逻辑结构：线性结构，树形结构，图结构

存储结构（物理结构）：顺序结构，链式结构，索引结构，散列结构

顺序存储结构定义结构体即可，链式存储因为要申请内存，所以定义结构体指针

常用操作：插入，删除，查找（按值，按位），求长，输出（遍历），判空

//malloc和free在头文件stdlib.h

malloc() //申请连续的内存空间

用法：(elemType\*)malloc(sizeof(elemType)\*size)

/\*申请size个elemType的内存空间（sizeof()返回数据类型的占多少字节），malloc()返回指向该新申请内存空间地址的指针，需要强转成所需的数据类型相应的指针\*/

free() //

new

delete

struct在不重命名的情况下，每次要用到时都要加上struct

struct name

{

….

};

使用时：

strct name a;

typedef：重命名数据类型

typedef int zhengshu;

typedef int\* zhengshuzhizhen;

未使用前：int a; int \*p;

使用后： zhengshu a; zhengshuzhizhen p; //注意不加\*了

定义链表结构时，因为定义结构在重命名前，所以定义next指针要加上struct

定义结构一般形式：

typedef struct LNode

{

……….

//结构体中要用到LNode时，加上struct

}LNode,\*linkList

/\*上面代码等价于

struct LNode

{

……….

//结构体中要用到LNode时，加上struct

}LNode,\*LinkList

typedef struct LNode;

typedef struct \*LinkList;

重命名后，定义LNode类型：LNode n；

定义Lnode的指针：LinkList l; 注意不加\*

此时 LNode\* 等价于LinkList，传递实参时也是一样的

传递指针实参时，该指针必须初始化

要在函数中修改指针自身（如申请内存），必须使用引用或者指针的指针，修改指针指向的变量是地址传递，而修改指针自身则是值传递

\*/

是否用.和->，看前面的

.和->是否要解引用，看后面的，若要解引用，\*(a->p)

若.和->前面是指针的指针，则要解引用(\*a)->data

线性结构：数组/矩阵/广义表，串，线性表，栈，队列

1. 数组/矩阵/广义表：数组下标默认0开始，矩阵下标默认1开始

地址计算：

一维数组

下标0开始：起始地址+siezof(ekemType)\*下标

下标1开始：起始地址+siezof(ekemType)\*（下标-1）

矩阵 a[I][J] m行n列

下标0开始：起始地址+siezof(ekemType)\*（i\*n+j）

下标1开始：起始地址+siezof(ekemType)\*（(i-1)\*m+j-1）

等差数列公式：

求和：S=n(a1+an)/2 通项：an=a1+d(n-1)

对于矩阵a[2][4]=[ 1,2,3,4

5,6,7,8]

行优先存储：1 2 3 4 5 6 7 8

列优先存储：1 5 2 6 3 7 4 8

特殊矩阵的压缩存储：矩阵存储到一维数组中

注意三点：0.看是什么矩阵 1.判断下/上三角 i>/<j 2.行优先还是列优先

3.数组和矩阵下标从什么开始

数组默认0开始，若从1开始+1即可

矩阵默认1开始，这里的i就是第i行

若矩阵下标从1开始，这里的i就是第i+1行

不管矩阵从0还是从1开始，记住行数列数在逻辑上是不变的，从0开始时，将i，j都+1即可得到行列

对称矩阵：a[i][j]=a[j][i],n\*n方阵，\为主对角线，左下角(i>j)/右上角(i<j)，称为下/上三角区域，

一维数组需要存1+2+……+n=n(1+n)/2个元素，下标0到n(1+n)/2-1

矩阵a[i][j]的元素存到一维数组的下标K计算：求a[j][i]只需求a[i][j]即可

行优先：

I>j(下三角)：下标K=1+2+…(行-1)+列-1=行(行-1)/2+列-1

I<j(上三角)：下标K=n+n-1+……+(an=n-行+2)+列-行+1-1=(行-1)(2n-行+2)/2+(列-行+1)-1

列优先：

I>j(下三角)：下标K= n+n-1+……+(an=n-列+2)+行-列+1-1=(列-1)(2n-列+2)/2+(行-列+1)-1

I<j(上三角)：下标K=1+2+…..+(列-1)+行-1=列(列-1)/2+行-1

三角矩阵：

下三角矩阵：上三角全是相同的数

上三角矩阵：下三角全是相同的数

需要存到容量：n(1+n)/2+1 的一维数组 下标：0到n(1+n)/2

计算K与对称矩阵相同，只是K= n(1+n)/2 存放常数

对角矩阵（也叫带状矩阵）：除了第一行最后一行外，沿着主对角线\对称

三对角矩阵：除了第一行最后一行2个元素，其他行3个元素对角线对称

n\*n三对角矩阵需要：2+3+3…..+2=3\*行-2个空间，下标0到3\*行-3

K计算：

行=1，K=列-1

其他行： K=3\*(行-2)+2+(列-行+2)-1

稀疏矩阵：

广义表：

1. 串

KMP算法：

主串失配指针不动，next数组存放下一个要与主串失配处对比的模式串下标，nextval优化了next

求next：模式串数组下标0开始，next[0]=-1 next[1]=0

其他值则是最长前后缀的长度

模式串下标1开始（与next下标无关），则next的值+1

求nextval：If( T[next[i]]==T[i] ) nextval[i]=next[next[i]] ekse照写

散列表（哈希表）：典型的空间换时间，没有冲突的情况时间复杂度O(1)

