数据结构和算法
（Python描述）

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

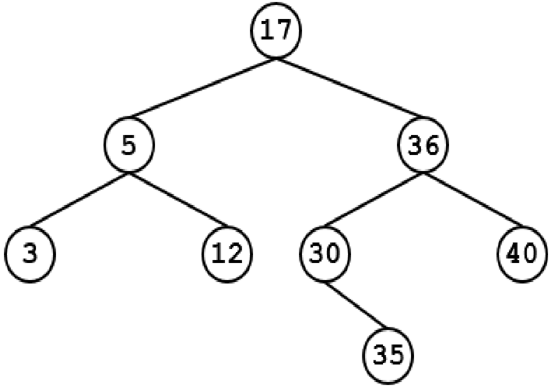
二叉排序树和平衡二叉树

二叉排序树
的概念和操作

福建福鼎太姥山

二叉排序树(二叉查找树)

- 是一棵二叉树
- 每个节点存储关键字(key)和值(value)两部分数据
- 对每个节点X，其左子树中的全部节点的key都小于X的key，且X的key小于其右子树中的全部节点的key
- 一个二叉搜索树中的任意一棵子树都是二叉搜索树



性质：一个二叉树是二叉搜索树，当且仅当其中序遍历序列是递增序列

略作修改就可以处理树节点key可以重复的情况。

二叉排序树的查找

递归过程，查找key为X的节点的value

- 如果X和根节点相等，则返回根节点的value，查找结束
- 如果X比根节点小，则递归进入左子树查找
- 如果X比根节点大，则递归进入右子树查找

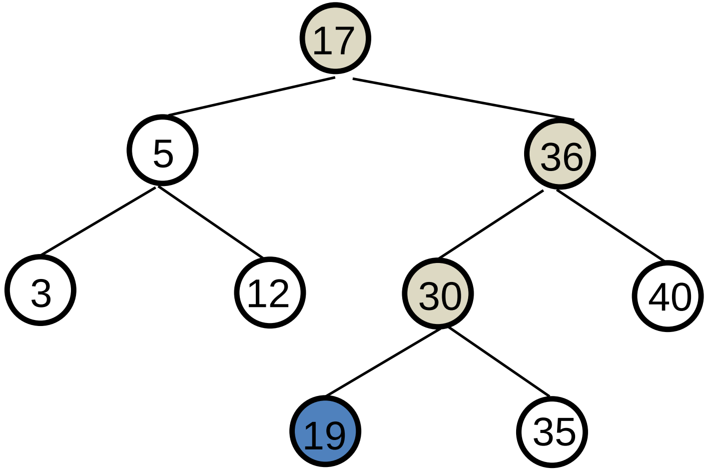
二叉排序树插入节点

递归过程，插入key为X的节点

- 如果X和根节点相等，则更改根节点的value
- 如果X比根节点小，则递归插入到左子树。如果没有左子树，则新建左子节点，存放要插入的key和value，插入工作结束。
- 如果X比根节点大，则递归插入到右子树。如果没有右子树，则新建右子节点，存放要插入的key和value，插入工作结束。

二叉排序树插入节点

插入19:



二叉排序树删除节点

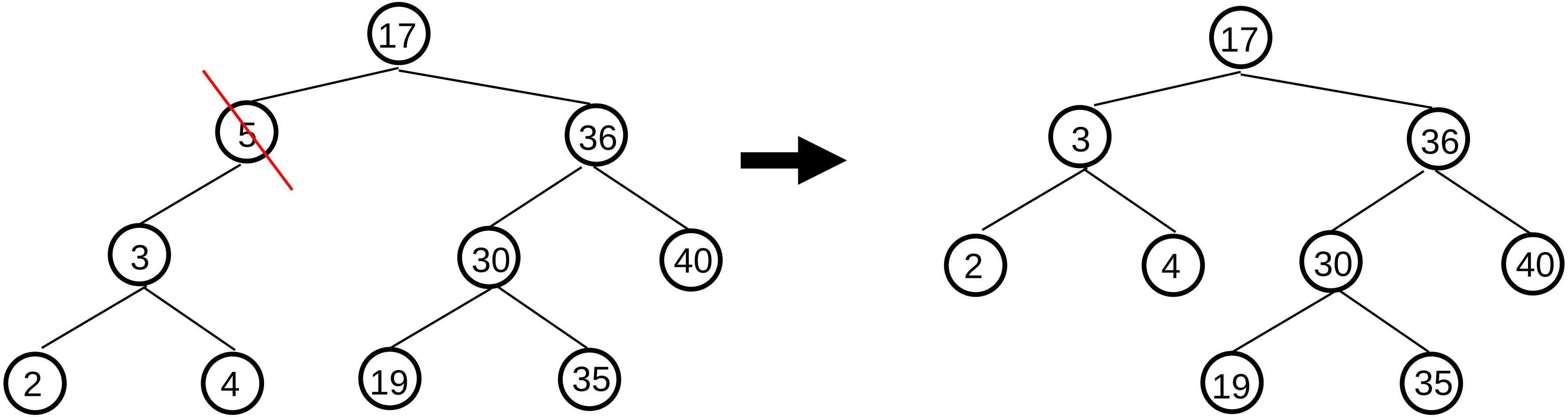
删除节点X，可以递归实现。分以下几种情况讨论：

1) X是叶子节点：直接删除，即X的父节点去掉X这个子节点

二叉排序树删除节点

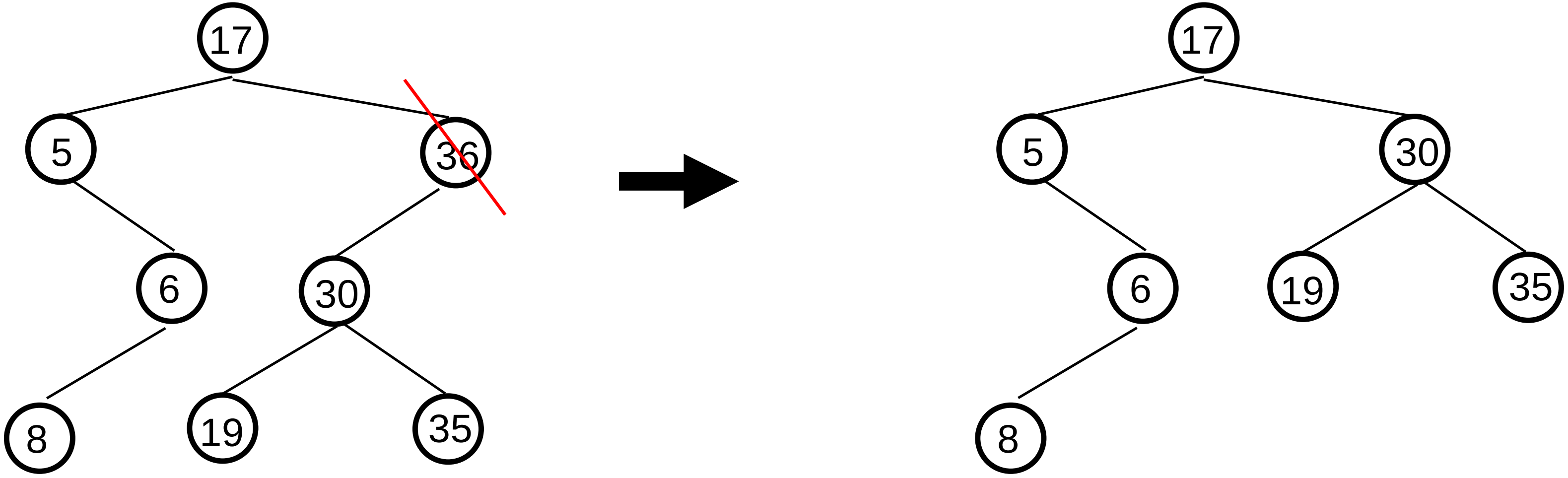
2) X只有左子节点，则其左子节点取代X的地位

若X是父亲的左儿子，则X的左儿子作为X父亲的新左儿子



二叉排序树删除节点

2) X只有左子节点，则其左子节点取代X的地位
若X是父亲的右儿子，则X的左儿子作为X父亲的新右儿子



二叉排序树删除节点

2) X只有左子节点，则其左子节点取代X的地位

若X没父亲，即X是树根，则X的左儿子成为新的树根

二叉排序树删除节点

3) X只有右子节点：则其右子节点取代X的地位

若X是父亲的左儿子，则X的右儿子作为X父亲的新左儿子
若X是父亲的右儿子，则X的右儿子作为X父亲的新右儿子
若X没父亲，即是树根，则X的右儿子成为新的树根

二叉排序树删除节点

4）X既有左子节点，又有右子节点，有两种做法：

做法1： 找到X的中序遍历后继节点，即X右子树中最小的节点Y，用Y的key和value覆盖X中的key和value，然后递归删除Y。

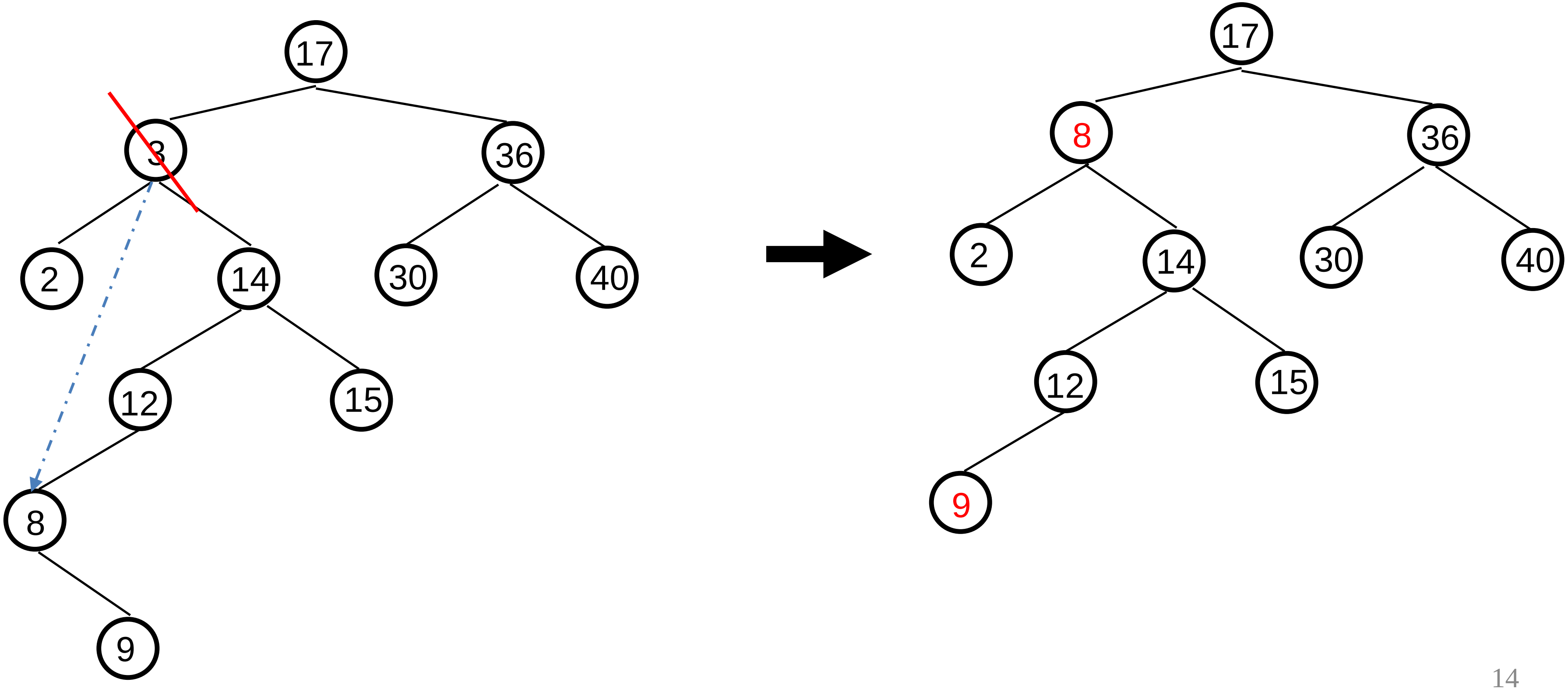
如何找Y: 进入X的右子节点，然后不停往左子节点走，直到没有左子节点为止。

做法2：找到X的中序遍历前驱节点，即X左子树中最大的节点Y，用Y的key和value覆盖X中的key和value，然后递归删除Y。

如何找Y: 进入X的左子节点，然后不停往右子节点走，直到没有右子节点为止。

二叉排序树删除节点

4) X左右子节点都有



二叉排序树删除节点

4）X既有左子节点，又有右子节点:

找到X的中序遍历后继节点，即X右子树中最小的节点Y，用Y的key和value覆盖X中的key和value，然后递归删除Y。

如何找Y: 进入X的右子节点，然后不停往左子节点走，直到没有左子节点为止。

二叉排序树的实现

二叉排序树的实现

```
class BinarySearchTree:
    class Node:   # 结点类
        def __init__(self, key, data, left=None, right=None):
            self.key, self.value, self.left, self.right = \
               key, data, left, right

    def __init__(self,less=lambda x,y:x<y ):
        self.root,self.size = None,0 #root是树根, size是结点总数
        self.less = less             #less是比较函数
```

二叉排序树的实现

```
def _find(self,key): #查找值为key的结点，返回值是找到的结点及其父亲
    def find(root,father): #在以root为根的子树中查找
        #father 是root的父亲，返回找到的结点及其父亲
        if self.less(key, root.key):
            if root.left:
                return find(root.left, root)
            else:
                return None,None   #找不到
        elif self.less(root.key, key):
            if root.right:
                return find(root.right,root)
            else:
                return None,None
        else:
            return root, father
    if self.root is None:
        return None,None
    return find(self.root,None)
```

二叉排序树的实现

```
def _insert(self,key,data): #插入结点(key,data)
    def insert(root): #返回值表示是否插入了新结点
        if self.less(key, root.key):
            if root.left is None:
                root.left = BinarySearchTree.Node(key, data)
                return True #插入了新结点
            else:
                return insert( root.left)
        elif self.less(root.key, key):
            if root.right is None:
                root.right = BinarySearchTree.Node(key, data)
                return True
            else:
                return insert(root.right)
        else:
            root.value = data  # 相同关键字，则更新
            return False
```

