SESSDSA'20

问题解答

BST和AVL树讲解

【H5】AVL树作业详解

数据结构与算法（Python）-17/0428

陈斌 gischen@pku.edu.cn 北京大学地球与空间科学学院

目录

问题解答

- 【H5】怎么做？
- del操作如何实现，操作平衡因子怎么写，del后怎么再平衡（8次）
- h5中hash值变化的问题，两个数据的哈希值大小关系也不固定，如何保证中序遍历的顺序
- 如果h5中添加的两个key值一样怎么办？在输出str是self[key]只能找到一个
  （类似问题：对于作业的自编字典，如果有两个相同的key，但对应的value不同，要求是按一般字典那样实现更新，还是两个键值都储存?）
- 自编字典是不是只有输出的key或value是字符串时要带自引号，其余的整数浮点数类型的都不用带?

问题解答

- 字典的输出，.keys()，.values()输出的顺序是不是没有要求?
- 请问.keys()方法和.values()方法是输出仅只需是一个表格吗?还是要像自带字典那样还要额外输出一个dickeys(再加上【表格】)
- 寻找右子树中的最小节点，以及迭代器，明明是在BST中进行的，相关代码为什么要写到Treenode类里面?
- 9. 如果要获得一个 treenode 的 value ，是直接使用node.value还是node.getValue ()比较好?虽然老师鼓励使用接口，但是这样一来就比较繁琐了(有时候会忘了加()导致出现bug)

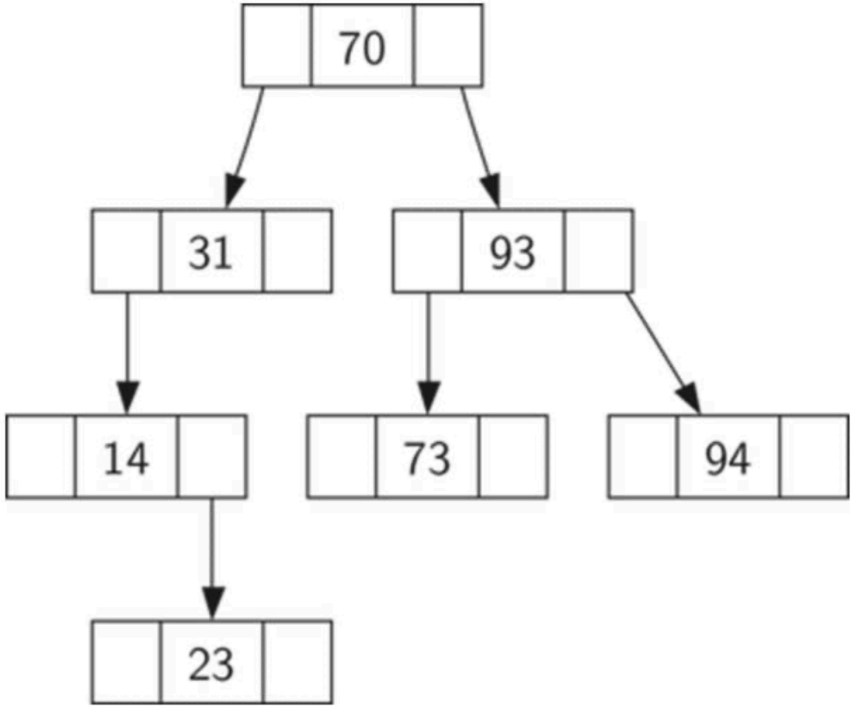
问题解答

- 请讲一下yield语句
- 迭代空树应该怎么写
- 树的高度包括根吗？希望有个实例
- getRoot()输出的样式是什么
- remove方法怎么update
- OJ详解

