递归

需要有一个递归终止条件

In [4]:
```
def func3(x):
    if x>0:
        print(x)
        func3(x-1)
def func4(x):
    if x>0:
        func4(x-1)
        print(x)
```

In [5]:
```
func4(3)
```

```
1
2
3
```

In [6]:
```
func3(3)
```

```
3
2
1
```

- 汉诺塔问题：

三根柱子abc，从下往上由大到小摞着64个圆盘，每次只能移动一个圆盘并且大圆盘不能再小圆盘上，最后将这一罗圆盘移动到另外一根柱子上

考虑三个圆盘的情况，

第一步需要把最上面的两个圆盘经过b移动到c，

第二步把底部的圆盘移动到柱子b上，

第三步通过柱子a把c上的顶部两个圆盘移动到b

假设有n个圆盘,则可以把顶部的n-1个圆盘看作一个整体，递归的使用这种移动方式来实现全部圆盘的移动

In [13]:
```
def hanoi(n,a,c,b):
    if n > 0:
        hanoi(n-1,a,b,c) ##step1 把顶部的n-1个盘子从a经过b移动到c
        print("moving from %s to %s" % (a,b))#step2 把底部的那个盘子a移动到b
        hanoi(n-1,c,a,b) #step3 把stpe1中经过移动到c的n-1个圆盘经过a移动到b
```

In [14]:
```
hanoi(3,"a","c","b")
```

```
moving from a to b
moving from a to c
moving from b to c
```

```
moving from a to b
moving from c to a
moving from c to b
moving from a to b
```

In [2]:
```
s1 = "acdt"
s2 = "at"
```

Out[2]: False

汉诺塔的时间复杂度:

假设移动n个盘子的时间为$h(n)$

则$h(n) = h(n-1) + 1 +h(n-1)$ 解得到$h(n) \sim 2^n$

列表查找

- 查找：在一些数据元素中通过一定的方法找出与给定关键字相同的数据元素的过程
- 列表查找(线性表查找):从列表中查找指定元素

  --- 输入:列表、待查找元素

  --- 输出:元素下标(未找到返回-1或None)

- 内置列表查找函数:index()---这是线性查找，因为人为指定的列表可能是无序的，而对一个列表排序的时间复杂度>O(n)

顺序查找(Linear Serach)

从列表第一个元素开始，顺序进行搜索,时间复杂度的为O(n)

In [15]:
```
def linear_search(li,val):
    for i,v in enumerate(li):
        if v == val:
            return i
        else:
            return None
```

二分查找(Binary Search)

针对有序列表的查找方法 从有序列表的初始候选区li[0:n]开始，通过对查找的值和候选区域中间值进行比较，可以使候选区域减半，维护候选区是关键

我们用left和right来标记候选区域的头和尾的index，中间值为li[(left+right/)2]，如果比查找值小，则维护右边区域为候选区，即让left <- (left+right/)2 +1，如果比查找值大，则维护左边区域为候选区，即让right <- (left+right/)2 -1

重复上面步骤 值得某个候选区的中间值等于待查找值结束，并返回这个中间值的索引

或者候选区域为空集(left>right)时结束，此时说明列表中没有待查找值

In [7]:
```
def binary_search(li,val):
    left = 0
```

```
    right = len(li) -1
    while (left <= right):  ##候选区有值时需要进行循环
        mid = (left + right) // 2
        if li[mid] == val:
            return mid
        elif li[mid] > val:  ##待查找值再候选区域左边
            right = mid - 1
        else:
            left = mid + 1
    else:
        return None
```

In [ ]:
```
def f(li,val):
    left = 0
    right = len(li) - 1
    while left <= right:
        mid = (left+right) // 2
        if li[mid] == val:
            return mid
        elif li[mid] < val:
            left = mid + 1
        else:
            right = mid -1
    return None
```

In [6]:
```
a = [1,2,3,5,8]
binary_search(a,4)
```

Out[6]: False

二分查找的复杂度为O(logn)(2为底)。因为当问题的规模为n时，每次循环都会减半问题的规模，所以只需要logn次就会把问题规模减到0

列表排序

- 排序：将一组无序的记录序列调整为有序的记录序列
- 列表排序: 将无序列表变成有序列表

  ----输入: 列表

  ----输出: 有序列表

- 升序与降序
- 内置排序函数 sort()

常见排序方法:

冒泡排序，选择排序，插入排序

快速排序，堆排序，归并排序

希尔排序，计数排序，；基数排序

冒泡排序(Bubble Sort)

- 列表每两个相邻的数，如果前面的数比后面的数大，则交换这两个数
- 冒泡排序的一趟 无序区会减少一个数，有序区会增加一个数

从第一个数开始，跟后一个位置的数比较，若更大则交换位置，值到后面位置的数比他更大，然后再从这个数开始跟他后面的位置的数比较(大的数往上冒，直到被一个更大的数盖住)。这一趟完成之后，最后一个位置的数一定是最大的(即无序区较少一个数，有序区增加一个数)

第二趟就只需要比较到倒数第二个位置的数即可

假设列表长度为n，第0趟结束后有序数区域有1个，而无序数区域有n-1个;第n-2趟结束后有序数区域有n-1个无序数区域有一个就是第一个位置的数，而它本身就是最小的数，不需要再被冒泡了，所以对一个长度为n的列表排序总共需要进行n-1趟冒泡

代码的关键点：趟，无序区范围 第0趟时 无序区有n个数 范围为[0:n-1]，第1趟时 无序区有n-1个数 范围为[0:n-2]，第i趟时 无序区有n-i个数 范围为[0:n-i-1]，第n-1趟时 无序区有1个数 范围为[0:0]

In [14…

```
def bubble_sort(li):
    for i in range(len(li) - 1): ##总共需要冒泡n-1趟
        for j in range(len(li)-i-1): ##指针在无序数区域内移动
            if li[j]>li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]
```

In [14…

```
import random
li = [random.randint(0,100) for i in range(100)]
print(li)
bubble_sort(li)
print(li)
```

```
[7, 18, 12, 31, 49, 26, 54, 91, 59, 79, 98, 13, 45, 75, 82, 15, 29, 65, 20, 6, 47, 22,
32, 79, 90, 66, 37, 58, 56, 40, 78, 25, 48, 8, 51, 58, 92, 25, 72, 35, 61, 10, 57, 20,
69, 20, 65, 3, 6, 1, 13, 39, 46, 83, 65, 34, 89, 60, 82, 64, 74, 10, 44, 83, 2, 1, 53,
38, 50, 45, 21, 49, 17, 68, 29, 3, 92, 31, 1, 58, 44, 88, 42, 61, 7, 20, 65, 79, 64,
8, 97, 80, 83, 82, 43, 61, 58, 99, 88, 58]
[1, 1, 1, 2, 3, 3, 6, 6, 7, 7, 8, 8, 10, 10, 12, 13, 13, 15, 17, 18, 20, 20, 20, 20, 2
1, 22, 25, 25, 26, 29, 29, 31, 31, 32, 34, 35, 37, 38, 39, 40, 42, 43, 44, 44, 45, 45,
46, 47, 48, 49, 49, 50, 51, 53, 54, 56, 57, 58, 58, 58, 58, 58, 59, 60, 61, 61, 61, 6
4, 64, 65, 65, 65, 65, 66, 68, 69, 72, 74, 75, 78, 79, 79, 79, 80, 82, 82, 82, 83, 83,
83, 88, 88, 89, 90, 91, 92, 92, 97, 98, 99]
```

In [14…

```
##只需要把冒泡过程的条件:比后一个元素大 改成比后一个元素小就能够实现降序排列
def bubble_sort(li):
    for i in range(len(li) - 1): ##总共需要冒泡n-1趟
        for j in range(len(li)-i-1): ##指针在无序数区域内移动
            if li[j]<li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]
```

冒泡排序的时间复杂度为O(n^2)

冒泡排序的改进: 如果一趟冒泡没有发生交换，则认为这个列表的排序已经完成

In [56]:

```
def bubble_sort(li):
    for i in range(len(li) - 1): ##总共需要冒泡n-1趟
        exchange = False
        for j in range(len(li)-i-1): ##指针在无序数区域内移动
            if li[j]>li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]
```

```
                exchange = True
        print(li)
        if not exchange:
            return
```

In [57]:
```
li = [9,8,1,7,2,6,3,4]
print(li)
bubble_sort(li)
```

```
[9, 8, 1, 7, 2, 6, 3, 4]
[8, 1, 7, 2, 6, 3, 4, 9]
[1, 7, 2, 6, 3, 4, 8, 9]
[1, 2, 6, 3, 4, 7, 8, 9]
[1, 2, 3, 4, 6, 7, 8, 9]
[1, 2, 3, 4, 6, 7, 8, 9]
```

