# 数据结构

## 1.列表 、栈、队列

### 1. 列表

> 列表:在其他的编程语言中叫数组,是一种基本的数据结构

#### 1.1 数组的存储,查找,以及时间复杂度

数组中的元素是存储在一块连续的内存中,而且必须 给定要存储的数据类型以及存储的长度,因此

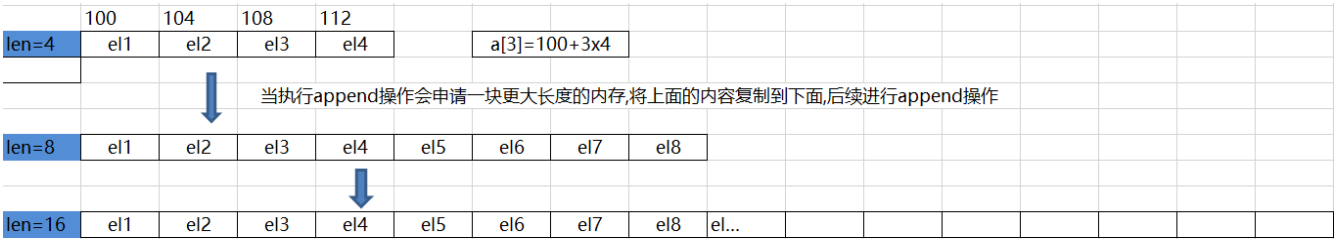
- 存储类型: 在创建时候必须声明存储的是那种的数据类型,这样数组中的元素的存储大小相同,加上是连续的存储方式,就可以依次进行查找
- 长度限制:声明长度,不能进行append操作,

#### 1.2 python中的列表

1.2.1 python 中的列表如何存储

注: 突破数组的两个限制 存储类型 | append等操作

(1)实现变长(可以append以及pop) 动态扩张和动态收缩




反之当执行pop操作时,会将剩余的元素,申请一块更小的内存,存储数据,原来开辟的内存,python会进行垃圾回收

(2) 如何实现存储不同类型的元素

lst = [3,3.15,'alex']


- 当修改值的时候,只是新开辟一块内存存储新的值,初始的列表内存修改新的地址,
- 此时没有引用的内存会被python回收 (引用计数的回收机制)

#### 1.3 python 列表操作的时间复杂度

- 存储过程虽然存在赋值的操作,但是时间复杂度为O(1) 摊还分析法(均摊)

- 另外pop不加内容的删除是O(1) ,给定内容的删除是O(n),存在位置的改变
- 赋值,修改等操作的时间复杂度为O(1)

## 2. 栈

### 2.1 栈的相关概述

- 栈的定义: 栈(Stack) 是一个数据集合,可以理解为只能在一端进行插入或删除操作
- 栈的特点:后进先出(last-in,first-out) > FIFO
- 栈的相关概念: 栈顶,栈底
- 栈的基本操作:进栈 (压栈),push,出栈,(pop),取栈顶(gettop)

注; 关于概念的面试问题

三个数的进展序列是ABC,下面哪些选项不可能是出栈序列

- CBA ,ABC,ACB,BAC,BCA,CAB (CAB不可能)

五个数的时候:ABCDE 进展序列

- ACBDE AEDCB ADBCE ABDCE

规律 三个元素 123进栈,出栈不可能是312

### 2.2 python实现栈

不需要自己定义,使用列表结构就可以

- 进栈函数 :append
- 出栈函数: pop
- 查看栈顶函数:li[-1]

栈的应用: 括号匹配问题: 给一个字符串，其中包含小括号、中括号、大括号，求该字符串中的括号是否匹配

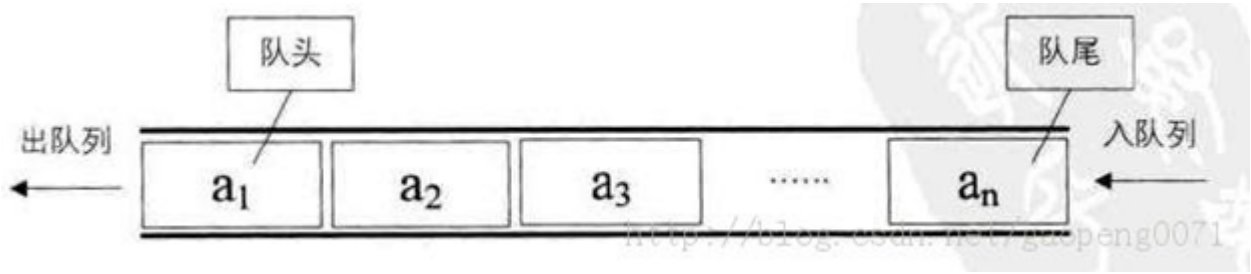
```
def brace_match(s):
    stack = []
    d = {'(':')','[':']','{':'}'}
    for ch in s:
        if ch in {'(','[','{':
            stack.append(ch)
        elif ch in {')', ']', '}':
            if len(stack) == 0:
                return False
            elif d[stack[-1]] == ch:
                stack.pop()
            else:
                return False
        else:
            pass
    if len(stack) > 0:
        return False
    else:
        return True
```

## 2.3 队列

**2.3.1 队列的概述:**

- 队列(Queue)是一个数据集合，仅允许在列表的一端进行插入，另一端进行删除。
- 进行插入的一端称为队尾(rear)，插入动作称为进队或入队
- 进行删除的一端称为队头(front)，删除动作称为出队
- 队列的性质：先进先出(First-in, First-out)

双向队列：队列的两端都允许进行进队和出队操作

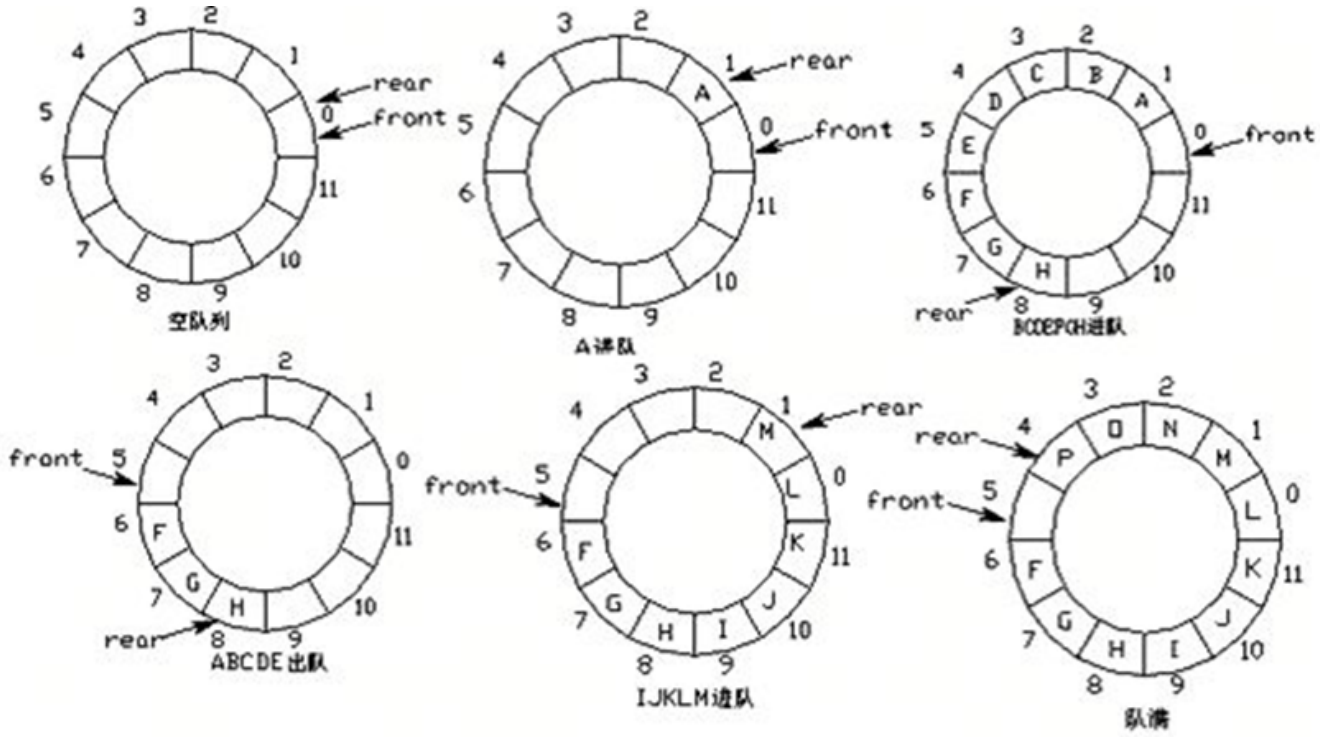



**2.3.2 队列的实现原理 --> 环形队列**

环形队列：当队尾指针front == Maxsize + 1时，再前进一个位置就自动到0。

实现方式：求余数运算

- 队首指针前进1：front = (front + 1) % MaxSize
- 队尾指针前进1：rear = (rear + 1) % MaxSize
- 队空条件：rear == front
- 队满条件：(rear + 1) % MaxSize == front




