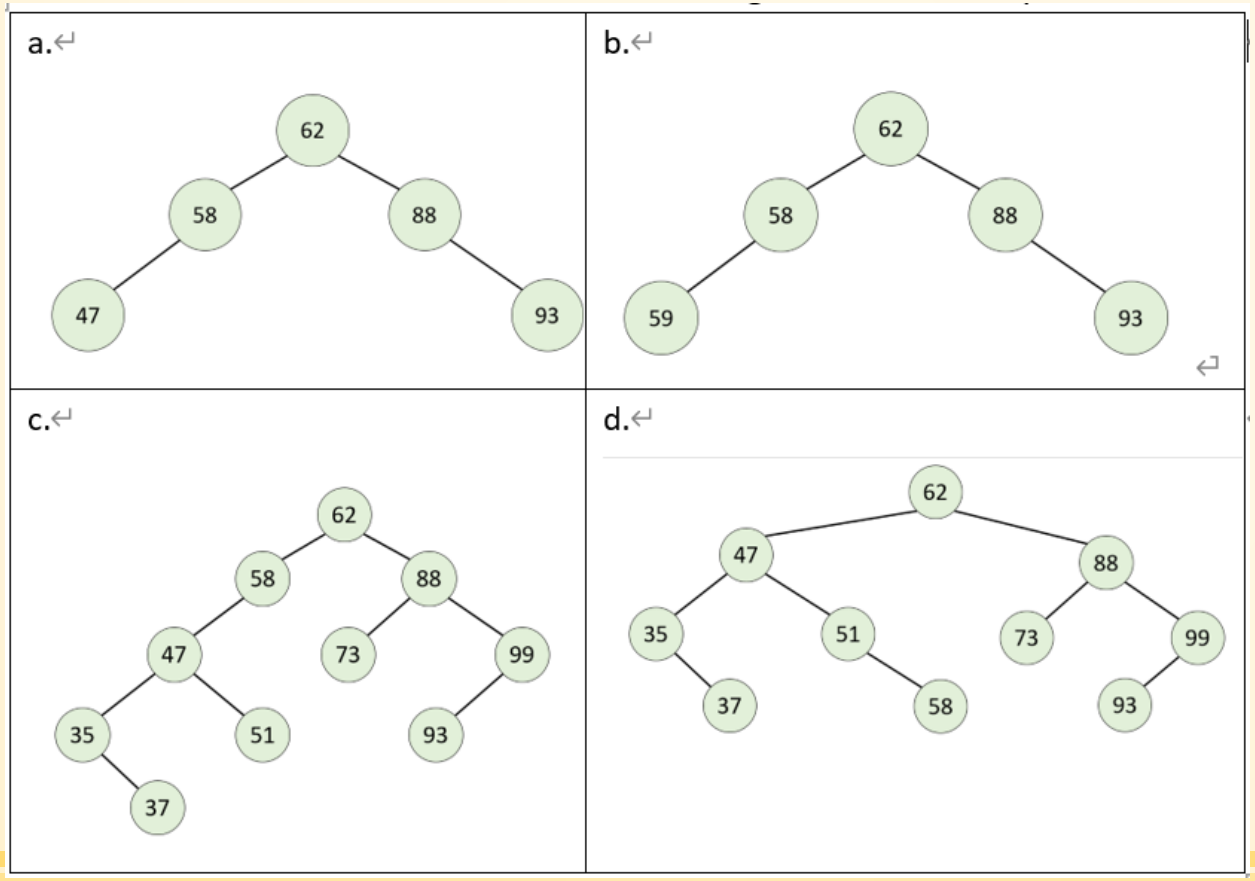
# Advanced Topics 習題

# Q1

- Please determine whether these trees are Height-Balanced Binary Search Tree.



# Q1_Ans

a. Yes, the balance factor of each node is no more than 1, and it is BST.

b. No, it is not BST, since Node 59 should not be the left node of Node 58.

c. No, the balance factor of Node 58 is 3, which is more than 1, so it is not height-balanced.

d. Yes, the balance factor of each node is no more than 1, and it is BST.

# Q2

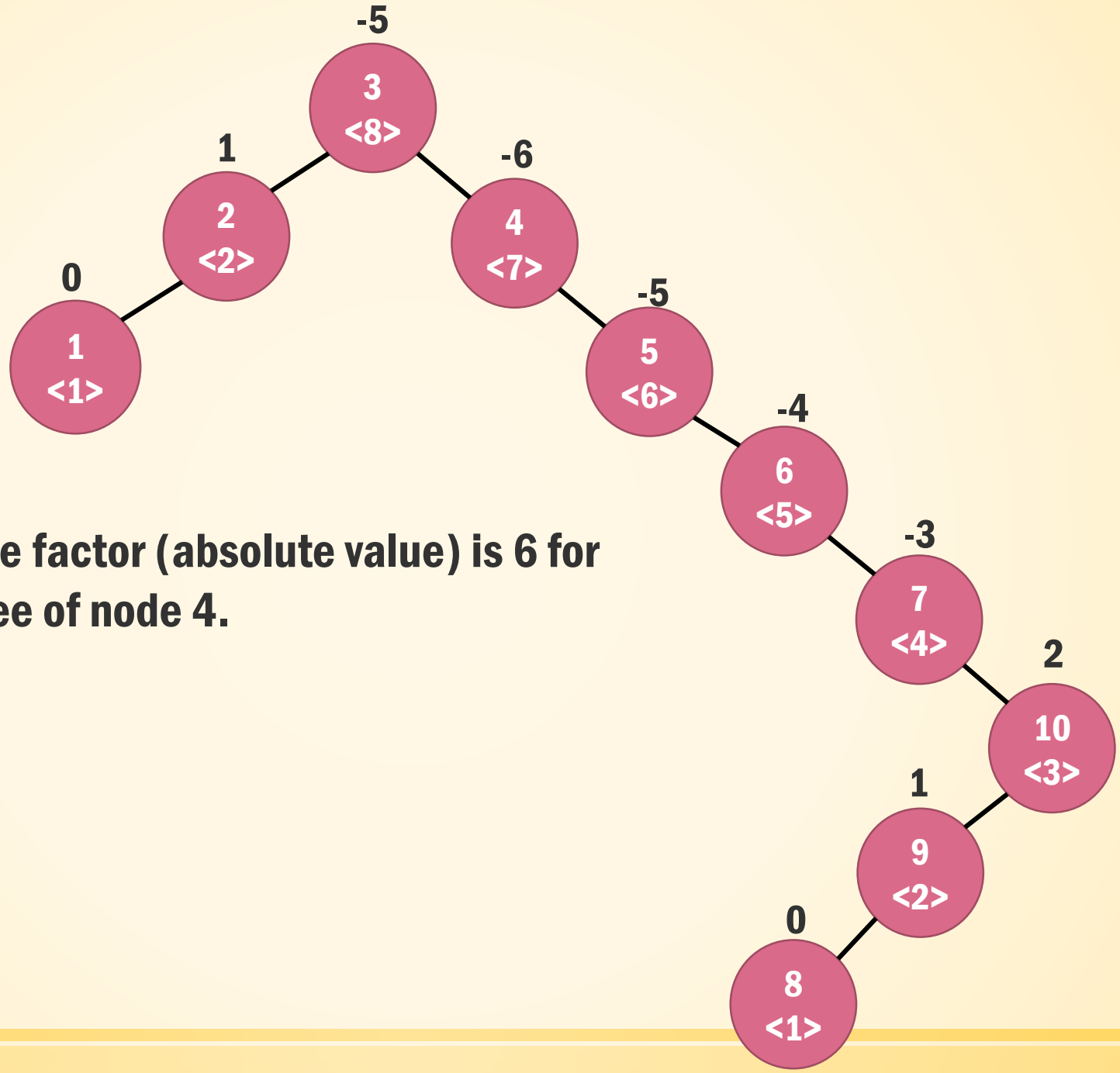
- Given an array a[10]={3,2,1,4,5,6,7,10,9,8}, answer the following questions.

a. Build a corresponding Binary Search Tree from index 0, and find the node with the largest balance factor (i.e. the absolute value of the difference of the height of the left subtree and the height of the right subtree).

b. Build an AVL Tree of the array from index 0, and list all the tree building steps.

# Q2_Ans (a)

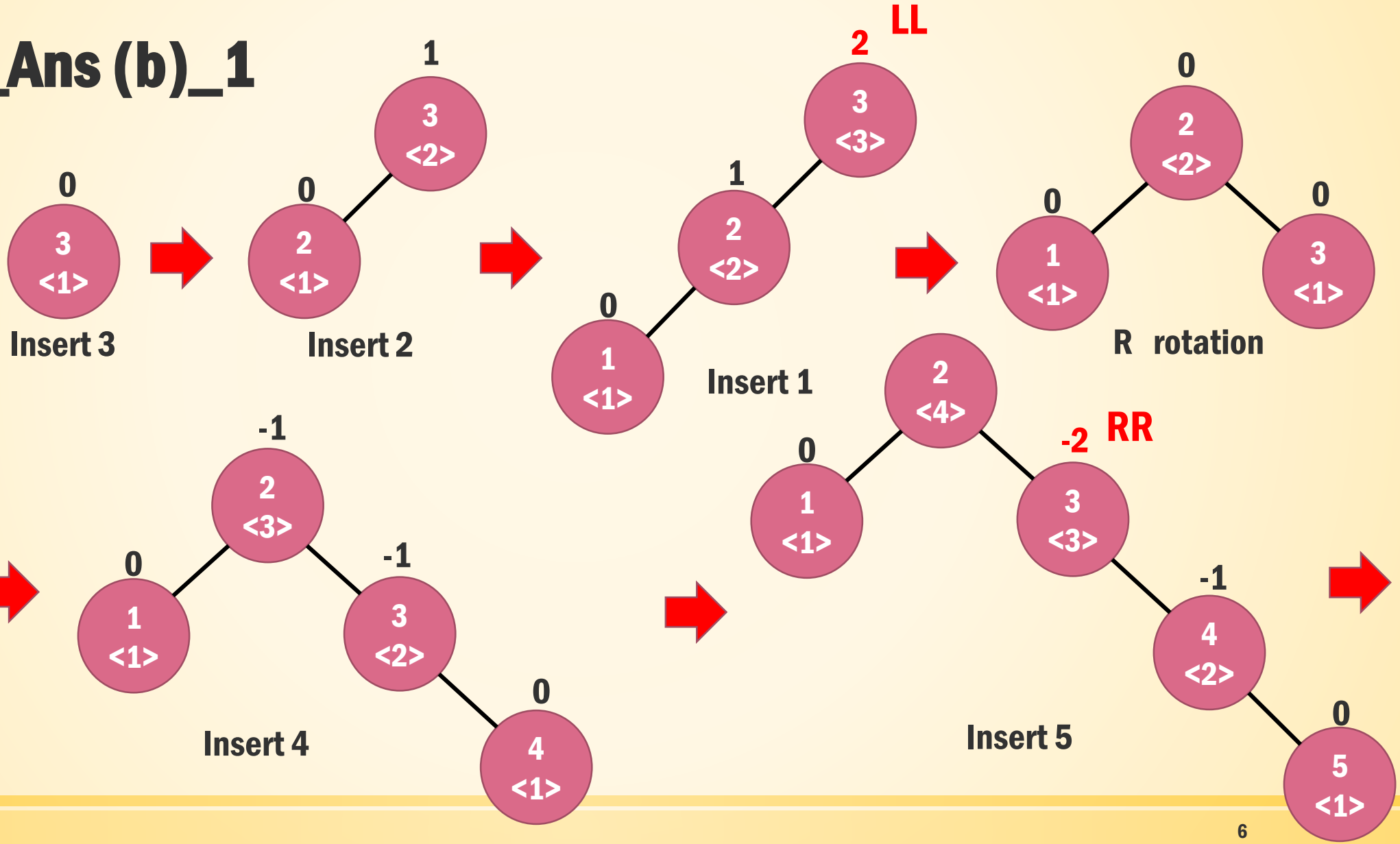
- a.




- Largest balance factor (absolute value) is 6 for the right subtree of node 4.

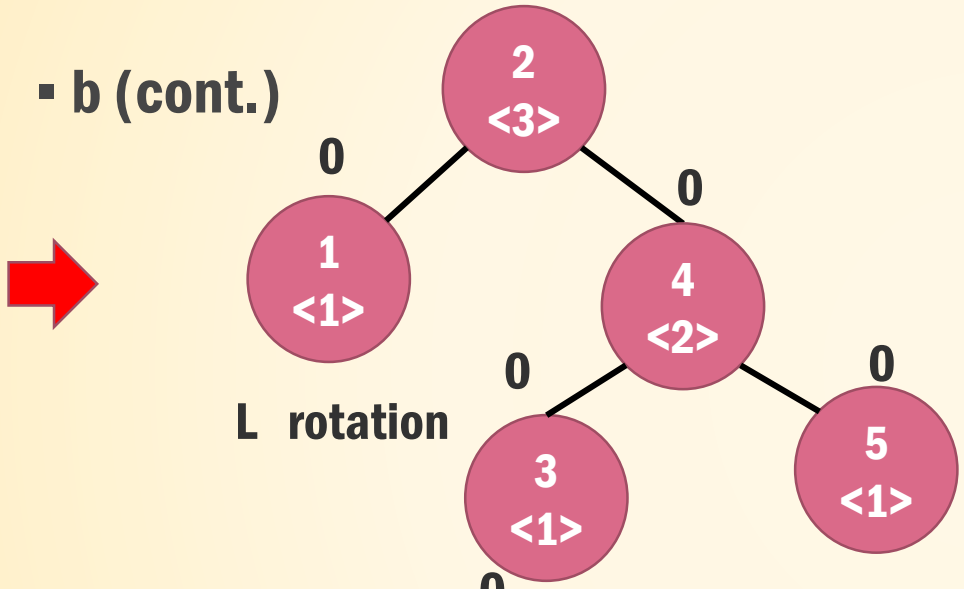
# Q2_Ans (b)_1

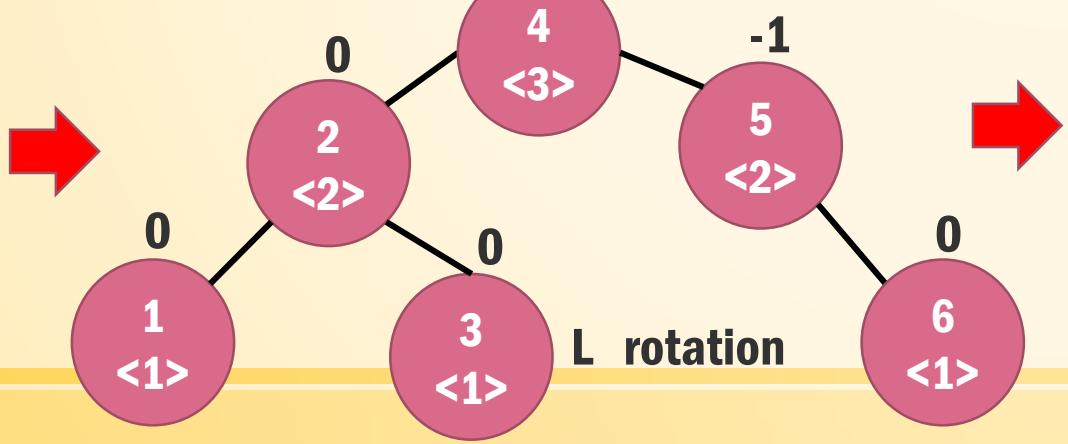
- b.

