F74109016_hw24 葉惟欣

AVL tree:

AVL tree 的高度為 log(N) **證明:** 當一個節點高度為 1 兩個節點高度為 2

Recurrent formula :當有兩個節點以上有 n 個節點，高度 h 的 AVL tree 包含一個根結點，其中個 subtree 的高度為 n-1 另一個為 n-2 $n(h) = n(h-1) + n(h-2) + 1$

$n(h-1) > n(h-2)$

➔$n(h) > n(h-2) + n(h-2) + 1 > 2n(h-2)$

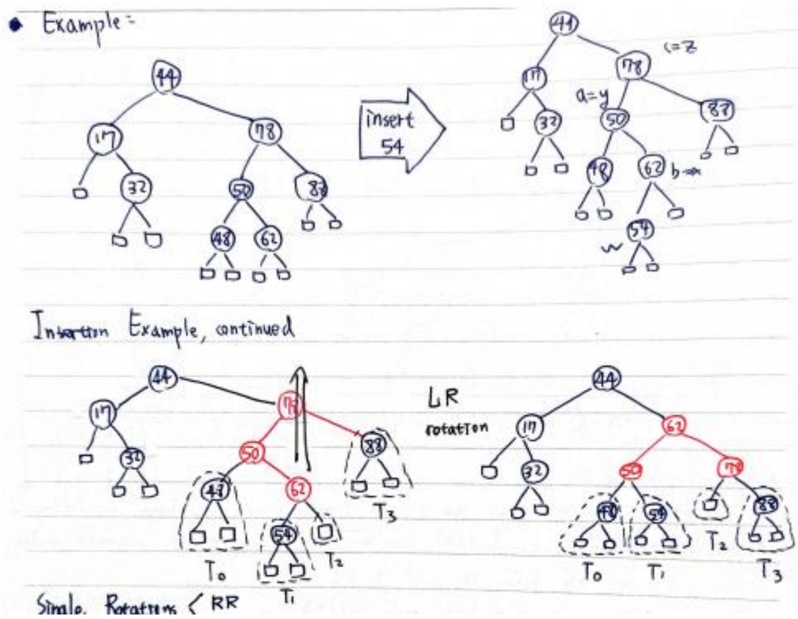
➔$n(h) > 2n(h-2) > 2^2n(h-4) > 2^3n(h-6) > 2^4n(h-8) > 2^in(h-2i)$

➔ $n(h) > 2^{h/2-1}$ ➔ $h < 2\log(n(h)) + 2$

因此樹的高度為 O(log n)

Insert 永遠都插在外部節點，要搜尋要插在哪時間複雜度跟 BST 一樣為 O(log N)。插入後會導致不平衡，需要做調整。調整分成 RR LL LR RL。但調整只跟指標的改變有關，所以調整的動作時間複雜度為 O(1)。

舉例插入 54 導致不平衡。



插入的程式碼:

```
static AVL_NODE * avltree_insertNode(AVL_NODE * tree,elementType key){
    if(tree == NULL){
        AVL_NODE *node = create_node(key,NULL,NULL);
        tree = node;
    }
    else if(key < tree->key){
        tree->left = avltree_insertNode(tree->left,key); //遞迴找插入的節點
        //插入節點後可能引起二叉樹的不平衡， 所以要再進行判斷
        if(getNode_height(tree->left)-getNode_height(tree->right)==2){
            //判斷是LL還是LR
            if(key < tree->left->key){
                //LL旋轉
                tree = LL_rotation(tree);
            }
            else{
                //LR旋轉
                tree = LR_rotation(tree);
            }
        }
    }
    else if(key > tree->key){
        tree->right = avltree_insertNode(tree->right,key);
        if(getNode_height(tree->right)-getNode_height(tree->left)==2){
            //判斷是RR還是RL
            if(key > tree->right->key){
                tree = RR_rotation(tree);
            }
            else{
                tree = RL_rotation(tree);
            }
        }
    }
    //重新調整二元數的高度
    tree->height = MAX(getNode_height(tree->left),getNode_height(tree->right)) +1;
    return tree;
}
```

➔遞迴搜尋時間複雜度 O(log N)

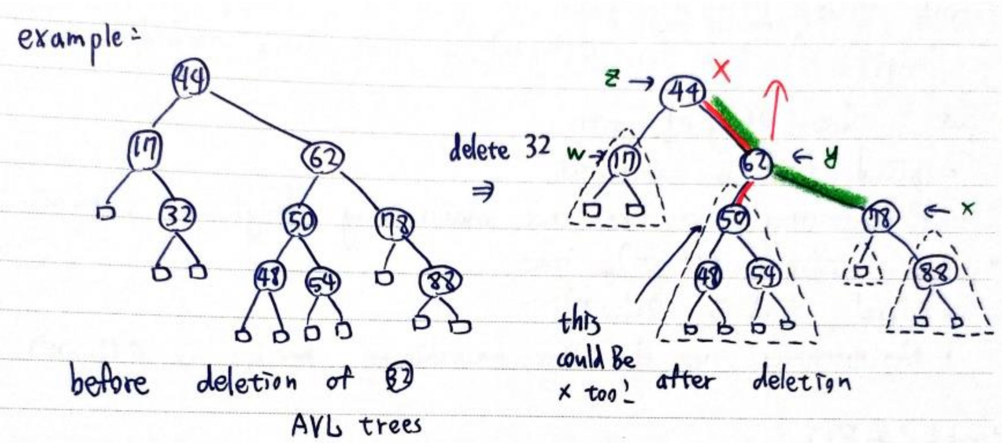
左子樹的高度比右子樹高度多兩個不 balance 判斷是 LL 旋轉還是 LR 旋轉
時間複雜度 O(1)

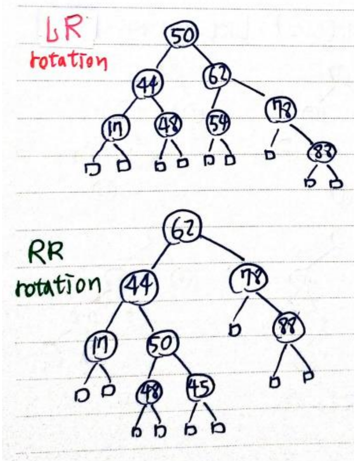
右子樹的高度比左子樹高度多兩個不 balance 判斷是 RR 旋轉還是 RL 旋轉。
時間複雜度 O(1)

