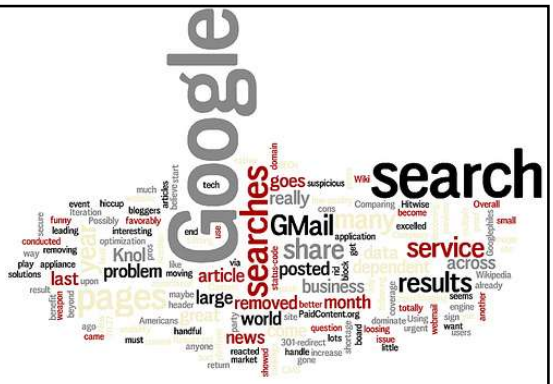
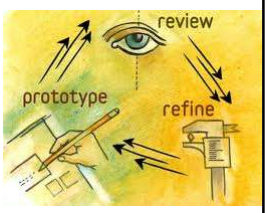


Chương 4.
Tìm kiếm (tiếp)

nguyenduyhiep@gmail.com
hiepnd@soict.hut.edu.vn

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

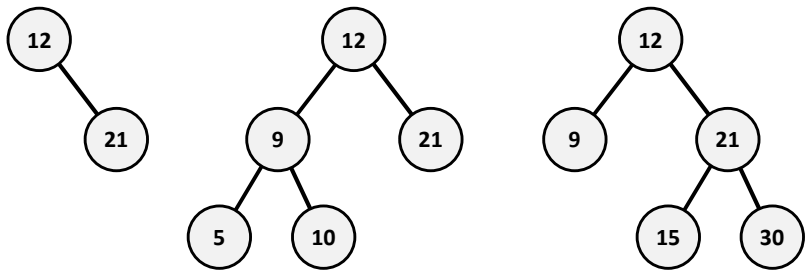


Cây tìm kiếm nhị phân cân bằng AVL Tree

G. M. ADELSON-VELSKII và E. M. LANDIS

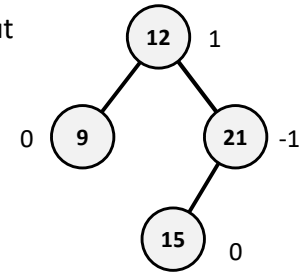
AVL tree - (Adelson-Velsky and Landis Tree)

- 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)
  - Hoặc đơn giản lưu độ cao của nút
- 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
{
    int data;
    Balance_factor balance;
    struct TreeNode* left;
    struct TreeNode* right;
} AVLNode;

AVLNode* createNode(int key) {
    AVLNode* node = (AVLNode*)malloc(sizeof(AVLNode));
    node->key = key;
    node->left = node->right = NULL;
    node->balance = equal_height;
    return node;
}
```

AVL tree

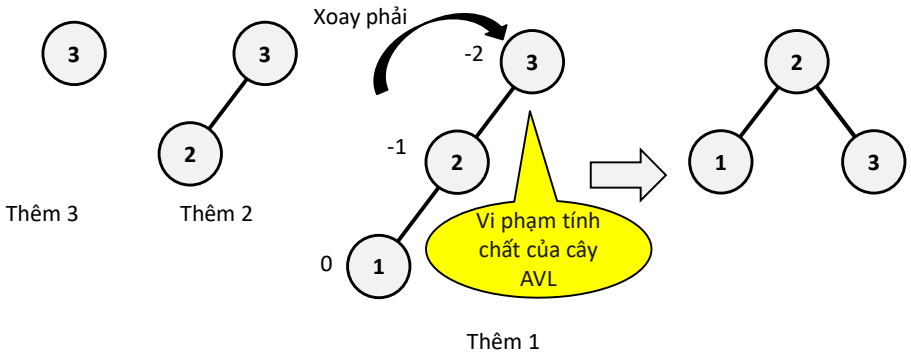
```
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
} AVLNode;
```

