1. **Introduce red-black tree**
   - Explain the algorithm, including search, inserting and deleting
   - Show its advantage, compared with other tree structures such as AVL tree

**Rules of Red-Black Tree**

1. 節點是紅色或黑色
2. 根是黑色
3. 所有葉子都是黑色（葉子是NIL節點）
4. 每個紅色節點必須有兩個子節點並且都是黑色的□葉子到根的路徑不能有兩個連續的紅色節點
5. 從任一節點到其每個葉子的所有簡單路徑都包含相同數目的黑色節點

- **Explain the algorithm, including search, inserting and deleting**

  紅黑樹是一種自平衡二元搜尋樹，常用於實現關聯陣列，又稱為「對稱二叉B樹」；紅黑樹為高效並有著良好的最壞情況執行時間（可在O(log n)時間內做尋找，插入和刪除，n是元素數目）

**(1) Insert node**

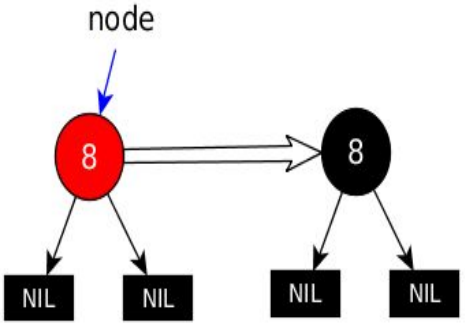
預設插入的節點皆為紅色。

N為插入節點，P為N的父節點，G為P的父節點，S為P的兄弟節點

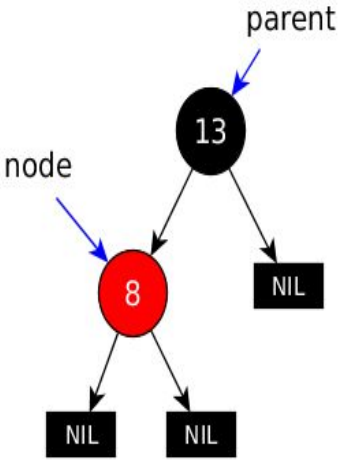
**Case 1：**

如果N是root，由於原樹是NIL，違反性質2，因此直接把此結點塗為黑色（如picture 1）

picture 1




picture 2



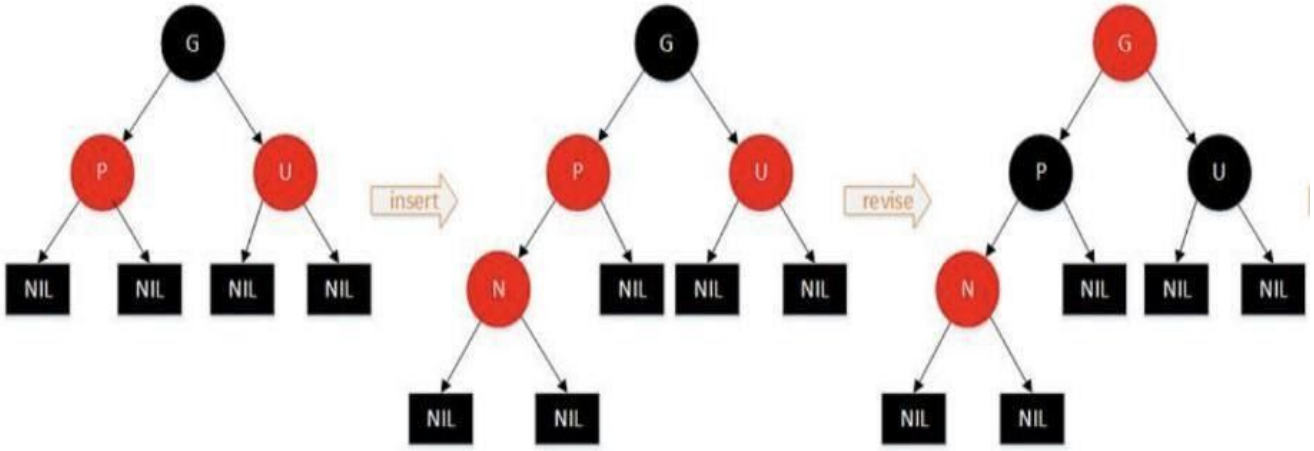