散列表中的关键字地址都可以用一个散列函数映射

映射到同一地址的关键字互为同义词，此时产生冲突：

链接法（拉链法），数组中存放链表表头，每条链表存放同义词

若查找x，数组相应位置中没有表头，则查找长度=0，对空指针的比较不算做比较，因为没有比较，查找长度是基于比较的

ASL失败=散列表各下标关键字数量之和/散列表表长，这个也叫装填因子，散列表中存放关键字越多，冲突就越多，装填因子就更大，查找效率就变低

设计散列函数：

1. 除留取余法：H(key)=key%p 若散列表表长为m，则p不大于m且最接近m或等于m的质数（质数，也叫素数，只能被1和自己整除，反之叫合数），虽然m!=p时，会损失一些空间，但在一些情况下（如存储的全是偶数），能够减少冲突，但这也不是一定的，比如存储连续关键字，冲突更多了
2. 直接定址法：H(key)=key，或者H(key)=a\*key+b

不会产生冲突但会浪费一些空间，适合用于存放关键字基本连续的情况，若不连续，则会造成更多空间的浪费

1. 数字分析法：适合用于存放较为均匀的若干位，如一个数码，前几位出现不均匀只有几种，而后几位则较为均匀出现频率一样，此时就可以将这后几位作为散列地址，如手机号
2. 平方取中法：取关键字平方值得中间几位作为散列地址，具体取几位根据实际情况，这样会使散列地址分布较为均匀，产生冲突可能性更低一些，用用于各位数都不均匀的数码，如身份证

处理冲突的方法：

拉链法：java最常用

再散列法（再哈希法）：设计多个散列函数，发生冲突时使用其他的散列函数

开放定址法：散列表空闲位置同时向同义词和非同义词开放，Hi=(H(key)+di)%m

M是散列表表长，因为%m，有可能落在散列函数所浪费的空间；di是增列序列；查找时也是根据Hi；

查找失败时，空位置的比较也算作一次比较，查找到空位置就表明查找失败，所以删除元素不能直接删除，加个标志位，计算ASL失败，每个哈希值查找长度之和/哈希函数值域；

根据di不同，有以下三种方法

1. 线性探测法，di=0,1,2,….,m-1
2. 平方探测法：di=0,1^2,-1^2,2^2,-2^2,….,k^2,-k^2 k<=m/2 解决线性探测法堆积的冲突较多问题
3. 伪随机序列法：di都是伪随机数

③栈、队列

栈的应用：后缀表达式

准备：

中缀/后缀表达式：zhong hou

优先级对象：属性为运算符，用数字表示优先级

辅助栈1（中缀转后缀）

辅助栈2（后缀计算）

过程：

（1）中缀转后缀（注意加入空格的四个地方），

1、遍历zhong，每个元素为i，若i：

\*为非运算符，直接加入到hou

\*若为运算符，若i为：

\*‘(’，直接入栈

\* +-\*/等，先加入一个空格到hou,若：

\*若栈空/栈顶为'('/i优先级大于栈顶元素，入栈

\*否则，依次出栈加入到hou并并随后加入一个空格，直到栈空/栈顶为'('/i优先级大于栈顶，此时i入栈，

\*')'，依次出栈先加入一个空格i再加入到hou，直到栈顶为'('结束，删除栈顶的'('

2、遍历完成，依次出栈，先加一个空格i再加入hou

（2）后缀表达式计算：

之前加入空格是为了hou的各个数字和运算符分割开，方便处理。

1、hou根据空格转化成数组

2、遍历数组，若i为：

\*数字，入栈

\*运算符，则出栈两个元素b和a（b比a先出栈），根据运算符符号运算 a运算符b，将结果入栈

3、遍历完成，栈剩下一个元素，就是计算结果

树形结构

树：

总结点数=总分支数+1=n0+n1+n2+….nx nx表示度数为x结点个数

总分支数=0\*n0+1+n1+2\*n2+……x\*nx x\*nx表示度数为x\*个数

树与二叉树之间转化,森林与二叉树之间转化，都是左孩子右兄弟，森林二叉树互相转换石森林的每棵树的根结点看成兄弟

树的遍历：层序，先根（相当于二叉树先序思想），后根（相当于二叉树后序思想）

森林的遍历：先序=每颗树的先根，中序=每颗树的后根

将树或森林转化成二叉树，树的先根/后根序列=转化的二叉树先序/中序序列，森林的先序/中序序列=转化后的二叉树的先序/中序序列

二叉树：

叶子结点数=度为2结点数+1 空分支数=总结点数+1

满二叉树：总结点数=2^n-1 第n层结点数=2^(n-1)

