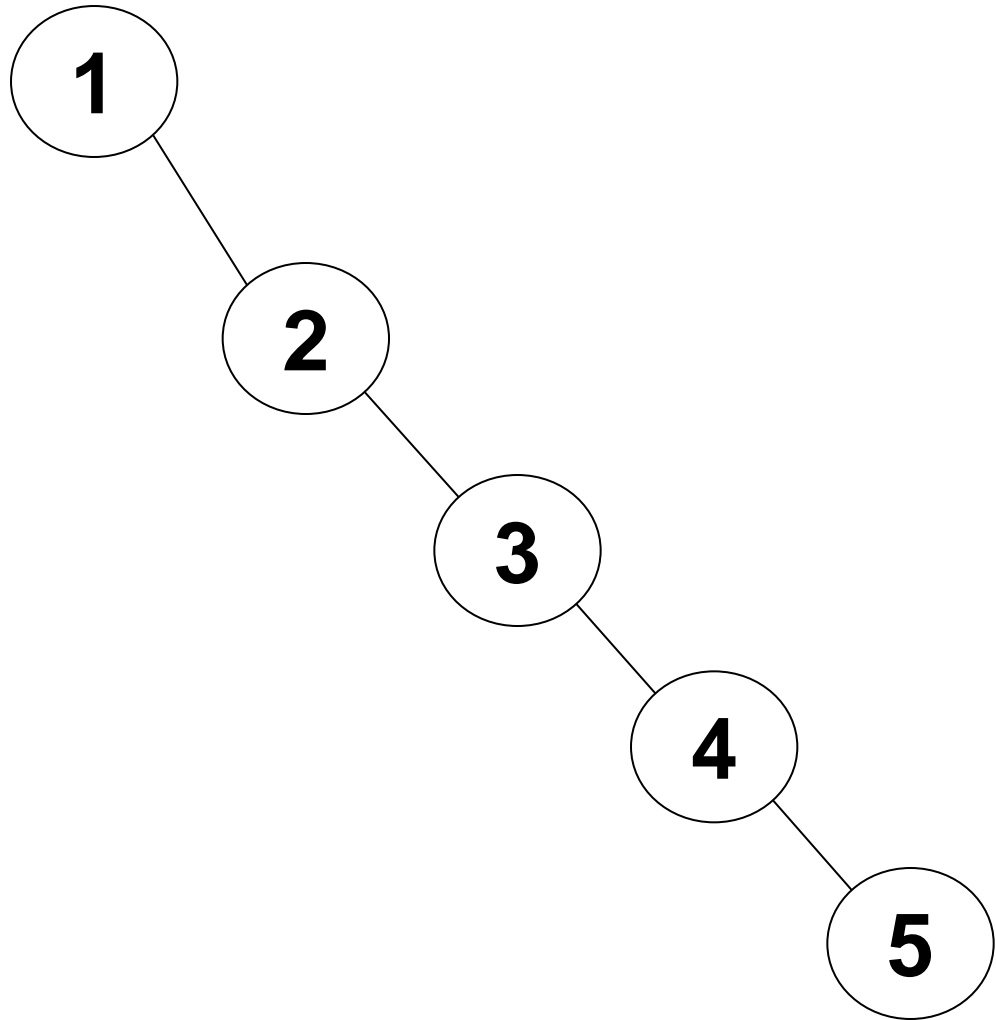


# Đánh giá tìm kiếm

1

□ 1, 2, 3, 4, 5



# Giới thiệu AVL Tree

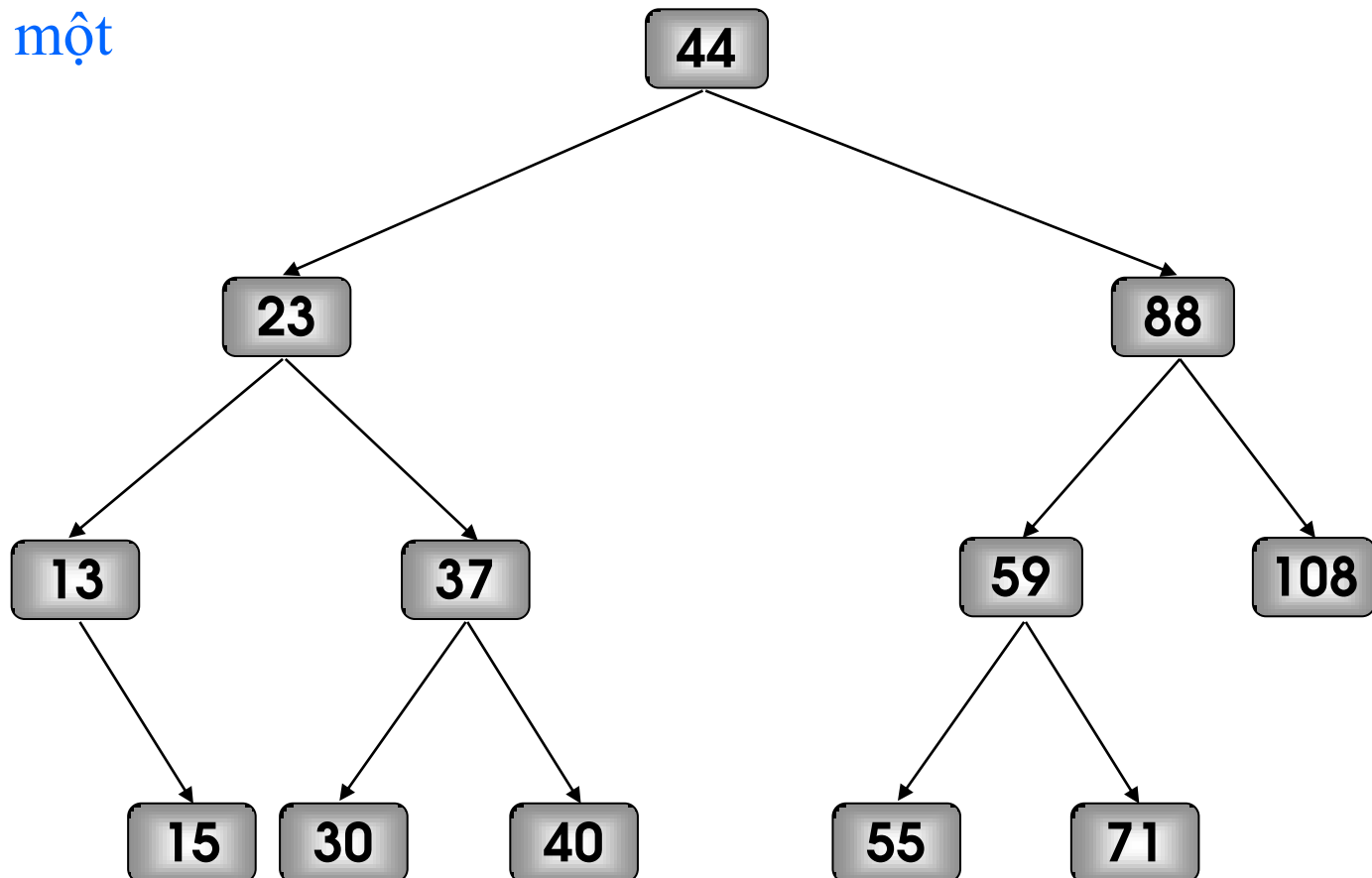
2

- Phương pháp chèn trên CNPTK có thể có những biến dạng mất cân đối nghiêm trọng
  - ▣ **Chi phí cho việc tìm kiếm** trong trường hợp xấu nhất đạt tới  **$n$**
  - ▣ VD: 1 triệu nút  $\Rightarrow$  chi phí tìm kiếm = 1.000.000 nút
- Nếu có một cây tìm kiếm nhị phân cân bằng hoàn toàn, **chi phí cho việc tìm kiếm** chỉ xấp xỉ  **$\log_2 n$** 
  - ▣ VD: 1 triệu nút  $\Rightarrow$  chi phí tìm kiếm =  $\log_2 1.000.000 \approx 20$  nút
- G.M. **A**delson-**V**elsky và E.M. **L**andis đã đề xuất một tiêu chuẩn cân bằng (sau này gọi là cân bằng **AVL**)
  - ▣ Cây AVL có chiều cao  $O(\log_2(n))$

# AVL Tree - Định nghĩa

3

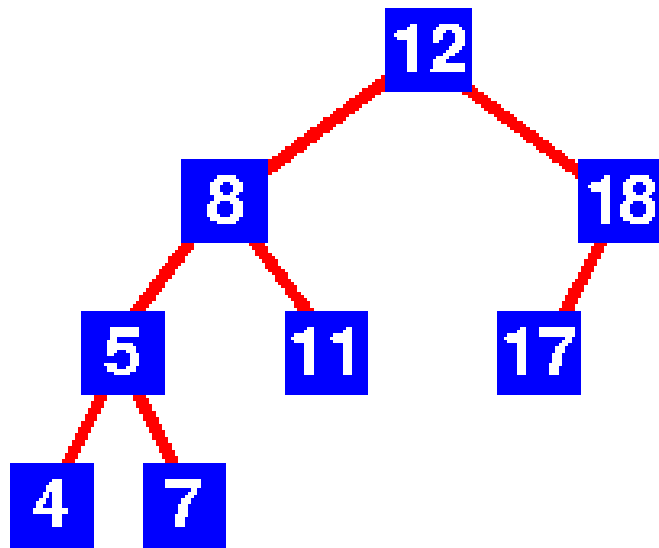
