Binary search tree and its operations
二叉查找树及操作

BinarySearch Tree
二叉查找树

In the implementation of ADTMap, different data structures and search algorithms can be used to save and search key

在ADTMap的实现方案中，可以采用不同的数据结构和搜索算法来保存和查找key

two schemes have been implemented before

前面已经实现了两个方案

- Ordered table data structure + binary search algorithm
  有序表数据结构+二分搜索算法
- Hash table data structure + hash and its collision's resolution algorithm
  散列表数据结构+散列及冲突解决算法

Let's try to save the key with a binary search tree to achieve fast key search

下面我们来试试用二叉查找树保存key，实现key的快速搜索

Binary search tree: ADTMap
二叉查找树：ADTMap

Review the operation of ADTMap:
复习一下ADTMap的操作：

Map(): Creates an empty map

Map()：创建一个空映射

put(key,val): Add the key-val association pair to the map, if the key already exists, replace the old association value with val;

put(key,val)：将key-val关联对加入映射中，如果key已经存在，则将val替换旧关联值；

get(key): Given the key, return the associated data value, if it does not exist, return None;

get(key)：给定key，返回关联的数据值，如不存在，则返回None；

del: delete the key-val association through the statement form of delmap[key];

del：通过delmap[key]的语句形式删除key-val关联；

len(): returns the number of key-val associations in the map;

len()：返回映射中key-val关联的数目；

in: Returns whether the key exists in the association through the statement form of keyinmap, a boolean value

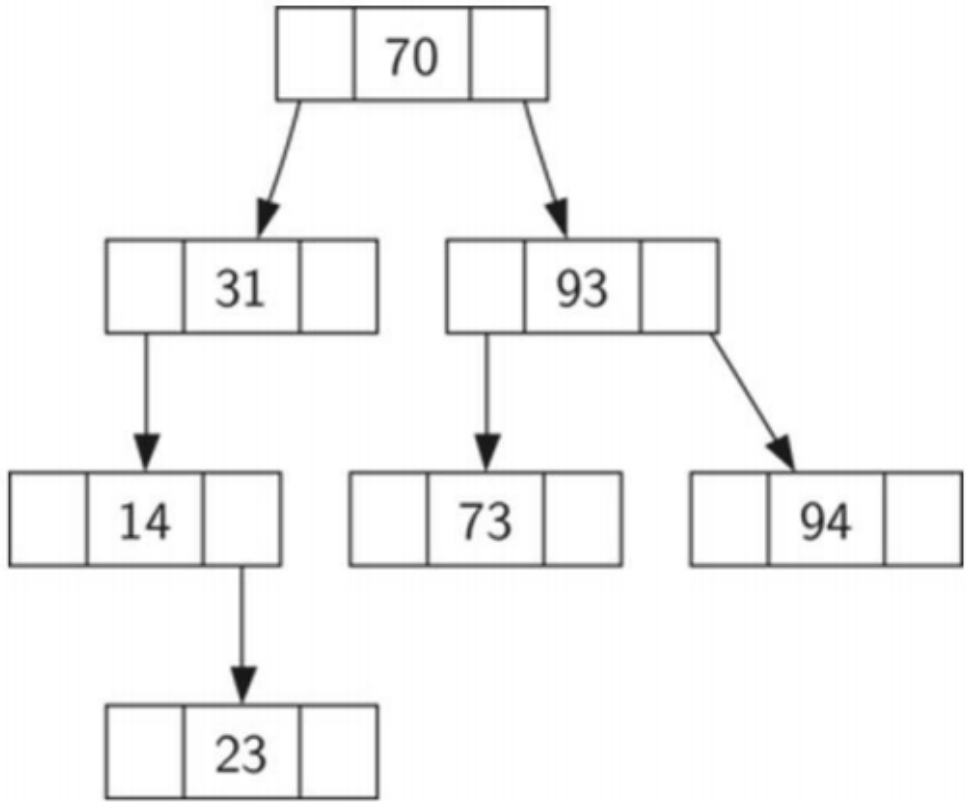
in：通过keyinmap的语句形式，返回key是否存在于关联中，布尔值

Properties of Binary Search Tree BST

二叉查找树BST的性质

Keys smaller than the parent node appear in the left subtree, and keys larger than the parent node appear in the right subtree.

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。



Properties of Binary Search Tree BST
二叉查找树BST的性质

Insert in the order 70,31,93,94,14,23,73

按照70,31,93,94,14,23,73的顺序插入

The first 70 inserted becomes the root of the tree

首先插入的70成为树根

31 is smaller than 70, put it on the left child node

31比70小，放到左子节点

93 is bigger than 70, put it on the right child node

93比70大，放到右子节点

94 is bigger than 93, put it on the right child node

94比93大，放到右子节点

14 is smaller than 31, put it on the left child node

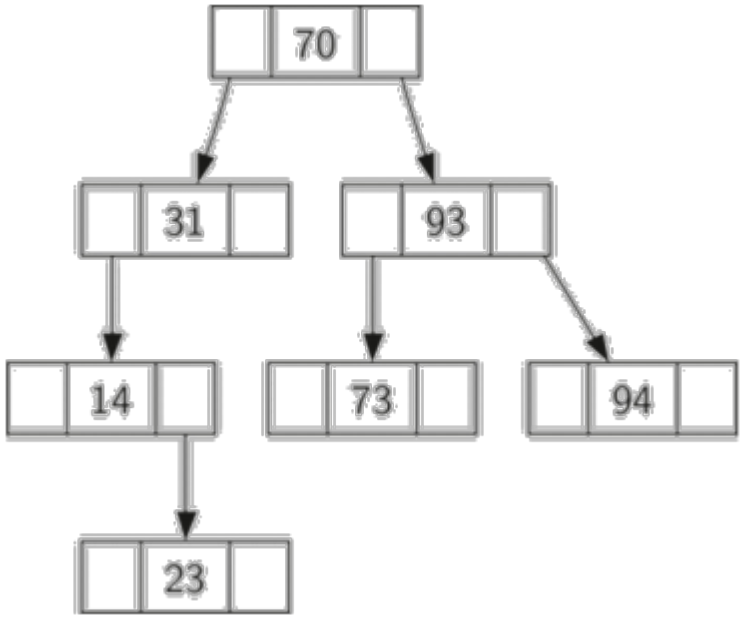
14比31小，放到左子节点

23 is bigger than 14, put it to the right

23比14大，放到其右

73 is smaller than 93, put it to the left

73比93小，放到其左



Note: The insertion order is different, the resulting BST is also different

注意：插入顺序不同，生成的BST也不同

Implementation of Binary Search Tree: Node and Link Structure
二叉搜索树的实现：节点和链接结构

We need to use two classes(BST and TreeNode), the root member of BST refers to the root node TreeNode

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

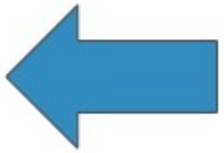
```
class BinarySearchTree:

    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__()
```



Implementation of binary search tree: TreeNode class
二叉搜索树的实现：TreeNode类

```
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
```

key value
键值
data item
数据项
Left and right child node parent node
左、右子节点父节点

Implementation of binary search tree: TreeNode class
二叉搜索树的实现：TreeNode类

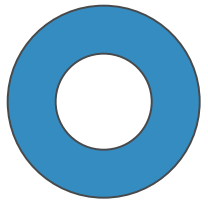
```
def isRoot(self):
    return not self.parent

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

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

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

def replaceNodeData(self,key,value,lc,rc):
    self.key = key
    self.payload = value
    self.leftChild = lc
    self.rightChild = rc
    if self.hasLeftChild():
        self.leftChild.parent = self
    if self.hasRightChild():
        self.rightChild.parent = self
```



