用 Python 建造森林系列：AVL 树

顺煌

2021 年 6 月 7 日 警察

评价我： 5.00/5 (6 票)

使用 Python 构建 AVL 树

在这篇文章中，我们看一下 AVL 树如何在插入和删除操作上引入一些复杂度以保持其平衡，例如 AVL 树的自平衡能力如何为基本操作提供 O(lg n) 时间复杂度，这比性能更好常规二叉搜索树。

介绍

在红黑树讨论之后，本文将实现自平衡二叉搜索树的另一个变体：AVL 树。

项目设置

遵循与构建森林系列中的其他文章相同的风格和假设，实现假设 Python 3.9 或更新版本。本文增加了两个模块，以我们的项目：*avl_tree.py*的AVL树的实现和*test_avl_tree.py*它的单元测试。添加这两个文件后，我们的项目布局就变成了这样：

复制代码

```
forest-python
├── forest
│   ├── __init__.py
│   ├── binary_trees
│   │   ├── __init__.py
│   │   ├── avl_tree.py
│   │   ├── binary_search_tree.py
│   │   ├── double_threaded_binary_tree.py
│   │   ├── red_black_tree.py
│   │   ├── single_threaded_binary_trees.py
│   │   └── traversal.py
│   └── tree_exceptions.py
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── test_avl_tree.py
    ├── test_binary_search_tree.py
    ├── test_double_threaded_binary_tree.py
    ├── test_red_black_tree.py
    ├── test_single_threaded_binary_trees.py
    └── test_traversal.py
```

（完整的代码可在Forest-python 中找到。）

什么是 AVL 树?

AVL 树（以发明者 Adelson-Velsky 和 Landis 的名字命名）是一种自平衡二叉搜索树。除了二叉搜索树属性之外，AVL 树还维护 AVL 树属性以保持平衡：

- 对于 AVL 树中的每个节点，其左子树和右子树的高度最多相差 1。

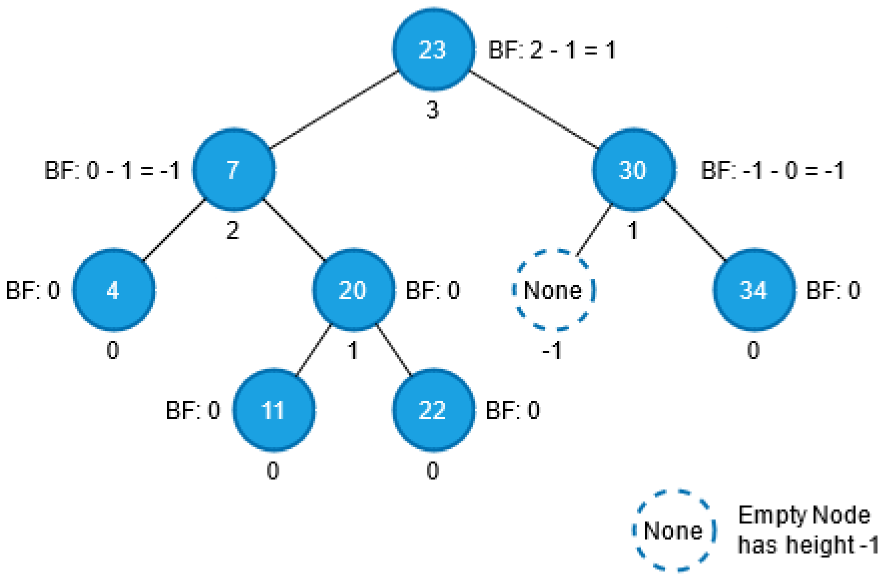
该属性也称为**平衡因子**，可以重写为以下公式：

$$BalanceFactorofanode = Heightofitsleftsubtree - Heightofitsrightsubtree, where BalanceFactorofanode \in \{-1, 0, 1\}$$

如果一个节点的平衡因子 > 0，我们称它为**left-heavy**。如果一个节点的平衡因子 < 0，我们称它为**right-heavy**。如果节点的平衡因子 = 0，则称为**平衡**。

（请注意，有些人将平衡因子定义为其右子树的高度 - 其左子树的高度。在这种情况下，左重成为节点的平衡因子 < 0，而右重发生在节点的平衡因子 > 0.但是，无论使用哪种定义，AVL树的概念都是一样的。）

典型的 AVL 树可以在下图中可视化：



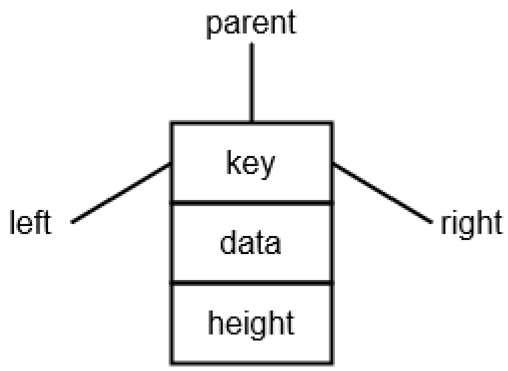
在上图中，BF 表示节点的平衡因子。节点下的数字是节点的高度。如果节点为空，即`None`，则其高度为`-1`。这种方式可以更容易地计算节点的平衡因子。

构建 AVL 树

本节将介绍 AVL 树的实现以及实现选择背后的一些想法。

节点

我们不是每次需要时都计算节点的高度，而是将高度存储在每个节点中。因此，节点的结构比二叉搜索树节点多一个字段。



存储高度可以节省计算时间，因此我们不需要每次检查平衡因子时都计算高度。然而，它带来了成本——我们需要在修改 AVL 树时保持高度最新，例如插入节点或删除节点。有关高度更新的更多详细信息将在插入和删除部分提供。

与其他二叉树节点一样，我们利用数据类来定义 AVL 树节点。

Python 复制代码

```python
from dataclasses import dataclass

@dataclass
class Node:
    """AVL Tree node definition."""

    key: Any
    data: Any
    left: Optional["Node"] = None
    right: Optional["Node"] = None
    parent: Optional["Node"] = None
    height: int = 0
```

课程概览

与Build the Forest项目中的其他类型的二叉树一样，AVL 树类具有类似的功能。

Python 缩小▲ 复制代码

```python
class AVLTree:

    def __init__(self) -> None:
        self.root: Optional[Node] = None

    def __repr__(self) -> str:
        """Provie the tree representation to visualize its layout."""
        if self.root:
            return (
                f"{type(self)}, root={self.root}, "
                f"tree_height={str(self.get_height(self.root))}"
            )
        return "empty tree"

    def search(self, key: Any) -> Optional[Node]:
        …

    def insert(self, key: Any, data: Any) -> None:
        …

    def delete(self, key: Any) -> None:
        …

    @staticmethod
    def get_leftmost(node: Node) -> Node:
        …
```

```
    @staticmethod
    def get_rightmost(node: Node) -> Node:
        …

    @staticmethod
    def get_successor(node: Node) -> Optional[Node]:
        …

    @staticmethod
    def get_predecessor(node: Node) -> Optional[Node]:
        …

    @staticmethod
    def get_height(node: Optional[Node]) -> int:
        …

    def _get_balance_factor(self, node: Optional[Node]) -> int:
        …

    def _left_rotate(self, node_x: Node) -> None:
        …

    def _right_rotate(self, node_x: Node) -> None:
        …

    def _insert_fixup(self, new_node: Node) -> None:
        …

    def _transplant(self, deleting_node: Node, replacing_node: Optional[Node]) -> None:
        …

    def _delete_no_child(self, deleting_node: Node) -> None:
        …

    def _delete_one_child(self, deleting_node: Node) -> None:
        …

    def _delete_fixup(self, fixing_node: Node) -> None:
        …
```

