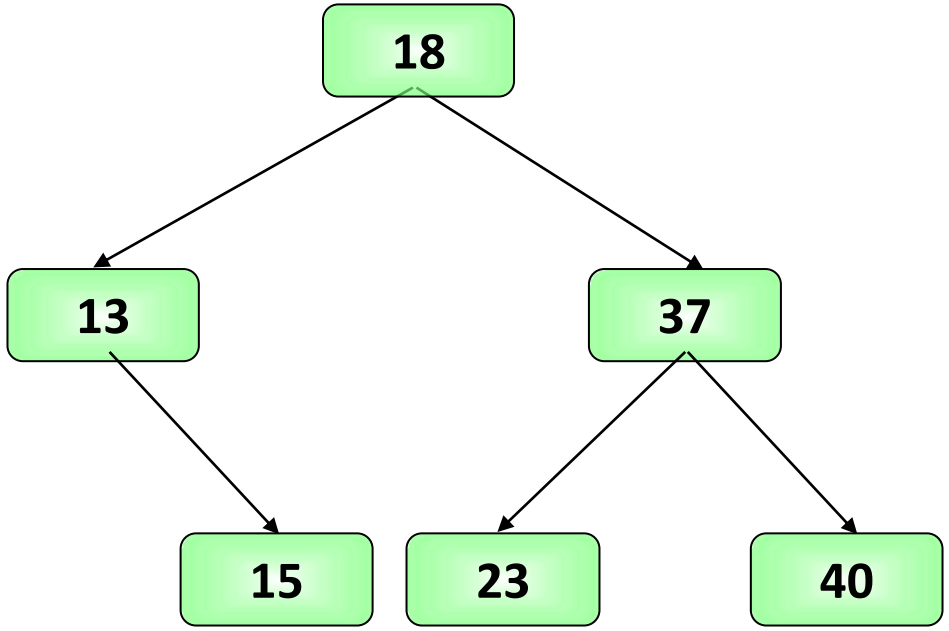
NỘI DUNG

CÂY NHỊ PHÂN TÌM KIẾM
CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Định nghĩa cây nhị phân tìm kiếm

- Cây nhị phân
- Bảo đảm nguyên tắc bố trí khoá tại mỗi nút:
  - Các nút trong cây trái nhỏ hơn nút hiện hành
  - Các nút trong cây phải lớn hơn nút hiện hành

Ví dụ:



Ưu điểm của cây nhị phân tìm kiếm

- Nhờ trật tự bố trí khóa trên cây :
  - Định hướng được khi tìm kiếm
- Cây gồm N phần tử :
  - Trường hợp tốt nhất h = $\log_2 N$
  - Trường hợp xấu nhất h = Ln
  - Tình huống xảy ra trường hợp xấu nhất ?

Cấu trúc dữ liệu của cây nhị phân tìm kiếm

- *Cấu trúc dữ liệu của 1 nút*

```
typedef struct tagTNode
{
    int    Key; //trường dữ liệu là 1 số nguyên
    struct tagTNode *pLeft;
    struct tagTNode *pRight;
}TNode;
```

- *Cấu trúc dữ liệu của cây*

```
typedef TNode *TREE;
```

Các thao tác trên cây nhị phân tìm kiếm

- Tạo 1 cây rỗng
- Tạo 1 nút có trường Key bằng x
- Thêm 1 nút vào cây nhị phân tìm kiếm
- Xoá 1 nút có Key bằng x trên cây
- Tìm 1 nút có khoá bằng x trên cây

Tạo cây rỗng

- Cây rỗng -> địa chỉ nút gốc bằng NULL

```
void CreateTree(TREE &T)
{
    T=NULL;
}
```

Tạo 1 nút có Key bằng x

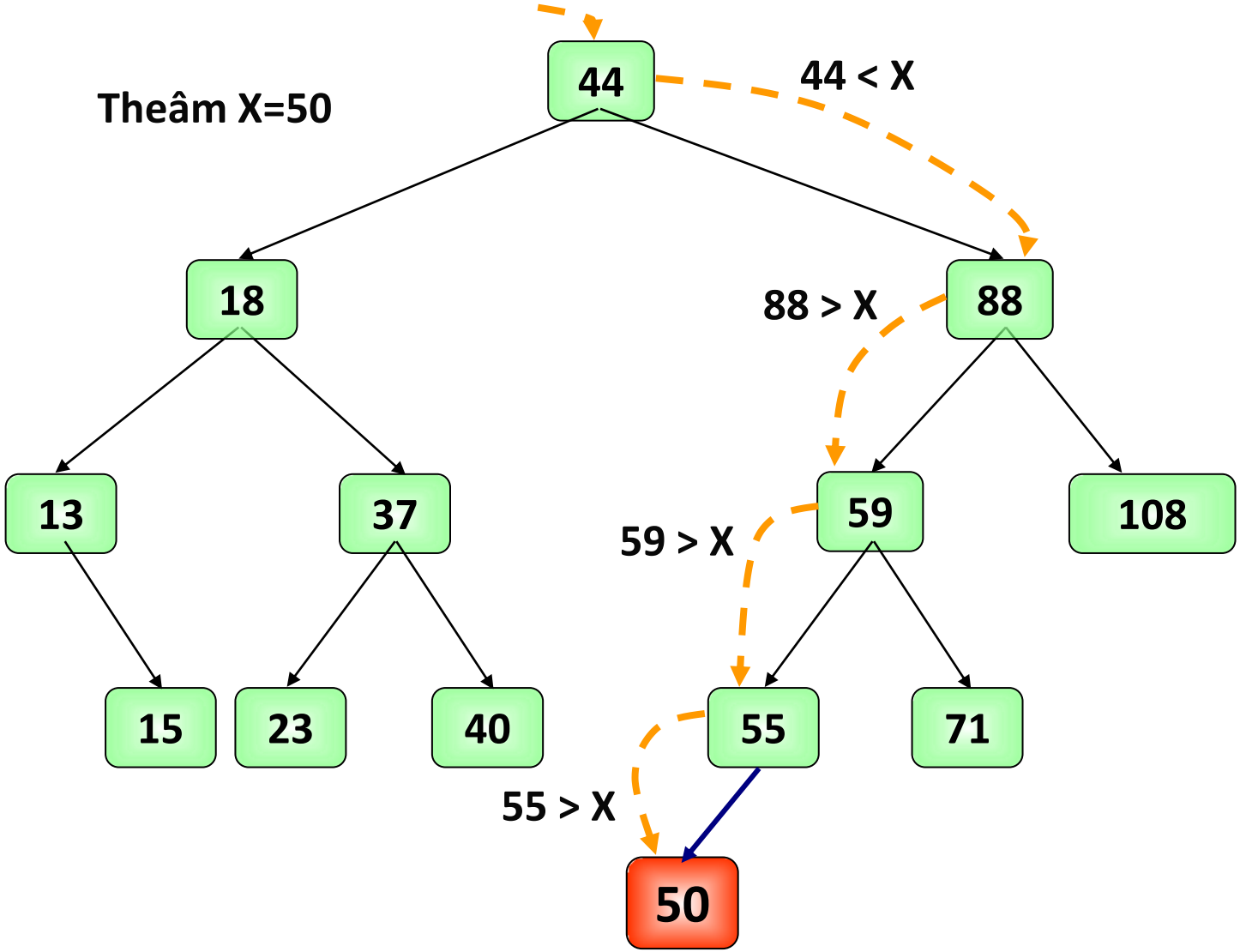
```
TNode *CreateTNode(int x)
{
        TNode *p;
        p = new TNode; //cấp phát vùng nhớ động
        if(p==NULL)
                exit(1); // thoát
        else
        {
                p->key = x; //gán trường dữ liệu của nút = x
                p->pLeft = NULL;
                p->pRight = NULL;
        }
        return p;
}
```


