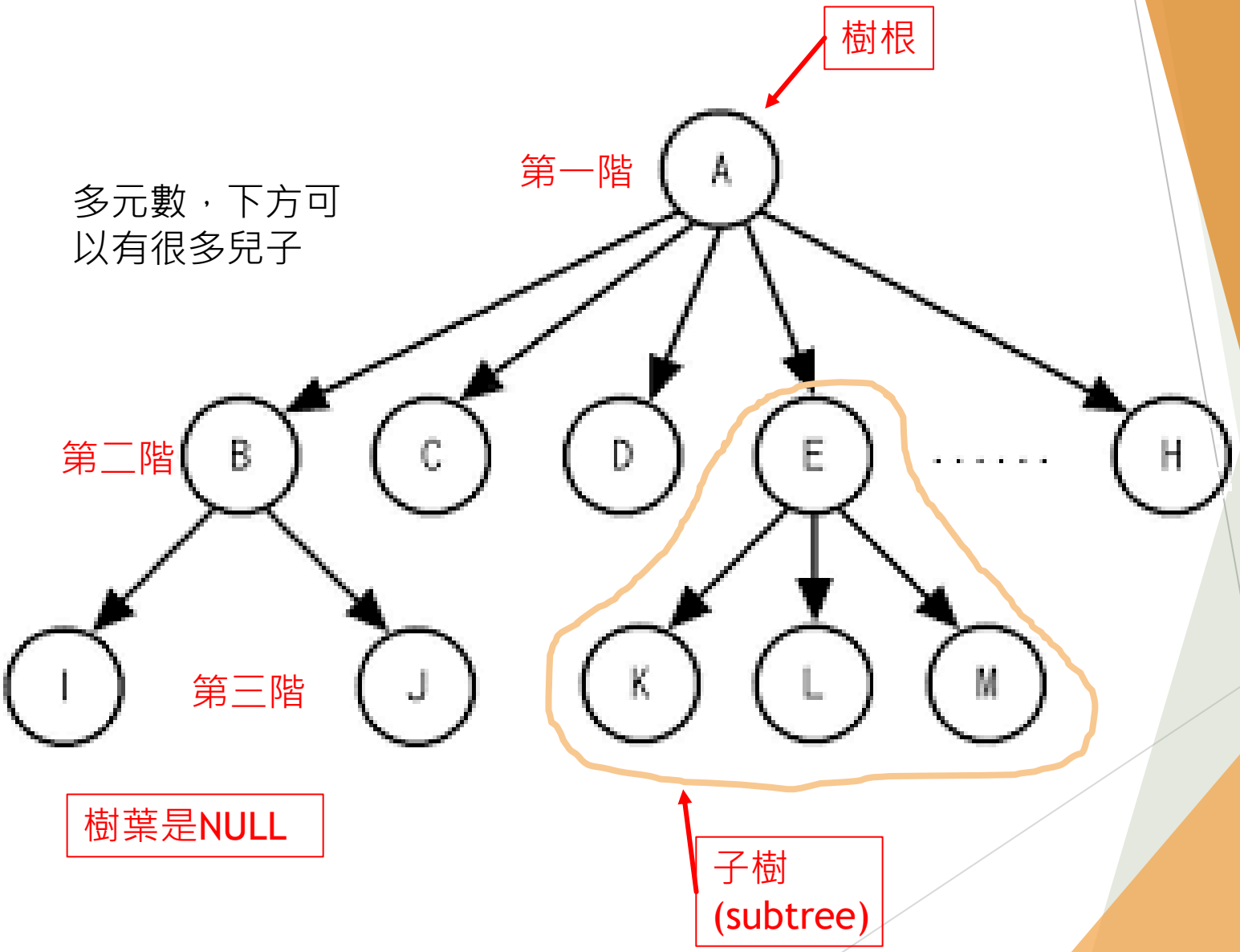
資料結構與C++進階班
樹

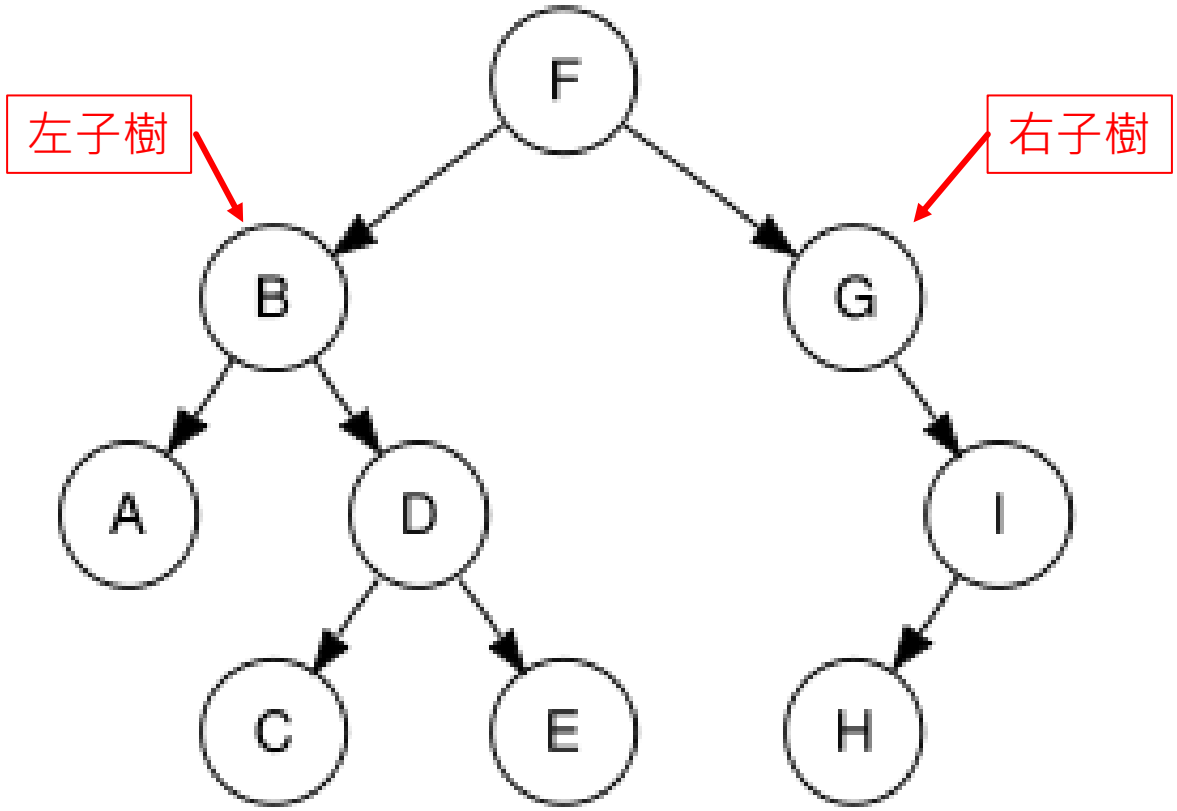
講師：黃銀鵬

E-mail: yinpenghuang@gmail.com

樹

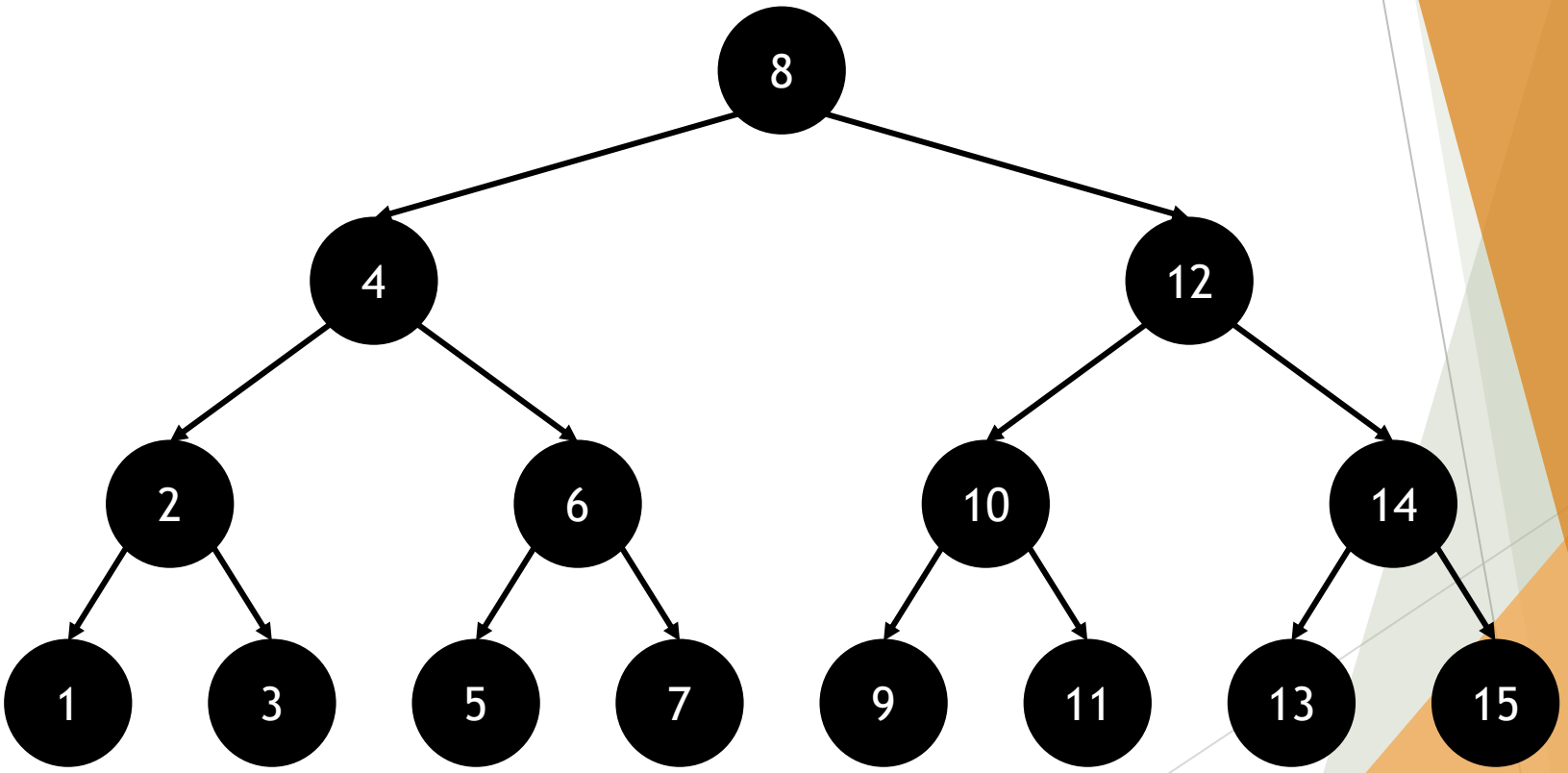

樹根
第一階
A
多元數，下方可
以有很多兒子
第二階
B
C
D
E
H
I
第三階
J
K
L
M
樹葉是NULL
子樹
(subtree)

