数据结构与算法 (Python)

07 树及算法

谢正茂 webg@PKU-Mail

北京大学计算机系

May 18, 2021

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

本章目标

- 理解树数据结构及其应用
- 树用于实现 ADT Map
- 用列表来实现树
- 用类和引用来实现树
- 以递归方式实现树
- 用堆来实现优先队列

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

树的例子

- 在学习了栈、队列等线性数据结构，以及递归算法之后；
- 本章我们来讨论一种基本的“非线性”数据结构——树；
- 树在计算机科学的各个领域中被广泛应用
  - 操作系统、图形学、数据库管理系统、计算机网络
- 跟自然界中的树一样，数据结构树也分为：根、枝和叶等三个部分
- 但在图示中，习惯把树画成倒的：根放在上方，叶放在下方

树的例子：物种分类树

- 首先我们看到的分类树是层次化的
  - 越接近顶部的层越普遍
  - 越接近底部的层越独特
  - 界、门、纲、目、科、属、种
- 分类树的用法：辨认物种
  - 从顶端开始，沿着箭头方向向下
  - 门：脊索动物还是节肢动物？
  - 纲：哺乳动物么？
  - 目：食肉动物？
  - 科：猫科？
  - 属：猫属？
  - 种：家猫！

树的例子：物种分类树

- 分类树的第二个特征：一个节点的子节点与另一个节点的子节点相互之间是隔离、独立的
  - 猫属 Felis 和蝇属 Musca 下面都有 Domestica 的同名节点，但相互之间并无任何关联，可以修改其中一个 Domestica 而不影响另一个。
- 分类树的第三个特征：每一个叶节点都具有唯一性
  - 可以用从根开始到达每个种的完全路径来唯一标识每个物种
  - 动物界-> 脊索门-> 哺乳纲-> 食肉目-> 猫科-> 猫属-> 家猫种
  - Animalia->Chordate->Mammal->Carnivora->Felidae->Felis->Domestica

树的例子：Linux 文件系统

树的例子：Windows 文件系统

树的例子：HTML 文档（嵌套标记）

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

树的例子：域名体系

ICANN
.com
.cn
.name
...
google
apple
facebook
edu
chenbin
www
mail
pku
tsinghua
ruc
www
www
its
mail

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

术语与定义

- 节点 Node：组成树的基本部分
  - 每个节点具有名称，或“键值”，节点还可以保存额外数据项，数据项根据不同的应用而变
- 边 Edge：边是组成树的另一个基本部分
  - 每条边恰好连接两个节点，表示节点之间具有关联，边具有出入方向；
  - 每个节点（除根节点）恰有一条来自另一节点的入边；
  - 每个节点可以有零条/一条/多条连到其它节点的出边。
    - 如果加限制不能有“多条边”，这里树结构就特殊化为线性表
- 根 Root：树中唯一一个没有入边的节点
- 路径 Path：由边依次连接在一起的节点的有序列表
  - 如：哺乳纲 → 食肉目 → 猫科 → 猫属，是一条路径

术语与定义

- 子节点 Children：入边均来自于同一个节点的若干节点，称为这个节点的子节点
- 父节点 Parent：一个节点是其所有出边所连接节点的父节点
- 兄弟节点 Sibling：具有同一个父节点的节点之间称为兄弟节点
- 子树 Subtree：一个节点和其所有子孙节点，以及相关边的集合

术语与定义

- 叶节点 Leaf Node：没有子节点的节点称为叶节点
- 层级 Level：从根节点开始到达一个节点的路径，所包含的边的数量，称为这个节点的层级。
  - 如 D 的层级为 2，根节点的层级为 0
- 高度 Height：树中所有节点的最大层级称为树的高度
  - 如右图树的高度为 2

术语与定义：树的定义 1

- 树由若干节点，以及两两连接节点的边组成，并具有如下性质：
  - 其中一个节点被设定为根；
  - 每个节点 n(除根节点)，都恰连接一条来自节点 p 的边，p 是 n 的父节点；
  - 每个节点从根开始的路径是唯一的
  - 如果每个节点最多有两个子节点，这样的树称为“二叉树 binary tree”后面讨论“树”主要是指“二叉树”。

术语与定义：树的定义 2（递归定义）

- 树 “约定” 为：
  - 空集合（实现为 []，而不是 None）称为空树；
  - 或者由根节点及 0 或多个子树构成（其中子树也是树），每个 “非空” 子树的根到根节点具有边相连。
- 递归定义在下面的嵌套列表法中马上用到

实现二叉树：嵌套列表法

- 二叉树是一种基本而重要的树，我们用它为例来实现树。
- 首先我们尝试用 Python List 来实现二叉树树数据结构；
- 递归的嵌套列表实现二叉树，由具有 3 个元素的列表实现：
  - 第 1 个元素为根节点的值；
  - 第 2 个元素是左子树（所以第 2 个元素是一个列表）；
  - 第 3 个元素是右子树（同理，它也是一个列表）。
- 以右图的示例，一个 6 节点的二叉树
  - 根是 myTree[0]，左子树 myTree[1]，右子树 myTree[2]
- 嵌套列表法的优点
  - 子树的结构与树相同，是一种递归数据结构

```
myTree = ['a',   #root
      ['b',  #left subtree
       ['d' [], []],
       ['e' [], []] ],
      ['c',  #right subtree
       ['f' [], []],
       [] ]
     ]
```

实现二叉树：嵌套列表法

- 我们通过定义一系列函数来辅助操作嵌套列表
  - BinaryTree 创建仅有根节点的二叉树
  - insertLeft/insertRight 将新节点插入树中作为 root 直接的左/右子节点，原来的左/右子节点变为新节点的左/右子节点。为什么？不为什么，一种实现方式而已。
  - get/setRootVal 则取得或返回根节点
  - getLeft/RightChild 返回左/右子树
- 没有为 BinaryTree 设计 class 来实现

```
def BinaryTree(r, left=[], right=[]):
    return [r, left, right]

def insertLeft(root, newBranch):
    root[1] = BinaryTree(newBranch, \
            left=getLeftChild(root))
    return root

def insertRight(root, newBranch):
    root[2] = BinaryTree(newBranch, \
            right=getRightChild(root))
    return root

def getRootVal(root):
    return root[0]

def setRootVal(root, newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

实现二叉树：嵌套列表法

- 请画出 r 的图示
- 请通过图示讲解操作

