1. 什么是数据结构？ 数据结构是一门研究非数值计算的程序设计问题中的操作对象，以及它们之间的关系和操作等相关问题的学科. 程序设计 = 数据结构 + 算法 传统上，我们把数据结构分为逻辑结构和物理结构。 逻辑结构：是指数据对象中数据元素之间的相互关系，也是我们今后最需要关注和讨论的问题。 { 集合结构(仅仅同属一个集合),线性结构（一对一的关系），　树形结构（一对多的层次关系），　图形结构（多对多的关系） } 物理结构：是指数据的逻辑结构在计算机中的存储形式。 ｛ 顺序存储：是把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的。例如我们编程语言的数组结构就是这样滴。 链式存储：是把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的。（因此需要用一个指针存放数据元素的地址，这样子通过地址就可以找到相关联数据元素的位置。） ｝
2. 算法 算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。
* 算法的特征：输入、输出、有穷性、确定性和可行性。 输入：算法具有零个或多个输入。 输出：算法至少有一个或多个输出。 有穷性：指算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成。 确定性：算法的每一个步骤都具有确定的含义，不会出现二义性。 可行性：算法的每一步都必须是可行的，也就是说，每一步都能够通过执行有限次数完成。
* 算法设计的要求： 正确性：算法的正确性是指算法至少应该具有输入、输出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案。 大体分为以下四个层次： 算法程序没有语法错误。 算法程序对于合法输入能够产生满足要求的输出。 算法程序对于非法输入能够产生满足规格的说明。 算法程序对于故意刁难的测试输入都有满足要求的输出结果。 可读性：算法设计另一目的是为了便于阅读、理解和交流。 健壮性：当输入数据不合法时，算法也能做出相关处理，而不是产生异常、崩溃或莫名其妙的结果。 时间效率高和存储量低： ３. 时间复杂度和空间复杂度
* int i, sum = 0, n = 100; // 执行1次
for( i=1; i <= n; i++ ) // 执行了n+1次
{
 sum = sum + i; // 执行n次
}
* int sum = 0, n = 100; // 执行1次
sum = (1+n)\*n/2; // 执行1次
* 第一种算法执行了1+(n+1)+n=2n+2次。 第二种算法，是1+1=2次 我们研究算法的复杂度，侧重的是研究算法随着输入规模扩大增长量的一个抽象，而不是精确地定位需要执行多少次。 我们在分析一个算法的运行时间时，重要的是把基本操作的数量和输入模式关联起来。 函数的渐近增长：给定两个函数f(n)和g(n)，如果存在一个整数N，使得对于所有的n>N，f(n)总是比g(n)大。那么，我们说f(n)的增长渐近快于g(n)。 判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项（最高项）的阶数。 算法时间复杂度的定义：在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况并确定T(n)的数量级。 算法的时间复杂度，也就是算法的时间量度，记作：T(n)= O(f(n))。 它表示随问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中f(n)是问题规模n的某个函数。 这样用大写O()来体现算法时间复杂度的记法，我们称之为大O记法。 一般情况下，随着输入规模n的增大，T(n)增长最慢的算法为最优算法。 显然，由此算法时间复杂度的定义可知，我们的三个求和算法的时间复杂度分别为O(1)，O(n)，O(n^2)。
* 推导大O阶方法: 用常数1取代运行时间中的所有加法常数。 在修改后的运行次数函数中，只保留最高阶项。 如果最高阶项存在且不是1，则去除与这个项相乘的常数。 得到的最后结果就是大O阶。 线性阶:一般含有非嵌套循环涉及线性阶，线性阶就是随着问题规模n的扩大，对应计算次数呈直线增长。 平方阶: 对数阶: 在应用中，这是一种最重要的需求，通常除非特别指定，我们提到的运行时间都是最坏情况的运行时间。
* 算法的空间复杂度: 算法的空间复杂度通过计算算法所需的存储空间实现，算法的空间复杂度的计算公式记作：S(n)=O(f(n))，其中，n为问题的规模，f(n)为语句关于n所占存储空间的函数。
