数算知识点总结

一、DP

1.背包问题

0-1 背包：有n种物品，每种物品只有一个。每个物品有自己的重量和价值。有一个给定容量的背包，问这个背包最多能装的最大价值是多少。

Solution 1 : 二维数组

```
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost = [0], [0]
for i in range(n):
    cur_c, cur_w = map(int, input().split())
    w.append(cur_w)
    cost.append(cur_c)

#该初始化代表背包不一定要装满
dp = [[0 for j in range(v+1)] for i in range(n+1)]

for i in range(1, n+1):
    for j in range(1, v+1):    #可优化成 for j in range(cost[i], v+1):
        if j < cost[i]:
            dp[i][j] = dp[i-1][j]
        else:
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-cost[i]]+w[i])
print(dp[n][v])
```

Solution 2 : 滚动数组

```
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost = [0], [0]
for i in range(n):
    cur_c, cur_w = map(int, input().split())
    w.append(cur_w)
    cost.append(cur_c)

#该初始化代表背包不一定要装满
dp = [0 for j in range(v+1)]

for i in range(1, n+1):
    #注意：第二层循环要逆序循环
    for j in range(v, 0, -1):       #可优化成 for j in range(v, cost[i]-1,
-1):
        if j >= cost[i]:#否则j<cost[i],dp[i][j]=dp[i-1][j],也就是dp[j]无需更新
            dp[j] = max(dp[j], dp[j-cost[i]]+w[i])

print(dp[v])
```

2.完全背包问题

有N种物品和一个容量是V的背包，每种物品都有无限件可用。第i种物品的体积是vi，价值是wi。求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大，求出最大总价值。

Solution： 滚动数组

```
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost = [0], [0]
for i in range(n):
    cur_c, cur_w = map(int, input().split())
    w.append(cur_w)
    cost.append(cur_c)

#该初始化代表背包不一定要装满
dp = [0 for j in range(v+1)]

for i in range(1, n+1):
    #只需要将0-1背包一维DP解法中的二层循环改为顺序循环
    for j in range(1, v+1):
        if j >= cost[i]:
            dp[j] = max(dp[j], dp[j-cost[i]]+w[i])

print(dp[v])
```

为什么顺序遍历就能解决问题？对于当前物品 i ,要么不拿为dp[j]，要么拿为dp[j-cost[i]]+w[i]，而dp[j-cost[i]]代表之前的状态中，也包含拿过物品 i 的状态，这样就包含了多次拿取物品 i 的情况。

优化：若两件物品A，B满足A的体积大于B并且A的价值不大于B，那么可以直接排除使用A的可能。优化复杂度在$O(n^2)$

3.多重背包问题

有N种物品和一个容量是V的背包。第i种物品最多有si件，每件体积是vi，价值是wi。将哪些物品装入背包，可使物品总体积和不超过背包容量，且总价值和最大。

Solution 1： 0-1背包的变式：将每件物品的件数 Mi 作为独立的物品

```
# n, v分别代表物品数量，背包容积
n, v = map(int, input().split())
# w为物品价值，c为物品体积（花费）
w, cost, s = [0], [0], [0]
for i in range(n):
    cur_c, cur_w,cur_s= map(int, input().split())
    w += [cur_w]*cur_s
    cost += [cur_c]*cur_s

n = len(w)-1

#该初始化代表背包不一定要装满
dp = [0 for j in range(v+1)]

for i in range(1, n+1):
    for j in range(v, cost[i]-1, -1):
```

```
        if j >= cost[i]:
            dp[j] = max(dp[j], dp[j-cost[i]]+w[i])

print(dp[v])
```

Solution 2： 优化的转化到0-1背包问题

```
class Solution:
    # 0-1背包问题的写法
    def max_value(self, n, m, v, w):
        dp = [0] * (m + 1)
        for i in range(1, n + 1):
            for j in range(m, v[i] - 1, -1):
                dp[j] = max(dp[j], dp[j - v[i]] + w[i])
        return dp[-1]


if __name__ == '__main__':
    import sys

    n, m = map(int, input().split())
    lines = sys.stdin.readlines()
    v, w = [0], [0]
    n = 0
    for line in lines:
        line = list(map(int, line.split()))
        k = 1
        while k <= line[2]:  # 假设line[2]=13,k取1,2,4之后，line[2] = 6 < k = 8 退出循环
            v.append(k * line[0])
            w.append(k * line[1])
            line[2] -= k
            k *= 2
            n += 1  # 物品总数加1
        if line[2]:
            v.append(line[2] * line[0])
            w.append(line[2] * line[1])
            n += 1
    print(Solution().max_value(n, m, v, w))
```

二、并查集

```
class DjsSet:
    def __init__(self,N):
        self.parent=[i for i in range(N+1)]
        self.rank=[0 for i in range(N+1)]
    def find(self,x):
        if self.parent[x]==x:
            return x
        else:
            result=self.find(self.parent[x])
            self.parent[x]=result
            return result
    def union(self,x,y):
```

```
        xset=self.find(x)
        yset=self.find(y)
        if xset==yset:
            return
        if self.rank[xset]>self.rank[yset]:
            self.parent[yset]=xset
        else:
            self.parent[xset]=yset
            if self.rank[xset]==self.rank[yset]:
                self.rank[yset]+=1
```

三、图

1.Dijskra

```
def djs(start,end,graph):
    heap=[(0,start,[start])]
    heapq.heapify(heap)
    has_gone=set()
    while heap:
        (length,start,path)=heapq.heappop(heap)
        if start in has_gone:
            continue
        has_gone.add(start)
        if start==end:
            return path
        for i in graph[start]:
            if i not in has_gone:
                heapq.heappush(heap,(length+graph[start][i],i,path+[i]))
```

2.Prim

```
from collections import defaultdict
from heapq import *
def prim(vertexs, edges,start='D'):
    adjacent_dict = defaultdict(list) # 注意：defaultdict(list)必须以list做为变量
    for weight,v1, v2 in edges:
        adjacent_dict[v1].append((weight, v1, v2))
        adjacent_dict[v2].append((weight, v2, v1))
    minu_tree = []  # 存储最小生成树结果
    visited = [start] # 存储访问过的顶点，注意指定起始点
    adjacent_vertexs_edges = adjacent_dict[start]
    heapify(adjacent_vertexs_edges) # 转化为小顶堆，便于找到权重最小的边
    while adjacent_vertexs_edges:
        weight, v1, v2 = heappop(adjacent_vertexs_edges) # 权重最小的边，并同时从堆中删除。
        if v2 not in visited:
            visited.append(v2)  # 在used中有第一选定的点'A'，上面得到了距离A点最近的点'D',举例是5。将'd'追加到used中
            minu_tree.append((weight, v1, v2))
            # 再找与d相邻的点，如果没有在heap中，则应用heappush压入堆内，以加入排序行列
            for next_edge in adjacent_dict[v2]: # 找到v2相邻的边
                if next_edge[2] not in visited: # 如果v2还未被访问过，就加入堆中
                    heappush(adjacent_vertexs_edges, next_edge)
```

```
    return minu_tree

```

3.kruskal

```
def kruskal():
    n,m,tot_weight,graph=build()
    djsset=Dsjset(n)
    kruskal_weight=0
    cnt=0
    for edge in graph:
        if djsset.find(edge.start)!=djsset.find(edge.end):
            djsset.union(edge.start,edge.end)
            cnt+=1
            kruskal_weight+=edge.weight
        if cnt==n-1:
            break
    return tot_weight-kruskal_weight
```

4.topo-seq

```
class Node:
    def __init__(self,name):
        self.name=name
        self.indeg=0
        self.out=[]
    def __lt__(self,o):
        if self.indeg<o.indeg:
            return True
        elif self.indeg==o.indeg:
            return self.name<o.name
        else:
            return False
def build():
    n,m=map(int,input().split())
    graph=[Node(i) for i in range(n)]
    for _ in range(m):
        a,b=map(int,input().split())
        a,b=a-1,b-1
        graph[b].indeg+=1
        graph[a].out.append(graph[b])
    return n,graph
def topo_seq():
    import heapq
    n,graph=build()
    start=[]
    for i in range(n):
        if graph[i].indeg==0:
            start.append(graph[i])
    heapq.heapify(start)
    seq=[]
    while start:
        temp=heapq.heappop(start)
        seq.append(temp.name)
        for i in temp.out:
```

```
            i.indeg-=1
            if i.indeg==0:
                heapq.heappush(start,i)
        if len(seq)==n:
            return seq
```

5.判断图是否连通并查集

6.判断图是否成环

无向图并查集；有向图topo（不连通注意多起点）

7.关键路径和关键活动

```
## 拓扑排序和AOE网络问题
## 首先建立edge对象，依据数据得到邻接矩阵和被邻接矩阵
## 得到拓扑排序序列
## 依据拓扑排序序列得到时间最早和最晚开始时间和最快时长
## 确定关键事件，进而确定关键活动

class Edge:
    def __init__(self, e, w):
        self.e, self.w = e, w

    def __lt__(self, other):
        return self.e < other.e
n, m = map(int, input().split())
G = [[] for i in range(n)]
H = [[] for i in range(n)]
for _ in range(m):
    s, e, w = map(int, input().split())
    s = s-1
    e = e-1
    G[s].append(Edge(e, w))
    H[e].append(Edge(s, w))
inDegree = [0]*n
for i in range(n):
    inDegree[i] = len(H[i])
import queue
q = queue.Queue()
seq = []
for i in range(n):
    if inDegree[i] == 0:
        q.put(i)
while not q.empty():
    k = q.get()
    seq.append(k)
    for edge in G[k]:
        inDegree[edge.e] -= 1
        if inDegree[edge.e] == 0:
            q.put(edge.e)
earliest = [0] * n
for i in seq:
    for edge in G[i] :
        earliest[edge.e] = max(earliest[edge.e], earliest[i] + edge.w)
T = max(earliest)
```

```
latest = [T] * n
for j in seq[::-1]:
    for edge in H[j]:
        latest[edge.e] = min(latest[edge.e], latest[j] - edge.w)
event = []
for i in range(n):
    if earliest[i] == latest[i]:
        event.append(i)
event.sort()
print(T)
for i in event:
    G[i].sort()
    for edge in G[i]:
        if edge.e in event and abs(earliest[edge.e]-earliest[i]) == edge.w:
            print(i+1, edge.e+1)
```