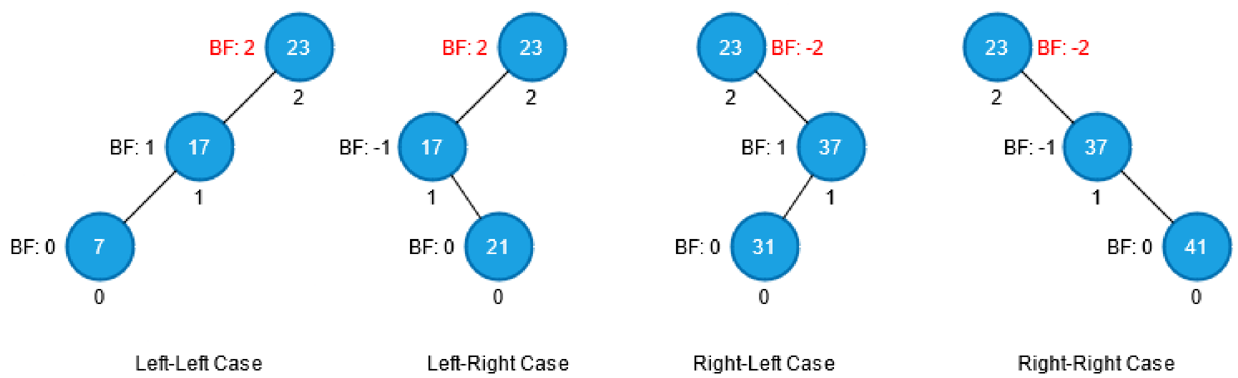
我们可以像常规二叉搜索树一样实现大部分 AVL 树功能，例如搜索和大多数辅助功能。我们也可以使用二叉树遍历中的遍历函数来遍历一棵AVL树。

插入和删除是两个可能导致 AVL 树不平衡的操作。因此，`AVLTree`类有方法，可帮助保持平衡的树，其中包括`_left_rotate()`，`_right_rotate()`，`_insert_fixup()`，和`_delete_fixup()`。由于这些辅助方法主要保持 AVL 树的平衡，因此我们将它们定义为私有函数并且对客户端代码透明。

轮换

插入或删除后恢复违反的 AVL-tree-property 的方法是旋转（类似于红黑树：旋转）。

下图展示了 AVL-tree-property 可能被破坏的四种情况：left-left、left-right、right-left 和 right-right。

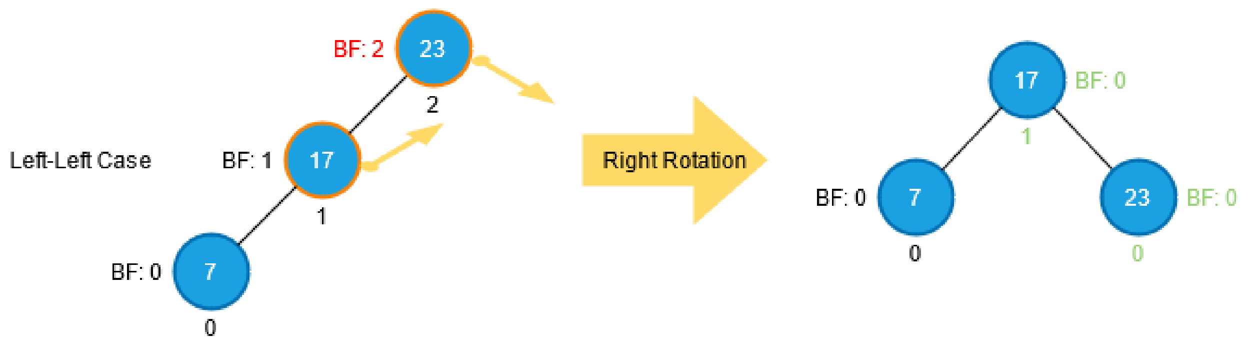


当我们处理一棵不平衡的 AVL 树时，我们总是从最底部的不平衡节点开始（即，该节点要么是*BF > 1*要么是*BF < -1*）。因此，例如，上图中的节点 23 是最底部的不平衡节点。然后检查其**身高较高的孩子**的平衡系数。如果最底部的不平衡节点是left-heavy（即BF > 0）并且其高度较高的子节点是left-heavy，则为left-left情况（上图中最左边的情况）。如果身高较高的孩子的平衡因子是右重（即 BF < 0），则是左右情况（上图中第二个最左边的情况）。图中的其他情况（右-左和右-右）与左-左情况和左右情况对称。

以下小节展示了旋转如何重新平衡不平衡情况。

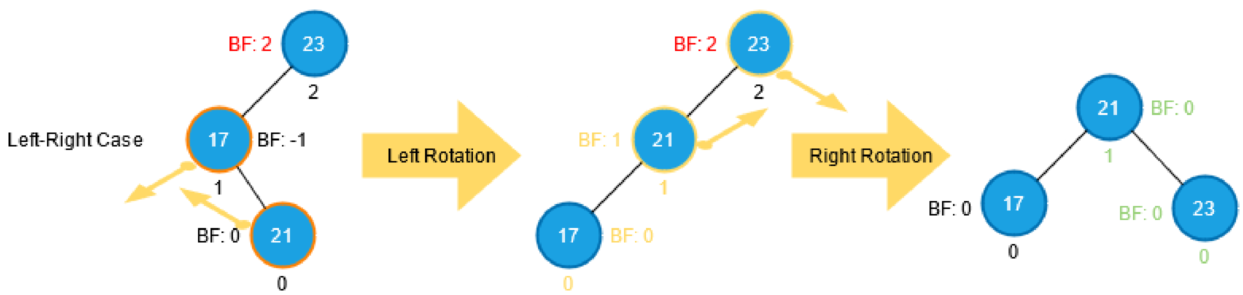
右旋（Left-Left Case）

对于左-左情况，我们在最底部的不平衡节点（在本例中为节点 23）执行右旋转。旋转后，节点 17 成为节点 23 的父节点。此外，旋转后节点 17 和节点 23 的高度和平衡因子都发生了变化。



左右旋转（左右情况）

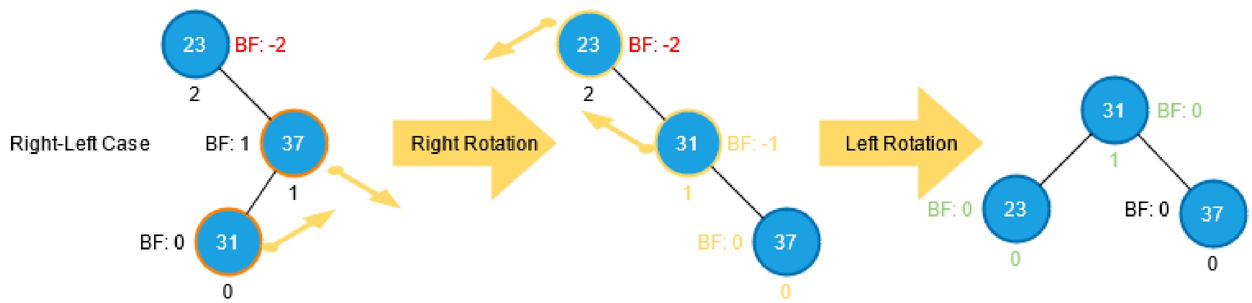
如果最底部的不平衡节点是左重节点，但它的子节点（高度更高的节点）是右重节点，我们首先对子节点（本例中的节点 17）执行左旋转，然后对最底部的不平衡节点执行右旋转（节点 23）。



