树

学习目标

- 理解树的定义及其使用方法
- 使用树来实现映射
- 使用列表来实现树
- 使用类和引用实现树
- 用递归实现树
- 用堆实现优先队列

树的例子

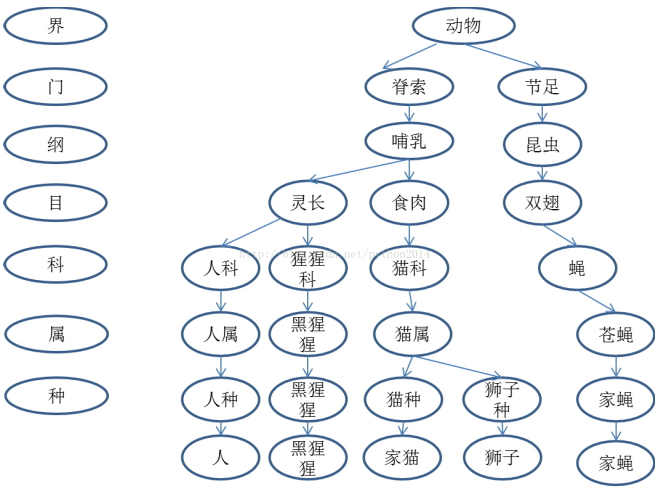
树在计算机科学中应用广泛，如

- 操作系统
- 图形学
- 数据库系统
- 网络

都要用到树。

树和自然界中的树非常相似，也有根、有分枝、有叶子。不同之处是，数据结构的树，根在顶上，而叶子在底部。

例1 生物学上的分级树



由这个例子中可以看到树的一些性质：

- 树是分级的。
  - 通过分级，形成层级结构，越接近上层的越是一般的共性，底部则是一些具体的东西。
  - 顶级的是“界”，第二级是“门”，然后是“纲”，如此等等。
  - 不管到分级树的多少层，所有的生物体都是动物界。

你查以从顶部开始，沿着圆圈和箭头组成的路径一直下到底部。在树的每层上，我们都可以问自己一个问题，而且沿着同意的答案走。例如你可以问：“这个动物是脊索动物还是节足动物？”，如果答案是“脊索动物”，那么就沿着脊索动物向下，“这个脊索动物是哺乳动物吗？”如果不是，我们就被困住了（当然，仅限于图中的资料才会困住），如果是在哺乳动物层次上问，“这个哺乳动物是灵长类还是食肉类？”，这种问题可以一直问下去，直到最底层找到动物的名字。

- 一个节点的所有孩子独立于另一个节点的孩子。
  - 猫属有两个孩子，家猫种和狮子种
  - 蝇属有一个孩子叫家蝇种
  - 但是它的节点不同，与猫属的孩子完全独立。

    这意味着，我们改变蝇属的孩子不会影响到猫属的孩子。

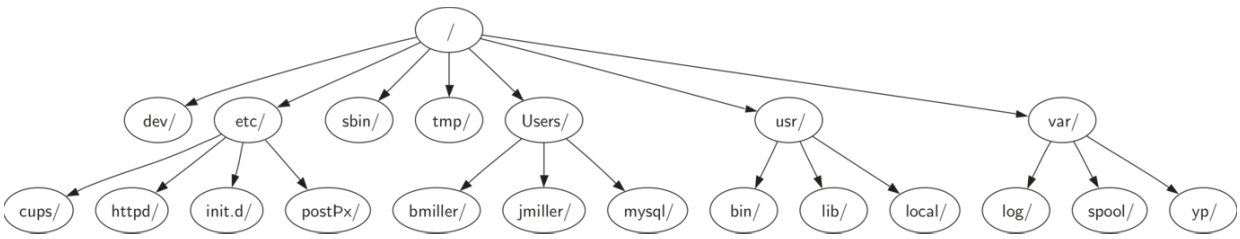
- 每个叶子节点是唯一的。

  我们可以从动物“界”开始，从根到叶沿着规定的路线唯一地定义每一个物种。例如

  动物界→脊索动物门→哺乳动物纲→食肉目→猫科→猫属→猫种→家猫

例2、文件系统。

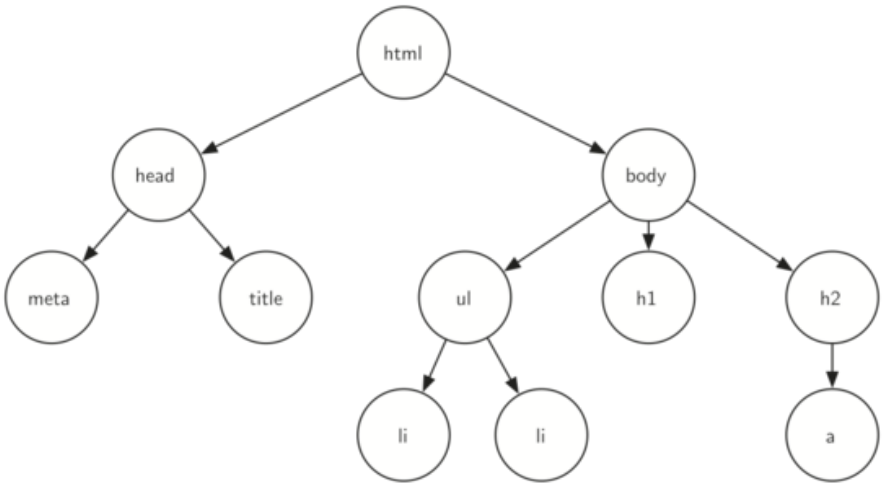
文件系统里中的每个目录或文件夹都是是树形结构。



文件系统与生物层级系统很相似，也可以沿着一条路径从根到任何目录，路径可以唯一地确定一个子目标和他下面的所有文件。

网页

In [ ]:
```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">

<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>

<body>

<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>

<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>

</body>

</html>
```



树的定义和基本概念

节点

节点是树的基本元素。

- 它可以有个名字，叫做“键值”。
- 节点也许有更多的附加信息，称为“负载”。
- 在树的算法中，负载不是核心问题，但在实际问题中用到树的时候，负载经常成为关键元素。

边

边是树的另一个基本元素。

- 一条边把两个节点连起来表示他们之间的关系。
- 每个节点（除根节点外）都有一条从其他节点伸过来的入边连着，而且每个节点可能伸出去好几条出边。

根

根是树结构中，唯一没有入边的节点。

路径

路径就是边连接而成的有序列表，元素是节点。

例如：哺乳纲→ 肉食目 → 猫科 → 猫属 → 家猫种 就是一条路径。

孩子

入边来自同一个节点的节点集合，叫做那个节点的孩子。

父母

一个节点的出边指向其他节点，它就是这些节点的父母。

兄弟

同一父母的节点叫做兄弟。

子树

子树是指一个父母节点和其子孙节点和边组成的集合。

叶子节点

叶子节点指没有孩子的节点。

层次

节点的层次，是指从根到这个节点经过的边的数量。

高度

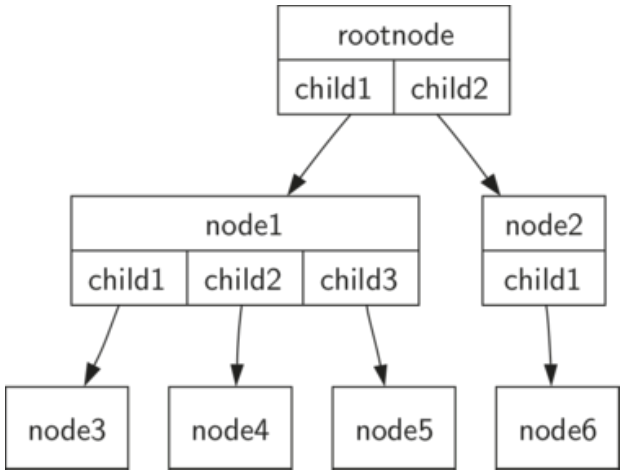
树的高度是指它所有节点层次的最大值。

树的定义(I)

树包括节点的集合以及连接节点的边的集合。树有以下性质：

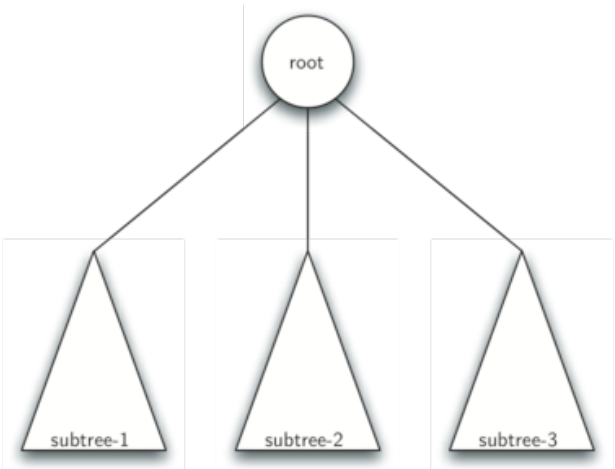
- 一个节点的树称为根节点;
- 对于任意节点 n，除了根节点，都要被另外的唯一节点 p 所连接，此时 p 是 n 的父母;
- 从根到任意节点有一条唯一的路径。

如果每个节点最多有两个孩子，我们把这种树称为二叉树。



树的定义(II)

