数据结构与算法（Python版）

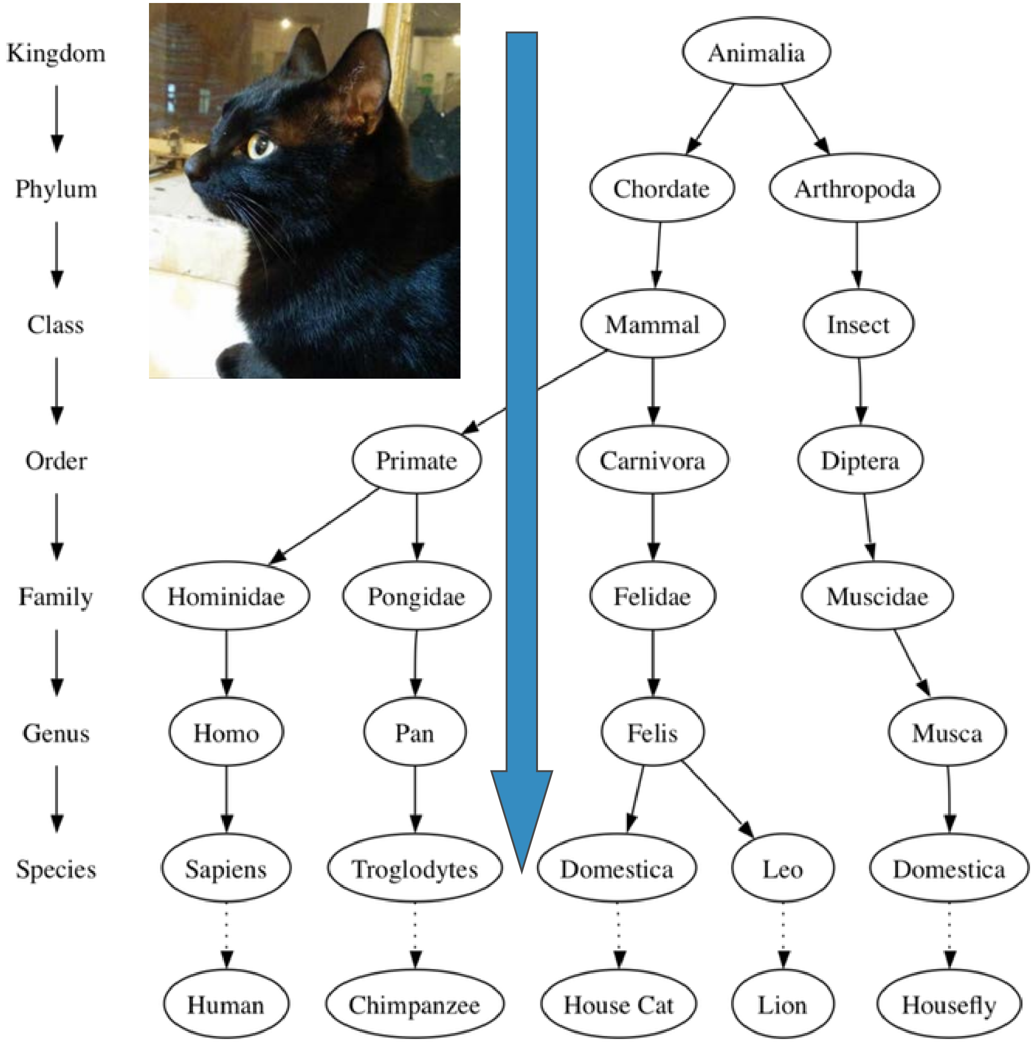
什么是树

陈斌 北京大学 gischen@pku.edu.cn

树的例子

- 本章我们来讨论一种基本的“**非线性**”数据结构——树；
- 树在计算机科学的各个领域中被广泛应用
  操作系统、图形学、数据库系统、计算机网络
- 跟自然界中的树一样，数据结构树也分为：根、枝和叶等三个部分
  一般数据结构的图示把根放在上方，叶放在下方

树的例子：生物学物种分类体系



树的例子：生物学物种分类体系

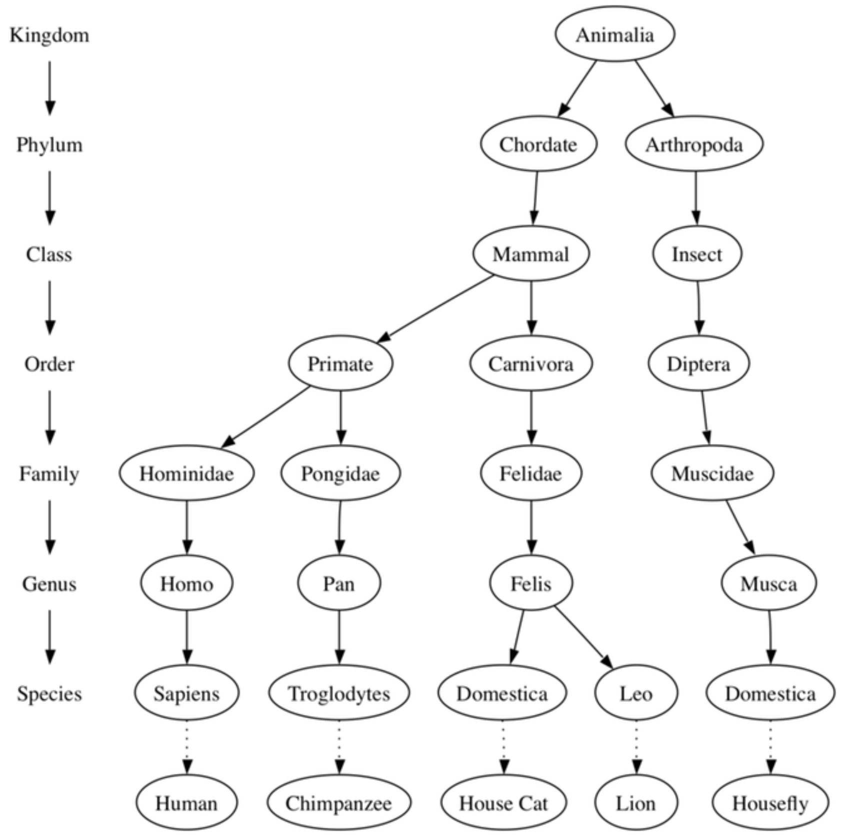
- ❖ **首先分类体系是层次化的**

  树是一种分层结构

  越接近顶部的层越普遍

  越接近底部的层越独特

  界、门、纲、目、科、属、种



树的例子：生物学物种分类体系

- **分类树的第二个特征：一个节点的子节点与另一个节点的子节点相互之间是隔离、独立的**

  猫属Felis和蝇属Musca下面都有Domestica的同名节点

  但相互之间并无任何关联，可以修改其中一个Domestica而不影响另一个。

树的例子：生物学物种分类体系

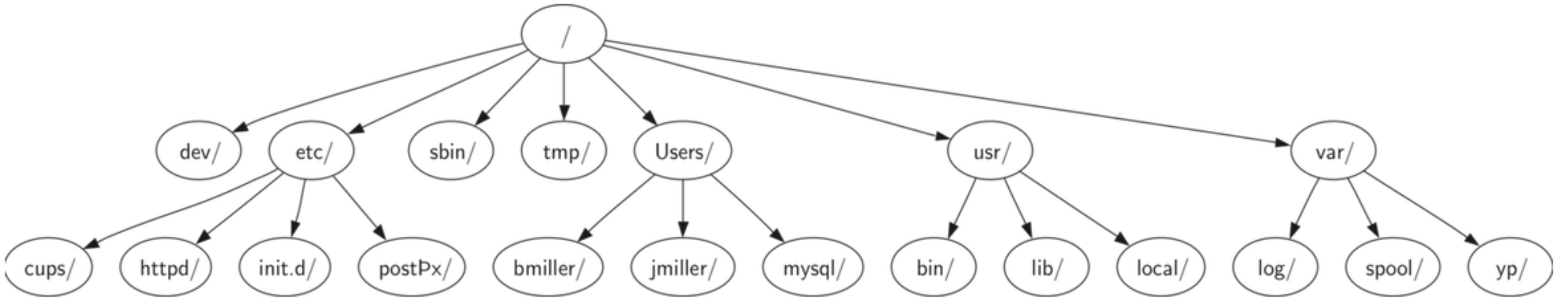
- **分类树的第三个特征：每一个叶节点都具有唯一性**

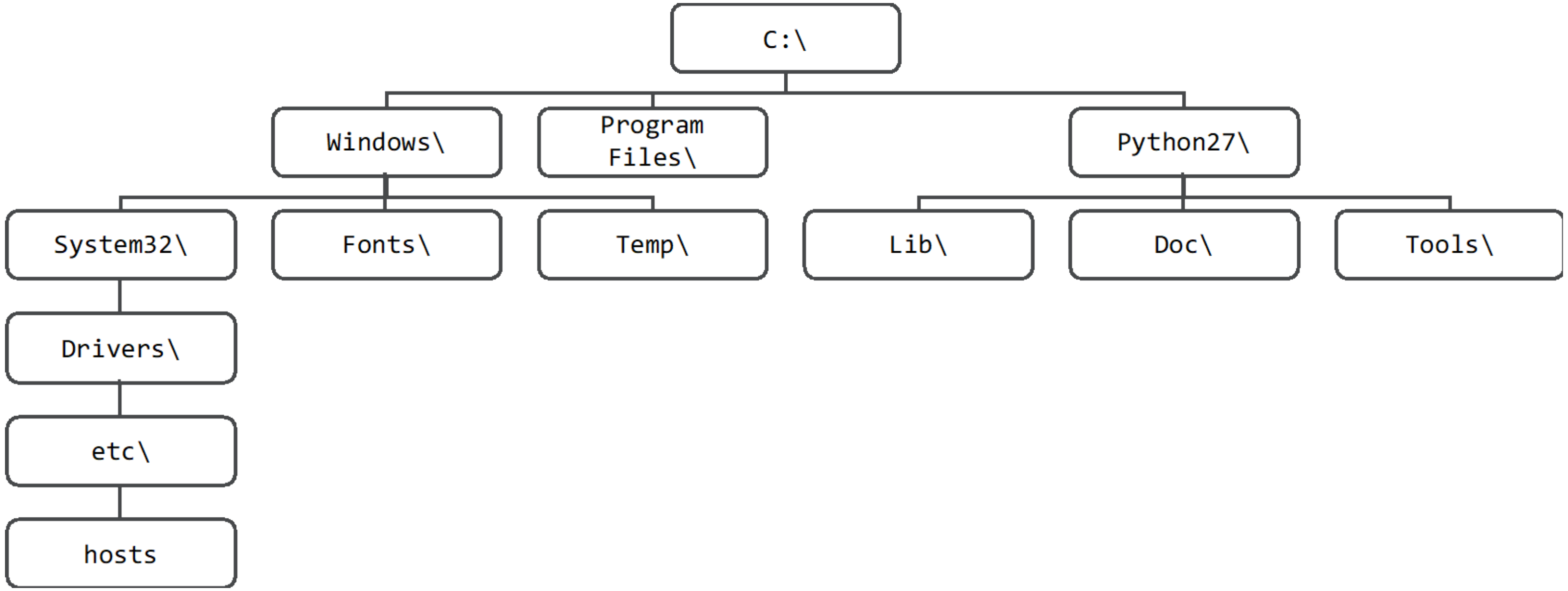
  可以用从根开始到达每个种的完全路径来唯一标识每个物种

  动物界->脊索门->哺乳纲->食肉目->猫科->猫属->家猫种

  Animalia->Chordate->Mammal->Carnivora->Felidae->Felis->Domestica

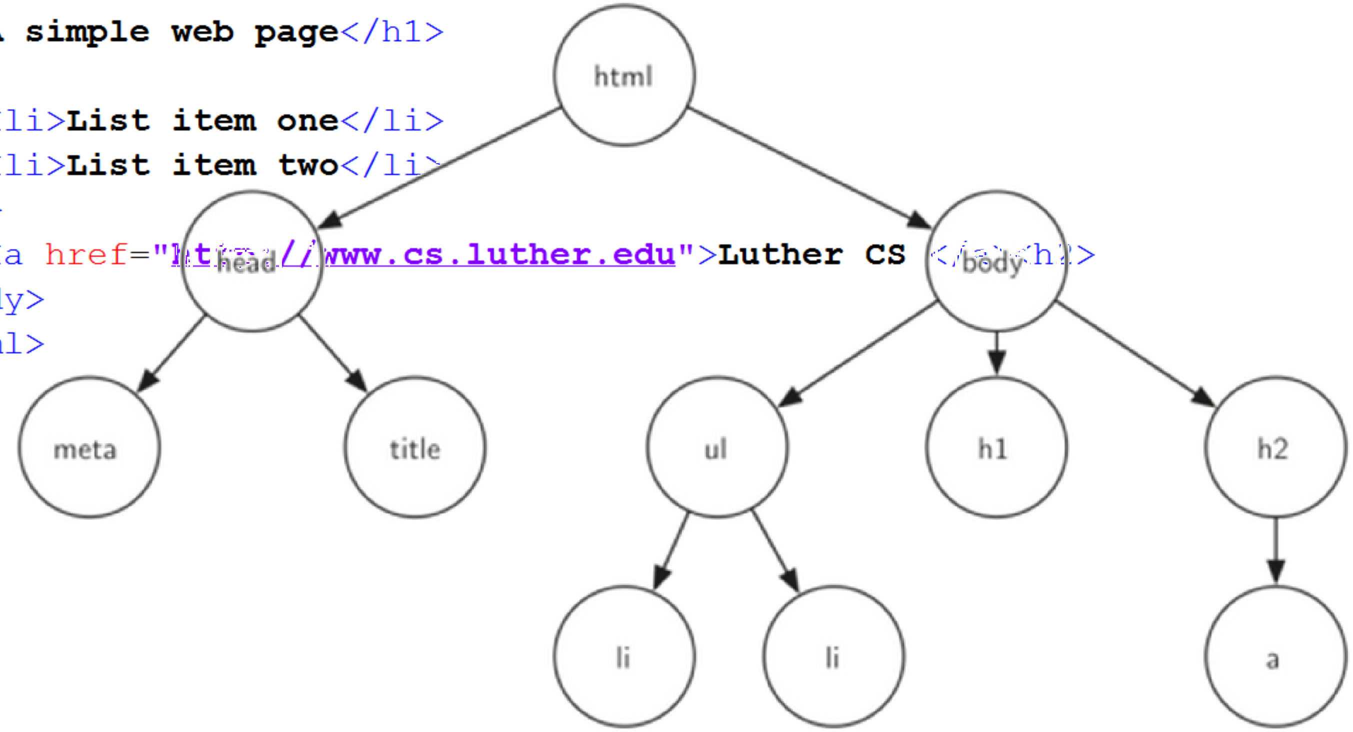
树的例子：文件系统


/
dev/
etc/
sbin/
tmp/
Users/
usr/
var/
cups/
httpd/
init.d/
postfix/
bmiller/
jmiller/
mysql/
bin/
lib/
local/
log/
spool/
yp/

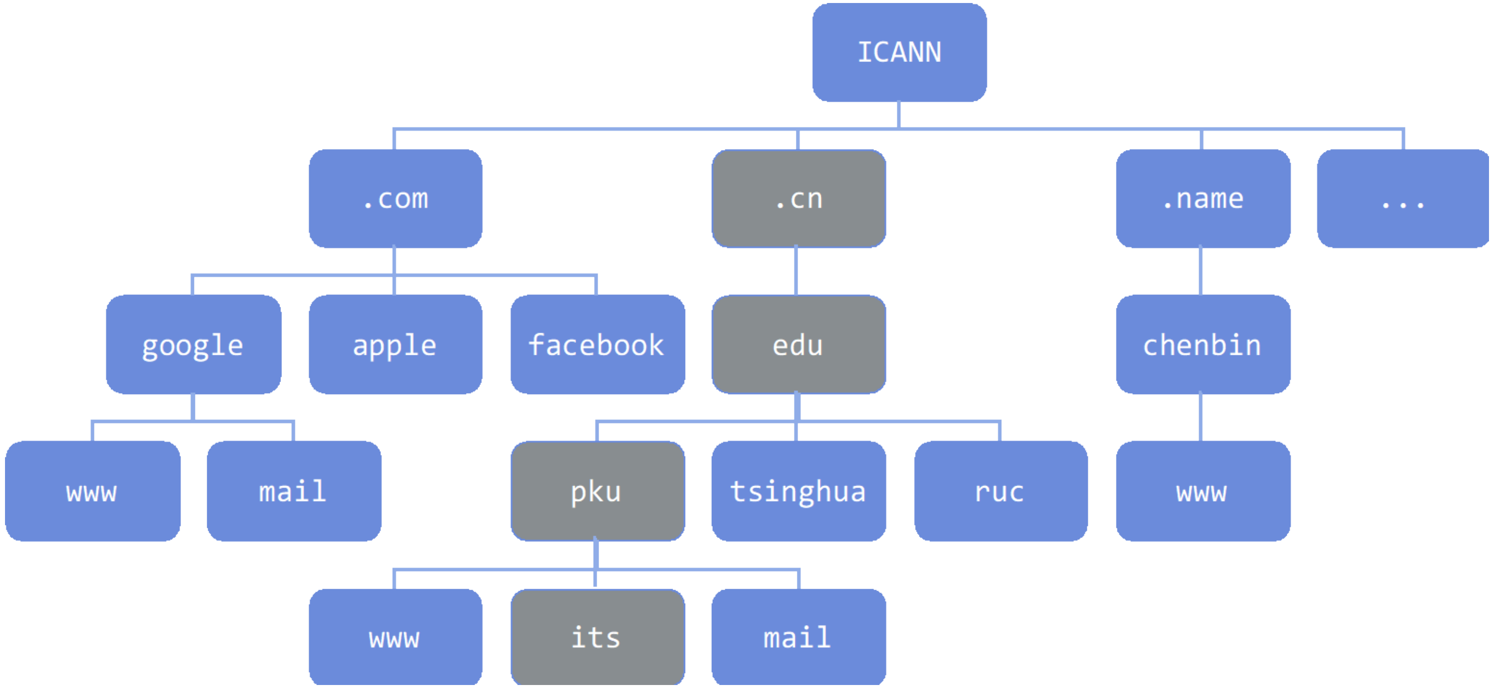

C:\
Windows\
Program Files\
Python27\
System32\
Fonts\
Temp\
Lib\
Doc\
Tools\
Drivers\
etc\
hosts

树的例子：HTML文档（嵌套标记）


```html
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a></h2>
</body>
</html>
```



树的例子：域名体系


ICANN
.com
.cn
.name
...
google
apple
facebook
edu
chenbin
www
mail
pku
tsinghua
ruc
www
www
its
mail

