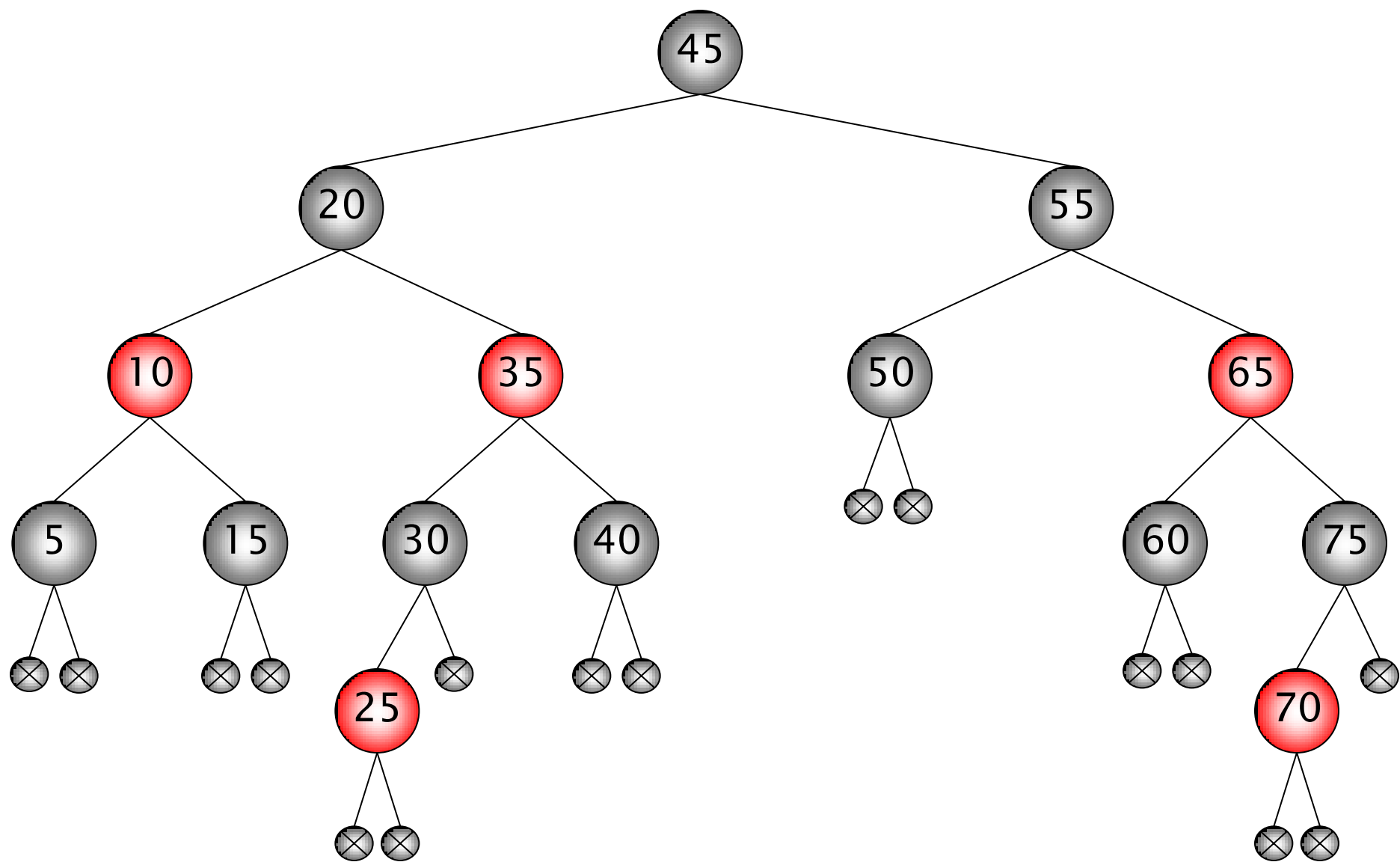
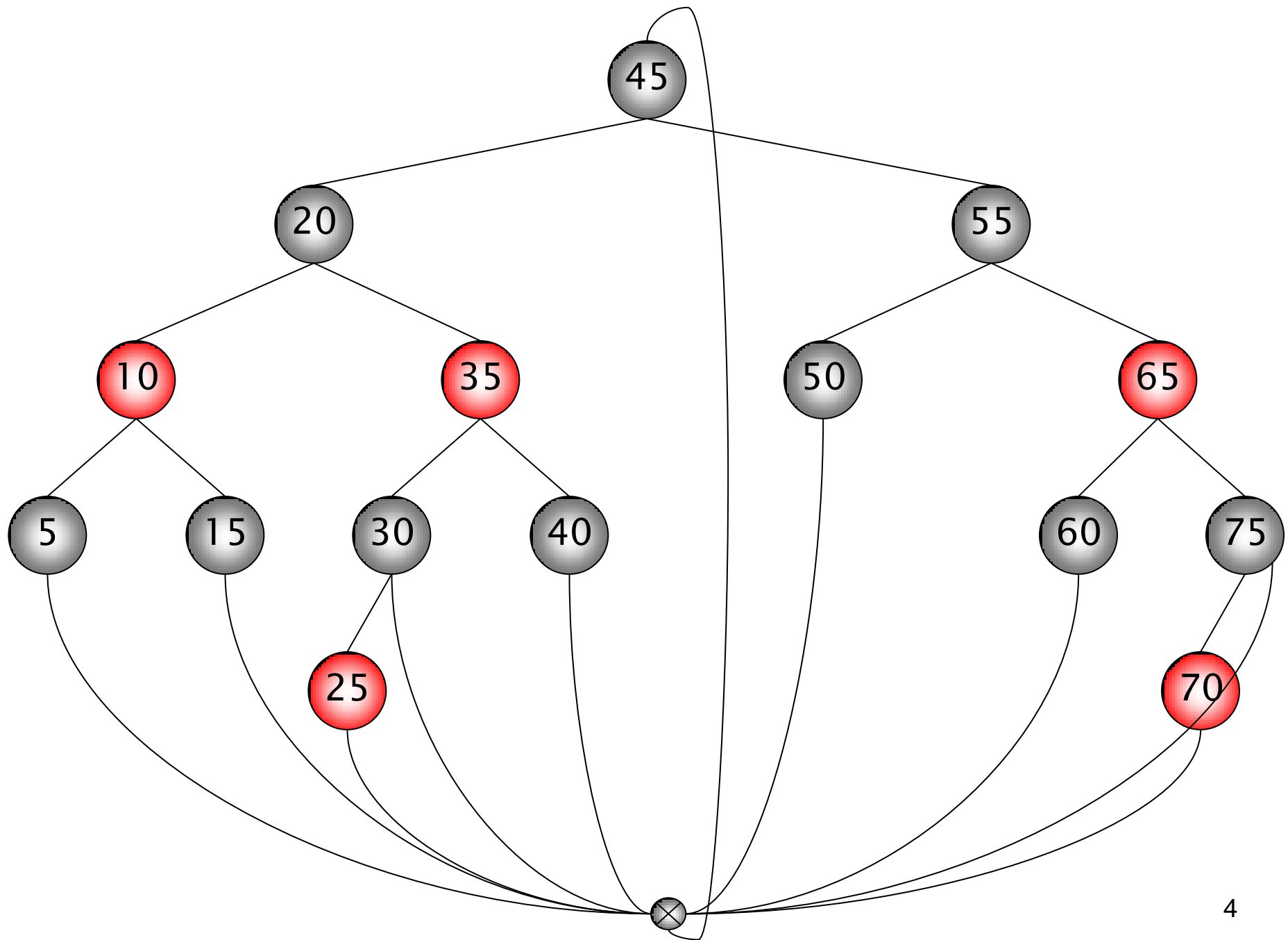


