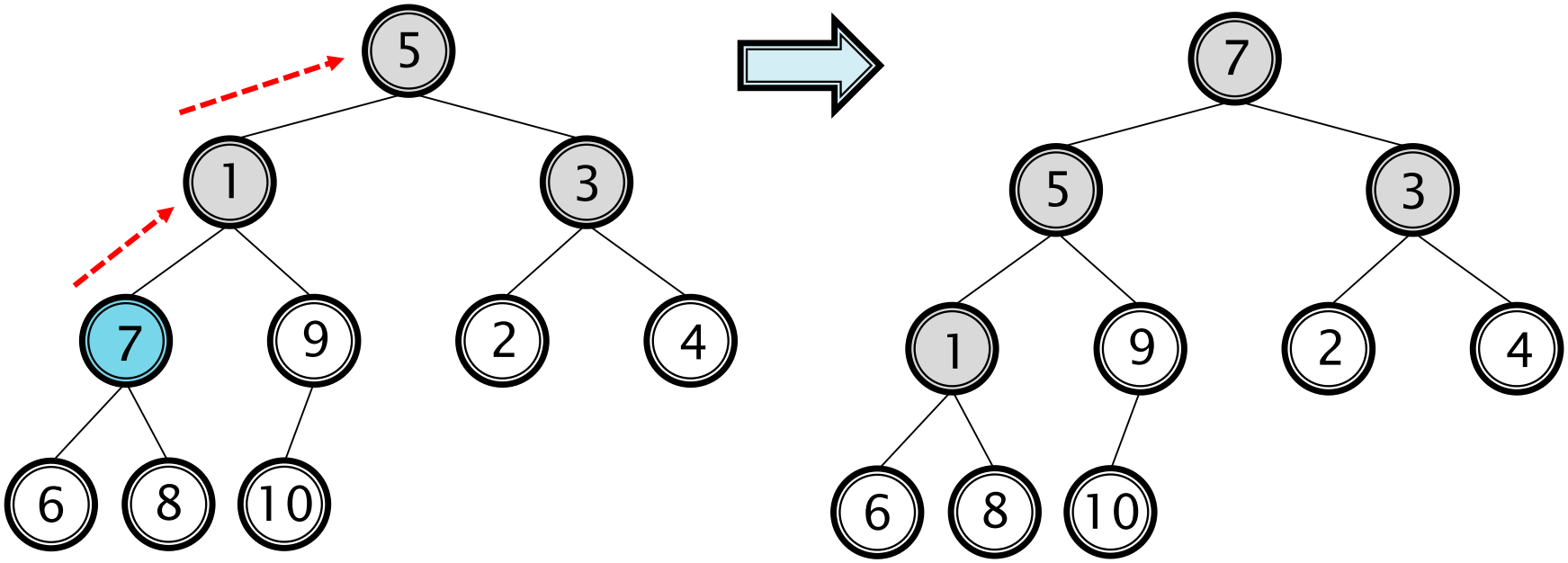
樹狀資料結構

- 由上而下建立最大堆積



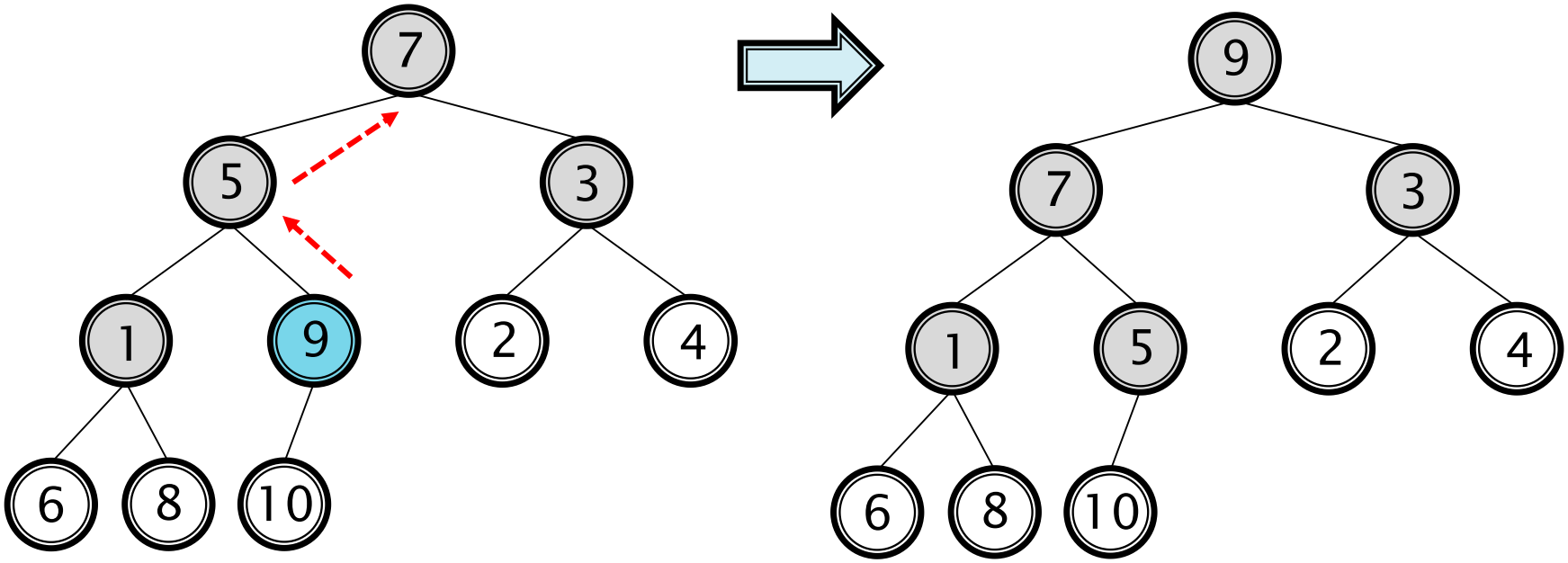
樹狀資料結構

- 由上而下建立最大堆積



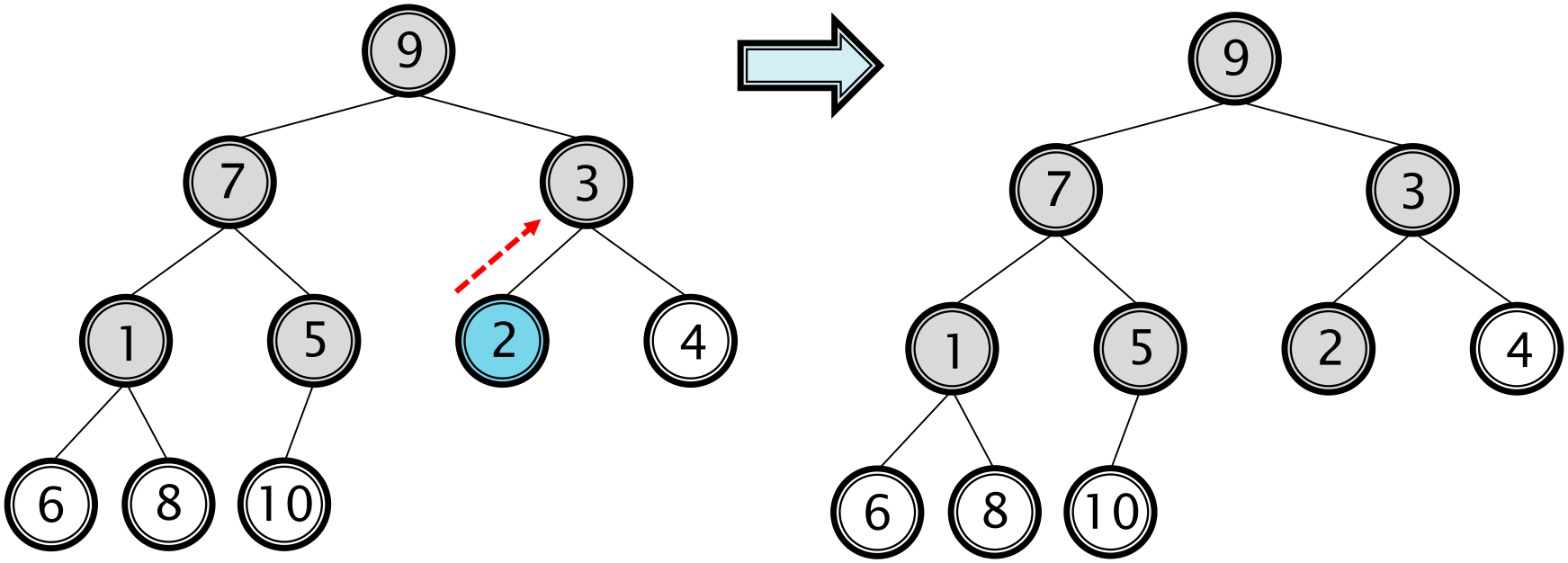
樹狀資料結構

- 由上而下建立最大堆積



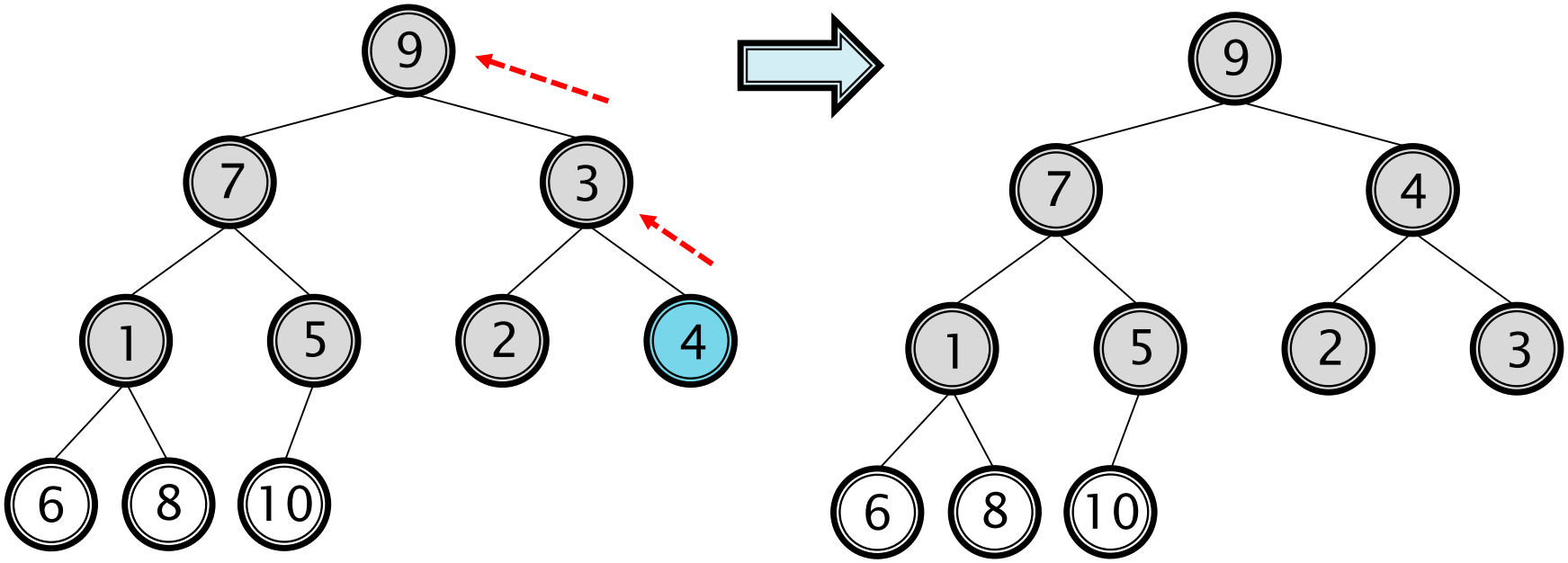
樹狀資料結構

- 由上而下建立最大堆積



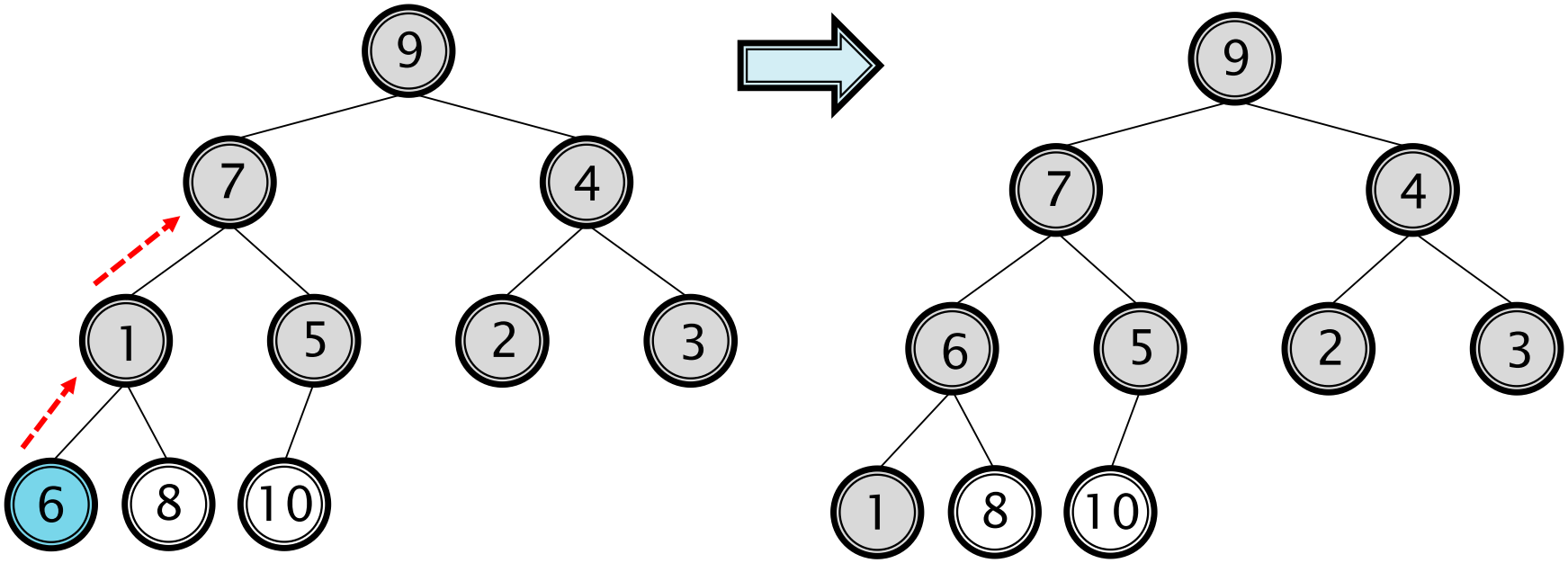
樹狀資料結構

- 由上而下建立最大堆積



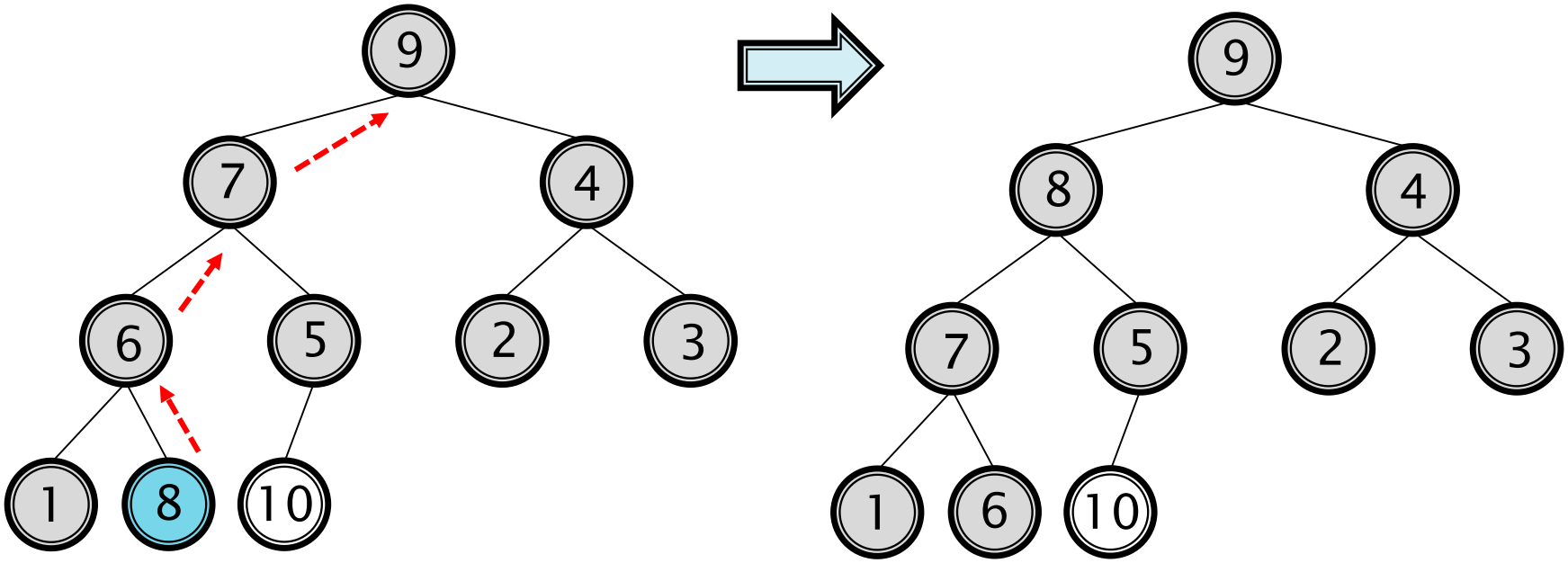
樹狀資料結構

- 由上而下建立最大堆積



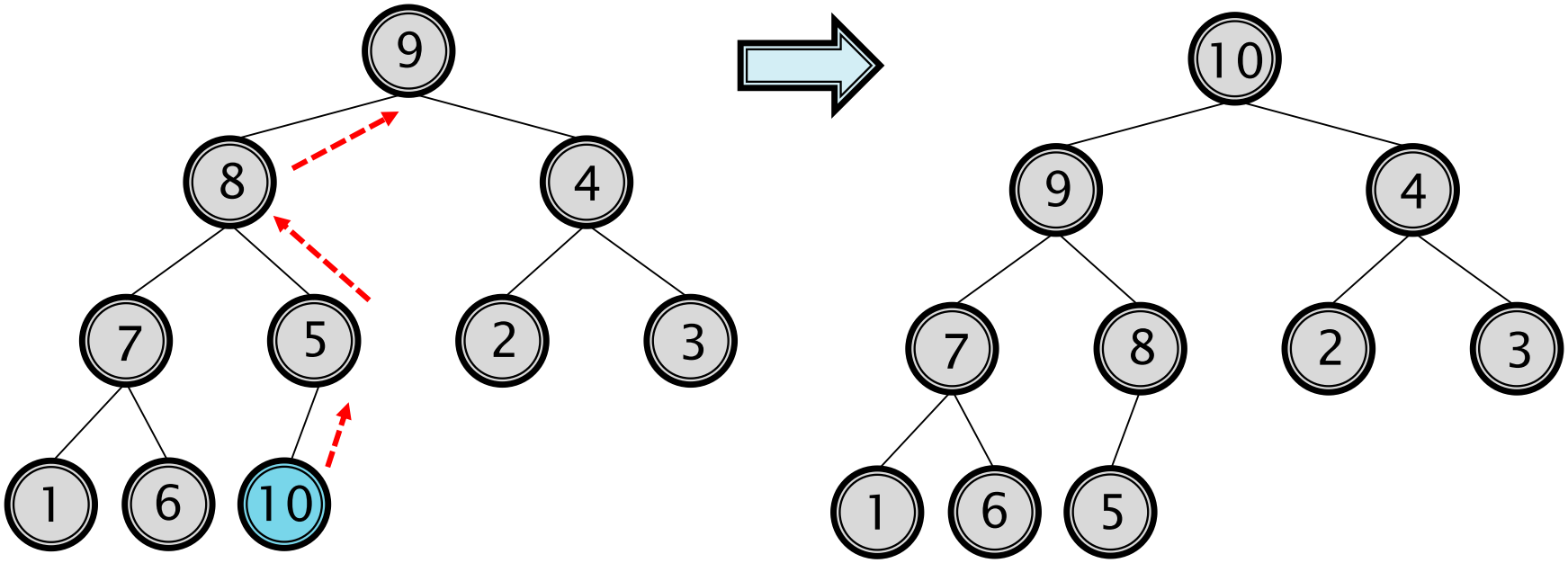
樹狀資料結構

