

树与二叉树

- 树是一种数据结构, 比如: 目录结构
- 树是一种可以递归定义的数据结构
- 树是由 n 个节点组成的集合:
 - 如果 $n=0$, 那是一颗空树;
 - 如果 $n>0$, 那存在1个节点作为树的根节点, 其他节点可以分为 m 个集合, 每个集合本身又是一棵树.

树的实例: 模拟文件系统

链式存储

```
class Node:
    def __init__(self, name, type='dir'):
        self.name = name
        self.type = type # 'dir' or 'file'
        self.children = []
        self.paent = None
        # 链式存储

    def __repr__(self):
        return self.name
```

```
class FileSystemTree:
    def __init__(self):
        self.root = Node('/')
        self.now = self.root

    def mkdir(self, name): # 当前目录下创建一个目录
        # name 以/结尾
        if name[-1] != '/':
            name += '/'
        node = Node(name)
        self.now.children.append(node)
        node.parent = self.now

    def ls(self): # 展示当前目录下的所有目录
        return self.now.children

    def cd(self, name): # 切换目录
        # '相对路径下面一层'
        # '../var/python/'
        if name[-1] != '/':
            name += '/'
        if name == '../':
            self.now = self.now.parent
            return
        for child in self.now.children:
            if child.name == name:
                self.now = child # 切换到下面一层
```

```
        return
    raise ValueError('invalid dir') # 否则才报错
```

```
tree = FileSystemTree()
tree.mkdir('var/')
tree.mkdir('bin/')
tree.mkdir('usr/')
tree.root.children
```

```
[var/, bin/, usr/]
```

```
tree.ls()
```

```
[var/, bin/, usr/]
```

```
tree.cd('bin/')
tree.mkdir('python/')
```

```
tree.cd('bin/')
```

二叉树

- 二叉树的链式存储:将二叉树的节点定义为一个对象,节点之间通过类似链表的链接方式来连接.
- 节点定义:

```
class BiTreeNode:
    def __init__(self, data):
        self.data = data
        self.lchild = None # 左孩子
        self.rchild = None # 右孩子
```

```
a = BiTreeNode('A')
b = BiTreeNode('B')
c = BiTreeNode('C')
d = BiTreeNode('D')
e = BiTreeNode('E')
f = BiTreeNode('F')
g = BiTreeNode('G')
```

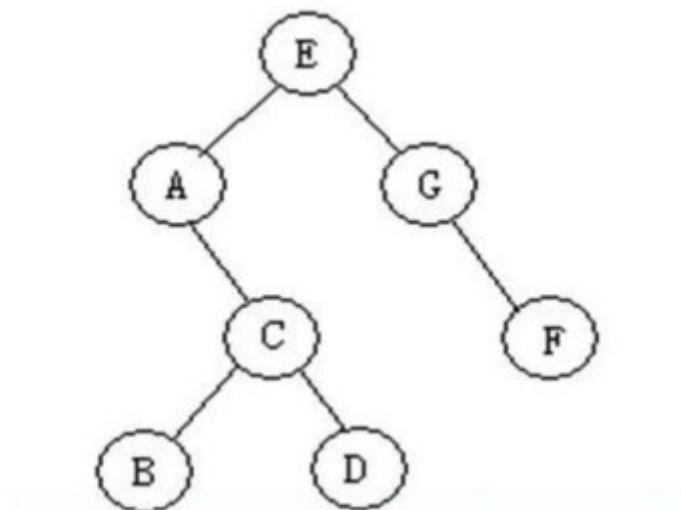
```
e.lchild = a
e.rchild = g
a.rchild = c
c.lchild = b
c.rchild = d
g.rchild = f
```

```
root = e
```

```
root.lchild.rchild.data
```

```
'c'
```

二叉树的遍历



- 二叉树的遍历方式:
 - 前序遍历:EACBDGF
 - 中序遍历:ABCDEGF
 - 后序遍历:BDCAFGE
 - 层次遍历:EAGCFBD
- 前序遍历

```
def pre_order(root): #先左后右
    if root:
        print(root.data, end=',')
        pre_order(root.lchild)
        pre_order(root.rchild)
```

```
pre_order(root)
```

```
E,A,C,B,D,G,F,
```

- 中序遍历

```
def in_order(root):  
    if root:  
        in_order(root.lchild)  
        print(root.data, end=',')  
        in_order(root.rchild)
```

```
in_order(root)
```

```
A,B,C,D,E,G,F,
```

- 后序遍历

```
def post_order(root):  
    if root:  
        post_order(root.lchild)  
        post_order(root.rchild)  
        print(root.data, end=',')
```

