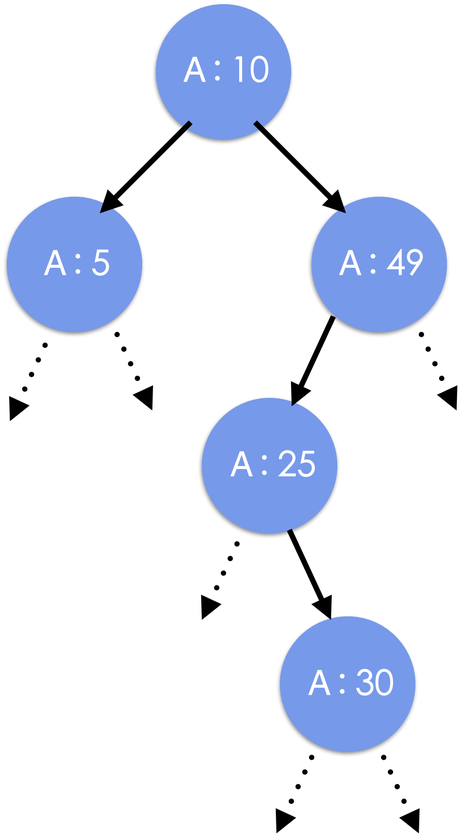
# Binary Search Tree

# 大綱

1. 二元搜尋樹的定義
2. 情境比較
3. 儲存方式
4. Insert
5. 平衡 v.s 歪斜
6. 解法：AVL樹 , 紅黑樹

# 定義

1. 是一種二元樹：
   因此每個節點只會有左節點右節點
2. 左節點的值**小於等於**父節點的值
3. 右節點的值**大於**父節點的值

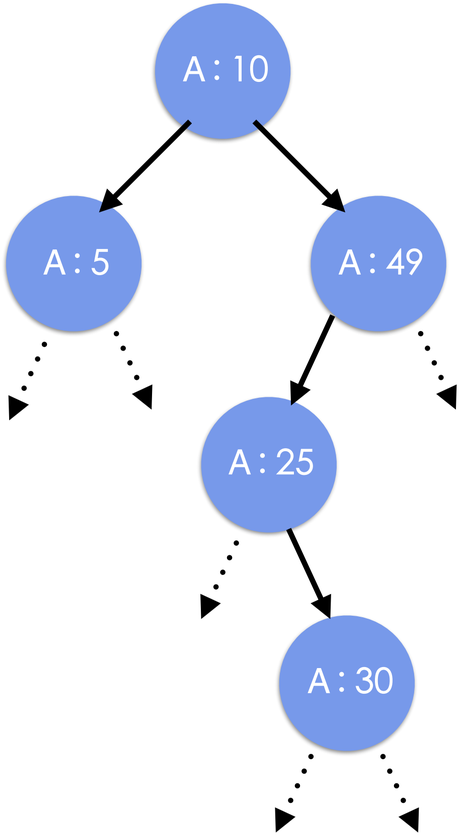


# 問題意識

嘗試找出小於15的數字。在這個Case之下二元搜尋樹效率較佳。


逐項比較
耗費五次比較時間

V.S

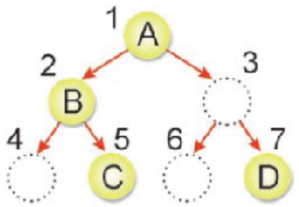
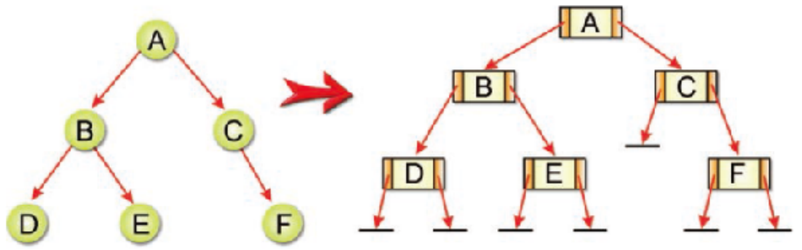



根節點比較完之後，
直接取用左子樹，
僅花一次比較時間

然而需要使用較多
指標位置

# 儲存方式

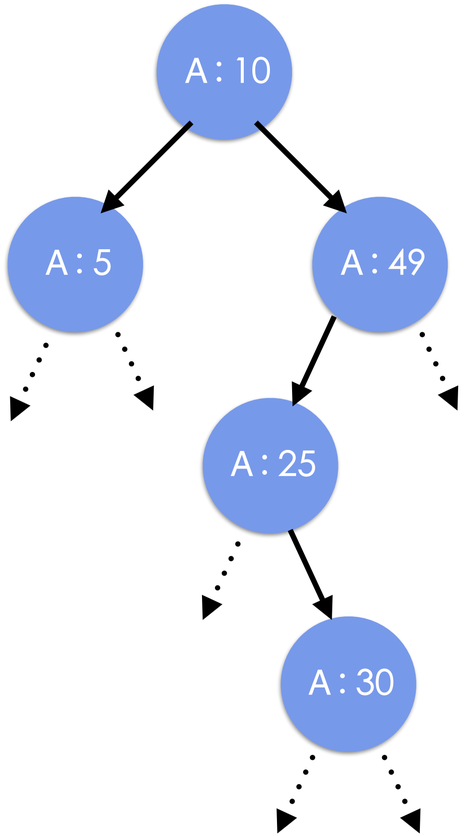
建立二元搜尋樹(BST)有兩種儲存方式，一個利於更新一個利於回溯

| | 優點 | 缺點 | 示意圖 |
|---|---|---|---|
| **Array** | 容易找到父節點 | 歪斜樹會浪費空間 |  <br>索引值 1 2 3 4 5 6 7<br>內容值 A B C D |
| **Linked List** | 容易新增修改 | 不易找到父節點 |  |

