Red Black

Binary Search Tree

CS523 Group 2
Red-Black Tree

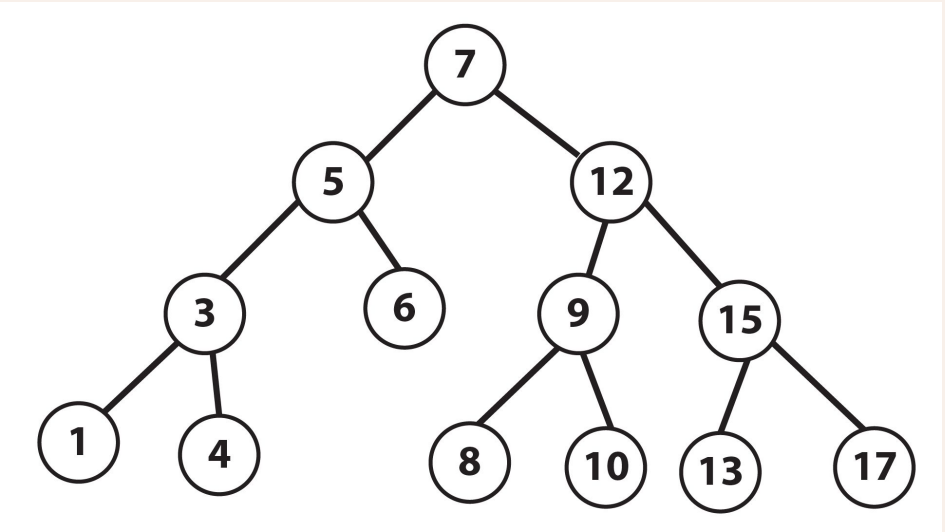
Nguyen Hoang Tan
Le Huynh Khanh Duy
Nguyen Ngoc Quang Huy

Table of contents

01

Introduction

Binary search tree

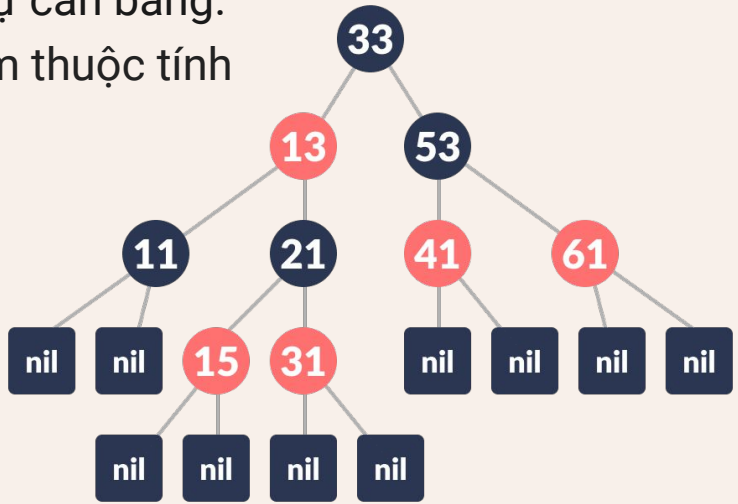


Search, Insert, Deletion

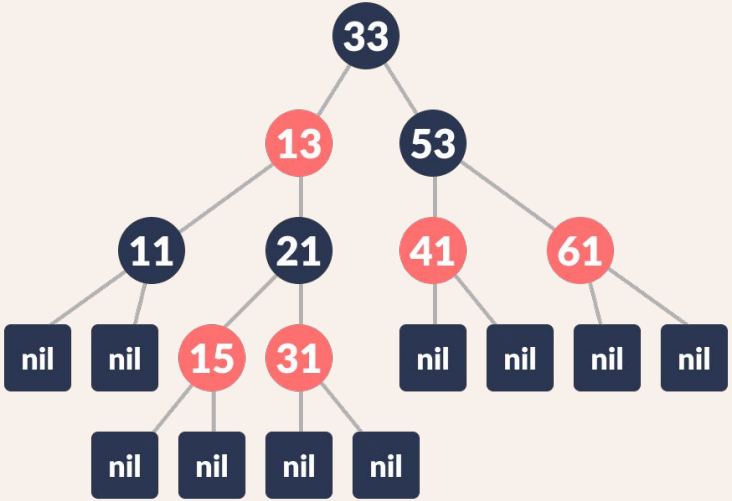
Binary search tree

What ?

- Red-Black Tree là cây nhị phân tìm kiếm tự cân bằng.
- Cấu trúc các node của cây đỏ đen có thêm thuộc tính để quy ước màu của node.



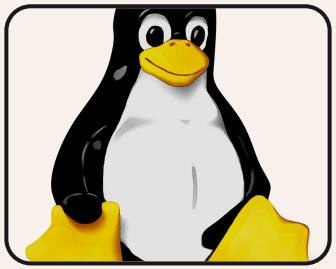
Why ?



- Nếu dữ liệu quá lớn thì việc cân bằng ở cây AVL sẽ mất rất nhiều công sức.
- Do đó cây đỏ đen ra đời để khắc phục yếu điểm này.

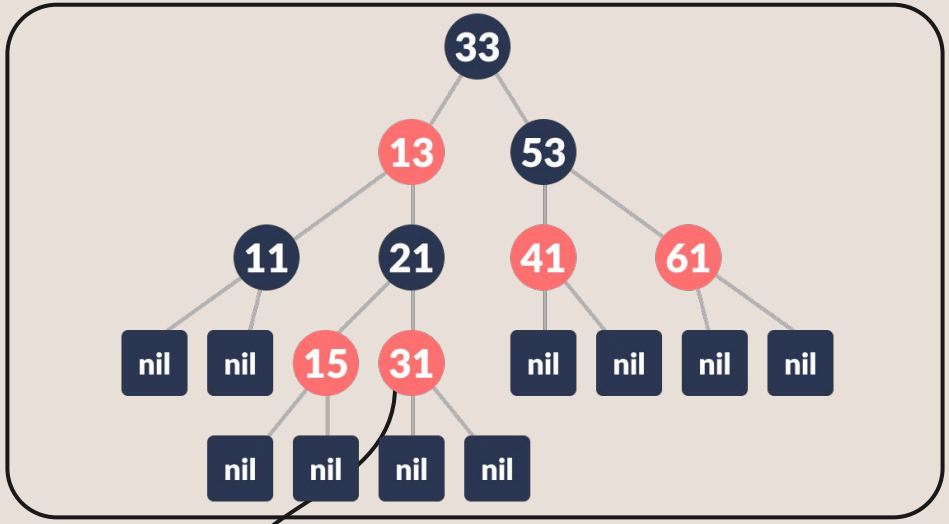
02 Application and demo

Application



Lập lịch tiến trình CPU cho hệ điều hành Linux.

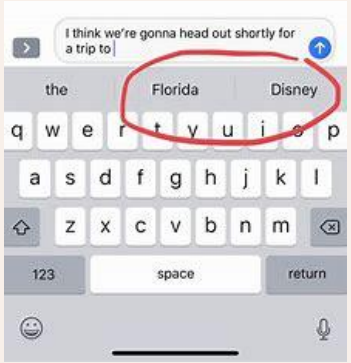
Áp dụng vào trong thuật toán phân cụm K-mean nhằm giảm độ phức tạp về thời gian.



- Hầu hết các chức năng của thư viện cây BST tự cân bằng trong C++ và Java đều sử dụng cấu trúc cây đỏ đen.

Red-Black Tree

- Bộ chỉ mục
- Bộ gõ dự đoán
- Ngôn ngữ tự động hóa thiết kế



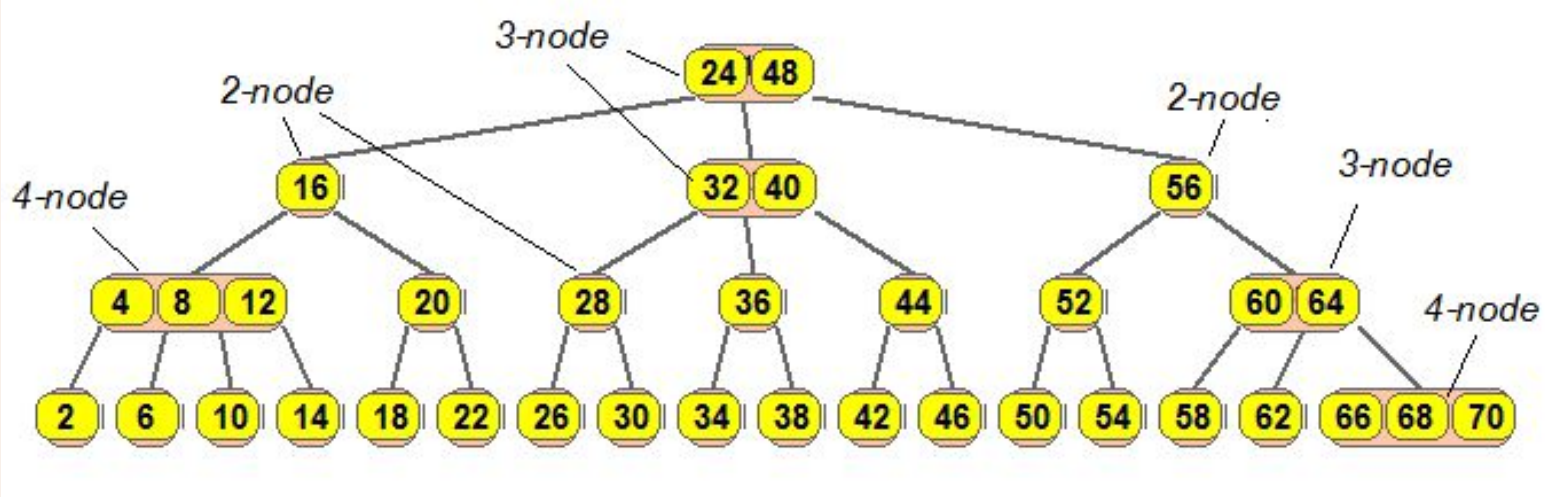
03

Properties

2-3-4 Tree

Node chứa 1,2 hoặc 3 keys và có tương ứng 2,3 hoặc 4 con

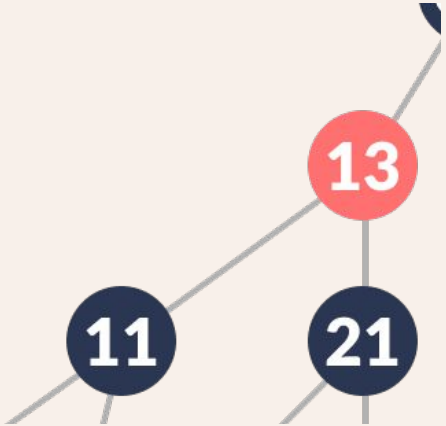
Mọi node lá đều có cùng chiều sâu



Properties of Red Black Tree:

Mọi node đều phải có màu đỏ hoặc đen.

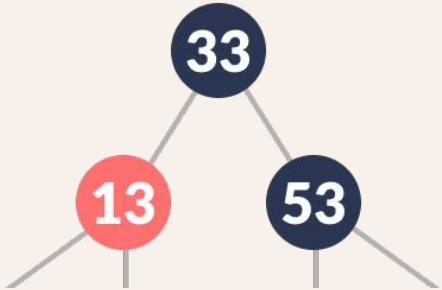
1



Properties of Red Black Tree:

Node gốc (root) luôn phải mang màu đen

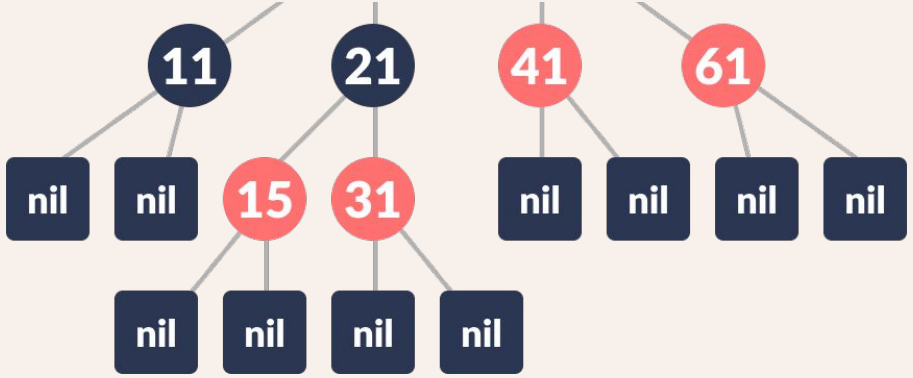
2



Properties of Red Black Tree:

Tất cả các node lá (NULL) đều màu đen.

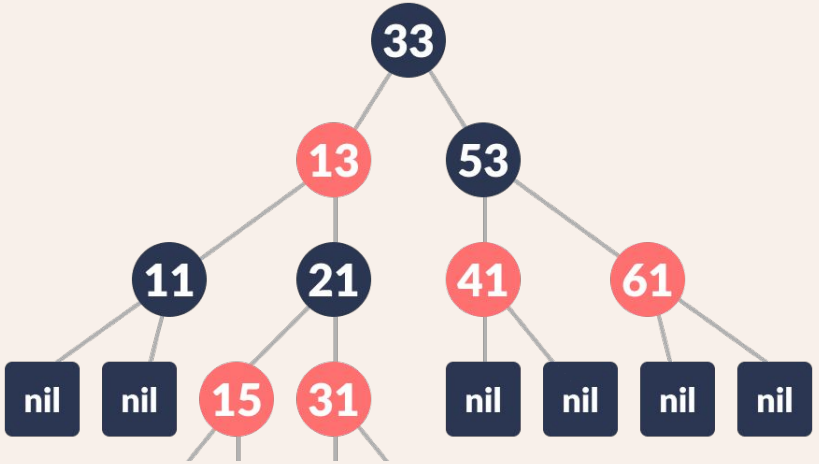
3



Properties of Red Black Tree:

Mọi đường dẫn từ một root đến NULL có cùng số lượng node đen.

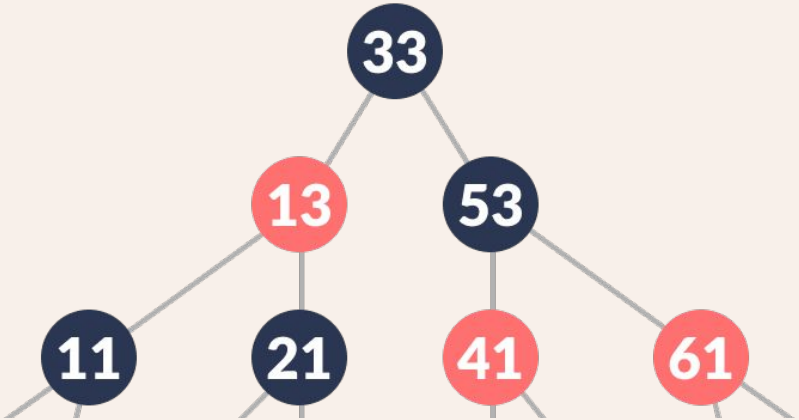
4



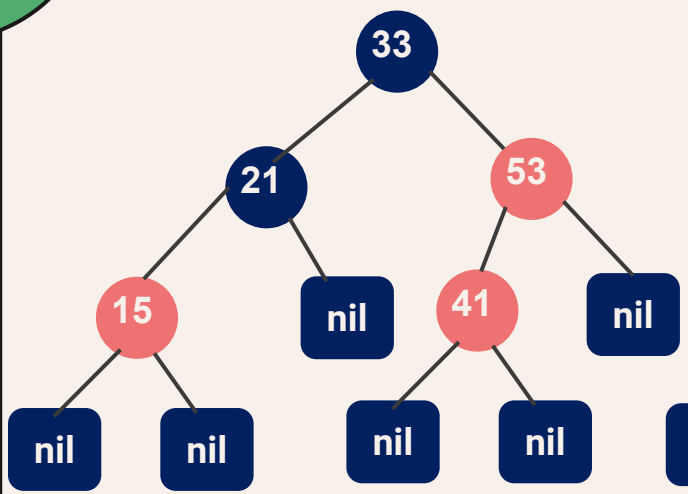
Properties of Red Black Tree:

Con của một node **đỏ** phải là một node **đen**

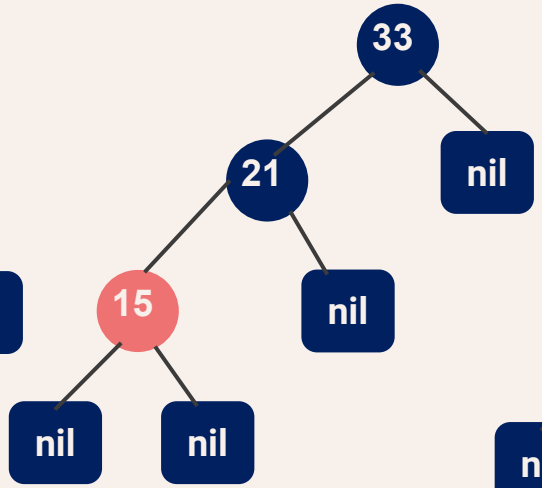
5



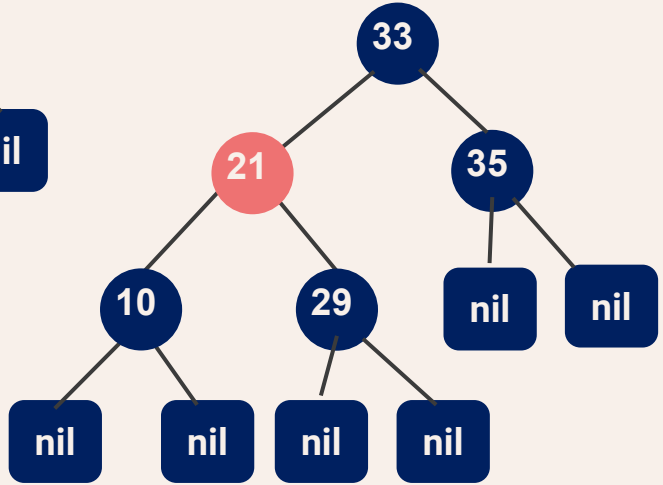
Which of these trees satisfies the properties of a Red-Black tree



A



B

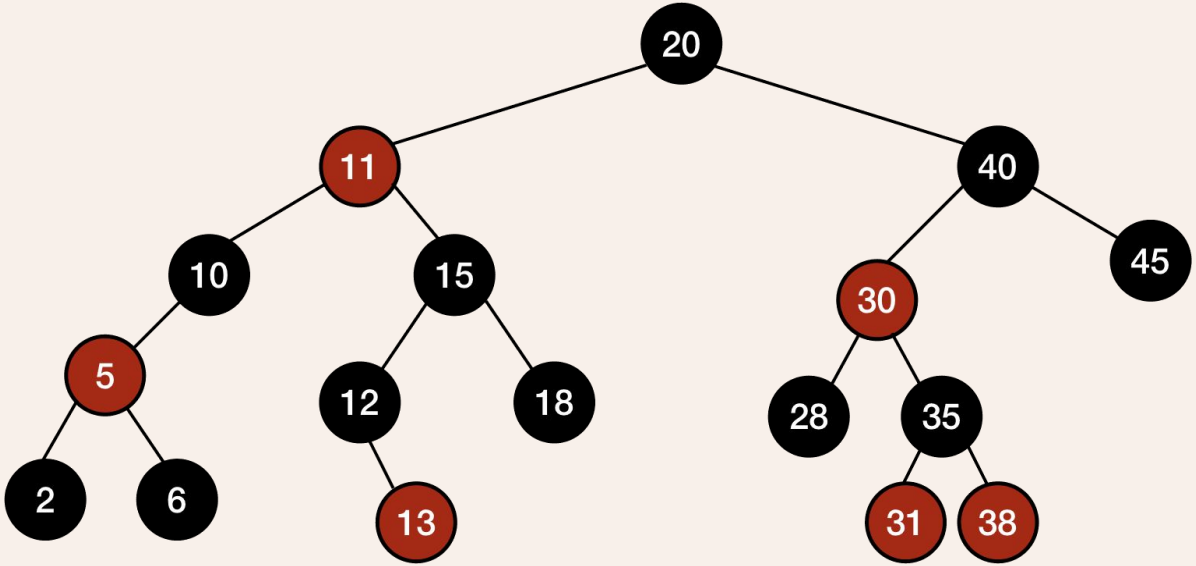


C

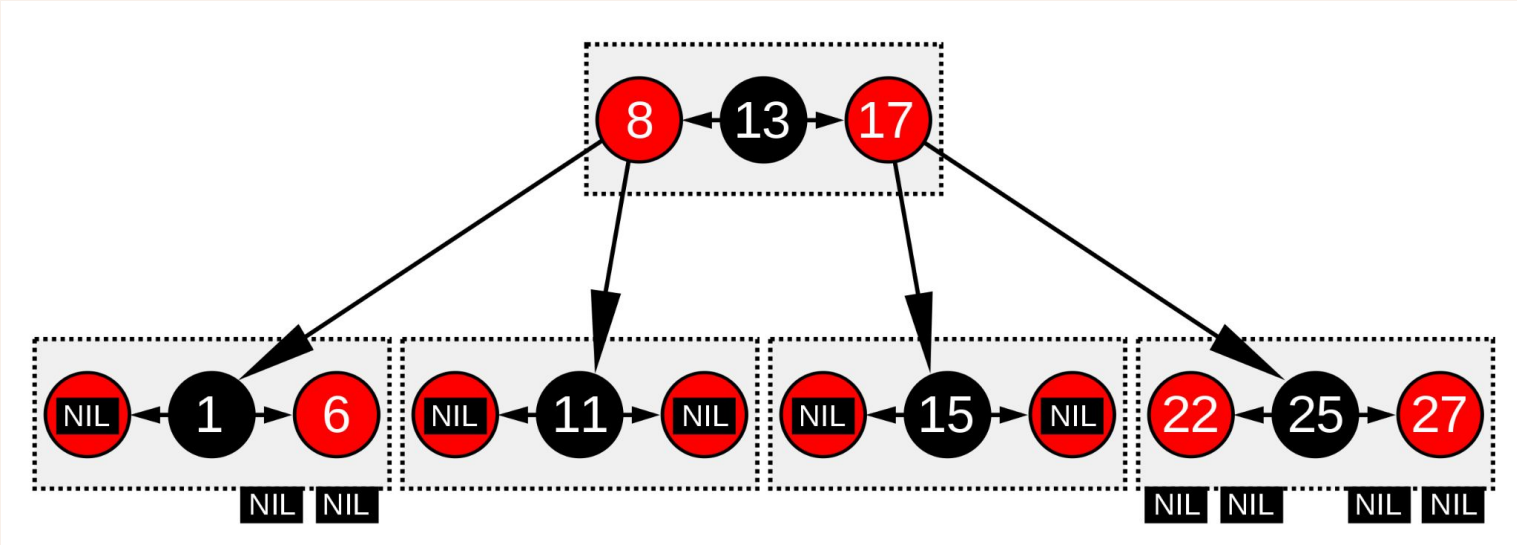
Max height

Red-Black tree đạt chiều cao tối đa khi cây ở trạng thái cân bằng hoàn hảo

Chiều cao tối đa của một cây RB có **n** nút là $2 * \log_2(\mathbf{n} + 1)$.



Analogy to B-trees of order 4



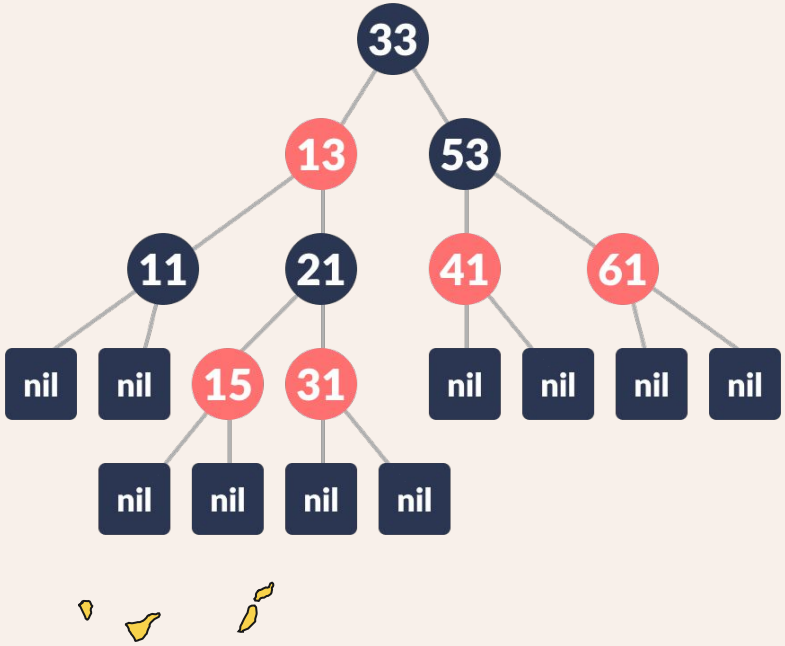
Data Structrue

- Cần phải biết node cha (parent) màu gì → thêm 1 con trỏ parent
- Có thuộc tính quy ước màu (color) cho node đó:
  - 1 – True: **màu đỏ**
  - 0 – False: **màu đen**

```
struct Node {

 int data;

 Node* left;

 Node* right;

 Node* parent;

 bool color;

};
```

04 Operations

Search Operation



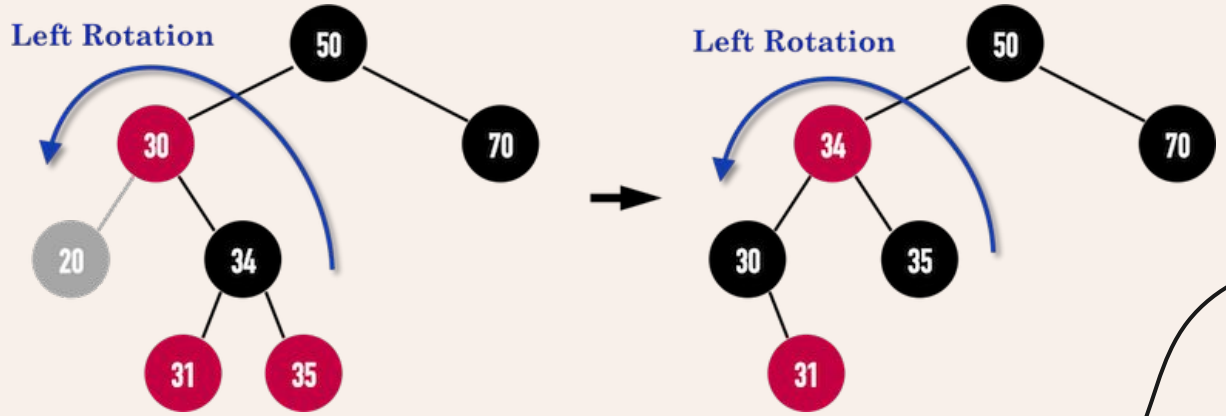
Là cây nhị phân tìm kiếm tự cân bằng
→ Các thao tác (tìm kiếm, in cây, tìm giá trị,...) hoàn toàn giống như cây BST bình thường

Complexity:
O(log n)

Insert Operation

Cây đỏ đen cũng có những phép quay như cây AVL, tuy nhiên sẽ hơi phức tạp hơn

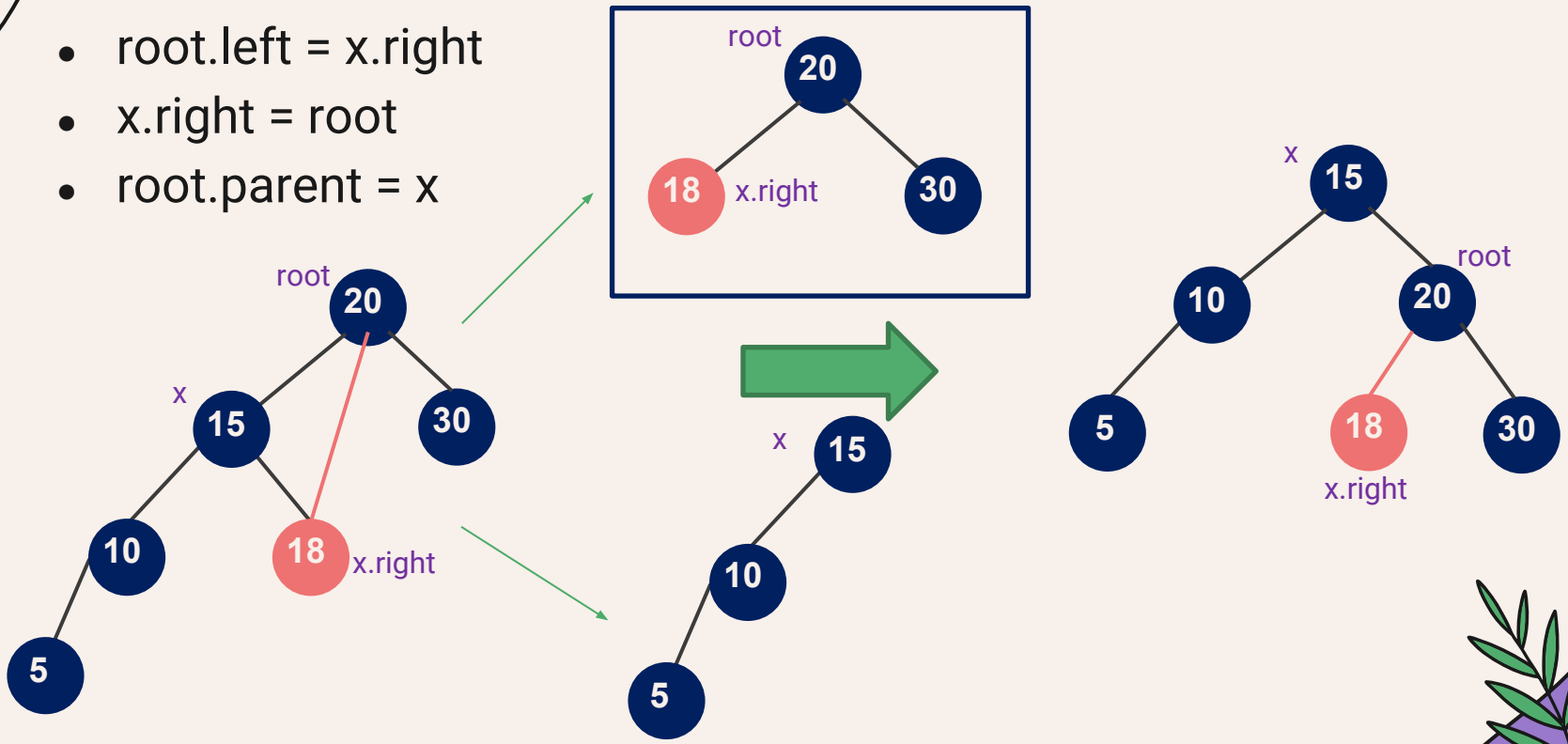
Rotate



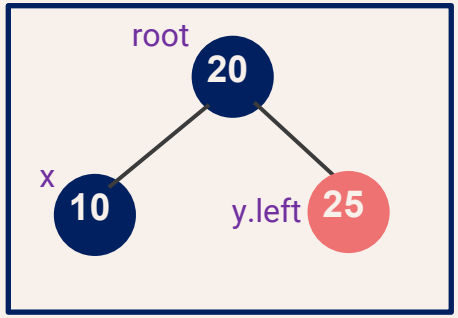
Complexity:
O(log n)

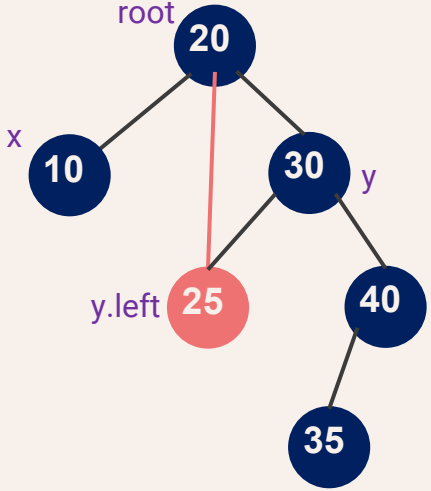
Rotate Right

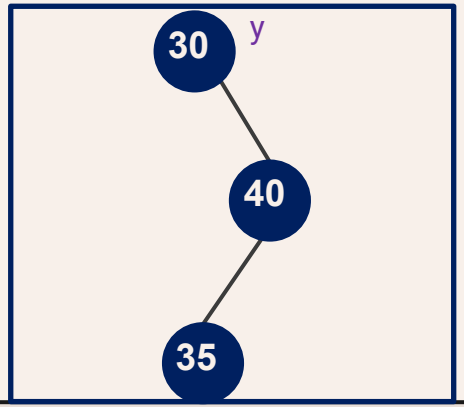
- root.left = x.right
- x.right = root
- root.parent = x

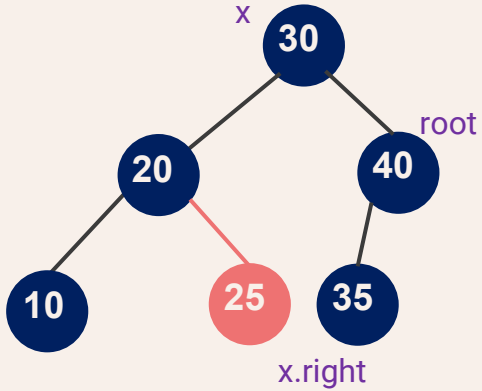


Rotate Left

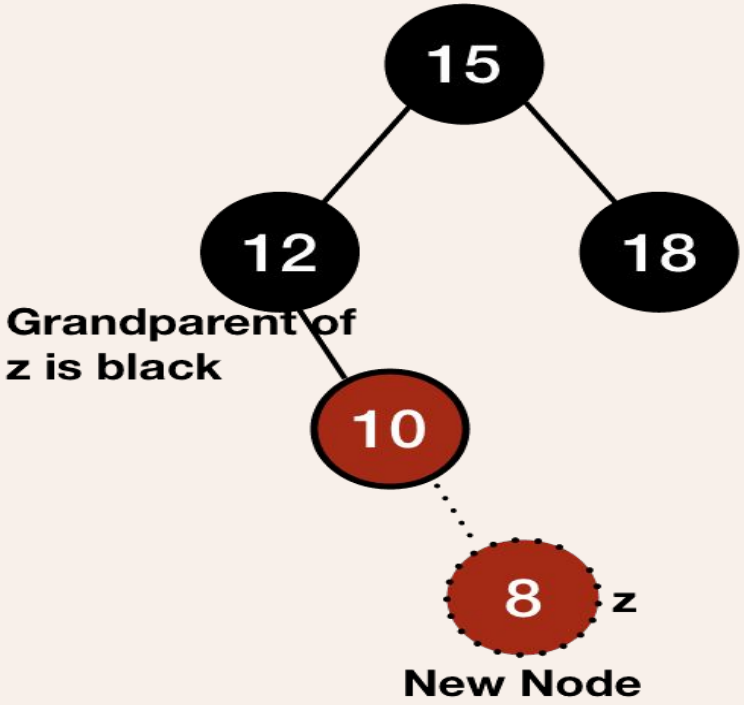

root
20
x
10
y.left
25


root
20
x
10
30
y
y.left
25
40
35


30
y
40
35


x
30
root
20
40
10
25
35
x.right

Conflict Resolver



- Các node mới được thêm vào luôn luôn là màu đỏ.
- Chỉ cần kiểm tra node cha, nếu mang màu đỏ thì chỉ cần xử lý xung đột đỏ.

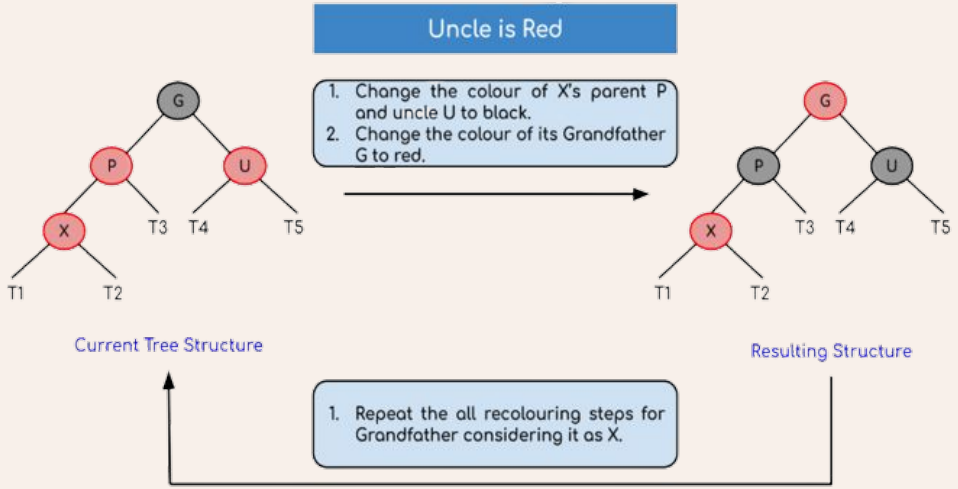
Conflict Resolver

- TH1: Cha đỏ - Chú đỏ:

  Node cha: đỏ → đen

  Node chú: đỏ → đen

  Node ông: đen → đỏ

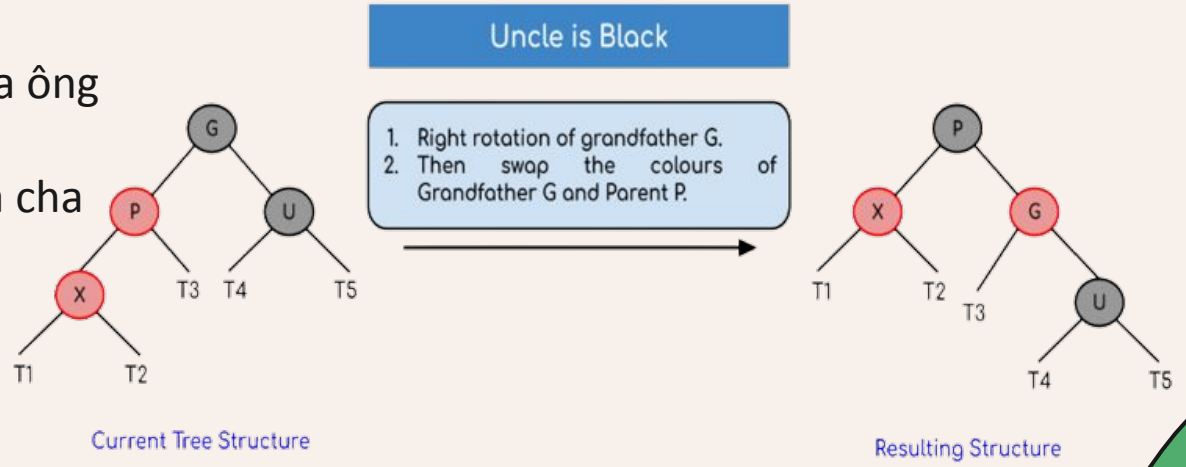


Conflict Resolver

- TH2: Cha đỏ - Chú đen:

Left Left Case (LL rotation):

- Node cha: là con trái của ông
- Node X : là con trái của cha



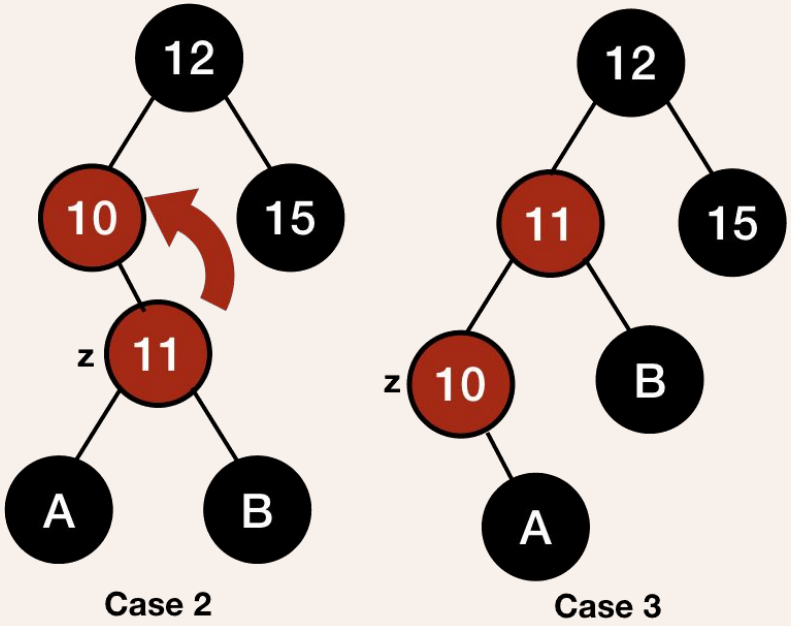
Conflict Resolver

- TH2: Cha đỏ - Chú đen:

Left Right Case (LR rotation):

Node cha: là con trái của ông

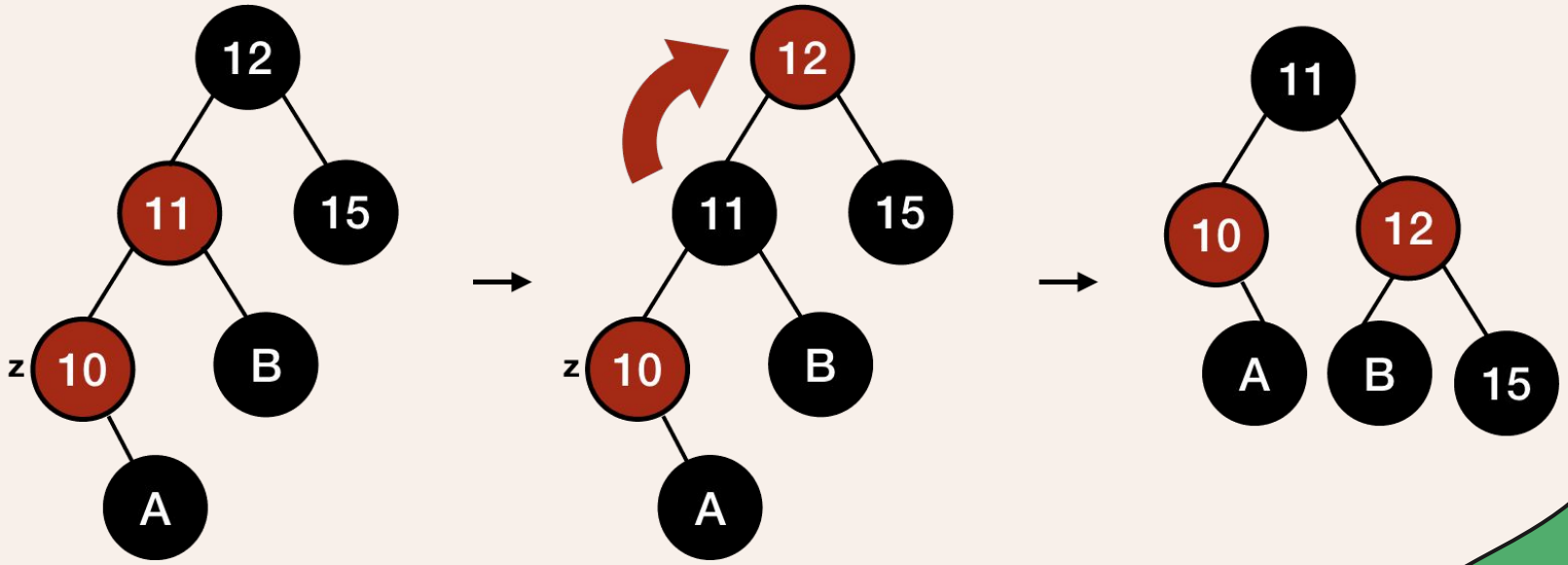
Node X : là con phải của cha



Conflict Resolver

- TH2: Cha đỏ - Chú đen:

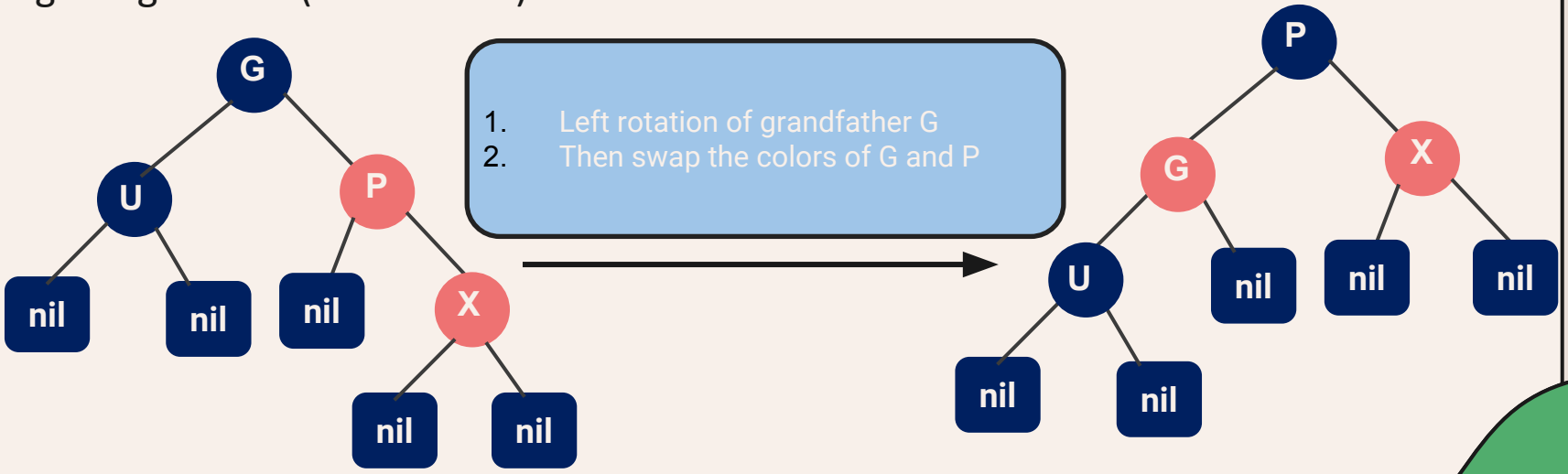
Left Right Case (LR rotation):



Conflict Resolver

- TH2: Cha đỏ - Chú đen:

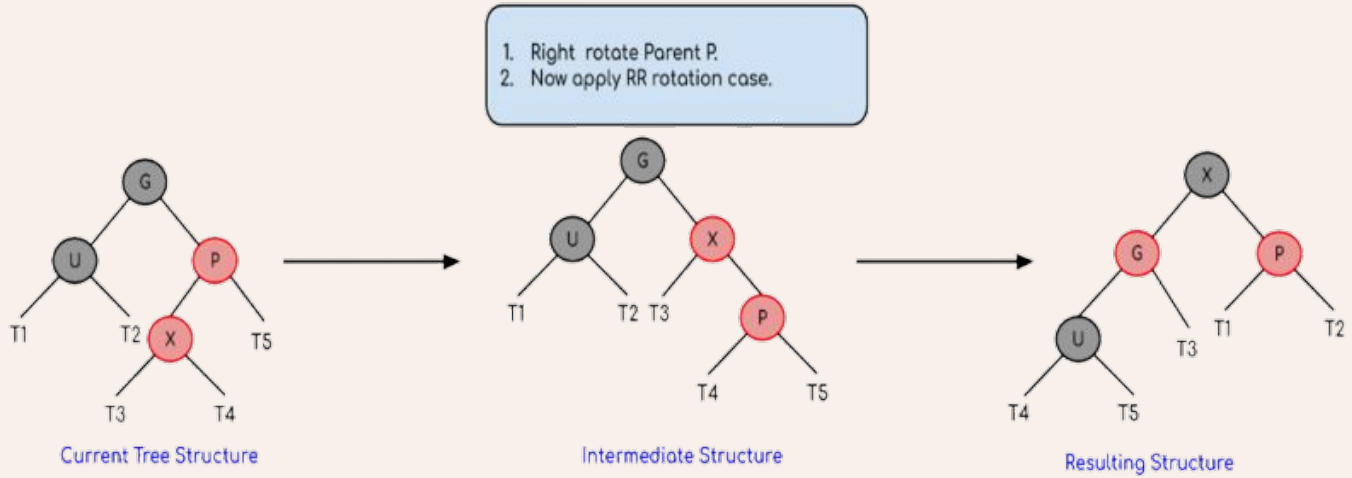
Right Right Case (RR rotation):



Conflict Resolver

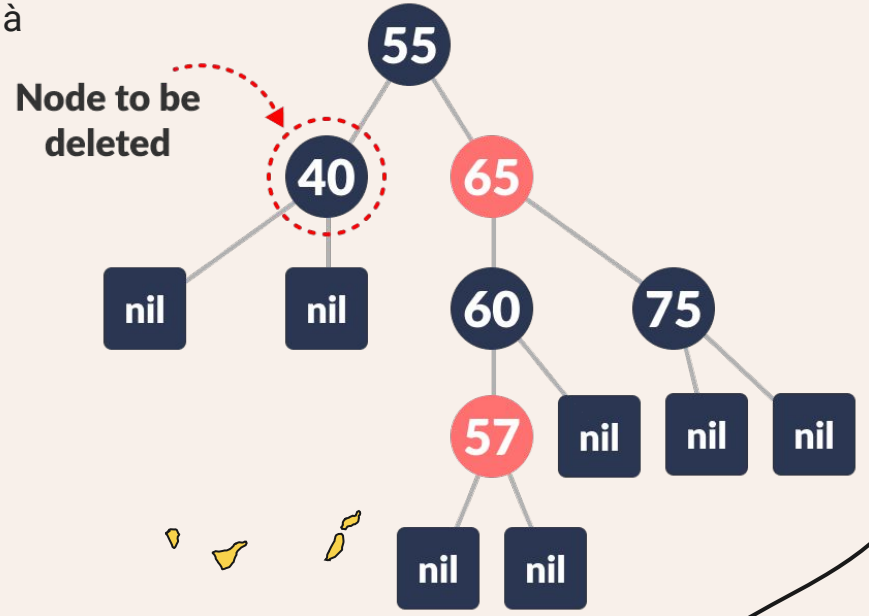
- TH2: Cha đỏ - Chú đen:

Right Left Case (RL rotation)



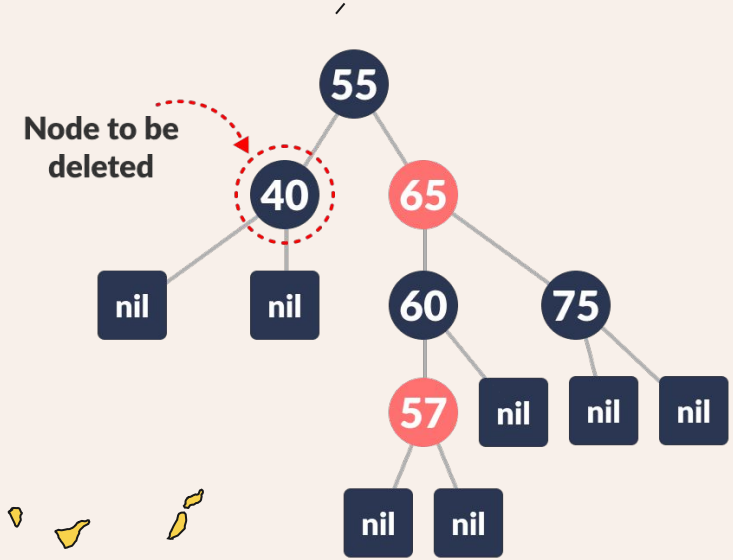
Delete Operation

- vDelete: Là node bị xóa (Đôi khi gọi tắt là Node v)
- uReplace: node sẽ thay thế vDelete



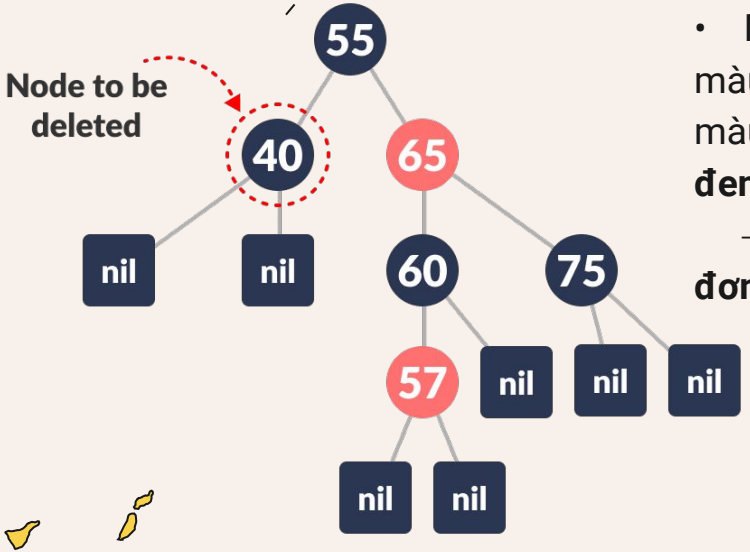
Complexity:
O(log n)

Delete Operation



- Khi xóa đi một Node đen → ít nhất một đường dẫn bị giảm số lượng Node đen đi 1.
- Nếu vDelete hoặc uReplace màu đỏ → xóa đi Node màu đỏ với giá trị data của vDelete.

Delete Operation

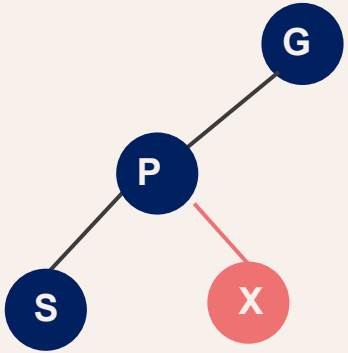


- Nếu vDelete và uReplace màu đen: Khi một nút màu đen bị xóa (v) và được thay thế bằng nút con màu đen (u), nút con đó (u) được đánh dấu là nút **đen đôi** .

→ Ta cần chuyển những nút **đen đôi** thành **đen đơn**

Fix Double Black

Ta phải dựa chia các trường hợp dựa trên Node Anh Em (sibling) của Node bị xóa (vDelete).



Fix Double Black

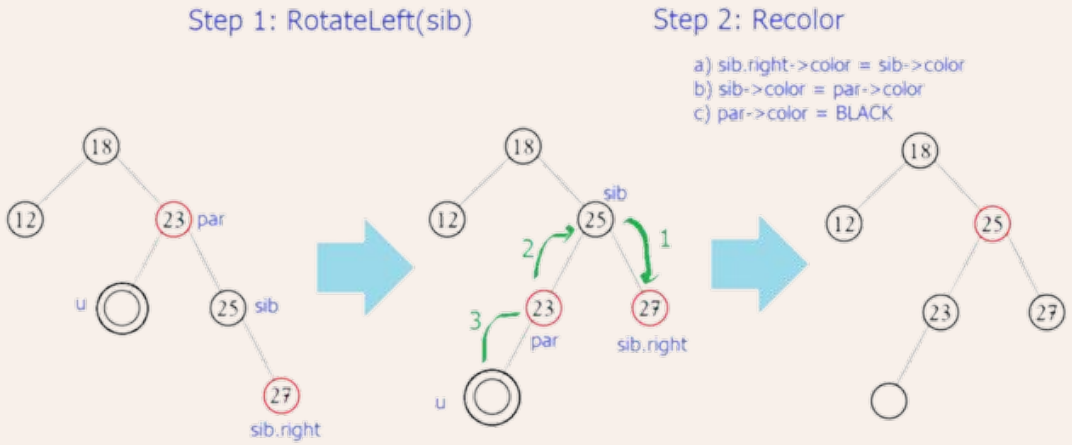
1. Node Anh Em màu đen và có con đỏ.