Cây Đỏ–Đen

Cây Đỏ–Đen

- Cây Đỏ–Đen thuộc nhóm cấu trúc cây tìm kiếm, duy trì tính “cân bằng” để đảm bảo các thao tác: thêm, hủy, tìm kiếm có chi phí lớn nhất là $O(\log n)$
- Cây AVL (đệ qui): chiều duyệt xuống dùng để tìm và thêm/hủy phần tử, chiều duyệt lên để cập nhật tính cân bằng
- Cây Đỏ–Đen: quá trình lặp thay cho đệ qui khiến việc thực thi đơn giản và hiệu quả hơn





Định nghĩa: Cây Đỏ-Đen là cây nhị phân tìm kiếm thỏa các tiêu chuẩn sau:

- Mỗi nút hoặc là nút đỏ, hoặc là nút đen
- Nút gốc luôn luôn là nút đen
- Nếu một nút là đỏ thì nút con của nó phải là đen
- Mỗi đường đi từ một nút bất kỳ đến các nút lá của cây đều có cùng số lượng các nút *đen* (chiều cao đen – *black height*)

Nhận xét:

- Cây Đỏ–Đen với n nút trong có chiều cao tối đa là $2\log_2(n + 1)$
- Các nút lá **NIL** (nút ngoài) luôn là nút đen
- Một cạnh dẫn đến một nút đen gọi là cạnh đen (*black edge*)
- Nếu không tồn tại nút cha hay nút con của một nút thì những con trỏ tương ứng sẽ trỏ về **NIL**
 - Một lính canh (**Sent**) sẽ đại diện cho tất cả các nút **NIL**

```
typedef struct Node * ref;
struct Node {
    int key;
    int color;
    ref parent;
    ref left;
    ref right;
}
```

```
ref getNode(int key, int color, ref nil) {
    p = new Node;
    p->key = key;
    p->color = color;
    p->left = p->right = p->parent = nil;
    return p;
}
```

Trạng thái ban đầu

```
ref nil, root;
```

```
...
```

```
nil = new Node;
```

```
nil->color = BLACK;
```

```
nil->left =
```

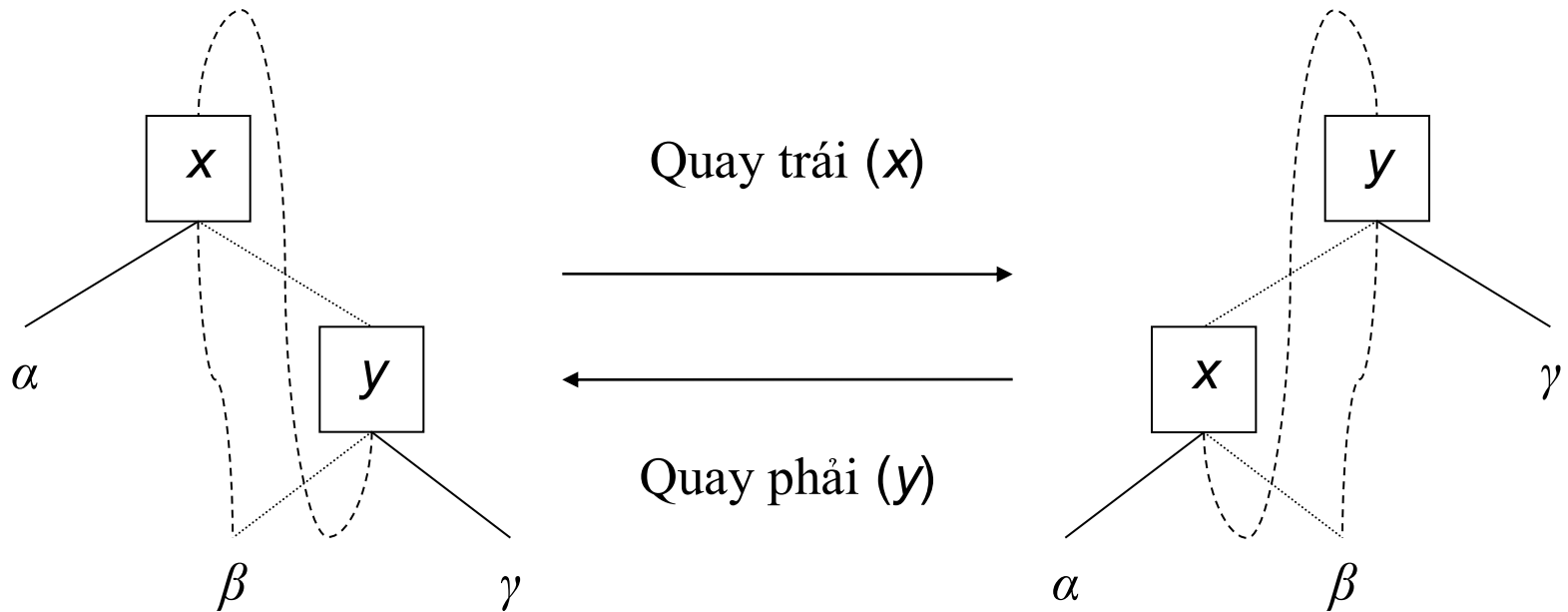
```
nil->right =
```

```
nil->parent = nil;
```

```
...
```

```
root = nil;
```


Thao tác quay



```

void    leftRotate(ref &root, ref x) {
    y    = x->right;
    x->right = y->left;

    if (y->left != nil) y->left->parent = x;
    y->parent = x->parent;

    if (x->parent == nil) root = y;
    else
        if (x == x->parent->left)
            x->parent->left = y;
        else
            x->parent->right = y;
    y->left = x;
    x->parent = y;
}

```

Thao tác chèn (Insertion)

Phần tử chèn vào luôn luôn có màu đỏ

Giải thuật:

Bước 1: Chèn phần tử (tương tự cây nptk)

Bước 2: Cập nhật các trường thông tin cho nút mới

Bước 3: Để duy trì tính “cân bằng”, nếu cần, phép tô màu nút và/hoặc phép quay sẽ được thực hiện

