# 0总目标

2023年1月30日 22:01

C&C++：读研机试（、校内考试）（标*表示读研机试也极少会考）

Python：读研机试（标*表示读研机试机少会考，**表示都可不看）

Java：读研机试、蓝桥杯（标*表示仅蓝桥杯要学，读研机试机少会考）

# ①方法细心

2022年1月31日　　14:33

***先把I/O、库、二叉树定义+两种构建方式+中序遍历+层序遍历+BST的插入&查找的板子都码了！**

①细心

*include时勿忘using namespace std!

*空用例或长度为1的用例格外小心！考虑全面（拒绝想当然！）

**为防止四舍五入导致精度损失，小数后面+eps=1e-9!

*（Python和Java必看）ACM模式下涉及到类定义程序的实现：除了定义时在class定义的类中定义外，其他（函数和主函数）都在class外定义，调用时顺着需求调用即可

*树递归函数中需保证若返回树，左右子树都必须有定义（哪怕为空！）

②方法

（Ⅰ）语言选择（仅限于读研机试且有多种选择时）

1.C[++]/Java:全用C[++]

2.C[++]/Java/Python:

AVL：C[++]

其他：Python→（只可能是因为超时）C[++]

（Ⅱ）技巧

0.（若语言为C++:）万能头文件：`#include <bits/stdc++.h>`

1.由题目广阔联想寻找思路，尽量向高效算法靠近（遍历可以试着从右往左遍历，可能更简单！）

2.直接想到的思路不行的话试着换个角度/对象/操作试试（尤其是滑动窗口!）（看上去好像还需要回溯、非单调时仔细判断！）

3.DSA“对号入座”（典型的几个：BFS/DFS,dp，二分查找（慎用！），二叉搜索树，若这些也优化不了考虑线性表变为哈希表）

（很多都能转换成哈希表，多尝试转换成查询的模型，O(n)变成o(1)的利器！）

4.针对目的和支持操作采取不同的数据结构（以下都支持插入、删除的任意操作组合）（再根据具体可用语言情况选择~）：

查找[、求最值]:AVL

求最值:二叉堆

*求和[、查找]（此时不可能再要求求最值）:线段树、树状数组

5.实在不行再试试栈、队列、链表、树、堆这几种数据结构！

*6.根据题目结合树形、二进制编码等思想自己构造结构！

★ PS：

*打表、找规律

*数组不会可以试着先排序一下/强迫自己一遍遍历看看应该怎么实现（是否需要先排序）！

*判断算法是否可行时尤其注意dp和哈希表的判断，不要轻易放弃！

*滑动窗口可以两端各设立一个指针！

*图论的edge数组尽量就直接开成n*n的

*特殊情况（树退化为链表等）可能会造成TLE的：随机化↓

```
for(i=1;i<len;i++)
{
        j=1+rand()%len; //rand()函数生成1~MAX（计算机定义）的伪随机数
        swap(arr[i],arr[j]); //通过交换来随机化
}
```

# *XDU

2022年10月30日　　20:17

***勿忘复习C++&图、哈夫曼编码！**

***先把I/O、库、二叉树定义+两种构建方式+中序遍历+层序遍历+BST的插入&查找的板子都码了！**

***记一下XDOJ的网址！**

①细心

include时勿忘using namespace std!

**XDOJ中gets用gets_s代替！**

空用例或长度为1的用例格外小心！考虑全面（拒绝想当然！）

②方法

*树：递归/BFS~

*直接想到的思路不行的话试着换个角度/对象/操作试试

*尝试各种数据结构！（树转换成堆勿忘！）

*数组不会可以试着先排序一下/强迫自己一遍遍历看看应该怎么实现（是否需要先排序）！

*BST:对于XDOJ这种算法要求不高的OJ，若没有特别的涉及树的要求（遍历等等），可以直接用数组+sort函数实现插入+排序功能

# 0勿忘看机试部分！！

2023年2月23日　　14:36

# ①细心

2023年2月23日　　10:00

*写程序语句勿忘最后的分号！
*计算数组地址：勿忘乘上每个元素占的字节数！
*图节点和边性质的问题：关注是否含有**环路**！
*画线性或链式的哈希表时下标为从0到len-1而非从0到m！

# ②方法

2023年2月23日　　10:33

*辨析（特点、优缺点）（如存储结构、数据结构、逻辑结构等）：
　　总体上从时空角度，分别再从初始化等的所有操作分别讨论~
*算法设计题：
　　*插入记得判断是否已满，弹出判断是否为空
　　*链表设计：
　　　　*按改动的节点为核心对象写，每个节点先写后半节再写前半节
　　　　*关系较乱（如遍历+操作型）可尝试双指针法（t1=head,t2=head->next)
　　*若递归不适合则考虑队列（即用循环实现）

# ③小结论

2023年2月24日　　22:30

*二叉树图论
设节点总数为n，ni为度数为i的节点的数量，i∈{0,1,2}，则：
由此推得小结论：
**n0=floor((n-1)/2)+1**
**n1=(n+1)%2**
**n2=floor((n-1)/2)**

# ④知识

2023年2月23日　　10:34

# （I）数据结构共性

2023年2月22日　　18:16

**0数据结构存在的目的

*针对终极目的（静态：查找、求和、*求最值）和题目要求支持的中间操作过程（动态：**插入**、删除[、修改]）的组合，分别有对应的最高效的容器类型，即数据结构。

*按时空复杂度等效分为五大类：查找/修改、求和、*求最值（[局部]排序）；插入、删除（实际上=查找+插入！）

*本质上、广义上也属于算法，狭义上仅属于支持算法等操作的物质性容器！

*查找具体指查询是否存在

①架构&关系图



*数组不是线性结构，一维数组才是！

*存储结构是逻辑结构的存储影像

*逻辑结构独立于存储结构（反之不成立！），且二者不互相唯一决定！

②常见操作大全

创建（根据输入（一般都是逐一插入法）、全空、*自定义）

静态：判空、获取长度、根据下标/迭代器获取值、查找值对应的下标/迭代器、计数某个值出现次数

动态：局部改变（增一个/*多个（即合并）、改、删一个/*所有）、整体改变（[逆]排序、遍历）

③知识

*数据结构是指相互之间存在一种或多种特定关系的数据元素的集合

*数据结构的三要素：数据的逻辑结构、数据的存储结构、数据的运算

*数据结构在计算机中的表示（又称影像）称为数据的物理结构

*存储密度：数据元素的值所需的存储量/该数据元素所需的存储总量

顺序存储结构存储密度为100%（优点：存储密度大）

链表存储密度一般<100%

# 1.顺序&链式存储

2023年2月22日　　18:12

①基本概念
*数据结构名词解释：
线性表、栈、队列：
1.
线性表：由 n 个数据元素构成的有限序列，除去第一个结点和最后一个结点外，其余结点都各有一个前驱、一个后继；第一个结点只有一个后继；最后一个结点只有一个前驱。（4 分）
栈：限定在表尾做插入、删除的线性表；也称为后进先出的线性表。（4 分）
队列：只允许在一端做插入、另一端做删除的线性表；也称为先进先出的线性表。（4 分）
注：
*算法设计题勿忘head->next=**NULL**;!!!
*链表头指针指向的头结点（若链表非空）为首个数据节点的前一个节点！（即同循环队列）
*循环队列：
1.概念
*头指针(front)：指向**队首元素的前一个位置（所以队首元素不是front！！）**
*队尾指针(rear)：指向队尾元素
2.计算公式
*队列满的条件是：
(rear+1)%QueueSize == front
*队列为空的条件是：
rear=front
*通用的计算队列长度的公式为：
(rear - front+ **QueueSize**)%QueueSize
*队首元素的位置：
(rear-len+1+QueueSize)%QueueSize
*队尾元素的位置：
(front+len+QueueSize)%QueueSize
*链表：
*不可以随机访问任一元素
*所需长度与线性长度成正比
*稀疏矩阵的三元组顺序表：
(row,col,val)，下标从1开头
*对称矩阵：默认存**下**三角
*三对角矩阵：

$$A=\begin{pmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix}$$

*广义表：
*各种层级括号及其中元素的组合

*是一种非线性的数据结构，它的表元素可以是原子或者广义表的一种线性表的扩展结构。

- 广义表的长度：为表中最上层元素的个数
- 广义表的深度：为表中括号的最大层数

- 表头和表尾：当广义表非空时，第一个元素为广义表的表头，其余元素组成的表是广义表的表尾

*操作：

GetHead:获取头部元素（去除最外层括号）

GetTail:去除头部元素的其余，去除最外层括号后，剩余元素整体外面加一个括号（即使原来整体外面已经有括号了！）（即除掉第一个元素剩下的元素组成的广义表）

*前缀编码：任何两个编码中间不能出现其中一个是另一个的前缀

*串：

*在T中寻找等于P的子串的过程称为模式匹配/子串定位

*串也可以链式存储

②实现方法

*串的模式匹配：

（I）库函数

下标一般从0开始

```
substr/SubString(Sub,S,pos,len) //用Sub返回串S的第下标为pos的字符起长度为len的子串（初始条件：串S存在，1<=pos<=SrrLength(S)且0<=StrLength(S)-pos+1）
Index(S,T,pos) //若主串S中存在和串T值相同的子串，则返回它在主串S下标为pos的字符之后第一次出现的位置，否则函数值为0
```

（II）手搓KMP（BF略~）

1.next数组意义&求解：

**next[j]函数表明当模式中第j个字符与主串中相应字符失配时，在模式中需重新和主串中该字符进行比较的字符的位置。其定义如下：**

$$next[j]=\begin{cases}0, & j=1\\ \max(\{1\}\cup\{k|1<k<j\text{且}T.ch[1..k-1]=T.ch[j-k+1..j-1]\}), & j>1\end{cases}$$

- **j=1, next[j]=0, 这时i前进一个位置**
- **j=2, next[j]=1**

**为了使T的右移不丢失任何匹配成功的可能，当存在多个满足条件的k值时，应取最大的，这样向右“滑动”的距离最短，“滑动”的字符为j-next[j]个；**

位置下标从1开始

2.代码实现

```
int Index_KMP(string main,string pattern)
{    //完整KMP算法过程
    int i=pos,j=1;
    while(i<=main[0] && j<=pattern[0])
    {
        if(!j || main[i]==pattern[j]) //继续比较后继字符串
        {
            i++;
            j++
        }
        else j=next[j]; //pattern，即模式串向右移动
    }
    if(j>pattern[0]) return i-pattern[0];//匹配成功
    else return 0;
}
```

```
void get_next(string pattern,int *next)
{    //求next数组过程
    int i=1,j=0;next[1]=0;
    while(i<pattern[0])
    {
        if(!j || pattern[i]==pattern[j])
        {
            i++;
            j++;
            next[i]=j;
        }
        else j=next[j]; //next[1]到next[j]已经有值
    }
}
```

③高级辨析

*顺序结构和链式结构的优缺点比较：

顺序表：优点是 1）随机存取，可在 O(1)时间按序号取到元素，2）无需为表示元素间的逻辑关系增加额外存储，存储密度 100%，3）各种高级语言都容易实现；缺点是 1）插入删除的时间复杂度是 O(n)，2）存储分配需预先进行。(3 分)

单链表：优点是 1）插入删除时无需大量移动元素，只需修改指针，2）存储分配是现用现分配；缺点是 1）不可以随机存取，而是顺序存取，按序号取元素的时间是 O(n)，2）存储密度不是 100%，3）有些高级语言没有指针，不易实现。(3 分)

*链栈与顺序栈相比，优点是不会发生上溢

*线性表的链式存储结构中，头指针与头结点的区别：

要点：在线性表的链式存储结构中，头指针是指链表的指针，头指针具有标识作用，故常用头指针冠以链表的名字（3 分）。

头结点是为了操作的统一、方便而设立的，放在第一个元素结点之前，有头结点后，对在第一元素结点前插入结点和删除第一个结点，其操作与对其它结点的操作就统一了（2 分）；此时无论链表是否为空，头指针都不为空（1 分）。

*高级语言中系统实现函数调用的内在机制与栈的关系

4. 函数调用基于栈来实现。调用函数时，将参数、返回地址保存在栈中，并在栈中分配局部变量的存储空间；函数执行完返回时，释放栈中的数据区。每当调用一个函数，就在栈中形成一层数据区；每当函数退出，就将该层数据区退栈退掉。当前运行函数的数据区就在栈顶。(5 分)

# 2-1.一般树

2022年10月28日　　15:41

①基本概念

*默认元素值不重！

*二叉树是度<=2的有序树（不是=2！）（度仅限于子树的个数，而非图论中无向图的度的含义，即广义上的连通分支数！）

*层序遍历：默认下标从1开始

*前驱/后继节点：对一棵二叉树进行*序遍历，遍历后的顺序，当前节点的前/后一个节点为该节点的前驱/后继节点（具体哪个顺序看总体要求，如前序线索化则为前序遍历的顺序）
*二叉树的等价：两棵二叉树完全相等~
*n个结点的线索二叉树中，线索有n+1个
*图论性质：
设节点总数为n，ni为度数为i的节点的数量，i∈{0,1,2}，则：
　　!n2=n0+1
　　n0+n1+n2=n
　　n1=(n+1)%2
由此推得小结论：
　　**n0=floor((n-1)/2)+1**
　　**n1=(n+1)%2**
　　**n2=floor((n-1)/2)**

②分类介绍

*各种特征的二叉树定义：

　　*完全二叉树：只有最后一层的最右边有空缺

　　*满二叉树：没有单个的左孩子或右孩子（但未必只有最右边才有空缺！）

　　*完美二叉树：2^n-1~

　　*顺序二叉树：二叉树的顺序存储（层序遍历，即从顶层往下逐层分别从左往右遍历形成对应的数组）

*BST：

　　*引入二叉搜索树的目的是：加快查找结点的前驱或后继的速度

　　*构建：默认以每次的首（即最左边）元素为根节点，将其余分成两边~

　　*删除有左右子树的某结点后：左、右子树都能移上去填补（死规！）

*大顶堆：

堆可以定义为一颗 二叉树，树的节点包含键（每个节点一个键），并且满足下面两个条件：

1. **堆的形状**要求是一颗**完全二叉树**，意识就是说，树的每一层都是满的，除了最后一层最右边的元素可能有缺位
2. **父母优势**要求，就是说每一个节点的键都要大于或等于它子女的键。

*哈夫曼树/编码、三位等长编码：

　　*定义：

　　当用 n 个结点（都做叶子结点且都有各自的权值）试图构建一棵树时，如果构建的这棵树的带权路径长度最小，称这棵树为“最优二叉树”，有时也叫“赫夫曼树”或者“哈夫曼树”。
　　*带权路径长度(WPL)：（对所有节点：）Σ((深度-1)*权值)
　　*若初始森林中共有n棵二叉树，最终求得的哈夫曼树共有2n-1个结点（×：需进行2n-1次合并后才能剩下一棵最终的哈夫曼树）

　　*构建方法：

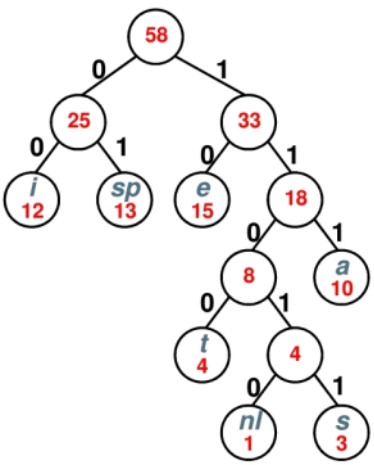
　　*哈夫曼树：

　　重复地让当前权值最小的两个根结点作为左右孩子，生成新的根结点，新结点权值为它们的权值之和，直至形成一颗二叉树

　　*哈夫曼编码：

〖例〗哈夫曼编码

| $C_i$ | a | e | i | s | t | sp | nl |
|---|---|---|---|---|---|---|---|
| $f_i$ | 10 | 15 | 12 | 3 | 4 | 13 | 1 |



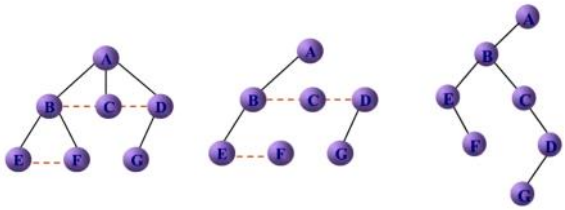

a : 111
e : 10
i : 00
s : 11011
t : 1100
sp : 01
nl : 11010

Cost = 3×10 + 2×15
+ 2×12 + 5×3
+ 4×4 + 2×13
+ 5×1
= 146

③转换

*森林、树、二叉树的相互转换：

（I）树→二叉树：

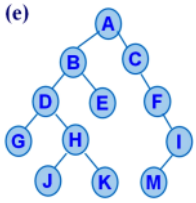


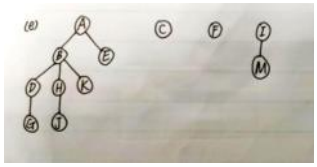
1. 在兄弟结点间加连线
2. 保留双亲与第一个孩子连线，删去与其他孩子的连线
3. 顺时针转动，层次分明

（II）森林→二叉树

1. 将森林中的每棵树转换成二叉树
2. 从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子，**（可以这样做的原因是，每次将树转化成二叉树的过程中，根结点都是只有左孩子）**当所有二叉树连起来后，此时所得到的二叉树就是由森林转换得到的二叉树。

（III）二叉树→树/森林



左孩子的左孩子就是原位置，左孩子的右孩子变为左孩子的亲兄弟，右孩子另起一棵树。

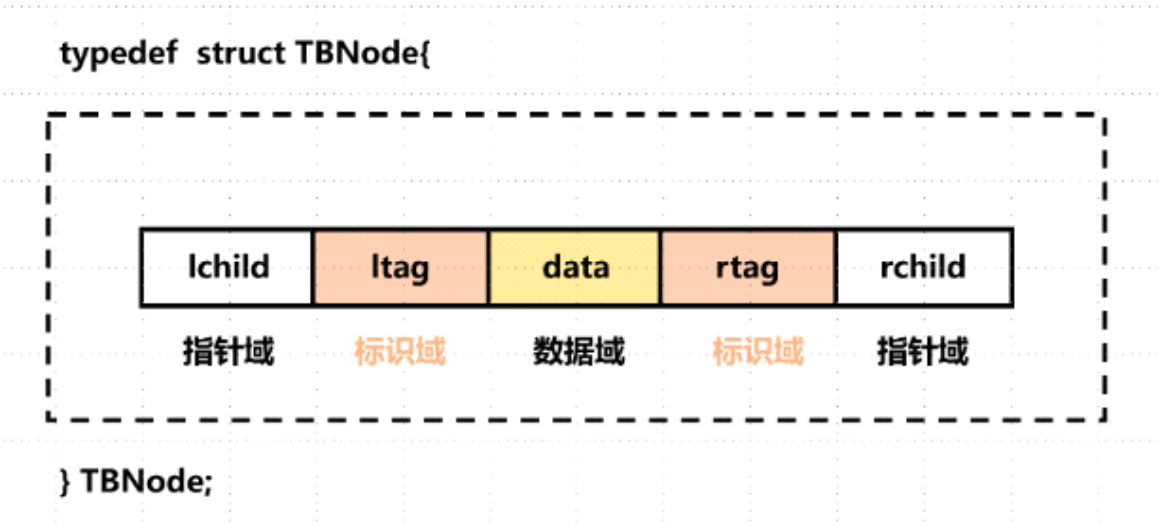
③其他

若二叉树前序遍历和后序遍历相同，则此二叉树一定为空或者只有一个结点

# 2-2.线索二叉树

2022年10月28日　　15:53

①基本概念




**标识域：**

1. 如果**ltag=0**，表示指向节点的**左孩子**。如果**ltag=1**，则表示**lchild为线索**，指向节点的**直接前驱**
2. 如果**rtag=0**，表示指向节点的**右孩子**。如果**rtag=1**，则表示**rchild为线索**，指向节点的**直接后继**

②画图构建
若已经有左右子树则不用再画，若缺失左/右子树则寻找相应的前驱/后继节点

③程序实现

（Ⅰ）前序

1.线索化

```
//prev需要传引用进去，才能记录下上次访问的节点，否则节点的右子树的线索化不能完成
void _PrevOrder_Th(Node* root, Node*& prev)
    {
        if (root == NULL)
        {
            return;
        }
        if (root->_LeftNode == NULL)
        {
            root->_LeftNode = prev;
            root->LeftTag = THREAD;
        }
        if (prev&&prev->_RightNode == NULL)
        {
            prev->_RightNode = root;
            prev->RightTag = THREAD;
        }
```

```
        prev = root;
        if (root->LeftTag == LINK)
        {
            _PrevOrder_Th(root->_LeftNode, prev);
        }
        if (root->RightTag == LINK)
        {
            _PrevOrder_Th(root->_RightNode, prev);
        }
    }
```

2.遍历

**遇到先对其进行进行访问，再对其左子树进行遍历访问，直到找到最左的那个节点；再根据线索化的指向对其右子树进行遍历访问。

```
void PreOrder()
    {
        Node* cur = _root;
        while (cur)
        {
            while (cur->LeftTag == LINK)
            {
                cout << cur->_data<<" ";
                cur = cur->_LeftNode;
            }
            cout << cur->_data<<" ";
            cur = cur->_RightNode;
        }
        cout << endl;
    }
```