1. 线性表 线性表（List）：由零个或多个数据元素组成的有限序列。 首先它是一个序列，也就是说元素之间是有个先来后到的 若元素存在多个，则第一个元素无前驱，而最后一个元素无后继，其他元素都有且只有一个前驱和后继。 另外，线性表强调是有限的，事实上无论计算机发展到多强大，它所处理的元素都是有限的。 若将线性表记为（a1,…,ai-1,ai,ai+1,…an）,则表中ai-1领先于ai,ai领先于ai+1,称ai-1是ai的直接前驱元素,ai+1是ai的直接后继元素。 所以线性表元素的个数n(n>=0)定义为线性表的长度，当n=0时，称为空表。
* 第一个元素无前驱，而最后一个元素无后继，其他元素都有且只有一个前驱和后继。
* 数据类型：是指一组性质相同的值的集合及定义在此集合上的一些操作的总称。 { 原子类型：不可以再分解的基本类型，例如整型、浮点型、字符型等。 结构类型：由若干个类型组合而成，是可以再分解的，例如整型数组是由若干整型数据组成的。 } 抽象：是指抽取出事物具有的普遍性的本质。它要求抽出问题的特征而忽略非本质的细节，是对具体事物的一个概括。抽象是一种思考问题的方式，它隐藏了繁杂的细节。 我们对已有的数据类型进行抽象，就有了抽象数据类型。 抽象数据类型（Abstract Data Type，ADT）是指一个数学模型及定义在该模型上的一组操作。 抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关。 “抽象”的意义在于数据类型的数学抽象特性。 描述抽象数据类型的标准格式： ADT 抽象数据类型名 Data 数据元素之间逻辑关系的定义 Operation 操作 endADT
* 所谓抽象数据类型就是把数据类型和相关操作捆绑在一起。 总结下，顺序存储结构封装需要三个属性: 存储空间的起始位置，数组data，它的存储位置就是线性表存储空间的存储位置。 线性表的最大存储容量：数组的长度MaxSize。 线性表的当前长度：length。
* 数组的长度是存放线性表的存储空间的总长度，一般初始化后不变。 而线性表的当前长度是线性表中元素的个数，是会变化的。
* 线性表顺序存储结构的优缺点: 线性表的顺序存储结构，在存、读数据时，不管是哪个位置，时间复杂度都是O(1)。而在插入或删除时，时间复杂度都是O(n)。 优点： 无须为表示表中元素之间的逻辑关系而增加额外的存储空间。 可以快速地存取表中任意位置的元素。 缺点： 插入和删除操作需要移动大量元素。 当线性表长度变化较大时，难以确定存储空间的容量。 容易造成存储空间的“碎片”。
* 线性表链式存储结构定义: 线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素，这组存储单元可以存在内存中未被占用的任意位置。比起顺序存储结构每个数据元素只需要存储一个位置就可以了。 单链表: 现在链式存储结构中，除了要存储数据元素信息外，还要存储它的后继元素的存储地址（指针）。 也就是说除了存储其本身的信息外，还需存储一个指示其直接后继的存储位置的信息。 我们把存储数据元素信息的域称为数据域，把存储直接后继位置的域称为指针域。 指针域中存储的信息称为指针或链。 这两部分信息组成数据元素称为存储映像，称为结点(Node)。 n个结点链接成一个链表，即为线性表(a1, a2, a3, …, an)的链式存储结构。 因为此链表的每个结点中只包含一个指针域，所以叫做单链表。 我们把链表中的第一个结点的存储位置叫做头指针，最后一个结点指针为空(NULL)。
* 头指针是指链表指向第一个结点的指针，若链表有头结点，则是指向头结点的指针。 头指针具有标识作用，所以常用头指针冠以链表的名字（指针变量的名字）。 无论链表是否为空，头指针均不为空。 头指针是链表的必要元素。
* 头结点是为了操作的统一和方便而设立的，放在第一个元素的结点之前，其数据域一般无意义（但也可以用来存放链表的长度）。 有了头结点，对在第一元素结点前插入结点和删除第一结点起操作与其它结点的操作就统一了。 头结点不一定是链表的必须要素。 获得链表第i个数据的算法思路： 声明一个结点p指向链表第一个结点，初始化j从1开始； 当jnext = p->next;p->next = s; 单链表第i个数据插入结点的算法思路： 声明一结点p指向链表头结点，初始化j从1开始； 当j<1时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1； 若到链表末尾p为空，则说明第i个元素不存在； 否则查找成功，在系统中生成一个空结点s； 将数据元素e赋值给s->data； 单链表的插入刚才两个标准语句； 返回成功。 