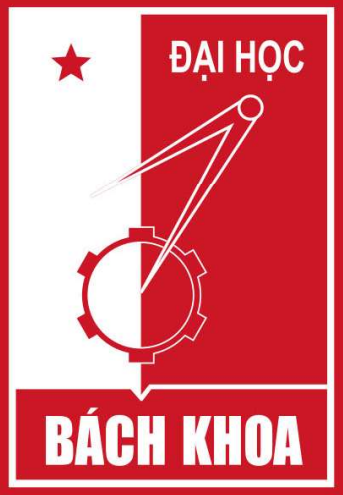


ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

Tìm kiếm

Phần 1. Thuật toán tìm kiếm

- Bài toán tìm kiếm
- Tìm kiếm tuần tự
- Tìm kiếm nhị phân
- Cây quyết định

Bài toán tìm kiếm

- Một số bài toán tìm kiếm:
  - Cho tên một người, tìm kiếm số điện thoại người đó trong danh bạ
  - Cho tên một tài khoản, tìm kiếm các giao dịch của tài khoản đó
  - Cho tên một sinh viên, tìm kiếm thông tin về kết quả học tập của sinh viên đó
- Khóa(key) là thông tin mà chúng ta dùng để tìm kiếm
- Tìm kiếm trong và tìm kiếm ngoài
  - Số lượng dữ liệu
  - Tốc độ truy nhập dữ liệu

Tìm kiếm tuần tự

Tìm kiếm tuần tự

- **Tìm kiếm tuần tự**:
  - Là phương pháp đơn giản nhất
  - Duyệt từ đầu đến cuối danh sách cho đến khi tìm được bản ghi mong muốn
  - Hoặc là đến cuối mà không tìm được

Tìm kiếm tuần tự

Tìm kiếm trên mảng

```
int sequentialSearch(int records[], int key, int &position)
{
    int i;
    for(i=0;i<MAX; i++)
    {
       if(records[i]==key)
       {
           position=i; //tim thay
           return 0;
       }
    }
    return -1;//khong tim thay
}
```

Tìm kiếm tuần tự

- Tìm kiếm trên danh sách móc nối

```
typedef struct node
{
    char hoten[30];
    char diachi[50];
    float diemTB;
    struct node *pNext;
} NODE;
```

Tìm kiếm tuần tự

```
int sequentialSearch(NODE *pHead, char key[], NODE *&retVal)
{
    NODE *ptr=pHead;
    while(ptr!=NULL)
    {
        if(strcmp(ptr->hoten,key)==0)
        {
            retVal=ptr;
            return 0;
        }
        else ptr=ptr->pNext;
    }
    return -1;
}
```

Phân tích

- Giả sử tồn tại bản ghi chứa khóa cần tìm.
- Trường hợp tốt nhất: chỉ cần 1 phép so sánh
- Trường hợp tồi nhất: cần n phép so sánh
- Trung bình cần :

$$\frac{1+2+3+...+n}{n}$$

- Độ phức tạp : $O(n)$

Chỉ phù hợp cho danh sách ít phần tử

Tìm kiếm tuần tự

- **Bài tập**: Viết lại hàm tìm kiếm tuần tự để có thể đếm được số lượng phép so sánh cần thiết trong quá trình tìm kiếm (đối với danh sách móc nối).

Tìm kiếm nhị phân

Tìm kiếm nhị phân

- Trường hợp các bản ghi đã được sắp xếp theo thứ tự, tìm kiếm tuần tự không hiệu quả
  - Tìm kiếm từ "study" trong từ điển
  - Tìm kiếm "Tran Van C" trong danh ba điện thoại

- Tìm kiếm nhị phân:
  - So sánh khóa với phần tử trung tâm danh sách, sau đó ta chỉ xét nửa trái hoặc nửa phải danh sách căn cứ vào khóa đứng trước hay sau phần tử trung tâm

Tìm kiếm nhị phân

- **Yêu cầu**: Danh sách phải được sắp xếp theo một thứ tự nào đó trước (danh sách có thứ tự).

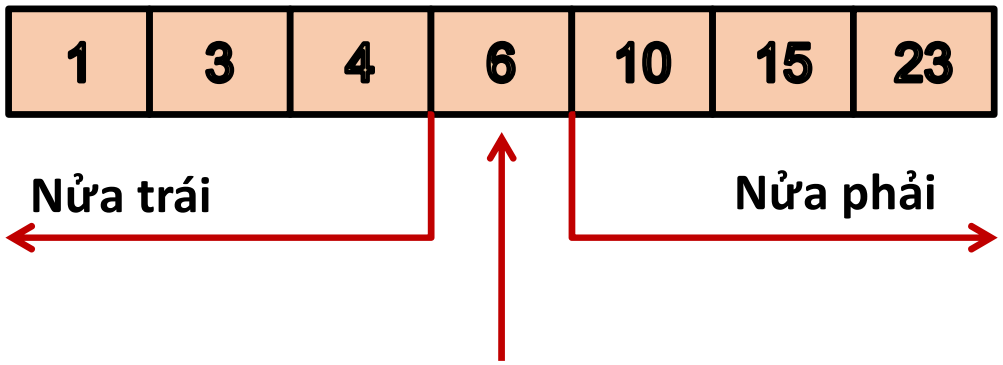
  Ví dụ:
  - các từ trong từ điển,
  - tên người trong danh bạ điện thoại

- **Danh sách có thứ tự**: là danh sách trong đó mỗi mục chứa một khóa theo thứ tự.

  Nếu mục ***i*** đứng trước ***j*** trong danh sách, thì khóa của mục ***i*** sẽ nhỏ hơn hoặc bằng khóa của mục ***j***

Tìm kiếm nhị phân

khóa **15**



Tìm kiếm nhị phân

```
int binarySearch1_Re(int A[], int key, int begin,
                                        int end, int &position)
{
    if(begin>end)return -1;
    int mid=(begin+end)/2;
    if(A[mid]==key)
    {
        position=mid;
        return 0;
    }
    if(key<A[mid]) return binarySearch(A,key,begin,
                                        mid-1,position);
    else return binarySearch(A,key,mid+1,end,position);
}
```

Tìm kiếm nhị phân

```
int binarySearch1(int A[], int key, int begin, int end,
                                                int &position)
{
    int mid;
    while(begin<=end)
    {
        mid=(begin+end)/2;
        if(A[mid]==key)
        {
            position=mid;
            return 0;
        }
        if(key<A[mid]) end=mid-1;
        else begin=mid+1;
    }
    return -1;
}
```

Ví dụ

VD1.Khi tìm kiếm bảng chứa 5000 dữ liệu bằng phép tìm kiếm nhị phân (binary search method), số lần so sánh bình quân sẽ là bao r

$Log_{10}2 = 0.3010$

A. 8
B. 9
C. 10
D. 11
E. 12

Ví dụ

- Trong tìm kiếm nhị phân, khi số lượng dữ liệu đã sắp xếp tăng gấp 4 lần thì số lượng phép so sánh tối đa tăng bao nhiêu?

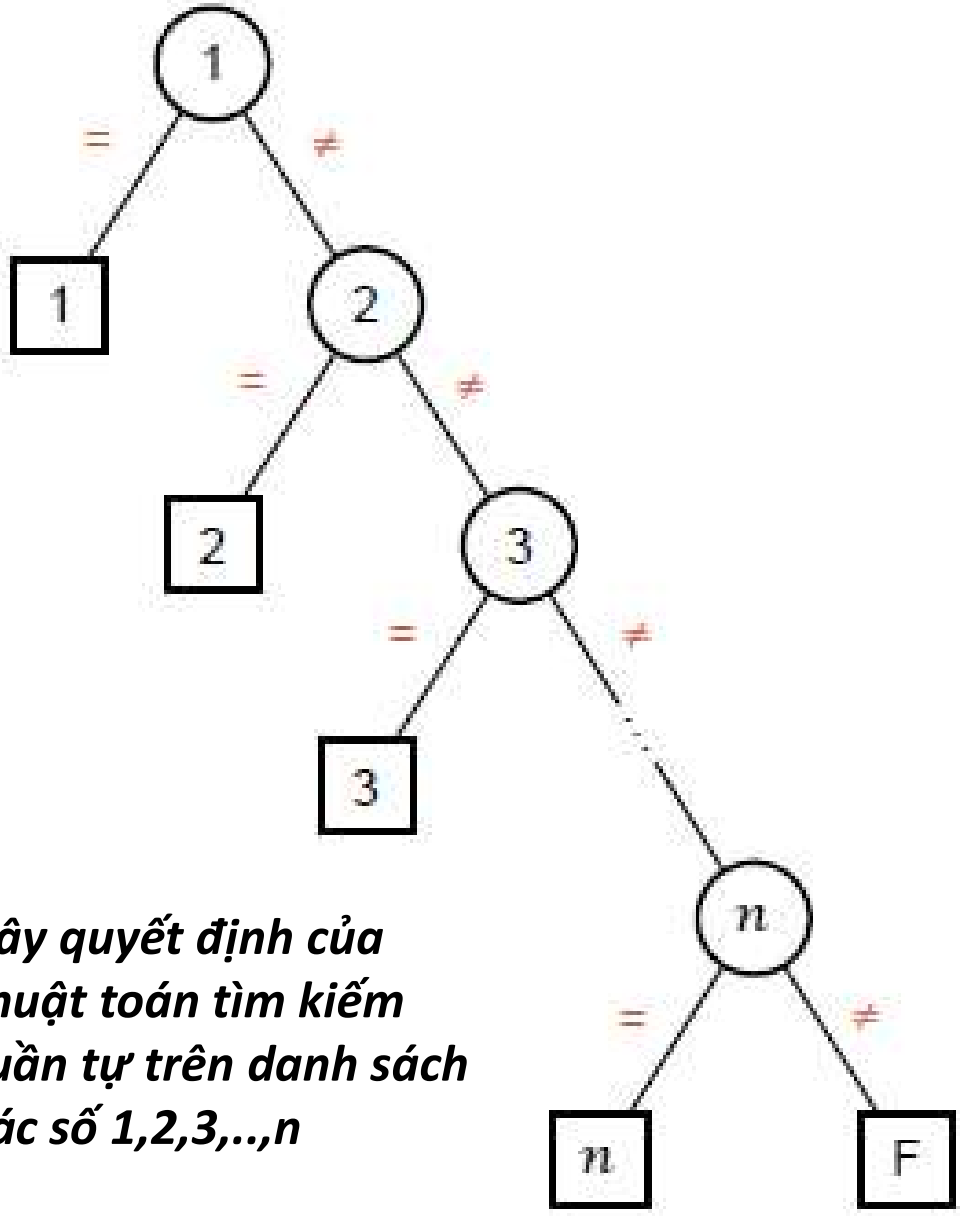
Cây quyết định

Cây quyết định

- Tên khác: cây so sánh, cây tìm kiếm

- Cây quyết định của một thuật toán thu được bằng cách theo vết các hành động của thuật toán
  - Biểu diễn mỗi phép so sánh khóa bằng 1 nút của cây (biểu diễn bằng hình tròn)
  - Cạnh vẽ từ đỉnh biểu diễn các khả năng có thể xảy ra của phép so sánh
  - Kết thúc thuật toán ta đặt F nếu không tìm thấy, hoặc là vị trí tìm thấy khóa (đây gọi là các lá của cây)

Cây quyết định



Cây quyết định của thuật toán tìm kiếm tuần tự trên danh sách các số 1,2,3,..,n

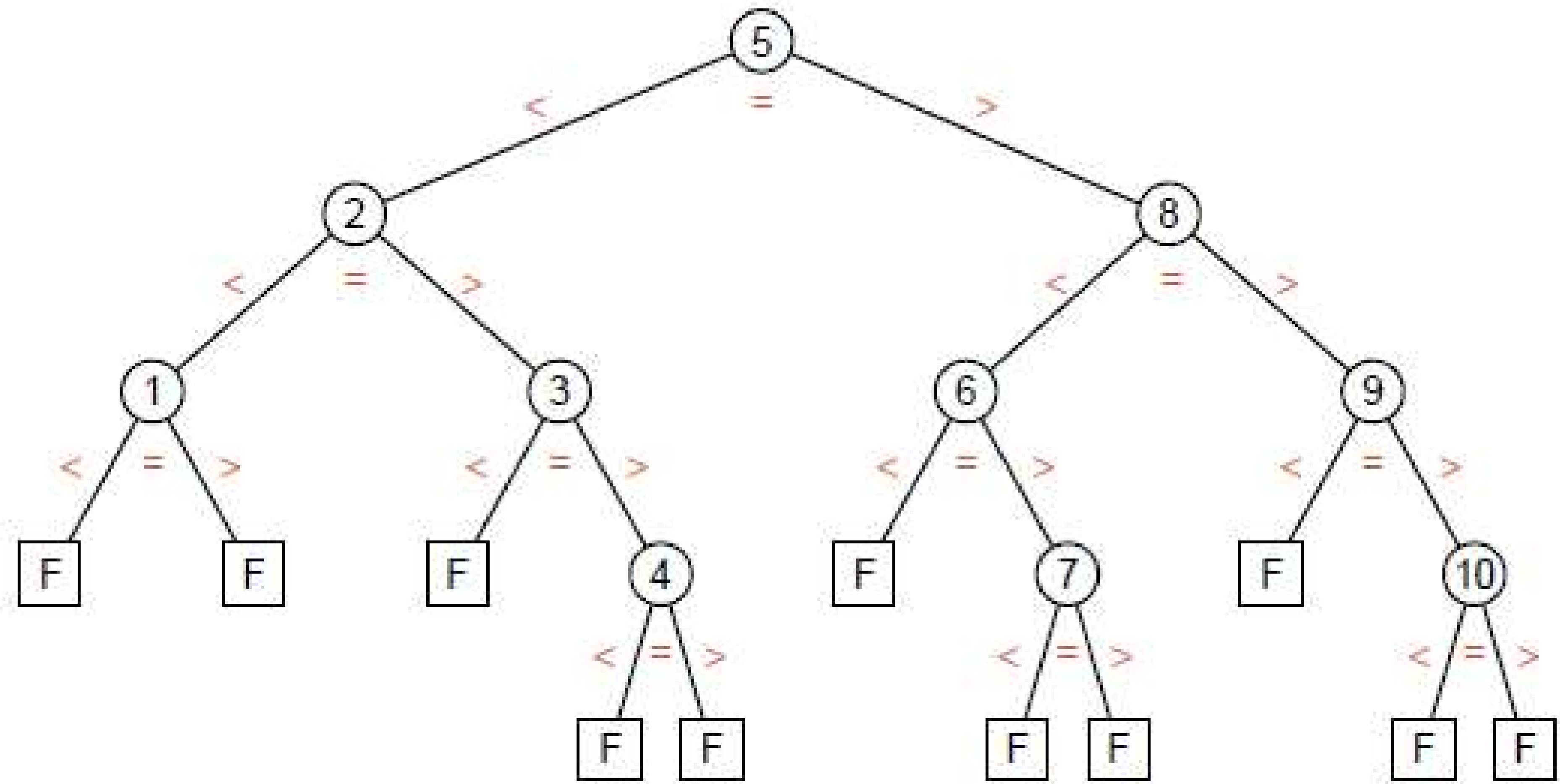
Gốc: đỉnh của cây

Chiều cao của cây: số lượng nút trên đường đi dài nhất từ gốc đến lá

Mức của đỉnh: số lượng cạnh trên đường từ gốc đến đỉnh đó

Số lượng phép so sánh trong một trường hợp cụ thể là số lượng nút trong

Cây quyết định



Cây quyết định cho thuật toán tìm kiếm nhị phân 1 trên dãy số 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

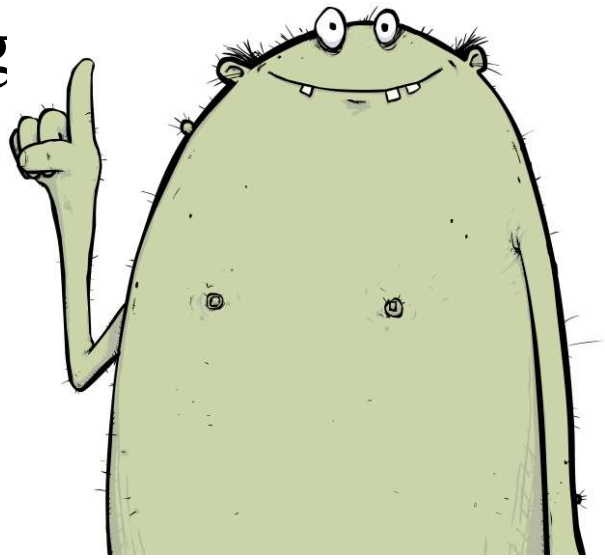
Cây quyết định

- Cây quyết định là 2-tree: các nút trong đều chỉ có 2 nút con.

- Số lượng phép so sánh trong thuật toán tìm kiếm nhị phân ở trên trong trường hợp tìm không thấy là $2\log(n+1)$

- Số phép so sánh trung bình trong trường

$$\frac{2(n+1)}{n}\log(n+1)-3$$



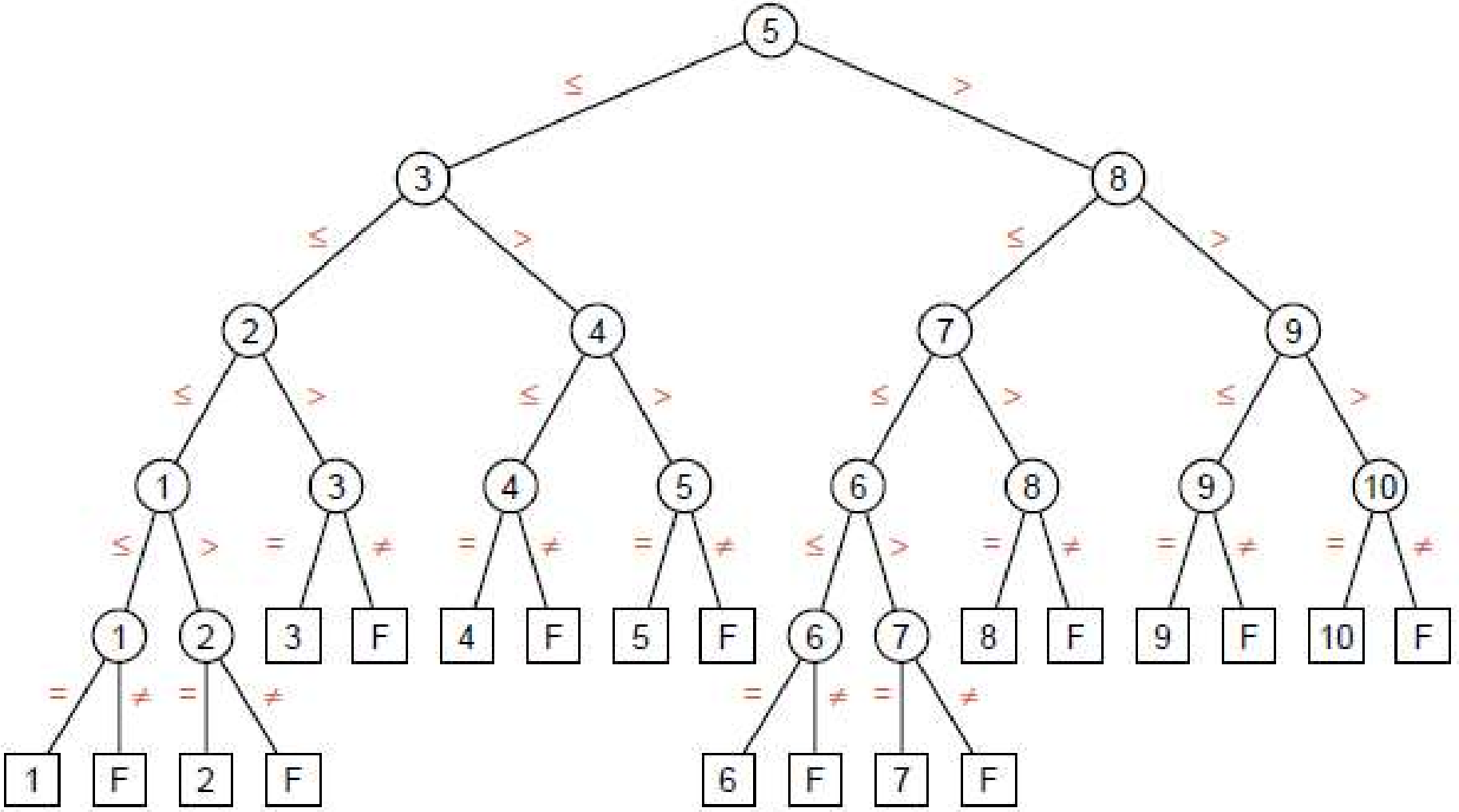
Extra section

Tìm kiếm nhị phân

Một cài đặt khác của tìm kiếm nhị phân

```
int binarySearch2(int A[], int key, int begin, int end, int &position)
{
    int mid;
    while(begin<end)
    {
        mid=(begin+end)/2;
        if(key<=A[mid]) end=mid-1;
        else begin=mid;
    }
    if(begin>end) return -1;
    else if(A[begin]=key) //begin=end
    {
        position=begin;
        return 0;
    }
}
```

Tìm kiếm nhị phân

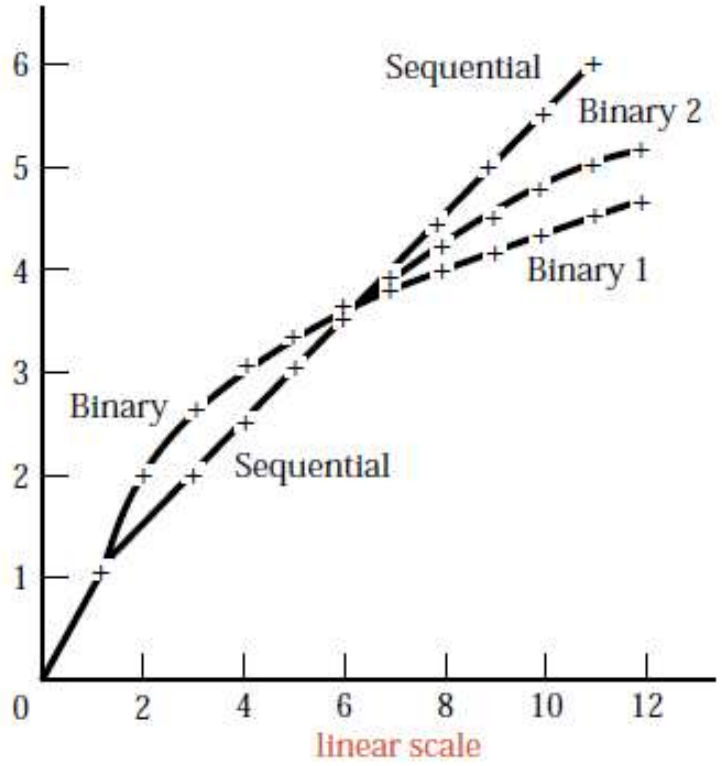


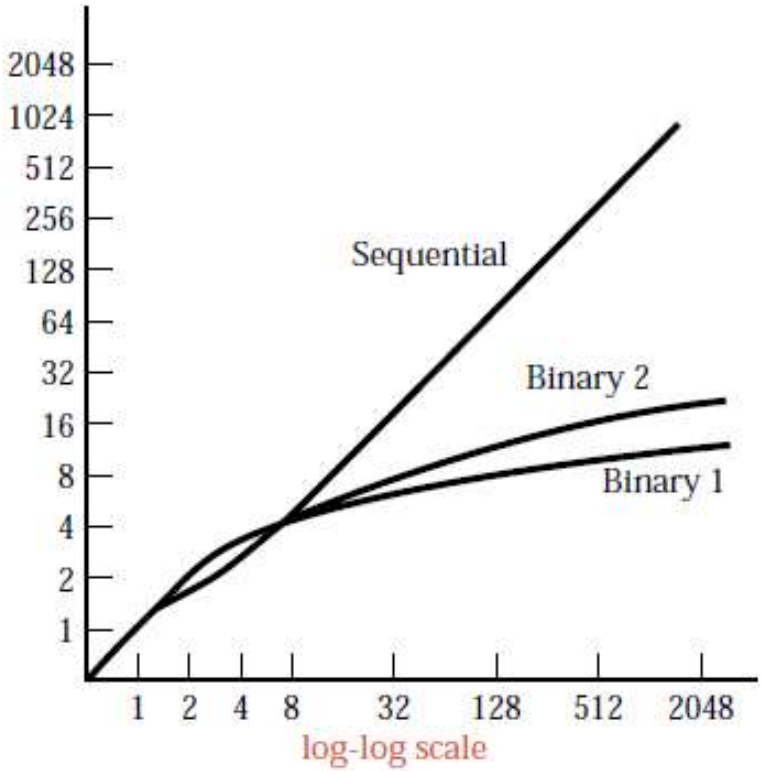
- Cây tìm kiếm tương ứng với thuật toán tìm kiếm nhị phân 2 trên dãy 1,2,3,4,5,6,7,8,9,10

