数据算法与结构整理

Part 1 琐碎的算法

本部分记录了一些上个学期和这个学期学到的有趣的小知识。上个学期其实有过这样的念头，只是学习新算法对我来说本就是个不小的负担，期末考试也是用了别人提供的cheatsheet。现在一方面是为了完成课程大作业，另一方面在假期之后对于这些基本算法遗忘现象非常严重。好在重拾的过程比初学要轻松太多太多，重新回顾一些经典题目，还是会一边读代码一边拍手称快。于是我打算把它们写下来放在这里。

1.递归

能用递归解决的问题通常能被分解成相似的子问题。问题本身难以直接入手，然而子问题与母问题之间的关系却是清晰明了的。所以递归的算法实现思路是"自上而下"的，通过在函数中进行自调用而实现。以下是一个例子。

04147:汉诺塔问题(Tower of Hanoi)

http://cs101.openjudge.cn/practice/04147

题文：

有三根杆子A，B，C。A杆上有N个(N>1)穿孔圆盘，盘的尺寸由下到上依次变小。要求按下列规则将所有圆盘移至C杆： 每次只能移动一个圆盘； 大盘不能叠在小盘上面。 提示：可将圆盘临时置于B杆，也可将从A杆移出的圆盘重新移回A杆，但都必须遵循上述两条规则。问：如何移？最少要移动多少次？

思路：

母问题：借助B杆将N个圆盘从A杆移动到C杆
步骤：1）先将上面N-1个盘挪到B（子问题：借助C杆将N-1个圆盘从A杆移动到B杆） 2）将第N个盘挪到C 3）将N-1个盘挪到C（子问题：借助A杆将N-1个圆盘从B杆移动到C杆）

代码

```
def move(disk,begin,end):
    print("{}:{}->{}".format(disk, begin, end))
def operation(disk,begin,mid,end):
    if disk==1:
        move(1,begin,end)
    else:
        operation(disk-1,begin,end,mid)#借助end把begin上面的n-1移动到mid
        move(disk,begin,end)
        operation(disk-1,mid,begin,end)#借助begin把mid上面的n-1个盘移动到end
k,begin,mid,end=input().split()
operation(int(k),begin,mid,end)
```

02694：波兰表达式

http://cs101.openjudge.cn/2024sp_routine/02694/

代码

```
s = input().split()
def cal():
    cur = s.pop(0)
    if cur in "+-*/":
        return str(eval(cal() + cur + cal()))
    else:
        return cur
print("%.6f" % float(cal()))
```

2.二分查找

常常需要用到逆向思维，并不是那么好想，成品的代码倒是一般比较简洁。

具体来说，一般是先设计一个函数，用以判断参数设定是否符合条件，再二分地去寻找最优参数。

08210：河中跳房子

```
l,n,m=map(int,input().split())
stone=[0]
for i in range(n):
    stone.append(int(input()))
stone.append(l)

def check(L,m,stone):
    distance=0
    off=0
    for i in range(1,n+2):
        distance+=stone[i]-stone[i-1]
        if distance<L:
            off+=1
        else:
            distance=0
    if off<=m:
        return off
    return False

mi,ma=0,l
ans=-1
while mi!=ma:
    middle=(mi+ma)//2
    if check(middle,m,stone):
        mi=middle+1
        ans=middle#注意具体案例中，查找过程的设置。此处若check(middle)，则middle可能就是
答案
    else:
        ma=middle
print(ans)
```

04135:月度开销

OpenJudge - 04135:月度开销

题文：

Farmer John is an astounding accounting wizard and has realized he might run out of money to run the farm. He has already calculated and recorded the exact amount of money (1 ≤ *moneyi* ≤ 10,000) that he will need to spend each day over the next *N* (1 ≤ *N* ≤ 100,000) days.

FJ wants to create a budget for a sequential set of exactly *M* (1 ≤ *M* ≤ *N*) fiscal periods called "fajomonths". Each of these fajomonths contains a set of 1 or more consecutive days. Every day is contained in exactly one fajomonth.

FJ's goal is to arrange the fajomonths so as to minimize the expenses of the fajomonth with the highest spending and thus determine his monthly spending limit.

代码

```
n, m = map(int, input().split())
L = list(int(input()) for x in range(n))

def check(x):#看看以x为一个fajo月总开销可不可行
    num, cut = 1, 0
    for i in range(n):
        if cut + L[i] > x:
            num += 1
            cut = L[i]  #在L[i]左边插一个板，L[i]属于新的fajo月
        else:
            cut += L[i]

    if num > m:
        return False
    else:
        return True

maxmax = sum(L)
minmax = max(L)
while minmax < maxmax:
    middle = (maxmax + minmax) // 2
    if check(middle):   #表明这种插法可行，那么看看更小的插法可不可以
        maxmax = middle
    else:
        minmax = middle + 1#这种插法不可行，改变minmax看看下一种插法可不可以

print(maxmax)
```

对于二分查找的递归实现，每次递归调用都会将问题规模减半，因此递归的深度就是问题规模的对数级别。在最坏情况下，递归的深度达到$\log_2 N$。每次递归调用会占用栈空间，而栈的使用情况可以通过递归调用的最大深度来估计。因此，递归调用栈的最小容量应为最大递归深度的值。由此，递归调用栈的最小容量为$\lceil \log_2 N \rceil$（向上取整）

3.并查集

一种可以动态维护若干个不重叠的结合，并支持**合并**与**查询**两种操作的一种数据结构。

01182 食物链

```
def find(x):    # 并查集查询
    if p[x] == x:
        return x
    else:
        p[x] = find(p[x])    # 父节点设为根节点。目的是路径压缩。
        return p[x]

n,k = map(int, input().split())

p = [0]*(3*n + 1)
for i in range(3*n+1):    #并查集初始化，设置父节点为其自身
    p[i] = i

ans = 0
for _ in range(k):
    a,x,y = map(int, input().split())
    if x>n or y>n:#超出范围
        ans += 1; continue

    if a==1:
        if find(x+n)==find(y) or find(y+n)==find(x):#x吃y，或者y吃x
            ans += 1; continue

        # 合并
        p[find(x)] = find(y)
        p[find(x+n)] = find(y+n)#吃的
        p[find(x+2*n)] = find(y+2*n)#被吃
    else:
        if find(x)==find(y) or find(y+n)==find(x):#x和y是同类，或者y吃x
            ans += 1; continue
        p[find(x+n)] = find(y)#x吃y
        p[find(y+2*n)] = find(x)#y被x吃
        p[find(x+2*n)] = find(y+n)#吃x的被y吃

print(ans)
```

Part 2 时间复杂度

Sorting Algorithm

稳定性在原队列中$A_1 = A_2$，且A_1排在A_2前面。若排序后两者位置一定不发生改变则为稳定，否则不稳定。

冒泡排序（Bubble Sort）

一遍又一遍从左到右扫过整个队列，比较相邻两个元素，将大的放在右边。第一次结束，最大的数被移到了最右边。

```
def bubbleSort(arr):
    n=len(arr)
    for i in range(n):
        swapped=False
        for j in range(0,n-i-1):#最大的i个数已经在最右边排好了
            if arr[j]>arr[j+1]:
                arr[j],arr[j+1]=arr[j+1],arr[j]
                swapped=True
        if (swapped==False):
            break
```

