

# CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG (AVL TREE)

---

DATA STRUCTURES AND ALGORITHMS

TS. Nguyễn Đình Hiền  
hiennd@uit.edu.vn

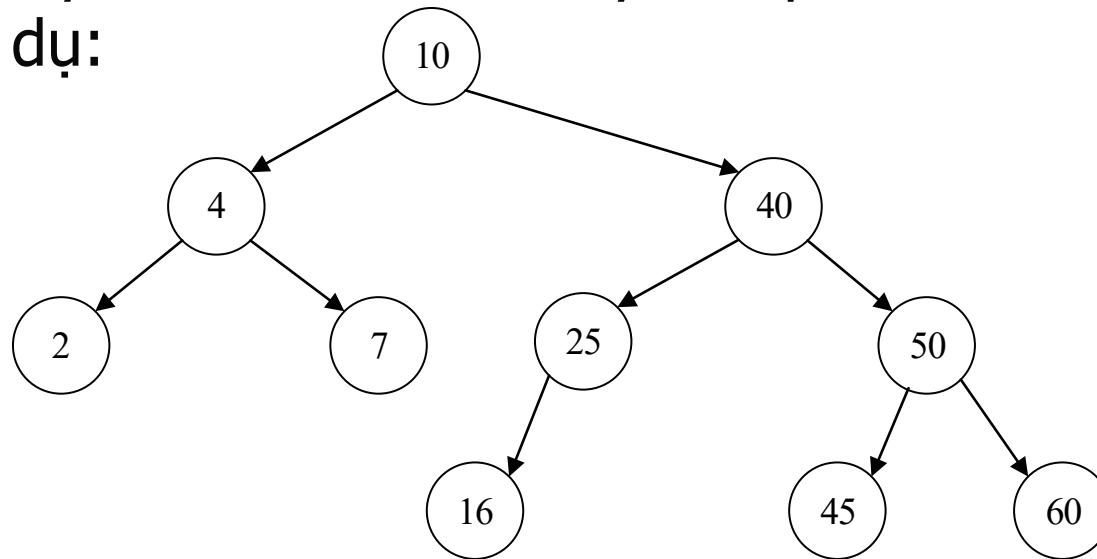
# Cây nhị phân tìm kiếm cân bằng



- Định nghĩa
- Tổ chức dữ liệu
- Các trường hợp mất cân bằng
- Các thao tác trên cây cân bằng
- Ứng dụng
- Bài tập

# Định nghĩa

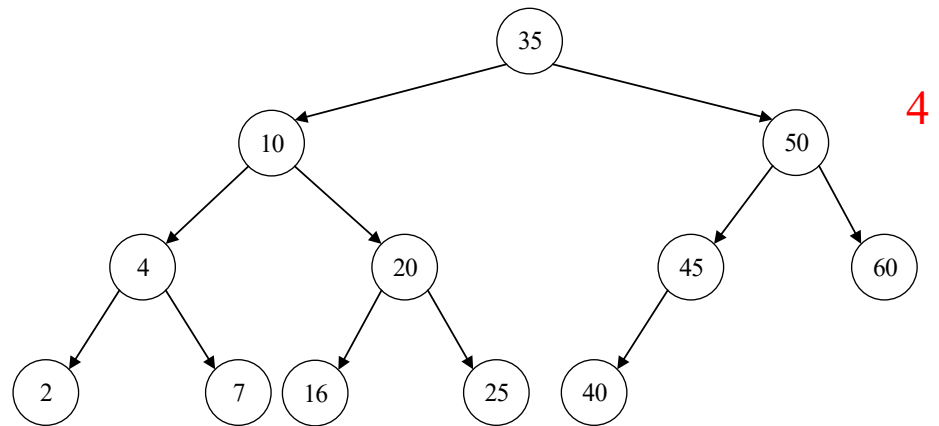
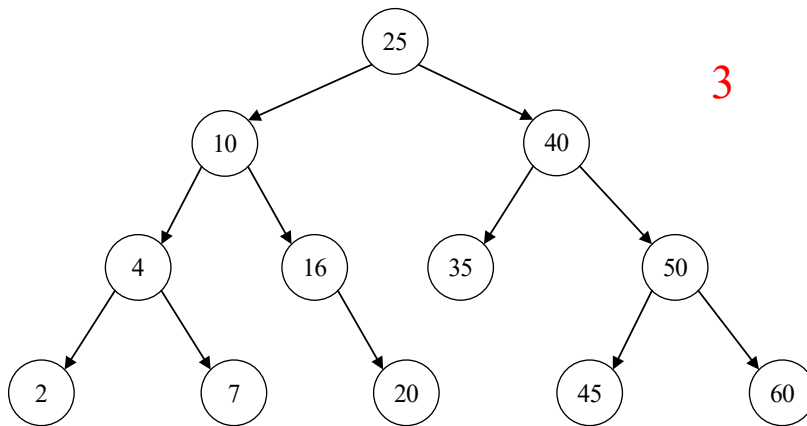
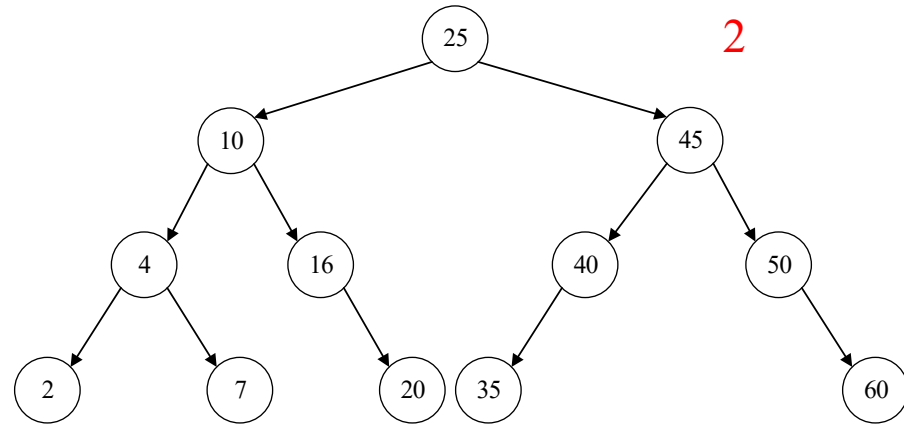
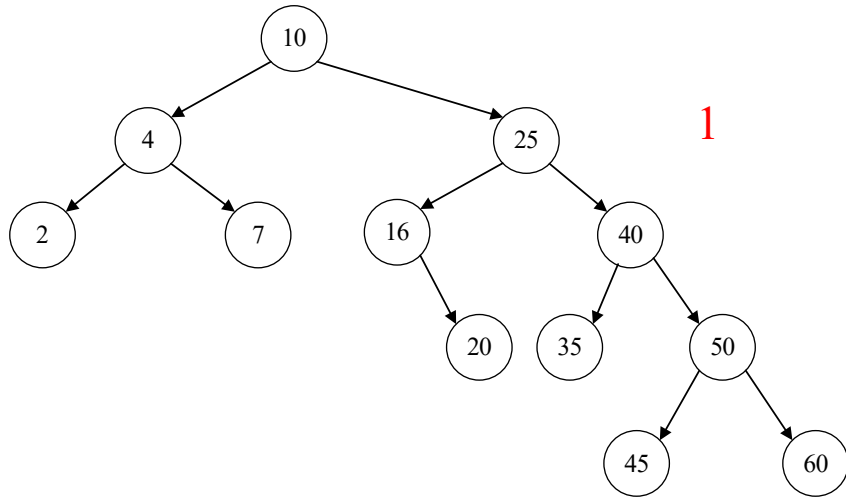
- In computer science, an AVL tree is a **self-balancing binary search tree**.
- Cây nhị phân tìm kiếm cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một. Ví dụ:



(The AVL tree is named after its two Soviet inventors, Georgy **Adelson-Velsky** and Evgenii **Landis**, who published it in their 1962 paper "An algorithm for the organization of information")

# Câu hỏi:

- Hình nào sau đây là cây AVL?



➤ Chỉ số cân bằng = độ lệch giữa cây trái và cây phải của một nút

➤ Các giá trị hợp lệ :

- $CSCB(p) = 0 \Leftrightarrow$  Độ cao cây trái (p) = Độ cao cây phải (p)
- $CSCB(p) = 1 \Leftrightarrow$  Độ cao cây trái (p) < Độ cao cây phải (p)
- $CSCB(p) = -1 \Leftrightarrow$  Độ cao cây trái (p) > Độ cao cây phải (p)

# Tổ chức dữ liệu(tt)

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

```
struct AVLNODE {
    DATA key;
    char balFactor; //chỉ số cân bằng
    AVLNODE* pLeft;
    AVLNODE* pRight;
};

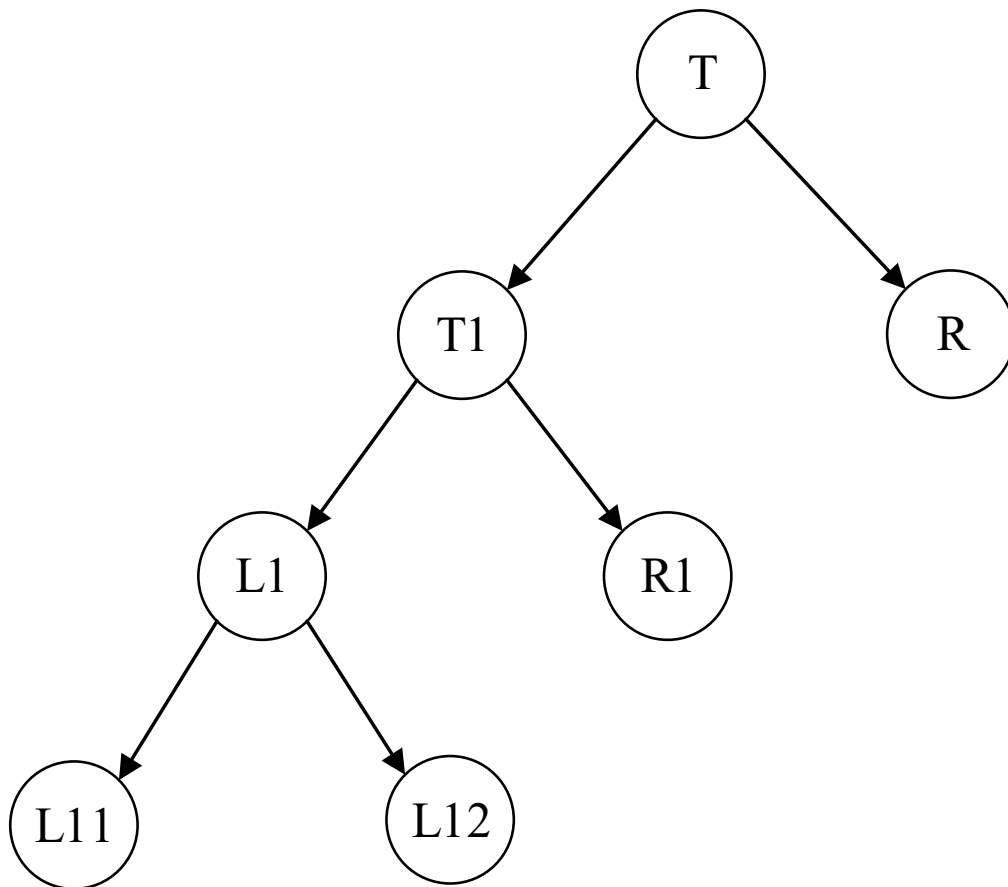
typedef AVLNODE* AVLTREE;
```