BST和AVL树

二叉查找树BST的性质

- 比父节点小的key都出现在左子树，比父节点大的key都出现在右子树。



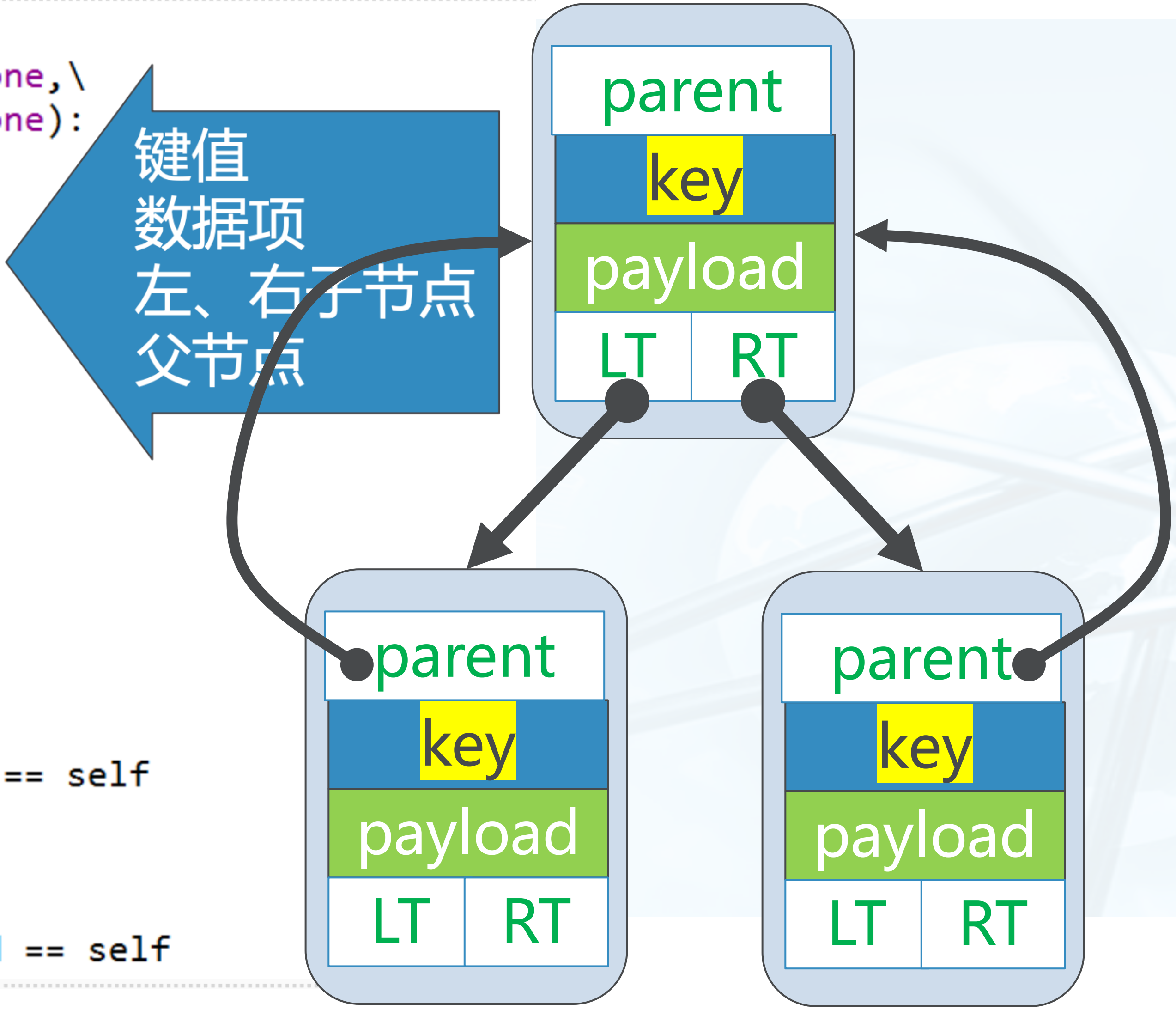
二叉搜索树的实现：节点和链接结构

- 需要用到BST和TreeNode两个类，BST的root成员引用根节点TreeNode

```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

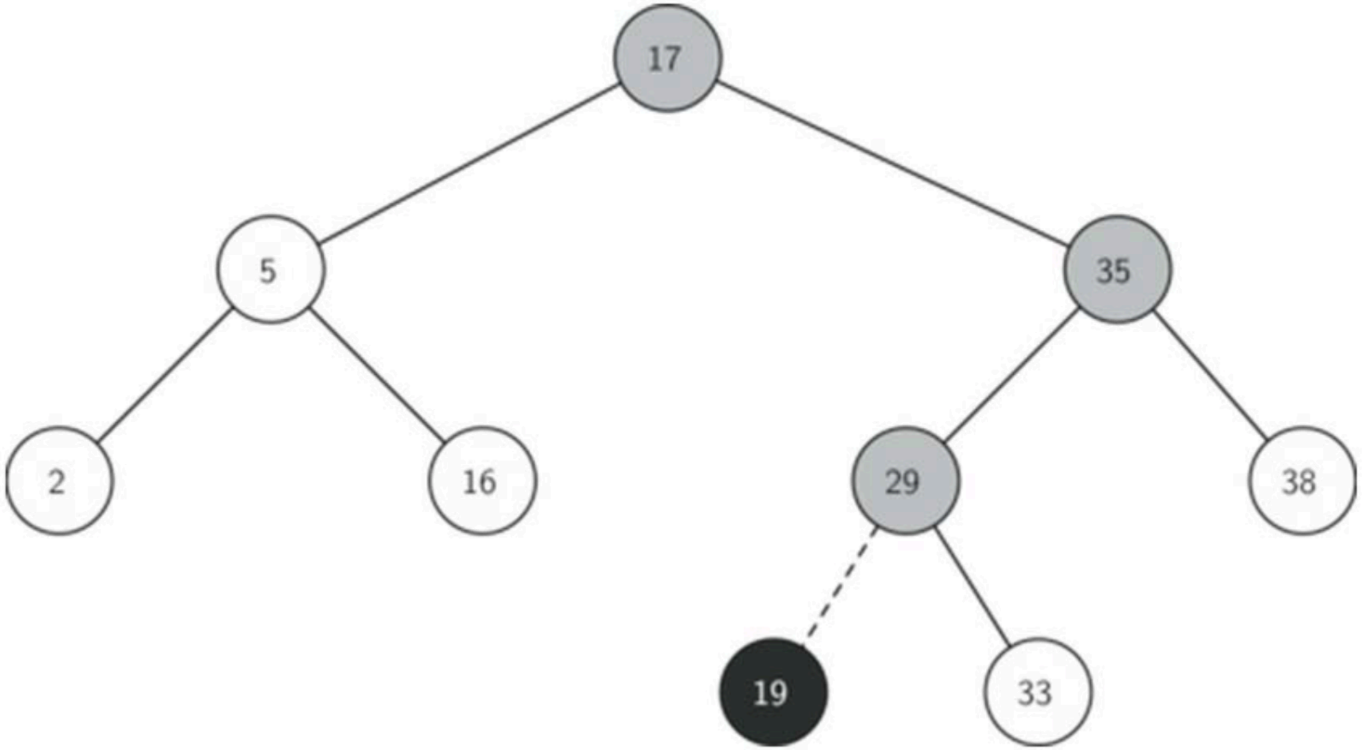
二叉搜索树的实现：TreeNode类

```python
class TreeNode:
    def __init__(self,key,val,left=None,\
                      right=None,parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and \
               self.parent.rightChild == self
```



二叉搜索树的实现：BST.put图示

❖ 插入key=19，currentNode的变化过程（灰色）：



二叉搜索树的实现：迭代器

❖ TreeNode类中的__iter__迭代器

迭代器函数中用了for迭代，实际上是递归函数

yield是对每次迭代的返回值

中序遍历的迭代

```
def __iter__(self):
    if self:
        if self.hasLeftChild():
                for elem in self.leftChild:
                    yield elem
        yield self.key
        if self.hasRightChild():
                for elem in self.rightChild:
                    yield elem
```

yield的另一个例子