二元樹



二元搜尋樹(Binary Search Tree)

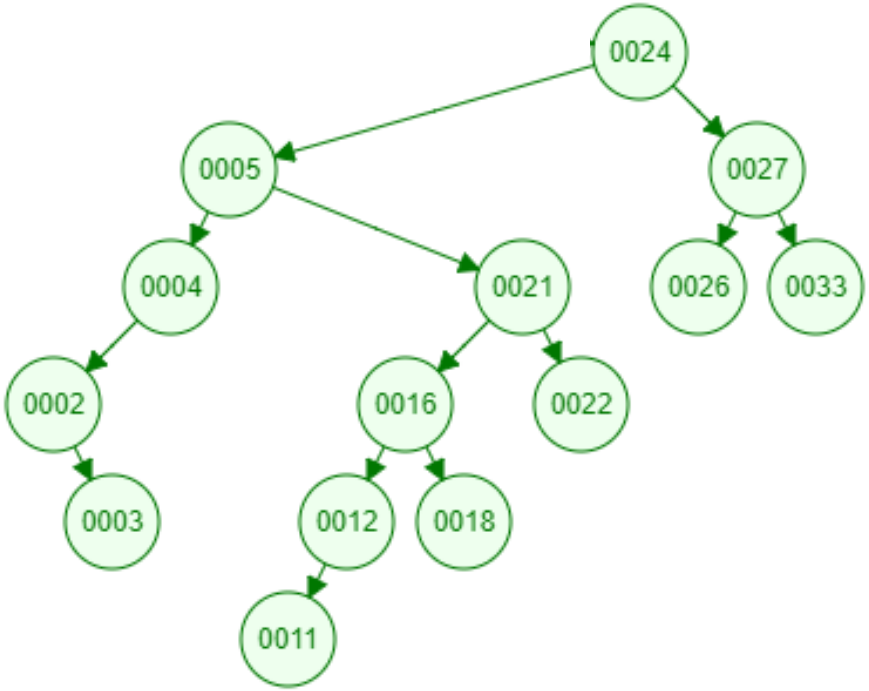
左小右大



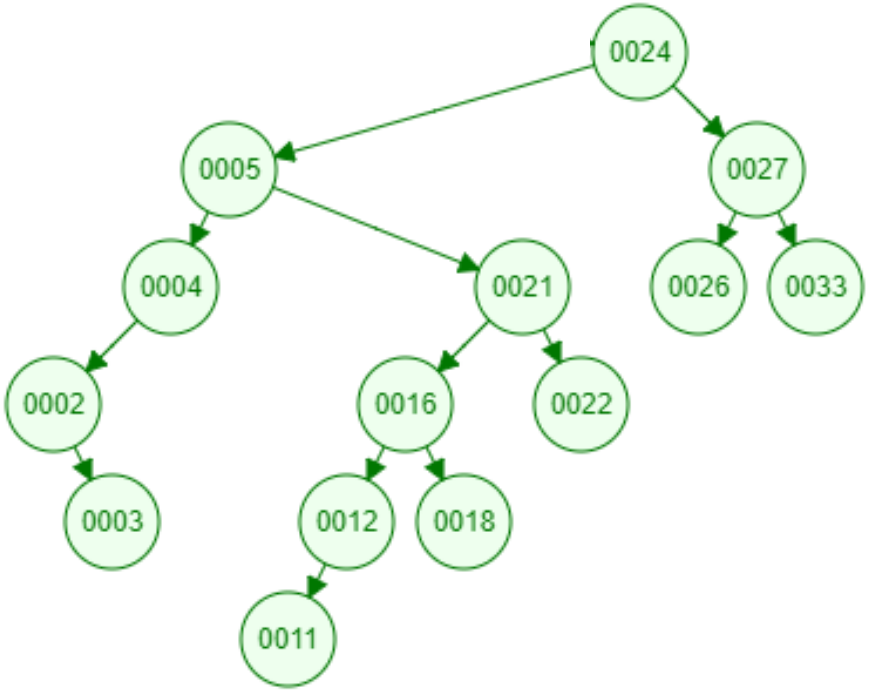
二元樹的走訪(來判斷此樹是否正確)

- 中序走訪：左節點→自身→右節點
  - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
- 前序走訪：自身→左節點→右節點
  - 8, 4, 2, 1, 3, 6, 5, 7, 12, 10, 9, 11, 14, 13, 15
- 後序走訪：左節點→右節點→自身
  - 1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8
- 先左再右

紙筆練習：


0024
0005
0027
0004
0021
0026
0033
0002
0016
0022
0003
0012
0018
0011

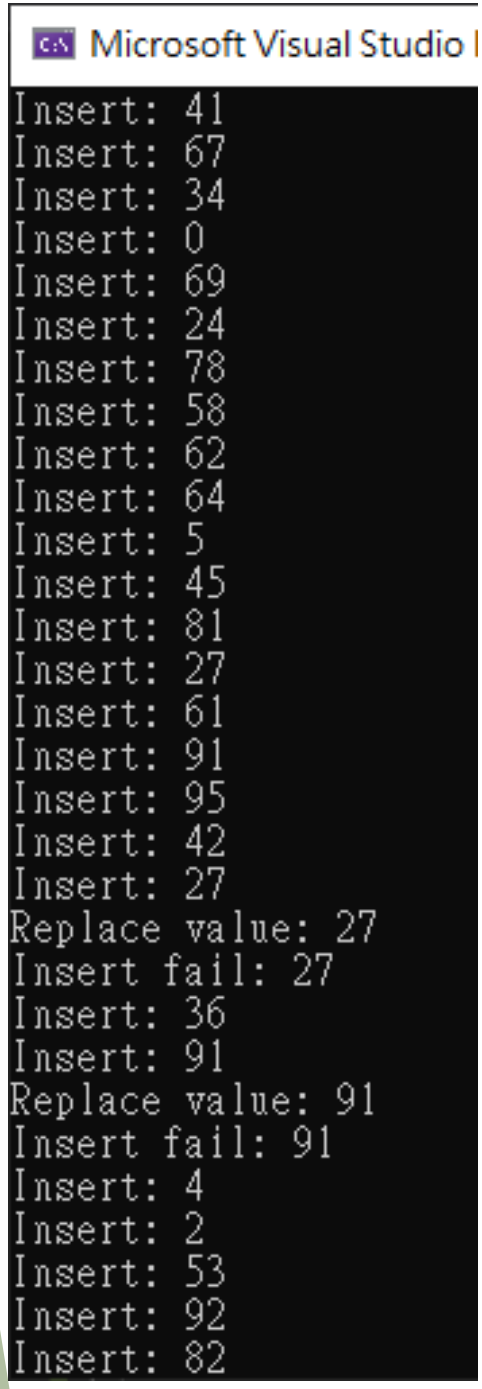
紙筆練習：

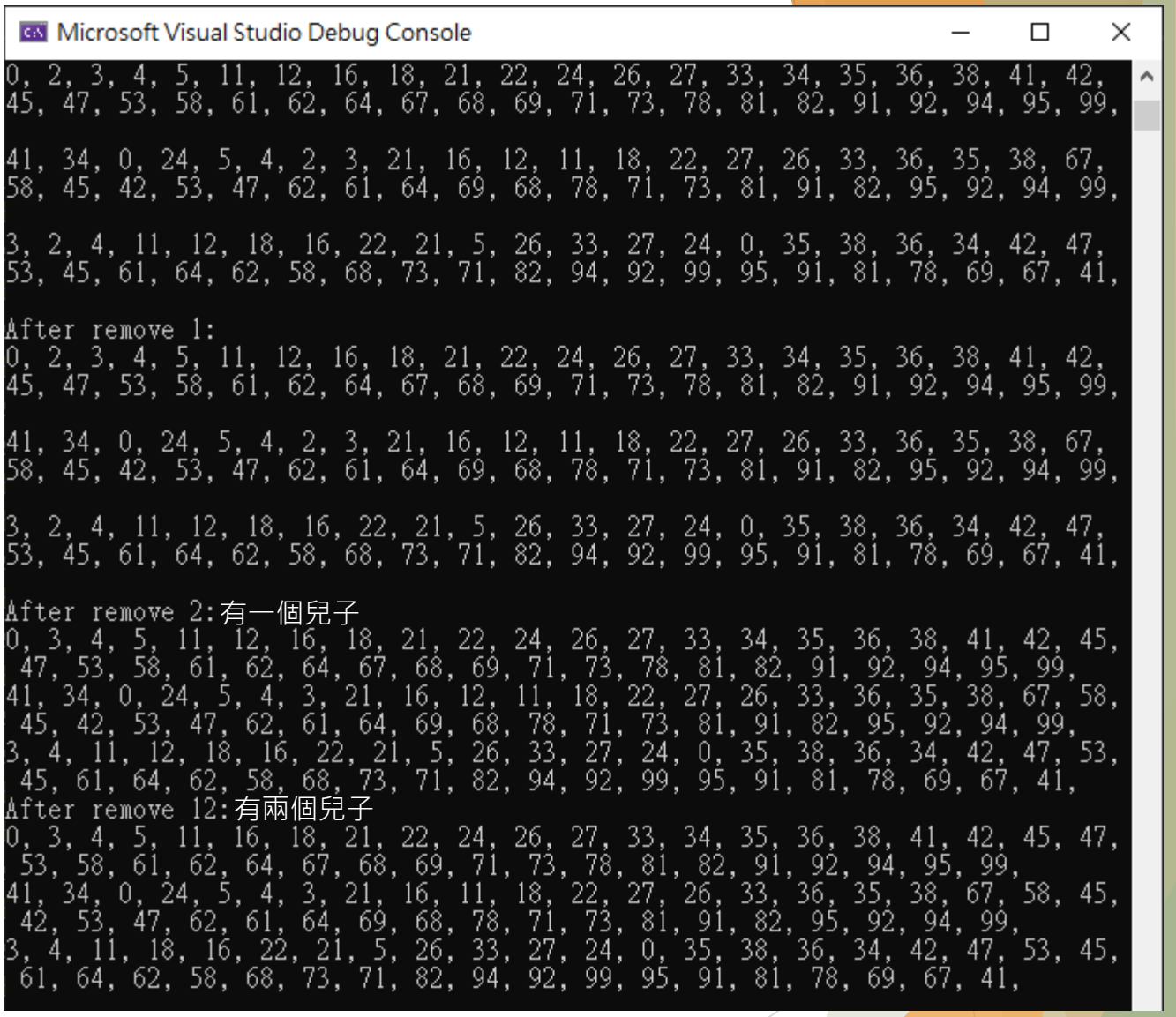


中序：2, 3, 4, 5, 11, 12, 16, 18, 21, 22, 24, 26, 27, 33
前序：24, 5, 4, 2, 3, 21, 16, 12, 11, 18, 22, 27, 26, 33
後序：3, 2, 4, 11, 12, 18, 16, 22, 21, 5, 26, 33, 27, 24