# Các trường hợp mất cân bằng do lệch trái

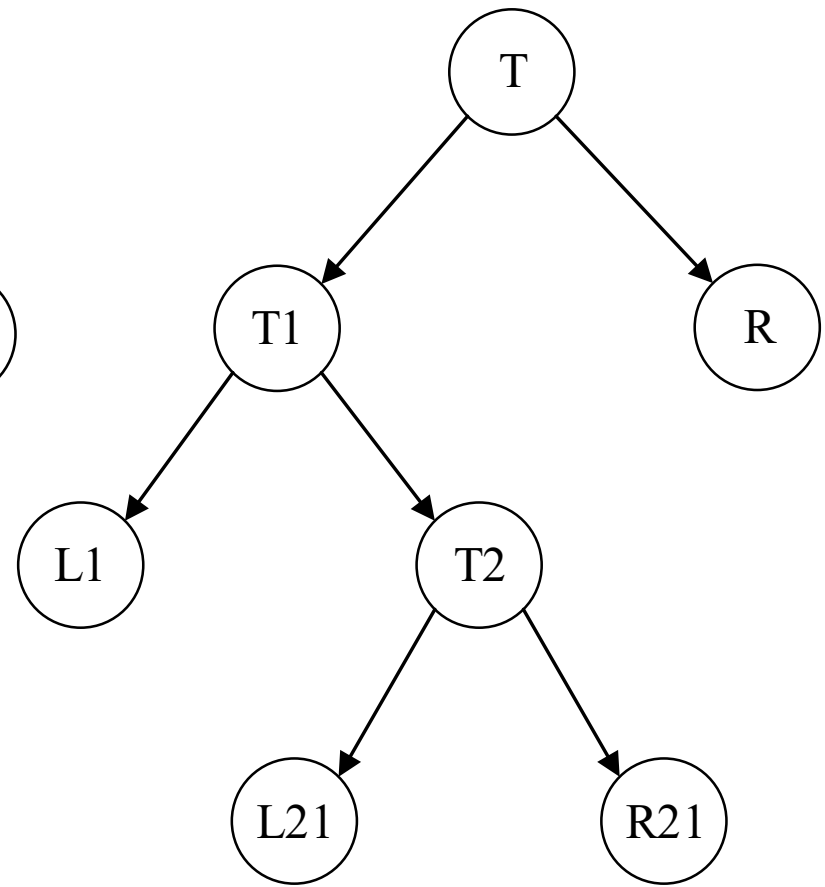


## ➤ Cây mất cân bằng tại nút T

TH1: Left-Left



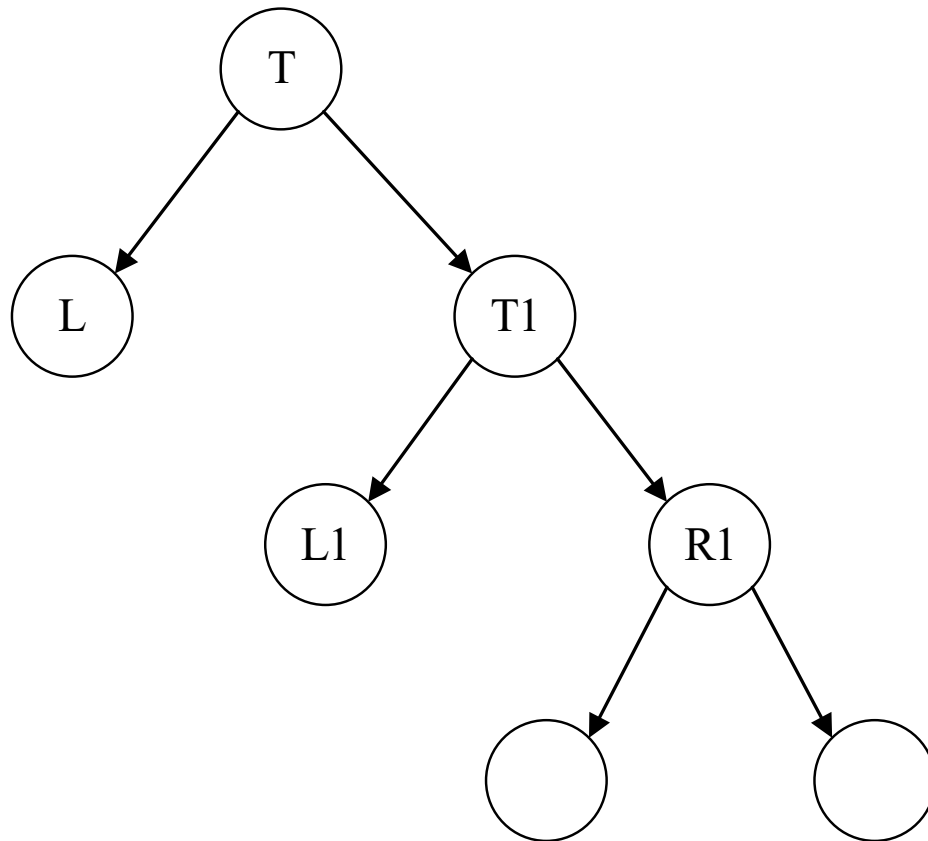
TH2: Left-Right



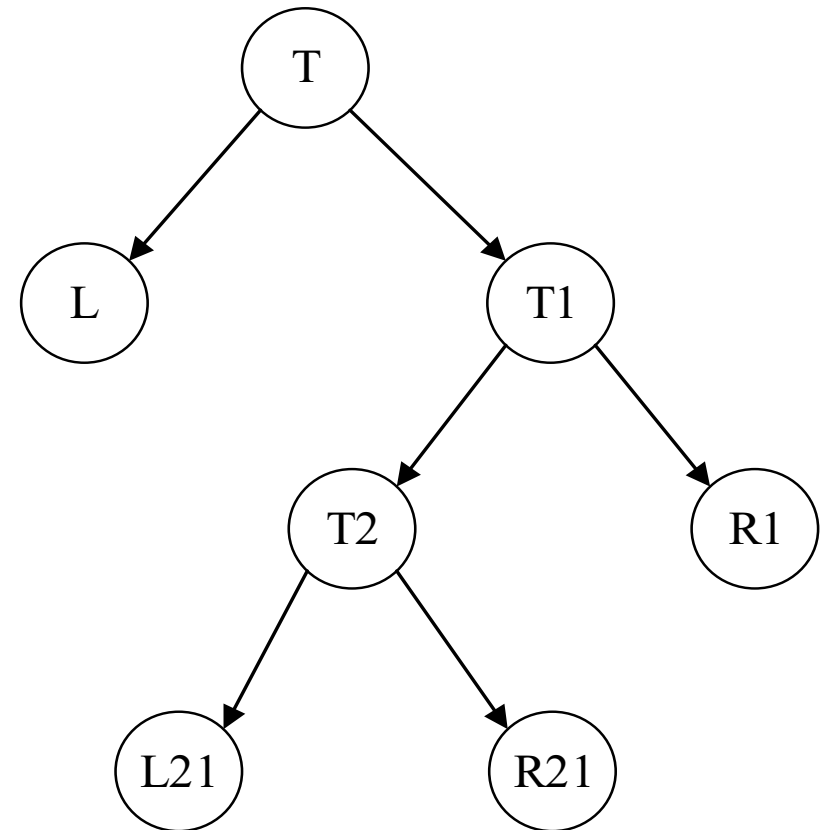
# Các trường hợp mất cân bằng do lệch phải

➤ Cây mất cân bằng tại nút T

TH3: Right-Right



TH4: Right-Left

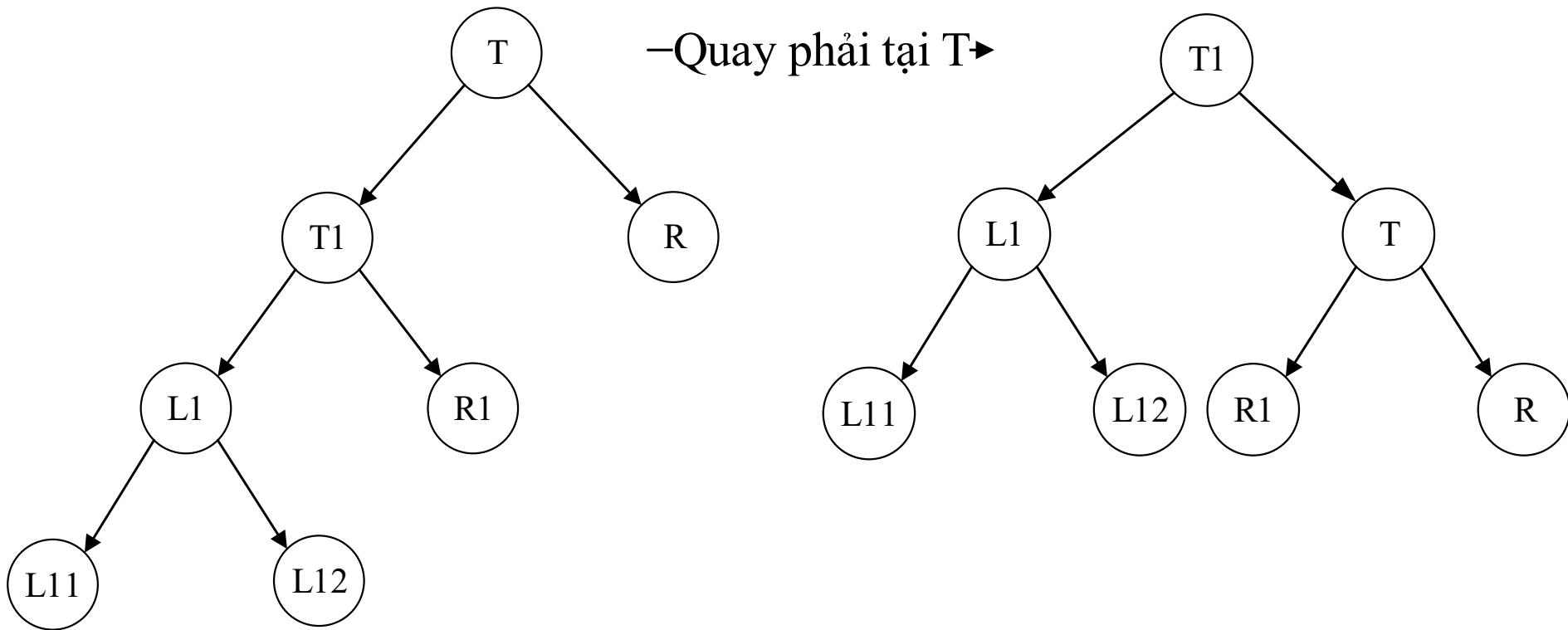




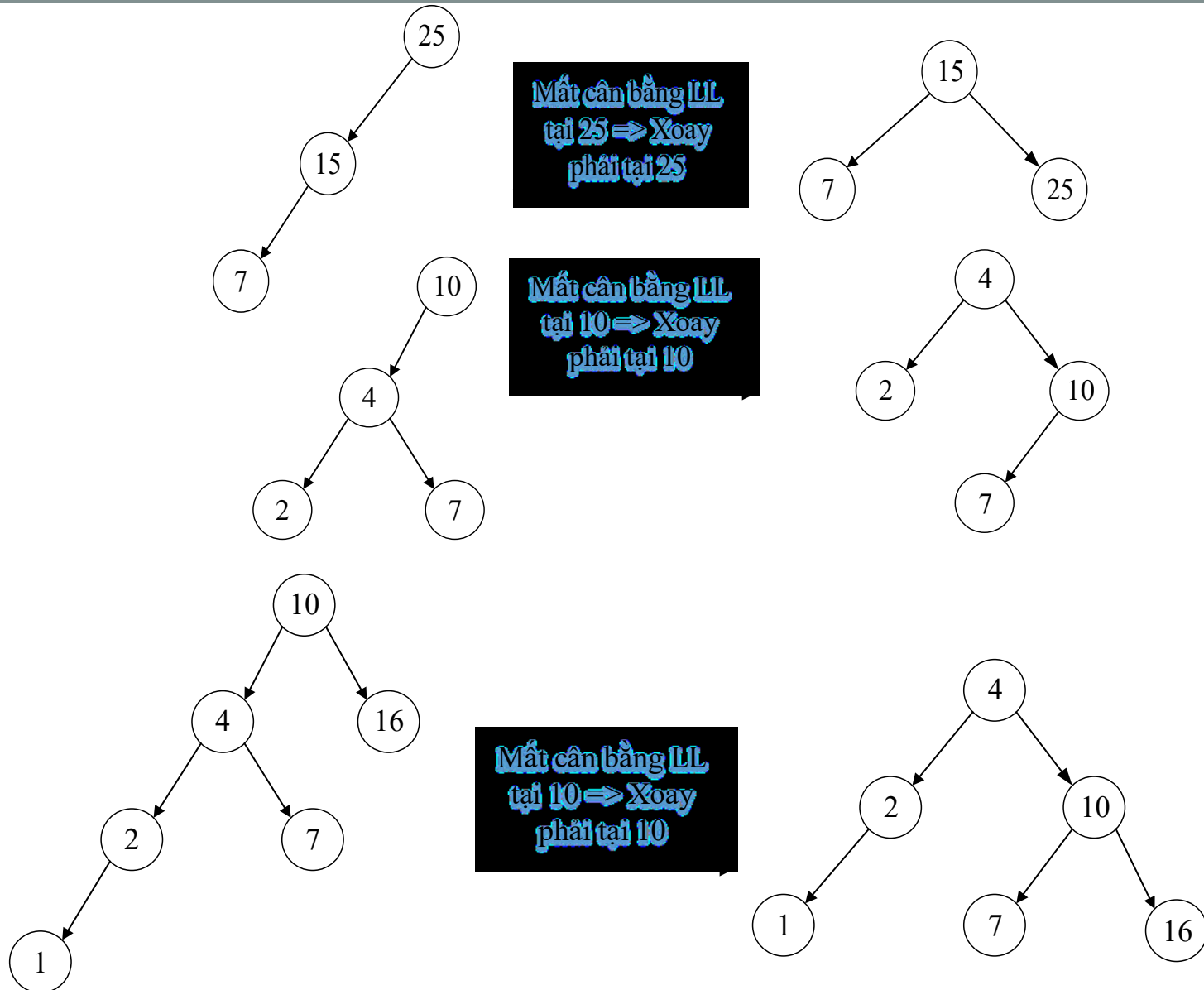
# Các thao tác trên cây cân bằng

- Khi **thêm** hay **xoá** 1 nút trên cây, có thể làm cho cây mất tính cân bằng, khi ấy ta phải tiến hành cân bằng lại.
- Cây có khả năng mất cân bằng khi thay đổi chiều cao:
  - Lệch nhánh trái, thêm bên trái
  - Lệch nhánh phải, thêm bên phải
  - Lệch nhánh trái, hủy bên phải
  - Lệch nhánh phải, hủy bên trái
- Cân bằng lại cây: tìm cách bố trí lại cây sao cho chiều cao 2 cây con cân đối:
  - Kéo nhánh cao bù cho nhánh thấp
  - Phải bảo đảm cây vẫn là Nhị phân tìm kiếm