Implementation and Algorithm Analysis of Binary Search Tree

二叉查找树实现及算法分析

Implementation of binary search tree: BST.put method
二叉搜索树的实现：BST.put方法

put(key,val) method: insert key to construct BST
put(key,val)方法：插入key构造BST

First, see if the BST is empty. If there is no node, then the key becomes the root node root

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

Otherwise, a recursive function _put(key,val,root) is called to place the key

否则，就调用一个递归函数_put(key,val,root)来放置key

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

Implementation of binary search tree: _put helper method
二叉搜索树的实现：_put辅助方法

_put(key,val,currentNode) process
_put(key, val, currentNode)的流程

If key is smaller than currentNode, then _put to the left subtree

如果key比currentNode小，那么_put到左子树

- But if there is no left subtree, then the key becomes the left child

但如果没有左子树，那么key就成为左子节点

If key is greater than currentNode, then _put to the right subtree

如果key比currentNode大，那么_put到右子树

- But if there is no right subtree, then the key becomes the right child

但如果没有右子树，那么key就成为右子节点

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key,val,currentNode.leftChild)
        else:
            currentNode.leftChild = \
                TreeNode(key,val,parent=currentNode)
    else:
        if currentNode.hasRightChild():
            self._put(key,val,currentNode.rightChild)
        else:
            currentNode.rightChild = \
                TreeNode(key,val,parent=currentNode)
```

recursive left subtree
递归左子树

recursive right subtree
递归右子树

Implementation of Binary Search Tree: Index Assignment
二叉搜索树的实现：索引赋值

Make __setitem__ a special method (double underscore before and after)

随手把__setitem__做了特殊方法(前后双下划线)

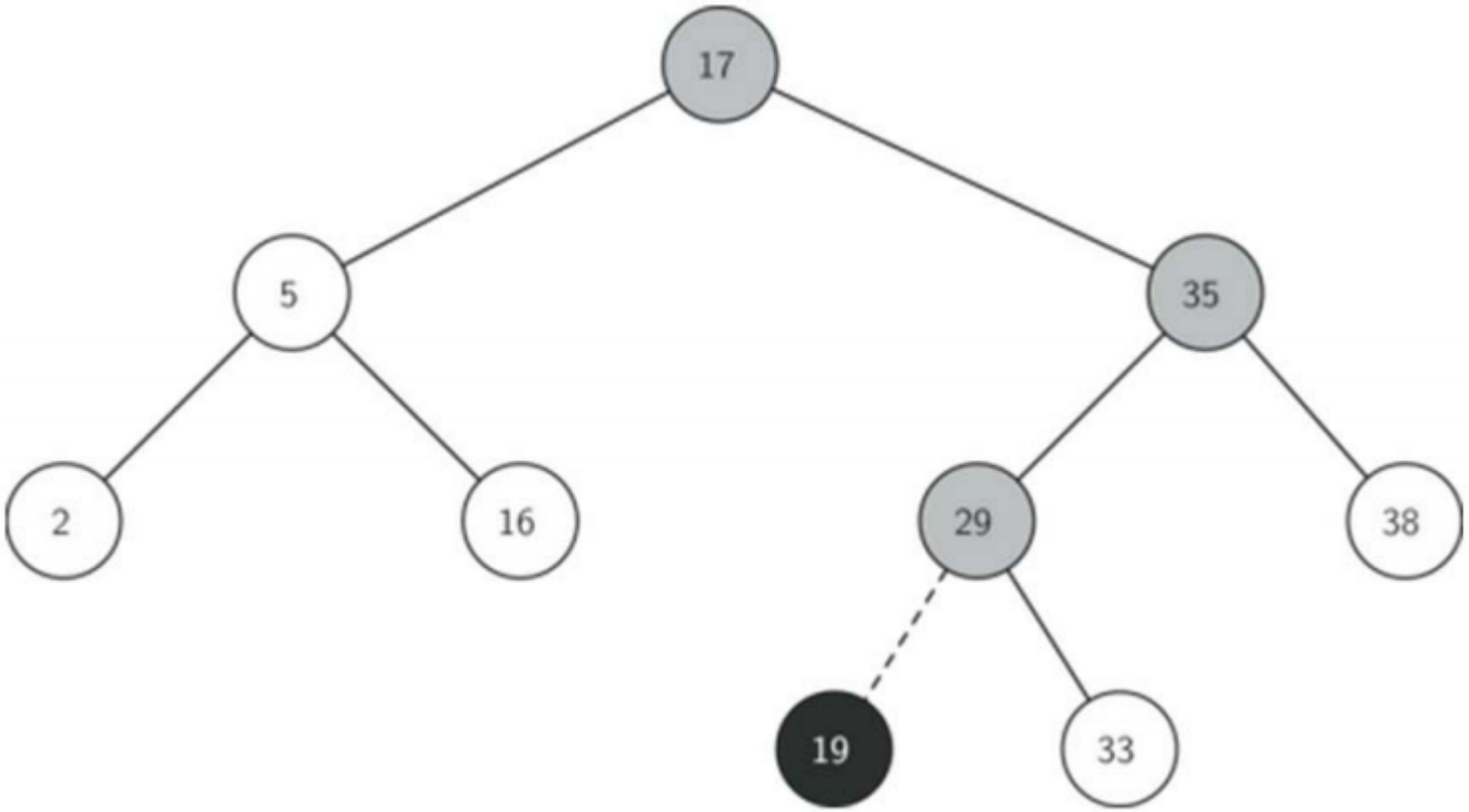
```
myZipTree['JNU']=100871
```

```
def __setitem__(self,k,v):
    self.put(k,v)
```

```
mytree = BinarySearchTree()
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"
```

Implementation of binary search tree: BST.put diagram
二叉搜索树的实现：BST.put图示

Insert key=19, the change process of currentNode (gray):
插入key=19，currentNode的变化过程(灰色)：

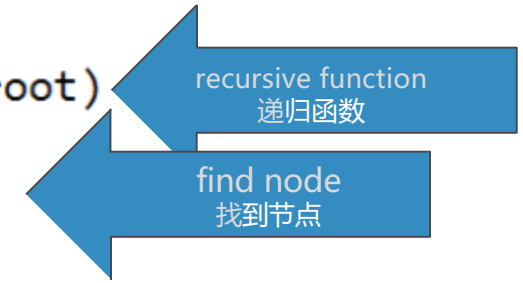


Implementation of binary search tree: BST.get method
二叉搜索树的实现：BST.get方法

Find the node where the key is located in the tree and get the payload
在树中找到key所在的节点取到payload

```
def get(self,key):
    if self.root:
        res = self._get(key,self.root)
        if res:
                return res.payload
        else:
                return None
    else:
        return None

def _get(self,key,currentNode):
    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)
```



Implementation of Binary Search Tree: Index and Attribution Judgment
二叉搜索树的实现：索引和归属判断

__getitem__ special method
__getitem__特殊方法

Implement val=myZipTree['PKU']

实现val=myZipTree['PKU']

__contains__ special method
__contains__特殊方法

Implement the attribution operator in of 'PKU' inmyZipTree

实现'PKU'inmyZipTree的归属判断运算符in

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

def __contains__(self,key):
    if self._get(key,self.root):
        return True
    else:
        return False
```

```
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"

print(3 in mytree)
print(mytree[6])
```

