致谢

2015 地空数算课程教材第 6 章翻译小组

组长

张虎来

组员

于润泽、余晓辉、宋欣源、汪建峰、高鸿宇、周安、黄佳旺、卢思奇、刘明辰、张沙洲（排名不分先后）

本文为教材第 6 章

树和树算法

目标

- 了解树的数据结构及使用；
- 树用来实现映射 map 数据结构；
- 采用 List 来实现树；
- 采用类和引用来实现树；
- 树实现为递归数据结构；
- 采用堆 heap 实现优先队列。

树的例子

我们已经学过了像栈和队列这样的线性数据结构，同时我们对递归也有了一定的了解，现在让我们来看看另一种常见的数据结构——树结构（Tree）。树结构在计算机科学里面应用广泛，包括操作系统，图形学，数据库系统和计算机网络。树结构和真正的树有许多相像的地方，也包括根、枝和叶，它们的不同在于计算机中树结构的根在顶层而它的叶子则在下面的底层。

在我们开始学习树结构之前，让我们先来看看几个常见的树结构的例子。首先让我们看看生物学中的分类。图一是一个动物分类的例子，从中我们可以看出树结构的几个特点。第一，

这个例子说明树结构是分级的，这里分级的意思是树结构的顶层部分更加宽泛一般而，而底层部分更加精细具体。在这个例子中，最上级是“界”，它下面的一层（上层的子级）是“门”，然后以此类推是“纲”等等。但是，无论我们细分到多少层，这里面包含的生命体也都是动物。

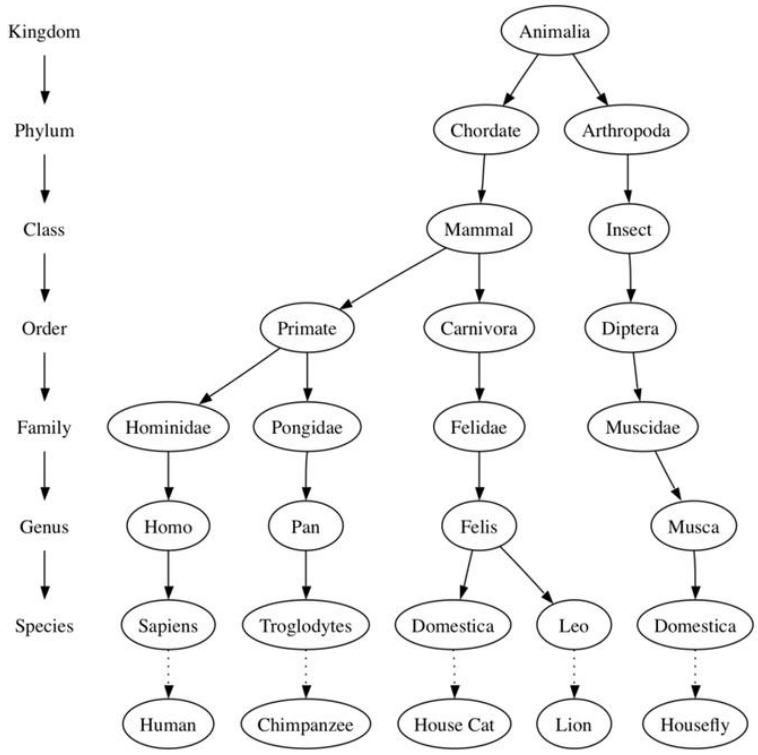


图 1：一些动物的分类树

我们注意到可以从树的顶层开始然后沿着圆圈和箭头构成的一条路径到达树的底层。在树的每一层我们都可以问自己一个问题，然后沿着相符的那条路径继续下去。比如我们可以问“这个动物是脊椎动物还是无脊椎动物”，如果回答是“脊椎动物”我们就沿着脊椎动物这条路下去然后接着问“这个脊椎动物是哺乳动物吗”，如果回答“不是哺乳动物”我们就卡在这里了（不过仅限于这个简单的例子会有这种情况）。当我们到达哺乳动物这一层的时候我们问自己“这个哺乳动物是灵长类还是食肉动物”。我们可以沿着路径一直走下去直到树的最底层，这也就是我们平常的命名了。

树结构的第二个特点是一个节点(node)的所有子节点（children）和另一个节点的子节点是完全独立的。比如“猫属”有两个子节点“家生”和“野生”，“蝇属”中也有一个“家生”，但它和“猫属”中的“家生”完全不同而且相互独立。这意味着我们可以在不影响“猫属”的子节点的情况下更改“蝇属”的子节点。

树结构的第三个特点就是每个它的叶节点（leaf）都是不同的。对每一种动物，我们都可以从根节点（root）开始沿着一条特定的路径找到它对应的叶节点，并把它和其他动物区分开，例如对于家猫，我们可以沿着动物界——脊索动物门——哺乳动物纲——食肉动物目——猫科——猫属——家猫找到它。

另一个树结构的例子就是你每天都会用到的文件系统。在文件系统中，磁盘的分支或者说子目录都是运用了树结构来构建的。图 2 展示了 Unix 文件系统的一小部分的分层情况。

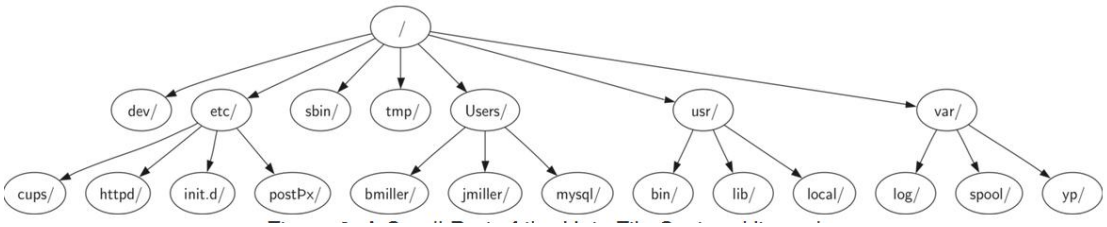


图 2 ： Unix 文件系统的一小部分的分层情况

这个树结构的文件系统和真正的树也非常相像。你可以从根节点出发沿着一条路径到任意分支。这条路径会把这个子分支（包括它里面的所有文件）和其他分支区别开。树结构的另一重要特点，就是你可以将树下层的所有部分（叫做子树 subtree）移动到树的另一位置而不影响更下层的情况，这是由树的分级方式决定的。例如，我们可以将所有标注/etc 的子树从根节点下移动到 usr/下面。这样做会将 httpd 的路径从/etc/httpd 改变成/usr/etc/httpd，但是对 httpd 的内容和子节点的内容不会有影响。

最后一个树结构的例子是一个网页。下图是一个利用超文本标记语言（HTML）编写的简单网页。图 3 是构成网页的超文本标记语言中的标签相互关联关系所构成的树。

```
<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>
```

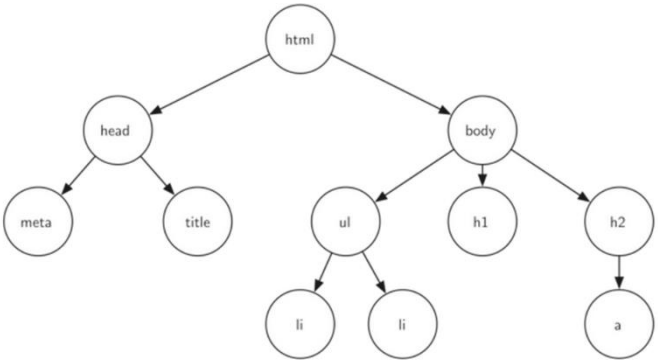


图 3：网页的标记符之间的相互关联所构成的树

上面的超文本标记的代码和它对应的树说明了另一种分级方式。我们发现树的每一层都对应超文本标记符的一层嵌套。代码的第一个标记符是<html>同时最后一个是</html>。这一页中所有其他的标记符也都是成对的。试一下你就会发现这种嵌套的特点在树的每一层都是成立的。

术语表与定义

现在我们已经看了几个树的例子了，我们将要正式定义树结构以及构成它的要素。

节点（Node）
节点（Node）是树的基本构成部分。它可能有其他专属的名称，我们称之为“键（key）”。一个节点也可能有更多的信息，我们称之为“负载”。虽然负载信息和树的许多算法并不直接相关，但是它对于树的应用至关重要。

边（Edge）
边（Edge）也是树的基本构成部分。边链接两个节点，并表示它们之间存在联系。除了根节点外每个节点都有且只有一条与其他节点相连的入边（指向该节点的边），每个节点可能有许多条出边（从该节点指向其他节点的边）。

根节点（Root）
根节点是树种中唯一一个没有入边的节点。在图 2 中，“/”是树的根节点。

路径（Path）
路径是由边链接起来的节点的有序排列。例如：（动物界——脊索动物门——哺乳动物纲——食肉动物目——猫科——猫属——家猫 ）就是一条路径。

子节点集（Children）
当一个节点的入边来自另一个节点时，我们称前者是后者的子节点，同一个节点的所有子节点构成子节点集。在图 2 中， 节点 log/, spool/, yp/构成节点 var/.的子节点集。

父节点（Parent）
一个节点是它出边所连接的所有节点的父节点。在图 2 中， 节点 var/是节点 log/, spool/, yp/的父节点。

兄弟节点（Sibling）
同一个节点的所有子节点互为兄弟节点，在文件系统树中节点 etc/和节点 usr/是兄弟节点。

子树（Subtree）
子树是一个父节点的某个子节点的所有边和后代节点所构成的集合。

叶节点（Leaf Node）
没有子节点的节点成为称为叶节点。例如图一中的“人”和“黑猩猩”就是叶节点。

层数（Level）
一个节点的层数是指从根节点到该节点的路径中的边的数目。例如，图 1 中“猫属”的层数是 5.定义根节点的层数为 0.

高度（Height）
树的高度等于所有节点的层数的最大值。图 2 中树的高度为 2.

我们已经定义好所需的术语了，现在可以正式定义树了。我们将用两种方式定义，一种需要用到节点和边，而另一种更为有效的定义方式是利用递归定义。

定义一：树是节点和连接节点的边的集合，它有以下特征：

•有一个节点被设计为根节点。
•除了根节点的每一个节点 n，都通过一条边与它唯一的父节点相连。
•可以沿着唯一的路径从根节点到每个节点。
•如果这个树的每个节点都至多有两个子节点，我们称它为二叉树。

图 3 展示了一个符合定义一的树。每条边的箭头指出了连接的方向。

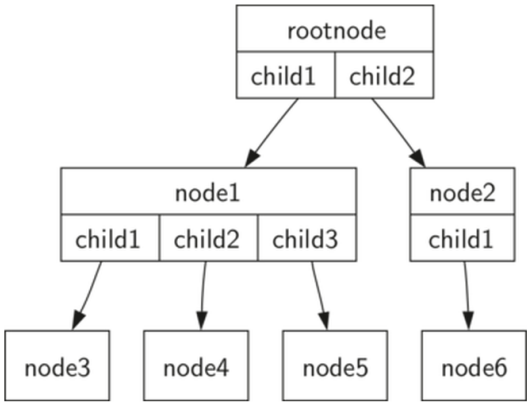


图 3：由节点和边构成的树

定义二：每个树或者为空或者包含一个根节点和零个或多个子树，其中每个子树也符合这样的定义。每个子树的根节点和其父树的根节点之间通过边相连。图 4 描绘了这种递归定义的树。通过这种树的递归定义，我们知道图 4 中的树至少有 4 个节点，因为每个三角形所代表的子树必须有根。它也可能有更多的节点，但我们需要更深入的了解这棵树来得到答案。

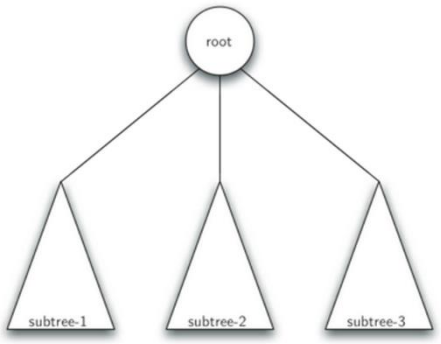


图 4：递归法定义的树

实现

“嵌套列表”表示树

在用嵌套列表表示树时，我们将用 Python 的列表数据结构来编写上面定义的功能。虽然把界面写成列表的一系列方法与我们已实现其他的抽象数据类型有些不同，但这样做的是有趣的，因为它为我们提供一个简单、可以直接检查的递归数据结构。在列表实现树时，我们将存储根节点作为列表的第一个元素的值。列表的第二个元素的本身是一个表示左子树的列表。这个列表的第三个元素将是表示在右子树的另一个列表。为了说明这个存储技术，让我们来看一个例子。图 1 示出一个简单的树以及相应的列表实现。

