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

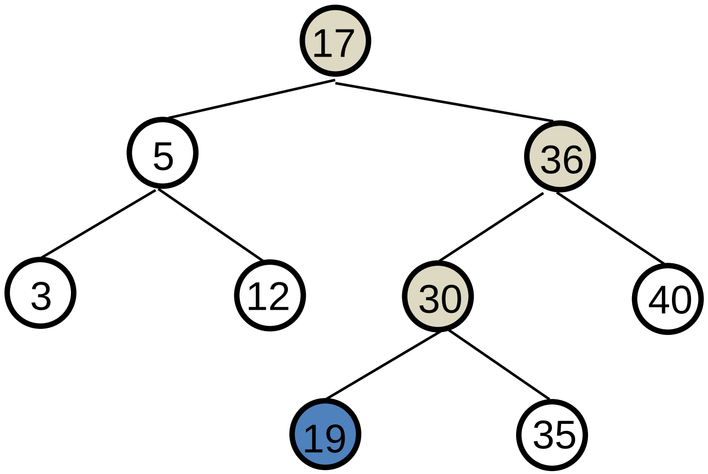
二叉排序树插入节点

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

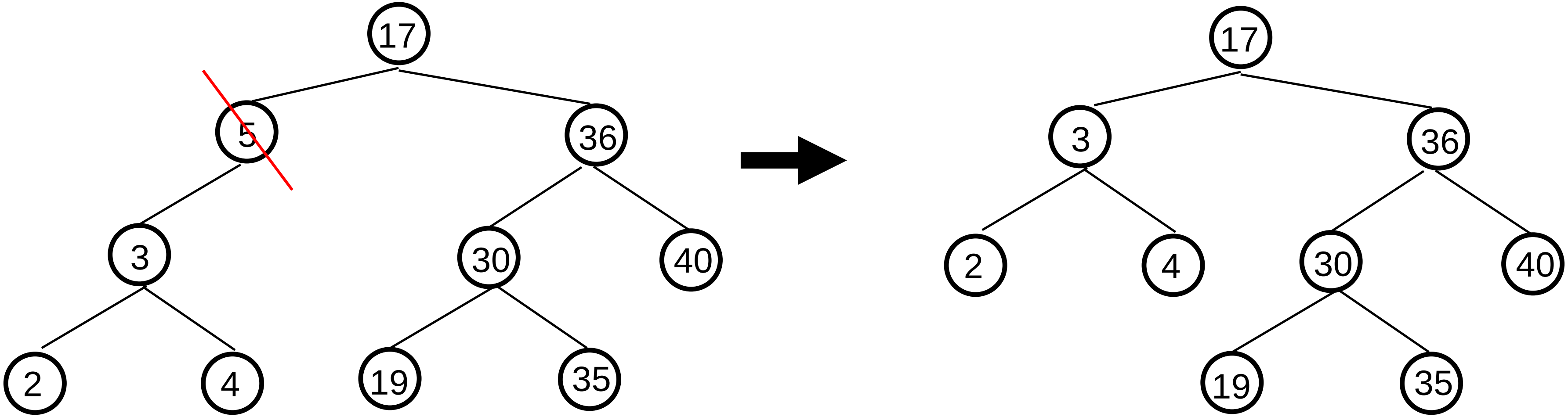
删除节点X，可以递归实现。分以下几种情况讨论：