选择排序

In [12…
```
def select_sort_simple(li):
    sort_li = []
    n = len(li)
    for i in range(n-1):
        t= 0
        min_ = li[t]
        for j in range(1,len(li)):
            if min_ >= li[j]:
                min_ = li[j]
                t = j
        sort_li.append(li[t])
        li.pop(t)
    sort_li.append(li[0])

    return  sort_li
```

In [12…
```
li = [3,1,4,2,7,5]

select_sort_simple(li)
```

Out[128]:
```
[1, 2, 3, 4, 5, 7]
```

In [12…
```
##简单的排序算法
def select_sort_simple(li):
    sort_li = []
    n = len(li)
    for i in range(n):
        min_val = min(li)
        sort_li.append(min_val)
        li.remove(min_val)
    return  sort_li
```

缺陷:

- 重新申请一块内存来存放sort_li
- 最小值运算是O(n)的

- 删除运算时O(n)的，删掉某个元素后还要把后面的元素往前面挪
- 这个算法的时间复杂度为O(n^2)
- 要考虑类似冒泡排序一样的原地排序

确定一个列表的无序区和有序区，第零趟时 无序区为整个列表[0,n-1]，经过无序区中一次选择最小的元素，把无序区第一个位置的元素与这个最小的元素交换位置，那么无序区的个数会减一[1:n-1],有序区个数会增加一[0]，第i次选择时，无序区的个数有n-i个，范围是[i,n-1], 把从无序区选择出来的最小元素与无序区的第一个元素交换,经过n-1次后无序区为列表的最后一个元素，而他本身就是最大的数

算法关键点 **有序区和无序区，无序区最小数的位置**

In [7]:
```
dp = [[0 for i in range(5)] for j in range(3)]
```

In [9]:
```
dp[1][1] = 1
```

In [10]:
```
dp
```

Out[10]:
```
[[0, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 0, 0, 0]]
```

In [3]:
```
def select_sort_simple(li):
    for i in range(len(li)-1):  ##i是第几趟
        min_loc = i
        for j in range(i,len(li)): ##确定无序区的位置[i:n]
            if li[min_loc] >= li[j]:
                min_loc = j
            li[i] ,li[min_loc] = li[min_loc], li[i]
    return li
```

In [6]:
```
li = [3,1,4,2,7,5]
select_sort_simple(li)
```

Out[6]:
```
[1, 2, 3, 4, 5, 7]
```

选择排序的时间复杂度为O(n)

插入排序

- 初始时有序区只有一个数(第一个位置的数)，每次让无序区的第一个位置的数插入到有序区的正确位置

In [16…
```
def insert_sort(li):
    for i in range(1,len(li)): #无序区的范围从[1:n]直到[n:n]
        for j in range(i):
            if li[i-j] < li[i-1-j]:
                li[i-j], li[i-1-j] = li[i-1-j], li[i-j]
        print(li)
    return li
```

In [16…
```
```

```
li = [3,1,4,2,7,5]
insert_sort(li)
```

```
[1, 3, 4, 2, 7, 5]
[1, 3, 4, 2, 7, 5]
[1, 2, 3, 4, 7, 5]
[1, 2, 3, 4, 7, 5]
[1, 2, 3, 4, 5, 7]
```

Out[165]: [1, 2, 3, 4, 5, 7]

In [9]:
```
def insert_sort(li):
    n = len(li)
    for i in range(n-1):
        j = i ##有序区的最后一个数的下标
#         tmp = li[i]  ##无序区的第一个数
        tmp = li[j+1]
        while j >= 0 and li[j] > tmp: ##j>0 以及 针对的数比tmp更大则需要把这个数往后
#             li[j+1] =  li[j]
            li[j+1],li[j] = li[j],li[j+1]
            j -= 1
#         li[j+1] = tmp
    return li
```

In [12]:
```
li = [3,2]
insert_sort(li)
```

Out[12]: [2, 3]

插入排序的时间复杂度为O(n^2)

In [9]:
```
res = "123"
res[::-1]
```

Out[9]: '321'

In [10]:
```
res
```

Out[10]: '123'

In [ ]:
```
a = 3
```

快速排序

In [57]:
```
# def partition(li,left,right):
#     tmp = li[left]
#     while (left < right) and (li[right] >= tmp): ##从右边找比tmp小的数
#         right -= 1
#     li[left] = li[right]

#     while (left<right) and (li[left] <= tmp) :
```

```
#         left += 1
#     li[right] = li[left]

#     li[left] = tmp
#     return left




# def quick_sort(data,left,right):
#     if left < right : #两个元素以上才需要递归
#         mid = partition(data,left,right)
#         quick_sort(data,left,mid-1)
#         quick_sort(data,mid+1,right)

#     return li
```

In [1]:

```
def partition(li,left,right):
    tmp = li[left]
    while left < right:
        while left < right and li[right] >= tmp:
            right -= 1
        li[left] = li[right]

        while left < right and li[left] <= tmp:
            left += 1
        li[right] = li[left]
    li[left] = tmp
    return left

def quick_sort(li,left,right):
    if left < right:
        mid = partition(li,left,right)
        quick_sort(li,left,mid-1)
        quick_sort(li,mid+1,right)
    return li

li = [3, 1, 5, 4, 18,7, 2]
print(quick_sort(li, 0, len(li)-1))
```

```
[1, 2, 3, 4, 5, 7, 18]
```

归并排序

合并一个两部分分别有序的列表为一整个有序列表

In [11…

```
def merge(li,low,mid,high):
    i = low
    j= mid+1
    l = []

    while i <= mid and j <= high:
        if li[i] <= li[j]:
            l.append(li[i])
            i += 1
        else:
            l.append(li[j])
            j += 1

    while i <= mid:
        l.append(li[i])
        i += 1
```

```
    while j <= high:
        l.append(li[j])
        j += 1

    li[low:high+1] = l



def merge_sort(li,low,high):
    if low < high: ###至少有两个元素
        mid = (low + high) // 2
        merge_sort(li, low, mid)
        merge_sort(li, mid+1, high)
        merge(li,low,mid,high)

    return li
```

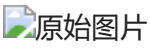
In [11…

```
li = [3,1,5,7,2,4]
merge_sort(li,0,len(li)-1)
```

Out[118]: [1, 2, 3, 4, 5, 7]

堆排序

满二叉树：每一层的节点数都达到最大值
完全二叉树：叶节点只能出现在最下层和次下层， 并且最下面一层的节点都集中在该层最左边的若干位置的二叉树



子节点下标找父节点下标为 子节点下标-1再整除2

大根堆：一颗完全二叉树，满足任意节点比其子节点大

小根堆：一颗完全二叉树，满足任意节点比其子节点小

堆的向下调整，节点左右子树都是堆，但自身不是堆 可以进行一次向下调整为堆

In [58…

```
def sift(ll,low,high): #high用来标记堆的最后一个位置 ， low标记堆的根节点位置
    i = low    #i指向根节点
    j = 2*i +1  #j是i 的 左孩子
    tmp = li[low] #存储堆顶
    while j <= high: ##j的位置没有越界  比较tmp和j 位置的值
        if j+1 <= high and li[j+1] >li[j]: #如果右子节点更大 把j 指向右孩子 从右分支
            j = j + 1
        if li[j] > tmp:
            li[i] = li[j]
            i = j   #往下看一层
            j = 2 * i + 1
        else: #tmp更大时 放到i所在的位置 即找到了合适的非叶节点的位置
            li[i] = tmp
            break
    else:
        li[i] = tmp  #j越界了 即i的位置已经找到了叶节点上 tmp只能再叶节点上
```

```
def heap_sort(li):
    n = len(li)
    #建堆
    for i in range((n-2)//2, -1, -1): # i为当前要调整的子树的根的下标
        sift(li,i,n-1)  ##这里的high 不需要计算每个调整的子树的最后一个位置  只需要整
                        ## 因为当i下移到叶节点上时 j的大小一定大于整颗树的最后一个位置
        print(li)
    #出数排序
    for i in range(n-1,-1,-1): # i指向当前堆的最后一个元素
        li[0],li[i] = li[i],li[0] ##将最后一个元素放上去 堆的大小减一

        sift(li,0,i-1) # 最后一个叶节点已经上去了 所以堆的最后一个元素向前挪一个位置
    print(li)
```

In [58…

```
li = [i for i in range(10)]
import random
random.shuffle(li)
heap_sort(li)
```

```
[7, 4, 8, 6, 5, 1, 2, 9, 3, 0]
[7, 4, 8, 9, 5, 1, 2, 6, 3, 0]
[7, 4, 8, 9, 5, 1, 2, 6, 3, 0]
[7, 9, 8, 6, 5, 1, 2, 4, 3, 0]
[9, 7, 8, 6, 5, 1, 2, 4, 3, 0]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [51…

```
### 堆的内置模块
import heapq
li = list(range(10))
random.shuffle(li)
heapq.heapify(li) ##建立一个小根堆
```

In [51…

```
heapq.heappop(li) #弹出最小的元素
```

Out[514]: 1

In [2]:

```
for i in range(3):
    for j in range(4):
        print(i+j)
        if j > 3:
            break
```

```
0
1
2
3
1
2
3
4
2
3
4
5
```

堆排序的时间复杂度

sift一次 最多循环 一颗树的高度那么多次 因为它每次把根节点与较大的子节点进行比较 时间复杂度为O(logn)

heap_sort 的 时间复杂度为O(nlogn)

堆排序---topK问题

先排序后切片 O(nlogn)
冒泡 O(Kn)
堆排序 O(nlogK)：利用列表的前K个元素建立一个小根堆 则堆顶是当前最小元素(即第K大的元素) 然后逐步的用列表后面的元素与堆顶进行比较 如果这个元素比堆顶的值更大 那么它就入堆 并更新 如果比这个元素更小 就跳过 循环完成之后堆里面就是K个最大的数 ，整个过程就是不断用列表后面的数更新K个数当中最小的数

In [63…

```
##topk需要建立小根堆
def sift_d(li, low, high):  # high用来标记堆的最后一个位置 ，low标记堆的根节点位置
    i = low  # i指向根节点
    j = 2 * i + 1  # j是i 的 左孩子
    tmp = li[low]  # 存储堆顶
    while j <= high:  ##j的位置没有越界  比较tmp和j 位置的值
        if j + 1 <= high and li[j + 1] < li[j]:  # 如果右子节点更小 把j 指向右孩子 从
            j = j + 1
        if li[j] < tmp:
            li[i] = li[j]
            i = j  # 往下看一层
            j = 2 * i + 1
        else:  # tmp更大时 放到i所在的位置 即找到了合适的非叶节点的位置
            li[i] = tmp
            break
    else:
        li[i] = tmp  # j越界了 即i的位置已经找到了叶节点上 tmp只能再叶节点上


def top_k(li, k):
    # 第一步取列表前k个元素 建立一个小根堆
    heap = li[0:k]
    for i in range((k - 2) // 2, -1, -1):  #从底部的第一个非叶子节点开始  递减到0结束
        sift_d(heap, i, k - 1)
    ##第二步 从第k+1个元素开始 与堆顶 比较 并更新堆
    for i in range(k, len(li)):
        if li[i] > heap[0]:
            heap[0] = li[i]
            sift_d(heap, 0, k-1)

    ##第三步 对最后的堆出数遍历
    for i in range(k - 1, -1, -1):
        heap[0], heap[i] = heap[i], heap[0]
        sift_d(heap, 0, i-1)

    return heap


li = [i for i in range(20)]

import random
random.shuffle(li)
top_k(li,7)
```

Out[631]: [19, 18, 17, 16, 15, 14, 13]

数据结构

相互之间存在一种或多种关系的数据元素的集合和该集合中数据元素之间的关系组成

- 数据机构按照其逻辑结构()可分为线性结构、树结构、图结构

  ---线性结构：数据结构的元素存在着一对一的关系 例如 列表

  ---树结构：数据结构中的元素存在着一对多的关系 例如 树(父子关系)

  ---图结构: 数据结构中的元素存在着多对多的关系

列表(顺序表)

- 列表中的元素顺序存储在内存中，占用一块连续的内存
- 列表的增删查改的时间复杂度
- 列表是如何实现的

In [64…

```
# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        l1_head = ListNode(l1[0])
        for i in range(0,len(l1)-1):
            l1_node = ListNode(l1[i])
            l1_node.next = ListNode(l1[i+1])
        l2_head = ListNode(l2[0])
        for i in range(0,len(l1)-1):
            l1_node = ListNode(l1[i])
            l1_node.next = ListNode(l1[i+1])
```

In [65…

```
li = [1,2,3]
a = ListNode(1)
for i in range(0,len(li)-1):

    tmp = ListNode(li[i])
    tmp.next = ListNode(li[i +1])
```

In [18]:

```
from functools import reduce
def rev(s):
    return reduce(lambda x, y : y + x, s)
```

In [21]:

```
def add(x, y) :            # 两数相加
    return x + y
sum1 = reduce(add, [1,2,3,4,5])   # 计算列表和：1+2+3+4+5
```

```
sum2 = reduce(lambda x, y: x+y, [1,2,3,4,5])  # 使用 lambda 匿名函数
print(sum1)
print(sum2)
```

```
15
15
```

In [19]:
```
rev("absc")
```

Out[19]: 'csba'

In [14]:
```
s = "abcd"
```

Out[14]: ['abcd']

In [13…
```
a = {'s':1}
print(id(a))
b = {'s':2}
print(id(b))

c = 5
d = 5
print(id(c))
print(id(d))
```

```
2309316434104
2309316434264
140710123577872
140710123577872
```

LIST

python 中的list是一种顺序存储结构，它可以存储不同类型的元素，所以在存储时它时存储的元素的位置，这样可以通过偏移量来进行索引，然后通过其地址来取出元素的值

字典

字典的存储方式是键值对的存储方式，它通过一个hash函数把键转化为一个hash值 作为存放value的唯一地址，hash函数要求是一个一一映射，同一个键会有同样的hash值，可以hash的对象必须是不可变对象。

In [2]:
```

```

Out[2]: [1, 2]

可变对象与不可变对象

- python 中 可变对象包括list 字典 集合，不可变对象包括 整型 字符串 浮点型 布尔值 元组
- 可变对象指的是可以通过对象的引用而改变对象的值，改变后引用指向的仍然是这块内存
- 不可变对象指的是不能通过对象的引用而改变对象的值，当引用发生变化时，实际会指向另外一个新的对象（新的地址），原来对象的内存将会被垃圾回收器回收

In [1]:
```
k = (1,2)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-1-c3f133218b62> in <module>
      1 k = (1,2)
----> 2 a = {'a':1, {1:2}:'d'}
      3 a

TypeError: unhashable type: 'dict'
```

数组的查找，数组存储了首个元素地址的内存 32位机器上一个整数占4个字节，一个地址也占4个字节(64位机器一个地址占8个字节) 不管元素占的内存多少个字节 它对应的地址始终只占4个字节 要查找某个位置i上的元素，那么只要在首地址加上4*i再读取这个位置的内存即可所以数组的查找时O(1)

数组与列表的不同：数组元素类型相同，数组长度固定

python列表如何解决元素类型不同的问题：

----在python中由于列表中的数据类型可以不一样，所以实际内存上存储的是每个元素的地址，那么列表的查找就等同于数组的查找，不过查找到的是指定位置的元素的地址，然后在根据地址取出列表的值

python列表如何解决长度不固定的问题：

----当对列表添加元素时，如果内存不够了，python会自动申请一块新的更大的内存(不一定要刚好)，然后拷贝下之前列表存储的元素的地址再进行添加操作

append的复杂度为O(1)

删除操作:删除指定位置的元素后需要把后边的内容依次往前面挪 时间复杂度为O(n)

插入：同样也需要挪后面的内容 O(n)

链表数据结构相似于列表，他的删除和插入时间复杂度会更低

栈

- 栈是一个数据集合，只能在一端进行插入或删除操作的列表
- 栈的特点是后进先出
- 栈的概念:栈顶、栈底
- 栈的基本操作：进栈(push)、出栈(pop)、取栈顶(gettop)(不拿走该元素)

栈的python实现

使用一般的列表结构即可实现栈：

----进栈：li.append

----出栈: li.pop

----取栈顶: li[-1]

In [20]:
```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self,element):
        self.stack.append(element)
    def pop(self):
```

```
        return self.stack.pop()
    def get_top(self):
        if len(self.stack)>0:
            return self.stack[-1]
        else:
            return None
    def is_empty(self):
        return len(self.stack) == 0
```

In [17…

```
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
```

In [17…

```
stack.pop()
```

Out[173]: 3

栈的应用:括号匹配问题： 给定一个字符串，包含小括号 中括号 大括号 求该字符串中的括号是否匹配(如()[][{}])

解决思路:字符串第一个开始考察，如果是左括号则执行入栈操作，如果是右括号，则考察栈顶是否有与之匹配的左括号，如果有则执行栈的pop操作，如果没有则说明不匹配。读完整个字符串之后，看栈是否为空，如果不为空说明一定少了某个右括号，说明不匹配。否则说明匹配

In [17…

```
def brace_match(s):
    stack = Stack()
    match = {"}":"{",")":"(","]":"["}
    for ch in s:
        if ch in {"(","[","{":  ##左括号入栈
            stack.push(ch)
        else:     ##如果是右括号的情况,则需要判断栈顶是否匹配该右括号
            if stack.is_empty():  #栈为空 说明前面差一个对应的左括号
                return False
            elif stack.get_top() == match[ch]: ##栈顶元素匹配的情况 则出栈操作
                stack.pop()
            else: ##不匹配
                return False

    if stack.is_empty():
        return True
    else:  ##可能还会差一个右括号 没能把栈里面的左括号取出来
        return False
```

In [18…

