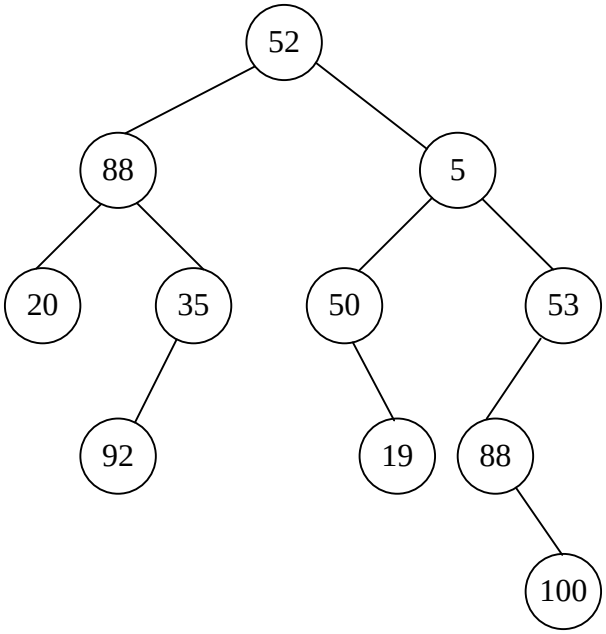
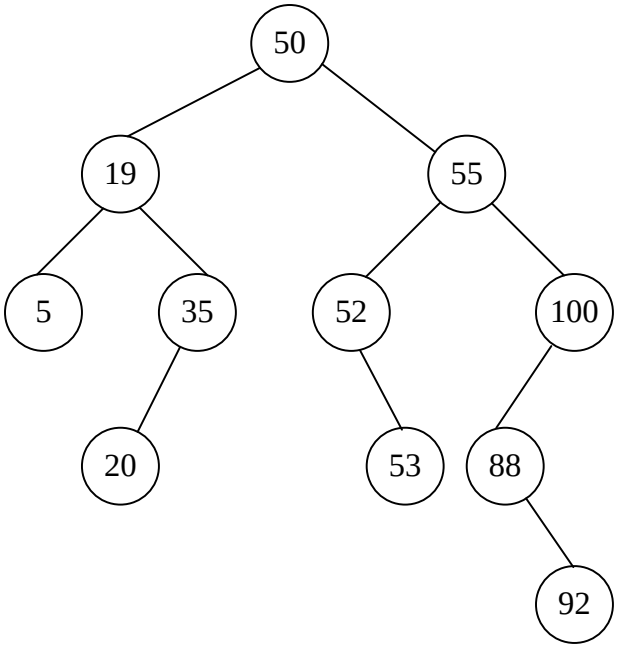
数据结构与算法B

08-二叉搜索树

二叉树

左右两棵二叉树的异同、特点？





二叉搜索树

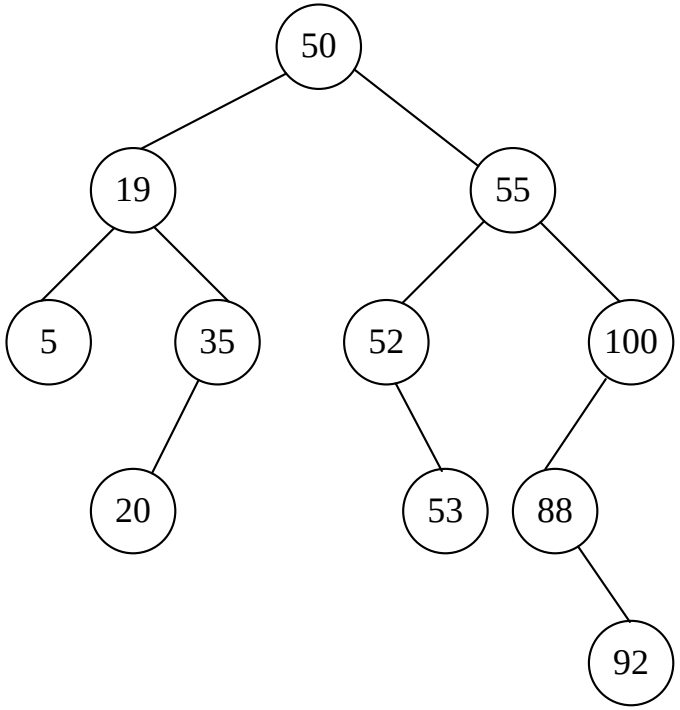
- 二叉树的一个主要用途是提供对数据（包括索引）的快速检索，而一般的二叉树对此并不具有性能优势
- 常用名称（同义词）
  - 二叉搜索树（Binary Search Tree,简称BST）
  - 二叉查找树
  - 二叉检索树
  - 二叉排序树

二叉搜索树：定义

- 假定存储在二叉树中的数据元素包含若干个域（field），其中一个称为码（key）K的域作为检索的依据，则二叉搜索树如下定义：
- 或者是一棵空树；
- 或者是具有下列性质的二叉树：
  - 对于任何一个结点，设其值为K
  - 则该结点的左子树(若不空)的任意一个结点的值都小于 K；
  - 该结点的右子树(若不空)的任意一个结点的值都大于 K；
  - 而且它的左右子树也分别为BST

二叉搜索树的性质

- 按照中序遍历一棵二叉搜索树得到的序列将是按照码值由小到大的排列



二叉搜索树上的操作

- 检索
- 插入（生成）
- 删除

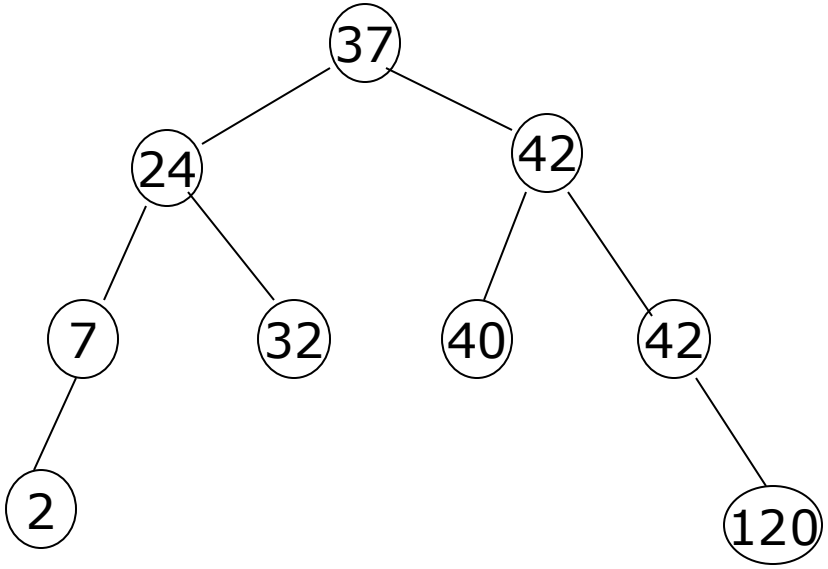
二叉搜索树上的检索

- 从根结点开始，在二叉搜索树中检索值K。
  - 如果根结点储存的值为K，则检索结束
  - 如果K小于根结点的值，则只需检索左子树
  - 如果K大于根结点的值，就只检索右子树

- 一直持续到K被找到或者遇上了一个树叶
  - 如果遇上树叶仍没有发现K，那么K就不在该二叉搜索树中

检索示例

- 查找码为37的结点
- 查找码为2的结点
- 查找码为40的结点
- 查找码为60的结点



比较次数？

时间复杂度？

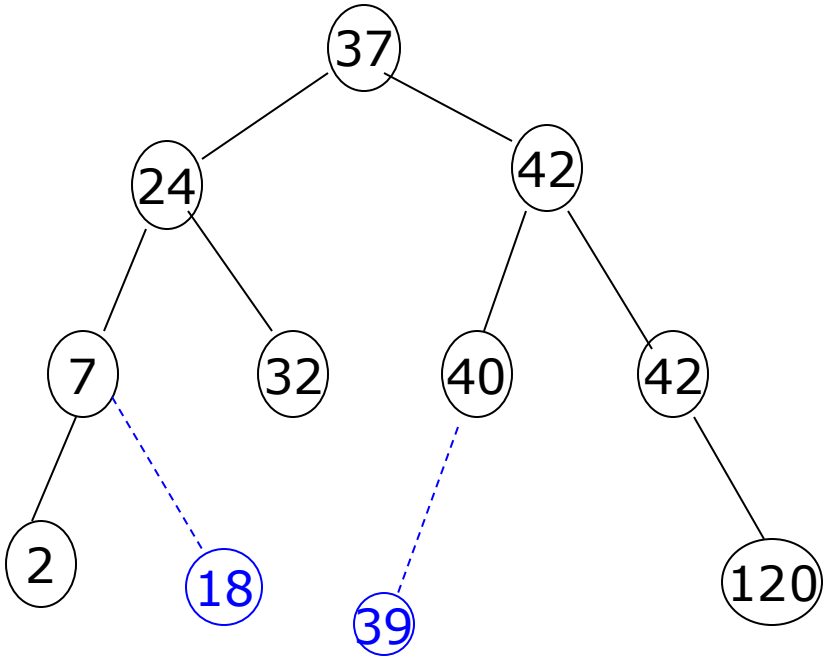
每一次只需与结点的一棵子树相比较

二叉搜索树上的插入

- 设待插入的结点码为 K:
  - 从根结点开始，若根结点为空，则将码K的结点作为根插入，操作结束
  - 如果K小于根结点的值，则将其按照二叉搜索树的要求插入左子树
  - 如果K大于根结点的值，就将其按照二叉搜索树的要求插入右子树
- 保证结点插入后仍符合二叉搜索树的定义