Implementation of a binary search tree: iterators
二叉搜索树的实现：迭代器

We can enumerate all keys in the dictionary with a *for* loop

我们可以用for循环枚举字典中的所有key

ADTMap should also implement such iterator function

ADTMap也应该实现这样的迭代器功能

The special method __iter__ can be used to implement for iteration

特殊方法__iter__可以用来实现for迭代

The __iter__ method in the BST class directly calls the method of the same name in the TreeNode

BST类中的__iter__方法直接调用了TreeNode中的同名方法

```
mytree = BinarySearchTree()
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"
```

iterative loop
迭代循环

```
print(3 in mytree)
print(mytree[6])
del mytree[3]
print(mytree[2])
for key in mytree:
    print key, mytree[key]
```

Implementation of a binary search tree: iterators
二叉搜索树的实现：迭代器

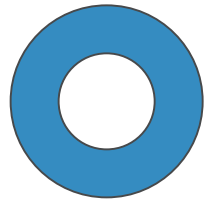
__iter__ iterator in TreeNode class
TreeNode类中的__iter__迭代器

The *for* iteration is used in the iterator function, which is actually a recursive function
迭代器函数中用了for迭代，实际上是递归函数

yield is the return value for each iteration
yield是对每次迭代的返回值

Iteration of inorder traversal
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
```



Implementation of binary search tree: BST.delete method
二叉查找树的实现：BST.delete方法

There is an increase and a decrease, the most complicated delete method:

有增就有减，最复杂的delete方法：

Use _get to find the node to be deleted, and then call remove to delete it, if it is not found, it will prompt an error

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')
```

Implementation of binary search tree: BST.delete method
二叉查找树的实现：BST.delete方法

__delitem__ special method
__delitem__特殊方法

Implement a statement operation like delmyZipTree['PKU']

实现delmyZipTree['PKU']这样的语句操作

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

In delete, the most complicated is the remove node method after finding the node corresponding to the 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！

Implementation of binary search tree: BST.remove method
二叉查找树的实现：BST.remove方法

To remove a node from BST, it is also required to maintain the properties of BST, which can be divided into the following three situations:

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：

①This node has no children

这个节点没有子节点

②This node has 1 child node

这个节点有1个子节点

③This node has 2 child nodes

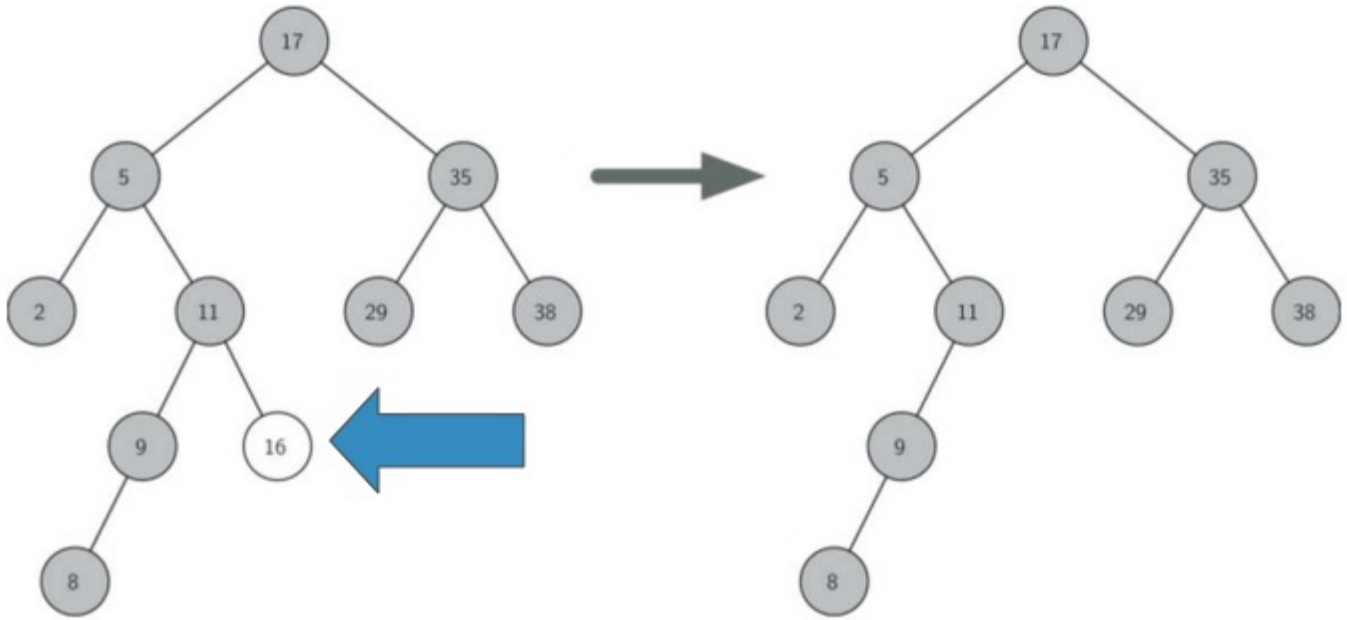
这个节点有2个子节点

Implementation of binary search tree: BST.remove method
二叉查找树的实现：BST.remove方法

if there is no child node, then it is easy to do, just delete it directly

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
```



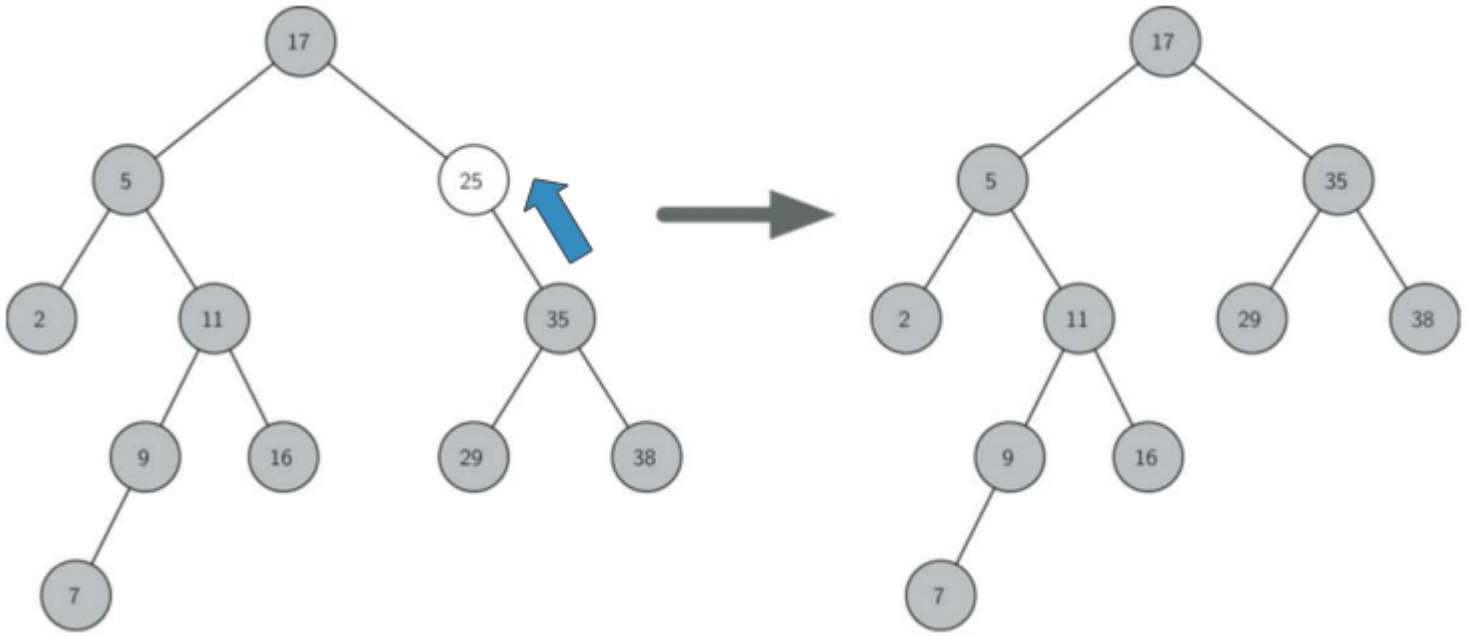
Implementation of binary search tree: BST.remove method
二叉查找树的实现：BST.remove方法

The second case is slightly more complicated: the deleted node has 1 child node

第2种情形稍复杂：被删节点有1个子节点

Solution: Move the only child node up and replace the position of the deleted node

解决：将这个唯一的子节点上移，替换掉被删节点的位置



Implementation of binary search tree: BST.remove method
二叉查找树的实现：BST.remove方法

But the replacement operation needs to distinguish between several cases:
但替换操作需要区分几种情况：

- Is the child node of the deleted node left? Or the right child node?
  被删节点的子节点是左？还是右子节点？
- Is the deleted node itself the left of its parent node? Or the right child node?
  被删节点本身是其父节点的左？还是右子节点？
- Is the deleted node itself the root node?
  被删节点本身就是根节点？

```
else: # this node has one child
  if currentNode.hasLeftChild():
    if currentNode.isLeftChild():
      currentNode.leftChild.parent = currentNode.parent
      currentNode.parent.leftChild = currentNode.leftChild
    elif currentNode.isRightChild():
      currentNode.leftChild.parent = currentNode.parent
      currentNode.parent.rightChild = currentNode.leftChild
    else:
      currentNode.replaceNodeData(currentNode.leftChild.key,
                                  currentNode.leftChild.payload,
                                  currentNode.leftChild.leftChild,
                                  currentNode.leftChild.rightChild)
  else:
    if currentNode.isLeftChild():
      currentNode.rightChild.parent = currentNode.parent
      currentNode.parent.leftChild = currentNode.rightChild
    elif currentNode.isRightChild():
