


Chương 4.  
Tìm kiếm (tiếp)

nguyenduyhiiep@gmail.com  
hiepnid@soict.hut.edu.vn

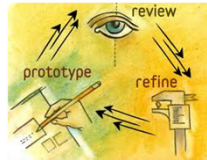
Cây tìm kiếm nhị phân cân bằng  
AVL Tree

G. M. ADELSON-VELSKII và E. M. LANDIS

### Tìm kiếm

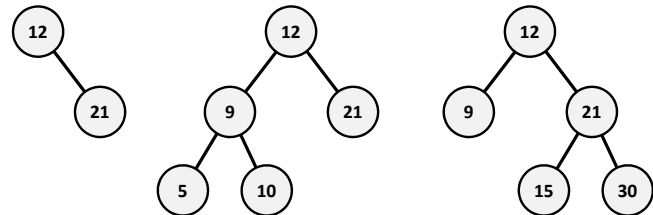


- Đặc điểm của cấu trúc cây tìm kiếm nhị phân
  - Kiểu cấu trúc liên kết
  - Thao tác tìm kiếm, thêm, xóa thực hiện dễ dàng
  - Thời gian thực hiện các thao tác trong trường hợp tốt nhất  $O(\log n)$ , tồi nhất  $O(n)$
  - Trường hợp tồi khi cây bị suy biến
  - Cây cân bằng cho thời gian thực hiện tốt nhất
- Cải tiến cấu trúc cây tìm kiếm nhị phân để luôn thu được thời gian thực hiện tối ưu



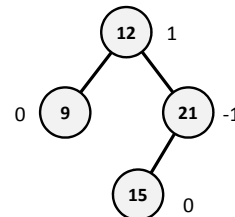
### AVL tree

- Cây tìm kiếm nhị phân cân bằng – AVL tree:
  - Là cây tìm kiếm nhị phân
  - Chiều cao của cây con trái và cây con phải của gốc chênh nhau không quá 1
  - Cây con trái và cây con phải cũng là các cây AVL



## AVL tree

- Quản lý trạng thái cân bằng của cây
  - Mỗi nút đưa thêm 1 thông tin là hệ số cân bằng (balance factor) có thể nhận 3 giá trị
    - Left\_higher (hoặc -1)
    - Equal\_height (hoặc 0)
    - Right\_higher (hoặc +1)

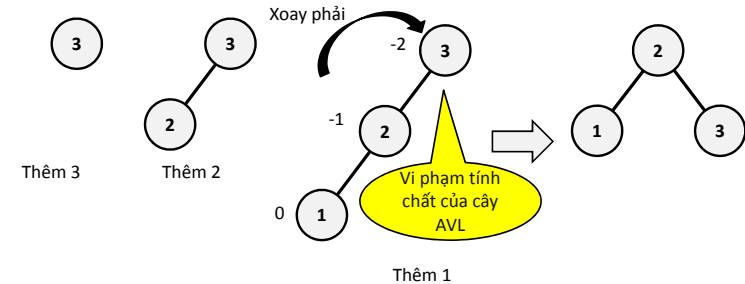


- Hai thao tác làm thay đổi hệ số cân bằng của nút:
  - Thêm nút
  - Xóa nút



## AVL tree

- Thêm các nút 3, 2, 1, 4, 5, 6, 7 vào cây AVL ban đầu rỗng



Xử lý bằng phép xoay nút



## AVL tree

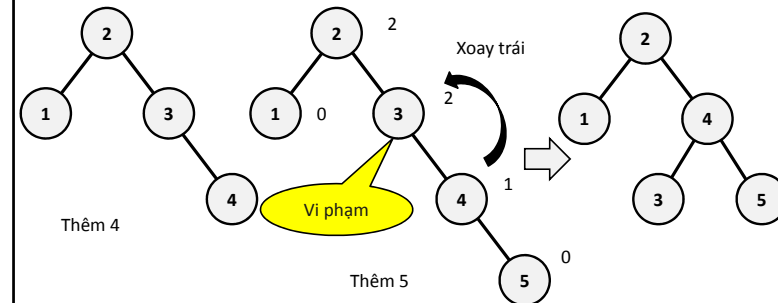
- Khai báo cấu trúc 1 nút cây AVL

```
enum Balance_factor { left_higher, equal_height, right_higher };
```

```
typedef struct AVLNode
{
    int data;
    Balance_factor balance;
    struct TreeNode *leftChild;
    struct TreeNode *rightChild;
} AVLNODE;
```

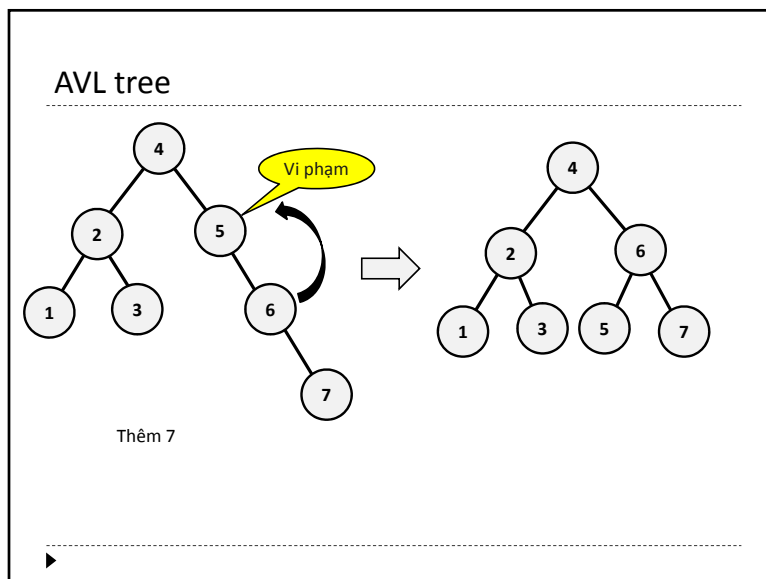
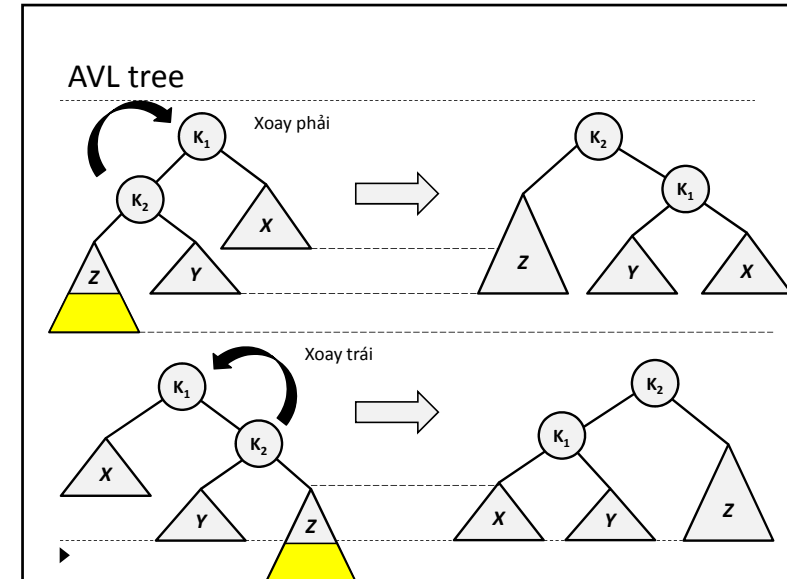
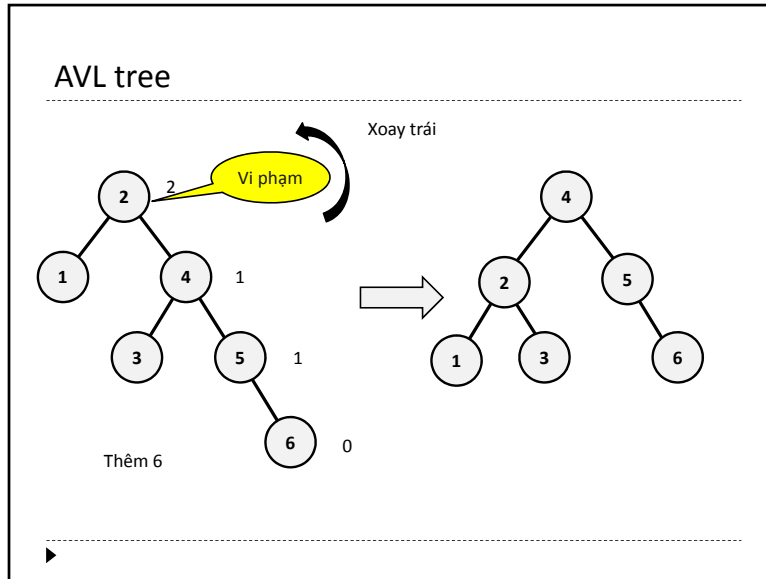


