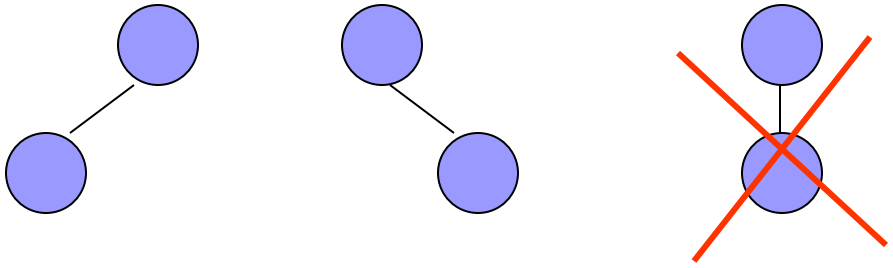
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Chương 5: Cây nhị phân

Định nghĩa

- Cây nhị phân
  - Cây rỗng
  - Hoặc có một node gọi là gốc (root) và 2 cây con gọi là cây con trái và cây con phải
- Ví dụ:
  - Cây rỗng:
  - Cây có 1 node: là node gốc
  - Cây có 2 node:



Các định nghĩa khác

- Mức:
  - Node gốc ở mức 0.
  - Node gốc của cây con có mức m thì các con có mức *m+1*.
- Chiều cao:
  - Cây rỗng là 0.
  - Chiều cao lớn nhất của 2 cây con cộng 1
  - (Hoặc: mức lớn nhất của các node cộng 1)
- Đường đi (path)
  - Tên các node của quá trình đi từ node gốc theo các cây con đến một node nào đó.

Các định nghĩa khác (tt.)

- Node trước, sau, cha, con:
  - Node x là trước node y (node y là sau node x), nếu trên đường đi đến y có x.
  - Node x là cha node y (node y là con node x), nếu trên đường đi đến y node x nằm ngay trước node y.
- Node lá, trung gian:
  - Node lá là node không có cây con nào.
  - Node trung gian không là node gốc hay node lá.

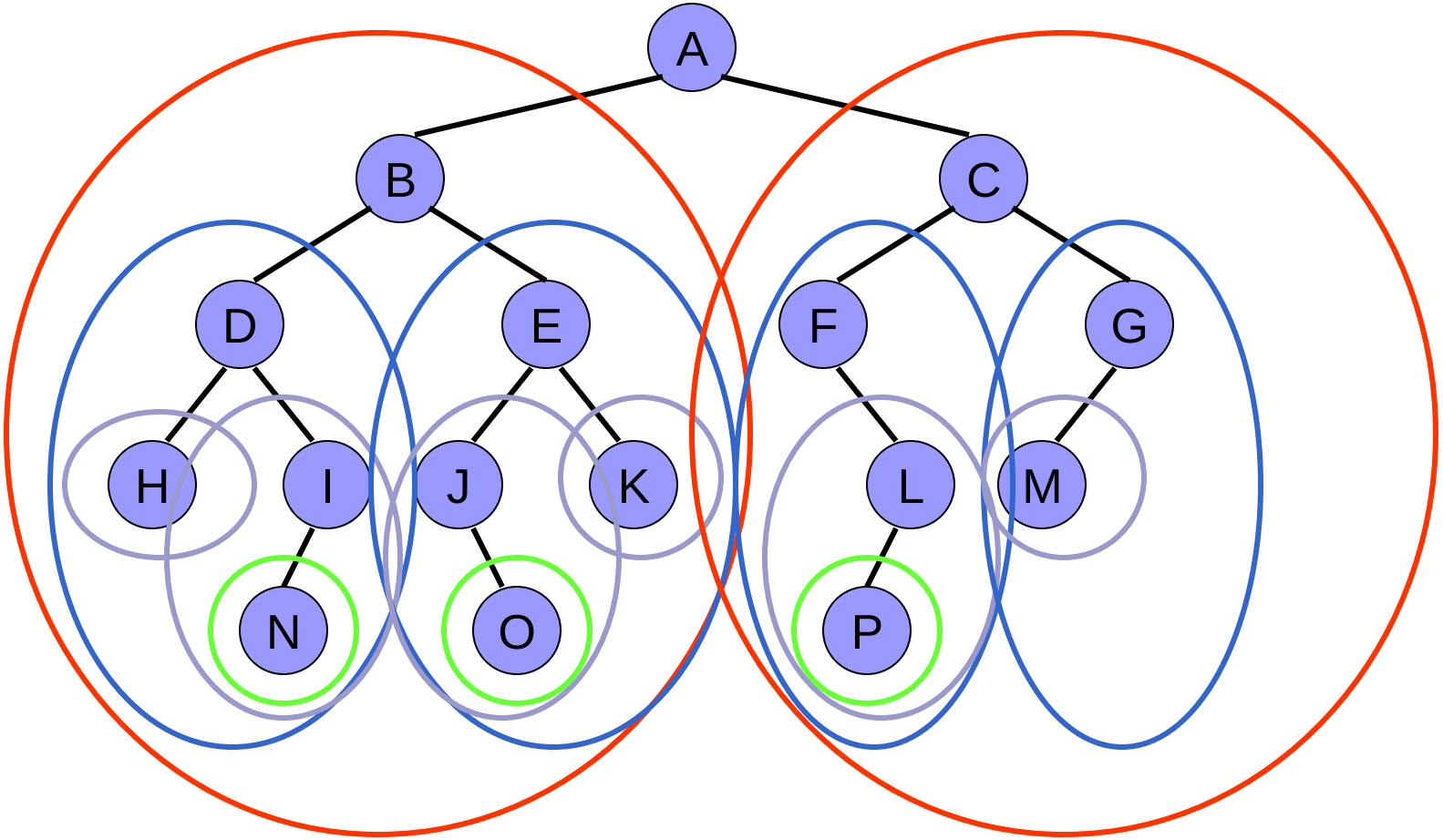
Các tính chất khác

- Cây nhị phân đầy đủ, gần đầy đủ:
  - Đầy đủ: các node lá luôn nằm ở mức cao nhất và các nút không là nút lá có đầy đủ 2 nhánh con.
  - Gần đầy đủ: Giống như trên nhưng các node lá nằm ở mức cao nhất (hoặc trước đó một mức) và lấp đầy từ bên trái sang bên phải ở mức cao nhất.
- Chiều cao của cây có *n* node:
  - Trung bình $h = [\log_2 n] + 1$
  - Đầy đủ $h = \log_2(n + 1)$
  - Suy biến $h = n$
- Số phần tử tại mức *i* nhiều nhất là 2^i

Phép duyệt cây

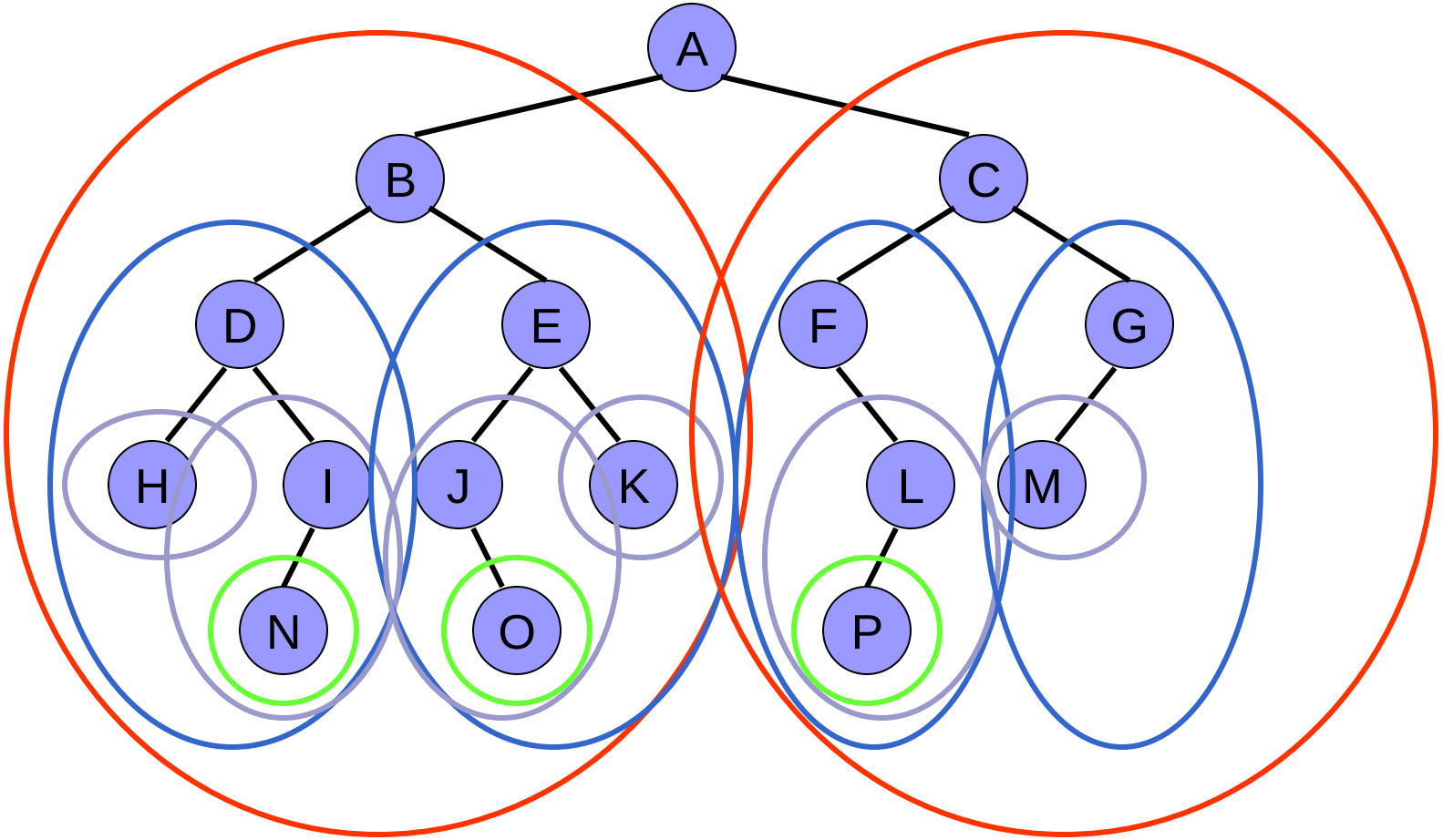
- Duyệt qua từng node của cây (mỗi node 1 lần)
- Cách duyệt:
  - Chính thức: NLR, LNR, LRN, NRL, RNL, RLN
  - Chuẩn: NLR (preorder), LNR (inorder), LRN (postorder)