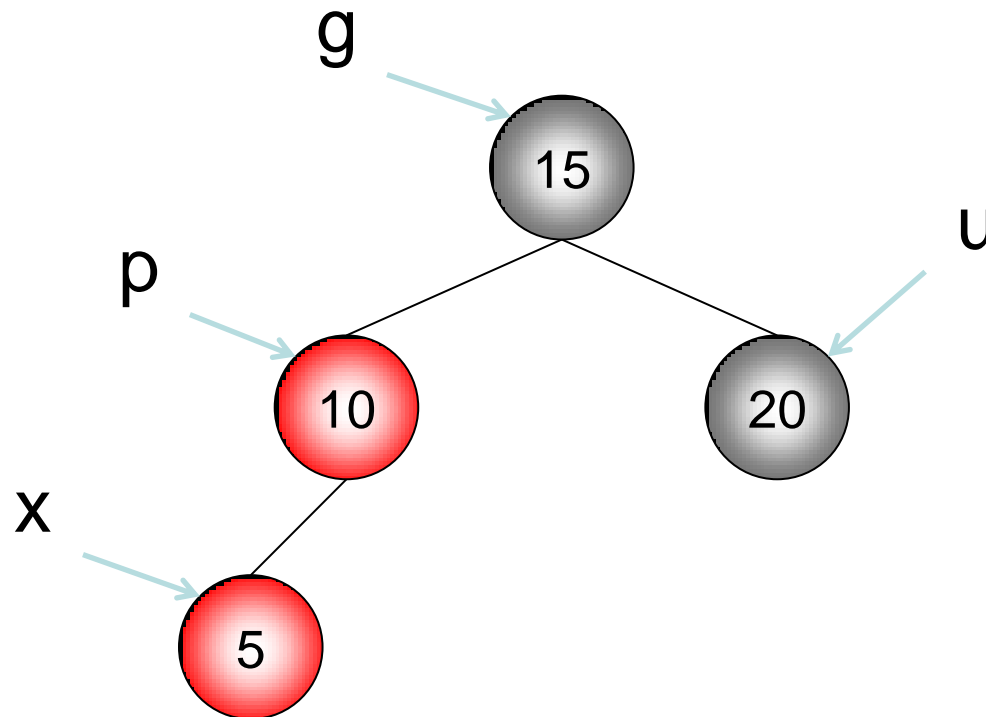
```
void    RBT_Insertion(ref & root, int key) {  
    x = getNode(key, RED, nil);  
    BST_Insert(root, x);  
    Insertion_FixUp(root, x);  
}
```

```

void    BST_Insert(ref &root, ref x) {
    y = nil; z = root;
    while (z != nil) {
        y = z;
        if (x->key < z->key) z = z->left;
        else                z = z->right;
    }
    x->parent = y;
    if (y == nil)
        root = x;
    else
        if (x->key < y->key) y->left = x;
        else                y->right = x;
}

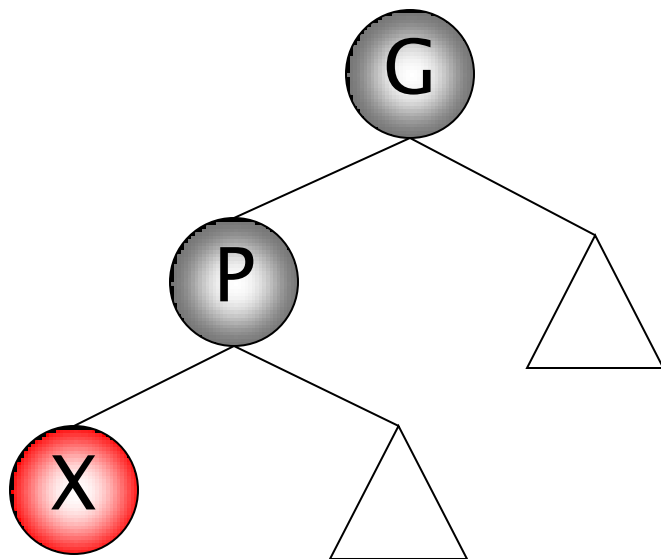
```

- Gọi:
 - x : con trỏ, trỏ đến nút vừa chèn vào
 - p : con trỏ, trỏ đến nút cha của nút trỏ bởi x
 - u : con trỏ, trỏ đến nút anh em của nút trỏ bởi p
 - g : con trỏ, trỏ đến nút cha của nút trỏ bởi p

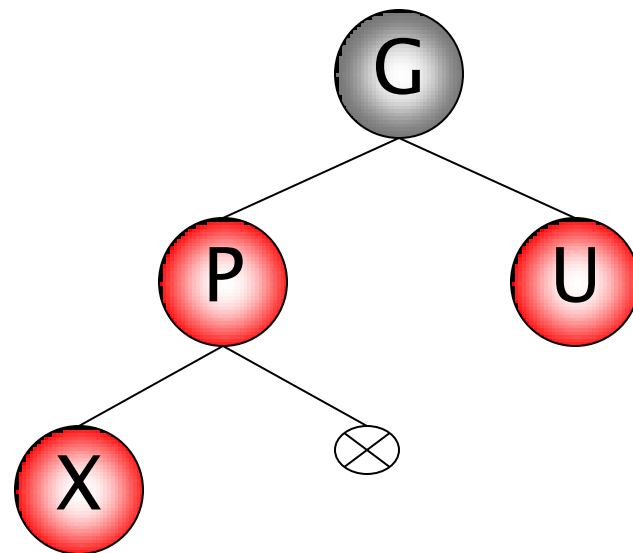


Các trường hợp mất cân bằng

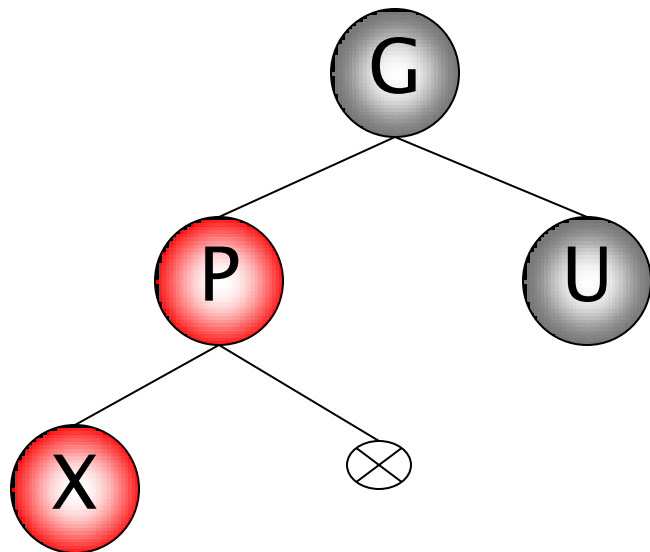
- Nếu p đen: Dừng (1)
- Nếu p đỏ:
 - Nếu u đỏ: đảo màu 3 nút u , p và g (2)
 - Nếu u là đen:
 - x là cháu ngoại của g (3): đảo màu cha và ông, thực hiện phép quay tại ông
 - x là cháu nội của g (4): quay tại cha \rightarrow (3)