树或者是一个空集，或者是包括一个根节点和0个或多个子树，每个子树也是树。每个子树的根连到父母树的根。



树的实现

在树的定义的基础上，用以下函数创建并操作二叉树：

- `BinaryTree()` 创建一个二叉树实例
- `getLeftChild()` 返回节点的左孩子
- `getRightChild()` 返回节点的右孩子
- `setRootVal(val)` 把val变量值赋给当前节点
- `getRootVal()` 返回当前节点对象。
- `insertLeft(val)` 创建一个新二叉树作为当前节点的左孩子
- `insertRight(val)` 创建一个新二叉树作为当前节点的右孩子。

实现树的关键点是合适的存储技术。

Python提供两种可用方法：

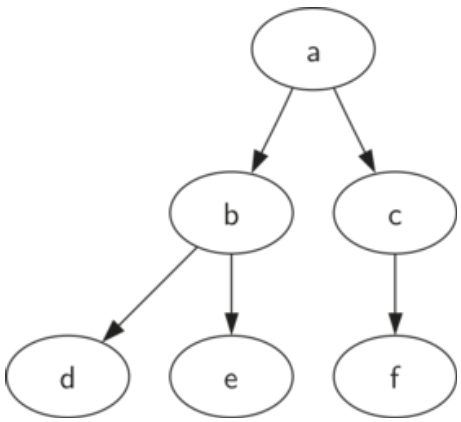
- 列表的列表
- 节点与引用

列表的列表表示法

该方法中，我们使用列表来实现上面的函数，虽然这种方式与之前的抽象数据结构不一样，但它是一个简单的递归结构，可以直接查看和检查。

在列表的列表树中，

- 根节点的数值是列表的第一个元素，
- 第二个元素是它的左子树，
- 第三个元素就是它的右子树。



In [12]:
```
myTree = ['a',            #root
          ['b',           #left subtree
           ['d', [], []],
           ['e', [], []] ],
          ['c',           #right subtree
           ['f', [], []],
           [] ]
         ]
```

- 直接用列表的索引来访问子树。
  - 树根是myTree[0]
  - 左子树是myTree[1]
  - 右子树是myTree[2]。

In [13]:
```
print(myTree)
print('left subtree = ', myTree[1])
print('root = ', myTree[0])
print('right subtree = ', myTree[2])
```

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

下面的代码就是用列表创建了一棵树，完成之后，就可以访问它的根和子树，它的好处在于列表中的“元素列表”就代表了子树，与树有相同的结构，所以它的结构是递归的。如果一个子树有根节点，但是左右子树都是空列表，那么它就是叶子。另一个好处是这种方法产生的树可以推广到“多叉树”而不仅是二叉树，因为另一个子树也不过是一个列表而已。

现在来为这种树型结构定义一些函数以方便使用。

注意：我们没有定义一个二叉树类，这些函数只是帮助操作一个列表，即使工作对象是一个树。

In [14]:
```
def BinaryTree(r):
    return [r, [], []]
```

BinaryTree函数简单地创建了一个列表，其中有一个根节点和两个孩子（为空列表）。

要增加左子树的话，就要插入一个新列表到根列表的第1号位置上。

注意：如果第1号位置上已经保存了数据，需要跟踪这个数据，并把它下压一级，作为新插入列表的左孩子。

In [18]:
```
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，要插入左孩子，先得到原来左孩子位置的列表（可能是空的），当加入新左孩子的时候，把原来的左孩子作为新节点的孩子。这样，我们可以把新节点放在树中任何位置。

插入右孩子的代码也很相似

In [19]:
```
def insertRight(root,newBranch):
    t= root.pop(2)
    if len(t) > 1:
       root.insert(2, [newBranch, [], t])
    else:
       root.insert(2, [newBranch, [], []])
    return root
```

以下为取值函数和赋值函数，针对根节点和左、右子树。

In [22]:
```
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]
```

完整代码

In [29]:
```
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
```

In [30]:
```
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]
```

In [31]:
```
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)))
```

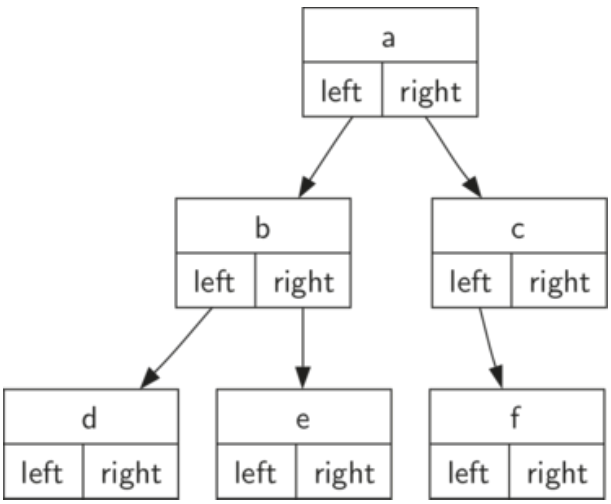
```
[5, [4, [], []], []]
[3, [9, [4, [], []], []], [7, [], [6, [], []]]]
[3, [9, [11, [4, [], []], []], []], [7, [], [6, [], []]]]
[6, [], []]
```

节点与引用方法

第二种表示方法使用节点和引用。

我们将定义一个类，其属性包括根和左右子树。因该方法更加面向对象，后面会着重介绍。

使用节点和引用，树形结构形如下图：



初始化二叉树

- 开始时，只有一个节点的定义和两个子树的引用。
- 左右子树的引用的，也是这个类的实例。

  例如，若要插入一个新的左子树，应该先创建一个新的树对象，并且修改根节点的`self.leftChild`以指向新树。

In [32]:
```
class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```

注意上面代码中，构造函数需要一个对象来保存在根节点。既然列表可以保存任意对象，那么树的根可以是任意对象。在前面的例子中，我们在根节点中保存节点的名字。在图2中那样的树，我们要创建6个`BinaryTree`的对象。

增加左孩子

需要新建一个二叉树对象，并把这个对象赋值给`leftChild`。

In [33]:
```
def insertLeft(self, newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t
```

上面代码中考虑了两种情况：

1. 没有左孩子，只需把节点加到树上。

2. 此节点已经有左孩子，这时要把现存的左孩子下移一级作为插入节点的左孩子。

增加右孩子

插入右孩子也是一样，要么没有右孩子，否则把节点插入到根与现存右孩子之间。

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

获取左、右孩子，设置和获取根结点的值

In [ ]:
```
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
```

测试程序

In [35]:

```
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
```

我们先创建一个简单的树包含一个根和两个节点b， c。下面的代码就是创建树，并为键，左孩子和右孩子赋值。注意左孩子和右孩子和根都是同一个类BinaryTree的不同对象，如同前面我们的递归定义一样，这使得我们可以象处理二叉树一样处理它的子树。

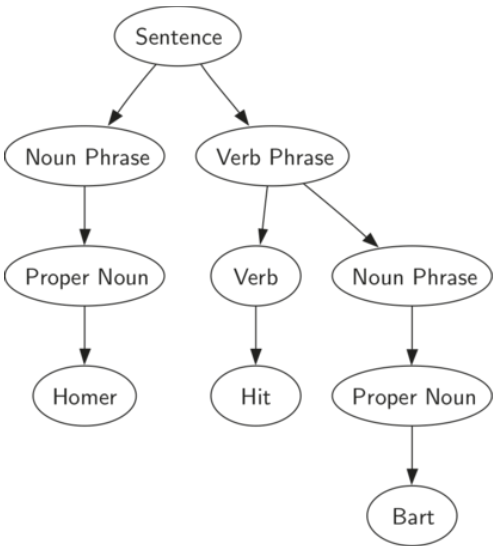
In [36]:
```
r = BinaryTree('a')
print(r.getRootVal())
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

