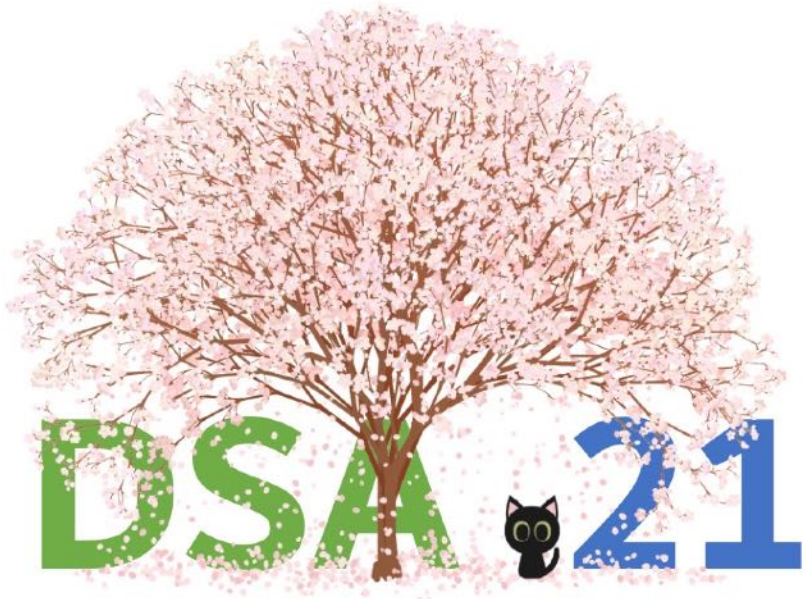
数据结构与算法（Python）-09/第10周

北京大学 陈斌

2021.05.11

线下课堂

- 本周内容小结：树及算法（上）
- 问题解答
- 【K09】课堂练习

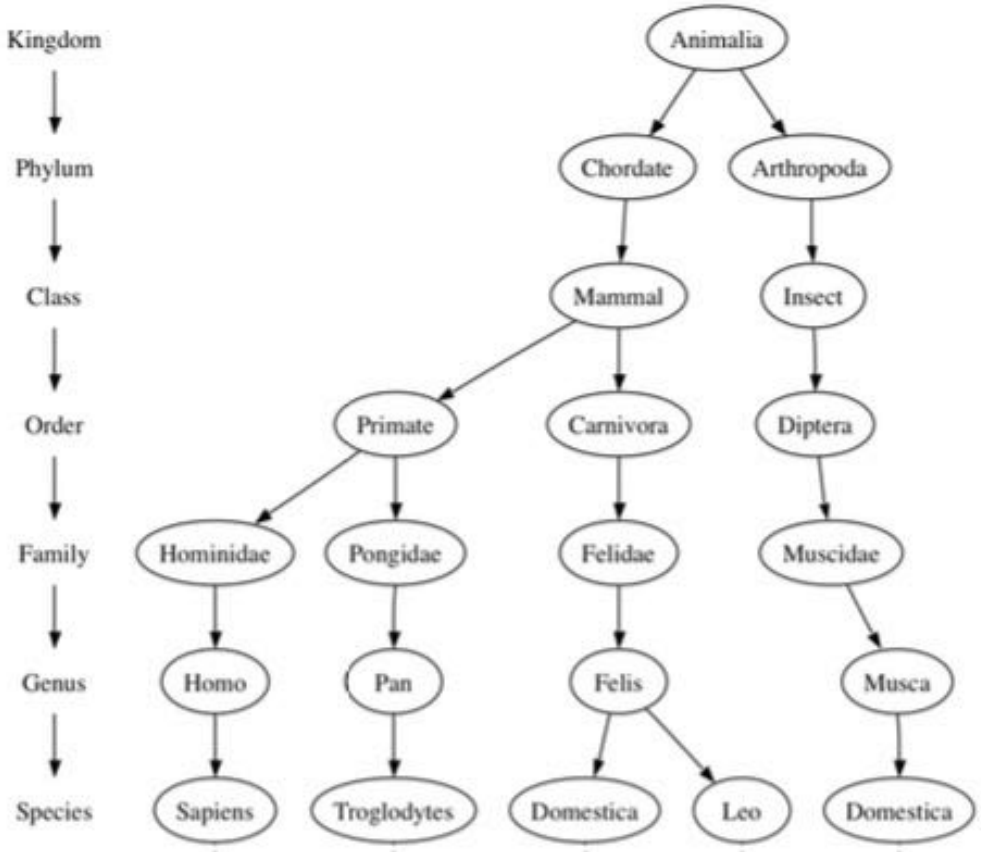


W09-树及算法（上）

- 601 什么是树 8m37s
- 602 树结构相关术语 8m23s
- 603 树的嵌套列表实现 11m00s
- 604 树的链表实现 6m57s
- 605 树的应用：表达式解析（上） 13m03s
- 606 树的应用：表达式解析（下） 15m15s
- 607 树的遍历 10m11s
- 608 优先队列和二叉堆 11m45s
- 609 二叉堆的Python实现 13m14s

601 什么是树

- **线性结构**
  每个数据项都有唯一前驱和唯一后继
  第一个没有前驱（首）
  最后一个没有后继（尾）
- **树结构**
  每个数据项都有唯一前驱和若干后继
  恰有一个没有前驱（树根）
  以及若干个没有后继（树叶）
- **层次化的结构**
- **子节点相互隔离**
- **根到叶的路径唯一**