```python

if __name__=="__main__" :
    r = BinaryTree(3)
    insertLeft(r, 4)
    insertLeft(r, 5)
    insertRight(r, 6)
    insertRight(r, 7)
    l = getLeftChild(r)
    print(l)

    setRootVal(l, 9)
    print(r)
    insertLeft(l, 11)
    print(r)
    print(getRightChild(getRightChild(r)))

```

```
Mini2-1:dsa2020 zhengmaoxie$ python3 code/binaryTree.py
[5, [4, [], []], []]
[3, [9, [4, [], []], []], [7, [], [6, [], []]]]
[3, [9, [11, [4, [], []], []], []], [7, [], [6, [], []]]]
[6, [], []]
```

实现树：节点链接法

- 同样可以用类似链表的节点链接法来实现树
  - 每个节点保存根节点的数据项，以及指向左右子树的链接
- 定义一个 BinaryTree 类
  - 成员 key 保存根节点数据项
  - 成员 left/rightChild 则保存指向左/右子树的引用（同样是 BinaryTree 对象）

```
class BinaryTree:
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

实现二叉树：节点链接法

- insertLeft/Right 方法，实现逻辑与嵌套列表法相同。
- 请画出 r 的图示

```python
r = BinaryTree('a')
r.insertLeft('b')
r.insertRight('c')
r.getRightChild().setRootVal('hello')
r.getLeftChild().insertRight('d')
```

```python
    def insertLeft(self, newNode):
        self.leftChild = BinaryTree(newNode, \
                left=self.leftChild)

    def insertRight(self, newNode):
        self.rightChild = BinaryTree(newNode, \
                right = self.rightChild)

    def getRightChild(self):
        return self.rightChild;

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self):
        return self.key
```

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
  - 解析树
  - 哈夫曼树
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

二叉树的应用：解析树 Parse Tree（语法树）

- 将树用于表示语言中句子，可以分析句子的各种语法成分，对句子的各种成分进行处理
- 语法分析树
  - 主谓宾，定状补
- 程序设计语言的编译
  - 词法、语法检查
  - 从语法树生成目标代码
- 自然语言处理
  - 机器翻译、语义理解

树的应用：解析树 Parse Tree（表达式树）

- 表达式的几种形式：前缀、中缀、后缀、全括号。
- 我们还可以将表达式表示为树结构
  - 叶节点保存操作数，内部节点保存操作符
- 全括号表达式 ((7+3)*(5-2))
  - 由于括号的存在，需要计算 * 的话，就必须先计算 7+3 和 5-2
  - 表达式树的层次帮助我们了解表达式计算的优先级
  - 越底层的表达式，优先级越高
- 树中每个子树都表示一个子表达式
  - 将子树替换为子表达式值的节点，即可实现求值
- 得到了表达式树之后，生成它的前缀/后缀/中缀表示都异常简单

Figure 2: Parse Tree for $((7+3)*(5-2))$

二叉树的应用：表达式树

- 下面，我们用树结构来做如下尝试
  - 从全括号表达式构建表达式解析树
  - 利用表达式解析树对表达式求值
  - 从表达式解析树恢复原表达式的字符串形式
- 首先，全括号表达式要分解为单词 Token 列表
  - 其单词分为括号“()”、操作符“+-*/”和操作数“0 ∼ 9”这几类
  - 左括号就是表达式的开始，而右括号是表达式的结束

二叉树的应用：表达式树算法思路

- 一棵表达式树的根、左子树、右子树分别对应它最后或最上一个操作的操作符、左操作数/子树和右操作数/子树。
- 从左到右扫描全括号表达式的每个单词，依据规则建立解析树
  - 如果当前单词是"("：为当前节点添加一个新节点作为其左子节点，当前节点下降为这个新节点，准备接收左操作树/子树。
  - 如果当前单词是操作符 ['+','-','/','*']：将当前节点的值设为此符号，为当前节点添加一个新节点作为其右子节点，当前节点下降为这个新节点，准备接收右操作数/子树。
  - 如果当前单词是操作数，则接收操作数：将当前节点的值设为此数，当前节点上升返回到父节点
  - 如果当前单词是")"：则当前节点上升返回到父节点

从全括号表达式建立表达式解析树：图示

- 全括号表达式：(3+(4*5))
  - 分解为单词表 ['(', '3', '+', '(', '4', '*', '5' ,')',')']
- 创建表达式解析树过程
  - 创建空树，当前节点为根节点
  - 读入'('，创建了左子节点，当前节点左下降
  - 读入'3'，当前节点设置为 3，上升到父节点
  - 读入'+'，当前节点设置为 +，创建右子节点，当前节点右下降

从全括号表达式建立表达式解析树：图示

- 创建表达式解析树过程
  - 读入'('，创建左子节点，当前节点左下降
  - 读入'4'，当前节点设置为 4，上升到父节点
  - 读入'*'，当前节点设置为 *，创建右子节点，当前节点右下降
  - 读入'5'，当前节点设置为 5，上升到父节点
  - 读入')'，上升到父节点。
- 细心的同学看到这里会有疑问 ……??