二元樹-刪除元素

- 刪除結點沒有子結點，可直接刪除。
- 刪除結點僅有一個子結點，將其父結點指向唯一的子結點後刪除。
- 刪除結點有兩個子結點：
  - 挑左子樹最大的結點取代被刪除的結點後刪除原節點。
  - 或挑右子樹最小的結點取代被刪除的結點後刪除原節點。





老師這裡是不能傳入相等的數，
其實可以根據需求下去定義

樹是沒有位置概念的，
只有數字

```cpp
#include "CBinarySearchTree.h"

int main()
{
    int i;
    int a[] = { 41, 67, 34, 0, 69, 24, 78, 58, 62, 64,\
        5, 45, 81, 27, 61, 91, 95, 42, 27, 36,\
        91, 4, 2, 53, 92, 82, 21, 16, 18, 95,\
        47, 26, 71, 38, 69, 12, 67, 99, 35, 94,\
        3, 11, 22, 33, 73, 64, 41, 11, 53, 68 };
    CBinarySearchTree<int> bst;
    for (i = 0; i < sizeof(a) / sizeof(int); ++i)
    {
        std::cout << "Insert: " << a[i] << std::endl;
        if (!bst.Insert(a[i]))
        {
            std::cout << "Insert fail: " << a[i] << std::endl;
            //break;
        }
    }
    bst.Inorder(bst.m_Root);
    std::cout << std::endl;
    bst.Preorder(bst.m_Root);
    std::cout << std::endl;
    bst.Posorder(bst.m_Root);
    std::cout << std::endl;
    std::cout << "After remove 1:" << std::endl;
    bst.Remove(1);
    bst.Inorder(bst.m_Root);
    std::cout << std::endl;
    bst.Preorder(bst.m_Root);
    std::cout << std::endl;
    bst.Posorder(bst.m_Root);
    std::cout << std::endl;
    std::cout << "After remove 2:" << std::endl;
    bst.Remove(2);
    bst.Inorder(bst.m_Root);
    std::cout << std::endl;
    bst.Preorder(bst.m_Root);
    std::cout << std::endl;
    bst.Posorder(bst.m_Root);
    std::cout << std::endl;
    std::cout << "After remove 12:" << std::endl;
    bst.Remove(12);
    bst.Inorder(bst.m_Root);
    std::cout << std::endl;
    bst.Preorder(bst.m_Root);
    std::cout << std::endl;
    bst.Posorder(bst.m_Root);
    std::cout << std::endl;
    return 0;
}
```

```cpp
#pragma once

#include <iostream>

template<class T>
class CNode
{
public:
    CNode<T>* m_Left;
    CNode<T>* m_Right;
    T m_Value;
    bool m_IsEmpty;

    CNode()
        : m_Value()
        , m_Left(NULL)
        , m_Right(NULL)
        , m_IsEmpty(true) { };
    ~CNode() {};
};
```

```cpp
template <class T>
class CBinarySearchTree
{
public:
    CNode<T>* m_Root;
    CBinarySearchTree();
    ~CBinarySearchTree();
    bool Insert(T value);
    bool Remove(T value);
    void Inorder(CNode<T>* root);
    void Preorder(CNode<T>* root);
    void Posorder(CNode<T>* root);

private:
    void DeleteTree(CNode<T>* root);
    CNode<T>* GetEmptyNode(CNode<T>* root, T value);
    CNode<T>* GetDeleteNode(CNode<T>* root, T value);
    bool SetNode(CNode<T>* node, T value);
    bool DeleteNode(CNode<T>* node);
    CNode<T>* GetMaxInLeftTree(CNode<T>* root);

};

template<class T>
inline CBinarySearchTree<T>::CBinarySearchTree()
{
    m_Root = new CNode<T>;
}

template<class T>
inline CBinarySearchTree<T>::~CBinarySearchTree()
{
    DeleteTree(m_Root);
}
```

永遠指向樹根(放public 不太好喔，如果別人破壞到你的樹根就毀了)，但在外面必須得到這個樹根

```cpp
template<class T>
inline void CBinarySearchTree<T>::DeleteTree(CNode<T>* root)
{
    if (root == NULL)
        return;
    DeleteTree(root->m_Left);
    DeleteTree(root->m_Right);
    delete root;
}
```

```cpp
template<class T>
inline void CBinarySearchTree<T>::Inorder(CNode<T>* root)
{
    if (root->m_IsEmpty) return;
    Inorder(root->m_Left);
    std::cout << root->m_Value << ", ";
    Inorder(root->m_Right);
}

template<class T>
inline void CBinarySearchTree<T>::Preorder(CNode<T>* root)
{
    if (root->m_IsEmpty) return;
    std::cout << root->m_Value << ", ";
    Preorder(root->m_Left);
    Preorder(root->m_Right);
}

template<class T>
inline void CBinarySearchTree<T>::Posorder(CNode<T>* root)
{
    if (root->m_IsEmpty) return;
    Posorder(root->m_Left);
    Posorder(root->m_Right);
    std::cout << root->m_Value << ", ";
}
```

```cpp
template<class T>
inline bool CBinarySearchTree<T>::Insert(T value)
{
    CNode<T>* insertNode = GetEmptyNode(m_Root, value);
    if (!insertNode) return false;
    return SetNode(insertNode, value);
}
```

```cpp
template<class T>
inline CNode<T>* CBinarySearchTree<T>::GetEmptyNode(CNode<T>* root, T value)
{
    if (root->m_IsEmpty) return root;
    if (root->m_IsEmpty == false && value == root->m_Value)
    {
        std::cout << "Replace value: " << value << std::endl;
        return NULL;
    }
    if (value < root->m_Value)
    {
        if (root->m_Left->m_IsEmpty)
            return root->m_Left;
        else
            return GetEmptyNode(root->m_Left, value);
    }
    else if (value > root->m_Value)
    {
        if (root->m_Right->m_IsEmpty)
            return root->m_Right;
        else
            return GetEmptyNode(root->m_Right, value);
    }
    return NULL;
}
```

```cpp
template<class T>
inline bool CBinarySearchTree<T>::SetNode(CNode<T>* node, T value)
{
    if (!node) return false;
    node->m_Left = new CNode<T>;
    if (!node->m_Left) return false;
    node->m_Right = new CNode<T>;
    if (!node->m_Right)
    {
        delete node->m_Left;
        return false;
    }
    node->m_Value = value;
    node->m_IsEmpty = false;
    return true;
}
```

```cpp
template<class T>
inline bool CBinarySearchTree<T>::Remove(T value)
{
    CNode<T>* deleteNode = GetDeleteNode(m_Root, value);
    if (!deleteNode) return false;
    return DeleteNode(deleteNode);
}
```

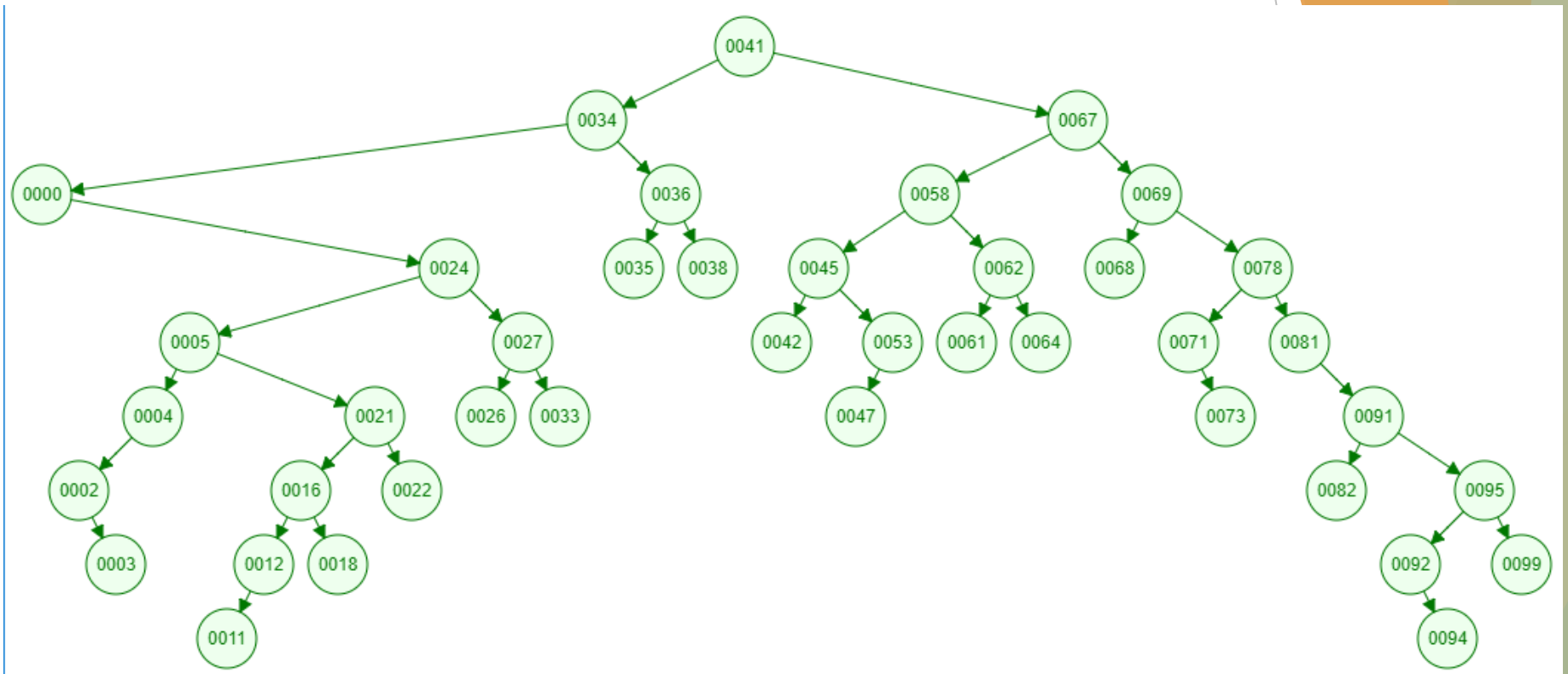
```cpp
template<class T>
inline CNode<T>* CBinarySearchTree<T>::GetDeleteNode(CNode<T>* root, T value)
{
    if (root->m_IsEmpty) return NULL;
    if (root->m_Value == value)
        return root;
    else if (root->m_Value > value)
        return GetDeleteNode(root->m_Left, value);
    else if (root->m_Value < value)
        return GetDeleteNode(root->m_Right, value);
    else
        return NULL;
}
```

```cpp
template<class T>
inline CNode<T>* CBinarySearchTree<T>::GetMaxInLeftTree(CNode<T>* root)
{
    CNode<T>* now = root;
    while (!now->m_Right->m_IsEmpty)
        now = now->m_Right;
    return now;
}
```

```cpp
template<class T>
inline bool CBinarySearchTree<T>::DeleteNode(CNode<T>* node)
{
    if (node->m_Left->m_IsEmpty && node->m_Right->m_IsEmpty)
    {
        delete node->m_Left;
        node->m_Left = NULL;
        delete node->m_Right;
        node->m_Right = NULL;
        node->m_IsEmpty = true;
    }
    else if (!node->m_Left->m_IsEmpty && node->m_Right->m_IsEmpty)
    {
        T regValue = node->m_Value;
        node->m_Value = node->m_Left->m_Value;
        if (!DeleteNode(node->m_Left))
        {
            node->m_Value = regValue;
            return false;
        }
    }
    else if (node->m_Left->m_IsEmpty && !node->m_Right->m_IsEmpty)
    {
        T regValue = node->m_Value;
        node->m_Value = node->m_Right->m_Value;
        if (!DeleteNode(node->m_Right))
        {
            node->m_Value = regValue;
            return false;
        }
    }
```

```cpp
    else
    {
        CNode<T>* now = GetMaxInLeftTree(node->m_Left);
        T regValue = node->m_Value;
        node->m_Value = now->m_Value;
        if (!DeleteNode(now))
        {
            node->m_Value = regValue;
            return false;
        }
    }
    return true;
}
```