```
def fib():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b

for fn in fib():
    print(fn)
    if fn> 100000:
        break

#fib 1, 1, 2, 3, 5, 8, 13, 21, 34, ……
#01) a, b
#02)    a, b
#03)       a, b
#04)          a, b
```

改成return?

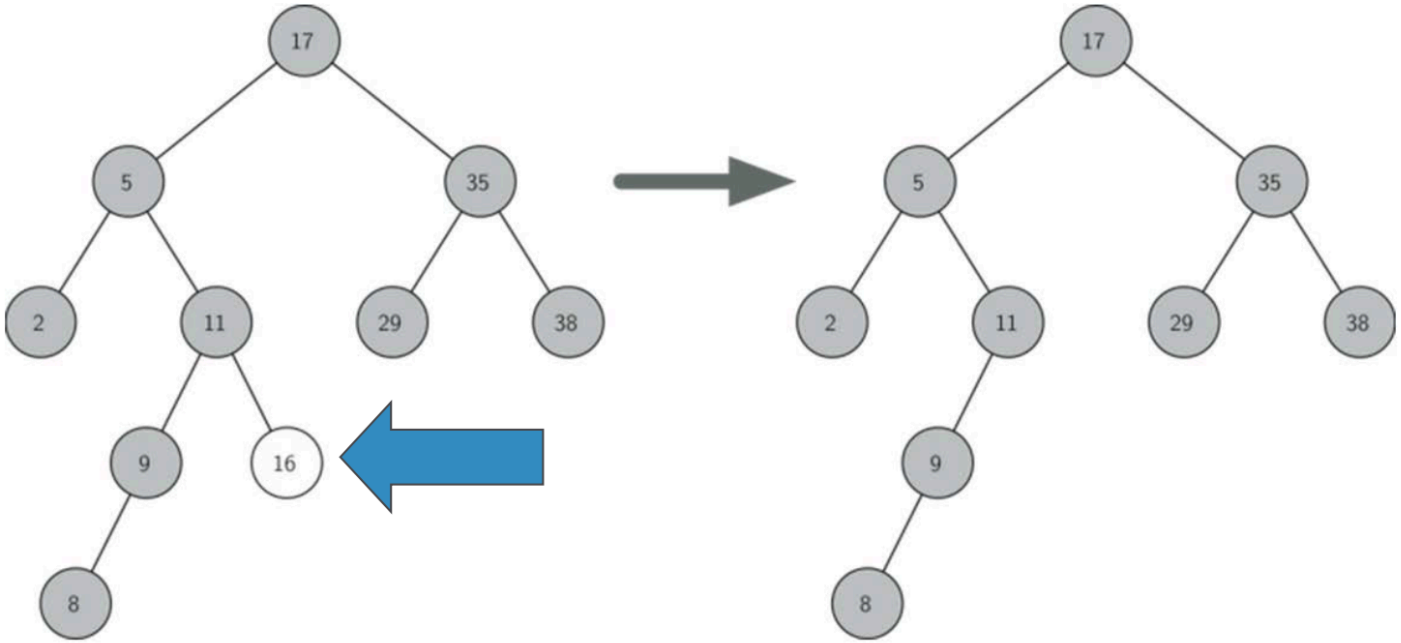
二叉查找树的实现：BST.remove方法

- 从BST中remove一个节点，还要求仍然保持BST的性质，分以下3种情形：
  - 这个节点没有子节点
  - 这个节点有1个子节点
  - 这个节点有2个子节点

二叉查找树的实现：BST.remove方法

❖ **没有**子节点的情况好办，直接删除

