PREFACE

序言

資料結構（Data Structures）是資訊學科的核心課程之一，也是撰寫程式必備的知識。筆者具有相當豐富的資料結構教學經驗，因此了解應如何闡述資料結構的每一主題，期使讀者達到事半功倍的效果。

C++ 一直是最受歡迎的物件導向程式語言，歷久不衰。因為它除了有強大的指標外還有封裝、繼承、多型，以及樣版（template）的特性，對往後的系統維護相當的容易，所以本書以 C++ 語言加以實作之。讓您了解如何此處假設您已學過 C++ 程式語言。在撰寫內文時，儘量以易懂的方式呈現之，這有異於市面上的「翻譯書」，希望在內文不會讓讀者感到模稜兩可，不易閱讀。

每一章的每一小節幾乎都有練習題，旨在測驗您對此節的了解程度，書後也附有練習題參考解答，不過提醒您，要做完才能對照解答喔。

除了練習題外，在每一章末也有 "動動腦時間"，這些題目有些來自歷屆的高考或研究所的考題，一些則是根據內文加以設計的題目。題目的後面皆標明其出自那一小節，如 [5.2]，它表示只要您詳讀 5.2 節即可輕鬆地作答。

第五版增加了紅黑樹和伸展樹這兩個效率很好的二元搜尋樹，授課老師可以考慮是否有足夠的時數教這兩章，您可以安排在進入圖形結構前或是留在最後才講解。最後在此謝謝各位老師和讀者，因有您們的支持與指教使得本書更加精彩，若發現內文有誤或表達不清楚之處，懇請來信指教，萬分感謝。

mjtsai168@gmail.com

CHAPTER

1

演算法分析

1.1 演算法

演算法(Algorithms)是一個解決問題(problems)的有限步驟程序，也可以說是產生解答中一步一步的程序。舉例來說，現有一問題為：判斷數字 X 是否在一已排序好的數字串列 S 中，其演算法為：從 S 串列的第一個元素開始依序的比較，直到 X 被找到或是 S 串列已達盡頭，假使 X 被找到，則印出 Yes；否則，印出 No。

可是當問題變的很複雜時，上述敘述性的演算法就難以表達出來。因此，演算法大都先以類似程式語言的方式來表達，繼而利用您所熟悉的程式語言執行之。本書乃直接以 C 程式語言來撰寫，因此筆者假設您已具備撰寫 C++語言的能力。

您是否常常會問這樣的一個問題：「他的程式寫得比我好嗎？」，答案不是因為他是班上第一名，所以他所寫出來的程式一定就是最好的。而是應該用一種比較科學的方法來比較之，常用的方法是利用效率分析(performance analysis)方法來進行評估，而效率分析方法通常可分為時間複雜度分析(time complexity analysis)和空間複雜度分析(space complexity analysis)，由於時間複雜度分析較常為人們使用，因此我們選擇此種分析方法來加以評估其效率。首先必須求出程式中每一敘述的執行次數(其中‘{’ 和 ‘}’不加以計算)，之後將這些執行次數加總起來，最後再求出其 Big-O，所求出的結果就是此程式的時間複雜度(關於 Big-O 的部份，1.2 節有詳細的解說)。讓我們先看以下四個範例：

1.1.1 陣列元素相加

陣列元素相加乃將陣列中每一元素的值加總起來，其所對應的 C++片段程式如下：

C++片段程式》

	執行次數
`int sum(int arr[], int n)`	
`{`	
`  int i, total = 0;`	1
`  for (i=0; i<n; i++)`	n+1
`  total += arr[i];`	n
`  return total;`	1
`}`	———
	2n+3

其中在 for 迴圈內的敘述會重複 n 次(由 0，1，2，…，n-1)，但在 i=n 的時候 for 本身仍然會判斷，所以 for 敘述一共做了 n+1 次。

1.1.2 矩陣相加

矩陣相加表示將相對應的元素相加，如 $\begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}+\begin{bmatrix}2 & 3\\3 & 4\end{bmatrix}=\begin{bmatrix}7 & 9\\10 & 12\end{bmatrix}$，而片段程式如下：

C++片段程式》

	執行次數
`void add (int a[][], int b[][], int c[][], int n)`	
`{`	
`  Int i, j;`	n+1
`  for (i=0; i<n; i++)`	n(n+1)
`  for (j=0; j<n; j++)`	n^2
`  c[i][j] = a[i][j] + b[i][j]`	———
`}`	$2n^2+2n+1$

注意！for 敘述本身皆執行 n+1 次，進入迴圈主體後，才執行 n 次。同時，我們假設兩個矩陣皆為 n*n 個元素。

1.1.3 矩陣相乘

矩陣相乘的做法為$\begin{bmatrix}a & b\\c & d\end{bmatrix}\times\begin{bmatrix}e & f\\g & h\end{bmatrix}=\begin{bmatrix}ae+bg & af+bh\\ce+dg & cf+dh\end{bmatrix}$，對應的片段程式如下：

C++片段程式》

	執行次數
`void mul(int a[][], int b[][], int c[][], int n)`	
`{`	
`  int i, j, k, sum;`	1
`  for (i=0; i<n; i++)`	n+1
`  for (j=0; j<n; j++) {`	n(n+1)
`  sum = 0;`	n^2
`  for (k=0; k<n; k++)`	$n^2(n+1)$
`  sum = sum + a[i][k] * b[k][j];`	n^3
`  c[i][j] = sum;`	n^2
`  }`	———
`}`	$2n^3+4n^2+2n+2$