Thêm một nút x

- **Rằng buộc**: Sau khi thêm cây đảm bảo là cây nhị phân tìm kiếm.

```
int insertNode(TREE &T, Data X)
{ if(T)
  {     if(T->Key == X)       return 0;
        if(T->Key > X) return insertNode(T->pLeft, X);
        else    return insertNode(T->pRight, X);}
        T       = new TNode;
        if(T == NULL)         return -1;
        T->Key        = X;
        T->pLeft =T->pRight = NULL;
   return 1;
}
```


Minh họa thêm 1 phần tử vào cây



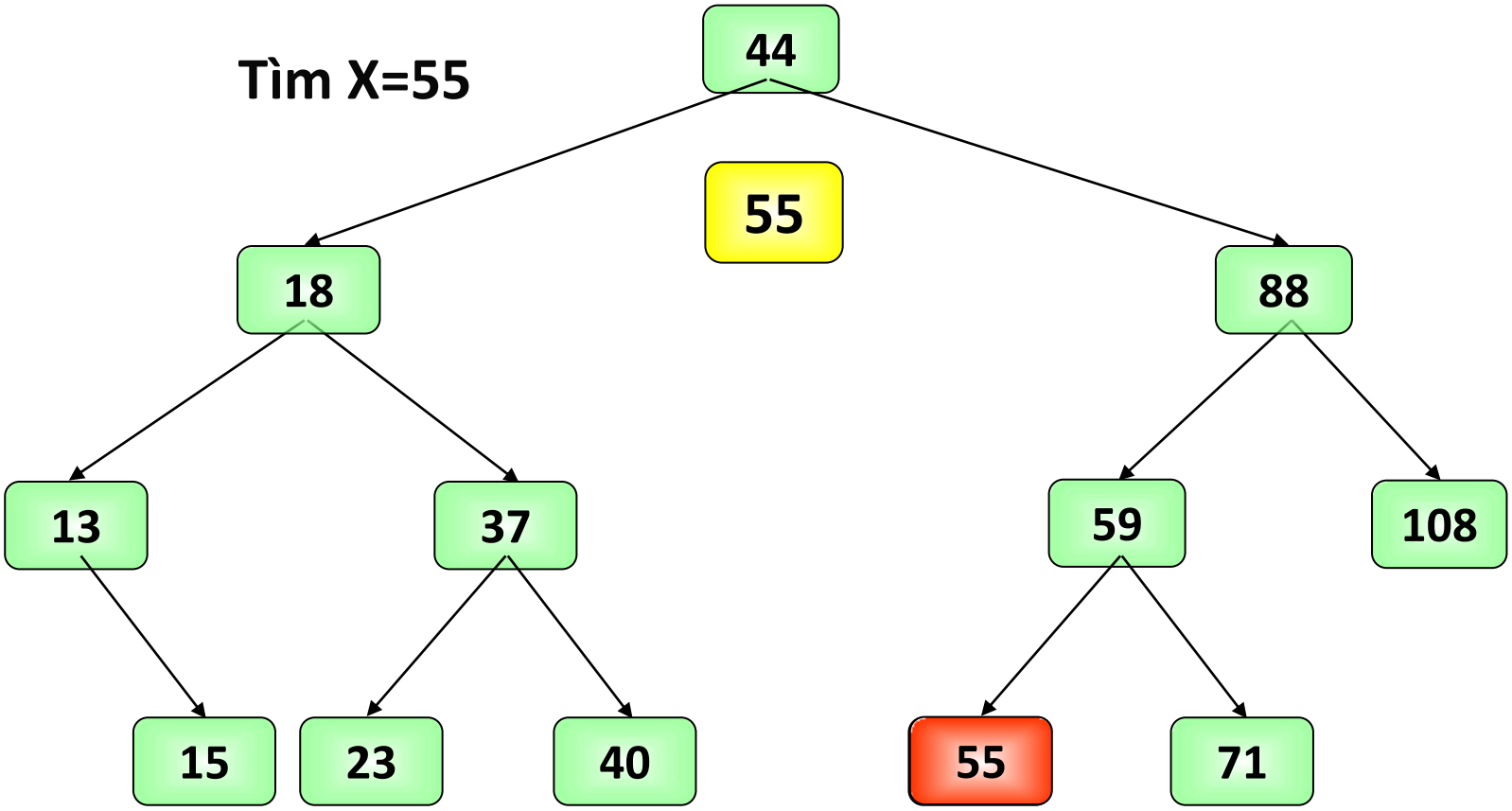
Tìm nút có khoá bằng x (không dùng đệ quy)

```
TNode * searchNode(TREE Root,  Data x)
{       Node *p = Root;
        while (p != NULL)
        {       if(x == p->Key) return p;
                else
                if(x < p->Key)   p = p->pLeft;
                else  p = p->pRight;
        }
        return NULL;
}
```


Tìm nút có khoá bằng x (dùng đệ quy)

```
TNode *SearchTNode(TREE T, int x)
{
   if(T!=NULL)
   {
        if(T->key==x)
                return T;
        else
                if(x>T->key)
                        return SearchTNode(T->pRight,x);
                else
                        return SearchTNode(T->pLeft,x);
   }
   return NULL;
}
```