與樹高相關，樹的高度愈平均，搜尋效率好，若樹高相差過大，則搜尋效率不好，如果偏向一邊，則為歪斜樹



作業：計算樹高(不用交，當練習)

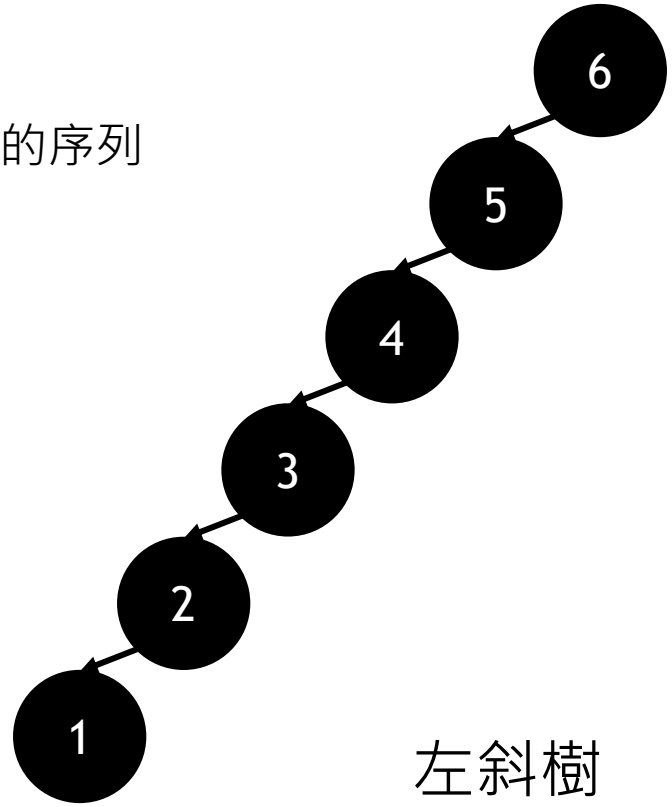
- 請新增一個函式，會傳出樹的高度。

作業：層序走訪

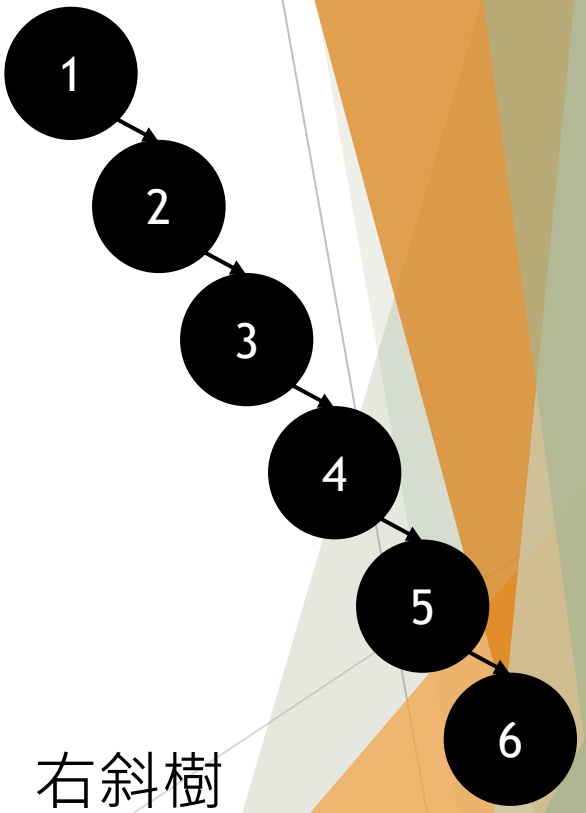
- 依照樹的階層，走訪整棵樹: 依序秀出層序的元素，再往下到下一層。
- (那層看完的在看下層，不留空比較好做)

歪斜樹

有序的序列



左斜樹



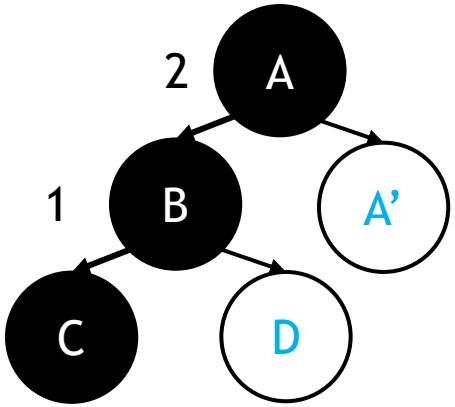
右斜樹

(樹的)高度平衡樹(AVL Tree)

- 自平衡二元搜尋樹。
- AVL樹中任何節點的兩個子樹(左、右子樹)的高度最大差別為一。
- 搜尋、插入和刪除在平均和最壞情況下都是O(log2 n)。(n是元素數量)
- 增加和刪除可能需要通過一次或多次樹”旋轉”來重新平衡這個樹。

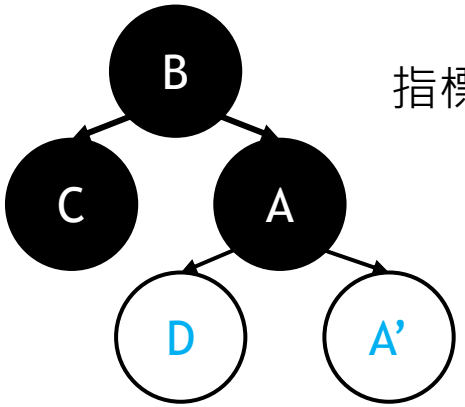
樹旋轉(1/2)

左子樹樹高減右子樹樹高為平衡因子，
為正看左腳，負看右腳

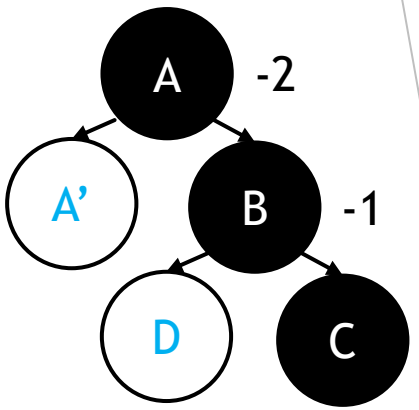


LL

高度差都為正，
左左型(++)

