

数据结构与算法 (Python)

07 树及算法

谢正茂 webg@PKU-Mail

计算机学院数据所

May 13, 2025

目录

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



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
- 二叉树的遍历
- 二叉堆实现的优先队列
- 二叉搜索树
- 树与树林

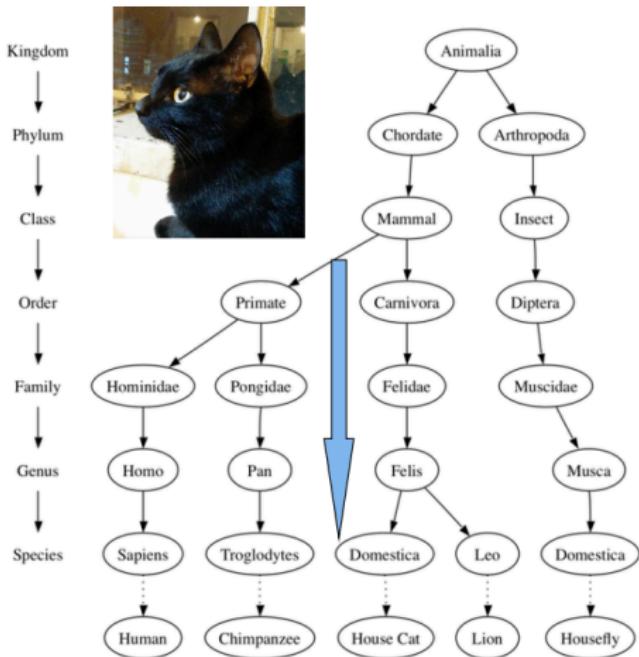


树的例子

- 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、枝和叶三个部分
- 不同的是，习惯把树画成倒的：根放在上方，叶放在下方
- 不要被各种表象迷惑，本质是特定的“前驱/后继”关系：
 - 除根结点之外，每个结点有且仅有一个直接前驱结点
 - 每个结点有零到多个直接后继结点；或者说除叶结点之外，每个结点有一到多个直接后继结点
- 也可以叫“父/子”或“祖先/后代”关系

树的例子：物种分类树

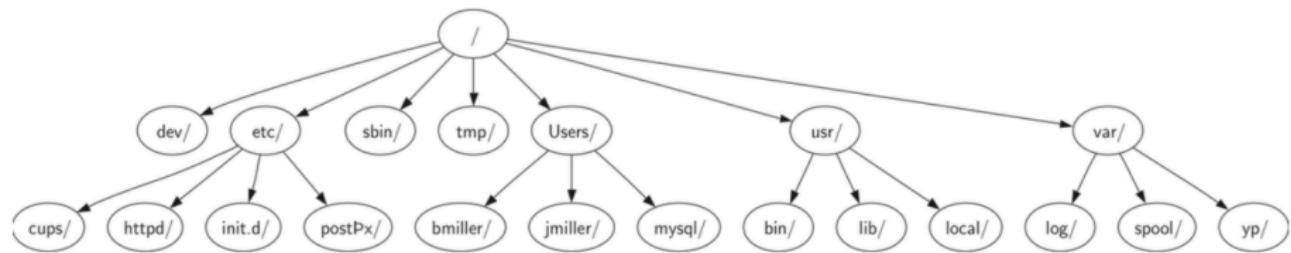
- 首先我们看到的分类树是层次化的
 - 越接近顶部的层越普遍
 - 越接近底部的层越独特
 - 界、门、纲、目、科、属、种
- 分类树的用法：辨认物种
 - 从顶端开始，沿着箭头方向向下
 - 门：脊索动物还是节肢动物？
 - 纲：哺乳动物么？
 - 目：食肉动物？
 - 科：猫科？
 - 属：猫属？
 - 种：家猫！



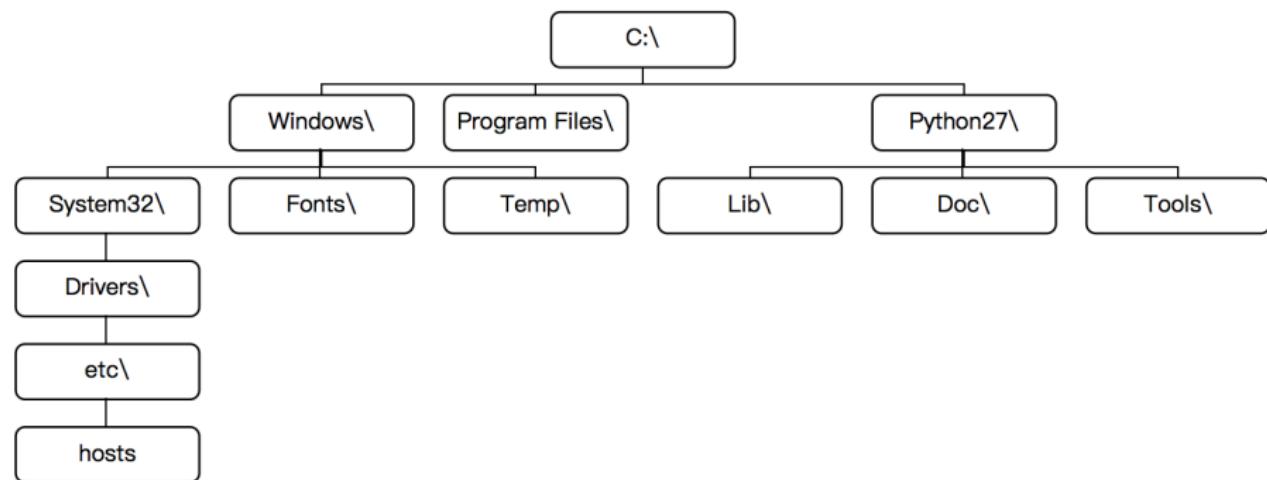
树的例子：物种分类树

- 分类树的第二个特征：一个结点的子结点与另一个结点的子结点相互之间是隔离、独立的
 - 猫属 **Felis** 和蝇属 **Musca** 下面都有 **Domestica** 的同名结点，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：每一个叶结点都具有唯一性
 - 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

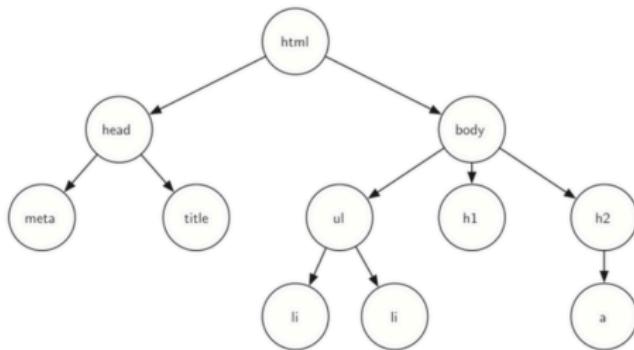


树的例子：Windows 文件系统

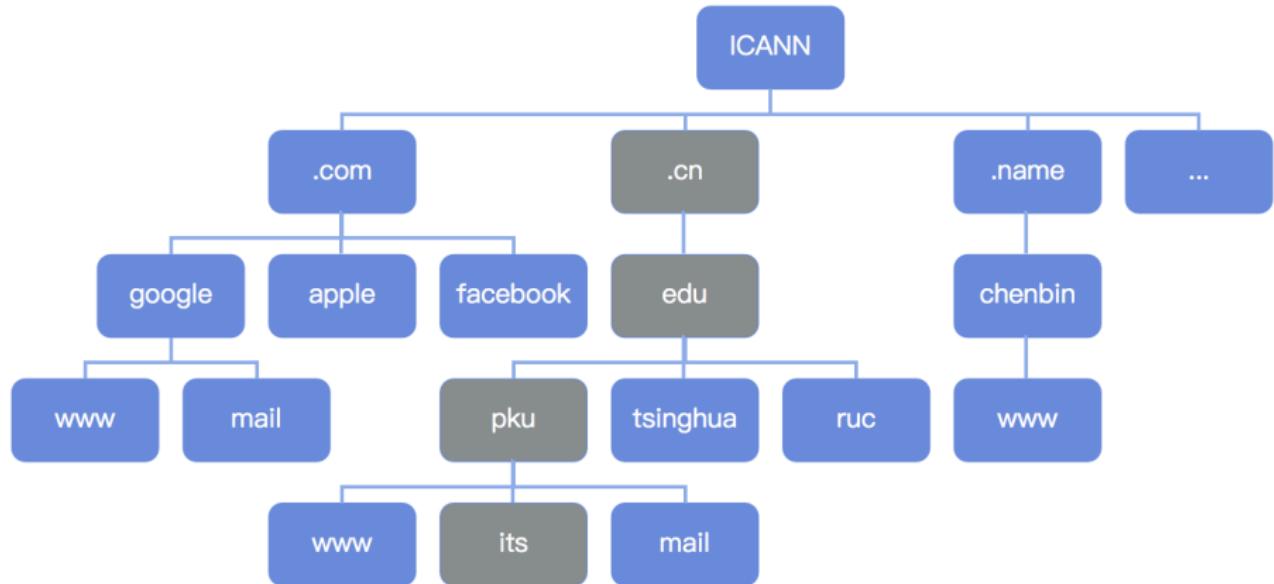


树的例子：HTML 文档（嵌套标记）

```
<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>
```

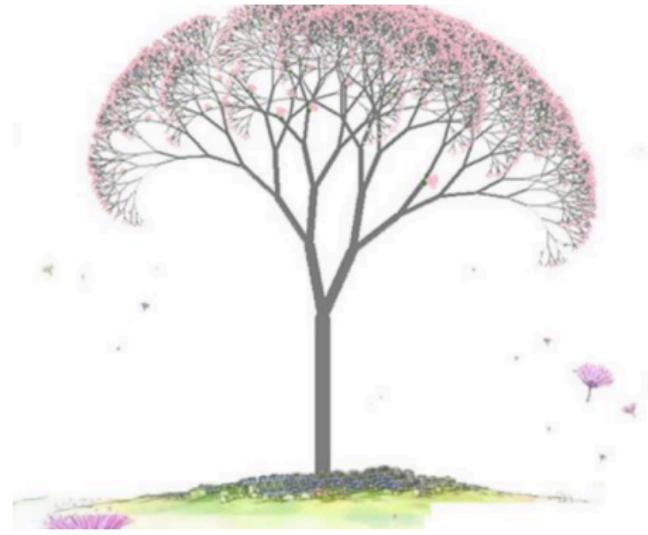


树的例子：域名体系



目录

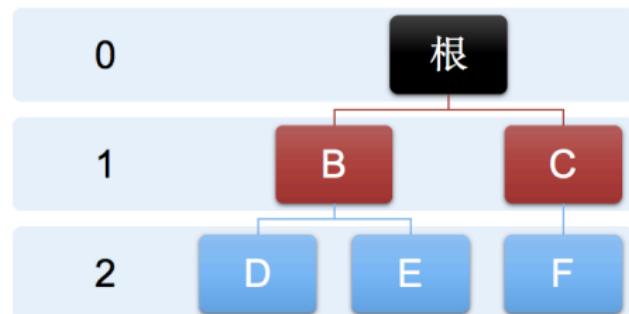
- 本章目标
- 树的例子
- **实现树**
- 二叉树应用
- 二叉树的遍历
- 二叉堆实现的优先队列
- 二叉搜索树
- 树与树林