      currentNode.rightChild.parent = currentNode.parent
      currentNode.parent.rightChild = currentNode.rightChild
    else:
      currentNode.replaceNodeData(currentNode.rightChild.key,
                                  currentNode.rightChild.payload,
                                  currentNode.rightChild.leftChild,
                                  currentNode.rightChild.rightChild)
```

delete left child
左子节点删除

delete right child node
右子节点删除

root node delete
根节点删除

delete left child
左子节点删除

delete left child
右子节点删除

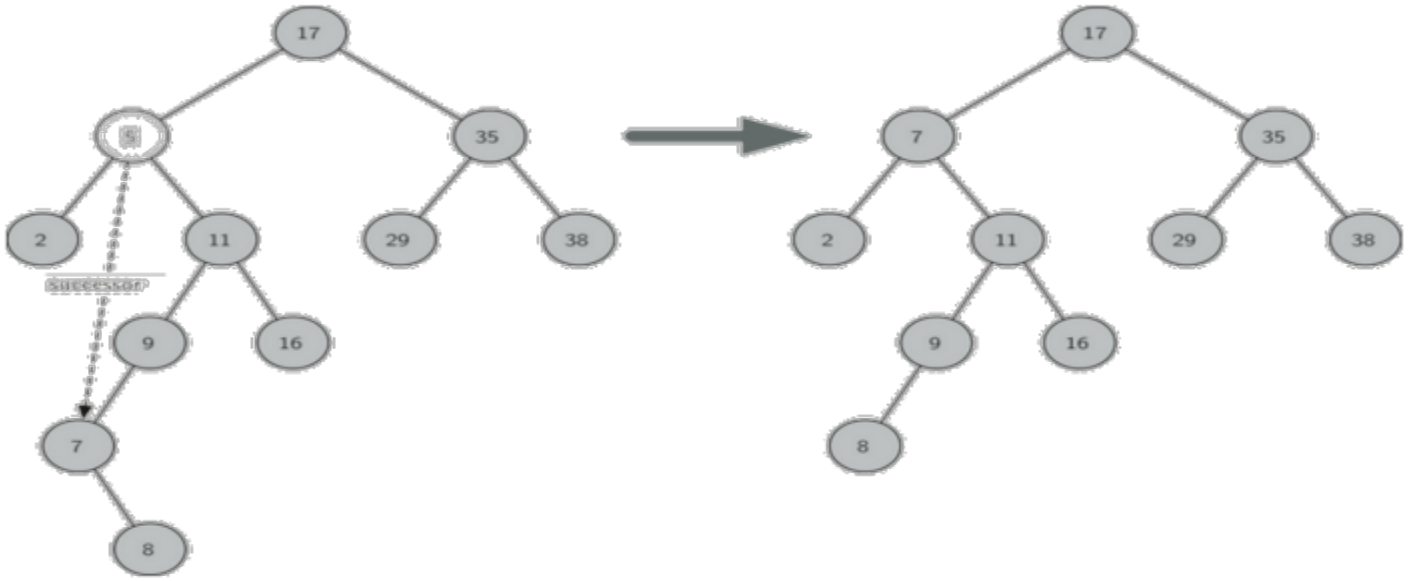
root node delete
根节点删除

Implementation of binary search tree: BST.remove method
二叉查找树的实现：BST.remove方法

The third case is the most complicated: the deleted node has 2 children
第3种情形最复杂：被删节点有2个子节点

At this time, it is not possible to simply move a child node up to replace the deleted node
这时无法简单地将某个子节点上移替换被删节点

However, another suitable node can be found to replace the deleted node. This suitable node is the next key value node of the deleted node, that is, the smallest one in the right subtree of the deleted node, which is called "successor"
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”



Implementation of binary search tree: BST.remove method
二叉查找树的实现：BST.remove方法

The third case is the most complicated: the deleted node has 2 children

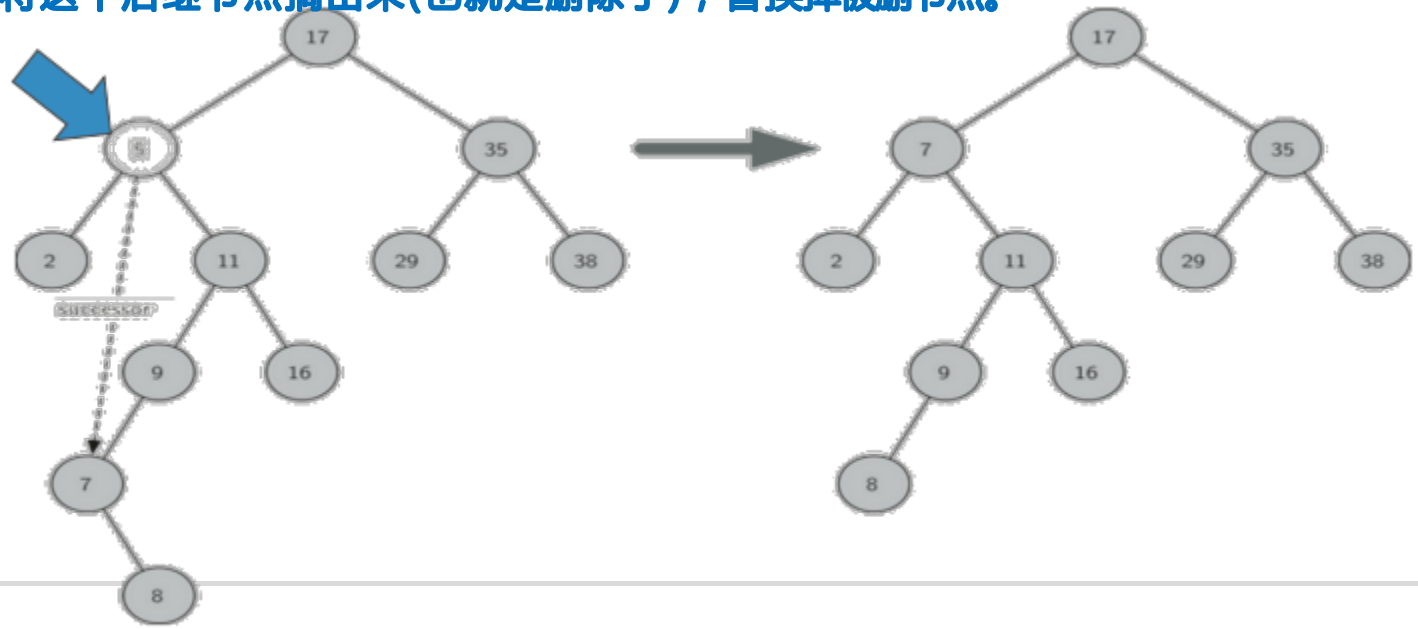
第3种情形最复杂：被删节点有2个子节点

It is certain that this successor node has at most 1 child node (it is a leaf node, or only has a right subtree)

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)

The successor node is picked out (that is, deleted), and the deleted node is replaced.

将这个后继节点摘出来(也就是删除了)，替换掉被删节点。



Implementation of binary search tree: BST.remove method

二叉查找树的实现：BST.remove方法

BinarySearchTree class: remove method (case 3)

BinarySearchTree类 ： remove法 (情形3)

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

Implementation of binary search tree: BST.remove method

二叉查找树的实现：BST.remove方法

TreeNode class: find successor nodes

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
```