- Cây nhị phân tìm kiếm cân bằng (AVL) là cây mà tại mỗi nút **độ cao** của cây con trái và của cây con phải **chênh lệch không quá một**



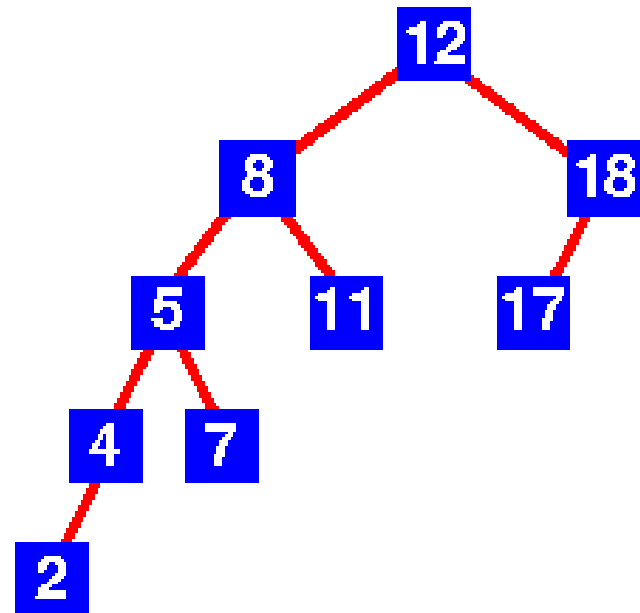
# AVL Tree – Ví dụ

4

AVL Tree ?



~~AVL Tree?~~

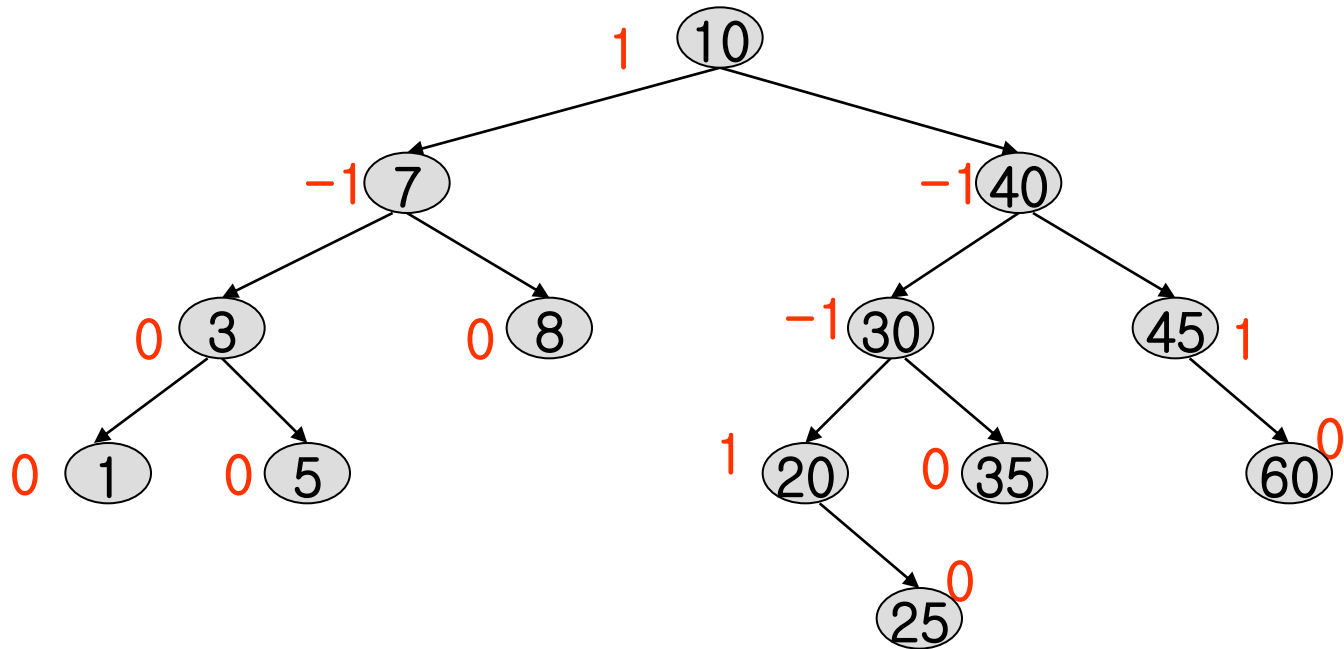


# AVL Tree

5

- Chỉ số cân bằng của một nút:
  - ▣ Định nghĩa: Chỉ số cân bằng của một nút là **hiệu** của chiều cao cây con phải và cây con trái của nó
  - ▣ Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:
    - $\text{CSCB}(p) = 0 \Leftrightarrow \text{Độ cao cây phải}(p) = \text{Độ cao cây trái}(p)$
    - $\text{CSCB}(p) = 1 \Leftrightarrow \text{Độ cao cây phải}(p) > \text{Độ cao cây trái}(p)$
    - $\text{CSCB}(p) = -1 \Leftrightarrow \text{Độ cao cây phải}(p) < \text{Độ cao cây trái}(p)$