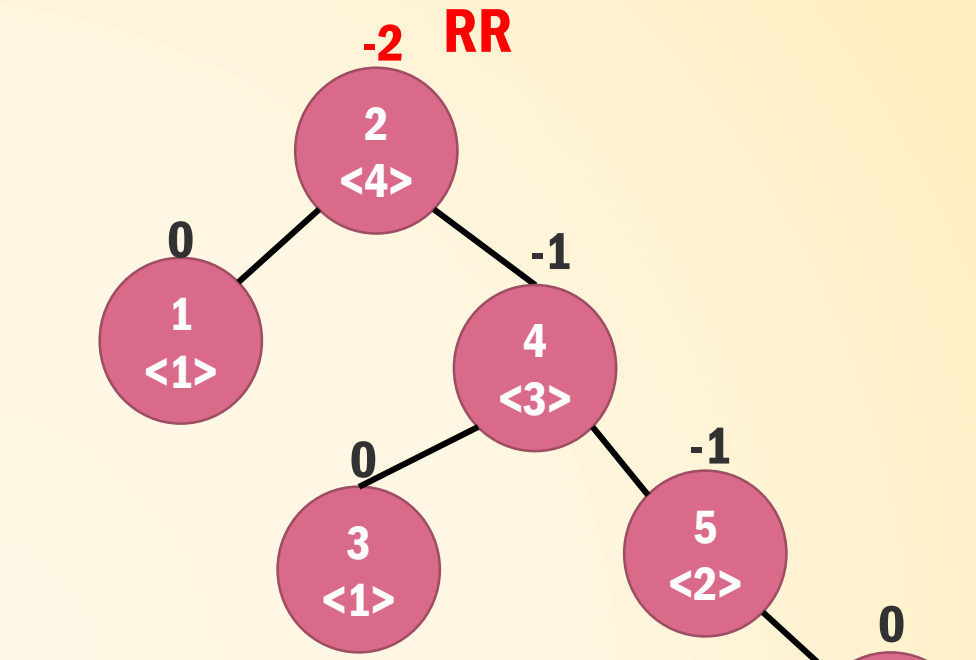

0
3
<1>
Insert 3
1
3
<2>
0
2
<1>
Insert 2
2
LL
3
<3>
1
2
<2>
0
1
<1>
Insert 1
0
2
<2>
0
1
<1>
3
<1>
R rotation
2
<4>
0
1
<1>
-2
RR
3
<3>
-1
4
<2>
0
5
<1>
Insert 5
-1
2
<3>
0
1
<1>
-1
3
<2>
0
4
<1>
Insert 4

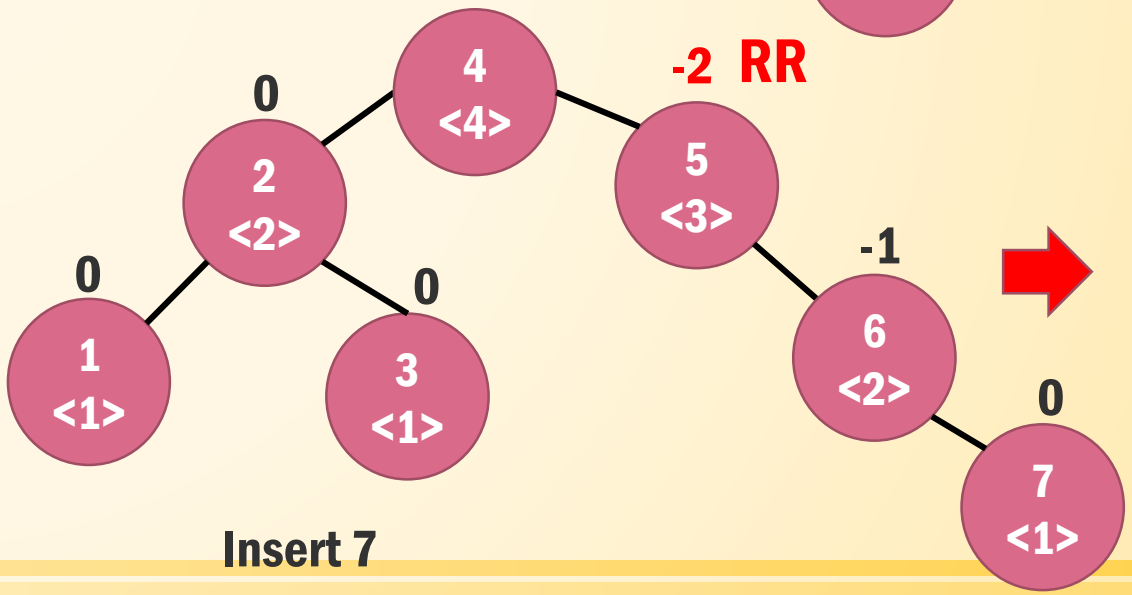
# Q2_Ans (b)_2 1

- b (cont.)



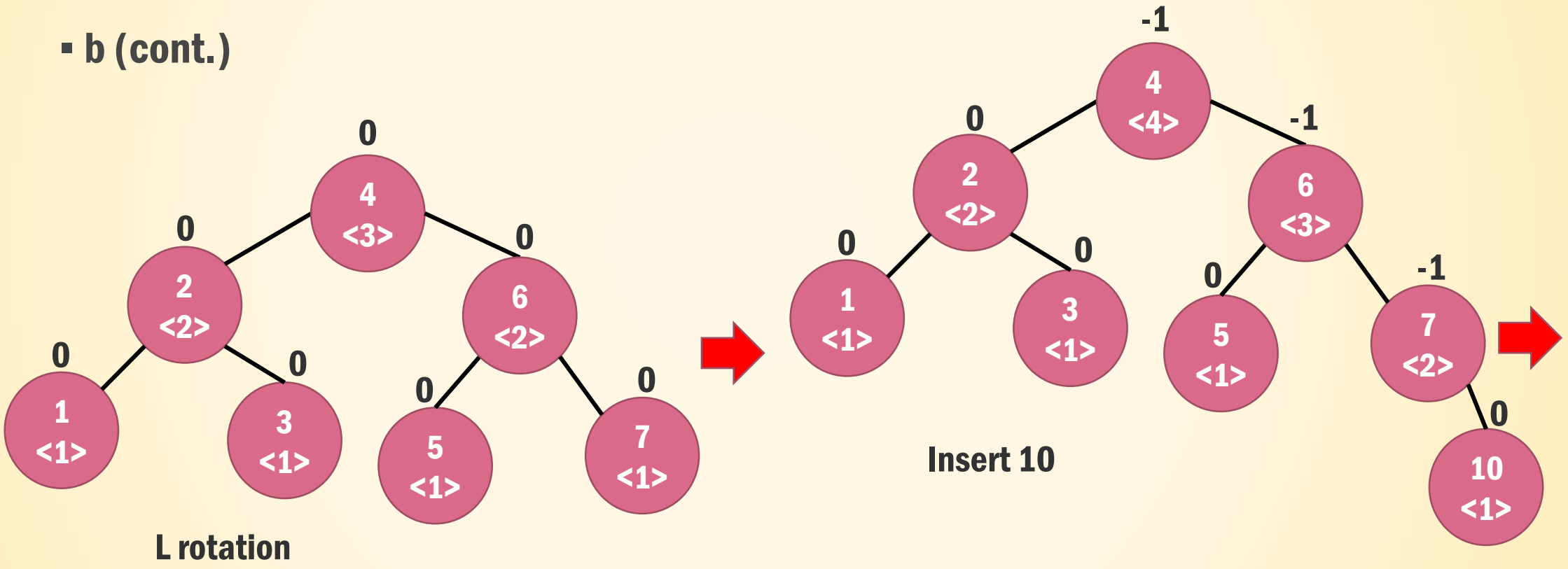



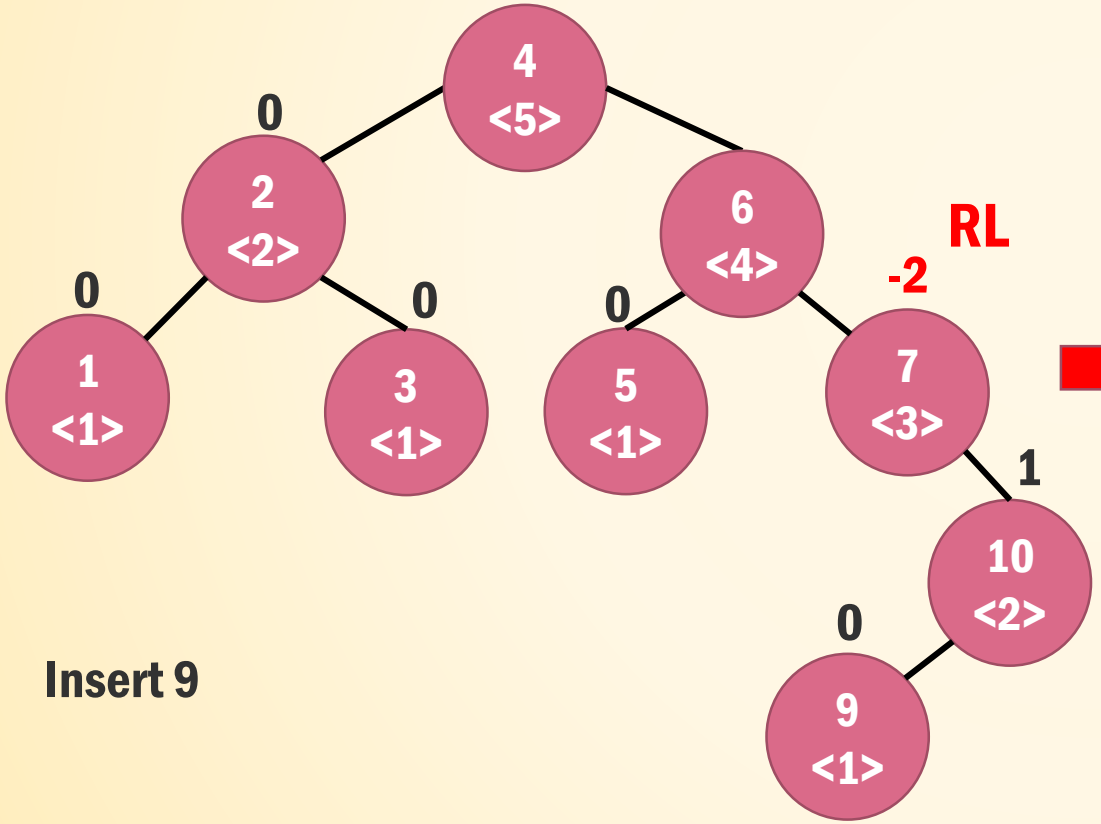
# Q2_Ans (b)_3

- b (cont.)

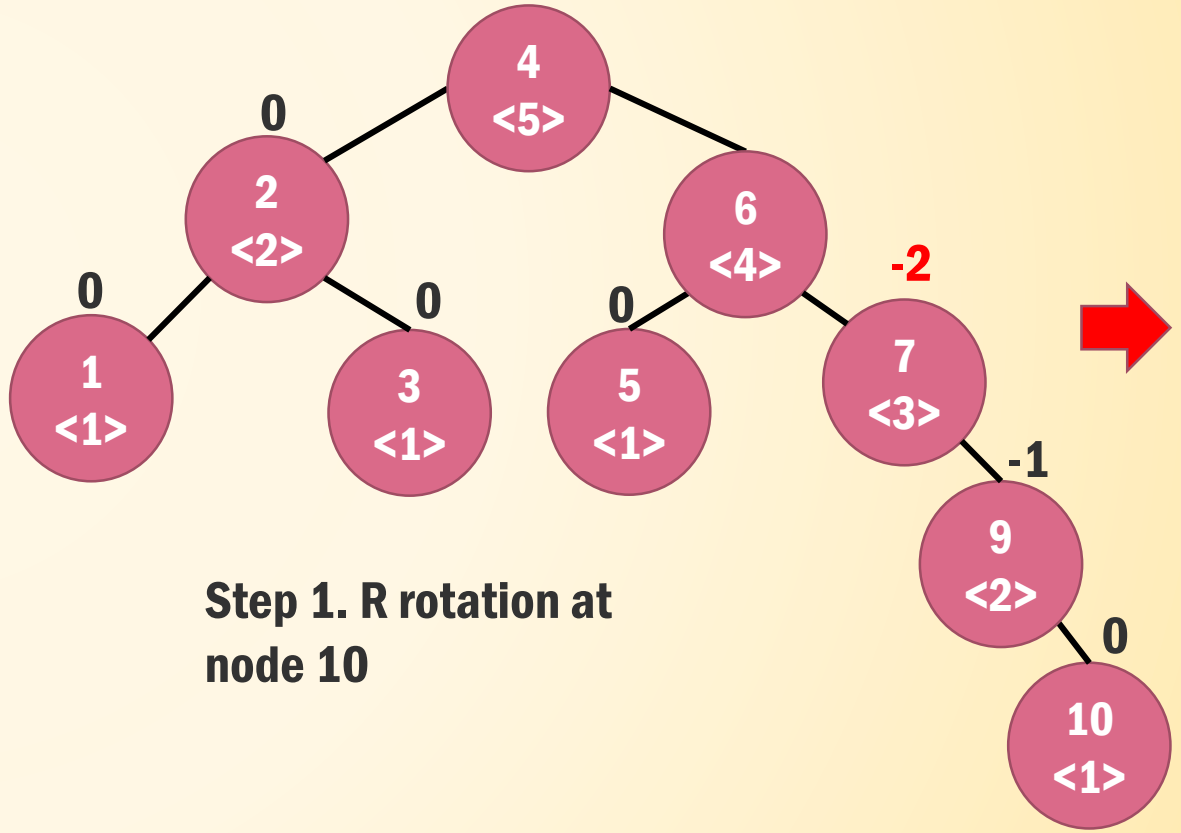


# Q2_Ans (b)_4

- b (cont.)



