第一章 数据结构绪论

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

数据：描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合。“整型、实型等数值类型和字符、声音、图像、视频等非数值类型”

数据元素：组成数据的、有一定意义的基本单位，在计算机中通常作为整体处理，也被称为记录。

数据项：一个数据元素可以由若干个数据项组成。“不可分割的最小单位”

数据对象：是性质相同的数据元素的集合，是数据的子集。

数据结构：相互间存在一种或多种特定关系的数据元素的集合。

逻辑结构（数据对象中数据元素之间的相互关系）：---------->面向问题的

- 集合结构（数据元素属于同一个集合，但互相之间没有关系）
- 线性结构（数据元素之间是一对一的关系）
- 树形结构（数据元素之间是一对多的关系）
- 图形结构（数据元素之间是多对多的关系）

物理结构（数据的逻辑结构在计算机中的存储形式）：---------->面向计算机的

- 顺序存储结构（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只的，如数组）
- 链式存储结构（针对经常要变化的结构。把数据元素存放在任意的存储单元里，用指针存放数据元素的地址，并通过指针找到相关联数据的位置）

数据类型：是指一组性质相同的值的集合以及定义在此集合上的一些操作的总称。

抽象是指抽取出事物具有的普遍性的本质。

抽象数据类型（ADT）：是指一个数学模型以及定义在该模型上的一组操作。它体现了程序设计问题中问题分解、抽象和信息隐藏的特性。

第二章 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。

算法的五个基本特性：

- 输入（零或多个输入）
- 输出（至少一个或多个输出）
- 有穷性（算法在执行有限的步骤后，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（算法的每一步都有确定含义，不会出现二义性）
- 可行性（算法的每一步都能通过执行有限的次数完成）

算法设计的要求：

1，正确性

（1）没有语法错误

（2）对合法输入产生满足要求的输出结果

（3）对非法输入得出满足规格的说明结果

（4）对精心选择的甚至刁难的测试数据都有满足要求的输出结果。

2，可读性（好的算法应该容易理解、便于阅读和交流）

3，健壮性（当输入数据不合法时，算法也能做出相关处理。而不是产生异常或莫名其妙的结果）

4，时间效率高和存储量低

算法效率的度量方法：事后统计方法（较差，不科学，不准确），事前分析估计算法

一个高级语言编写的程序在计算机上运行时所消耗的时间取决于：算法采用的策略、方法，编译产生的代码质量（算法好坏的根本），问题的输入规模（软件支持），机器执行指令的速度（硬件性能）。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与输入规模关联起来，即基本操作的数量必须表示成输入规模的函数。

函数的渐进增长：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)的增长渐进快于 g(n)。

算法的时间复杂度，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，称作算法的渐进时间复杂度。其中，f(n)是问题规模 n 的某个函数。

常见的时间复杂度：
O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)

平均时间复杂度和最坏时间复杂度，一般要求计算时，均指计算最坏的时间复杂度

算法的空间复杂度：是通过计算算法所需的存储空间实现，算法空间复杂度的计算公式：S（n）=O（f(n)），其中，n 为问题规模，f(n)为语句关于 n 所占存储空间的函数。一般情况下，一个程序在机器上执行时，除了所需要存储程序本身的指令、常数、变量和输入数据外，还需要存储对数据操作的存储单元。

第三章 线性表

线性表（List）：零个或多个数据元素的有限序列。（强调有序、有限）

1、描述顺序存储结构需要三个属性：

- 存储空间的起始位置：数组 data，它的存储位置就是存储空间的存储位置
- 线性表的最大存储容量：数组长度 MaxSize
- 线性表的当前长度：length

2、插入算法的思路（如买票插队）：

- 如果插入位置不合理，抛出异常
- 如果线性表长度大于等于数组长度，则抛出异常或动态增加容量
- 从最后一个元素开始向前遍历到第 i 个位置，分别将它们都向后移动一个位置
- 将要插入元素填入位置 i 处
- 表长加 1

3、删除算法的思路（如有人离开买票队伍）：

- 如果删除位置不合理，抛出异常
- 取出删除元素
- 从删除元素位置开始遍历到最后一个元素位置，分别将它们都向前移动一个位置
- 表长减 1

4、线性表的顺序存储结构的优缺点：

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一位置的元素

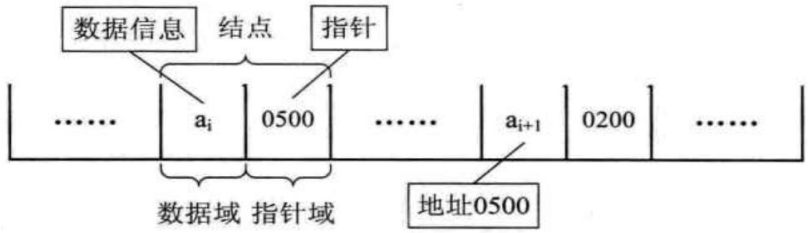
缺点：

- 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
- 造成存储空间的“碎片”