```
AVLNode* createNode(int key) {
    AVLNode* node = (AVLNode*)malloc(sizeof(AVLNode));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 0;
    return node;
}
```

- Lưu thông tin chiều cao của nút
  - Sẽ tốn bộ nhớ hơn lưu trạng thái cân bằng (2 bit)
  - Thao tác xử lý cây liên quan đến chiều cao, độ sâu nút sẽ nhanh hơn
  - Tràn số nếu chiều cao cây quá lớn ?

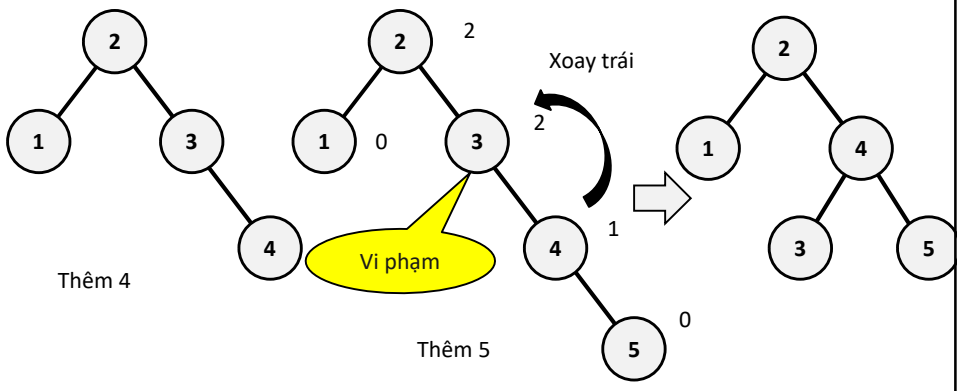
AVL tree

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



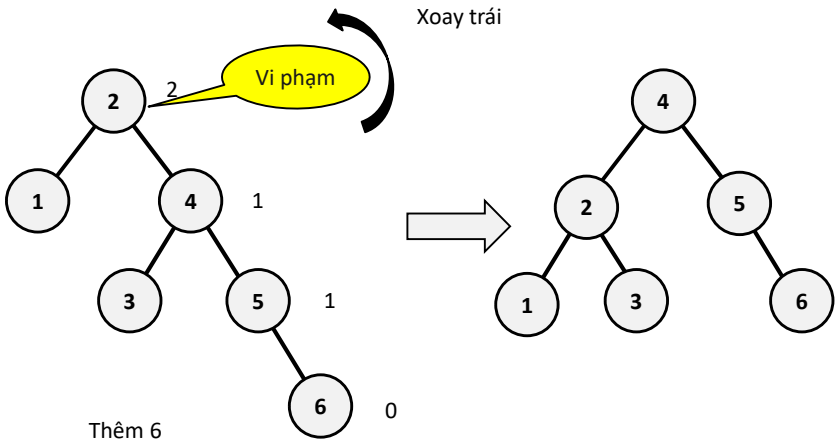
Xử lý bằng phép xoay nút

AVL tree

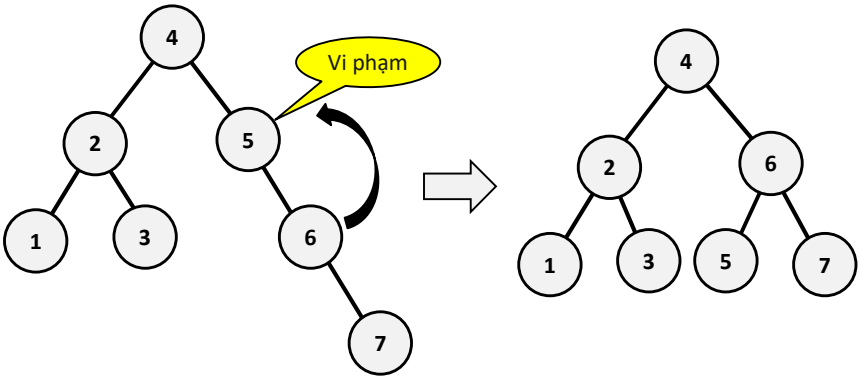


Xoay giữa nút vi phạm và nút con của nó

AVL tree

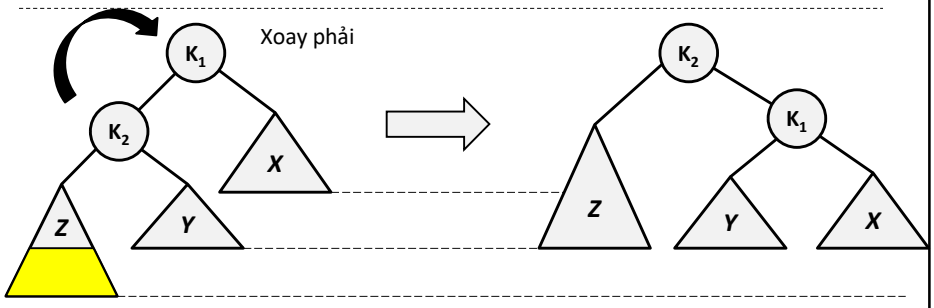


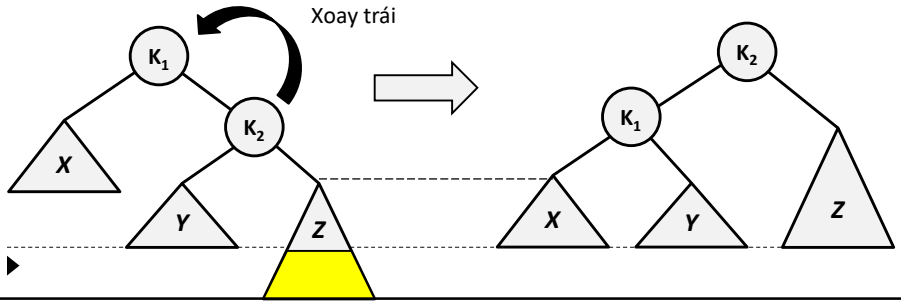
AVL tree



Thêm 7

AVL tree





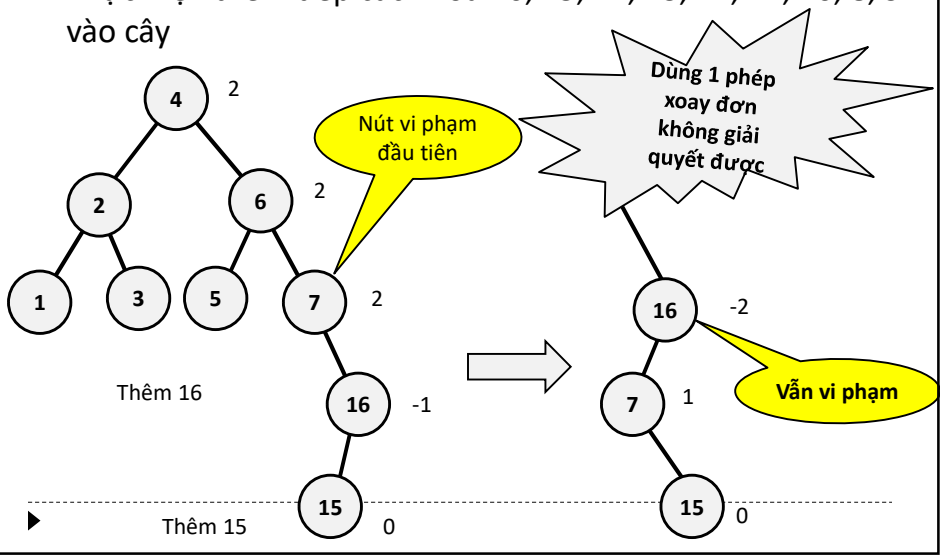
AVL tree

Phép xoay đơn – single rotation:

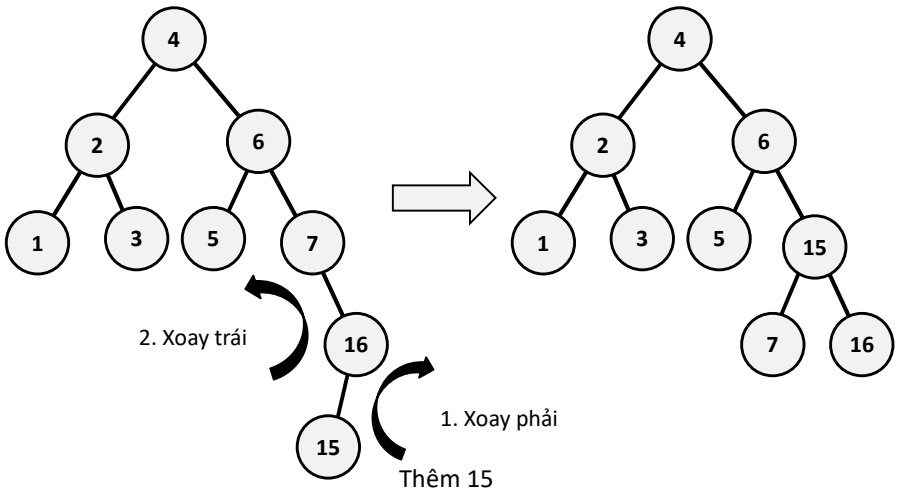
- Dùng để điều chỉnh khi mà nút mới thêm vào trong trường hợp:
  - (i) Cây con trái của nút con trái, hoặc
  - (ii)Cây con phải của nút con phải của nút
- Thực hiện tại nút vi phạm đầu tiên trên đường từ vị trí mới thêm trở về gốc
- Xoay giữa nút vi phạm và nút con trái (xoay phải) – TH i) (hoặc con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

AVL tree

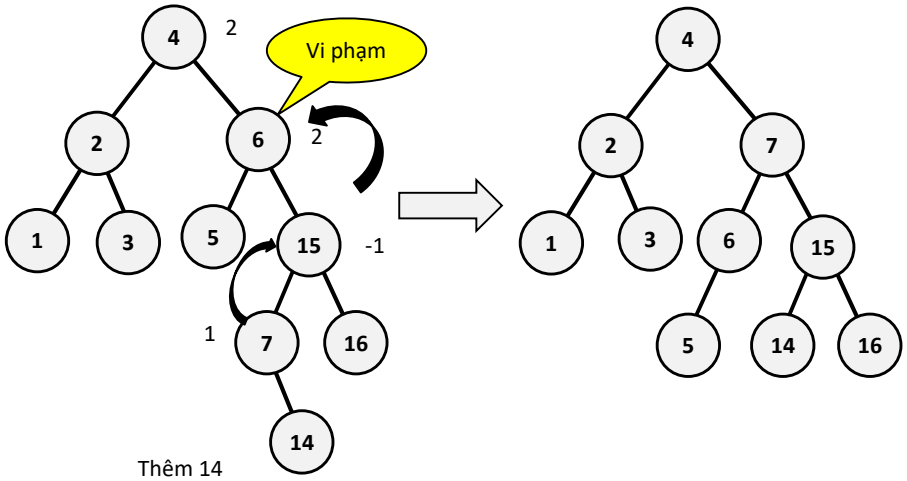
- Thực hiện thêm tiếp các khóa 16, 15, 14, 13, 12, 11, 10, 8, 9 vào cây



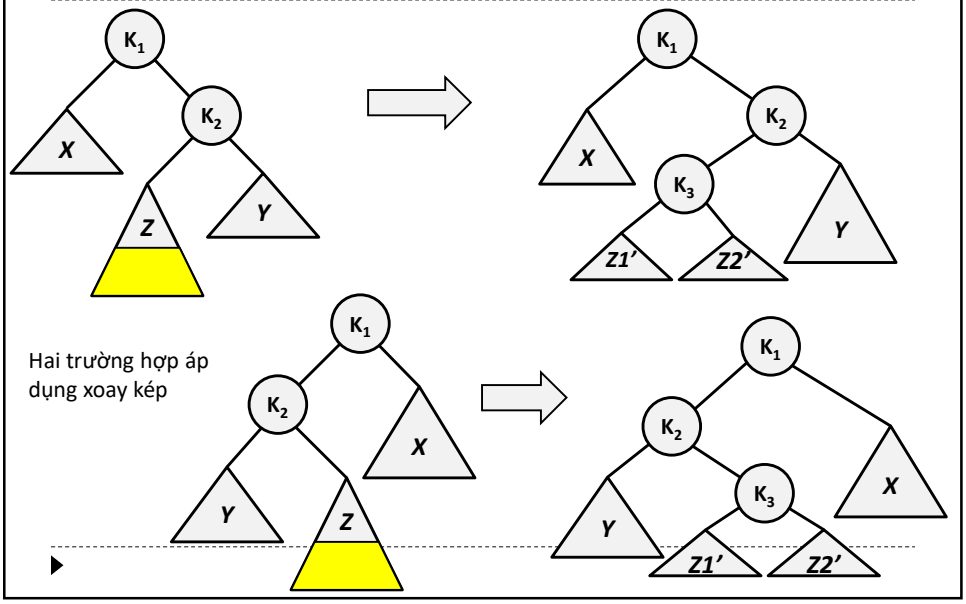
AVL tree



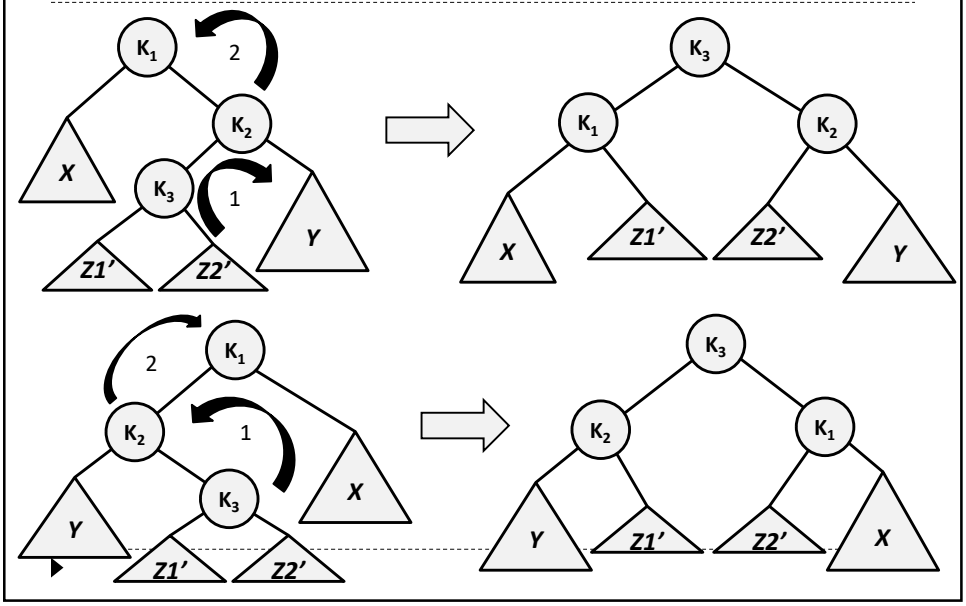
AVL tree



AVL tree



AVL tree

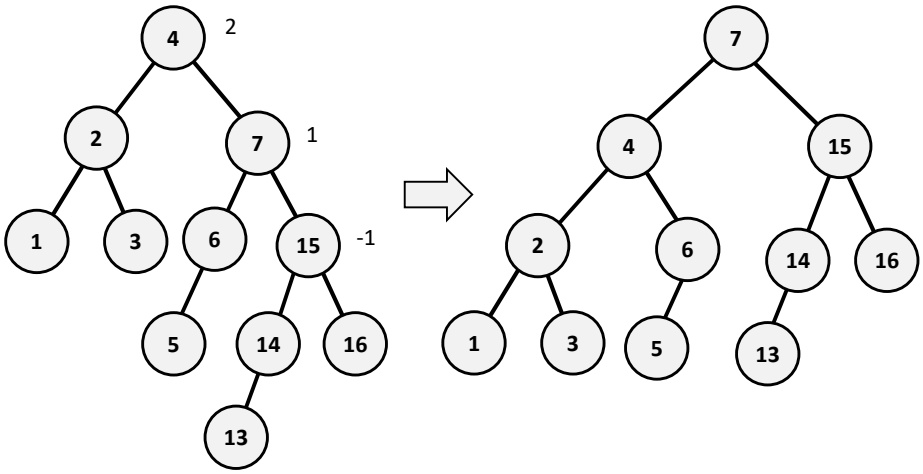


AVL tree

Phép xoay kép – double rotation:

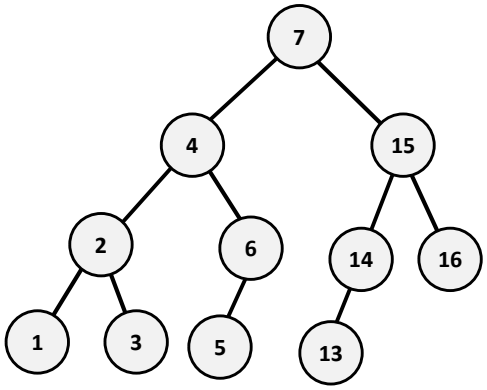
- Dùng để điều chỉnh khi mà nút mới thêm vào trong trường hợp:
  - (i) Cây con phải của nút con trái, hoặc
  - (ii)Cây con trái của nút con phải của nút
- Thực hiện tại nút vi phạm đầu tiên trên đường từ vị trí mới thêm trở về gốc
- Xoay giữa nút vi phạm, nút con, và nút cháu (con của nút con)
- Xoay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

AVL tree



Thêm 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
    //right-left 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);
    }
}
```

```c
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int balance;
} AVLNode;
```

```c
int height(Node* node) {
    if (node == NULL) return 0;
    int left = height(node->left);
    int right = height(node->right);
    return max(left, right) + 1;
}

void updateBalance(Node* node) {
    if (node != NULL)
        node->balance = height(node->left) - height(node->right);
}
```

▶

```c
AVLNode* createNode(int key) {
    AVLNode* node =
(AVLNode*)malloc(sizeof(AVLNode));
    node->key = key;
    node->left = node->right = NULL;
    node->balance = 0;
    return node;
}
```

```c
AVLNode* rightRotate(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;

    x->right = y;
    y->left = T2;

    updateBalance(y);
    updateBalance(x);

    return x;
}
```