- 由上而下建立最大堆積



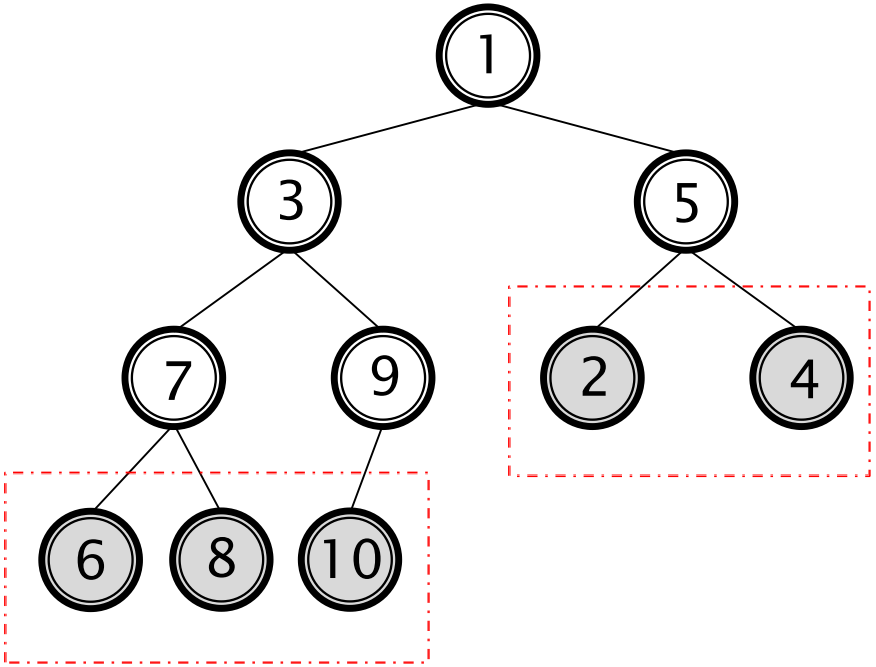
樹狀資料結構

- 由上而下建立最大堆積



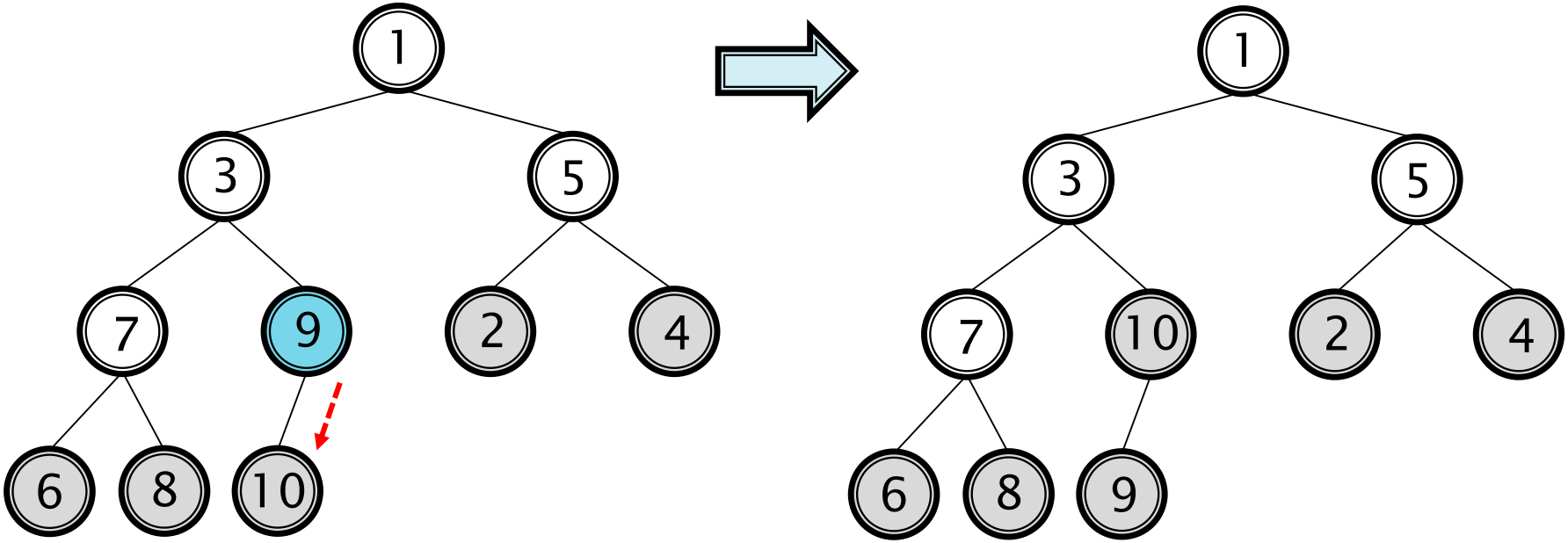
樹狀資料結構

- 由下而上建立最大堆積



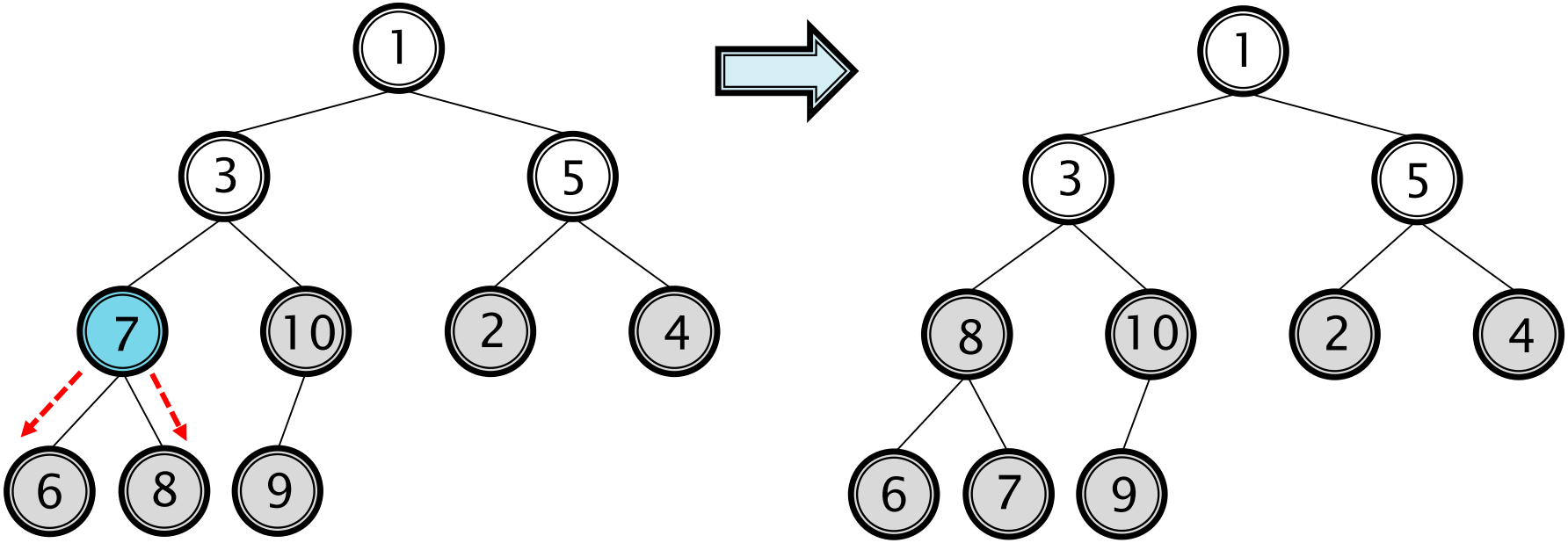
樹狀資料結構

- 由下而上建立最大堆積



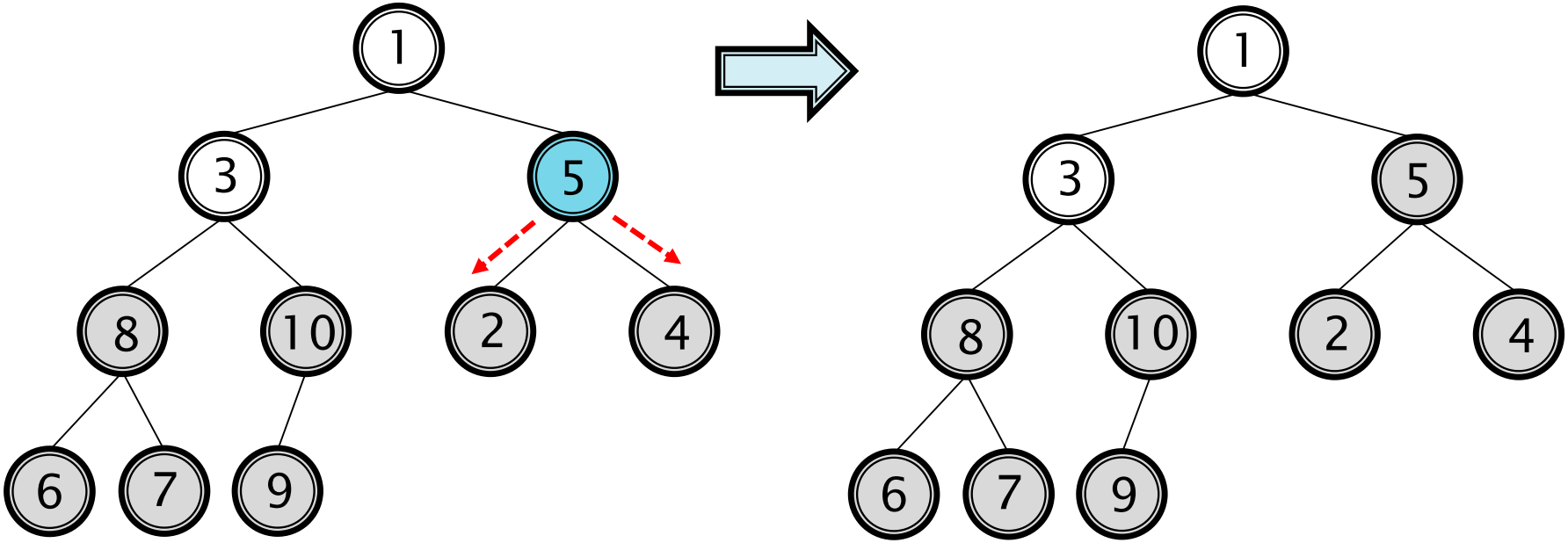
樹狀資料結構

- 由下而上建立最大堆積



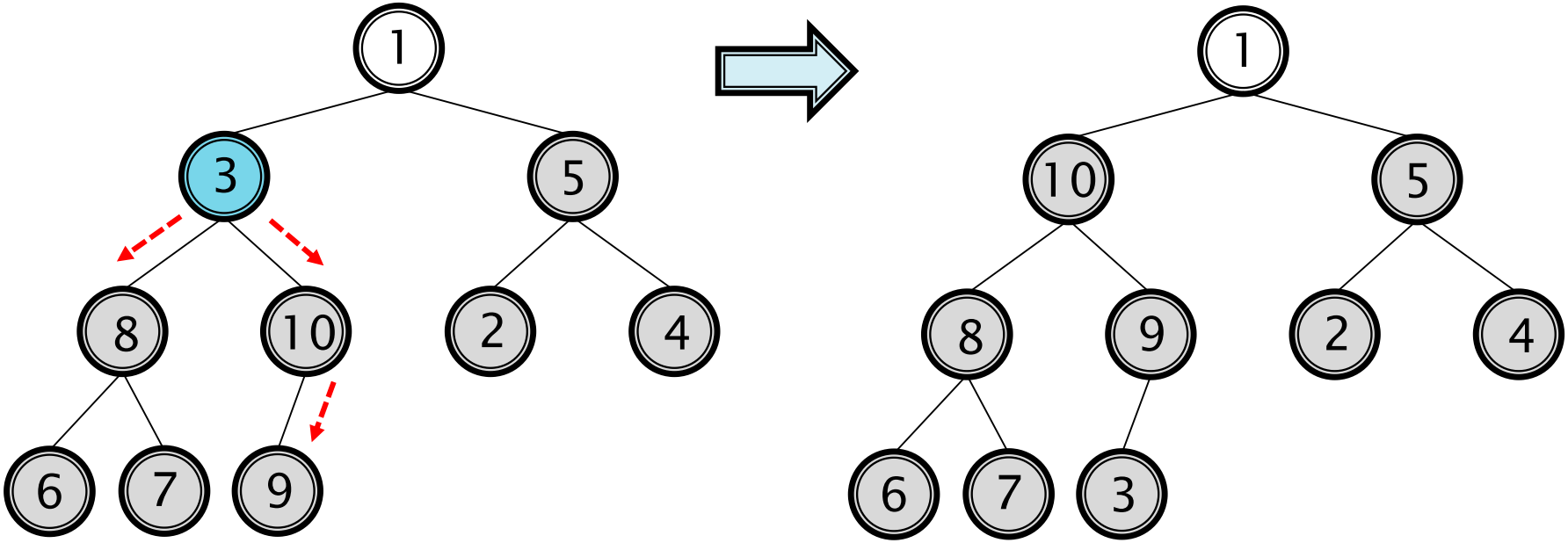
樹狀資料結構

- 由下而上建立最大堆積



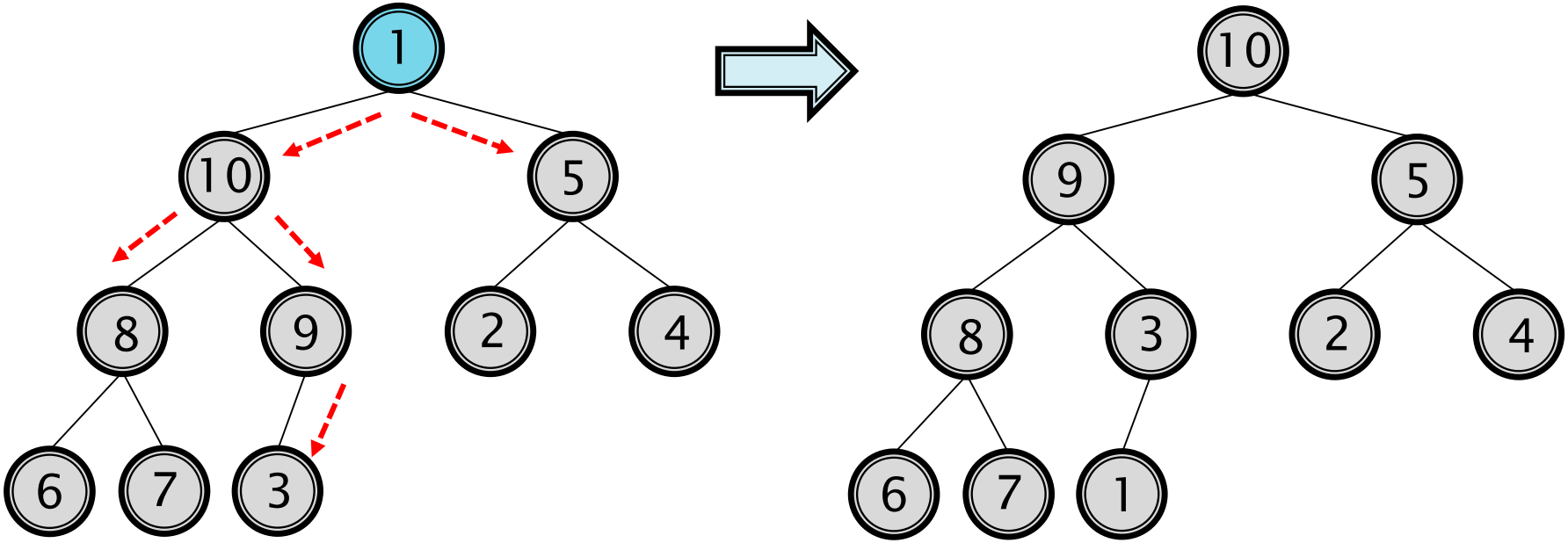
樹狀資料結構

- 由下而上建立最大堆積



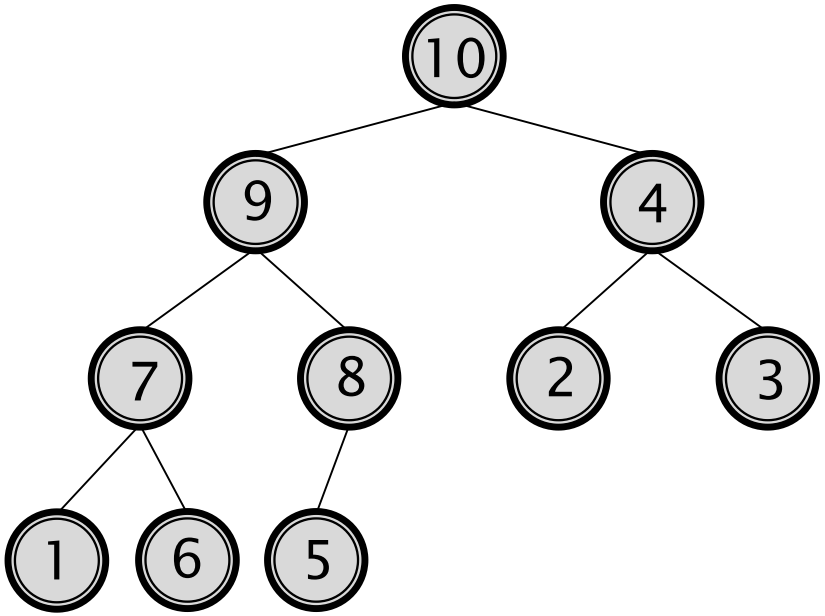