- 给出前序遍历和中序遍历，分析出数的结构，并给出后序遍历的结果。

先根据前序遍历确定root，在根据中序遍历**确定左右子树**，再根据前序遍历确定左右子节点，不断反复分析，得出结果。

- 层次遍历

```
from collections import deque
```

```
def level_order(root):  
    queue = deque() # 创建队列  
    queue.append(root)  
    while len(queue) > 0: # 只要队不空  
        node = queue.popleft()  
        print(node.data, end=',')  
        if node.lchild:  
            queue.append(node.lchild)  
        if node.rchild:  
            queue.append(node.rchild)
```

二叉搜索树

- 二叉搜索是一颗二叉树且满足性质：设x是二叉树的一个节点。如果y是x左子树的一个节点，那么 $y.key \leq x.key$ ；如果y是x右子树的一个节点，那么 $y.key \geq x.key$ 。
- 二叉搜索树的操作：查询、插入、删除

二叉搜索树:插入

```
class BiTreeNode:
    def __init__(self, data):
        self.data = data
        self.lchild = None # 左孩子
        self.rchild = None # 右孩子
        self.parent = None
```

```
class BST:
    def __init__(self, li=None):
        self.root = None
        if li:
            for val in li:
                self.insert_no_rec(val)

    def insert(self, node, val): # 相同的元素不插入
        if not node:
            node = BiTreeNode(val)
        elif val < node.data:
            node.lchild = self.insert(node.lchild, val)
            node.lchild.parent = node
        elif val > node.data:
            node.rchild = self.insert(node.rchild, val)
            node.rchild.parent = node
        return node

    def insert_no_rec(self, val):
        p = self.root
        if not p: #空树
            self.root = BiTreeNode(val)
            return
        while True:
            if val < p.data:
                if p.lchild:
                    p = p.lchild # 往左子树走,继续循环
                else: #左孩子不存在
                    p.lchild = BiTreeNode(val)
                    p.lchild.parent = p
                    return
            elif val > p.data:
                if p.rchild:
                    p = p.rchild
                else:
                    p.rchild = BiTreeNode(val)
                    p.rchild.parent = p
                    return
            else:
                return

    def query(self, node, val):
        if not node:
            return None
        if node.data < val:
            return self.query(node.rchild, val)
        elif node.data > val:
```

```

        return self.query(node.lchild, val)
    else:
        return node

def query_no_rec(self, val):
    p = self.root
    while p:
        if p.data < val:
            p = p.rchild
        elif p.data > val:
            p = p.lchild
        else:
            return p
    return None

def pre_order(self, root): #先左后右
    if root:
        print(self.root.data, end=',')
        pre_order(self.root.lchild)
        pre_order(self.root.rchild)

def in_order(self, root): # 二叉树中中序序列一定是升序
    if root:
        in_order(self.root.lchild)
        print(self.root.data, end=',')
        in_order(self.root.rchild)

def post_order(self, root):
    if root:
        post_order(self.root.lchild)
        post_order(self.root.rchild)
        print(self.root.data, end=',')

def __remove_node_1(self, node): #情况1:node是叶子节点
    if not node.parent:
        self.root = None
    if node == node.parent.lchild: #node是他父亲的左孩子
        node.parent.lchild = None
        node.parent = None
    else: # 右孩子
        node.parent.rchild = None

def __remove_node_21(self, node): #情况2.1:node只有一个左孩子
    if not node.parent: # 根节点
        self.root = node.lchild
        node.lchild.parent = None
    elif node == node.parent.lchild:
        node.parent.lchild = node.lchild
        node.lchild.parent = node.parent
    else:
        node.parent.rchild = node.lchild
        node.lchild.parent = node.parent

def __remove_node_22(self, node): #情况2.2:node只有一个右孩子
    if not node.parent:
        self.root = node.rchild
        node.rchild.parent = None
    elif node == node.parent.lchild:

```

```

        node.parent.lchild = node.rchild
        node.rchild.parent = node.parent
    else:
        node.parent.rchild = node.rchild
        node.rchild.parent = node.parent

def delete(self, val):
    if self.root: #不是空树
        node = self.query_no_rec(val)
        if not node: # 不存在
            return False
        if not node.lchild and not node.rchild: # 1. 叶子节点
            self.__remove_node_1(node)
        elif not node.rchild: # 2.1 只有一个左孩子
            self.__remove_node_21(node)
        elif not node.lchild: # 2.2 只有一个右孩子
            self.__remove_node_22(node)
        else: # 3. 两个孩子都有
            min_node = node.rchild # 找右子树最小的node
            while min_node.lchild:
                min_node = min_node.lchild
            node.data = min_node.data #把原来的替换为min_node
            # 删除min_node
            if min_node.rchild:
                self.__remove_node_22(min_node)
            else:
                self.__remove_node_1(min_node)