二叉排序树的实现

```
        if self.root is None:
            self.root = BinarySearchTree.Node(key,data)
            self.size = 1
        else:
            self.size += insert(self.root)

    def _findMin(self,root,father):  #找以root为根的子树的最小结点及其父亲
        #father是root的父亲
        if root.left is None:
            return root,father
        else:
            return self._findMin(root.left,root)
    def _findMax(self,root,father):  #找以root为根的子树的最大结点及其父亲
        if root.right is None:
            return root,father
        else:
            return self._findMax(root.right,father)
```

二叉排序树的实现

```
def pop(self,key):
    #删除键为key的结点，返回该结点的data。如果没有这样的元素，则引发异常
    nd,father = self._find(key)
    if nd is None:
        raise Exception("key not found")
    else:
        self.size -= 1
        self._deleteNode(nd,father)
        return nd.value
```

```
def _deleteNode(self,nd,father):  #删除结点nd,nd的父结点是father
    if nd.left and nd.right:  #nd左右子树都有
        minNd,father = self._findMin(nd.right,nd)
        nd.key,nd.value = minNd.key,minNd.value
        self._deleteNode(minNd,father)
    elif nd.left: #nd只有左子树
        if father and father.left is nd:  #nd是父亲的左儿子
            father.left = nd.left
        elif father  and father.right is nd:
            father.right = nd.left
        else:  #nd是树根
            self.root = nd.left
    elif nd.right :  #nd只有右子树
        if father and father.right is nd:
            father.right = nd.right
        elif father and father.left is nd:
            father.left = nd.right
        else:
            self.root = nd.right
```

```
        else: #nd是叶子
            if father and father.left is nd:
                father.left = None
            elif father and father.right is nd:
                father.right = None
            else: #nd是树根
                self.root = None

    def _inorderTraversal(self):
        def inorderTraversal(root):
            if root.left:
                yield from inorderTraversal(root.left)
            yield root.key,root.value
            if root.right:
                yield from inorderTraversal(root.right)
        if self.root is None:
            return
        yield from inorderTraversal(self.root)
```

```
def __contains__(self, key): #实现 in
    return self._find(key)[0] is not None
def __iter__(self): #返回迭代器
    return self._inorderTraversal()
def __getitem__(self,key):  #实现右值 []
    nd,father = self._find(key)
    if nd is None:
        raise Exception("key not found")
    else:
        return nd.value
def __setitem__(self, key, data): #实现左值 []
    nd,father = self._find(key)
    if nd is None:
        self._insert(key,data)
    else:
        nd.value = data
def __len__(self):
    return self.size
```

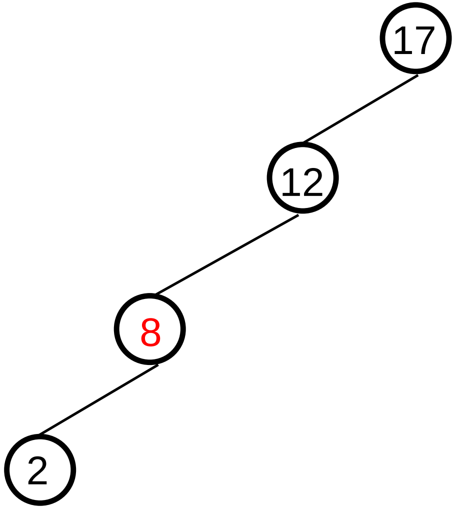
```
import random
random.seed(2)
s = [i for i in range(8)]
tree = BinarySearchTree()
#若 tree = Tree(lambda x ,y : y <x) 则从大到小排
random.shuffle(s)
for x in s:
    tree[x] = x        #加入关键字为x，值为x的元素
print(len(tree))     #>>8
for x in tree:   #首先会调用tree.__iter__()返回一个迭代器
    print(f"({x[0]},{x[1]})",end = "")  #从小到大遍历整个树
#>>(0,0)(1,1)(2,2)(3,3)(4,4)(5,5)(6,6)(7,7)
print()
print(3000 in tree)   #>>False
print(3 in tree)       #>>True
print(tree[3])         #>>3  输出关键字为3的元素的值
```

```
try:
    print(tree[3000])   #关键字为3000的元素不存在，此句引发异常
except Exception as e:
    print(e)            #>>key not found
tree[3000] = "ok" #添加关键字为3000，值为"ok"的元素
print(tree[3000],len(tree))     #>>ok 9
tree[3000] = "bad"    #将关键字为3000的元素的值改为"bad"
print(tree[3000],len(tree))     #>>bad 9
try:
    tree.pop(354)     #关键字为354的元素不存在，此句引发异常
except Exception as e:
    print(e)                 #>>key not found
tree.pop(3)
print(len(tree))            #>>8
```