602 树结构相关术语

术语集合

节点Node，边Edge

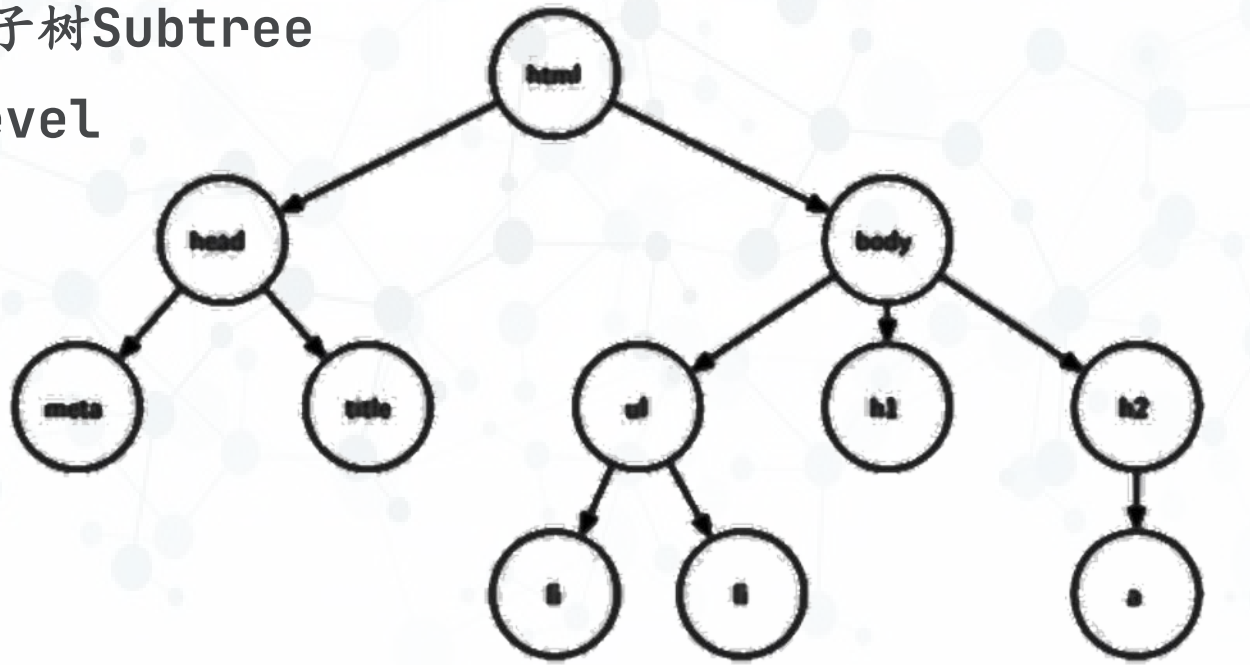
树根Root，路径Path

子节点Children，父节点Parent

兄弟节点Sibling，子树Subtree

叶节点Leaf，层级Level

高度Height

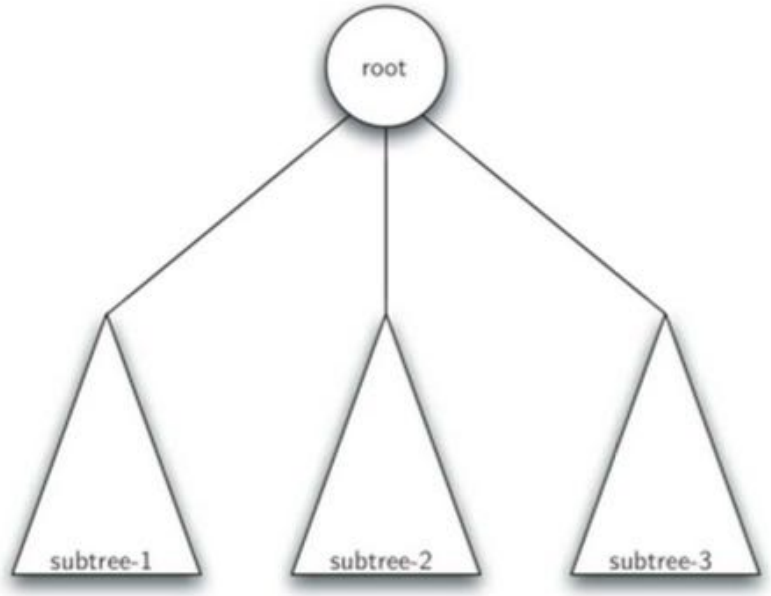


树的定义2（递归定义）

- **树是：**

  空集；

  或者由根节点及0或多个子树构成（其中子树也是树），每个子树的根到根节点具有边相连。



603 树的嵌套列表实现

❖ 递归的嵌套列表实现二叉树，由具有3个元素的列表实现：

第1个元素为根节点的值；

第2个元素是左子树（所以也是一个列表）；

第3个元素是右子树（所以也是一个列表）。

[root, left, right]

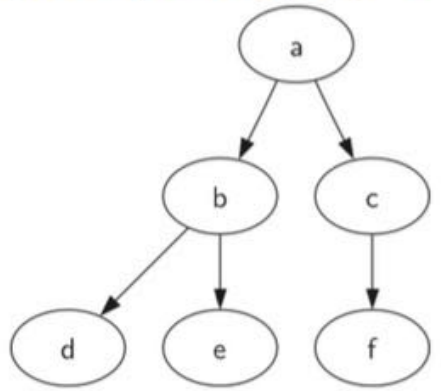
```
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

❖ 嵌套列表法的优点

子树的结构与树相同，是一种递归数据结构

很容易扩展到多叉树，仅需要增加列表元素即可

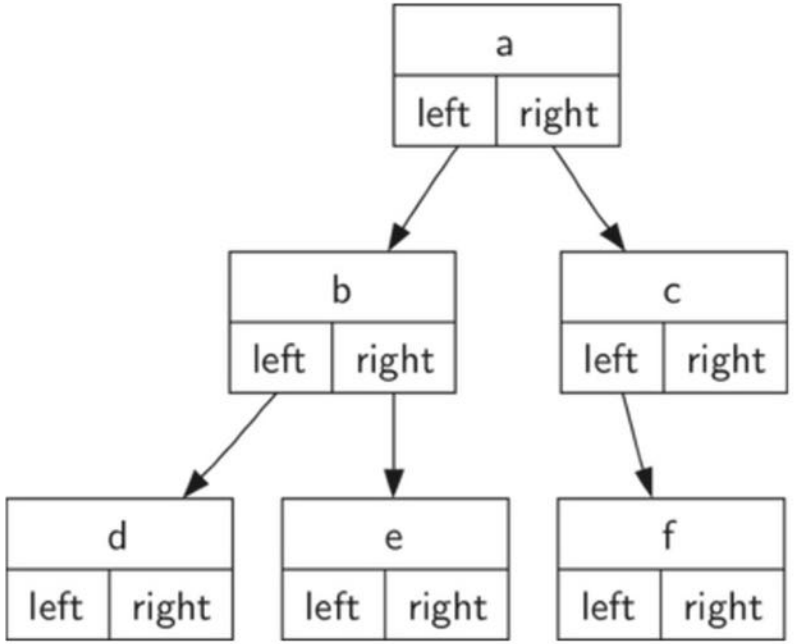
```
1  myTree = ['a', # 树根
2            ['b', # 左子树
3             ['d', [], []],
4             ['e', [], []] ],
5            ['c', # 右子树
6             ['f', [], []],
7             [] ]
8            ]
```



604 树的链表实现

递归数据结构

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

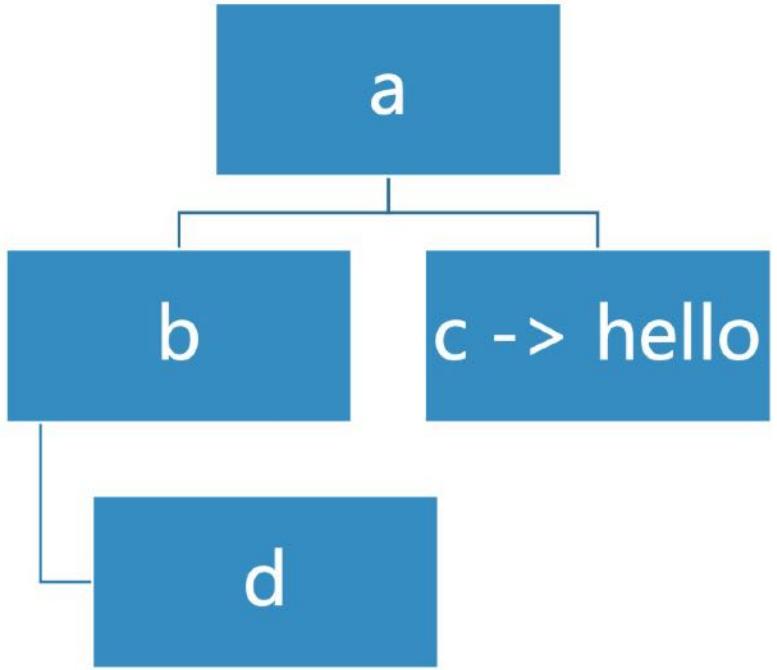