完全二叉树：除了最后一层，其他是满二叉树，完全二叉树最后一层左边必须是满的

层数=log2 (x)向下取整+1=log2 (x+1)向下取整 x为总结点数

二叉树存储结构：

顺序结构：只适用于完全二叉树，不是完全二叉树要填充0形成完全二叉树

存放到一维数组中：i的左右孩子分别在

数组0开始：左=2i+1 有=2i+2

数组1开始：左=2i 有=2i+1

链式结构：

广度优先遍历（程序遍历）：利用一个队列

1.根结点入队 2.出队，并查看是否有子树，有则3. 入队，然后重复2

深度优先遍历（先序，中序，后序）

先序: 中序： 后序：

cout<<T->data; visit(T->leftChild); visit(T->leftChild);

visit(T->leftChild); cout<<T->data; visit(T->rightChild);

visit(T->rightChild); visit(T->rightChild); cout<<T->data;

一个先序/后序序列，和一个中序序列能唯一确定一颗二叉树

1.先序/后序找根结点 2.中序找左右子树 3.重复1

利用栈可以实现非递归的先序/中序/后序遍历

二叉树应用：

堆：分为大根堆和小根堆

用树的顺序结构存储，所以堆一定是完全二叉树，大根堆/小根堆的每个根结点都大于/小于它的孩子，而孩子没有次序之分

堆的插入：由于是完全二叉树，每次都是从上至下，从左至右，插入后，检查它与它的父结点大小关系并处理，若发生了交换变成了父结点，则它继续和它的父对比处理，每次都是一次关键字对比

堆的删除：删除一个元素后，将堆的最后一个元素替换到删除位置，检查它与它的孩子大小关系并处理，若发生交换变成了孩子，要继续检查它的孩子，每次都是孩子数量的关键字对比，这个过程下坠

线索二叉树：分为先序/中序/后序线索二叉树，将二叉树空指针（空链域）指向前驱后继，成为线索，左孩子指向前驱，右孩子指向后继，结构体增加ltag，rtag标志位，0时是孩子，1时是线索

创建先序线索二叉树要注意防止闭环

线索二叉树找前驱后继：

利用线索可以实现另一种先序/中序遍历

中序线索二叉树：

深度优先遍历时，需要在递归前加上约束，if(ltag==0) if(rtag==0)

二叉排序树（也叫二叉查找树，BST），二叉平衡树（AVL树）：

二叉排序树：默认左<根<右,不能有相等的数值

中序遍历得到有序序列

BST的平均查找长度ASL：需要比较的次数平均值

例子：B<-A->C

查找成功的ASL：(1\*1+2\*2)/3=5/3

查找失败的ASL：看空指针，上例有4个空指针，每个要比较两次，(2\*4)/4

平衡二叉树的平均查找长度最短

查找时间复杂度：最好O(logn)，最坏O(n)

BST删除节点：

1. 删除的结点无孩子：直接删除
2. 删除的结点有且只有一个左孩子或右孩子：删除结点，并将它的孩子替换到删除的位置
3. 删除的结点有两个孩子：两种方案
4. 前驱替换：前驱在它的左子树的最右边，先用规则①②删除前驱，再用前驱替换删除的结点
5. 后继替换：后继在它的柚子树的最左边，先用规则①②删除后继，再用后继替换删除的结点

**BST可以在结构体中增加计数变量用来统计相等值的个数，用来统计词频**

AVL树：LL,LR,RL,RR，查找时间复杂度：O(logn)

平衡因子：左子树高度-有子树高度，平衡因子的绝对值小于等于1的BST就是AVL

最小不平衡子树：插入结点出发，第一个不平衡的子树，只要它平衡了，整棵树就平衡了

平衡方法：先找出最小不平衡子树t

简便方法： 从整棵树的根结点C出发，到失去平衡的插入结点路径的前三个结点C,B,A。这三个结点在中序序列中处于中间位置的结点作为最小不平衡子树的根结点

第一个L/R：左/右孩子，第二个L/R：左/右子树，最小不平衡子树的根结点用t表示

LL：右旋：t的左孩子a右旋成为根结点，若a原来有右孩子c，则c成为t的左孩子

LR：先左旋，后右旋：

1. t的左孩子a的右孩子b左旋，b成为c左孩子，若b有左孩子c，则c成为a的右孩子
2. b右旋成为根结点，若b有右孩子d，则d成为t左孩子

RL：先右旋，后左旋：

1. t的右孩子a的左孩子b右旋，b成为c右孩子，若b有右孩子c，则c成为a的左孩子
2. b左旋成为根结点，若b有左孩子d，则d成为t右孩子

RR；左旋：t的右孩子a左旋成为根结点，若a原来有左孩子c，则c成为t的右孩子

B树：

是一个m叉排序树，分叉数称为B树的阶即m阶B树，可递减递增，每个结点最多有m个分叉，有分叉数-1个关键字分成分支数个区间，特别注意B树最后一层空链域是叶子结点，表示查找失败，倒数第二层是终端结点，讨论B树的高度不包括叶子结点

因为每个结点中的关键字是有序的，也可用折半查找结点内关键字

B树的必要条件：

1. 除了根结点外每个非叶子结点中分支数最少有m/2向上取整个，这是为了减少B树高度提高查找效率
2. 每颗子树高度差为0，与AVL高度差小于1不一样是为了简化代码实现