插入示例

- 插入39
- 插入18



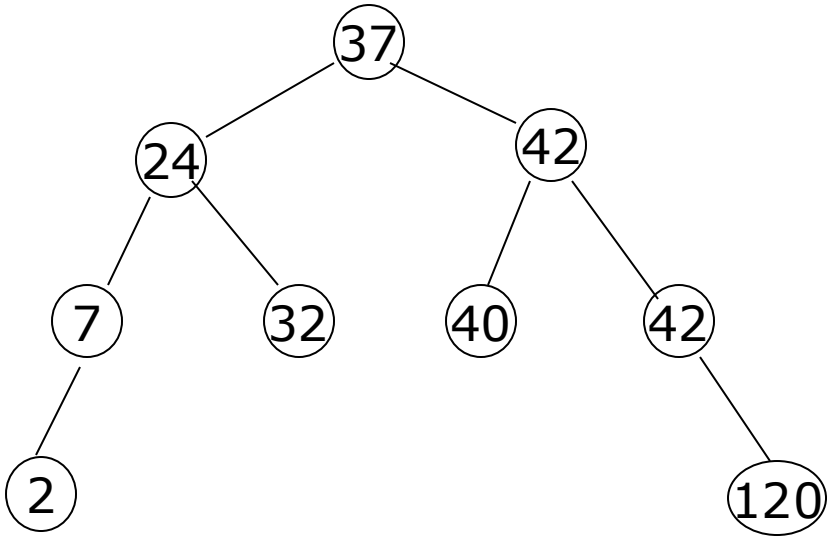
二叉搜索树插入

- 代价分析
  - 在执行插入操作时，也不必像在有序线性表中插入元素那样要移动大量的数据，而只需改动某个结点的空指针插入一个叶结点即可
  - 与查找结点的操作一样，插入一个新结点操作的时间复杂度是根到插入位置的路径长度，因此在树形比较平衡时二叉搜索树的效率相当高

二叉检索树上的删除

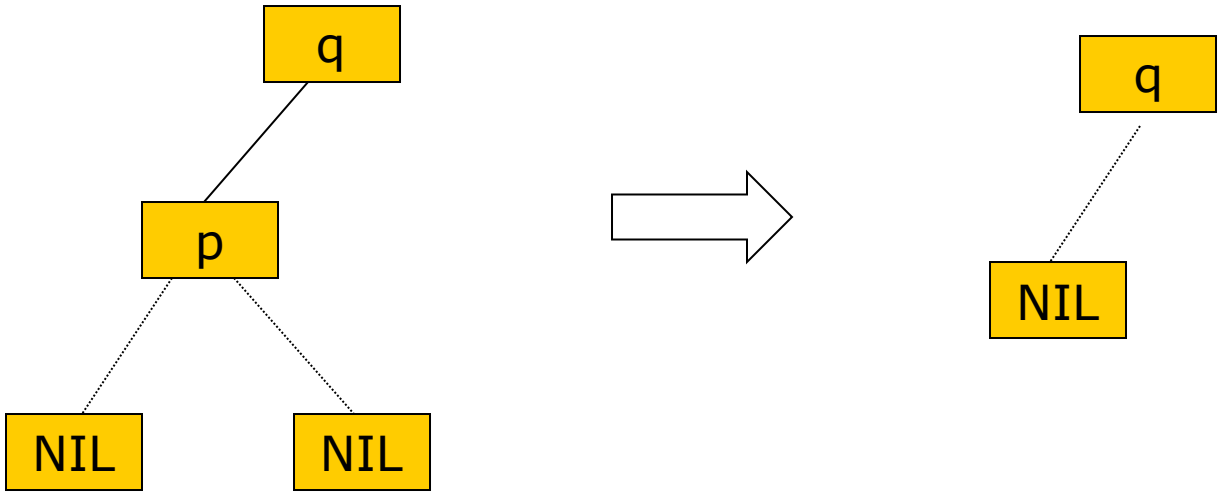
- 在二叉检索树删除一个结点，相当于删除有序序列中的一个记录，要求删除后仍能保持二叉检索树的排序特性，并且树高变化较小

删除2



删除示例1

情况1. 叶结点可以直接删除，其父结点的相应指针置为空

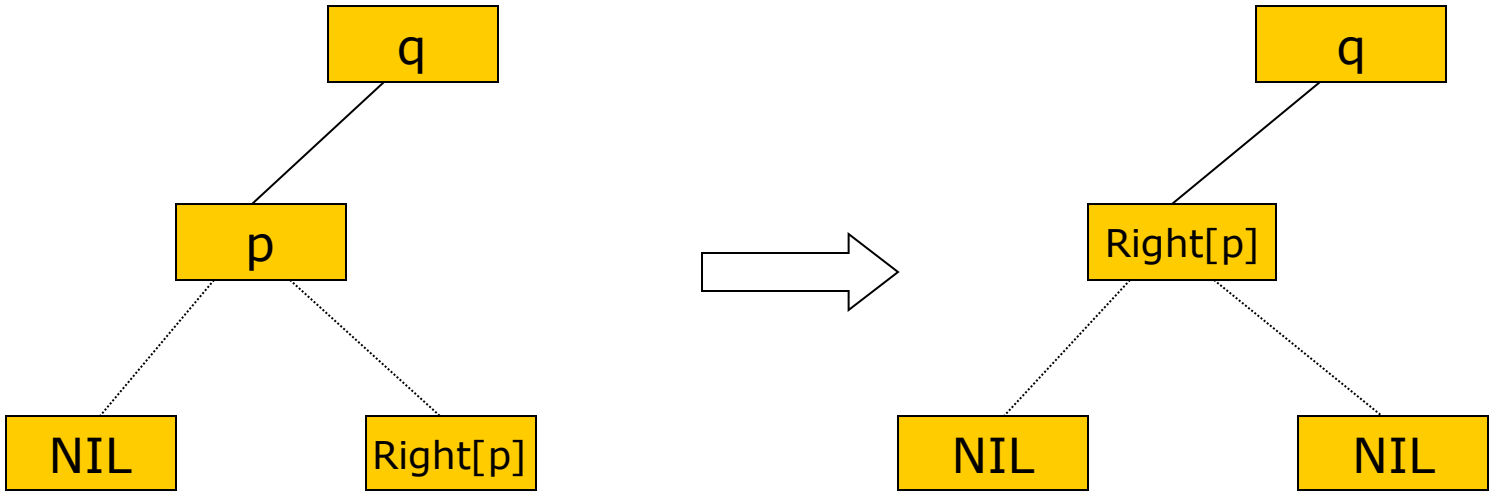


删除示例2

情况2. 只有一个子女的结点p被删除时，分以下情况：

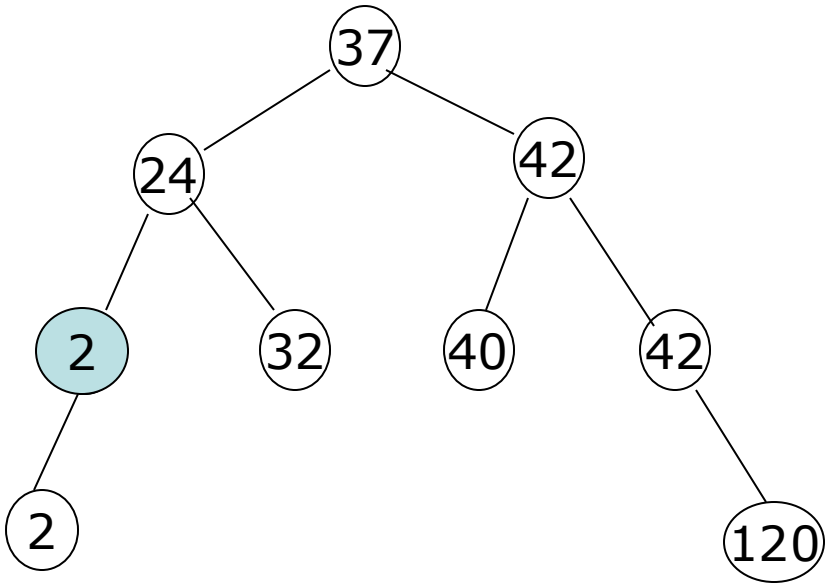
1. p是q的左子结点， p只有左子结点或右子结点
2. p是q的右子结点， p只有左子结点或右子结点

可让此子女直接代替即可



删除示例2

删除7



删除示例3

情况3. 被删除结点 p 的左右子女皆不空。

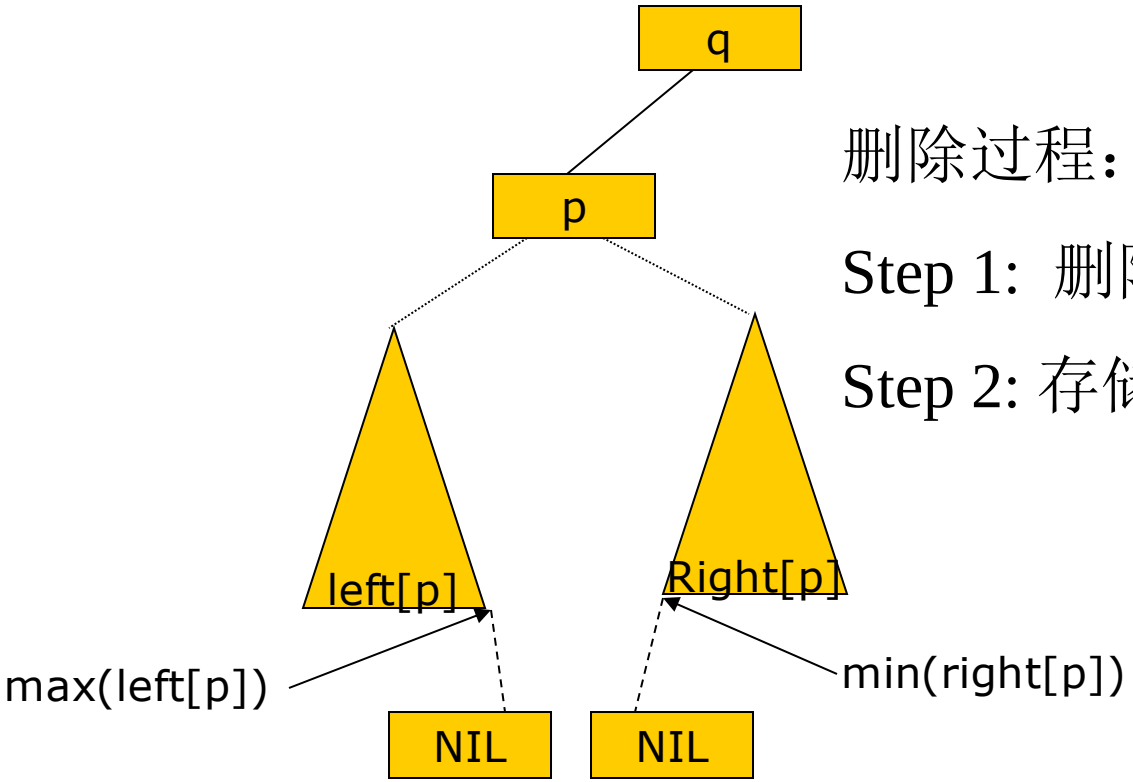
根据二叉检索树的性质，此时要寻找能代替此结点的结点必须是：比 p 左子树中的所有结点都大，比p的右子树的所有结点都小（或不大于）。