```
print(brace_match("[{(()}]"))
```

```
False
```

In [48]:

```
dic = {"a":1}
dic[list(dic.keys())[0]] = 2
```

In [55]:

```
dic = {"a":1,"b":2}
dic.
```

队列(Queue)

- 队列是一个数据集合，仅允许在列表的一端进行插入，另一端进行删除
- 插入的一端称为队尾(rear)
- 删除的一端称为队头(front)
- 队列的性质是先进先出

如果用列表来实现队列，出队用pop时,后面的元素要往前挪，时间复杂度为O(n)，所以出队时让fornt指向队头的下一个元素

当front指向队的最后一个元素时，再想入队时就没有位置了,如果再申请其余空间则太浪费内存，所以提出环形队列

 原始图片

队空：rear = front (front指向位置的下一个位置才是队列中的第一个元素的位置，队列的元素由front+1位置的元素直到rear指向位置的元素) rear前面的元素都是再队伍中的，front前面的都是没在队伍中的

满队列: rear + 1 = front

入队: rear +1

出队: front +1

 原始图片

rear指向11时还想要入队的话，用rear+1指针会变成12而不是0,所以把指针改成 (rear+1) % 12(队列的大小)

同样的出队、队满时也需要考虑front已经指向11的情况

- 队首指针前进1 : front = (front+1) % maxsize(队列长度)
- 队尾指针前进1 : rear = (rear+1) % maxsize
- 队空: rear = front
- 队满: (rear+1) % maxsize = maxsize

队列的手动实现

In [20]:

```
# 仅仅考虑长度固定的队列，长度不固定的队列会在队满的情况下开一个新的队列 并将原队列的
class Queue:
    def __init__(self,size):
        self.size = size
        self.queue = [0 for i in range(size)]
        self.rear = 0  #队尾
        self.front = 0 #队首
    def push(self,element):  ##将元素element 执行入队操作
        if not self.is_full():
            self.rear = (self.rear+1) % self.size
            self.queue[self.rear] = element
        else:
            raise IndexError("Queue is filled")
    def pop(self): ##将队首元素出队
        if not self.is_empty():
```

```
            self.front = (self.front + 1) % self.size
        else:
            raise IndexError("Queue is empty")
        return self.queue[self.front]
    def is_empty(self):
        return self.front == self.rear
    def is_full(self):
        return self.front ==  (self.rear+1) % self.size
```

In [32]:
```
q = Queue(5)
for i in range(4):
    q.push(i)
```

In [33]:
```
q.is_full()
```

Out[33]: True

In [35]:
```
q.pop() ##先进先出
```

Out[35]: 0

队列的内置模块

生成的是一个双向的队列，队首队尾都可以执行出队和入队操作

- q.append() 队尾入队
- q.appendleft() 队首入队
- q.popleft() 队首出队
- q.pop() 队尾出队 (想象成一个列表左边是队首)

队如果满了，还要执行入队操作的话，会把另外一端的元素挤走

In [ ]:
```
```

In [47]:
```
q = deque([1,2,3,4,5],5) ## 第二个参数指定队的长度
##q.append()
q.append(6)
q.popleft()
```

Out[47]: 2

In [7]:
```
q = deque([1,2,3,4,5],5) ## 第二个参数指定队的长度
##q.append()
q.appendleft(0)
q.pop()
```

Out[7]: 4

例:打印一个文件的后几行(用索引的话会占用内存)

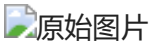
In [19]:
```
def tail(n):
    with open("f.txt","r") as f:
        q = deque(f,n) ##f中的元素出队 入队按照单项队列的规则实现
        return q
```

In [18]:
```
for line in tail(5):
    print(line,end="")
```

```
aa
cc
56a
zsh
whsxhn
```

文本长度比队列长度更长时，后面的元素入队会让前面的元素出队

迷宫问题

 原始图片

---栈 深度有限搜索(回溯法)

按照上右下左的顺序来分析每个点可走的路径，只要可走就走，直到走不动为止，则回退，回退到能走为止(走过的位置不能再走)

可以用栈来存当前路径

In [5]:
```
dirs =[
    lambda x,y:(x,y+1),
    lambda x,y:(x+1,y),
    lambda x,y:(x,y-1),
    lambda x,y:(x-1,y),
    ]
```

In [30]:
```
dirs[0](1,2)
```

Out[30]: (1, 3)

In [3]:
```
maze = [
    [1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,0,0,1,1,0,0,1],
    [1,0,1,1,1,0,0,0,0,1],
    [1,0,0,0,1,0,0,0,0,1],
    [1,0,1,0,0,0,1,0,0,1],
    [1,0,1,1,1,0,1,1,0,1],
    [1,1,0,0,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1,1]
    ]
```

In [13]:
```
def stack_maze_path(maze,x1,y1,x2,y2): ##(x1,y1)起点, (x2,y2)终点
    stack = []
```

```
    stack.append((x1,y1))
    while len(stack)>0:       ##栈空表示没有通路
        curNode = stack[-1]
        if (curNode[0] == x2) and (curNode[1] == y2):
            for p in stack:
                print(p)
            return True
        ## 上:(x,y+1),右:(x+1,y),下:(x,y-1),左(x-1,y)
        for dir in dirs:
            nextNode = dir(curNode[0],curNode[1])
            if maze[nextNode[0]][nextNode[1]] == 0: #如果下一个节点能走 则入栈
                stack.append(nextNode)
                maze[nextNode[0]][nextNode[1]] = 2 # 走过了则需要标记
                break

#         if stack[-1] == curNode:
#             stack
        else:  #如果这个节点无路可走，需要回退，同时他需要被标记
            maze[nextNode[0]][nextNode[1]] = 2
            stack.pop()
    else:
        print('没有路')
        return False
```

In [95]:

```
maze = [
    [1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,0,0,1,1,0,0,1],
    [1,0,1,1,1,0,0,0,0,1],
    [1,0,0,0,1,0,0,0,0,1],
    [1,0,1,0,0,0,1,0,0,1],
    [1,0,1,1,1,0,1,1,0,1],
    [1,1,0,0,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1,1]
    ]

stack_maze_path(maze,1,1,8,8)
```

```
(1, 1)
(1, 2)
(2, 2)
(3, 2)
(3, 1)
(4, 1)
(5, 1)
(5, 2)
(5, 3)
(6, 3)
(6, 4)
(6, 5)
(7, 5)
(8, 5)
(8, 6)
(8, 7)
(8, 8)
```

Out[95]: True

--- 队列 广度优先搜索 每一步同时考虑可以走的方向生成分叉的路径，最后得到的路径是最短路径

队列用来存储每一步搜索的最末位置

找路径： 用一个额外的2维列表来记录每个位置是由哪个位置出队导致他进队的 第一维记录入队的位置情况 第二维记录第一位位置是由哪个位置出队所导致入队的在第一维位置的坐标

难点在于用一个额外列表来记录每个位置的前继位置

In [10…

```
from collections import deque

def print_r(path):
    curNode = path[-1] #从记录的路径最后一个位置开始
    realpath = []
    while curNode[2] != -1:
        realpath.append(curNode[0:2]) #记录当前位置
        curNode = path[curNode[2]] #记录它的前继位置
    realpath.append(curNode[0:2])  #还要记录第一个位置
    realpath.reverse()
    for node in realpath:
        print(node)



def queue_maze_path(maze,x1,y1,x2,y2):
    queue = deque()
    path = [] #存放所有入队的位置
    queue.append((x1,y1,-1)) ## -1 是这个位置是由 前面哪个位置出队所带来的 位置的索引
    while len(queue) > 0: ##说明所有路出队之后都没位置入队(死路)
        curNode = queue.popleft()  ##队首出队 考察该位置所有后继位置
#         print(curNode)
        path.append(curNode)
        if  curNode[0] == x2 and curNode[1] == y2:
            print_r(path)
            return True
        for dir in dirs:
            nextNode = dir(curNode[0],curNode[1])
            if maze[nextNode[0]][nextNode[1]] == 0:
                queue.append((nextNode[0],nextNode[1],len(path)-1)) ##这个位置的入队一
#                 print('入队:',(nextNode[0],nextNode[1],len(path)-1))
                maze[nextNode[0]][nextNode[1]] = 2
    else:
        print('没有路')
        return False
```

In [99]:

```
maze = [
    [1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,1,0,0,0,1,0,1],
    [1,0,0,0,0,1,1,0,0,1],
    [1,0,1,1,1,0,0,0,0,1],
    [1,0,0,0,1,0,0,0,0,1],
    [1,0,1,0,0,0,1,0,0,1],
    [1,0,1,1,1,0,1,1,0,1],
    [1,1,0,0,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1,1]
    ]
queue_maze_path(maze,1,1,8,8)
```