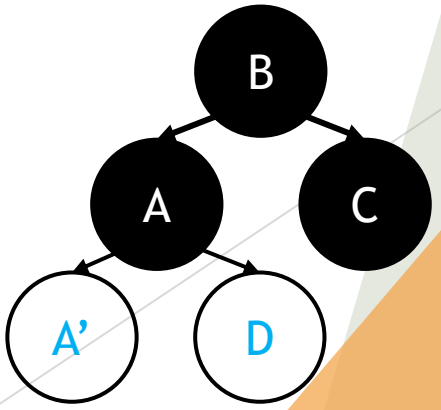


指標搬移

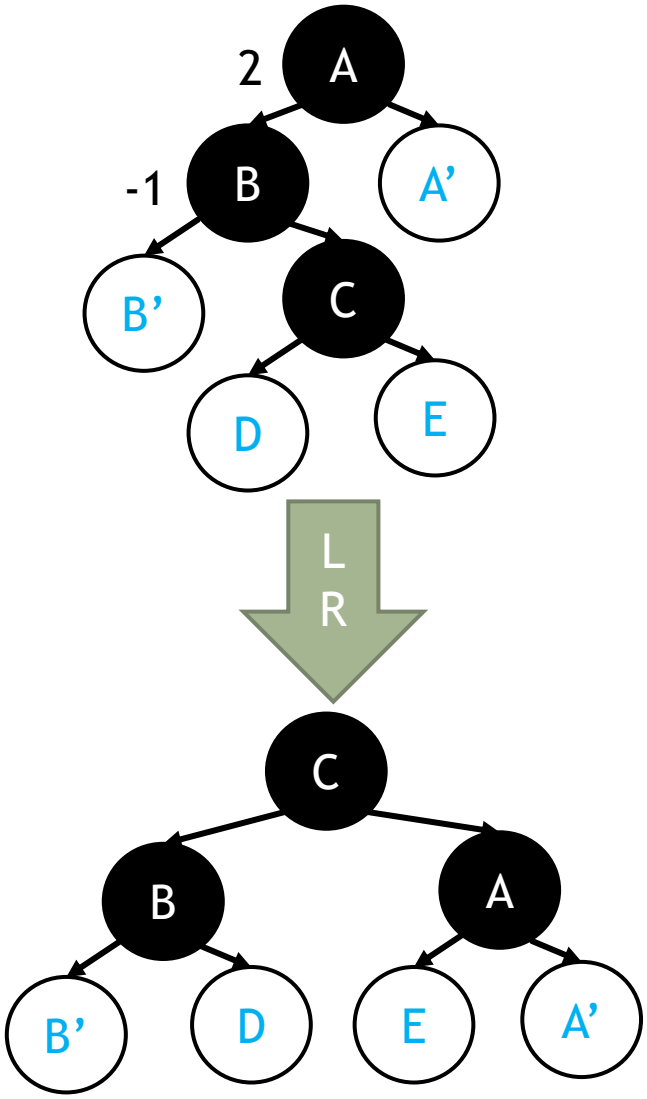


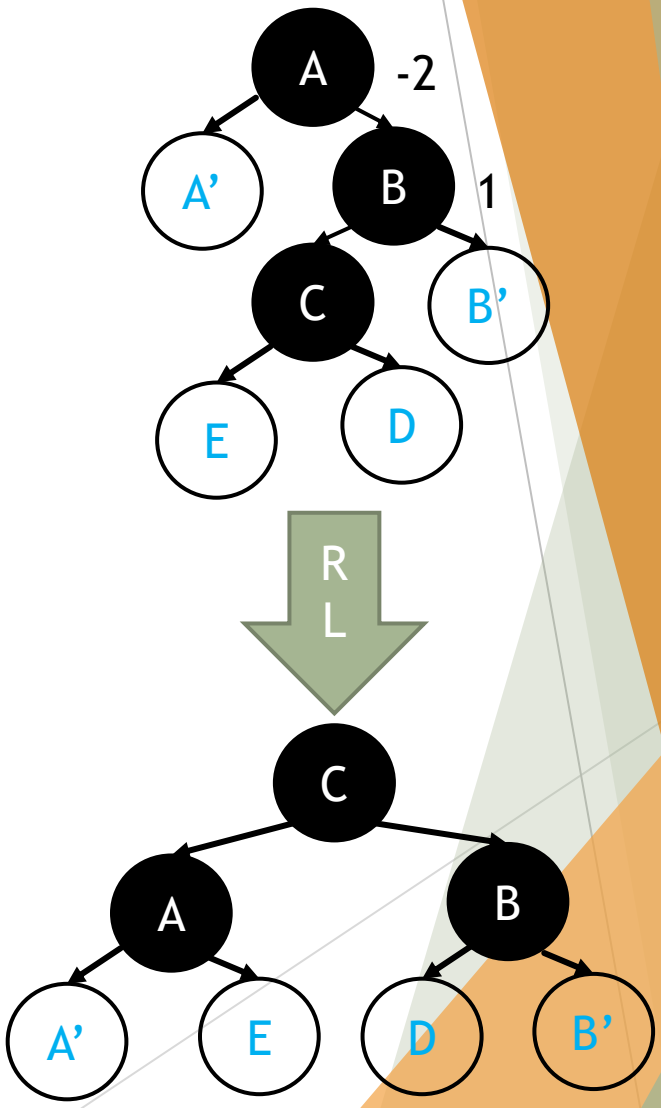
RR

高度差都為正，
右右型(--)

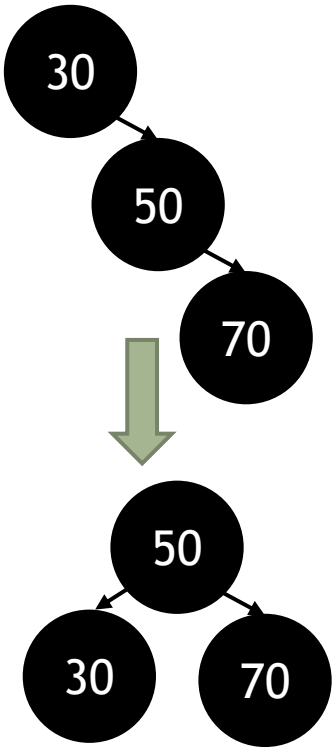


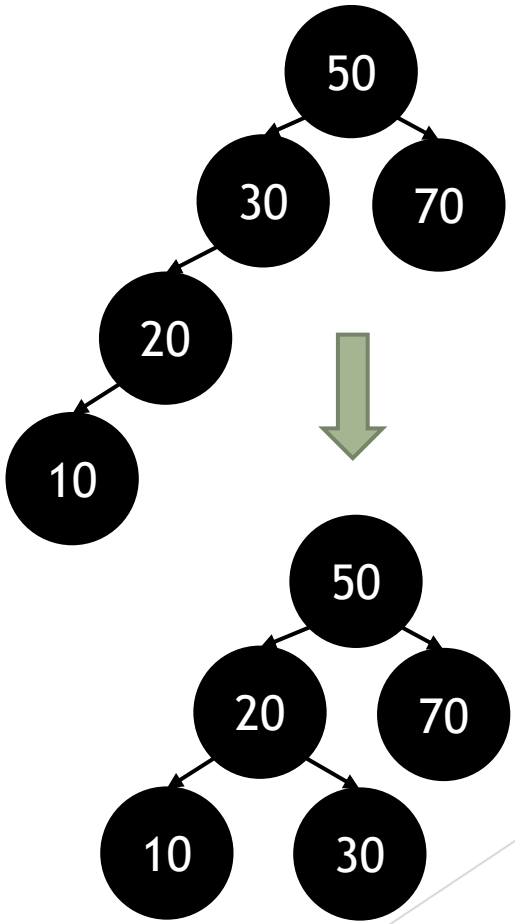
樹旋轉(2/2)