樹狀資料結構

- 由下而上建立最大堆積

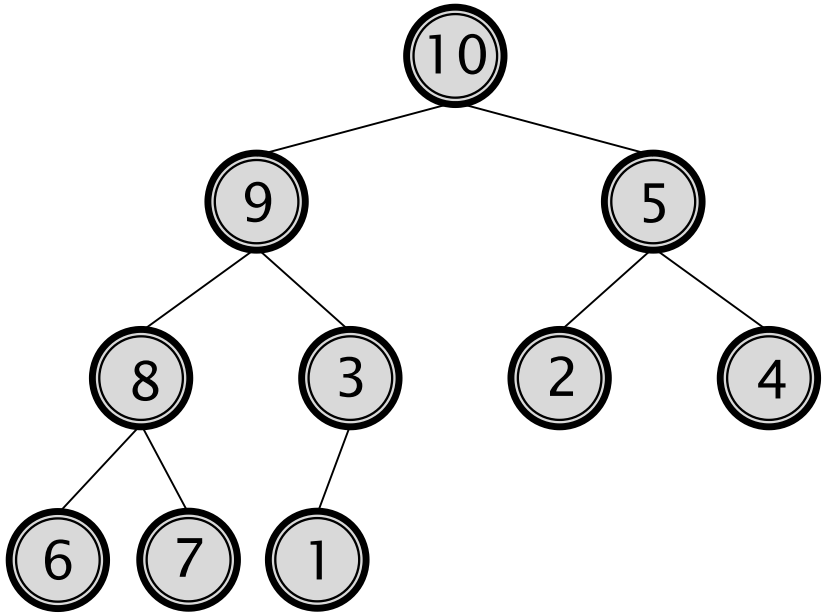


樹狀資料結構

- 這二種方法所建立起來的最大堆積，內容並不相同，但都符合堆積的原則，可見相同的資料所建立的堆積並非唯一。



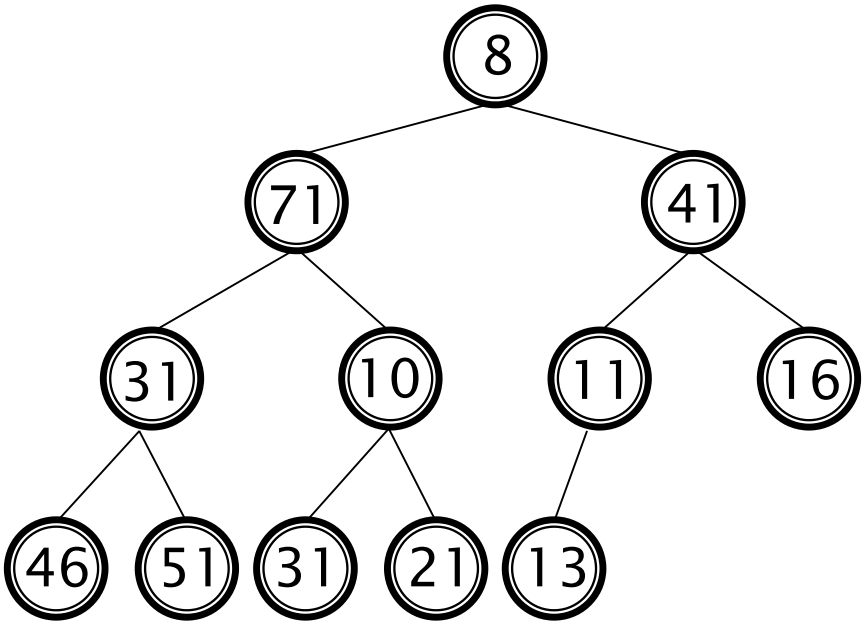
由上而下



由下而上

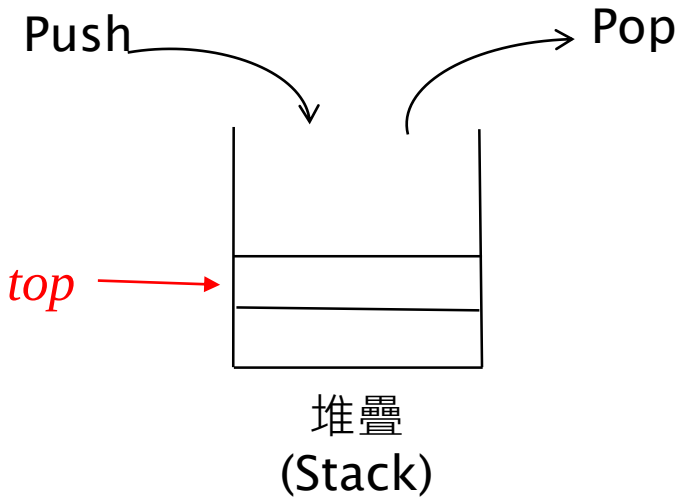
隨堂練習

請利用由下而上方法建置最大堆積



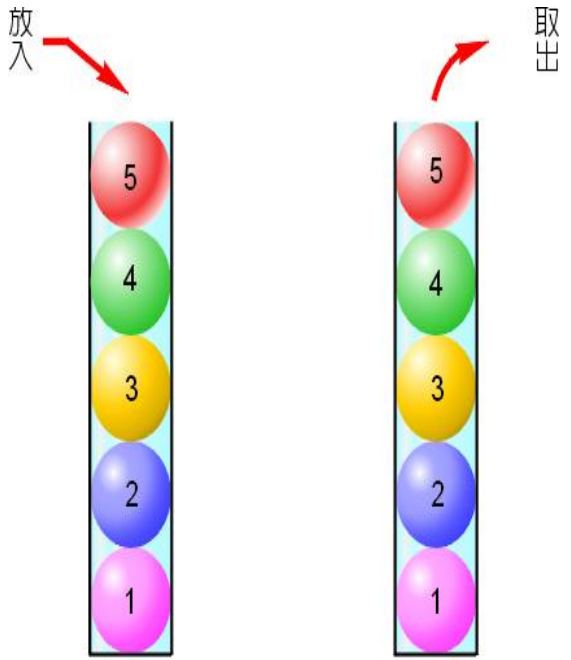
抽象資料結構

- 堆疊(stack)的概念很類似餐廳放盤子的做法，洗好烘乾後的盤子會依序一個一個疊起來，而顧客都是從最上面那個盤子先取用，因此盤子的使用是依照先進後出 (First In Last Out) 或是後進先出 (Last In First Out) 的方式進行的。
