演算法概論 Exercise #2

Red-black tree 紅黑樹 Report

1. Environment

以 C++為基礎寫的程式(Code blocks)

2. Methods or solution

可以通過範例測資答案正確

```
Insert: 5, 11, 9, 7, 6, 12, 5, 4, 1
key: 1 parent: 4 color: red
key: 4 parent: 5 color: black
key: 5 parent: 6 color: red
key: 5 parent: 5 color: black
key: 6 parent:   color: black
key: 7 parent: 9 color: black
key: 9 parent: 6 color: red
key: 11 parent: 9 color: black
key: 12 parent: 11 color: red
Delete: 11, 5
key: 1 parent: 4 color: black
key: 4 parent: 6 color: red
key: 5 parent: 4 color: black
key: 6 parent:   color: black
key: 7 parent: 9 color: black
key: 9 parent: 6 color: red
key: 12 parent: 9 color: black
Insert: 2, 3
key: 1 parent: 2 color: red
key: 2 parent: 4 color: black
key: 3 parent: 2 color: red
key: 4 parent: 6 color: red
key: 5 parent: 4 color: black
key: 6 parent:   color: black
key: 7 parent: 9 color: black
key: 9 parent: 6 color: red
key: 12 parent: 9 color: black
```

參考資料:

維基百科 https://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91

網路文章 http://alrightchiu.github.io/SecondRound/red-black-tree-introjian-jie.html

為什麼要用 Red Black Tree?

在 BinarySearchTree 中的操作，不論是 Insert 或是 Delete，皆需要先做 Search，效率，取決於 BST 的 height(樹高)，如果一棵樹越矮、越平衡(balanced)，則在此 BST 中搜尋資料的速度較快，理想狀況為 Complete Binary Tree(時間複雜度：O(logN)。反之，若由於輸入資料的順序使得 BST 沒長好、偏一邊，則在此 BST 中搜尋資料的最壞情況將有可能如同在 Linked List 做搜尋(時間複雜度：O(N)。

RBT 可以說是 BST 的進階版，不過 RBT 的 node 比 BST 多加了「顏色」(紅色或黑色)，而正因為多了「顏色」，便能修正 BST 有可能退化成 Linked list 的潛在缺陷。

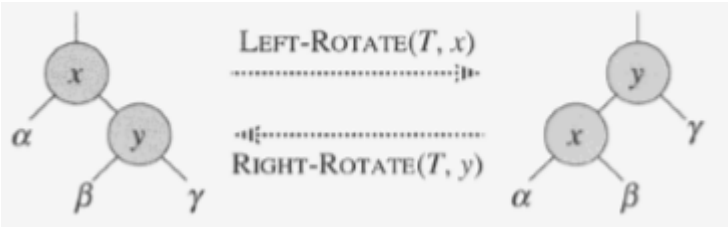
RBT 的條件與特徵

1. RBT 中的每一個 node 不是黑色就是紅色。
2. `root` 一定是黑色。
3. 每一個 leaf node(也就是 `NIL`)一定是黑色。
4. 如果某個 node 是紅色，那麼其兩個 child 必定是黑色，不能有兩個紅色 node 相連。
5. 站在任何一個 node 上，所有從該 node 走到其任意 descendant leaf 的 path 上之黑色 node 數必定相同。

Code:

使用 class 建 rbt 的物件並設立節點包含所需要的資料

Rotation:



Insert:

首先要先考慮是否是第一次加入節點(root)且顏色為黑色

之後要加入的節點先依照數值大小安排要插入的位置，預設填上紅色後檢查

rbt 的顏色正確性，也就是當新增的 node 接在原本存在且為紅色的 parent 時

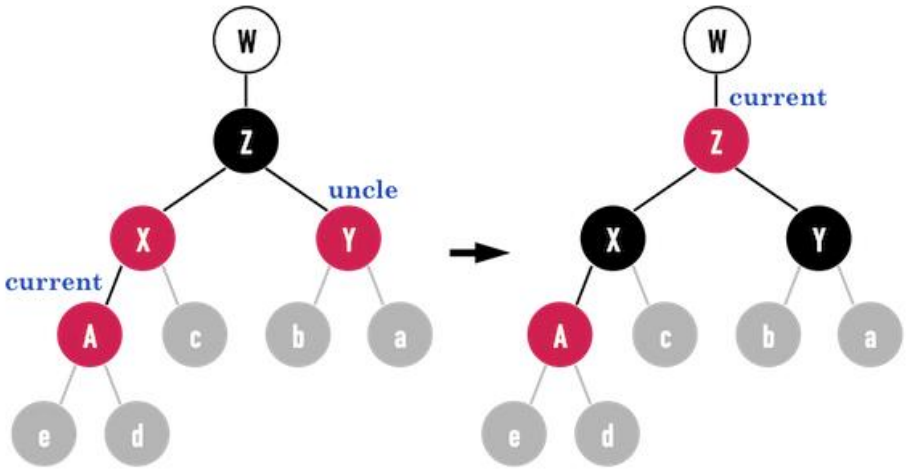
要做修正 (Insert_Fixed)，又區分成 3 種情況

- Case1：`uncle` 是紅色，不論新增的 node 是 node(X)的 `leftchild` 或 `rightchild`
- Case2：`uncle` 是黑色，而且新增的 node 為 node(X)的 `rightchild`
- Case3：`uncle` 是黑色，而且新增的 node 為 node(X)的 `leftchild`