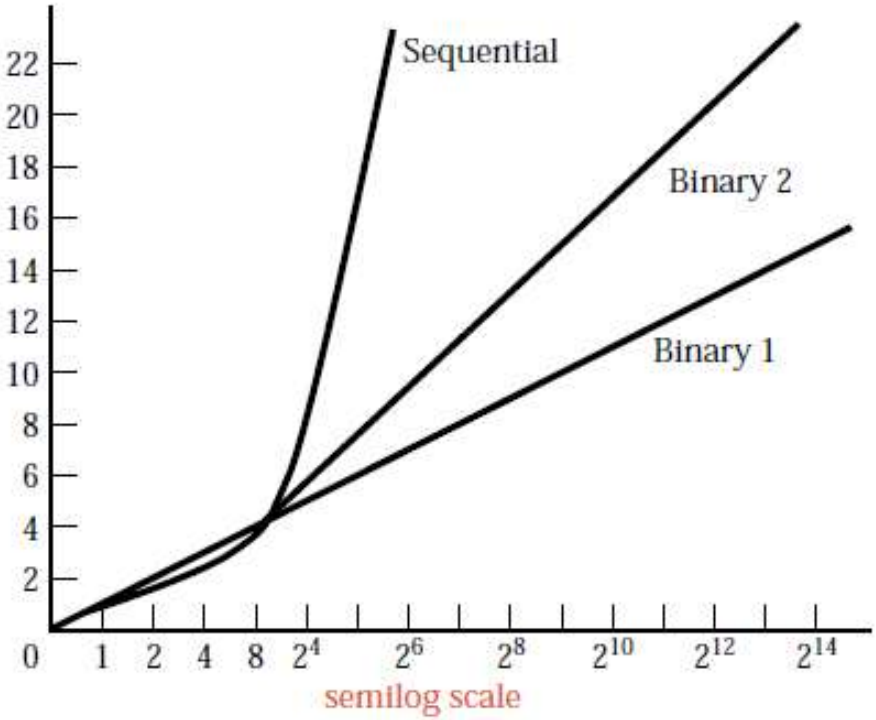
Tìm kiếm nhị phân

- So sánh hai cài đặt của thuật toán tìm kiếm nhị phân

	Tìm kiếm thành công	Tìm kiếm không thành công
binarySearch1	$\log n + 1$	$\log n + 1$
binarySearch2	$2\log n - 3$	$2 \log n$


Sequential
Binary 2
Binary 1
Binary
Sequential
0
1
2
3
4
5
6
2
4
6
8
10
12
linear scale


2048
1024
512
256
128
64
32
16
8
4
2
1
Sequential
Binary 2
Binary 1
1
2
4
8
32
128
512
2048
log-log scale


Sequential
Binary 2
Binary 1
22
20
18
16
14
12
10
8
6
4
2
0
1
2
4
8
2^4
2^6
2^8
2^{10}
2^{12}
2^{14}
semilog scale

Nhận xét

- Thuật toán tìm kiếm nhị phân là thuật toán tốt nhất trong các thuật toán tìm kiếm dựa trên việc so sánh giá trị các khóa trong dãy.
- Ý tưởng mở rộng thuật toán tìm kiếm: Nếu chúng ta biết khoảng tìm kiếm của mỗi khóa thì chúng ta có thể thu hẹp phạm vi của việc tìm kiếm

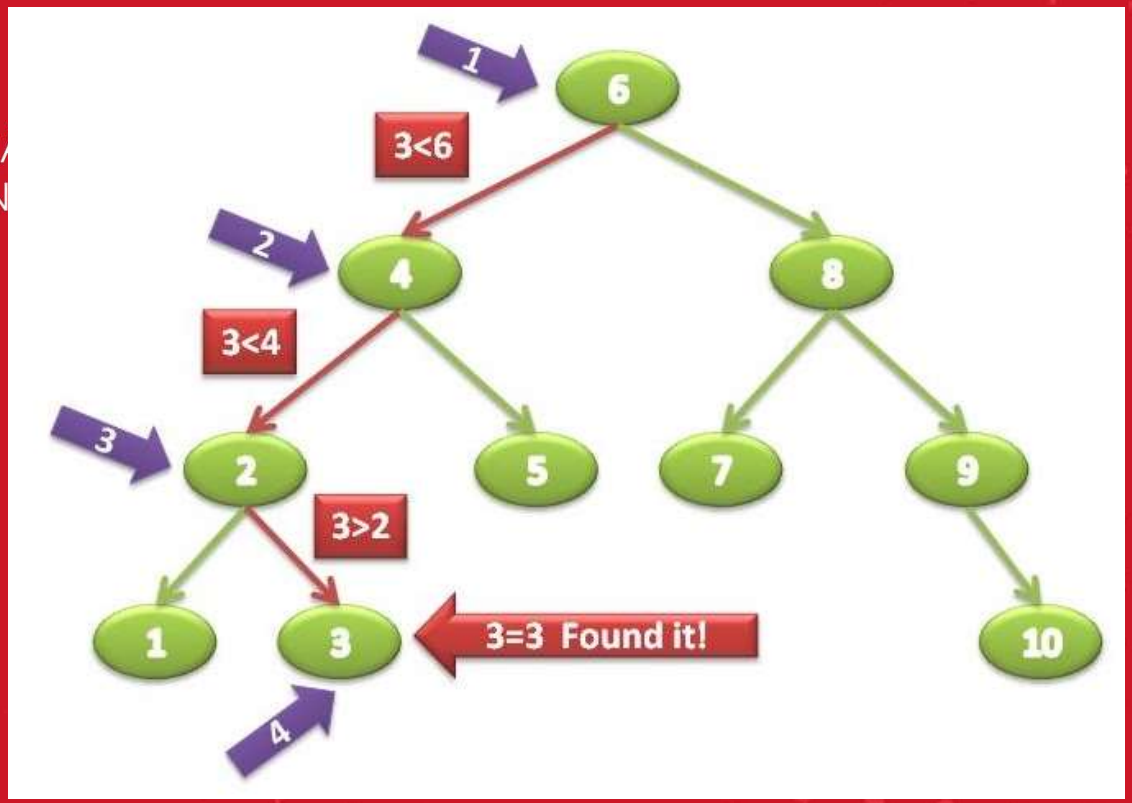
- Thuật toán ***interpolation search :***
  $O(\log\log n)$ trong trường hợp các khóa phân bố đều
  $O(n)$ trong trường hợp tồi nhất

  http://en.wikipedia.org/wiki/Interpolation_search


WWW.CARLONEWORLD.IT

Phần 2. Cây nhị phân tìm kiếm

- Cây nhị phân tìm kiếm tổng quát
- Thêm xóa nút trên cây tìm kiếm nhị phân
- Cây tìm kiếm nhị phân cân bằng AVL
- Cây tìm kiếm nhị phân cân bằng R-B tree



Cây tìm kiếm nhị phân
Binary search tree

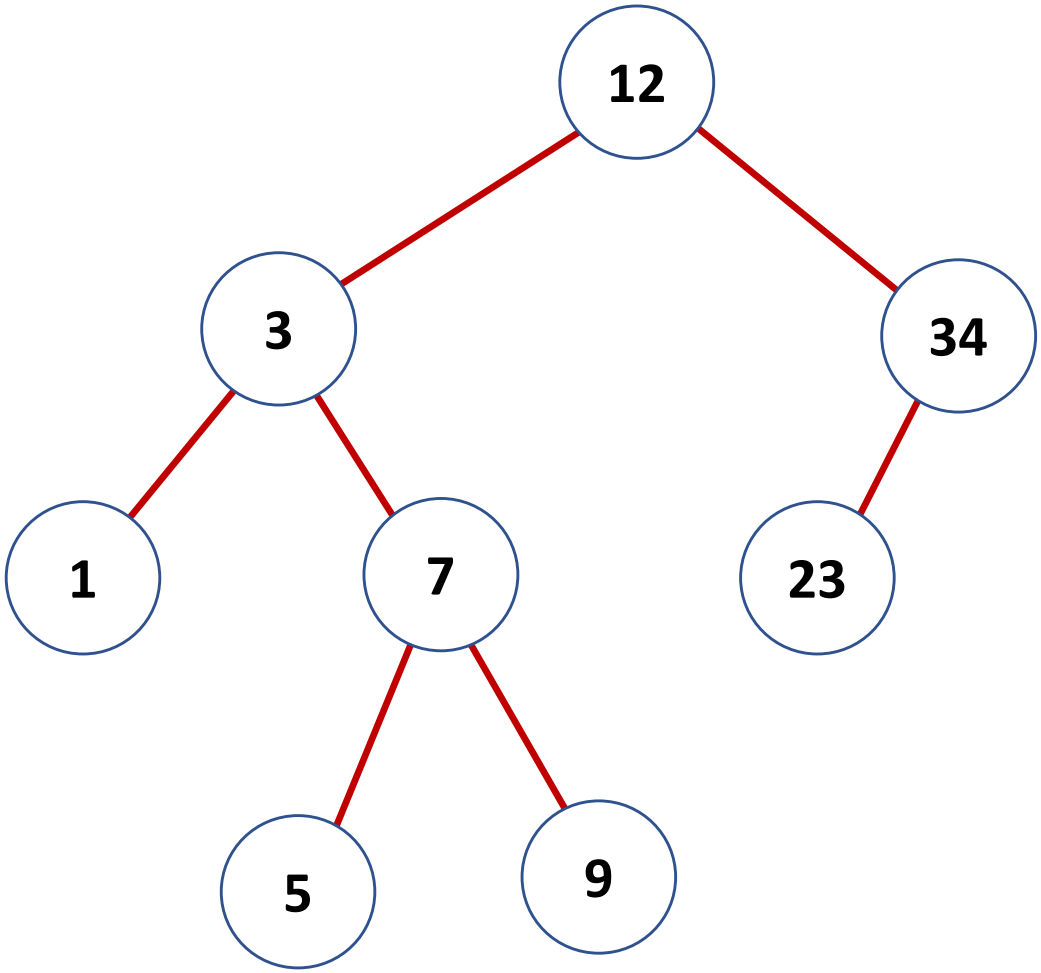
- Áp dụng tìm kiếm nhị phân trên cấu trúc liên kết (danh sách liên kết) ?
- Thêm, xóa phần tử trên cấu trúc liên tiếp (mảng) ?
  - Chi phí về thời gian?
  - Chi phí về bộ nhớ?

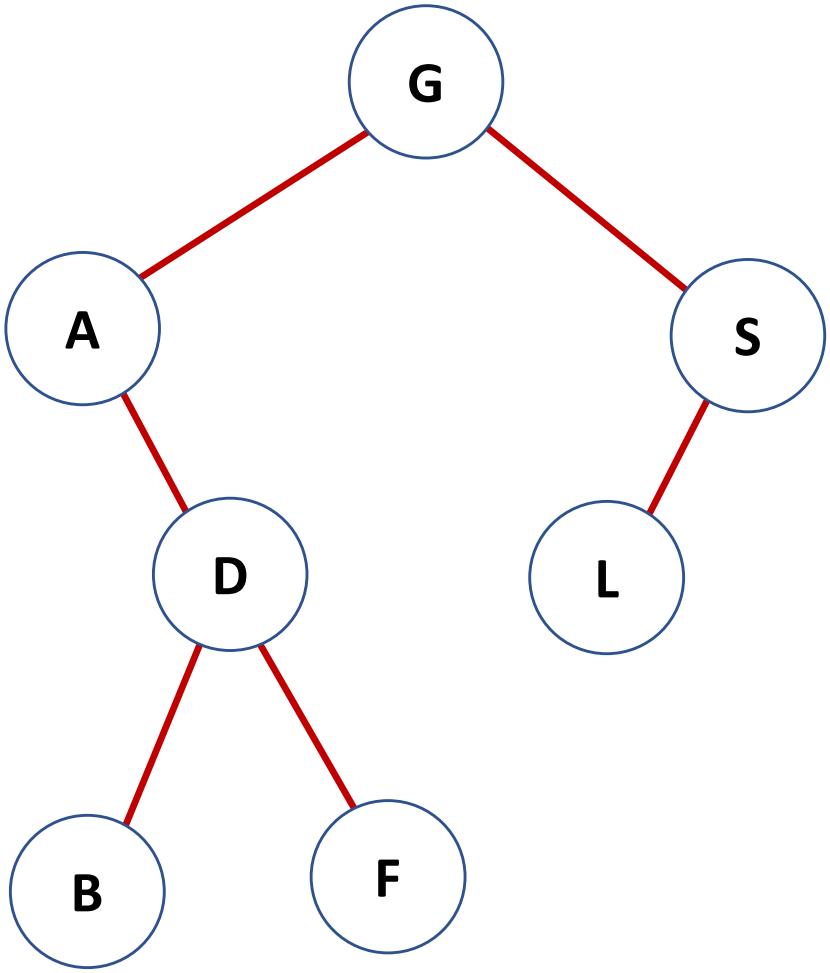
Cây tìm kiếm nhị phân

- Cây tìm kiếm nhị phân – binary search tree (BST):
  - Thời gian thực hiện tìm kiếm nhanh ($O(\log n)$)
  - Thêm, xóa các phần tử dễ dàng

- Cây tìm kiếm nhị phân là cây nhị phân rỗng hoặc mỗi nút có một giá trị khóa thỏa mãn các điều kiện sau:
  - Giá trị khóa của nút gốc (nếu tồn tại) lớn hơn giá trị khóa của bất kỳ nút nào thuộc cây con trái của gốc
  - Giá trị khóa của nút gốc (nếu tồn tại) nhỏ hơn giá trị khóa của bất kỳ nút nào thuộc cây con phải của gốc
  - Cây con trái và cây con phải của gốc cũng là các cây nhị phân tìm kiếm

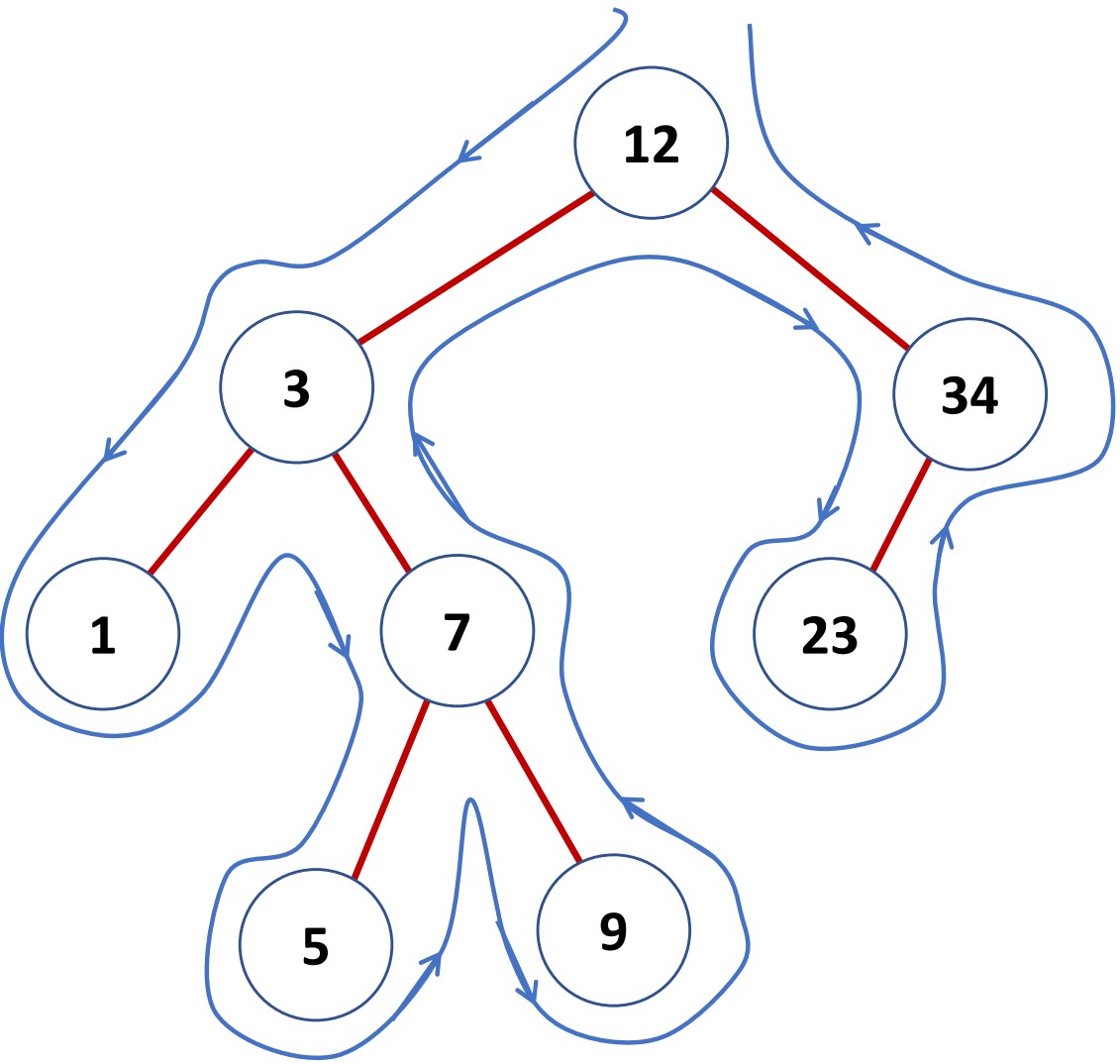
Cây tìm kiếm nhị phân





- Các nút trên cây nhị phân phải có khóa phân biệt !

Cây tìm kiếm nhị phân



- Duyệt cây tìm kiếm nhị phân
- Thứ tự giữa
  1, 3, 5, 7, 9, 12, 23, 34
- Thứ tự trước
  12, 3, 1, 7, 5, 9, 34, 23
- Thứ tự sau
  1, 5, 9, 7, 3, 23, 34, 12

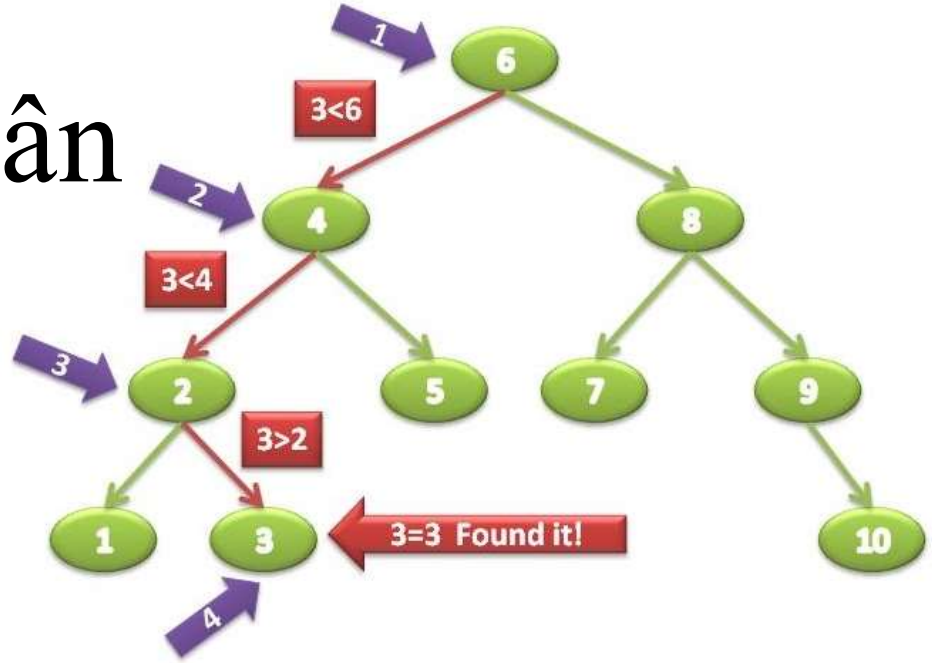
Cây tìm kiếm nhị phân

- Biểu diễn cây tìm kiếm nhị phân: giống biểu diễn cây nhị phân thông thường

- Biểu diễn bằng cấu trúc liên kết

```
struct TreeNode
{
   DATA_TYPE info;
   struct TreeNode *leftChild;
   struct TreeNode *rightChild;
}
```

Cây tìm kiếm nhị phân



Tìm kiếm:

- Nếu cây rỗng → không tìm thấy
- So sánh khóa cần tìm với khóa của nút gốc
  - Nếu bằng → Tìm thấy
  - Ngược lại lặp lại quá trình tìm kiếm ở cây con trái (hoặc cây con phải) nếu khóa cần tìm nhỏ hơn (lớn hơn) khóa của nút gốc

Cây tìm kiếm nhị phân

- Một số thao tác cơ bản của ADT cây tìm kiếm nhị phân
  - Search_for_node(): tìm kiếm xem một giá trị khóa có xuất hiện trên cây không
  - Insert(): Chèn một nút mới vào cây
  - Remove(): Gỡ bỏ một nút trên cây

Thao tác tìm kiếm

- Cài đặt đệ quy

```
typedef struct
TreeNode
{
    int info;
    struct TreeNode
    *leftChild;
    struct TreeNode
    *rightChild;
}NODE;
```

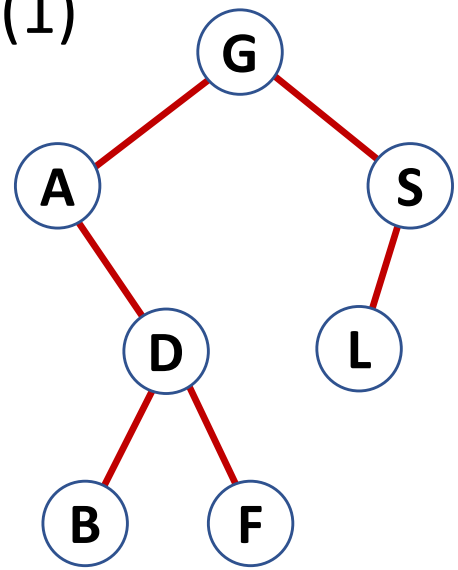
```
NODE* search_for_node(NODE *root, int key)
{
  if(root==NULL || root->info==key) return root;
  if(key<root->info)
       return search_for_node(root->leftChild, key);
  else return search_for_node(root->rightChild, key);
}
```