- 此二叉排序树，不允许节点关键字重复
- 要支持重复关键字，可以改成val部分是可以包含多个值的列表

二叉树排序树复杂度

- 二叉排序树建树复杂度可以认为是O(nlog(n))。平均情况下，建好的二叉排序树深度是log(n)
- 不能保证查询、插入、查找的log(n)复杂度。如果树退化成一根杆，复杂度就是O(n)



- 要保证log(n)复杂度，应该做到任意一个节点的左右子树节点数目基本相同，"平衡二叉树"可以做到这一点

平衡二叉树
(AVL树)

美国加州太浩湖

平衡二叉树(AVL树)

- 以二叉排序树为基础构造

- 为每个节点引入"平衡因子"(Balance Factor)属性，表示左子树高度和右子树高度的差

- AVL树确保任何节点的平衡因子都是1,0或-1。如果超出这个范围(失衡)，就会立即进行树的形状的调整，调整后依然保持这一特性

- 平衡因子的限制，确保任何节点的左右子树的节点数目差不多，从而实现log(n)的查询、插入、删除复杂度

AVL树添加节点

- 添加节点X后：

1）X必然是叶子节点且BF=0。从X的父节点开始，向上修改祖先节点的BF，直到某个祖先BF变为0，或树根的BF也被修改为止。若修改过程中未发现失衡节点（BF > 1 或 BF<-1），则修改完成后，添加节点完成。

2）结论3：如果修改祖先节点BF的过程中，发现某个节点V失衡，则立即调整以V为根的子树，调整完毕后，设新树根为Y，则Y.BF = 0，且所有Y的祖先的BF都不需要修改，添加节点完成。

显然，发现V失衡时，V的子孙节点都没有失衡，且V的祖先节点还没来得及看，也不需要看了。

AVL树添加节点

➢ 如何修改祖先BF

1）如果X是新增的叶子节点，则X的父亲的BF显然需要+1或-1

2） 结论1： 如果v的BF修改后不为0，则v的父亲的BF也需要修改。

若v是左子节点： v.father.BF +=1
若v是右子节点： v.father.BF -=1

因为：若v的BF修改后不为0，则以v为根的子树的高度一定增加了1，因此v.father.bf需要修改（分情况讨论证明）

3） 结论2： 如果v的BF被修改成0，则v的父亲及祖先的BF都不需要修改，因为以v为根的子树高度没有增加

AVL树添加节点

- 插入结点x，则调用 insertionUpdateBF(x)进行x祖先的BF修改

```
def insertionUpdateBF(nd):   #插入过程中修改nd的祖先的BF
    if nd.BF == 2 or nd.BF == -2:   # nd失衡
        insertionRebalance(nd) #insertionRebalance函数调整以nd为根的子树
        return                 #体现结论3：调整完毕后，修改祖先BF的过程就结束
    if nd.father:   #nd有父亲，故不是树根 ，则下面要体现结论1
        if nd.father.left is nd:   #nd是其父亲的左儿子
            nd.father.BF += 1
        else:                      #nd是右儿子
            nd.father.BF -= 1
        if nd.father.BF != 0:
            #体现结论2：若祖先的BF修改后变为0，则结束，不为0则继续递归修改
            insertionUpdateBF(nd.father)
```