1.1.4 循序搜尋

循序搜尋乃表示在一陣列中，由第 1 個元素開始找起，依序搜尋。此處我們假設要找的資料一定在陣列中，其片段程式如下：

C++片段程式》

	執行次數
`int search(int data[], int target, int n)`	
`{`	
`    int  i;`	1
`    for (i=0; i<n; i++)`	n+1
`        if (target == data[i])`	n
`            return i;`	1
`}`	
	2n+3

練習題

試回答下列片段程式中 x = x+1;這一敘述執行多少次。

```
(a) for (i=1; i<=n; i++)
        for (j=i; j<=n ; j++)
            x = x + 1;

(b) for (i=1; i<=n; i++) {
        k = i + 1;
        do {
            x = x + 1;
        } while (k++ <= n);
    }
```

1.2 Big-O

如何去計算完成一程式或演算法所需要的執行時間呢？在程式或演算法中，每一敘述(statement)的執行時間為：(1)此敘述執行的次數、(2)每一次執行此敘述所需的時間，兩者相乘即為此敘述的執行時間。由於每一敘述所需的時間必需考慮到機器和編譯器的功能，通常假設所需的時間為固定的，因此通常只考慮執行的次數即可。

算完程式中每一敘述的執行次數，並將其加總後，再利用 Big-O 來表示此程式的時間複雜度(time complexity)。

Big-O 的定義如下：

f(n)=O(g(n))，若且唯若存在一正整數 c 及 n_0，使得 f(n) ≤ cg(n)，對所有的 n，n ≥ n_0。

上述的定義表示我們可以找到 c 和 n0，使得 $f(n) \le cg(n)$，此時，我們說 f(n) 的 Big-O 為 g(n)。請看下列範例：

(a) $3n+2 = O(n)$，∵我們可找到 $c=4$，$n_0=2$，使得 $3n+2 \le 4n$

(b) $10n^2+5n+1 = O(n^2)$，∵我們可以找到 $c=11$，$n_0=6$ 使得 $10n^2+5n+1 \le 11n^2$

(c) $7*2^n+n^2+n = O(2^n)$，∵我們可以找到 $c=8$，$n_0=5$ 使得 $7*2^n+n^2+n \le 8*2^n$

(d) $10n^2+5n+1 = O(n^3)$，這可以很清楚的看出，原來 $10n^2+5n+1 \in O(n^2)$，而 n^3 又大於 n^2，理所當然 $10n^2+5n+1=O(n^3)$是沒問題的。同理也可以得知 $10n^2+5n+1 \neq O(n)$，∵f(x)沒有小於等於 c*g(n)。

由上面的幾個範例得知 f(n)為一多項式，表示一程式完成時所需要計算的時間，而其 Big-O 只要取其最高次方的項目即可。

根據上述的定義，得知陣列元素值加總的時間複雜度為 O(n)，矩陣相加的時間複雜度為 $O(n^2)$，而矩陣相乘的時間複雜度為 $O(n^3)$，循序搜尋的時間複雜度為 O(n)。

其實我們可以加以證明，當 $f(n) = a_m n^m + \cdots + a_1 n + a_0$ 時，$f(n) = O(n^m)$。

證明

$$f(n) \le \sum_{i=0}^{m} |a_i| n^i$$

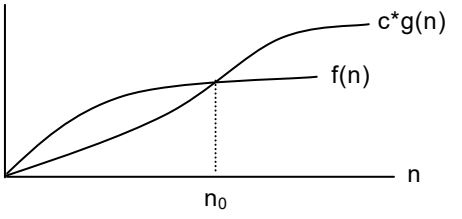
$$\le n^m * \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\le n^m * \sum_{i=0}^{m} |a_i|$$，對 $n \ge 1$ 而言

⇨ $f(n) \in O(n^m)$．∵可將 $\sum_{i=0}^{m} |a_i|$ 視為 c，而 n^m 為 g(n)

亦即 Big-O 乃取其最大指數的部份即可，因此前述的範例中，陣列元素相加的 Big-O 為 O(n)，矩陣相加 Big-O 為 $O(n^2)$，而矩陣相乘的 Big-O 為 $O(n^3)$。

Big-O 的圖形表示如下：

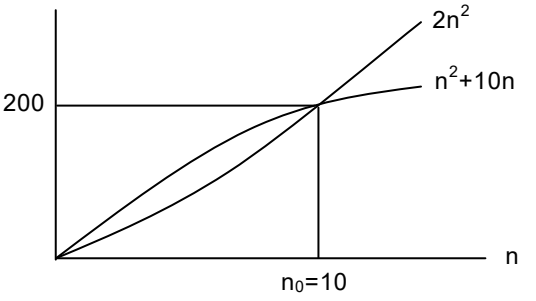


例如有一程式執行的時間為 n^2+10n，則其 Big-O 為 (n^2)，表示程式執行所花的時間最多有 n^2 的時間；換個角度說，就是在最壞的情況下也不會大於 n^2。

以 Big-O 的定義：

$n^2+10n \le 2n^2$ ，當 $c = 2$ 時，$n_0 \ge 10$ 時

$=> n^2+10n \in O(n^2)$



一般常見的 Big-O 有以下的幾種類別：

Big-O	類別
$O(1)$	常數時間 (constant)
$O(\log_2 n)$	對數時間 (logarithmic)
$O(n)$	線性時間 (linear)
$O(n\log_2 n)$	對數線性時間 (log linear)
$O(n^2)$	平方時間 (quadratic)
$O(n^3)$	立方時間 (cubic)
$O(2^n)$	指數時間 (exponential)
$O(n!)$	階層時間 (factorial)
$O(n^n)$	n 的 n 次方時間

