CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Bài 05: Tìm kiếm

Nguyễn Thị Tâm
nguyenthitam.hus@gmail.com

Ngày 4 tháng 11 năm 2025

Nội dung

Bài toán tìm kiếm

Bài toán tìm kiếm

Cho dãy cách phần tử $a[0..n-1]$ và phần tử x, ta cần tìm vị trí *index* sao cho $a[index] = x$ hoặc trả về giá trị -1 nếu không có phần tử x trong dãy.

Tìm kiếm tuần tự

Tìm kiếm tuần tự - linear seach or sequential search

Thuật toán: Bắt đầu từ phần tử đầu tiên, duyệt qua từng phần tử cho đến khi tìm được phần tử đích hoặc kết luận không tìm được

Tìm kiếm tuần tự

Tìm kiếm tuần tự - linear seach or sequential search

Thuật toán: Bắt đầu từ phần tử đầu tiên, duyệt qua từng phần tử cho đến khi tìm được phần tử đích hoặc kết luận không tìm được

```
int linearSearch (int arr[], int x)
{
    for (int i = 0; i<arr.length; i++){
        if (arr[i]==x){
            return i;
        }
    }
    return -1;
}
```

Tìm kiếm nhị phân trên mảng được sắp

Tìm kiếm nhị phân - binary search

Ý tưởng: Chia đôi mảng, mỗi lần so sánh phần tử giữa với x, nếu phần tử x nhỏ hơn thì xét nửa trái, ngược lại xét nửa phải

Tìm kiếm nhị phân trên mảng được sắp

Tìm kiếm nhị phân - binary search

Ý tưởng: Chia đôi mảng, mỗi lần so sánh phần tử giữa với x, nếu phần tử x nhỏ hơn thì xét nửa trái, ngược lại xét nửa phải

```
int binarySearch(int arr[], int x)
{
    int l = 0, r = arr.length - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}
```

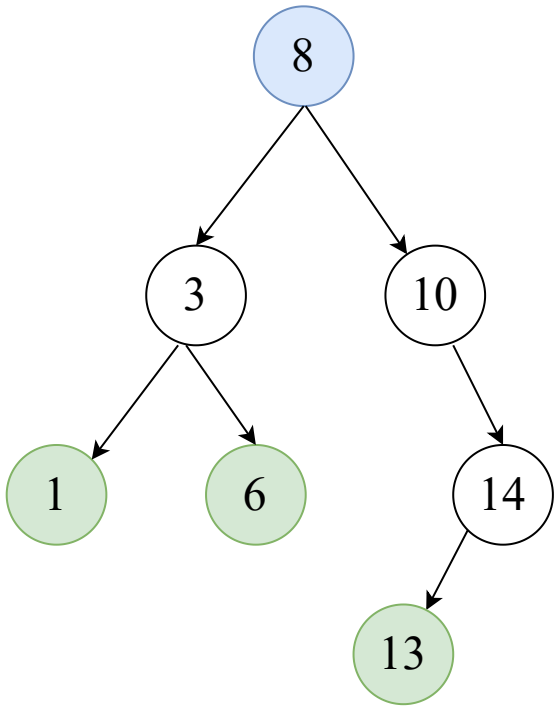
Nội dung

Cây tìm kiếm nhị phân

Định nghĩa cây tìm kiếm nhị phân - Binary Search Tree - BST

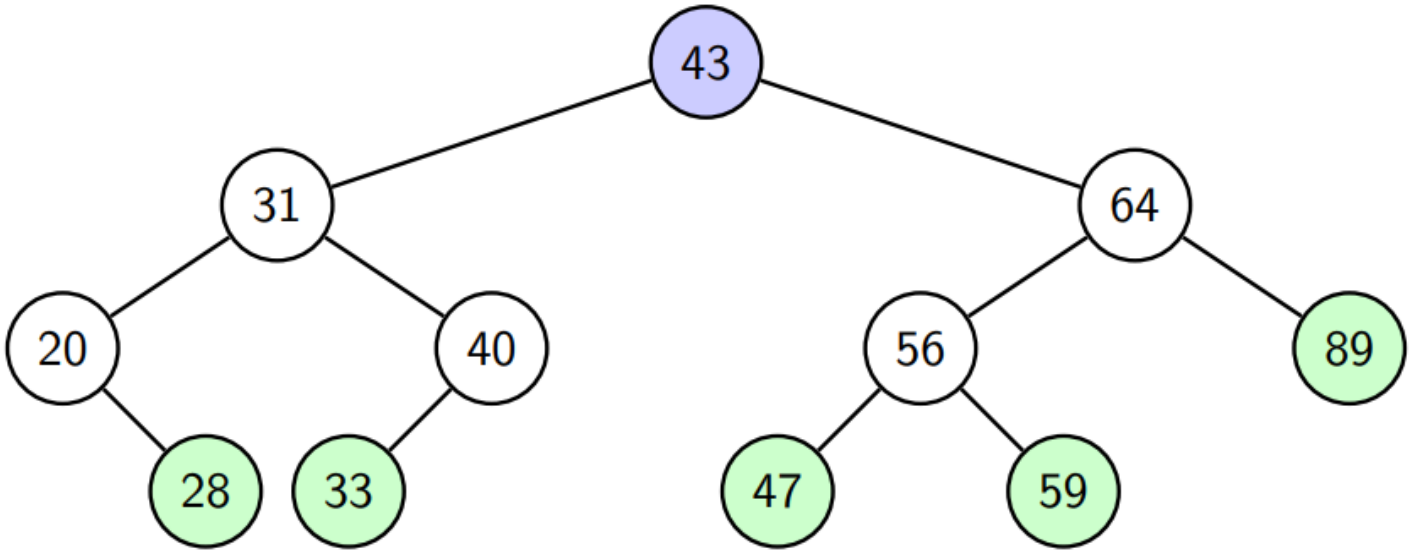
Cây tìm kiếm nhị phân là cây nhị phân mà mỗi nút trong cây đều có một khoá, mọi khoá k đều thoả mãn điều kiện:

- Mọi khoá trên cây con trái đều nhỏ hơn khoá k
- Mọi khoá trên cây con phải đều lớn hơn khoá k



Cây tìm kiếm nhị phân

Duyệt cây theo thứ tự giữa cho ta dãy khoá được sắp:
20; 28; 31; 33; 40; 43; 47; 56; 59; 64; 89



Biểu diễn cây BST

```
class BSTNode {
    int data;
    BSTNode left;
    BSTNode right;
    public BSTNode(int data)
    {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

Các phép toán cơ bản trên cây tìm kiếm nhị phân

- search - Tìm kiếm nút có giá trị khoá bằng k trên cây
- findMax - Trả về nút có giá trị khoá lớn nhất trên cây
- findMin - Trả về nút có giá trị khoá nhỏ nhất trên cây
- successor - Trả lại nút kế cận sau một nút
- predecessor - Trả lại nút kế cận trước một nút
- insert - Chèn một nút mới vào cây
- delete - Xoá nút khỏi cây

Các thao tác cơ bản trên BST

Tìm kiếm

- Nếu khoá cần tìm bằng nút hiện tại
- Nếu khoá cần tìm nhỏ hơn nút hiện tại thì tìm tiếp cây con trái
- Nếu khoá cần tìm lớn hơn nút hiện tại thì tìm tiếp cây con phải

Các thao tác cơ bản trên BST

Tìm kiếm

- Nếu khoá cần tìm bằng nút hiện tại
- Nếu khoá cần tìm nhỏ hơn nút hiện tại thì tìm tiếp cây con trái
- Nếu khoá cần tìm lớn hơn nút hiện tại thì tìm tiếp cây con phải

```
public BSTNode searchRecursive(BSTNode root, int key) {
    if (root == null || root.data == key) {
        return root;
    }
    if (key < root.data) {
        return searchRecursive(root.left, key);
    }
    return searchRecursive(root.right, key);
}
```

Các thao tác cơ bản trên BST

Tìm kiếm

- Nếu khoá cần tìm bằng nút hiện tại
- Nếu khoá cần tìm nhỏ hơn nút hiện tại thì tìm tiếp cây con trái
- Nếu khoá cần tìm lớn hơn nút hiện tại thì tìm tiếp cây con phải

Các thao tác cơ bản trên BST

Tìm kiếm

- Nếu khoá cần tìm bằng nút hiện tại
- Nếu khoá cần tìm nhỏ hơn nút hiện tại thì tìm tiếp cây con trái
- Nếu khoá cần tìm lớn hơn nút hiện tại thì tìm tiếp cây con phải

```
public BSTNode searchIterative(BSTNode root, int key) {
    BSTNode current = root;
    while (current != null && current.data != key) {
        if (key < current.data) {
            current = current.left;
        } else {
            current = current.right;
        }
    }
    return current;
}
```

Các thao tác cơ bản trên cây BST

Tìm kiếm phần tử lớn nhất, nhỏ nhất

Việc tìm phần tử nhỏ nhất (lớn nhất) trên cây BST có thể thực hiện nhờ việc di chuyển trên cây

- Để tìm phần tử nhỏ nhất, ta đi theo con trái đến khi gặp NULL
- Để tìm phần tử lớn nhất, ta đi theo con phải đến khi gặp NULL

Các thao tác cơ bản trên cây BST

Tìm kiếm phần tử lớn nhất, nhỏ nhất

```
public int findMax(BSTNode root) {
    if (root == null) {
            throw new IllegalStateException("Tree is null");
    }
    if (root.right == null) {
        return root.data;
    }
    return findMax(root.right);
}
```

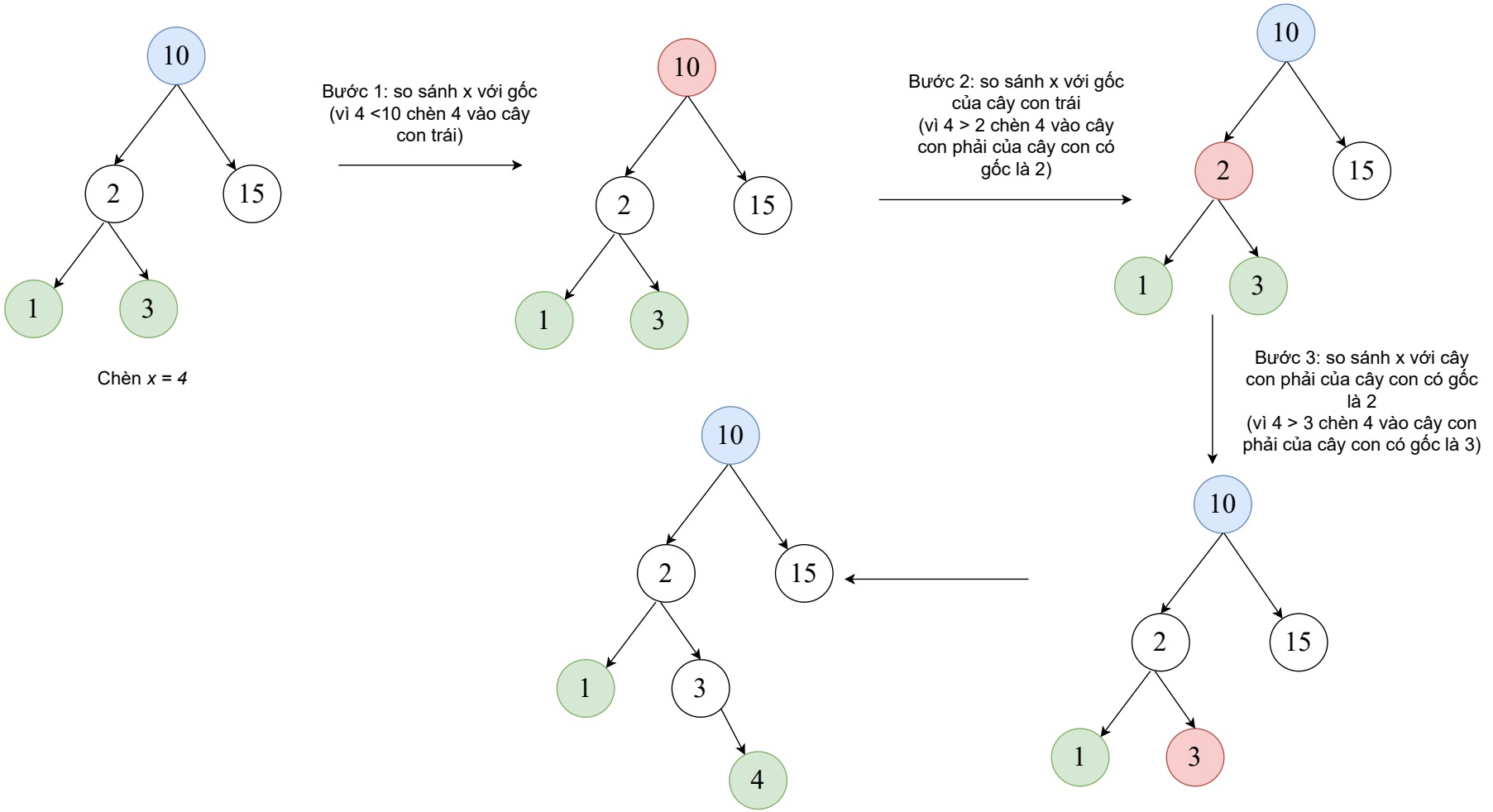
Các thao tác cơ bản trên BST

Chèn phần tử

- Trường hợp 1: Cây rỗng
  - Tạo một nút mới
  - Cho cây con trái và cây con phải của nút mới nhận giá trị **null**
- Trường hợp 2: Trong cây có tồn tại phần tử
  - So sánh giá trị của nút mới thêm vào với gốc.
  - Nếu giá trị mới thêm vào nhỏ hơn gốc thì thêm vào cây con trái (thực hiện đệ quy việc thêm nút ở cây con trái)
  - Nếu giá trị mới thêm vào lớn hơn gốc thì thêm vào cây con phải (thực hiện đệ quy việc thêm nút ở cây con phải)
  - Gắn nút con là nút con của nút cha tìm được. Chú ý là nút mới thêm vào luôn là nút lá

Các thao tác cơ bản trên cây BST

Chèn phần tử



Các thao tác cơ bản trên cây BST

Chèn phần tử