調整樹的高度

刪除:

也會導致樹的不平衡，需要做調整。調整分成 RR LL LR RL。但調整只跟指標的改變有關，所以調整的動作時間複雜度為 O(1)。





如果調整後該子樹的高度改變了可能會一直往上調整，因為調整可能會造成另一個 node 無法 balance，所以要持續檢查到 root，看是否每個節點都還維持

balance。 Worst case

刪除的程式碼:

```
static AVL_NODE * avltree_delete(AVL_NODE * tree, elementType key){
    AVL_NODE * z;

    if ((z = search_node(tree, key)) != NULL)
        tree = delete_node(tree, z);
    return tree;
}
```

```
static AVL_NODE * delete_node(AVL_NODE * tree,AVL_NODE *z){
    if(tree == NULL || z==NULL){
        free(tree);
        return NULL;
    }
    if(tree->left == NULL && tree->right == NULL){
        return tree;
    }
    if(z->key < tree->key){
        tree->left = delete_node(tree->left,z);
        if(HEIGHT(tree->right) - HEIGHT(tree->left) == 2){
            AVL_NODE *r = tree->right;
            if(HEIGHT(r->left) > HEIGHT(r->right)){
                tree = RL_rotation(tree);
            }
            else{
                tree = RR_rotation(tree);
            }
        }
    }

    else if(z->key > tree->key){
        tree->right = delete_node(tree->right,z);
        if(HEIGHT(tree->left) - HEIGHT(tree->right) == 2){
            AVL_NODE *l = tree->left;
            if(HEIGHT(l->right) > HEIGHT(l->left)){
                tree = LR_rotation(tree);
            }
            else{
                tree = LL_rotation(tree);
            }
        }
    }
    else{
        //左右小孩都非空
        if((tree->left) && (tree->right)){
            if(HEIGHT(tree->left)>HEIGHT(tree->right)){
                AVL_NODE * max = maximun_node(tree->left);
                tree->key = max->key;
                tree->left = delete_node(tree->left,max);
            }
            else{
                AVL_NODE * min = minimun_node(tree->right);
                tree->key = min->key;
                tree->right = delete_node(tree->right,min);
            }
        }
        else{
            AVL_NODE *tmp = tree;
            tree = tree->left ? tree->left : tree->right;
            free(tmp);
        }
    }
    return tree;
}
```

尋找要刪除的節點

遞迴尋找要刪除的節點，時間複雜度為 O(logN)

遞迴往上(檢查到 root 節點)檢查是否路徑上的每個節點都是 balance，如果沒有就調整。調整一定是 RL 或 RR 因為被刪除的節點已經判斷是在是在上一個節點的右 subtree。

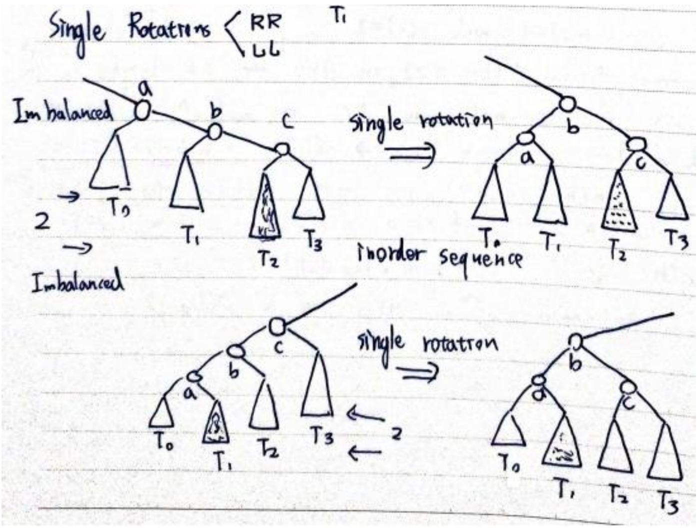
調整的方式也是遞迴其時間複雜度為 O(log N)