请注意，只有涉及旋转的节点才会更改其平衡因子和高度。例如，在向左旋转后，节点 17 和节点 21 改变了它们的高度和平衡因子（以黄色突出显示）。右旋后，只有节点 21 和节点 23 改变了它们的高度和平衡因子（绿色高度）。

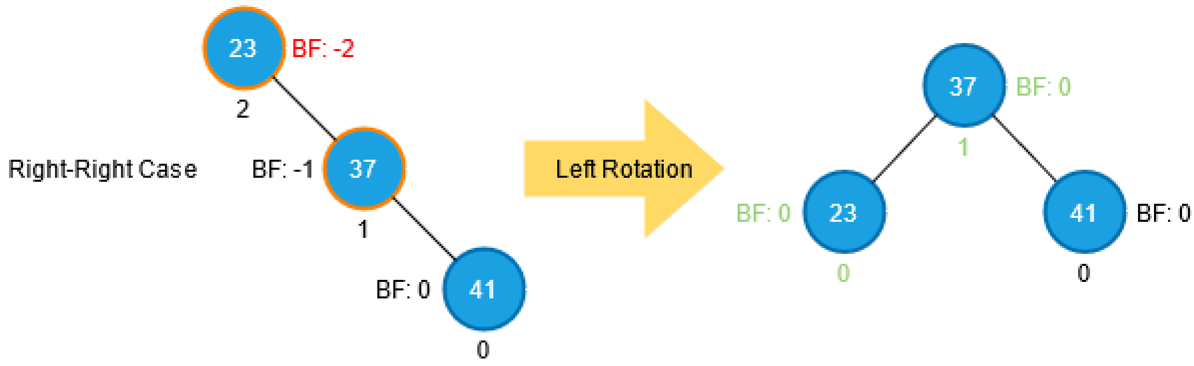
左右旋转（左右情况）

左右情况与左右情况对称。因此，我们先执行右旋转，然后执行左旋转。



左旋转（右-右情况）

左右情况与左右情况对称。所以我们可以进行左旋，使其平衡。



概括

下表总结了不平衡的情况及其解决方案。

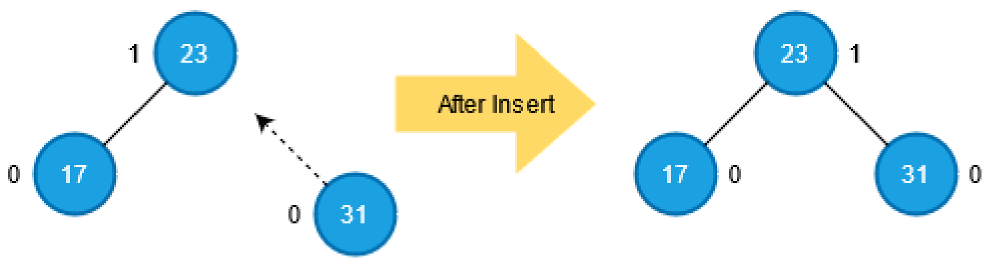
Case	Bottommost unbalanced Node	The child who has the higher height	Rotation
Left-Left	Left-heavy (BF > 0)	Left-heavy (BF >= 0)	Right rotation
Left-Right	Left-heavy (BF > 0)	Right-heavy (BF < 0)	Left-Right rotation
Right-Left	Right-heavy (BF < 0)	Left-heavy (BF > 0)	Right-Left rotation
Right-Right	Right-heavy (BF < 0)	Right-heavy (BF <= 0)	Left rotation

插入

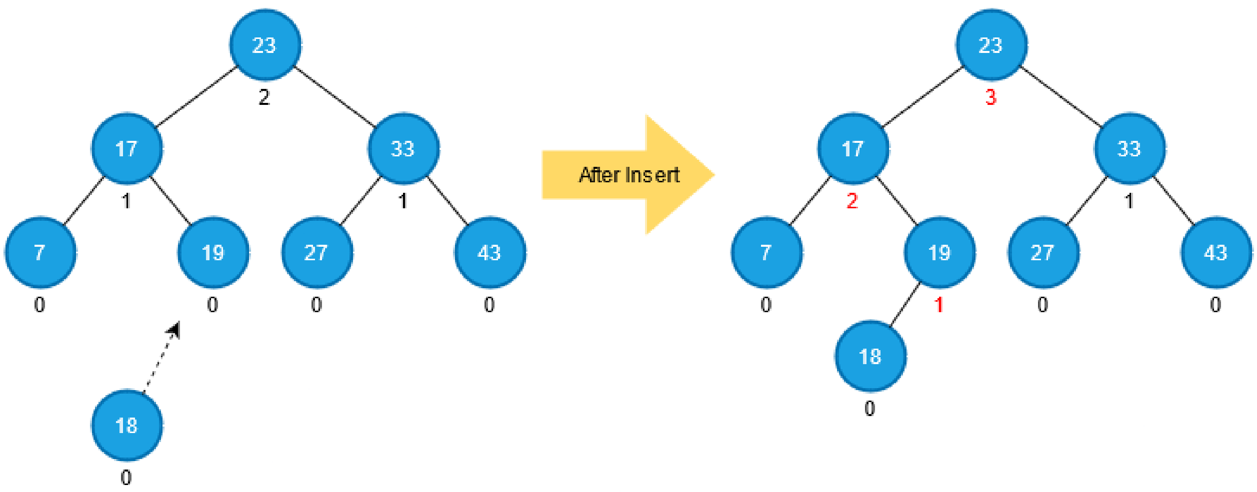
在 AVL 树中插入有一个实质性的影响：更新高度。因此，当我们将一个节点插入到 AVL 树中时，新节点可能会改变树的高度，从而违反了 AVL-tree-property。当这种情况发生时，我们执行特定的旋转来重新平衡树。

高度更新

在实现该`insert`功能之前，我们需要了解插入是如何改变高度的。首先要注意，新插入的节点插入后必须成为叶子节点。因此，新节点的高度必须为0，其平衡因子也为0。此外，如果新节点的父节点在新节点插入之前有子节点，则父节点和整个树的高度保持不变：高度不变，没有AVL 树属性违规。



其次，只有当新节点的父节点在插入之前没有子节点时，才会发生高度变化。在这种情况下，我们将新节点的祖先的高度一直更新到根（如下图），并且只有新节点的祖先有高度更新。当高度发生变化时，意味着可能会发生潜在的 AVL-tree-property 违规。



（上图并没有违反AVL-tree-property。但是，我们将在下面的部分中处理插入后AVL树变得不平衡的情况。）

插入算法是从常规二叉搜索树插入修改而来的。

1. 插入高度为 0 的新节点与二叉搜索树插入的方式相同：通过从根开始遍历树并比较新节点，找到合适的位置（即新节点的父节点）插入新节点节点的键与沿途每个节点的键。
2. 更新高度并通过从新节点到根的回溯来检查违反的 AVL-tree-property 是否发生。在返回根节点的过程中，如有必要，在途中更新每个节点的高度。如果我们发现一个不平衡的节点，执行一定的旋转来平衡它。旋转后，插入结束。如果没有发现不平衡节点，则在到达根并更新其高度后完成插入。