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

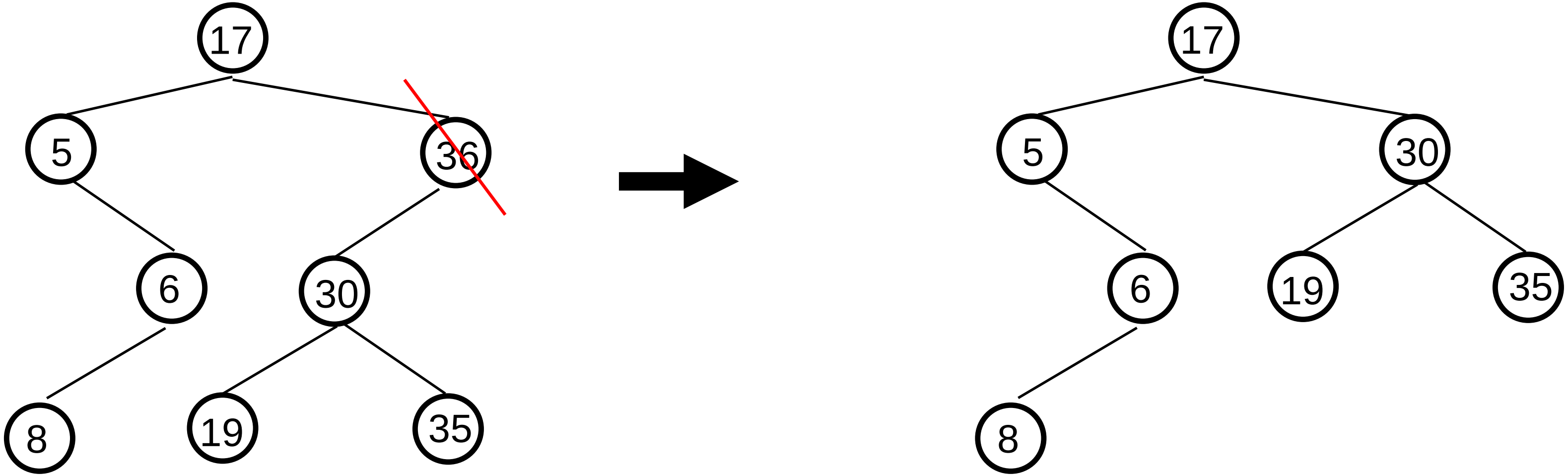
若X是父亲的左儿子，则X的左儿子作为X父亲的新左儿子



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

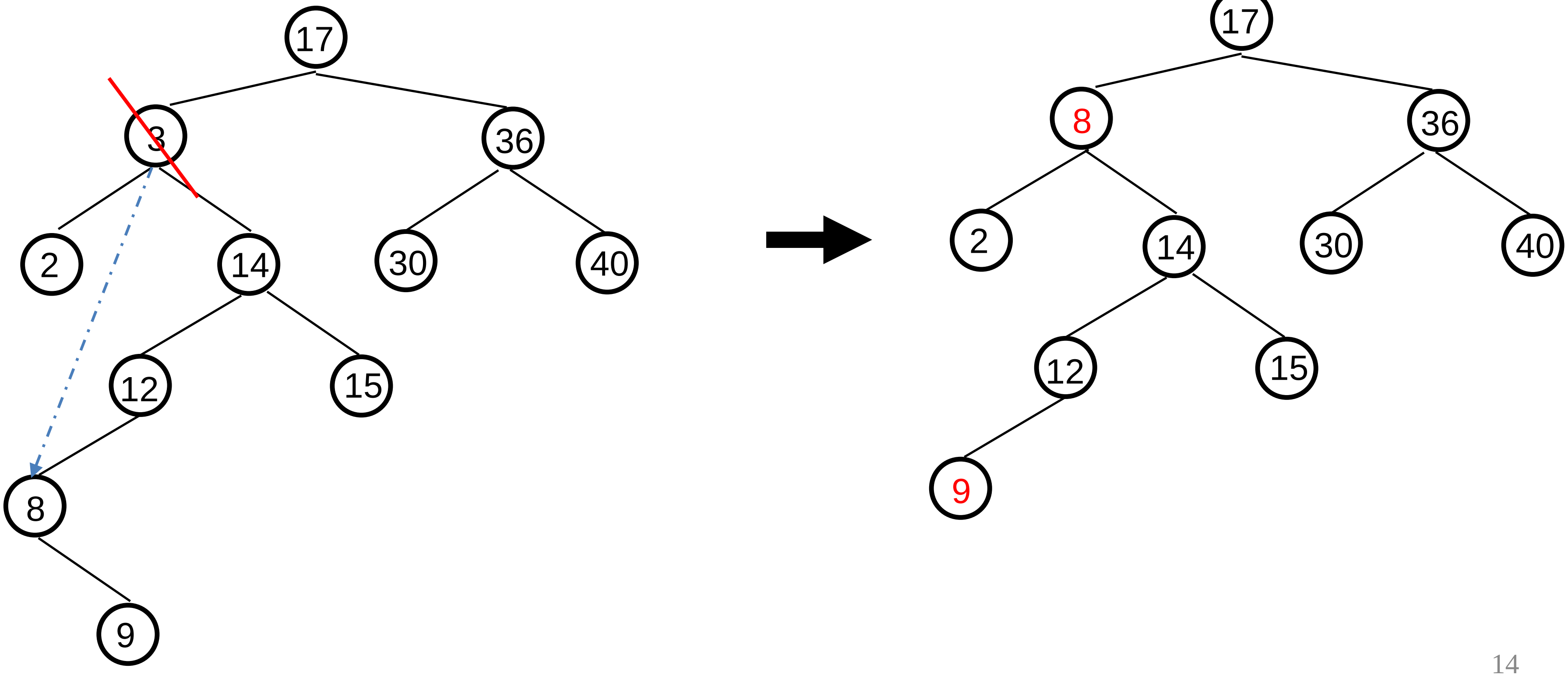
4）X既有左子节点，又有右子节点:

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 TreeNode:  #节点类
    #father在删除节点的时候有用,以及找一个节点的后继节点的时候有用
    #节点包含关键字和值, 排序树按关键字排序
    def __init__(self,key,val,father=None,left=None,right=None):
        self.key,self.val,self.left,self.right ,self.father \
            = key,val,left,right,father
    def isLeftChild(self):
        return self.father and self.father.left == self
    def isRightChild(self):
        return self.father and self.father.right == self
```

```
class Tree:
    def __init__(self,NodeType = TreeNode,less=lambda x,y:x<y ):
        self.root,self.size = None,0 #root是树根，size是节点总数
        self.less = less #less是比较函数，
        self.NodeType = NodeType #NodeType是节点类型
    def __find(self,key,root): #__开头的是私有方法，不宜也不易直接访问
        #查找关键字为key的节点
        if self.less(key,root.key):
            if root.left:
                return self.__find(key,root.left)
            else:
                return None
        elif self.less(root.key,key):
            if root.right:
                return self.__find(key,root.right)
            else:
                return None
        else:
            return root
```

```
def insert(self,key,val):  #插入节点
    if self.root == None:
        self.root = self.NodeType(key,val)
        self.size += 1
    else:
        if self.__insert(key,val,self.root):
            self.size += 1
```

```
def __insert(self,key,val,root):
    if self.less(key,root.key) :
        if root.left == None:
            root.left = self.NodeType(key, val,root) #root是father
            return True
        else:
            return self.__insert(key,val,root.left)
    elif self.less(root.key,key):
        if root.right == None:
            root.right = self.NodeType(key,val,root)
            return True
        else:
            return self.__insert(key,val,root.right)
    else:
        root.val = val  #相同关键字，则更新
        return False
```

```
def findMin(self):  #寻找最小节点
    nd = self.__findMin(self.root)
    return (nd.key,nd.val)

def __findMin(self,root):
    if root.left == None:
        return root
    else:
        return self.__findMin(root.left )
```

二叉排序树的实现

```
def pop(self,key):
    #删除键为key的元素，返回该元素的值。如果没有这样的元素，则引发异常
    nd = self.__find(key,self.root)
    if nd == None:
        raise Exception("key not found")
    else:
        self.size -= 1
        self.__deleteNode(nd)
```

二叉排序树的实现