选择排序（Selection Sort）

在未排序序列中找到最小的元素，放到排序序列的起始位置。在从剩余未排序的元素中寻找最小元素，放在已排序序列的末尾。

```
for i in range(len(s)):
    min_idx=i
    for j in range(i+1,len(s)):
        if s[min_idx]>s[j]:
            min_idx=j
    s[i],s[min_idx]=s[min_idx],s[i]#找到的最小数放在已排序的后面
```

快速排序（Quick Sort）

选择一个元素作为枢轴（pivot），将比它小的数放在左边，大的放在右边。此时枢轴已经处于正确的位置上，再对左右两个序列重复操作。

```
def quicksort(arr,left,right):#序列从left到right排序
    if left<right:
        partition_pos=partition(arr,left,right)
        quicksort(arr,left,partition_pos-1)
        quicksort(arr,partition_pos+1,right)
def partition(arr,left,right):
    i=left
    j=right-1
    pivot=arr[right]
    while i<=j:
        while i<=right and arr[i]<pivot:
            i+=1
        while j>=left and arr[j]>=pivot:
            j-=1
        if i<j:
            arr[i],arr[j]=arr[j],arr[i]
        if arr[i]>pivot:
            arr[i],arr[right]=arr[right],arr[i]
    return i
```

插入排序（Insertion Sort）

把第一个元素看成有序队列（一个元素自身就是有序的），第二个至最后一个元素看成无序队列。每次将无序队列第一个元素插入有序队列。（一个一个比较，但不是换位置，而是比较完成后一次性插入）

归并排序（Merge Sort）

把序列拆成两部分，对每一部分递归操作，然后拼贴在一起。注意，递归逻辑是，先把左边排好了，再排右边。

```
def mergeSort(arr):
    if len(arr)>1:
        mid=len(arr)//2#分成两半
        L=arr[:mid]
        R=arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i,j,k=0,0,0
        while i<len(L) and j < len(R):#将排好序的L，R合并到一起
            if L[i]<=R[j]:
                arr[k]=L[i]
                i+=1
            else:
                arr[k]=R[j]
                j+=1
            k+=1
        #以下两个循环实际上只会执行一个
        while i<len(L):#剩余元素放队尾，不能遗漏
            arr[k]=L[i]
            i+=1
            k+=1
        while j<len(R):
            arr[k]=R[j]
            j+=1
            k+=1
```

希尔排序（Shell Sort）

按照gap作为间隔排序，再缩短间隔。可以看成是直接选择排序的升级版。

```
def shellSort(arr,n):
    gap=n//2
    while gap>0:
        j=gap
        while j<n:
            i=j-gap
            while i>=0:
                if arr[i+gap]>arr[i]:
                    break
                else:
                    arr[i+gap],arr[i]=arr[i],arr[i+gap]
                i=i-gap
            j+=1
        gap=gap//2
```

各个排序方法的时间复杂度列在下面的表中。

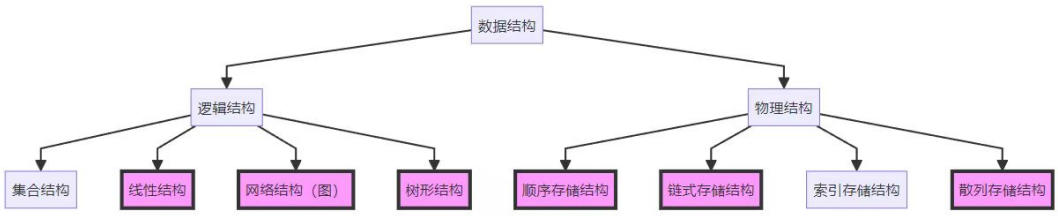
Name	Best	Average	Worst	Memory	Stable
In-place merge sort	—	—	$n \log^2 n$	1	Yes
堆排序 Heap sort	$n \log n$	$n \log n$	$n \log n$	1	No
归并排序 Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes
Tim sort（目前最快算法）	n	$n \log n$	$n \log n$	n	Yes
快速排序 Quick sort	$n \log n$	$n \log n$	n^2	$\log n$	No
希尔排序 Shell sort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No
插入排序 Insertion sort	n	n^2	n^2	1	Yes
冒泡排序 Bubble sort	n	n^2	n^2	1	Yes
选择排序 Selection sort	n^2	n^2	n^2	1	No

接下来将会探讨一系列的数据结构。首先一些基本的概念：

（1）数据元素是数据的基本单位。

（2）数据项是数据不可分割的最小单位。

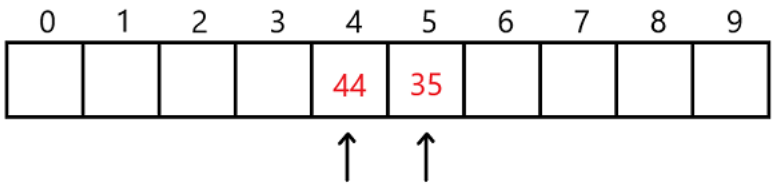
其次明确一下逻辑结构和储存结构的区别。前者是抽象出来的，数据之间的逻辑关系，比如线性结构（线性表，栈，队列，串）、树形结构、图状结构；后者是逻辑结构在计算机中的储存方式，主要包括顺序存储、链式存储、散列存储，此处先补充一下散列表。



散列表

通过设计映射函数，根据给定的值计算出应当储存的位置。这样的函数也叫做**散列函数**或者**哈希函数**，相应的查找表叫做**哈希表**。因为可以通过key值直接计算出储存的位置，所以散列表的查找的时间复杂度基本上与数据规模n无关，为O(1)。

常遇到的问题是不同的key计算出的值有所冲突，需要额外设定规则。比如向后顺延直至该位置未被占用，称为线性探测法，只要表没满就可以找到一个位置。

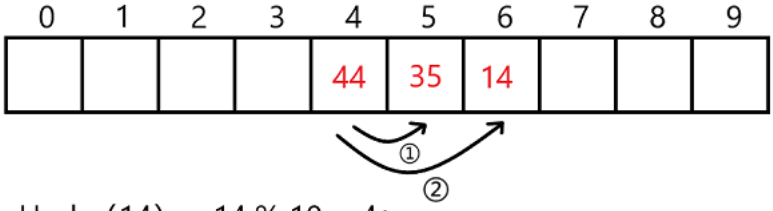


Hash（44）= 44 % 10 = 4；

Hash（35）= 35 % 10 = 5；

元素44：add[4]；

元素35：add[5]；



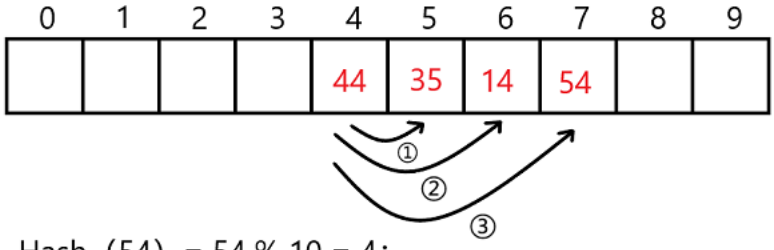
Hash（14）= 14 % 10 = 4；

① $d_1 = 1$；

② $d_2 = 2$；

线性探测

所以，元素14：add[4] -> add[5] -> add[6]；



Hash（54）= 54 % 10 = 4；

① $d_1 = 1$；

② $d_2 = 2$；

③ $d_3 = 3$；

线性探测

所以，元素54：add[4] -> add[5] -> add[6] -> add[7]；

二次探测法，是按照i^2步伐来移动，即$d_i = i^2$，以规避冲突。

或者可以用单链表记录计算出来的地址相同的元素。也叫做开散列法，示意图如下：

设散列函数 H(key) = key %13,
用链地址法处理冲突，构造散列表。

{19，14，23，1，68，20，84，27，55，11，10，79}

H(14) =14%13=1;
H(1) =1%13=1;
H(27) =27%13=1;
H(79)=79%13=1;