```
a
None
<__main__.BinaryTree instance at 0x7eff2afae1b8>
b
<__main__.BinaryTree instance at 0x7eff2a742878>
c
hello
```

解析树

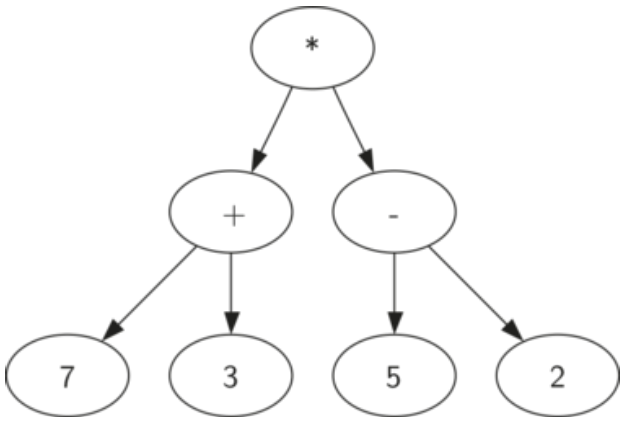
我们已经知道了树的定义及其操作，现在我们来利用树去解决一些实际问题。

首先我们来研究解析树，它常用于真实世界的结构表示，例如句子或数学表达式。



该图显示了一个简单句子的层级结构。

将一个句子表示为一个树，我们就可以利用子树来处理句子中每个独立的结构。



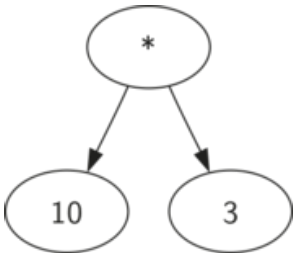
该图将一个`((7+3) * (5-2))`的数学表达式表示成一棵解析树。

树的层级结构可以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`。

利用树的层级结构，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：



接下来我们将更加详细地研究解析树。尤其是：

- 怎样根据一个完全括号数学表达式来建立其对应的解析树
- 怎样计算解析树中数学表达式的值
- 怎样根据一个解析树还原数学表达式

建立解析树的步骤

1. 将表达式字符串分解成字符保存在列表里。这里考虑四种字符：左右括号、操作符和操作数
2. 依次读入每个字符，
   - 读到左括号'('，开始一个新的表达式，创建一个子树来对应这个新的表达式。
   - 读到右括号')'，结束该表达式。
   - 读到操作数，它将成为叶子节点，或其所属操作符的子节点。
   - 读到操作符，它应该有一个左子节点和一个右子节点。

四条规则：

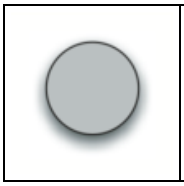
- 读到左括号 '('，添加新节点作为当前节点的左子节点，并将当前节点下降到左子节点处。
- 读到操作符 ['+', '-', '/', '*']，将当前节点的值设置为该操作符。添加一个新的节点作为当前节点的右子节点，并将当前节点下降到右子节点处。
- 读到数字，将当前节点的值设置为该数字，并将当前节点返回到它的父节点。
- 读到右括号 ')' ，返回当前节点的父节点。

考虑表达式 `(3 + (4 * 5))`

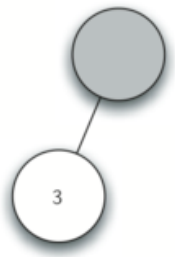
将其分解为字符列表：

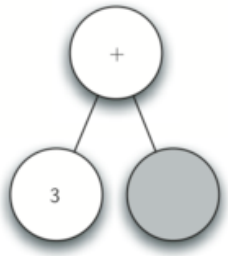
```
['(', '3', '+', '(', '4', '*', '5' ,')', ')']
```

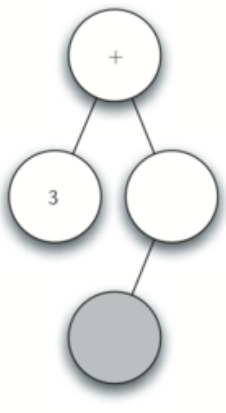
一开始，我们从一个仅包括一个空的根节点的解析树开始。下面的图说明了随着每个新的字符被读入后该解析树的内容和结构。

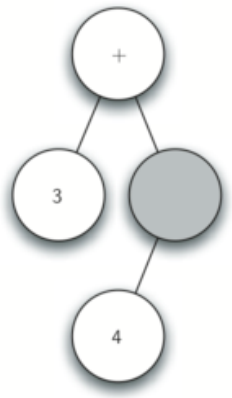
	创建一棵空树

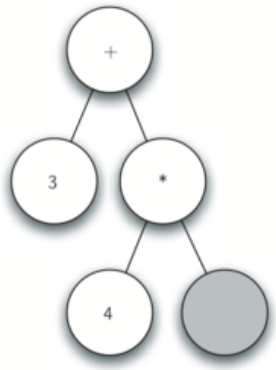
	读如`'('`，创建一个新节点作为当前节点的左子结点，并将当前节点下移到该左子节点。

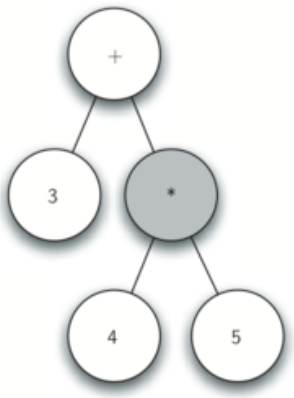
	读入`'3'`，将当前节点的值设置为`'3'`，然后当前结点返回至其父节点。

	读入 `'+'`，将当前节点的值设置为 `'+'`，然后添加一个新节点作为其右子节点，并将当前节点下移到该右子节点。

	读入 `'('`，创建一个新节点作为当前节点的左子结点，并将当前节点下移到该左子节点。

	读入 ``'4'``，将当前节点的值设置为 '4'， 然后返回至其父节点

	读入 ``'*'``，将当前节点的值设置为 ``'*'`` ，然后添加一个新的节点作为其右子节点，并将当前节点下移至该右子节点。

	读入 `'5'`，将当前节点的值设置为 `'5'`，然后返回至其父节点

- 读入 `')'`，将当前节点上移至其父节点。

- 读入 `')'`，将当前节点上移至其父节点。因为当前节点没有父节点，故解析树的构建完成。

通过上述过程可以看出，我们需要跟踪当前节点和其父节点。

- 获得子节点的方法： `getLeftChild` 和 `getRightChild`

- 如何获取父节点呢？

在遍历树的过程中，利用栈来跟踪父节点。

- 当前节点下移到其子节点时，先将当前节点压入栈。
- 想返回当前节点的父节点时，从栈中弹出其父节点。

下面我们将使用栈和二叉树来来创建解析树。

In [15]:
```
class Stack:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[len(self.items)-1]

    def size(self):
        return len(self.items)
```

In [25]:

```
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

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

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

In [26]:

```
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.preorder()  #defined and explained in the next section
```

```
*
+
10
5
3
```

树的遍历

访问树的全部节点，一般有三种模式，这些模式的不同之处，仅在于访问节点的顺序不同。我们把这种对节点的访问称为“遍历”，这三种遍历模式叫做前序、中序和后序。

前序遍历

在前序遍历中，先访问根节点，然后用递归方式前序遍历它的左子树，最后递归方式前序遍历右子树。

中序遍历

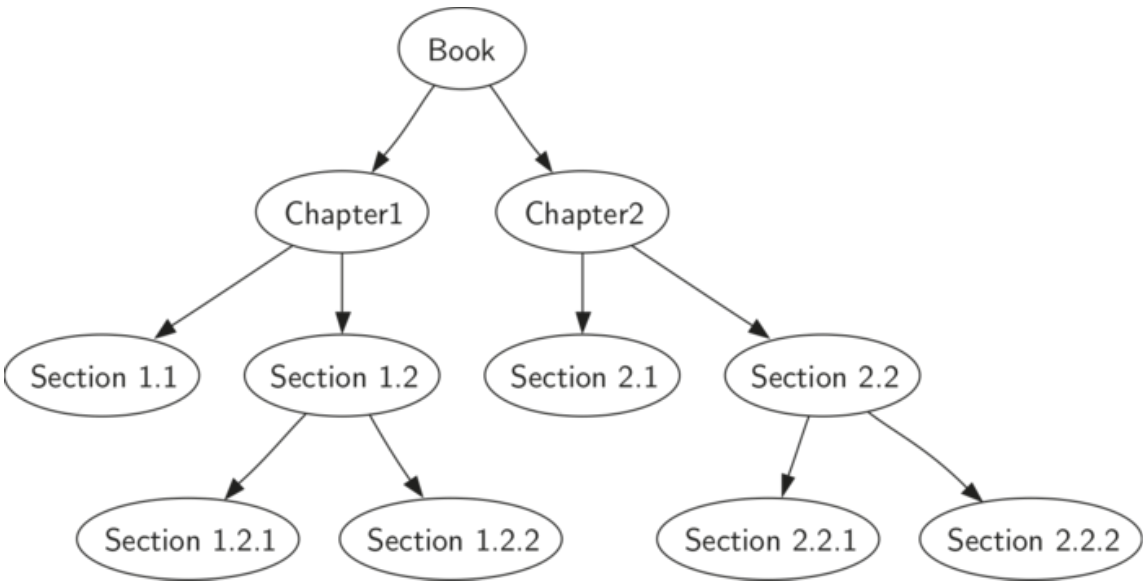
在中序遍历中，先递归中序遍历左子树，然后访问根节点，最后递归中序遍历右子树。

后序遍历

在后序遍历中，先递归后序遍历左子树，然后是右子树，最后是根节点。

前序遍历

象读一本书一样。书本是根节点，每一章是一个孩子，每一节又是本章的孩子，等等。



从头到尾读一本书，正好就是一个前序遍历。

- 从整本书开始，然后递归阅读左孩子（即Chapter 1），然后是读Chapter 1的左孩子(即Section 1.1)。
  - Section 1.1没有孩子，不再递归，此时读完本节的内容
  - 读完Section 1.1，再读Section 1.2，Section 1.2有孩子，要先读Section 1.2.1，再读Section 1.2.2 。Section 1.2读完后，回到第1章，这时再读第2章。