**Case 2：**

當P為黑色時，N為紅色而路徑上只多一個紅色節點，因此滿足性質（滿足性質4，5）（如picture 2）

**Case 3：**

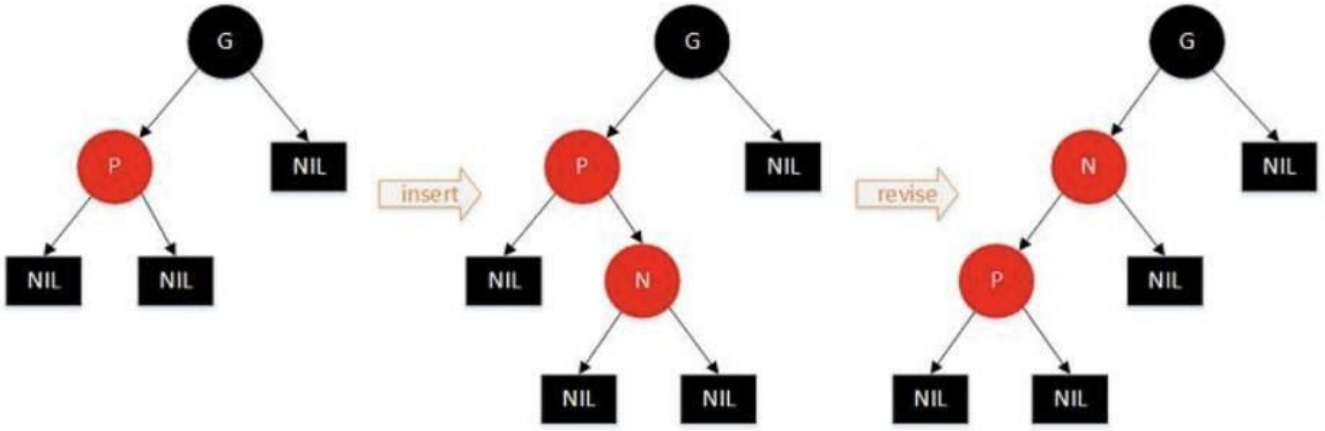
當P和U皆為紅色，N加進來也為紅色，因此將P和U設為黑色，再將G設為紅色，但

此時G可能為root或是G的父節點為紅色，因此G要重新分析

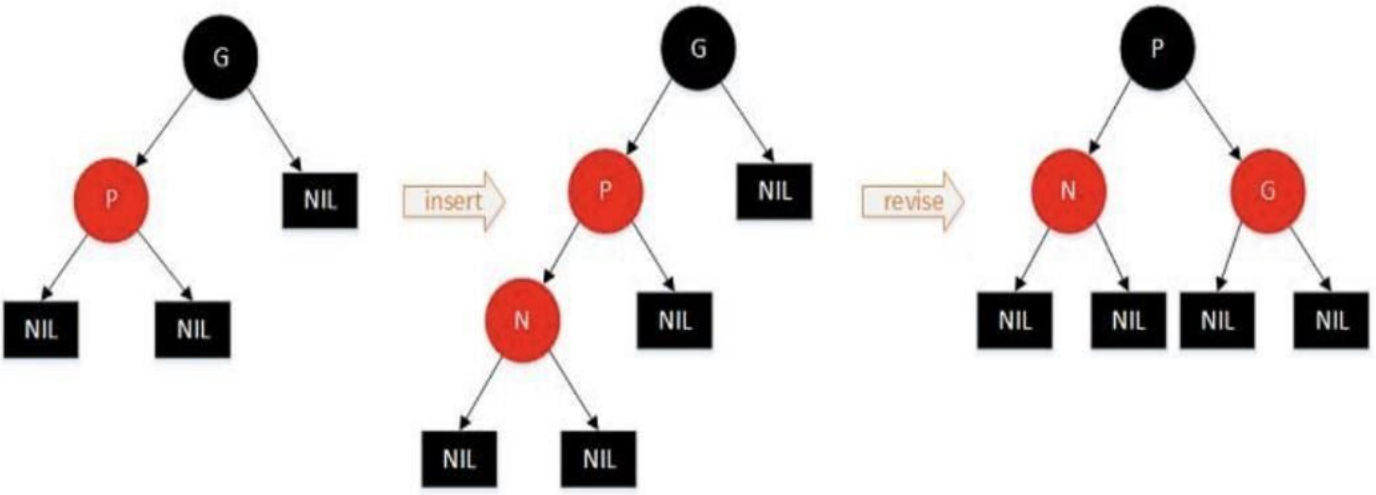


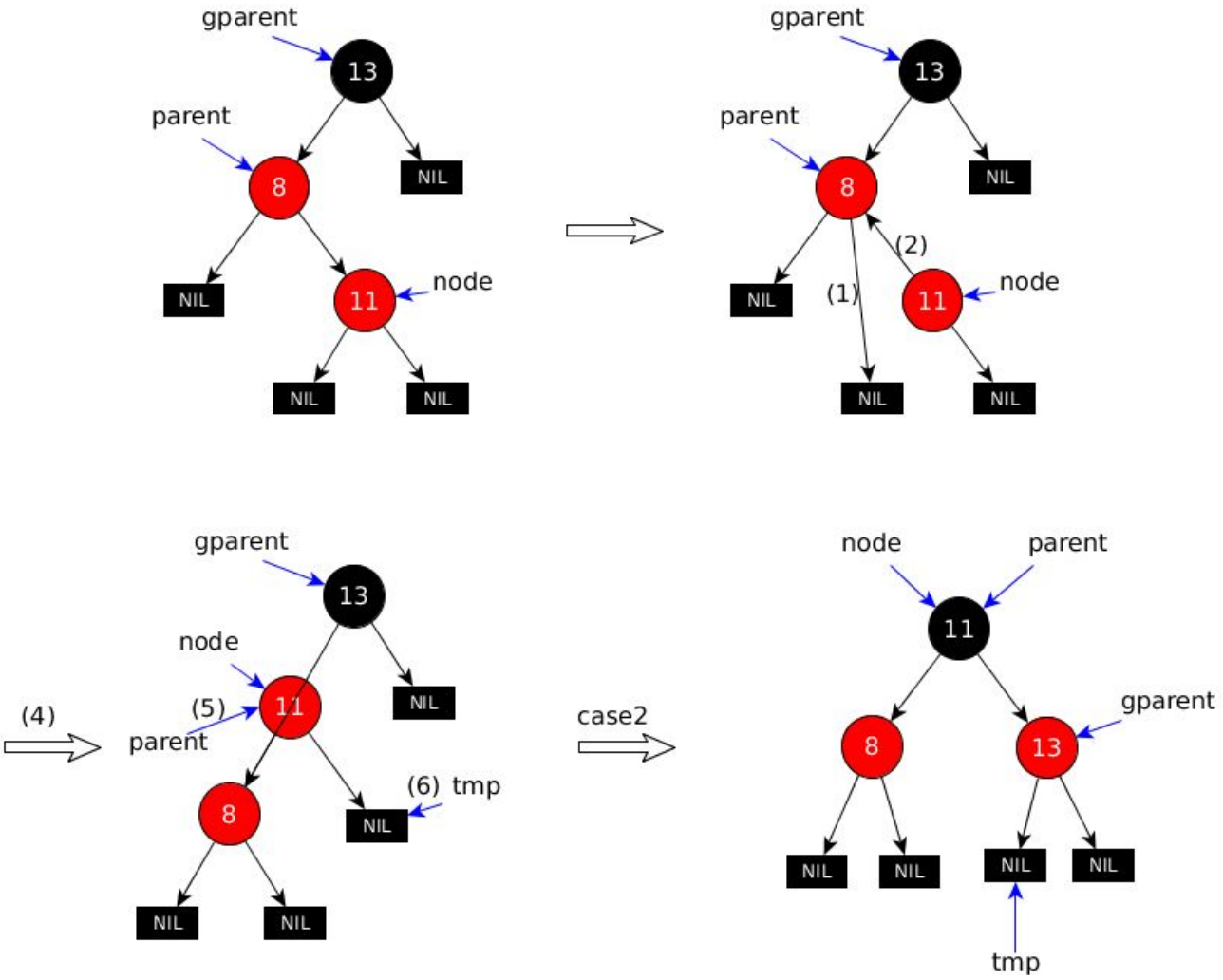

**Case 4：**

P是G的左節點，若P為紅色U是黑色或NIL，而N是P的右節點（如下圖中間），此時對P及N做左旋，但此時P及N都是紅色。（若P是G的右節點，其餘性質不變，則對P及N做右旋）