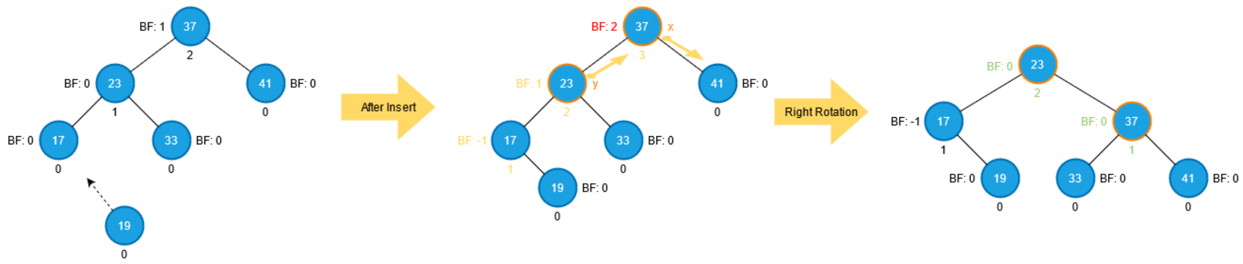
Python 缩小▲ 复制代码

```python
def insert(self, key: Any, data: Any) -> None:
    new_node = Node(key=key, data=data)
    parent: Optional[Node] = None
    current: Optional[Node] = self.root
    while current:
        parent = current
        if new_node.key < current.key:
            current = current.left
        elif new_node.key > current.key:
            current = current.right
        else:
            raise tree_exceptions.DuplicateKeyError(key=new_node.key)
    new_node.parent = parent
    # If the tree is empty, set the new node to be the root.
    if parent is None:
        self.root = new_node
    else:
        if new_node.key < parent.key:
            parent.left = new_node
        else:
            parent.right = new_node

        # After the insertion, fix the broken AVL-tree-property.
        # If the parent has two children after inserting the new node,
        # it means the parent had one child before the insertion.
        # In this case, neither AVL-tree property breaks nor
        # heights update requires.
        if not (parent.left and parent.right):
            self._insert_fixup(new_node)
```

修理

正如轮换部分提到的，有四种潜在的不平衡情况。并且我们执行特定的旋转来恢复 AVL-tree-property。在我们修复 AVL-tree-property 之后，AVL 树变得平衡。无需一直跟踪祖先的平衡因子。以下小节描述了每种情况的修正。

左右大小写



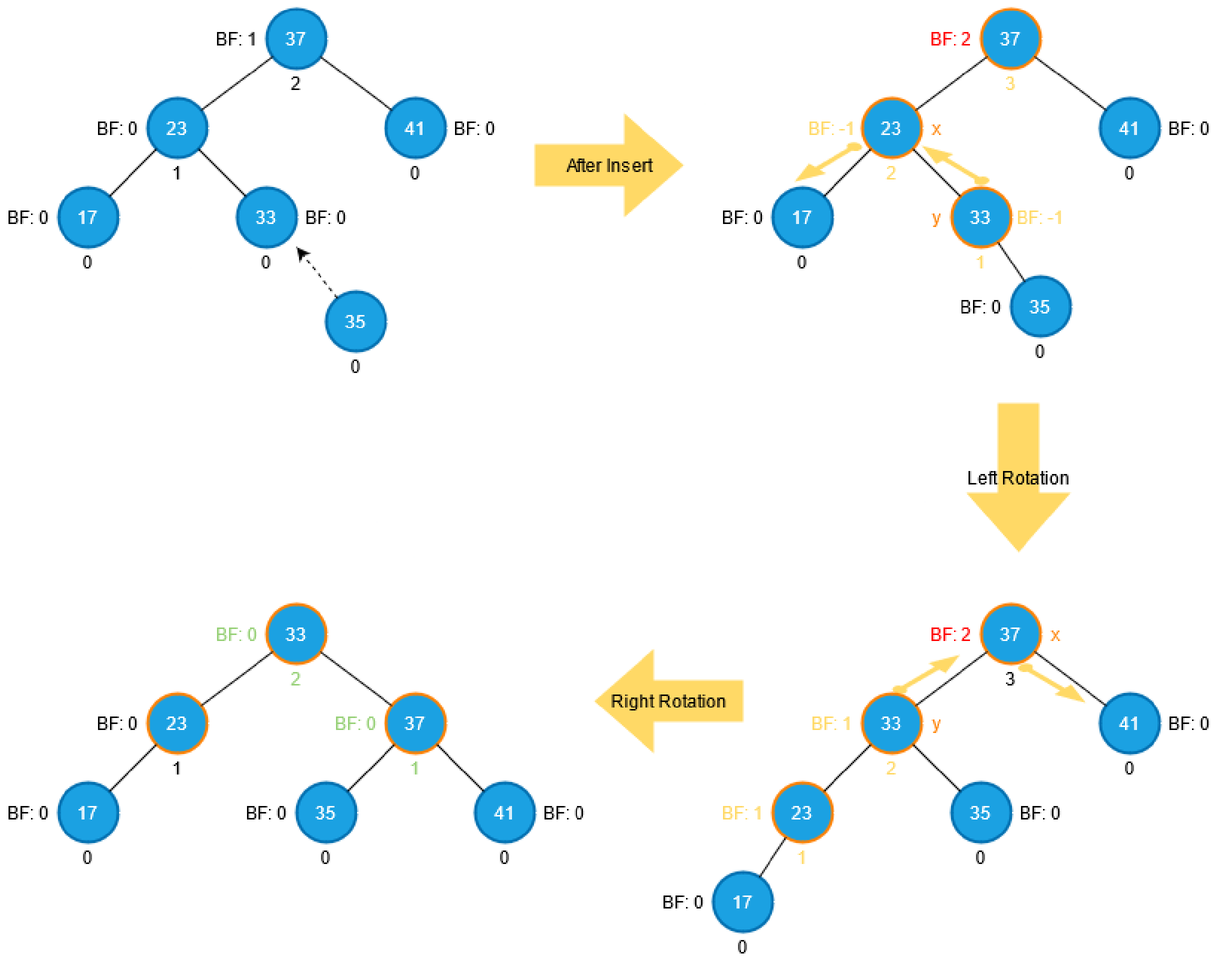
在上图中，我们添加节点19。插入后，我们从节点17 开始检查节点19 的祖先的平衡因子。然后，我们发现节点 37 的平衡因子不平衡且左重。我们还需要检查具有更高高度的子节点，以确定要执行的旋转。在插入的情况下，具有更高高度的子节点必须出现在包含新节点的路径中，即本例中的节点 23，因为插入后节点 23 的平衡因子为 1。我们确定这是左左情况。

所以我们对节点37进行右旋。右旋后，节点23成为新的根节点，节点37成为节点23的右孩子。请注意，只有参与旋转的节点具有高度和平衡因子的变化。因此，在这种情况下，只有节点 23 和节点 37 具有高度，并且平衡因子发生了变化。

为什么不需要在轮换后继续检查祖先的平衡系数?