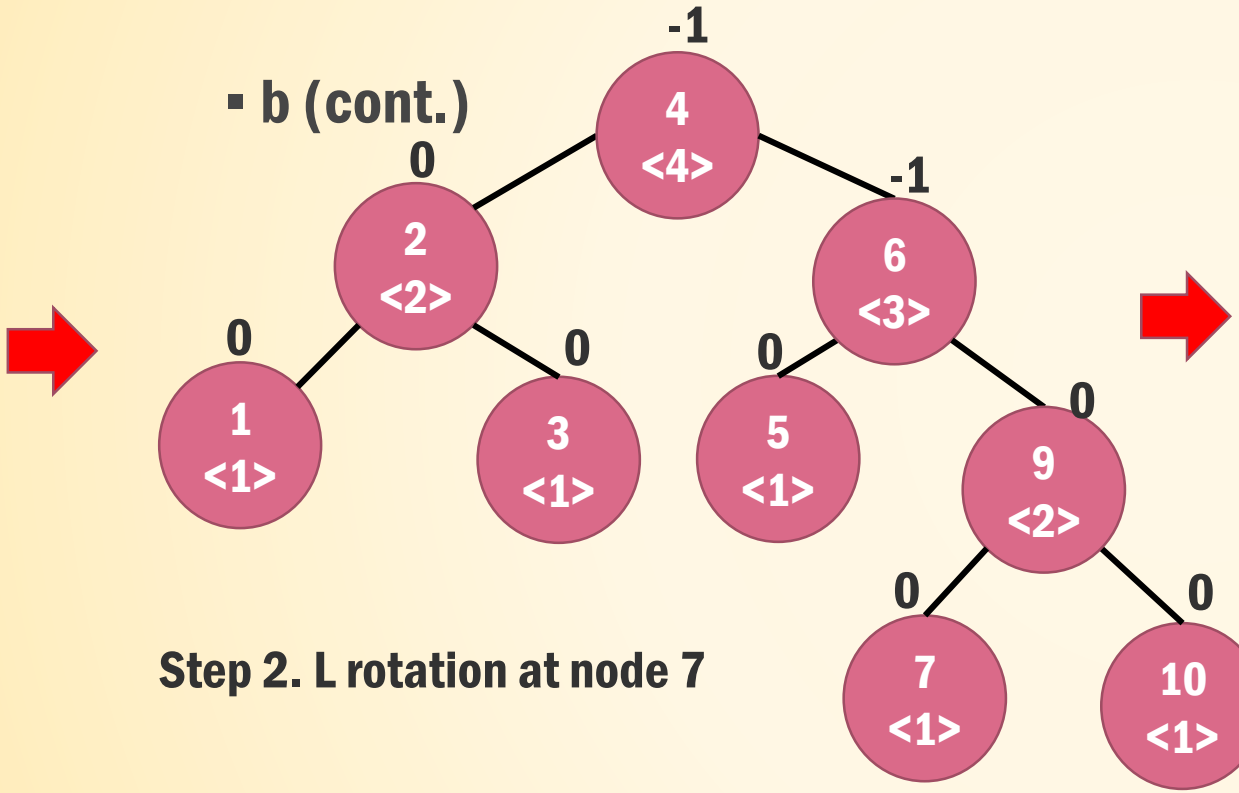

Insert 9



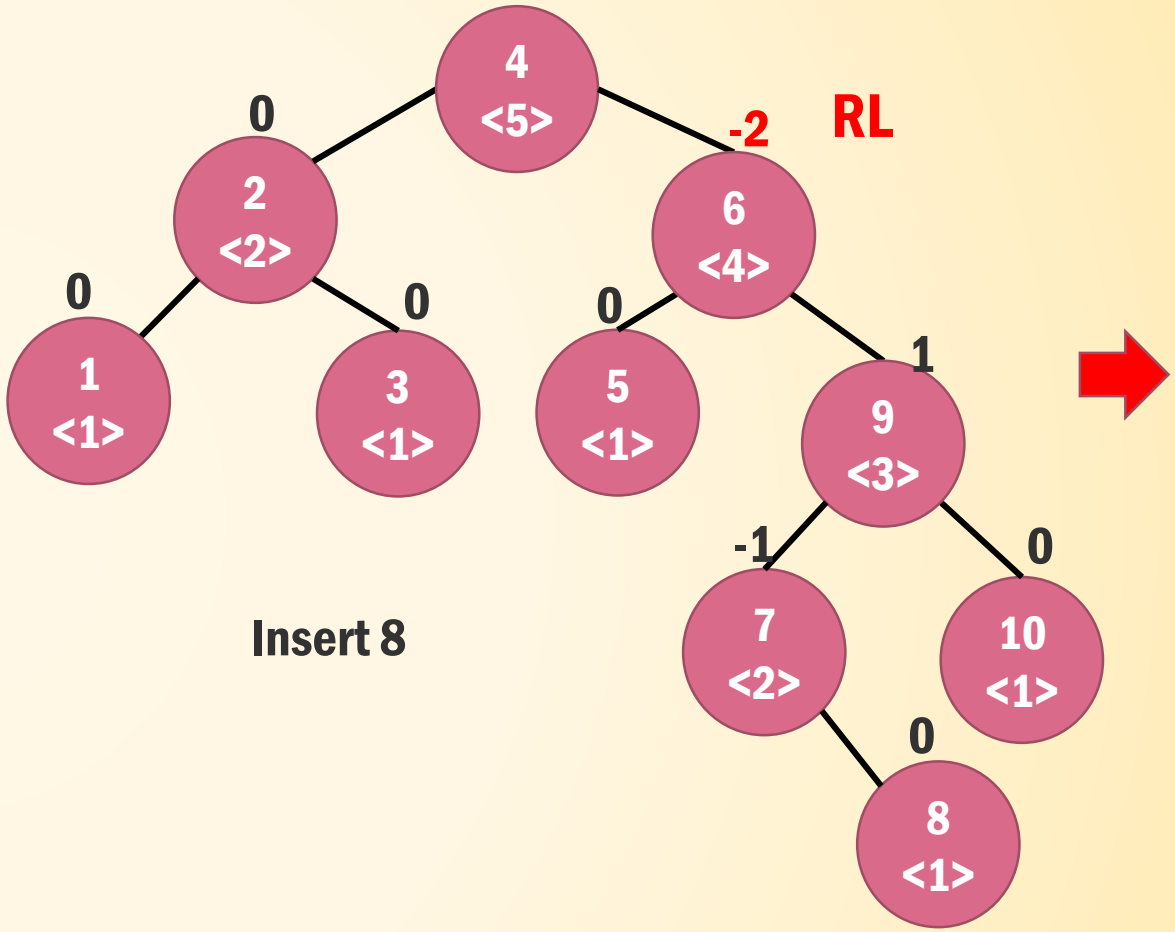

Step 1. R rotation at node 10

# Q2_Ans (b)_5

- b (cont.)



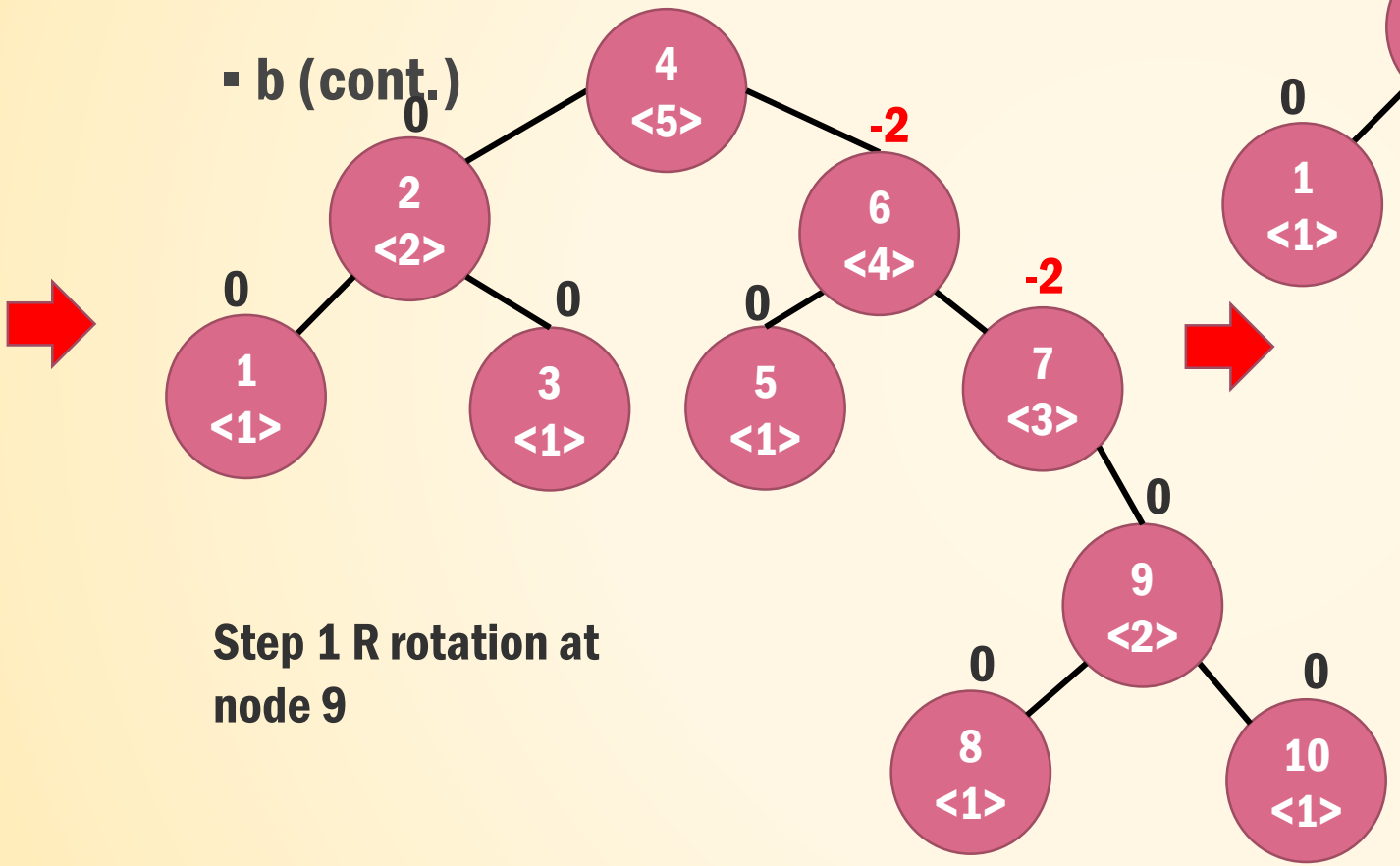

Step 2. L rotation at node 7



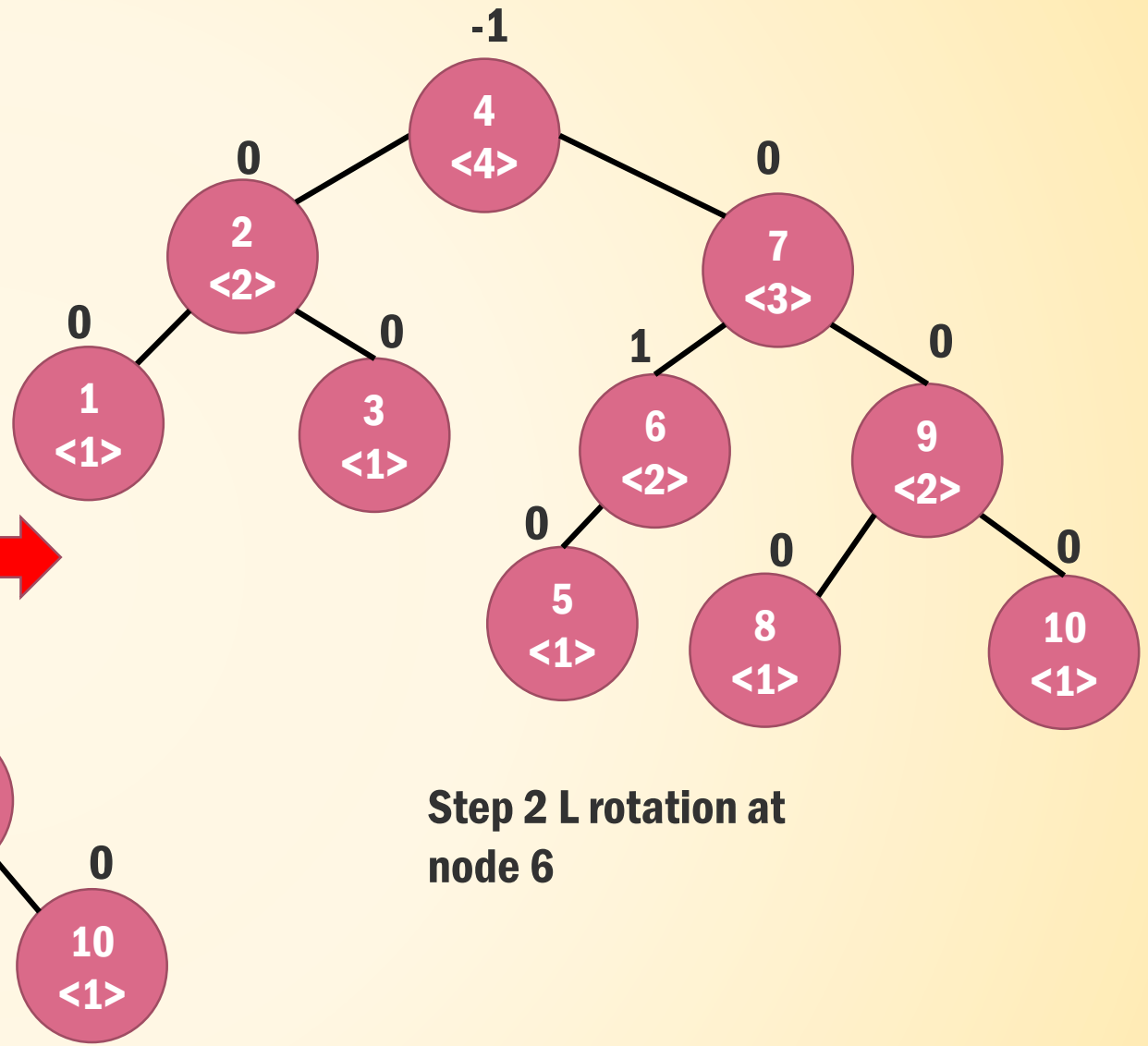

Insert 8

# Q2_Ans (b)_6

- b (cont.)



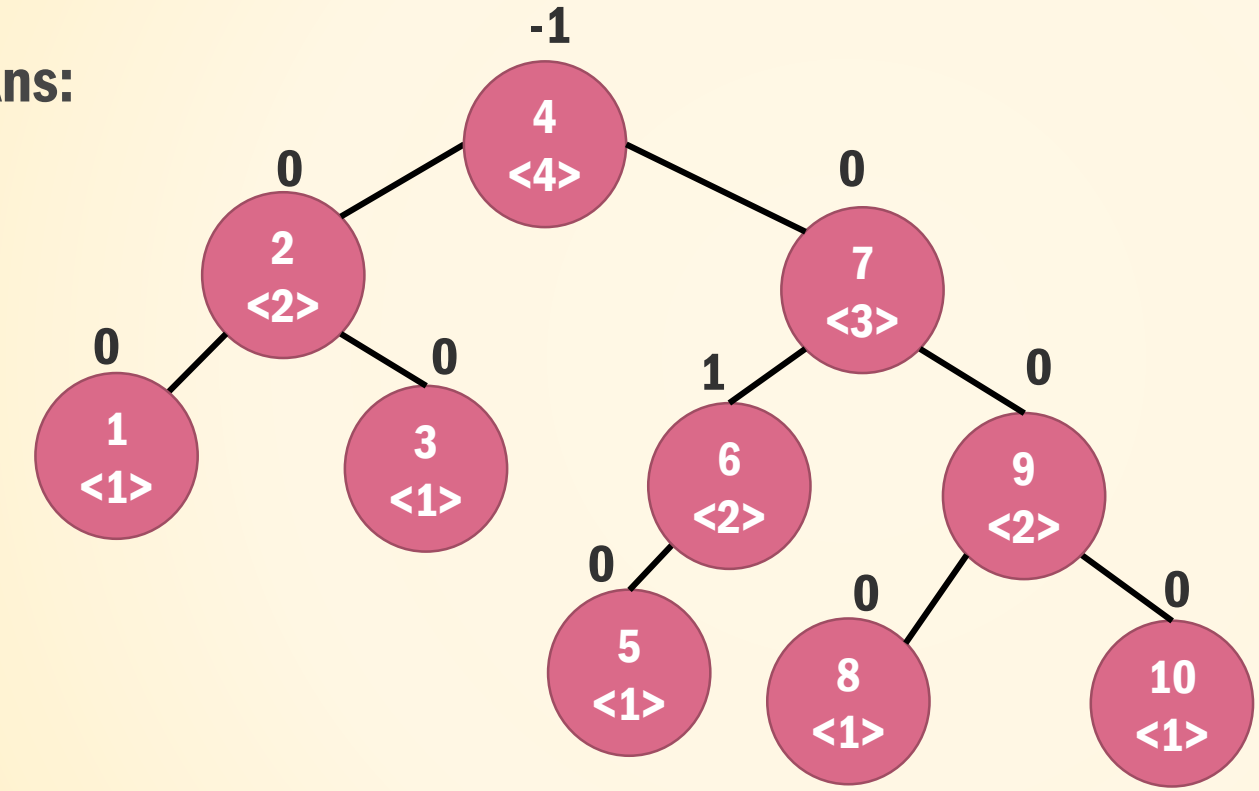

Step 1 R rotation at node 9




Step 2 L rotation at node 6

# Q2_Ans (b)_7

- b. Ans:



# Q3

- Given a partial AVL Tree code as follows, please answer the following questions.