**2.3.3 队列的内置模块**

- 使用方法：from collections import deque
- 创建队列：queue = deque(li)
- 进队：append()

- 出队：popleft()
- 双向队列队首进队：appendleft
- 双向队列队尾进队：pop

```
from collections import deque

q = deque()
q.append(1)
q.append(2)
q.append(3)
print(q.popleft())
```

问题1：如何读取一个文件的后五行?

```
print(list(deque(open('abc.txt', 'r', encoding='utf-8'), 5)))

# q = deque([1,2,3,4,5], 3)
# print(list(q))               #[3, 4, 5]
```

问题2:如何用两个栈实现一个队列

```
#进队：  进栈1；  出队：出栈2；如果栈2空，就将栈1一次出栈并进到栈2，在从栈2出栈一次

class Queue:
    def __init__(self):
        self.stack1 =[]
        self.stack2 =[]

    def push(self,val):
        self.stack1.append(val)

    def pop(self):
        if len(self.stack2) > 0:
            return self.stack2.pop()
        elif len(self.stack1) > 0:
            while len(self.stack1) > 0:
                self.stack2.append(self.stack1.pop())
            return self.stack2.pop()
        else:
            raise ValueError('empty queue')
```

#### 4. 用栈和队列解决迷宫问题

# 2.链表

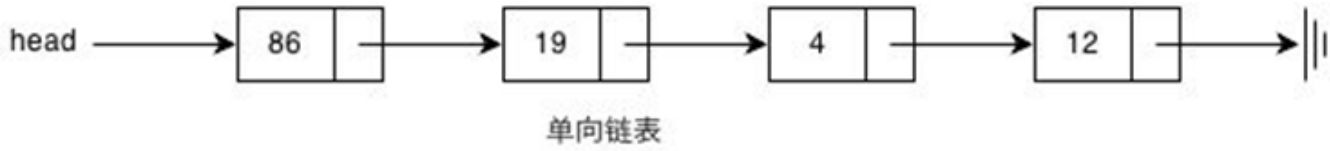
## 1. 单向链表

### 1.1 单向链表的概述

- 链表中每一个元素都是一个对象，每个对象称为一个节点，
- 包含有数据域key和指向下一个节点的指针next。通过各个节点之间的相互连接，最终串联成一个链表

节点的定义:

```
class Node(object):
    def __init__(self,data=None):
        self.data = data
        self.next = None
```




单向链表

### 1.2 单链表的建立与遍历

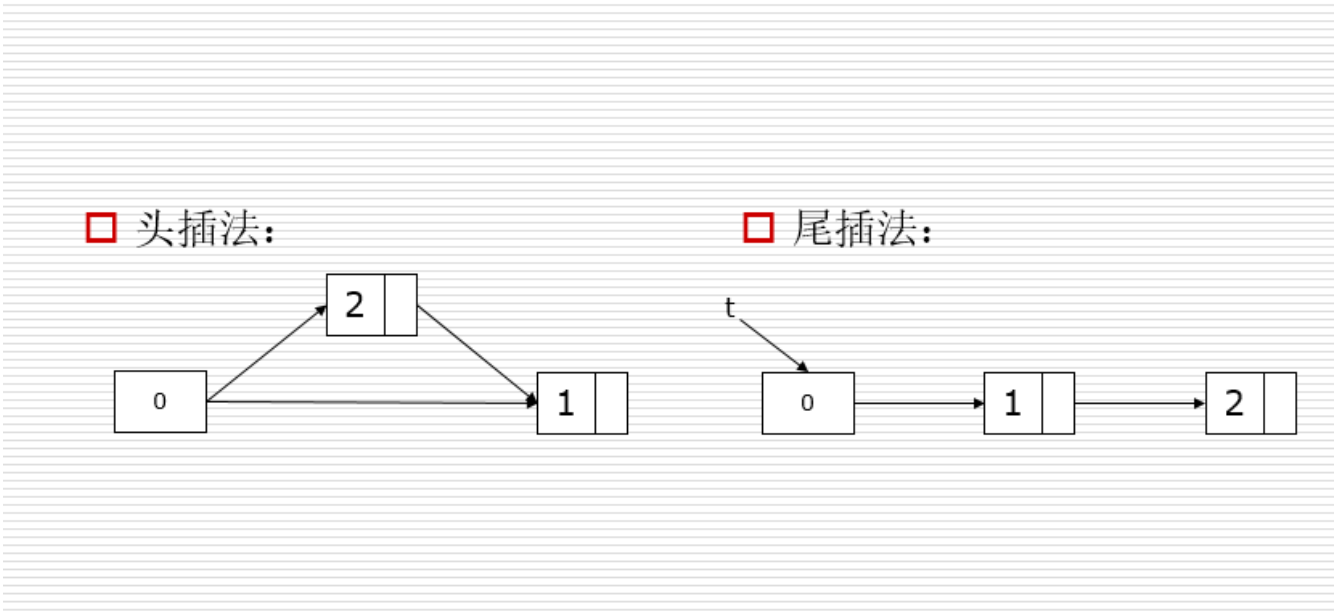
头插法

```
#头插法 -> 带头结点的链表    (头结点什么也不存)

class Node(object):
    def __init__(self,data=None):
        self.data = data
        self.next = None

def create_linklist(li):
    head = Node()
    for val in li:
        p = Node(val)                   #创建节点
        p.next = head.next              #创建的节点的下一个节点为当前链表中头结点的下个节点
        head.next = p                   #将p作为将头结点的下个节点
    return head       #头结点代表整个链表

#遍历整个链表

def traverse_linklist(head):
    p = head.next
    while p:
        yield p.data
        p = p.next

head = create_linklist([1,2,3,4,5])
for val in traverse_linklist(head):
    print(val)
```

尾插法

```
class Node(object):
    def __init__(self,data=None):
        self.data = data
        self.next = None

#遍历整个链表
```

```
def traverse_linklist(head):
    p = head.next
    while p:
        yield p.data
        p = p.next


def create_linklist_tail(li):
    head = Node()
    tail = head
    for val in li:
        p = Node(val)                #创建节点
        tail.next = p                #将当前的节点作为尾节点的下一个节点
        tail = p                     #将当前的节点作为尾节点
    return head

head = create_linklist_tail([1,2,3,4,5])
for val in traverse_linklist(head):
    print(val)
```

