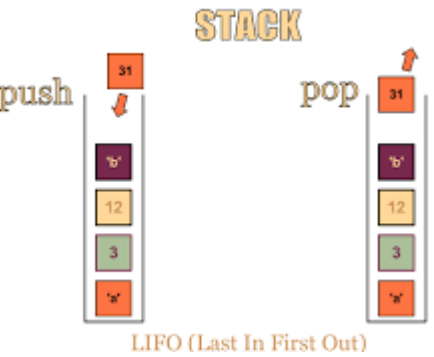
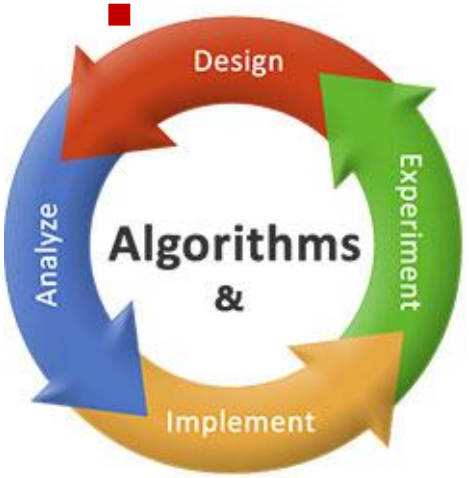
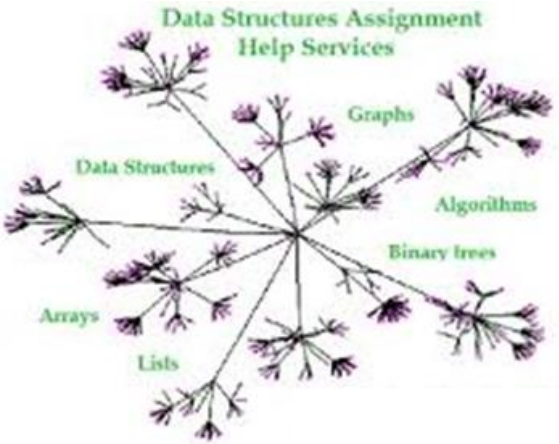


CẤU TRÚC DỮ LIỆU

&

GIẢI THUẬT





Lê Văn Hạnh

levanhanhvn@gmail.com

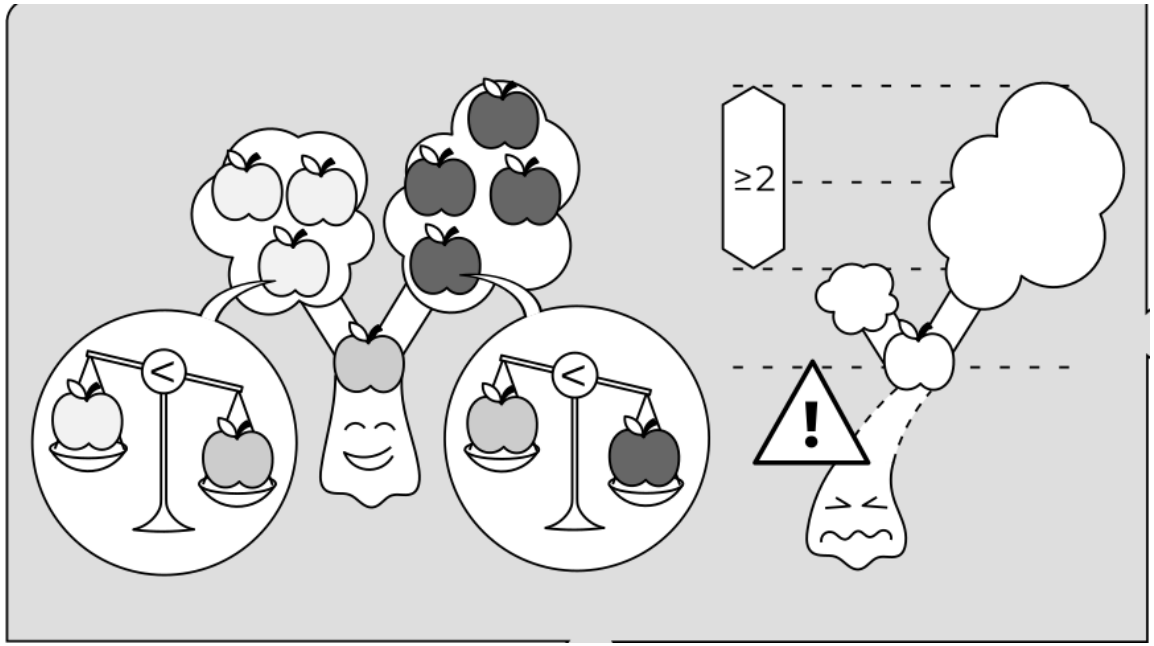
NỘI DUNG MÔN HỌC

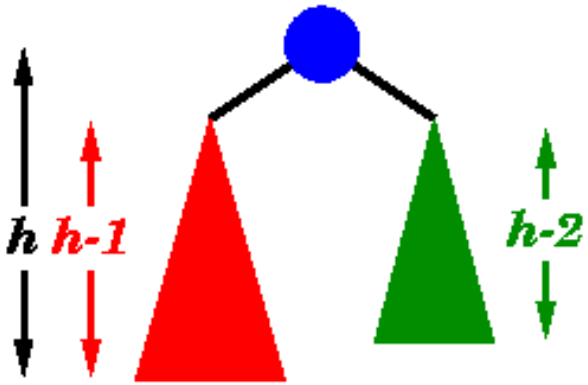
- Chương 1: Ôn tập ngôn ngữ lập trình C
- Chương 2: Kiểu dữ liệu con trỏ
- Chương 3: Tổng quan về cấu trúc dữ liệu và giải thuật
- Chương 4: Danh sách kề (Danh sách tuyến tính)
- Chương 5: Các giải thuật tìm kiếm trên danh sách kề
- Chương 6: Các giải thuật sắp xếp trên danh sách kề
- Chương 7: Danh sách liên kết động (*Linked List*)
- Chương 8: Ngăn xếp *(Stack)*
- Chương 9: Hàng đợi *(Queue)*
- Chương 10: Cây nhị phân tìm kiếm *(Binary Search Tree)*
- **Chương 11: Cây NPTK cân bằng** *(Balanced binary search tree – AVL tree)*
- Chương 12: Bảng băm *(Hash Table)*

Chương 9

CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

(Balanced binary search tree)





MỤC TIÊU

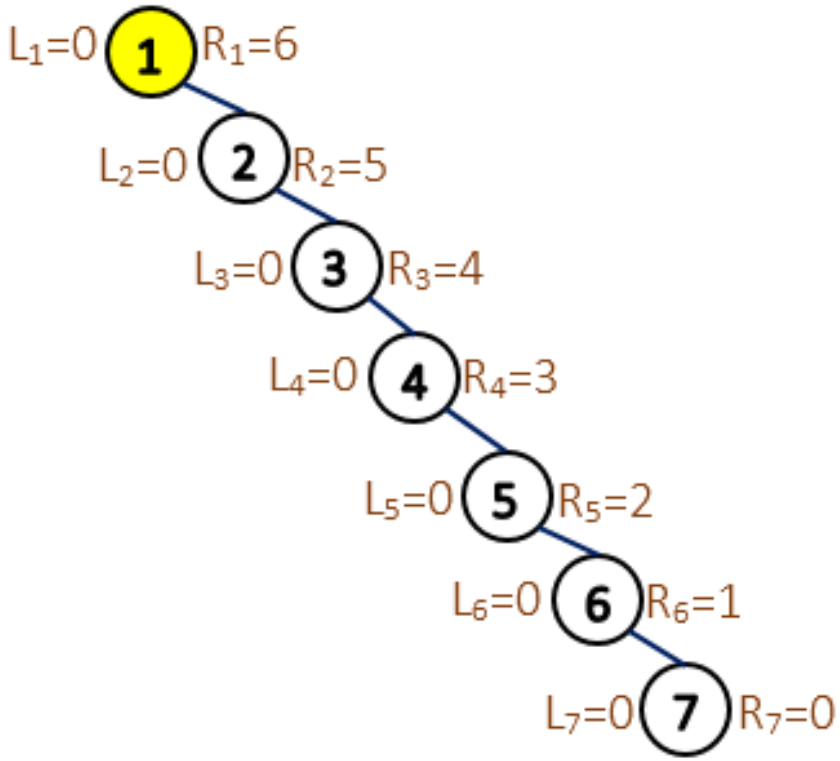
Sau khi học xong bài này, sinh viên có thể:

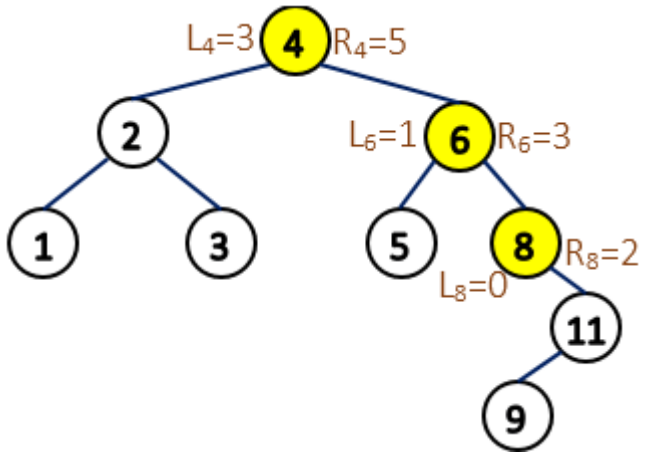
- Hiểu được cấu trúc cây nhị phân tìm kiếm cân bằng
- Nắm được các trường hợp khi thêm xóa nút làm cây mất cân bằng, cân bằng lại cây
- Cài đặt được các thao tác trên cây AVL
- Vận dụng cấu trúc cây AVL để giải các bài toán cụ thể

1. GIỚI THIỆU

1.1. Cây nhị phân tìm kiếm hoàn toàn cân bằng

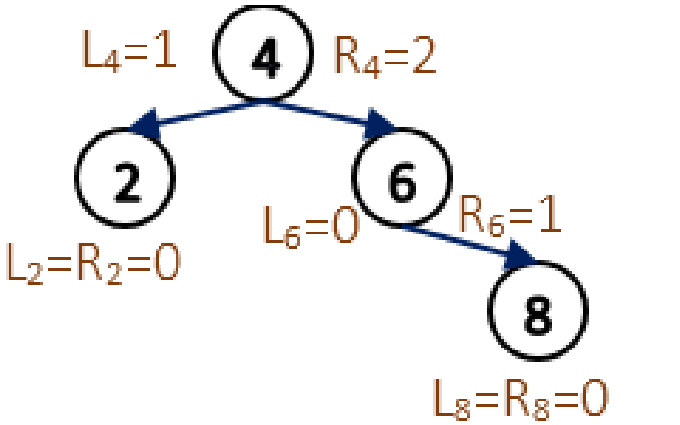
- Người ta dùng cây nhị phân tìm kiếm với mục đích thực hiện các tác vụ tìm kiếm cho nhanh, tuy nhiên để tìm kiếm trên cây nhanh thì cây cần phải cân đối.



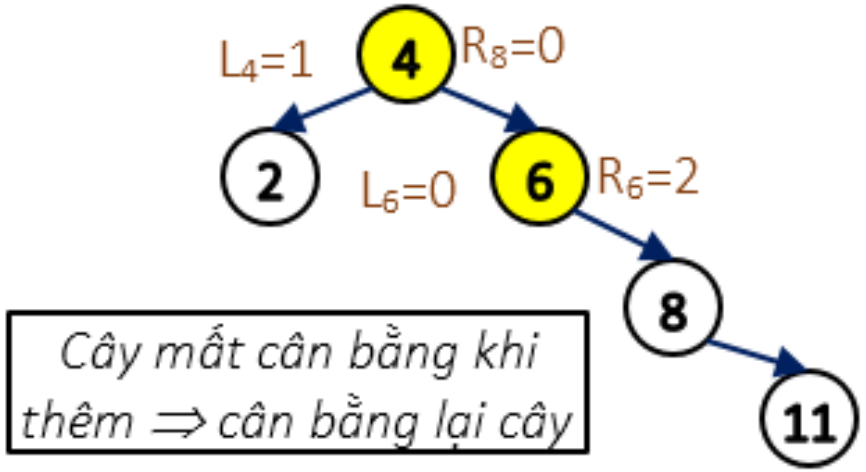


1.1. Cây nhị phân tìm kiếm hoàn toàn cân bằng

- Cây nhị phân tìm kiếm hoàn toàn cân bằng là cây có sự khác biệt của tổng số node cây con bên trái và tổng số node của cây con bên phải không quá 1.
- Trường hợp tối ưu nhất là cây nhị phân tìm kiếm hoàn toàn cân bằng.
- Tuy nhiên khi thêm vào 1 hoặc xóa nút trên cây này rất dễ làm cây mất cân bằng, và chi phí để cân bằng lại cây rất lớn vì phải thao tác trên toàn bộ cây.



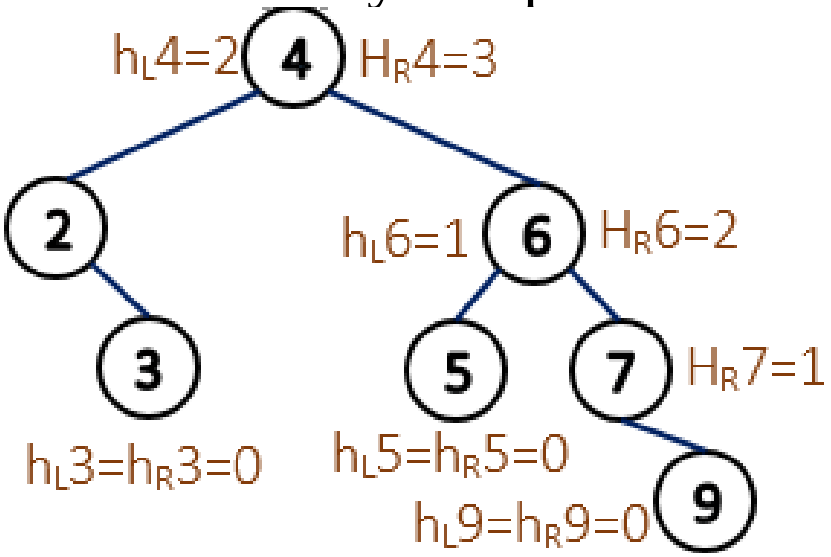
Cây NPTK hoàn toàn cân bằng



Cây mất cân bằng khi thêm ⇒ cân bằng lại cây

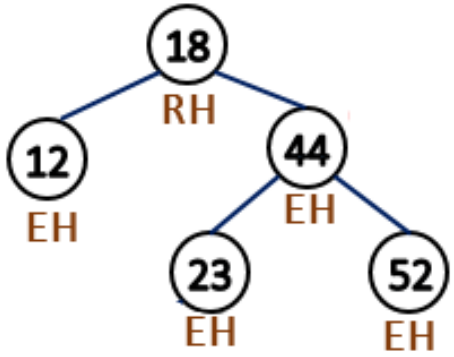
1.2. Cây nhị phân tìm kiếm hoàn toàn cân bằng

- Do có thể phải mất nhiều chi phí (thời gian) để cân bằng lại cây nhị phân tìm kiếm hoàn toàn cân bằng ⇒ cần tìm một cây nhị phân tìm kiếm đạt trạng thái cân bằng yếu hơn nhằm làm giảm thiểu chi phí cân bằng khi thêm nút hay xóa nút.
- Một dạng cây cân bằng là cây nhị phân tìm kiếm cân bằng do hai nhà toán học người Nga là *Adelson Velski* và *Landos* xây dựng vào năm 1962 nên còn được gọi là cây AVL. Tại mỗi nút trong cây này, độ cao của cây con trái và cây con phải chênh lệch không quá 1.

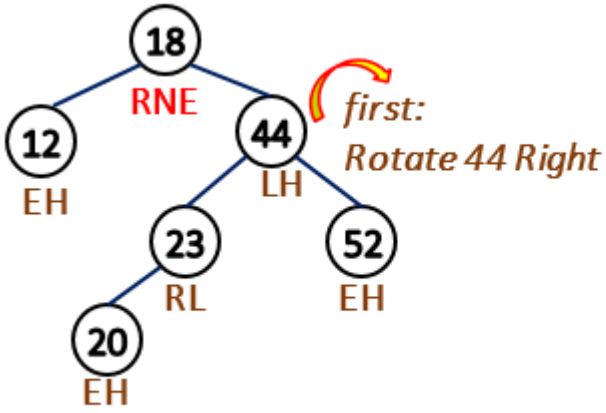


1.2. Cây nhị phân tìm kiếm hoàn toàn cân bằng

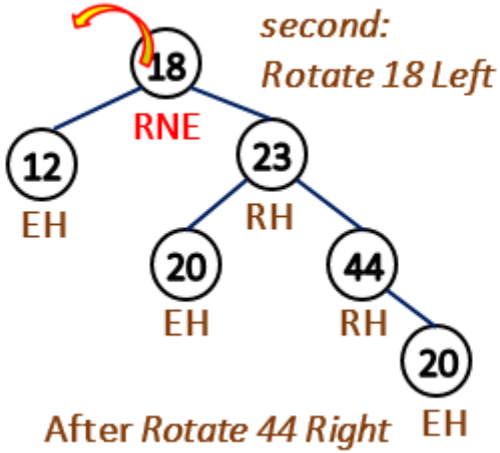
- Các thao tác trên cây AVL tương tự như BST
- Khác biệt: khi thêm/xoá sẽ làm mất cân bằng: sử dụng thao tác xoay để cân bằng lại cây



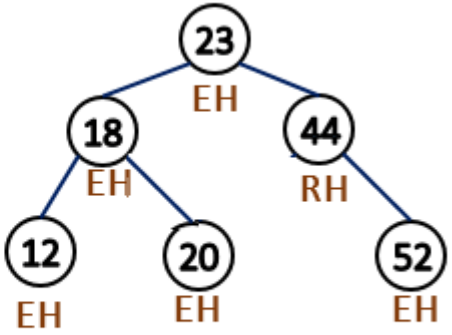
AVL tree ban đầu



AVL tree sau khi thêm 20



After *Rotate 44 Right*



After *Rotate 18 Left*

1.3. Chỉ số cân bằng (Balance factor – bf)

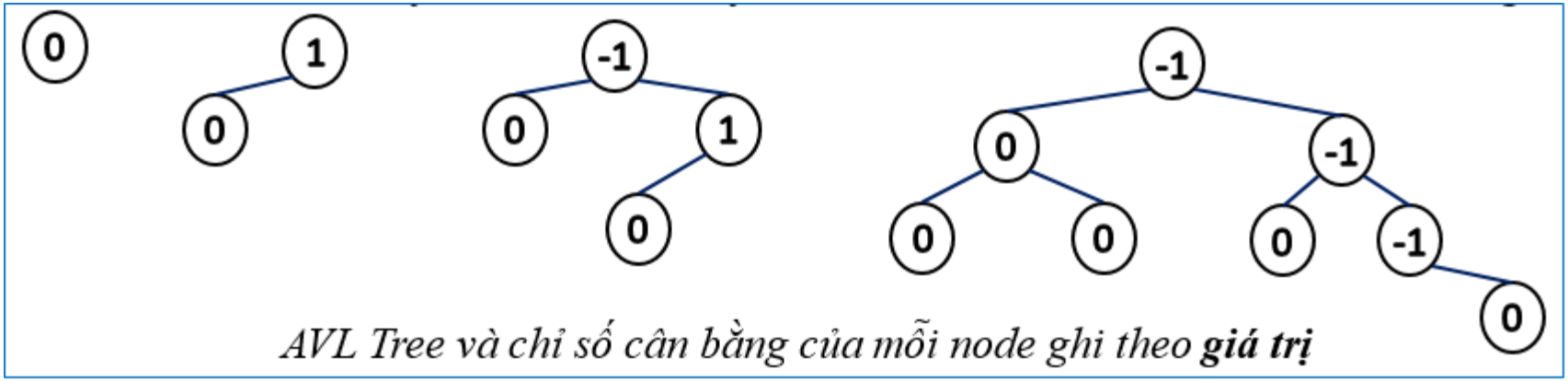
- ***Chiều cao của cây con***: gọi lh(p) và rh(p) là chiều cao của cây con bên trái và chiều cao của cây con bên phải của nút p.
- ***Chỉ số cân bằng (bf) của 1 node trên cây AVL***: là hiệu của chiều cao cây con bên trái và chiều cao của cây con bên phải.
- Các giá trị có thể có của chỉ số cân bằng

bf(p) = (h_L-h_R)				
2	1	0	-1	-2
LNE *(Left Not Equal)*	**LH** *(Left Higher)*	**EH** *(equal height)*	**RH** *(Right Higher)*	**RNE** *(Right Not Equal)*
Node mất cân bằng bên trái	Node lệch trái	Node cân bằng	Node lệch phải	Node mất cân bằng bên phải
Cây mất cân bằng	Cây cân bằng			Cây mất cân bằng

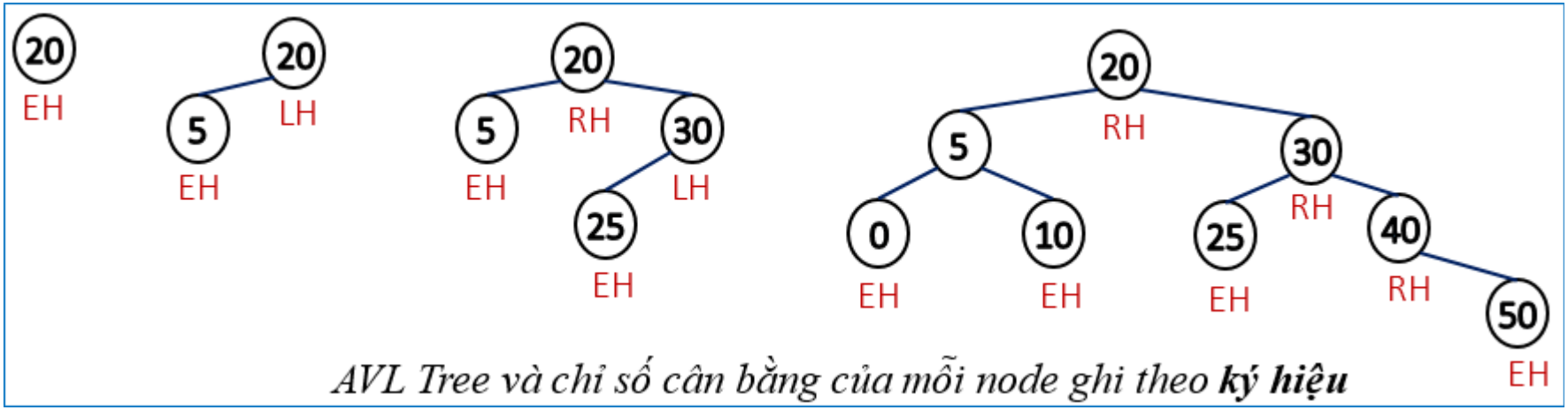
1.3. Chỉ số cân bằng (Balance factor – bf)

- ***Cây nhị phân tìm kiếm cân bằng AVL***: chỉ số cân bằng của mọi node trên cây nằm trong khoảng từ -1 đến +1.

