



数据结构与算法 (Python版)

二叉查找树实现及算法分析 (下)

陈斌 北京大学 gischen@pku.edu.cn

二叉查找树的实现：BST.delete方法

❖ 有增就有减，最复杂的delete方法：

用_get找到要删除的节点，然后调用remove来删除，找不到则提示错误

```
def delete(self, key):
    if self.size > 1:
        nodeToRemove = self._get(key, self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = 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 = self.size - 1
    else:
        raise KeyError('Error, key not in tree')
```

二叉查找树的实现：BST.delete方法

❖ __delitem__ 特殊方法

实现 `del myZipTree['PKU']` 这样的语句操作

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

❖ 在delete中，最复杂的是找到key对应的节点之后的remove节点方法！

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

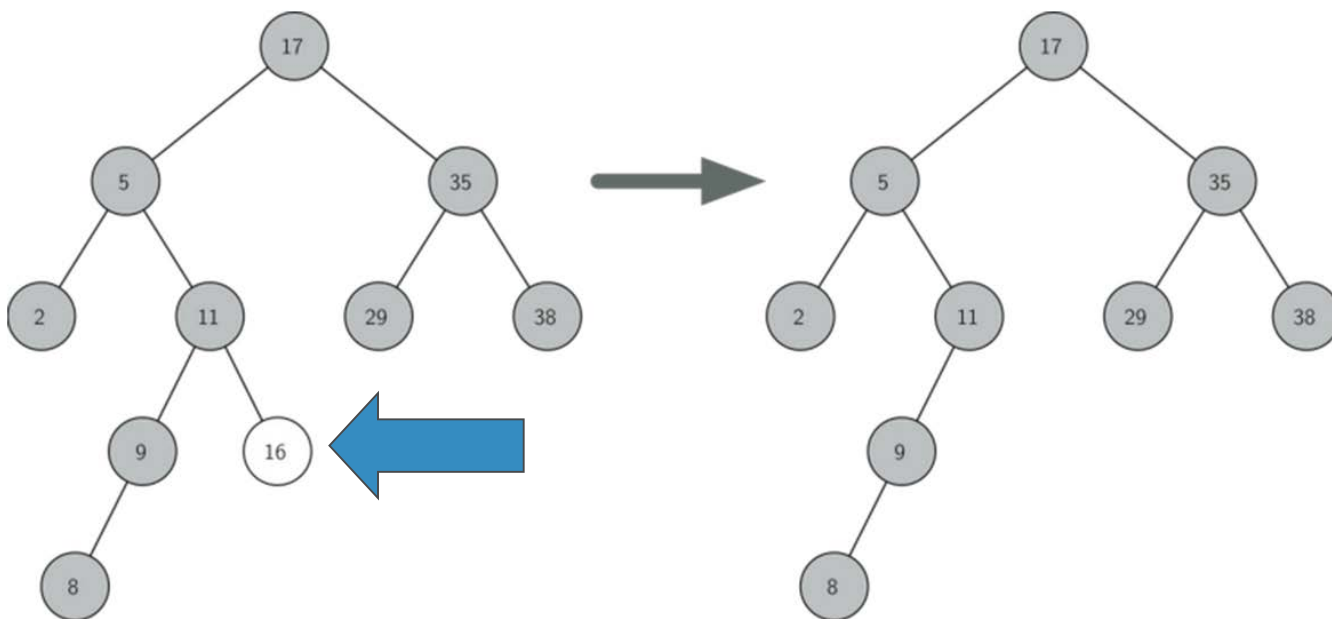
这个节点有1个子节点

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

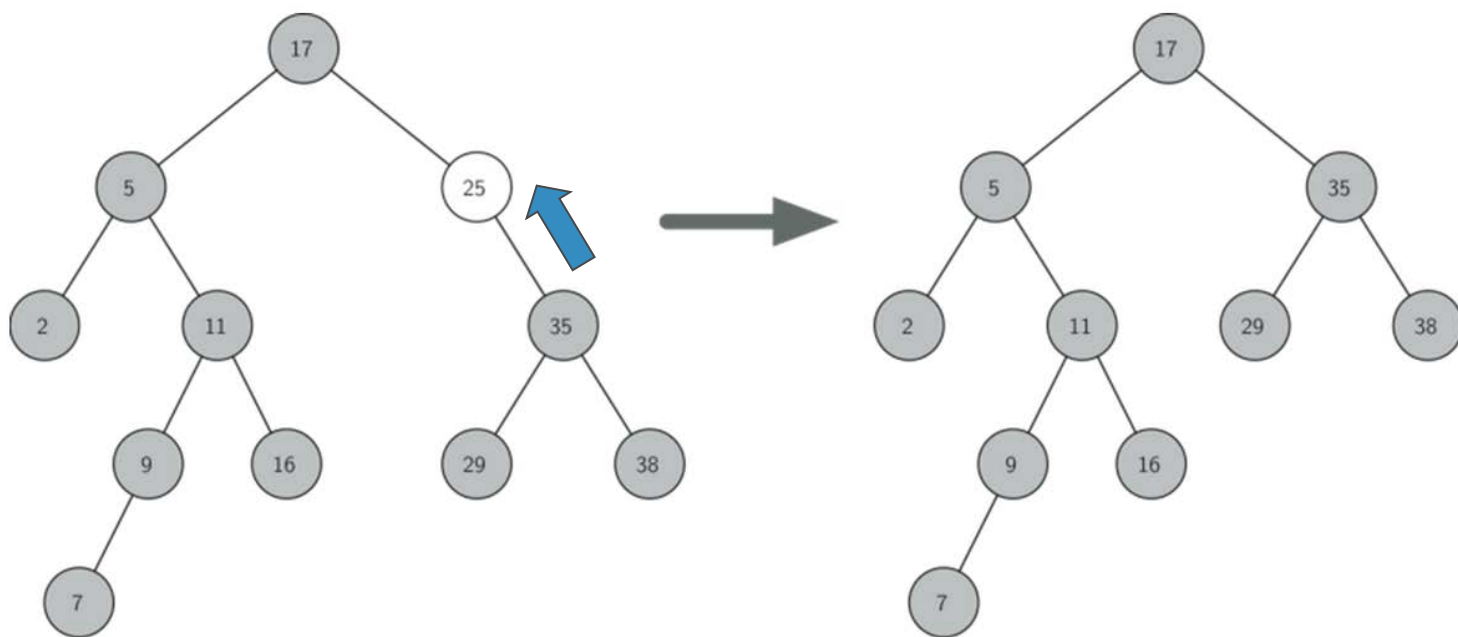
```
if currentNode.isLeaf(): #leaf
    if currentNode == currentNode.parent.leftChild:
        currentNode.parent.leftChild = None
    else:
        currentNode.parent.rightChild = None
```



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



二叉查找树的实现：BST.remove方法

❖ 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？


```
else: # this node has one child
```

```
    if currentNode.hasLeftChild():
```

```
        if currentNode.isLeftChild():
```

左子节点删除

```
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.leftChild
```

```
        elif currentNode.isRightChild():
```

右子节点删除

```
            currentNode.leftChild.parent = currentNode.parent
            currentNode.parent.rightChild = currentNode.leftChild
```

```
    else:
```

根节点删除

```
        currentNode.replaceNodeData(currentNode.leftChild.key,
                                     currentNode.leftChild.payload,
                                     currentNode.leftChild.leftChild,
                                     currentNode.leftChild.rightChild)
```

```
    else:
```

```
        if currentNode.isLeftChild():
```

左子节点删除

```
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.leftChild = currentNode.rightChild
```

```
        elif currentNode.isRightChild():
```

右子节点删除

```
            currentNode.rightChild.parent = currentNode.parent
            currentNode.parent.rightChild = currentNode.rightChild
```

```
    else:
```

根节点删除

```
        currentNode.replaceNodeData(currentNode.rightChild.key,
                                     currentNode.rightChild.payload,
                                     currentNode.rightChild.leftChild,
                                     currentNode.rightChild.rightChild)
```


