專題報告

從根談起紅黑樹

中央資工系　一年級　楊子慶

壹、 簡介

本專題從探討樹型資料結構開始，證明二分搜尋樹的空間優勢，說明平衡性導致的效能問題。接著介紹針對此問題而衍伸出的解決方案：B 樹，概括定義、操作和優缺點分析。之後進入正題，以「模擬 B 樹」的視角重新理解晦澀的紅黑樹，並分析其樹高與平衡性。然後回顧紅黑樹的定義、簡述其家族以及和 AVL 樹做比較。最後討論紅黑樹的實作，並附上自己的程式碼連結。

貳、 正文

一、 二分搜尋樹

在眾多資料結構（往後稱資結）裡，最基本的線型資結在增刪改查分別有突出的表現，如 ArrayList 的改查及 LinkedList 的增刪，複雜度都是 $O(1)$ 級別；樹型資結在這些基本操作中，相較於線型資結則有更加均衡的效能表現，使其成為統計上更優秀的常用資料結構。在這之中，二分搜尋樹（往後稱 BST）最基本，特色是在各樹中擁有最低的空間消耗，證明見**藍框**。

BST 在多路搜尋樹中有最小空間浪費率之證明

假設有 n 個節點，表示存有 n 筆資料。對一棵 k 路搜尋樹而言，共 $n*k$ 個連結中，有 $n-1$ 個指標用在資料連結上，其餘則否。

由上計算空間浪費率：

$$\frac{\text{浪費連結數}}{\text{總連結數}} = 1 - \frac{\text{使用連結數}}{\text{總連結數}} = \frac{nk-(n-1)}{nk}$$

當 n 值很大時，忽略常數後得浪費率 $\approx \frac{k-1}{k}$。

又由樹的自然條件 $k \geq 2 \land k \in \mathbb{Z}^+$，經四則運算：

$$k \geq 2 \leftrightarrow 2k \geq k+2 \leftrightarrow 2(k-1) \geq k \leftrightarrow \frac{k-1}{k} \geq \frac{1}{2}$$

可知當 $k=2$ 時有最小值 $\frac{1}{2}$，故二分搜尋樹有最低空間浪費

依照 BST 的規則，由 n 筆資料建構出的 BST，一般操作一筆資料的複雜度是 $O(h)$，h 為樹高，其值詳見下一頁**綠框**推論。又平衡性越好高度越低，因此平衡性對效能有顯著的影響。考慮到 BST 插入近乎有序的資料時易退化成 LinkedList，效能表現上沒有保證，因此針對平衡性的維持，誕生了有名的 AVL 樹和 B 樹系列。

二、 B 樹

如果允許一棵樹擁有不同數量的子節點，那麼理論上可以設計出能自平衡的樹。B 樹就是一種，它藉著分裂與融合，維護絕對平衡性，以下兩點將討論 B 樹的基礎。

(一)、 定義概述

一棵 B 樹由其最大子節點數 m（稱為階，order）決定，對一個 m 階 B 樹而言，所有節點至少存有 m-1 個鍵及 m 個指標，且所有葉節點都在同一層（絕對平衡），各節點遵守大小關係如圖一。

特別的，2-3 樹即 m=3 的 B 樹；2-3-4 樹即 m=4 的 B 樹，因前者有 2 和 3 節點，後者多了一種 4 節點。

BST 樹高 h 的範圍

對一棵有 n 個結點（n_i 個內節點、n_e 個外結點）、高 h 的 BST，若定義起始高度 $h=0$，考慮最極端的兩種情況：

1. 完全退化成鏈結序列
2. 絕對平衡

可得：

$$h \le n_i \le \sum_{x=0}^{h-1} 2^x = 2^h - 1 \quad \wedge \quad 1 \le n_e \le 2^h$$