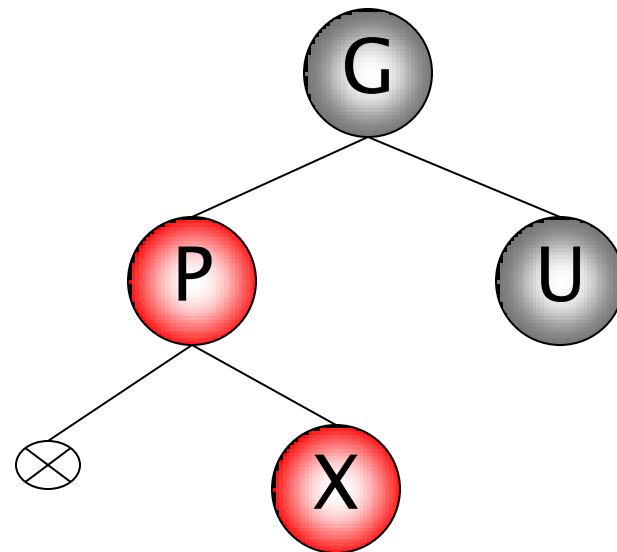
Trường hợp 1



Trường hợp 2

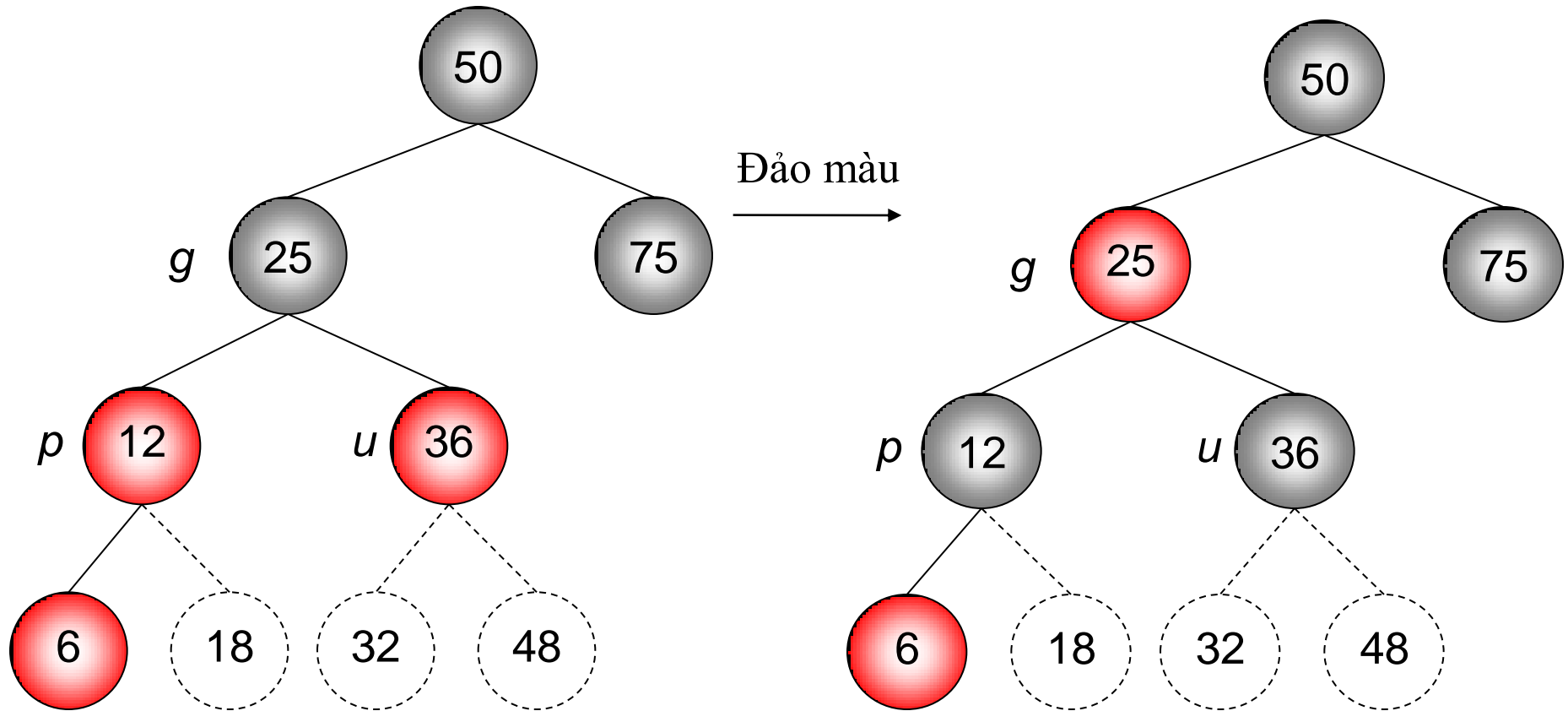


Trường hợp 3

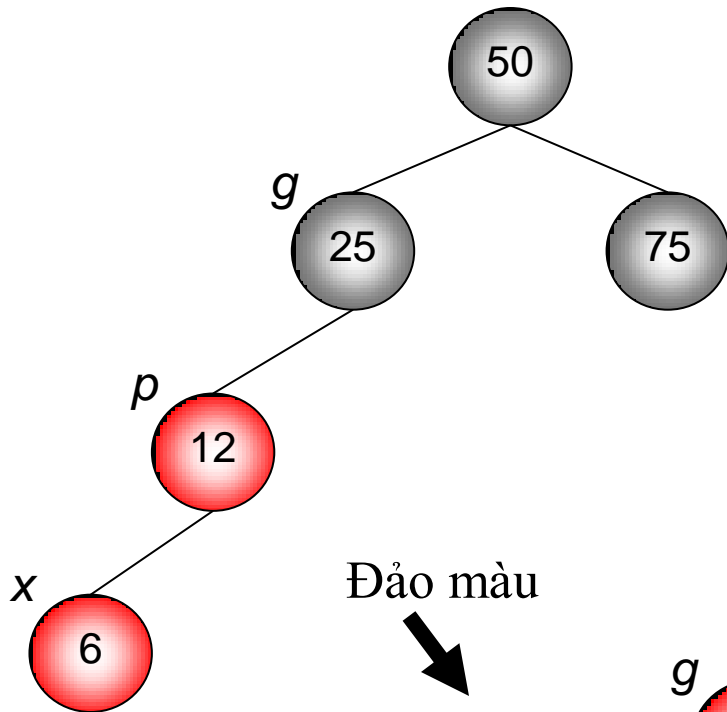


Trường hợp 4

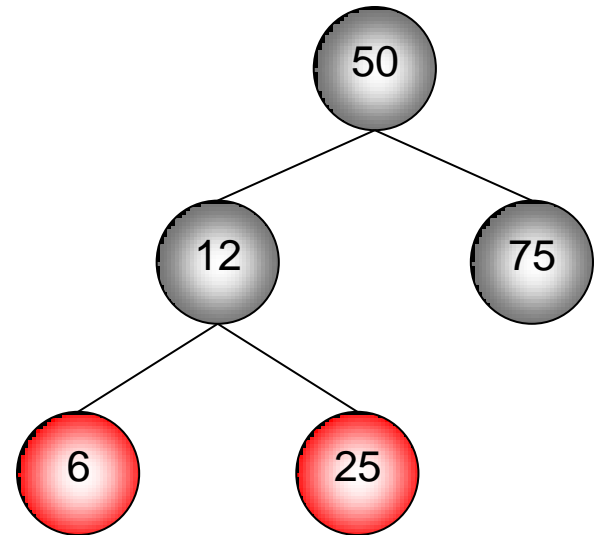
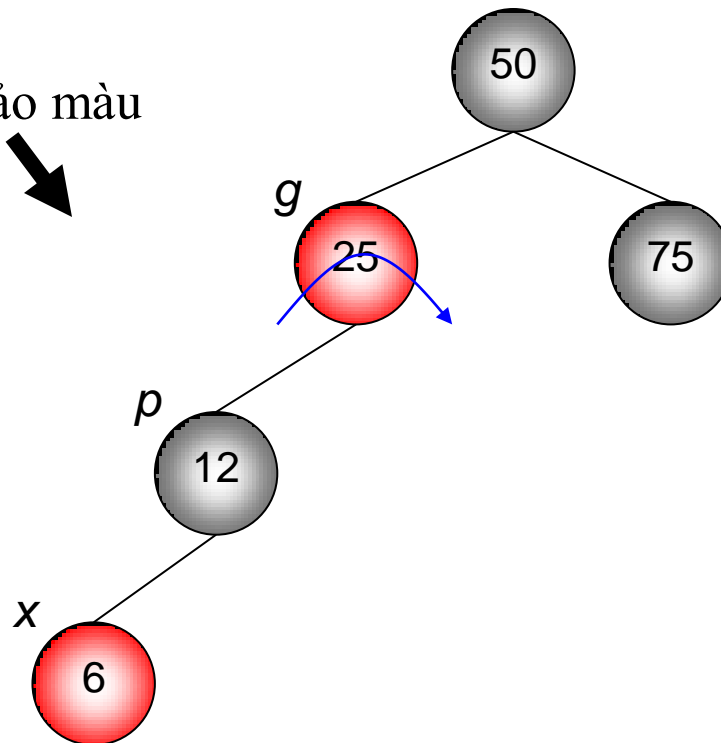
Trường hợp 2