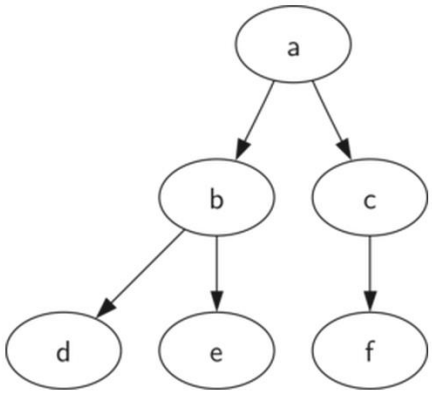


图 1: 简单树

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

请注意，我们可以使用标准索引访问列表的子树。树的根是 myTree[0]，根的左子树是 myTree[1]，和右子树是 myTree[2]。 ActiveCode1 说明了如何用列表创建简单树。一旦树被构建，我们可以访问根和左、右子树。嵌套列表法的一个非常好的特性是子树的结构与树相同;本身是递归的！子树具有根值和两个表示叶节点空列表。列表方法列表的另一个优点是它容易扩展到多叉树。在树不仅仅是一个二叉树的情况下，另一个子树只是另一个列表。

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

使用索引来访问子树（tree_list1）

让我们提供一些功能，使我们很容易像树的数据结构定义一样应用列表 list。请注意，我们不会去定义一个二叉树类。我们将编写的功能将只是操作标准列表使之类似于树类型。

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

该二叉树功能只是构建一个根节点和两个空子节点列表。给左子树添加到树的根，我们需要插入一个新的列表到根列表的第二位置。我们必须注意，如果列表中已经有一些在第二的位置，我们需要跟踪它，将新节点插入树中作为其直接的左子节点。表 1 显示了插入左子节点 Python 代码。

表 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
```

请注意，插入一个左子节点，我们首先获取对应于当前左子节点的（可能是空的）列表。然后，我们添加新的左子，将原来的左子结点作为新节点的左子节点。这使我们能够将新节点插入到树中的任何位置。对于 insertRight 的代码类似于 insertLeft，如表 2 中。

表 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
```

为了完善实现树的功能（参见表 3），让我们写几个用于获取和设置的根值的功能，以及获得的左边或右边子树的功能。

表 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]
```

ActiveCode2 练习我们刚才实现树的功能。你应该自己尝试一下，其中的练习要求你吸取这一章的知识来实现树的结构。

```
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)))
```

用 Python 来实现基本的树的功能（bin_tree）

自我检查

Q-32：根据下列描述

```
x = BinaryTree('a')
insertLeft(x,'b')
insertRight(x,'c')
insertRight(getRightChild(x),'d')
insertLeft(getRightChild(getRightChild(x)),'e')
```

下面哪一个是树的正确表示？

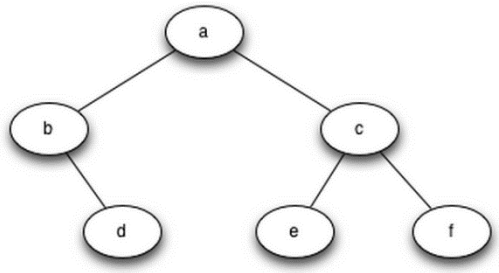
a) ['a', ['b', [], []], ['c', [], ['d', [], []]]]

b) ['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]

c) ['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]

d) ['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]

编写一个函数 buildTree,用嵌套列表方法生成下图的树：



节点和引用

我们第二种表示数的方式 用节点和引用。在这种情况下，我们将定义具有根值，以及左和右子树属性的类。由于这种表示更紧密地跟随面向对象的编程方式，我们将继续使用这种表示完成本章的其余部分。

使用节点和引用，我们可能会认为该树的结构类似于图 2 中所示。

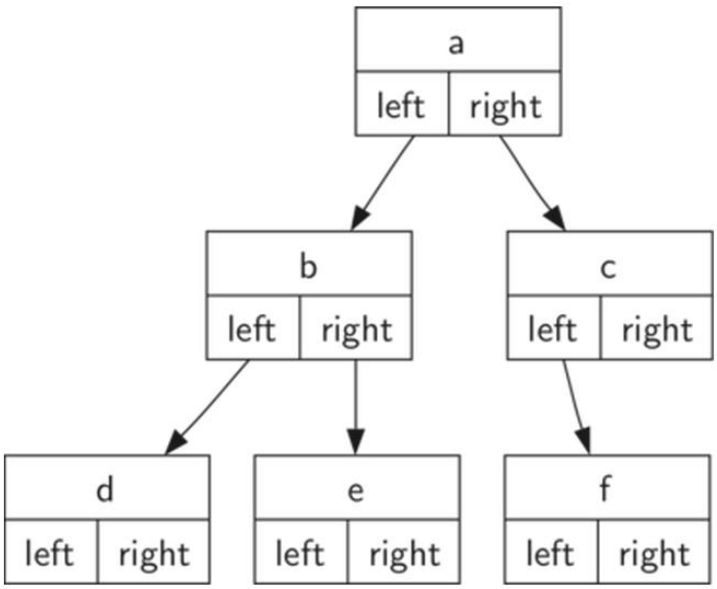


图 2：简单树使用节点和引用

我们将开始用简单的节点和引用的类定义如表 4 所示。重要的是要记住这种表示的是左和右属将引用其他二叉树实例的。例如，当我们插入一个新的左子节点到树上时，我们创建了二叉树的另一个实例，修改了根节点的”self leftChild”使之指向新的树。

表 4

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

注意表 4 中，构造函数需要得到一些类型的对象存储在根中。就像你可以在列表中储存

你喜欢的任何一种类型，树的根对象可以指向任何一种类型。对于我们的早期例子中，我们将存储节点为根值的名称。使用节点和引用来表示树在图 2 中，我们将创建二叉树类的六个实例。

接下来让我们看一下我们需要构建的根节点以外的功能。为了添加左子节点，我们将创建一个新的二叉树，并设置根的左属性以指向这个新对象。对于 insertLeft 的代码表 5 所示。

表 5

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

我们必须考虑两种情况进行插入。第一种情况下的特征是，没有现有左子节点。当没有左子节点时，将新节点添加即可。第二种情况的特征是，当前存在左子节点。在第二种情况下，我们插入一个节点并将之前的子节点降级。第二种情况是由 else 语句表上 54 行处理。

对于 insertRight 的代码必须考虑一个对称组的情况。有要么没有正确的子节点，或者我们必须插入根和现有的右子之间的节点。插入代码表 6 所示。

表 6

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

为了完成一个简单的二叉树数据结构的定义，我们会写访问（参见表 7）左右子节点和根值的方法。

表 7

```
def getRightChild(self):
    return self.rightChild

def getLeftChild(self):
    return self.leftChild

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

def getRootVal(self):
```

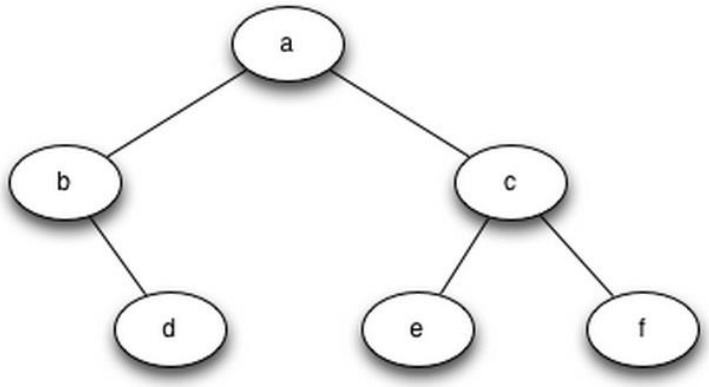
```
    return self.key
```

既然我们已经有了所有创建和操作二叉树的方法，让我们再进一步检查它的结构。让我们把每一个节点比作一个简单的树的根，并添加节点 B 和 C 作为子节点。 ActiveCode1 创建树，并存储一些关键值，左，右。注意，两个左和右的根的子节点本身是二叉树类的不同实例。正如我们树的原始定义中说的，这使我们能够把一个二叉树的任何子节点当成二叉树本身。

运行 Codelens 保存

```
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):
```

写一个 buildTree 函数，通过调用 BinaryTree 类方法，生成如图所示的二叉树：



解析树

完成树（Tree）数据结构的实现之后，现在我们来看一个一个例子，它将告诉你怎么样利用树（Tree）去解决一些实际的问题。在这个章节，我们将关注一些语法树。语法树常常用于表示真实世界中的结构，例如：句子或者数学表达式。

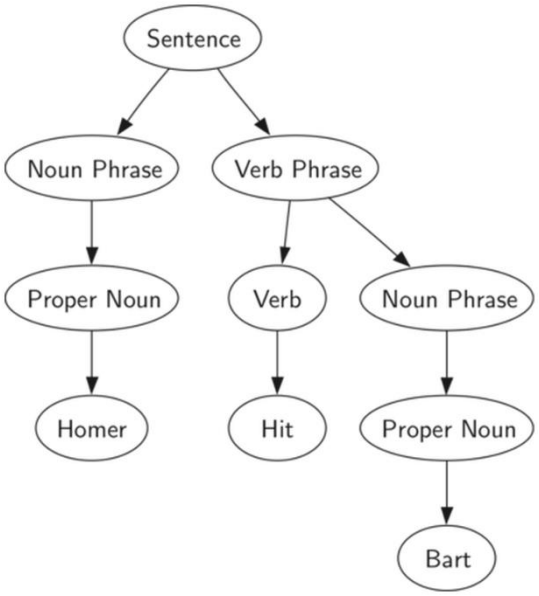


图 1：一个简单句子的语法树

图 1 显示了一个简单句子的层次结构。将一个句子表征为一个树（Tree）的结构，能使我们通过利用子树来处理句子中的每个独立成分。

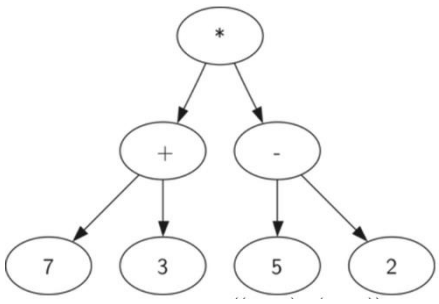


图 2：（（7+3）*（5-2））的语法树

如图 2 所示，我们能将一个类似于（（7+3）*（5-2））的数学表达式表达为一个语法树。我们已经看见了那个全括号表达式，那么我们怎样理解这个表达式呢？我们知道乘法比加或者减有着更高的优先级。因为表达式中的括号，我们知道在做乘法运算之前，需要先计算括号内的加法或者减法。树的层次结构帮助我们理解整个表达式的运算顺序。在计算最顶上的乘法运算前，我们先要计算子树中的加法和减法运算。左子树的加法运算结果为 10，右子树的减法运算结果为 3。利用树的层次结构，一旦我们计算出了子节点中表达式的结果，我们能够将整个子树用一个节点来替换。运用这个替换步骤，我们得到一个简单的树，如图 3 所示。

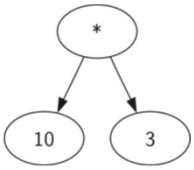


图 3：((7+3)*(5−2))的化简后的语法树

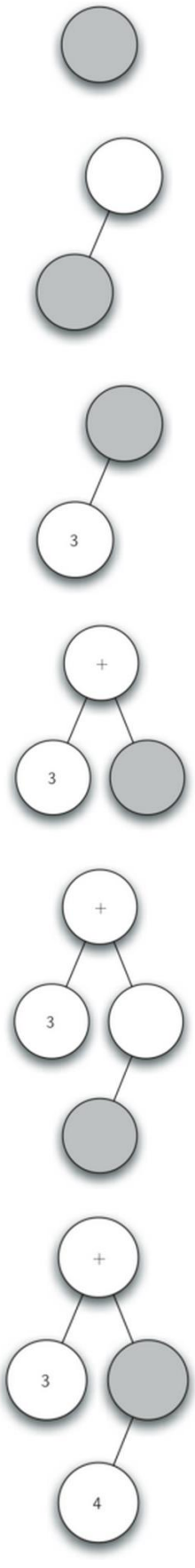
在本章的剩余部分，我们将更加详细地验证语法树。尤其是：

- 怎样根据一个全括号数学表达式来建立其对应的语法树
- 怎样计算存在语法树中的数学表达式的值
- 怎样根据一个语法树恢复出原先的数学表达式

建立语法树的第一步，将表达式字符串（string）分解成符号列表。这里有四种符号需要我们考虑：左括号，操作符和操作数。我们知道当读到一个左括号时，我们将开始一个新的表达式，因此我们创建一个对应于这个表达式的新的树。相反，无论什么时候我们读到一个右括号，我们就得结束这个表达式。另外，操作数将成为叶节点（leaf node）和他们所属的操作符的子节点（children）。最后，我们知道每个操作符都应该有一个左子节点和一个右子节点。通过上面的分析我们定义以下四条规则：

1. 如果当前读入的字符是’(’，添加一个新的节点（node）作为当前节点的左子节点，并下降到左子节点处。
2. 如果当前读入的字符在列表[‘+’, ’-‘, ’/’, ’*’]中，将当前节点的根值（root value）设置为当前读入的字符。添加一个新的节点（node）作为当前节点的右子节点，并下降到右子节点处。
3. 如果当前读入的字符是一个数字，将当前节点的根植设置为该数字，并返回它的父节点（parent）。
4. 如果当前读入的字符是’)’，返回当前节点的父节点（parent）。

在我们编写 Python 代码之前，让我们一起看一个上述概述的例子。我们将使用（3+（4*5））这个表达式。我们将表达式分列为如下字符的列表：['(', '3', '+', '(', '4', '*', '5' ,')',')']。一开始，我们从一个仅包括一个空的根节点的语法树开始。如图 4，随着每个新的字符被读入，该图说明了该语法树的内容和结构。


3
+
3
+
3
+
3
4

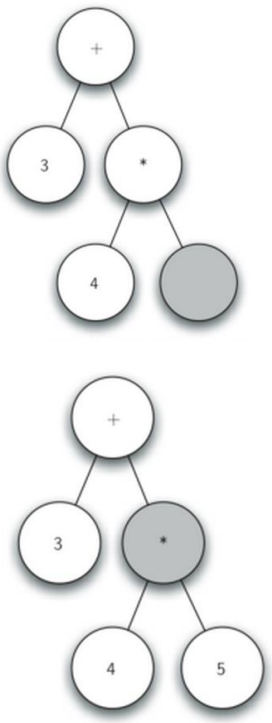


图 4：语法树结构的描摹图
看着图 4，让我们一步一步地过一遍：

a. 创建一个空的树。
b. 读如’(‘作为第一个字符，根据规则 1，创建一个新的节点作为当前节点的左子结点，并将当前节点变为这个新的子节点。
c. 读入‘3’作为下一个字符。根据规则 3，将当前节点的根值赋值为 3 然后返回当前节点的父节点。
d. 读入‘+’作为下一个字符。根据规则 2，将当前节点的根值赋值为+，然后添加一个新的节点作为其右子节点，并且将当前节点变为这个新的子节点。
e. 读入’(‘作为下一个字符。根据规则 1，创建一个新的节点作为当前节点的左子结点，并将当前节点变为这个新的子节点。
f. 读入‘4’作为下一个字符。根据规则 3，将当前节点的根值赋值为 4 然后返回当前节点的父节点
g. 读入‘*’作为下一个字符。根据规则 2，将当前节点的根值赋值为*，然后添加一个新的节点作为其右子节点，并且将当前节点变为这个新的子节点。
h. 读入‘5’作为下一个字符。根据规则 3，将当前节点的根值赋值为 5 然后返回当前节点的父节点
i. 读入’)’作为下一个字符。根据规则 4，我们将当前节点变为当前节点’*’的父节点。
j. 读入’)’作为下一个字符。根据规则 4，我们将当前节点变为当前节点’+’的父节点，然而当前节点没有父节点，所以我们已经完成语法树的构建。

通过上面给出的例子，很明显我们需要保持追踪当前节点和当前节点的父节点。树的结构提供给我们一个获得一个节点的子节点的方法——通过 getLeftChild 和 getRightChild 方法，但是我们怎么样来追踪一个节点的父节点呢？一个简单的方法就是利用堆栈（stack），在我们

遍历整个树的过程中保持跟踪父节点。当我们想要下降到当前节点的子节点时，我们先将当前节点压入栈中。当我们想要返回当前节点的父节点时，我们从堆栈中弹出该父节点。

通过上述的规则，使用堆栈（Stack）和二叉树（BinaryTree）操作，我们现在准备编写一个创建语法树的 Python 函数。语法树生成函数的代码如 ActiveCode1 中展示的。