▶

```
AVLNode* leftRotate(AVLNode* x)
{
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

    y->left = x;
    x->right = T2;

    updateBalance(x);
    updateBalance(y);

    return y;
}
```

```
AVLNode* insert(AVLNode* node, int key) {
    if (node == NULL) return createNode(key);

    if (key < node->key) node->left = insert(node->left, key);
    else if (key > node->key) node->right = insert(node->right, key);
    else return node;  // Không cho phép trùng

    updateBalance(node);

    // Kiểm tra mất cân bằng
    if (node->balance > 1 && key < node->left->key)
        return rightRotate(node); // Left Left

    if (node->balance < -1 && key > node->right->key)
        return leftRotate(node);  // Right Right

    if (node->balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left); // Left Right
        return rightRotate(node);
    }

    if (node->balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right); // Right Left
        return leftRotate(node);
    }

    return node;
}
```

```c
void inorder(AVLNode* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
}
```

```c
int main() {
    AVLNode* root = NULL;
    int keys[] = { 30, 20, 40, 10, 25, 35, 50 };
    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++)
        root = insert(root, keys[i]);

    printf("Inorder AVL Tree: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

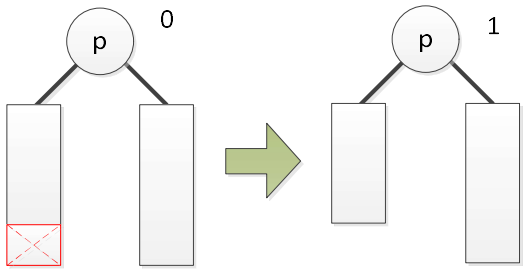
AVL tree

- 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)

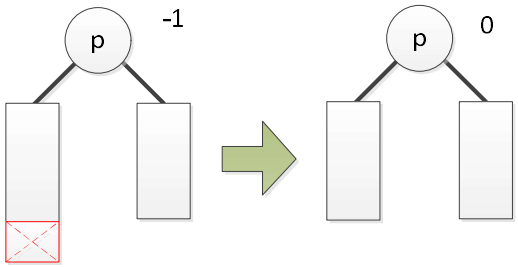
AVL tree



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

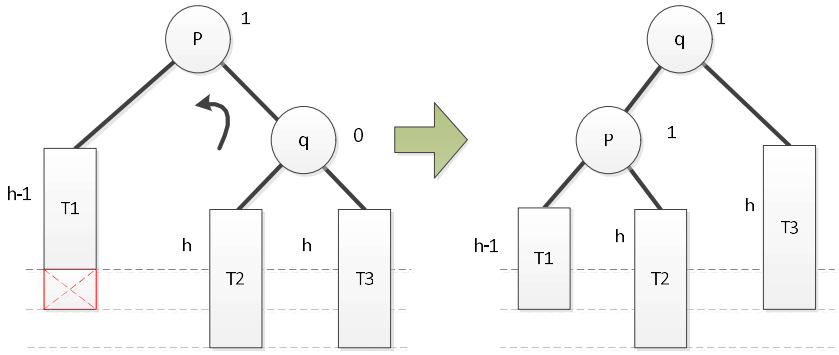
- **Trường hợp 3**: cây đang bị lệch và nút bị xóa nằm trên nhánh thấp hơn.
  - Để cân bằng lại cây ta phải thực hiện các phép xoay.

  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

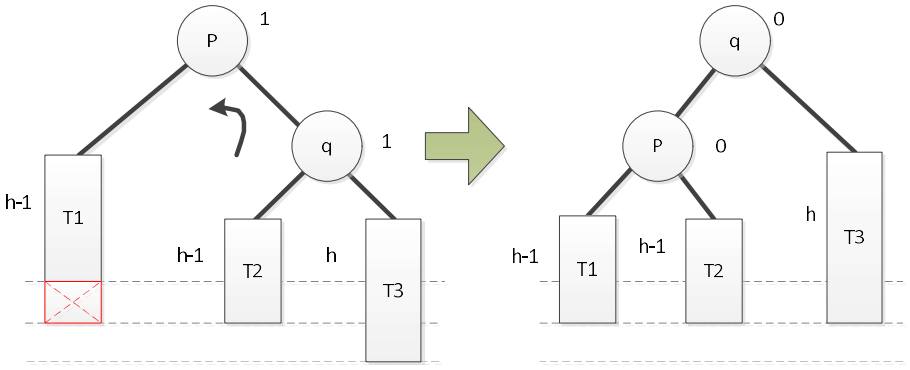


Chiều cao cây không đổi

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

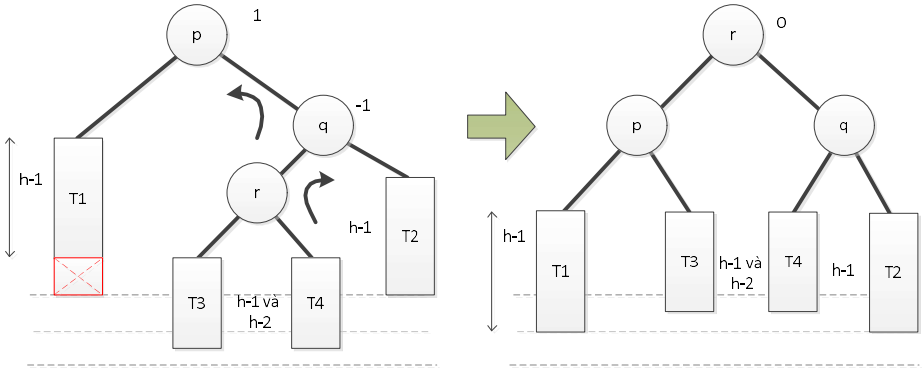


Chiều cao cây thay đổi

AVL tree

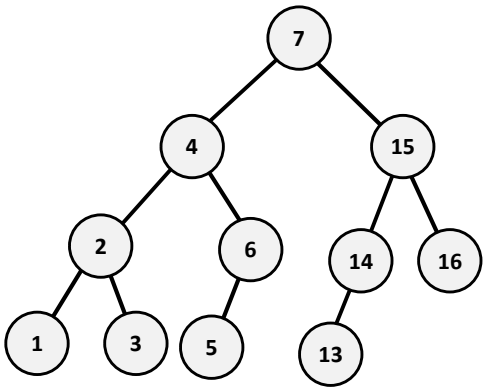
- Trường hợp 3.3: nút q bị lệch cùng phía với nhánh bị xóa. Nếu nhánh bị xóa là nhánh trái của p thì q bị lệch trái và ngược lại
  - Để tái cân bằng cho p ta phải thực hiện 2 phép xoay giữa nút con của q, nút q, và nút p
  - Sau khi xoay, chiều cao của cây giảm đi, p trở về trạng thái cân bằng

AVL tree



Chiều cao cây thay đổi

AVL tree



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

```
AVLNode* deleteNode(AVLNode* root, int key) {
    if (root == NULL)
        return NULL;

    // Step 1: tìm và xóa nút
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        // Node có một hoặc không con
        if (root->left == NULL || root->right == NULL) {
            Node* temp = root->left ? root->left : root->right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
            else {
                *root = *temp; // copy dữ liệu
            }
            free(temp);
        }
        // Node có 2 con
        else {
            Node* temp = minValueNode(root->right);
```

```
    // Nếu cây chỉ có 1 node thì return
    if (root == NULL) return NULL;
    // Step 2: Cập nhật balance
    updateBalance(root);

    // Step 3: Kiểm tra mất cân bằng và quay
    // Left Left
    if (root->balance > 1 && root->left->balance >= 0)
        return rightRotate(root);

    // Left Right
    if (root->balance > 1 && root->left->balance < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right
    if (root->balance < -1 && root->right->balance <= 0)
        return leftRotate(root);

    // Right Left
    if (root->balance < -1 && root->right->balance > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}
```

```c
int main() {
    Node* root = NULL;
    int keys[] = { 50, 30, 70, 20, 40, 60, 80 };
    int n = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < n; i++)
        root = insert(root, keys[i]);

    printf("Inorder before delete: ");
    inorder(root);
    printf("\n");

    root = deleteNode(root, 70);
    printf("Inorder after delete 70: ");
    inorder(root);
    printf("\n");

    return 0;
}
```

Ứng dụng của AVL tree

- Trong quản lý cơ sở dữ liệu
  - Tạo và quản lý chỉ mục (Indexing): Tìm kiếm nhanh mã khách hàng, ID sản phẩm, hoặc ngày giao dịch trong cơ sở dữ liệu lớn
  - Quản lý khóa chính (Primary Key): kiểm tra trùng trước khi thêm
  - Hỗ trợ tìm kiếm nhanh: Lưu trữ các bản ghi hoặc tham chiếu tới bản ghi dựa trên các thuộc tính quan trọng (ví dụ: tên, mã, hoặc ngày).
  - Quản lý dữ liệu phân đoạn (Partitioned Data): Quản lý và định vị các phân đoạn dữ liệu dựa trên phạm vi giá trị hoặc thuộc tính
  - Xử lý các truy vấn phạm vi (Range Queries)
  - Lưu trữ tạm thời trong bộ nhớ (In-Memory Data Storage): Lưu trữ thông tin truy cập gần đây trong các hệ thống cache
  - Quản lý dữ liệu không gian (Spatial Data)

Ứng dụng của AVL tree

- Trong quản lý hệ thống lưu trữ tệp (File Systems)
  - Quản lý cấu trúc thư mục: khóa là tên file/thư mục để tìm kiếm nhanh
  - Chỉ mục tệp (File Indexing): lưu trữ thông tin như ngày tạo, kích thước,.. E.g Linux ext4
  - Quản lý bảng phân bổ tệp (File Allocation Table - FAT): xác định block tiếp theo của file
  - Tìm kiếm phạm vi tệp (Range Queries): theo khoảng tạo/truy cập,..
  - Quản lý hệ thống tệp phân tán (Distributed File Systems): Quản lý thông tin về vị trí lưu trữ của các khối dữ liệu hoặc tệp trên các máy chủ khác nhau. E.g Google Drive, Dropbox
  - Quản lý metadata (Metadata Management): tên, quyền truy cập, ngày tạo, và ngày sửa đổi.
  - Tìm kiếm nội dung trong file: Windows Search, Spotlight trên macOS
  - Nén và phân mảnh dữ liệu: Quản lý các đoạn dữ liệu nén của tệp hoặc vị trí của các phần bị phân mảnh

Ứng dụng của AVL tree

- Trong Hệ thống quản lý bộ nhớ (Memory Management)
  - Quản lý vùng trống trong bộ nhớ (Free Space Management)
    - Lưu trữ thông tin về các vùng trống trong cây AVL, với khóa là kích thước hoặc địa chỉ vùng trống.
    - Khi cần cấp phát bộ nhớ:Tìm kiếm vùng trống có kích thước phù hợp (Best Fit hoặc First Fit) $O(\log n)$
    - Khi giải phóng bộ nhớ: Hợp nhất các vùng trống liền kề và cập nhật cây AVL.
  - Quản lý phân đoạn bộ nhớ (Memory Segmentation)
  - Quản lý phân trang bộ nhớ (Paging Management): ánh xạ giữa địa chỉ ảo (virtual address) và địa chỉ vật lý (physical address)
  - Phân bổ bộ nhớ động (Dynamic Memory Allocation)
  - Quản lý bộ nhớ trong hệ thống nhúng (Embedded Systems)
  - Phát hiện và khắc phục phân mảnh bộ nhớ (Memory Fragmentation)
  - Tăng hiệu suất cho hệ thống đa luồng (Multithreading)

AVL tree

- Bài toán 1. xây dựng ứng dụng tra cứu từ điển anh-việt với đầu vào là 1 file từ điển từ txt
- Bài toán 2. Cho 1 danh sách gồm n phần tử, hãy đưa ra thuật toán hiệu quả để lọc các phần tử bị trùng trong danh sách
- Bài toán 3. Cho 1 văn bản tiếng anh, hãy thống kê tần số xuất hiện của các từ trong văn bản một cách hiệu quả

Question

Splay tree
Cấu trúc tự điều chỉnh

Cấy trúc tự điều chỉnh

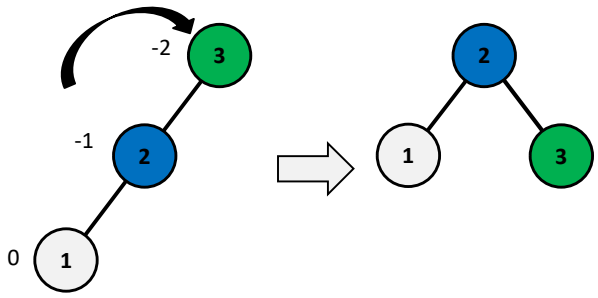
- Trong nhiều bài toán chúng ta cần một cấu trúc xử lý hiệu quả với những truy cập có số lượng lớn trên các bản ghi mới đưa vào.
- Ví dụ: bài toán quản lý thông tin bệnh nhân tại bệnh viện
  - Bệnh nhân ra khỏi bệnh viện thì có số lần truy cập thông tin ít hơn
  - Bệnh nhân mới vào viện thì sẽ có số 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

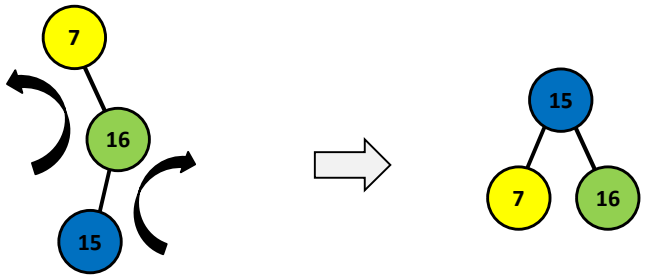
Cây splay

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



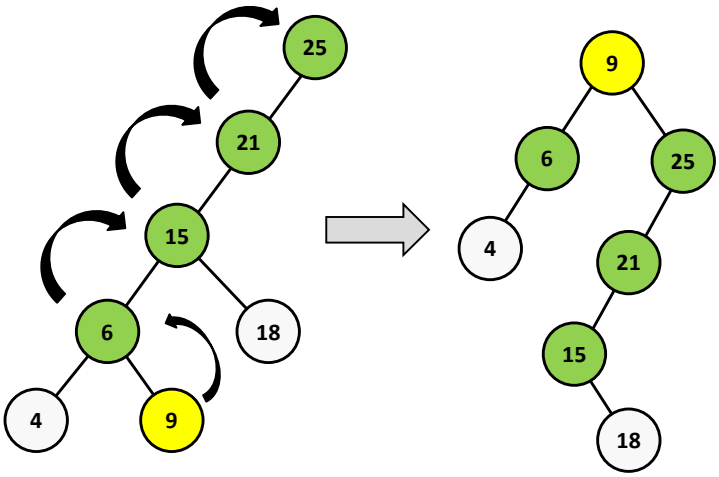
Cây splay

- **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

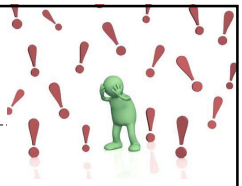


Cây splay

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



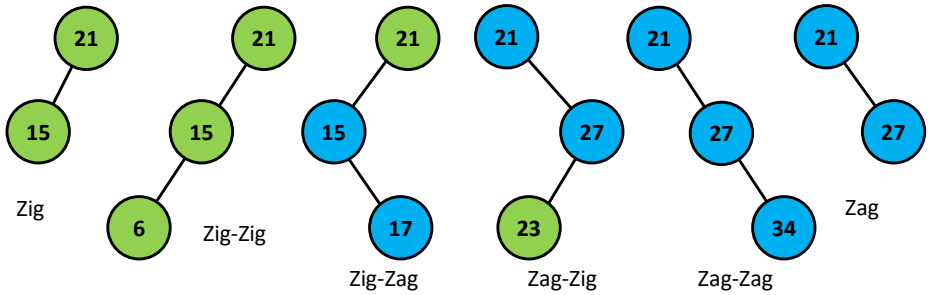
Cây splay



- **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***

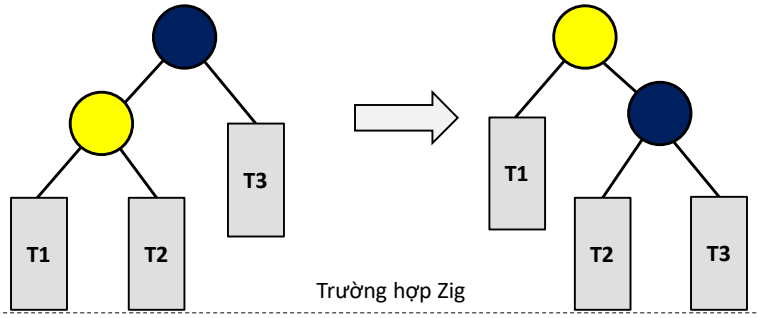
Cây splay

- 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



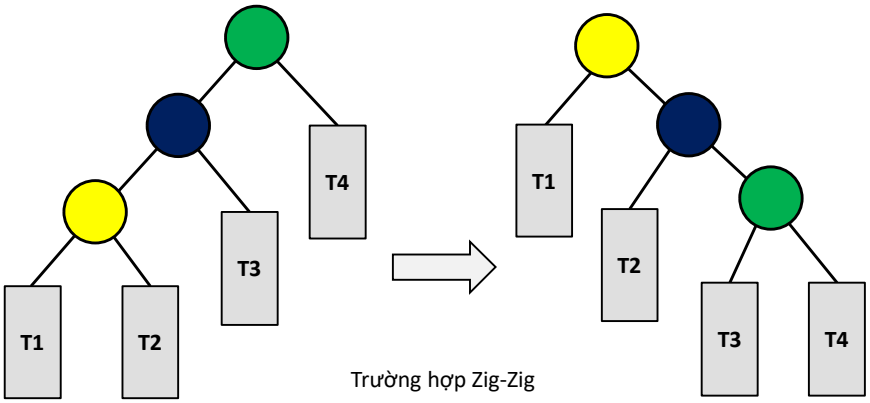
Cây splay

- **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)



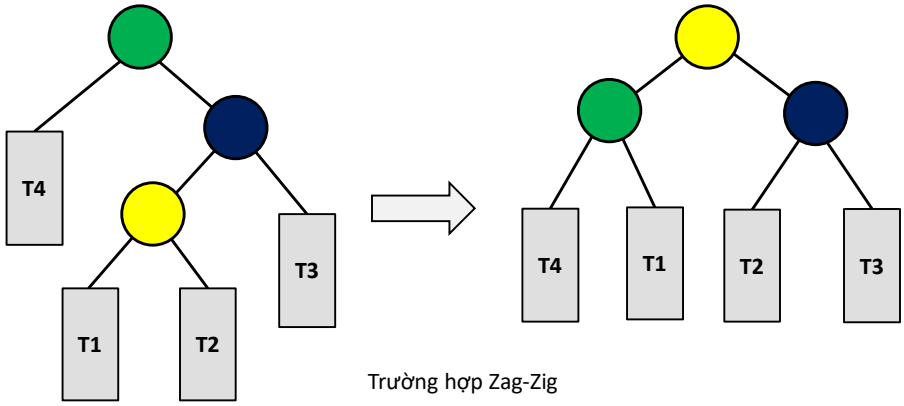
Trường hợp Zig

Cây splay



Trường hợp Zig-Zig

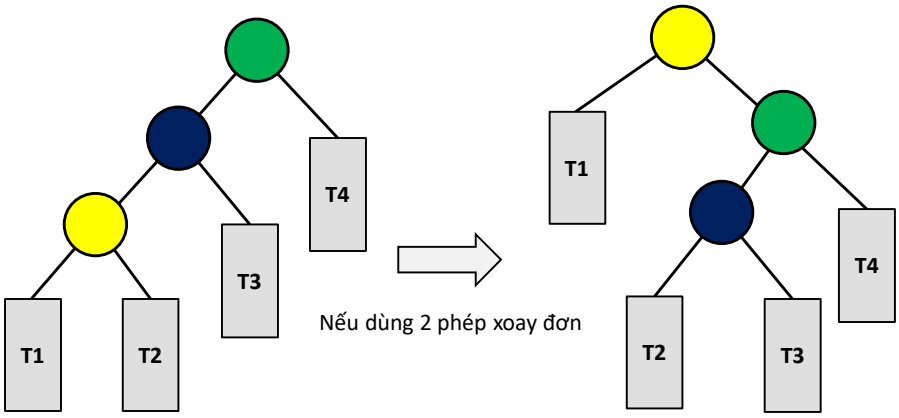
Cây splay



Trường hợp Zag-Zig

Cây splay

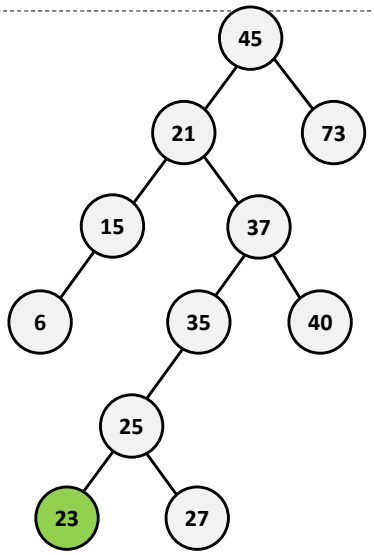
- Chú ý: trường hợp Zig-Zig (hoặc Zag-Zag) khác hoàn toàn với trường hợp dùng hai phép xoay đơn liên tiếp



Nếu dùng 2 phép xoay đơn

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