## AVL tree



Xoay giữa nút vi phạm và nút con của nó





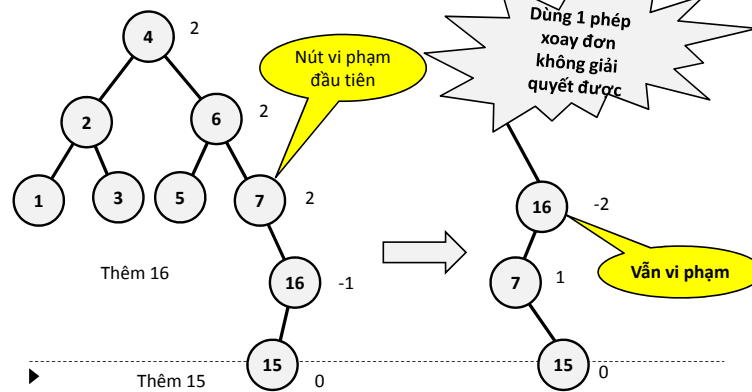
## AVL tree

### Phép xoay đơn – single rotation:

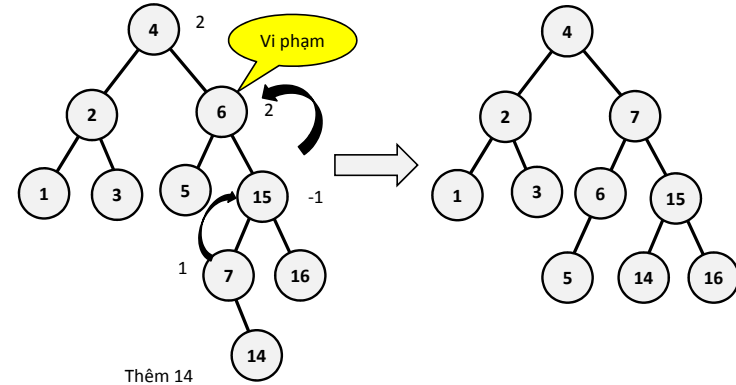
- ▶ Dùng để điều chỉnh khi mà nút mới thêm vào trong trường hợp:
  - ▶ (i) Cây con trái của nút con trái, hoặc
  - ▶ (ii) Cây con phải của nút con phải của nút
- ▶ Thực hiện tại nút vi phạm đầu tiên trên đường từ vị trí mới thêm trở về gốc
- ▶ Xoay giữa nút vi phạm và nút con trái (xoay phải) – TH i) (hoặc con phải (xoay trái) – TH ii)
- ▶ Sau khi xoay các nút trở nên cân bằng

### AVL tree

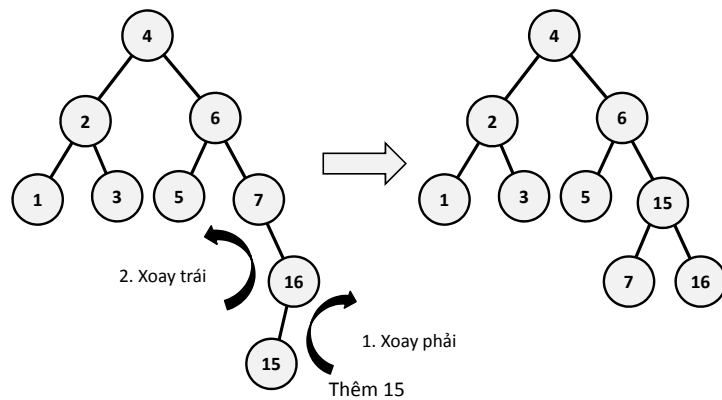
- Thực hiện thêm tiếp các khóa 16, 15, 14, 13, 12, 11, 10, 8, 9 vào cây