```
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
```

创建一个语法树（语法树创建）
这四条为了建立语法树的规则被编写为四个 if 从句，它们分别在第 11,15,19,24 行。如上面所说的，在这几处你都能看到规则的代码实现，并需要调用一些 BinaryTree 和 Stack 的方法。这个函数中唯一的差错检查是在 else 语句中，一旦我们从列表中读入的字符不能辨认，我们就会报一个 ValueError 的错误。

现在我们已经建立了一个语法树，我们能用它来干什么呢？作为第一个例子，我们将写一个函数来计算语法树的值，并返回该计算的数字结果。为了实现这个函数，我们将利用树的分层结构。重新看一下图 2，回想一下我们能够将原始的树替换为化简后的树（图 3）。这提示我们写一个能够通过递归地计算每个子树的值来计算整个语法树的值的算法。

就像我们以前实现递归算法那样，我们将从识别基本问题开始来设计递归计算表达式值的函数。这个递归算法的一个自然的基本问题是检查一个操作符是否为叶节点。在语法树中，叶节点总是操作数。因为数字的变量如整数和浮点数不需要更复杂的分析，这个求值函数只需要简单地返回叶节点中储存的数字就可以。使函数朝向基本情形前进的递归过程就是调用求值函数计算当前节点的左子树、右子树的值。递归调用有效的使我们朝着叶节点，沿着树下降。

为了将两个递归调用的值整合在一起，我们只需简单地使用将存在父节点中的操作符应用到两个子节点返回的结果。在图 3 中，我们能看到两个子节点的值，分别为 10 和 3。对他们使用乘法运算得到最终结果 30。

递归求值函数的代码如 Listing1 所示，我们得到当前节点的左子节点、右子节点的参考。如果左右子节点的值都是 None，我们就能知道这个当前节点是一个叶节点。这个检查在第 7 行。如果当前节点不是一个叶节点，查看当前节点中的操作符，并将它运用到通过递归求值得到的左右子节点的结果计算上来。

为了实现这个算法，我们使用了字典（dictionary），键值分别为'+'，'-'，'*'和'/'。存在字典里的值是 Python 的操作数模块（operator module）中的函数。这个操作数模块为我们提供了很多常用函数的操作符。当我们在字典中查找一个操作符时，相应的操作数变量被取回。因为这个取回的变量是一个函数，我们按通常函数调用的方式来调用它们，如函数名（变量 1，变量 2）。所以查找操作符'+'（2,2）就等价与 operator.add(2,2)。

表 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，我们将在图 4 中创建的语法树上遍历求值。当我们第一次调用求值函数时，我们传递语法树参数 parseTree，作为整个树的根。然后我们获得左右子树的引用来确保他们一定存在。递归调用发生在第 9 行。我们从查看树根中的操作符开始，这是一个'+'。这个'+'操作符查找到 operator.add 函数调用，且有两个变量。通常对一个 Python 函数调用而言，Python 第

一件做的事情就是评估传递给函数的参数。通过从左到右的求值过程，第一个递归调用从左边开始。在第一个递归调用中，求值函数用来计算左子树。我们发现这个节点没有左、右子树，所以我们在一个叶节点上。当我们在叶节点上时，我们仅仅是返回这个叶节点存储的数字值作为求值函数的结果。因此我们返回整数 3.

现在，为了顶层调用 operator.add 函数，我们已经计算好其中一个参数了。但是我们还没有完成。继续从左到右计算参数，我现在递归调用求值函数用来计算根节点的右子节点。我们发现这个节点既有左节点又有右节点，所以我们查找这个节点中存储的操作数，是'*'，然后调用这个操作数函数并将它的左右子节点作为函数的两个参数。此时，你将明白所有的递归调用都将去往叶节点，各自的值分别为整数 4 和 5。求出这两个参数值后，我们返回 operator.mul(4,5)的值。此时，我们已经计算好了顶层操作符'+'的两个操作数了，所有需要做的只是完成调用函数 operator.add(3,20)即可。这个结果就是整个表达式树(3+(4*5))的值，这个值是 23.

树的遍历

之前我们已经了解了树结构的基本功能了，现在我们来看一看一些使用方法。使用方法按照访问节点的方式不同分为 3 种。这三种方式经常被用于访问树中的所有节点，它们之间的不同在于访问每个节点的次序不同。我们称这种对所有节点的访问为遍历（traversal）。这三种遍历分别叫做前序遍历（preorder），中序遍历（inorder）和后序遍历（postorder）。我们先来给出它们的详细定义，然后举例看看它们的应用。

前序遍历（preorder）
在前序遍历中，我们先访问根节点，然后递归地用前序遍历访问左子树，再递归地前序遍历访问右子树。

中序遍历（inorder）
在中序遍历中，我们递归地中序遍历访问左子树，然后访问根节点，最后再递归地中序遍历访问右子树。

后序遍历（postorder）
在后序遍历中，我们先递归地后序遍历访问左子树和右子树，最后访问根节点。

现在我们用几个例子来说明这三种不同的遍历。首先我们先看看前序遍历。我们树结构来表示这本书来看看前序遍历的方式。书是树的根节点，每一章是根节点的子节点，每一节是章的子节点，每一小节是每一节的子节点，以此类推。图 5 是一本书只取了两章的一部分。虽然遍历的算法适用于含有任意多子树的树结构，但我们目前为止只用了二叉树。

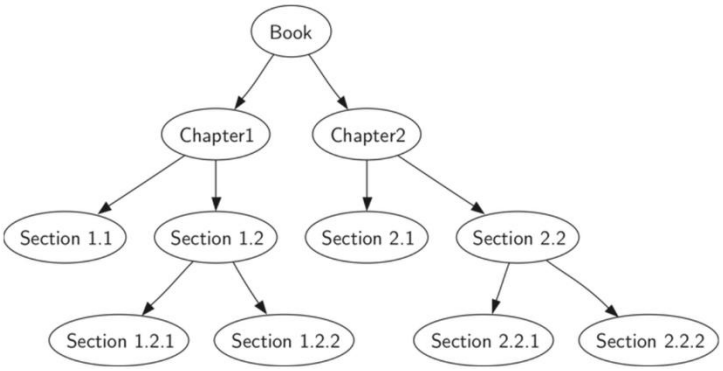


图 5：用树结构来表示一本书

设想你要从头到尾阅读这本书。前序遍历恰好符合这种顺序。从根节点（书）开始我们按照前序遍历的顺序来阅读。我们递归地前序遍历左子树，在这里是第一章，我们继续递归地前序遍历访问左子树第一节 1.1。第一节 1.1 没有子节点，我们不再递归下去。当我们阅读完 1.1 节后我们回到第一章，这时我们还需要递归地访问第一章的右子树 1.2 节。由于我们先访问左子树，我们先看 1.2.1 节，再看 1.2.2 节。当 1.2 节读完后，我们又回到第一章。之后我们再返回根节点（书）然后按照上述步骤访问第二章。

由于用递归来编写遍历，前序遍历的代码异常的间接简洁优美。表 2 给出了一个二叉树的前序遍历的 Python 代码。

表 2：

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

我们也可以把前序遍历作为二叉树类中的内置方法，这部分代码如表 3 所示。注意这一代码从外部移到内部所产生的变化。概括的说，我们我们只是将“tree”换成了“self”。但是我们也要修改代码的主体。内置的方法在递归进行前序遍历之前必须检查左右子树是否存在。

表 3：

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

内置和外置这两种方法哪种更好一些呢？一般来说外置函数更好。原因在于你很少需要单纯遍历整个树。多数情况下你只是想利用基本的遍历方法来实现其他的事情。事实上我们马上就会看到后序遍历的算法和我们之前写的表达式树求值的代码关系紧密。因此我们接下来将按照外部函数的形式书写遍历的代码。

后序遍历的代码如表 4 所示，它除了将 print 语句移到末尾之外和前序遍历的代码几乎一样。

表 4：

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

我们已经见过了后序遍历的一种一般应用，也就是表达式树求值。我们再来看表 1，我们先求左子树的值，再求右子树的值，然后将它们利用根节点的运算连在一起。假设我们的二叉树只储存表达式树的数据。我们来改写求值函数并尽量模仿后序遍历的代码，如表 5 所示。

表 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()
```

我们发现表 5 的形式和表 4 是一样的，区别在于在表 4 中我们输出键而在表 5 中我们返回键。这使我们可以通过第 6 行和第 7 行将递归得到的值储存起来。之后我们利用这些保存起来的值和第 9 行的运算符一起运算。

在这节的最后我们来看看中序遍历。在中序遍历中，我们先访问左子树，之后是根节点，最后访问右子树。表 6 给出了中序遍历的代码。我们发现这三种遍历的函数代码只是调换了输出语句的位置而不改动递归语句。

表 6：

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

如果我们对表达式树进行一个简单的中序遍历我们将得到没有圆括号的原始表达式。我们尝试修改中序遍历的算法使我们得到全括号表达式。只要做如下修改：在递归访问左子树之前输出左括号，然后在访问右子树之后输出右括号。修改的代码见表 7.

