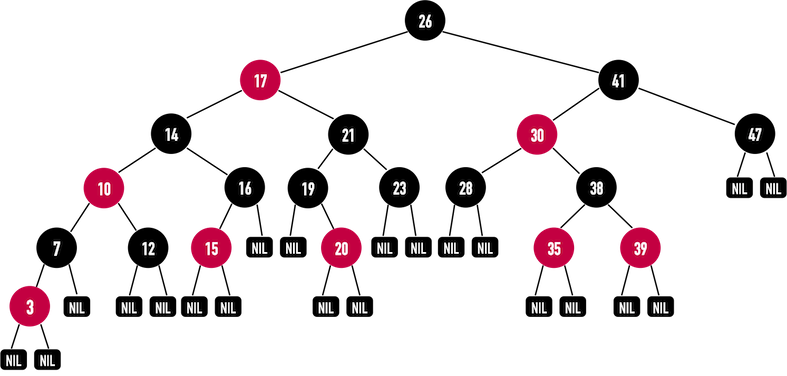
Red Black Tree的特徵

Red Black Tree(RBT)是node塗了「顏色」的Binary Search Tree(BST)，藉由控制顏色，能夠保證在RBT中，**最長path(路徑)不會超過最短path的兩倍**(若最短的path是5最長的path至多只能是10，如此，RBT便能夠近似地視為平衡，如圖四。



**圖四：最短的path為**3 **(最右path:26-41-47)，  
其餘path最長只能是**6 **(最左path:26-17-14-10-7-3)。  
若蓋住NIL與顏色，此即為BST。**

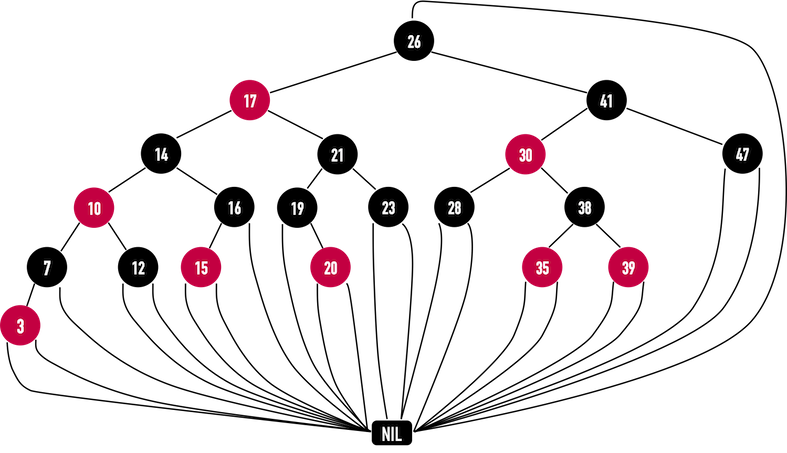
圖四中，所有原本在BST中指向NULL的pointer，在RBT中，全部指向了NIL。什麼是NIL？NIL是永遠為黑色、並且實際占有記憶體的node，因為有配置記憶體，因此能夠以Node->color的方式取得某個node之顏色(若使用NULL則無法)，這項設計將在後續介紹如何於RBT中Insert(新增資料)與Delete(刪除資料)時派上用場。

接著來看RBT的五項特徵：

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，如圖四中的node(17)、node(30)。
   * 若某個node為黑色，其child之顏色沒有限制，如圖四中的node(38)、node(26)、node(21)。
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。以圖四為例，站在node(14)上，所有從node(14)走向其descendant leaves(也就是NIL)的path上之黑色node數必為3：
   * path1:14(b)-10(r)-7(b)-3(r)-NIL(b)；
   * path2:14(b)-10(r)-7(b)-NIL(b)；
   * path3:14(b)-10(r)-12(b)-NIL(b)；
   * path4:14(b)-16(b)-15(r)-NIL(b)；
   * path5:14(b)-16(b)-NIL(b)；