Thao tác tìm kiếm trên cây

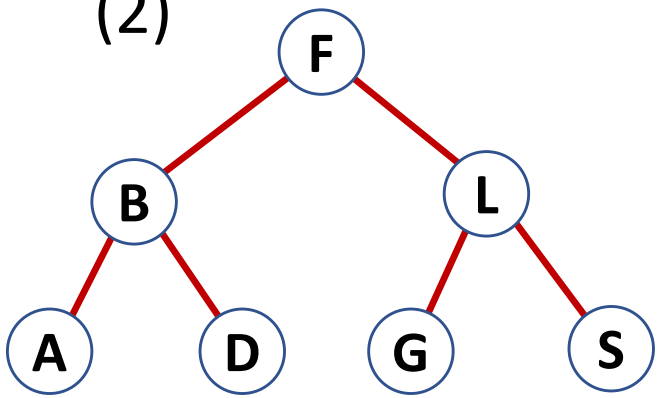
- Cài đặt không đệ quy

```
NODE* search_for_node(NODE *root, int key)
{
  while(root!=NULL && root->info!=key)
    if(key<root->info) root=root->leftChild;
    else root=root->rightChild;
  return root;
}
```

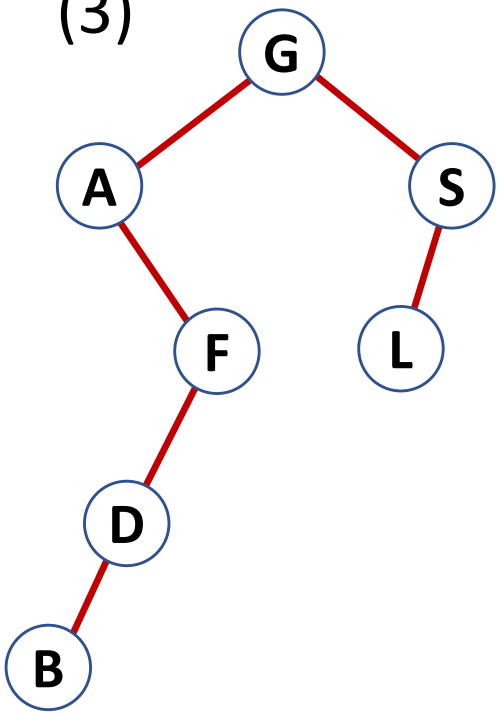
(1)



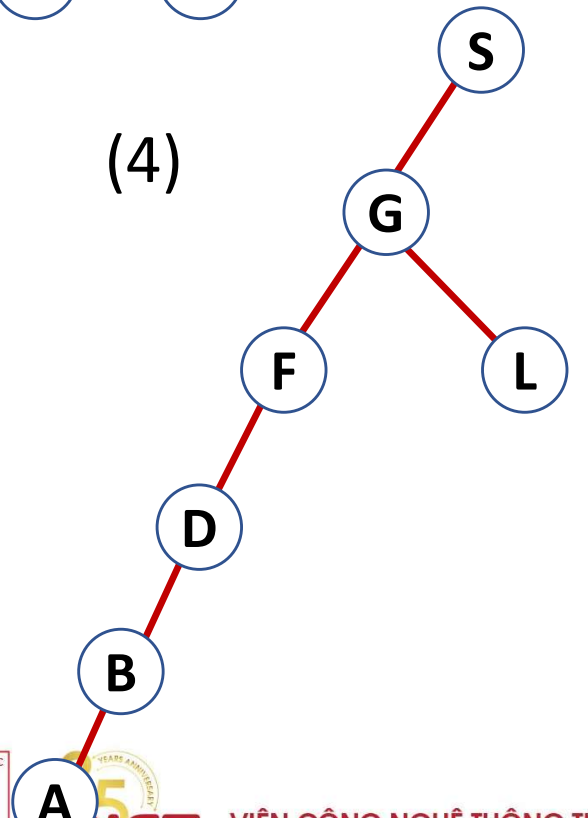
(2)



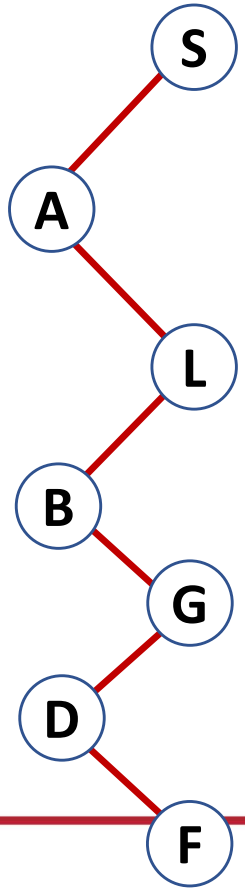
(3)



(4)



(5)

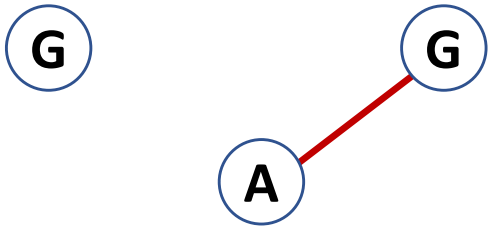


Cây nhị phân tìm kiếm cân bằng cho thời gian tìm kiếm là nhỏ nhất $O(\log n)$

Thêm nút mới vào cây

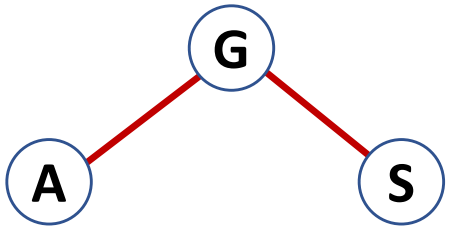
- Khi thêm nút mới phải đảm bảo những đặc điểm của cây nhị phân tìm kiếm không bị vi phạm

Thêm lần lượt các khóa : G, A, S, D, L, B, F vào cây nhị phân tìm kiếm ban đầu rỗng

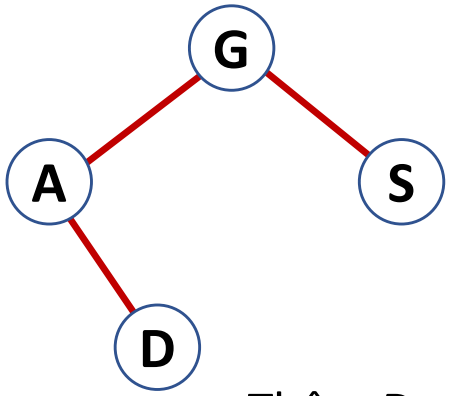


Thêm G

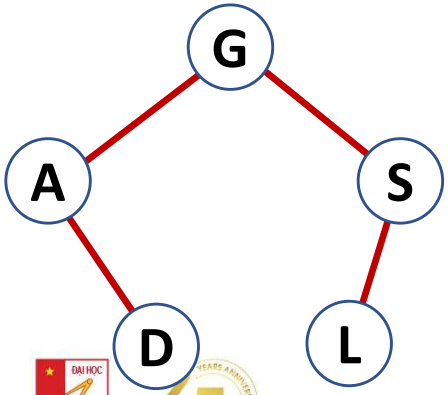
Thêm A



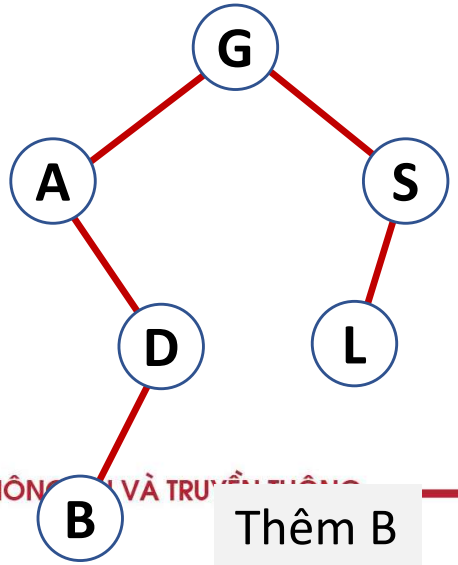
Thêm S



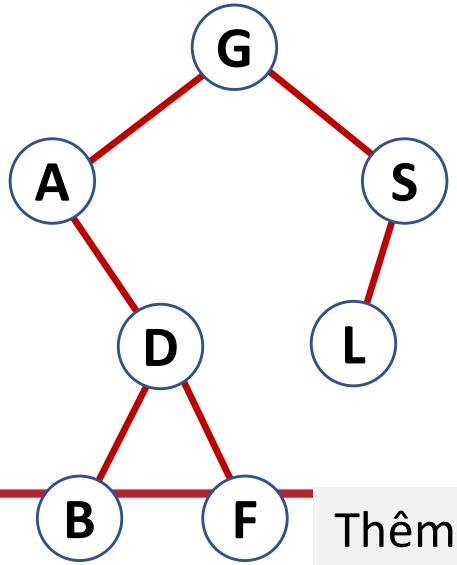
Thêm D



Thêm L



Thêm B



Thêm F

Thêm nút mới vào cây

- Nhận xét:
  - Thứ tự thêm các nút khác nhau ***có thể*** tạo ra các cây nhị phân tìm kiếm khác nhau
  - Khóa đầu tiên thêm vào cây rỗng sẽ trở thành nút gốc của cây
  - Để thêm các khóa tiếp theo ta phải thực hiện tìm kiếm để tìm ra vị trí chèn thích hợp
  - Các khóa được thêm tại nút lá của cây
  - Cây sẽ bị suy biến thành danh sách liên kết đơn nếu các nút chèn vào đã có thứ tự (tạo ra các cây lệch trái hoặc lệch phải)

Thêm nút mới vào cây

- Cài đặt đệ quy

```
int insert(NODE *&root, int value)
{
  if(root==NULL)
  {
    root = (NODE*)malloc(sizeof(NODE));
    root->info = value;
    root->leftChild = NULL;
    root->rightChild = NULL;
    return 0;// success
  }
  else if(value < root->info) insert(root->leftChild,value);
  else if(value > root->info) insert(root->rightChild,value);
  else return -1;//duplicate
}
```

Thêm nút mới vào cây

- Cài đặt không đệ quy

```
int insert(NODE *&root, int value)
{
   NODE *pRoot = root;
   while(pRoot!=NULL && pRoot->info!=key)
      if(key<pRoot->info) pRoot=pRoot->leftChild;
      else pRoot=pRoot->rightChild;

   if(pRoot!=NULL) return -1;//duplicate
   else
   {
      pRoot = (NODE*)malloc(sizeof(NODE));
      pRoot->info = value;
      pRoot->leftChild = NULL;
      pRoot->rightChild = NULL;
      return 0;// success
   }
}
```

Thêm nút mới vào cây

- **Nhận xét**: Thời gian thực hiện của thuật toán phụ thuộc vào dạng của cây
  - Cây cân bằng : thời gian cỡ $O(\log n)$
  - Cây bị suy biến (lệch trái, phải hoặc zig-zag): thời gian cỡ $O(n)$

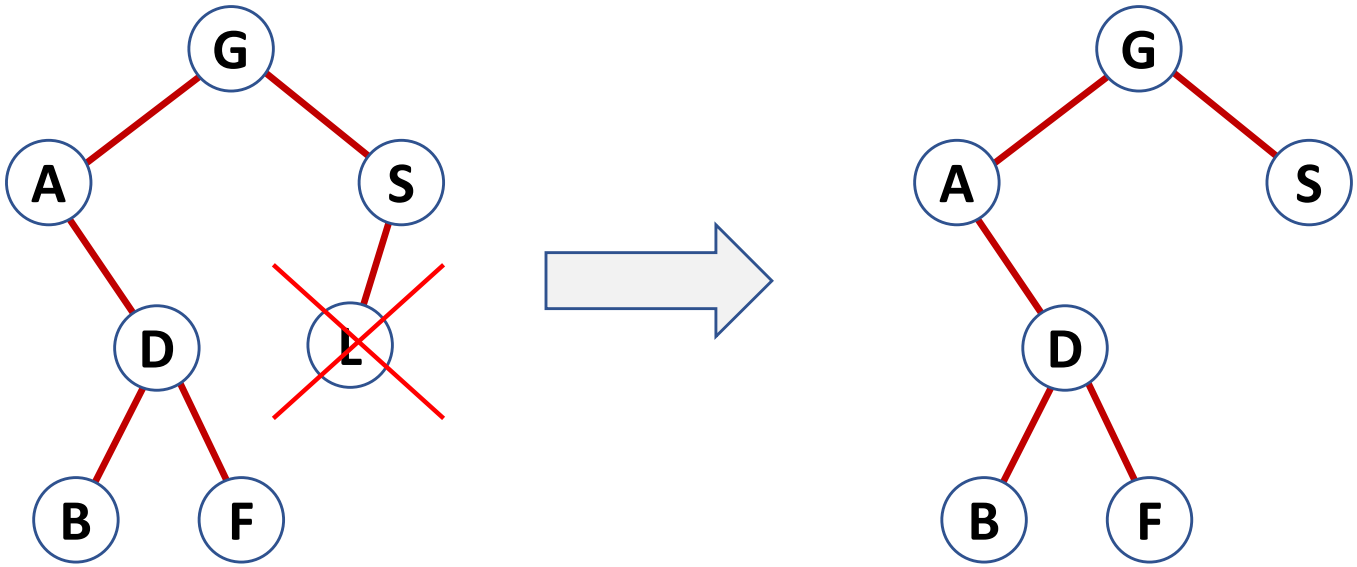
$$O(\log n) \leq T(n) \leq O(n)$$

  - **Thuật toán sắp xếp (treeSort)**: thêm lần lượt các phần tử vào cây nhị phân tìm kiếm, sau đó duyệt theo thứ tự giữa. Số phép so sánh: $1.39\, n \log n + O(n)$

Loại bỏ nút khỏi cây

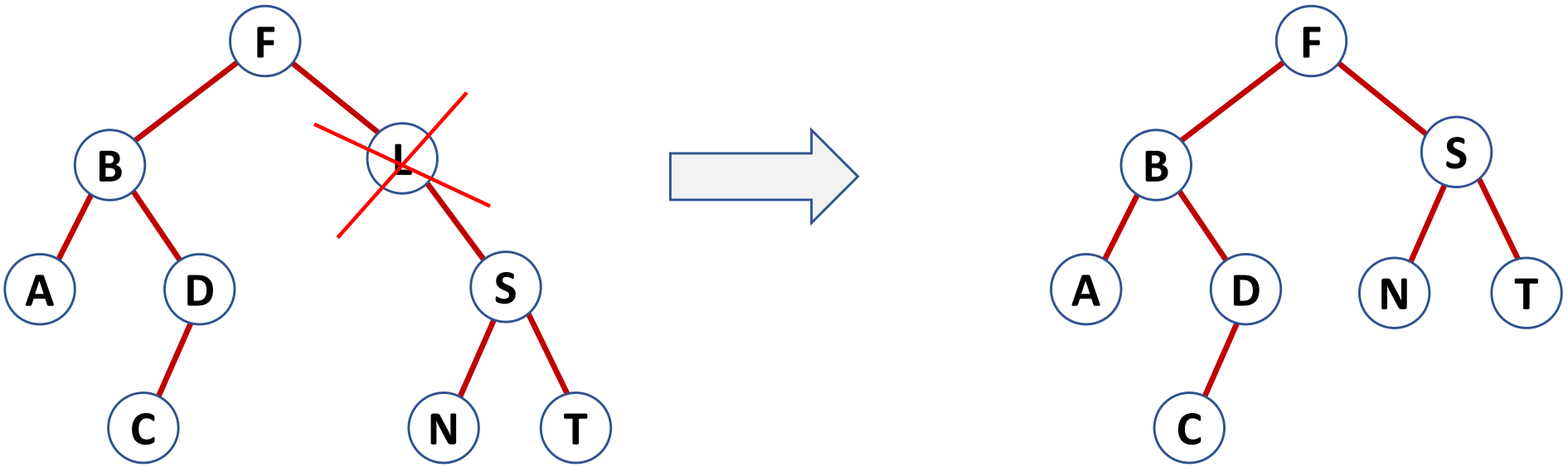
- **Loại bỏ nút trên cây**: cần đảm bảo những đặc điểm của cây không bị vi phạm

- Trường hợp loại bỏ nút lá:



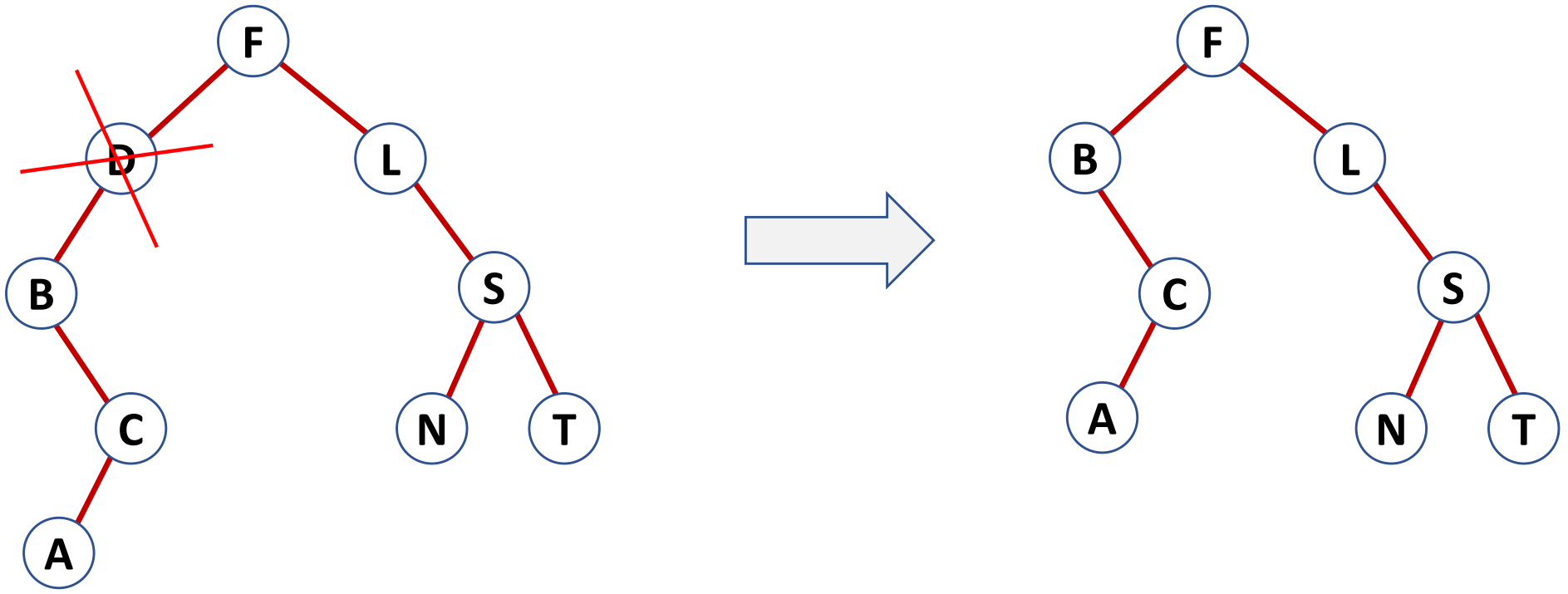
Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong: nút trong khuyết con trái hoặc con phải



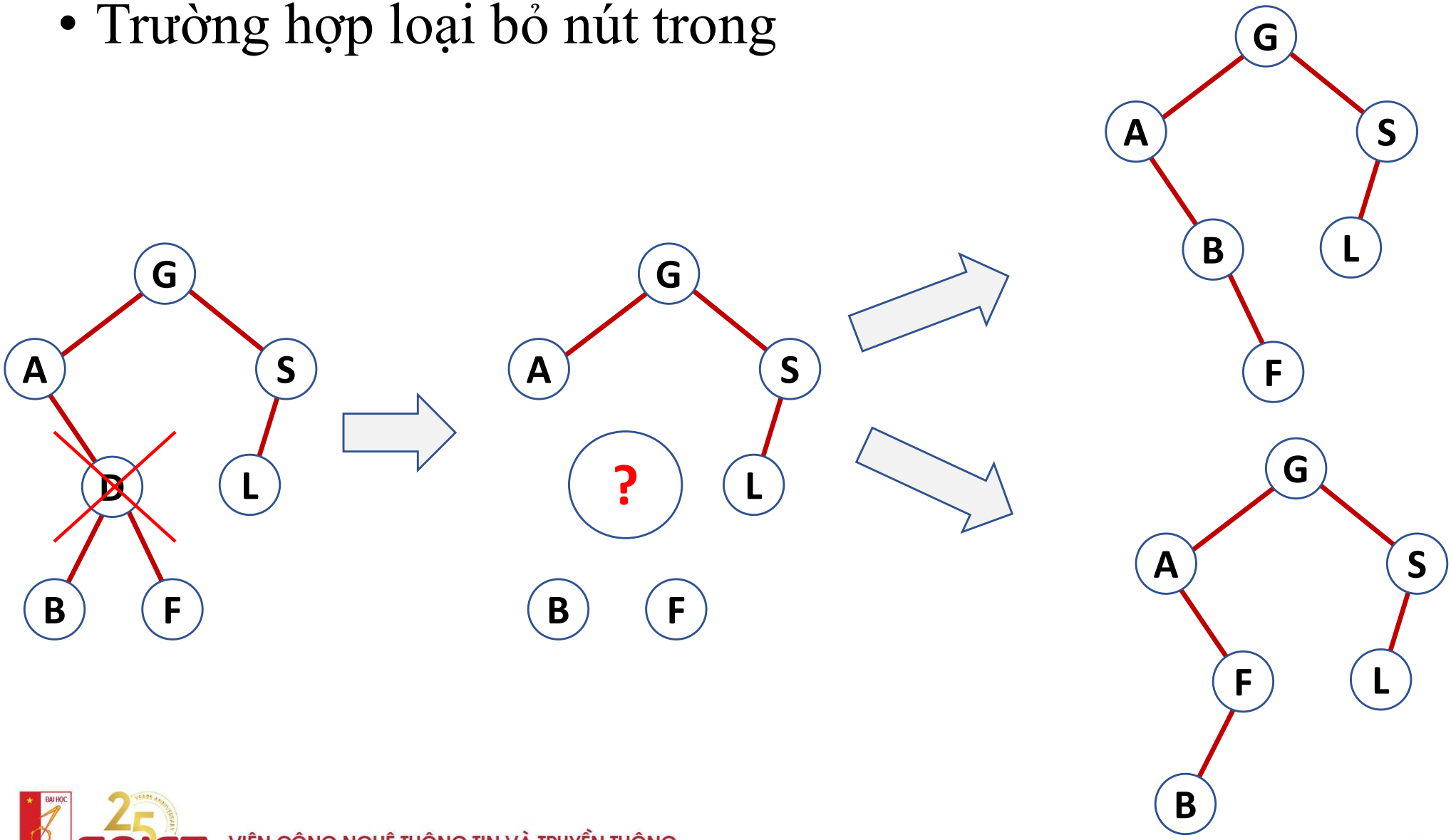
Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong: nút trong khuyết con trái hoặc con phải

