二叉树精讲

MLE 算法指北

2025 年 2 月 4 日

1　二叉树（Binary Tree）

1.1　定义

二叉树（Binary Tree）是一种树形数据结构，其中每个节点最多有两个子节点，分别称为**左子节点**（Left Child）和**右子节点**（Right Child）。

1.2　基本术语

- **根节点（Root）**：树的顶端节点，没有父节点。
- **叶子节点（Leaf）**：没有子节点的节点。
- **深度（Depth）**：节点到根节点的路径长度。
- **高度（Height）**：节点到其最远叶子节点的路径长度。
- **满二叉树（Full Binary Tree）**：所有节点要么没有子节点，要么有两个子节点。
- **完全二叉树（Complete Binary Tree）**：除了最后一层，所有层都填满，且最后一层的叶子节点从左到右排列。

2　二叉搜索树（Binary Search Tree, BST）

2.1　定义

二叉搜索树（BST）是一种特殊的二叉树，满足以下性质：

- **左子树的所有节点值 < 根节点的值**。
- **右子树的所有节点值 > 根节点的值**。
- **左右子树本身也是二叉搜索树**。

2.2　基本操作

2.2.1　搜索（Search）

从根节点开始：

- 若 $k ==$ 根节点值，返回该节点。
- 若 $k <$ 根节点值，递归搜索左子树。
- 若 $k >$ 根节点值，递归搜索右子树。

时间复杂度：

- **最坏情况（退化成链表）**：$O(n)$
- **平均情况（平衡 BST）**：$O(\log n)$

2.2.2　插入（Insert）

从根节点开始，找到合适的位置：

- 若值小于当前节点，递归插入左子树。
- 若值大于当前节点，递归插入右子树。

2.2.3 删除（Delete）

删除节点的情况：

- 若节点无子节点，直接删除。
- 若节点有一个子节点，用其子节点代替删除节点。
- 若节点有两个子节点，找到右子树的最小值（后继节点），用其替换删除的节点，再删除后继节点。

Python 实现：

```python
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
            return
        self._insert(self.root, value)

    def _insert(self, node, value):
        if value < node.value:
            if node.left:
                self._insert(node.left, value)
            else:
                node.left = TreeNode(value)
        else:
            if node.right:
                self._insert(node.right, value)
            else:
                node.right = TreeNode(value)
```

3 平衡二叉树（Balanced Binary Tree）

3.1 定义

平衡二叉树是指任何节点的左右子树高度之差不超过 1，即：

$$|h_{\text{left}} - h_{\text{right}}| \leq 1$$

常见的平衡二叉树：

- **AVL 树**（严格平衡）
- **红黑树**（近似平衡）

3.2 AVL 树

AVL 树是一种严格平衡的二叉搜索树，满足：

- 每个节点的左右子树高度差不超过 1。
- 通过**旋转（Rotation）**来维持平衡。

3.2.1 旋转操作

- **右旋（Right Rotation）**：左子树过高时使用。
- **左旋（Left Rotation）**：右子树过高时使用。
- **左右旋（Left-Right Rotation）**：左子树的右子树过高时使用。

- **右左旋（Right-Left Rotation）**：右子树的左子树过高时使用。

时间复杂度：

- **搜索**：$O(\log n)$
- **插入/删除**：$O(\log n)$（旋转操作为常数时间）

3.2.2 Python 实现

```
class AVLNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def get_height(self, node):
        return node.height if node else 0

    def get_balance(self, node):
        return self.get_height(node.left) - self.get_height(node.right) if node else 0

    def right_rotate(self, z):
        y = z.left
        z.left = y.right
        y.right = z
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def left_rotate(self, z):
        y = z.right
        z.right = y.left
        y.left = z
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y
```

3.3 红黑树（Red-Black Tree, RBT）

红黑树（Red-Black Tree, RBT）是一种**自平衡二叉搜索树**，每个节点额外存储一个颜色信息（红色或黑色）。红黑树通过**颜色约束和旋转操作**来维持平衡，保证基本操作的时间复杂度为 $O(\log n)$。

3.3.1 红黑树性质

红黑树必须满足以下五个性质：

1. **每个节点是红色或黑色**。
2. **根节点必须是黑色**。
3. **红色节点的子节点必须是黑色**（即**不能有连续的两个红色节点**）。
4. **从任意节点到其所有叶子节点的路径上，必须包含相同数量的黑色节点**。
5. **新插入的节点默认是红色**。

这些性质确保了红黑树的 ** 最长路径不会超过最短路径的两倍 **,从而保证树的高度维持在 $O(\log n)$ 级别。

3.3.2 基本操作

搜索（Search） 红黑树的搜索操作与普通二叉搜索树（BST）相同：

- 若 $k ==$ 当前节点值，返回该节点。
- 若 $k <$ 当前节点值，递归搜索左子树。
- 若 $k >$ 当前节点值，递归搜索右子树。

时间复杂度：

- **最坏情况**：$O(\log n)$
- **平均情况**：$O(\log n)$

3.3.3 插入（Insert）

红黑树的插入操作分为以下步骤：

1. ** 按照二叉搜索树（BST）规则插入节点 **，新节点默认为红色。
2. ** 检查是否违反红黑树性质 **：
   - 若插入的节点是**根节点**，则将其变为黑色。
   - 若父节点是黑色，则不需要修复。
   - 若父节点是红色，则会破坏性质（不能有连续两个红色节点），需要**调整**。
3. ** 通过旋转（Rotation）和重新着色（Recoloring）修复红黑树 **：
   - **左旋（Left Rotation）**：当父节点是祖父节点的左子树，且叔节点是黑色时使用。
   - **右旋（Right Rotation）**：当父节点是祖父节点的右子树，且叔节点是黑色时使用。
   - **变色（Recoloring）**：当叔节点是红色时，将祖父节点变为红色，父节点和叔节点变为黑色。

3.3.4 删除（Delete）

删除红黑树的节点较为复杂，主要分为：

1. 按照二叉搜索树（BST）规则删除节点。
2. 若删除的节点是红色，不影响红黑树的平衡。
3. 若删除的节点是黑色，可能会破坏红黑树的性质，需要**调整**：
   - ** 双黑修复 **：若被删除的黑色节点有一个黑色子节点，则进行变色或旋转修复。
   - ** 旋转修复 **：若兄弟节点是红色，则进行旋转操作。

3.4 红黑树的旋转操作

- **左旋（Left Rotation）**：用于修复右倾斜的情况。
- **右旋（Right Rotation）**：用于修复左倾斜的情况。
- **左右旋（Left-Right Rotation）**：先左旋，再右旋。
- **右左旋（Right-Left Rotation）**：先右旋，再左旋。

Python 实现：

```python
class Node:
    def __init__(self, value, color="RED"):
        self.value = value
        self.color = color  # RED or BLACK
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.TNULL = Node(0, "BLACK")
```

```python
        self.root = self.TNULL

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
            y.left.parent = x
        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.TNULL:
            y.right.parent = x
        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y
```

4 总结

数据结构	搜索	插入	删除	是否严格平衡
二叉搜索树（BST）	$O(\log n)$ / $O(n)$	$O(\log n)$ / $O(n)$	$O(\log n)$ / $O(n)$	
AVL 树	$O(\log n)$	$O(\log n)$	$O(\log n)$	
红黑树	$O(\log n)$	$O(\log n)$	$O(\log n)$	近似平衡