Ví dụ về phép duyệt cây NLR



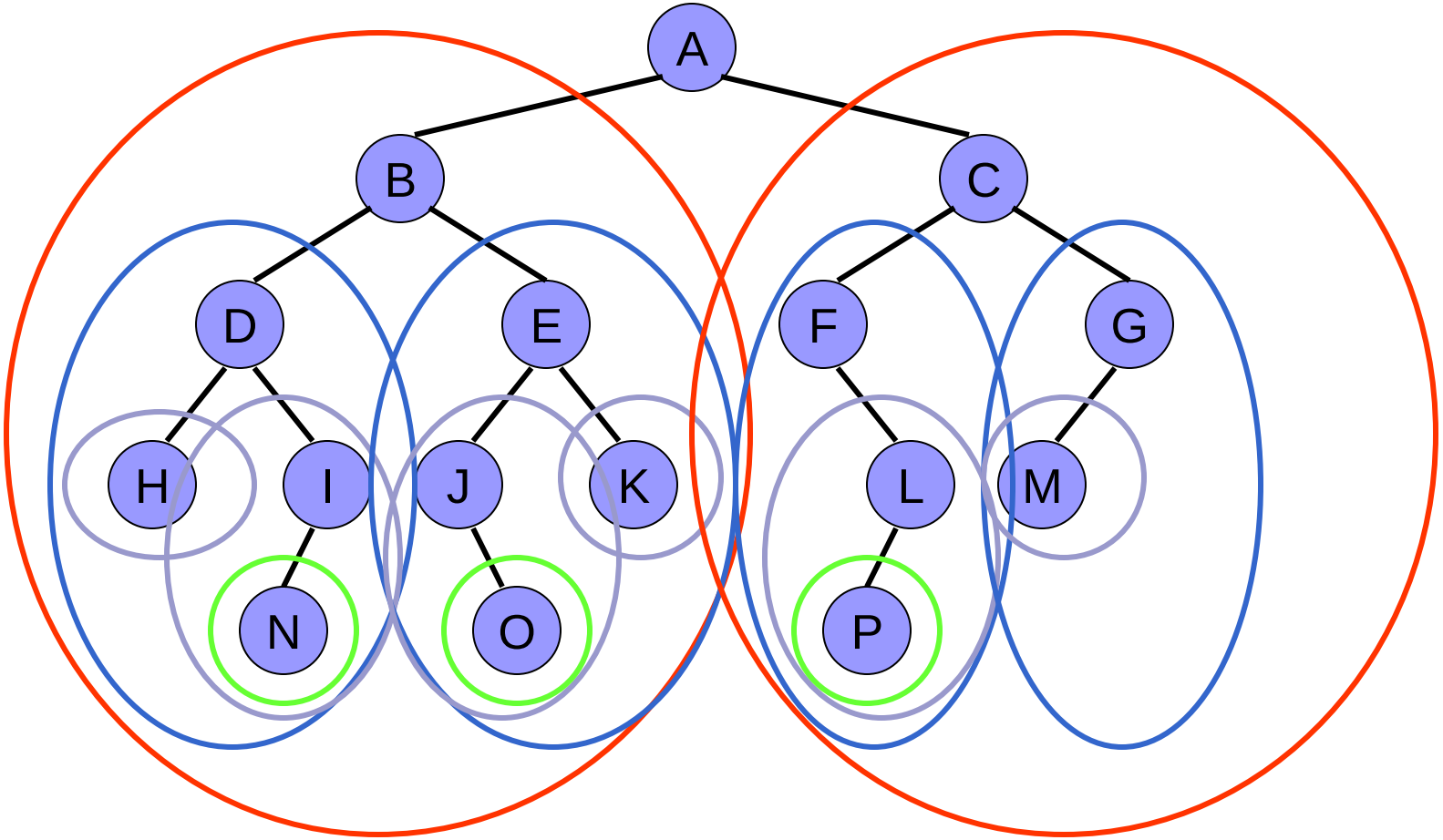
Kết quả: A B D H I N E J O K C F L P G M

Ví dụ về phép duyệt cây LNR



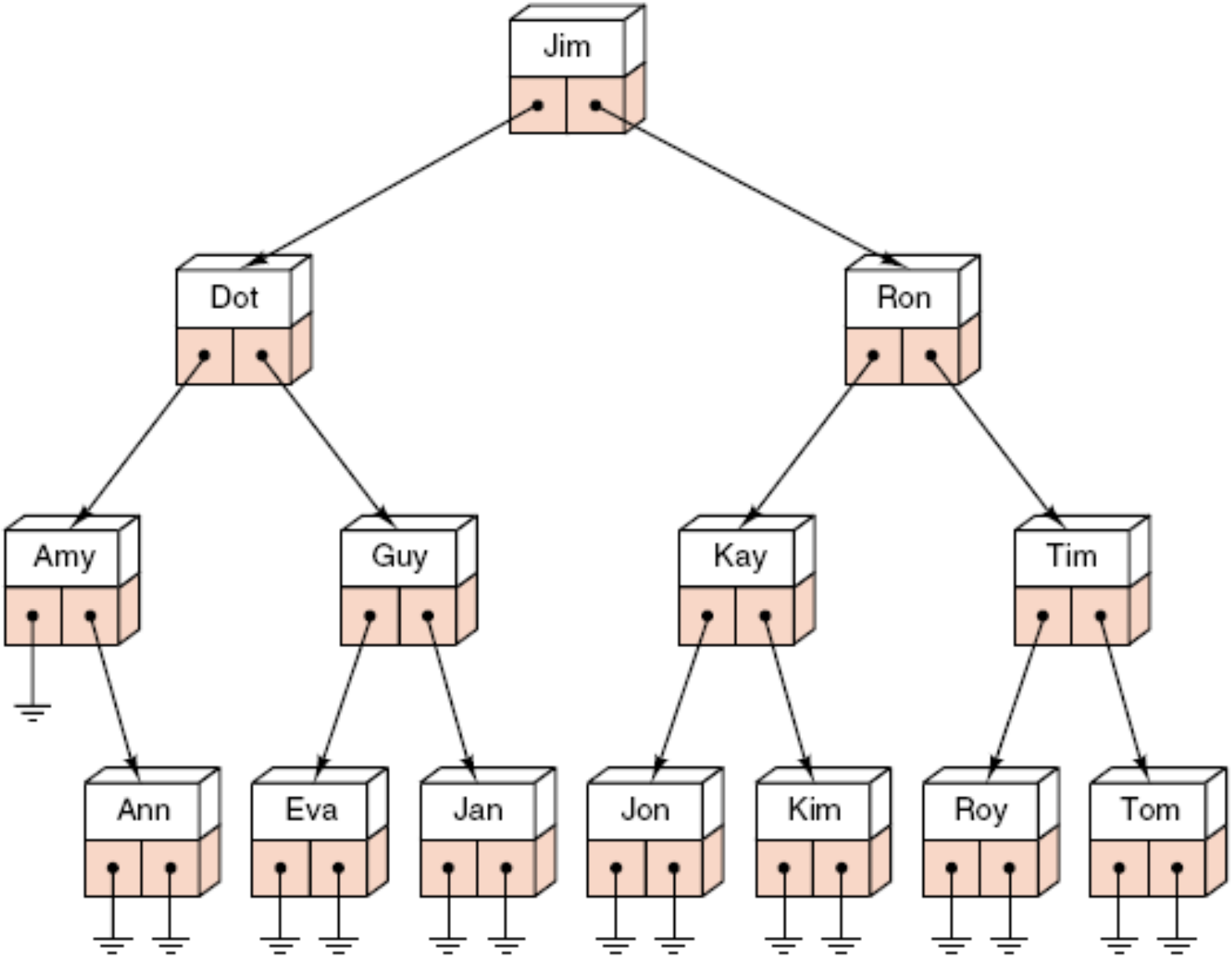
Kết quả: H D N I B J O E K A F P L C M G

Ví dụ về phép duyệt cây LRN



Kết quả: H N I D O J K E B P L F M G C A

Cây liên kết



Thiết kế cây liên kết

```
template <class Entry>
struct Binary_node {
    // data members:
    Entry data;
    Binary_node<Entry> *left, *right;
    // constructors:
    Binary_node( );
    Binary_node(const Entry &x);
};
```

```
template <class Entry>
class Binary_tree {
public:
    // Add methods here.
protected:
    // Add auxiliary function prototypes here.
    Binary_node<Entry> *root;
};
```

Khởi tạo và kiểm tra rỗng

```
template <class Entry>
Binary_tree<Entry>::Binary_tree() {
    root = NULL;
};
```

```
template <class Entry>
bool Binary_tree<Entry>::empty() {
    return root == NULL;
};
```

Thiết kế các phép duyệt cây

```
template <class Entry>
void Binary_tree<Entry> :: inorder(void (*visit)(Entry &)) {
    recursive_inorder(root, visit);
}
```

```
template <class Entry>
void Binary_tree<Entry> :: preorder(void (*visit)(Entry &)) {
    recursive_preorder(root, visit);
}
```

```
template <class Entry>
void Binary_tree<Entry> :: postorder(void (*visit)(Entry &)) {
    recursive_postorder(root, visit);
}
```

Giải thuật duyệt cây inorder

Algorithm recursive_inorder
Input: subroot là con trỏ node gốc và hàm *visit*
Output: kết quả phép duyệt

1. **if** (cây con không rỗng)
   1.1. **Call** recursive_inorder với nhánh trái của subroot
   1.2. Duyệt node subroot bằng hàm *visit*
   1.3. **Call** recursive_inorder với nhánh phải của subroot

End recursive_inorder

Mã C++ duyệt cây inorder

```
template <class Entry>
void Binary_tree<Entry> ::recursive_inorder
            (Binary_node<Entry> *sub_root, void (*visit)(Entry &)) {
    if (sub_root != NULL) {
        recursive_inorder(sub_root->left, visit);
        (*visit)(sub_root->data);
        recursive_inorder(sub_root->right, visit);
    }
}
```

Khai báo cây nhị phân

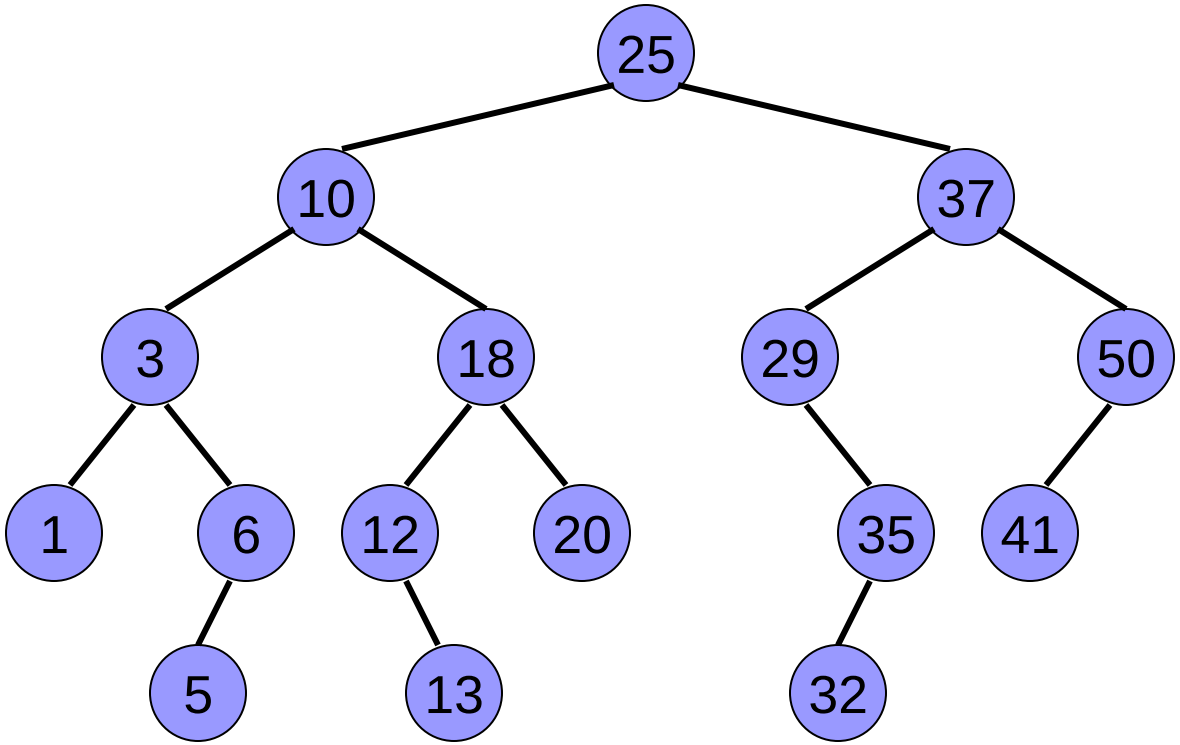
```
template <class Entry>
class Binary_tree {
public:
    Binary_tree( );
    bool empty( ) const;
    void preorder(void (*visit)(Entry &));
    void inorder(void (*visit)(Entry &));
    void postorder(void (*visit)(Entry &));
    int size( ) const;
    void clear( );
    int height( ) const;
    void insert(const Entry &);
    Binary_tree (const Binary_tree<Entry> &original);
    Binary_tree & operator = (const Binary_tree<Entry> &original);
    ~Binary_tree( );
protected:
    Binary_node<Entry> *root;
};
```