In [3]:
```
# 外部函数
def preorder(tree):
    if tree:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

也可以在 BinaryTree 类里写一个方法：

In [4]:
```
# 内部方法
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.left.preorder()
    if self.rightChild:
        self.right.preorder()
```

外部函数与内部方法，哪种比较好？

`preorder` 作为一个外部函数比较好。

原因是，我们很少是为了遍历而遍历，这个过程中总是要做点事情的，事实上后面我们在后序遍历中举例使用的方法就很象此前我们用过的计算分析树的的代码，所以其余的代码我们都用了外部函数的方法。

后续遍历

与前序遍历很相近，区别只是把打印操作移到了函数的后面。

In [6]:
```
def postorder(tree):
    if tree:
        print(tree.getRootval())
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
```

我们来看一个后续遍历的实际应用，即解析树求值。

具体做法是，计算左右子树的值，然后用操作函数结合在一起。

In [12]:
```
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()
```

中序遍历

先访问左子树，然后访问根，最后访问右子树。

In [13]:
```
def inorder(tree):
    if tree:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())
```

当对一颗解析树做中序遍历时，得到的表达式没有任何括号。现在我们来修改算法，以得到表达式的完全括号表达式。

唯一的改变是：在调用左子树之前，先输出一个左括号，然后在调用右子树之后，输出一个右括号。

In [14]:
```
def printexp(tree):
    sVal = ''
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal
```

优先队列的二叉堆实现

我们已经知道，队列(Queue)是一种“先进先出(FIFO)”的数据结构。队列有一种变体：“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。
- 但在优先队列的内部，元素的次序却由“优先级”来决定：高优先级的元素排在队首，而低优先级的元素则排在后面。
  这使得优先队列的入队(Enqueue)操作变得复杂，需要将元素根据优先级排队。

我们很自然地会想到用排序算法和队列的方法来实现优先队列。

- 在列表里插入一个元素的时间复杂度是$O(n)$，对列表进行排序的时间复杂度是$O(n\log n)$。
- 我们可以用别的方法来降低时间复杂度。一个实现优先队列的经典方法便是采用二叉堆(Binary Heap)。二叉堆能将优先队列的入队和出队复杂度都保持在$O(\log n)$。

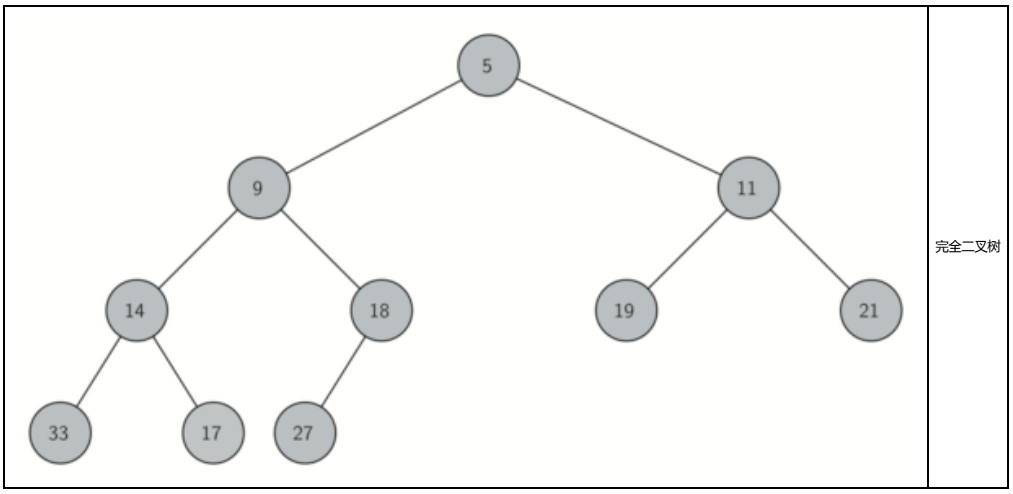
二叉堆是一种特殊的堆，二叉堆是完全二叉树或者是近似完全二叉树。

二叉堆有两种：

- 最大堆：父结点的键值总是大于或等于任何一个子节点的键值；
- 最小堆：父结点的键值总是小于或等于任何一个子节点的键值。

这里我们使用最小堆。

完全二叉树：叶节点只能出现在最下层和次下层，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。



完全二叉树

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。

二叉堆的操作

二叉堆的基本操作定义如下：

- `BinaryHeap()`：创建一个空的二叉堆对象
- `insert(k)`：将新元素加入到堆中
- `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()`：返回堆中节点的个数
- `buildHeap(list)`：从一个包含节点的列表里创建新堆

In [ ]:

```
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())
```

二叉堆结构的性质

为了更好地实现堆，我们采用二叉树。

我们必须始终保持二叉树的“平衡”，就要使操作始终保持在对数数量级上。

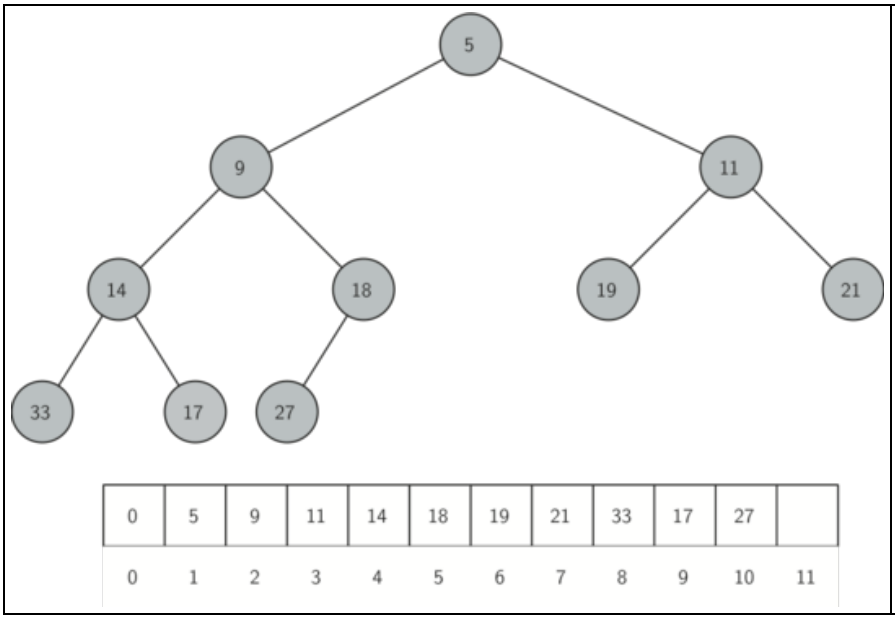
平衡的二叉树根节点的左右子树的子节点个数相同。在堆的实现中，我们采用“完全二叉树”的结构来近似地实现“平衡”。

用单个列表就能实现完全二叉树，而不需要使用“节点+引用”或“嵌套列表”。

因为对于完全二叉树，如果节点在列表中的下标为 `p`，那么其左子节点下标为 `2p`，右节点为 `2p+1`。

当我们要找某个节点的父节点时，可以直接使用 `python` 的整除。如果节点在列表中下标为 `n`，那么父节点下标为 `n//2`.

下图是一个完全二叉树及其列表表示法。注意父节点与子节点之间 `2p` 与 `2p+1` 的关系。完全二叉树的列表表示法结合了完全二叉树的特性，使我们能够使用简单的数学方法高效地遍历一棵完全二叉树，也使我们能高效实现二叉堆。



完全二叉树及其列表表示法

堆次序的性质

堆中存储元素的方法依赖于堆的次序。所谓堆次序，是指堆中任何一个节点 x，其父节点 p 的键值均小于或等于 x 的键值。上图是一颗满足堆次序性质的完全二叉树。

二叉堆操作的实现

__init__函数

因可用一个列表来保存堆的数据，故构造函数只需要初始化一个列表和一个 `currentSize` 来表示堆当前的大小。

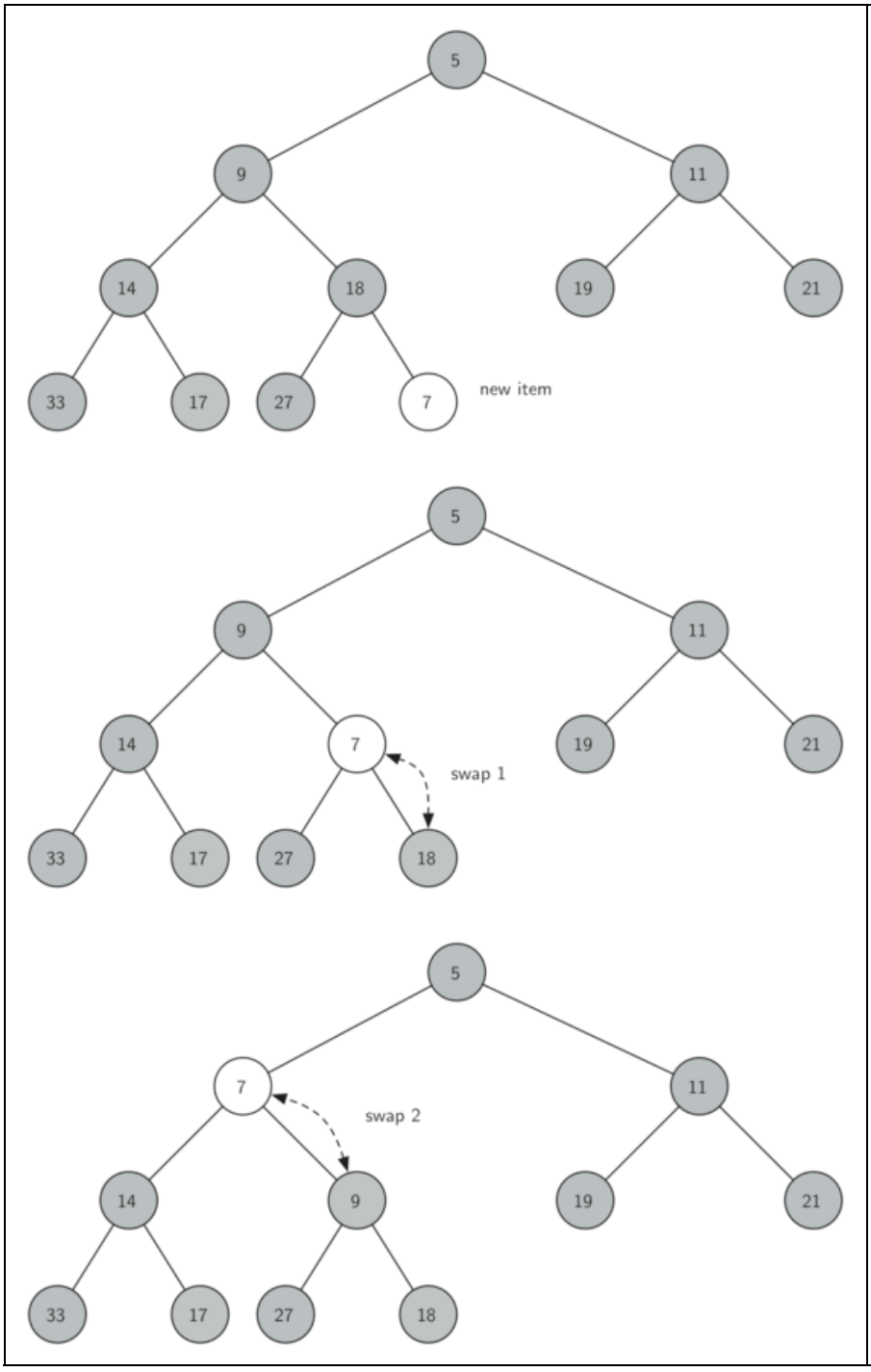
In [37]:
```
class BinHeap:
    def __init__(self):
        self.heapList = [0] #没有用到，但为了后面的代码可以方便地使用整除，我们仍然保留它。
        self.currentSize = 0
```

insert 方法

为了满足"完全二叉树"的性质，新键值应该添加到列表的末尾。但如果新键值简单地添加在列表末尾，将无法满足堆次序。

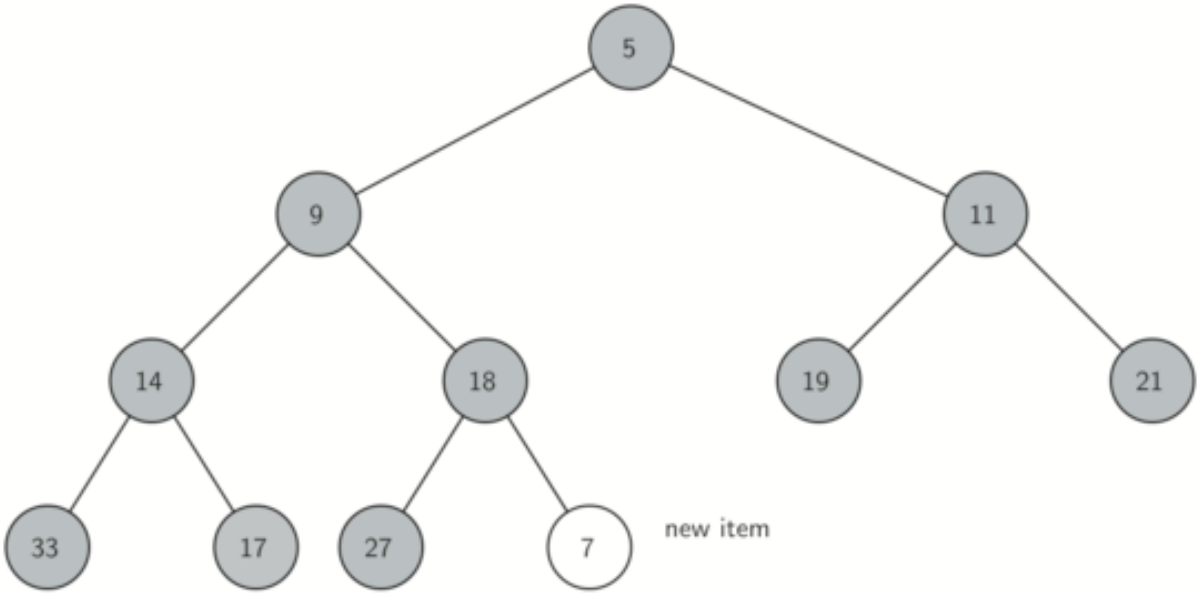
可比较父节点和新加入元素来重新满足堆次序。

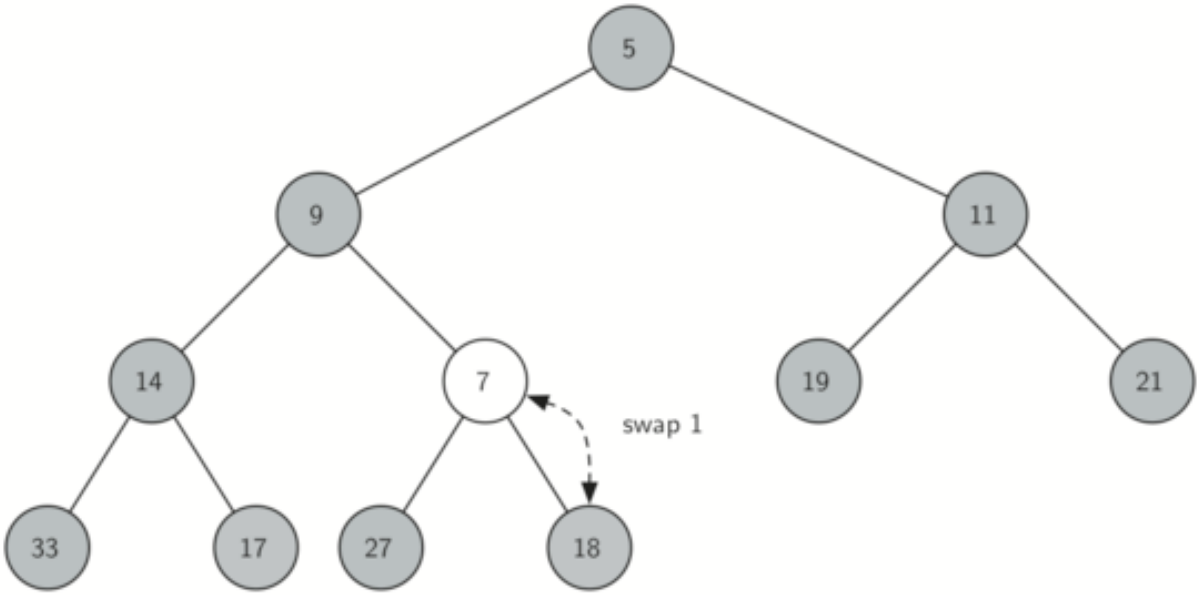
- 如果新加入元素比父节点要小，与父节点互换位置。

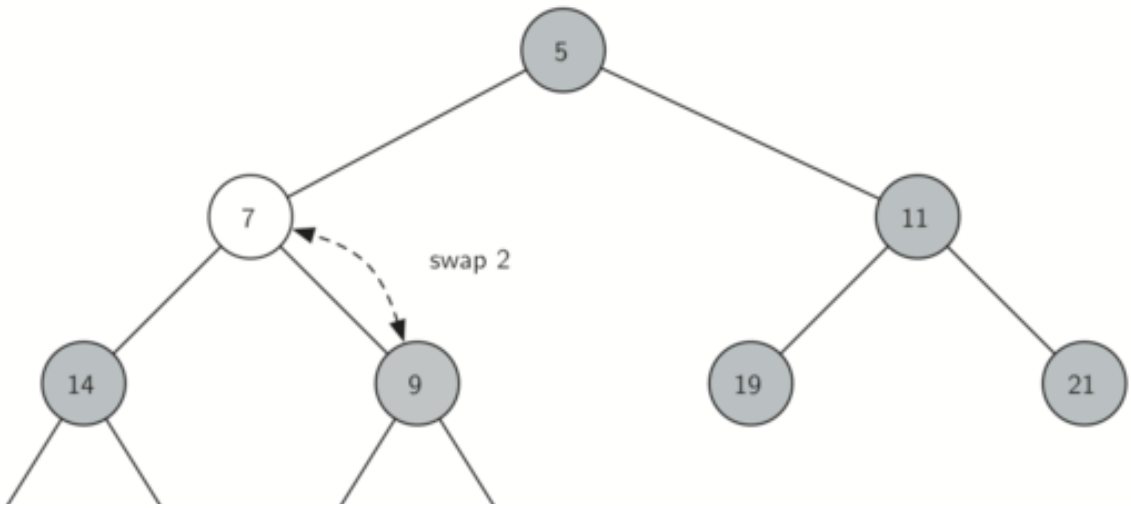


通过一系列交换操作，使新加入元素“上浮”到正确的位置。

让一个元素“上浮”时，必须保证新节点与父节点以及其他兄弟节点之间的堆次序。


5
9
11
14
18
19
21
33
17
27
7
new item


5
9
11
14
7
19
21
swap 1
33
17
27
18


5
7
11
swap 2
14
9
19
21

- 如果新节点非常小，仍然需要将它交换到其他层。事实上，我们需要不断交换，直到到达树的顶端。

Listing 2 所示的是“上浮”方法，它把一个新节点“上浮”到其正确位置来满足堆次序。这里很好地体现了我们之前在headlist中没有用到的元素 0 的重要性。这样只需要做简单的整除，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。

In [40]:
```
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
```

In [32]:
```
def insert(self, k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