Trường hợp 3

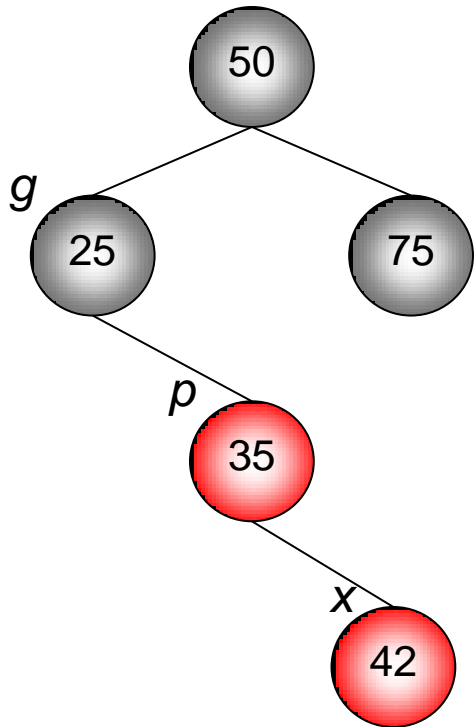


Đảo màu

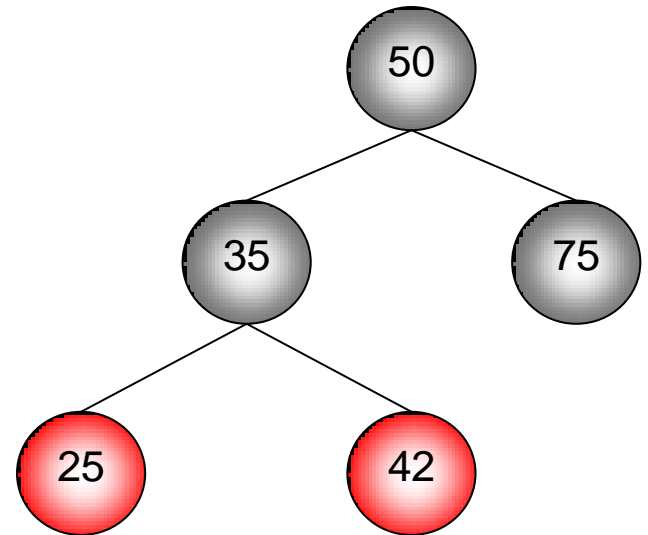
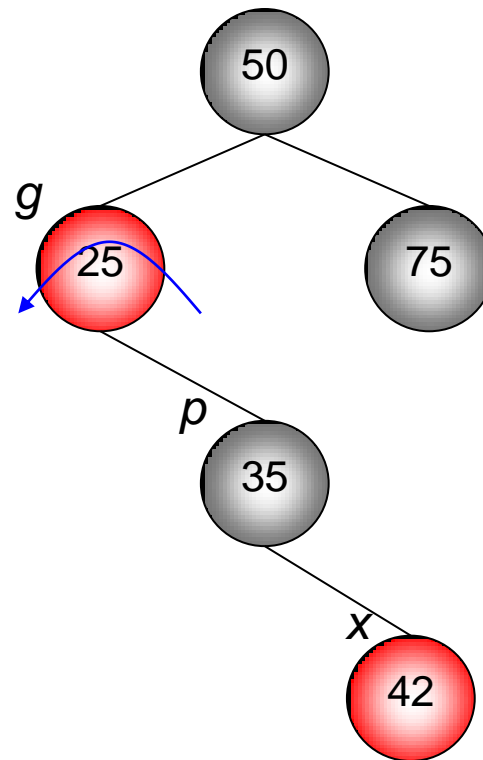