从全括号表达式建立表达式解析树：思路

- 从图示过程中我们看到，创建树过程中关键的是对当前节点的跟踪
  - 当前节点创建左右子树，可以调用 BinaryTree.insertLeft/Right
  - 当前节点设置值，可以调用 BinaryTree.setRootVal
  - 当前节点下降到左右子树，可以调用 BinaryTree.getLeft/RightChild
  - 但是，当前节点上升到父节点，这个没有方法支持！
- 我们可以用一个栈来记录跟踪父节点
  - 当前节点下降时，将下降前的节点 push 入栈
  - 当前节点需要上升到父节点时，上升到 pop 出栈的节点即可！

从全括号表达式建立表达式解析树：代码

```
def buildParseTree(fpexp):
    fplist = fpexp.split()                      #分解单词：输入表达式中单词需用空格分开
    pStack = []                                 #python list是多面手，可以把它当栈用。
    currentTree = eTree = BinaryTree('')        #创建一棵树空树
    pStack.append(currentTree)                  #这是在干什么？？？？
    for i in fplist:
        if i=="(":                              #子表达式开始
            currentTree.insertLeft('')
            pStack.append(currentTree)          #入栈左下降
            currentTree = currentTree.getLeftChild()
        elif i in ['+', '-', '*', '/']:         #操作符
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.append(currentTree)          #入栈右下降
            currentTree = currentTree.getRightChild()
        elif i == ')':                          #子表达式结束
            currentTree = pStack.pop()
        elif ord('0')<= ord(i)<= ord('9'): #操作数
            currentTree.setRootVal(int(i))
            currentTree = pStack.pop()          #出栈上升
        else:
            raise ValueError
    return eTree
```

从全括号表达式建立表达式解析树：栈深度的变化

单词	(	3	+	(	4	*	5	)	)
栈深度	1	0	1	2	1	2	1	0	-1?

利用表达式解析树求值：思路

- 创建了表达式解析树之后，我们可以用来进行表达式求值
- 由于 BinaryTree 是一个递归数据结构，自然地，可以用递归算法来处理 BinaryTree，包括求值函数 evaluate
  - 由前述对子表达式的描述，可以从树的底层子树开始，逐步向上层求值，最终得到整个表达式的值
- 求值函数 evaluate 的递归三要素：
  - 基本结束条件：叶节点是最简单的子树，没有左右子节点，其根节点的数据项即为子表达式树的值
  - 缩小规模：将表达式树分为左子树、右子树，即为缩小规模
  - 调用自身：分别调用 evaluate 计算左子树和右子树的值，其基本操作是将左右子树的值依根节点的操作符进行计算，从而得到表达式的值

利用表达式解析树求值：思路

- 一个增加程序可读性的技巧：函数引用
  - import operator
  - op = operator.add

```
>>> import operator
>>> operator.add
<built-in function add>
>>> operator.add(1,2)
3
>>> op= operator.add
>>> n= op(1,2)
>>> n
3
```

利用表达式解析树求值：代码

```
import operator
def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, \
             '*':operator.mul, '/':operator.truediv}

    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()

    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()
```

缩小规模

递归调用

基本结束条件

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
  - 解析树
  - 哈夫曼树
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

字符的编码问题：定长编码

- 所有字符编码成等长的二进制 0/1 序列
- 下图中“林妹妹哭哭哭哭哭哭了”编码后的串长度为 20bits
- 现实中的 unicode 采用的就是这种方式–每个字符都编码成 4bytes(32bits)
  - ASCII、中文字符，考虑了世界上绝大多数语言的字符
  - 优点：所有字符采用统一的编码长度，处理比较方便。
  - 缺点：空间太浪费了，ASCII 1byte $\Rightarrow$ 4bytes

林:00，妹:01，哭:10，了:11

字符串	林	妹	妹	哭	哭	哭	哭	哭	哭	了
编码	00	01	01	10	10	10	10	10	10	11

字符的编码问题：不定长编码

- 字符编码成不等长的 0/1 序列
- “林妹妹哭哭哭哭哭哭了” 编码长度为 16bits
- 诀窍：让出现的最多的字符使用最短的编码
  - 林:1，妹:2，哭:6，了:1

林:110，妹:10，哭:0，了:111

字符串	林	妹	妹	哭	哭	哭	哭	哭	哭	了
编码	110	10	10	0	0	0	0	0	0	111

哈夫曼 Haffman 编码树

- 所有的 m 个编码字符放在树的叶节点上
- 左子边标为 0；右子边标为 1
- 从根到叶节点路径上的标记连起来就是字符对应的编码
- 叶节点中的数字 w_i 表示字符的频率；
- 叶节点的层级为 l_i，它同时也是编码字符 i 使用的比特数
- 中间节点的数字表示子树中所有叶节点的频率和
- 编码后文本的总长度为 W，目标是使它最小

$$W = \sum_{i=1}^{m} w_i l_i$$

哈夫曼编码思路

- 让常见的字符使用最短的编码;
- 反过来，让罕见的字符使用最长的编码，处在树的最底端
- 贪心法
  - 从集合中取出 w 最小的两个节点组成树;
  - 根节点的 w 为它们 w 的和;
  - 把根节点放回集合
- 需要从集合中取最小值，我们用优先队列实现集合
- 使用嵌套列表法表示二叉树，elem[0] 存放 (字符, 权重)，中间节点的字符为 None

哈夫曼建树代码

- 建立哈夫曼编码树

```python
from typing import List, Tuple
def haffmanTree(alist: List[Tuple[str, int]]):
    pq = BinHeap(key=lambda x:x[0][1])
    pq.buildHeap([[i,[],[]]for i in alist])        #以字符元素为根，左右子树为空集
    while pq.currentSize >= 2 :
        left = pq.delMin()                          #取出两个最小的元素，
        right = pq.delMin()                         #组成一个新的元素插入优先队列
        pq.insert([(None, left[0][1]+right[0][1]), left, right])
    else:
        return pq.delMin()                          #返回唯一的元素
```

每一个循环都减少一个元素

- 输出哈夫曼编码

```python
def _printCode(haff, path):
    if haff[1]==[] and haff[2]==[]:
        #字符所在的叶节点
        print(haff[0], ''.join(path))
    else: #修改path，对左右子树进行递归调用
        _printCode(haff[1], path+['0'])
        _printCode(haff[2], path+['1'])
```

```python
def printCode(haff):
    curr = haff
    path = []    #用一个栈来保存路径
    _printCode(curr, path)
```