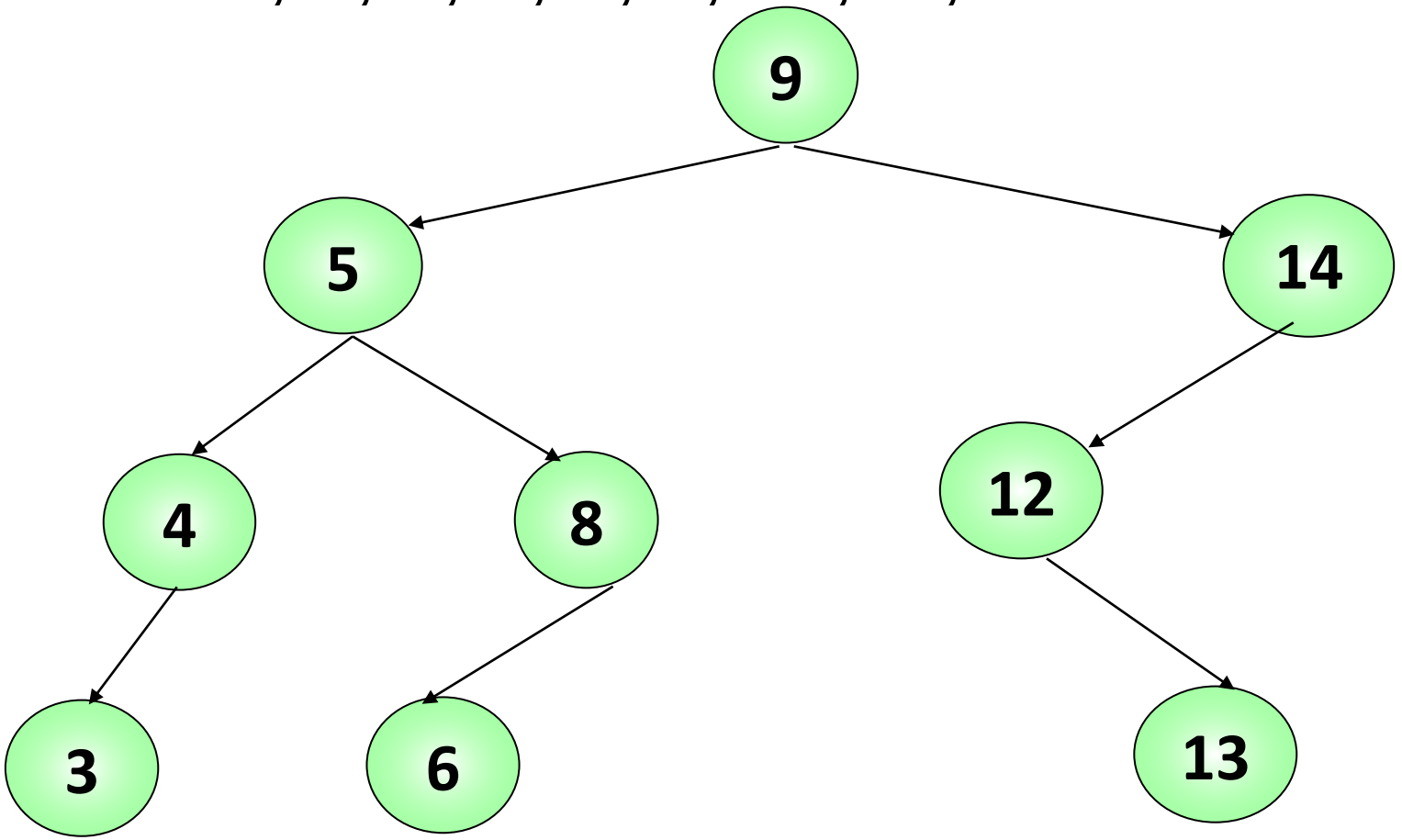

Minh hoạ tìm một nút



Tìm thấy X=55

Minh hoạ thành lập 1 cây từ dãy số

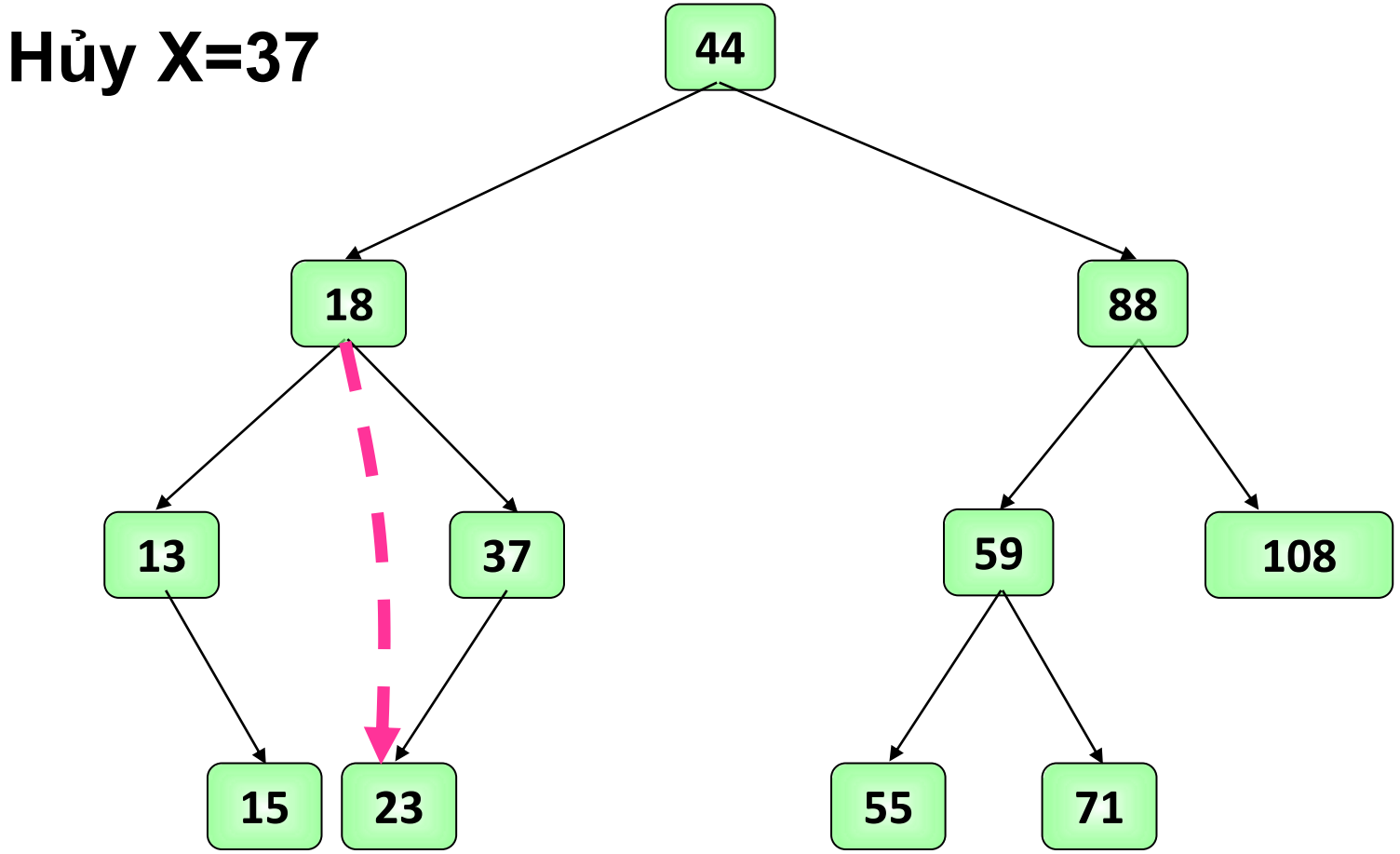
9, 5, 4, 8, 6, 3, 14,12,13



Hủy 1 nút có khoá bằng X trên cây

- Hủy 1 phần tử trên cây phải đảm bảo điều kiện ràng buộc của Cây nhị phân tìm kiếm
- Có 3 trường hợp khi hủy 1 nút trên cây
  - TH1: X là nút lá
  - TH2: X chỉ có 1 cây con (cây con trái hoặc cây con phải)
  - TH3: X có đầy đủ 2 cây con
- TH1: Ta xoá nút lá mà không ành hưởng đến các nút khác ttrên cây
- TH2: Trước khi xoá x ta móc nối cha của X với con duy nhất cùa X.
- TH3: Ta dùng cách xoá gián tiếp