数据结构与算法（Python版）

树结构相关术语

陈斌 北京大学 gischen@pku.edu.cn

树结构相关术语

- **节点Node：组成树的基本部分**

  每个节点具有名称，或“键值”，节点还可以保存额外数据项，数据项根据不同的应用而变

- **边Edge：边是组成树的另一个基本部分**

  每条边恰好连接两个节点，表示节点之间具有关联，边具有出入方向；

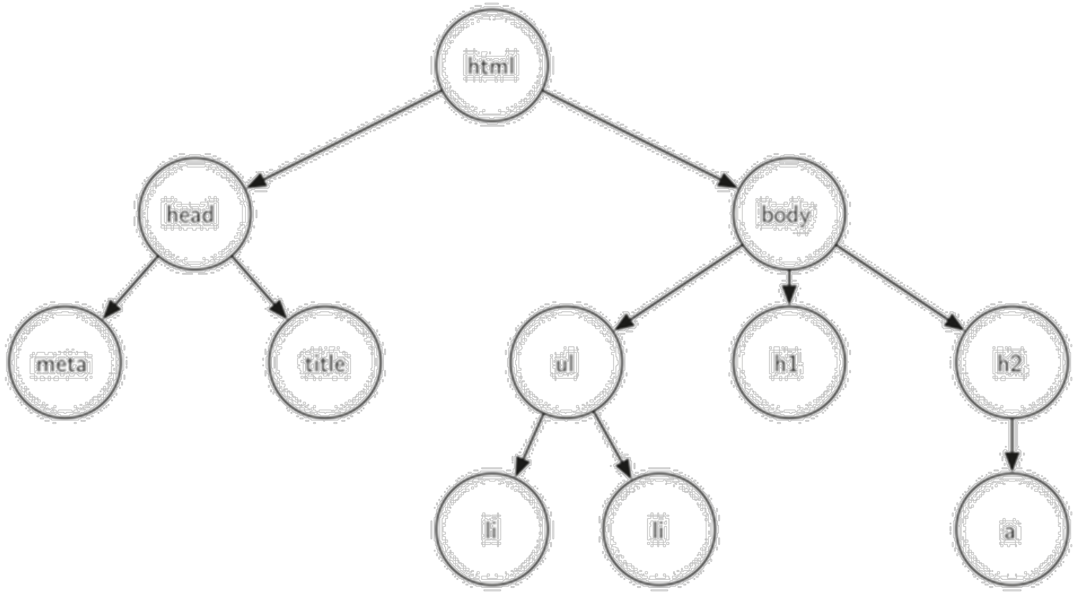
  每个节点（除根节点）恰有一条来自另一节点的入边；

  每个节点可以有多条连到其它节点的出边。

树结构相关术语

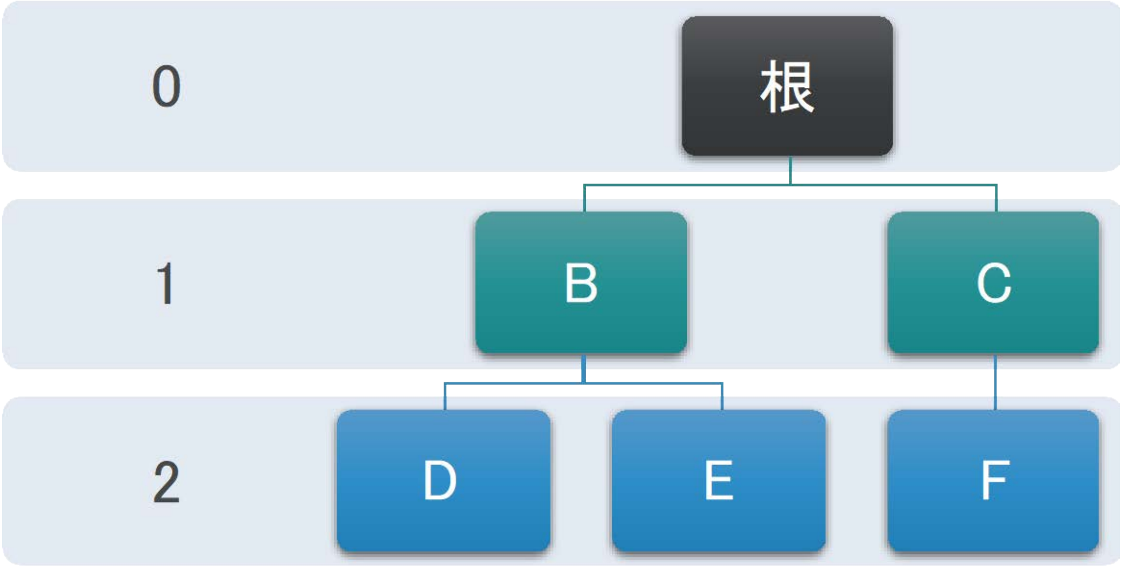
- ❖ **根Root：树中唯一一个没有入边的节点**
- ❖ **路径Path：由边依次连接在一起的节点的有序列表**

如：HTML->BODY->UL->LI，是一条路径



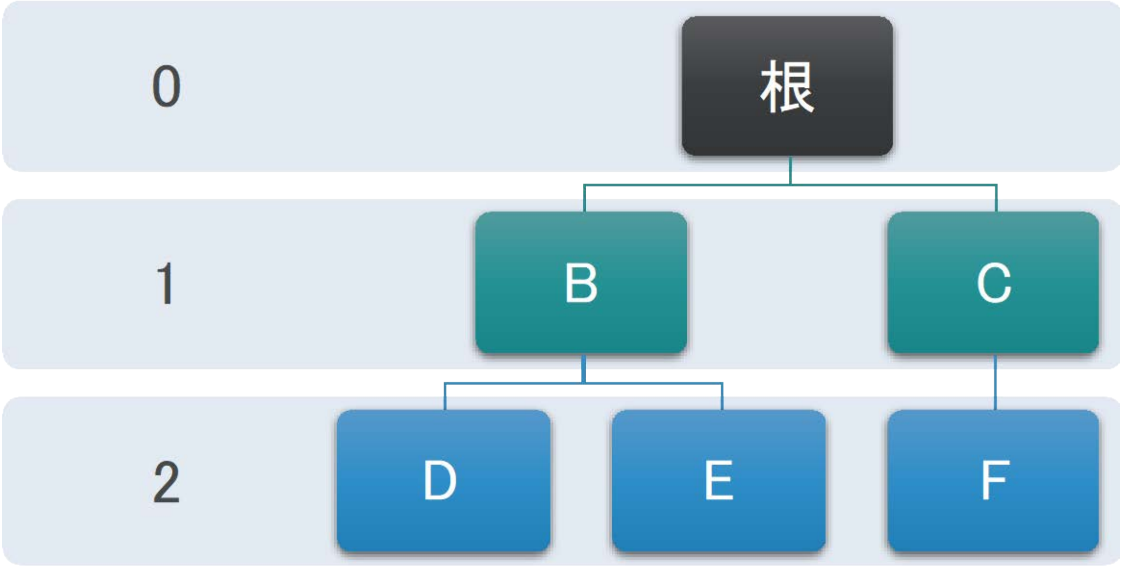
树结构相关术语

- 子节点Children：入边均来自于同一个节点的若干节点，称为这个节点的子节点
- 父节点Parent：一个节点是其所有出边所连接节点的父节点



树结构相关术语

- 兄弟节点Sibling：具有同一个父节点的节点之间称为兄弟节点
- 子树Subtree：一个节点和其所有子孙节点，以及相关边的集合



树结构相关术语

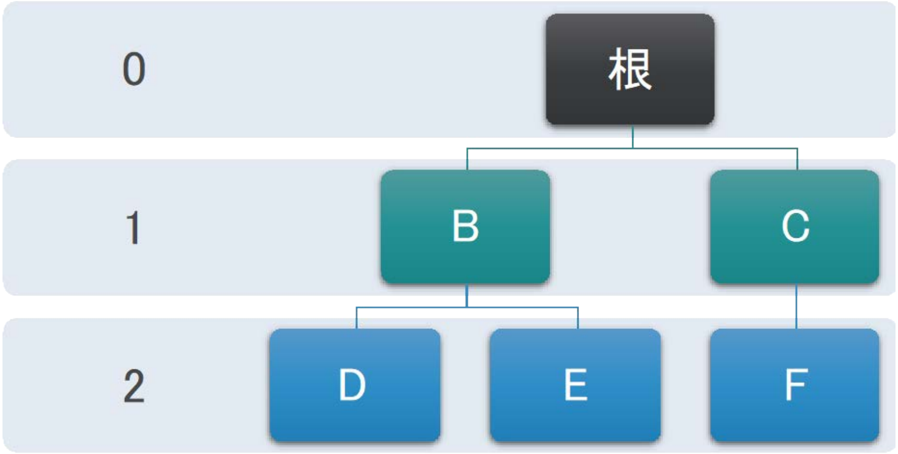
❖ **叶节点Leaf：没有子节点的节点称为叶节点**

❖ **层级Level：从根节点开始到达一个节点的路径，所包含的边的数量，称为这个节点的层级。**

如D的层级为2，根节点的层级为0

❖ **高度：树中所有节点的最大层级称为树的高度**

如右图树的高度为2



树的定义1

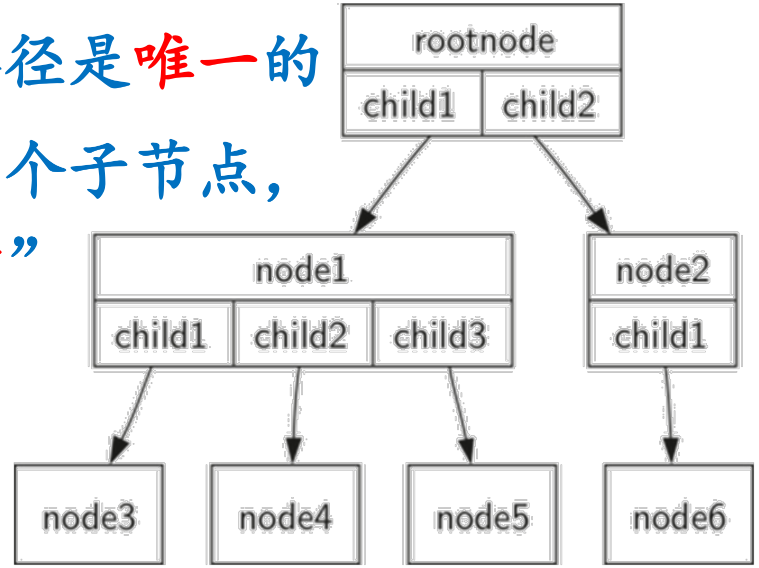
❖ **树由若干节点，以及两两连接节点的边组成，并有如下性质**

其中一个节点被设定为根；

每个节点n(除根节点)，都恰连接一条来自节点p的边，p是n的父节点；

每个节点从根开始的路径是唯一的

如果每个节点最多有两个子节点，这样的树称为“二叉树”

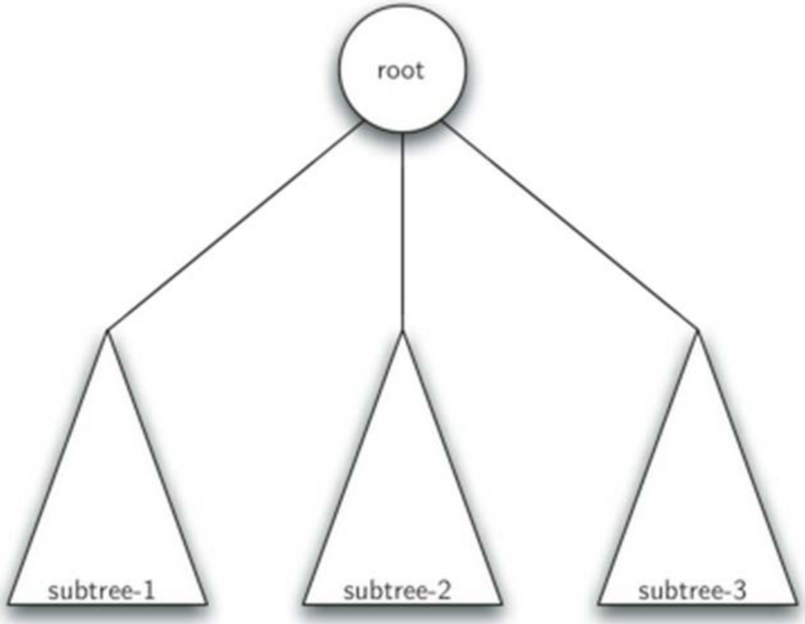


树的定义2（递归定义）

- **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。



数据结构与算法（Python版）

树的嵌套列表实现

陈斌 北京大学 gischen@pku.edu.cn

实现树：嵌套列表法

- 首先我们尝试用Python List来实现二叉树树数据结构；
- 递归的嵌套列表实现二叉树，由具有3个元素的列表实现：

  第1个元素为根节点的值；

  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]
```

实现树：嵌套列表法

- **以右图的示例，一个6节点的二叉树**

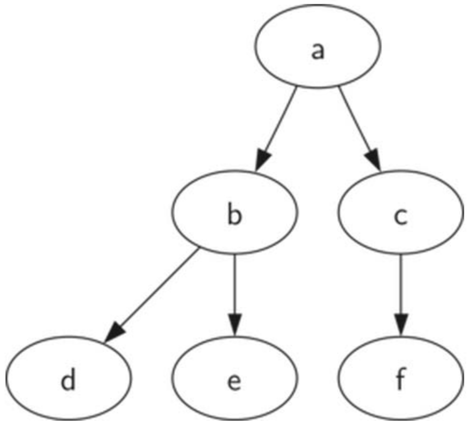
  根是myTree[0]，左子树myTree[1]，右子树myTree[2]

- **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

```
myTree = ['a', #树根
          ['b', #左子树
           ['d', [], []],
           ['e', [], []] ],
          ['c', #右子树
           ['f', [], []],
           [] ]
         ]
```



实现树：嵌套列表法

❖ **我们通过定义一系列函数来辅助操作嵌套列表**

BinaryTree创建仅有根节点的二叉树

insertLeft/insertRight将新节点插入树中作为其直接的左/右子节点

get/setRootVal则取得或返回根节点

getLeft/RightChild返回左/右子树

嵌套列表法代码


```python
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
```

嵌套列表法代码


```python
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]
```

实现树：嵌套列表法


```
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, [], []]
>>> 
```

数据结构与算法（Python版）

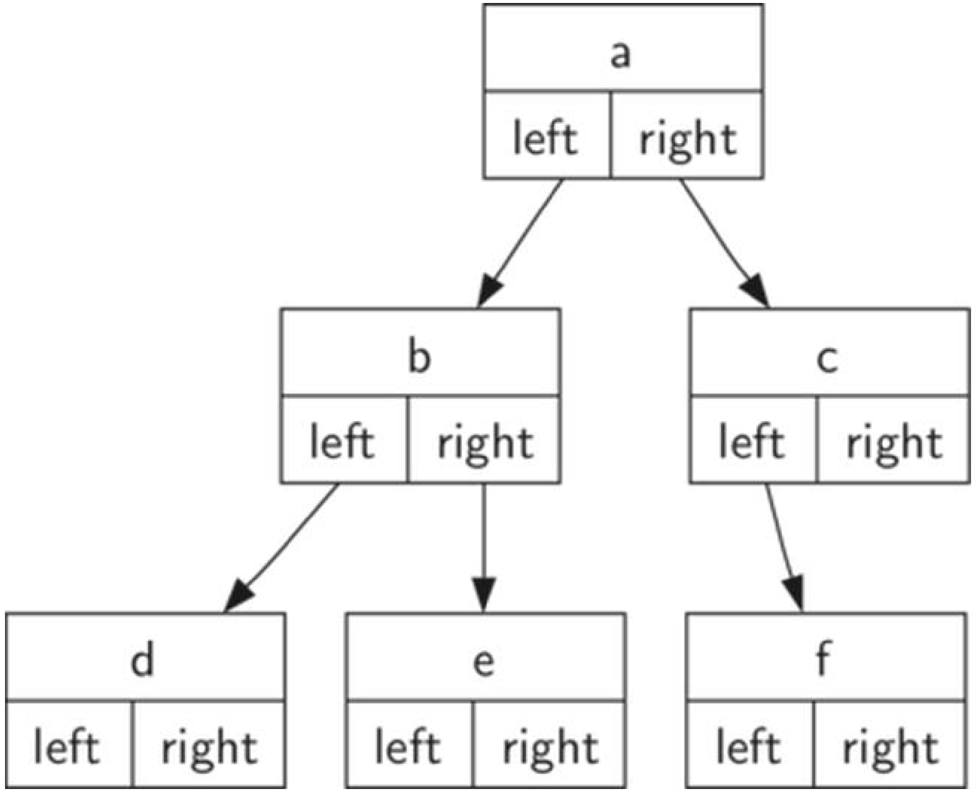
树的链表实现

陈斌 北京大学 gischen@pku.edu.cn

实现树：节点链接法

❖ 同样可以用节点链接法来实现树

每个节点保存根节点的数据项，以及指向左右子树的链接



实现树：节点链接法

❖ 定义一个BinaryTree类

成员key保存根节点数据项

成员left/rightChild则保存指向左/右子树的引用（同样是BinaryTree对象）

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

实现树：节点链接法


```python
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
```

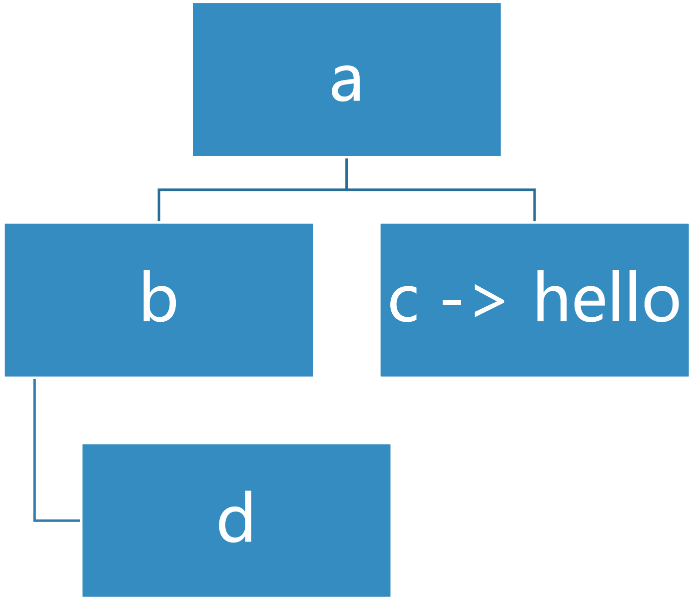
实现树：节点链接法