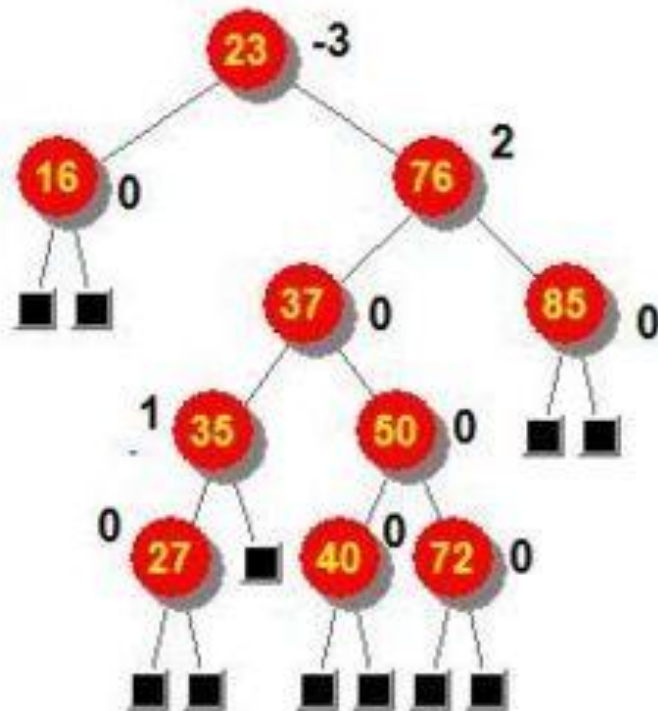
# Ví dụ - Chỉ số cân bằng của nút



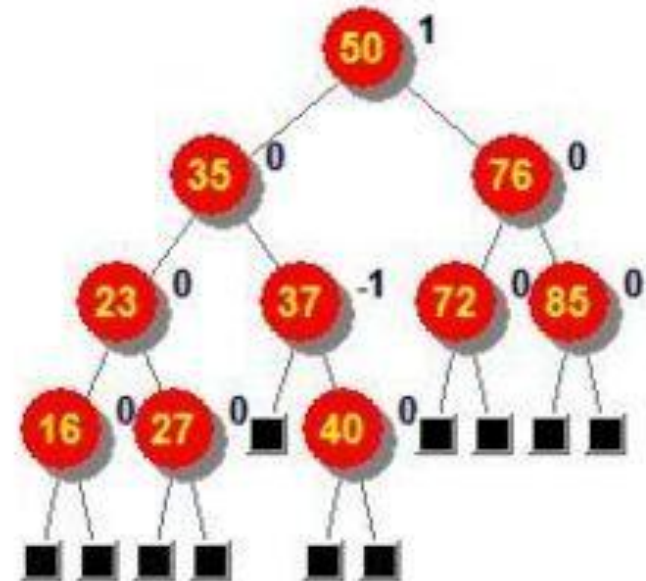
- What is the balance factor for each node in this AVL tree?
- Is this an AVL tree?

# AVL Tree – Ví dụ

7



1. Cây tìm kiếm nhị phân  
không là cây AVL



2. Cây AVL

Hai cây được tạo từ cùng dãy khóa

23;76;37;85;50;40;72;35;16;27;

# AVL Tree – Biểu diễn

8

```
#define RH      1      /* Cây con phải cao hơn */
#define EH      0      /* Hai cây con bằng nhau */
#define LH     -1      /* Cây con trái cao hơn */

struct AVLNode{
    char          balFactor;      // Chỉ số cân bằng
    DataType      data;
    AVLNode*      pLeft;
    AVLNode*      pRight;
};

typedef AVLNode*  AVLTree;
```



# AVL Tree – Biểu diễn

9

- Các thao tác đặc trưng của cây AVL:
  - ▣ **Thêm** một phần tử vào cây AVL
  - ▣ **Hủy** một phần tử trên cây AVL
  - ▣ **Cân bằng lại** một cây vừa bị mất cân bằng (**Rotation**)
- Trường hợp **thêm** một phần tử trên cây AVL được thực hiện giống như thêm trên CNPTK, tuy nhiên sau khi thêm phải cân bằng lại cây
- Trường hợp **hủy** một phần tử trên cây AVL được thực hiện giống như hủy trên CNPTK và cũng phải cân bằng lại cây
- Việc **cân bằng lại** một cây sẽ phải thực hiện sao cho chỉ ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng

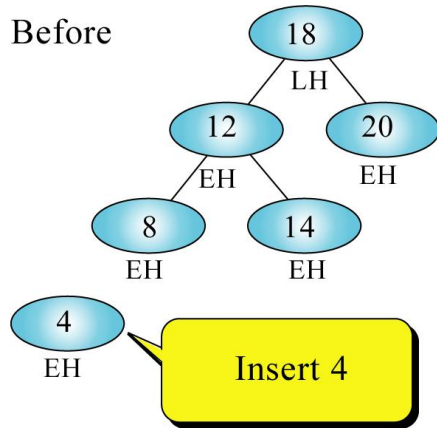
# AVL Tree - Các trường hợp mất cân bằng

10

- Không khảo sát tính cân bằng của 1 cây nhị phân bất kỳ mà chỉ quan tâm đến khả năng mất cân bằng xảy ra khi **chèn** hoặc **xóa** một nút trên cây AVL
- Các trường hợp mất cân bằng:
  - ▣ Sau khi chèn (xóa) cây con **trái lệch trái** (left of left)
  - ▣ Sau khi chèn (xóa) cây con **trái lệch phải** (right of left)
  - ▣ Sau khi chèn (xóa) cây con **phải lệch phải** (right of right)
  - ▣ Sau khi chèn (xóa) cây con **phải lệch trái** (left of right)