# Insert

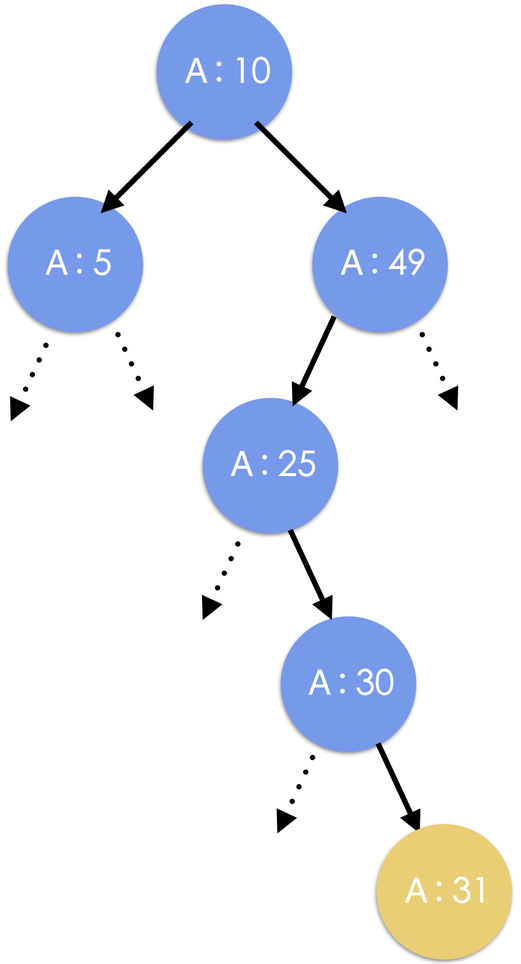
Insert基本上就是將新輸入的節點依照二元搜尋樹的規則加入樹中。

新Node

# Insert

Insert基本上就是將新輸入的節點依照二元搜尋樹的規則加入樹中。



# 歪斜 V.S 平衡

若建樹的過程中， List有被排序過會
出現歪斜的現象使得搜索效率變差

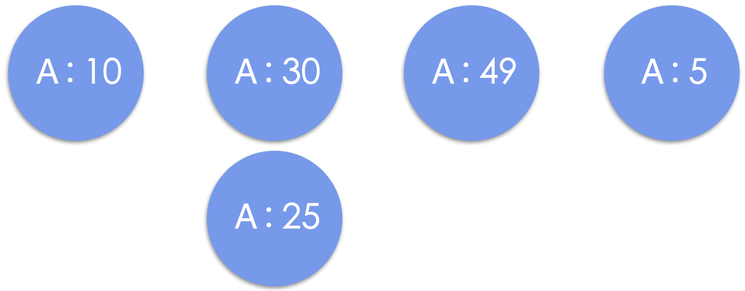
平衡


歪斜

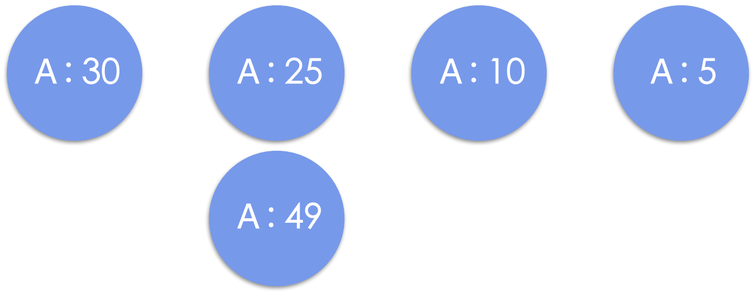
# 歪斜 V.S 平衡

若建樹的過程中， List有被排序過會
出現歪斜的現象使得搜索效率變差

平衡