```
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
```

实现树：节点链接法

❖ 请画出r的图示

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



数据结构与算法（Python版）

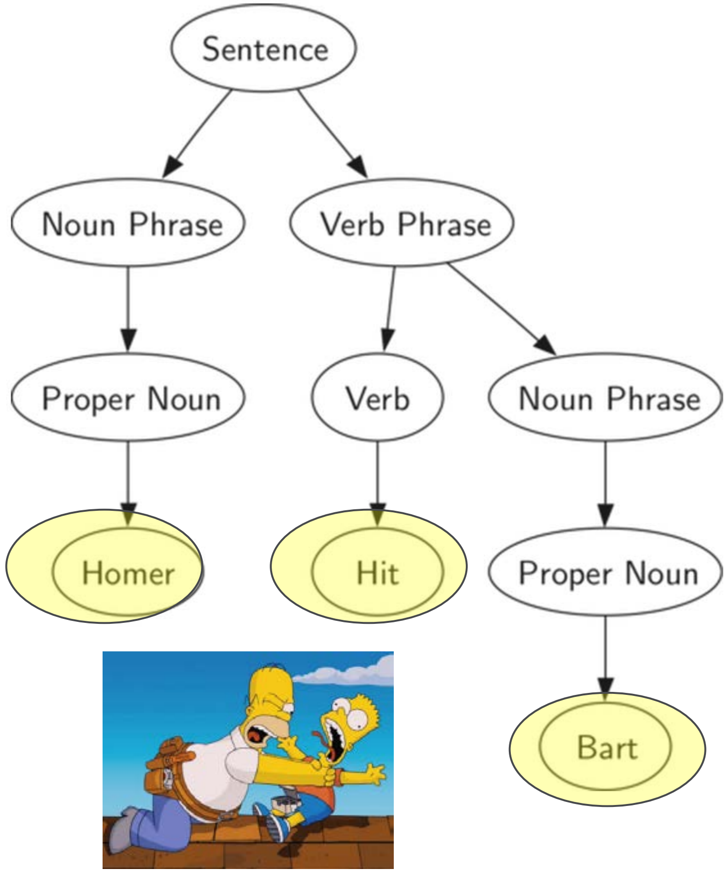
树的应用：表达式解析（上）

陈斌 北京大学 gischen@pku.edu.cn

树的应用：解析树（语法树）

- 将树用于表示语言中句子，可以分析句子的各种语法成分，对句子的各种成分进行处理
- 语法分析树

  主谓宾，定状补



树的应用：解析树（语法树）

- **程序设计语言的编译**
  - 词法、语法检查
  - 从语法树生成目标代码
- **自然语言处理**
  - 机器翻译、语义理解

树的应用：表达式解析

❖ **我们还可以将表达式表示为树结构**

叶节点保存操作数，内部节点保存操作符

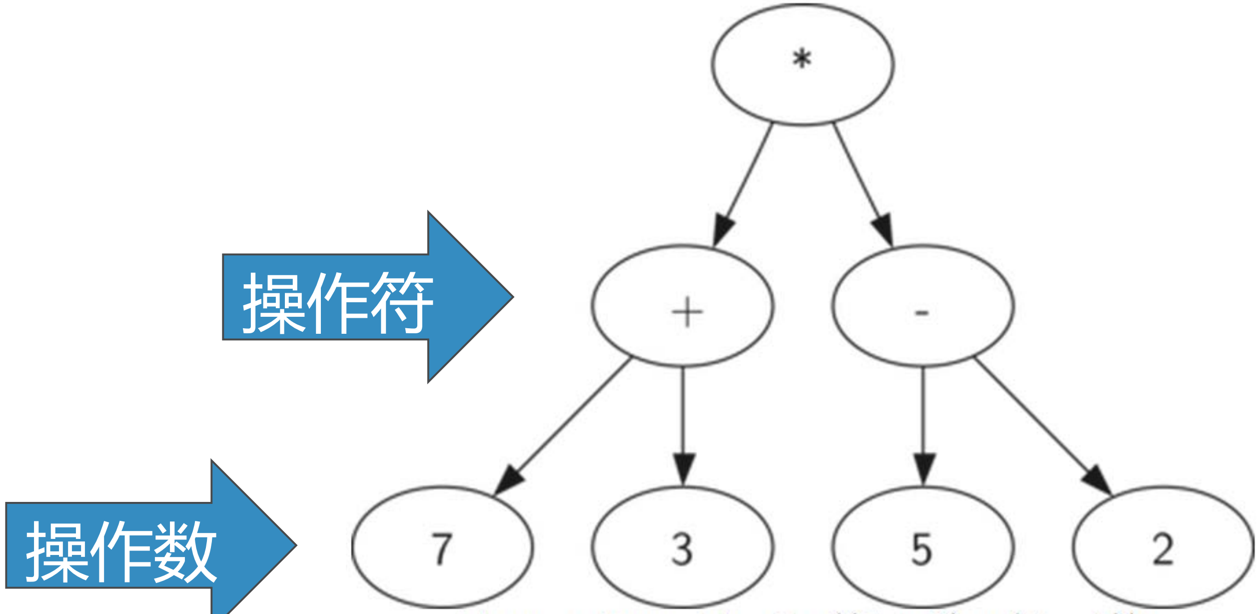


Figure 2: Parse Tree for $((7+3)*(5-2))$

树的应用：表达式解析

- 全括号表达式((7+3)*(5-2))

由于括号的存在，需要计算*的话，就必须先计算7+3和5-2，表达式层次决定计算的优先级

越底层的表达式，优先级越高

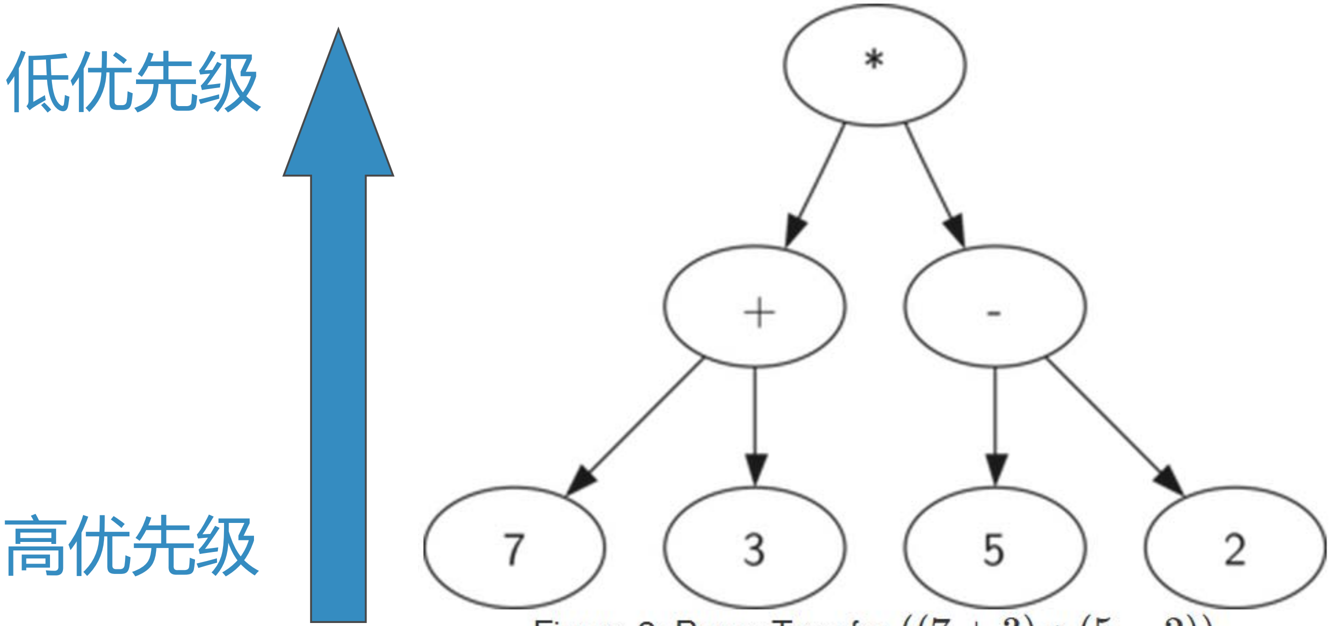
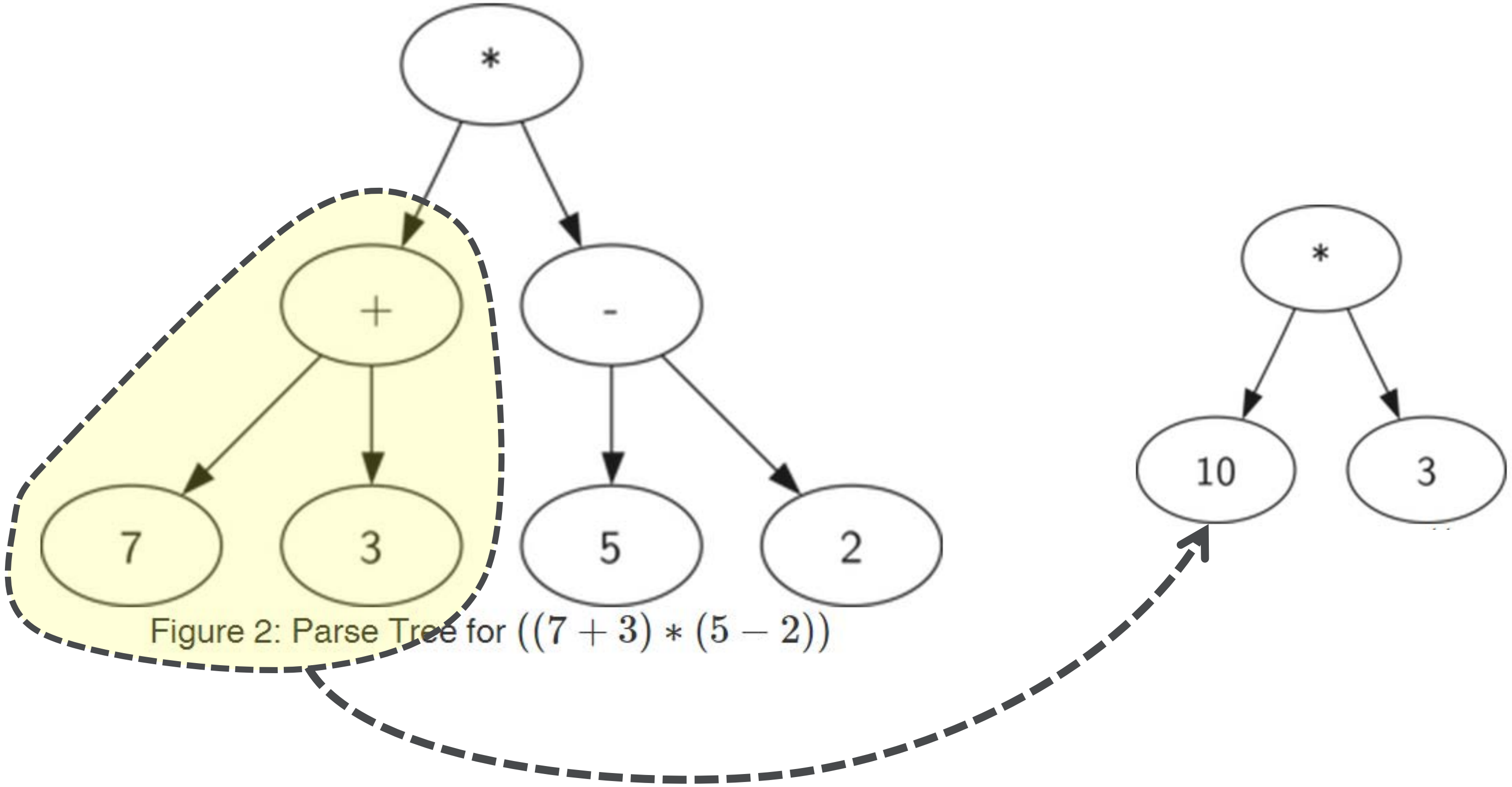


Figure 2: Parse Tree for $((7+3)*(5-2))$

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



表达式解析树

- 下面，我们用树结构来做如下尝试

  从全括号表达式构建表达式解析树

  利用表达式解析树对表达式求值

  从表达式解析树恢复原表达式的字符串形式

- 首先，全括号表达式要分解为单词Token列表

  其单词分为括号“（）”、操作符“+-*/”和操作数“0~9”这几类

  左括号就是表达式的开始，而右括号是表达式的结束

建立表达式解析树：实例

❖ **全括号表达式：(3+(4*5))**

分解为单词表

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

(3+(4*5))

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

建立表达式解析树：实例

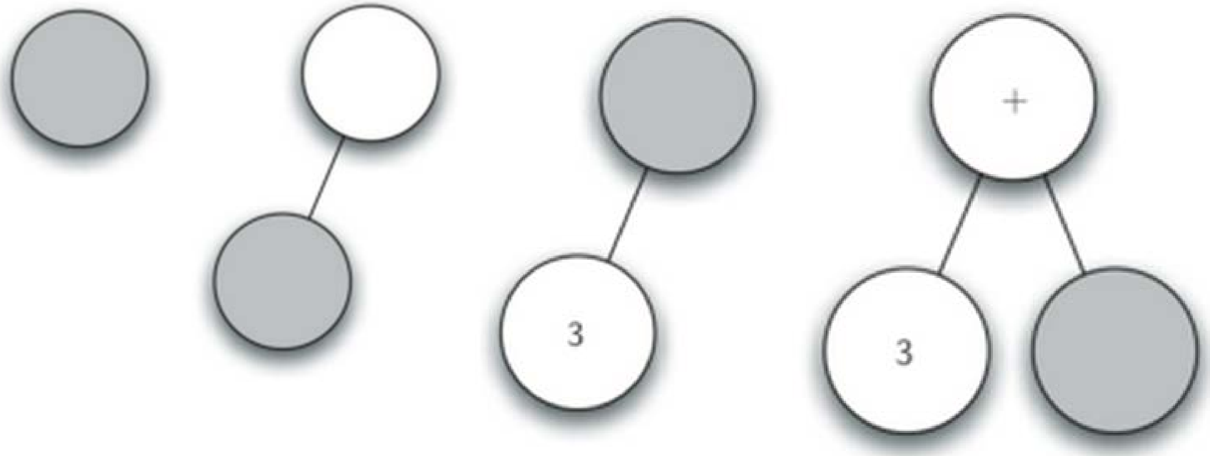
❖ 创建表达式解析树过程

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



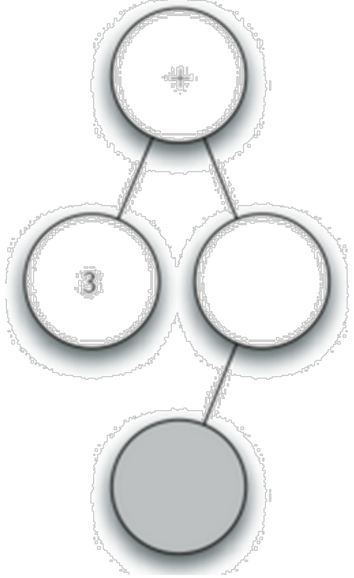
建立表达式解析树：实例

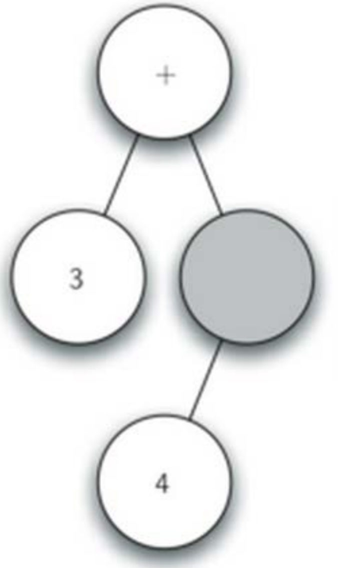
❖ 创建表达式解析树过程

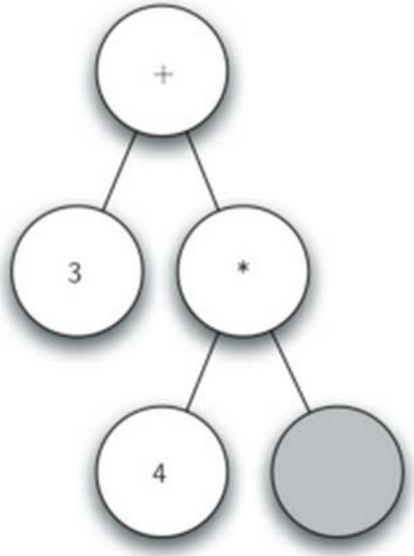
读入'('，创建左子节点，当前节点下降

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







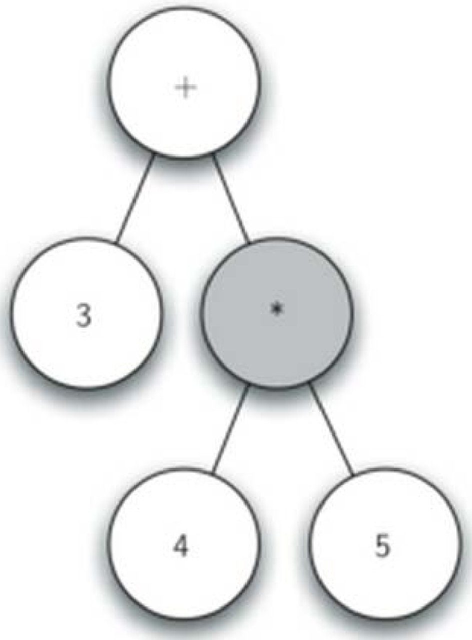
建立表达式解析树：实例

❖ 创建表达式解析树过程

读入'5'，当前节点设置为5，上升到父节点

读入')'，上升到父节点

读入')'，再上升到父节点



建立表达式解析树：规则

- ❖ 从左到右**扫描**全括号表达式的每个**单词**，依据规则建立解析树

如果当前单词是"("：为当前节点添加一个新节点作为其左子节点，当前节点下降为这个新节点

如果当前单词是操作符"+,-,/,*"：将当前节点的值设为此符号，为当前节点添加一个新节点作为其右子节点，当前节点下降为这个新节点

如果当前单词是操作数：将当前节点的值设为此数，当前节点上升到父节点

如果当前单词是")"：则当前节点上升到父节点

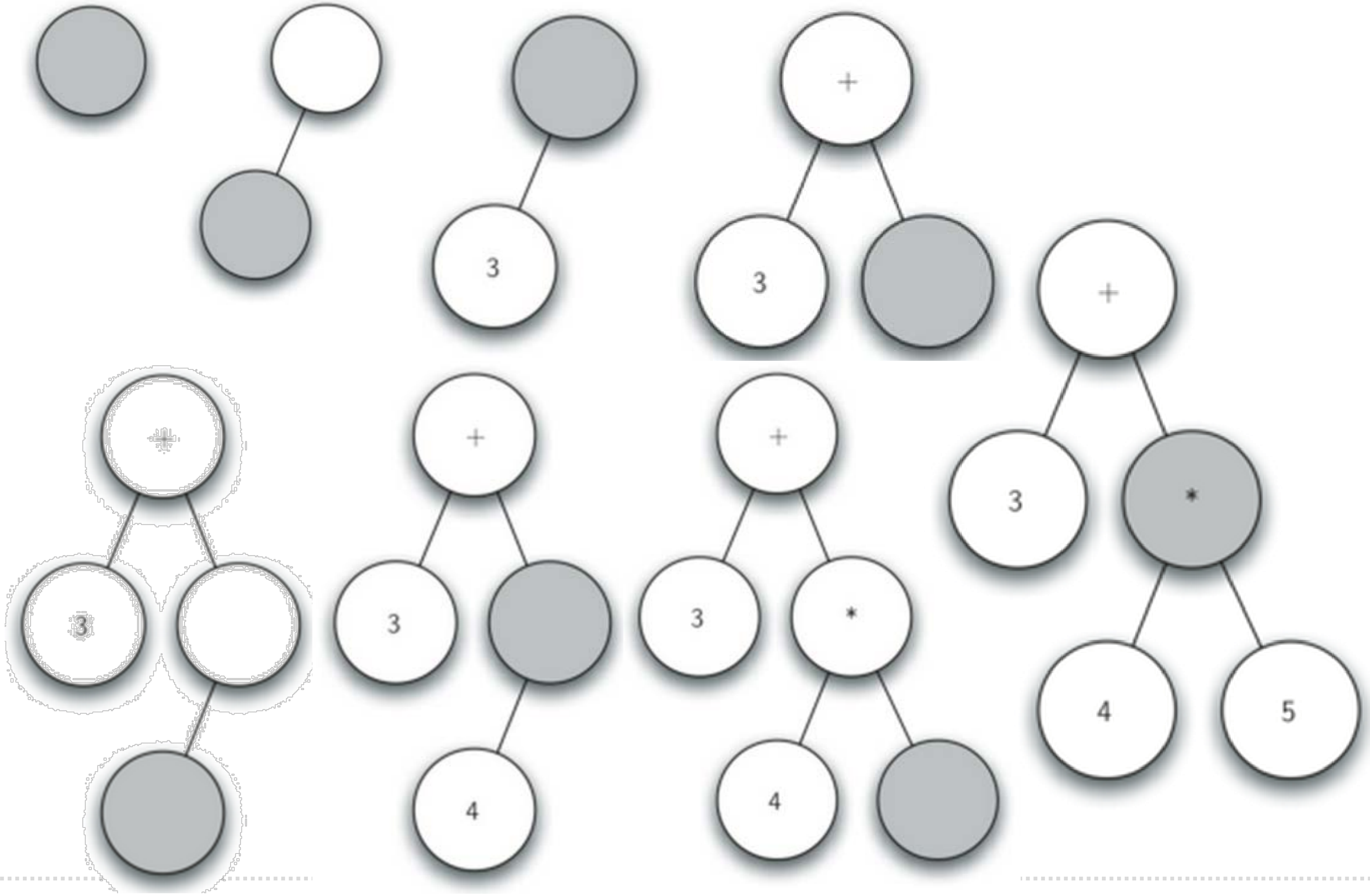
数据结构与算法（Python版）

树的应用：表达式解析（下）

陈斌 北京大学 gischen@pku.edu.cn

建立表达式解析树：实例

❖ 全括号表达式：(3+(4*5))



建立表达式解析树：思路

- 从图示过程中我们看到，创建树过程中关键的是对当前节点的跟踪

  创建左右子树可调用insertLeft/Right

  当前节点设置值，可以调用setRootVal

  下降到左右子树可调用getLeft/RightChild

  但是，上升到父节点，这个没有方法支持！

- 我们可以用一个**栈**来记录跟踪父节点

  当前节点下降时，将下降前的节点push入栈

  当前节点需要上升到父节点时，上升到pop出栈的节点即可！

建立表达式解析树：代码


```
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
```

入栈下降

表达式开始

入栈下降

操作数

出栈上升

操作符

表达式结束

出栈上升

利用表达式解析树求值：思路

- 创建了表达式解析树，可用来进行求值
- 由于二叉树BinaryTree是一个递归数据结构，自然可以用递归算法来处理
- 求值递归函数evaluate

  由前述对子表达式的描述，可从树的底层子树开始，逐步向上层求值，最终得到整个表达式的值

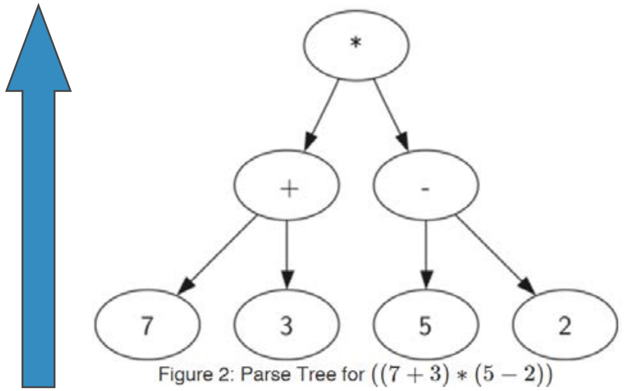


Figure 2: Parse Tree for $((7+3)*(5-2))$

利用表达式解析树求值：思路

- **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

利用表达式解析树求值：思路

❖ **一个增加程序可读性的技巧：函数引用**

```
import operator

op= operator.add
```

```
>>> import operator
>>> operator.add
<built-in function add>
>>> operator.add(1,2)
3
>>> op= operator.add
>>> n= op(1,2)
>>> n
3
```

利用表达式解析树求值：代码


```
import operator
def evaluate(parseTree):
    opers = {'+':operator.add, '-':operator.sub, \
             '*':operator.mul, '/':operator.truediv}

    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()