表 7：

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

我们发现 printexp 函数对每个数字也加了括号，这些括号显然没必要加。在本章练习的最后会有练习要求你修改 printexp 函数来去除这些括号。

二叉堆 Binary Heap 实现的优先队列

在前面的章节里我们学习了“先进先出”（FIFO）的数据结构：队列（Queue）。队列有一种变体叫做“优先队列”（Priority Queue）。优先队列的出队（Dequeue）操作和队列一样，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(Enqueue)操作就比较复杂，需要将数据项根据优先级尽量挤到队列前方。我们将会发现，对于下一章要学的图算法，优先队列是很有用的数据结构。

我们很自然地会想到用排序函数和队列的简单方法来实现优先队列。但是，在列表里插入一个数据项的复杂度是 O(n)，列表排序的复杂度是 O（nlogn）。我们可以用别的方法来降低复杂度。一个实现优先队列的经典的方法便是采用二叉堆（Binary Heap）数据结构。二叉堆能将优先队列的入队和出队复杂度都保持在 O(logn).

二叉堆的有趣之处在于，其逻辑结构上像二叉树，却是用非嵌套的列表来实现的！二叉堆有两个常见变量:最小成员 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）”。在这一节里我们会使用最小堆。我们把最大堆的使用留作练习题。

二叉堆操作

数据结构二叉堆的基本操作定义如下：

- BinaryHeap（）：创建一个空二叉堆对象
- insert（key）：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

动态代码 1 所示是一些二叉堆方法的示例。可以看到无论我们以哪种顺序把数据项添加到堆里，每次都是移除最小的数据项。我们接下来要集中实现这种想法。

```
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())
```

二叉堆实现

二叉堆结构性质

为了使堆操作高效运行，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”。平衡的二叉树树根拥有相同数量的节点。在我们的堆实现中，我们采用“完全二叉树”的结构来近似实现“平衡”。完全二叉树，指每个内部节点都有两个子节点，最多可有一个节点例外。图 1 所示是一个完全树。

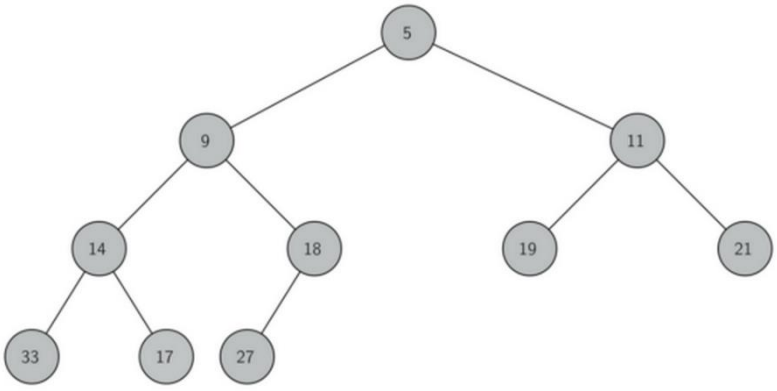


图 1：完全树

另一个有趣的特性是我们用单个列表就能实现完全树。我们不需要使用节点，标记或嵌套列表。因为对于完全树，如果节点在列表中的下标为 p，那么其左子节点下标为 2p.类似的，右节点为 2p+1.当我们找任何节点的父节点时，可以直接使用 python 的整数除法。如果节点在列表中下标为 n，那么父节点下标为 n//2.图 2 所示是一个完全树和树的列表表示。注意父节点与子节点之间 2p 与 2p+1 的关系。完全树的列表表示和整个数据结构的性质，让我们

能够使用简单的数学方法高效地遍历一棵完全树。这也带来了二叉堆的高效实现。

堆次序的性质

我们用来在堆里储存数据项的方法依赖于维持堆次序。所谓堆次序，是指堆中任何一个节点 x，其父节点 p 中的 key 均小于等于 x 中的 key.图 2 所示是具备堆次序性质的完全树。

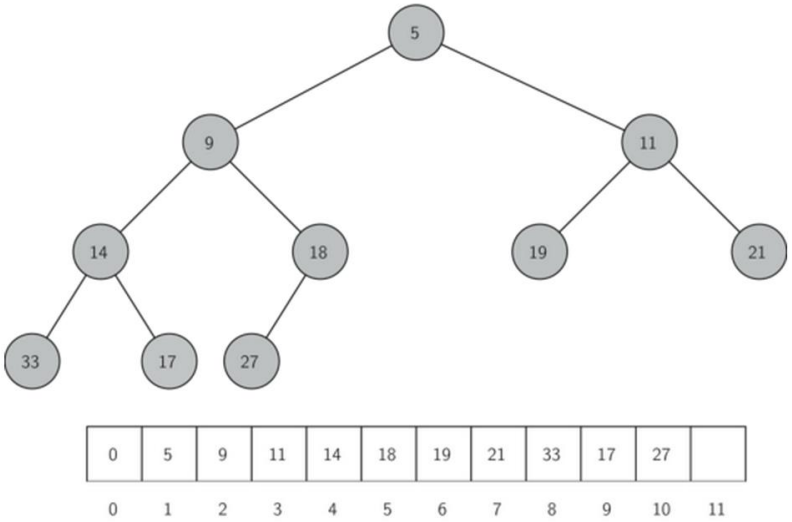


图 2：完全树和它的列表表示

二叉堆操作的实现

接下来我们开始实现二叉堆。因为可以采用一个列表来保存堆数据，构造函数仅仅需要初始化一个列表和一个 currentSize 来表示堆当前的大小。表 1 所示是是构造二叉堆的 python 代码。其中表首下标为 0 的项并没有用到，但为了后面代码可以用到简单的整数乘除法，仍保留它。

表 1：

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

我们接下来要实现的是 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。然而新 key 简单地添加在列表末尾，显然无法保持堆次序。但我们可以通过比较新加入的数据项和父节点的方法来重新达到堆次序。如果新加入的数据项比父节点要小，可以把与父节点互换位置。图 2 所示是一系列交换操作来使新加入数据项“上浮”到正确位置。

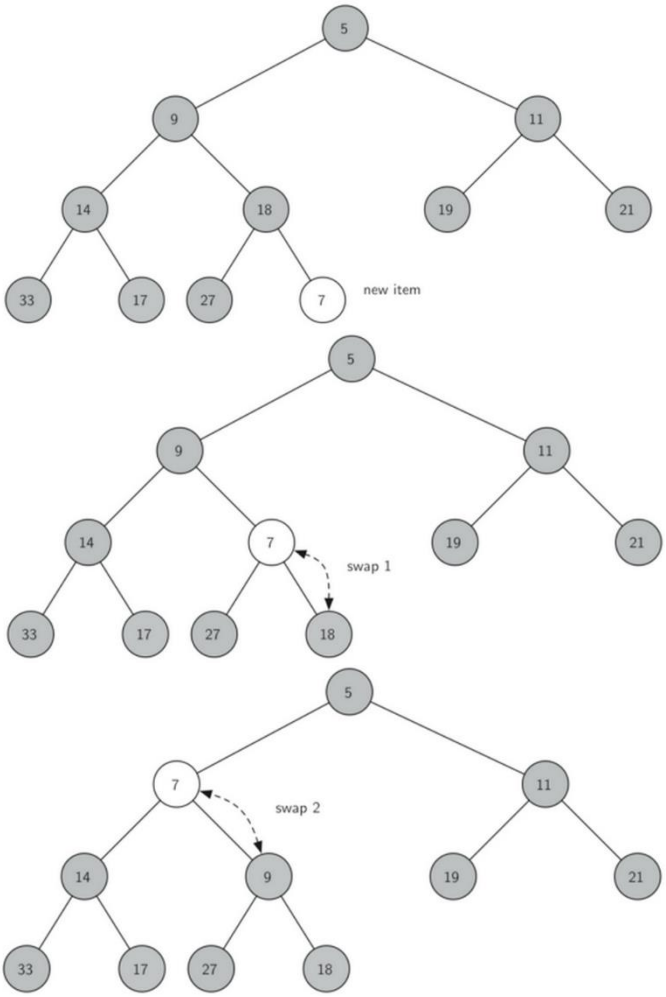


图 2：新节点“上浮”到其正确位置

当我们让一个数据项“上浮”时，我们保证了新节点与父节点以及其他兄弟节点之间的堆次序。当然，如果新节点非常小，我们可能仍需要把它向上交换到另一个层级。事实上，我们需要不断交换，直到到达树的顶端。表 2 所示是“上浮”方法，它把一个新节点“上浮”到其正确位置来保持堆次序。这里很好地体现了我们之前没有用到的元素 0 的重要性。只需要做简单的整数除法:将当前节点的下标除以 2，我们就能计算出任何节点的父节点。

如图 3，我们已经可以写出 insert（key）方法的代码。insert（key）方法很大一部分是由 percUp 函数完成的。当树添加新节点时，调用 percUp 就可以将新节点放到正确的位置上。

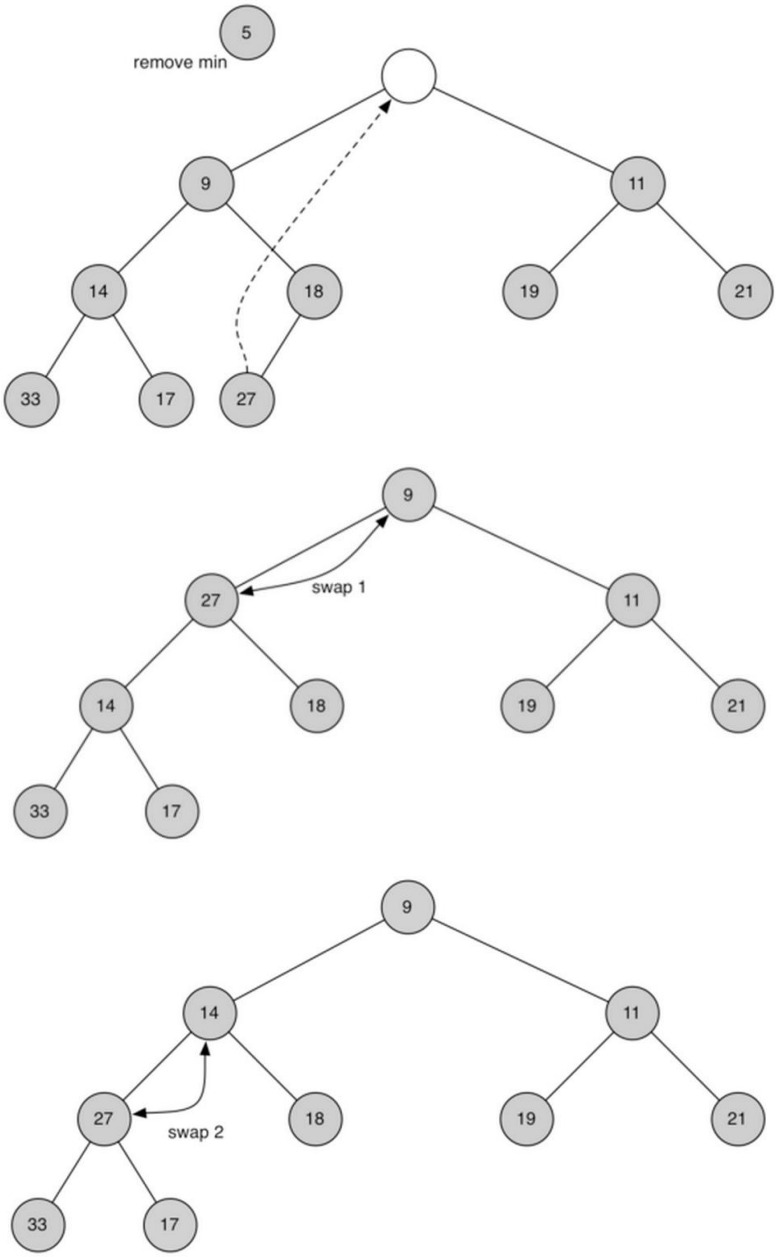
表 2：

```
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
```