四、树

0.预备知识

树的定义

定义一：树由节点及连接节点的边构成。树有以下属性： ❑ 有一个根节点； ❑ 除根节点外，其他每个节点都与其唯一的父节点相连； ❑ 从根节点到其他每个节点都有且仅有一条路径； ❑ 如果每个节点最多有两个子节点，我们就称这样的树为二叉树。

定义二：一棵树要么为空，要么由一个根节点和零棵或多棵子树构成，子树本身也是一棵树。每棵子树的根节点通过一条边连到父树的根节点。图3展示了树的递归定义。从树的递归定义可知，图中的树至少有4个节点，因为三角形代表的子树必定有一个根节点。这棵树或许有更多的节点，但必须更深入地查看子树后才能确定。

树的组成

节点 Node：节点是树的基础部分。 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础部分。 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； 每个节点可以有零条/一条/多条连到其它节点的出边。如果加限制不能有 "多条边"，这里树结构就特殊化为线性表

根节 Root: 树中唯一没有入边的节点。

路径 Path：由边依次连接在一起的有序节点列表。比如，哺乳纲→食肉目→猫科→猫属→家猫就是一条路径。

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连接节点的父节点。

兄弟节点 Sibling：具有同一父节点的节点之间为兄弟节点。

子树 Subtree：一个节点和其所有子孙节点，以及相关边的集合。

叶节点 Leaf Node：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。

高度Height ： 所有节点层级的最大值。

二叉树深度：从根节点到叶节点依次经过的节点形成的树的一条路径，最长路径的节点个数为树的深度。

特别注意：根据定义，**depth=height+1**

树的表示方法

（1）嵌套括号表示

例如：（A(D,E(F,G)),B,C)

（2）树形表示

（3）venn图表示

（4）凹入表

1.前序、中序、后序，层序遍历

```
def preorder(root):
    if root is None:
        return []
    result=[]
    result+=[root.val]
    for i in root.children:
        result+=preorder(i)
    return result
def inorder(root):
    result=[]
    if root is not None:
        result+=postorder(root.left)
        result+=[root.val]
        result+=postorder(root.right)
    return result
def postorder(root):
    result=[]
    if root is not None:
        result+=postorder(root.left)
        result+=postorder(root.right)
        result+=[root.val]
    return result
def level(root):
    if root is None:
        return []
    result=[]
    queue=[root]
    while queue:
        node=queue.pop(0)
        result.append(node.val)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return result
```

2.Huffman树/最优二叉树总路径权值最小

```python
import heapq
class Node:
    def __init__(self, weight, char=None):
        self.weight = weight
        self.char = char
        self.left = None
        self.right = None
    def __lt__(self, other):
        return self.weight<other.weight
def build_huffman_tree(characters):
    heap = []
    for char, weight in characters.items():
        heapq.heappush(heap, Node(weight, char))
    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(left.weight + right.weight)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)
    return heap[0]
def build_code(root):
    codes={}
    def traverse(node,code):
        if node.char:
            codes[node.char]=code
        else:
            traverse(node.left,code+'0')
            traverse(node.right,code+'1')
    traverse(root,'')
    return codes
def encoding(codes,string):
    encoded=''
    for char in string:
        encoded+=codes[char]
       return encoded
def decoding(root,encoded_string):
    decoded=''
    node=root
    for bit in encoded_string:
        if bit==0:
            node=node.left
        else:
            node=node.right
        if node.char:
            decoded+=node.char
            node=root
    return decoded
def external_path_length(node,depth=0):
    if node is None:
        return 0
    if node.left is None and node.right is None:
        return depth*node.weight
```

```
    return (external_path_length(node.left,depth+1)+external_path_length(node.right,depth+1))
n=int(input())
characters={}
lst=list(map(int,input().split()))
for i in range(len(lst)):
    characters[i]=lst[i]
root=build_huffman_tree(characters)
print(external_path_length(root))
```

3.堆

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
    def percUp(self, i):
        while i // 2 > 0:
            if self.heapList[i] < self.heapList[i // 2]:
                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2
    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
    def percDown(self, i):
        while (i * 2) <= self.currentSize:
            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:
                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc
    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1
    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval
    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
```

```
        while (i > 0):
            self.percDown(i)
            i = i - 1
n = int(input().strip())
bh = BinHeap()
for _ in range(n):
    inp = input().strip()
    if inp[0] == '1':
        bh.insert(int(inp.split()[1]))
    else:
        print(bh.delMin())
```

4.根据各序遍历建树

```
def buildtree(preorder,inorder):
    if not preorder or not inorder:
        return None
    root=Node(preorder[0])
    rootindex=inorder.index(root.val)
    root.left=buildtree(preorder[1:rootindex+1],inorder[:rootindex])
    root.right=buildtree(preorder[rootindex+1:],inorder[rootindex+1:])
    return root
def build(postorder,inorder):
    if not postorder or not inorder:
        return None
    root_val=postorder[-1]
    root=node(root_val)
    mid=inorder.index(root_val)
    root.left=build(postorder[:mid],inorder[:mid])
    root.right=build(postorder[mid:-1],inorder[mid+1:])
    return root
```

5.前缀树

```
class TrieNode:
    def __init__(self, char):
        self.char = char
        self.is_end = False
        self.children = {}
class Trie(object):
    def __init__(self):
        self.root = TrieNode("")
    def insert(self, word):
        node = self.root
        for char in word:
            if char in node.children:
                node = node.children[char]
            else:
                new_node = TrieNode(char)
                node.children[char] = new_node
                node = new_node
        node.is_end = True
    def dfs(self, node, pre):
        if node.is_end:
            self.output.append((pre + node.char))
```