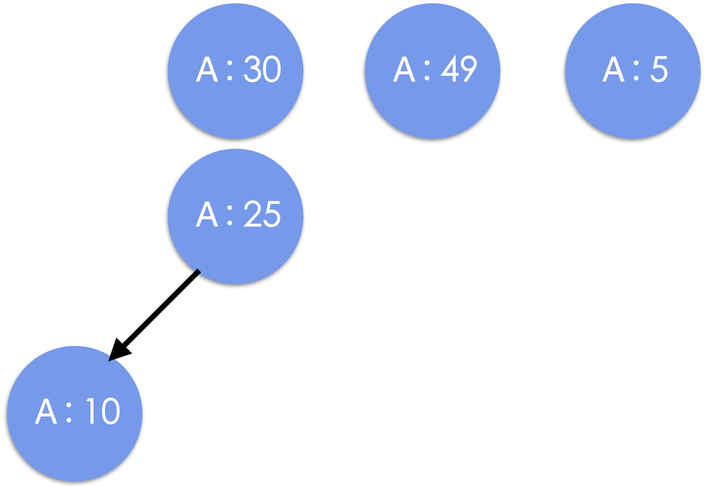

歪斜



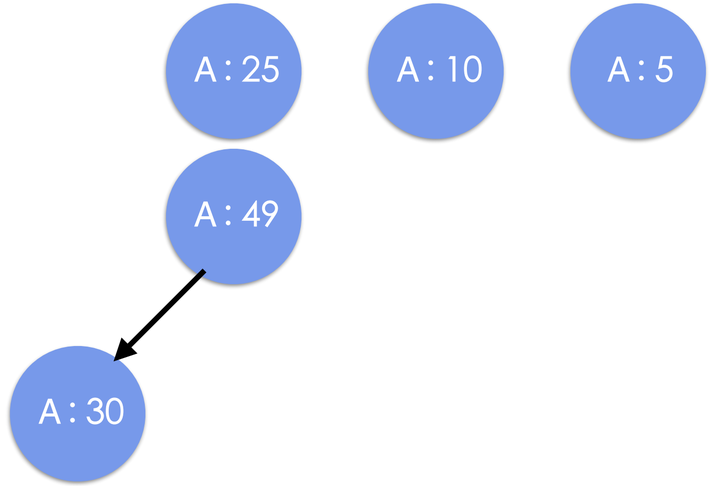
# 歪斜 V.S 平衡

若建樹的過程中， List有被排序過會
出現歪斜的現象使得搜索效率變差

平衡



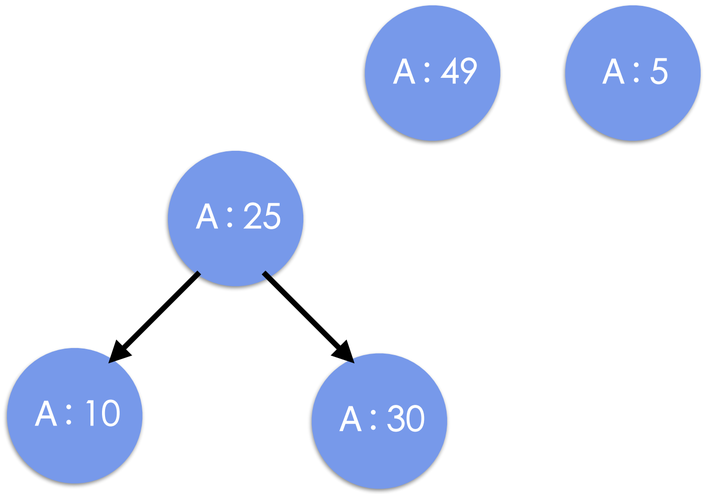

歪斜



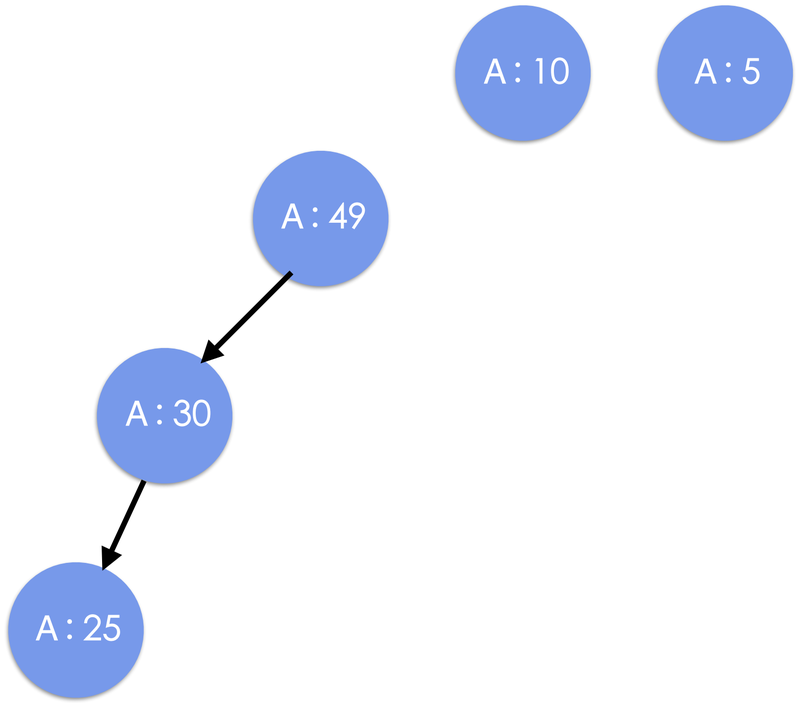
# 歪斜 V.S 平衡

若建樹的過程中， List有被排序過會
出現歪斜的現象使得搜索效率變差

平衡



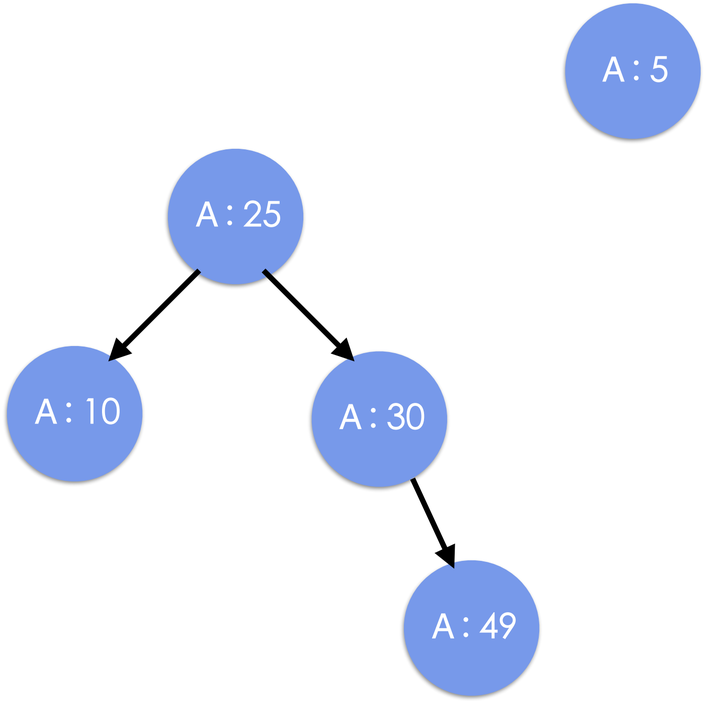

歪斜