Quay phải

Trường hợp 3

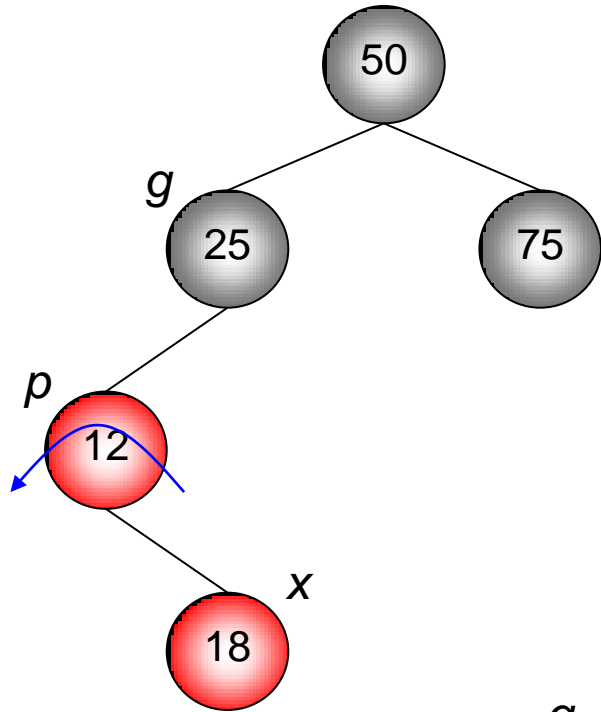


Đảo màu
↓

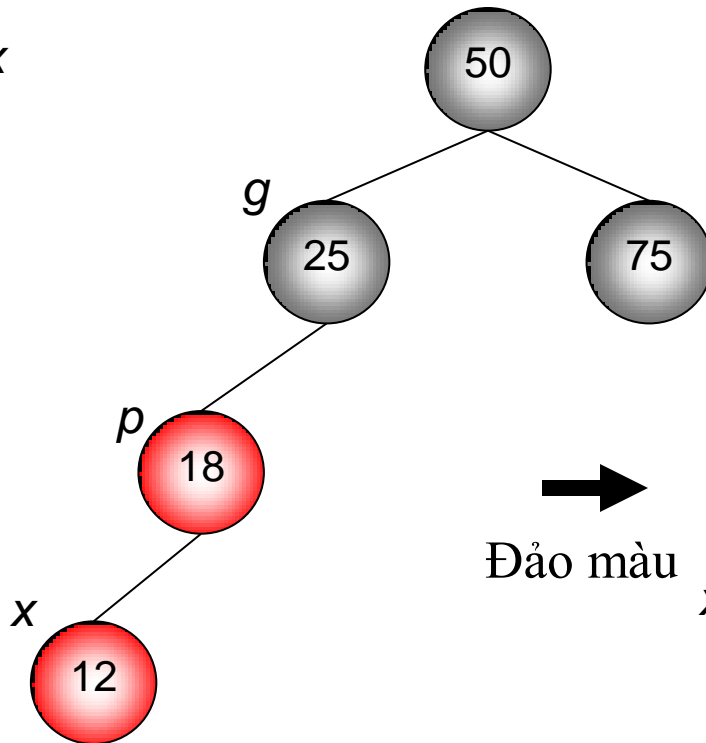


↗ Quay trái

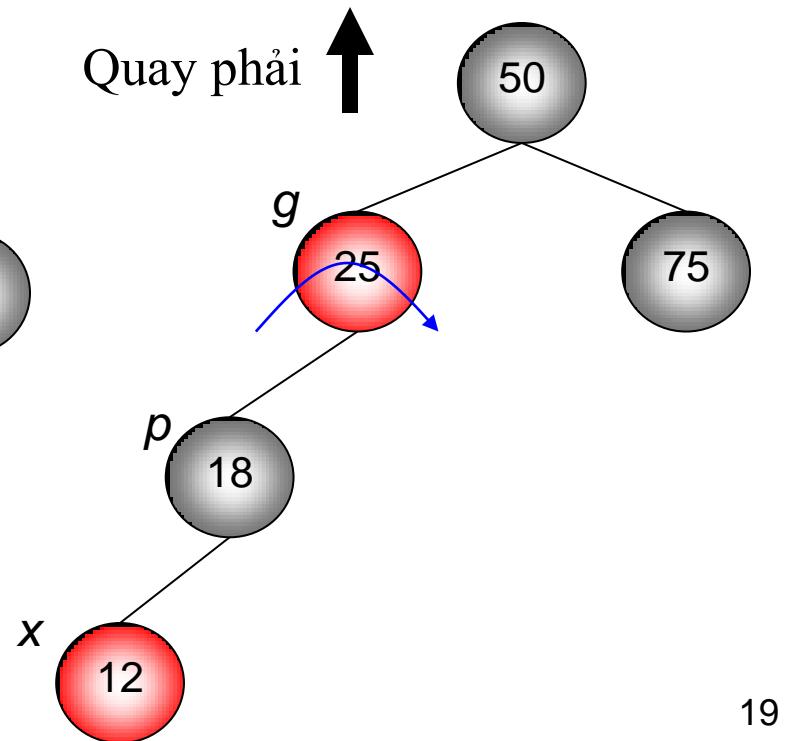
Trường hợp 4



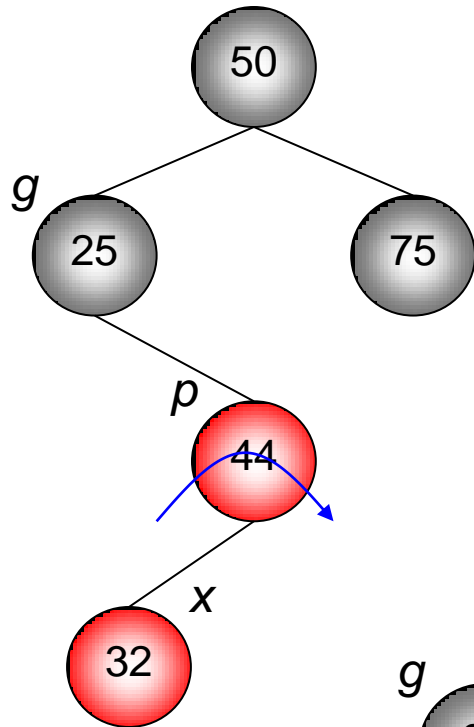
Quay trái



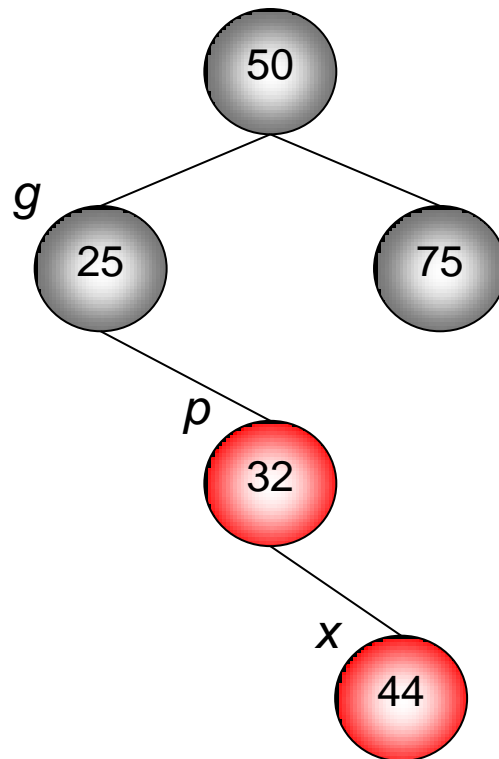
Đảo màu



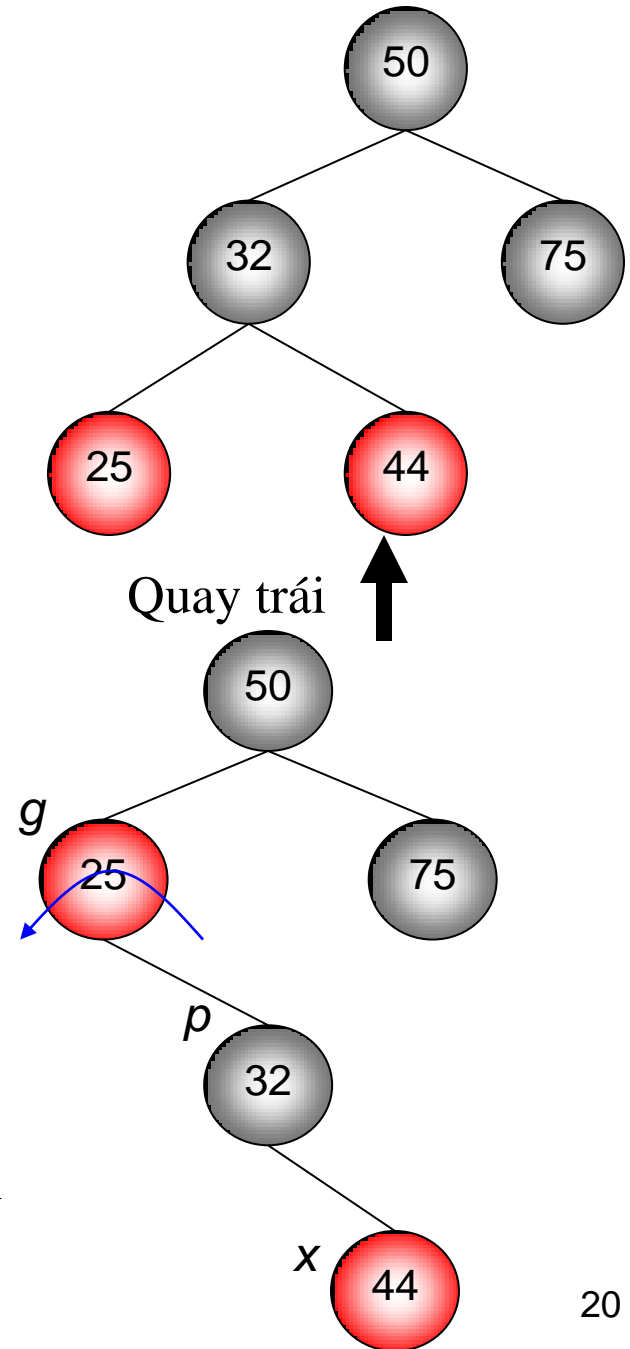
Trường hợp 4



Quay phải



Đảo màu



```
void    Insertion_FixUp(ref &root, ref x) {  
    while (x->parent->color == RED)  
        if (x->parent == x->parent->parent->left)  
            ins_leftAdjust(root, x);  
        else  
            ins_rightAdjust(root, x);  
    root->color = BLACK;  
}
```

```

void    ins_leftAdjust(ref &root, ref &x) {
    u = x->parent->parent->right;
    if (u->color == RED) {
        x->parent->color = BLACK;
        u->color          = BLACK;
        x->parent->parent->color = RED;
        x = x->parent->parent;
    }
    else {
        if (x == x->parent->right) {
            x = x->parent;
            leftRotate(root, x);
        }
        x->parent->color = BLACK;
        x->parent->parent->color = RED;
        rightRotate(root, x->parent->parent);
    }
}

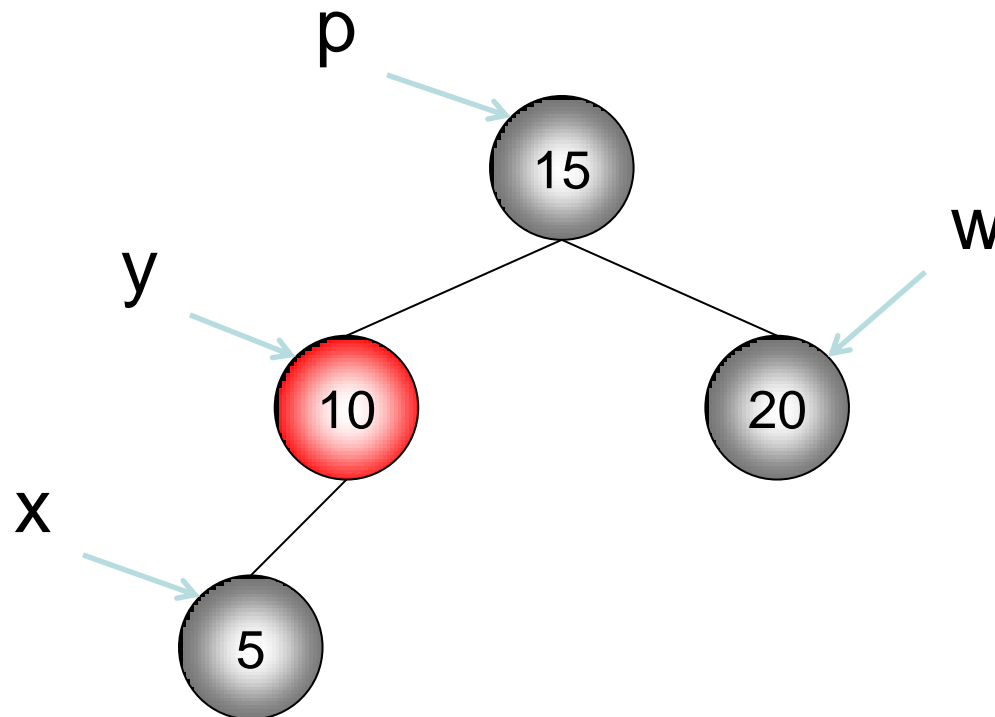
```