```
if currentNode.isLeaf(): #leaf
   if currentNode == currentNode.parent.leftChild:
       currentNode.parent.leftChild = None
   else:
       currentNode.parent.rightChild = None
```

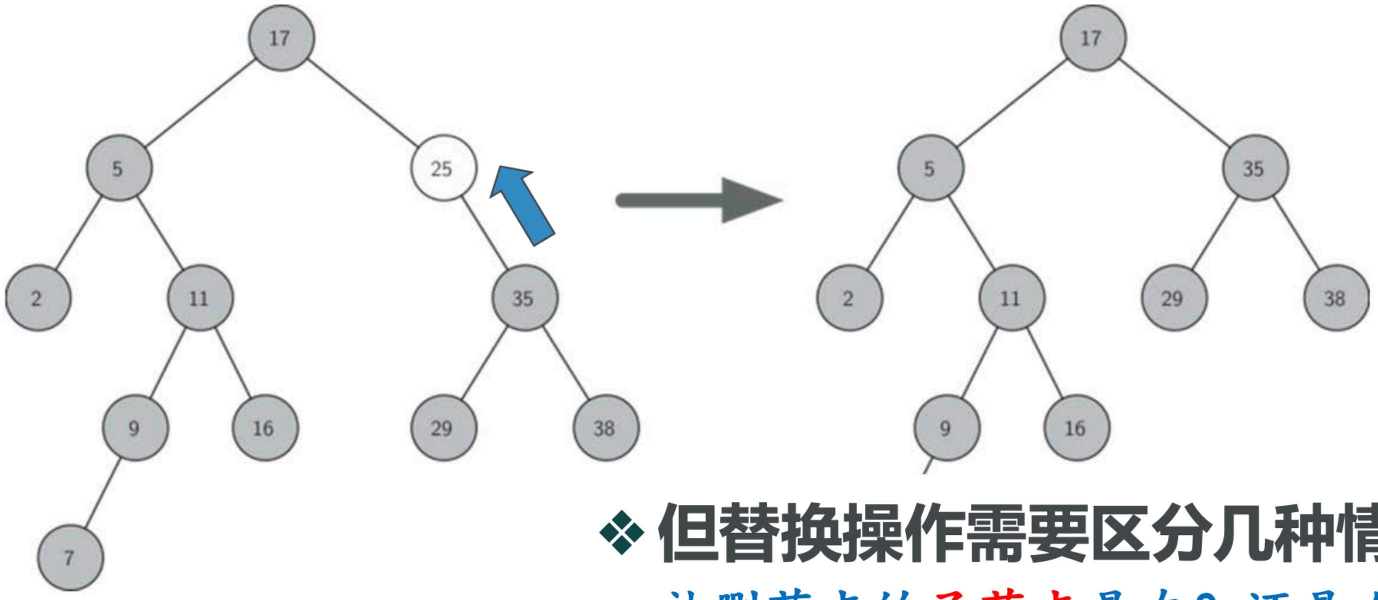


二叉查找树的实现：BST.remove方法

- ❖ **第2种情形稍复杂：被删节点有1个子节点**

  解决：将这个唯一的子节点上移，替换掉被删节点的位置

特别注意链接操作的次序！



- ❖ **但替换操作需要区分几种情况：**

  被删节点的子节点是左？还是右子节点？

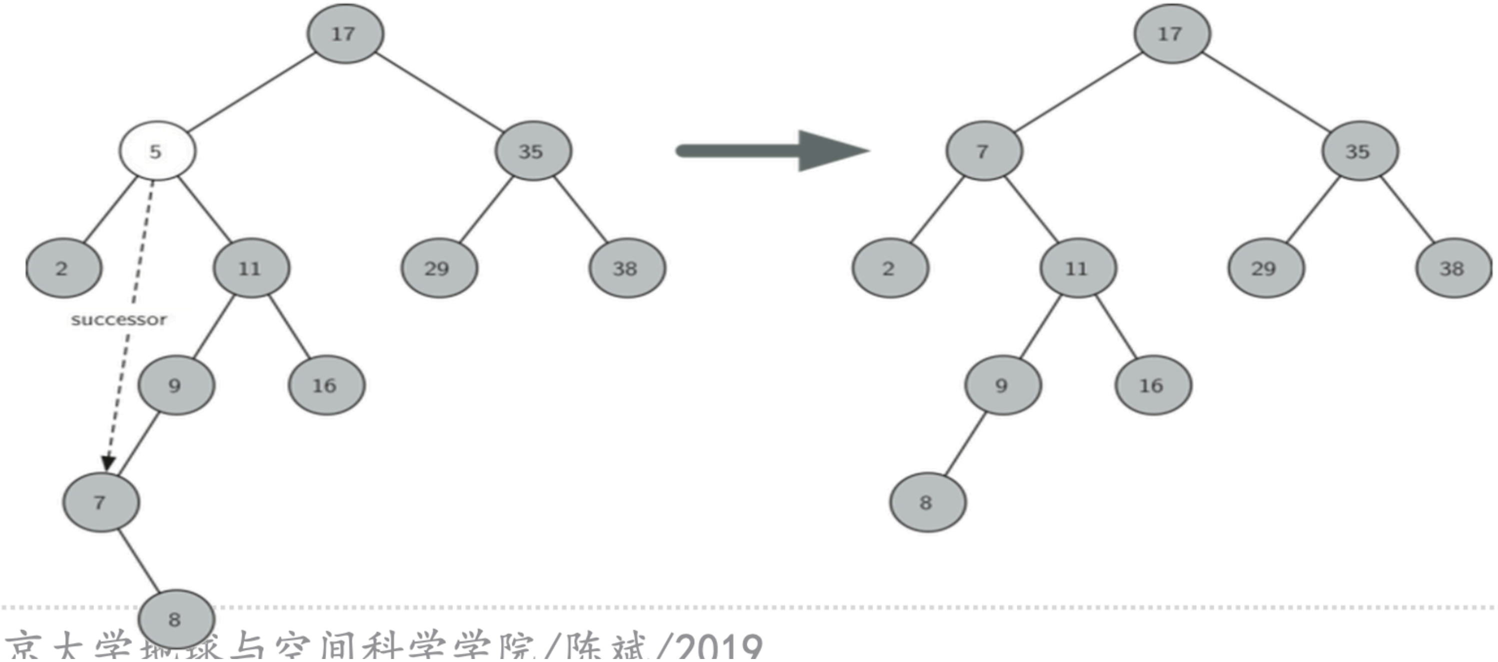
  被删节点本身是其父节点的左？还是右子节点？

  被删节点本身就是根节点？

❖ 第3种情形最复杂：被删节点有2个子节点

这时无法简单地将某个子节点上移替换被删节点

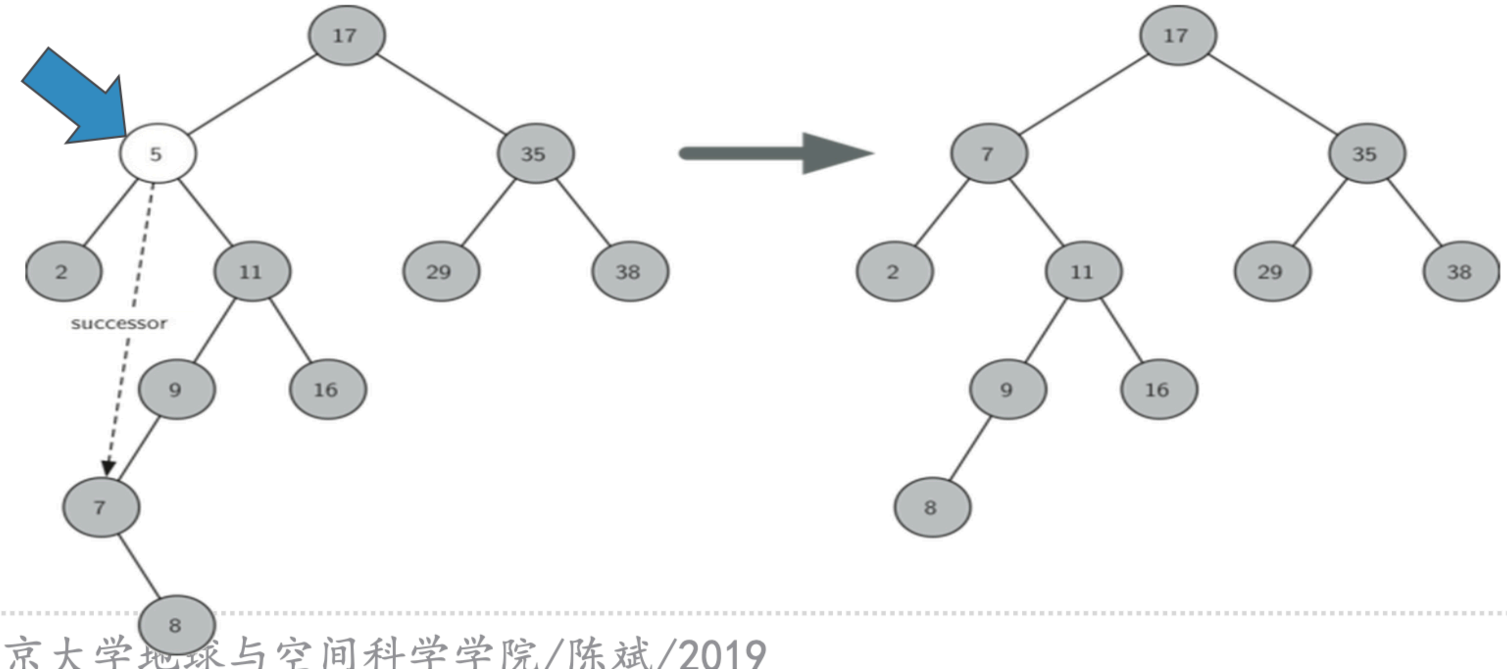
但可以找到另一个合适的节点来替换被删节点，这个合适节点就是被删节点的下一个key值节点，即被删节点右子树中最小的那个，称为“后继”



❖ 第3种情形最复杂：被删节点有2个子节点

可以肯定这个后继节点**最多**只有**1个**子节点（本身是叶节点，或仅有右子树）

将这个后继节点摘出来（也就是删除了），替换掉被删节点。



❖ TreeNode类：寻找后继节点

```
def findSuccessor(self):
  succ = None
  if self.hasRightChild():
      succ = self.rightChild.findMin()
  else:
      if self.parent:
            if self.isLeftChild():
                succ = self.parent
            else:
                self.parent.rightChild = None
                succ = self.parent.findSuccessor()
                self.parent.rightChild = self
  return succ

def findMin(self):
  current = self
  while current.hasLeftChild():
      current = current.leftChild
  return current
```

为什么？

目前不会遇到

到左下角

❖ TreeNode类：摘出节点spliceOut()