表 3：

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

我们已经写好了 insert（key）方法，那我们来看看 delMin（）方法。堆次序要求根节点是树中最小的数据项，因此很容易找到最小项。比较困难的是移走根节点的数据项后如何维护堆结构和堆次序。我们可以分两步走。首先，用最后一个节点来代替根节点。移走最后一个节点维持了堆结构的性质。这么简单的替换，还是可能破坏堆次序。这就要用到第二步:将新节点“下沉”来恢复堆次序。图 3 所示是一系列交换操作来使新节点“下沉”到正确位置。



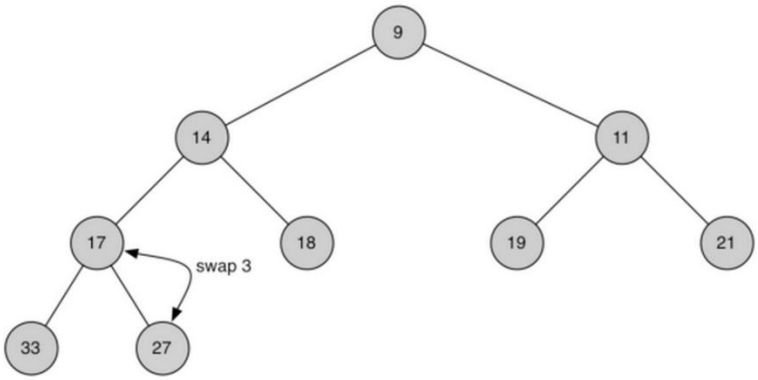


图 3：新节点下沉

为了维持堆次序，我们需将新的根节点沿着一条路径“下沉”，直到比两个子节点都小。在选择下沉路径时，如果新根节点比子节点大，那么选择较小的子节点交换下沉。表 4 所示是下降新节点所需的 percDown 和 minChild 方法的代码。

表 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
```

表 5 所示是 delMin()操作的代码。可以看到困难的部分由一个辅助函数处理，即 percDown。

表 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
```

有关二叉堆的最后一部分便是找到方法从无序表生成一个“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插入到堆中。对于一个排好序的列表，我们可以用二分搜索找到合适的位置来插入下一个 key，操作复杂度是 O(logn).然而插入一个数据项到列表中间需要将列表其他数据项移动为新节点腾出位置，操作复杂度是 O(n).因此用 insert（key）方法的总代价是 O(nlogn)。其实，我们能直接将整个列表生成堆，将总代价控制在 O（n）。表 6 所示是生成堆的操作。

表 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
```

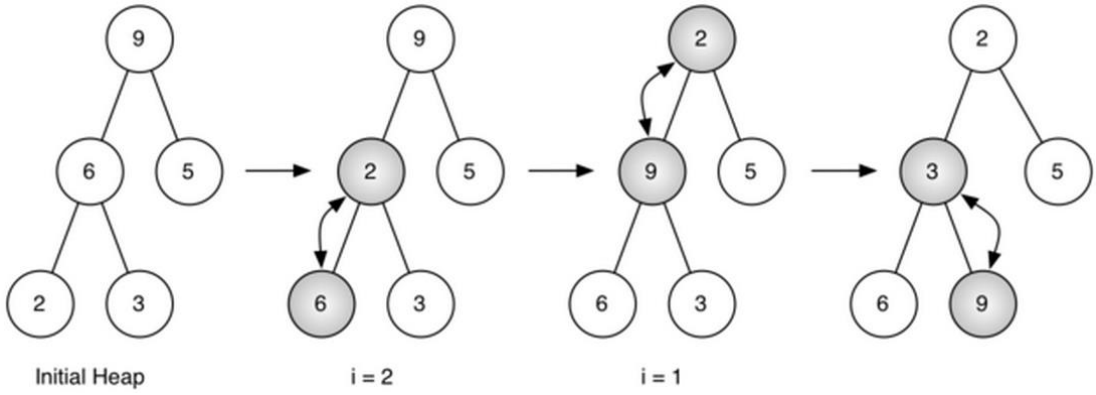


图 4：从列表[9, 6, 5, 2, 3]生成一个二叉堆

图 4 所示是 buildHeap（list）方法将初始树[9, 6, 5, 2, 3]中的节点移动到正确的位置时所做的交换操作。尽管我们从树中间开始，然后回溯到根节点，但 percDown 方法保证了最大子节点总是“下沉”。因为堆是完全树，任何经过中间点的节点都是叶节点，因此没有子节点。注意，当 i=1 时，我们从根节点开始下沉，这就需要大量的交换操作。可以看到，图 4 最右边的两颗树，首先 9 从根节点的位置上移走，移到下一层级之后，percDown 进一步检查它此时的子节点，保证它下降到不能下降为止，即下降到正确的位置。这就导致了第二次交换：9 和 3 的交换。由于 9 已经移到了树最底层的层级，便无法进一步交换了。比较一下图 4 所示的一系列交换的列表表示和树表示是很有用的。

```
i = 2  [0, 9, 5, 6, 2, 3]
i = 1  [0, 9, 2, 6, 5, 3]
i = 0  [0, 2, 3, 6, 5, 9]
```

动态代码 1 所示是完全二叉堆的实现。

```
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]:
```

完全二叉堆示例（completeheap）

能在 O(n)的代价下能生成二叉堆看起来可能有点神秘，并且其证明超出了本书的范围。但是，要理解用 O(n)的代价能生成堆的关键是记住 logn 因子来源于树的高度。而对于 buildHeap 方法中许多操作来说，树的高度比 logn 要小。

既然我们已经能在时间复杂度为 O(n)情况下从列表生成堆，那么在本章的最后我们可以做一个练习：用堆来构建一个排序算法来对列表进行复杂度为 O(nlogn)的排序。

二叉搜索树

我们已经看到在一个集合中得到键值对的两种不同的方法。回忆一下这些集合是如何实现 ADT MAP 的。这两种我们讨论的 ADT MAP 的实现方式是列表的二分查找和散列表。在这个部分，我们将要学习二叉搜索树作为从键指向值的另一种方式，在这种情形中我们对数据在树中的实际位置不感兴趣，但是我们对用二叉树结构来提供更有效率的搜索感兴趣。

搜索树操作

在我们看这种实现方式之前，让我们回顾一下 ADT MAP 提供的接口。我们会发现，这种接口和 Python 的字典非常相似。

Map()创造了一个新的空 map。

put(key,val) 在 map 中增加了一个新的键值对 。如果这个键已经在这个 map 中了，那么就用新的数据来代替旧的数据。

get(key) 提供一个键，返回 mao 中保存的数据，或者返回 None。.

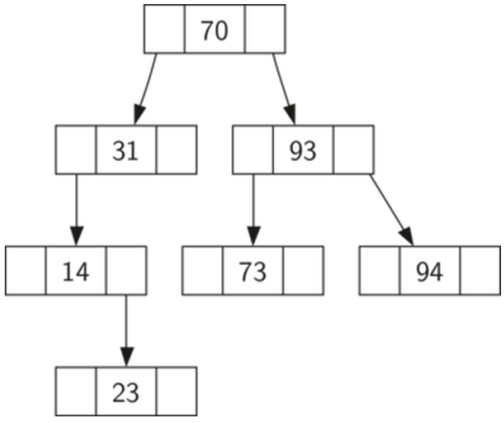
del 用 del map[key]这种形式从 map 中删除键值对。

len() 返回 map 中保存的键值对的数目

in 对给定的键是否存在 map 中进行判断，如果所给的键在 map 中，返回 True。

搜索树实现

一个二叉搜索树，如果左边的子树中键值 Key 都小于父树，而右边边的子树中键值 Key 都大于父树，我们将这种树称为 BST 搜索树。如上节所述，当我们实现 Map 方法时，BST 方法将引导我们实现这一点。图 1 显示了二叉搜索树的这一特性，显示的键没有任何关联的值。注意：这种属性适用于每个父树和子树。所有在左子树的键值是小于根的键值的，所有的键值在右子树均大于根。



图一：一个简单的二叉搜索树

现在你知道一个二叉搜索树是什么了，我们再看看二进制搜索树的构造:在我们插入以下键值显示的顺序后，图 1 搜索子树代表存在的节点：70,31,93,94,14,23,73。因为 70 是被插入到子树的第一个关键，它变成了根。接下来，31 小于 70，因此是 70 的左子树。接下来，93 大于 70，因此是 70 的右子树。我们现在有两个层次的子树填充，所以接下来的重点，将会是 31 或者 93 左右的子树。由于 94 大于 70 和 93，会成为 93 的右子树。同样，14 小于 70 和

31，因此成为 31 的左子树。23 也低于 31，因此必须是 31 的左子树。然而，它大于 14，因此也是 14 的右子树。

执行二进制搜索树，我们将使用节点和引用方法，这类似于一个我们用来实现链表和表达式树的过程。因为我们必须能够创建和使用一个空二叉搜索树，所以我们的实现将使用两类：第一类我们称为 BinarySearchTree，第二类我们称之为 TreeNode。 BinarySearchTree 类有一个 TreeNode 类的引用作为二叉搜索树的根，在大多数情况下，外部方法中要定义一个函数，以便在类外看看子树是否是空的，如果在子树上有节点，要求有基本的方法，在 BinarySearchTree 类中，把根定义为参数。在这种情况下，随着一些其他的多功能 BinarySearchTree 类构造函数的代码，如果子树是空的或者我们想在子树的根删除的键值，我们就必须采取特别行动。如表 1 所示。

表 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，使方法更容易做到。如列表 2 所示，一个子树节点的构造，是这些在 BinarySearchTree 类的辅助函数中实现。正如你可以看到上述的这些辅助功能，可以根据自己的位置是一个子树对应一个节点（左或右），还是子树的节点。TreeNode 类也将明确地为每个节点的属性保持父树跟踪的应用。当我们讨论 del 算子的实现时，你将看到为什么这是重要的。

对列表 2 中的子树节点，实现的另一个有趣的方面是，我们使用 Python 作为可选参数。可选的参数很容易让我们在几个不同的情况下创建一个子树节点，有时虽然我们已经有了一个父树和子树，但仍会想建立一个新的子树节点。与现有的父树和子树一样，我们可以通过父树和子树作为参数。有时我们也会创建一个键-值对的树，我们不会再用父树或子树做任何参数。在这种情况下，我们使用可选参数的默认值。

表 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，我们用在 BinarySearchTree 类 shell 和 TreeNode，它将使我们建立我们的二叉搜索树。该方法是一个类的方法，称之为 BinarySearchTree。它将检查是否已经有一根子树。如果没有，我们将创建一个新的子树节点并安装它，作为子树的根。如果一个根节点已经到位，我们就调用它自己，进行递归，用辅助功能_put 按下列算法的搜索树：

- 我们在当前的关键节点的二进制子树，从子树的根开始搜索比较新的键值，如果新的键值是小于当前节点。搜索左子树，如果新的关键是大于当前节点，搜索右子树。
- 当无左（或右）子树的搜索，我们发现的位置就是应该在子树中进行安装新节点的位置。
- 向子树中添加一个节点，在上一步发现插入对象的位置创建一个新的TreeNode。

表 3 显示，在子树中插入新节点的 Python 代码的_put 函数，编写递归要按照上述的步骤。注意，当一个新的子树插入时，CurrentNode 为父树传给新的子树。

对插入重复键值处理的好不好，是我们实施中遇到的一个重要问题，我们的子树实现了复制的键值，它将创建一个与原来的节点具有相同键值的右子树作为新节点。这样做的结果是，我们用了一个更好的方式来处理复制键值的插入和替换，旧的值被新键所关联的值替换，但新的关键节点将不会在搜索过程中被发现。我们把修复这个 bug，作为练习留给你。

表 3：

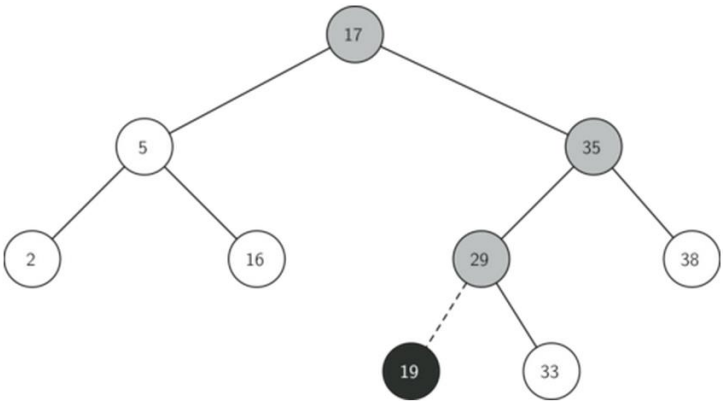
```
def put(self,key,val):
    if self.root:
        self._put(key,val,self.root)
    else:
        self.root = TreeNode(key,val)
    self.size = self.size + 1