```
def insertLeft(self,newNode):
    if self.leftChild == None:
        self.leftChild = BinaryTree(newNode)
    else:
        t = BinaryTree(newNode)
        t.leftChild = self.leftChild
        self.leftChild = t

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

❖请画出r的图示

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



605/6 树的应用：表达式解析

- 树中每个子树都表示一个子表达式

  将子树替换为子表达式值的节点，即可实现求值

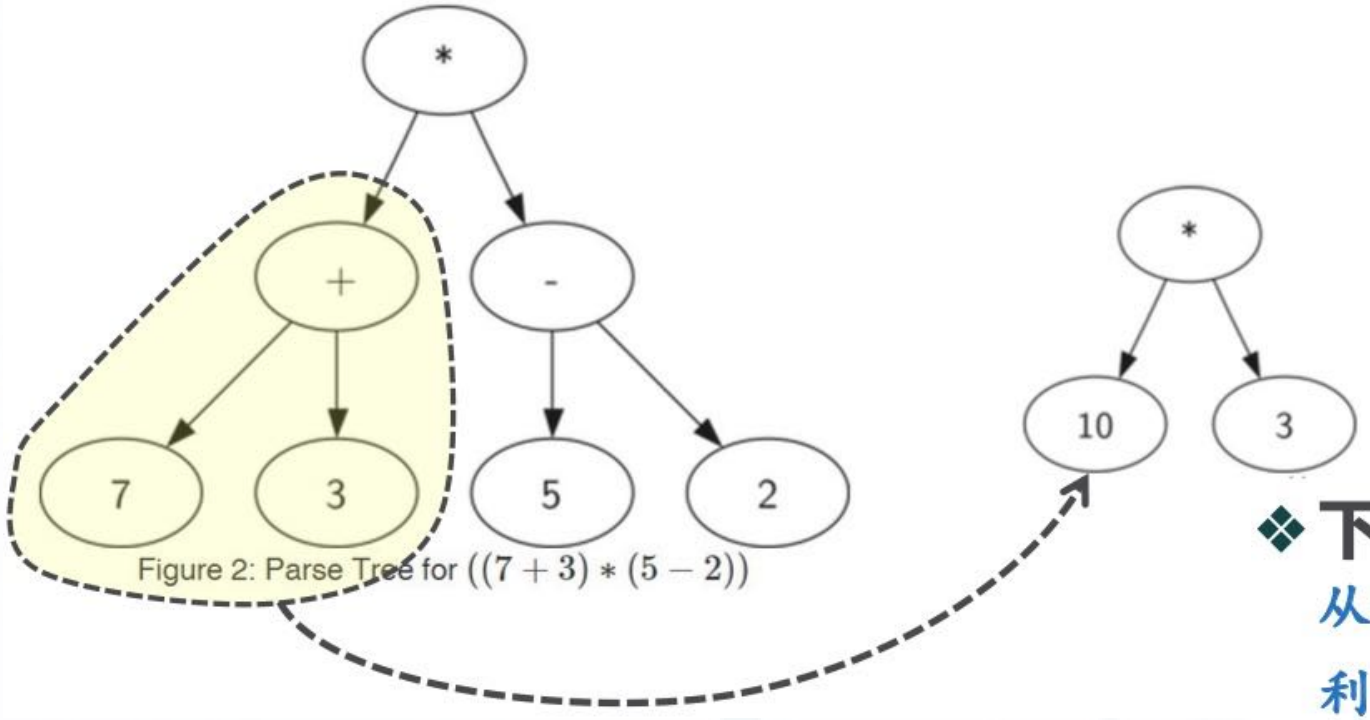


Figure 2: Parse Tree for $((7+3)*(5-2))$

- 下面，我们用树结构来做如下尝试

  从全括号表达式构建表达式解析树

  利用表达式解析树对表达式求值

  从表达式解析树恢复原表达式的字符串形式

从全括号表达式建立解析树

(3+(4*5))

（表达式开始/ ）表达式结束/ 数：叶节点/ 算符：子树根

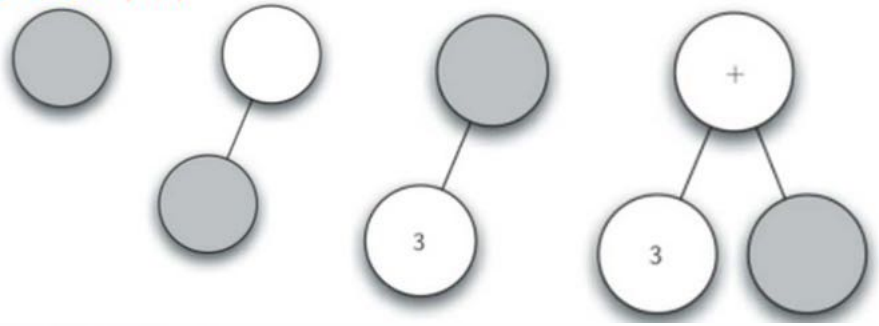
['(', '3', '+', '(', '4', '*', '5', ')', ')']