  Ví dụ: Hình vẽ sau đây minh họa các cây AVL, mỗi nút có một số là chỉ số cân bằng đó.



*AVL Tree và chỉ số cân bằng của mỗi node ghi theo **giá trị***

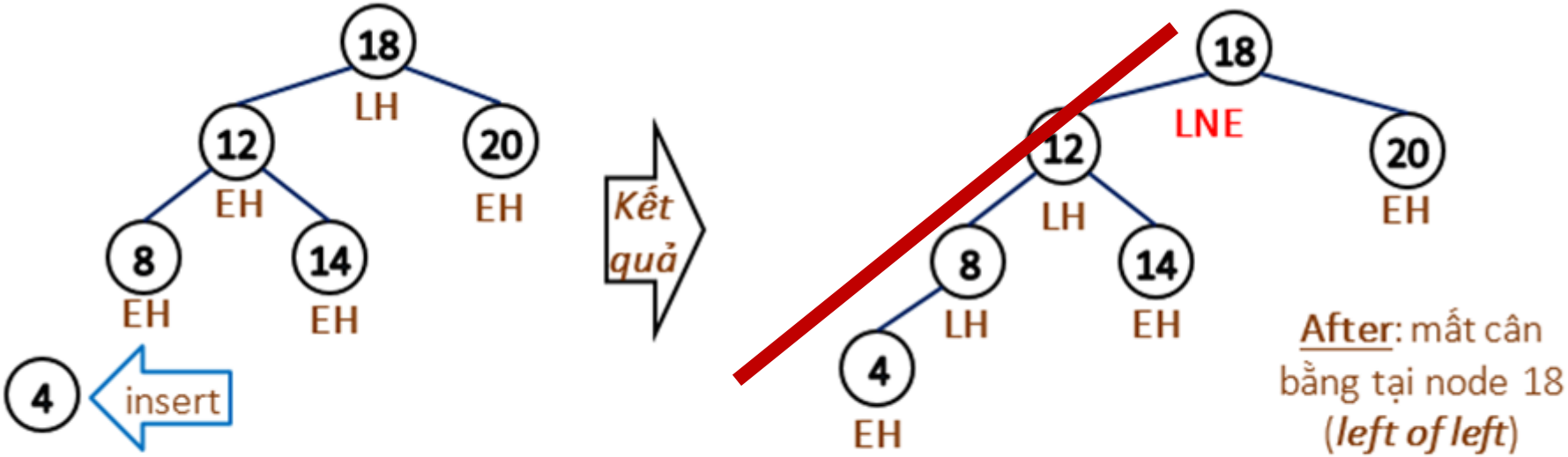


*AVL Tree và chỉ số cân bằng của mỗi node ghi theo **ký hiệu***

2. CÁC DẠNG MẤT CÂN BẰNG

2.1. Cây con trái lệch trái (*Left of Left*)

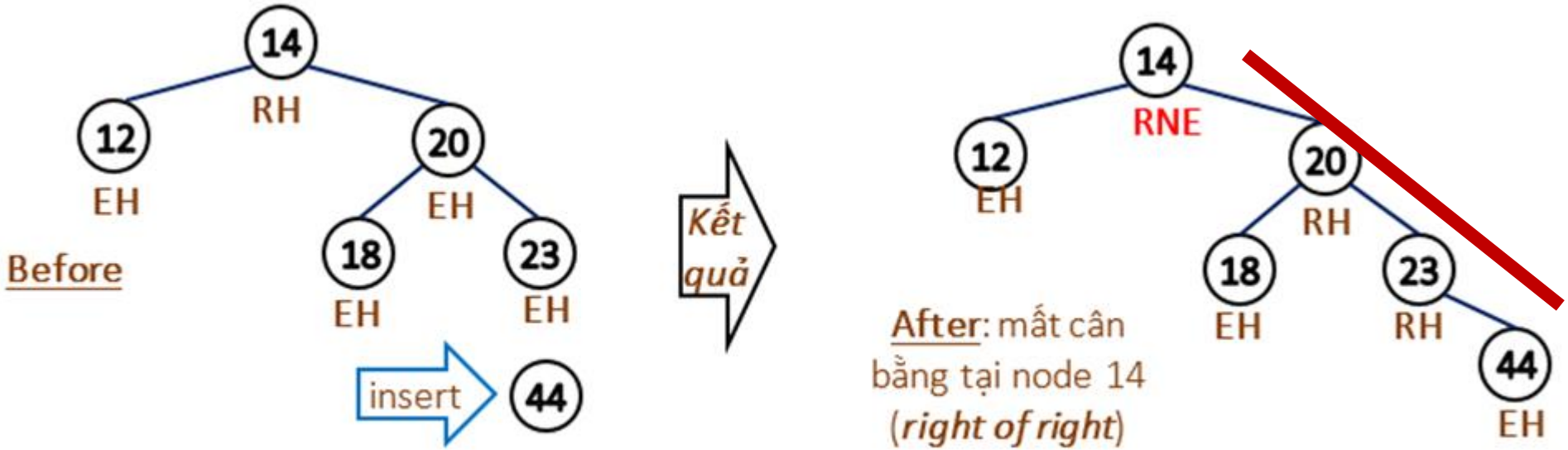
- Minh họa



Cây con trái lệch trái

2.2. Cây con phải lệch phải (*Right of Right*)

- Minh họa



Cây con phải lệch phải

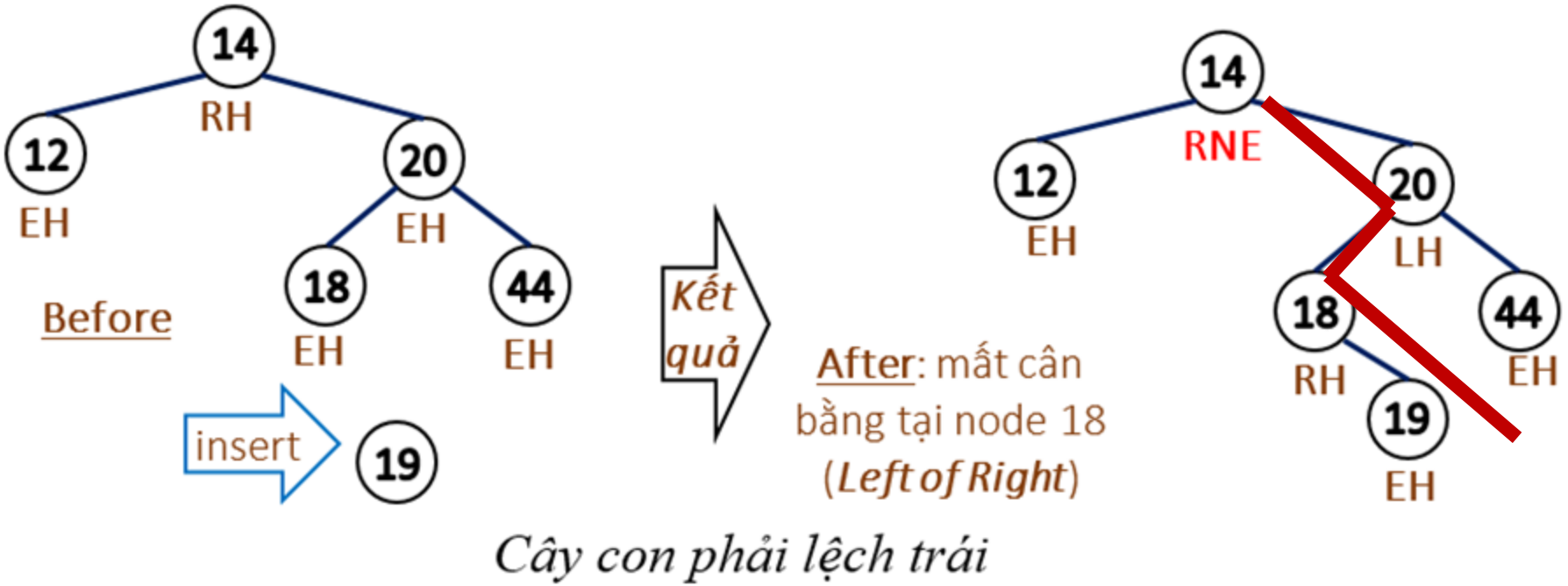
2.3. Cây con trái lệch phải (*Right of Left*)

- Minh họa

Cây con trái lệch phải

2.4. Cây con phải lệch trái (*Left of Right*)

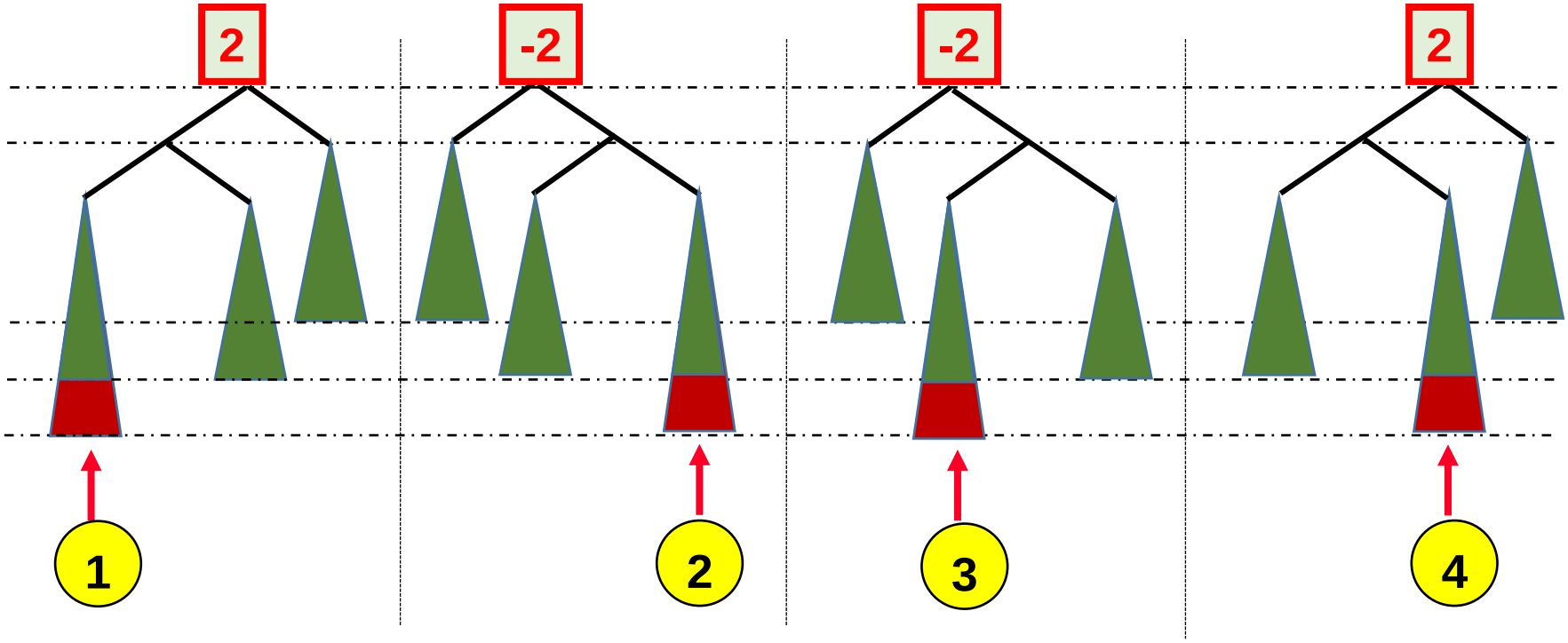
- Minh họa



Cây con phải lệch trái

Nhận xét

Tổng quát về các trường hợp mất cân bằng



Trong đó:

- ① (*Left of Left*) và ② (*Right of Right*) là các ảnh đối xứng
- ④ (*Right of Left*) và ③ (*Left of Right*) là các ảnh đối xứng

3. CÁC TÁC VỤ XOAY

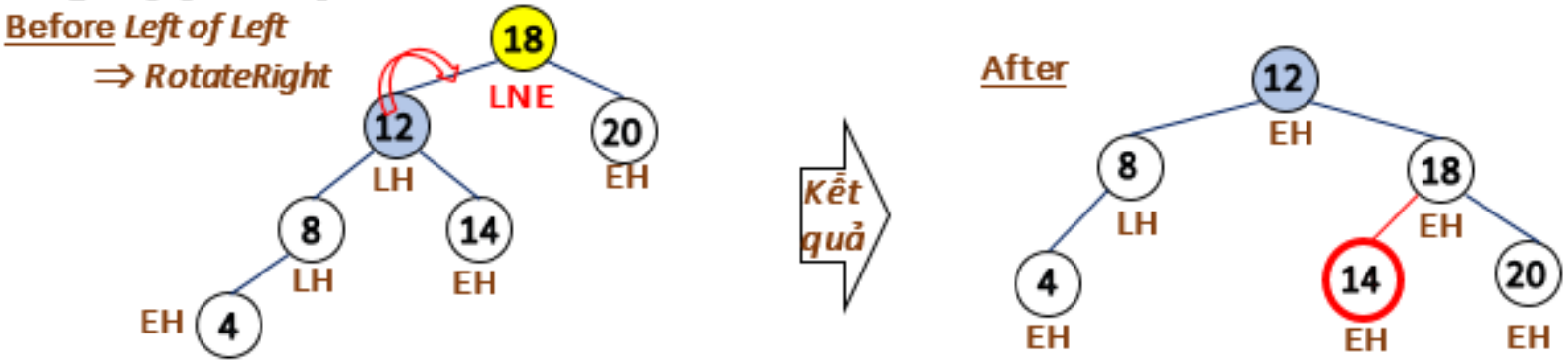
3.1. Tác vụ xoay đơn (*single rotation*)

3.1.1. Tác vụ xoay phải *(RotateRight)* khi cây mất cân bằng dạng *LeftOfLeft*

- Minh họa
  - Trường hợp đơn giản

Trường hợp đơn giản của tác vụ xoay đơn - xoay phải

  - Trường hợp phức tạp

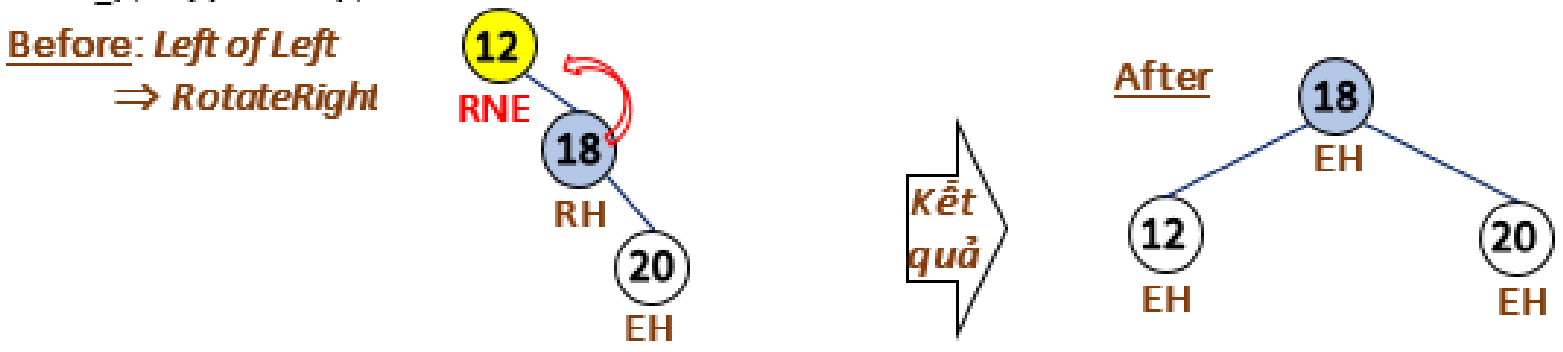


Trường hợp phức tạp của tác vụ xoay đơn – xoay phải

3.1. Tác vụ xoay đơn (*single rotation*)

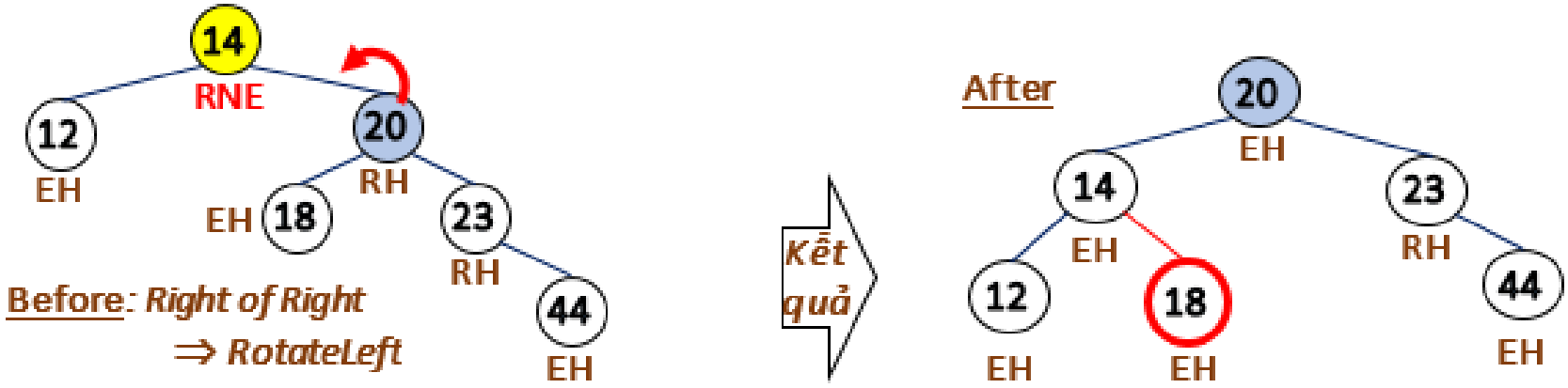
3.1.2. Tác vụ xoay trái (*RotateLeft*) khi cây mất cân bằng dạng *RightOfRight*