单链表的删除: 可以这样：p->next = p->next->next; 也可以是：q=p->next; p->next=q->next; 单链表第i个数据删除结点的算法思路： 声明结点p指向链表第一个结点，初始化j=1； 当j<1时，就遍历链表，让P的指针向后移动，不断指向下一个结点，j累加1； 若到链表末尾p为空，则说明第i个元素不存在； 否则查找成功，将欲删除结点p->next赋值给q； 单链表的删除标准语句p->next = q->next； 将q结点中的数据赋值给e，作为返回； 释放q结点。 效率PK: 我们最后的环节是效率PK，我们发现无论是单链表插入还是删除算法，它们其实都是由两个部分组成：第一部分就是遍历查找第i个元素，第二部分就是实现插入和删除元素。 从整个算法来说，我们很容易可以推出它们的时间复杂度都是O(n)。 再详细点分析：如果在我们不知道第i个元素的指针位置，单链表数据结构在插入和删除操作上，与线性表的顺序存储结构是没有太大优势的。 但如果，我们希望从第i个位置开始，插入连续10个元素，对于顺序存储结构意味着，每一次插入都需要移动n-i个位置，所以每次都是O(n)。 而单链表，我们只需要在第一次时，找到第i个位置的指针，此时为O(n)，接下来只是简单地通过赋值移动指针而已，时间复杂度都是O(1)。 显然，对于插入或删除数据越频繁的操作，单链表的效率优势就越是明显啦~ 单链表整表创建的算法思路如下： 声明一结点p和计数器变量i； 初始化一空链表L； 让L的头结点的指针指向NULL，即建立一个带头结点的单链表； 循环实现后继结点的赋值和插入。 头插法建立单链表:头插法从一个空表开始，生成新结点，读取数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到结束为止。 简单来说，就是把新加进的元素放在表头后的第一个位置： 先让新节点的next指向头节点之后 然后让表头的next指向新节点 尾插法建立单链表: 头插法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。 就像现实社会我们鄙视插队不遵守纪律的孩子，那编程中我们也可以不这么干，我们可以把思维逆过来：把新结点都插入到最后，这种算法称之为尾插法。
* 单链表整表删除的算法思路如下： 声明结点p和q； 将第一个结点赋值给p，下一结点赋值给q； 循环执行释放p和将q赋值给p的操作；
* 单链表结构与顺序存储结构优缺点: 存储分配方式： 顺序存储结构用一段连续的存储单元依次存储线性表的数据元素。 单链表采用链式存储结构，用一组任意的存储单元存放线性表的元素。 时间性能： 查找 顺序存储结构O(1) 单链表O(n) 插入和删除 顺序存储结构需要平均移动表长一半的元素，时间为O(n) 单链表在计算出某位置的指针后，插入和删除时间仅为O(1) 空间性能： 顺序存储结构需要预分配存储空间，分大了，容易造成空间浪费，分小了，容易发生溢出。 单链表不需要分配存储空间，只要有就可以分配，元素个数也不受限制。 结论： 若线性表需要频繁查找，很少进行插入和删除操作时，宜采用顺序存储结构。 若需要频繁插入和删除时，宜采用单链表结构。 比如说游戏开发中，对于用户注册的个人信息，除了注册时插入数据外，绝大多数情况都是读取，所以应该考虑用顺序存储结构。 而游戏中的玩家的武器或者装备列表，随着玩家的游戏过程中，可能会随时增加或删除，此时再用顺序存储就不太合适了，单链表结构就可以大展拳脚了。 当线性表中的元素个数变化较大或者根本不知道有多大时，最好用单链表结构，这样可以不需要考虑存储空间的大小问题。 而如果事先知道线性表的大致长度，比如一年12个月，一周就是星期一至星期日共七天，这种用顺序存储结构效率会高很多。 快速找到未知长度单链表的中间节点? 既然是面试题就一定有普通方法和高级方法，而高级方法无疑会让面试官大大加分！ 普通的方法很简单，首先遍历一遍单链表以确定单链表的长度L。然后再次从头节点出发循环L/2次找到单链表的中间节点。 算法复杂度为：O(L+L/2)=O(3L/2)。 能否再优化一下这个时间复杂度呢？ 有一个很巧妙的方法：利用快慢指针！ 利用快慢指针原理：设置两个指针*search、*mid都指向单链表的头节点。其中\* search的移动速度是*mid的2倍。当*search指向末尾节点的时候，mid正好就在中间了。这也是标尺的思想。
* 将单链表中终端结点的指针端由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相接的单链表成为单循环链表，简称循环链表。 判断单链表中是否有环 有环的定义是，链表的尾节点指向了链表中的某个节点。 方法一：使用p、q两个指针，p总是向前走，但q每次都从头开始走，对于每个节点，看p走的步数是否和q一样。如图，当p从6走到3时，用了6步，此时若q从head出发，则只需两步就到3，因而步数不等，出现矛盾，存在环。 方法二：使用p、q两个指针，p每次向前走一步，q每次向前走两步，若在某个时候p == q，则存在环。