两个选择：

- 左子树中最大者
- **右子树中最小者**

而这二者都至多只有一个子结点

删除示例3



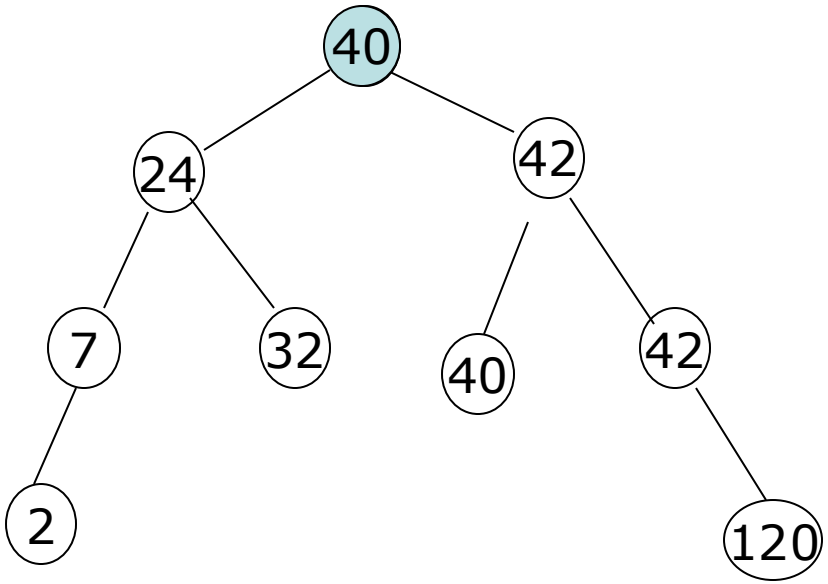
删除过程：

Step 1: 删除min(right[p])

Step 2: 存储min(right[p])到p中

删除示例3

- 删除37



二叉搜索树的删除

- 综合3种情况，二叉搜索树结点删除算法的基本思想：

① 若结点pointer没有右子树，则用pointer左子树的根代替被删除的结点pointer

② 若结点pointer有右子树，则在右子树里找到按中序遍历的第一个结点temppointer（即右子树中的最小结点）并将其从二叉搜索树里删除

③ 由于temppointer没有左子树，删除该结点只需用temppointer的右子树代替temppointer，然后用temppointer结点代替待删除的结点pointer

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
  - –root引用TreeNode对象
  - –size表示节点总个数
  - –__iter__（self)实现迭代器功能

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

    def __len__(self):
        return self.size

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

二叉搜索树的实现：TreeNode类

- class TreeNode
  - –key为键值
  - –value是数据项
  - –left/right_child
  - –另外加了个parent引用

```
class TreeNode:
    def __init__(self, key, value, left=None, right=None, parent=None):
        self.key = key
        self.value = value
        self.left_child = left
        self.right_child = right
        self.parent = parent

    def has_left_child(self):
        return self.left_child

    def has_right_child(self):
        return self.right_child

    def is_left_child(self):
        return self.parent and self.parent.left_child == self

    def is_right_child(self):
        return self.parent and self.parent.right_child == self
```

二叉搜索树的实现：TreeNode类

```
    def is_root(self):
        return not self.parent

    def is_leaf(self):
        return not (self.left_child or self.right_child)

    def has_any_children(self):
        return self.left_child or self.right_child

    def has_both_children(self):
        return self.left_child and self.right_child

    def replace_value(self, key, value, left, right):
        self.key = key
        self.value = value
        self.left_child = left
        self.right_child = right
        if self.has_left_child():
            self.left_child.parent = self
        if self.has_right_child():
            self.right_child.parent = self
```


二叉搜索树的实现：BST.put方法

- put(key, value)方法：插入key构造BST
  - 首先看BST是否为空，如果一个节点都没有，那么key成为根节点root
  - 否则，就调用一个递归函数_put(key, value, root)来放置key
- _put(key, value, current_node)的算法流程
  - 如果key比current_node.key小，那么_put到current_node左子树
    - 但如果没有左子树，那么key就成为左子节点
  - 如果key比current_node.key大，那么_put到current_node右子树
    - 但如果没有右子树，那么key就成为右子节点

```
def put(self, key, value):
    if self.root:
        self._put(key, value, self.root)
    else:
        self.root = TreeNode(key, value)
    self.size += 1
```

二叉搜索树的实现：_put辅助方法

- 注意！这个代码没有处理插入重复key的情况
  - 其实也就是把value替换一下的事

递归左子树

递归右子树

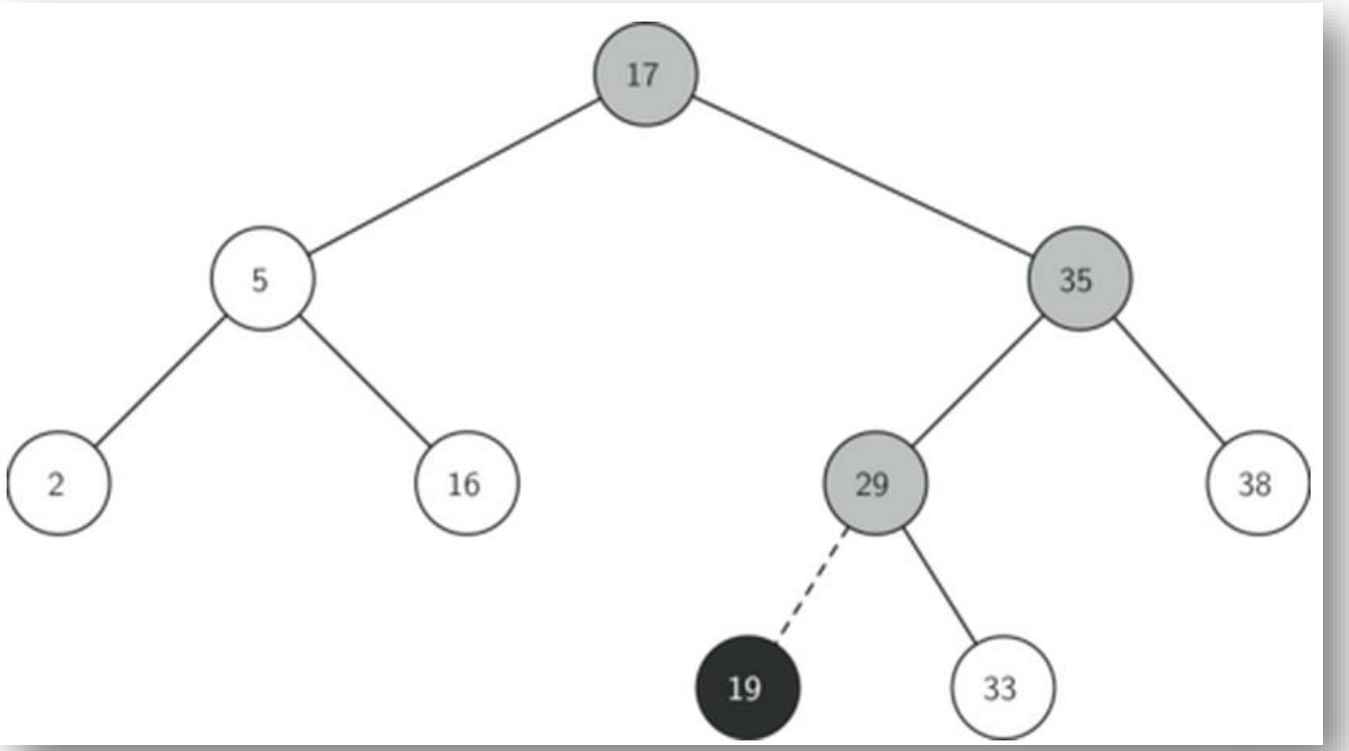
```
def _put(self, key, value, current_node):
    if key < current_node.key:
        if current_node.has_left_child():
            self._put(key, value, current_node.left_child)
        else:
            current_node.left_child = TreeNode(key, value, parent=current_node)
    else:
        if current_node.has_right_child():
            self._put(key, value, current_node.right_child)
        else:
            current_node.right_child = TreeNode(key, value, parent=current_node)
```

- 随手把__setitem__做了：
  - 可以myZipTree['PKU'] = 100871

```
def __setitem__(self, key, value):
    self.put(key, value)
```


二叉搜索树的实现： BST.put图示

- 插入key=19，current_node的变化过程（灰色）：



二叉搜索树的实现： BST.get方法

- 一旦BST构建起来，下一个方法就是从树中按照key来取value，即在树中找到key所在的节点
  - –从root开始，递归向下，直到找到，或者下到最底层的叶节点也未找到。
- get方法
  - –查看root是否存在？
  - –调用递归函数_get
- _get方法
  - –空引用返回None
  - –匹配当前节点，成功
  - –否则递归进入左/右子树

```
def get(self, key):
    if self.root:
        result = self._get(key, self.root)
        if result:
            return result.value
    return None

def _get(self, key, current_node):
    if not current_node:
        return None
    elif current_node.key == key:
        return current_node
    elif key < current_node.key:
        return self._get(key, current_node.left_child)
    else:
        return self._get(key, current_node.right_child)
```


二叉搜索树的实现：BST.get方法

- __getitem__特殊方法
  - 实现val= myZipTree['PKU']
- __contains__特殊方法
  - 实现'PKU' in myZipTree的集合判断运算符in

```
def __getitem__(self, key):
    return self.get(key)

def __contains__(self, key):
    return bool(self._get(key, self.root))
```