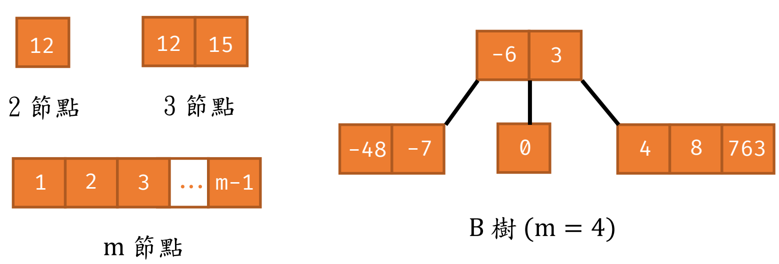
合併以上二式，得：

$$h + 1 \le n \le 2^{h+1} - 1$$

經移項後得：

$$log_2(n+1) - 1 \le h \le n - 1$$

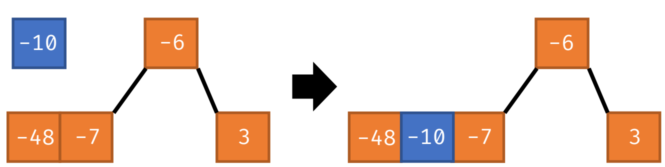
當 n 很大時，平衡樹的樹高以對數級別成長，高度失衡的樹高則是線性增長，差距顯著。



圖一 B 樹與節點示意圖

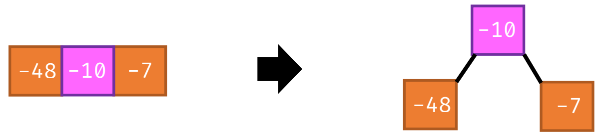
(二)、 新增時的自平衡

無論是新增或刪除，首要步驟都如同 BST：找到正確的位置。新增時，跟 BST 不同的是，新資料會直接加入查找到的節點，如圖二，藍色表示新加入資料。



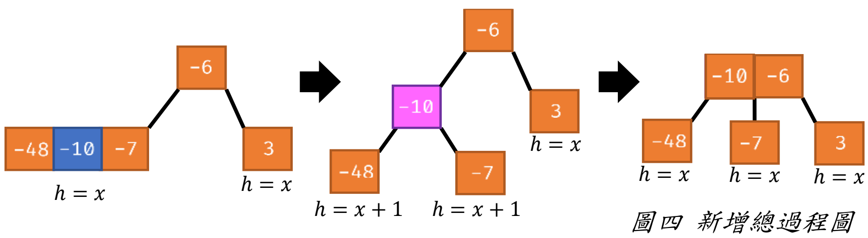
圖二 B 樹 (m=3) 插入新資料

顯然這破壞了 B 樹的階數性質，因此當新臨時節點階數大於定義時，會進行階段性分裂如圖三，粉色表示分裂節點。



圖三 臨時 4 節點分裂

分裂出來的節點會試圖向上融合，成為新的臨時節點，若新臨時節點的階數仍超過定義的階數，就會繼續向上分裂融合，直到成功維持此 B 樹的階數性值，圖四為過程簡圖。



圖四 新增總過程圖

B 樹新增節點時的平衡維持

要討論平衡性的維持，假設：

加入結點處樹高 = 其它處樹高（設為 h）= x

情況分為以下兩種：

1. 無須分裂：葉節點高度不變，故平衡。
2. 遞迴向上分裂：每次分裂時，分裂處枝幹高度 h'加一，而融合時正好相反，h' 減一，如此一來最後一步會有兩種結果如下：
   a. 在某節點融合：$h' = h = x$，故平衡。
   b. 分裂出新的根節點：$h' = h = x + 1$，故平衡。

因此 B 樹維持平衡。

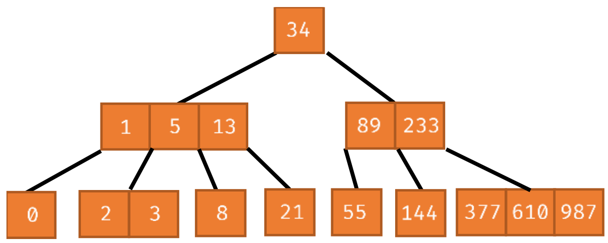
另外刪除時需分成更多情況，在此略過。

由橘框討論，B 樹擁有「絕對平衡」，將給予優秀的效能保證。不過現實是，要實作和維護不同大小的節點絕非易事。若節點選擇以普通陣列實作，方才藍框告訴我們，整棵樹會有巨大的閒置空間；若改以動態線型資結實作，空間消耗也不俗。麻煩的是，以上兩種實現都要面臨不同程度的資料複製、搬移或加上刪除標記而影響效能，可見 B 樹不盡理想。