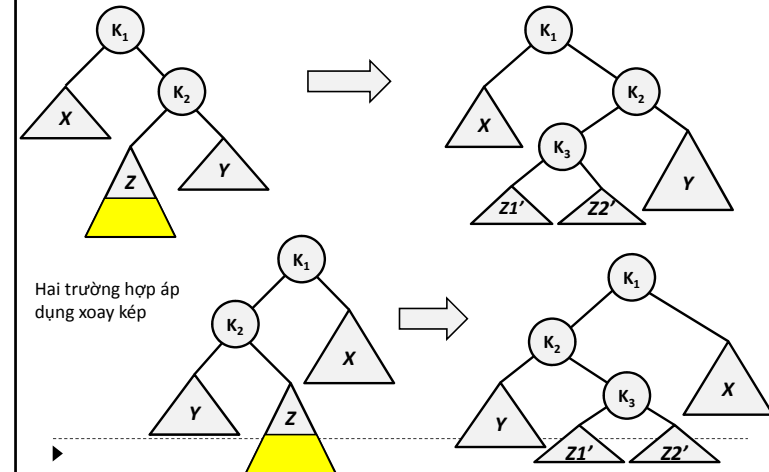
### AVL tree

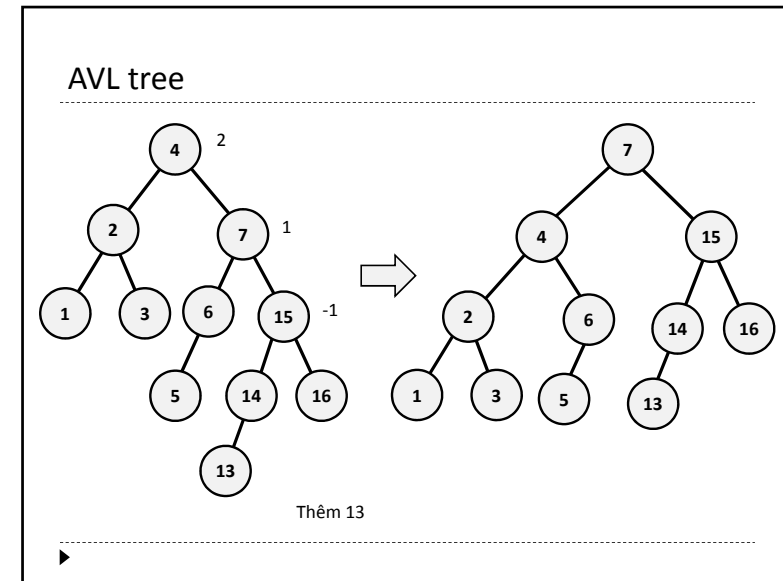
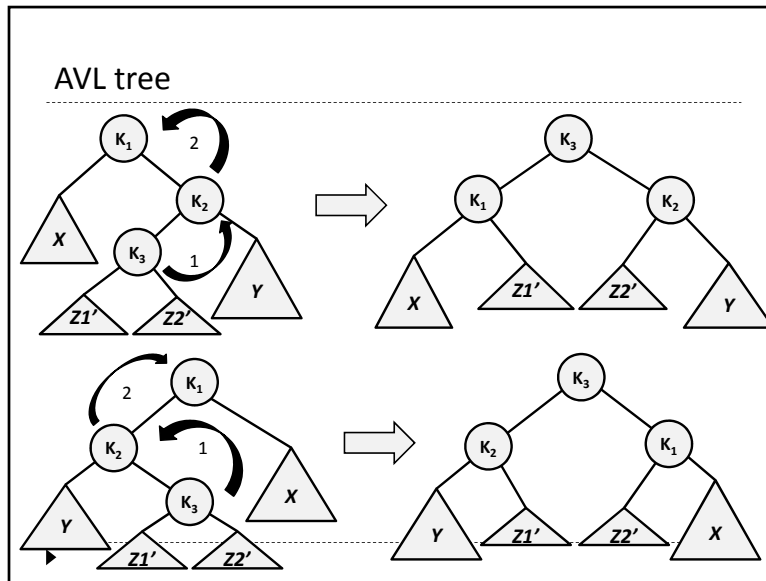


### AVL tree



### AVL tree



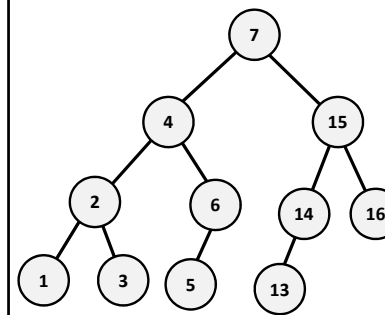


## AVL tree

### Phép xoay kép – double rotation:

- ▶ Dùng để điều chỉnh khi mà nút mới thêm vào trong trường hợp:
  - ▶ (i) Cây con phải của nút con trái, hoặc
  - ▶ (ii) Cây con trái của nút con phải của nút
- ▶ Thực hiện tại nút vi phạm đầu tiên trên đường từ vị trí mới thêm trở về gốc
- ▶ Xoay giữa nút vi phạm, nút con, và nút cháu (con của nút con)
- ▶ Xoay kép gồm 2 phép xoay trái và xoay phải
- ▶ Số nút trong quá trình thực hiện xoay là 3

## AVL tree



Thêm 12

AVL tree

---

Thêm 11



AVL tree

---

Thêm 8



AVL tree

---

Thêm 10



AVL tree

---

Thêm 9



## AVL tree

- ▶ Mỗi phép xoay có 2 trường hợp, khi cài đặt sẽ phải có 4 trường hợp
  - ▶ Trái – trái (xoay đơn)
  - ▶ Phải – phải (xoay đơn)
  - ▶ Trái – phải (xoay kép)
  - ▶ Phải – trái (xoay kép)
- ▶ Sau mỗi lần xoay, trạng thái cân bằng lại được xác lập lại tại nút vi phạm



## AVL tree

```
void rotate_right(AVLNODE *&root)
{
    if(root==NULL || root->leftChild==NULL)
        //error, because it's impossible
    {
        printf("It's must be a mistake when using this function!\n");
    }
    else
    {
        AVLNODE *pLeft = root->leftChild;
        root->leftChild = pLeft->rightChild;
        pLeft->rightChild = root;
        root = pLeft;
    }
}
```



## AVL tree

```
//2 single rotations
void rotate_left(AVLNODE *&root)
{
    if(root==NULL || root->rightChild==NULL)
        //error, because it's impossible
    {
        printf("It's must be a mistake when using this function!\n");
    }
    else
    {
        AVLNODE *pRight = root->rightChild;
        root->rightChild = pRight->leftChild;
        pRight->leftChild = root;
        root = pRight;
    }
}
```



## AVL tree

```
void left_balance(AVLNODE *&root)
//balance function for insert in left subtree
{
    AVLNODE *pLeft = root->leftChild;
    if(pLeft->balance == equal_height)
    {
        printf("It's must be a mistake when using this function!\n");
    }
    else if(pLeft->balance == left_higher)
        //left-left case (single rotation)
    {
        root->balance = equal_height;
        pLeft->balance = equal_height;
        rotate_right(root);
    }
    else
        //left-right case (double rotation:(1)rotate left,(2)rotate right)
```



```

{
    AVLNODE *pLeftRight = root->leftChild->rightChild;
    if(pLeftRight->balance == left_higher)
    {
        pLeft->balance = equal_height;
        root->balance = right_higher;
    }
    else if(pLeftRight->balance == equal_height)
    {
        pLeft->balance = equal_height;
        root->balance = equal_height;
    }
    else
    {
        pLeft->balance = left_higher;
        root->balance = equal_height;
    }

    pLeftRight->balance = equal_height;
    rotate_left(pLeft);
    root->leftChild = pLeft;
    rotate_right(root);
}
}