Cây nhị phân tìm kiếm – Binary search tree (BST)

- Một cây nhị phân tìm kiếm (BST) là một cây nhị phân rỗng hoặc mỗi node của cây này có các đặc tính sau:
  - 1. Khóa của con trái< khóa gốc< khóa con phải
  - 2. Các cây con (bên trái, phải) là BST
- Tính chất:
  - Chỉ cần đặc tính 1 là đủ
  - Duyệt inorder sẽ được danh sách có thứ tự

Ví dụ BST

Cho dãy: 25 ,10, 37, 3, 50, 18, 29, 41, 12, 35, 20, 32, 6, 1, 5, 13



Duyệt inorder:

1 3 5 6 10 12 13 18 20 25 29 32 35 37 41 50

Các tính chất khác của BST

- Node cực trái (hay phải):
  - Xuất phát từ node gốc
  - Đi sang trái (hay phải) đến khi không đi được nữa
- Khóa của node cực trái (hay phải) là nhỏ nhất (hay lớn nhất) trong BST
- BST là cây nhị phân có tính chất:
  - Khóa của node gốc lớn (hay nhỏ) hơn khóa của node cực trái (hay cực phải)

Thiết kế BST

```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
    //Viết lại phương thức chèn vào, loại bỏ để đảm bảo vẫn là BST
    Error_code insert(const Record &new_data);
    Error_code remove(const Record &old_data);

    //Thêm phương thức tìm kiếm dựa vào một khóa
    Error_code tree_search(Record &target) const;
private:
    // Add auxiliary function prototypes here.
};
```

Tìm kiếm trên BST

- Chọn hướng tìm theo tính chất của BST:
  - So sánh với node gốc, nếu đúng thì tìm thấy
  - Tìm bên nhánh trái (hay phải) nếu khóa cần tìm nhỏ hơn (hay lớn hơn) khóa của node gốc
- Giống phương pháp tìm kiếm nhị phân
- Thời gian tìm kiếm
  - Tốt nhất và trung bình: $O(\lg_2 n)$
  - Tệ nhất: $O(n)$

Giải thuật tìm kiếm trên BST

Algorithm BST_search
Input: subroot là node gốc và target là khóa cần tìm
Output: node tìm thấy

1. **if** (cây rỗng)
   1.1. **return** not_found
2. **if** (target trùng khóa với subroot)
   2.1. **return** subroot
3. **if** (target có khóa nhỏ hơn khóa của subroot)
   3.1. Tìm bên nhánh trái của subroot
4. **else**
   4.1. Tìm bên nhánh phải của subroot

End BST_search

Mã C++ tìm kiếm trên BST

```
template <class Record>
Binary_node<Record> *Search_tree<Record> :: search_for_node
          (Binary_node<Record>* sub_root, const Record &target) const {

   if (sub_root == NULL || sub_root->data == target)
      return sub_root;
   else if (sub_root->data < target)
      return search_for_node(sub_root->right, target);
   else return search_for_node(sub_root->left, target);
}
```

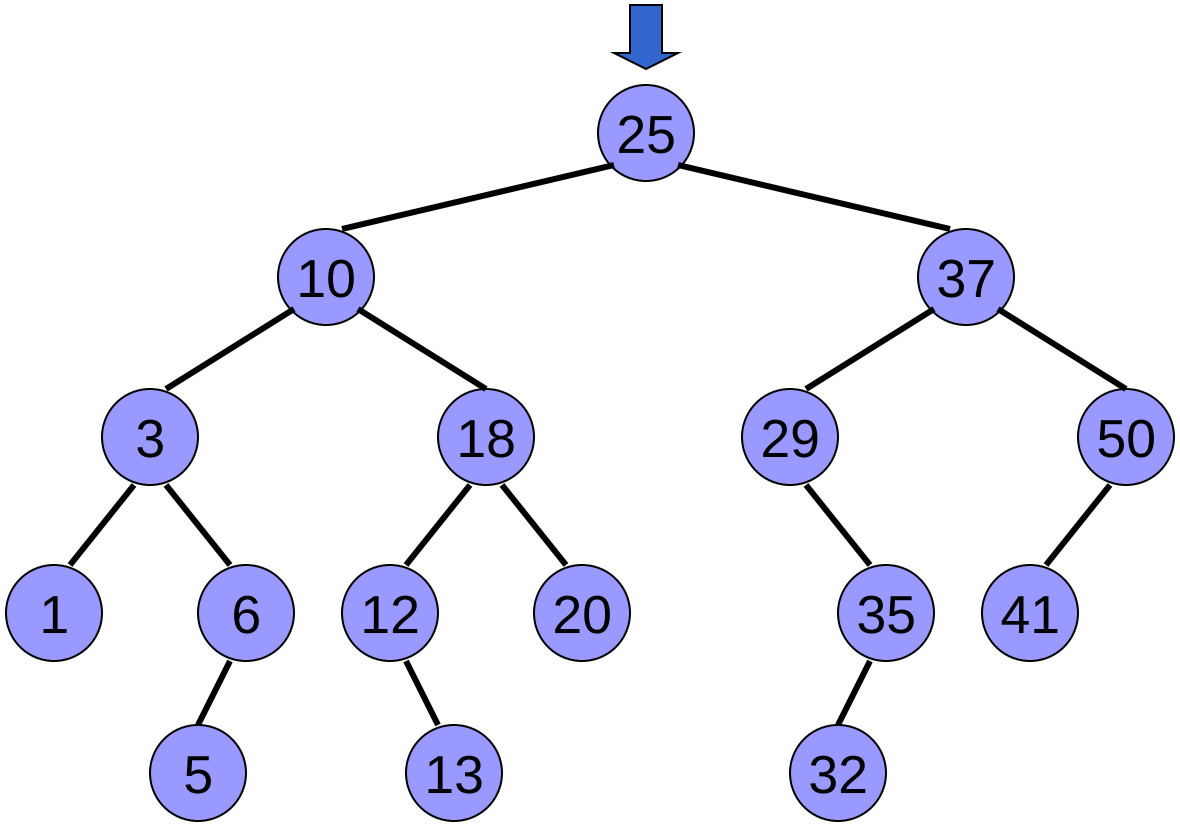
Mã C++ tìm kiếm trên BST(không đệ qui)

```
template <class Record>
Binary_node<Record> *Search_tree<Record> :: search_for_node
          (Binary_node<Record>* sub_root, const Record &target) const {

   while (sub_root != NULL && sub_root->data != target)
      if (sub_root->data < target) sub_root = sub_root->right;
      else sub_root = sub_root->left;
   return sub_root;
}
```

Phương thức tree_search

```
template <class Record>
Error_code Search_tree<Record> :: tree_search(Record &target) const {

    Error_code result = success;
    Binary_node<Record> *found = search_for_node(root, target);
    if (found == NULL)
        result = not_present;
    else
        target = found->data;
    return result;
}
```