```
        for child in node.children.values():
            self.dfs(child, pre + node.char)
    def search(self, x):
        node = self.root
        for char in x:
            if char in node.children:
                node = node.children[char]
            else:
                return []
        self.output = []
        self.dfs(node, x[:-1])
        #print(x)
        #print(self.output)
        return self.output
```

6.AVL树

当二叉搜索树不平衡时，get和put等操作的性能可能降到O(n)。本节将介绍一种特殊的二叉搜索树，它能**自动维持平衡。**这种树叫作 AVL树，以其发明者G. M. **A**delson-**V**elskii和E. M. **L**andis的姓氏命名。

实现：AVL树实现映射抽象数据类型的方式与普通的二叉搜索树一样，唯一的差别就是性能。实现AVL树时，要**记录每个节点的平衡因子。**我们**通过查看每个节点左右子树的高度来实现**这一点。更正式地说，我们**将平衡因子定义为左右子树的高度之差。**

$$BalanceFactor=height(leftSubTree)-height(rightSubTree)$$

根据上述定义，如果平衡因子大于零，我们称之为**左倾**；如果平衡因子小于零，就是**右倾**；如果平衡因子等于零，那么树就是**完全平衡**的。

为了实现AVL树并利用平衡树的优势，我们**将平衡因子为-1、0和1的树都定义为平衡树。**一旦某个节点的**平衡因子超出这个范围，我们就需要通过一个过程让树恢复平衡。**

假设现在已有一棵平衡二叉树，那么可以预见到，在往其中插入一个结点时，一定会有结点的平衡因子发生变化，此时可能会有结点的平衡因子的绝对值大于 1（这些平衡因子只可能是 2 或者 -2），这样以该结点为根结点的子树就是失衡的，需要进行调整。显然，只有在从根结点到该插入结点的路径上的结点才可能发生平衡因子变化，因此只需对这条路径上失衡的结点进行调整。

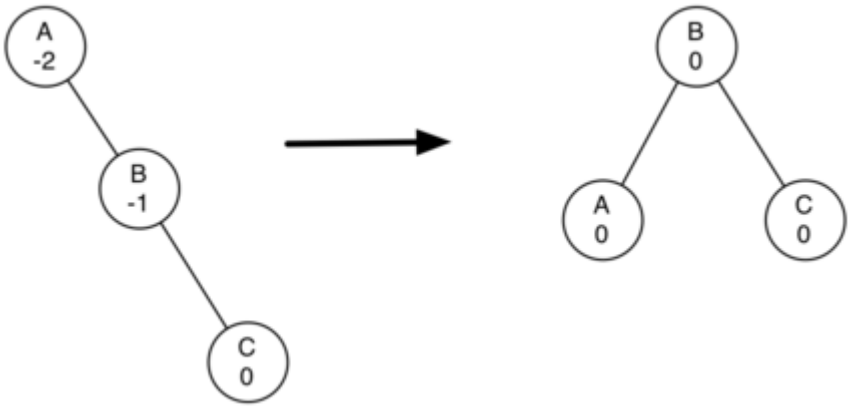
可以证明，**只要把最靠近插入结点的失衡结点调整到正常，路径上的所有结点就都会平衡。**

AVL的两种旋转操作

- 左旋

  如果需要进行再平衡，该怎么做呢？高效的再平衡是让AVL树发挥作用同时不损性能的关键。为了让AVL树恢复平衡，需要在树上进行一次或多次旋转。

  要理解什么是旋转，来看一个简单的例子。考虑图2中左边的树。这棵树失衡了，平衡因子是-2。要让它恢复平衡，我们围绕以节点A为根节点的子树做一次左旋。



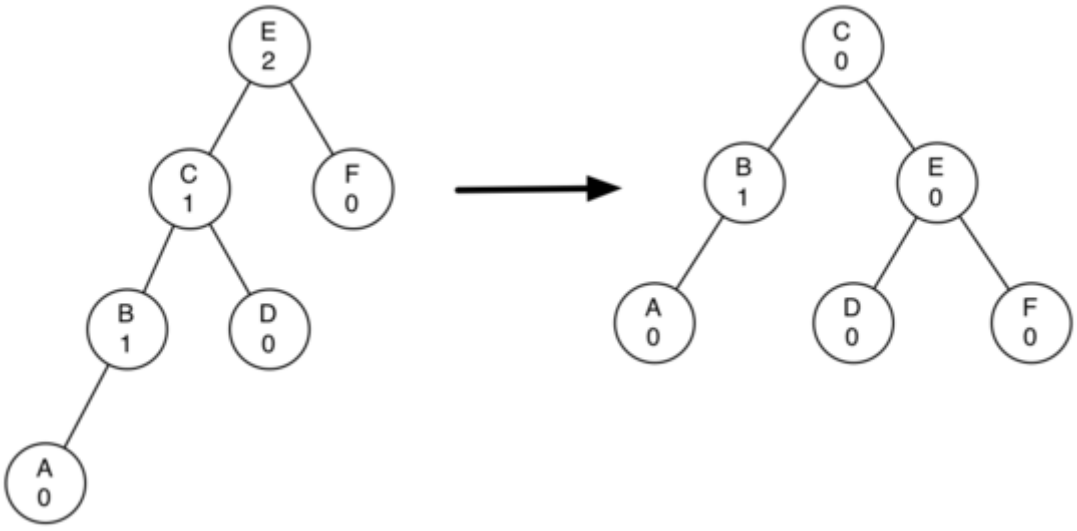
本质上，左旋包括以下步骤。

1. **将右子节点（节点B）提升为子树的根节点。**
2. **将旧根节点（节点A）作为新根节点的左子节点。**
3. **如果新根节点（节点B）已经有一个左子节点，将其作为新左子节点（节点A）的右子节点。**

注意，因为节点B之前是节点A的右子节点，所以此时节点A必然没有右子节点。因此，可以为它添加新的右子节点，而无须过多考虑。

- 右旋

我们来看一棵稍微复杂一点的树，并理解右旋过程。图4左边的是一棵左倾的树，根节点的平衡因子是2。右旋步骤如下。



1. **将左子节点（节点C）提升为子树的根节点。**
2. **将旧根节点（节点E）作为新根节点的右子节点。**
3. **如果新根节点（节点C）已经有一个右子节点（节点D），将其作为新右子节点（节点E）的左子节点。注意，因为节点C之前是节点E的左子节点，所以此时节点E必然没有左子节点。因此，可以为它添加新的左子节点，而无须过多考虑。**

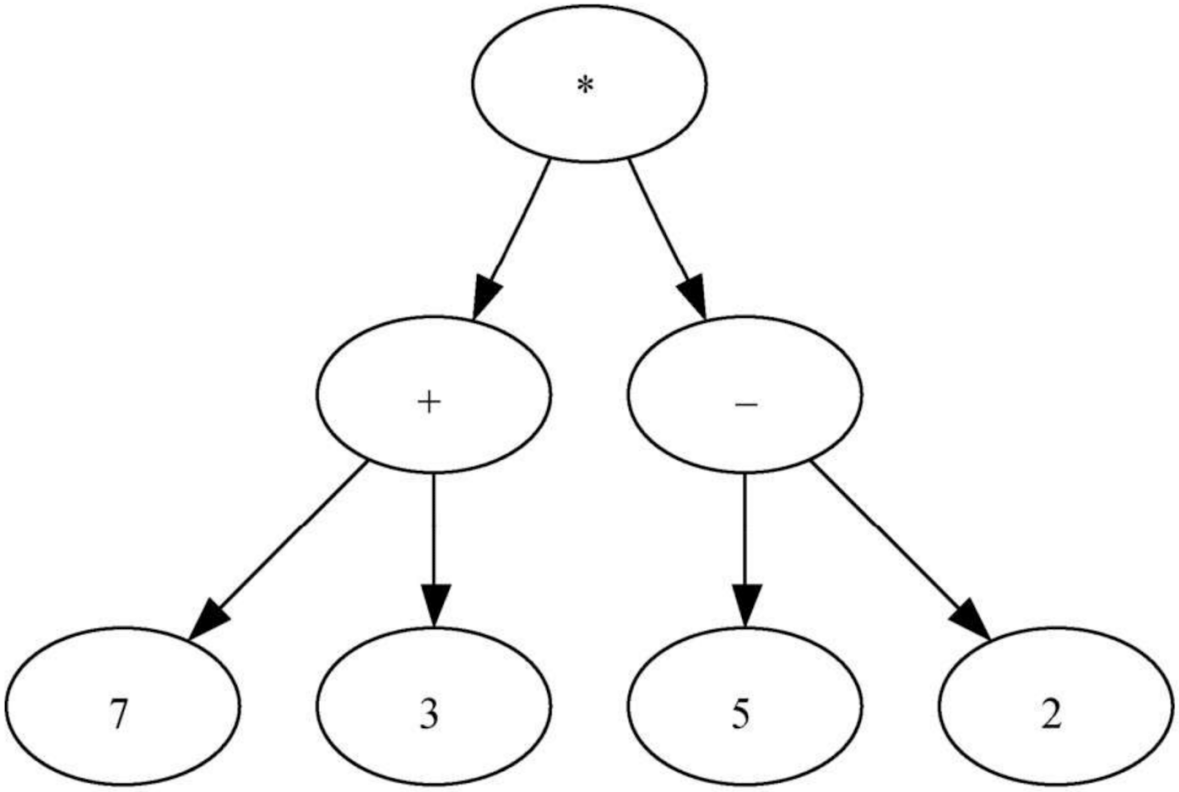
四种树形以及调整方法

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1
```

```
class AVL:
    def __init__(self):
        self.root = None
    def insert(self, value):
        if not self.root:
            self.root = Node(value)
        else:
            self.root = self._insert(value, self.root)
    def _insert(self, value, node):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(value, node.left)
        else:
            node.right = self._insert(value, node.right)
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        balance = self._get_balance(node)
        if balance > 1:
            if value < node.left.value: # 树形是 LL
                return self._rotate_right(node)
            else:   # 树形是 LR
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)
        if balance < -1:
            if value > node.right.value:    # 树形是 RR
                return self._rotate_left(node)
            else:   # 树形是 RL
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)
        return node
    def _get_height(self, node):
        if not node:
            return 0
        return node.height
    def _get_balance(self, node):
        if not node:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)
    def _rotate_left(self, z):
        y = z.right
        T2 = y.left
        y.left = z
        z.right = T2
        z.height = 1 + max(self._get_height(z.left),
self._get_height(z.right))
        y.height = 1 + max(self._get_height(y.left),
self._get_height(y.right))
        return y
    def _rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self._get_height(y.left),
self._get_height(y.right))
```

```
59          x.height = 1 + max(self._get_height(x.left),
self._get_height(x.right))
60          return x
61      def preorder(self):
62          return self._preorder(self.root)
63      def _preorder(self, node):
64          if not node:
65              return []
66          return [node.value] + self._preorder(node.left) +
self._preorder(node.right)
```

7.解析树



构建解析树

构建解析树的第一步是将表达式字符串拆分成标记列表。

需要考虑4种标记：左括号、右括号、运算符和操作数。

我们知道，左括号代表新表达式的起点，所以应该创建一棵对应该表达式的新树。

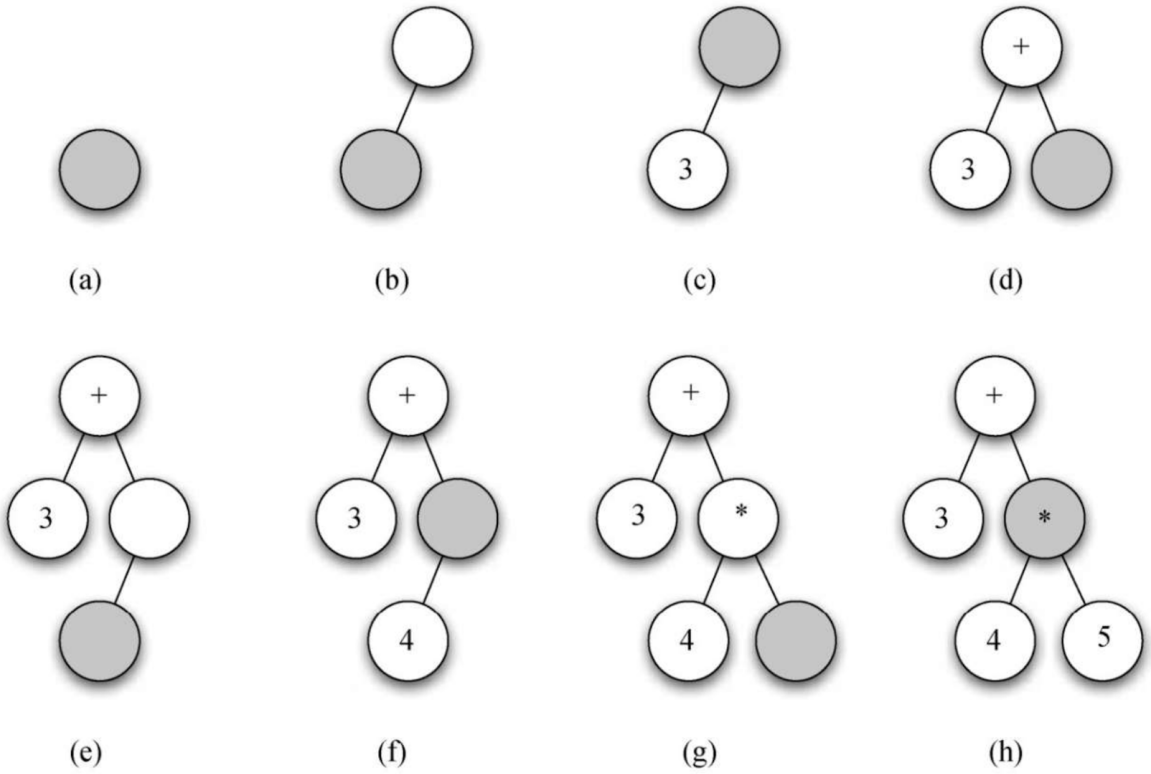
反之，遇到右括号则意味着到达该表达式的终点。我们也知道，操作数既是叶子节点，也是其运算符的子节点。