（Ⅱ）中序

1.线索化

```
void _InOrder_Th(Node* root, Node*& prev)
    {
        if (root == NULL)
        {
            return;
        }
        else
        {
            _InOrder_Th(root->_LeftNode, prev);
            if (root->_LeftNode == NULL) //NULL表示为叶子节点
            {
                root->_LeftNode = prev;
                root->LeftTag = THREAD; //THREAD表示非叶子结点
            }
            if (prev&&prev->_RightNode == NULL)//这里的判断条件及处理没有太理解
            {
                prev->_RightNode = root;
```

```
                prev->RightTag = THREAD;
            }
            prev = root;
            _InOrder_Th(root->_RightNode, prev);
        }
    }
```

2.遍历

```
void InOrder()
    {
        Node* cur = _root;

        while (cur)
        {
            //找到最左的点
            while (cur->LeftTag == LINK)
            {
                cur = cur->_LeftNode;
            }
            cout << cur->_data << " ";
            while (cur->RightTag == THREAD)
            {
                cur = cur->_RightNode;
                cout << cur->_data << " ";
            }
            cur = cur->_RightNode;
        }
        cout << endl;
    }
```

（Ⅲ）后序

略，同前序~

# 3图

2023年2月23日　　11:19

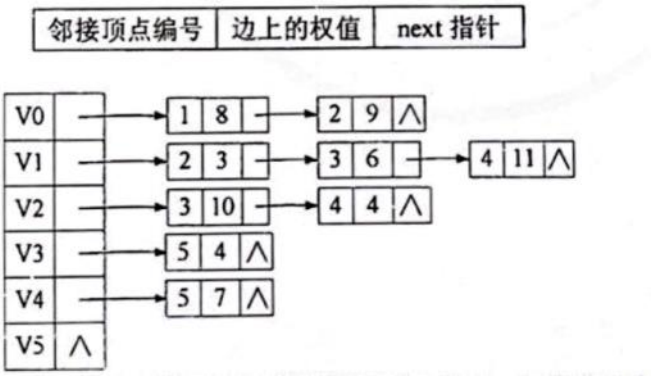
①存储方式

*邻接表、邻接矩阵

*下标默认从1开始

*都是无、有向图都能存储

*邻接表画法：（如果没有权值则为两个参数/"方格"）




*十字链表（了解即可~）：

将有向图的邻接表和逆邻接表结合在一起


*邻接多重表（了解即可~）：

存储无向图


②基本概念

两个点连通不一定是直接存在一条边将二者连接！

③方法

*拓扑排序

*每次筛选此刻入度为0的节点，由这些节点向其他节点遍历、入度--

*所有拓扑排序序列：同一轮中筛选出的（入度为0点）节点次序的所有排列方式（即每一轮的序列组合可能总数都为阶乘级别，再根据乘法原理相乘）

**BFS/DFS

遍历序列都为其生成树的前序遍历~
*关键路径（AOE）的概念&公式

- 事件$V_j$ 的最早发生时间$ve(j)$
  是从源点$V_0$到顶点$V_j$的最长路径长度。
- 事件$V_j$ 的最迟发生时间$vl(j)$
  是在保证汇点$V_{n-1}$在$ve(n-1)$时刻完成的前提下，事件$V_j$的允许的最迟开始时间。
- 活动$a_i$ 的最早开始时间$e(i)$
  设活动$a_i$在弧$<V_j, V_k>$上，则$e(i)$是从源点$V_0$到顶点$V_j$的最长路径长度。因此，$e(i) = ve(j)$。
- 活动$a_i$ 的最迟开始时间$l(i)$
  $l(i)$是在不会引起时间延误的前提下，该活动允许的最迟开始时间。$l(i)= vl(k) - dur(<j, k>)$。其中，$dur(<j, k>)$是完成$a_i$所需的时间。

*ve(j)、vl(j)公式：
*省流：ve(j)通过dp求，其他根据图中公式代入即可~
*详细：

- 从$ve(0) = 0$开始，向前递推

$$ve(j) = \max_i\{ ve(i) + dur(<V_i, V_j>) \},$$

$$<V_i, V_j> \in T,\ j = 1, 2, \ldots, n-1$$

其中$T$是所有以$V_j$为头的弧的集合。

- 从$vl(n-1) = ve(n-1)$开始，反向递推

$$vl(i) = \min_j\{ vl(j) - dur(<V_i, V_j>) \},$$

$$<V_i, V_j> \in S,\ i = n-2, n-3, \ldots, 0$$

其中$S$是所有以$V_i$为尾的弧的集合。

- $e(i) = ve(j)$, $l(i)= vl(k) - dur(<j, k>)$

- 时间余量 $l(i) - e(i)$
  表示活动$a_i$的最早开始时间和最迟开始时间的时间余量。
  $l(i) == e(i)$表示活动$a_i$是没有时间余量的关键活动。

*关键路径判断：

■为找出关键活动，需要求各个活动的 $e(i)$ 与 $l(i)$，以判别是否 $l(i) == e(i)$。

■为求得$e(i)$与 $l(i)$，需要先求得从源点$V_0$到各个顶点$V_j$的 $ve(j)$和 $vl(j)$。

# （II）算法共性

2023年2月1日　　20:54

①基本概念

*五大特性：有穷性、确定性、可行性、有输入、有输出

*问题规模：输入量

*O(n^2)：执行时间与n^2成正比！！（？）

*算法的实现依赖于采用的逻辑结构这句话是错的！（因为还依赖于很多东西。。）

②时间复杂度计算

主定理：

- 若有实数大于零($\varepsilon > 0$)，$f(n) = O(n^{log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{log_b a})$；
- 若 $f(n) = \Theta(n^{log_b a})$，则 $T(n) = \Theta(n^{log_b a}\ logn)$；
- 若有实数大于零($\varepsilon > 0$)，$f(n) = \Omega(n^{log_b a + \varepsilon})$，且有一个实数小于一($c < 1$)，使得较大的 $n$，满足 $af(\frac{n}{b}) \leqslant cf(n)$，这时候则 $T(n) = \Theta(f(n))$。

# 1.查找

2022年12月7日　　18:08

0总体架构

**下标默认从1开始

（Ⅰ）静态查找（仅支持判断存在性和查找元素特性）

1.线性表：顺序查找、二分/折半查找、分块查找（索引顺序表查找）

*2.树表

3.哈希表（计算式查找）

（Ⅱ）动态查找（还支持增删操作）

BST、AVL、B-/+树

①所用DS介绍

（Ⅰ）静态查找

*顺序查找（带岗哨/监视哨）（待查找元素下标从1开始，arr[0]为岗哨，存储此刻要查找的元素，意义在于减少了每次排查找时while都要对数组是否左边越界的判断，但本质上不能降低复杂度）：

```
int Search_Seq_2(SSTable S,ElemType key)
{
    S.elem[0]=key;
    int i=S.length-1;
    while(S.elem[i]!=key)
    { i--; }
    return i;
}
```

*二分查找

mid=(left+right)/2（即向下取整），下标（即初始left）默认从1开始

判定树（存放的元素是数值而不是下标！）



长度为len的有序表最多需要查找次数：cnt=ceiling(log2(len))

*分块查找：

在块内可以采用顺序查找或者折半查找，数据变化时可以通过修改指针实现，适用于既能快速查找又能适应动态变化需求的情况





*哈希表

装填因子α=线性表长度/散列表长度

1.构造方法：

*直接定址法（一次函数）

*数字分析法（去分布均匀的若干位或它们的组合）

*平方取中法（取关键字的平方的中间几位）

*折叠法（将关键字分为若干部分，取它们的叠加和）

*除留余数法（模M）

*随机数法

2.解决哈希法冲突的方法：

*开放定址法:

*Hi=(H(key)+di)MOD(m)（即向右逐一探测，若哈希表长度len相对于模值M还有空余空间则**在右边继续递增地开辟**（即此处MODm为**广义值**！），否则循环回0重新向右探测）

*Hi为最终的地址序列，H(key)为哈希函数，m为哈希表长

*取增量di的三种方法：

*线性探测再散列:di=i

*二次探测再散列:di=1,-1,4,-4

*随机探测再散列:di=伪随机数列

*链地址法（用链表~，链表中插入元素时）

画法：

**建立公共溢出区法

哈希函数值应当以同等（不是平均！）概率取其值域的每个值

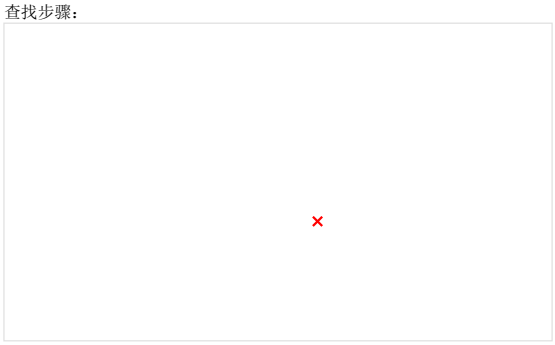
（II）动态查找

**B+、B-树（不算传统意义上的树，不作为机试的考点，只是应用于磁盘管理等）

*B-树（这里的-不是相对于+的减号，而只是一个连接符！）：



查找步骤：



*B+树（与B-树的区别）：

1.B+树内节点不存储数据，所有 data 存储在叶节点导致查询时间复杂度固定为 log n。而B-树查询时间复杂度不固定，与 key 在树中的位置有关，最好为O(1)。

2. B+树叶节点两两相连可大大增加区间访问性，可使用在范围查询等，而B-树每个节点 key 和 data 在一起，则无法区间查找。

3.B+树更适合外部存储。由于内节点无 data 域，每个节点能索引的范围更大更精确。

②ASL（小结论）

（I）静态查找（树表不讨论~）

*线性表：

*顺序:

*无序表：

ASL成功=(n+1)/2

ASL失败=n

*有序表：

ASL成功=(n+1)/2

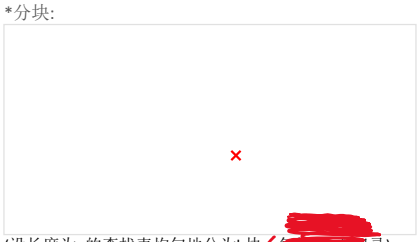
ASL失败=(n+2)/2

*二分:

ASL成功=（所有节点：）∑(判定树中节点的深度)/节点总数

ASL失败=（所有空子树：）∑(空子树的深度-1)/空子树的总数（同BST的ASLunsucc）

*分块:



(设长度为n的查找表均匀地分为b块，每块有s个记录)

*哈希表：

***线性中空也算一次查找，但链地址不算（即只有数值算次数，若地址为空则为0）！**

***二者的ASL失败除的都是m而非len**

***勿忘查找时可能要越出m-1！**

（II）动态查找

BST（AVL即为BST中的最好情况）：

ASL成功=~

ASL失败=((所有空子树:)∑(空子树的深度-1))/空子树的总数（同二分查找的ASLunsucc）

# 2.排序

2022年12月16日　20:33

一、总体关系

**下标除了归并排序从0开始以外，其他都默认从1开始

①分类：

（ı）内部排序（能在内存中完成，不需要访问外存，如软盘、硬盘）

*插入类：插入排序（直接、折半（即每次插入时用二分查找）、*2-路、*表插入、希尔排序）

*交换类：[*双向]冒泡排序、快速排序

*选择类：选择排序（简单、树形/锦标赛排序、堆排序）

*归并类：[2-路]归并排序（2-路就是一般的归并排序~）

*其他：基数排序（多关键字的排序、链式基数排序）、*桶排序

（ıı）外部排序（数量级很大，不能在内存中完成，需要访问外存）

无~

PS：唯一一个不需要基于关键字比较的算法：基数排序

①性质

| 排序方式 | 平均时间复杂度 | 最坏时间复杂度 | 最好时间复杂度 | 空间复杂度 | 稳定性 | 备注 |
|---|---|---|---|---|---|---|
| 快速 | $O(n\log_2(n))$ | $O(n^2)$ | $O(n\log_2(n))$ | $O(\log_2 n)$ | 不稳定 | 最坏比较次数$\frac{n(n-1)}{2}$ |
| 归并 | $O(n\log_2(n))$ | $O(n\log_2(n))$ | $O(n\log_2(n))$ | $O(n)$ | 稳定 | |
| 堆 | $O(n\log_2(n))$ | $O(n\log_2(n))$ | $O(n\log_2(n))$ | $O(1)$ | 不稳定 | |
| 冒泡 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 | 最坏比较次数$\frac{n(n-1)}{2}$ |
| 选择 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 | |
| 插入 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 | 最坏比较次数$\frac{n(n-1)}{2}$ |
| 希尔 | $O(n^{(1.3-2)})$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 不稳定 | |
| 基数 | $*O(d(n+r))$ | $*O(d(n+r))$ | $*O(d(n+r))$ | $*O(r)$ | 稳定 | *d为位数，r为基数 |
| 计数 | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | 稳定 | k是基数的范围 |

稳定性指的是原来若存在两个相同大小的元素，排序后是否按照原来顺序！！

（勿忘！）②灵活运用：

（1）当数据规模较小的时候，可以用简单的排序算法如直接插入排序或直接选择排序。

（2）当文件的初态已经基本有序时，可以用直接插入排序或冒泡排序。（此时快速排序最费时间）

（3）当数据规模比较大且较为无序时，应用快速排序。

（4）堆排序不会出现快排那样的最坏情况，且堆排序所需的辅助空间比快排要少，但这两种算法都不是稳定的。若要求排序时稳定的，可以考虑用归并排序。

*（5）归并排序可以用于内排序，也可以用于外排序。在外排序时，通常采用多路归并，并且通过解决长顺串的合并，产生长的初始串，提高主机与外设并行能力等措施，以减少访问外存额次数，提高外排序的效率。

二、总体思想&代码实现

①插入类

1.[直接]插入排序

（Ⅰ）总体思想：一共有n-1轮（i从1取到n-1），每次将第i+1个数插入到前面i个排好序的数中

（Ⅱ）代码实现：

设置岗哨/监视哨（排序元素下标从1开始，arr[0]为岗哨，存储此刻要插入排序的元素，意义在于减少了每次排序时while都要对数组是否左边越界的判断，但本质上不能降低复杂度）的插入排序：

```
void insert_sort( int arr[], int length)
{
    int i,j;
    for(i=2; i<=length; i++){
        arr[0] = arr[i];//将待插入元素存放到监视哨中
        for(j=i-1; arr[j]>arr[0]; j--)
            arr[j+1] = arr[j];//将大于待插入元素的全部向后移一个
        arr[j+1] = arr[0];//插入待插入元素
    }
}
```

2.希尔排序

（Ⅰ）总体思想：设置一个gap变量（增量），gap每次除以2并取整直到为1，每一轮中以gap为间隔的子数组分别各自内部排好序

（Ⅱ）代码实现：

//希尔排序其中的一趟

```
void ShellInsert(SqList &L, int dk)
/* 本算法和一趟直接插入排序相比，作了以下修改：(1) 前后
记录位置的增量是 dk, 而不是 1; (2) r[0] 只是暂存单元，不是哨兵。
当 j<=0 时，插入位置已找到。 */
{ for (i=dk+1; i<=L.length; ++i)
  if (LT(L.r[i].key, L.r[i-dk].key))
  { // 需将 L.r[i] 插入有序增量子表
    L.r[0] = L.r[i];          // 暂存在 L.r[0]
    for (j=i-dk; j>0 && LT (L.r[0].key, L.r[j].key); j-=dk)
          L.r[j+dk] = L.r[j];     // 记录后移，查找插入位置
    L.r[j+dk] = L.r[0];           // 插入
  }
} // ShellInsert
```

//按增量序列dlta[0...t-1]对顺序表L进行完整的希尔排序过程（即多趟）

```
void ShellSort(SqList &L, int dlta[], int t)
{
  for (int k=0; k<t; ++k)
    ShellInsert(L, dlta[k]); // 一趟增量为 dlta[k] 的
    插入排序
} // ShellSort
```

②交换类

1.冒泡排序

默认一共n-1轮，第i轮从1向右到n+1-i两两交换（即n-1次）

2.快速排序

（Ⅰ）总体思想：将每次的最左数作为基准数（枢轴值），两边各设置一个哨兵分别往中间走，遇到大于/小于基准数的就交换，直到哨兵相遇，此时基准数已经到了最终的位置，且其左右的数分别比它都小/大（实际上隐含了一颗递归树）（其中第一枢轴值为第一次此过程完成后两个哨兵交汇点的数组元素值）

（Ⅱ）代码实现：

*若要求输出每一轮结果（即中途必须老老实实交换值）：

```
void Qs(int *n,int start,int end)
{
    int temp=n[start],left=start,right=end;//一定要记录下初始的起止点！！！
    if(left>right) return;
    while(left<right)//真正的快排中途是会交换值的，只不过是我自己改写的简便写法不需要！！！
    {
        while(left<right && n[right]>=temp) right--;
        n[left]=n[right];
        while(left<right && n[left]<=temp) left++;
        n[right]=n[left];
    }
    n[left]=temp; //此时的left为哨兵相遇的地方
    Qs(n,start,left-1);//这里的起始点是start而非left！！！
    Qs(n,left+1,end);//同上！
    return;
}
```

*仅仅是为了省时间（简便写法，可不交换值）

```
void qs(int *n, int len)
{
        if(len<2) return;
        int t=n[0],i,*l,*r,llen=0,rlen=0;
        l=(int*)malloc(sizeof(int));
        r=(int*)malloc(sizeof(int));
        for(i=0;i<len;i++)
        {
                if(n[i]<=t) l[llen++]=n[i];
                else r[rlen++]=n[i];
        }
        qs(l,llen);
        qs(r,rlen);
        for(i=0;i<llen;i++) n[i]=l[i];
        n[llen]=t;
        for(i=0;i<rlen;i++) n[i+llen+1]=r[i];
        return;
}
```

（Ⅲ）特点：效率大多时候极高、不稳定

③选择类

1.树形/锦标赛排序

（Ⅰ）总体思想：

类似选择排序，但是选出最值的方法类似NBA季后赛的两两淘汰赛制，即对n个关键字两两比较，然后对其中ceiling(n/2)个较小者再以此类推地比较

（Ⅱ）代码实现：

略~

2.堆排序

（Ⅰ）总体思想：

*第一步：建立最大堆

先将数组按从左到右顺序以层序顺序填到堆中1-2调整：

算法：从第一个具有孩子的结点 i 开始（ i=⌊n/2⌋ ），如果以这个元素为根的子树已是最大堆，则不需调整，否则需调整子树使之成为堆。继续检查 i-1 ，i-2 等结点为根的子树，直到检查整个二叉树的根结点（其位置为 1 ）。

注意：

*每次必须让整个子树都满足，即可能得从上往下调整多层！

**若需要排序且两子树都满足条件，默认根据最大/小堆选择二者中更大/小的

*第二步：排序

然后每次删除堆顶点（当前最大值）并将其传到记录的数组里

（Ⅱ）代码实现：

*直接库函数实现（无论是要求输出每一轮结果还是仅仅是为了省时间）

```
#include<algorithm>
#include<functional>
using namespace std;
int heap[Max],ans[Max];
for(int i=1;i<=len;i++)
{
    make_heap(heap+1,heap+len+2-i,[less/greater<int>()]);//less为大顶堆
    ans[i]=heap[0];
    heap[0]=heap[len+1-i];
    //输出每一轮结果：
    //int j;
        //每一轮已排好序的元素：
        //for(j=1;j<=i;j++) cout<<ans[j]<<" ";
        //cout<<endl;
        //每一轮剩下的堆：
        //for(j=1;j<=len-i;j++) cout<<heap[j]<<" ";
}
```

*具体过程

```
//heap下标从1开始
void HeapAdjust(int *n, int p, int len)
{
    int i,temp;
    temp=n[p];
    for(i=2*p;i<=len;i*=2)
    {
        if(i<len && heap[i]<heap[i+1]) i++;
        if(temp>=heap[i]) break;
        heap[p]=heap[i];
        p=i;
    }
    heap[p]=temp;
}

int* HeapSort(int *heap, int len)
{
    int i,ans[Max];
    //构造大顶堆
    for (i=len/2;i>0;i--) HeapAdjust(heap,i,len);
    //依次取出当前的最大元素
    for (i=len;i>=1;i--)
    {
        ans[i]=heap[1];
        heap[1]=heap[i];
        HeapAdjust(heap,i,i-1);
    }
    return ans;//ans为升序排列！
}
```

⑤归并类

[2-路]归并排序

（Ⅰ）总体思想：将数组不断两两拆分成子数组，直到不能再分后，先各自排好序，后不断递归将子数组两两合并，直到全部合并

（Ⅱ）代码实现：

```
#include<stdio.h>
#define ArrLen 20
void printList(int arr[], int len)
{
   int i;
   for (i = 0; i < len; i++) printf("%d\t", arr[i]);
}

void merge(int arr[], int start, int mid, int end)
{
        int result[ArrLen];
        int k = 0;
        int i = start;
        int j = mid + 1;
        while (i <= mid && j <= end)
    {
                if (arr[i] < arr[j]){
                    result[k++] = arr[i++];
        }
        else{
                result[k++] = arr[j++];
        }
        }
        if (i == mid + 1) {
                while(j <= end)
                        result[k++] = arr[j++];
```

```
        }
        if (j == end + 1) {
                while (i <= mid)
                        result[k++] = arr[i++];
        }
        for (j = 0, i = start ; j < k; i++, j++) {
                arr[i] = result[j];
        }
    }
}

void mergeSort(int arr[], int start, int end)
{
        if (start >= end)
                return;
        int mid = ( start + end ) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        merge(arr, start, mid, end);
}

int main()
{
        int arr[] = {4, 7, 6, 5, 2, 1, 8, 2, 9, 1};
        mergeSort(arr, 0, 9);
        printList(arr, 10);
        system("pause");
        return 0;
}
```

（lll1）特点：效率高、稳定

⑥其他类

基数排序

（Ⅰ）总体思想：

由低位依次向更高位，每轮（一位）按基数（0-9，a-z）排好序（*一样则按原序）（若位数不同则可以填充最小值）

（Ⅱ）代码实现：三维数组即可，略–

# ⑤死规

2023年2月23日　　9:35

*算法设计题尽量不用全局变量实现！
*链表设计：
一般先写Q=P->next/prior;，用Q来操作！（此时Q指向的一直是原来p的[直接]后继结点的地址，即使是删除后！）
**竖直向下的节点：默认为左子树

# C[++]

2023年1月31日 16:45

# *0容器、迭代器概览

2022年1月25日　　14:29

①容器

**通过迭代器的begin(),end()遍历~**

（Ⅰ）顺序容器

*排放顺序是与其加入容器时的位置相对应
*合法语法为==、!=、>、<
*勿忘对应头文件（<vector>等等）
*具体分类：
1.声明：vector<T>+名称（下同）：动态数组（通常用它）
2.list：双向链表，双向顺序访问/遍历(链表不支持元素的随机访问），list在任何位置的插入和删除速度都很快，很方便。
3.string：字符串容器
注意：若用的是C++中的功能，调用的库函数必须是<cstring.h>!