一般而言，這幾種類別由 $O(1)$，$O(\log_2 n)$ ，…，$O(n!)$，$O(n^n)$之效率按照排列的順序愈來愈差，也可以下一種方式表示。

$$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

$O(1) < O(\log_2 n)$表示後者所花的時間大於前者，因此效率上前者較後者優。往後我們可以利用 Big-O 來評量程式或演算法的效率為何。當 n 愈大時，更能顯示出其間的差異，如表 1.1 所示。若某位同學的程式 Big-O 為 $O(n\log_2 n)$，而你的程式為 $O(n)$，則你的程式之執行效率比那位同學來得優。

表 1.1　各種 Big-O 的比較表

n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296

表 1.1 明顯的可以看出，當 n 愈來愈大時，$n\log_2 n$，n^2，n^3 和 2^n 之間的差距便愈來愈大，如 n=32 時，$\log_2 n$ 才為 5，但 n^2 就等於 1,024，n^3 更大，已為 32,768，而 2^n 此時已達到 4,294,967,296，之間的差距是相當大的，在表 1.1 我們省略了 n!，因為當 n=32 時，幾乎印出的數字差不多有 30 幾位數囉!各位讀者只要清楚 Big-O 類別之間的排列便可。

除了 Big-O 之外，用來衡量效率的方法還有Ω 和 Θ，以下是它們的定義。

Ω 的定義如下：

> f(n) = Ω(g(n)),若且唯若，存在正整數 c 和 n0，使得 f(n) ≥ cg(n)，對所有的 n，n ≥ n0。

請看下面幾個範例：

(a) $3n+2 = \Omega(n)$，∵我們可找到 $c = 3$，$n_0 = 1$

使得 $3n+2 \geq 3n$

(b) $200n^2+4n+5 = \Omega(n^2)$，∵我們可找到 $c = 200$，$n_0 = 1$

使得 $200n^2+4n+5 \geq 200\ n^2$

(c) $10n^2+4n+2 = \Omega(n)$，為什麼呢？

∵ 從定義得知 $10n^2+4n+2 = \Omega(n^2)$，

由於 $n^2 > n$，∴理所當然 $10n^2+4n+2$ 也可視為 $\Omega(n)$。

Θ 的定義如下：

> f(n)= Θ (g(n)),若且唯若，存在正整數 c_1，c_2 及 n，使得 c_1*g(n) ≤ f(n) ≤ c_2*g(n)，對所有的 n，n ≥ n_0。

我們以下面幾個範例加以說明：

(a) $3n+1 = \Theta(n)$，∵我們可以找到 $c_1 = 3$，$c_2 = 4$，且 $n_0 = 2$，

使得 $3n \leq 3n+1 \leq 4n$

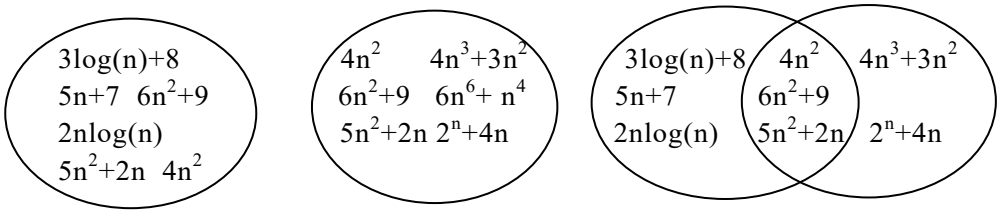
(b) $10n^2+4n+6 = \Theta(n^2)$，∵只要 $c_1 = 10$，$c_2 = 11$ 且 $n_0 = 10$

便可得 $10n^2 \leq 10n^2+4n+6 \leq 11n^2$

(c) 注意！$3n+2 \neq \Theta(n^2)$，$10n^2+n+1 \neq \Theta(n)$

讀者可加以思考一下下。

下圖為 Big-O, Ω, Θ 的表示情形：



(a) $O(n^2)$ (b) $\Omega(n^2)$ (c) $\Theta(n^2)$，只有中間交集部份

有些問題，只要知道其做法便可求出其 Big-O，底下我們舉一些範例說明之。

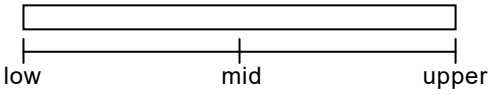
循序搜尋 (sequential search) 的情形可分為三種，第一種為最壞的情形，當要搜尋的資料放置在檔案的最後一個，因此需要 n 次才會搜尋到(假設有 n 個資料在檔案中)；第二種為最好的情形，此情形與第一種剛好相反，表示欲搜尋的資料在第一筆，故只要 1 次便可搜尋到；最後一種為平均狀況，其平均搜尋到的次數為：

$$\sum_{k=1}^{n}(k*(1/n)) = (1/n)* \sum_{k=1}^{n}k = (1/n)(1+2+\cdots+n) = 1/n*(n(n+1)/2)= (n+1)/2$$

因此得知其 Big-O 為 O(n)。

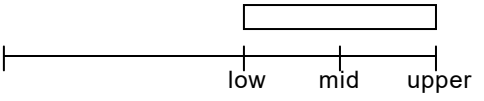
二元搜尋 (Binary search) 的情形和循序搜尋不同，二元搜尋法乃是資料已經排序好，因此由中間的資料 (mid) 開始比較，便可知道欲搜尋的鍵值 (key) 是落在 mid 的左邊還是右邊，之後，再將左邊或右邊中間的資料拿出來與欲搜尋的鍵值相比，而每次所要調整的只是每個段落的起始位址或是最終位址。

例如：



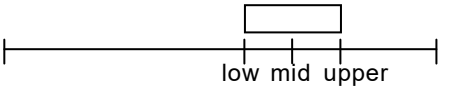
當 key > data[mid] 時，low = mid +1，而 upper 不變，如下圖所示：