```
def __deleteNode(self,nd):  #删除节点nd
    if nd.left and nd.right: #左右子树都有
        minNd = self.__findMin(nd.right)
        nd.key,nd.val = minNd.key,minNd.val
        self.__deleteNode(minNd)
    elif nd.left:  #只有左子树
        if nd.isLeftChild():
            nd.father.left = nd.left
            nd.left.father = nd.father
        elif nd.isRightChild():
            nd.father.right = nd.left
            nd.left.father = nd.father
        else:  #是树根
            self.root = nd.left
            self.root.father = None
```

二叉排序树的实现


```
elif nd.right : #只有右子树
    if nd.isRightChild():
        nd.father.right = nd.right
        nd.right.father = nd.father
    elif nd.isLeftChild():
        nd.father.left = nd.right
        nd.right.father = nd.father
    else:
        self.root = nd.right
        self.root.father = None
else: #nd是叶子
    if nd.isLeftChild():
        nd.father.left = None
    elif nd.isRightChild():
        nd.father.right = None
    else:
        self.root = None
```

二叉排序树的实现

```
def inorderTravelSeq(self): #中序遍历生成器
    if self.root == None:
        return
    stack = [[self.root,0]] #0表示self的左子树还没有遍历过
    while len(stack) > 0:
        node = stack[-1]
        if node[0] == None: #node[0]是树节点
            stack.pop()
            continue
        if node[1] == 0: #左子树还没有遍历过
            node[1] = 1
            stack.append([node[0].left,0])
        elif node[1] == 1: #左子树已经遍历过
            yield (node[0].key,node[0].val)
            node[1] = 2
            stack.append([node[0].right, 0])
        else: #右子树也遍历完了
            stack.pop()
```

二叉排序树的实现


```
def __contains__(self, key): #支持运算符 in
    return self.__find(key,self.root) != None

def __iter__(self): #返回迭代器
    return self.inorderTravelSeq()

def __getitem__(self,key): #支持只读运算符  [ ]
    nd = self.__find(key,self.root)
    if nd == None:
        raise Exception("key not found")
    else:
        return nd.val
def __setitem__(self, key, value): #支持写入运算符  [ ]
    nd = self.__find(key,self.root)
    if nd == None:
        self.insert(key,value)
    else:
        nd.val = value
```

二叉排序树的实现


```
def __str__(self):    #支持转换成字符串
    if self.root == None:
        return ""
    return self.__toString(self.root)
def __toString(self,root):
    result = ""
    if root.left:
        result += self.__toString(root.left)
    result += f"({root.key},{root.val})"
    if root.right:
        result += self.__toString(root.right)
    return result
def __len__(self):
    return self.size
```

二叉排序树的实现

```
#用法:
import random
random.seed(2)
s = [i for i in range(8)]
tree = Tree()   #或 tree = Tree(lambda x ,y : x > y) 则倒序
random.shuffle(s)
for x in s:
    tree.insert(x,x)
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
```

二叉排序树的实现

```
#用法:
try:
    print(tree[3000])
except Exception as e:
    print(e)            #>>key not found
tree[3000] = "ok"
print(tree[3000])       #>>ok
try:
    tree.pop(354)
except Exception as e:
    print(e)            #>>key not found
tree.pop(3)
print(tree)             #tree被自动转换成字符串，通过__str__支持
#>>(0,0)(1,1)(2,2)(4,4)(5,5)(6,6)(7,7)(3000,ok)
print()
for x in range(100,106):
    tree.insert(x,x)
```

二叉排序树的实现

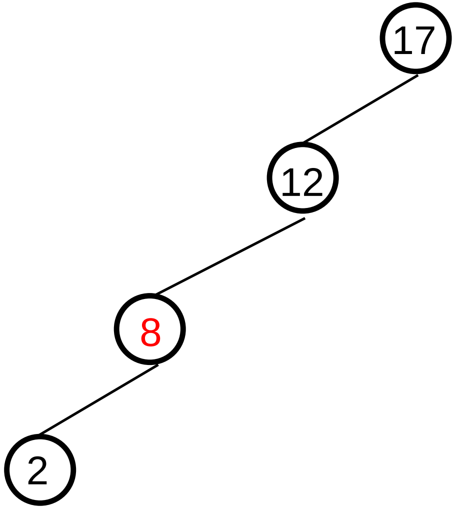