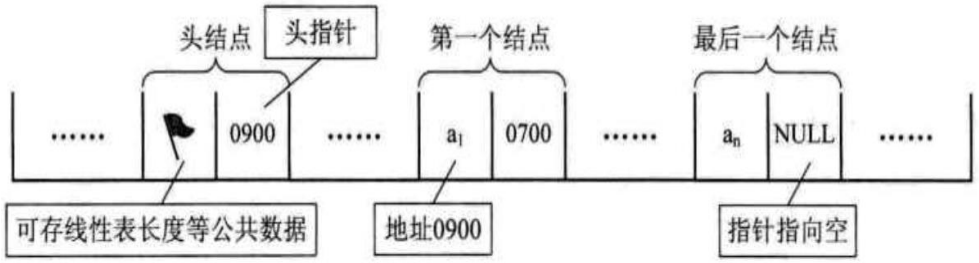
5、单链表

结点：数据域（存放数据元素）+ 指针域（存放后继结点地址）

n 个结点（ai 的存储映像）链结成一个链表，即为线性表（a1,a2,...,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。单链表正是通过每个结点的指针域将线性表的数据元素按其逻辑次序链接在一起



有时，为了更加方便地对链表进行操作，会在单链表的第一个结点前附设一个结点，称为头结点。头结点的数据域可以不存储任何信息，也可以存储如线性表的长度等附加信息，头结点的指针域存储指向第一个结点的指针



6、头指针与头结点的异同

头指针：

- 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
- 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：

- 头结点是为了操作的统一和方便而设立的，放在第一元素的结点之前，其数据域一般无意义（也可存放链表的长度）
- 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，其操作与其它结点的操作就统一了
- 头结点不一定是链表必须要素

7、获取链表第 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 j<i 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，返回结点 p 的数据

8、单链表第 i 个数据插入结点的算法思路

- 声明一结点 p 指向链表第一个结点，初始化 j 从 1 开始
- 当 j<i 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
- 单链表的插入标准语句 s->next=p->next;p->next=s
- 返回成功

9、单链表第 i 个数据删除结点的算法思路

- 声明一结点 p 指向链表第一个结点，初始化 j 从 1 开始
- 当 j<i 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，将欲删除的结点 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
- 返回成功

10、单链表整表创建的算法思路

- 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：1、生成一新结点赋值给 p　2、随机生成一数字赋值给 p 的数据域 p->data　3、将 p 插入到头结点与前一新节点之间

11、单链表的整表删除

- 声明一结点 p 和 q
- 将第一个结点赋值给 p
- 循环：1、将下一结点赋值给 q 2、释放 p 3、将 q 赋值给 p

12、单链表结构和顺序存储结构做对比

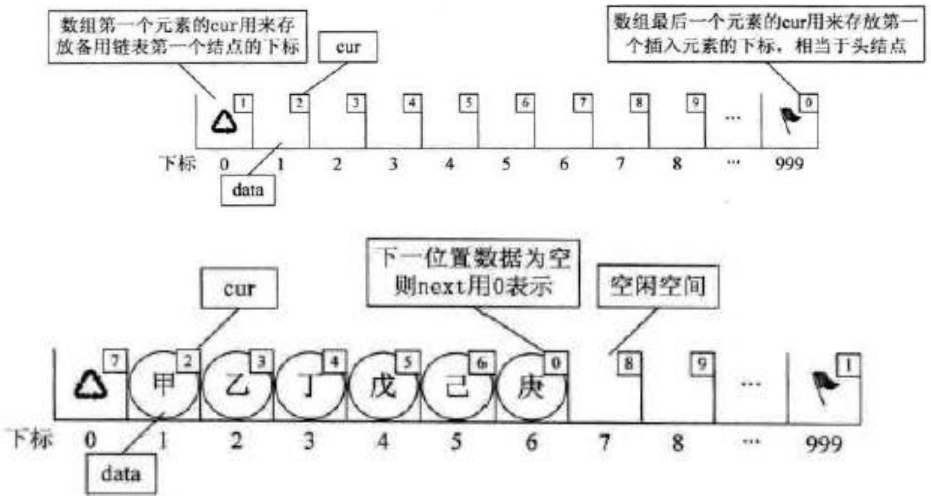
1、存储分配方式：顺序存储结构用一段连续的存储单元依次存储线性表的数据元素；单链表采用链式存储结构，用一组任意的存储单元存放线性表的元素。