H(68)=68%13=3;

H(55)=55%13=3;

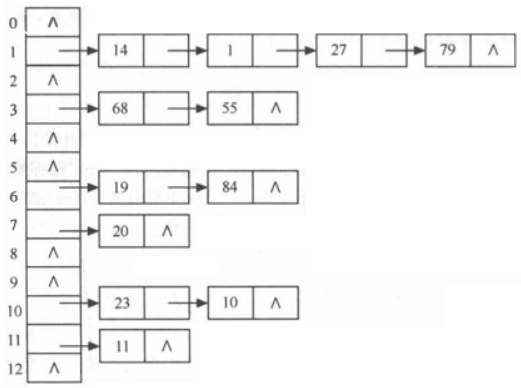
H(11)=11%13=11;

H(19)=19%13=6;
H(84)=84%13=6;

H(20)=20%13=7;

H(23)=23%13=10;

H(23)=23%13=10;



Part 3 栈（stack）和队列（queue）

1.基本概念

线性表描述了元素按线性顺序排列的规则。常见的线性表有**数组**和**链表**。（树不是线性结构）

数组/顺序表的储存是连续的，元素按照顺序存储在内存中连续地址空间上。访问元素时间复杂度为O(1)，插入和删除为O(n)，因为要移动其他元素来保持连续存储的特性。

链表的每个节点包含元素本身以及指向下一个节点的指针，插入和删除为O(1)，因为只需要调整指针。访问为O(n)，因为必须从头开始遍历链表。链表分为：

（1）**单向链表（单链表）**：每个节点只有一个指向下一个节点的指针，最后一个节点指针为空（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。

（3）**循环链表**：最后一个节点的指针指向链表的头部。

注意，在python中，list是使用动态数组（Dynamic Array）实现的。动态数组是一种连续的、固定大小的内存块，可以在需要时自动调整大小。list支持快速随机访问和高效的尾部操作如append和pop

队列和栈是两种重要的数据结构，它们具有push k和pop操作。push k是将数字k加入到队列或栈中，pop则是从队列和栈取一个数出来。队列和栈的区别在于取数的位置是不同的。
队列（queue）是**先进先出**的：把队列看成横向的一个通道，则push k是将k放到队列的最右边，而pop则是从队列的最左边取出一个数。

一种特殊的队列是**循环队列**，其特点在于最后一个元素的位置指向第一个元素，从而在逻辑上形成了环状结构。front指向队头而rear指向队尾。循环队列事先约定好了最大容量m，有以下关系式成立：

（1）若$front = rear$则队列为空；

（2）若$(rear + 1)\%m = front$则队列已满；

（3）队列长度$(rear - front + m)\%m$；

（4）一个新元素入队，$rear = (rear + 1)\%m$；

（5）一个元素出队，$front = (front + 1)\%m$

栈(stack)是**后进先出**的：把栈也看成横向的一个通道，则push k是将k放到栈的最右边栈顶，而pop也是从栈的最右边取出一个数。栈的左边称为栈底。

用类的方式实现栈（stack）

```
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)
s = Stack()#这样就创造了一个空栈
s.push(8)#调用函数的示例
```

类

好像之前并不知道类怎么用，那么现在来讲一讲。字符串、列表、字典其实都是不同的对象，而类就是赋予我们**自定义对象**的能力。一个对象的数据，称为对象的**属性**；一个对象的函数，称为对象的**方法**。

```
class circle(object):
    def __init__(self,r):#作用是表明接受的参数赋予谁，self是对象，但是还没有实例化
        self.r=r
    pi=3.14
    def area(self):
        return self.pi*self.r**2
circle1=circle(2)#circle1就相当于一个半径为1的圆
S=circle1.area()#不要忘记括号
```

python有其特殊的**魔法方法**（magic method），可以大致理解成在类的内部调用python自身的函数。

```
__cmp__(self,other)#self<other返回负整数 self==other返回0 self>other返回正整数
__eq__(self,other)#==
__ne__(self,other)#!=
__lt__(self,other)#<
__gt__(self,other)#>
__le__(self,other)#<=
__ge__(self,other)#>=
```

2.前序、中序、后序表达式

下面这个表格呈现了一些例子。

中序表达式	前序表达式	后序表达式
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *

24591:中序表达式转后序表达式

https://cs101.openjudge.cn/practice/24591/

思路：Shunting Yard算法

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。
   (1)如果是操作数（数字），则将其添加到输出栈。
   (2)如果是左括号，则将其推入运算符栈。
   (3)如果是运算符：(3.1)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。(3.2)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。
   (4)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

代码

```
def infix_to_postfix(expression):
    precedence={'+':1,'-':1,'*':2,'/':2}
    stack=[]#运算符栈
    postfix=[]#输出栈
    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]]:
                    #注意是<=号，这和后序表达式逻辑有关
                    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)
n=int(input())
for _ in range(n):
    expression=input()
    print(infix_to_postfix(expression))
```

3. deque

deque是双端队列。

Part 4 树（Tree）

1.基本概念

节点**Node**：每个节点都有名称，或者储存的“键值”。（子节点，父节点，兄弟节点都可以根据字面意思理解）

根节点**Root**：树中唯一没有入边的节点。

层级**Level**： 从根节点到达一个结点的路径，所包含的边的数量，称为这个节点的层级。（根节点的层级为0，最大层级数称为**高度Height**，最长路径的节点个数为**树的深度**）

度数**Degree**：树的节点的度数是该节点的子节点个数，树的度是树内各节点最大的度。

2.前序遍历，中序遍历，后序遍历，层次遍历

前序遍历 在前序遍历中，先访问根节点，然后递归地前序遍历左子树，最后递归地前序遍历右子树。

中序遍历 在中序遍历中，先递归地中序遍历左子树，然后访问根节点，最后递归地中序遍历右子树。

后序遍历 在后序遍历中，先递归地后序遍历右子树，然后递归地后序遍历左子树，最后访问根节点。

（只用定义最简单二叉树的遍历规则，使用递归就可以了）

24729: 括号嵌套树

http://cs101.openjudge.cn/practice/24729/

注：这个题目集合了树的括号表示方法，以及前序、后序遍历的代码实现。

代码

```
class Treenode:
    def __init__(self,value):
        self.value=value
        self.children=[]#每个节点的子节点
def parse_tree(s):
    stack=[]
    node=None
    for char in s:
        if char.isalpha():
            node=Treenode(char)#创建新节点
            if stack:
                stack[-1].children.append(node)#如果栈不为空，即根节点已经存在，把当前节点作为子节点加入栈顶节点的子节点列表中
        elif char=='(':#左括号意味着当前节点可能有子节点，加入栈中
            if node:
                stack.append(node)
        elif char==')':#当前节点不再有新的子节点
            if stack:
                node=stack.pop()#弹出当前节点
    return node#最后被弹出的一定是根节点，所以返回值是根节点
def preorder(node):#前序遍历函数，node是根节点
    output=[node.value]
    for child in node.children:
```

```
        output.extend(preorder(child))#递归地遍历子树，已经保证从左开始
    return ''.join(output)
def postorder(node):#后序遍历函数
    output=[]
    for child in node.children:
        output.extend(postorder(child))#递归地遍历子树
    output.append(node.value)#前序和后序的不同通过加入根节点的时机表现出来
    return ''.join(output)
def main():
    s=input().strip()
    s=''.join(s.split())
    root=parse_tree(s)
    print(preorder(root))
    print(postorder(root))
if __name__=='__main__':
    main()
```