# Cân bằng lại trường hợp 1



# Ví dụ minh họa Trường hợp mất cân bằng LL:

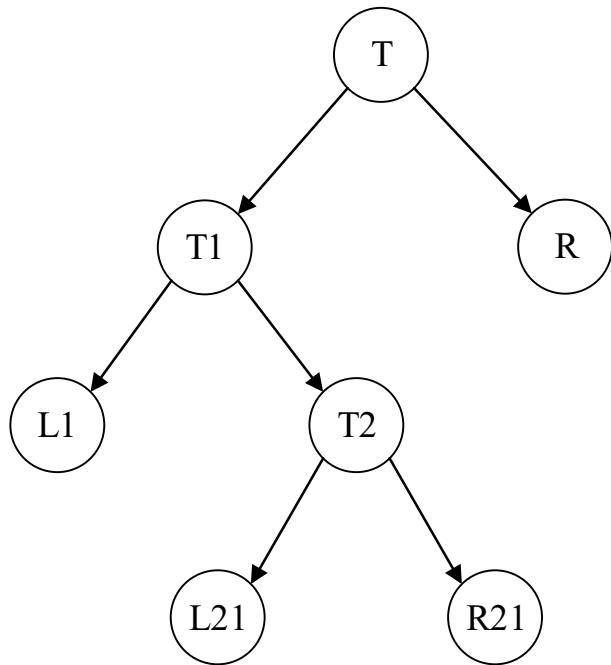


# Cài đặt cân bằng lại cho trường hợp 1

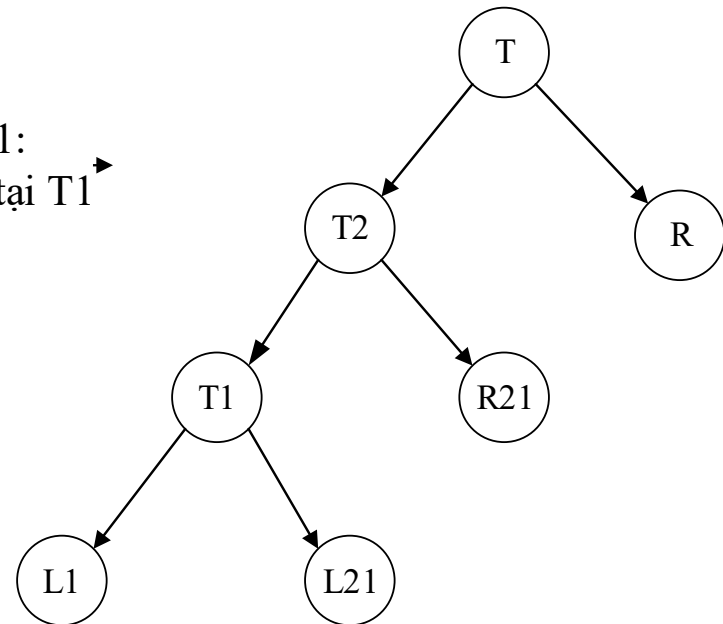


```
void LL(AVLTree &T) {
    AVLNode* T1 = T->pLeft;
    T->pLeft = T1->pRight;
    T1->pRight = T;
    switch (T1->balFactor) {
        case LH:    T->balFactor = EH;
                    T1->balFactor = EH; break;
        case EH:    T->balFactor = LH;
                    T1->balFactor = RH; break;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

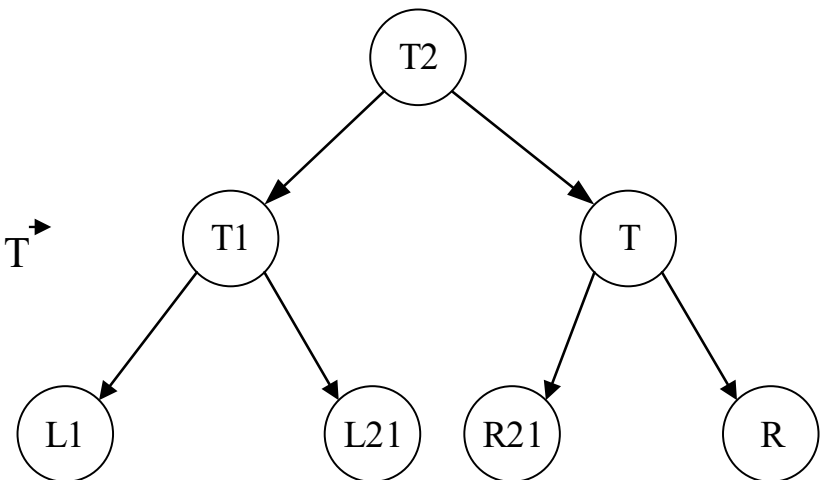
# Cân bằng lại trường hợp 2



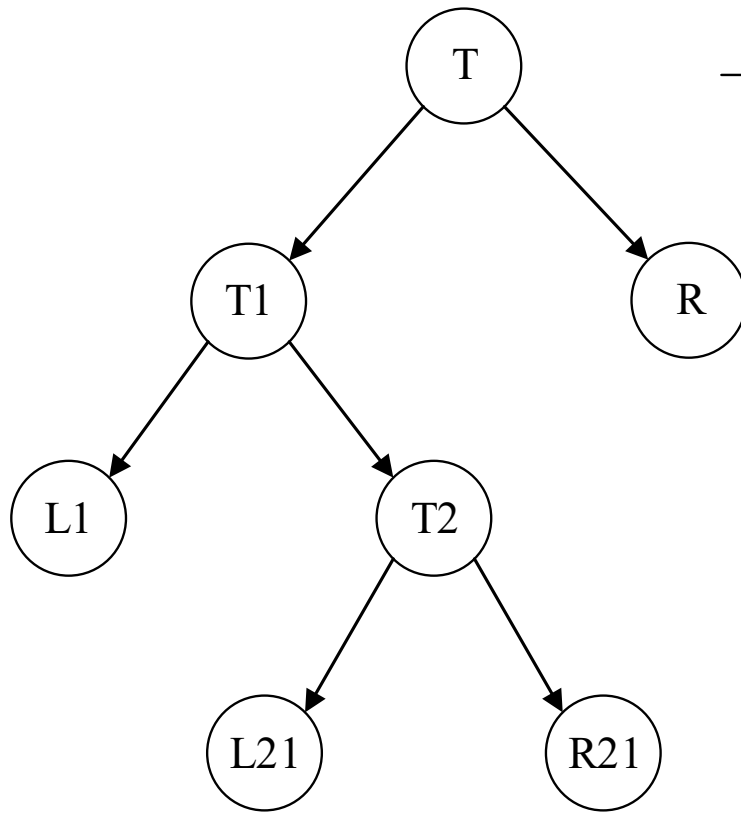
Bước 1:  
- Quay trái tại T1 →



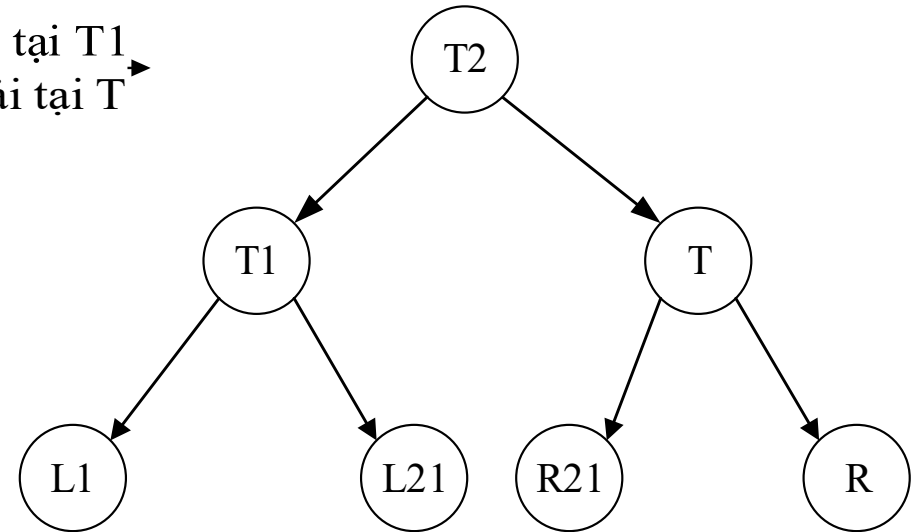
Bước 2  
- Quay phải tại T →



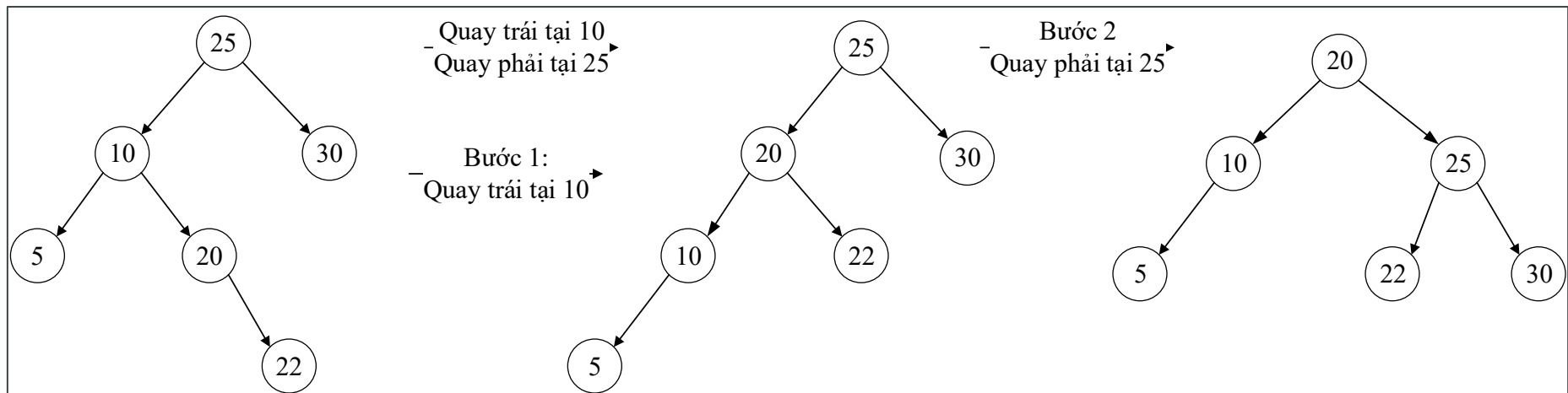
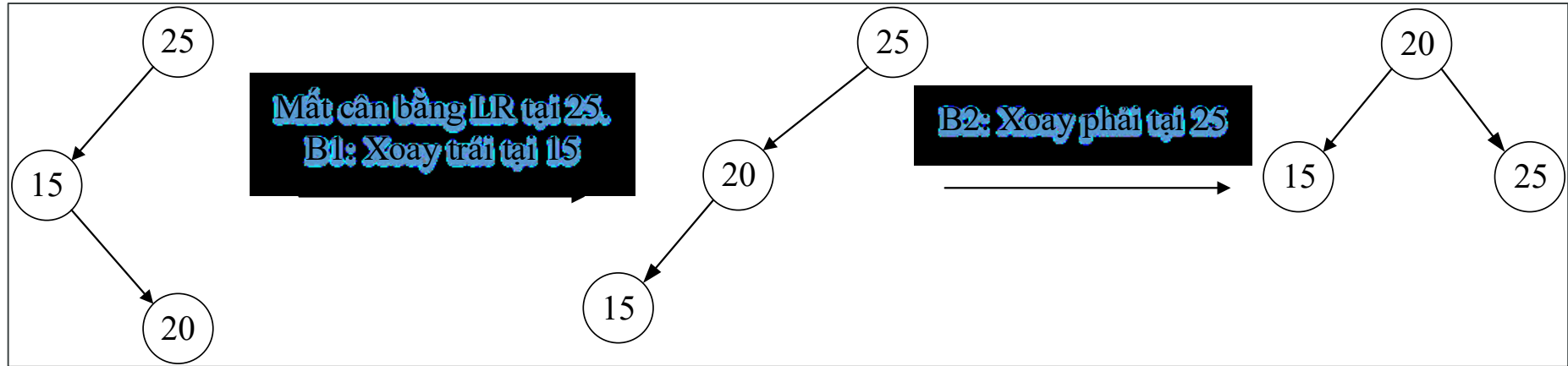
# Cân bằng lại trường hợp 2



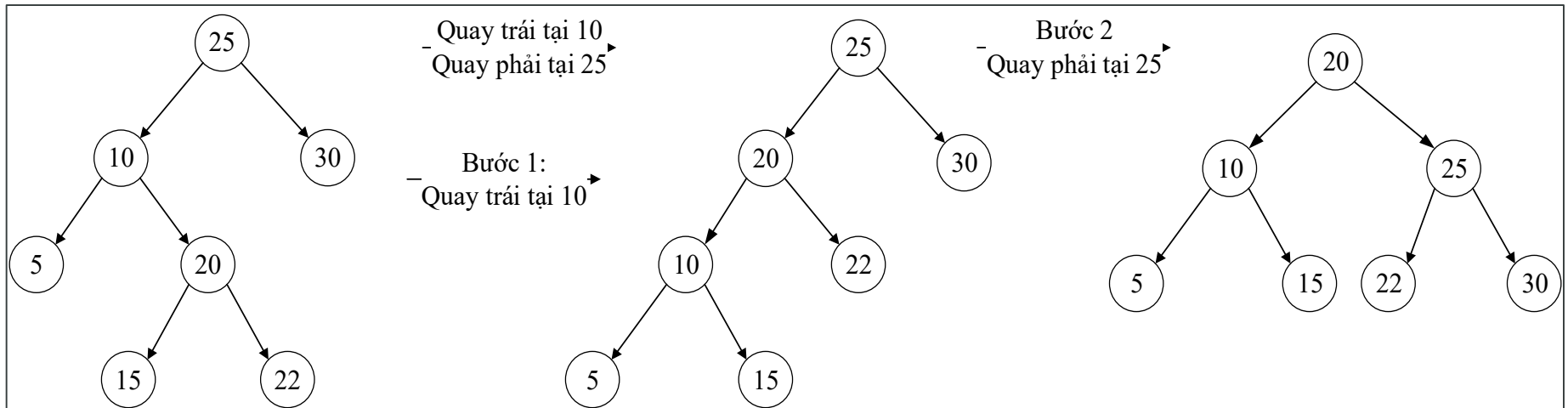
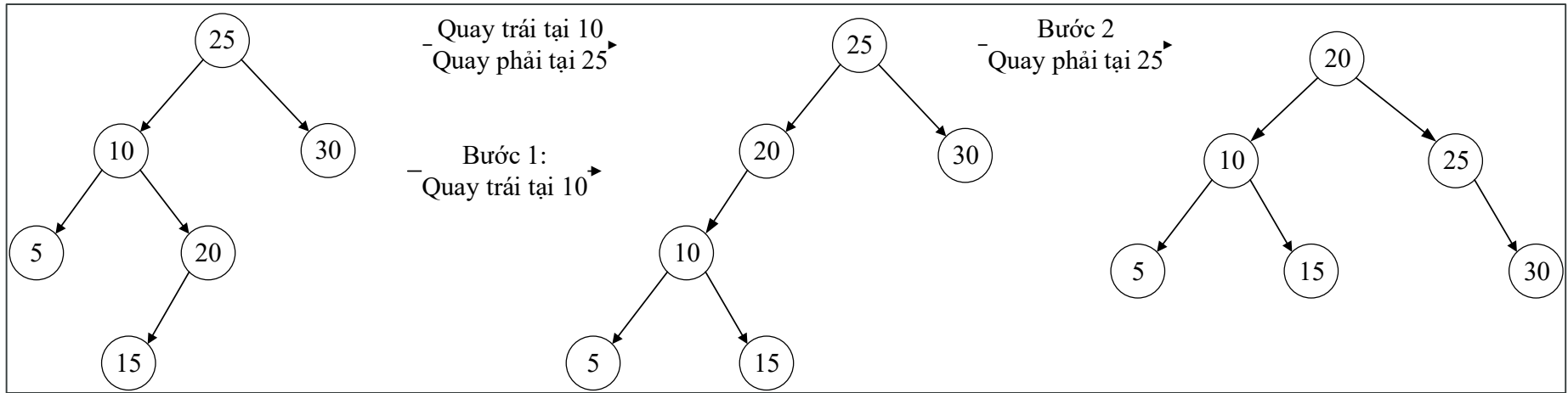
– Quay trái tại T1 →  
– Quay phải tại T →



# Ví dụ minh họa Trường hợp mất cân bằng LR:



# Ví dụ minh họa Trường hợp mất cân bằng LR:



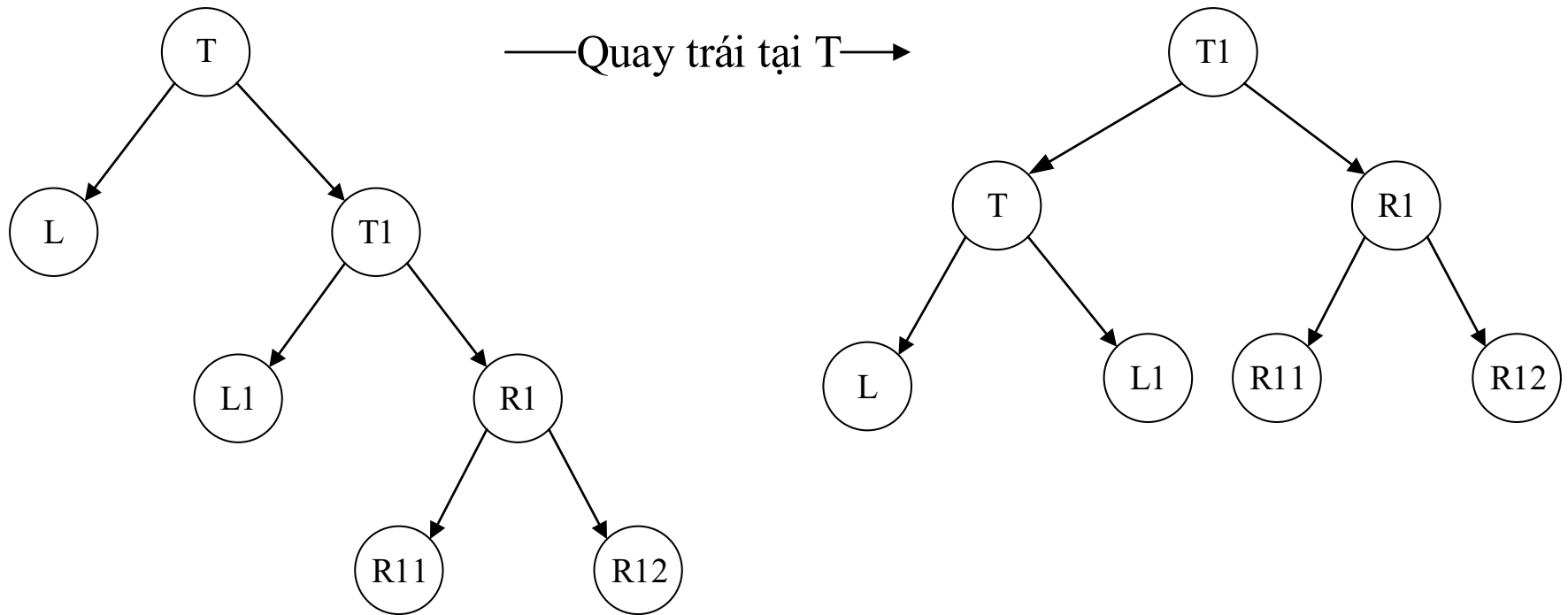


# Cài đặt cân bằng lại cho trường hợp 2

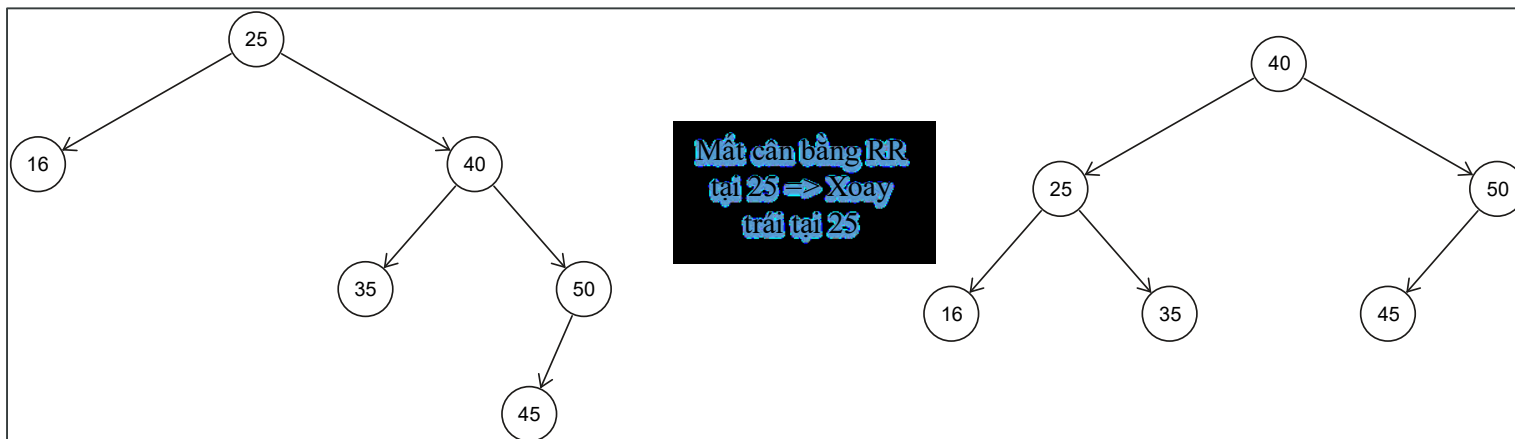
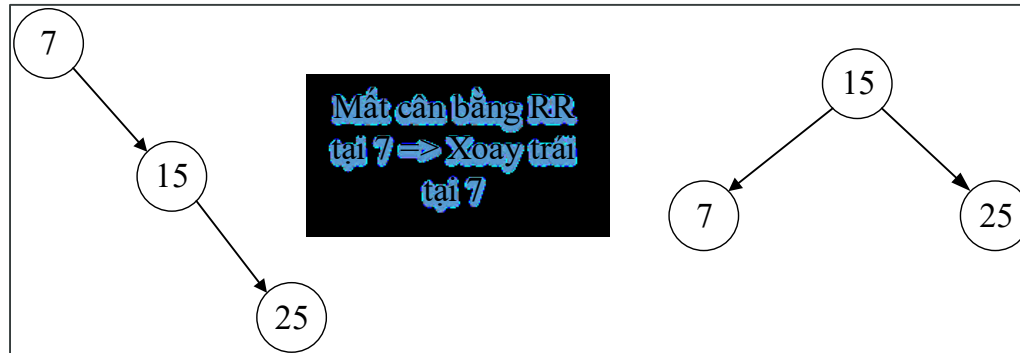


```
void LR(AVLTree &T) {
    AVLNode *T1 = T->pLeft;
    AVLNode *T2 = T1->pRight;
    T->pLeft = T2->pRight;
    T2->pRight = T;
    T1->pRight = T2->pLeft;
    T2->pLeft = T1;
    switch (T2->balFactor) {
        case LH: T->balFactor = RH;
                 T1->balFactor = EH; break;
        case EH: T->balFactor = EH;
                 T1->balFactor = EH; break;
        case RH: T->balFactor = EH;
                 T1->balFactor = LH; break;
    }
    T2->balFactor = EH;
    T = T2;
}
```

# Cân bằng lại trường hợp 3



# Ví dụ minh họa Trường hợp mất cân bằng RR:

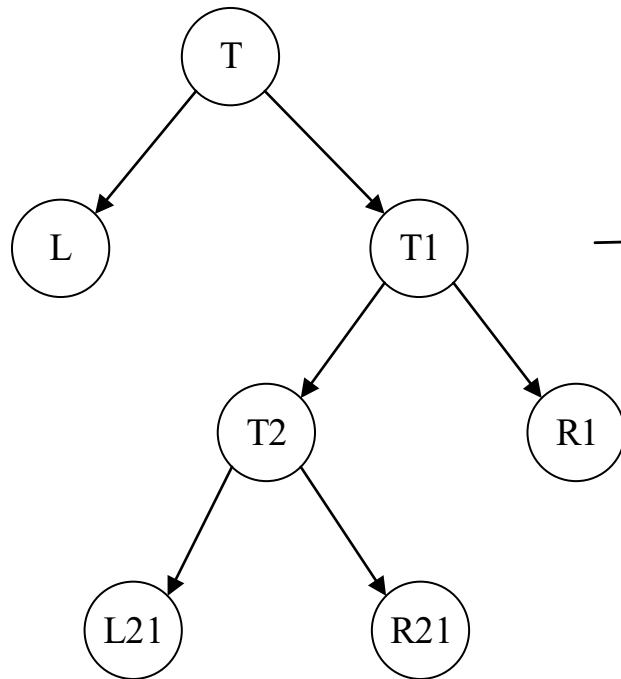


# Cài đặt cân bằng lại cho trường hợp 3

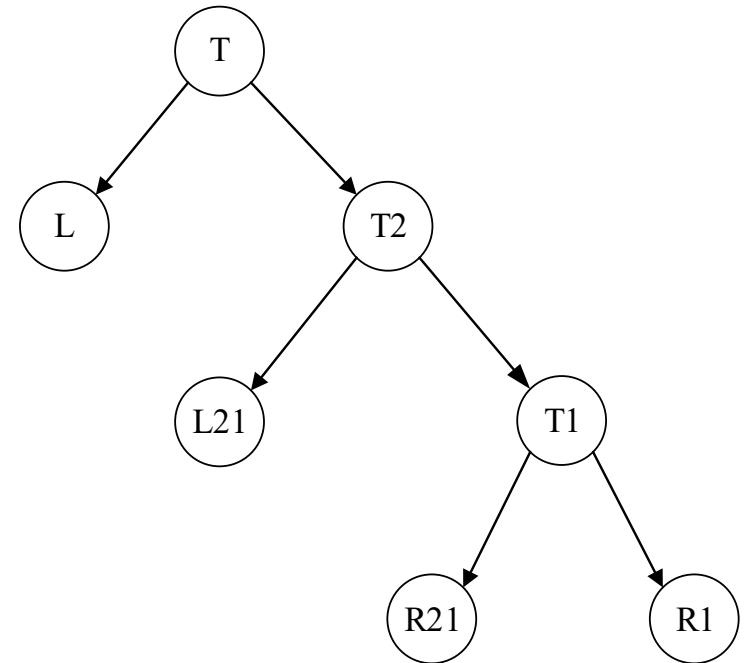


```
void RR(AVLTree &T) {
    AVLNode* T1 = T->pRight;
    T->pRight = T1->pLeft;
    T1->pLeft = T;
    switch (T1->balFactor) {
        case RH:    T->balFactor = EH;
                   T->balFactor = EH; break;
        case EH:    T->balFactor = RH;
                   T1->balFactor = LH; break;
 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

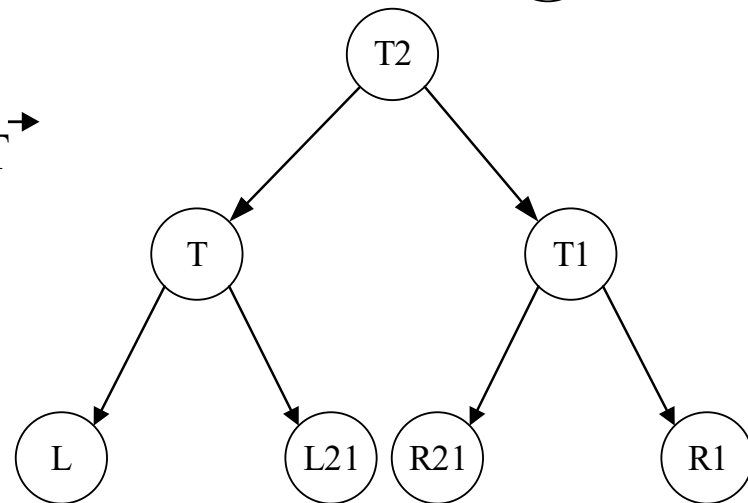
# Cân bằng lại trường hợp 4



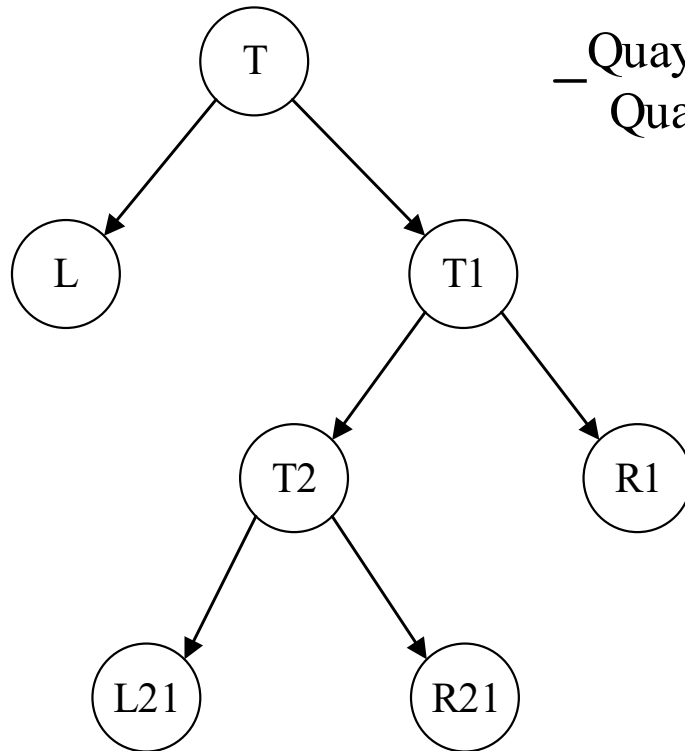
Bước 1:  
– Quay phải tại T1 →



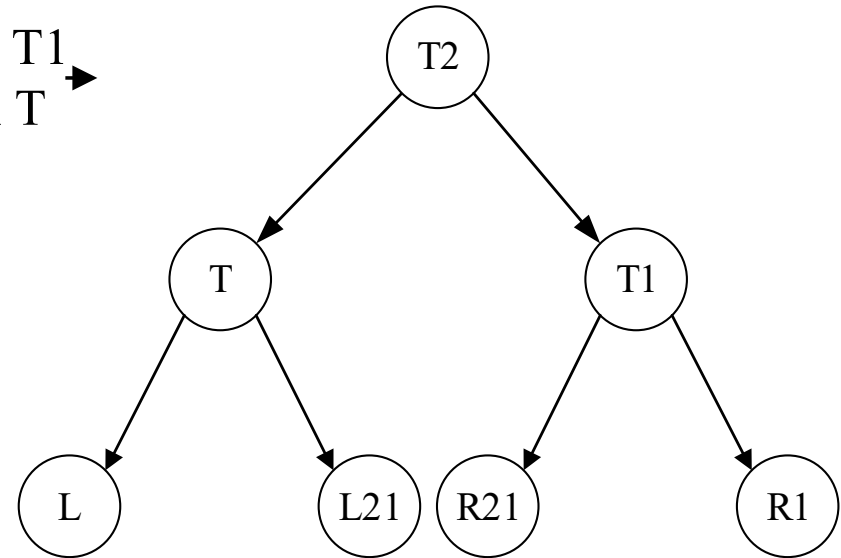
Bước 2:  
– Quay trái tại T →



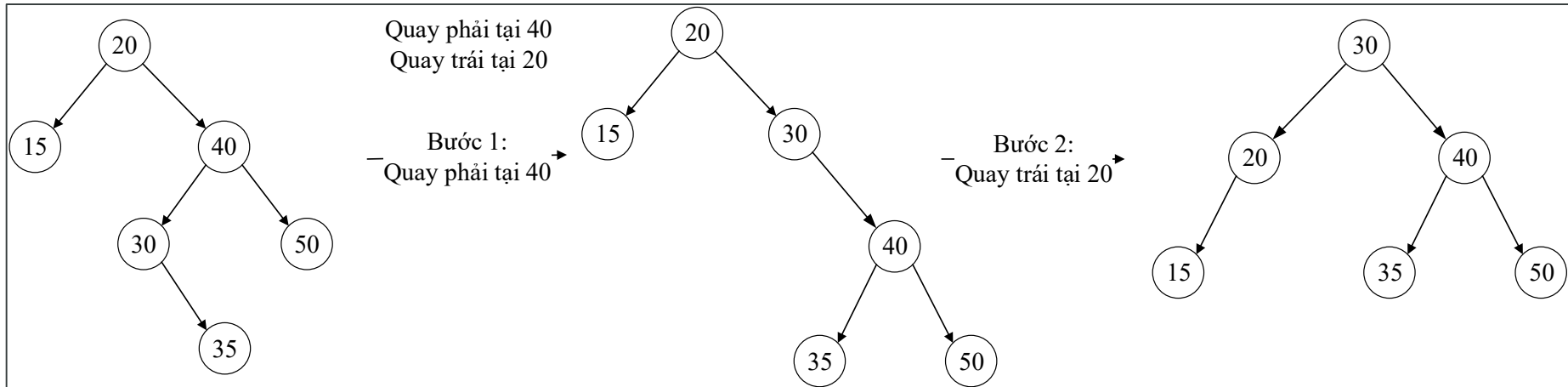
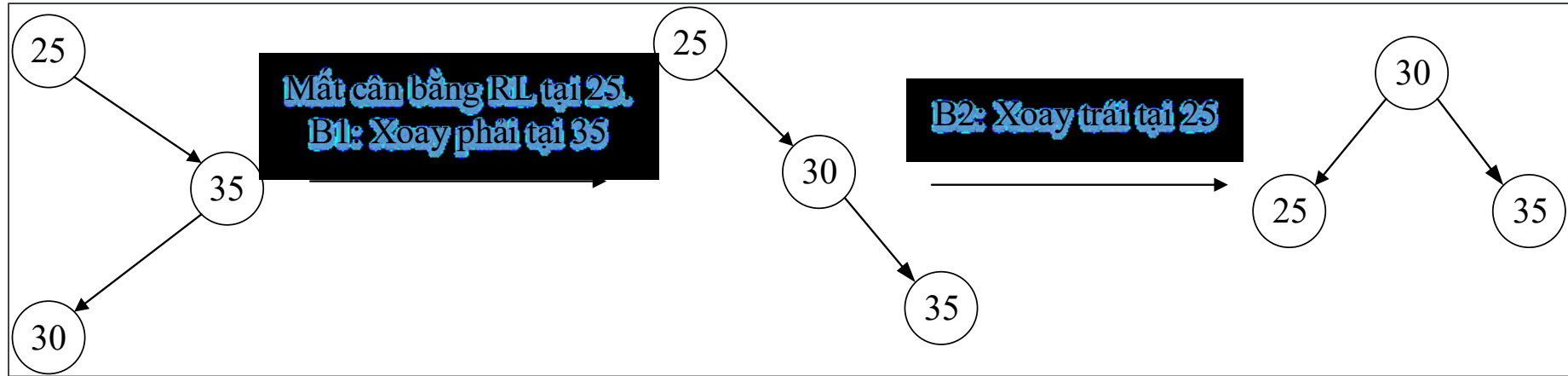
# Cân bằng lại trường hợp 4: (tổng hợp 2 bước)



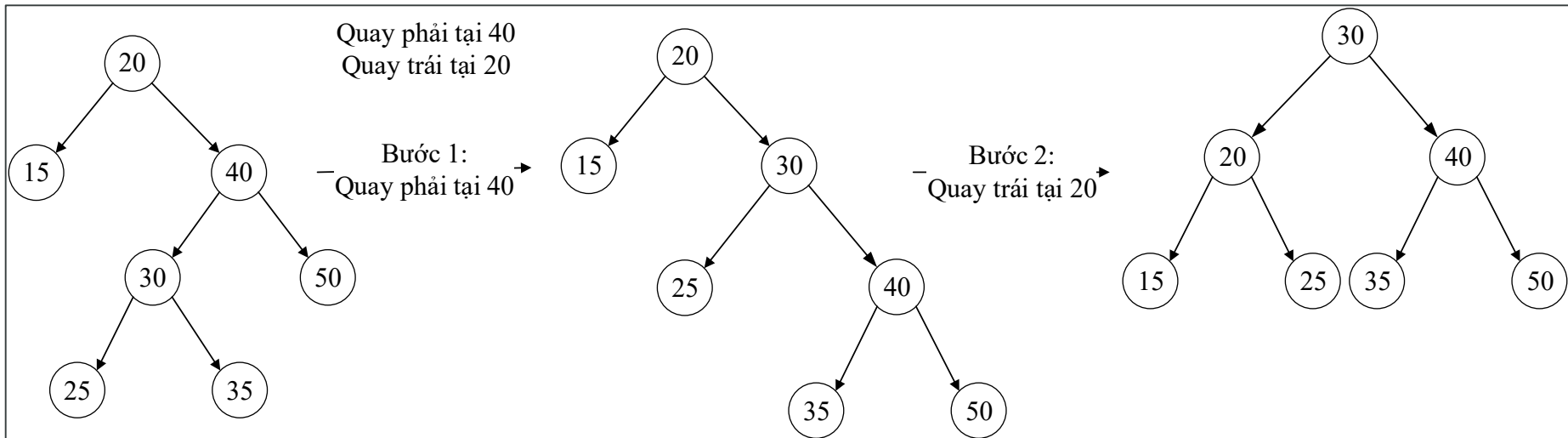
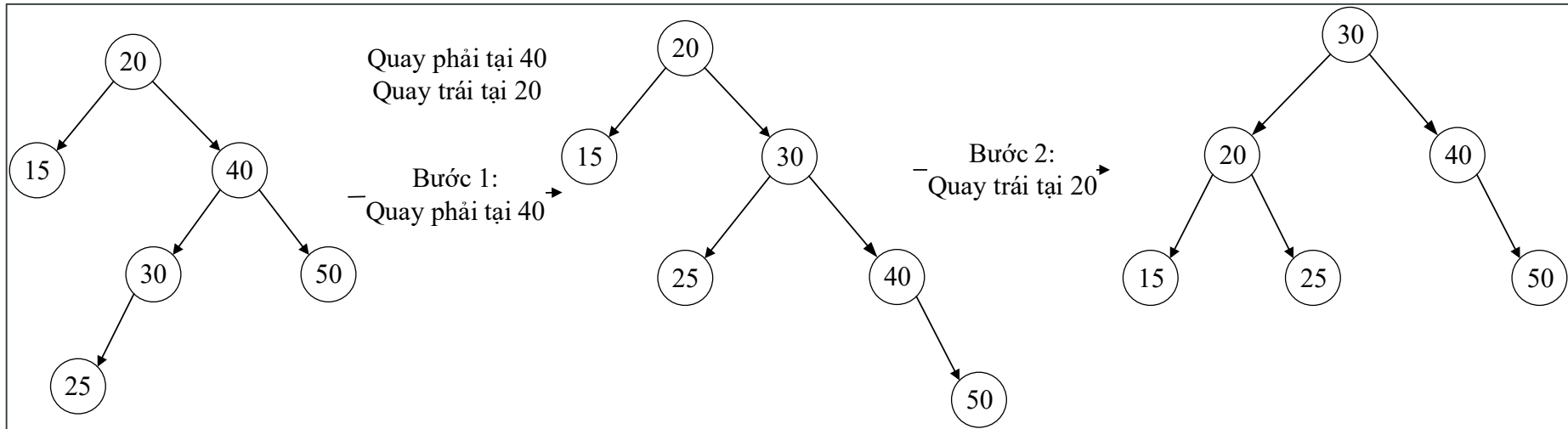
— Quay phải tại T1  
Quay trái tại T →



# Ví dụ minh họa Trường hợp mất cân bằng RL:



# Ví dụ minh họa Trường hợp mất cân bằng RL:





# Cài đặt cân bằng lại cho trường hợp 4



```
void RL(AVLTree &T) {
    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;
}
```



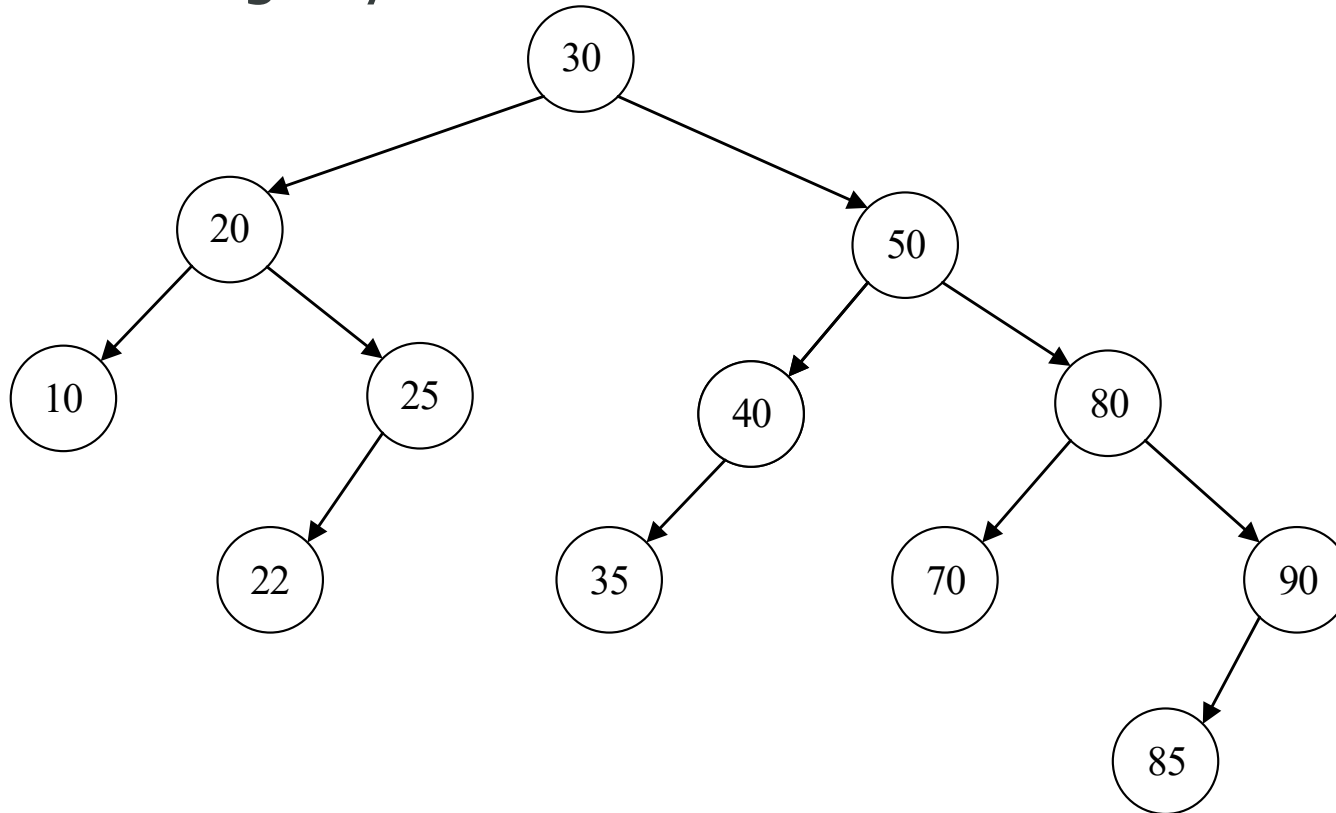
- Thêm bình thường như trường hợp cây NPTK
- Nếu cây tăng trưởng chiều cao
  - Lăn ngược về gốc để phát hiện nút bị mất cân bằng
  - Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp
- Việc cân bằng lại chỉ cần thực hiện 1 lần nơi mất cân bằng

# Hủy 1 nút

- Hủy bình thường như trường hợp cây NPTK
- Nếu cây giảm chiều cao:
  - Lăn ngược về gốc để phát hiện nút bị mất cân bằng
  - Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp
  - Tiếp tục lăn ngược lên nút cha...
- Việc cân bằng lại có thể lan truyền lên tận gốc

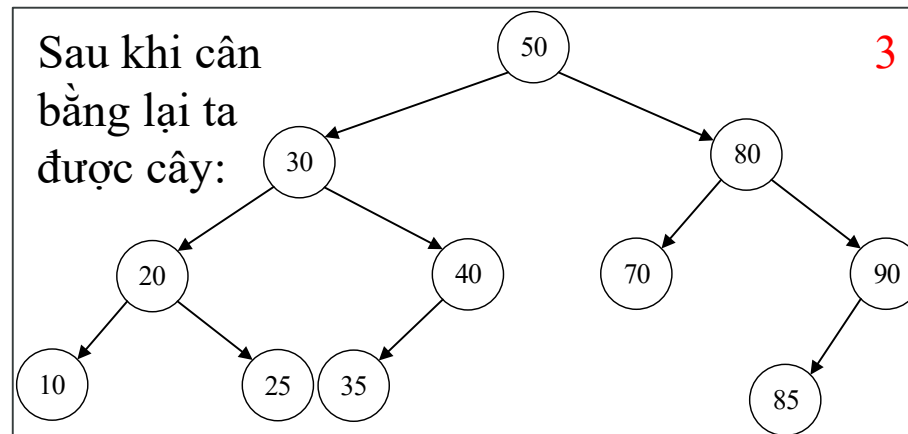
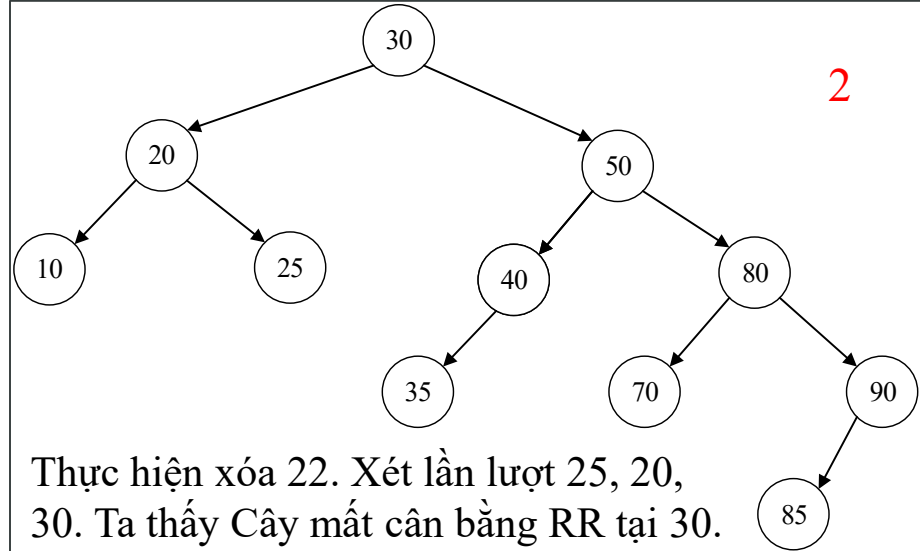
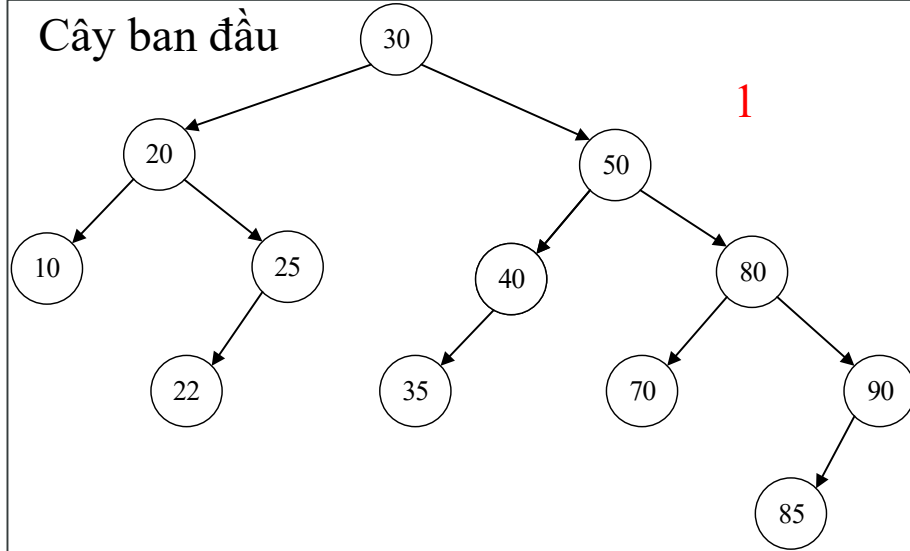
## Ví dụ 1:

- Hủy số 22 trong cây sau:



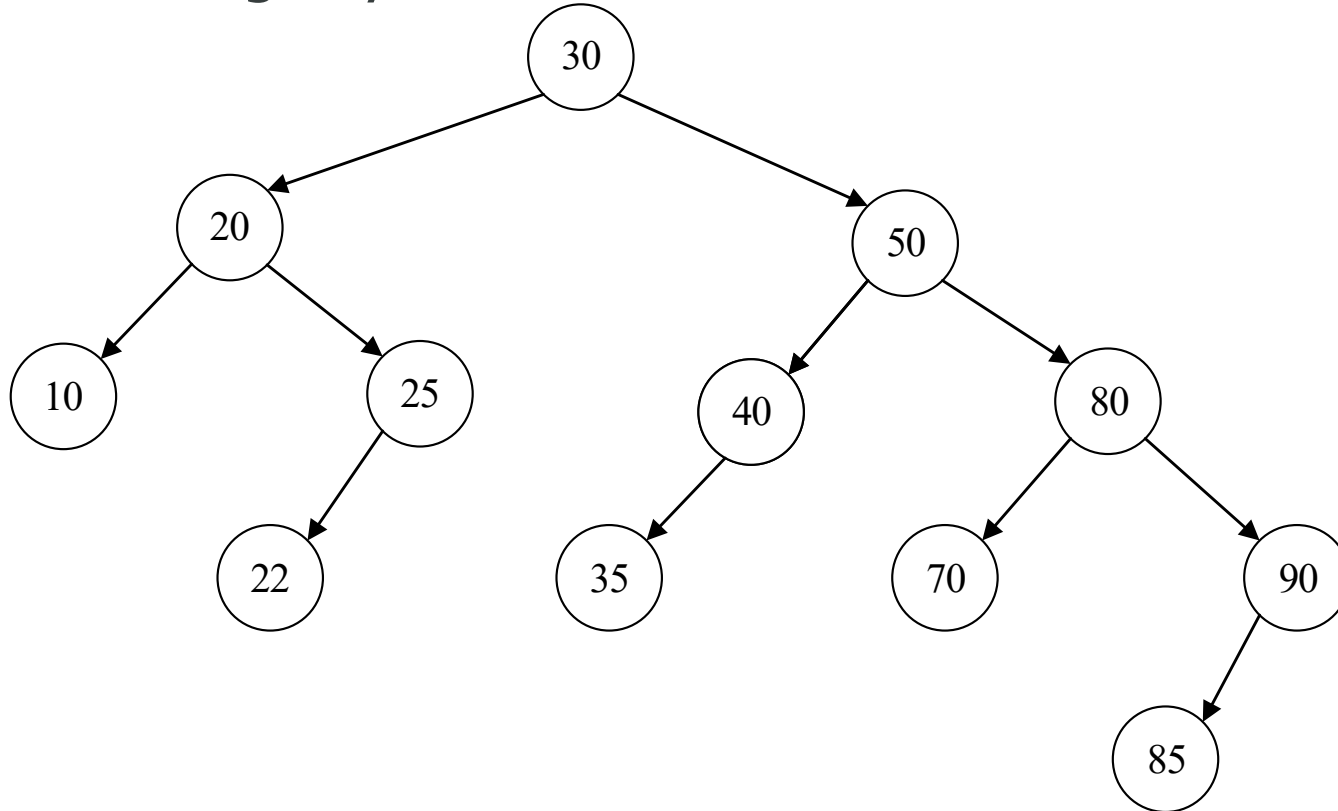
- Ta nhận thấy sau khi xóa số 22 thì chiều cao cây con trái của node 25 thay đổi => Lần lượt xét sự mất cân bằng của các node 25, 20, 30 và cân bằng lại nếu bị mất cân bằng.

# Ví dụ 1:



## Ví dụ 2: (Hủy lan truyền)

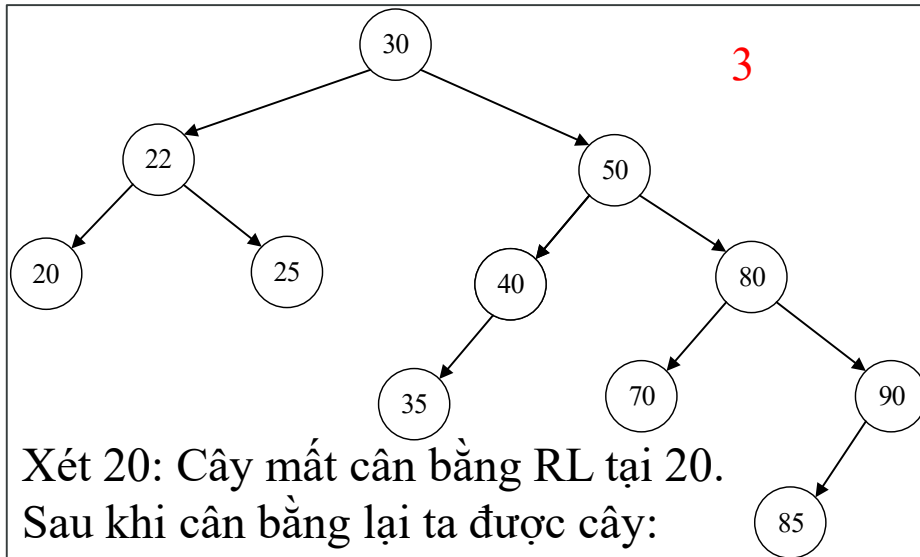
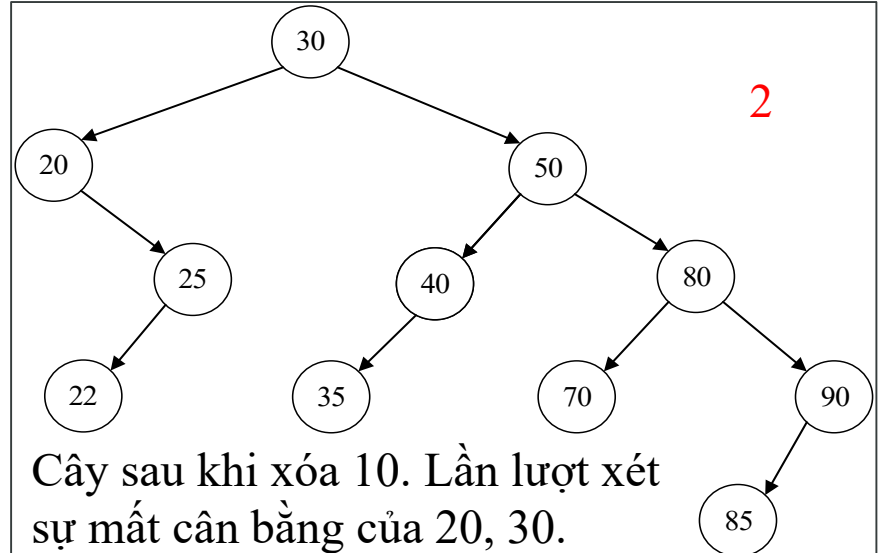
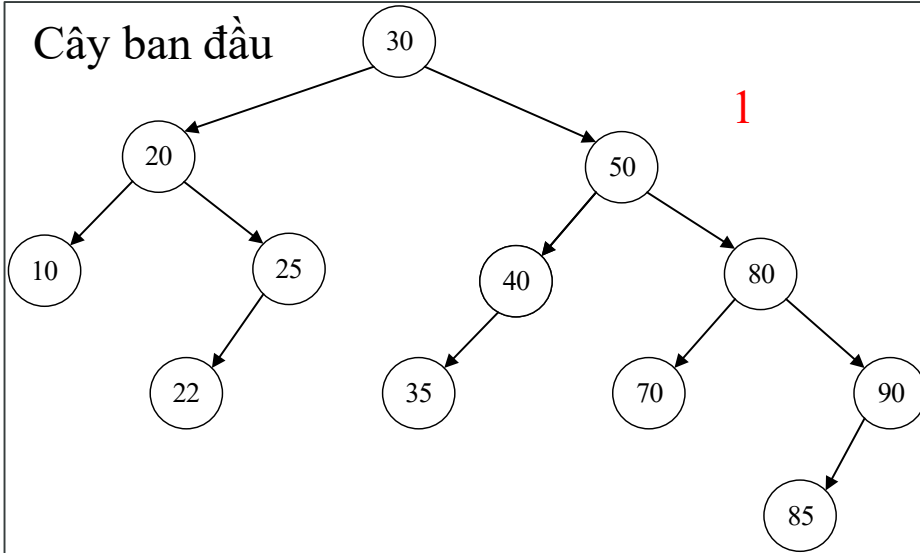
- Hủy số 10 trong cây sau:



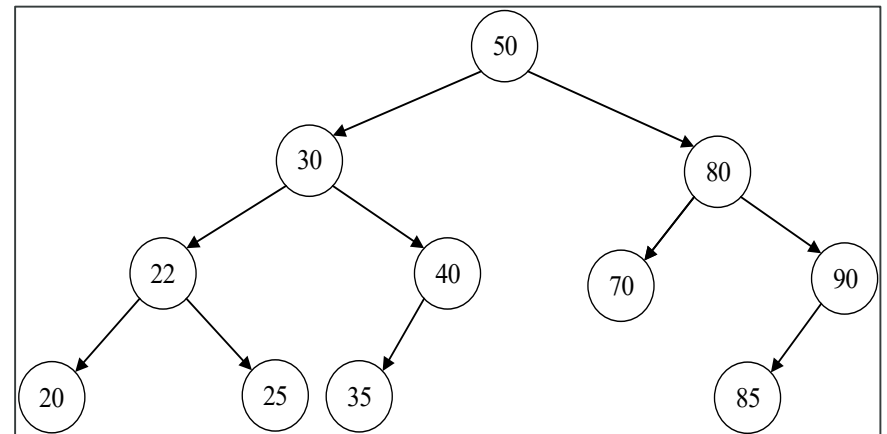
- Ta nhận thấy sau khi xóa số 10 thì chiều cao cây con trái của node 20 thay đổi => Lần lượt xét sự mất cân bằng của các node 20, 30 và cân bằng lại nếu bị mất cân bằng.

## Ví dụ 2: (Hủy lan truyền)

Cây ban đầu

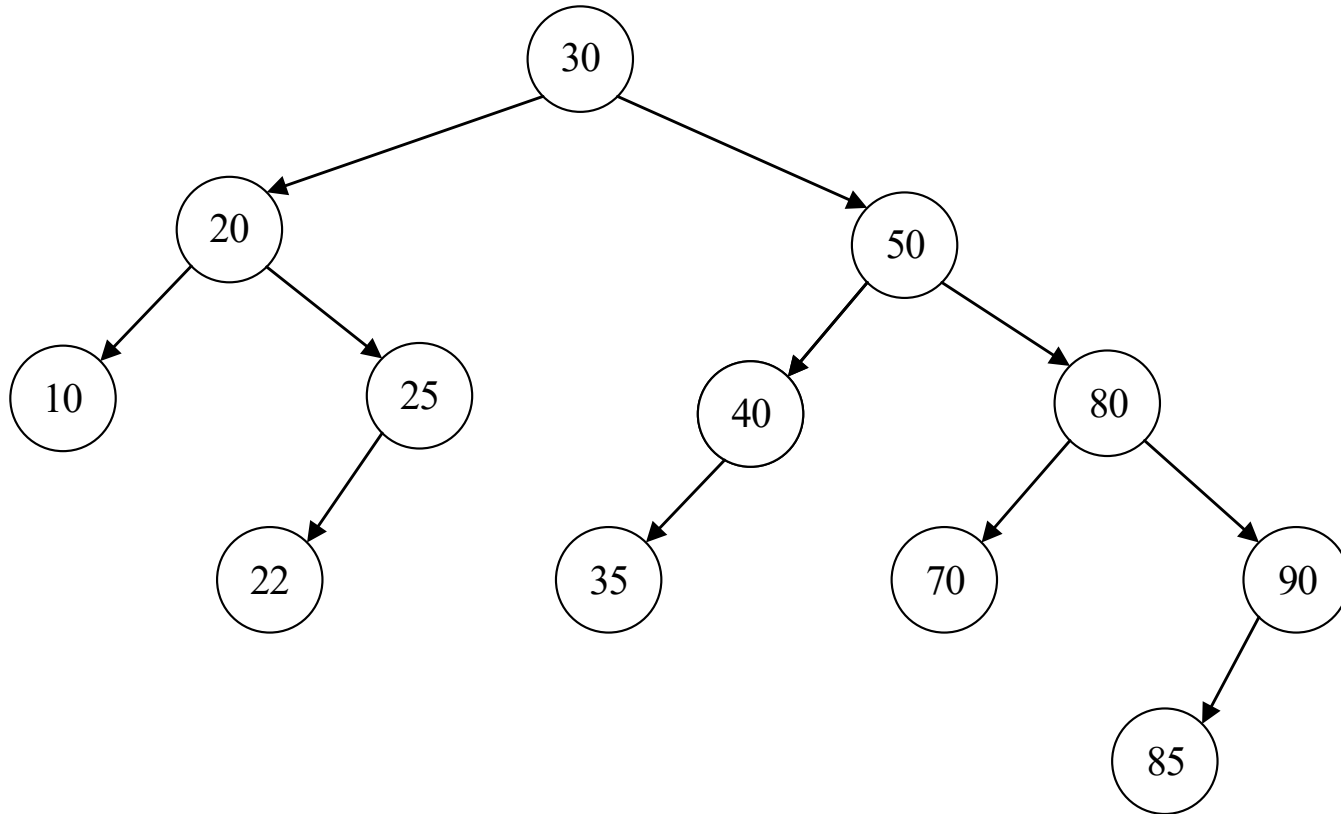


Xét 30: Cây mất cân bằng RR tại 30.  
Sau khi cân bằng lại ta được cây:



## Ví dụ 3.1:

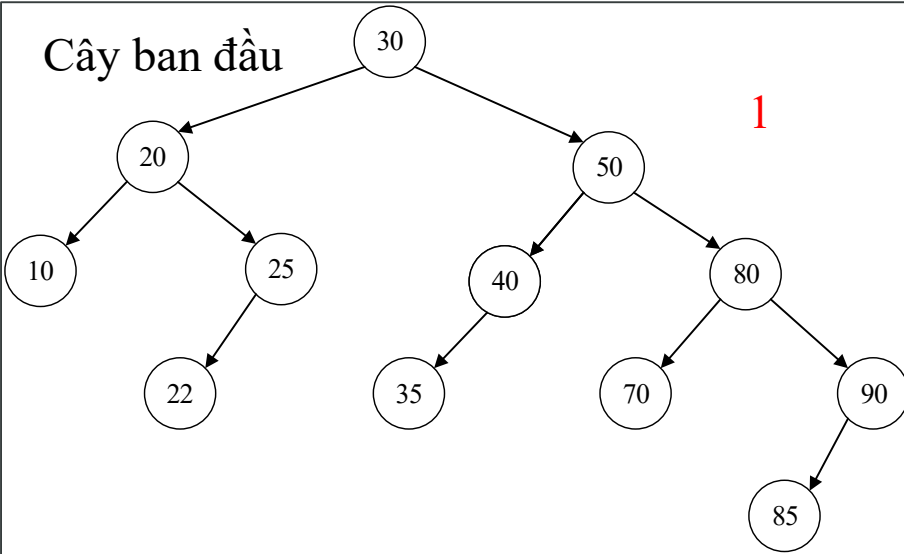
- Hủy số 30 trong cây sau: (Chọn 25 thay thế)



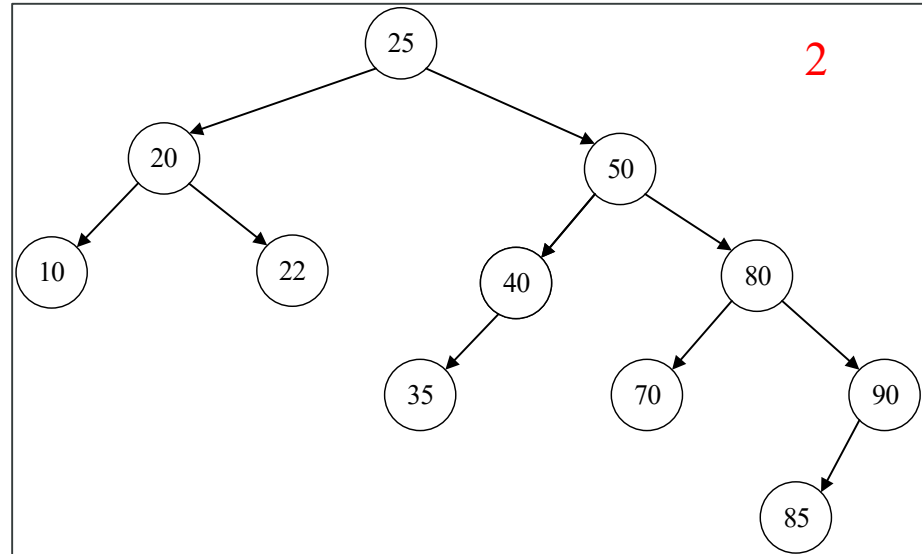


## Ví dụ 3.1:

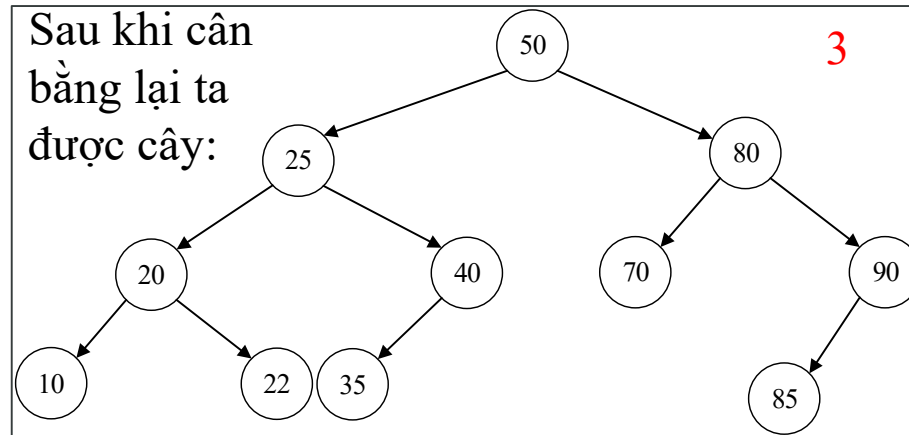
Cây ban đầu



Thực hiện xóa 30. Lấy 25 thay thế. Xét lần lượt sự mất cân bằng ở node 20, 25. Ta thấy cây mất cân bằng RR tại 25.

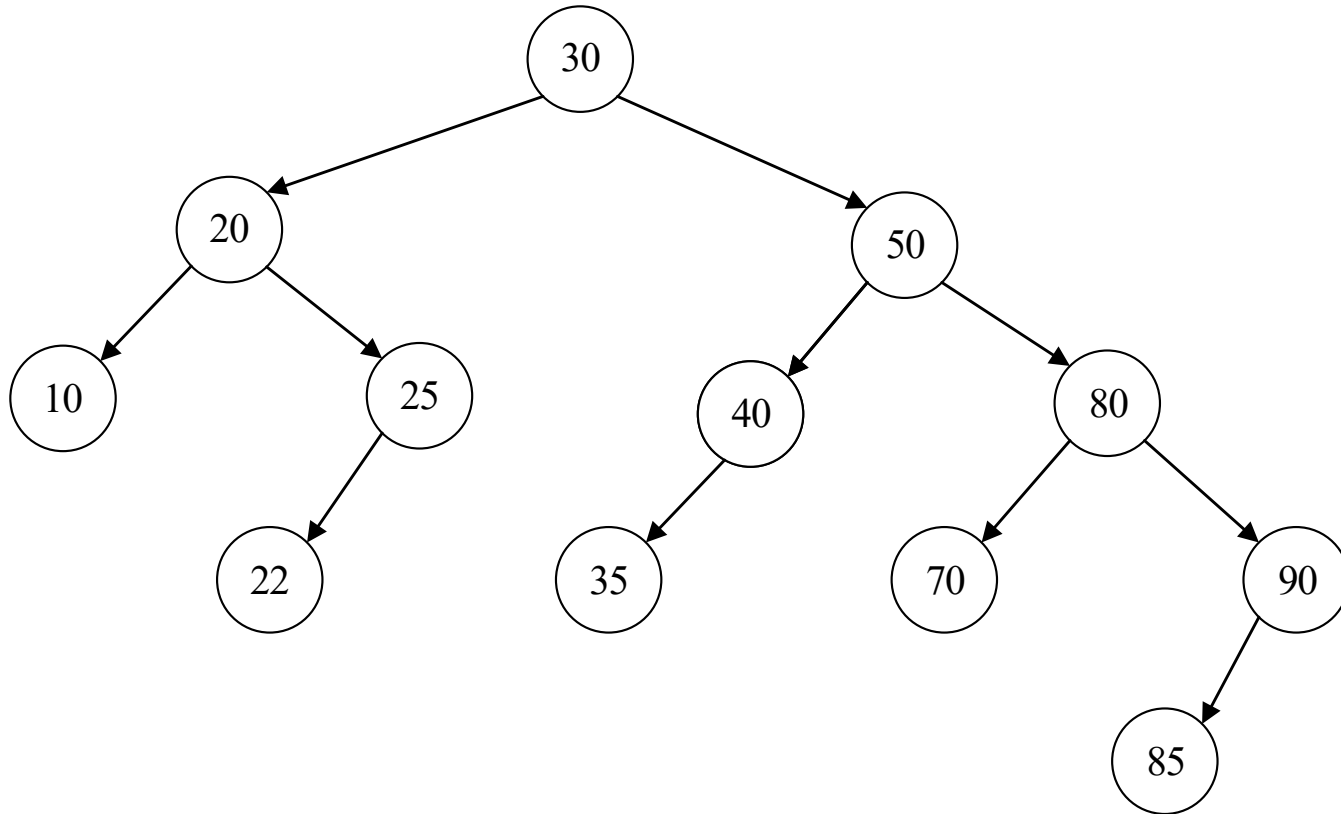


Sau khi cân bằng lại ta được cây:



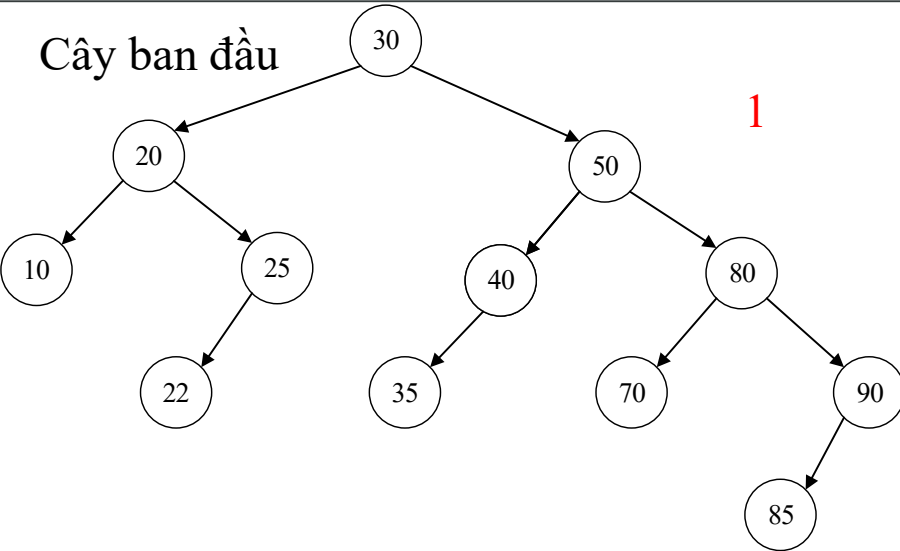
## Ví dụ 3.2:

- Hủy số 30 trong cây sau: (Chọn 35 thay thế)

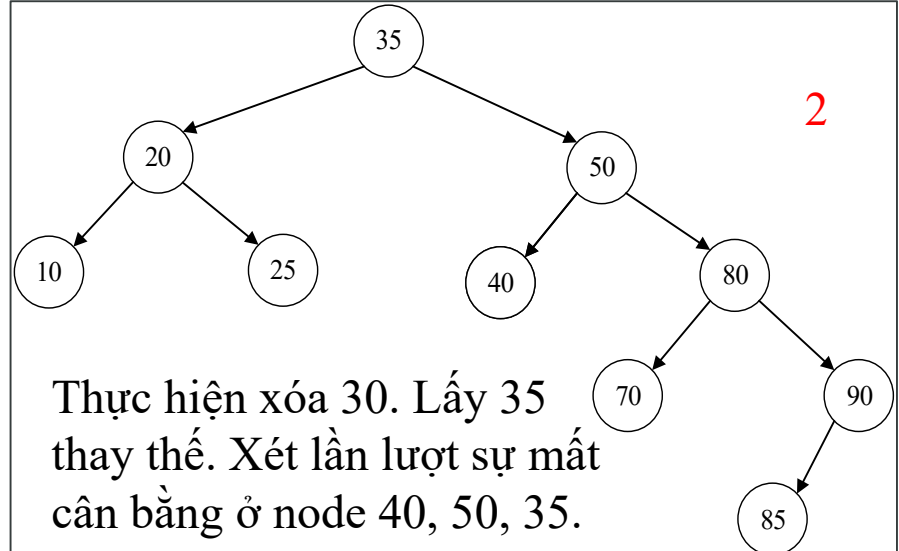


## Ví dụ 3.2:

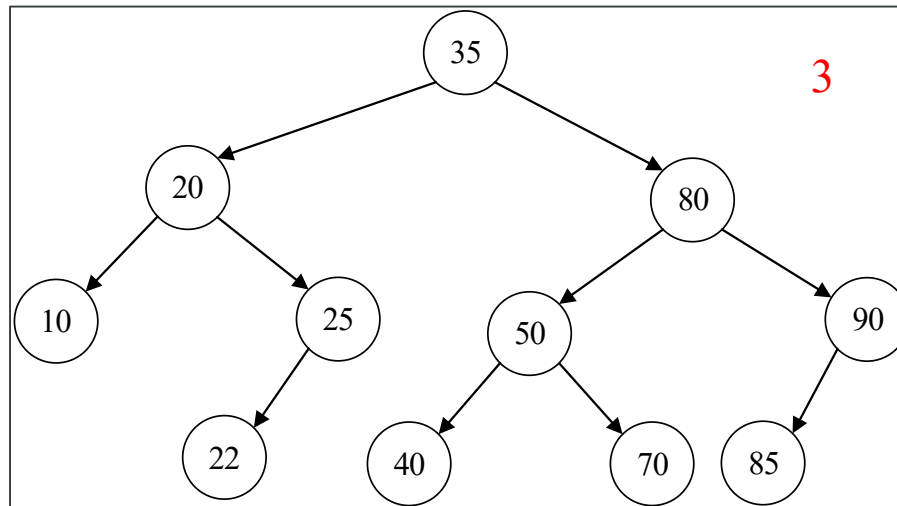
Cây ban đầu



Thực hiện xóa 30. Lấy 35 thay thế. Xét lần lượt sự mất cân bằng ở node 40, 50, 35.



Xét 50: Cây mất cân bằng RR tại 50. Sau khi cân bằng lại ta được cây:



# Ứng dụng:



# Bài tập

1. Xét thuật giải tạo cây nhị phân tìm kiếm cân bằng. Nếu thứ tự các khóa nhập vào như sau:

3 5 2 20 11 30 9 18 4

thì hình ảnh cây tạo được như thế nào?

Sau đó, nếu hủy lần lượt các nút 5, 20 thì cây sẽ thay đổi như thế nào trong từng bước hủy, vẽ sơ đồ.

2. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.
3. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm cân bằng.
4. Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh-Việt.

**Chúc các em học tốt!**