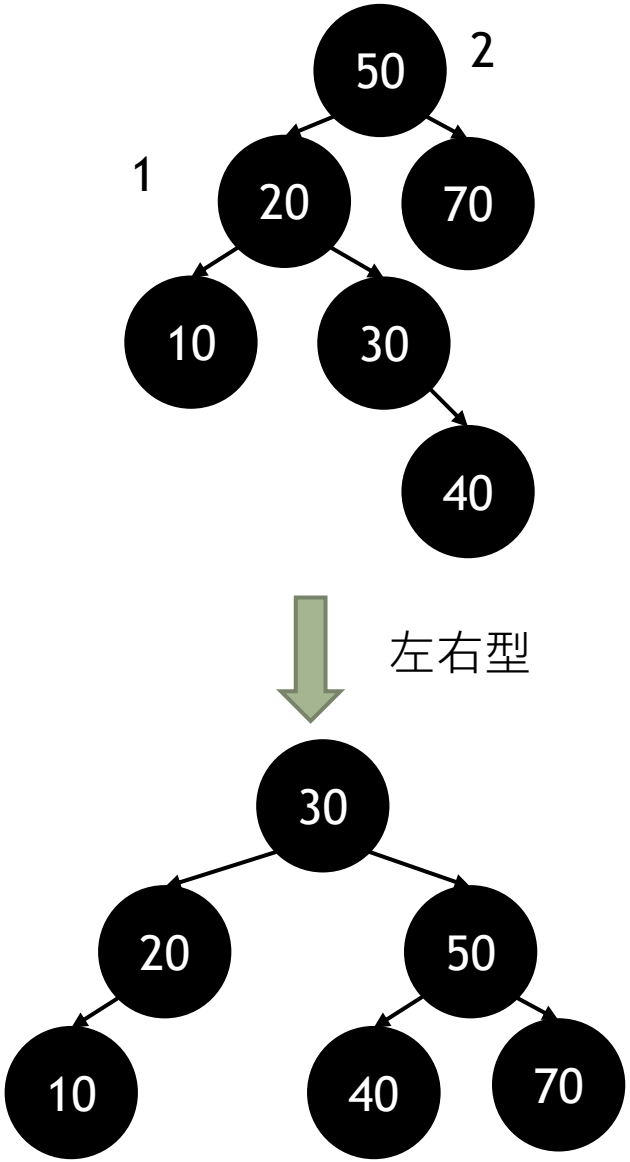


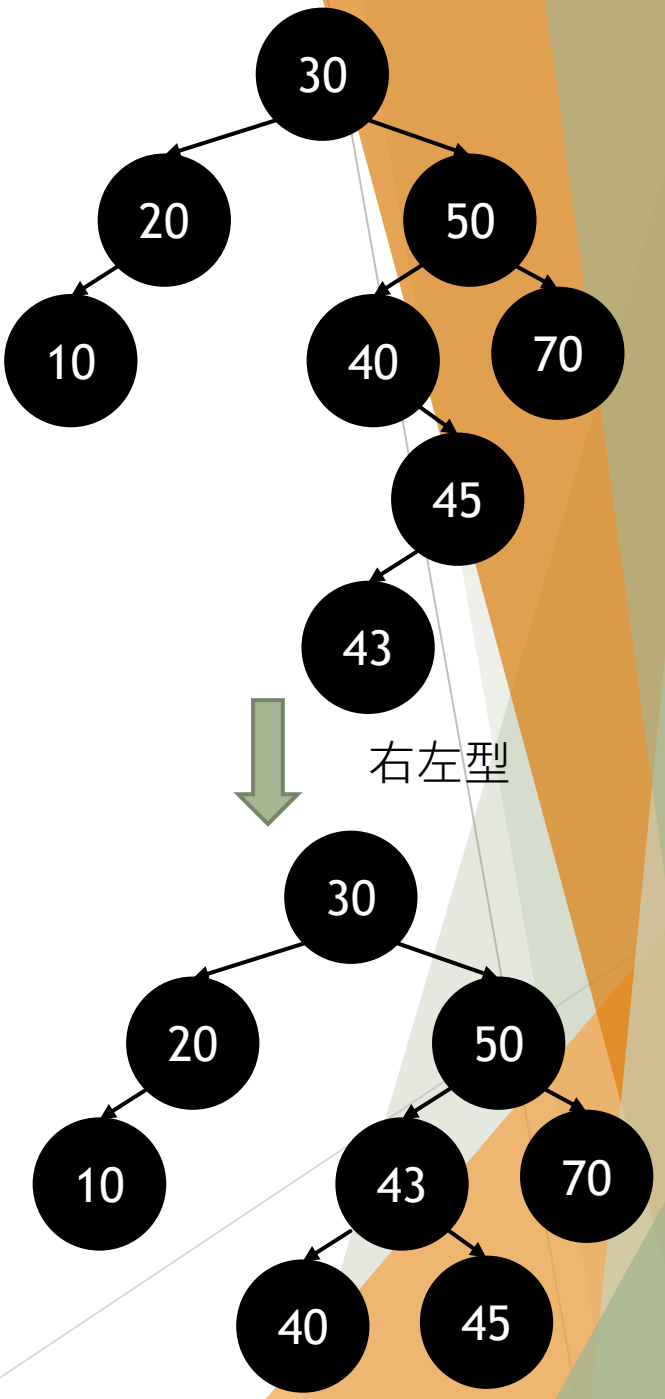
實例：30, 50, 70, 20, 10, 40, 45, 43



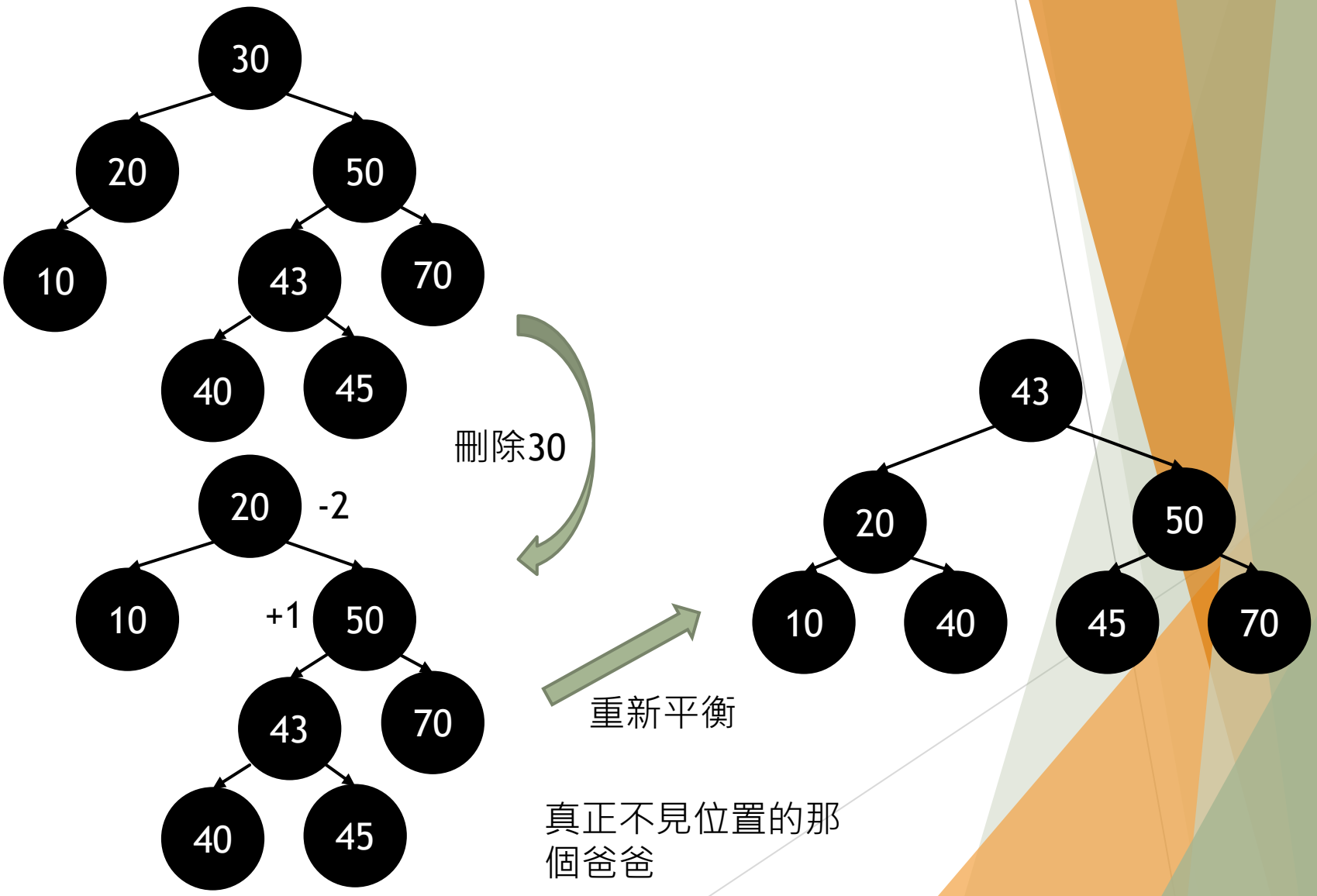


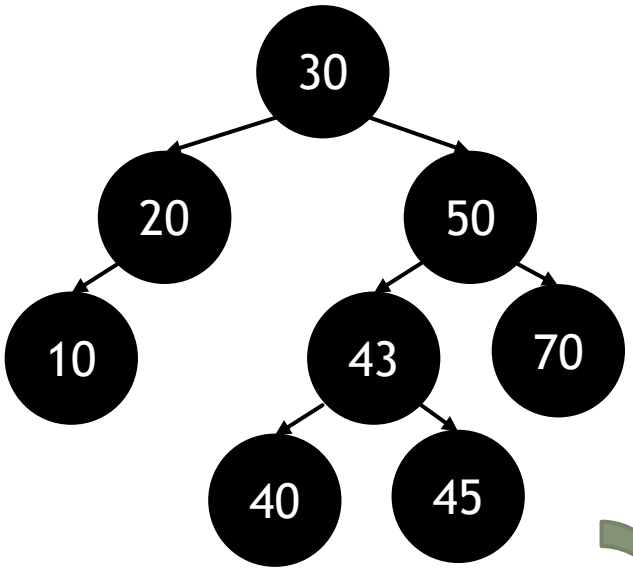
往上追，先轉，轉完後平衡了，就不用再轉了


50
2
1
20
70
10
30
40
左右型
30
20
50
10
40
70

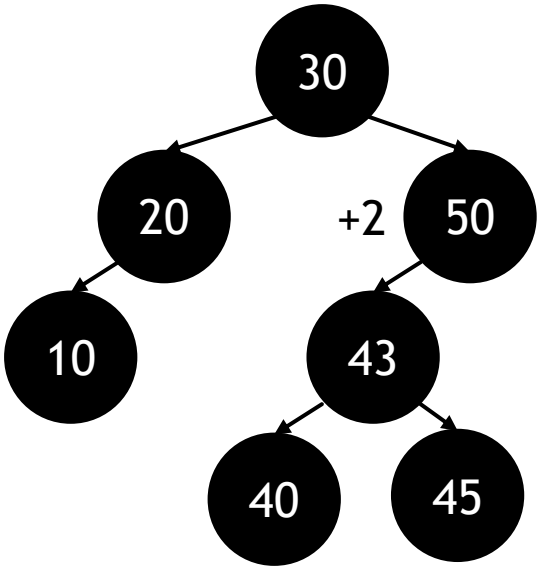

30
20
50
10
40
70
45
43
右左型
30
20
50
10
43
70
40
45

刪除結點

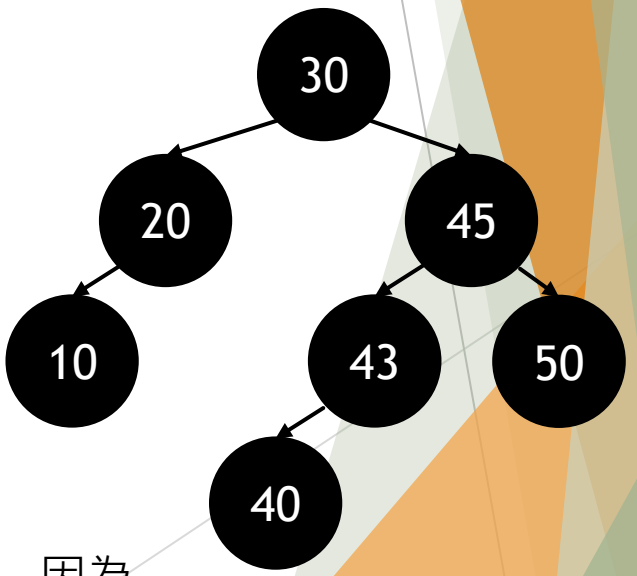

30
20
50
10
43
70
40
45
刪除30
20
-2
10
+1
50
43
70
40
45
重新平衡
43
20
50
10
40
45
70
真正不見位置的那個爸爸