层次遍历与以上三者有很大的不同。不需要使用递归处理，使用队列即可。代码实现如下

```
def level_order_traversal(root):#层次遍历,traversal就有“遍历”的意思
    queue=[root]
    traversal=[]
    while queue:
        node=queue.pop(0)
        traversal.append(node.value)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return traversal
```

3.二叉树

最笼统的二叉树的定义是简单的，但是有很多特殊定语修饰的概念还是需要了然于心。

3.1 二叉搜索树（二叉排序树）

左子树中的每个节点的值都小于根节点；右子树中每个节点的值都大于根节点。

```
#由任意无重复数字的数列产生二叉搜索树的代码实现
class TreeNode:
    def __init__(self,value):
        self.value=value
        self.left=None
        self.right=None
def insert(node,value):#向二叉树中加入新的节点
    if node is None:
        return TreeNode(value)
    if value < node.value:
        node.left=insert(node.left,value)#在每个节点处判断应该在左子树还是右子树，若已经存在子节点则向下递归
    elif value>node.value:
        node.right=insert(node.right,value)
    return node
```

```
numbers=list(map(int,input().strip().split()))
numbers=list(dict.fromkeys(numbers))#去除重复数字
root=None
for num in numbers:
    root=insert(root,num)#每一个数字都从根节点开始找位置，一旦插入就不再更改位置
```

3.2 二叉堆

完全二叉树：每一层从左到右依此填充，全部填充满后才开始填充下一层。

我们可以利用完全二叉树来实现**堆**，或者说是**二叉堆**，每一层都是从左到右填充。有趣的是，完全二叉树可以用列表来表示，因为对于每一个节点其位置为p（注意这里的位置不是从0开始，而是从1开始），左子节点$2p$，右子节点$2p+1$，其次堆具有**有序性**，即子节点x必定不小于父节点p。