此外，每个运算符都有左右子节点。

有了上述信息，便可以定义以下4条规则：

(1) 如果当前标记是"("，就为当前节点添加一个左子节点，并下沉至该子节点； (2) 如果当前标记在列表`['+', '-', '/', '*']`中，就将当前节点的值设为当前标记对应的运算符；为当前节点添加一个右子节点，并下沉至该子节点； (3) 如果当前标记是数字，就将当前节点的值设为这个数并返回至父节点；(4) 如果当前标记是)，就跳到当前节点的父节点。

以（3+（4*5））为例。




```python
class Stack(object):
    def __init__(self):
        self.items = []
        self.stack_size = 0

    def isEmpty(self):
        return self.stack_size == 0

    def push(self, new_item):
        self.items.append(new_item)
        self.stack_size += 1

    def pop(self):
        self.stack_size -= 1
        return self.items.pop()

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

    def size(self):
        return self.stack_size


class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:  # 已经存在左子节点。此时，插入一个节点，并将已有的左子节点降一层。
            t = BinaryTree(newNode)
```

```
            t.leftChild = self.leftChild
            self.leftChild = t
    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t
    def getRightChild(self):
        return self.rightChild
    def getLeftChild(self):
        return self.leftChild
    def setRootVal(self, obj):
        self.key = obj
    def getRootVal(self):
        return self.key
    def traversal(self, method="preorder"):
        if method == "preorder":
            print(self.key, end=" ")
        if self.leftChild != None:
            self.leftChild.traversal(method)
        if method == "inorder":
            print(self.key, end=" ")
        if self.rightChild != None:
            self.rightChild.traversal(method)
        if method == "postorder":
            print(self.key, end=" ")
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("Unknown Operator: " + i)
    return eTree
exp = "( ( 7 + 3 ) * ( 5 - 2 ) )"
pt = buildParseTree(exp)
for mode in ["preorder", "postorder", "inorder"]:
```

```
    pt.traversal(mode)
    print()
"""
* + 7 3 - 5 2
7 3 + 5 2 - *
7 + 3 * 5 - 2
"""
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()
print(evaluate(pt))
# 30

#后序求值
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()

print(postordereval(pt))
# 30

#中序还原完全括号表达式
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal

print(printexp(pt))
# (((7)+3)*((5)-2))
```

8.二叉搜索树

二叉搜索树（Binary Search Tree，BST），它是映射的另一种实现。我们感兴趣的不是元素在树中的确切位置，而是如何利用二叉树结构提供高效的搜索。

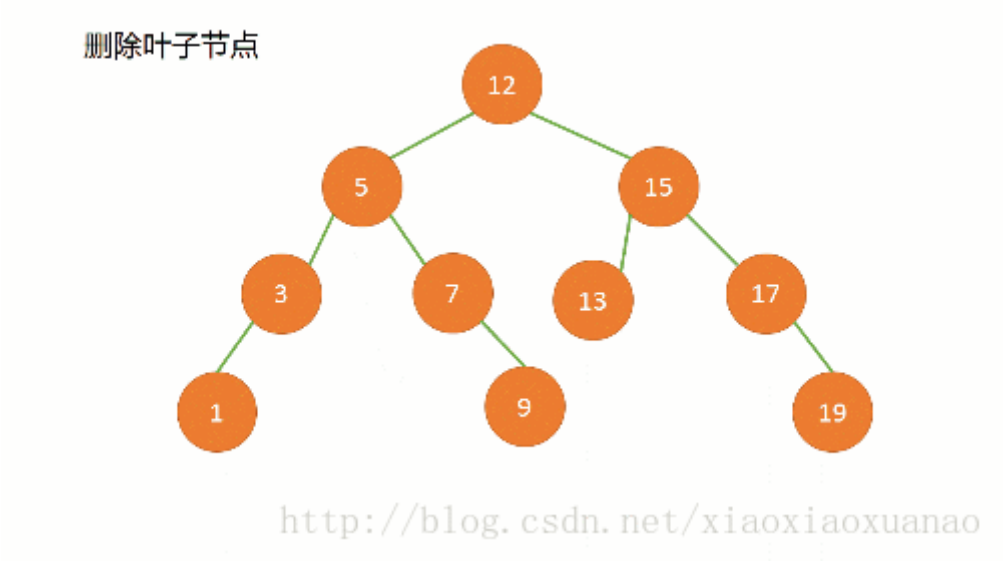
二叉搜索树依赖于这样一个性质：小于父节点的键都在左子树中，大于父节点的键则都在右子树中。我们称这个性质为二叉搜索性。

依赖于这一特性，二叉搜索树的中序遍历是有序的。通过这个办法可以实现树排序，平均时间复杂度为$nlogn$.但是当树的平衡性很差时时间复杂度会坍塌到$O(n^2)$.

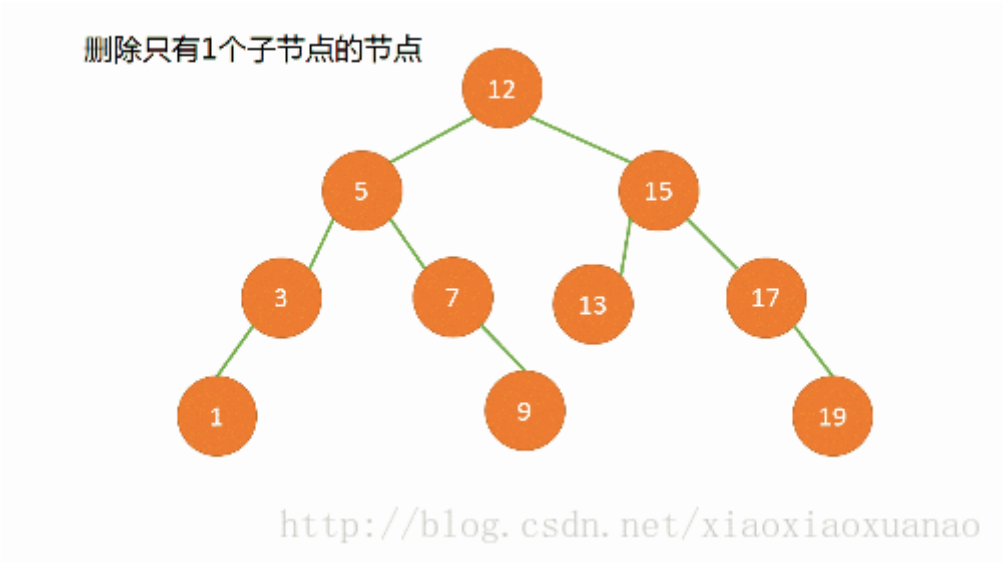
二叉搜索树中节点的删除

二叉查找树删除节点可以分成三种情况：
（1）删除叶子节点

叶子节点删除是最简单的情况，由于叶子节点没有左右子树，删除后不会破坏原有的树形结构，所以我们只需要找到节点并且把它置为null即可。

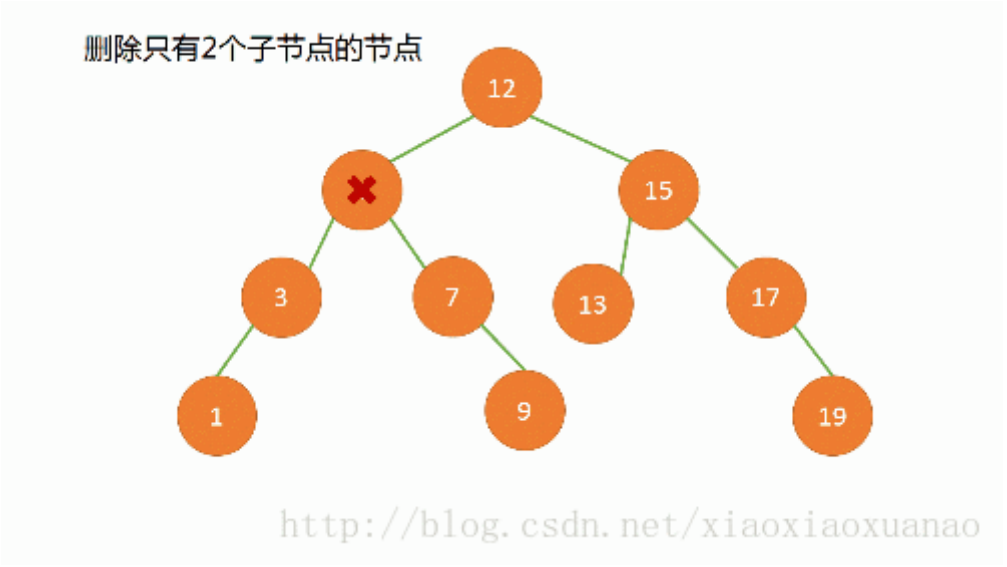


（2）被删除的节点只有一个子节点
比如我们要删除上图中3所在的节点，3只有一个左子树1。
实际上我们只需要把5所在节点的左子树指向原来3的左子树即可。



（3）被删除的节点左右子树都有
这种情况是比较复杂的，为了不破坏二叉查找书的结构，我们可以按照以下操作进行：

- 找出左子树中最大或者右子树中最小的值val
- 将当前节点的值替换为val
- 在左子树或者右子树中找到val删除



五、排序算法

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

1.归并排序，求逆序对数

```
def merge_sort(lst):
    l=len(lst)
    if l<=1:
        return lst,0
    middle=l//2
    left=lst[:middle]
    right=lst[middle:]
    merged_left,left_inv=merge_sort(left)
    merged_right,right_inv=merge_sort(right)
    merged,merge_inv=merge(merged_left,merged_right)
    return merged,merge_inv+left_inv+right_inv
def merge(left,right):
    i=j=0
    merge_inv=0
```