```

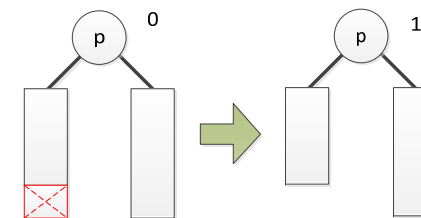
## AVL tree

- ▶ Xóa nút khỏi cây:
  - ▶ Chuyển bài toán xóa nút đầy đủ thành xóa nút có nhiều nhất một con.
  - ▶ Xóa nút có nhiều nhất một con bị xóa làm chiều cao của nhánh bị giảm
  - ▶ Căn cứ vào trạng thái cân bằng tại các nút từ nút bị xóa trên đường trở về gốc để cân bằng lại cây nếu cần (giống với khi thêm một nút mới vào cây)

## AVL tree

- ▶ Xóa nút khỏi cây:
  - ▶ Nếu nút cần xóa là nút đầy đủ: chuyển về xóa nút có nhiều nhất 1 nút con
    - ▶ Thay thế nút cần xóa bằng nút phải nhất trên cây con trái
    - ▶ hoặc , nút trái nhất trên cây con phải
    - ▶ Copy các thông số của nút thay thế giống với thông số của nút bị xóa thực sự
  - ▶ Nếu nút bị xóa là nút có 1 con: thay thế nút đó bằng nút gốc của cây con
  - ▶ Nếu nút bị xóa là nút lá: gỡ bỏ nút, gán con trở của nút cha nó bằng NULL

## AVL tree

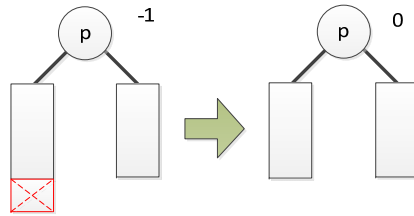


Chiều cao cây không đổi

**Trường hợp 1:** nút p đang ở trạng thái cân bằng (equal)  
Xóa một nút của cây con trái (hoặc phải) làm cây bị lệch nhưng chiều cao không đổi



## AVL tree



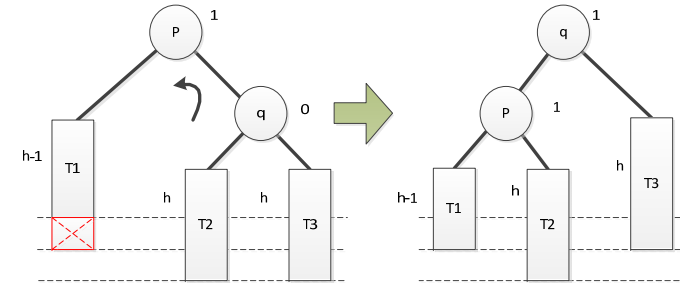
Chiều cao cây thay đổi

**Trường hợp 2:** nút p đang ở trạng thái lệch trái hoặc phải  
Nút bị xóa là nút của nhánh cao hơn, sau khi xóa cây trở về trạng thái cân bằng và chiều cao của cây giảm



## AVL tree

Trường hợp 3.1



Chiều cao cây không đổi



## AVL tree

- ▶ **Trường hợp 3:** cây đang bị lệch và nút bị xóa nằm trên nhánh thấp hơn.
  - ▶ Để cân bằng lại cây ta phải thực hiện các phép xoay.
- Căn cứ vào tình trạng cân bằng của nút con còn lại q của p mà ta chia thành các trường hợp nhỏ sau:

**Trường hợp 3.1:** Nút q đang ở trạng thái cân bằng

- ▶ Thực hiện phép xoay đơn (xoay trái hoặc xoay phải)
- ▶ Sau khi xoay, p trở về trạng thái cân bằng

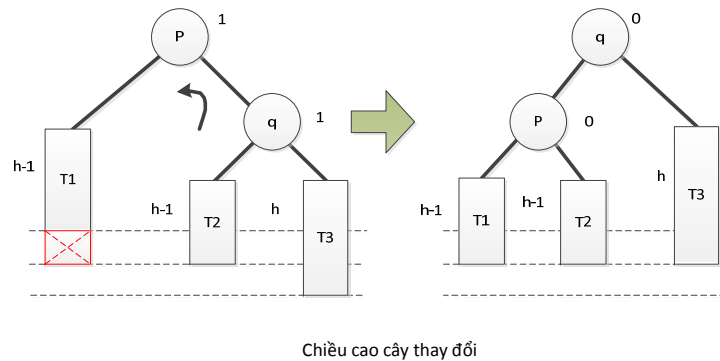


## AVL tree

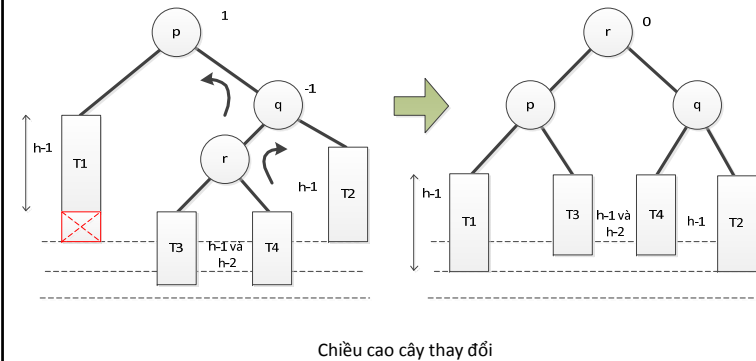
- ▶ **Trường hợp 3.2:** nút q bị lệch trái (nếu q là con phải của p) hoặc lệch phải (nếu q là con trái của p)
- ▶ Cân bằng p bằng cách thực hiện phép xoay đơn giữa q và p
- ▶ Sau khi xoay p trở về trạng thái cân bằng và chiều cao của p bị giảm đi



### AVL tree



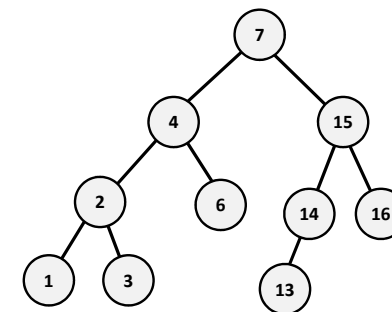
### AVL tree



### AVL tree

- ▶ Trường hợp 3.3: nút q bị lệch cùng phía với nhánh bị xóa. Nếu nhánh bị xóa là nhánh trái của p thì q bị lệch trái và ngược lại
- ▶ Để tái cân bằng cho p ta phải thực hiện 2 phép xoay giữa nút con của q, nút q, và nút p
- ▶ Sau khi xoay, chiều cao của cây giảm đi, p trở về trạng thái cân bằng

### AVL tree



Xóa một trong các nút 4, 7, 15 trên cây AVL

# Question



## Cây trúc tự điều chỉnh

- ▶ Trong nhiều bài toán chúng ta cần một cấu trúc xử lý hiệu quả với những truy cập có số lượng lớn trên các bản ghi mới đưa vào.
- ▶ Ví dụ: bài toán quản lý thông tin bệnh nhân tại bệnh viện
  - ▶ Bệnh nhân ra khỏi bệnh viện thì có số lần truy cập thông tin ít hơn
  - ▶ Bệnh nhân mới vào viện thì sẽ có số lượng truy cập thông tin thường xuyên
  - ▶ Ta cần cấu trúc mà có thể tự điều chỉnh để đưa những bản ghi mới thêm vào ở gần gốc để cho việc truy cập thường xuyên dễ dàng.