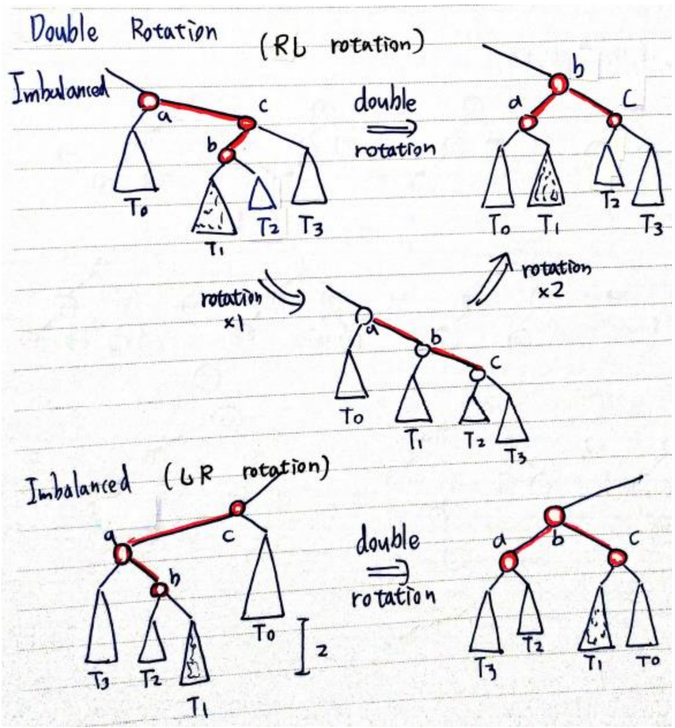
要刪除的節點就為該節點。

如果左子樹的比較高，就拿左子樹最大的節點來取代，並且在左子樹刪除那個要去取代的節點。

如果右子樹的比較高，就拿右子樹最小的節點來取代，並且在右子樹刪除那個要去取代的節點。

旋轉 **RR LL RL LR** 的操作。






```
static AVL_NODE * LL_rotation(AVL_NODE * tree){
    AVL_NODE *k2 = tree->left;
    tree->left = k2->right;
    k2->right = tree;
    tree->height = MAX(getNode_height(tree->left),getNode_height(tree->right)) +1;
    k2->height = MAX(getNode_height(tree->left),getNode_height(tree->right)) +1;
    return k2;
}
static AVL_NODE * RR_rotation(AVL_NODE * tree){
    AVL_NODE *k3 = tree->right;
    tree->right = k3->left;
    k3->left = tree;
    tree->height = MAX(getNode_height(tree->left),getNode_height(tree->right)) +1;
    k3->height = MAX(getNode_height(tree->left),getNode_height(tree->right)) +1;
    return k3;
}
static AVL_NODE * LR_rotation(AVL_NODE * tree){
    tree->left = RR_rotation(tree->left);
    tree = LL_rotation(tree);
    return tree;
}
static AVL_NODE * RL_rotation(AVL_NODE * tree){
    tree->right = LL_rotation(tree->right);
    tree = RR_rotation(tree);
    return tree;
}
```

搜尋的程式碼:

```
static AVL_NODE * search_node(AVL_NODE * tree,elementType key){
    AVL_NODE* n = tree;
    while(n!= NULL){
        if(n->key == key){
            return n;
        }
        else if(key < n->key){
            n = n->left;
        }
        else{
            n = n->right;
        }
    }
    return n;
}
```

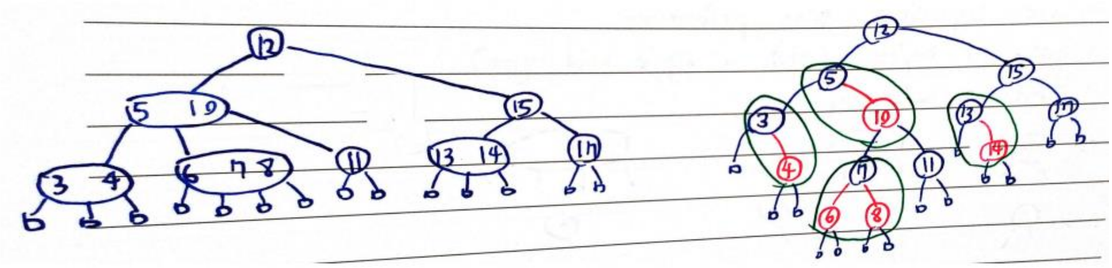
AVL 樹的時間複雜度:

AVL樹	子操作的時間複雜度	總共時間複雜度
插入	O(logN)花在搜尋要插在哪個節點，調整使滿足balance的複雜度為O(1)	O(logN)
搜尋	O(logN)花在binary search tree搜尋要插在哪個節點	O(logN)
刪除	O(logN)花在搜尋要插在哪個節點，調整使滿足balance的複雜度為O(1)，但要檢查是否導致其他subtree的node不balance，需要走訪節點到root，時間複雜度O(log N)	2O(logN)

Red black tree 是一種 234 樹的二元樹的表示方法，它的每一個節點被塗成紅色或黑色。因此他跟 234 樹有相同的時間複雜度，就差在他比較容易超做，因為單一節點只存一筆資料。

Red black tree 的高度大約是 234 樹的兩倍，因此如果存了 n 筆的資料，樹的高度大概是 O(logN)，而紅黑樹的搜尋演算法就如同二元搜尋樹一樣，所以紅黑樹的搜尋時間複雜度為 O(long N)。

左邊 234 樹，右邊紅黑樹的高度大概為 234 樹的兩倍。



當要插入 insert 值到紅黑樹裡面時，為了滿足下面的特性:

1. root property : the root is black.
2. External property: every leaf is black.
3. Internal property: 紅色節點的 children 必須為黑色節點。每一條路徑，不能連續兩個紅色節點出現。
4. Depth property:每個子節點都有相同的黑色節點數量。

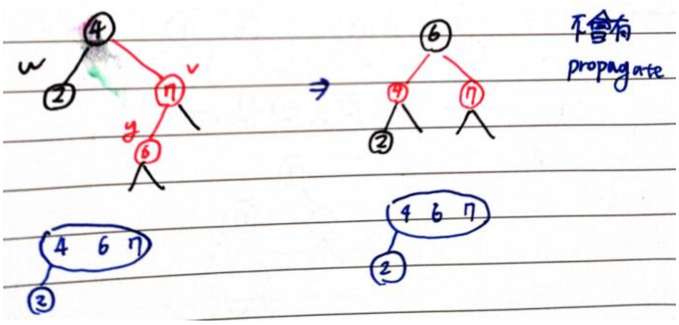
所以插入的新節點都是紅色的。如果插入後節點的 parent 為黑節點，則不用做任何事情，時間複雜度為 O(1)。

倘若插入後的節點的 parent 為紅節點(則不滿足特性 3.Internal property)，則需要做調整來符合上述的特性➔這種情況稱為 double red。

發生 double red 又分成兩種狀況，

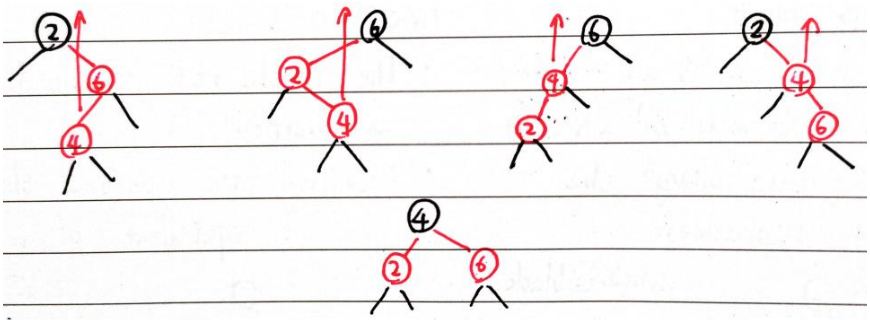
CASE1 : (要做 Restruct)