Ví dụ tìm kiếm trên BST

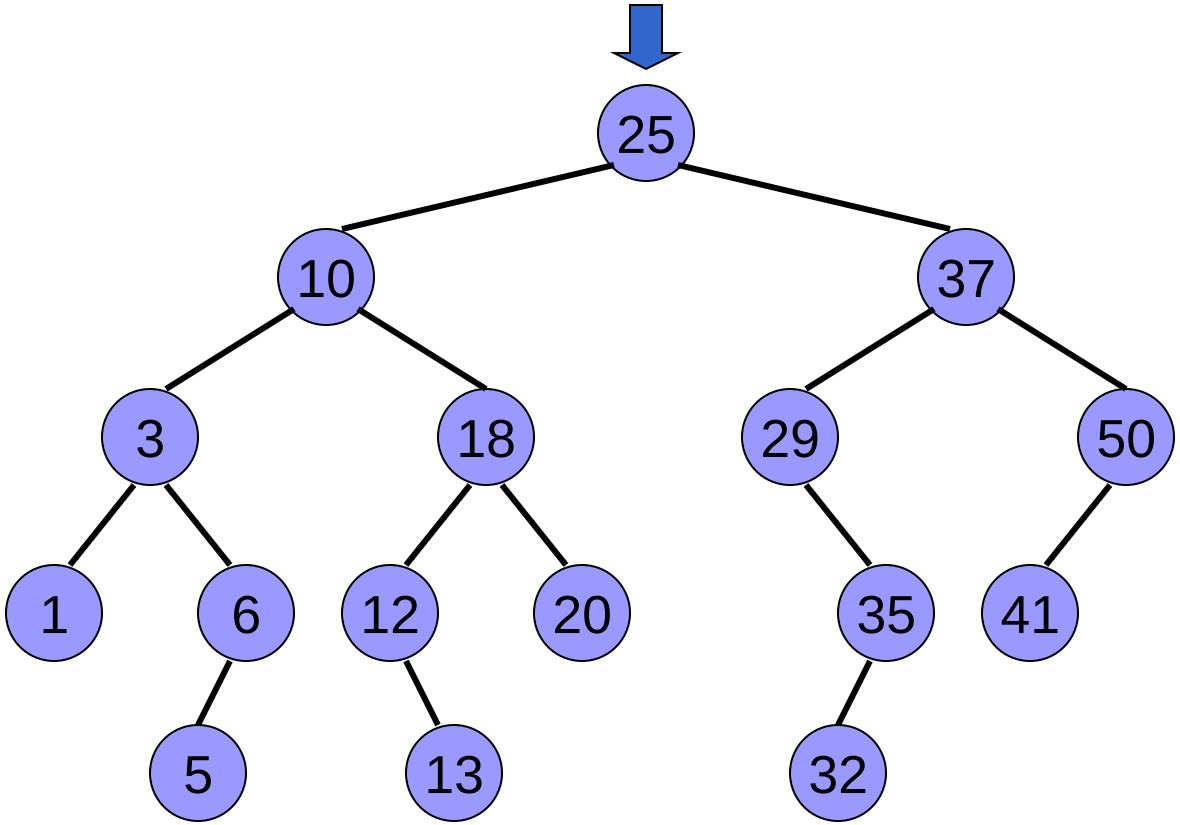


Tìm kiếm 13

Tìm thấy

Số node duyệt: 5
Số lần so sánh: 9

Ví dụ tìm kiếm trên BST



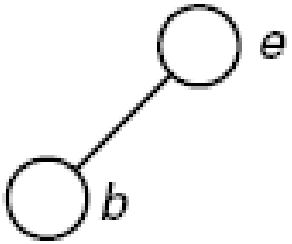
Khóa gốc nhỏ hơn

Tìm kiếm 14 Không tìm thấy

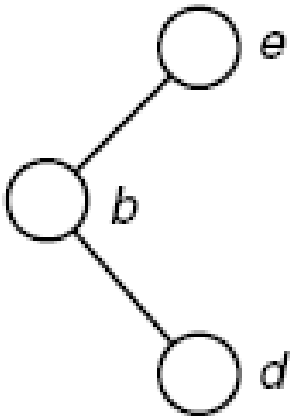
Số node duyệt: 5
Số lần so sánh: 10

Thêm vào BST

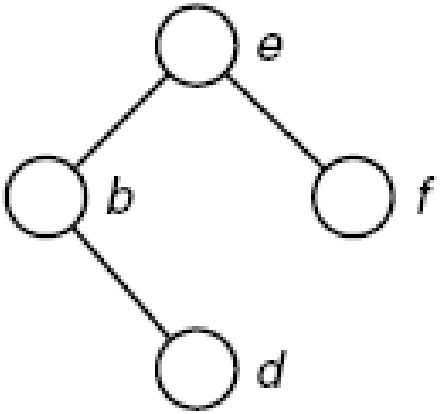
(a) Insert *e*



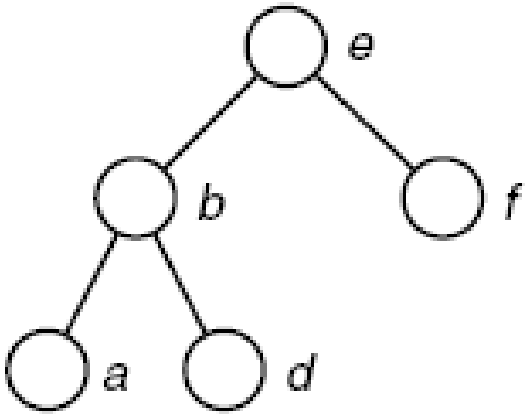
(b) Insert *b*



(c) Insert *d*



(d) Insert *f*



(e) Insert *a*

Giải thuật thêm vào BST

Algorithm BST_insert
Input: subroot là node gốc và new_data là dữ liệu cần thêm vào
Output: BST sau khi thêm vào

1. **if** (cây rỗng)
   1.1. Thêm vào tại vị trí này
2. **if** (target trùng khóa với subroot)
   2.1. **return** duplicate_error
3. **if** (new_data có khóa nhỏ hơn khóa của subroot)
   3.1. Thêm vào bên nhánh trái của subroot
4. **else**
   4.1. Thêm vào bên nhánh phải của subroot

End BST_insert

Mã C++ thêm vào BST

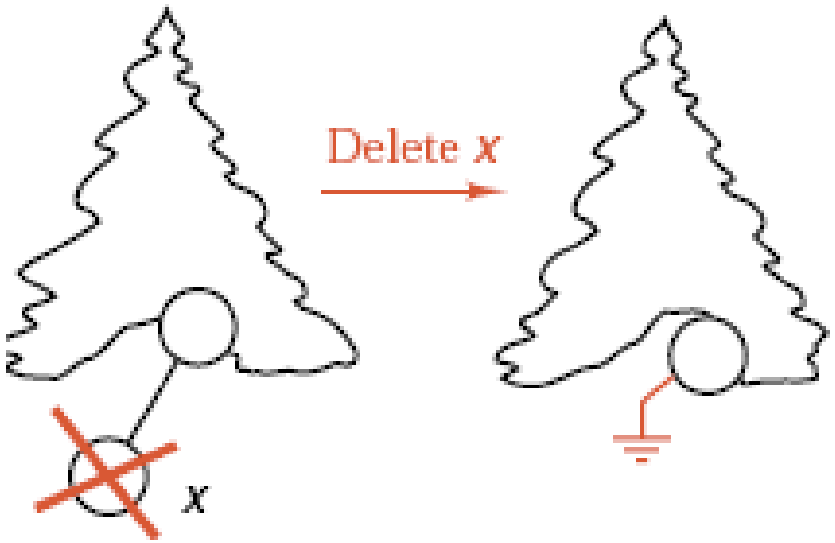
```
template <class Record>
Error_code Search_tree<Record> :: search_and_insert
          (Binary_node<Record> * &sub_root, const Record &new_data) {

   if (sub_root == NULL) {
      sub_root = new Binary_node<Record>(new_data);
      return success;
   }
   else if (new_data < sub_root->data)
      return search_and_insert(sub_root->left, new_data);
   else if (new_data > sub_root->data)
      return search_and_insert(sub_root->right, new_data);
   else return duplicate_error;
}
```

Giải thuật thêm vào BST (không đệ qui)

Algorithm BST_insert
Input: subroot là node gốc và new_data là dữ liệu cần thêm vào
Output: BST sau khi thêm vào
1. parent là rỗng và left_or_right là “left”
2. **while** (subroot không rỗng)
   2.1. **if** (target trùng khóa với subroot)
      2.1.1. **return** duplicate_error
   2.2. **if** (new_data có khóa nhỏ hơn khóa của subroot)
      2.2.2. parent là subroot và left_or_right là “left”
      2.2.1. Chuyển subroot sang nhánh trái của subroot
   2.3. **else**
      2.3.2. parent là subroot và left_or_right là “right”
      2.3.1. Chuyển subroot sang nhánh phải của subroot
3. **if** (subroot là rỗng) *//Thêm vào tại vị trí này*
      3.1. **if** (parent là rỗng)
         3.1.1. Tạo node gốc của cây
      3.2. **else**
         3.2.1. Tạo node bên trái hay phải parent tùy theo left_or_right

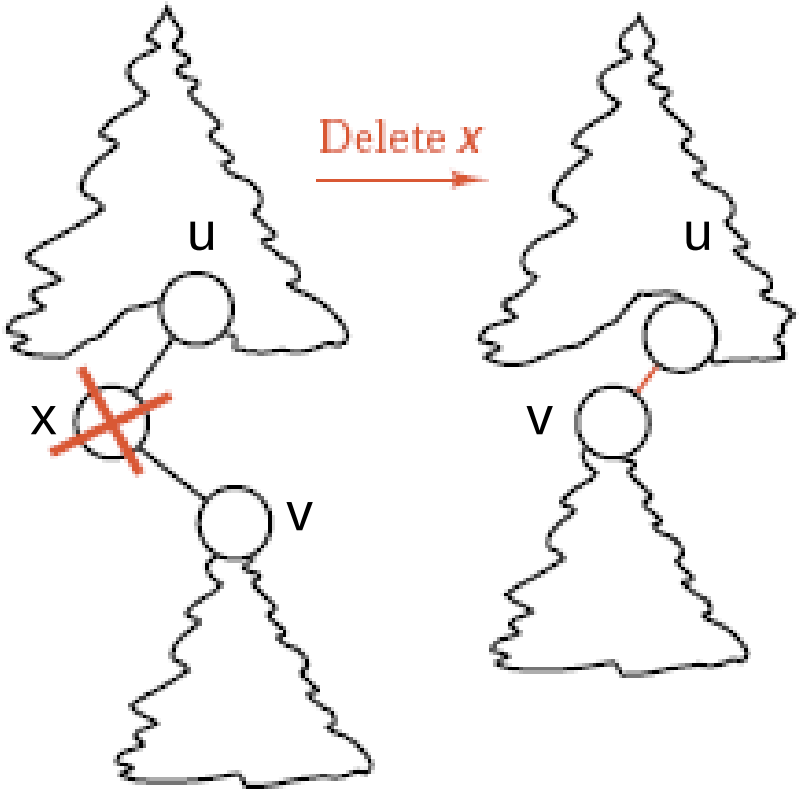
End BST_insert

Xóa một node lá khỏi BST



1. Xóa node này
2. Gán liên kết từ cha của nó thành rỗng

Xóa một node chỉ có một con



1. Gán liên kết từ cha của nó xuống con duy nhất của nó
2. Xóa node này

A. Đường dẫn đến các node của cây con v có dạng:

… u x v …

B. Không còn node nào trong cây có đường dẫn có dạng như vậy.

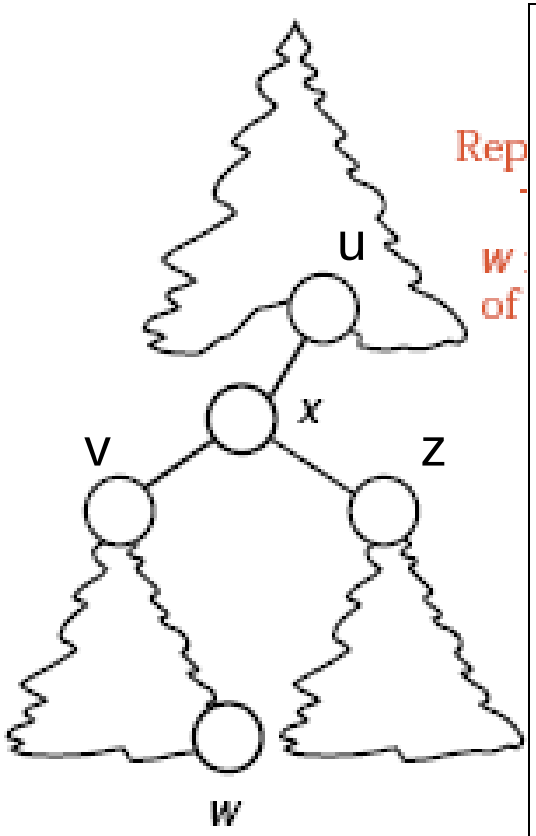
C. Sau khi xóa node x, đường dẫn đến các node của cây con v có dạng:

… u v …

D. Đường dẫn của các node khác trong cây không đổi.

E. Trước đó, các node của cây con v nằm trong nhánh con của x là bên trái (bên phải) của u và bây giờ cũng nằm bên trái (bên phải) của u nên vẫn thỏa mãn BST