Splay tree  
Cấu trúc tự điều chỉnh

## Cây splay

- ▶ **Splay tree**
  - ▶ Là cây tìm kiếm nhị phân
  - ▶ Mỗi khi truy cập vào một nút trên cây (thêm, hoặc xóa) thì nút mới truy nhập sẽ được tự động chuyển thành gốc của cây mới
  - ▶ Các nút được truy cập thường xuyên sẽ ở gần gốc
  - ▶ Các nút ít được truy cập sẽ bị đẩy xa dần gốc
- ▶ Để dịch chuyển các nút ta dùng các phép xoay giống với trong AVL tree
- ▶ Các nút nằm trên đường đi từ gốc đến nút mới truy cập sẽ chịu ảnh hưởng của các phép xoay

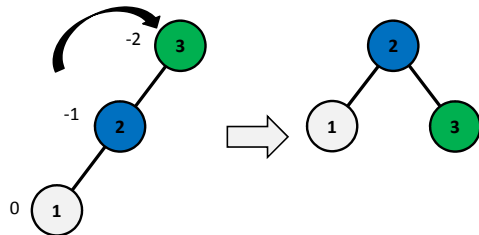


### Cây splay

#### ► Nhắc lại về các phép xoay

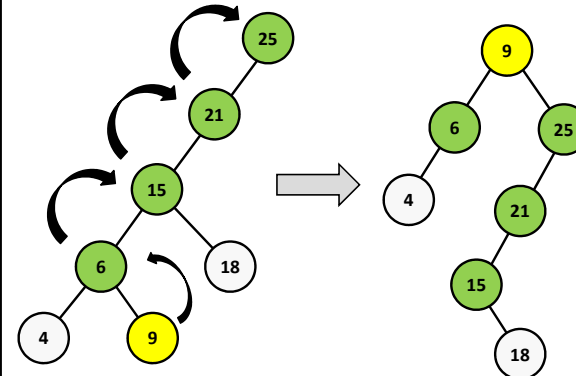
#### ► Xoay đơn – single rotation:

- Nút cha xuống thấp 1 mức và nút con lên 1 mức



### Cây splay

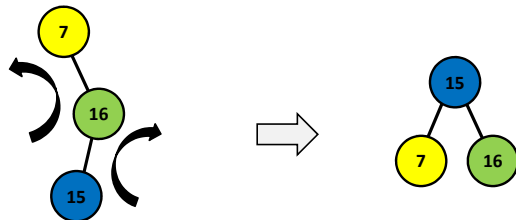
#### ► Trường hợp chỉ dùng phép xoay đơn để điều chỉnh nút



### Cây splay

#### ► Xoay kép – double rotation:

- gồm 2 phép xoay đơn liên tiếp.
- Nút tăng lên 1 mức, còn các nút còn lại lên hoặc giảm xuống nhiều nhất 1 mức



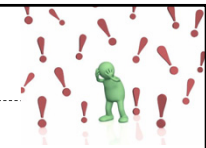
### Cây splay

#### ► Nhận xét:

- Nút mới truy cập (nút 9) được chuyển thành nút gốc của cây mới
- Tuy nhiên nút 18 lại bị đẩy xuống vị trí của nút 9 trước

#### ► Như vậy:

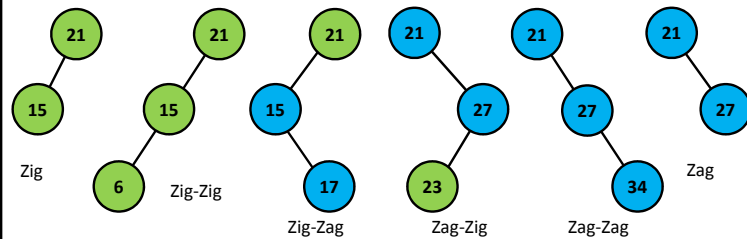
- Truy cập tới 1 nút sẽ đẩy các nút khác xuống sâu hơn.
- Tốc độ của nút bị truy cập được cải thiện nhưng không cải thiện tốc độ truy cập của các nút khác trên đường truy cập
- Thời gian truy cập với  $m$  nút liên tiếp vẫn là  $O(m * n)$
- Ý tưởng dùng chỉ phép xoay đơn để biến đổi cây là **không đủ tốt**



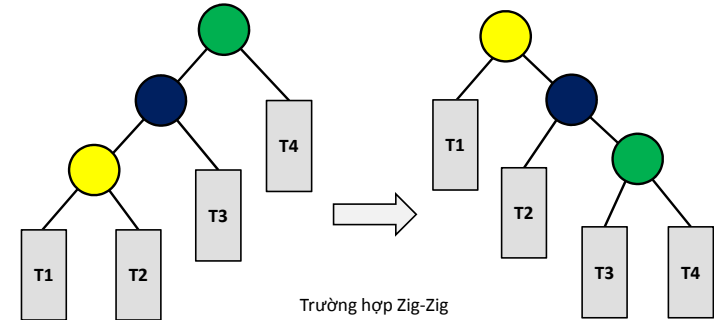
## Cây splay

### Ý tưởng mới:

- Tại mỗi bước ta di chuyển nút liên 2 mức
- Xét các nút trên đường đi từ gốc đến nút mới truy nhập
  - Nếu ta di chuyển trái (từ gốc xuống), ta gọi là Zig
  - Ngược lại, di chuyển phải ta gọi là Zag



## Cây splay

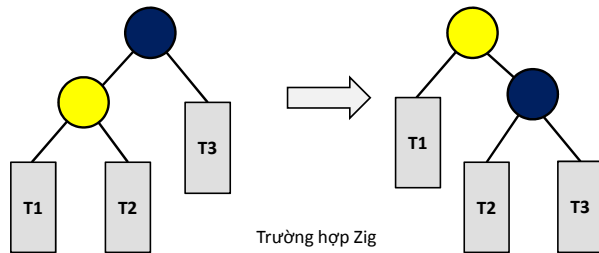


Trường hợp Zig-Zig

## Cây splay

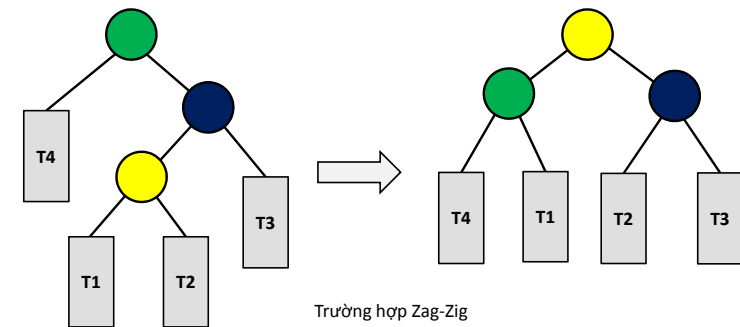
### Dịch chuyển:

- Nếu nút đang xét nằm ở mức sâu hơn hoặc bằng 2 ta dịch chuyển 2 mức mỗi lần
- Nếu nút ở mức 1: ta chỉ dịch chuyển 1 mức (trường hợp Zig hoặc Zag)



Trường hợp Zig

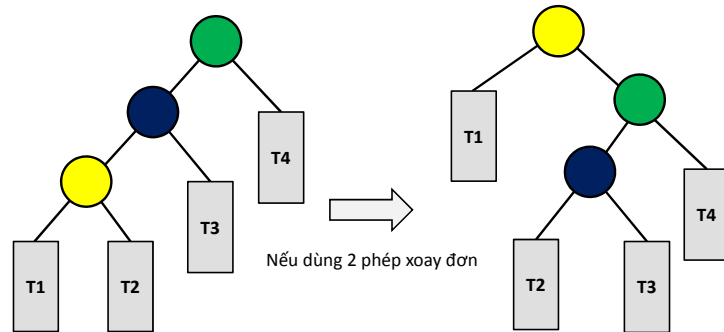
## Cây splay



Trường hợp Zag-Zig

### Cây splay

- Chú ý: trường hợp Zig-Zig (hoặc Zag-Zag) khác hoàn toàn với trường hợp dùng hai phép xoay đơn liên tiếp

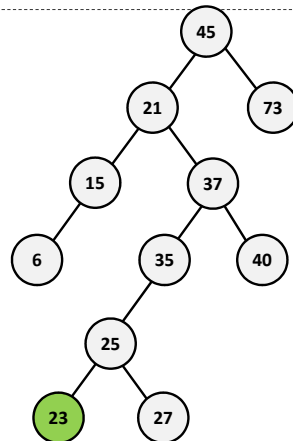


### Cây splay

- Nhận xét về cây splay:
  - Cây không cân bằng (thường bị lệch)
  - Các thao tác có thời gian thực hiện khác nhau từ  $O(1)$  tới  $O(n)$
  - Thời gian thực hiện trung bình của một thao tác trong một chuỗi thao tác là  $O(\log n)$
  - Thực hiện giống như cây AVL nhưng không cần quản lý thông tin về trạng thái cân bằng của các nút

### Cây splay

- Thực hiện splay tại nút 23



Cây 2-3

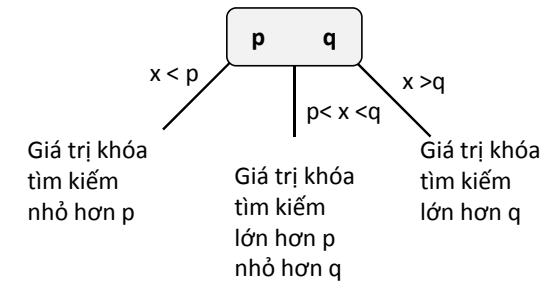
### Cây 2-3

- ▶ Đảm bảo cây luôn luôn cân bằng
  - ▶ Chi phí thực hiện các thao tác luôn là  $O(\log n)$
- ▶ Cây 2-3:
  - ▶ Mỗi nút trong có 2 tới 3 nút con
  - ▶ Nút lá có 1 tới 2 giá trị
  - ▶ Dữ liệu được lưu trên nút lá hoặc nút trong
  - ▶ ĐÂY KHÔNG PHẢI CÂY NHỊ PHÂN
  - ▶ Trạng thái cân bằng của cây được duy trì dễ dàng hơn so với cây AVL



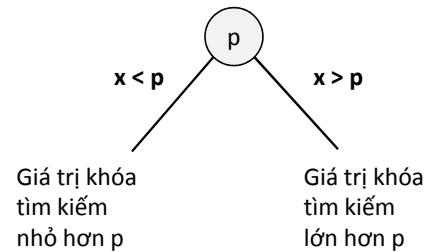
### Cây 2-3

- ▶ Nút trong có 3 con – nút 3
  - ▶ Nút chứa 2 phần tử
  - ▶ Có 3 nút con

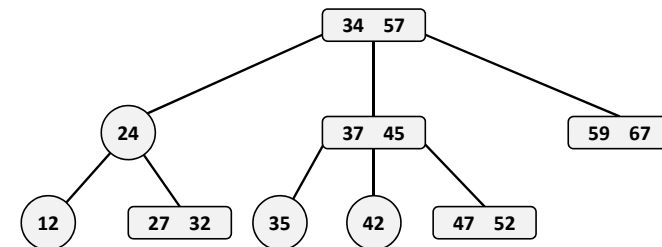


### Cây 2-3

- ▶ Nút trong có 2 con – nút 2
  - ▶ Nút chứa 1 phần tử
  - ▶ Có 2 nút con



### Cây 2-3



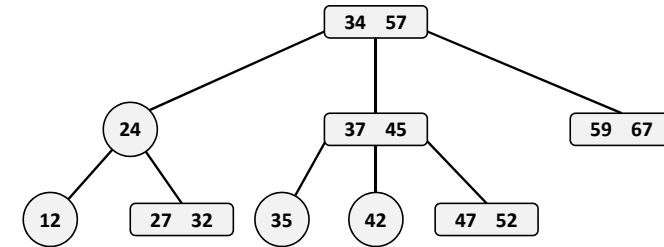
### Cây 2-3

- Định nghĩa cấu trúc 1 nút

```
struct TreeNode
{
    DATA_TYPE smallItem, largeItem;
    struct TreeNode *left, *middle, *right;
    struct TreeNode *parent; //to make your life easier
}
```



### Cây 2-3

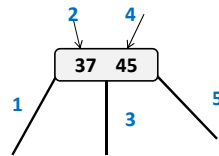


- Duyệt cây theo thứ tự giữa:  
12, 24, 27, 32, 34, 35, 37, 42, 45, 47, 52, 57, 59, 67



### Cây 2-3

- Duyệt cây theo thứ tự giữa – in-order traversal
  - (1) Duyệt cây con trái
  - (2) Xử lý nội dung khóa nhỏ hơn tại nút
  - (3) Duyệt cây con giữa
  - (4) Xử lý nội dung khóa lớn hơn tại nút
  - (5) Duyệt cây con phải



### Cây 2-3

- Tìm kiếm
  - Nếu nút hiện tại rỗng → không tìm thấy
  - Nếu giá trị tìm kiếm k xuất hiện trên nút hiện tại → tìm thấy
  - Nếu  $k <$  giá trị khóa nhỏ hơn → tìm tiếp tại cây con trái
  - Nếu khóa nhỏ hơn  $< k <$  khóa lớn hơn → tìm tại cây con giữa
  - Ngược lại → tìm tại cây con phải