```
#用法:
while tree.size > 0:
    nd = tree.findMin()
    print(nd[0],end=",")
    tree.pop(nd[0])
```

- **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

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，二叉排序树深度是log(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

美国加州太浩湖

平衡二叉树

- 以二叉排序树为基础构造
- 确保每个节点的左右子树节点数目大致相同，从而实现log(n)的查询、插入、删除复杂度
- 添加或删除节点以后，如果导致某棵子树失衡（左右子树节点数目差超过一定范围）则需要进行树的形状的调整，调整后依然保持这一特性。
- AVL树，红黑树等都是平衡二叉树

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

3）如果X的BF修改后不为0，则X的父亲的BF也需要修改。

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

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

AVL树添加节点

➢ 如何修改祖先BF

```
#节点x插入后，立即调用upgradeBalance(x)来更新x的祖先的BF
    def upgradeBalance(self,nd): #更新nd的祖先的BF,nd是上页的x
        if nd.bf > 1 or nd.bf <-1: #nd失衡
            self.rebalance(nd) #调整以nd为根的子树
            return
        if nd.father:
            if nd.isLeftChild(): #能走到这里说明是左子树长高
                nd.father.bf += 1
            else:
                nd.father.bf -= 1
            if nd.father.bf != 0:
#nd.father.bf由非0变成0,则说明子树nd.father的高度没变化，那么不用更新祖先的bf了
                self.upgradeBalance(nd.father)