Xóa một node có đủ 2 nhánh con



A. Đường dẫn đến các node của cây con v và z có dạng:

… u x v …

… u x z …

B. Nếu xóa node x thì đường dẫn đến các node của cây con v và z có dạng:

… u v …

… u z …

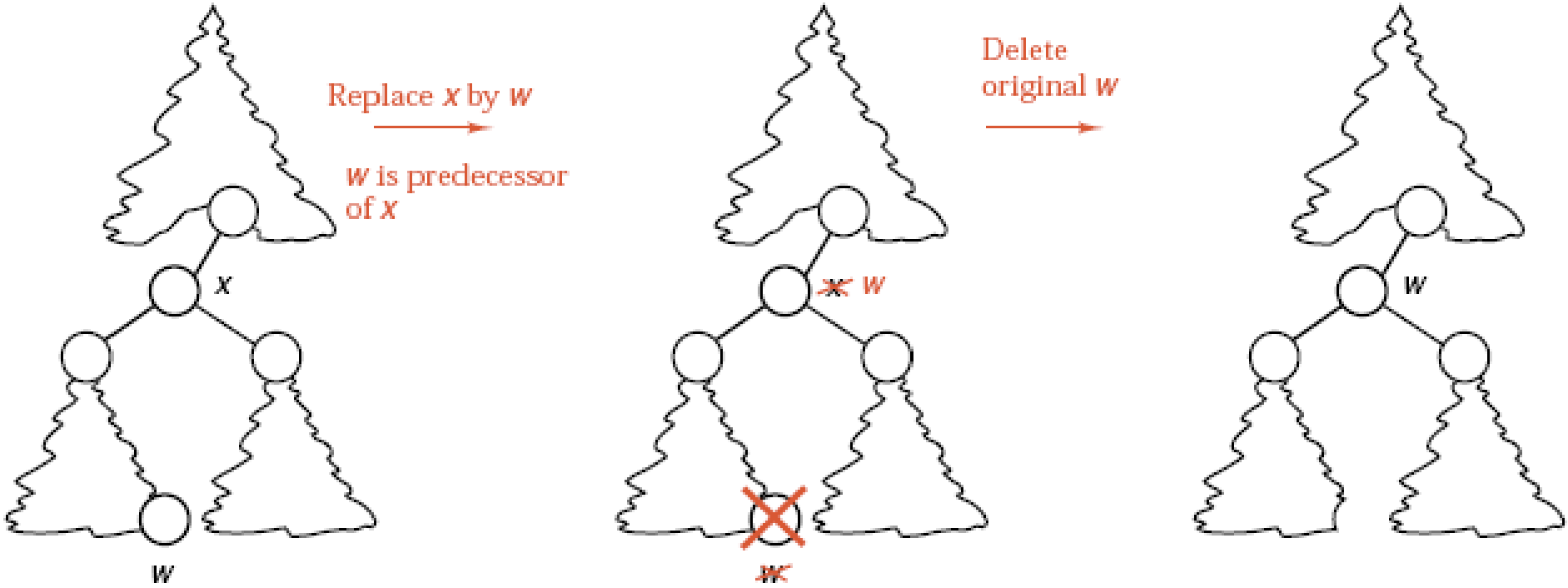
D. Điều này chỉ xảy ra khi cây con u và v nằm về 2 phía của u => không còn là BST.

E. Giải pháp là thay giá trị x bằng giá trị w thuộc cây này sao cho:

w lớn hơn tất cả khóa của các node của cây con v

w nhỏ hơn tất cả khóa của các node của cây con z

Xóa một node có đủ 2 nhánh con (tt.)



1. Tìm w là node trước node x trên phép duyệt cây inorder (chính là node cực phải của cây con bên trái của x)
2. Thay x bằng w
3. Xóa node w cũ (giống trường hợp 1 hoặc 2 đã xét)

Giải thuật xóa một node

Algorithm BST_remove_root
Input: subroot là node gốc cần phải xóa
Output: BST sau khi xóa xong subroot

1. **if** (trường hợp 1 hoặc 2) *//subroot là node lá hoặc có một con*
   1.1. Gán liên kết cha đến rỗng hoặc nhánh con duy nhất của subroot
   1.2. Xóa subroot
2. **else** *//trường hợp 3: có 2 nhánh con*
   //Tìm node cực phải của cây con trái
   2.1. to_delete là node con trái của subroot
   2.2. **while** (nhánh phải của to_delete không rỗng)
      2.2.1. di chuyển to_delete sang node con phải
   2.2. Thay dữ liệu của subroot bằng dữ liệu của to_delete
   2.4. **Call** BST_remove_root với to_delete

End BST_remove_root

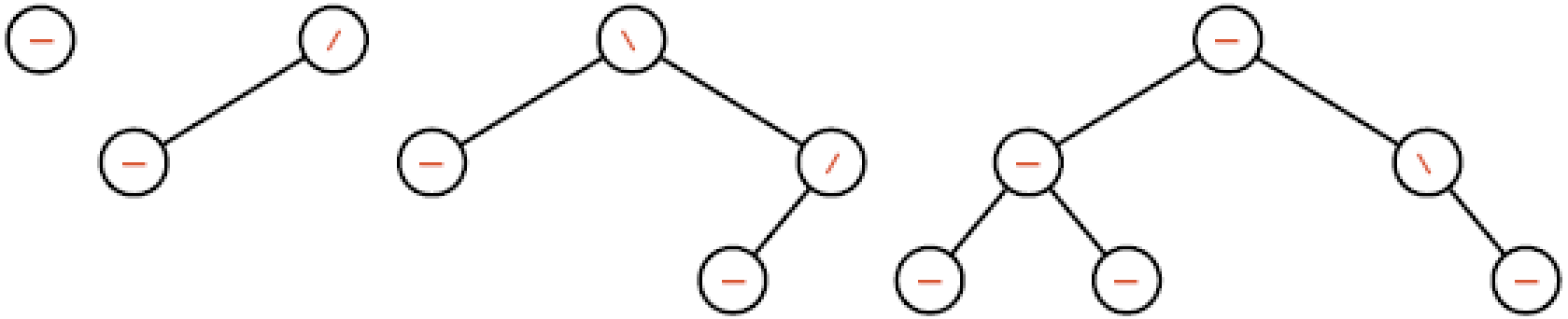
Mã C++ xóa một node

```
template <class Record>
Error_code Search_tree<Record> :: remove_root
            (Binary_node<Record> * &sub_root) {
   if (sub_root == NULL) return not_present;
   Binary_node<Record> *to_delete = sub_root;
   if (sub_root->right == NULL) sub_root = sub_root->left;
   else if (sub_root->left == NULL) sub_root = sub_root->right;
   else { to_delete = sub_root->left;
      Binary_node<Record> *parent = sub_root;
      while (to_delete->right != NULL) {
         parent = to_delete;
         to_delete = to_delete->right; }
      sub_root->data = to_delete->data;
      if (parent == sub_root) sub_root->left = to_delete->left;
      else parent->right = to_delete->left; }
   delete to_delete;
   return success;  }
```

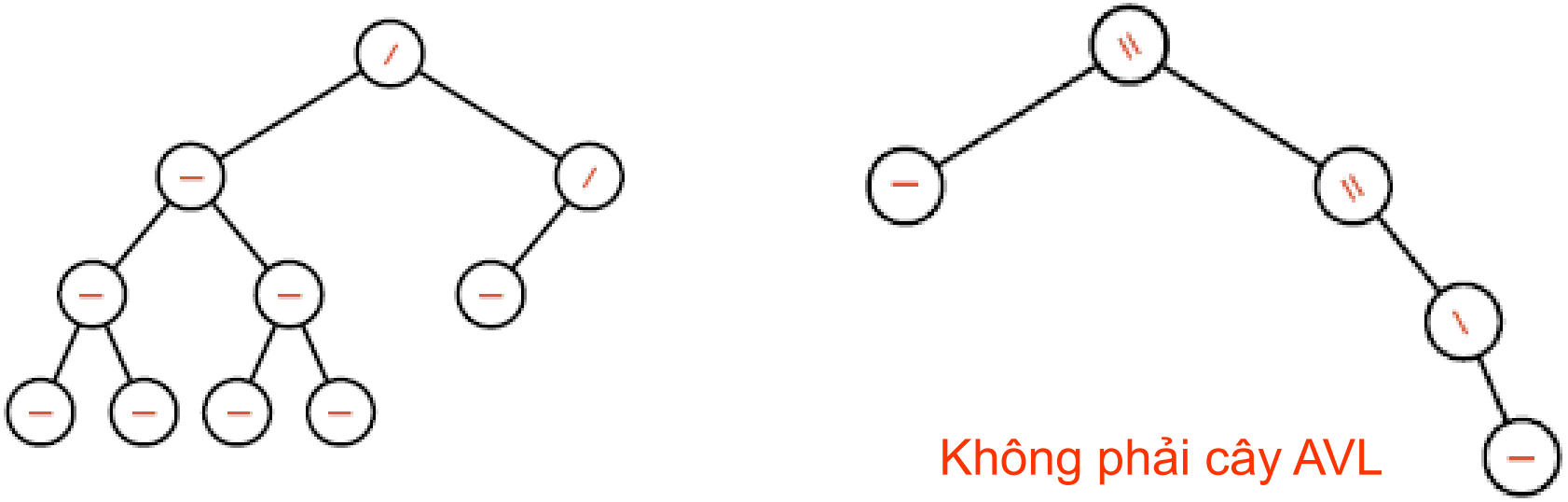
Cây cân bằng chiều cao - AVL

- Cây cân bằng hoàn toàn:
  - Số node của nhánh trái và nhánh phải chênh nhau không quá 1.
- ĐN cây AVL:
  - BST
  - Tại node bất kỳ, chiều cao nhánh trái và nhánh phải chênh nhau không quá 1.
- Ký hiệu cho mỗi node của cây AVL:
  - Node cân bằng: '-'
  - Nhánh trái cao hơn: '/'
  - Nhánh phải cao hơn: '\'