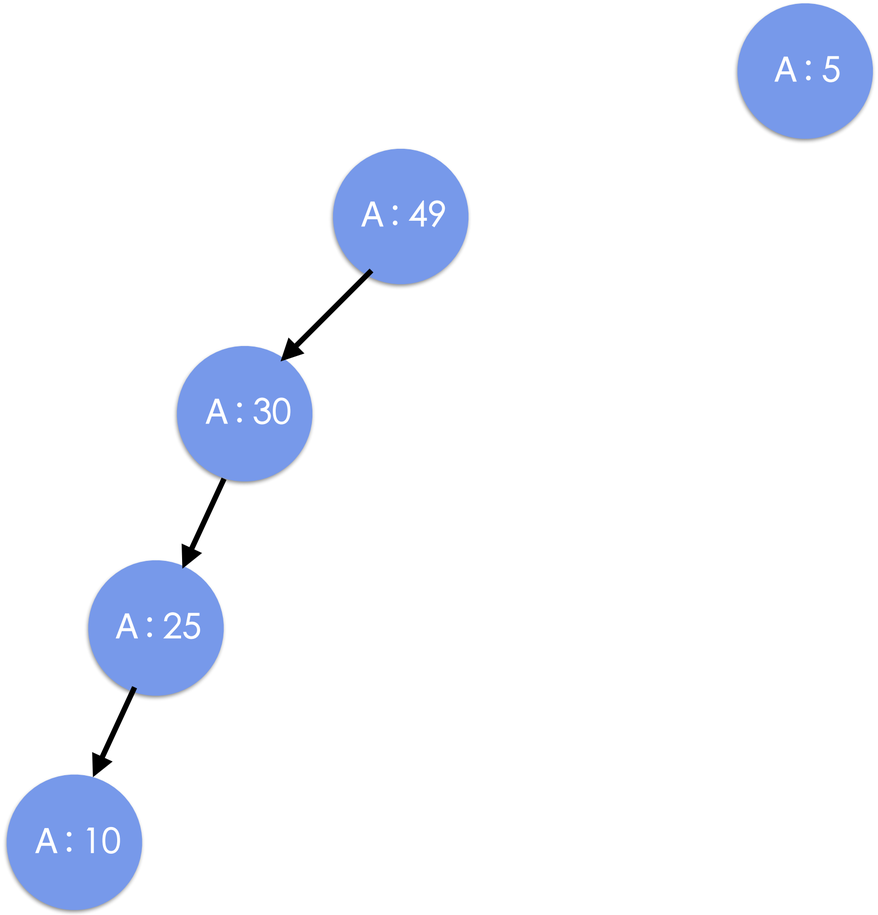
# 歪斜 V.S 平衡

若建樹的過程中， List有被排序過會
出現歪斜的現象使得搜索效率變差

平衡



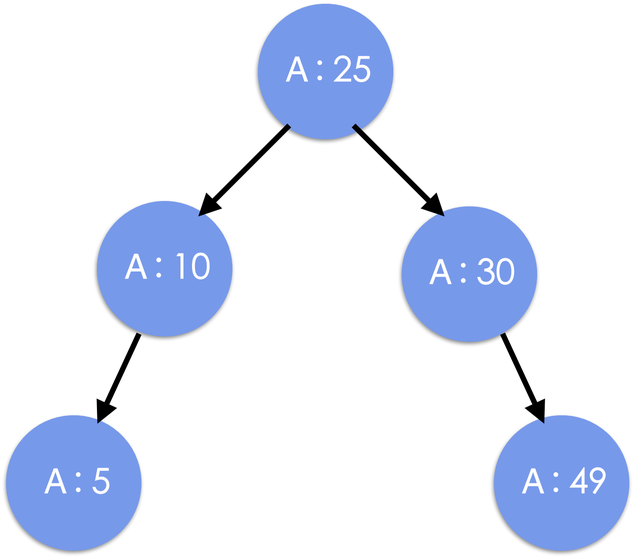

歪斜



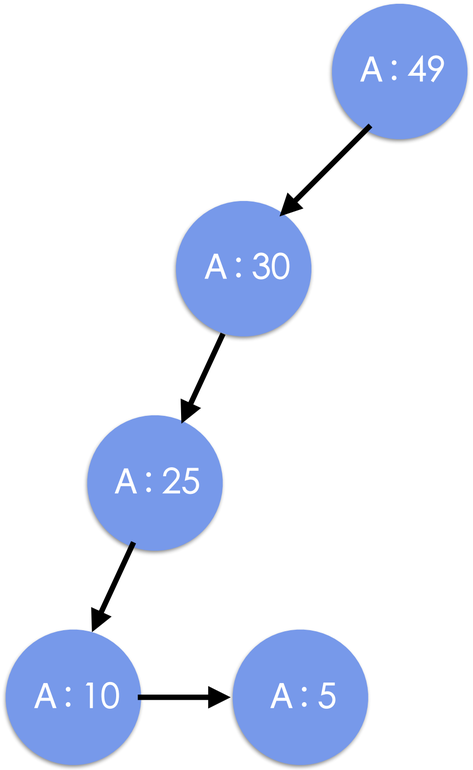
# 歪斜 V.S 平衡

若建樹的過程中， List有被排序過會
出現歪斜的現象使得搜索效率變差

平衡



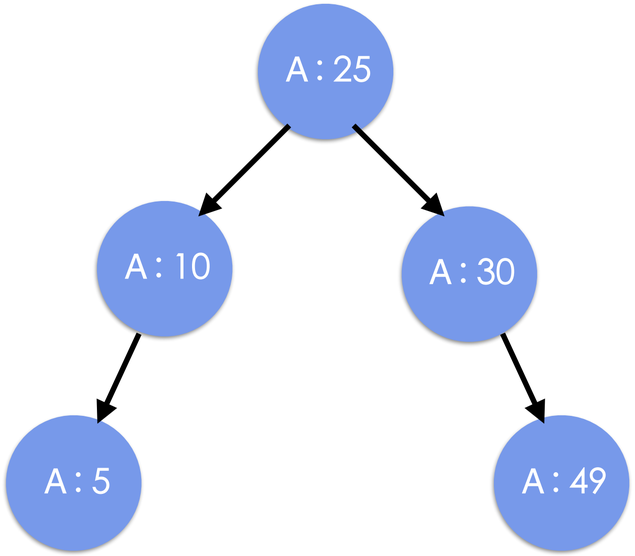

歪斜



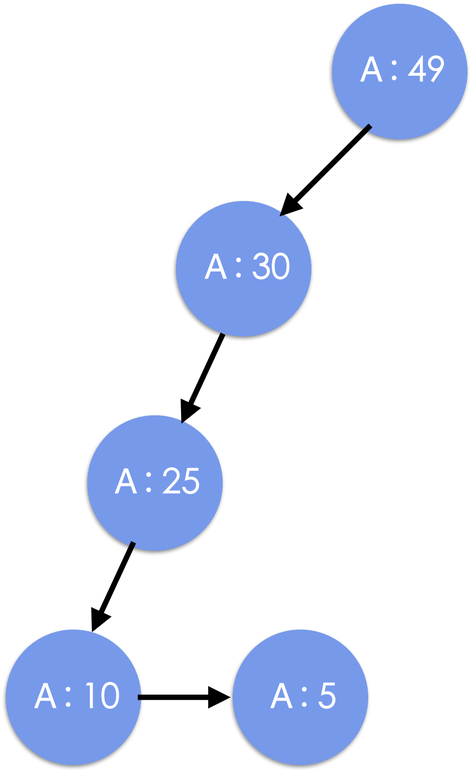