                #nd.father.bf不论是由0变成非0,
                #还是由1变2或者-1变-2，都说明子树nd.father高度加了1
```

AVL树添加节点

- a失衡后如何调整根为a的子树

要进些四种旋转操作之一：

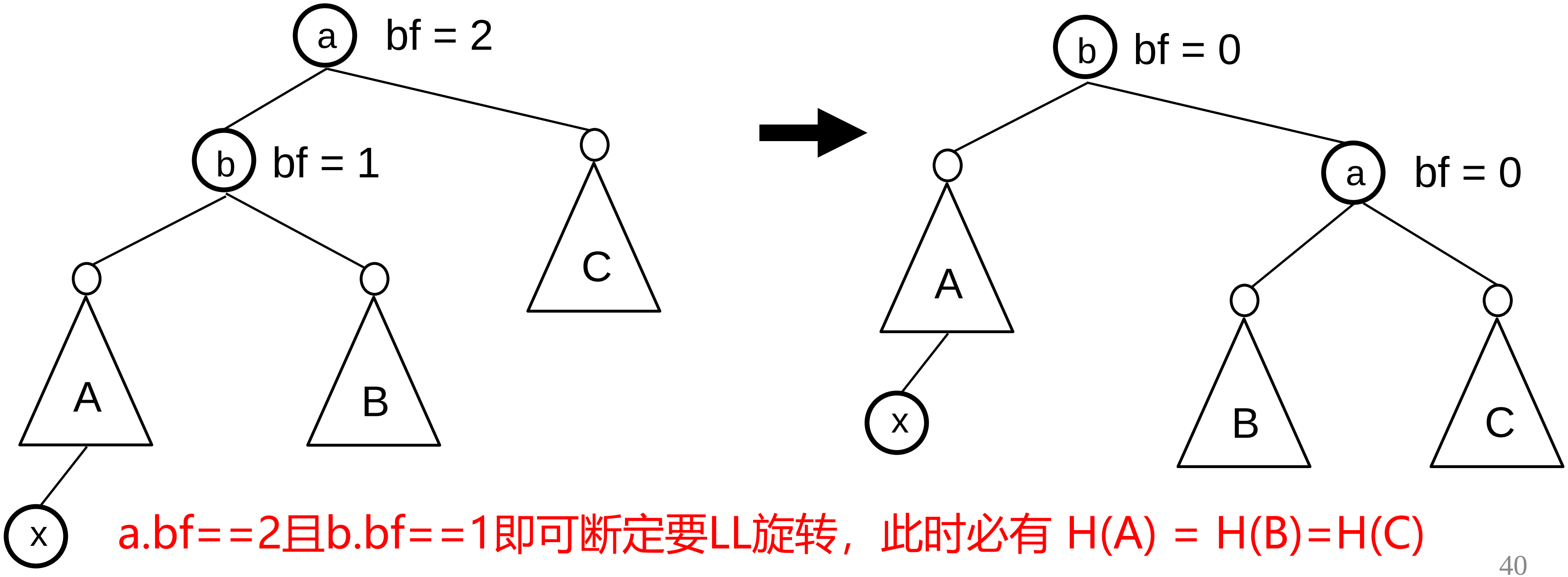
1） LL旋转： 新增的节点位于a的左子树的左子树
2） RR旋转： 新增的节点位于a的右子树的右子树
3） LR旋转： 新增的节点位于a的左子树的右子树
4） RL旋转： 新增的节点位于a的右子树的左子树

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

- ➢ 对根为a的子树进行LL旋转rotateLL

适用场景：新增的节点x位于a的左子树的左子树

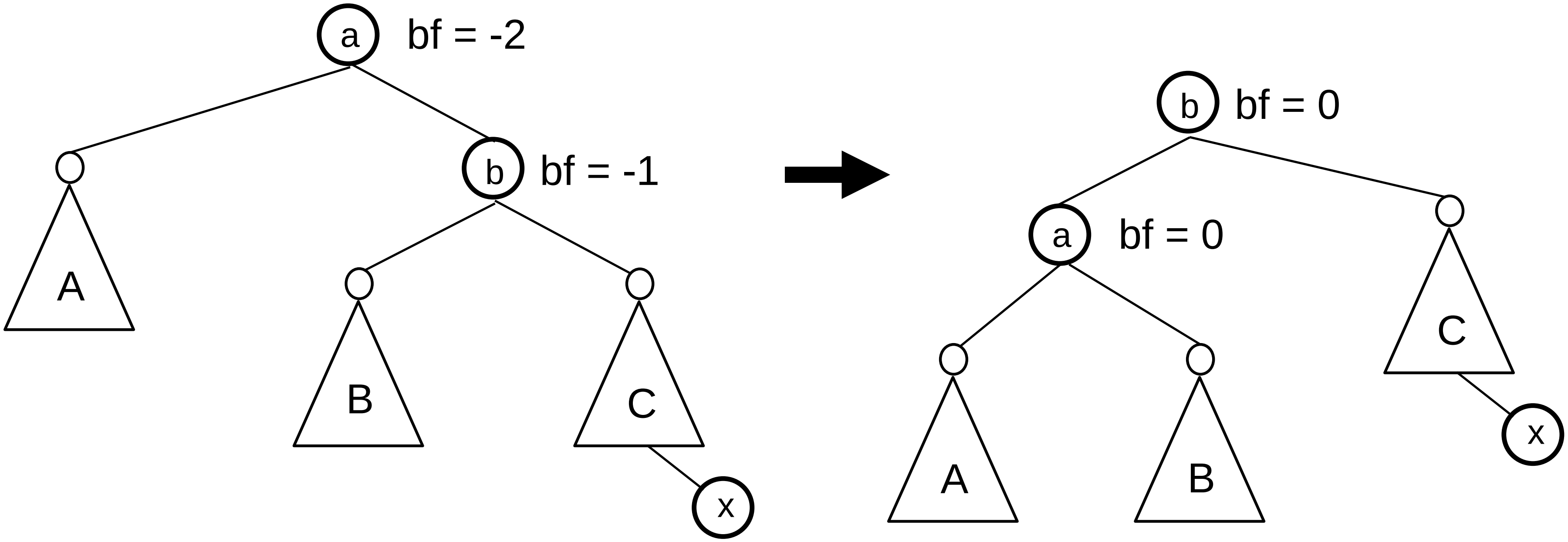


a.bf==2且b.bf==1即可断定要LL旋转，此时必有 H(A) = H(B)=H(C)

AVL树添加节点

➢ 对根为a的子树进行RR旋转rotateRR

适用场景：新增的节点位于a的右子树的右子树

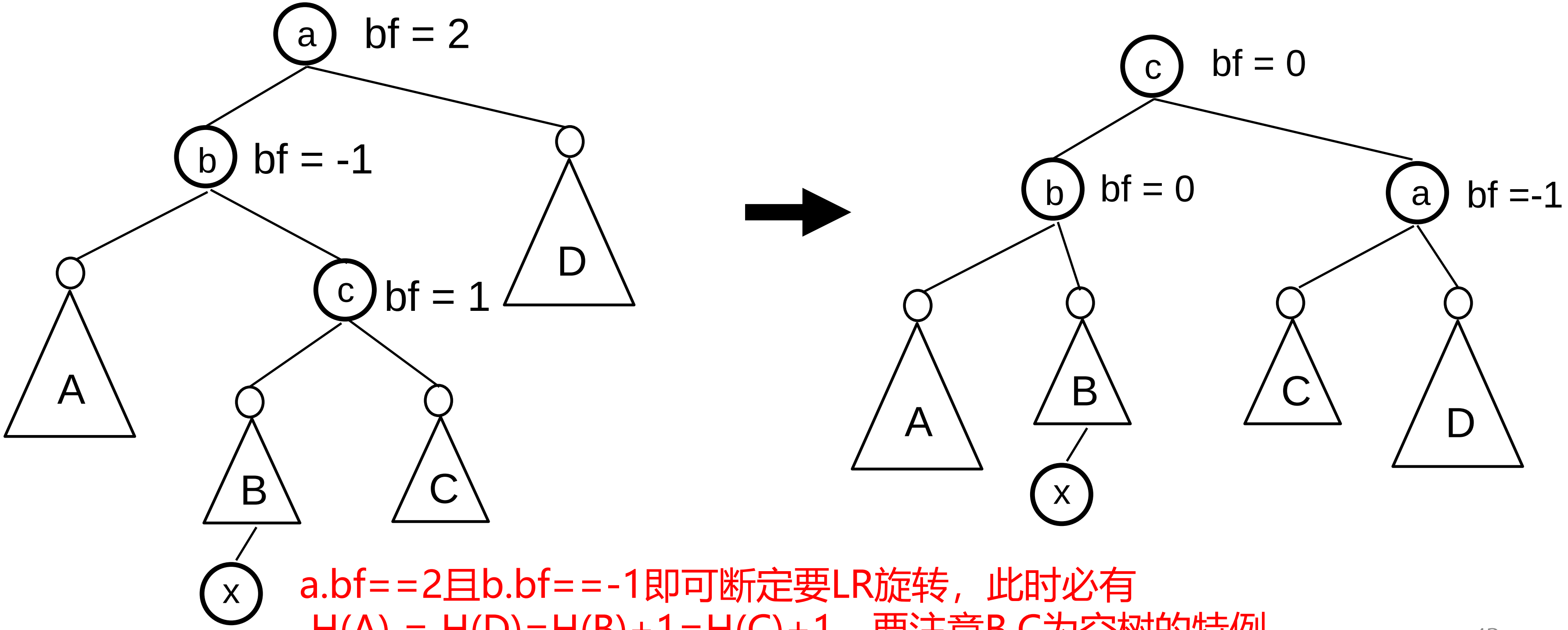