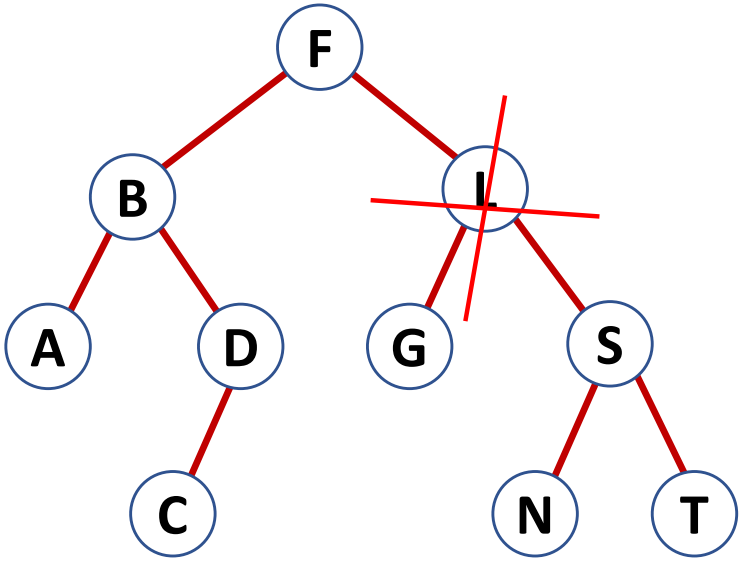


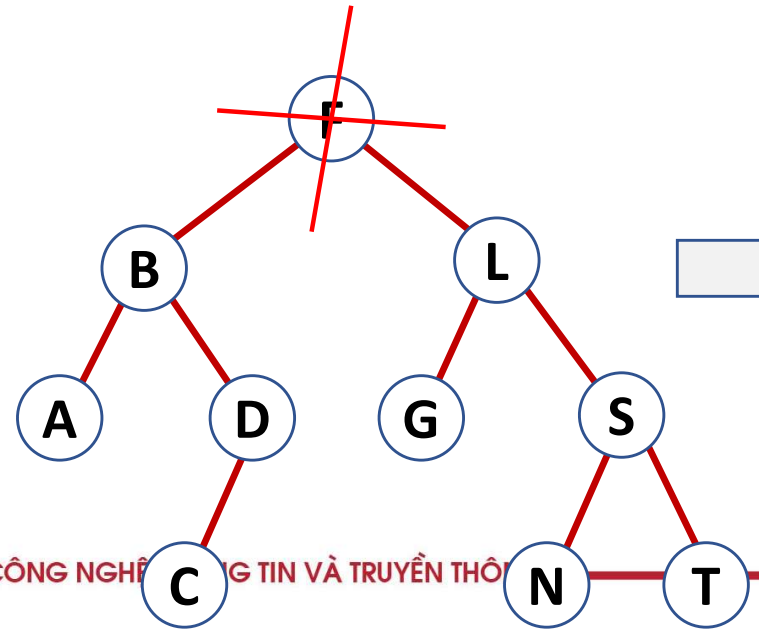
Loại bỏ nút khỏi cây

- Trường hợp loại bỏ nút trong



Loại bỏ nút khỏi cây





Loại bỏ nút khỏi cây

- Nhận xét:
  - Thực hiện tìm kiếm để xem khóa cần xóa có trên cây
  - Nếu nút có khóa cần xóa là nút lá: ngắt bỏ kết nối với nút cha của nó, giải phóng bộ nhớ cấp phát cho nút đó
  - Nếu nút cần xóa là nút trong không đầy đủ (khuyết con trái hoặc phải): Thay thế bằng cây con không khuyết
  - Nếu nút cần xóa là nút trong đầy đủ (có đủ các con): cần tìm một nút thuộc để thay thế cho nó, sau đó xóa nút được thay thế. Nút thay thế là nút ở liền trước (hoặc liền sau trong duyệt theo thứ tự giữa)
    - Tìm nút phải nhất của cây con trái (*)
    - Hoặc, nút trái nhất thuộc cây con phải

```
int remove_node(NODE *&root)
{
  if(root == NULL) return -1; //remove null
  NODE *ptr = root; //remember this node for delete later
  if(root->leftChild == NULL) root=root->rightChild;
  else if(root->rightChild == NULL) root=root->leftChild;
  else //find the rightmost node on the left sub tree
  {
    NODE *preP = root;
    ptr = root->leftChild;
    while(ptr->rightChild != NULL)
    {
      preP = ptr;
      ptr = ptr->rightChild;
    }
    root->info = ptr->info;
    if(preP == root) root->leftChild = ptr->leftChild;
    else preP->rightChild = ptr->leftChild;
  }
  free(ptr);
  return 0;// remove success
}
```

Loại bỏ nút khỏi cây

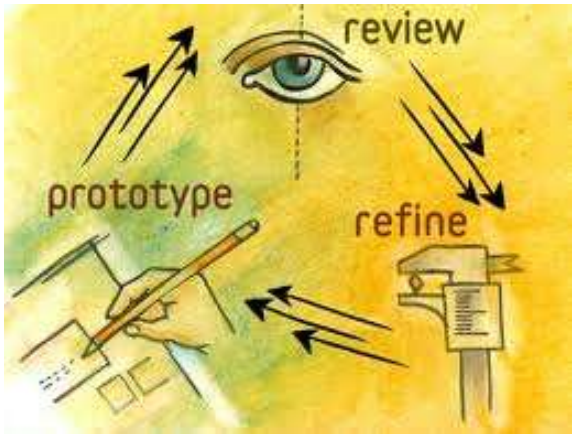
```
int remove(NODE *&root, int key)
{
  if(root==NULL || key==root->info)
       return remove_node(root);
  else if(key < root->info)
       return remove_node(root->leftChild,key);
  else return remove_node(root->rightChild,key);
}
```

Phần 3. Cây nhị phân tìm kiếm nâng cao

- Cây AVL
- Cây SPLAY

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

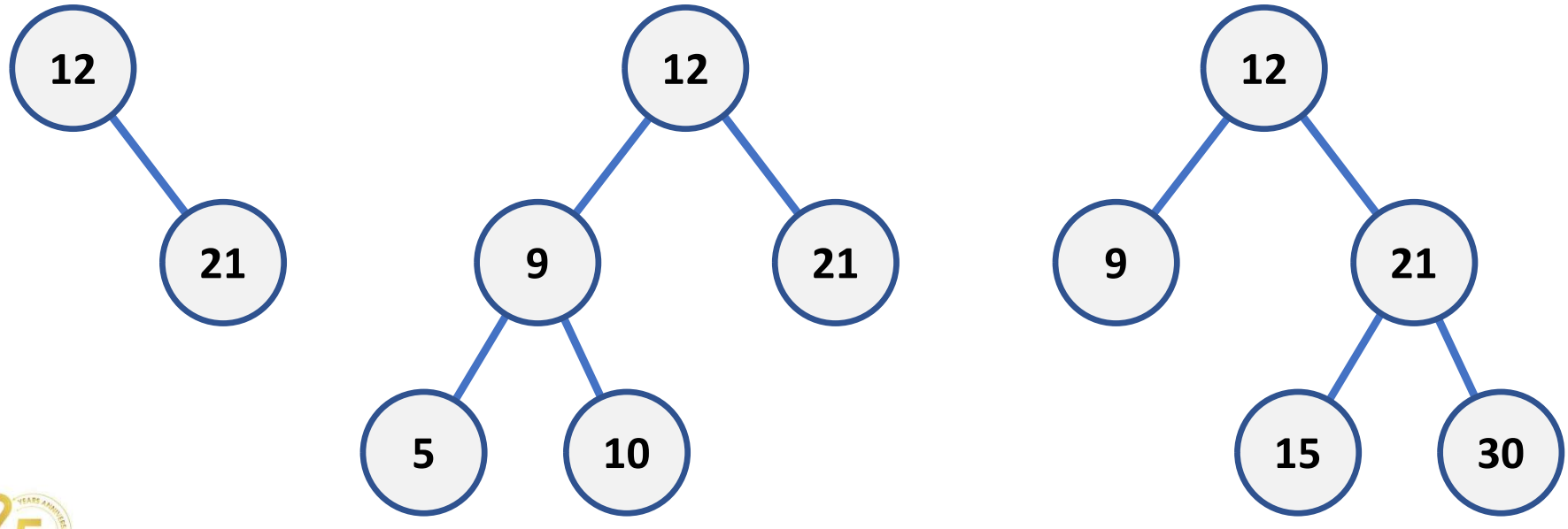


Cây tìm kiếm nhị phân cân bằng AVL Tree

G. M. ADELSON-VELSKII và E. M. LANDIS

AVL tree

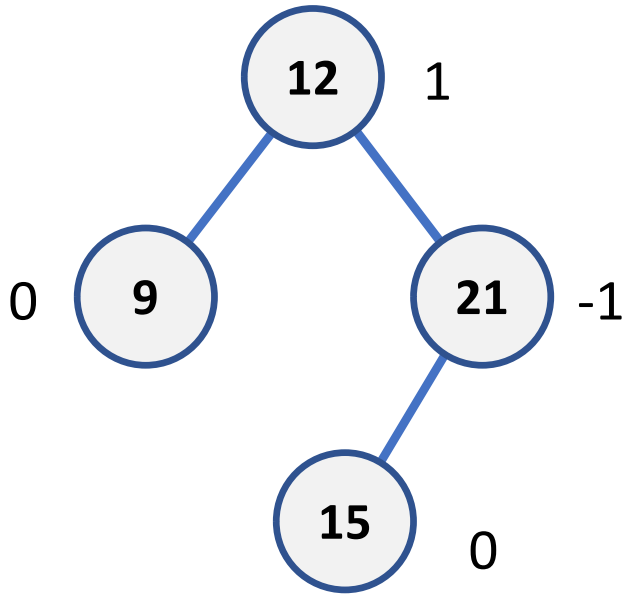
- Cây tìm kiếm nhị phân cân bằng – AVL tree:
  - Là cây tìm kiếm nhị phân
  - Chiều cao của cây con trái và cây con phải của gốc chênh nhau không quá 1
  - Cây con trái và cây con phải cũng là các cây AVL