# 歪斜 v.s 平衡

試找小於5的數字

平衡




歪斜



二元搜尋樹如果建的歪斜會導致搜索效率變差。
是否有方法可以讓樹建立的比較平衡？

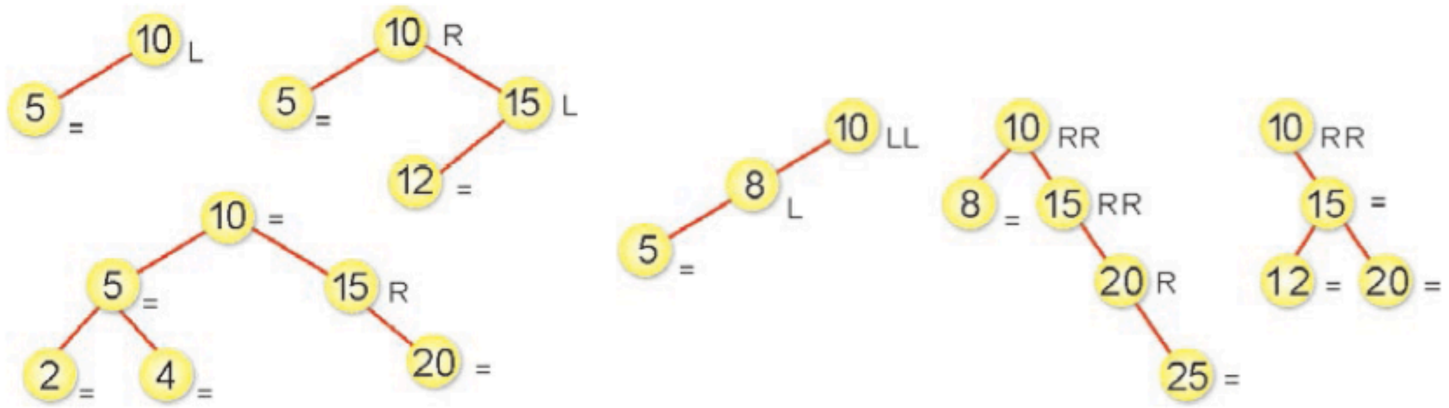
# 解法1 . AVL

AVL樹( Adelson Velsky ,Evgenii Landis)是一種平衡樹，樹上的每個左右子樹高差都不超過1

T 是一個非空的二元樹，$T_l$ 及 $T_r$ 分別是它的左右子樹，若符合下列兩條件，則稱 T 是個高度平衡樹：

1. $T_l$ 及 $T_r$ 也是高度平衡樹。
2. $|h_l-h_r| \leq 1$，$h_l$ 及 $h_r$ 分別為 $T_l$ 與 $T_r$ 的高度。