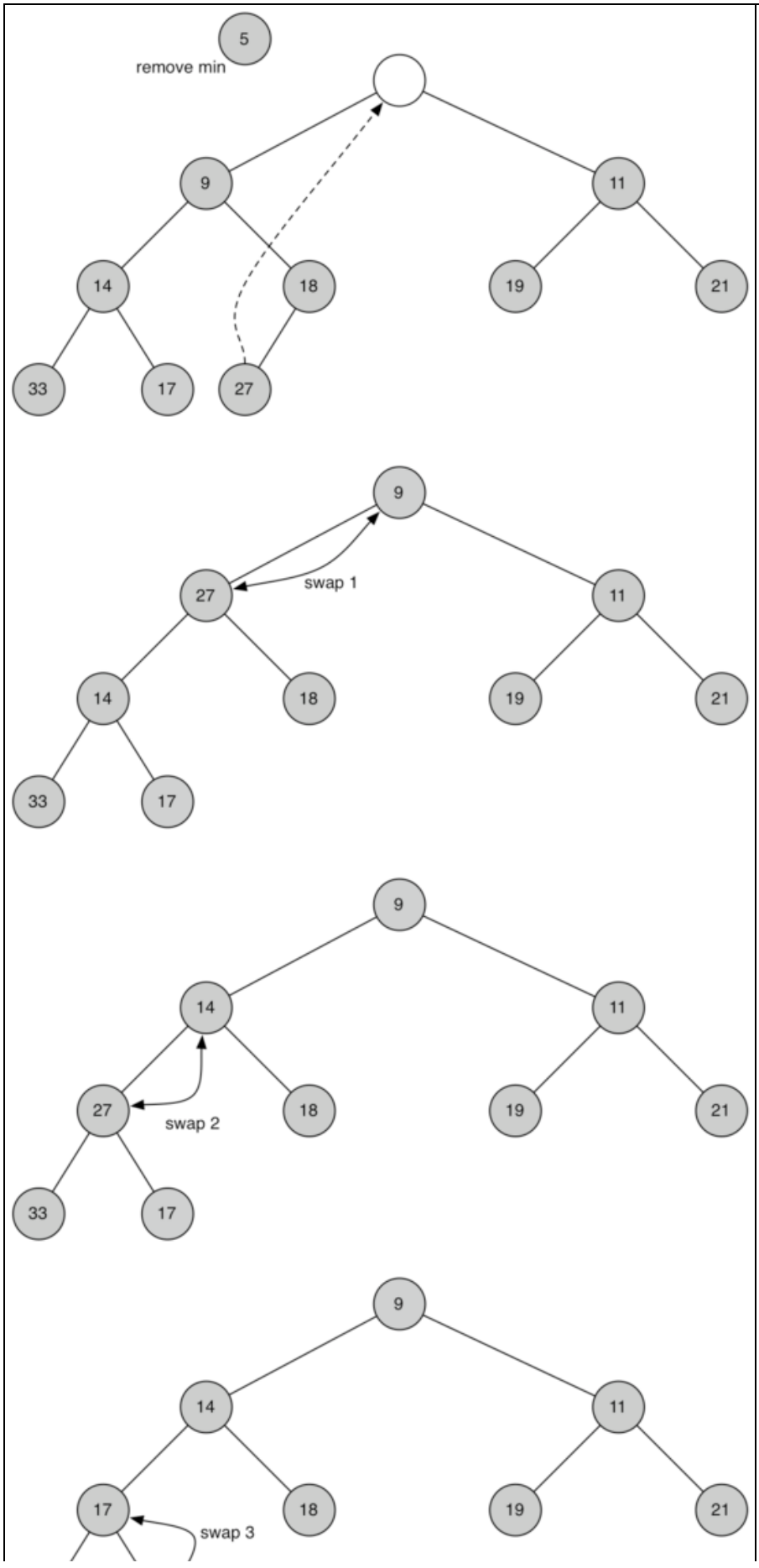
delMin 方法

堆次序要求根节点是树中最小的元素，因此找到最小项非常容易。而删除根节点的元素后，如何保持堆结构和堆次序则不那么容易。

可以分两步走：

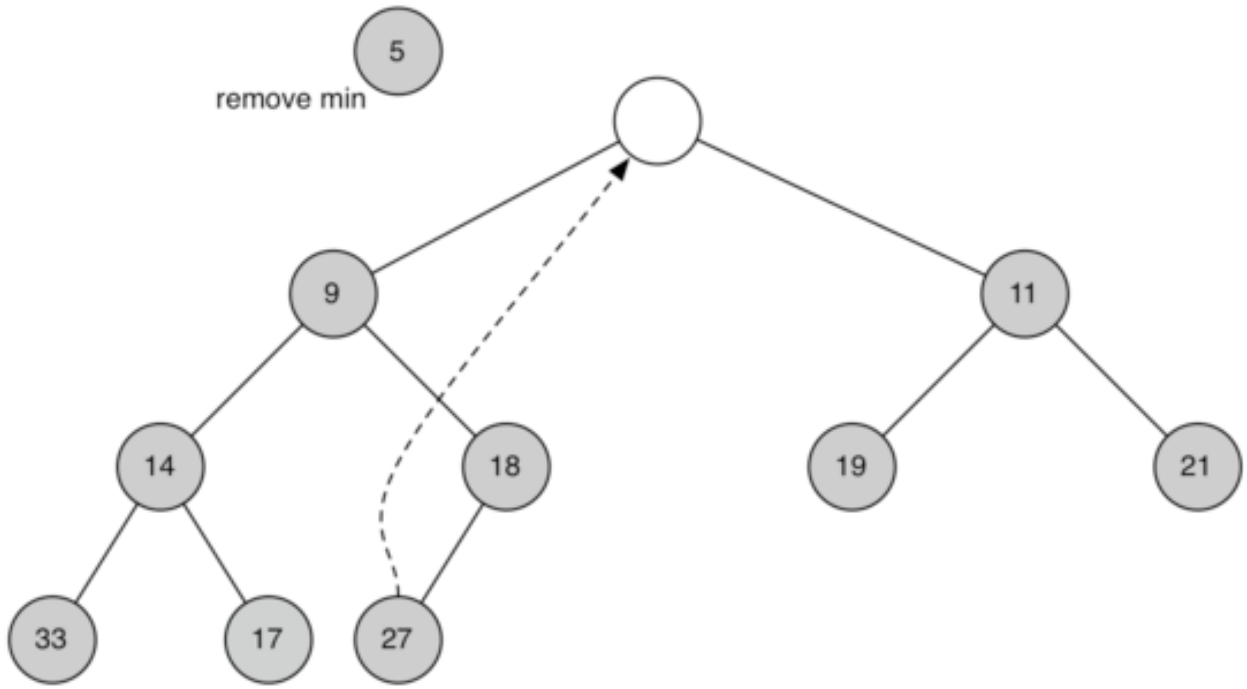
1. 用最后一个节点来代替根节点。移走最后一个节点保持了堆结构的性质。这么简单的替换，还是会破坏堆次序。