满足这两个条件才是B树，B树也叫多路平衡查找树

因为B树绝对平衡（高度差=0），所以根结点不是终端结点时至少有两颗子树，叶子结点（空链域）都在最后一层

性质：

n个元素的B树，叶子结点数（空链域数）=n+1

n个元素的B树高度（不包括叶子结点层）：

最小（每个结点都有m个分叉，m-1个关键字）：

n<=(m-1)(1+m+m2+m3+…..m^(h-1))=m^h-1

h>=log(m底)(n+1)

最大（每个结点有m/2向上取整个分叉）：

n>=1+2((m/2)^(h-1)-1)

h<=log(m/2底)((n+1)/2)+1

因此：

log(m底)(n+1)<=h<= log(m/2底)((n+1)/2)+1

B树的插入删除：

插入：

1. 每次插入都是插入到终端结点中
2. 当插入的结点关键字溢出时（即m=关键字数），将第m/2向上取整个关键字（特别注意计算时已经插入了元素）插入到父结点（若没有父结点则创建一个，当插入到父结点也溢出出，重复分裂操作）
3. 将m/2向上取整的左右部分分裂成两个孩子

删除：

1. 若删除的元素在终端结点中且删除后该结点依然有m/2向上取整-1个关键字，直接删除
2. 删除的元素在非终端结点，找出直接前驱或直接后继，若用直接前驱/直接后继替换掉删除结点且删除后所在终端结点依然有m/2向上取整-1个关键字，直接删除替换，反之则要加上③的处理
3. 删除的元素在终端结点且删除后该结点元素小于m/2向上取整-1个

1.若左兄弟够借，设删除元素的结点为a，兄弟结点为b，父结点为c

将父结点c中的a的前驱插入到a中，将b中的a的前驱的前驱插入到父结点c中

2.若右兄弟够借，设删除元素的结点为a，兄弟结点为b，父结点为c

将父结点c中的a的后继插入到a中，将b中的a的后继的后继插入到父结点c中

3.左右兄弟都不够借，设删除元素的结点为a，一个兄弟结点为b，父结点为c

将c中在排在a，b中间的那个元素和a，b合并成新结点，合并后c可能非法，再根据③处理c

B+树：结构类似于分块查找

m阶B+树每个结点最多有m个分支，最少有m/2向上取整分支

非叶根结点至少有两颗子树，每颗子树高度差=0

与B树不同，叶子结点就是树定义保存数据的叶子结点

与B树不同，非叶子结点中，关键字数=分支数

非叶子结点的关键字存储的是每个分支的孩子中的数据最大值且是有序的

叶子结点保存了所有的关键字（就是实际数据的指针），和指向对应记录的指针，所有叶子结点是链表链接起来，且是有序的，并有一个指针用于顺序查找叶子结点

与B树不同，只有走到叶子结点才能真正的查找结束，非叶结点只是提供索引

B+树查找是分块查找，每个结点都是有序的在结点内可以折半查找

因为叶子结点是有序链表，也可以直接在叶子结点比较高效的顺序查找

哈夫曼树（最优二叉树）：

结点的全脂：具有某种现实意义的数值，数值越大越重要

结点的带权路径长度：根结点到该结点路径上的边数\*权值

树的带权路径长度（WPL）：所有叶子结点的带权路径长度之和

哈夫曼树就是WPL最短的二叉树

由叶子结点构建哈夫曼树：

每次找权值最小的两个结点构建出二叉树，其根结点权值是它们的和，再将此根结点与剩余未使用的其他根结点叶子结点构建，重复操作，得到哈夫曼树后只保留叶子结点的权值

哈夫曼树性质：

没有度为1结点，总结点数2n-1，权值越大的叶子结点边数越少（越重要）

哈夫曼编码：是前缀编码（任何一个码都不是另一个码的前缀）

从根结点出发到一个叶子结点，每经过左边的边编码0，右边的边编码1，经过x条边，编码就有x位，每个叶子结点的编码位数有长有短

树剩余知识：非递归遍历二叉树和二叉树性质，树和森林存储遍历，二叉线索树前驱后继，AVL旋转代码和高度计算，哈夫曼树代码

图结构：

基本概念：

无权图，每个边都没有数值；带权图（也叫网），每条边都有权值

简单图，无重复边且无自己到自己得到环；反之是多重图

连通图，每个顶点连在一起；非连通图，部分顶点没有连在一个图中，形成n个独立的图；

强连通图，从任意顶点出发都有到各个顶点的路径；连通分量，非连通子图的数量，连通图为1

存储结构：

存储无向图有向图均可：邻接矩阵，邻接表，邻接矩阵空间复杂度高存放稀疏图会很浪费但存储结构唯一，邻接表存储有向边时找入度麻烦需要遍历整个表，邻接表存储结构不唯一

邻接矩阵：一维数组（存储结点）+二维数组（存储边），空间复杂度O(n2)，无权图用0或False，1或True表示无边，有边；带权图中，有边的地方存放权值，无边的地方用无穷符号表示（代码中可以#define常量定义无穷，如int的最大值），主对角线可以用0或者无穷表示