    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC),evaluate(rightC))
    else:
        return parseTree.getRootVal()
```

缩小规模

递归调用

基本结束条件

数据结构与算法（Python版）

树的遍历

陈斌 北京大学 gischen@pku.edu.cn

树的遍历Tree Traversals

- 对一个数据集中的所有数据项进行访问的操作称为“遍历Traversal”
- 线性数据结构中，对其所有数据项的访问比较简单直接

  按照顺序依次进行即可

- 树的非线性特点，使得遍历操作较为复杂

树的遍历Tree Traversals

- 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。

前序遍历的例子：一本书的章节阅读

数据结构与算法（Python版）

- Book-> Ch1-> S1.1-> S1.2-> S1.2.1-> S1.2.2-> Ch2-> S2.1-> S2.2-> S2.2.1-> S2.2.2

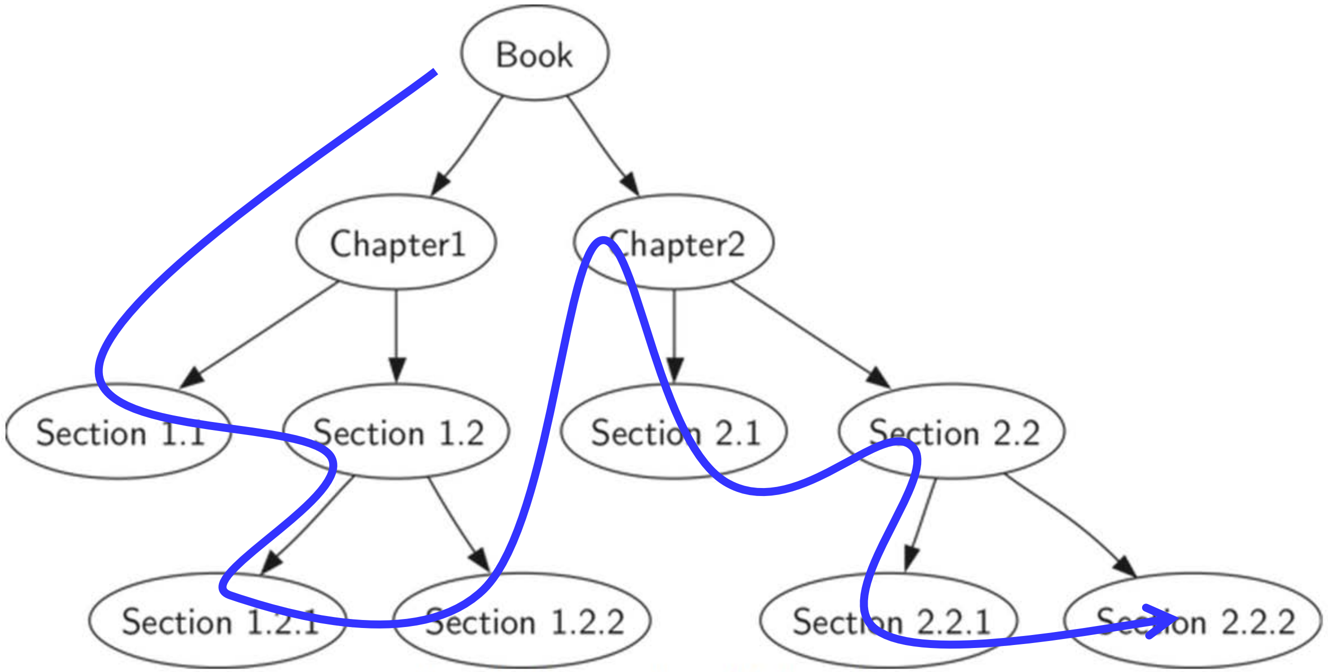


Figure 5: Representing a Book as a Tree

树的遍历：递归算法代码

❖ 树遍历的代码非常简洁！

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

❖ 后序和中序遍历的代码仅需要调整顺序

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

树的遍历：递归算法代码

❖ **也可以在BinaryTree类中实现前序遍历的方法：**

需要加入子树是否为空的判断

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

后序遍历：表达式求值

- ❖ 回顾前述的表达式解析树求值，实际上也是一个后序遍历的过程
- ❖ 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：生成全括号中缀表达式

- 采用中序遍历递归算法来生成全括号中缀表达式

  下列代码中对每个数字也加了括号，请自行修改代码去除（课后练习）

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

数据结构与算法（Python版）

优先队列和二叉堆

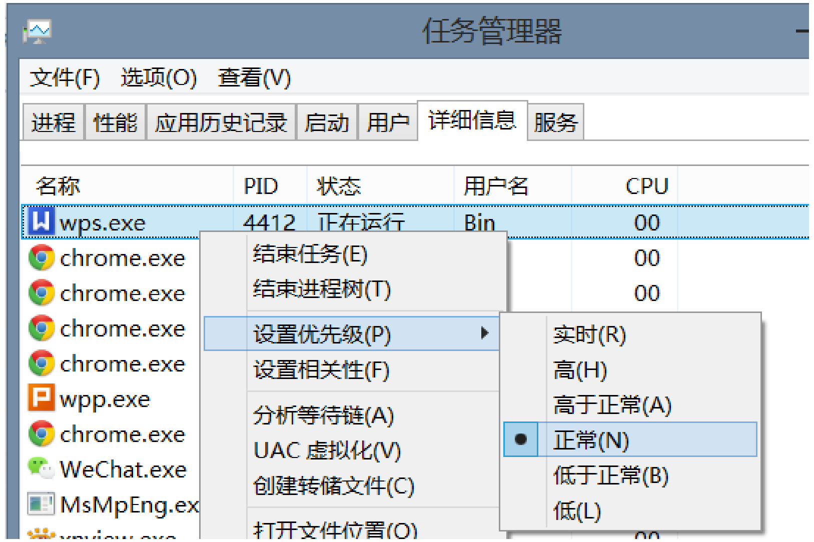
陈斌 北京大学 gischen@pku.edu.cn

优先队列Priority Queue

- **前面我们学习了一种FIFO数据结构队列**
- **队列有一种变体称为“优先队列”。**

银行窗口取号排队，VIP客户可以插到队首

操作系统中执行关键任务的进程或用户特别指定进程在调度队列中靠前



优先队列Priority Queue

- 优先队列的出队跟队列一样从**队首**出队；
- 但在优先队列内部，数据项的次序却是由“**优先级**”来确定：

  高优先级的数据项排在队首，而低优先级的数据项则排在后面。

  这样，优先队列的入队操作就比较复杂，需要将数据项根据其优先级尽量挤到队列前方。

- 思考：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？

二叉堆Binary Heap实现优先队列

- 实现优先队列的经典方案是采用**二叉堆**数据结构

  二叉堆能够将优先队列的入队和出队复杂度都保持在O(log n)

- 二叉堆的有趣之处在于，其逻辑结构上象二叉树，却是用非嵌套的列表来实现的！

- 最小key排在队首的称为“最小堆min heap”

  反之，最大key排在队首的是“最大堆max heap”

二叉堆Binary Heap实现优先队列

❖ **ADT BinaryHeap的操作定义如下：**

BinaryHeap()：创建一个空二叉堆对象；

insert(k)：将新key加入到堆中；

findMin()：返回堆中的最小项，最小项仍保留在堆中；

delMin()：返回堆中的最小项，同时从堆中删除；

isEmpty()：返回堆是否为空；

size()：返回堆中key的个数；

buildHeap(list)：从一个key列表创建新堆

ADT BinaryHeap的操作示例

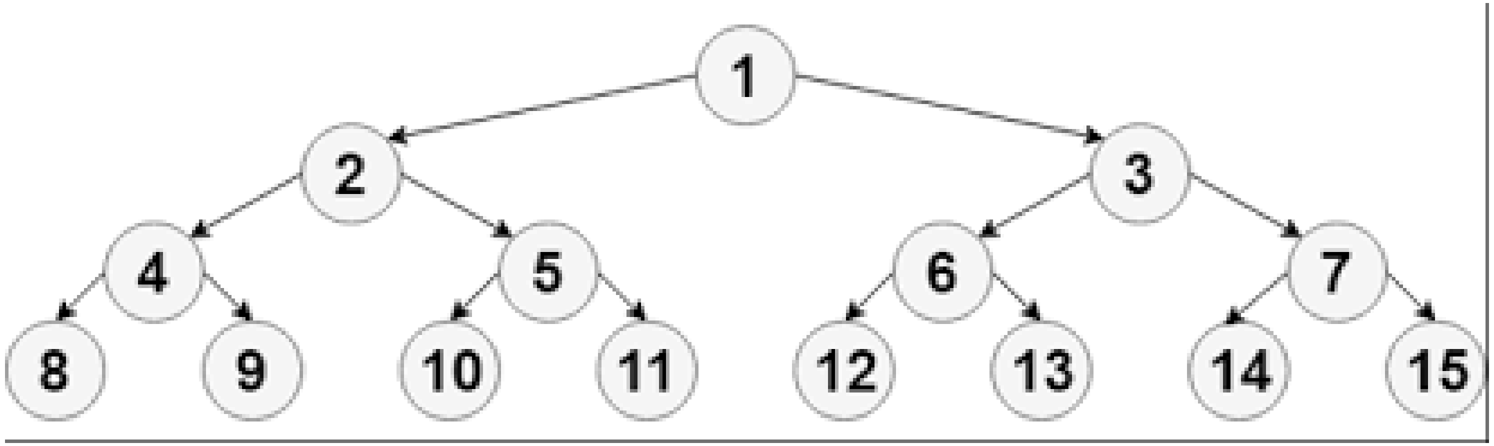

```
from pythonds.trees.binheap import BinHeap

bh = BinHeap()
bh.insert(5)
bh.insert(7)
bh.insert(3)
bh.insert(11)

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

```
>>> ========
>>>
3
5
7
11
>>>
```

用非嵌套列表实现二叉堆

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

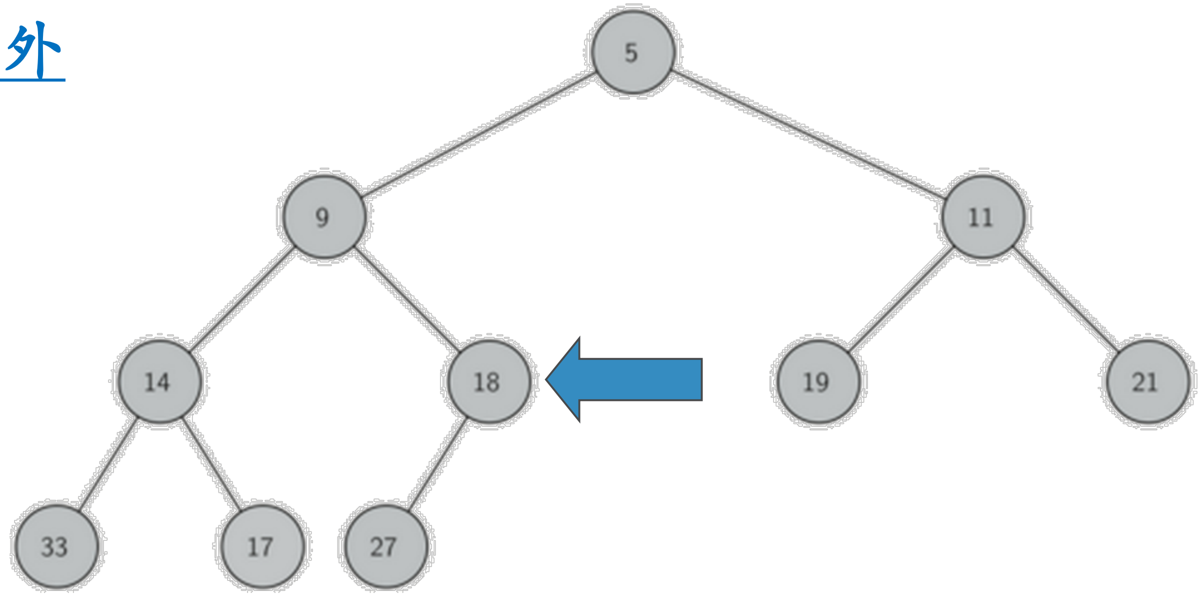
  树根左右子树拥有相同数量的节点



用非嵌套列表实现二叉堆

- 我们采用“**完全二叉树**”的结构来**近似**实现“平衡”

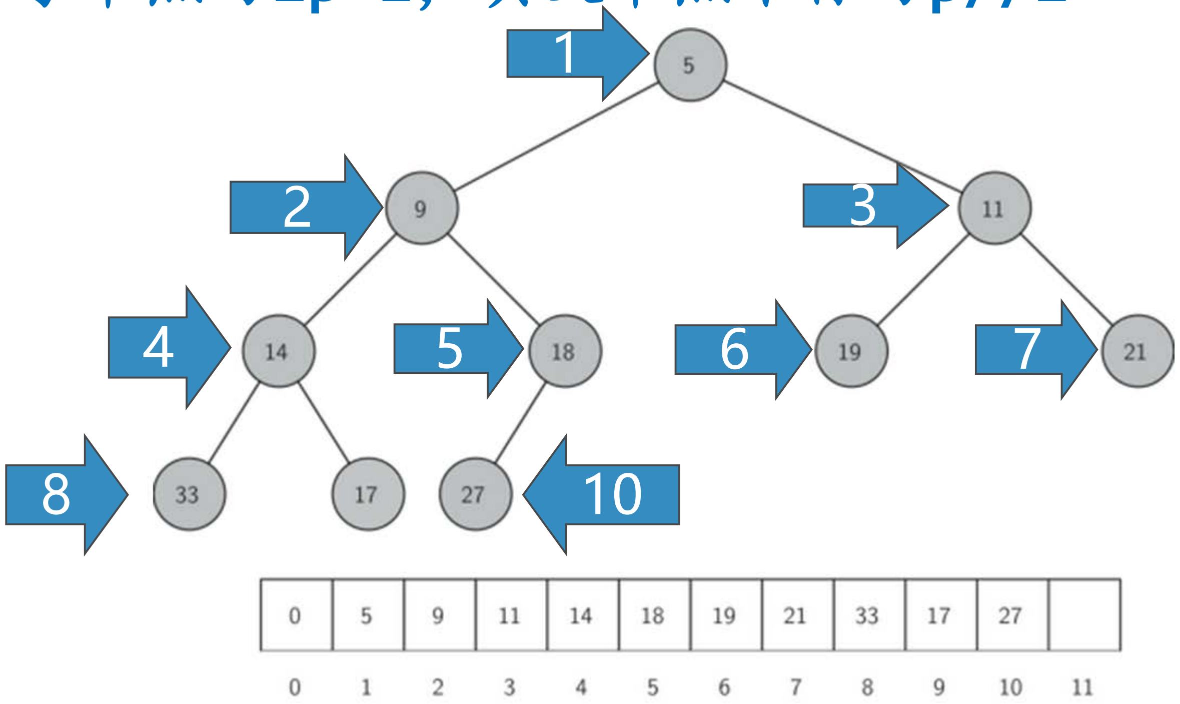
  完全二叉树，叶节点最多只出现在最底层和次底层，而且最底层的叶节点都连续集中在最左边，每个内部节点都有两个子节点，最多可有1个节点例外



完全二叉树的列表实现及性质

❖ **完全二叉树由于其特殊性，可以用非嵌套列表，以简单的方式实现，具有很好性质**

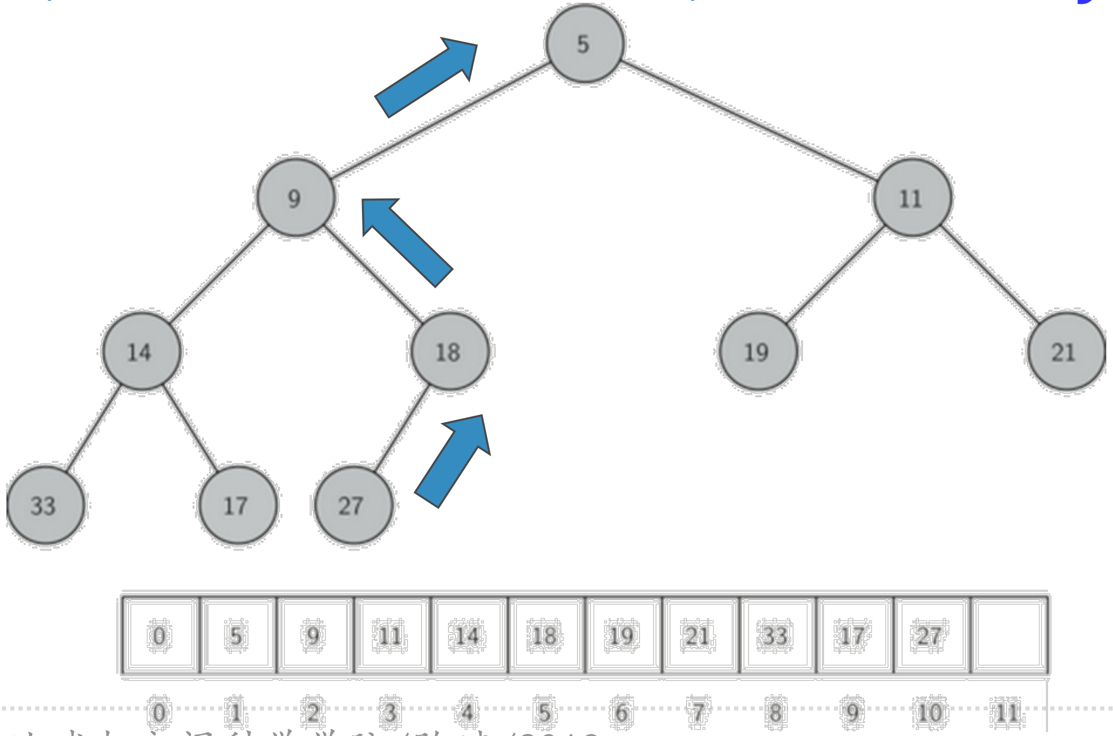
如果节点的下标为p，那么其左子节点下标为2p，右子节点为2p+1，其父节点下标为p//2



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



数据结构与算法（Python版）

二叉堆的Python实现

陈斌 北京大学 gischen@pku.edu.cn

二叉堆操作的实现

❖ 二叉堆初始化

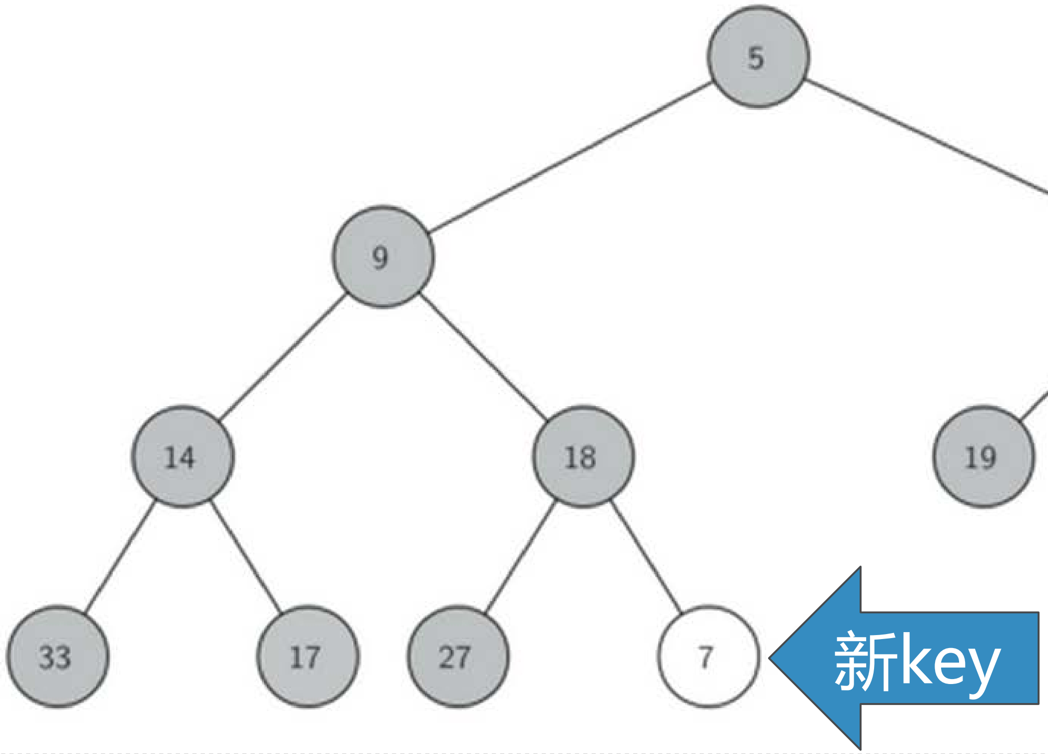
采用一个列表来保存堆数据，其中表首下标为0的项无用，但为了后面代码可以用到简单的整数乘除法，仍保留它。