前述相同，當N為P的左節點時，對P及G做右旋轉，因P為紅色G必為黑色，將P及G的顏色互換。（若P是G的右節點，其餘性質不變，則對P及G做左旋轉）






**(2) Delete node**

刪除的節點D的子節點個數可分成三種（P是D的父節點，B是D的兄弟節點）

紅黑樹是一種特殊的二叉查找樹，其刪除結點要按二叉查找樹刪除結點的演算法進行

普通二叉查找樹刪除一個結點：

A. 待刪除結點沒有子結點，即它是一個葉結點，此時直接刪除
B. 待刪除結點只有一個子結點，則可以直接刪除；如果待刪除結點是根結點，則它的子結點變為根結點；如果待刪除結點不是根結點，則用它的子結點替代它
C. 待刪除結點有兩個子結點，首先找出該結點的右子樹中數值最小的那個結點，將兩個結點進行值交換

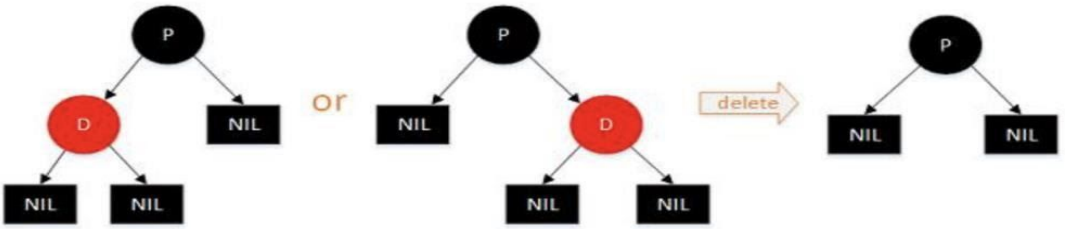
紅黑樹的刪除結點演算法

A. 直接把該結點調整為葉結點，刪除葉結點（沒有子結點）
   - 若該結點是紅色，則可直接刪除，不影響紅黑樹的性質，演算法結束
   - 若該結點是黑色，則刪除後紅黑樹不平衡，此時要進行調整
B. 刪除結點有一個外部結點
C. 刪除結點有兩個外部結點