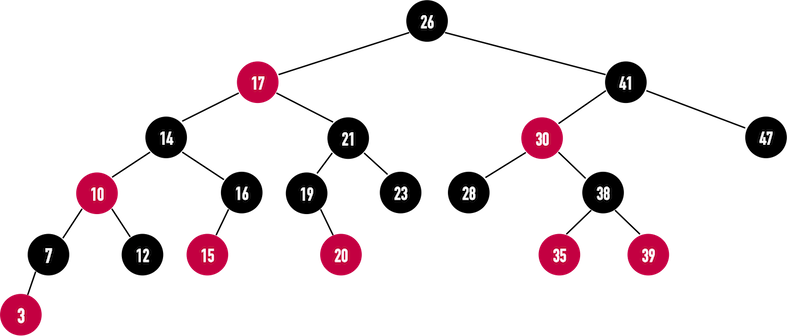
根據上述特徵的第四點與第五點，RBT中path可能的長度最小值一定是全部node皆為黑色(如圖四最右path)，而path可能的長度最大值並定是紅色-黑色相間(如圖四最左path)，如此便確保RBT擁有**最長path(路徑)不會超過最短path的兩倍**的特性。

實際的程式實現上，會把所有NIL視為同一個NIL，並把root的parent指向NIL，以節省記憶體空間，如圖五。



**圖五：。**

為求畫面簡潔，往後的篇幅裡將把RBT示意圖中的NIL隱藏起來，只顯示RBT中的internal node，如圖六，不過心裡要記得，RBT無時無刻都被NIL充滿著。



**圖六：。**

Red Black Tree: Rotation(旋轉)

於RBT中進行Insert與Delete後竟然不符合RBT

現考慮如圖一(a)之RBT，欲進行Insert(新增資料)與Delete(刪除資料)：



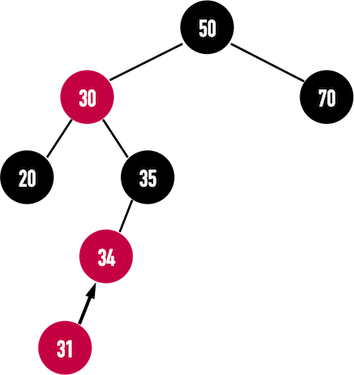
**圖一(a)：一棵符合規則的RBT。**

為滿足RBT之特徵第五點：從root走向任何一個leaf node(NIL)的任何一條path上之黑色node數皆相同，一般在RBT中Insert(新增資料)時，會先將欲新增的node塗成紅色。  
若幸運地，新增的node接在黑色node上，則RBT仍然是一棵RBT，如圖一(b)，新增node(15)。



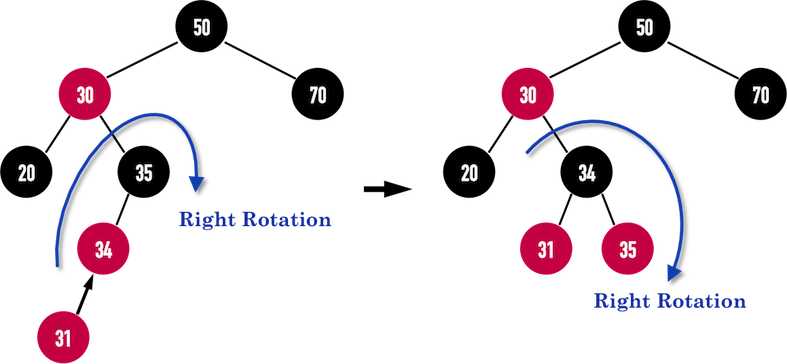
**圖一(b)：在RBT中新增node(15)，新增完後仍然滿足RBT之規則。**

但是，若要在RBT中新增node(31)，因為node(34)是紅色，此處便會出現紅色與紅色相連的情況，違反RBT之第四點特徵，因此需要修正，如圖一(c)。



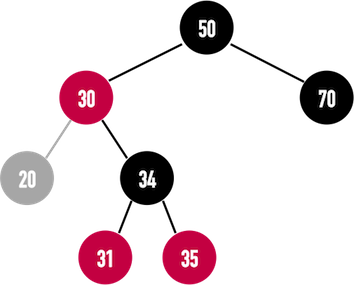
**圖一(c)：在RBT中新增node(31)，新增完後無法滿足RBT之規則。**

修正的方法便是將node(35)塗成紅色，node(34)塗成黑色，並且對**node(35)**進行**Right Rotation(向右旋轉)**，如此一來，RBT便能維持其基本特徵。  
值得注意的是，在執行Rotation時，有時會順便調整RBT的height(樹高)，使得RBT維持在平衡(balanced)的狀態。