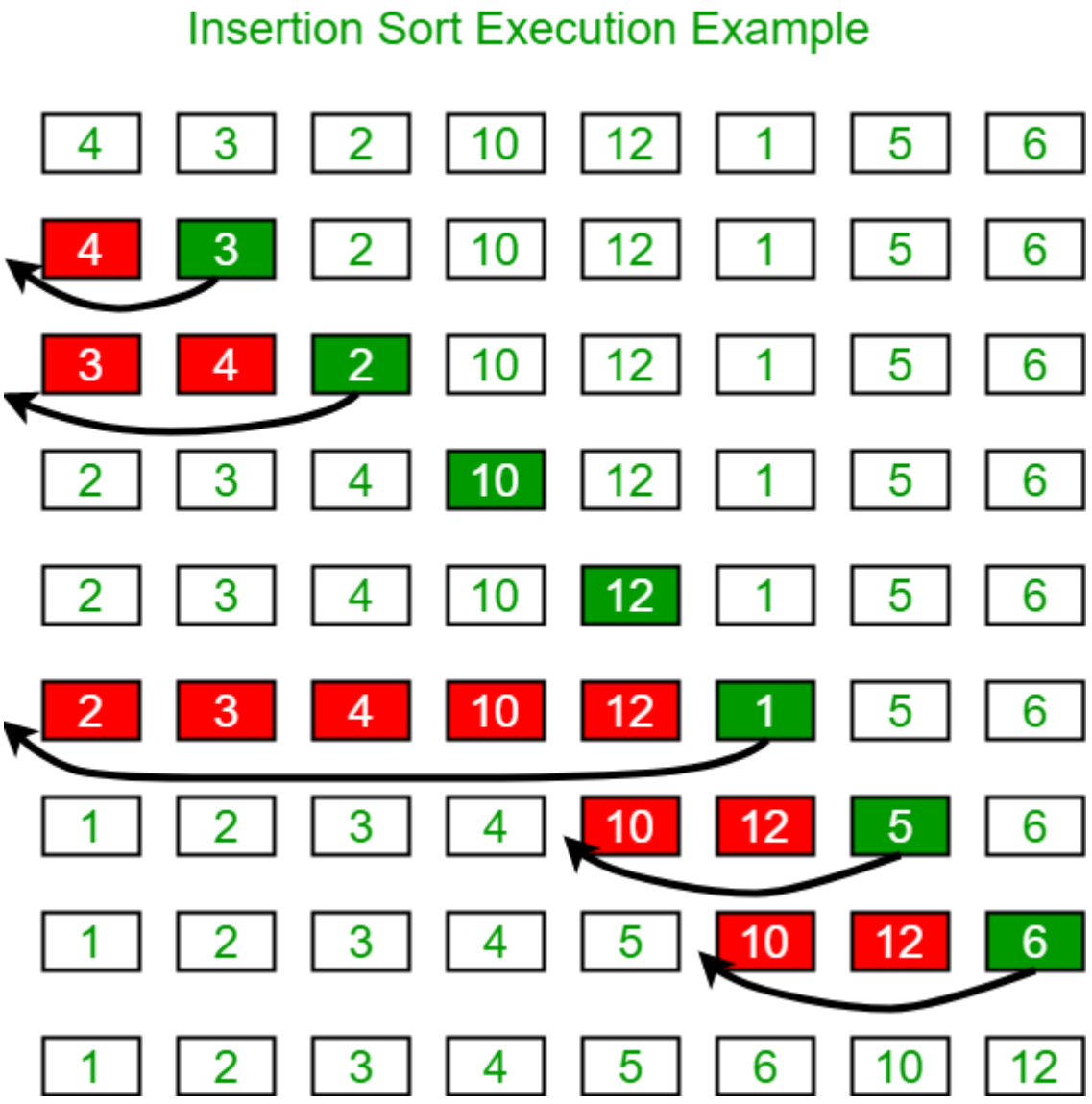
```
    merged=[]
    while i<len(left) and j<len(right):
        if left[i]<=right[j]:
            merged.append(left[i])
            i+=1
        else:
            merged.append(right[j])
            j+=1
            merge_inv+=len(left)-i
    merged+=left[i:]
    merged+=right[j:]
    return merged,merge_inv
```

2.插入排序

1） 将序列分成有序的部分和无序的部分。有序的部分在左边，无序的部分在右边。开始有序部分只有1个元素

2） 每次找到无序部分的最左元素（设下标为i），将其插入到有序部分的合适位置(设下标为k,则原下标为k到i-1的元素都右移一位)，有序部分元素个数+1

3） 直到全部有序

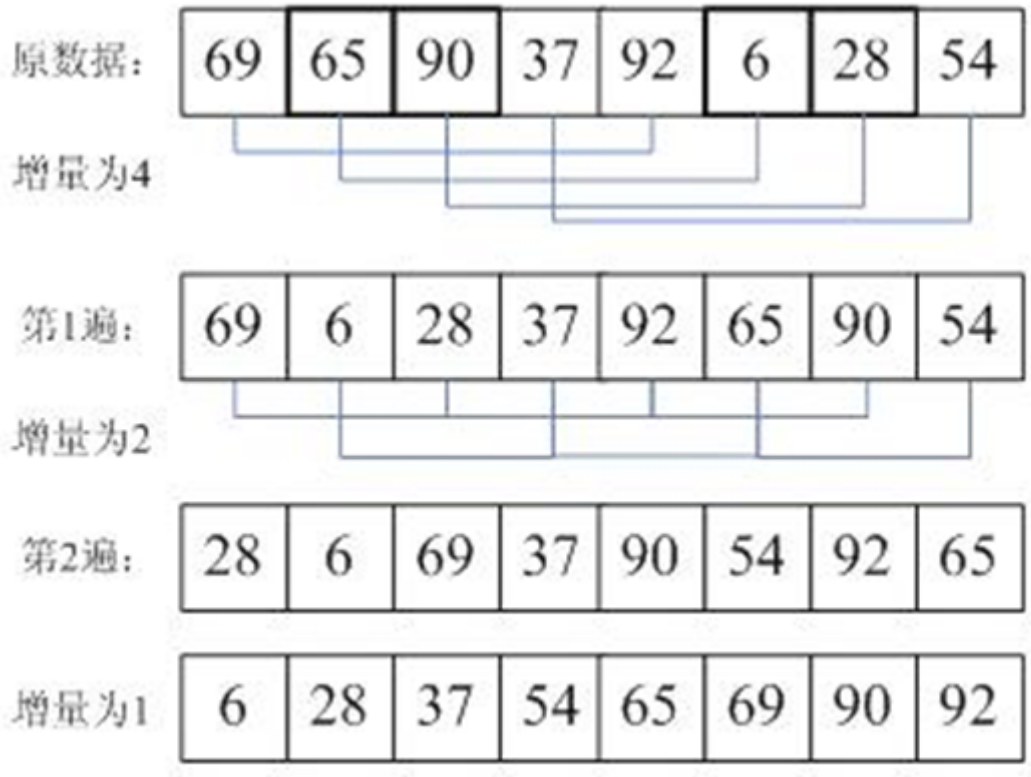


```python
def insertion_sort(arr):
    for i in range(1,len(arr)):
        j=i
        while arr[j-1]>arr[j] and j>0:
            arr[j-1],arr[j]=arr[j],arr[j-1]
```

改进：二分法找插入位置寻找插入位置时，用二分法。总体复杂度量级没有改进，因为要移动元素。且为了保证稳定性，插入位置必须是最后一个相等元素的后面。

- 规模很小的排序可优先选用(比如，元素个数10以内)
- 特别适合元素基本有序的情况(复杂度接近O(n))
- 许多算法会在上述两种情况下采用插入排序。例如改进的快速排序算法、归并排序算法，在待排序区间很小的时候就不再递归快排或归并，而是用插入排序

3.一种改进的插入排序：希尔排序（Shell)



1)选取增量(间隔)为D，根据增量将列表分为多组，每组分别插入排序:

第一组：A0 , A0+D , A0+2D, ……

第二组：A1 , A1+D , A1+2D, ……

第三组：A2 , A2+D , A2+2D, ……

若D==1，则插入排序后，整个排序结束

2)D = D//2 ， 转1

初始增量D可以为 n//2, n是元素总数

也许D还可以有别的选取法

最好：O(n)，平均O(n1.5)，最坏O(n2)

4.冒泡排序

In Bubble Sort algorithm:

- traverse from left and compare adjacent elements and the higher one is placed at right side. 从左到右扫一遍临项比较，大的置于右侧。
- In this way, the largest element is moved to the rightmost end at first. 每次扫完，当前最大的在最右侧。
- This process is then continued to find the second largest and place it and so on until the data is sorted.

```
def bubble_sort(arr):
    n=len(arr)
    for i in range(n):
        swapped=False #改进
        for j in range(0,n-i-1):
            if arr[j]>arr[j+1]:
                arr[j],arr[j+1]=arr[j+1],arr[j]
                swapped=True#如果发现某一轮扫描时，没有发生元素交换的情况，则说明已经排好序了，就不用再扫描了
        if swapped==False:
            break
```

- 无论最好、最坏、平均，语句(1）必定执行(n-1)+...+3+2+1次，复杂度O(n2)
- 稳定性：稳定
- 额外空间：O(1)

改进：最好情况，即基本有序时，可以做到O(n)**

5.选择排序

基本思想

1） 将序列分成有序的部分和无序的部分。有序的部分在左边，无序的部分在右边。开始有序部分没有元素

2） 每次找到无序部分的最小元素（设下标为i），和无序部分的最左边元素（设下标为j)交换。有序部分元素个数+1。

3） 2)做n-1次，排序即完成

```
def selection_sort(a):
    n=len(a)
    for i in range(n-1):
        minPos=i
        for j in range(i+1,n):
            if a[j]<a[minPos]:
                minPos=j
        if minPos!=i:
            a[minPos],a[i]=a[i],a[minPos]
```

- 无论最好、最坏、平均，语句(1）必定执行(n-1)+...+3+2+1次，复杂度O(n2)
- 稳定性：不稳定，因a[i]被交换时，可能越过了其后面一些和它相等的元素

- 额外空间：O(1)

平均效率低于插入排序，没啥实际用处

6.快速排序

数组排序任务可以如下完成：

- 设k=a[0], 将k挪到适当位置，使得比k小的元素都在k左边,比k大的元素都在k右边，和k相等的，不关心在k左右出现均可 （O（n)时间完成）
- 把k左边的部分快速排序
- 把k右边的部分快速排序

```
def quick_sort(arr,s,e):#将arr[s:e+1]进行排序
    if s>=e:
        return
    i,j=s,e
    while i!=j:
        while i<j and a[i]<=a[j]:
            j-=1
        a[i],a[j]=a[j],a[i]
        while i<j and a[i]<=a[j]:
            i+=1
        a[i],a[j]=a[j],a[i]
    quick_sort(arr,s,i-1)
    quick_sort(arr,i+1,e)
```

- 最坏情况(已经基本有序或倒序): O(n2)
- 平均情况: O(nlog(n))
- 最好情况: O(nlog(n))
- 稳定性：不稳定
- 额外空间：两次递归的普通写法：最坏情况需要递归n层，需要n层栈空间，复杂度O(n)。最好情况和平均情况递归log(n)层，复杂度O(log(n))

7.堆排序

1)将待排序列表a变成一个堆(O(n))

2)将a[0]和a[n-1]交换，然后对新a[0]做下移，维持前n-1个元素依然是堆。此时优先级最高的元素就是a[n-1]

3)将a[0]和a[n-2]交换，然后对新a[0]做下移，维持前n-2个元素依然是堆。此时优先级次高的元素就是a[n-2]

……

直到堆的长度变为1，列表a就按照优先级从低到高排好序了。

整个过程相当不断删除堆顶元素放到a的后部。堆顶元素依次是优先级最高的、次高的....