```
def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():
                self.parent.leftChild = None
        else:
                self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
                if self.isLeftChild():
                    self.parent.leftChild = self.leftChild
                else:
                    self.parent.rightChild = self.leftChild
                self.leftChild.parent = self.parent
        else:
                if self.isLeftChild():
                    self.parent.leftChild = self.rightChild
                else:
                    self.parent.rightChild = self.rightChild
                self.rightChild.parent = self.parent
```

摘出叶节点

为什么？

目前不会遇到

摘出带右子节点的节点

AVL树的要点

插入节点时，平衡因子会发生什么变化？

每个新插入的节点，都是**叶节点**，平衡因子=0

它作为左子节点（+1）或右子节点（-1），会让父节点平衡因子发生变化

如果父节点平衡因子**从原来的0变成非0**（+-1），表示这子树高度变化，平衡因子的影响需要向根方向传播

超出范围就需要rebalance

```python
def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)      # 重新平衡
        return
    if node.parent != None:
        if node.isLeftChild():
            node.parent.balanceFactor += 1
        elif node.isRightChild():
            node.parent.balanceFactor -= 1

        if node.parent.balanceFactor != 0:
            self.updateBalance(node.parent)      # 调整父节点因子
```

AVL树的实现：rebalance重新平衡

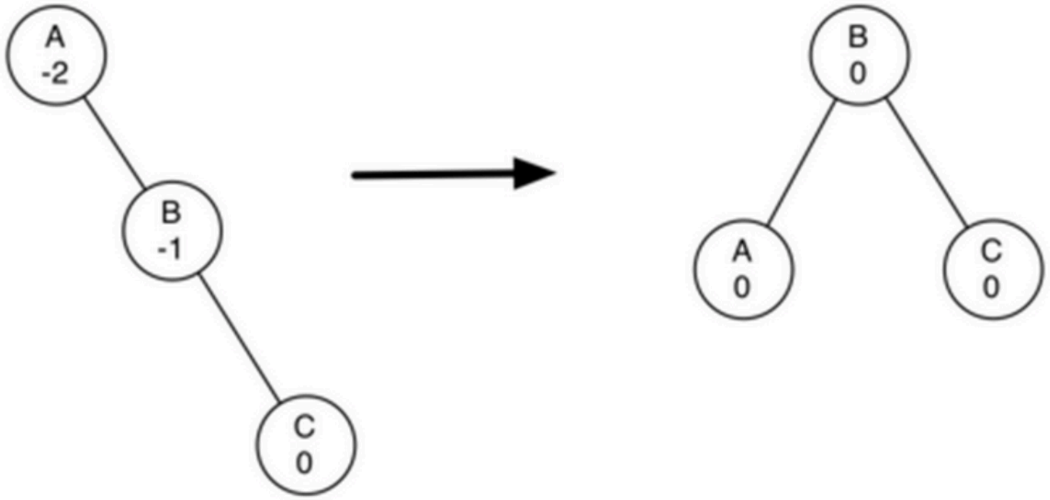
- **主要手段：将不平衡的子树进行旋转 rotation**

  视“左重”或者“右重”进行不同方向的旋转

  同时更新相关父节点引用，更新旋转后被影响节点的平衡因子

左旋转

右旋转



AVL树的实现：更复杂的情形

- **所以，在左旋转之前检查右子节点的因子**

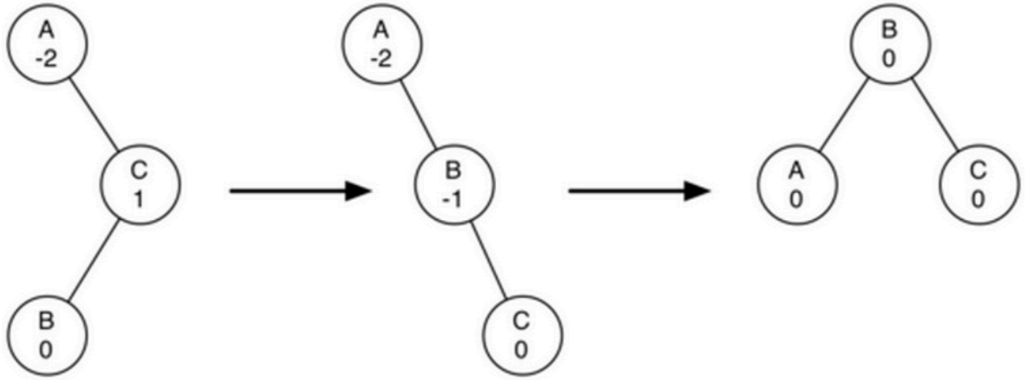
  如果右子节点“左重”的话，先对它进行右旋转

  再实施原来的左旋转

- **同样，在右旋转之前检查左子节点的因子**

  如果左子节点“右重”的话，先对它进行左旋转

  再实施原来的右旋转

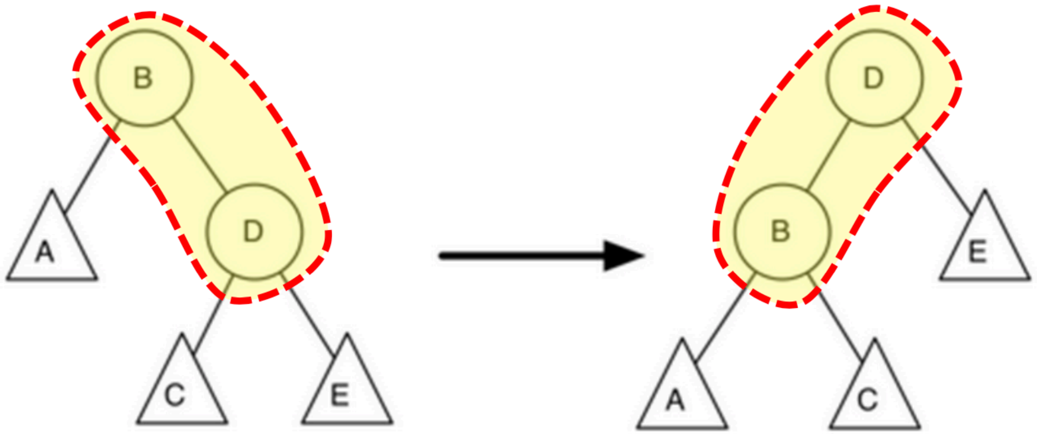


先右旋再左旋

先左旋再右旋

旋转后，两个相关节点平衡因子的调整

AVL树的实现：如何调整平衡因子



❖ 我们来看看B的变化

新B= hA- hC

旧B= hA- 旧hD

而：

旧hD= 1+ max(hC, hE)，所以旧B= hA- (1+ max(hC, hE))

新B- 旧B= 1+ max(hC, hE)- hC

新B= 旧B+ 1+ max(hC, hE)- hC；把hC移进max函数里就有

新B= 旧B+ 1+ max(0, -旧D) <==> <u>新B= 旧B+ 1- min(0, 旧D)</u>