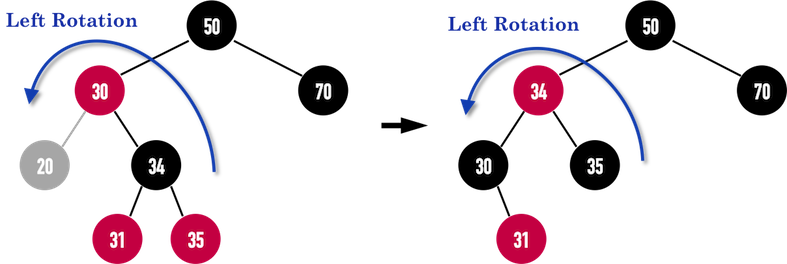
**圖一(d)：使用**Rotation**對Insert(新增資料)進行修正。**

在正確新增node(31)後，若要刪除node(20)，因為node(20)是黑色，若將其刪除則違反RBT之第五點特徵：從root走向任何一個leaf node(NIL)的任何一條path上之黑色node數皆相同，因此同樣需要進行修正。



**圖一(e)：因為node(20)是黑色，若將其刪除會違反RBT之第五點特徵。**

修正的方法則是將node(30)塗黑，node(34)塗紅，然後對**node(34)**進行**Left Rotation(向左旋轉)**，如此一來，RBT便能維持其基本特徵。



**圖一(f)：使用**Rotation**對Delete(刪除資料)進行修正。**

經過以上說明，應該能體會到Rotation在修正Insert(新增資料)與Delete(刪除資料)時的威力，接著便來實地走訪Rotation實際上是哪些pointer在指來指去。

Red Black Tree: Insert(新增資料)與Fixup(修正)

## 如同於BST中Insert(新增資料)

需要修改/擴充的部分有三處：

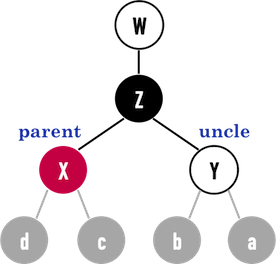
1. NIL：所有原先在BST中指向NULL的pointer，在RBT中需要修正成指向NIL。
2. 顏色：如同在[RBT：Rotation(旋轉)](http://alrightchiu.github.io/SecondRound/red-black-tree-rotationxuan-zhuan.html#fail)所說，一般預設新增node為紅色，因此，若新增node接在黑色node之後，仍能滿足RBT的特徵。但是若新增node接在紅色node之後，則違反了[RBT之第四點特徵](http://alrightchiu.github.io/SecondRound/red-black-tree-introjian-jie.html" \l "rbt)，必須進行修正。
3. 進行修正。

## 修正

什麼情況需要對RBT做修正？當新增node接在紅色的node的child pointer，形成紅色與紅色相連時。

考慮以下情況，如圖一(a)，新增的node將要接在node(X)上：

* node(X)為其parent，顏色為紅色；
* node(Y)為其uncle，其顏色**可能為紅色或黑色**。
* node(Z)為其grandparent，顏色必定為黑色(因為node(X)是紅色)；
* node(W)的顏色**可能是紅色或黑色**。
* 所有灰色node(如node(a)、node(b)、node(c)、node(d))表示：只要不影響RBT之特徵，其顏色、是否實際攜帶資料或是否為NIL並不影響結果。



**圖一(a)：。**

根據uncle的顏色是紅色或者黑色，可以將修正(FixUp)分成三種情形(case)：

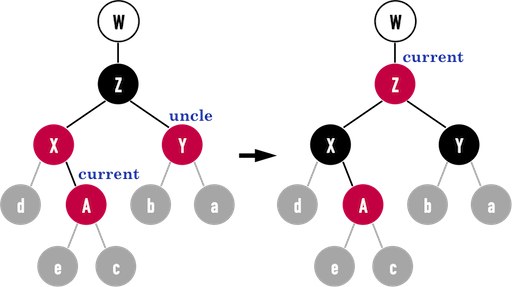
1. Case1：uncle是紅色，不論新增的node是node(X)的leftchild或rightchild；
2. Case2：uncle是黑色，而且新增的node為node(X)的rightchild；
3. Case3：uncle是黑色，而且新增的node為node(X)的leftchild。

### Case1

圖一(b)左，此時current指向新增的node(A)，而node(A)成為node(X)的rightchild，其uncle:node(Y)是紅色的。  
修正的方法就是「把債還給上一代的上一代」：

* 將parent塗成黑色：node(X)塗成黑色；
* 將uncle塗成黑色：node(Y)塗成黑色；
* 將parent->parent塗成紅色：node(Z)塗成紅色：
* 將current從node(A)移到node(Z)。