（Ⅱ）关联容器

*元素的位置由相关联的关键字值决定
*平衡二叉树（查找、修改元素各种操作时间复杂度都为O(n))
*多为const_...：只能用==、!=进行比较
*声明：set<T>+名称（下同）, multiset, map, multimap(有无序、可否有重复元素）

注意：#include<...>头文件勿忘！()

*具体分类：

1.multiset<T> st;
*存放并自动排序元素(默认为从小到大）
*multiset<int,greater<int>>:从大到小排序的multiset（也可以是自定义的规则Rule1）
2.set 插入元素可能不成功（因为元素不能重复）
3.map:键值对
*pair<T1,T2>类型等价于：

```
struct{
        T1 first/key;
        T2 second/value;
};
make_pair(T1,T2)//生成一个pair<T1,T2>变量
```

```
typedef map<string,int>MP;
MP mp;
mp[key]=[value];
```

4.multimap<T1,T2>mp;
*元素按照first排序（默认升序），并可以按first进行查找

②迭代器
*每个种类的容器都有其对应的迭代器类型，且迭代器本身也可以选择定义类型（二者为组合关系，并不冲突）
*由容器决定的迭代器具体分类：输入/输出/前向/双向(p++或--,不能直接访问p[i]、比大小、互相相减)/随机访问(p+或-i)迭代器

| | |
|---|---|
| array | 随机访问迭代器 |
| vector | 随机访问迭代器 |
| deque | 随机访问迭代器 |
| list | 双向迭代器 |
| set / multiset | 双向迭代器 |
| map / multimap | 双向迭代器 |
| forward_list | 前向迭代器 |
| unordered_map / unordered_multimap | 前向迭代器 |
| unordered_set / unordered_multiset | 前向迭代器 |
| stack | 不支持迭代器 |
| queue | 不支持迭代器 |

*定义类型及整个迭代器的声明方式：

| 迭代器定义方式 | 具体格式 |
|---|---|
| 正向迭代器 | 容器类名::iterator 迭代器名; |
| 常量正向迭代器 | 容器类名::const_iterator 迭代器名; |
| 反向迭代器 | 容器类名::reverse_iterator 迭代器名; |
| 常量反向迭代器 | 容器类名::const_reverse_iterator 迭代器名; |

*初始化（同指针）（即构造函数）：iter1=ivec.begin();//指向容器ivec首元素
*具体功能：
*名称.find(v):若找到元素值i,返回一个迭代器，指向key==1，若未找到，返回end()
*名称.count(v):遍历并返回对应元素值i出现的次数
*名称.insert(i,v)在下标为i处插入元素，值为v（PS: “,v”可省略)
*名称.erase(i)删除迭代器i指向的元素

# ①常用库函数

2022年7月5日　20:45

零、内置（无需导入）

strlwr、strupr：字符串全部小写/大写

一、algorithm库
*min、max、swap
*gcd(x,y)
*reverse(lst+1,lst+n+1)
*unique(lst+1,lst+n+1) //返回最后一个数的地址
*partial_sum(vec.begin(),vec.end(),back_inserter(dst) //求前缀和
*next_permutation(vec.begin(),vec.end()) //求全排列

二、string库

```
#include<iostream>
#include<string>
```

（Ⅰ）获取长度
length()/size()
（Ⅱ）拷贝
1.一般：直接=即可
2.特别的：
*string str2=string(str1,index) 从字符串str1第index个字符开始到结束（注：此处str2需为string对象，若为字符数组则变成前index个字符，下同！）
*string str2=string(str1,index,len) 从字符串str1第index个字符开始，拷贝len个字符
（Ⅲ）修改
*追加：str1.append(str2)
*插入：str1.insesrt(index,str2) 在str1的index位置插入字符串str2
*删除：str.erase(index,len)将str1从index开始的len个字符删除
*替换：str1.replace(index,len,str2)将str1从index开始的len个字符替换成str2
（Ⅳ）比较
str1.compare(str2) 结果为±1，0
（Ⅴ）交换
str1.swap(str2)
（Ⅵ）查找位置
str1.find(str2)查找str2在str1中第一次出现的位置

str1.rfind(str2)查找str2在str1中最后一次出现的位置

三、STL库

*使用参考顺序：单独的排序/查找功能→借助set或者map自动排序

提醒：勿忘#include以及using namespace std;!

①广义的sort函数排序

```
sort(数组名1+n1,数组名2+n2,排序规则结构名())
struct 结构名//(例如：StudentRule1)
 {
    bool operator()(const 数据类型&a1, const 数据类型&a2) const
  {
   //建立一定的排序规则
   //若a1应该放在a2前面，则返回TRUE，反之返回FALSE
  if...
      return True;
 }
}
```

②（二分）查找（实际上应用于容器返回迭代器，此处为数组返回指针）
（1）binary_search(数组名+n1, 数组名+n2,值,（可有可无：排序结构名）
*在从小到大排好序的基本类型数组上进行二分查找
*查找区间为左闭右开！
*仅返回bool值！
(2)lower_bound(数组名+n1, 数组名+n2,值,(排序规则名()))
*返回一个指针*p
*这个*p查找区间内下标最小的不小于对应值的元素，若找不到则指向下标为n2的元素
(3)upper_bound():同理~（下标仍为最小！）
(4)find(数组名+n1, 数组名+n2,值):可以不排好序

③其他（数组操作）
*string类型输出时：

```
printf("%s",string.c_str()); //这个不建议
cout<<string<<… //勿忘include<iostream>!
```

*find()函数

```
*p=find(left,right,target) //left和right相当于sort函数中参数的指针形式，左闭右开，其指向的是在区域内查找到的第一个目标元素；如果查找失败，则该迭代器的指向和right相同
```

*merge()函数

```
merge(left1,right1,left2,right2,result) //用于合并两个同时升/降序的数组，参数都是迭代器
```

# ②unordered_set

2023年1月31日　　11:07

*就是集合（Python里的set）~

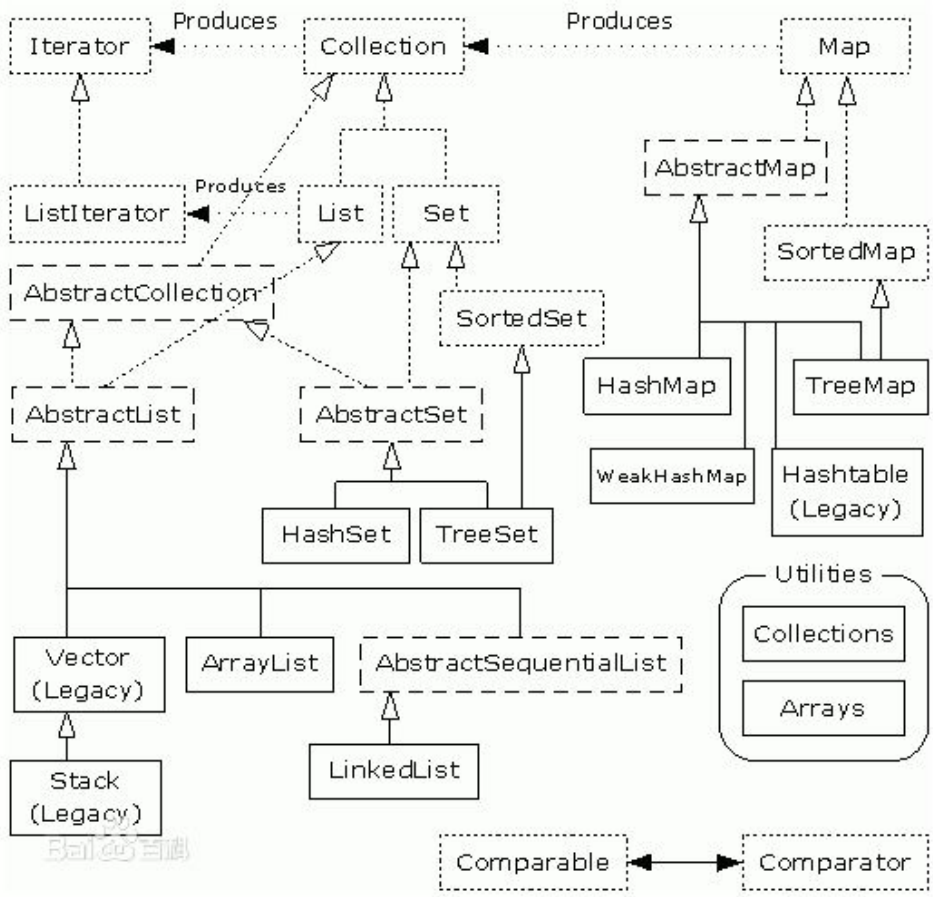
```
#include<unordered_set>
using namespace std;
unordered_set <string> s;
s.size();
s.empty();
s.find(ele);
    //没找到则返回最后元素之后位置的迭代器（和s.end()相等说明没找到，类似于一般迭代器）
s.count(ele);
s.insert(ele);
s.erase(ele);
s.clear();
```

# Java

2023年1月31日 16:45

# *DS类架构图

2023年2月1日　　20:41



# 常用内置函数

2023年1月31日 13:09

*记不清具体包名就`import java.util.*;`

*min,max

*最大公约数：（无需调包！）

```
1 BigInteger gcd=x.gcd(y);
```

# ①栈、队列、链表

2023年1月30日 22:49

*注：双向队列也能实现栈~

# C[++]

2022年4月24日　　15:33

零、共性操作

```
#include<Lib_Name>
using namespace std;
[std::]Lib_Name ds[(length[,ele])];
//Lib_Name即DS类型,ds即DS的名称,不加ele时默认为0
b=ds.empty();
len=ds.size();
val=*iter;//获取迭代器所指元素的值（由此修改）
*p=find(iter_left,iter_right,target_val); //若查找失败，返回iter_right
sum=ds.count(val);//计数
    //合并操作除了要求两个容器都已经排好序的merge函数外无库函数，直接手搓~
ds.clear();
ds.sort(iter_left,iter_right,comp_func);
      //逆向排序与其写compare函数调用，不如直接手搓循环~
for(i=ds.begin();i!=ds.end();i++){…}//遍历
```

一、栈

①应用方向

单调栈：按序遍历，栈中放置的因素单调递增/减，若此时遍历到的元素 < />栈末尾的元素（如果栈非空），则栈一路往回剔除元素直到此元素放入后可以继续单调

②操作

（Ⅰ）手搓（推荐）~

（Ⅱ）库函数

```
#include<stack>
t=s.top();//获取栈顶元素
s.pop(); //弹出栈顶元素，但不返回其值
s.push(n);//入栈

```

二、队列

*优先队列：相当于完全二叉树的一维数组结构~（类似于堆）

①单向队列Queue

（Ⅰ）手搓~

（Ⅱ）库函数（推荐）

②双向队列Deque

```
#include<queue>
val=q.front(); //获取队首
val=q.back(); //获取队尾
q.push(n);//入队
q.pop(); //出队，无返回值

```

（Ⅰ）手搓~

（Ⅱ）库函数（推荐）

```
#include<deque>
d.emplace/pop_front/back(val);//emplace比push省去复制移动元素过程
d.insert(index,val);
d.erase(index);
```

三、链表

①使用情况：需要频繁在头、尾部增删操作时

②操作

（Ⅰ）库函数（极力推荐）

0.创建：

```
#include<list>
using namespace std;
list<string> l;//声明一个空链表
```

```
4  list<string> l(n);//声明一个含n个元素的链表并全部初始化为0
```

1.增：

```
1  l.insert/emplace();          // 插入一个元素到list中(emplace更高效)
2  l.push_back();               //在list的末尾添加一个元素
3  l.push_front();              //在list的头部添加一个元素
4  l.merge();                   //合并两个list
```

2.删：

```
1  l.erase();                   //删除一个元素
2  l.pop_back();                //删除最后一个元素
3  l.remove();                  //从list删除指定元素
4  l.unique();                  //删除list中重复的元素
```

3.改：

```
1  l.reverse();                 //把list的元素倒转
```

4.查：

```
1  list<int>::iterator itr=l.begin();   // 返回第一个元素的迭代器
2  l.front();                   //返回第一个元素
3  l.back();                    //返回最后一个元素
```

5.排序：

```
1  l.sort();                    //给list排序
```

（Ⅱ）手搓（**仅限笔试使用！**）

1.构建

```
1  typedef struct node
2  {
3      int val;
4      struct node *next=NULL;
5  }node;
6
7  head=(node*)malloc(sizeof(node));
8  head->val=value;
9  t=head;
10 while(t->val !=NULL)
11 {
12         ...
13         t=t->val;
14 }
```

2.增删改查
略~

# Python

2022年2月4日　　16:21

*零、共性操作

```
b=ds.empty()
length=len(ds)
val=ds[index]
index=ds.index(val) #查找，若为找到则返回-1
cnt=ds.count(index) #计数
ds1.extend(ds2) #合并
ds.clear()
ds[left:right]=sorted(ds[left:right],reverse=b)
#局部/整体排序(用sort无效！)
for i/i,j in ds/enumerate(ds):…
```

一、栈

①应用方向

单调栈：按序遍历，栈中放置的因素单调递增/减，若此时遍历到的元素＜/>栈末尾的元素（如果栈非空），则栈一路往回剔除元素直到此元素放入后可以继续单调

②操作

**手搓和库函数一样，都是通过列表直接实现！**

```
lst.append/extend()
ele=lst[-1]
lst.pop()
ele=lst.popleft()
```

二、队列

由于Deque库函数语句全覆盖Queue，所以此处双向队列Deque和单向队列Queue专门不区分。

（Ⅰ）手搓~

（Ⅱ）库函数（推荐）

```
from collections import deque
d=deque() #Deque()（双向队列） 创建一个空的新 deque。它不需要参数，并返回空的 deque
d.append/extend[left]()
[val=]d.pop[left]()
```

**优先队列：相当于完全二叉树的一维数组结构~（类似于堆）

三、链表

**无库函数，只能手搓：**

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

（增删改查操作略~）

*一直只用a、a.next按某种规律遍历某个链表后依然可以得到一个完整的新链表！（最后返回的表头也得是a）

*链表变换时注意末尾的next置空！

# Java

2023年1月29日　11:03

零、共性操作

```
len=ds.size();
//判空通过长度即可~
b=ds.contains(val);//是否存在
val=ds.get(index);//获取值
    //没有统一的查找、计数、合并函数，只能手搓~
ds.clear();
for(DataType i:ds){…}
    //遍历（用下标i从0到.size()-1,用.get()操作值亦可！）
ds.sort(ds,left_index,right_index);
    //逆向排序与其写compare函数调用，不如直接手搓循环~
```

一、栈

①应用方向

单调栈：按序遍历，栈中放置的因素单调递增/减，若此时遍历到的元素 < />栈末尾的元素（如果栈非空），则栈一路往回剔除元素直到此元素放入后可以继续单调

②操作

（Ⅰ）手搓

适当使用Array类方法~

（Ⅱ）调包

Stack类方法（使用时直接tm=方法名(…)即可）：

```
import java.util.*;
```

| | |
|---|---|
| 1 | boolean empty()<br>测试堆栈是否为空。 |
| 2 | Object peek( )<br>查看堆栈顶部的对象，但不从堆栈中移除它。 |
| 3 | Object pop( )<br>移除堆栈顶部的对象，并作为此函数的值返回该对象。 |
| 4 | Object push(Object element)<br>把项压入堆栈顶部。 |
| 5 | int search(Object element)<br>返回对象在堆栈中的位置，以 1 为基数。 |

*PS:Java堆栈Stack类已经过时，Java官方推荐使用Deque（见下）替代Stack使用。

二、队列

①单向队列Queue（用ArrayList类实现）

```
import java.util.ArrayList;
ArrayList<DataType>array_name=new ArrayList<DataType>();
array_name.add()//追加
array.name.set(index,reset)//修改
array_name.remove(index)//删除
array_name.sort();//排序
```

②双向队列Deque

```
import java.util.Deque;//Deque类实现的子类有LinkedList、ArrayDeque
Deque <ArrayDeque> d=new Deque<ArrayDeque>();
d.peek/getFirst/Last();
    //返回队首元素,若为空peek返回null,get抛异常
d.offer/addFirst/Last(ele);
    //如果不能添加元素,offer返回false,add会抛异常
d.poll/removeFirst/Last();
    //若为空poll返回null,remove抛异常
```

三、链表

①应用方向：需要频繁在头、尾部增删操作时

②操作

**就直接调包，不搞手搓了！**

```
import java.util.LinkedList;
```

```
Linklist<String> l;
l.getFirst(…);//获取头结点（尾部唯为getLast）
l.add(…);//尾部追加
l.removeFirst(…);//移除头结点（尾部唯为removeLast）
//修改操作包内无函数，必须遍历再改~
```

# ②并查集（BFSet）

2023年1月30日　　22:48

一、C[++]

0应用方向

源头感：适合求解类似于BFS的最值问题和“蔓延问题”，及求图论中的连通性（类似于前面的蔓延问题）、环数及边数的问题

①创建&查询（路径压缩）

```
#include <unordered_map>
unordered_map <float, float> f;//支持string型
float find(float x)

{

        if(!f.count(x)||f[x]==x) return f[x]=x;
        return f[x]=find(f[x]);
}
```

②合并（比较规模大小，选择更省时的方式）

```
void union(float x,float y)
{
        f[find(x)]=find(y);
}
```

二、Python

0应用方向

源头感：适合求解类似于BFS的最值问题和“蔓延问题”，及求图论中的连通性（类似于前面的蔓延问题）、环数及边数的问题

①创建&查询（路径压缩）

```
f={}
def find(x):
    f.setdefault(x,x)
    if f[x]!=x:f[x]=find(f[x])
    return f[x]
```

②合并（比较规模大小，选择更省时的方式）

```
def union(x, y):
    f[find(y)] = find(x)
```

三、Java

0应用方向

源头感：适合求解类似于BFS的最值问题和“蔓延问题”，及求图论中的连通性（类似于前面的蔓延问题）、环数及边数的问题

①创建、查询、合并

```
import java.util.HashMap;
public class BFSet{
    HashMap<float,String> f=new HashMap<float,String>();
    //查询
    public String find(float key)
    {
        if(!f.containsKey(key))
            f.put(key,key);
        if(f[key]!=key)
            f[key]=find(key);
        return f[key];
    }
```

```
    //合开
    public void(float key1,float key2)
    {
        f[find(key1)])]=find(key2);
    }
}
```

# ③散列表（哈希表）

2022年4月28日　　16:45

①作用

*数组下标太大，内存不够，需适当“压缩”

*时空的权衡

*用键值对的方式进行高效“匹配式”查询（即if …==…），如多个元素出现次数的计数（字典）

②应用方法：

*构造散列函数的方法：除法取余、乘法取小数部分（集中在0-1）后乘以某常数取整（再略做分散）

*多个元素被操作至同一个单元中时：构造链表将其组合/构造二维数组(下标可以分别是取模的商和余数）

*关键在于构造键值对！

注：

*有元素加到哈希表某一处时（若此处还未开出数组）再开出数组！

*字典就是一个哈希表！（无需额外处理，直接常规操作就行了）

*想要手动“==”判断或计数时勿忘字典的存在以及Counter函数！（写得快而且时间复杂度更低!）

③库函数（**这个不能手搓~**）

***获取某个值都是通过val=Map_name[key]!**

*排序无意义，没写~

（Ⅰ）C[++]

```
#include <unordered_map>
unordered_map <float, string> f;
f.size();
f.empty();
f.find(val); //没找到就指向.end()
f.count(key);
f.insert(key,value);
f.erase(key)/(left_iter,right_iter);
f.clear();
for(i=f.begin();i!=f.end();i++){…}//遍历
```

（Ⅱ）Python

```
b=ds.empty()
length=len(ds)
val=ds[index]
index=ds.index(val) #查找，若为找到则返回-1
cnt=ds.count(index) #计数
ds[index]=val #添加
ds1.update(ds2) #合并
ds.pop(key) #删除
```

```
ds.clear()
ds[left:right]=sorted(ds[left:right],reverse=b)
    #局部/整体排序(用sort无效！)
```