- 堆疊的二個常用指令，Push 將資料放入堆疊，Pop 則將資料取出堆疊。



抽象資料結構

- 堆疊(stack)的概念很類似餐廳放盤子的做法，洗好烘乾後的盤子會依序一個一個疊起來，而顧客都是從最上面那個盤子先取用，因此盤子的使用是依照先進後出 (First In Last Out) 或是後進先出 (Last In First Out) 的方式進行的。
- 範例
  - 最早放進去的1號球會在球桶的最下方，而最後放進去的5號球會在球桶的最上方。
  - 要用球時，首先拿到的是球桶最上方的5號球最後才會拿到1號球。



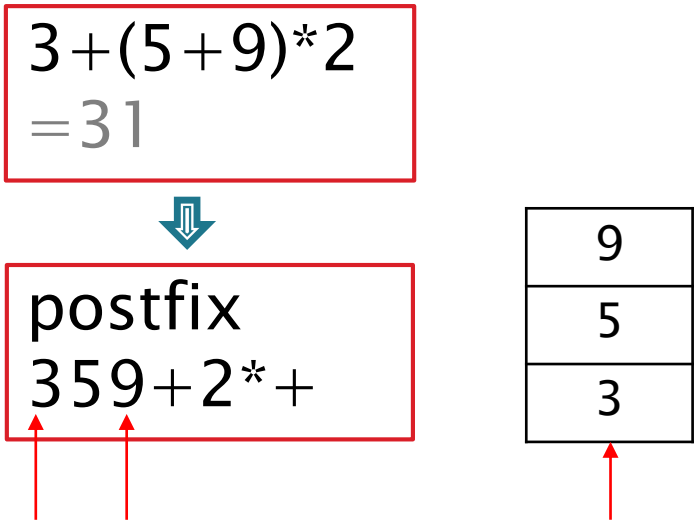
抽象資料結構

- 使用堆疊就可以很容易來檢查括弧是否被正確使用。其實方法很簡單，我們由左往右掃描，碰到左括弧就 Push 到堆疊，碰到右括弧就 Pop 堆疊，如果最後掃描完，堆疊為空，就表示括弧的使用正確。

((()(())))

計算後序式