```
BSTNode root;
void insert(int data) {
    root = insertNode(root, data);
}

BSTNode insertNode(BSTNode root, int data) {
    // TH1
    if (root == null) {
        root = new BSTNode(data);
        return root;
    }
    // TH2
    else if (data < root.data)
        root.left = insertNode(root.left, data);
    else if (data > root.data)
        root.right = insertNode(root.right, data);
    return root;
}
```

Các thao tác cơ bản trên cây BST

Xoá phần tử

Khi loại bỏ một nút, cần phải đảm bảo cây thu được vẫn là cây nhị phân tìm kiếm. Có 4 trường hợp có thể xảy ra khi xoá nút

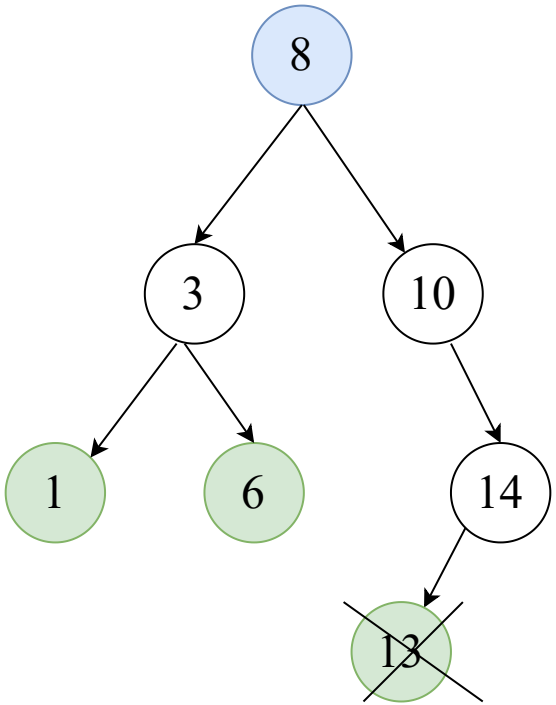
- Trường hợp 1: Nút cần xoá là lá
- Trường hợp 2: Nút cần xoá chỉ có con trái
- Trường hợp 3: Nút cần xoá chỉ có con phải
- Trường hợp 4: Nút cần xoá có hai con

Các thao tác cơ bản trên cây BST

Xoá phần tử

Trường hợp 1: Nút cần xoá là nút lá

Thao tác: Chữa lại nút cha của nút cần xoá có con rỗng

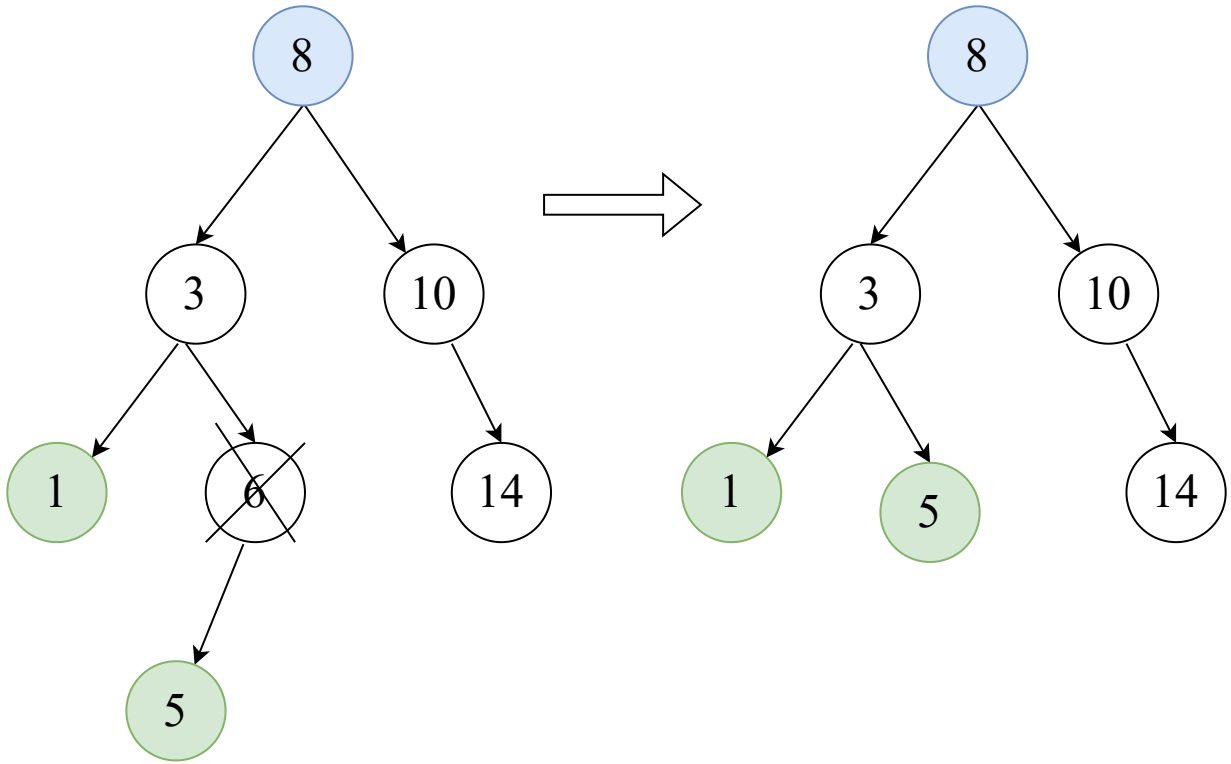


Các thao tác cơ bản trên cây BST

Xoá phần tử

Trường hợp 2: Nút cần xoá có con trái mà không có con phải

Thao tác: Gắn cây con trái của nút cần xoá vào nút cha của nút cần xoá

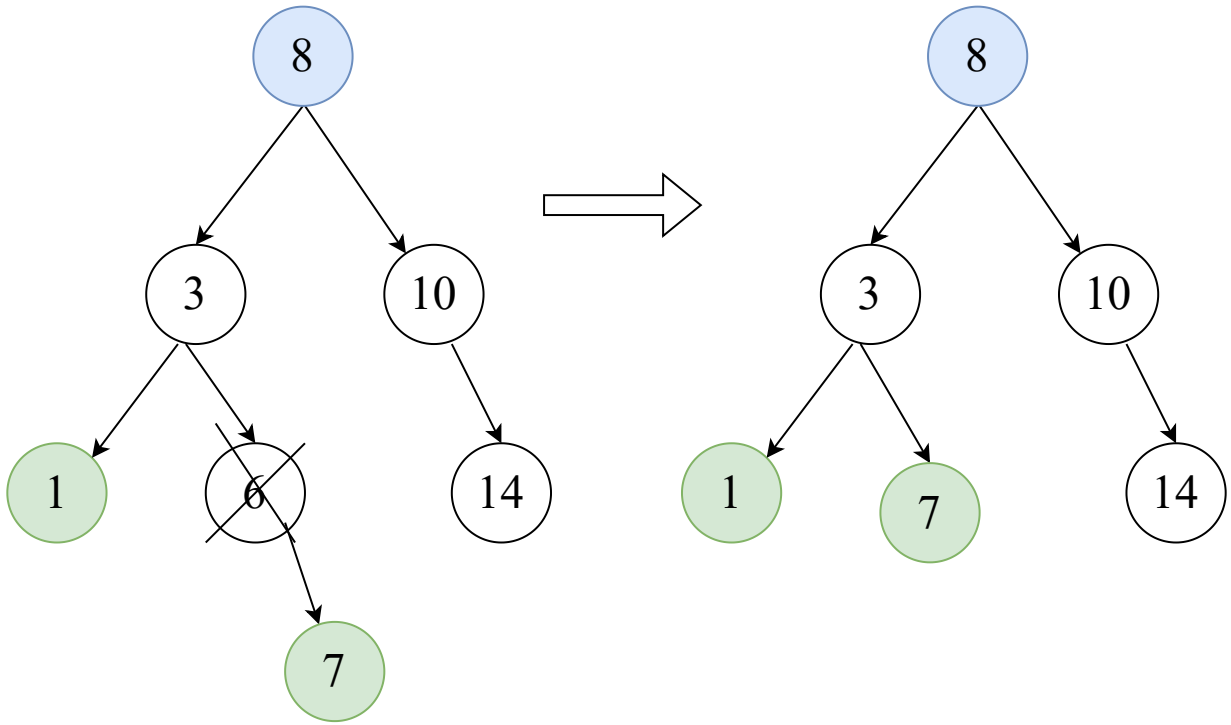


Các thao tác cơ bản trên cây BST

Xoá phần tử

Trường hợp 3: Nút cần xoá có con phải mà không có con trái

Thao tác: Gắn cây con phải của nút cần xoá vào nút cha của nút cần xoá



Các thao tác cơ bản trên cây BST

Xoá phần tử

Trường hợp 4: Nút cần xoá x có đầy đủ hai con

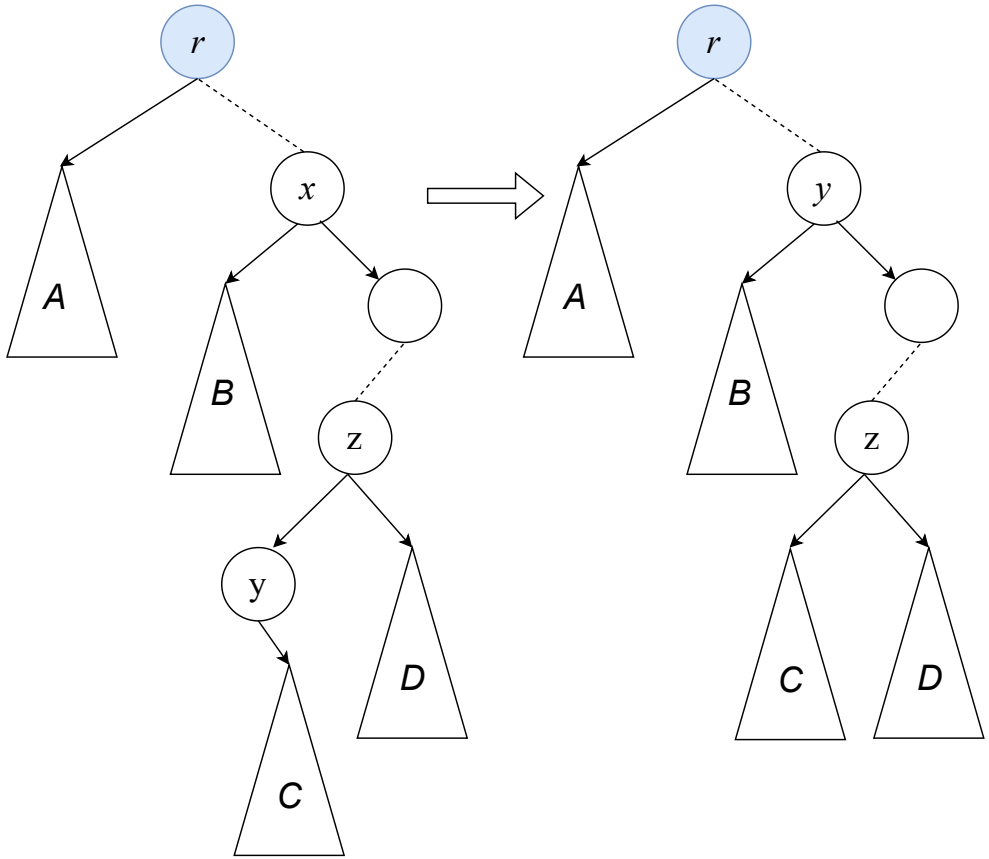
Thao tác:

1. Chọn nút y để thế vào chỗ của nút x, nút y sẽ là nút kế tiếp của nút x. Như vậy, y là giá trị nhỏ nhất còn lớn hơn x, nói cách khác, y là giá trị nhỏ nhất của cây con phải của x
2. Gỡ nút y khỏi cây
3. Nối con phải của y vào cha của y
4. Thay thế y vào nút cần xoá

Các thao tác cơ bản trên cây BST

Xoá phần tử

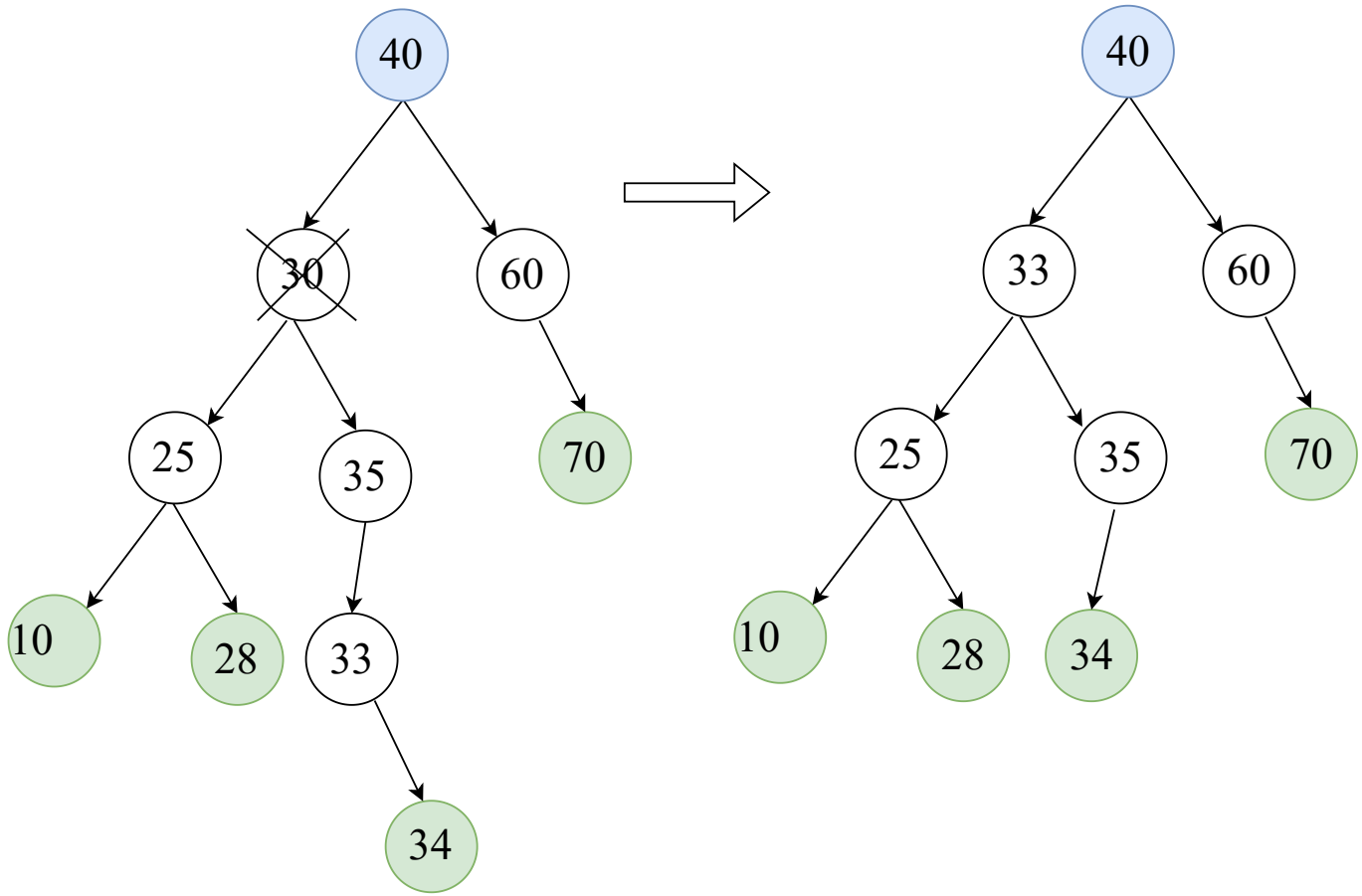
Trường hợp 4: Nút cần xoá x có đầy đủ hai con



Các thao tác cơ bản trên cây BST

Xoá phần tử

Trường hợp 4: Nút cần xoá x có đầy đủ hai con



Các thao tác cơ bản trên cây BST

Xoá phần tử

```
public BSTNode deleteNode(BSTNode root, int key) {
    if (root == null)
        return root;
    if (key < root.data)
        root.left = deleteNode(root.left, key);
    else if (key > root.data)
        root.right = deleteNode(root.right, key);
    else {
        if (root.left == null)
            return root.right;
        else if (root.right == null)
            return root.left;
        BSTNode node = findMin(root.right);
        root.data = node.data;
        root.right = deleteNode(root.right, node.data);
    }
    return root;
}
```

Các thao tác cơ bản trên cây BST

Xoá phần tử

```
private BSTNode findMin(BSTNode node) {
    BSTNode current = node;
    while (current.left != null) {
        current = current.left;
    }
    return current;
}
```

Sắp xếp sử dụng BST

Sắp xếp sử dụng BST

Duyệt cây BST theo thứ tự giữa ra dãy khoá được sắp xếp. Nên ta có thể sử dụng cây BST để giải quyết bài toán sắp xếp:

- Xây dựng cây BST tương ứng với dãy số đã cho bằng cách chèn từng khoá trong dãy vào cây BST
- Duyệt cây BST thu được theo thứ tự giữa để đưa ra dãy được sắp

Phân tích hiệu quả của sắp xếp sử dụng cây BST

- Tình huống trung bình: $O(nlogn)$ vì chèn phần tử thứ i tốn khoản $log_2(i)$ phép so sánh
- Tình huống tồi nhất: $O(n^2)$ bởi vì bổ sung phần tử $i+1$ tốn i phép so sánh (ví dụ dãy được sắp)

Độ phức tạp của việc sắp xếp sử dụng BST

- Độ phức tạp trung bình của các thao tác. Do độ cao trung bình của cây BST là : $h = O(logn)$, từ đó suy ra độ phức tạp trung bình của các thao tác với BST là:
  - Chèn $O(logn)$
  - Xoá $O(logn)$
  - Tìm giá trị lớn nhất/nhỏ nhất $O(logn)$
  - Sắp xếp $O(nlogn)$
- Trường hợp cây bị mất cân đối: chiều cao của cây là $h = n$

Độ phức tạp của việc sắp xếp sử dụng BST

Vấn đề đặt ra

Có cách nào để tạo ra một cây BST sao cho chiều cao của cây là nhỏ nhất có thể, hay nói cách khác chiều cao $h = logn$. Có hai cách tiếp cận

- Luôn giữ cho cây cân bằng tại mọi thời điểm
- Thỉnh thoảng kiểm tra xem cây có mất cân bằng hay không

Cây AVL (Adelson-Velskii and Landis) Trees

Định nghĩa cây AVL

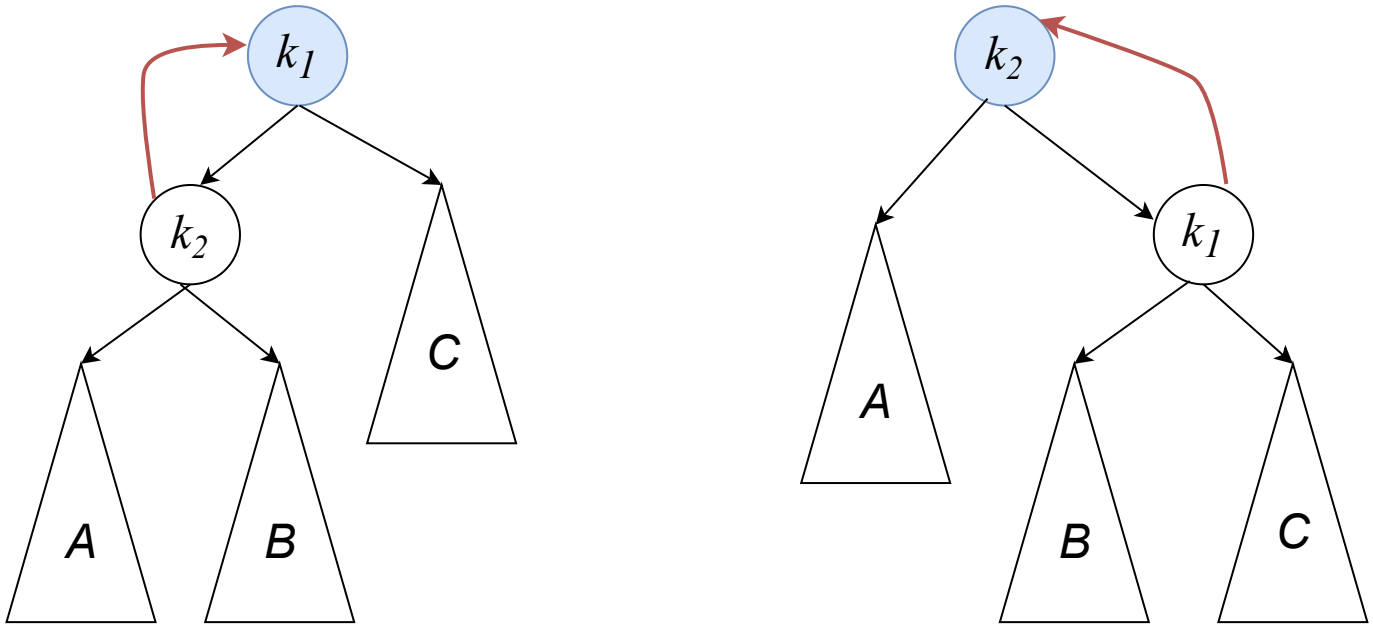
Một cây được gọi là cây AVL nếu:

- Là cây tìm kiếm nhị phân
- Với mỗi nút bất kì, chiều cao của cây con trái và chiều cao của cây con phải không vượt quá 1.
- Cả cây con phải và cây con trái đều là AVL

Hệ số cân bằng - balance factor

Hệ số cân bằng của nút x, ký kiệu là $bal(x)$ là hiệu chiều cao của cây con trái trừ chiều cao cây con phải của nút x

Các phép quay cây AVL



Quay phải quanh k_1 Quay trái quanh k_2

Hình 1: Hai phép quay cơ bản không làm mất tính chất của cây AVL

Các phép quay cây AVL

Phép quay phải

```
BSTNode rotateRight(BSTNode x) {
    if (x == null) {
        return x;
    }

