树和树算法

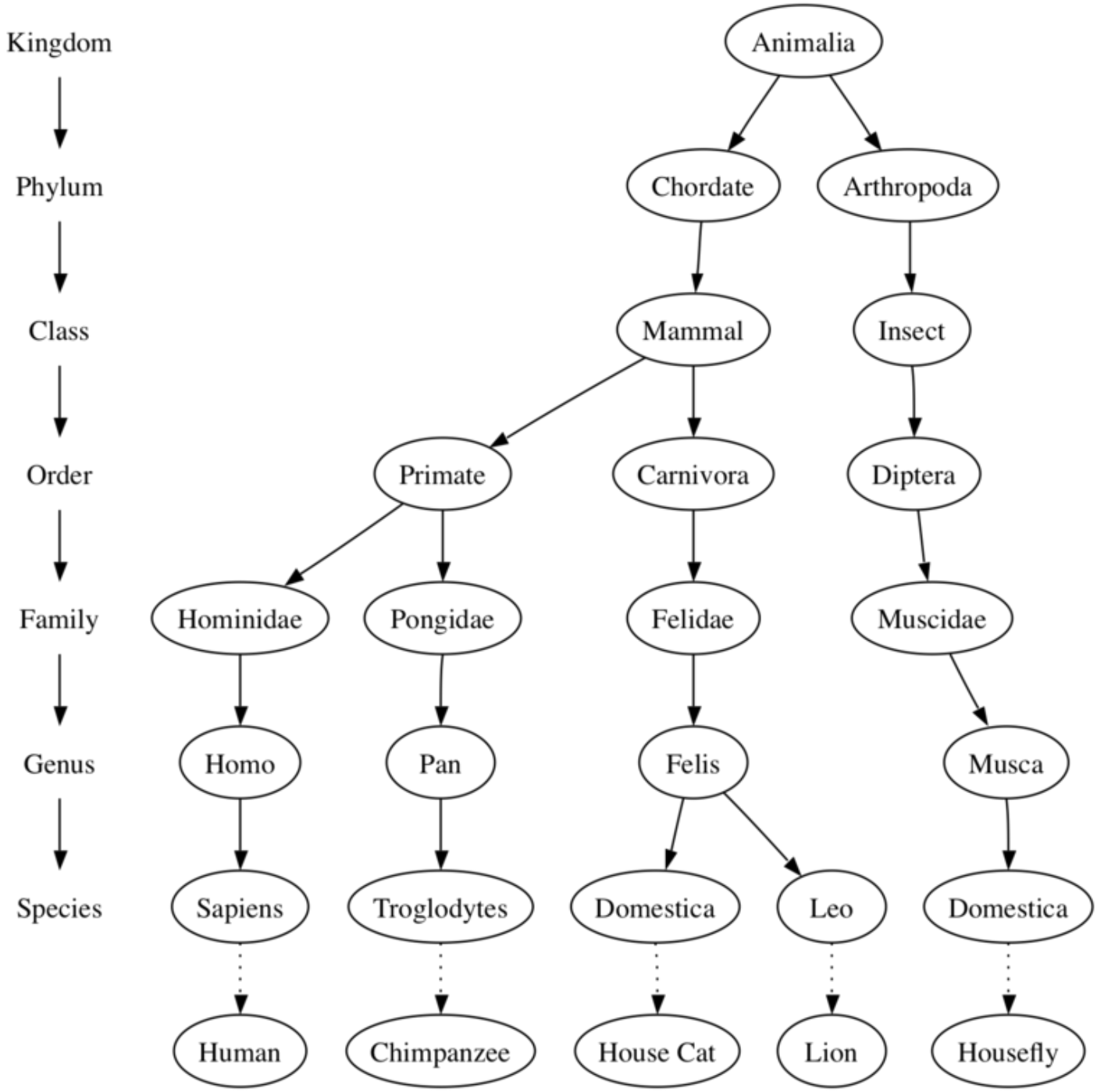
6.1 目标

- 了解树形数据结构以及其使用方式。
- 了解树如何实现Map数据结构的。
- 使用列表来实现树。
- 使用类和引用来实现树。
- 以递归数据结构的形式来实现树。
- 使用堆来实现优先队列。

6.2 树的例子

到目前位置，读者已经学习了一些线性数据结构，比如栈、队列，对于递归也有一些了解了，接下来研究一种常见的数据结构————**树（tree）**。树用于计算机科学的很多领域，包括操作系统，图像，数据库系统以及计算机网络。树形数据结构和自然界中的树有许多相似点。树形数据结构也有根、枝、叶，不同之处在于自然树根在底，而计算机科学中的树形数据结构的根则在顶部，而叶在底部。

在研究树形数据结构之前，先看一些常见的例子。第1个例子是生物学中的分类树。从这个简单的例子可以看出一些树的属性。第一个属性是，树是一种分级结构，即树是按层来构建的，并且顶部更具广泛性，而底部更加具体。最顶层的是界，其下一层是门，再下层是类，以此类推。然而，无论细分到哪一层，所有的生命体都是动物。



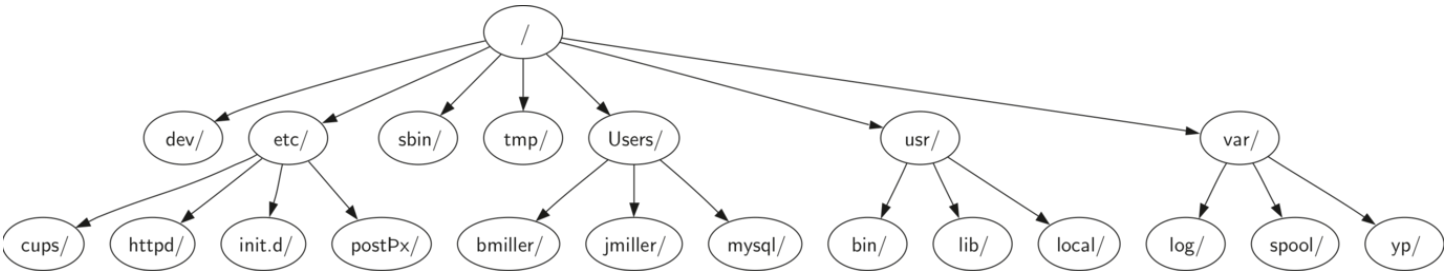
image

注意，可以从树的顶部出发沿着某条由圆圈和箭头构成的路径到达底部。在树的每一层，都可以判断一个问题，然后继续沿着符合答案的路径走下去。

树的第二个属性是，某个节点的子类是跟另一节点的子类无关的。比如说，猫属有家养和野生两个子类，蝇属也有家养和野生两个子类。然而这两个显然是完全无关的。这意味着，更改猫属的子类节点并不会影响蝇属的子类。

第三个属性是，每片叶子都是位移的。从树根出发可以找出且仅能找出一条路径到达某个物种。比如说，动物界-脊椎动物-哺乳动物-食肉动物-猫科-家猫。

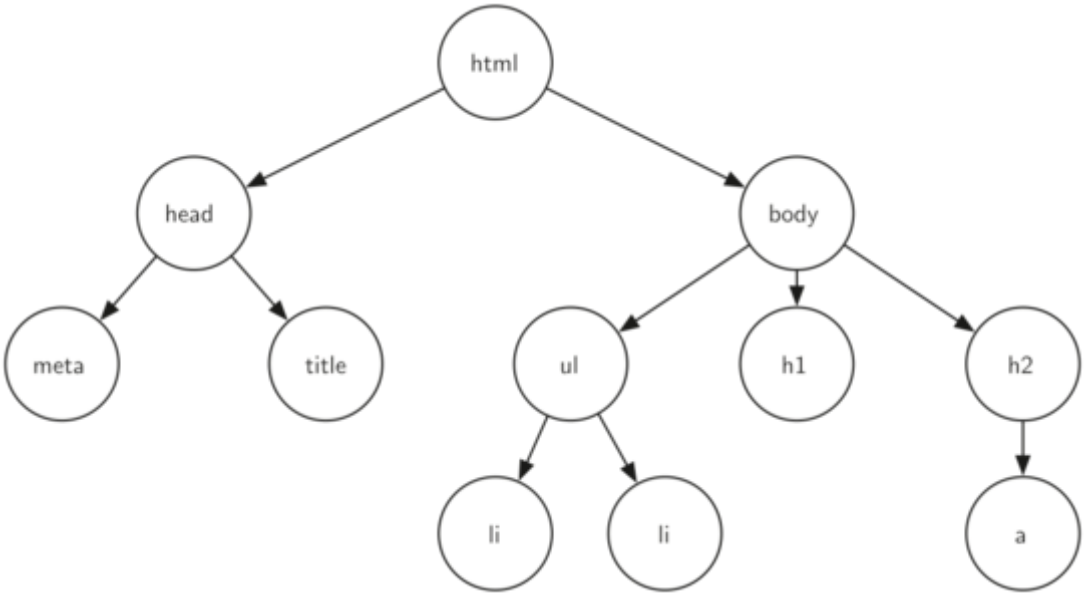
另一个读者每天都接触到的树形结构就是文件系统。在文件系统中，目录或者说文件夹是按照树形来构建的。图2是Unix文件系统层级。



文件系统与生物分类树有很多相似点，可以沿着一条路径到达任意目录，该路径唯一地指明了该目录及其内部的文件。树的另一个重要属性，来源于其分级性，就是可以将树的某个部分（被称为**子树（subtree）**）移到另一个位置而对该层以下不造成影响。比如说，可以将整个以/etc/开始的子树从根目录移除，将其连接至usr/，这也会改变httpd的路径名，由/etc/httpd变为了/usr/etc/httpd，但并不会影响其内容以及httpd的子类。

最后一个例子是网页。以下是一个简单的使用HTML的网页。图3是与该网页对应的树形结构。

```
<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>
```



image

HTML源代码和与其对应的树型结构说明了另一种分级结构。注意到，树的每一层都对应了HTML标签内的嵌套。源代码的第一个标签是<html>，最后一个是</html>，页面内所有其它的标签都位于这对标签对内。

6.3 术语及定义

现在读者已经接触了几个树形的例子，现在来正式给出树及其构成的定义。

节点（Node）
节点是树的基本构成。它可以有名字，称之为“**键（key）**“。节点也可附带其它信息，这些信息被称为”**负载（payload）**“。尽管负载信息与许多算法没什么关系，但它通常是应用树形结构的关键。

边（Edge）
边是树的另一种基本构成。边可以将两个节点连接起来，表示这两者之间存在某种关系。每个节点（除了根节点）都有且仅有1条来自于其它节点的入边。每个节点可能有多能出边。

根（Root）
树的跟节点是唯一一个没有入边的节点。

路径（Path）
路径是由边连接起来的节点的有序列表。

子节点（Children）
对于某个节点集，若其入边都来自同一个节点，则该节点集中任意一个都是该节点的子节点。

父节点（Parent）
对于某一个节点来说，它是其外边连接的所有节点的父节点。

兄弟节点（Sibling）
同一父节点的子节点互为兄弟节点。

子树（Subtree）
子树是某个父节点及其所有后代的所有节点、边构成的集合。

叶节点（Leaf）
没有子节点的节点被称为叶节点。

层数（Level）
某一节点的层数n是指从根节点到达该节点的路径的边数。

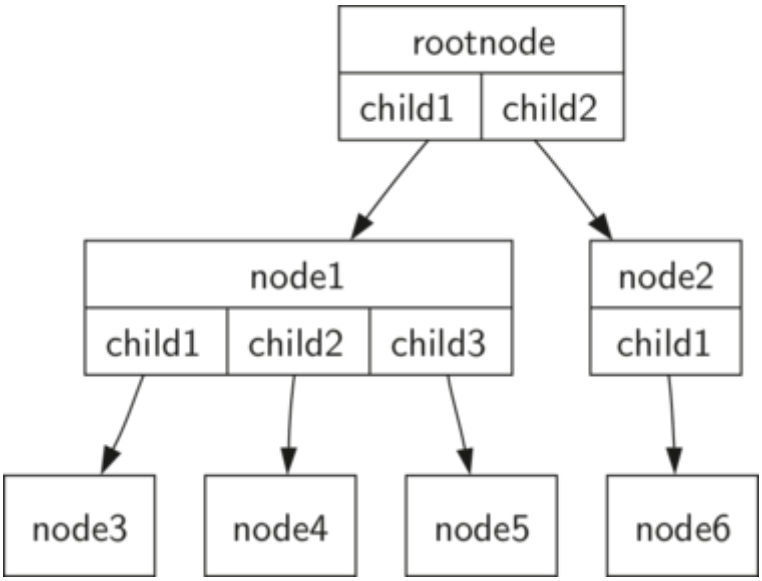
高度（Height）
树的高度是指树中节点层数的最大值。

使用以上术语，现在可以给出树的正式定义了。实际上，树有两种定义。一种定义由节点和边给出，另一种则是递归定义。

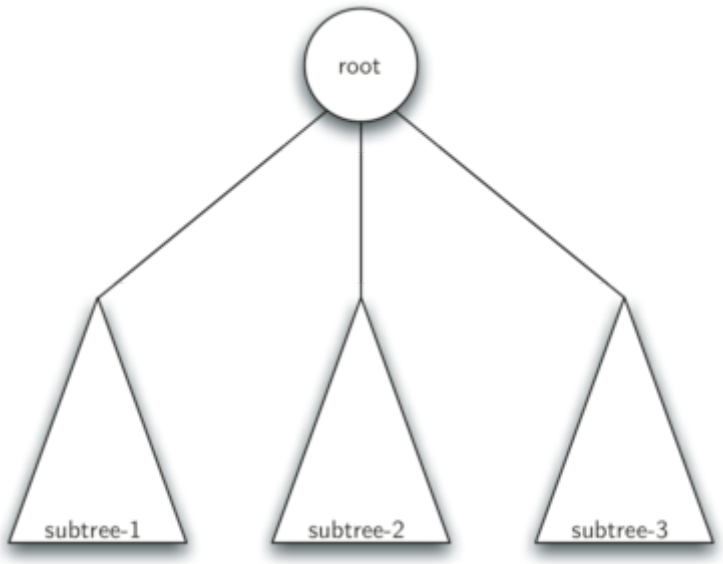
定义1:树是节点及连接节点的边所构成的集合。树具有以下性质：

- 树中必有1个根节点。
- 每个节点n，除了根节点外，都有且仅有1条边外边与另一节点p相连，其中p是n的父节点。
- 从根部出发，有且仅有1条路径到达任意节点。
- 若某棵树的任意节点之多有两个子节点，则称该树为**二叉树（binary tree）**。

图3是一颗符合定义1的树。边上的箭头表明连接方向。



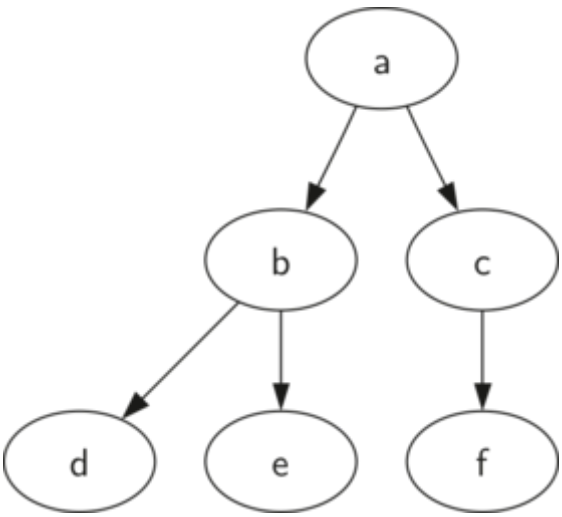
定义2:树要么为空，要么包含1个根节点以及0个更多子树。每个子树的根节点都通过边与父树的根节点相连。图4树的递归定义的一个示意图。使用树的递归定义，可以确定图4中的树至少有4个节点，因为每个代表子树的三角形都有一个根节点。它可能有不止4个节点，但是现在并不确定，除非进一步地向下走。



image

** 实现：嵌套列表 **

对于用嵌套列表实现的树，这里先用Python的list数据结构来实现之前定义的函数。虽然与之前实现的抽象数据类型有点不同，这里将接口写成列表上的一系列方法，但是这样做是很有趣的，因为它提供了一种简单的递归数据结构，便于直接观察和测试。在嵌套列表实现树时，将列表中的第1个元素作为根节点，第2个元素本身也是个列表并且被作为左侧的子树，第3个元素同样也是个列表并且代表了右侧的树。以一个例子来说明该存储结构，如图1所示的简单树，其对应的列表实现。



../_images/smalltree.png

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

```
        [] ]
      ]
```

注意，可以通过标准的列表缩影来访问列表中的子树。树的根节点是myTree[0]，左侧的子树是myTree[1]，右侧的子树是myTree[2]。可执行代码1演示了使用列表来生成一个简单的树。一旦树被创建，便可以访问根节点，左节点，右节点。嵌套列表实现的一个很好的属性为，使用列表的结构来表示子树是与树本身定义的结构相符的，即递归结构。若子树有1个根节点值，并且有2个空列表，那么它就是个叶节点。嵌套列表实现的另一个有点是它对于多叉树也是适用的，此时，另一个子树不过是另一个列表而已。

可执行代码1:适用索引来访问子树

```
myTree = ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c', ['f',[],[]], []]
]
print(myTree)
print('left subtree = ', myTree[1])
print('root = ', myTree[0])
print('right subtree = ', myTree[2])
```

现在根据树形数据结构的定义来进一步完善，通过提供一些函数，可以更简便地将列表作为树来使用。注意，这里并不是在定义二叉树类，只是给出的函数只是用来像操作树一样来操作标准列表。

```
def BinaryTree(r):
    return [r, [], []]
```

Binary函数只是构造了一个列表，它具有1个根节点和2个子列表用来表示子节点。要实现向树的根节点加入1个子树，需要向第2个位置插入1个新的列表，这必须小心处理。如果第2个位置已经有东西了，需要对其进行跟踪，并将其作为新加入的列表的左侧子节点。

代码1

```
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

注意，当插入1个左侧子节点时，首先应当获取当前左侧子节点的列表（可能是空的），然而将新的左侧子节点插入，将原来的左侧子节点作为新节点的左侧子节点。这样便可以将新节点拼接至树的任意位置。insertRight与insertLeft类似，如代码2所示。

代码2

```
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root
```

为了完善这组创建树形结构的函数，再来写一些用于get和set根节点值的函数，当然也左侧和右侧子树的也要有。

代码3

```
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]
```

可执行代码2对编写的函数进行了试用。读者应该自行尝试。

可执行代码2:演示树形结构基本函数的Python代码块

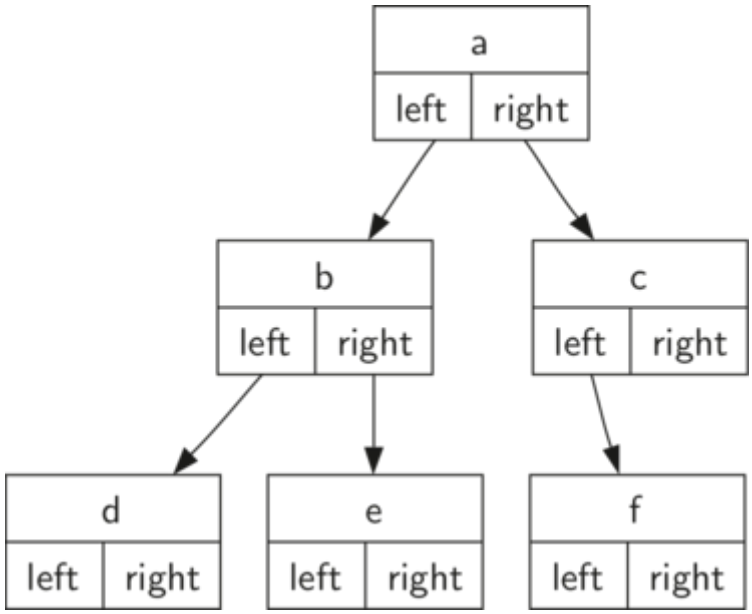
```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
```

```
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    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]

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)))
```

6.5 节点和引用

第二种方法使用节点和引用来实现树，现在将定义一个类，它有根节点值、左子树、右子树3个属性。因为这种实现方式更加符合面向对象编程模式，在本章的剩余内容也将采取这种方式。

使用节点和引用，可以按图2所示来考虑树。



image

先给出使用节点和引用实现的类的简单模式，如图4所示。记住，属性中的left和right是指向其它BinaryTree类的实例的引用。比如说，当插入1个新的左子节点到该树时，便又创建了另一个BinaryTree的实例，并且将self.leftChild的根节点指向新树。

代码4

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

注意在代码4的构造器函数中，根节点其实可以是对象。就像列表中可以存放对象一样，根节点也可以是指向任何对象的引用。在之前的例子中是将根节点的名字作为根节点值存储起来的。使用节点和引用来实现的如图2所示的树，仅需要创建6个BinaryTree类的实例。

接下来考虑下根节点之外的其它需要实现的函数。为了向树添加左节点，需要创建1个新的对象并且将根节点的left属性设置为指向新对象的引用。insertLeft如代码5所示。

代码5

```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```

插入操作必须要考虑两种情况。第1种是该节点没有左子节树。如果没有左子节点树，直接在该树上再加入1个节点即可。第2种是该节点已经存在1个左子树了。此时，插入1个节点并且将现有的子树降1级。第2种情况在代码5种通过一个else语句处理。

insertRight的代码也必须考虑类似的情况，如代码6所示。

代码6

```
def insertRight(self,newNode):
    if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t
```

为了完善二叉树数据结构的定义，还要编写访问左、右子节点和根节点的值的方法。

代码7

```
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
```

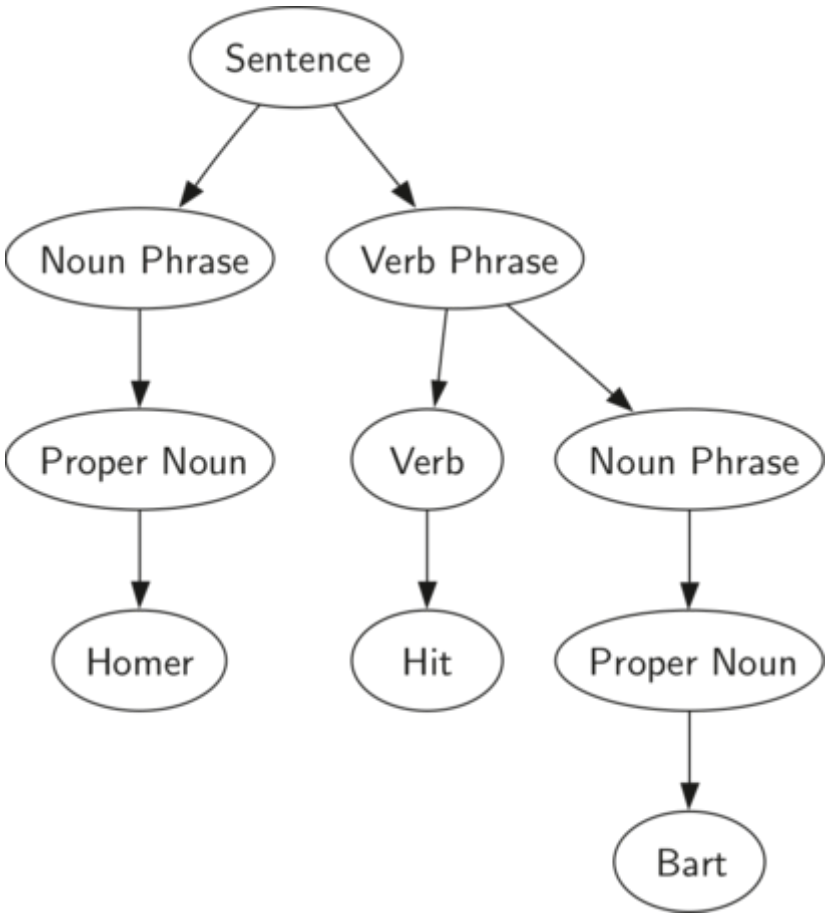
现在已经有了创建和操作二叉树所有必要的部分了，将它们组合起来并进行检验，如可执行代码1所示。
** 可执行代码1:使用节点和引用的实现 **

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

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t


    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


r = BinaryTree('a')
print(r.getRootVal())
```

```
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

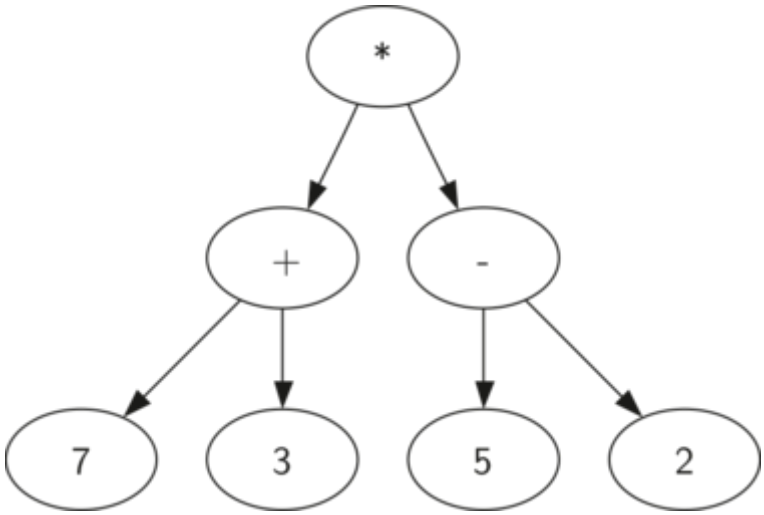
6.6 解析树

实现了二叉树后，现在来看一个用树解决实际问题的例子。本节将研究解析树，解析树可以用来表示真实世界的一些结构体比如句子、数学表达式。



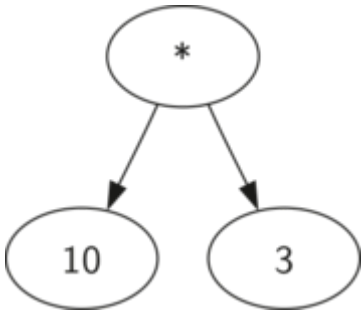
image

图1是1个简单句子的分层结构。将句子表示为树结构后便可按照子树的方式处理句子的各个独立成分。



image

也可以将数学表达式，比如说（7+3)*(5-2)表示为解析树，如图2所示。之前已经研究过全括号表达式，那么读者应该还记得关于这种表达式的一些知识，比如乘法比加法和减法的优先级更高。分层树有助于理解整个表达式的计算顺序。在计算顶层的乘法前，必须要计算出加法和减法的结果。加法，本例中的左子树，计算结果为10；减法，位于右子树，其结果为3。在计算出子树的结果后，利用树的分层结构，可以简便地将整个子树用1个节点来表示，如图3所示。



本节的剩余部分将对解析树作深入的测试，尤其是以下内容：

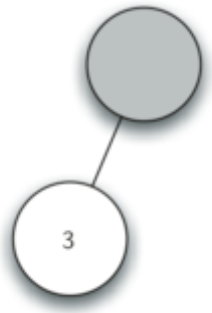
- 如何利用全括号表达式来建立解析树
- 如何计算解析树中的表达式
- 如何从解析树还原原始表达式

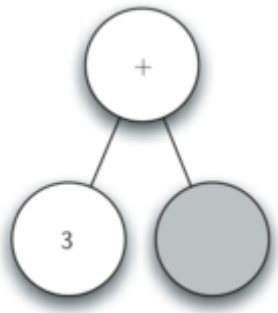
建立解析树的第一步是将表达式拆分为符号列表。有4种符号需要分开考虑：左括号，右括号，运算符号，运算数。可以确定，一旦读取到1个左括号，就意味着出现了1个新的表达式，应该为该表达式创建一个新的树。反之，当读取到1个右括号时，说明该表达式已经结束。另外，运算数将会称为其运算符的叶节点和子节点。最后，每个运算符都有1个左子节点和1个右子节点。

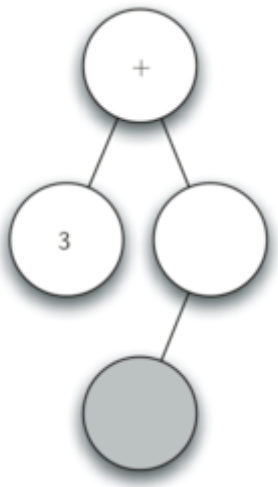
根据以上信息，可以给出以下4条规则：

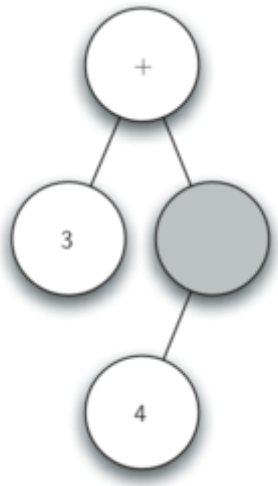
1. 若当前符号为'('，则插入1个新节点作为当前节点的左子节点，然后下移到该新的左子节点。
2. 若当前符号位于列表['+','-','/','*']中，将当前节点的根节点值设为该符号代表的运算符。在当前节点的右子节点加入1个新的节点，并且下移到该新的右子节点。
3. 若当前符号是1个数字，将当前节点的根节点值设置为该数字，并返回该节点的父节点。
4. 若当前节点是')'，返回该节点的父节点。

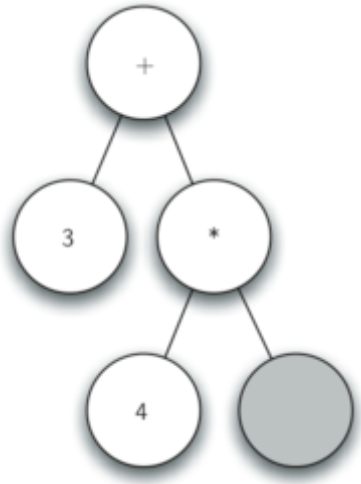
在编写Python代码之前，先看看以上规则下的运行实例。比如说（3+（45））。*将该表达式转化为列表*['(','3','+','(','4','',')',')']。先初始化1个根节点为空的解析树。图4给出了该解析树的结构和内容，以及每个符号的处理过程。

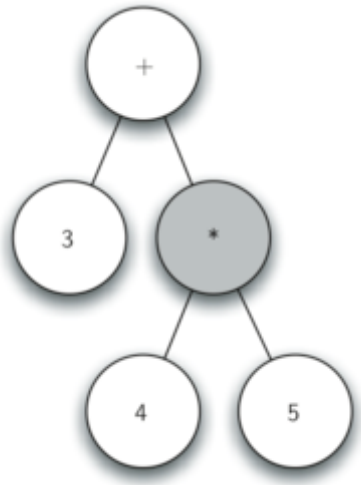












利用图4来一步一步地分析该例子：

1. 创建1个空树。
2. 读取'('作为第1个符号。根据规则1，为根节点创建1个左子节点，移动到该新节点。
   ...后续省略...

从上面的例子可以看出，要同时对当前节点以及当前节点的父节点进行追踪，树的接口提供了获取节点的子节点的办法，即getLeftChild和getRightChild方法，但如何跟踪父节点呢？一个简单的方法（在

遍历树时）是使用栈。当需要沿着当前节点向下移动时，首先将当前节点放入栈中。当需要访问当前节点的父节点时，将父节点从栈中pop出来即可。

使用上述规则，以及Stack和BinaryTree的操作，现在可以写出创建解析树的Python函数了，如可执行代码1所示。

** 可执行代码1:生成解析树 **

```
from pythonds.basic.stack import Stack
from pythonds.trees.binaryTree import BinaryTree

def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in ['+', '-', '*', '/', ')']:
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in ['+', '-', '*', '/']:
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError
    return eTree

pt = buildParseTree("( ( 10 + 5 ) * 3 )")
pt.postorder()  #defined and explained in the next section
```

生成解析树的4条规则被编写为了11，15，19，24行的if语句的前4个从句，代码对每条规则都进行了实现，并且在其中调用了BinaryTree和Stack方法。该函数中唯一的错误检测是通过else从句中报1个

ValueError错误，即列表中出现无法识别的符号时。

现在已经有了解析树，那么该怎么对其进行使用？作为第1个例子，可以编写一个函数计算解析树的值，返回数字结果。编写该函数需要利用树的分层性。回过头看看图2，回想一下，可以将原始树用一个简化的树（如图3）表示。这意味着可以编写一个算法，通过递归计算每个子树的值来计算出最终结果。

跟之前的递归算法一样，首先要给出递归计算函数的约束条件，很自然地，检测是否为叶节点可以是一个约束条件。在解析树中，叶节点必定是运算数，由于数字对象，比如整数和浮点数不需要再做解析了，evaluate函数直接将叶节点中存储的数值返回即可。通过在左侧和右侧子节点同时调用evaluate函数，使得递归不断向约束条件收敛。递归调用可以沿着树快速向叶节点移动。

为了将两个递归调用的结果合并，可以直接使用存储在父节点中的操作符对从子节点返回的两个值作运算。在图3的例子中可以看到，根节点的两个子节点的结算结果就是其本身，即10和3，对它们作乘法运算可以得到30。

递归的evaluate函数如代码1所示。首先获得当前节点的左、右子节点的引用。如果左右子节点都为None，那么可以确定当前节点是叶节点，行7对此作了检测。若当前节点不是叶节点，将当前节点的运算符应用到递归计算左右子节点所返回的结果。

为了实现数字计算，使用'+','-','*','/'作为键。字典中存储的值是来组于Pythond的operator模块的函数。operator模块提供许多常用运算符的函数版本。在字典中查询运算符并取得对应的函数对象，因为返回的对象是函数，可以通过一般的方式，即function(param1, param2)来调用。因此**opera['+'](2,2)**等价于operator.add(2,2)。

代码1

```
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()
```

最后，在图4建立的解析树中跟踪evaluate函数。当第1次调用evaluate时，将整棵树的根节点作为parseTree传入，然后获得左右子树的引用来保证它们的存在性。递归调用发生在行9。首先，根据树的根节点中的运算符进行查询，即'+'，而'+'映射到operator.add函数，它需要2个参数。同一般的Python函数调用一致，Python首先会计算传入该函数的参数，在这个例子中，这2个参数都是对evaluate函数

的递归调用。根据从左到右的计算顺序，第1次递归调用发生在左侧参数。然后发现该节点没有左右子节点，因此到达了1个叶节点，此时返回叶节点中存储的值作为计算结果，即3。

此时位于顶层的调用已经获得了1个参数传入operator.add，但还没有完。继续计算参数，对根节点的右子节点进行递归函数调用来计算值，该节点有左、右节点，查询该节点中存储的运算符可知为'*'，该函数又需要将该节点的左右子节点作为参数传入，可以发现，这两个子节点是叶节点，计算结果分别是4和5，那么现在有了这两个参数，计算operator.mul(4,5)的结果。现在顶层的调用operator.add的2个参数已经齐了，对其进行调用即operator.add(3,20)。

6.7 树的遍历

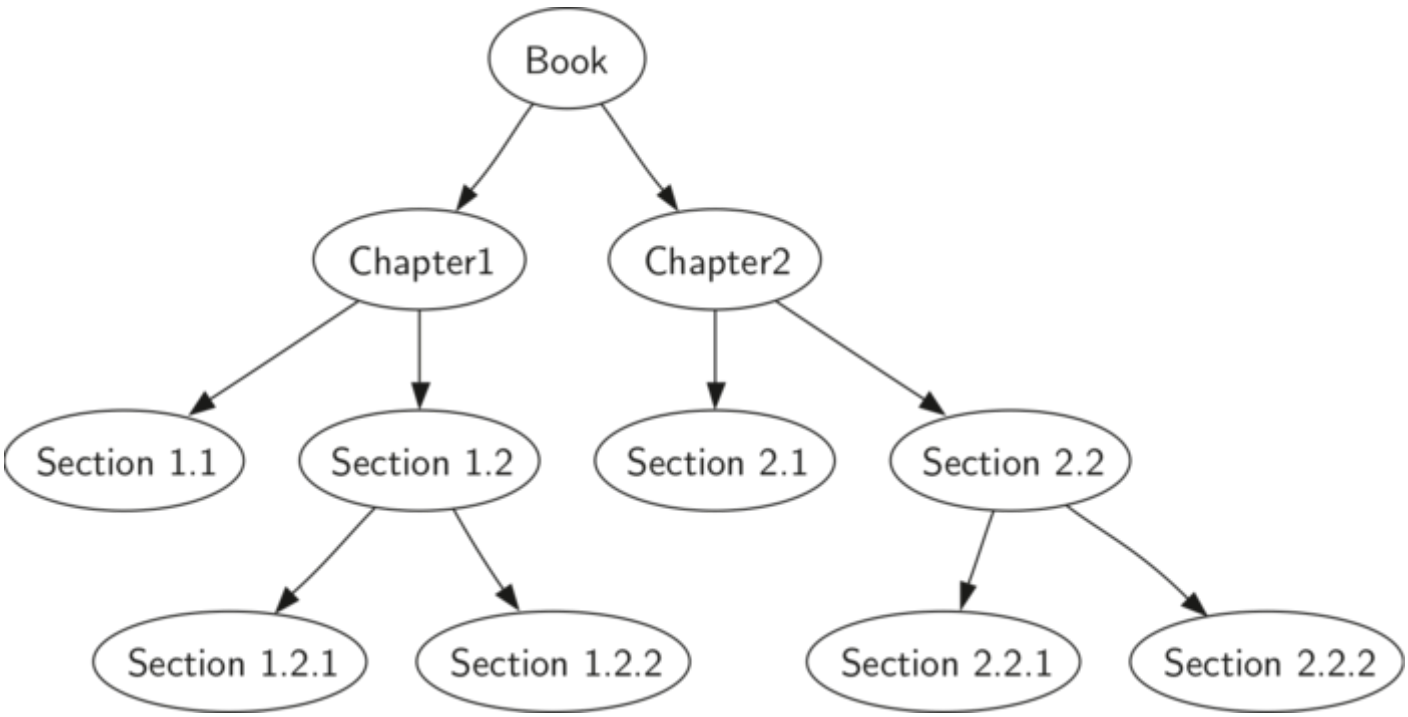
现在已经对编写的树形数据结构的基本功能进行测试，现在来看看树的一些其它运用模式，按照访问树的节点方式不同可以分为3种，这些模式的差别在于访问节点的顺序不同，这种对节点访问称之为**遍历（traversal）**。这三种遍历分别称之为**前序遍历（preorder）**，**中序遍历（inorder）**，以及**后序遍历（postorder）**。首先对这3中遍历作详细的解释，然后再看看其使用的一些例子。

前序遍历
按照前序遍历，首先访问根节点，然后前后对左子树和右子树递归地作前序遍历。

中序遍历
按照中序遍历，首先对左子树作中序遍历，然后访问根节点，最后对右子树作中序遍历。

后序遍历
按照后序遍历，数显对右子树作后序遍历，然后对右子树作后序遍历，最后访问根节点。

以下分别演示这3种遍历方式。
首先是前序遍历。以本书为例，书是该树的根节点，每1章都是该节点的子节点，章内的每1节都是该章的子节点，每1小节又是其对应的节的子节点，以此类推。图5是仅取了2章的示意图。注意，遍历算法对n叉树都应有效，这里暂时只关注二叉树。



image

假设读者要从头至尾阅读本书，前序遍历便刚好符合。从树的根节点出发（Book节点），按照前序遍历方式，对左子树递归地使用preorder，即Chapter1，然后再对该左子树递归地调用preorder，得到Section1.1，由于Section1.1没有子节点，因此不必再进行递归调用了，向上回到树Chapter1，此时还需要访问Chapter1的右子树，即Section1.2，类似地，现在到了Section1.2.1，然后访问节点Section1.2.2，这时Section1.2便完成了，回到Chapter1，再回到Book节点对Chapter2作相同的步骤。

由于这种遍历是用递归编写的，其代码会异常简洁，如代码2所示。

读者可能疑惑，前序遍历这种算法应该怎么来编写？应该写一个仅将树作为一种数据结构的函数，还是将其作为树形数据结构的一个方法？代码2中，将前序遍历写为以二叉树作为参数的外部函数，外部函数尤其简洁，因为其约束条件是检测树是否存在，如果树参数为None，该函数直接返回即可。

代码2

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

也可以将preorder实现为BinaryTree类的方法，如代码3所示。注意，将外部修改为内部时代码作了一些调整，总的来看，只是将tree参数换位了self。然而，约束条件也做了修改。内部方法在递归调用preorder前必须检测左、右子节点是否存在。

代码3

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

这两种实现方式哪个更优？在这种情况下，将preorder作为外部函数实现可能是更优的选择。原因是，编程中一般很少对树仅仅作个单纯的遍历，在绝大多数情况下，使用这种遍历模式时也会作一些其它的事情。实际上，在下个例子中可以看出，postorder跟之前写的用于计算解析树的代码很类似。因此，剩下的遍历模式都将作为外部函数实现。

postorder算法如代码4所示，跟preorder几乎等同，除了将print移动到函数的末尾。

代码4

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

读者已经遇到过后序遍历的使用了，即对解析树求值。再回过头看看代码1，当时是先对左子树求值，再对右子树求值，最后通过对运算符函数的调用将其在根节点合并。假设二叉树只存储表达式的树形数据，现在重写evaluation函数，但是尽量接近代码4中的postorder代码（如代码5）。

代码5

```
def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul,
'/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()
```

注意，代码4和代码5的形式几乎是一样的，但是在后者并没有使用print而是将结果返回了，这样便可以选择将递归函数的结果存储起来，然后将存储的值与运算符一并在行9使用。

最后一种将要研究的遍历方式是中序遍历。按中序遍历的方式，先访问左子树，然后是根节点，最后是右子树，如代码6所示。注意，这3种遍历函数在实现时都是仅改变了print语句相对另2个递归调用函数的位置。

代码6

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

如果对解析树使用简单的中序遍历，那么得到的是原始的表达式，然而并不会有括号了。现在对该初始中序遍历算法做些改进来得到表达式的全括号形式。只需以下调整即可：在对左子树递归调用前打印1个左括号，在对右子树调用完成后打印1个右括号。改进版如代码7所示。

代码7

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

注意，这里实现的printexp函数在每个数字两侧都会加上括号，这些括号显然是没有必要的。在本章最后的练习中，要求读者将printexp进行改进，来去除不必要的括号。

二叉堆实现优先队列

在之前的章节中，读者已经了解了一种先进先出的数据结构，即队列。队列的一种重要变体是**优先队列**。优先队列。优先队列同队列一样，也是从队首出队。然而，优先队列中，队列中各项的逻辑顺序由其优先级决定。最高优先级的项位于队首，而最低的则位于队尾。因此，当元素入队时，有可能会移至前端。优先嘟咧对于图像算法来说是一种很有用的数据结构。

读者也许已经相处了很多使用排序函数和队列的办法来实现优先队列。然而，向列表中插入元素是O(n)的操作，而排序则是O(nlogn)，实际上还可以优化。实现优先队列的经典方式是使用一种叫**二叉堆（binary heap）**的数据结构。二叉堆的入队，出队操作都是O(logn）。

二叉堆的有趣之处在于，当使用示意图来研究它时，它看起来就像二叉树一样，但是在实现它时，又只需要用1个列表。常见的二叉堆有两种：**最小堆（min heap）**，最小的键被放于队首；**最大堆（max heap）**，最大的键放于队首。本节将实现最小堆，最大堆就作为练习了。

6.9 二叉堆的操作

这里实现的二叉堆的基本操作如下：

- BinaryHeap()生成1个新的空二叉堆。
- insert(k)向堆中插入1个新元素。
- findMin()返回最小键所对应的项，该项仍留在堆中。
- delMin()返回最小键所对应的项，并从堆中删除该项。

- isEmpty()若堆为空则返回True，反之False。
- size()返回队中元素数。
- buildHeap(list)根据键的列表中生成1个新的二叉堆。

可执行代码1演示了二叉堆的一些方法。注意不论以何顺序向堆中添加元素，每次都会移除最小的项，接下来将堆其进行实现。

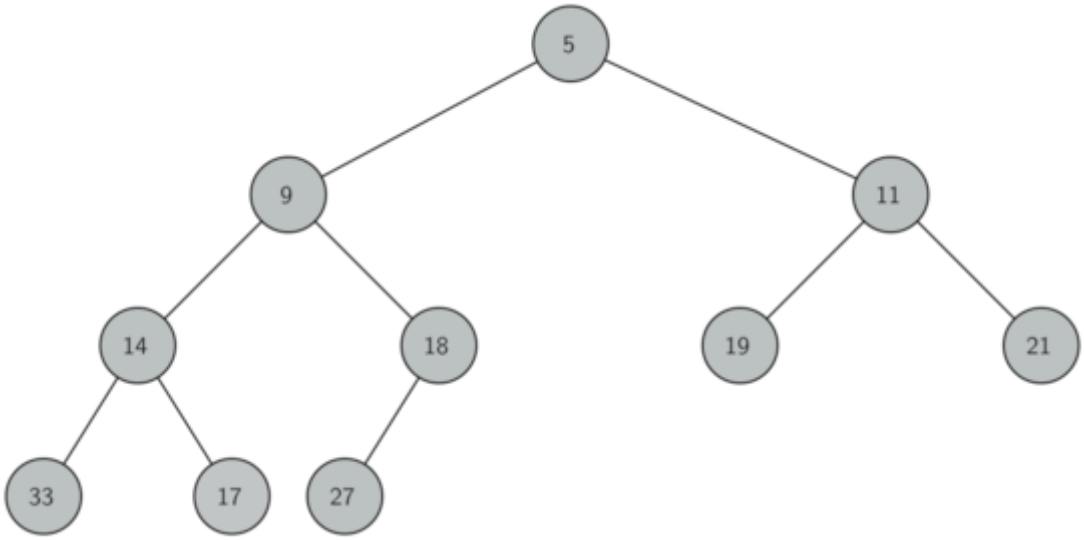
可执行代码1:使用二叉堆

```
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())
```

6.10 二叉堆实现

6.10.1 结构树形

要实现高性能的二叉堆，可以利用二叉树的对数复杂度特性来实现，而为了保证对数复杂度，必须要保证整棵树的平衡型。平衡二叉树在根节点的左、右子树两侧的节点数大致相同。这里的实现通过生成**完全二叉树（complete binary tree）**来达到树的平衡。完全二叉树指，树内各层的节点都有左、右两个子节点，除了最后一层，并且在最后一层保证优先填充左侧节点，如图1所示即为1个例子。

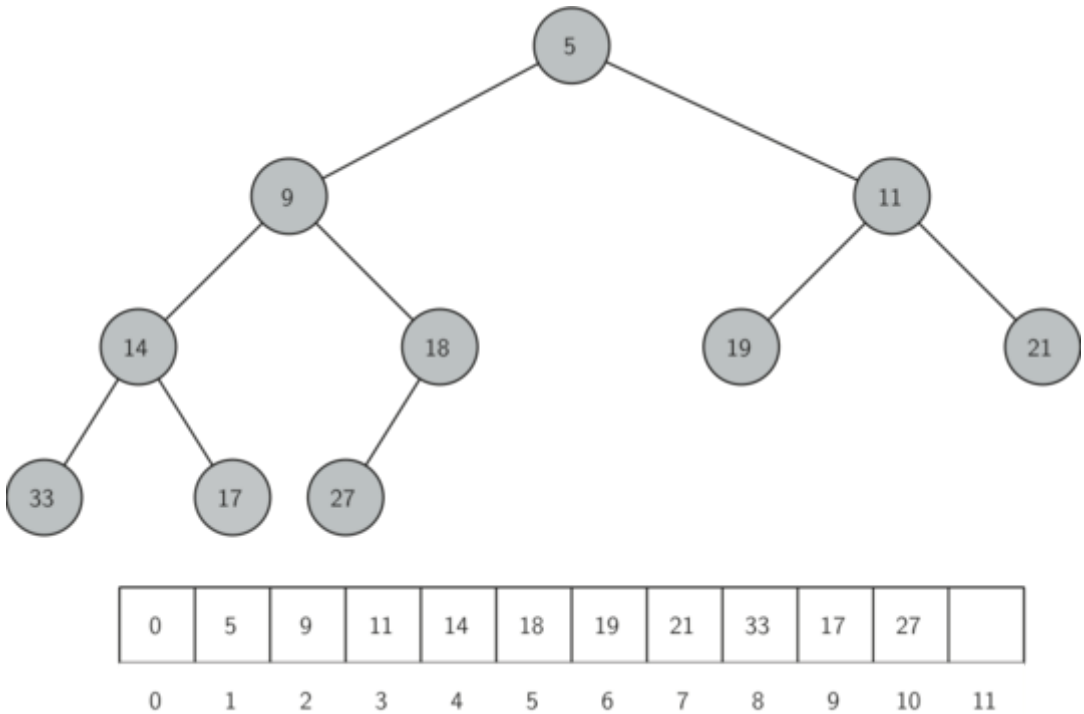


image

完全二叉树的另一个有趣的属性在于可以用1个列表将其实现，不需要使用节点和引用甚至嵌套列表。对于完全二叉树来说，父节点（位置为p）的左子节点位于列表中的2p位置，类似地，右子节点位于2p+1。从树中的任意节点出发，想要找到其父节点，直接使用Python的整数出发即可，若该节点在位置n，那么其父节点在n/2。图2是一颗完全二叉树，并且也给出了其对应的列表实现。注意父节点和子节点的2p和2p+1关系。二叉树的列表实现及其结构特征，使得对完全二叉树的遍历可以通过几个简单的数学操作高效完成。之后可以看到，这也使得二叉堆的快速实现得以完成。

6.10.2 堆的有序性

在堆中存放数据必须保证堆的有序性。堆的有序性是指：子堆中，对于任意节点x及其父节点p，p中的键必须小雨等于x的键。图2是1个具备有序性的完全二叉树。



image

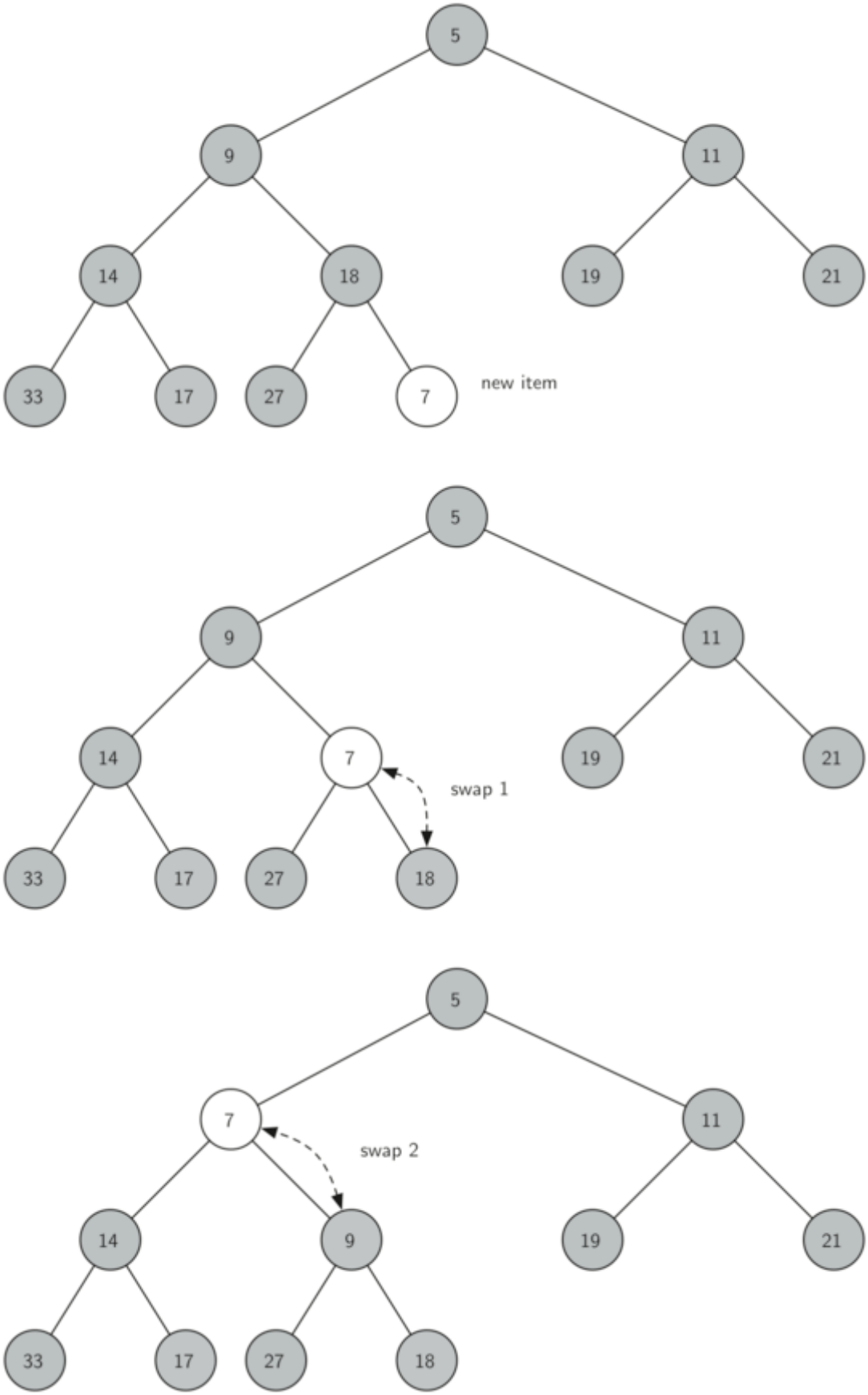
6.10.3 堆的操作

从构造器开始实现二叉堆。因为整个二叉堆都可以用1个列表表示，构造器需要做的仅是初始化该列表以及属性currentSize来跟踪堆当前的大小，如代码1所示。读者可以发现，空二叉堆在heapList的第1个元素为0，这个0实际上是不会使用，其意义在于在后面的方法中保证可以使用整数除法。

代码1

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

下1个要实现的方式是insert。最简单最快速地项列表中添加元素的方法是将该元素append至列表的末尾。使用append的好处在于可以保持完全二叉树的性质，坏处则是很容易破坏二叉堆结构性质。然而，可以写个方法通过将新元素与其父节点进行比较来维持二叉堆的结构特性。如果新元素比其父节点小，将之与其父元素互换位置。图2演示了新加入元素到达二叉树中正确位置所需要的一系列位置交换操作。



image

可以发现，通过将新元素上移，便可以实现新元素与其父元素之间的有序性，同时也维持了兄弟节点之间的有序性。当然，如果新元素非常小，必须还要将其交换到更高的层取。实际上，可能会到达树顶。代码2给出了percUp，将新元素向上移动到维持堆有序性为止。此时，heapList中的那个多余元素

便起作用了。注意，对于任意节点的父节点都可以使用简单的整数除法，当前节点的父节点可以通过将当前节点的索引值处以2得到。

现在着手编写insert方法（如代码3所示）。insert方法的大多数工作都在percUp中完成了。当新元素被append到二叉树中时，percUp对新元素进行处理并定位其正确位置。

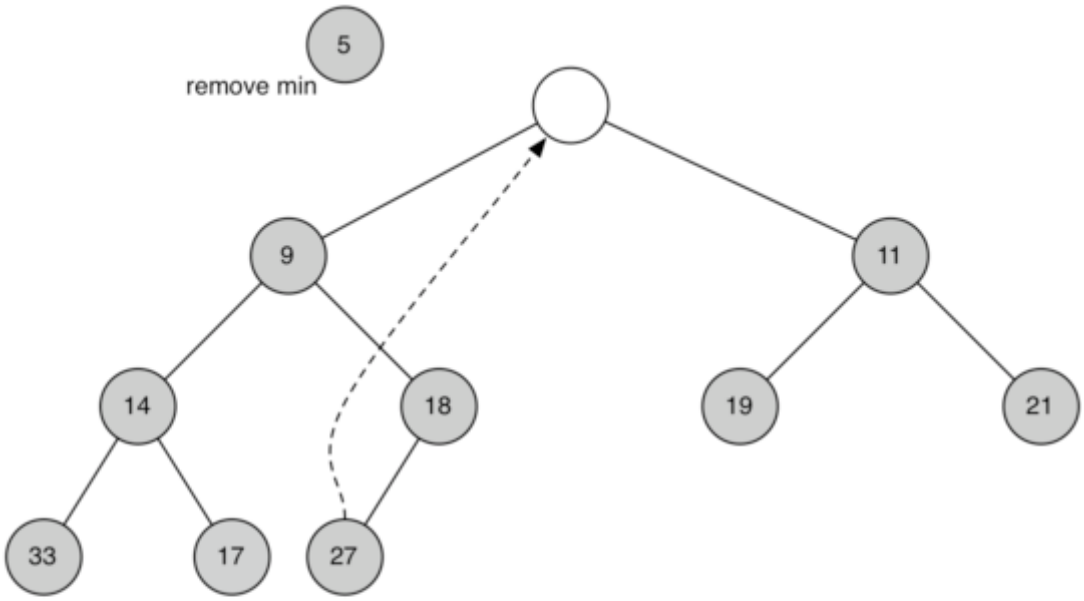
代码2

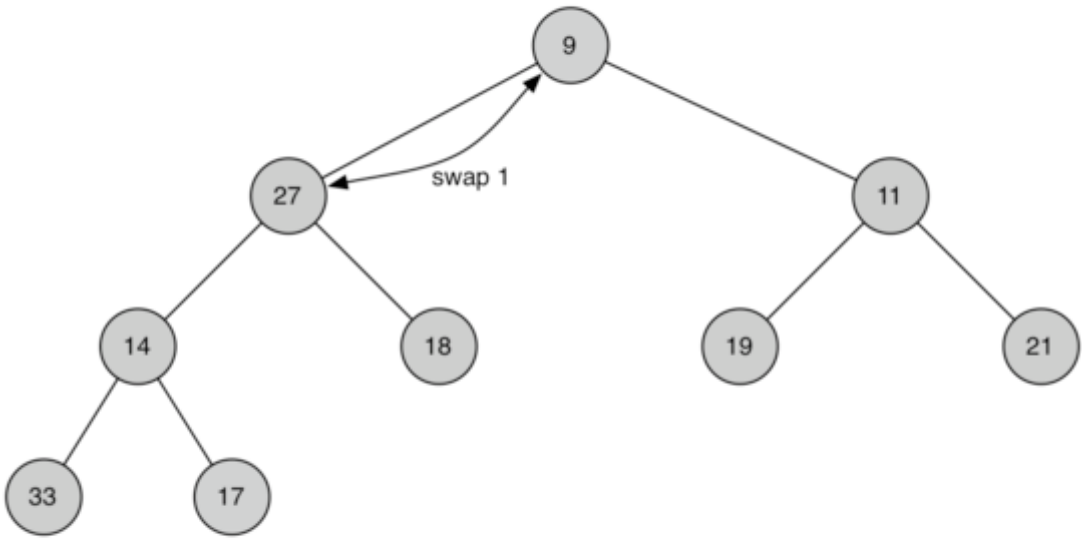
```
def percUp(self,i):
    while i // 2 > 0:
      if self.heapList[i] < self.heapList[i // 2]:
         tmp = self.heapList[i // 2]
         self.heapList[i // 2] = self.heapList[i]
         self.heapList[i] = tmp
      i = i // 2
```

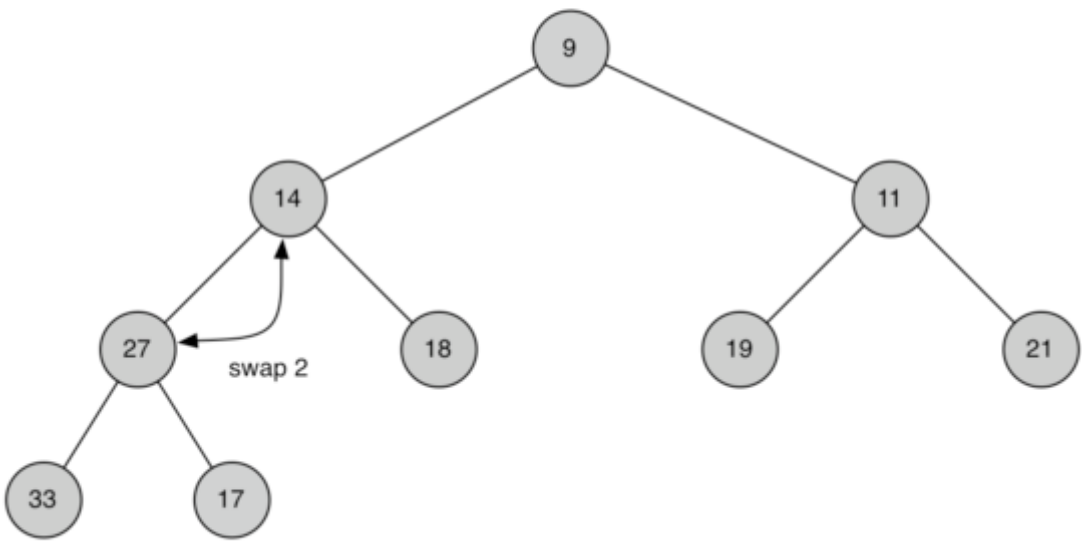
代码3

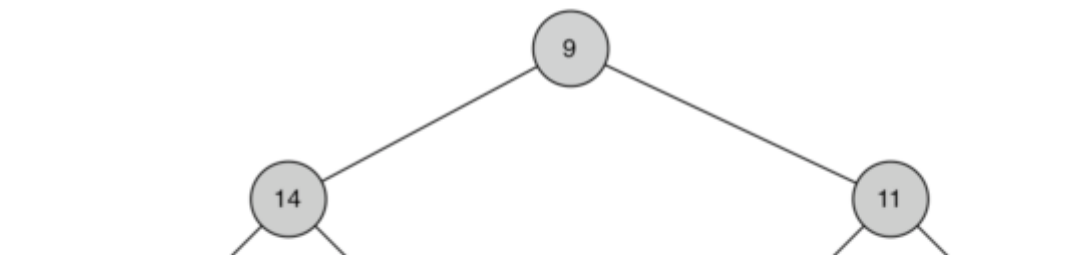
```
def insert(self,k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

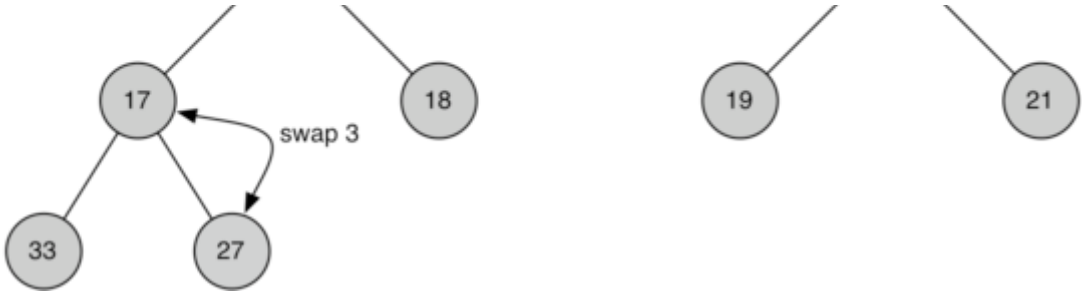
现在继续编写delMin方法。因为堆的有序性要求二叉树的根节点必须是树中最小的元素，找到最小项很容易。delMin的主要难度在于保证去除根节点后维持堆结构和其有序性，这可以通过两部完成。首先，将列表中的最后1项移动，并作为根节点，这样便维持了堆的结构性。然而这样可能会破坏二叉堆的有序性。因此，第二步将新的根节点下移到其正确位置。图3演示了将新的根节点移动其正确位置所进行的一系列位置交换。


5
remove min
9
11
14
18
19
21
33
17
27


9
27
swap 1
11
14
18
19
21
33
17


9
14
11
27
swap 2
18
19
21
33
17


9
14
11



image

为了维持堆的有序性，需要将新的根节点与比它小的子节点互换位置。在此之后，可能还需要继续这种交换直到该节点到达其正确位置。将元素下移的代码在代码4中的percDown和minChild方法中。

代码4

```
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
```

delMin如代码5所示。注意，主要工作也被辅助函数percDown完成了。

代码5

```
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
```

最后，实现一个从键列表来生成堆的方法。读者想到的第一个方法可能如下：给定某个键的列表，通过逐次插入键即可。由于列表中已有1个元素，且该列表是有序的，因此可以用二分搜索来下1个键的正确位置，消耗大概是$O(logn)$。然而，记住在列表的中部插入对象可能会导致O(n)的复杂度，所以将n个键插入整个列表中可能会需要O(nlogn)的复杂度。但是，如果直接以整个列表来生成，可以将复杂度控制为O(n)，如代码6所示。

代码6

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

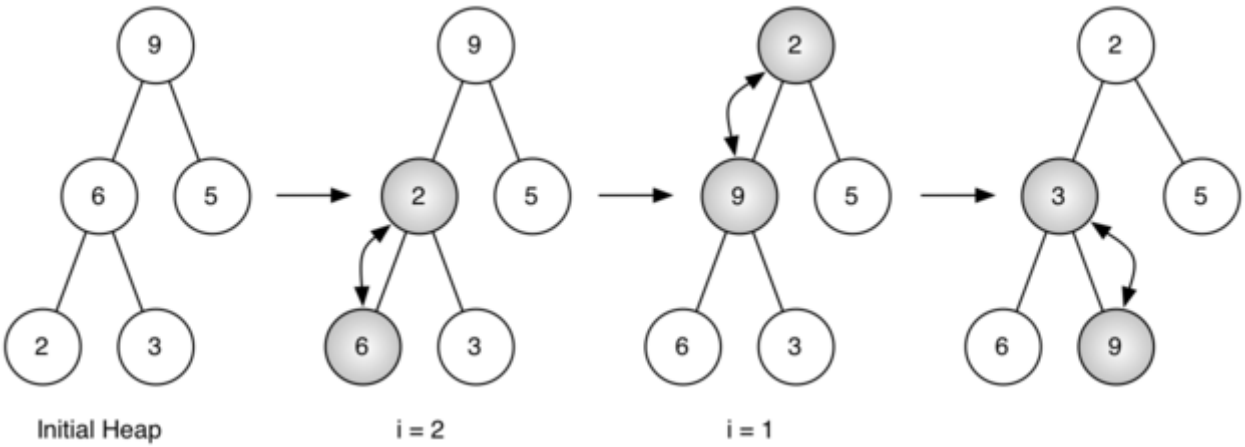


图4演示了buildHeap方法在堆初始二叉树[9,6,5,2,3]中的节点进行移动时所做的位置交换。尽管是从树的中部开始然后向上到达根节点，但是percDown方法保证了最大子节点必定向下移动。由于二叉堆是完全二叉树，任何经过了中点的节点都是叶节点因此没有子节点。注意当i=1时，从根节点开始下移，所以可能需要多次交换。如图4最右侧的2棵树所示，在开始的时候，首先将9从根节点下移，并且percDown会再次检测它目前的子节点，保证它到达正确的位置，最终与3进行了第2次位置交换。现在9已经到达树的最底层了，不可能再作位置交换了。

完整的二叉堆实现如可执行代码1所示。

可执行代码1:完全二叉树示例

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


    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

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

    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

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

bh = BinHeap()
bh.buildHeap([9,5,6,2,3])

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

断言二叉堆的时间复杂度为O(n)看上来有点让人迷茫，不过其证明超过了本书的范畴。但可以指出其关键在于，从树的高度来看，需要有一个logn的因子。buildHeap中的大多数操作都保证了树的复杂度低于logn。

利用二叉堆的生成时间为O(n)，作为练习，读者应当使用二叉堆来实现列表的O(nlogn)排序算法。

6.11 二叉搜索树

读者已经学习了多种从容器中获得键-值对的方法，回忆一下，这些容器实现了抽象数据类型Map。前文讨论过的两种Map的实现方式分别是列表二分搜索和哈希表。本节将学习**二分搜索树（binary search tree）**作为另一种实现键到值的映射的方法，在这里，元素在树中的准确位置并不重要，仅是使用二叉树结构来达到高效搜索。

6.12 二叉搜索树的操作

在研究其实现前，先回顾一下Map所提供的接口。读者可以看出，这些接口与Python的字典非常接近。

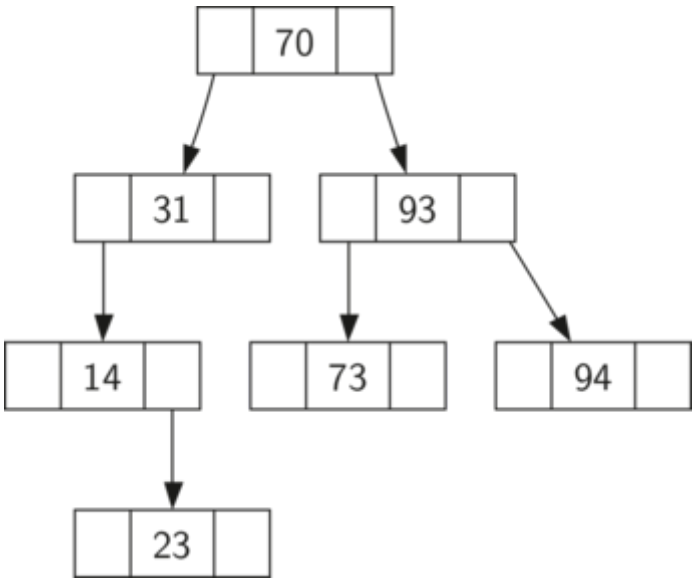
- Map()生成新的，空map。返回空map容器。
- put(key,val)将一组新的键-值对放入map中，如果键已在map中，则将旧值替换为新值。
- get(key)给定键，返回map中存储的值或者None。
- del 以del map[key]语句的形式删除键-值对。

- len() 返回map中存储的键-值对数量。
- in 以key in map语句的形式返回True或者False。

6.12 二叉搜索树的实现

二叉搜索树必须具有以下性质：左子节树的键都小于父类且右子节树的键都大于父类，即BST性质。在实现上述Map接口时，BST属性可以给予有效的指导。

图1演示了具有该性质的二叉树，其中仅展示了键而没有关联值。注意任意一个父节点和子节点组合都具有该性质。左子树中的所有键都比根节点的键小，右子树的所有键都比根节点的键大。



../_images/simpleBST.png

现在读者应该已经知道何为二分搜索树了，接下来研究其构造。图1所示的搜索树是按70,31,93,94,14,23,73的顺序插入后形成的节点。70是第1个插入元素的节点，因此将其作为根节点。接下来，由于31比70小，因此将其作为70的左子节点，再然后，93比70大，于是将其作为70的右子节点。现在便填充了树的两层了，下1个键要么是31或93的左节点或右节点。类似地，14比70和31小，因此将其作为31的左子节点，23也比31小，因此它必须位于31的左子树中。然而，它又比14大，因此将其作为14的右子节点。

为了实现二分搜索树，这里要使用类似于实现链表和表达式树时所使用的节点和引用。然 而，由于必须允许创建和处理空二叉搜索树，该实现要用到两个类。第1个类命名为BinarySearchTree，第2个命名为TreeNode。BinarySearchTree类有1个指向TreeNode的引用，这个TreeNode是二分搜索树的根节点。在大多数情况下都是在外部类定义外部方法来检测该树是否为空。如果树中有节点，该请求被直接传递给BinarySearchTree中定义的以树的根节点为参数的私有方法。当树为空或者要删除树根节点的键时，必须作小心处理。BinarySearchTree构造器及一些其它方法的代码如代码1所示。

代码1

```
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__()
```

TreeNode类提供了许多辅助函数来简化BinarySearchTee类中的方法。TreeNode的构造器及这些辅助函数如代码2所示。可以看到，代码中许多辅助函数用于帮助根据该节点作为子节点的位置（左或右）以及其子节点的类型来对其进行分类。TreeNode类也通过属性值来直接对其父节点进行跟踪，在后面可以看到这对于del操作符的实现很重要。

代码2中TreeNode的实现的另一个有趣之处是使用了Python的**可选参数（optional parameter）**。可选参数可便于在不同情况下生成TreeNode对象。有时需要创建已经有父节点和子节点的TreeNode，此时将它们作为参数传入即可。有时需要利用键-值对来创建TreeNode对象，并不会传入parent或者child，在这种情况下，可选参数使用其缺省值。

代码2

```
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

    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
```

现在已经有了BinarySearchTree作为shell，以及TreeNode，现在就来实现put方法。put方法是BinarySearchTree类的方法，该方法检测树中是否有根节点，如果没有，则put创建1个新的TreeNode对象并将其放置在树的根节点。若已经有根节点，put函数调用其私有的递归辅助函数_put来在树中根据以下算法进行搜索：

- 从树的节点出发开始搜索，在二分树中将新键与当前节点的键进行比较。若新键比当前节点小，则沿着其左子树继续搜索；若比当前节点大，则沿着其右子树继续搜索。
- 当没有左（或右）节点可供继续搜索了，便找到了新节点的正确安放位置。
- 创建1个新的TreeNode对象，将其插入上一步中发现的位置。

代码3给出了将新节点插入到树中的代码。_put函数按照上述步骤写作递归形式。注意，当新的子节点被插入到树中时，当前节点传递到新树中作为其父节点。

插入的实现中有1个很严重的问题是没有对重复键进行适合当处理。按照这种实现，出现重复键时会生成1个新的具有相同键的节点放置在具有相同键的节点的右侧。结果时，在搜索操作时该节点将不会被

找到。更好的处理方式是将该键对应的旧值替换为新的值。这个bug就作为练习留给读者了。

代码3

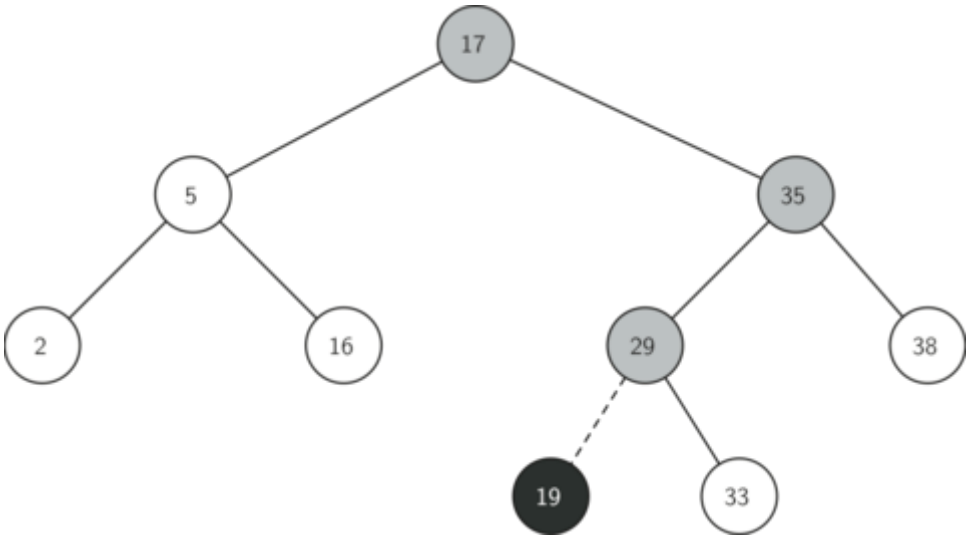
```
def put(self,key,val):
    if self.root:
        self._put(key,val,self.root)
    else:
        self.root = TreeNode(key,val)
    self.size = self.size + 1

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

写好了put方法，便可以通过让__setitem__方法调用put方法便可以实现[]操作符重载用于赋值，如此以来便可以使用myZipTree['Plymouth'] = 55446 这种语句了，就像Python的字典一样。

代码4

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

图2演示了将新节点插入到二分搜索树的过程。浅色阴影节点表示该节点在插入操作中将被访问。



../_images/bstput.png

在构造了树之后，下一个任务就是实现给定键后取出对应值了。get方法比put方法还要简单一些，因为它仅需要对树进行搜索直到到达不匹配的叶节点或者匹配的键。若找到了匹配的键，返回存储在节点的负载中的值。

代码5给出了get，_get和__getitem__方法的代码。_get方法的搜索代码使用与_put方法相同的左右节点选择逻辑。注意，_get方法返回1个TreeNode对象到get，这使得_get函数可以灵活地用于BinarySearchTree中的其它方法来对TreeNode中的负载以及其它数据进行处理。

通过实现__getitem__方法，便可以编写看起来就像字典访问一样的语句了，然而事实上却用的是二分搜索树，比如说z = myZipTree['Fargo']。可以看出，__get__方法做的唯一shicing就是调用get。

代码5

```
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)

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

利用get，可以通过编写__contains__方法来为BinarySearchTree实现in操作符。__contains__方法直接调用get，若get获取到值则返回True，反之则返回False。__contains__如代码6所示。

代码6

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

回想一下，__contains__会重载in操作符，于是便可以写出类似这样的语句了：

```
if 'Northfield' in myZipTree:
    print("oom ya ya")
```

最后，集中精力来完成二分搜索树中最难的方法，删除某个键（如代码7所示）。第1个任务是通过搜索来找到要删除的节点。如果树中有大于1个节点，则使用_get方法来找到需要被移除的TreeNode对象。若该树仅有1个节点，这意味着要删除的是该树的根节点，但仍需要却根节点的键就是待删除的键。不管是哪种情况，若没有找到该键，del操作符便会报错。

代码7

```
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')

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

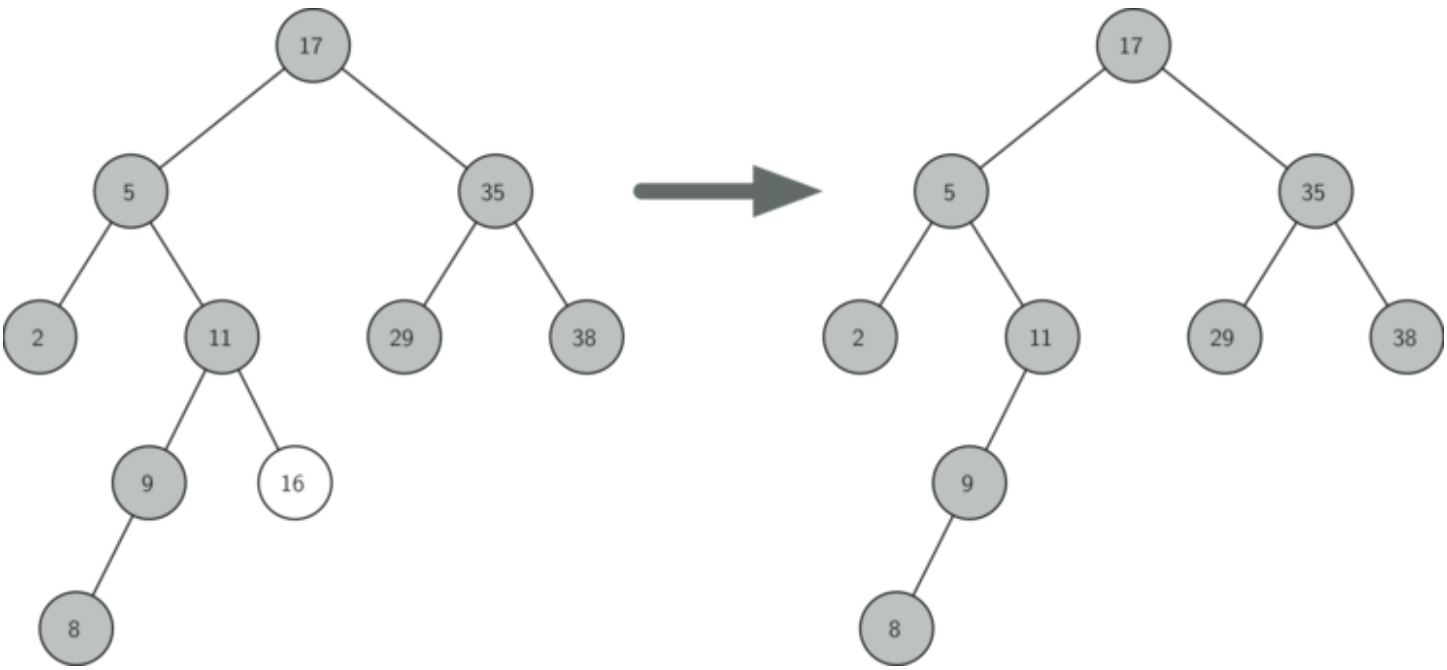
找到了想要删除的节点，便有3种情况需要考虑。

1. 待删除节点没有子节点（如图3所示）
2. 待删除节点有1个子节点（如图4所示）
3. 待删除节点有2个子节点（如图5所示）

第1种情况很好处理（如代码8所示）。若当前节点没有子节点，只需要将该节点删除，然后将其父类中指向该节点的引用删除。

代码8

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



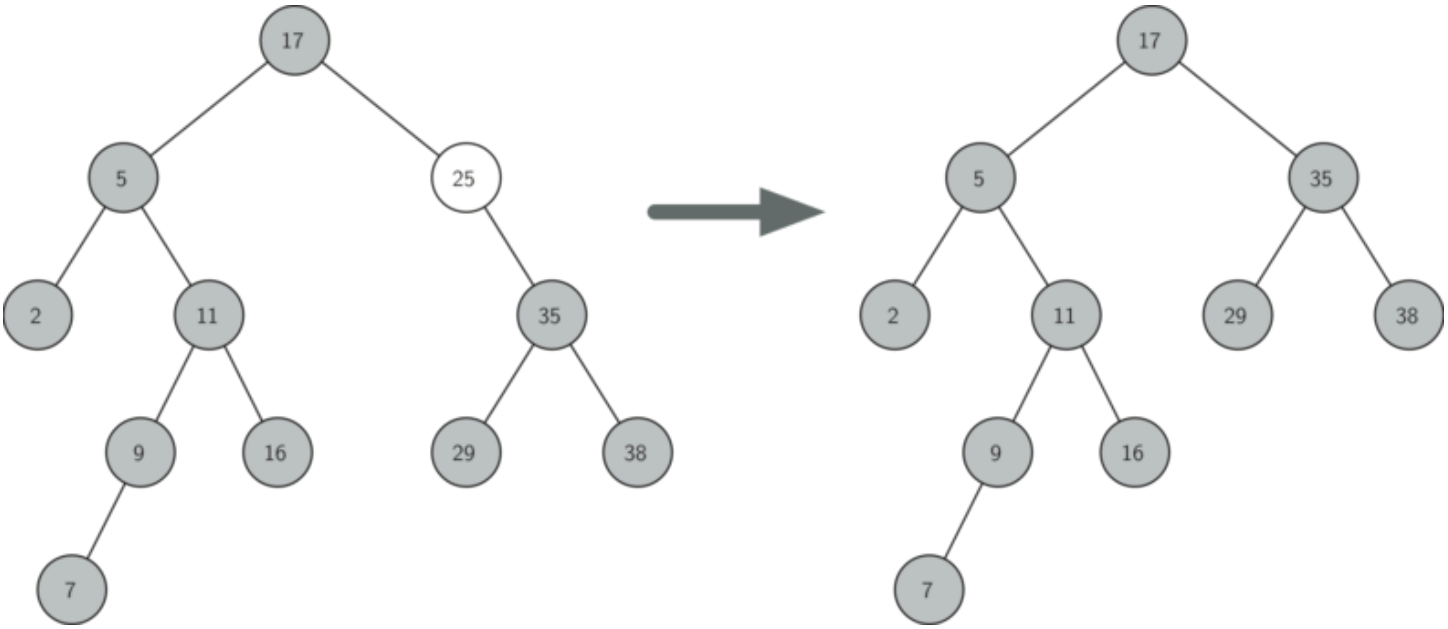
../_images/bstdel1.png

第二种情况只是稍微复杂1点（如代码9所示）。若该节点只有1个子节点，将该子节点上移，用来取待父节点（也就是将要删除的这个节点）即可。仔细研究代码，读者会发现有6种情况需要考虑。由于左、右节点是对称的，因此只用研究当前节点有1个左子节点的情况即可。处理顺序如下：

1. 若当前节点是其父节点的左子节点，只需要将当前节点的左子节点的父节点引用指向当前节点的父节点，同时将当前节点的父节点的左子节点引用指向为当前节点的左子节点。
2. 若当前节点是其父节点的右子节点，只需要将当前节点的左子节点的父节点引用指向当前节点的父节点，同时将当前节点的父节点的右子节点引用指向为当前节点的左节点即可。
3. 若当前节点没有父节点，则当前节点必是根节点，此时仅需要在根节点上使用replaceNodeData方法，将其key,payload,leftChild和rightChild替换。

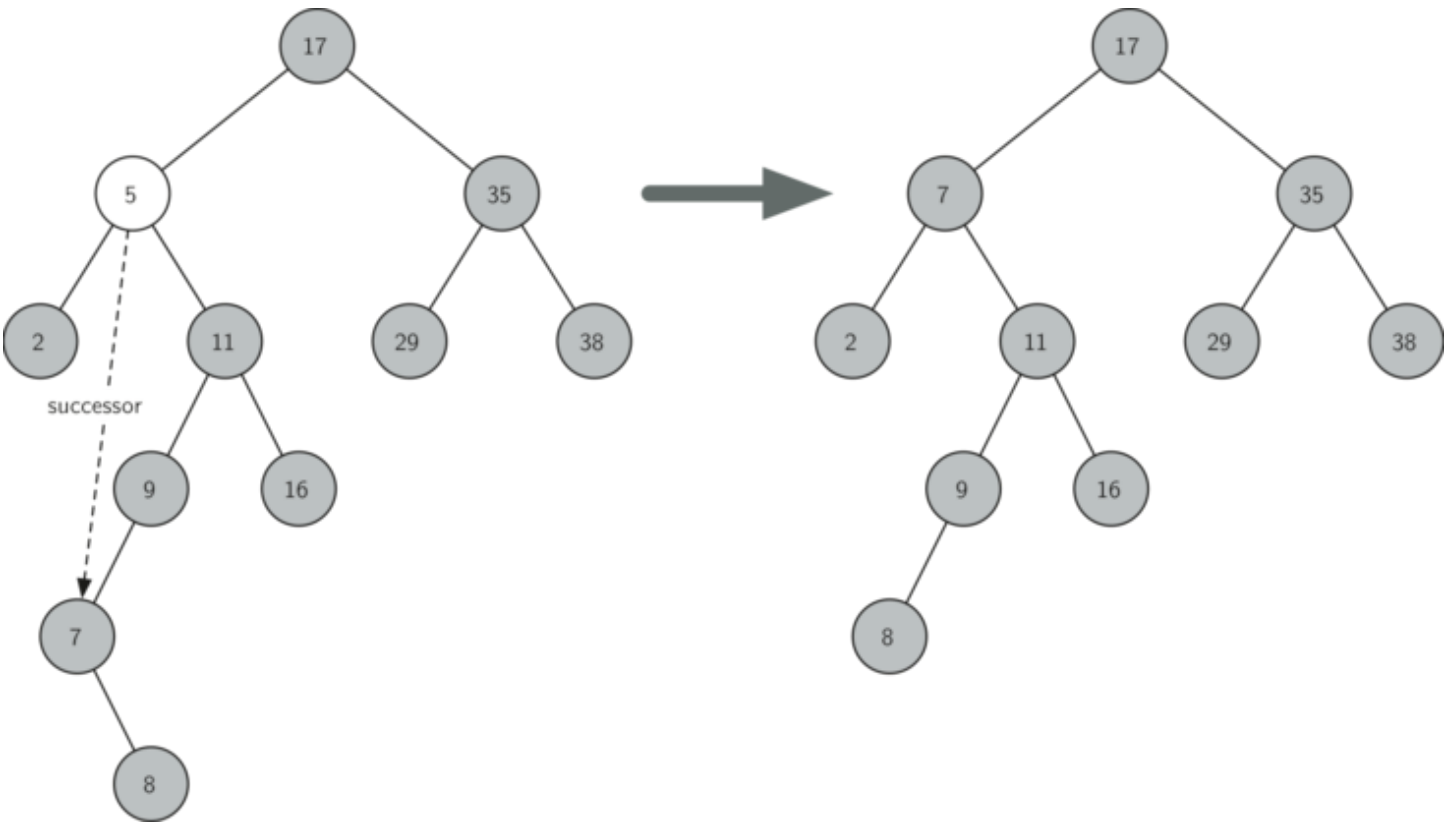
代码9

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



../_images/bstdel2.png

第3种情况是最难处理的（如代码10）。如果当前节点有2个子节点，直接将其中1个位置上移到当前节点是不可行的。然而，可以在当前节点对应的子树中进行搜索找到1个节点用来替换准备删除的节点，该节点应继续保持二分搜索树现有左右子树的关系。满足该要求的节点的键在树中的排序比当前节点的键大且仅大1个位置，该节点被称为**继任节点（successor）**，*[译者注：为了维持二分搜索数的结构性，必须保证被删除节点的位置由树中其它节点来代替；为了维持有序性，必须保证键的大小比被删除节点的左子树中任意节点都大，而比右子树的任意节点都小————那就是整棵树中比被删除的键大1个位置的节点，无论其当前位置，只要将其移动过来即可（但是移动操作要维持结构性和有序性）]*接下来马上便会给出找到继任节点的办法。继任节点至多有不超过1个子节点，因此可以用已实现的两种删除情况将其移除，然后将其放置在被删除的节点上。



../_images/bstdel3.png

处理第3种情况的代码如代码10所，注意它使用了辅助函数findSuccessor和findMin来搜索继任节点，利用方法spliceOut来移除该继任节点,spliceOut直接移动到继任节点并做正确修改，也可以直接递归调用delete，但是那样会浪费很多时间在对键的重复搜索上。

代码10

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

找到继任节点的代码如下（代码11），它是TreeNode类的1个方法。该代码利用二叉搜索树的中序遍历从大到校打印出树中的各节点。寻找继任节点时，有3种情况要考虑。

1. 若当前节点有右子节点，那么该继任节点就是当前节点的右子树中最小的键。*[译者注：最容易想到的1种情况，比当前节点键大的肯定在当前节点右子树上（如果有的话）]*
2. 若当前节点没有右子节点，并且是其父节点的左子节点，那么当前节点的父节点就是继任节点。*[译者注：如果没有右子树，则比当前节点键大的元素只能往父级上走了：若当前节点位于其父节点的左子树上，那当前节点键键必然小于父节点的键，利用二叉树有序性，可以保证父节点的键比当前节点的键大且仅大1个等级]*
3. 若当前节点没有右子节点，并且是其父节点的右子节点，那么当前节点的继任节点就是当前节点的父节点的继任节点（除去当前节点）。*[译者注：同上，如果当前节点位于其父节点的右子树上，根据有序性，当前节点树必然都比其父节点大，现在如果还要找继任节点，只能继续往上层走，这种往上走肯定就要考虑使用递归了（使用迭代过于麻烦），那就是寻找当前节点父节点的继承节点便可以实现往上走，但是根据1.当前节点父节点的继承节点实际上就是当前节点，因此必须除去当前节点，将当前节点的父节点视作没有右子树的节点]*

第1种情况是唯一在从二叉树中删除节点时有影响的。然而，findSuccessor方法还有其它用处，并且将会作为本章末的练习。

findMin方法用来查找子树中最小的键，读者应当知道二叉搜索树中的最小键位于该树的最左侧。因此，findMin方法只需要树中每个节点的leftChild引用一路向下直到某个节点没有左子节点。

代码11

```
def findSuccessor(self):
    succ = None
    if self.hasRightChild():
        succ = self.rightChild.findMin()
```

```
    else:
        if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor()
                    self.parent.rightChild = self
    return succ

def findMin(self):
    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current

def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():
                self.parent.leftChild = None
        else:
                self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():
                if self.isLeftChild():
                   self.parent.leftChild = self.leftChild
                else:
                   self.parent.rightChild = self.leftChild
                self.leftChild.parent = self.parent
        else:
                if self.isLeftChild():
                   self.parent.leftChild = self.rightChild
                else:
                   self.parent.rightChild = self.rightChild
                self.rightChild.parent = self.parent
```

再来看看二叉搜索树最后一个方法。假设想要对树中的键按顺序作1个迭代，就像字典那样，二叉树当然也可以。读者已经知道如何在中序遍历算法下按顺序对二叉树遍历了。然而，编写迭代器还需要一些额外的工作，因为迭代器在每次调用它时都只返回1个节点。

Python提供了1中非常强大的函数用来生成迭代器，即函数yield，它近似于return，也会向调用者返回1个值。但是，yield也有额外的步骤的来保存当前该函数的状态，这样当下次该函数被调用时，它可以直接从前1个状态开始。创建可迭代对象的函数被称为生成器。

二分树的中序迭代器如下面的代码所示，仔细研究它，一眼看上去读者可能觉得它并不是递归的。回忆一下，__iter__重写了迭代的for x in操作，因此它是递归的。由于它是在TreeNode实例间进行递归，因此__iter__方法被定义在TreeNode类中。

```
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
```

以下是BinarySearchTree和TreeNode方法的所有代码：

可执行代码1

```
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

    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


class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

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

    def _put(self,key,val,currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                    self._put(key,val,currentNode.leftChild)
```

```
            else:
                    currentNode.leftChild =
TreeNode(key,val,parent=currentNode)
                elif key == currentNode.key:
                        currentNode.value = val
        else:
            if currentNode.hasRightChild():
                    self._put(key,val,currentNode.rightChild)
            else:
                    currentNode.rightChild =
TreeNode(key,val,parent=currentNode)

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

    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)

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

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

```
    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')

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

    def spliceOut(self):
       if self.isLeaf():
           if self.isLeftChild():
                   self.parent.leftChild = None
           else:
                   self.parent.rightChild = None
       elif self.hasAnyChildren():
           if self.hasLeftChild():
                   if self.isLeftChild():
                      self.parent.leftChild = self.leftChild
                   else:
                      self.parent.rightChild = self.leftChild
                   self.leftChild.parent = self.parent
           else:
                   if self.isLeftChild():
                      self.parent.leftChild = self.rightChild
                   else:
                      self.parent.rightChild = self.rightChild
                   self.rightChild.parent = self.parent

    def findSuccessor(self):
      succ = None
      if self.hasRightChild():
          succ = self.rightChild.findMin()
      else:
```

```
        if self.parent:
            if self.isLeftChild():
                succ = self.parent
            else:
                self.parent.rightChild = None
                succ = self.parent.findSuccessor()
                self.parent.rightChild = self
    return succ

  def findMin(self):
    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current

  def remove(self,currentNode):
      if currentNode.isLeaf(): #leaf
        if currentNode == currentNode.parent.leftChild:
            currentNode.parent.leftChild = None
        else:
            currentNode.parent.rightChild = None
      elif currentNode.hasBothChildren(): #interior
        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload

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

```
                currentNode.parent.leftChild = currentNode.rightChild
            elif currentNode.isRightChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.rightChild
            else:

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




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

print(mytree[6])
print(mytree[2])
```

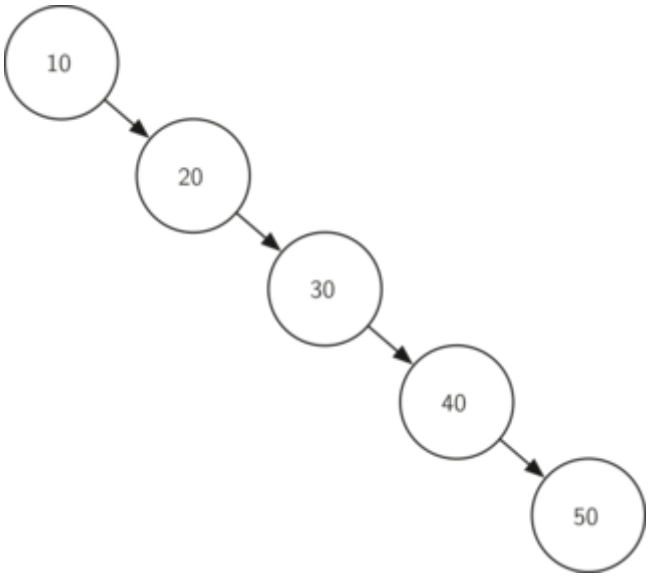
6.14 分析二分搜索树

完成了二分搜索树的实现后，现在来看看已实现的方法的分析。首先是put方法，其性能主要受限于二分搜索树的高度，因为在寻找待插入元素的正确位置时，几乎要在树的每一层都要进行对比。

二分搜索树的高度是多少？答案取决于键加入树中的方式。如果键以随机的顺序加入，树的高度大概会是logn，其中n是树的节点数。因为如果键是随机分布的，大概其中的1半都会比根节点大，另一半比根节点小。完全平衡二叉树的节点数是$2^{h+1}-1$，其中h是树的高度。

完全平衡二叉树左右两子树的节点树是一样的，put最差情况下是O(logn)的，其中n是树的节点数。注意，这与前一段的计算是个相反的过程。因此logn是树的高度，并且也是put在为新节点搜索正确位置时需要进行的最大比较次数。

不幸的是，这种二叉树也有可能是通过插入已经排好序的键来形成的，如图6所示，这样一来，put方法的性能就是O(n)了。



../_images/skewedTree.png

现在读者已经懂了为什么put方法性能受限于树的高度了，可以猜测其它方法get,in和del也是如此。由于get在树中进行搜索来找到目标键，在最差的情况下要到达树底并且没有找到对应键。乍看之下，del可能很复杂，因为它在进行删除操作前可能要查找继承节点，但是可以确定的是，在最坏情况下，为了找到继承节点也是受限于树的高度，这样一来只是将任务量加倍了*[译者注：搜索要删除的键+搜索继承节点]*，然而这只是乘了个常数，并不会改变最差情况的时间复杂度，即O(树的高度)。

6.15 平衡二叉搜索树

在前1节中讨论了二叉搜索树的建立。读者已经知道，二叉搜索树的一些操作的性能比如get和put在非平衡时会降低到O(n)。本节研究一种特殊的二叉搜索树，它使得树一直保持平衡，即AVL树，它是以其发明者G.M. Adelson-Velskii and E.M. Landis命名的。

AVL树的Map抽象数据类型实现同常规二叉搜索树一致，唯一的区别在于运行方式。为了实现AVL树，在树中的每个节点都记录了**平衡因子（balance factor）**，它是每个节点的左右子树的高度差，用公式来表达即为：

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

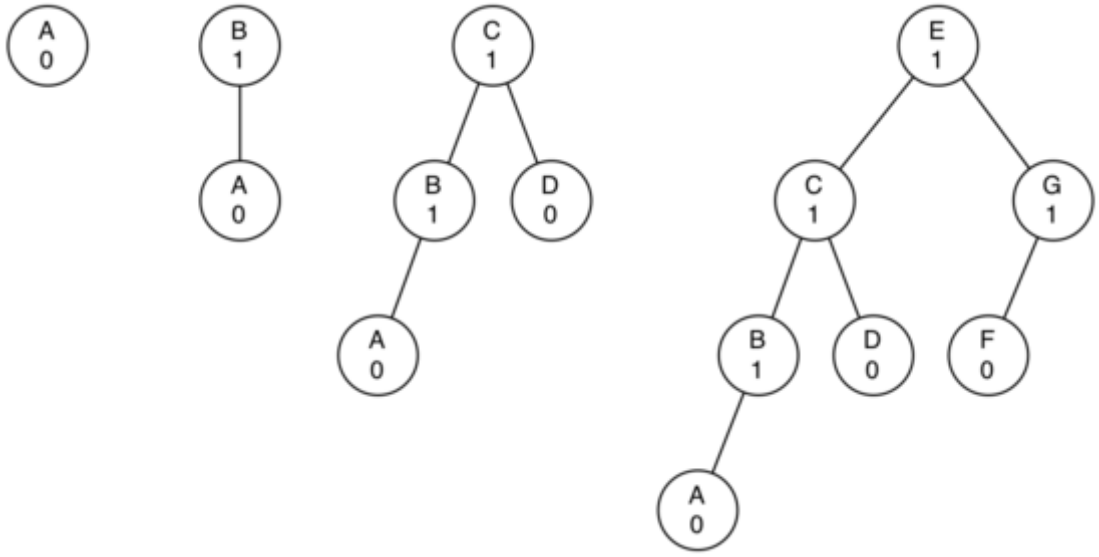
基于上述定义的平衡因子，当平衡因子大于0时，则认为子树是偏左的，小于0则偏右，等于0则是完全平衡。为了实现AVL树并且对平衡二叉树进行利用，这里将平衡因子为-1，0，1的树定义为平衡状态。若某树中某个节点的平衡因子不在此范围内，则需要一定处理来恢复平衡。

图1给出了不平衡偏右的1棵树作为例子，并标示了每个节点的平衡因子。

6.16 AVL树性能

在继续之前，先回顾一下引入平衡因子的结果，先给一个论断是，通过确保树的平衡因子为1或-1或0，可以获得更底的键操作时间复杂度。想一想，这种平衡状态下的最差情况是什么样子的？有两种可

能情况，偏左或偏右的树。以高度为0，1，2，3的树来考虑，图2给出了在新规则下可能出现的最偏左的树。



../_images/worstAVL.png

观察一下树中节点总数，可以看出高度为0的树只有1个节点，高度为1的树有2个节点，高度为2个有1+1+2个节点，高度为3的有1+2+4=7个节点。普遍的数学表达式为：

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

读者可能对这个递推关系很熟悉，它和斐波那契序列很接近。根据此式可以根据节点数推导出AVL树的高度。对于斐波那契序列i_{th}个数为：

$$F_0 = 0$$

$$F_1 = 1$$

$$F_i = F_{i-1} + F_{i-2} \text{ ，其中} i >= 2$$

一个重要的数学结论是，当斐波那契序列越来越大时，比值F_i/F_{i-1}越来越接近黄金比例$\Phi = \frac{1+\sqrt{5}}{2}$。上述公式的推导过程可以查阅数学书。这里将直接使用$F_i = \Phi^i/\sqrt{5}$来近似F_i。利用该近似值，可以将N_h写作：

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

将斐波那契项用其黄金比例近似值替代可以得到：

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

整理，两边取对数，可解得h为：

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

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

$$h = 1.44 \log N_h$$

由该式可知，AVL树的高度为树中节点数的对数与常数之积。这对于在AVL中搜索来说是个好消息，可以将其限制在O(logN)了。

6.17 AVL树的实现

本书已经演示了平衡态的AVL树可以大大提高性能，现在来研究如何改进向树中插入新元素。由于加入的新键本身可以看作1个叶节点，并且可以确定新的叶节点的平衡因子为0，因此对于刚插入的元素也没有什么新的条件了。但是一旦将新键添加进AVL树中，就必须要更新其父节点的平衡因子了。新加入的叶节点对于其父节点的影响主要取决于该叶节点是左字节点还是右子节点。若新节点是右子节点，父节点的平衡因子便要减1，反之加1，这种计算关系也用于该新节点的更高的父节点，即祖先节点，一直到树根。因为这是1个递归调用过程，可以给出2个约束条件用于更新平衡因子。

- 递归调用已经到达树的根节点。
- 父节点的平衡因子已变为0。读者应当明白，一旦1个子树的平衡因子为0，那么该子树的祖先节点的平衡因子就不用修改了。

这里将AVL树作为BinarySearchTree的子类实现。第1步，将_put方法重写，并且编写1个新的updateBalance作为辅助方法，如代码1所示。读者可以注意到，_put函数与二叉搜索树中的几乎完全一样，除了在行7和行13进行了对updateBalance的调用外。

代码1

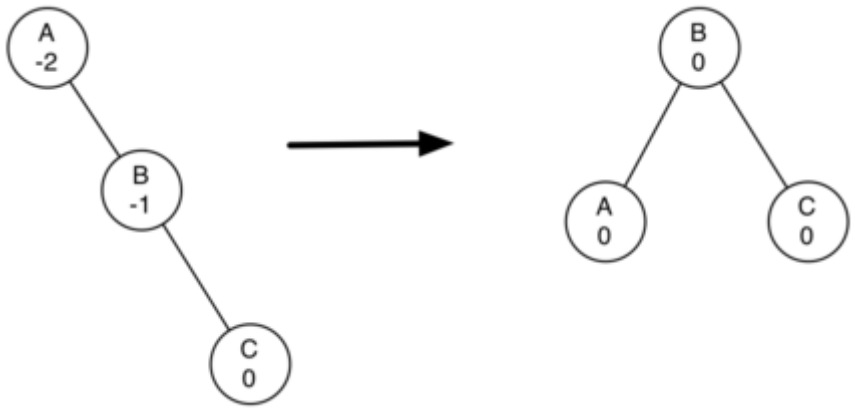
```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
                self._put(key,val,currentNode.leftChild)
```

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

def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        return
    if node.parent != None:
        if node.isLeftChild():
                node.parent.balanceFactor += 1
        elif node.isRightChild():
                node.parent.balanceFactor -= 1

        if node.parent.balanceFactor != 0:
                self.updateBalance(node.parent)
```

大部分事情都由updateBalance方法完成了，它实现了之前说的递归过程。updateBalance方法首先洁厕当前节点是否已不平衡到需要进行再平衡（行16），如果是，则立即对该节点对应的子树进行再平衡，这样就不用再对该节点的父节点或祖先节点进行平衡了。如果当前节点并不需要再平衡，那么便对该节点的父节点的平衡因子进行调整，若父节点的平衡因子为非0，那么算法通过不断递归地对父节点或者说祖先节点调用updateBalance方法向AVL树的根节点靠近。

那么再平衡应该怎么做？高效再平衡是保证AVL树的高性能的关键，为了实现再平衡，对树进行1次或者多次**旋转（rotation）**操作。

为了理解何为旋转，先来看1个简单例子，考虑下图3左侧的树，该树是不平衡的，其平衡因子为2。要将该树恢复平衡，将以节点A为根节点的子树作**左旋转**。



../_images/simpleunbalanced.png

所谓左旋转，即按以下步骤执行：

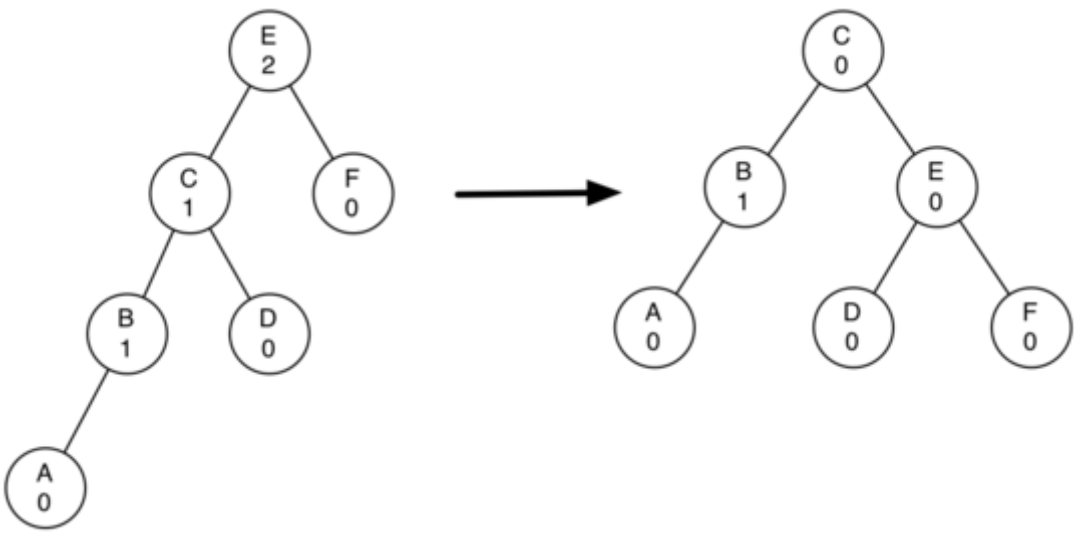
- 将右子节点上移，作为子树的根节点。
- 将原根节点（A）作为新根节点的左节点。
- 若新根节点（B）已经有1个左子节点，则将该左子节点变为新的左子节点（A）的右子节点。注意：由于新的节点B曾是A的右子节点，因此A的右子节点在此时必定为空。

[译者注：对于平衡因子为-2的节点，说明其偏右。由于|-2/2|=1，]

以上步骤在概念上来说还是很简单的，但在代码实现上还是有些难度，因为需要在保证二分搜索树的结构和有序性的前提下移动各节点。此外，还需要正确地更新各个父节点的指针。

接下来以1棵更加复杂的树来对右转进行演示。图4的左侧是1棵偏左且根节点平衡因子为2的树。右旋转即严格地按以下步骤执行：

- 将左子节点（C）上移作为子树的根节点。
- 将原根节点（E）作为新根节点的右子节点。
- 若新节点（C）已经有右子节点（D），则将其作为新右子节点（E）的左子节点。由于新根节点（C）曾是E的左子节点，因此E的左子节点在此时必为空。



../_images/rightrotate1.png

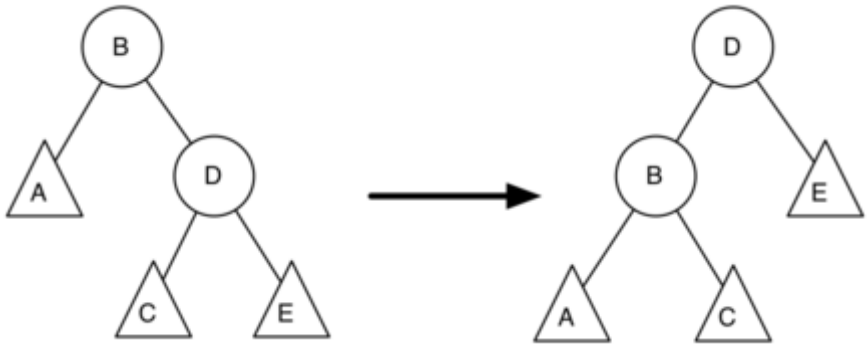
现在读者应该已经明白什么是所谓的旋转了，代码2给出了左旋和右旋的代码。在行2，生成1个空变量来跟踪子树的新根节点，如前所述，新根节点是原根节点的的右子节点，将指向右子节点的引用保存到1个临时变量中，然后把原根节点的右子节点替换为新根节点的左子节点。

下一步是调整这2个节点的父节点引用。若newRoot有1个左子节点，那么该左子节点的新父节点将变为原根节点。新根节点的父节点被设置为原根节点的父节点。若原根节点是整棵树的根节点，则必须将整棵树的根节点设置为新节点。若原根节点是1个左子节点，便将其父节点的左子节点指向为新根节点。若根节点是1个右子节点，便将其父节点的右子节点指向为新根节点（行10-13）。最后将旧根节点的父节点设置为新根节点。这个过程就像记账一样麻烦复杂，建议读者一边看函数代码一边看图3的示意图。rotateRight方法与rotateLeft是对称的，所以rotateRightde代码就留给读者自己研究了。

代码2

```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + 1 -
min(newRoot.balanceFactor, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + 1 +
max(rotRoot.balanceFactor, 0)
```

最后，行16-17需要多说两句。这两行对新、旧节点的平衡因子做了更新。因为其它所有的移动都是将整棵子树作了移动，因此子树中的其它节点的平衡因子是不受影响的。但是如何实现在不重新计算新子树高度的情况下更新平衡因子？以下推导可以说服读者。



../_images/bfderive.png

图5给出了1个左旋转。B和D是中心节点，A，C，E则是它们的子树。令h_x表示以节点x为根节点的某子树，通过定义可知：

$$newBal(B) = h_A - h_C$$

$$oldBal(B) = h_A - h_D$$

但是D的原高度也可以用 $1 + max(h_C, h_E)$ 表示，即D的高度是其2棵子树中高度的最大值+1，记住 h_C 和 h_E 的高度没有改变。因此直接将其代入到第2个式子可得：

$$oldBal(B) = h_A - (1 + max(h_C, h_E))$$

将2个作差以简化：

$$newBal(B) - oldBal(B) = h_A - h_C - (h_A - (1 + max(h_C, h_E)))$$

$$newBal(B) - oldBal(B) = h_A - h_C - h_A + (1 + max(h_C, h_E))$$

$$newBal(B) - oldBal(B) = h_A - h_A + 1 + max(h_C, h_E) - h_C$$

$$newBal(B) - oldBal(B) = 1 + max(h_C, h_E) - h_C$$

接下来将oldBal(B)移动到等式右侧，利用$max(a, b) - c = max(a - c, b - c)$可得:

$$newBal(B) = oldBal(B) + 1 + max(h_C - h_C, h_E - h_C)$$

但是，$h_E - h_C$等于$-oldBal(D)$。因此可以使用其等价命题$max(-a,-b)=-min(a,b)。据此通过以下步骤可完成newBal(B)的推导：

$$newBal(B) = oldBal(B) + 1 + max(0, -oldBal(D))$$

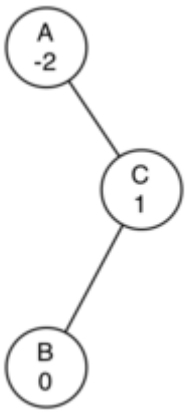
$$newBal(B) = oldBal(B) + 1 - min(0, oldBal(D))$$

现在方程中的各项都是已知的了，考虑到B是**rotRoot**且D是**newRoot**，则可以看出行16对应下式：

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

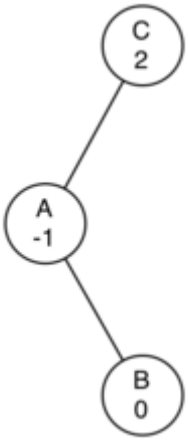
类似的推导可以给出更新节点B的方程，以及在右旋后的平衡因子，这就作为练习了。

现在读者可能认为已经差不多ok了。现在已经知道如何进行左、右旋转了，并且也知道合适应该进行左/右旋转。不过，考虑下在左旋转时发生了什么。



../_images/hardunbalanced.png

如图7所示，在完成左旋后，进入了另一种不平衡的状态。如果使用右旋来修正，结果会回到起点。

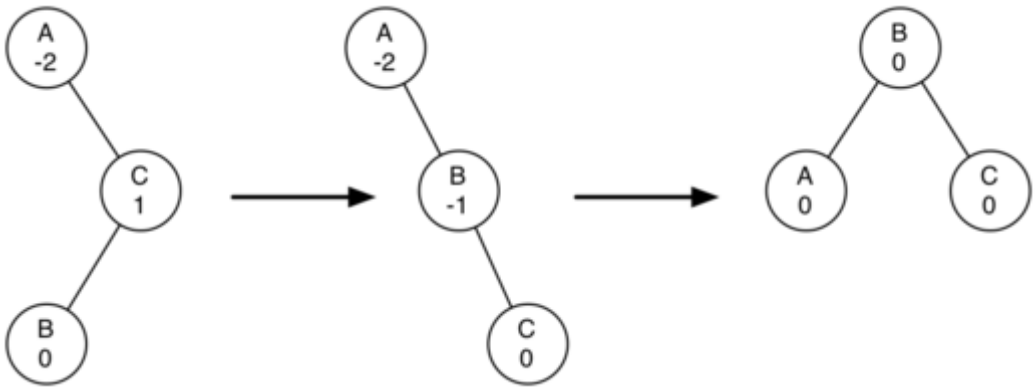


../_images/badrotate.png

为了解决该问题，必须按以下规则进行：

- 如果1个子树需要进行左旋来将其恢复平衡，首先检查其右子节点的平衡因子。如果右子节点偏左，则对右子节点作右旋，然后再对原子树作左旋。
- 如果1个子树需要进行右旋来将其恢复平衡，首先检查其左子节点的平衡因子。如果左子节点偏右，则对左子节点作左旋，然后再对原子树作右旋。

如8所示，图6和图7中的困境可以利用以上规则解决。以C为中心右旋转后，再以A为中心左旋转，便可以将整棵子树恢复平衡。



../_images/rotatelr.png

rebalance方法中实现了以上规则，如代码3所示。

```
def rebalance(self,node):
  if node.balanceFactor < 0:
         if node.rightChild.balanceFactor > 0:
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
         else:
            self.rotateLeft(node)
  elif node.balanceFactor > 0:
         if node.leftChild.balanceFactor < 0:
            self.rotateLeft(node.leftChild)
            self.rotateRight(node)
         else:
            self.rotateRight(node)
```

通过始终维持树的AVL态，可以保证get方法以$O(\log(n))$的复杂度运行。那么put方法的复杂度为多少？将其分为put方法的每1步操作来看。由于新节点本身是作为叶节点插入的，对其所有祖先节点的

平衡因子进行调整需要logn次操作（树的每一层对应1次）。

此时已经实现了可供使用的AVL树，当然需要有删除节点功能，这就作为练习了。

6.18 总结：抽象数据类型Map的实现

在以上两张中，已经研究了几种可以用于实现抽象数据类型Map的数据结构。列表的二分搜索，哈希表，二分搜索树以及平衡二分搜索树。这里将总结各实现的性能。

operation	有序列表	哈希表	二分搜索树	AVL树
put	O(n)O(n)	O(1)O(1)	O(n)O(n)	O(log2n)O(log2n)
get	O(log2n)O(log2n)	O(1)O(1)	O(n)O(n)	O(log2n)O(log2n)
in	O(log2n)O(log2n)	O(1)O(1)	O(n)O(n)	O(log2n)O(log2n)
del	O(n)O(n)	O(1)O(1)	O(n)O(n)	O(log2n)

6.19 总结

本章研究了树形数据结构，它可以用来实现很多有趣的算法。

- 用二叉树解析及计算表达式。
- 用二叉树实现Map ADT。
- 用平衡二分搜索树（AVL）实现Map ADT。
- 用二分树实现堆。
- 用堆实现优先队列。