三、 紅黑樹的根本 － 模擬 B 樹

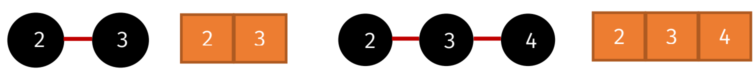
B 樹的設計保證絕對平衡性，理論上效率優異，其中兩個特例：2-3 樹和 2-3-4 樹也同樣擁有。實際上，B 樹共同發明人之一：Rudolf Bayer 教授，在同年（1972 年）發明一個本質上用 2 節點模擬 2-3-4 樹的資結：**紅黑樹**，成為當今最常用的樹型資結。它有著讓初學者困惑的五條定義，我將改從其根源重新描述。

（一）、 紅黑樹的本質：以 2 節點模擬 2-3-4 樹



圖五 2-3-4 樹

對一棵 2-3-4 樹如圖五，將所有 3、4 節點替換成 2 節點的組合，彼此間牽上紅線表示源自同一節點，如圖六：

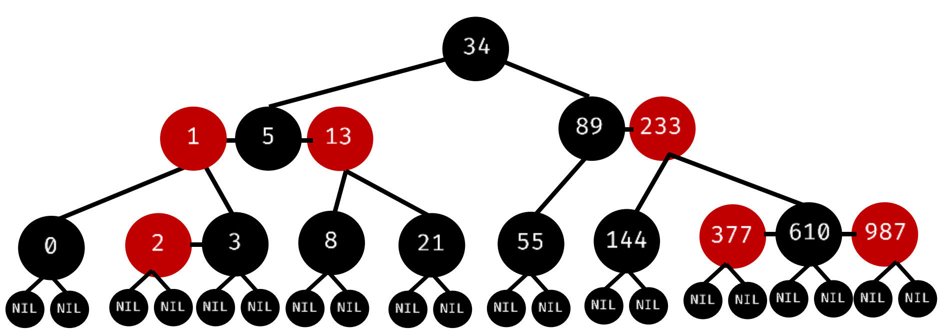


圖六 3、4 節點拆解替換

然後將紅線傳遞給一方，前者傳遞到任意一端皆可，後者則傳給兩端節點，得圖七：

圖七 紅、黑節點的由來

上述變換規定黑色於高位；然後給所有葉節點補上黑色 NIL (null) 節點，上述兩者見後面定義討論。如此一來我們將方才的 2-3-4 樹轉成圖八：



圖八 同構的紅黑樹

由此可知，紅黑樹（之後簡稱 RBT）具備 BST 的特色，簡化實作難度並節省空間，且具有 B 樹的平衡性。B 樹所擁有的「絕對平衡性」，在 RBT 的模擬下，可直覺地看出體現在黑節點上，即「黑平衡」。

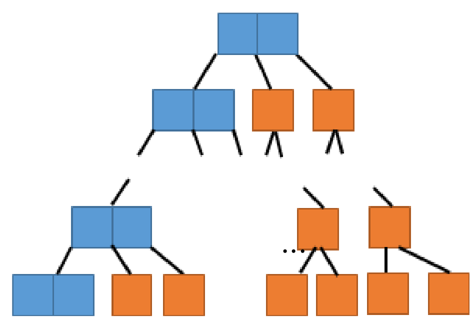
紫框將延續**綠框**，討論 RBT 樹高 h_{rb}。當 n 很大時，h_{rb} 在 $log_2(n) \sim 2log_2(n)$ 之間，高度差最多約為 h_{rb} 的一半，是相當平衡的 BST。