此時，如圖一(b)右，從node(Z)出發往其descendant leaves的任一path上之黑色node數皆相同，這個subtree便滿足了RBT的特徵。

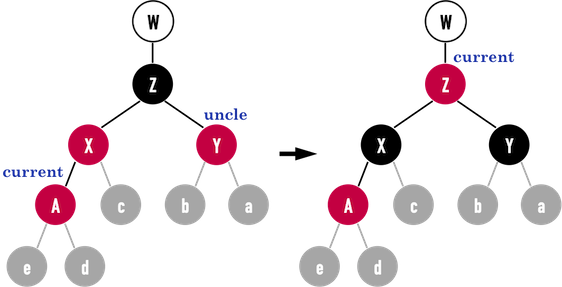


**圖一(b)：。**

接著必需根據node(W)的顏色採取不同行動：

* 若node(W)為黑色，就不需要再做調整；
* 若node(W)為紅色，則node(Z)與node(W)再次形成紅色node與紅色node相連，必須重複同樣的判斷流程。

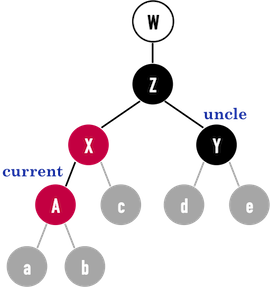
若node(A)成為node(X)的leftchild，如圖一(c)，修正的方法同上。



**圖一(c)：。**

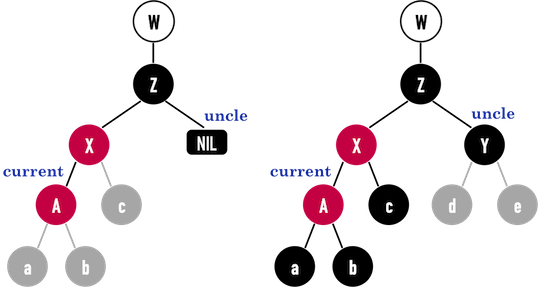
### Case3

圖一(d)，新增的node(A)成為node(X)的leftchild，其uncle:node(Y)是黑色。



**圖一(d)：。**

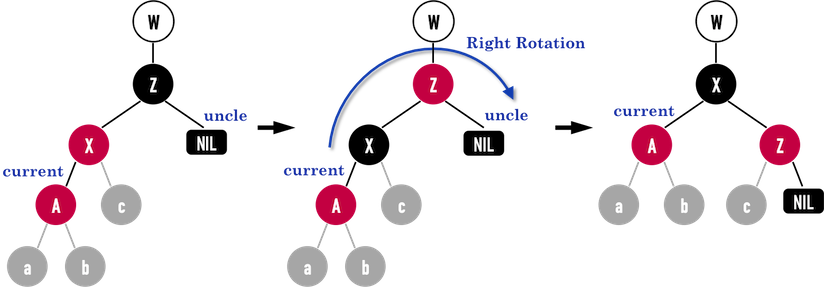
事實上，若current指向之node(此為node(A))是新增的node，則根據RBT之第五點特徵，其uncle:node(Y)必定是NIL，如圖一(e)左。不過，在稍後的範例中將會看到，current不一定是「剛剛新增的node」，也有可能是「修正到一半，出現紅色與紅色相連的node」，但因為是「修正到一半」，尚未調整node(Z)的顏色，因此所有從node(Z)往其descendant leaves的任意path上之黑色node數必定不變，此時，若node(Y)不為NIL，則node(X)以及node(A)必定還有黑色的child pointer，如圖一(e)右所示，node(a)、node(b)與node(c)皆為黑色。



**圖一(e)：。**

修正方法如下，見圖一(f)：

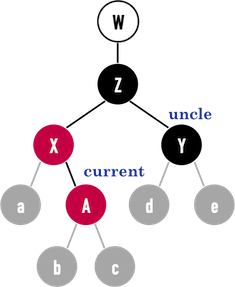
* 將parent塗成黑色：node(X)塗成黑色；
* 將parent->parent塗成紅色：node(Z)塗成紅色；
* 對parent->parentnode(Z)進行**Right Rotation(向右旋轉)**。