    BSTNode xL = x.left;
    BSTNode xLR = xL.right;
    // Perform rotation
    x.left = xLR;
    xL.right = x;
     // Update heights
    x.height = 1 + Math.max(height(x.left), height(x.right));
    xL.height = 1 + Math.max(height(xL.left), height(xL.right));
    return xL;
}
```

Các phép quay cây AVL

Phép quay trái

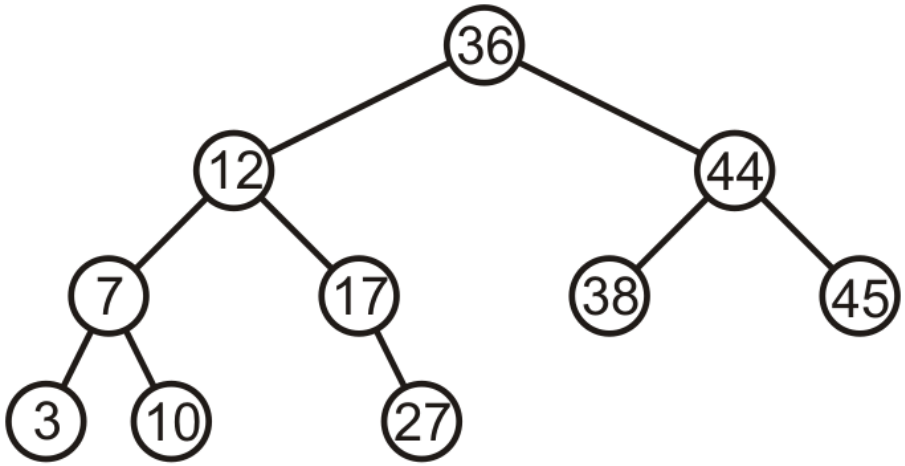
```
BSTNode rotateLeft(BSTNode x) {
    if (x == null) {
        return x;
    }
    BSTNode xR = x.right;
    BSTNode xRL = xR.left;
    // Perform Rotation
    x.right = xRL;
    xR.left = x;
    // Update heights
    x.height = 1 + Math.max(height(x.left), height(x.right));
    xR.height = 1 + Math.max(height(xR.left), height(xR.right));