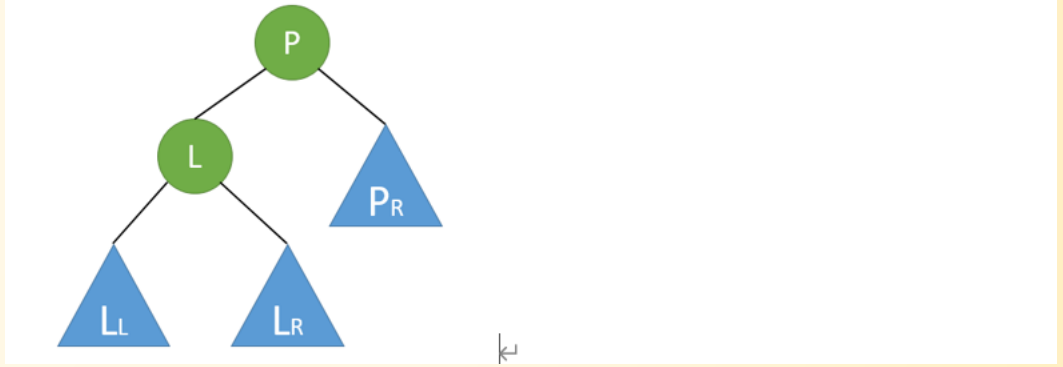
```
class BiTNode{
    private:
        int data;
        int bf;     //bf=balanced factor
        BiTNode *lchild,*rchild;
    public:
        BiTNode(); //Skip to define constructor
        int getBF(){
            return bf;
        }
        BiTNode* getLeft(){
            return this->lchild;
        }
        BiTNode* getRight(){
            return this->rchild;
        }
        void setLeft(BiTNode *node){
            this->lchild=node;
        }
        void setRight(BiTNode *node){
            this->rchild=node;
        }
};

class AVLTree{
    private:
        BiTNode *nodeArray;
    public:
        AVLTree(); // Skip to define constructor
        void R_Rotate(BiTNode **P);
        void L_Rotate(BiTNode **P);
        void LeftBalance(BiTNode **T);
```

# Q3

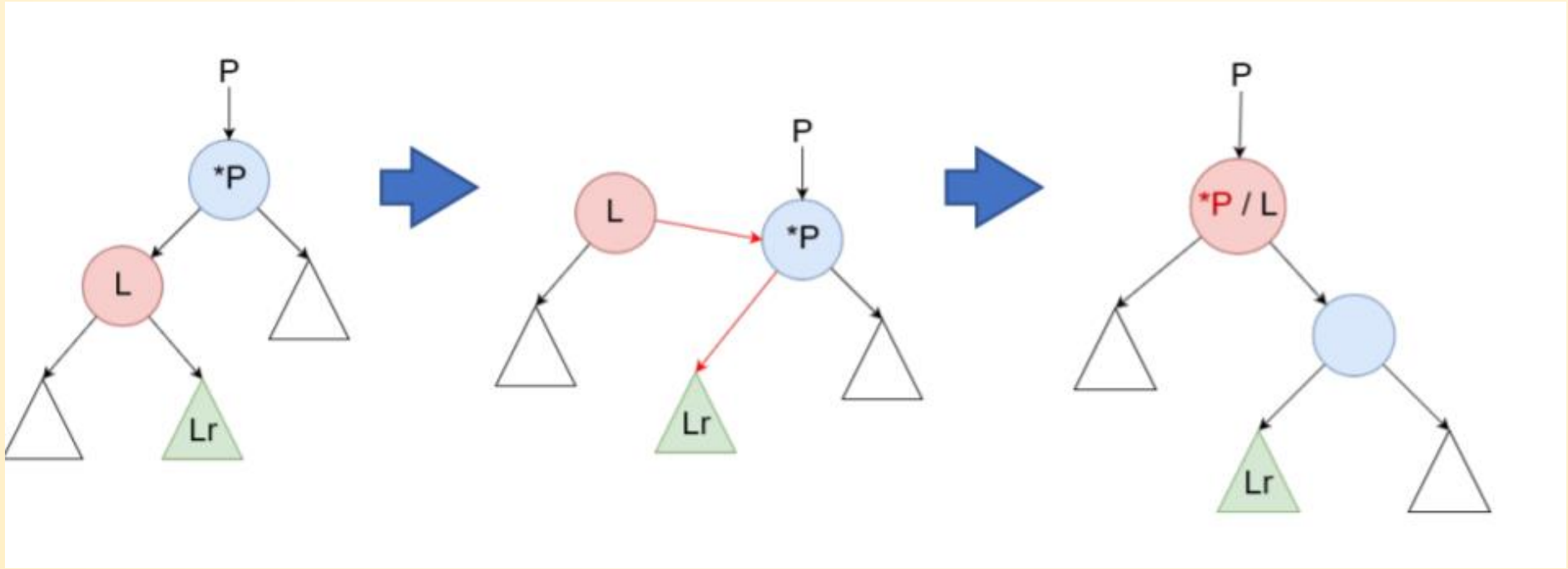
- a. Please explain R_Rotate(BiTNode *P)
- b. Design L_Rotate(BiTNode **P).
- c. Try to implement an AVL tree with insert function

```
void AVLTree::R_Rotate(BiTNode **P){
    BiTNode *L;
    L=(*P)->getLeft();
    (*P)->setLeft(L->getRight());
    L->setRight(**P);
    *P=L;
}
```



# Q3_Ans (a)

- **a.** 右旋，首先取得原根節點 ***P**的左子節點**L (**下圖左**)**，接著把 ***P** 設為左子樹 **L** 的右子樹 **(**下圖中**)**；最後把 ***P** 指向旋轉後的新節點 **L (**下圖右**)**。



# Q3_Ans (b)

- b.

```
void AVLTree::L_Rotate(BitNode**P){
    BitNode *R;
    R = (*P)->getRight();
    (*P)->setRight(R->getLeft());
    R->setLeft(**P);
    *P=R;
}
```

# Q3_Ans (c)

- c. **Please refer to**: https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

# Q4

- Following the previous question, please complete the LeftBalaced(BiTNode **T)

```
#define LH +1  // Left height
#define EH 0   // Equal height
#define RH -1  // Right height

/*對以指標 T 所指節點為根的二元樹做左旋轉平衡處理*/
/*本演算法結束時，指標 T 指向新的根節點*/
void AVLTree::LeftBalance(BiTNode **T){
    BiTNode* L,Lr;
    L=(*T)->getLeft;
    switch(L->getBF()){
        /* 檢查 T 的左子樹平衡度，並做相應的平衡處理 */
        case LH:  /* 新節點插入在 T 的左孩子的左子樹上，要做右旋轉處理 */
            // TODO(1)
        case RH:  /* 新節點插入在 T 的左孩子的右子樹上，要做雙旋轉處理 */
            // TODO(2)
    }
}
```

# Q4_Ans

```
void AVLTree::LeftBalance(BiTNode **T){
    BiTNode* L,Lr;
    L=(*T)->getLeft;
    switch(L->getBF()){
        case LH:  /* 新節點插入在 T的左孩子的左子樹上，要做右旋轉處理 */
            // TODO(1)
            R_Rotate(T);
        case RH:  /* 新節點插入在 T的左孩子的右子樹上，要做雙旋轉處理 */
            // TODO(2)
            L_Rotate(&L);
            R_Rotate(T);
    }
}
```

# Q5

- Can any node in an AVL Tree has the balanced factor equal to ±3? Why?

# Q5 Ans

- No, the balance factor should be -1, 0 or 1, or the tree will not be balanced.

# Q6

(1)Explain the concept of the red-black tree. What are the differences between the red-black tree and the AVL tree.

(2)Describe the rules of insertion and deletion.

(3)Build a red-black tree following the sequence 10,20,30,15

# Q6_Ans (1)_1

- (1)
- Red black colored node definition:

  RB1: The root and all external nodes are **black**.

  RB2: No root-to-external-node path has two consecutive red nodes.

  RB3: All root-to-external-node paths have the same number of **black** nodes.
- Red black colored pointer (edge) definition:

  RB1': Pointer to an external node is **black**.

  RB2': No root-to-external-node path has two consecutive red pointers (edges).

  RB3': All root-to-external-node paths have the same number of **black** pointers.

參考影片：https://youtu.be/4WjwmHeKa1Q

# Q6_Ans (1)_2

- (1) (continue)
- Red black tree vs AVL tree

  a. AVL tree沒有分紅黑色

  b. Red-black tree 沒有要求達到complete balanced

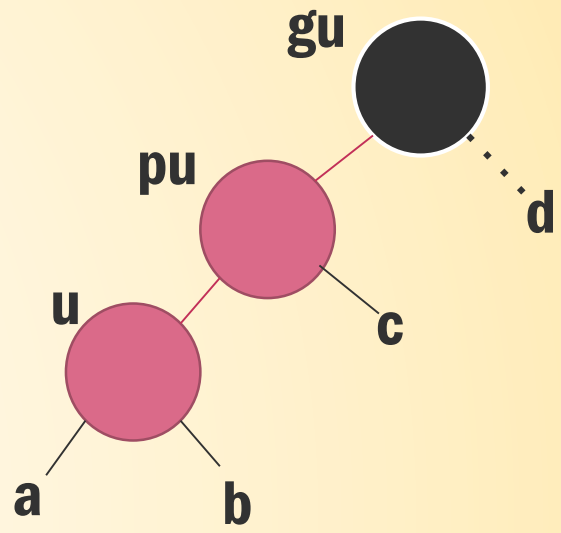
  c. AVL tree 的search 比較快

  d. Red-black tree 的insertion 和deletion 比較快

  e. AVL tree 的node多存了balance factor，所以需要O(n)的extra space

# Q6_Ans (2)_1: Insertion

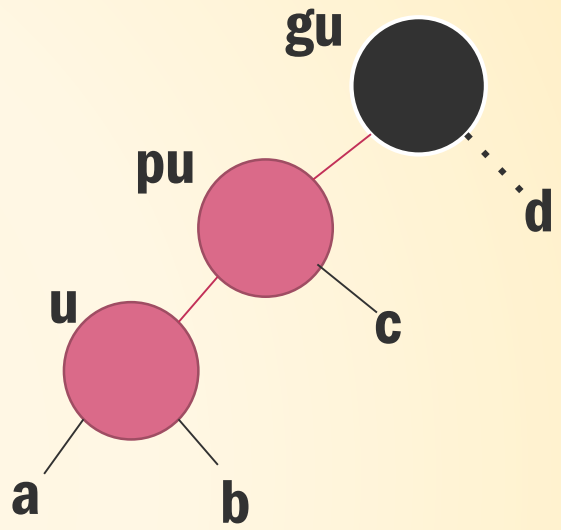
- (2) Inserting into a Red-Black Tree:
- A new element u is first **inserted** as the ordinary binary search tree.
- Assign the new node to **red**.
- The new node may or may not violate RB2 (**imbalance**).
  - One root-to-external-node path may have two consecutive red nodes.
  - It can be handled **by changing colors or a rotation**.



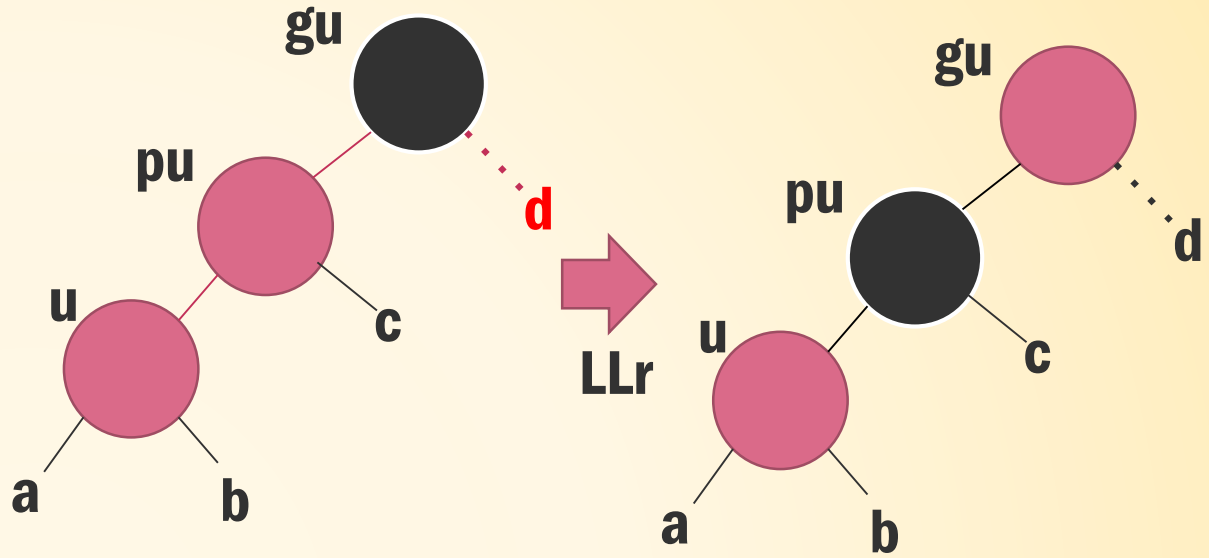

參考影片：https://youtu.be/fP1taNiz7ZI

# Q6_Ans (2)_2 : Insertion

- u: new node, red
- pu: parent of u, red
- gu: grandpa of u, black
- LLb, LLr
  - Left child, then left child
  - LLb: the other child of gu, d, is black
  - LLr: the other child of gu, d, is red
- LRb, LRr
- RRb, RRr
- RLb, RLr (8 case)



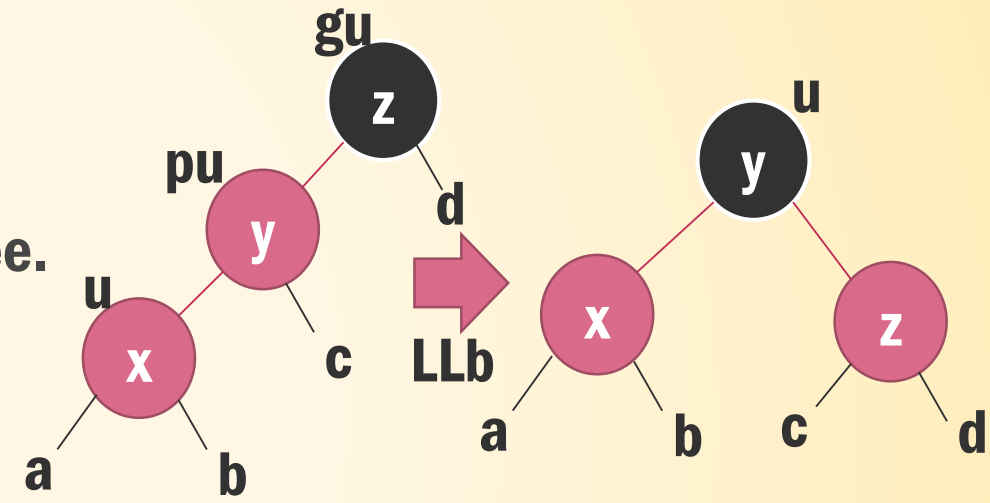
# Q6_Ans (2)_3 : Insertion

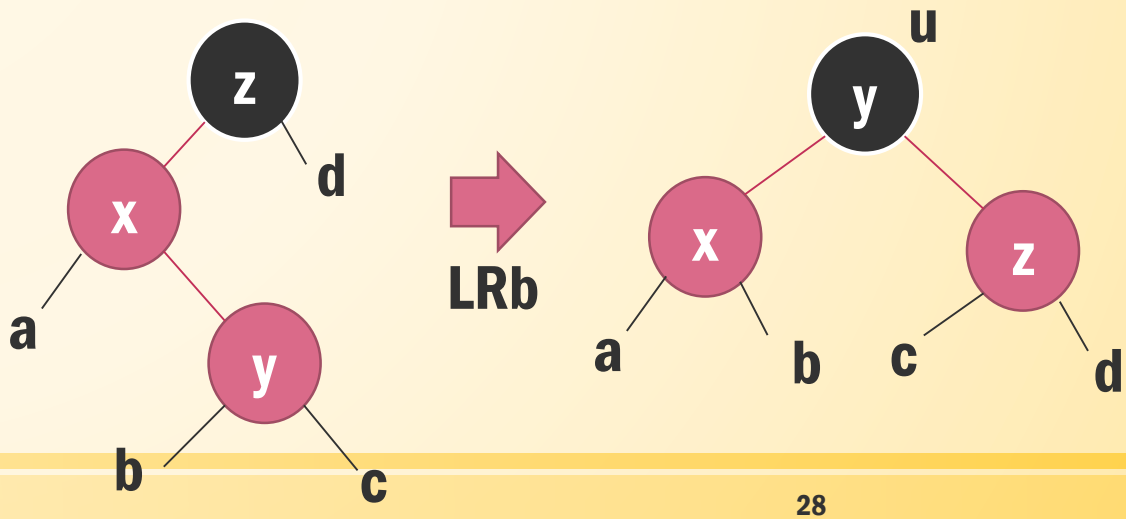



- Color Change of LLr, LRr, RRr, RLr
- If red gu violated by RB2, then original gu becomes new u, and up two levels set pu, gu to be solved by 8 case.
  - Continue rebalancing if necessary.
    - If RB2 is satisfied, stop propagation.
    - If gu is the root, force gu to be black (The number of black node on all root-to-external-node paths increased by 1.)
    - Otherwise, continue color change or rotation.