AVL树添加节点

➢ v失衡后如何调整根为v的子树

要进些四种旋转操作之一：

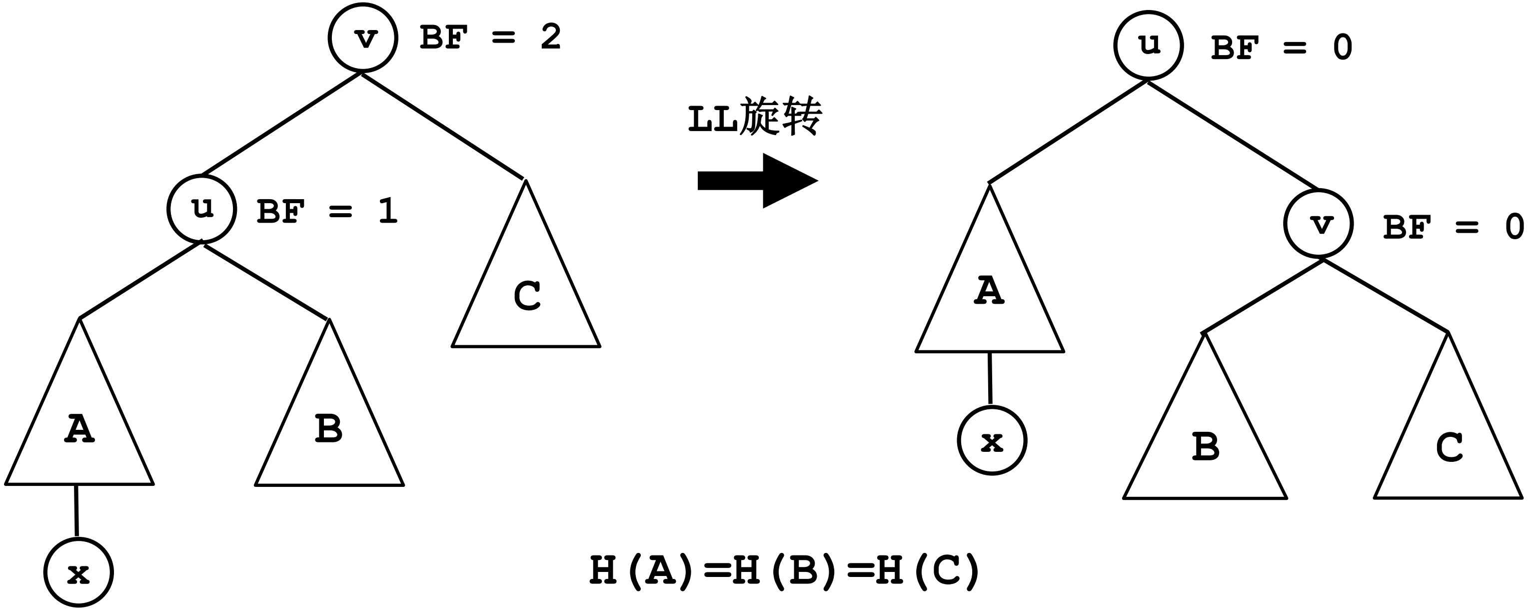
1） LL旋转： 新增的节点位于a的左子树的左子树
2） RR旋转： 新增的节点位于a的右子树的右子树
3） LR旋转： 新增的节点位于a的左子树的右子树
4） RL旋转： 新增的节点位于a的右子树的左子树

旋转完成后，该子树的根换成了别的节点Y，且Y.bf=0。而且该子树高度没有比添加节点前增加，因此祖先的BF都不用调整。

AVL树添加节点

- LL旋转rotateLL

适用场景：新增的节点x位于祖父v的左子树的左子树



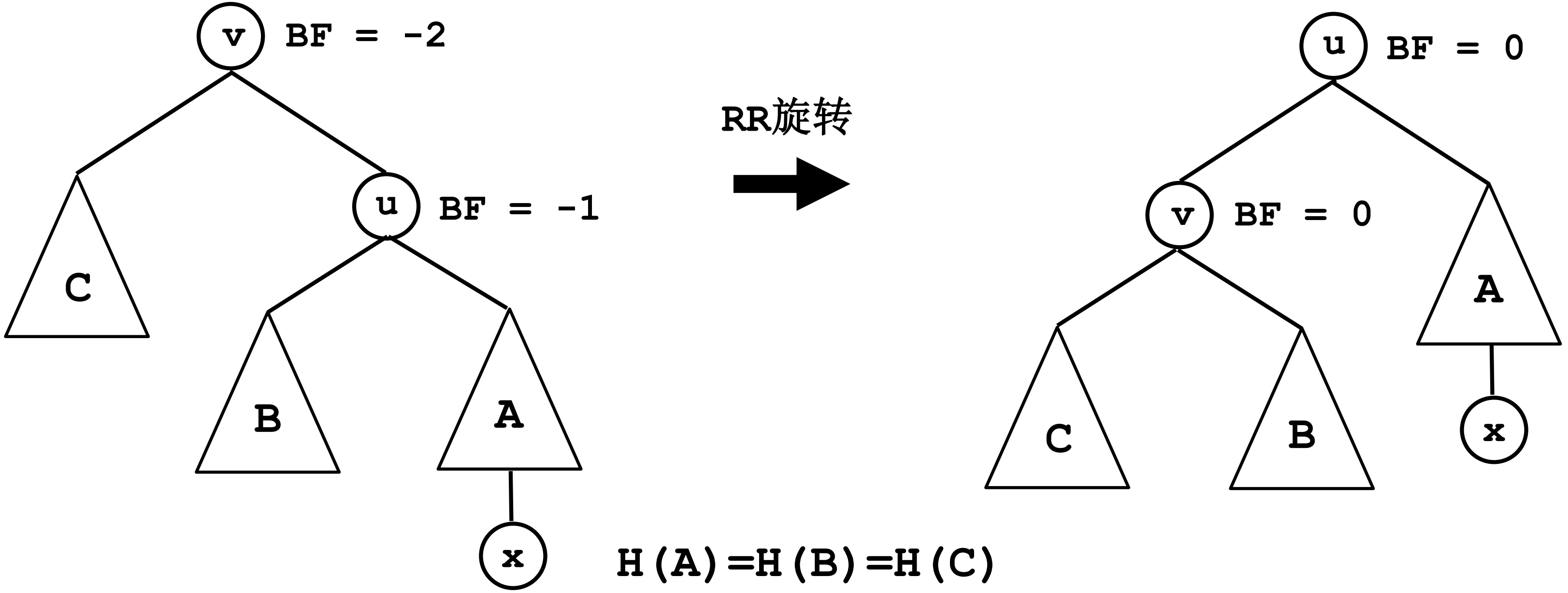
H(A)=H(B)=H(C)

v.BF==2且u.BF==1即可断定要LL旋转，此时必有 H(A) = H(B)=H(C)

