**1. Thêm 15 node vào cây đỏ đen (Red Black Tree)**

Red-Black Tree là một loại cây nhị phân tìm kiếm đặc biệt, đảm bảo các tính chất sau để cân bằng độ cao của cây:

1. **Mỗi node hoặc là màu đỏ, hoặc là màu đen.**
2. **Root node phải là màu đen.**
3. **Mỗi lá (leaf) là một node đen đặc biệt (NIL).**
4. **Nếu một node là đỏ, thì cả hai con của nó phải là màu đen.**
5. **Đường dẫn từ một node bất kỳ đến các lá của nó (các NIL nodes) đều có cùng số node đen.**

Khi thêm một node mới vào cây, quá trình chèn sẽ cần phải thỏa mãn những tính chất trên. Nếu một tính chất nào đó bị vi phạm, sẽ cần phải thực hiện các thao tác như **xoay cây (rotate)** và **đổi màu (recolor)** để khôi phục tính chất của cây đỏ đen.

**Các bước chèn một node mới vào cây đỏ đen:**

1. **Chèn như một cây nhị phân tìm kiếm thông thường (BST).**
2. **Gán màu đỏ cho node mới.**
3. **Kiểm tra và sửa chữa để đảm bảo các tính chất của cây đỏ đen, bao gồm:**
   * Đổi màu.
   * Xoay trái hoặc xoay phải nếu cần thiết.

**Chèn 15 node vào cây:**

Ví dụ, chèn lần lượt các giá trị sau vào một cây đỏ đen ban đầu rỗng:

20, 15, 25, 10, 5, 1, 30, 35, 40, 12, 18, 22, 28, 27, 26

**2. Tóm tắt nội dung rút ra sau quá trình thực hiện**

Sau khi hoàn tất quá trình chèn 15 node vào Red-Black Tree, chúng ta có thể rút ra các kết luận như sau:

* **Cân bằng tự động:** Red-Black Tree giúp cân bằng tự động nhờ các quy tắc về màu sắc và xoay cây, điều này giúp cây có độ cao hợp lý (O(log n)) và đảm bảo hiệu suất tìm kiếm, thêm, xóa ở mức tốt.
* **Đổi màu và xoay cây:** Các thao tác này thường xảy ra để sửa lỗi sau khi vi phạm các quy tắc của cây đỏ đen. Mặc dù chúng thêm vào phức tạp hơn so với cây nhị phân tìm kiếm bình thường, nhưng đảm bảo độ cân bằng.
* **Khả năng thêm nhanh:** Thao tác thêm vào Red-Black Tree vẫn giữ được độ phức tạp O(log n), nhờ vào việc giới hạn chiều cao cây thông qua quy tắc màu sắc và số lượng node đen.

**3.. Code minh họa**

|  |
| --- |
| class Node:  def \_\_init\_\_(self, data, color="red", parent=None):  self.data = data  self.color = color  self.parent = parent  self.left = None  self.right = None  class RedBlackTree:  def \_\_init\_\_(self):  self.NIL = Node(data=None, color="black") # NIL leaf node  self.root = self.NIL  def insert(self, data):  new\_node = Node(data)  new\_node.left = self.NIL  new\_node.right = self.NIL  if self.root == self.NIL:  self.root = new\_node  self.root.color = "black"  else:  self.\_insert\_node(self.root, new\_node)  self.\_fix\_insert(new\_node)    def \_insert\_node(self, current, new\_node):  if new\_node.data < current.data:  if current.left == self.NIL:  current.left = new\_node  new\_node.parent = current  else:  self.\_insert\_node(current.left, new\_node)  else:  if current.right == self.NIL:  current.right = new\_node  new\_node.parent = current  else:  self.\_insert\_node(current.right, new\_node)  def \_fix\_insert(self, node):  while node != self.root and node.parent.color == "red":  if node.parent == node.parent.parent.left:  uncle = node.parent.parent.right  if uncle.color == "red": # Case 1  node.parent.color = "black"  uncle.color = "black"  node.parent.parent.color = "red"  node = node.parent.parent  else:  if node == node.parent.right: # Case 2  node = node.parent  self.\_rotate\_left(node)  node.parent.color = "black" # Case 3  node.parent.parent.color = "red"  self.\_rotate\_right(node.parent.parent)  else:  uncle = node.parent.parent.left  if uncle.color == "red":  node.parent.color = "black"  uncle.color = "black"  node.parent.parent.color = "red"  node = node.parent.parent  else:  if node == node.parent.left:  node = node.parent  self.\_rotate\_right(node)  node.parent.color = "black"  node.parent.parent.color = "red"  self.\_rotate\_left(node.parent.parent)  self.root.color = "black"  def \_rotate\_left(self, node):  temp = node.right  node.right = temp.left  if temp.left != self.NIL:  temp.left.parent = node  temp.parent = node.parent  if node.parent is None:  self.root = temp  elif node == node.parent.left:  node.parent.left = temp  else:  node.parent.right = temp  temp.left = node  node.parent = temp  def \_rotate\_right(self, node):  temp = node.left  node.left = temp.right  if temp.right != self.NIL:  temp.right.parent = node  temp.parent = node.parent  if node.parent is None:  self.root = temp  elif node == node.parent.right:  node.parent.right = temp  else:  node.parent.left = temp  temp.right = node  node.parent = temp  def search(self, data):  return self.\_search\_tree(self.root, data)  def \_search\_tree(self, node, key):  if node == self.NIL or key == node.data:  return node  if key < node.data:  return self.\_search\_tree(node.left, key)  return self.\_search\_tree(node.right, key)  def delete(self, data):  pass # Xử lý hàm xóa (delete) sẽ phức tạp hơn và yêu cầu bổ sung các trường hợp đặc biệt.    def print\_tree(self, node, indent="", last=True):  if node != self.NIL:  print(indent, "`- " if last else "|- ", node.data, "(" + node.color + ")", sep="")  indent += " " if last else "| "  self.print\_tree(node.left, indent, False)  self.print\_tree(node.right, indent, True)  # Chạy thử chương trình  tree = RedBlackTree()  values = [20, 15, 25, 10, 5, 1, 30, 35, 40, 12, 18, 22, 28, 27, 26]  for value in values:  tree.insert(value)  tree.print\_tree(tree.root) |