创建空树，当前节点为根节点

读入'('，创建了左子节点，当前节点下降

读入'3'，当前节点设置为3，上升到父节点

读入'+'，当前节点设置为+，创建右子节点，当前节点下降

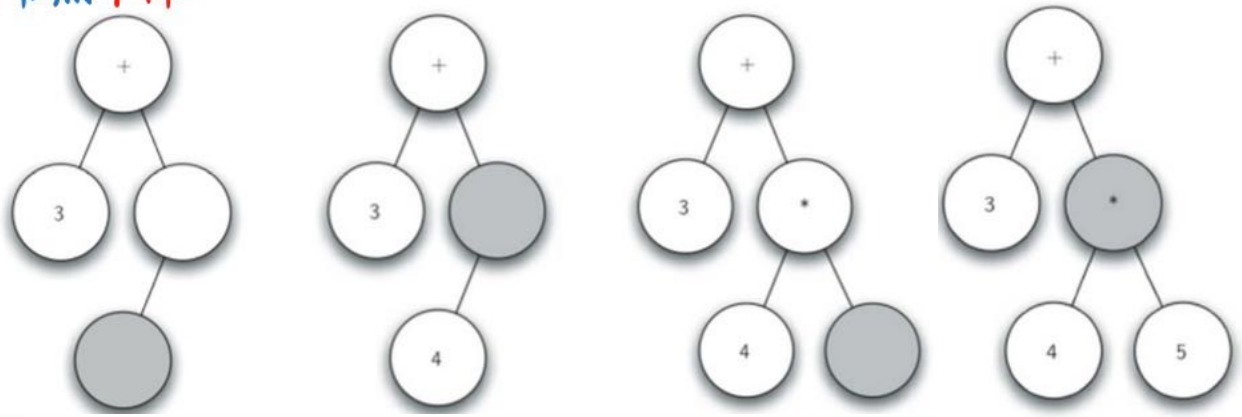


读入'('，创建左子节点，当前节点下降

读入'4'，当前节点设置为4，上升到父节点

读入'*'，当前节点设置为*，创建右子节点，当前节点下降

不用栈，
可以上升到父节点么？



表达式解析树求值：递归算法

- **求值函数evaluate的递归三要素：**

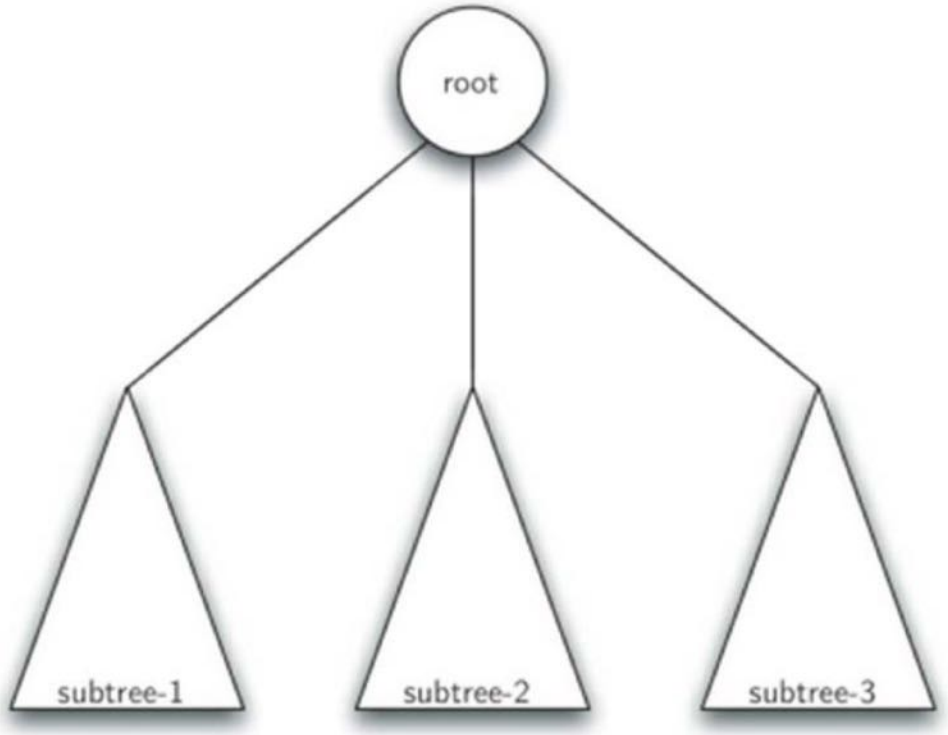
  基本结束条件：叶节点是最简单的子树，没有左右子节点，其根节点的数据项即为子表达式树的值