此例為調整起始位址。



當 key < data[mid]時，upper = mid －1，而 low 不變；如下圖所示：

此例為調整最終位址。



CHAPTER

15

紅黑樹

即使我們已學過 AVL 樹和 2-3 tree 與 2-3-4 tree，但由於 AVL–tree 在刪除時可能要許多的重構和旋轉的動作，而 2-3 tree 與 2-3-4 tree 在加入與刪除時可能要處理一些分裂與融合。但紅黑樹就不需要這些缺點，它只要花費 O(1) 的時間就可以完成結構的改變，而又保持平衡的狀態。

15.1 紅黑樹的定義

顧名思義，紅黑樹(red-black tree)的節點會有紅(red)與黑(black)之分，它於 1972 年由 Rudolf Bayer 所發明的，其定義如下：

1. 樹根的節點一定是黑色。
2. 每一外部節點都是黑色。
3. 紅色節點的子節點一定是黑色，因為不可以兩個連續的紅色節點。
4. 從樹根到每一外部節點，其黑色線條的個數要相同。

例如圖 15.1 是一棵紅黑樹

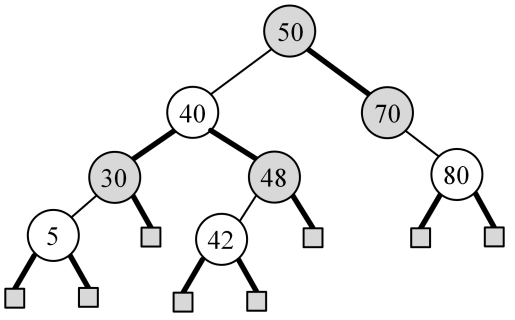


圖 15.1 一棵紅黑樹

圖中所顯示節點與節點之間的線條，若指向的節點是黑色節點，則線條以粗線表示，若指向的節點是紅色節點，則線條以細線表示。由於是黑白印刷，所以黑色節點填滿灰階，而紅色節點沒有，以此表示之。

15.1.1 加入一節點

加入一紅色節點時可能會由原來是一棵紅黑樹變為不是一棵紅黑樹，此時要加以調整，使其成為是一棵紅黑樹。以下以 n 表示新加入的節點，p 表示新加入節點的父節點，g 表示新加入節點的祖先節點 。