```
rotRoot.balanceFactor = rotRoot.balanceFactor + \
                        1 - min(newRoot.balanceFactor, 0)
```

【H5】AVL树作业详解：实现mydict字典（ADT Map）

- **AVL树要求支持多种类型的key，多种类型value**
- **用AVL树保持各种操作的优良性能（log(n)）**

get/put/del/in/iter

mydict
- getRoot
- __init__
- __setitem__
- __getitem__
- __delitem__
- __len__
- __contains__
- clear
- __str__
- keys
- values

TreeNode
- __init__
- getLeft
- getRight
- isLeftChild
- isRightChild
- isRoot
- isLeaf
- hasAnyChildren
- hasBothChildren
- replaceNodeData
- replaceNodeValue
- __str__
- __iter__
- findSuccessor
- findMin
- spliceOut

AVLTree
- __init__
- __len__
- __iter__
- __str__
- put
- _put
- updateBalance
- rebalance
- rotateLeft
- rotateRight
- get
- _get
- __contains__
- delete
- remove
- updateBalanceRemove

```python
class mydict:
    """
    以AVL树作为内部实现的字典
    """
    def getRoot(self):  # 返回内部的AVL树根
        return self.avlTree.root

    def __init__(self):  # 创建一个空字典
        self.avlTree = AVLTree()

    def __setitem__(self, key, value):  # 将key:value保存到字典
        # md[key]=value
        self.avlTree[key] = value

    def __getitem__(self, key):  # 从字典中根据key获取value
        # v = md[key]
        # key在字典中不存在的话，请raise KeyError
        return self.avlTree[key]

    def __delitem__(self, key):  # 删除字典中的key
        # del md[key]
        # key在字典中不存在的话，请raise KeyError
        del self.avlTree[key]

    def __len__(self):  # 获取字典的长度
        # l = len(md)
        return len(self.avlTree)
```

```python
    def __contains__(self, key):  # 判断字典中是否存在key
        # k in md
        return key in self.avlTree

    def clear(self):  # 清除字典
        self.avlTree = AVLTree()

    def __str__(self):  # 输出字符串形式，参照内置dict类型，输出按照AVL树中序遍历次序
        # 格式类似：{'name': 'sessdsa', 'hello': 'world'}
        return str(self.avlTree)
    __repr__ = __str__

    def keys(self):  # 返回所有的key，类型是列表，按照AVL树中序遍历次序
        retlist = []
        for k in self.avlTree:
            retlist.append(k)
        return retlist

    def values(self):  # 返回所有的value，类型是列表，按照AVL树中序遍历次序
        retlist = []
        for k in self.avlTree:
            retlist.append(self.avlTree[k])
        return retlist
```

```python
class TreeNode:
    """
    二叉树节点
    请自行完成节点内部的实现，并实现给出的接口
    """
    def __init__(self, key, val, left=None, right=None, parent=None):  # 初始化方法
        self.key = hash(key)
        self.ori_key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent
        self.balanceFactor = 0

    def getLeft(self):  # 获取左子树 (不存在时返回None)
        return self.leftChild

    def getRight(self):  # 获取右子树 (不存在时返回None)
        return self.rightChild

    hasLeftChild = getLeft
    hasRightChild = getRight

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self
```

```python
    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.leftChild or self.rightChild)

    def hasAnyChildren(self):
        return self.leftChild or self.rightChild

    def hasBothChildren(self):
        return self.leftChild and self.rightChild

    def replaceNodeData(self, key, val, lc, rc):
        self.key = hash(key)
        self.ori_key = key
        self.payload = val
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self

    def replaceNodeValue(self, key, val):
        self.key = hash(key)
        self.ori_key = key
        self.payload = val
```

key是hash之后的，仅用于比较

ori_key用来输入/输出

iter和yield

```python
    def __str__(self):
        # 层次结构嵌套列表方式输出TreeNode和所有子节点
        lcstr = f"[{str(self.leftChild)}]" if self.leftChild else "[]"
        rcstr = f"[{str(self.rightChild)}]" if self.rightChild else "[]"
        return f"[{repr(self.ori_key)}/{self.balanceFactor}, {lcstr}, {rcstr}]"
    __repr__ = __str__

    def __iter__(self):
        # 中序遍历输出
        if self.hasLeftChild():
            for ori_key in self.leftChild:
                yield ori_key
        yield self.ori_key
        if self.hasRightChild():
            for ori_key in self.rightChild:
                yield ori_key
```

```python
    def findSuccessor(self):  # 找到后继节点
        succ = None
        if self.hasRightChild():
            succ = self.rightChild.findMin()
        else:
            if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
        return succ

    def findMin(self):  # 找到子树上最小的节点
        current = self
        while current.hasLeftChild():
            current = current.leftChild
        return current
```

```python
def spliceOut(self):  # 摘掉节点
    if self.isLeaf():
        if self.isLeftChild():
            self.parent.leftChild = None
        else:
            self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
            if self.isLeftChild():
                self.parent.leftChild = self.leftChild
            else:
                self.parent.rightChild = self.leftChild
            self.leftChild.parent = self.parent
        else:
            if self.isLeftChild():
                self.parent.leftChild = self.rightChild
            else:
                self.parent.rightChild = self.rightChild
            self.rightChild.parent = self.parent
```

```python
class AVLTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size

    def __iter__(self):
        if self.root:  # 如果是空树，就啥都不返回
            return iter(self.root)
        return iter([])