插入節點的 uncle(父節點的兄弟)為黑色，此時需要做 restruct。相當於在 234 樹中插入節點中的節點的數量有 3 個。

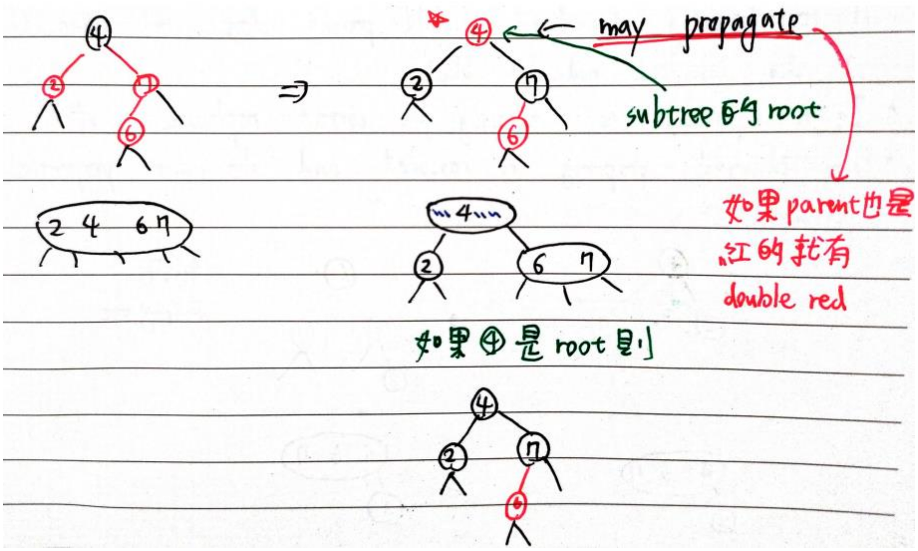


此情況不會造成還要調整 grandparent 節點的，因為只要調整插入附近的，使附

近維持上述特性。則整棵樹都不需要變動。
可能會發生的狀態以下面 4 種。就調整成中間下面那種狀態。
中間下面的狀態最上面的節點為黑色，也就說明這種調整不會影響到上面的節點。不會因為調整完又再次發生 double red，需要做調整。



CASE2 : (要做 Recorling)
當插入節點的 uncle 節點為紅色的，則相當於在 234 樹中原先已經有 3 個值的節點插入第 4 個值，這樣則不符合規則。
要如何解決這個問題就是相當於在 234 樹中進行分裂。
操作:將插入節點的父節點變成黑色，節點的 grandparent 節點變成紅色，原先插入節點的 uncle 節點為紅色改成黑色。



如果節點 4 不是整棵樹的 root 則他的紅色會造成 propagation delay，而為什麼要將結點 4 recolor 成紅色，因為不調整紅色，則這棵 subtree 的每個路徑的黑色節點數量會增加一。調整成紅色，則有可能更上面節點也為紅色，又會再次發生 double red 的問題。因此一路調整上去其時間複雜度 worst case 相當於樹的高度 O(logN)。
如果節點 4 為 root 則直接改成黑色節點，符合 Root property 與 Internal property。

總結 ：紅黑樹的插入時間複雜度，插入前會先 BST 搜尋時間複雜度 O(logN)

紅黑樹插入		操作	時間複雜度
插入節點的parent為黑色節點		直接插入	O(1)
插入節點的parent為紅色節點發生double red	case1:uncle為黑色	restruct不會發生double red propagation	O(1)
	case2:uncle為紅色	recolor，如果grandparent不為root可能發生double propagation	worstcase O(logN)

此函數為找到要插入節點的位置。時間複雜度為 O(log N)

```
static void rbtree_insert(int key) {
    RB_NODE* inserted_node = new_node(key, RED, NULL, NULL);
    if (RB_tree == NULL) {
        RB_tree = inserted_node;
    }
    else {
        RB_NODE* n = RB_tree;
        while (1) {
            int comp_result = compare(key, n->key);
            if (comp_result == 0) {
                free (inserted_node);
                return;
            }
            else if (comp_result < 0) {
                if (n->left == NULL) {
                    n->left = inserted_node;
                    break;
                }
                else { n = n->left; }
            }
            else {
                if (n->right == NULL) {
                    n->right = inserted_node;
                    break;
                }
                else { n = n->right; }
            }
        }
        inserted_node->parent = n;
    }
    insert_case1(inserted_node);
}
```

```
static void insert_case1(RB_NODE* n) {
    if (n->parent == NULL)
        n->color = BLACK;
    else insert_case2(n);
}
```

沒有 parent 的話直接插入

```
static void insert_case2(RB_NODE* n) {
    if (node_color(n->parent) == BLACK)
        return;
    else insert_case3(n);
}
```

如果 parent 是黑色的話直接插入

```
static void insert_case3(RB_NODE* n) {
    if (node_color(uncle(n)) == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(grandparent(n));
    }
    else  insert_case4(n); //uncle是黑色的
}
```