```
(1, 1)
(2, 1)
(3, 1)
```

```
(4, 1)
(5, 1)
(5, 2)
(5, 3)
(6, 3)
(6, 4)
(6, 5)
(7, 5)
(8, 5)
(8, 6)
(8, 7)
(8, 8)
```

Out[99]:
```
True
```

In [96]:
```
maze
```

Out[96]:
```
[[1, 1, 1, 1, 2, 2, 2, 1, 1, 1],
 [1, 0, 2, 1, 2, 2, 2, 1, 0, 1],
 [1, 0, 2, 2, 2, 2, 2, 1, 0, 1],
 [1, 2, 2, 2, 2, 1, 1, 0, 0, 1],
 [1, 2, 1, 1, 1, 0, 0, 0, 0, 1],
 [1, 2, 2, 2, 1, 0, 0, 0, 0, 1],
 [1, 0, 1, 2, 2, 2, 1, 0, 0, 1],
 [1, 0, 1, 1, 1, 2, 1, 1, 0, 1],
 [1, 1, 0, 0, 0, 2, 2, 2, 2, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

链表

链表是一系列节点组成的元素集合，它包含两个部分 数据域item和指向下一个节点的指针next。通过节点之间的相互连接 最终串联成一个链表

In [12…
```
class Node:
    def __init__(self,item):
        self.item = item
        self.next = None
```

In [13…
```
a = Node(1)
b = Node(2)
c = Node(3)
```

In [13…
```
a.next = b
b.next = c
```

In [13…
```
print(a.next.item)
```

```
2
```

In [13…
```
print(a.next.next.item)
```

```
3
```

In [13…
```
print(a.next.next.next.item)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
<ipython-input-134-aa41d90fbba3> in <module>
----> 1 print(a.next.next.next.item)

AttributeError: 'NoneType' object has no attribute 'item'
```

In [11…

```
li = [1,2,3,4,5]
```

创建链表

--- 头插法: 把新的节点的指针指向头一个节点

--- 尾插法：把最后一个位置节点的指针指向新的节点

In [13…

```
def create_linklist(li):
    head = Node(li[0])
    for element in li[1:]:
        node = Node(element)
        node.next = head  ##头插法 让当前节点指向头节点
        head = node       ## 并将头节点指向该节点
    return head

def print_linklist(lk):
    while lk:  ##只要lk不是none
        print(lk.item,end = ',')
        lk = lk.next
```

In [13…

```
lk = create_linklist([1,2,3])  ##lk是一个头节点
```

In [13…

```
print_linklist(lk)
```

```
3,2,1,
```

In [13…

```
## 尾插法
def create_linklist_tail(li):
    tail = Node(li[0])
    head = tail
    for element in li[1:]:
        node = Node(element)
        tail.next = node
        tail = node
    return head
```

In [14…

```
lk = create_linklist_tail([1,2,3])
print_linklist(lk)
```

```
1,2,3,
```

链表节点的插入

-- 列表的插入(insert(i,n))的时间复杂度是O(n) -- 链表的存储不是顺序存储 它需要断开当前的指针 将插入的节点指向原来后面的那个节点 将前面的指针指向待插入的节点，只需两个步骤 时间复杂度很低O(1)

p.next = curNode.next

curNode.next = p

链表节点的删除

把待删除的节点的前继节点与后继节点连接起来 再删掉该节点 时间复杂度很低O(1)

curNode.next = p

curNode.next = p.next

del p

双链表

节点包括三个元素 多了一个prior指向前面的一个节点

双链表的插入

假设 p 为待插入节点 curNode 为当前节点,插入步骤如下：

p.next = curNode.next

curNode.next.prior = p

p.prior = curNode

curNode.next = p

双链表的删除

p = curNode.next

p.next.prior = curNode

curNode.next = p.next

del p

链表的复杂度分析

按下标查找 列表(顺序表) O(1) 链表O(n)

插入和删除 列表(顺序表) O(1) 链表 O(1)

--- 链表在插入删除上明显快于顺序表

--- 链表的内存可以更灵活的分配 实现栈和队列

哈希表

哈希表通过一个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

插入键值对 insert(key,value)

返回键对应的值 get(key) 删除指定键的值 delete(key)

直接寻指表

当关键字的全域U比较小时 直接寻址是一种简单而有效的方法 key 是 k 的元素放在 k 的位置上

--- 当域U很大时 消耗内存很大

--- 可能 U很大 但是实际出现的key很少 很大空间被浪费

--- 无法处理key 不是数字的情况

哈希

是对直接寻址表的改进：构建大小为m的寻址表， key为k的元素放在h(k)的位置上， h(k)是一个函数 将key 的域 U 映射到表T[0,1,...,m-1]上

哈希表

是一种线性表的存储结构 有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

例如 长度为7的哈希表 哈希函数为h(k) = k % 7. 元素集合{14,22,3,5} 的存储位置{0,1,3,5}

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线的 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的情况， 这种情况叫做哈希冲突 比如 h(k) = k % 7, h(0) = h(7) = h(14) = ...

哈希冲突的解决

--- 开放寻址法: 向后查探新的位置来存储这个值 ---------- 包括 线性探查 若位置i被占 探查 i+1,i+2 ....; 二次探查 i+1^2 , i-1^2, i+2^2, i-2^2,... ;二度哈希 n个哈希函数 如果第一个哈希函数发生冲突 则尝试使用第二个。。。。

--- 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的链表的最后 (相当于把hash表升级成二维)

常见哈希函数

除法哈希: h(k) = k % m

乘法哈希

全域哈希

In [24]:

```
class LinkList:
    class Node:
        def __init__(self,item = None):
            self.item = item
            self.next = None
    class LinkListIterator:
```

```
        def __init__(self,node):
            self.node = node
        def __next__(self):
            if self.node: ## 如果头节点 不为空 则返回该节点的值 并且将node 存为下一个
                cur_node = self.node
                self.node = cur_node.next
                return cur_node.item
            else:
                raise StopIteration
        def __iter__(self):
            return self

    def __init__(self,iterable = None):
        self.head = None
        self.tail = None
        if iterable:
            self.extend(iterable) ##循环插入 尾插

    def append(self,obj):
        s = LinkList.Node(obj)
        if not self.head: ##空链表的情况
            self.head = s
            self.tail = s
        else:
            self.tail.next = s
            self.tail = s

    def extend(self,iterable):
        for obj in iterable:
            self.append(obj)

    def find(self, obj):
        for n in self:
            if n == obj:
                return True
        else:
            return False

    def __iter__(self):  ## 把链表变成可迭代对象
        return self.LinkListIterator(self.head)
    def __repr__(self):
#         print("<<" + ",".join(map(str,self)) + ">>")
        return "<<" + ",".join(map(str,self)) + ">>" ## self 是可迭代对象 所以用可以
```

In [25]:
```
lk = LinkList([1,2,3,4,5])
```

In [26]:
```
lk.extend([6,7])
```

In [27]:
```
lk.__repr__()
```

Out[27]: '<<1,2,3,4,5,6,7>>'

In [13]:
```
for element in lk:  #for 语句会调用容器对象中的 iter()函数， 该函数返回一个定义了 __n
    print(element)
```

```
1
2
```

```
3
4
5
6
7
```

In [21…

```
class HashTable:
    def __init__(self,size = 101):
        self.size = size
        self.T = [LinkList() for i in range(self.size)] ##哈希表的每个位置上都是一个L

    def h(self,k):
        return k % self.size

    def find(self,k):
        i = self.h(k)
        return self.T[i].find(k) ##是否出现在hash表位置i的链表里面

    def insert(self,k):
        i = self.h(k) ##k插入hash表的位置上
        if self.find(k):
            print('Dumplicated Insert.')
        else:
            self.T[i].append(k)
```

In [21…

```
ht = HashTable()
```

In [21…

```
ht.insert(1)
ht.insert(4)
```

In [21…

```
ht.insert(102)
```

In [22…

```
print(','.join(map(str,ht.T)))
```

```
<<>>,<<1,102>>,<<>>,<<>>,<<4>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,
<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<
<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<
>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>
>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>
>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>,<<>>
```

In [22…

```
ht.find(12)
```

Out[221]: False

哈希函数的应用

- 字典和集合的实现
- md5 算法

树

递归定义的数据结构，涉及到的概念 根节点 叶节点 深度 ， 度(所有节点分叉最多的分叉数)， 子节点 父节点 子树

In [24…

```
##模拟文件系统树

class Node:
    def __init__(self,name,type = 'dir'): #默认的类型是文件夹
        self.name = name
        self.type = 'dir' or 'file'
        self.children = []  #子目录的连接
        self.parent = None #返回目录
    def __repr__(self):
        return self.name
```

In [23…

```
n = Node('hello')
n2 = Node('world')
n.children.append(n2)
n2.parent = n
```

In [23…

```
n.children[0].name
n2.parent.name
```

Out[237]: 'hello'