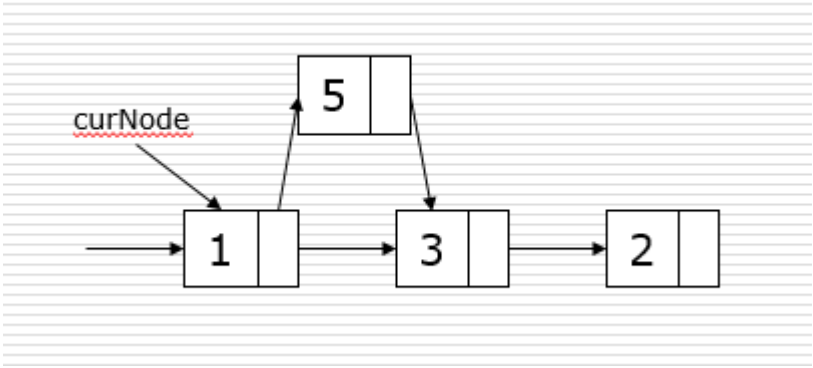



### 1.3 单链表节点的插入和删除

#### 1.3.1 插入

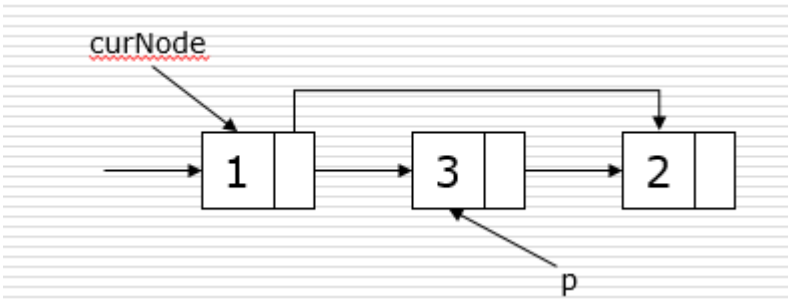
在当前指定的节点 curNode 后面插入一个节点 p

```
p.next = curNode.next    #将当前节点的下一个节点作为节点p的下一个节点
curNode.next = p             #将p作为当前节点的下一个节点
```




#### 1.3.2 删除

```
p = curNode.next
curNode.next = curNode.next.next
del p        #可以手动删除或者后续会自动回收
```



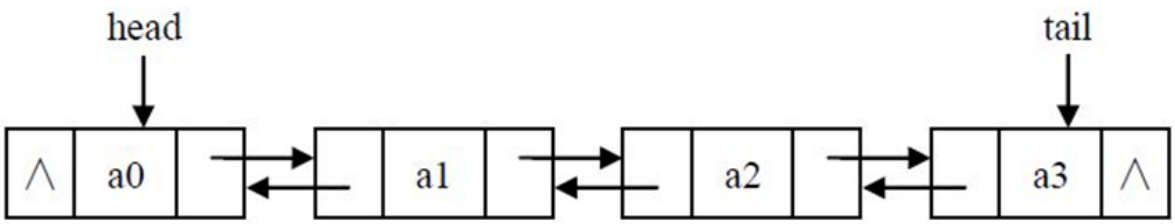

## 2. 双向链表

### 2.1 双链表的概述

双链表中每个节点有两个指针：一个指向后面节点、一个指向前面节点

节点的定义:

```
class DNode:
    def __init__(self,data = None):
        self.data = data
        self.next = None
        self.prior = None
```




### 2.2 双链表的建立和遍历

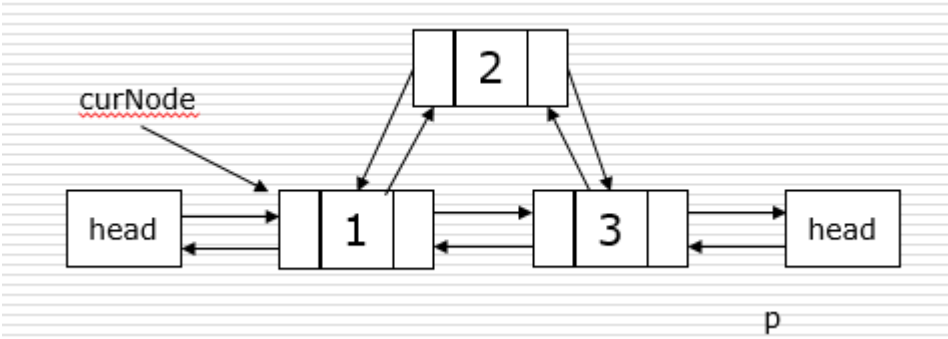
```
class DNode:
    def __init__(self,data = None):
```

```
        self.data = data
        self.next = None
        self.prior = None

def create_linklist_tail(li):
    head = DNode()
    tail = head
    for val in li:
        p = DNode(val)
        tail.next = p
        p.prior = tail
        tail = p
    return head,tail

def traverse_linklist(head):
    p = head.next
    while p:
        yield p.data
        p = p.next

#往前遍历
def traverse_linklist_back(tail):
    p = tail
    while p.prior:
        yield p.data
        p = p.prior

head, tail = create_linklist_tail([1,2,3,4,5])
for val in traverse_linklist_back(tail):
    print(val)
```

### 2.3 双链表节点的插入和删除

插入

```
p.next = curNode.next
curNode.next.prior = p
p.prior = curNode
curNode.next = p
```




删除

```
p = curNode.next
curNode.next = p.next
p.next.prior = curNode
del p
```

# 3.哈希表

## 1. 哈希表的引入

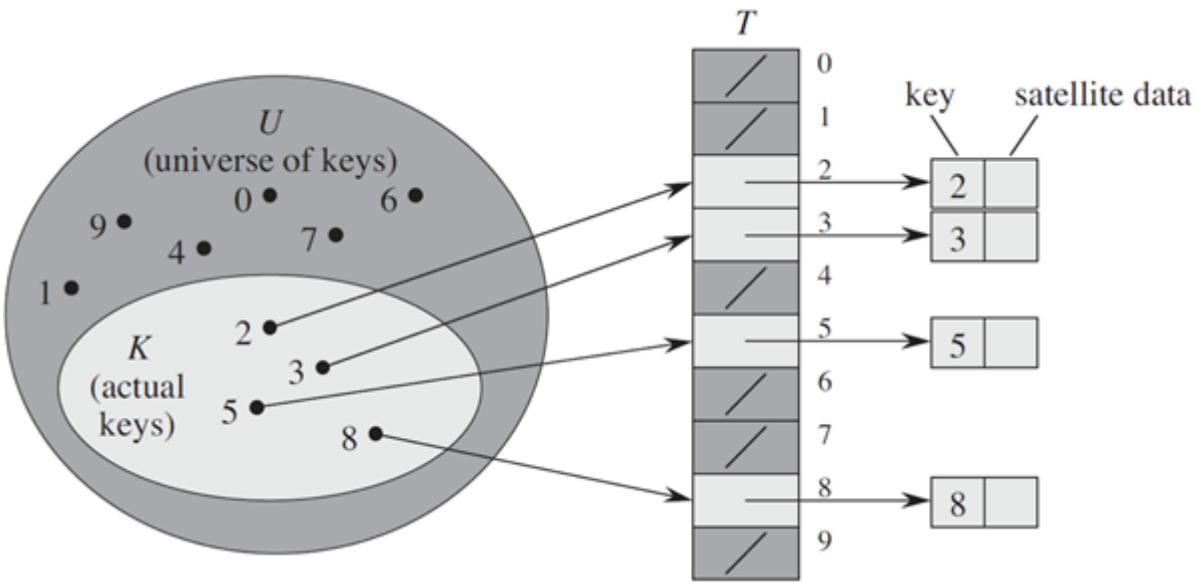
### 1.1 哈希表的简单概述

哈希表一个通过哈希函数来计算数据存储位置的数据结构，通常支持如下操作 (高效的操作)：python中的字典是通过哈希表实现的

- insert(key, value)：插入键值对(key,value)
- get(key)：如果存在键为key的键值对则返回其value，否则返回空值
- delete(key)：删除键为key的键值对

### 1.2.直接寻址表

当关键字的key 的 全域U(关键字可能出现的范围)比较小时，直接寻址是一种简单而有效的方法




- 存储 : 如上图将数组的下标作为key,将数值存储在对应的下表位置 key为k的元素放到k位置上
- 删除 : 当要删除某个元素时,将对应的下标的位置值置为空

直接寻址技术缺点：

- 当域U很大时，需要消耗大量内存，很不实际
- 如果域U很大而实际出现的key很少，则大量空间被浪费
- 无法处理关键字不是数字的情况,因为key可以是其他的数据类型

## 2. 哈希与哈希表

### 2.1 改进直接寻址表: 哈希

- 构建大小为m的寻址表T
- key为k的元素放到h(k)位置上
- h(k)是一个函数，其将域U映射到表T[0,1,...,m-1]