- Minh họa
  - Trường hợp đơn giản



Trường hợp đơn giản của tác vụ xoay đơn – xoay trái

  - Trường hợp phức tạp

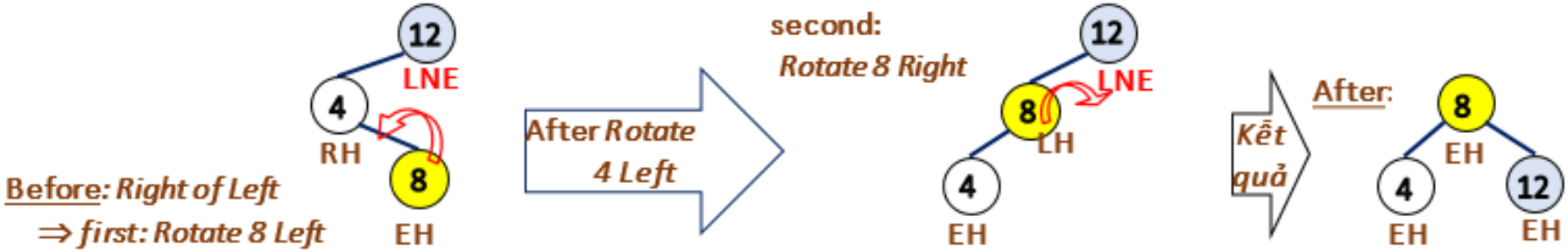


Trường hợp phức tạp của tác vụ xoay đơn – xoay trái

3.2. Tác vụ xoay kép *(double rotation)*

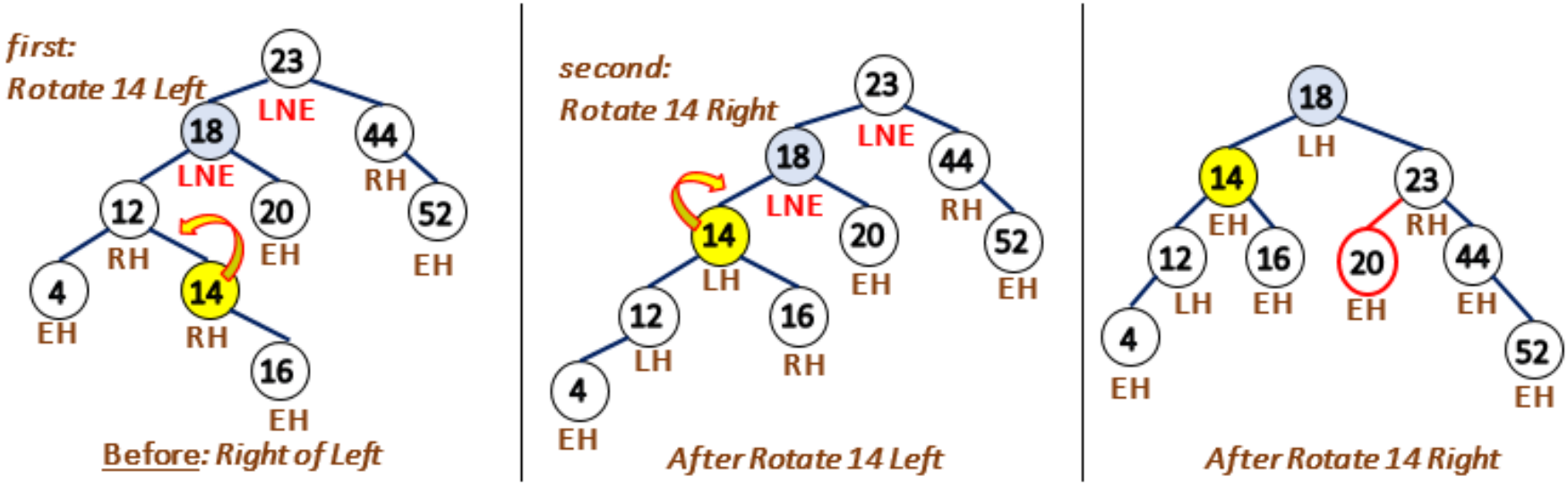
3.2.1. Tác vụ xoay trái *(RotateLeft)* khi cây con trái lệch phải *(Right of Left)*

- Minh họa
  - Trường hợp đơn giản



Trường hợp đơn giản của tác vụ xoay kép – xoay trái khi cây con trái lệch phải

  - Trường hợp phức tạp

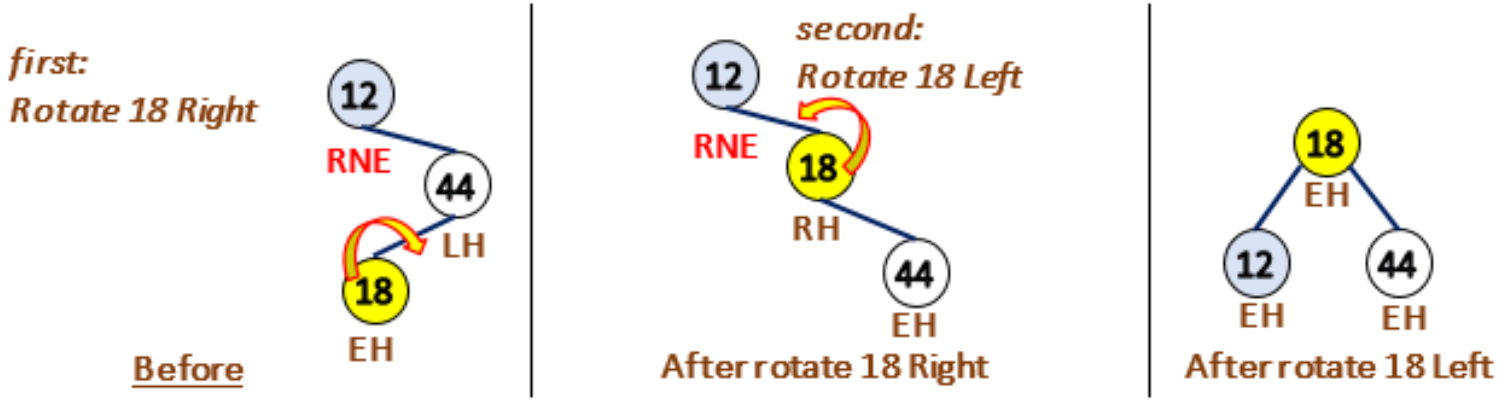


Trường hợp phức tạp của tác vụ xoay kép – xoay trái khi cây con trái lệch phải

3.2. Tác vụ xoay kép *(double rotation)*

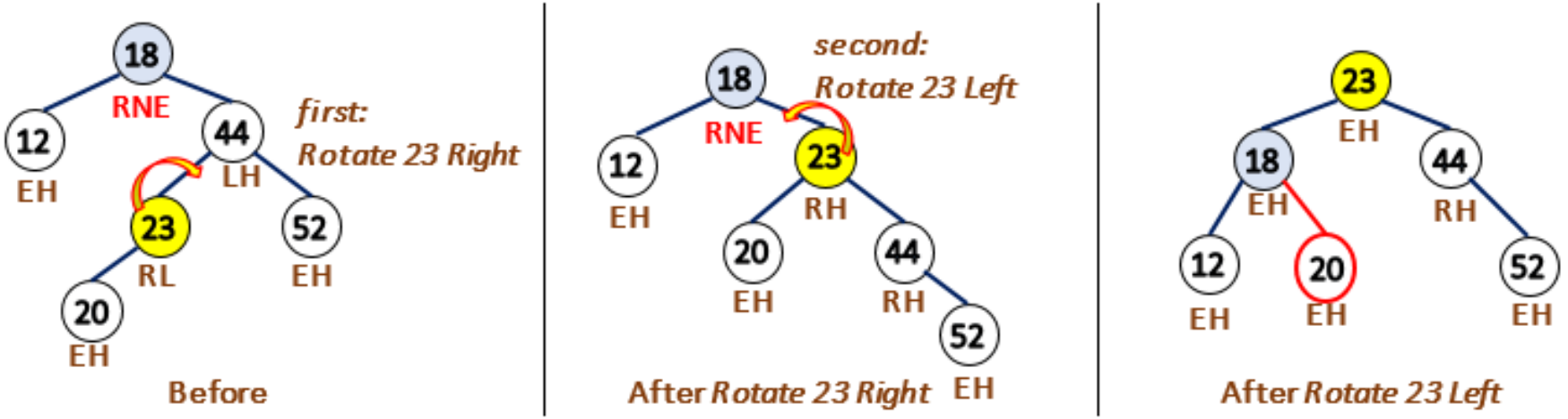
3.2.2. Tác vụ xoay phải *(RotateRight)* khi cây con phải lệch trái *(Left of Right)* - Minh họa

- Trường hợp đơn giản



Trường hợp đơn giản của tác vụ xoay kép – xoay phải khi cây con phải lệch trái

- Trường hợp phức tạp



Trường hợp phức tạp của tác vụ xoay kép – xoay phải khi cây con phải lệch trái

4. THÊM VÀ XÓA TRÊN CÂY CÂN BẰNG

4.1. Thêm

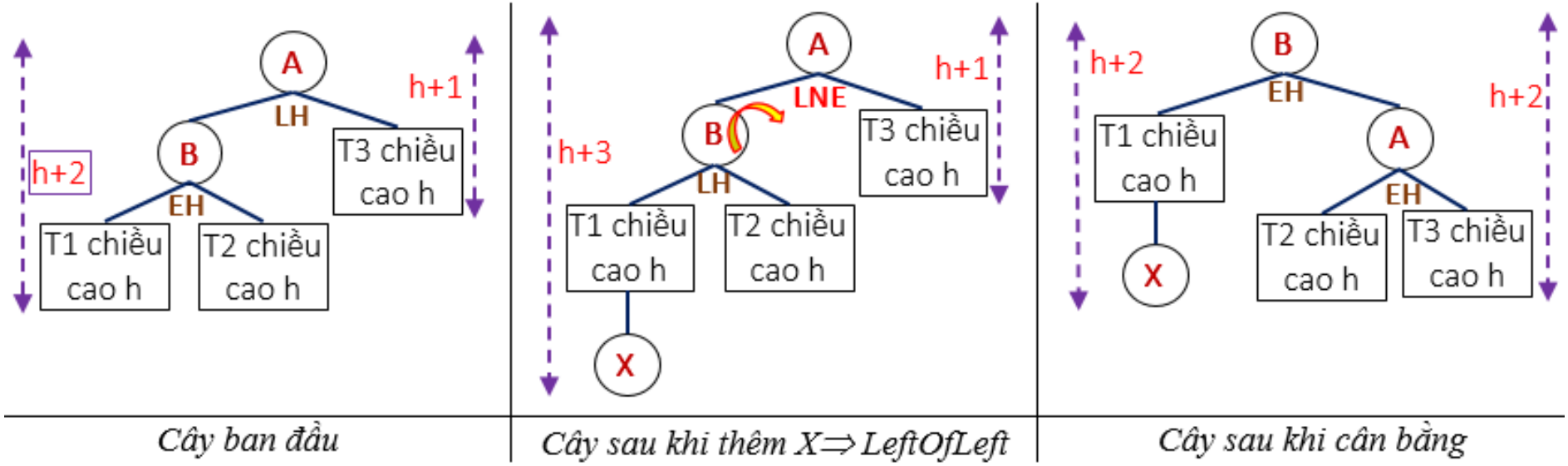
4.1.1. Các bước thực hiện khi thêm trên AVL tree

- ***B1***: Thêm nút vào cây AVL như thêm nút vào cây nhị phân tìm kiếm, nghĩa là nút mới thêm vào sẽ là nút lá ở vị trí thích hợp trên cây.
- ***B2***: Tính lại chỉ số cân bằng của các nút có bị ảnh hưởng.
- ***B3***: Nếu cây bị mất cân bằng, xác định dạng mất cân bằng của cây để chọn phép xoay cho phù hợp.

4.2. Các trường hợp thêm vào làm cây mất cân bằng

4.2.1. Trường hợp thêm dẫn đến cây con trái lệch trái

(Left of Left)

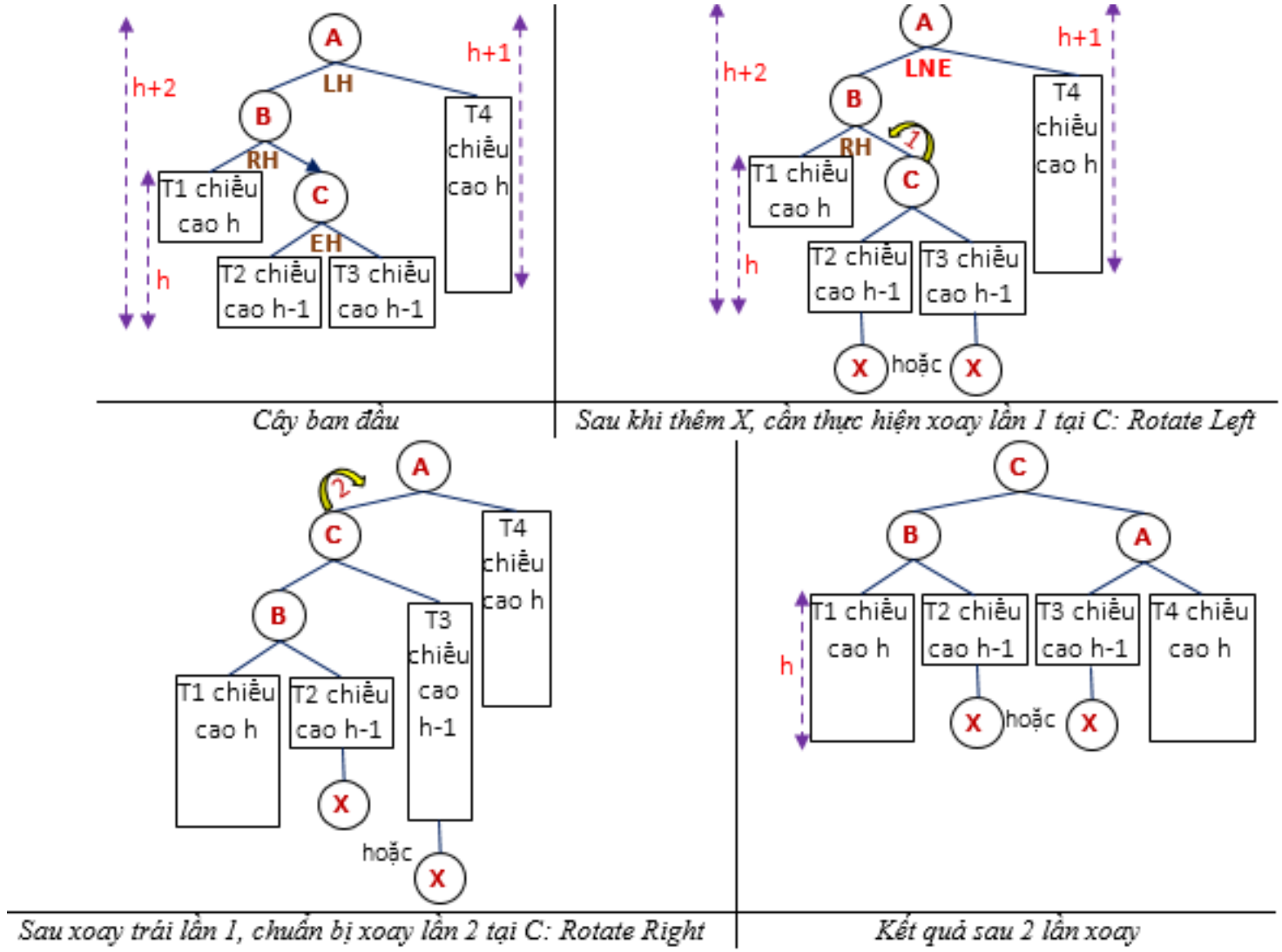


Cây ban đầu | *Cây sau khi thêm X⇒ LeftOfLeft* | *Cây sau khi cân bằng*

Minh họa trường hợp dẫn đến cây con trái lệch trái và việc thực hiện cân bằng

4.2. Các trường hợp thêm vào làm cây mất cân bằng

4.2.2. Trường hợp thêm dẫn đến cây con trái lệch phải *(Left of Right)*

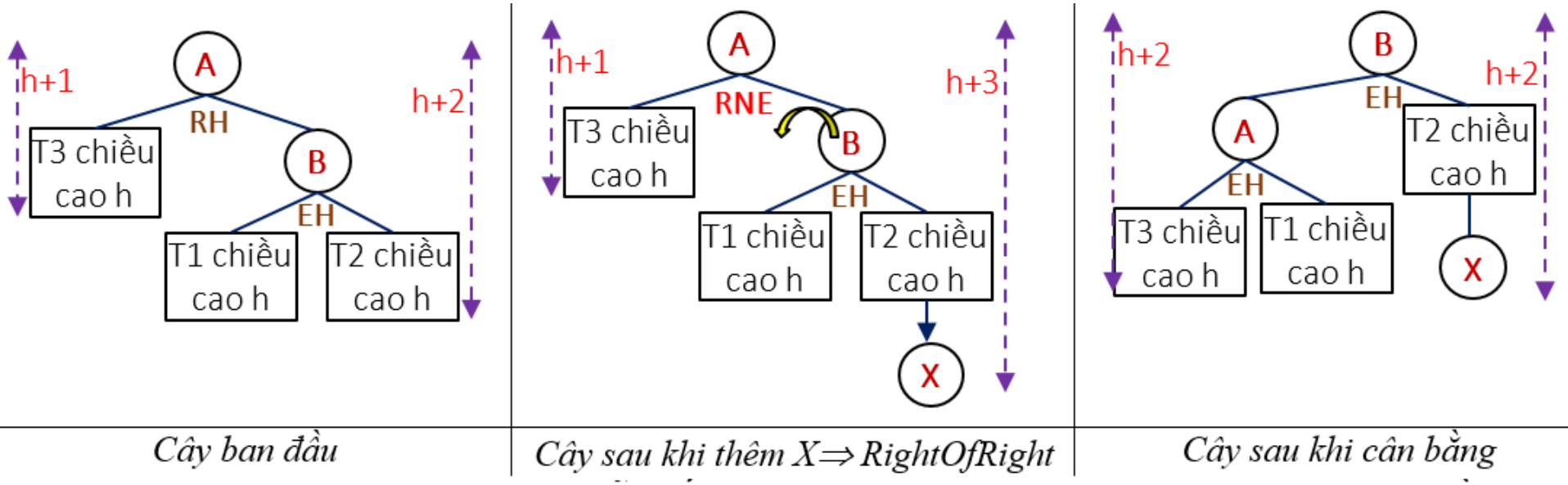


Minh họa trường hợp dẫn đến cây con trái lệch phải và việc thực hiện cân bằng

4.2. Các trường hợp thêm vào làm cây mất cân bằng

4.2.3. Trường hợp thêm dẫn đến cây con phải lệch phải

(Right of Right)