W
484
174
90
43
19
310
0
0
75
的
6485
维
2330
损
1084
厅
558
氖
160
聘
160
杆
84
赍
47
铿
24
相
13
珙
6
吡
111 001

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树的遍历
- 二叉堆实现的优先队列
- 二叉搜索树

树的遍历 Tree Traversals

- 对某一数据结构中的所有元素逐个访问的操作称为“遍历 Traversal”
- 线性数据结构中，遍历操作比较简单直接，
- 树的非线性特点，使得遍历操作较为复杂，我们按照对节点访问次序的不同来区分 3 种遍历
  - 前序遍历（preorder）：先访问根节点，再递归地前序访问左子树、最后前序访问右子树；
  - 中序遍历（inorder）：先递归地中序访问左子树，再访问根节点，最后中序访问右子树；
  - 后序遍历（postorder）：先递归地后序访问左子树，再后序访问右子树，最后访问根节点。
  - “前”、“中”、“后”，都是针对根节点来说的。

前序遍历的例子：一本书的章节阅读

- Book→ Ch1→ S1.1→ S1.2→ S1.2.1→ S1.2.2→
- Ch2→ S2.1→ S2.2→ S2.2.1→ S2.2.2

Figure 5: Representing a Book as a Tree

树的遍历：递归算法代码

- 树遍历的代码非常简洁！

```
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

- 也可以在 BinaryTree 类中实现前序遍历的方法：
  - 需要加入子树是否为空的判断

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()
```

- 后序遍历和中序遍历的代码仅需要调整语句顺序：

```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

```
def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())
```

后序遍历：表达式求值

- 回顾前述的表达式解析树求值，实际上也是一个后序遍历的过程，只是代码没有明显表现出遍历的次序
- 采用后序遍历法重写表达式求值代码：

```
import operator
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub, \
             '*':operator.mul, '/':operator.truediv}
    if tree:
        left = postordereval(tree.getLeftChild())          #左子树
        right = postordereval(tree.getRightChild())        #右子树
        if left and right:
            return opers[tree.getRootVal()](left, right)   #计算根节点
        else:
            return tree.getRootVal()                       #直接返回根节点
    else:
        return None                                        #空集返回None

```

中序遍历：生成全括号中缀表达式

- 采用中序遍历递归算法来生成全括号中缀表达式
  - 下列代码中对每个数字也加了括号，请自行修改代码去除（课后练习）

```
def printexp(tree):
  sVal = ""
  if tree:
      sVal = '(' + printexp(tree.getLeftChild())
      sVal = sVal + str(tree.getRootVal())
      sVal = sVal + printexp(tree.getRightChild())+')'
  return sVal
```

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

优先队列 Priority Queue

- 在第 3 章我们学习了一种 FIFO 数据结构队列 queue
- 队列有一种变体称为“优先队列”。
  - 银行窗口取号，VIP 客户优先级高，可以插到队首
  - 操作系统中执行关键任务的进程或用户特别指定进程在调度队列中靠前
- 排队的考虑因素是某种优先级，而不是时间先后。
- 既然是“队列”，自然会问它是不是稳定的：就是说在相同的优先级下，能不能保证 FIFO？

优先队列 Priority Queue

- 优先队列的出队 dequeue 操作跟队列一样，都是从队首出队；
- 但在优先队列内部，数据项的次序却是由“优先级”来确定：高优先级的数据项排在队首，而低优先级的数据项则排在后面。
  - 这样，优先队列的入队操作就比较复杂，需要将数据项根据其优先级尽量挤到队列前方。
- 思考：用已经学过的线性表如何实现优先队列？
  - 出队和入队的复杂度大概是多少？

二叉堆 Binary Heap 实现优先队列

- 实现优先队列的经典方案是采用二叉堆数据结构
  - 二叉堆能够将优先队列的入队和出队复杂度都保持在 O(log n)
- “堆”：想象成一团具有流动性的物质：轻的浮上去；重的沉下来。
- 二叉堆的有趣之处在于，其逻辑结构上象二叉树，却是用“非”嵌套列表来实现的！
- 最小 key 排在队首的称为“最小堆 min heap”
  - 反之，最大 key 排在队首的是“最大堆 max heap”

- ADT BinaryHeap 的操作定义如下：
  - BinaryHeap()：创建一个空二叉堆对象；
  - insert(k)：将新 key 加入到堆中；
  - findMin()：返回堆中的最小项，最小项仍保留在堆中；
  - delMin()：返回堆中的最小项，同时从堆中删除；
  - isEmpty()：返回堆是否为空；
  - size()：返回堆中 key 的个数；
  - buildHeap(list)：从一个 key 列表创建新堆

ADT BinaryHeap 的操作示例

```python
from pythonds.trees.binheap import BinHeap

bh = BinHeap()
bh.insert(5)
bh.insert(7)
bh.insert(3)
bh.insert(11)

print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
print(bh.delMin())
```

```
>>> ========
>>>
3
5
7
11
>>>
```

用非嵌套列表实现二叉堆

- 为了使堆操作能保持在对数水平上，就必须采用二叉树结构；
- 同样，如果要使操作始终保持在对数数量级上，就必须始终保持二叉树的“平衡”——树根左右子树拥有相同数量的节点
- 我们采用“完全二叉树”的结构来近似实现“平衡”
  - 完全二叉树，指每个内部节点都有两个子节点，最多可有 1 个节点例外

完全二叉树的列表实现及性质

- 完全二叉树由于其特殊性，可以用非嵌套列表，以简单的方式实现，并具有很好的性质：
  - 如果节点在列表中的下标为 p，那么其左子节点下标为 2p，右子节点为 2p+1
  - 如果节点在列表中下标为 n，那么其父节点下标为 n//2

堆次序 Heap Order

- 所谓“堆”的特性，是指堆中任何一个节点 x，其父节点 p 中的 key 均小于等于 x 中的 key。
  - 这样，符合“堆”性质的二叉树，其中任何一条路径，均是一个已排序数列
  - 符合“堆”性质的二叉树，其树根节点中的 key 最小

二叉堆操作的实现

- 二叉堆初始化
  - 采用一个列表来保存堆数据，其中表首下标为 0 的项无用（浪费一点空间），但为了后面代码可以用到简单的整数乘除法，仍保留它。
- insert(key) 方法
  - 首先，为了保持“完全二叉树”的性质，新 key 应该添加到列表末尾
  - 问题?

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
```