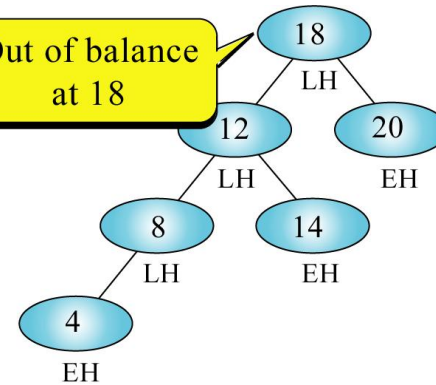
# Ví dụ: Các trường hợp mất cân bằng

Before



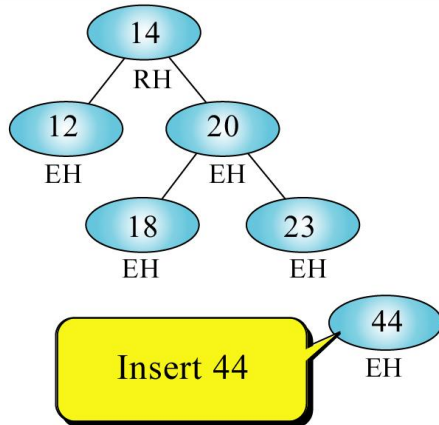
Out of balance  
at 18

After



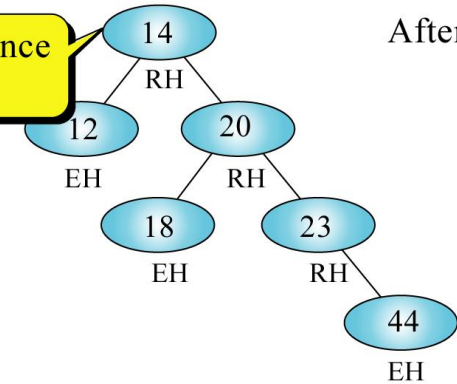
(a) Case 1: left of left

Before



Out of balance  
at 14

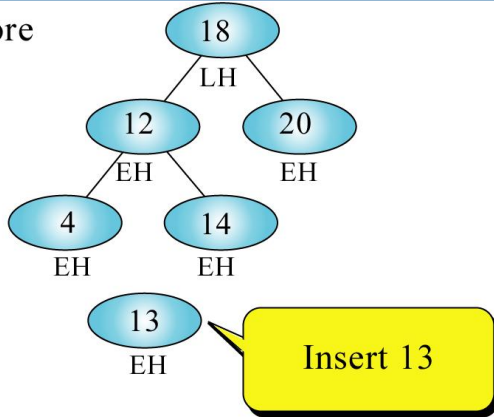
After



(b) Case 2: right of right

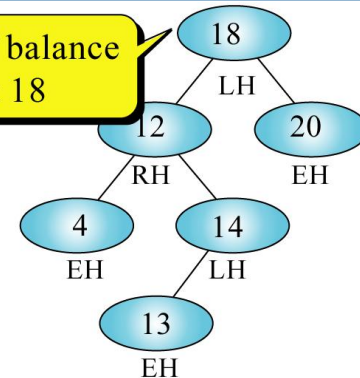
# Ví dụ: Các trường hợp mất cân bằng

Before



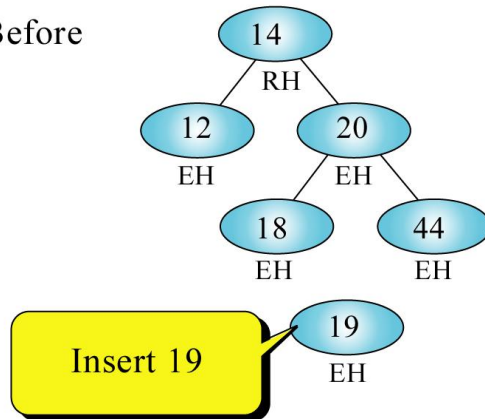
Out of balance  
at 18

After



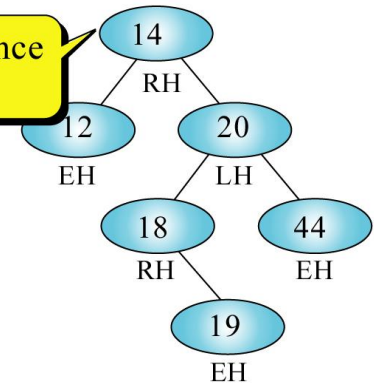
(c) Case 3: right of left