（Ⅲ）Java

```
import java.util.HashMap;
HashMap<float, String>hm=new HashMap<float, String>();
hm.isEmpty();
len=hm.size();
hm.containsKey();//是否存在某键
val=hm.get(key);//获取键对应的值
//无查找函数，只能手搓遍历~
hm.put(key,value);
hm.remove(key);
hm.clear();
hm.keySet()/values();//获取所有键/值
```

# C[++]&数组（所有语言，仅限于①②③需要）

2023年1月28日 22:28

# ①基本操作

2022年4月24日　　17:15

①定义

```
struct TreeNode
{
        int val;
        struct TreeNode *left;
        struct TreeNode *right;
};
typedef  struct TreeNode TreeNode;
```

②构建

（这里为空时的值有两种，整型的-1和字符型的'#'）

（Ⅰ）数组实现（tree数组设置为全局变量，数组从零开始）（**若从1开始，则为*2和*2+1！！！**）

```
void BuildTree(int root)
{
        int temp;
        cin >> temp;
        if (temp == -1)
                return;
        else
        {
                tree[root] = temp;
                BuildTree(root * 2 + 1);
                BuildTree(root * 2 + 2);
        }
}
```

（Ⅱ）结构体指针实现

```
TreeNode* create() //中序遍历输入
{
        int val;
        cin>>val;
        if (value == '#') return NULL;
        else
        {
                TreeNode *root;
                root = (TreeNode*)malloc(sizeof(TreeNode));
                root->val = value;
                root->left = create();
                root->right = create();
                return root;
        }
    }
```

③前/中/后序遍历

（仅展示中序遍历，另外两个略）

（Ⅰ）数组实现

```
void MiddleOrder(int root)
{
        if (tree[root] <= 0)
                return;
        MiddleOrder(root * 2 + 1);
        cout << tree[root] << " ";
        MiddleOrder(root * 2 + 2);
```

```
}
```

（Ⅱ）结构体指针实现

```
int book[maxsize];
void midorder(TreeNode *root)
{
        if(!root) return;
        midorder(root->left);
        book[l]=root->val;
        l++;
        midorder(root->right);
}
```

④层序遍历
*输入形参为二叉树结点，输出result数组名（相当于指针）
*用l记录lay的长度，用l=0将其清空

*用memcpy(stack,lay,l)将lay复制给stack（库函数？）

*result可以用队列，也可以用length记录其长度然后一个个加~

（Ⅰ）结果放在一个一维列表中：

1.数组实现

只需将原数组中不为-1的值全部剔除即可！

2.结构体指针实现

```
void Levelorder(TreeNode *root)
{
    TreeNode que[100],t;
    int front=0,rear=0;
    if(root==NULL) return;
    que[++rear]=root;
    while(front!=rear)
    {
        t=que[++front];
        cout<<t->data;
        if(t->left) que[++rear]=t->left;
        if(t->right) que[++rear]=t->right;
    }
    return;
}
```

*（Ⅱ）结果逐层放在二维列表中：

1.数组实现

```
level=1;//头结点为第一层，level_order[i]用来记录第level层的遍历结果，每层结果从下标0开始
while(pow(2,level-1)-1<=max)
{
    int temp_len=0;
    for(int i=pow(2,level-1)-1;i<=pow(2,level)-2;i++)
    {
        if(tree[i]!=-1) level_order[level][temp_len++]=tree[i];
    }
    level++;
}
```

2.结构体指针实现

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
```

```
        if not root:
            return
        result = []
        stack = [root]
        while stack:
            lay=[]
            lay_value=[]
            for node in stack:
                lay_value.append(node.val)
                if node.left:
                    lay.append(node.left)
                if node.right:
                    lay.append(node.right)
            stack = lay
            result.append(lay_value)
        return result
```

# ②基本问题

2022年4月28日　　8:24

（所有问题的数组实现方法都显而易见，略）

①求树的高度

```python
def maxDepth(self, root):
        if not root:  # 递归边界
            return 0
        else:
            l = 1 + self.maxDepth(root.left)  # 递归
            r = 1 + self.maxDepth(root.right)
            return max(l, r)

```

②求叶子结点的个数

qq即字符'#'的个数除以2~

③找父节点

```java
public TreeNode getParent(TreeNode root,TreeNode p)
{
    if(root == null ||root.left == p || root.right == p) return root; //树为空，或者p是root的子节点，返回root
    TreeNode left = getParent(root.left,p); //root的左孩子 是否为p的父节点
    if(left != null) return left;  //是的话，return left，无需找右子树
    TreeNode right = getParent(root.right,p); //root的右孩子 是否为p的父节点
    if(right != null) return right;  //是的话，return right
    return left; //左右子树都不包含p，返回 null; (return right 同样是null)
}

```

# ③二叉排序/搜索树（BST）

2022年5月19日　　20:05

①构建（**此处不使用一次次插入的方法！**）

（Ⅰ）数组实现

```
void Create(int root,int *n,int len)
{
    if(!len) return;
    BST[root]=n[1];
    int small[10001],big[10001],len_s=0,len_b=0,i;
    for(i=2;i<=len;i++)
    {
        if(n[i]<n[1]) small[++len_s]=n[i];
        else big[++len_b]=n[i];
    }
    Create(2*root,small,len_s);
    Create(2*root+1,big,len_b);
    return;
}
```

（Ⅱ）结构体实现

```
TreeNode* Create(int *n,int len)
{
    if(!len) return NULL;
    TreeNode *root;
    root=(TreeNode*)malloc(sizeof(TreeNode));
    root->val=n[1];
    int small[10001],big[10001],len_s=0,len_b=0,i;
    for(i=2;i<=len;i++)
    {
        if(n[i]<n[1]) small[++len_s]=n[i];
        else big[++len_b]=n[i];
    }
    root->left=Create(small,len_s);
    root->right=Create(big,len_b);
    return root;
}
```

②插入

（Ⅰ）数组实现

```
int BST[Max]={0};
//下标从1开始，(即左右子树为*2和*2+1)
//Max需要比较大使得总能有位置插入
//若BST中允许出现0，则换一个绝对值大的负数

bool Insert(int root,int val)
//主函数中调用函数语句为"Insert(1,val);",即从下标1开始查询插入位置
{
    if(BST[root]==val) return false;
    else if(!BST[root]) //仍为缺省值0，说明还没插入过，还是空的
    {
        BST[root]=val;
        return true;
    }
    else if(BST[root]<val) return Insert(2*root+1,val);
    else return Insert(2*root,val);
}
```

（Ⅱ）结构体实现

```
bool Insert(BSTree *bst, int key)
{
        if (NULL == *bst)  //空树
        {
                *bst = BuyNode(key);    //插入根节点
```

```
                return true;
        }
        BSTNode *p;
        //先在二叉排序树中查找要插入的值是否已经存在
        if (!Search(*bst, key, NULL, &p))  //如果查找失败，则插入；此时p指向遍历的最后一个节点
        {
                BSTNode *pNew = BuyNode(key);
                if (key < p->data)  //将s作为p的左孩子
                {
                        p->lchild = pNew;
                }
                else if (key > p->data)  //将s作为p的右孩子
                {
                        p->rchild = pNew;
                }
                return true;  //插入成功
        }
        else
        {
                printf("\nThe node(%d) already exists.\n", key);
        }
        return false;
}
```

③遍历（略）（若为数组实现则参考“二叉树的基本操作”一栏中的数组实现二叉树遍历）
④查找

（Ⅰ）数组实现

```
bool Search(int root,int val)
{
    if(!BST[root])//已经为叶子节点
        return false;
    else if(BST[root]==val) return true;
    else if(BST[root]<val) return Search(2*root+1,val);
    else return Search(2*root,val);
}
```

（Ⅱ）结构体指针实现

```
bool Search(BSTree bst, int key, BSTree f, BSTree *p)  //查找成功时，p指向值为key的节点。如果查找失败，则p指向遍历的最后一个节点
{
        if (!bst)
        {
                *p = f;
                return false;
        }
        if (bst->data == key)  //查找成功，直接返回
        {
                *p = bst;
                return true;
        }
        else if (bst->data < key)
        {
                return Search(bst->rchild, key, bst, p);
        }
        return Search(bst->lchild, key, bst, p);
}
```

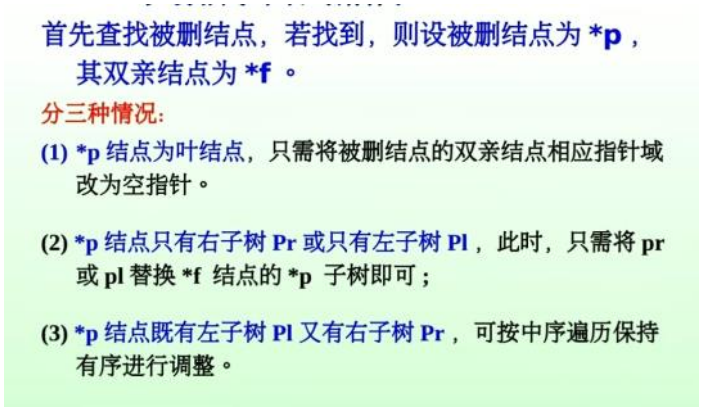
⑤删除

（Ⅰ）数组实现

查找到此节点并恢复缺省值（原始数据）

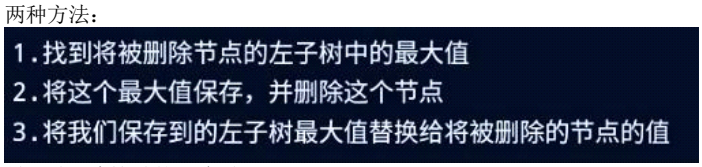
（Ⅱ）结构体指针实现

1.具体步骤：




（3）的详解：

两种方法：




另一种即为这种的对称型，同理~

2.代码实现：

```
/*
删除分三种情况:
(1)被删除的节点无孩子，说明该节点是叶子节点，直接删
(2)被删除的节点只有左孩子或者右孩子，直接删，并将其左孩子或者右孩子放在被删节点的位置
(3)被删除的节点既有右孩子又有右孩子
*/

BSTNode* FindParent(BSTree bst, BSTNode *child)
{
        if (NULL == bst)
        {
                return NULL;
        }

        if (bst->lchild == child || bst->rchild == child)
        {
                return bst;
        }
        else if(NULL != bst->lchild)
        {
                FindParent(bst->lchild, child);
        }
        else if (NULL != bst->rchild)
        {
                FindParent(bst->rchild, child);
        }
}

void Delete(BSTree *bst, int key)
{
        if (NULL == *bst)
        {
                exit(1);  //空树直接报错
        }
        BSTNode *p;
        BSTNode *f = NULL;
        BSTNode *q, *s;
        if (Search(*bst, key, NULL, &p)) //确实存在值为key的节点,则p指向该节点
        {

                if (NULL == p->lchild && NULL != p->rchild)  //无左孩子,有右孩子
                {
                        q = p->rchild;
                        p->data = q->data;     //因为两个节点之间本质的不同在于数据域的不同，而与放在哪个地址没有关系
                        p->rchild = q->rchild;
                        p->lchild = q->lchild;
                        free(q);
                }
                else if (NULL == p->rchild && NULL != p->lchild)  //无右孩子,有左孩子
                {
                        q = p->lchild;
                        p->data = q->data;
                        p->rchild = q->rchild;
                        p->lchild = q->lchild;
                        free(q);
                }
                else if (NULL != p->rchild && NULL != p->lchild) //既有左孩子，又有右孩子
                {
```

```
58              {
59                      q = p;
60                      s = p->lchild;    //找左孩子的最右孩子
61                      while (s->rchild)
62                      {
63                              q = s;
64                              s = s->rchild;
65                      }
66                      p->data = s->data;
67
68                      if (q != p)
69                      {
70                              q->rchild = p->lchild;
71                      }
72                      else
73                      {
74                              q->lchild = s->lchild;
75                      }
76                      free(s);
77              }
78              else
79              {
80                      if (*bst == p)    //只有一个根节点
81                      {
82                              free(*bst);
83                              *bst = NULL;
84                              return;
85                      }
86
87                      BSTNode* parent = FindParent(*bst, p);
88                      if (parent->lchild == p)
89                      {
90                              parent->lchild = NULL;
91                      }
92                      else
93                      {
94                              parent->rchild = NULL;
95                      }
96                      free(p);
97              }
98      }
```

# ④平衡二叉树（AVL）

2023年1月31日　　10:51

***手搓太麻烦，只给库函数了~**

```
#include <set>
multiset <string> avl;
avl.size();
avl.empty();
avl.find(val);//不存在则返回avl.end()
upper/lower_bound(val);//返回首个值>val的元素的迭代器/最后一个值<val的迭代器
avl.insert(val);
avl.erase(iter)/(begin,end)/val;
//删除iter/区间上迭代器所指的/值为val的元素，返回下一个元素的迭代器
avl.clear();
avl.sort(iter_left,iter_right,comp_func);//少用
for(i=avl.begin();i!=avl.end();i++){…}//遍历（少用）
```

# ⑤字典树

2023年1月31日　　12:06

①应用方向

已给一个由许多字符串构成的字典，之后再给某个字符串，判断其是否在字典内

②代码实现

```
int nex[100005][26],cnt=0;//nex即为Trie树
bool exist[100005]={0};//该结点结尾的字符串是否存在
void insert(char *s, int l)
{  // 插入字符串
    int p = 0;
    for (int i = 0; i < l; i++)
   {
      int c = s[i] - 'a';
      if (!nex[p][c]) nex[p][c] = ++cnt;  // 如果没有，就添加结点
      p = nex[p][c];
    }
    exist[p] = 1;
 }
bool find(char *s, int l) {  //查找字符串
    int p = 0;
    for (int i = 0; i < l; i++) {
              int c = s[i] - 'a';
              if (!nex[p][c]) return 0;
              p = nex[p][c];
    }
    return exist[p];
}
```

# ⑥二叉堆

2023年2月1日　　20:25

0应用方向

求最值，支持插入、删除（仅限于顶部最值）

①代码实现

**手搓太麻烦，只写了库函数~**

```
#include<algorithm>
#include<functional> //用来支持大顶堆
using namespace std;
int heap[Max];
make_heap(heap+1,heap+cur_len+1,[less/greater<int>()]);//less为大顶堆
      //不管是一开始就建好堆还是逐一插入（然后每次问最值）都用make_heap
      //当前最小/大值即为heap[0]
      //若要每次删除顶部最值，只需弹出后把顶部和尾部值交换后再make_heap即可
```

# *⑦线段树

2023年1月31日 23:36

①应用方向

*维护（快速修改）区间信息，快速获取信息（求和等）

*线段树完全替代树状数组功能且树状数组相较其不能修改维护，但线段树更复杂

**是区间树（红黑树的一种变式，属于红黑树）的一种特殊情况~

②代码实现

见Python~

# Python

2023年1月28日 22:28

# ①基本操作

2023年1月30日　　20:08

①构建

（Ⅰ）定义

1.库函数（为第三方库，机试用不了！！）

```
from binarytree import Node
root=Node(value) #Node已经全部初始化好了!
root.left=…
```

2.手搓

```
class TreeNode: #构建时用TreeNode(val)
    def init(self,val,left,right,parent):
     #下面初始化的参数也可以不写在这里面（甚至可以只有self）
            self.val = val
            self.left = None
            self.right = None
            self.parent = None #这个较不常见
```

（Ⅱ）输入用'#'表示叶子结点的层次遍历序列，构建二叉树

```
global lst=input(),index=0
length=len(lst)
def Build(self):
        if indexlength-1 or lst[index]'#': return None
        #理论上可以不用比较index和length-1
        root=TreeNode(lst[index])
        index+=1
        root.left=self.Build()
        root.right=self.Build()
        return root
```

②遍历

（Ⅰ）前中后序

1.库函数

```
from binarytree import Node
root.preorder/inorder/postorder #前中后序遍历
```

2.手搓（以中序为例）

```
def inorder(self,root):
    if root:
        self.inorder(root.left)
        self.traverse_path.append(root.val)
        self.inorder(root.right)
```

（Ⅱ）层序遍历（类似于BFS）

1.结果放在一个一维列表中：

```python
class Solution:
def levelOrder(self, root: TreeNode) -> List[int]:
    if not root: return
    result = []
    stack = [root]
    while stack:
        lay=[]
        for node in stack:
        result.append(node.val)
        if node.left: lay.append(node.left)
        if node.right: lay.append(node.right)
        stack = lay
        return result
```

2.结果逐层放在二维列表中：

*库函数：

```python
import binarytree
list=root.levelorder #二维列表
```

*手搓：

```python
class Solution:
      def levelOrder(self, root: TreeNode) -> List[List[int]]:
            if not root: return
            result = []
            stack = [root]
            while stack:
                  lay=[]
                  lay_value=[]
                  for node in stack: lay_value.append(node.val)
                  if node.left: lay.append(node.left)
                  if node.right: lay.append(node.right)
                  stack = lay
                  result.append(lay_value)
            return result
```

# ②基本问题

2022年2月5日　　9:24

0 `import binarytree`

①求树的高度

```
root.height（库函数）
```

```
 def maxDepth(self, root):
        if not root:  # 递归边界
            return 0
        else:
            l = 1 + self.maxDepth(root.left)  # 递归
            r = 1 + self.maxDepth(root.right)
            return max(l, r)
```

注：用C语言时，尽量将结构体指针传入函数参数（更快），即

```
void averge(struct stu *stus,int len)//stu是结构体的具体类型，stus是个结构体数组名
```

*②层序遍历（类似于BFS）（前中后序遍历见基本操作栏笔记）

```
list=root.levelorder #库函数
```

（Ⅰ）结果放在一个一维列表中：

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        if not root:
            return
        result = []
        stack = [root]
        while stack:
            lay=[]
            for node in stack:
                result.append(node.val)
                if node.left:
                    lay.append(node.left)
                if node.right:
                    lay.append(node.right)
            stack = lay
        return result
```

（Ⅱ）结果逐层放在二维列表中：

```
class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return
        result = []
        stack = [root]
```

```
        while stack:
            lay=[]
            lay_value=[]
            for node in stack:
                lay_value.append(node.val)
                if node.left:
                    lay.append(node.left)
                if node.right:
                    lay.append(node.right)
            stack = lay
            result.append(lay_value)
        return result
```

# ③二叉排序/搜索树（BST）

2022年2月4日　20:46

无库函数。

①常见操作

（0）构建：见C[++]…~

（Ⅰ）查找：（替换就比它多一步）

```
    def search(self, data):
        node=self.root #self是个实例对象，此处先将node初始化，使其指向根节点
        while node:
            if node.val==data:
                return node
            elif node.val>data:
                node=node.left
            else:
                node = node.right
        return []
```

（Ⅱ）插入：

*总体思路：

从根节点开始，依次比较要插入的数据与节点的大小关系。如果插入的数据比节点的大，并且节点的右子树为空，就将数据直接插到右子节点的位置；如果右子树不为空，就在递归遍历右子树，查找插入位置。如果要插入的数据比节点的小，并且节点的左子树为空，就将数据直接插到左子树的位置；如果左子树不为空，就再递归遍历左子树，查找插入位置。

```
def insert(self, data):
        if not self.root:
            self.root = TreeNode(data) #对象类型TreeNode需说明白
        else:
            node = self.root
            while node:
                p = node
                if data < node.val:
                    node = node.left
                else:
                    node = node.right
            TreeNode(data).parent = p #这步看情况要不要
            if data < p.val:
                p.left = nTreeNode(data)
            else:
                p.right = TreeNode(data)
        return True
```

（Ⅲ）删除：

*总体思路：分三种情况进行讨论：
1.要删除的节点没有子节点：直接将父结点中指向要删除节点的指针置为null。
2.要删除的节点只有一个子节点（只有左子节点或者右子节点）：更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点即可。
3.要删除的节点有两个子节点：首先找到要删除节点的右子树中的最小节点，把它的值替换到要删除的节点上。然后再删除掉这个最小节点即可。

```
    def _del(self, node):
        # 1
        if node.left is None and node.right is None:
            if node == self.root:
                self.root = None
            else:
                if node.val < node.parent.val:
                    node.parent.left = None
                else:
                    node.parent.right = None
                node.parent = None
        # 2
        elif node.left is None and node.right is not None:
            if node == self.root:
                self.root = node.right
                self.root.parent = None
                node.right = None
            else:
                if node.val < node.parent.val:
                    node.parent.left = node.right
                else:
                    node.parent.right = node.right
                node.right.parent = node.parent
                node.parent = None
                node.right = None
        elif node.left is not None and node.right is None:
            if node == self.root:
                self.root = node.left
                self.root.parent = None
                node.left = None
```

```
            else:
                if node.val < node.parent.val:
                    node.parent.left = node.left
                else:
                    node.parent.right = node.left
                node.left.parent = node.parent
                node.parent = None
                node.left = None
        # 3
        else:
            min_node = node.right
            while min_node.left:
                min_node = min_node.left
            if node.val != min_node.val:
                node.val = min_node.val
                self._del(min_node)
            else:
                self._del(min_node)#这两行没明白
                self._del(node)
```

# ④平衡二叉树（AVL）

2022年2月8日　12:28

*0库函数实现（有序列表，**但为第三方库，机试用不了！**）

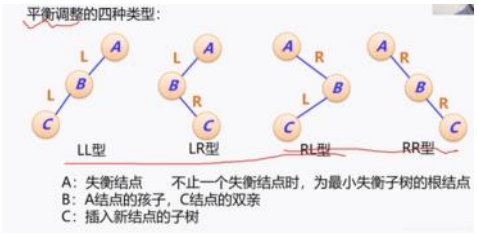
```
pip install sortedcontainers
```

```
from sortedcontainers import SortedList
lst=SortedList()
val=lst.index(index) #查找
lst.count(val)
lst.add(val)
lst.update(iter) #把一新容器的数都插入进去
lst.bisect_left/right(val)
      #插入，若存在相同元素则插入其左/右侧并返回索引值
val=lst.pop(index) #index不写的话默认为-1
lst.discard/remove(val) #discard不报错，remove抛异常
lst.clear()
```