In [31…

```
class FileSystemTree:
    def __init__(self):
        self.root = Node('/') #定义根目录
        self.now = self.root
    def mkdir(self,name): #name是一个文件夹(以 / 结尾)
        if name[-1] != '/':
            name += '/'
        node = Node(name)
        self.now.children.append(node)
        node.parent = self.now

    def ls(self):
        return self.now.children

    def cd(self,name):
        if name[-1] != '/':
            name += '/'

        if name == '../':
            self.now = self.now.parent
            return

        for child in self.root.children:
            if child.name == name:
                self.now = child
                return
        else:
            raise ValueError('invalid dir')

#     def cd_m(self,name):
#         dirm = name.split('/')[1:-1]
#         for d in dirm:
#             node = Node(d+'/')
#             self.now.children.append(node)
```

```
#             node.parent = self.now
#             self.now = node
```

In [31…
```
ft = FileSystemTree()
# ft.mkdir("var/")
# ft.mkdir('bin/')
# ft.mkdir('usr/')
```

In [31…
```
ft.cd_m('/var/python')
```

In [27…
```
ft.ls()
```

Out[279]:
```
[var/, bin/, usr/]
```

In [29…
```
ft.cd('bin/')
```

In [29…
```
ft.mkdir('python/')
```

In [29…
```
ft.ls()
```

Out[299]:
```
[python/]
```

In [30…
```
ft.cd('../')
```

In [30…
```
ft.ls()
```

Out[302]:
```
[var/, bin/, usr/]
```

In [32…
```
'/var/python/'.split('/')[1:-1]
```

Out[323]:
```
['var', 'python']
```

二叉树(度不超过2)

链式存储 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连接方式来连接 不会造成空间的浪费

In [2]:
```
class BiTreeNode:
    def __init__(self,data):
        self.data = data
        self.lchild = None
        self.rchild = None
        self.parent = None
    def __inorder(self):
```

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

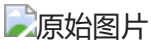
In [ ]:

In [3]:
```
e.lchild = a
e.rchild = g
a.rchild = c
c.lchild = b
c.rchild = d
g.rchild = f
```

In [17]:
```
root = e
root.lchild.data
```

Out[17]: 'A'

二叉树的遍历



In [15]:
```
##前序遍历 依次访问树的左节点和右节点
def pre_order(root):
    if root:
        print(root.data)
        pre_order(root.lchild)
        pre_order(root.rchild)
```

In [16]:
```
pre_order(root)
```

```
E
A
C
B
D
G
F
```

In [70]:
```
def in_order(root):
    stack = []
    res = []
    head = root
    while stack or head:
        while head:
            stack.append(head)
            head = head.lchild
        head = stack.pop()
        if stack:
            tmp = stack.pop()
```

```
            root = tmp.rchild
            res.append(tmp.data)
#         res.append(head.data)
#         if not head.rchild:
#             head = None
#         else:
#             head = head.rchild
        return res
```

In [69]:

```
def in_order(root):
    stack = []
    result = []
     while root or stack:
          while root: #  遍历左子树节点  入栈
                stack.append(root)
                root = root.lchild
          if stack:
                tmp=stack.pop() #出栈
                root=tmp.rchild #  访问右子树
                result.append(tmp.data)
     return result
# print(inorder(root))
```

```
  File "<ipython-input-69-dd9303530ac8>", line 4
    while root or stack:
    ^
SyntaxError: invalid character in identifier
```

In [16]:

```
## 中序遍历 左中右的顺序遍历
def in_order(root):
    if root:
        in_order(root.lchild)
        print(root.data, end = ',')
        in_order(root.rchild)
```

In [9]:

```
## 后序遍历  左右中的顺序遍历
def host_order(root):
    if root:
        host_order(root.lchild)
        host_order(root.rchild)
        print(root.data, end = ',')
```

可以根据前序遍历和中序遍历 或者 后序遍历和中序遍历还原一棵树 但是不能通过前序遍历和后续遍历还原一棵树(只能确定根节点，左右子树无法确定)

In [71]:

```
## 层次遍历 一层一层从左至右遍历
print(in_order(root))
```

```
['E']
```

In [37]:

```
root.data
root.lchild.lchild
```

In [35…

```
from collections import deque
```

```
# 利用队列实现层次遍历 最开始根节点在队列  出队打印值之后 将根节点的左右子节点加入队列
def level_order(root):
    queue = deque()
    queue.append(root)
    while len(queue) > 0:
        node = queue.popleft()
        print(node.data,end = '\n')
        if node.lchild:
            queue.append(node.lchild)
        if node.rchild:
            queue.append(node.rchild)
```

In [35…

```
level_order(root)
```

```
E
A
G
C
F
B
D
```

二叉树的三种遍历方式的非递归实现

In [5]:

```
class BinaryTree:
    class TreeNode:
        def __init__(self,x):
            self.val = x
            self.left = None
            self.right = None

    def __init__(self,data):
        self.data = data
        self.a = [self.TreeNode(data[i]) for i in range(len(data))]

        self.res = [[],[],[]]

    def preorder(self,root):
        if not root:
            return self.res
        stack = [root]
        while stack:
            node = stack.pop()
            self.res[0].append(node.val)
            if node.right:
                stack.append(node.right)
            if node.left:
                stack.append(node.left)
    def inorder(self,root):
        if not root:
            return self.res
        stack = []
        cur = root
        while cur or stack:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            self.res[1].append(cur.val)
            cur = cur.right
#             if not cur.right:
```

```
#                 cur = None
#             else:
#                 cur = cur.right
    def postorder(self,root):
        if not root:
            return self.res
        stack = []
        cur = root
        pre = None
        while stack or cur:
            while cur:
                stack.append(cur)
                cur = cur.left
            cur = stack.pop()
            if not cur.right or pre == cur.right:
                self.res[2].append(cur.val)
                pre = cur
                cur = None
            else:
                stack.append(cur)
                cur = cur.right
```

In [18]:
```
li = [1,2,3,4]
Btree = BinaryTree(li)
r = Btree.a
r[0].left = r[1]
r[0].right = r[2]
r[1].right = r[3]
```

In [19]:
```
Btree.postorder(r[0])
Btree.res
```

Out[19]: [[], [], [4, 2, 3, 1]]

In [20]:
```
Btree.inorder(r[0])
Btree.res
```

Out[20]: [[], [2, 4, 1, 3], [4, 2, 3, 1]]

In [21]:
```
Btree.preorder(r[0])
Btree.res
```

Out[21]: [[1, 2, 4, 3], [2, 4, 1, 3], [4, 2, 3, 1]]

In [29]:
```
li = [1]
li[1:1]
```

Out[29]: []

二叉搜索树

父节点的值总是比左子节点的值更大， 比右子节点值更小

In [44…

```
class BST:
    def __init__(self,li):
        self.root = None
        if li:
            for val in li:
                self.insert_no_rec(val)

    def insert(self,node,val):  ##递归方法的插入 node用来进行递归
        if not node:
            node = BiTreeNode(val)
        elif val < node.data:
            node.lchild = self.insert(node.lchild,val)
            node.lchild.parent = node
        elif val > node.data:
            node.rchild = self.insert(node.rchild,val)
            node.rchild.parent = node
        return node

    def insert_no_rec(self,val):
        p = self.root
        if not p:  ##空树的插入
            self.root = BiTreeNode(val)
            return
        while True:
            if val < p.data:
                if p.lchild:
                    p = p.lchild
                else:
                    p.lchild = BiTreeNode(val)
                    p.lchild.parent = p
                    return
            if val > p.data:
                if p.rchild:
                    p = p.rchild
                else:
                    p.rchild = BiTreeNode(val)
                    p.rchild.parent = p
                    return
            else:  ##已经存在这个节点
                return


    def query(self,node,val):
        if not node:  ##node递归之后是空的 说明找不到这个值
            return None
        if node.data > val:
            return self.query(node.lchild,val)
        if node.data < val:
            return self.query(node.rchild,val)
        else:
            return node
    def query_no_rec(self,val):
        p = self.root
        while p:
            if p.data < val:
                p = p.rchild
            elif p.data > val:
                p = p.lchild
            else:
```

```
                return p
        else:
            return None
    def __remove_node_1(self,node):##删除的节点node为叶子节点的情况
        if not node.parent: #该节点为根节点,删掉之后就是一颗空树
            self.root = None
        if node == node.parent.lchild:  ##该节点是左子节点
            node.parent.lchild = None
        else:  ##该节点为右子节点
            node.parent.rchild = None

    def __remove_node_21(self,node):##删除的节点只有一个左孩子
        if not node.parent: #为根的情况 删掉之后左孩子为根
            self.root = node.lchild
            node.lchild.parent = None
        elif node == node.parent.lchild: #如果它是它父亲的左孩子 则让它父亲与它的左孩
            node.parent.lchild = node.lchild
            node.lchild.parent = node.parent
        else: #它是父亲的右孩子的情况 则让它父亲的右孩子连接到它的左孩子
            node.parent.rchild = node.lchild
            node.lchild.parent = node.parent

    def __remove_node_22(self,node):  ## 删除的节点只有一个右孩子
        if not node.parent:
            self.root = node.rchild
            node.rchild.parent = None
        elif node == node.parent.lchild:
            node.parent.lchild = node.rchild
            node.rchild.parent = node.parent
        else:
            node.parent.rchild = node.rchild
            node.rchild.parent = node.parent

    def delete(self,val):  ##删除的节点有两个孩子的情况
        if self.root:   #不是空树
            node = self.query_no_rec(val)
            if not node: #待删除的值不在树中
                return False
            if not node.lchild and not node.rchild: ##如果是叶子节点
                self.__remove_node_1(node)
            elif not node.rchild: #如果没有右孩子 那么一定只有左孩子
                self.__remove_node_21(node)
            elif not node.lchild: # 如果没有左孩子 那么只有左孩子
                self.__remove_node_22(node)
            else:   ## 两个孩子都有的情况 寻找右子树的最小节点
                tmp_node = node.rchild
                while tmp_node.lchild: ##只要它的左孩子存在就一直往下找
                    tmp_node = tmp_node.lchild
                node.data = tmp_node.data #用这个最小节点替换待删除的节点
                if not tmp_node.rchild:   ##删除这个最小节点 有两种情况 它是叶子节点
                    self.__remove_node_1(tmp_node)
                else:
                    self.__remove_node_22(tmp.node)



    def pre_order(self,root):
        if root:
            print(root.data)
            self.pre_order(root.lchild)
            self.pre_order(root.rchild)
    def in_order(self,root):
        if root:
            self.in_order(root.lchild)
```