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



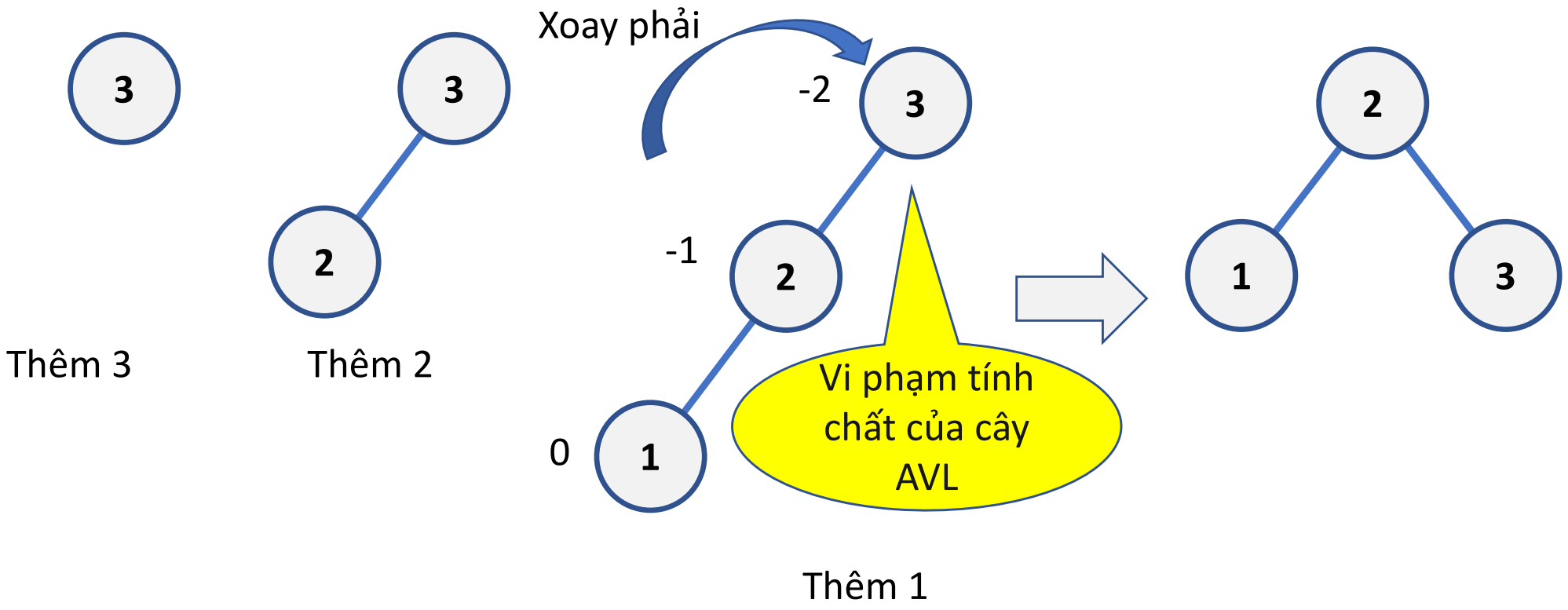
AVL tree

- 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;
```

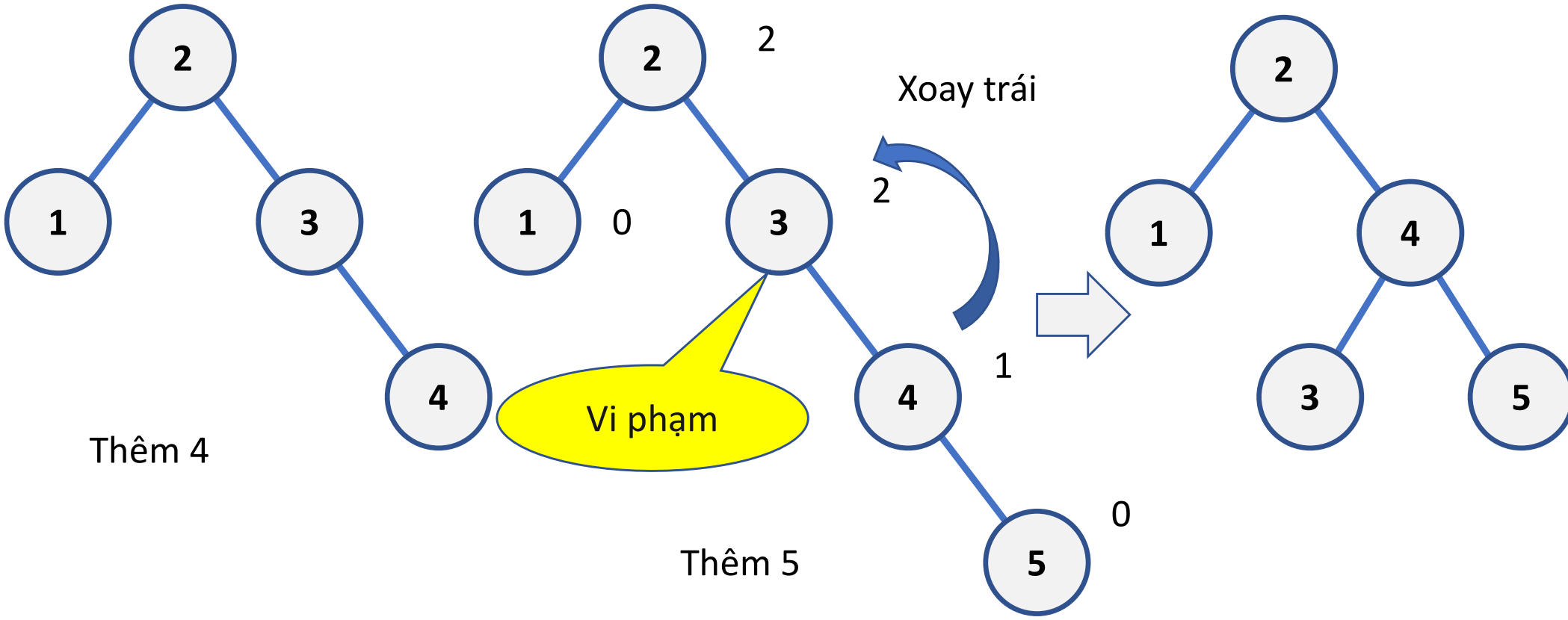
AVL tree

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



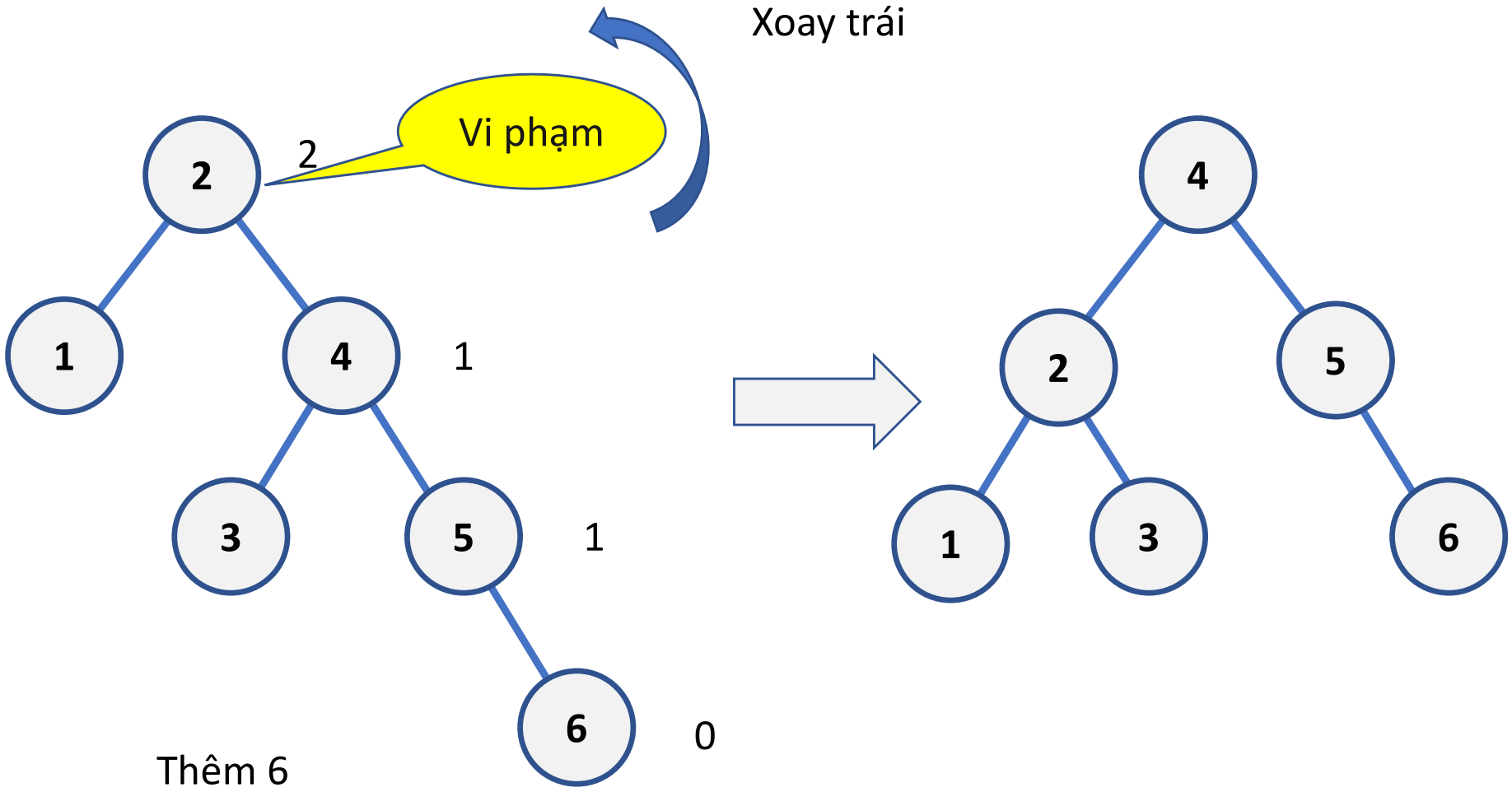
Xử lý bằng phép xoay nút

AVL tree

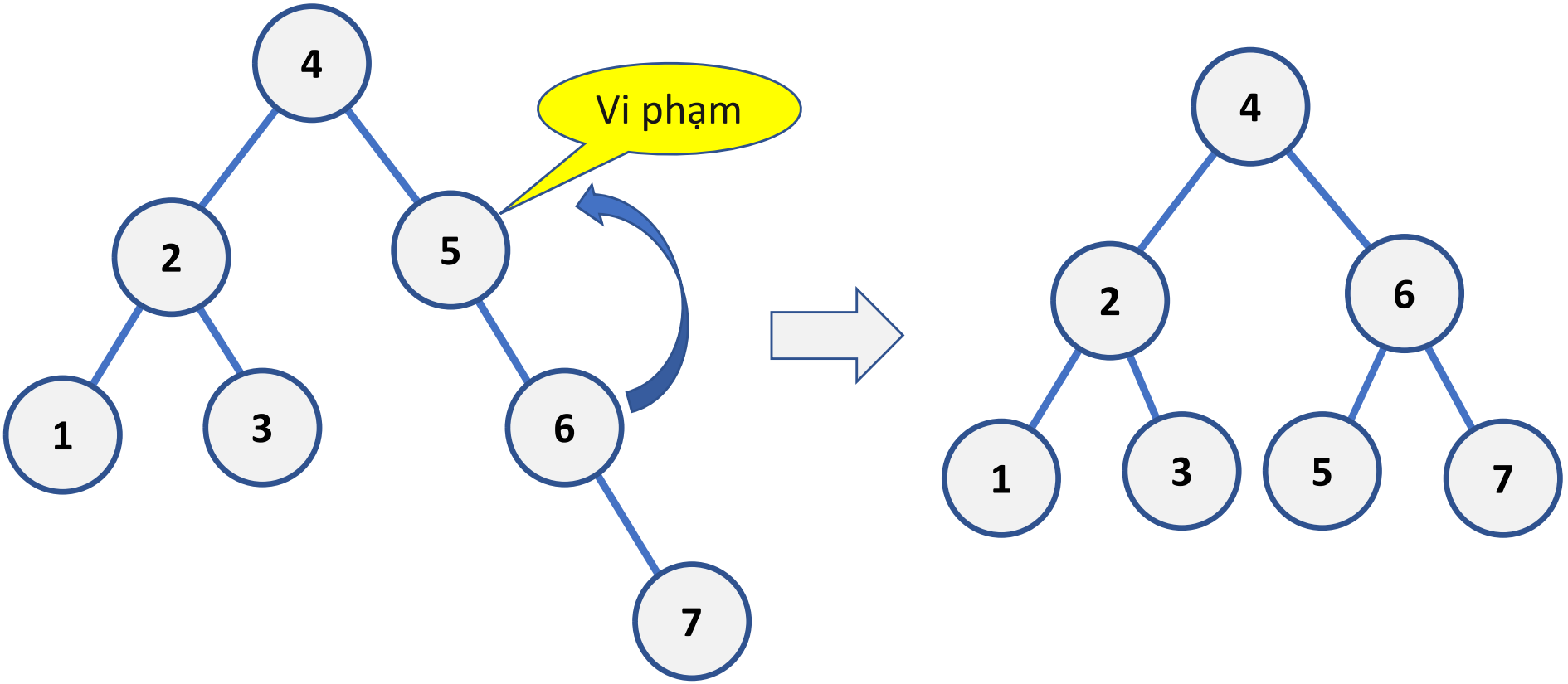


Xoay giữa nút vi phạm và nút con của nó

AVL tree

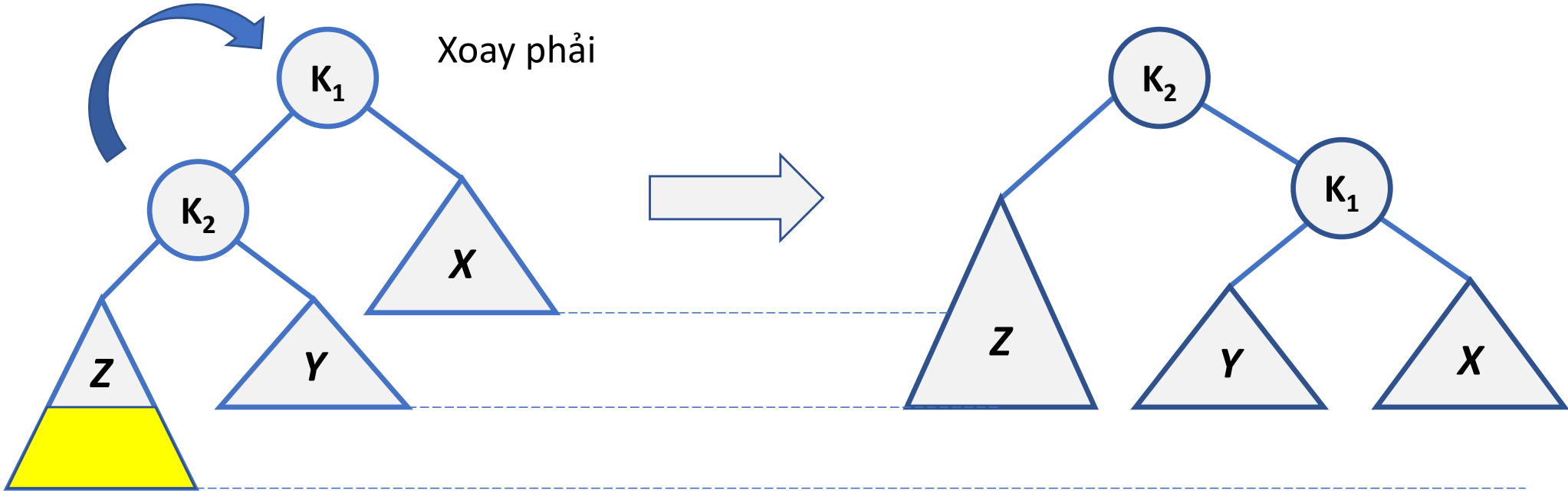


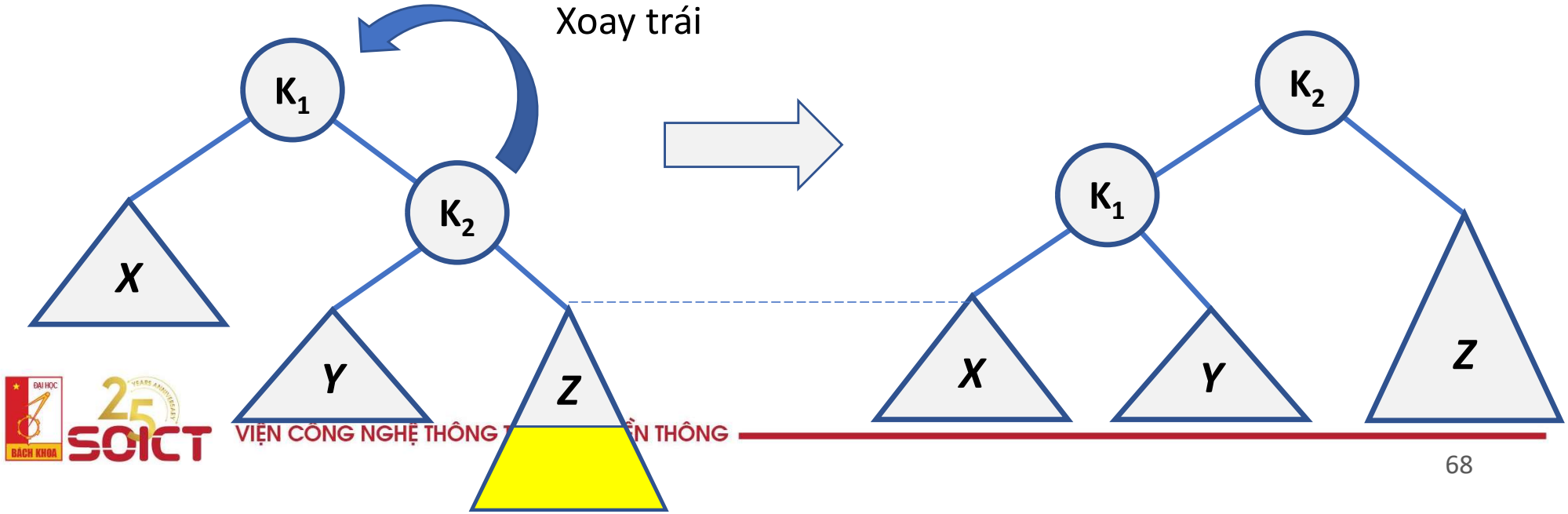
AVL tree



Thêm 7

AVL tree





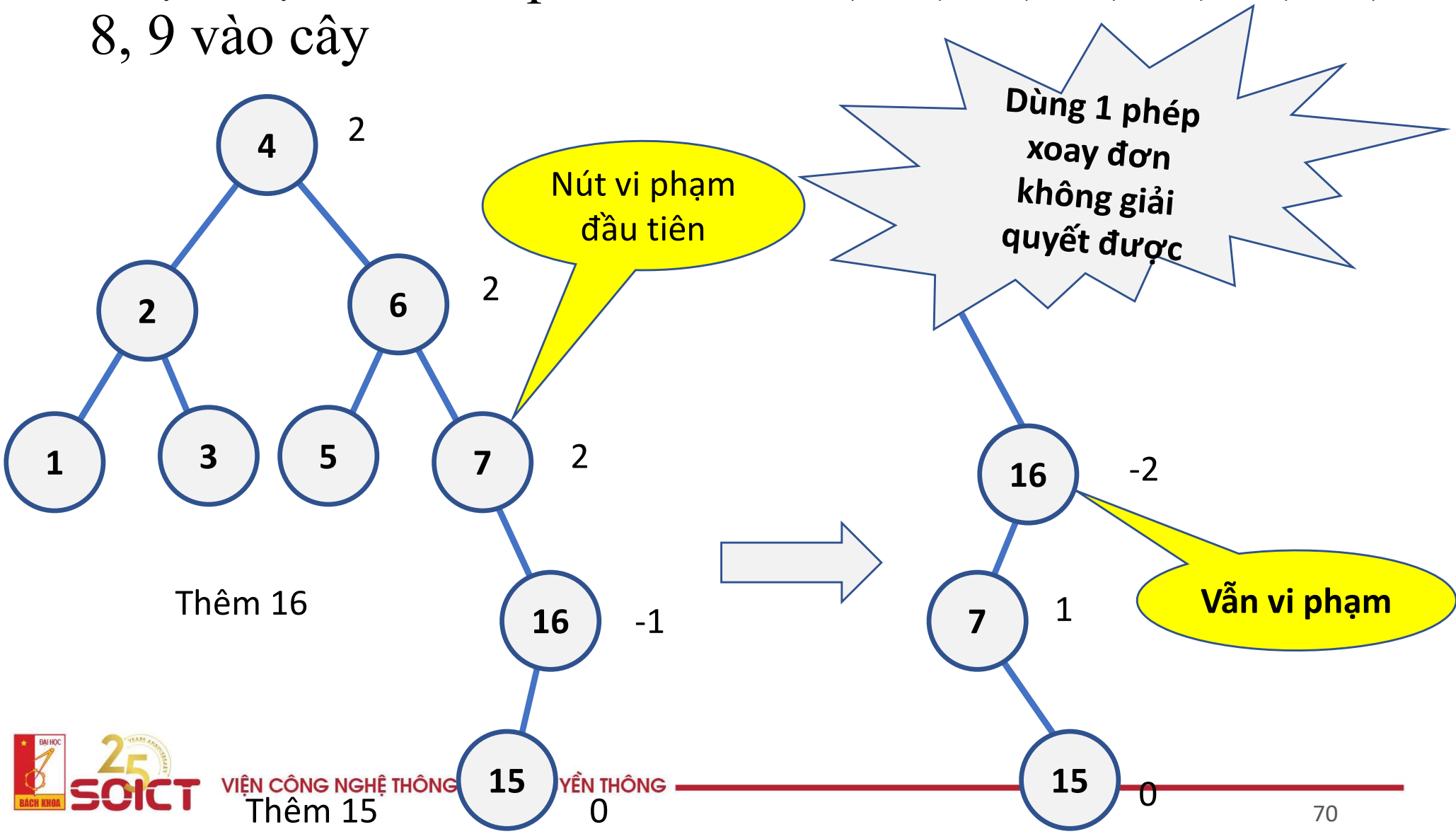
AVL tree

Phép xoay đơn – single rotation:

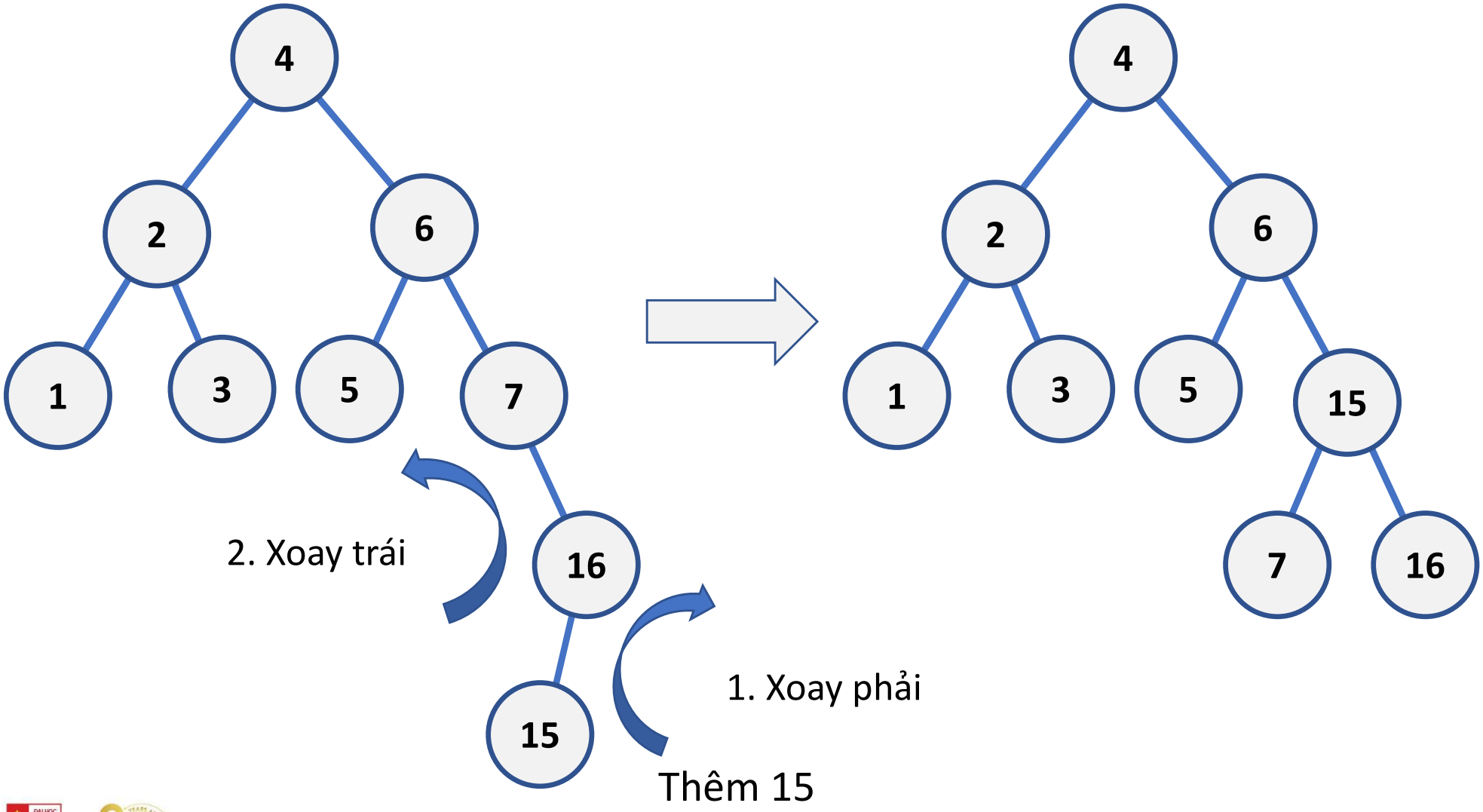
- Dùng để điều chỉnh khi mà nút mới thêm vào trong trường hợp:
  - (i) Cây con trái của nút con trái, hoặc
  - (ii)Cây con phải của nút con phải của nút
- Thực hiện tại nút vi phạm đầu tiên trên đường từ vị trí mới thêm trở về gốc
- Xoay giữa nút vi phạm và nút con trái (xoay phải) – TH i) (hoặc con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

AVL tree

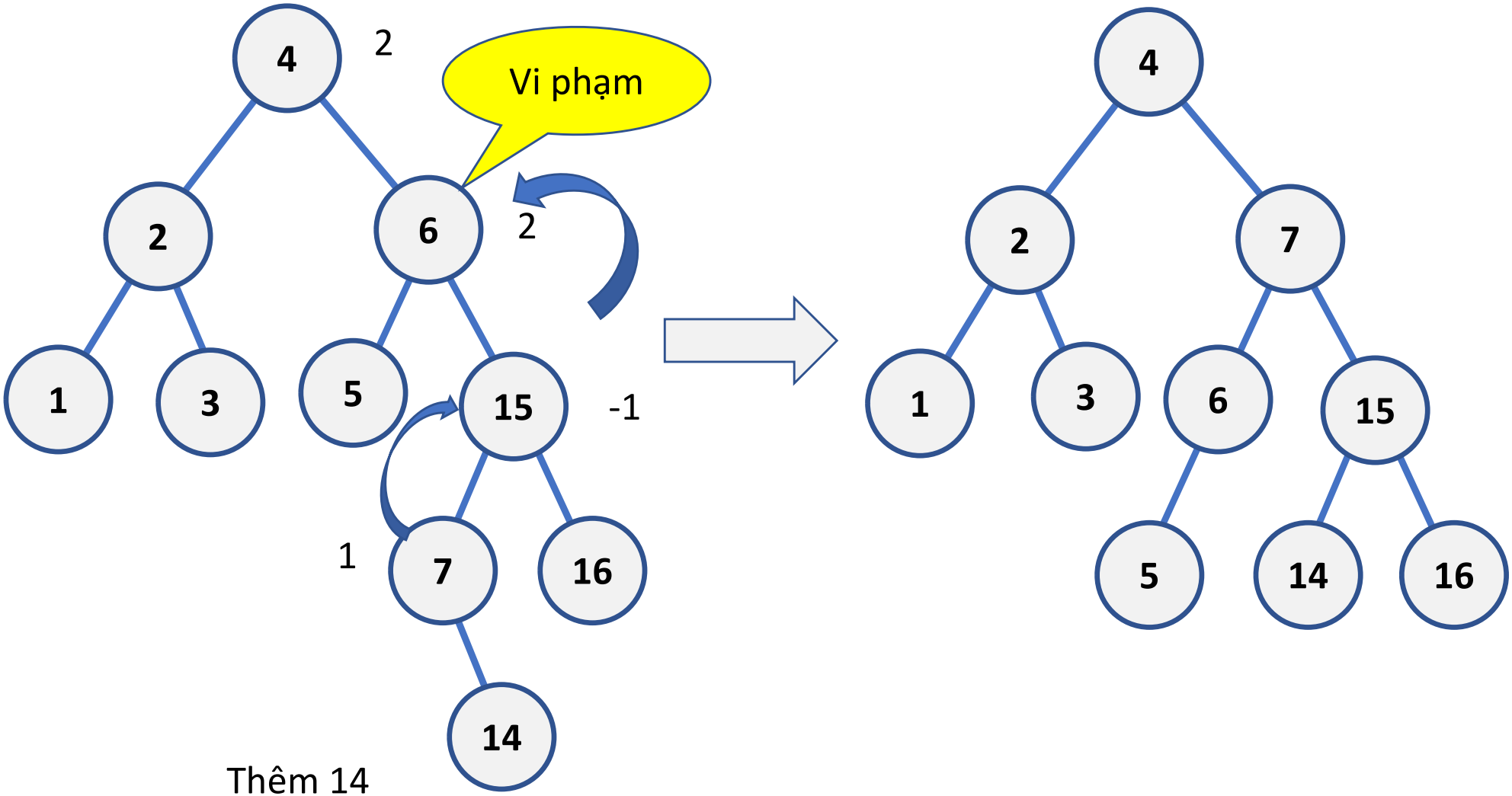
- Thực hiện thêm tiếp các khóa 16, 15, 14, 13, 12, 11, 10, 8, 9 vào cây



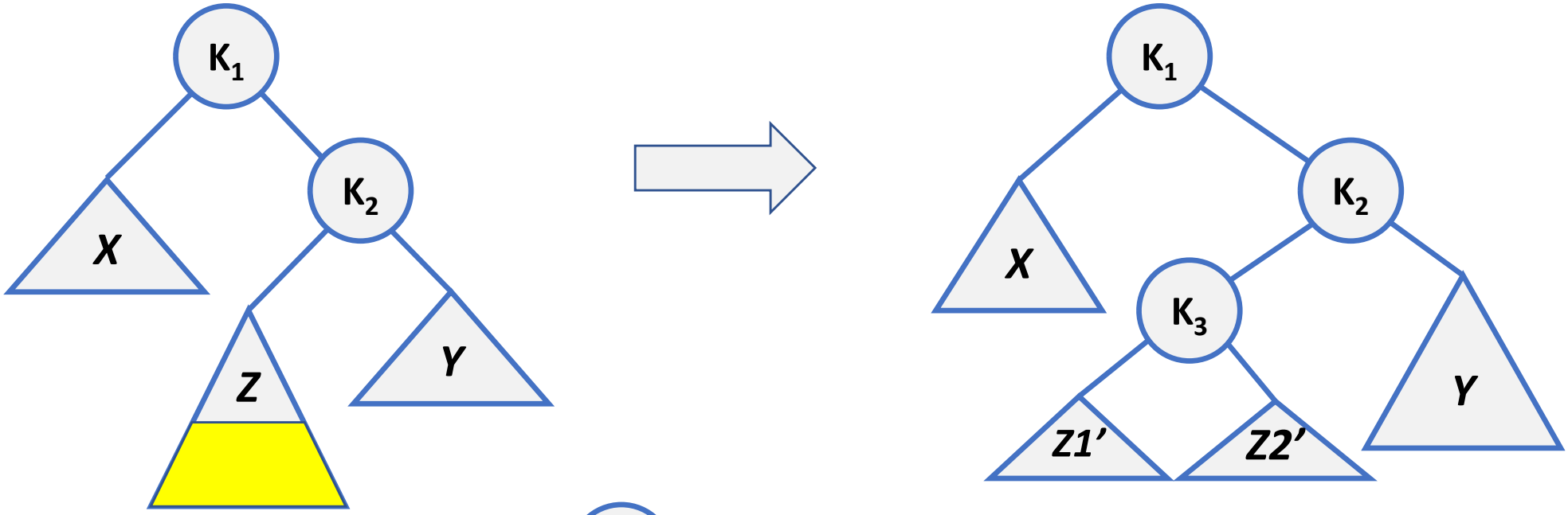
AVL tree



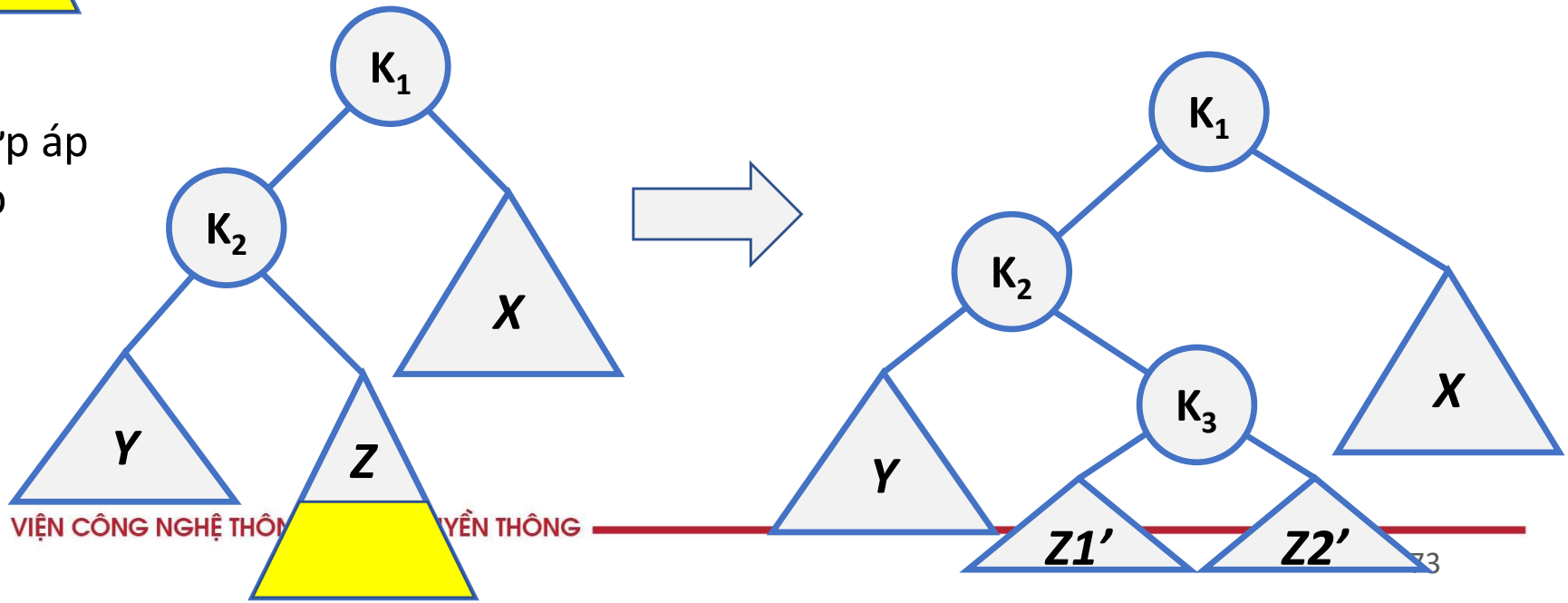
AVL tree



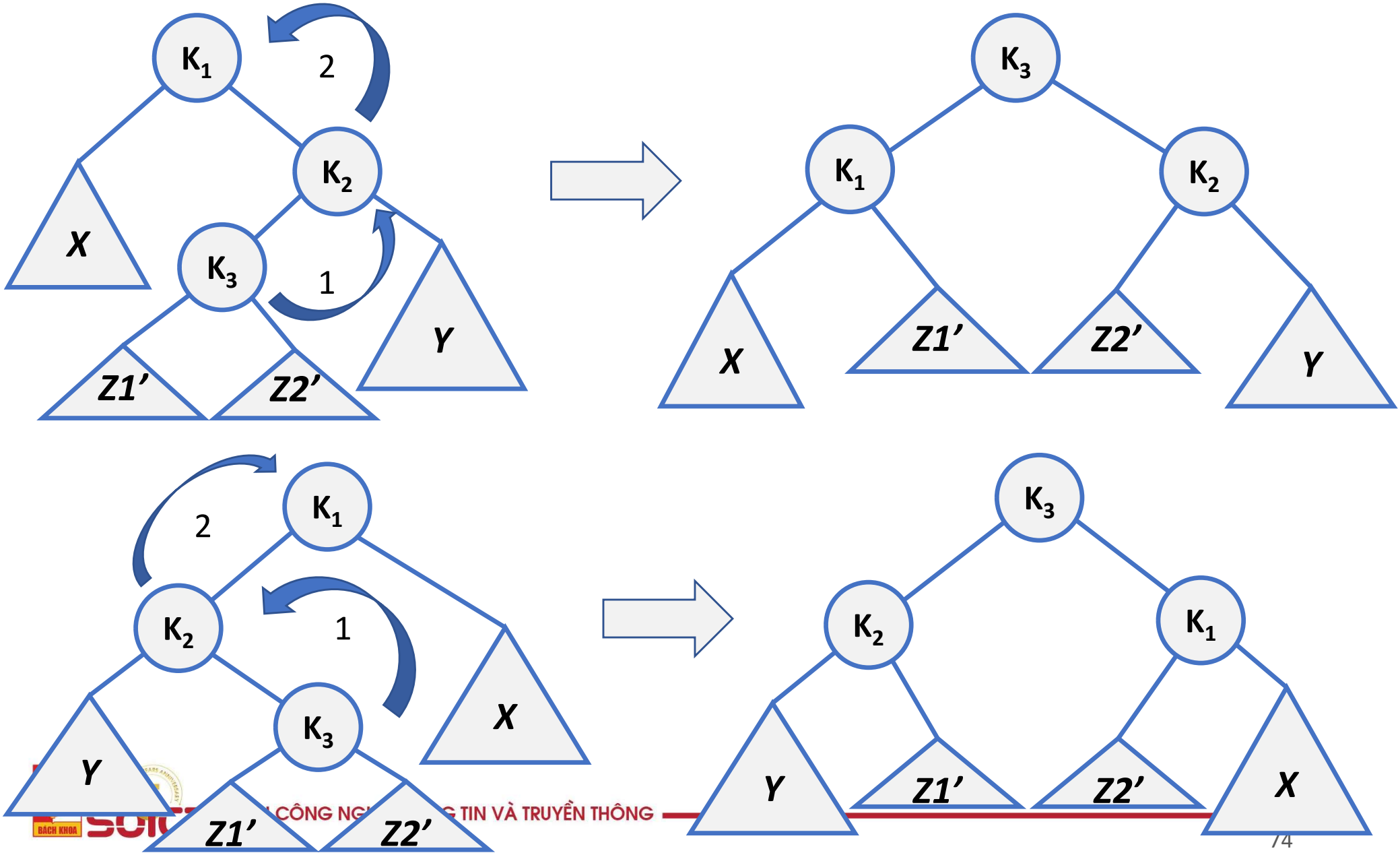
AVL tree



Hai trường hợp áp dụng xoay kép



AVL tree

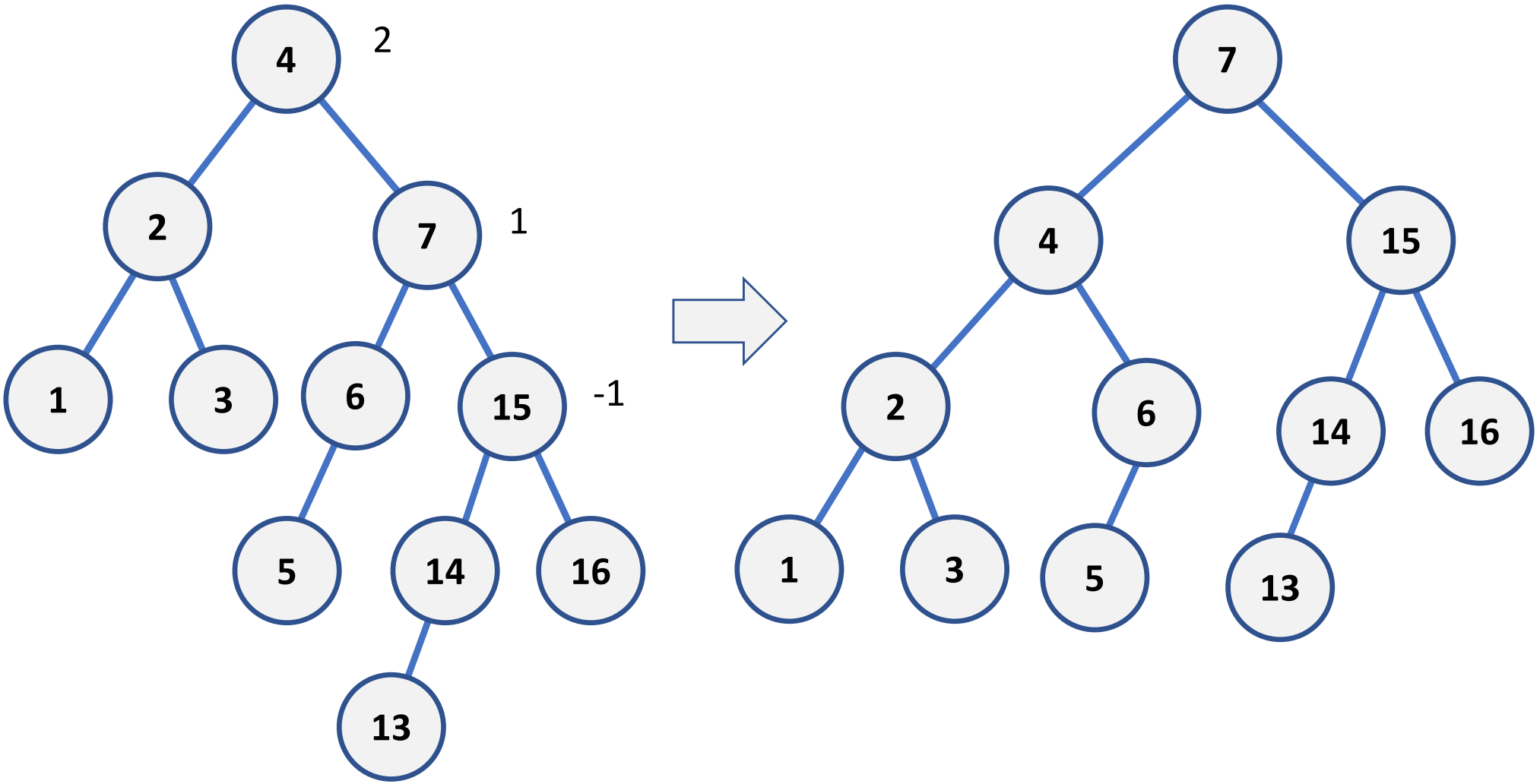

K1
2
K2
X
K3
1
Y
Z1'
Z2'
K3
K1
K2
X
Z1'
Z2'
Y
2
K1
K2
1
X
Y
K3
Z1'
Z2'
K3
K2
K1
Y
Z1'
Z2'
X

AVL tree

Phép xoay kép – double rotation:

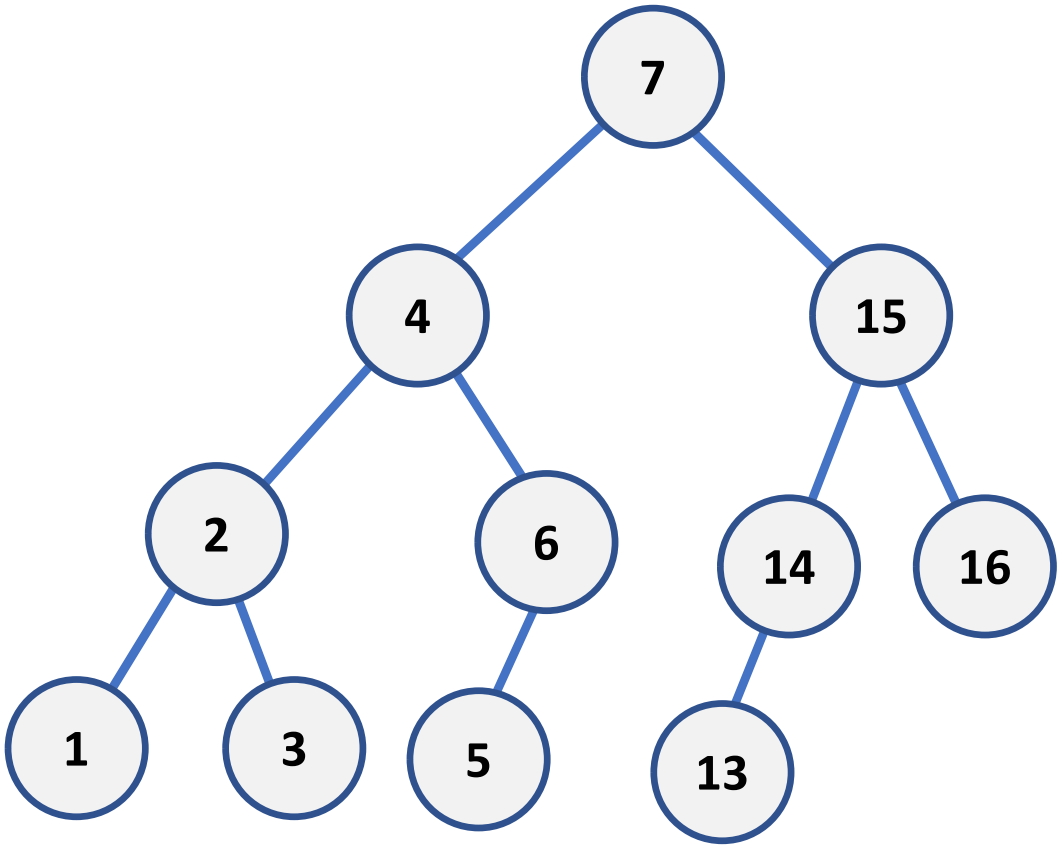
- Dùng để điều chỉnh khi mà nút mới thêm vào trong trường hợp:
  - (i) Cây con phải của nút con trái, hoặc
  - (ii)Cây con trái của nút con phải của nút
- Thực hiện tại nút vi phạm đầu tiên trên đường từ vị trí mới thêm trở về gốc
- Xoay giữa nút vi phạm, nút con, và nút cháu (con của nút con)
- Xoay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

AVL tree



Thêm 13

AVL tree



Thêm 12

AVL tree

Thêm 11

AVL tree

Thêm 10

AVL tree

Thêm 8

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