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

二叉堆操作的实现

❖ 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。会有问题吗？

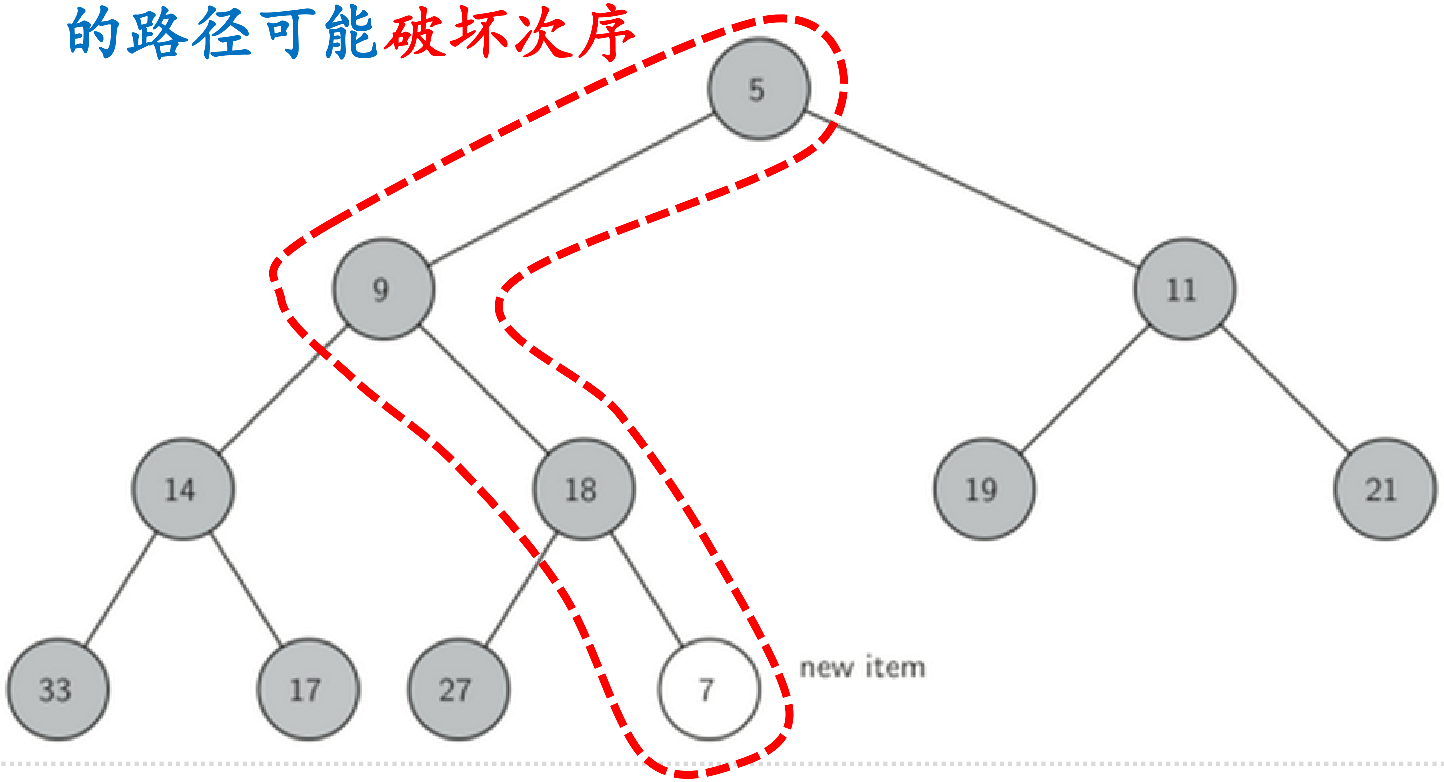


二叉堆操作的实现

❖ insert(key)方法

新key加在列表末尾，显然无法保持“堆”次序

虽然对其它路径的次序没有影响，但对于其到根的路径可能破坏次序

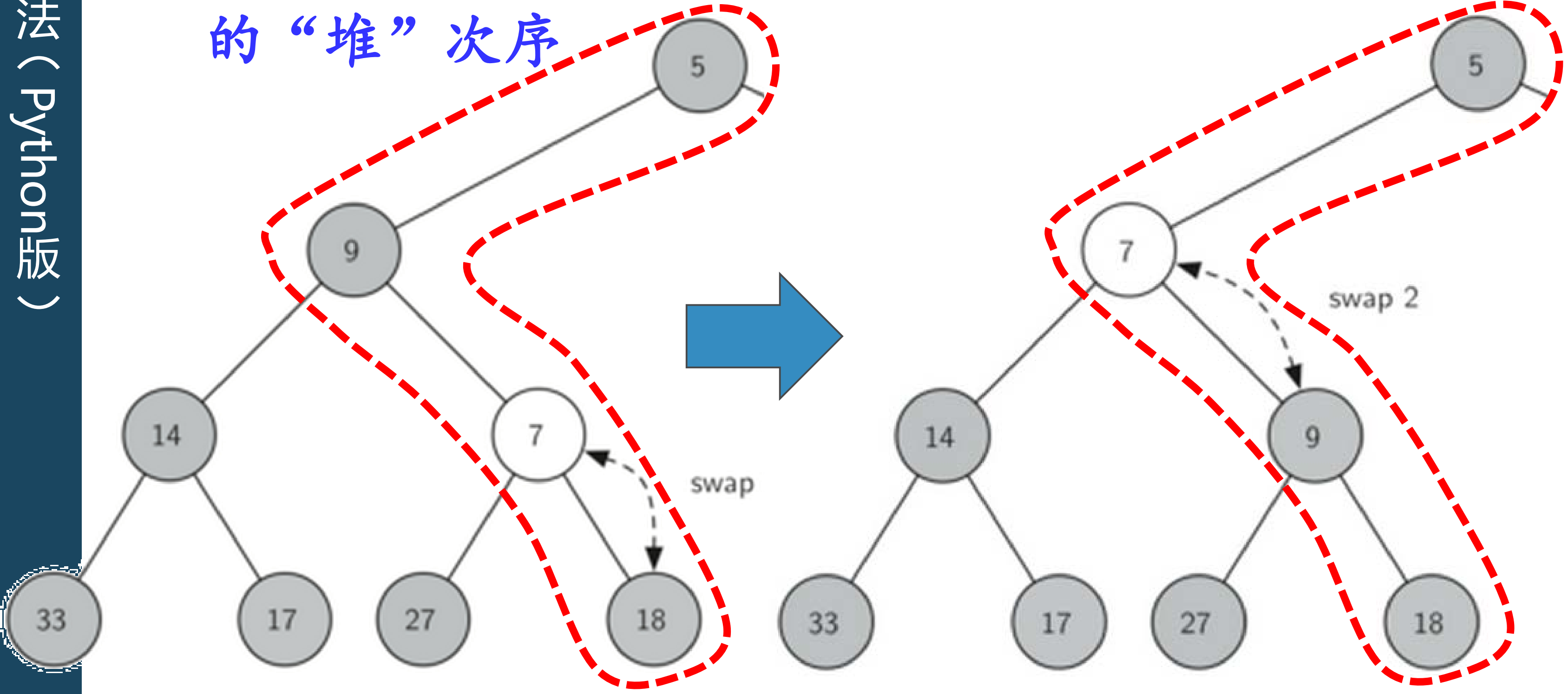


二叉堆操作的实现

❖ **insert(key)方法**

需要将新key沿着路径来“上浮”到其正确位置

注意：新key的“上浮”不会影响其它路径节点的“堆”次序



二叉堆操作的实现：insert代码


```python
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)
```

与父节点交换

沿路径向上

添加到末尾

新key上浮

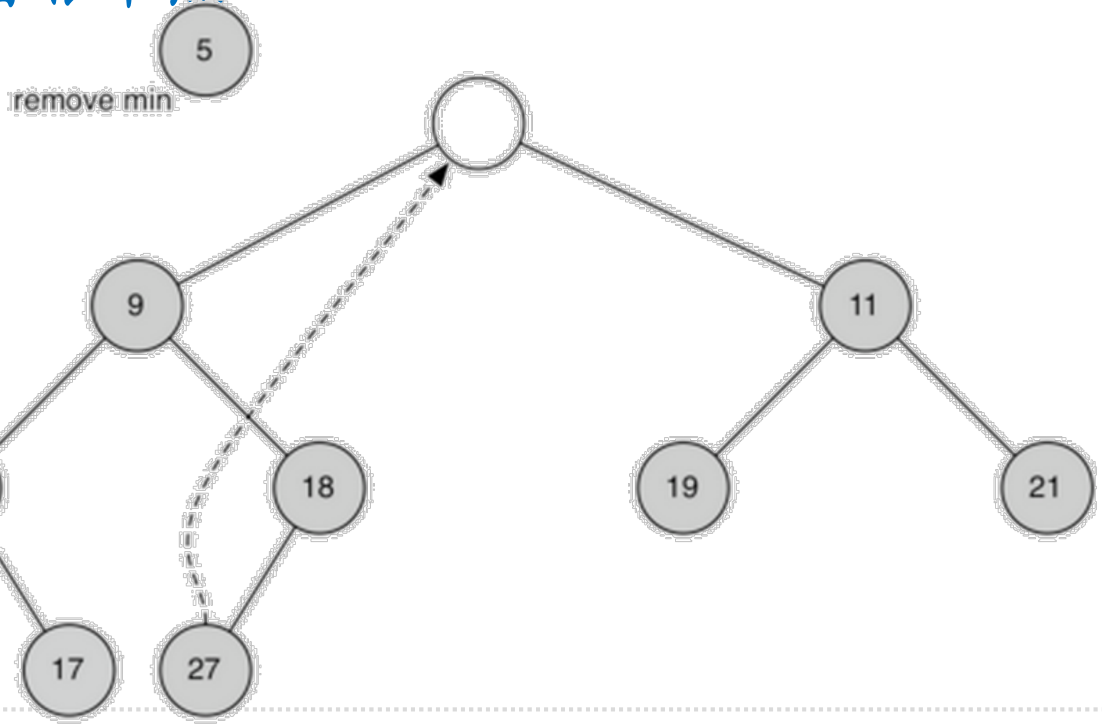
二叉堆操作的实现

❖ delMin()方法

移走整个堆中最小的key：根节点heapList[1]

为了保持“完全二叉树”的性质，只用最后一个节点来代替根节点

问题？

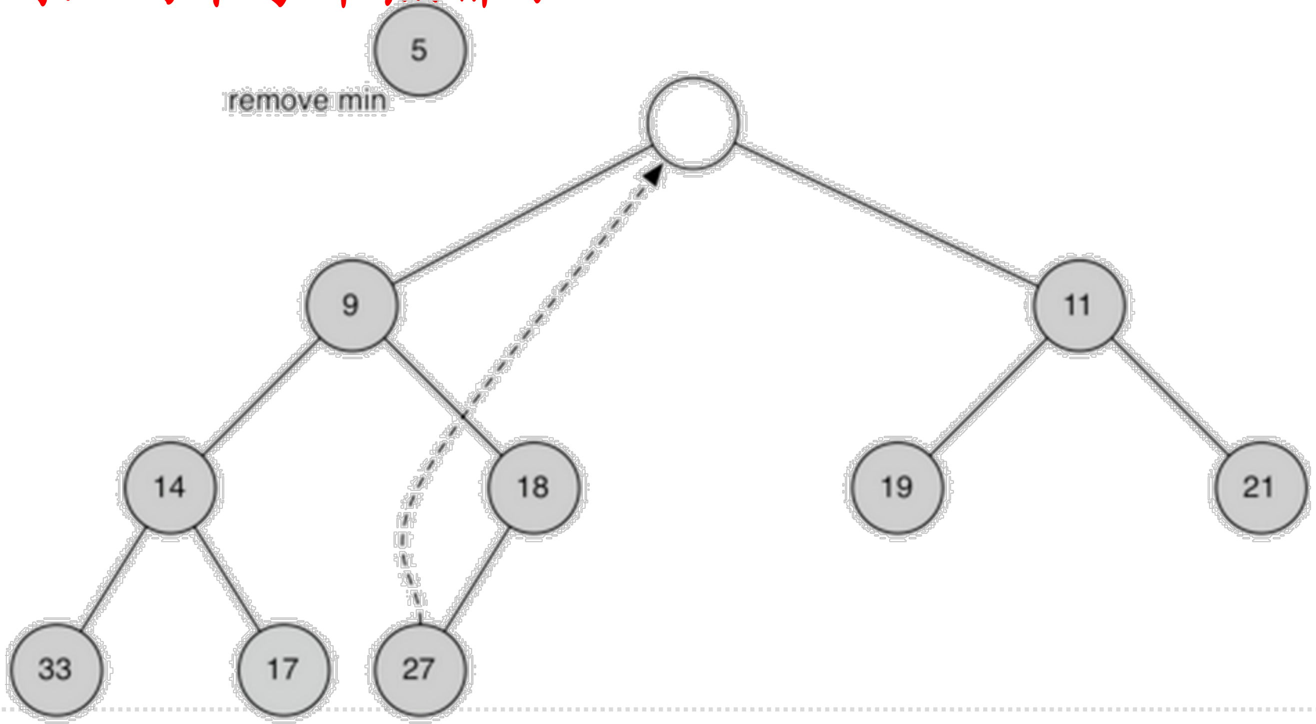


二叉堆操作的实现

❖ delMin()方法

同样，这么简单的替换，还是破坏了“堆”次序

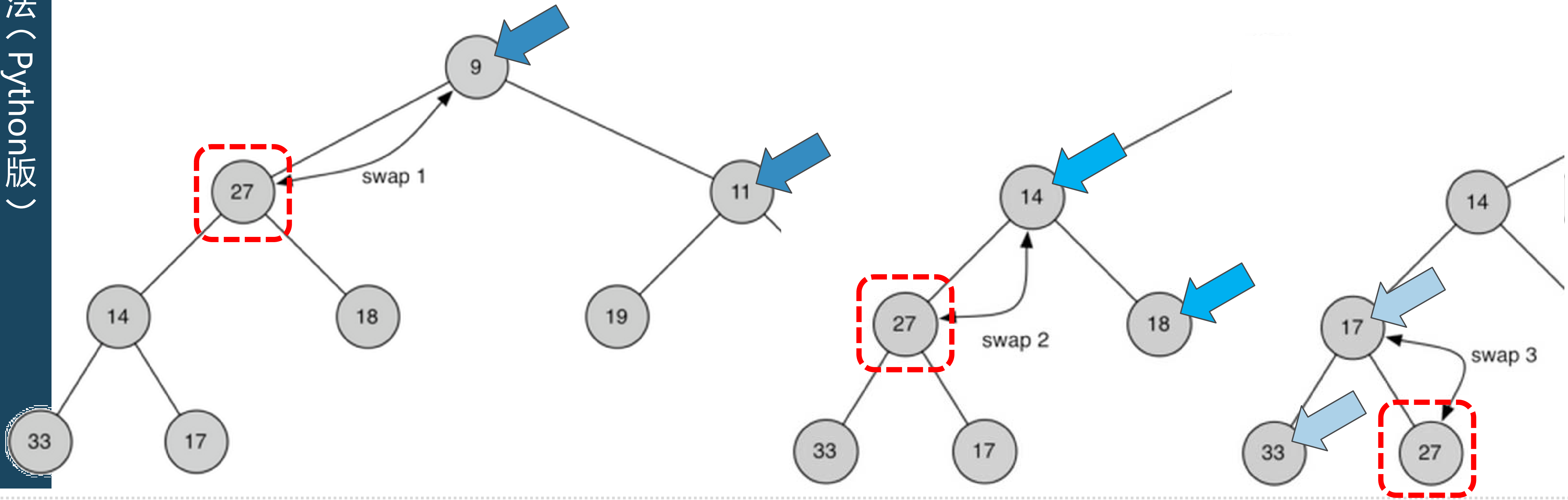
解决方法：将新的根节点沿着一条路径“下沉”，直到比两个子节点都小



二叉堆操作的实现

❖ delMin()方法

“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



```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc

def minChild(self,i):
    if i * 2 + 1 > self.currentSize:
        return i * 2
    else:
        if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
            return i * 2
        else:
            return i * 2 + 1

def delMin(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.currentSize]
    self.currentSize = self.currentSize - 1
    self.heapList.pop()
    self.percDown(1)
    return retval
```

交换下沉

沿路径向下

唯一子节点

返回较小的

移走堆顶

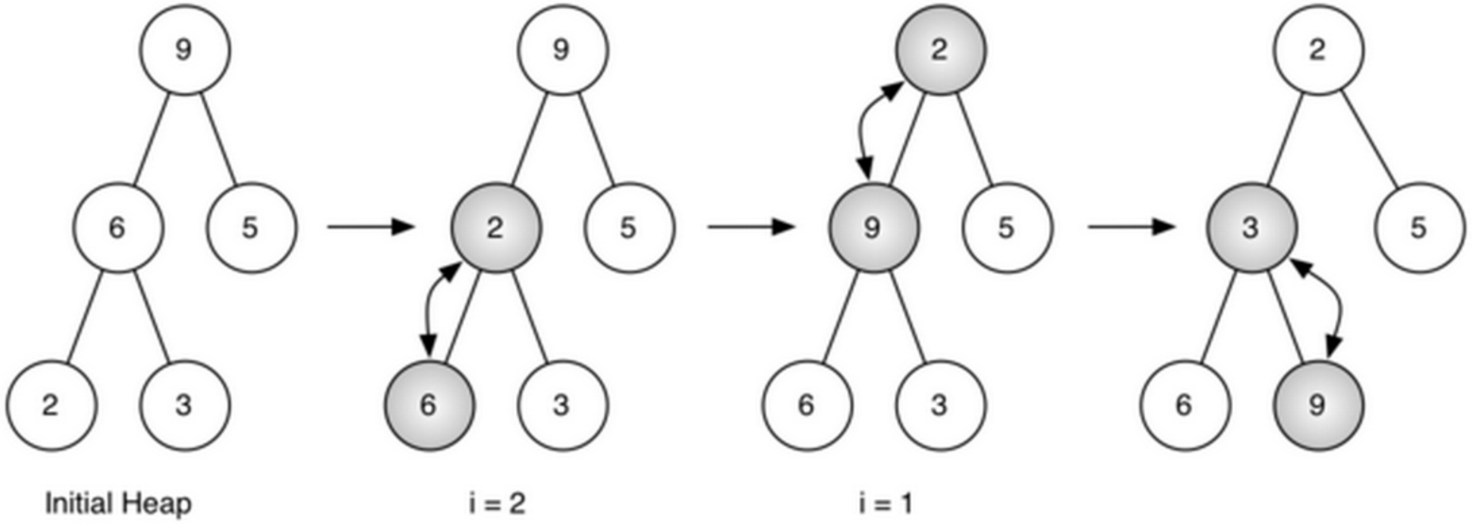
新顶下沉

二叉堆操作的实现

❖ **buildHeap(lst)方法：从无序表生成“堆”**

我们最自然的想法是：用insert(key)方法，将无序表中的数据项逐个insert到堆中，但这么做的总代价是O(nlog n)

其实，用“下沉”法，能够将总代价控制在O(n)



二叉堆操作的实现

❖ **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

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

二叉堆操作的实现

- ❖ **思考：利用二叉堆来进行排序？**

  “堆排序”算法：O(nlog n)