二叉堆操作的实现

- insert(key) 方法
  - 但是，新 key 简单地加在列表末尾，显然无法保持“堆”次序
  - 虽然对其它路径的次序没有影响，但对于其到根的路径可能破坏次序
  - 于是需要将新 key 沿着路径来“上浮”到其正确位置
  - 注意：新 key 的“上浮”不会影响其它路径节点的“堆”次序，上浮路径上的节点值只会变小

二叉堆操作的实现：insert 代码

与父节点交换

沿路径向上

```
def percUp(self,i):
    while i // 2 > 0:
        if self.heapList[i] < self.heapList[i//2]:
            tmp = self.heapList[i // 2]
            self.heapList[i // 2] = self.heapList[i]
            self.heapList[i] = tmp
        i = i // 2
```

添加到末尾

新key上浮

```
def insert(self,k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

二叉堆操作的实现：delMin

- delMin() 方法
  - 首先，移走整个堆中最小的 key——位于堆首位的根节点 heapList[1]
  - 为了保持“完全二叉树”的性质，用最后一个节点来代替根节点
  - 问题?

二叉堆操作的实现：delMin

- delMin() 方法
  - 同样，这么简单的替换，还是破坏了“堆”次序
  - 解决方法：将新的根节点沿着一条路径“下沉”，直到比两个子节点都小
  - “下沉”路径的选择：如果比子节点大，那么选择较小的子节点交换下沉

二叉堆操作的实现：delMin 代码

交换下沉

沿路径向下

```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc
```

唯一子节点

返回较小的

```
def minChild(self,i):
    if i * 2 + 1 > self.currentSize:
        return i * 2
    else:
        if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
            return i * 2
        else:
            return i * 2 + 1
```

移走堆顶

新顶下沉

```
def delMin(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.currentSize]
    self.currentSize = self.currentSize - 1
    self.heapList.pop()
    self.percDown(1)
    return retval
```

二叉堆操作的实现：建堆

- buildHeap(lst) 方法：从无序表生成“堆”
  - 我们最自然的想法是：用 insert(key) 方法，将无序表中的数据项逐个 insert 到堆中，但这么做的总代价是 O(nlog n)
  - 其实，用“下沉”法，能够将总代价控制在 O(n)
    - 叶节点不需调用 percDown()
    - 不同节点调用 percDown() 所执行的比较次数也不一样，与节点离叶节点的距离有关
    - 1/2 的节点比一次，1/4 的节点比 2 次，...

从最后节点的父节点开始
因叶节点无需下沉

```python
def buildHeap(self,alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    print(len(self.heapList), i)
    while (i > 0):
        print(self.heapList, i)
        self.percDown(i)
        i = i - 1
    print(self.heapList,i)
```

二叉堆建堆操作的算法复杂度证明

- 为了便于讨论，我们定义树中节点的高度 height 为该节点到叶节点的最大路径长度。那么，
  - 叶节点的高度为零
  - 根节点的高度等于树的高度
- 在 percDown(self,i) 下沉 i 节点的过程中，执行的比较次数

$$T_{percDown}(i) \leq 3 * height(i)$$

- 假设完全二叉树的节点个数为 n，那么整个建堆过程的比较次数

$$T(n) = \sum_{1 \leq i \leq n} T_{percDown}(i) \leq \sum 3 * height(i)$$

- 爸爸节点 $_{height \geq 1}$ 的个数为 n//2，爷爷节点 $_{height \geq 2}$ 的个数为 n//4，太爷爷节 $_{height \geq 3}$ 点的个数为 n//8...
- 高度恰好为 k 的节点的个数为 $n//2^k - n//2^{k+1}$

二叉堆建堆操作的算法复杂度证明 Continue

$$
\begin{aligned}
T(n) \leq & 3 * \sum_{k \geq 1} k * (n//2^k - n//2^{k+1}) \\
= & 3 * (1 * n//2^1 + 2 * n//2^2 + 3 * n//2^3 + 4 * n//2^4 \ldots) \\
& - 3 * (1 * n//2^2 + 2 * n//2^3 + 3 * n//2^4 + 4 * n//2^5 \ldots) \\
= & 3 * (n//2^1 + n//2^2 + n//2^3 + n//2^4 + \ldots) \\
\leq & 3 * (n/2^1 + n/2^2 + n/2^3 + n/2^4 + \ldots) \\
= & 3 * n \\
= & O(n)
\end{aligned}
$$

堆排序算法思路

- 排序的目标是一个线性结构，“堆” 只是一个树形结构。
- 先建堆，然后在堆的基础上进行排序。
- 思考：利用二叉堆来进行排序的具体实现？
  - “堆排序” 算法复杂度：O(nlog n)

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树

二叉搜索树 (Binary Search Tree)

- 二叉搜索树也属于二叉树的应用之一，因为太重要/内容太多，所以单独做为一节来讲。
- 数据的管理是计算机学科的核心问题之一，尤其关心数据的 “随机” 访问效率。
- 一般都会通过索引的方式，通过 key->data 的方式来进行数据访问，这就是 ADT Map 的主要功能。
- 在 ADT Map 的具体实现方案中，可以采用不同的数据结构和搜索算法来保存和查找 Key，前面已经实现了两个方案
  - 有序表数据结构 + 二分搜索算法
  - 散列表数据结构 + 散列及冲突解决算法
- 下面我们来试试用二叉搜索树保存 key，实现 key 的快速搜索以及完整的 ADT Map

二叉搜索树 Binary Search Tree：ADT Map

- 复习一下 ADT Map 的操作：
  - Map()：创建一个空映射
  - put(key, val)：将 key-val 关联对加入映射中，如果 key 已经存在，则将 val 替换原来的旧关联值；
  - get(key)：给定 key，返回关联的数据值，如不存在，则返回 None；
  - del：通过 del map[key] 的语句形式删除 key-val 关联；
  - len()：返回映射中 key-val 关联的数目；
  - in：通过 key in map 的语句形式，返回 key 是否存在于关联中，布尔值
- 复习一下树的实现形式：
  - 嵌套列表法
  - 节点链接法
  - 非嵌套列表法（只适用于完全二叉树）

二叉搜索树 (BST) 的性质

- 二叉搜索树的每个节点上都存储了 key
- 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。
- 右图是一个简单的 BST，按照 70,31,93,94,14,23,73 的顺序插入
- 画图的时候，都只画了 key，省略了 value

- 首先插入的 70 成为树根
  - 31 比 70 小，放到左子节点
  - 93 比 70 大，放到右子节点
  - 94 比 93 大，放到右子节点
  - 14 比 31 小，放到左子节点
  - 23 比 14 大，放到其右
  - 73 比 93 小，放到其左

- 把二叉搜索树 “拍扁”，就能得到一个有序列表
  - flat = lambda x: flat(x.left)+[x]+flat(x.right) if x else []
- 注意：插入顺序不同，生成的 BST 也不同！

二叉搜索树的实现：节点链接法

- 需要用到 BST 和 Node 两个类，BST 的 root 成员引用根节点 Node
- class BinarySearchTree
  - root 引用 TreeNode 对象
  - size 表示节点总个数
  - 特殊函数 __iter__（以后再说）