```
//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 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

```
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);
    }
}
```

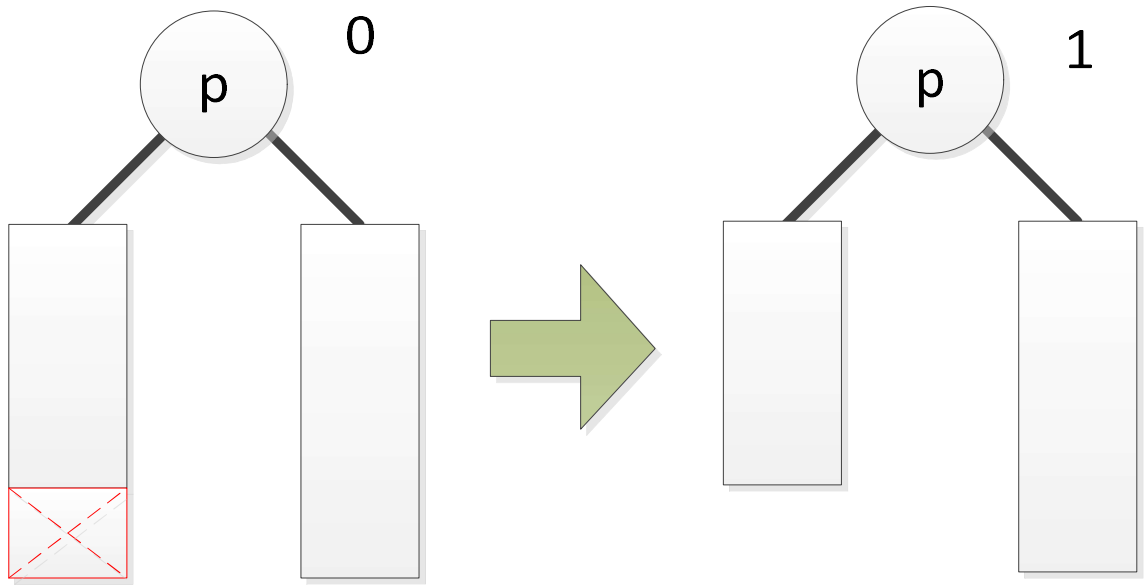
AVL tree

- Xóa nút khỏi cây:
  - 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 trỏ của nút cha nó bằng NULL

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)

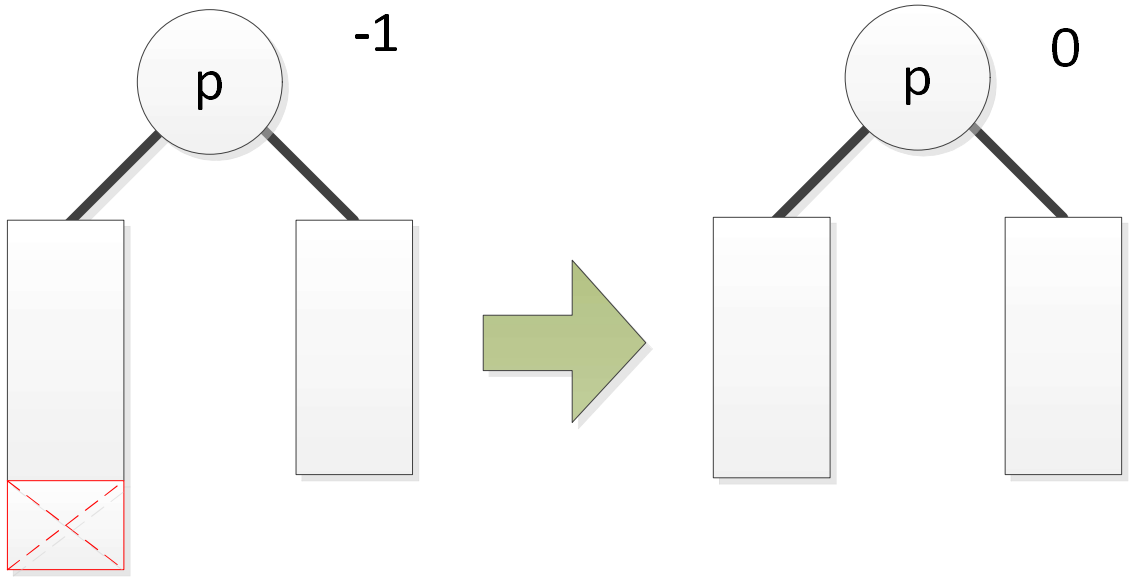
AVL tree



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**: 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.

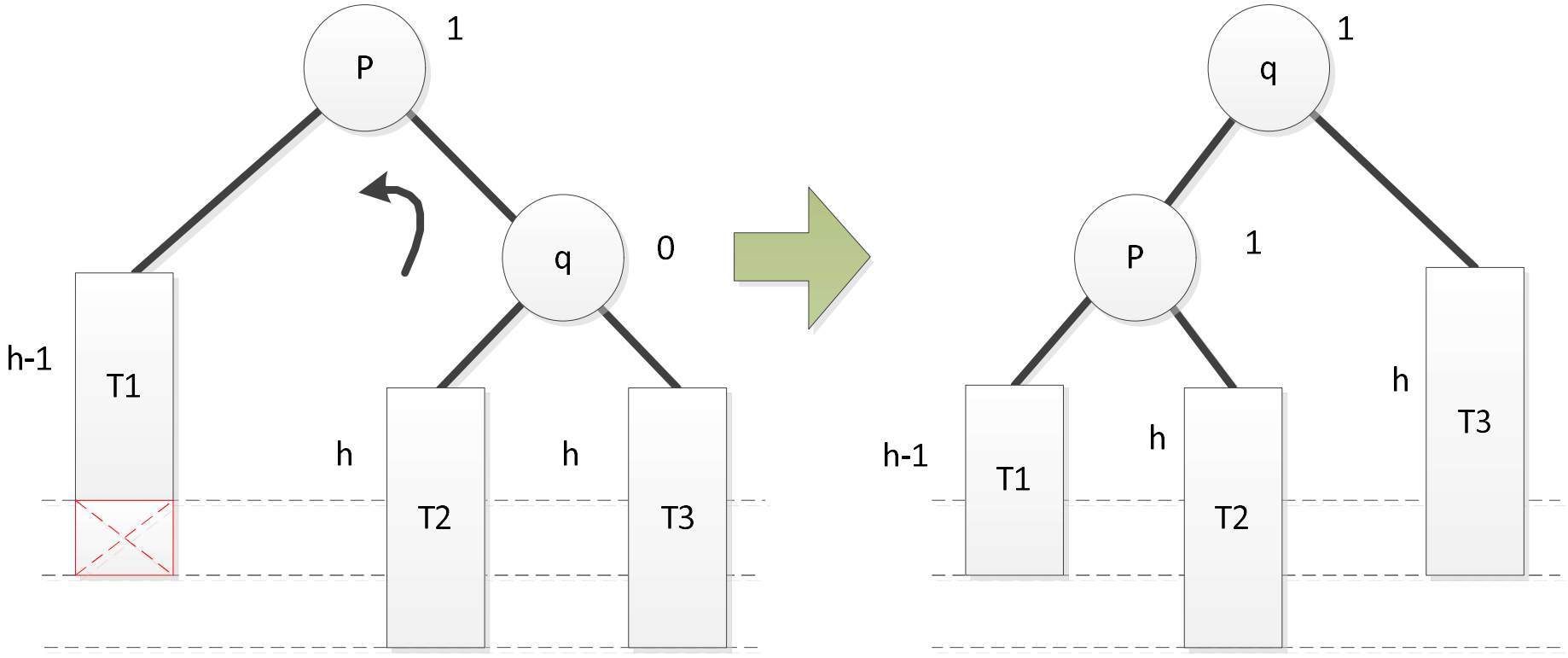
Căn cứ vào tình trạng cân bằng của nút con còn lại q của p mà ta chia thành các trường hợp nhỏ sau:

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 3.1

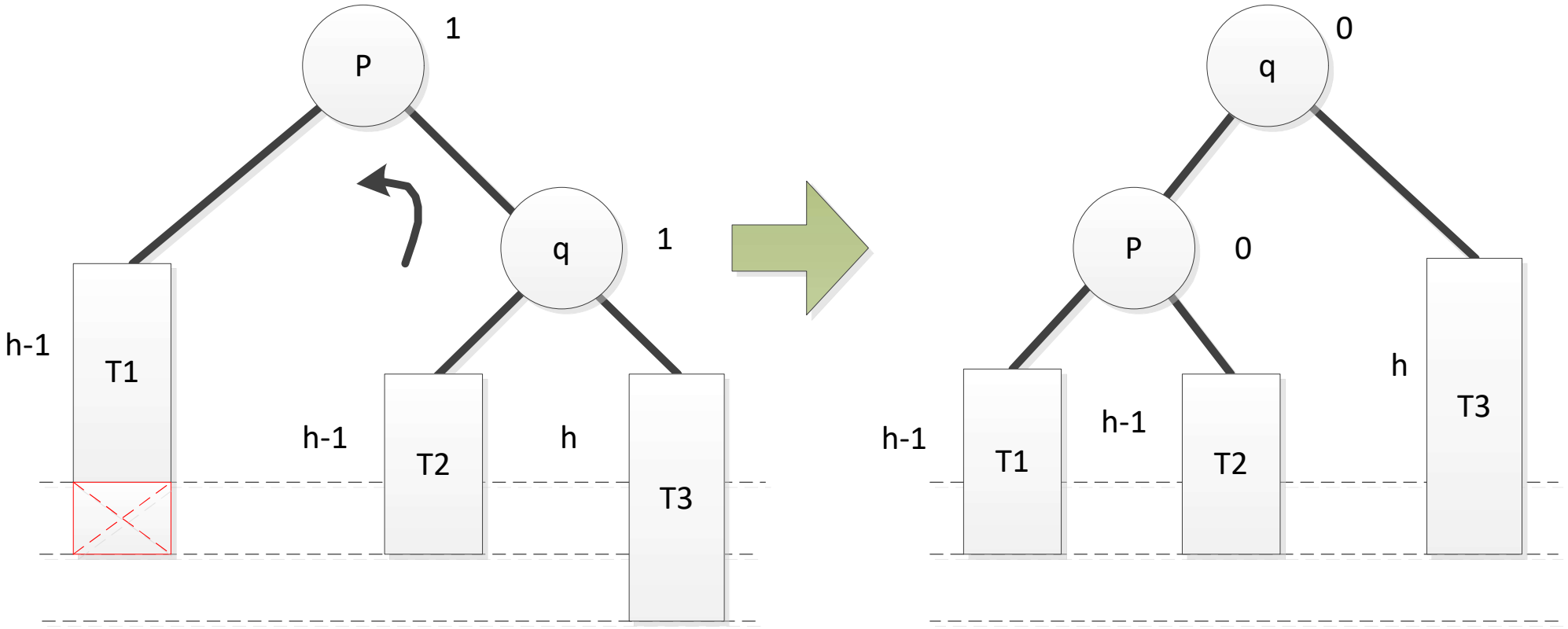


Chiều cao cây không đổi

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

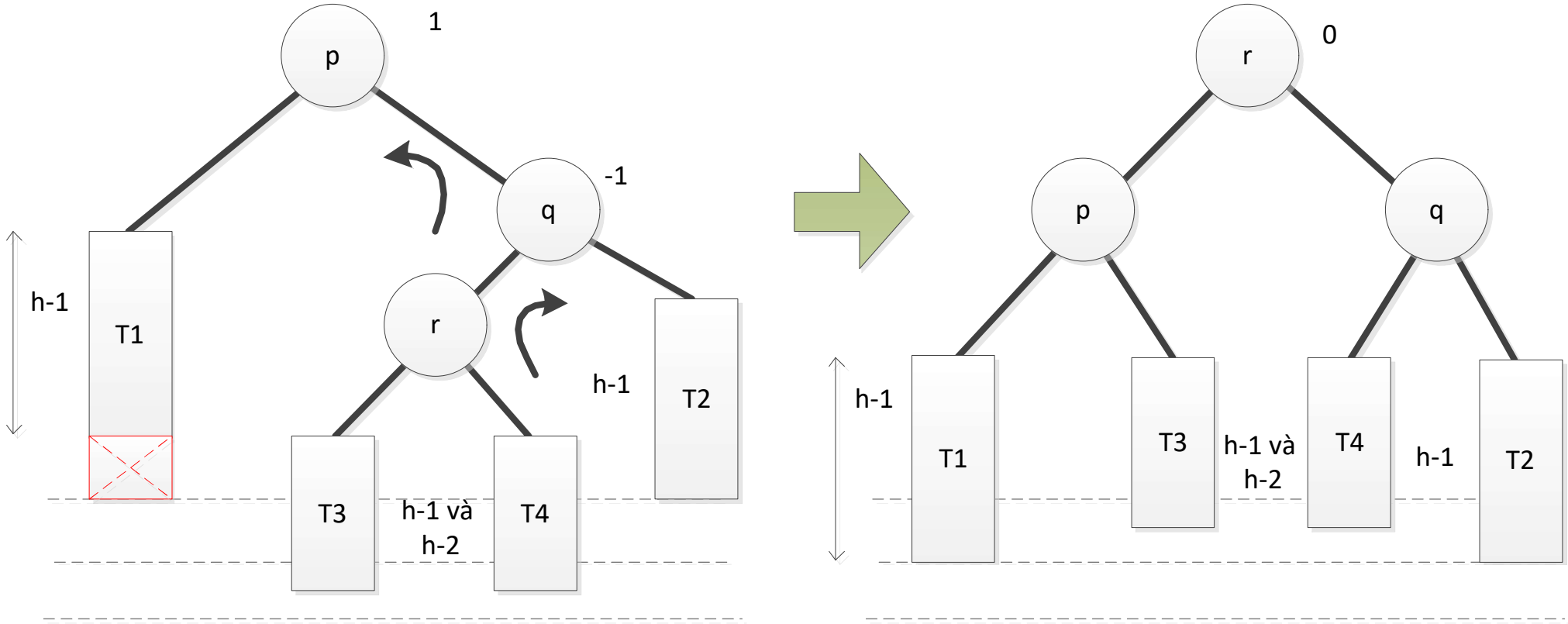


Chiều cao cây thay đổi

AVL tree

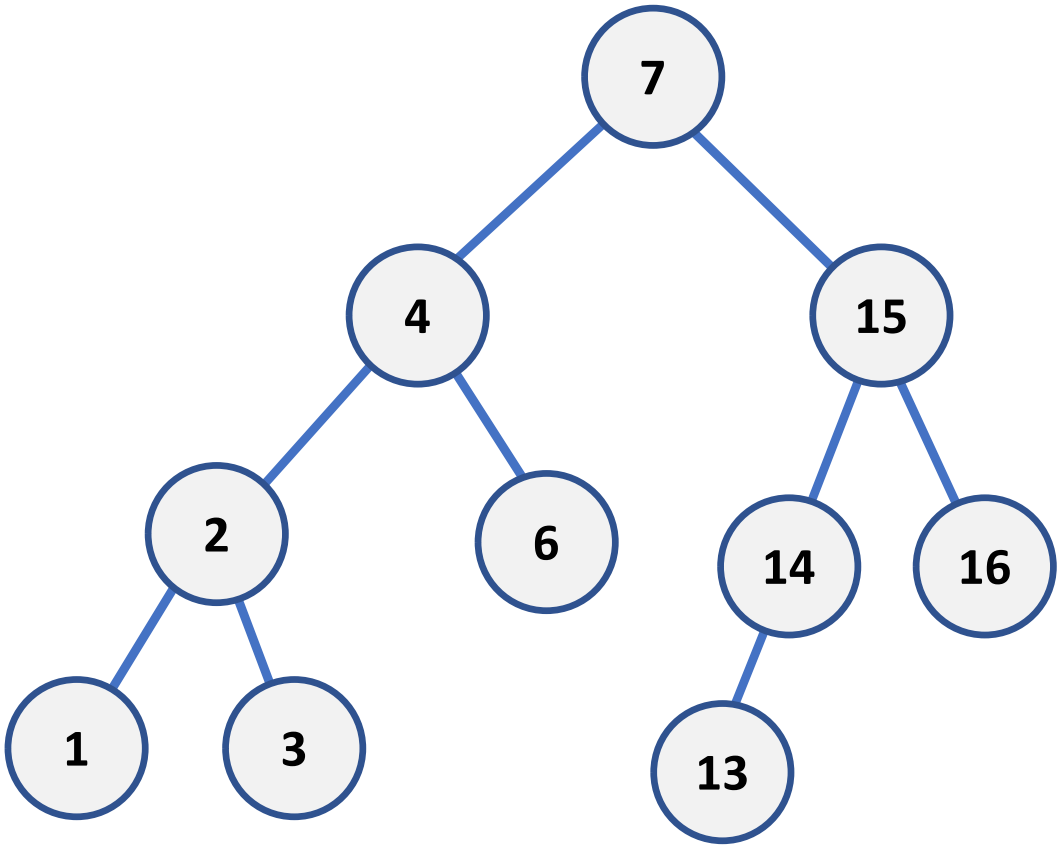
- Trường hợp 3.3: nút q bị lệch cùng phía với nhánh bị xóa. Nếu nhánh bị xóa là nhánh trái của p thì q bị lệch trái và ngược lại
  - Để tái cân bằng cho p ta phải thực hiện 2 phép xoay giữa nút con của q, nút q, và nút p
  - Sau khi xoay, chiều cao của cây giảm đi, p trở về trạng thái cân bằng

AVL tree



Chiều cao cây thay đổi

AVL tree



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

Question

Splay tree
Cấu trúc tự điều chỉnh

Cấy 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.

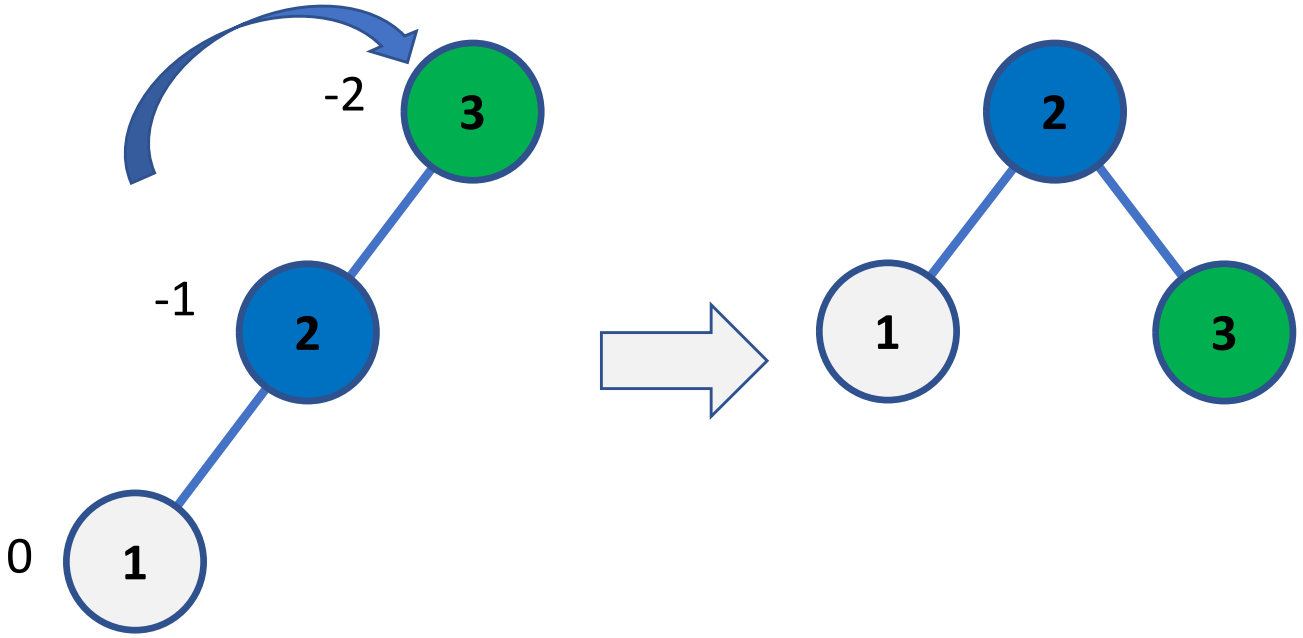
Cây splay

- **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

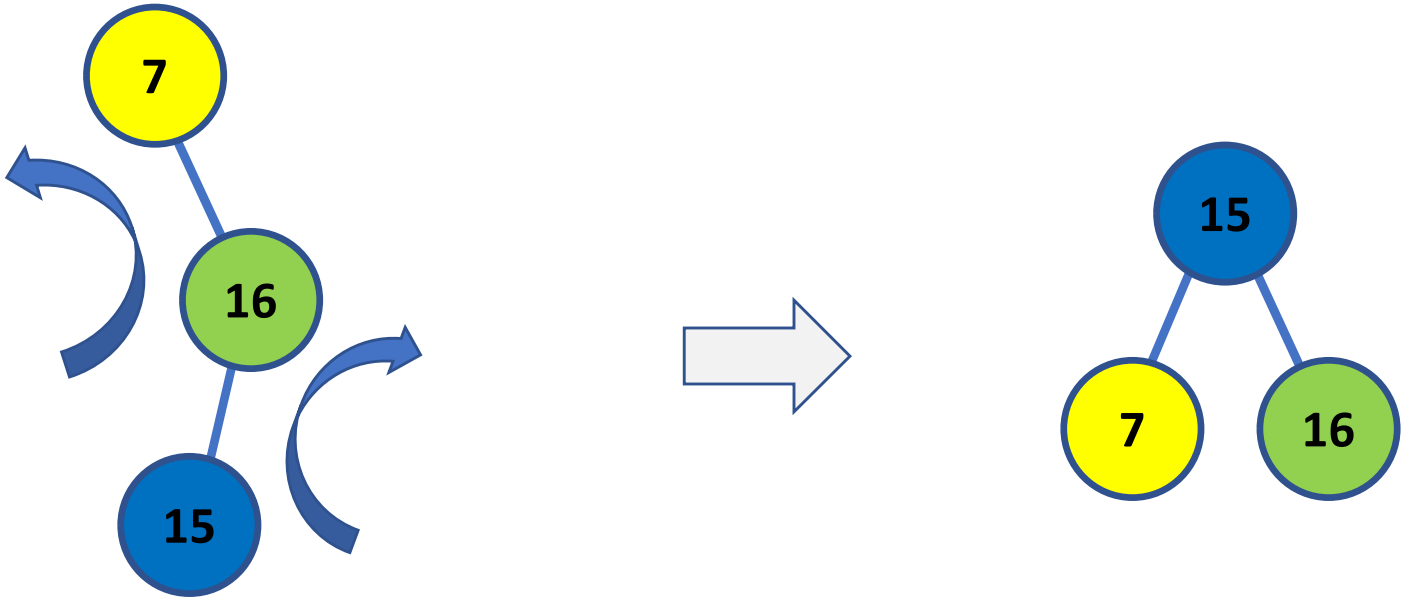
Cây splay

- **Nhắc lại về các phép xoay**
- **Xoay đơn – single rotation**:
  - Nút cha xuống thấp 1 mức và nút con lên 1 mức



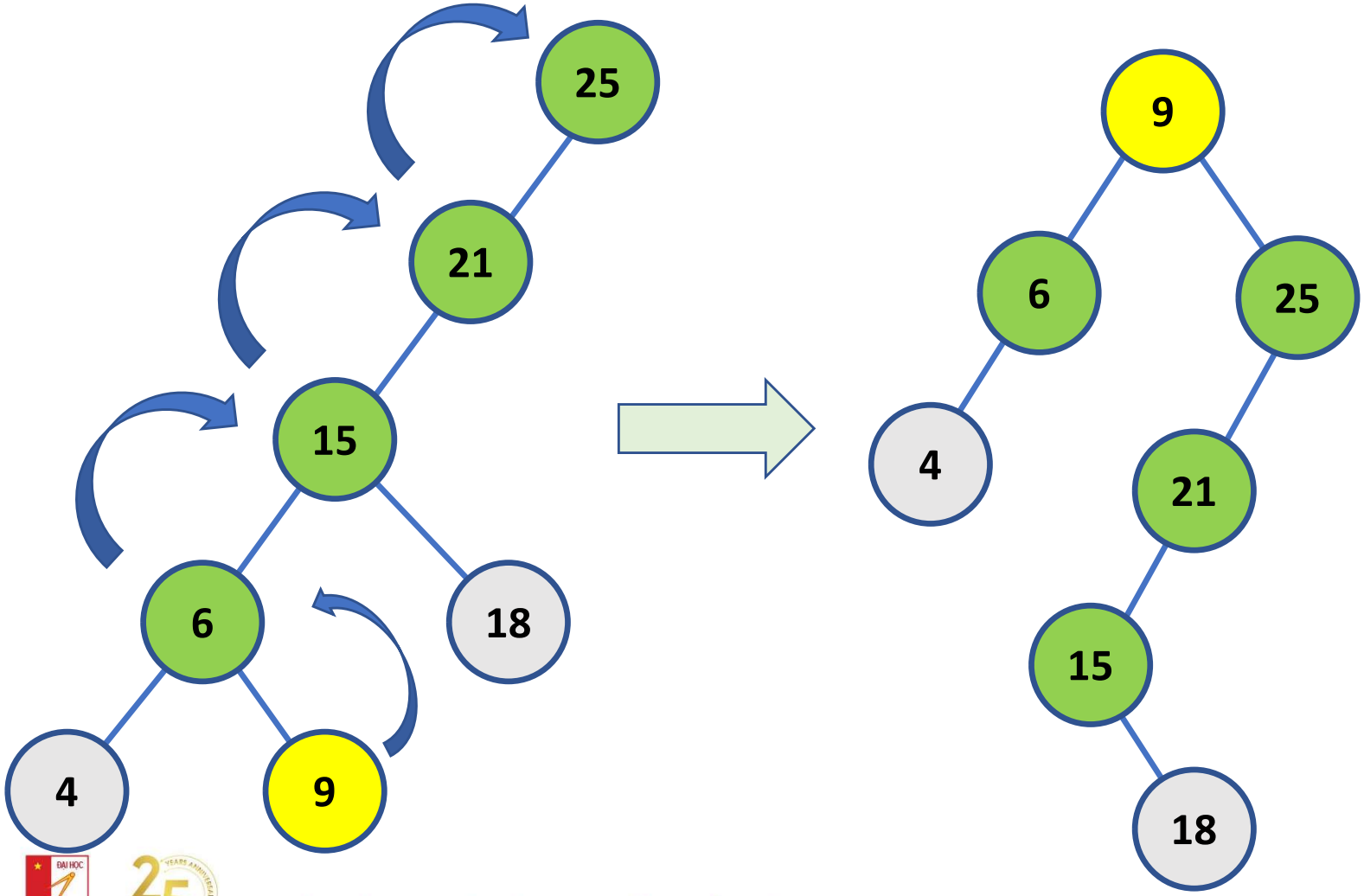
Cây splay

- **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

- Trường hợp chỉ dùng phép xoay đơn để điều chỉnh nút

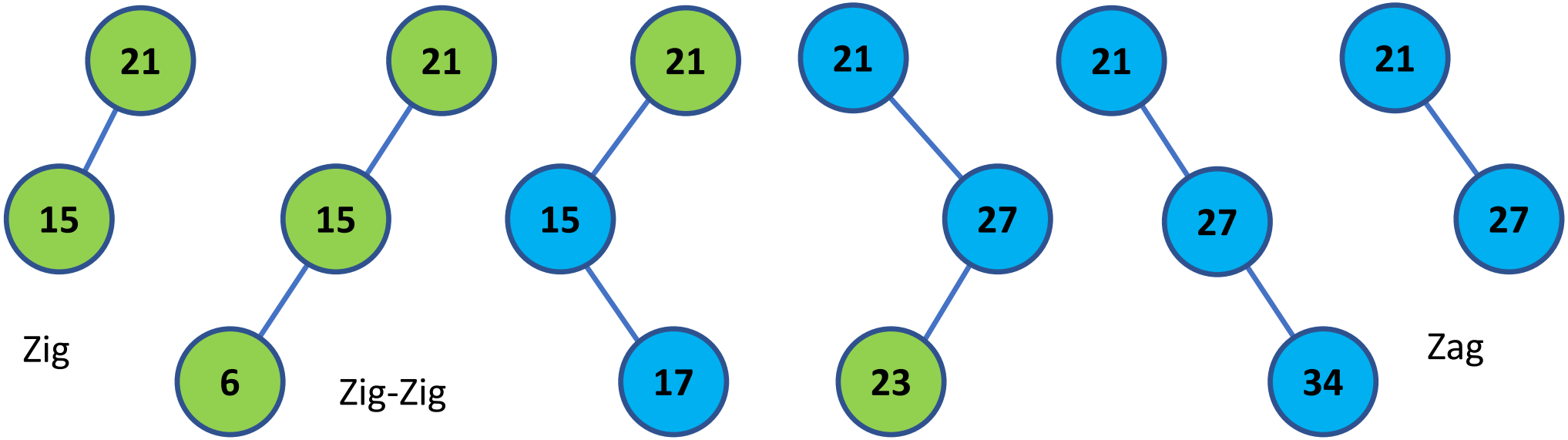


Cây splay

- **Nhận xét:**
  - Nút mới truy cập (nút 9) được chuyển thành nút gốc của cây mới
  - Tuy nhiên nút 18 lại bị đẩu 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***

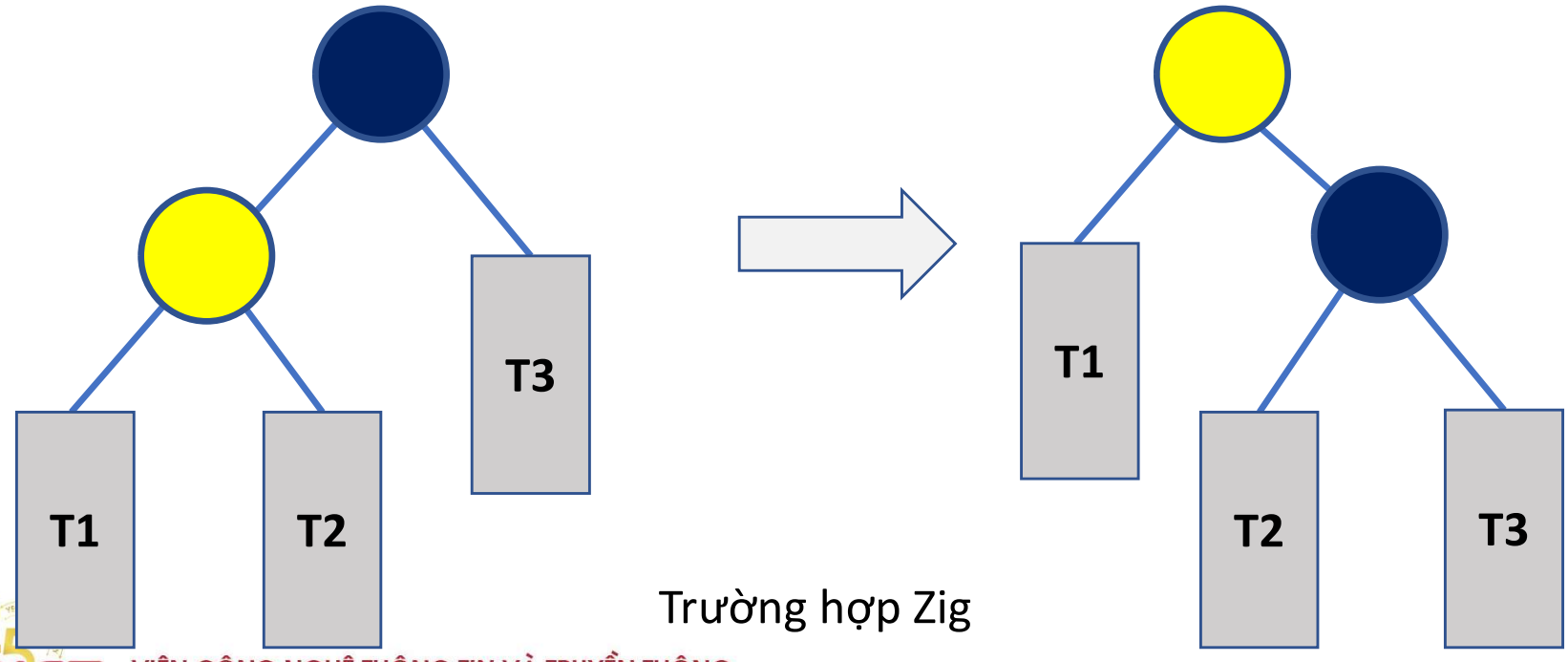
Cây splay

- Ý tưởng mới:
  - Tại mỗi bước ta di chuyển nút liền 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



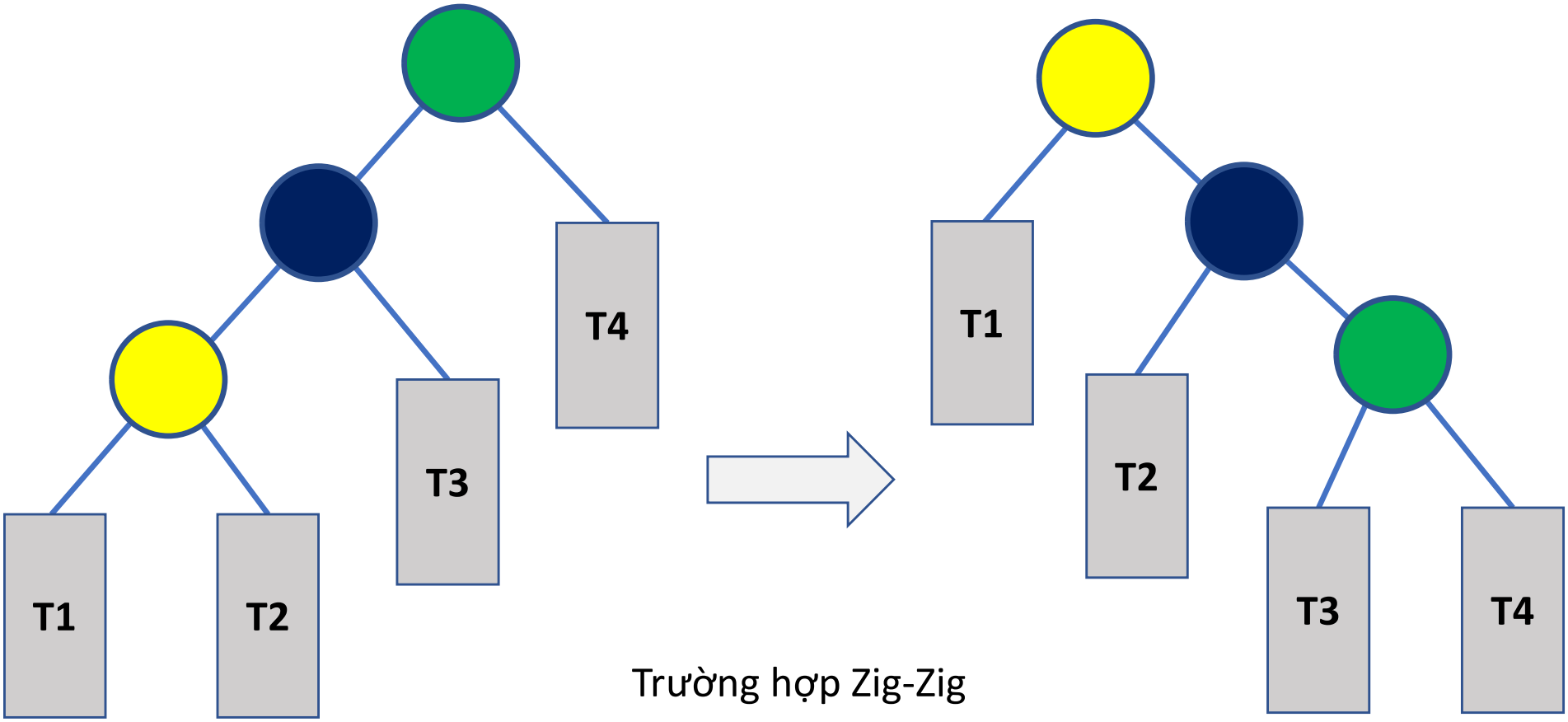
Cây splay

- **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)