2、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）单链表在选出某位置的指针后，插入和删除时间仅为 O (1) 。

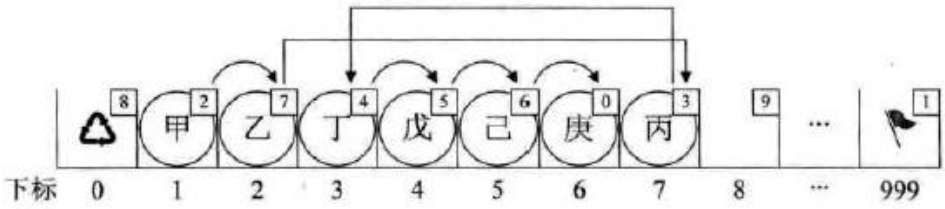
3、空间性能：顺序存储结构需要预分配存储空间，大了浪费，小了上溢；单链表不需要预分配存储空间，只要有就可以分配，元素个数也不受限制。

13、静态链表

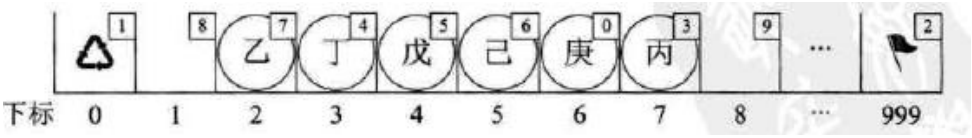
在没有指针的语言中，用数组来代替指针：每个数组的元素都有两个数据域，data 和 cur，data 用来存放数据元素，游标 cur 相当于单链表中的指针。这种用数组描述的链表称为静态链表。



插入丙：



删除甲：



静态链表的优缺点：优点：在插入和删除操作时，只需要修改游标，不需要移动元素，进而改进了在顺序存储结构中的插入和删除操作需要移动大量元素的缺点。缺点：1、没有解决连续存储分配带来的表长难以确定的问题。2、失去了顺序存储结构随机存取的特性。

总的来说，静态链表其实是为了给没有指针的高级语言设计的一种实现单链表能力的方法。

14、循环链表

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称为单循环链表，简称循环链表（circular linked list）

循环链表带有头结点的空链表如图 3-13-4：

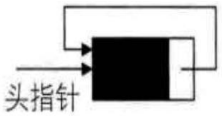
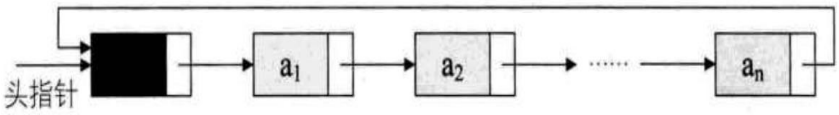


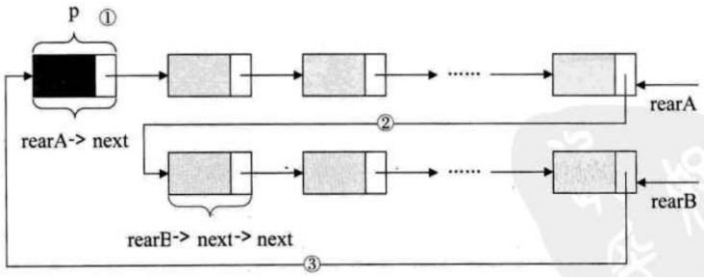
图 3-13-3

对于非空的循环链表就如图 3-13-4 所示。



循环链表和单链表的主要差异就在于循环的判断条件上，原来是判断 p->next 是否为空，现在则是 **p->next 不等于头结点，则循环未结束**

15、合并两个循环链表



16、双向链表

double linked list 是在单链表的每个结点中，再设置一个指向其前驱结点的指针域。

双向链表的循环带头结点的空链表如图 **3-14-3** 所示。

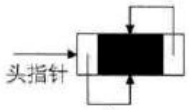


图 3-14-3

非空的循环的带头结点的双向链表如图 **3-14-4** 所示。

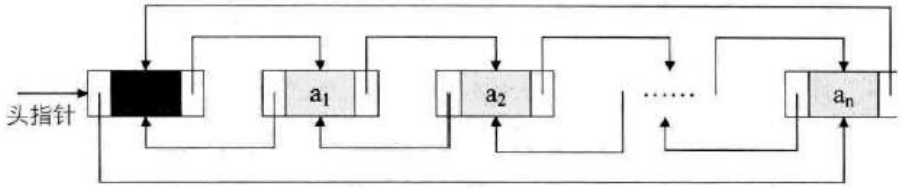
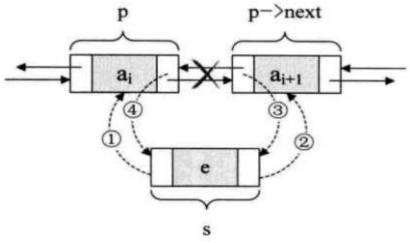