    return xR;
}
```

Duy trì sự cân bằng của cây

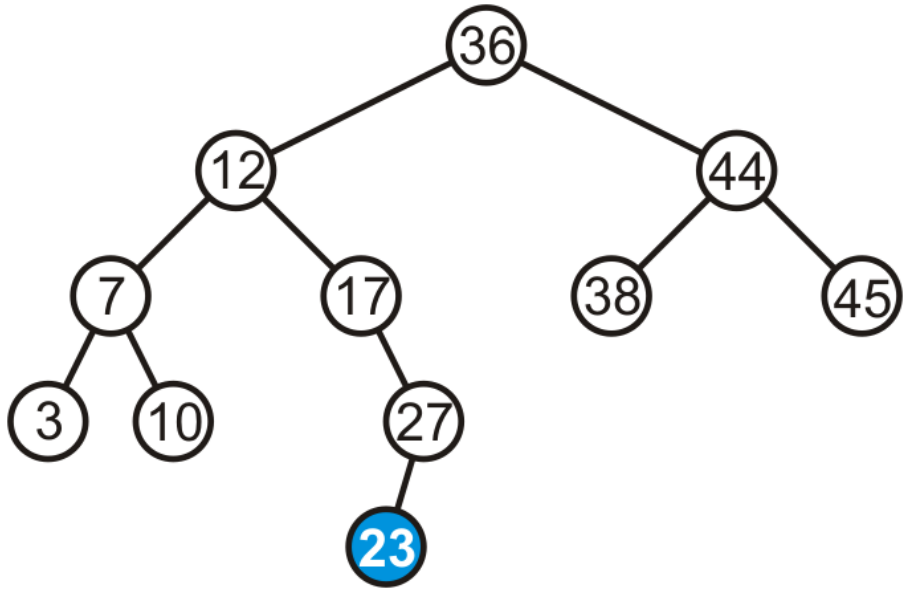
Nếu một cây là cây AVL đã cân bằng, khi chèn một nút có thể làm mất tính chất cân bằng của cây

- Chiều cao của cây con khác nhau 1
- Phép chèn làm tăng chiều cao của cây con sâu hơn lên 1



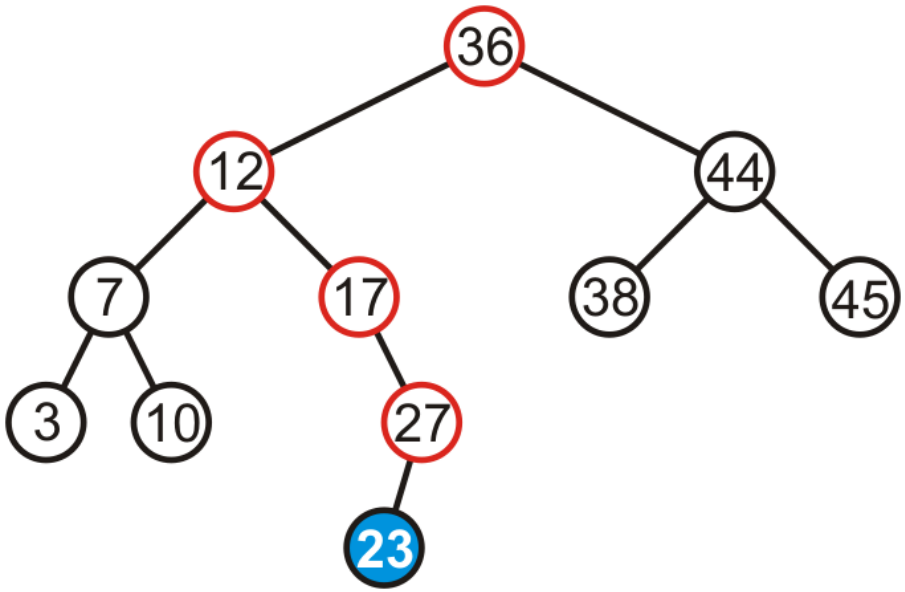
Duy trì sự cân bằng của cây (1)

Chèn nút 23 vào cây



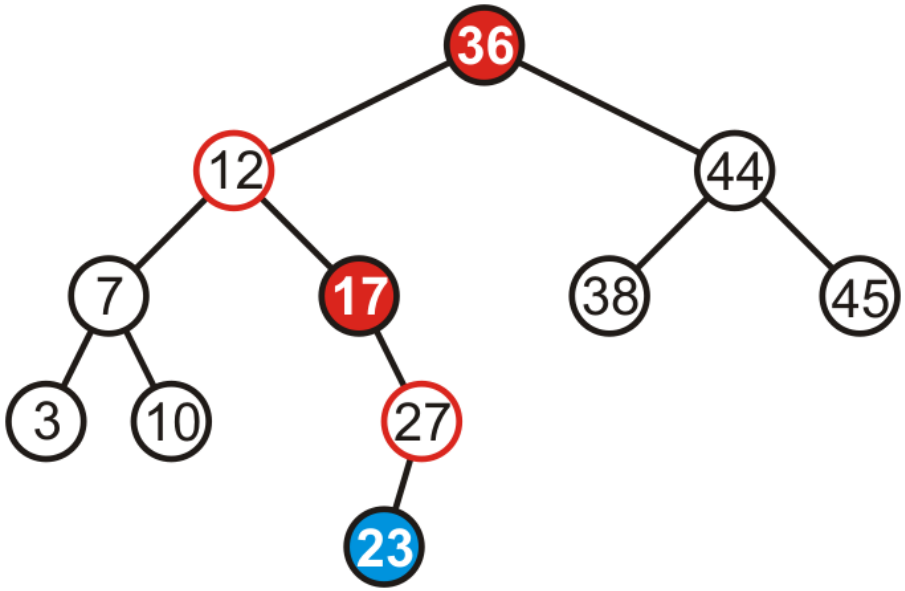
Duy trì sự cân bằng của cây (2)

Chiều cao của một số cây con thay đổi (tăng lên 1)



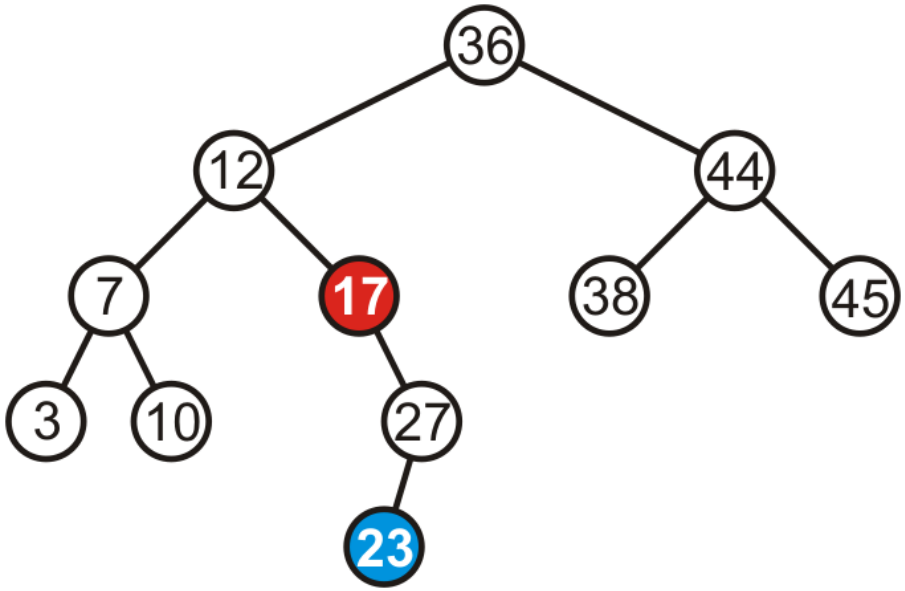
Duy trì sự cân bằng của cây (3)

Hai nút bị mất tính cân bằng: 17 và 36



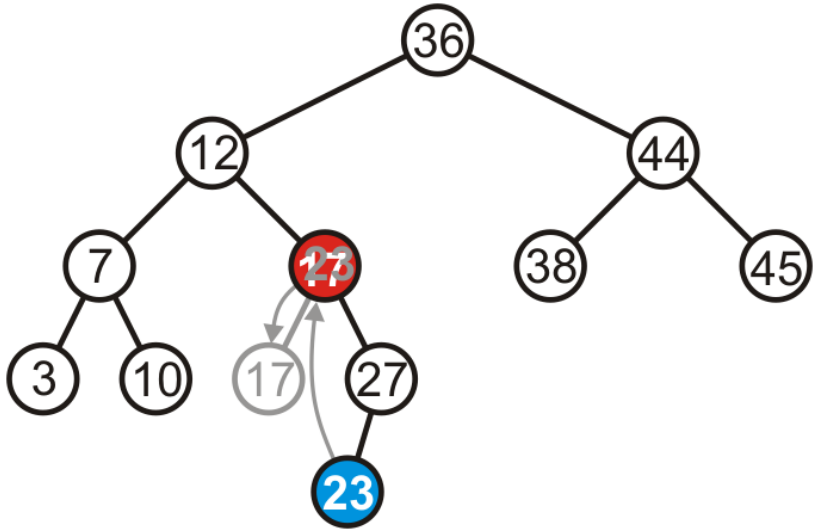
Duy trì sự cân bằng của cây (4)

Duy trì sự cân bằng tại nút 17

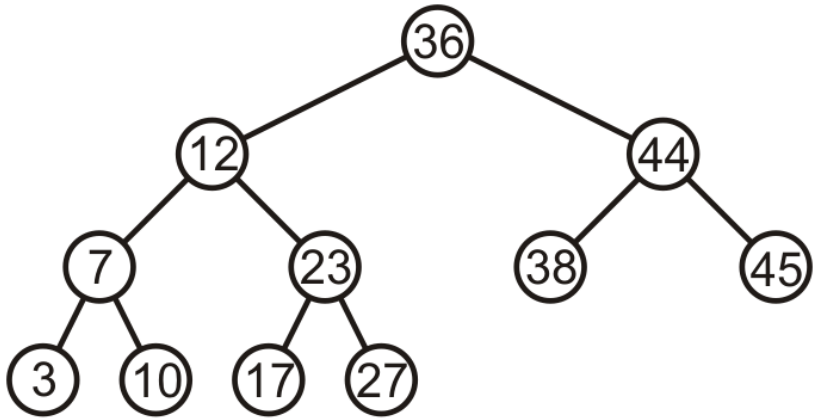


Duy trì sự cân bằng của cây (5)

Đổi nút 23 thế chỗ 17, biến 17 thành con trái của 23

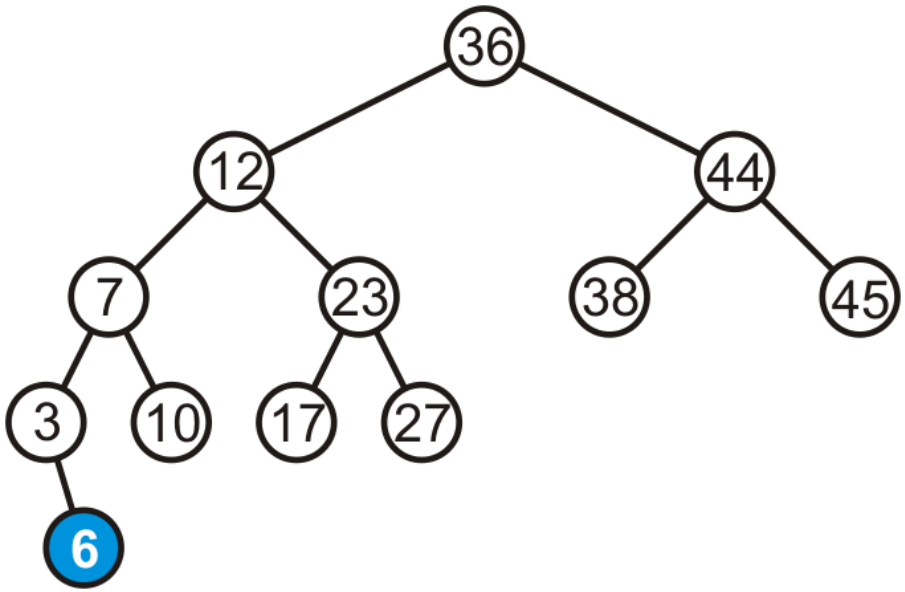


Nút mới đã cân bằng, ngẫu nhiên, nút gốc lúc này cũng cân bằng



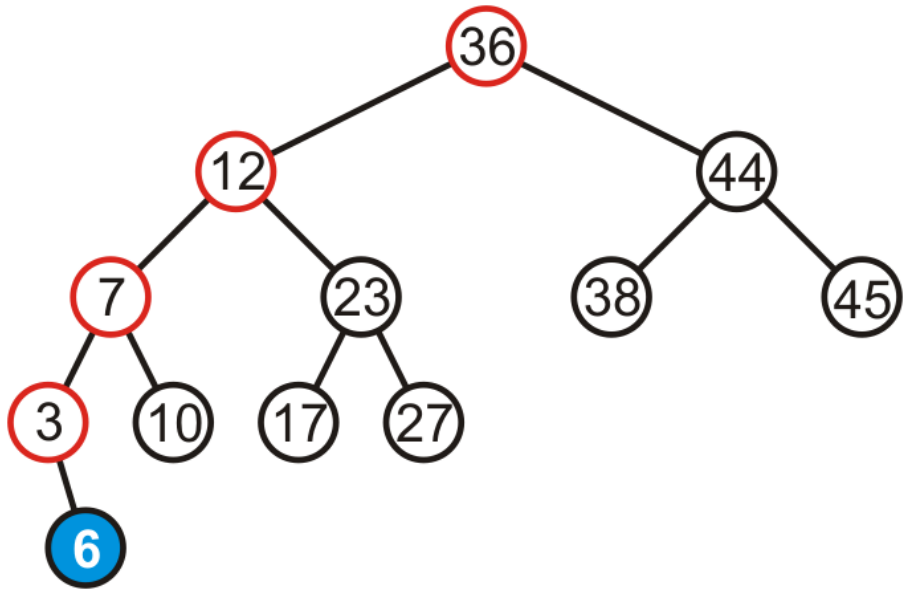
Duy trì sự cân bằng của cây (6)

Chèn thêm 6



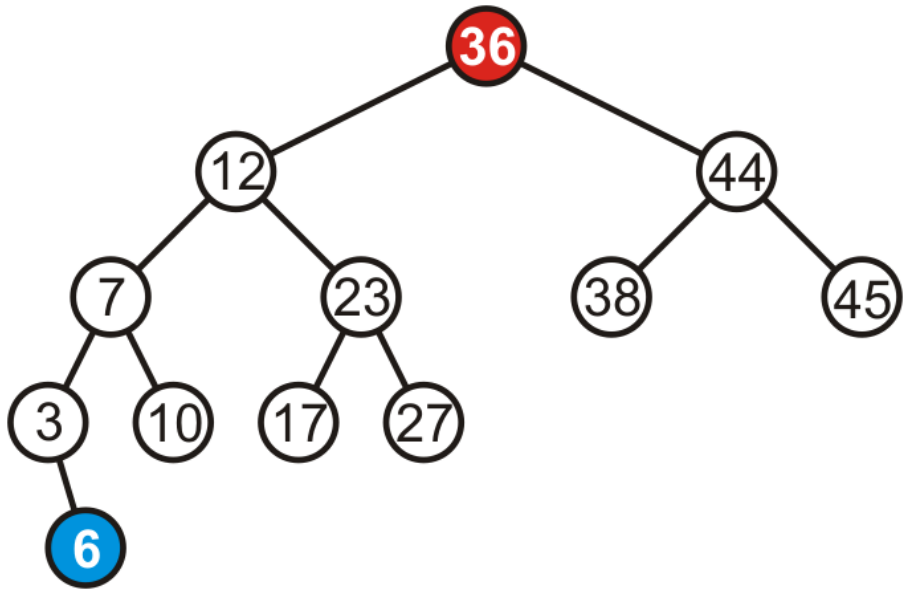
Duy trì sự cân bằng của cây (7)

Chiều cao của một số cây con thay đổi (tăng lên 1)



Duy trì sự cân bằng của cây (8)

Nút gốc mất cân bằng,...

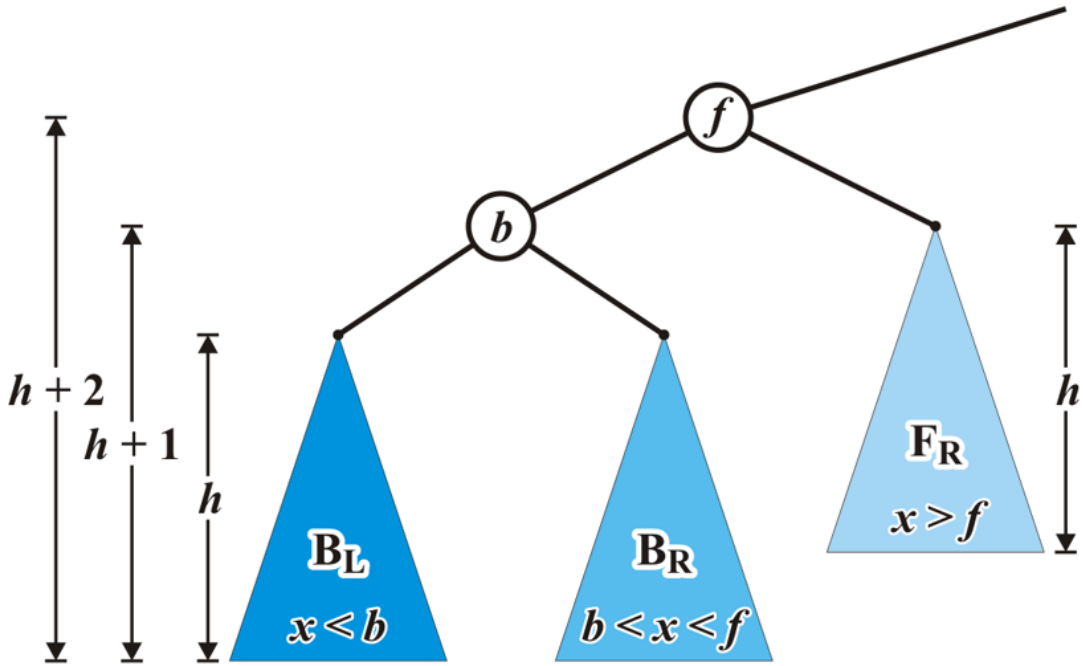


Để cân bằng cây, xét trường hợp tổng quát như sau

Duy trì sự cân bằng của cây

Trường hợp 1: LL

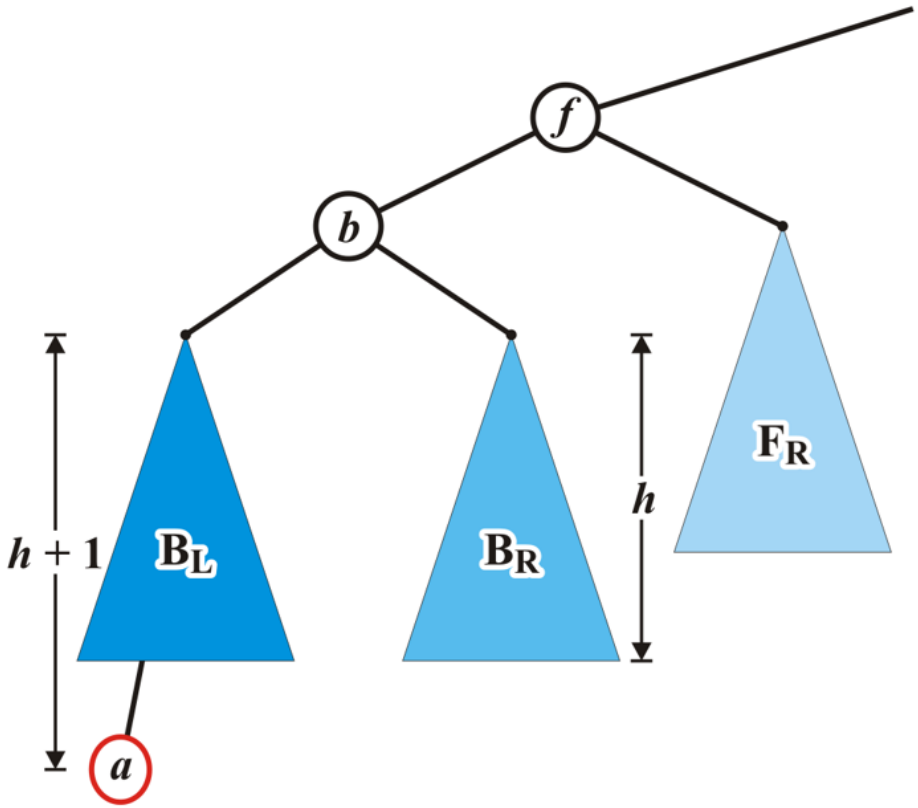
Mỗi hình tam tác màu xanh biểu diễn cho một cây có độ cao h



Duy trì sự cân bằng của cây

Trường hợp 1: LL