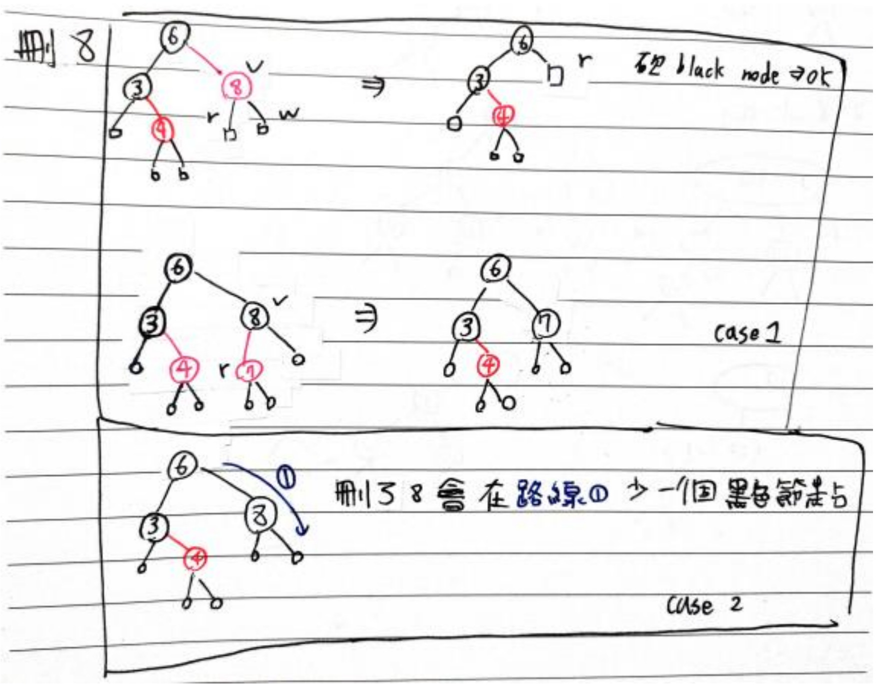
如果 uncle 是紅色的話➔Recolor

➔有可能造成 double red 的狀況 propagation up

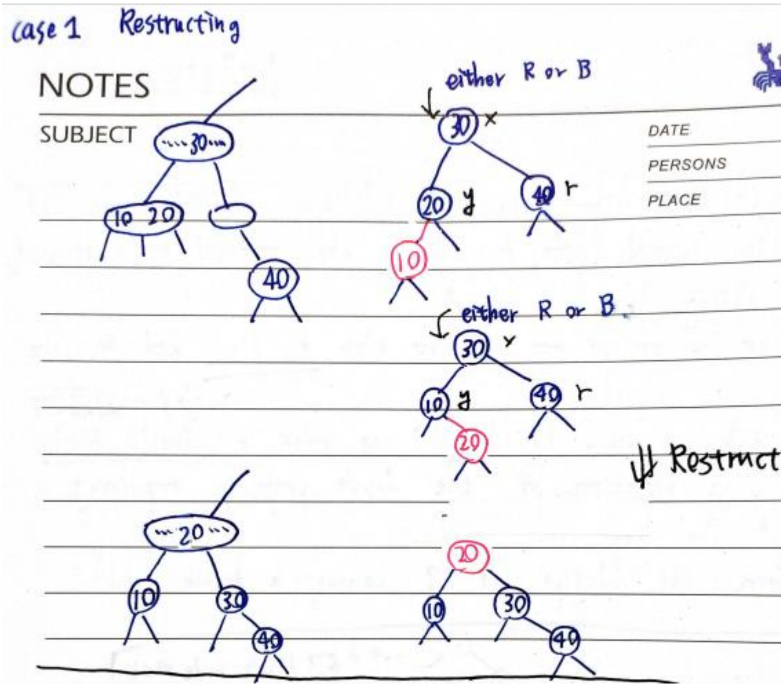
Uncle 是黑色的 Restruct 有四種狀況 RR RL LL LR

```
static void insert_case4(RB_NODE* n) {
    if (n == n->parent->right && n->parent == grandparent(n)->left) { //在grandparent的左右
        rotate_left(n->parent);
        n = n->left;
    }
    else if (n == n->parent->left && n->parent == grandparent(n)->right) { //在grandparent的右左
        rotate_right(n->parent);
        n = n->right;
    }
    //右右或左左
    insert_case5(n);
}
static void insert_case5(RB_NODE* n) {
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n == n->parent->left && n->parent == grandparent(n)->left) { //左左
        rotate_right(grandparent(n));
    }
    else if(n == n->parent->right && n->parent == grandparent(n)->right){ //右右
        rotate_left(grandparent(n));
    }
}
static void rotate_left(RB_NODE* n) { ////在grandparent的左右 左轉
    RB_NODE* r = n->right;
    replace_node(n, r); //
    n->right = r->left;
    if (r->left != NULL) {
        r->left->parent = n;
    }
    r->left = n;
    n->parent = r;
}

static void rotate_right(RB_NODE* n) {
    RB_NODE* L = n->left;
    replace_node(n, L);
    n->left = L->right;
    if (L->right != NULL) {
        L->right->parent = n;
    }
    L->right = n;
    n->parent = L;
}

static void replace_node(RB_NODE* oldn, RB_NODE* newn) {
    if (oldn->parent == NULL) {
        RB_tree = newn;
    }
    else {
        if (oldn == oldn->parent->left)
            oldn->parent->left = newn;
        else
            oldn->parent->right = newn;
    }
    if (newn != NULL) {
        newn->parent = oldn->parent;
    }
}
```

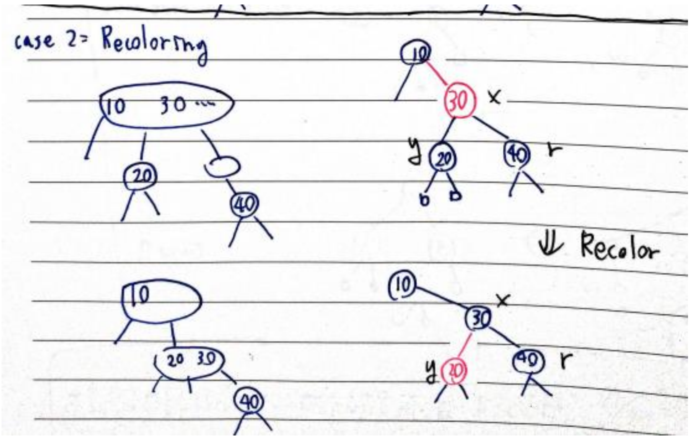
紅黑樹的刪除:(要先 binary search tree 找到要刪除的節點時間複雜度 O(logN)
v 是要被移除的 internal node。W 是外部節點，r 是 w 的 sibling。
CASE1.當 v 或 r 有一個節點是紅色，則將 r 變成黑色後再直接刪除。
➔因此時間複雜度為 O(1)，如附圖 case1
CASE2.但當 v 或 r 兩個節點都是黑色的，則刪除 V 會導致路徑少一個黑色的節點(不符合 Depth property)➔此情況稱為 double black。如附圖 case2。