**1. D沒有子節點**

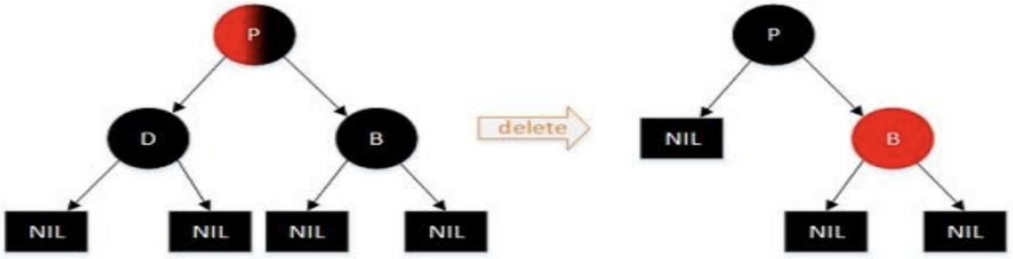
**Case 1：**

若D為紅色，則直接刪除



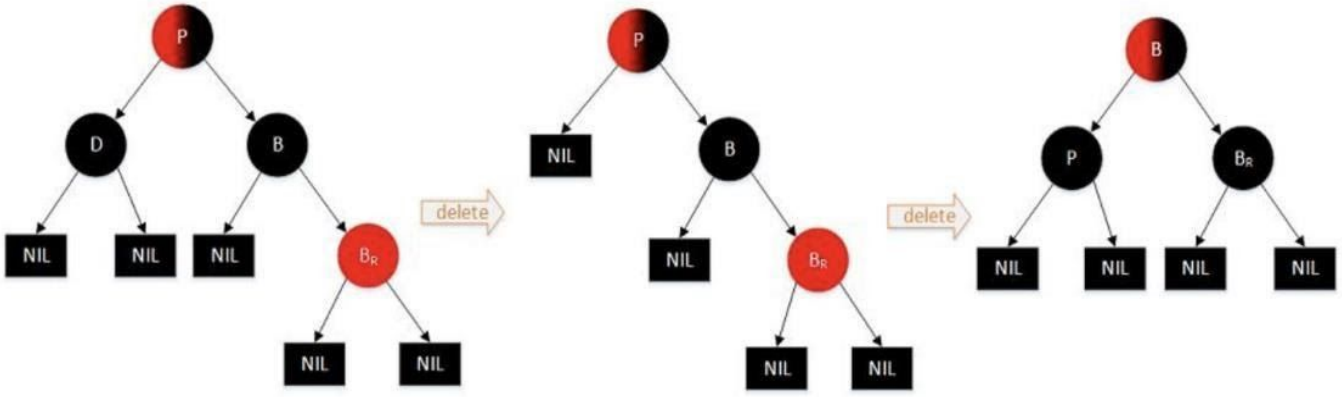

**Case 2：**

若D為黑色，B沒有子節點，把P設為黑色，B設為紅色

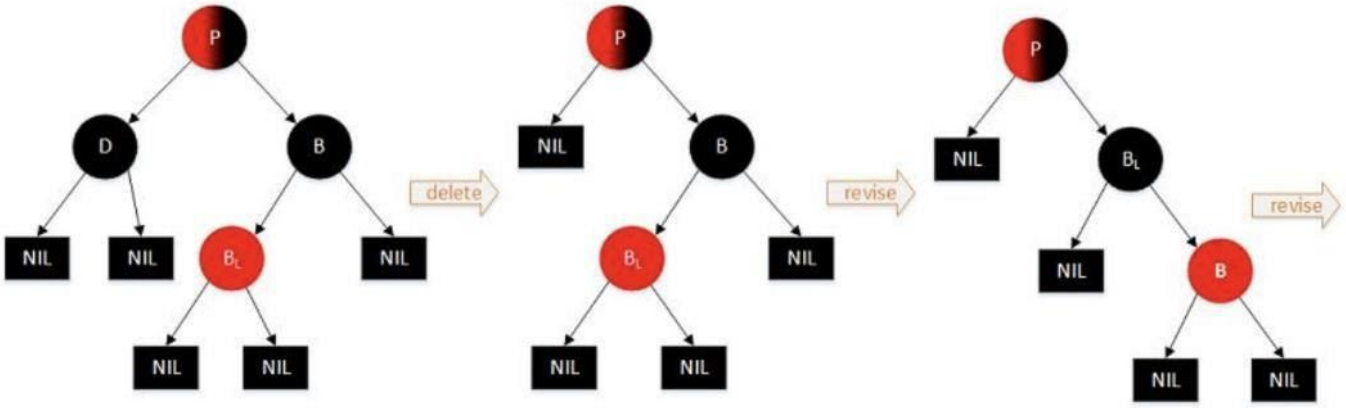



**Case3：**

若D為P的左節點，D為黑色，有一個B的右節節點不為NIL，把B的右節點改為黑色，B改為P原來的顏色，P變黑色，然後對P及B做左旋轉；若D為P的右節點，其餘性質不變，則對P及B做右旋轉