### 2.2 哈希表

- 哈希表（Hash Table，又称为散列表），是一种线性表的存储结构。哈希表由一个直接寻址表和一个哈希函数组成。
- 哈希函数h(k)将元素关键字k作为自变量，返回元素的存储下标。

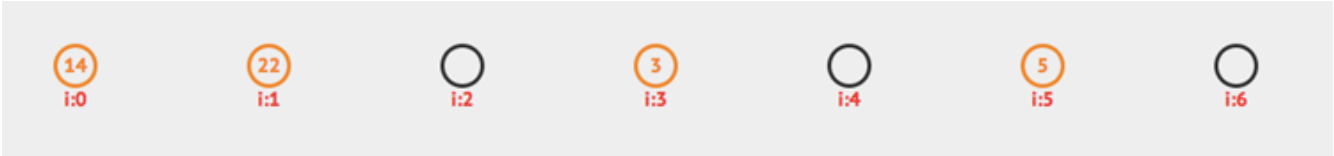
#### 2.2.1简单的hash函数

- 除法哈希：h(k) = k mod m
- 乘法哈希：h(k) = floor(m(kA mod 1)) 0<A<1

#### 2.2.3存储机制

以除法哈希为例讨论下存储机制以及存在的问题

假设有一个长度为7的数组，哈希函数h(k)=k mod 7,元素集合{14,22,3,5}的存储方式如下图。




解释:

- 存储 : key对数组长度取余,余数作为数组的下标,将值存储在此处
- 存在的问题 :比如：h(k)=k mod 7, h(0)=h(7)=h(14)=...

## 3.哈希冲突 &解决方法

### 3.1 哈希冲突

由于哈希表的大小是有限的，而要存储的值的总数量是无限的，因此对于任何哈希函数，都会出现两个不同元素映射到同一个位置上的情况，这种情况叫做哈希冲突。

### 3.2 解决哈希冲突

#### 3.2.1开放寻址法

如果哈希函数返回的位置已经有值，则可以向后探查新的位置来存储这个值。

- 线性探查：如果位置i被占用，则探查i+1, i+2,.....
- 二次探查：如果位置i被占用，则探查i+1^2,i+2^2.....
- 二度哈希：有n个哈希函数，当使用第1个哈希函数h1发生冲突时，则尝试使用h2,h3,.....

保证有空位存储 - > 动态扩张

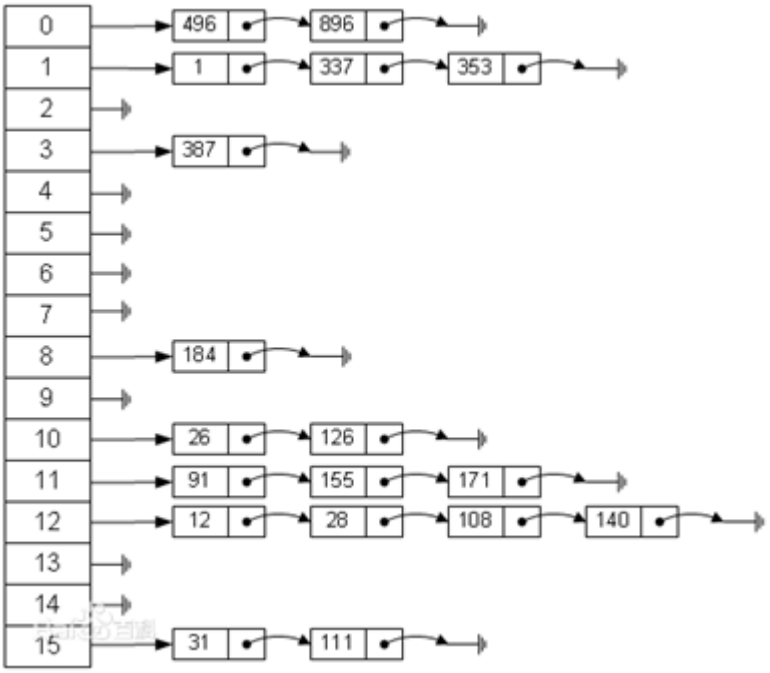
同样装载率因子 a = n / m a超过阈值是,将哈希表扩张一倍,重新计算哈希函数值,将值重新进行存储

查询:线性探查的问题 : 会导致大量的空格,大量的值连在一起,导致查询的时候变慢 ,当值分散开较好

为了解决上述问题,使用二次探查的方法

#### 3.2.2拉链法

哈希表每个位置都连接一个链表，当冲突发生时，冲突的元素将被加到该位置链表的最后




当一个位置后边的链表太长,再查找的时候会很慢

同样 拉链法也需要在装载率因子超过阈值的时候动态扩张 (但是他可以大于1)

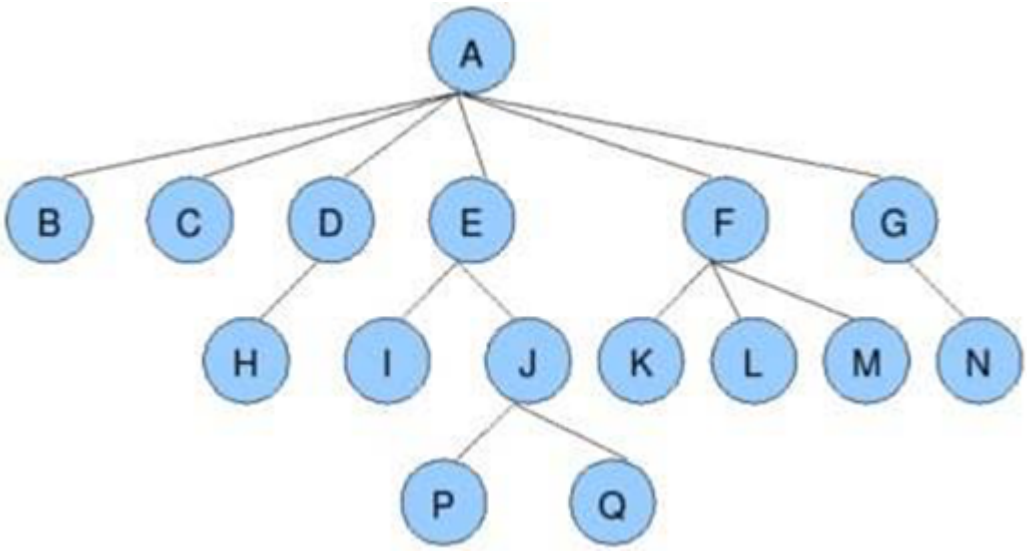
注: 由于key必须通过hash函数计算出一个整数,因此必须是不可变的数据类型

# 4.树

## 1. 树的概念

### 1.1 简单概述

- 树是一种数据结构 比如：目录结构
- 树是一种可以递归定义的数据结构
- 树是由n个节点组成的集合：
- 如果n=0，那这是一棵空树；
- 如果n>0，那存在1个节点作为树的根节点，其他节点可以分为m个集合，每个集合本身又是一棵树。

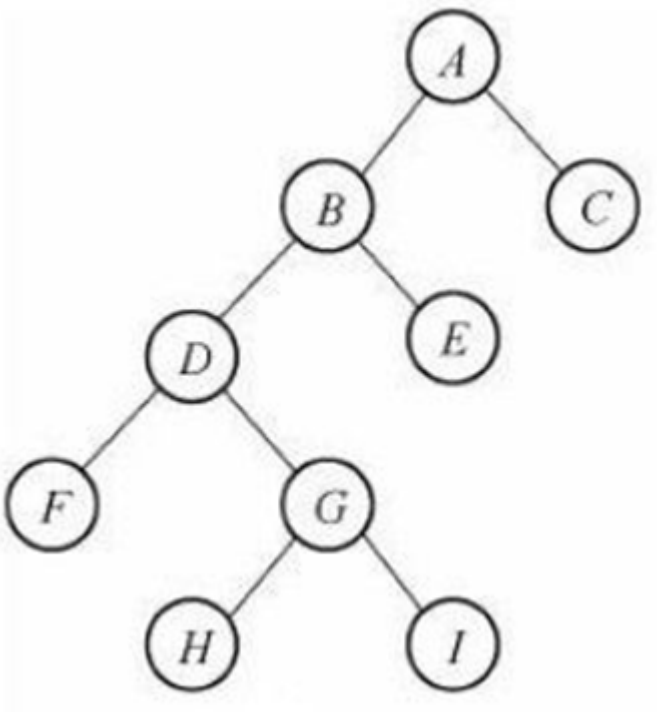



### 1.2 一些基本的概念

- 根节点、叶子节点 : A 根节点 P,Q是叶子结点
- 树的深度（高度）: 4 共有4层
- 树的度 :任意一个节点有几个叉
- 孩子节点/父节点 : 有上下级关系的节点
- 子树