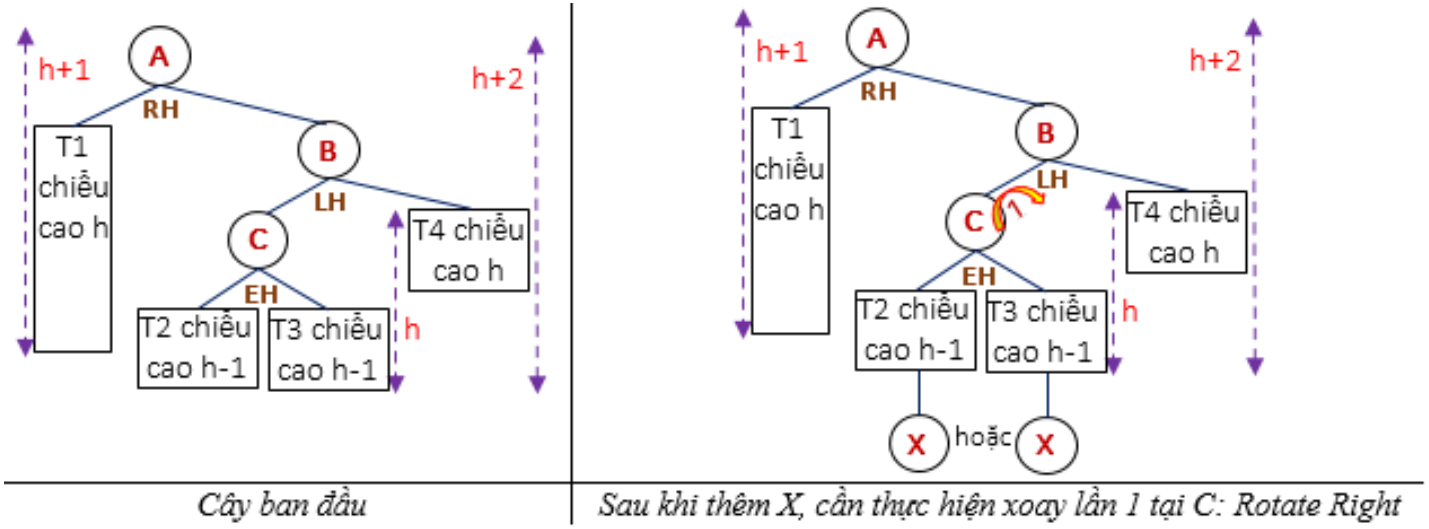
Cây ban đầu | *Cây sau khi thêm X⇒ RightOfRight* | *Cây sau khi cân bằng*

Minh họa trường hợp dẫn đến cây con phải lệch phải và việc thực hiện cân bằng

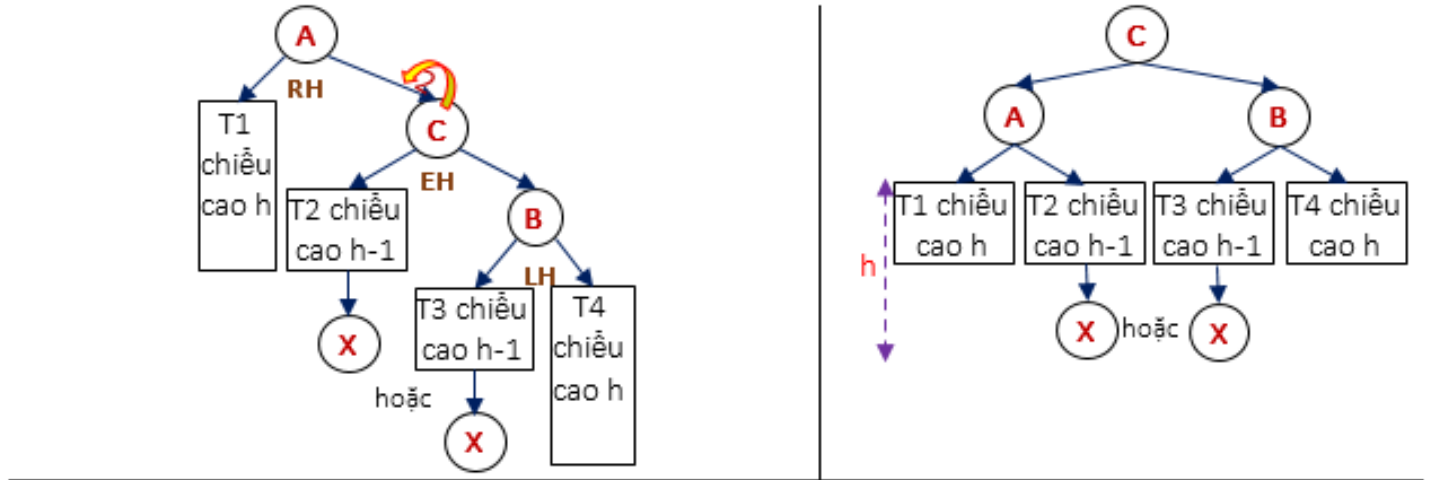
4.2. Các trường hợp thêm vào làm cây mất cân bằng

4.2.4. Trường hợp thêm dẫn đến cây con phải lệch trái

(Right of Left)



Cây ban đầu | *Sau khi thêm X, cần thực hiện xoay lần 1 tại C: Rotate Right*

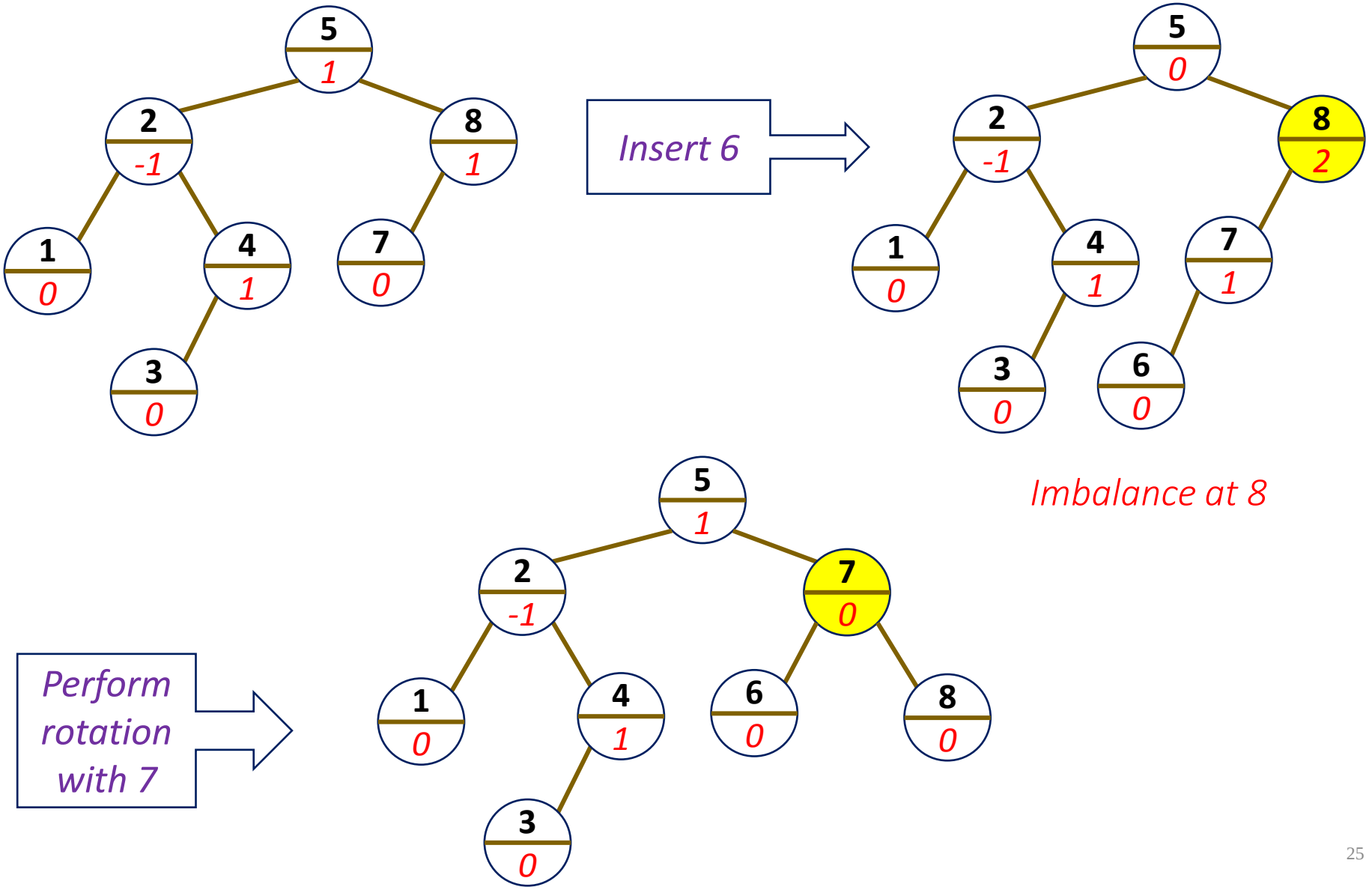


Sau khi xoay trái lần 1, chuẩn bị xoay lần 2: Rotate Left | *Kết quả sau 2 lần xoay*

Minh họa trường hợp dẫn đến cây con phải lệch trái và việc thực hiện cân bằng

4.2. Các trường hợp thêm vào làm cây mất cân bằng

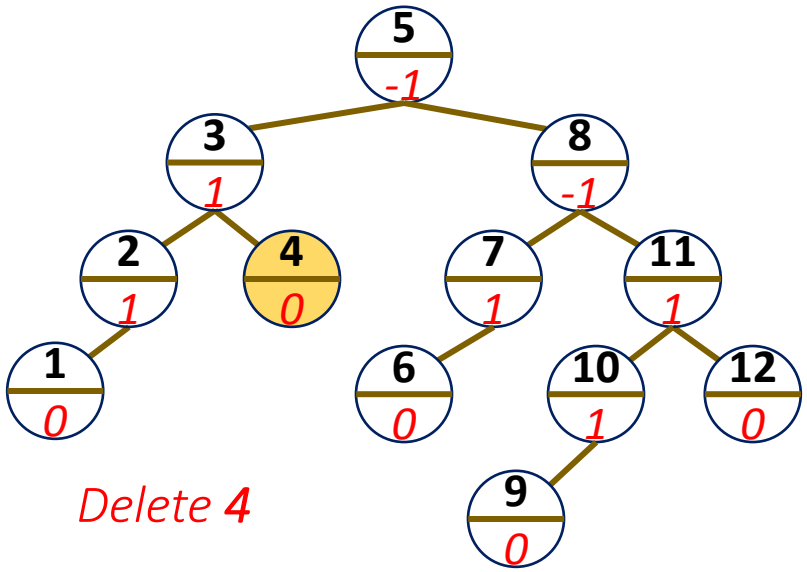
Thực hành 1: Thêm node có giá trị =6 vào cây sau:



4.3. Xóa trên AVL tree

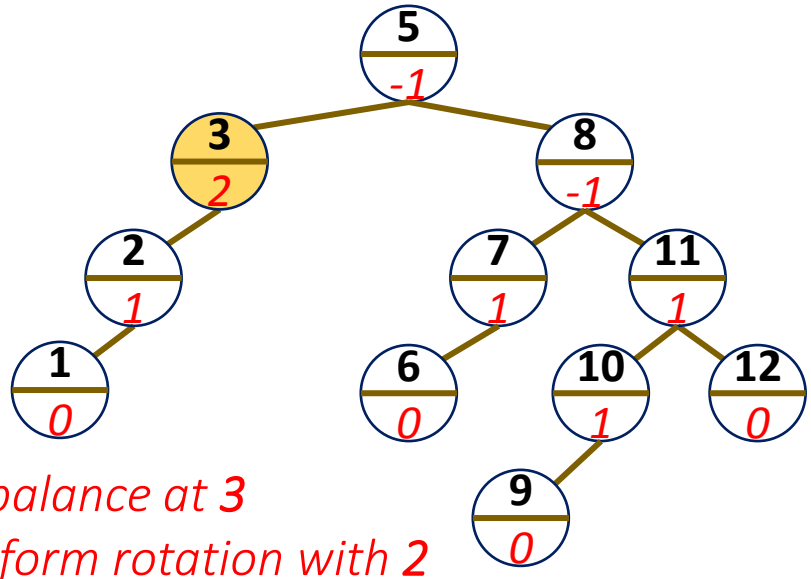
- Tương tự như trong cây nhị phân tìm kiếm (BST), để xóa một node cũng được chia làm ba trường hợp:
  - DelNode là node lá,
  - DelNode là node trung gian có 01 cây con,
  - DelNode là node có đủ 02 cây con.
- Trong trường hợp DelNode có đủ 02 cây con, sử dụng phương pháp xóa phần tử thế mạng vì theo phương pháp này sẽ làm cho chiều cao của cây ít biến động.
- Sau khi xóa, 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 xóa sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền.

4.3. Xóa trên AVL tree



Delete **4**

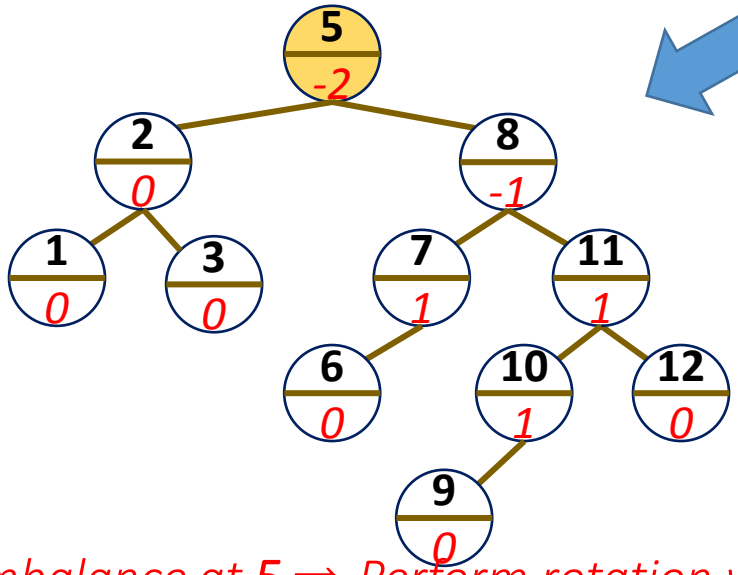
①



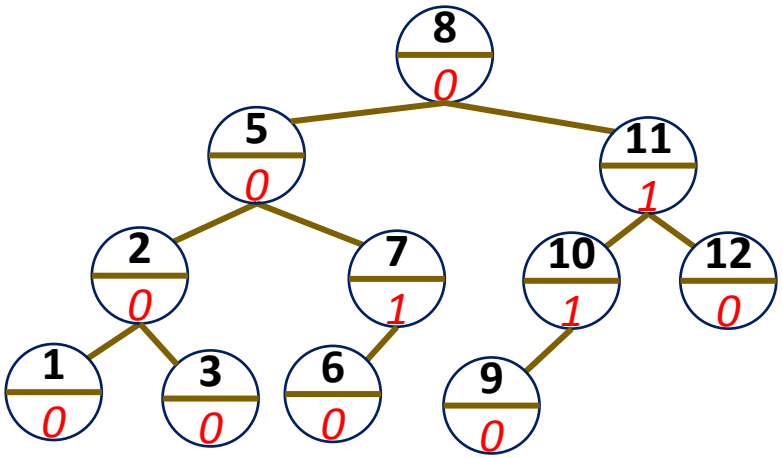
Imbalance at **3**

Perform rotation with **2**

②



③



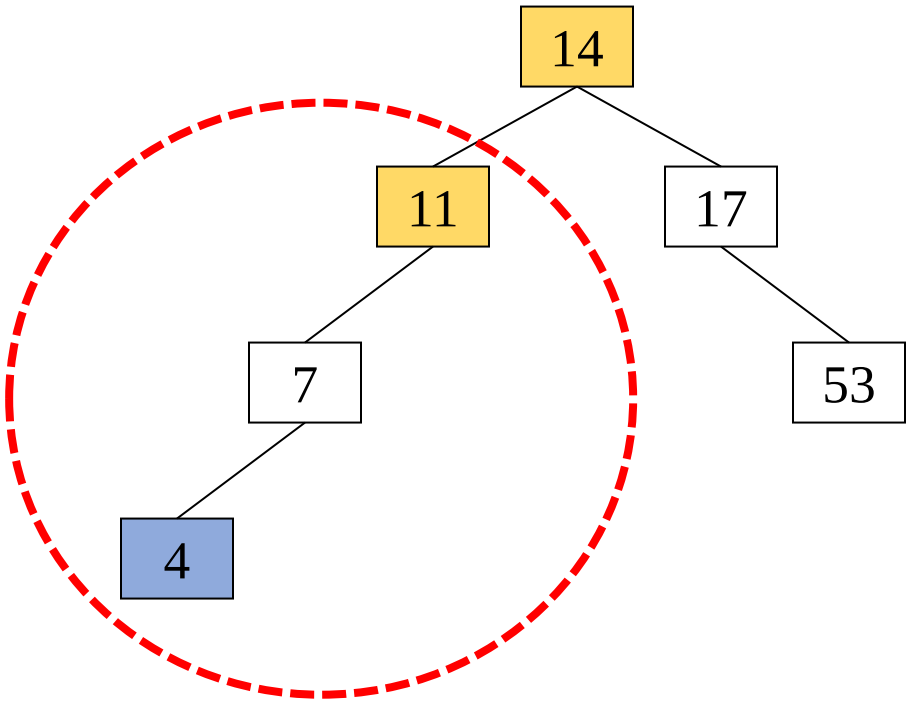
Imbalance at **5** ⇒ Perform rotation with **8**

4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ***14, 17, 11, 7, 53, 4, 13, 12, 8***.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái (*Replace it with the **rightest in its left branch***)



After insert 4 ⇒ Imbalance at 11

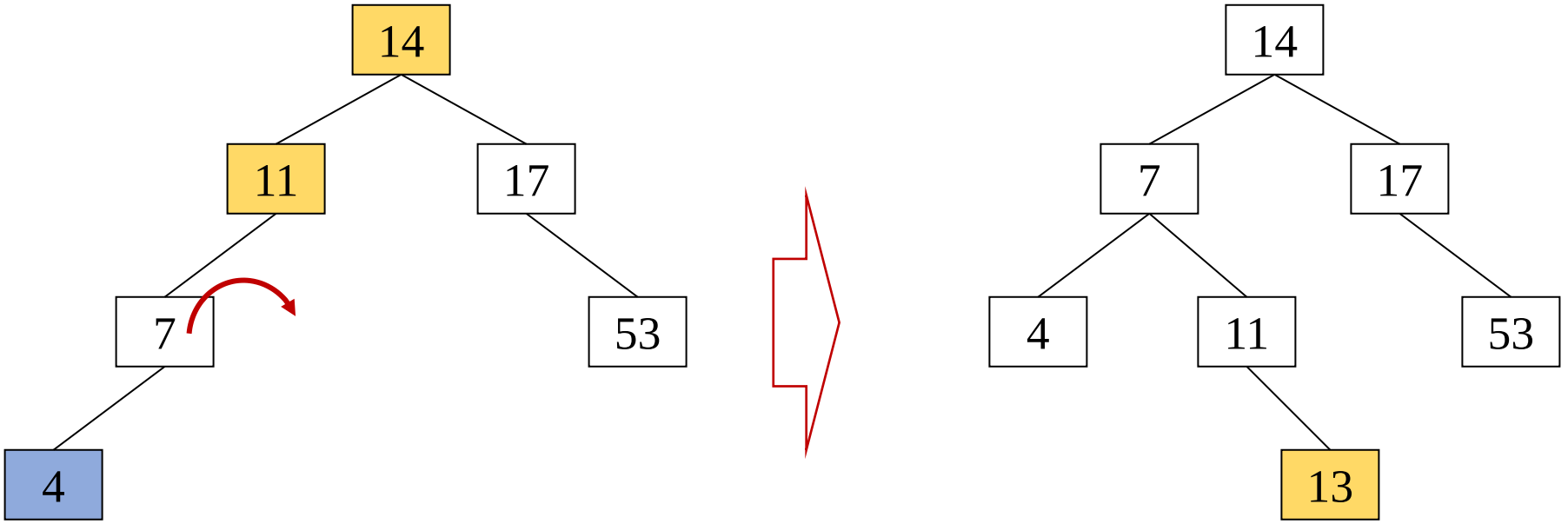
*Perform rotation **right** with 7*

4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ***~~14, 17, 11, 7, 53, 4~~***, ***13***, ***12***, ***8***.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