二叉搜索树的实现： BST.delete方法

- 有增就有减，最复杂的delete方法：
  - 首先，看树中有多少节点，超过1个的话，就先用_get找到要删除的节点，然后调用_delete来删除，找不到则提示错误；
  - 如果仅有1个节点（就是只有根节点了），那就看是否匹配根key，能匹配的话就删除根节点，匹配不了则报错。
- __delitem__特殊方法
  - 实现del myZipTree['PKU']这样的语句操作
- 复杂在_delete方法！

```
def __delitem__(self, key):
    self.delete(key)
```

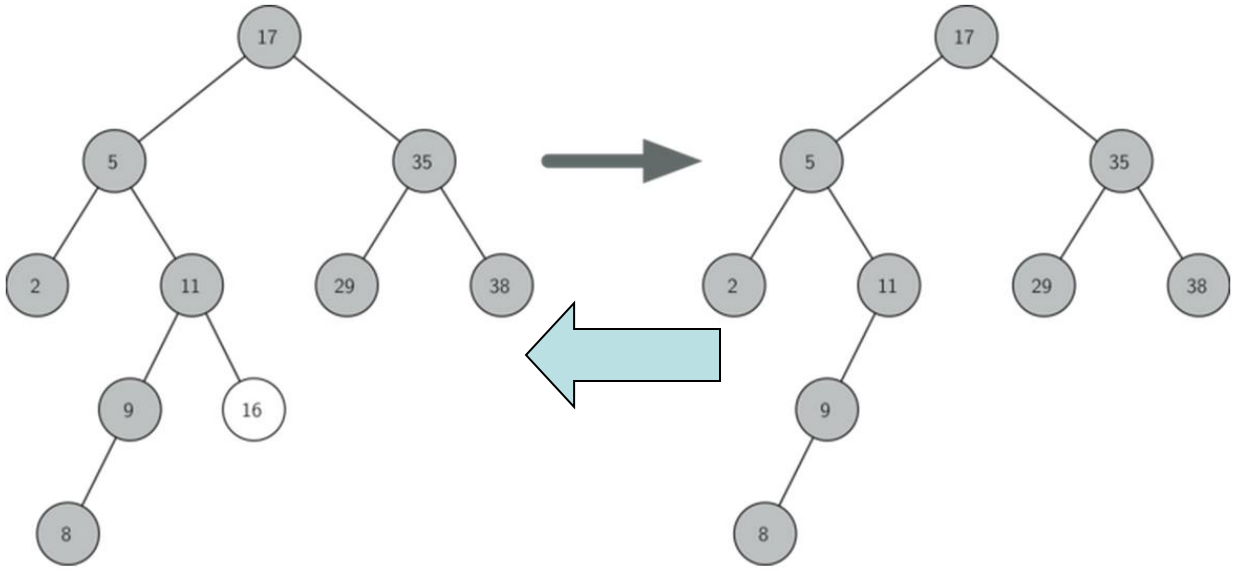
```
def delete(self, key):
    if self.size > 1:
        node_to_delete = self._get(key, self.root)
        if node_to_delete:
            self._delete(node_to_delete)
            self.size -= 1
        else:
            raise KeyError('Error, key not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size -= 1
    else:
        raise KeyError('Error, key not in tree')
```

二叉搜索树的实现： BST._delete方法

- 从BST中_delete一个节点，还仍然保持BST的性质，分以下3种情形
  - 节点没有子节点
  - 节点有1个子节点
  - 节点有2个子节点

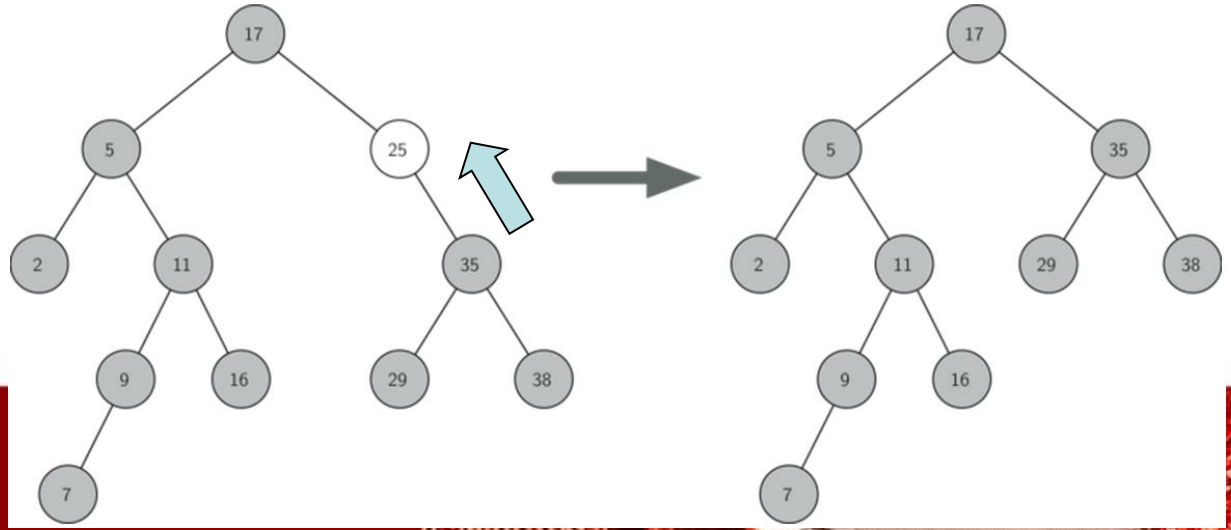
```
if current_node.is_leaf():
    if current_node.is_left_child():
        current_node.parent.left_child = None
    else:
        current_node.parent.right_child = None
```

- 没有子节点的情况好办，直接删除



二叉搜索树的实现： BST._delete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
- 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？



左子节点删除

右子节点删除

根节点删除

左子节点删除

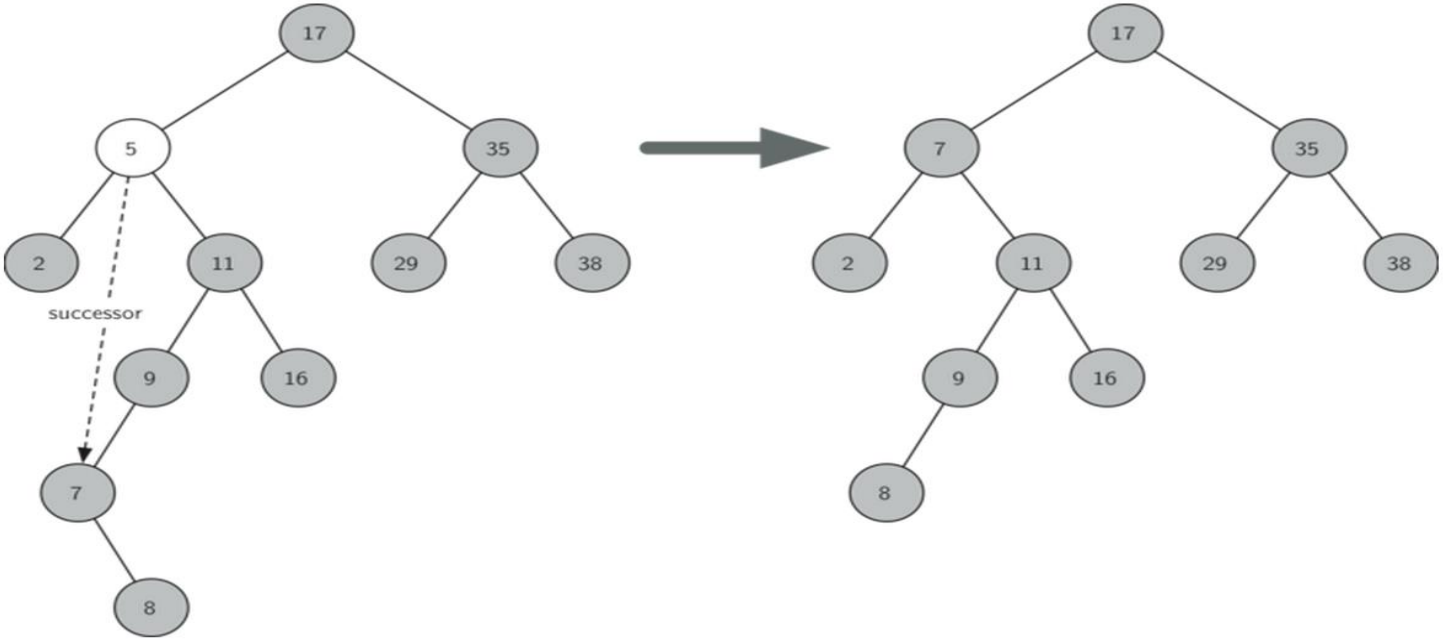
右子节点删除

根节点删除

```
else:
    if current_node.has_left_child():
        if current_node.is_left_child():
            current_node.left_child.parent = current_node.parent
            current_node.parent.left_child = current_node.left_child
        elif current_node.is_right_child():
            current_node.left_child.parent = current_node.parent
            current_node.parent.right_child = current_node.left_child
        else:
            current_node.replace_value(current_node.left_child.key,
                                       current_node.left_child.value,
                                       current_node.left_child.left_child,
                                       current_node.left_child.right_child)
    else:
        if current_node.is_left_child():
            current_node.right_child.parent = current_node.parent
            current_node.parent.left_child = current_node.right_child
        elif current_node.is_right_child():
            current_node.right_child.parent = current_node.parent
            current_node.parent.right_child = current_node.right_child
        else:
            current_node.replace_value(current_node.right_child.key,
                                       current_node.right_child.value,
                                       current_node.right_child.left_child,
                                       current_node.right_child.right_child)
```


二叉搜索树的实现： BST._delete方法

- 第3种情形最为复杂：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”
  - 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。



二叉搜索树的实现： BST._delete方法

- BinarySearchTree类：_delete方法（情形3）

```
elif current_node.has_both_children():
    successor = current_node.find_successor()
    successor.splice_out()
    current_node.key = successor.key
    current_node.value = successor.value
```

二叉搜索树的实现： BST._delete方法

- TreeNode类： 寻找后继节点find_successor()
  - –调用找到最小节点find_min()

```
def find_successor(self):
    successor = None
    if self.has_right_child():
        successor = self.right_child.find_min()
    else:
        if self.parent:
            if self.is_left_child():
                successor = self.parent
            else:
                self.parent.right_child = None
                successor = self.parent.find_successor()
                self.parent.right_child = self
    return successor
```

目前不会遇到

到左下角

```
def find_min(self):
    current = self
    while current.has_left_child():
        current = current.left_child
    return current
```

二叉搜索树的实现： BST._delete方法

- TreeNode类：摘出节点spliceOut()

摘出叶节点

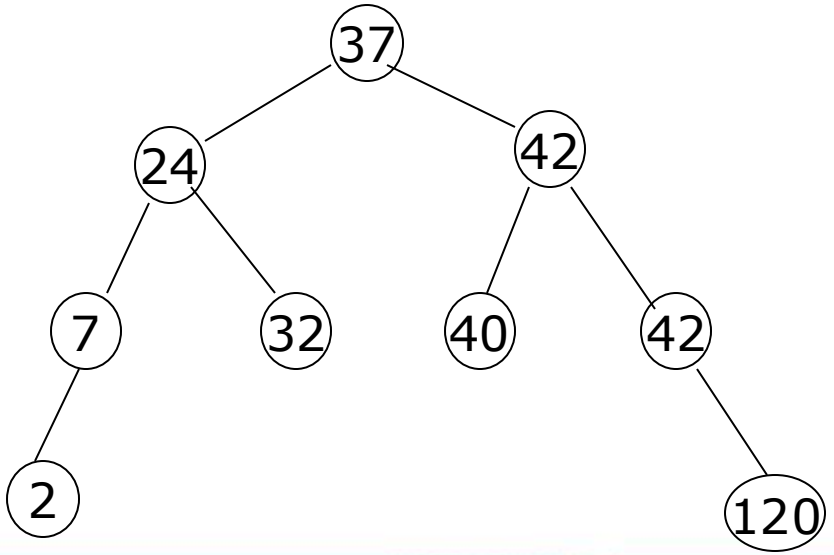
目前不会遇到

摘出带右子节点的节点

```
def splice_out(self):
    if self.is_leaf():
        if self.is_left_child():
            self.parent.left_child = None
        else:
            self.parent.right_child = None
    elif self.has_any_children():
        if self.has_left_child():
            if self.is_left_child():
                self.parent.left_child = self.left_child
            else:
                self.parent.right_child = self.left_child
            self.left_child.parent = self.parent
        else:
            if self.is_left_child():
                self.parent.left_child = self.right_child
            else:
                self.parent.right_child = self.right_child
            self.right_child.parent = self.parent
```

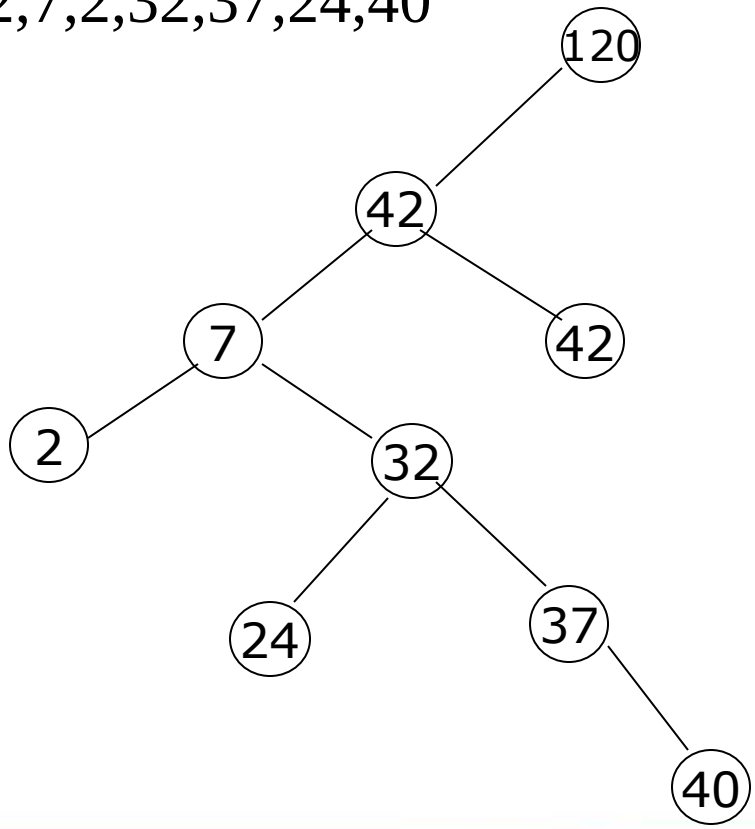
二叉搜索树的生成

- 对于给定的关键码集合，可以从一个空的二叉搜索树开始，按照检索路径搜索这棵二叉树，将关键码一个个插入到相应的叶结点位置，从而动态生成二叉搜索树
- 插入顺序：37,24,42,7,2,40,42,32,120



二叉搜索树生成：示例

插入顺序：120,42,42,7,2,32,37,24,40



二叉检索树的效率衡量

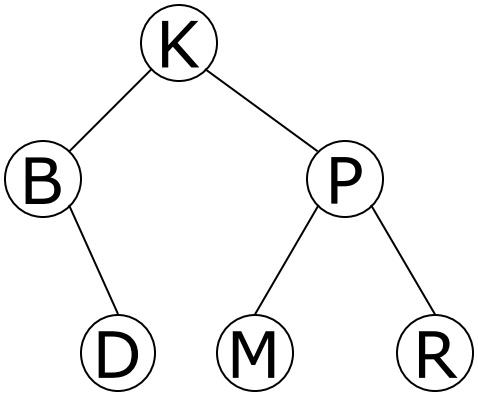
- 检索、插入、删除等操作的效率均依赖于二叉检索树的高度h，时间代价为O(h)
  - –最佳：高度（尽可能）最小
  - –最差：退化成线性结构

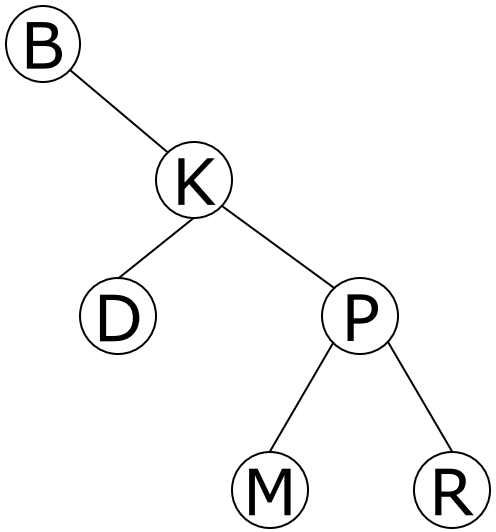
何为一棵最佳二叉检索树？

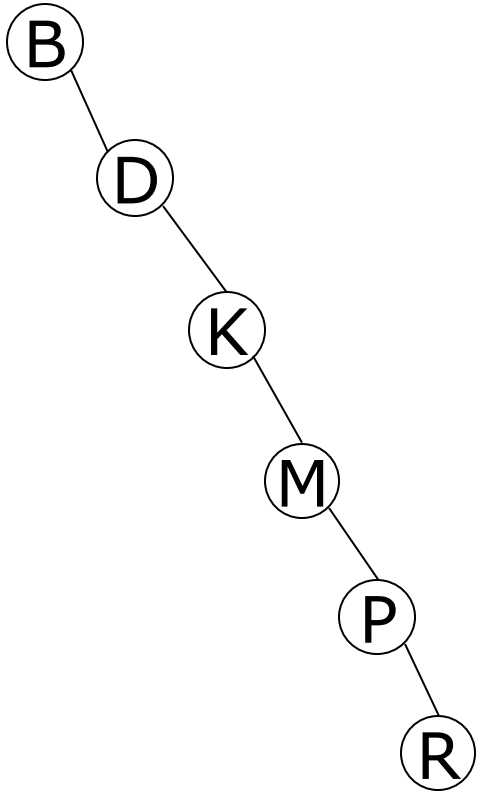
如何构筑一棵最佳二叉检索树？

如何保持二叉搜索树的最佳特性？

二叉搜索树



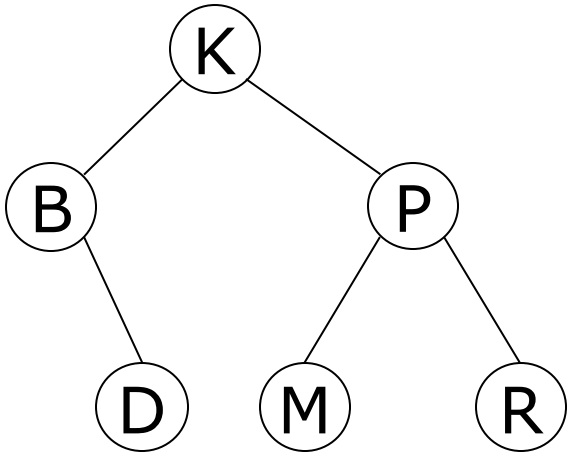




检索的效率 树的形状、特点

最佳二叉搜索树

- 检索概率相同时，先对序列进行排序：B D K M P R，然后用二分法依次插入这些关键码



如何动态地保持最佳？

- 检索概率相同时，根据关键码集合，可以构造出最佳二叉检索树
- 问题：静态，经过若干次插入、删除后可能会失去平衡，检索性能变坏
- 如何动态保持一棵二叉检索树的平衡，从而有较高的检索效率

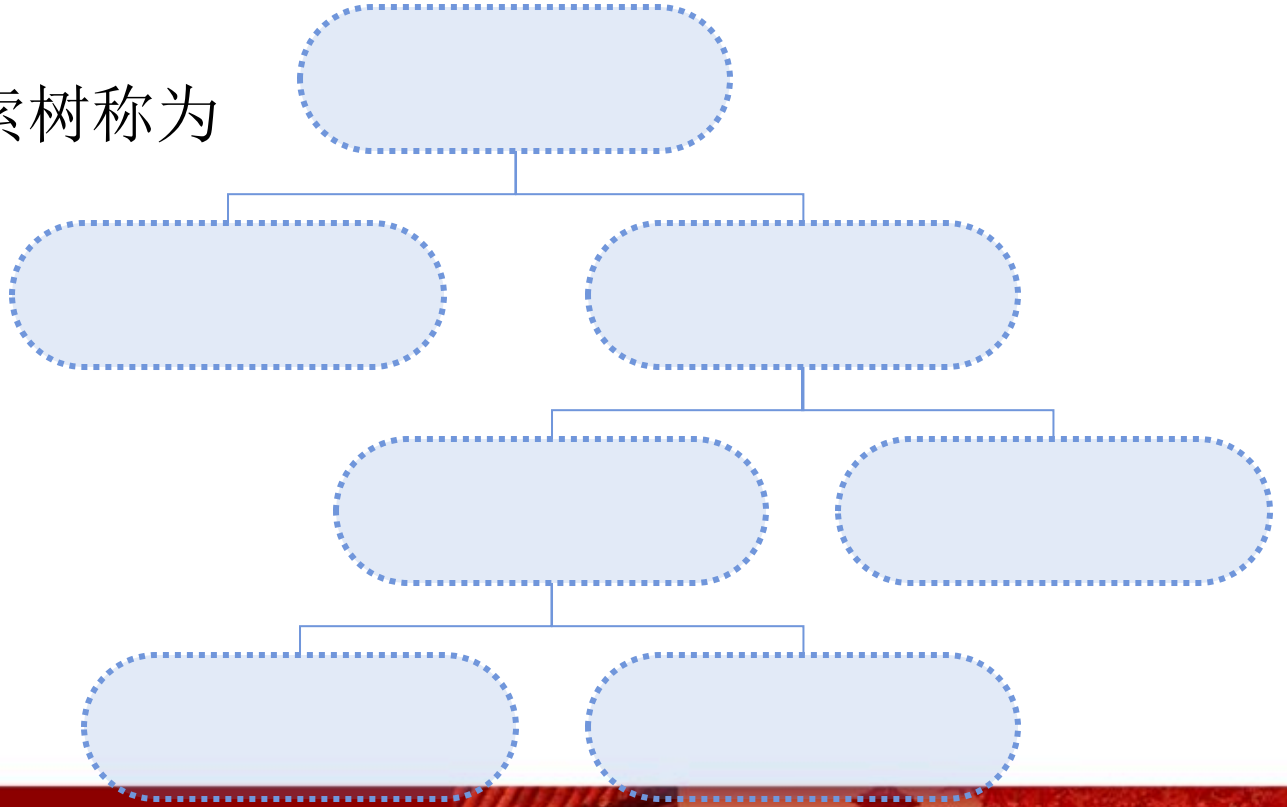
平衡树技术

平衡二叉搜索树：AVL树的定义

- 我们来看看能够在key插入时一直保持平衡的二叉搜索树：AVL树
  - AVL是发明者的名字缩写：G.M. Adelson-Velskii and E.M. Landis
- 利用AVL树实现ADT Map，基本上与BST的实现相同，不同之处仅在于二叉树的生成与维护过程
- AVL树的实现中，需要对每个节点跟踪“平衡因子balance factor”参数，平衡因子是根据节点的左右子树的高度来定义的，确切地说，是左右子树高度差：
  - *balanceFactor = height(left_subtree) − height(right_subtree)*
  - 如果平衡因子大于0，称为“左重left-heavy”，小于零称为“右重right-heavy”
  - 平衡因子等于0，则称作平衡。

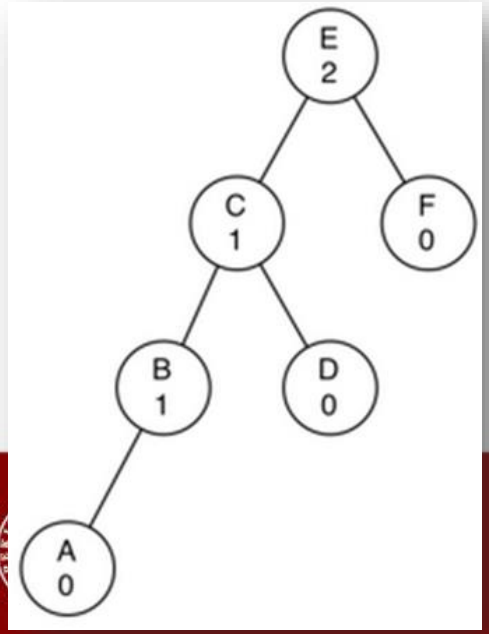
平衡二叉搜索树：平衡因子

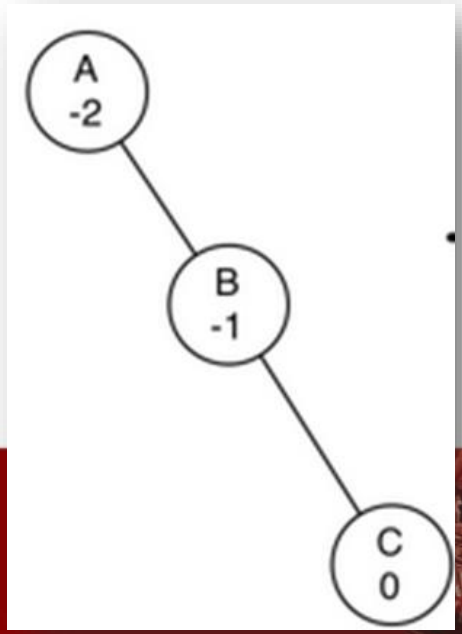
- 如果一个二叉搜索树中每个节点的平衡因子都在-1，0，1之间
- 则把这个二叉搜索树称为平衡树

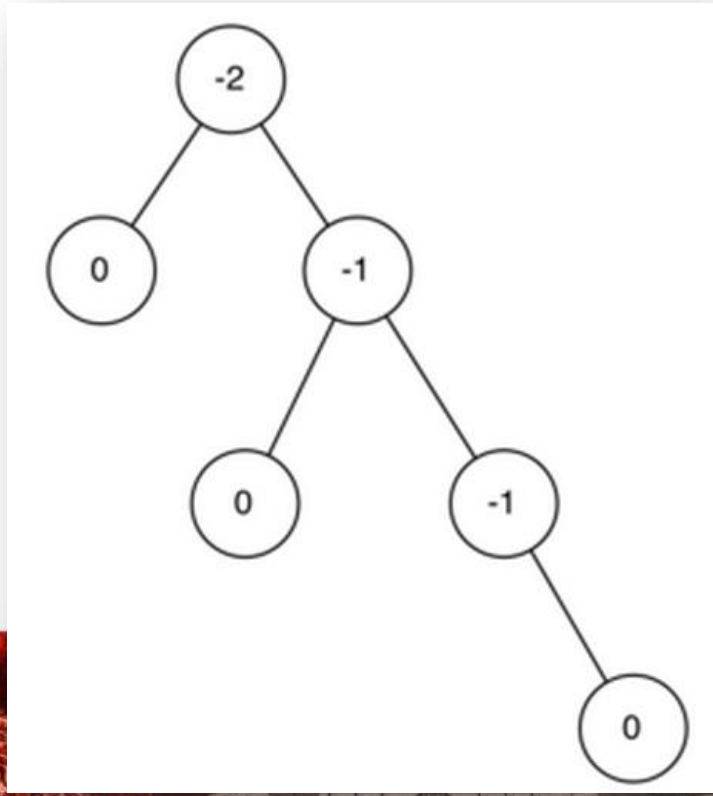


平衡二叉搜索树：AVL树的定义

- 一旦在平衡树操作过程中，有节点的平衡因子超出此范围，则需要一个重新平衡的过程
  - 要保持BST的性质！左子树所有节点都小于根，右子树都大于根
- 如右图是一个“右重”的不平衡树
- 思考：如果重新平衡，应该变成什么样？

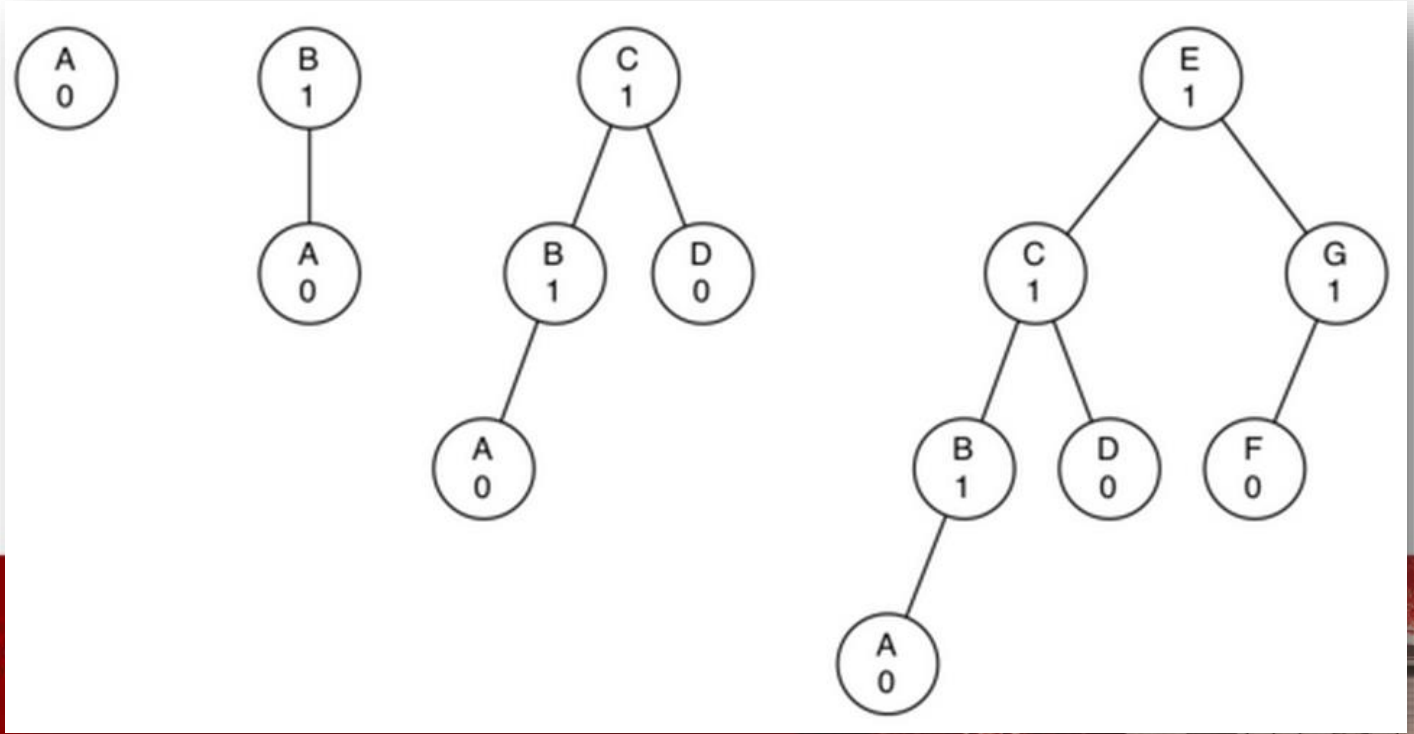






AVL树的性能

- 实现AVL树之前，来看看AVL树是否确实能够达到一个很好的性能
- 我们来分析AVL树最差情形下的性能：即平衡因子为1或者-1
  - 下图列出平衡因子为1的“左重”AVL树，树的高度从1开始，来看看问题规模（总节点数N）和比对次数（树的高度h）之间的关系如何？



AVL树性能分析

- 观察上图h=1、2、3、4时，总节点数N的变化：
  - h= 1, N= 1
  - h= 2, N= 2= 1+ 1
  - h= 3, N= 4= 1+ 1+ 2
  - h= 4, N= 7= 1+ 2+ 4

$$N_h = 1 + N_{h-1} + N_{h-2}$$

- 观察这个通式，很接近斐波那契数列！
  - 定义斐波那契数列F_i
  - 利用F_i重写N_h

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \text{ for all } i \geq 2$$

$$N_h = F_{h+2} - 1, h \geq 1$$

- 斐波那契数列的性质：F_i/F_{i-1}趋向于黄金分割Φ
  - 可以写出F_i的通式

$$\Phi = \frac{1+\sqrt{5}}{2}$$

$$F_i = \Phi^i / \sqrt{5}$$

AVL树性能分析

- 将F_i通式代入到N_h中，得到N_h的通式
- 上述通式只有N和h了，我们解出h
- 可见，在最差情况下，AVL树在总节点数为N的搜索中，最多需要1.44倍N对数级别的搜索次数
- 可以说AVL树的搜索时间复杂度为O(log n)

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

$$\log N_h + 1 = (h+2)\log\Phi - \frac{1}{2}\log 5$$

$$h = \frac{\log N_h + 1 - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi}$$

$$h = 1.44 \log N_h$$

AVL树的实现

- 既然AVL平衡树确实能够改进BST树的性能，避免退化情形，我们来看看向AVL树插入一个新key，如何才能保持AVL树的平衡性质
- 首先，新key必定以叶节点形式插入到AVL树中
  - 叶节点的平衡因子是0，其本身无需重新平衡
  - 但会影响其父节点的平衡因子：
    - 如果作为左子节点插入，则父节点平衡因子会增加1；
    - 如果作为右子节点插入，则父节点平衡因子会减少1。
  - 这种影响可能随着其父节点到根节点的路径一直传递上去，直到：
    - 传递到根节点为止；
    - 或者某个父节点平衡因子被调整到0，不再影响上层节点的平衡因子为止。
      - （无论从-1或者1调整到0，都不会改变子树高度）
- 将AVL树作为BST树子类实现，TreeNode中增加balance_factor

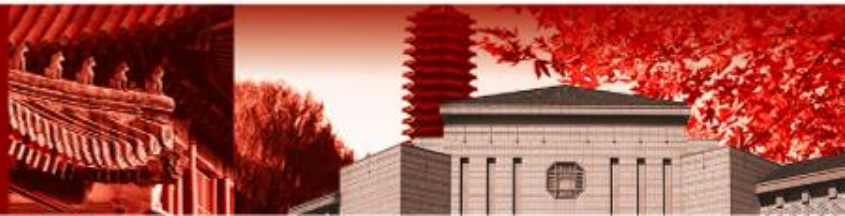
AVL树的实现：_put方法

- 重新定义_put方法即可

```
def _put(self, key, value, current_node):
    if key < current_node.key:
        if current_node.has_left_child():
            self._put(key, value, current_node.left_child)
        else:
            current_node.left_child = TreeNode(key, value, parent=current_node)
            self.update_balance(current_node.left_child)
    else:
        if current_node.has_right_child():
            self._put(key, value, current_node.right_child)
        else:
            current_node.right_child = TreeNode(key, value, parent=current_node)
            self.update_balance(current_node.right_child)
```

调整因子

调整因子



AVL树的实现：update_balance方法

重新平衡

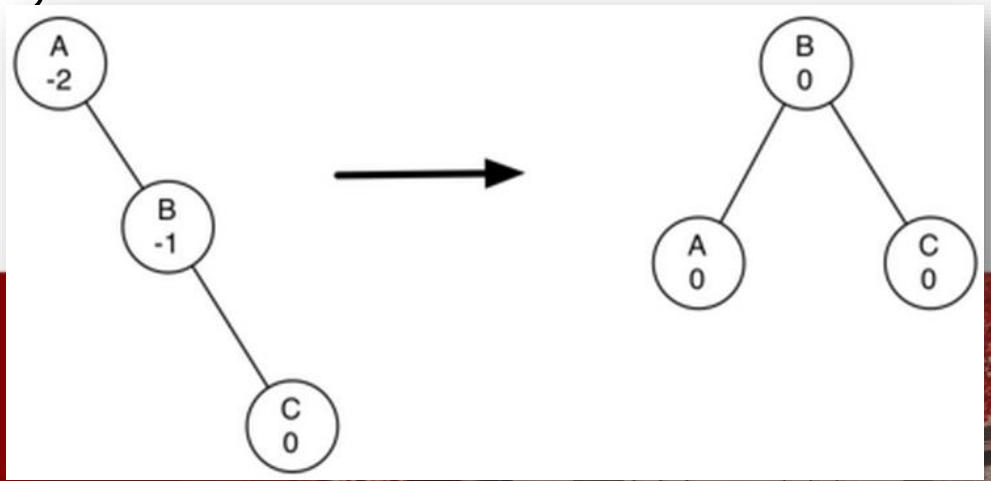
调整父节点因子

```
def update_balance(self, node):
    if node.balance_factor > 1 or node.balance_factor < -1:
        self.rebalance(node)
        return
    if node.parent != None:
        if node.is_left_child():
            node.parent.balance_factor += 1
        elif node.is_right_child():
            node.parent.balance_factor -= 1

        if node.parent.balance_factor != 0:
            self.update_balance(node.parent)
```

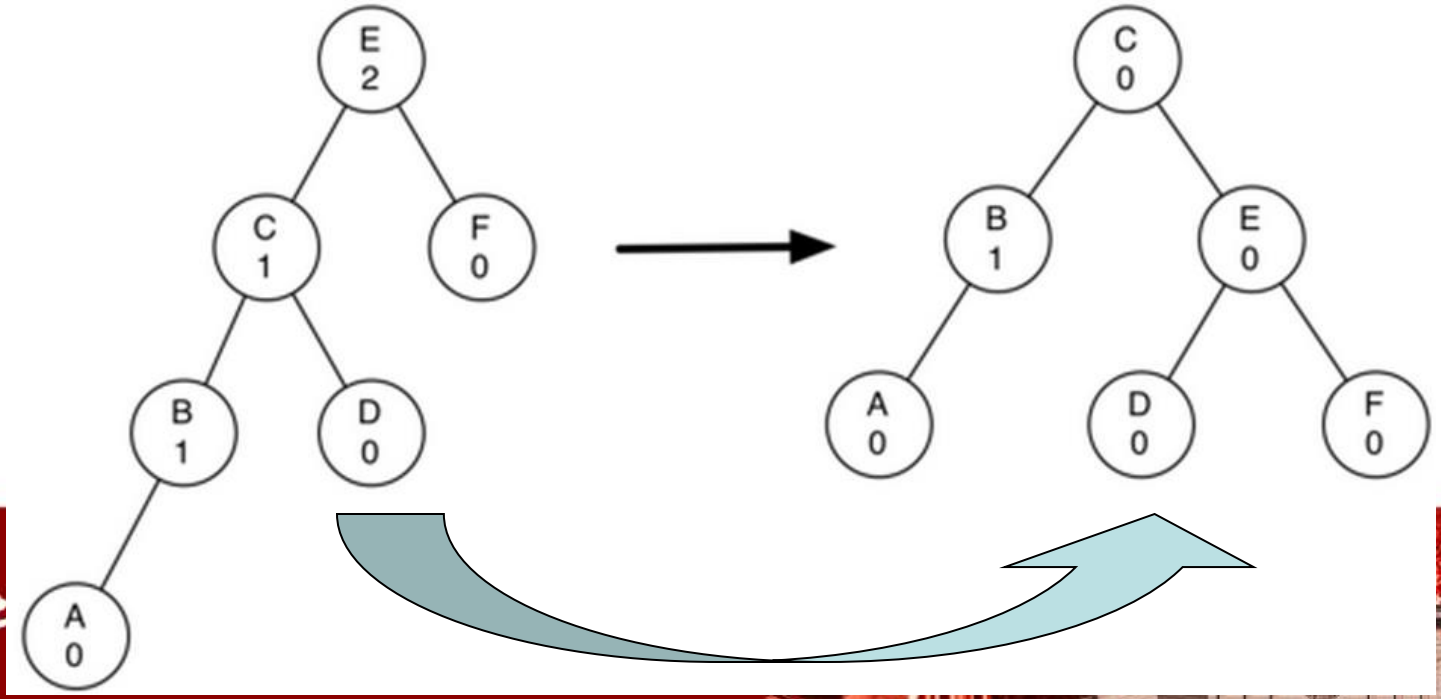
AVL树的实现：rebalance重新平衡

- 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
- 如图，是一个“右重”子树A的左旋转（并保持BST性质）
  - –将右子节点B提升为子树的根
  - –将旧根节点A作为新根节点B的左子节点
  - –如果新根节点B原来有左子节点，则将此节点设置为A的右子节点
  - –（A的右子节点一定为空，为什么？）