双向链表的插入操作 插入操作其实并不复杂，不过顺序很重要，千万不能写反了。 s->next = p; s->prior = p->prior; p->prior->next = s; p->prior = s; 双向链表的删除操作 p->prior->next = p->next; p->next->prior = p->prior; free(p); 双向链表相对于单链表来说，是要更复杂一点，每个结点多了一个prior指针，对于插入和删除操作的顺序大家要格外小心。不过，双向链表可以有效提高算法的时间性能，说白了就是用空间来换取时间。 栈是一种重要的线性结构，可以这样讲，栈是前面讲过的线性表的一种具体形式。 清空一个栈 所谓清空一个栈，就是将栈中的元素全部作废，但栈本身物理空间并不发生改变（不是销毁）。 因此我们只要将s->top的内容赋值为s->base即可，这样s->base等于s->top，也就表明这个栈是空的了。 这个原理跟高级格式化只是但单纯地清空文件列表而没有覆盖硬盘的原理是一样的。 ClearStack(sqStack *s) { s->top = s->base; } 销毁一个栈 销毁一个栈与清空一个栈不同，销毁一个栈是要释放掉该栈所占据的物理内存空间，因此不要把销毁一个栈与清空一个栈这两种操作混淆。 DestroyStack(sqStack* s) { int i, len; len = s->stackSize; for( i=0; i < len; i++ ) { free( s->base ); s->base++; } s->base = s->top = NULL; s->stackSize = 0; } 计算栈的当前容量 计算栈的当前容量也就是计算栈中元素的个数，因此只要返回s.top-s.base即可。 注意，栈的最大容量是指该栈占据内存空间的大小，其值是s.stackSize，它与栈的当前容量不是一个概念哦。 int StackLen(sqStack s) { return(s.top – s.base); // 初学者需要重点讲解 } 逆波兰计算 逆波兰表达式 a+b —> a b + a+(b-c) —> a b c – + a+(b-c)*d —> a b c – d*  + a+d*(b-c)—> a d b c –*  + 队列的链式存储结构 我们将队头指针指向链队列的头结点，而队尾指针指向终端结点。（注：头结点不是必要的，但为了方便操作，我们加上了。） 空队列时，front和rear都指向头结点。 创建一个队列 创建一个队列要完成两个任务：一是在内存中创建一个头结点，二是将队列的头指针和尾指针都指向这个生成的头结点，因为此时是空队列。 入队列操作 InsertQueue(LinkQueue \*q, ElemType e) { QueuePtr p;