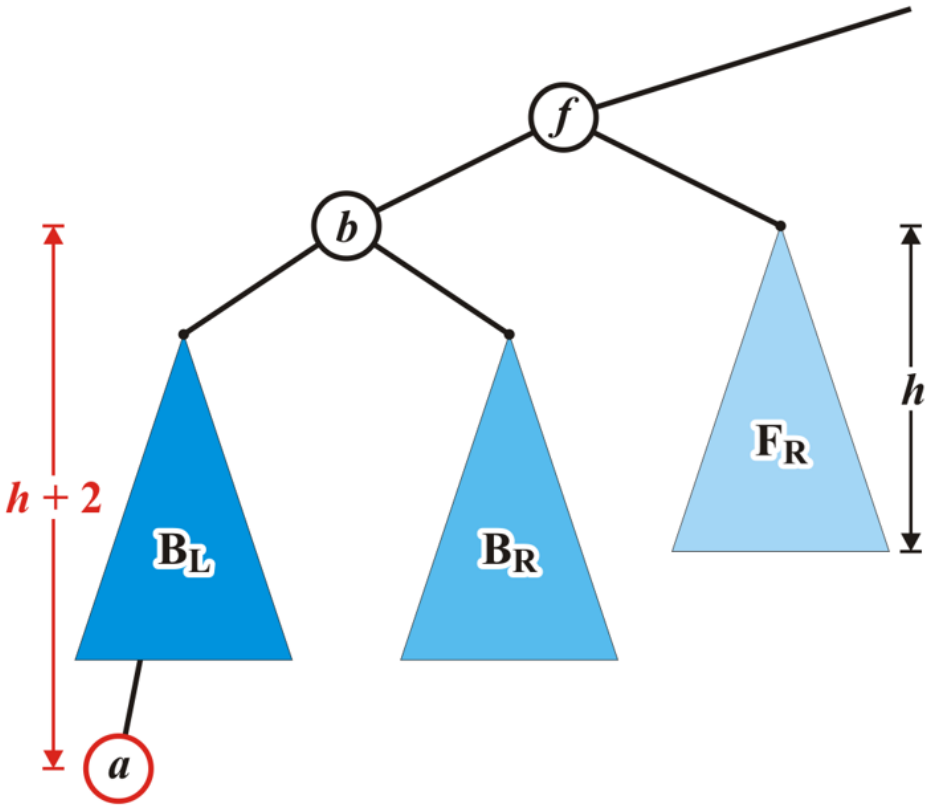
Chèn a vào cây: nếu a được chèn vào cây con trái của cây B_L của b. Nếu B_L cân bằng thì cây có gốc tại b cân bằng



Duy trì sự cân bằng của cây

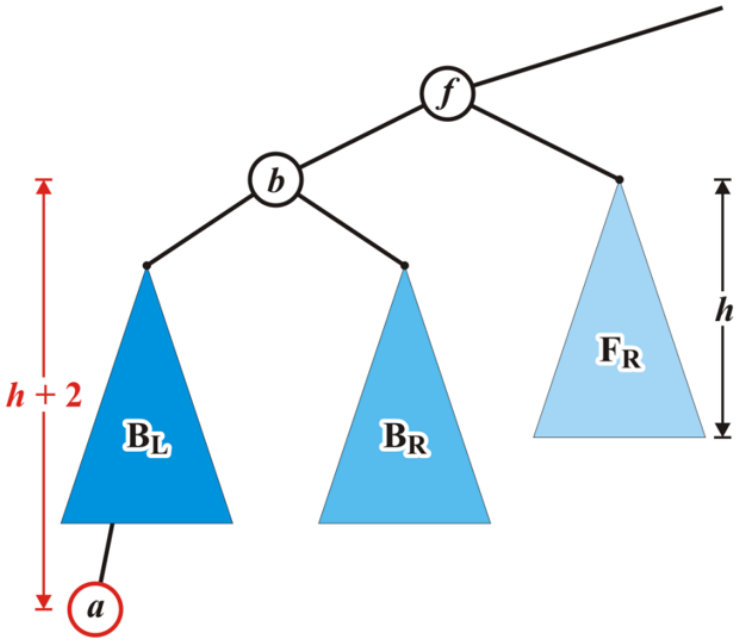
Trường hợp 1: LL

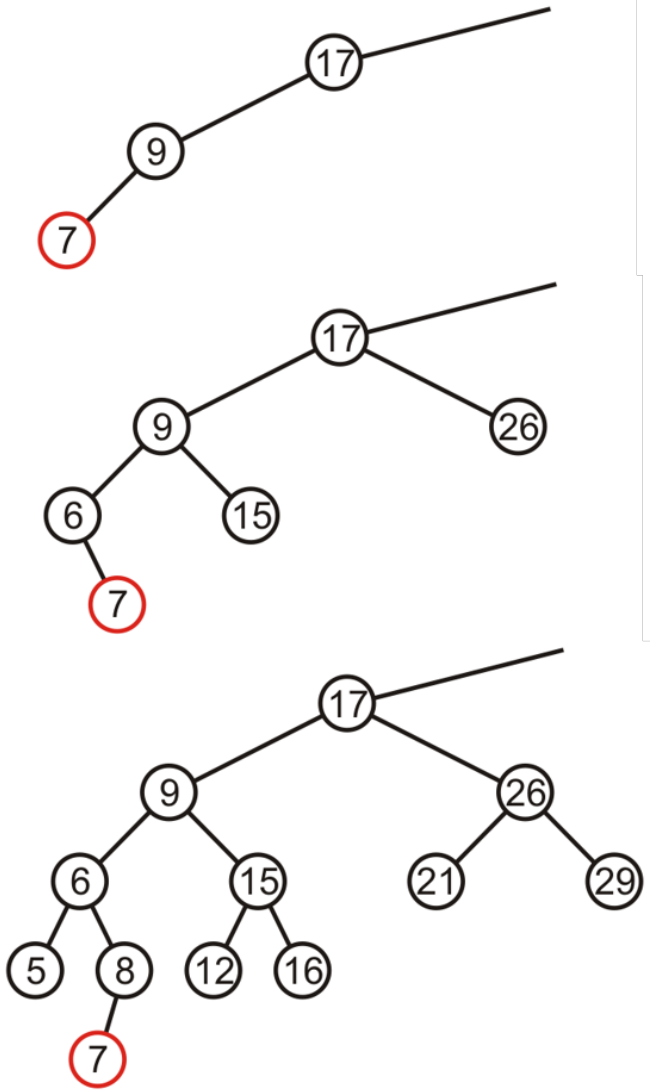
Cây có gốc tại f mất tính cân bằng. Cần phải duy trì sự cân bằng của cây có gốc tại f



Duy trì sự cân bằng của cây

Trường hợp 1: LL



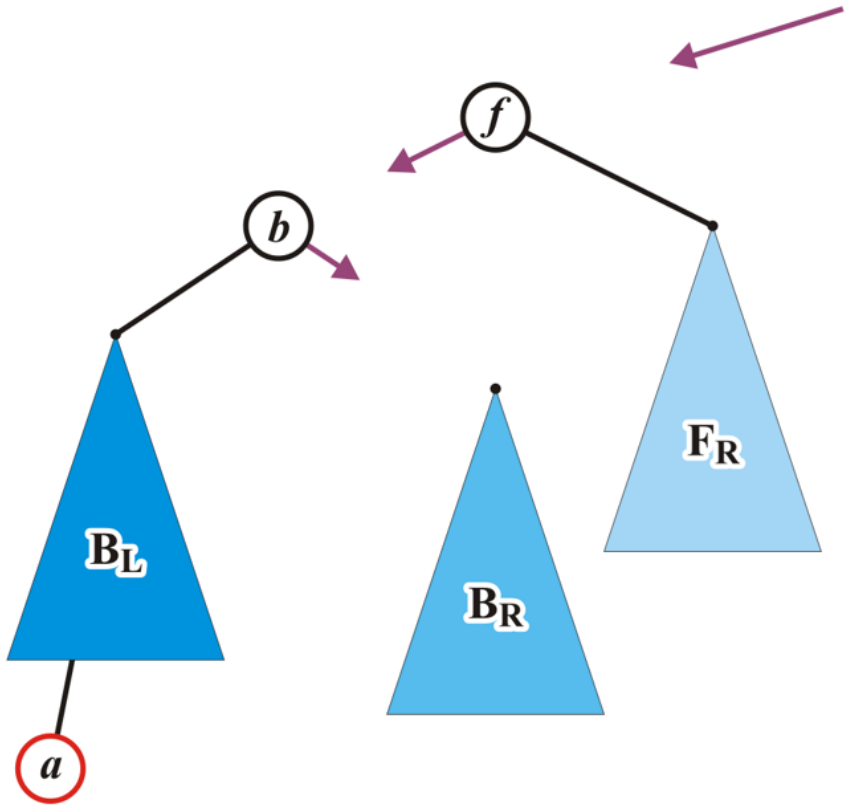


Duy trì sự cân bằng của cây

Trường hợp 1: LL

Cây có gốc tại f bị mất cân bằng

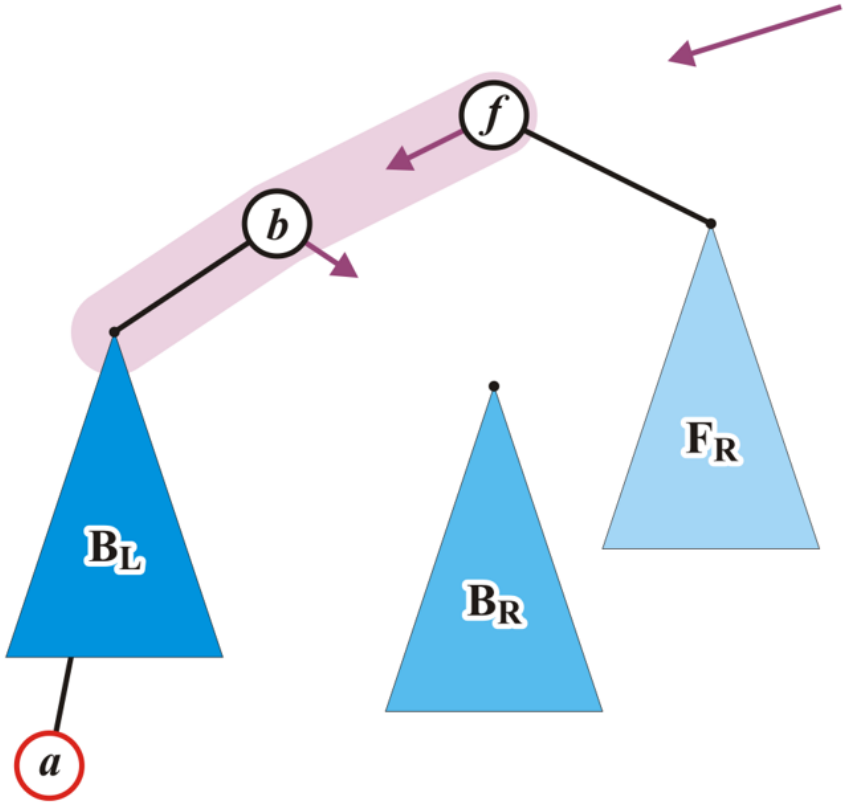
Cân bằng cây bằng cách thực hiện phép quay phải tại f



Duy trì sự cân bằng của cây

Trường hợp 1: LL

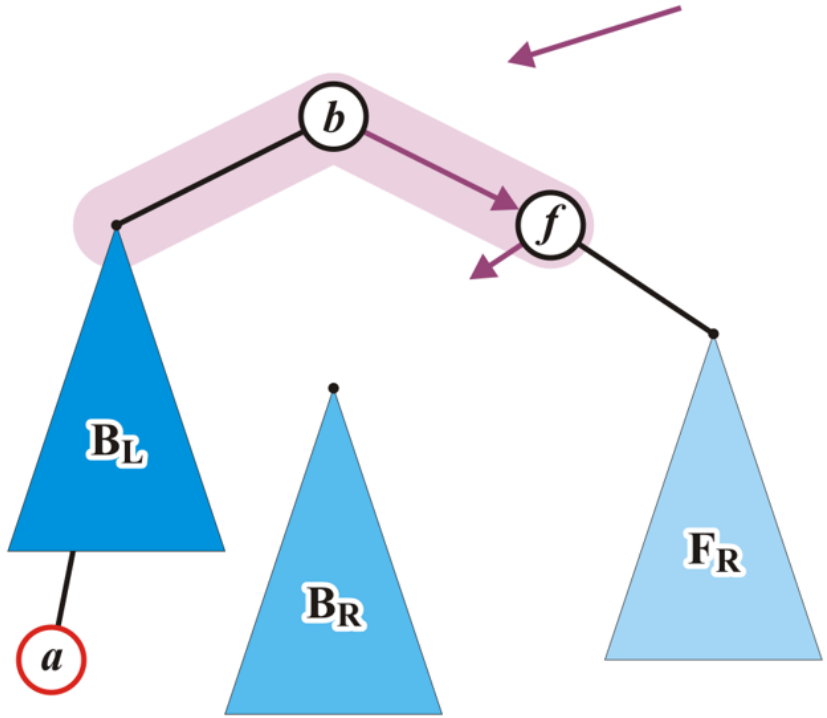
Lấy con phải của b



Duy trì sự cân bằng của cây

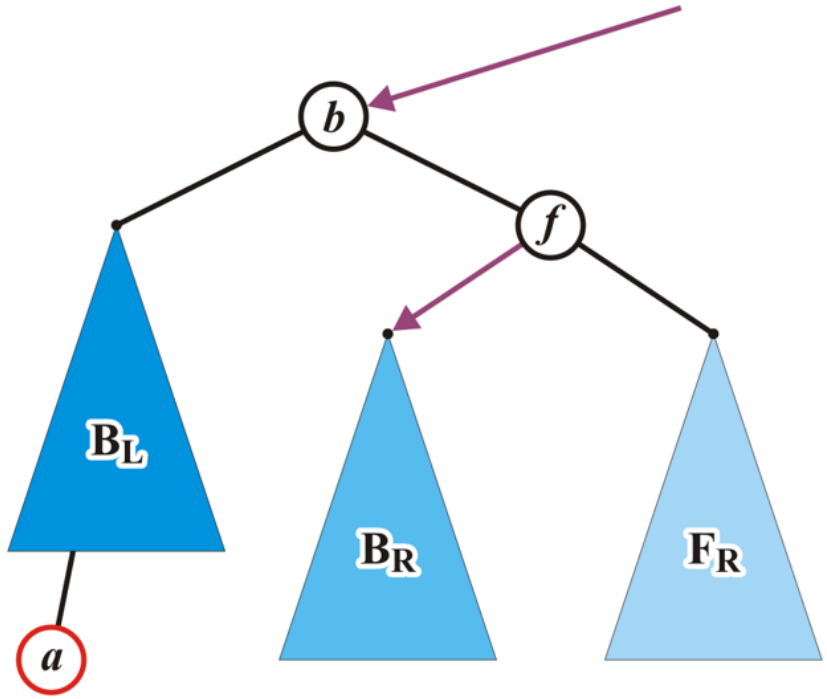
Trường hợp 1: LL

Để b lên làm gốc, f làm con phải của b



Duy trì sự cân bằng của cây

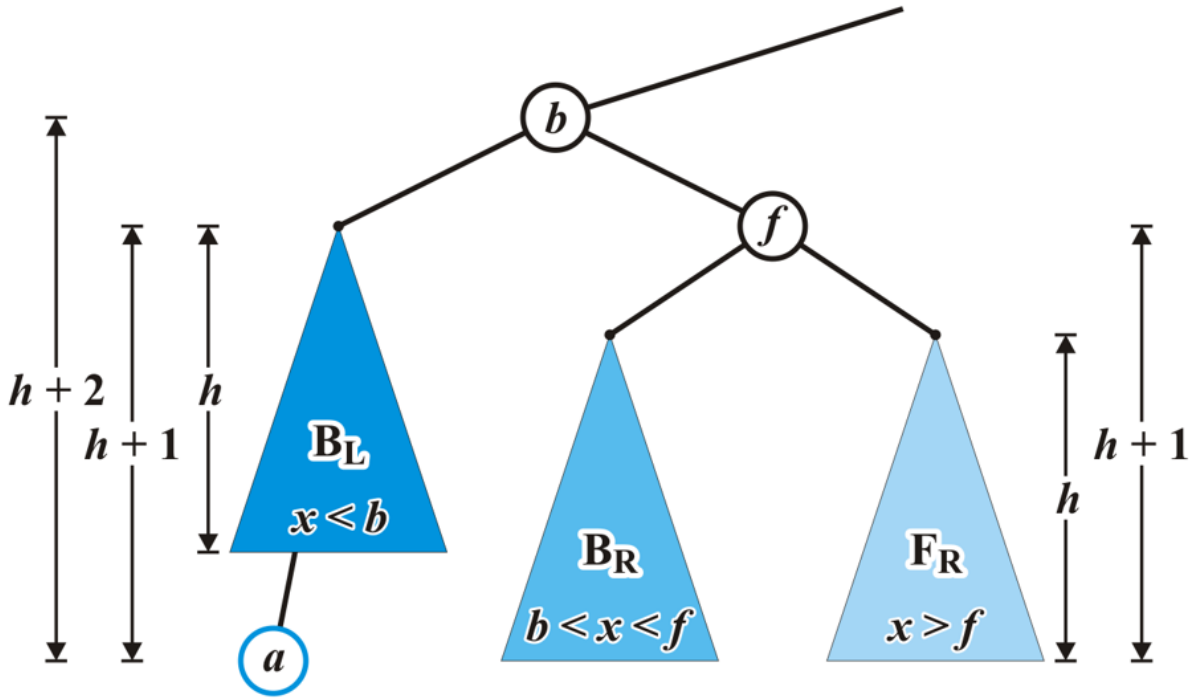
Trường hợp 1: LL



Duy trì sự cân bằng của cây

Trường hợp 1: LL

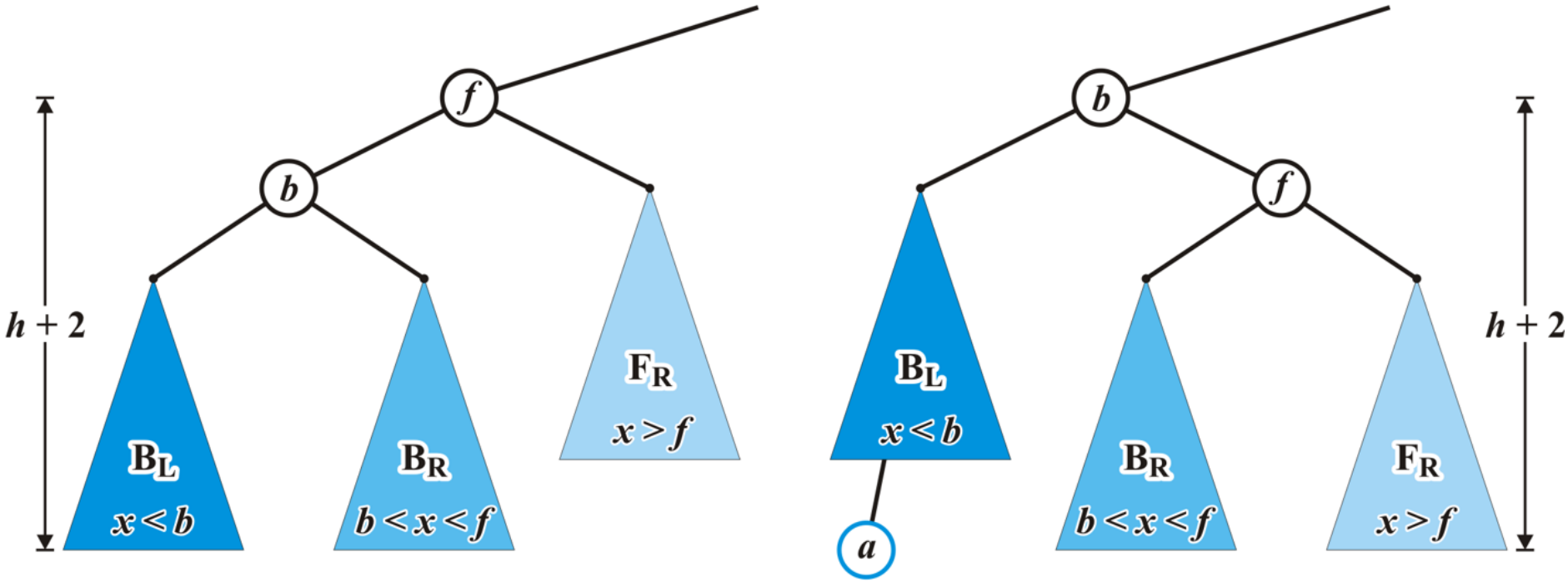
Lấy con phải cũ của b làm con trái của f



Duy trì sự cân bằng của cây

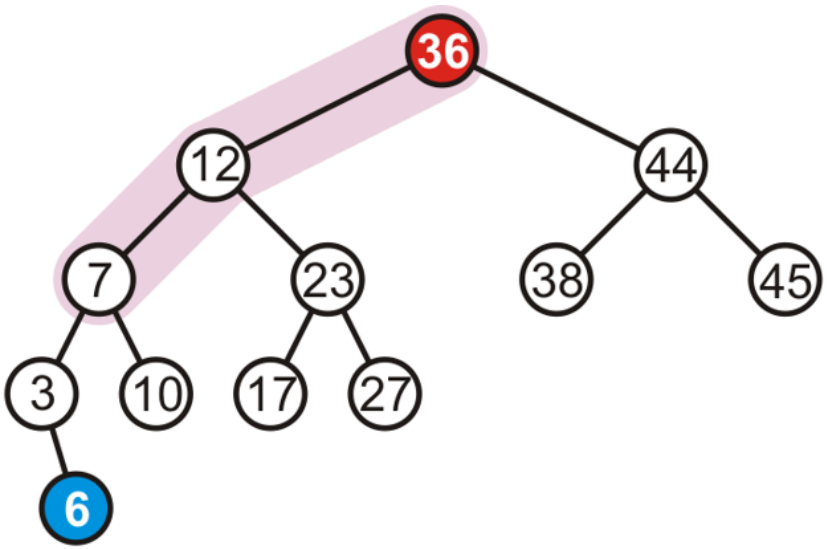
Trường hợp 1: LL

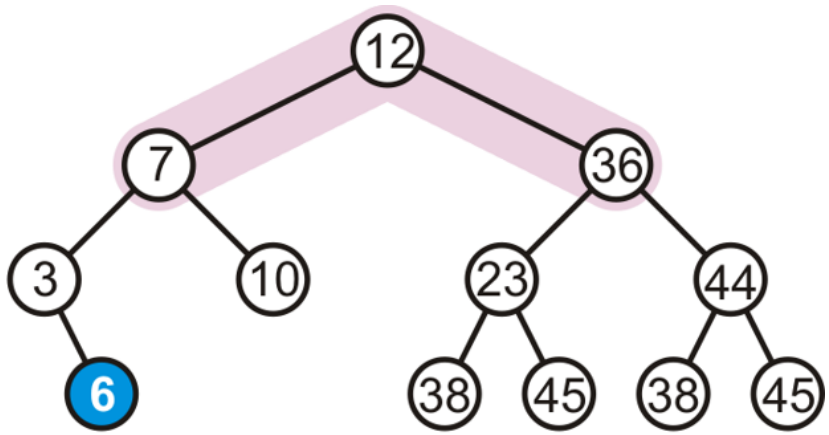
Chiều cao của cây có gốc tại b bằng chiều cao của cây con ban đầu có gốc tại f



Duy trì sự cân bằng của cây

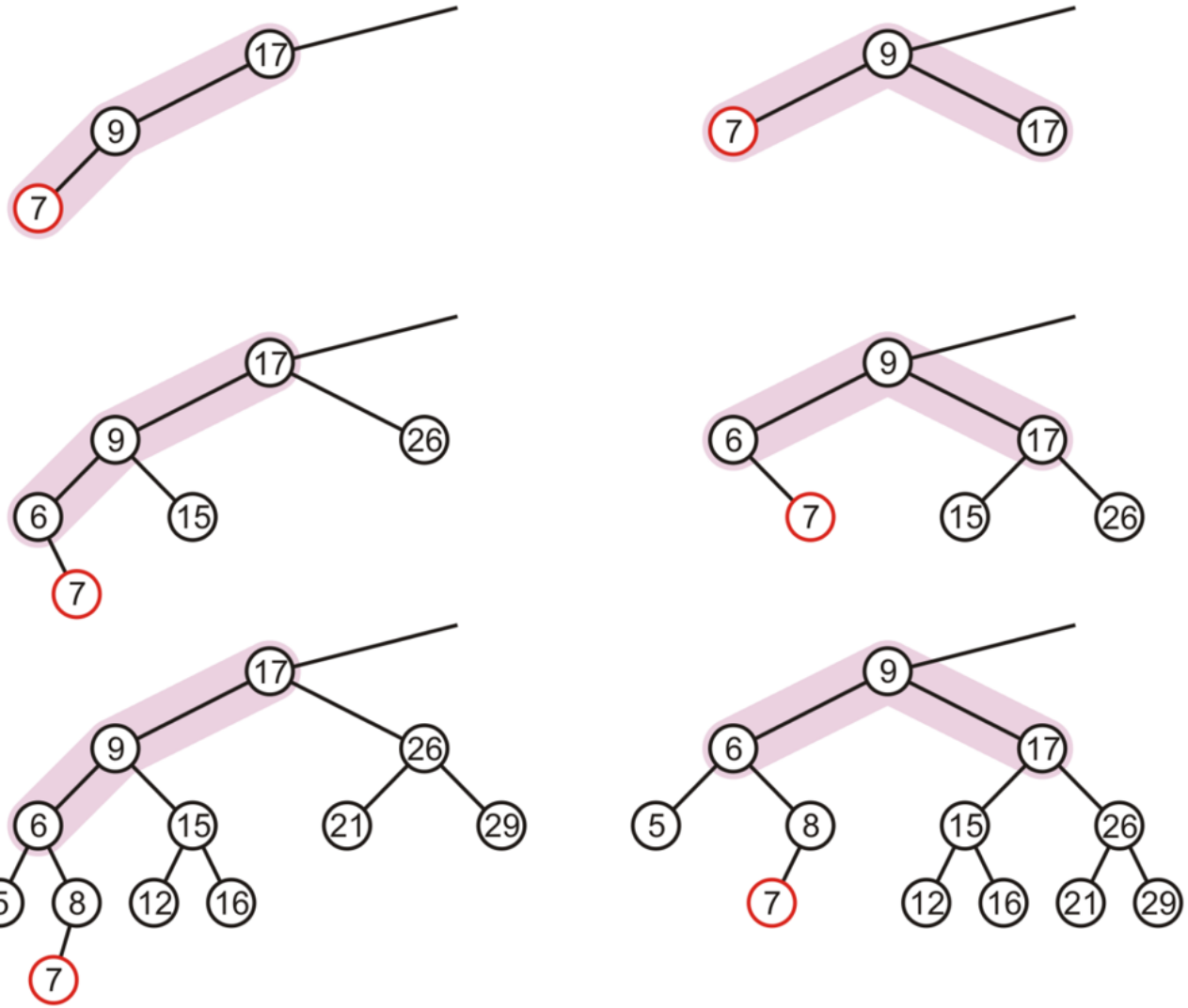
Ví dụ trường hợp 1





Duy trì sự cân bằng của cây

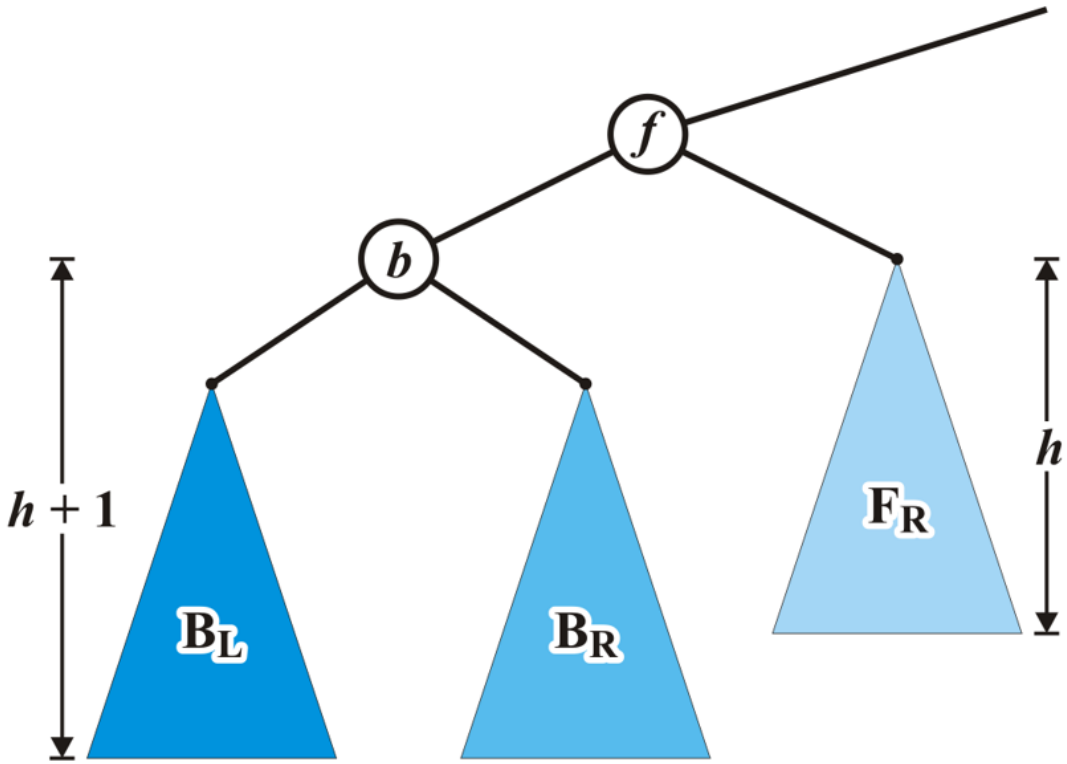
Ví dụ trường hợp 1



Duy trì sự cân bằng của cây

Trường hợp 2: LR