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

    def host_order(self,root):
        if root:
            self.host_order(root.lchild)
            self.host_order(root.rchild)
            print(root.data, end = ',')
    def level_order(self,root):
        queue = deque()
        queue.append(root)
        while len(queue) > 0:
            node = queue.popleft()
            print(node.data,end = '\n')
            if node.lchild:
                queue.append(node.lchild)
            if node.rchild:
                queue.append(node.rchild)
```

In [40…
```
bst = BST([1,10,2,8,9,5])
```

In [40…
```
bst.in_order(bst.root)
##中序遍历得到的结果是有序的
```

```
1,2,5,8,9,10,
```

In [40…
```
bst.query_no_rec(10).data
```

Out[405]: 10

In [40…
```
bst.delete(2)
```

In [40…
```
bst.in_order(bst.root)
```

```
1,5,8,9,10,
```

二叉搜索树的删除

有三种情况:

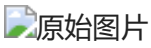
1 删除的节点是叶节点
2 删除的节点只有一个子节点 那么只需要将该节点的父节点与子节点相连接即可
3 删除的节点有两个子节点 那么我们需要用一个比他大一点点或小一点点的节点取代它 取右子树上最小的节点(这个节点一定没有左子节点)代替该节点 并采用 1或者2删除这个最小的节点

二叉搜索树的效率

平均情况，二叉搜索树由于每次查找只需要在树的一边查找，时间复杂度为O(logn)
最坏情况下，二叉搜索树可能非常偏斜 只有右子节点 形成了一个链表的结构 失去了树形结构

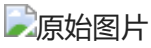
AVL树

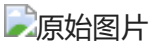
AVL是一颗自平衡的二叉搜索树 任何节点的左右子树高度差不能超过1 高度差定义了每一个节点的balance factor

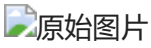


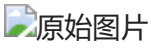
AVL树平衡的维护

插入节点后 坏破坏AVL树的平衡 通过4种旋转方式来重新平衡









In [44…

```
class AVLNode(BiTreeNode):
    def __init__(self,data):
        BiTreeNode.__init__(self,data)
        self.bf = 0

class AVLTree(BST):
    def __init__(self, li = None):
        BST.__init__(self,li)

    def rotate_left(self, p , c):
        s2 = c.lchild
        p.rchild = s2 #不论s2是否是空 都可以用p的rchild指向s2
        if s2: ## s2非空需要把parent指向p
            s2.parent = p
        c.lchild = p
        p.parent = c
        p.bf = 0
        c.bf = 0  #插入引起的不平衡经过右旋之后的平衡因子
        return c #返回根节点
    def rotate_right(self,p,c):
        s2 = c.rchild
        p.lchild = s2
        if s2:
            s2.parent = p
        c.rchild = p
        p.parent = c
        p.bf = 0
        c.bf = 0
        return c #返回根节点

    def rotate_right_left(self,p,c):
        g = c.lchild
        s3 = g.rchild
        s2 = g.lchild

        c.lchild = s3
        if s3:
            s3.parent = c
        g.rchild = c
        c.parent = g
        p.rchild = s2
        if s2:
            s2.parent = p
        g.lchild = p
```

```
        p.parent = g
        ## 平衡因子的更新跟插入g的左右节点有关系 如果插在左边则p.bf = 0,c.pf = 1 如果
        if g.bf > 0: ## 插入g的右边
            p.bf = -1
            c.bf = 0
        elif g.bf < 0:
            p.bf = 0
            c.bf = 1
        else:  ##当s1 s2 s3 s4 都是空的情况 插入的是g
            p.bf = 0
            c.bf = 0
        return g

    def rotate_left_right(self,p,c):
        g = c.rchild
        s2 = g.lchild
        s3 = g.rchild

        c.rchild = s2
        if s2:
            s2.parent = c

        g.lchild = c
        c.parent = g

        p.lchild = s3
        if s3:
            s3.parent = p
        g.rchild = p
        p.parent = g

        if g.bf > 0:
            p.bf = 0
            c.bf = -1
        elif g.bf < 0:
            c.bf = 0
            p.bf = 1
        else:
            p.bf = 0
            c.bf = 0
        return g #返回根节点

    def insert_no_rec(self,val):

        ###bf的变换规律
        ###从插入节点的父节点往上面走 如果插入的节点的在该节点的左边路径 那么该节点的b
        ### 如果插入节点在该节点的右边路径 那么该节点的bf加一
        ### bf 变化的传递 是从下往上走的 当某个节点的bf变成0时  则它上面的节点的bf 就不

        ## 根据bf 来判断哪个节点出现了不平衡 再根据子树的形状来进行旋转 再平衡
        #先插入
        p = self.root
        if not p:  ##空树的插入
            self.root = AVLNode(val)
            return
        while True:
            if val < p.data:
                if p.lchild:
                    p = p.lchild
                else:
                    p.lchild = AVLNode(val)
                    p.lchild.parent = p
                    node = p.lchild ##node 存储的是插入的节点
                    break
```

```
            if val > p.data:
                if p.rchild:
                    p = p.rchild
                else:
                    p.rchild = AVLNode(val)
                    p.rchild.parent = p
                    node = p.rchild
                    break

            else: ##已经存在这个节点
                return

        #再更新bf

        while node.parent:  ##一直检查到树的最顶端
            if node.parent.lchild == node: ##bf传递来自左子树
                if node.parent.bf < 0:#(-1) 左边沉 更新后变成-2
                #看node的bf 情况来确定旋转情况
                #如果node 是 左边沉 则 右旋
                #如果node 是 右边沉 则左旋 - 右旋
                    x = node.parent #旋转前子树的根
                    ance = node.parent.parent #为了连接旋转后的子树
                    if node.bf < 0:
                        n = self.rotate_right(node.parent,node)##旋转之后的根节点
                    else:
                        n = self.rotate_left_right(node.parent,node)
                    ##需要把 node 和 ance连接起来

                elif node.parent.bf > 0: #右边沉 更新后变成 0 无需旋转
                    node.parent.bf = 0
                    break
                else:
                    node.parent.bf = -1 #也无需循环
                    node = node.parent  #更新node 为上一节点再进行循环
                    continue

            else: ## bf 传递来自右子树
                if node.parent.bf < 0:#原来的树左边沉 更新后变成 0
                    node.parent.bf = 0

                if node.parent.bf >0: #原来的树右边沉 更新后变成 -2 需要旋转

                #看node的bf 情况来确定旋转情况
                #如果node 是 右边沉 则 左旋
                #如果node 是 左边沉 则右旋 - 左旋
                    x = node.parent
                    ance = node.parent.parent
                    if node.bf > 0:
                        n = self.rotate_left(node.parent,node)
                    else:
                        n = self.rotate_right_left(node.parent,node)
                else:
                    node.parent.bf = 1
                    node = node.parent
                    continue



            #连接旋转后的子树 ance 与 n 连接
            n.parent = ance
            if ance:  #插入节点的父节点的父节点不为空（被旋转的那部分树的父节点不为空
            ##
                if x == ance.lchild:
                    ance.lchild = n
```

```
                else:
                    ance.rchild = n
                break
            else:
                self.root = n
                break
```

In [45…

```
tree = AVLTree([9,8,7,6,5,4,3,2,1])
```

In [45…

```
tree.in_order(tree.root)
```

8,9,

AVL树的应用

B树 自平衡多叉树 用于数据库的索引

贪心算法

- 具体问题按照某个条件进行贪心

In [66…