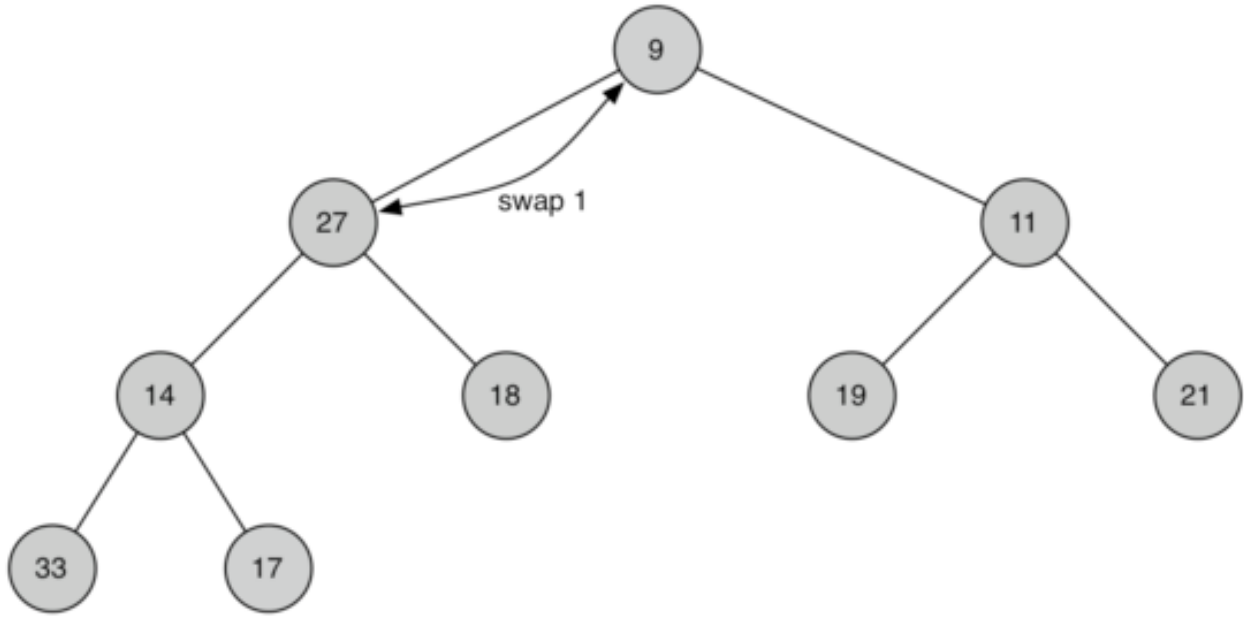
2. 将新节点“下沉”来恢复堆次序。

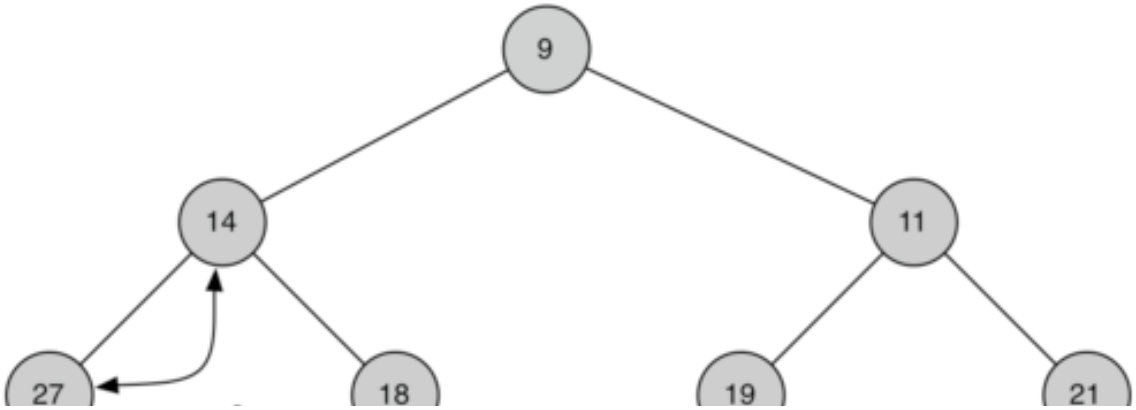


通过一系列交换操作，使新节点“下沉”到正确的位置。

让一个元素“上浮”时，必须保证新节点与父节点以及其他兄弟节点之间的堆次序。


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
18
19
21

为了保持堆次序，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与之交换。

In [42]:
```
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
```

In [ ]:
```
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
```

In [34]:
```
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
```

buildHeap 方法

如何从一个无序列表生成一个"堆"呢？

1. 我们首先想到的可能是将无序列表中的每个元素依次插入到堆中。

   - 对于一个排好序的列表，可用折半查找找到合适的位置，然后在下一个位置将这个键值插入到堆中，时间复杂度为$O(\log n)$。
   - 将一个元素插入到列表，需要移动列表的一些元素，为新节点腾出位置，时间复杂度为O(n)。因此用 `insert` 方法的总开销是$O(n\log n)$。

2. 然而，我们能直接将整个列表生成堆，将总开销控制在$O(n)$。

In [65]:
```
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
```

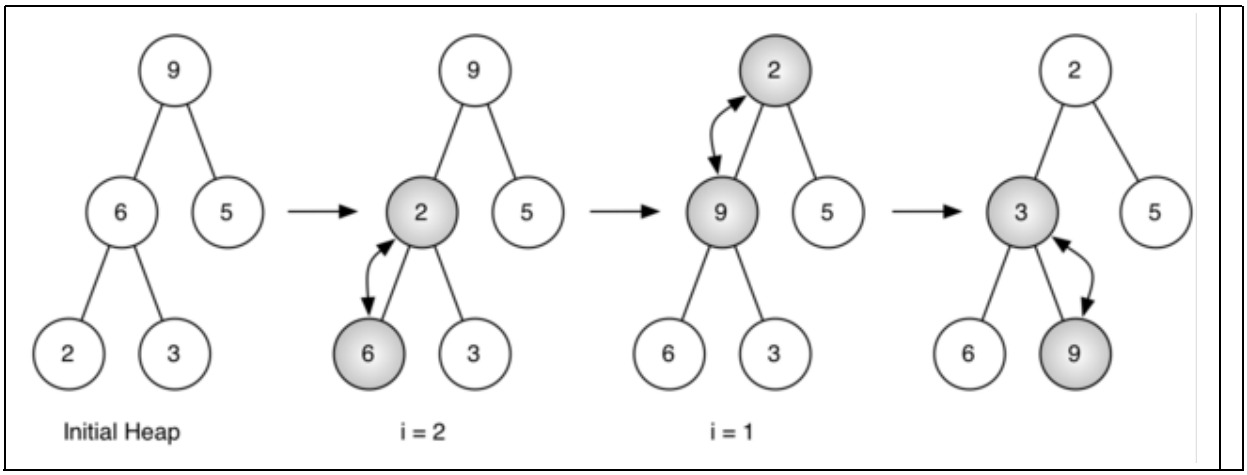


图 5 所示的是利用buildHeap方法将最开始的树[9, 6, 5, 2, 3]中的节点移动到正确的位置时所做的交换操作。尽管我们从树中间开始，然后回溯到根节点，但percDown方法保证了最大子节点总是"下沉"。因为堆是完全二叉树，任何在中间的节点都是叶节点，因此没有子节点。注意，当i=1时，我们从根节点开始下沉，这就需要进行大量的交换操作。可以看到，图 5 最右边的两颗树，首先 9 从根节点的位置移走，移到下一层级之后，percDown进一步检查它此时的子节点，保证它下降到不能再下降为止，即下降到正确的位置。然后进行第二次交换，9 和 3 的交换。由于 9 已经移到了树最底层的层级，便无法进一步交换了。比较一下列表表示法和图 5 所示的树表示法进行的一系列交换还是很有帮助的。

In [49]:
```
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)
```

In [ ]:
```
    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
```

In [ ]:
```
    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
```

In [50]:

```
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
```

In [51]:

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

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

```
2
3
5
6
9
```

二叉查找树

二叉查找树是一棵二叉树，可以是空树，否则满足如下性质：

- 树中结点有一个唯一的关键字；
- 如果有左子树，则左子树的所有关键字小于根的关键字；
- 如果有右子树，则右子树的所有关键字大于根的关键字；
- 左、右子树都是二叉查找树。

对于二叉查找树的每个节点 `node`，它的左子树中所有的关键字都小于 `node` 的关键字，而右子树的所有关键字都大于 `node` 的关键字。

二叉查找树特别适合同时需要查找、插入和删除三种操作的应用。

二叉查找树可以按关键字值操作，也可以按关键字序操作。

如：可在查找树中查找或删除关键字为key的结点，也可以删除关键字排第5小的结点；还可以插入一个结点，同时获得该结点关键字的大小位置。

初始化

In [63]:
```
class BinarySearchTree(object):
    def __init__(self, key):
        self.key = key
        self.leftChild = None
        self.rightChild= None
```

find 方法

假设要查找关键字是 `key` 的结点。

从根开始，若根是 `None`，树中无结点，查找失败；否则，比较 `key` 与根的关键字：

- 如果 `key` 等于根的关键字，查找成功，结束；
- 如果 `key` 小于根的关键字，因右子树的所有关键字不可能等于 `key`，故应查找根的左子树；
- 如果 `key` 大于根的关键字，应查找根的右子树。

In [64]:
```
def find(self, key):
    if key == self.key:
        return self
    elif key < self.key and self.leftChild:
        return self.leftChild.find(key)
    elif key > self.key and self.rightChild:
        return self.rightChild.find(key)
    else:
        return None
```

findMin 和 findMax 方法

分别返回树中的最小元素和最大元素的位置。

- `findMin`, 从根开始，并且只要有左孩子就向左查找，终止点是最小元素。
- `findMax`，从根开始，并且只要有右孩子就向右查找，终止点是最大元素。

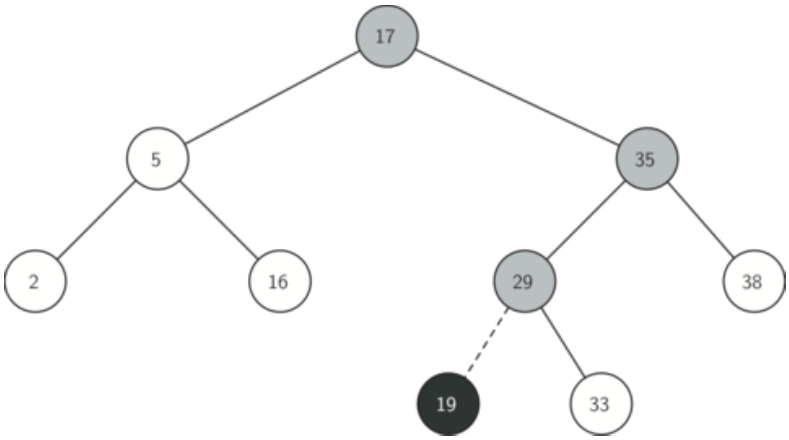
In [54]:
```
def findMin(self):
    if self.leftChild:
        return self.leftChild.findMin()
    else:
        return self
```