无向图邻接矩阵：因为是无向图所以a[i][j]=a[j][i]是一个对称矩阵，遍历每行或每列均可得到度数

有向图邻接矩阵：行表示出度顶点，列表示入度顶点，遍历行得到出度数，遍历列得到入度数

邻接矩阵A，A的n次方[i][j]的值表示从i到j路径上的边数为n的数量化

邻接表：一维结构体数组（存放出度顶点）+每个数组元素对应的链表，表示该数组元素到每个链表结点存储的顶点有连接

空间复杂度：无向图：O(V+2E)；有向图：O(V+E)

无向图遍历每个链表得到度数，有向图遍历每个链表得到出度数，遍历整个表得到入度数

解决了邻接矩阵空间复杂度高的问题，但存储无向图仍然需要存储两次，也带来了存储有向图找入度时需要遍历整个表的问题

只能存放有向图：十字链表

对邻接表进行优化，结构体增加一个指针指向其他有当前数组下标表示的顶点的链表，遍历该指正即可找到入度，解决了有向图邻接表找入度麻烦的问题

只能存放无向图：邻接多重表

解决了无向图邻接表每条边存放两次的冗余问题

图的遍历：广度优先搜索BFS，深度优先搜索DFS

邻接矩阵和邻接表的BFS/DFS比较抽象，但单纯的二维矩阵就好理解了

BFS：与树的层序遍历类似，只是要加上一个布尔数组标记已访问的结点，非连通图需要每次BFS后遍历布尔数组找出第一个False继续BFS

空间复杂度：辅助队列O(V)，时间复杂度：邻接矩阵O(V2)，邻接表O(V+E)

从一个顶点出发，邻接矩阵遍历序列唯一，邻接表因为本身不唯一，所以遍历序列不唯一，根据邻接表的存储边的顺序

广度优先生成树：根据BFS形成的树

广度优先生成森林：非连通图BFS形成的森林

DFS：与树的先根遍历类似，只是要加上一个布尔数组，非连通图每次DFS后需要遍历布尔数组找出第一个False继续DFS

时间复杂度与BFS一样

从一个顶点出发，邻接矩阵遍历序列唯一，邻接表因为本身不唯一，所以遍历序列不唯一，根据邻接表的存储边的顺序

遍历需要的BFS/DFS次数：无向图BFS/DFS次数=连通分量

有向图比较复杂，只要注意强连通图或者能从一个顶点找到其他顶点的顶点出发，次数就是一次

图的应用：最短路径，最小生成树，DAG图描述表达式，拓扑排序，关键路径

最短路径：三种算法都不能用于有负权值环路的图

单源最短路径：一个源头到其他顶点的最短路径

无权图：BFS，用dist数组记录路径长度（没有路径记无穷），path数组记录直接前驱（没有前驱记-1），在path中找路径结点

带权图无权图均可：Dijkstra算法（迪杰斯特拉算法）：

不能用于有负权值的图

时间复杂度O(V2)

用三个数组，final数组记录是否已经找到最短路径bool值，dist数组记录目前能找到的最短路径长度（没有路径记无穷），path记录直接前驱（没有前驱记-1）

过程：

1. 将源点的final设为true，将其相连顶点dist设为边权值
2. 找到final为false且dist最小的顶点，将其final设为true，path设置前驱为上一个true顶点
3. 计算final为false的顶点的相连顶点的dist，并且判断是否通过2得来的顶点权值能更小，若能则修改
4. 重复2和3

在path中找路径结点

所有顶点两两之间的最短路径：

Floyd算法（弗洛伊德算法）：动态规划

可以用于有负权值的图

空间复杂度O(V2)，时间复杂度O(V3)

用到两个矩阵，一个记录最短路径初始化为邻接矩阵相同，一个记录中转点（没有中转点记-1）

每轮增加一个中转点，若能通过该中转点得到更短路径，修改最短路径并记录中转点

中转点矩阵找路径结点时，用到递归

最小生成树：MST

生成树：无向连通图中，各个顶点都连通不构成环，且边是顶点数-1,的子图就是生成树，可能有多个生成树，连通图才有生成树，非连通图是生成森林

最小生成树：带权无向连通图的生成树权值之和最小的生成树，可能有多个最小生成树，若图本就是一棵树，最小生成树就是它自己

两个算法：因为可能有权值相同的边，所以MST可能不唯一

Prim算法（普里姆算法）：从某个顶点开始构建生成树，每次找出与生成树连接权值最小（权值相等就任选）的顶点

实现过程：用到一个布尔数组join和一个存放权值的数组cost，join存储某顶点是否参与构建生成树，cost是生成树（注意是生成树的所有顶点）到某顶点的最小权值

1. 从某个顶点出发，join相应的位置设为true，cost的值设为到该顶点的权值，没有连接则设为无穷，自身到自身的值设为0
2. 遍历cost，从cost找出值最小且不为0的顶点，相应的join设为true
3. 遍历cost，修改值，值为生成树到该顶点的最小权值
4. 重复2和3

Kruskal算法（克鲁斯卡尔算法）：每次从剩余的边中，找出权值最小（权值相等就任选）且不会构成环的边

实现过程：用到并查集，判断顶点是否在一个集合中就能判断新加入的边是否会构成环

合并集合的过程：集合a有1和2，集合b有2和5，a和b就合并成新集合c有1,2,5

1. 对边权值排序
2. 找出权值最小且连接的两个顶点不在同一个集合中的边构建生成树，删除该边，并将这两个顶点构建集合或合并到一个集合
3. 重复1和2

Prim是找顶点，适合用于边稠密图，时间复杂度O(V2)；Kruskal是找边，适合用于边稀疏图，时间复杂度O(E\*logE)