```
##最少钱币找零问题 每次拿最大面额的钱币进行找零
t = [100,50,20,5,1]
def change(t,n):
    m = [0 for i in range(len(t))]
    for i,money in enumerate(t):
        m[i] = n // money
        n = n % money
    return m,n

change(t,376)
```

Out[666]: ([3, 1, 1, 1, 1], 0)

In [9]:

```
# ### 分数背包(固定背包重量)拿最大价值的东西 贪心算法：可以拿取的物品是可以分散拿的(粉
goods = [(60,10), (120,30), (100, 20)]
goods.sort(key = lambda x: x[0]/x[1],reverse = True)
def fractional_backpack(goods, w):
    total_v = 0
    m = [0 for i in range(len(goods))]
    for i, (prize, weight) in enumerate(goods):
        if w >= weight:  ##如果现有背包比商品重量更多，则把该商品拿完
            m[i] = 1
            total_v += prize
            w -= weight
        else:            ##如果现有背包比商品轻，则只能拿部分，且背包满了跳出循环
            m[i] = w / weight
            total_v += m[i] * prize
            w = 0
```

```
            break
    return m,total_v
```

In [10]:
```
fractional_backpack(goods,50)
```

Out[10]: ([1, 1, 0.6666666666666666], 240.0)

拼接最大数字问题 给定一系列数字字符串找出它们拼接后最大的数字

"96","87","128","1286","728","7286"

利用冒泡排序和特殊的排序准则: 如果两个数反着拼接比正着拼接更大则把前面的数换到后边来

In [7]:
```
li = [32,94,128,1286,6,71]
def number_join(li):
    li = list(map(str,li))
    n = len(li)
    for i in range(n-1):
        j = 0
        while j <= n-i-2:
            if li[j] + li[j+1] <= li[j+1] + li[j]:
                li[j], li[j+1] = li[j+1],li[j]
                j += 1
            else:
                j += 1

    return "".join(li)
```

In [16]:
```
number_join(li)
li = list(map(str,li))
```

In [20]:
```
"".join(li)
```

Out[20]: '32941281286671'

活动选择问题

现有一堆活动 已知 起止时间 规定只有一块活动场地 如何安排使得场地举办活动个数最多

我们总是安排当前结束时间最早的活动，以留存更多的时间给后续活动做安排

理论上来说 最优解一定包含当前结束最早的活动，若不然用另外那个结束最早的活动仍然构成最优解

In [46]:
```
li = [(1,4),(3,5),(0,6),(5,7),(3,9),(5,9),(6,10),(8,11),(8,12),(2,14),(12,16)]
```

In [55]:
```
#活动按结束时间排好序
li.sort(key = lambda x: x[1])
def activity_choose(li):
    res = [li[0]]
    for i in range(1,len(li)):
        if li[i][0]>= res[-1][1]: ##如果当前活动的开始时间晚于上个活动的结束时间 不冲
            res.append(li[i])
    return res
```

In [56]:
```
activity_choose(li)
```

Out[56]: [(1, 4), (5, 7), (8, 11), (12, 16)]

In [22]:
```
a = True
b = False
a or b
```

Out[22]: True

动态规划

斐波拉契数列:$F_n = F_{n-1} + F_{n-2}$ 用递归和非递归的方式求其数列的第n项

In [59]:
```
def feibonacci(n):
    if n==1 or n==2:
        return 1
    else:
        return feibonacci(n-1) + feibonacci(n-2)
```

In [66]:
```
##非递归
def f_rec(n):
    f = [0,1,1]
    if n>2:
        for i in range(n-2):
            num = f[-1] + f[-2]
            f.append(num)
    return f[n]
```

In [ ]:
```
def step(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return step(n-1)+step(n-2)

def opimal_step(n):
    a = [1,2]
    for i in range(2,n):
        a.append(a[-1]+a[-2])
    return a[n-1]
```

In [15…
```
op_step(100)
```

Out[152]: 573147844013817084101

In [85]:
```
def root(c,x):
    while 1:
```

```
        x_star = x/2 + c/(2*x)
        if abs(x_star - x) < 1e-7:
            break
        x = x_star
    return x_star
```

In [89]:
```
root(80,20)
```

Out[89]: 8.94427190999916

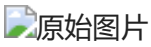
递归方式中子问题会有重复的计算 (注:并非所有情况递归方式都会有重复计算的问题) n = 4时 计算了两次 f[2] n = 6时 计算了两次 f[2], f[4]

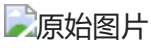
非递归方式存储了之前的计算结果 利用空间换取时间

DP问题 = 最优子结构 + 重复子问题

钢条切割问题

假设现在有钢条长度 和其对应的价格，给定一个长度的钢条求通过切割后能产生的最大 价值





最优子结构：如果小问题的最优解可以用来构造大问题的最优解时，则称为最优子结构。

长度为n的钢条，切一刀为k和n-k(k 从 1到 n-1)，只需要关心k的最优解 和n-k的最优解，因为子问题的最优切割方式一定蕴含在大问题的最优切割方式之中，若不然大问题的最优解就不成立，所以设长度为n的钢条经过切割后的收益为rn，可以得出递推式

---- $r_n = max(p_n,r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1}+r_1)$，$p_n$是不切割的价值

In [28]:
```
li = [(1,1),(2,5),(3,8),(4,9),(5,10),(6,17),(7,17),(8,20),(9,24),(10,30)]
a = list(map(lambda x: x[1],li))

def optimal_part(a, n):
    if n == 0:
        return 1
    else:
        r = a[n]
        for i in range(1,n):
            r = max(r, optimal_part(a,i) + optimal_part(a,n-i-1))
        return r
```

In [29]:
```
optimal_part(a,9)
```

Out[29]: 30

In [5]:
```
def popsort(li):
    n = len(li)
    for i in range(n-2):
```

```
        j=0
        while j < n-i-1:
            if li[j] >= li[j+1]:
                li[j],li[j+1] = li[j+1],li[j]
                j = j+1
            else:
                j = j+1
    return li
```

In [22]:
```
li = [3,1,2,5,8]
popsort(li)
```

Out[22]: [1, 2, 3, 5, 8]

In [12]:
```
def select_sort(li):
    n = len(li)
    for i in range(n-1):
        min_ = li[i]
        loc = i ## 如果确实找不到比有序区最后一个数小的，本身与本身进行交换，所以一开始
        for j in range(i+1,n):
            if li[j] <= min_:
                min_ = li[j]
                loc = j
        li[i] = min_
        li[loc] = li[i]
    return li
```

In [13]:
```
select_sort(li)
```

Out[13]: [1, 2, 3, 5, 8]

In [14]:
```
def insert_sort(li):
    n = len(li)
    for i in range(n-1):
        j = i
        while j>=0 and li[j+1] <= li[j]:
            li[j+1],li[j] = li[j],li[j+1]
            j -= 1

    return li
```

In [16]:
```
insert_sort(li)
```

Out[16]: [1, 2, 3, 5, 8]

In [26]:
```
def partition(li,left,right):
    tmp = li[left]
    while left < right:
        while left< right and li[right] > tmp:
            right -= 1
```

```
        li[left] = li[right]

        while left < right and li[left] < tmp:
            left += 1
        li[right] = li[left]
    li[left] = tmp
    return left

def Quick_sort(li,left,right):
    if left < right:
        mid = partition(li,left,right)
        Quick_sort(li,left,mid-1)
        Quick_sort(li,mid+1,right)
    else:
        return None
    return li
```

In [28]:
```
Quick_sort(li,0,len(li) -1)
```

Out[28]: [1, 2, 3, 5, 8]

In [48]:
```
s = "love"
li = []
for i in s:
    li.append(i)
n = len(li)
j = n-1
while j > 0 :
    li[j],li[j-1] = li[j-1],li[j]
    j -= 1

"".join(li)
```

Out[48]: 'elov'

In [49]:
```
li
```

Out[49]: ['e', 'l', 'o', 'v']

In [12]:
```
import sys
```

In [22]:
```
data = []
for line in sys.stdin:
    line = line.strip().split(" ")
    line[2] = int(line[2])
    line[-1] = int(line[-1])
    data.append(line)
n = len(data)
nodes = set([data[i][2] for i in range(n)])
result = []
for node in nodes:
    line_result = [node]
    pass_player= 0
    total_time = 0
    n_palyer = 0
    for i in range(n):
```

```
        if data[i][2] == node:
            n_player += 1
    for i in range(n):
        if data[i][2] == node and data[i][3] == True:
            pass_player += 1
            total_time += data[i][-1]
    pass_prop = pass_player / n_player
    avg_time = total_time / n_player
    remain_player = n_player - pass_player
    line_result.append(pass_prop)
    line_result.append(remain_player)
    line_result.append(avg_time)
    result.append(line_result)
```

In [19]:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-89e6c98d9288> in <module>
----> 1 b

NameError: name 'b' is not defined
```