RBT 樹高討論與計算

1. 最矮情況：2-3-4 樹（之後稱「原型」）全由 2 或 4 節點組成，轉成的 RBT 為絕對平衡 BST，故：

$$h = log_2(n+1) - 1\text{，}h\text{ 為原型樹高。}$$

2. 最高情況：見圖例，原型的一條枝幹全以 3 節點構成，其餘為 2 節點。換成 RBT 就是只有一條枝幹以黑紅相間組成，其餘都是黑節點，以原型的視角分批計算如下：
   a. 將 3 節點先看成 2 節點，計算把整棵 2-3-4 樹視為平衡 BST 的資料數：$2^{h+1} - 1$。
   b. 補上每層 3 節點派生出的平衡 BST 資料數：
      $2^0 + \cdots + 2^{h-1} = 2^h - 1$
      $2^0 + \cdots + 2^{h-2} = 2^{h-1} - 1$
      …
      $2^0 = 2^1 - 1$
      $0 = 2^0 - 1$
      等號右邊加總得：$2^{h+1} - h - 2$。
   c. 補上 3 節點相較 2 節點多出的資料：$h + 1$。



圖例 最高 RBT 之等價 2-3-4 樹

將 a、b、c 加總，得總資料數：$n = 2^{h+2} - 2$。

接著將原型樹高 h 轉成 RBT 樹高 h_{rb}，由 3 節點轉為 2 節點組合的行為可推得 $h_{rb} = 2h + 1$。另外，最大高度差正是 $h + 1$（即 3 節點數、RBT 紅節點數）。

最後經四則運算，得 $h_{rb} = 2\,log_2(n+2) - 3$。

（二）、 再看 RBT 的常見定義

1. **節點分紅、黑色**

2. **根節點為黑色**：原型的節點轉成 RBT 節點時，規定黑節點於上位，因此原型根節點換成 RBT 後的根節點必為黑色。

3. **所有葉節點（空節點）為黑**：這使 RBT 為空樹時依然保持根節點為黑色。

4. **紅節點的子節點必為黑色**：一個紅節點的子節點必是原型的其中一種節點，而原型節點轉換後於高位者（對接節點）必為黑色，因此紅節點必然與黑節點接壤。

5. **從任意一個節點縱向移動到葉節點，經過同樣數量的黑節點**：此乃原型的絕對平衡使然。絕對平衡意味著同高的節點深度相同，縱向經過的原型節點數量一致，而每個原型節點換成 RBT 時都會衍生出一個以黑節點為首（在高位）的紅黑節點組合，故有「黑平衡」的性質。

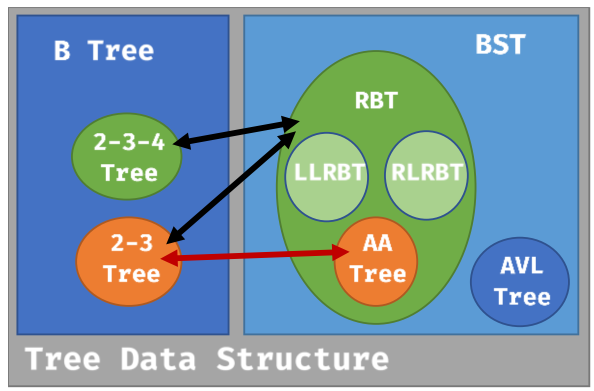
不難發現，一開始黑節點於高位的設計在第二、四、五條定義都起到關鍵作用。意義在於縱向遍歷節點時，標記此次遍歷已經正式進入原型的某節點，因此成為第五定義的基礎，並衍生出第二定義（邊界條件）與第四定義。

（三）、 RBT 家族

RBT 雖然一定程度避免了 B 樹的問題，但增刪時依然會遇到很多情況，為了簡化平衡性調整邏輯，誕生了左、右傾 RBT，分別在 3 節點的轉換選擇左、右傾向。

實際上 RBT 也可以模仿 2-3 樹，這樣的好處在減少平衡性調整時遭遇的情況（搭配左、右傾），實作更簡單。

如果將方才的改變更進一步，使用 AVL 樹高度因子的概念代替 RBT 的顏色標記而引入「黑節點高度因子」，那便是另一個變種：AA 樹（右傾）。



圖九 樹型資料結構簡易文式圖

（四）、 RBT 的比較

RBT 作為高等樹的一種，最常與之比較的便是更早發明出的 AVL 樹。

	RBT	AVL Tree
樹高	$\log n \sim 2\log n$	$\log n$
平衡性	黑平衡	近乎絕對平衡（高度差 ≤ 1）
平衡操作次數	少	多
優勢	增刪	改查