AVL树添加节点

➢ RR旋转rotateRR

适用场景：新增的节点x位于祖父v的右子树的右子树

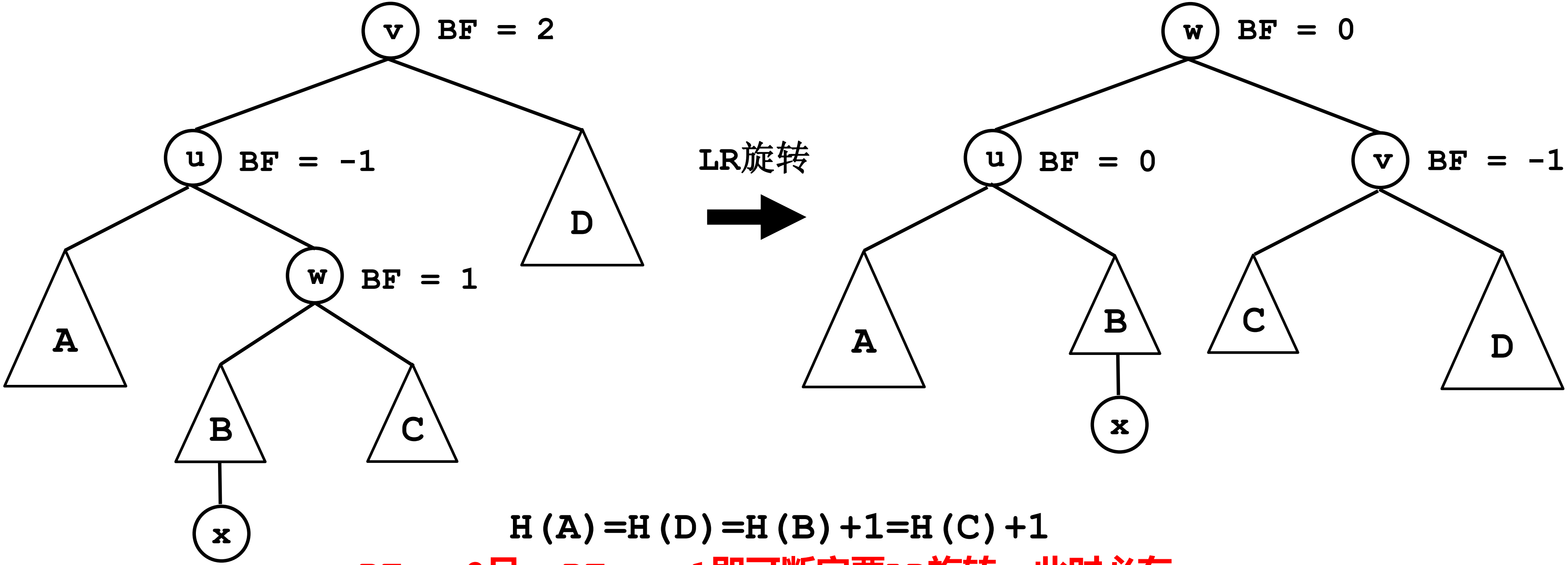


v.BF==-2且u.BF==-1即可断定要RR旋转，此时必有 H(A) = H(B)= H(C)