Minh hoạ hủy phần tử x có 1 cây con



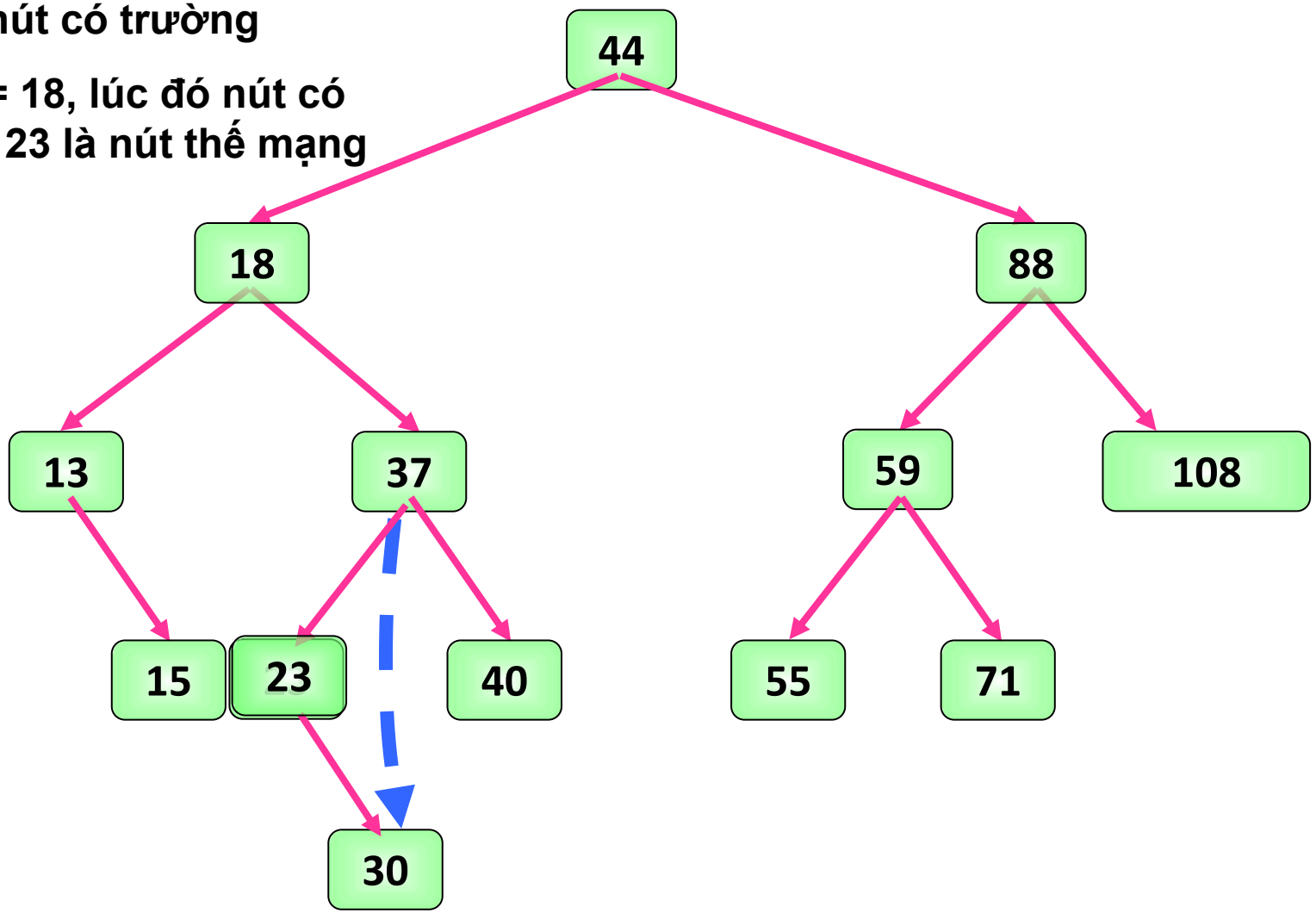
Hủy 1 nút có 2 cây con

- Ta dùng cách hủy gián tiếp, do X có 2 cây con
- Thay vì hủy X ta tìm phần tử thế mạng Y. Nút Y có tối đa 1 cây con.
- Thông tin lưu tại nút Y sẽ được chuyển lên lưu tại X.
- Ta tiến hành xoá hủy nút Y (xoá Y giống 2 trường hợp đầu)
- Cách tìm nút thế mạng Y cho X: Có 2 cách
  - C1: Nút Y là nút có khoá nhỏ nhất (trái nhất) bên cây con phải X
  - C2: Nút Y là nút có khoá lớn nhất (phải nhất) bên cây con trái của X

Minh họa hủy phần tử X có 2 cây con

Xoá nút có trường

Key = 18, lúc đó nút có khoá 23 là nút thế mạng



Cài đặt thao tác xoá nút có trường Key = x

```
void DeleteNodeX1(TREE &T,int x)
{
    if(T!=NULL)
    {
        if(T->Key<x)        DeleteNodeX1(T->Right,x);
        else
        {
            if(T->Key>x)        DeleteNodeX1(T->Left,x);
            else  //tim thấy  Node có trường dữ liệu = x
            {       TNode *p;
                    p=T;
                    if (T->Left==NULL)          T = T->Right;
                    else
                    {       if(T->Right==NULL)          T=T->Left;
                            else       ThayThe1(p, T->Right);// tìm bên cây con
phải
                    }
                    delete p;
            }
        }
    }
    else  printf("Khong tim thay phan can xoa tu");}
```

Hàm tìm phần tử thế mạng

```
void ThayThe1(TREE &p, TREE &T)
{   if(T->Left!=NULL)
        ThayThe1(p,T->Left);
    else
    {
        p->Key = T->Key;
        p=T;
        T=T->Right;
    }
}
```