```
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__()
```

二叉搜索树的实现：TreeNode 类

- class TreeNode
  - key 为键值
  - payload 是 value
  - left/rightChild
  - 另外加了个 parent 引用

```
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
```

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
```

二叉搜索树的实现：BST.put 方法

- put(key, val) 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, val, root) 来放置 key
- _put(key, val, currentNode) 的算法流程
  - 如果 key 比 currentNode.key 小，那么 _put 到 currentNode 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 currentNode.key 大，那么 _put 到 currentNode 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,val=0):
        if self.root:
            self.size += self._put(key,val,self.root)
        else:
            self.root = TreeNode(key, val)
            self.size += 1
```

二叉搜索树的实现：_put 辅助方法

- 当出现重复 key 时对 val 进行替换；返回 0 表示没有新增节点。

```
96      def _put(self, key, val, currentNode):
97          if key < currentNode.key:              #递归左子树
98              if currentNode.hasLeftChild():
99                  return self._put(key, val, currentNode.leftChild)
100             else:
101                 currentNode.leftChild = \
102                     TreeNode(key, val, parent=currentNode)
103                 return 1
104         elif key > currentNode.key:            #递归右子树
105             if currentNode.hasRightChild():
106                 return self._put(key, val, currentNode.rightChild)
107             else:
108                 currentNode.rightChild = \
109                     TreeNode(key, val, parent=currentNode)
110                 return 1
111         else: #key == currentNode.key          #出现重复key
112             currentNode.payload = val
113             return 0
```

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

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

二叉搜索树的实现：BST.put 图示

- 插入 key=19，currentNode 的变化过程（灰色）：

随堂练习

- 按照顺序 5, 30, 2, 40, 25, 4 插入 key，生成的 BST 是:

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 val，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:
        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)
```

二叉搜索树的实现：BST.get 相关的特殊方法

- ___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):
    if self._get(key,self.root):
        return True
    else:
        return False
```

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 remove 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 remove 方法！

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

```
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')
```

BST.remove 方法：没有子节点的情况

- 从 BST 中 remove 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
- 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
```

BST.remove 方法：被删节点有一个子节点

- 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？

BST.remove 方法：有一个子节点的处理代码

```
176         else:                                   #只有一个子节点的情况
177             # 取得唯一子节点，不关心左右
178             child = currentNode.leftChild \
179                     if currentNode.hasLeftChild() \
180                     else currentNode.rightChild  #可以砍掉一半的源代码
181             if currentNode.isLeftChild():     #用子节点替换当前节点
182                 currentNode.parent.leftChild = child
183                 child.parent = currentNode.parent
184             elif currentNode.isRightChild():
185                 currentNode.parent.rightChild = child
186                 child.parent = currentNode.parent
187             else :                            #当前节点是根节点，直接替换成子节点
188                 currentNode.replaceNodeData(child.key,
189                                             child.payload,
190                                             child.leftChild,
191                                             child.rightChild)
192
```

BST.remove 方法：有两个子节点的情况

- 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。

BST remove 4

BST.remove 方法：有两个子节点的处理代码

- BinarySearchTree 类：remove 方法（情形 3）

```
171         elif currentNode.hasBothChildren():     #有左右两个子节点的复杂情况
172             succ = currentNode.findSuccessor() #前驱、后继肯定都有
173             currentNode.key = succ.key
174             currentNode.payload = succ.payload
175             self.remove(succ)                   #succ没有左子节点，为什么？不会出现第二层递归
```

二叉搜索树的实现：BST.remove 方法

- TreeNode 类：寻找后继节点 findSuccessor()
  - 调用找到最小节点 findMin()

```
49      def findSuccessor(self):
50          if self.hasRightChild():          #在BinarySearchTree.remove()中的唯一可能
51              return self.rightChild.findMin()
52          elif self.isRoot():
53              return None
54          elif self.isLeftChild():
55              return self.parent
56          else : #self.isRightChild()        #在parent子树中的最末
57              self.parent.rightChild = None #暂时移除自己
58              succ = self.parent.findSuccessor()
59              self.parent.rightChild = self #恢复自己
60              return succ
61
62      def findMin(self):
63          current = self
64          while current.hasLeftChild():     #往左下角
65              current = current.leftChild
66          return current
```