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)
```

随着 put 方法的定义，调用（参见清单 4）的方法，我们可以很容易地重载[ ]作为操作符，添加 __setitem__的方法。这使我们能够编写 Python 语句。比如写 myZTiptree ['plymouth’] = 55446 一样，这看上去就像 Python 字典。

列表 4

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

图 2 显示了用于插入新节点到一个二叉搜索树的过程。如下的灰色节点显示了这个插入过

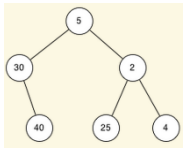


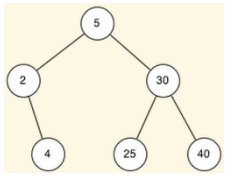
程中的树的节点遍历的顺序。

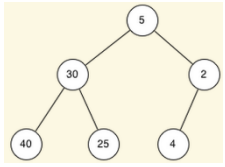
图 2: 插入一个节点键值 Key= 19

自我检测：

问题 33：如下子树显示了键值插入的顺序是 5，30，2，40，25，4 的正确的二进制搜索树。

a) 

b) 

c) 

一旦有了子树的构建，接下来的任务就是为一个给定的关键值的实现检索。GET 方法更容易因为它搜索子树进行递归，直到发现不匹配的叶节点或找到一个匹配的键值。当一个匹配的键值被发现后，就会找到存储在节点的负载值。

表 5 显示了代码_get 和__getitem__。用_get 方法搜索代码，使用相同的逻辑，选择左或右子树用_put 方法。请注意，使用_get 方法获得返回一个树节点时，除了有效载荷等灵活的数据辅助方法外，子树还可以使用 BinarySearchTree 方法。二者我们都要学会利用。

通过实施__getitem__方法，我们可以写若干 Python 语句，使它们看起来就像我们访问字典一样，而事实上我们只是在用一个二叉搜索树，例如 Z = myziptree ['fargo’]。正如你可以看到，所有的__getitem__方法大同小异。

表 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)
```

__contains__调用 get 函数，看得到返回值返回真或假，如果它是 True 就返回值，如果它是 False 就返回 False。我们就可以用 BinarySearchTree 一__contains__方法来操作。如表 6 所示

表 6：

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

回顾__contains__重载了操作符，这允许了我们写这样的语句：

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

最后，我们来关注最具挑战性的方法：在二进制搜索树，删除一个键值（参见列表 7）。首要任务是找到要删除的搜索树的节点。如果子树有一个以上的节点，我们使用_get 方法搜索找到需要删除的节点；如果子树只有一个节点，这意味着我们要分解子树的根，但是我们仍然必须检查以确保根的键值是否匹配要删除的键值。在以上两种情况下，如果未发现键值，del 算子就会报错误。

表 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)
```

一旦我们发现含有要删除的关键节点，有三种情况，我们必须考虑：

1. 要删除的节点没有子树 (见图三).
2. 要删除的节点没有子树(见图四).
3. 要删除的节点没有子树(见图五)

第一种情况是简单的（参见列表 8）。如果当前节点没有子树，所有我们需要做的是引用删除节点和删除父树节点的方法。本例的代码显示在这里

表 8：

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

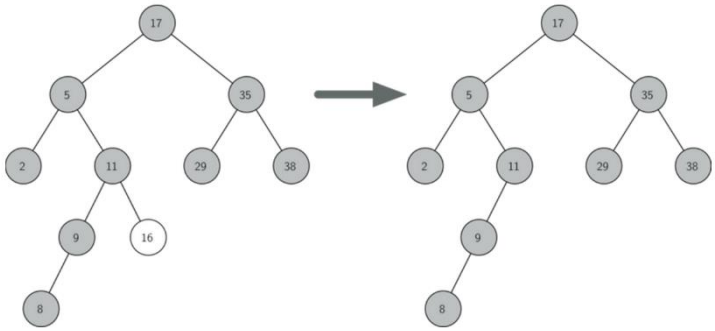


图 3：删除节点 16，而这个节点没有子树

情况 2 只是稍微复杂（参见列表 9）。如果一个节点只有一个子树，那我们可以促进子树代替其父树的位置。在这种情况下，代码如下表所示。你看这段代码，就会看到有六种情况考虑。由于有一个左或右子树的情况，我们将只讨论在当前节点有左子树的情况下，然后做对称对称。决策过程如下：

如果当前节点是左子树，那我们只需要更新左子树指向当前节点的父树引用，然后将引用从父树左子树更新到当前节点的左子树。

如果当前节点是一个正确的子树，那我们只需要将引用从右子树更新到指向当前节点的父树，然后从父树更新到指向当前节点的右子树。
如果当前节点没有父树，它必须是根。在这种情况下，我们只需更换键值，有效载荷，leftChild 和 rightChild。方法是由根调用 replaceDodeDNata 右子树数据的方法。

表 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

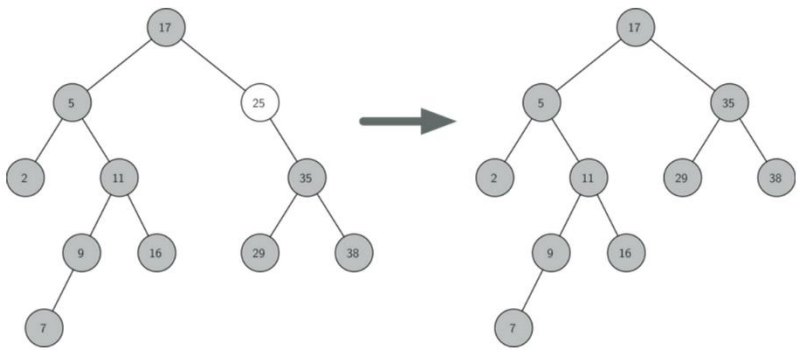


图 4: 删除节点 25,它是只有一个子树的节点

第三种情况是最难处理的情况，（参见列表 10）。如果一个节点有两个子树，我们可以简单地推断其中一个作为节点的位置是不可能的，这就需要寻找一个节点，用来代替一个计划删

除的子树，我们需要的这个节点，对现有的左、右子树以及保存的二叉搜索树都有影响。该节点不只是一个节点，更是子树下一个最大的关键节点。我们称这个节点为继任者，然后我们将一路寻找继任者，继任者保证没有一个以上的子树，所以既然我们已经知道如何处理这两种情况，我们就可以实现它了。一旦继任者已被删除，我们把它放在将被删除的子树节点处。

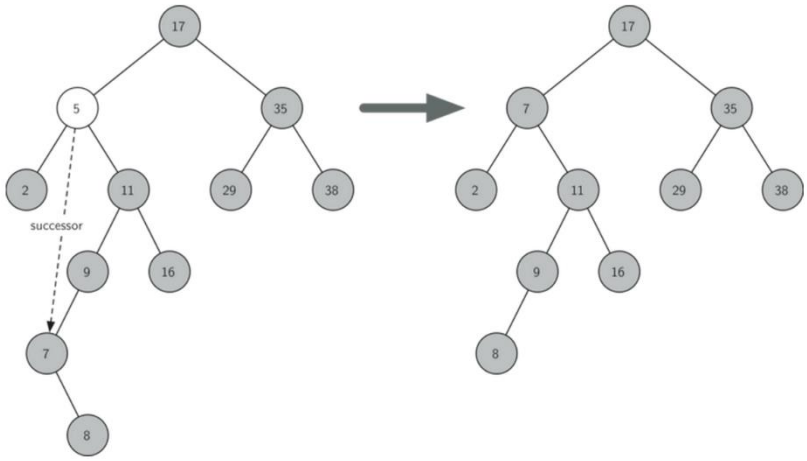


图 5：删除节点 5，有两个子树的节点

第三种情况是，是处理一个带着两个子树的节点，代码如下表所示。 注意，我们是用辅助方法 findSuccessor 和 findMin 来找到继任者的。而删除继任者，我们利用方法 spliceOut。我们用 spliceOut 的原因是它能帮助我们拼出节点，做出正确的变化。我们也可以调用递归删除，但那样我们就会浪费时间反复寻找关键节点。

表 10：

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

代码找到继任者所示（参见列表 11），你可以看到一个 TreeNode 类的方法。这个代码利用二叉搜索子树的中序遍历，按从最小到最大相同的性质，打印出子树中的节点。有三种情况时需要考虑继任者：

如果节点有右子树，那么继任者是在右子树中最小的关键。

如果节点没有右子树，是其父树的左子树，那么父树是继任者。

如果节点是其父树的右子树，而本身无右子树，那么继任者这个节点是其父树的继任者，而不包括这个节点。

现在首要的问题是删除一个节点，利用二进制搜索子树的方法 findMin，找到子树中的最小关键。而 findSuccessor 方法，我们将在本章最后练习中探讨其他用途。

findMin 方法是找到子树中的最小键值。要相信，最小值在任何二叉搜索子树中都是子树的左子树。因此 findMin 方法只简单地追踪左子树，直到找到没有左子树的叶节点。

表 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
```

我们还需要看看二进制搜索子树的最后一个接口方法。假设我们已经按顺序简单地遍历了子树上所有的键值，就是我们用字典做的事，就会有疑问：为什么不是树？我们已经知道如何使用中序遍历二叉树的中序遍历算法，然而，写一个迭代器需要更多一点的工作，因为每次调用迭代器时，一个迭代器只返回一个节点。

Python 提供了一个非常强大的功能，使用时创建一个迭代器。调用函数时的 yield。yield 类似于它返回一个值给调用者。然而，yield 也需要额外的步骤来暂停它的功能状态，以便下

次调用函数继续执行时早做准备。它的功能——创建可迭代的对象称为生产的功能。
二叉树为迭代器代码如下表所示。仔细看看这个代码：乍一看，你可能会认为代码是非递归的。但是请记住，__iter__重写 for x in 的操作符运行迭代，所以它确实是递归！因为它是递归的迭代器__iter__在 TreeNode 实例方法，所以__iter__是在 TreeNode 类定义。

```
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
```

在这一点上，你可能想下载含 BinarySearchTree 和 TreeNode 类完整代码的文档。

```
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)
        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])
```

搜索树分析

通过现在完成的二叉树的实现，我们将对我们已经实现的方法进行一个快速的分析。让我们先看一下 put 这个方法。对它的执行效果造成限制的是二叉树的高度。回想一下语法阶段，树的高度指的是根和最深的叶节点之间的边的数目。高度作为一种限制因素是因为当我们在树中寻找一个插入节点的合适位置时，我们将会需要最多对树的每一级做比较。

二叉树的高度可能是什么样的呢？对这个问题的答案取决于键是怎么被加到树中的。如果键是以一个随机的顺序加到树中的，那么当节点的数目为 N 时，树的高度将会大概在 LOGN。这是因为键是随机地散布的，大约有一半的节点会比根节点大，另一半会比它小。记住在二叉树中，根上有一个节点，下一级有两个，再下一集有四个。当级数为 d 时，这一级的节点数目为 2 的 d 次方。当 h 代表树的高度的时候，一个完美平衡二叉树中节点的总数目是 2 的 h+1 次减 1。

一个完美平衡二叉树中的节点数目和一个平衡二叉树中的左子树和右子树的数目是一样多的。当树中有 N 个节点时 Put 执行的最差结果是 log2n 的复杂度。要注意到这与之前段落里说的关系是逆运算的关系。所以 log2n 给了我们树的高度，这代表了把一个新节点插入一个合适位置所需要做的搜索中比较的次数。

不幸的是，通过插入排序过后的键，建造一个高度为 n 的搜索树是可能的。图 6 就展示了这种树的一个例子，在这个情形下，这个 put 方法的执行复杂度为 O(N)。

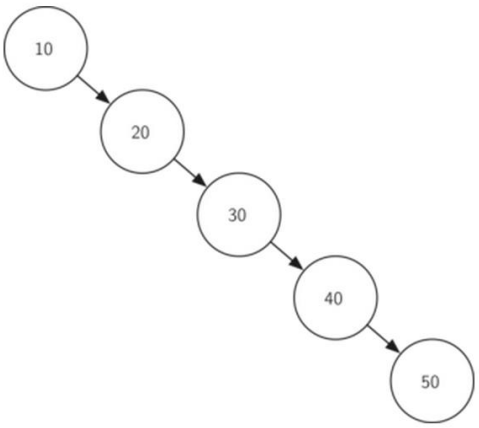


图 6 一个倾斜的二叉搜索树会有一个很差的表现

现在你已经理解了 put 方法的执行会受到树的高度的限制，你可能猜测其他的方法，get，in 和 del 也是受限制的，因为要搜索树来发现键，在最坏的情形下，要一直搜索树到底而找不到键。乍一看 del 也许看上去更复杂，因为它也许需要在删除操作完成之前一直查找下一个继承节点。但是记得，寻找继承节点的最坏情形也只是树的高度，这意味着你只需要简单地把工作加倍，因为加倍是乘以一个常数因子，所以它没有改变最坏的情形。