**圖一(f)：。**

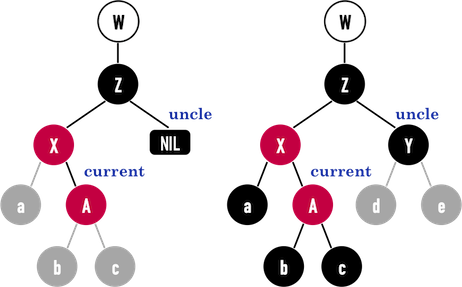
### Case2

圖一(h)，新增的node(A)成為node(X)的rightchild，其uncle:node(Y)是黑色。



**圖一(h)：。**

如同Case3，圖一(h)的uncle:node(Y)同樣有兩種可能：攜帶實際資料的黑色node，或者NIL，如圖一(i)：

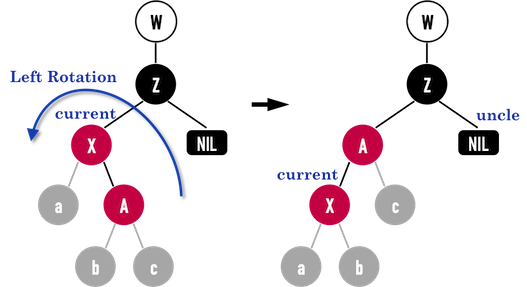


**圖一(i)：。**

而修正Case2的方法就是將其轉換成Case3，再利用上述Case3的方法調整成正確的RBT。  
從Case2調整成Case3，如圖一(j)：

* 將current移至current->parent：將current從node(A)移到node(X)；
* 對新的current進行Left Rotation：對node(X)進行Left Rotation。

圖一(j)右符合Case3：「current成為其parent的leftchild，且其unclenode(Y)是黑色」，因此，只要再進行如同圖一(f)之修正流程即可。

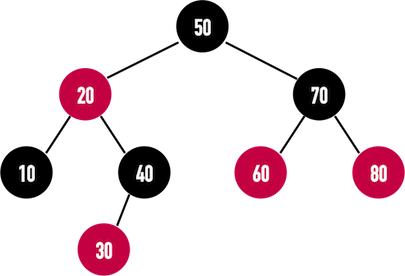


**圖一(j)：。**

### 幾個範例

**Example1**

考慮一棵RBT如圖二(a)：



**圖二(a)：。**

若想新增node(75)，由於其將接在node(80)的leftchild位置上，而node(80)為紅色，因此需要進行修正。  
接著觀察，node(75)之uncle:node(60)，是紅色，因此可以使用Case1的方法修正，如圖二(b)：

* 將parent與uncle塗黑：node(80)與node(60)塗黑；
* 將grandparent塗紅：node(70)塗紅；
* 將current移至node(70)；
* **進入下一個迴圈**判斷node(70)是否與其parent形成紅色與紅色node相連。

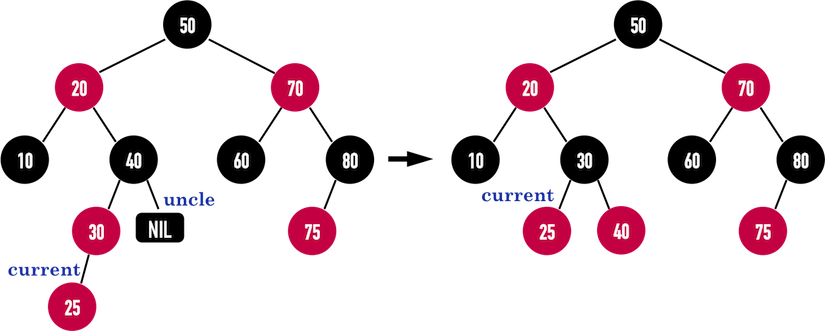
恰好，node(50)為root，一定是黑色，因此新增node(75)便算是完成。



**圖二(b)：。**

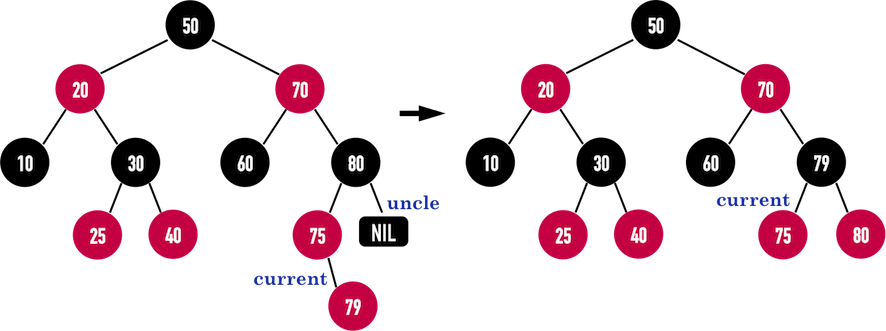
若想繼續新增node(25)，由於其將接在node(30)的leftchild，而node(30)為紅色，因此需要修正。  
接著觀察，node(25)之uncle為NIL是黑色，而node(25)本身是leftchild，因此可以使用Case3的方法，如圖二(c)：