Thao tác hủy

- Diễn ra theo 2 giai đoạn:
 - Hủy nút: Tương tự cây nhị phân tìm kiếm
 - ✓ Nút lá
 - ✓ Nút có một cây con
 - ✓ Nút có đầy đủ hai cây con: tìm nút thay thế
 - Cân bằng lại cây
 - ✓ Chỉ quan tâm đến nút bị xóa thật sự

- Gọi:

- y : con trỏ, trỏ đến nút bị xóa thật sự
- x : con trỏ, trỏ đến nút con của nút trỏ bởi y
→ Sẽ thay thế nút trỏ bởi y
- w : con trỏ, trỏ đến nút anh em của nút trỏ bởi y
- p : con trỏ, trỏ đến nút cha của nút trỏ bởi y



- Sau khi xóa:
 - Nếu y là nút đỏ: dừng
 - Nếu y là nút đen:
 - Mọi con đường đi qua y sẽ giảm chiều cao đen
 - Thậm chí, p và x có thể cùng là nút đỏ
- Mất cân bằng

```

void RBT_Deletion(ref &root, int k) {
    z = searchTree(root, k);
    if (z == nil) return;
    y = (z->left == nil) || (z->right == nil) ?
        z : TreeSuccessor(root, z);
    x = (y->left == nil) ? y->right : y->left;
    x->parent = y->parent;
    if (y->parent == nil)    root = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    if (y != z) z->key = y->key;
    if (y->color == BLACK)
        Deletion_FixUp(root, x);
    delete y;
}

```

- *Dấu hiệu đen (black token)*
 - Gán cho nút trở bởi x (con của nút bị xoá thật sự)
 - *Dấu hiệu đen* đi ngược lên cây cho đến khi tiêu chuẩn về *chiều cao đen* được giải quyết
 - Nếu nút chứa *dấu hiệu đen* là:
 - Nút đen → nút đen kép (*doubly black node*)
 - Nút đỏ → nút đỏ-đen (*red-black node*)
- *Dấu hiệu đen* chỉ là khái niệm trừu tượng

Trường hợp 1

Nút đang xét chứa *dấu hiệu đen* và là:

- Nút đỏ (nút đỏ–đen), hoặc
- Nút gốc

Xử lý:

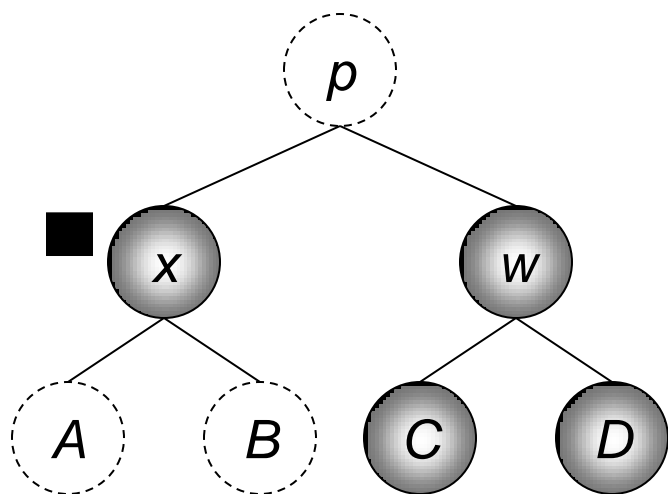
- Tô màu nút đỏ–đen thành đen
- Loại bỏ *dấu hiệu đen* và kết thúc

Trường hợp 2

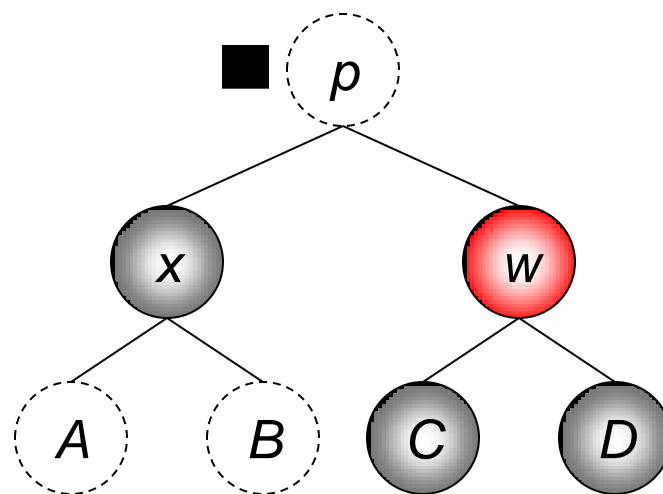
- Nút đang xét (x) là nút đen kép
- Nút anh em w màu đen
- Hai nút con của nút anh em w (nút cháu) màu đen

Xử lý:

- Đổi màu nút anh em w sang đỏ
- Di chuyển *dấu hiệu đen* lên một cấp



Đổi màu →

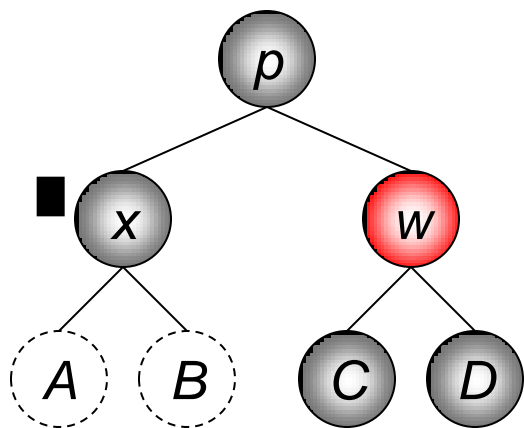


Trường hợp 3

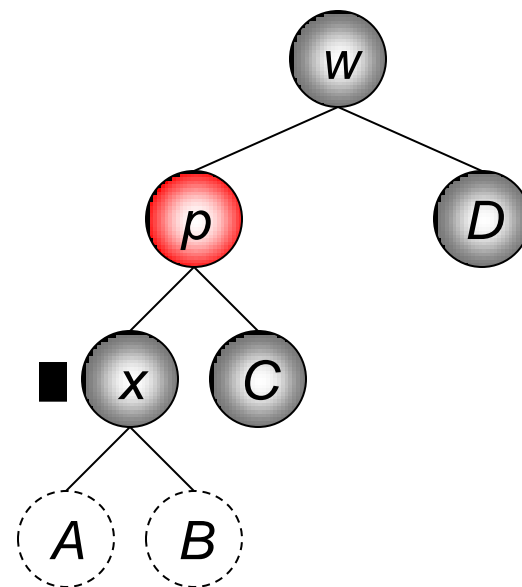
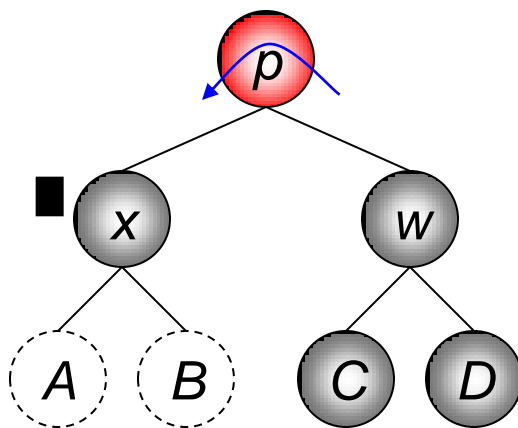
- Nút đang xét (x) là nút đen kép
- Nút anh em w màu đỏ