平衡二叉搜索树

在上一节中我们看到研究建立了一个二叉搜索树。我们知道，当树变得不平衡时 get 和 put 操作会使二叉搜索树的性能降低到 O（n）。在这一节中我们将看到一种特殊的二叉搜索树，它可以自动进行调整，以确保树在所有时间保持平衡。这种树被称为 AVL 树，由发明他的人：G.M.Adelson-Velskii 和 E.M.Landis 而命名。

AVL 树实现图（Map）的抽象数据类型，就像一个普通的二叉搜索树，唯一不同的是这棵树的工作方式。为实现我们的 AVL 树我们需要在树中的每个节点加入一个平衡因子（balance factor）并跟踪其变化情况。我们通过比较每个节点的左右子树的高度完成比较。更正式的定义是，一个节点的平衡因子定义为左子树的高度和右子树的高度之差。

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

利用以上的平衡因子的定义，如果平衡因子小于零，我们称子树“左重”（left-heavy）。 如果平衡因子小于零，那么子树是“右重”（right-heavy）。 如果平衡因子是零，树是完美的平衡。为实现 AVL 树的目的，并获得具有平衡的树，我们将定义如果平衡因子是-1，0 或 1,那么这个树是平衡的。一旦树中的节点的平衡因子超出了这个范围，我们需要有一个把树恢复平衡的过程。图 1 是一个不平衡的“右重”树的例子，其中每个节点都标注了平衡因子。

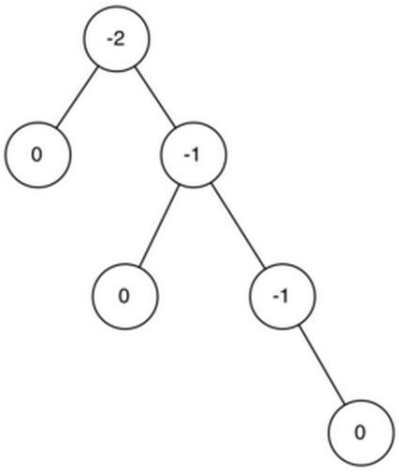


图 1：一个平衡因子不平衡的右重树

AVL 树性能

在我们继续进行之前让我们看看引入这个新的平衡因素的结果。我们的要求是，确保树上总是有一个平衡因子- 1，0，或 1。我们可以通过一些关键操作得到更好的大 0 性能。首先，我们来思考有这个平衡条件后，最坏情况下的树发生了什么变化。有两个可能的考虑，左重树和右重树。如果我们考虑树的高度为 0，1，2 和 3，图 2 举出了在新规则下可能的出现的最不平衡的左重树的例子。

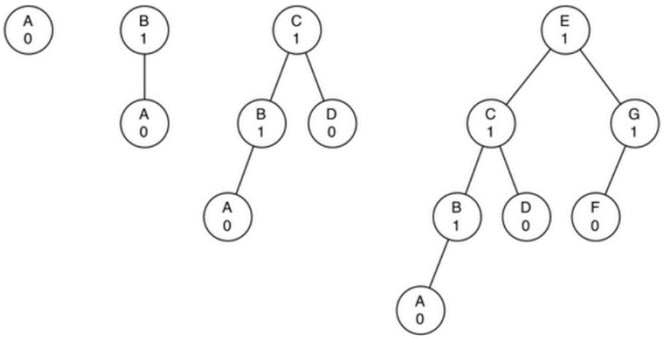


图 2：最坏的情况下，左重的 AVL 树

让我们看看树上的节点的总数。我们看到一棵高度为 0 的树有 1 个节点，一个高度为 1 的树有 1 + 1 = 2 个节点，一个高度为 2 的树有 1 + 1 + 2 = 4，一棵高度为 3 的树有 1 + 2 + 4 = 7 个节点。更普遍的模式，我们看到高度为 h 的树的节点数 h（Nh）是：

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

可能你很熟悉这个公式，因为它和斐波那契序列非常相似。我们可以利用这个公式通过树中的节点的数目推导出一个 AVL 树的高度。在我们的印象中，斐波那契数列和斐波那契数定

义为：

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

数学中一个重要的结果是，随着斐波那契序列的数字越来越大 Fi/Fi−1 越来越接近于黄金比例Φ（Φ= $\frac{1+\sqrt{5}}{2}$）。如果你想看到上式的推导过程你可以查阅数学教科书。我们将简单地使用这个方程近似 F_i:F_i=$\frac{\Phi^i}{\sqrt{5}}$。如果利用这种近似我们可以将 Nh 的方程改写为：

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

通过用黄金比例近似代替斐波那契数列的项我们可以得到：

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

如果我们整理这些方程的项，并两边以 2 为底取对数，然后求解 h，则可以导出：

$$\log N_h + 1 = (H+2)\log\Phi - \frac{1}{2}\log 5$$
$$h = \frac{\log N_h + 1 - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi}$$
$$h = 1.44 \log N_h$$

这个推导告诉我们，在任何时候我们的 AVL 树的高度等于树中节点数以 2 为底的对数的常数（等于 1.44）倍。这对搜索我们的 AVL 树来说是好消息因为它限制搜索复杂度到 O（logN）。

AVL 树实现

既然，我们已经证明，保持一个 AVL 树的平衡将是一个很大的性能提升，让我们看看我们如何通过插入一个新的键值到树来完善程序。因为所有的新键是作为叶节点插入树的，我们知道一个新叶的平衡因子为零，所以我们对刚刚插入的节点没有新的要求。但是一旦有新叶插入我们必须更新其父节点的平衡因子。这新叶会如何影响父节点的平衡因子取决于叶节点是左节点还是右节点。如果新节点是右节点，父节点的平衡因子将减少一个。如果新节点是左节点，父节点的平衡因子将增加一。这种关系可以递归地应用于新的节点的前两个节点，并且有可能影响每一个前面的节点一直到树的根。由于这是一个递归过程，我们可以考察两个更新平衡因子的基本条件：

- 递归调用已达到树的根。.
- 父节点的平衡因子已调整为零。你应该知道，一旦子树平衡因子为零，那么父节点的平衡因子不会改变。

我们将 AVL 树作为 BinarySearchTree 的子类实现。首先，我们将覆盖_put 方法写一个新的

updateBalance 辅助方法。这些方法如表 1 所示。除了第 7 行和第 13 行对 updateBalance 的调用，你会注意到_put 的定义和简单的二叉搜索树是完全相同的。

表 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)
```

_put 的大部分工作正是由这个新的 updateBalance 方法完成的。这实现了我们刚才描述的递归过程。这个再平衡方法首先检查当前节点是否十分不平衡，以至于需要重新平衡（16 行）。如果当前节点需要再平衡，那么只需要对当前节点进行再平衡，而不需要进一步更新父节点。如果当前节点不需要再平衡，那么父节点的平衡因子需要调整。如果父节点的平衡因子非零，那么算法通过母节点递归调用 updateBalance 方法继续往上传递到树的根。

当对一棵树进行再平衡是必要的，我们该怎么做呢？有效的再平衡是使 AVL 树很好地工作而不牺牲性能的关键。为了让一个 AVL 树恢复平衡，我们会在树上执行一个或多个“旋转”（rotation）。

为了了解什么是旋转，让我们看一个很简单的例子。思考一下图 3 的左半部分树。这棵树是不平衡的，平衡因子为- 2。为了让这棵树平衡我们将根的子树节点 A 用一个左旋转。

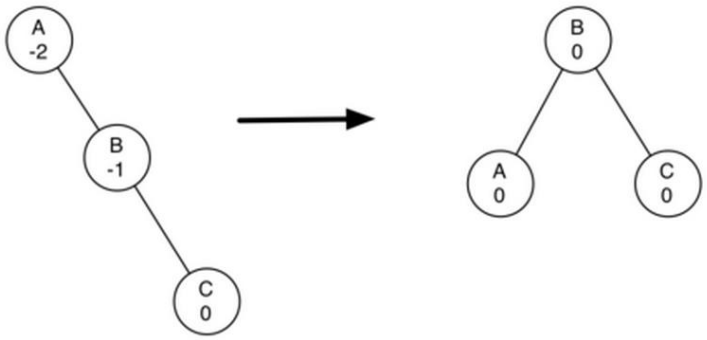


图 3：使用左旋转变换不平衡树

执行一个左旋转我们需要做到以下几点：

- 使右节点（B）成为子树的根。
- 移动旧根（A）到新根的左节点。
- 如果新根（B）原来有左节点，那么让原来 B 的左节点成为新根左节点（A）的右节点。注：由于新根（B）是 A 的右节点，在这种情况下移动后的 A 的右节点一定是空的。这使得我们不用多想就可以直接给移动后的 A 添加右节点。

虽然这个程序概念上相当简单，但是代码的细节有点棘手，因为为了维持二叉搜索树的所有性质，必须以绝对正确的顺序把节点移来移去。此外，我们需要确保正确地更新了所有的 parent 指针。

让我们通过观察一个稍微复杂的树来说明右旋转。图 4 的左边展现了一棵“左重”的树，根的平衡因子为 2。执行一个正确的右旋转，我们需要做以下几点：

- 使左节点（C）成为子树的根。
- 移动旧根（E）到新根的右节点。
- 如果新根（C）原来有右节点（D），那么让 D 成为新根右节点（E）的左节点。注：由于新根（C）是 E 的左节点，在这种情况下移动后的 E 的左节点一定是空的。这使得我们不用多想就可以直接给移动后的 E 添加左节点。

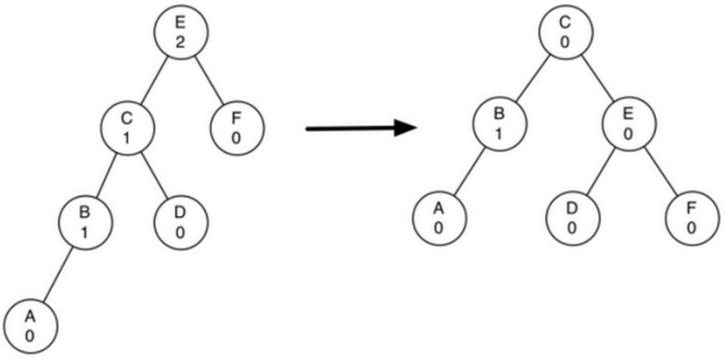


图 4：使用正确的旋转变换不平衡树

既然现在你已经看过了旋转的过程，了解了旋转的方法，让我们看看代码。表 2 同时显示了右旋转和左旋转代码。在第 2 行，我们创建一个临时变量来跟踪新的子树的根。正如我们之前所说的新的根是旧根的右节点。现在，右节点已经经被存储在这个临时变量。我们将旧根的右节点替换为新根的左节点。

下一步是调整两节点的父指针。如果新根 t 原来有左节点，左节点的新父节点变成老根。新根的父节点将成为旧根的父节点。如果旧根是整个树的根，那么我们必须让整棵树的根指向这个新的根。否则，如果旧的根是左节点，那么我们改变左节点的父节点到一个新的根；否

则，我们改变右节点的父节点到一个新的根（行 10-13）。最后我们设置的旧根的父节点成为新的根。这里有很多复杂的中间过程，所以我们建议你一边浏览这个函数的代码，一边看图 3。rotateRight 方法和 rotateLeft 是对称的，所以我们会让您自学 rotateRight 代码。

表 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 需要一些解释。在这两行我们更新旧根和新根的平衡因子。因为所有其他的动作是移动整个子树，被移动的子树内的节点的平衡因子不受旋转的影响。但我们如何在没有完全重新计算新的子树的高度的情况下更新平衡因子？下面的推导将让你明白，这些代码都是正确的。

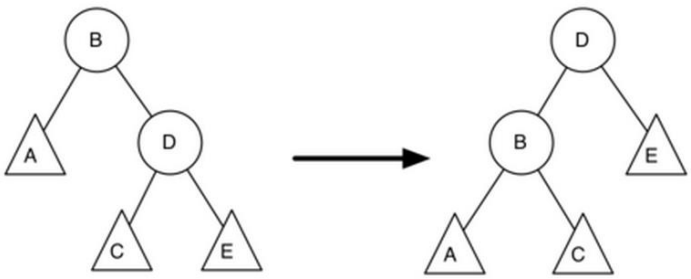


图 5：左旋转

图 5 显示了一个左旋转。B 和 D 是中心节点，A，C，E 是其子树。让 hX 表示以 X 为根节点的子树的高度。通过定义我们知道：

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

但我们知道，D 的旧的高度也可以通过 1+max(hC,hE)给定，也就是说，D 的高度为两子树

高度中较大者加 1。记住，hC 和 hE 没有改变。所以，把上式代入第二个方程，可以得到：

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

然后两方程作差。下面是作差的步骤，使用了一些代数方法简化方程 newBal（B）。

$$\begin{aligned} newBal(B) - oldBal(B) &= h_A - h_C - (h_A - (1 + max(h_C, h_E))) \\ newBal(B) - oldBal(B) &= h_A - h_C - h_A + (1 + max(h_C, h_E)) \\ newBal(B) - oldBal(B) &= h_A - h_A + 1 + max(h_C, h_E) - h_C \\ newBal(B) - oldBal(B) &= 1 + max(h_C, h_E) - h_C \end{aligned}$$

接下来我们移动 oldBal（B）到方程的右端并利用 max(a,b)−c=max(a−c,b−c)。

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

但是，hE−hC 和−oldBal(D)相同。所以我们可以用另一个相同的说法：max(−a,−b)=−min(a,b)，所以我们可以通过以下步骤完成对 newBal（b）的推导：

$$\begin{aligned} newBal(B) &= oldBal(B) + 1 + max(0, -oldBal(D)) \\ newBal(B) &= oldBal(B) + 1 - min(0, oldBal(D)) \end{aligned}$$

现在方程所有的项都是已知数，我们很容易知道。如果我们记得 B 是 rotRoot，D 是 newRoot，我们可以看出这完全符合 16 行的语句，或：

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

一个类似的推导给出更新节点 D 的方程，以及右旋转后的平衡因子。我们把这些作为你的训练。

现在你可能会认为我们已经做完了。我们知道如何做左右旋转，我们知道我们应该在什么时候做一个向左或向右转动，但看看图 6。由于节点 A 的平衡因子是- 2 我们应该做一个左旋转。但是，当我们在左旋转时会发生什么？

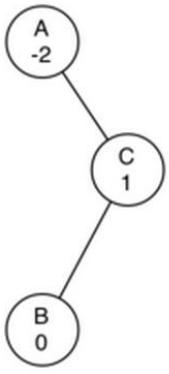


图 6：一个更难平衡的不平衡的树

图 7 告诉了我们，左旋转后，我们仍然不平衡。如果我们要做一个右旋转来试图再平衡，我们回到开始的状态。

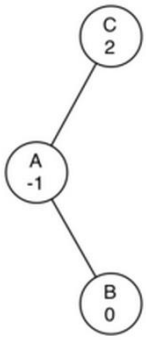


图 7：左旋转后树在其他方向不平衡

要纠正这个问题，我们必须使用以下规则：

- 如果子树需要左旋转使之平衡，首先检查右节点的平衡因子。如果右节点左重则右节点做右旋转，然后原节点左旋转。
- 如果子树需要右旋转使之平衡，首先检查左节点的平衡因子。如果左节点右重然后左节点左旋转，然后原节点右旋转。

图 8 显示了这些规则如何解决我们在图 6 和图 7 中遇到的困境。首先，以 C 为中心右旋转，树变成一个较好的形状；然后，以 A 为中心左旋转，整个子树恢复平衡。

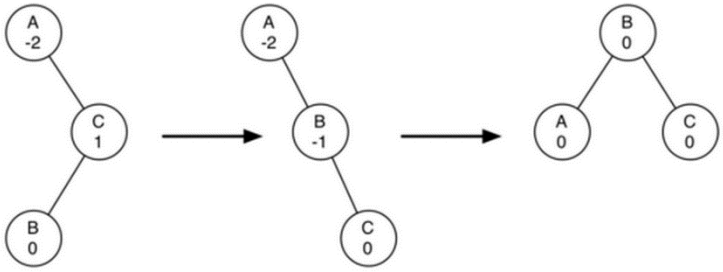


图 8：右旋转后左旋转

实现这些规则的代码可以从我们“再平衡”（rebalance）的方法中找到，如代码 3 所示。上面的第一条规则从第二行 if 语句中开始实现。第二条规则是由第 8 行 elif 语句开始实现的。

表 3：