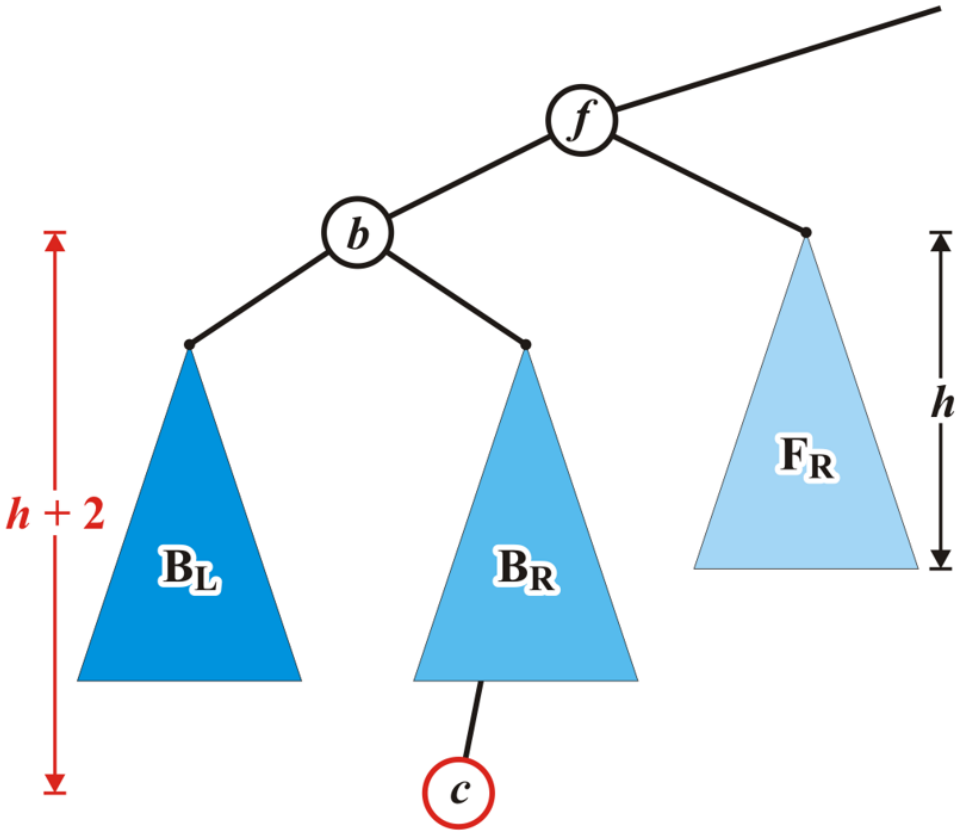
Cây ban đầu



Duy trì sự cân bằng của cây

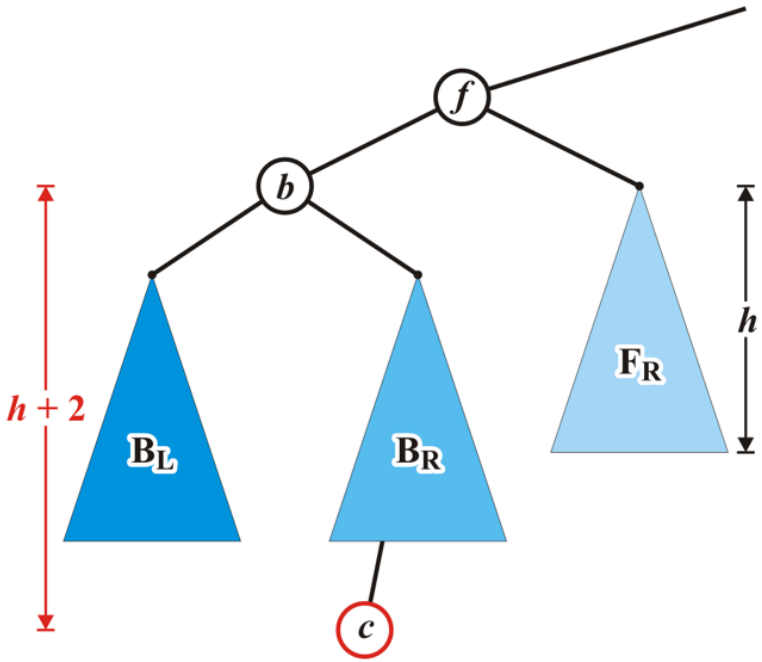
Trường hợp 2: LR

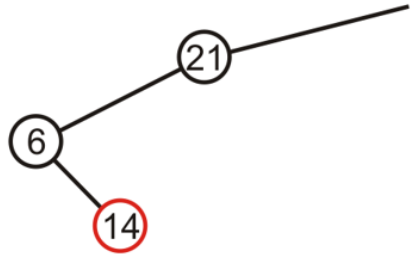
Xem xét trường hợp chèn c với $b < c < f$. Giả sử rằng nếu chèn c sẽ làm tăng chiều cao của cây con B_R. Lúc này f mất cân bằng

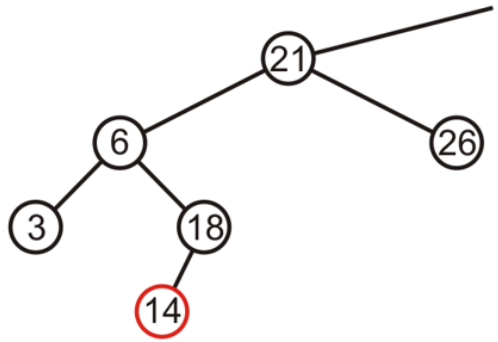


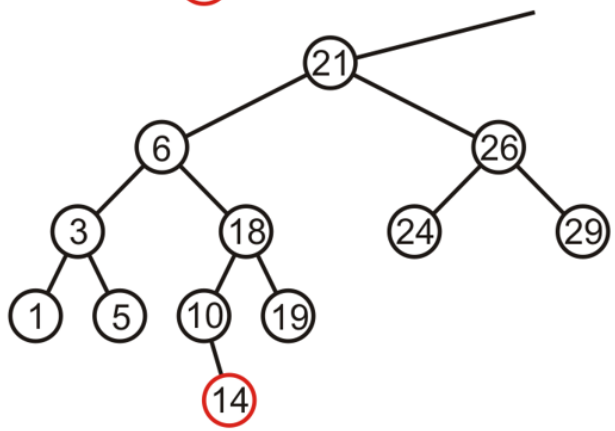
Duy trì sự cân bằng của cây

Trường hợp 2: LR



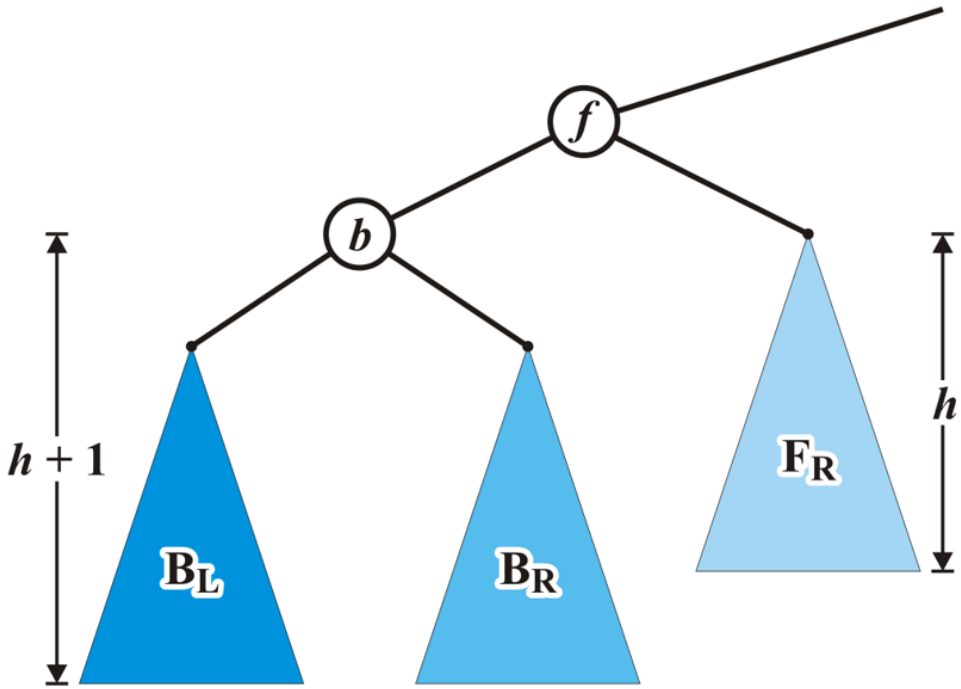






Duy trì sự cân bằng của cây

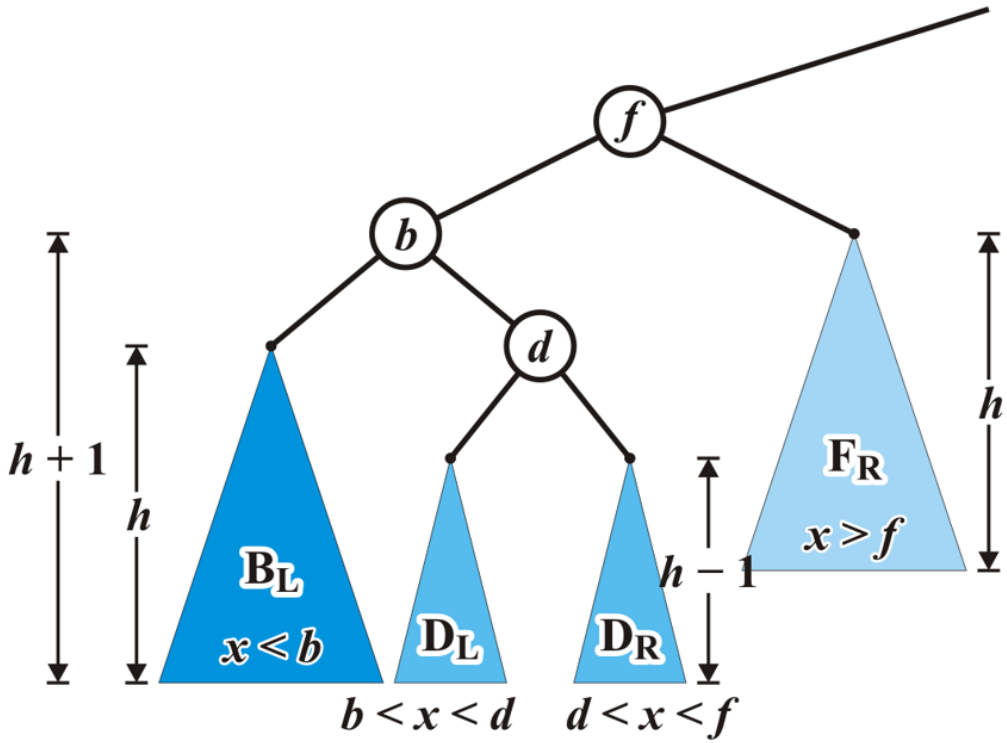
Trường hợp 2: LR



Duy trì sự cân bằng của cây

Trường hợp 2: LR

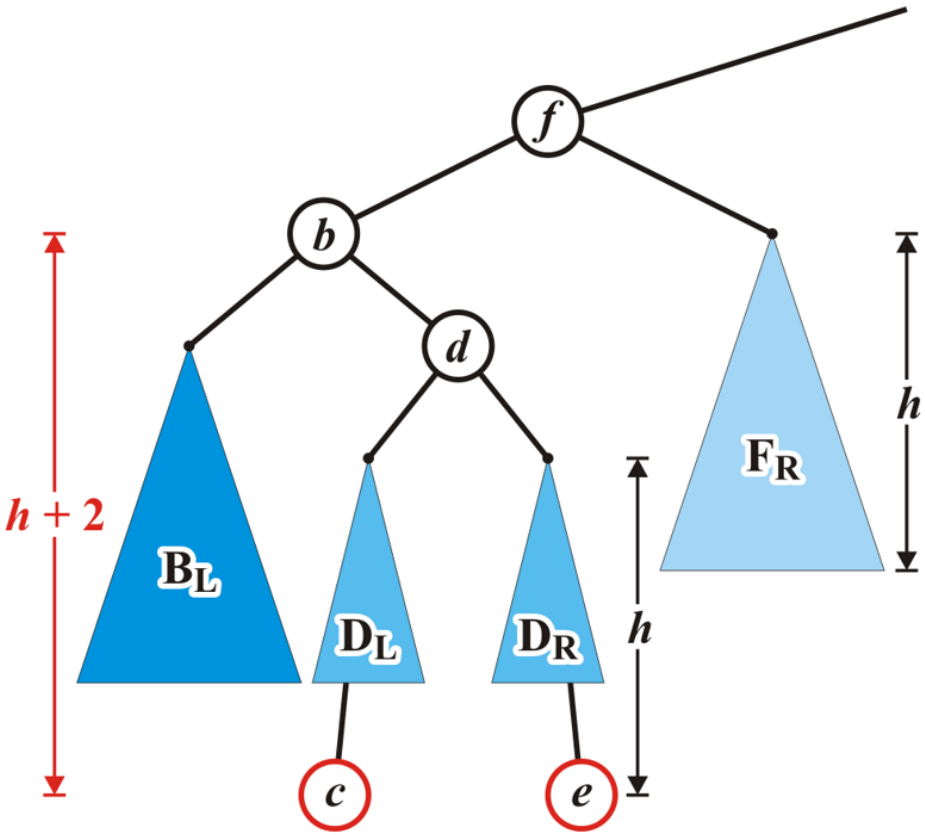
Gán lại nhãn cho cây bằng cách chia cây B_R thành cây có gốc tại d và với chiều cao $h-1$



Duy trì sự cân bằng của cây

Trường hợp 2: LR

Chèn vào cây sẽ gây ra mất cân bằng tại f

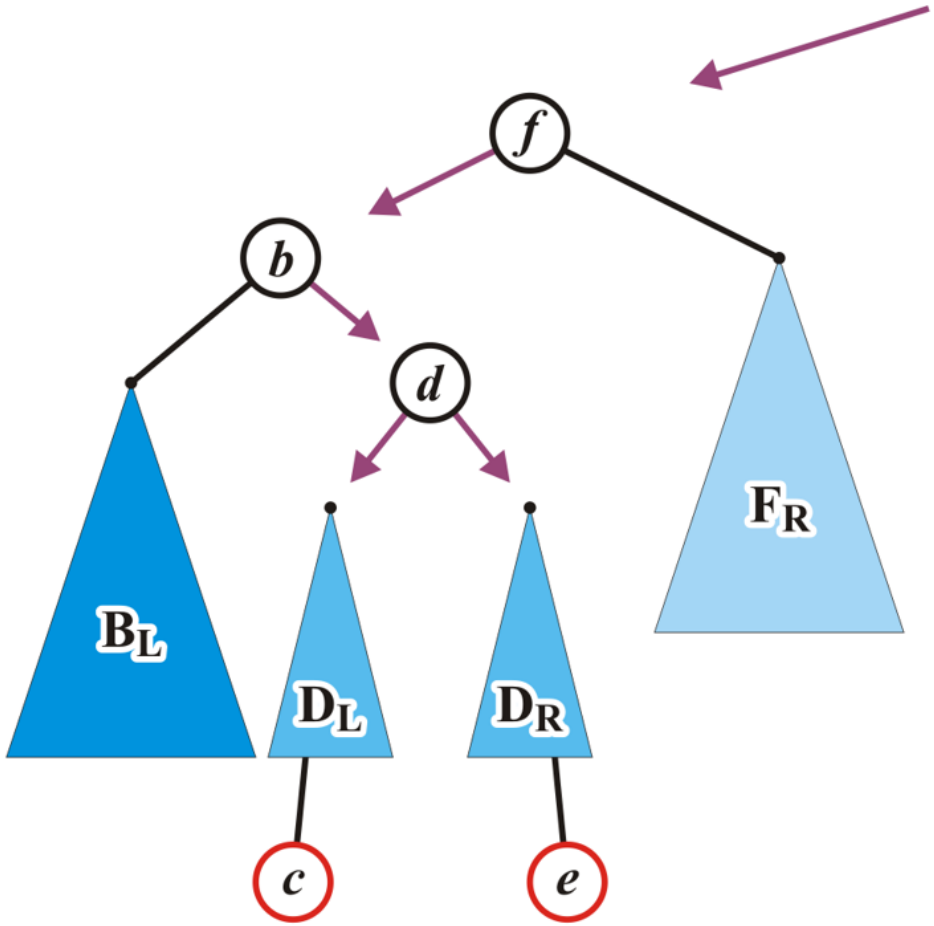


Duy trì sự cân bằng của cây

Trường hợp 2: LR

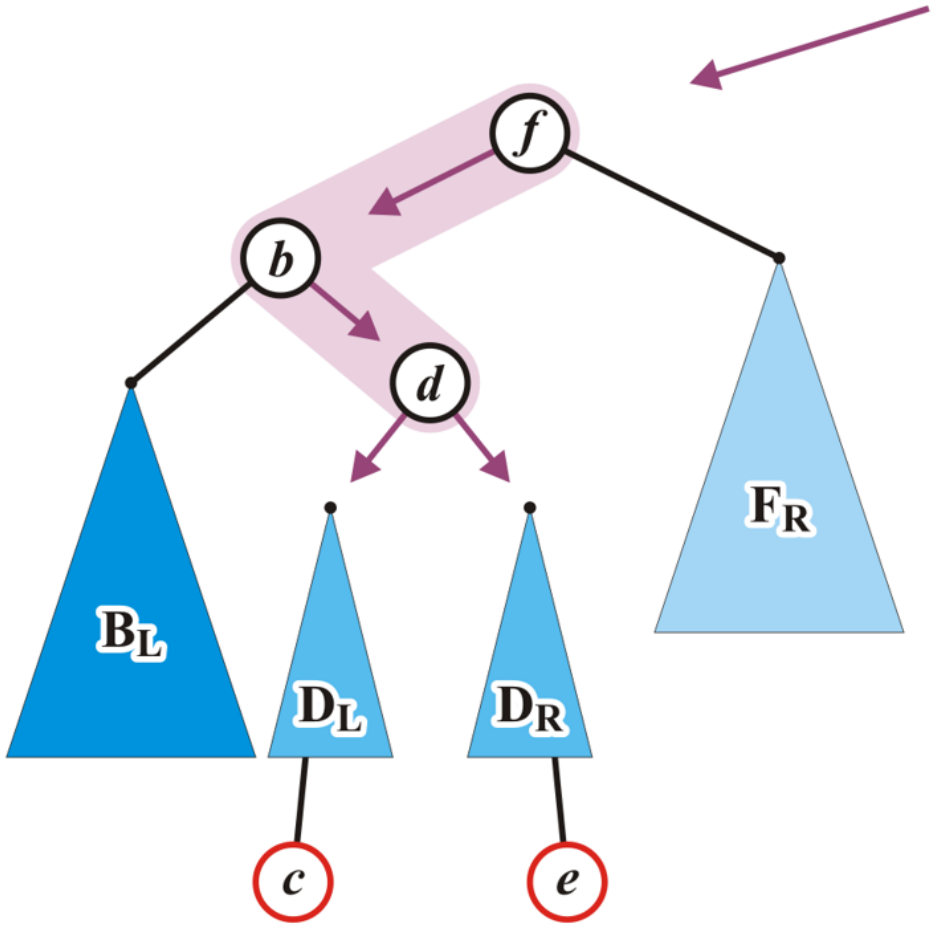
Cân bằng cây bằng cách thực hiện hai phép quay:

- Quay trái tại b
- Quay phải tại f



Duy trì sự cân bằng của cây

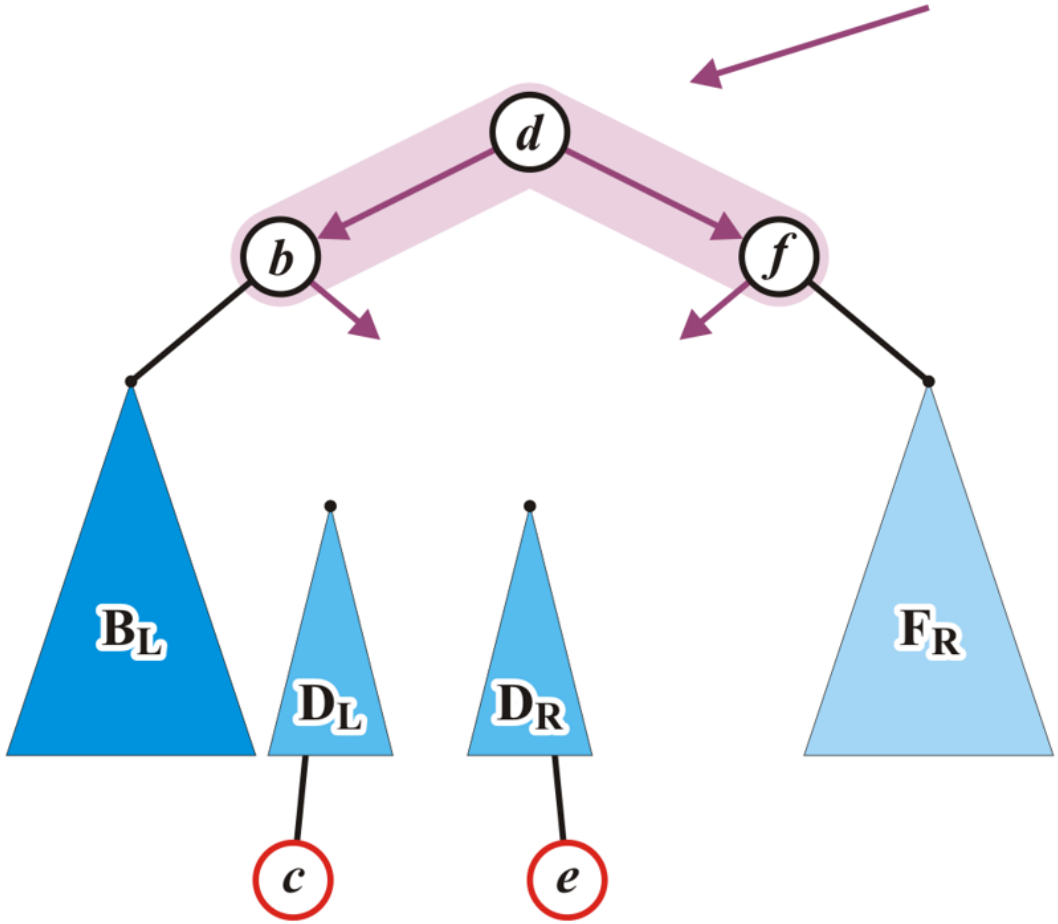
Trường hợp 2: LR



Duy trì sự cân bằng của cây

Trường hợp 2: LR

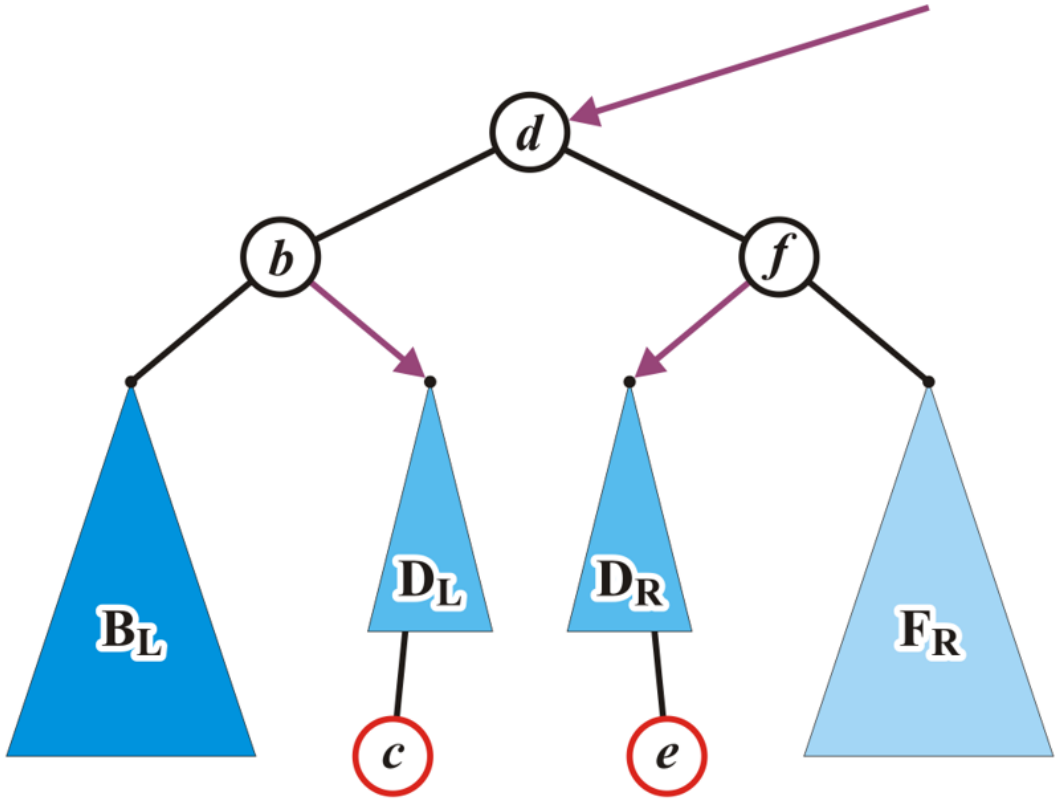
Để d làm gốc Lấy con trái và con phải của d



Duy trì sự cân bằng của cây

Trường hợp 2: LR

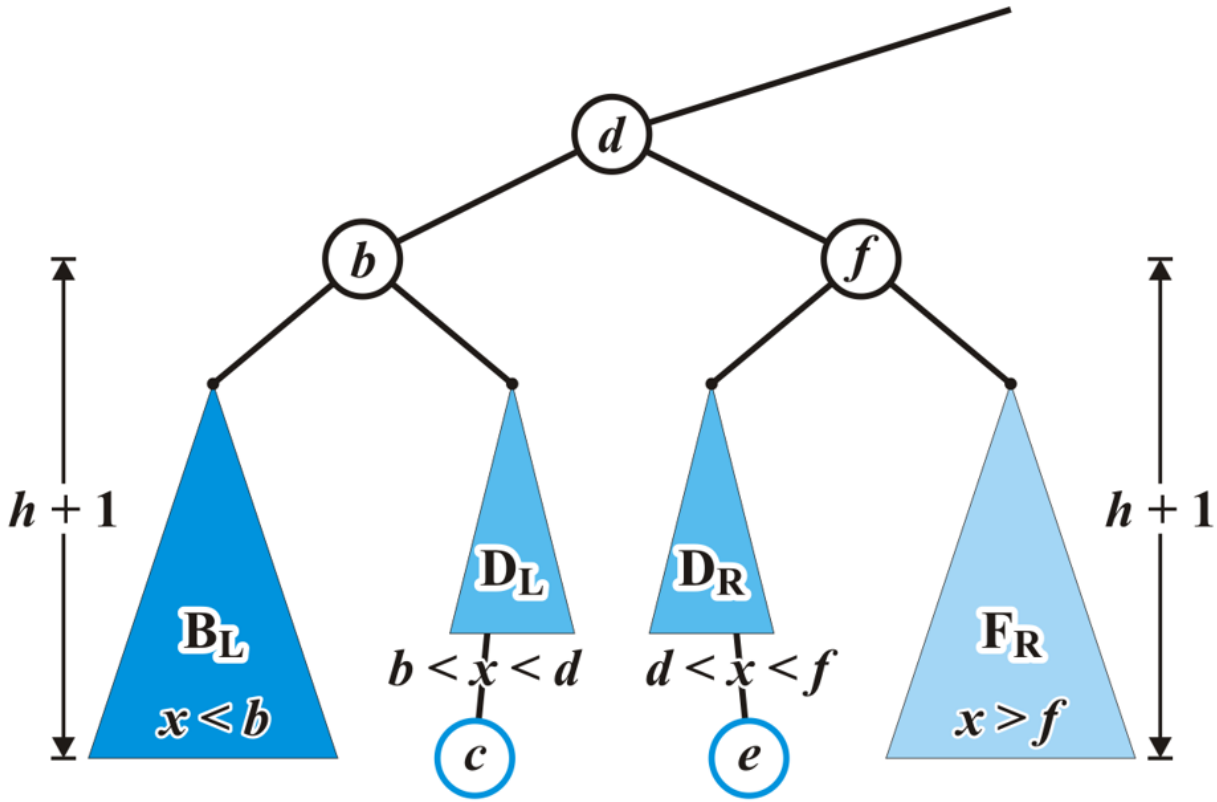
Để con trái của d làm con phải của b, để con phải của d làm con trái của f



Duy trì sự cân bằng của cây

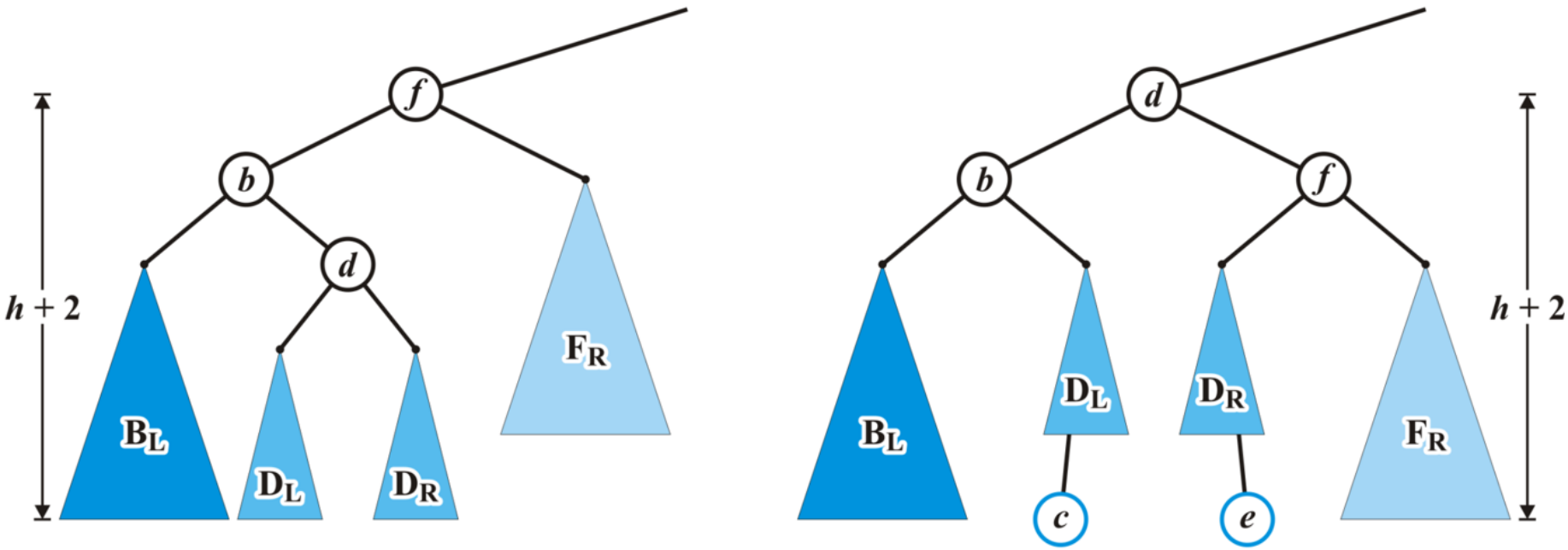
Trường hợp 2: LR