①构建

（Ⅰ）旋转

*类型及对应方法简介




LL/RR:一次右/左旋转

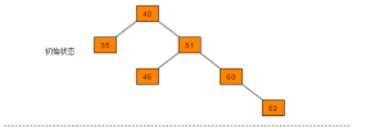
LR/RL:左旋转→右旋转/右旋转→左旋转

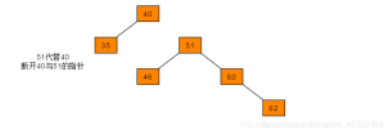
注：此处讲的仅是总体上不平衡的情况，细节上仍需加入一些可能的操作！

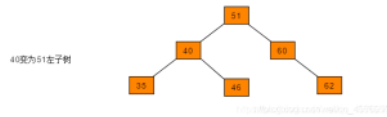
→总体上不平衡属于LL型

→此处的E也可以为空

**RR：

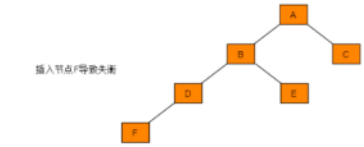




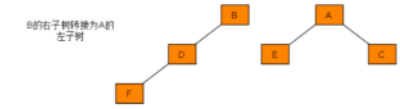


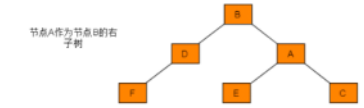
*加入细节操作后的示例（分类）

1.LL（右旋转）




注意E肯定比A小！

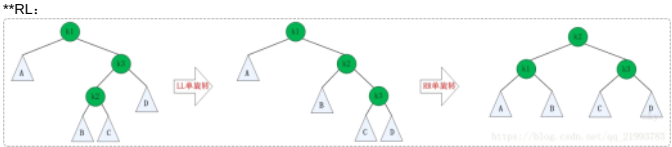




代码实现：

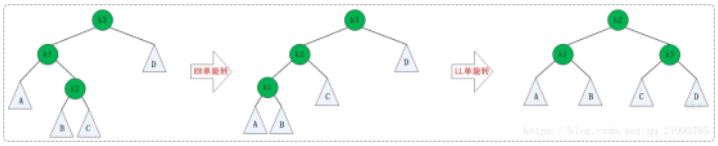
```
# LL右旋转
def right_rotate(node):
    if node is None:
        return
    # 创建新的结点，以当前根结点的值
    new_node = copy.deepcopy(node)
    # 把新结点的右子树设为当前结点的右子树
    new_node.right = node.right
    # 把新结点的左子树设为当前结点的左子树的右子树
    new_node.left = node.left.right
    # 把当前结点的值替换成它的左子结点的值
    node.val = node.left.val
    # 把当前结点的左子树设置成当前结点的左子树的左子树
    node.left = node.left.left
    # 把当前结点的右子结点设置成（指向）新的结点
    node.right = new_node
    return
```

**RL：



2.LR（左旋转→右旋转）

可以看成先左旋转化归成LL的情况，然后再右旋转一次！




代码实现：就是一次LL的代码加上一次RR的代码，略~

（Ⅱ）完整构造过程代码：

```
class AVLTree(object):
    def __init__(self):
        self.root = None
    def left_height(self, node):  # 开始传入根结点，后面传入每颗子树的根结点
        if node is None:
            return 0
        return self.tree_height(node.left)
    def right_height(self, node):  # 开始传入根结点，后面传入每颗子树的根结点
        if node is None:
            return 0
        return self.tree_height(node.right)
    def tree_height(self, node):
        if node is None:
            return 0
        return max(self.tree_height(node.left), self.tree_height(node.right)) +1
    def left_rotate(self, node):
        ... #略，见上
    def add(self, val):#插入节点，此处的插入函数不包括后续的调整（最好能包括）
        ... #略，见之前笔记（实在不行见CSDN收藏）
    def jude_node(self, node):  # 对AVL树进行必要的调整
        if self.right_height(node) - self.left_height(node) > 1:
            if node.right and self.left_height(node.right) >
               self.right_height(node.right):
                self.right_rotate(node.right)
                self.left_rotate(self.root)
            else:
                self.left_rotate(self.root)
        if self.left_height(node) - self.right_height(node) > 1:
            if node.left and self.right_height(node.left) >
                self.left_height(node.left):
                self.left_rotate(node.left)
                self.right_rotate(self.root)
            else:
                self.right_rotate(self.root)
        return
#接下来构造AVL树时就是每次插入新节点时调用一次add函数，再调用一次jude_node函数判断根节点是否需要调整（即node是root），代码略~（不应该每个节点都要调整一遍？）
```

```
def add(self, val):
        node = TreeNode(val)
        if self.root is None:
            self.root = node
            return
        queue = [self.root]
        while queue:
            temp_node = queue.pop(0)
            # 判断传入结点的值和当前子树结点的值关系
            if node.val < temp_node.val:
                if temp_node.left is None:
                    temp_node.left = node
                    return
                else:
                    queue.append(temp_node.left)
            if node.val >= temp_node.val:
                if temp_node.right is None:
                    temp_node.right = node
                    return
                else:
                    queue.append(temp_node.right)
```

*②AVL树的插入、删除（查找操作同二叉搜索树，略）

就是二叉搜索树中的插入、删除操作再加上一个jude_node函数判断根节点是否需要调整！（代码略~）

*③AVL树的判断方法

平衡因子=结点左子树的高度−结点右子树的高度（即叶子的平衡因子为0）

平衡二叉树上所有结点的平衡因子只能是-1、0、1。

# ⑤字典树（Trie）

2022年3月21日　　22:55

①应用方向

已给一个由许多字符串构成的字典，之后再给某个字符串，判断其是否在字典内

②代码实现

```
Trie={}
def insert(word):#不用另建一个Trie类，直接放在其他函数里面即可
    t=Trie
    for i in word:
        if i not in t:t[i]={}
        t=t[i]
    t['#']=True

def search(word):
    t=Trie
    for i in word:
        if i not in t:return False
        t=t[i]
    if '#' not in t:return False
    return True
```

# ⑥（二叉）堆

2022年2月8日　　22:32

```
import heapq #导入标准库中的heap库，heap指的是最小堆
```

**只用学库函数即可，手搓部分仅为留在这！**

①构建（最大堆）

（Ⅰ）库函数

```
heapq.heapify(list) #使数组转化为堆（若初始化heapq可以直接写heap=[]）
```

（Ⅱ）手搓

```
class MaxHeap(object):
    def __init__(self, maxsize=None):
        self.maxsize = maxsize
        self._elements = Array(maxsize)
        self._count = 0
```

②插入

（Ⅰ）库函数

```
    heapq.heappush(heap,item)
```

（Ⅱ）手搓

```
def add(self, value):
        if self._count >= self.maxsize:
            raise Exception("The heap is full!")
        self._elements[self._count] = value
        self._count += 1
        self._siftup(self._count-1)

    def _siftup(self, index):
        if index > 0:
            parent = int((index - 1) / 2)
            if self._elements[parent] > self._elements[index]:
                self._elements[parent], self._elements[index] = self._elements[index], self._elements[parent]
                self._siftup(parent)
```

③删除（堆顶元素）

（Ⅰ）第三方库

```
    heapq.heappop(heap)
```

```
    heapq.heapreplace(heap, item)#删除最小值并添加新值
```

（Ⅱ）手搓

```
    def extract(self):#删除堆顶元素并返回此元素值
        if self._count <= 0:
            raise Exception('The heap is empty!')
        value = self._elements[0]
        self._count -= 1
        self._elements[0] = self._elements[self._count]
        self._siftdown(0)
```

```
        return value

    def _siftdown(self, index):
        if index < self._count:
            left = 2 * index + 1
            right = 2 * index + 2
            if left < self._count and right < self._count \
                and self._elements[left] <= self._elements[right] \
                and self._elements[left] <= self._elements[index]:
                self._elements[left], self._elements[index] = self._elements[index], self._elements[left]
                self._siftdown(left)
            elif left < self._count and right < self._count \
                and self._elements[left] >= self._elements[right] \
                and self._elements[right] <= self._elements[index]:
                self._elements[right], self._elements[index] = self._elements[index], self._elements[right]
                self._siftdown(left)
            if left < self._count and right > self._count \
                and self._elements[left] <= self._elements[index]:
                self._elements[left], self._elements[index] = self._elements[index], self._elements[left]
               self._siftdown(left)
```

④其他功能（库函数）

*查堆中最大的n个数：

```
1  heapq.nlargest (n, heap)
```

# *⑦线段树

2022年2月12日　　18:10

一、应用方向

*维护（快速修改）区间信息，快速获取信息（求和等）

*线段树完全替代树状数组功能且树状数组相较其不能修改维护，但线段树更复杂

**是区间树（红黑树的一种变式，属于红黑树）的一种特殊情况~

二、代码实现

①线段树

树状的线段（区间），每个线段是类定义的self，val是满足某个目标（下面是区间上的最大值）的区间上的函数，l、r是线段的左右端点。而它们又用一个大小为4n的数组（n为总线段的长度）保存，类似于堆，下标之间存在乘除2加减一这样的数值关系。

（Ⅰ）定义

```
# 定义树节点，l,r, val表示该节点记录的是区间[l, r]的最大值是val
class Tree():
    def __init__(self):
        self.l = 0
        self.r = 0
        self.lazy = 0
        self.val = 0
```

（Ⅱ）构建

```
# 二叉树是堆形式，可以用一维数组存储，注意数组长度要开4倍空间
tree = [Tree() for i in range(10*4)]
# 建树，用cur<<1访问左子树，cur<<1|1访问右子树，位运算操作很方便
def build(cur, l, r):
    tree[cur].l, tree[cur].r, tree[cur].lazy, tree[cur].val = l, r, 0, 0
    # 当l==r的时候结束递归
    if l < r:
        mid = l + r >> 1
        build(cur<<1, l, mid)
        build(cur<<1|1, mid+1, r)

```

（Ⅲ）更新

```
# 当子节点计算完成后，用子节点的值来更新自己的值
def pushup(cur):
    tree[cur].val = max(tree[cur<<1].val, tree[cur<<1|1].val)
```

1.单点更新

```
def add(cur, x, v):
    if tree[cur].l == tree[cur].r:
        tree[cur].val += v
    else:
        mid = tree[cur].r + tree[cur].l >> 1
        if x > mid:
            add(cur>>1|1, x, v)
        else:
            add(cur<<1, x, v)
        pushup(cur)
```

2.区间更新

```
  # 将lazy标记向下传递一层
  def pushdown(cur):
      if tree[cur].lazy:
          lazy = tree[cur].lazy
          tree[cur<<1].lazy += lazy
          tree[cur<<1|1].lazy += lazy
          tree[cur<<1].val += lazy
          tree[cur<<1|1].val += lazy
          tree[cur].lazy = 0

  # 区间更新
  def update(cur, l, r, v):
      if l <= tree[cur].l and tree[cur].r <= r:
          tree[cur].lazy += v
          tree[cur].val += v
          return
      if r < tree[cur].l or l > tree[cur].r:
          return
      if tree[cur].lazy:
          pushdown(cur)#为什么要先pushdown再pushup？
      update(cur<<1, l, r, v)
      update(cur<<1|1, l, r, v)
      pushup(cur)
```

（Ⅳ）区间查询

```
def query(cur, l, r):
    if l <= tree[cur].l and tree[cur].r <= r:
        return tree[cur].val
    if tree[cur].l > r or tree[cur].r < l:
        return 0
    if tree[cur].lazy:
        pushdown(cur)
    return max(query(cur<<1, l, r), query(cur<<1|1))
```

②树状数组

见Java~

# **⑧红黑树（查找操作略）

2022年2月5日　　18:02

无库函数。

①基本特征

红黑树是一棵二叉树， 有五大特征：

特征一： 节点要么是红色，要么是黑色（红黑树名字由来）。

特征二： 根节点是黑色的

特征三： 每个叶节点(nil或空节点)是黑色的。

特征四： 每个红色节点的两个子节点都是黑色的（相连的两个节点不能都是红色的）。

特征五： 从任一个节点到其每个叶子节点的所有路径都是包含相同数量的黑色节点。

从五大特征直观上总结出来几个点：

1 对每个红色节点，子节点只有两种情况：要么都没有，要么都是黑色的。（不然会违反特征四）

2 对黑色节点，如果只有一个子节点，那么这个子节点，必定是红色节点。（不然会违反特征五）

3 假设从根节点到叶子节点中，黑色节点的个数是h, 那么树的高度H范围 h<= H <= 2H（特征四五决定）。

②与平衡二叉树的区别与联系

*时间复杂度都为$O(\log n)$

*红黑树放弃了追求完全平衡，只追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单，性能更优，而平衡二叉树追求绝对平衡，条件比较苛刻，每次插入新节点之后需要旋转的次数不能预知。

③构建：就是一次次的插入，插入代码见下，略~

*④插入

就是AVL树的插入（+调整步骤）+重新着色与再调整，前两步见AVL树笔记，着色与再调整思路和代码见下。

（Ⅰ）思路：

各种情况的核心思路都是：将红色的节点移到根节点；然后，将根节点设为黑色

**（Ⅱ）具体情况：

根据被插入节点的父节点的情况，可以将"当节点z被着色为红色节点，并插入二叉树"划分为三种情况来处理。
① 情况说明：被插入的节点是根节点。
　处理方法：直接把此节点涂为黑色。
② 情况说明：被插入的节点的父节点是黑色。
　处理方法：什么也不需要做。节点被插入后，仍然是红黑树。
③ 情况说明：被插入的节点的父节点是红色。
　处理方法：那么，该情况与红黑树的"特性(5)"相冲突。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据"叔叔节点的情况"，将这种情况进一步划分为3种情况(Case)。

| | 现象说明 | 处理策略 |
|---|---|---|
| Case 1 | 当前节点的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。 | (01) 将"父节点"设为黑色。<br>(02) 将"叔叔节点"设为黑色。<br>(03) 将"祖父节点"设为"红色"。<br>(04) 将"祖父节点"设为"当前节点"(红色节点)；即，之后继续对"当前节点"进行操作。 |
| Case 2 | 当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子 | (01) 将"父节点"作为"新的当前节点"。<br>(02) 以"新的当前节点"为支点进行左旋。 |
| Case 3 | 当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子 | (01) 将"父节点"设为"黑色"。<br>(02) 将"祖父节点"设为"红色"。<br>(03) 以"祖父节点"为支点进行右旋。 |

**（Ⅲ）代码实现：

```
def RBInsert( T, z):
    y = T.nil
    x = T.root
    while x != T.nil:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if y == T.nil:
        T.root = z
    elif z.key < y.key:
        y.left = z
```

```
    else:
        y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = 'red'
    RBInsertFixup(T, z)
    return z.key, '颜色为', z.color

def RBInsertFixup( T, z):#红黑树的上色
    while z.parent.color == 'red':
        if z.parent == z.parent.parent.left:
            y = z.parent.parent.right
            if y.color == 'red':
                z.parent.color = 'black'
                y.color = 'black'
                z.parent.parent.color = 'red'
                z = z.parent.parent
            else:
                if z == z.parent.right:
                    z = z.parent
                    LeftRotate(T, z)
                z.parent.color = 'black'
                z.parent.parent.color = 'red'
                RightRotate(T,z.parent.parent)
        else:
            y = z.parent.parent.left
            if y.color == 'red':
                z.parent.color = 'black'
                y.color = 'black'
                z.parent.parent.color = 'red'
                z = z.parent.parent
            else:
                if z == z.parent.left:
                    z = z.parent
                    RightRotate(T, z)
                z.parent.color = 'black'
                z.parent.parent.color = 'red'
                LeftRotate(T, z.parent.parent)
    T.root.color = 'black'
```

**⑤删除

就是AVL树的插入（+调整步骤）+重新着色与再调整，前两步见AVL树笔记，着色与再调整思路和代码见下：

（Ⅰ）思路

> 前面我们将"删除红黑树中的节点"大致分为两步，在第一步中"将红黑树当作一颗二叉查找树，将节点删除"后，可能违反"特性(2)、(4)、(5)"三个特性。第二步需要解决上面的三个问题，进而保持红黑树的全部特性。
> 为了便于分析，我们假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。为什么呢?
> 通过RB-DELETE算法，我们知道：删除节点y之后，x占据了原来节点y的位置。既然删除y(y是黑色)，意味着减少一个黑色节点；那么，再在该位置上增加一个黑色即可。这样，当我们假设"x包含一个额外的黑色"，就正好弥补了"删除y所丢失的黑色节点"，也就不会违反"特性(5)"。因此，假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。
> 现在，x不仅包含它原本的颜色属性，x还包含一个额外的黑色。即x的颜色属性是"红+黑"或"黑+黑"，它违反了"特性(1)"。
>
> 现在，我们面临的问题，由解决"违反了特性(2)、(4)、(5)三个特性"转换成了"解决违反特性(1)、(2)、(4)三个特性"。RB-DELETE-FIXUP需要做的就是通过算法恢复红黑树的特性(1)、(2)、(4)。RB-DELETE-FIXUP的思想是：将x所包含的额外的黑色不断沿树上移(向根方向移动)，直到出现下面的姿态：
> a) x指向一个"红+黑"节点。此时，将x设为一个"黑"节点即可。
> b) x指向根。此时，将x设为一个"黑"节点即可。
> c) 非前面两种姿态。
>
> 将上面的姿态，可以概括为3种情况。
> ① 情况说明：x是"红+黑"节点。
> 处理方法：直接把x设为黑色，结束。此时红黑树性质全部恢复。
> ② 情况说明：x是"黑+黑"节点，且x是根。
> 处理方法：什么都不做，结束。此时红黑树性质全部恢复。
> ③ 情况说明：x是"黑+黑"节点，且x不是根。
> 处理方法：这种情况又可以划分为4种子情况。这4种子情况如下表所示：

| | 现象说明 | 处理策略 |
| --- | --- | --- |
| Case 1 | x是"黑+黑"节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。 | (01) 将x的兄弟节点设为"黑色"。<br>(02) 将x的父节点设为"红色"。<br>(03) 对x的父节点进行左旋。<br>(04) 左旋后，重新设置x的兄弟节点。 |
| Case 2 | x是"黑+黑"节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。 | (01) 将x的兄弟节点设为"红色"。<br>(02) 设置"x的父节点"为"新的x节点"。 |
| Case 3 | x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。 (其实这里要区分，x是父节点的左子树还是右子树，x是左子树则按这里的规则处理；若x是右子树，则左变右，右变左) | (01) 将x兄弟节点的左孩子设为"黑色"。<br>(02) 将x兄弟节点设为"红色"。<br>(03) 对x的兄弟节点进行右旋。<br>(04) 右旋后，重新设置x的兄弟节点。 |
| Case 4 | x是"黑+黑"节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色。 (同上) | (01) 将x父节点颜色 赋值给 x的兄弟节点。<br>(02) 将x父节点设为"黑色"。<br>(03) 将x兄弟节点的右子节设为"黑色"。<br>(04) 对x的父节点进行左旋。<br>(05) 设置"x"为"根节点"。 |

（Ⅱ）代码实现：

```
def RBDelete(T, z):
    y = z
    y_original_color = y.color
    if z.left == T.nil:
        x = z.right
        RBTransplant(T, z, z.right)
    elif z.right == T.nil:
        x = z.left
        RBTransplant(T, z, z.left)
    else:
        y = TreeMinimum(z.right)
        y_original_color = y.color
        x = y.right
        if y.parent == z:
            x.parent = y
        else:
            RBTransplant(T, y, y.right)
            y.right = z.right
            y.right.parent = y
        RBTransplant(T, z, y)
        y.left = z.left
        y.left.parent = y
        y.color = z.color
    if y_original_color == 'black':
        RBDeleteFixup(T, x)

def RBDeleteFixup( T, x):#上色
    while x != T.root and x.color == 'black':
        if x == x.parent.left:
            w = x.parent.right
            if w.color == 'red':
                w.color = 'black'
                x.parent.color = 'red'
                LeftRotate(T, x.parent)
                w = x.parent.right
            if w.left.color == 'black' and w.right.color == 'black':
                w.color = 'red'
                x = x.parent
            else:
                if w.right.color == 'black':
                    w.left.color = 'black'
```

```
                    w.color = 'red'
                    RightRotate(T, w)
                    w = x.parent.right
                w.color = x.parent.color
                x.parent.color = 'black'
                w.right.color = 'black'
                LeftRotate(T, x.parent)
                x = T.root
        else:
            w = x.parent.left
            if w.color == 'red':
                w.color = 'black'
                x.parent.color = 'red'
                RightRotate(T, x.parent)
                w = x.parent.left
            if w.right.color == 'black' and w.left.color == 'black':
                w.color = 'red'
                x = x.parent
            else:
                if w.left.color == 'black':
                    w.right.color = 'black'
                    w.color = 'red'
                    LeftRotate(T, w)
                    w = x.parent.left
                w.color = x.parent.color
                x.parent.color = 'black'
                w.left.color = 'black'
                RightRotate(T, x.parent)
                x = T.root
    x.color = 'black'
```