加入一節點時，若不符合紅黑樹時，其調整的型式共有八種型式，分別如下：

1. LL_r 型式

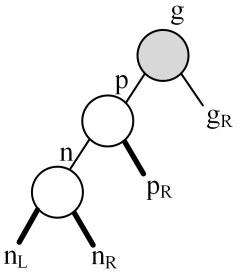


圖 15.2 加入 n 節點後，不是一棵紅黑樹

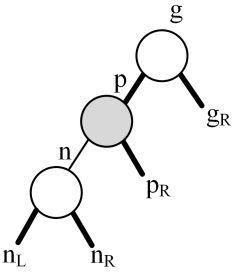


圖 15.3 經由 LL_r 調整之後的紅黑樹

此型式是將圖 15.2 的 p 節點轉為黑色，而 g 節點轉為紅色，但其右線條轉為黑色，表示將指向黑色節點。如圖 15.3 所示。

2. RR_r 型式

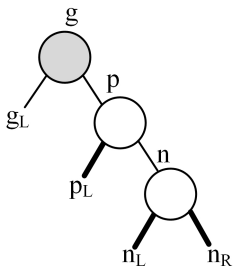


圖 15.4 加入 n 節點後，不是一棵紅黑樹

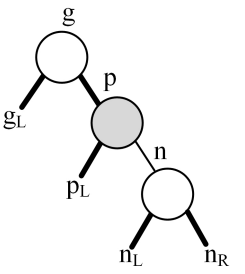


圖 15.5 經由 RR_r 調整之後的紅黑樹

此型式是將圖 15.4 的 p 節點轉為黑色，而 g 節點轉為紅色，但其左線條將為黑色，表示將指向黑色節點。如圖 15.5 所示。

3. LR_r 型式

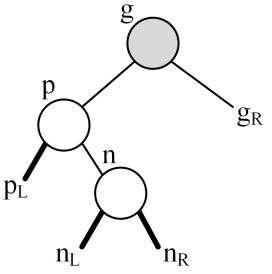


圖 15.6 加入 n 節點後，不是一棵紅黑樹

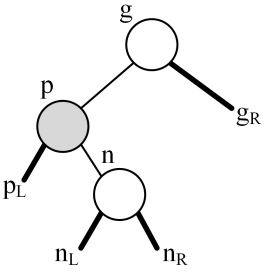


圖 15.7 經由 LR_r 調整之後的紅黑樹

此型式是將圖 15.6 的 p 節點轉為黑色，而 g 節點轉為紅色，但其右線條將為黑色，表示將指向黑色節點。如圖 15.5 所示。

4. RL_r 型式

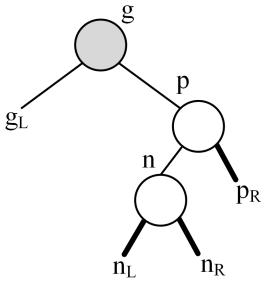


圖 15.8 加入 n 節點後，不是一棵紅黑樹

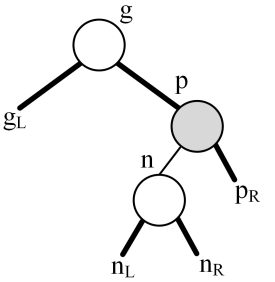


圖 15.9 經由 RL_r 調整之後的紅黑樹

此型式是將圖 15.8 的 p 節點轉為黑色，而 g 節點轉為紅色，但其左線條將為黑色，表示將指向黑色節點。如圖 15. 9 所示。

以上所談四種調整型式的是加入的 n 節點，其祖先節點 g 的左邊或右邊的線條是紅色。下面所談的其線條則是黑色。

5. LL_b 型式

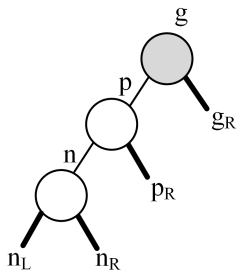


圖 15.10 加入 n 節點後，不是一棵紅黑樹

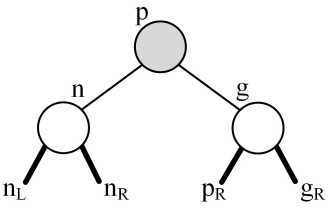


圖 15.11 經由 LL_b 調整之後的紅黑樹

此型式是將圖 15.10 的 p 節點轉為黑色，而 g 節點轉為紅色，並將 p 節點向右旋轉，使 p 節點成為它是 n 和 g 的父節點。如圖 15.11 所示。

6. RR_b 型式

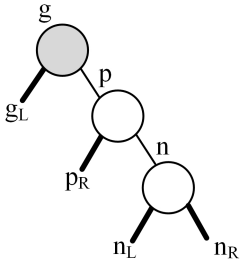


圖 15.12 加入 n 節點後，不是一棵紅黑樹

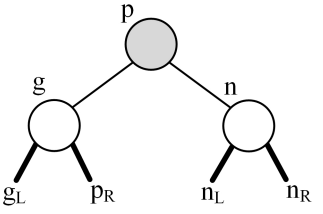


圖 15.13 經由 RR_b 調整之後的紅黑樹

此型式是將圖 15.12 的 p 節點轉為黑色，而 g 節點轉為紅色，並將 p 節點向左旋轉，使 p 節點成為它是 n 和 g 的父節點。如圖 15.13 所示。

7. LR_b 型式

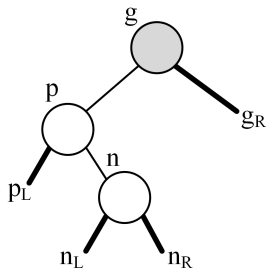


圖 15.14 加入 n 節點後，不是一棵紅黑樹

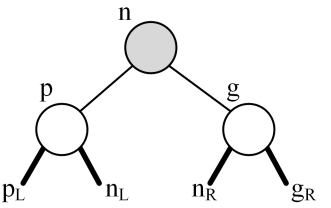


圖 15.15 經由 LR_b 調整之後的紅黑樹

此型式是將圖 15.14 的 n 節點轉為黑色，而 g 節點轉為紅色，並將 n 節點向上拉起，使 n 節點成為它是 p 和 g 的父節點。如圖 15.15 所示。

8. RL_b 型式

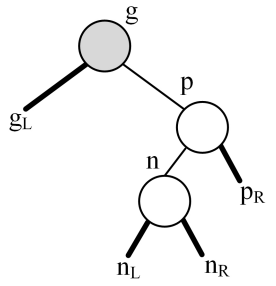


圖 15.16 加入 n 節點後，不是一棵紅黑樹

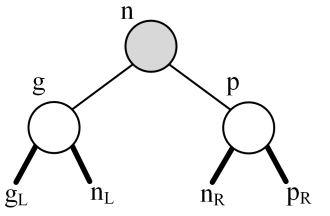


圖 15.17 經由 RL_b 調整之後的紅黑樹

此型式是將圖 15.16 的 n 節點轉為黑色，而 g 節點轉為紅色，並將 n 節點向上拉起，使 n 節點成為它是 p 和 g 的父節點。如圖 15.17 所示。