Cây có gốc tại d đã cân bằng



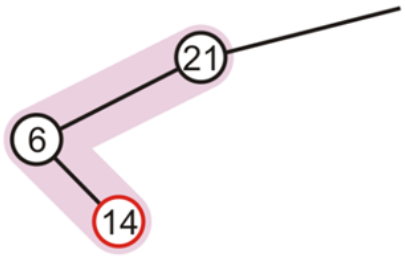
Duy trì sự cân bằng của cây

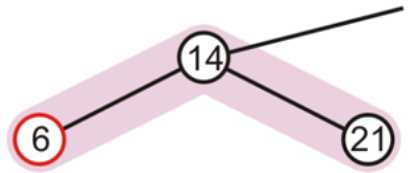
Trường hợp 2: LR

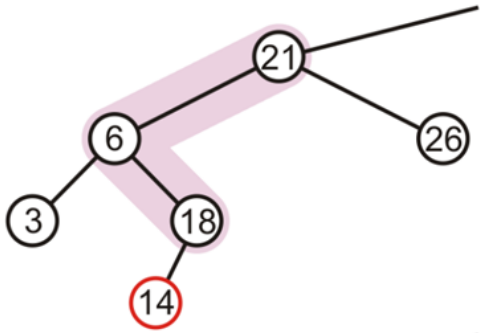


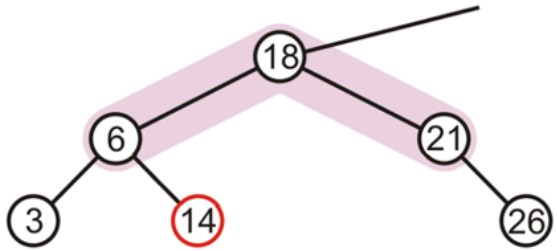
Duy trì sự cân bằng của cây

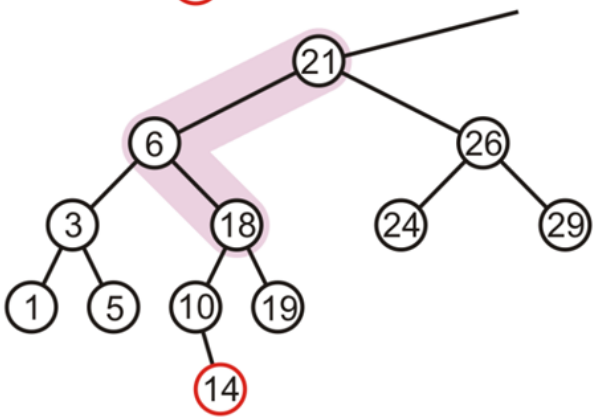
Ví dụ trường hợp 2

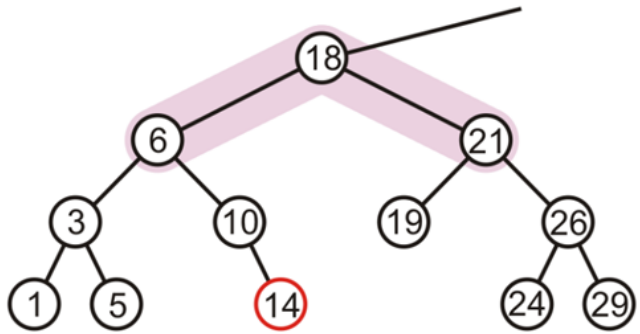






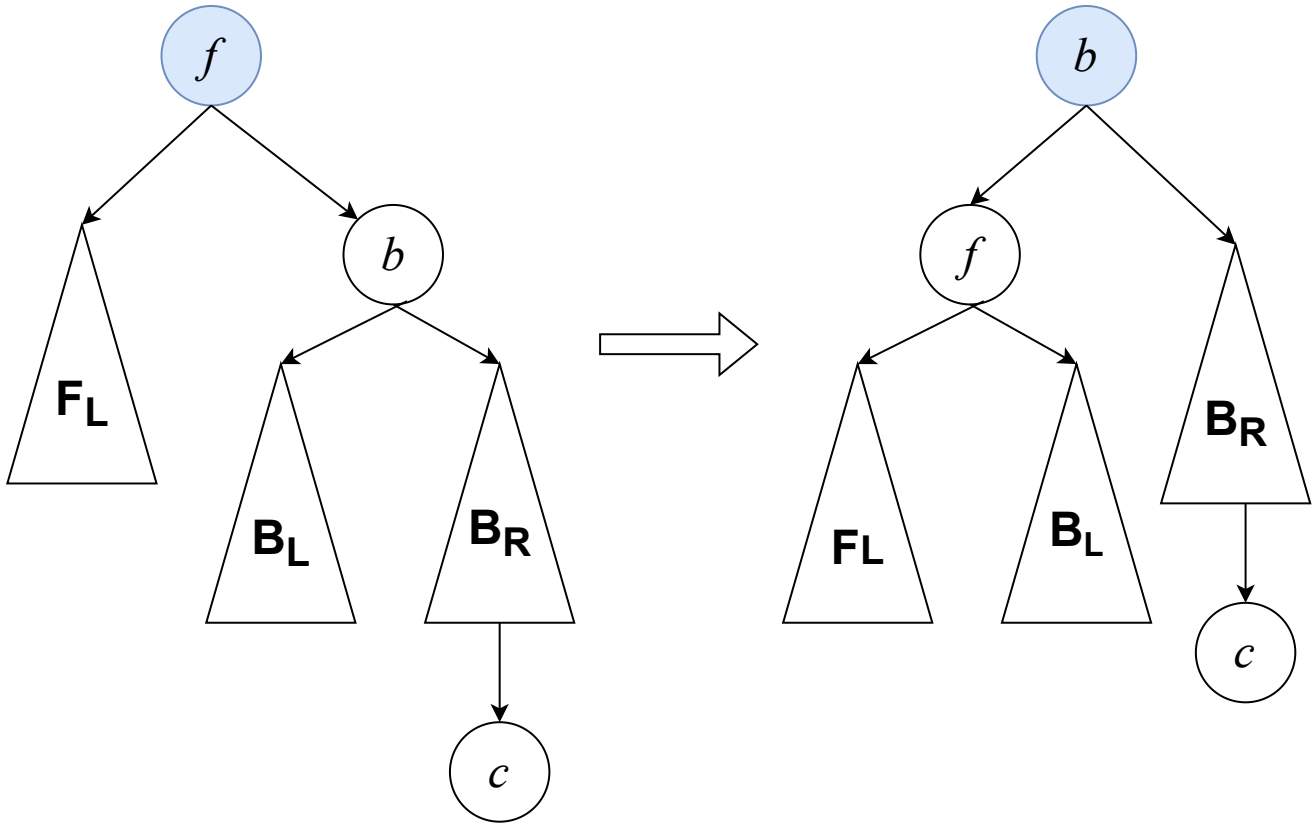






Duy trì sự cân bằng của cây

Trường hợp 4: RR

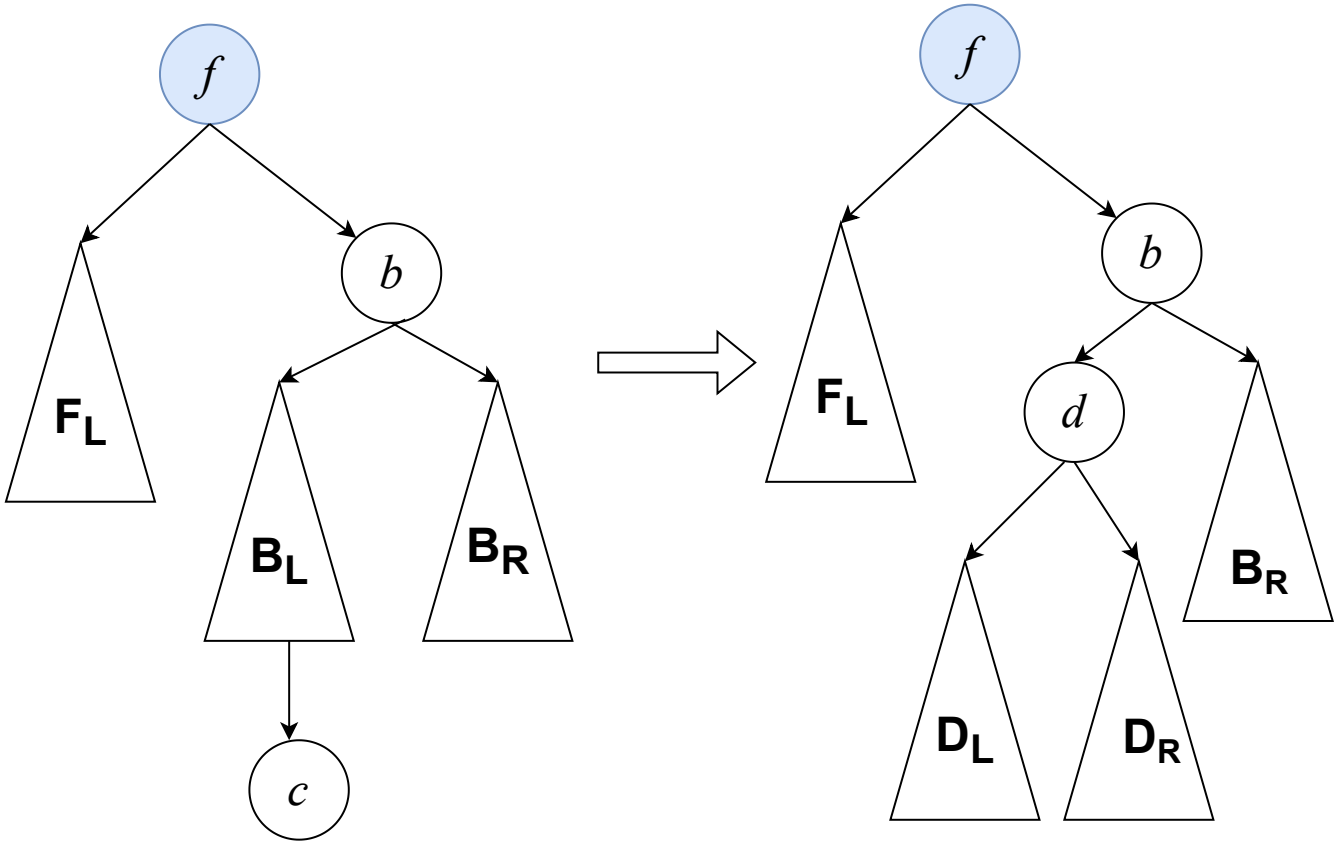


Hình 2: Quay trái quanh nút f

Duy trì sự cân bằng của cây

Trường hợp 4: RL

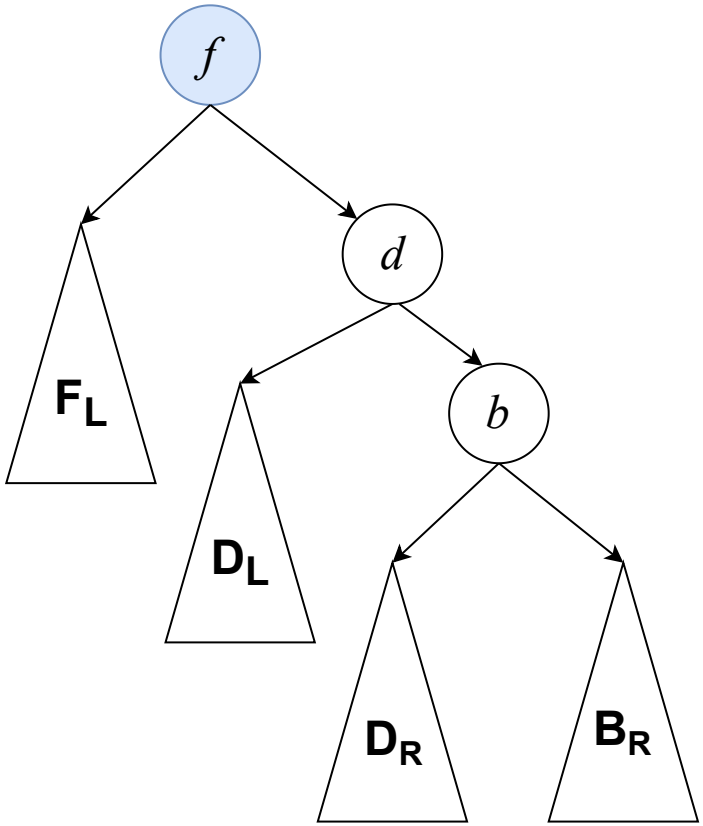
Cây ban đầu được vẽ lại như sau



Duy trì sự cân bằng của cây

Trường hợp 4: RL

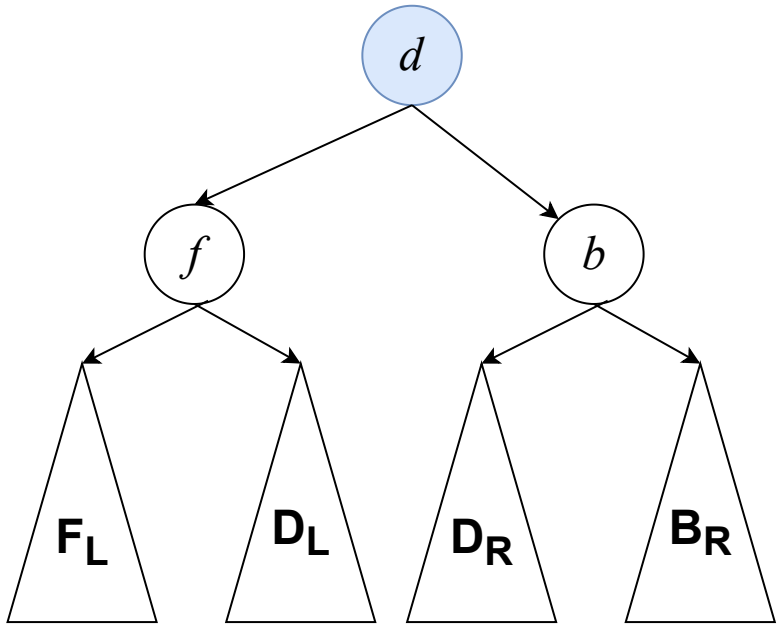
Quay phải quanh cây b



Duy trì sự cân bằng của cây

Trường hợp 4: RL

Quay trái quanh cây f



Ý tưởng thực hiện lập trình cài đặt cây AVL

- Dùng trường dữ liệu chiều cao (height) để xác định hệ số cân bằng cho nút cha
- Nút mới luôn có chiều cao là 1
- Khi chèn cũng như xoá một nút cần cập nhật giá trị chiều cao để phát hiện sự mất cân bằng

Các bước chèn

- Thực hiện chèn như cây BST bình thường
- Nút hiện tại sẽ là nút tổ tiên của nút mới k đang chèn, cập nhật lại chiều cao của nó
- Tính hệ số cân bằng mới của nút hiện tại (chiều cao cây con trái - chiều cao cây con phải)
- Nếu hệ số lớn hơn 1 sẽ có thể xảy ra hai tình huống
  - Tình huống 1
  - Tình huống 2
- Nếu hệ số nhỏ hơn -1 sẽ có thể xảy ra hai tình huống
  - Tình huống 3
  - Tình huống 4

Các bước chèn