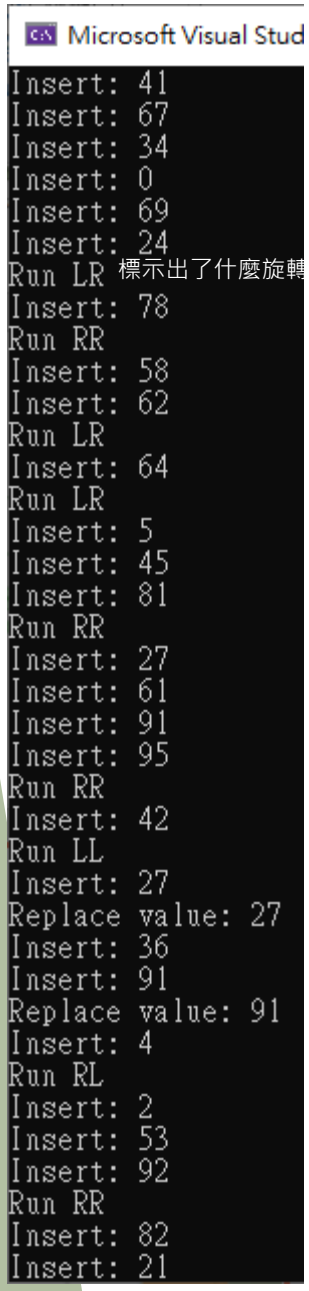
刪除70



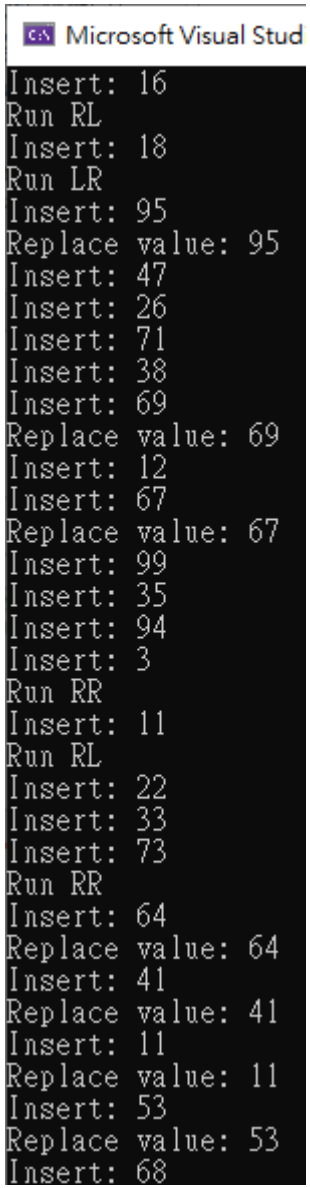
重新平衡



左左或左右都可，因為兒子的平衡因子為0，挑簡單的做



標示出了什麼旋轉



中序一定是一個有序的結果

```
0, 2, 3, 4, 5, 11, 12, 16, 18, 21, 22, 24, 26, 27, 33, 34, 35, 36, 38, 41,
42, 45, 47, 53, 58, 61, 62, 64, 67, 68, 69, 71, 73, 78, 81, 82, 91, 92, 94,
 95, 99,
41, 16, 4, 2, 0, 3, 11, 5, 12, 24, 21, 18, 22, 34, 27, 26, 33, 36, 35, 38,
67, 58, 45, 42, 53, 47, 62, 61, 64, 91, 78, 71, 69, 68, 73, 81, 82, 95, 92,
 94, 99,
0, 3, 2, 5, 12, 11, 4, 18, 22, 21, 26, 33, 27, 35, 38, 36, 34, 24, 16, 42,
47, 53, 45, 61, 64, 62, 58, 68, 69, 73, 71, 82, 81, 78, 94, 92, 99, 95, 91,
 67, 41,
After remove 42:
Run RL
0, 2, 3, 4, 5, 11, 12, 16, 18, 21, 22, 24, 26, 27, 33, 34, 35, 36, 38, 41,
45, 47, 53, 58, 61, 62, 64, 67, 68, 69, 71, 73, 78, 81, 82, 91, 92, 94, 95,
 99,
41, 16, 4, 2, 0, 3, 11, 5, 12, 24, 21, 18, 22, 34, 27, 26, 33, 36, 35, 38,
67, 58, 47, 45, 53, 62, 61, 64, 91, 78, 71, 69, 68, 73, 81, 82, 95, 92, 94,
 99,
0, 3, 2, 5, 12, 11, 4, 18, 22, 21, 26, 33, 27, 35, 38, 36, 34, 24, 16, 45,
53, 47, 61, 64, 62, 58, 68, 69, 73, 71, 82, 81, 78, 94, 92, 99, 95, 91, 67,
 41,
After remove 99:
Run LR
0, 2, 3, 4, 5, 11, 12, 16, 18, 21, 22, 24, 26, 27, 33, 34, 35, 36, 38, 41,
45, 47, 53, 58, 61, 62, 64, 67, 68, 69, 71, 73, 78, 81, 82, 91, 92, 94, 95,

41, 16, 4, 2, 0, 3, 11, 5, 12, 24, 21, 18, 22, 34, 27, 26, 33, 36, 35, 38,
67, 58, 47, 45, 53, 62, 61, 64, 91, 78, 71, 69, 68, 73, 81, 82, 94, 92, 95,

0, 3, 2, 5, 12, 11, 4, 18, 22, 21, 26, 33, 27, 35, 38, 36, 34, 24, 16, 45,
53, 47, 61, 64, 62, 58, 68, 69, 73, 71, 82, 81, 78, 92, 95, 94, 91, 67, 41,
```

```cpp
#include "CAVLTree.h"

int main()
{
    int i;
    int a[] = { 41, 67, 34, 0, 69, 24, 78, 58, 62, 64,
        5, 45, 81, 27, 61, 91, 95, 42, 27, 36,
        91, 4, 2, 53, 92, 82, 21, 16, 18, 95,
        47, 26, 71, 38, 69, 12, 67, 99, 35, 94,
        3, 11, 22, 33, 73, 64, 41, 11, 53, 68 };
    CAVLTree<int> avl;
    for (i = 0; i < sizeof(a) / sizeof(int); ++i)
    {
        std::cout << "Insert: " << a[i] << std::endl;
        avl.Insert(a[i]);
    }
    avl.Inorder();
    std::cout << std::endl;
    avl.Preorder();
    std::cout << std::endl;
    avl.Posorder();
    std::cout << std::endl;
    std::cout << "After remove 42:" << std::endl;
    avl.Remove(42);
    avl.Inorder();
    std::cout << std::endl;
    avl.Preorder();
    std::cout << std::endl;
    avl.Posorder();
    std::cout << std::endl;
    std::cout << "After remove 99:" << std::endl;
    avl.Remove(99);
    avl.Inorder();
    std::cout << std::endl;
    avl.Preorder();
    std::cout << std::endl;
    avl.Posorder();
    std::cout << std::endl;
    return 0;
}
```

```cpp
#pragma once
#include <iostream>

template<class T>
class CNode
{
public:
    CNode<T>* m_Left;
    CNode<T>* m_Right;
    CNode<T>* m_Father;
    T m_Value;
    bool m_IsEmpty;

    CNode(CNode<T>* father = NULL)
        : m_Value()
        , m_Left(NULL)
        , m_Right(NULL)
        , m_IsEmpty(true) {
        m_Father = father;
    };
    ~CNode() {};
};
```

函式參數預設值

有爸爸的情況下才會new一個點，那root的爸爸嚴格說起來是NULL

```cpp
enum RotationType
{
    LL, RR, LR, RL
};

template <class T>
class CAVLTree
{
public:

    CAVLTree();
    ~CAVLTree();
    bool Insert(T value);
    bool Remove(T value);
    void Inorder(CNode<T>* root = NULL);
    void Preorder(CNode<T>* root = NULL);
    void Posorder(CNode<T>* root = NULL);

private:
    CNode<T>* m_Root;

    void DeleteTree(CNode<T>* root);
    CNode<T>* GetEmptyNode(CNode<T>* root, T value);
    CNode<T>* GetDeleteNode(CNode<T>* root, T value);
    bool SetNode(CNode<T>* node, T value);
    int GetBalenceFactor(CNode<T>* node);
    int GetTreeHeight(CNode<T>* node);
    bool DeleteNode(CNode<T>* node);
    void TraceBalencePath(CNode<T>* node);
    void TreeRotation(CNode<T>* node, RotationType type);
    CNode<T>* GetMaxInLeftTree(CNode<T>* root);

};
```

```cpp
template<class T>
inline CAVLTree<T>::CAVLTree()
{
    m_Root = new CNode<T>;
}

template<class T>
inline CAVLTree<T>::~CAVLTree()
{
    DeleteTree(m_Root);
}

template<class T>
inline bool CAVLTree<T>::Insert(T value)
{
    CNode<T>* insertNode = GetEmptyNode(m_Root, value);
    if (!insertNode) return false;
    if (!SetNode(insertNode, value))
        return false;
    TraceBalencePath(insertNode);
    return true;
}
```

```cpp
template<class T>
inline void CAVLTree<T>::DeleteTree(CNode<T>* root)
{
    if (root == NULL)
        return;
    DeleteTree(root->m_Left);
    DeleteTree(root->m_Right);
    delete root;
}
```

跟剛剛一樣

```cpp
template<class T>
inline CNode<T>* CAVLTree<T>::GetEmptyNode(CNode<T>* root, T value)
{
    if (root->m_IsEmpty) return root;
    if (root->m_IsEmpty == false && value == root->m_Value)
    {
        std::cout << "Replace value: " << value << std::endl;
        return NULL;
    }
    if (value < root->m_Value)
    {
        if (root->m_Left->m_IsEmpty)
            return root->m_Left;
        else
            return GetEmptyNode(root->m_Left, value);
    }
    else if (value > root->m_Value)
    {
        if (root->m_Right->m_IsEmpty)
            return root->m_Right;
        else
            return GetEmptyNode(root->m_Right, value);
    }
    return NULL;
}
```

```cpp
template<class T>
inline bool CAVLTree<T>::SetNode(CNode<T>* node, T value)
{
    if (!node) return false;
    node->m_Left = new CNode<T>(node);
    if (!node->m_Left) return false;
    node->m_Right = new CNode<T>(node);
    if (!node->m_Right)
    {
        delete node->m_Left;
        return false;
    }
    node->m_Value = value;
    node->m_IsEmpty = false;
    return true;
}
```

```cpp
template<class T>
inline void CAVLTree<T>::TraceBalencePath(CNode<T>* node)
{
    int balenceFactor;
    CNode<T>* now = node;
    while (now) {
        balenceFactor = GetBalenceFactor(now);
        if (abs(balenceFactor) >= 2)
            break;
        now = now->m_Father;
    }
    if (now == NULL) return;
    RotationType type;
    if (balenceFactor >= 2 && GetBalenceFactor(now->m_Left) >= 0)
        type = RotationType::LL;
    else if (balenceFactor >= 2 && GetBalenceFactor(now->m_Left) < 0)
        type = RotationType::LR;
    else if (balenceFactor <= -2 && GetBalenceFactor(now->m_Right) > 0)
        type = RotationType::RL;
    else if (balenceFactor <= -2 && GetBalenceFactor(now->m_Right) <= 0)
        type = RotationType::RR;
    TreeRotation(now, type);
}
```