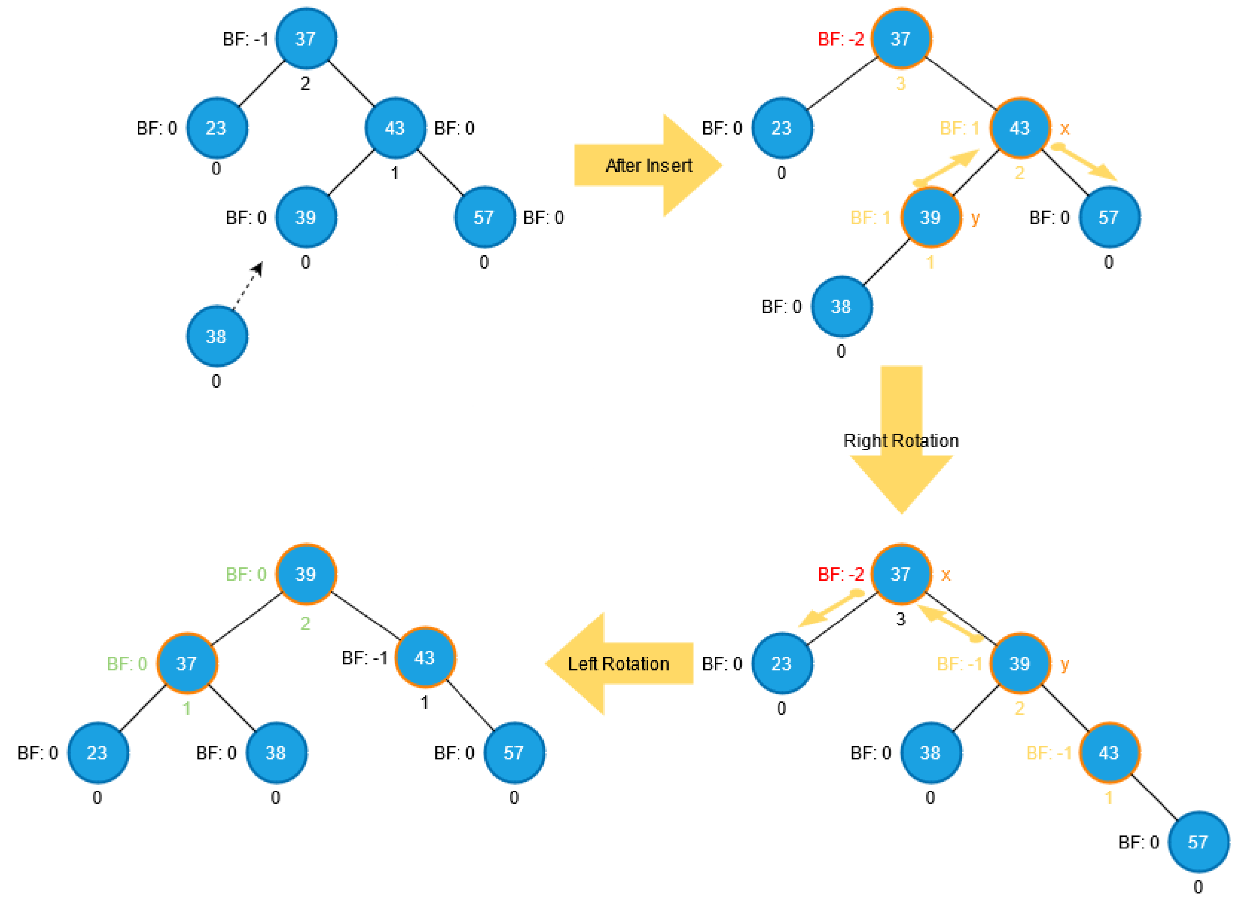
插入前节点37的高度为2，节点23的高度为1。插入后，节点37的高度变为3，节点23的高度变为2。然后我们进行了旋转。旋转后，节点23取到节点37原来的位置，节点23的高度变成了2，节点37的高度变成了1。因此，相同位置的高度保持不变，即根（节点37）之前的高度为2插入；旋转后，根（节点 23）的高度仍为 2。换句话说，如果节点 37在旋转之前有一个父节点（比如*x*），在旋转之后，*x*的左子节点变成节点 23，但 x 的高度不受影响。所以，我们不需要在旋转后继续检查 x 的祖先的高度。

左右大小写



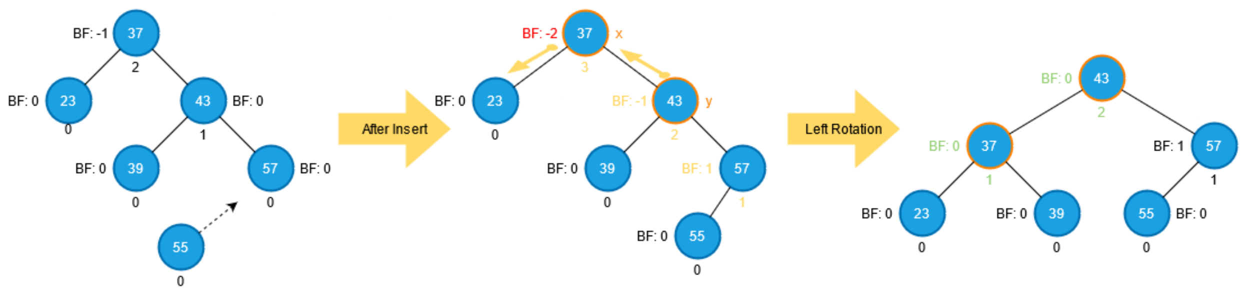
在我们通过检查最底部不平衡节点（节点 37）及其具有较高高度的子节点（节点 23）的平衡因子来确定这是左右情况后，我们对节点 23 执行左旋转，因此它变成了左-左情况。然后，我们在不平衡节点（节点 37）上执行正确的旋转。之后，我们恢复违反的 AVL-tree-property。

左右大小写



这种情况与左右情况是对称的，所以我们对不平衡节点（节点37）的右孩子（节点43）进行了右转，所以变成了左右情况。然后，我们在不平衡节点（节点 37）上执行左旋转。之后，我们修复违反的 AVL-tree-property。

右右案例



左右情况与左右情况对称，因此我们可以通过执行左旋转来恢复其 AVL-tree 属性。

经过修复分析后，我们可以实现如下`_insert_fixup`功能。请注意，当我们在树上行走时和旋转之前，我们总是更新节点的高度。

Python 缩小▲ 复制代码

```python
def _insert_fixup(self, new_node: Node) -> None:
    parent = new_node.parent

    while parent:
        parent.height = 1 + max(
            self.get_height(parent.left), self.get_height(parent.right)
        )
        grandparent = parent.parent
        # grandparent is unbalanced
        if grandparent:
```

```
            if self._get_balance_factor(grandparent) > 1:
                # Case Left-Left
                if self._get_balance_factor(parent) >= 0:
                    self._right_rotate(grandparent)
                # Case Left-Right
                elif self._get_balance_factor(parent) < 0:
                    self._left_rotate(parent)
                    self._right_rotate(grandparent)
                # Since the fixup does not affect the ancestor of the unbalanced
                # node, exit the loop to complete the fixup process.
                break
            elif self._get_balance_factor(grandparent) < -1:
                # Case Right-Right
                if self._get_balance_factor(parent) <= 0:
                    self._left_rotate(grandparent)
                # Case Right-Left
                elif self._get_balance_factor(parent) > 0:
                    self._right_rotate(parent)
                    self._left_rotate(grandparent)
                # Since the fixup does not affect the ancestor of the unbalanced
                # node, exit the loop to complete the fixup process.
                break
        parent = parent.parent
```

搜索

搜索功能与二叉搜索树相同： Search。

删除

AVL 树删除的基本思想类似于常规的二叉搜索树。要删除的节点有三种情况：没有子节点、只有一个子节点、两个子节点。我们还使用相同的`transplant`方法将以 为根的子树替换为`deleting_node`以 为根的子树`replacing_node`。