我們舉一範例來說明之：

以下的序列圖是與前一圖形相關的。

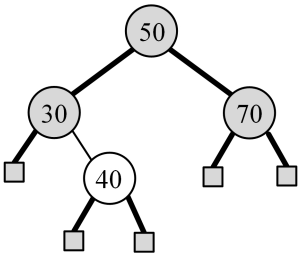


圖 15.18 初始紅黑樹

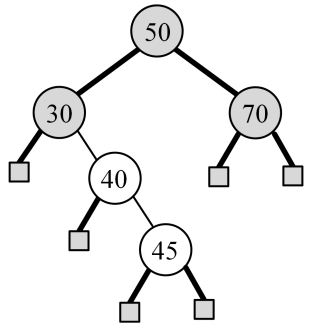


圖 15.19 加入 45

圖 15.19 加入 45 後變為不是一棵紅黑樹。

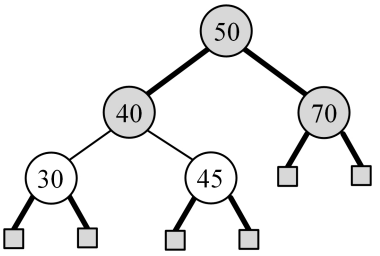


圖 15.20 經由 RR_b 調整後的紅黑樹

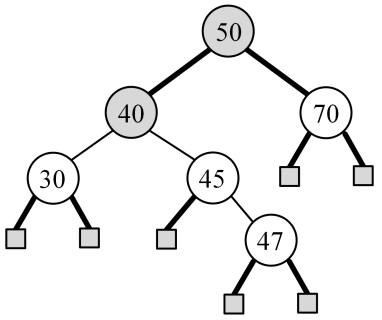


圖 15.21 加入 47

圖 15.20 是將圖 15.19 經由 RR_b 的型式調整為紅黑樹。圖 15.21 加入 47 後變為不是一棵紅黑樹。

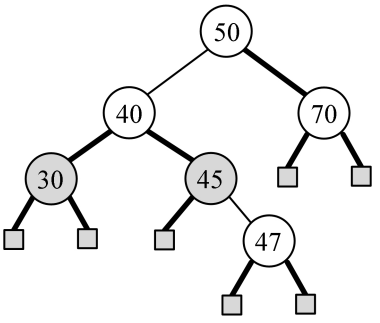


圖 15.22 經由 RR_r 調整後的紅黑樹

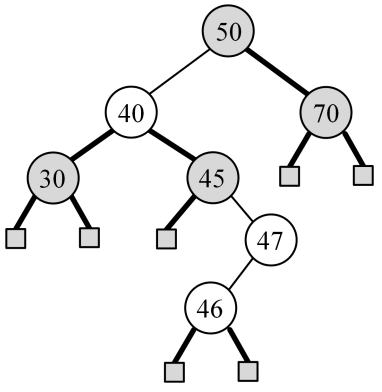


圖 15.23 加入 46

圖 15.22 是將圖 15.21 經由 RR_r 的型式調整而來。圖 15.23 加入 46 後變為不是一棵紅黑樹。

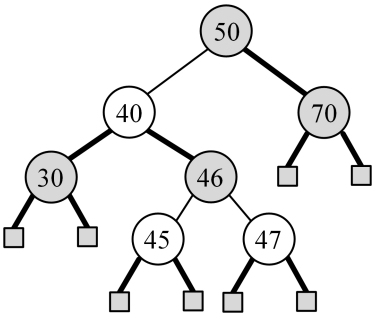


圖 15.24 經由 RL_b 調整後的紅黑樹

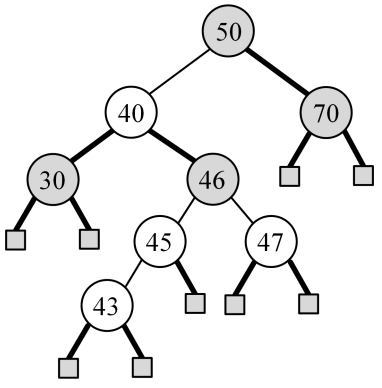


圖 15.25 加入 43

圖 15.24 是將圖 15.23 經由 RL_b 的型式調整而來。圖 15.25 加入 43 後變為不是一棵紅黑樹。

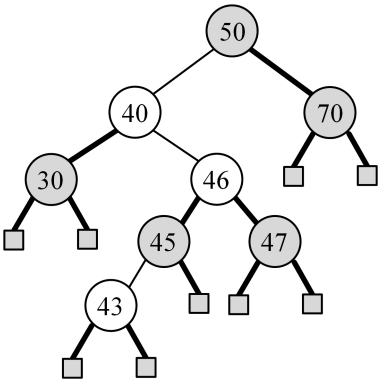


圖 15.26 經由 LL_r 調整後的紅黑樹

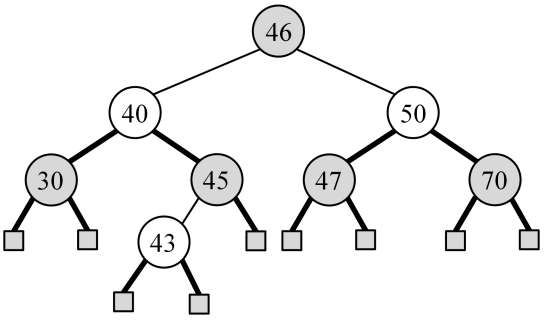


圖 15.27 再經由 LR_b 調整後的紅黑樹

圖 15.26 是將圖 15.25 經由 LL_r 的型式調整而來，但此時還不是一棵紅黑樹，因為節點 40 和節點 46 連續兩個紅色節點。圖 15.27 再經由 LR_b 的型式調整而來。此時已是一棵紅黑樹。