# Java

2023年1月30日 16:50

# ①-⑦除了④

2023年1月31日 15:56

①-③、⑤：见C[++]…~

⑥⑦：见Python~

# ④平衡二叉树（AVL）

2023年2月1日　　18:10

```
import java.util.TreeMap;
avl.empty();
avl.size();
avl.get(); //若找不到返回null
avl.put(key[,value]); //插入
avl.remove(key[,value]);
avl.clear();
avl.entrySet()/keySet()/values(); //键值对/键/值的遍历
```

# C[++]&Java

2023年1月28日 23:00

# 0有/无向图有无环的判定

2023年1月30日　　21:42

①有向图

（Ⅰ）拓扑排序

1. 创建一个队列q，将入度为0的顶点全部加入队列。
2. 当队列不为空时，取出首结点，[访问输出]，遍历其所有后继顶点，另后继结点的入度减一，判断若入度减一后为0，则将这后继顶点加入队列。再清除首结点从它出发的边，即后继结点。
3. 步骤2结束意味着队列为空。判断加入拓扑序列的顶点数和图的顶点数n是否相等，若相等则证明拓扑排序成功，和该图是有向无环图。

```
#include<iostream>//为了引入string型
#include<string.h>
#include<unordered_map>
unordered_map<string,int>indegrees;//用字典来记录入度
string topoSort(string *graph,int l) //graph的元素是长度为2的字符数组
{
        int i,l1=0,l2=0;//l1，l2分别为ans，q的长度，ans为拓扑排序后的结果
        string ans[N],q[N];
        for(i=1;i<=l;i++) indegrees[graph[i][0]]=0; //初始化入度
        for(i=1;i<=l;i++) indegrees[graph[i][1]]+=1;
        for(pair<string,int> node:indegrees)
        {
                if(!indegrees.second) q[++l2]=indegrees.first);
        }
        while(q)
        {
                strcpy(t,q[l2--]);
                ans[++l1]=t;
                for(i=1;i<=l;i++)
                {
                        if(!strcpy(graph[i][0],t))
                        {
                                indegrees[grpah[i][1]]-=1;
                                if(!indegrees[graph[i][1]])
                                {
                                        q.push(graph[i][1]);
                                        break;
                                }
                        }
                }
                if(ans.size()==num) //输出的顶点是否与图中的顶点数相等
                        return false; //无环
                return true; //有环
}
```

（Ⅱ）DFS

```
// 因为是有向图两个顶点也可以成环
void dfs(Graph g, int i)
{
        int j;
        color[i] = gray; //灰色 表示正在访问
        printf("%c ", g.vex[i]);
        for (j = 0; j < g.vex_num; j++) {
                if (i != j && g.edge[i][j] != INFINITY) {//两顶点有边相连
                        if (color[j] == white) {
                                dfs(g, j);//如果该节点未访问 继续访问其临近节点
                        }
                        else if(color[j] == gray){
                                loop_num++;
                                is_dag = false; //有环
                        }
                }
        }
        color[i] = black;
}

void dfs_trvsal(Graph g)
{
        int i;
        for (i = 0; i < g.vex_num; i++) {
                color[i] = white;
                loop_num = 0;
        }
        link_component = 0;
        for (i = 0; i < g.vex_num; i++) {
                if (color[i] == white) {
                        link_component++;
                        dfs(g, i);
                }
        }
}
```

②无向图

（Ⅰ）并查集

首先我们把每个点看成独立的集合{0}，{1}，{2}，然后规定如果两个点之间有边相连，如果这两个点不属于同一个集合，那就将他们所属的结合合并，看边0-1，直接将这两个点代表的集合合并{0，1}，其中让1来当父节点，看边1-2，它们分别属于不同的集合，合并集合之后是{1，2}，让2来当父节点，依照这种逻辑关系，0的祖先节点就是2，然后在看边0-2，他们属于一个集合，因为他们有着共同的祖先2，这就说明0-2之间在没有0-2这条边之前已经连通了，如果在加上这条边的话那从0到2就有两条路径可达，就说明存在一个环了.

```
int main(){
   int n,m,f1,f2;
   int pre[1000];
       //每个点互相独立，自成一个集合，从1编号到1000的上级都是自己
   for(i=1;i<=1000;++i){
       pre[i]=i;
   }
   while(scanf("%d,%d",&m,&n)==2){ //m和n相连
       f1=find(n);
       f2=find(m);
       if(f2==f1)
          //两点已经连通了，那么这条路有一个环
       else
           unio(n,m);
    }
    return 0;
}
```

（Ⅱ）DFS

```
bool dfs(int i,int pre)</span>{
    visit[i]=true;
    for(int j=1;j<=v;j++)
        if(g[i][j])
        {
            if(!visit[j])
                return dfs(j,i);
            else if(j!=pre)  //如果访问过，且不是其父节点，那么就构成环
                return true;
        }
}
```

# ①拓扑排序

2022年4月27日　　20:28

①原问题

```
#include<iostream>//为了引入string型
#include<string.h>
#include<unordered_map>
unordered_map<string,int>indegrees;//用字典来记录入度
string topoSort(string *graph,int l) //graph的元素是长度为2的字符数组
{
        int i,l1=0,l2=0;//l1，l2分别为ans，q的长度
        string ans[N],q[N];
        for(i=1;i<=l;i++) indegrees[graph[i][0]]=0; //初始化入度
        for(i=1;i<=l;i++) indegrees[graph[i][1]]+=1;
        for(pair<string,int> node:indegrees)
        {
                if(!indegrees.second) q[++l2]=indegrees.first);
        }
        while(q)
        {
                strcpy(t,q[l2--]);
                ans[++l1]=t;
                for(i=1;i<=l;i++)
                {
                        if(!strcpy(graph[i][0],t))
                        {
                                indegrees[grpah[i][1]]-=1;
                                if(!indegrees[graph[i][1]])
                                {
                                        q.push(graph[i][1]);
                                        break;
                                }
                        }
                }
                if(ans.size()==num) //输出的顶点是否与图中的顶点数相等
                        return ans;
                return Null;
}
```

②推广

*思想很重要！

# ②单源最短路径（仅限有向图！）

2022年4月28日　　23:45

①Dijkstra

（Ⅰ）应用方向：

主要针对的是有向图的单元最短路径问题，且不能出现权值为负的情况

（Ⅱ）总体思想：

假设存在G=<V,E>，源顶点为V0，S={V0},distance[i]记录V0到i的最短距离，matrix[i][j]记录从i到j的边的权值，即两点之间的距离。
1）从V-S中选择使dist[i]值最小的顶点i，将i加入到U中；
2）更新与i直接相邻顶点的dist值。dist[j]=min(dist[j],dist[i]+matrix[i][j])
3）直到S=V，所有顶点都包含进来了，算法停止。
（Ⅲ）算法实现：
*先将dis和book初始化为无穷大~

```
dis[t]=0;//t为目标节点
for(i=1;i<=n-1;i++)
{
        //找到离1号最近的顶点
        min=inf;
        for(j=1;j<=n;j++)
        {
                if(book[j]==0&&dis[j]<min)
                {
                        min=dis[j];
                        u=j;
                }
         }
        book[u]=1;
        for(v=1;v<=n;v++)
        {
                if(e[u][v]<inf)
                {
                        if(dis[v]>dis[u]+e[u][v]) dis[v]=dis[u]+e[u][v];
                 }
        }
}
```

②Bellman ford

（Ⅰ)应用方向：

有向图，且允许出现负权重~

（Ⅱ）总体思想：

首先初始化，对所有的节点V来说，所有的边E进行松弛操作，再然后循环遍历每条边，如果d[v] > d[u] + w(u,v),表示有一个负权重的环路存在。最后如果没有负权重环路，那么d[v] 是最小路径值。

（Ⅲ）代码实现：（graph即保存节点与边信息的二维数组）

```
def bellman_ford(graph, source):
    dist = {}
    p = {}
    max = 10000
    for v in graph:
        dist[v] = max  #赋值为负无穷完成初始化
        p[v] = None
    dist[source] = 0

    for i in range(len( graph ) - 1):
        for u in graph:
            for v in graph[u]:
                if dist[v] > graph[u][v] + dist[u]:
                    dist[v] = graph[u][v] + dist[u]
                    p[v] = u     #完成松弛操作，p为前驱节点

    for u in graph:
        for v in graph[u]:
            if dist[v] > dist[u] + graph[u][v]:
                return None, None  #判断是否存在环路

    return dist, p
```

（Ⅳ）特点：复杂度略高（二次方）

③SPFA算法

（Ⅰ）应用方向：

有向图，且允许出现负权重~

（Ⅱ）总体思想：

每次选取队首顶点u的所有边进行松弛操作，假设有一条u到v的边，如果通过这条边使得源点到顶点v的最短路程变短，且顶点v不在当前队列中，就将顶点v放入队尾。

（Ⅲ）代码实现：（此处为Python代码）

```
from collections import deque
q=deque()
q.append/extend(…)
book=[]
def spfa():
      while not q:
            t=q.popleft()
            for i in graph[t]: #t的所有邻边
                  if ans[u]+edge[i][j]<ans[v]:
                        q.append(v)
                        if book[v]>n-1: #有负环
                              return
```

# ★ ③深度优先搜索（DFS）

2022年4月27日　　19:50

（Ⅰ）应用方向：

*几乎可以应用于任何问题！

*适用于找出所有解的问题

*在深度较大时效率较低

（Ⅱ）总体思想：

*就是枚举

*每次都把路先走到最深，实在走不下去了再一步步逐一回溯搜索，用递归函数或栈来实现

*同样是深搜，也要尽量寻找简单的思路！（如果涉及了太多条件则也要思考有无简单办法）

（Ⅲ）代码实现：

大致代码：（python）

（C语言dfs就用void型）

```
def dfs(x,y):#参数在问题为一维时也可以是step
    if not 0<=x<m or 0<=y<n or book[x][y] or … :#已经走到尽头了
    (也有可能是step==n等等)
        …#执行走到尽头时该干的事（检验是否满足条件然后count++等
        等）
        return
    book[x][y]=1 #标记这路走过了
    … #此时还需要执行的操作
    dfs(x-1,y) #开始遍历所有的路
    dfs(x,y+1)
    dfs(x+1,y)
    dfs(x,y-1)
    #有的题目还需要book[x][y]=0来把走过的路收回！
#写主函数
for i … :
    for j … :
        if book[i][j]==0 and … :
            dfs(i, j)
```

★ PS：本来为取值为0，1的二维题可以省去book！

2.具体示例：

（对应问题： 求出123的全排列，即123　132　213　231　312　321）

```c
#include <stdio.h>
#include <stdlib.h>
//在全局变量中,数组默认为0
int a[5], book[5], n = 3;
void Dfs(int step)
{
	int i;
	//当数组a存放了1-3所有元素且每个数都是被标记过的,证明该序列满足
	if (step == n + 1){
		//执行路走到尽头时该做的操作
		for (i = 1; i <= n; i++)
			printf("%d", a[i]);
		printf("\n");
		return;
	}
	for (i = 1; i <= n; i++){
		//判断该位置是否被标记过
		if (book[i] == 0){
			//再往深里走一步时要执行的操作，此处即把i存入到数组的第step的位置中
			a[step] = i;
			//标记i位置证明i已经存入到a数组当中
			book[i] = 1;
			//更新step继续执行循环
			Dfs(step + 1);
			//将刚标记的位置收回,执行下一次尝试，这是非常重要且容易忘的一步！！！
			book[i] = 0;
		}
	}
	return;
}

int main()
{
	//首先从第一个数字开始
	Dfs(1);
	system("pause");
	return 0;

}
```

# ★ ④广度优先搜索（BFS）

2022年4月27日　　19:51

（Ⅰ）应用方向：

适用于求解最值问题（因为它搜索到的解一定是路径最短的解）

（Ⅱ）总体思想：

*就是枚举

*每次都把能走的路都试了，不再回溯

*找到了即停止搜索

（Ⅲ）代码实现：

*用队列（若用C语言则不用queue库函数，而用底层模拟的办法，即把队列中数值对形式的元素以二维数组的形式记录）

1.大致代码：（python）

PS：popleft()函数需要额外导入库：

```
from collections import deque
        queue=deque()
        queue.append((x,y))
```

```
for i … : #实际上属于"一main到底",此处看情况需不需要对queue进行初始的元素加入(如果这样一般无需再双重循环遍历)
    for j … :
        if book[i][j]==0 and … :
            queue.append([x,y])#queue不是一般的列表，而是双向队列deque！
            book[i][j]=1
            cnt+=1 #记录广搜的总次数（看题意需不需要记录，记得cnt=0初始化！）
            while queue:
                x,y=queue.popleft()#注意这里是popleft而非pop！
                for nx,ny in [[x-1,y],…]:
                    if book[nx][ny]==0 and … :
                        book[nx][ny]=1
                        queue.append([nx,ny])
```

★ PS：本来为取值为0，1的二维题可以省去book！

*还有一种广搜的代码类型（适合求解最优解时用）：

```
queue.append(n,0) #n为最终目标，0为目前步数
while queue:
      tm,step=queue.popleft()
      for i … : #遍历所有可能路径
            if … : return step+1 #已经满足目标则返回
            elif … and not book[…]: #仍没达目标但满足题目条件时，若未标记则继续加入队列
                  queue.append(tm-…,step+1)
                  book[…]=1
return -1 #全部遍历完仍未搜索到则返回-1
```

★ 若为要求记录走过的路径：不再用队列而是用栈！！！（PS：此时一般用DFS顺手一点，除非题目特意要求出路径最短的解

```
int col_boundary,row_boundary,n,map[100][100],i,j,start_x,start_y,end_x,end_y,nend_xt[4][2]={{-1,0},{0,1},{1,0},{0,-1}},ans[10000][2]={0},len_ans=0,len_stack=0,stack[10000][2];
stack[len_stack][0]=start_x,stack[len_stack++][1]=start_y;
while(len_stack)
{
    int temp_x=stack[len_stack-1][0],temp_y=stack[len_stack--][1];
    ans[len_ans][0]=temp_x,ans[len_ans++][1]=temp_y;
    if(temp_x==end_x && temp_y==end_y)
    {
        for(i=0;i<len_ans;i++)
            printf("(%d,%d)->",ans[i][0],ans[i][1]);
        printf("\b\b");
        break;
    }
    for(i=0;i<4;i++)
    {
        if(1<=temp_x+nend_xt[i][0] && temp_x+nend_xt[i][0]<=m && temp_y+nend_xt[i][1]>=1 && temp_y+nend_xt[i][1]<=n && !map[temp_x+nend_xt[i][0]][temp_y+nend_xt[i][1]])
        {
            map[temp_x+nend_xt[i][0]][temp_y+nend_xt[i][1]]=1;
            stack[len_stack][0]=temp_x+nend_xt[i][0];
            stack[len_stack++][1]=temp_y+nend_xt[i][1];
        }
    }
    while(len_ans)
    {
        int flag=0;
        for(j=0;j<len_stack;j++)
        {
            if(stack[j]==ans[len_ans-1])
            {
                flag=1;
                break;
            }
        }
        if(!flag) len_ans--;
        else break;
    }
    if(!len_ans)
    {
        printf("Not found.");
        return 0;
    }
}
```

2.具体示例：

（啊哈算法里的拯救小美）

```
#include<iostream>
#include<algorithm>
#include<stack>
#include<set>
#include<queue>
#include<map>
#include<string>
#include<cstring>
#include<vector>
#include<ctype.h>
#include<sstream>
#include<unordered_map>
using namespace std;
typedef long long ll;
```

```cpp
typedef unsigned long long ull;
struct node
{
    int x,y;
    int s;
    int f;//父亲在队列中的编号。？？？干啥用的呢？有点像课本上的那个呀。
};
node que[2501];
int head,tail;
//mmp数组表示地图
int mmp[51][51];
//Book数组用来记录哪些点已经在队列中，防止一个点被重复扩展.....为啥教材上没有呢？
int Book[51][51];
//定义一个用于表示走的方向的数组
int Next[4][2]={{0,1},{1,0},{0,-1},{-1,0}};

int main()
{
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
        cin>>mmp[i][j];
    int startx,starty,p,q;
    cin>>startx>>starty>>p>>q;
    head=tail=1;
    que[tail].x=startx;
    que[tail].y=starty;
    que[tail].f=0;
    que[tail].s=0;
    tail++;
    Book[startx][starty]=1;
    int flag=0;
    while(head<tail)
    {
        int tx,ty;
        for(int i=0;i<3;i++)
        {
            tx=que[head].x+Next[i][0];
            ty=que[head].y+Next[i][1];
            if(tx<1||ty<1||tx>n||ty>m) continue;
            if(Book[tx][ty]==0&&mmp[tx][ty]==0)
            {
                que[tail].x=tx;
                que[tail].y=ty;
                tail++;
                Book[tx][ty]=1;
                mmp[tx][ty]=1;
            }
        }
        if(que[head].x==p&&que[head].y==q)
        {
            flag=1;
            break;
        }
        head++;
    }
    if(flag==1) cout<<que[head].x<<" "<<que[head].y<<endl;
    return 0;

}

/*
5 4
0 0 1 0
0 0 0 0
0 0 1 0
0 1 0 0
0 0 0 1
1 1 4 3
*/
```

## ★ ⑤最小生成树（MST）

2022年8月29日　16:06

①Prim算法

（Ⅰ）算法思想

*以点为操作对象

1. 设置一个顶点集合S和一个边集合TE，S和TE的初始状态皆为空集。
2. 选定图中的任意顶点K，从K开始生成最小生成树。(K加入集合S)。
3. 重复下列操作，直到选取了n-1条边为止。
   (1)选取一条权最小的边(X，Y)，要求是X要是集合S的元素，Y不是集合S的元素。
   (2)将顶点加入到集合S中，将边（X，Y）加入集合TE中。
4. 得到最小生成树T，T=（S，TE）。

（Ⅱ）代码实现

```
/* Prim算法生成最小生成树 */
void MiniSpanTree_Prim(MGraph G){
	int min, i, j, k;
	int adjvex[MAXVEX];	//保存相关顶点下标
	int lowcost[MAXVEX];	//保存相关顶点间边的权值
	lowcast[0] = 0;	//初始化第一个权值为0，即v0加入生成树，lowcost的值为0，在这里就是此下标的顶点已经加入生成树
	adjvex[0] = 0;	//初始化第一个顶点下标为0
	for(i = 1;i<G.numVertexes;i++){	//循环除下标为0外的全部顶点
		lowcost[i] = G.arc[0][i];		//将v0顶点与之有边的权值存入数组
		adjvex[i] = 0;	//初始化都为v0的下标
	}
	for(i=1;i<G.numVertexes;i++){
		min = INFINITY;	//初始化最小权值为无穷大，通常设置为很大的数字
		j = 1;
		k = 0;
		while(j<G.numVertexes){	//循环全部顶点
			if(lowcost[j] != 0 && lowcost[j] < min){ //如果权值不为0且权值小于min
				min = lowcost[j];	//让当前权值成为最小值
				k = j;		//将当前最小值的下标存入k
			}
			j++;
		}
		printf("(%d,%d)",adjvex[k],k);	//打印当前顶点边中权值最小边
		lowcost[k] = 0;	//将当前顶点的权值设置为0，表示此顶点已经完成任务
		for(j=1;j<G.numVertexes;j++){ //循环所有顶点
			if(lowcost[j] != 0 && G.arc[k][j] < lowcost[j]){	//若下标为k顶点各边权值小于此前这些顶点未被加入生成树权值
				lowcost[j] = G.arc[k][j];		//将较小权值存入lowcost
				adjvex[j] = k;	//将下标为k的顶点存入adjvex
			}
		}
	}
}
```

②Kruskal算法

（Ⅰ）算法思想

*以边为核心对象

1. 设最小生成树为T，T=（S，TE），TE初始状态为空集。
2. 将图中的边权从小到大依次排序。
3. 选取权最小的边，若这条边没有使T构成回路，就将这条边加入TE中（T保留了这条边）；若构成回路，则舍弃，不能加入TE中。
4. 再选取最小边，重复执行第三步，直到TE中包含n-1条边为止，最后的T即为最小生成树。

（Ⅱ）代码实现

```
/* 对边集数组Edge结构的定义 */
```

```c
typedef struct{
	int begin;
	int end;
	int weight;
}Edge;

/* kruskal算法生成最小生成树 */
void MiniSpanTree(MGraph G){  //生成最小生成树
	int i, n, m;
	Edge edges[MAXEDGE];    //定义边集数组
	int parent[MAXVEX];     //定义一数组用来判断边与边是否形成环路
	/* 此处省略将邻接矩阵G转化为边集数组edges并按权由小到大排序的代码 */
	for(i = 0;i<G.numVertexes;i++)
		parent[i] = 0;    //初始化数组值为0
	for(i = 0;i<G.numEdges;i++){  //循环每一条边
		n = Find(parent, edges[i].begin);
		m = Find(parent, edges[i].end);
		if(n != m){       //假如n与m不等，说明此边没有与现有生成树形成环路
			parent[n] = m;    //将此边的结尾顶点放入下标为起点的parent中，表示此顶点已经在生成树集合中
			printf("(%d,%d) %d", edges[i].begin, edges[i].end, edges[i].weight);
		}
	}
}

int Find(int* parent, int f){ //查找连线顶点的尾部下标
	while(parent[f] > 0)
		f = parent[f];
	return f;
}
```