3.2.1 小根堆/大根堆

父节点的值小于/大于其子节点的值。

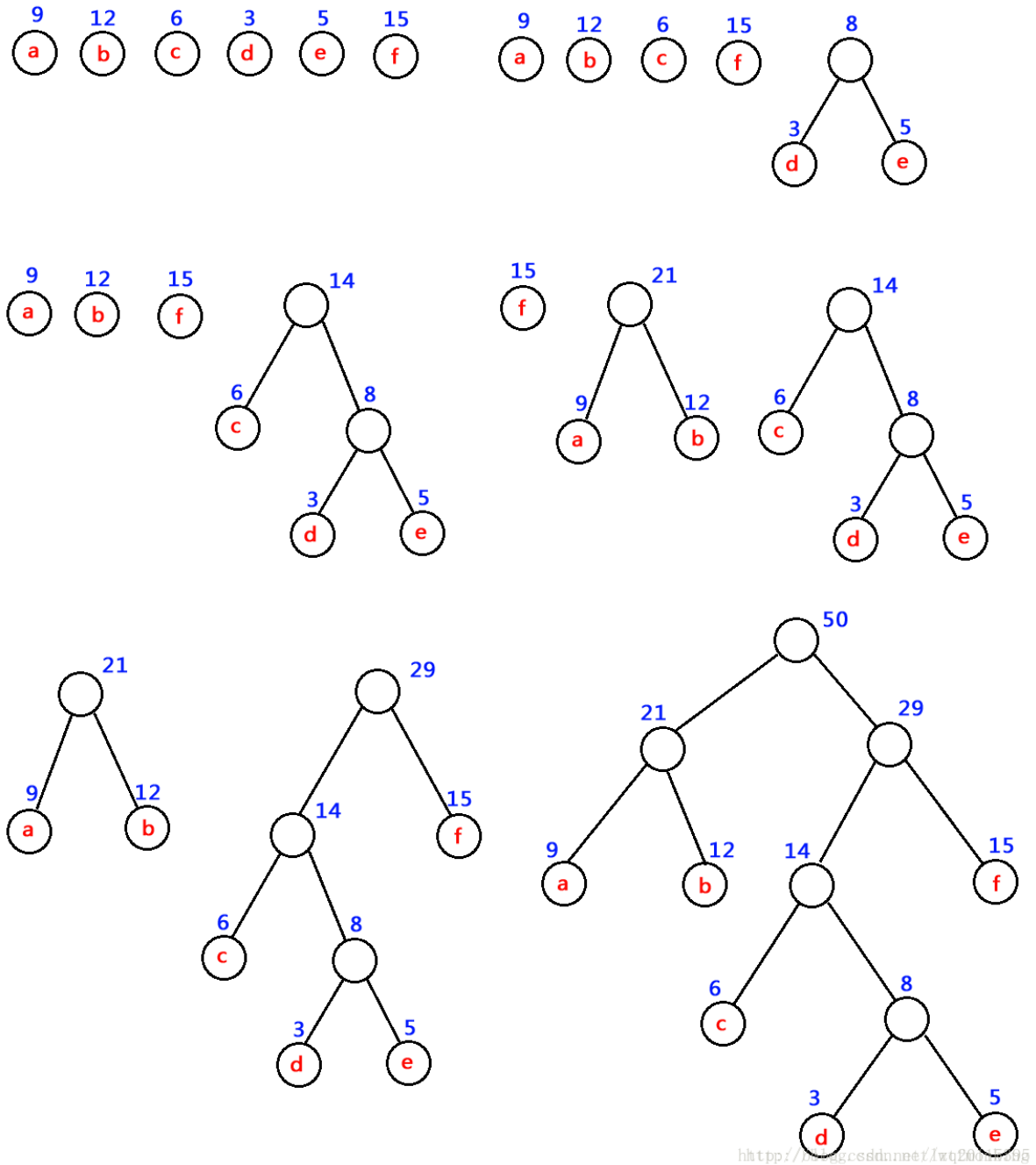
以下是手搓最小堆的实现代码。

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

    def percUp(self, i):#前i-1保证最小堆，排序使i个元素构成最小堆
        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):#加入一个数据并保持最小堆
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def percDown(self, i):#i位置的数和子节点比较，较小的上移
        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):#找最小子节点
        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):#删除最小元素并保持最小堆
        retval = self.heapList[1]
```

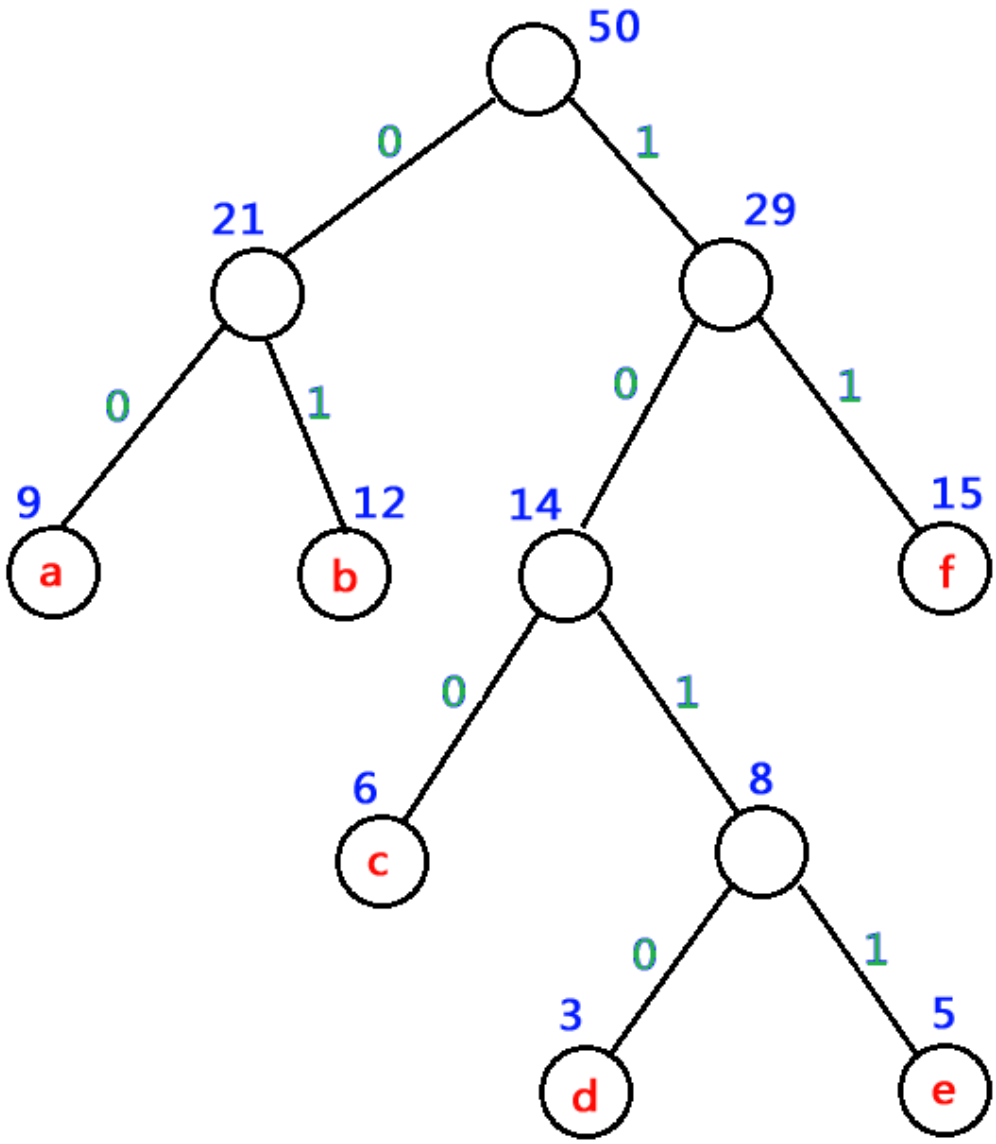
```
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval

    def buildHeap(self, alist):#建立最小堆
        i = len(alist) // 2#前一半正好是非叶节点，后一半都是叶节点
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):#将前一半逐个下沉
            self.percDown(i)
            i = i - 1
```

3.2.2 哈夫曼编码

哈夫曼编码的特点，是依据频率对字符的不定长编码。具体操作过程为，每次从森林中选取根节点权值最小的两棵树，将其合并为一棵树加回到森林中。这棵树的根节点权值为两棵树根节点权值之和，左右节点分别为两棵树的根节点，并且左节点小于右节点。下图是对六个点的编码过程的示例：



编码过程为，从根节点开始，为其路径上左分支赋值0，右分支赋值1，示例如下：



例如，a的编码为00，而d的编码为1010。

完整的代码清单如下

```
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):#定义了heap的排序方法，不然会runtime error
        if self.weight==other.weight:
            return self.char < other.char
        return self.weight<other.weight
def bulid_huffman_tree(characters):
    heap=[]
    for char,weight in characters.items():#items返回可遍历的键值元组
        heapq.heappush(heap,Node(weight,char))
    while len(heap)>1:
        left=heapq.heappop(heap)
        right=heapq.heappop(heap)
        merged=Node(left.weight+right.weight)#合并后，char为None
```

```
            merged.left=left
            merged.right=right
            heapq.heappush(heap,merged)
    return heap[0]
def encode_huffman_tree(root):#确定编码值
    codes={}#字典，存储每个字符对应的编码
    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 huffman_encoding(codes,string):#对特定字符串编码
    encoded=''
    for char in string:
        encoded+=codes[char]
    return encoded
def huffman_decoding(root,encoded_string):#对01串解码
    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
```

3.3 平衡二叉搜索树

最常见的是AVL树，特点在于记录每个节点的平衡因子，**平衡因子**定义为左右子树的高度之差。平衡因子大于零，称为**左倾**，平衡因子小于零称为**右倾**。平衡因子为-1，0，1的树都定义为**平衡树**。

最不平衡的情况下，当书的高度为h时，其节点数满足递推关系$N_h = 1 + N_{h-1} + N_{h-2}$

以下是实现AVL树的代码。

```
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))
        x.height = 1 + max(self._get_height(x.left), self._get_height(x.right))
        return x

    def preorder(self):
```

```
        return self._preorder(self.root)

    def _preorder(self, node):
        if not node:
            return []
        return [node.value] + self._preorder(node.left) +
self._preorder(node.right)

n = int(input().strip())
sequence = list(map(int, input().strip().split()))

avl = AVL()
for value in sequence:
    avl.insert(value)

print(' '.join(map(str, avl.preorder())))
```

要实现从AVL树中删除节点，需要添加一个删除方法，并确保在删除节点后重新平衡树。

下面是更新后的代码，包括删除方法 `_delete`：

```
class AVL:
    # Existing code...

    def delete(self, value):
        self.root = self._delete(value, self.root)

    def _delete(self, value, node):
        if not node:
            return node

        if value < node.value:
            node.left = self._delete(value, node.left)
        elif value > node.value:
            node.right = self._delete(value, node.right)
        else:
            if not node.left:
                temp = node.right
                node = None
                return temp
            elif not node.right:
                temp = node.left
                node = None
                return temp

            temp = self._min_value_node(node.right)
            node.value = temp.value
            node.right = self._delete(temp.value, node.right)

        if not node:
            return node

        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        balance = self._get_balance(node)
```