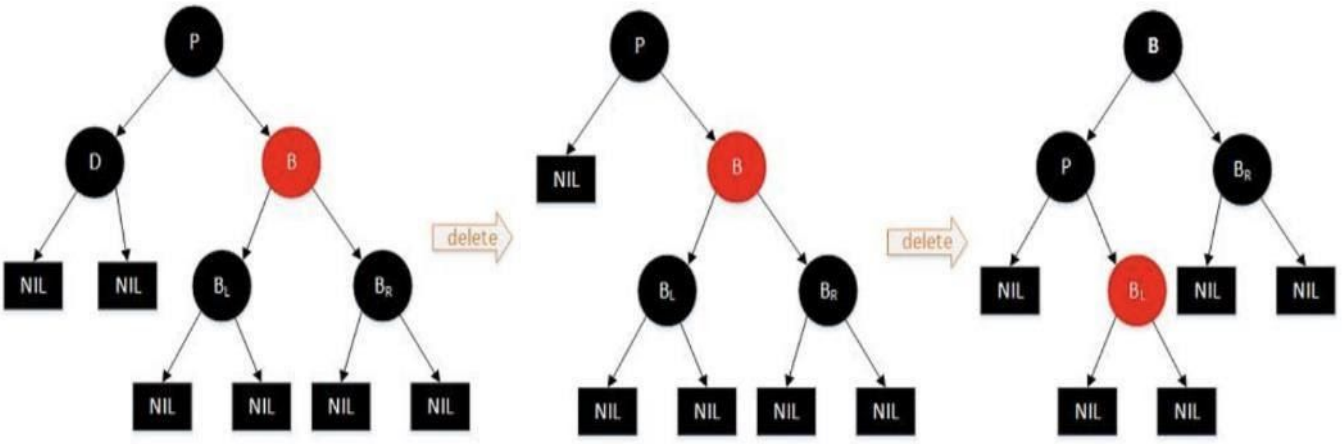



D為P的左節點且D為黑色，B的左節點不為NIL則改為黑色，B改為紅色，其他依照前面繼續判斷

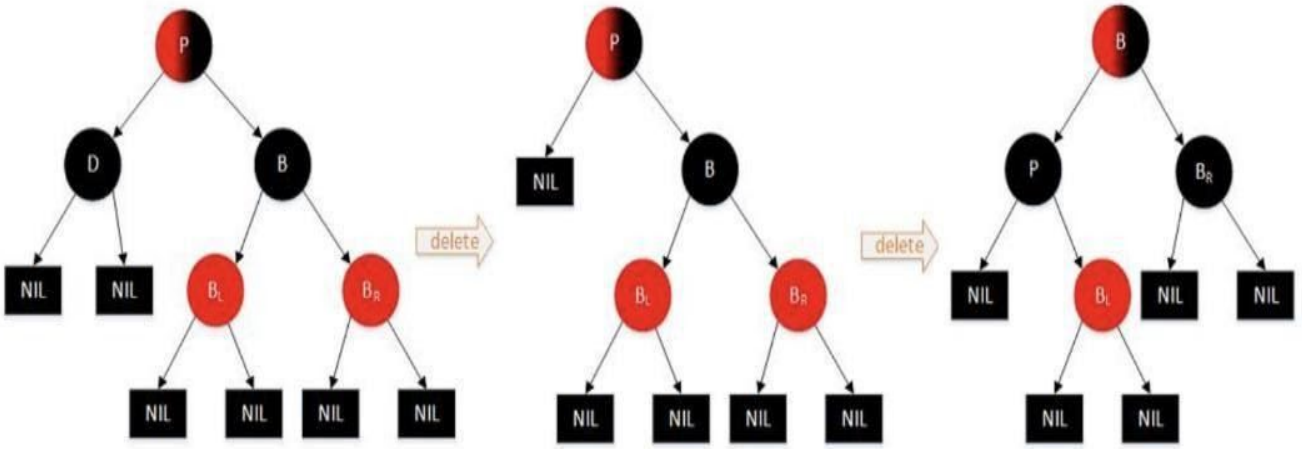


**Case 4：**

D為P的左節點且D為黑色，B有兩個孩子，若B紅色則B的兩個孩子必為黑色，將B改為黑色，B的左節點改為紅色，對P及B做左旋轉；若D為P的右節點，則對P及B做右旋轉




D為P的左節點且D為黑色，B有兩個孩子，若B黑色則B的兩個孩子必為紅色，將P改為黑色，B的右節點改為黑色，B改為P原來的顏色，對P及B做左旋轉；若D為P的右節點，則對P及B做右旋轉