移植

该`transplant`方法与二叉搜索树相同：删除。

Python 复制代码

```python
def _transplant(self, deleting_node: Node, replacing_node: Optional[Node]) -> None:
    if deleting_node.parent is None:
        self.root = replacing_node
    elif deleting_node == deleting_node.parent.left:
        deleting_node.parent.left = replacing_node
    else:
        deleting_node.parent.right = replacing_node

    if replacing_node:
        replacing_node.parent = deleting_node.parent
```

与插入类似，删除 AVL 树可能会更新树的高度并可能违反 AVL-tree-property，因此我们需要检查 AVL 树是否变得不平衡并在删除后修复它。

整个删除过程就像一个经过一些修改的常规二叉搜索树。

1. 找到要删除的节点（`deleting_node`）。
2. 如果`deleting_node`没有孩子，使用的`transplant`方法来代替`deleting_node`使用`None`。然后，执行修复操作。
3. 如果`deleting_node`只有一个孩子，则使用只有一个孩子的`transplant`方法来替换`deleting_node`。然后执行修复操作。
4. 如果`deleting_node`有两个孩子，则找到`deleting_node`的继任者作为`replacing_node`。然后，将`deleting_node`的键和数据替换为`replacing_node`的键和数据，因此`deleting_node`被替换为，`replacing_node`但保持其原始平衡因子和高度（即，高度和平衡因子没有变化意味着没有违反 AVL-tree-property）。之后，删除`replacing_node`，这与第 2 步（没有孩子）或第 3 步（只有一个孩子）相同。

为了让实现更清晰，我们定义了_delete_no_child待删除节点没有子_delete_one_child节点和待删除节点只有一个子节点的情况下的方法。如果删除带有两个子节点的节点，我们可以相应地重用_delete_no_child和_delete_one_child。此外，由于要删除的节点有两个使用_delete_no_childand 的子节点_delete_one_child，因此只有这两个方法需要调用_delete_fixup函数来修复不平衡的节点。因此，我们可以实现delete, _delete_no_child, 和_delete_one_child函数如下：

Python 缩小▲ 复制代码

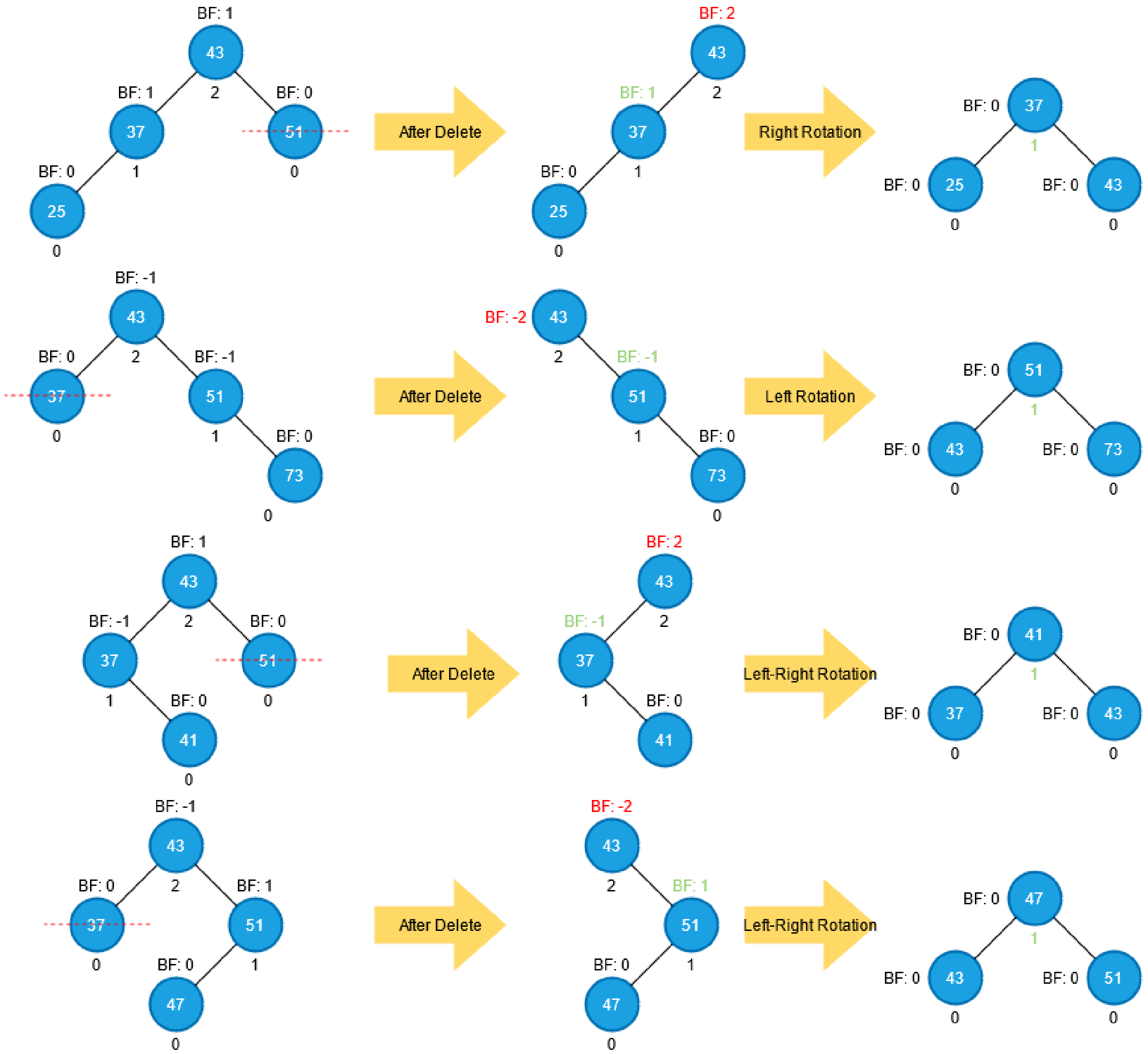
```python
def delete(self, key: Any) -> None:
    if self.root and (deleting_node := self.search(key=key)):
        # Case: no child
        if (deleting_node.left is None) and (deleting_node.right is None):
            self._delete_no_child(deleting_node=deleting_node)
        # Case: Two children
        elif deleting_node.left and deleting_node.right:
            replacing_node = self.get_leftmost(node=deleting_node.right)
            # Replace the deleting node with the replacing node,
            # but keep the replacing node in place.
            deleting_node.key = replacing_node.key
            deleting_node.data = replacing_node.data
            if replacing_node.right:  # The replacing node cannot have left child.
                self._delete_one_child(deleting_node=replacing_node)
            else:
                self._delete_no_child(deleting_node=replacing_node)
        # Case: one child
        else:
            self._delete_one_child(deleting_node=deleting_node)

def _delete_no_child(self, deleting_node: Node) -> None:
    parent = deleting_node.parent
    self._transplant(deleting_node=deleting_node, replacing_node=None)
    if parent:
        self._delete_fixup(fixing_node=parent)

def _delete_one_child(self, deleting_node: Node) -> None:
    parent = deleting_node.parent
    replacing_node = (
        deleting_node.right if deleting_node.right else deleting_node.left
    )
    self._transplant(deleting_node=deleting_node, replacing_node=replacing_node)
    if parent:
        self._delete_fixup(fixing_node=parent)
```

修理

我们知道删除操作可能会改变高度并违反 AVL 树属性。与插入一样，有四种潜在的不平衡情况。并且我们执行特定的旋转来恢复 AVL-tree-property。我们还从最底部的不平衡节点开始修复程序。最底部的不平衡节点必须是要删除的节点的祖先之一。然而，与插入修正不同，不平衡的平衡因子可能会在我们执行旋转的节点上方传播。因此，在我们恢复最底层的不平衡节点后，我们需要检查它的父节点。如果它的父级变得不平衡，请修复它。重复这个过程，直到我们到达根部，并且根部也平衡了。