# Q6_Ans (2)_4 : Insertion

- Rotation and Color Change of LLb, LRb, RRb, RLb
- Same as the rotation schemes taken for an AVL tree.
- LLb rotation:
  - LL rotation of AVL tree
- LRb rotation:
  - LR rotation of AVL tree
- RRb is symmetric to LLb.
- RLb is symmetric to LRb.

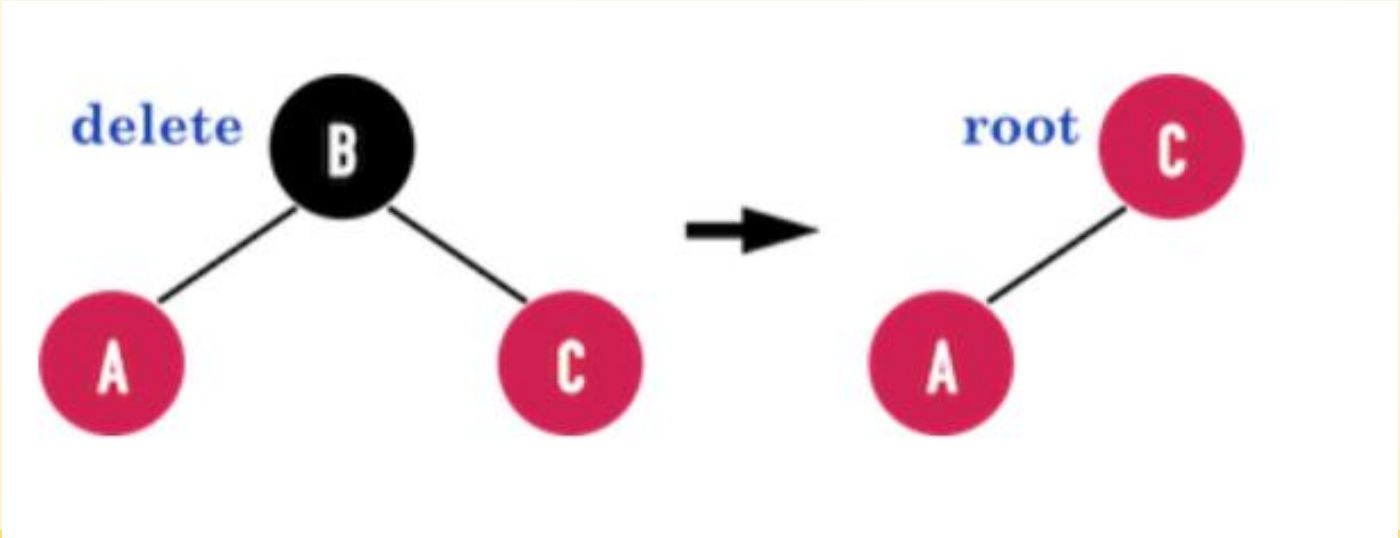


# Q6_Ans (2)_1: Deletion

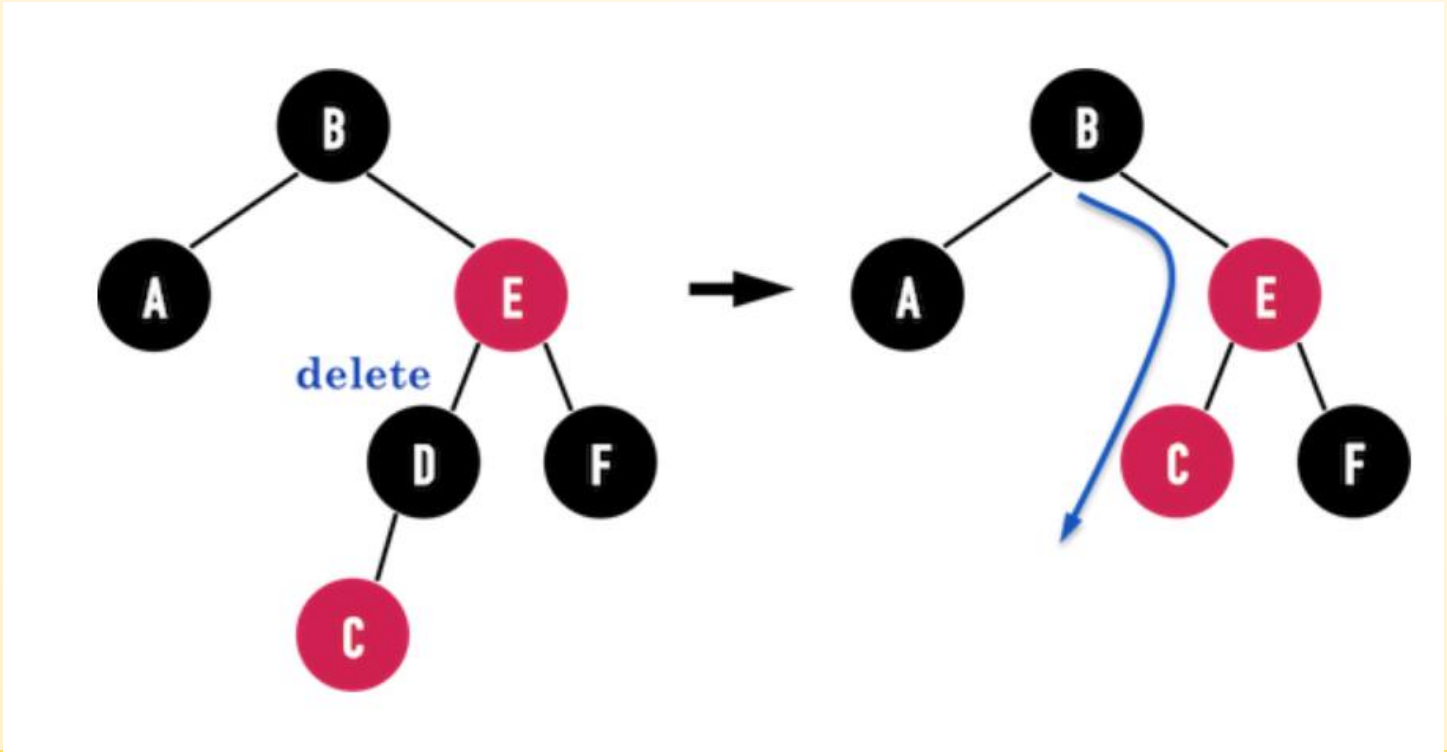
- **(2) Deletion:**
  - 在**RBT**中執行**Delete(**刪除資料**)**時，若刪除之**node**為黑色，有可能違反三點**RBT**特徵：

1. 若要刪除的**node**恰好為**root**，而刪除後恰好是紅色的**node**遞補成為新的**root**，此時便違反**RBT**特徵：**root**一定要是黑色；**(RB1)**



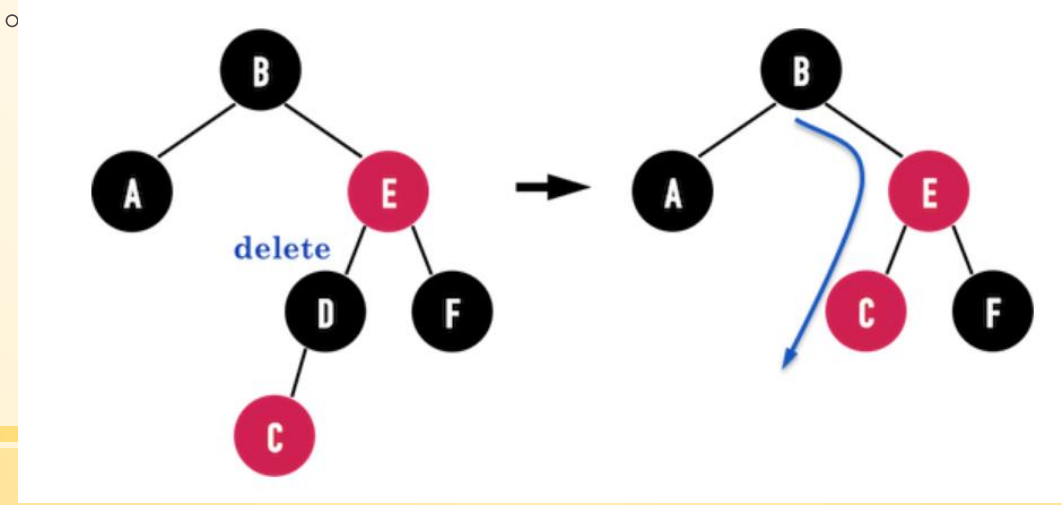
# Q6_Ans (2)_2: Deletion

2. 若刪除**node**後，出現紅色與紅色**node**相連之情形，則違反**RBT**特徵：紅色**node**之**child**一定要是黑色；**(RB2)**



# Q6_Ans (2)_3: Deletion

3. **若刪除之node是黑色**，而且恰好不是**root**，那麼所有包含被刪除node的**path上之黑色node數必定會減少**，將會違反**RBT**特徵:「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」**(RB3)**
   - 圖左：從**root:node(B)**出發至任意**leaf**的**path**上都有三個黑色**node(**包含**NIL)**；
   - 圖右：刪除**node(D)**後，**path:node(B)-node(E)-node(C)**上之黑色**node**數剩下**2**個**(**包含**NIL)**。

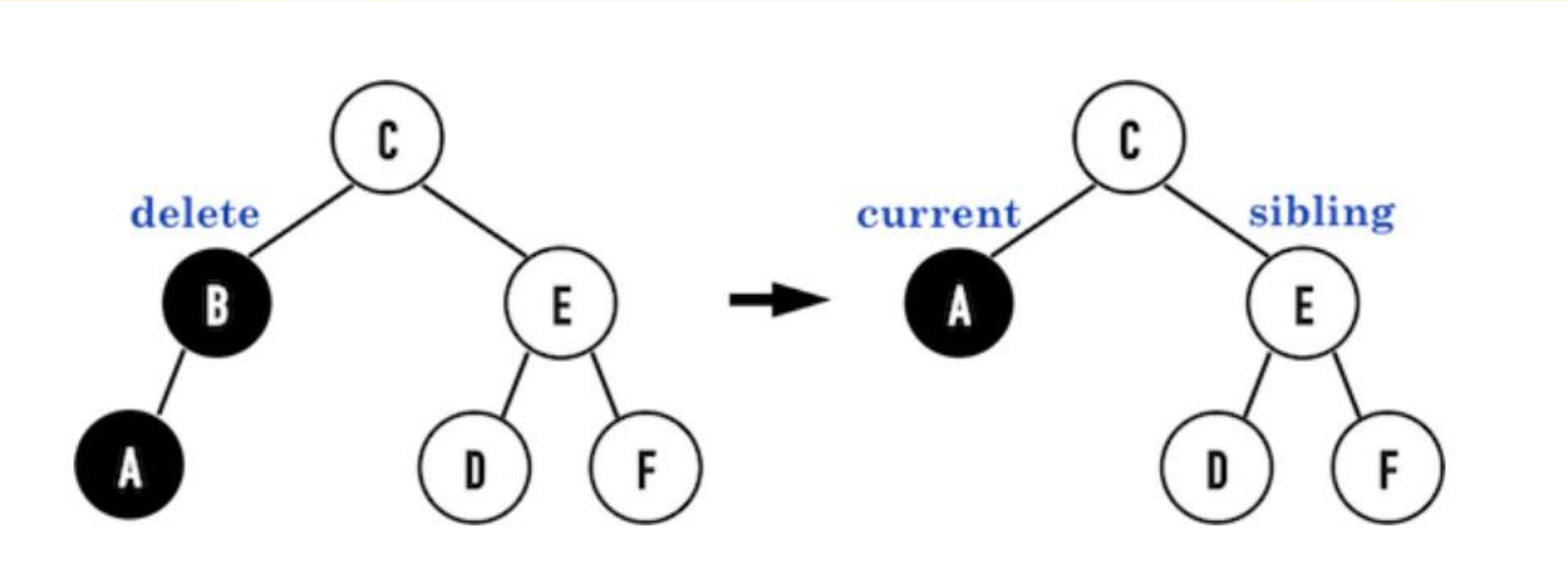


# Q6_Ans (2)_4: Deletion

- 第一部分，如同**DeleteBST()**，依照欲刪除之**node**的**child**個數分成三種情形處理：
  - 先確認**BST**中有沒有要刪除的**node**；
  - 把要刪除的**node**調整成「至多只有一個**child**」；
  - 把要刪除的**node**的**child**指向新的**parent**；
  - 把要刪除的**node**的**parent**指向新的**child**；
  - 若實際上刪除的是「替身」，再把替身的資料放回**BST**中；
- 第二部分，若刪除的**node是黑色**，需要進行修正**(Fix-Up)**，引進函式：**DeleteFixedUpRBT()**。

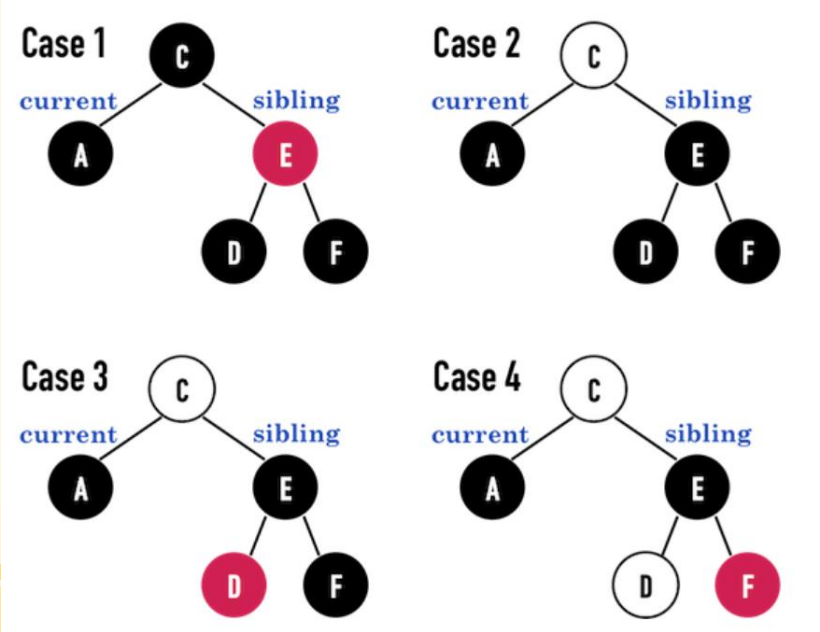
# Q6_Ans (2)_5: Deletion

- **DeleteFixUpRBT()**
  - 考慮在圖之**RBT**中刪除**node(B)**，由於**node(B)**是黑色，必定違反**RBT**之特徵，因此需要修正。
    - **(**以下圖示中，白色的**node**表示顏色可能為黑色也可能為紅色，而且可能是一棵**subtree**或是**NIL**，需視情況而定。**)**



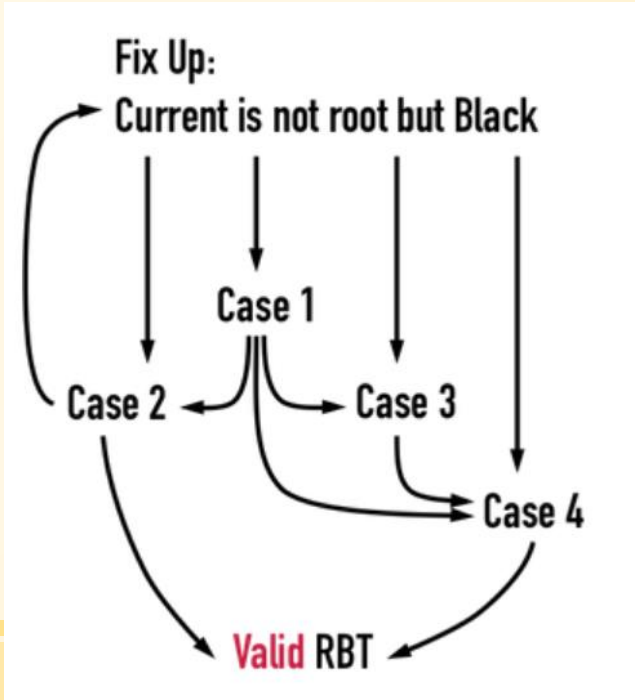
# Q6_Ans (2)_6: Deletion

- 根據**sibling**之顏色與**sibling**之**child**之顏色，可以分為下列四種情形**(Case)**
  - **Case1**：**sibling**為紅色；
  - **Case2**：**sibling**為黑色，而且**sibling**的兩個**child**都是黑色；
  - **Case3**：**sibling**為黑色，而且**sibling**的**rightchild**是黑色，**leftchild**是紅色；
  - **Case4**：**sibling**為黑色，而且**sibling**的**rightchild**是紅色。