* 將parent塗成黑色：node(30)塗成黑色；
* 將parent->parent塗成紅色：node(40)塗成紅色；
* 對parent->parent:node(40)進行**Right Rotation(向右旋轉)**。



**圖二(c)：。**

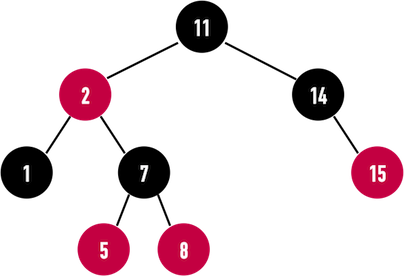
若想繼續新增node(79)，由於其將接在node(75)的rightchild，而node(75)為紅色，因此需要修正。  
接著觀察，node(79)之uncle為NIL是黑色，而node(79)本身是rightchild，因此可以使用Case2的方法，先將問題從Case2轉換成Case3，再由Case3之方法修正，如圖二(d)：



**圖二(d)：。**

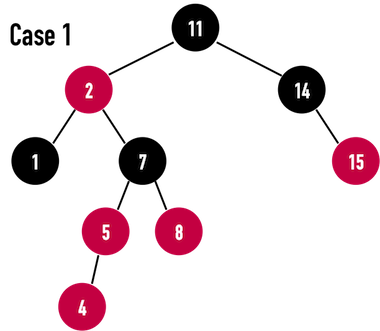
**Example2**

考慮一棵RBT如圖三(a)：

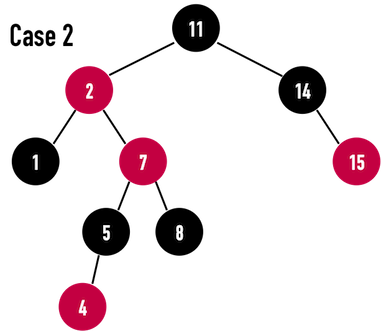


**圖三(a)：。**

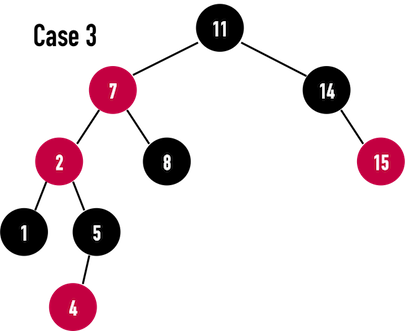
要在其中新增node(4)，則會依序經歷case1、case2直到case3完成修正，如圖三(b)、圖三(c)、圖三(d)與圖三(e)所示。



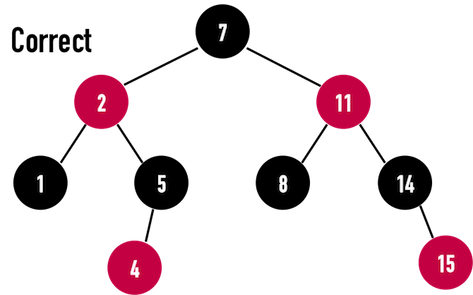
**圖三(b)：。**



**圖三(c)：。**



**圖三(d)：。**



**圖三(e)：。**

根據以上說明，可以歸納出對於RBT的修正(Fix-Up)之情形(Case)間的循環圖，如圖四：

* 當新增node之parent為紅色時，需要對RBT進行修正；
* 若進入Case1，有可能執行一次即完成，也有可能再次出現紅色與紅色相連的情況，如圖三(b)-(c)；
* 若進入Case2，就轉換成Case3的情境；
* 一旦進入Case3，經過修正後必定能滿足RBT之特徵限制。

資料來源：

1. http://alrightchiu.github.io/SecondRound/red-black-tree-introjian-jie.html
2. http://alrightchiu.github.io/SecondRound/red-black-tree-rotationxuan-zhuan.html
3. http://alrightchiu.github.io/SecondRound/red-black-tree-insertxin-zeng-zi-liao-yu-fixupxiu-zheng.html