如下圖所示：



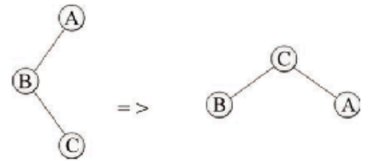

(a) AVL 樹　　(b) 非 AVL 樹

# 解法1 . AVL

每次加入新的Node時，從葉節點往上計算每個Node的平衡因子，若平衡因子超過2做Roatation

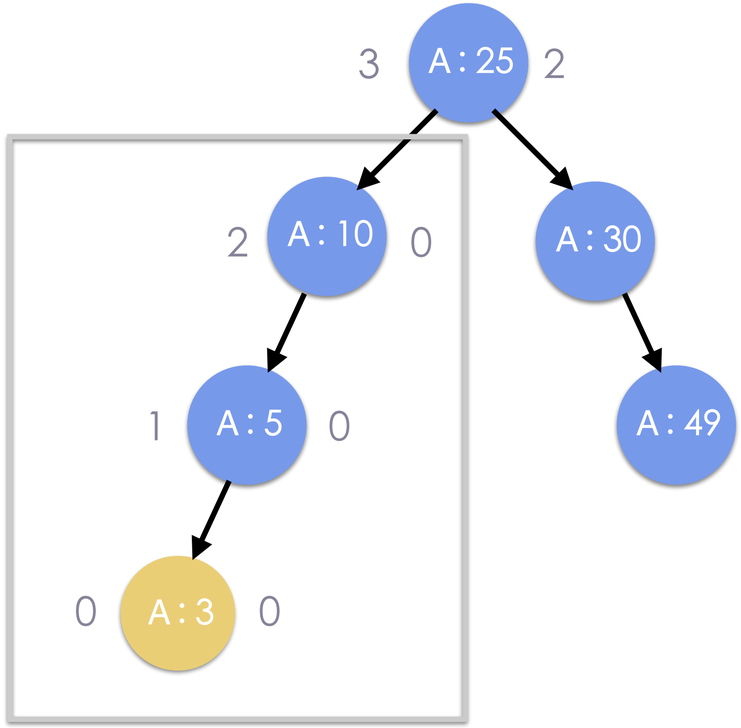
- LL 型


- LR 型

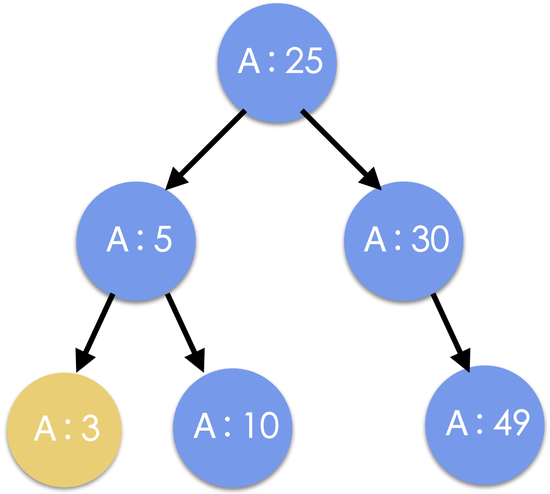



- RR 型


- RL 型


Do LL Rotation

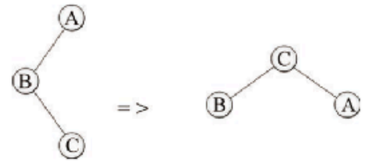


# 解法1 . AVL

每次加入新的Node時，從葉節點往上計算每個Node的平衡因子，若平衡因子超過2做Roatation

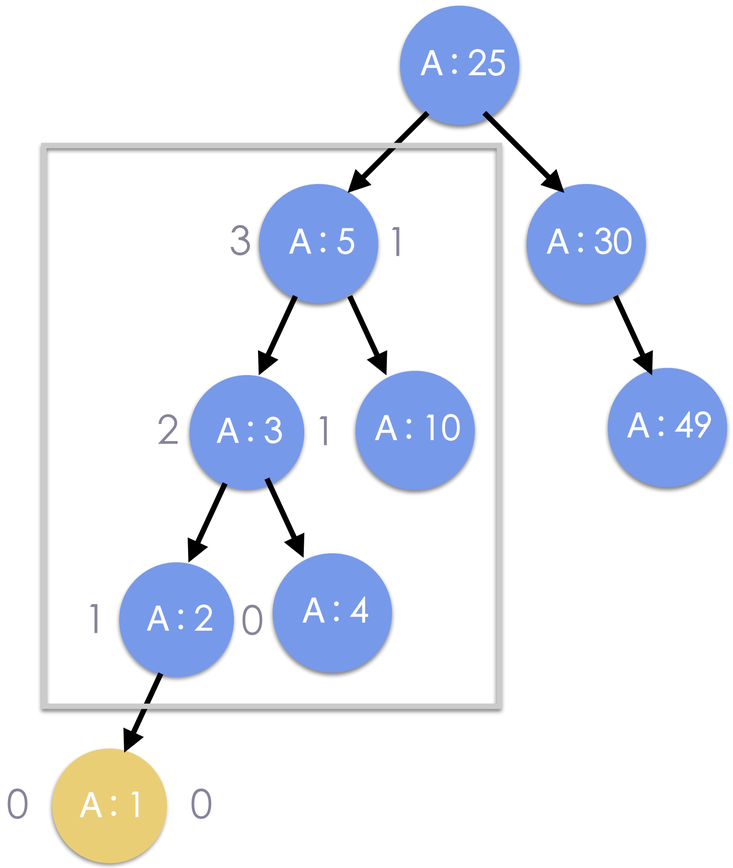
- LL 型


- LR 型

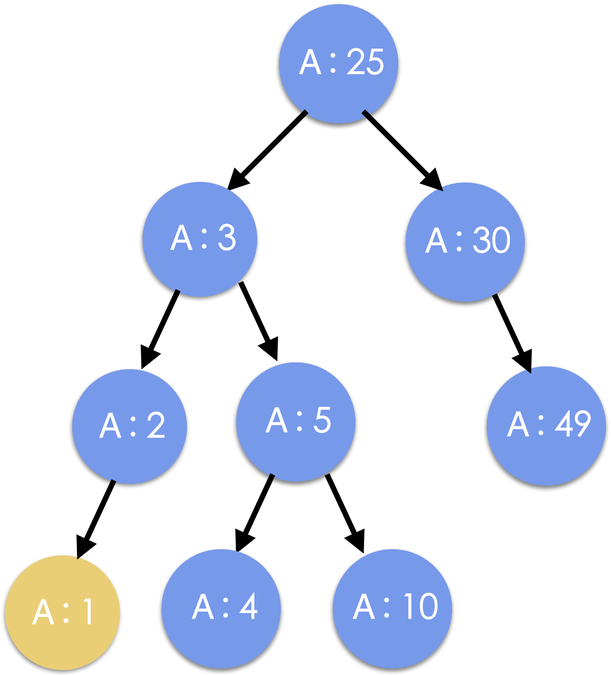