```

```

tree = BST([4,6,7,9,2,1,3,5,8])
tree.pre_order(tree.root)
print('')
tree.in_order(tree.root)
print('')
tree.post_order(tree.root)

```

```

4,2,1,3,6,5,7,9,8,
1,2,3,4,5,6,7,8,9,
1,3,2,5,8,9,7,6,4,

```

```

import random

```

```

li = list(range(500))
random.shuffle(li)
tree = BST(li)
tree.in_order(tree.root) # 二叉树中中序序列一定是升序

```

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266,267,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,283,284,285,286,287,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,326,327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346,347,348,349,350,351,352,353,354,355,356,357,358,359,360,361,362,363,364,365,366,367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,385,386,387,388,389,390,391,392,393,394,395,396,397,398,399,400,401,402,403,404,405,406,407,408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426,427,428,429,430,431,432,433,434,435,436,437,438,439,440,441,442,443,444,445,446,447,448,449,450,451,452,453,454,455,456,457,458,459,460,461,462,463,464,465,466,467,468,469,470,471,472,473,474,475,476,477,478,479,480,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496,497,498,499,

二叉搜索树:查询

```
tree = BST(li)
tree.query_no_rec(4).data
```

4

```
tree.query_no_rec(3).data
```

3

二叉搜索树:删除

- 1.如果要删除的节点是**叶子节点**:**直接删除**
- 2.如果要删除的节点**只有一个孩子**:**将此节点的父亲与孩子连接,然后删除该节点.**
- 3.如果要删除的节点有**两个孩子**:**将其右子树的最小节点(该节点最多有一个右孩子)删除,并替换当前节点.**


```

tree = BST([1,4,2,5,3,8,6,9,7])
tree.in_order(tree.root)
print('')

tree.delete(4)
tree.delete(1)
tree.in_order(tree.root)

```

```

1,2,3,4,5,6,7,8,9,
2,3,5,6,7,8,9,

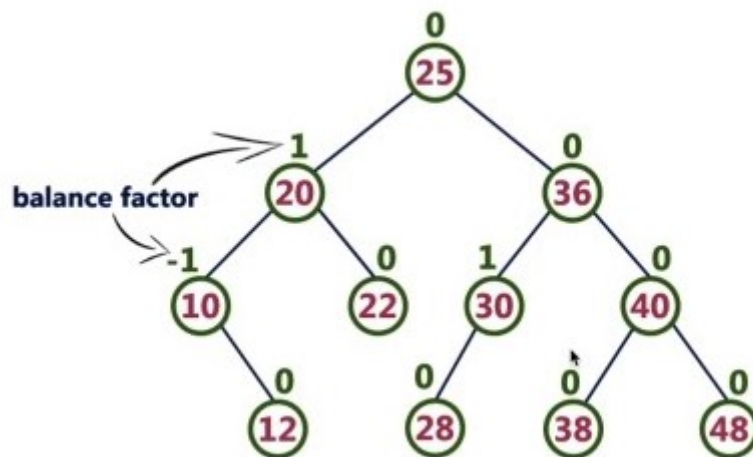
```

二叉搜索树的效率

- 平均情况下,二叉搜索树进行搜索的时间复杂度为 $O(\lg n)$
- 最坏情况下,二叉搜索树可能非常偏斜.
- 解决方案:
 - 随机化插入
 - AVL树