失去平衡的點　要旋轉的型態

now 是NULL代表
沒有失去平衡

用遞迴方式，先往下探，再往回走，每個點都要走到

```cpp
template<class T>
inline int CAVLTree<T>::GetBalenceFactor(CNode<T>* node)
{
    if (!node) return 0;
    return GetTreeHeight(node->m_Left)
        - GetTreeHeight(node->m_Right);
}

template<class T>
inline int CAVLTree<T>::GetTreeHeight(CNode<T>* node)
{
    if (node == NULL) return 0;
    int leftTreeHeight = GetTreeHeight(node->m_Left);
    int rightTreeHeight = GetTreeHeight(node->m_Right);
    if (leftTreeHeight >= rightTreeHeight)
        return leftTreeHeight + 1;
    else return rightTreeHeight + 1;
}
```

```cpp
template<class T>
inline void CAVLTree<T>::TreeRotation(CNode<T>* node, RotationType type)
{
    CNode<T>* reg, *B, *C, *CL, *CR;
    CNode<T>* A = node;
    switch (type)
    {
    case RotationType::LL:
        std::cout << "Run LL" << std::endl;
        B = A->m_Left;
        if (!A->m_Father) m_Root = B;
        else
            A->m_Father->m_Left == A ?
            A->m_Father->m_Left = B : A->m_Father->m_Right = B;
        reg = B->m_Right;
        B->m_Right = A;
        A->m_Left = reg;
        B->m_Father = A->m_Father;
        A->m_Father = B;
        if (reg)reg->m_Father = A;
        break;
```

B取代A

確定A的的地位再去被B取代

由下往上串，A->m_Father這個指標還沒改

```cpp
    case RotationType::LR:
        std::cout << "Run LR" << std::endl;
        B = A->m_Left;
        C = B->m_Right;
        if (!A->m_Father) m_Root = C;
        else
            A->m_Father->m_Left == A ?
            A->m_Father->m_Left = C : A->m_Father->m_Right = C;
        CL = C->m_Left;
        CR = C->m_Right;
        C->m_Right = A;
        C->m_Left = B;
        A->m_Left = CR;
        B->m_Right = CL;
        C->m_Father = A->m_Father;
        A->m_Father = C;
        B->m_Father = C;
        if (CR) CR->m_Father = A;
        if (CL) CL->m_Father = B;
        break;
```

```cpp
case RotationType::RR:
    std::cout << "Run RR" << std::endl;
    B = A->m_Right;
    if (!A->m_Father) m_Root = B;
    else
        A->m_Father->m_Left == A ?
        A->m_Father->m_Left = B : A->m_Father->m_Right = B;
    reg = B->m_Left;
    B->m_Left = A;
    A->m_Right = reg;
    B->m_Father = A->m_Father;
    A->m_Father = B;
    if (reg)reg->m_Father = A;
    break;
```

```cpp
case RotationType::RL:
    std::cout << "Run RL" << std::endl;
    B = A->m_Right;
    C = B->m_Left;
    if (!A->m_Father) m_Root = C;
    else
        A->m_Father->m_Left == A ?
        A->m_Father->m_Left = C : A->m_Father->m_Right = C;
    CL = C->m_Left;
    CR = C->m_Right;
    C->m_Left = A;
    C->m_Right = B;
    A->m_Right = CL;
    B->m_Left = CR;
    C->m_Father = A->m_Father;
    A->m_Father = C;
    B->m_Father = C;
    if (CL) CL->m_Father = A;
    if (CR) CR->m_Father = B;
    break;
}
```

```cpp
template<class T>
inline void CAVLTree<T>::Inorder(CNode<T>* root)
{
    if (root == NULL) root = m_Root;
    if (root->m_IsEmpty) return;
    Inorder(root->m_Left);
    std::cout << root->m_Value << ", ";
    Inorder(root->m_Right);
}

template<class T>
inline void CAVLTree<T>::Preorder(CNode<T>* root)
{
    if (root == NULL) root = m_Root;
    if (root->m_IsEmpty) return;
    std::cout << root->m_Value << ", ";
    Preorder(root->m_Left);
    Preorder(root->m_Right);
}

template<class T>
inline void CAVLTree<T>::Posorder(CNode<T>* root)
{
    if (root == NULL) root = m_Root;
    if (root->m_IsEmpty) return;
    Posorder(root->m_Left);
    Posorder(root->m_Right);
    std::cout << root->m_Value << ", ";
}
```

```cpp
template<class T>
inline bool CAVLTree<T>::Remove(T value)
{
    CNode<T>* deleteNode = GetDeleteNode(m_Root, value);
    if (!deleteNode) return false;
    if (!DeleteNode(deleteNode)) return false;
    return true;
}
```

一樣

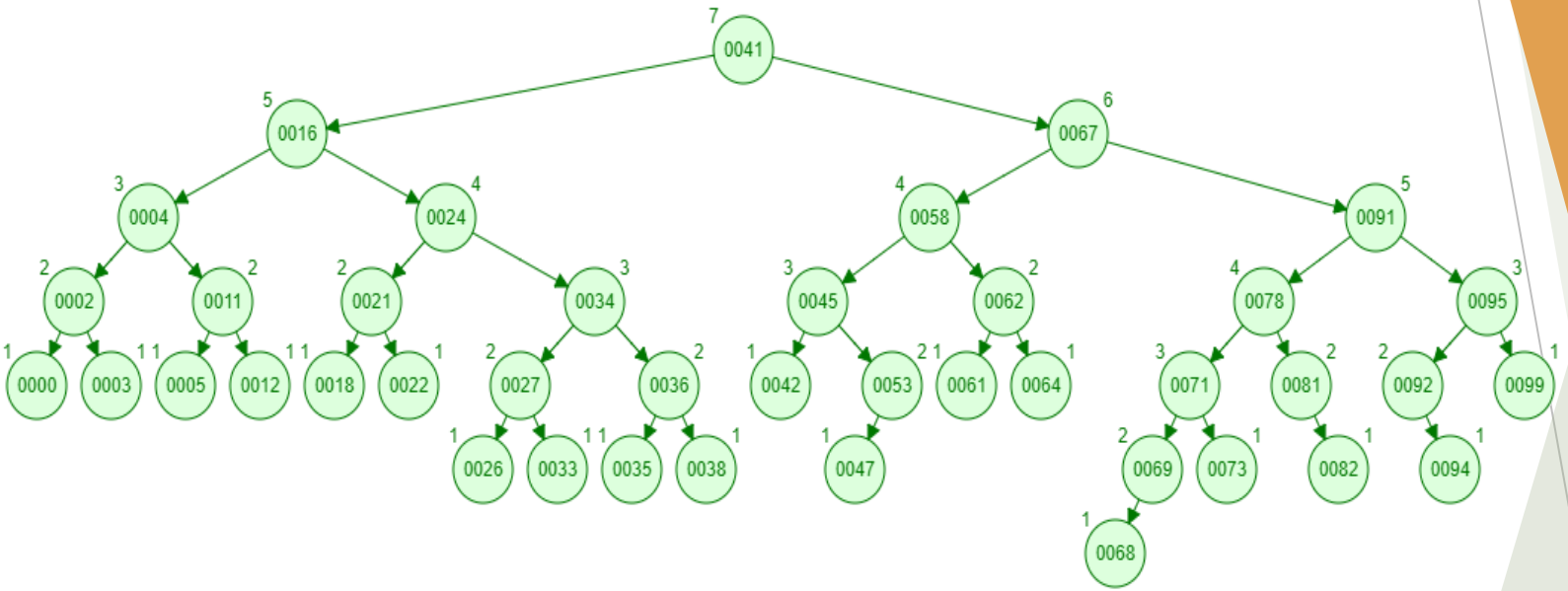
```cpp
template<class T>
inline CNode<T>* CAVLTree<T>::GetDeleteNode(CNode<T>* root, T value)
{
    if (root->m_IsEmpty) return NULL;
    if (root->m_Value == value)
        return root;
    else if (root->m_Value > value)
        return GetDeleteNode(root->m_Left, value);
    else if (root->m_Value < value)
        return GetDeleteNode(root->m_Right, value);
    else
        return NULL;
}
```

```cpp
template<class T>
inline bool CAVLTree<T>::DeleteNode(CNode<T>* node)
{
    if (node->m_Left->m_IsEmpty && node->m_Right->m_IsEmpty)
    {
        delete node->m_Left;
        node->m_Left = NULL;
        delete node->m_Right;
        node->m_Right = NULL;
        node->m_IsEmpty = true;
        TraceBalencePath(node->m_Father);
    }
    else if (!node->m_Left->m_IsEmpty && node->m_Right->m_IsEmpty)
    {
        node->m_Value = node->m_Left->m_Value;
        delete node->m_Left->m_Left;
        node->m_Left->m_Left = NULL;
        delete node->m_Left->m_Right;
        node->m_Left->m_Right = NULL;
        node->m_Left->m_IsEmpty = true;
        TraceBalencePath(node);
    }
```

從自己的爸爸開始找

```cpp
    else if (node->m_Left->m_IsEmpty && !node->m_Right->m_IsEmpty)
    {
        node->m_Value = node->m_Right->m_Value;
        delete node->m_Right->m_Left;
        node->m_Right->m_Left = NULL;
        delete node->m_Right->m_Right;
        node->m_Right->m_Right = NULL;
        node->m_Right->m_IsEmpty = true;
        TraceBalencePath(node);
    }
    else
    {
        CNode<T>* now = GetMaxInLeftTree(node->m_Left);
        node->m_Value = now->m_Value;
        delete now->m_Left;
        now->m_Left = NULL;
        delete now->m_Right;
        now->m_Right = NULL;
        now->m_IsEmpty = true;
        TraceBalencePath(now->m_Father);
    }
    return true;
}
```

```
template<class T>
inline CNode<T>* CAVLTree<T>::GetMaxInLeftTree(CNode<T>* root)
{
    CNode<T>* now = root;
    while (!now->m_Right->m_IsEmpty)
        now = now->m_Right;
    return now;
}
```


7
0041
5
0016
6
0067
3
0004
4
0024
4
0058
5
0091
2
0002
2
0011
2
0021
3
0034
3
0045
2
0062
4
0078
3
0095
1
0000
1 1
0003
1 1
0005
1 1
0012
1 1
0018
1
0022
2
0027
2
0036
1
0042
2 1
0053
1 1
0061
1
0064
3
0071
2
0081
2
0092
1
0099
1
0026
1 1
0033
1 1
0035
1
0038
1
0047
2
0069
1
0073
1
0082
1
0094
1
0068

想想看

- 樹的操作如果不使用遞迴應該要怎麼設計？
  - 遞迴的缺點: 會爆掉，跳來跳去，要記錄跳回來的位置(存在stack區)，的回可能弄爆堆疊區，因為課堂的是小型樹，所以不會爆，但一般的情況下，是好幾萬個元素，樹很大，會爆掉。
  - 若不用遞迴，迴圈(for)搭配堆疊去做，for不會一直消耗堆疊，指呼叫一次函式)

樹:紅黑樹、b Tree，圖跟雜湊還沒教，自己去探索。