- 结点 **Node**: 组成树的基本部分
 - 每个结点具有名称，或“键值”，结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结点，表示结点之间具有关联，边具有出入方向；
 - 每个结点（除根结点外）恰有一条来自另一结点的入边；
 - 每个结点可以有零条/一条/多条连到其它结点的出边。
 - 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结点
- 路径 **Path**: 由边依次连接在一起的结点的有序列表
 - 如：哺乳纲 → 食肉目 → 猫科 → 猫属，是一条路径

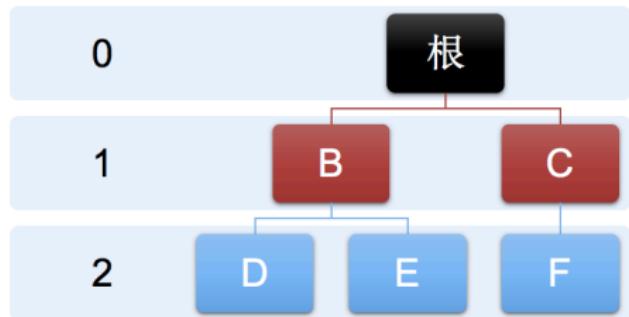
术语与定义

- 子结点 **Children**: 入边均来自于同一个结点的若干结点，称为这个结点的子结点
- 父结点 **Parent**: 一个结点是其所有出边所连接结点的父结点
- 兄弟结点 **Sibling**: 具有同一个父结点的结点之间称为兄弟结点
- 子树 **Subtree**: 一个结点和其所有子孙结点，以及相关边的集合



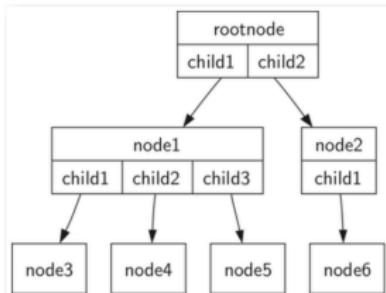
术语与定义

- 叶结点 **Leaf Node**: 没有子结点的结点称为叶结点
- 一个结点的子结点的数量，称为该结点的“度”
- 层级 **Level**: 从根结点开始到达一个结点的路径，所包含的边的数量，称为这个结点的层级。
 - 如 D 的层级为 2，根结点的层级为 0
- 高度 **Height**: 树中所有结点的最大层级称为树的高度
 - 如右图树的高度为 2



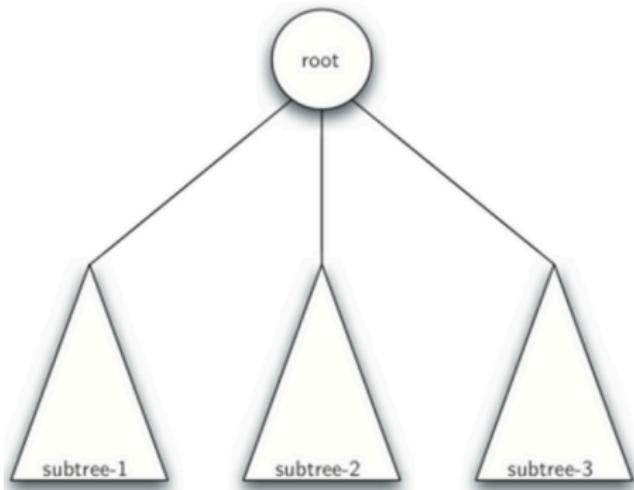
术语与定义：树的定义 1

- “父子”关系：除根结点外，每个结点都有唯一的前驱结点
- 树由若干结点，以及两两连接结点的边组成，并具有如下性质：
 - 唯一没有“前驱”的结点被设定为根；
 - 每个结点 n (除根结点)，都恰连接一条来自结点 p 的边， p 是 n 的父结点；
 - 每个结点从根开始的路径是唯一的
- “兄弟”关系：同一个父结点的子结点之间只是位置不同
 - 如果每个结点有且仅有左右两个子结点，这样的结构称为“二叉树 binary tree”
 - 二叉树的“兄弟”关系：左右子结点之间互相不可替代
- “树”和“二叉树”有相同的“父子”关系，在不严格的情况下统称为“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；
 - 或者由根结点及 0 或多个子树构成（其中子树也是树），每个“非空”子树的根到根结点具有边相连。
- 递归定义在下面的嵌套列表法中马上用到
- 概念上的困惑：空树有没有树根？

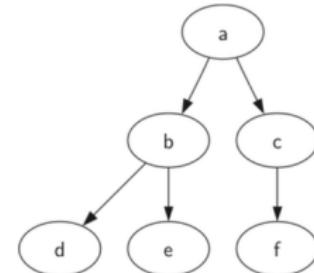


术语与定义：二叉树

- 因为“二叉树”与“树”的关系，可以沿用大部分树的术语
- 二叉树结点的“度”：该结点非空子结点的数量
- 在不引起混淆的情况下，往往把“非空”两个字省掉
- 例题：某二叉树共 101 个结点，其中度数为 1 的结点数有 30 个，问度数为 0 的结点有_____ 个。

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 个元素为根结点的值；
 - 第 2 个元素是左子树（用列表表示）；
 - 第 3 个元素是右子树。
- 以右图的示例，一个 6 结点的二叉树
 - 用下标 0、1、2 分别访问 myTree 的树根、左子树、右子树
- 嵌套列表法的优点
 - 子树的结构与树相同，是一种递归结构
 - 可以很容易扩展到多叉树，仅需要增加列表元素即可



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

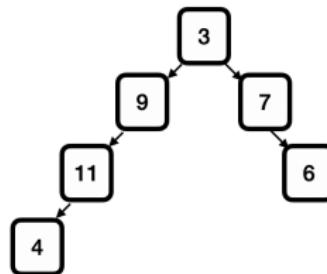
实现二叉树：嵌套列表法

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

```
1 def BinaryTree(r, left=[], right=[]):
2     return [r, left, right]
3
4 def insertLeft(root, newBranch):
5     root[1] = BinaryTree(newBranch,
6                          left=getLeftChild(root))
7     return root
8
9 def insertRight(root, newBranch):
10    root[2] = BinaryTree(newBranch,
11                         right=getRightChild(root))
12    return root
13
14 def getRootVal(root):
15     return root[0]
16
17 def setRootVal(root, newVal):
18     root[0] = newVal
19
20 def getLeftChild(root):
21     return root[1]
22
23 def getRightChild(root):
24     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

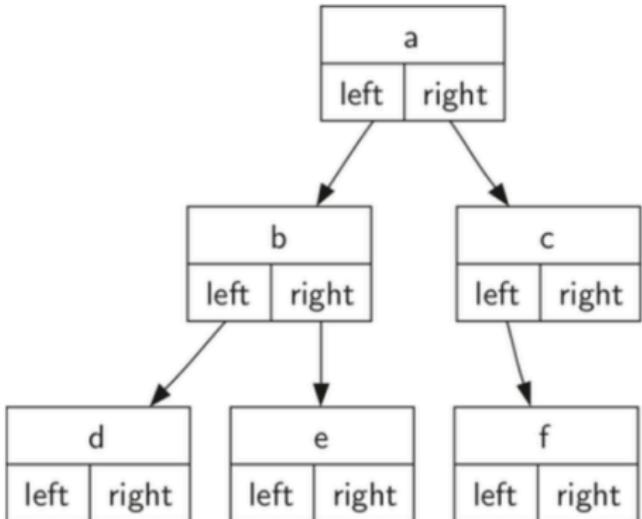


```
25
26 if __name__=="__main__":
27     r = BinaryTree(3)
28     insertLeft(r, 4)
29     insertLeft(r, 5)
30     insertRight(r, 6)
31     insertRight(r, 7)
32     l = getLeftChild(r)
33     print(l)
34
35     setRootVal(l, 9)
36     print(r)
37     insertLeft(l, 11)
38     print(r)
39     print(getRightChild(getRightChild(r)))
40
```

```
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：结点链接法

- 同样可以用类似链表的结点链接法来实现树
 - 每个结点保存该结点的数据项，以及指向左右子树的链接
- 定义一个 `BinaryTree` 类
 - 成员 `key` 保存根结点数据项
 - 成员 `left/rightChild` 则保存指向左/右子树的引用（同样是 `BinaryTree` 对象）
- 树和结点使用同一个类来实现

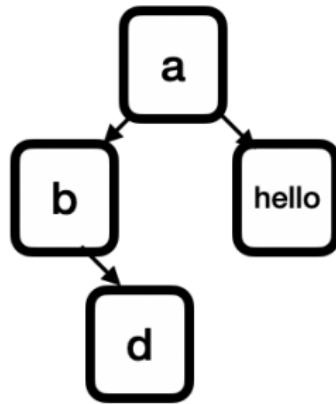


```
class BinaryTree:  
    def __init__(self, rootObj, \  
                 left=None, right=None):  
        self.key = rootObj  
        self.leftChild = left  
        self.rightChild = right
```

实现二叉树：结点链接法

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

```
68     r = BinaryTree('a')
69     r.insertLeft('b')
70     r.insertRight('c')
71     r.getRightChild().setRootVal('hello')
72     r.getLeftChild().insertRight('d')
```



```
33     def insertLeft(self, newNode):
34         self.leftChild = BinaryTree(newNode, \
35                                     left=self.leftChild)
36
37     def insertRight(self, newNode):
38         self.rightChild = BinaryTree(newNode, \
39                                      right = self.rightChild)
40
41     def getRightChild(self):
42         return self.rightChild;
43
44     def getLeftChild(self):
45         return self.leftChild
46
47     def setRootVal(self, obj):
48         self.key = obj
49
50     def getRootVal(self):
51         return self.key
```

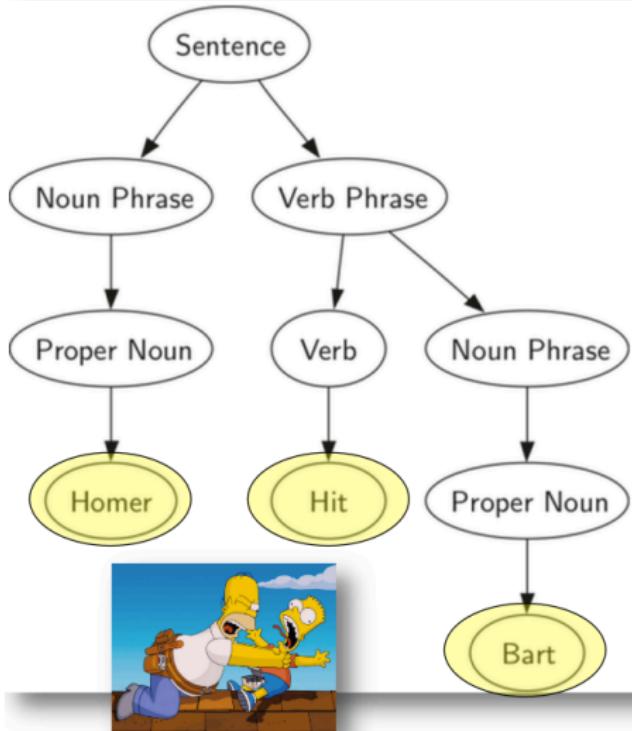
目录

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



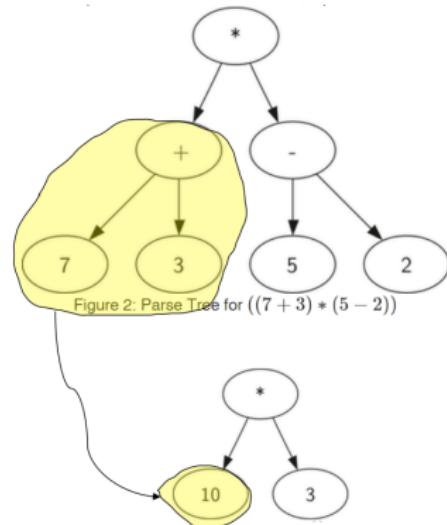
二叉树的应用：解析树 Parse Tree (语法树)

- 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
 - 叶结点保存操作数，内部结点保存操作符
- 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结点，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



二叉树的应用：表达式树

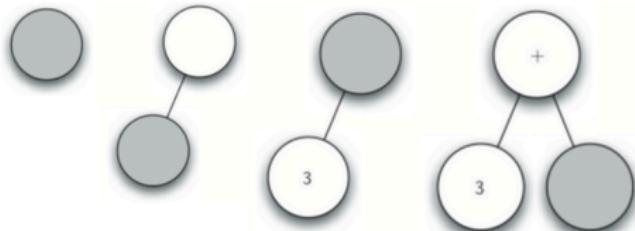
- 下面，我们用树结构来做如下尝试
 - 从全括号表达式构建表达式解析树
 - 利用表达式解析树对表达式求值
 - 从表达式解析树恢复原表达式的字符串形式
- 首先，全括号表达式要分解为单词 Token 列表
 - 其单词分为括号“()”、操作符“+*/”和操作数“0~9”这几类
 - 左括号就是表达式的开始，而右括号是表达式的结束

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"("：为当前结点添加一个新结点作为其左子结点，当前结点下降为这个新结点，准备接收左操作树/子树。
 - 如果当前单词是操作符 ['+', '-', '/', '*']：将当前结点的值设为此符号，为当前结点添加一个新结点作为其右子结点，当前结点下降为这个新结点，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：将当前结点的值设为此数，当前结点上升返回到父结点
 - 如果当前单词是")"：则当前结点上升返回到父结点

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，当前结点为根结点
 - 读入`'('`，创建了左子结点，当前结点左下降
 - 读入`'3'`，当前结点设置为 3，上升到父结点
 - 读入`'+'`，当前结点设置为 +，创建右子结点，当前结点右下降

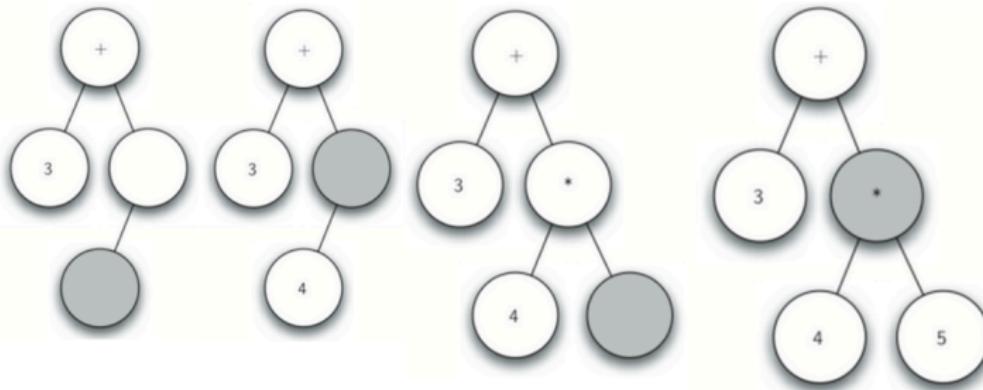


从全括号表达式建立表达式解析树：图示

- 创建表达式解析树过程

- 读入'(', 创建左子结点, 当前结点左下降
- 读入'4', 当前结点设置为 4, 上升到父结点
- 读入'*', 当前结点设置为 *, 创建右子结点, 当前结点右下降
- 读入'5', 当前结点设置为 5, 上升到父结点
- 读入')', 上升到父结点。

- 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

- 最外面的闭括号其实是没用的，处理不好反而带来程序 bug
- 从图示过程中我们看到，创建树过程中关键的是对当前结点的跟踪
 - 当前结点创建左右子树，可以调用 `BinaryTree.insertLeft/Right`
 - 当前结点设置值，可以调用 `BinaryTree.setRootVal`
 - 当前结点下降到左右子树，可以调用 `BinaryTree.getLeft/RightChild`
 - 但是，当前结点上升到父结点，这个没有方法支持！
- 于是我们可以用一个栈来记录、跟踪父结点
 - 当前结点下降时，将下降前的结点 `push` 入栈
 - 当前结点需要上升到父结点时，上升到 `pop` 出栈的结点即可！

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 = BinaryTreeNode('')      #创建一棵树空树
    pStack.append(currentTree)                  #这是在干什么？？？
    for i in fplist:
        if i == "(":
            currentTree.insertLeft('')
            pStack.append(currentTree)          #入栈左下降
            currentTree = currentTree.getLeftChild()
        elif i in ['+', '-', '*', '/']:
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.append(currentTree)          #入栈右下降
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()          #子表达式结束
        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：叶结点是最简单的子树，没有左右子结点，其根结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结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
- 二叉树的遍历
- 二叉堆实现的优先队列
- 二叉搜索树
- 树与树林



字符的编码问题：定长编码

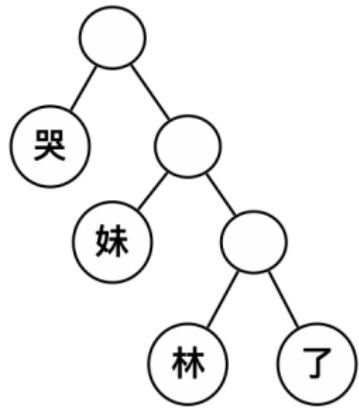
- 所有字符编码成等长的二进制 0/1 序列
- 下图中“林妹妹哭哭哭哭了”编码后的串长度为 20bits
- 现实中的 unicode 采用的就是这种方式—每个字符都编码成 4bytes(32bits)
 - ASCII、中文字符，考虑了世界上绝大多数语言的字符
 - 优点：所有字符采用统一的编码长度，处理比较方便。
 - 缺点：空间太浪费了，ASCII 1byte \Rightarrow 4bytes

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

- 前缀码：
 - 所有符号串之间互不为前缀
 - 编码的符号串会连在一起使用，这样才能避免歧义
- 字符编码成不等长的 0/1 序列
- “林妹妹哭哭哭哭哭了”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



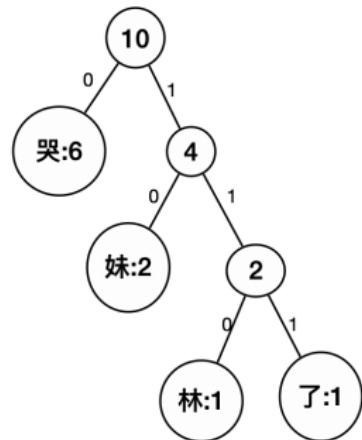
林:110, 妹:10, 哭:0, 了:111

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 Huffman 编码树

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

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



哈夫曼编码思路

- 让常见的字符使用最短的编码；
- 反过来看，也就是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 最小的两个结点组成树；
 - 根结点的 w 为它们 w 的和；
 - 把根结点放回集合
- 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），中间结点的字符为 `None`

哈夫曼建树代码

● 建立哈夫曼编码树

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

● 输出哈夫曼编码

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

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

随堂练习-哈夫曼编码

- www.cncorpus.org 网站从 2000 万字的现代汉语语料中统计了 5708 个汉字字频。
- 我们从中等间距抽取了 12 个汉字及字频，请为它们构建 Huffman 树并写出它们的编码。

序号	汉字	字频	序号	汉字	字频
1	的	759834	3001	秤	150
501	维	6485	3501	啧	84
1001	损	2330	4001	铿	80
1501	厅	1084	4501	柏	70
2001	氛	528	5001	珙	60
2501	聘	291	5501	妣	12

- 简单一点：编码 a(3),b(4),c(9),d(12),e(15),f(20) 字符集，括号中的数字为字符出现的频率。
- 计算编码的文本总长度 WPL(也称哈夫曼树的加权路径长度)

目录

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

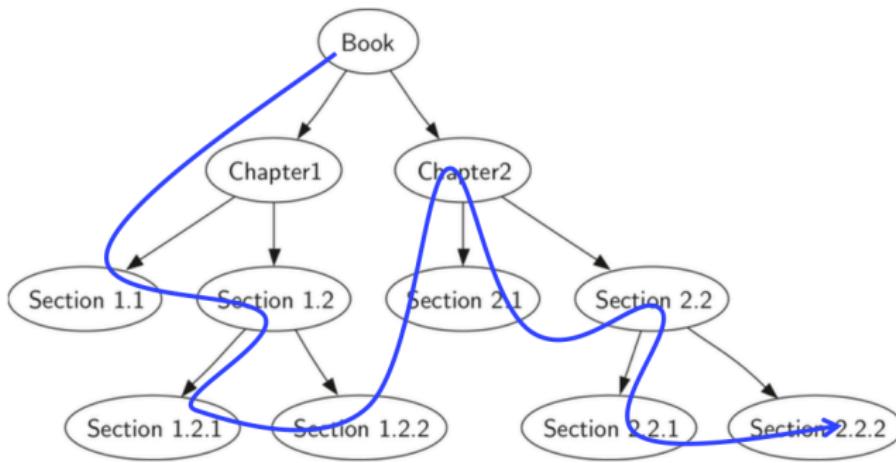


二叉树的遍历 Binary Tree Traversals

- 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，我们按照对结点访问次序的不同来区分 3 种遍历
 - 前序遍历 (preorder): 先访问根结点，再递归地前序访问左子树、最后前序访问右子树；也叫先根遍历
 - 中序遍历 (inorder): 先递归地中序访问左子树，再访问根结点，最后中序访问右子树；也叫中根遍历
 - 后序遍历 (postorder): 先递归地后序访问左子树，再后序访问右子树，最后访问根结点；也叫后根遍历
 - “前”、“中”、“后”，都是针对根结点来说的。

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



二叉树的遍历：递归算法代码

- 树遍历的代码非常简洁！

- 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

- 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 preorder(tree):  
    if tree:  
        print(tree.getRootVal())  
        preorder(tree.getLeftChild())  
        preorder(tree.getRightChild())  
  
def preorder(self):  
    print(self.key)  
    if self.leftChild:  
        self.leftChild.preorder()  
    if self.rightChild:  
        self.rightChild.preorder()
```

```
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：

```
53 import operator
54 def postordereval(tree):
55     opers = {'+':operator.add, '-':operator.sub, \
56              '*':operator.mul, '/':operator.truediv}
57     if tree:
58         left = postordereval(tree.getLeftChild())          #左子树
59         right = postordereval(tree.getRightChild())        #右子树
60         if left and right:
61             return opers[tree.getRootVal()](left, right)    #计算根节点
62         else:
63             return tree.getRootVal()                         #直接返回根节点
64     else:
65         return None                                       #空集返回None
66
```

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
- 二叉树的遍历
- 二叉堆实现的优先队列
- 二叉搜索树
- 树与树林



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

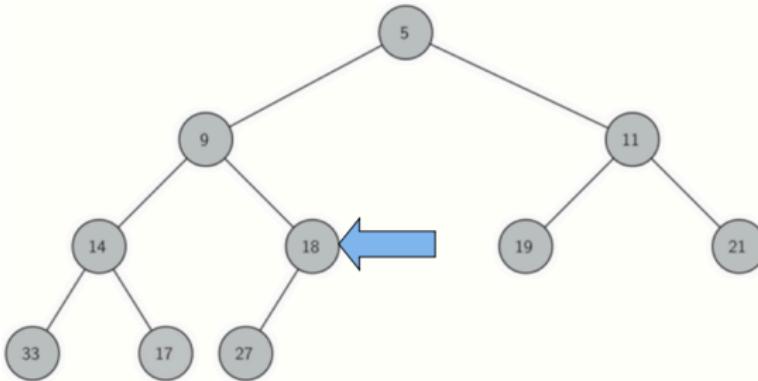
ADT BinaryHeap 的操作示例

```
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  
>>>
```

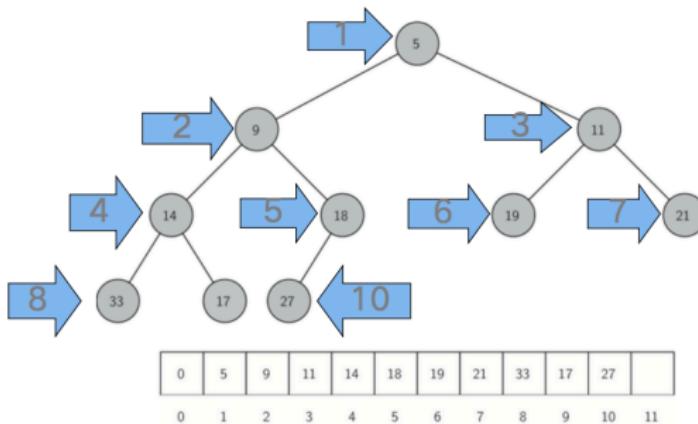
用“非嵌套”列表实现二叉堆

- 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结点
- 我们采用“完全二叉树”的结构来近似实现“平衡”
 - 完全二叉树：高度为 n 的二叉树，第 i ($i < n$) 层结点数为 2^i ；第 n 层的结点从左往右紧密排列。
 - 最多有一个结点有且仅有两个子结点，且必须是左子结点。
 - 叶子结点出现在下面“两层”。
 - 我们从 1 开始，从上到下，从左到右给所有结点编号。



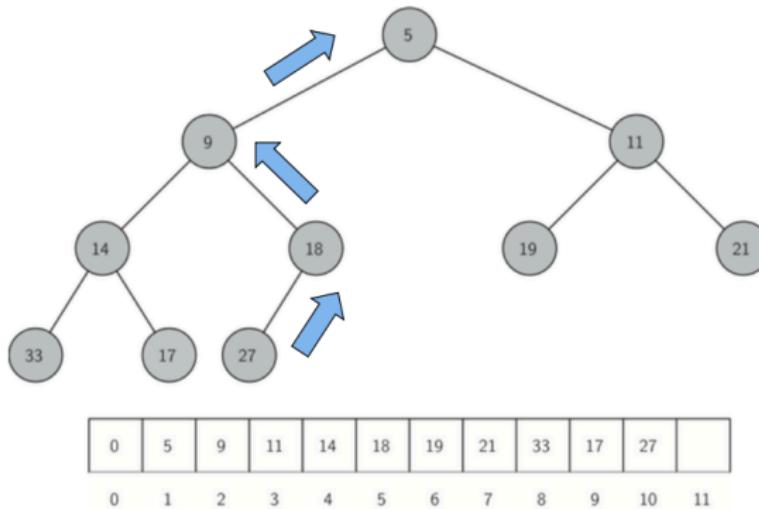
完全二叉树的列表实现及性质

- 完全二叉树由于其特殊性，可以把它放到一个“非嵌套”的列表中，并具有很好的性质：
 - 列表的 0 下标元素弃用
 - 如果结点在列表中的下标为 p ，那么其左子结点下标为 $2p$ ，右子结点为 $2p+1$
 - 如果结点在列表中下标为 n ，那么其父结点下标为 $n//2$



堆次序 Heap Order

- 所谓“堆”的特性，是指堆中任何一个结点 x ，其父结点 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，其树根结点中的 key 最小



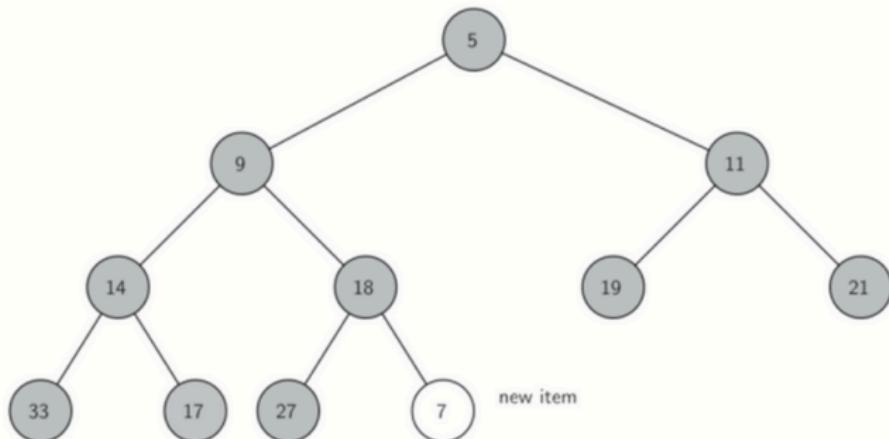
二叉堆操作的实现

- 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？

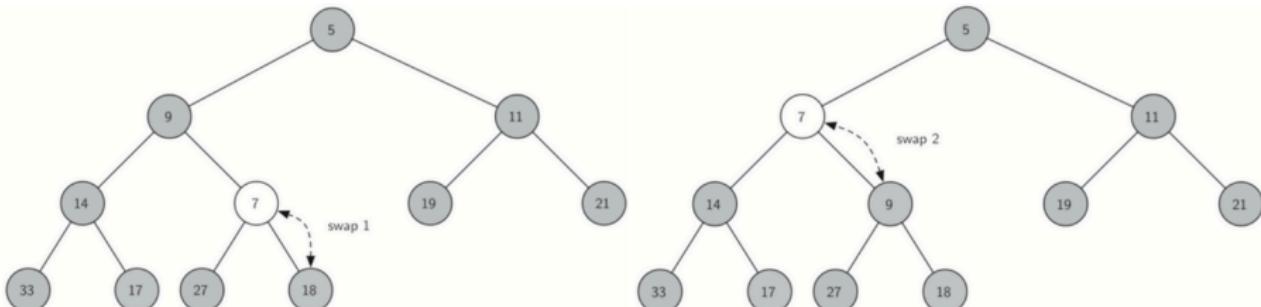


```
class BinHeap:  
    def __init__(self):  
        self.heapList = [0]  
        self.currentSize = 0
```

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”不会影响其它路径结点的“堆”次序，上浮路径上的结点值只会变小



二叉堆操作的实现：insert 代码

```
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
```

与父节点交换

沿路径向上

添加到末尾

新key上浮

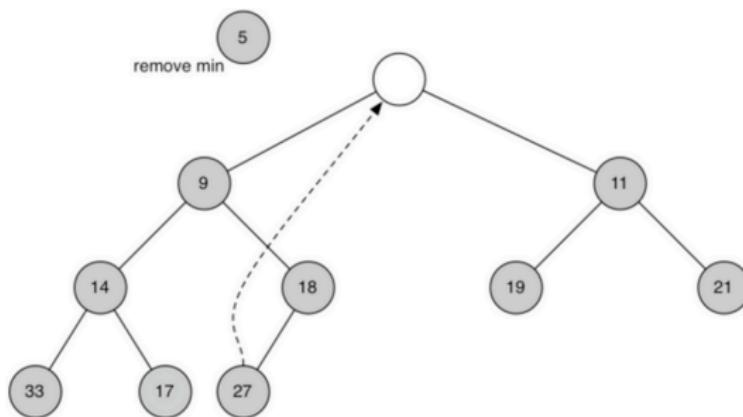
```
def insert(self, k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self_percUp(self.currentSize)
```

- 代码中的瑕疵？
- 参考：<http://github.com:dsaClass/dsa2020.git, course/binHeap.py>

二叉堆操作的实现：delMin

- **delMin()** 方法

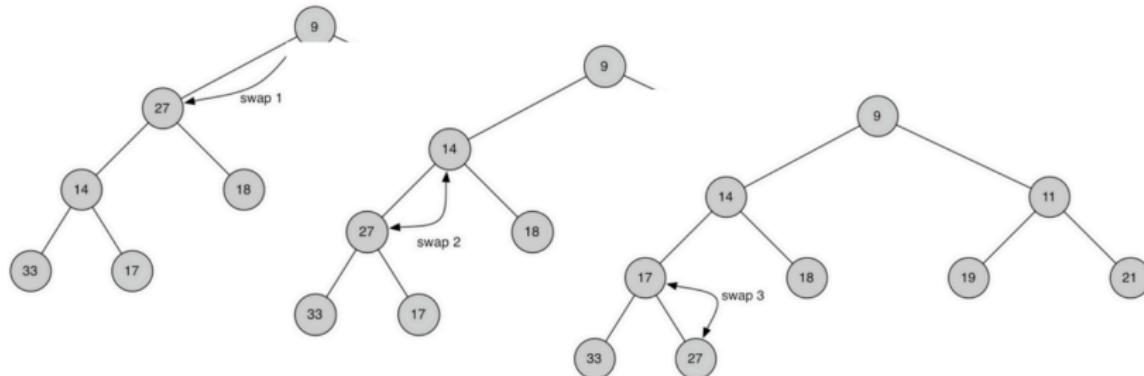
- 首先，移走整个堆中最小的 key——位于堆首位的根结点 `heapList[1]`
- 为了保持“完全二叉树”的性质，用最后一个结点来代替根结点
- 问题？



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：将新的根结点沿着一条路径“下沉”，直到比两个子结点都小
- “下沉”路径的选择：如果比子结点大，那么选择较小的子结点交换下沉



二叉堆操作的实现：delMin 代码

```
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

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

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
```

交换下沉

沿路径向下

唯一子节点

返回较小的

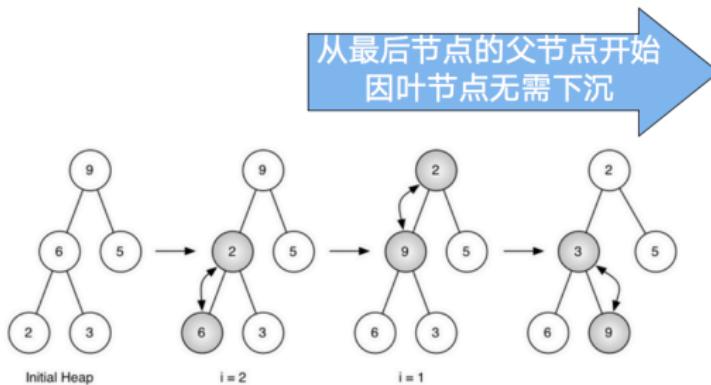
移走堆顶

新顶下沉

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 $O(n \log n)$
- 其实，用“下沉”法，能够将总代价控制在 $O(n)$
 - 叶结点不需调用 **percDown()**
 - 不同结点调用 **percDown()** 所执行的比较次数也不一样，与结点离叶结点的距离有关
 - $1/2$ 的结点进行一组比较， $1/4$ 的结点进行 2 组比较，...



```
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，我们定义树中结点的高度 $height$ 为该结点到叶结点的最大路径长度。那么，
 - 叶结点的高度为零
 - 根结点的高度等于树的高度
- 在 $percDown(self,i)$ 下沉 i 结点的过程中，每下降一个高度进行三次比较，合计执行的比较次数

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

- 假设完全二叉树的结点个数为 n ，那么整个建堆过程的比较次数

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

- 高度不小于 1 的结点数为 $n//2$ ，不小于 2 的结点数为 $n//4$ ，不小于 3 的结点数为 $n//8$ ，… 剩下约 $n//2$ 的叶结点高度为 0
- 高度恰好为 k 的结点的个数为 $n//2^k - n//2^{k+1}$

二叉堆建堆操作的算法复杂度证明 Continue

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

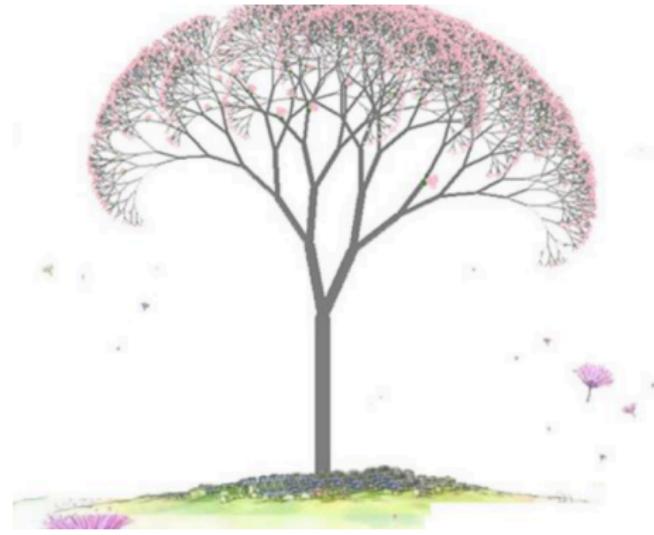
- 2024 陈同学提问：建堆时为什么不能先沉堆顶？
- 有了“最小堆”，我想用“最大堆”怎么办？

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： $O(n \log n)$
 - “堆排序”是稳定的吗？
 - “堆排序”能原地工作吗？
- 取 N 个数据中取最小 m 个 ($m \ll N$) 算法复杂度： $O(N+m\log N)$
- 二叉堆在各种算法中很常用，Python 提供的实现
 - `import heapq`
 - 提供的堆操作：`heappush`, `heappop`, `heapify`
 - 不提供的堆操作：`percUp`, `percDown`
- 二叉堆的结构特点对应的 OJ 题
 - OJ24686-树的重量
 - OJ24637-宝藏二叉树

目录

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



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**

- 复习一下 ADT Map 的操作:

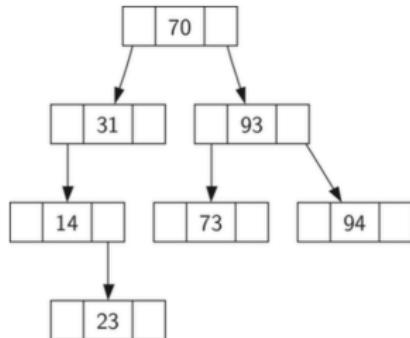
- Map(): 创建一个空映射
- put(key, val): 将 key-val 关联对加入映射中, 如果 key 已经存在, 则将 val 替换原来的旧关联值;
- get(key): 给定 key, 返回关联的数据值, 如不存在, 则返回 None;
- del: 通过 del map[key] 的语句形式删除 key-val 关联;
- len(): 返回映射中 key-val 关联的数目;
- in: 通过 key in map 的语句形式, 返回 key 是否存在于关联中, 布尔值

- 复习一下树的实现形式:

- 嵌套列表法
- 结点链接法
- 非嵌套-列表法 (只适用于完全二叉树)
- 提问: 上面哪些是顺序存储?

二叉搜索树 (BST) 的性质

- 二叉搜索树的每个结点上都存储了 key
- 比父结点小的 key 都在左子树，比父结点大的 key 都在右子树。
- 右图是一个简单的 BST，按照 70,31,93,94,14,23,73 的顺序插入
- 画图的时候只画 key 而省略 value
- 首先插入的 70 成为树根
 - 31 比 70 小，放到左子结点
 - 93 比 70 大，放到右子结点
 - 94 比 93 大，放到右子结点
 - 14 比 31 小，放到左子结点
 - 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 []`
- 是不是也可以叫二叉排序树？中序遍历得到有序表
- 注意：插入顺序不同，生成的 BST 也不同！



二叉搜索树的实现：结点链接法

- BST 和 Node 分别用类来实现，BST 的 root 成员引用根 Node
- class BinarySearchTree
 - root 引用 TreeNode 对象
 - size 表示结点总个数
 - 特殊函数 `__iter__`（以后再说）
- 参考代码：
`course/binarySearchTree.py`
- 和 Python dict 一样，
BinarySearchTree 是 ADT Map 的另一个实现

```
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
 - 只是简单调用一个递归函数 `_put(key, val, root)` 来放置 key
 - 并且把该函数的返回赋值给 root
- `_put(key, val, currentNode)` 的算法流程
 - 如果当前结点为 None，生成一个新的结点，并把 size 计数器加一。
 - 如果 key 比 `currentNode.key` 小，那么 `_put` 到 `currentNode` 左子树
 - 如果 key 比 `currentNode.key` 大，那么 `_put` 到 `currentNode` 右子树
 - 把左或右子结点的 `parent` 指向当前结点
 - 如果出现重复 key，更新它的 payload！不同于“排序”
 - 所以 BST 中的'S' 是 search，而不是 sort

```
93     def put(self, key, val=0):
94         # 调用递归形式的 _put
95         self.root = self._put(key, val, self.root)
```

二叉搜索树的实现：_put 辅助方法

```
97     def _put(self, key, val, curr):
98         """
99             按照二叉搜索树的规则，采用递归的方式在子树中插入新结点。
100            返回子树的根
101        """
102        if curr is None:
103            self.size += 1
104            return self.TreeNode(key, val)
105        if key < curr.key:          #递归左子树
106            curr.leftChild = self._put(key, val, curr.leftChild)
107            curr.leftChild.parent = curr
108        elif key > curr.key:       #递归右子树
109            curr.rightChild = self._put(key, val, curr.rightChild)
110            curr.rightChild.parent = curr
111        else: #key == curr.key
112            curr.payload = val      #更新结点，无新增
113        return curr
```

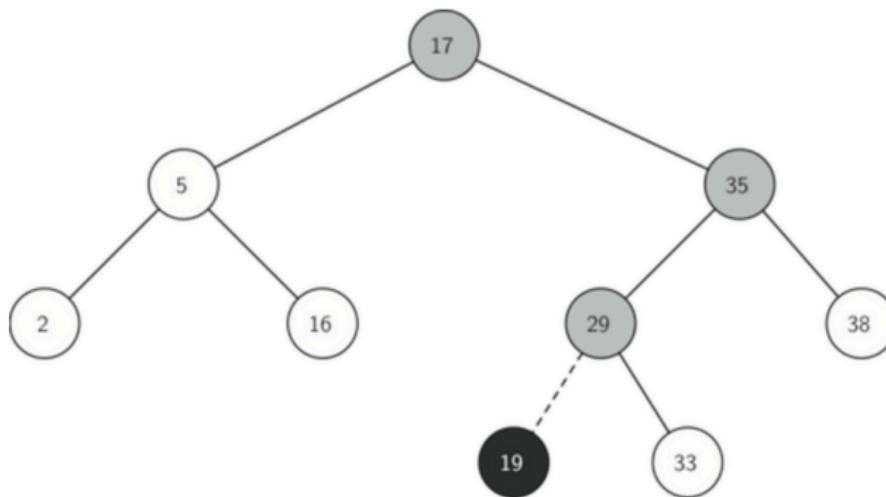
- 随手把 `__setitem__` 做了：

- 可以 `myZipTree['PKU'] = 100871`

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

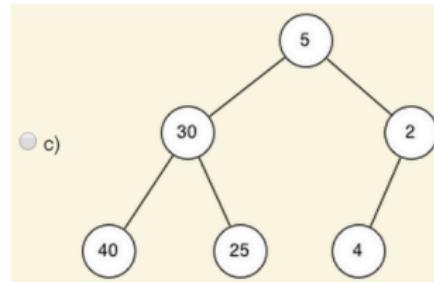
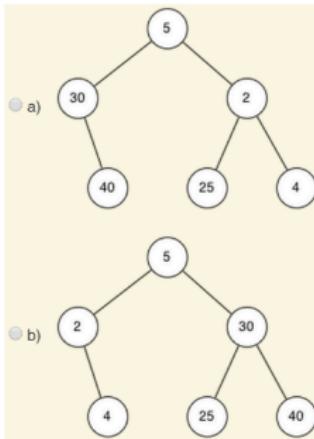
二叉搜索树的实现：BST.put 图示

- 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 所在的结点
 - 从 root 开始，递归向下，直到找到，或者下到最底层的叶结点也未找到。

• get 方法

- 调用递归函数 _get
- 根据 _get 的结果，返回结点的值或空

• _get 方法

- curr 为空返回 None
- 匹配当前结点，成功
- 否则递归进入左/右子树

```
118     def get(self, key):  
119         if res := self._get(key, self.root):  
120             return res.payload  
121         else:  
122             return None  
123  
124     def _get(self, key, curr):  
125         """  
126             返回匹配 key 的结点  
127         """  
128         if not curr:  
129             return None  
130         elif curr.key == key:  
131             return curr  
132         elif curr.key > key:  
133             return self._get(key, curr.leftChild)  
134         else: # curr.key < key  
135             return self._get(key, curr.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

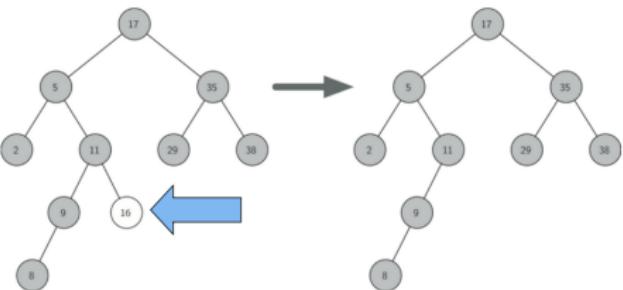
- 有增就有减，`delete` 方法比较复杂：
 - 首先用 `_get` 找到要删除的结点，然后调用 `remove` 来删除结点，找不到则提示错误；
 - 输入参数不同：`delete` 删除的目标是 `key`，而 `remove` 删除的目标是 `node`
- `__delitem__` 特殊方法
 - 实现 `del zipTree['PKU']` 这样的语句操作
- 很多细节由 `remove` 方法来处理

```
149     def delete(self, key):
150         nodeToRemove = self._get(key, self.root)
151         if nodeToRemove:
152             self.remove(nodeToRemove)
153             self.size -= 1
154         else:
155             raise KeyError("'{0}' not found.".format(key))
156
157     def __delitem__(self, key):
158         self.delete(key)
```

BST.remove 方法：叶结点的情况

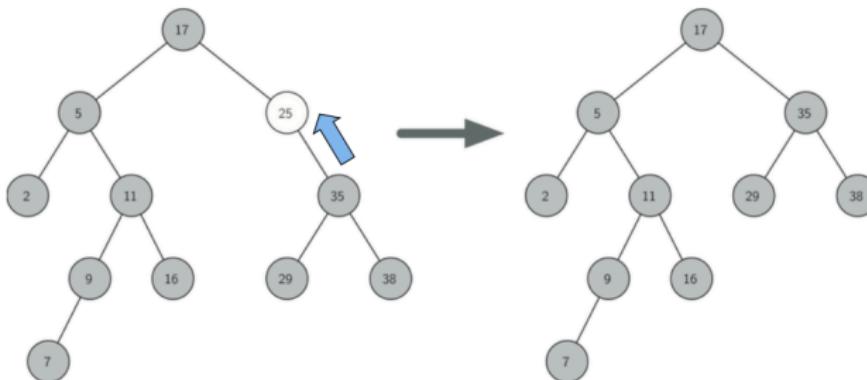
- 从 BST 中 remove 一个结点，还仍然保持 BST 的性质，分以下 3 种情形
 - 结点没有子结点
 - 结点有 1 个子结点
 - 结点有 2 个子结点
 - 没有子结点的情况好办，直接删除

```
165     if curr.isLeaf():    #作为叶节点的最简单情况
166         #移除叶节点只需要在它的父节点中把它移除
167         p = curr.parent
168         if curr.isLeftChild():
169             p.leftChild = None
170         elif curr.isRightChild():
171             p.rightChild = None
172         else:                      #删除唯一结点
173             self.root = None
174         return p
```



BST.remove 方法：被删结点有一个子结点

- 第 2 种情形稍复杂：被删结点有 1 个子结点
 - 解决：将这个唯一的子结点上移，替换掉被删结点的位置
- 但替换操作需要区分几种情况：
 - 被删结点的子结点是左？还是右子结点？
 - 被删结点本身是其父结点的左？还是右子结点？
 - 被删结点本身就是根结点？



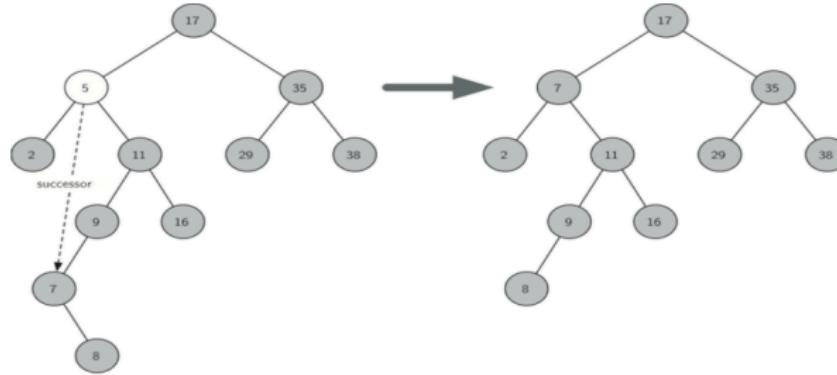
BST.remove 方法：有一个子结点的处理代码

- 当前结点是根结点时，需要特殊处理

```
181     else:                      #只有一个子节点的情况
182         # 取得唯一子节点，不关心左右
183         child = curr.leftChild \
184             if curr.hasLeftChild() \
185             else curr.rightChild #可以砍掉一半的源代码
186         p = child.parent = curr.parent
187         if curr.isLeftChild():   #在子、父结点之间跳过当前节点
188             p.leftChild = child
189         elif curr.isRightChild():
190             p.rightChild = child
191         else :                  #当前节点是根节点，直接替换成子节点
192             self.root = child
193     return p
```

BST.remove 方法：有两个子结点的情况

- 第 3 种情形最为复杂：被删结点有 2 个子结点
 - 这时无法简单地将某个子结点上移替换被删结点
 - 但可以找到另一个合适的结点来替换被删结点，这个合适结点就是被删结点的下一个 key 值结点，即被删结点右子树中最小的那个，称为“后继”
 - 可以肯定这个后继结点最多只有 1 个子结点（本身是叶结点，或仅有右子树）
 - 将后继结点的内容移动到被删结点，再摘除后继结点。

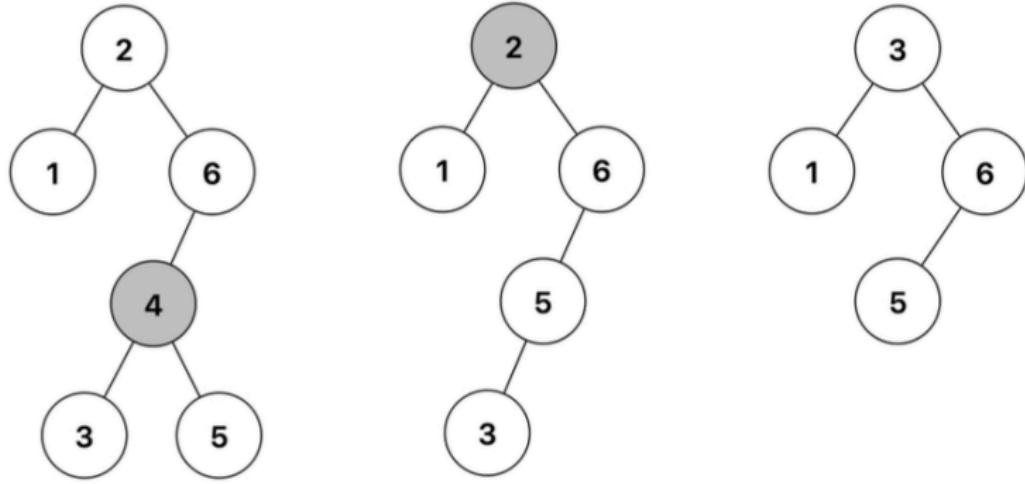


BST.remove 方法：有两个子结点的处理代码

- BinarySearchTree 类: remove 方法 (情形 3)

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

BST.remove 方法：更多的例子



- remove() 完整代码见 [github:course/binarySearchTree.py](https://github.com/course/binarySearchTree.py)

TreeNode 类：寻找后继结点 findSuccessor()

- 调用 findMin(), 找到最小结点

```
49     def findSuccessor(self):  
50         if self.hasRightChild():    #在remove()中的唯一可能  
51             return self.rightChild.findMin()  
52         elif self.isRoot():  
53             return None  
54         elif self.isLeftChild():    #是左子树且无右子女  
55             return self.parent  
56         else : #是右子树且无右子女，处在parent子树中的末位  
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。
- `yield` 函数与 `return` 函数的区别？`yield` 函数有记忆。

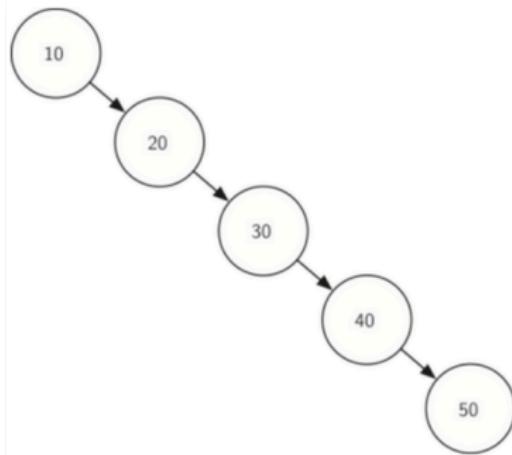
```
mytree = BinarySearchTree()  
mytree[3] = "red"  
mytree[4] = "blue"  
mytree[6] = "yellow"  
mytree[2] = "at"  
  
print(3 in mytree)  
print(mytree[6])  
del mytree[3]  
print(mytree[2])  
for key in mytree:  
    print(key, mytree[key])
```

```
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，那么大于和小于根结点 **key** 的键值大致相等，那么 BST 的高度就是 $\log_2 n$ (n 是结点的个数)，这样的树就是平衡树，**put** 方法的时间复杂度为 $O(\log n)$ 。

- 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** 方法的时间复杂度为 $O(n)$
- 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
- 二叉树的遍历
- 二叉堆实现的优先队列
- 二叉搜索树
 - 平衡二叉搜索树
- 树与树林



平衡二叉搜索树：AVL 树的定义

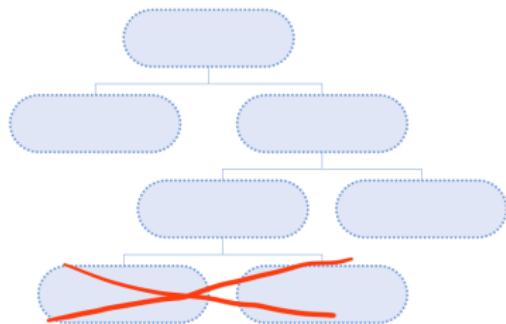
- 下面介绍一种能够在 key 插入时一直保持平衡的二叉搜索树：AVL 树
 - AVL 是发明者的名字缩写：G.M. Adelson-Velskii and E.M. Landis
- 利用 AVL 树实现 ADT Map，基本上与前面的 BST 实现相同，不同之处在于二叉树的变化过程中需要维持平衡。
- AVL 树的实现中，需要对每个结点跟踪“平衡因子 balance factor”参数，平衡因子是根据结点的左右子树的高度来定义的，确切地说，是左右子树高度差：

$$\text{balanceFactor} = \text{height}(\text{leftSubTree}) - \text{height}(\text{rightSubTree})$$

- 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。
- $\text{AVL} \subset \text{BST}$

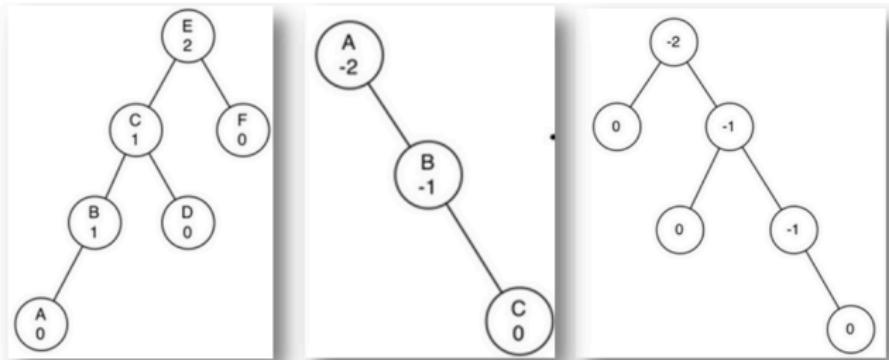
平衡树的定义

- 如果一个二叉搜索树中每个结点的平衡因子都在 $-1, 0, 1$ 之间
- 也可以说每个结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



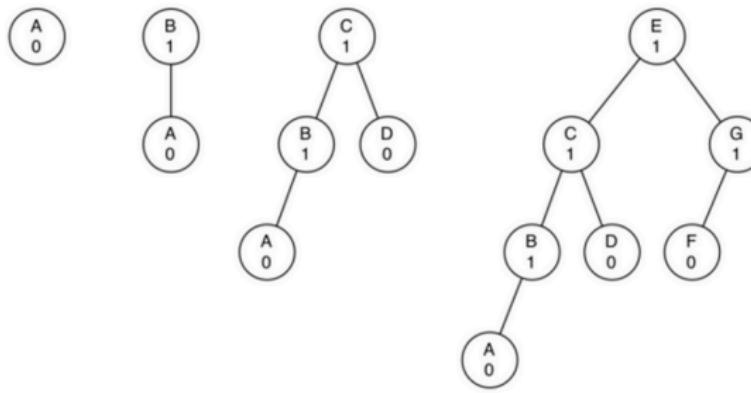
AVL 树的平衡策略

- 一旦在平衡树操作过程中，有结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！左子树所有结点都小于根，右子树都大于根
- 下面是几颗“左重/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，先看看 AVL 树是否确实有一个很好的性能
- 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：固定结点数，最大树高；或者固定树高，最少结点数。
 - 下图列出平衡因子为 1 的“左重”AVL 树，树的高度从 0 开始，来看树高 h 和问题规模（总结点数 N ）之间的关系如何？



AVL 树性能分析

- 观察上图 $h=0, 1, 2, 3$ 时，总结点数 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 F_i
 - 归纳法，用 F_i 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 F_i 的近似通式

$$F_0 = 0$$

$$F_1 = 1$$

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

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

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

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

AVL 树性能分析

- 将 F_i 通式代入到 N_h 中，得到 N_h 的通式
- 上述通式只有 N 和 h 了，我们解出 h
- 可见，在最差情况下，AVL 树在总结点数为 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 $O(\log n)$

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

$$\begin{aligned} \log(N_h + 1) &= (h + 2) \log \Phi - \frac{1}{2} \log 5 \\ h &= \frac{\log(N_h + 1) + \frac{1}{2} \log 5}{\log \Phi} - 2 \\ h &\approx 1.44 \log N_h \end{aligned}$$

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 必定以叶结点形式插入到 AVL 树中
 - 叶结点的平衡因子是 0，其本身无需重新平衡
 - 但会影响其父结点的平衡因子：
 - 如果作为左子结点插入，则父结点平衡因子会增加 1；
 - 如果作为右子结点插入，则父结点平衡因子会减少 1。
 - 这种影响可能随着其父结点到根结点的路径一直传递上去，直到：
 - 传递到根结点为止；
 - 或者某个父结点平衡因子被调整到 0，不再影响上层结点的平衡因子为止。
 - 发生“再平衡”子树的高度会回到平衡被破坏之前的值

AVLTreeNode 的实现

- 将 AVL 树作为 BST 树子类实现，TreeNode 中增加 height 属性，balanceFactor 通过左右子树的 height 计算得到
- 参考代码：github:course/avl.py

```
1 from binarySearchTree import TreeNode, BinarySearchTree
2 class AVLTreeNode(TreeNode):
3     def __init__(self, key, val, parent=None):
4         super().__init__(key, val, parent=parent)
5         self.height = 1
6
7     def renew_height(self):
8         l = self.leftChild.height if self.leftChild else 0
9         r = self.rightChild.height if self.rightChild else 0
10        self.height = 1 + max(l, r)
11
12    def balance(self):
13        l = self.leftChild.height if self.leftChild else 0
14        r = self.rightChild.height if self.rightChild else 0
15        return l - r
```

AVL 树的实现：put 方法

- 重新定义 `_put` 方法即可
- `_put` 用递归实现：新增结点后自下而上检查平衡因子，实际只需要平衡一次

```
39     def _put(self, key, val, curr):  
40         if curr is None:  
41             self.size += 1  
42             return self.TreeNode(key, val)  
43         if key < curr.key:          #递归左子树  
44             curr.leftChild = self._put(key, val, curr.leftChild)  
45             curr.leftChild.parent = curr  
46         elif key > curr.key:      #递归右子树  
47             curr.rightChild = self._put(key, val, curr.rightChild)  
48             curr.rightChild.parent = curr  
49         else: #key == curr.key  
50             curr.payload = val      #更新结点，无新增  
51             #curr = super().__put__(key, val, curr)  
52         return self.checkBalance(curr)
```

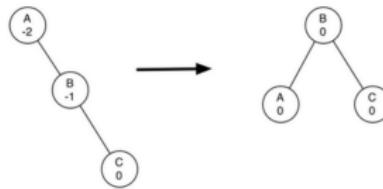
AVL 树的实现: checkBalance 方法

- 在递归过程自下而上，逐一检查
- balanceFactor 分为-2(R+), -1(R), 0, 1(L), 2(L+) 五种情况
- 通过左右旋转的办法平衡

```
21     def checkBalance(self, curr):  
22         """  
23             检查curr结点的平衡因子，并负责再平衡。  
24             返回平衡之后新子树的根结点，由caller维护父结点对他的指向  
25         """  
26         bal = curr.balance()  
27         if bal > 1: #L+  
28             if curr.leftChild and curr.leftChild.balance() == -1 : #L+R  
29                 curr.leftChild = self.rotateLeft(curr.leftChild)  
30                 curr = self.rotateRight(curr)  
31         elif bal < -1: #R+  
32             if curr.rightChild and curr.rightChild.balance() == 1: #R+L  
33                 curr.rightChild = self.rotateRight(curr.rightChild)  
34                 curr = self.rotateLeft(curr)  
35         else:      #没有发生旋转，子树的高度也需要重算  
36             curr.renew_height()  
37         return curr
```

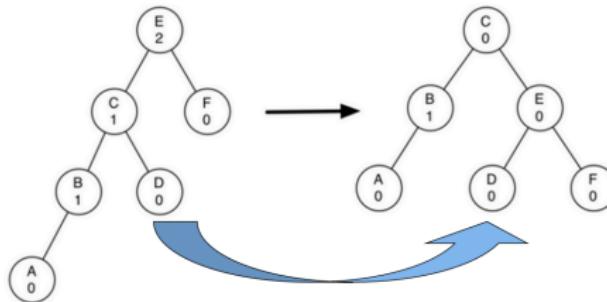
AVL 树：平衡的实现

- 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，同时更新相关父结点引用
 - 更新旋转后被影响结点的平衡因子
- 如图，是一个“右重”子树 A 的左旋转（并保持 BST 性质）
 - 将右子结点 B 提升为子树的根
 - 将旧根结点 A 作为新根结点 B 的左子结点



AVL 树：平衡的实现，子树移植

- 前面没考虑到的情况：如图的“左重”子树右旋转，新根结点原来有右子结点。
 - 旋转后，新根结点 C 将旧根结点 E 作为右子结点，新根结点原来已有右子结点不能扔掉，需要给它重新找位置！
 - 新根结点 C 原有的右子结点 D 移植到旧根结点 E 的左子结点
 - 旧根结点 E 的左子结点 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，其他结点为根的子树没有变化

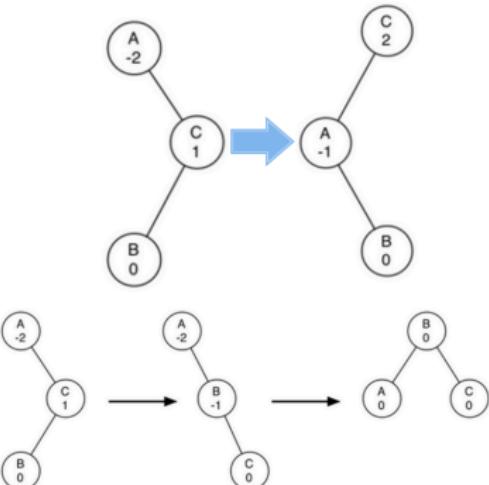


AVL 树的实现: rotateRight 代码

```
83     def rotateRight(self, rotRoot):          #旧的和新的根结点
84         o, n = rotRoot, rotRoot.leftChild      #移植新根结点的右子结点
85         tr   = n.rightChild
86
87         n.parent = o.parent                  #新根上连
88         n.rightChild, o.parent = o, n        #重建新旧根之间的联系
89         o.leftChild = tr                    #-->到旧根的左子结点
90         if tr : tr.parent = o
91
92         #新 / 旧根的高度需要重算
93         o.renew_height()
94         n.renew_height()
95         #返回新根结点
96         return n;
```

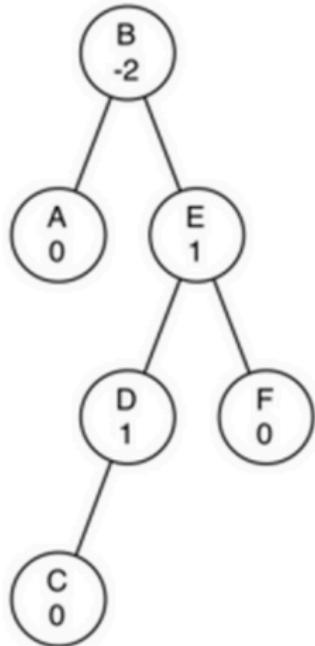
AVL 树的实现：更复杂的情形

- 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结点的因子
 - 如果右子结点“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结点的因子
 - 如果左子结点“右重”的话，先对它进行左旋转
 - 再实施原来的右旋转



AVL 树的平衡：操练一下

- 如图所示的“右重”子树如何平衡？
- 请图示旋转过程
- 一个重要现象：所有的平衡操作都把子树的高度“降一”
- 对于由 `put` 引起的不平衡，“降一”刚好恢复了 `put` 之前的树高，不会影响上一级子树的平衡
- 由 `delete` 引起的不平衡，“降一”可能减少了上一级子树的高度，产生连锁的不平衡



AVL 树的实现: delete 方法

- 从发生 remove 的结点开始，沿路经向上逐个检查平衡。
- 如果发生旋转，子树的根结点会被换掉，需要重建其父结点对它的连接

```
43     def delete(self, key):  
44         nodeToRemove = self._get(key, self.root)  
45         if nodeToRemove:  
46             curr = self.remove(nodeToRemove) #得到remove结点的父结点  
47             self.size -= 1  
48             while curr :  
49                 if curr.isLeftChild():  
50                     curr = self.checkBalance(curr)  
51                     curr.parent.leftChild = curr  
52                 elif curr.isRightChild():  
53                     curr = self.checkBalance(curr)  
54                     curr.parent.rightChild = curr  
55                 else:                      #isRoot  
56                     curr = self.checkBalance(curr)  
57                     self.root = curr  
58                     curr = curr.parent  
59             else:  
60                 raise KeyError("'{0}' notFound.".format(key))  
61
```

AVL 树的实现：结语

- 经过复杂的方法，AVL 树始终维持平衡性质，树的高度为 $O(\log n)$
- 这样 get 方法也始终保持 $O(\log n)$ 的高性能
- put 和 delete 方法的代价有多大？
 - 首先同 put 方法一样，需要 $O(\log n)$ 的代价找到新结点插入的位置，或者需要删除的结点
 - 其次需要 $O(\log n)$ 次 checkBalance，每次的代价是 $O(1)$
 - 综合起来，put 和 delete 方法的代价也是 $O(\log n)$
- 算法优化的目标：
 - 1 让最常用的操作最快
 - 2 重点改进复杂度较高的操作
 - 3 甚至可以牺牲一点低复杂度操作的性能

ADT Map 的实现方法小结

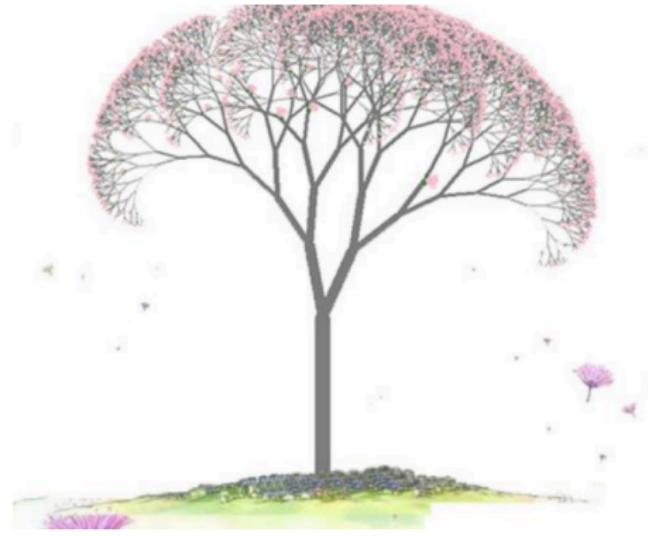
- 我们采用了多种数据结构和算法来实现 ADT Map，其时间复杂度数量级如下表所示：

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
put	$O(n)$	$O(1) \rightarrow O(n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$
get	$O(\log_2 n)$	$O(1) \rightarrow O(n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$
in	$O(\log_2 n)$	$O(1) \rightarrow O(n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$
del	$O(n)$	$O(1) \rightarrow O(n)$	$O(\log_2 n) \rightarrow O(n)$	$O(\log_2 n)$

- 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，一个结点存很多 key，最大可能限制树的深度

目录

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



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，它在父结点的位置“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。
 - 二叉树叶结点的定义：没有“非空”子结点。

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 的子结点 E、F、G、H、I 组成“线性表”。
 - E 为 B 的最左子结点
 - 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

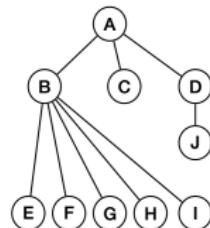
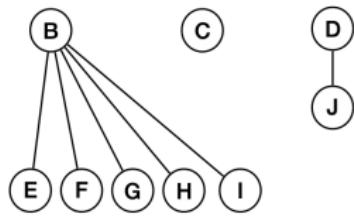
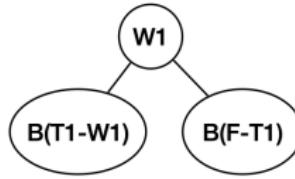


Figure: 森林



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 $F = \{T_1, T_2, \dots, T_n\}$ 是树 T_1, T_2, \dots, T_n 组成的森林，递归转换成二叉树 $B(F)$:
 - 若 F 为 \emptyset , 即 $n = 0$, 则 $B(F)$ 为空树
 - 若 F 非 \emptyset , 即 $n > 0$, 则:
 - $B(F)$ 的根结点是森林中第一棵树 T_1 的根结点 W_1
 - $B(F)$ 的左子树是 T_1 去除 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: $B(F - T_1)$

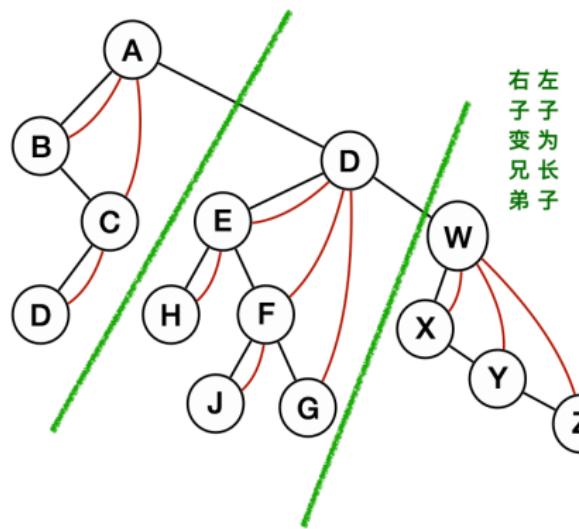


森林与二叉树的等价转换

- 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 $F(B_L)$



- 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
- 通过构建哈夫曼树对字符进行编码
- 用二叉堆实现了优先队列
- 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
- 二叉树与森林及树之间的互相转换