图 3-14-4

插入：



17、线性表的总结

线性表				
顺序存储结构	链式存储结构			
	单链表	静态链表	循环链表	双向链表

第四章 栈与队列

1、栈（stack）：是限定仅在表尾进行插入和删除操作的线性表，我们把允许插入和删除的一端称为栈顶（top），另一端称为栈底（bottom），不含任何数据元素的栈称为空栈。栈成为后进先出（Last In First Out）的线性表，简称 LIFO 结构。

2、插入（入栈--push）和删除（出栈--pop）都是时间复杂度 O（1）的操作。

3、当栈存在一个元素时，top 等于 0，因此通常把空栈的判定条件定位 top 等于-1（索引值从 0 开始）

4、用一个数组来存储两个栈（一般用于相反关系时，比如股票，有人卖出有人买入）

当程序中同时使用两个栈时，可以将两个栈的栈底设在向量空间的两端，让两个栈各自向中间延伸。如下图所示：

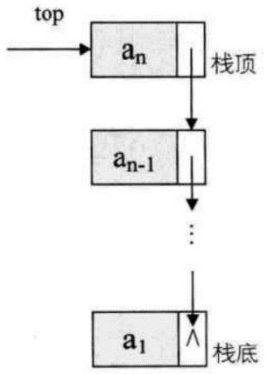
当一个栈的元素较多，超过向量空间的一半时，只要另一个栈的元素不多，那么前者就可以占用后者的部分存储空间。

只有当整个向量空间被两个栈占满（即两个栈顶相遇）时，才会发生上溢，因此两个栈共享一个长度为n的向量空间。

数组有两个端点，两个栈有两个栈底，让一个栈的栈底为数组的始端，即下标为 0 处，另一个栈为栈的末端，即下标为数组长度 n-处。这样，如果两个栈增加元素，就是两端点向中间延伸

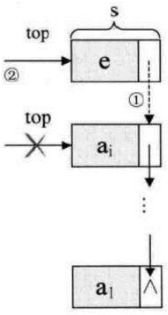
两个栈见面之时，也就是两个指针之间相差 1 时，即 top1+1==top2 为栈满

5、栈的链式存储结构

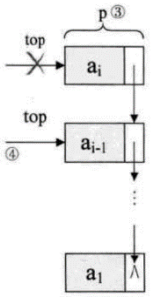


链栈一般不存在栈满的情况，绝大部分操作都与单链表类似：

Push：



Pop：



6、递归定义：一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称做递归函数。每个递归定义必须至少有一个条件，满足时递归不再进行，即不再引用自身而是返回值退出。

递归使用栈来实现的。

7、逆波兰表达式

标准四则运算表达式（中缀表达式）9+（3-1）* 3+ 10 / 2 的后缀表达式为：931-3*+10 2 / +

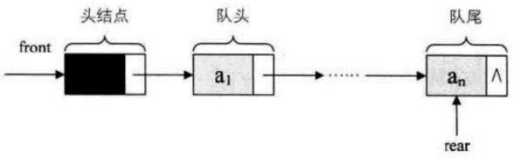
后缀表达式的运算规则：从左到右遍历每个数字和符号，遇到数字就进栈，遇到符号，就将栈顶的两个数字出栈，进行运算，运算结果进栈，一直到最终获得结果。

中缀转后缀的规则：从左到右遍历每个数字和符号，遇到数字就输出，遇到符号，则判断其与栈顶符号的优先级，是右括号或者优先级低于栈顶符号，则栈顶元素依次出栈并输出（如果是右括号，依次输出到直至左括号，如果是低级符号，弹出之前入栈的所有符号，再将当前低级符号进栈）一直到最终输出后缀表达式为止。

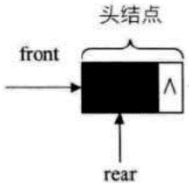
8、队列：只允许在一段进行插入操作，而在另一端进行删除操作的线性表，先进先出（FIFO）允许插入的一端称为队尾，允许删除的一端称为队头。