```
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)
```

本章的“问题讨论”给了你一个需要先左旋转后右旋转的再平衡树的实例。此外，“问题讨论”为你提供机会来平衡一些树，比在图 8 中的树更复杂一点。

通过保持树始终平衡，我们可以确保 get 方法运行的时间复杂度为 O（log2n）。但问题是 put 方法的时间复杂度是多少？我们把 put 方法分解为 put 的每一步操作。由于每一个新节点都是作为叶节点插入的，每一轮更新所有父节点的平衡因子最多只需要 log2n 次操作，每一层一次。如果子树是不平衡的最多需要两个旋转把子树恢复平衡。但是，每个旋转的操作是 O（1）的复杂度，即使我们 put 操作仍然是 O（log2（n））的复杂度。

现在，我们已经实现了一个能使用的 AVL 树，除非你需要删除一个节点。我们把删除节点和随后的更新和再平衡作为你的练习。

ADT Map 实现小结

通过过去两章，我们已经看过了好几种可以用于实现 Map 抽象数据类型的数据结构。列表的二分搜索，哈希表，二叉搜索树和平衡二叉树。作为这一章的结束，让我们总结这些数据结构在 Map 的对键值 key 的操作中的性能（见表 1）。

表 1：比较不同 Map 实现的性能表现

操作	已经排好序的列表	哈希表	二叉搜索树	AVL 树
put	O(n)	O(1)	O(n)	O(log2n)
get	O(log2n)	O(1)	O(n)	O(log2n)
in	O(log2n)	O(1)	O(n)	O(log2n)
del	O(n)	O(1)	O(n)	O(log2n)

小结

这几章我们讨论了树的数据结构。树数据结构使我们能实现很多有意思的算法。在这章我们已经用树数据结构做了下面这些事情：

- 用二叉树对表达式进行语法分析和求值
- 用二叉树实现 ADT map
- 用平衡二叉树（AVL 树）实现 ADT map
- 用二叉树实现最小堆
- 用最小堆实现优先队列

关键词

AVL 树	二叉堆	二叉搜索树
二叉树	子节点	完全二叉树
边	堆顺序属性	高度
中序	叶节点	层数
ADT map	最大最小堆	节点

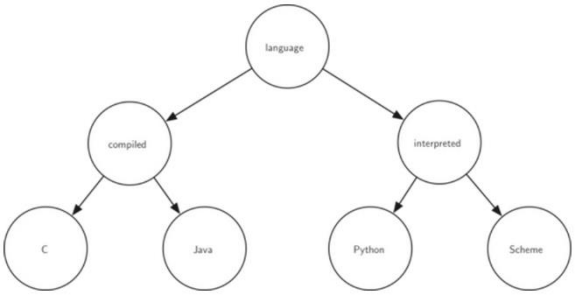
父节点	路径	后序
前序	优先队列	根
旋转	兄弟节点	后继
子树	树	

问题讨论

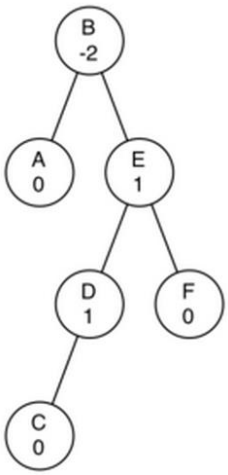
1. 根据下列对树函数的调用，画出相应的树结构：

```
>>> r = BinaryTree(3)
>>> insertLeft(r,4)
[3, [4, [], []], []]
>>> insertLeft(r,5)
[3, [5, [4, [], []], []], []]
>>> insertRight(r,6)
[3, [5, [4, [], []], []], [6, [], []]]
>>> insertRight(r,7)
[3, [5, [4, [], []], []], [7, [], [6, [], []]]]
>>> setRootVal(r,9)
>>> insertLeft(r,11)
[9, [11, [5, [4, [], []], []], []], [7, [], [6, [], []]]]
```

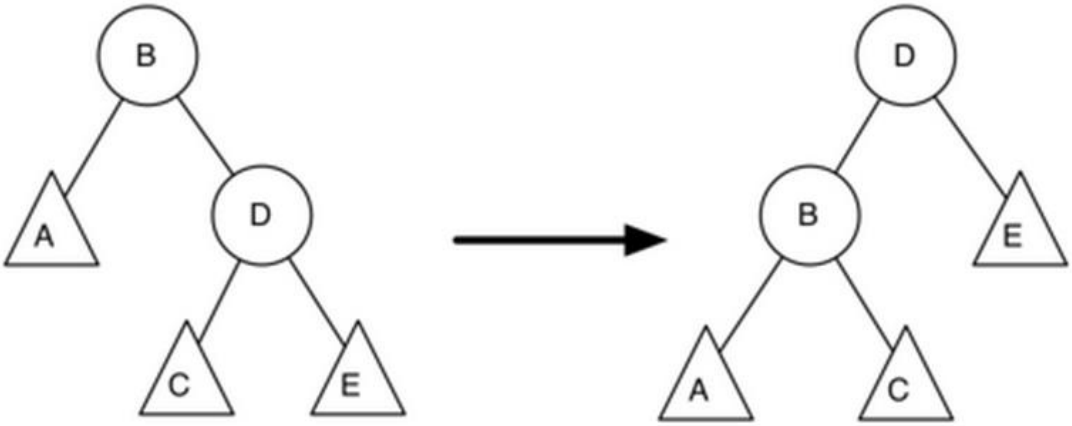
2. 创建(4*8)/6−3 的表达式树，然后画出该算法的流程。
3. 考虑以下整数列表：[1,2,3,4,5,6,7,8,9,10]。通过插入方法生成该列表的树，请画出该二叉树。
4. 考虑以下整数列表：[10,9,8,7,6,5,4,3,2,1]。通过插入方法生成该列表的树，请画出该二叉树。
5. 生成一个随机整数列表。通过一次插入列表中一个整数的方法生成二叉堆树，请画出该二叉堆树。
6. 根据前面问题的列表，通过将列表作为 buildHeap 方法的参数生成二叉堆树，请画出该二叉堆树。
7. 按照顺序插入下列 key 值：68,88,61,89,94,50,4,76,66 和 82，画出生成的二叉搜索树的结果。
8. 生成一个随机整数的列表。通过插入该列表中的整数生成二叉搜索树，画出生成的二叉搜索树的结果。
9. 考虑下列整数列表：[1,2,3,4,5,6,7,8,9,10]，通过一次插入一个整数的方法生成二叉堆，画出生成的二叉堆的结果。
10. 考虑下列整数列表：[10,9,8,7,6,5,4,3,2,1]，通过一次插入一个整数的方法生成二叉堆，画出生成的二叉堆的结果。
11. 考虑我们使用的两种不同的用于实现二叉树的遍历的方式。当我们将它实现为一个方法（method），为什么我们要在调用前序遍历之前检查，然而当我们用函数（function）来实现的时候，可以在函数内部检测？
12. 请说明在构建下列二叉树的过程中必要的函数调用：



13. 给定下面的树，请说明经过怎样合适的旋转后可以让它变得平衡。



14. 将下面的情形作为起始点，推导给出更新的 D 节点平衡因子的方程。



编程练习

1. 扩展 buildParseTree 函数，使它能够解决没有空格分隔的数学表达式。
2. 修改 buildParseTree 函数和 evaluate 函数，用它们解决布尔表达式（与 and，或 or 和非 not）。记住非是一个一元操作符，这会让你的编程稍稍复杂一点。
3. 使用 findSuccessor 方法，写一个二叉搜索树的非递归的前序遍历。
4. 修改二叉搜索树的代码使其成为链式的。写一个链式的二叉搜索树的非递归的前序遍历方法。一个链式的二叉搜索树中，每个节点只有对其后继的引用（即没有 parent 这个参量——译者注）。

5. 修改二叉搜索树代码，使其能够正确解决重复键值问题。也就是说，如果一个键值（key）已经存在于该树当中，新的负载（value）需要取代旧的而不是添加另外一个相同的键值。
6. 实现一个大小有限的二叉堆。也就是说，这个堆仅仅保存 n 个优先级最高的数据项。如果堆的大小即将超过 n，最不重要的节点将被丢弃。
7. 整理 printexp 函数，使其不包括多余的括号。
8. 使用 buildHeap 方法，写一个排序函数，它能在 O(nlogn)时间内将一个列表排序。
9. 写一个函数，它可以为数学表达式构建解析树，并能够计算某些特殊形式的数学表达式的导函数。
10. 实现二叉堆，作为一个最大堆。
11. 使用 BinaryHeap 类，实现一个新的类 PriorityQueue。你的 PriorityQueue 类应该包括构造函数，加上入队 enqueue、出队 dequeue 方法。