```
        # Rebalance the tree
        if balance > 1:
            if self._get_balance(node.left) >= 0:
                return self._rotate_right(node)
            else:
                node.left = self._rotate_left(node.left)
                return self._rotate_right(node)

        if balance < -1:
            if self._get_balance(node.right) <= 0:
                return self._rotate_left(node)
            else:
                node.right = self._rotate_right(node.right)
                return self._rotate_left(node)

        return node

    def _min_value_node(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    # Existing code...
```

这段代码中的 `_delete` 方法用于删除节点。它首先检查树中是否存在要删除的节点，然后根据节点的左右子树情况执行相应的操作，以保持AVL树的平衡。

3.4 树、森林、二叉树的相互转化

将一棵普通**树**转化成二叉树的过程可以记住这个简洁的口诀：**左儿子，右兄弟。**下面是具体的步骤：

（1）将树的根节点直接作为二叉树的根节点。

（2）将树的根节点的**第一个子节点**作为根节点的左儿子。如果该子节点存在兄弟节点，将该子节点的第一个兄弟节点（方向从左往右）作为该子节点的右兄弟。

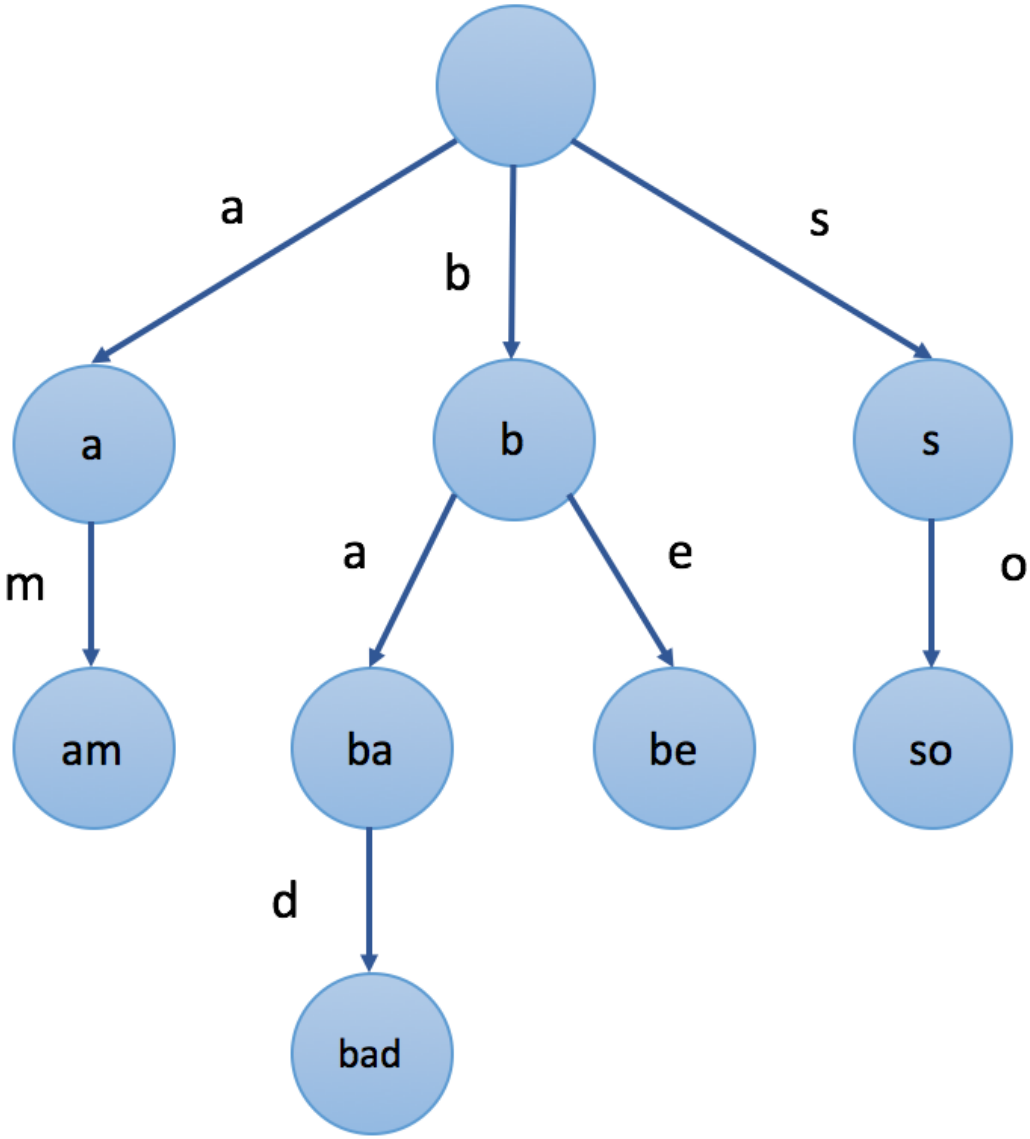
（3）依序将树中的剩余节点按照上一步的方式添加到二叉树中，直到树中所有的节点都在二叉树中。

将**森林**转化为二叉树，将森林中的所有树视为兄弟，第一棵二叉树不变，第二棵二叉树的根节点添加到第一棵树根节点的右子节点。

如何判断一棵二叉树是森林还是树？若根节点右子节点非空则为森林，反之为树。

4.字典树/前缀树

顾名思义，“前缀树”可以用来处理一大堆和单词有关的问题。比如下图就是一个实例，建立前缀树以储存am,bad,be,so四个单词，发现一个节点所有的后代的单词全部拥有相同的前缀。



具体到代码实现上，主要是使用嵌套的字典，字典的值表示该节点的字符串。

```
class Trie:
    def __init__(self):
        self.root={}
        self.end=-1
    def add(self,word):
        node=self.root
        for letter in word:
            if letter not in node:
                node[letter]={}#如果不在键中，意味着产生了新的分支，建立新的字典
            node=node[letter]#沿着子树接着走
        node[self.end]=True#代表此处为一个单词的结束位置
    def search(self,word):#查找word是否在字典树中
        node=self.root
        for letter in word:
            if letter not in node:
                return False
            node=node[letter]
        if self.end not in node:#这个地方不是单词的结尾
            return False
        return True
```

Part 5 图（Graph）

图由**顶点**（Vertex）和**边**（Edge）组成，记号G(V,E)表示图G的顶点集为V、边集为E。顶点可以有自己的值，边既可以是单向的，也可以是双向的。定点和边都可以有一定的属性，量化的属性称为**权值（Weight）**。由此，可以用三元组（u,v,w）表示边，其中u,v是顶点，而w代表权值。

度（Degree）顶点的度指和顶点相连的边的条数。对于有向图还可以区分出度和入度。

连通图：无向图中，任意两个顶点都是连通的。

强连通图：在有向图中，对任意顶点V_i, V_j，从V_i到V_j和从V_j到V_i都连通。

1.如何实现图?

1.1 邻接矩阵

想法很简单，没两个点之间的关系用二维矩阵的一个元素表示。具体而言，a_{vw}是从顶点v到顶点w的有向边的权重。我们立刻可以知道，邻接矩阵的对角元都是0；且对于无向图，其对应的邻接矩阵应当是对称矩阵。

1.2 邻接表

利用列表和字典来实现记录图。建立一个包含所有顶点的列表，对每个顶点建立一个字典，其中的键是到达的顶点，值是对应的边的权重。很显然，这里是针对有向图而言的。

2.最小生成树

2.1 生成树

- 一个连通图可以有多个生成树；
- 一个连通图的所有生成树都包含相同的顶点个数和边数；
- 生成树当中不存在环；
- **移除生成树中的任意一条边都会导致图的不连通**， 生成树的边最少特性；
- 在生成树中**添加一条边会构成环。**
- 对于包含n个顶点的连通图，生成树包含n个顶点和n-1条边；
- 对于包含n个顶点的无向完全图最多包含 n^{n-2} 颗生成树。

2.2 最小生成树

对于带权图而言，最小生成树是边的权值之和最小的生成树。实现最小生成树的算法主要有两种。

2.2.1 Kruskal 算法

1.将图中的所有边按照权重从小到大进行排序。

2.初始化一个空的边集，用于存储最小生成树的边。

3.重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：
  3.1选择排序后的边集中权重最小的边。
  3.2如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。

4.返回最小生成树的边集作为结果。

一般来说，在实现代码的过程中会用到并查集

2.2.2 Prim 算法

基本想法：贪心。构建两个集合，一个是目前可到达的点集A，一个是目前不可到达的点集B。每次从B中挑选最短的可以连接到A的路径添加。

兔子与星空