  缩小规模：将表达式树分为左子树、右子树，即为缩小规模

  调用自身：分别调用evaluate计算左子树和右子树的值，然后将左右子树的值依根节点的操作符进行计算，从而得到表达式的值

607 树的遍历

- 访问树中的每个节点
- 作为递归数据结构，可以有什么样的次序？



二叉树的遍历

- 我们按照对节点访问次序的不同来区分3种遍历

前序遍历（preorder）：先访问根节点，再递归地前序访问左子树、最后前序访问右子树；

中序遍历（inorder）：先递归地中序访问左子树，再访问根节点，最后中序访问右子树；

后序遍历（postorder）：先递归地后序访问左子树，再后序访问右子树，最后访问根节点。

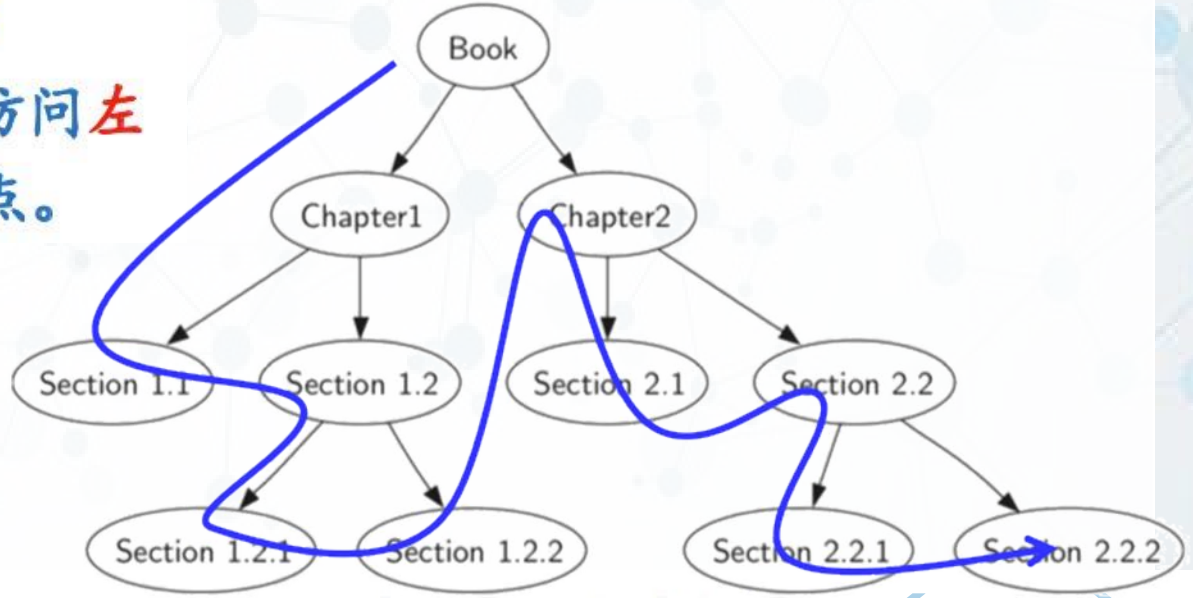


Figure 5: Representing a Book as a Tree

后序遍历

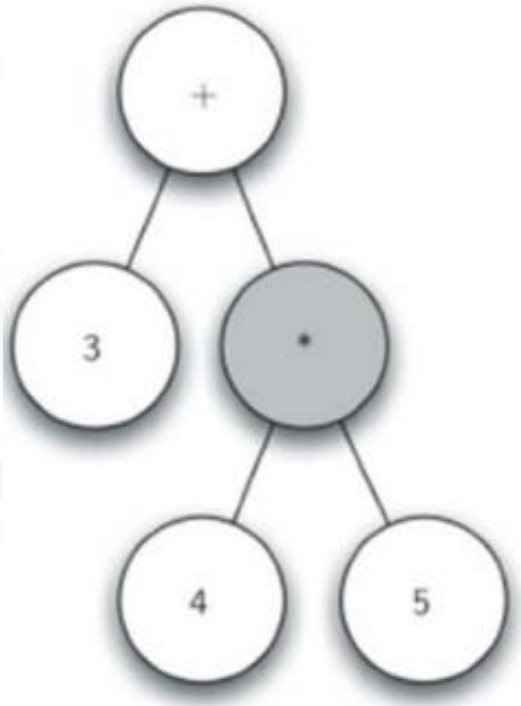
❖ 采用后序遍历法重写表达式求值代码：

```
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()
```

左子树

右子树

根节点



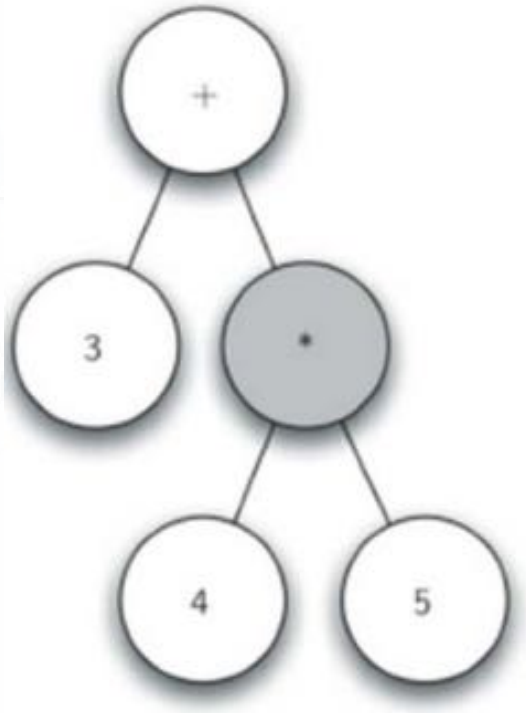
中序遍历

- 采用中序遍历递归算法来生成全括号中缀表达式

  下列代码中对每个数字也加了括号，请自行修改代码去除（课后练习）

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

((3)+((4)*(5)))



608 优先队列和二叉堆

- 优先队列：key最小的最先出队
- 思考：有什么方案可以用来实现优先队列？

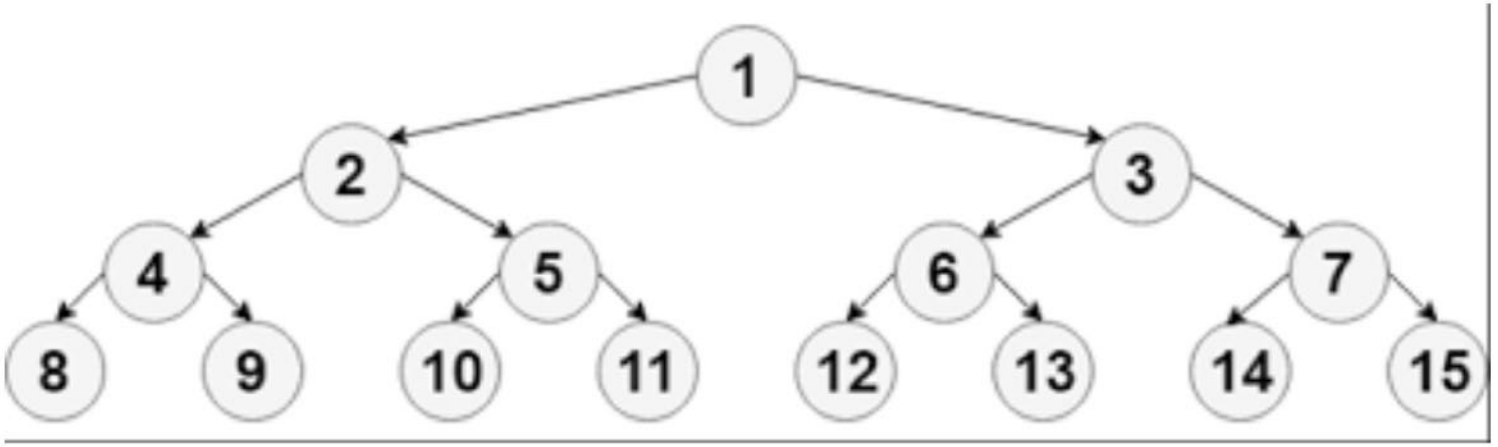
  出队和入队的复杂度大概是多少？
- 实现优先队列的经典方案是采用二叉堆数据结构