AVL树的实现：rebalance重新平衡

- 更复杂一些的情况：如图的“左重”子树右旋转
  - 旋转后，新根节点将旧根节点作为右子节点，但是新根节点原来已有右子节点，需要将原有的右子节点重新定位！
  - 原有的右子节点D改到旧根节点E的左子节点
  - 同样，E的左子节点在旋转后一定为空



AVL树的实现：rotate_left代码

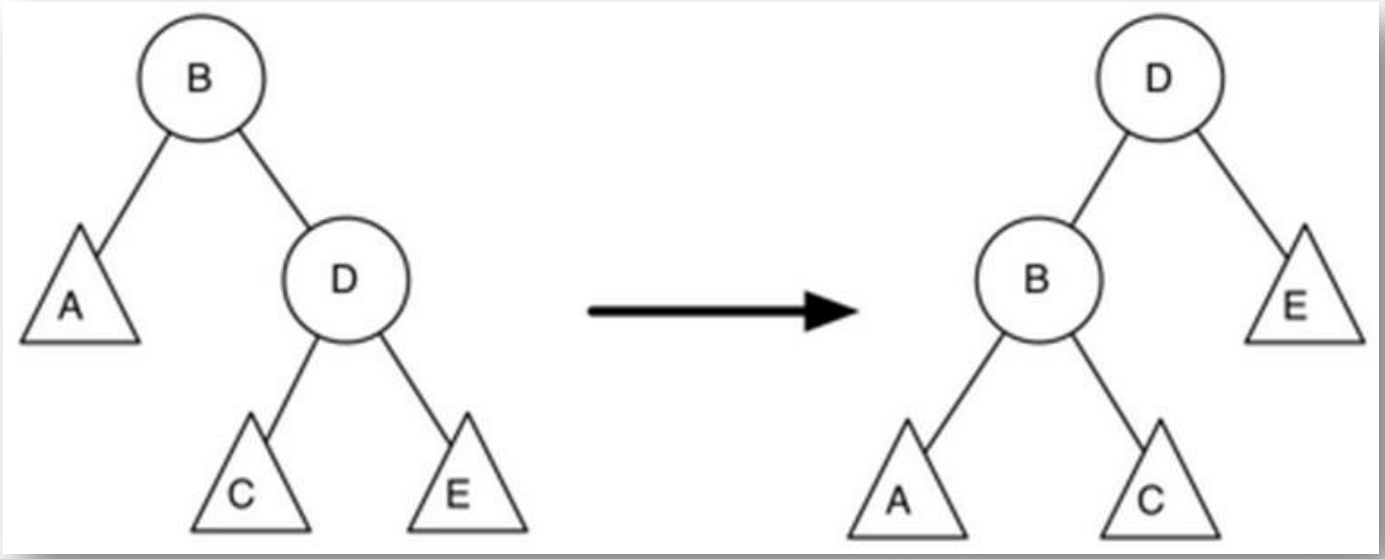
综合两种右重情形

```
def rotate_left(self, node):
    new_root = node.right_child
    node.right_child = new_root.left_child
    if new_root.left_child != None:
        new_root.left_child.parent = node
    new_root.parent = node.parent
    if node.is_root():
        self.root = new_root
    else:
        if node.is_left_child():
            node.parent.left_child = new_root
        else:
            node.parent.right_child = new_root
    new_root.left_child = node
    node.parent = new_root
    node.balance_factor = node.balance_factor + 1 - min(new_root.balance_factor, 0)
    new_root.balance_factor = new_root.balance_factor + 1 + max(node.balance_factor, 0)
```

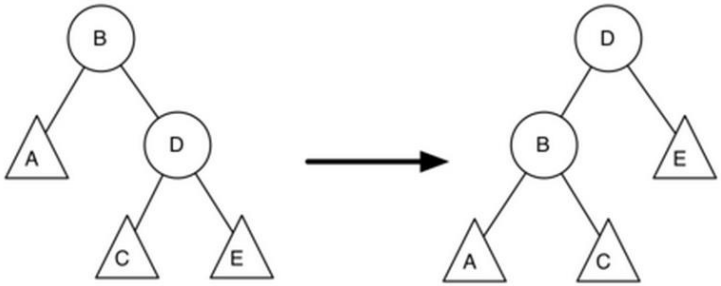
仅有两个节点需要调整因子

AVL树的实现：如何调整平衡因子

- 看看左旋转对平衡因子的影响
  - –保持了次序ABCDE
  - –ACE的平衡因子不变
    - $h_A/h_C/h_E$不变
  - –主要看BD新旧关系



AVL树的实现：如何调整平衡因子

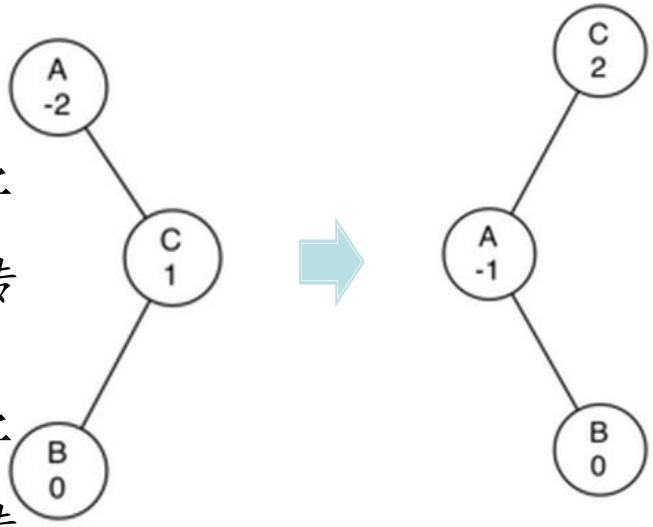


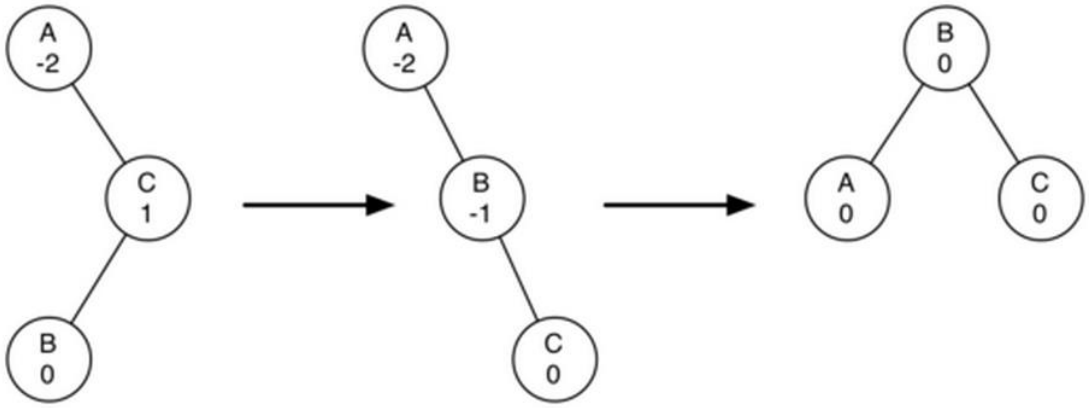
- 我们来看看B的变化
  - –新B= h_A- h_C
  - –旧B= h_A- 旧h_D
  - –而：旧h_D= 1+ max(h_C, h_E)，所以旧B= h_A- (1+ max(h_C, h_E))
  - –新B- 旧B= 1+ max(h_C, h_E)- h_C
  - –新B= 旧B+ 1+ max(h_C, h_E)- h_C；把h_C移进max函数里就有
  - –新B= 旧B+ 1+ max(0, -旧D) <==> 新B= 旧B+ 1- min(0, 旧D)

```
node.balance_factor = node.balance_factor + 1 - min(new_root.balance_factor, 0)
```

AVL树的实现：更复杂的情形

- 下图的“右重”子树，单纯的左旋转无法实现平衡
  - 左旋转后变成“左重”了
  - “左重”再右旋转，还回到“右重”
- 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





AVL树的实现：rebalance代码

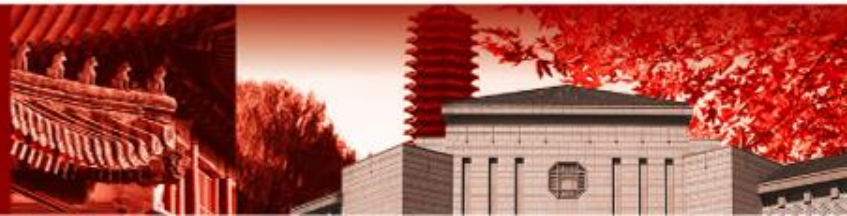
右重需要左旋

右子节点左重
先右旋

左重需要右旋

左子节点右重
先左旋

```
def rebalance(self, node):
    if node.balance_factor < 0:
        if node.right_child.balance_factor > 0:
            self.rotate_right(node.right_child)
            self.rotate_left(node)
        else:
            self.rotate_left(node)
    elif node.balance_factor > 0:
        if node.left_child.balance_factor < 0:
            self.rotate_left(node.left_child)
            self.rotate_right(node)
        else:
            self.rotate_right(node)
```

AVL树的实现：结语

- 经过复杂的put方法，AVL树始终维持平衡性质，这样get方法也始终保持O(log n)的高性能
  - 不过，put方法的代价有多大？
- 将AVL树的put方法分为两个部分：
  - 需要插入的新节点是叶节点，更新其所有父节点和祖先节点的代价最多为O(log n)
  - 如果插入的新节点引发了不平衡，重新平衡最多需要2次旋转，但旋转的代价与问题规模无关，是常数O(1)
  - 所以整个put方法的时间复杂度还是O(log n)
- 从AVL树删除一个节点，如何保持平衡？

AVL树的实现：课堂作业

- 如图所示的“右重”子树如何平衡？
- 请图示旋转过程