AVL树

- AVL树:是一棵自平衡的二叉搜索树.
- AVL树具有以下性质:
 - 根的左右子树的高度之差的绝对值不能超过1
 - 根的左右子树都是平衡二叉树



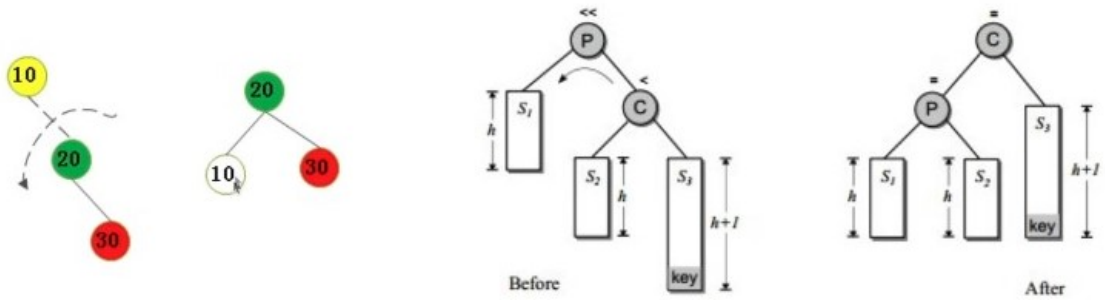
AVL:旋转

AVL树--插入

- 插入一个节点可能会破坏AVL树的平衡,可以通过**旋转**操作来进行修正.
- 插入一个节点后,只有从插入节点到根节点的路径上的节点的平衡可能被改变.我们需要找出**第一个破坏了平衡条件的节点**,称之为K. K的两颗子树的高度差2.
- 不平衡的出现可能有4种情况.

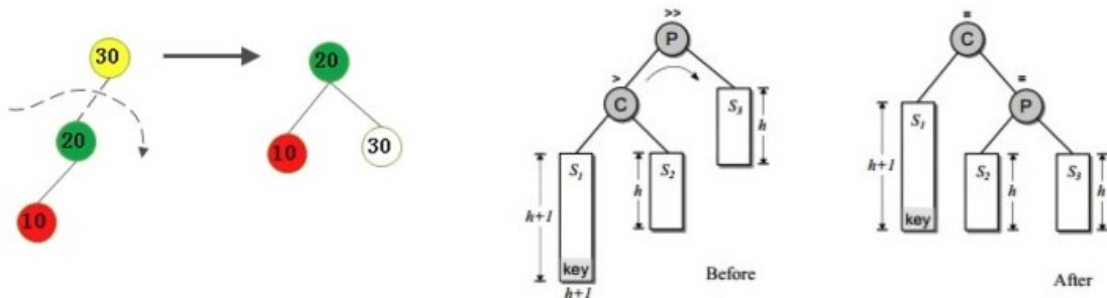
左旋

- 不平衡是由于对K的右孩子的右子树插入导致的：左旋



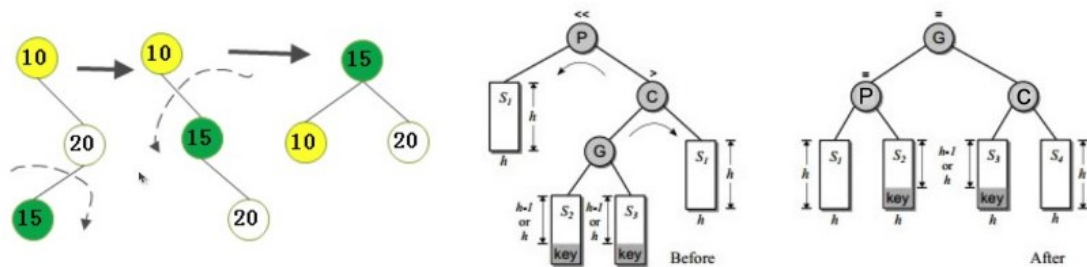
右旋

- 不平衡是由于对K的左孩子的左子树插入导致的：右旋



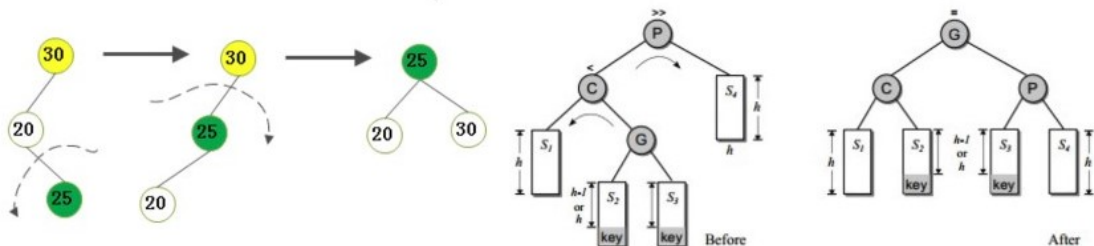
右旋-左旋

- 不平衡是由于对K的右孩子的左子树插入导致的：右旋-左旋



左旋-右旋

- 不平衡是由于对K的左孩子的右子树插入导致的：左旋-右旋



```

class AVLNode(BiTreeNode):
    def __init__(self):
        BiTreeNode.__init__(self, data)
        self.bf = 0 # balance_factor