  二叉堆能够将优先队列的入队和出队复杂度都保持在O(log n)

一种性能很好的实现

- 为了使堆操作能保持在对数水平上，就必须采用二叉树结构；
- 同样，如果要使操作**始终**保持在对数数量级上，就必须始终保持二叉树的“平衡”

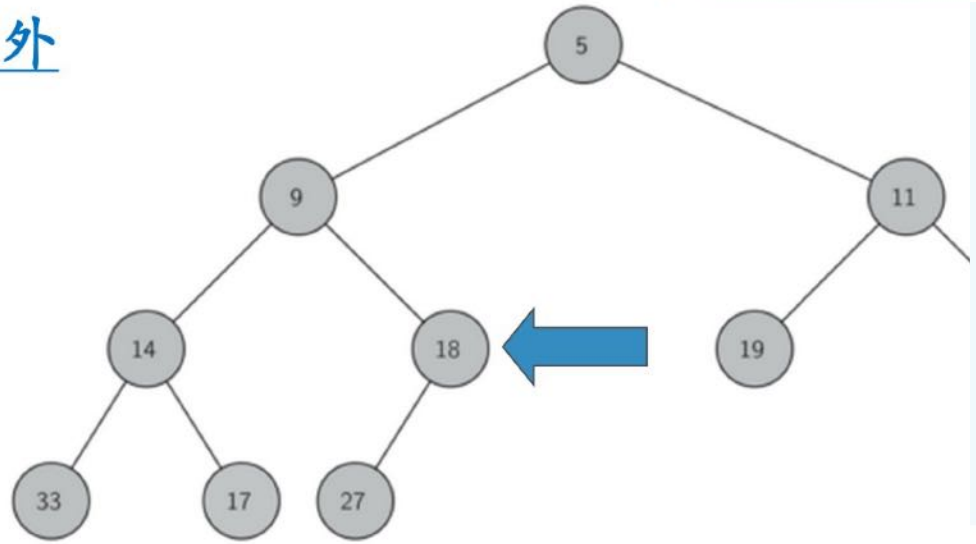
  树根左右子树拥有相同数量的节点

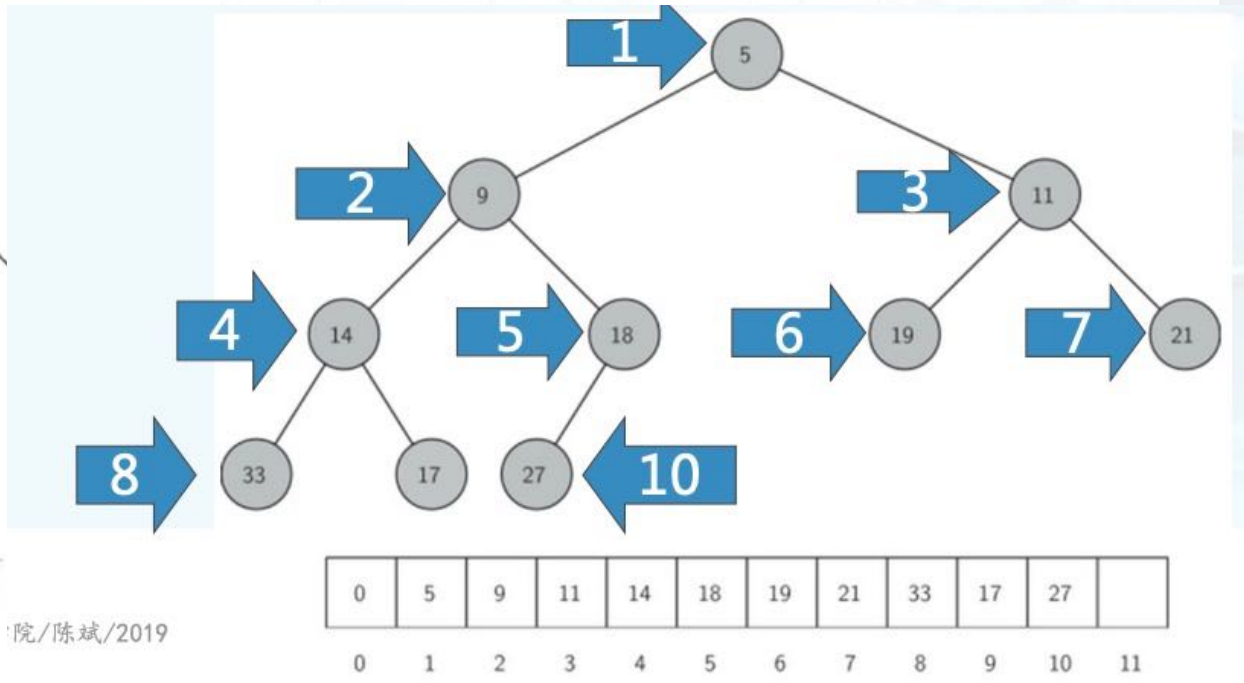


完全二叉树

- 我们采用“完全二叉树”的结构来近似实现“平衡”