a.bf==-2且b.bf==-1即可断定要RR旋转，此时必有 H(A) = H(B)=H(C)

AVL树添加节点

➢ 对根为a的子树进行LR旋转rotateLR

适用场景：新增的节点x位于a的左子树的右子树（x在C下面类似）

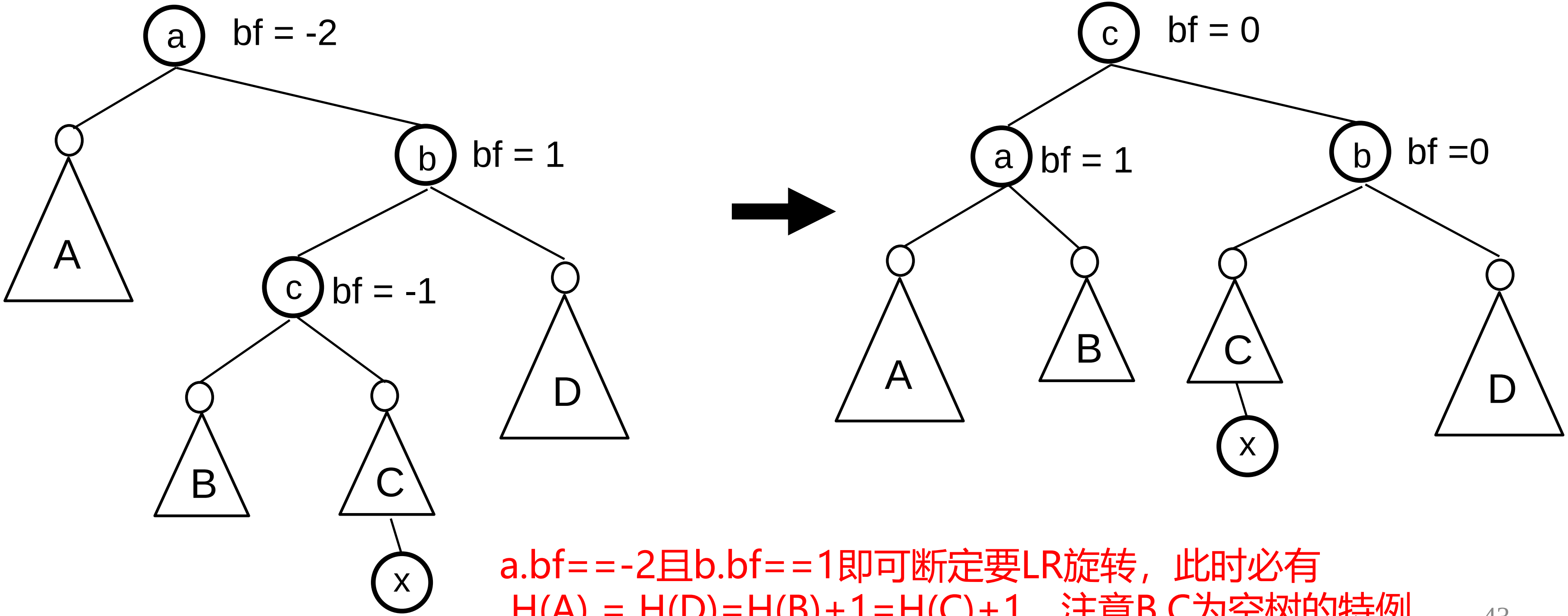


a.bf==2且b.bf==-1即可断定要LR旋转，此时必有
H(A) = H(D)=H(B)+1=H(C)+1。要注意B,C为空树的特例

AVL树添加节点

➢ 对根为a的子树进行RL旋转rotateRL

适用场景：新增的节点位于a的右子树的左子树（x在B下面类似）



a.bf==-2且b.bf==1即可断定要LR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 rebalance(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树删除节点

- 麻烦，略

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，红黑树等平衡二叉树能实现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