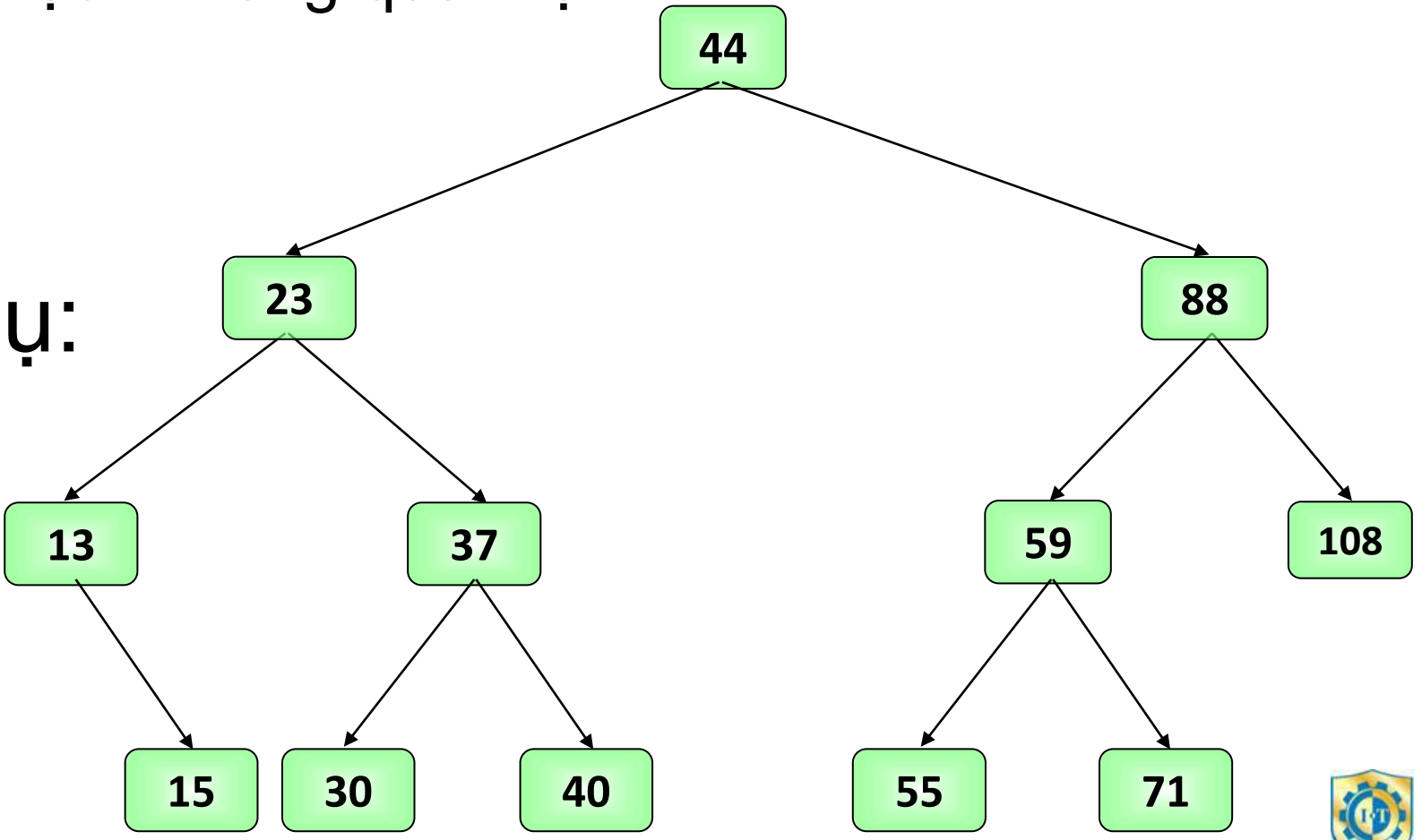

Cây nhị phân tìm kiếm cân bằng

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

Định nghĩa

- 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:



Tổ chức dữ liệu

- **Chỉ số cân bằng = độ lệch giữa cây trái và cây phải của một nút**
- **Các giá trị hợp lệ :**
  - **CSCB(p) = 0 ⇔ Độ cao cây trái (p) = Độ cao cây phải (p)**
  - **CSCB(p) = 1 ⇔ Độ cao cây trái (p) < Độ cao cây phải (p)**
  - **CSCB(p) = -1 ⇔ Độ cao cây trái (p) > Độ cao cây phải (p)**

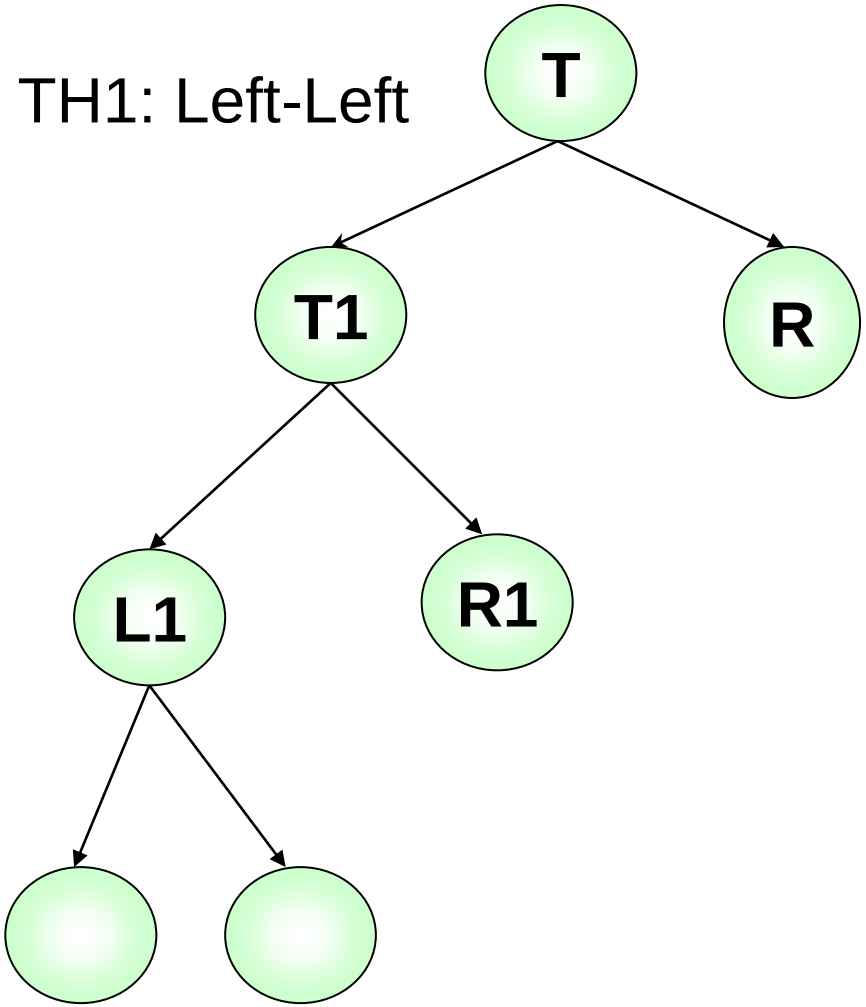
Tổ chức dữ liệu(tt)

```
#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
typedef struct tagAVLNode
{  char    balFactor; //chỉ số cân bằng
   Data    key;
   struct tagAVLNode*       pLeft;
   struct tagAVLNode*       pRight;
}AVLNode;
typedef AVLNode    *AVLTree;
```