Ví dụ cây AVL



Cây AVL



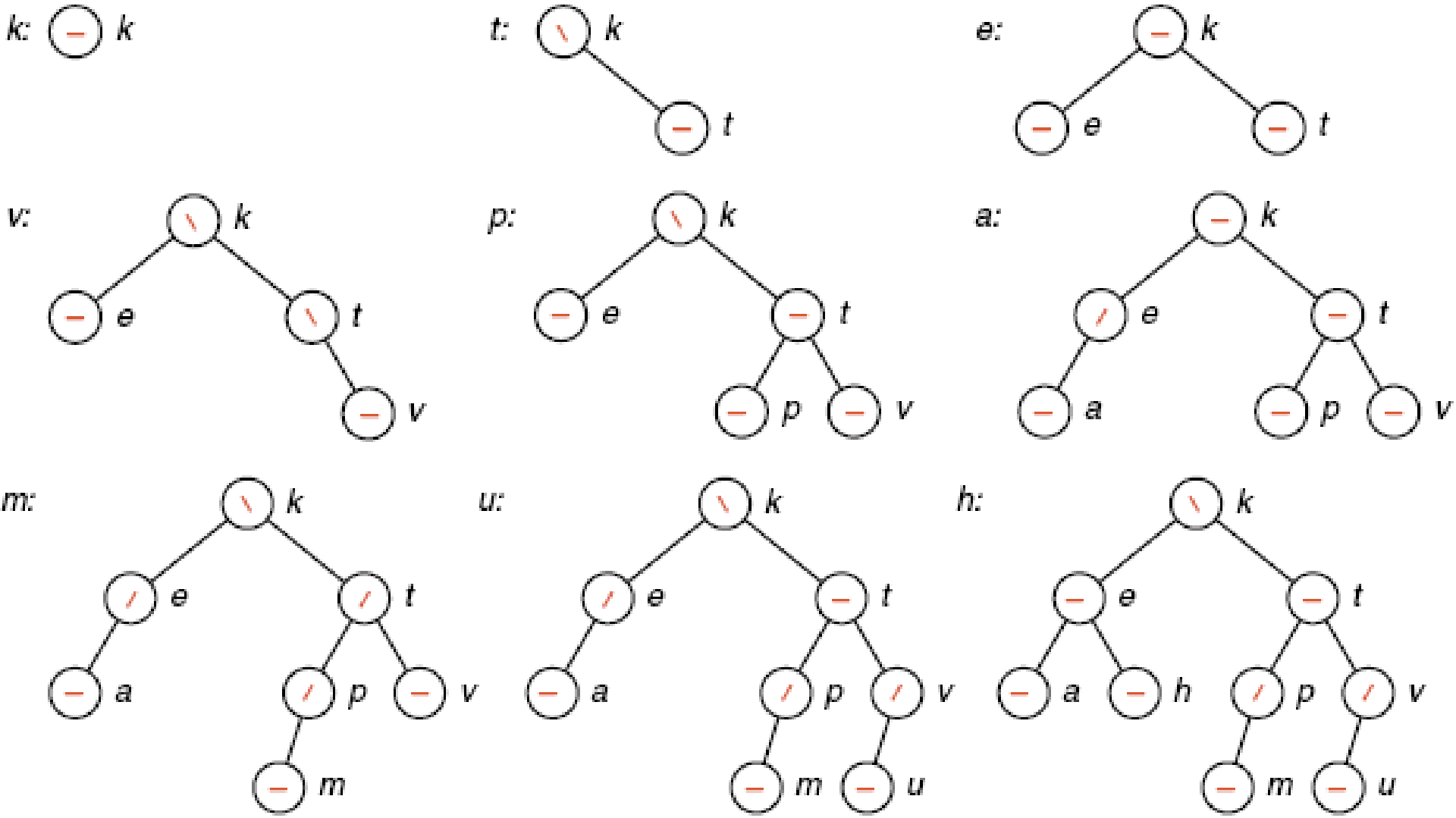
Không phải cây AVL

Khai báo cây AVL

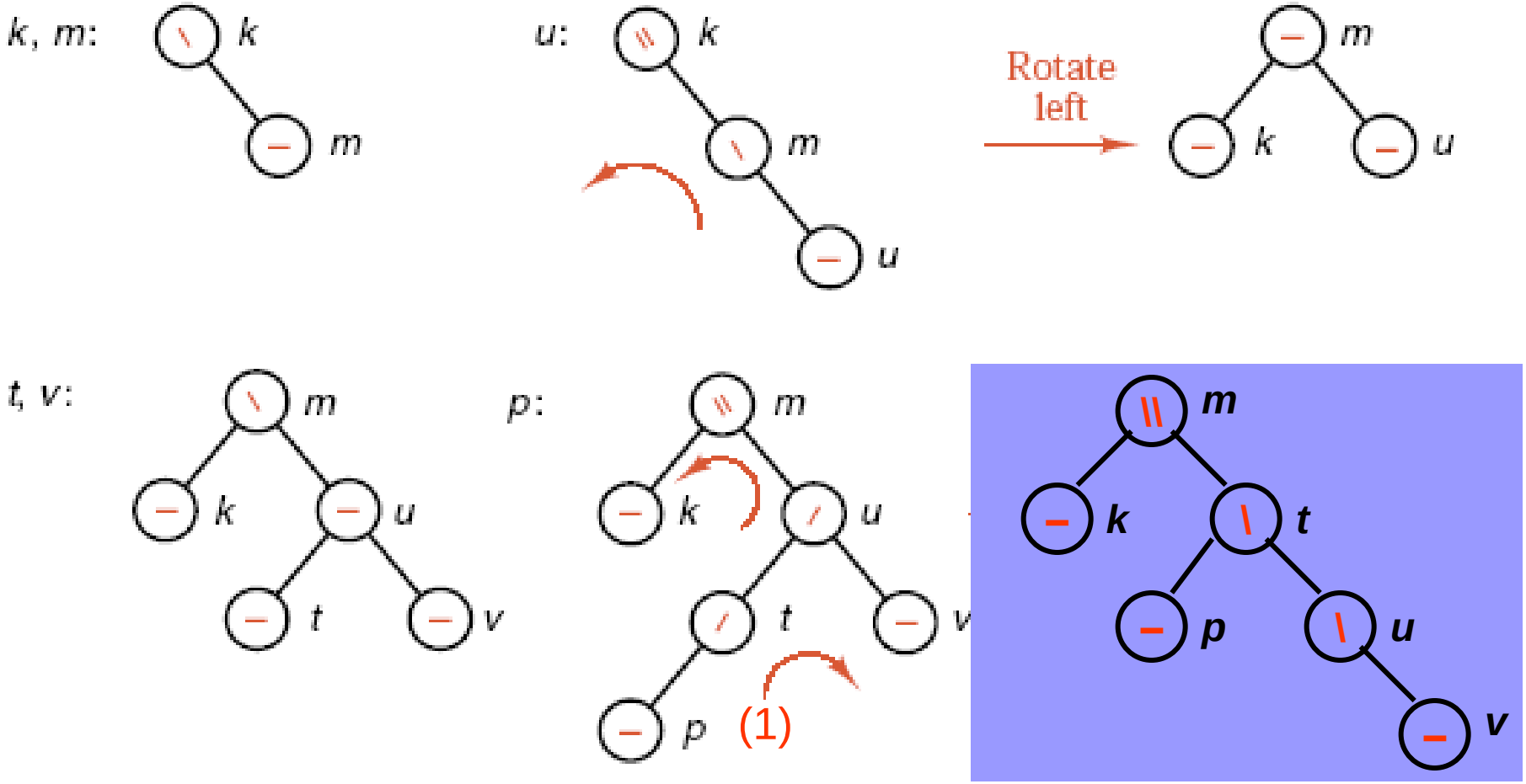
```
enum Balance_factor { left_higher, equal_height, right_higher };
template <class Record>
struct AVL_node: public Binary_node<Record> {
    // additional data member:
    Balance_factor balance;
    AVL_node( );
    AVL_node(const Record &x);
    void set_balance(Balance_factor b);
    Balance_factor get_balance( ) const;
};
```

```
template <class Record>
class AVL_tree: public Search_tree<Record> {
public:
    Error_code insert(const Record &new data);
    Error_code remove(const Record &old data);
private:
    // Add auxiliary function prototypes here.
};
```

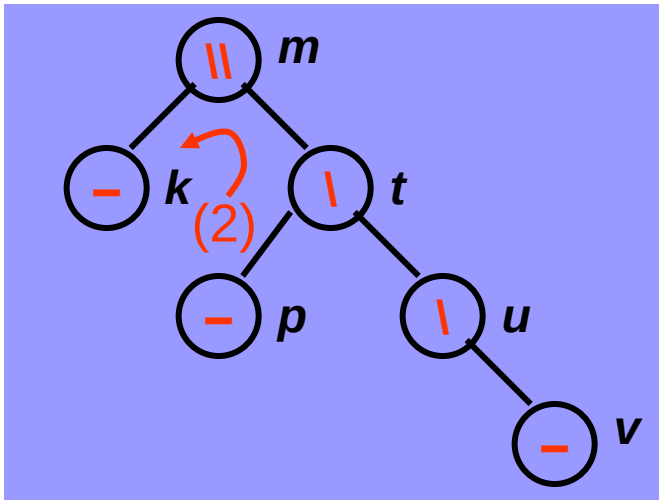
Ví dụ 1 thêm vào cây AVL

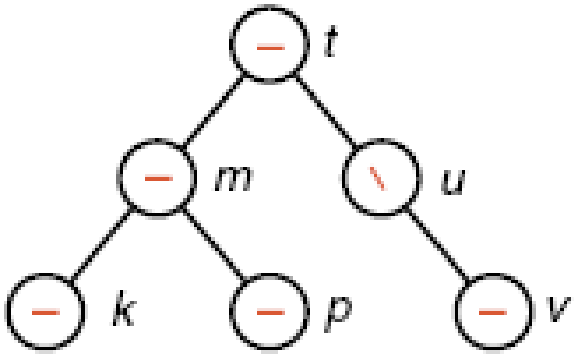


Ví dụ 2 thêm vào cây AVL



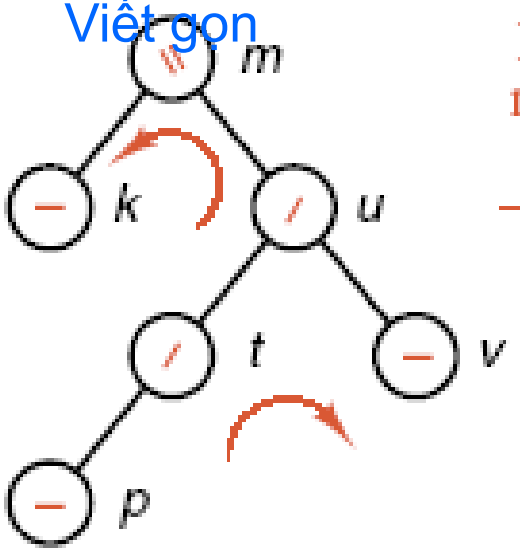
Ví dụ 2 thêm vào cây AVL (tt.)



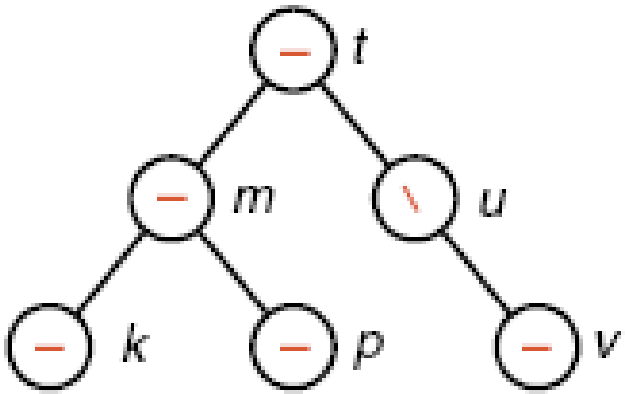


Viết gọn

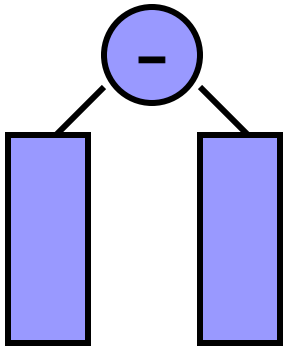
p:



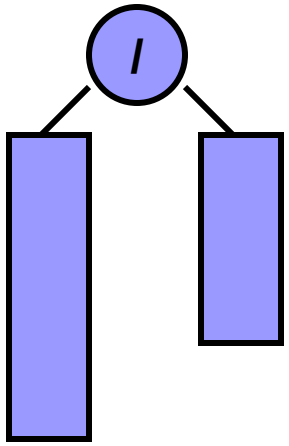
Double rotation left



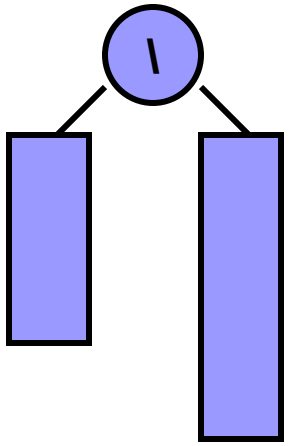
Các trạng thái khi thêm vào



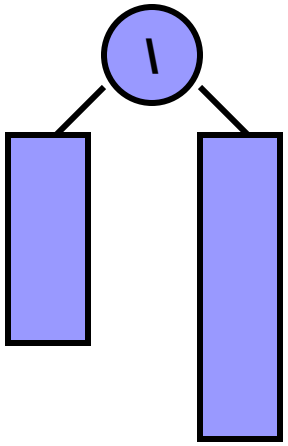
Thêm vào bên phải và làm bên phải cao lên