In [62]:
```
def findMax(self):
    current = self
    if tree:
        while tree.rightChild:
            tree = tree.rightChild
    return tree
```

insert 方法

往二叉查找树中插入关键字为 `key` 的结点。

首先应该检查该节点是否已经出现在树中，故应先执行 `find` 操作，如果找到 `key`，则什么也不做；否则，将 `key` 插入到遍历路径的最后一点。



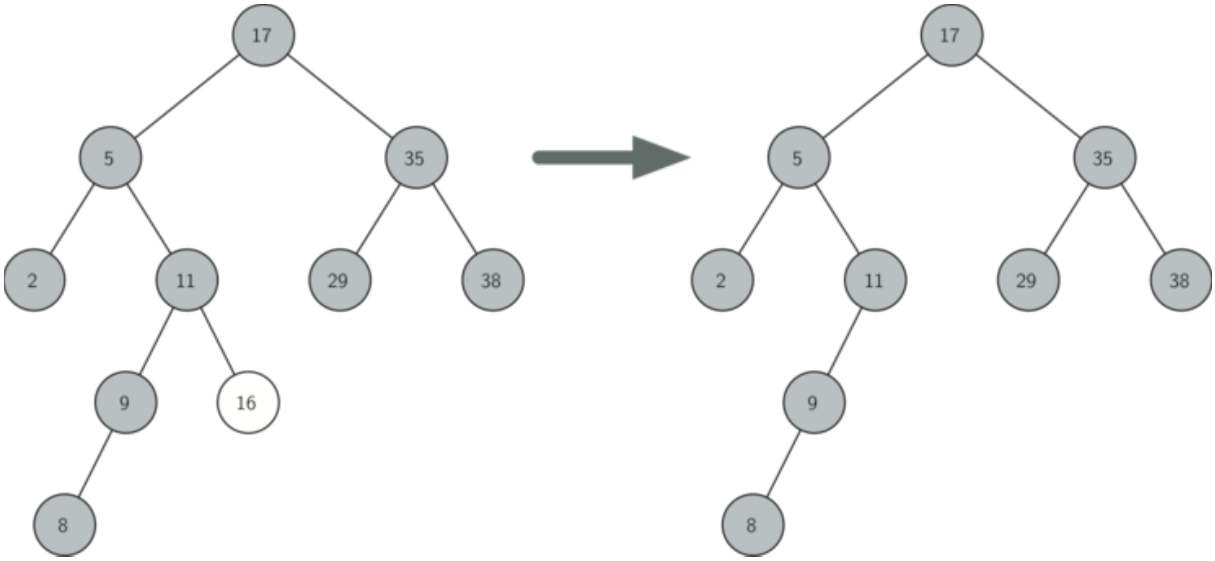
In [58]:

```
def insert(self, key):
    if key < self.key:
        if self.leftChild:
            self.leftChild.insert(key)
        else:
            tree = BinarySearchTree(key)
            self.leftChild = tree
    elif key > self.key:
        if self.rightChild:
            self.rightChild.insert(key)
        else:
            tree = BinarySearchTree(key)
            self.rightChild = tree
```

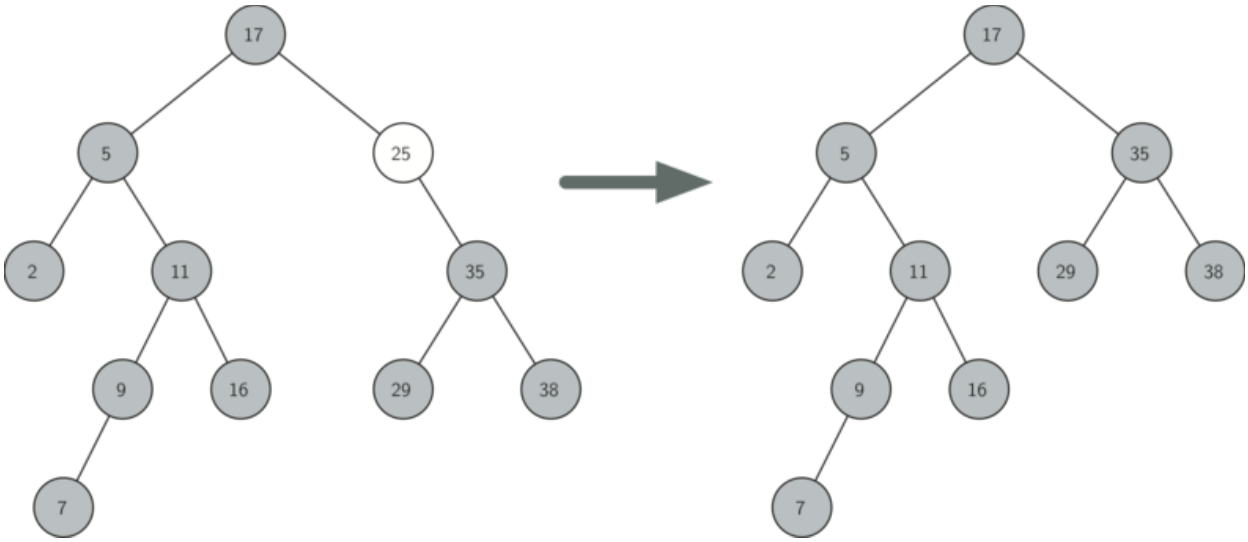
delete 方法

删除某节点有三种情况：

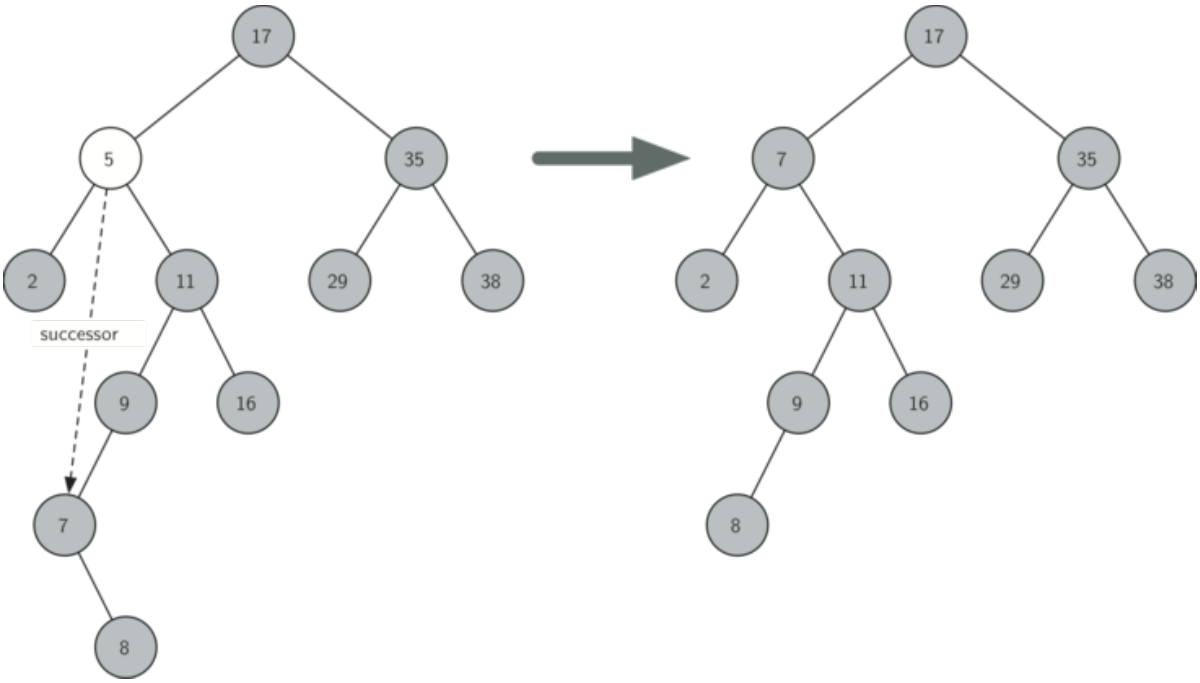
- 若节点是叶子，则可直接删除。



- 若节点只有一个孩子，则把它唯一的孩子放在删除结点原来的位置即可。



- 如果节点有两个孩子，则将其右子树的最小数据代替此节点的数据，并将其右子树的最小数据（不可能有左孩子，只有一个右孩子）删除。



In [61]:

```
def delete(self, key):
    if self.find(key):
        if key < self.key:
            self.leftChild = self.leftChild.delete(key)
            return self
        elif key > self.key:
            self.rightChild = self.rightChild.delete(key)
            return self
        elif self.leftChild and self.rightChild:  # 有两个孩子
            tmp = self.rightChild.findMin().key
            self.key = tmp
            self.rightChild = self.rightChild.delete(tmp)
            return self
        else:
            if self.leftChild:
                return self.leftChild
            else:
                return self.rightChild
    else:
        return self
```