## 2. 二叉树

### 2.1 定义; 度不超过2的树（节点最多有两个叉）




### 2.2 两种特殊的二叉树

**满二叉树 :** 一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树。

**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

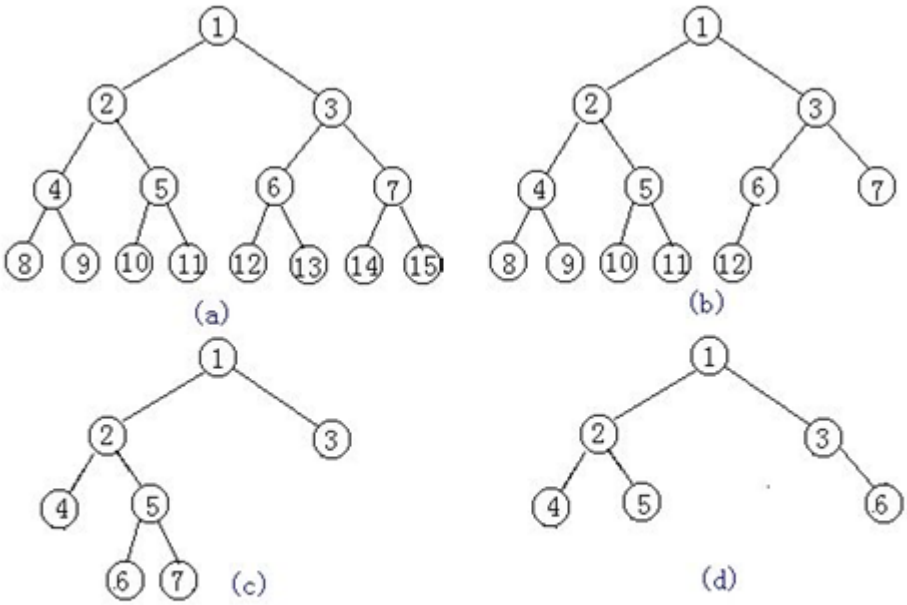
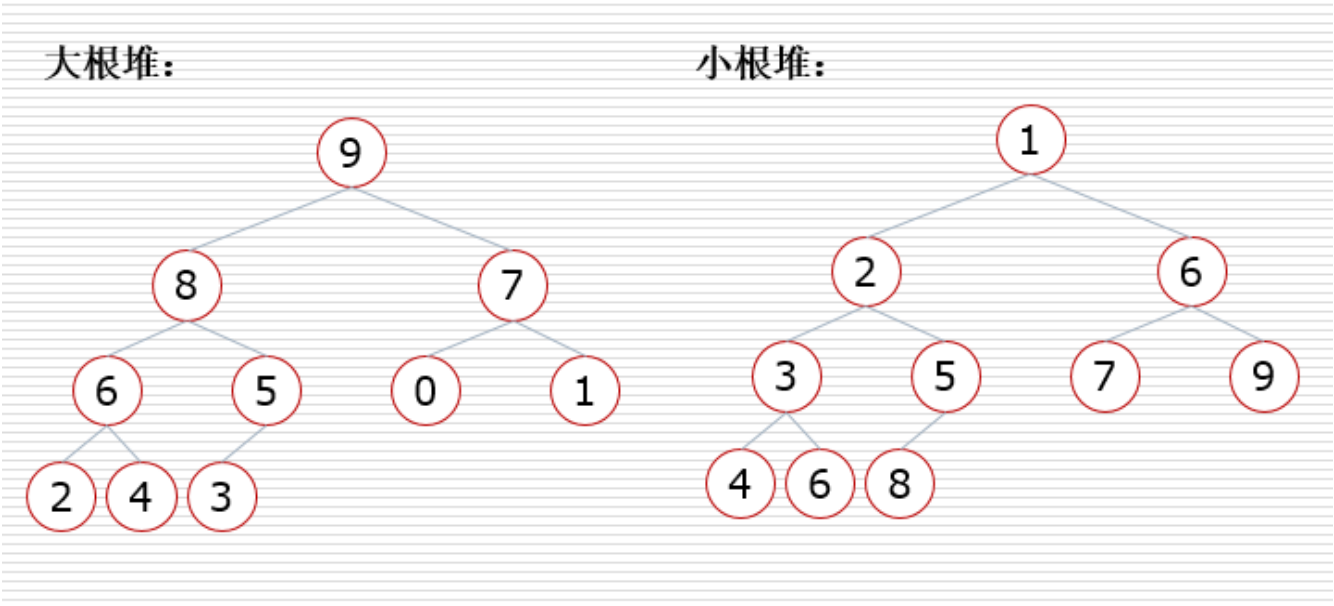



图6.5　特殊形态的二叉树

（a）满二叉树；（b）完全二叉树；（c）和（d）非完全二叉树。

### 2.2.1 堆

- 大根堆：一棵完全二叉树，满足任一节点都比其孩子节点大
- 小根堆：一棵完全二叉树，满足任一节点都比其孩子节点小



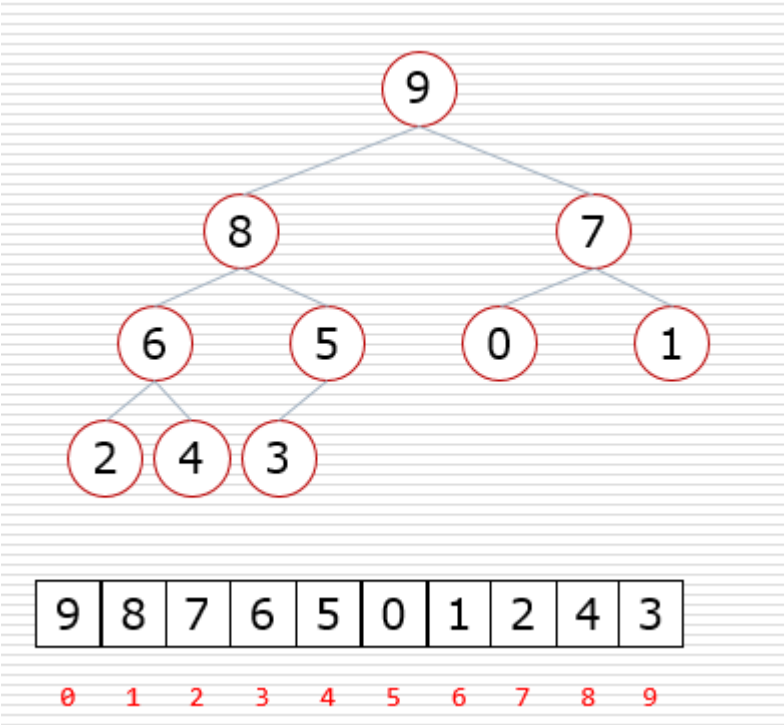

**堆的向下调整性质**

当根节点的左右子树都是堆时，可以通过一次向下的调整来将其变换成一个堆

在堆里插入一个元素：将新元素插到堆的末尾，使用**堆的向上调整**保证堆的性质

### 2.3 二叉树的存储方式

(1) 顺序的存储方式 (列表) ---- 只限于完全二叉树 ,其他的情况会浪费空间



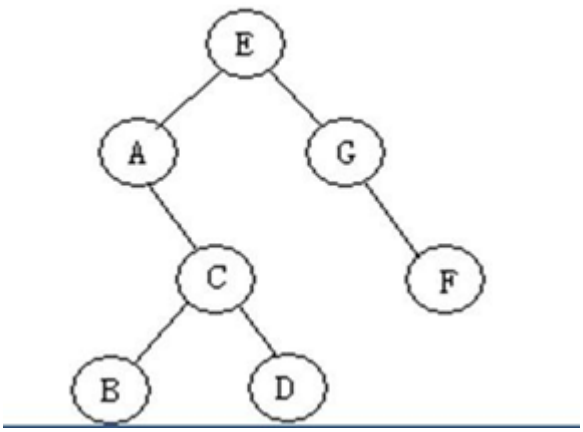

- 父节点和左孩子节点的编号下标有什么关系？0-1 1-3 2-5 3-7 4-9 i -> 2i+1
- 父节点和右孩子节点的编号下标有什么关系？0-2 1-4 2-6 3-8 4-10 i -> 2i+2