二叉搜索树的实现：迭代器

- 对于 Python 字典，我们可以用 for i in dict: 这样的方法枚举字典中的所有 key，ADT Map 也应该实现这样的迭代器功能
- BinarySearchTree 类中的 __iter__ 方法直接调用了 TreeNode 中的同名特殊方法

二叉搜索树的实现：迭代器

- TreeNode 类中的 __iter__ 迭代器
  - 实际上是一个递归函数，想到上节 BinaryTree 中的 __str__ 递归方法么？
  - yield 是对每次迭代的返回值。

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

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

```
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
```

二叉搜索树：算法分析（以 put 方法为例）

- 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，那么 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($log_2 n$)。

- 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，如右图
  - 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？不受 key 插入顺序的影响？

目录

- 本章目标
- 树的例子
- 实现树
- 二叉树应用
- 树遍历
- 二叉堆实现的优先队列
- 二叉搜索树
  - 平衡二叉搜索树

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：

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

  - 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。
- $AVL \subset BST$

平衡树的定义

- 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
- 也可以说每个节点的左右子树高度差不超过 1
- 则把这个二叉搜索树称为平衡树

AVL 树的平衡策略

- 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
  - 同时要保持 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
- 一次查找所需的最多比较次数等于树高加一
- 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
- 构造最差情况：固定节点数，最大树高；或者固定树高，最少节点数。
  - 下图列出平衡因子为 1 的“左重”AVL 树，树的高度从 0 开始，来看看树高 h 和问题规模（总节点数 N）之间的关系如何？

AVL 树性能分析

- 观察上图 h=0、1、2、3 时，总节点数 N 的变化：
  - $N_0 = 1$
  - $N_1 = 2 = 1 + 1$
  - $N_2 = 4 = 1 + 1 + 2$
  - $N_3 = 7 = 1 + 2 + 4$
  - $N_h = 1 + N_{h-1} + N_{h-2}$
- 观察这个通式，很接近斐波那契数列！
  - 定义斐波那契数列 Fi
  - 归纳法，用 Fi 重写 N_h
- 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 Fi 的通式

$$F_0 = 0$$

$$F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2}, i \geq 2$$

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

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

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

AVL 树性能分析

- 将 Fi 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 log(N_h) 的搜索次数
- 可以说 AVL 树的搜索时间复杂度为 O(log n)

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

$$log(N_h + 1) = (h+2)log\Phi - \frac{1}{2}log5$$

$$h = \frac{log(N_h + 1) + \frac{1}{2}log5}{log\Phi} - 2$$

$$h \approx 1.44 log N_h$$

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 balanceFactor

AVL 树的实现：put 方法

- 重新定义 _put 方法即可
- _put 用递归实现：如果新增节点，就需要从它开始调整平衡因子

```
def _put(self, key, val, currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            return self._put(key, val, currentNode.leftChild)
        else:
            currentNode.leftChild = TreeNode(key, val, parent=currentNode)
            self.updateBalance(currentNode.leftChild)  #调整平衡因子
            return 1
    elif key > currentNode.key:
        if currentNode.hasRightChild():
            return self._put(key, val, currentNode.rightChild)
        else:
            currentNode.rightChild = TreeNode(key, val, parent=currentNode)
            self.updateBalance(currentNode.rightChild)  #调整平衡因子
            return 1
    else: #key == currentNode.key
        currentNode.payload = val          #无新增节点，平衡因子不变
        return 0
```

AVL 树的实现：UpdateBalance 方法

- 递归过程向上传递
- parent.balanceFactor 修改后为 0 时（-1->0 or 1->0），无需调整
- balanceFactor 修改后大于 1 或小于-1，调整平衡后子树高度不变，递归终止

```
21    def updateBalance(self, node):
22        if node.balanceFactor > 1 or node.balanceFactor < -1: #先看自己是否要调整
23            self.rebalance(node)                              #重新平衡，并且不会向上传递
24            return
25        if node.parent != None:                               #更新父节点平衡因子
26            if node.isLeftChild():
27                node.parent.balanceFactor += 1
28            elif node.isRightChild():
29                node.parent.balanceFactor -= 1
30            if node.parent.balanceFactor != 0:                #调整父节点平衡因子
31                self.updateBalance(node.parent)               #"=0"也会阻断递归的传递
```

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

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，新根节点原来有右子节点。
  - 旋转后，新根节点 C 将旧根节点 E 作为右子节点，新根节点原来已有右子节点不能扔掉，需要给它重新找位置！
  - 新根节点 C 原有的右子节点 D 改到旧根节点 E 的左子节点
  - 旧根节点 E 的左子节点 C 已经移到上面去了，空位置刚好可以用来放 D。
  - 除了 C 和 E 之外，其他节点为根的子树没有变化

只有C和E需要改变平衡因子

AVL 树的实现：rotateRight 代码

```
54      def rotateRight(self, rotRoot):
55          newRoot = rotRoot.leftChild
56          rotRoot.leftChild = newRoot.rightChild    #56，58新根的右子转为旧根的左子
57          if newRoot.rightChild != None:
58              newRoot.rightChild.parent = rotRoot
59          newRoot.parent = rotRoot.parent           #59，64/66新根取代旧根和父建立关系
60          if rotRoot.isRoot():
61              self.root = newRoot
62          else:
63              if rotRoot.isLeftChild():
64                  rotRoot.parent.leftChild = newRoot
65              else:
66                  rotRoot.parent.rightChild = newRoot
67          newRoot.rightChild = rotRoot              #67，68旧根下沉为新根的子节点
68          rotRoot.parent = newRoot
69          #调整新、旧根节点的平衡因子，为什么这样？马上推导。
70          rotRoot.balanceFactor = rotRoot.balanceFactor + \
71                                  1 - min(newRoot.balanceFactor, 0)
72          newRoot.balanceFactor = newRoot.balanceFactor + \
73                                  1 + max(rotRoot.balanceFactor, 0)
```