Before



Out of balance  
at 14

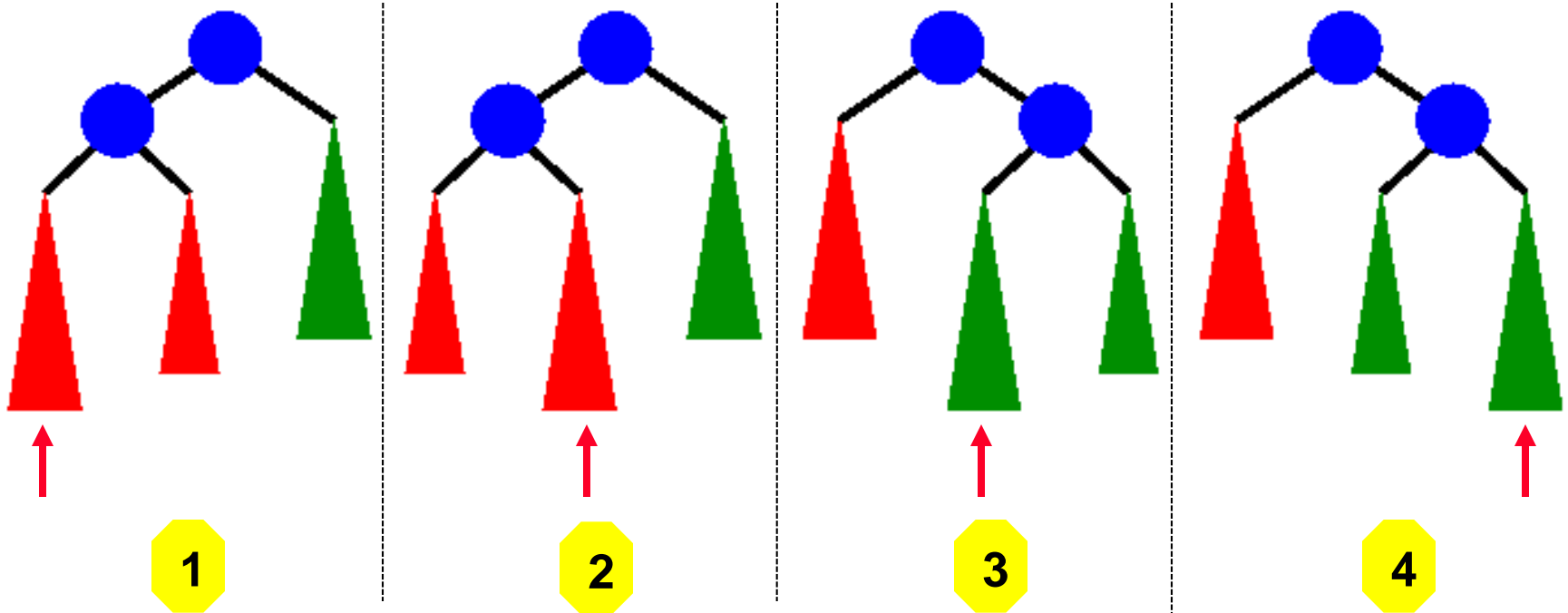
After



(d) Case 4: left of right

# AVL Tree - Các trường hợp mất cân bằng

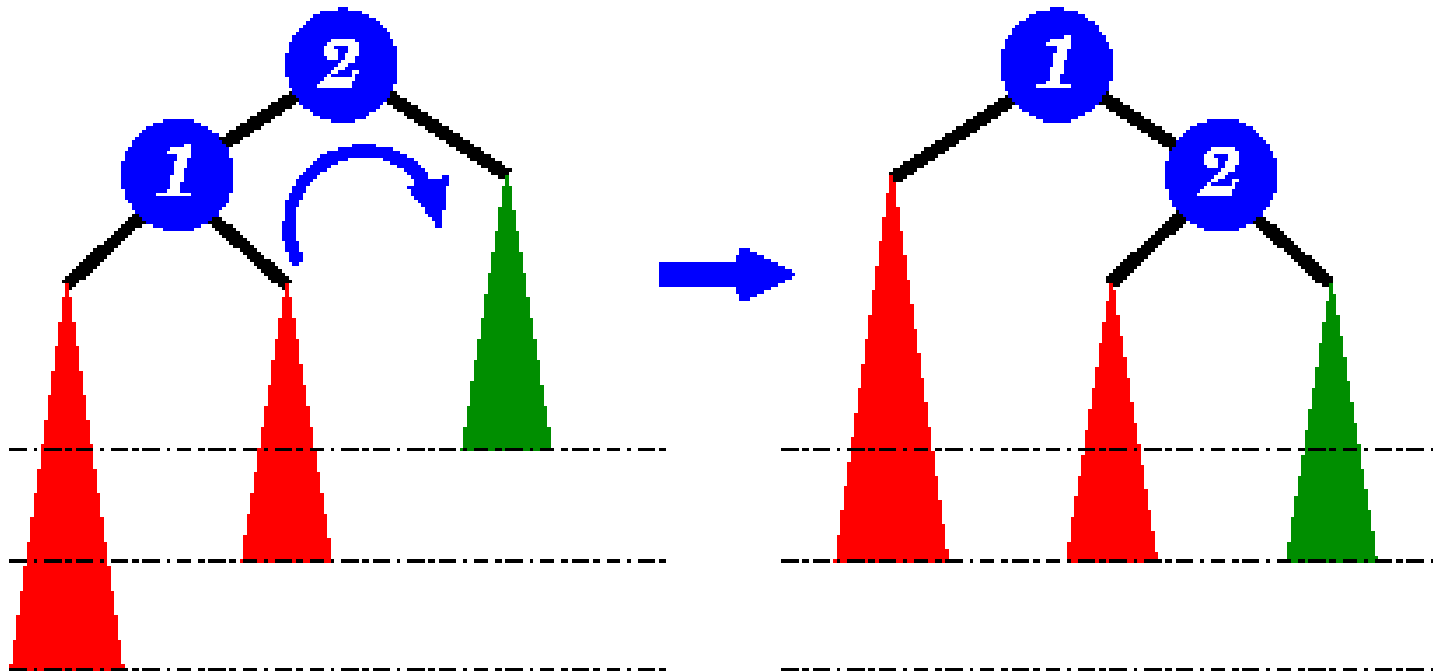
## □ Chèn nút vào cây AVL



- 1 và 4 là các ảnh đối xứng
- 2 và 3 là các ảnh đối xứng

# Cây AVL – Tái cân bằng

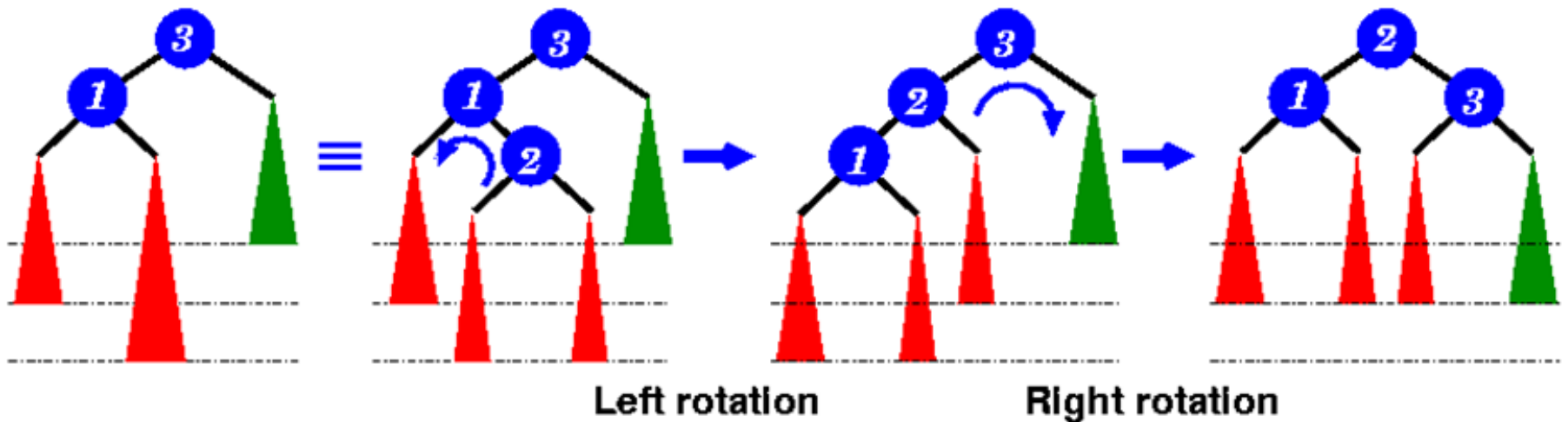
- Trường hợp 1 được giải bởi phép quay:



- Trường hợp 4 là quay một ảnh đối xứng

# Cây AVL – Tái cân bằng

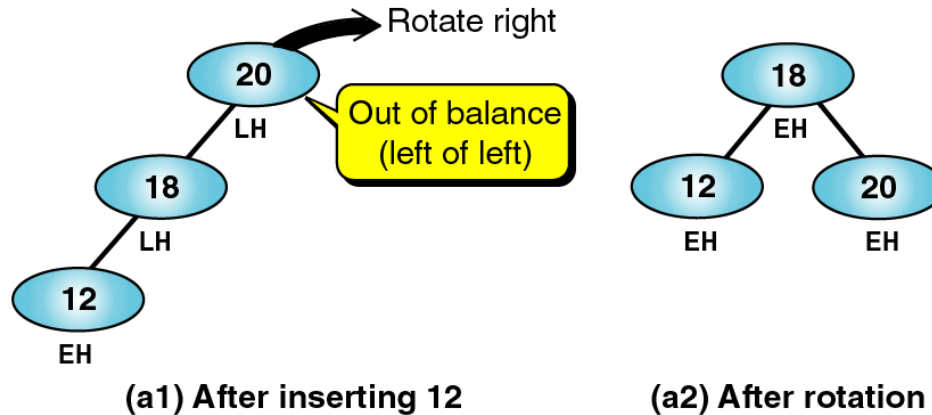
- Trường hợp 2 cần một phép quay kép (*double*)