数据结构与算法（Python版）

二叉查找树及操作

陈斌 北京大学 gischen@pku.edu.cn

二叉查找树Binary Search Tree

- 在ADT Map的实现方案中，可以采用不同的数据结构和搜索算法来保存和查找Key，前面已经实现了两个方案

  有序表数据结构+二分搜索算法

  散列表数据结构+散列及冲突解决算法

- 下面我们来试试用二叉查找树保存key，实现key的快速搜索

二叉查找树：ADT Map

❖ **复习一下ADT Map的操作：**

Map()：创建一个空映射

put(key，val)：将key-val关联对加入映射中，如果key已经存在，则将val替换旧关联值；

get(key)：给定key，返回关联的数据值，如不存在，则返回None；

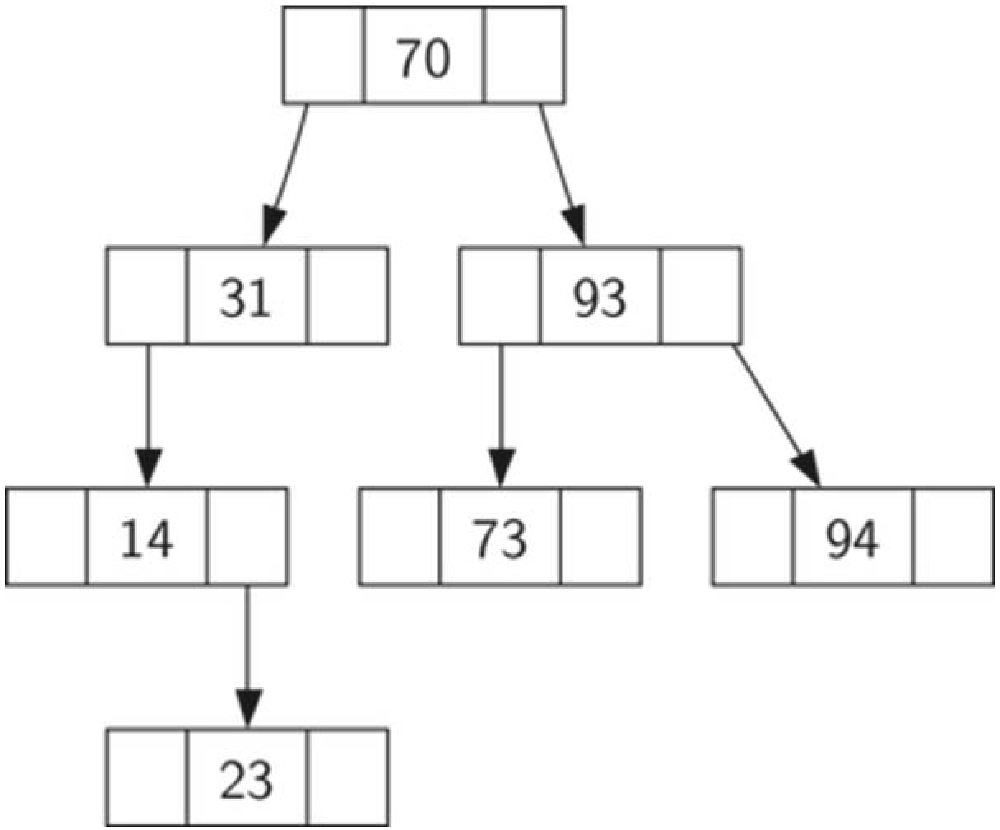
del：通过del map[key]的语句形式删除key-val关联；

len()：返回映射中key-val关联的数目；

in：通过key in map的语句形式，返回key是否存在于关联中，布尔值

二叉查找树BST的性质

❖ 比父节点小的key都出现在左子树，比父节点大的key都出现在右子树。



二叉查找树BST的性质

- 按照70,31,93,94,14,23,73的顺序插入
- 首先插入的70成为**树根**

31比70小，放到左子节点

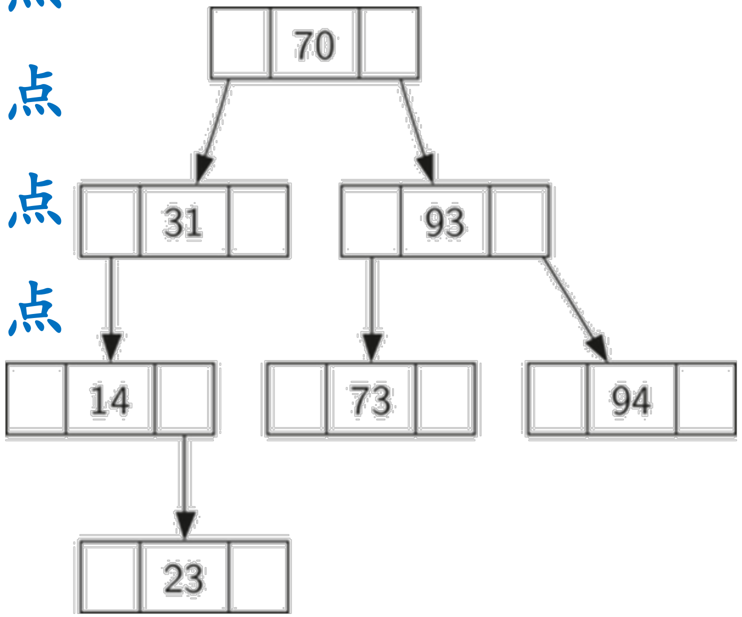
93比70大，放到右子节点

94比93大，放到右子节点

14比31小，放到左子节点

23比14大，放到其右

73比93小，放到其左



- **注意：插入顺序不同，生成的BST也不同**

二叉搜索树的实现：节点和链接结构

- ❖ **需要用到BST和TreeNode两个类，BST的root成员引用根节点TreeNode**

```
class BinarySearchTree:

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

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

二叉搜索树的实现：TreeNode类


```python
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
```

键值
数据项
左、右子节点
父节点

二叉搜索树的实现：TreeNode类


```python
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
```

数据结构与算法（Python版）

二叉查找树实现及算法分析（上）

陈斌 北京大学 gischen@pku.edu.cn

二叉搜索树的实现：BST.put方法

❖ **put(key, val)方法：插入key构造BST**

首先看BST是否为空，如果一个节点都没有，那么key成为根节点root

否则，就调用一个递归函数_put(key, val, root)来放置key

```
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
```

二叉搜索树的实现：_put辅助方法

❖ _put(key, val, currentNode)的流程

如果key比currentNode小，那么_put到左子树

- 但如果没有左子树，那么key就成为左子节点

如果key比currentNode大，那么_put到右子树

- 但如果没有右子树，那么key就成为右子节点

```
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)
```

递归左子树

递归右子树

二叉搜索树的实现：索引赋值

- 随手把__setitem__做了
- 特殊方法（前后双下划线）

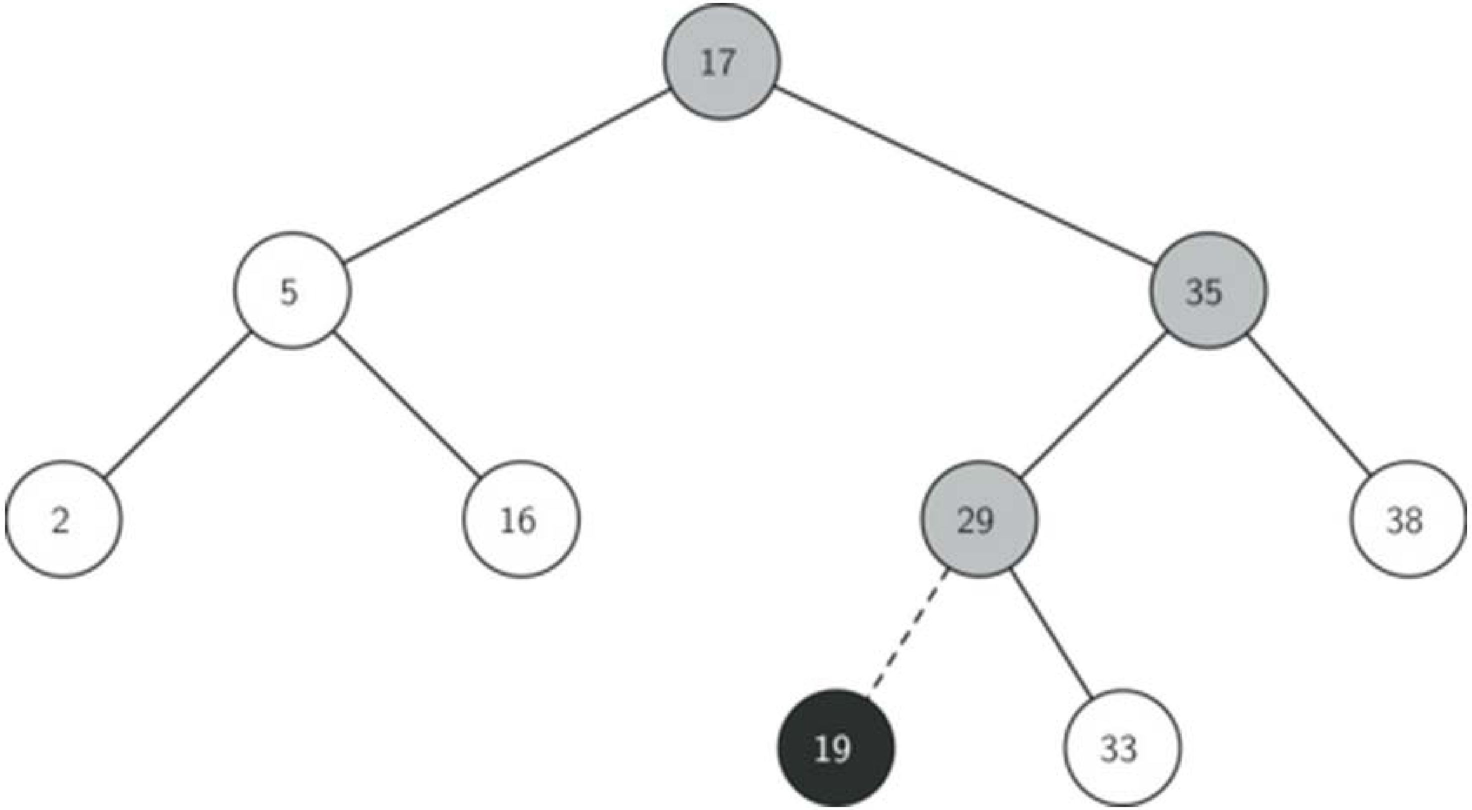
  可以myZipTree['PKU'] = 100871

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

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

二叉搜索树的实现：BST.put图示

- 插入key=19，currentNode的变化过程（灰色）：



二叉搜索树的实现：BST.get方法

❖ 在树中找到key所在的节点取到payload

```
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)
```

递归函数

找到节点

二叉搜索树的实现：索引和归属判断

- __getitem__特殊方法
  实现val= myZipTree['PKU']
- __contains__特殊方法
  实现'PKU' in myZipTree的归属判断运算符in

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

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

```
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"

print(3 in mytree)
print(mytree[6])
```

二叉搜索树的实现：迭代器

- ❖ **我们可以用for循环枚举字典中的所有key**

  ADT Map也应该实现这样的迭代器功能

- ❖ **特殊方法__iter__可以用来实现for迭代**

  BST类中的__iter__方法直接调用了TreeNode中的同名方法

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

```
print(3 in mytree)
print(mytree[6])
del mytree[3]
print(mytree[2])
for key in mytree:
    print key, mytree[key]
```

迭代循环

二叉搜索树的实现：迭代器

❖ TreeNode类中的__iter__迭代器

迭代器函数中用了for迭代，实际上是递归函数

yield是对每次迭代的返回值

中序遍历的迭代

```python
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
```

数据结构与算法（Python版）

二叉查找树实现及算法分析（下）

陈斌 北京大学 gischen@pku.edu.cn

二叉查找树的实现：BST.delete方法

❖ **有增就有减，最复杂的delete方法：**

用_get找到要删除的节点，然后调用remove来删除，找不到则提示错误

```
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')
```

二叉查找树的实现：BST.delete方法

- ❖ __delitem__特殊方法

  实现del myZipTree['PKU']这样的语句操作

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

- ❖ 在delete中，最复杂的是找到key对应的节点之后的**remove**节点方法！

二叉查找树的实现：BST.remove方法

- 从BST中remove一个节点，还要求仍然保持BST的性质，分以下3种情形：

  这个节点没有子节点

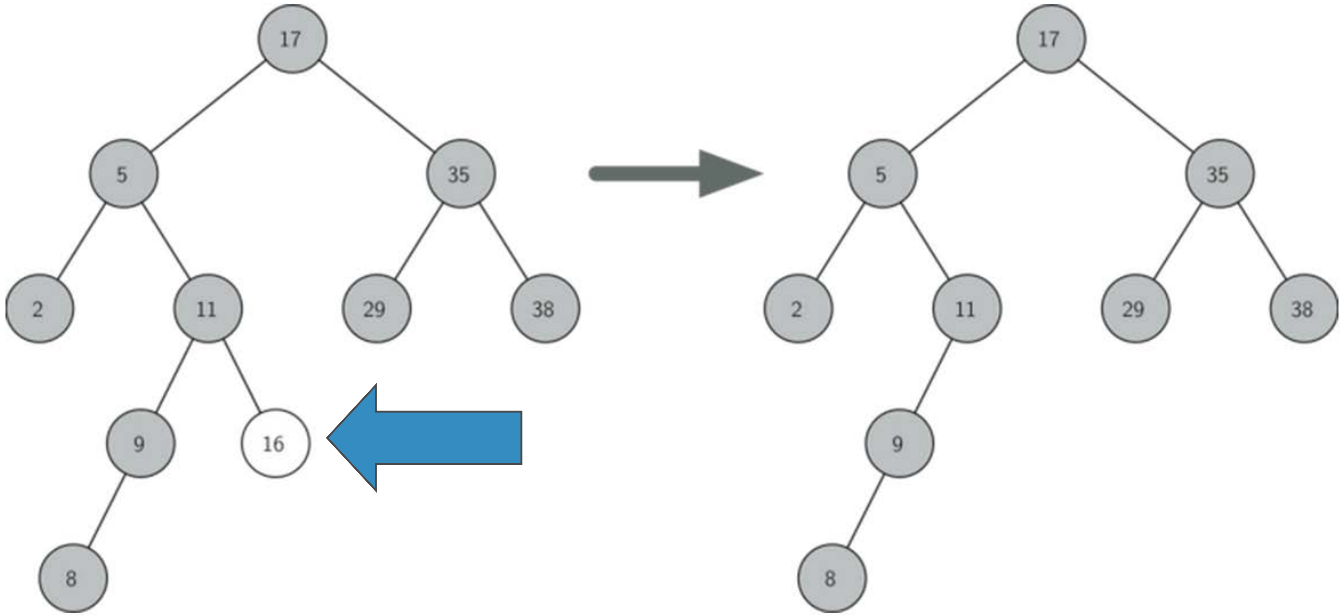
  这个节点有1个子节点

  这个节点有2个子节点

二叉查找树的实现：BST.remove方法

❖ 没有子节点的情况好办，直接删除

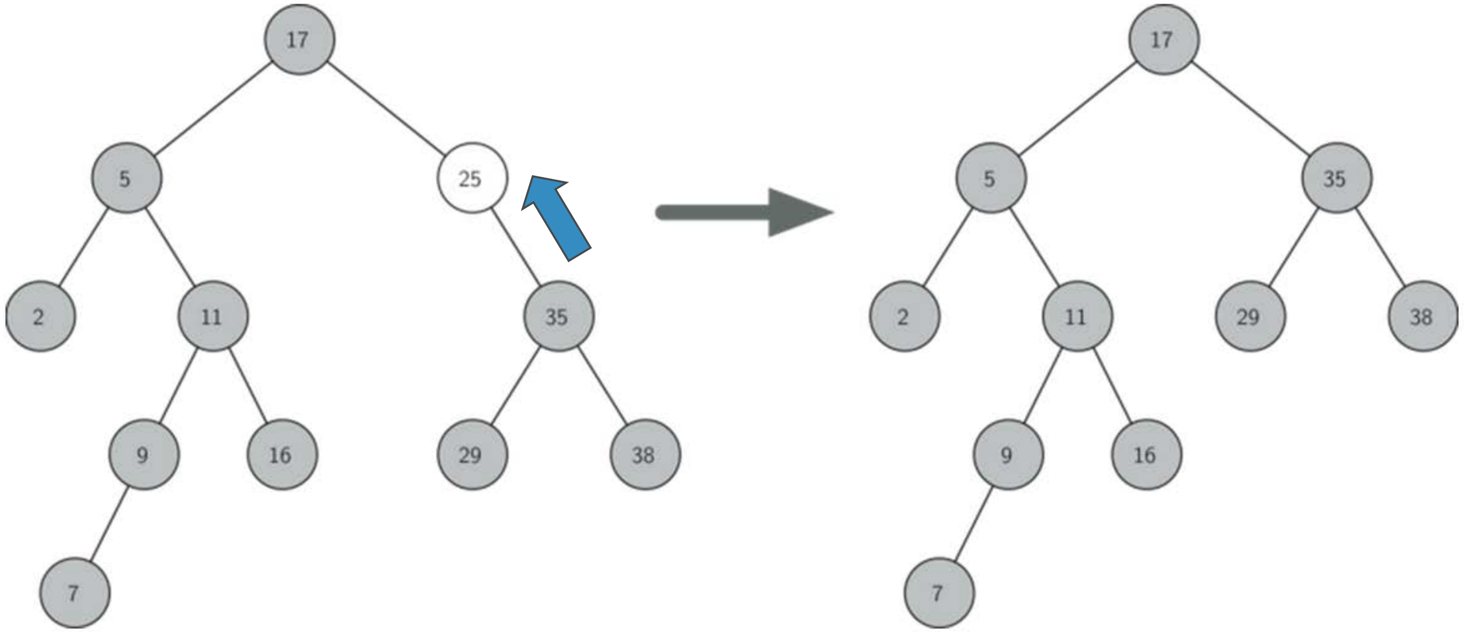
```
if currentNode.isLeaf(): #leaf
    if currentNode == currentNode.parent.leftChild:
        currentNode.parent.leftChild = None
    else:
        currentNode.parent.rightChild = None
```



二叉查找树的实现：BST.remove方法

❖ **第2种情形稍复杂：被删节点有1个子节点**

解决：将这个唯一的子节点上移，替换掉被删节点的位置



二叉查找树的实现：BST.remove方法

- **但替换操作需要区分几种情况：**

  被删节点的子节点是左？还是右子节点？

  被删节点本身是其父节点的左？还是右子节点？

  被删节点本身就是根节点？

```
else: # this node has one child
  if currentNode.hasLeftChild():
    if currentNode.isLeftChild():
```

左子节点删除

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

右子节点删除

```
      currentNode.leftChild.parent = currentNode.parent
      currentNode.parent.rightChild = currentNode.leftChild
    else:
```

根节点删除

```
      currentNode.replaceNodeData(currentNode.leftChild.key,
                                  currentNode.leftChild.payload,
                                  currentNode.leftChild.leftChild,
                                  currentNode.leftChild.rightChild)
  else:
    if currentNode.isLeftChild():
```

左子节点删除

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

右子节点删除