 p = (QueuePtr)malloc(sizeof(QNode));
 if( p == NULL )
 exit(0);

 p->data = e;
 p->next = NULL;
 q->rear->next = p;
 q->rear = p;
}
出队列操作
出队列操作是将队列中的第一个元素移出，队头指针不发生改变，改变头结点的next指针即可。
DeleteQueue(LinkQueue \*q, ELemType \*e)
{
 QueuePtr p;

 if( q->front == q->rear )
 return;

 p = q->front->next;
 \*e = p->data;
 q->front->next = p->next;

 if( q->rear == p )
 q->rear = q->front;

 free(p);
}
销毁一个队列
由于链队列建立在内存的动态区，因此当一个队列不再有用时应当把它及时销毁掉，以免过多地占用内存空间。
DestroyQueue(LinkQueue \*q)
{
 while( q->front )
 {
 q->rear = q->front->next;
 free( q->front );
 q->front = q->rear;
 }
}
树结构
树(Tree)是n(n>=0)个结点的有限集。当n=0时成为空树，在任意一棵非空树中：
有且仅有一个特定的称为根(Root)的结点；
当n>1时，其余结点可分为m(m>0)个互不相交的有限集T1、T2、…、Tm，其中每一个集合本身又是一棵树，并且称为根的子树(SubTree)。