DAG和描述表达式：

拓扑排序：

AOV网：顶点表示活动的网，一定是一个DAG

一个活动完成才能进行下一个活动，拓扑排序可能有多个

手算：每次选入度为0的顶点，可以画一个矩阵，每一行写每一步的入度

代码：

关键路径：

AOE网：边表示活动的网，边的权值代表时间开销，AOE网一定是DAG

顶点表示瞬间事件，只有入度边的所有活动完成才能开始此事件；只有该事件完成才能开始出度活动，出度活动可以并行

0入度顶点为开始顶点，0出度顶点为结束顶点

关键路径就是从开始到结束权值之和最大的路径，关键路径上的活动称为关键活动

1. 增加关键活动的时间会增加整个工程时间
2. 减小关键活动时间会减少整个活动时间，但关键活动的时间减少到一定程度，可能会变成非关键活动
3. 有多条关键路径的工程，要减少整个工程时间，一是多条关键路径一起减少时间，二是减少共有的关键活动的时间

关键路径求法：

手算：权值之和最大的路径

代码：时间余量为0的活动是关键活动

事件的：

最早开始时间VE：从开始顶点开始算，弧头VE=弧尾VE+边权值（开始顶点VE=0）

最晚开始时间VL：从结束顶点开始算，弧尾VL=弧头VL-边权值（结束顶点VL=VE）

活动的：

最早开始时间EE：=弧尾VE

最晚开始时间EL：=弧头VL-权值

时间余量：=EL-EE

查找算法：

顺序/折半/分块查找，散列查找，KMP算法，BST/AVL/B树/B+树，BFS/DFS

基本概念：

查找表：被查的结构，包括线性/树/图，查找一般有两种操作，一是查找，此时是静态查找表；二是插入/删除，此时是动态查找表

关键字：能唯一确定查找表的数据元素

某个数据元素查找长度：查找需要对比的次数

平均查找长度ASL：(每个数据元素查找长度\*概率)之和，*默认查找每个元素的概率一样相当于求平均值，若概率不同，则需要\*各自概率*，分为ASL成功，ASL失败

顺序查找：时间复杂度O(n)

通常用于线性表（顺序表/链表）

查找判定树：只有右孩子的二叉树，左孩子空链域表示查找失败

ASL成功：(1+2+…..n)n=(n+1)/2

ASL失败：n，若是有序查找表ASL失败可以优化(1+2+…..n)n=(n+1)/2，，若是有概率的查找表，可以优化，按概率降序

折半查找（二分查找）：时间复杂度O(logn)

只能用于有序顺序表

查找判定树：是一个平衡二叉树且是完全二叉树具有一样的性质，因为(low+high)/2是向下取整，所以判定树根结点右子树结点数大于(查找表偶数个元素)等于(查找表奇数个元素)左子树

查找判定树画法：每个结点都是每一步的mid，查找表奇数个元素时每次左子树结点数=右子树结点数，偶数个时，每次左+1=右

分块查找：

块内无序，块间有序

查找表分为n个分块，并建立一个索引表，索引表的每个元素是每个分块的最大值关键字，索引表每个元素都有一个闭区间，表示该块在查找表的下标范围

查找过程：

先在索引表中折半查找找到所在的分块，再根据闭区间到查找表相应的分块中顺序查找

求ASL比较复杂，一般考特殊情况：

n个元素的查找表，分为b个块，每块s个元素，n=sb

用顺序查找索引表：b=n/s,ASL=(b+1)/2+(s+1)/2=s/2+1+n/2s，对ASL求导求ASL最小值：当s=根号n，ASL是最小值 (根号n)+1

用折半查找索引表：log(b+1)向上取整+(s+1)/2

排序算法：

指标：时空复杂度，稳定性，磁盘读取次数（外部排序特有）

平均时间复杂度：（最好+最坏）/2

稳定性：若有相同的关键字，排序后相对位置不变则是稳定的

内部排序：所有数据直接在内存排序

外部排序：数据量太大，分几次放到内存排序

以下算法升序描述

三种插入排序：

直接插入排序：

思想：

左表有序，右表无序，将右表元素依次插入左表相应位置，插入的位置右边元素右移

过程：

1. 从第二个元素开始到最后一个元素，共n-1趟
2. 只有待排序元素小于前一个元素，才需要插入，否则已是有序
3. 插入：从前一个元素递减到下标0找到待排序元素的插入位置，右边元素右移

每一趟是待排序元素插入到相应位置

性能：

时间复杂度：主要开销在于移动元素，所以更适合链式结构，这里分析的是顺序结构

最好（原本已经有序）：O(n) 最坏（原本逆序：O(n2) 平均：O(n2)

原本越有序越接近O(n)

空间复杂度：O(1)

稳定性：稳定

适用：

顺序链式都可以，更适合链式

折半插入排序：优化直插，用折半查找找到插入位置，但却不适用于链式结构

过程：直插的2.3中间加入一个折半查找，插入位置为high+1，3的for循环条件修改为j>=high+1

每一趟和直插一样

时空复杂度，稳定性也一样

不适用于链式结构

希尔排序：

思想：

优化直插，利用直插越有序效率越高的特点

每一趟按增量d划分子表，每一趟缩小增量，直到d=1为止

I+d，i+2d，i+3d……为一个子表，d=1时只有一个子表就是整个表

增量序列d一般是n/2，n/2/2…直到1每次除以2，也可采用其他序列

划分子表后对每个子表交替进行直接插入排序，每一趟结束会得到比前一趟更有序的表

过程：

1. for缩小增量，表示每一趟
2. 从每个子表第二个元素开始，for (i = d; i < n; i++) 交替处理各个子表
3. 对各个子表直接插入排序

每一趟是各个子表都排好序

性能：

时间复杂度：取决于d，目前没有准确的值，n在一定范围内能达到O(n1.3)，但当d只有1时退化成直插，

空间复杂度：O(1)

稳定性：不稳定

适用：

只适用顺序结构

两种交换排序；

冒泡排序：

思想：

相邻对比，每一趟把最大或最小放到最终位置，下一趟忽略已排好序的元素，设置一个布尔值若排序过程中没有交换元素，说明整个表有序，提前结束

过程：

1. 从第一个元素到倒数第二个元素，共n-1趟
2. 忽略已排好序元素，从后往前冒泡/从前往后（j>i/j<i），依次对比两个相邻元素，最终把最小值/最大值放到最终位置，若有交换元素，布尔值设为true
3. 一趟结束，若布尔值为false’，则结束排序，否则继续

每一趟是把最值放到最终位置

性能：

时间复杂度：最好（原本已经有序）：O(n) 最坏（原本逆序：O(n2) 平均：O(n2)

原本越有序越接近O(n)

空间复杂度：O(1)

稳定性：稳定

适用：

顺序链式都可以

快速排序：

思想：

每一趟选择一个枢轴（基准）元素，把小于/大于枢轴元素全部放到左边/右边，设置low/high指向第一个/最后一个元素

第一步low不动，若high指向元素大于枢轴元素，high—直到high的元素小于枢轴元素，a[low]=a[high]；

下一步high不动，若low指向元素小于枢轴元素，lo++直到low的元素大于枢轴元素，a[high]=a[low]；

如此往复，直到low==high则是枢轴元素的最终位置，这样就完成了一次划分，再递归左右子表

过程：

1. 当low<high时，用patition()进行划分并返回枢轴下标，再根据枢轴递归左右子表
2. patition()的实现：while(low<high)时，找到第一个high元素小于枢轴的元素，a[low] = a[high];再找到第一个low大于枢轴的元素，a[high] = a[low];while结束时low==high即是枢轴最终位置，返回枢轴下标

每一趟是各个子表的枢轴都找到最终位置

性能：

快排n个元素递归二叉树的深度是：logn向下取整+1<h<n

时间复杂度：每一趟O(n\*递归深度)=最好O(nlogn) 最坏O(n2)，因为枢轴选取的是 每个子表的第一个元素，此时若表原来有序或逆序就是最坏情况，可以优化枢轴选取，1.每个子表的第一个，中间，最后一个元素，选取值在中间的元素作为枢轴，2.也可以锁机选取枢轴

空间复杂度：就是递归树的深度

最好O(logn) 最坏O(n)

稳定性：不稳定

适用：

顺序链式都可以

两种选择排序：

简单选择排序：

思想：每一趟把最小值或最大值放到左边/右边

过程：

1. 遍历整个表
2. min=i，若有更小的，min=j
3. min==i已是最小值不处理，min!=i，则swap(a[min],a[i])

每一趟是最值放到最终位置

性能：

时间复杂度：任何情况都是O(n2)

空间复杂度：O(1)

稳定性;不稳定

适用：

顺序链式都可以

堆排序

思想：

将数组调整成大根堆或小根堆，每次能很快选出最值，每一趟都要忽略已有序元素重新调整堆

因为每次决定元素最终位置都是在后面，所以基于大根堆得到升序序列，基于小根堆得到降序序列

过程;

1. 1.建堆2.从最后一个元素开始到第一个元素，共n-1趟 3.将大根堆最大值放到I 4.传入根结点0调整堆 5.重复3,4
2. 建堆过程，从最后一个非叶子结点（(n - 2) / 2）开始到根结点，传入i调整堆
3. 调整堆过程，传入的结点为k，检查k左右孩子大小并处理，若有交换则要k=孩子下标，以便将k调整到最终位置，继续检查孩子的孩子大小关系，

每一趟是最值放到最终位置，调整好堆

性能：

时间复杂度：建堆n，n-1趟排序，调整堆logn，n+nlogn=O(nlogn)

空间复杂度：O(1)