```
import heapq

def prim(graph, start):
    mst = []
    used = set([start])  # 已经使用过的点
    edges = [
        (cost, start, to)
        for to, cost in graph[start].items()
    ]  # (cost, frm, to) 的列表
    heapq.heapify(edges)  # 转换成最小堆

    while edges:  # 当还有边可以选择时
        cost, frm, to = heapq.heappop(edges)    # 弹出最小边
        if to not in used:  # 如果这个点还没被使用过
            used.add(to)  # 标记为已使用
            mst.append((frm, to, cost))  # 加入到最小生成树中
            for to_next, cost2 in graph[to].items():  # 将与这个点相连的边加入到堆中
                if to_next not in used:  # 如果这个点还没被使用过
                    heapq.heappush(edges, (cost2, to, to_next))  # 加入到堆中

    return mst  # 返回最小生成树

n = int(input())
graph = {chr(i+65): {} for i in range(n)}
for i in range(n-1):
    data = input().split()
    node = data[0]
    for j in range(2, len(data), 2):
        graph[node][data[j]] = int(data[j+1])
        graph[data[j]][node] = int(data[j+1])

mst = prim(graph, 'A')  # 从A开始生成最小生成树
print(sum([cost for frm, to, cost in mst]))  # 输出最小生成树的总权值
```

27880：繁忙的厦门

```
from heapq import*
class Cross:
    def __init__(self,name):
        self.name=name
        self.roads={}


def build_road(cross,start):
    ans=[]
    visited=[start]
    edges=[[cross[start].roads[new],new] for new in cross[start].roads]
    heapify(edges)
    while edges:
        newroad = heappop(edges)
```

```
        if newroad[1] not in visited:
            ans.append(newroad[0])
            visited.append(newroad[1])
            for new in cross[newroad[1]].roads:
                if new not in visited:
                    heappush(edges,
[max(newroad[0],cross[newroad[1]].roads[new]),new])
    return ans

n, m = map(int,input().split())
cross=[Cross(i) for i in range(n)]
for i in range(m):
    a,b,c=map(int,input().split())
    cross[a-1].roads[b-1]=c
    cross[b-1].roads[a-1]=c
ans=build_road(cross,0)
print(len(ans),max(ans))
```

2.3 Dijkstra算法

这个算法用于解决单源最短路径问题。对于给定的源节点，找到它到其余所有节点的最短路径。缺点在于，不能处理包含负值路径的体系。具体操作流程如下：

（1）初始化一个距离数组，用于记录源节点到其他所有节点的**最短**距离。初始时，源节点到自身距离为0，到其余点为无穷大；

（2）选择一个未访问的节点中，距离最小的节点作为**当前节点**；

第一次这个节点就是源节点，所有直接与它相邻的顶点的距离都会由无穷大更新为与源节点直接相邻路径的长度

（3）对于当前节点的所有邻居节点，若通过当前节点到达邻居节点的路径比已知最短路径更短，则更新最短路径；

（4）标记当前节点为已访问，重复以上步骤直到所有节点都被访问或者所有节点的最短路径都被确定。

```
import heapq

def dijkstra(N, G, start):
    INF = float('inf')
    dist = [INF] * (N + 1)  # 存储源点到各个节点的最短距离
    dist[start] = 0  # 源点到自身的距离为0
    pq = [(0, start)]  # 使用优先队列，存储节点的最短距离
    while pq:
        d, node = heapq.heappop(pq)  # 弹出当前最短距离的节点
        if d > dist[node]:  # 如果该节点已经被更新过了，则跳过
            continue
        for neighbor, weight in G[node]:  # 遍历当前节点的所有邻居节点
            new_dist = dist[node] + weight  # 计算经当前节点到达邻居节点的距离
            if new_dist < dist[neighbor]:  # 如果新距离小于已知最短距离，则更新最短距离
                dist[neighbor] = new_dist
                heapq.heappush(pq, (new_dist, neighbor))  # 将邻居节点加入优先队列
    return dist



N, M = map(int, input().split())
G = [[] for _ in range(N + 1)]  # 图的邻接表表示
```

```
for _ in range(M):
    s, e, w = map(int, input().split())
    G[s].append((e, w))


start_node = 1  # 源点
shortest_distances = dijkstra(N, G, start_node)  # 计算源点到各个节点的最短距离
print(shortest_distances[-1])  # 输出结果
```

3. 图算法

3.1 宽度优先搜索（BFS）

时间复杂度为O(V+E)，空间复杂度为O(V)，其中V是顶点数，E是边数。

特点在于遍历完同一深度的节点才会往下搜索，常用于寻找最短路径、连通域。关于最短路径的讨论上个学期已经有了很多，这里补充一个新的问题——词梯。

28046: 词梯

http://cs101.openjudge.cn/practice/28046/

这个问题大致意思是说，如何从一个单词变到另一个单词，每次只能改变一个字母，并且改变一个字母后必须还是一个单词。我们可以建立很多的词筒，使得每次变换前后的单词处于同一个筒中。

例如对于sale，假设改变其第三个字母，则改变前后只可以在词筒sa*e中选择。实现代码如下。

```
from collections import deque#双端队列

def construct_graph(words):#对给定的单词建立所有可能的词筒，并记录在字典graph中返回
    graph = {}
    for word in words:
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            if pattern not in graph:
                graph[pattern] = []
            graph[pattern].append(word)
    return graph


def bfs(start, end, graph):
    queue = deque([(start, [start])])
    visited = set([start])

    while queue:
        word, path = queue.popleft()
        if word == end:
            return path
        for i in range(len(word)):
            pattern = word[:i] + '*' + word[i + 1:]
            if pattern in graph:
                neighbors = graph[pattern]
                for neighbor in neighbors:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, path + [neighbor]))
```