没有孩子

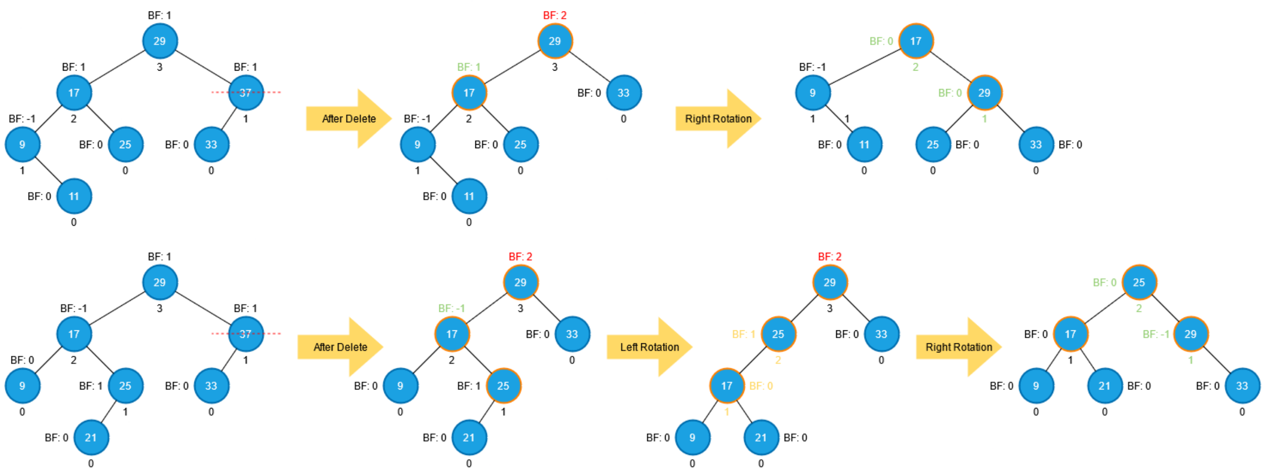
就像我们在旋转部分提到的那样，有四种情况可能会违反 AVL 树属性。下图显示了这四种情况，以及如果要删除的节点没有子节点，如何恢复它们的平衡因子。



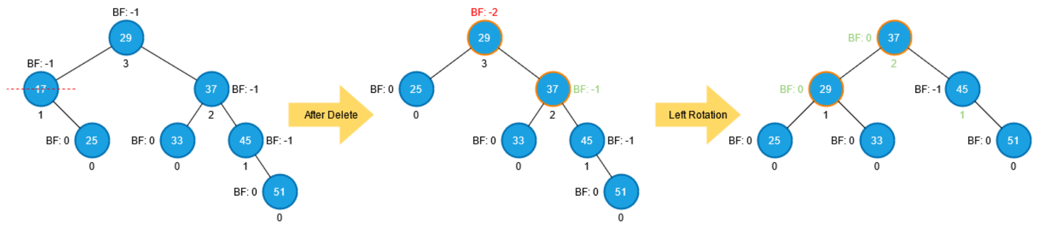
一个小孩

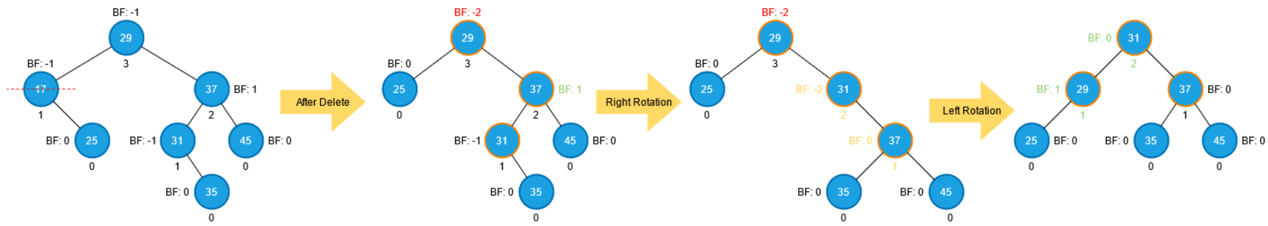
下面两张图展示了四种情况。

不平衡的节点是左重的。



不平衡的节点是右重的。

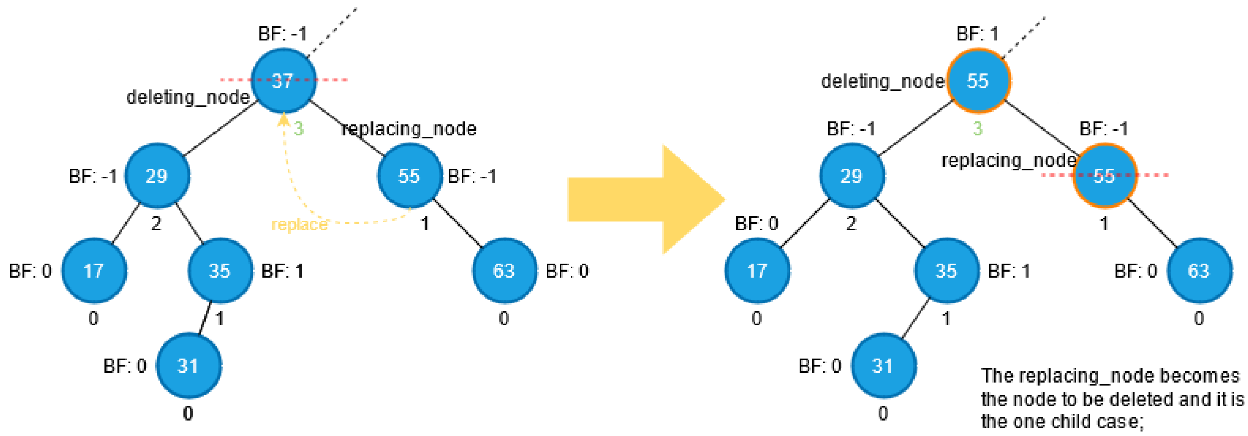




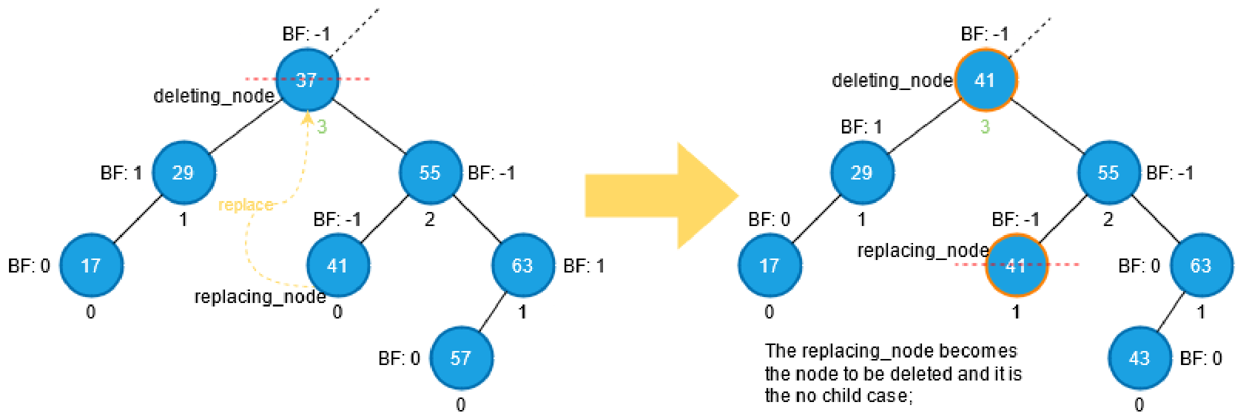
两个孩子

与其他二叉树的删除一样，我们可以将两个子树的情况看成两个子情况：替换节点是`deleting_node`的直接子节点，而`replacing_node`是`deleting_node`的最左边的节点。在任何一种情况下，我们都可以通过将 替换为`deleting_node`, `replacing_node`但保留原始高度和平衡因子，将两个孩子的案例转移到无孩子案例或独生子女案例。通过这样做，我们不会改变高度和平衡因素。之后，我们删除`replacing_node`，它变成无子情况或独子情况。下面的图片展示了两个孩子的删除是如何工作的，以及如何修复它的不平衡。