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則push至stack中
- 遇到運算子則從stack中pop出適當個數的運算元，經運算子處理後再push入stack中

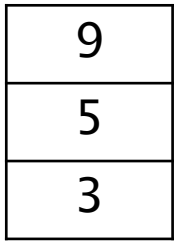


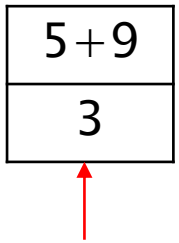
計算後序式

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則push至stack中
- 遇到運算子則從stack中pop出適當個數的運算元，經運算子處理後再push入stack中

3+(5+9)*2
=31

postfix
359+2*+



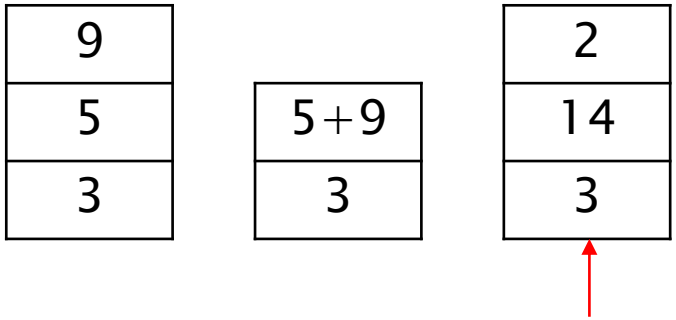


計算後序式

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則push至stack中