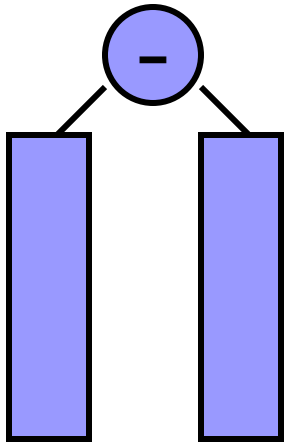
Thêm vào bên phải và làm bên phải cao lên



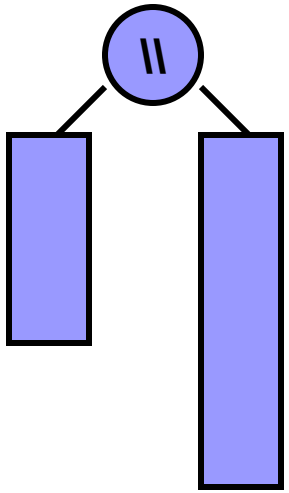
Thêm vào bên phải và làm bên phải cao lên



Chiều cao cây tăng

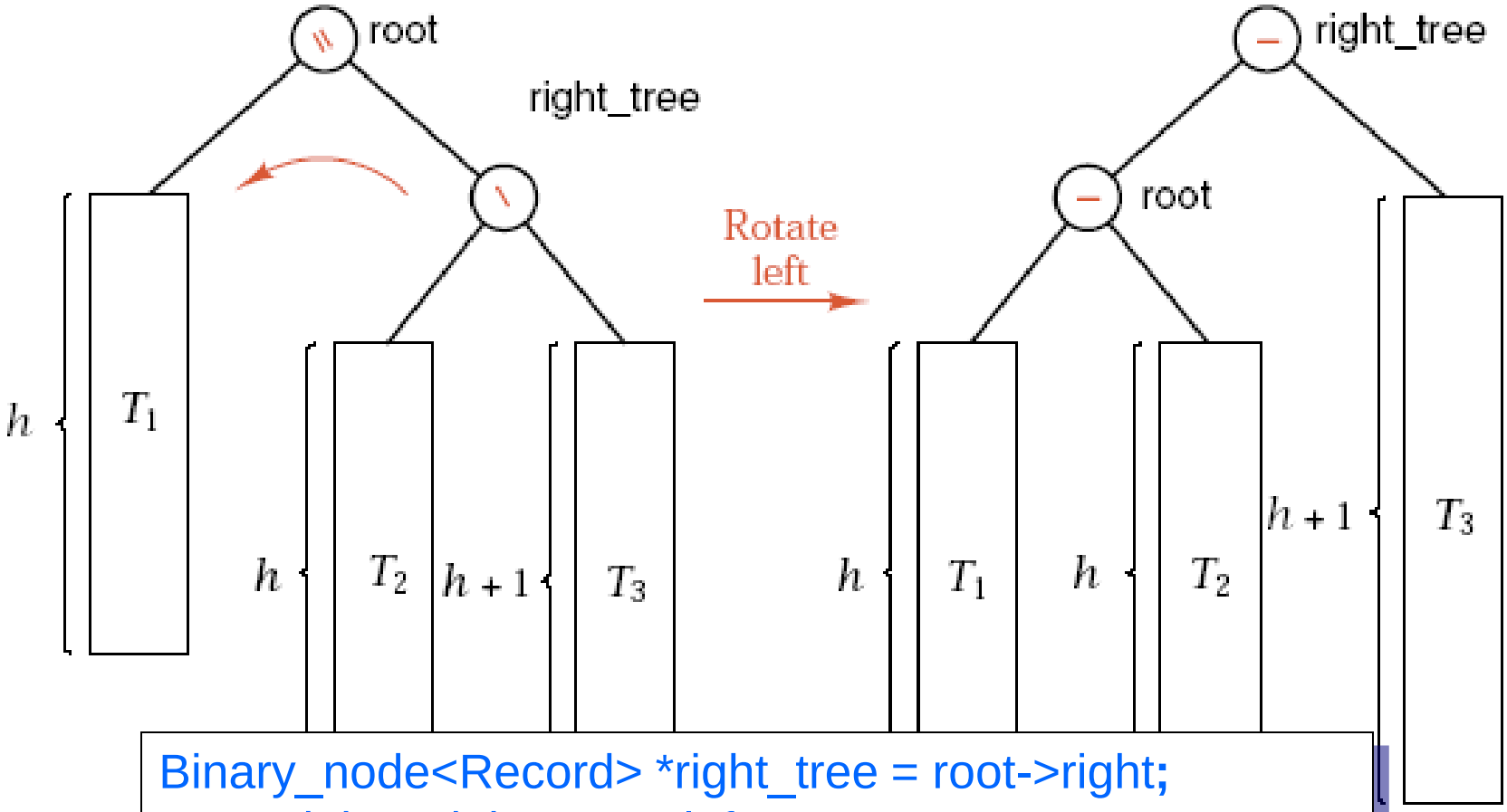


Chiều cao cây không đổi



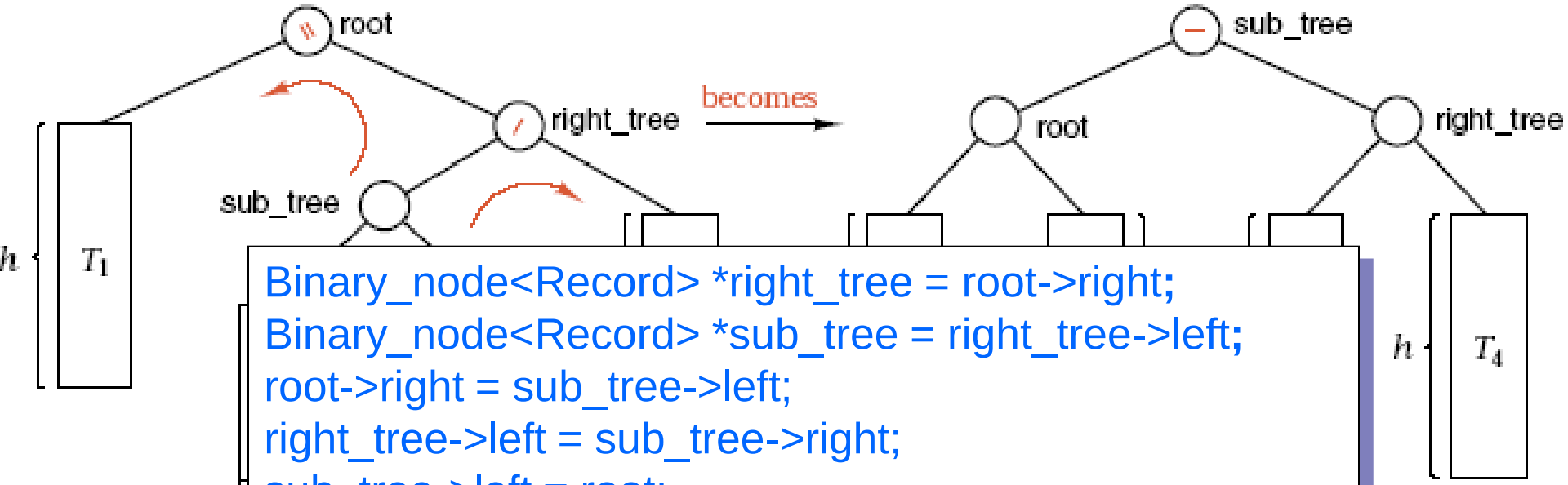
Mất cân bằng bên phải

Cân bằng cây AVL – Quay đơn



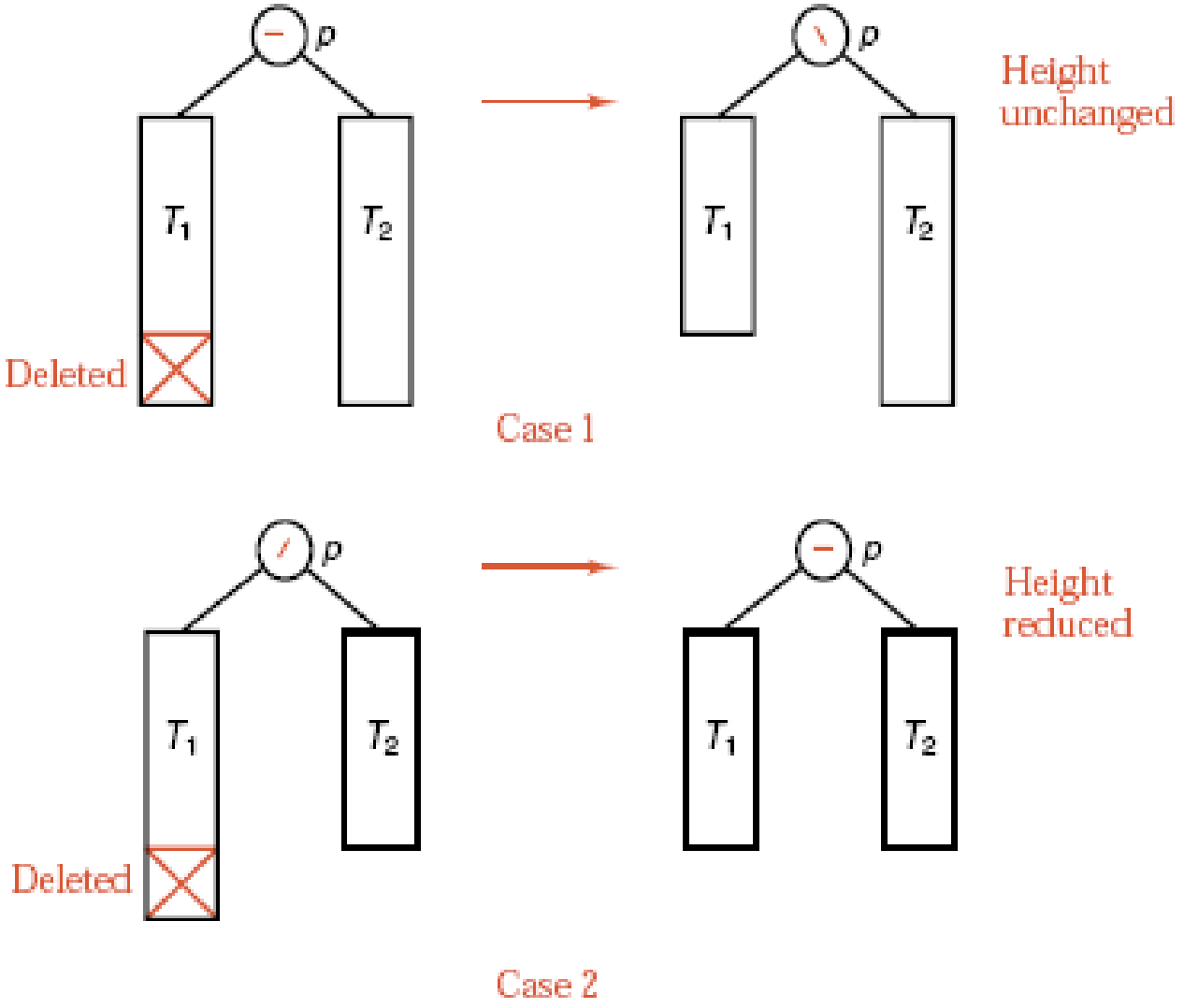

```
Binary_node<Record> *right_tree = root->right;
root->right = right_tree->left;
right_tree->left = root;
root = right_tree;
```