一共要做n次下移，每次下移O(log(n))，因此总复杂度O(nlog(n))

如果用递归实现，需要O(log(n))额外栈空间(递归要进行log(n)层)。

如果不用递归实现，需要O(1)额外空间。

```python
import heapq
def heapSorted(iterable): #iterable是个序列
#函数返回一个列表，内容是iterable中元素排序的结果，不会改变iterable
    h = []
    for value in iterable:
        h.append(value)
    heapq.heapify(h)  #将h变成一个堆
    return [heapq.heappop(h) for i in range(len(h))]
a = (2,13,56,31,5)
print(heapSorted(a)) #>>[2, 5, 13, 31, 56]
print(heapq.nlargest(3,a)) #>>[56, 31, 13]
print(heapq.nlargest(3,a,lambda x:x%10))
#>>[56, 5, 13] 取个位数最大的三个
print(heapq.nsmallest(3,a,lambda x:x%10))
#>>[31, 2, 13] 取个位数最小的三个
```

```python
def heapSort(a,key = lambda x:x): #对列表a进行排序
    def makeHeap(): #建堆
        i = (heapSize - 1 - 1) // 2 #i是最后一个叶子的父亲
        for k in range(i,-1,-1):
            shiftDown(k)
    def shiftDown(i): #a[i]下移
        while i * 2 + 1 < heapSize:  #只要a[i]有儿子就做
            L,R = i * 2 + 1, i * 2 + 2
            if R >= heapSize or key(a[L]) < key(a[R]):
                s = L
            else:
                s = R
            if key(a[s]) < key(a[i]):
                a[i],a[s] = a[s],a[i]
                i = s
            else:
                break
    heapSize = len(a)
    makeHeap()
    for i in range(len(a)-1,0,-1):
        a[i],a[0] = a[0],a[i]
        heapSize -= 1
        shiftDown(0)
    n = len(a)
    for i in range(n//2): #颠倒a
        a[i],a[n-1-i] = a[n-1-i],a[i]

a = [3,21,4,76,12,3]
heapSort(a)
print(a) #>>[3, 3, 4, 12, 21, 76]
```

- 最坏情况: O(nlog(n))
- 平均情况: O(nlog(n))
- 最好情况: O(nlog(n))
- 稳定性：不稳定
- 额外空间：O(1) (可以用非递归写法，或编译器、解释器自动尾递归优化)

8.桶排序，基数排序

- 如果待排序元素只有m种不同取值，且m很小（比如考试分数只有0-100),可以采用桶排序
- 设立m个桶，分别对应m种取值。桶和桶可以比大小，桶的大小就是其对应取值的大小。把元素依次放入其对应的桶，然后再按先小桶后大桶的顺序，将元素都收集起来，即完成排序
- 复杂度O(n+m)，且稳定。n是待排序元素个数。
- 额外空间：O(n+m)

例如：将考试分数分到0-100这101个桶里面，然后按照0、1、2...100的顺序收集桶里的分数，即完成排序

```
def bucketSort(s,m,key=lambda x:x):
    buckets = [[] for i in range(m)]
    for x in s:
        buckets[key(x)].append(x)
    i = 0
    for bkt in buckets:
        for e in bkt:
            s[i] = e
            i += 1
lst = [2, 3, 4, 8, 9, 12, 3, 2, 4, 12]
bucketSort(lst, 13)
print(lst) #>>[2, 2, 3, 3, 4, 4, 8, 9, 12, 12]
lst = [(-2, "Jack"), (8, "Mike"), (2, "Jane"), (2, "John")]
bucketSort(lst, 12, lambda x: x[0]+2)  # 取值范围写大一些也无妨
print(lst) #>>[(-2, 'Jack'), (2, 'Jane'), (2, 'John'), (8, 'Mike')]
lst = ["Jack", "Mike", "Lany", "Ada"]
bucketSort(lst, 26, lambda x: ord(x[0])-ord("A"))
print(lst) #>>['Ada', 'Jack', 'Lany', 'Mike']
```

多轮分配排序（基数排序）

1)将待排序元素看作由相同个数的原子构成的元组（$e_1, e_2 \ldots e_n$)。长度不足的元素，用最小原子补齐左边空缺的部分。

2)原子种类必须很少。有n种原子，就设立n个桶

3)先按e_n将所有元素分配到各个桶里，然后从小桶到大桶收集所有元素，得到序列1,然后将序列1按e_n-1分配到各个桶里再收集成为序列2.....直到按e_0分配到桶再完成收集得到序列n，序列n就是最终排序结果。

多轮分配排序（基数排序）

- **对序列 73,22,93,43,55,14,28,65,39,81 排序**

- 根据个位数，将每个数分配到0到9号桶
- 按桶号由小到大收集这些数，得到序列1：
81,22,73,93,43,14,55,65,28,39
- 按顺序将序列1中的每个数，根据十位数，重新分配到0到9号桶
- 按桶号由小到大收集这些数，得到序列2：
14,22,28,39,43,55,65,73,81,93

- **基数排序的复杂度是 O(d*(n+radix))**
  n：要排序的元素的个数(假设每个元素由若干个原子组成)
  radix: 桶的个数，即组成元素的原子的种类数
  d: 元素最多由多少个原子组成

  对序列 73,22,93,43,55,14,28,65,39,81 排序:
  n = 10, d = 2, radix = 10(或9)

- 一共要做 d 轮分配和收集
- 每一轮, 分配的复杂度 O(n)，收集的复杂度O(radix)
  （一个桶里的元素可以用链表存放，便于快速搜集）
- 总复杂度 O(d * (n + radix))

```python
def radixSort(s, m, d, key):
    #key(x,k)可以取元素x的第k位原子
    for k in range(d):
        buckets = [[] for j in range(m)]
        for x in s:
            buckets[key(x, k)].append(x)
        i = 0
        for bkt in buckets: #这样收集复杂度O(len(s))
            for e in bkt:
                s[i] = e
                i += 1
def getKey(x, i):
    #取非负整数x的第i位。个位是第0位
    tmp = None
    for k in range(i + 1):
        tmp = x % 10
        x //= 10
    return tmp

lst = [123,21,48,745,143,62,269,87,300,6]
```

```
radixSort(lst, 10, 3, getKey)
print(lst) #>>[6, 21, 48, 62, 87, 123, 143, 269, 300, 745]
```

六、二分搜索法

```
def check(mincost,m,prices):
    cnt=1
    i=0
    temp=0
    n=len(prices)
    if max(prices)>mincost:
        return False
    while i<n:
        temp+=prices[i]
        if temp>mincost:
            cnt+=1
            temp=prices[i]
        i+=1
    return cnt<=m

n,m=map(int,input().split())
prices=[]
for i in range(n):
    prices.append(int(input()))
min=0
maxs=sum(prices)
mid=(min+maxs)//2+1
while mid<maxs:
    if check(mid,m,prices):
        maxs=mid
        mid=(min+mid)//2+1
    else:
        min=mid
        mid=(maxs+mid)//2+1
if check(mid-1,m,prices):
    print(mid-1)
elif check(mid,m,prices):
    print(mid)
else:
    print(mid+1)
```

七、线性结构（部分内容引用自武昱达同学的总结）

1.线性表

线性表是一种线性结构（**逻辑结构**，在说线性表的时候不考虑具体实现办法），常见的有**数组**和**链表**。

- **数组**是一种**连续存储结构**，它将线性表的元素按照一定的顺序依次存储在内存中的**连续地址空间**上。数组需要预先分配一定的内存空间，每个元素占用相同大小的内存空间，并可以通过索引来进行快速访问和操作元素。访问元素的时间复杂度为O(1)，因为可以直接计算元素的内存地址。然而，插入和删除元素的时间复杂度较高，平均为O(n)，因为需要**移动其他元素来保持连续存储的特性**。

- **链表**是一种存储结构，它是线性表的链式存储方式。链表通过节点的相互链接来实现元素的存储。每个节点包含元素本身以及指向下一个节点的指针。**链表的插入和删除操作非常高效，时间复杂度为O(1)**，因为只需要调整节点的 指针。然而，访问元素的时间复杂度较高，平均为O(n)，因为必须从头节点开始遍历链表直到找到目标元素。

线性表	插入复杂度	查找复杂度
数组	O（n）	O（1）
链表	O（1）	O（n）

链表是一种常见的数据结构，用于存储和组织数据。它**由一系列节点组成，每个节点包含一个数据元素和一个指向下一个节点（或前一个节点）的指针。**

在链表中，每个节点都包含两部分：

1. **数据元素（或数据项）**：这是节点存储的实际数据。可以是任何数据类型，例如整数、字符串、对象等。
2. **指针（或引用）**：该指针指向链表中的下一个节点（或前一个节点）。它们用于建立节点之间的连接关系，从而形成链表的结构。