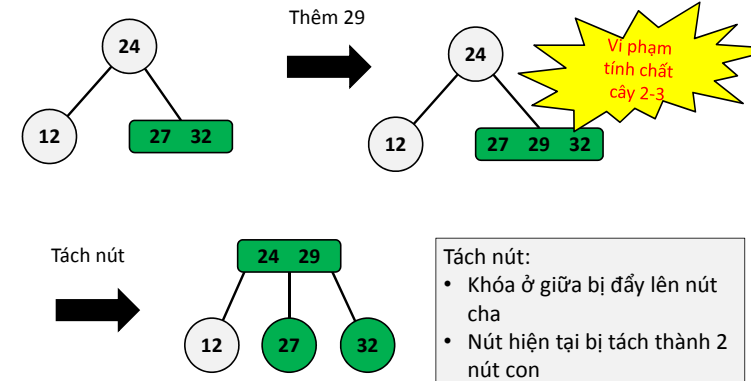


### Cây 2-3

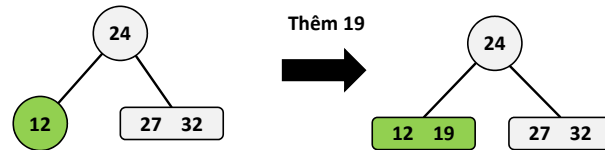
#### ► Thêm nút

- Phần tử mới được thêm vào tại nút lá của cây
- Nếu nút lá sau khi thêm có 3 phần tử ta phải thực hiện thao tác tách nút
- Khi tách nút, khóa giữa bị đẩy lên nút cha
- Tách nút tại nút lá có thể dẫn đến tách nút tại nút trong

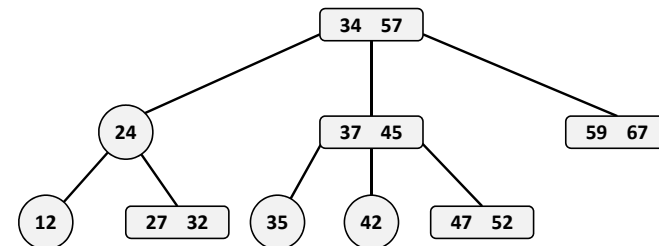
### Cây 2-3



### Cây 2-3



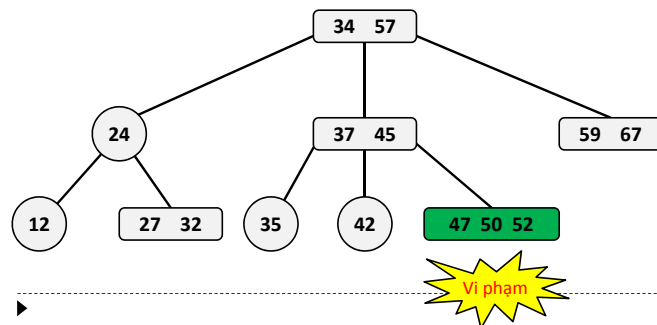
### Cây 2-3



Thêm nút 50 ?

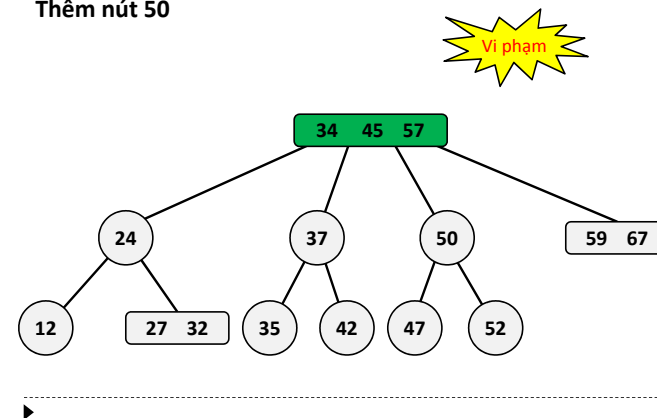
### Cây 2-3

Thêm nút 50



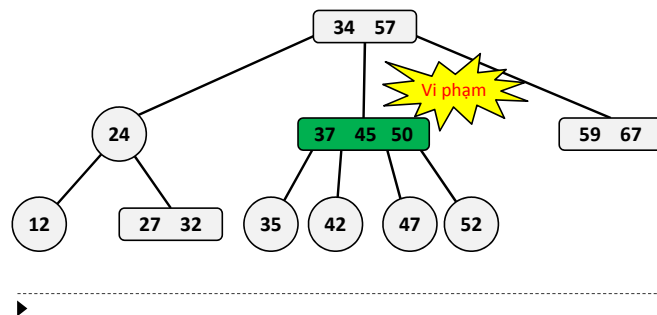
### Cây 2-3

Thêm nút 50



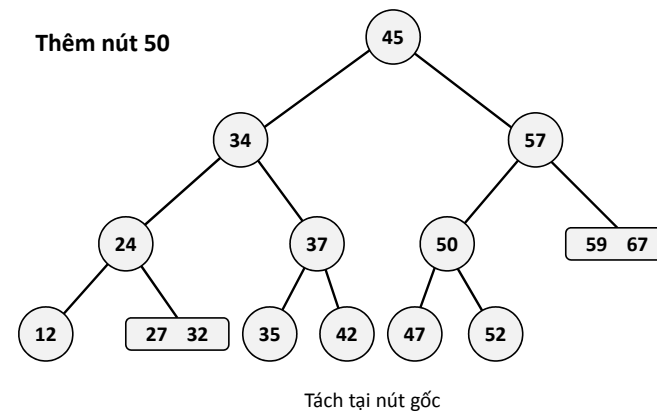
### Cây 2-3

Thêm nút 50

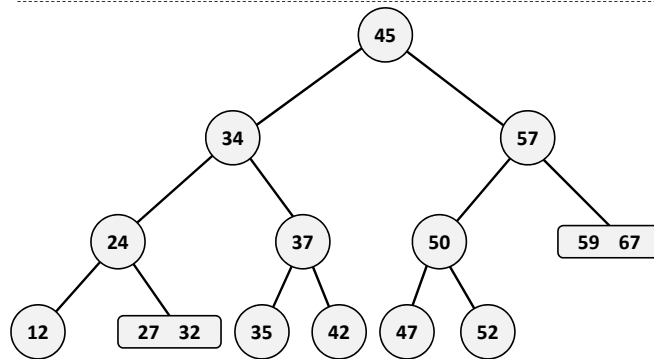


### Cây 2-3

Thêm nút 50



### Cây 2-3



Vẽ cây thu được sau khi thêm lần lượt các khóa  
5, 7, 29, 31, 70, 75 vào cây trên