not currently encountered
目前不会遇到

to the bottom left
到左下角

Implementation of binary search tree: BST.remove method

二叉查找树的实现：BST.remove方法

TreeNode class: pick out nodes spliceOut()

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
```

Pick out leaf nodes
摘出叶节点

not currently encountered
目前不会遇到

not currently encountered
摘出带右子节点的节点

Binary Search Tree: Algorithm Analysis (Take put as an example)

二叉查找树：算法分析(以put为例)

Its performance is determined by the height (maximum level) of the binary search tree, which in turn is affected by the order in which the keys of the data items are inserted.

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。

If the list of keys is randomly distributed, then the values of keys greater than and less than the root node's key are roughly equal

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

The height of BST is $\log_2 n$ (n is the number of nodes), and such a tree is a balanced tree

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

The worst performance of the put method is $O(\log_2 n)$.

put方法最差性能为$O(\log_2 n)$。

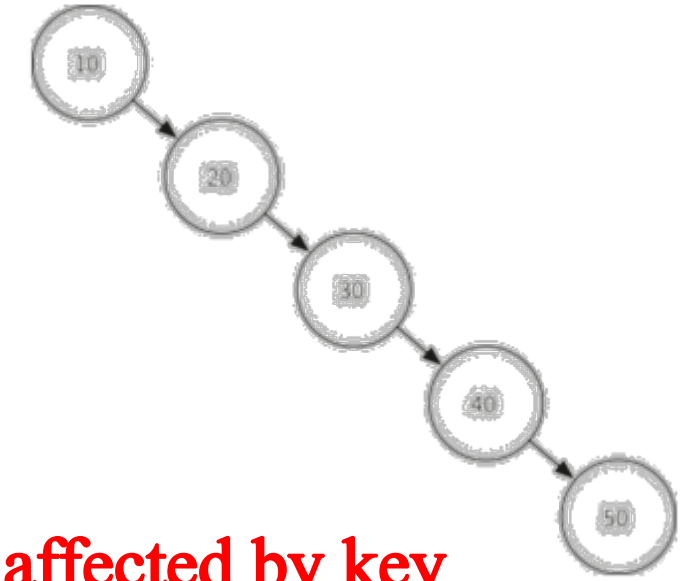
Binary Search Tree: Algorithm Analysis (Take put as an example)
二叉查找树：算法分析(以put为例)

But the extreme case of key list distribution is completely different
但key列表分布极端情况就完全不同

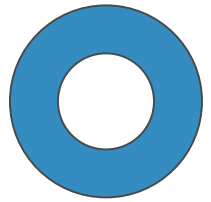
If inserted in order from smallest to largest, as shown below
按照从小到大顺序插入的话，如下图

At this time, the performance of the put method is O(n)
这时候put方法的性能为O(n)

Other methods are similar
其它方法也是类似情况



How to improve BST? Not affected by key insertion order?
如何改进BST？不受key插入顺序影响？



Definition and Properties of AVL Trees

AVL树的定义和性能

Balanced Binary Search Tree: Definition of AVL Tree
平衡二叉查找树：AVL树的定义

Let's take a look at a binary search tree that can always be balanced when the key is inserted: AVL tree

我们来看看能够在key插入时一直保持平衡的二叉查找树：AVL树

> AVL is the abbreviation of the inventor's name: G.M.Adelson-VelskiiandE.M.Landis
>
> AVL是发明者的名字缩写：G.M.Adelson-VelskiiandE.M.Landis

ADTMap is implemented using AVL tree, which is basically the same as the implementation of BST

利用AVL树实现ADTMap，基本上与BST的实现相同

The difference is only in the process of generating and maintaining the binary tree

不同之处仅在于二叉树的生成与维护过程

Balanced Binary Search Tree: Definition of AVL Tree

平衡二叉查找树：AVL树的定义

In the implementation of the AVL tree, it is necessary to track the "balance factor" parameter for each node. The balance factor is defined according to the height of the left and right subtrees of the node. To be precise, the height difference between the left and right subtrees:

AVL树的实现中，需要对每个节点跟踪“平衡因子”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：

$$balanceFactor=height(leftSubTree)-height(rightSubTree)$$

If the balance factor is greater than 0, it is called "left-heavy", and if it is less than zero, it is called "right-heavy". If the balance factor is equal to 0, it is called balance.

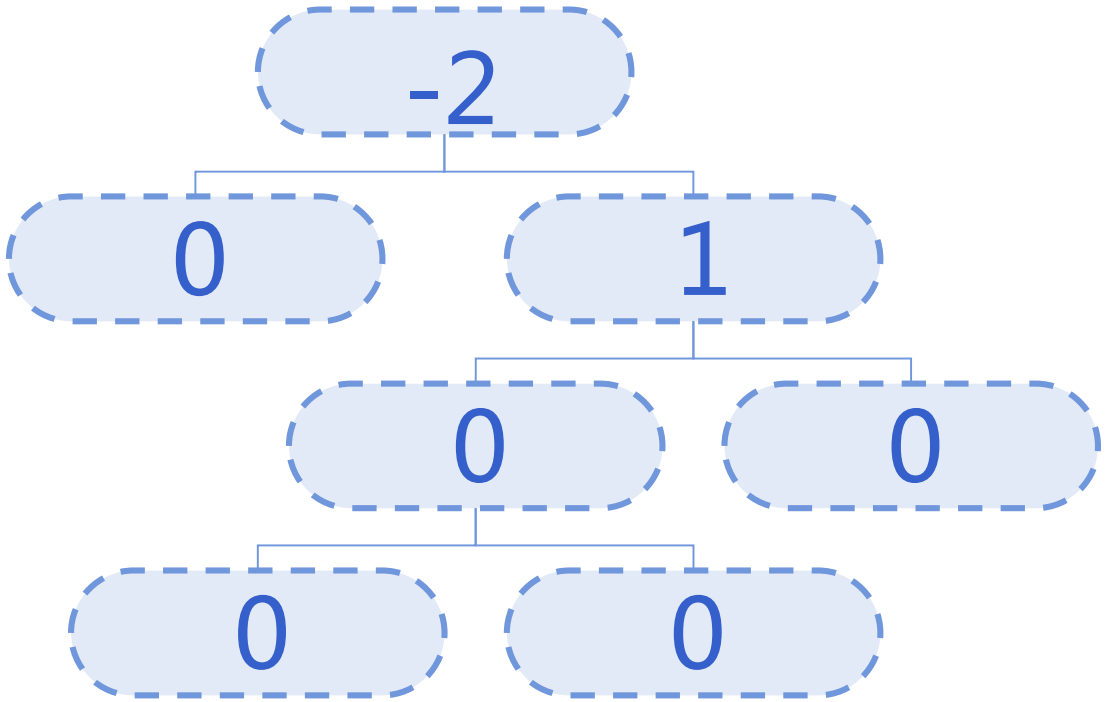
如果平衡因子大于0，称为“左重left-heavy”，小于零称为“右重right-heavy”平衡因子等于0，则称作平衡。

Balanced Binary Search Tree: Balance Factor

平衡二叉查找树：平衡因子

If the balance factor of each node in a binary search tree is between -1, 0, 1, the binary search tree is called a balanced tree

如果一个二叉查找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



Balanced Binary Search Tree: Definition of AVL Tree

平衡二叉查找树：AVL树的定义

In the process of balancing tree operation, if the balance factor of a node exceeds this range, a rebalancing process is required.

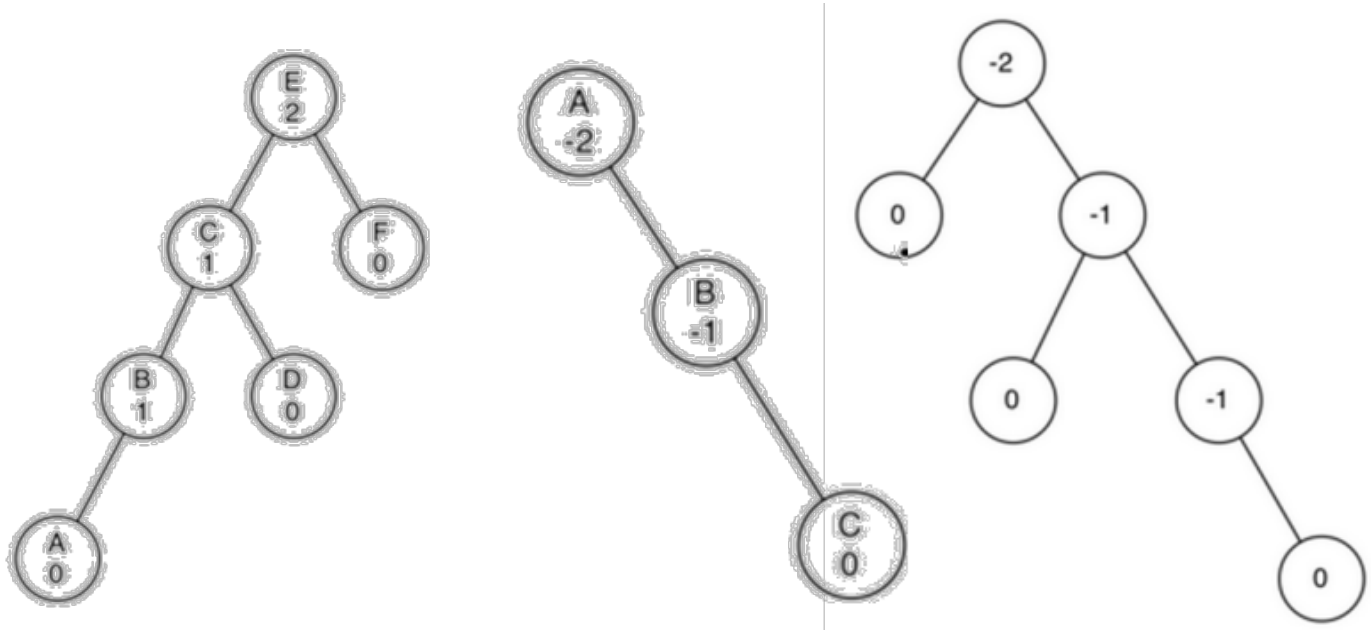
在平衡树操作过程中，有节点的平衡因子超出此范围，则需要一个重新平衡的过程

To maintain the property of BST!

要保持BST的性质！

Think: If rebalanced, what should it become?

思考：如果重新平衡，应该变成什么样？



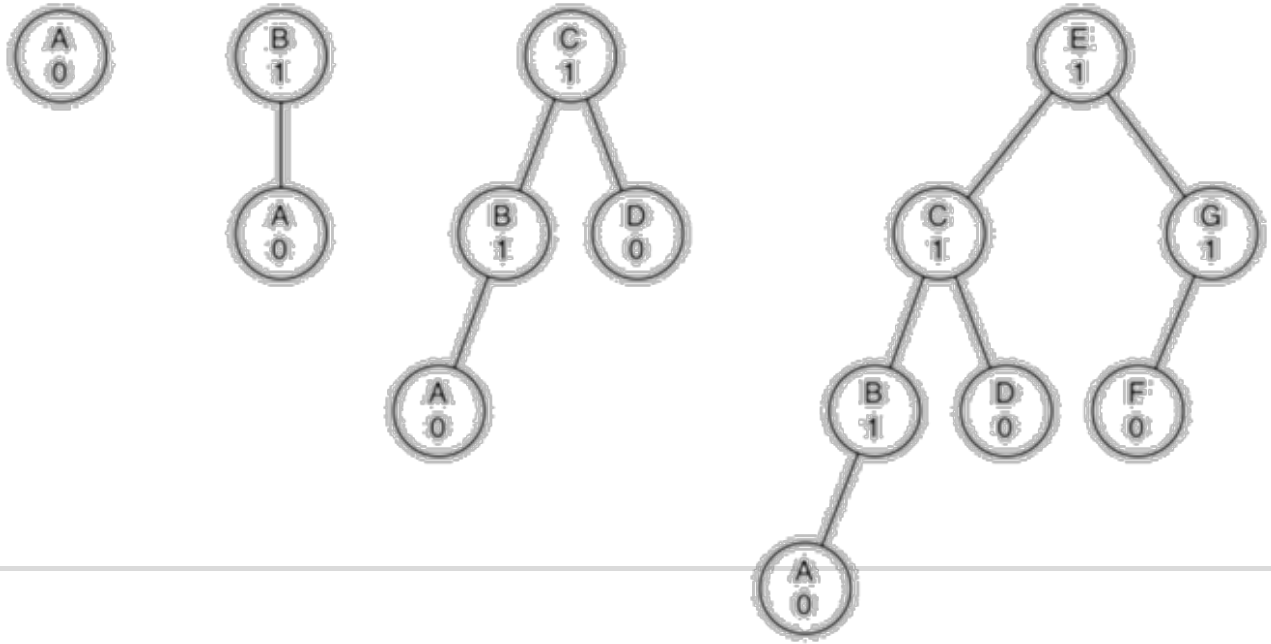
Let's first look at the performance of the AVL tree
先来看看AVL树的性能

Let's analyze the worst-case performance of the AVL tree: that is, the balance factor is 1 or -1

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