```
      currentNode.rightChild.parent = currentNode.parent
      currentNode.parent.rightChild = currentNode.rightChild
    else:
```

根节点删除

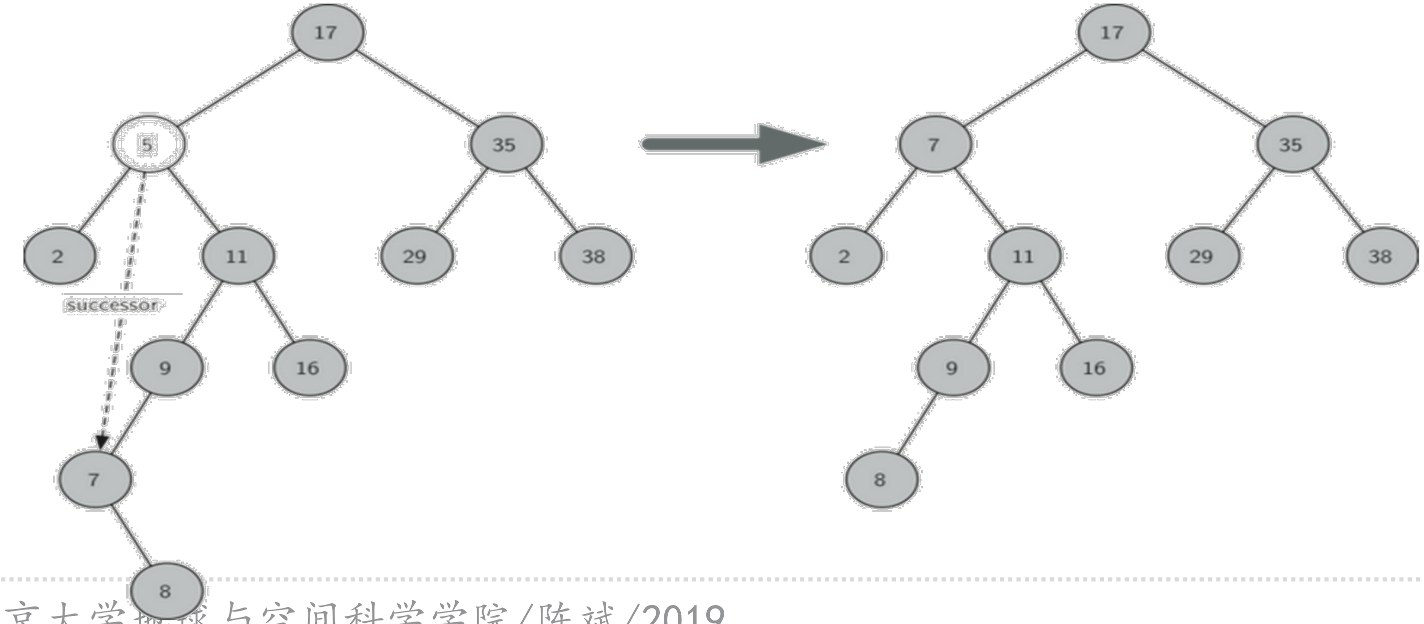
```
      currentNode.replaceNodeData(currentNode.rightChild.key,
                                  currentNode.rightChild.payload,
                                  currentNode.rightChild.leftChild,
                                  currentNode.rightChild.rightChild)
```

二叉查找树的实现：BST.remove方法

❖ 第3种情形最复杂：被删节点有2个子节点

这时无法简单地将某个子节点上移替换被删节点

但可以找到**另一个合适的节点**来替换被删节点，这个合适节点就是被删节点的**下一个key值节点**，即被删节点**右子树中最小的那个**，称为“**后继**”

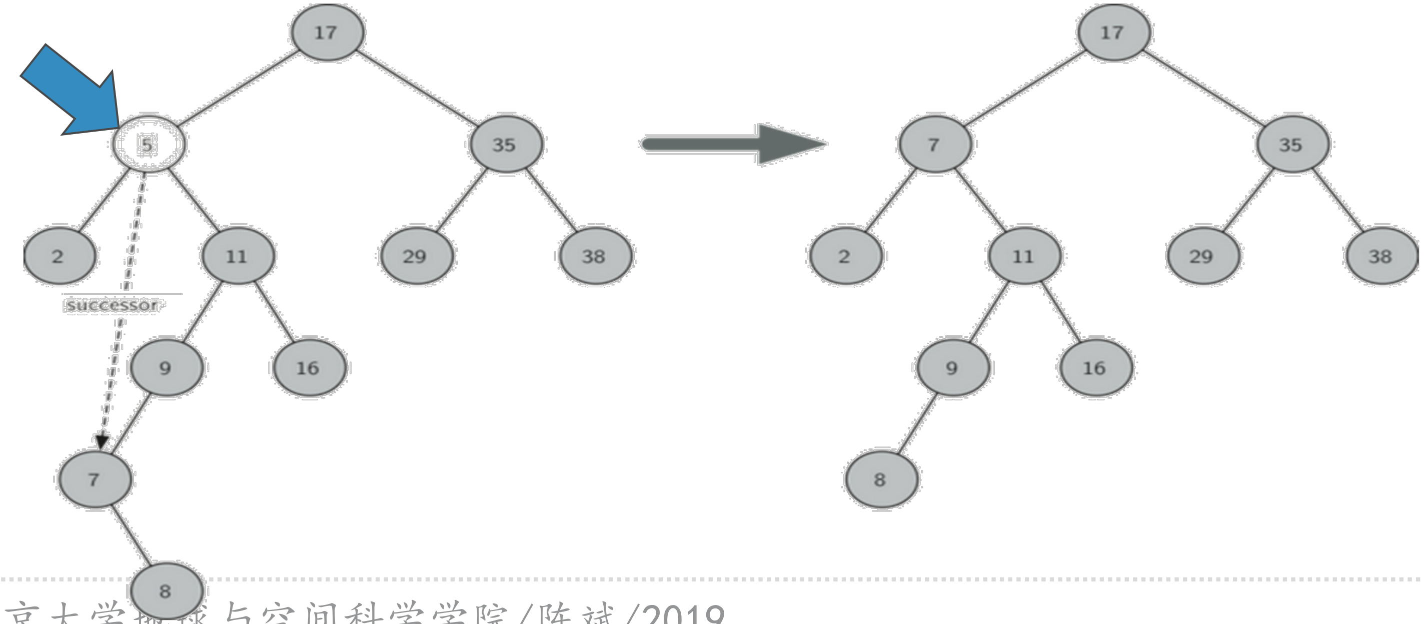


二叉查找树的实现：BST.remove方法

- ❖ **第3种情形最复杂：被删节点有2个子节点**

  可以肯定这个后继节点**最多只有1个**子节点（本身是叶节点，或仅有右子树）

  将这个后继节点摘出来（也就是删除了），替换掉被删节点。



二叉查找树的实现：BST.remove方法

- BinarySearchTree类：remove方法（情形3）

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

二叉查找树的实现：BST.remove方法

❖ TreeNode类：寻找后继节点

```python
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
```

目前不会遇到

到左下角

二叉查找树的实现：BST.remove方法

❖ TreeNode类：摘出节点spliceOut()

```
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
```

摘出叶节点

目前不会遇到

摘出带右子节点的节点

二叉查找树：算法分析（以put为例）

- 其性能决定因素在于二叉搜索树的**高度**（最大层次），而其高度又受数据项key插入**顺序**的影响。
- **如果key的列表是随机分布的话**，那么大于和小于根节点key的键值大致相等
- BST的高度就是$\log_2 n$（n是节点的个数），而且，这样的树就是平衡树
- put方法最差性能为$O(\log_2 n)$。

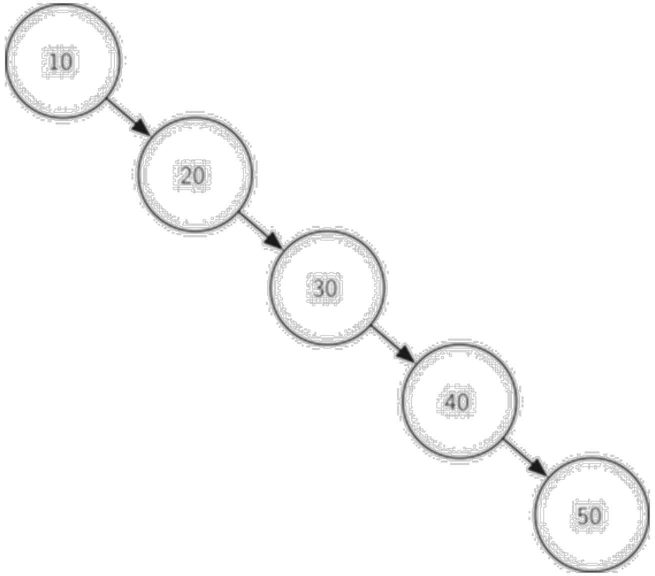
二叉查找树：算法分析（以put为例）

- 但key列表分布**极端情况**就完全不同

  按照从小到大顺序插入的话，如下图

  这时候put方法的性能为O(n)

  其它方法也是类似情况



- **如何改进BST？不受key插入顺序影响？**

数据结构与算法（Python版）

AVL树的定义和性能

陈斌 北京大学 gischen@pku.edu.cn

平衡二叉查找树：AVL树的定义

- 我们来看看能够在key插入时一直保持平衡的二叉查找树：AVL树

  AVL是发明者的名字缩写：G.M. Adelson-Velskii and E.M. Landis

- 利用AVL树实现ADT Map，基本上与BST的实现相同
- 不同之处仅在于二叉树的生成与维护过程

平衡二叉查找树：AVL树的定义

- AVL树的实现中，需要对每个节点跟踪"**平衡因子**balance factor"参数
- 平衡因子是根据节点的左右子树的高度来定义的，确切地说，是**左右子树高度差**：

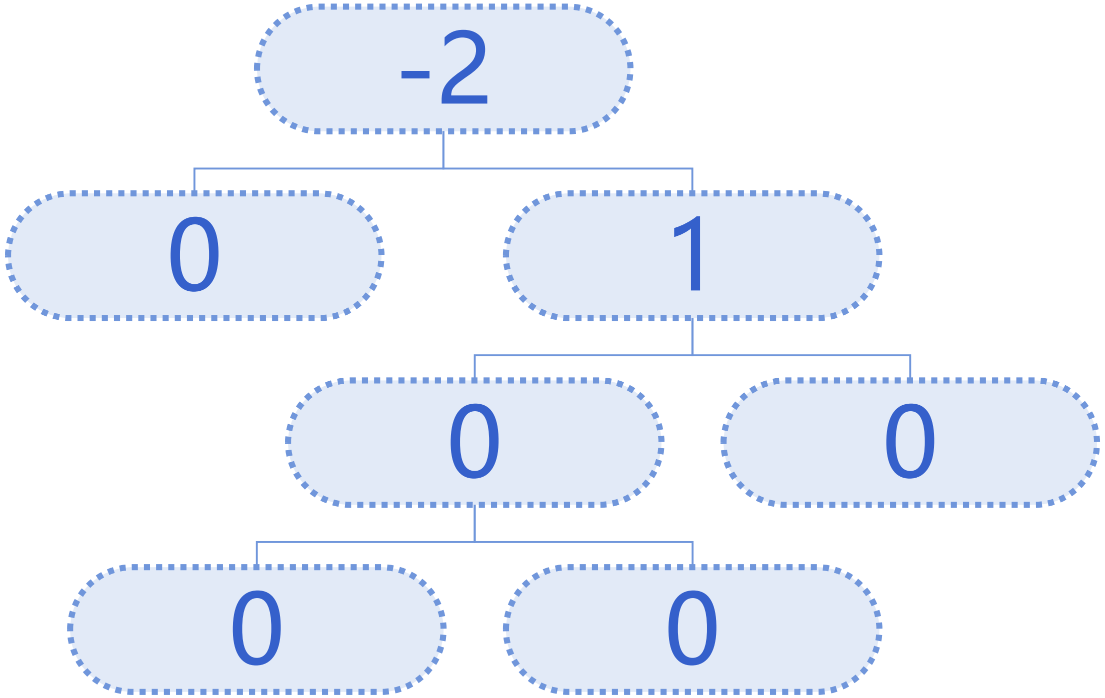
  balanceFactor = height(leftSubTree) - height(rightSubTree)

  如果平衡因子大于0，称为"**左重**left-heavy"，小于零称为"**右重**right-heavy"

  平衡因子等于0，则称作平衡。

平衡二叉查找树：平衡因子

- 如果一个二叉查找树中每个节点的平衡因子都在-1，0，1之间，则把这个二叉搜索树称为**平衡树**

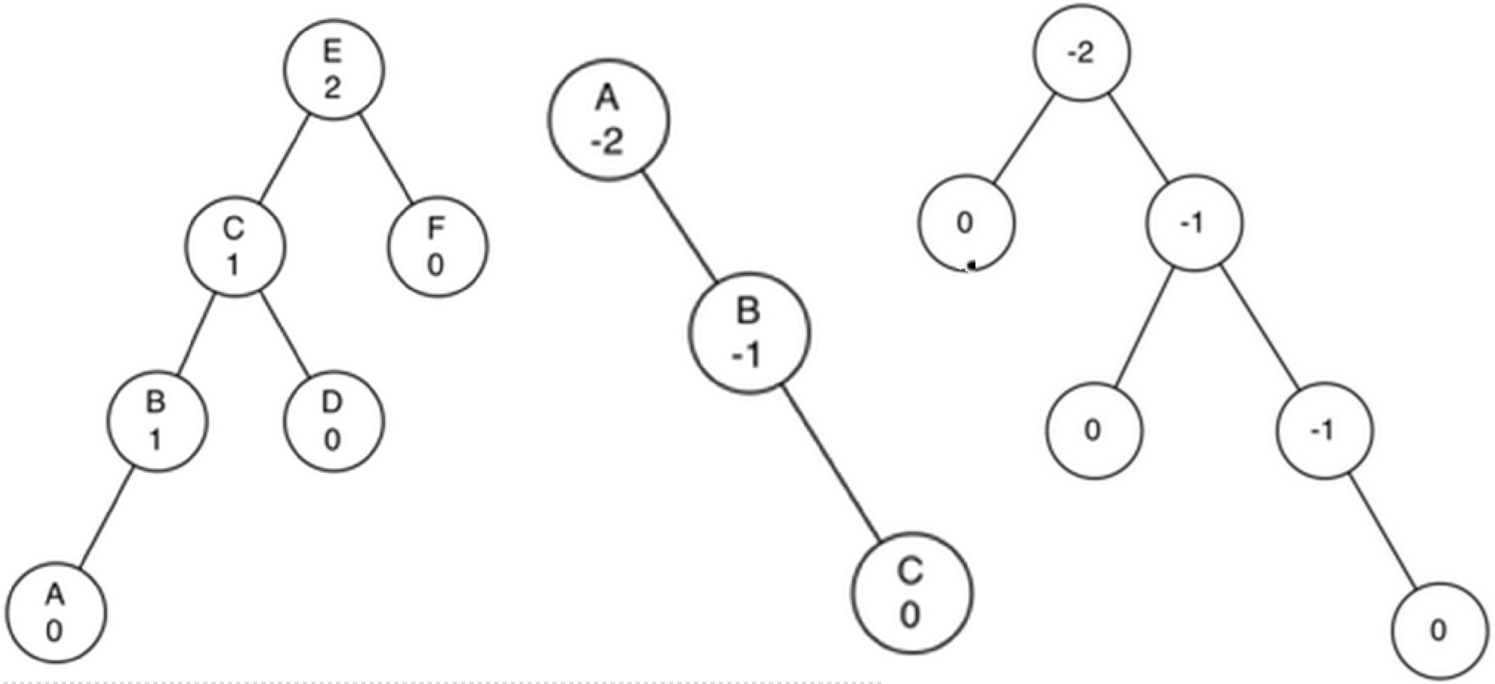


平衡二叉查找树：AVL树的定义

- 在平衡树操作过程中，有节点的平衡因子超出此范围，则需要一个重新平衡的过程

要保持BST的性质！

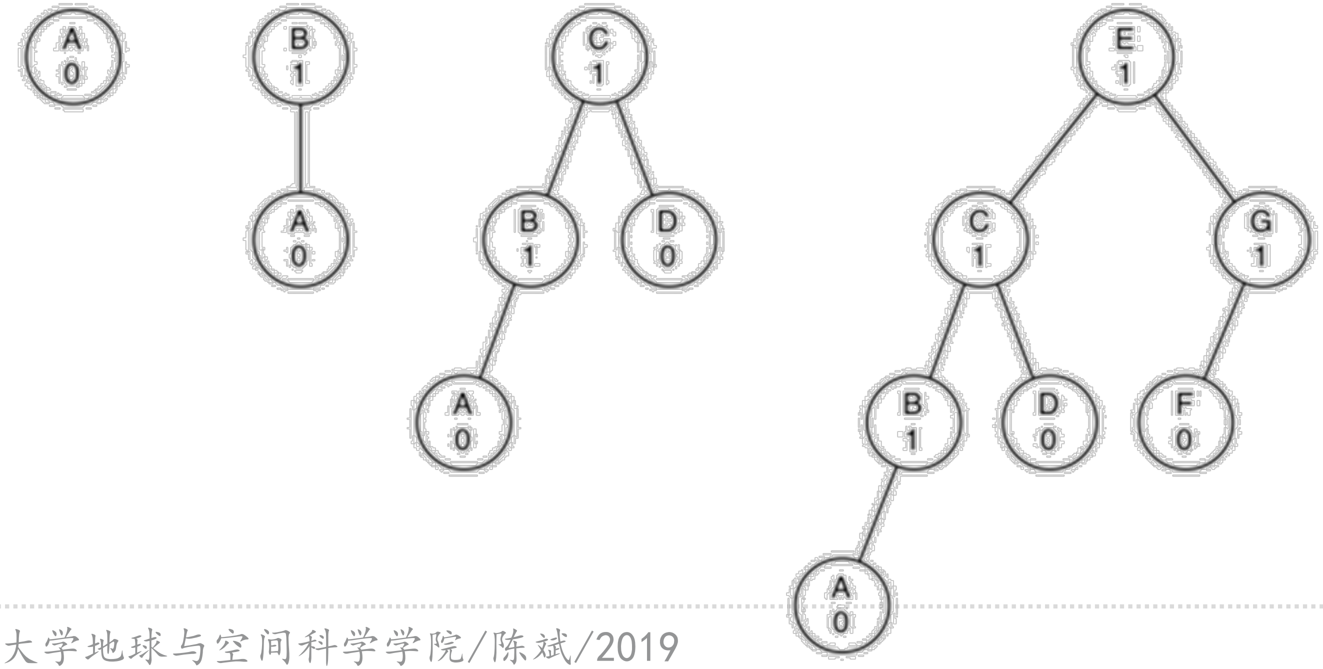
思考：如果重新平衡，应该变成什么样？



先来看看AVL树的性能

❖ **我们来分析AVL树最差情形下的性能：即平衡因子为1或者-1**

下图列出平衡因子为1的“左重”AVL树，树的高度从1开始，来看看问题规模（总节点数N）和比对次数（树的高度h）之间的关系如何？



AVL树性能分析

- ❖ **观察上图h=1～4时，总节点数N的变化**

  h= 1, N= 1

  h= 2, N= 2= 1+ 1

  h= 3, N= 4= 1+ 1+ 2

  h= 4, N= 7= 1+ 2+ 4

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

- ❖ **观察这个通式，很接近斐波那契数列！**

AVL树性能分析

❖ 定义斐波那契数列F_i

利用F_i重写N_h

$$F_0 = 0 \qquad N_h = 1 + N_{h-1} + N_{h-2}$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \text{ for all } i \geq 2 \qquad N_h = F_{h+2} - 1, h \geq 1$$

❖ 斐波那契数列的性质：F_i/F_{i-1}趋向于黄金分割Φ

可以写出F_i的通式

$$\Phi = \frac{1+\sqrt{5}}{2} \qquad F_i = \Phi^i/\sqrt{5}$$

AVL树性能分析

- 将F_i通式代入到N_h中，得到N_h的通式

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

- 上述通式只有N和h了，我们解出h

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

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

$$h = 1.44\log N_h$$

- 最多搜索次数h和规模N的关系，可以说AVL树的搜索时间复杂度为O(log n)

数据结构与算法（Python版）

AVL树的Python实现

陈斌 北京大学 gischen@pku.edu.cn

AVL树的Python实现

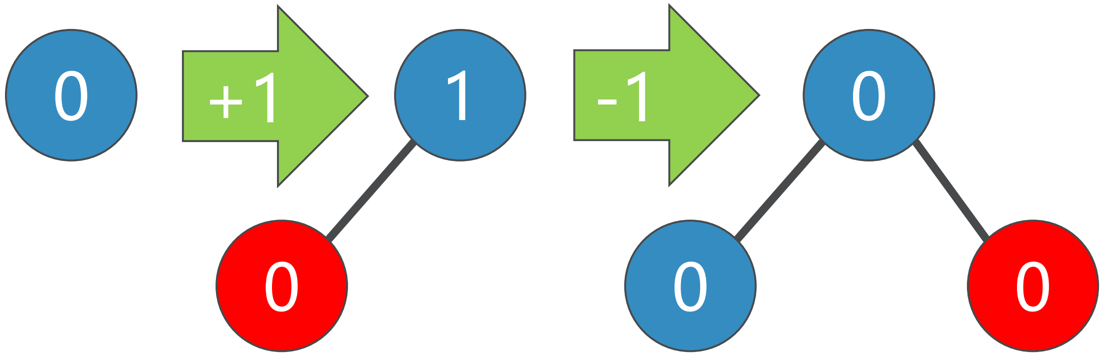
- ❖ 既然AVL平衡树确实能够改进BST树的性能，避免退化情形
- ❖ 我们来看看向AVL树插入一个新key，如何才能保持AVL树的平衡性质
- ❖ 首先，作为BST，新key必定以**叶节点**形式插入到AVL树中

AVL树的Python实现

- ❖ **叶节点的平衡因子是0，其本身无需重新平衡**
- ❖ **但会影响其父节点的平衡因子：**

  作为左子节点插入，则父节点平衡因子会增加1；

  作为右子节点插入，则父节点平衡因子会减少1。



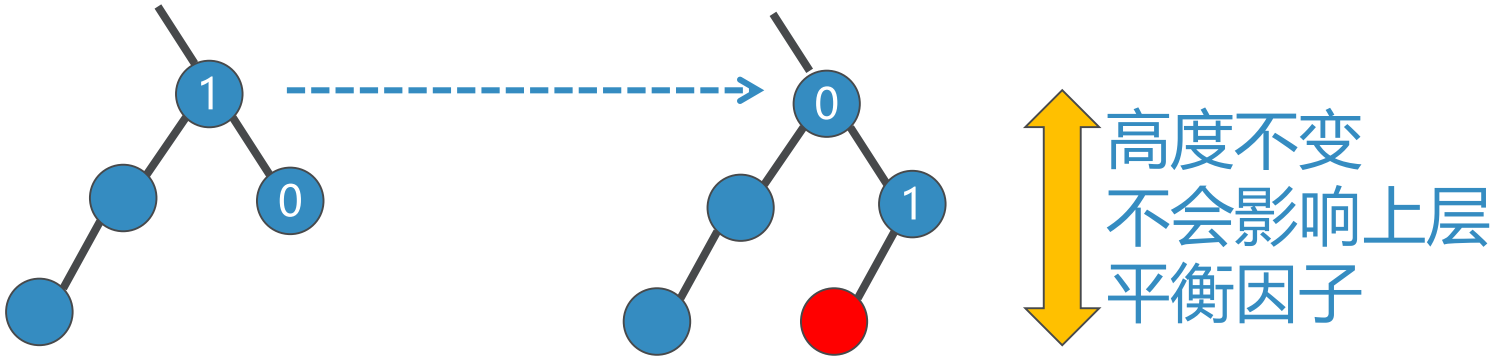
AVL树的Python实现

- **这种影响可能随着其父节点到根节点的路径一直传递上去，直到：**

  传递到根节点为止；

  或者某个父节点平衡因子被调整到0，不再影响上层节点的平衡因子为止。

  - （无论从-1或者1调整到0，都不会改变子树高度）



AVL树的实现：put方法

❖ 重新定义_put方法即可