AVL树添加节点

➢ LR旋转rotateLR

适用场景：新增的节点x位于v的左子树的右子树（x在C下面类似）



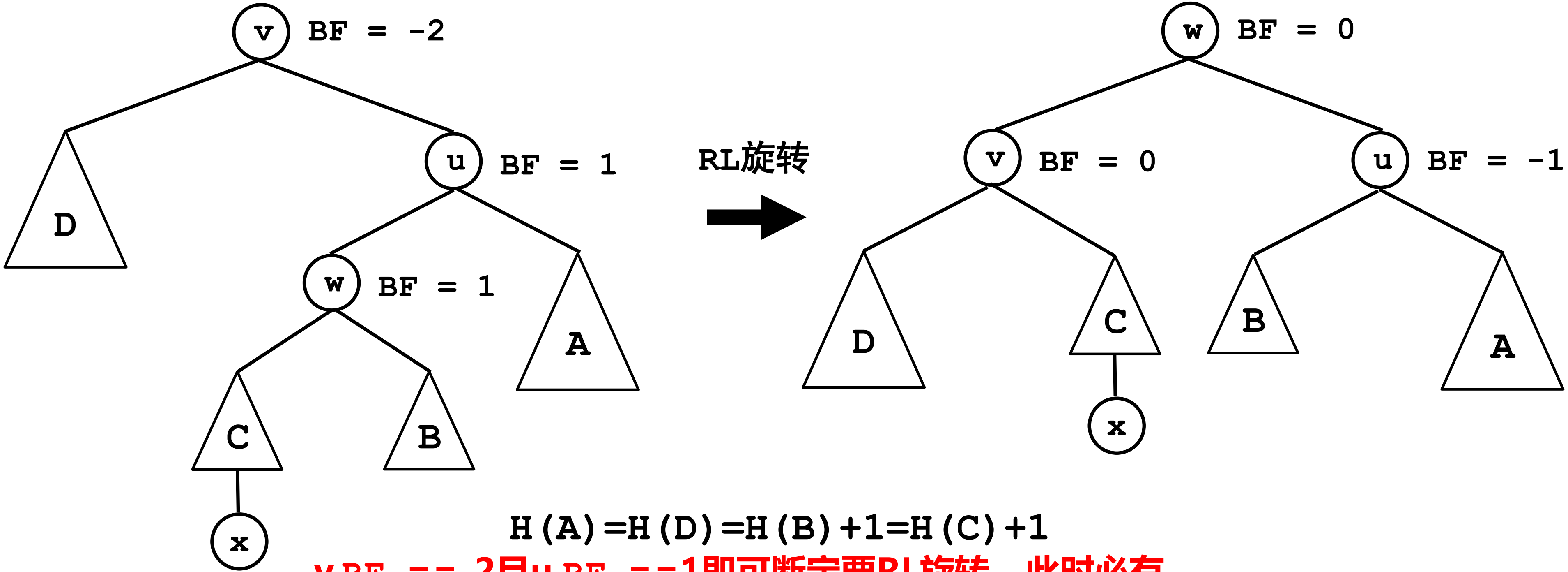
H(A)=H(D)=H(B)+1=H(C)+1

v.BF ==2且u.BF ==-1即可断定要LR旋转，此时必有 H(A) = H(D)=H(B)+1=H(C)+1。要注意B,C为空树的特例

AVL树添加节点

➢ RL旋转rotateRL

适用场景：新增的节点x位于v的右子树的左子树（x在B下面类似）



H(A)=H(D)=H(B)+1=H(C)+1

v.BF ==-2且u.BF ==1即可断定要RL旋转，此时必有
H(A) = H(D)=H(B)+1=H(C)+1。注意B,C为空树的特例

AVL树添加节点

➢ 调整子树

```
def insertionRebalance(self,nd): #nd失衡,nd.bf == 2或nd.bf == -2
    if nd.bf == 2: #新节点加在左子树
        if nd.left.bf == 1 : #LL旋转 #新节点加在左子树的左子树
            self.rotateLL(nd)
        elif nd.left.bf == -1: #新节点加在左子树的右子树
#nd.left.bf必然不可能为0，如果nd.left.bf ==0，则不会去更新nd.bf ,nd.bf就不可能变成2
            self.rotateLR(nd)
    if nd.bf == -2:
        if nd.right.bf ==  -1:
            self.rotateRR(nd)
        else:
            self.rotateRL(nd)
```

AVL树添加节点

➢ 复杂度分析

upgradeBalance 操作是沿着添加的叶子节点到树根的路径进行的，因此复杂度是O(log(n))

各类旋转操作复杂度O(1)，且添加一个节点时只会做一次

总复杂度O(log(n))

AVL树特点

- AVL树是比较复杂的数据结构，一般不会需要自己实现
- 堆和字典能替代其大部分使用场景
- AVL树能实现log(n)的插入、删除、查询，还能实现以下字典和堆无法实现的功能：

  log(n)查找小于某个值的最大元素
  log(n)查找大于某个值的最小元素
  不破坏结构的情况下，O(n)从小到大遍历元素

- Python不常用的第三方库 blist中的sorteddict有avl树的类似功能

红黑树简介

- AVL树查询效率很高,在增删的时候，由于对平衡性要求高，经常需要旋转，导致增删效率相对降低

- 红黑树降低一点平衡性（降低查询寻效率）,但减少增删时调整树结构的频率，以提高增删效率

红黑树简介

红黑树是满足以下几个条件的二叉查找树：

1） 结点要么是红色，要么是黑色。
2） 根结点是黑色。
3） 将结点中的空儿子指针都看作是一个“假点”，
且规定假点为黑色，则树根到每个假点的路径上经过的黑点数目都相同（黑色结点和假点都是黑点）。
4） 红色结点的子结点必是黑色，即任何一条从树根到假点的路径上不会出现连续两个红色结点。