The figure below shows a "left-heavy" AVL tree with a balance factor of 1. The height of the tree starts from 1. Let's see how the problem size (the total number of nodes N) and the number of comparisons (the height of the tree, h) are related?

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 tree performance analysis
AVL树性能分析

Observe the change of the total number of nodes N when h=1~4 in the above figure

观察上图h=1~4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

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

Observe this general formula, it is very close to the Fibonacci sequence!

观察这个通式，很接近斐波那契数列！

AVL tree performance analysis
AVL树性能分析

Define the Fibonacci sequence Fi
定义斐波那契数列F_i

Rewrite Nh with Fi

利用F_i重写N_h

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

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

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

The properties of the Fibonacci sequence: Fi/Fi-1 tends to the golden section Φ
斐波那契数列的性质：F_i/F_{i-1}趋向于黄金分割Φ

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

The general formula for Fi can be written
可以写出F_i的通式

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

AVL tree performance analysis

AVL树性能分析

Substitute the general formula of Fi into Nh to get the general formula of Nh

将F_i通式代入到N_h中，得到N_h的通式

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

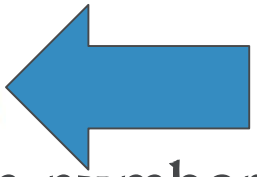
The above general formula only has N and h, we solve h

上述通式只有N和h了，我们解出h

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

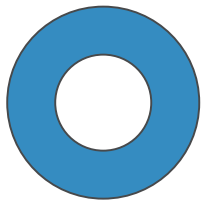
$$h = \frac{\log N_h + 1 - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi}$$

$$h = 1.44 \log N_h$$



The relationship between the maximum number of searches *h* and the size *N*, it can be said that the search time complexity of the AVL tree is O(logn)

最多搜索次数h和规模N的关系，可以说AVL树的搜索时间复杂度为O(logn)



Python implementation of AVL tree

AVL树的Python实现

Python implementation of AVL tree
AVL树的Python实现

Since AVL balanced trees can indeed improve the performance of BST trees and avoid degenerate situations

既然AVL平衡树确实能够改进BST树的性能，避免退化情形

Let's take a look at how to maintain the balanced nature of the AVL tree by inserting a new key into the AVL tree

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

First, as a BST, the new key must be inserted into the AVL tree as a leaf node

首先，作为BST，新key必定以叶节点形式插入到AVL树中

Python implementation of AVL tree
AVL树的Python实现

The balance factor of a leaf node is 0, which itself does not need to be rebalanced, but affects the balance factor of its parent node:

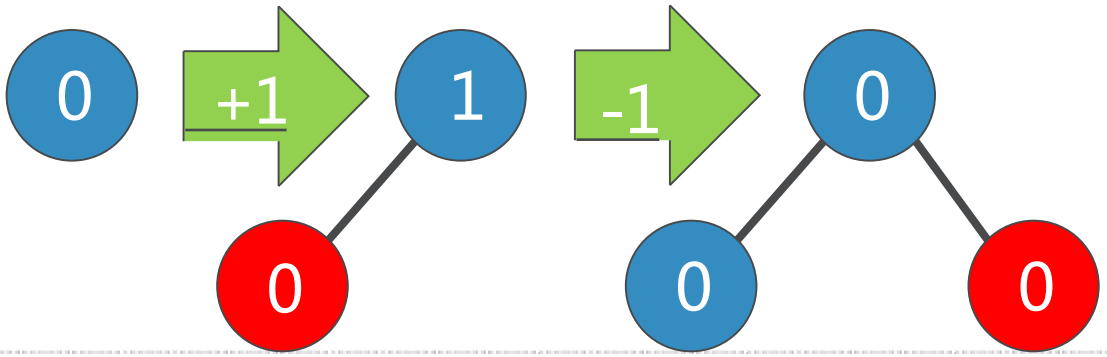
叶节点的平衡因子是0，其本身无需重新平衡，但会影响其父节点的平衡因子：

Inserted as the left child node, the parent node balance factor will increase by 1;

作为左子节点插入，则父节点平衡因子会增加1；

Inserted as a right child node, the parent node balance factor is reduced by 1.

作为右子节点插入，则父节点平衡因子会减少1。



Python implementation of AVL tree
AVL树的Python实现

This effect may be passed along the path from its parent node to the root node, until:

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：

passed to the root node;

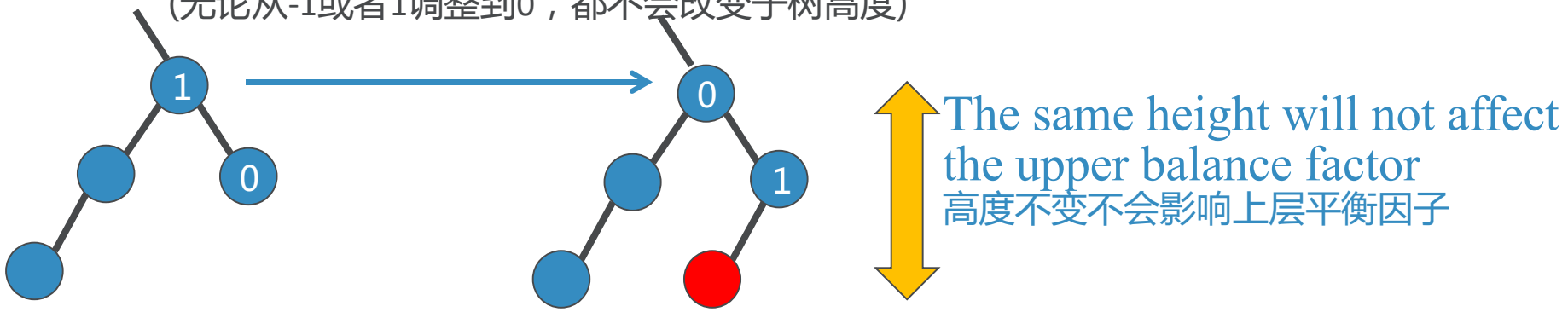
传递到根节点为止；

Or the balance factor of a parent node is adjusted to 0, and the balance factor of the upper node is no longer affected.

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。

- (no matter whether you adjust from -1 or 1 to 0, the subtree height will not be changed)

(无论从-1或者1调整到0，都不会改变子树高度)



Implementation of AVL tree: put method
AVL树的实现：put方法

Redefine the _put method
重新定义_put方法即可

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key,val,currentNode.leftChild)
        else:
            currentNode.leftChild = TreeNode(key,val,parent=currentNode)
            self.updateBalance(currentNode.leftChild)
    else:
        if currentNode.hasRightChild():
            self._put(key,val,currentNode.rightChild)
        else:
            currentNode.rightChild = TreeNode(key,val,parent=currentNode)
            self.updateBalance(currentNode.rightChild)
```