根据指针的类型和连接方式，链表可以分为不同类型，包括：

1. **单向链表（单链表）**：每个节点只有一个指针，指向下一个节点。链表的头部指针指向第一个节点，而最后一个节点的指针为空（指向 `None`）。
2. **双向链表**：每个节点有两个指针，一个指向前一个节点，一个指向后一个节点。双向链表可以从头部或尾部开始遍历，并且可以在任意位置插入或删除节点。
3. **循环链表**：最后一个节点的指针指向链表的头部，形成一个环形结构。循环链表可以从任意节点开始遍历，并且可以无限地循环下去。

链表相对于数组的一个重要特点是，链表的大小可以动态地增长或缩小，而不需要预先定义固定的大小。这使得链表在需要频繁插入和删除元素的场景中更加灵活。

然而，链表的访问和搜索操作相对较慢，因为需要遍历整个链表才能找到目标节点。与数组相比，链表的优势在于插入和删除操作的效率较高，尤其是在操作头部或尾部节点时。因此，链表在需要频繁插入和删除元素而不关心随机访问的情况下，是一种常用的数据结构

单链表的实现

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
```

```
    def delete(self, value):
        if self.head is None:
            return
        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next
    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print()
# 使用示例
linked_list = LinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()  # 输出：1 2 3
linked_list.delete(2)
linked_list.display()  # 输出：1 3
```

双向链表实现

```
class Node:
    def __init__(self, value):
        self.value = value
        self.prev = None
        self.next = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def insert_before(self, node, new_node):
        if node is None:  # 如果链表为空，将新节点设置为头部和尾部
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = node
            new_node.prev = node.prev
            if node.prev is not None:
                node.prev.next = new_node
            else:  # 如果在头部插入新节点，更新头部指针
                self.head = new_node
            node.prev = new_node
    def display_forward(self):
        current = self.head
        while current is not None:
            print(current.value, end=" ")
            current = current.next
```

```
        print()
    def display_backward(self):
        current = self.tail
        while current is not None:
            print(current.value, end=" ")
            current = current.prev
        print()
# 使用示例
linked_list = DoublyLinkedList()
# 创建节点
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
# 将节点插入链表
linked_list.insert_before(None, node1)  # 在空链表中插入节点1
linked_list.insert_before(node1, node2)  # 在节点1前插入节点2
linked_list.insert_before(node1, node3)  # 在节点1前插入节点3
# 显示链表内容
linked_list.display_forward()  # 输出：3 2 1
linked_list.display_backward()  # 输出：1 2 3
```

2.栈与队列

栈和队列可以认为是特殊线性表。

栈：后进先出，Python中容易实现。

队列：先进先出，Python中容易实现

```
# stack类的标准方法
s.isEmpty()
s.push(1)
s.peek()
s.size()
s.pop()
```

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```

Shunting Yard算法 中缀表达式转后缀表达式

Dijkstra Shunting Yard 调度场算法的主要思想是使用两个栈（运算符栈和输出栈）来处理表达式的符号。算法按照运算符的优先级和结合性，将符号逐个处理并放置到正确的位置。最终，输出栈中的元素就是转换后的后缀表达式。

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：
        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - 将当前运算符推入运算符栈。
    - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

```python
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''
    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <= precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()
    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)
    while stack:
        postfix.append(stack.pop())
    return ' '.join(str(x) for x in postfix)
```

队列的实现

```
class Queue:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0, item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)
```

双端队列实现

```
class Deque:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def addFront(self, item):
        self.items.append(item)
    def addRear(self, item):
        self.items.insert(0, item)
    def removeFront(self):
        return self.items.pop()
    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

八、散列表

0.预备知识

线性表是一种具有相同数据类型的有限序列，其特点是每个元素都有唯一的直接前驱和直接后继。换句话说，线性表中的元素之间存在明确的线性关系，每个元素都与其前后相邻的元素相关联。

线性结构是数据结构中的一种基本结构，它的特点是数据元素之间存在一对一的关系，即除了第一个元素和最后一个元素以外，其他每个元素都有且仅有一个直接前驱和一个直接后继。线性结构包括线性表、栈、队列和串等。

因此，线性表是线性结构的一种具体实现，它是一种最简单和最常见的线性结构。

在查找过程中只考虑各元素关键字之间的相对大小，记录在存储结构中的位置和其关键字无直接关系，其查找时间与表的长度有关，特别是当结点个数很多时，查找时要大量地与无效结点的关键字进行比较，致使查找速度很慢。如果能**在元素的存储位置和其关键字之间建立某种直接关系**，那么在进行查找时，就无需做比较或做很少次的比较，按照这种关系直接由关键字找到相应的记录。这就是**散列查找法**（Hash Search）的思想，它通过对元素的关键字值进行某种运算，直接求出元素的地址，即使用关键字到地址的直接转换方法，而不需要反复比较。因此，散列查找法又叫杂凑法或散列法。

1.散列函数的构造方法

构造散列函数的方法很多，一般来说，应根据具体问题选用不同的散列函数，通常要考虑以下因素:

(1) 散列表的长度; (2) 关键字的长度; (3) 关键字的分布情况; (4) 计算散列函数所需的时间; (5) 记录的查找频率。

构造一个“好”的散列函数应遵循以下两条原则:

(1)函数计算要简单，每一关键字只能有一个散列地址与之对应;

(2)函数的值域需在表长的范围内，计算出的散列地址的分布应均匀，尽可能减少冲突。

（1）数字分析法

如果**事先知道关键字集合**，且**每个关键字的位数比散列表的地址码位数多**，每个关键字由n位数组成，如$k_1, k_2, \ldots k_n$，则可以**从关键字中提取数字分布比较均匀的若干位作为散列地址。**

（2）平方取中法

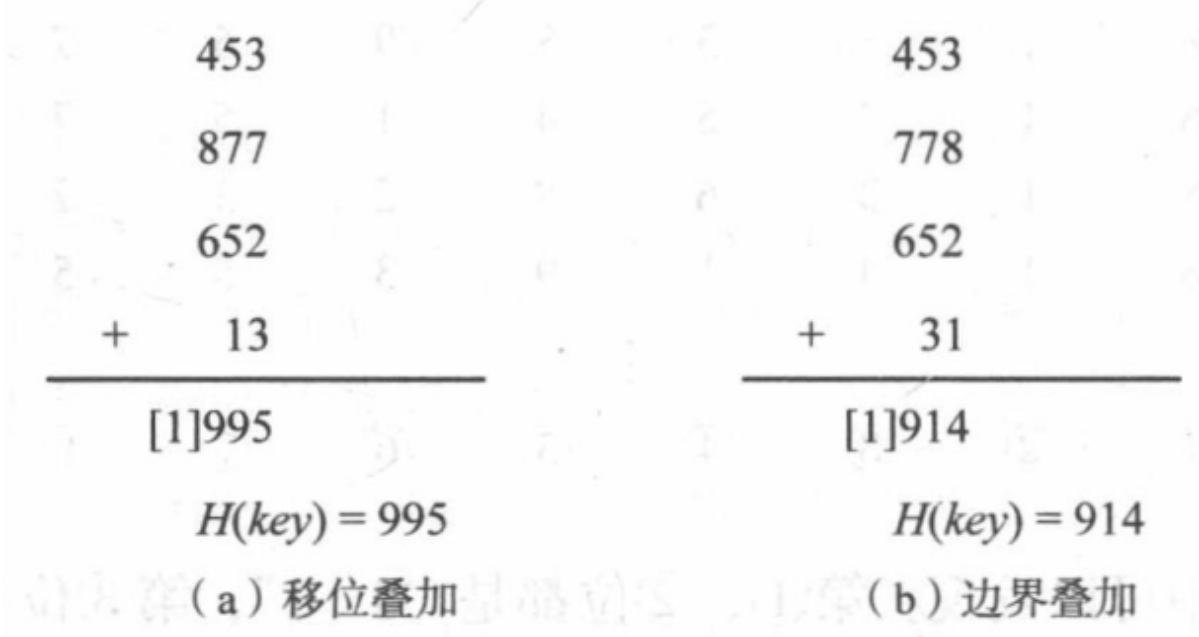
通常在选定散列函数时**不一定能知道关键字的全部情况**，取其中哪几位也不一定合适，而**一个数平方后的中间几位数和数的每一位都相关**，如果取关键字平方后的中间几位或其组合作为散列地址，则使随机分布的关键字得到的散列地址也是随机的，具体所取的位数由表长决定。平方取中法是一种较常用的构造散列函数的方法。

通常在选定散列函数时**不一定能知道关键字的全部情况**，取其中哪几位也不一定合适，而**一个数平方后的中间几位数和数的每一位都相关**，如果取关键字平方后的中间几位或其组合作为散列地址，则使随机分布的关键字得到的散列地址也是随机的，具体所取的位数由表长决定。平方取中法是一种较常用的构造散列函数的方法。

（3）折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址，这种方法称为折叠法。根据数位叠加的方式，可以把折叠法分为移位叠加和边界叠加两种。移位叠加是将分割后每一部分的最低位对齐，然后相加;边界叠加是将两个相邻的部分沿边界来回折看，然后对齐相加。

例如，当散列表长为 1000 时，关键字key=45387765213，从左到右按3 位数一段分割，可以得到 4个部分:453、877、652、13。分别采用移位叠加和边界叠加，求得散列地址为 995 和914，如图 1 所示。



折叠法的适用情况：适合于散列地址的位数较少，而关键字的位数较多，且难于直接从关键字中找到取值较分散的几位。

（4）除留余数法

假设散列表表长为 m（$len(散列) = m$），选择一个不大于的数，用去除关键字，除后所得余数为散列地址，即），选择一个不大于m的数p，用p去除关键字，除后所得余数为散列地址，即 $H(key) = key$

这个方法的关键是选取适当的p，**一般情况下，可以选p为小于表长的最大质数。**例如，表长m=100，可取p=97。

除留余数法计算简单，适用范围非常广，是最常用的构造散列函数的方法。它不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模，这样能够保证散列地址一定落在散列表的地址空间中。