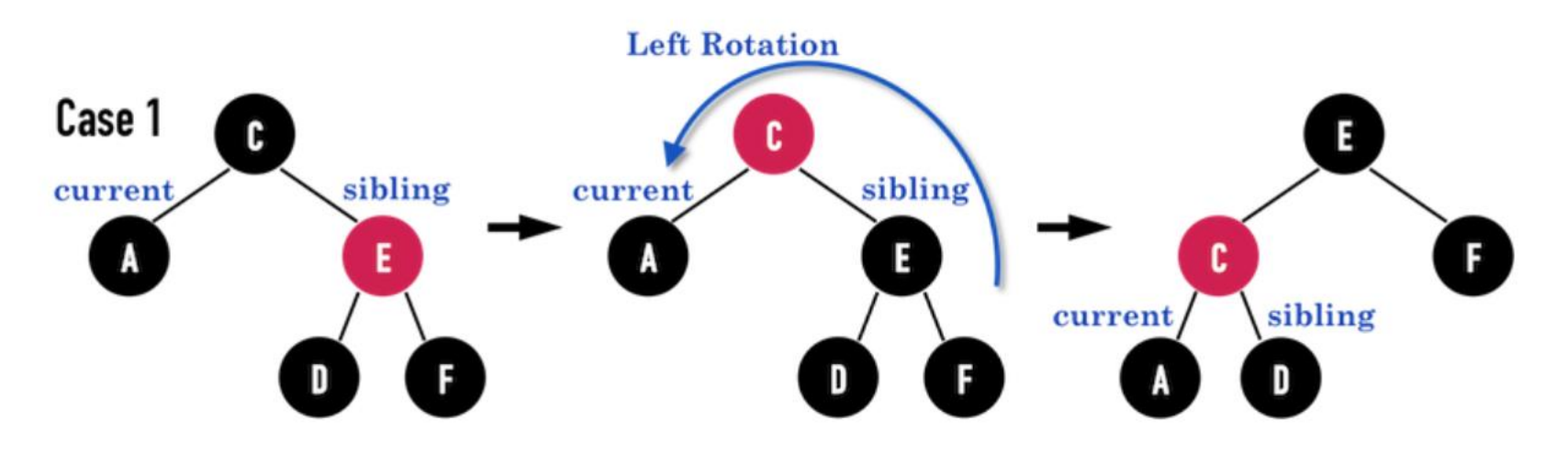
# Q6_Ans (2)_7: Deletion

- **DeleteFixUpRBT()**的情形**(Case)**較為複雜，圖是所有情形之循環圖：
  - 若**current**是黑色的，而且**current**不為**root**，則依情況進入四個**Case**；
  - 若進入**Case1**，修正後，將進入**Case2**、**Case3**或**Case4**；
  - 若進入**Case2**，有可能修正後即符合**RBT**特徵，也有可能根據新的**current**之情形重新判斷起；
  - 若進入**Case3**，修正後必定進入**Case4**；
  - 若進入**Case4**，修正後必定符合**RBT**之特徵



# Q6_Ans (2)_8: Deletion

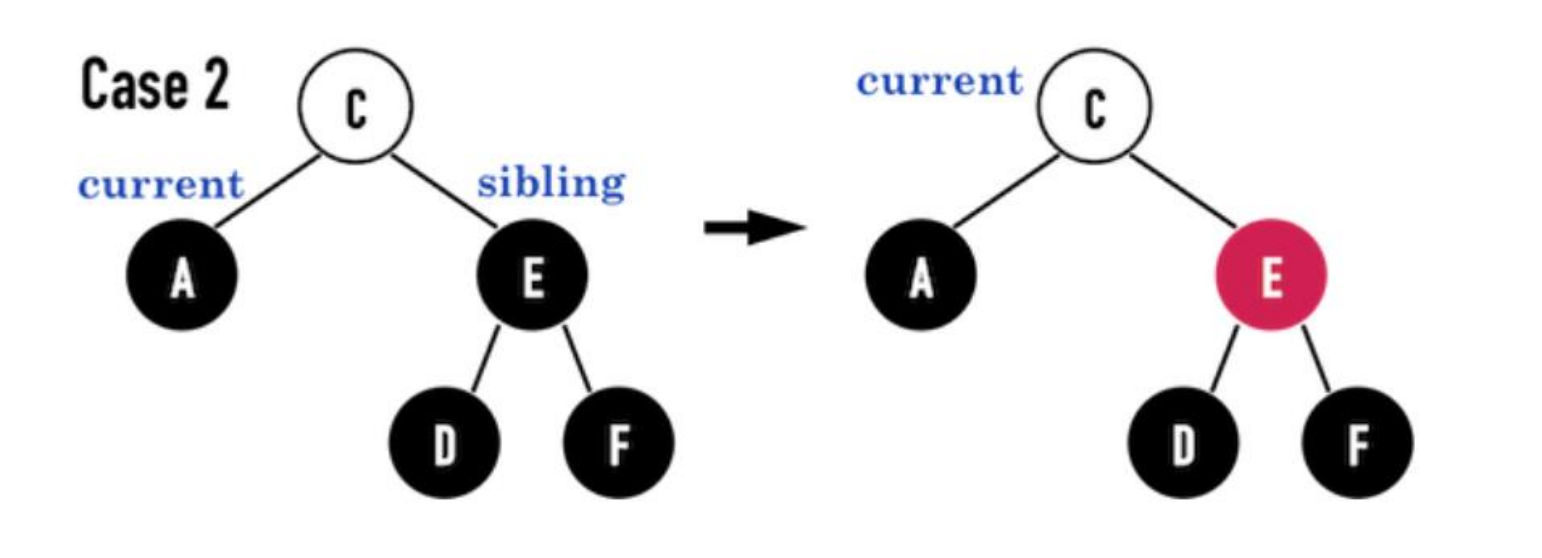
- **Case 1:**
  - 若**sibling**為紅色，修正方法如下：
    - 將**sibling**塗成黑色：**node(E)**塗成黑色；
    - 將**current**之**parent**塗成紅色：**node(C)**塗成紅色；
    - 對**current**之**parent**做**Left Rotation**：對**node(C)**做**Left Rotation**；
    - 將**sibling**移動到**current->parent**的**right child**：將**sibling**指向**node(D)**。




- 將進入**Case2**、**Case3**或**Case4**。

# Q6_Ans (2)_9: Deletion

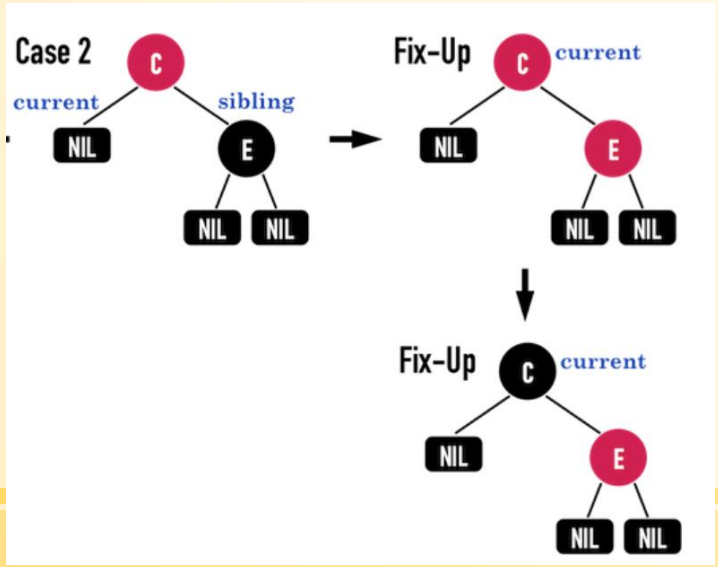
- **Case 2:**
  - 若**sibling**為黑色，並且**sibling**之兩個**child**皆為黑色，修正的方法如下
    - 將**sibling**塗成紅色：**node(E)**塗成紅色；
    - 將**current**移至**current**的**parent**：**current**移至**node(C)**。

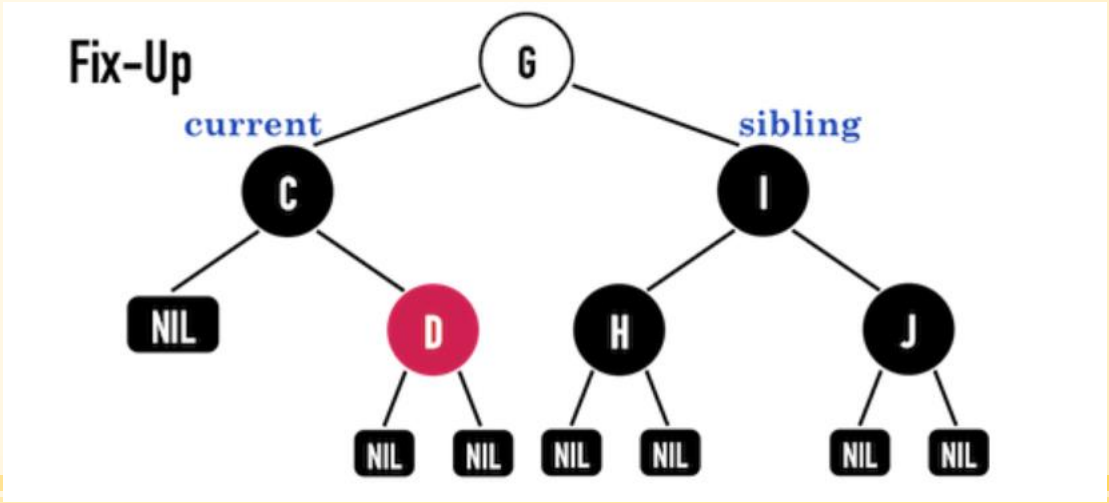



- 經過上述步驟，根據新的current: node(C)之顏色，可以分成兩種情形：

# Q6_Ans (2)_10: Deletion

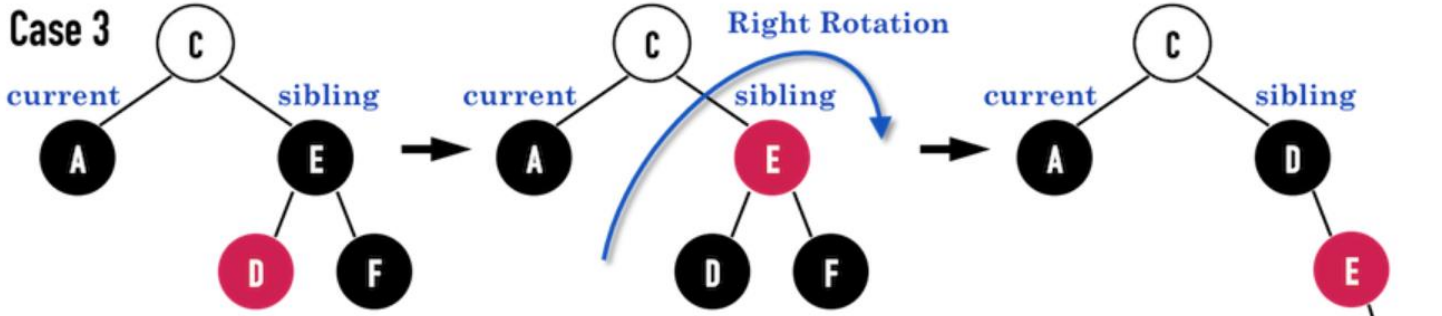
- 若**node(C)**為紅色，則跳出迴圈，把**node(C)塗黑**，即可滿足**RBT**特徵，其邏輯便是：將從**node(C)**出發往**leftchild**與**rightchildpath**的黑色數目調整成與刪除之前相同；
- 若**node(C)**為**黑色**，且**node(C)**不是**root**，則繼續下一輪迴圈，重新判斷其屬於四種情況之何者並修正，如圖右，從**node(G)**出發至任意**descendant leaf**之**path**上的黑色**node**數並不完全相同。



# Q6_Ans (2)_11: Deletion

- **Case 3:**
  - 若**sibling**為黑色，並且**sibling**之**rightchild**為黑色，**leftchild**為紅色，修正的方法如下：
    - 將**sibling**之**leftchild**塗成黑色：**node(D)**塗成黑色；
    - 將**sibling**塗成紅色：**node(E)**塗成紅色；
    - 對**sibling**進行**Right Rotation**：對**node(E)**進行**Right Rotation**；
    - 將**sibling**移至**current->parent**的**rightchild**：將**sibling**移至**node(D)**。



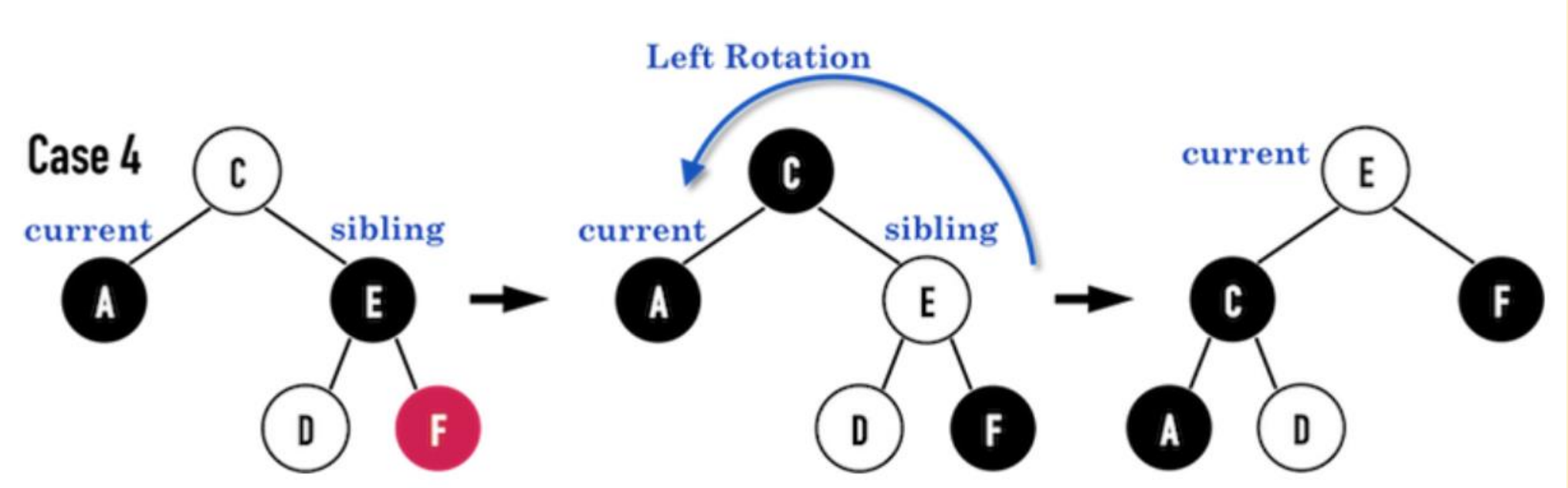

- 經過以上修正步驟，sibling之rightchild成為紅色，便進入Case4。

# Q6_Ans (2)_12: Deletion

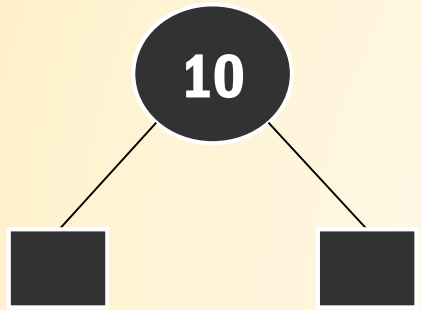
- **Case 4:**
  - 若**sibling**為黑色，並且**sibling**之**rightchild**為紅色，修正的方法如下：
  - 將**sibling**塗成**current**之**parent**的顏色：
    - 若**node(C)**是紅色，則將**node(E)**塗成紅色；
    - 若**node(C)**是黑色，則將**node(E)**塗成黑色；
  - 將**parent**塗成黑色：**node(C)**塗成黑色；
  - 將**sibling**之**rightchild**塗成黑色：**node(F)**塗成黑色；
  - 對**parent**進行**Left Rotation**：對**node(C)**做**Left Rotation**；
  - 將**current**移至**root**，把**root**塗黑。