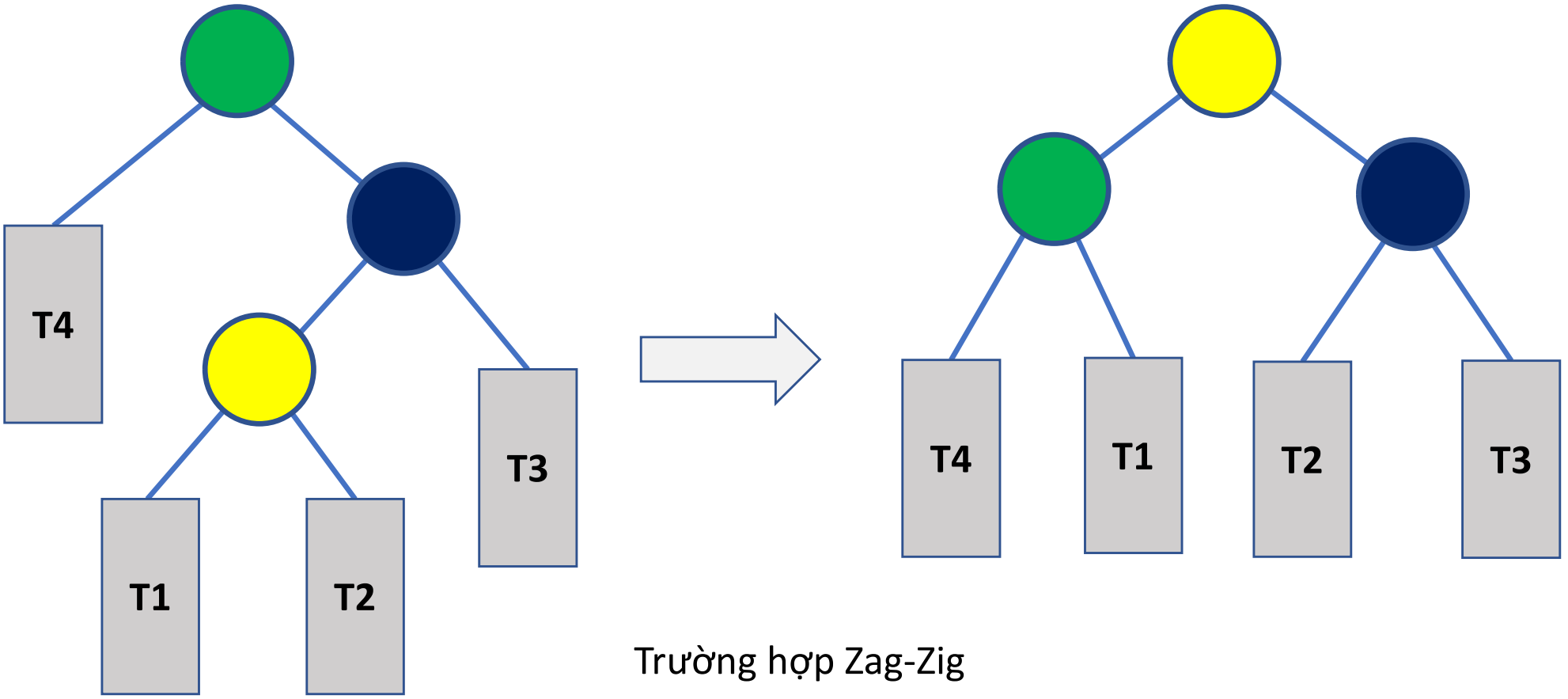
Trường hợp Zig

Cây splay



Trường hợp Zig-Zig

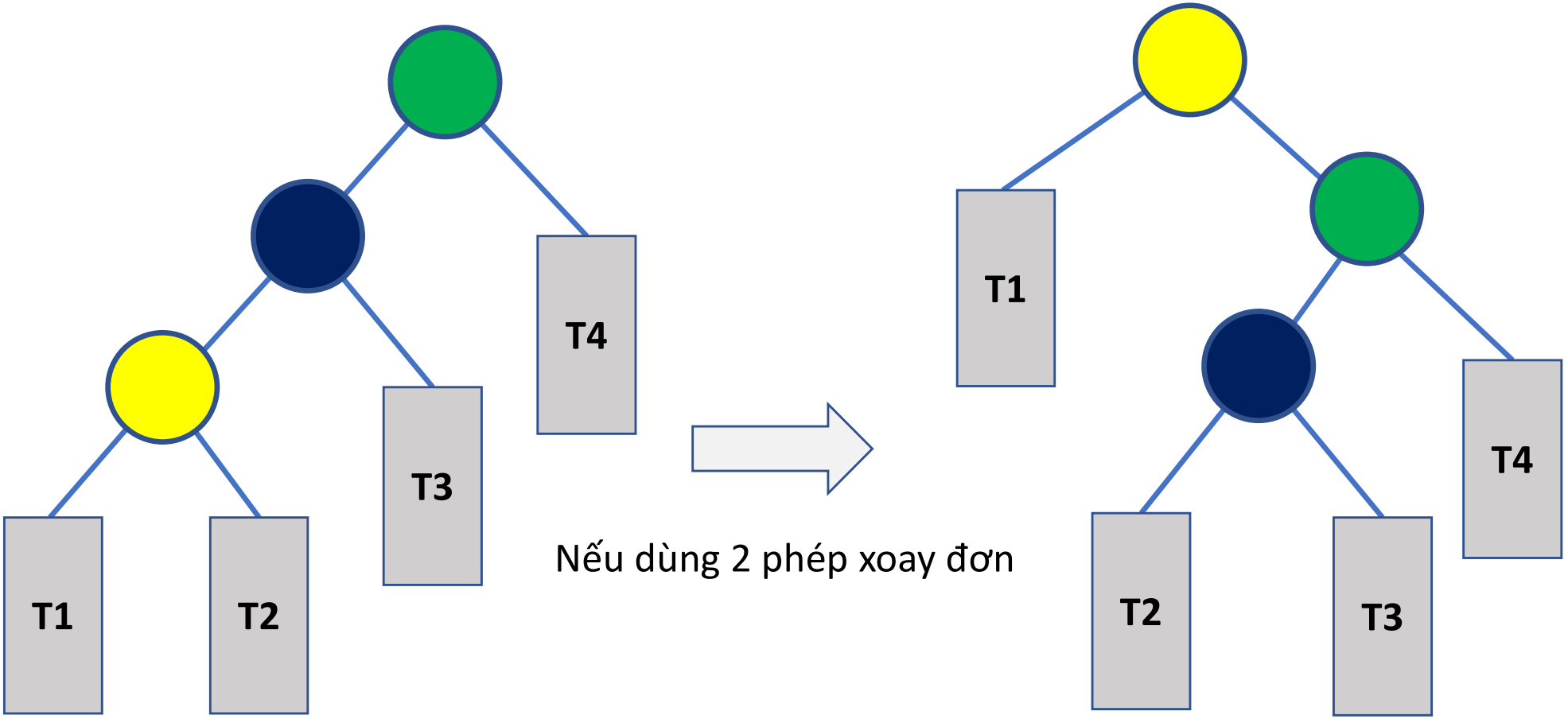
Cây splay



Trường hợp Zag-Zig

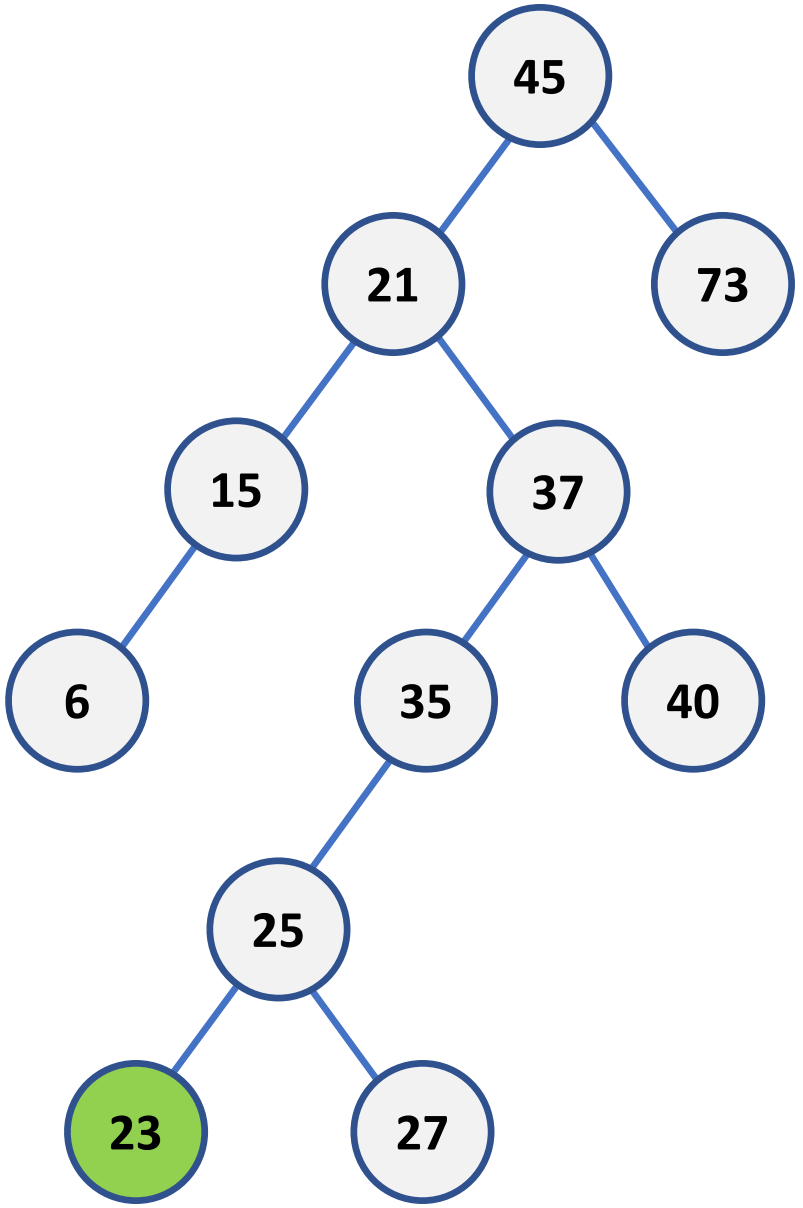
Cây splay

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



Cây splay

- Thực hiện splay tại nút 23



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

Phần 4. Bảng băm

- Bảng băm
- Một số phương pháp xây dựng hàm băm
- Đụng độ và khắc phục
- Bài tập

Băm

- Các phương pháp tìm kiếm :
  - Tìm kiếm tuần tự: O(n)
  - Tìm kiếm nhị phân O(log$n$)

- Phương pháp tổ chức dữ liệu để tìm kiếm nào tốt hơn ?

Bảng băm

Băm

- Ý tưởng bảng băm:
  - Dùng mảng lưu trữ các phần tử (bảng)
  - Mỗi phần tử sẽ được lưu tại một ô duy nhất trong bảng căn cứ vào giá trị khóa của nó,
  - Khi tìm kiếm thì căn cứ vào khóa cần tìm ta tìm đến ô tương ứng
  - Nếu ô đó có chứa phần tử thì tìm thấy, ngược lại là không tìm thấy

Băm

- VD. Sinhvien(1222, ‘Nguyễn văn A’, ‘Hà Nội’)
  Sinhvien(1101,’Trần văn C’,’Thái Bình’)
  Sinhvien(0097, ‘Nguyễn văn A’, ‘Hà Nội’)
  Sinhvien(1331,’Trần văn C’,’Thái Bình’)
  Sinhvien(1345, ‘Nguyễn văn A’, ‘Hà Nội’)
- Sử dụng bảng kích thước 100 để lưu các phần tử

Băm

- Tốt hơn tìm kiếm tuần tự khi mà dữ liệu được lưu trữ không có thứ tự.
- Không dựa trên so sánh giá trị các khóa của bản ghi mà dựa trên chính bản thân giá trị các khóa.
- Từ giá trị khóa ta tính ra một địa chỉ(địa chỉ tương đối) theo một quy tắc nào đó. Địa chỉ này sẽ dùng để lưu trữ bản ghi và cũng để tìm kiếm bản ghi đó.

hash function: key → an index

Địa chỉ lưu trữ

- **Hàm băm hoàn hảo**: mỗi khóa ứng với một giá trị nguyên duy nhất
- **Thời gian tìm kiếm O(1) ???**

Băm

- Hàm băm:
  - Tính toán dễ và nhanh
  - Phân phối đều các khóa

Một số phương pháp thông dụng:

- **Cắt bỏ (Truncation)** : dùng một phần của khóa làm chỉ số

  VD: khóa có 8 chữ số và bảng băm kích thước 1000 lấy chữ số thứ 4, 7 và 8 làm chỉ số
  21296876→ 976

  Nhanh nhưng phân bố khóa không đều!

Băm

- **Gấp (folding)**: chia khóa thành nhiều phần sau đó kết hợp các phần lại (thường dùng cộng hoặc nhân)

  VD. 21296876 chia thành 3 phần 212, 968 và 76 kết hợp 212+968+76 = 1256, cắt bỏ được 256

  Giá trị các thành phần trong khóa đều ảnh hưởng tới chỉ số. Cho phân phối tốt hơn phương pháp cắt bỏ

Băm

- **Phương pháp chia module**: lấy số dư của phép chia giá trị khóa cho kích thước của bảng băm để làm địa chỉ

$$h(k) = k \% m$$

- Thường chọn m là số nguyên tố nhỏ hơn, gần với kích thước bảng băm.
  - Bảng băm kích thước 1000 thì chọn m=997

Giá trị khóa		địa chỉ
5402		417
367	367	
1246		249
2983		989

Băm

- **Phương pháp nhân**: giá trị khóa được nhân với chính nó, sau đó lấy một phần kết quả để làm địa chỉ băm

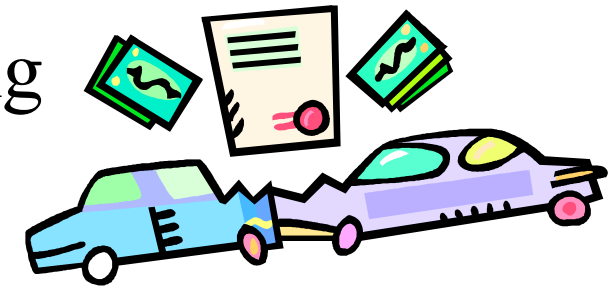
```
Giá trị khóa k k*k       địa chỉ

5402          29181604 181
367           00134689 134
1246          01552516 552
2983          08898289 898
```

Băm

- **Đụng độ**: hai khóa khác nhau có cùng giá trị chỉ số

  VD: hai khóa 21296876, 11391176

  Trong hàm băm dùng phương pháp cắt bỏ



- Giải pháp xử lý đụng độ:
  - Phương pháp đánh địa chỉ mở (Open Addressing)
  - Phương pháp xích ngăn cách (Separate Chaining)

Phương pháp đánh địa chỉ mở

Phương pháp đánh địa chỉ mở - Open Addressing

- **Dò tuyến tính**(Linear Probing): tại vị trí đụng độ, thực hiện tìm kiếm tuần tự để tìm ra khóa (khi tìm kiếm) hoặc vị trí trống (khi thêm mới)

VD: hàm băm

```
i=k%10
```

Các khóa {89, 18, 49, 58, 69} được thêm lần lượt vào bảng băm ban đầu rỗng

Phương pháp đánh địa chỉ mở

stt	Ban đầu	Thêm 89	Thêm 18	Thêm 49	Thêm 58	Thêm 69
0				49	49(*)	49(*)
1					58	58(*)
2						69
3						
4						
5						
6						
7						
8			18	18	18(*)	18
9		89	89	89(*)	89(*)	89(*)

Phương pháp đánh địa chỉ mở

- **Dò tuyến tính**
  - Xu hướng tạo thành các cụm khi bảng bắt đầu gần đầy nửa
  - Vị trí lưu trữ của khóa trong bảng và giá trị chỉ số ngày càng cách nhau xa, chi phí thực hiện tìm kiếm tuần tự ngày càng lớn

- **Khắc phục**: sử dụng các phương pháp lựa chọn vị trí phức tạp khi xảy ra đụng độ

  VD. Phương pháp băm lại (rehashing) sử dụng hàm băm thứ 2 để tạo chỉ số khi xảy ra đụng độ, nếu lại đụng độ thì sử dụng hàm băm thứ 3 ….

Phương pháp đánh địa chỉ mở

- **Dò bậc hai, dò toàn phương** (Quadratic Probing): nếu xảy ra đụng độ tại địa chỉ h, thì ta sẽ tìm kiếm tiếp theo tại các địa chỉ h+1, h+4, h+9,.. ; là các địa chỉ $h+i^2$

VD: hàm băm

```
i=k%10
```

Các khóa {89, 18, 49, 58, 69} được thêm lần lượt vào bảng băm ban đầu rỗng

Phương pháp đánh địa chỉ mở

stt	Ban đầu	Thêm 89	Thêm 18	Thêm 49	Thêm 58	Thêm 69
0				49	49	49(*)
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18(*)	18
9		89	89	89(*)	89(*)	89(*)

Phương pháp đánh địa chỉ mở

- **Dò toàn phương:**
  - Giảm được sự phân nhóm
  - Không phải tất cả các vị trí trong bảng đều được dò

    VD. Khi kích thước bảng là mũ của 2 thì 1/6 vị trí được dò, là số nguyên tố thì 1/2 được dò

Phương pháp đánh địa chỉ mở

- **Dò theo khóa**: trong trường hợp đụng độ tại h thì dò tiếp tại vị trí cách vị trí đó khoảng cách bằng giá trị phần tử tiên trong khóa (là số hoặc là mã ASCII).

- Ví dụ: nếu khóa **2**`9181`**60** xảy ra đụng độ thì dò tại ô tiếp theo cách ô đụng độ 2 vị trí

Phương pháp đánh địa chỉ mở

- **Dò ngẫu nhiên** (Random Probing): sử dụng cách sinh số giả "ngẫu nhiên" để tạo ra vị trí dò tiếp. Cách sinh này phải là duy nhất với 1 giá trị khóa.

- Ví dụ: khóa 123245 thì sinh ra số ngẫu nhiên duy nhất là 5

Phương pháp đánh địa chỉ mở

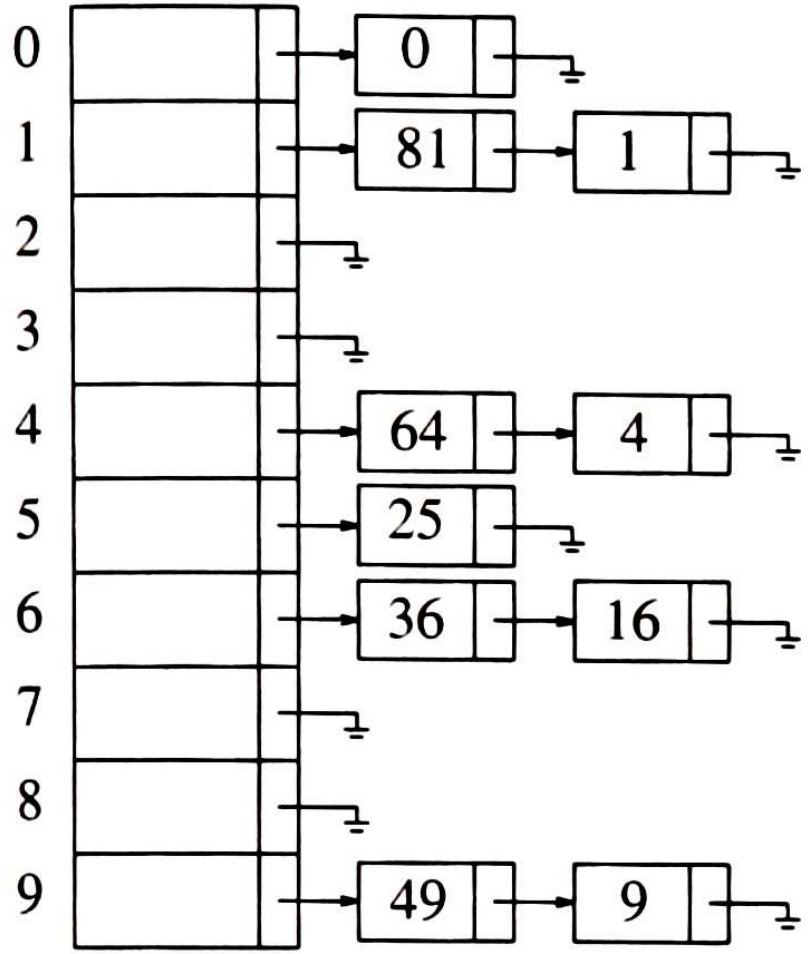
- **Chú ý**: không thể xóa phần tử trong bảng băm sử dụng phương pháp đánh địa chỉ mở theo cách thông thường!

  ➔ đánh dấu là xóa, nhưng vẫn được xét đến khi dò!

Giải pháp xử lý đụng độ

Phương pháp xích ngăn cách (Separate Chaining)

- Dùng mảng của danh sách móc nối
- **Ưu điểm**:
  - không bị giới hạn số phần tử,
  - Dễ thêm và xóa
  - xử lý đụng độ tốt
- **Nhược điểm** :
  - Phải lưu nhiều con trỏ NULL
  - Thực hiện tìm kiếm tuyến tính chậm



Băm

- Hàm băm tương ứng của số có 3 chữ số (a1 a2 a3) là **mod** (a1 + a2 + a3 , 5). Với các tổ hợp số dưới đây, giá trị băm bị xung đột là trường hợp nào?

A. 881 và 509

B. 913 và 426

C. 731 và 429

D. 102 và 297

E. 677 và 388

mod: chia lấy phần dư mod(5,3) = 2

Băm

Bài 2. Cho bảng băm với kích thước 13, chỉ số các phần tử từ 0 đến 12, và dãy khóa
10, 100, 32, 45, 58, 126, 3, 29, 200, 400, 0

a) Sử dụng hàm băm i=k%13, vẽ các bước khi thêm các khóa vào bảng sử dụng phương pháp xử lý đụng độ là dò tuyến tính và dò bậc hai.

b) Sử dụng hàm băm là tổng của các chữ số trong khóa chia lấy dư cho 13, vẽ lại bảng băm với hai phương pháp xử lý đụng độ như phần a

c) Tìm một hàm băm mà không xảy ra đụng độ cho dãy khóa trên

Nội dung

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict