RED BLACK TREE

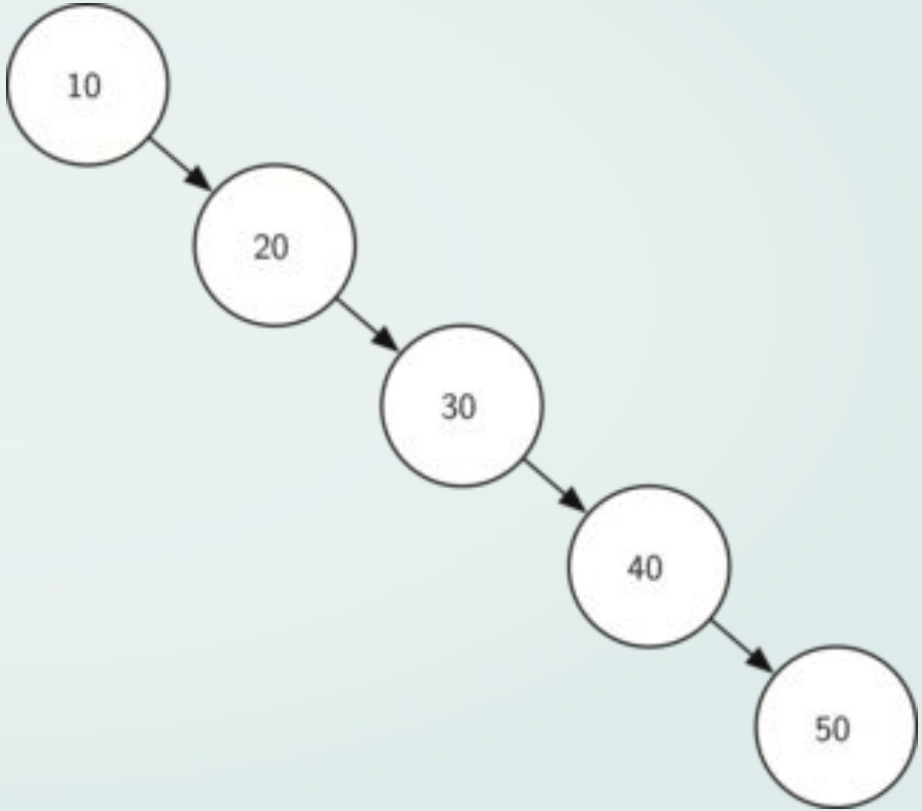
Cây đỏ đen

Nhóm 08
20521924 Trần Thành
19521485 Phạm Phúc Hậu
GVHD: Nguyễn Thanh Sơn

Mục lục

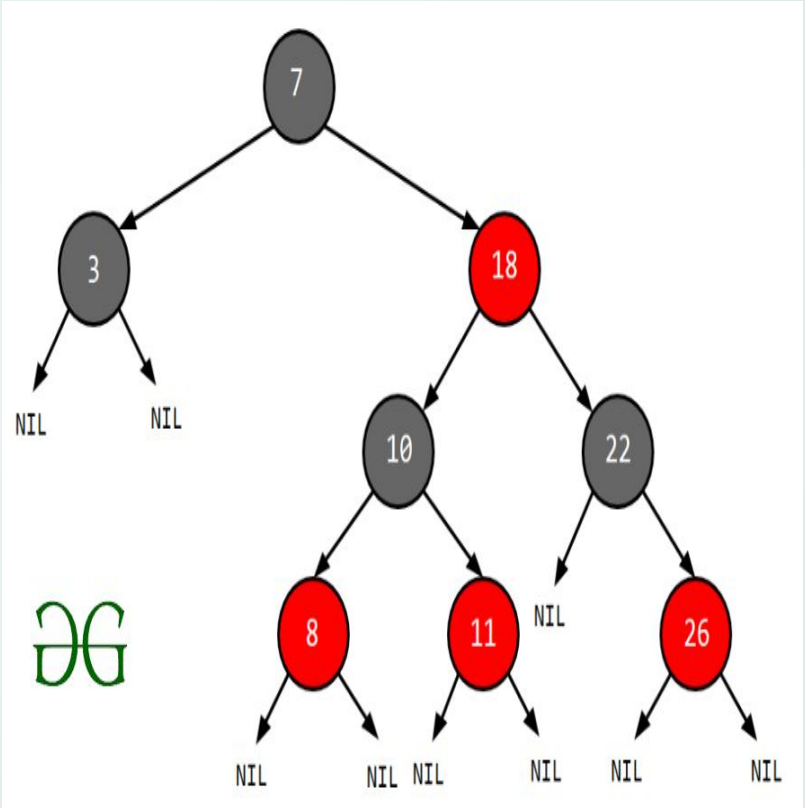
Giới thiệu

- Dữ liệu chèn vào đã được sắp xếp => Cây nhị phân bị lệch.
- Cây bị lệch => Khả năng tìm kiếm nhanh, xóa, sửa=> Bị hạn chế.
- Cây không cân bằng :
  - Giải quyết: **Cây đỏ đen**



Tổng quan

- Cây đỏ đen là một biến thể của cây nhị phân tìm kiếm mà đảm bảo rằng cây sẽ được ***cân bằng***.
  - Chiều cao của cây đỏ đen với n node là h<=2log(n+1)~log(n) [2]
- Các thao tác trên cây có độ phức tạp O(lg n) trong trường hợp xấu nhất

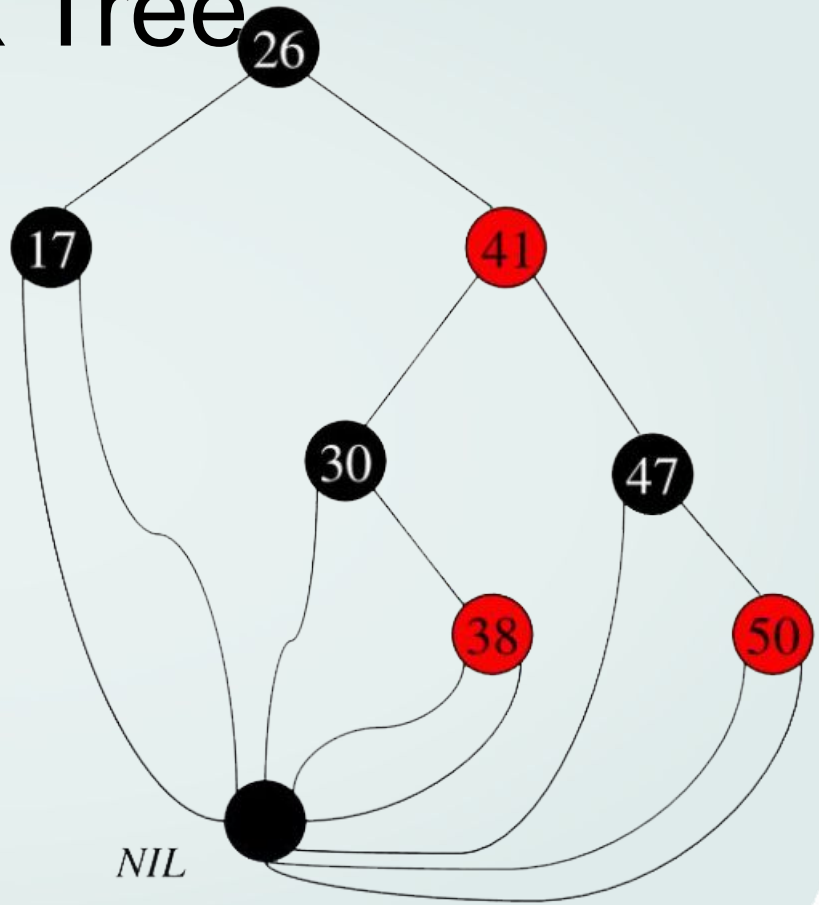


Ứng dụng

- **Database Engines**: Việc lập chỉ mục trong cơ sở dữ liệu sử dụng cây RB trực tiếp hoặc gián tiếp.
- **Machine Learning**: K-mean clustering algorithm
- **Linux Kernel**: Completely Fair Scheduler
- **Computational Geometry Data structures**
- **Sử dụng trong thư viện C++:** multiset, map, multimap.
- **Các gói package trong java bao gồm**: java.util.TreeMap và java.util.TreeSet.

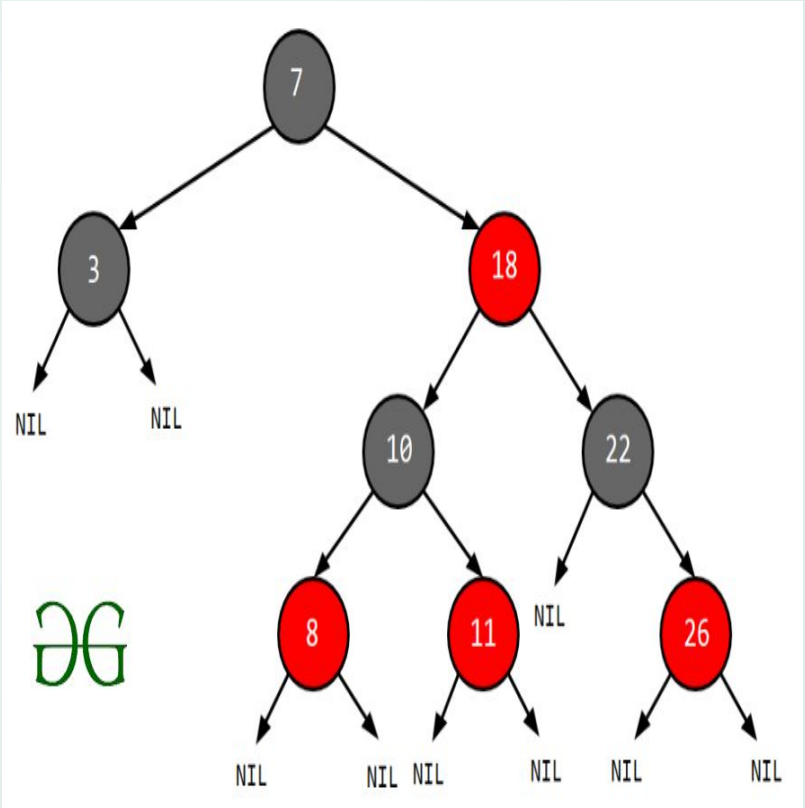
Red-black Tree

- Thêm 1 bit vào mỗi node của cây nhị phân để biểu diễn thuộc tính color có giá trị là đỏ hoặc đen.
- Có tất cả thuộc tính của cây nhị phân tìm kiếm.



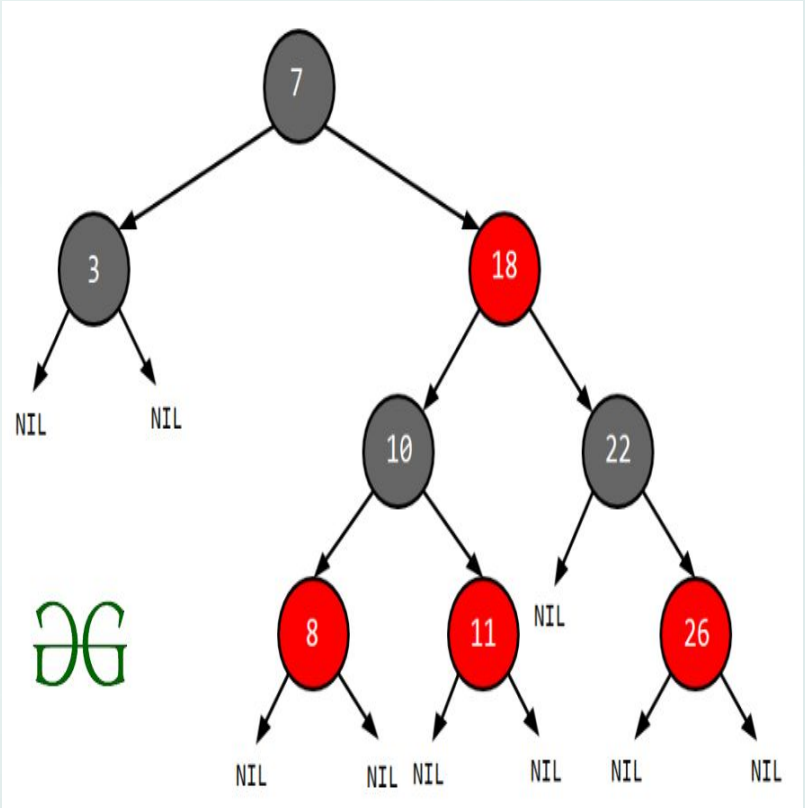
Cấu trúc của 1 Node

- Information: Thông tin của node.
- Color(Bool) Màu của node.
- Node* parent: Con trỏ đến node cha của nó
  - Node* left: Con trỏ đến node con trái.
  - Node* right: Con trỏ đến node con phải.



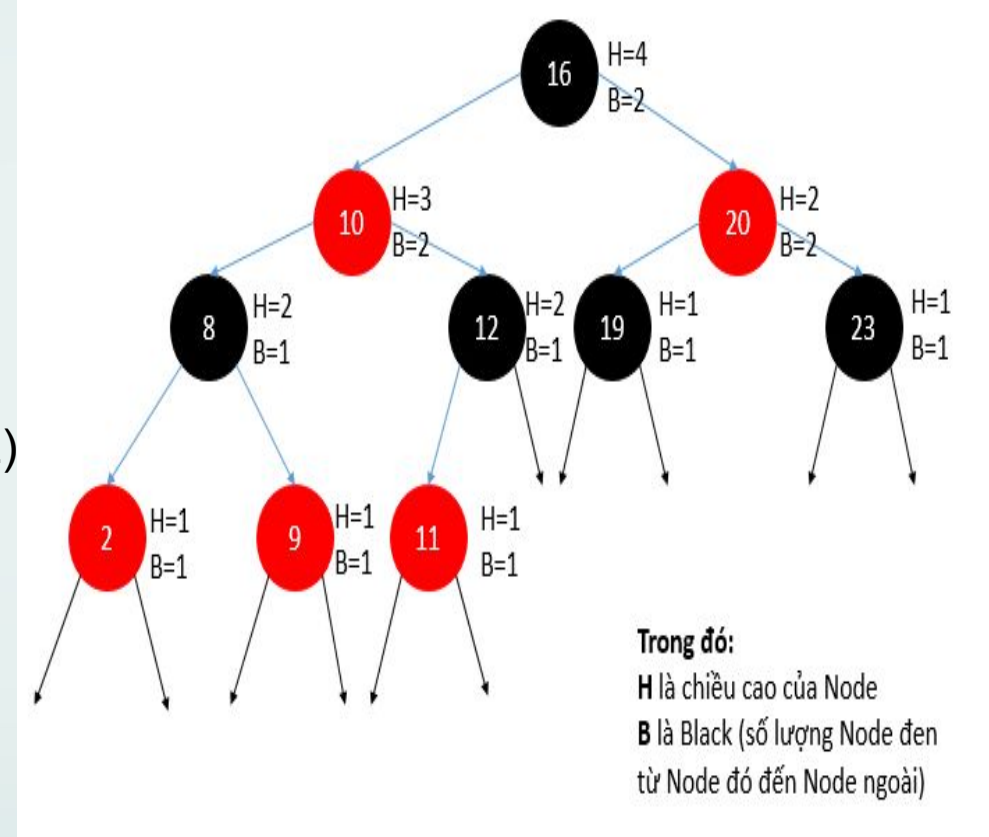
Thuộc tính cây đỏ đen

1. Mọi nút đều có màu đỏ hoặc đen.
2. Nút gốc và nút lá (NIL) phải luôn luôn đen.
3. Nếu nút màu đỏ có các nút con thì các nút con luôn có màu đen.
4. Đối với mỗi nút, bất kỳ đường đi nào từ nút này đến bất kỳ nút lá con nào của nó đều có cùng chiều cao đen màu đen (số lượng nút màu đen).



Chiều cao cây đỏ đen

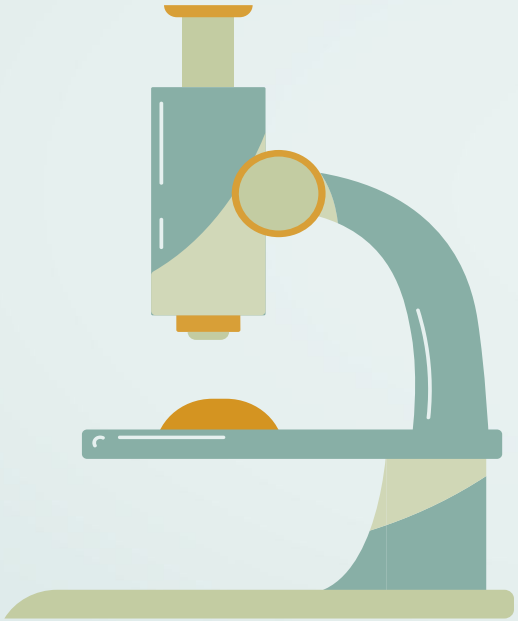
- Chiều cao h của node x: H(x)
  - Số cạnh của đường dài nhất đến nút lá.
- Chiều cao đen(Black-height) của một node x.
  - BH(x): số lượng node đen(bao gồm NIL) trên đường từ node x tới node lá(không bao gồm node đang x).
- $BH(x) \le H(x) \le 2\,BH(x)$ [2]

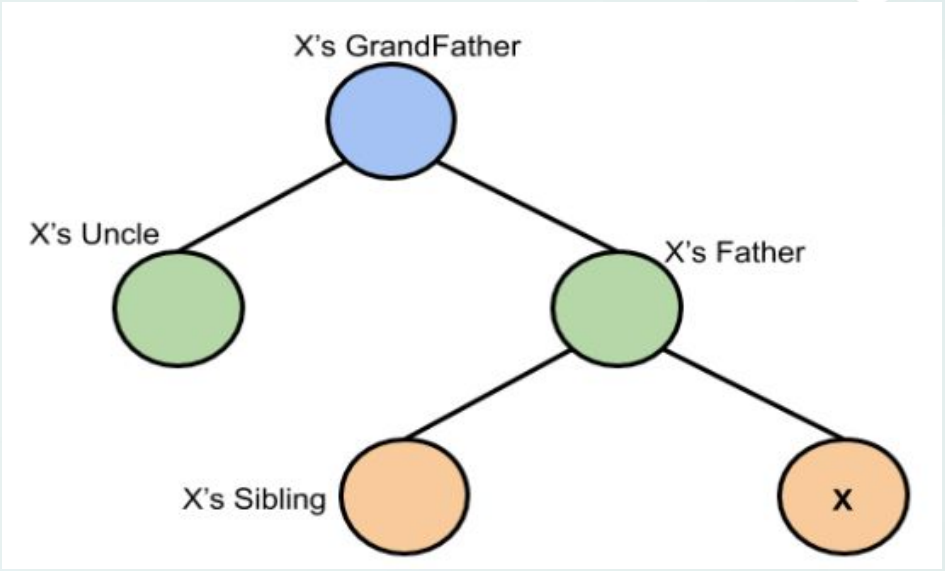


Thao tác trên cây

03

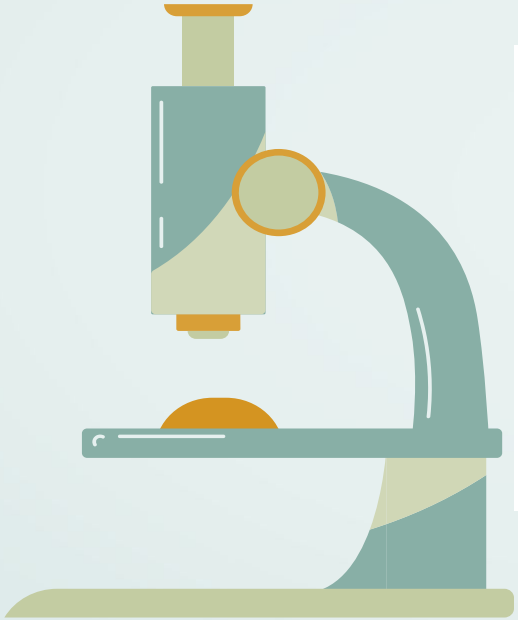
Thao tác trên cây

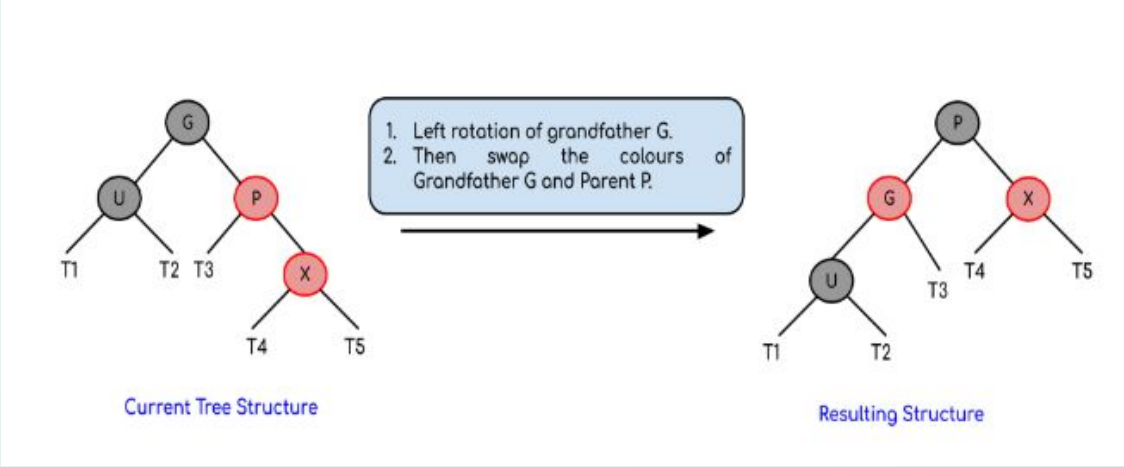




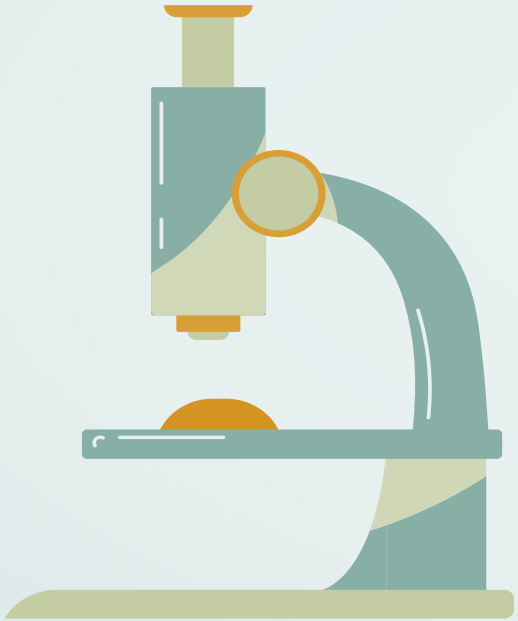
Thao tác trên cây

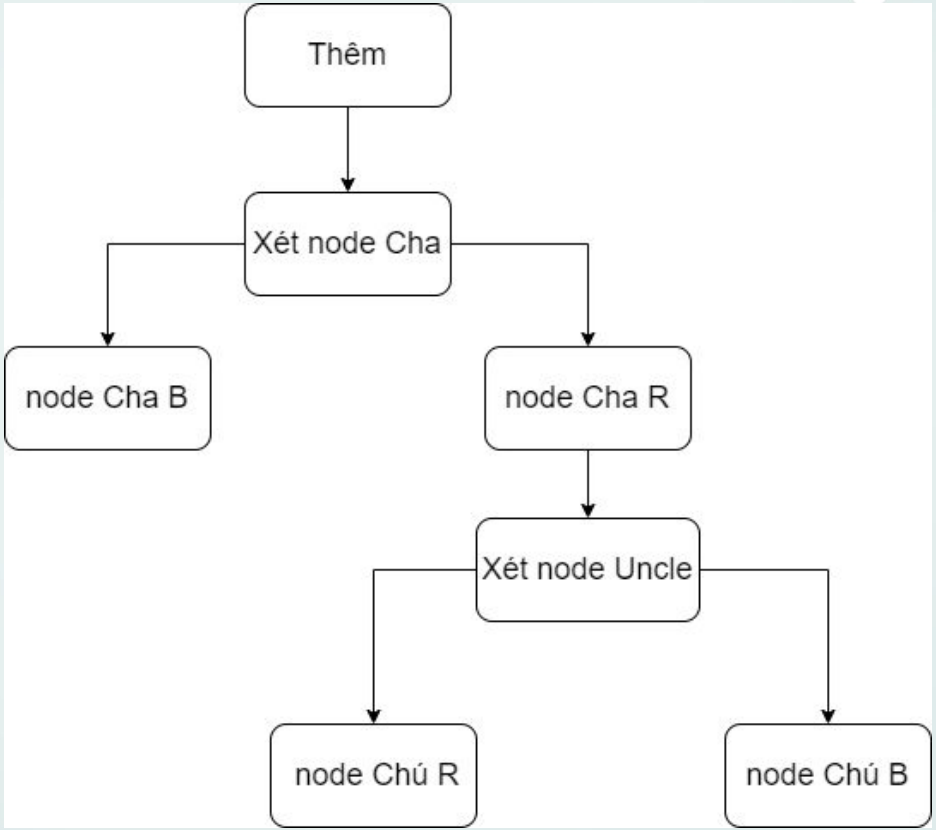
Sử dụng 2 công cụ: Recoloring and Rotation





Thao tác trên cây





Thao tác trên cây

Algorithm:

Let x be the newly inserted node.

1. Perform standard BST insertion and make the colour of newly inserted nodes as RED.
2. If x is the root, change the colour of x as BLACK (Black height of complete tree increases by 1).
3. Do the following if the color of x's parent is not BLACK **and** x is not the root.

   a) If x's uncle is RED (Grandparent must have been black from property 4)

   (i) Change the colour of parent and uncle as BLACK.

   (ii) Colour of a grandparent as RED.

   (iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.

   b) If x's uncle is BLACK, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**) (This is similar to AVL Tree)

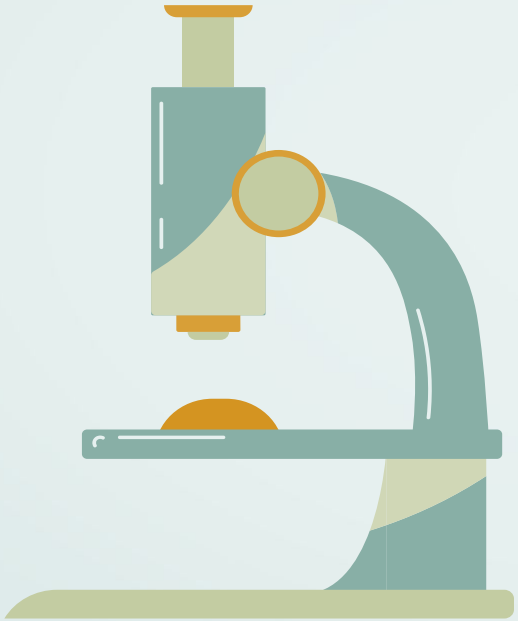
   (i) Left Left Case (p is left child of g and x is left child of p)

   (ii) Left Right Case (p is left child of g and x is the right child of p)

   (iii) Right Right Case (Mirror of case i)

   (iv) Right Left Case (Mirror of case ii)

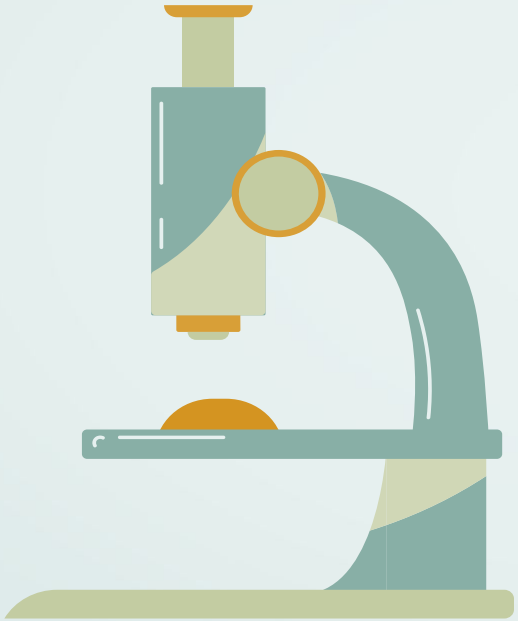
Thao tác trên cây

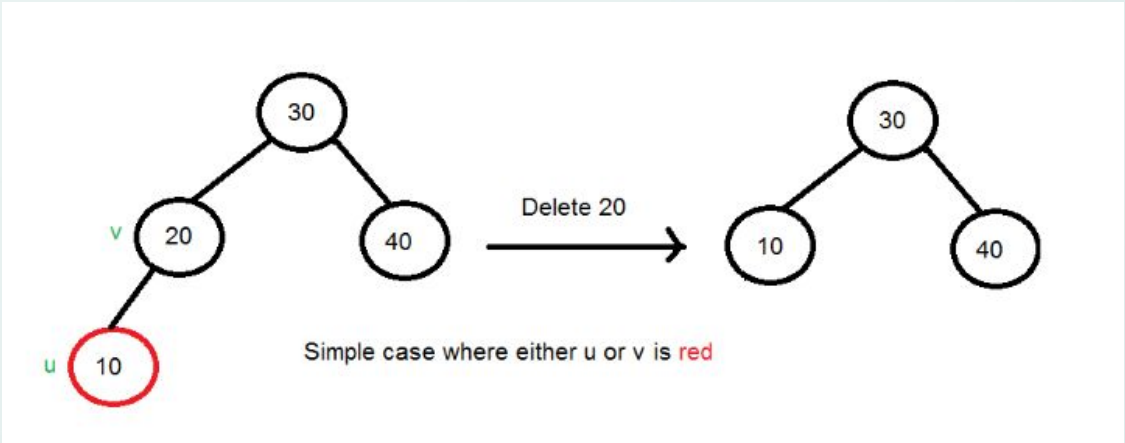


Thêm lần lượt: 3, 21, 32, 15, 12 vào một cây đỏ đen

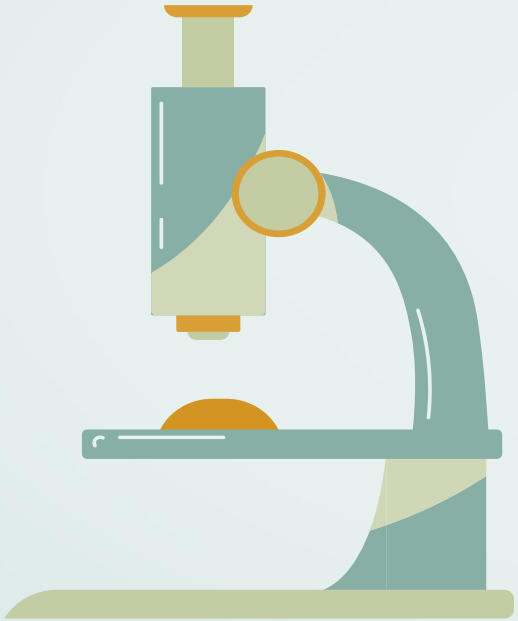
Thao tác trên cây

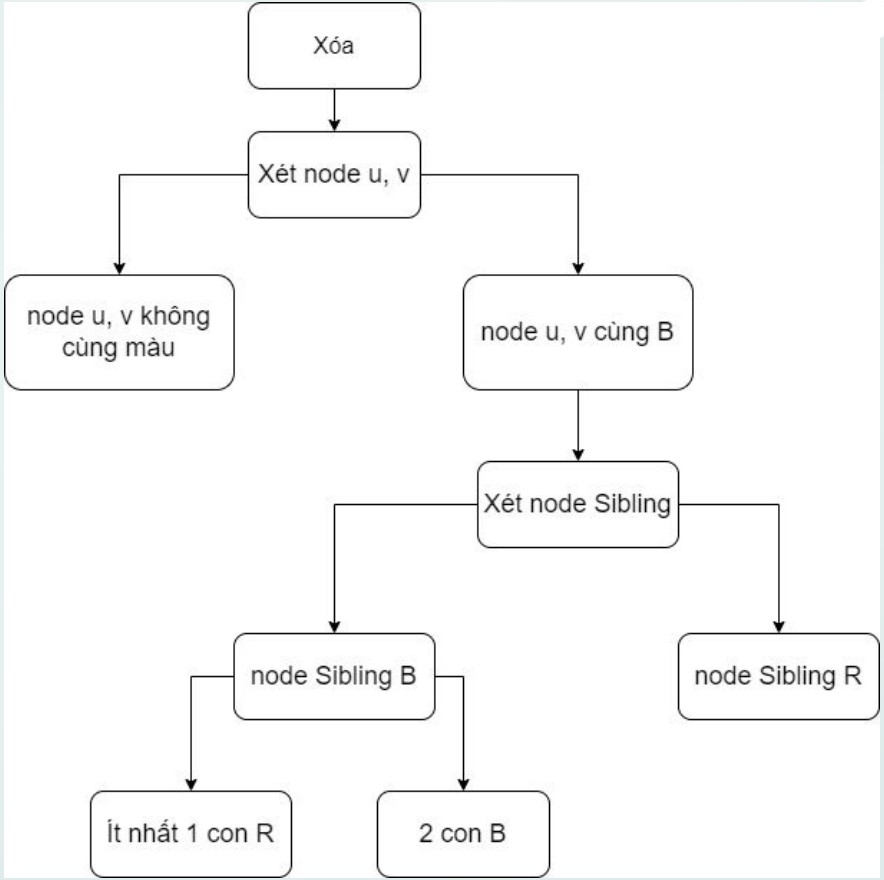
Node v là node bị xóa,
Node u là node con thay thế v





Thao tác trên cây





Thao tác trên cây

Deletion Steps

Following are detailed steps for deletion.

1) Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node to be deleted and u be the child that replaces v (Note that u is NULL when v is a leaf and color of NULL is considered as Black).

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that If v is leaf, then u is NULL and color of NULL is considered black. So the deletion of a black leaf also causes a double black.

Thao tác trên cây

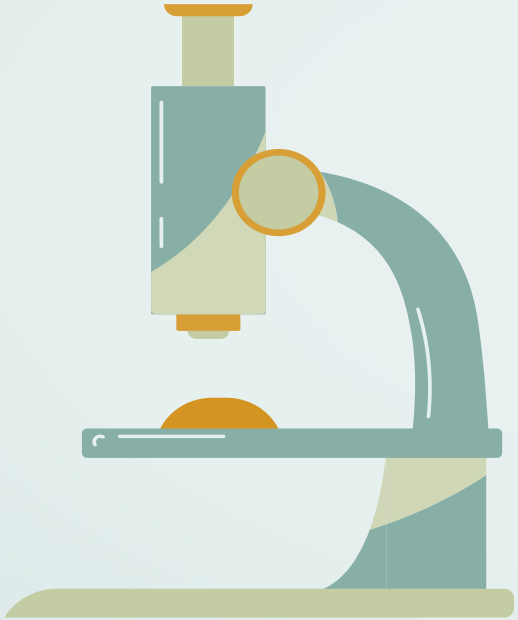
3.2) Do following while the current node u is double black, and it is not the root. Let sibling of node be **s**.
….**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.
……..**(i)** Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.
……..**(ii)** Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.
……..**(iii)** Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)
……..**(iv)** Right Left Case (s is right child of its parent and r is left child of s)

…..**(b): If sibling is black and its both children are black**, perform recoloring, and recur for the parent if parent is black. In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)
…..**(c): If sibling is red**, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.
……..**(i)** Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.
……..**(ii)** Right Case (s is right child of its parent). We left rotate the parent p.
3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

Thao tác trên cây

Thêm lần lượt: 3, 21, 32, 15, 12 vào một cây đỏ đen

Xóa lần lượt: 15, 12 ra khỏi cây đỏ đen



AVL VS RED Black Tree Performance

1. Dữ liệu: 1 triệu phần tử ngẫu nhiên=> Cây có 631.895 Node.
2. Cấu hình: CPU: E8500 3.16 GHZ và 4GB Ram.
3. OCaml compiler

7 Performance measurements

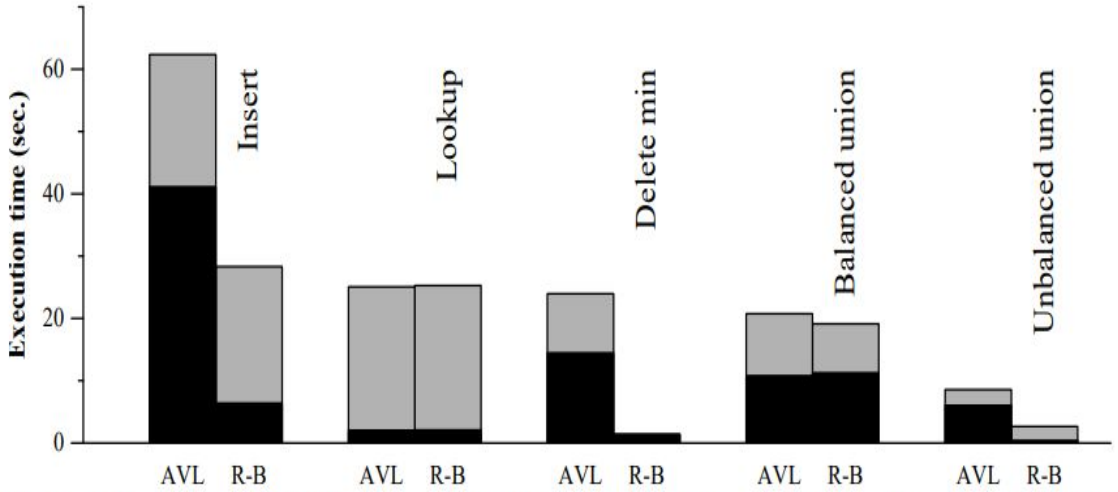


Fig. 1. Performance of AVL vs. Red-black trees. Black bars are actual running time with fast comparisons. Black+grey bars are actual running time with slow comparisons.

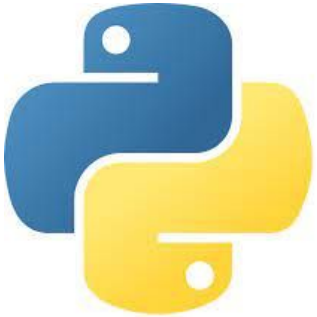
Cài đặt

Cài đặt

04

```
class Node {
public:
  int val;
  COLOR color;
  Node *left, *right, *parent;

  Node(int val) : val(val) {
    parent = left = right = NULL;

    // Node is created during insertion
    // Node is red at insertion
    color = RED;
  }

  // returns pointer to uncle
  Node *uncle() {
    // If no parent or grandparent, then no uncle
    if (parent == NULL or parent->parent == NULL)
      return NULL;

    if (parent->isOnLeft())
      // uncle on right
      return parent->parent->right;
    else
      // uncle on left
```

Cài đặt



Thư viện redblacktree

```
# Inserting
tree.insert(5) # Inserts a key with no value
tree[5] = None # Same as above
tree.insert(5, 'five')
tree[5] = 'five'

# Removing
tree.remove(5)
del tree[5]

# Query
x = tree[5]
if 5 in tree:
        print('5 is in the tree')

# Slicing also supported
tree[5:10] # Returns (key, value) pairs of all keys >= 5 and <= 10
tree[:5]   # All (key, value) pairs for keys <= 5
# Beware slicing with a step for example tree[1:100:-1] wont work

# Simple iteration:
for k, v in tree:
    print(k, v)
```

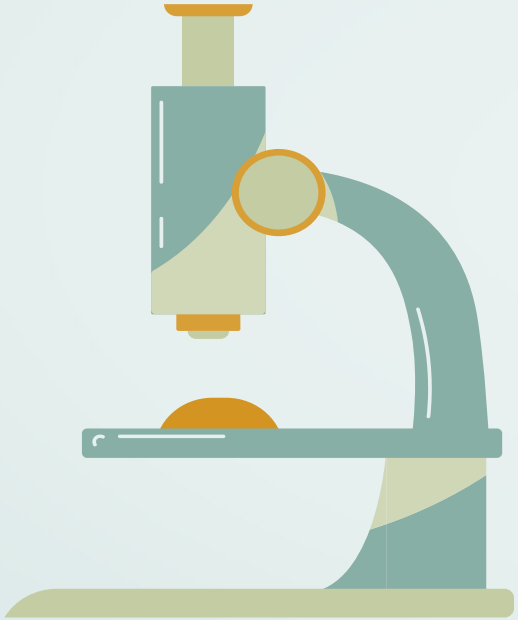
05

Bài tập

Bài tập trên lớp

Bài Quiz: https://forms.gle/r8regGkDMpd7imtn8

Bài tập về nhà



Bài 1: Chứng minh: từ một node A bất kì, tổng số node trong các đường đi từ node A tới hết trong đường đi dài nhất không vượt quá hai lần trong đường đi ngắn nhất.

Bài 2: Vẽ cây đỏ đen sau mỗi lần thêm, xóa:

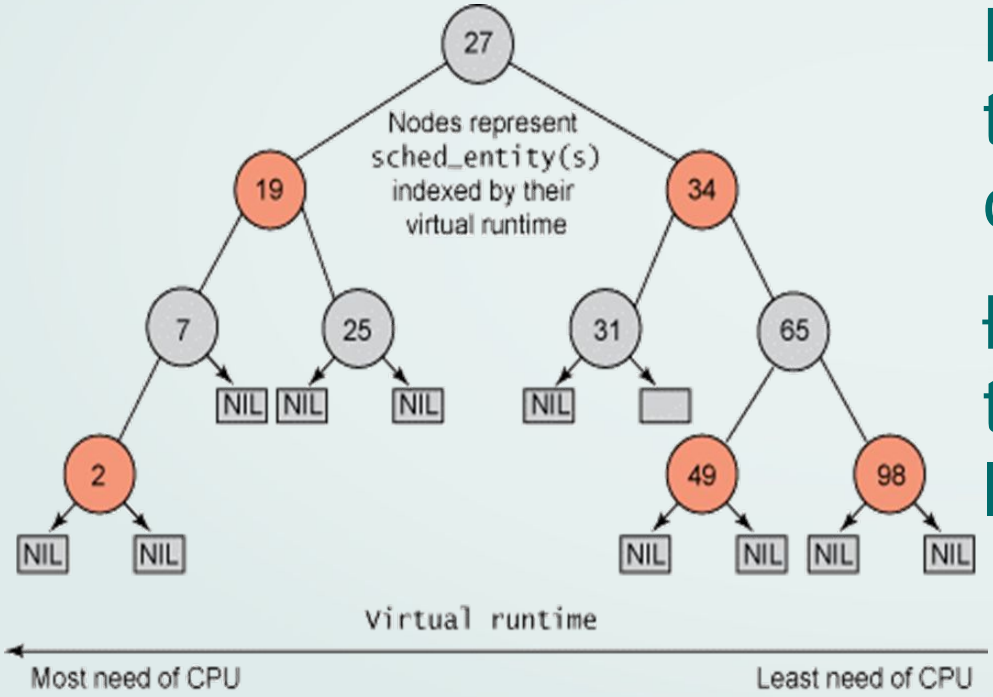
Thêm: 12, 13, 45, 11, 27, 60, 50, 55

Xóa: 60, 55

Demo

- **Linux Kernel**: Completely Fair Scheduler

Completely Fair Scheduler



Khóa cho mỗi node là thời gian đã được thực thi của 1 task trên CPU.

Để lựa chọn task tiếp theo để chạy thì chỉ việc lấy task ở node trái cùng.

Red Black Tree và Min HeapHeap

1. Cây đỏ đen có chiều cao là log(n) nên việc truy cập nhanh O(1), Việc thêm và xóa không cần tốn nhiều việc(độ phức tạp) và cây nó cũng sẽ có ít thao tác sắp xếp lại hơn là một mảng.
2. Một lần duyệt từ trên xuống có thể sử dụng cho việc chèn và xóa.
3. Chương mình mà có thể xử lý song song thì sử dụng cấu trúc cây sẽ dễ dàng tách ra => đưa vào thread hoặc hàng đợi.
4. Heap dựa trên mảng và nó cần một vùng nhớ liên tục trong kernal Space=> Nghẽn cổ chai=> Triển khai heap bằng con trỏ thì người ta dùng lại cấu trúc cây đỏ đen do nó đã có sẵn rồi.

Các nguồn tham khảo

Tài liệu tham khảo

[1] manasab220, “Applications, Advantages and Disadvantages of Red-Black Tree,” Geeksforgeeks, 25 11 2022. [Trực tuyến]. [Đã truy cập 25 11 2022].

[2] “Red-Black Trees,” Codesdope, [Trực tuyến]. Available: https://www.codesdope.com/course/data-structures-red-black-trees/. [Đã truy cập 25 11 2022].

[3] “redblacktree 1.0.2,” PyPI, [Trực tuyến]. Available: https://pypi.org/project/redblacktree/. [Đã truy cập 25 11 2022].

[4] U. o. Fancisco, “Red Black Tree Visualization,” University of Fancisco, [Trực tuyến]. Available: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html. [Đã truy cập 25 11 2022].

[5] A. W. APPEL, “Efficient Verified Red-Black Trees,” Efficient Verified Red-Black Trees, September 2011.

THANKS

Do you have any questions?