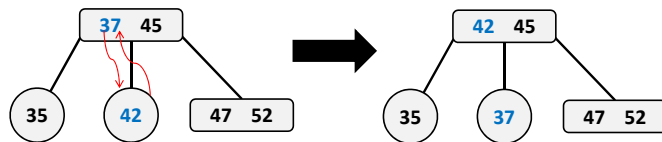
### Cây 2-3

- ▶ Xóa nút lá
  - ▶ Nếu nút lá sau khi xóa vẫn còn phần tử → kết thúc
  - ▶ Ngược lại ta phải dịch chuyển phần tử từ nút anh em của nó, hoặc từ nút cha của nó
    - ▶ (i) Nếu nút anh em của nó có 2 phần tử thì dịch một phần tử sang nút hiện tại
    - ▶ (ii) Nếu không thực hiện dịch được thì ta sẽ thực hiện kết hợp (điều này có thể dẫn đến giảm chiều cao của cây)



### Cây 2-3

- ▶ Xóa nút:
  - ▶ Nếu phần tử bị xóa ở trên nút trong:
    - ▶ Phải chuyển phần tử đó về nút lá để xóa
    - ▶ Thay thế bằng nút kế tiếp trong duyệt theo thứ tự giữa

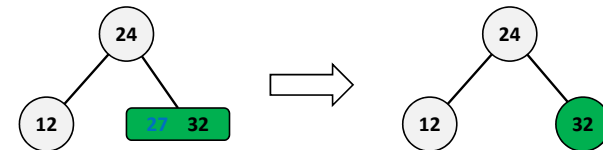


Xóa 37, ta thay bằng 42



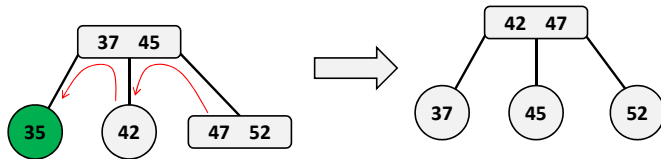
### Cây 2-3

- ▶ Xóa 27 (trường hợp nút lá sau khi xóa vẫn còn phần tử)



### Cây 2-3

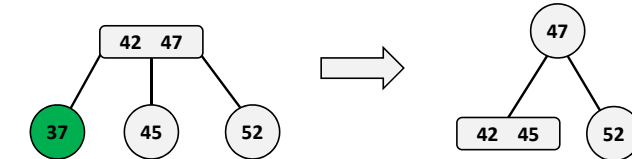
- Xóa 35 : trường hợp (i)



►

### Cây 2-3

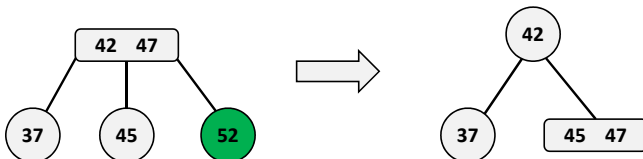
- Xóa 37: trường hợp (ii)



►

### Cây 2-3

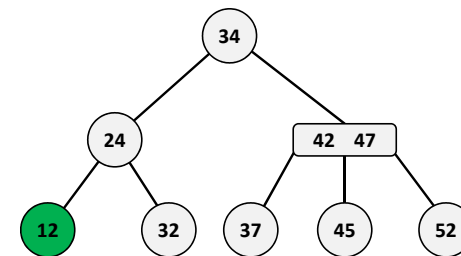
- Xóa 52: trường hợp (ii)



►

### Cây 2-3

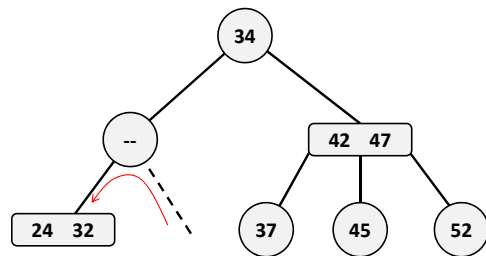
- Xóa 12: trường hợp (ii)



►

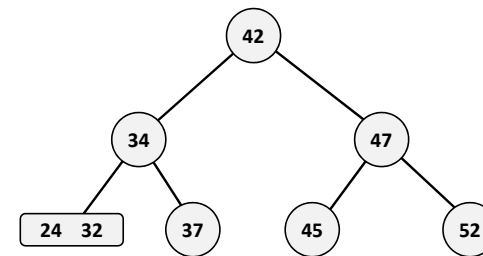
### Cây 2-3

- Xóa 12: trường hợp (ii)



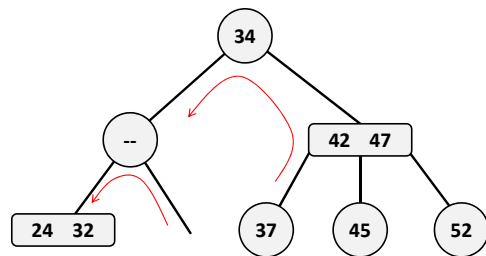
### Cây 2-3

- Xóa 12: trường hợp (ii)



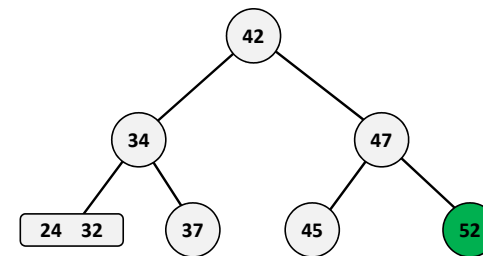
### Cây 2-3

- Xóa 12: trường hợp (ii)



### Cây 2-3

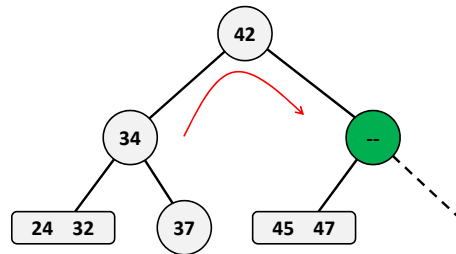
- Xóa 52:



### Cây 2-3

- ▶ Xóa 52:

**NOTE:** Khi không thực hiện được **địch chuyển** từ nút anh em thì ta thực hiện **kết hợp** các nút với nút cha của nó

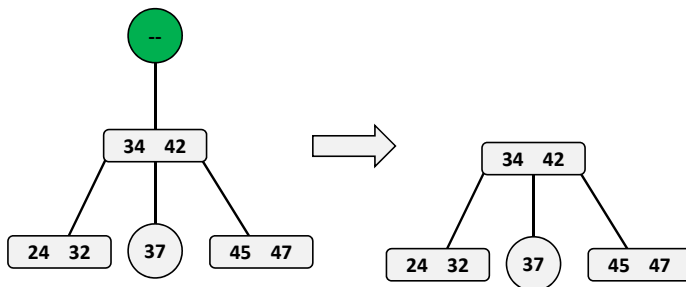


### Cây 2-3

- ▶ Thực hiện thêm lần lượt các nút sau vào cây 2-3 ban đầu rồi: 34, 65, 45, 23, 25, 76, 12, 9, 6, 48, 65, 5, 80, 7
- ▶ Với cây tạo được ở trên hãy xóa lần lượt các nút: 7, 9, 80, 23

### Cây 2-3

- ▶ Xóa 52:



# Question