(2) 链式的存储方式

将二叉树的节点定义为一个对象，节点之间通过类似链表的链接方式来连接

节点的定义:

```
class BiTreeNode:
    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None
```




```
class BiTreeNode:
    def __init__(self, data):
        self.data = data
        self.lchild = None
```

```
        self.rchild = None

a = BiTreeNode('A')
b = BiTreeNode('B')
c = BiTreeNode('C')
d = BiTreeNode('D')
e = BiTreeNode('E')
f = BiTreeNode('F')
g = BiTreeNode('G')

e.lchild = a
e.rchild = g
a.rchild = c
c.lchild = b
c.rchild = d
g.rchild = f

root = e
```

### 2.4 二叉树的遍历

- 前序遍历

```
# 深度优先搜索 - 前序遍历(先序遍历)
def pre_order(root):
    if root: # 如果不是空树
        print(root.data, end='')
        pre_order(root.lchild)
        pre_order(root.rchild)

pre_order(root)   #EACBDGF
```

注: 先打印根节点,在递归左右孩子节点,会实现深度优先遍历

- 中序遍历 -> 当用栈来解释,就是每次从左孩子返回,打印该节点

```
# 深度优先搜索 - 中序遍历
def in_order(root):
    if root: # 如果不是空树
        in_order(root.lchild)
        print(root.data, end='')
        in_order(root.rchild)

in_order(root)    #ABCDEGF
```

注: 先递归左孩子,再输出自己,再递归右孩子,意味着只有当每一个节点的左孩子全部打印,再打印自己,最后打印右孩子

- 后续遍历

```
# 深度优先搜索 - 后序遍历
def post_order(root):
    if root: # 如果不是空树
        post_order(root.lchild)
        post_order(root.rchild)
        print(root.data, end='')

post_order(root)    #BDCAFGE
```

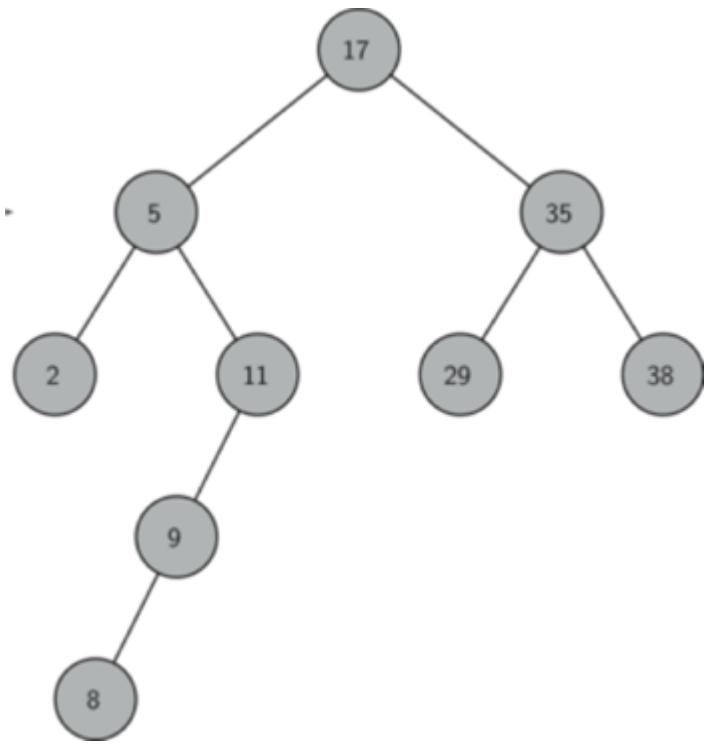
注: 先递归左子树,在递归有子树,最后输出自己 左子树,子树,节点

- 层次遍历

```
# 广度优先搜索 - 层次遍历
def level_order(root):
    q = deque()
    q.append(root)
    while len(q)>0: # 只要队列不空
        n = q.popleft()
        print(n.data, end='')
        if n.lchild:
            q.append(n.lchild)
        if n.rchild:
            q.append(n.rchild)

level_order(root)          #EAGCFBD
```

## 3. 二叉搜索树

二叉搜索树是一颗二叉树且满足性质：设x是二叉树的一个节点。如果y是x左子树的一个节点，那么y.key ≤ x.key；如果y是x右子树的一个节点，那么y.key ≥ x.key.



### 3.1 二叉树的创建 ,插入 ,遍历

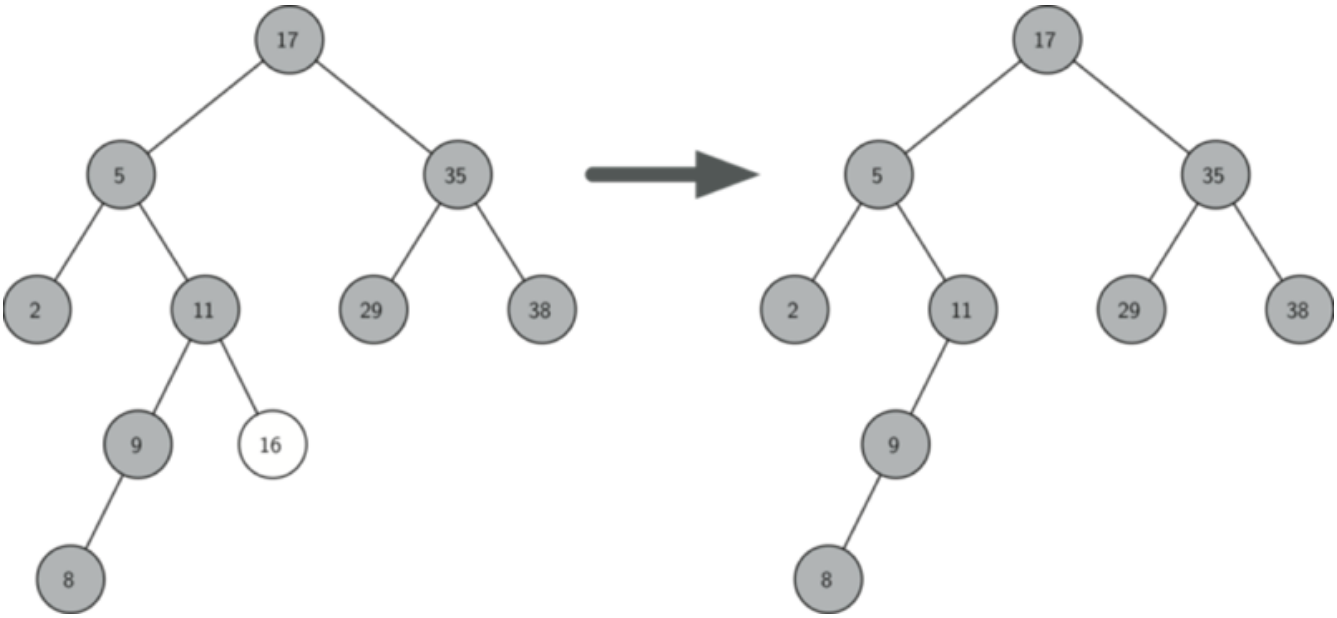
```
class BiTreeNode:
    def __init__(self, data):
        self.data = data
        self.lchild = None
        self.rchild = None


class BST:
    def __init__(self, li):
        self.root = None
        for v in li:
            self.insert(v)

    def insert(self, key):
        if not self.root:
            self.root = BiTreeNode(key)
            return
        p = self.root
        while p:
            if key < p.data:
                if p.lchild: # p有左子树
                    p = p.lchild
                else:
                    p.lchild = BiTreeNode(key)
                    return
            elif key > p.data:
                if p.rchild: # p有右子树
                    p = p.rchild
                else:
                    p.rchild = BiTreeNode(key)
                    return
            else:
                return


    def search(self, key):
        p = self.root
        while p:
            if key < p.data:
                p = p.lchild
            elif key > p.data:
                p = p.rchild
            else:
                return True
        return False


    def in_order(self):
        def _in_order(root):
            if root:  # 如果不是空树
                _in_order(root.lchild)
```

```
                print(root.data, end=',')
                _in_order(root.rchild)
        _in_order(self.root)
```