- RR 型


- RL 型


Do LL Rotation



# 解法2 .紅黑樹

將每個節點設定成黑色跟紅色，並且在符合演算的條件下可以建立出較平衡的樹

## Rule of Red Black Tree

Rule 1 . 每個Node都是紅色或黑色

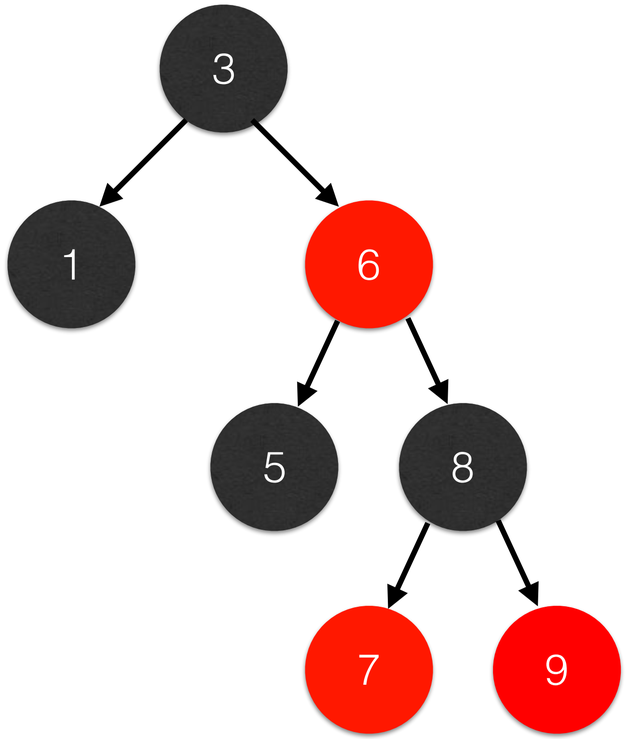
Rule 2 . 根節點永遠是黑色

Rule 3 . 新增的Node一定是紅色

Rule 4 . 每條路徑不能有連續的紅色Node

Rule 5 . 從根節點開始的每條路徑有一樣數量的黑色Node

Rule 6 . Null節點是黑色




## If Violate DO:

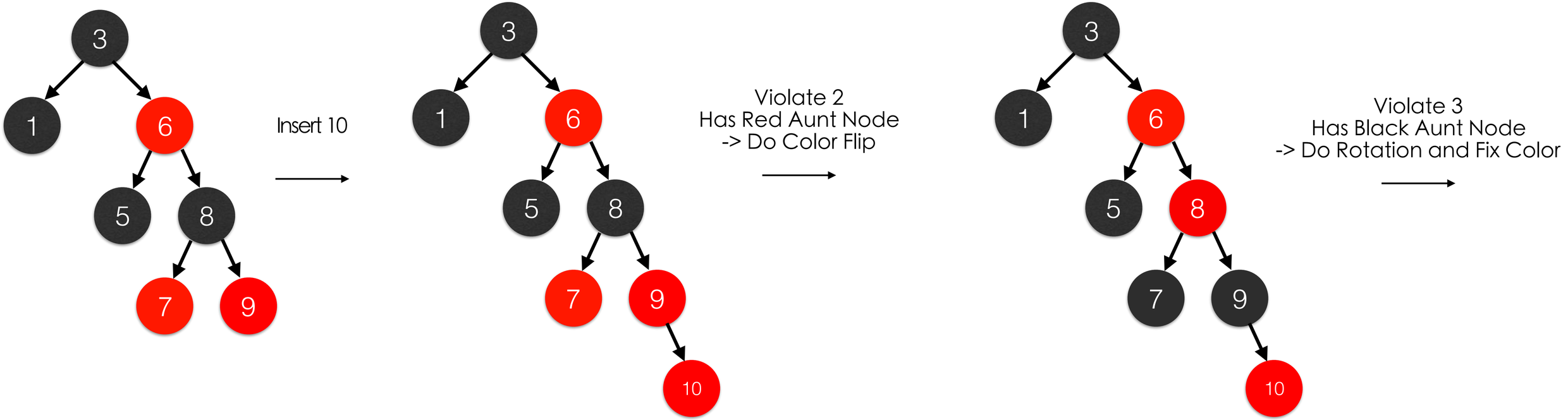
1 . Black Aunt Node:

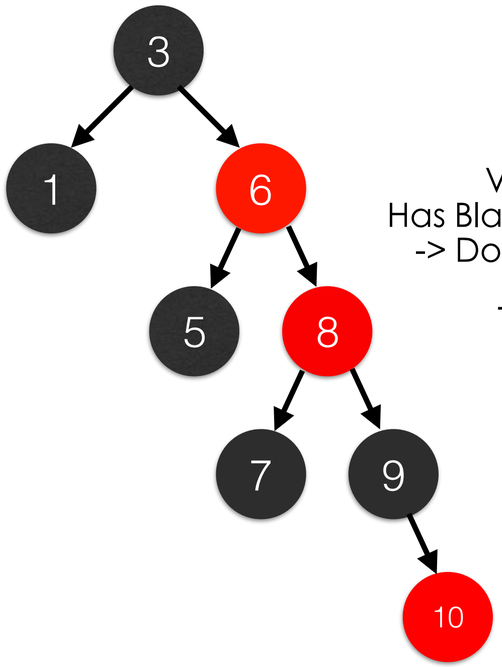
DO Rotate and Fix Color(Parent be black children be red)

2 . Red Aunt Node:

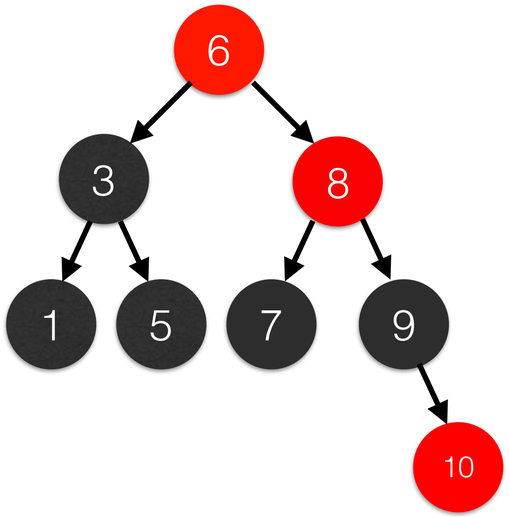
DO Color Flip

# 解法2 .紅黑樹

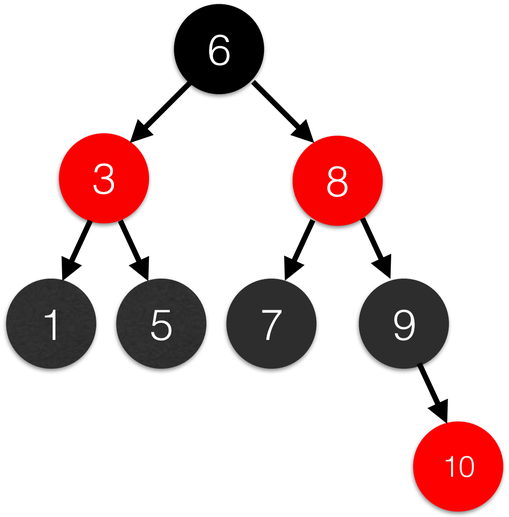

3
1
6
5
8
7
9
Insert 10
3
1
6
5
8
7
9
10
Violate 2
Has Red Aunt Node
-> Do Color Flip
3
1
6
5
8
7
9
10
Violate 3
Has Black Aunt Node
-> Do Rotation and Fix Color




Violate 3
Has Black Aunt Node
-> Do RR Rotation




Color Fixed



# Summary

1. 串列**不利搜尋**
2. 因此可以建二元搜尋樹**加速搜索**
3. 輸入順序會影響到樹的**歪斜**進而影響到搜索效率
4. 建樹的過程使用 **紅黑樹、AVL樹算法可以建出較平衡的樹**