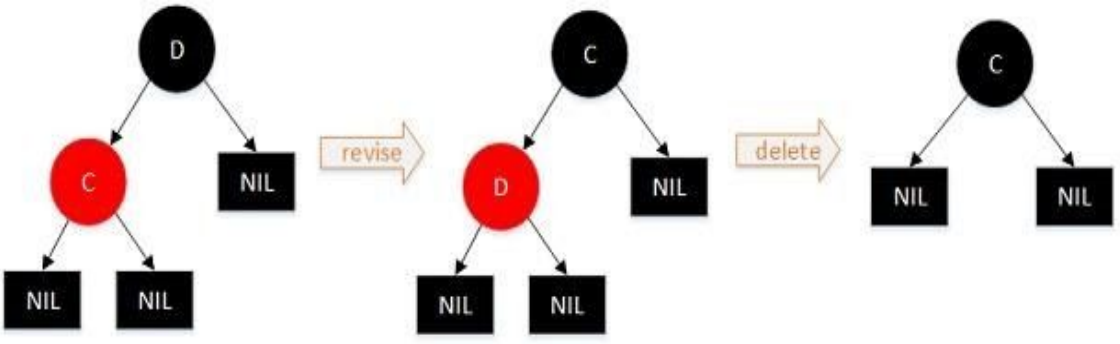



2. **D有一個子節點**

   D刪除後，這個結點必為紅色節點，交換D和子結點，被刪除節點變為子結點，並改為黑色

3. **D有兩個子節點**

   D有兩個子節點且必為紅色，直接用左子樹的最大節點或右子樹的最小節點取代D，並將其節點刪除，直到被刪除節點的兩個孩子都為NIL



**(3) Search node**

紅黑樹屬於二元搜尋樹，因此操作與普通二元搜尋樹相同。

1. 若b是空樹，則搜索失敗
2. 若x等於b的根節點的資料欄之值，則查找成功
3. 若x小於b的根節點的資料欄之值，則搜索左子樹
4. 若x大於b的根節點的資料欄之值，查找右子樹
5. 一層一層找，重複3、4，直到查找成功或失敗

- **Show its advantage, compared with other tree structures such as AVL tree**

A. 紅黑樹只要求部分平衡，因此降低對旋轉的要求而提高性能
B. 紅黑樹能夠以O(log n)的時間複雜度進行搜索、插入、刪除操作
C. 紅黑樹的設計能讓不平衡在三次旋轉之內解決

相比於BST，BST時間複雜度的最壞情況可以達到O(n)，而紅黑樹在最壞的情況下也可以保證O(log n)，因為紅黑樹可以確保樹的最長路徑不大於兩倍的最短路徑，因此比BST好。

相比於AVL，紅黑樹的時間複雜度和AVL相同，其兩者的查找效率皆為O(log n)，但AVL統計性能較紅黑樹弱，所以AVL在操作中所做的後期維護比紅黑樹更耗時，因此紅黑樹的應用效能仍是高於AVL；然而實際上插入速度取決於資料，如果數據為隨機產生系列數（分布較好），則比較適合採用AVL，但是如果想處理比較雜亂的情況，則紅黑樹會較為快速。

A. AVL trees are more rigidly balanced and hence provide faster look-ups. Thus for a look-up intensive task use an AVL tree
B. For an insert intensive tasks, use a Red-Black tree
C. AVL trees store the balance factor at each node. However, if we know that the keys that will be inserted in the tree will always be greater than zero, we can use the sign bit of the keys to store the colour information of a red-black tree
D. In general, the rotations for an AVL tree are harder to implement and debug than that for a Red-Black tree

**2. Reference：**


https://blog.csdn.net/cwcmcw/article/details/17242261
https://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91
https://zh.wikipedia.org/wiki/AVL%E6%A0%91
https://blog.csdn.net/hushujian/article/details/39778063
https://blog.csdn.net/jdbc/article/details/42846803
https://read01.com/4aDeRd.html
https://stackoverflow.com/questions/13852870/red-black-tree-over-avl-tree
https://blog.csdn.net/cwcmcw/article/details/17174891
https://en.wikipedia.org/wiki/Red%E2%80%93black_tree