**(Note: node(E)**未必是**RBT**之**root)**

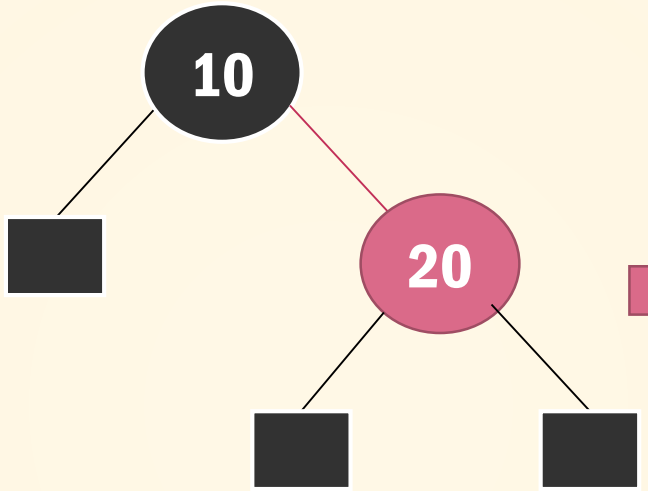
# Q6_Ans (2)_13: Deletion



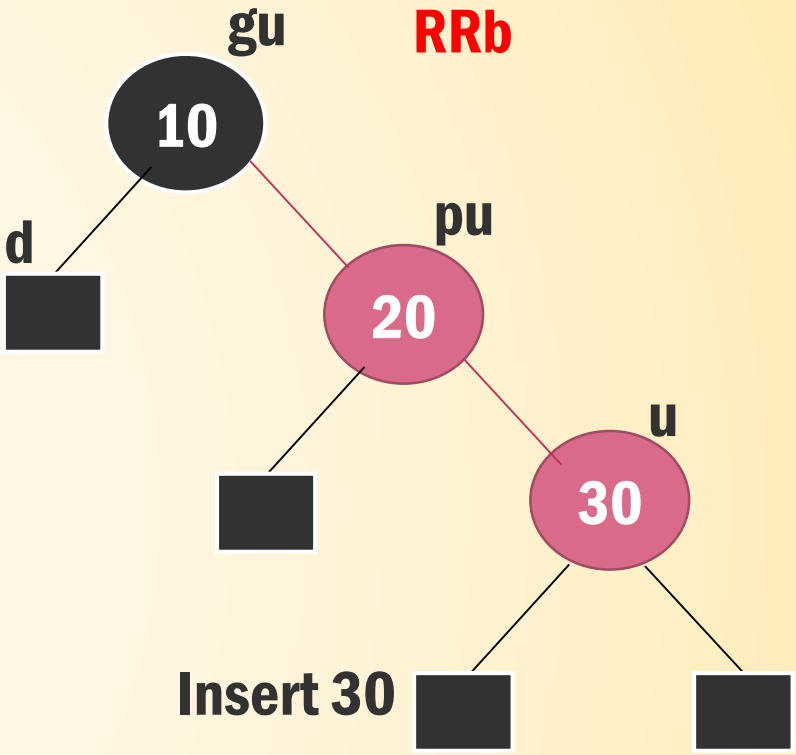
# Q6_Ans (3)_1




Insert 10

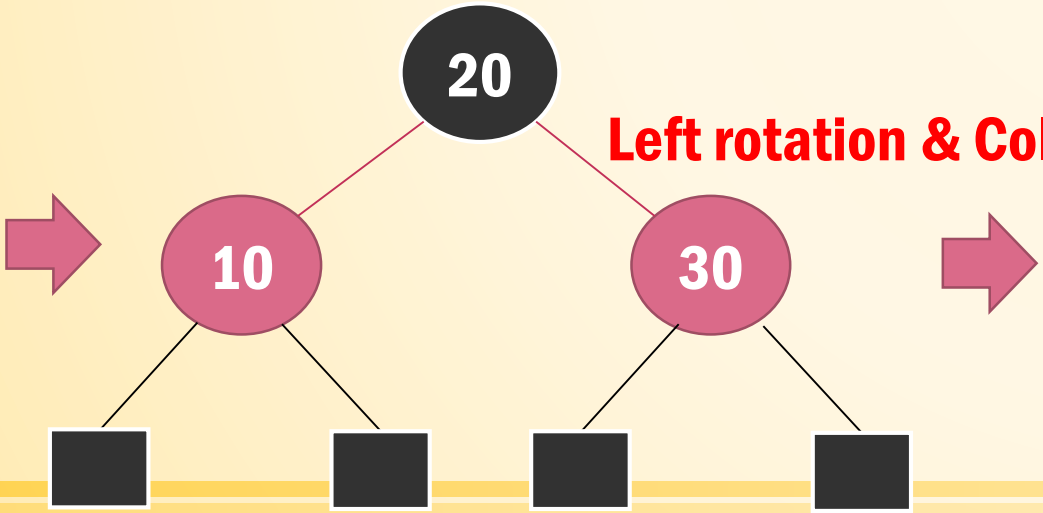



Insert 20

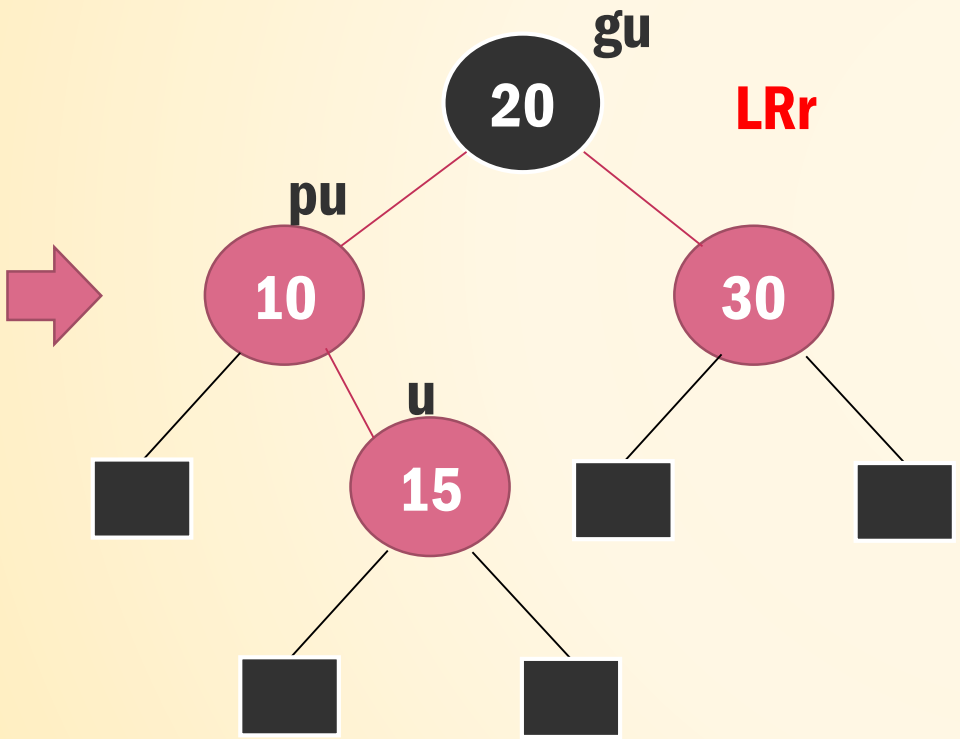



Insert 30

Left rotation & Color Change



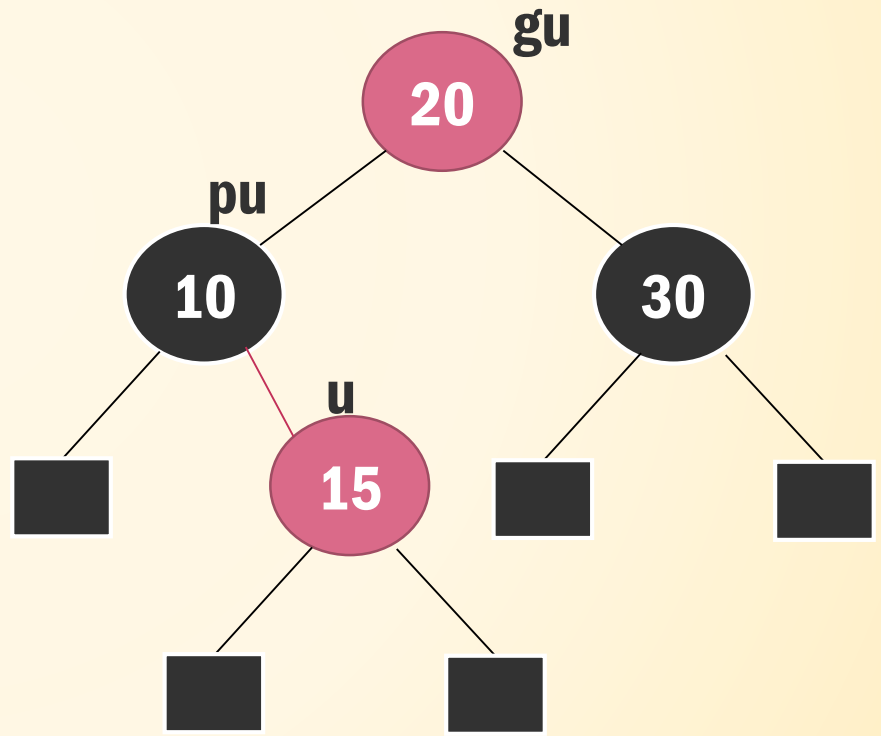
# Q6_Ans (3)_2




LRr

Insert 15

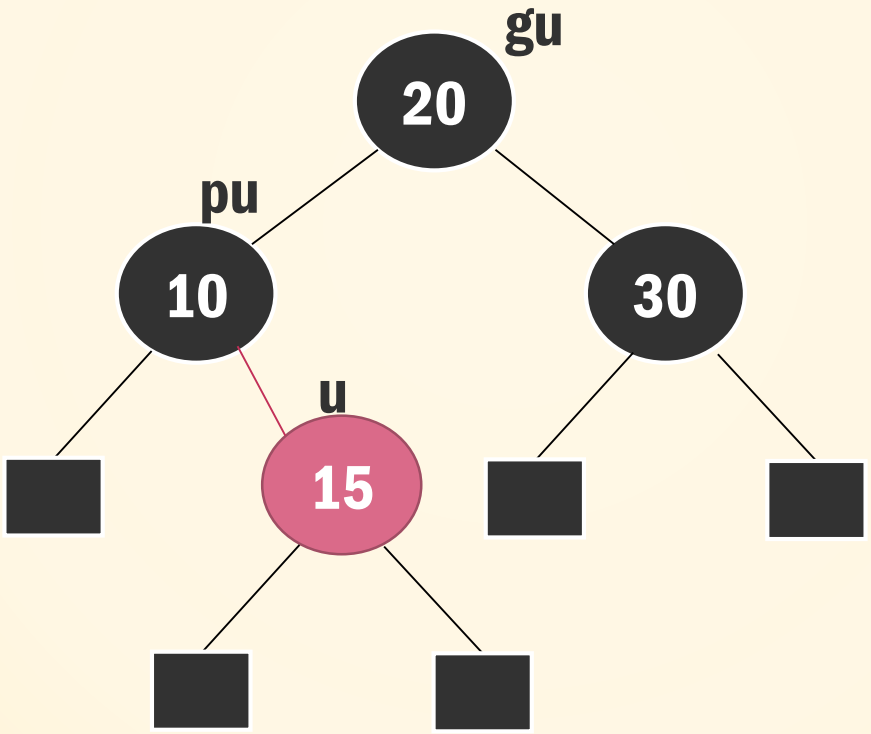
Color Change




Violate RB1 (root is black)

# Q6_Ans (3)_2

- Ans:



# Q7

- How to handle duplicated keys in an AVL tree. Describe your method.

# Q7 Ans

- 與原本 **AVL tree** 的結構相似，以下為有所更動的內容：

**(1)** 資料結構：在每個 **node** 中新增 **int count**，用來儲存數字的數目。

**(2)** 新增 **node**：除了原本應該初始化的資料，還需將 **count** 設為 **1**。

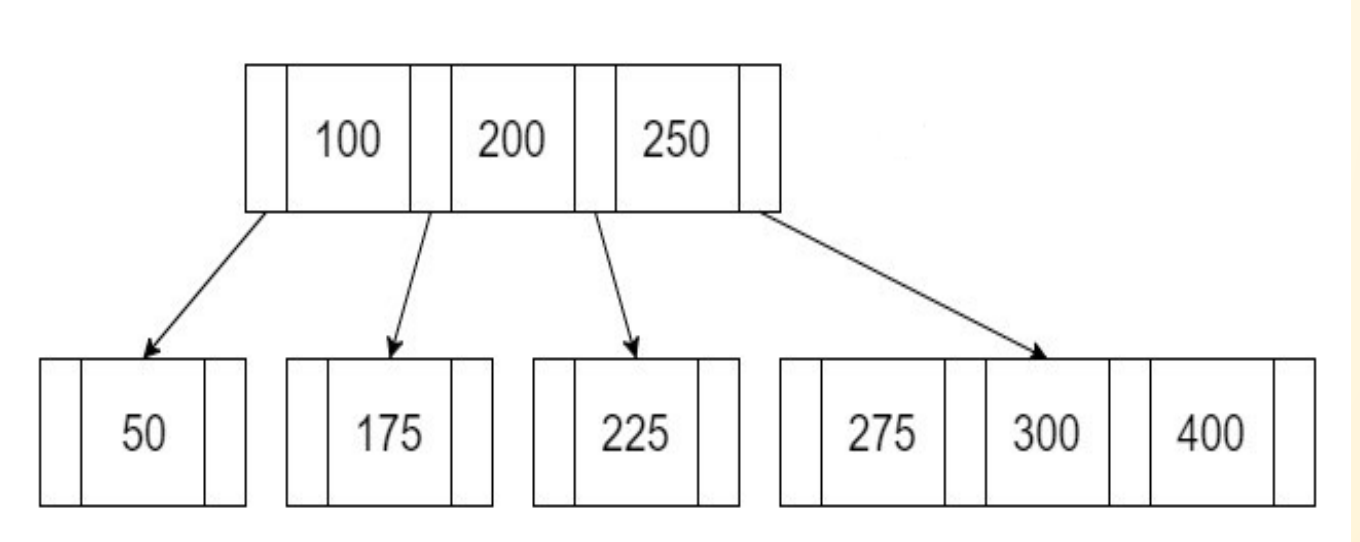
**(3) insertion**：在 **insertion** 時，若插入的數字已經存在，則只需 **count++**，不須進行其他動作。

**(4) deletion**：若要刪除的數字 **count>1**，則只需 **count--**。如果僅一個，則刪除後，須配合**rotation**來維持樹的平衡。

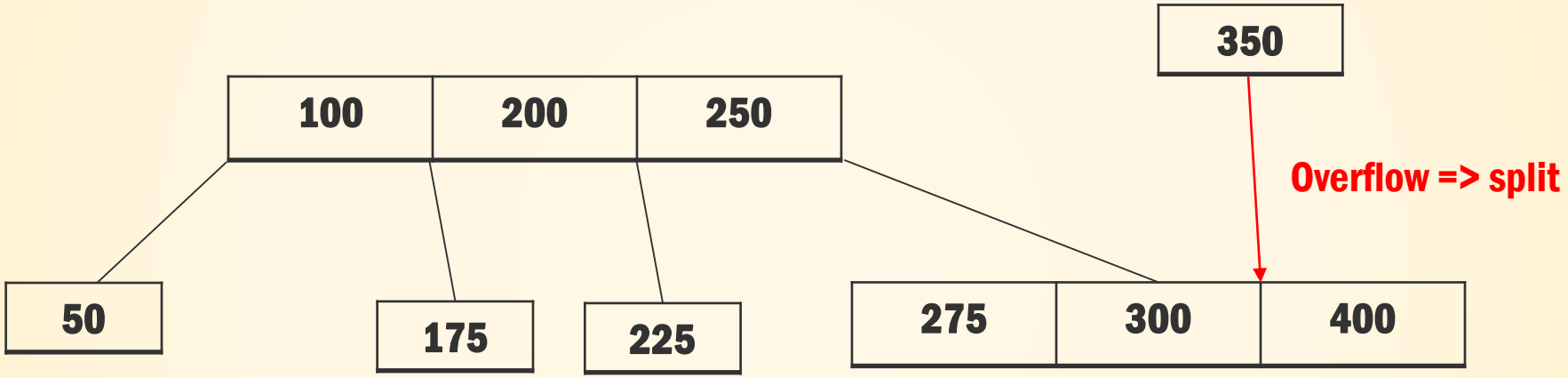
**(5)** 複製 **node**：在 **deletion** 中要刪除的節點具有兩個子節點時會用到，除了 **key value**、左右節點，也要複製 **count** 的數值。

# Q8

- The minimal degree of the following B tree is 2. Please insert a new key “350” into the B tree. Please explain the process step by step.