- Trường hợp 3 là phép quay ảnh đối xứng

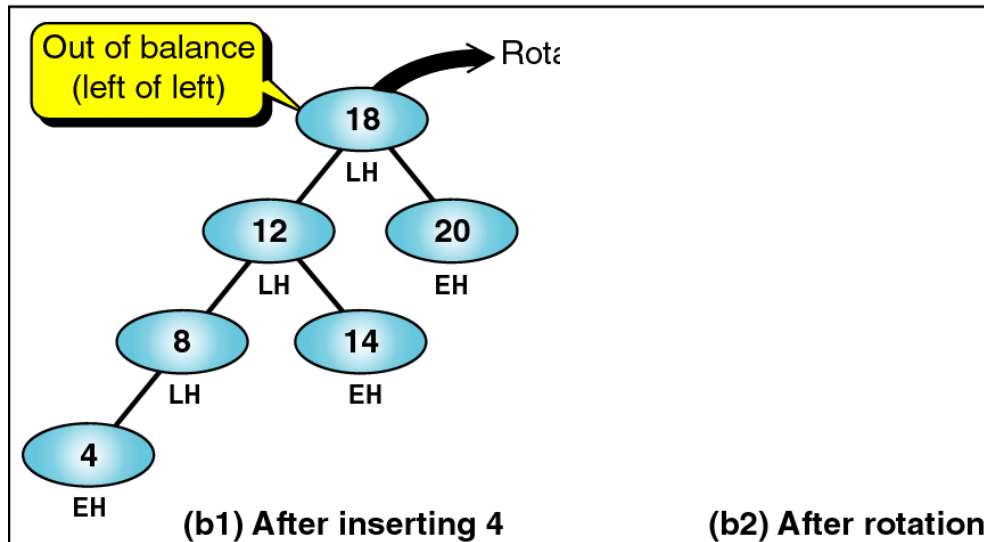
# Ví dụ: Tái cân bằng

16



(a) Simple right rotation

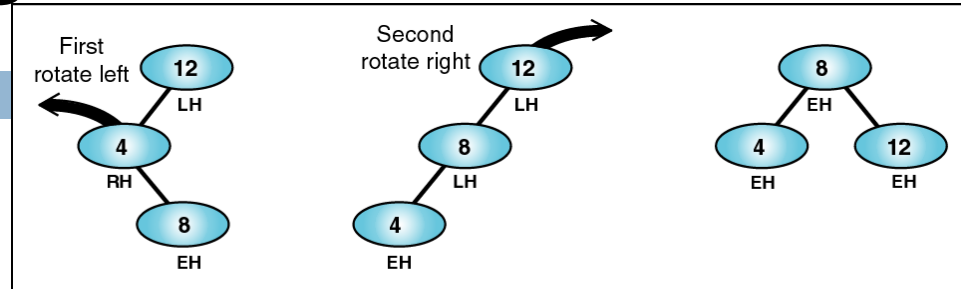
1. left of left



(b) Complex right rotation

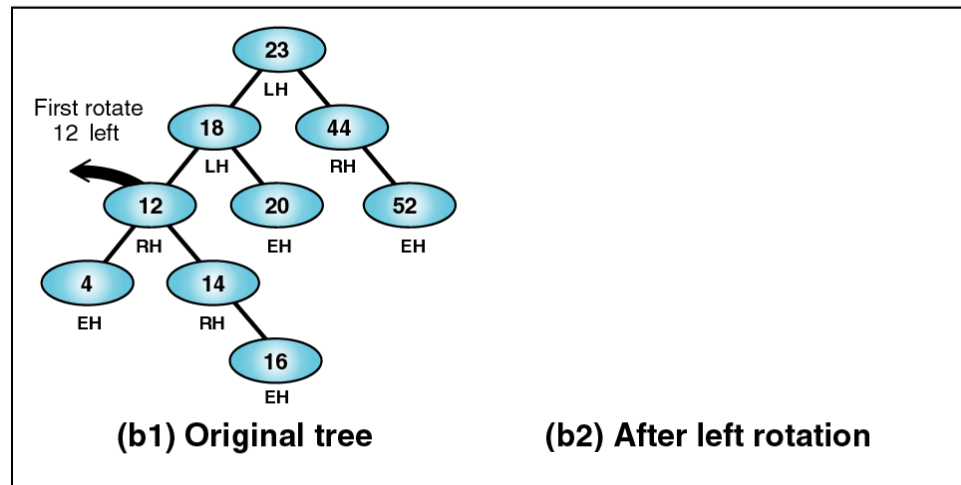


# Ví dụ: Tái cân bằng



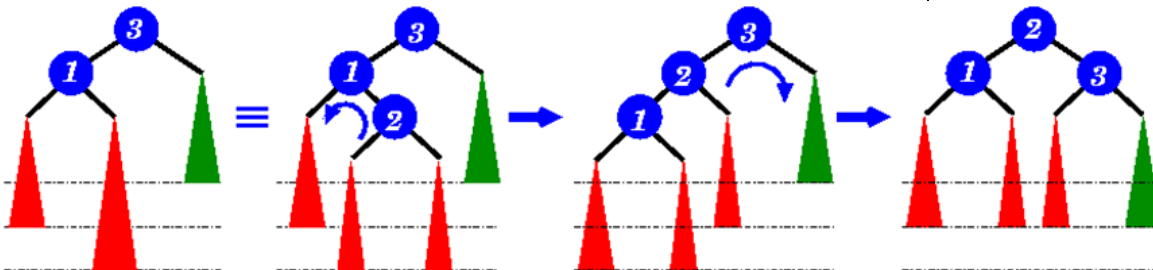
(a) Simple double rotation right

2. right of left



(b1) Original tree

(b2) After left rotation



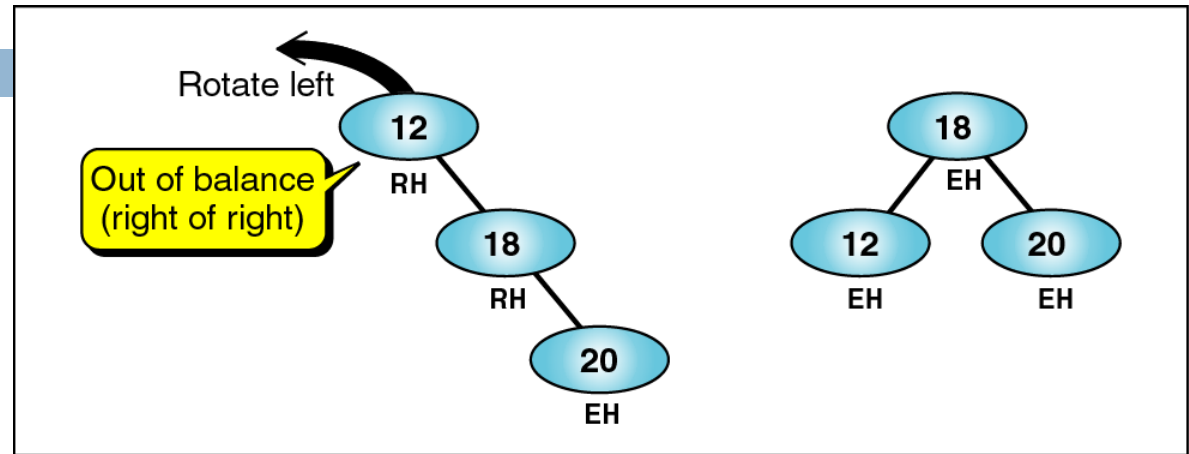
Left rotation

Right rotation

(b3) After right rotation

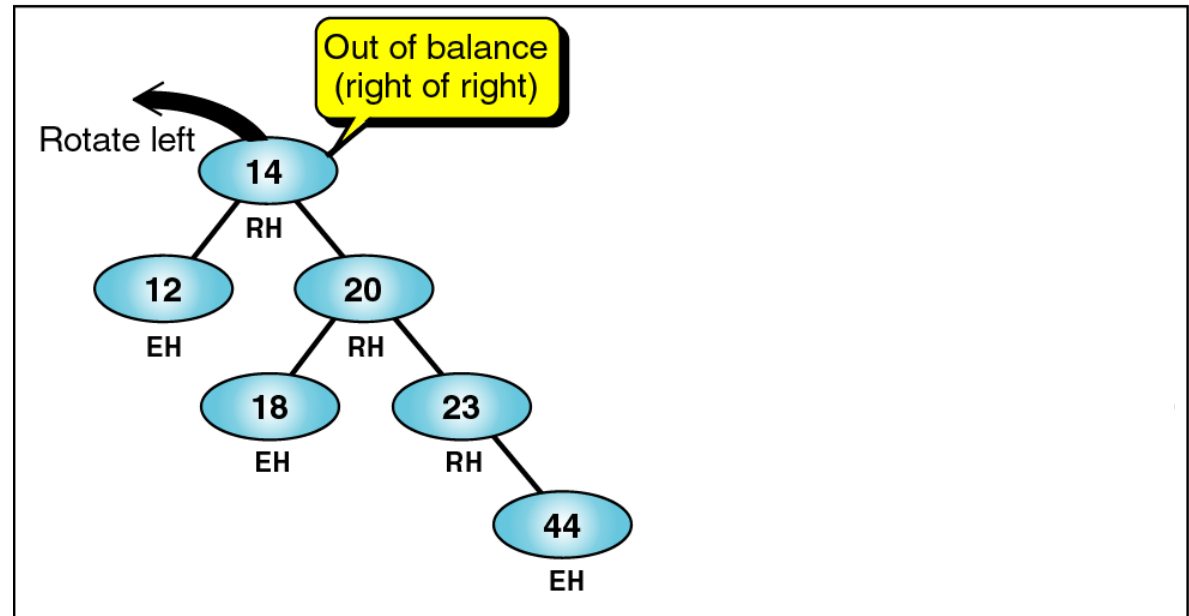
(b) Complex double rotation right

# Ví dụ: Tái cân bằng



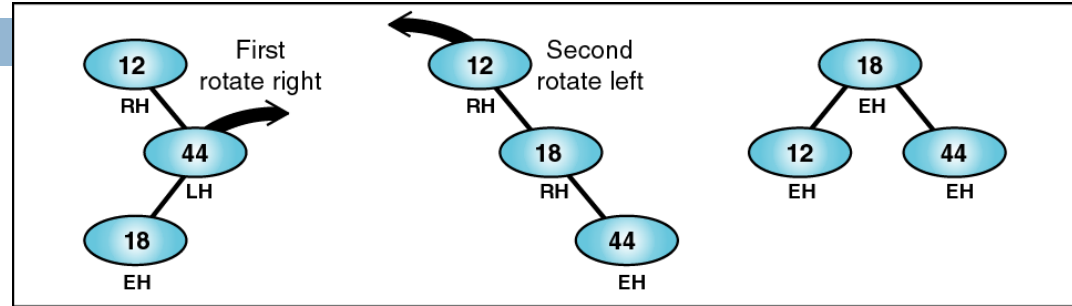
(a) Simple left rotation

## 3. right of right



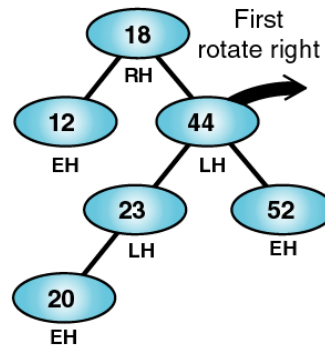
(b) Complex left rotation

# Ví dụ: Tái cân bằng



(a) Simple double rotation right

4. left of right



(b1) Original tree

(b2) After right rotation

(b3) After left rotation

(b) Complex double rotation right

# AVL Tree - Cân bằng lại cây AVL

32

## □ Quay đơn Left-Left:

```
void rotateLL(AVLTree &T) //quay đơn Left-Left
{
    AVLNode* T1 = T->pLeft;
    T->pLeft = T1->pRight;
    T1->pRight = T;
    switch(T1->balFactor) {
        case LH: T->balFactor = EH;
                  T1->balFactor = EH;
                  break;
        case EH: T->balFactor = LH;
                  T1->balFactor = RH;
                  break;
    }
    T = T1;
}
```

# AVL Tree - Cân bằng lại cây AVL

33

## □ Quay đơn Right-Right:

```
void rotateRR (AVLTree &T) //quay đơn Right-Right
{
    AVLNode* T1 = T->pRight;
    T->pRight = T1->pLeft;
    T1->pLeft = T;
    switch(T1->balFactor) {
        case RH: T->balFactor = EH;
                  T1->balFactor= EH;
                  break;
        case EH: T->balFactor = RH;
                  T1->balFactor= LH;
                  break;
    }
    T = T1;
}
```