```
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)
```

调整因子

调整因子

AVL树的实现：UpdateBalance方法

```python
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)
```

重新平衡

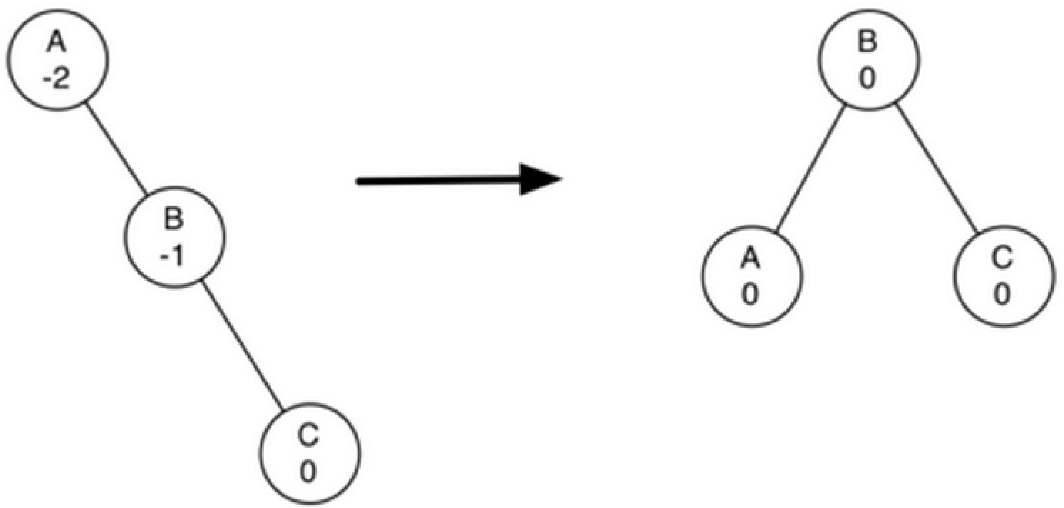
调整父节点因子

AVL树的实现：rebalance重新平衡

- **主要手段：将不平衡的子树进行旋转rotation**

视“左重”或者“右重”进行不同方向的旋转

同时更新相关父节点引用，更新旋转后被影响节点的平衡因子

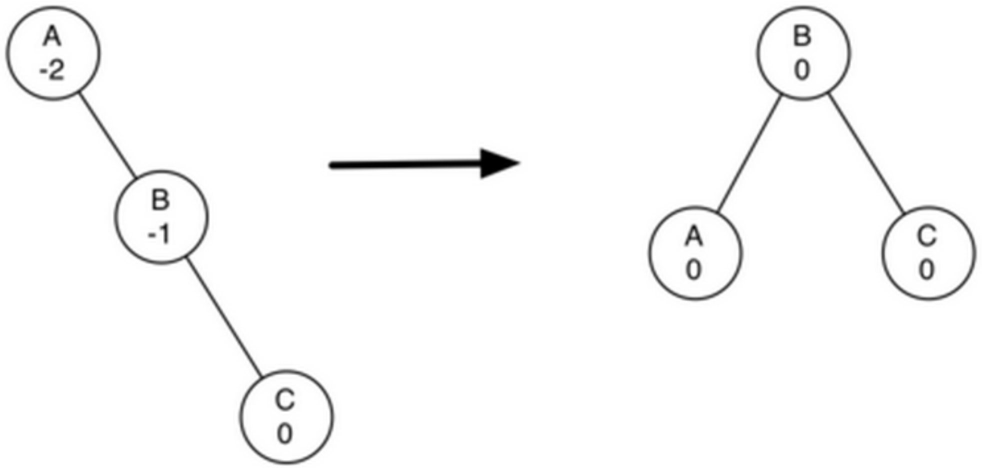


AVL树的实现：rebalance重新平衡

- 如图，是一个“右重”子树A的左旋转（**并保持BST性质**）

将右子节点B提升为子树的根，将旧根节点A作为新根节点B的左子节点

如果新根节点B原来有左子节点，则将此节点设置为A的右子节点（A的右子节点一定有空）



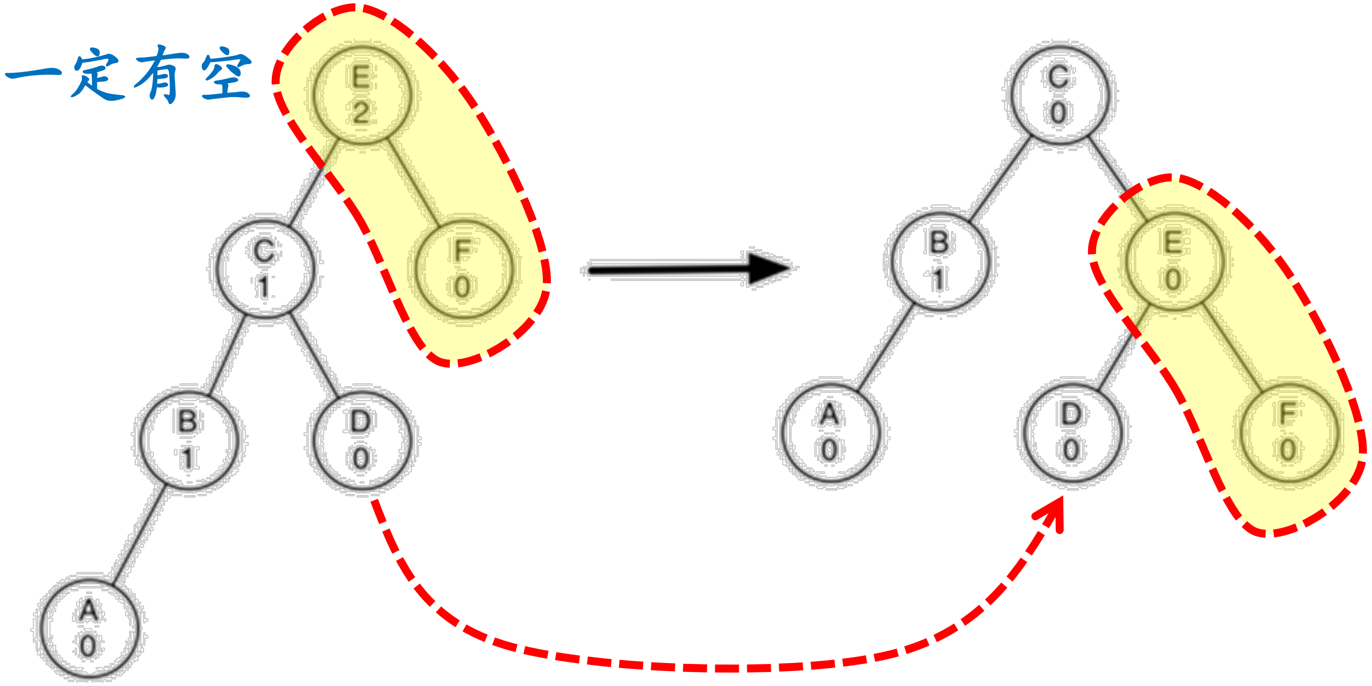
AVL树的实现：rebalance重新平衡

- 更复杂一些的情况：如图的“左重”子树右旋转

旋转后，新根节点将旧根节点作为右子节点，但是新根节点原来已有右子节点，需要将原有的右子节点重新定位！

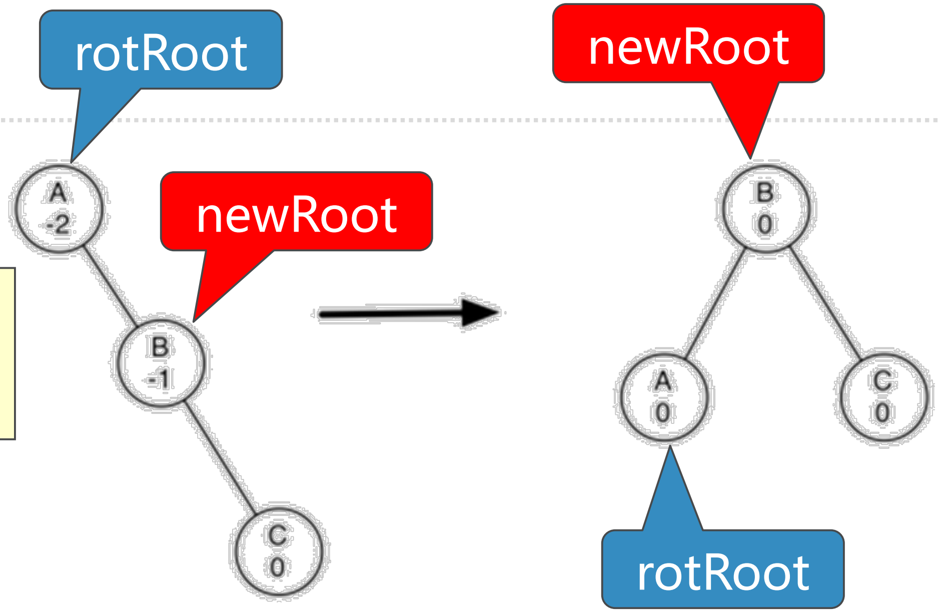
原有的右子节点D改到旧根节点E的左子节点

同样，E的左子节点在旋转后一定有空



AVL树的实现：rotateLeft代码


```python
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)
```



仅有两个节点
需要调整因子

AVL树的实现：如何调整平衡因子

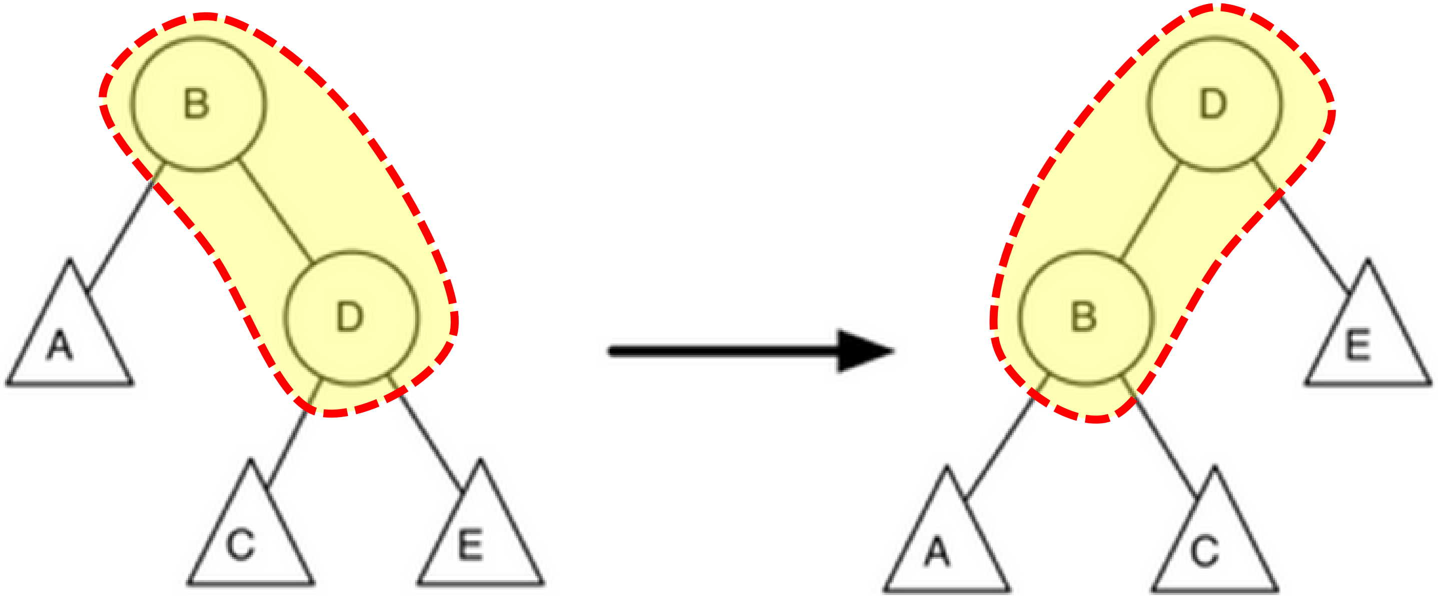
❖ 看看左旋转对平衡因子的影响

保持了次序ABCDE

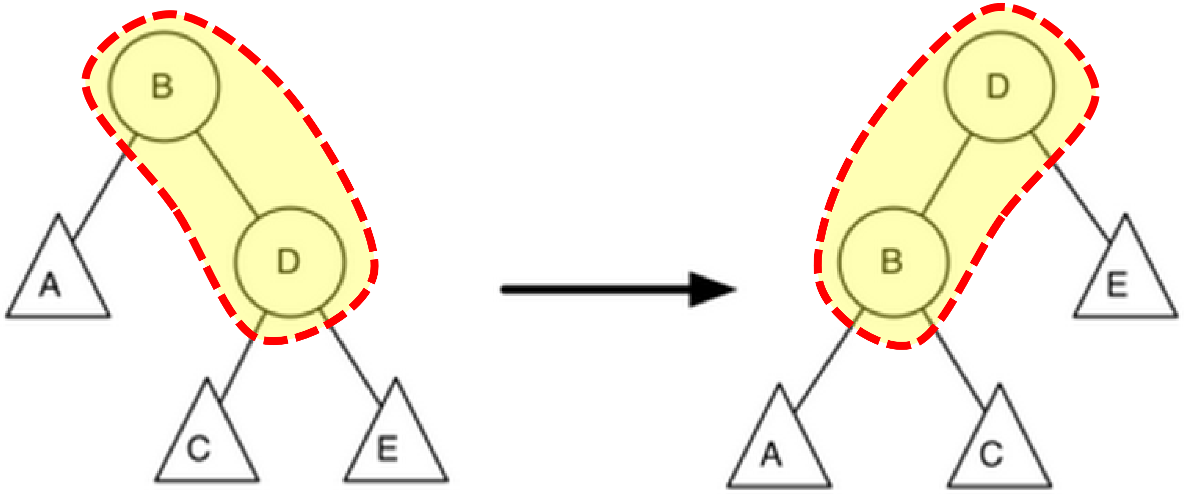
ACE的平衡因子不变

- hA/hC/hE不变

主要看BD新旧关系



AVL树的实现：如何调整平衡因子



❖ 我们来看看B的变化

新B= hA- hC

旧B= hA- 旧hD

而：

旧hD= 1+ max(hC, hE)，所以旧B= hA- (1+ max(hC, hE))

新B- 旧B= 1+ max(hC, hE)- hC

新B= 旧B+ 1+ max(hC, hE)- hC；把hC移进max函数里就有

新B= 旧B+ 1+ max(0, -旧D) <==> 新B= 旧B+ 1- min(0, 旧D)

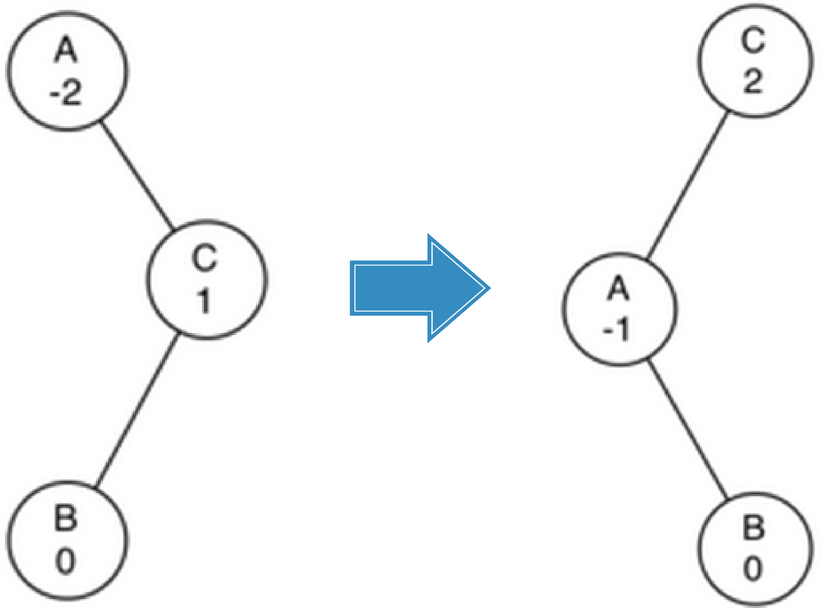
```
rotRoot.balanceFactor = rotRoot.balanceFactor + \
                        1 - min(newRoot.balanceFactor, 0)
```

AVL树的实现：更复杂的情形

- 下图的“右重”子树，单纯的左旋转无法实现平衡

左旋转后变成“左重”了

“左重”再右旋转，还回到“右重”



AVL树的实现：更复杂的情形

- ❖ **所以，在左旋转之前检查右子节点的因子**

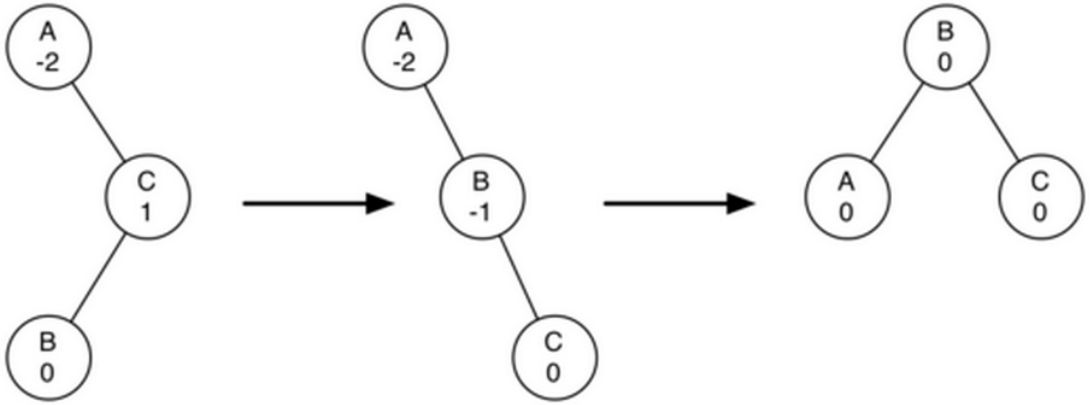
  如果右子节点“左重”的话，先对它进行右旋转

  再实施原来的左旋转

- ❖ **同样，在右旋转之前检查左子节点的因子**

  如果左子节点“右重”的话，先对它进行左旋转

  再实施原来的右旋转



AVL树的实现：rebalance代码


```
def rebalance(self,node):
    if node.balanceFactor < 0:
        if node.rightChild.balanceFactor > 0:
            # Do an LR Rotation
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
        else:
            # single left
            self.rotateLeft(node)
    elif node.balanceFactor > 0:
        if node.leftChild.balanceFactor < 0:
            # Do an RL Rotation
            self.rotateLeft(node.leftChild)
            self.rotateRight(node)
        else:
            # single right
            self.rotateRight(node)
```

右重需要左旋

右子节点左重
先右旋

左重需要右旋

左子节点右重
先左旋

AVL树的实现：结语

- **经过复杂的put方法，AVL树始终维持平衡，get方法也始终保持O(log n)高性能**

  不过，put方法的代价有多大？

- **将AVL树的put方法分为两个部分：**

  需要插入的新节点是叶节点，更新其所有父节点和祖先节点的代价最多为O(log n)

  如果插入的新节点引发了不平衡，重新平衡最多需要2次旋转，但旋转的代价与问题规模无关，是常数O(1)

  所以整个put方法的时间复杂度还是O(log n)

数据结构与算法（Python版）

树结构小结

陈斌 北京大学 gischen@pku.edu.cn

本章总结

- 本章介绍了“树”数据结构，我们讨论了如下算法：
- 用于**表达式**解析和求值的二叉树
- 用于实现ADT Map的**二叉查找树**BST树
- 改进了性能，用于实现ADT Map的平衡二叉查找树**AVL树**
- 实现了“最小堆”的完全二叉树：**二叉堆**

ADT Map的实现方法小结

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

	有序表	散列表	二叉查找树	AVL树
put	$O(n)$	$O(1)$ -> $O(n)$	$O(\log_2 n)$ -> $O(n)$	$O(\log_2 n)$
get	$O(\log_2 n)$	$O(1)$ -> $O(n)$	$O(\log_2 n)$ -> $O(n)$	$O(\log_2 n)$
in	$O(\log_2 n)$	$O(1)$ -> $O(n)$	$O(\log_2 n)$ -> $O(n)$	$O(\log_2 n)$
del	$O(n)$	$O(1)$ -> $O(n)$	$O(\log_2 n)$ -> $O(n)$	$O(\log_2 n)$