AVL 树的实现：rotateRight 示意图

AVL 树的实现：如何重新计算平衡因子

- 右旋、左旋进行的是完全对称的操作，下图用左旋转来说明
- 看看左旋转对平衡因子的影响
  - 保持了次序 ABCDE
  - A、C、E 的平衡因子不变
    - 子树高度 $h_A/h_C/h_E$ 不变
  - 只有旧根节点 B、新根节点 D 的平衡因子需要重新计算

AVL 树的实现：如何调整平衡因子

- 我们来推导 B 平衡因子的变化（D 的情况类似，略去...）
  - $B' = h_A - h_C$
  - $B = h_A - h'_D$ ’去掉，就是D的高度
  - 而：$h'_D = 1 + max(h_C, h_E)$，所以 $B = h_A - (1 + max(h_C, h_E))$
  - 进而 $B' - B = 1 + max(h_C, h_E) - h_C$
  - $B' = B + 1 + max(h_C, h_E) - h_C$；把 h_C 移进 max 函数里就有
  - $B' = B + 1 + max(0, -D)$
  - $B' = B + 1 - min(0, D)$

```
rotRoot.balanceFactor = rotRoot.balanceFactor + \
                        1 - min(newRoot.balanceFactor, 0)
```

AVL 树的实现：更复杂的情形

- 下图的“右重”(-2) 子树，单纯的左旋转无法实现平衡
  - 左旋转后变成“左重”(+2) 了
  - “左重”(+2) 再右旋转，还回到“右重”(-2)
- 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”(+1) 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

```
75      def rebalance(self, node):   #"<-1"或">1"才会被updateBalance调用
76          if node.balanceFactor < -1:  #右重需要左旋
77              if node.rightChild.balanceFactor > 0:
78                  # 右子节点左重，先对它进行一次右旋
79                  self.rotateRight(node.rightChild)
80              #正常左旋
81              self.rotateLeft(node)
82          elif node.balanceFactor > 1:
83              if node.leftChild.balanceFactor < 0:
84                  self.rotateLeft(node.leftChild)
85              self.rotateRight(node)
```

AVL 树的实现：操练一下

- 如图所示的“右重”子树如何平衡?
- 请图示旋转过程

①右子节点左重，故先右旋。

②整体左旋。

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？几次 updateBalance/rebalance
- 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)
- 课后练习：从 AVL 树删除一个节点，如何保持平衡？

ADT Map 的实现方法小结

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

list：put有插入操作，后移元素的复杂度是On

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
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)	$O(\log_2 n)$

- Map 驻留在内存中的情况，如果在文件中，就需要采用不同的标准
  - 数据库索引，B/B+ 树，时间复杂度关注的是访问磁盘的次数
  - 一次读取一个磁盘块，一个节点存很多 key，最大可能限制树的深度

本章总结

- 本章介绍了“树”数据结构，我们讨论了如下算法
- 用于表达式解析和求值的二叉树
- 实现了“最小堆”的完全二叉树：二叉堆
- 用于实现 ADT Map 的二叉搜索树 BST 树
- 改进了性能，用于实现 ADT Map 的平衡二叉搜索树 AVL 树

二叉树与树的关系与区别

- 二叉树是一类简单而又重要的树形结构
- 二叉树不是树
  - 二叉树的左右子树对应固定的位置，互相之间不可替代
    - 左子树被删除，它在父节点的位置“空”出来。
  - 树的第一子树被删除，第二子树自然顶替成为第一
- “k-叉树”与“树”，在把空集合定义为空树的情况下，k-叉树子女数量固定为 k，而树的子女数量不固定。
- 具体到二叉树，当我们说它的左子树为“空”（树）时，并不是说它没有左子树，
  - 准确的说，它没有“非空”左子树。
- 二叉树叶节点的定义：没有“非空”子节点。

关于树的进一步定义

- 在原有的树的定义 [16] 之外，新增下面的定义：
  - B 的子节点 E、F、G、H、I 组成 “线性表”。
  - E 为 B 的最左子节点
  - 上面线性表中的前驱/后继也可成为 “前/后” 兄弟
- 森林 (forest)：零棵或多棵不相交的树组成的有序的集合。
  - 可以是一个空集
- 树与森林的关系
  - 树可以看成一种特殊的森林：只有一颗树的森林。
  - 一棵树，删除树根，其子树就组成了森林。
  - 后面关于森林的操作，同样适用于树。

Figure: 树

Figure: 森林

森林与二叉树的等价转换

- 森林与二叉树之间可以互相转换，而且这种转换是一一对应的
  - 森林的相关操作都可以转换成对二叉树的操作
- 森林转化成二叉树的形式定义
  - 有序集合 $F = \{T_1, T_2, \ldots, T_n\}$ 是树 $T_1, T_2, \ldots, T_n$ 组成的森林，递归转换成二叉树 B(F)：
    - 若 F 为空，即 $n = 0$，则 B(F) 为空
    - 若 F 非空，即 $n > 0$，则：
      B(F) 的根节点是森林中第一棵树 T_1 的根节点 W_1
      B(F) 的左子树是 T_1 删去 W_1 后再转换来的二叉树：$B(T_1 - W_1)$
      B(F) 的右子树是 F 去除 T_1 后再转换来的二叉树：$B(F - T_1)$

森林与二叉树的等价转换

- 二叉树转化成森林或树的形式定义
  - 设 B 是一棵二叉树，root 是 B 的根，B_L 是 root 的左子树，B_R 是 root 的右子树，则对应于二叉树 B 的森林或树 F(B) 的形式定义是：
    - 若 B 为空，则 F(B) 是空的森林
    - 若 B 不为空，则 F(B) 是一棵树 T_1 加上森林 $F(B_R)$，其中树 T_1 的根为 root，root 的子树为 $F(B_L)$