# AVL Tree - Cân bằng lại cây AVL

34

## □ Quay kép Left-Right:

```
void rotateLR(AVLTree &T)//quay kép Left-Right
{
    AVLNode* T1 = T->pLeft;
    AVLNode* T2 = T1->pRight;
    T->pLeft = T2->pRight;
    T2->pRight = T;
    T1->pRight = T2->pLeft;
    T2->pLeft = T1;
    switch(T2->balFactor) {
        case LH: T->balFactor = RH; T1->balFactor = EH; break;
        case EH: T->balFactor = EH; T1->balFactor = EH; break;
        case RH: T->balFactor = EH; T1->balFactor = LH; break;
    }
    T2->balFactor = EH;
    T = T2;
}
```

# AVL Tree - Cân bằng lại cây AVL

35

## □ Quay kép Right-Left

```
void rotateRL(AVLTree &T)    //quay kép Right-Left
{
    AVLNode* T1 = T->pRight;
    AVLNode* T2 = T1->pLeft;
    T->pRight = T2->pLeft;
    T2->pLeft = T;
    T1->pLeft = T2->pRight;
    T2->pRight = T1;
    switch(T2->balFactor) {
        case RH: T->balFactor = LH; T1->balFactor = EH; break;
        case EH: T->balFactor = EH; T1->balFactor = EH; break;
        case LH: T->balFactor = EH; T1->balFactor = RH; break;
    }
    T2->balFactor = EH;
    T = T2;
}
```

# AVL Tree - Cân bằng lại cây AVL

36

- Cân bằng khi cây bị lệch về bên trái:

```
int balanceLeft(AVLTree &T)
//Cân bằng khi cây bị lệch về bên trái
{
    AVLNode* T1 = T->pLeft;

    switch(T1->balFactor)
    {
        case LH:    rotateLL(T); return 2;
        case EH:    rotateLL(T); return 1;
        case RH:    rotateLR(T); return 2;
    }
    return 0;
}
```



# AVL Tree - Cân bằng lại cây AVL

37

- Cân bằng khi cây bị lệch về bên phải

```
int balanceRight(AVLTree &T)
//Cân bằng khi cây bị lệch về bên phải
{
    AVLNode* T1 = T->pRight;

    switch(T1->balFactor)
    {
        case LH:    rotateRL(T); return 2;
        case EH:    rotateRR(T); return 1;
        case RH:    rotateRR(T); return 2;
    }
    return 0;
}
```

# AVL Tree - Thêm một phần tử trên cây AVL

38

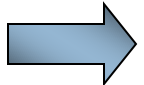
- Việc thêm một phần tử vào cây AVL diễn ra tương tự như trên CNPTK
- Sau khi thêm xong, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này
- Việc cân bằng lại chỉ cần thực hiện 1 lần tại nơi mất cân bằng
- Hàm *insertNode* trả về giá trị  $-1$ ,  $0$ ,  $1$  khi không đủ bộ nhớ, gặp nút cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng, giá trị  $2$  sẽ được trả về

**int insertNode(AVLTree &T, DataType X)**

# AVL Tree - Thêm một phần tử trên cây AVL

39

```
int insertNode(AVLTree &T, DataType X)
{  int res;
   if (T)
   {   if (T->key == X) return 0; //đã có
       if (T->key > X)
       {   res      = insertNode(T->pLeft, X);
           if(res < 2) return res;
           switch(T->balFactor)
           {   case RH: T->balFactor = EH;  return 1;
               case EH: T->balFactor = LH;  return 2;
               case LH: balanceLeft(T);    return 1;
           }
       }
   }
   .....
}
```

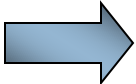


insertNode2

# AVL Tree - Thêm một phần tử trên cây AVL

40

```
int insertNode(AVLTree &T, DataType X)
{
    .....
    else // T->key < X
    {
        res = insertNode(T-> pRight, X);
        if(res < 2) return res;
        switch(T->balFactor)
        {
            case LH: T->balFactor = EH; return 1;
            case EH: T->balFactor = RH; return 2;
            case RH: balanceRight(T); return 1;
        }
    }
    .....
}
```



insertNode3

# AVL Tree - Thêm một phần tử trên cây AVL

41

```
int insertNode (AVLTree &T, DataType X)
{
    .....
    .....
    T = new TNode;
    if (T == NULL) return -1; //thiếu bộ nhớ
    T->key = X;
    T->balFactor = EH;
    T->pLeft = T->pRight = NULL;
    return 2; // thành công, chiều cao tăng
}
```

# AVL Tree - Hủy một phần tử trên cây AVL

42

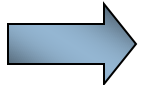
- Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên CNPTK
- Sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại
- Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền
- Hàm *delNode* trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi hủy, chiều cao cây bị giảm, giá trị 2 sẽ được trả về:

**int *delNode*(AVLTree &T, DataType X)**

# AVL Tree - Hủy một phần tử trên cây AVL

43

```
int delNode(AVLTree &T, DataType X)
{ int res;
  if (T==NULL)      return 0;
  if (T->key > X)
  { res = delNode (T->pLeft, X);
    if (res < 2)    return res;
    switch (T->balFactor)
    { case LH: T->balFactor = EH; return 2;
      case EH: T->balFactor = RH; return 1;
      case RH: return balanceRight(T);
    }
  } // if (T->key > X)
  .....
}
```

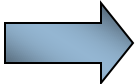


delNode2

# AVL Tree - Hủy một phần tử trên cây AVL

44

```
int delNode(AVLTree &T, DataType X)
{
.....
    if(T->key < X)
    {
        res      = delNode (T->pRight, X);
        if(res < 2)    return res;
        switch(T->balFactor)
        {
            case RH: T->balFactor = EH; return 2;
            case EH: T->balFactor = LH; return 1;
            case LH: return balanceLeft(T);
        }
    } // if(T->key < X)
.....
}
```



delNode3



# AVL Tree - Hủy một phần tử trên cây AVL

45

```
int delNode(AVLTree &T, DataType X)
{.....
    else //T->key == X
    {   AVLNode*    p = T;
        if(T->pLeft == NULL)           {   T = T->pRight; res = 2; }
        else if(T->pRight == NULL) {   T = T->pLeft;   res = 2; }
        else //T có đủ cả 2 con
        {   res = searchStandFor(p,T->pRight);
            if(res < 2) return res;
            switch(T->balFactor)
            {   case RH: T->balFactor = EH; return 2;
                case EH: T->balFactor = LH; return 1;
                case LH: return balanceLeft(T);
            }
        }
        delete p; return res;
    }
}
```

# AVL Tree - Hủy một phần tử trên cây AVL

46

```
int searchStandFor(AVLTree &p, AVLTree &q)
//Tìm phần tử thể mạng
{ int res;
  if(q->pLeft)
  { res = searchStandFor(p, q->pLeft);
    if(res < 2) return res;
    switch(q->balFactor)
    { case LH: q->balFactor = EH; return 2;
      case EH: q->balFactor = RH; return 1;
      case RH: return balanceRight(T);
    }
  } else
  { p->key = q->key; p = q; q = q->pRight; return 2;
  }
}
```