9、顺序存储结构：队列的头尾相接的顺序存储结构成为循环队列。

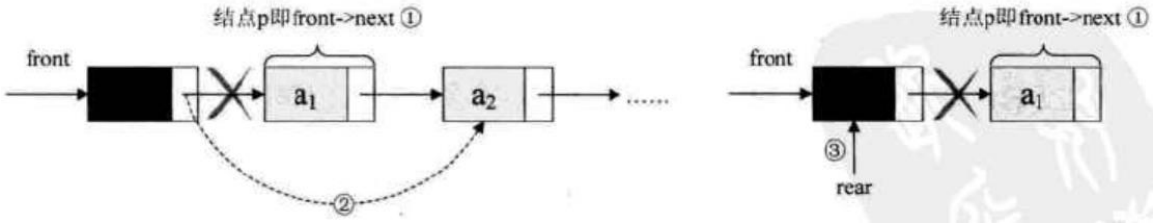
10、队列的链式存储结构，其实就是线性表的单链表，只不过它只能尾进头出而已，我们把它简称为链队列；队头指针指向链队列的头结点，而队尾指针指向终端结点



空队列时，front 和 rear 都指向头结点：



11、入队即单链表队尾的插入操作，出队是头结点的后继节点出队，将头结点的后继改为它后面的节点，若链表除了头结点只剩一个元素时，则需将 rear 指向头结点。



12、**在可以确定队列长度最大值的情况下，建议用循环队列，如果你无法预估队列的长度时，则用链队列。**

13、栈和队列的存储结构：

栈	队列
1、顺序栈 1.1、两栈共享空间 2、链栈	1、顺序队列 1.1、循环队列 2、链队列

第五章 串

1、串（string）是由零个或多个字符组成的有限序列，又名叫字符串。

2、2.1 空格串，是只包含空格的串，空格串是有内容有长度的，而且可以不止一个空格

2.2 串中任意个数的连续字符组成的子序列称为该串的子串，包含子串的串称为主串

2.3 子串在主串中的位置就是子串的第一个字符在主串中的序号

3、Unicode 和 ASCII 编码

计算机中的常用字符是使用标准的 ASCII 编码，更准确一点，由 7 位二进制数表示一个字符，总共可以表示 128 个字符。可是换做全世界估计要有成百上千种语言与文字，显然这 256 个字符是不够的，因此后来就有了 Unicode 编码，比较常用的是由 16 位的二进制数表示一个字符，这样总共就可以表示 216 个字符，总共约是 65 万多个字符，足够表示世界上所有语言的所有字符了。当然，为了和 ASCII 码兼容，Unicode 的前 256 个字符与 ASCII 码完全相同

4、两个字符串的比较

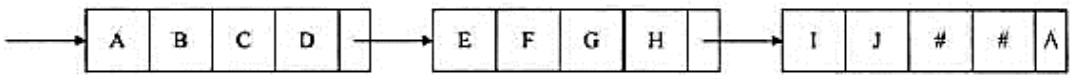
给定两个串：s="a1a2.....an"，t="b1b2.....bm"，当满足以下条件之一时，s<t

- n<m，且 ai=bi（i=1，2，.....n），例如当 s="hap"，t="happy"，就有 s<t。因为 t 比 s 多出了两个字母
- 存在某个 k≤min(m,n)，使得 ai=bi（i=1，2，.....，,k-1），ak<bk，例如当 s="happen"，t="happy"，因为两串的前 4 个字母均相同，而两串第 5 个字母（k 值），字母 e 的 ASCII 码是 101，而字母 y 的 ASCII 码是 121，显然 e<y，所以 s<t

5、串的逻辑结构和线性表很相似，只是串中的元素都是字符，那怕串中的数字，也是字符。

6、串的顺序存储结构：插入删除时移动大量字符，且在溢出时截尾。（C 语言中有自由存储区“堆”来动态分配，可以解决溢出问题）

7、串的链式存储结构：与线性表相似，但因为串的特殊性，结构中每个元素都是一个字符，如果也用简单的应用链表存储值，可能会浪费很多空间，所以，一个结点可以存放多个字符，最后一个结点为占满时，可以使用#或其他非串指字符补全：但串的链式存储结构除了连接串和串操作时有一定方便处外，总体不如顺序存储灵活，性能也不好。



8、匹配模式：子串的定位操作通常称作串的模式匹配。

朴素的匹配模式：暴力方式，时间复杂度为 O((n-m+1)*m)，n 为主串长，m 为子串长。

KMP 匹配模式：算法复杂度为：匹配过程 O(n) + 计酸 next 的 O（m）的时间，计 O（m+n）

五、树

1、一些概念

结点拥有的子树数称为结点的度（Degree），度为 0 的结点称为叶点（Leaf）或终端结点；度不为 0 的结点称为非终端结点或分支结点，除根结点之外，分支结点也成为内部结点，树的度是树内各结点的度的最大值，树中结点的最大层次称为树的深度（Depth）或高度