  完全二叉树，叶节点最多只出现在最底层和次底层，而且最底层的叶节点都连续集中在最左边，每个内部节点都有两个子节点，最多可有1个节点例外

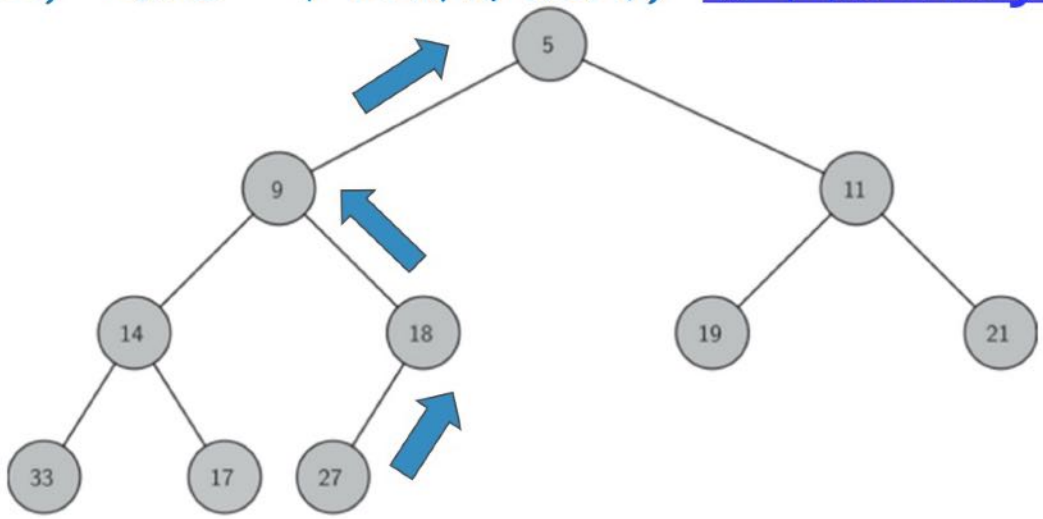




堆次序Heap Order

- 任何一个节点x，其父节点p中的key均小于x中的key

这样，符合“堆”性质的二叉树，其中任何一条路径，均是一个已排序数列，根节点的key最小

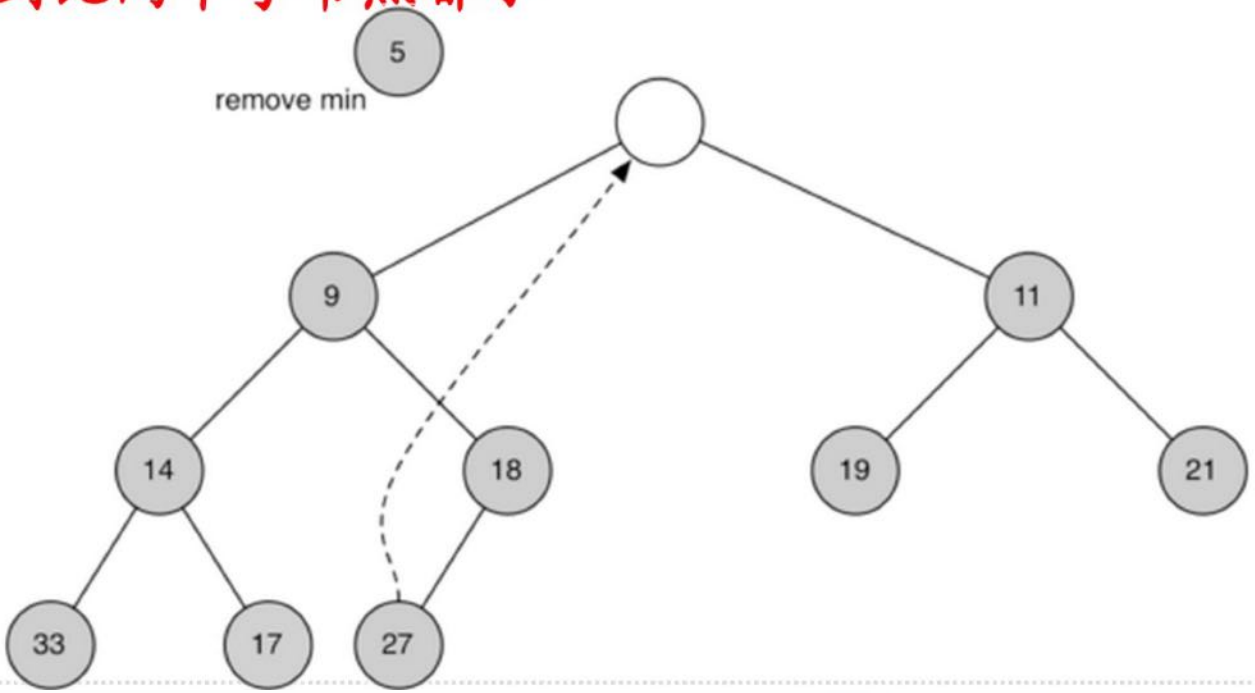


0	5	9	11	14	18	19	21	33	17	27	
0	1	2	3	4	5	6	7	8	9	10	11

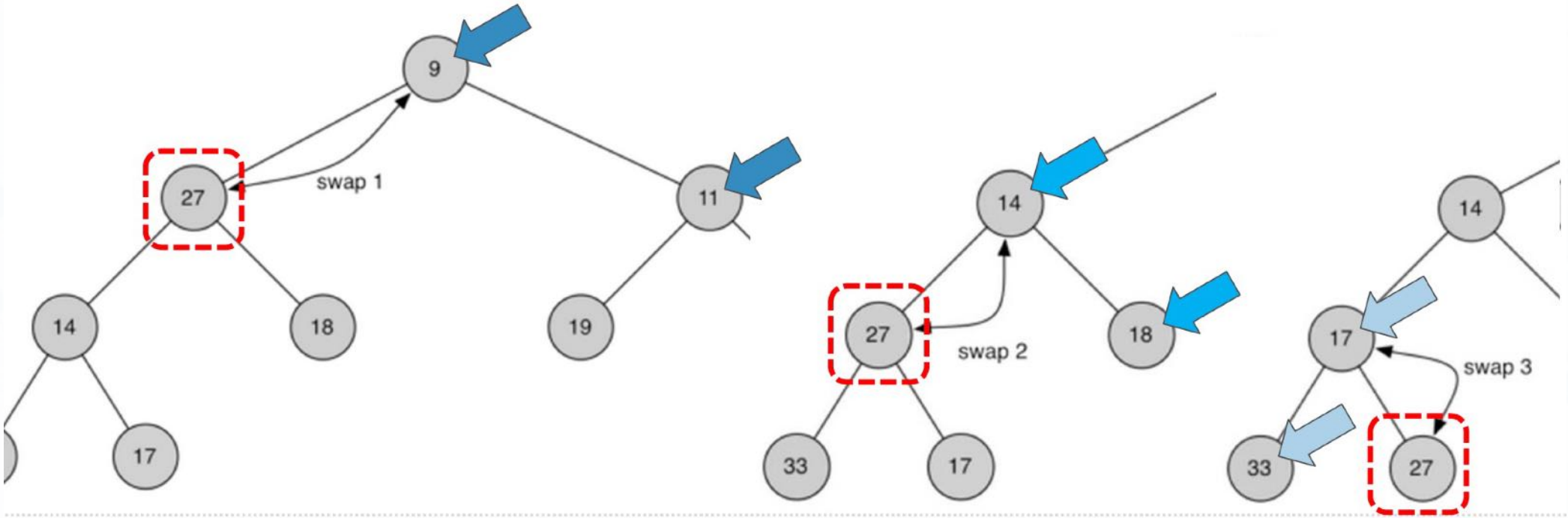
❖ delMin()方法

同样，这么简单的替换，还是破坏了“堆”次序

解决方法：将新的根节点沿着一条路径“下沉”，直到比两个子节点都小



问题解答：为什么要跟较小的子节点交换？



“下沉”路径的选择：如果比子节点大，那么选择较小的子节点交换下沉

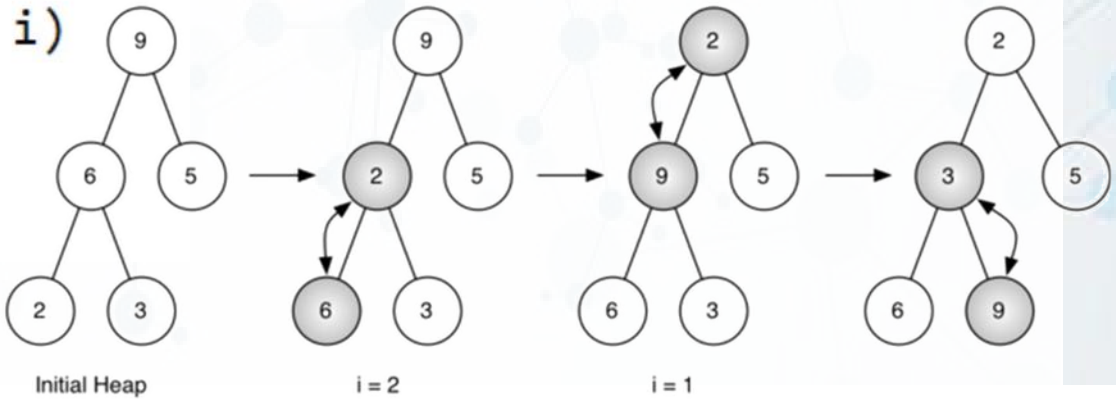
问题解答：堆排序：1-建堆；2-n次delMin

- **buildHeap(lst)方法：从无序表生成“堆”**

  其实，用“下沉”法，能够将总代价控制在O(n)

从最后节点的父节点开始
因叶节点无需下沉

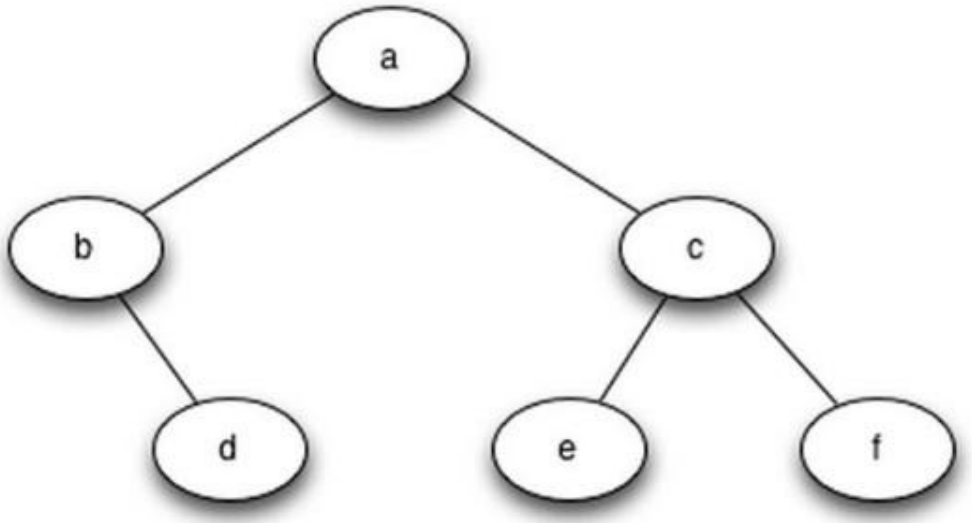
```
def buildHeap(self,alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    print(len(self.heapList), i)
