聽說要會很
多 喔!!

真討厭！

推廣教育資料結構與演算法
Topic 5 樹

Kuan-Teng Liao (廖冠登)

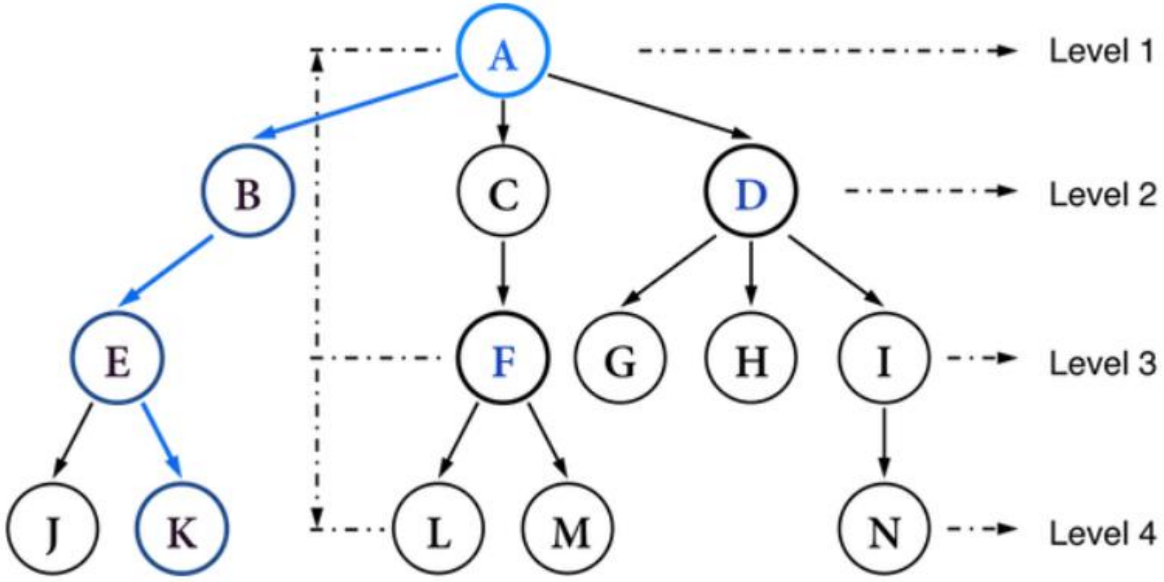
2021/05/15

大綱

簡介	二元樹	二元搜尋樹	AVL樹
定義 應用	建樹 新增節點 刪除節點 尋訪所節點	建樹 新增節點 刪除節點 尋訪所節點	平衡樹

樹(Tree) –1

- 廣泛來說
  - ✓樹可由串列(Linked list)連結構成，意味著有節點與節點間存在的關聯
  - ✓一般來說，建成樹的串列一般來說為雙向(環狀)串列
    - 若為單向，則會無法回溯樹的架構本身
    - E.g.,



樹(Tree) –2

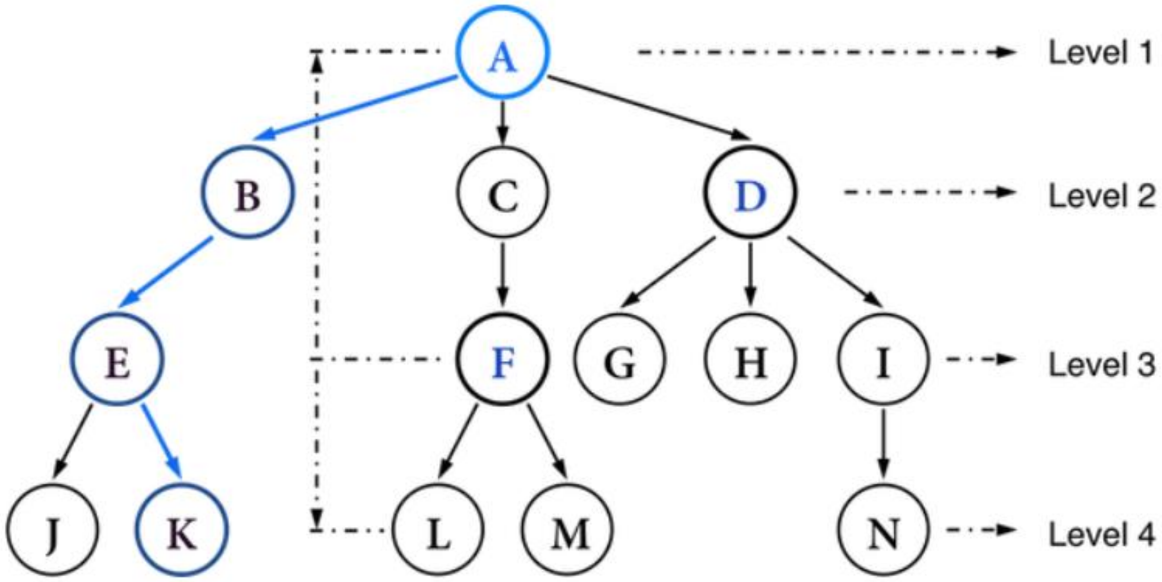
- 嚴謹來說
  - ✓樹是由節點(node)與關聯構成
    - 關聯一般來說只是要為單向指示(實作為雙向)
    - 節點有以下種類
      - 根節點(root)：樹中最上層的節點，也是唯一一個其父節點為NULL的節點
      - 內部節點(internal node)：有子節點或子樹的節點
      - 外部節點(external node)：沒有子節點或子樹的節點
        - 又稱「葉節點」(leaf)

樹(Tree) –3

- 討論樹時需注意的術語
  - ✓父(parent)、子(child)節點：兩節點父為上；孩子為下
  - ✓分岐度(**degree**)：代表某一節點它有幾個分枝(或稱為有幾個子樹)
  - ✓高度(**height**)：為某個葉節點至某個節點的高度(從0開始算)
  - ✓深度(**depth**)：為根節點至某個節點的深度(從0開始算)
  - ✓兄弟姊妹(**siblings**)：同時「指向同一個上一層的父節點」的節點
  - ✓子嗣(**descendant**)：站在某節點上，所有能夠以「parent指向child」的方式找到的節點
  - ✓祖先(**ancestor**)：站在某節點上，所有能夠以「child指向parent」的方式找到的節點
  - ✓階層(**level**)：從根節點開始，可定義出每一層所屬節點(根節點由1開始)
  - ✓路徑(**path**)：從某節點至某節點所經過的所有節點

樹(Tree) –4

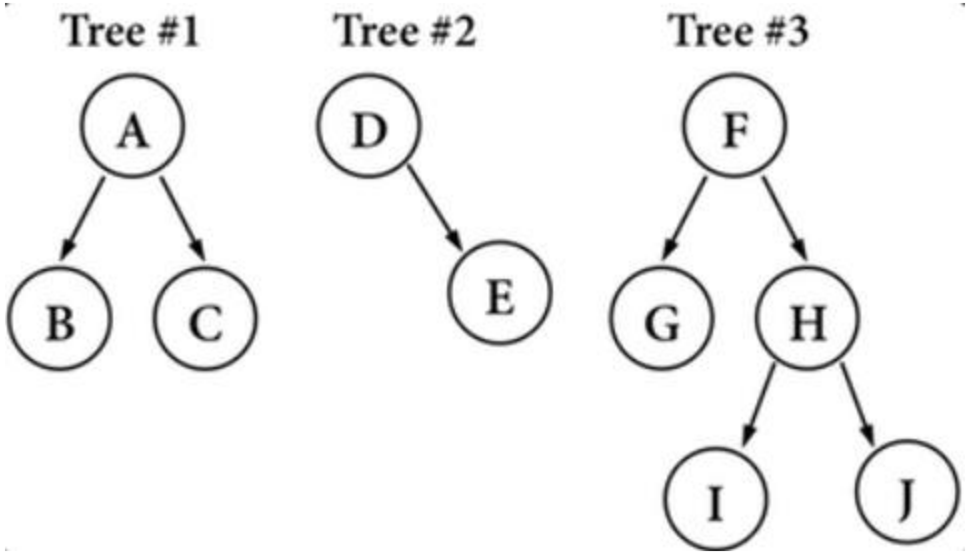
- E.g.,



- A為根節點
- B, C, D, E, F, I 為內部節點
- G, H, J, K, L, M, N為外部節點
- F是C 的child；C是F的parent
- D的分歧度為3
- M至C的高度為2
- A至K的深度為3
- G,H, 和I為siblings
- E, F, G, H, 和I為同一階層
- E的祖先有B和A
- B的子嗣有E,J,和K

樹(Tree) –5

- 樹完整嚴謹定義，可以有兩種方式進行
  - ✓以下列出兩種互相等價的Tree(樹)的定義：
    - Tree是由一個或多個節點所組成的有限集合，並且滿足：
      - 存在且只有一個稱為root(樹根)的節點；
      - 其餘的節點可以分割成任意正整數個(包含零個)互斥(disjoint)的集合：T_1、...、T_n，其中每一個集合也都滿足樹的定義，這些集合又稱為這棵樹的subtree(子樹)。
    - 樹是由一個或多個節點(nodes)/點(vertices)以及邊(edge)所組成，而且沒有形成圈(cycle)的集合(set)。



樹(Tree) –6

- 一般來說要設計樹時，需找出樹中在所有節點中最大的分歧度(degree)
- 緊接著即可定義出每個節點
  - ✓E.g., max degree = 2

main.cpp

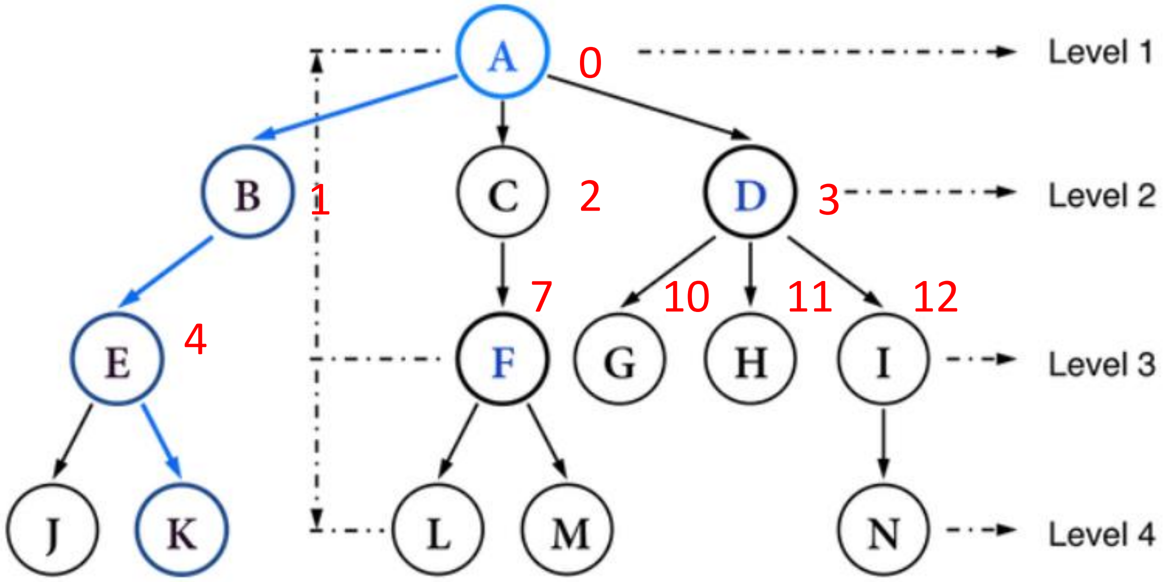
```
#include <iostream>
template <class CL1>
class TNode{
    public: TNode* op_Par;
    public: TNode* op_LCh;
    public: TNode* op_RCh;
    public: CL1 op_Data;
    public: TNode(CL1 op_Data){
        op_Par= nullptr;
        op_LCh= nullptr;
        op_RCh= nullptr;
        this->op_Data = op_Data;
    }
};

template <class CL1>
class Tree{
    public: TNode<CL1>* op_Root;
    public: Tree(){
        op_Root= nullptr;
    }
};

int main(){
    Tree<int> o_Tree;
    return 0;
}
```

樹與陣列的轉換 –1

- 一般來說，樹與陣列是可以互換的，但前提是需要以下技術
  - ✓需用到指標與動態記憶體配置
  - ✓需知道樹中最大的節點分歧度(degree)為何
    - E.g., 圖中為 3
  - ✓再利用分歧度進行階層式分配

父與子索引關聯公式:

$$Child = \{3n+1, 3n+2, 3n+3\}, n \in child\ index$$

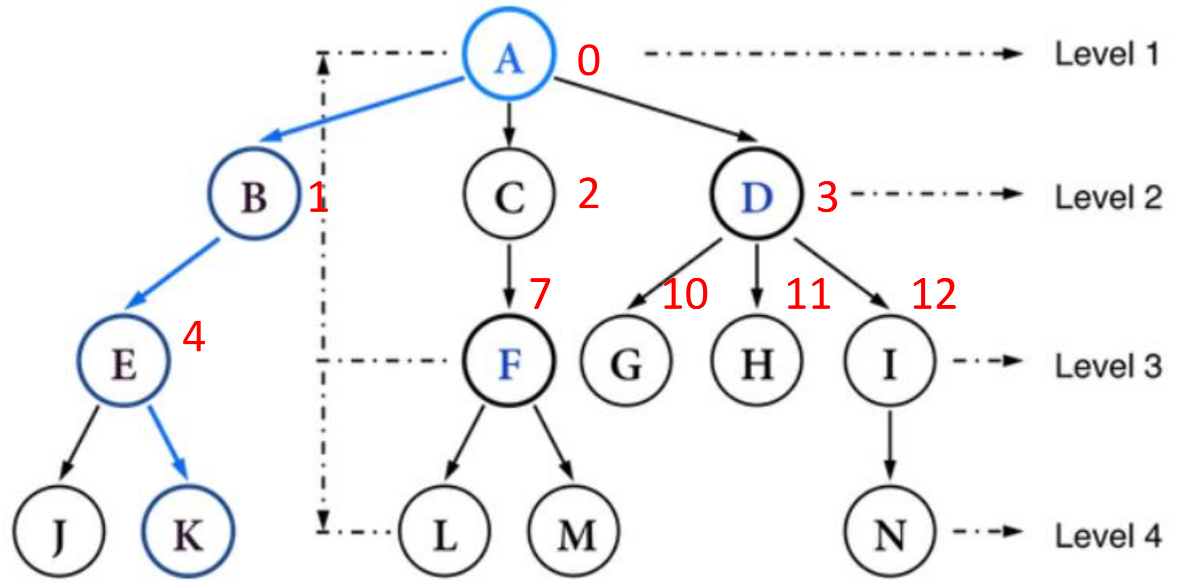


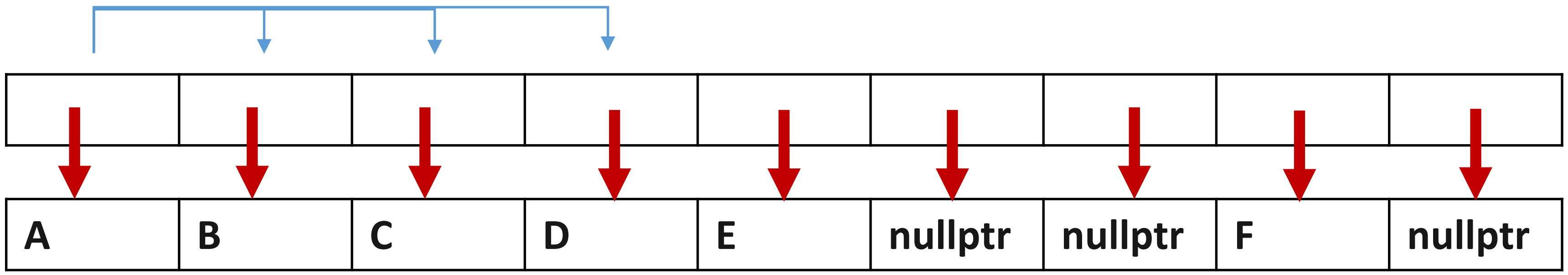
樹與陣列的轉換 –2

- 一般來說，樹與陣列是可以互換的，但前提是需要以下技術 (cont’d)

父與子索引關聯公式:

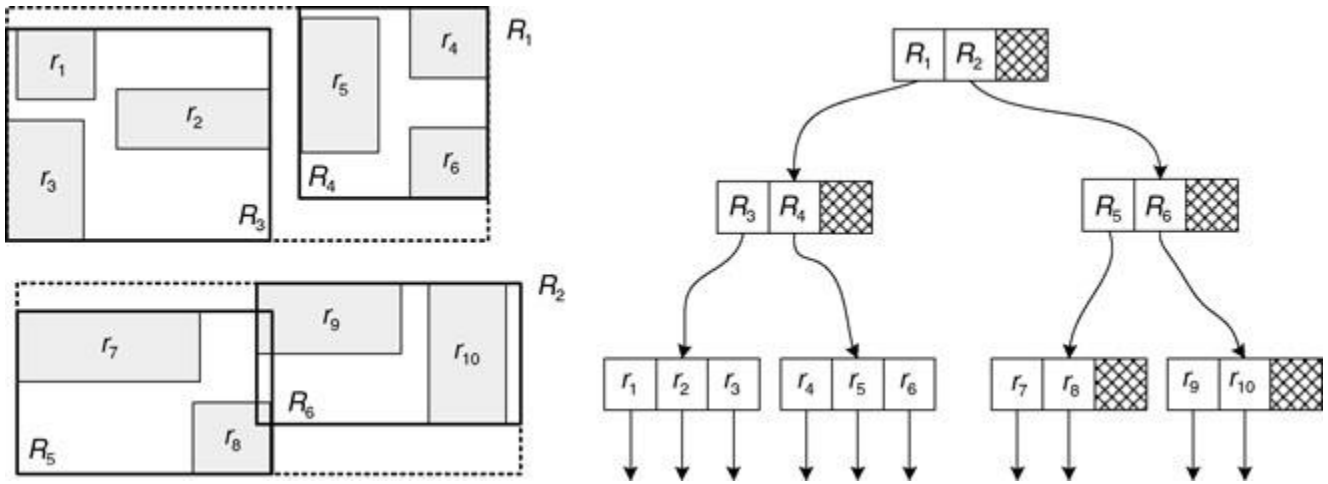
$$Child = \{3n+1, 3n+2, 3n+3\}, n \in index$$





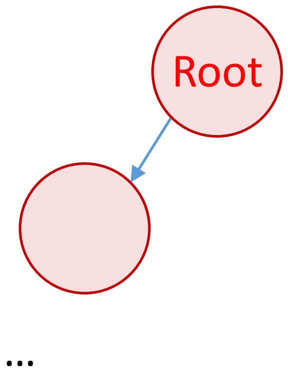
樹的應用 –1

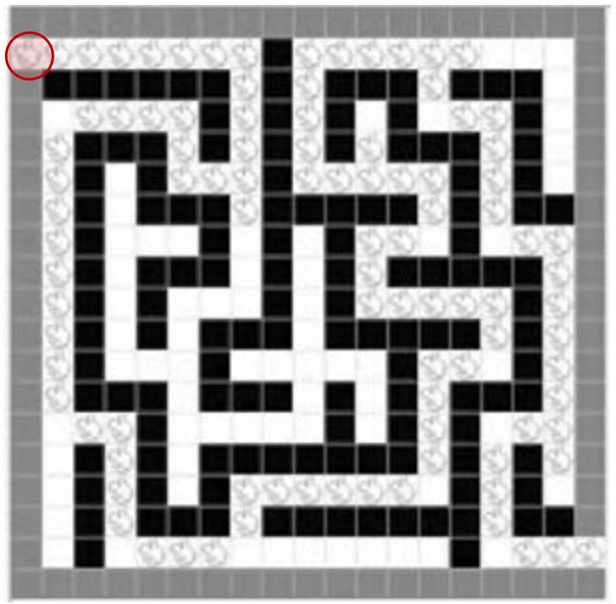
- 一般來說有下來情況基本上會以樹的方式進行
  - ✓當預作項目有明顯的階層結構(hierarchical structure)
    - E.g., 編寫檔案總管；資料夾與資料皆有階層關聯
  - ✓將資料分層並架構其索引(index)，以加速「搜尋」
    - E.g., 將資料所在範圍進行反覆切塊，並架構其階層，使其可以於日後快速索引找到該筆資料



樹的應用 –2

- 一般來說有下來情況基本上會以樹的方式進行(cont’d)
  - ✓與階層(level)或是遞迴或堆疊有關的問題
    - E.g., 老鼠走迷宮
      - 老鼠只能走四近鄰進行選擇，因此要往前走將只有三個方向，但是迷宮裡的路不只有單單一個岔路，因此每走一步可以利用建樹建立子節點，當進入死胡同時，則可以藉由樹進行返回





二元樹(Binary tree) –1

- 在上個主題中，可知道欲設計一樹，需預先知道該樹中每個節點的最大(*k*)分歧度(degree)才得以進行樹的設計
  - ✓因為需設計一節點於建立樹時可適用於所有節點的情況
  - ✓因此當所有節點最大的分歧度值為2(k=2)時，其樹稱為二元樹(binary tree)
    - 定義：
      - 每個節點(node)至多有2個子節點(child nodes)
  - ✓因此可延伸至k元樹
    - 定義：
      - 每個節點(node)至多有*k*個子節點(child nodes)

二元樹(Binary tree) –2

- 在*k*元樹中，除了常見一般的*k*元樹中，有三種較常討論種類
  - ✓由於*k*元素開始較為複雜，因此先假設*k*=2
    - **滿**二元樹 (**Full** binary tree)
    - **完整**二元樹(**Complete** binary tree)
    - **完美**二元樹(**Perfect** binary tree)
- 滿二元樹(Full binary tree)
  - ✓定義：除了一般二元樹定義外，需再有每個節點可有0或2的子節點，排子節點不用從左邊開始排列

二元樹(Binary tree) –3

- 滿二元樹(Full binary tree)(cont’d)
  - ✓E.g.,

```
          18
      /        \
   15            30
 /   \         /   \
40     50    100    40
```

```
             18
           /    \
         15      20
        /  \
      40    50
     /  \
   30    50
```

```
       18
   /        \
40            30
             /  \
          100    40
```

二元樹(Binary tree) –4

- 完整二元樹(Complete binary tree)
  - ✓定義：除了一般二元樹定義外，需再有每個節點可有0至2的子節點，且依階層(level)進行排列，而且**需由左至右排**，在最後一層可允許其上一層父節點(parent)只有一個子節點(child)
  - ✓E.g.,

```
          18
       /      \
     15          30
   /   \       /   \
40      50  100     40
```

```
            18
         /      \
       15          30
     /   \       /   \
  40      50  100     40
 /  \    /
8    7  9
```

二元樹(Binary tree) –5

- 完美二元樹(Perfect binary tree)
  - ✓定義：除了一般二元樹定義外，需再有每個節點可有0或2的子節點，且在每層的階層(level)必須塞滿節點
  - ✓E.g.,

```
          18
       /      \
     15        30
    /  \      /  \
  40    50  100   40
```

```
       18
     /    \
   15      30
```

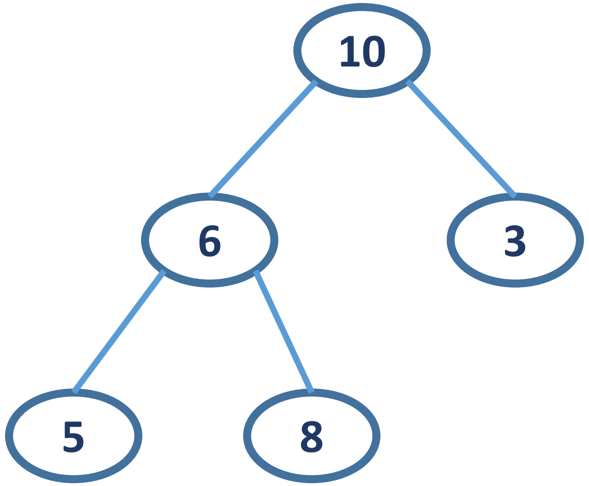
二元樹(Binary tree) –6

- 看起來二元樹中的完整二元樹與完美二元樹最有意義?
  - ✓不，若要使用應該查資料與資料間連結的意義，再決定要使用哪種樹
  - ✓如對滿二元樹而言，其階層代表下方的值依存於上方值產生之後(即關聯性)
    - E.g., 每個數字代表資料夾內有多少檔案且資料夾內會出現0或2個資料夾(隱含階層關係)

```
            18
          /    \
       15       20
      /  \
    40      50
   /    \
30    50
```

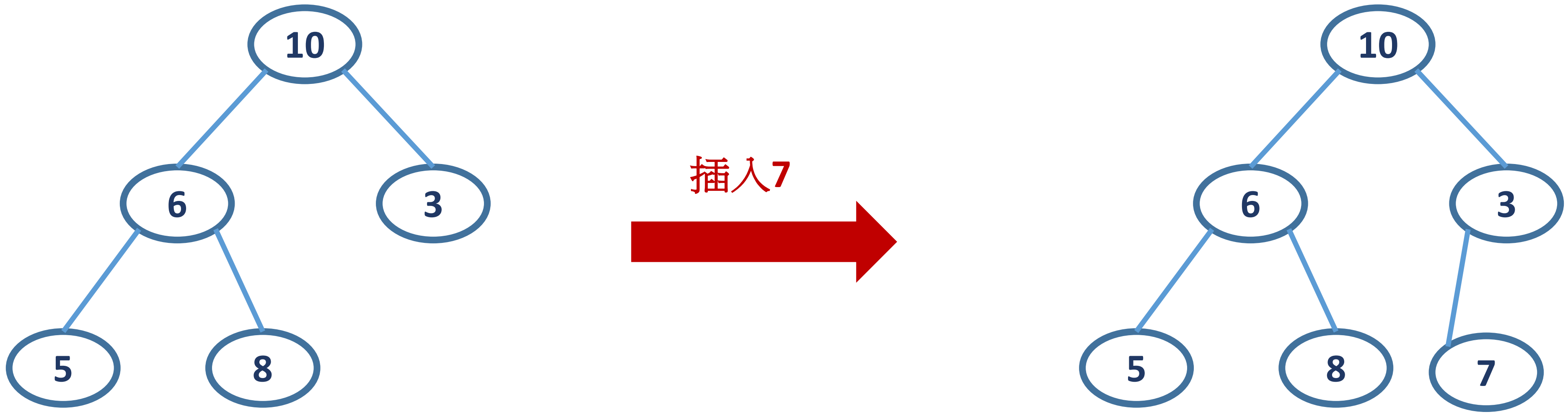
二元樹(Binary tree) – 7

- 一般來說，二元樹的建立基本上是利用「階層低優先且由左至右排序所生成」
  - ✓所以只有建樹下的話，正常會形成完整二元樹(Complete binary tree)



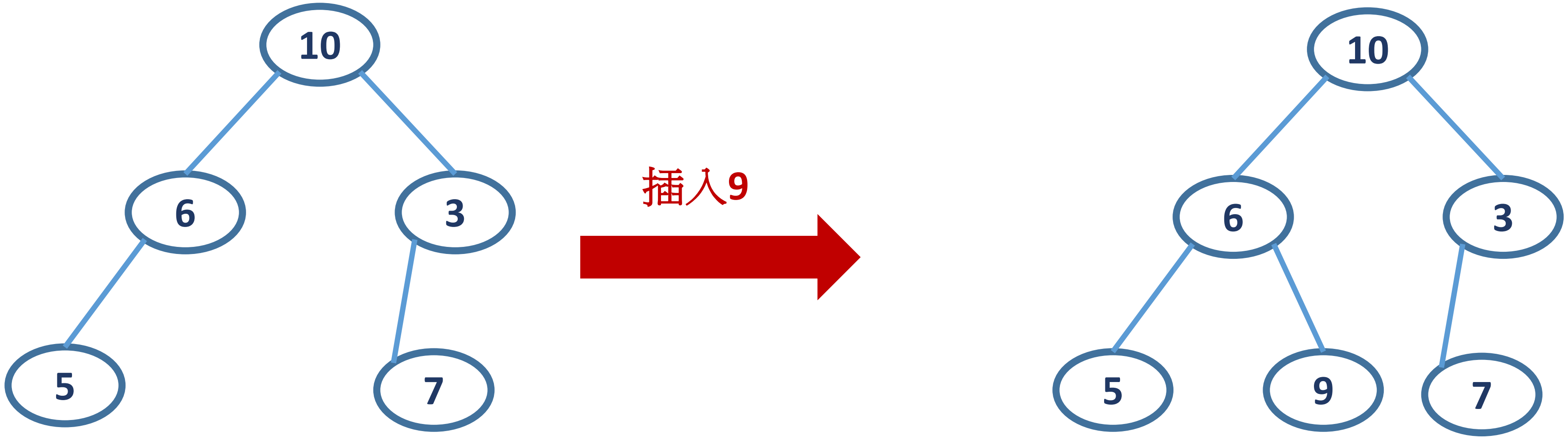
二元樹(Binary tree) –8

- 當建完樹後，一般來說，會有以下操作，包括將元素視為節點「**插入**」於樹中、「**搜尋**」某個節點，「刪除」某個節點，以及「尋訪」所有節點
  - ✓插入功能
    - 若為完整二元樹，則接下來直接插入下一葉節點位置



二元樹(Binary tree) –9

- 當建完樹後，一般來說，會有以下操作，包括將元素視為節點「**插入**」於樹中、「**搜尋**」某個節點，「刪除」某個節點，以及「尋訪」所有節點
  - ✓插入功能 (cont’d)
    - 若不為完整二元樹，則接下來直接插入空置區域



二元樹(Binary tree) –10

- 插入程式碼
  - ✓定義節點部分

main.cpp

```
#include <iostream>
#include <queue>

template<class CL1>
class Node {
    public: CL1 cl1_Field;
    public: Node<CL1>* op_Lt;
    public: Node<CL1>* op_Rt;
    public: Node(){
        op_Lt = nullptr;
        op_Rt = nullptr;
    }
};
```

二元樹(Binary tree) –11

- 插入程式碼
  - ✓定義樹

main.cpp

```cpp

template<class CL1>
class Tree{
    public: Node<CL1>* op_Root;
    public: Tree(CL1 cl1_Data){
        op_Root = nullptr;
        op_Root = fn_CreNode(cl1_Data);
    }
    public: ~Tree(){
        fn_TraInOrder(op_Root, true);
    }
    public: Node<CL1>* fn_CreNode(CL1 cl1_Data){
        Node<CL1>* o_NewNode = new Node<CL1>();
        try{
            if (!o_NewNode) {
                throw "Memory allocation error\n";
            }
        }
        catch(const char* cp_Msg){
            std::cout<< cp_Msg;
            exit(1);
        }
        o_NewNode->cl1_Field = cl1_Data;
        return o_NewNode;
    }
```

二元樹(Binary tree) –12

- 插入程式碼
  - ✓定義樹(cont'd)

main.cpp

```
    public: void fn_InsNode(CL1 cl1_Data){
        std::queue<Node<CL1>*> o_Qu;
        o_Qu.push(op_Root);

        for (;!o_Qu.empty();) {
            Node<CL1>* op_Tmp = o_Qu.front();
            o_Qu.pop();

            if (op_Tmp->op_Lt != nullptr)
                o_Qu.push(op_Tmp->op_Lt);
            else {
                op_Tmp->op_Lt = fn_CreNode(cl1_Data);
                return;
            }

            if (op_Tmp->op_Rt != nullptr)
                o_Qu.push(op_Tmp->op_Rt);
            else {
                op_Tmp->op_Rt = fn_CreNode(cl1_Data);
                return;
            }
        }
    }
```

二元樹(Binary tree) –13

- 插入程式碼
  - ✓定義樹(cont'd)

main.cpp

```cpp
    void fn_TraInOrder(Node<CL1>* op_Tmp, bool b_Mode = false){
        if (op_Tmp == nullptr)
            return;

        fn_TraInOrder(op_Tmp->op_Lt, b_Mode);
        Node<CL1>* op_TmpRt = op_Tmp->op_Rt;

        if(b_Mode == false){
            std::cout << op_Tmp->cl1_Field << ' ';
        }
        else{
            delete op_Tmp;
        }
        fn_TraInOrder(op_TmpRt, b_Mode);
    }
};
```

二元樹(Binary tree) –14

- 插入程式碼
  - ✓定義樹(cont’d)

main.cpp

```cpp
int main(){
    Tree<int> o_Tree(10);
    o_Tree.fn_InsNode(11);
    o_Tree.fn_InsNode(7);
    o_Tree.fn_InsNode(9);
    o_Tree.fn_InsNode(15);
    o_Tree.fn_InsNode(8);

    std::cout << "Inorder traversal before insertion: ";
    o_Tree.fn_TraInOrder(o_Tree.op_Root);
    std::cout << "\n";

    int i_Key = 12;
    o_Tree.fn_InsNode(i_Key);

    std::cout << "Inorder traversal after insertion: ";
    o_Tree.fn_TraInOrder(o_Tree.op_Root);
    std::cout << "\n";

    return 0;
}
```

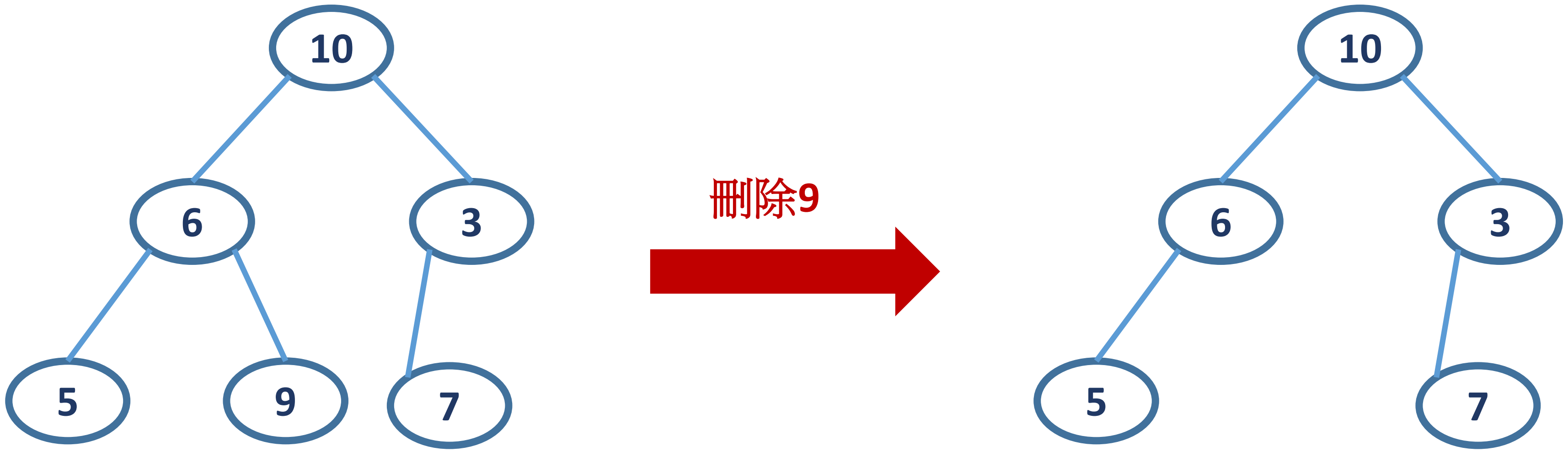
```
E:\CodeWorkShop\CodeBlock\ProjectDsAlg\BinaryTree\bin\Debug\Bi
Inorder traversal before insertion: 9 11 15 10 8 7
Inorder traversal after insertion: 9 11 15 10 8 7 12

Process returned 0 (0x0)   execution time : 0.074 s
Press any key to continue.
```

程式碼網址：https://github.com/altoliaw2/BinaryTree

二元樹(Binary tree) –15

✓刪除功能

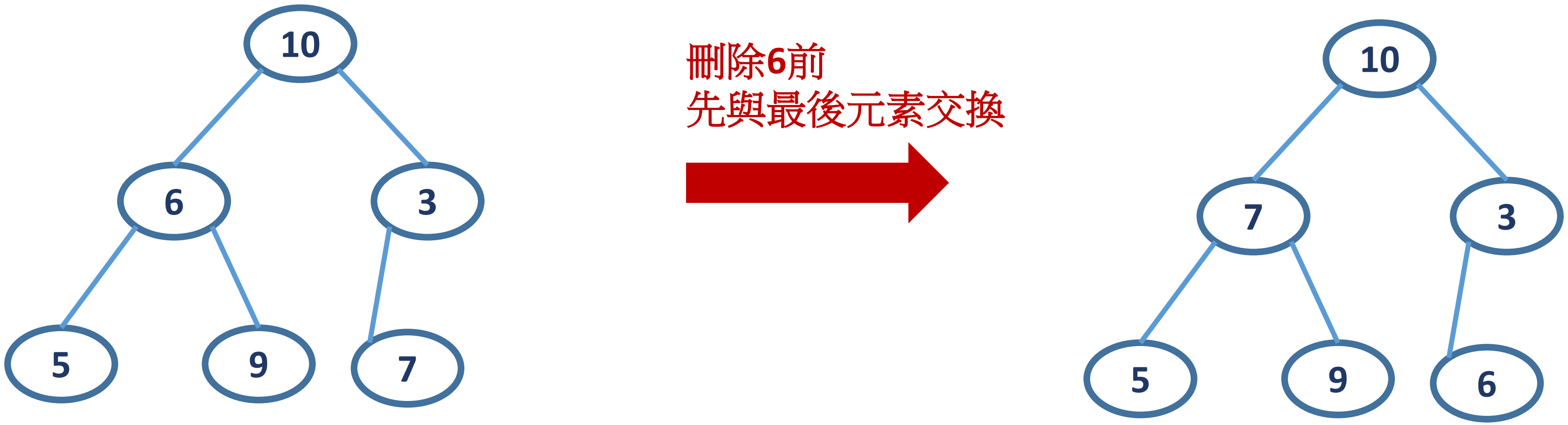
- 請自行練習寫於Tree的class中，方法為void fn_DelNode(CL1 o_Key)
- 可利用上述插入中的*fn_TraInOrder*方法找到該筆節點，然後需判斷該節點狀態
  - 若遇刪除的點為外部節點(葉節點)
    - 則直接刪除



二元樹(Binary tree) –16

✓刪除功能

- 請自行練習寫於Tree的class中，方法為void fn_DelNode(CL1 o_Key)
- 可利用上述插入中的*fn_TraInOrder*方法找到該筆節點，然後需判斷該節點狀態(cont'd)
  - 若不為外部節點
    - 則需找出最後一筆資料並對兩者內容進行交換後，再進行刪除



二元樹(Binary tree) –17

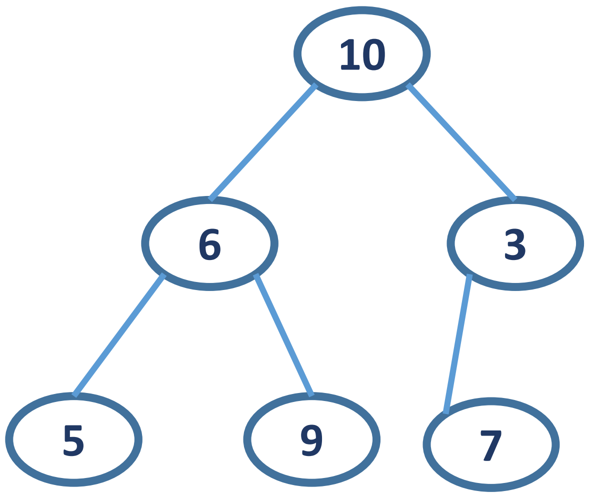
✓刪除功能

- 請自行練習寫於Tree的class中，方法為void fn_DelNode(CL1 o_Key)
- 可利用上述插入中的*fn_TraInOrder*方法找到該筆節點，然後需判斷該節點狀態(cont’d)
  - 若不為外部節點
    - 則需找出最後一筆資料並對兩者內容進行交換後，再進行刪除

二元樹(Binary tree) –18

✓尋訪功能

- 代表著要列出樹中的所有節點
  - 有兩種方法為分別為廣度搜尋尋訪(Breadth-First Search Traversal)與深度搜尋尋訪(Depth First Search Traversal)
- 廣度搜尋尋訪
  - Breadth-First Search Traversal ， BFS。又名 Level Order Traversal
  - 按照各階層由小到大，由左至右進行尋訪
  - 如插入功能中的，*fn_TraIndor(.)*即為BFS

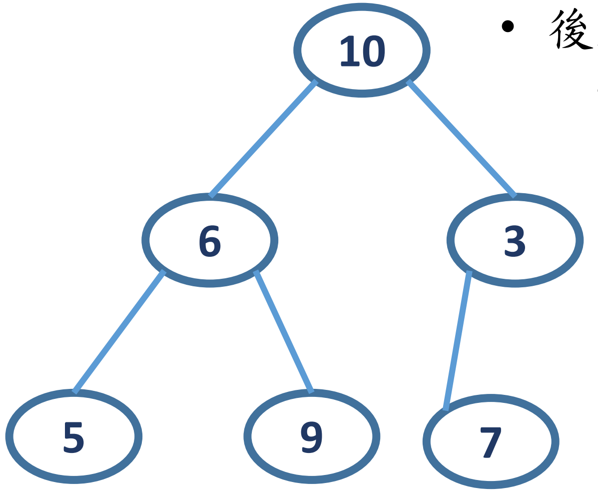


輸出為： 10 6 3 5 9 7

二元樹(Binary tree) –19

✓尋訪功能(cont’d)

- 深度搜尋尋訪
  - Depth First Search Traversal，DFS
  - 顧名思義，就是以同一分支進行尋訪為主
  - 當進行深度搜尋時，其包含了三種方式進行
    - 中序(inorder)
      - 尋訪順序：先左、再中、後右(由下至上)，LVR
    - 前序(prefix order)
      - 尋訪順序：先中、再左、後右，VLR
    - 後序(postfix order)
      - 尋訪順序：先左、在右、後中，LRV



中序輸出為：5 6 9 10 7 3
前序輸出為：10 6 5 9 3 7
後序輸出為：5 9 6 7 3 10

二元樹(Binary tree) –20

- 時間複雜度(以二元樹為例)
  - ✓建樹不用說，因為要把所有資料建成樹，所以今天有n筆資料，建樹必備要把n筆資料建入樹中
    - 因為有使用queue進行紀錄
    - 所以最好的時間複雜度=平均複雜度 =O(n) + $O(\lfloor nlog_2^{n+1} \rfloor)$ = $O(\lfloor nlog_2^{n+1} \rfloor)$
  - ✓從樹中搜尋某節點，
    - 平均情況是—資料建樹為平均最糟情況為完整二元樹情況 $O(\lfloor log_2^{n+1} \rfloor)$
    - 最糟情況—樹為傾斜樹，全部倒向一邊，且為最後一個元素找到 O(n)
    - 最好情況—第一個元素就找到O(1)

```
          0  18
           /      \
     1 15           2   30
       /  \            /  \
3  40    4  50    5 100   6 40
```

Level 1

Level 2

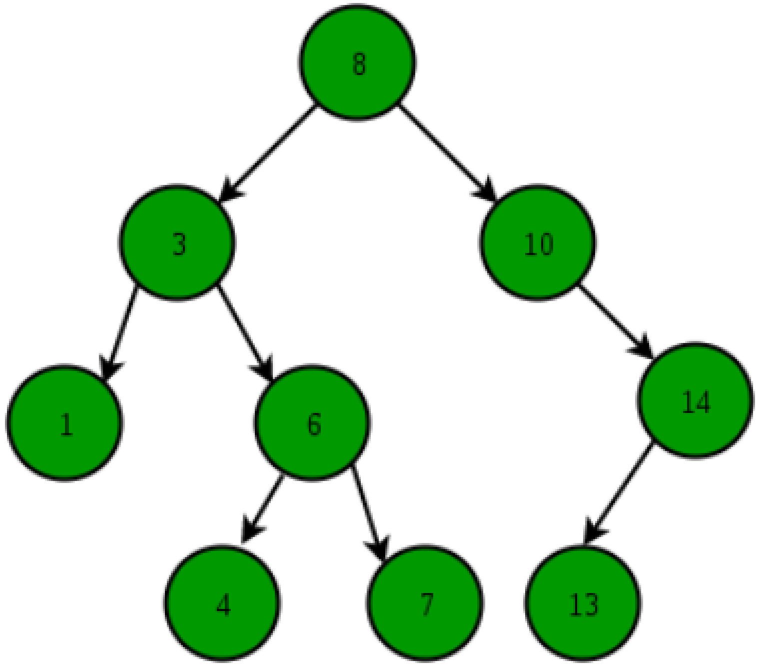
Level 3

二元樹(Binary tree) –11

- 時間複雜度(以二元樹為例)
  - ✓刪除某元素，元素為外部節點(葉節點)
    - 為搜尋時間 + O(1)，即刪除時間
  - ✓刪除某元素，元素為內部節點
    - 所以為搜尋時間+ 交換時間O(1) + O(1)
  - ✓新增某元素，為利用判斷左右子樹有沒有缺，因此時間為樹的所需高度
    - 為找尋左右是否哪邊有缺 $O(\lfloor log_2^{n+1} \rfloor)$ + 新增時間O(1)
  - ✓尋訪
    - 不用想，因為要印樹中所有節點，因此時間複雜度為O(n)

二元搜尋樹(Binary Search Tree, BST) –1

- 二元搜尋為二元樹加上以下限制
  - ✓限制為：左邊子節點的值，一定比父節點的值小；右邊子節點的值一定比父節點的值大
  - ✓不一定會是**滿、完整或是完美二元樹**，**要看資料一開始資料排列**的順序



二元搜尋樹(Binary Search Tree, BST) –2

- 由於二元搜尋樹定義了父節點與子節點的關聯，因此關聯從階層式邏輯轉換成了左小右大
- 一般來說，在探討二元搜尋樹時，需討論以下重點
  - ✓建樹
  - ✓新增節點
  - ✓刪除節點
  - ✓更新節點(變更值)
  - ✓搜尋節點
  - ✓尋訪節點

二元搜尋樹(Binary Search Tree, BST) –3

- 建樹
  - ✓根據規則，將輸入依序讀入後，假設第一個讀入的樹做為根，根據節點下的左子樹值會比該節點小，右子樹所有節點會比該節點大

main.cpp

```cpp
#include <iostream>

template <class CL1>
class Node{
    public: CL1 cl1_Field;
    public: Node<CL1>* op_Lt;
    public: Node<CL1>* op_Rt;
    public: Node(CL1 cl1_Field){
        this->cl1_Field = cl1_Field;
        op_Lt = nullptr;
        op_Rt = nullptr;
    }
};
```

二元搜尋樹(Binary Search Tree, BST) –4

- 建樹
  - ✓根據規則，將輸入依序讀入後，假設第一個讀入的樹做為根，根據節點下的左子樹值會比該節點小，右子樹所有節點會比該節點大(cont’d)

main.cpp

```
template<class CL1>
class Tree{
    public: Node<CL1>* op_Root;

    public: Tree(CL1 cl1_Data){
        op_Root = nullptr;

        op_Root = new Node<CL1>(cl1_Data);
    }

    public: ~Tree(){
        fn_TraInOrder(op_Root, true);
    }
```

二元搜尋樹(Binary Search Tree, BST) –5

- 建樹
  - ✓根據規則，將輸入依序讀入後，假設第一個讀入的樹做為根，根據節點下的左子樹值會比該節點小，右子樹所有節點會比該節點大(cont'd)

main.cpp

```cpp
    public: void InsNode(CL1 cl1_Data){
        Node<CL1>* op_NewN = new Node<CL1>(cl1_Data);

        for(Node<CL1>* op_Tmp= op_Root;;){
            if(op_Tmp->cl1_Field > cl1_Data){
                if(op_Tmp->op_Lt == nullptr){
                    op_Tmp->op_Lt = op_NewN;
                    break;
                }
                else{
                    op_Tmp = op_Tmp->op_Lt;
                }
            }
            else{
                if(op_Tmp->op_Rt == nullptr){
                    op_Tmp->op_Rt = op_NewN;
                    break;
                }
                else{
                    op_Tmp = op_Tmp->op_Rt;
                }
            }
        }
    }
```

二元搜尋樹(Binary Search Tree, BST) –6

- 建樹
  - ✓根據規則，將輸入依序讀入後，假設第一個讀入的樹做為根，根據節點下的左子樹值會比該節點小，右子樹所有節點會比該節點大(cont’d)

main.cpp

```cpp
    public: void fn_TraInOrder (Node<CL1>* op_Node, bool b_DMode =false){
        if (op_Node == nullptr){
            return;
        }
        fn_TraInOrder(op_Node->op_Lt, b_DMode);
        Node<CL1>* op_TmpR = op_Node->op_Rt;
        if(b_DMode == false){
            std::cout<<op_Node->cl1_Field<<" ";
        }
        else{
            delete op_Node;
        }
        fn_TraInOrder(op_TmpR, b_DMode);
    }
};
```

二元搜尋樹(Binary Search Tree, BST) –7

- 建樹
  - ✓根據規則，將輸入依序讀入後，假設第一個讀入的樹做為根，根據節點下的左子樹值會比該節點小，右子樹所有節點會比該節點大(cont’d)

main.cpp

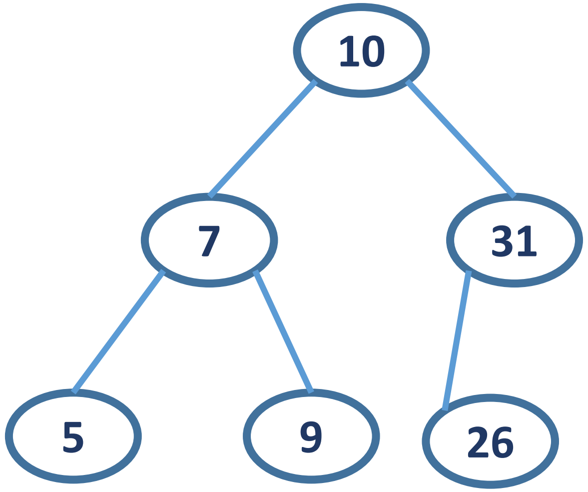
```cpp
int main(){
    Tree<int> o_Tree(10);
    o_Tree.InsNode(5);
    o_Tree.InsNode(1);
    o_Tree.InsNode(7);
    o_Tree.InsNode(2);
    o_Tree.InsNode(40);
    o_Tree.InsNode(50);
    std::cout<<"Inorder traversal of the constructed tree: \n";
    o_Tree.fn_TraInOrder(o_Tree.op_Root);

    return 0;
}
```

二元搜尋樹(Binary Search Tree, BST) –8

- 新增節點
  - ✓因為新增時利用fn_InsNode(.)進行即可交新增節點插入至樹中合適的位址，因此使用上述fn_InsNode(.)即可
- 刪除節點
  - ✓由於刪除節點後，依舊要滿足原二元搜尋樹條件，因此需考慮下列狀況
    - Case 1: 刪除的節點下面沒有任何的子節點
    - Case 2: 刪除的節點下面有一個子節點(不管事左或是右)
    - Case 3: 刪除的節點下面有兩個子節點

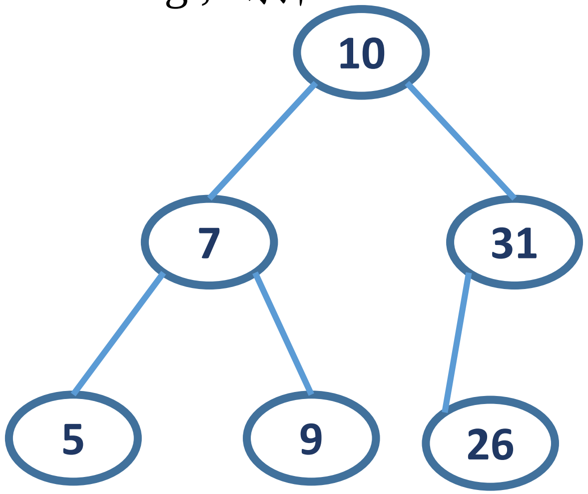
二元搜尋樹(Binary Search Tree, BST) –9

- 刪除節點
  - ✓由於刪除節點後，依舊要滿足原二元搜尋樹條件，因此需考慮下列狀況 (cont’d)
    - Case 1: 刪除的節點下面沒有任何的子節點
      - 直接刪就好了，別想太多
        - E.g., 刪掉5



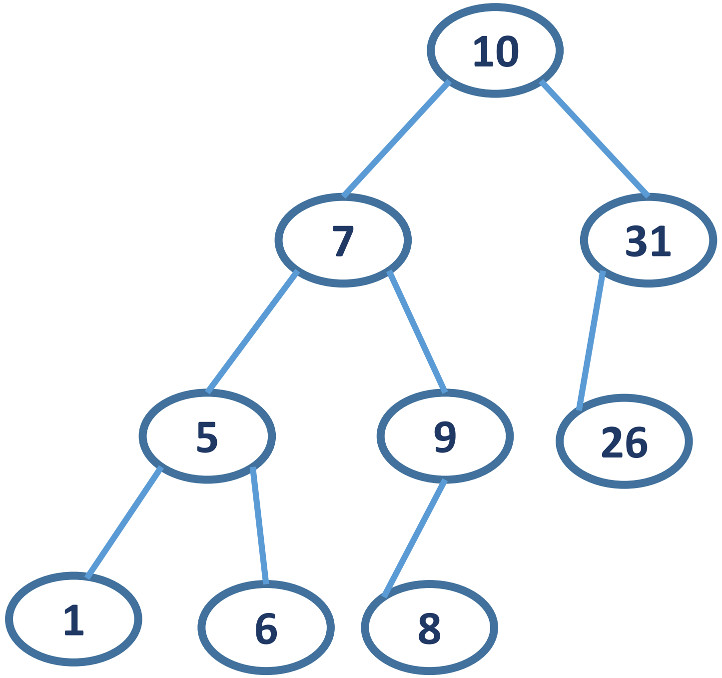
二元搜尋樹(Binary Search Tree, BST) –10

- 刪除節點
  - ✓由於刪除節點後，依舊要滿足原二元搜尋樹條件，因此需考慮下列狀況(cont’d)
    - Case 2: 刪除的節點下面有一個子節點(不管事左或是右)
      - 由於刪除節點下方節點一定會滿足BST條件，所以只要將刪除的節點的子節點與欲刪除的父節點進行連結即可
        - E.g., 刪掉31



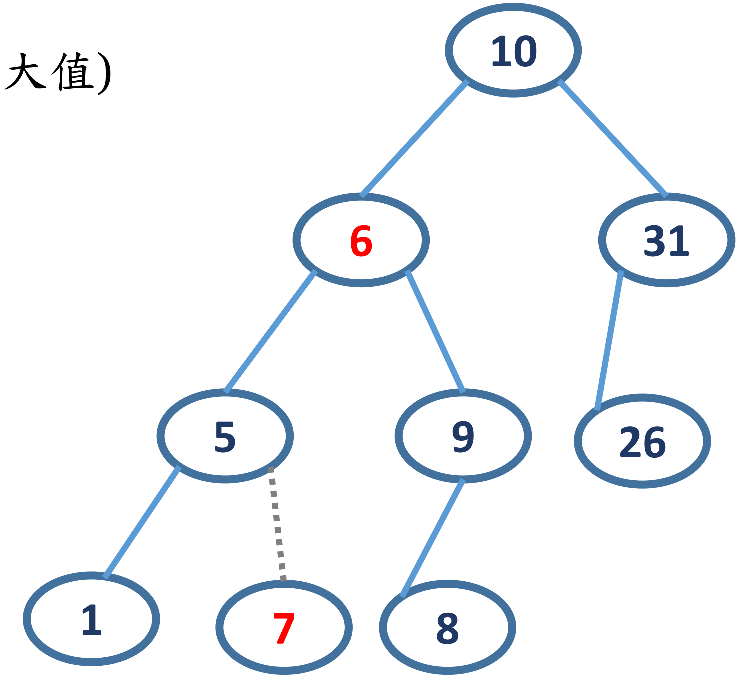
二元搜尋樹(Binary Search Tree, BST) –11

- 刪除節點
  - ✓由於刪除節點後，依舊要滿足原二元搜尋樹條件，因此需考慮下列狀況 (cont'd)
    - Case 3: 刪除的節點下面有兩個子節點
      - 把刪除節點中「左子樹的最大值節點」或是「右子樹的最小值節點」則依進行值的交換，再刪除交換後的值即可
        - E.g., 刪掉7



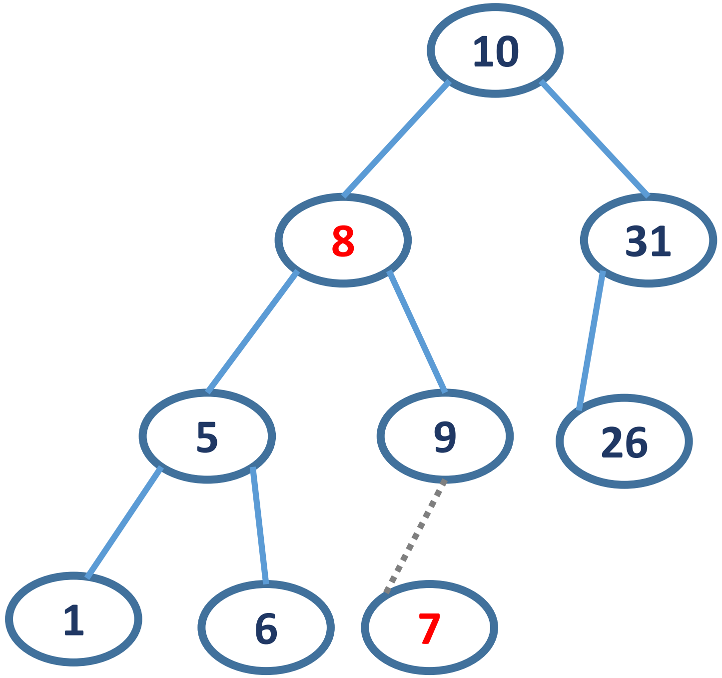
二元搜尋樹(Binary Search Tree, BST) –12

- 刪除節點
  - ✓由於刪除節點後，依舊要滿足原二元搜尋樹條件，因此需考慮下列狀況(cont'd)
    - Case 3: 刪除的節點下面有兩個子節點
      - 把刪除節點中「左子樹的最大值節點」或是「右子樹的最小值節點」則依進行值的交換，再刪除交換後的值即可
        - E.g., 刪掉7(左子樹最大值)



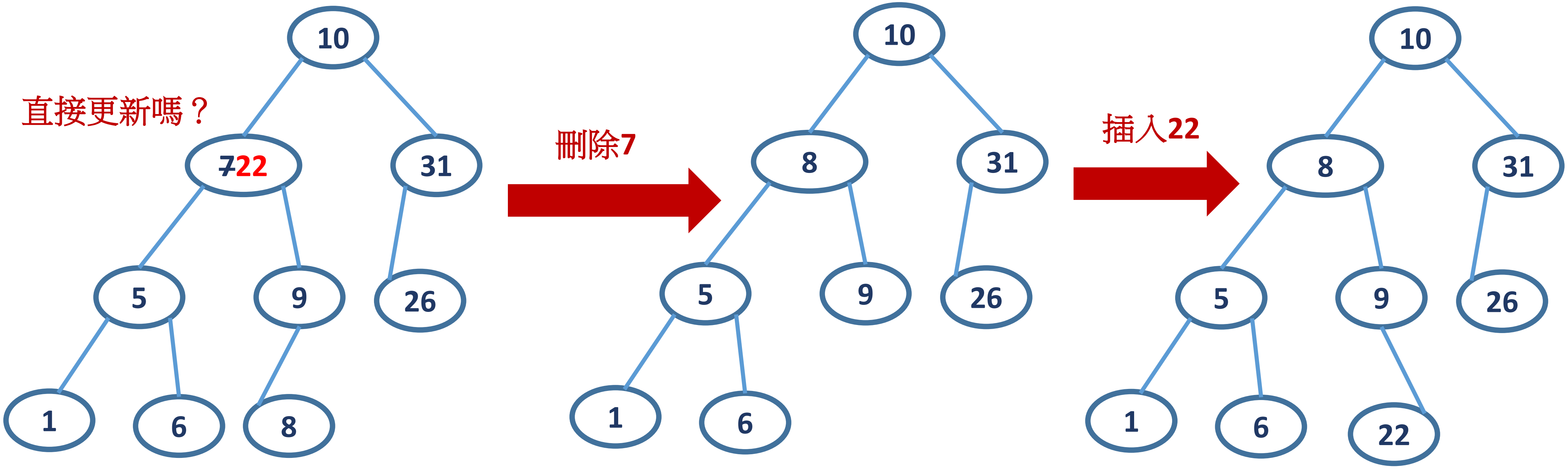
二元搜尋樹(Binary Search Tree, BST) –13

- 刪除節點
  - ✓由於刪除節點後，依舊要滿足原二元搜尋樹條件，因此需考慮下列狀況 (cont’d)
    - Case 3: 刪除的節點下面有兩個子節點
      - 把刪除節點中「左子樹的最大值節點」或是「右子樹的最小值節點」則依進行值的交換，再刪除交換後的值即可
        - E.g., 刪掉7



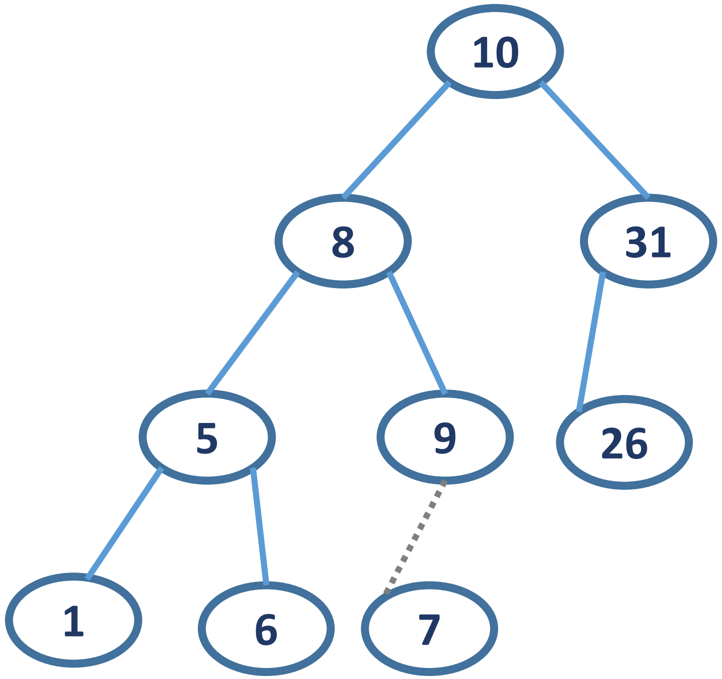
二元搜尋樹(Binary Search Tree, BST) –14

- 更新節點
  - ✓刪除更新節點，再直接重新插入新的值



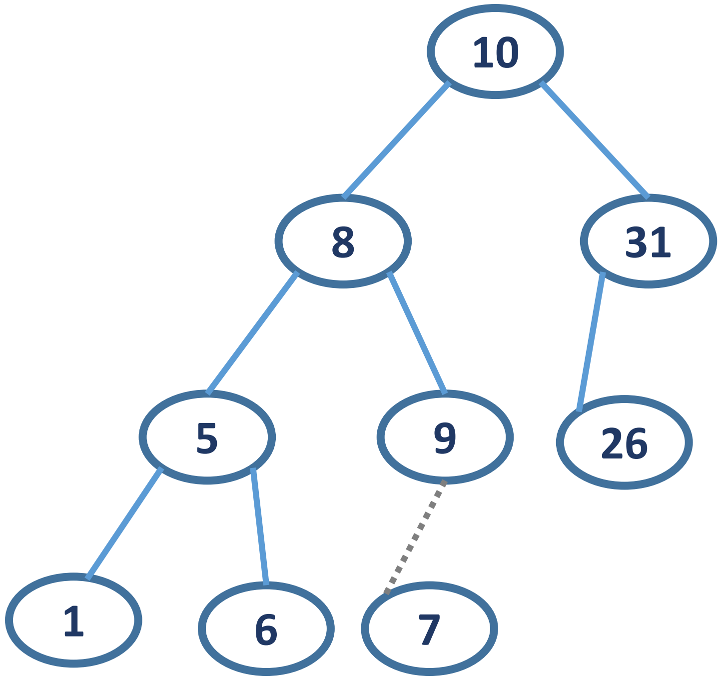
二元搜尋樹(Binary Search Tree, BST) –15

- 搜尋節點
  - ✓直接利用二元樹特性進行搜尋



二元搜尋樹(Binary Search Tree, BST) –15

- 尋訪節點
  - ✓跟二元樹一樣，有分成中序(in-order)、前序(pre-order)跟後序(post-order)

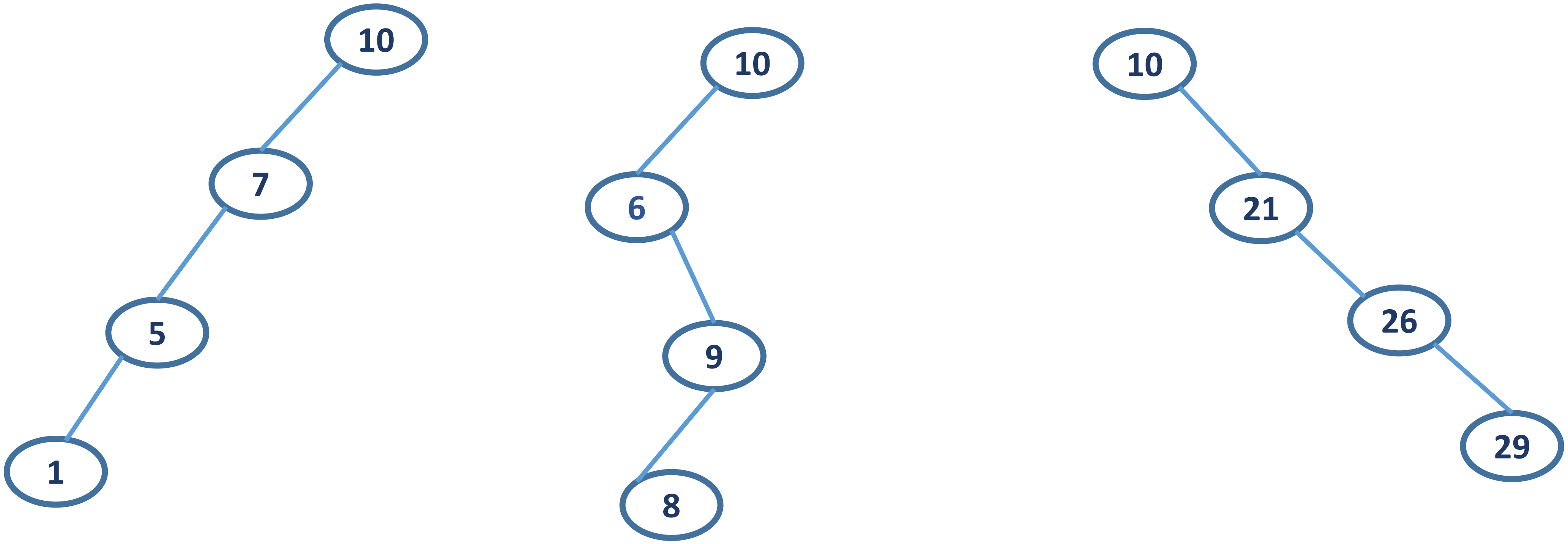


AVL樹(1)

- 為二元搜尋樹的進階版
- AVL代表兩個人，這棵樹是以人命名
  - ✓Adelson-Velsky and Landis Tree
- 又稱為**平衡二元搜尋樹**
- 為何會有AVL樹？動機在於二元搜尋樹本身存在問題
  - ✓二元搜尋樹本身有父節點最多只有兩個子節點，且左子樹中的節點會比父節點小；右子樹中的節點會比父節點大
  - ✓當輸入的input資料順序讓二元搜尋樹產生歪斜樹時(skewed tree)時

AVL樹(2)

- E.g., Skewed trees



AVL樹(3)

- 何會有AVL樹？動機在於二元搜尋樹本身存在問題(cont'd)
  - ✓當二元搜尋樹為歪斜樹時，樹高=資料筆數
  - ✓這意味著在執行上述二元搜尋樹時，全部資料皆為線性，因此
    - 沒有任何節省時間的部分
      - 搜尋、刪除、變更值等操作
  - ✓**因此最佳情況是，樹本身要「平衡」，才能發揮二元搜尋樹中的操作擁有最佳性能**

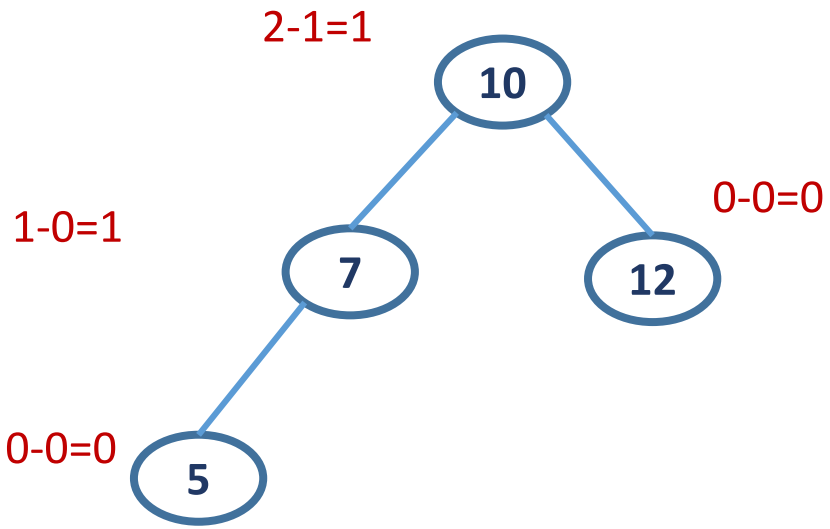
AVL樹(4)

- 平衡方法
  - ✓旋轉(rotation)。~~旋轉、跳躍、我不停歇。~~
    - 共四種，為LL、RR、LR與RL
    - 上述四種旋轉方法主要倚靠平衡因子(**Balance Factor**)決定是否要旋轉
- 平衡因子
  - ✓定義：
    - ||欲求節點之左子樹至最深的葉節點高度 -欲求節點之右子樹至最深的葉節點高度||
      - 若上述值=0或1，代表平衡
      - 若大於等於2則代表非平衡
        - 請旋轉
  - ✓用後置尋訪方法進行調整

AVL樹(5)

✓平衡樹

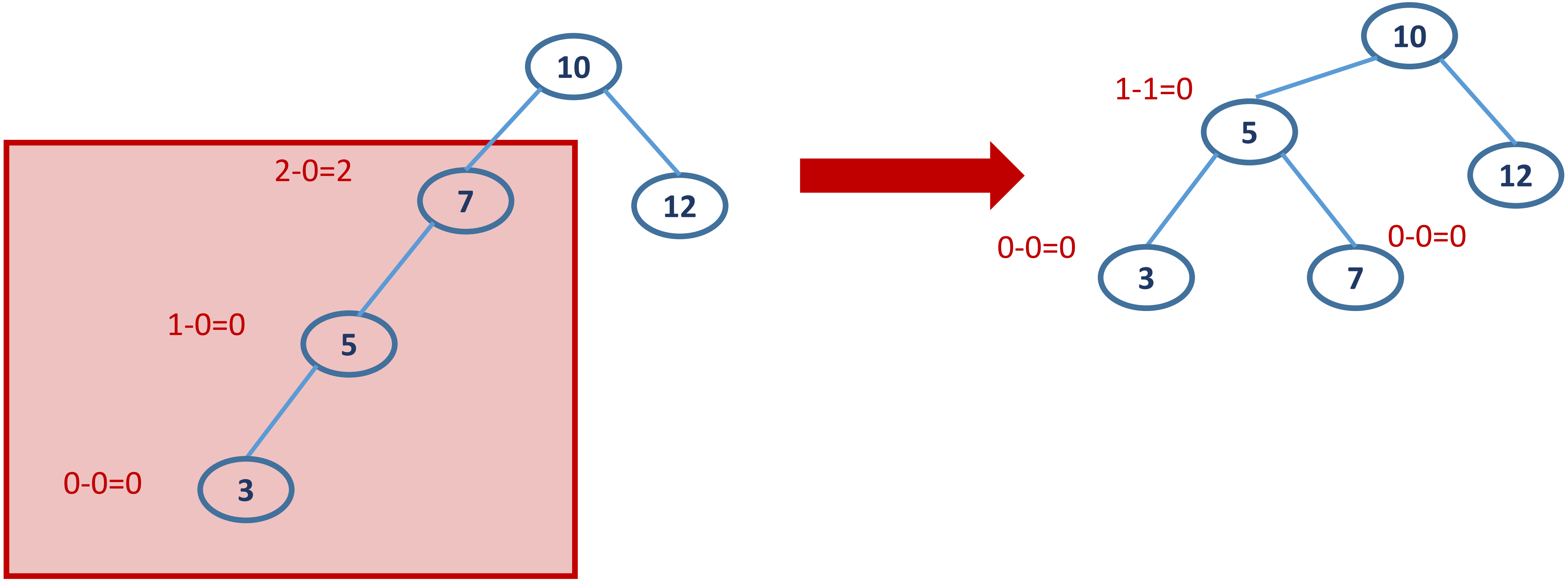
- E.g.,



AVL樹(5)

✓左傾樹(LL)

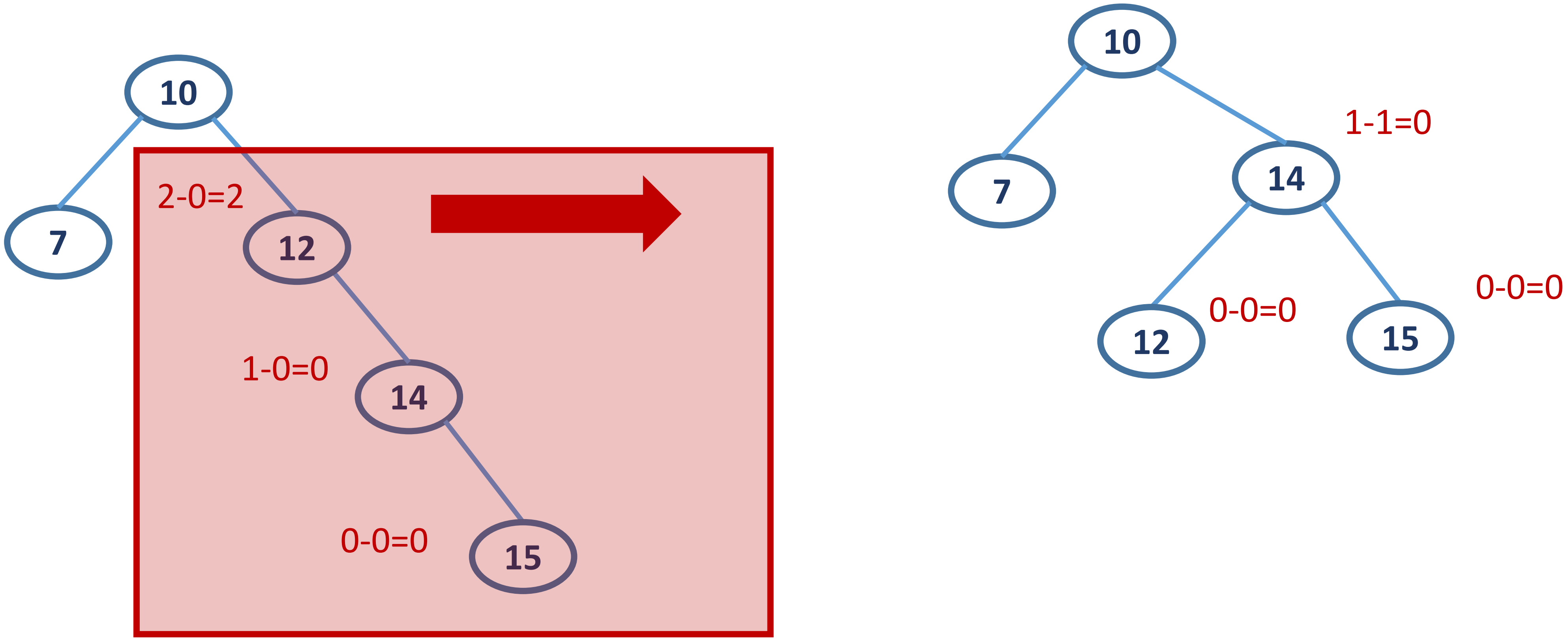
- E.g.,



AVL樹(6)

✓右傾樹(LR)

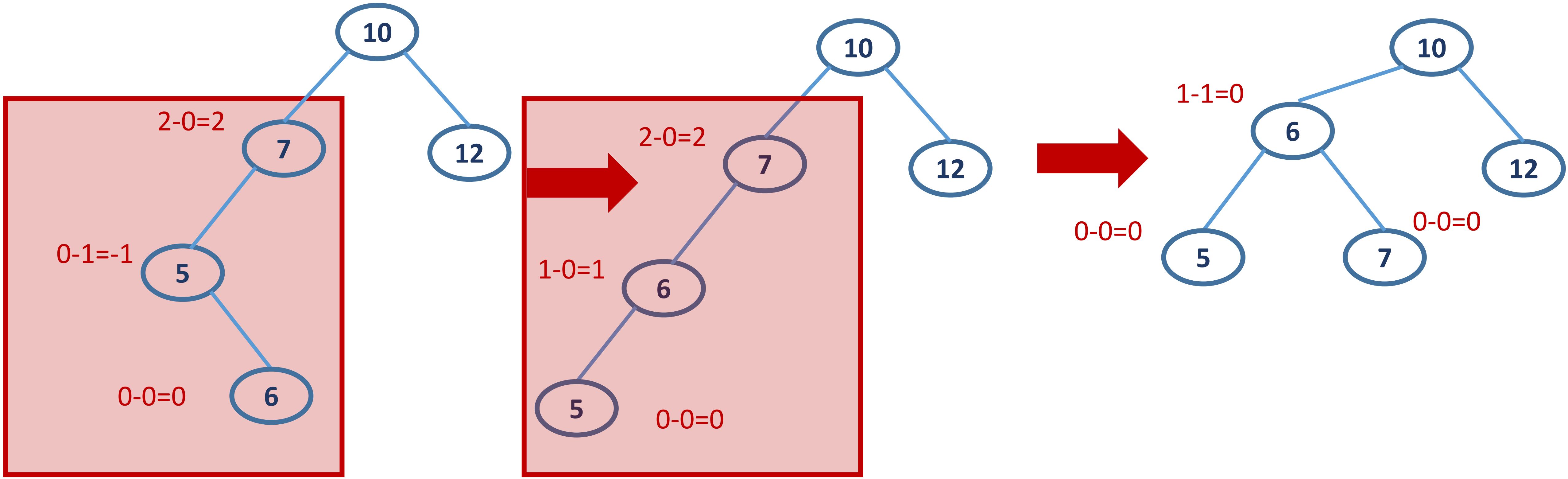
- E.g.,



AVL樹(7)

✓左-右傾樹(LR)

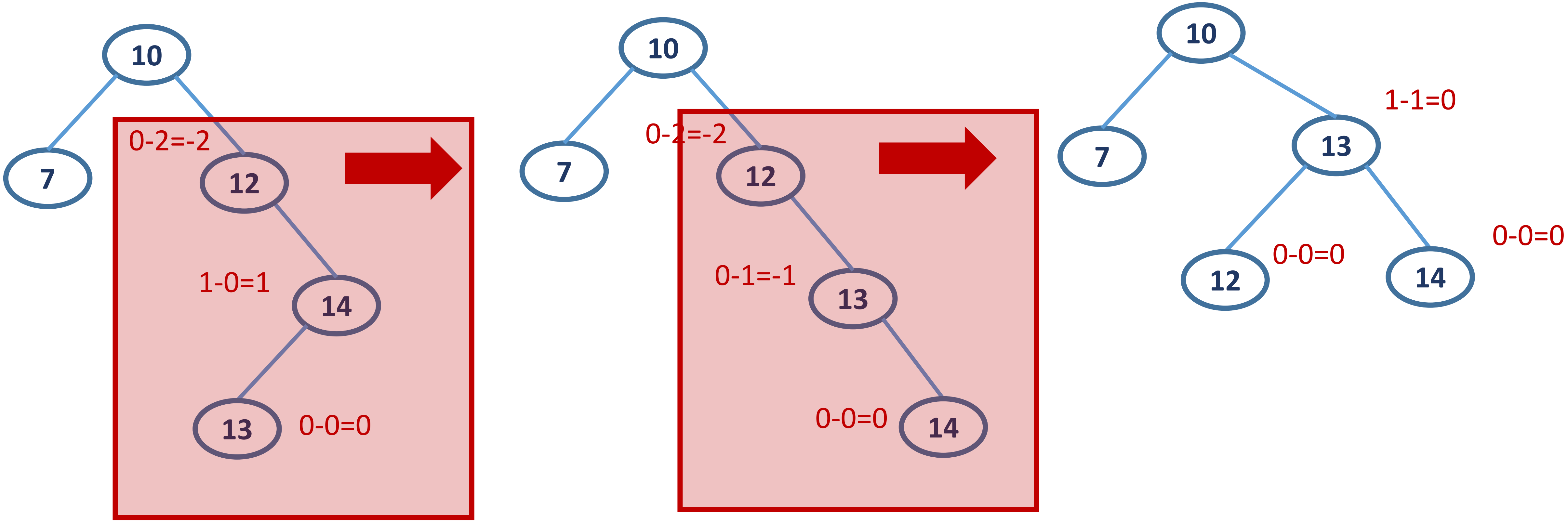
- E.g.,



AVL樹(8)

✓右-左傾樹(RL)

- E.g.,



補充

- 根據圖與樹的關聯