在增刪改查這些操作數量均勻時，RBT 有更好的效率；若需要較多的改查操作，則 AVL 樹的表現會更優，使用時可以依情境選擇最適合的一種。事實上，當插入的資料近乎無序時，純粹的 BST 對比 RBT 和 AVL 樹在效率上是伯仲之間。

四、 討論左傾紅黑樹

以下將以左傾紅黑樹（之後稱為 LLRBT）模擬 2-3 樹為例，討論實作細節。

（一）、 節點宣告與初始化

和 BST 相似，節點以鍵-值對儲存資料，另有兩個子結點指標以及顏色標記，初始顏色為紅色，意義是標記新節點為待融合節點，也因此每次新增後都要將根節點設為黑色。

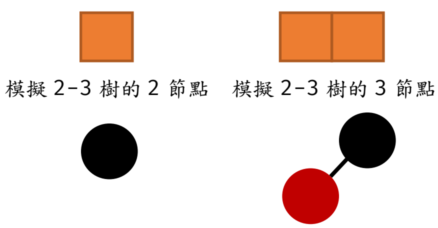
（二）、 改查操作

LLRBT 作為 BST 的一種，修改與查詢操作與 BST 無異。

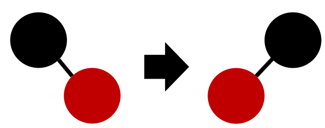
（三）、 新增節點細節討論

除了根節點外，新的紅節點可能加在圖十的兩種目標節點上。

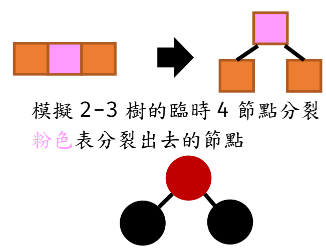
前者可以直接融合。為了符合 LLRBT 定義，若加在右邊就需要「左旋轉」；後者表示融合後會變成臨時 4 節點，如方才橘框圖介紹，需進行分裂，此時應形如圖十二。