2、子树、子结点、双亲结点、兄弟、祖先、子孙、森林（m 棵不想交的树的集合）

3、双亲表示法：以一组连续空间存储树的结点，同时在每个结点中，附设一个指示器指示其双亲结点到链表中的位置;由于根结点是没有双亲的，所以我们约定根结点的位置域设置为-1

data （数据域）	parent（指针域）

这样的存储结构，我们可以根据结点的 parent 指针很容易找到它的双亲结点，知道 parent 为-1 时，表示找到了树结点的根。可如果我们要知道结点的孩子是什么，需要遍历整个结构。

4、多重链表表示法

由于树中每个结点可能有多棵子树，可以考虑用多重链表，即每个结点有多个指针域，其中每个指针指向一棵子树的根节点，我们把这种方法叫做多重链表表示法；不过，树的每个结点的度，也就是孩子个数是不同的，所以可以设计两种方案来解决:

方案一：指针域的个数就等于树的度（树的度是树各个结点度的最大值）

data	child1	child2	child3	·····	childd

其中 data 是数据域，child1 到 childd 是指针域，用来指向该结点的孩子结点。这种方法对于树中各结点的度相差很大时，显然是很浪费空间的，因为有很多的结点，它的指针域都是空的。

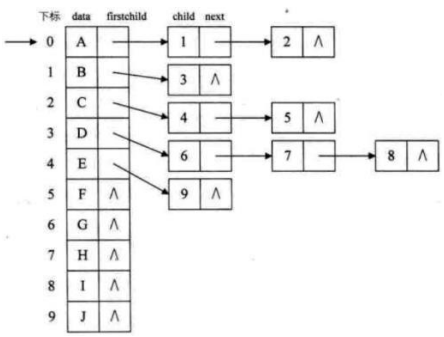
方案二：每个结点指针域的个数等于该结点的度，我们专门取一个位置来存储结点指针域的个数。

data	degree	child1	child2	·····	childd

这种方法提升了空间利用率，但是由于各个结点的链表是不相同的结构，加上要维护结点的度的数值，在运算上就会带来时间上的损耗。

5、孩子表示法：

把每个结点的孩子结点排列起来，以单链表作存储结构，则 n 个结点有 n 个孩子链表，如果是叶子结点则此单链表为空。然后 n 个头指针又组成一个线性表，采用顺序存储结构，存放进一个一维数组中



为此，设计两种结点结构，一个是孩子链表的孩子结点，如表 6-4-7 所示。

表 6-4-7

child	next

其中 child 是数据域，用来存储某个结点在表头数组中的下标。next 是指针域，用来存储指向某结点的下一个孩子结点的指针。

另一个是表头数组的表头结点，如表 6-4-8 所示。

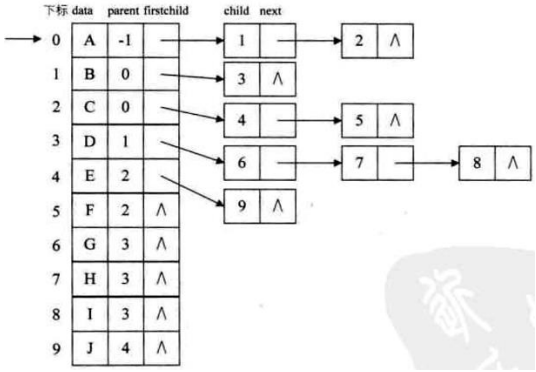
表 6-4-8

data	firstchild

其中 data 是数据域，存储某结点的数据信息。firstchild 是头指针域，存储该结点的孩子链表的头指针。

这样的结构对于查找孩子、兄弟，只需要查找这个结点的孩子单链表即可，对于遍历整棵树也很方便，对头结点的数组循环即可。但查找双亲比较麻烦。

6、双亲孩子表示法



7、孩子兄弟表示法

任意一棵树，它的结点的第一个孩子如果存在就是唯一的，它的右兄弟如果存在也是唯一的。因此，我们设置两个指针，分别指向该结点的第一个孩子和此结点的右兄弟

data	firstchild	rightsib

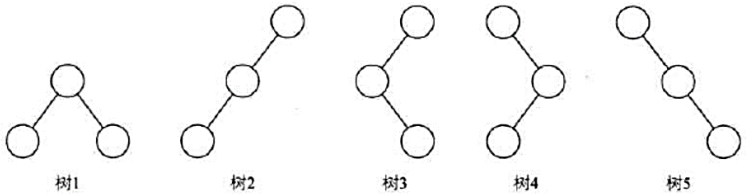
data 是数据域，firstchild 为指针域，存储该结点的第一个孩子结点的存储地址，rightsib 是指针域，存储该结点的右兄弟结点的存储地址，这个表示法的最大好处是它把一棵复杂的树变成了一棵二叉树，如需方便查找双亲，则再添加一个 parent 指针域。