替换节点是删除节点的直接子节点。



替换节点是删除节点的最左边的节点。



完整修复操作的实现如下。与插入类似，我们在沿着树向上走时更新节点的高度。

Python 复制代码

```python
def _delete_fixup(self, fixing_node: Node) -> None:
    while fixing_node:
```

```
        fixing_node.height = 1 + max(
            self.get_height(fixing_node.left), self.get_height(fixing_node.right)
        )

        if self._get_balance_factor(fixing_node) > 1:
            # Case Left-Left
            if self._get_balance_factor(fixing_node.left) >= 0:
                self._right_rotate(fixing_node)
            # Case Left-Right
            elif self._get_balance_factor(fixing_node.left) < 0:
                # The fixing node's left child cannot be empty
                self._left_rotate(fixing_node.left)
                self._right_rotate(fixing_node)
        elif self._get_balance_factor(fixing_node) < -1:
            # Case Right-Right
            if self._get_balance_factor(fixing_node.right) <= 0:
                self._left_rotate(fixing_node)
            # Case Right-Left
            elif self._get_balance_factor(fixing_node.right) > 0:
                # The fixing node's right child cannot be empty
                self._right_rotate(fixing_node.right)
                self._left_rotate(fixing_node)

        fixing_node = fixing_node.parent
```

辅助功能

辅助函数，例如获取最左边的节点和获取节点的后继节点，与二叉搜索树：辅助函数相同，其实现可在Github 存储库中找到。与二叉搜索树不同的唯一辅助函数是获取高度的函数。

获取高度

由于每个节点都存储了其高度，因此获取节点的高度变得非常简单：只需返回高度即可。

Python 复制代码

```
@staticmethod
def get_height(node: Optional[Node]) -> int:
    if node:
        return node.height
    # None has height -1
    return -1
```

遍历

尽管 AVL 树节点比普通的二叉搜索树节点多了一个字段（即高度），但我们仍然可以使用我们在二叉树遍历中所做的确切实现来遍历 AVL 树。我们需要做的唯一修改是添加 AVL 树作为支持的类型。

Python 复制代码

```
# Alisa for the supported node types. For type checking.
SupportedNode = Union[None, binary_search_tree.Node, avl_tree.Node]

SupportedTree = Union[binary_search_tree.BinarySearchTree, avl_tree.AVLTree]
"""Alisa for the supported tree types. For type checking."""
```

（有关完整的源代码，请参阅traversal.py。）

支持的类型用于类型检查，正如我们在Binary Tree Traversal: Function Interface 中讨论的那样。

测试

与往常一样，我们应该尽可能多地对我们的代码进行单元测试。检查test_avl_tree.py以获取完整的单元测试。

分析

AVL 树是一种自平衡二叉搜索树，它的高度为 O(lg n)，其中 n 是节点的数量（这可以通过利用斐波那契数来证明。有关更多详细信息，请参阅AVL 树 Wikipedia）。因此，一棵AVL树的时间复杂度可以总结在下表中：

Operations	Average	Worst
Insert	$O(log_2 n)$	$O(log_2 n)$
Search	$O(log_2 n)$	$O(log_2 n)$
Delete	$O(log_2 n)$	$O(log_2 n)$
Leftmost (Min)	$O(log_2 n)$	$O(log_2 n)$
Rightmost (Max)	$O(log_2 n)$	$O(log_2 n)$
Predecessor	$O(log_2 n)$	$O(log_2 n)$
Successor	$O(log_2 n)$	$O(log_2 n)$

例子

与红黑树一样，AVL树因其自平衡能力而被广泛应用于软件程序中。例如，本节使用我们在这里实现的 AVL 树来实现一个 key-value `Map`。

Python 缩小▲ 复制代码

```python
from typing import Any, Optional

from forest.binary_trees import avl_tree
from forest.binary_trees import traversal

class Map:
    """Key-value Map implemented using AVL Tree."""

    def __init__(self) -> None:
        self._avlt = avl_tree.AVLTree()

    def __setitem__(self, key: Any, value: Any) -> None:
        """Insert (key, value) item into the map."""
        self._avlt.insert(key=key, data=value)

    def __getitem__(self, key: Any) -> Optional[Any]:
        """Get the data by the given key."""
        node = self._avlt.search(key=key)
        if node:
            return node.data
        return None

    def __delitem__(self, key: Any) -> None:
```

```
        """Remove a (key, value) pair from the map."""
        self._avlt.delete(key=key)

    def __iter__(self) -> traversal.Pairs:
        """Iterate the data in the map."""
        return traversal.inorder_traverse(tree=self._avlt)

    @property
    def empty(self) -> bool:
        """Return `True` if the map is empty; `False` otherwise."""
        return self._avlt.empty

if __name__ == "__main__":

    # Initialize the Map instance.
    contacts = Map()

    # Add some items.
    contacts["Mark"] = "mark@email.com"
    contacts["John"] = "john@email.com"
    contacts["Luke"] = "luke@email.com"

    # Retrieve an email
    print(contacts["Mark"])

    # Delete one item.
    del contacts["John"]

    # Check the deleted item.
    print(contacts["John"])  # This will print None

    # Iterate the items.
    for contact in contacts:
        print(contact)
```

（完整示例可在avlt_map.py 中找到。）

概括

与红黑树一样，AVL 树在插入和删除操作上引入了一些复杂度以保持其平衡，但是 AVL 树的自平衡能力为基本操作提供了 O(lg n) 的时间复杂度，这比常规操作具有更好的性能二叉搜索树。

（最初于2021 年 6 月 6 日在舜的葡萄园发表。）

历史

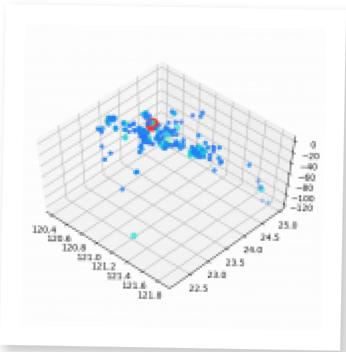
- 2021 年 6 月7 日：初始版本

执照

本文以及任何相关的源代码和文件均根据The Code Project Open License (CPOL)获得许可

分享

关于作者



顺煌

软件开发人员（高级）
美国

手表
该会员

我叫舜。我是一名软件工程师，也是一名基督徒。我目前在一家初创公司工作。
我的网站：https
://shunsvineyard.info 邮箱：zsh@shunsvineyard.info

评论和讨论

添加评论或问题 ?

电子邮件提醒 Search Comments

-- 本论坛暂无消息 --