(uncle 是 parent 的 sibling)

按照規則修正顏色後即可實作 rbt 的 insert

EX:



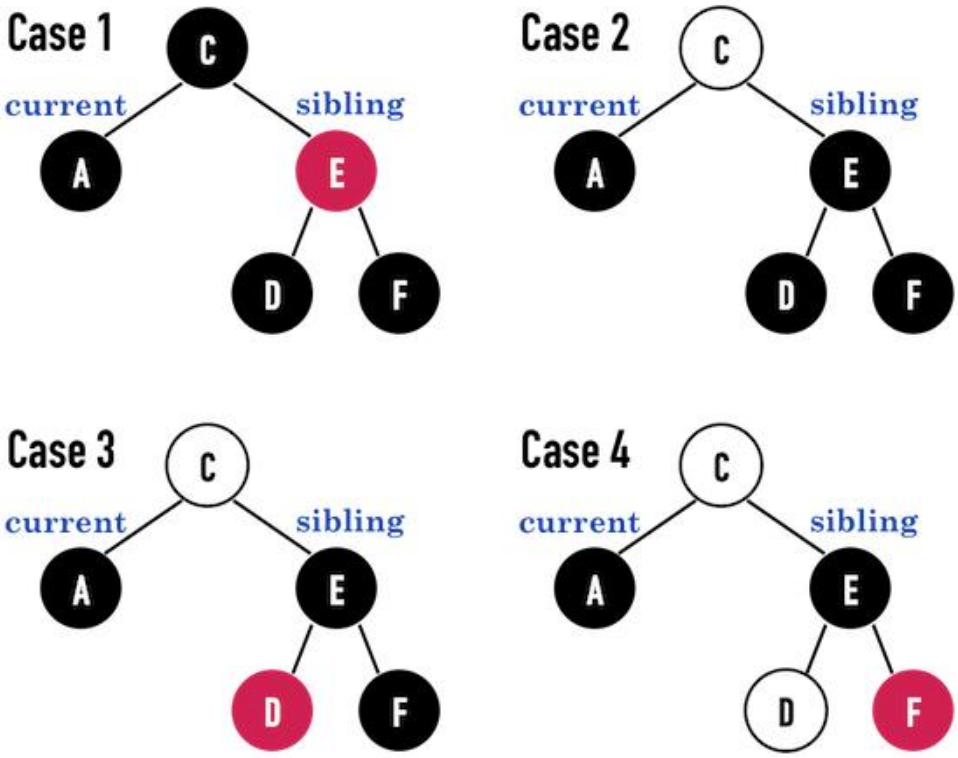
Delete:

類似於 insert，delete 一個 node 也會有使得原本的 rbt 不符合條件的情況，更多情況

刪除之 node 為黑色，有可能違反三點 RBT 特徵，所以又要做修正

1. 刪除的 node 恰好為 root，而刪除後恰好是紅色的 node 遞補成為新的 root，此時便違反 RBT -> root 一定要是黑色；
2. 刪除 node 後，出現紅色與紅色 node 相連
3. 刪除之 node 是黑色，且不是 root，那麼所有包含被刪除 node 的 path 上之黑色 node 數必定會減少，將會違反 RBT 之第五點特徵：

- Case1：`sibling` 為紅色；
- Case2：`sibling` 為黑色，而且 `sibling` 的兩個 `child` 都是黑色；
- Case3：`sibling` 為黑色，而且 `sibling` 的 `rightchild` 是黑色，`leftchild` 是紅色；
- Case4：`sibling` 為黑色，而且 `sibling` 的 `rightchild` 是紅色。



心得:

原本以為這是蠻簡單的作業，想說照著打就不會有問題，後來實際寫 CODE 才其實蠻複雜的，一開始沒有用 class 只用單純的 struct 設定 parent ,left, right 結果寫一寫一直生出奇怪的 node 和多開記憶體 debug de 好久弄不好，後來上網查資料和跟同學討論才換成現在的寫法，中途遇到很多次 pointer 只到 null 的 node 然後程式直接掛掉又不知道問題在哪，一行一行印出來 debug….花了很久的時間尤其是 delete 的部分，越寫越發現一開始沒有考慮到那麼多問題，又重新來了好幾遍，才做出來。一開始也沒想到這個作業會花這麼多時間…