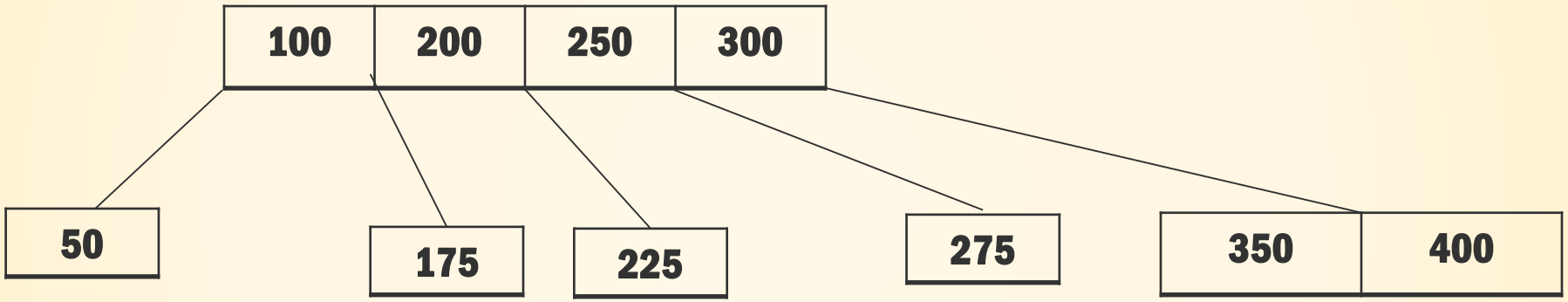


# Q8_Ans (1)

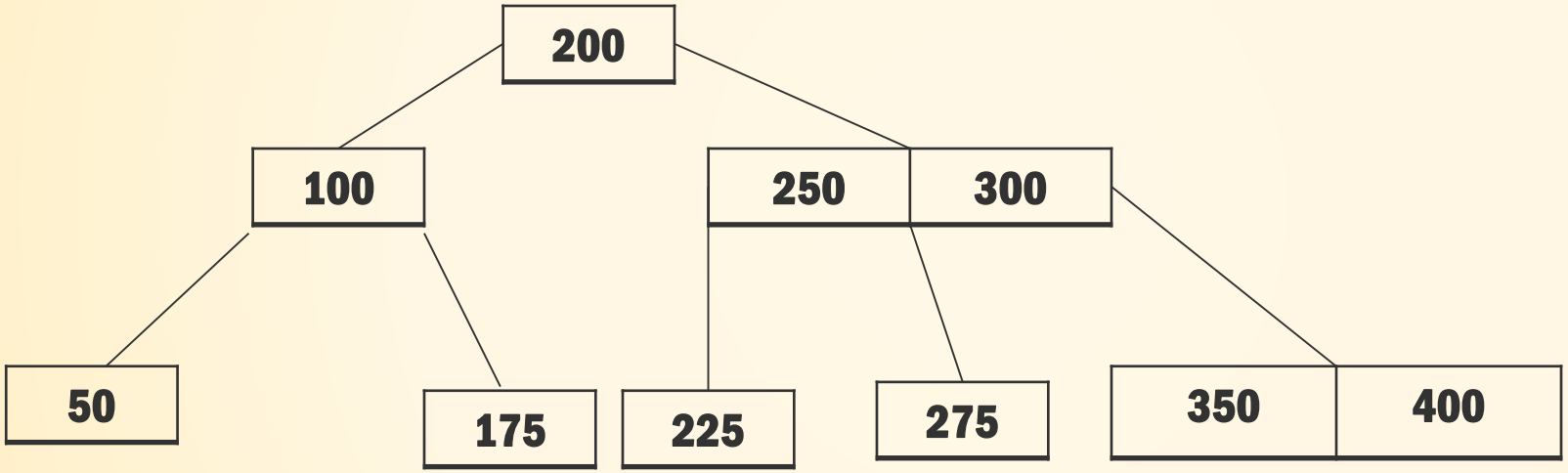


# Q8_Ans (2)

Overflow => split (200跟250選一作為parent)

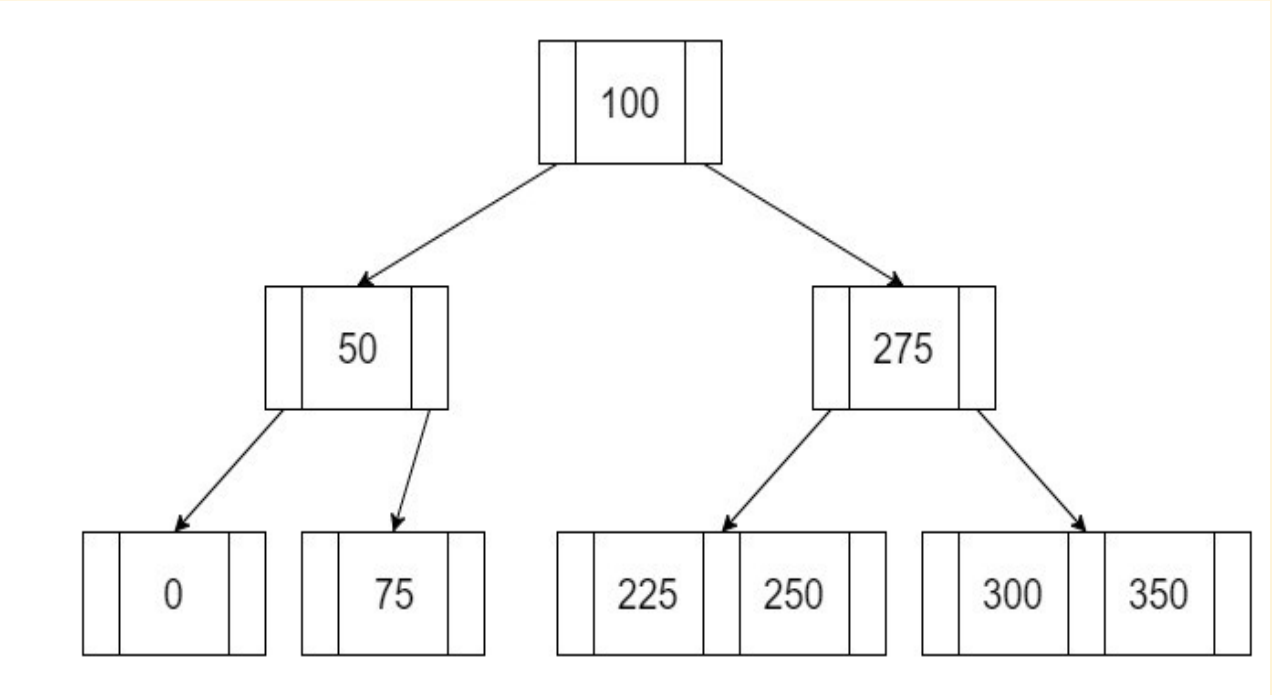


# Q8_Ans (3)

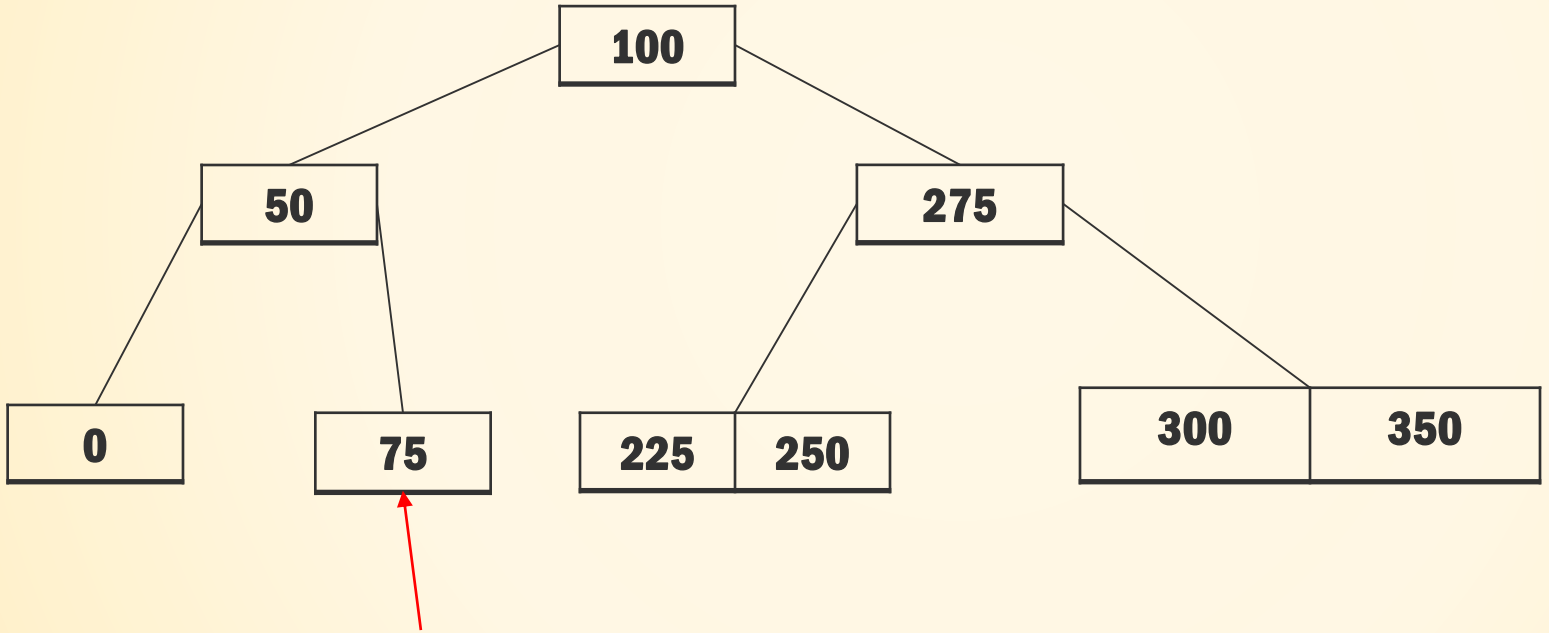


# Q9

- The minimal degree of the following B tree is 2. Please delete the key “75” in the B tree. You need to explain the process step by step.



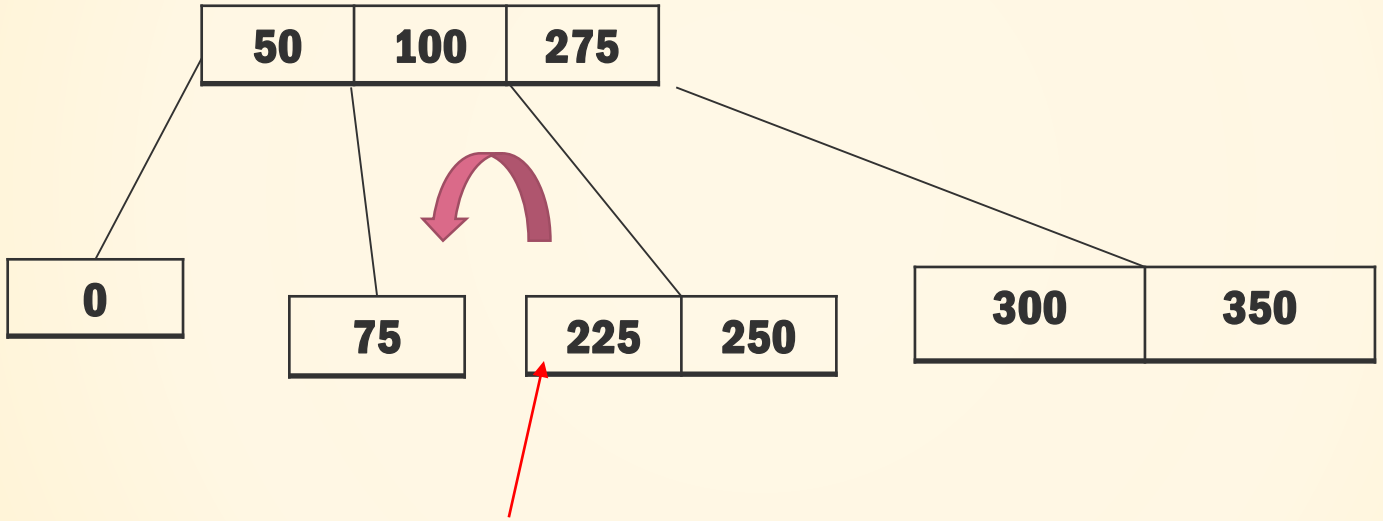
# Q9_Ans (1)




**Delete,** 但是要維持每一個**node**點都要有兩個**way**，所以無法直接移除。且在同一**parent node 50** 的其他**child node**也沒有足夠的**key value**進行 **redistribution**，所以只能往**merge**處理。

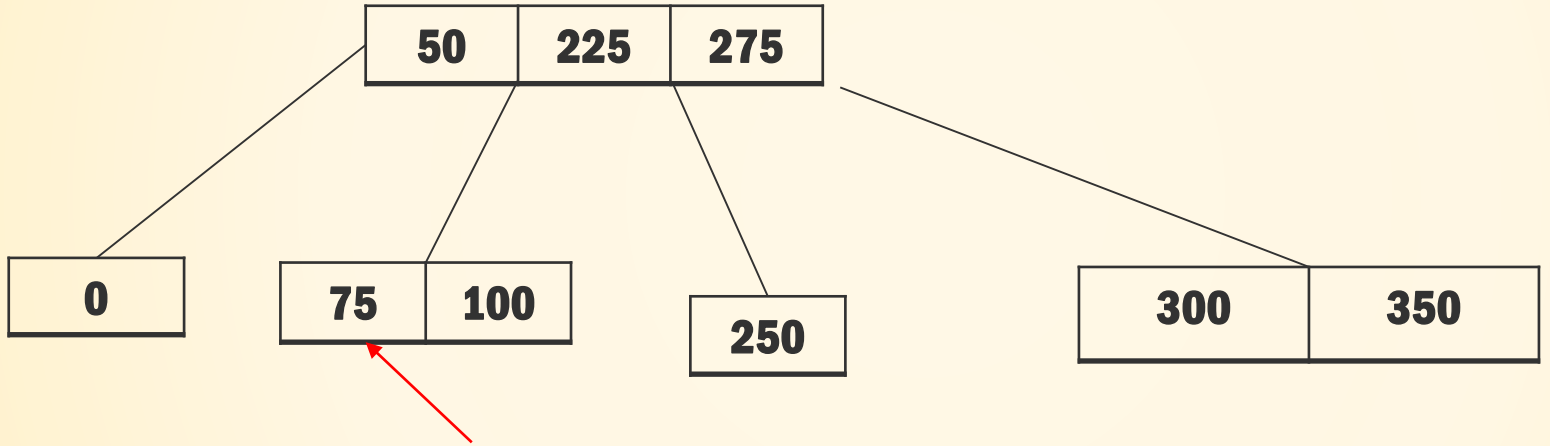
# Q9_Ans (2)

所以能進行的就是將上層 **Merge (or Combination)**起來




我們發現這邊有足夠的**key value**能夠進行**left shift rotation (redistribution)**

# Q9_Ans (3)




此時，發現這邊有足夠的維持每一個節點都有兩個**way**，就可以安心的**delete 75**

# Q9_Ans (4)