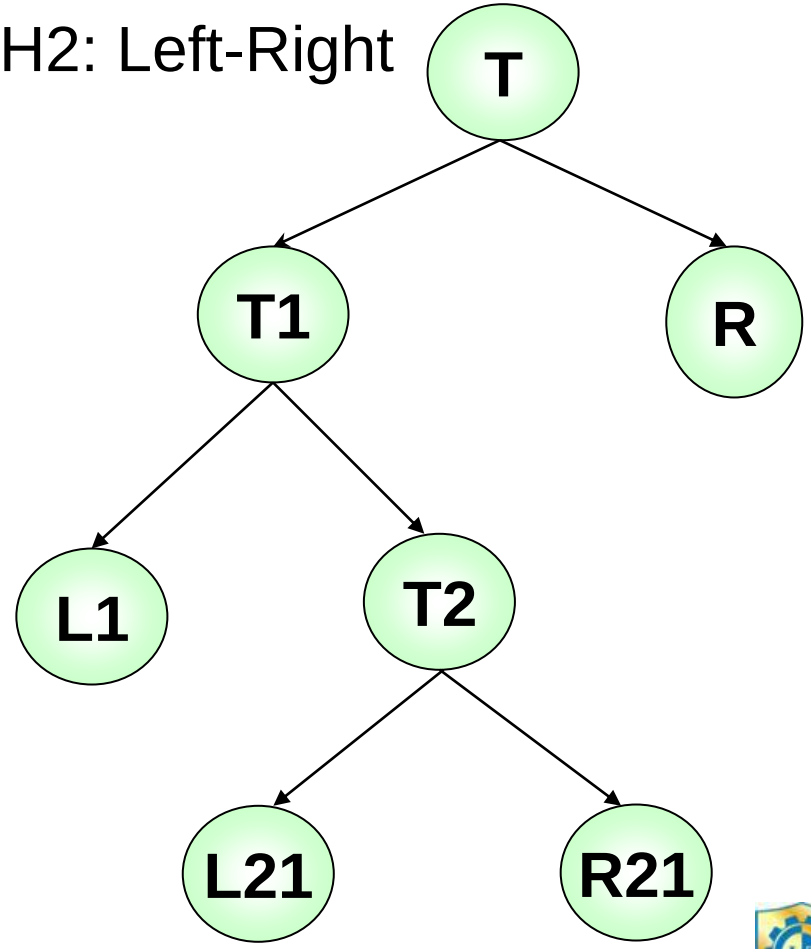

Các trường hợp mất cân bằng do lệch trái

- Cây mất cân bằng tại nút T

TH1: Left-Left



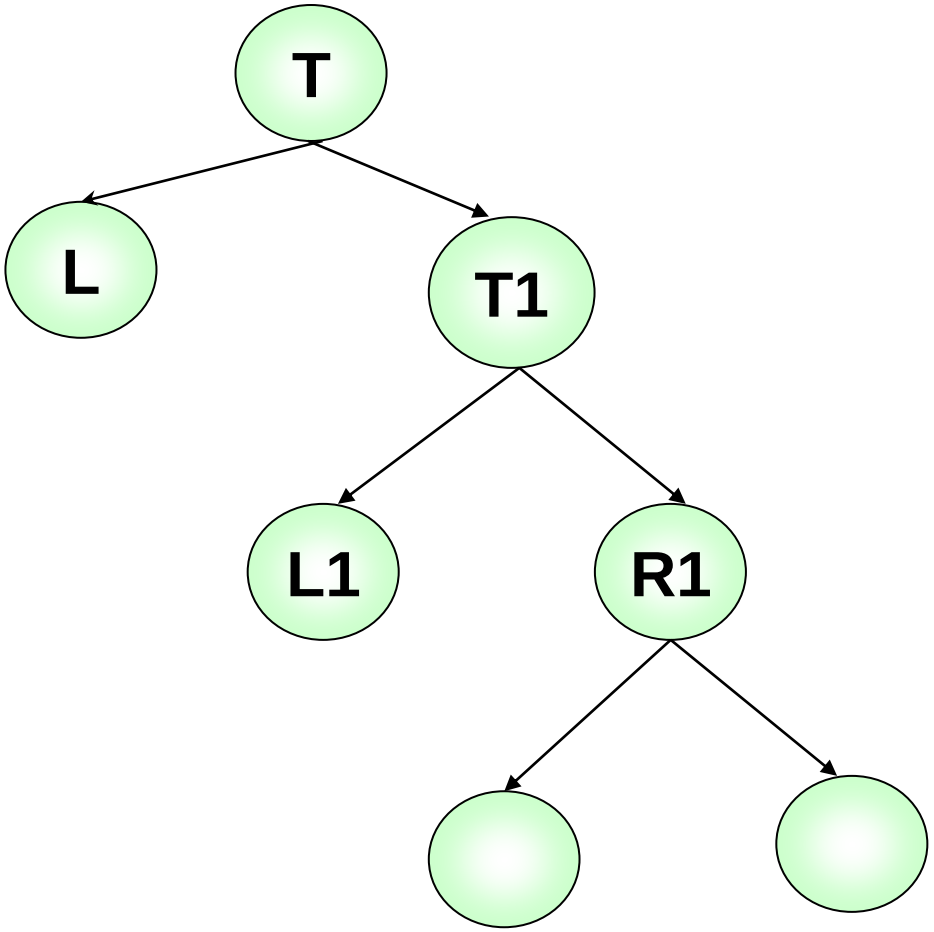
TH2: Left-Right



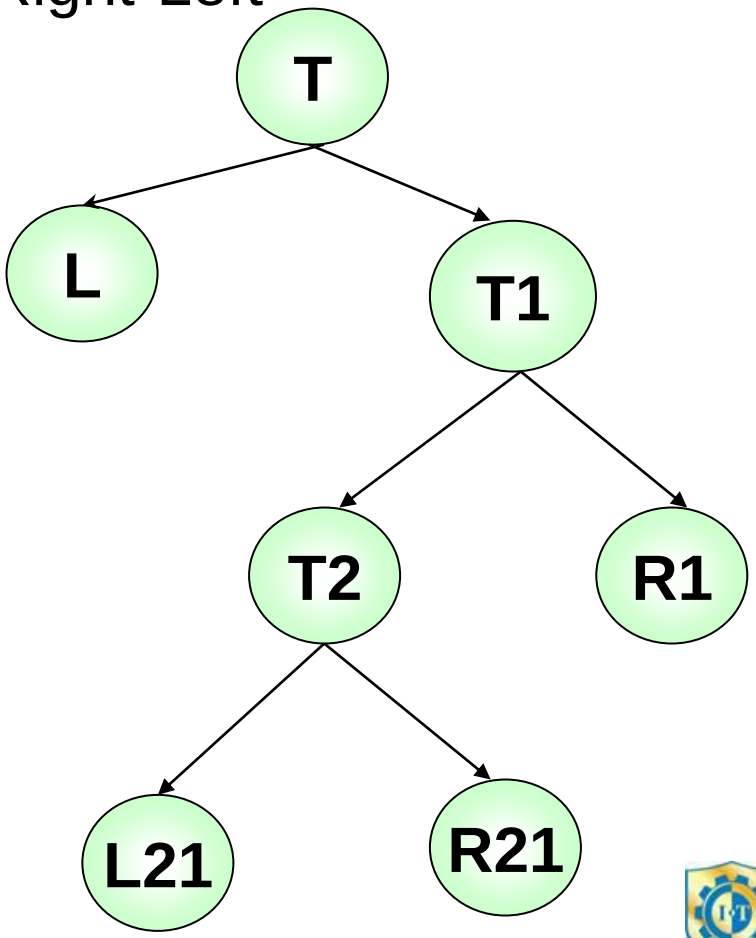
Các trường hợp mất cân bằng do lệch phải