二叉查找树的实现：BST.remove方法

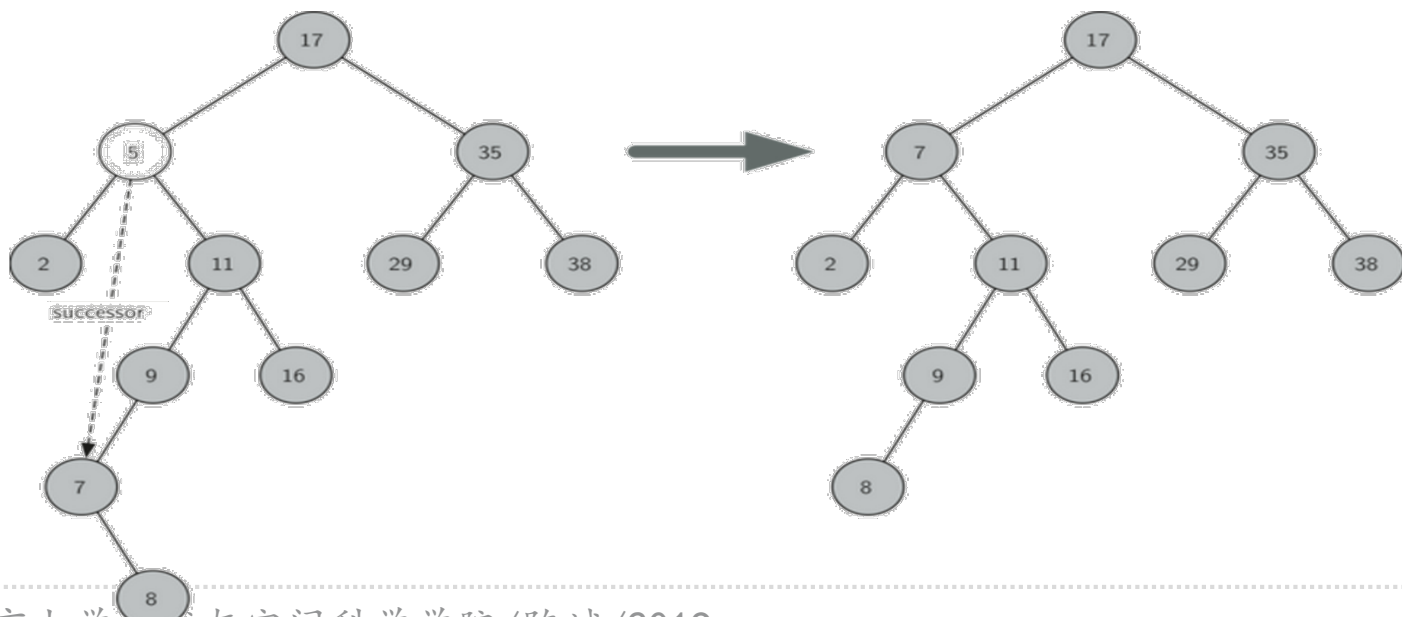
❖ 第3种情形最复杂：被删节点有2个子节点

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”