```
    return None


def word_ladder(words, start, end):
    graph = construct_graph(words)
    return bfs(start, end, graph)


n = int(input())
words = [input().strip() for _ in range(n)]
start, end = input().strip().split()

result = word_ladder(words, start, end)

if result:
    print(' '.join(result))
else:
    print("NO")
```

3.2 深度优先搜索（DFS）

时间复杂度为O(V+E)，空间复杂度为O(V)，其中V是顶点数，E是边数。

28050:骑士周游

OpenJudge - 28050:骑士周游

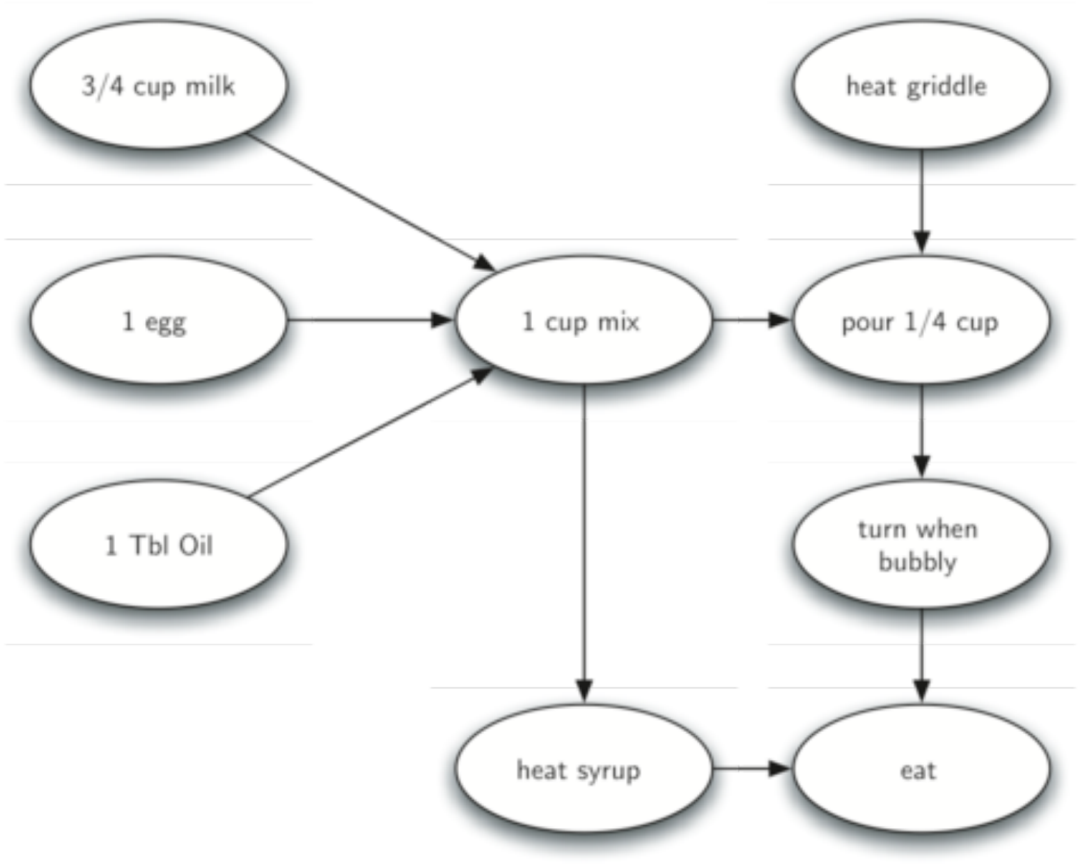
感觉上是一个非典型的DFS，关于周游问题可以讨论的还有很多。下面的代码中，遍历优先去往可到达点较少的点，这样可以提高算法的性能，称为启发式技术。

```
def knight_tour(n, sr, sc):
    moves = [(-2, -1), (-2, 1), (-1, -2), (-1, 2),
             (1, -2), (1, 2), (2, -1), (2, 1)]

    visited = [[False] * n for _ in range(n)]

    def is_valid_move(row, col):  #判断移动是否合法，且是否没有到达过
        return 0 <= row < n and 0 <= col < n and not visited[row][col]

    def count_neighbors(row, col):  #判断有多少合法移动
        count = 0
        for dr, dc in moves:
            next_row, next_col = row + dr, col + dc
            if is_valid_move(next_row, next_col):
                count += 1
        return count

    def sort_moves(row, col):
        neighbor_counts = []
        for dr, dc in moves:
            next_row, next_col = row + dr, col + dc
            if is_valid_move(next_row, next_col):
                count = count_neighbors(next_row, next_col)
                neighbor_counts.append((count, (next_row, next_col)))
        neighbor_counts.sort() #优先访问可到达地点较少的点
        sorted_moves = [move[1] for move in neighbor_counts]
        return sorted_moves
```

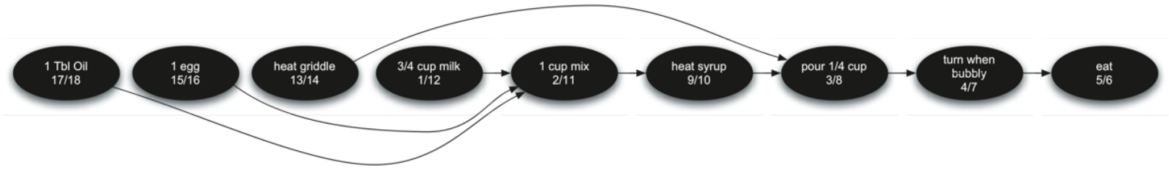
```
    visited[sr][sc] = True
    tour = [(sr, sc)]

    while len(tour) < n * n:
        current_row, current_col = tour[-1]
        sorted_next_moves = sort_moves(current_row, current_col)
        if not sorted_next_moves: #无路可走
            return "fail"
        next_row, next_col = sorted_next_moves[0]
        visited[next_row][next_col] = True
        tour.append((next_row, next_col))

    return "success"

n = int(input())
sr, sc = map(int, input().split())
print(knight_tour(n, sr, sc))
```

3.3 拓扑排序

直观来看一个例子，拓扑排序就是把下图（制作松饼流程图）：



变成如下所示的样子：



具体来说，拓扑排序是给出了一个图中所有的顶点的一种线性排列，满足：

（1）每个顶点只出现一次；

（2）若存在从A到B的有向边，则A在B的前面。

拓扑排序有很多应有，例如可以用来呈现事物优先级，也可以用来判断图中有无环。

3.3.1 判断无向图中是否有环

（1）将所有度<=1的节点加入队列；

（2）队列不为空时，弹出节点，并移去其所有的边（相邻节点度-1），如出现度<=1的节点则加入队列；

（3）若所有节点都进入过队列，则无环。

```
def find(nodes):#nodes为储存顶点的队列
    queue=deque()
    result=[]
    for node in nodes:
        if node.indegree <= 1:
            queue.append(node)
    while queue:
        u = queue.popleft()#需要调用collections里的deque
        result.append(u)
        for v in u.reach:
            v.degree -= 1
            if v.degree <= 1:
                queue.append(v)
    if len(result) == n:
        return result
    return False
```

3.3.2 判断有向图中是否有环（Kahn算法）

此算法的时间复杂度为O(V+E)，具体思路是：

（1）将所有入度为0的节点加入队列；

（2）队列不为空时，弹出节点，并移去所有从其出发的边（该节点指向的所有点入度-1），如出现入度为0的节点则加入队列；

（3）若所有节点都进入过队列，则无环。

```
def find(nodes):#nodes为储存顶点的队列
    queue=deque()
    result=[]
    for node in nodes:
        if node.indegree == 0:
            queue.append(node)
    while queue:
        u = queue.popleft()#需要调用collections里的deque
        result.append(u)
        for v in u.reach:
            v.indegree -= 1
            if v.indegree == 0:
                queue.append(v)
```

```
    if len(result) == n:
        return result#返还拓扑序列
    return False#有环
```