2.处理冲突的方法

处理冲突的方法与散列表本身的组织形式有关。按组织形式的不同，通常分两大类:开放地址法和链地址法。

（1）开放地址法

开放地址法的基本思想是：把记录都存储在散列表数组中，当某一记录关键字 key 的初始散列地址 H0 = H(key)发生冲突时，以 H0 为基础，采取合适方法计算得到另一个地址 H1，如果 H1 仍然发生冲突，以为基础再求下一个地址 H2，若 H2 仍然冲突，再求得 H3。依次类推，直至 Hk 不发生冲突为止，则 Hk 为该记录在表中的散列地址。

- 线性探测法 $d_i = 1, 2, 3, \ldots, m-1$

这种探测方法可以将散列表假想成一个循环表，发生冲突时，从冲突地址的下一单元顺序寻找空单元，如果到最后一个位置也没找到空单元，则回到表头开始继续查找，直到找到一个空位，就把此元素放入此空位中。如果找不到空位，则说明散列表已满，需要进行溢出处理。

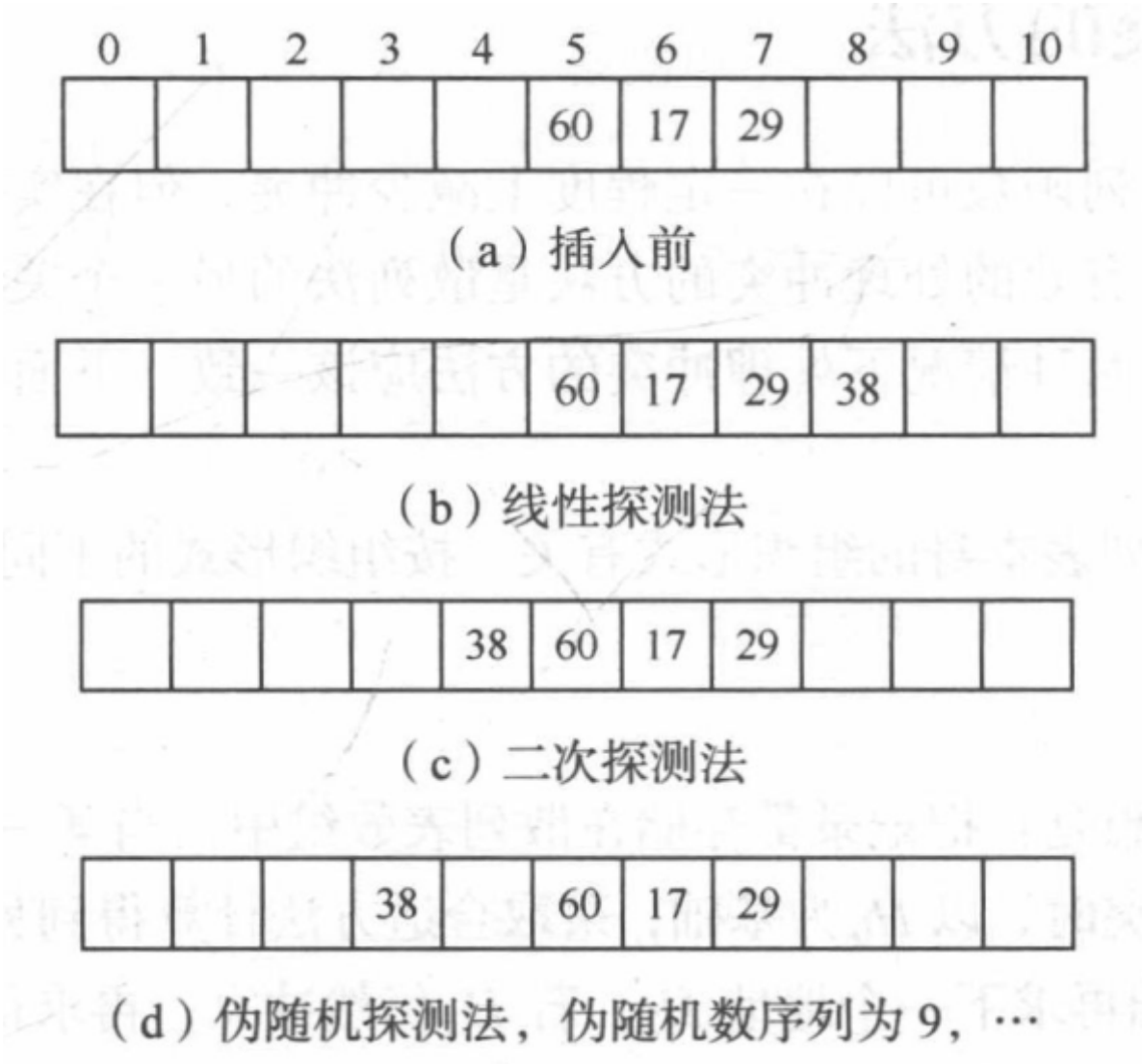
- 二次探测$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, -3^2, \ldots\ldots, +k^2, -k^2 (k \leq m/2)$
- 伪随机探测法

$d_i = 伪随机数序列$ 例如，散列表的长度为 11，散列函数 $H(key) = key$，假设表中已填有关键字分别为 17、60、29 的记录，如图`7.29(a)`所示。现有第四个记录，其关键字为38，由散列函数得到散列地址为 5，产生冲突。

若用线性探测法处理时，得到下一个地址6，仍冲突；再求下一个地址7，仍冲突；直到散列地址为8的位置为“空”时为止，处理冲突的过程结束，38填入散列表中序号为8的位置，如图2(b)所示。

若用二次探测法，散列地址5冲突后，得到下一个地址6，仍冲突；再求得下一个地址 4，无冲突，38填入序号为4的位置，如图 2(c)所示。

若用伪随机探测法，假设产生的伪随机数为9，则计算下一个散列地址为(5+9)%11=3，所以38 填入序号为3 的位置，如图 2(d)所示。



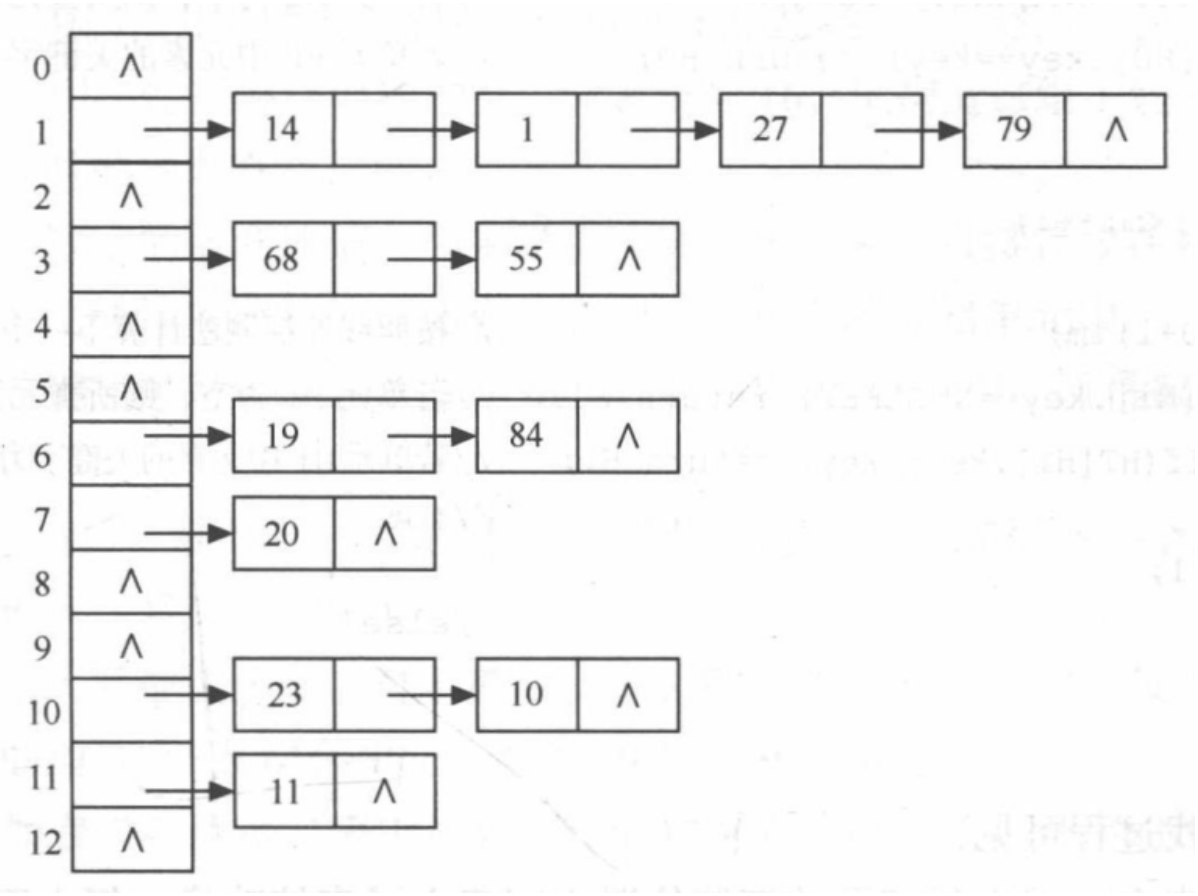
在处理冲突过程中发生的两个第一个散列地址不同的记录争夺同一个后继散列地址的现象称作“**二次聚集**”(或称作“**堆积**”)，即在处理同义词的冲突过程中又添加了非同义词的冲突。

可以看出，上述三种处理方法各有优缺点。线性探测法的优点是：只要散列表未填满，总能找到一个不发生冲突的地址。缺点是：会产生“二次聚集”现象。而二次探测法和伪随机探测法的优点是：可以避免“二次聚集”现象。缺点也很显然：不能保证一定找到不发生冲突的地址。

（2）链地址法

用链地址法处理冲突，试构造这组关键字的散列表。

由散列函数 H(key)=key %13 得知散列地址的值域为 0~12，故整个散列表有 13 个单链表组成，用数组 HT[0..12]存放各个链表的头指针。如散列地址均为1的同义词 14、1、27、79 构成一个单链表，链表的头指针保存在 HT[1]中，同理，可以构造其他几个单链表，整个散列表的结构如图 3 所示。


0
1
2
3
4
5
6
7
8
9
10
11
12
14
1
27
79
68
55
19
84
20
23
10
11