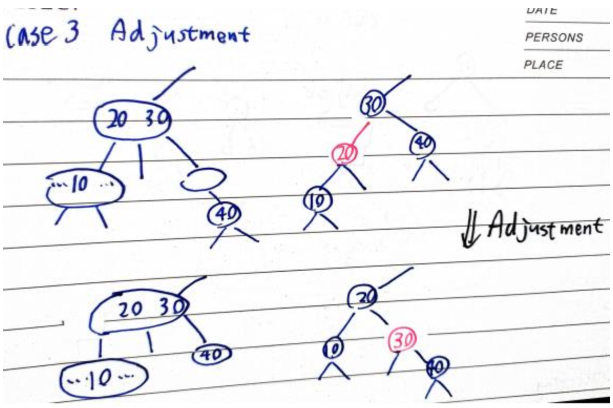
Case 2.1 Restruct : y 是黑色，且有一個小孩為紅色節點。時間複雜度 O(1)



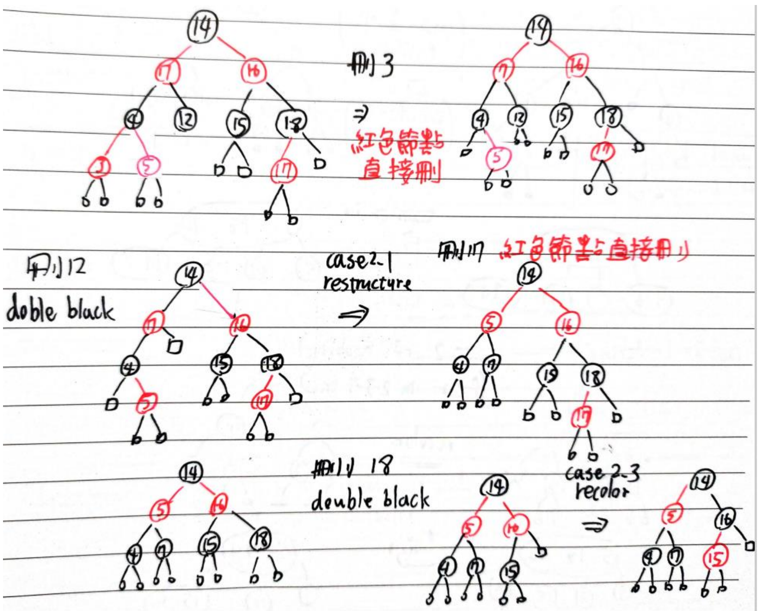
Case 2.2 Recolor: y 是黑色，且兩個小孩都是黑色。將 x 改為黑色，會造成 double black 的狀況被 propagate up。也就是要繼續處理上面節點以維持紅黑樹的特性。最壞狀況時間複雜度 O(log N)相當於樹的高度

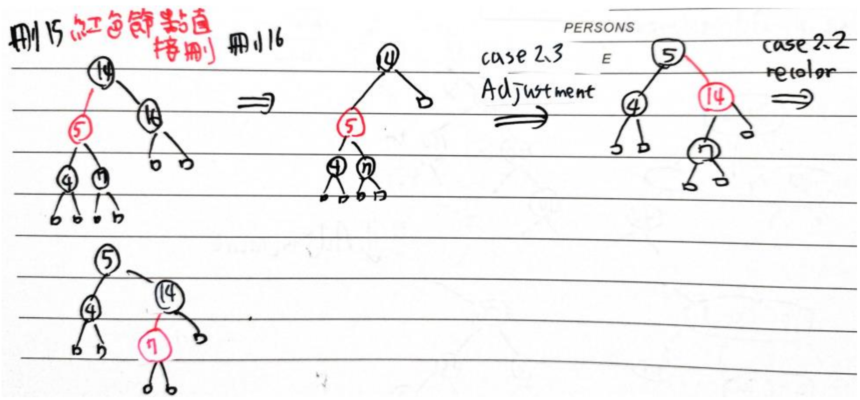


Case 2.3 Adjustment : y 是紅色。相當於換 234 樹中有 3 個值的節點的另一種紅黑數表示法。時間複雜度 O(1)



舉例子





紅黑樹的刪除總結:刪除前會先 BST 搜尋時間複雜度 O(logN)

紅黑樹刪除		操作	時間複雜度
刪除節點的parent為紅色節點或是刪除的節點有個child是紅色節點		直接刪除	O(1)
刪除的節點兩個child都是黑色節點 發生double black	case1:Restruct		O(1)
	case2:Recolor	會造成double black 的狀況 propogation up	worstcase O(logN)
	case3:Adjustment		O(1)

此函數再找到要刪除的函數。

```
static void rbtree_delete(int key) {
    RB_NODE* child;
    int comp_result = compare(key, RB_tree->key);
    RB_NODE* n = lookup_node(key);
    if (n != NULL){
        if (n->left != NULL && n->right != NULL) {
            RB_NODE* pred = maximum_node(n->left);
            n->key  = pred->key;
            n = pred;
        }
        if((n->left == NULL || n->right == NULL)){
            child = n->right == NULL ? n->left  : n->right;
            if (node_color(n) == BLACK) {
                n->color = node_color(child);
                delete_case1(n);
            }
            replace_node(n, child);
            if (n->parent == NULL && child != NULL)
                child->color = BLACK;
            free(n);
        }
    }
}
```