- Cây mất cân bằng tại nút T

TH3: Right-Right



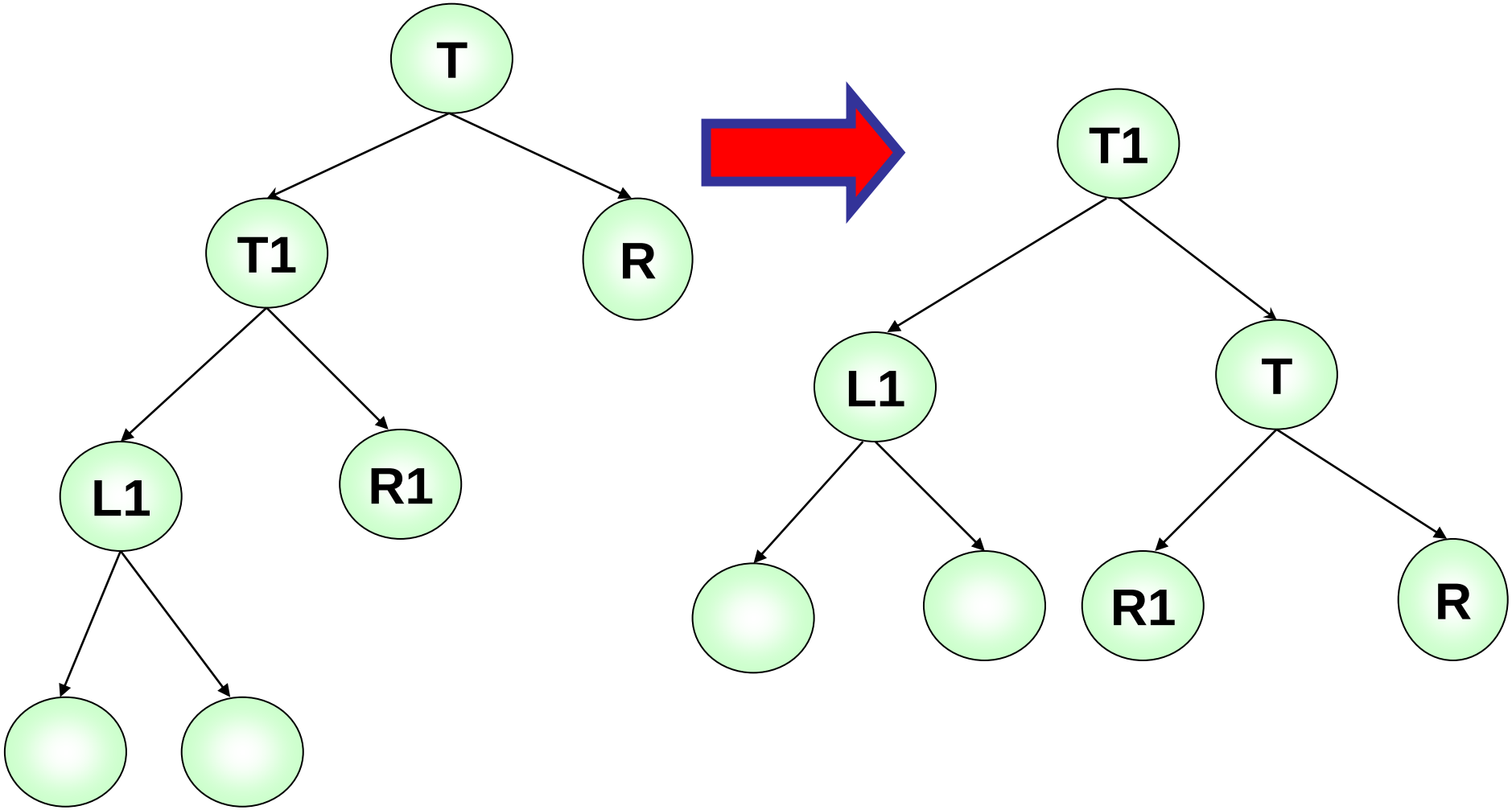
TH4: Right-Left



Các thao tác trên cây cân bằng

- Khi thêm hay xoá 1 nút trên cây, cĩ 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