Xử lý:

- Đảo màu nút cha p và nút anh em w
- Thực hiện phép quay tại cha
- *Dấu hiệu đen* vẫn thuộc nút x ban đầu



Đổi màu ↘



↗ Quay trái

Trường hợp 4

- Nút đang xét (x) là nút đen kép
- Nút anh em w là đen
- Nút anh em có tối thiểu một nút con là đỏ

Xử lý: Xét 2 nút cháu của cha của nút đen kép

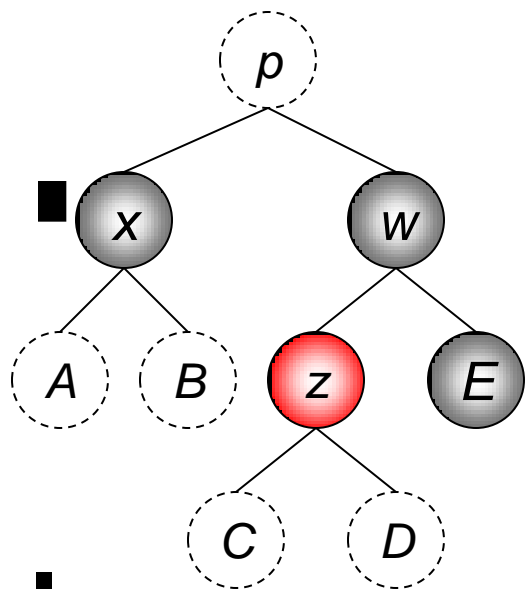
a) Nút cháu ngoại là đen

- Đảo màu cháu nội và nút anh em (w)
- Quay tại nút anh em (w)
- Chuyển sang trường hợp b

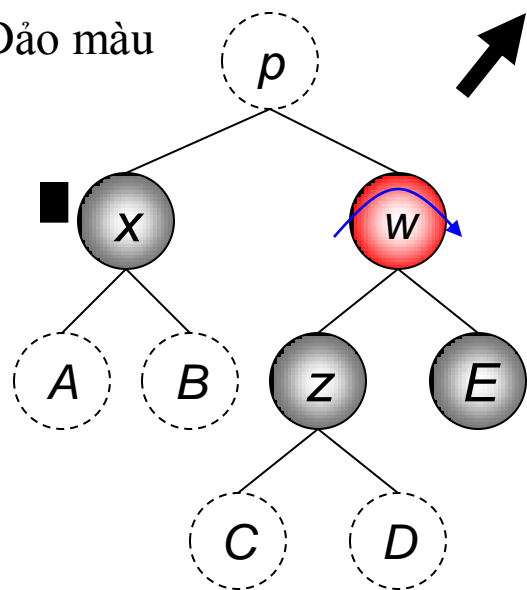
b) Nút cháu ngoại là đỏ

- Nút anh em w (sẽ là gốc cục bộ) nhận màu nút cha
- Nút ông và nút cháu ngoại nhận màu đen
- Quay tại nút ông

Trường hợp (a)

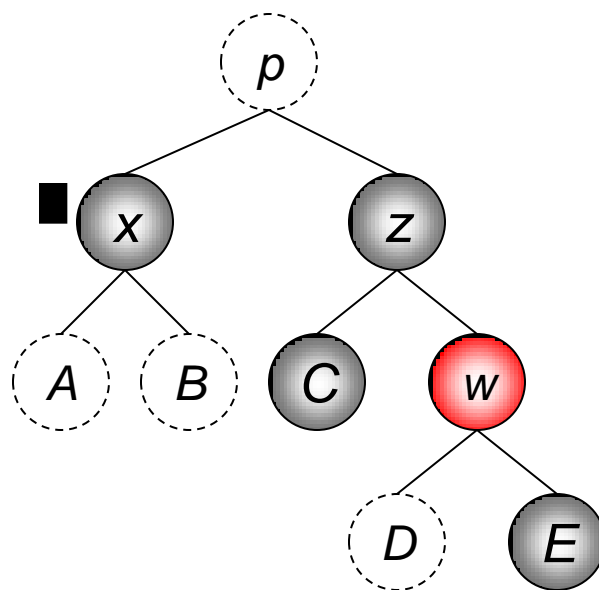


Đảo màu

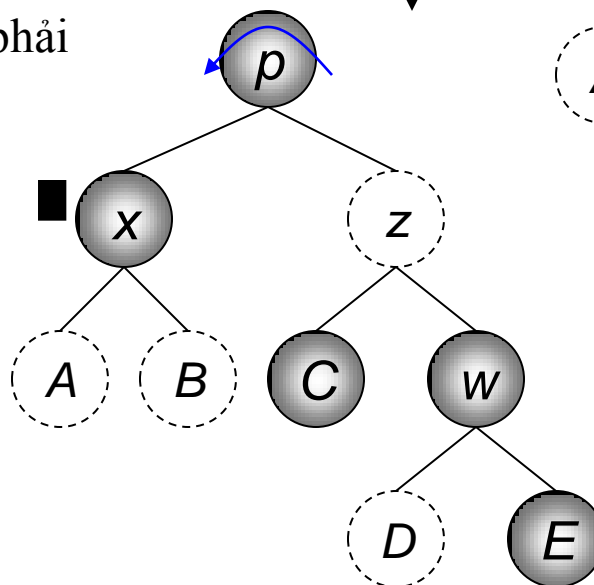


Quay phải

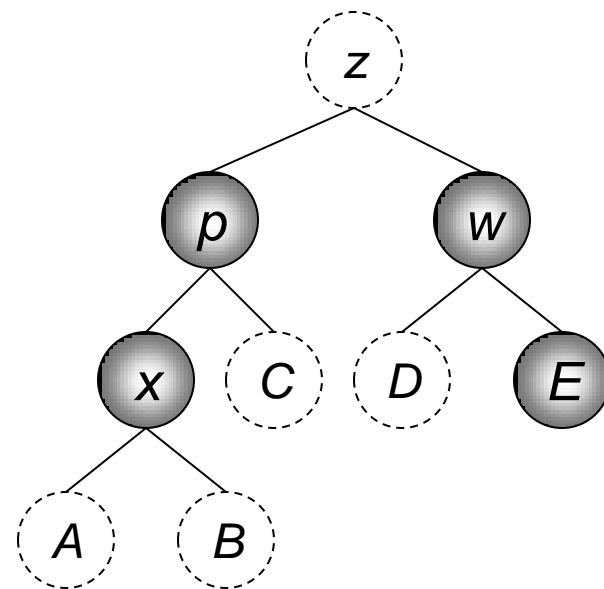
Trường hợp (b)



Chuyển màu



Quay trái



```

void    Deletion_FixUp(ref root, ref x) {
    while ((x->color == BLACK) && (x != root))
        if (x == x->parent->left)
            del_leftAdjust(root, x);
        else
            del_rightAdjust(root, x);
    x->color = BLACK;
}

void    del_leftAdjust(ref & root, ref & x) {
    w = x->parent->right;
    if (w->color == RED) {
        w->color          = BLACK;
        x->parent->color = RED;
        leftRotate(root, x->parent);
        w = x->parent->right;
    }
}

```

```

if ((w->right->color == BLACK) &&
    (w->left->color == BLACK)) {
    w->color = RED; x = x->parent;
}
else {
    if (w->right->color == BLACK) {
        w->left->color = BLACK;
        w->color = RED;
        rightRotate(root, w);
        w = x->parent->right;
    }
    w->color = x->parent->color;
    x->parent->color = BLACK;
    w->right->color = BLACK;
    leftRotate(root, x->parent);
    x = root;
}
}

```