先將值取代。

Child 取代 root。

```c
static void delete_case1(RB_NODE* n) {
    if (n->parent == NULL)
        return;
    else
        delete_case2(n);
}

static void delete_case2(RB_NODE* n) {
    if (node_color(sibling(n)) == RED) {
        n->parent->color = RED;
        sibling(n)->color = BLACK;
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
    }
    delete_case3(n);
}

static void delete_case3(RB_NODE* n) {
    if (node_color(n->parent) == BLACK &&
        node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == BLACK &&
        node_color(sibling(n)->right) == BLACK)
    {
        sibling(n)->color = RED;
        delete_case1(n->parent);
    }
    else
        delete_case4(n);
}

static void delete_case4(RB_NODE* n) {
    if (node_color(n->parent) == RED &&
        node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == BLACK &&
        node_color(sibling(n)->right) == BLACK)
    {
        sibling(n)->color = RED;
        n->parent->color = BLACK;
    }
    else
        delete_case5(n);
}

static void delete_case5(RB_NODE* n) {
    if (n == n->parent->left &&
        node_color(sibling(n)) == BLACK &&
        node_color(sibling(n)->left) == RED &&
        node_color(sibling(n)->right) == BLACK)
    {
        sibling(n)->color = RED;
        sibling(n)->left->color = BLACK;
        rotate_right(sibling(n));
    }
    else if (n == n->parent->right &&
            node_color(sibling(n)) == BLACK &&
            node_color(sibling(n)->right) == RED &&
            node_color(sibling(n)->left) == BLACK)
    {
        sibling(n)->color = RED;
        sibling(n)->right->color = BLACK;
        rotate_left(sibling(n));
    }
    delete_case6(n);
}
```

```c
static void delete_case6(RB_NODE* n) {
    sibling(n)->color = node_color(n->parent);
    n->parent->color = BLACK;
    if (n == n->parent->left) {
        if(node_color(sibling(n)->right) == RED){
            sibling(n)->right->color = BLACK;
            rotate_left(n->parent);
        }
    }
    else
    {
        if(node_color(sibling(n)->left) == RED){
            sibling(n)->left->color = BLACK;
            rotate_right(n->parent);
        }
    }
}
```

AVL 與 Red Black Tree 的比較:

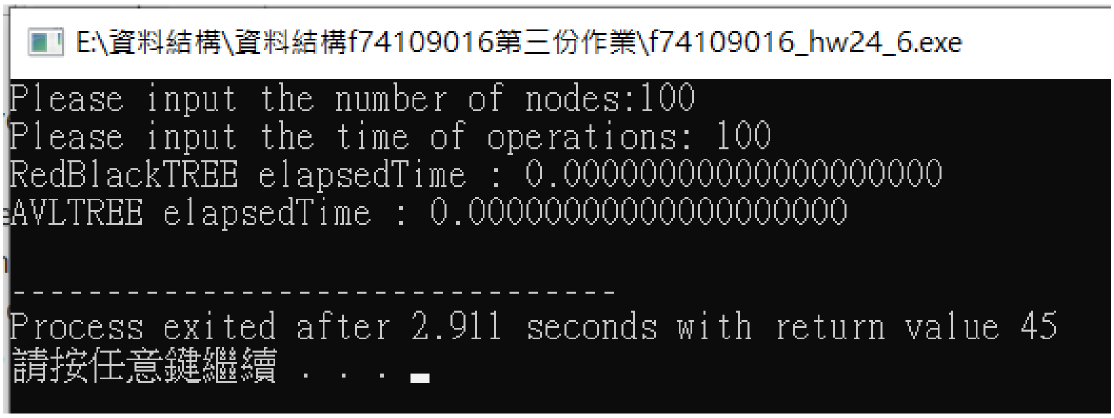
紅黑樹插入		操作	時間複雜度
插入節點的parent為黑色節點		直接插入	O(1)
插入節點的parent為紅色節點發生double red	case1:uncle為黑色	restruct不會發生double red propagation	O(1)
	case2:uncle為紅色	recolor，如果grandparent不為root可能發生double red propagation up	worstcase O(logN)
紅黑樹搜尋	O(logN)花在binary search tree搜尋要插在哪個節點		O(logN)
紅黑樹刪除		操作	
刪除節點的parent為紅色節點或是刪除的節點有個child是紅色節點		直接刪除	O(1)
刪除的節點兩個child都是黑色節點發生double black	case1:Restruct		O(1)
	case2:Recolor	會造成double black 的狀況 propogation up	worstcase O(logN)
	case3:Adjustment		O(1)

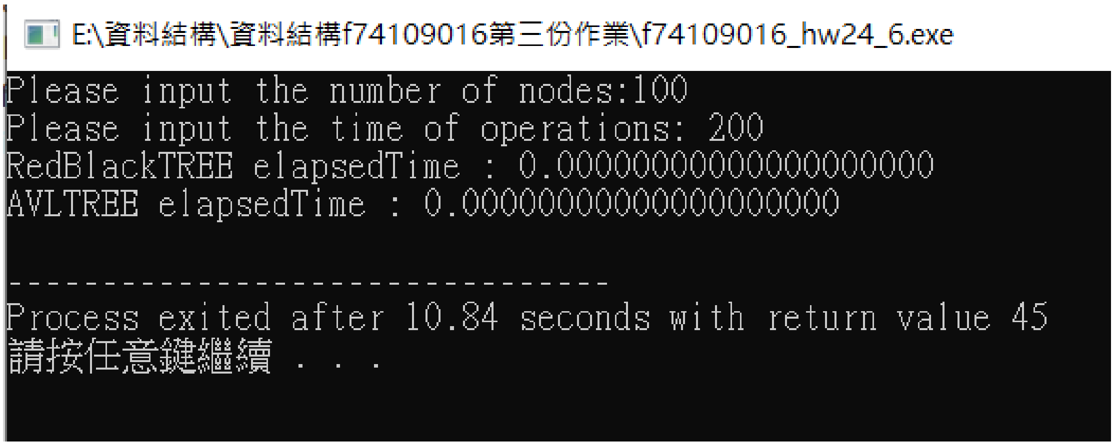
AVL樹	子操作的時間複雜度	總共時間複雜度
插入	O(logN)花在搜尋要插在哪個節點，調整使滿足balance的複雜度為O(1)	O(logN)
搜尋	O(logN)花在binary search tree搜尋要插在哪個節點	O(logN)
刪除	O(logN)花在搜尋要插在哪個節點，調整使滿足balance的複雜度為O(1)，但要檢查是否導致其他subtree的node不balance，需要走訪節點到root，時間複雜度O(log N)	2O(logN)