adjustment factor
调整因子

adjustment factor
调整因子

Implementation of AVL tree: UpdateBalance method

AVL树的实现：UpdateBalance方法

```
def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        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)
```

rebalance
重新平衡

Adjust parent node factor
调整父节点因子

Implementation of AVL tree: rebalance

AVL树的实现：重新平衡

Main means: rotate the unbalanced subtree

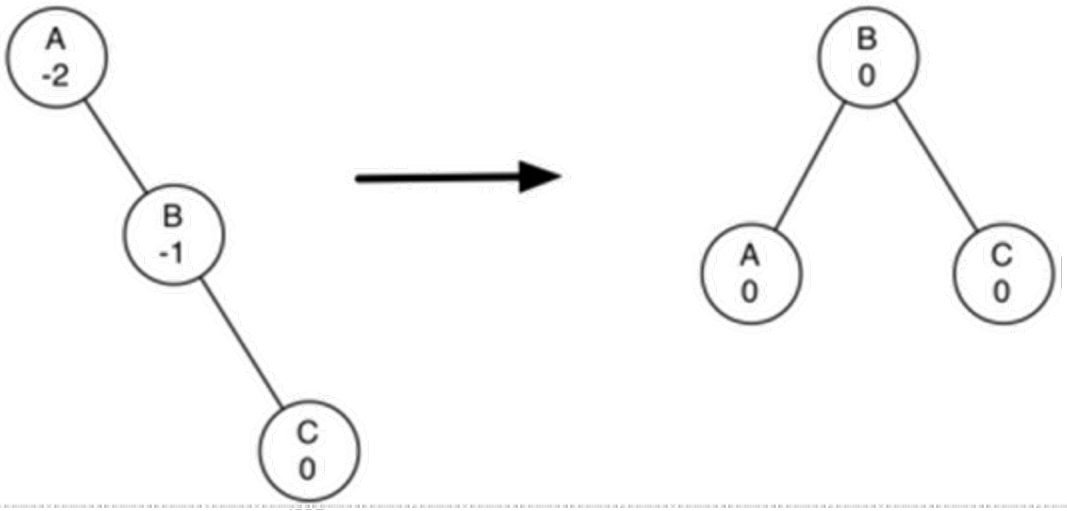
主要手段：将不平衡的子树进行旋转

Rotate in different directions depending on "left weight" or "right weight"

视“左重”或者“右重”进行不同方向的旋转

At the same time, update the reference of the relevant parent node, and update the balance factor of the affected node after rotation.

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



Implementation of AVL tree: rebalance

AVL树的实现：rebalance重新平衡

As shown in the figure, it is a left rotation of a "right-heavy" subtree A (and maintains the BST property)

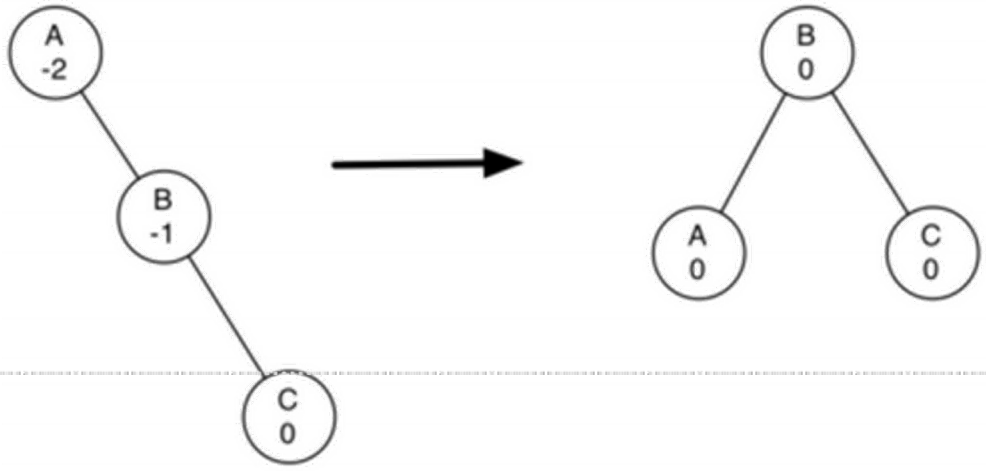
如图，是一个“右重”子树A的左旋转(并保持BST性质)

Promote the right child node B to the root of the subtree, and make the old root node A the left child of the new root node B

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

If the new root node B originally has a left child node, set this node as the right child node of A (the right child node of A must be empty)

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的右子节点一定有空)



Implementation of AVL Trees: Epilogue

AVL树的实现：结语

After the complex *put* method, the AVL tree always maintains a balance, and the get method always maintains O(logn) high performance

经过复杂的put方法，AVL树始终维持平衡，get方法也始终保持O(logn)高性能

Yet, how expensive is the cost of *put* method?

不过，put方法的代价有多大？

Divide the *put* method of the AVL tree into two parts:

将AVL树的put方法分为两个部分：

The new node that needs to be inserted is a leaf node, and the cost of updating all its parents and ancestors is at most O(logn)

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(logn)

If the insertion of a new node causes an imbalance, rebalancing requires at most 2 rotations, but the cost of rotation is constant O(1) independent of the problem size

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)

So the time complexity of the entire *put* method is still O(logn)

所以整个put方法的时间复杂度还是O(logn)

Chapter Summary
本章总结

This chapter introduces the "tree" data structure, and we discuss the following algorithms:

本章介绍了“树”数据结构，我们讨论了如下算法：

①Binary tree for expression parsing and evaluation

用于表达式解析和求值的二叉树

②Binary search tree BST tree for implementing ADTMap

用于实现ADTMap的二叉查找树BST树

③Improved performance for implementing ADTMap's balanced binary search tree AVL tree

改进了性能，用于实现ADTMap的平衡二叉查找树AVL树

Summary of the implementation method of ADTMap
ADTMap的实现方法小结

We use a variety of data structures and algorithms to implement ADTMap, and its time complexity order of magnitude is shown in the following table:

我们采用了多种数据结构和算法来实现ADTMap，其时间复杂度数量级如下表所示：

	ordered list 有序表	hash table 散列表	binary search tree 二叉查找树	AVL tree AVL树
put	O(n)	O(1)->O(n)	$O(\log_2 n)$->O(n)	$O(\log_2 n)$
get	$O(\log_2 n)$	O(1)->O(n)	$O(\log_2 n)$->O(n)	$O(\log_2 n)$
in	$O(\log_2 n)$	O(1)->O(n)	$O(\log_2 n)$->O(n)	$O(\log_2 n)$
del	O(n)	O(1)->O(n)	$O(\log_2 n)$->O(n)	$O(\log_2 n)$