圖十　新增前的目標節點

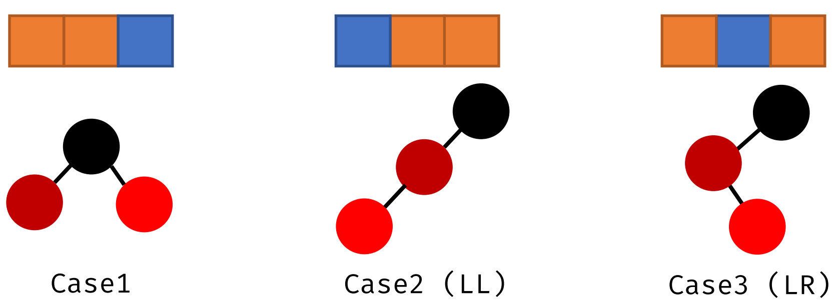


圖十一 2 節點左旋轉



圖十二 完成分裂的 4 節點

於是新的臨時 4 節點會有以下三種情況，注意新加入的紅節點以鮮紅色區別：



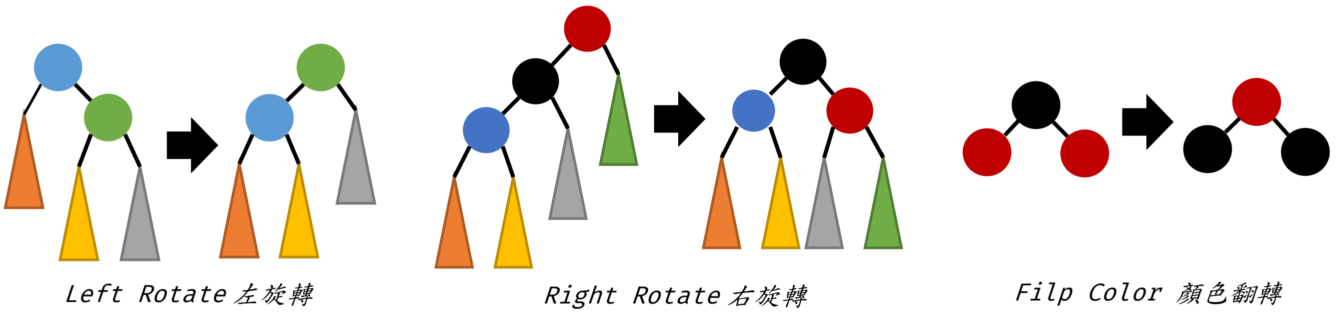
圖十三　可能的臨時 4 節點，藍色表示新節點對應的 2-3 樹節點資料

將上述三種情況分別討論：

1. Case1: 顯而易見，離完成分裂僅一步之遙：「顏色翻轉」。
2. Case2: 可以經「右旋轉」（並更新顏色）變成 Case1。

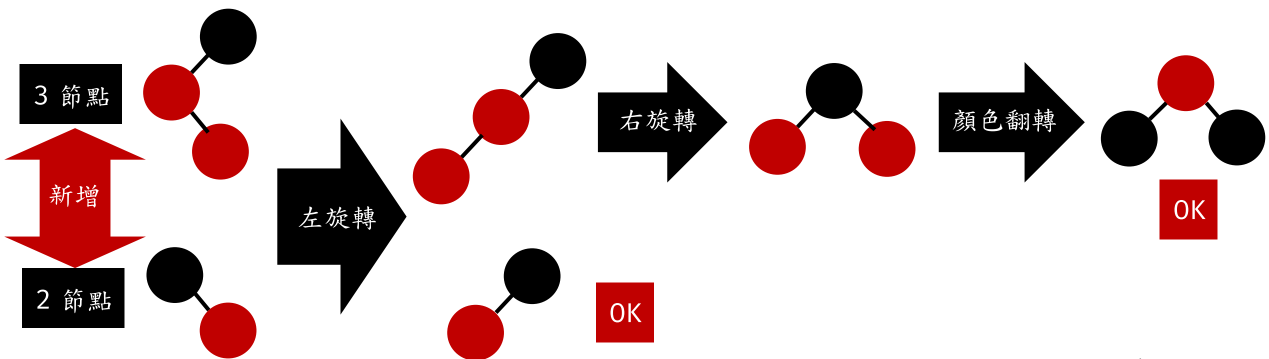
3. Case3: 可對紅節點進行「左旋轉」後變成 Case2。

因此左旋轉、右旋轉與顏色翻轉是重要的中間過程，詳見圖十四。



圖十四 LLRBT 的重要中間過程

應對不同情況的操作，經整理後可以梳理出一條邏輯鏈：



圖十五　新增節點的邏輯鏈

也就是每次新增完節點後，由上而下依序檢查並操作：

- if (IsRed(right.color) and !IsRed(left.color)) then do LeftRotate
- if (IsRed(left.color) and IsRed(left.left.color)) then do RightRotate
- if (IsRed(left.color) and IsRed(right.color)) then do FilpColor

其中 IsRed(node) 函數（見下方虛擬碼）幫助我們判斷一個節點是否為紅色，利用前面談到的定義 3：葉節點（NIL）為黑色，來判斷邊界情況。

```
Function IsRed(node)
    if (node == NIL) then return false // NIL must be black
    return (node.color == RED)
```

如此便能維護 LLRBT 的性質。

刪除操作由於較為繁瑣，故略之。

最後附上我以 Java 實作 LLRBT 增改查的程式碼連結：

https://github.com/Shiritai/Java_Learning_Record/blob/master/Data_Structure/12_RedBlackTree/RedBlackTree.java

參、 結論

本專題總結出一套理解紅黑樹與相關資料結構的知識體系，以此揭示紅黑樹的設計原理。相信在學習不同演算法、資料結構後，去體會、連結並歸納出自己的算法思想，是未來研究、改進和發明抽象事物的關鍵。

肆、 參考資料

一、 Wikipedia

https://en.wikipedia.org/wiki/B-tree

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

https://en.wikipedia.org/wiki/Left-leaning_red%E2%80%93black_tree

https://en.wikipedia.org/wiki/AA_tree\

二、 Others websites

https://github.com/liuyubobobo/Play-with-Data-Structures

https://cloud.tencent.com/developer/article/1734282

三、 Textbook

Introduction to Algorithms

四、 Open Course

http://mirlab.org/jang/courses/dsa/schedule.asp