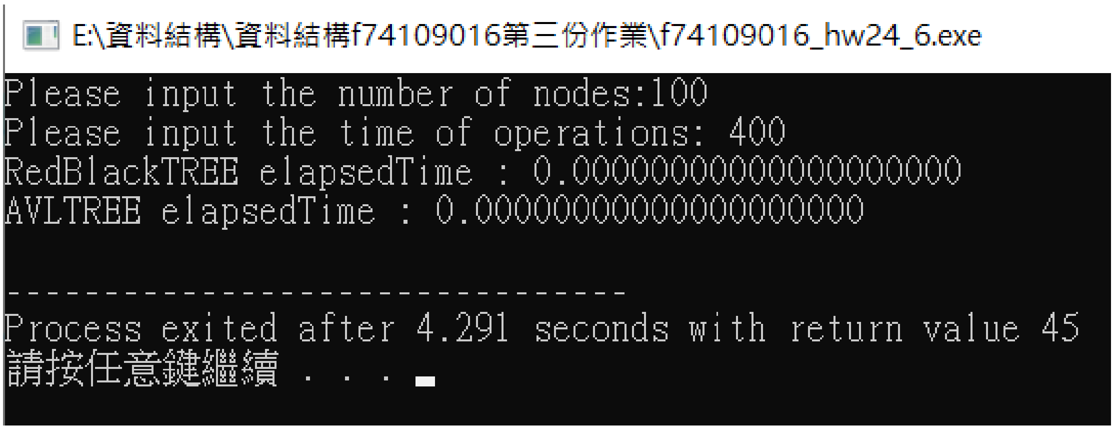
兩個資料結構在搜尋的時間複雜度相同，在插入與刪除的時候也會先進行 BST 時間複雜度為 O(log N)，而紅黑樹在插入與刪除後，調整以滿足特性時的時間複雜度只有在 worst case 是 O(lon N)。而研究顯示，這種調整的比率並不大。通常都是可以直接插入與刪除。

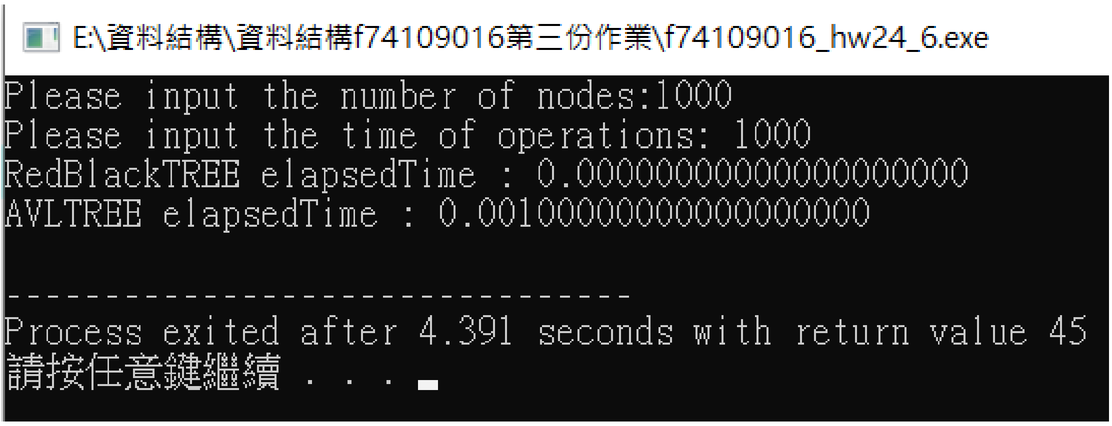
而 AVL 在刪除後只要有改變樹高就要一直檢查到 root 所以時間複雜度為 worst case 2O(logN)。而改變樹高的頻率並不算低。所以從理論推 AVL 的效率較 Red Black Tree 低。

由實驗數據也可以看出 RedBlackTree 的操作效率比 AVLTree 來的好。









E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:1000
Please input the time of operations: 2000
RedBlackTREE elapsedTime : 0.00000000000000000000
AVLTREE elapsedTime : 0.00100000000000000000

--------------------------------
Process exited after 2.916 seconds with return value 45
請按任意鍵繼續 . . .
```

E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:1000
Please input the time of operations: 4000
RedBlackTREE elapsedTime : 0.00100000000000000000
AVLTREE elapsedTime : 0.00100000000000000000

--------------------------------
Process exited after 6.033 seconds with return value 45
請按任意鍵繼續 . . .
```

E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:10000
Please input the time of operations: 10000
RedBlackTREE elapsedTime : 0.00400000000000000010
AVLTREE elapsedTime : 0.00500000000000000010

--------------------------------
Process exited after 4.986 seconds with return value 45
請按任意鍵繼續 . . .
```

E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:10000
Please input the time of operations: 20000
RedBlackTREE elapsedTime : 0.00600000000000000010
AVLTREE elapsedTime : 0.00700000000000000010

--------------------------------
Process exited after 8.48 seconds with return value 45
請按任意鍵繼續 . . .
```

E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:10000
Please input the time of operations: 40000
RedBlackTREE elapsedTime : 0.010000000000000000
AVLTREE elapsedTime : 0.010999999999999900

--------------------------------
Process exited after 7.195 seconds with return value 45
請按任意鍵繼續 . . .
```

E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:100000
Please input the time of operations: 100000
RedBlackTREE elapsedTime : 0.043999999999999700
AVLTREE elapsedTime : 0.064000000000000100

--------------------------------
Process exited after 7.728 seconds with return value 45
請按任意鍵繼續 . . .
```

E:\資料結構\資料結構f74109016第三份作業\f74109016_hw24_6.exe

```
Please input the number of nodes:100000
Please input the time of operations: 200000
RedBlackTREE elapsedTime : 0.065000000000000200
AVLTREE elapsedTime : 0.087999999999999500

--------------------------------
Process exited after 11.44 seconds with return value 45
請按任意鍵繼續 . . .
```

				每一步	每一步
		RedBlackTREE	AVLTREE	RedBlackTREE	AVLTREE
n=100	m=100	0	0	0	0
	m=200	0	0	0	0
	m=400	0	0	0	0
n=1000	m=1000	0	0.001	0	0.5us
	m=2000	0	0.001	0	0.33us
	m=4000	0.001	0.001	0.2us	0.2us
n=10000	m=10000	0.004	0.005	0.2us	0.25us
	m=20000	0.006	0.007	0.2us	0.2333us
	m=40000	0.01	0.0109	0.2us	0.218us
n=100000	m=100000	0.044	0.064	0.22us	0.32us
	m=200000	0.065	0.088	0.216666us	0.2933us
平均每步				0.206111us	0.293075us