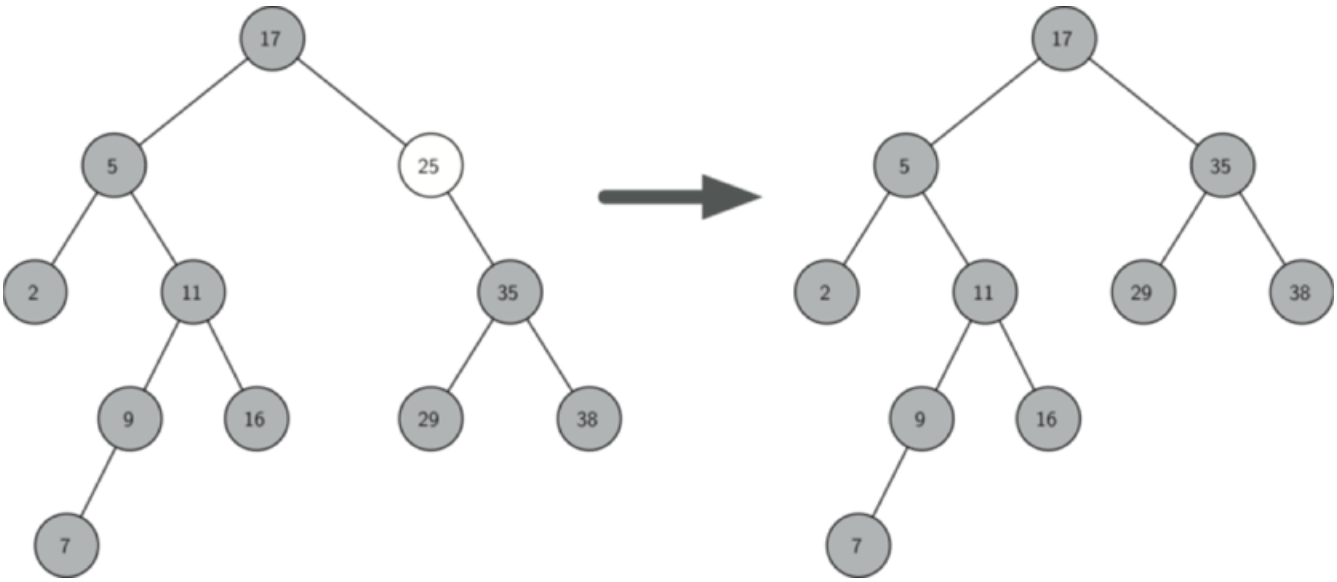
注: 二叉搜索树的中序遍历是升序,也可以做排序,排序 O(nlogn) 但是占了空间

**3.2 二叉搜索树的删除 O(logn)**

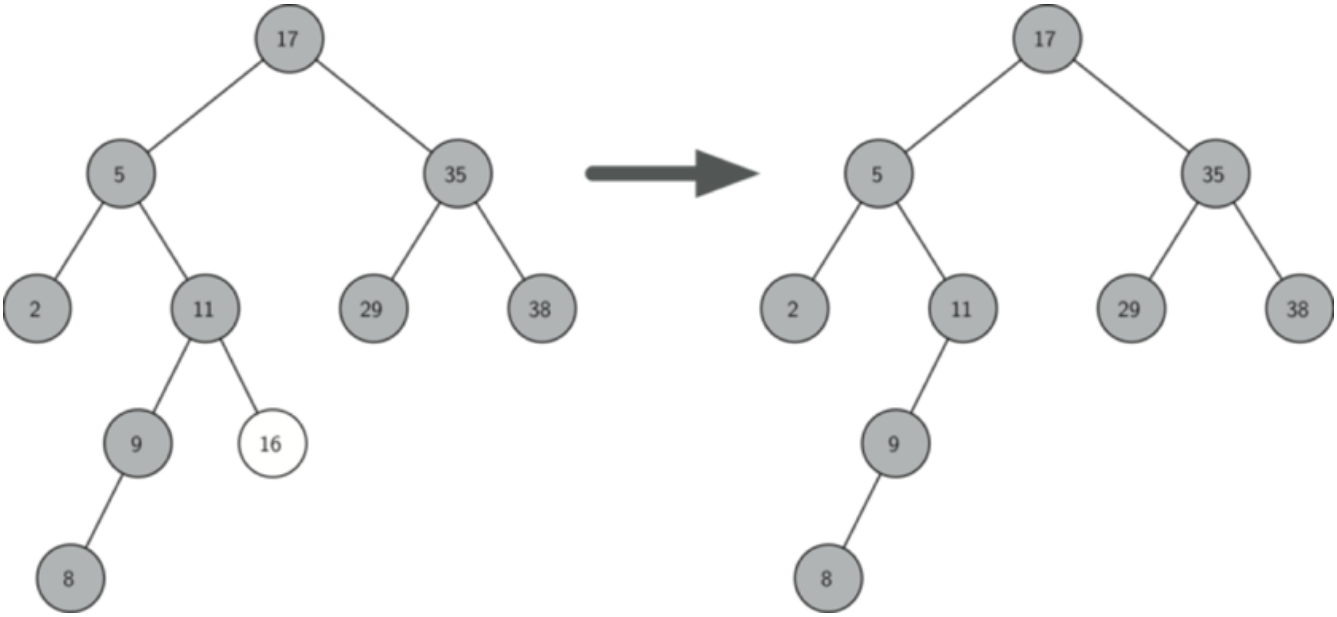
如果要删除的是叶子结点,直接删除




如果要删除的节点只有一个孩子：将此节点的父亲与孩子连接，然后删除该节点

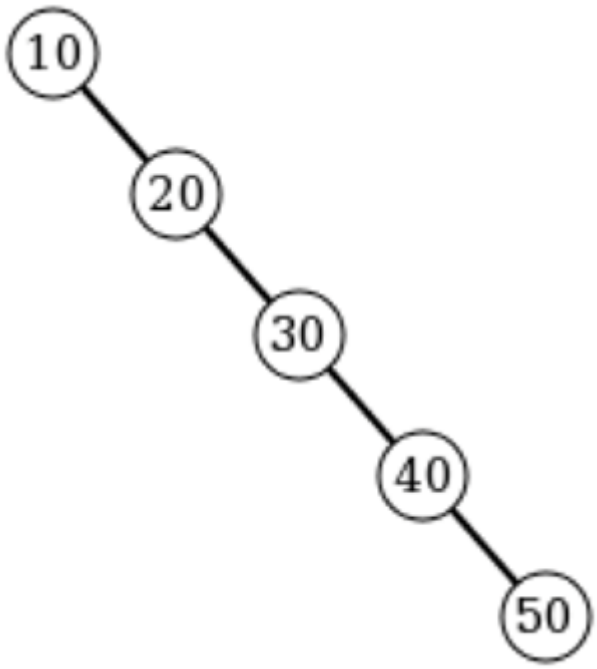



如果要删除的节点有两个孩子：将其右子树的最小节点（该节点最多有一个右孩子）删除，并替换当前节点




**3.3 二叉搜索树的效率**

- 平均情况下，二叉搜索树进行搜索的时间复杂度为O(nlogn)。
- 最坏情况下，二叉搜索树可能非常偏斜 O(n^2)