Dựa trên vị trí Node sibling và con đỏ của nó để phân chia các trường hợp con. Khá quen thuộc đó là Left left, Left right, Right right, Right left.

Fix Double Black

1. Node Anh Em màu đen và có con đỏ.

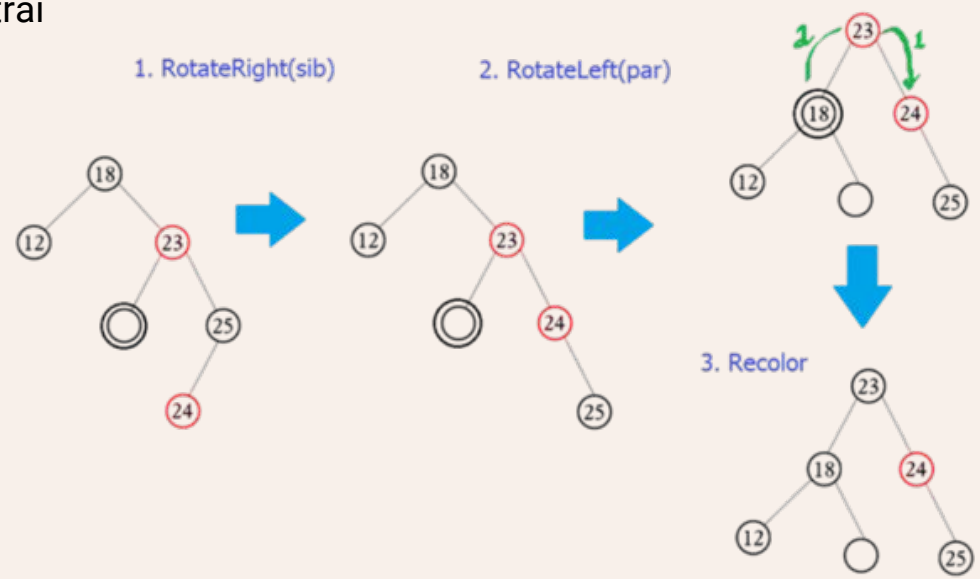
Right right case – Phải phải



Fix Double Black

1. Node Anh Em màu đen và có con đỏ.

Right left case – Phải trái



Fix Double Black

1. Node Anh Em màu đen và có con đỏ.

Left left case – Trái trái

Sib là con trái và
sib.left -> color == RED

- sib.left->color = sib.color
- sib->color = sib.parent->color
- par->color = BLACK
- rotateRight(par)

Fix Double Black

1. Node Anh Em màu đen và có con đỏ.

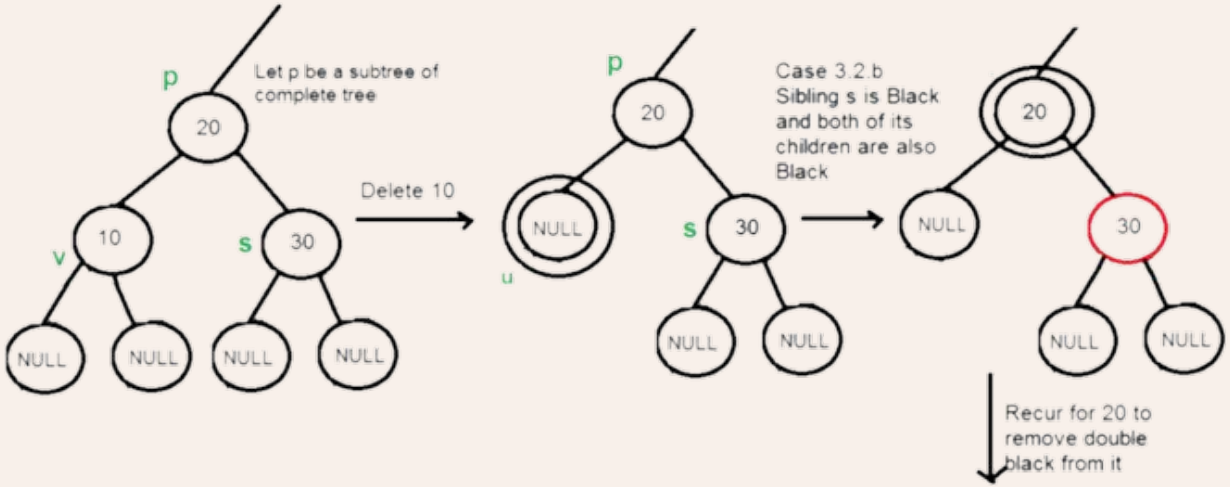
Left right case – Trái phải

sib là con trái và
sib.left->color == RED

- sib.right->color = par.color.
- par->color = BLACK
- rotateLeft(sib)
- rotateRight(par)

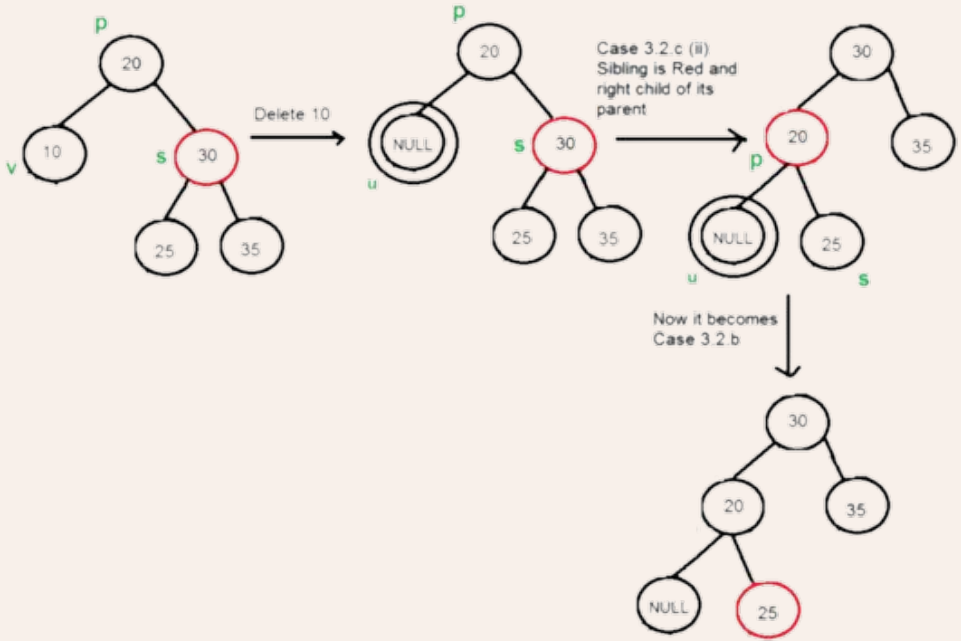
Fix Double Black

2. Node Anh Em màu đen và **không** có con đỏ.



Fix Double Black

3. Node Anh Em màu **đỏ**



Fix Double Black

3. Node Anh Em là node **NULL**

Chuyển vai trò DoubleBlack của u sang cho Node cha. Rồi FixDoubleBlack(par) là xong

05

Pros and cons

Pros

- Khả năng cân bằng tương đối tốt
- Độ phức tạp luôn là O(log n)
- Có thể sử dụng linh hoạt trong nhiều ứng dụng và tình huống khác nhau
- Sử dụng dễ dàng mà vẫn đảm bảo hiệu suất
- Trong một vài trường hợp cụ thể sẽ hoạt động cực kì nhanh

Cons

- Tốn nhiều không gian bộ nhớ hơn
- Có thể sẽ mất nhiều thời gian để cân bằng hơn so với các cây khác
- Việc tự cân bằng phải trả giá bằng chi phí bổ sung
- Mặc dù cho hiệu suất tốt ở các trường hợp trung bình nhưng trong trường hợp tệ nhất có thể sẽ chậm so với các cấu trúc dữ liệu khác

	AVL	RB Tree
• Độ cân bằng	Cân bằng tuyệt đối	Cân bằng tương đối
• Thời gian thực thi	Thường chậm hơn	Thường nhanh hơn
• Không gian lưu trữ	Nhiều hơn	Ít hơn
• Triển khai và sử dụng	Khó hơn	Dễ hơn

Thanks!

Group 2