15.1.2 刪除一節點

若要刪除紅黑樹的某一節點，則會有以下七種調整的型式。

1. 若刪除的節點是樹葉節點而且又是紅色，如圖 15.28 所示：

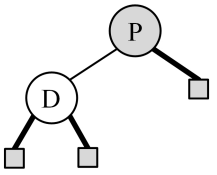


圖 15.28 刪除節點 D 是樹葉節點而且是紅色

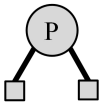


圖 15.29 刪除後的紅黑樹

此情形則直接刪除，如圖 15.29 所示。

2. 若刪除的節點的父節點是紅色，如圖 15.30 所示。

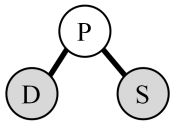


圖 15.30 刪除節點 D 的父節點是紅色

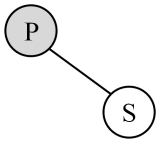


圖 15.31 調整後的紅黑樹

此情形將父節點變數黑色，兄節點變為紅色，如圖 15.31 所示。

3. 若刪除的節點的兄弟節點是紅色。如圖 15.32 所示：

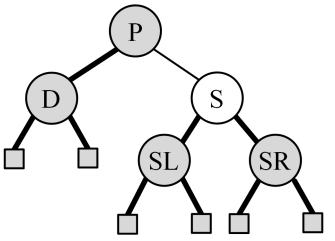


圖 15.32 刪除節點 D 的兄弟節點 S 是紅色

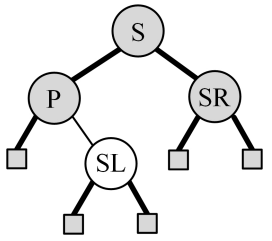


圖 15.33 調整後的紅黑樹

此情形調整方式如下：

(a) 先將兄弟節點變為黑色，並將靠近刪除節點的近親侄子節點變為紅色，

(b) 再往刪除節點的方向旋轉，

如圖 15.33 所示。

4. 若刪除的節點的遠親姪子是紅色，如圖 15.34 所示：

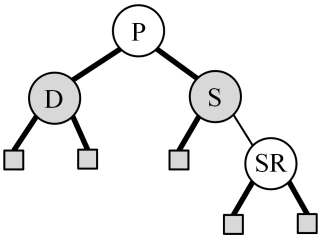


圖 15.34 刪除節點 D 的遠親姪子 SR 是紅色

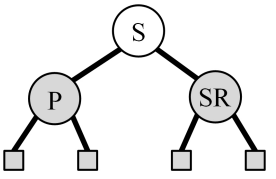


圖 15.35 調整後的紅黑樹

此情形的調整方式如下：

(a) 先將其欲刪除節點的兄弟節點轉為與父節點同顏色，

(b) 再將父節點與遠親姪子節點設定為黑色，

(c) 最後若欲刪除節點是在遠親姪子的左邊，則以其兄弟節點為軸心向左旋轉，如圖 15.35 所示。若欲刪除節點是在遠親姪子的右邊，則以其兄弟節點為軸心向右旋轉。

5. 若刪除的節點的近親姪子是紅色，如圖 15.36 所示：

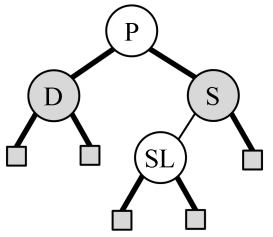


圖 15.36 刪除節點 D 的近親姪子 SL 是紅色

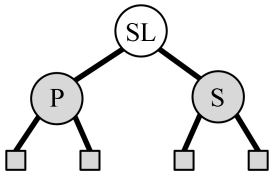


圖 15.37 調整後的紅黑樹

此情形的調整方式如下：

(a) 先將欲刪除節點 D 的近親姪子節點 SL 轉為與欲刪除節點的父節點 P 相同的顏色，

(b) 再將父節點 P 設定為黑色，

(c) 最後將近親姪子 SL 往上提，如圖 15.37 所示。

6. (a) 若刪除節點與父節點和兄弟節點都是黑色，如圖 15.38 所示。

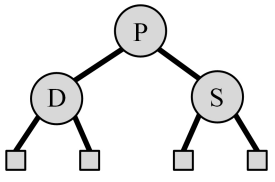


圖 15.38 刪除節點 D 與父節點 P 和兄弟節點 S 都是黑色

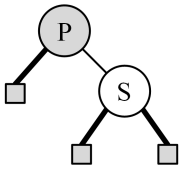


圖 15.39 調整後的紅黑樹

此情形的調整方式如下：先將欲刪除節點 D 的兄弟節點 S 變為紅色，如圖 15.39 所示。

(b) 若刪除節點的父節點兄弟節點、祖先節點都是黑色，如圖 15.40 所示：

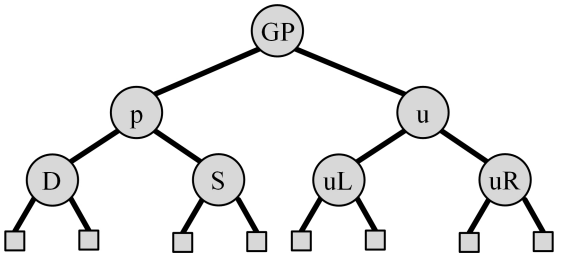


圖 15.40 刪除節點 D 與父節點、兄弟節點，以及祖先節點都是黑色

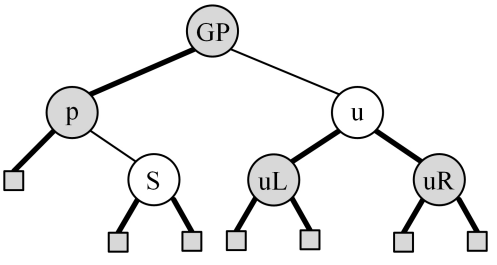


圖 15.41 調整後的紅黑樹

此情形的調整方式如下：先將欲刪除節點 D 的兄弟節點變 S 為紅色，接著再將其父節點 P 的兄弟節點 u 變為紅色，如圖 15.41 所示。

(c) 刪除節點與父節點和兄弟節點是黑色，但祖先節點都是紅色，如圖 15.42 所示：

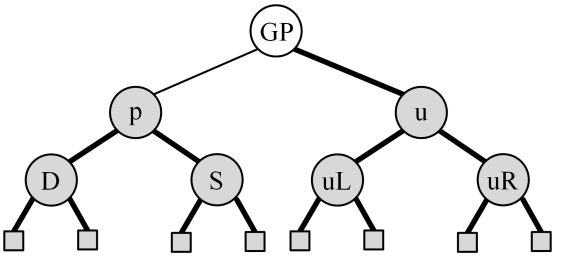


圖 15.42 刪除節點 D 與父節點和兄弟節點是黑色，但祖先節點都是紅色

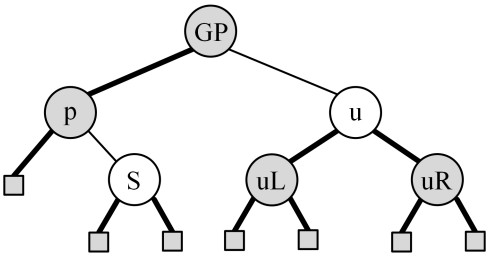


圖 15.43 調整後的紅黑樹

此情形的調整方式如下：先將欲刪除節點 D 的兄弟節點 S 變為紅色，接著再將其父節點的兄弟節點 u 變為紅色，最後將祖父節點 GP 變為黑色，如圖 15.43 所示。