```

```

class AVLTree(BST):
    def __init__(self, li=None):
        BST.__init__(self, li)

    def rotate_left(self, p, c): # 左旋, 方法不适用于删除
        s2 = c.lchild
        p.rchild = s2
        if s2:
            s2.parent = p

        c.lchild = p
        p.parent = c
        # 更新balance factor
        p.bf = 0
        c.bf = 0
        return c

    def rotate_right(self, p, c): # 右旋
        s2 = r.rchild
        p.lchild = s2
        if s2:
            s2.parent = p

        c.rchild = p
        p.parent = c

        p.bf = 0
        c.bf = 0
        return c

    def rotate_right_left(self, p, c):
        g= c.lchild

        s3 = g.rchild
        c.lchild = s3

        if s3:
            s3.parent = c
            g.rchild = c
            c.paent = g

        s2 = g.lchild
        p.rchild = s2
        if s2:
            s2.parent = p
            g.lchild = p
            p.parent = g

        # 更新bf
        if g.bf > 0:

```

```

        p.bf = -1
        c.bf = 0
    elif g.bf < 0:
        p.bf = 0
        c.bf = 1
    else: # 插入的是g
        p.bf = 0
        c.bf = 0
    g.bf = 0
    return g

def rotate_left_right(self, p, c):
    g = c.rchild

    s2 = g.lchild
    c.rchild = s2

    if s2:
        s2.parent = c
        g.lchild = c
        c.parent = g

    s3 = g.rchild
    p.lchild = s3
    if s3:
        s3.parent = p
        g.rchild = p
        p.parent = g

    # 更新bf
    if g.bf < 0:
        p.bf = 1
        c.bf = 0
    elif g.bf > 0:
        p.bf = 0
        c.bf = -1
    else:
        p.bf = 0
        c.bf = 0
    g.bf = 0
    return g

def insert_no_rec(self, val):
    # 1. 和BST一样, 插入
    p = self.root
    if not p:
        self.root = BiTreeNode(val)
        return
    while True:
        if val < p.data:
            if p.lchild:
                p = p.lchild
            else: # 左孩子不存在
                p.lchild = BiTreeNode(val)
                p.lchild.paent = p
                node = p.lchild # node 存储的是插入的节点
                break
        elif val > p.data:

```

```

        if p.rchild:
            p = p.rchild
        else:
            p.rchild = BiTreeNode(val)
            p.rchild.parent = p
            node = p.rchild
            break
    else: # val == p.data
        return

# 2 更新balance factor
while node.parent: # node.paent不空
    if node.parent.lchild == node: # 传递是从左子树来的。左子树更沉了
        # 更新node.parent的bf -= 1
        if node.parent.bf < 0: # 原来node.parent.bf == -1, 更新后变成-2
            # 做旋转
            # 看node哪边沉
            g = node.parent.parent # 为了连接旋转之后的子树
            x = node.parent # 旋转前子树的根
            if node.bf > 0:
                # 旋转之后子树新的根
                n = self.rotate_left_right(node.parent, node)
            else:
                n = self.rotate_right(node.parent, node)
            # 记得: 把n和g连起来
        elif node.parent.bf > 0: # 原来node.parent.bf = 1, 更新之后变成0
            node.parent.bf = 0
            break
        else: # 原来node.parent.bf = 0, 更新之后变成-1
            node.parent.bf = -1
            node = node.parent
            continue
    else: # 传递是从右子树来的, 右子树更沉了
        # 更新node.parent.bf += 1
        if node.parent.bf > 0: # 原来node.parent.bf == 1, 更新后变成2
            # 做旋转
            # 看node哪边沉
            g = node.parent.parent
            x = node.parent # 旋转前子树的根
            if node.bf < 0: # node.bf = -1
                n = self.rotate_right_left(node.parent, node)
            else: # node.bf = 1
                n = self.rotate_left(node.parent, node)
            # 记得连起来
        elif node.parent.bf < 0: # 原来node.parent.bf == -1, 更新后变成0
            node.parent.bf = 0
            break
        else:
            node.parent.bf = 1
            node = node.parent
            continue

# 连接旋转后的子树
n.parent = g
if g: # g不是空
    if x == g.lchild:
        g.lchild = n # node.parent
    else:

```

```

        g.rchild = n
        break
    else:
        self.root = n
        break

```

```
tree = AVLTree([10,8,7,6,5,4,3,2,1])
```

```

tree.pre_order(tree.root)
print("")
tree.in_order(tree.root)

```

```

10,8,7,6,5,4,3,2,1,
1,2,3,4,5,6,7,8,10,

```

二叉搜索树扩展应用--B树

- B树(B-Tree): B树是一棵自平衡的多路搜索树。常用于数据库的索引。