- 遇到運算子則從stack中pop出適當個數的運算元，經運算子處理後再push入stack中

3+(5+9)*2
=31

postfix
359+2*+

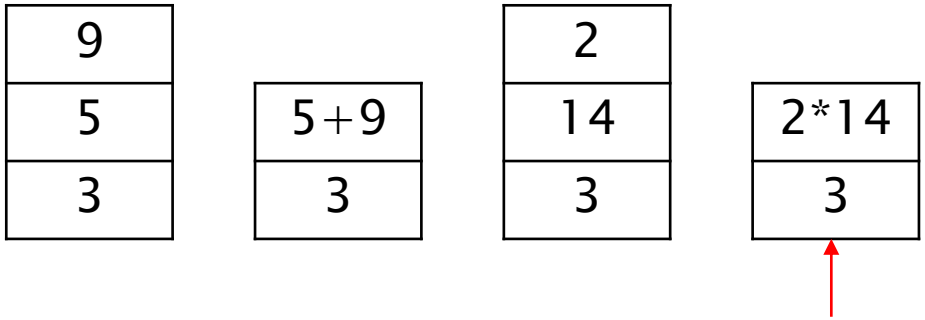


計算後序式

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則push至stack中
- 遇到運算子則從stack中pop出適當個數的運算元，經運算子處理後再push入stack中

3+(5+9)*2
=31

postfix
359+2*+

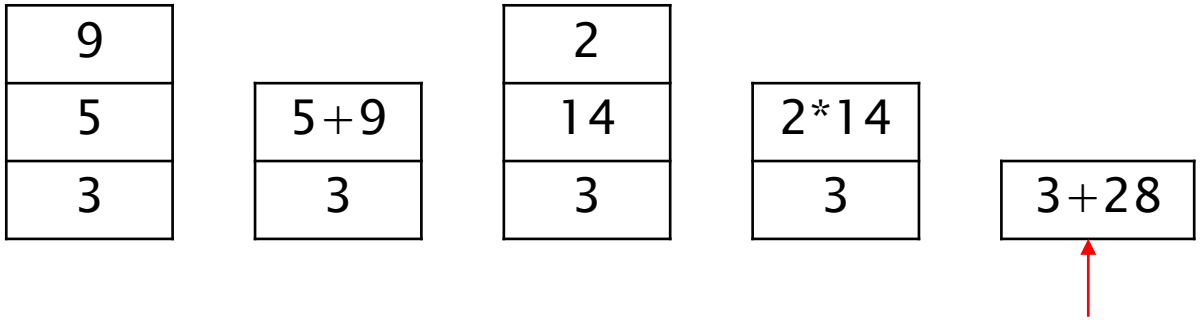


計算後序式

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則push至stack中
- 遇到運算子則從stack中pop出適當個數的運算元，經運算子處理後再push入stack中