虽然从概念上很容易理解树，但是有两点还是需要大家注意下：
n>0时，根结点是唯一的，坚决不可能存在多个根结点。
m>0时，子树的个数是没有限制的，但它们互相是一定不会相交的。

结点分类
刚才所有图片中，每一个圈圈我们就称为树的一个结点。
结点拥有的子树数称为结点的度-(Degree)，树的度取树内各结点的度的最大值。
度为0的结点称为叶结点(Leaf)或终端结点；
度不为0的结点称为分支结点或非终端结点，除根结点外，分支结点也称为内部结点。

树中结点的最大层次称为树的深度(Depth)或高度。

二叉树的特点 每个结点最多有两棵子树，所以二叉树中不存在度大于2的结点。（注意：不是都需要两棵子树，而是最多可以是两棵，没有子树或者有一棵子树也都是可以的。） 左子树和右子树是有顺序的，次序不能颠倒。 即使树中某结点只有一棵子树，也要区分它是左子树还是右子树， 二叉树的性质一：在二叉树的第i层上至多有2(i-1)个结点(i>=1) 二叉树的性质二：深度为k的二叉树至多有2k-1个结点(k>=1) 二叉树的性质三：对任何一棵二叉树T，如果其终端结点数为n0，度为2的结点数为n2，则n0=n2+1 首先我们再假设度为1的结点数为n1，则二叉树T的结点总数n=n0+n1+n2 其次我们发现连接数总是等于总结点数n-1，并且等于n1+2*n2 所以n-1=n1+2*n2 所以n0+n1+n2-1=n1+n2+n2 最后n0=n2+1 二叉树的性质四：具有n个结点的完全二叉树的深度为⌊log₂n⌋+1 二叉树的性质五：如果对一棵有n个结点的完全二叉树(其深度为⌊log₂n⌋+1)的结点按层序编号，对任一结点i(1<=i<=n)有以下性质： 如果i = 1，则结点 i 是二叉树的根，无双亲；如果i > 1，则其双亲是结点⌊i/2⌋ 如果2i > n，则结点 i 无做左孩子(结点 i 为叶子结点)；否则其左孩子是结点2i 如果2i+1 > n，则结点 i 无右孩子；否则其右孩子是结点2i+1

 既然顺序存储方式的适用性不强，那么我们就要考虑链式存储结构啦。二叉树的存储按照国际惯例来说一般也是采用链式存储结构的。

二叉树每个结点最多有两个孩子，所以为它设计一个数据域和两个指针域是比较自然的想法，我们称这样的链表叫做二叉链表。
 typedef struct BiTNode
{
 ElemType data;
 struct BiTNode \*lchild, \*rchild;
} BiTNode, \*BiTree;
二叉树的遍历方式可以很多，如果我们限制了从左到右的习惯方式，那么主要就分为一下四种：
前序遍历
 若二叉树为空，则空操作返回，否则先访问根结点，然后前序遍历左子树，再前序遍历右子树。
中序遍历
 若树为空，则空操作返回，否则从根结点开始（注意并不是先访问根结点），中序遍历根结点的左子树，然后是访问根结点，最后中序遍历右子树。
后序遍历
 若树为空，则空操作返回，否则从左到右先叶子后结点的方式遍历访问左右子树，最后访问根结点。
层序遍历
 若树为空，则空操作返回，否则从树的第一层，也就是根结点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

赫夫曼树定义与原理
 我们先把这两棵二叉树简化成叶子结点带权的二叉树（注：树结点间的连线相关的数叫做权，Weight）。
 结点的路径长度：从根结点到该结点的路径上的连接数。
 树的路径长度：树中每个叶子结点的路径长度之和。
 结点带权路径长度：结点的路径长度与结点权值的乘积。
 WPL的值越小，说明构造出来的二叉树性能越优。
名词解释：定长编码，变长编码，前缀码
定长编码：像ASCII编码
变长编码：单个编码的长度不一致，可以根据整体出现频率来调节
前缀码：所谓的前缀码，就是没有任何码字是其他码字的前缀
图的定义
 图（Graph）是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：G(V,E)，其中，G表示一个图，V是图G中顶点的集合，E是图G中边的集合。
对于图的定义，我们需要明确几个注意的地方：
 线性表中我们把数据元素叫元素，树中叫结点，在图中数据元素我们则称之为顶点(Vertex)。
 线性表可以没有数据元素，称为空表，树中可以没有结点，叫做空树，而图结构在咱国内大部分的教材中强调顶点集合V要有穷非空。
 线性表中，相邻的数据元素之间具有线性关系，树结构中，相邻两层的结点具有层次关系，而图结构中，任意两个顶点之间都可能有关系，顶点之间的逻辑关系用边来表示，边集可以是空的。
无向边：若顶点Vi到Vj之间的边没有方向，则称这条边为无向边(Edge)，用无序偶(Vi,Vj)来表示。
有向边：若从顶点Vi到Vj的边有方向，则称这条边为有向边，也成为弧(Arc)，用有序偶<Vi,Vj>来表示，Vi称为弧尾，Vj称为弧头。
简单图：在图结构中，若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图。
无向完全图：在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。含有n个顶点的无向完全图有n\*(n-1)/2条边。
有向完全图：在有向图中，如果任意两个顶点之间都存在方向互为相反的两条弧，则称该图为有向完全图。含有n个顶点的有向完全图有n\*(n-1)条边。
稀疏图和稠密图：这里的稀疏和稠密是模糊的概念，都是相对而言的，通常认为边或弧数小于n\*logn（n是顶点的个数）的图称为稀疏图，反之称为稠密图。
有些图的边或弧带有与它相关的数字，这种与图的边或弧相关的数叫做权(Weight)，带权的图通常称为网(Network)。
假设有两个图G1=(V1,E1)和G2=(V2,E2)，如果V2⊆V1，E2⊆E1，则称G2为G1的子图(Subgraph)。
边(V1,V2)依附(incident)于顶点V1和V2，或者说边(V1,V2)与顶点V1和V2相关联。
顶点V的度(Degree)是和V相关联的边的数目，记为TD(V)，如下图，顶点A与B互为邻接点，边(A,B)依附于顶点A与B上，顶点A的度为3。
对于有向图G=(V,E)，如果有<V1,V2>∈E，则称顶点V1邻接到顶点V2，顶点V2邻接自顶点V1。
以顶点V为头的弧的数目称为V的入度(InDegree)，记为ID(V)，以V为尾的弧的数目称为V的出度(OutDegree)，记为OD(V)，因此顶点V的度为TD(V)=ID(V)+OD(V)。
如果G是有向图，则路径也是有向的。
下图用红线列举顶点B到顶点D的两种路径，而顶点A到顶点B就不存在路径啦：
路径的长度是路径上的边或弧的数目。
第一个顶点到最后一个顶点相同的路径称为回路或环(Cycle)。
序列中顶点不重复出现的路径称为简单路径，除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为简单回路或简单环。
连通图:在无向图G中，如果从顶点V1到顶点V2有路径，则称V1和V2是连通的，如果对于图中任意两个顶点Vi和Vj都是连通的，则称G是连通图(ConnectedGraph)
无向图中的极大连通子图称为连通分量。
如果一个有向图恰有一个顶点入度为0，其余顶点的入度均为1，则是一棵有向树。
邻接矩阵（无向图）:
 图的邻接矩阵(Adjacency Matrix)存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组(称为邻接矩阵)存储图中的边或弧的信息。
对称矩阵：所谓对称矩阵就是n阶矩阵的元满足a[i][j]=a[j][i](0<=i,j<=n)。即从矩阵的左上角到右下角的主对角线为轴，右上角的元与左下角相对应的元全都是相等的。
邻接矩阵（有向图）
可见顶点数组vertex[4]={V0,V1,V2,V3}，弧数组arc[4][4]也是一个矩阵，但因为是有向图，所以这个矩阵并不对称，例如由V1到V0有弧，得到arc[1][0]=1，而V0到V1没有弧，因此arc[0][1]=0。
另外有向图是有讲究的，要考虑入度和出度，顶点V1的入度为1，正好是第V1列的各数之和，顶点V1的出度为2，正好是第V1行的各数之和。
邻接矩阵（网）
在图的术语中，我们提到了网这个概念，事实上也就是每条边上带有权的图就叫网。
邻接表（无向图）
邻接矩阵看上去是个不错的选择，首先是容易理解，第二是索引和编排都很舒服~
但是我们也发现，对于边数相对顶点较少的图，这种结构无疑是存在对存储空间的极大浪费。因此我们可以考虑另外一种存储结构方式，例如把数组与链表结合一起来存储，这种方式在图结构也适用，我们称为邻接表(AdjacencyList)。
图中顶点用一个一维数组存储，当然，顶点也可以用单链表来存储，不过数组可以较容易地读取顶点信息，更加方便。
图中每个顶点Vi的所有邻接点构成一个线性表，由于邻接点的个数不确定，所以我们选择用单链表来存储。
邻接表（有向图）
若是有向图，邻接表结构也是类似的，我们先来看下把顶点当弧尾建立的邻接表，这样很容易就可以得到每个顶点的出度：
但也有时为了便于确定顶点的入度或以顶点为弧头的弧，我们可以建立一个有向图的逆邻接表：
邻接表（网）
对于带权值的网图，可以在边表结点定义中再增加一个数据域来存储权值即可：
十字链表
 十字链表的好处就是因为把邻接表和逆邻接表整合在了一起，这样既容易找到以Vi为尾的弧，也容易找到以Vi为头的弧，因而容易求得顶点的出度和入度。
 十字链表除了结构复杂一点外，其实创建图算法的时间复杂度是和邻接表相同的，因此，在有向图的应用中，十字链表也是非常好的数据结构模型。
邻接多重表
 其中iVex和jVex是与某条边依附的两个顶点在顶点表中的下标。iLink指向依附顶点iVex的下一条边，jLink指向依附顶点jVex的下一条边。
 也就是说在邻接多重表里边，边表存放的是一条边，而不是一个顶点。
边集数组
 边集数组是由两个一维数组构成，一个是存储顶点的信息，另一个是存储边的信息，这个边数组每个数据元素由一条边的起点下标(begin)、终点下标(end)和权(weight)组成。
图的遍历
树的遍历我们谈了四种方式，大家回忆一下，树因为根结点只有一个，并且所有的结点都只有一个双亲，所以不是很难理解。
 但是谈到图的遍历，那就复杂多了，因为它的任一顶点都可以和其余的所有顶点相邻接，因此极有可能存在重复走过某个顶点或漏了某个顶点的遍历过程。
 对于图的遍历，如果要避免以上情况，那就需要科学地设计遍历方案，通常有两种遍历次序方案：它们是深度优先遍历和广度优先遍历。
 深度优先遍历
 我们可以约定右手原则：在没有碰到重复顶点的情况下，分叉路口始终是向右手边走，每路过一个顶点就做一个记号。
回溯法：
 之前我们谈过回溯法，还是那句话，指导思想很简单，就是一条路走到黑，碰壁了再回来一条路走到黑……一般和递归可以很好的搭配使用，还有深度优先搜索（DFS）。
哈密尔顿路径：
 图G中的哈密尔顿路径指的是经过图G中每个顶点，且只经过一次的一条轨迹。如果这条轨迹是一条闭合的路径（从起点出发不重复地遍历所有点后仍能回到起始点），那么这条路径称为哈密尔顿回路。