    while (i > 0):
        print(self.heapList, i)
        self.percDown(i)
        i = i - 1
    print(self.heapList,i)
```



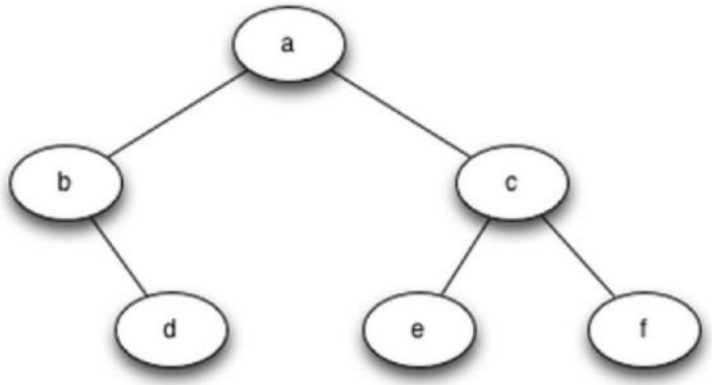
https://zhuanlan.zhihu.com/p/51224401

【K09】树的课堂练习

- 一、写一个buildTree函数（返回一个BinaryTree对象），函数通过调用BinaryTree类方法，返回如图所示的二叉树：
- 二、请为链接实现的BinaryTree类写一个__str__方法，把二叉树的内容用嵌套列表的方式打印输出
- 三、请为链接实现的BinaryTree类写一个height方法，返回树的高度。



一、写一个buildTree函数（返回一个BinaryTree对象），函数通过调用BinaryTree类方法，返回如图所示的二叉树：



二、请为链接实现的BinaryTree类写一个__str__方法，把二叉树的内容用**嵌套列表**的方式打印输出。

编写程序：

- 扩展了的BinaryTree类定义，以及buildTree函数的Python代码；
- print(buildTree())

三、请为链接实现的BinaryTree类写一个height方法，返回树的高度。

编写程序：

- BinaryTree类的height方法；
- print(buildTree().height())（能够返回3）

下课！