    def __str__(self):  # 字典方式输出AVL树
        allstr = ", ".join([f"{repr(key)}: {repr(self[key])}" for key in self])
        return f"{{{allstr}}}"
```

key是hash之后的，仅用于比较

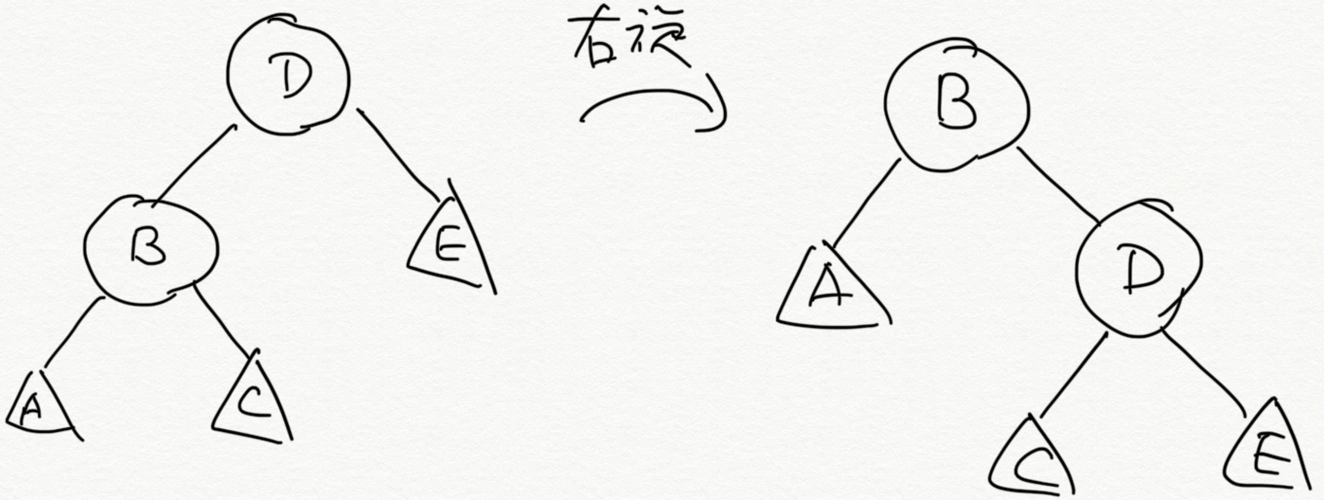
ori_key用来输入/输出

```python
    def put(self, key, val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = TreeNode(key, val)
            self.size = 1  # 注意这里的size初始化1
    __setitem__ = put

    def _put(self, key, val, currentNode):
        ori_key = key
        key = hash(ori_key)
        if key < currentNode.key:  # 用hash后的key进行对比
            if currentNode.hasLeftChild():
                self._put(ori_key, val, currentNode.leftChild)
            else:
                currentNode.leftChild = TreeNode(ori_key, val, parent=currentNode)
                self.size += 1  # 这里才会增加一个节点
                self.updateBalance(currentNode.leftChild)
        elif key > currentNode.key:  # 用hash后的key进行对比
            if currentNode.hasRightChild():
                self._put(ori_key, val, currentNode.rightChild)
            else:
                currentNode.rightChild = TreeNode(ori_key, val, parent=currentNode)
                self.size += 1  # 这里才会增加一个节点
                self.updateBalance(currentNode.rightChild)
        else: #   # 用hash后的key进行对比，相等，替换
            currentNode.replaceNodeValue(ori_key, val)  # 替换，不增加节点
```

```python
    def updateBalance(self, node):
        if node.balanceFactor > 1 or node.balanceFactor < -1:
            self.rebalance(node)
            return
        if node.parent:
            if node.isLeftChild():  # 添加为左子节点了
                node.parent.balanceFactor += 1
            elif node.isRightChild():  # 添加为右子节点了
                node.parent.balanceFactor -= 1
            # 如果父节点没有消化这个新增的平衡因子，就向上传播
            if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)

    def rebalance(self, node):
        if node.balanceFactor < 0:  # 右重
            if node.rightChild.balanceFactor > 0:  # 右子节点左重先右旋
                self.rotateRight(node.rightChild)
                self.rotateLeft(node)  # 左旋
            else:
                self.rotateLeft(node)
        elif node.balanceFactor > 0:  # 左重
            if node.leftChild.balanceFactor < 0:  # 左子节点右重先左旋
                self.rotateLeft(node.leftChild)
                self.rotateRight(node)  # 右旋
            else:
                self.rotateRight(node)  # 右旋
```

```python
    # 左旋并调整平衡因子
    def rotateLeft(self, rotRoot):
        newRoot = rotRoot.rightChild  # newRoot是旋转节点的右子节点
        rotRoot.rightChild = newRoot.leftChild
        if newRoot.hasLeftChild():  # newRoot的左子树挂到旋转节点的右边
            newRoot.leftChild.parent = rotRoot
        newRoot.parent = rotRoot.parent  # newRoot继承旋转节点的父节点
        # 调整旋转节点父节点的对应引用到newRoot
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        # newRoot和旋转节点的父子关系对调
        newRoot.leftChild = rotRoot
        rotRoot.parent = newRoot
        # 调整旋转节点的平衡因子
        rotRoot.balanceFactor = rotRoot.balanceFactor + 1 \
                                - min(newRoot.balanceFactor, 0)
        # 调整newRoot的平衡因子
        newRoot.balanceFactor = newRoot.balanceFactor + 1 \
                                + max(rotRoot.balanceFactor, 0)
```