# ⑥多源最短路径

2022年4月28日　　23:52

Floyd算法：

```
for(i=1;i<=n;i++)
{
	for(j=1;j<=n;j++)
	{
		for(k=1;k<=n;k++)
		{
			if(e[i][j]>e[i][k]+e[k][j]) e[i][j]=e[i][k]+e[k]
[j];
		}
	}
}
```

# ⑦计算几何学

2022年2月13日　　22:16

①线段相交、三点共线

判断各自线段的两个端点是否在对方线段的两侧（用反映顺逆时针顺序的叉积(p1-p0)X(p2-p0)=(x1-x0)(y2-y0)-(x2-x0)(y1-y0)来判断（叉积为零即三点共线，不要再按横坐标是否相等对直线解析式分类讨论了！）

②确定任意一对线段是否相交

需要扫除技术（算法）和红黑树，略~

③寻找凸包

Graham扫描法：通过设置一个候选点的堆栈S来解决凸包问题。输入集合Q中的每个点都被压入栈一次，非凸包CH(Q)中顶点的点被弹出堆栈，算法终止时，堆栈S中仅包含CH(Q)中的顶点，其顺序为各点在边界上出现的逆时针方向排列的顺序。
具体伪码和案例及其正确定证明不描述，主要说下算法过程：首先选择参照点p0，一般选择顶点（最下边或最右边这种）；其次将所有点按照和p0极角大小递增顺序压入堆栈；接着依次弹出右转的非凸包顶点，最后剩下就是凸包的顶点。

④寻找最近点对

直线划分点阵→分治算法~

# ⑧关键路径

2022年2月13日　　20:02

PS：若要求仅为编程求解最短路径及最小权值和，则直接用dp即可！
*关键路径（AOE）的概念&公式

- 事件$V_j$ 的最早发生时间$ve(j)$
  是从源点$V_0$到顶点$V_j$ 的最长路径长度。
- 事件$V_j$ 的最迟发生时间$vl(j)$
  是在保证汇点$V_{n-1}$ 在$ve(n-1)$时刻完成的前提下，事件$V_j$ 的允许的最迟开始时间。
- 活动$a_i$ 的最早开始时间$e(i)$
  设活动$a_i$ 在弧$<V_j, V_k>$上，则$e(i)$是从源点$V_0$到顶点$V_j$ 的最长路径长度。因此，$e(i) = ve(j)$。
- 活动$a_i$ 的最迟开始时间$l(i)$
  $l(i)$是在不会引起时间延误的前提下，该活动允许的最迟开始时间。 $l(i)= vl(k) - dur(<j, k>)$。其中，$dur(<j, k>)$是完成$a_i$所需的时间。

*ve(j)、vl(j)公式：
*省流：ve(j)通过dp求，其他根据图中公式代入即可~
*详细：

- 从$ve(0) = 0$开始，向前递推

$$ve(j) = \max_i\{ ve(i) + dur(<V_i, V_j>) \},$$

$$<V_i, V_j> \in T,\ j = 1, 2, \ldots, n-1$$

其中$T$是所有以$V_j$为头的弧的集合。

- 从$vl(n-1) = ve(n-1)$开始，反向递推

$$vl(i) = \min_j\{ vl(j) - dur(<V_i, V_j>) \},$$

$$<V_i, V_j> \in S,\ i = n-2, n-3, \ldots, 0$$

其中$S$是所有以$V_i$为尾的弧的集合。

- $e(i) = ve(j),\ l(i)= vl(k) - dur(<j, k>)$

- 时间余量 $l(i) - e(i)$
  表示活动$a_i$ 的最早开始时间和最迟开始时间的时间余量。
  $l(i) == e(i)$表示活动$a_i$ 是没有时间余量的关键活动。

*关键路径判断：

- 为找出关键活动，需要求各个活动的 $e(i)$ 与 $l(i)$，以判别是否 $l(i) == e(i)$。
- 为求得$e(i)$与 $l(i)$，需要先求得从源点$V_0$到各个顶点$V_j$的 $ve(j)$和 $vl(j)$。

# Python

2023年1月28日 23:00

# 0、⑦、⑧

2023年2月23日　　19:06

见C[++]~

# ①拓扑排序

2022年2月12日　　22:23

①原问题

```
def topoSort(graph):
    in_degrees = dict((u,0) for u in graph)   #初始化所有顶点入度为0
    num = len(in_degrees)
    for u in graph:
        for v in graph[u]:
            in_degrees[v] += 1     #计算每个顶点的入度
    Q = [u for u in in_degrees if in_degrees[u] == 0]    # 筛选入度为0的顶点
    Seq = []
    while Q:
        u = Q.pop()        #默认从最后一个删除
        Seq.append(u)
        for v in graph[u]:
            in_degrees[v] -= 1     #移除其所有出边
            if in_degrees[v] == 0:
                Q.append(v)            #再次筛选入度为0的顶点
    if len(Seq) == num:        #输出的顶点数是否与图中的顶点数相等
        return Seq
    else:
        return None

```

②推广

*思想很重要！

# ②单源最短路径

2022年2月12日　　22:24

①Dijkstra

（Ⅰ）应用方向：

主要针对的是有向图的单元最短路径问题，且不能出现权值为负的情况

（Ⅱ）总体思想：

假设存在G=<V,E>，源顶点为V0，S={V0},distance[i]记录V0到i的最短距离，matrix[i][j]记录从i到j的边的权值，即两点之间的距离。
1）从V-S中选择使dist[i]值最小的顶点i，将i加入到U中；
2）更新与i直接相邻顶点的dist值。dist[j]=min{dist[j],dist[i]+matrix[i][j]}
3）直到S=V，所有顶点都包含进来了，算法停止。

（Ⅲ）代码实现：

```
def dijkstra(s):
    distance[s] = 0
    while True:
        # v在这里相当于是一个哨兵，对包含起点s做统一处理！
        v = -1
        # 从未使用过的顶点中选择一个距离最小的顶点
        for u in range(V):
            if not used[u] and (v == -1 or distance[u] < distance[v]):
                v = u
        if v == -1:
            # 说明所有顶点都维护到S中了！
            break

        # 将选定的顶点加入到S中，同时进行距离更新
        used[v] = True
        # 更新U中各个顶点到起点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离；例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。
        for u in range(V):
            distance[u] = min(distance[u], distance[v] + cost[v][u])
```

（Ⅳ）特点：复杂度略高（二次方）

②Bellman ford

（Ⅰ）应用方向：

有向图，且允许出现负权重~

（Ⅱ）总体思想：

首先初始化，对所有的节点V来说，所有的边E进行松弛操作，再然后循环遍历每条边，如果d[v] > d[u] + w(u,v),表示有一个负权重的环路存在。最后如果没有负权重环路，那么d[v] 是最小路径值。

（Ⅲ）代码实现：（graph即保存节点与边信息的二维数组）

```
def bellman_ford(graph, source):
    dist = {}
    p = {}
    max = 10000
    for v in graph:
        dist[v] = max  #赋值为负无穷完成初始化
        p[v] = None
    dist[source] = 0

    for i in range(len( graph ) - 1):
        for u in graph:
            for v in graph[u]:
                if dist[v] > graph[u][v] + dist[u]:
                    dist[v] = graph[u][v] + dist[u]
                    p[v] = u    #完成松弛操作，p为前驱节点

    for u in graph:
        for v in graph[u]:
            if dist[v] > dist[u] + graph[u][v]:
                return None, None  #判断是否存在环路

```

```
22          return dist, p
```

（Ⅳ）特点：复杂度略高（二次方）

③SPFA算法

（Ⅰ）应用方向：

有向图，且允许出现负权重~

（Ⅱ）总体思想：

每次选取队首顶点u的所有边进行松弛操作，假设有一条u到v的边，如果通过这条边使得源点到顶点v的最短路程变短，且顶点v不在当前队列中，就将顶点v放入队尾。

（Ⅲ）代码实现：

```
from collections import deque
q=deque()
q.append/extend(…)
book=[]
def spfa():
    while not q:
        t=q.popleft()
        for i in graph[t]: #t的所有邻边
            if ans[u]+edge[i][j]<ans[v]:
                q.append(v)
                if book[v]>n-1: #有负环
                    return
```

# ③深度优先搜索（DFS）

2022年1月28日　　22:10

（Ⅰ）应用方向：

*几乎可以应用于任何问题！

*适用于找出所有解的问题

*在深度较大时效率较低

（Ⅱ）总体思想：

*就是枚举

*每次都把路先走到最深，实在走不下去了再一步步逐一回溯搜索，用递归函数或栈来实现

*同样是深搜，也要尽量寻找简单的思路！（如果涉及了太多条件则也要思考有无简单办法）

（Ⅲ）代码实现：

大致代码：（python）

```
def dfs(x,y):#参数在问题为一维时也可以是step
    if not 0<=x<m or 0<=y<n or book[x][y] or … :#已经走到尽头了(也有可能是step==n等等）
        …#执行走到尽头时该干的事（检验是否满足条件然后count++等等）
        return
    book[x][y]=1 #标记这路走过了
    … #此时还需要执行的操作
    dfs(x-1,y) #开始遍历所有的路
    dfs(x,y+1)
    dfs(x+1,y)
    dfs(x,y-1)
    #有的题目还需要book[x][y]=0来把走过的路收回！
#写主函数
for i … :
    for j … :
        if book[i][j]==0 and … :
            dfs(i, j)
```

★ PS：本来为取值为0，1的二维题可以省去book！

2.具体示例：

（对应问题： 求出123的全排列，即123　132　213　231　312　321）

```
#include <stdio.h>
#include <stdlib.h>
//在全局变量中,数组默认为0
int a[5], book[5], n = 3;
void Dfs(int step)
{
        int i;
        //当数组a存放了1-3所有元素且每个数都是被标记过的,证明该序列满足
        if (step == n + 1){
             //执行路走到尽头时该做的操作
              for (i = 1; i <= n; i++)
                    printf("%d", a[i]);
```

```
12                  printf("%d ", a[i]);
13            printf("\n");
14            return;
15      }
16      for (i = 1; i <= n; i++){
17            //判断该位置是否被标记过
18            if (book[i] == 0){
19                  //再往深里走一步时要执行的操作，此处即把i存入到数组的第step的位置中
20                  a[step] = i;
21                  //标记i位置证明i已经存入到a数组当中
22                  book[i] = 1;
23                  //更新step继续执行循环
24                  Dfs(step + 1);
25                  //将刚标记的位置收回,执行下一次尝试，这是非常重要且容易忘的一步！！！
26                  book[i] = 0;
27            }
28      }
29      return;
30  }
31
32  int main()
33  {
34      //首先从第一个数字开始
35      Dfs(1);
36      system("pause");
37      return 0;

}
```

# ④广度优先搜索（BFS）

2022年1月29日　　14:29

（Ⅰ）应用方向：

适用于求解最值问题（因为它搜索到的解一定是路径最短的解）

（Ⅱ）总体思想：

*就是枚举

*每次都把能走的路都试了，不再回溯

*找到了即停止搜索

（Ⅲ）代码实现：

*用队列

1.大致代码：（python）

PS：popleft()函数需要额外导入库：

```
from collections import deque
        queue=deque()
        queue.append((x,y))
```

```
for i … : #实际上属于"一main到底",此处看情况需不需要对queue进行初始的元素加入(如果这样一般无需再双重循环遍历)
    for j … :
        if book[i][j]==0 and … :
            queue.append([x,y])#queue不是一般的列表，而是双向队列deque！
            book[i][j]=1
            cnt+=1 #记录广搜的总次数（看题意需不需要记录，记得cnt=0初始化！）
            while queue:
                x,y=queue.popleft()#注意这里是popleft而非pop！
                for nx,ny in [[x-1,y],…]:
                    if book[nx][ny]==0 and … :
                        book[nx][ny]=1
                        queue.append([nx,ny])
```

★ PS：本来为取值为0，1的二维题可以省去book！

*还有一种广搜的代码类型（适合求解最优解时用）：

```
queue.append(n,0) #n为最终目标，0为目前步数
while queue:
    tm,step=queue.popleft()
    for i … : #遍历所有可能路径
        if … : return step+1 #已经满足目标则返回
        elif … and not book[…]: #仍没达目标但满足题目条件时，若未标记则继续加入队列
            queue.append(tm-…,step+1)
            book[…]=1
return -1 #全部遍历完仍未搜索到则返回-1
```

★ *若为要求记录走过的路径：不再用队列而是用栈！！！（PS：此时一般用DFS顺手一点，除非题目特意要求求出路径最短的解）

```
stack=[]
stack.append([start_x,start_y]) #起点
next=[[-1,0],[0,1],[1,0],[0,-1]]
ans=[]
l=0
while stack:
    temp_x,temp_y=stack.pop()
    ans.append([temp_x,temp_y])
    l+=1
    if temp_x==end_x and temp_y==end_y:
        for i in ans:
            print("({},{})->".format(i[0],i[1]),end="")
        print("\b\b")
        break
    for i in next:
        if up_boundary<=temp_x+i[0]<=down_boundary and left_boundary<=temp_y+i[1]<=right_boundary and not maze[temp_x+i[0]][temp_y+i[1]]: #maze意为迷宫
            maze[temp_x+i[0]][temp_y+i[1]]=1
            stack.append([temp_x+i[0],temp_y+i[1]])
    while ans:
        if ans[-1] not in stack: ans.pop()
    if not ans:
        print("Not found.")
        break
```

2.具体示例：

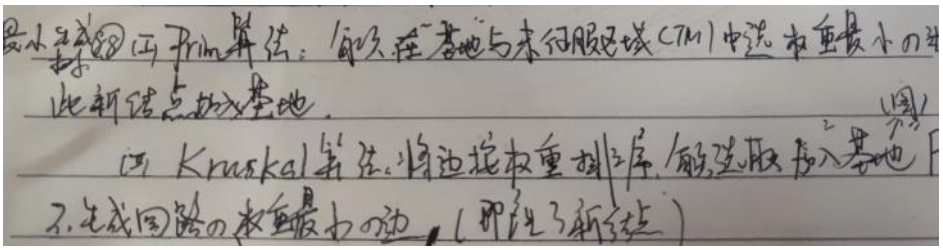
（啊哈算法里的拯救小美）

```cpp
#include<iostream>
#include<algorithm>
#include<stack>
#include<set>
#include<queue>
#include<map>
#include<string>
#include<cstring>
#include<vector>
#include<ctype.h>
#include<sstream>
#include<unordered_map>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
struct node
{
    int x,y;
    int s;
    int f;//父亲在队列中的编号。？？？干啥用的呢？有点像课本上的那个呀。
};
node que[2501];
int head,tail;
//mmp数组表示地图
int mmp[51][51];
//Book数组用来记录哪些点已经在队列中，防止一个点被重复扩展.....为啥教材上没有呢？
int Book[51][51];
//定义一个用于表示走的方向的数组
int Next[4][2]={{0,1},{1,0},{0,-1},{-1,0}};

int main()
{
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
        cin>>mmp[i][j];
    int startx,starty,p,q;
    cin>>startx>>starty>>p>>q;
    head=tail=1;
    que[tail].x=startx;
    que[tail].y=starty;
    que[tail].f=0;
    que[tail].s=0;
    tail++;
    Book[startx][starty]=1;
    int flag=0;
    while(head<tail)
    {
        int tx,ty;
        for(int i=0;i<3;i++)
        {
            tx=que[head].x+Next[i][0];
            ty=que[head].y+Next[i][1];
            if(tx<1||ty<1||tx>n||ty>m) continue;
            if(Book[tx][ty]==0&&mmp[tx][ty]==0)
            {
                que[tail].x=tx;
                que[tail].y=ty;
                tail++;
                Book[tx][ty]=1;
                mmp[tx][ty]=1;
            }
        }
        if(que[head].x==p&&que[head].y==q)
        {
            flag=1;
            break;
        }
        head++;
    }
    if(flag==1) cout<<que[head].x<<" "<<que[head].y<<endl;
    return 0;

}

/*
5 4
0 0 1 0
0 0 0 0
0 0 1 0
0 1 0 0
0 0 0 1
1 1 4 3
*/
```

# ⑤最小生成树（MST）

2022年8月29日　　12:51

*Kruskal、Prim算法:



```
# Prim算法
base=[]
tm=pointset
base=tm.pop()
n=len(edge)
inf=10000… #一个超过给定边权最大值的数
edge=[[inf for i in range(n)] for i in range(n)]
edge… #输入
while tm:
    for i in base:
        best=inf
        for j in tm:
            if edge[i][j]!=inf and edge[i][j]<best:
                best=edge[i][j]
                ans=j
            base.append(ans)
            tm.remove(ans)
```

# ⑥多源最短路径

2022年2月13日　　10:07

①Floyd

```
for k in range(n):
      for i in range(n):
            for j in range(n):
                  graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])
```

②Johnson

（Ⅰ）应用方向：

对于稀疏矩阵，减少了其在空间上的存储浪费，同时提高了效率

（Ⅱ）总体思想：

结合单源最短路径中的几个算法（Bellman-Ford+Dijkstra）来解决（但不能理解？）

（Ⅲ）代码实现：

```
# 求解稀疏矩阵图
from copy import deepcopy

def johnson(G):
    G = deepcopy(G)
    s = object()
    G[s] = {v: 0 for v in G}
    h, _ = bellman_ford(G, s)
    del G[s]
    for u in G:
        for v in G[u]:
            G[u][v] += h[u] - h[v]
    D, P = dict(), dict()
    for u in G:
        D[u], P[u] = dijkstra(G, u)
        for v in G:
            D[u][v] += h[v] - h[u]
    return D, P
```

# *（XDU）邻接矩阵转邻接表

2022年12月2日　　10:05

*只允许用链表实现！（指针数组+malloc分配的内存可能互相干扰，不连续）

```
#include<malloc.h>
#define Max ...

typedef struct ListNode
{
    struct ListNode *next=NULL;
    int val;
}ListNode;

int map[Max][Max];
ListNode *table_head[Max],*table_tail[Max];

for(i=1;i<=N;i++)
    table_tail[i]=table_head[i];

for(int i=1;i<=N;i++)
{
    for(int j=1;j<=N;j++)
    {
        if(map[i][j])
        {
            ListNode *temp;
            temp=(ListNode*)malloc(sizeof(ListNode));
            temp->val=j;
            table_tail[i]->next=temp;
            table_tail[i]=table_tail[i]->next;
        }
    }
}
```

# 树状数组

2023年2月4日 19:48

①应用方向

*维护（快速修改）区间信息，快速获取信息（求和等）

*线段树完全替代树状数组功能且树状数组相较其不能修改维护，但线段树更复杂

②代码实现（Java版，其他语言类似）

```
import java.util.scanner;
public class Main
{
    //下标都从1开始~
    static int maxlenth=MAX;
    static int [] c=new int[maxlenth];//树状数组
    static int [] pre=new int [len_max];//前缀和数组
    static int lowbit(int x)
        { //计算能整除x的最小的2的幂
        return x&(-x);
    }
    static void add(int i,int value) //插入
        { //i为插入的下标
        while(i<=maxlenth)
           {
            c[i]+=value;
            i+=lowbit(i);
        }
    }
    static int sum(int i) //前i个的前缀和
        {
        int sum=0;
        while(i>0)
            {
            sum+=c[i];
            i-=lowbit(i);
        }
        return sum;
    }
    public static void main(String[] args)
   {
        for(int i=1;i<=len;i++)
        {
            int t=scanner.nextInt();
            add(i, t);
        }
        for(int i=1;i<=len;i++) pre[i]=sum(i);
```

```
    //之后每次查询局部和只需根据前缀和数组作差即可~
    }
}
```

# ①排序

2022年5月4日　18:12

0各种排序算法的比较和灵活运用：

（Ⅰ）比较：

| 排序方法 | 平均时间 | 最好时间 | 最坏时间 |
| --- | --- | --- | --- |
| 桶排序(不稳定) | O(n) | O(n) | O(n) |
| 基数排序(稳定) | O(n) | O(n) | O(n) |
| 归并排序(稳定) | O(nlogn) | O(nlogn) | O(nlogn) |
| 快速排序(不稳定) | O(nlogn) | O(nlogn) | O(n^2) |
| 堆排序(不稳定) | O(nlogn) | O(nlogn) | O(nlogn) |
| 希尔排序(不稳定) | O(n^1.25) | | |
| 冒泡排序(稳定) | O(n^2) | O(n) | O(n^2) |
| 选择排序(不稳定) | O(n^2) | O(n^2) | O(n^2) |
| 直接插入排序(稳定) | O(n^2) | O(n) | O(n^2) |

（Ⅱ）灵活运用：

（1）当数据规模较小的时候，可以用简单的排序算法如直接插入排序或直接选择排序。

（2）当文件的初态已经基本有序时，可以用直接插入排序或冒泡排序。

（3）当数据规模比较大且较为无序时，应用速度快的排序算法，可以考虑用快速排序。

（4）堆排序不会出现快排那样的最坏情况，且堆排序所需的辅助空间比快排要少，但这两种算法都不是稳定的。若要求排序时稳定的，可以考虑用归并排序。

*（5）归并排序可以用于内排序，也可以用于外排序。在外排序时，通常采用多路归并，并且通过解决长顺串的合并，产生长的初始串，提高主机与外设并行能力等措施，以减少访问外存额次数，提高外排序的效率。