7. 若刪除的節點只有左子樹葉，而且這樹葉節點是紅色，如圖 15.44 所示：

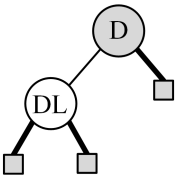


圖 15.44 刪除節點 D 的子節點 DL 是紅色

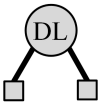


圖 15.45 調整後的紅黑樹

此情形的調整方只要將左或右的樹葉節點轉為黑色就可以，如圖 15.45 所示。

接著我們來看一範例，並從中說明其調整的型式。以下的序列圖有連帶關係。

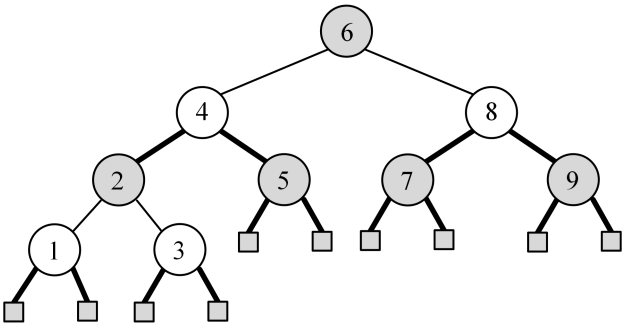


圖 15.46 初始紅黑樹

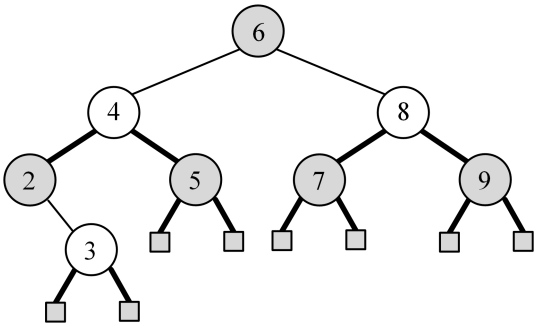


圖 15.47 刪除 1 調整後的紅點樹(應用調整型式 1)

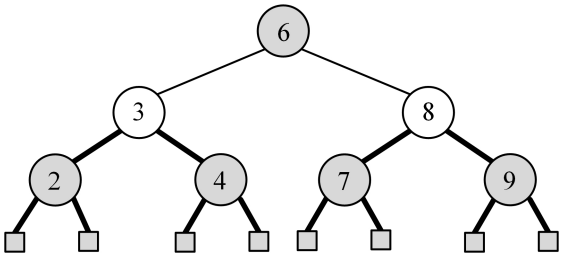


圖 15.48 刪除 5 調整後的紅點樹
(應用調整型式 5)

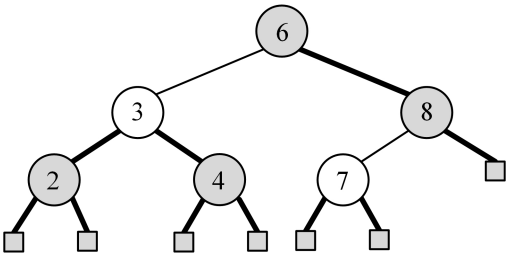


圖 15.49 刪除 9 調整後的紅點樹
(應用調整型式 2)

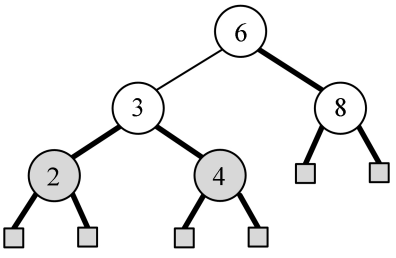


圖 15.50 刪除 7 調整後的紅點樹
(應用調整型式 1)

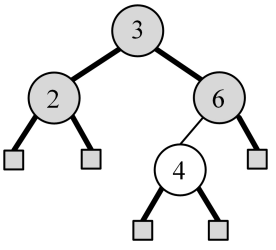


圖 15.51 刪除 8 調整後的紅點樹
(應用調整型式 3)

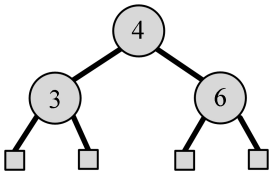


圖 15.52 刪除 2 調整後的紅點樹
(應用調整型式 5)

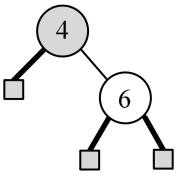


圖 15.53 刪除 3 調整後的紅點樹
(應用調整型式 6(a))

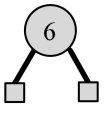
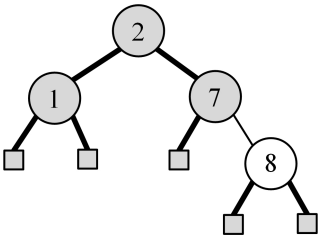


圖 15.54 刪除 4 調整後的紅點樹
(應用調整型式 7)

練習題

1. 刪除的規則在節所述的內文中，刪除節點都是在左子樹，依此類推畫出刪除的節點在右子樹的這些規則之圖形。
2. 若有一棵紅黑樹如下：

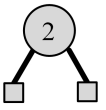


依序加入 9，12，11 的資料，並寫出依據哪一規則加以調整的。

3. 承 2，依序刪除 9，7，8，1，2，11 所對應的紅黑樹。並寫出依據哪一個規則加以調整的。

15.2 動動腦時間

1. 若有一棵紅黑樹如下：



請依序加入 5，10，13，12，6 的資料，並寫出依據哪一規則加以調整的。

2. 承 1 的結果，依序刪除 10，12，16，2，5 所對應的紅黑樹。