Cân bằng cây AVL – Quay kép

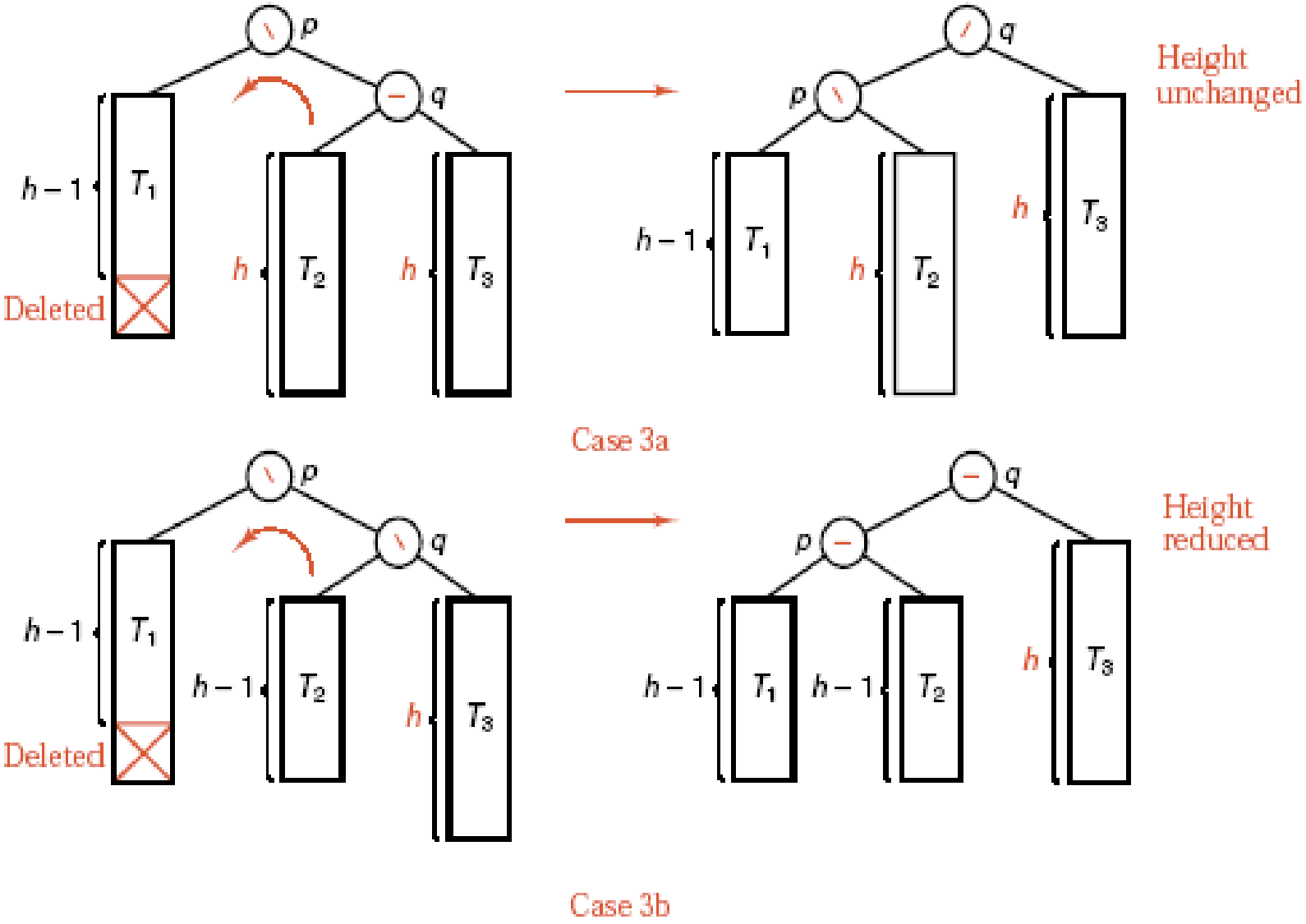



```
Binary_node<Record> *right_tree = root->right;
Binary_node<Record> *sub_tree = right_tree->left;
root->right = sub_tree->left;
right_tree->left = sub_tree->right;
sub_tree->left = root;
sub_tree->right = right_tree;
root = sub_tree;

Hoặc:

rotate_right(right_tree);
rotate_left(root);
```

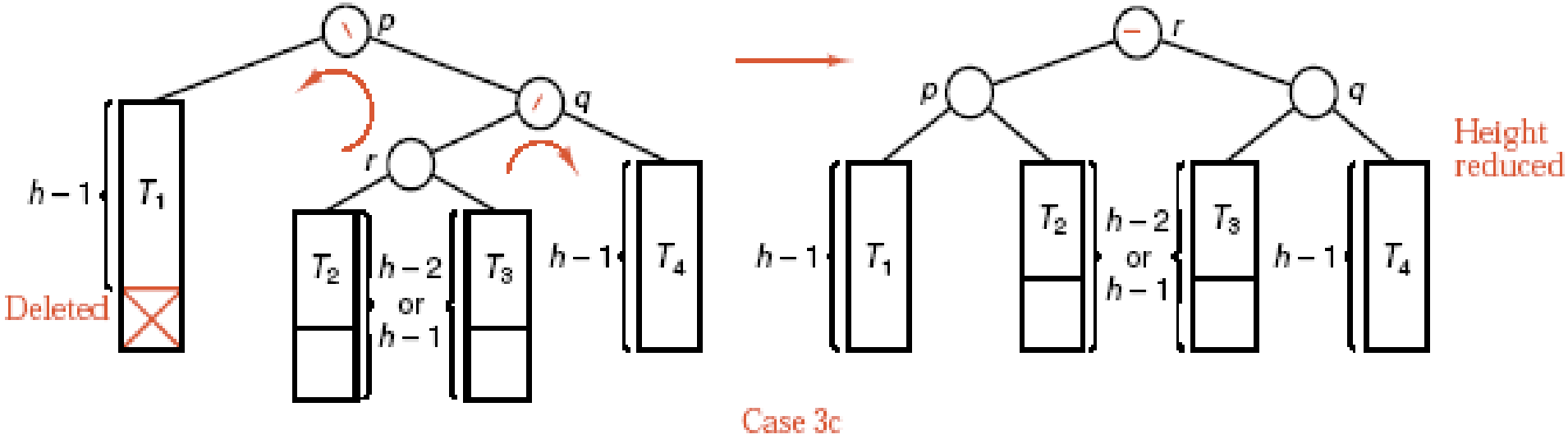
Các trạng thái khi xóa node



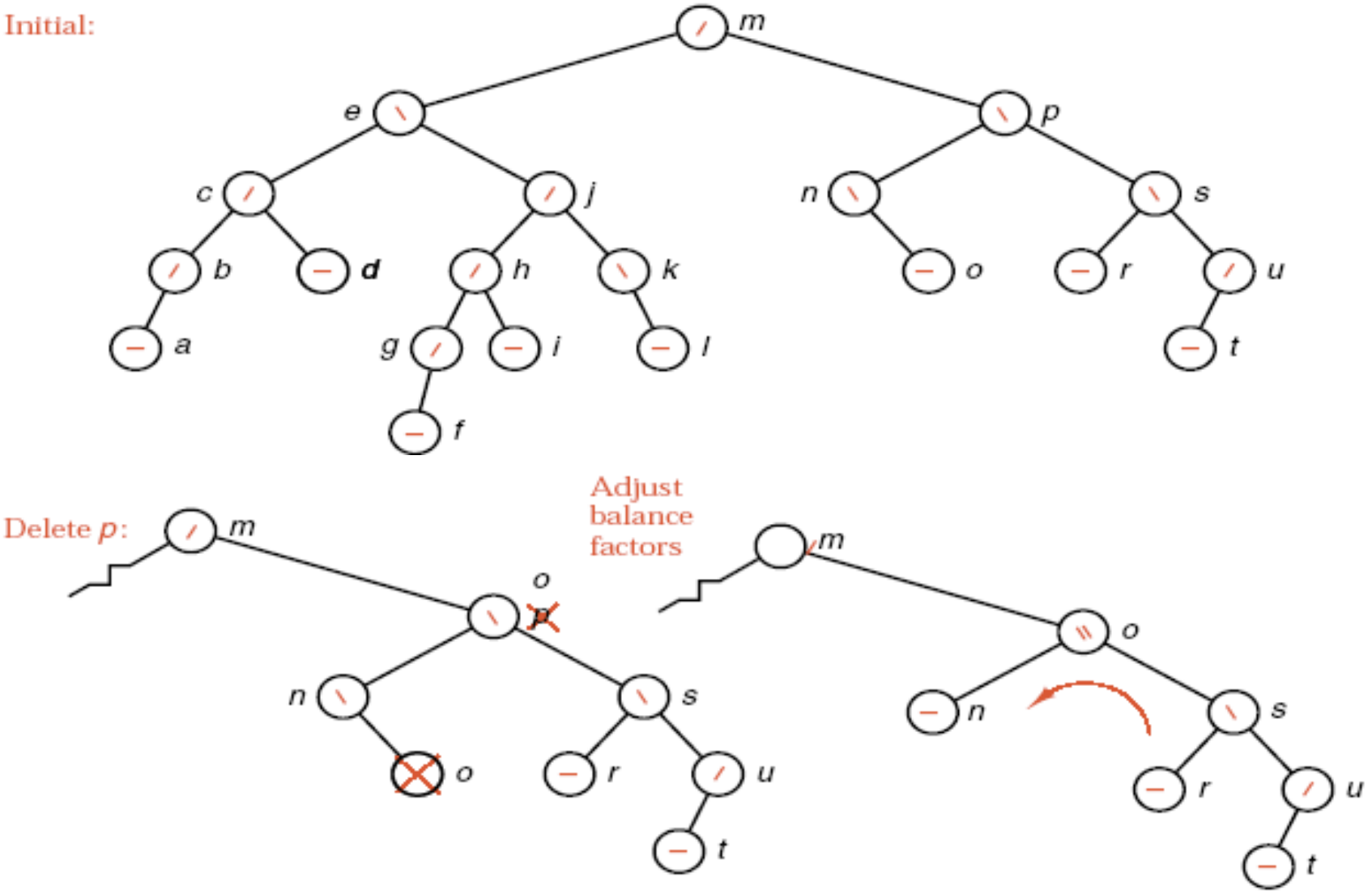
Các trạng thái khi xóa node (tt.)



Các trạng thái khi xóa node (tt.)



Ví dụ xóa node của cây AVL



Ví dụ xóa node của cây AVL (tt.)