*After insert 4 ⇒ Imbalance at **11***

*Perform rotation **right** with **7***

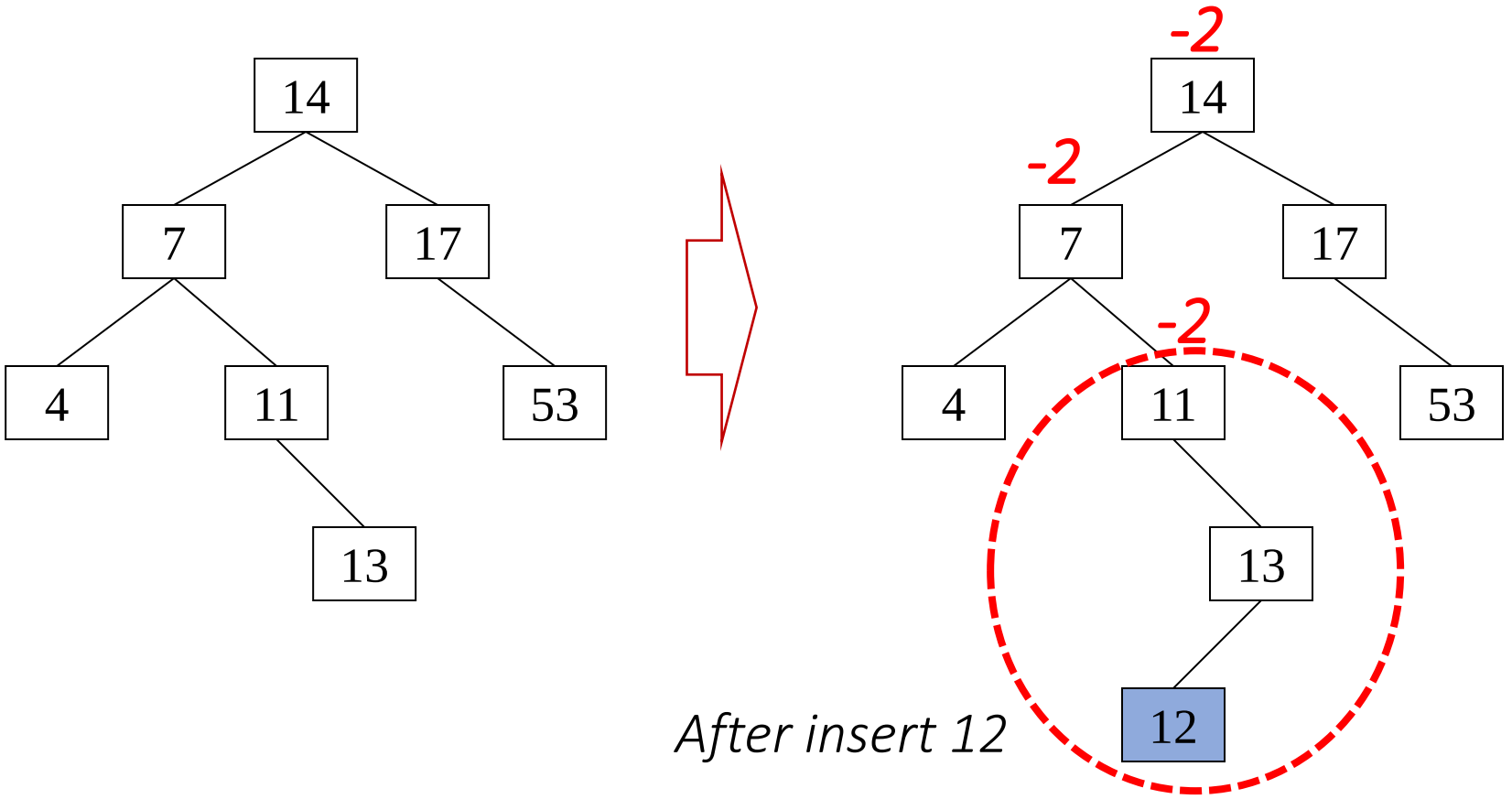
After insert 13

4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13***~~, ***12***, ***8***.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



Imbalance at 11, 7, 14

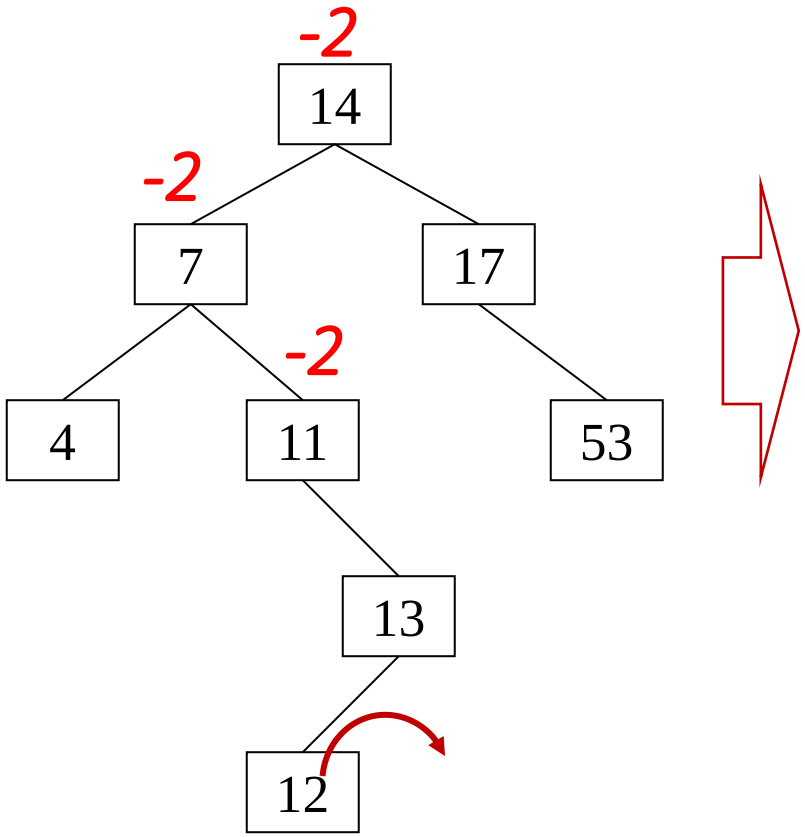
Perform double rotation with 12

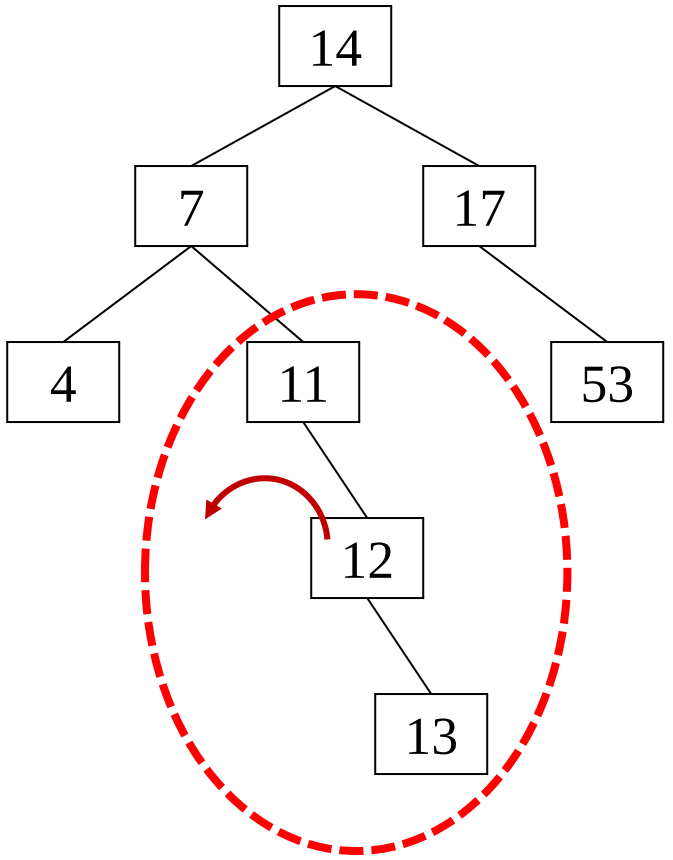
4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13***~~, *12*, ***8***.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái





Imbalance at 11, 7, 14

Perform double rotation with 12

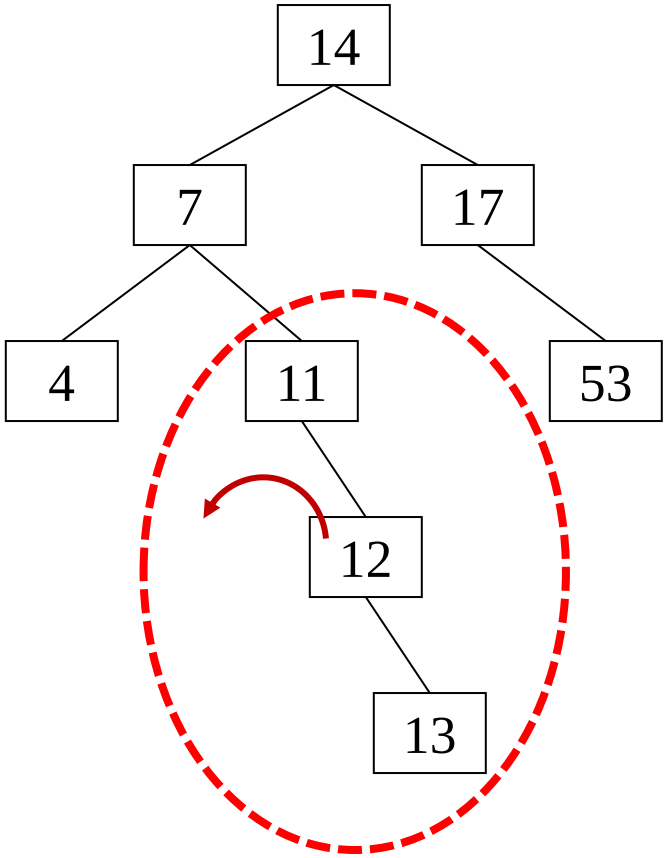
*Next, Perform rotation **left** with 12*

4. Thêm và xóa trên cây cân bằng

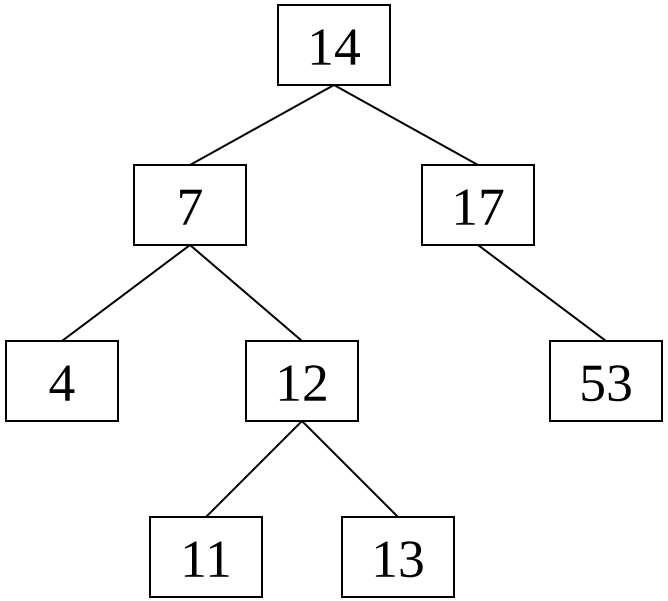
Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12***~~***, 8***.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



*Next, Perform rotation **left** with 12*



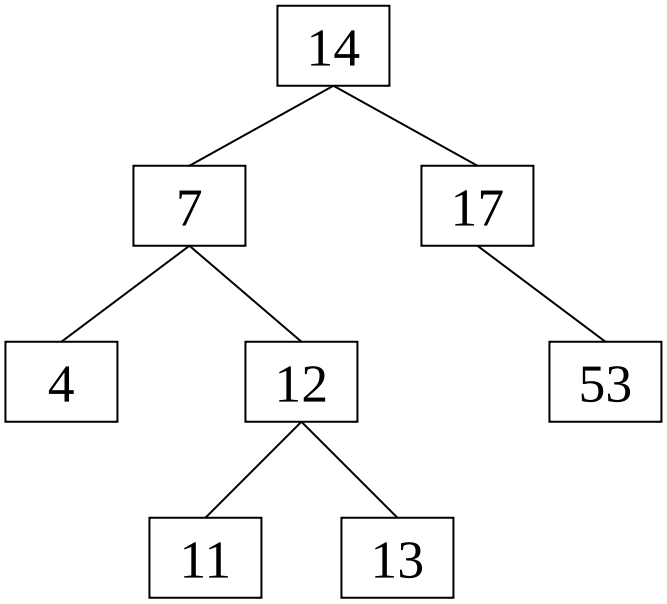
The AVL tree is balanced

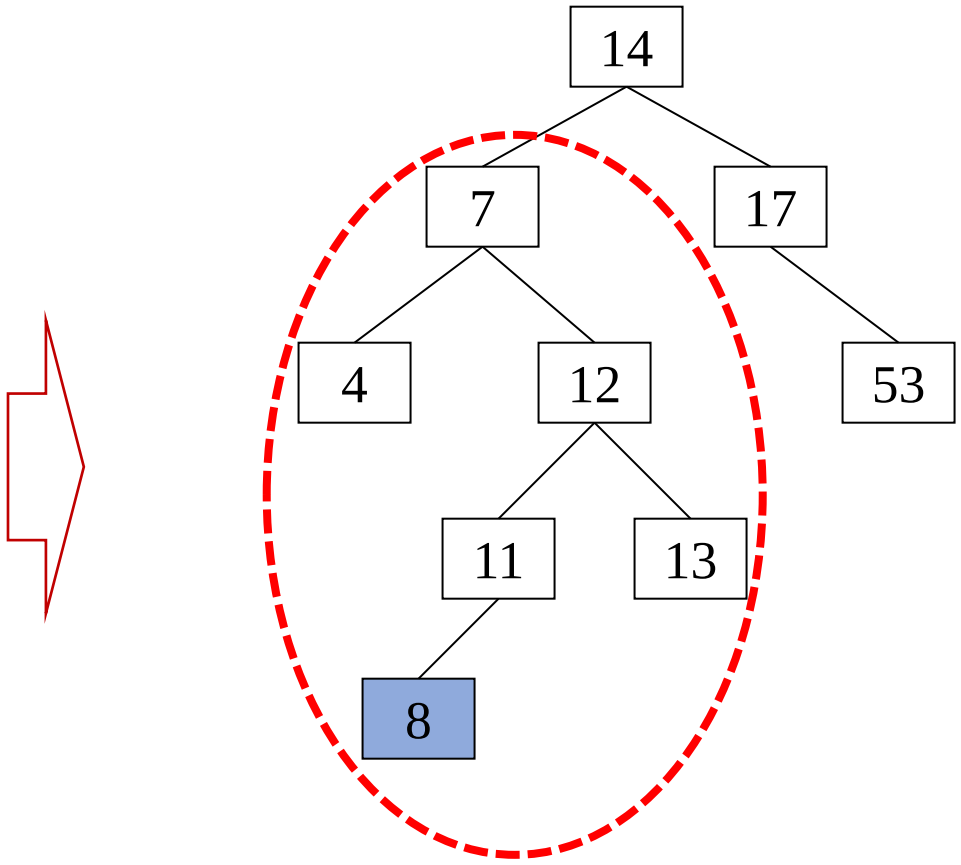
4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12,***~~ ***8***.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái





Imbalance at 7, 14

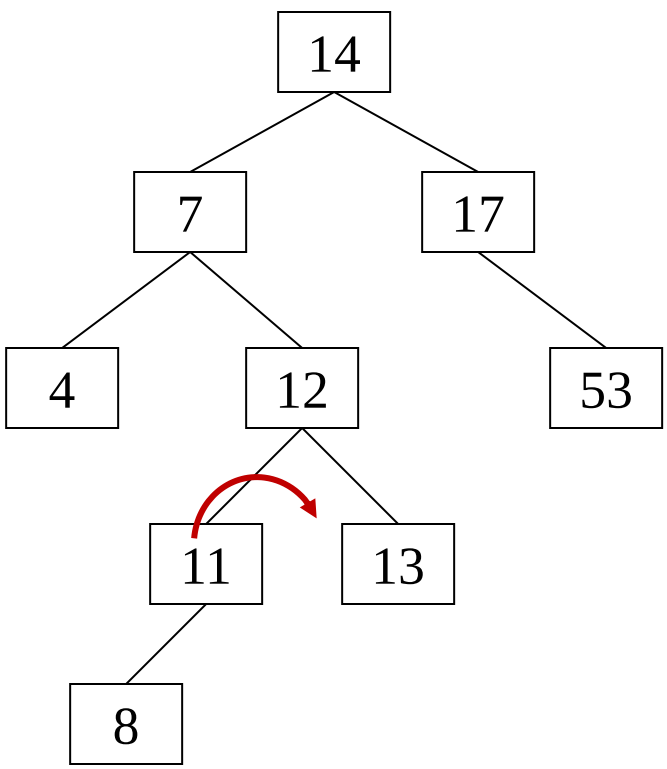
Perform double rotation with 12

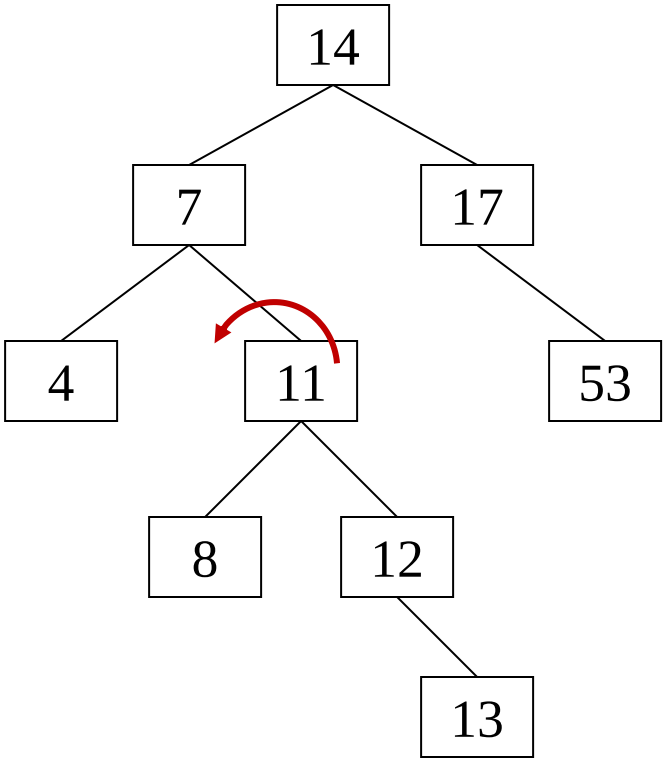
4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái





*Imbalance at **7, 14***

*Perform double rotation with **11***

Perform rotation right with 11

Next,

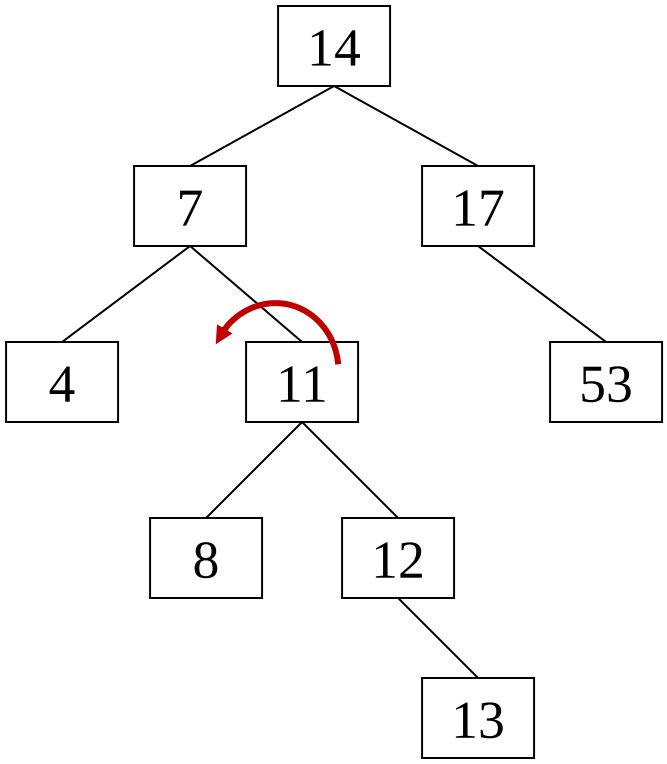
*perform rotation **left** with **11***

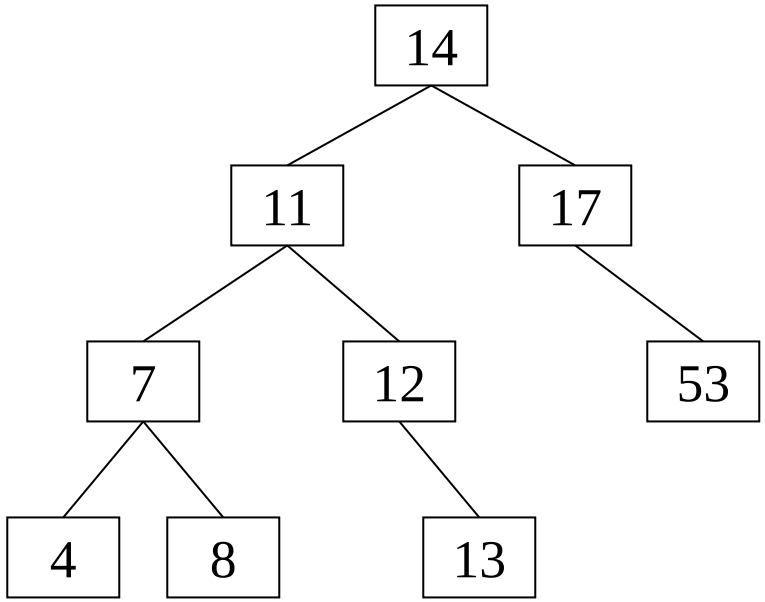
4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái





Next,
*perform rotation **left** with **11***

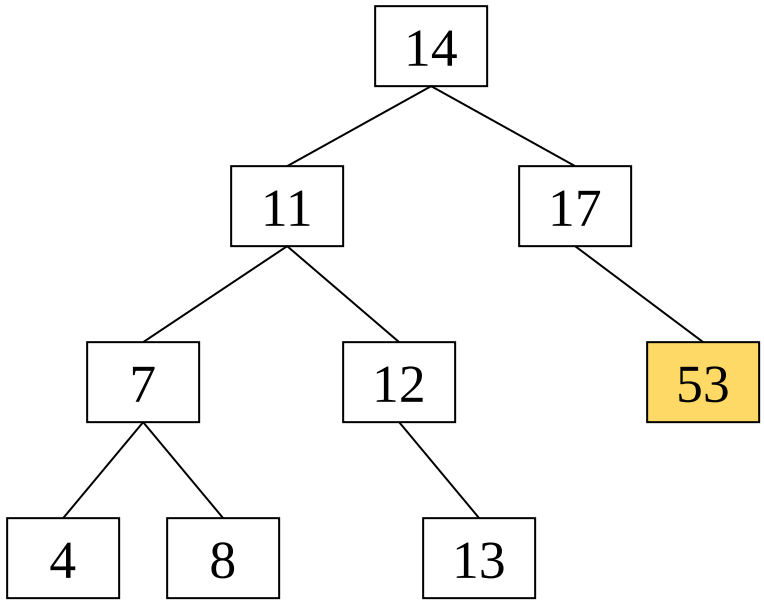
The AVL tree is balanced

4. Thêm và xóa trên cây cân bằng

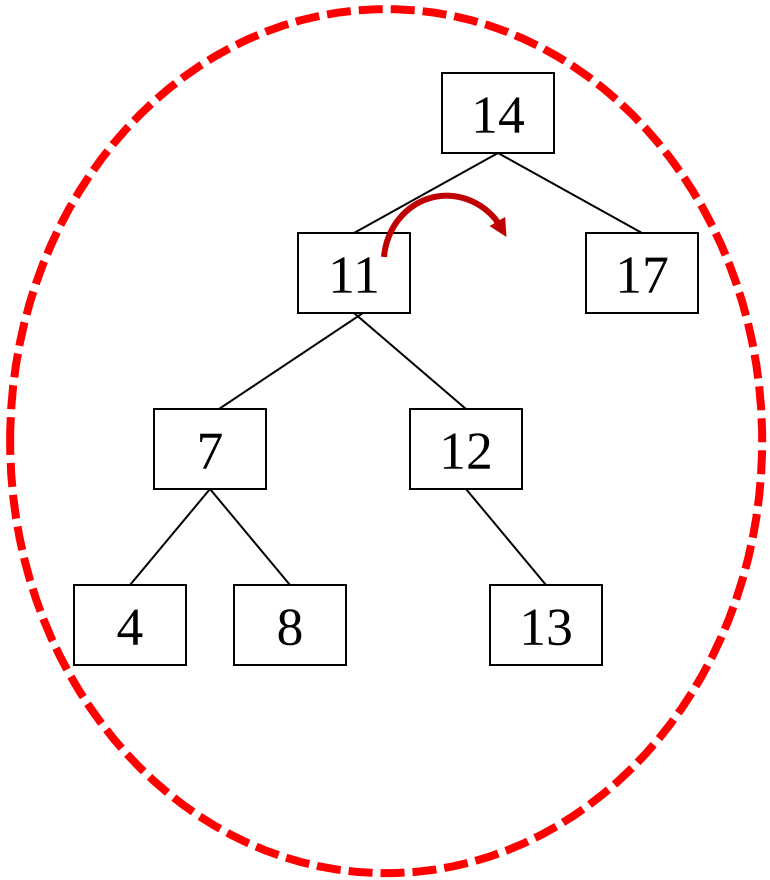
Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



Delete 53



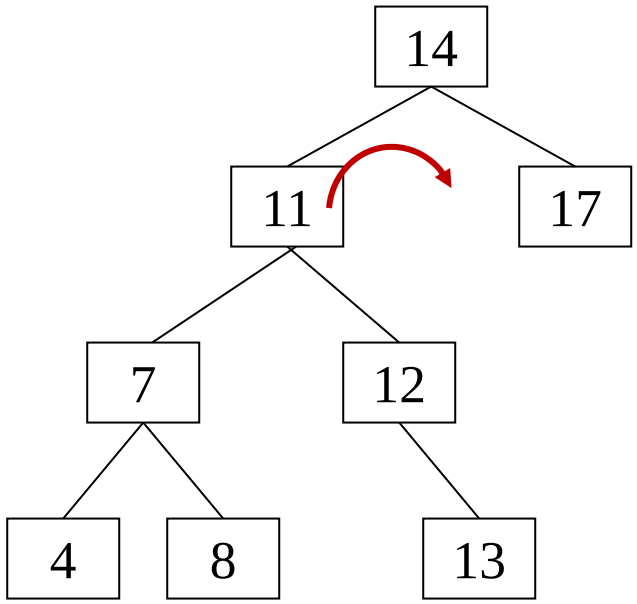
Perform rotation right with 11

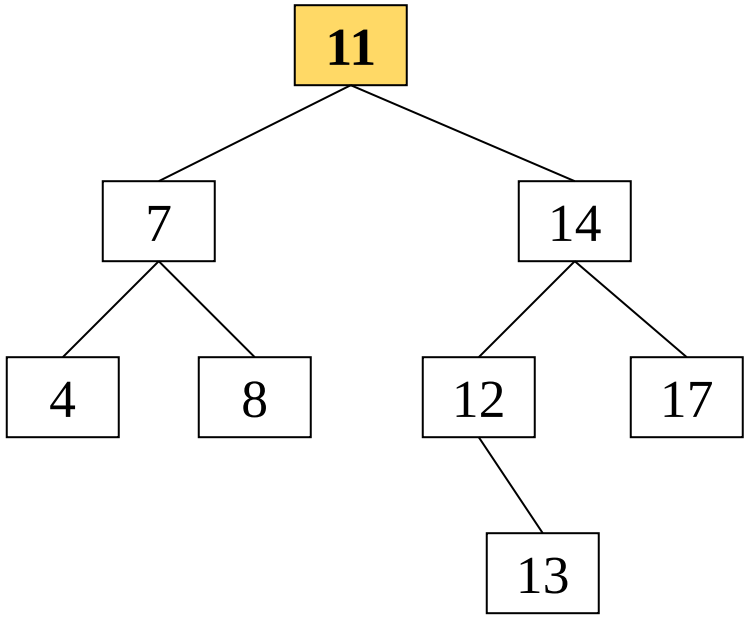
4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ~~***53***~~, ***11***, ***8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái





Remove ***11***

Replace it with the

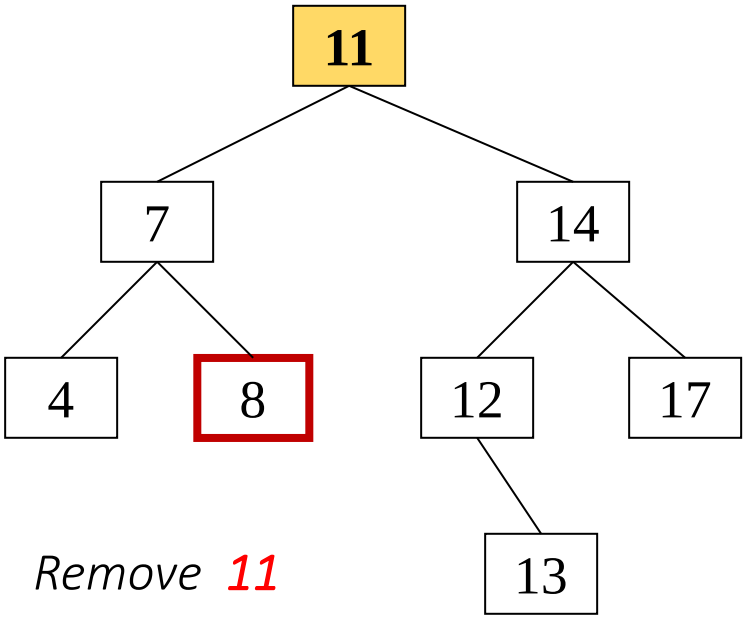
rightest in its left branch

4. Thêm và xóa trên cây cân bằng

Thực hành

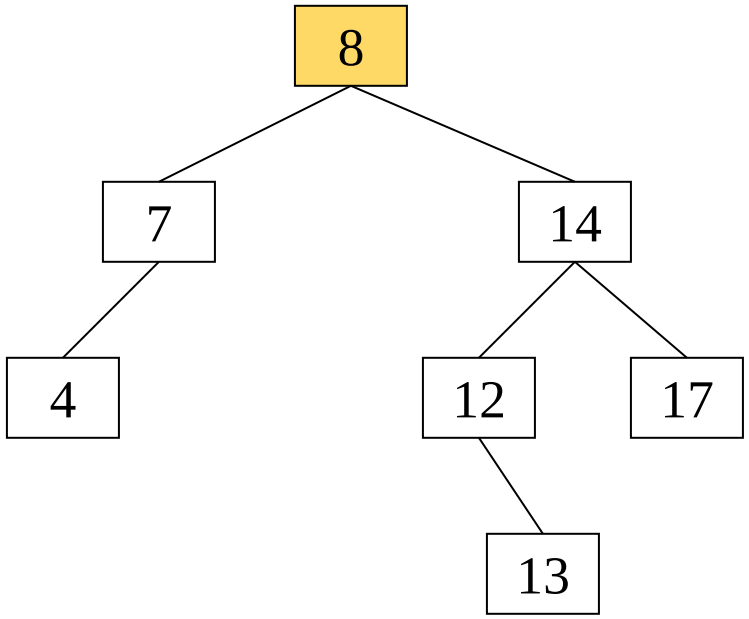
a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ~~***53***~~, ***11***, ***8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



Remove 11

Replace it with the
rightest in its left branch



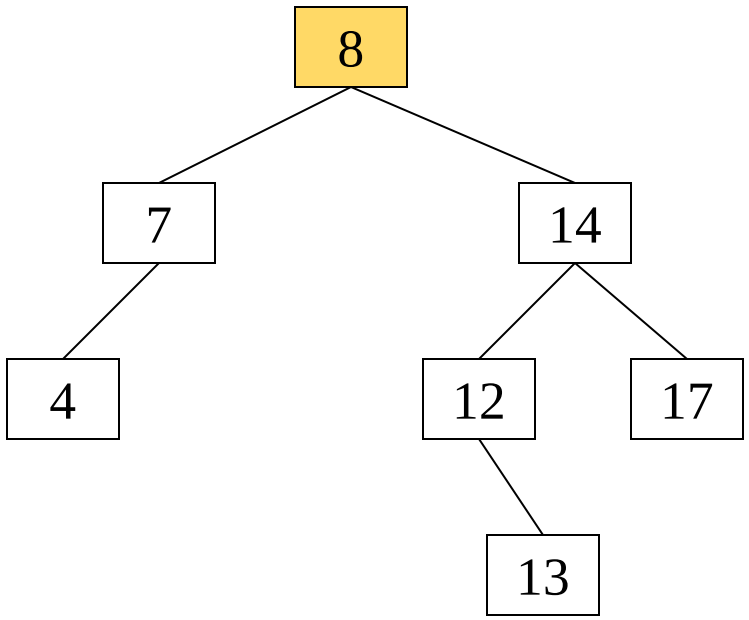
After Remove 11

4. Thêm và xóa trên cây cân bằng

Thực hành

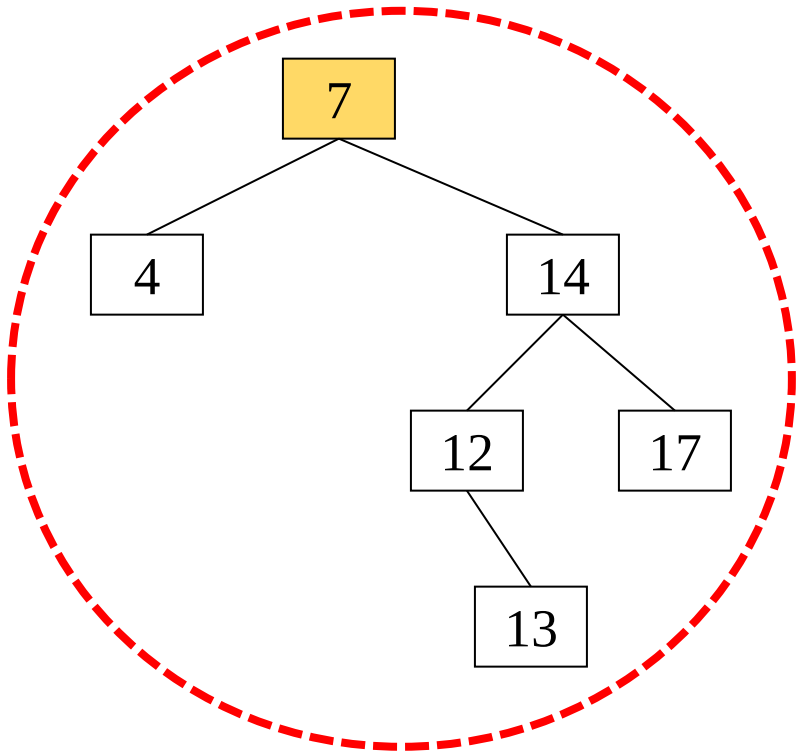
a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ~~***53, 11***~~, ***8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



Remove 8

Replace it with
*the **rightest** in its left branch*



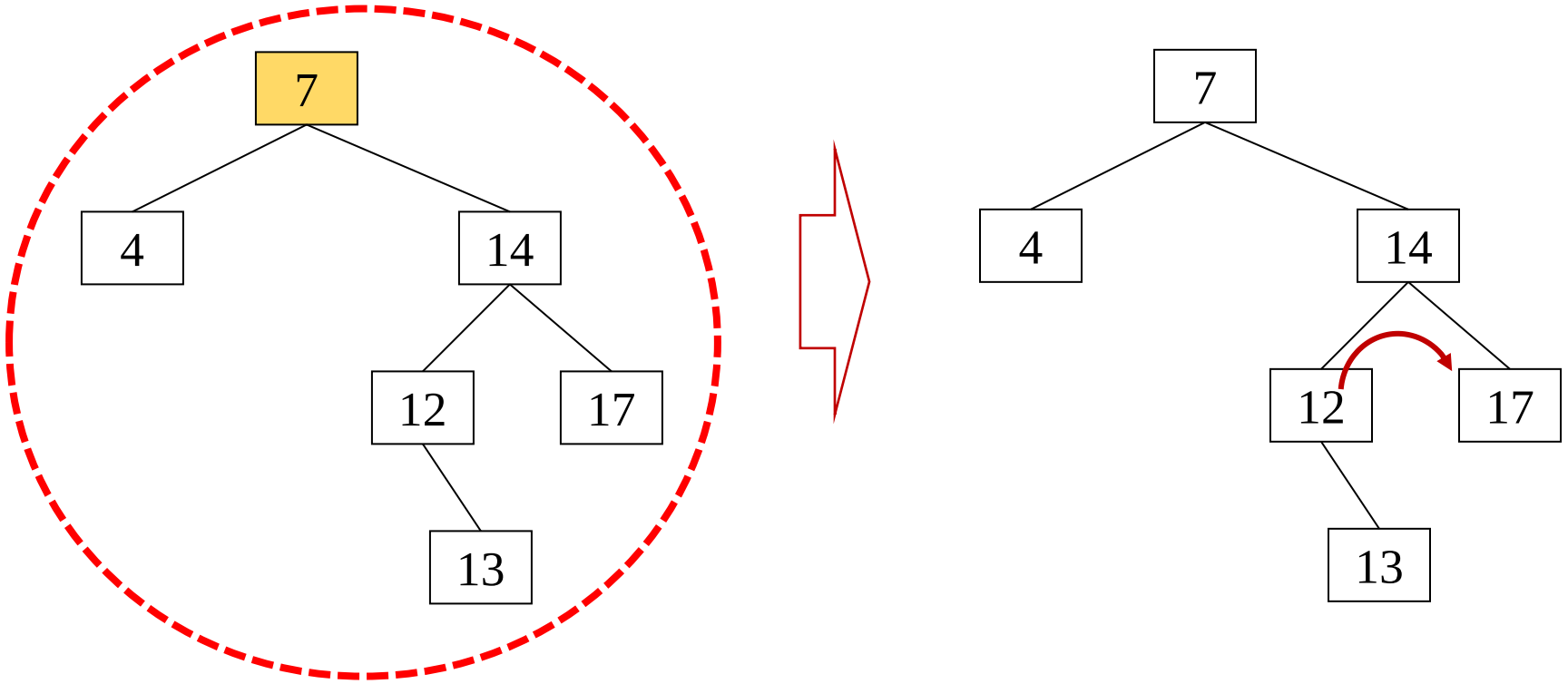
After Remove 8

4. Thêm và xóa trên cây cân bằng

Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

b. Lần lượt xóa các giá trị ~~***53, 11***~~, ***8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



After Remove 8, unbalanced at 7

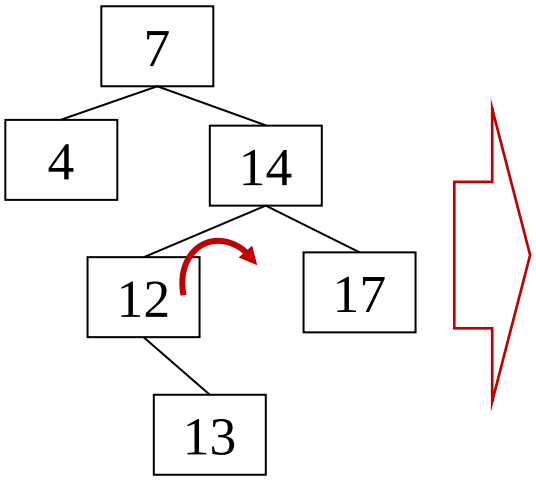
*Next, perform rotation right with **12***

4. Thêm và xóa trên cây cân bằng

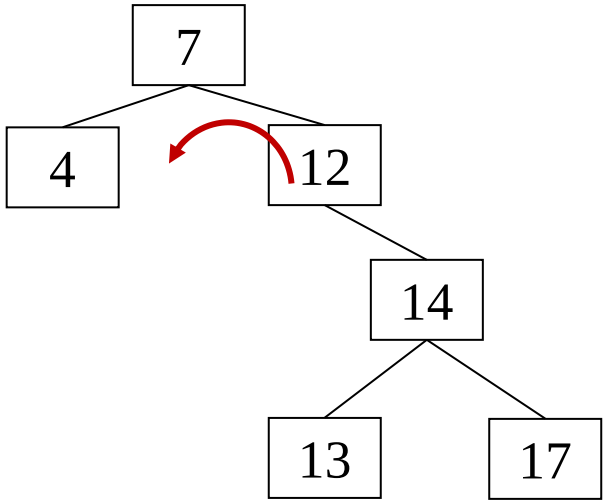
Thực hành

a. Thêm các giá trị sau vào cây AVL rỗng: ~~***14, 17, 11, 7, 53, 4, 13, 12, 8***~~.

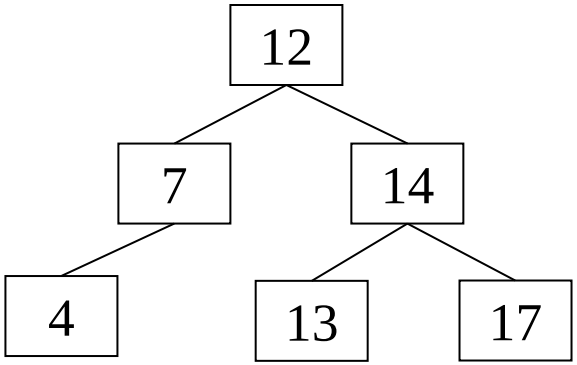
b. Lần lượt xóa các giá trị ~~***53, 11***~~, ***8***. Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái



*Perform rotation right with **12***



*After rotation right with **12***



*After rotation left with **12***

4.4. Nhận xét về thêm và xóa trên AVL

- ***Độ phức tạp***
  - Thao tác thêm một node có độ phức tạp O(1).
  - Thao tác hủy một node có độ phức tạp O(h).
  - Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại.
- ***Nhận xét***
  - Việc hủy 1 node có thể phải cân bằng dây chuyền các node từ gốc cho đên phần tử bị hủy trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ
  - Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn log2n, nhưng việc cân bằng lại đơn giản hơn nhiều
  - Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số node trên cây là bao nhiêu.

5. CÀI ĐẶT AVL TREE

5.1. Khai báo cấu trúc: Để ghi nhận mức độ cân bằng tại mỗi node gốc cây con, dùng thêm thành phần ***balFactor*** trong cấu trúc dữ liệu của mỗi node.

```
#define RNE  -2   /* Mất cân bằng bên phải*/
#define RH   -1   /* Cây con phải cao hơn*/
#define EH    0   /* Hai cây con bằng nhau*/
#define LH    1   /* Cây con trái cao hơn*/
#define LNE   2   /* Mất cân bằng bên trái*/
struct AVLNode{
    char     balFactor; //Chỉ số cân bằng
    DataType data;
    AVLNode* pLeft;
    AVLNode* pRight;
};
AVLNode *AVLTree;
```

5.2. Các tác vụ của cây AVL

- Phần lớn các tác vụ được dùng lại từ cây nghị phân tìm kiếm, trừ tác vụ thêm và xóa một nút trên cây AVL. Nên phần này ta chỉ xét tác vụ thêm một phần tử vào cây AVL.

5.2. Các tác vụ của cây AVL

5.2.1. Tác vụ xoay đơn Left - Left

```
void rotateLL(AVLTree &T) //xoay 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;
}
```

5.2. Các tác vụ của cây AVL

5.2.2. Tác vụ xoay đơn Right - Right

```
void rotateRR (AVLTree &T)   //xoay 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;
}
```

5.2. Các tác vụ của cây AVL

5.2.3. Tác vụ xoay đơn Left - Right

```
void rotateLR(AVLTree &T)//xoay 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;
}
```

5.2. Các tác vụ của cây AVL

5.2.4. Tác vụ xoay đơn Right - Left

```
void rotateRL(AVLTree &T)      //xoay 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;
}
```

5.2. Các tác vụ của cây AVL

5.2.5. Tác vụ cân bằng khi cây 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;
}
```

5.2. Các tác vụ của cây AVL

5.2.6. Tác vụ cân bằng khi cây 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;
}
```

5.2. Các tác vụ của cây AVL

5.2.7. Tác vụ thêm một phần tử vào cây AVL

```
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;
        }
    }
```

5.2. Các tác vụ của cây AVL

5.2.7. Tác vụ thêm một phần tử vào cây AVL (tt)

```
//phần sau là mã lệnh tiếp theo của slide liền trước
    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;
         }
    }
    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
}// End of insertNode function
```

5.2. Các tác vụ của cây AVL

5.2.8. Xóa node

Hàm ***delNode*** sau đây trả về giá trị 1 khi hủy thành công hoặc 0 khi 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)
{   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);
            }// switch
    } // if(T->key > X)
    // xem tiếp slide sau
```

5.2. Các tác vụ của cây AVL

5.2.8. Xóa node

```
// tiếp theo nội dung slide liền trước
    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)
    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;
    }
}// End of delNode function
```

5.2. Các tác vụ của cây AVL

5.2.8. Xóa node

Hàm hủy một phần tử trên cây AVL

```
int searchStandFor(AVLTree &p, AVLTree &q)
//Tìm phần tử thế 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;
    }
}
```

6. NHẬN XÉT VỀ THÊM VÀ HỦY TRÊN AVL

6.1. Độ phức tạp

- Thao tác thêm một node có độ phức tạp O(1)
- Thao tác hủy một node có độ phức tạp O(h)
- Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại

6.2. Nhận xét

- Việc hủy 1 node có thể phải cân bằng dây chuyền các node từ gốc cho đên phần tử bị hủy trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ
- Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn $\log_2 n$, nhưng việc cân bằng lại đơn giản hơn nhiều
- Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số node trên cây là bao nhiêu

6.3. Chiều cao của AVL

Gọi n là số node và h là chiều cao của AVL tree:

- Chiều cao tối thiểu (h_{min}) là $\log_2 n$.
- Chiều cao tối đa (h_{max}) là $1.44*\log_2 n$.
- Với h là chiều cao hiện tại của AVL tree thì số node tối đa có trên cây là $2^{h+1} - 1$.
- Với h là chiều cao hiện tại của AVL tree thì số node tối thiểu có trên cây là:

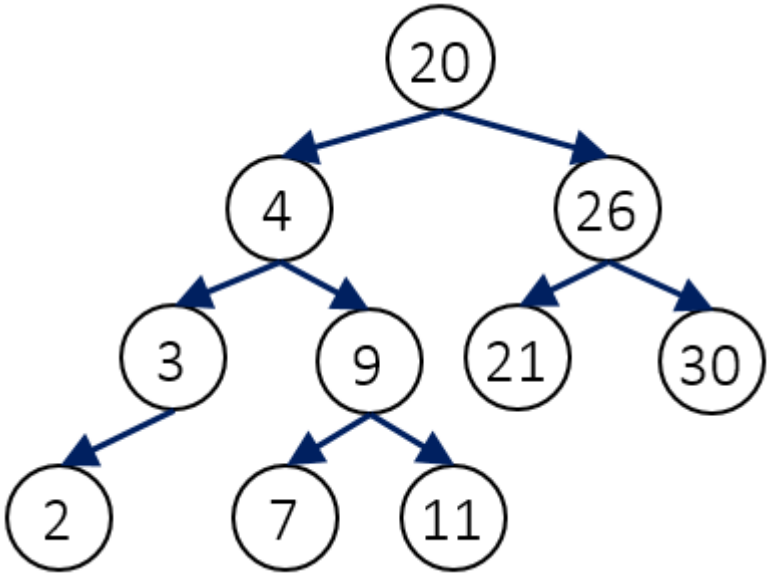
$$n(h) = n(h-1) + n(h-2) + 1$$

Với $n>2$ trong đó $n(0)=1$ và $n(1)=2$.

- Độ phức tạp của các thao tác tìm kiếm, thêm, xóa trên AVL tree là $O(\log n)$.

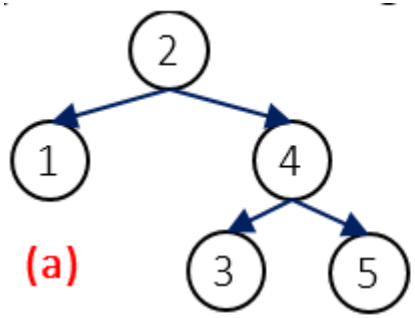
10. THỰC HÀNH

10.1. Cho AVL như hình bên, ghi ra hệ số cân bằng của mỗi node. Thêm node **15** vào cây. Cân bằng, vẽ lại cây và ghi ra hệ số cân bằng của mỗi node

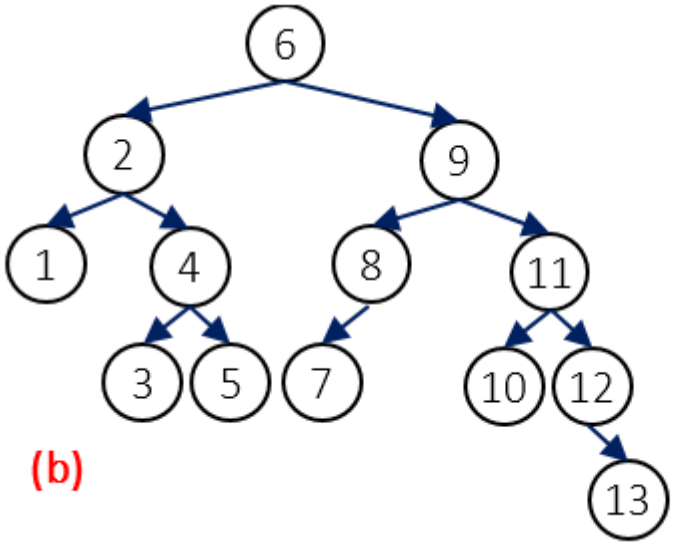


10. THỰC HÀNH

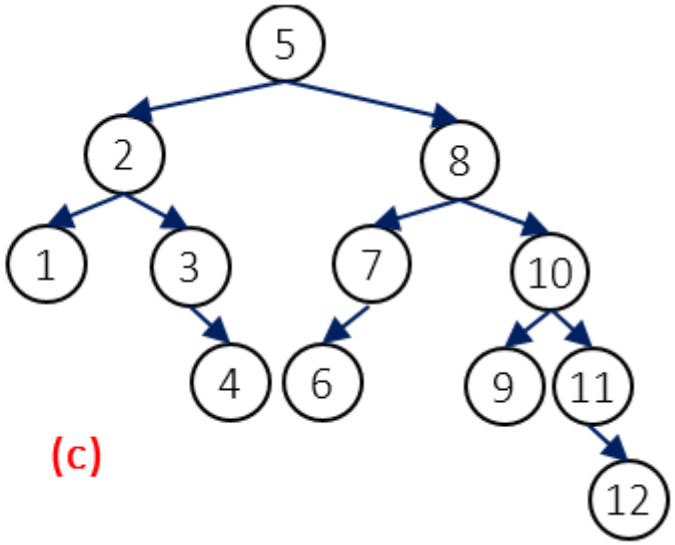
10.2. Thực hiện xóa node có giá trị **=1** trên mỗi cây sau. Ghi ra hệ số cân bằng của mỗi node. Vẽ lại cây và ghi ra hệ số cân bằng của mỗi node sau khi cân bằng.



(a)



(b)



(c)

10.3. Thêm (insert) và xóa (remove) trên AVL tree. Yêu cầu chung:

- ***Khi thêm***:
  - Mỗi bài đều xuất phát từ cây AVL ban đầu rỗng, thực hiện thêm node vào cây,
  - Sau mỗi lần thêm, nếu cây bị mất cân bằng, cần thực hiện cân bằng trước khi tiếp tục thêm các phần tử khác.
- ***Khi xóa***:
  - Nếu cần tìm phần tử thế mạng thì chọn phần tử phải nhất của cây con trái.
  - Sau mỗi lần xóa, nếu cây bị mất cân bằng, cần thực hiện cân bằng trước khi tiếp tục xóa các phần tử khác.

10.3. Thêm (insert) và xóa (remove) trên AVL tree.

a. Bài a

- Thêm các giá trị sau vào cây AVL rỗng: ***15, 20, 24, 10, 13, 7, 30, 36, 25***.
- Lần lượt xóa các giá trị ***24, 20, 15***. Nếu cần tìm phần tử thế mạng, sẽ tìm phần tử trái nhất của cây con phải.

b. Bài b

- Thêm các giá trị sau vào cây AVL rỗng: ***10, 20, 15, 25, 30, 16, 18, 19***.
- Lần lượt xóa các giá trị ***30***. Nếu cần tìm phần tử thế mạng, sẽ tìm phần tử phải nhất của cây con trái.

c. Bài c

- Thêm các giá trị sau vào cây AVL rỗng: ***8, 3, 5, 2, 20, 11, 30, 9, 18, 4***.
- Vẽ lại hình ảnh của cây trên nếu ta lần lượt xoá 2 nút ***5*** và ***20***. Nếu cần tìm phần tử thế mạng, sẽ tìm phần tử trái nhất của cây con phải.

10.3. Thêm (insert) và xóa (remove) trên AVL tree.

d. Bài d

- Thêm các giá trị sau vào cây AVL rỗng: ***9, 27, 50, 15, 2, 21, 36***.

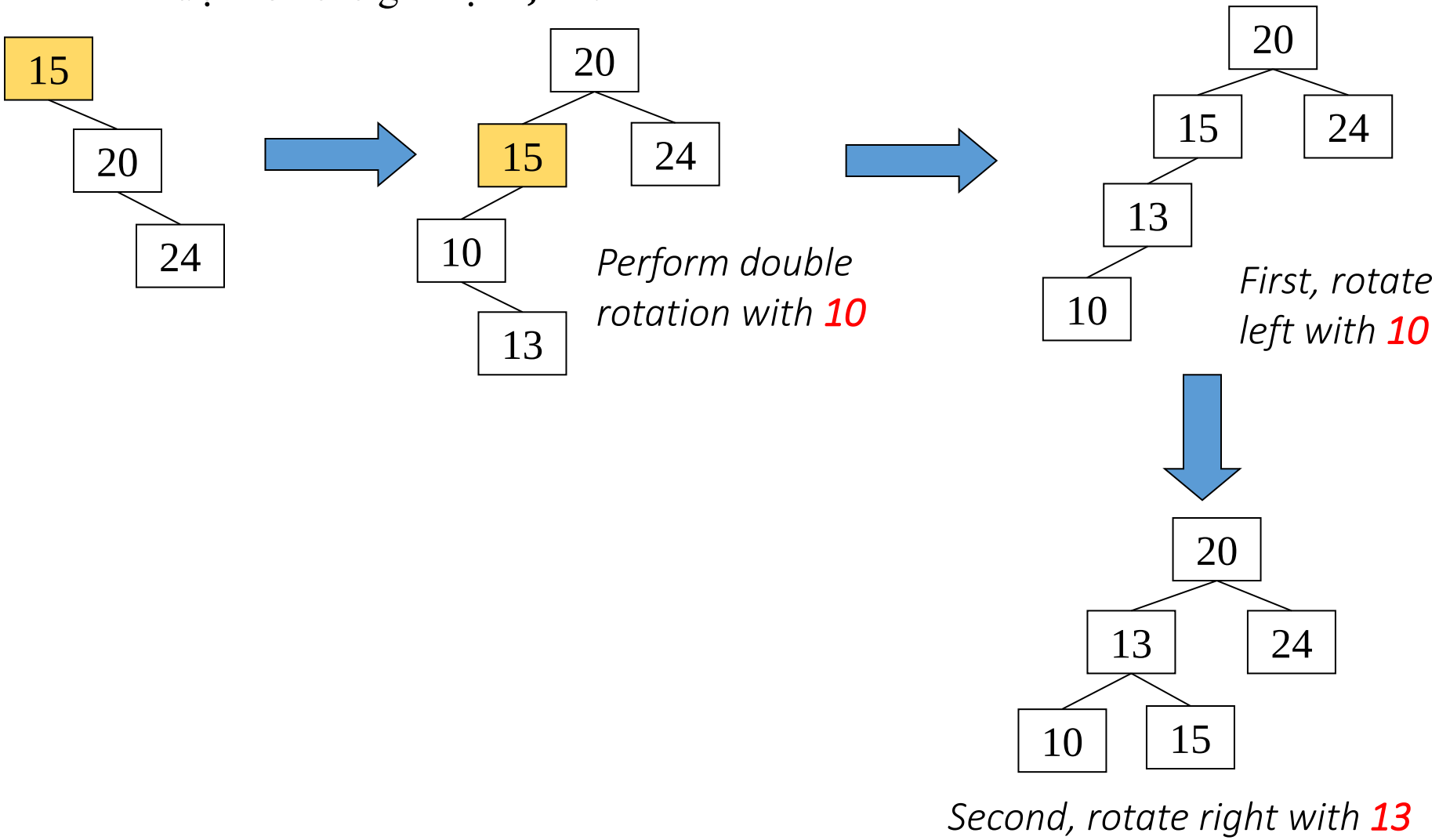
e. Bài e

- Thêm các giá trị sau vào cây AVL rỗng: ***14, 17, 11, 7, 53, 4, 13, 12, 8***.
- Lần lượt xóa các giá trị ***53, 11, 8***. Nếu cần tìm phần tử thế mạng, sẽ tìm phần tử phải nhất của cây con trái.

10.3. Thêm (insert) và xóa (remove) trên AVL tree.

a. Bài a

Thêm các giá trị sau vào cây AVL rỗng: ***15, 20, 24, 10, 13, 7, 30, 36, 25***.
Lần lượt xóa các giá trị ***24, 20***.

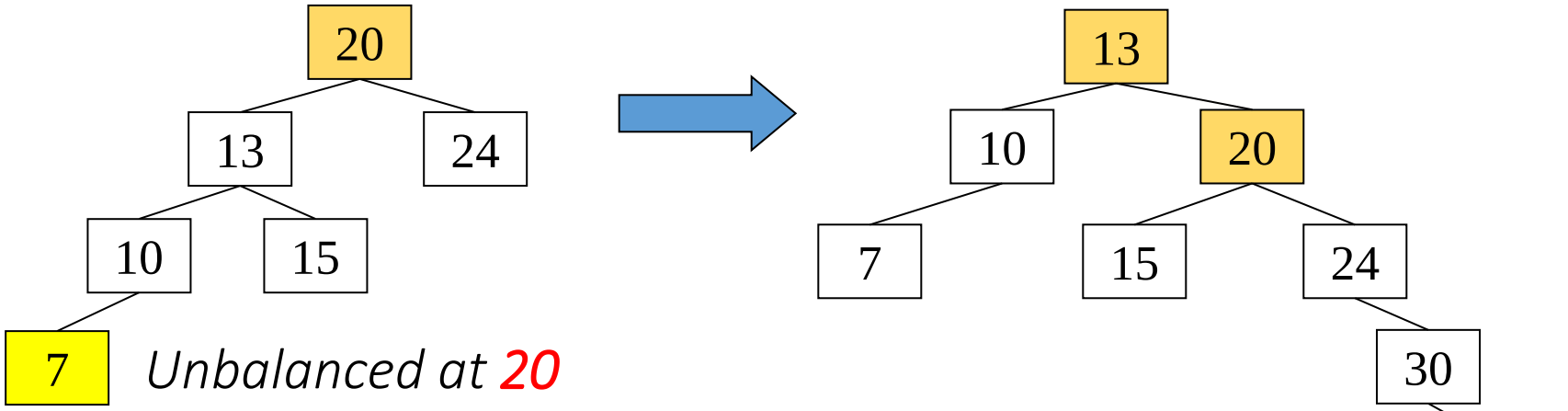


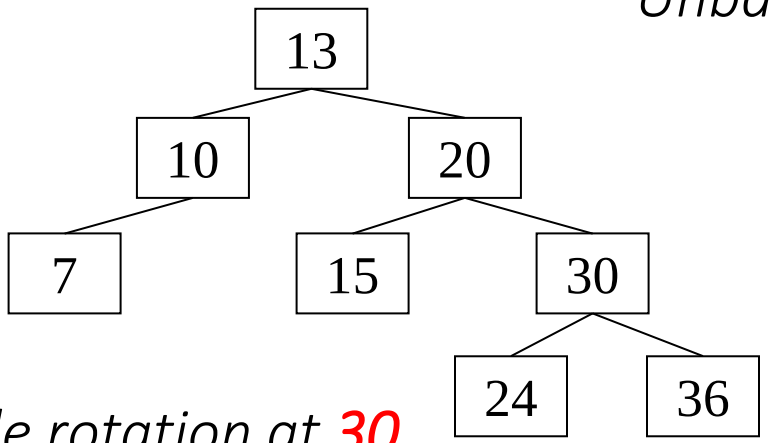
10.3. Thêm (insert) và xóa (remove) trên AVL tree.

a. Bài a

Thêm các giá trị sau vào cây AVL rỗng: ***15, 20, 24, 10, 13, 7, 30, 36, 25***.

Lần lượt xóa các giá trị ***24, 20***.



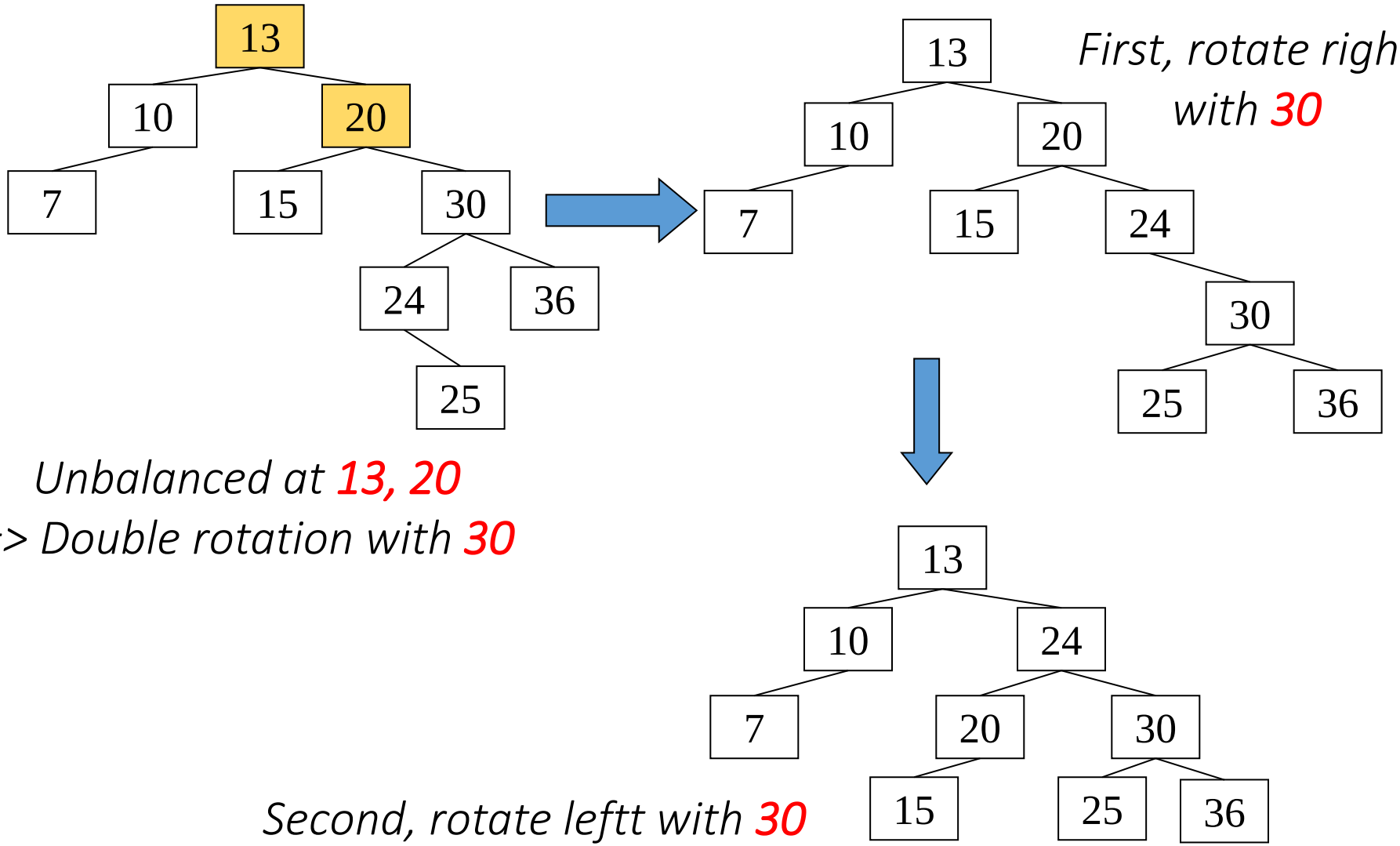


10.3. Thêm (insert) và xóa (remove) trên AVL tree.

a. Bài a

Thêm các giá trị sau vào cây AVL rỗng: ***15, 20, 24, 10, 13, 7, 30, 36, 25***.

Lần lượt xóa các giá trị ***24, 20***.

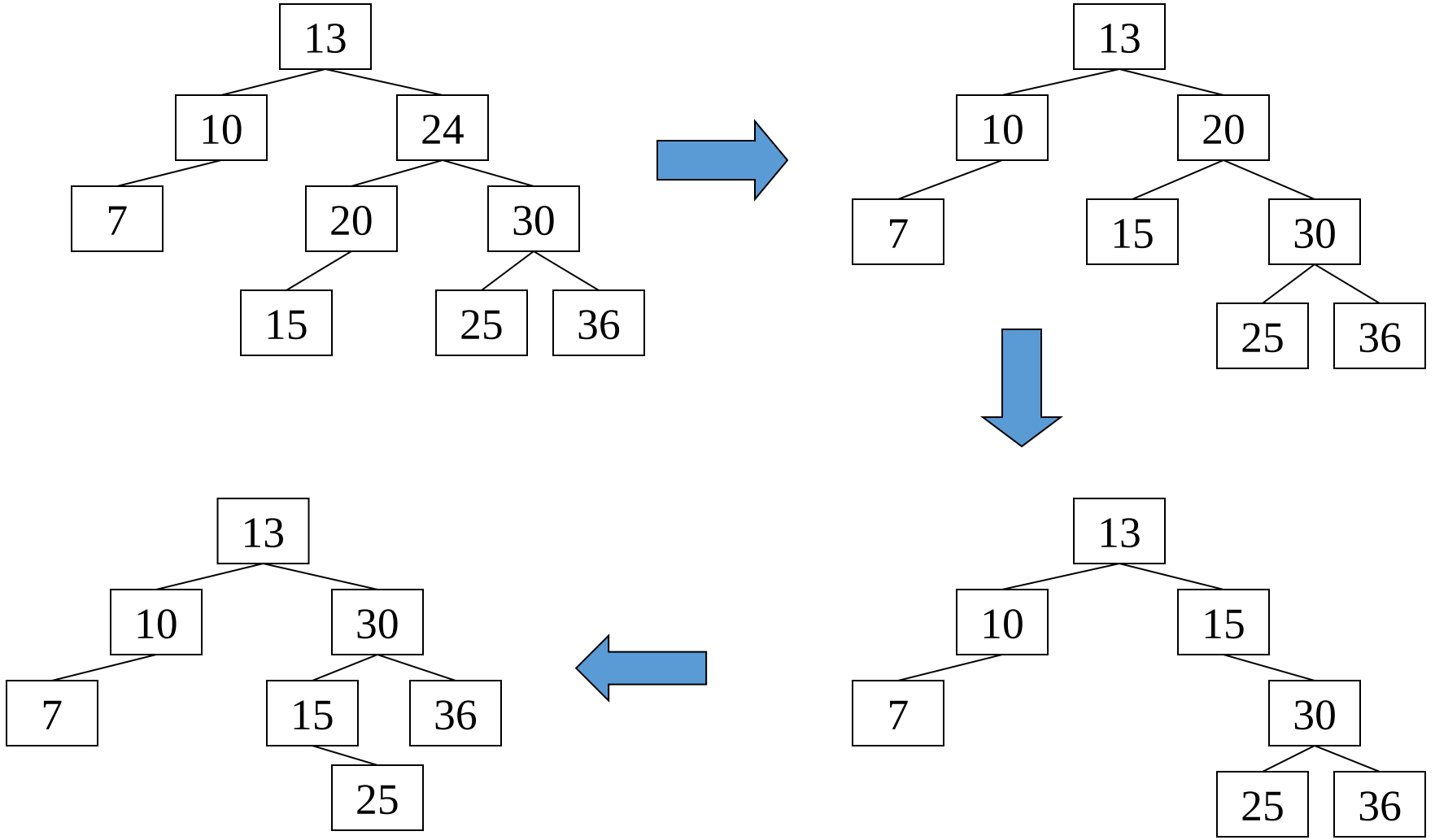


10.3. Thêm (insert) và xóa (remove) trên AVL tree.

a. Bài a

Thêm các giá trị sau vào cây AVL rỗng: ***15, 20, 24, 10, 13, 7, 30, 36, 25***.

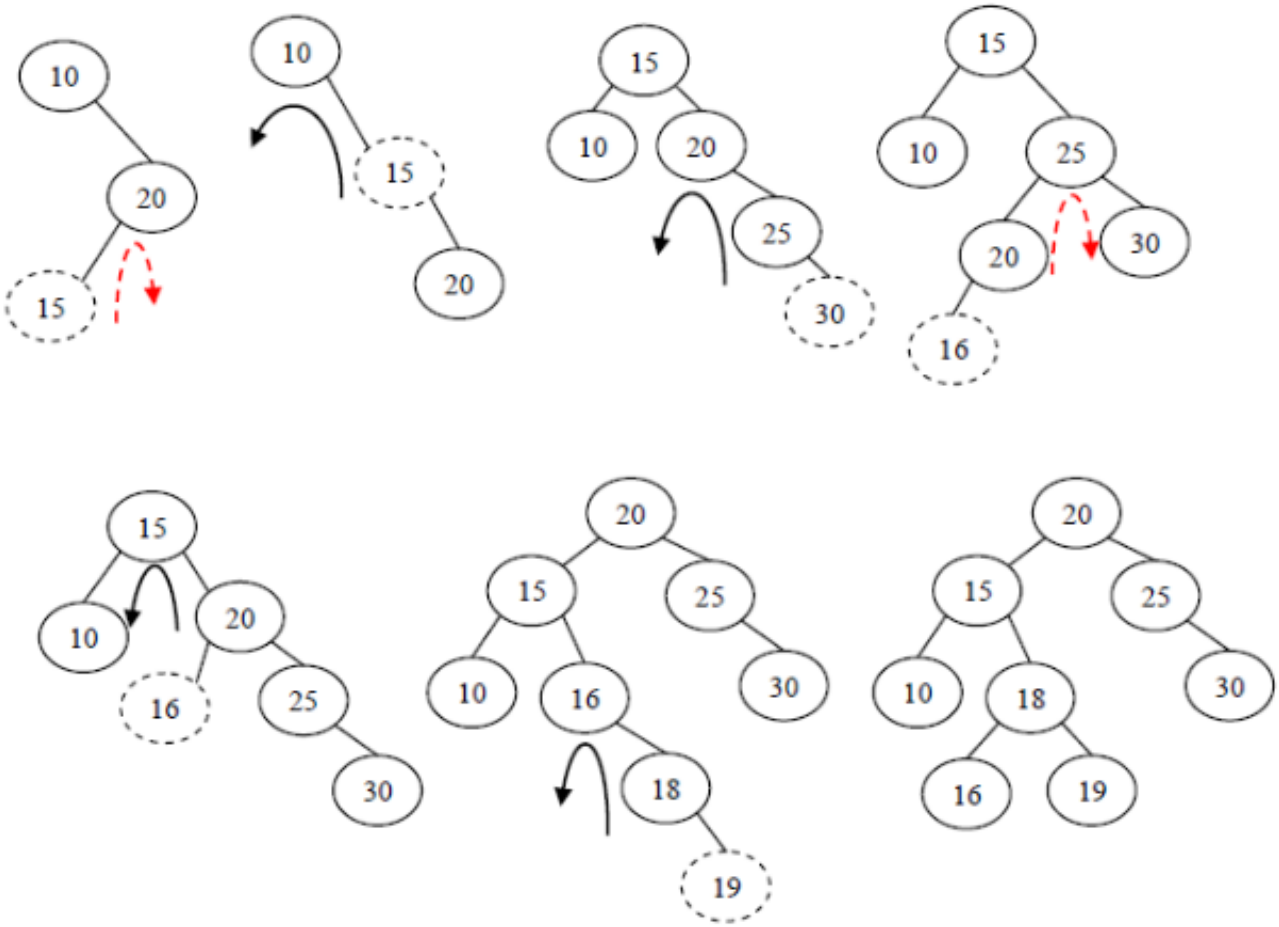
Lần lượt xóa các giá trị ***24***, ***20***.



10.3. Thêm (insert) và xóa (remove) trên AVL tree.

b. Bài b

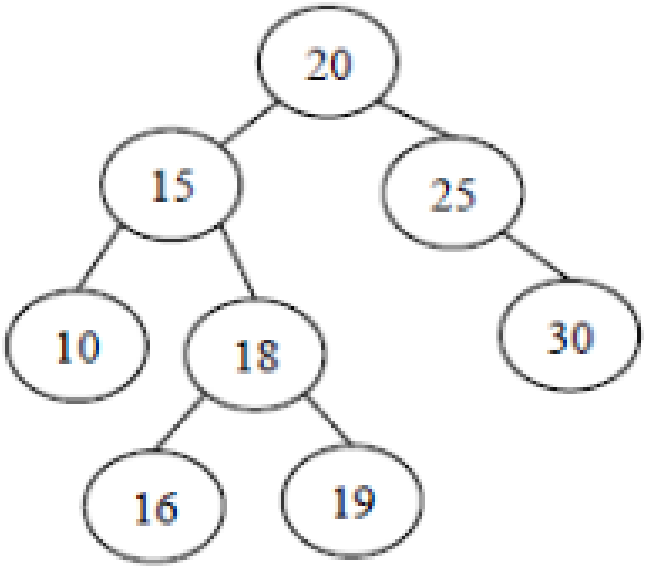
- Thêm các giá trị sau vào cây AVL rỗng: ***10, 20, 15, 25, 30, 16, 18, 19***.
- Lần lượt xóa các giá trị ***30***.

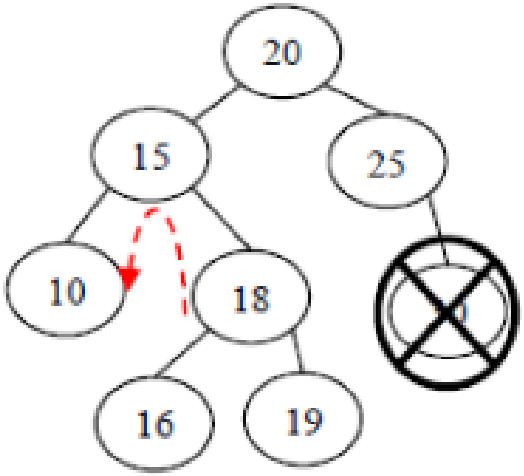


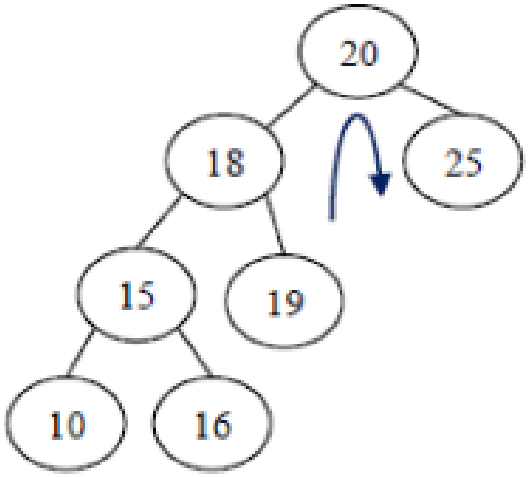
10.3. Thêm (insert) và xóa (remove) trên AVL tree.

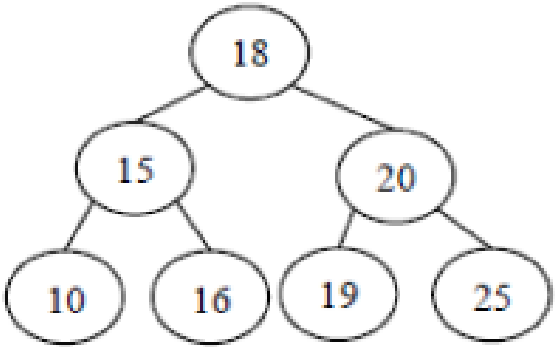
b. Bài b

- Thêm các giá trị sau vào cây AVL rỗng: ***10, 20, 15, 25, 30, 16, 18, 19***.
- Xóa giá trị ***30***.



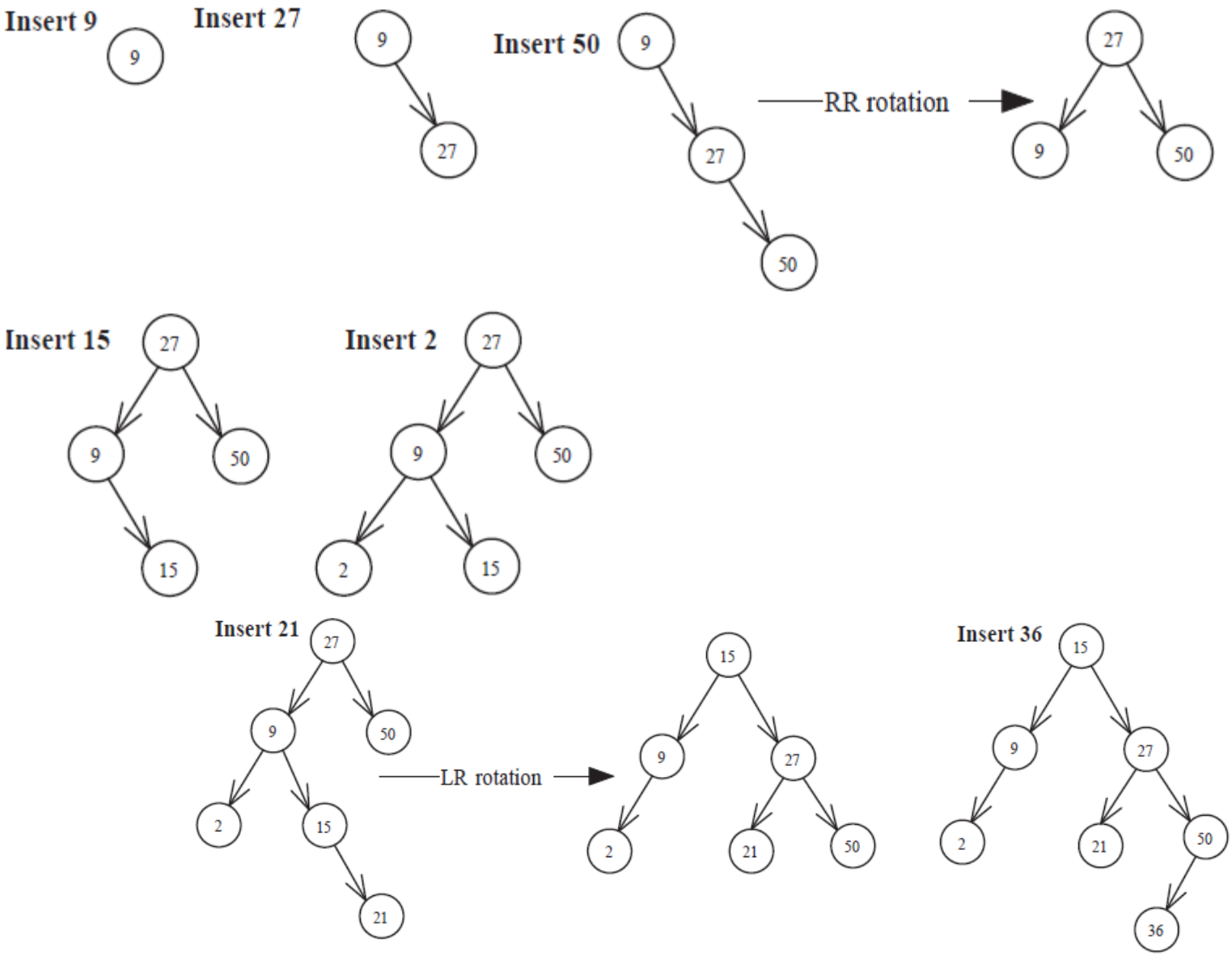






10.3. Thêm (insert) và xóa (remove) trên AVL tree.

b. Bài c: Thêm các giá trị sau vào cây AVL rỗng: ***9, 27, 50, 15, 2, 21, 36***.




Q&A
How
Why
When
Where
What
Who