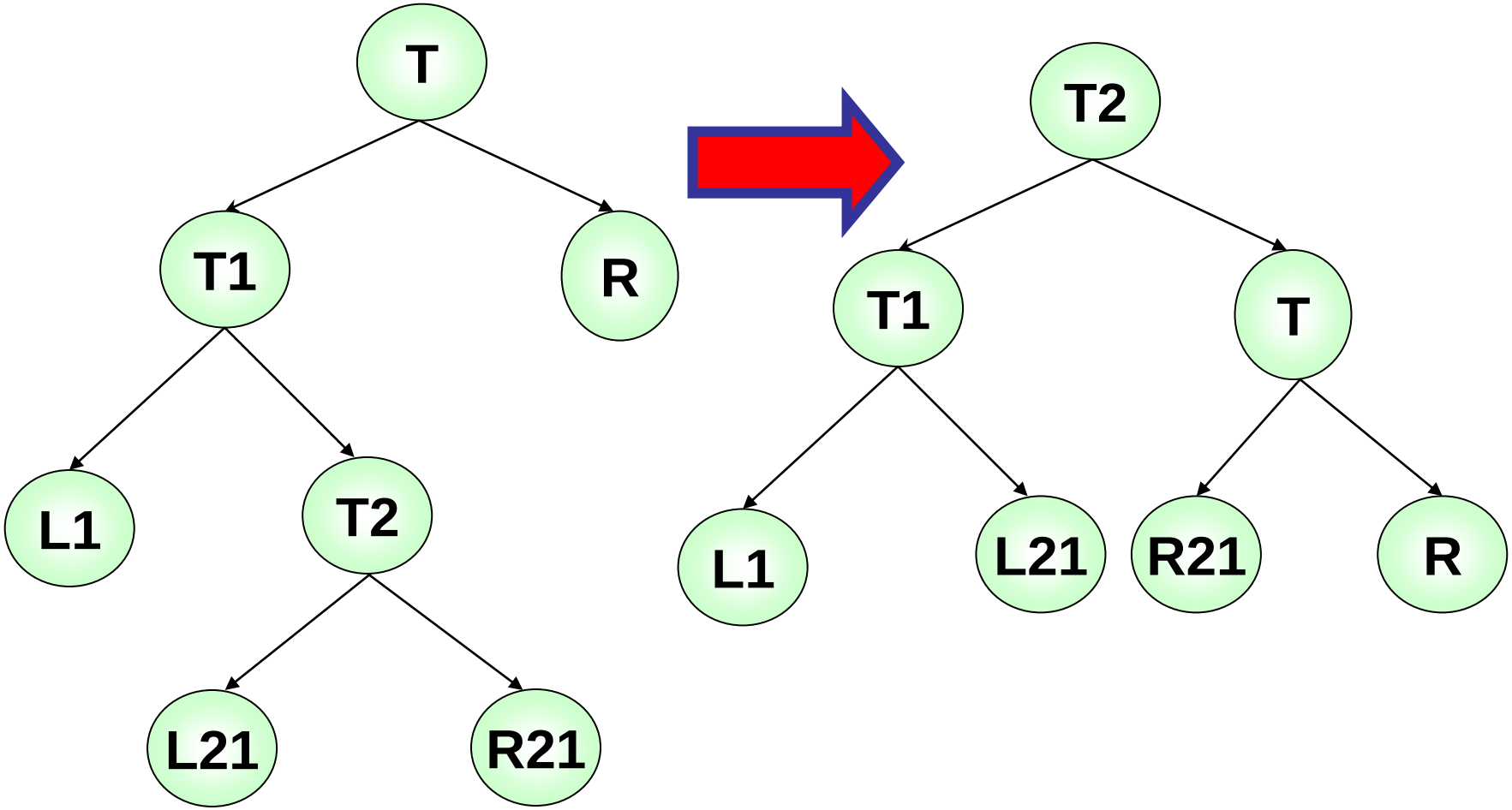
Cân bằng lại trường hợp 1



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;
}
```

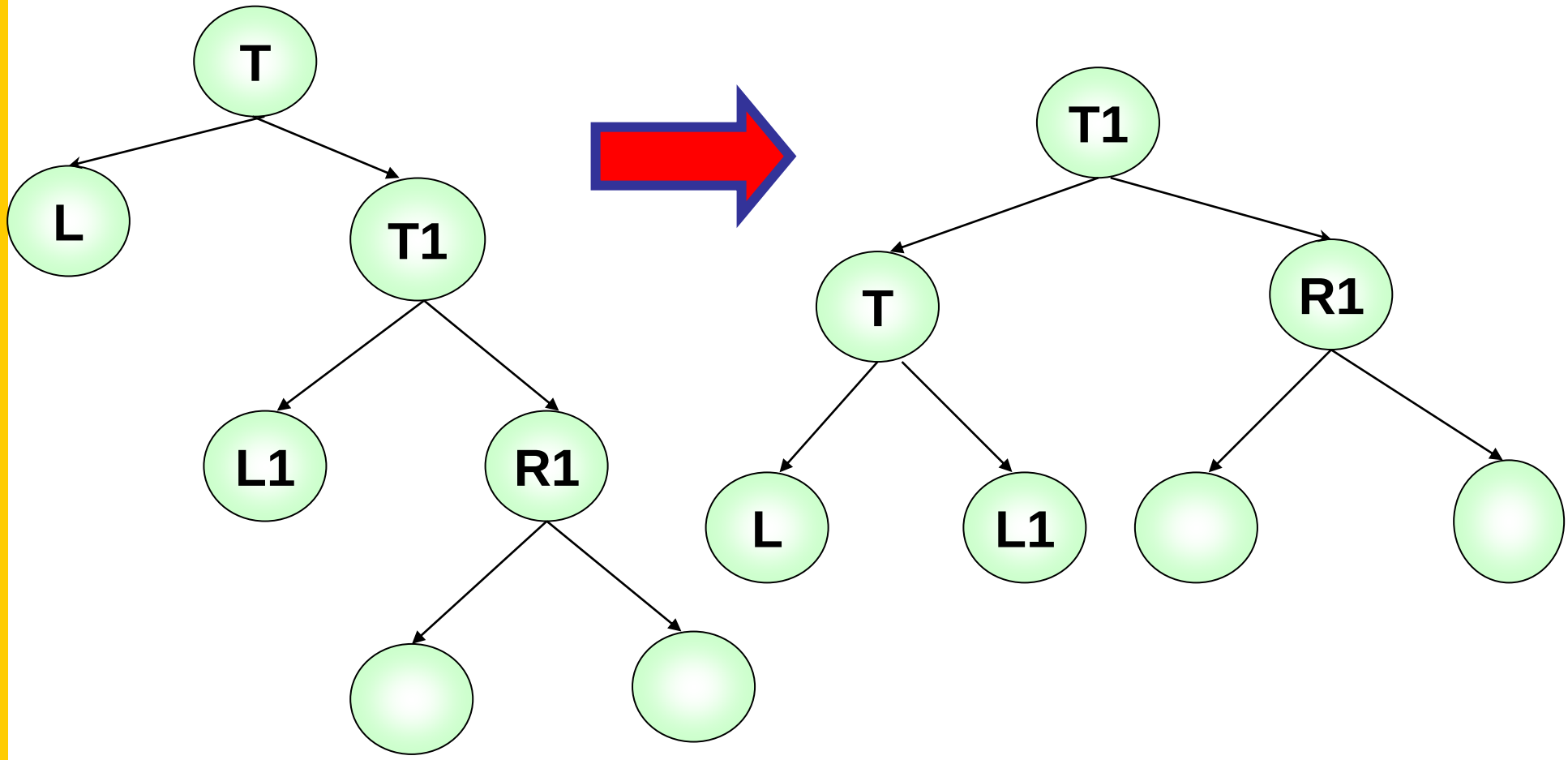

Cân bằng lại trường hợp 2



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;}
```

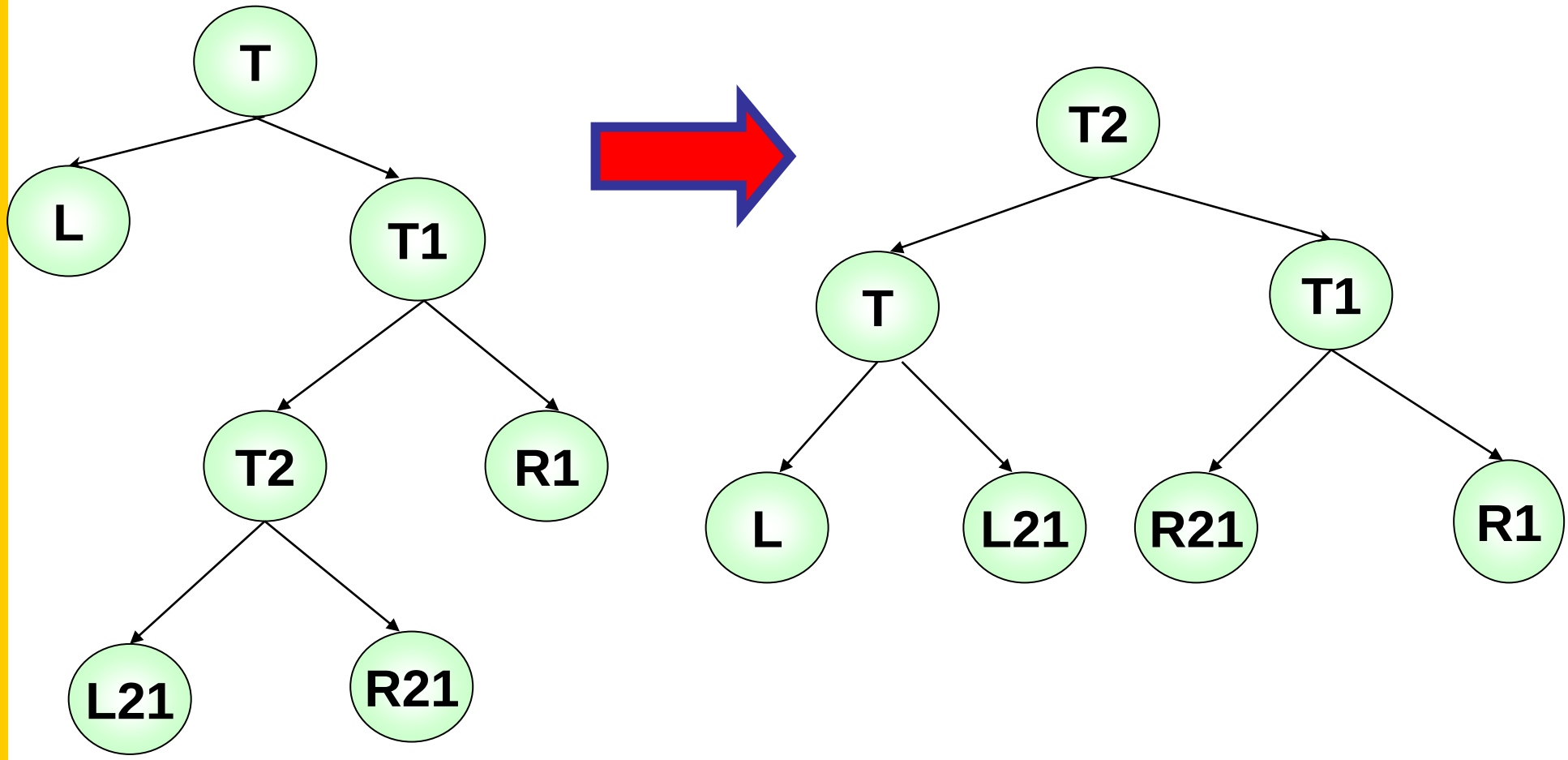
Cân bằng lại trường hợp 3



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



Cài đặt cân bằng lại cho trường hợp 4

```
void RR(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 1 nút

- Thêm bình thường như trường hợp cây NPTK
- Nếu cây tăng trưởng chiều cao
  - Lần 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:
  - Lần 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 lần ngược lên nút cha…
- Việc cân bằng lại co thể lan truyền lên tận gốc

Câu hỏi và Bài tập

1. Hãy trình bày định nghĩa, đặc điểm và hạn chế của cây nhị phân tìm kiếm.

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

8 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.

Câu hỏi và Bài tập

3. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm.

4. Cài đặt chương trình mô phỏng trực quan các thao tác trên cây nhị phân tìm kiếm cân bằng.

5. Viết chương trình cho phép tạo, tra cứu và sửa chữa từ điển Anh-Việt.