```
public BSTNode insertToAVL(BSTNode root, int data) {
    /*1. chen*/
    if (root == null) {
        return new BSTNode(data);
    }
    if (data < root.data) {
        root.left = insertToAVL(root.left, data);
    } else if (data > root.data) {
        root.right = insertToAVL(root.right, data);
    } else {
        return root;
    }
    /* 2. cap nhat chieu cao */
    root.height = 1 + Math.max(height(root.left),
        height(root.right));
    /* 3. tinh toan he so can bang
    balance = height(root.left) - height(root.right);
   */
    int balance = calculateBalanceFactor(root);
```

Các bước chèn

```
        /* 4. kiem tra cac truong hop quay */
        if (balance > 1) {
            if (data > root.left.data) {
                root.left = rotateLeft(root.left);
            }
            return rotateRight(root);
        }
        if (balance < -1) {
            if (data < root.right.data) {
                root.right = rotateRight(root.right);
            }
            return rotateLeft(root);
        }

        return root;
    }
```

Nội dung

Đặt vấn đề

Cho bảng T và các bản ghi x với khoá và dữ liệu đi kèm, ta cần hỗ trợ các thao tác sau:

- Chèn: insert(T,x)
- Xoá: delete(T,x)
- Tìm kiếm: search(T,x)

Ta muốn thực hiện thao tác này một cách nhanh chóng mà không phải thực hiện việc sắp xếp các bản ghi. Bảng băm (**hash table**) là cách tiếp cận giải quyết vấn đề đặt ra

Chú ý

Ta chỉ xét các khoá là các số nguyên dương

Bảng băm

- Trong tình huống xấu nhất, việc tìm kiếm đòi hỏi thời gian $O(n)$ giống như tìm kiếm tuyến tính, nhưng trên thực tế bảng băm tìm kiếm hiệu quả hơn nhiều. Với một số giả thiết hợp lý, việc tìm kiếm phần tử trong bảng băm đòi hỏi thời gian $O(1)$
- Bảng băm có thể hiểu như sự mở rộng của mảng thông thường. Việc **địa chỉ hoá trực tiếp** trong mảng cho phép truy cập đến phần tử bất kỳ trong thời gian O(1)

Địa chỉ trực tiếp

Là một phương pháp lưu trữ và truy xuất dữ liệu dựa trên giá trị khóa (key) của chúng mà không cần ánh xạ phức tạp. Giả sử rằng:

- Các khoá là các số trong khoảng từ 0 đến $m-1$. Ví dụ, nếu $m = 10$, các khóa có thể là $0, 1, 2, ..., 9$.
- Mỗi khóa là duy nhất, không có hai phần tử nào trong tập hợp có cùng giá trị khóa. Ví dụ, nếu có phần tử khóa 3, thì không thể có một phần tử khác cũng mang khóa 3. Điều này đảm bảo không xảy ra xung đột khi lưu trữ.

Ý tưởng: thiết lập mảng $T[0...m-1]$, trong đó:

- $T[i] = x$ nếu $x \in T$ và $key[x] = i$
- $T[i] = null$ nếu trái lại

T được gọi là bảng địa chỉ trực tiếp (direct-address table) các phần tử trong bảng T sẽ được gọi là các ô.

Địa chỉ trực tiếp

Ví dụ

- $m = 5$, các khoá từ 0 đến 4
- Tập hợp dữ liệu $\{x_1, x_2, x_3\}$ với $key[x_1] = 2$, $key[x_2] = 0$, $key[x_3] = 4$
- Thiết lập bảng T
  - $T[0] = x_2$ vì $key[x_2] = 0$
  - $T[1] = null$ vì không phần tử nào có khoá là 1
  - $T[2] = x_1$ vì $key[x_1] = 2$
  - $T[3] = null$ vì không phần tử nào có khoá là 3
  - $T[4] = x_3$, vì $key[x_3] = 4$
- Truy xuất: Muốn tìm phần tử với khoá là 2? Chỉ cần tìm $T[2]$, trả về x_1

Đánh địa chỉ trực tiếp

Ưu điểm:

- Truy xuất nhanh: Thời gian tìm kiếm, chèn, xóa là $O(1)$ vì chỉ cần truy cập trực tiếp vào chỉ số tương ứng với khóa.
- Đơn giản: Không cần hàm băm hay xử lý xung đột.

Hạn chế:

- Tốn bộ nhớ: Nếu m lớn nhưng số phần tử thực tế ít, nhiều ô trong T sẽ là *null*, gây lãng phí không gian.
- Giới hạn khóa: Chỉ áp dụng khi khóa là số nguyên trong phạm vi nhỏ và xác định trước.

Cách giải quyết là ánh xạ khoá vào khoảng biến đổi nhỏ hơn m. Ánh xạ này dược gọi là hàm băm (***hash function***)

Bảng băm

Một số phương pháp thông dụng

- Cắt bỏ (Truncation) : dùng một phần của khóa làm chỉ số.
  - Ví dụ: khóa có 8 chữ số và bảng băm kích thước 1000 lấy chữ số thứ 4, 7 và 8 làm chỉ số 21296876 $\rightarrow$ 976
  - Hạn chế: nhanh nhưng phân bố khoá không đều
- 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 khoá k	**k*k**	**Địa chỉ**
5402	29181604	181
367	00134689	134
1246	01552516	552
2983	08898289	898

Bảng băm

Một số phương pháp thông dụng

- Gấp (folding): chia khóa thành nhiều phần sau đó kết hợp các phần lại (thường dùng cộng hoặc nhân)
  - Ví dụ: 21296876 chia thành 3 phần 212, 968 và 76 kết hợp 212+968+76 = 1256, cắt bỏ được 256
  - Hạn chế: giá trị các thành phần trong khóa đều ảnh hưởng tới chỉ số. Cho phân phối tốt hơn phương pháp cắt bỏ
- Phương pháp chia module: lấy số dư của phép chia giá trị khóa cho kích thước của bảng băm để làm đị chỉ

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

  - Thường chọn m là số nguyên tố nhỏ hơn, gần với kích thước bảng băm
  - Ví dụ: bảng băm kích thước 1000 thì chọn $m = 997$

Giá trị khoá	**Địa chỉ**
5402	417
367	367
1246	249
2983	989

Đụng độ

Đụng độ là gì?

Hai khóa khác nhau có cùng giá trị chỉ số

- Ví dụ: trong phương pháp cắt bỏ hai khóa 21296876, 11391176 có cùng chỉ số
- Giải pháp xử lý đụng độ
  - Phương pháp đánh địa chỉ mở (Open Addressing)
  - Phương pháp xích ngăn cách (Separate Chaining)

Phương pháp địa chỉ mở

Ý tưởng

Khi xảy ra xung đột, tìm một ô khác trong bảng T để lưu trữ dữ liệu của khoá đó. Tất cả dữ liệu dược lưu trữ trong T, không sử dụng cấu trúc phụ.

Cách thực hiện

Sử dụng một hàm thăm dò (probe function) để tìm ô trống tiếp theo khi xảy ra xung đột. Một số phương pháp thăm dò phổ biến:

- Dò tuyến tính (Linear Probing); Dò bậc hai (Quadratic Probing); Băm kép (Double Hashing)

Ví dụ

- Bảng băm T: có kích thước $m = 10$
- Hàm băm cơ bản $h(key) = key\%10$
- Dữ liệu đầu vào : Các khóa cần chèn là: 15,25,35,7,12,22,17
- Sau khi tính giá trị hàm băm: 15, 25, 35 đều ánh xạ vào ô $T(5)$; 7, 17 ánh xạ vào ô $T(7)$; 12, 22 ánh xạ vào ô $T(2)$

Phương pháp đánh địa chỉ mở

Dò tuyến tính

Ý tưởng

Khi xảy ra xung đột tại ô $h(key)$, thử các ô tiếp theo theo thứ tự tuyến tính: $h(key)+1, h(key)+2, \ldots$, cho đến khi tìm được ô trống. Nếu đến cuối bảng (ô *T[9]*) mà vẫn không tìm được ô trống, quay lại từ đầu (ô T[0]) – gọi là quấn vòng (wrap-around). Công thức: $h'(key) = (h(key)+i)\%m$

Ví dụ

Khoá	h(key)	Ô thử	Kết quả
15	5	T[5] trống	T[5] = 15
25	5	T[5] có 15, thử T[6] trống	T[6] = 25
35	5	T[5] có 15, T[6] có 25, thử T[7] trống	T[7] = 35
7	7	T[7] có 35, thử T[8] trống	T[8] = 7
...			

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

Ý tưởng

Khi xảy ra xung đột tại ô $h(key)$, thử các ô theo khoảng cách bậc hai $h(key) + 1^2, h(key) + 2^2, h(key) + 3^2, ...$ Công thức $h'(key) = (h(key) + i^2)\%m$

Ví dụ

Khoá	h(key)	Ô thử	Kết quả
15	5	T(5) trống	T[5] = 15
25	5	T[5] có 15, thử T[6] trống	T[6] = 25
35	5	T[5] có 15, T[9] trống	T[9] = 35
7	7	T[7] trống	T[7] = 7
...			

Phương pháp đánh địa chỉ mở

Dò bậc hai, dò toàn phương

- Giảm được sự phân nhóm
- Không phải tất cả các vị trí trong bảng đều được dò. Ví dụ, 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

Băm kép

Ý tưởng

$h(key) = (h_1(key) + i \cdot h_2(key))\%m$ trong đó, $h_1(key)$ và $h_2(key)$ là hàm băm bổ trợ; $i = 0, 1, ..., m - 1$

Phương pháp đánh địa chỉ mở

Một số cách dò khác

- Dò theo khoá: trong trường hợp đụng độ tại vị trí 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 2918160 xảy ra đụng độ thì dò tại ô tiếp theo cách ô đụng độ 2 vị trí
- 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

Chú ý: không thể xóa phần tử trong bảng băm sử dụng phương pháp đánh địa chỉ mở theo cách thông thường $\rightarrow$ đánh dấu là xóa, nhưng vẫn được xét đến khi dò

Phương pháp tạo chuỗi

Ý tưởng

- Khi xảy ra xung đột, thay vì tìm một ô khác để lưu trữ (như phương pháp địa chỉ mở), ta lưu tất cả các khoá ánh xạ vào ô đó trong một danh sách liên kết hoặc một cấu trúc dữ liệu tương tự
- Mỗi ô trong bảng băm T không chỉ chứa một phần tử mà là một danh sách các phần tử có cùng giá trị hàm băm

Cách hoạt động

- Khi chèn một khoá: tính chỉ số $i = h(key)$. Thêm khoá *key* vào danh sách liên kết tại $T[i]$ (thêm vào đầu hoặc cuối)
- Tìm kiếm: tính chỉ số $i = h(key)$. Duyệt dach sách liên kết $T[i]$ để tìm khoá *key*. Nếu tìm thấy, trả về phần tử; nếu không, trả về **null**