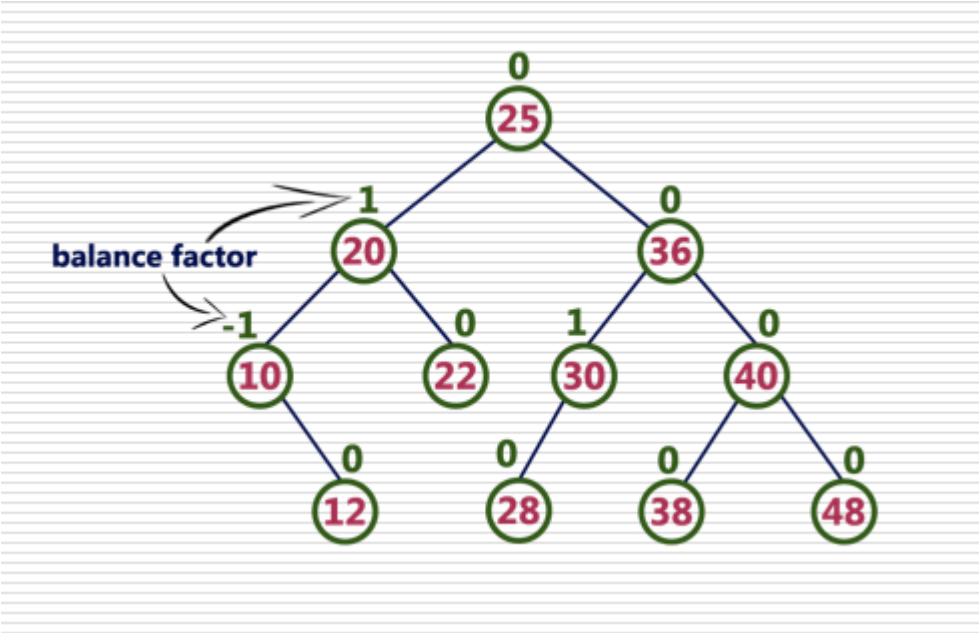



解决方案：

- 随机化插入
- AVL树

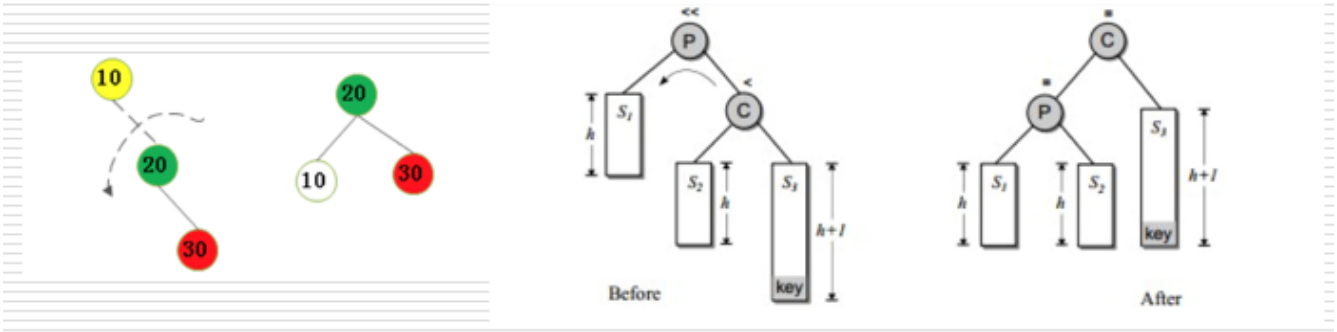
## 4. AVL树

AVL树：AVL树是一棵自平衡的二叉搜索树。 AVL树具有以下性质： 根的左右子树的高度之差的绝对值不能超过1 根的左右子树都是平衡二叉树

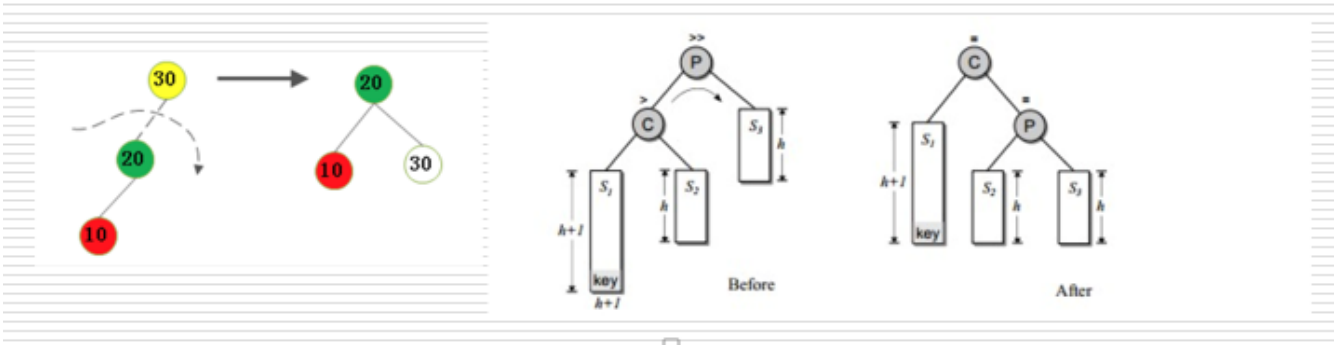



**AVL插入——旋转**

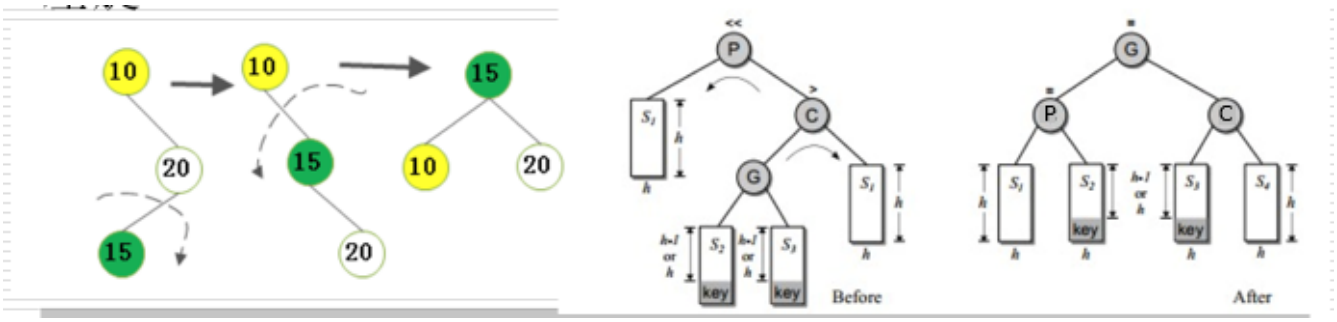
1.不平衡是由于对K的右孩子的右子树插入导致的：左旋



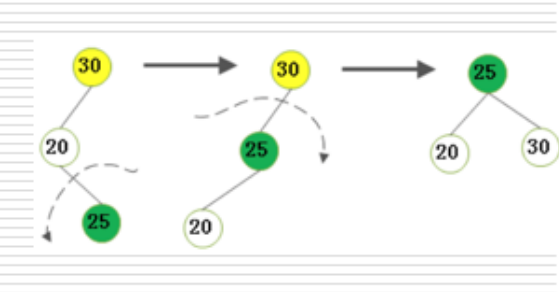

2.不平衡是由于对K的左孩子的左子树插入导致的：右旋

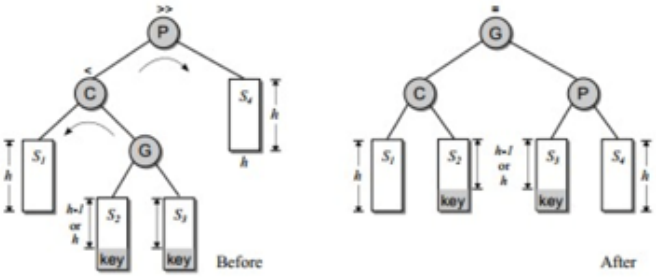



3.不平衡是由于对K的右孩子的左子树插入导致的：右旋-左旋



4.不平衡是由于对K的左孩子的右子树插入导致的：左旋-右旋




## 5. B树 | B+树**