稳定性：不稳定

适用：

只适用顺序结构，因为堆是树的顺序结构

归并排序：

思想：

每次归并的子表数，就是n路归并，一般用二路归并，需要一个辅助数组b

先把表递归左右划分子表直到每个子表只有一个元素（low==high）为止，再每一趟将两个子表合并成有序子表

low指向左子表第一个元素，mid指向左子表最后一个元素，high指向右子表最后一个元素

过程：

1. 递归当low<high，递归划分左右子表，然后合并
2. 合并过程，先给辅助数组赋值，用i，j分别指向辅助数组左右子表，对比左右子表，修改相应a数组的值；最后检查是否还有未处理的元素

每一趟是每一对左右子表都完成排序合并

性能：

时间复杂度：排序n，归并树深度logn，所以=O(nlogn)

空间复杂度：主要是辅助数组n，递归logn相对n少，所以O(n)

稳定性：稳定

使用：

顺序链式都可以

基数排序：唯一不需要对比关键字的排序

将数码拆分d元组（如千百十个），每一位起取值范围是0到r-1，r称为基数，十进制数的r=10，需要r个辅助队列，按权重从小到大分配和收集（十进制个位权重最小），共d趟

过程：

1. 分配：顺序扫描元素，元素的基数为x入队到队列q[x]中
2. 收集：从q[0]到q[r-1]依次出队，若一个队列有多个元素，先将所有元素出队

重复1，2，每一趟是分配收集完，共d趟

性能：分析链式结构

时间复杂度：分配O(n)，收集O(r)（链式结构），d趟，所以=O(d(n+r))

空间复杂度：r个辅助队列，O(r)

稳定性：稳定

适用：

顺序链式都可以，更适合链式，d，r应较小，n应较大

外部排序：

外存存储数据是以磁盘块为单位的，每次读写也是以磁盘块为单位，外部排序基于归并排序，这样内存中最少只需要申请三块磁盘块大小的区域就能完成排序，两个在输入缓冲区，一个在输出缓冲区，写回外存的是另一片区域，原区域被系统回收

1. 生成初始归并段：输入缓冲区两个磁盘块合在一起内部排序后通过输出缓冲区写回外存即可，写回的两个磁盘块共称为归并段
2. 将外存的两个归并段中分别最小的磁盘块读入内存进行归并排序，输出缓冲区一满立即写回外存，输入缓冲区1/2一满立即读入外存归并段1/2，如此往复过程2，形成比原来大一倍的各归并段

时间开销：读写时间（最主要）+过程1内部排序+过程2归并排序

优化1：增加归并排序路数，就能减少归并趟数，进而减少读写次数减少外部排序时间

k路归并树是k叉树，归并趟数=h-1=log(k底)r向上取整，r为初始归并段数

k不是越大越好，k越大，内存空间开销越大，内部排序时间也越大

优化2：减少归并段数量，增加归并段长度能减少归并段数量

败者树：

优化k路内部归并排序选出最值元素的关键字对比次数

1. 构建败者树，叶子结点为待比较元素，每个非叶子结点记录比较后失败者的位置，根结点保存最后失败的结点位置，根结点再连上一个父结点，记录最值（即成功）的元素位置，构建败者树后，非叶子节点不再修改，共需要n-1次关键字对比
2. 1选出的冠军磁盘块后一个元素替代冠军源叶子结点位置，依次与祖先败者结点对比，只需要对比logk向上取整次关键字即可再选出最值

由此可见，只要对比n-1次关键字构建败者树，以后每次选出最值只需要对比logk向上取整次关键字

败者树的高度不包括根结点的父结点

置换-选择排序

过程1由于输入缓冲区空间有限，只能初始化成每个归并段的长度就会被限制，而置换-选择排序能够在不增加输入缓冲区空间的情况下，增加归并段的长度，这样会使每个归并段长度不同，归并段越长，归并段数量越少，外部排序越快

过程：

1. 每次从输入缓冲区1中选择最小的加入输出缓冲区，并用MINIMAK变量记录该值
2. 从输入缓冲区2中顺序加入一个元素到输入缓冲区1中
3. 输入缓冲区1中选择最小的且值大于MINIMAK的元素加入输出缓冲区，MINIMAK记录该值
4. 重复2.3直到输入缓冲区1的元素都小于MINIMAK，就完成了一个归并段的构建，MINIMAK初始化为0，继续下一个归并段的构建

整个过程中输出缓冲区一满就写入外存，输入缓冲区2一空就读入一个磁盘块内存，结束后每个归并段长度会有不同

最佳归并树：

由置换-选择排序构建的每个归并段会有不同，把每个归并段所占的磁盘块数量作为叶子结点权值，由这些叶子结点构成的归并树的带权路径长度\*2就是读写总次数（读+写）

显而易见，最佳归并树就是带权路径长度最小的归并树

二叉最佳归并树就是一个哈夫曼树

k叉归并树构建与哈夫曼树类似，只是每次选择的元素个数为k个

但是k>2时，构建的归并树必须是严格的k叉树（只有度为k和度为0的结点），否则需要加入若干个虚段（虚段权值为0），虚段在外存内存读写中逻辑上是读写实际上不读写

具体加入几个虚段，由：

总结点数=0度结点数+k度结点数

总分支数=总结点数-1

0度结点数=初始归并段数+虚段数

得到：(初始归并段数+虚段数-1)%(k-1)=0 即可求得虚段数，虚段数==0则不需要加虚段