```c
// Cấu trúc nút của cây Splay
typedef struct Node {
    int key;
    struct Node* left, * right;
} Node;

// Tạo nút mới
Node* createNode(int key) {
    Node* newNode =
(Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

```c
// Xoay phải (right rotation)
Node* rightRotate(Node* x) {
    Node* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}
```

```c
// Xoay trái (left rotation)
Node* leftRotate(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
```

```c
// Hàm splay: Đưa nút có khóa key lên gốc
Node* splay(Node* root, int key) {
    if (root == NULL || root->key == key) return root;

    // Nếu key nhỏ hơn gốc
    if (key < root->key) {
        if (root->left == NULL) return root;

        // Zig-Zig (left-left)
        if (key < root->left->key) {
            root->left->left = splay(root->left->left, key);
            root = rightRotate(root);
        }
        // Zig-Zag (left-right)
        else if (key > root->left->key) {
            root->left->right = splay(root->left->right, key);
            if (root->left->right)
                root->left = leftRotate(root->left);
        }

        return (root->left == NULL) ? root : rightRotate(root);
    }
```

```
    // Nếu key lớn hơn gốc
    else {
        if (root->right == NULL) return root;

        // Zag-Zag (right-right)
        if (key > root->right->key) {
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }
        // Zag-Zig (right-left)
        else if (key < root->right->key) {
            root->right->left = splay(root->right->left, key);
            if (root->right->left)
                root->right = rightRotate(root->right);
        }

        return (root->right == NULL) ? root : leftRotate(root);
    }
}
```

```
Node* insert(Node* root, int key) {
    if (root == NULL) return createNode(key);

    // Splay cây để đưa nút gần key nhất lên gốc
    root = splay(root, key);

    // Nếu key đã tồn tại
    if (root->key == key) return root;

    Node* newNode = createNode(key);

    // Nếu key nhỏ hơn gốc
    if (key < root->key) {
        newNode->right = root;
        newNode->left = root->left;
        root->left = NULL;
    }
    // Nếu key lớn hơn gốc
    else {
        newNode->left = root;
        newNode->right = root->right;
        root->right = NULL;
    }

    return newNode;
}
```

```c
// Tìm kiếm một khóa
Node* search(Node* root, int key) {
    return splay(root, key);
}

// Tìm nút có giá trị nhỏ nhất
Node* findMin(Node* root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}
```

```c
// In cây theo thứ tự giữa (inorder)
void inorder(Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}
```

```c
// Giải phóng bộ nhớ
void freeTree(Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
```

```c
// Xóa một khóa khỏi cây
Node* delete(Node* root, int key) {
    if (root == NULL) return NULL;

    // Splay để đưa nút cần xóa lên gốc
    root = splay(root, key);

    // Nếu key không tồn tại
    if (root->key != key)
        return root;

    // Nếu nút gốc không có con trái
    if (root->left == NULL) {
        Node* temp = root->right;
        free(root);
        return temp;
    }

    // Nếu có con trái, tìm giá trị nhỏ nhất trong cây con trái
    Node* temp = splay(root->left, key);
    temp->right = root->right;
    free(root);
    return temp;
}
```

Cây đỏ đen
R-B Tree

Cây đỏ đen: R-B Tree

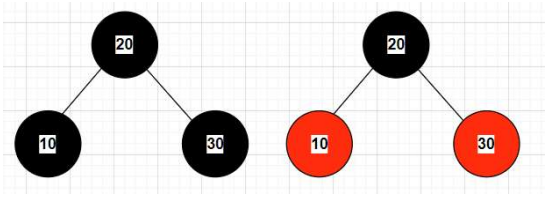
- Cây đỏ đen
  - Là cấu trúc cây nhị phân tìm kiếm tự cân bằng như AVL
  - Dùng thuộc tính màu sắc của nút để quản lý việc cân bằng (mỗi nút màu đen hoặc đỏ - chỉ mất 1 bit để biểu diễn)
  - Cây cân bằng không thực sự tốt như AVL nhưng các thao tác trên cây vẫn cỡ O(logn)
  - Được phát triển vào năm 1972 bởi Rudolf Bayer
- Yêu cầu cân bằng của cây
  1. Mỗi nút có màu đỏ hoặc đen
  2. Nút gốc luôn là màu đen
  3. Không có 2 nút đỏ liên tiếp (trên đường đi từ gốc tới lá)
  4. Cân bằng đen: Số lượng nút đen trên mọi đường từ gốc tới lá đều bằng nhau

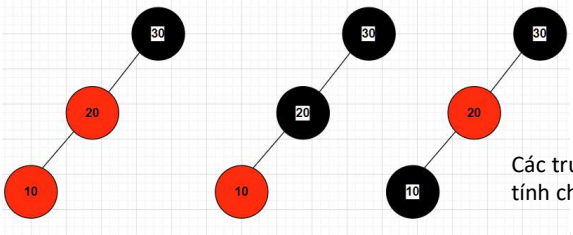
Cây đỏ đen: R-B Tree

- So sánh với cây AVL
  - R-B trên rằng buộc cân bằng không quá chặt như AVL (chiều cao cây R-B sẽ cao hơn so với AVL)
  - Việc tìm kiếm trên R-B tree sẽ tồi hơn 1 chút so với trên cây AVL
  - Cài đặt cây R-B sẽ đơn giản hơn so với AVL vì ít trường hợp phải điều chỉnh cây hơn
  - R-B tree phù hợp với bài toán mà việc thêm xóa diễn ra thường xuyên
  - AVL tree phù hợp với bài toán mà việc tìm kiếm diễn ra thường xuyên hơn (VD. bài toán tra cứu từ điển)