8、二叉树特点

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（没有子树或者有一棵子树都是可以的）。左子树和右子树是有顺序的，次序不能任意颠倒。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

9、二叉树五种基本形态

- 空二叉树
- 只有一个根结点
- 根结点只有左子树
- 根结点只有右子树
- 根结点既有左子树又有右子树

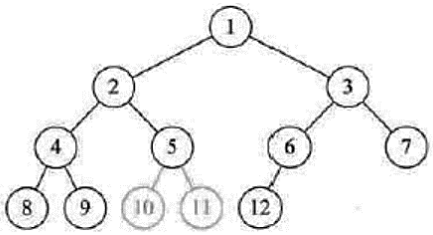


10、特殊二叉树

斜树：所有的结点都只有左子树的二叉树叫左斜树，所有结点都是只有右子树的二叉树叫右斜树，这两者统称为斜树。

满二叉树：在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。

完全二叉树：对一棵具有 n 个结点的二叉树按层序编号，如果编号为 i（1≤i≤n）的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中位置完全相同，则这棵二叉树称为完全二叉树



完全二叉树的特点

- 叶子结点只能出现在最下两层
- 最下层的叶子一定集中在左部连续位置
- 倒数二层，若有叶子结点，一定都在右部连续位置
- 如果结点度为 1，则该节点只有左孩子，即不存在只有右子树的情况
- 同样结点数的二叉树，完全二叉树的深度最小

11、二叉树的性质

1. 在二叉树的第 i 层上至多有 2 i-1 个结点（i≥1）

2. 深度为 k 的二叉树至多有 2k-1 个结点（k≥1）
3. 对任何一棵二叉树 T，如果其终端结点数为 n0，度为 2 的节点数为 n2，则 n0=n2+1

   终端结点数其实就是叶子结点数，而一棵二叉树，除了叶子结点外，剩下的就是度为 1 或 2 的结点数了，我们设 n_1 为度是 1 的结点数。则树 T 结点总数 $n=n_0+n_1+n_2$。

   比如图 6-6-1 的例子，结点总数为 10，它是由 A、B、C、D 等度为 2 结点，F、G、H、I、J 等度为 0 的叶子结点和 E 这个度为 1 的结点组成。总和为 4+1+5=10。

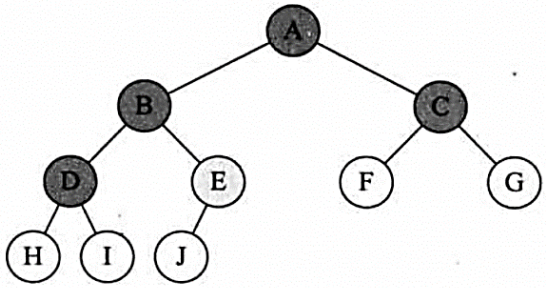


图 6-6-1

   我们换个角度，再数一数它的连接线数，由于根结点只有分支出去，没有分支进入，所以分支线总数为结点总数减去 1。图 6-6-1 就是 9 个分支。对于 A、B、C、D 结点来说，它们都有两个分支线出去，而 E 结点只有一个分支线出去。所以总分支线为 4×2+1×1=9。

   用代数表达就是分支线总数=n－1=n_1+2n_2。因为刚才我们有等式 $n=n_0+n_1+n_2$，所以可推导出 $n_0+n_1+n_2-1=n_1+2n_2$。结论就是 $n_0=n_2+1$。
4. 具有 n 个结点的完全二叉树的深度为⌊log2n⌋+1（⌊x⌋表示不大于 x 的最大整数）
5. 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（从第 1 层到第⌊log2n⌋+1 层，每层从左到右），对任一结点 i（1≤i≤n）有：
   - 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 无左孩子（结点 i 为叶子结点）；否则其左结点是结点 2i
   - 如果 2i+1>n，则结点 i 无右海子；否则其右孩子是结点 2i+1