①插入排序

（Ⅰ）总体思想：每次将第i个数插入到前面i-1个排好序的数中

（Ⅱ）代码实现：略~

②希尔排序

（Ⅰ）总体思想：设置一个gap变量，其从数组长度/2开始逐渐再除二取整（直到gap为1），期间以gap为间隔的子数组分别各自内部排好序

（Ⅱ）代码实现：

```c
#include <stdio.h>
#include <malloc.h>
void shellSort(int *a, int len)
{
    int i, j, k, tmp, gap;  // gap 为步长
    for (gap = len / 2; gap > 0; gap /= 2)
    {  // 步长初始化为数组长度的一半，每次遍历后步长减半，
        for (i = 0; i < gap; ++i)
        { // 变量 i 为每次分组的第一个元素下标
            for (j = i + gap; j < len; j += gap)
            { //对步长为gap的元素进行直插排序，当gap为1时，就是直插排序
                tmp = a[j];  // 备份a[j]的值
                k = j - gap;  // j初始化为i的前一个元素（与i相差gap长度）
                while (k >= 0 && a[k] > tmp)
                {
                    a[k + gap] = a[k]; // 将在a[i]前且比tmp的值大的元素向后移动一位
                    k -= gap;
                }
                a[k + gap] = tmp;
            }
        }
     //此处可以输出中途每一步操作的排序结果
    }
}

int main()
{
    int i, len, * a;
    printf("请输入要排的数的个数：");
    scanf("%d",&len);
    a = (int *)malloc(len * sizeof(int)); // 动态定义数组
    printf("请输入要排的数：\n");
    for (i = 0; i < len; i++)
```

```
        scanf("%d",&a[i]);
    shellSort(a, len); // 调用希尔排序函数
    printf("希尔升序排列后结果为: \n");
    for (i = 0; i < len; i++)
        printf("%d\t",a[i]);
    return 0;
}
```

③快速排序

（Ⅰ）总体思想：将每次的最左数作为基准数（枢轴值），两边各设置一个哨兵分别往中间走，遇到大于/小于基准数的就交换，直到哨兵相遇，此时基准数已经到了最终的位置，且其左右的数分别比它都小/大（实际上隐含了一颗递归树）（其中第一枢轴值为第一次此过程完成后两个哨兵交汇点的数组元素值）

（Ⅱ）代码实现：

1.若要求输出每一轮结果（即中途必须老老实实交换值）：

```
void Qs(int *n,int start,int end)
{
    int temp=n[start],left=start,right=end;//一定要记录下初始的起止点！！！
    if(left>right) return;
    while(left<right)//真正的快排（如此处）中途是会交换值的，只不过是我自己改写的简便写法不需要！！！
    {
        while(left<right && n[right]>=temp) right--;
        n[left]=n[right];
        while(left<right && n[left]<=temp) left++;
        n[right]=n[left];
    }
    n[left]=temp; //此时的left为哨兵相遇的地方
    Qs(n,start,left-1);//这里的起始点是start而非left！！！
    Qs(n,left+1,end);//同上！
    return;
}
```

2.仅仅是为了省时间（简便写法，可不交换值）

```
void qs(int *n, int len)
{
        if(len<2) return;
        int t=n[0],i,*l,*r,llen=0,rlen=0;
        l=(int*)malloc(sizeof(int));
        r=(int*)malloc(sizeof(int));
        for(i=0;i<len;i++)
        {
                if(n[i]<=t) l[llen++]=n[i];
                else r[rlen++]=n[i];
        }
        qs(l,llen);
        qs(r,rlen);
        for(i=0;i<llen;i++) n[i]=l[i];
        n[llen]=t;
        for(i=0;i<rlen;i++) n[i+llen+1]=r[i];
        return;
}
```

```
def qs(array):
    if len(array) < 2: return array
    t = array[0]
    l = [i for i in array[1:] if i <= t] #注意此处i即为array[…]！
    r = [i for i in array[1:] if i > t]
    return qs(l) + [t] + qs(r)  #这里可能会报错（不能相加），如果这样就开个temp[]列表去append元素t、extend列表l和r
```

（Ⅲ）特点：效率大多时候极高、不稳定

④堆排序

（Ⅰ）总体思想：建立最大堆，每次删除堆顶点（当前最大值）并将其传到记录的数组里

（Ⅱ）代码实现：

1.直接库函数实现（无论是要求输出每一轮结果还是仅仅是为了省时间）

```
#include<algorithm>
#include<functional> //用来支持大顶堆
using namespace std;
int heap[Max],ans[Max];
for(int i=1;i<=len;i++)
{
    make_heap(heap+1,heap+len+2-i,[less/greater<int>()]);//less为大顶堆
    ans[i]=heap[0];
    heap[0]=heap[len+1-i];
    //输出每一轮结果：
    //int j;
        //每一轮已排好序的元素：
        //for(j=1;j<=i;j++) cout<<ans[j]<<" ";
        //cout<<endl;
```

```
        //每一轮剩下的堆：
        //for(j=1;j<=len-i;j++) cout<<heap[j]<<" ";
}
```

2.具体过程

```
//heap下标从1开始
void HeapAdjust(int *n, int p, int len)
{
    int i,temp;
    temp=n[p];
    for(i=2*p;i<=len;i*=2)
    {
        if(i<len && heap[i]<heap[i+1]) i++;
        if(temp>=heap[i]) break;
        heap[p]=heap[i];
        p=i;
    }
    heap[p]=temp;
}

int* HeapSort(int *heap, int len)
{
    int i,ans[Max];
    //构造大顶堆
    for (i=len/2;i>0;i--) HeapAdjust(heap,i,len);
    //依次取出当前的最大元素
    for (i=len;i>=1;i--)
    {
        ans[i]=heap[1];
        heap[1]=heap[i];
        HeapAdjust(heap,1,i-1);
       //此处可以输出中途每一步操作的排序结果
    }
    return ans;//ans为升序排列!
}
```

⑤归并排序

（Ⅰ）总体思想：将数组不断两两拆分成子数组，直到不能再分后，先各自排好序，后不断递归将子数组两两合并，直到全部合并

（Ⅱ）代码实现：

```
#include<stdio.h>
#define ArrLen 20
void printList(int arr[], int len)
{
   int i;
   for (i = 0; i < len; i++) printf("%d\t", arr[i]);
}

void merge(int arr[], int start, int mid, int end)
{
        int result[ArrLen];
        int k = 0;
        int i = start;
        int j = mid + 1;
        while (i <= mid && j <= end)
    {
                if (arr[i] < arr[j]){
                    result[k++] = arr[i++];
        }
        else{
                result[k++] = arr[j++];
        }
        }
        if (i == mid + 1) {
                while(j <= end)
                        result[k++] = arr[j++];
        }
        if (j == end + 1) {
                while (i <= mid)
                        result[k++] = arr[i++];
        }
        for (j = 0, i = start ; j < k; i++, j++) {
                arr[i] = result[j];
```

```
        }
    }
}

void mergeSort(int arr[], int start, int end)
{
        if (start >= end)
                return;
        int mid = ( start + end ) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid + 1, end);
        merge(arr, start, mid, end);
}

int main()
{
        int arr[] = {4, 7, 6, 5, 2, 1, 8, 2, 9, 1};
        mergeSort(arr, 0, 9);
        printList(arr, 10);
        system("pause");
        return 0;
}
```

（Ⅲ）特点：效率高、稳定

⑥基数排序

（Ⅰ）总体思想：

由低位依次向更高位，每轮（一位）按基数（0-9，a-z）排好序（*一样则按原序）（若位数不同则可以填充最小值）

（Ⅱ）代码实现：三维数组即可，略~

⑦双向冒泡排序

（Ⅰ）总体思想：

双向冒泡排序是在冒泡排序的基础上改进而来的，其基本思想跟最原始的冒泡排序是一样的，只不过排序过程稍微优化了一点。

我们还是以整数升序排序为例来简单说说这种排序的过程：首先从前往后把最大数移到最后，然后反过来从后往前把最小的一个数移动到数组最前面，这一过程就是第一轮，然后重复这一过程，最终就会把整个数组从小到大排列好。双向冒泡排序要稍微优于传统的冒泡排序，因为双向排序时数组的两头都排序好了，我们只需要处理数组的中间部分即可，而单向即传统的冒泡排序只有尾部的元素是排好序的，这时每轮处理都需要从头一直处理到已经排好序元素的前面一个元素。

（Ⅱ）代码实现：略~

# ②滑动窗口、双指针

2022年4月27日　　20:22

①应用方向

单向遍历时具有类似于二分查找的单调性质（一定要仔细辨认r是否是单调递增，不再回溯！）

②代码实现

滑动窗口

```
l,r,cnt,t=0,-1,0,1
length=len(nums)
for l in range(length):
    if l:t/=nums[l-1]
    while r<length-1 and t*nums[r+1]<target:
        t*=nums[r+1]
        r+=1
          #此时r为满足题意的窗口最右端
        if r!=-1:cnt+=r-l+1 #r=-1的话就不能再加上这个结果了
```

双指针

```
for i in range(l1): #l1,l2分别为数组a,b的长度
    while j<l2 and check(b[j],a[i]):j+=1
    if j>0:j-=1 #非常容易忘的一步！！！
    if j==l2-1:ans=best(ans,func(a[i]-b[j])) #best()就是个取最值~
    else:ans=best(ans,func(a[i],b[j]),func(b[j+1],a[i]))
```

# ③字符串匹配

2022年4月27日　　17:23

*还有BM、RK这两个高效算法，此处略

一、C&C++

①法一：

库函数：

```
#include <cstring>
char p=strstr(str1,str2); //此函数用于判断字符串str2是否是str1的子串。如果是，则该函数返回str2在str1中首次出现的地址；否则，返回NULL
```

*②法二：

KMP算法：

（Ⅰ）总体思想：给pat建立一个状态的索引（只与pat有关），从而在匹配时只要单向遍历txt即可，无需回溯搜索

（Ⅱ）代码实现：

完整KMP算法过程：

```
int Index_KMP(string main,string pattern)
{    //KMP模式匹配算法
    int i=pos,j=1;
    while(i<=main[0] && j<=pattern[0])
    {
        if(!j || main[i]==pattern[j]) //继续比较后继字符串
        {
            i++;
            j++
        }
        else j=next[j]; //pattern，即模式串向右移动
    }
    if(j>pattern[0]) return i-pattern[0];//匹配成功
    else return 0;
}
```

next数组求解：

```
void get_next(string pattern,int *next)
{       //求next数组
    int i=1,j=0;next[1]=0;
    while(i<pattern[0])
    {
        if(!j || pattern[i]==pattern[j])
        {
            i++;
            j++;
            next[i]=j;
        }
        else j=next[j]; //next[1]到next[j]已经有值
    }
}
```

（Ⅲ）时间复杂度

O(M+N)！！！

二、Python

PS:若为字符串在字典中的匹配可以考虑字典树

定义：pat（长度为M）字符串在txt文本（也是字符串，长度为N）中进行匹配

0利用库函数：

```
if str2 in str1: … #布尔值
str1.find(str2) #匹配成功则返回str2在str1中的开始下标，若匹配不成功则返回-1
```

*①KMP算法

（Ⅰ）总体思想：给pat建立一个状态的索引（只与pat有关），从而在匹配时只要单向遍历txt即可，无需回溯搜索

（Ⅱ）代码实现：

```
#KMP算法
#首先计算next数组，即我们需要怎么去移位
```

```
#接着我们就是用暴力解法求解即可
#next是用递归来实现的
#这里是用回溯进行计算的
def calNext(str2):
    i=0
    next=[-1]
    j=-1
    while(i<len(str2)-1):
        if(j==-1 or str2[i]==str2[j]):#首次分析可忽略
            i+=1
            j+=1
            next.append(j)
        else:
            j=next[j]#会重新进入上面那个循环
    return next
print(calNext('abcabx'))#-1,0,0,0,1,2

def KMP(s1,s2,pos=0):#从那个位置开始比较
    next=calNext(s2)
    i=pos
    j=0
    while(i<len(s1) and j<len(s2)):
        if(j==-1 or s1[i]==s2[j]):
            i+=1
            j+=1
        else:
            j=next[j]
    if(j>=len(s2)):
        return i -len(s2)#说明匹配到最后了
    else:
        return 0
s1 = "acabaabaabcacaabc"
s2 = "abaabcac"
print(KMP(s1,s2))
```

（Ⅲ）时间复杂度

O(M+N)！！！

三、Java

```
int index=string1.indexOf(string2[,start_index]) //返回[从指定下标开始]第一次出现的指定子字符串在此字符串中的索引,若没找到则返回-1
```

# ④博弈论

2023年1月30日 22:19

0基础知识

后继（局面）：某步操作局面后每一步操作导致的局面

①所有ICG游戏（普适）

*对于一个ICG游戏，可以将其拆分为若干个分游戏，每个游戏的局面都可以用SG函数值描述状态，而总游戏的状态则为其分游戏异或的结果。

SG函数：

为0则必败，非零则必胜

SG(x)=mex{SG(y)|y是x的后继}；其中mex(A)=最小的不属于集合A的非负整数

②典型

（Ⅰ）Nim游戏

1.内容

若干堆石子，每堆石子的数量都是有限的，双方轮流选择一堆石子，并取走至少1个石子，取走最后一个石子的玩家获胜（对方无法操作）。

2.结论

假设有n堆石子，第i堆石子数量为ai，则若a1⊕a2⊕...an=0，先手必败，反之先手必胜。

3.证明提示

数列a都转化为二进制后，使得数列每一位上1的个数都为偶数

（Ⅱ）反nim游戏

1.内容

取走最后一个石子的玩家输~

2.结论

先手必胜当且仅当：

所有堆的石子数都为1且游戏的SG值为0
至少一堆石子数大于1且游戏的SG值不为0

（Ⅲ）树上删边

1.内容

给定一个有N个节点的树，节点1是树的根

双方轮流从树上选一条边删除，并移除所有和根节点不再相连的点（及与这些点关联的边）

先无法操作的玩家输掉游戏。

2.结论

叶子节点的SG值为0，其它节点的SG值为其所有子节点的SG值加1后异或的结果。

（PS:树上加简单环：删除树上的偶环并将奇环转换为长度为1的链后转化为普通问题，结果不变，即二者等效）

（Ⅳ）无向图删边

结论：将图中的偶环删掉，变为一个新点；将图中的奇环删掉，变为一个新点+新边，原图中所有与环相连的点连接到新点上

# ⑤数论

2022年2月12日　　15:23

①最大公约数

```
def gcd(a,b):
    if not b:
        return a
    else:
        return gcd(b, a%b)
```

int gcd(int m, int n) {return n? gcd(n, m% n): m;}

②中国剩余定理

（Ⅰ）定理内容

正整数m1,m2,...,mk两两互素，对b1,b2,...,bk的同余式组为

$$\begin{cases} x \equiv b_1 \ mod \ m_1 \\ x \equiv b_2 \ mod \ m_2 \\ \quad \vdots \\ x \equiv b_k \ mod \ m_k \end{cases}$$

在mod M

$$M = \prod_{i=1}^{k} m_i$$

的情况下有唯一解

$$x = (\sum_{i=1}^{k} b_i M_i M_i') \ mod \ M$$

其中

$$M_i = \frac{M}{m_i}$$

$$M_i' = M_i^{-1} \ mod \ m_i$$

注：Mi^-1为Mi的逆元，即二者相乘模mi为1

（Ⅱ）代码实现：

其他部分自己照着（Ⅰ）写，给出求逆元算法：

```
def inv(n,m):#n已经保证比m小了
    if n==1: return 1
    else: return (m-m//n)*inv(m%n)%m
```

# ⑥局部操作

2023年2月1日　　22:01

*快速幂

```
long pow (long cnt) {
        // 如何快速计算a的cnt次幂?
        if (cnt == 0) return 1;
        if (cnt == 1) return a;
        long part = pow(cnt / 2);
        if (cnt % 2 == 0) return part * part;
        return a * part * part;
    }
```

*通过加括号改变运算顺序，统计加括号的方法数：遍历所有运算符，以运算符为界分割成左右两部分，分别统计两部分的方法数之和（然后相乘，最后遍历一遍将其全部相加即可！）

（关键是不要一开始就O(N^2)地插入括号然后就分成三部分）

# dp

2022年4月27日　　17:49

①适用条件：

*适用于有重叠子问题和最优子结构性质的问题，并且记录所有子问题的结果（有重叠就有希望，看上去不太合理的一遍遍历也有可能成功！）

*无后效性：一旦一个子问题的求解得到结果，以后的计算过程就不会修改它

②基本思路

*核心在于建立状态转移方程，从而实现递归（PS:状态之间可能会有二维的关系，即还得分行列两种情况再递归下去）

*一种递归方式不行的话试试另一种思路

*建立备忘录或者每次更新一遍有关的变量（“我为人人/人人为我”）

*自底向上：递推

自顶向下：记忆化递归

*试着以做一些有价值的记录为切入口，用空间置换时间！

*可以尝试**状压DP**:进制编码化，用编号代表状态~

③一些经验技巧、注意点

*dp数组初始值可设置为-1，便于在dp某元素实际应有值就为零，遍历过后自然还是零时区分它有没有遍历过（不然还得用一个book数组去记录）

*dp数组设置长度时至少要包括已经定好的几个dp元素值！（如已经确定dp[0]=0,dp[1]=1,dp[2]=2,则dp长度至少应为3，此时用max(len(list),3)作为其长度即可）

*遍历可以试着从右往左遍历，可能更简单！

# ①二分查找

2022年4月27日　　17:27

①应用方向：

*适用于所有在单调线性的区间上进行定位的问题！（务必确定是完全的单调！）

*有时滑动窗口优于二分查找！

②代码实现：（供模板使用）

（Ⅰ）查找分＞、＜、=三种情况，只有一个相等的值时：（经典二分查找）

1.库函数

*C&C++

lower_bound(数组名+l，数组名+r，tm)：返回大于或等于目标值的第一个位置
upper_bound(数组名+l，数组名+r，tm)：返回大于目标值的第一个位置
binary_search(数组名+l，数组名+r，tm)：若目标值存在则返回true，否则返回false

*Python

库函数 bisect()，bisect_left()和bisect_right()：

bisect()和bisect_right()等同。

如果列表中没有元素x，那么bisect_left(ls, x)和bisec_right(ls, x)返回相同的值，该值是x在ls中"合适的插入点索引，使得数组有序"。

如果列表中只有一个元素等于x，那么bisect_left(ls, x)的值是x在ls中的索引，ls[index2] = x。而bisec_right(ls, x)的值是x在ls中的索引加1。

如果列表中存在多个元素等于x，那么bisect_left(ls, x)返回最左边的那个索引，此时ls[index2] = x。bisect_right(ls, x)返回最右边的那个索引加1。（结合两类函数的定义可以判断列表中是否含有此函数）

*Java

```
1  public static int binarySearch(Object[] a, Object key); //有则返回索引无则返回-1
```

2.手搓

```
l=head
r=tail
while l<=r:
    m=(l+r)//2
    if a[m]<TM:
        l=m+1
    elif a[m]>TM:
        r=m-1
    else:
        break
#a[m]即为TM值
```

（Ⅱ）只分两种情况（即从某个值开始往上/下就一直满足条件），用类型为布尔值的check函数检验，寻找值为True的最值时：

（无库函数，只能手搓~）

```
l=head
r=tail
while l<=r:
        m=(l+r)//2
        if check(): #如果是查找最小值则添一个not
            l=m+1
        else:
            r=m-1
#求最小/大值时，l/r为TM
```

# *②最大子数组问题

2022年1月28日　15:01

①问题：输入一个整形数组（有正数也有负数），数组中连续的、一个或多个元素组成一个子数组，每个子数组都有一个和。求所有子数组的元素和中间的最大值。

②解答

（Ⅰ）总体思想：

1）分--将原数组拆分成两部分，每个部分再拆分成新的两部分.....直到数组被分得只剩下一个元素；

2）治--每个小型的数组找最大子数组，只有一个元素的数组，解就是该元素；

3）合--将两个小型数组合并为一个数组，其中解有三种可能：

左边的返回值大，

右边的返回值大，

中间存在一个更大的子数组和；

返回值应选最大的。

其中两个数组合并的时候，位于两个数组中间位置存在最大和的情况，处理方法为：

从中间位置开始，分别向左和向右两个方向进行操作，通过累加找到两个方向的最大和，分别为l_max和r_max，因此存在于中间的最大和为（l_max+r_max）；

★ PS:动态规划也能做，时间复杂度更低，仅为O(N)！

（Ⅱ）代码实现：

```
int Divide(int *arry,int l,int r)//将分与治融合在一个函数里，写的顺序是先治后分，参数是l和r
 {
     if(l==r)//只有一个元素时，返回该元素
         return arry[l];
     else
     {
         int m=(l+r)/2;
         int l_max=MIN,r_max=MIN,m_max=MIN;
         l_max=Divide(arry,l,m);//左边和的最大值
         r_max=Divide(arry,m+1,r);//右边和的最大值
         m_max=MiddleMax(arry,l,r,m);//中间和的最大值
         //返回三个值中最大的一个
         if(l_max>=r_max && l_max>=m_max)
             return l_max;
         else if(r_max>=l_max && r_max>=m_max)
             return r_max;
         else
             return m_max;
     }
}

 int MiddleMax(int *arry,int l,int r,int m)
 {
     int l_max=MIN,r_max=MIN;//分别用于记录左、右方向累加的最大和
     int i;
     int sum;//用于求和
     sum=0;
     for(i=m;i>=l;i--)//中线开始向左寻找
     {
         sum+=arry[i];
         if(sum>l_max)
             l_max=sum;
     }
     sum=0;
     for(i=m+1;i<r;i++)//中线开始向右寻找
     {
         sum+=arry[i];
         if(sum>r_max)
             r_max=sum;
     }
     return (l_max+r_max);//返回左右之和
 }
```