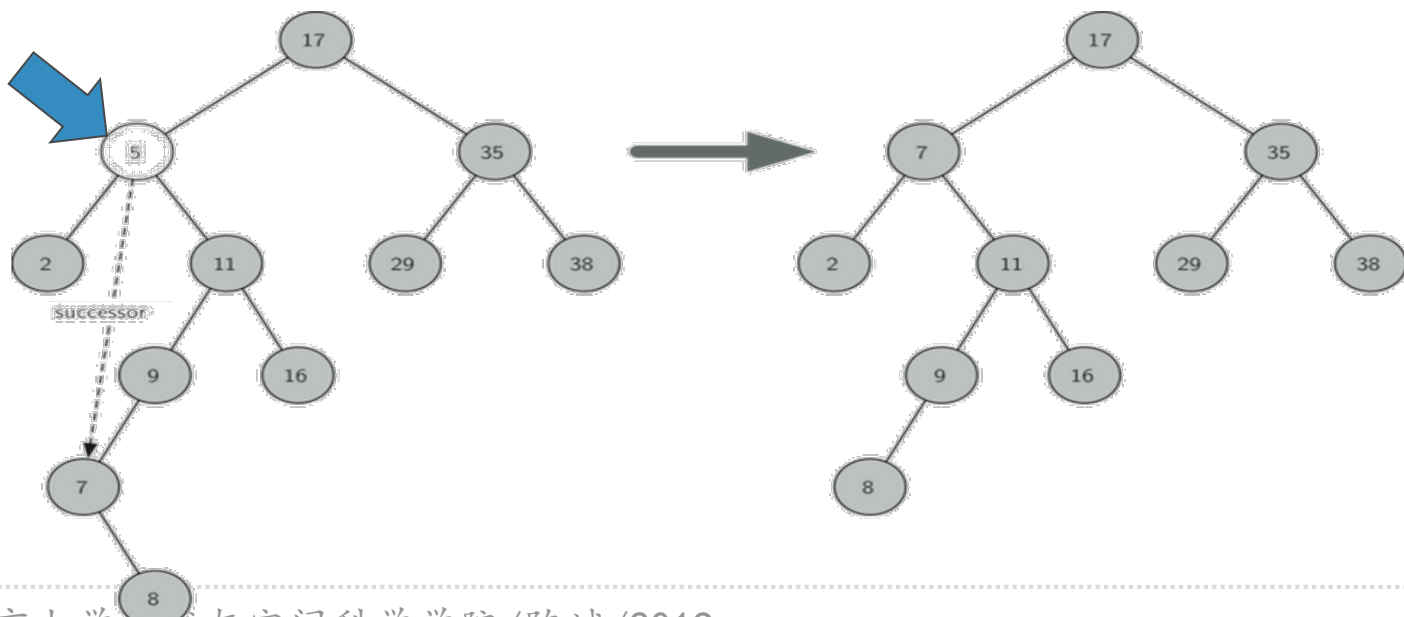


二叉查找树的实现：BST.remove方法

❖ 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。



二叉查找树的实现：BST.remove方法

❖ BinarySearchTree类：remove方法 (情形3)




```
elif currentNode.hasBothChildren(): #interior
    succ = currentNode.findSuccessor()
    succ.spliceOut()
    currentNode.key = succ.key
    currentNode.payload = succ.payload
```

二叉查找树的实现：BST.remove方法

❖ 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
```



二叉查找树的实现: BST.remove方法

❖ 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

目前不会遇到

摘出带右子节点的节点

二叉查找树：算法分析（以put为例）

- ❖ 其性能决定因素在于二叉搜索树的**高度**（最大层次），而其**高度又受数据项key插入顺序的影响**。
- ❖ **如果key的列表是随机分布的话**，那么大于和小于根节点key的键值大致相等
- ❖ **BST的高度就是 $\log_2 n$ （ n 是节点的个数），而且，这样的树就是平衡树**
- ❖ **put方法最差性能为 $O(\log_2 n)$ 。**

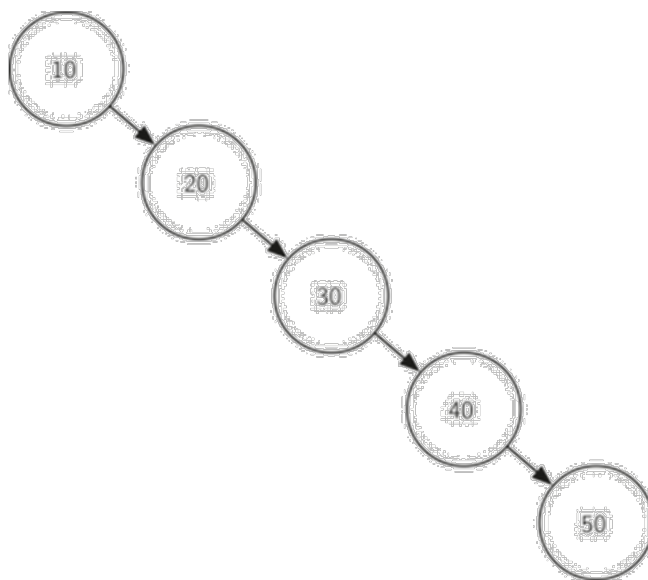
二叉查找树：算法分析（以put为例）

❖ 但key列表分布**极端情况**就完全不同

按照从小到大顺序插入的话，如下图

这时候put方法的性能为 $O(n)$

其它方法也是类似情况



❖ **如何改进BST？ 不受key插入顺序影响？**