```python
    def rotateRight(self, rotRoot):
        newRoot = rotRoot.leftChild  # newRoot是旋转节点的左子节点
        rotRoot.leftChild = newRoot.rightChild
        if newRoot.hasRightChild():  # newRoot的右子树挂到旋转节点的左边
            newRoot.rightChild.parent = rotRoot
        newRoot.parent = rotRoot.parent  # newRoot继承旋转节点的父节点
        # 调整旋转节点父节点的对应引用到newRoot
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        # newRoot和旋转节点的父子关系对调
        newRoot.rightChild = rotRoot
        rotRoot.parent = newRoot
        # 调整旋转节点的平衡因子
        rotRoot.balanceFactor = rotRoot.balanceFactor - 1\
                                - max(newRoot.balanceFactor, 0)
        # 调整newRoot的平衡因子
        newRoot.balanceFactor = newRoot.balanceFactor - 1\
                                + min(rotRoot.balanceFactor, 0)
```



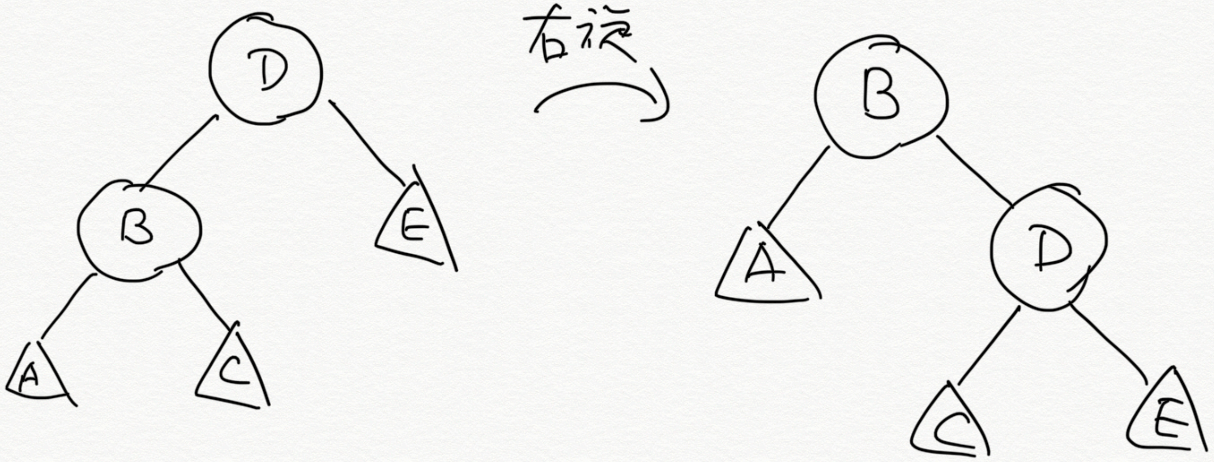
D : rotRoot

B : newRoot

$$nBal(D) = h_C - h_E$$

$$oBal(D) = h_B - h_E$$
$$= (1 + \max(h_A, h_C)) - h_E$$

$$nBal(D) - oBal(D) = h_C - (1 + \max(h_A, h_C))$$
$$nBal(D) = oBal(D) - 1 + h_C - \max(h_A, h_C)$$
$$= oBal(D) - 1 - (\max(h_A, h_C) - h_C)$$
$$= oBal(D) - 1 - \max(h_A - h_C, 0)$$
$$= oBal(D) - 1 - \max(oBal(B), 0)$$



$$nBal(B) = h_A - h_D = h_A - (1+\max(h_C, h_E))$$

$$oBal(B) = h_A - h_C$$

$$nBal(B) - oBal(B) = h_C - (1+\max(h_C, h_E))$$

$$\begin{aligned} nBal(B) &= oBal(B) - 1 + h_C - \max(h_C, h_E) \\ &= oBal(B) - 1 - (\max(h_C, h_E) - h_C) \\ &= oBal(B) - 1 - \max(0, h_E - h_C) \\ &= oBal(B) - 1 + \min(0, nBal(D)) \end{aligned}$$

注意：新Bal(D)

```python
    def get(self, key):
        return self._get(key, self.root).payload
    __getitem__ = get

    def _get(self, key, currentNode):
        ori_key = key
        key = hash(ori_key)
        if not currentNode:
            raise KeyError(ori_key)
        elif key == currentNode.key:  # 用hash后的key进行对比
            return currentNode
        elif key < currentNode.key:  # 用hash后的key进行对比
            return self._get(ori_key, currentNode.leftChild)
        else:  # 用hash后的key进行对比 key >
            return self._get(ori_key, currentNode.rightChild)

    def __contains__(self, key):
        try:
            v = self[key]
        except KeyError:
            return False
        return True
```

```python
    def delete(self, key):
        ori_key = key
        key = hash(ori_key)
        if self.size > 1:
            nodeToRemove = self._get(ori_key, self.root)
            if nodeToRemove:
                self.remove(nodeToRemove)
                self.size -= 1
            else:
                raise KeyError(ori_key)
        elif self.size == 1 and self.root.key == key:  # hash后的key对比
            self.root = None
            self.size = 0
        else:
            raise KeyError(ori_key)

    __delitem__ = delete
```

```python
    def remove(self, currentNode):
        if currentNode.isLeaf():  # 删除没有子节点的叶节点
            if currentNode.isLeftChild():
                currentNode.parent.leftChild = None
                currentNode.parent.balanceFactor -= 1  # 父节点少了左子节点
            else:
                currentNode.parent.rightChild = None
                currentNode.parent.balanceFactor += 1  # 父节点少了右子节点
            self.updateBalanceRemove(currentNode.parent)  # 从删除节点的父节点开始重新平衡
        elif currentNode.hasBothChildren():  # 删除有两个子节点的内部节点
            succ = currentNode.findSuccessor()  # 因为有两个子节点，其后继一定在右子树的左下角
            # 这个succ一定没有左子节点
            if succ.isLeftChild():
                succ.parent.balanceFactor -= 1
            else:
                succ.parent.balanceFactor += 1
            succ.spliceOut()
            currentNode.replaceNodeValue(succ.ori_key, succ.payload)
            self.updateBalanceRemove(succ.parent)  # 从摘除节点的父节点开始重新平衡
        else:  # 删除只有一个子节点的内部节点
```

```python
        else:  # 删除只有一个子节点的内部节点
            if currentNode.hasLeftChild():  # 只有左子节点
                if currentNode.isLeftChild():  # 自己是左子节点
                    currentNode.leftChild.parent = currentNode.parent
                    currentNode.parent.leftChild = currentNode.leftChild
                    currentNode.parent.balanceFactor -= 1  # 父节点少了左子节点
                elif currentNode.isRightChild():  # 自己是右子节点
                    currentNode.leftChild.parent = currentNode.parent
                    currentNode.parent.rightChild = currentNode.leftChild
                    currentNode.parent.balanceFactor += 1  # 父节点少了右子节点
                else:  # 自己是根结点
                    currentNode.leftChild.parent = None
                    self.root = currentNode.leftChild
            else:  # 只有右子节点
                if currentNode.isLeftChild():  # 自己是左子节点
                    currentNode.rightChild.parent = currentNode.parent
                    currentNode.parent.leftChild = currentNode.rightChild
                    currentNode.parent.balanceFactor -= 1  # 父节点少了左子节点
                elif currentNode.isRightChild():  # 自己是右子节点
                    currentNode.rightChild.parent = currentNode.parent
                    currentNode.parent.rightChild = currentNode.rightChild
                    currentNode.parent.balanceFactor += 1  # 父节点少了右子节点
                else:  # 自己是根结点
                    currentNode.rightChild.parent = None
                    self.root = currentNode.rightChild

            if currentNode.parent:  # 如果不是根结点就要重新平衡
                self.updateBalanceRemove(currentNode.parent)  # 从删除节点的父节点开始重新平衡
```

```python
    def updateBalanceRemove(self, node):
        if node.balanceFactor > 1 or node.balanceFactor < -1:
            self.rebalance(node)
            # 重新平衡后，继续看newRoot的平衡因子，如果变为0，还要继续传播
            node = node.parent
        if node.balanceFactor == 0 and node.parent:
            if node.isLeftChild():  # 左子节点少了
                node.parent.balanceFactor -= 1
            elif node.isRightChild():  # 右子节点少了
                node.parent.balanceFactor += 1
            # 向上传播
            self.updateBalanceRemove(node.parent)
```