图的遍历（广度优先遍历） 广度优先遍历（BreadthFirstSearch），又称为广度优先搜索，简称BFS。 一个无环的有向图称为无环图（Directed Acyclic Graph），简称DAG图。 在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，这样的有向图为顶点表示活动的网，我们称之为AOV网（Active On Vertex Network）。 拓扑序列含义： 设G=(V,E)是一个具有n个顶点的有向图，V中的顶点序列V1,V2,……,Vn满足若从顶点Vi到Vj有一条路径，则在顶点序列中顶点Vi必在顶点Vj之前。则我们称这样的顶点序列为一个拓扑序列。 对AOV网进行拓扑排序的方法和步骤如下： 从AOV网中选择一个没有前趋的顶点（该顶点的入度为0）并且输出它； 从网中删去该顶点，并且删去从该顶点发出的全部有向边； 重复上述两步，直到剩余网中不再存在没有前趋的顶点为止。 算法时间复杂度： 对一个具有n个顶点，e条边的网来说，初始建立入度为零的顶点栈，要检查所有顶点一次，执行时间为O(n)。 排序中，若AOV网无回路，则每个顶点入、出栈各一次，每个表结点被检查一次，因而执行时间是 O(n+e)。 所以，整个算法的时间复杂度是 O(n+e)。 我们把AOE网中没有入边的顶点称为始点或源点，没有出边的顶点称为终点或汇点。 etv(Earliest Time Of Vertex)：事件最早发生时间，就是顶点的最早发生时间； ltv(Latest Time Of Vertex)：事件最晚发生时间，就是每个顶点对应的事件最晚需要开始的时间，如果超出此时间将会延误整个工期。 ete(Earliest Time Of Edge)：活动的最早开工时间，就是弧的最早发生时间。 lte(Latest Time Of Edge)：活动的最晚发生时间，就是不推迟工期的最晚开工时间。 顺序查找 顺序查找又叫线性查找，是最基本的查找技术，它的查找过程是：从第一个（或者最后一个）记录开始，逐个进行记录的关键字和给定值进行比较，若某个记录的关键字和给定值相等，则查找成功。如果查找了所有的记录仍然找不到与给定值相等的关键字，则查找不成功。 二叉排序数（Binary Sort Tree）又称为二叉查找树，它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不为空，则左子树上所有结点的值均小于它的根结构的值； 若它的右子树不为空，则右子树上所有结点的值均大于它的根结构的值； 它的左、右子树也分别为二叉排序树（递归）。 散列表的查找步骤 当存储记录时，通过散列函数计算出记录的散列地址 当查找记录时，我们通过同样的是散列函数计算记录的散列地址，并按此散列地址访问该记录 排序的稳定性 假设ki=kj(1<=i<=n,1<=j<=n,i!=j)，且在排序前的序列中ri领先于rj(即i<j)。 如果排序后ri仍领先于rj，则称所用的排序方法是稳定的； 反之，若可能使得排序后的序列中rj领先ri，则称所用的排序方法是不稳定的。 冒泡排序的要点： 1. 两两注意是相邻的两个元素的意思 2. 如果有n个元素需要比较n-1次，每一轮减少1次比较 3. 既然叫冒泡排序，那就是从下往上两两比较，所以看上去就跟泡泡往上冒一样。 优化冒泡排序算法 我们发现如果使用正宗的冒泡排序算法，当i等于1执行完的时候，我们发现程序只进行两两相邻元素的比较，而不用进行任何移动，所以完全可以不用再继续循环。 希尔排序的原理 希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本，希尔排序是非稳定排序算法。 希尔排序是基于插入排序的以下两点性质而提出改进方法的： 插入排序在对几乎已经排好序的数据操作时， 效率高， 即可以达到线性排序的效率 但插入排序一般来说是低效的， 因为插入排序每次只能将数据移动一位 堆排序 上节课我们介绍的希尔排序是对直接插入排序的改进，而我们这节课谈的堆排序是对选择排序进行改进的排序算法，堆排序算法的时间复杂度和希尔排序是一样的，都是O（nlogn）. 堆是具有下列性质的完全二叉树：每个结点的值都大于或者等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于或者等于其左右孩子结点的值，称为小顶堆。 堆排序算法: 堆排序（Heap Sort）就是利用堆进行排序的算法，它的基本思想是： 将待排序的序列构造成一个大顶堆（或小顶堆）。 此时，整个序列的最大值就是堆顶的根结点。将它移走（就是将其与堆数组的末尾元素交换，此时末尾元素就是最大值）。 然后将剩余的n-1个序列重新构造成一个堆，这样就会得到n个元素中的此大值。 如此反复执行，便能得到一个有序序列了。 归并排序（递归实现） 归并”一词在中文含义中就是合并的意思，而在数据结构中的定义是将两个或者两个以上的有序表组合成一个新的有序表，就叫归并。 归并排序（Merge Sort）就是利用归并的思想实现的排序方法。它的原理是假设初始序列有n个记录，则 可以看成是n个有序的子序列，每个子序列的长度为1，然后两两归并，得到⌈n/2⌉个长度为2或1的有序子序列；再两两归并，……，如此重复，直至得到一个长度为n的有序序列为止，这种排序方法称为2路归并排序。 快速排序 排序算法（Quict Sort）的基本思想是： 通过一趟排序将待排序记录分割成独立地两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序的目的。