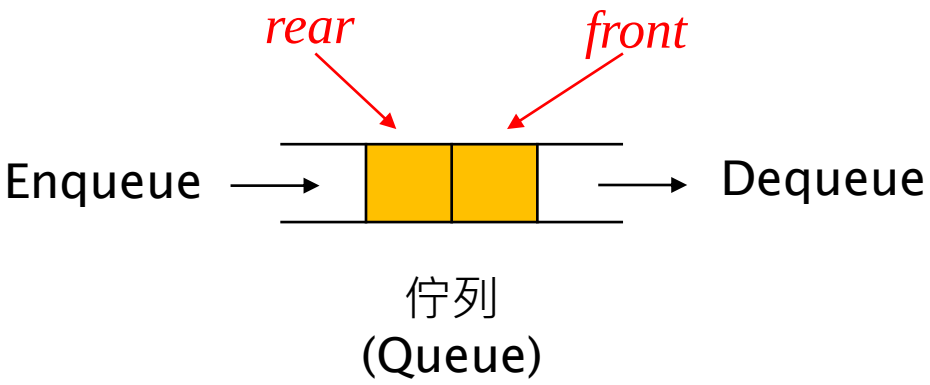
3+(5+9)*2
=31

postfix
359+2*+



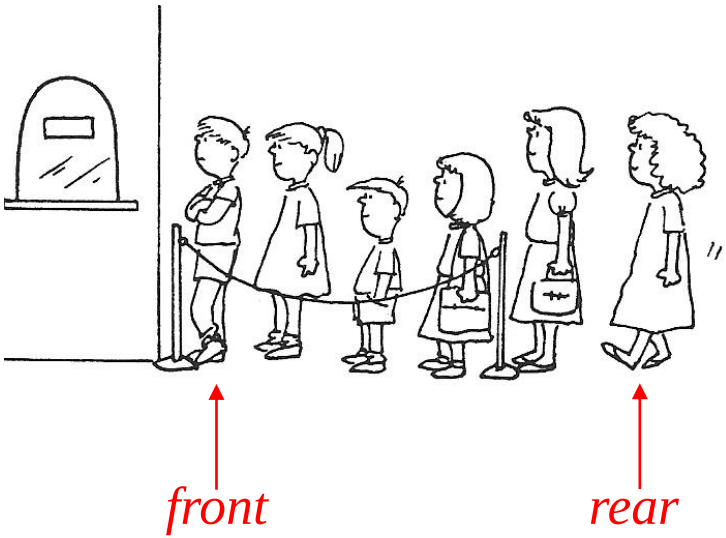
抽象資料結構

- 佇列(queue)的概念就像一般的排隊做法，先到的就排在前面，也就會優先被服務，這個次序是依照先進先出 (First In First Out) 或是後進後出 (Last In Last Out) 的方式進行的。
- 佇列也有二個常用指令，Enqueue 將資料放入佇列，Dequeue 則將資料取出。



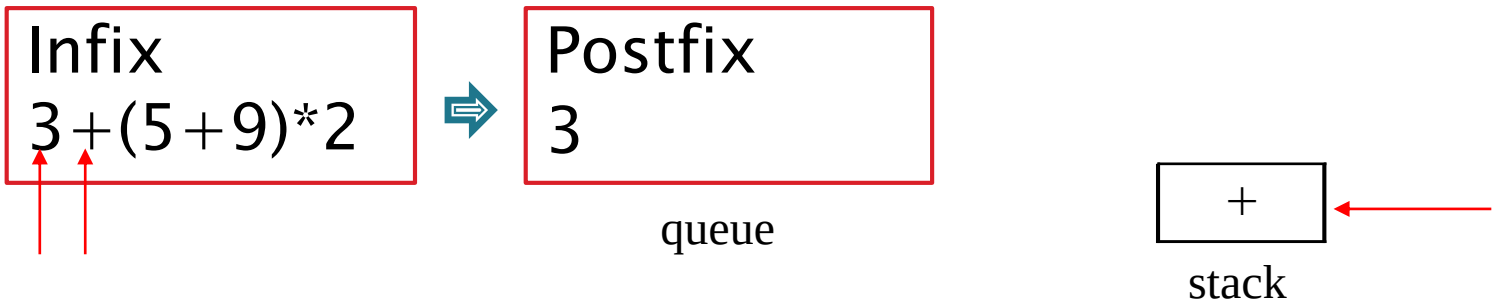
抽象資料結構

- 佇列(queue)的概念就像一般的排隊做法，先到的就排在前面，也就會優先被服務，這個次序是依照先進先出 (First In First Out) 或是後進後出 (Last In Last Out) 的方式進行的。
- 範例
  - front 代表佇列第一筆資料的位置
  - rear 代表佇列最後一筆資料的位置



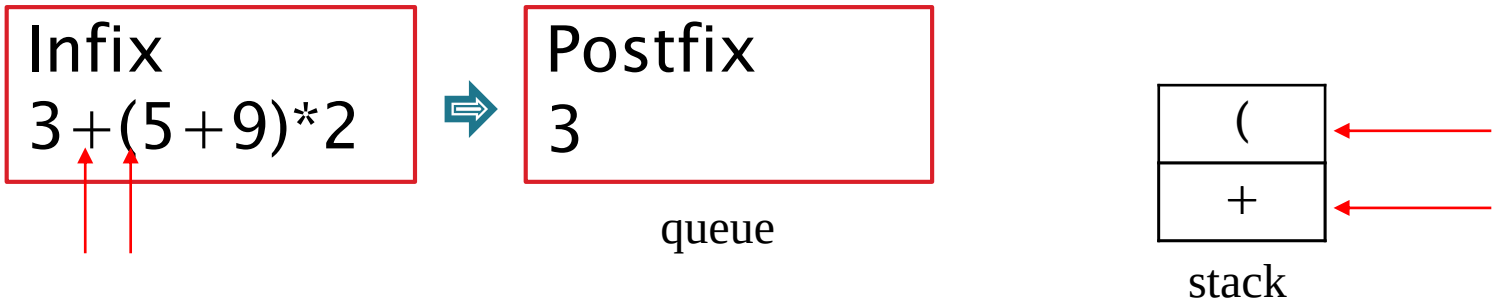
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



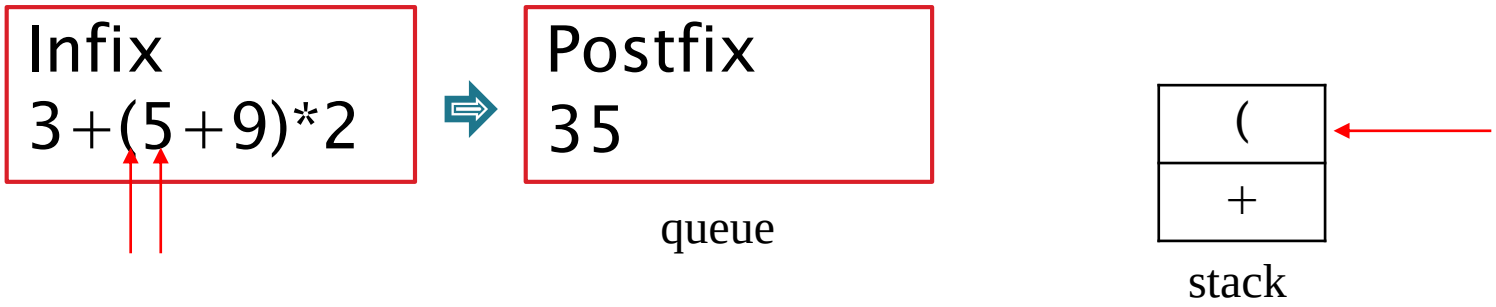
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。

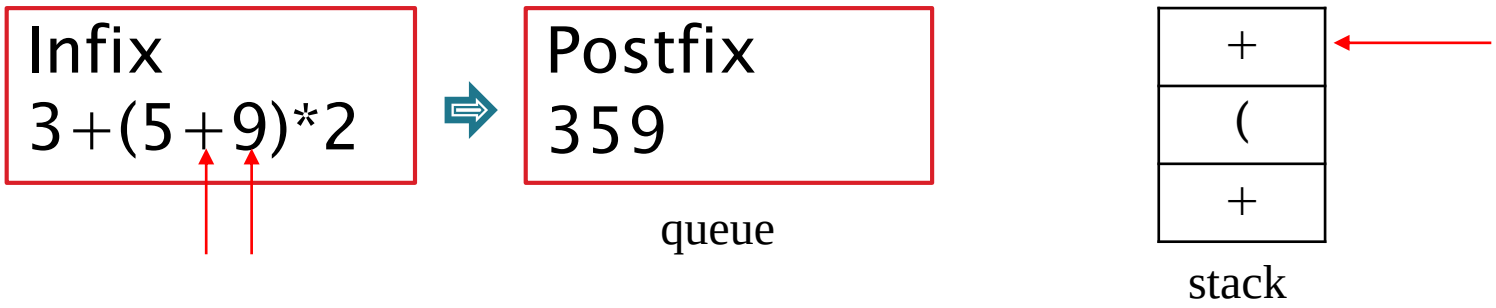


中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。

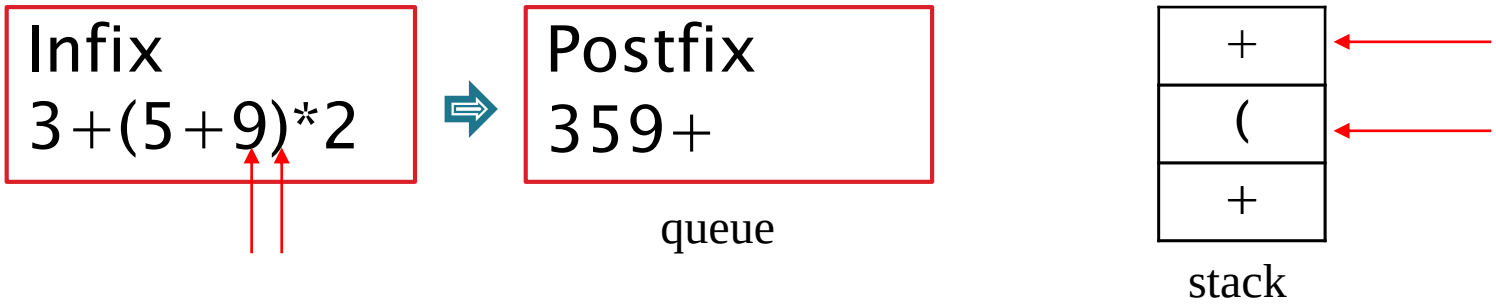
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