Cây đỏ đen: R-B Tree

- Các cây đỏ đen có thể tạo ra từ 3 nút





Các trường hợp còn lại đều vi phạm tính chất 3 hoặc 4 của r-b tree

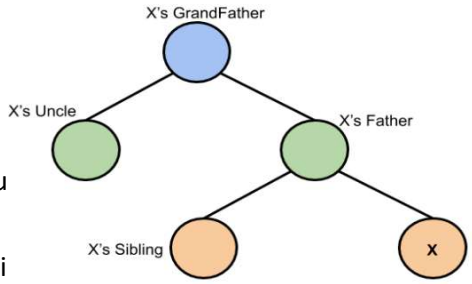
Cây đỏ đen: R-B Tree

- Đặc điểm của R-B tree
  - Số lượng nút đen trên mọi đường đi từ gốc tới lá đều bằng nhau. Cây chiều cao h sẽ có số lượng nút đen trên đường đi >h/2
  - Chiều cao cây với n nút là h<=2xlog(n+1)
  - Tất cả các nút lá NULL sẽ có màu là đen
- Ứng dụng của R-B tree
  - Dùng trong map và set của STL trong C++
  - Dùng trong TreeSet và TreeMap trong java
  - Dùng trong điều độ tiến trình trên HDH Linux
  - Dùng trong hệ quản trị MySQL để đánh index các bảng cơ sở dữ liệu

Cây đỏ đen: R-B Tree

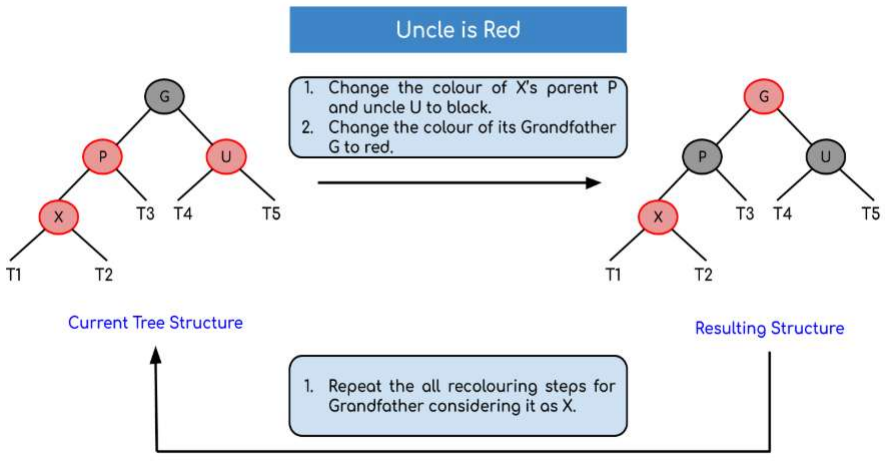
- Thêm nút vào cây đỏ đen
  - Cách điều chỉnh lại
    - Thay đổi màu của nút, hoặc
    - Xoay nút
  - Luôn thử đổi màu nút trước, nếu không thể mới chuyển qua xoay
  - Nút mới thêm vào mặc định sẽ là màu đỏ
  - Nếu cha nó là nút màu đen thì không cần phải làm gì thêm

Cây đỏ đen: R-B Tree



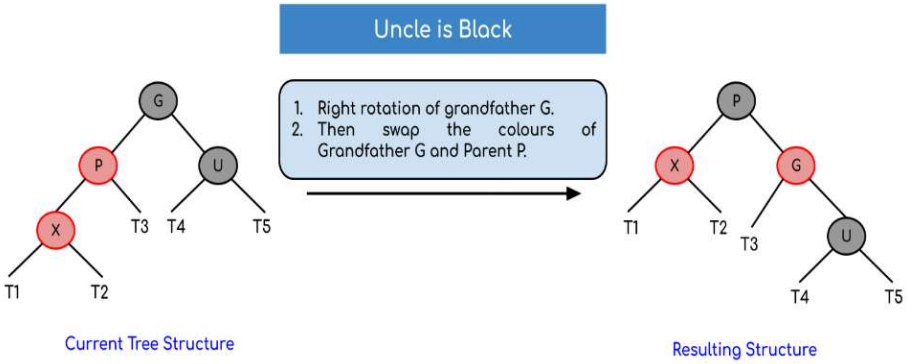
- Quá trình thêm nút như thêm nút vào cây tìm kiếm thông thường, sau đó sẽ đổi màu
- Nút mới thêm sẽ là màu đỏ
- Nếu nút mới thêm là nút gốc, đổi màu sang màu đen
- Ngược lại, check màu của nút cha
  - Nếu nút cha màu đen, vẫn giữ nút mới thêm màu đỏ
  - Ngược lại, check màu của nút Uncle (nút anh chị em với nút cha), nếu nó cũng màu đỏ thì đổi màu cả 2 nút này về màu đen, và đổi màu nút ông (GrandFather) về màu đỏ.
    Lặp lại quá trình check màu nếu cần với nút ông – GrandFather
  - Nếu nút uncle khác màu nút cha, cần phải xoay

Cây đỏ đen: R-B Tree



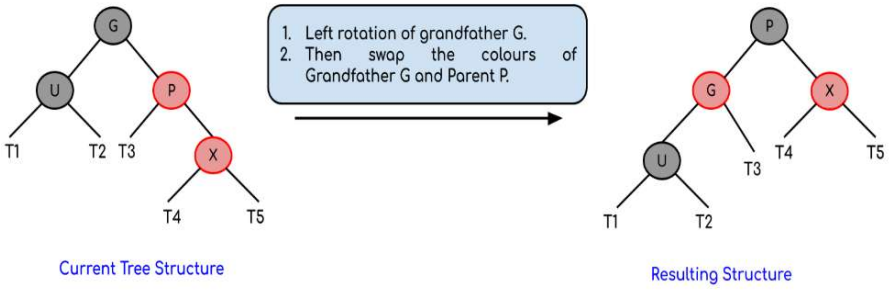
Cây đỏ đen: R-B Tree

- Nút **uncle khác màu với nút cha** ➔ 4 trường hợp xoay
- 1. Left Left Case (LL rotation): trái - trái



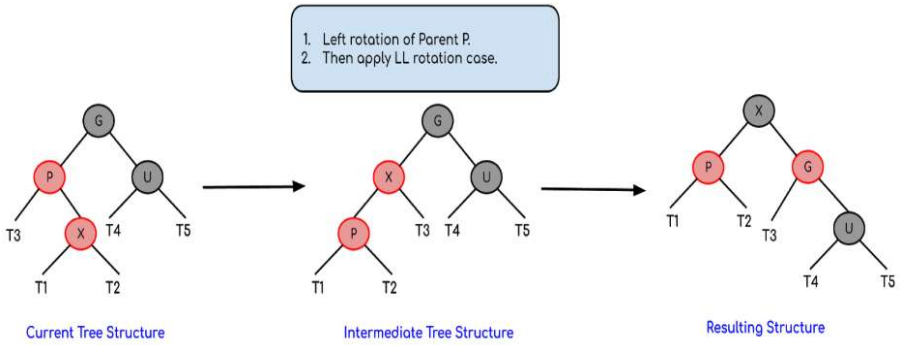
Cây đỏ đen: R-B Tree

- 2. Right Right Case (RR rotation): phải – phải



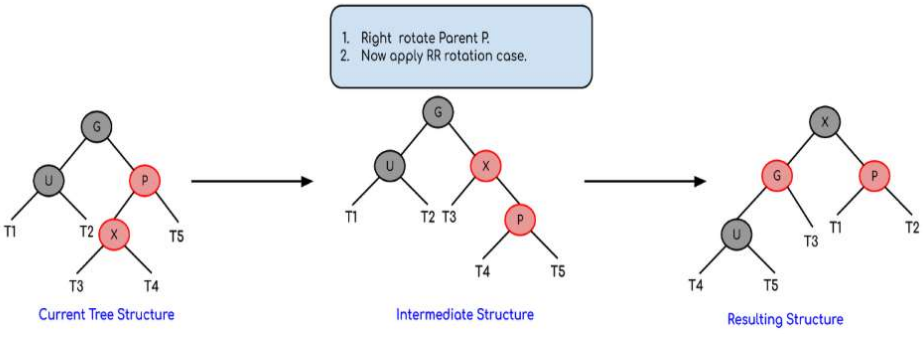
Cây đỏ đen: R-B Tree

- 3. Trường hợp phải – trái ➔ xoay kép tương tự AVL



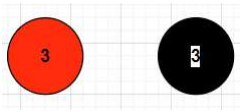
Cây đỏ đen: R-B Tree

- 4. Trường hợp trái – phải ➔ xoay kép

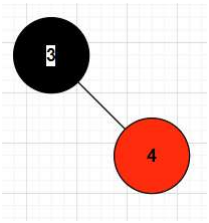


Cây đỏ đen: R-B Tree

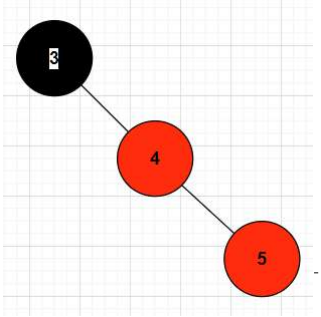
- Ví dụ. Thêm lần lượt các nút sau vào cây đỏ đen ban đầu rỗng 3, 4, 5, 7, 2, 10, 9, 8



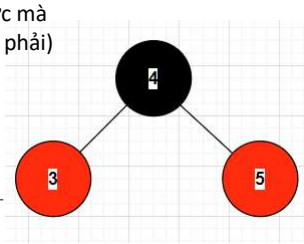
Thêm 3 là gốc → đổi màu



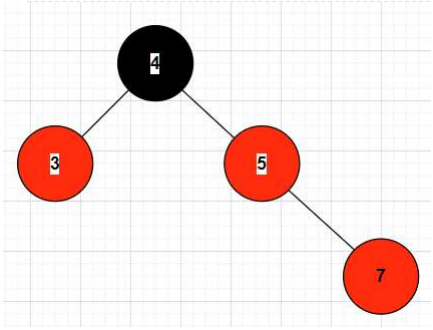
Thêm 4, không cần đổi màu

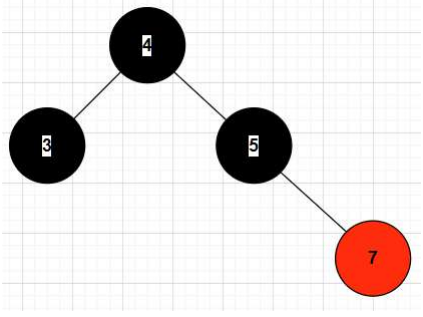


Thêm 5, đổi màu không được mà cần xoay (trường hợp Phải – phải)



Cây đỏ đen: R-B Tree





Thêm 7, cần đổi màu nút Parent và nút Uncle

Cây đỏ đen: R-B Tree

- Thêm 2, 10, 9, 8


```
typedef enum { RED, BLACK } Color;

typedef struct Node {
    int key;
    Color color;
    struct Node *left, *right, *parent;
} Node;

Node *root = NULL;
```

```
// Tạo nút mới
Node* createNode(int key) {
    Node *node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->color = RED;
    node->left = node->right = node->parent = NULL;
    return node;
}
```

Cây đỏ đen: R-B Tree

```
// Quay trái tại nút x và con phải của nó
// root là gốc cây để sử lý nếu x là gốc
void rotateLeft(Node **root, Node *x) {
    Node *y = x->right;
    x->right = y->left;
    if (y->left) y->left->parent = x;
    y->parent = x->parent;
    if (!x->parent)
        *root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
}
```

Cây đỏ đen: R-B Tree

```
// Quay phải giữa nút y và con trái của nó
// root là gốc cây để sử lý nếu x là gốc
void rotateRight(Node **root, Node *y) {
    Node *x = y->left;
    y->left = x->right;
    if (x->right) x->right->parent = y;
    x->parent = y->parent;
    if (!y->parent)
        *root = x;
    else if (y == y->parent->right)
        y->parent->right = x;
    else
        y->parent->left = x;
    x->right = y;
    y->parent = x;
}
```

```
// Sửa sau khi chèn với nút mới chèn thêm là z
// root là gốc của cây, sẽ được xử lý thêm nếu cần
void fixInsert(Node **root, Node *z) {
    while (z->parent && z->parent->color == RED) {
        if (z->parent == z->parent->parent->left) {
            Node *y = z->parent->parent->right;
            if (y && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->right) {
                    z = z->parent;
                    rotateLeft(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rotateRight(root, z->parent->parent);
            }
        }
```

```
    else {
            Node *y = z->parent->parent->left;
            if (y && y->color == RED) {
                z->parent->color = BLACK;
                y->color = BLACK;
                z->parent->parent->color = RED;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    rotateRight(root, z);
                }
                z->parent->color = BLACK;
                z->parent->parent->color = RED;
                rotateLeft(root, z->parent->parent);
            }
        }
    }
    (*root)->color = BLACK;
}
```

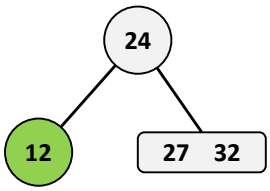
```c
// hàm thêm nút vào cây với gốc là root,
// và khóa mới thêm là key
void insert(Node **root, int key) {
    Node *z = createNode(key);
    Node *y = NULL;
    Node *x = *root;

    while (x != NULL) {
        y = x;
        if (z->key < x->key) x = x->left;
        else x = x->right;
    }

    z->parent = y;
    if (!y) *root = z;
    else if (z->key < y->key) y->left = z;
    else y->right = z;

    fixInsert(root, z);
}
```

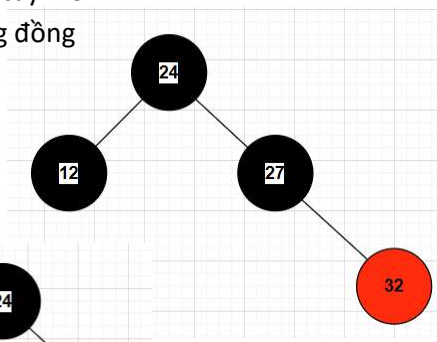
```c
void inorder(Node *root) {
    if (root) {
        inorder(root->left);
        printf("%d (%s)\n",root->key, root->color == RED?"R":"B");
        inorder(root->right);
    }
}
```

```c
int main() {
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 30);
    insert(&root, 15);
    insert(&root, 25);
    insert(&root, 5);

    printf("Inorder:\n");
    inorder(root);

    return 0;
}
```

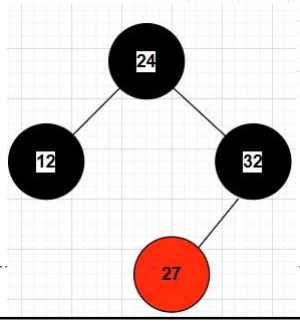
Cây đỏ đen: R-B Tree

- Mối tương đồng giữa cây đỏ đen và cây 2-3
  - Cây đỏ đen và cây 2-3 có sự tương đồng





Hoặc

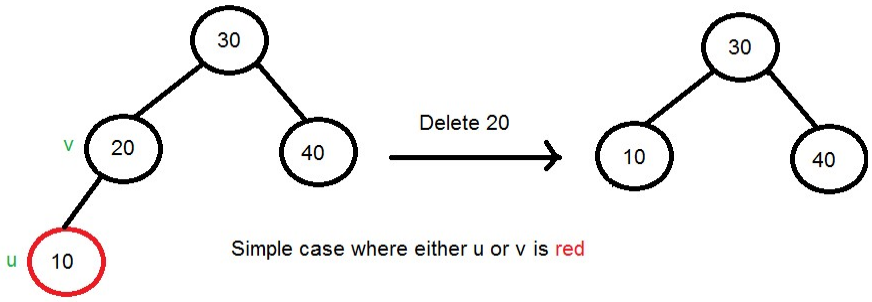


Cây đỏ đen: R-B Tree

- Xóa nút trên cây đỏ đen
  - Thực hiện bước xóa tương tự như trên cây nhị phân tìm kiếm thông thường
  - Trường hợp nút trong có 1 con: thay thế nó bằng nút con trực tiếp và gọi đệ quy xóa nút con trực tiếp của nó
  - Trường hợp nút trong có 2 con: tìm nút thay thế, và gọi đệ quy xóa nút thay thế (leftmost hoặc rightmost)
  - Nếu nút bị xóa là nút màu đen: sau khi xóa nút, check xem có phải điều chỉnh lại cây bằng việc (đảm bảo cân bằng đen)
    - Đổi màu nút, hoặc
    - Xoay nút

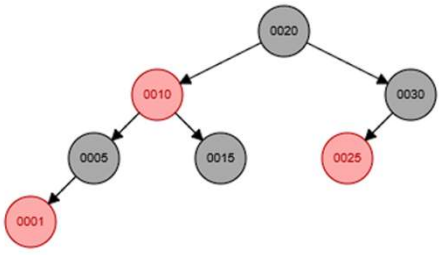
Cây đỏ đen: R-B Tree

- Trường hợp đơn giản: nút lá màu đỏ → xóa mà không cần điều chỉnh gì



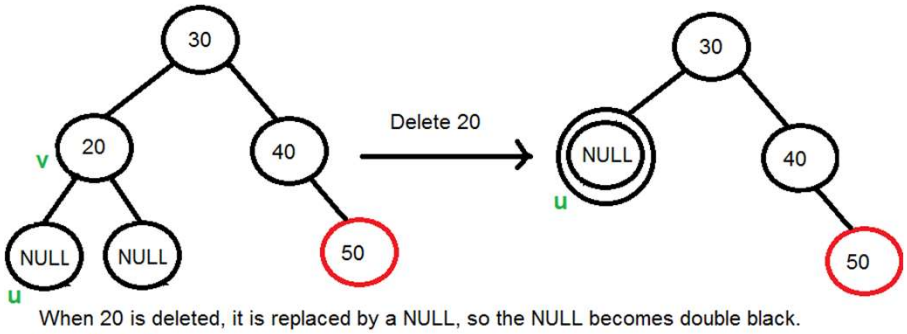
Cây đỏ đen: R-B Tree

- Trường hợp cần điều chỉnh – fixDelete: (nút bị xóa là nút màu đen)
  - Nếu nút thay thế là đỏ → Chuyển thành màu đen
  - Nếu nút thay thế là đen (hoặc là nút lá rỗng – TH nút bị xóa là lá).



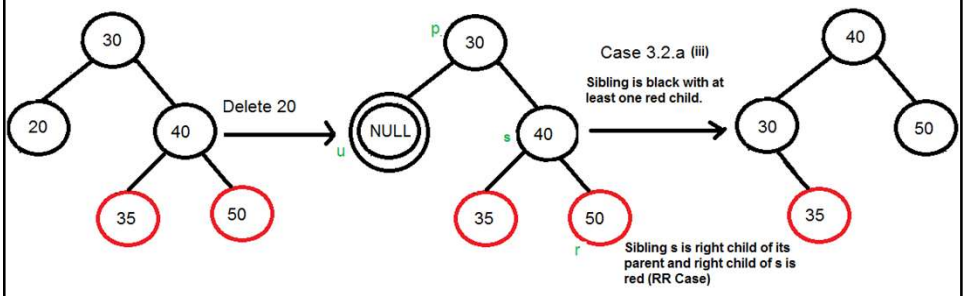
Xóa 30 và xóa 15 (hoặc 20)

Cây đỏ đen: R-B Tree

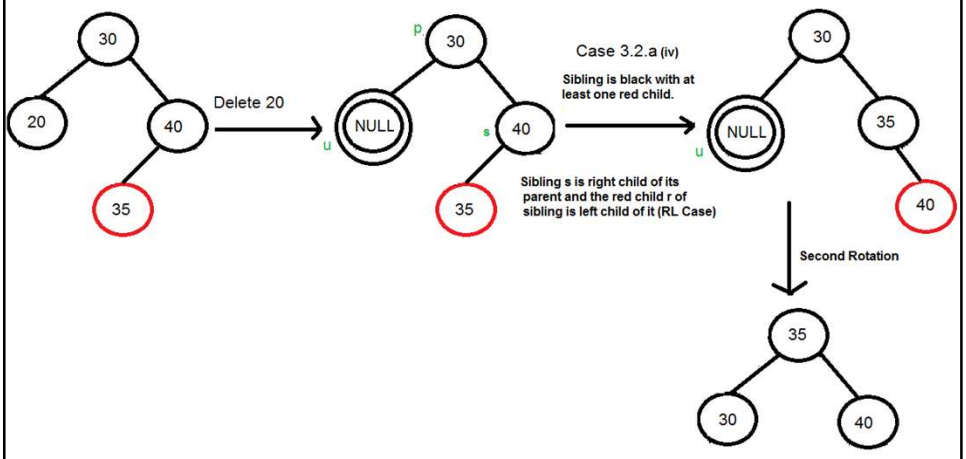


When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

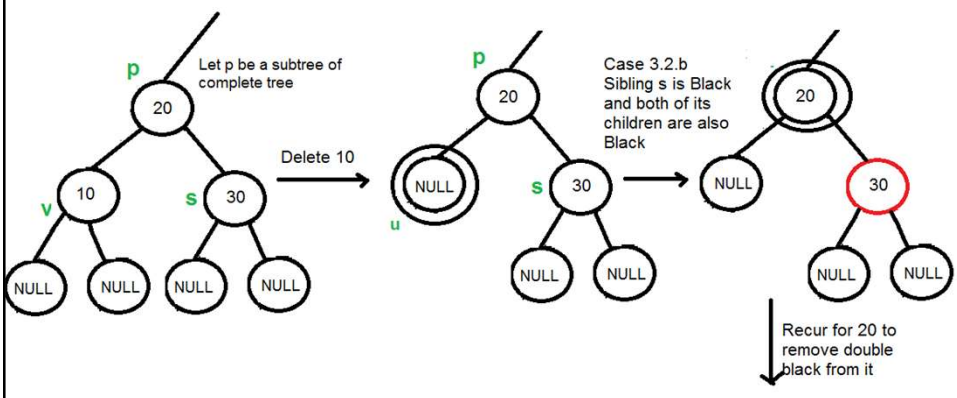
a. Double black và sibling đen có ít nhất 1 con đỏ



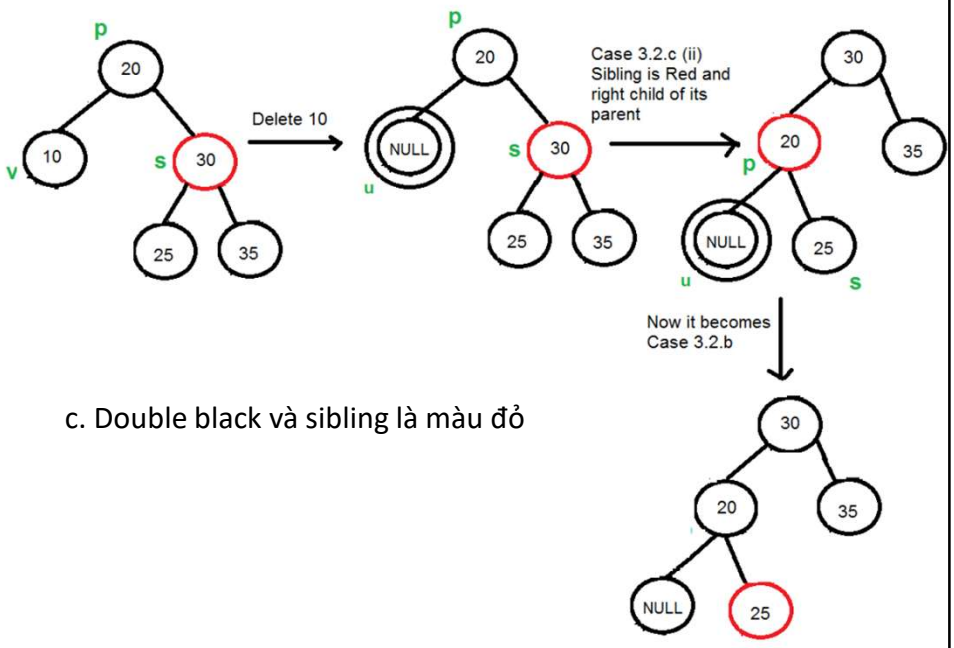
a. Double black và sibling đen có ít nhất 1 con đỏ



b. Double black và sibling đen có 2 con đen



- Đổi màu sibling sang đỏ và đệ quy tiếp nút cha (vẫn double black)



c. Double black và sibling là màu đỏ

```
// Tìm node nhỏ nhất
Node* treeMinimum(Node* x) {
    while (x->left != NULL) x = x->left;
    return x;
}
```

```
// Thay thế 1 node u bằng v
void rbTransplant(Node **root, Node *u, Node *v) {
    if (!u->parent) *root = v;
    else if (u == u->parent->left) u->parent->left = v;
    else u->parent->right = v;
    if (v) v->parent = u->parent;
}
```

```
// Sửa sau khi xóa
void fixDelete(Node **root, Node *x) {
    while (x != *root && (!x || x->color == BLACK)) {
        if (x == x->parent->left) {
            Node *w = x->parent->right;
            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                rotateLeft(root, x->parent);
                w = x->parent->right;
            }
            if ((!w->left || w->left->color == BLACK) &&
                (!w->right || w->right->color == BLACK)) {
                w->color = RED;
                x = x->parent;
            } else {
                if (!w->right || w->right->color == BLACK) {
                    if (w->left) w->left->color = BLACK;
                    w->color = RED;
                    rotateRight(root, w);
                    w = x->parent->right;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                if (w->right) w->right->color = BLACK;
                rotateLeft(root, x->parent);
                x = *root;
            }
```

```
        } else {
            Node *w = x->parent->left;
            if (w->color == RED) {
                w->color = BLACK;
                x->parent->color = RED;
                rotateRight(root, x->parent);
                w = x->parent->left;
            }
            if ((!w->right || w->right->color == BLACK) &&
                (!w->left || w->left->color == BLACK)) {
                w->color = RED;
                x = x->parent;
            } else {
                if (!w->left || w->left->color == BLACK) {
                    if (w->right) w->right->color = BLACK;
                    w->color = RED;
                    rotateLeft(root, w);
                    w = x->parent->left;
                }
                w->color = x->parent->color;
                x->parent->color = BLACK;
                if (w->left) w->left->color = BLACK;
                rotateRight(root, x->parent);
                x = *root;
            }
        }
    }
    if (x) x->color = BLACK;
```

```
void delete(Node **root, int key) {
    Node *z = *root, *x, *y;

    while (z && z->key != key) {
        if (key < z->key) z = z->left;
        else z = z->right;
    }

    if (!z) return;

    y = z;
    Color y_original_color = y->color;
    if (!z->left) {
        x = z->right;
        rbTransplant(root, z, z->right);
    } else if (!z->right) {
        x = z->left;
        rbTransplant(root, z, z->left);
```

```
    } else {
        y = treeMinimum(z->right);
        y_original_color = y->color;
        x = y->right;
        if (y->parent == z) {
            if (x) x->parent = y;
        } else {
            rbTransplant(root, y, y->right);
            y->right = z->right;
            if (y->right) y->right->parent = y;
        }
        rbTransplant(root, z, y);
        y->left = z->left;
        if (y->left) y->left->parent = y;
        y->color = z->color;
    }

    free(z);
    if (y_original_color == BLACK)
        fixDelete(root, x);
}
```

```c
int main() {
    insert(&root, 10);
    insert(&root, 20);
    insert(&root, 30);
    insert(&root, 15);
    insert(&root, 25);
    insert(&root, 5);

    printf("Inorder before delete:\n");
    inorder(root);

    delete(&root, 20);
    delete(&root, 25);

    printf("Inorder after delete:\n");
    inorder(root);

    return 0;
}
```

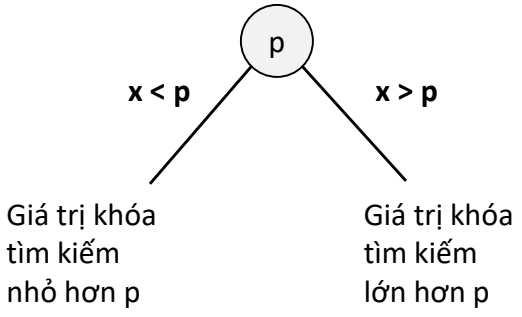
Cây 2-3

Cây 2-3

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

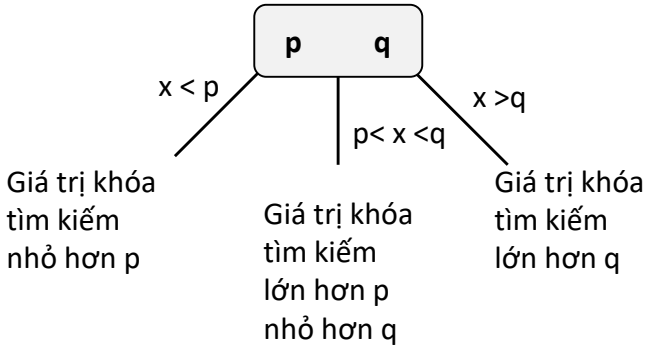
Cây 2-3

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

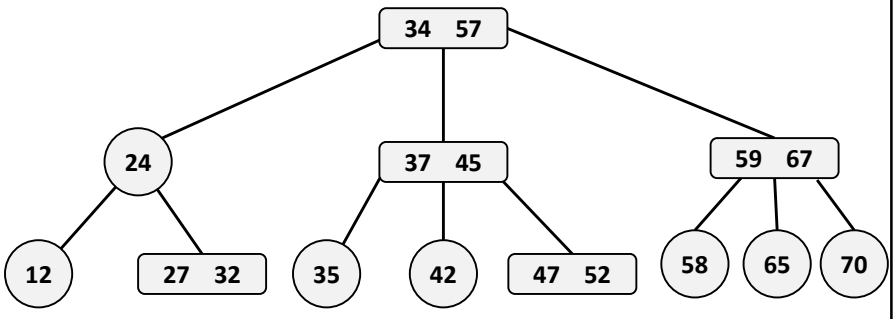


Cây 2-3

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



Cây 2-3



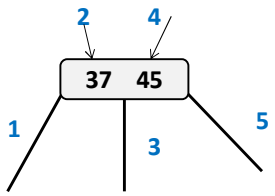
Cây 2-3

- Định nghĩa cấu trúc 1 nút

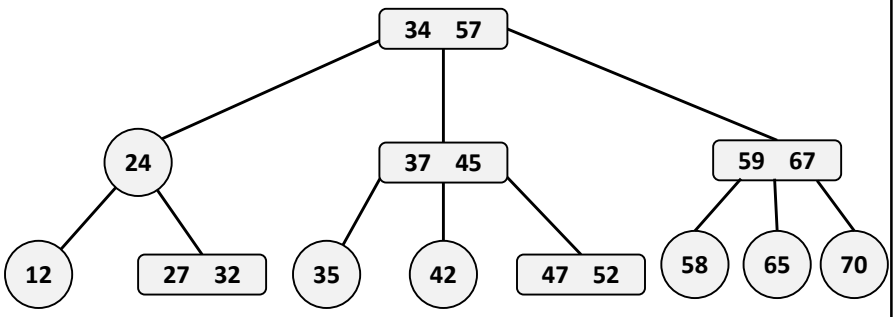
```
struct TreeNode
{
   DATA_TYPE smallItem, largeItem;
   struct TreeNode *left, *middle, *right;
   struct TreeNode *parent; //to make your life easier
}
```

Cây 2-3

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



Cây 2-3



- Duyệt cây theo thứ tự giữa:
  12, 24, 27, 32, 34, 35, 37, 42, 45, 47, 52, 57, 58, 59, 65, 67, 70

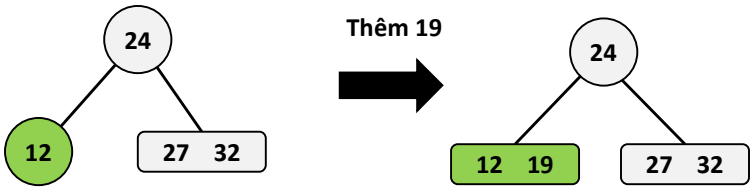
Cây 2-3

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

Cây 2-3

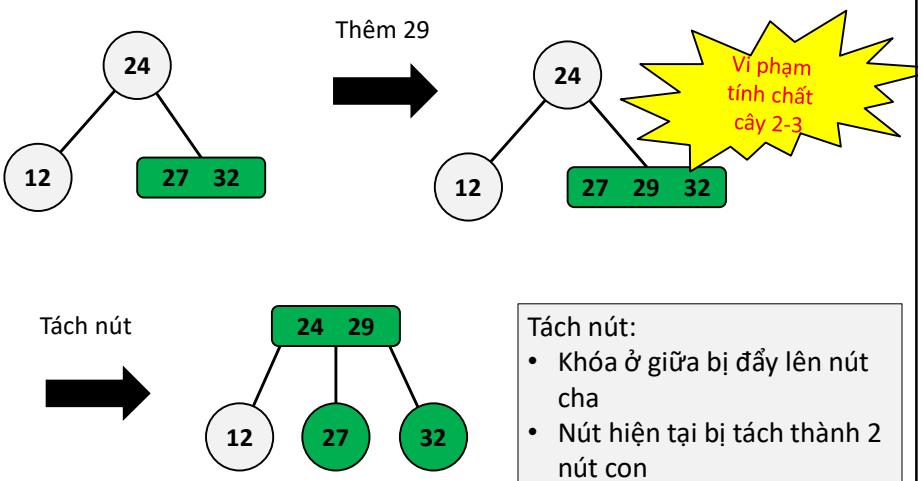
- Thêm nút
  - Phần tử mới được thêm vào tại nút lá của cây
  - Nếu nút lá sau khi thêm có 3 phần tử ta phải thực hiện thao tác tách nút
  - Khi tách nút, khóa giữa bị đẩy lên nút cha
  - Tách nút tại nút lá có thể dẫn đến tách nút tại nút trong

Cây 2-3



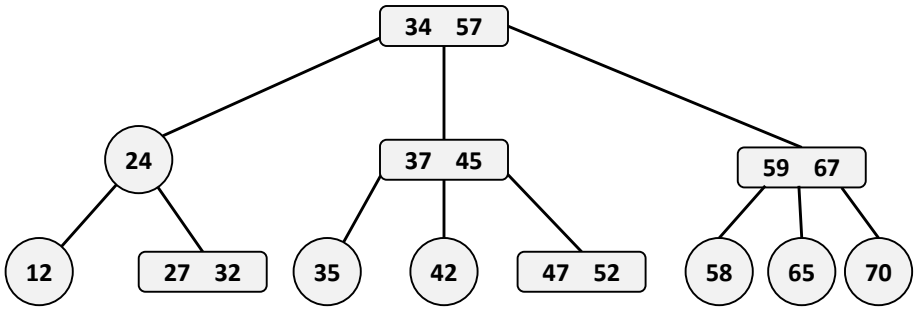
Không phải tách nút

Cây 2-3



Tách nút:
- Khóa ở giữa bị đẩy lên nút cha
- Nút hiện tại bị tách thành 2 nút con

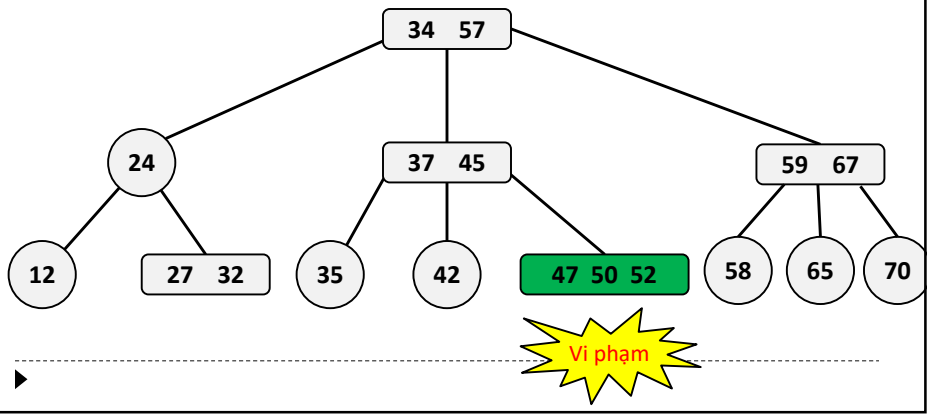
Cây 2-3



Thêm nút 50 ?

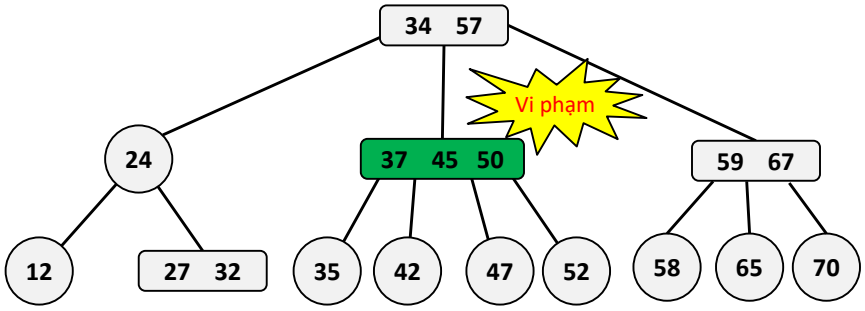
Cây 2-3

Thêm nút 50



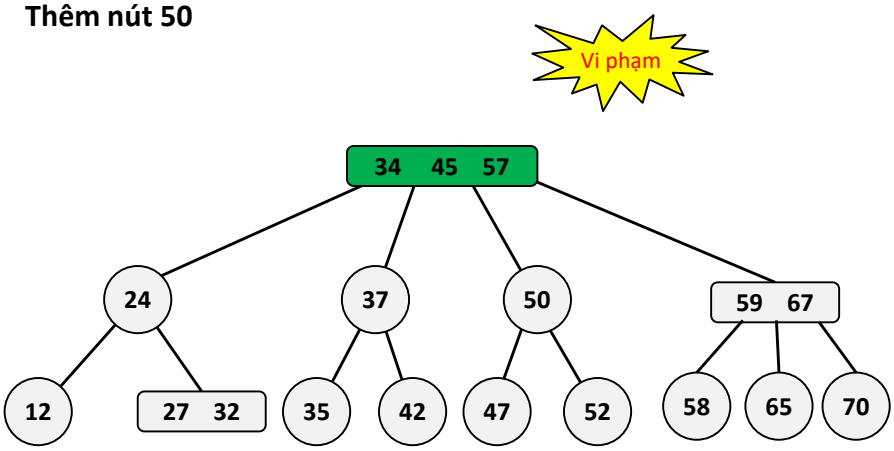
Cây 2-3

Thêm nút 50



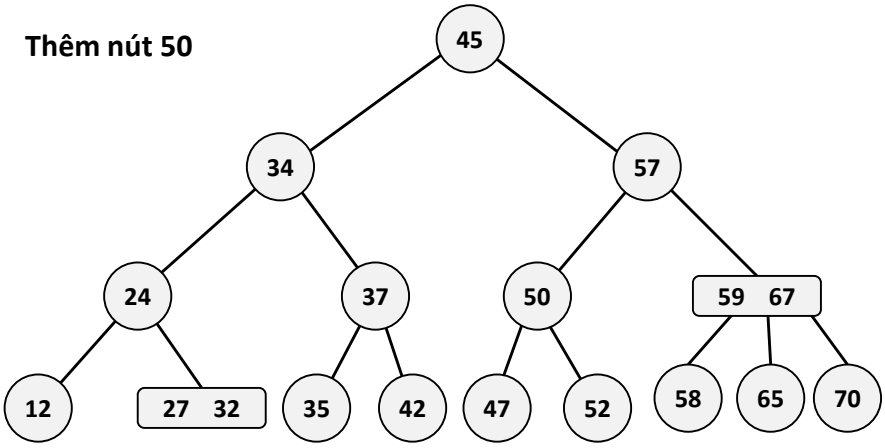
Cây 2-3

Thêm nút 50



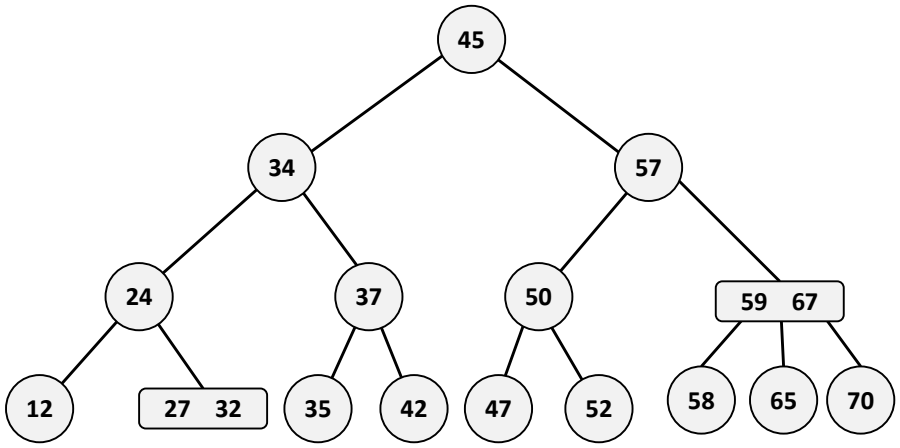
Cây 2-3

Thêm nút 50



Tách tại nút gốc

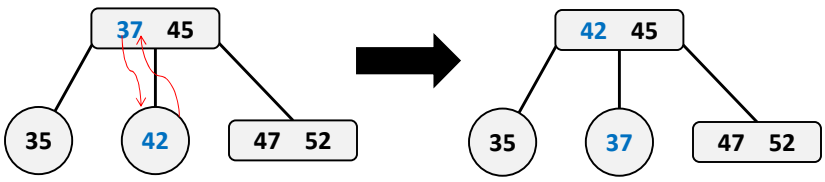
Cây 2-3



Vẽ cây thu được sau khi thêm lần lượt các khóa 5, 7, 29, 31, 68, 75 vào cây trên

Cây 2-3

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



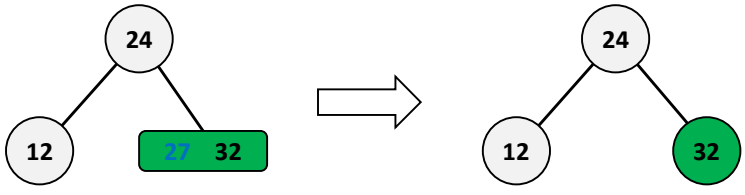
Xóa 37, ta thay bằng 42

Cây 2-3

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

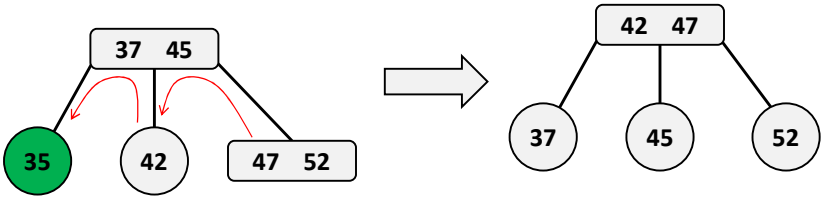
Cây 2-3

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



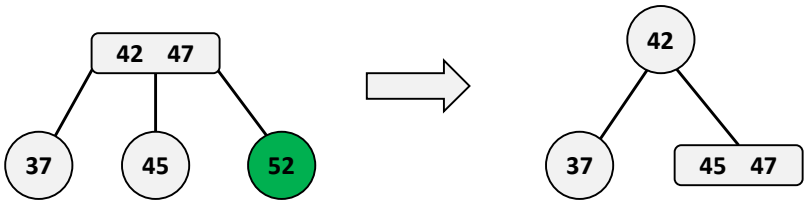
Cây 2-3

- Xóa 35 : trường hợp (i)



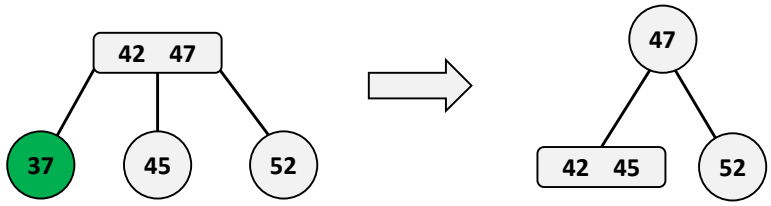
Cây 2-3

- Xóa 52: trường hợp (ii)



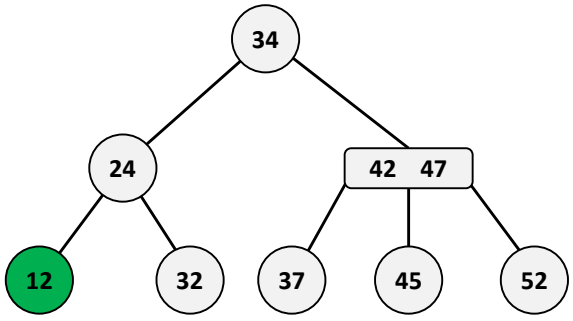
Cây 2-3

- Xóa 37: trường hợp (ii)



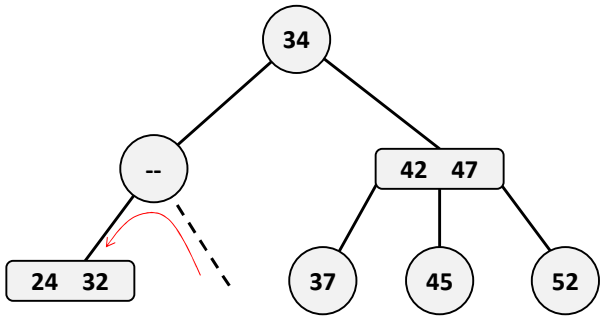
Cây 2-3

- Xóa 12: trường hợp (ii)



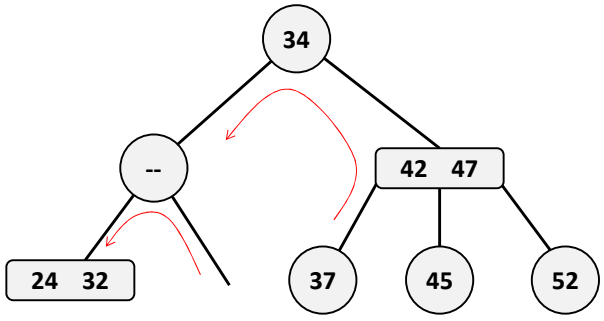
Cây 2-3

- Xóa 12: trường hợp (ii)



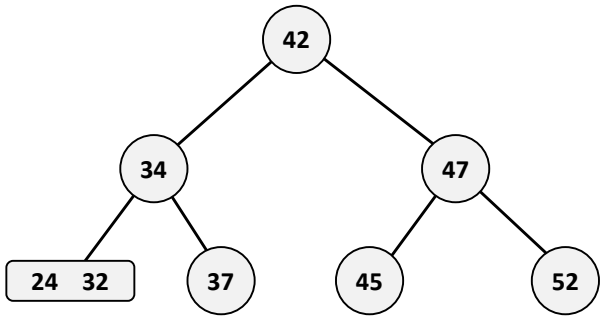
Cây 2-3

- Xóa 12: trường hợp (ii)



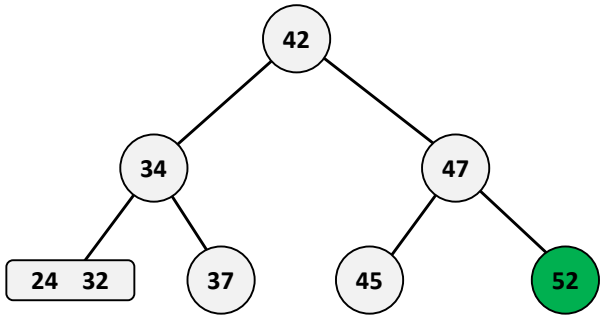
Cây 2-3

- Xóa 12: trường hợp (ii)



Cây 2-3

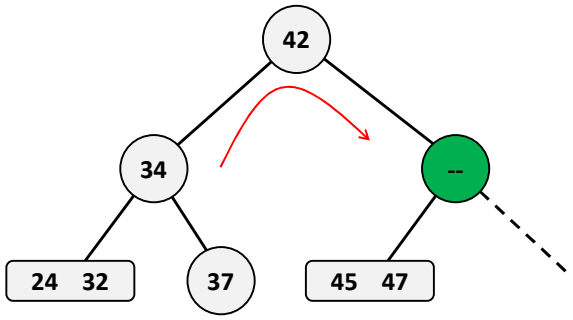
- Xóa 52:



Cây 2-3

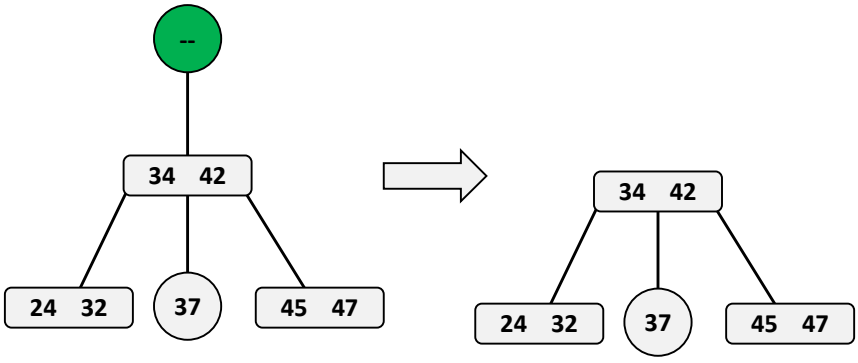
- Xóa 52:

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



Cây 2-3

- Xóa 52:



Cây 2-3

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

Question