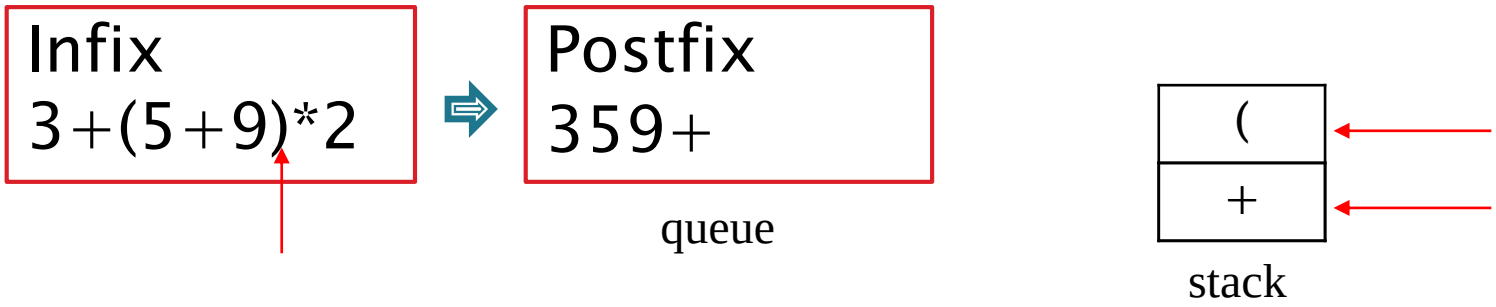
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



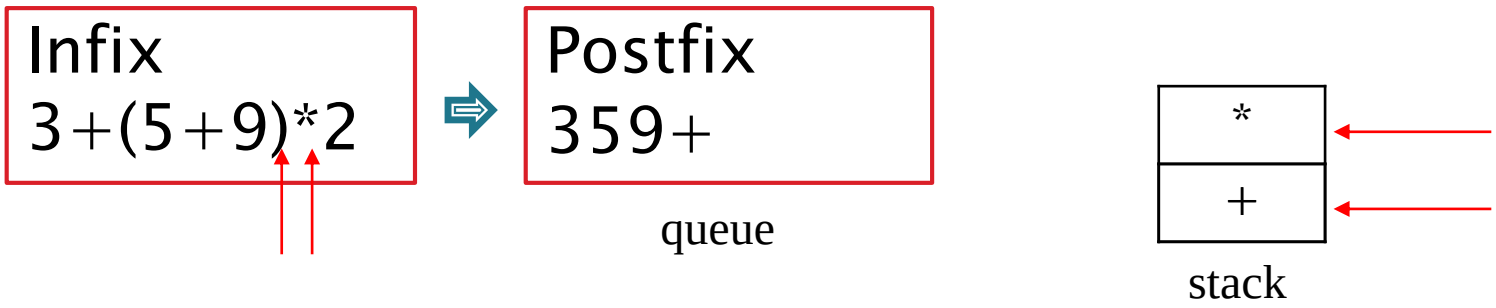
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



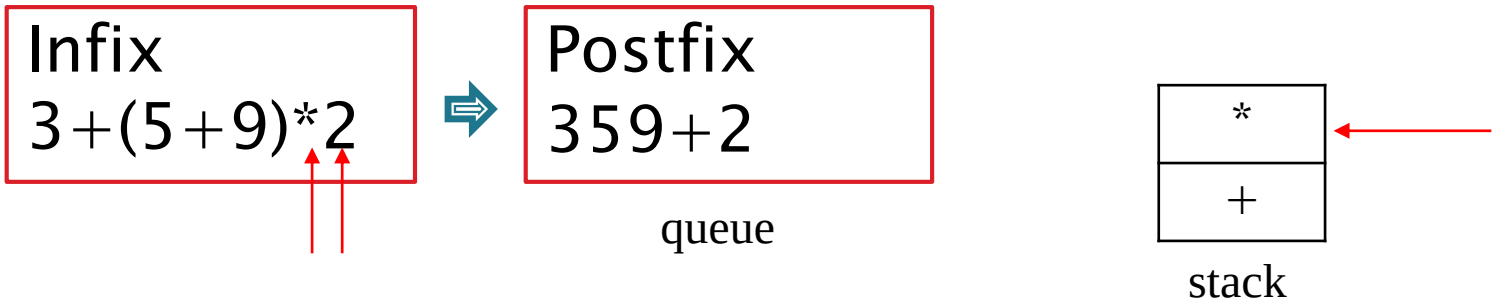
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



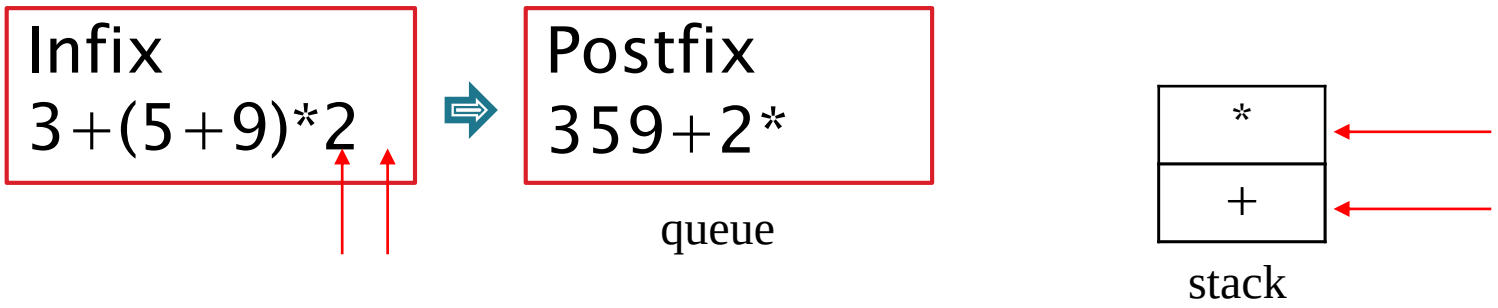
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



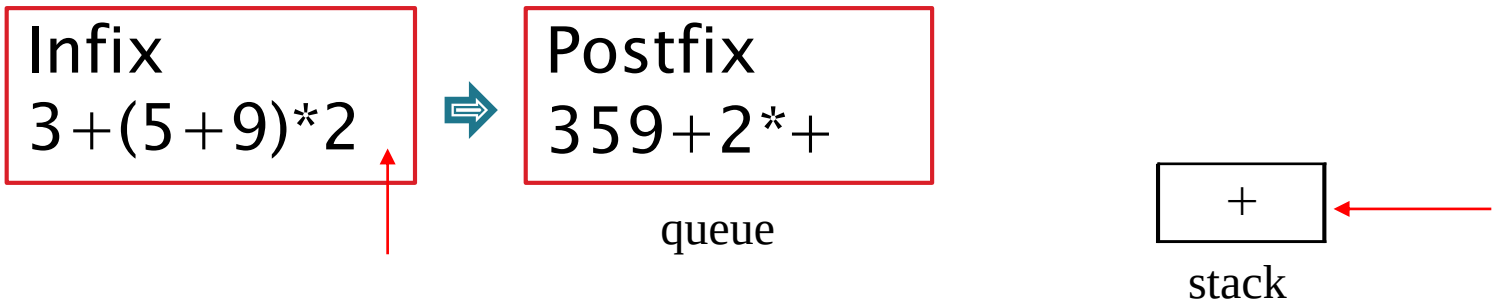
中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



中序轉後序

- 由左至右掃瞄
- 遇到運算元(ex. 數字)則送入佇列queue中
- 遇到運算子則檢查stack中的運算子，若優先權≥則pop並送到queue中，否則push入stack中。遇到左括弧一律push，遇到右括弧則一律pop運算子送進queue中，直到pop左括弧為止。掃描結束，則將stack所有元素pop至queue中。



資料結構運作

- 操作資料結構
  - Insert(), Delete(), Empty(), Full(), …