12、二叉树顺序存储和二叉链表

顺序存储适用于完全二叉树如下图 1，然而极端情况会有很多冗空间浪费，如下图 2

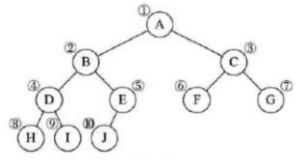
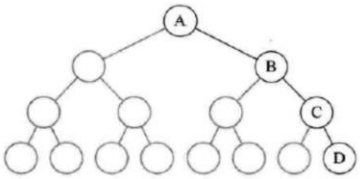


图 6-7-1

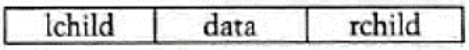
将这棵二叉树存入到数组中，相应的下标对应其同样的位置，如图 6-7-2 所示。

下标:	1	2	3	4	5	6	7	8	9	10
	A	B	C	D	E	F	G	H	I	J



下标:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	∧	B	∧	∧	∧	C	∧	∧	∧	∧	∧	∧	∧	D

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。



其中 data 是数据域，lchild 和 rchild 都是指针域，分别存放指向左孩子和右孩子的指针。

13、二叉树遍历方法

前序遍历：根左右（需要掌握递归和迭代算法）

中序遍历：左根右（需要掌握递归和迭代算法）

后序遍历：左右根（掌握递归算法即可）

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，按从左到右的顺序堆结点逐个访问。（需要掌握迭代算法）

preorder:

```
def iteration(self, root, result):
    stack = []
    while root or stack:
        if root:
            result.append(root.val)
            stack.append(root)
            root = root.left
        if not root:
            root = stack.pop().right
    return result
```

inorder:

```
def recursion(self, root, result):
    if not root:
        return result
    return self.recursion(root.left, result) + [root.val] + self.recursion(root.right, result)
def iteration(self, root, result):
    stack = []
    while root or stack:
        if root:
            stack.append(root)
```

```
            root = root.left
        if not root:
            root = stack.pop()
            result.append(root.val)
            root = root.right
    return result
```

LeverlOrder：

```
    if not root:
        return []
    queue = deque([root])
    result = []
    while queue:
        cur = []
        for _ in range(len(queue)):
            root = queue.popleft()
            cur.append(root.val)
            if root.left:
                queue.append(root.left)
            if root.right:
                queue.append(root.right)
        result.append(cur)
    return result
```

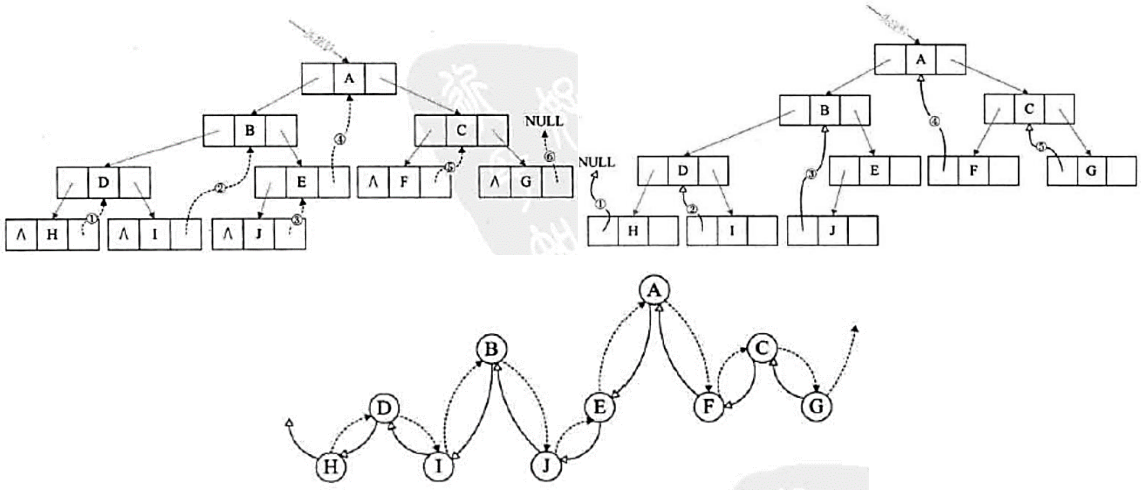
14、两个二叉树遍历的性质

- 已知前序遍历序列和中序遍历序列，可以唯一确定一棵二叉树
- 已知后序遍历序列和中序遍历序列，可以唯一确定一棵二叉树
- 但是已知前序和后序遍历，是不能确定一棵二叉树的

15、线索二叉树

指向前驱和后继的指针称为线索，加上线索的二叉链表称为线索链表，相应的二叉树就称为线索二叉树：

先将所有空指针域中的 rchild，指向其后继结点，再将所有空指针域的 lchild，指向当前的前驱



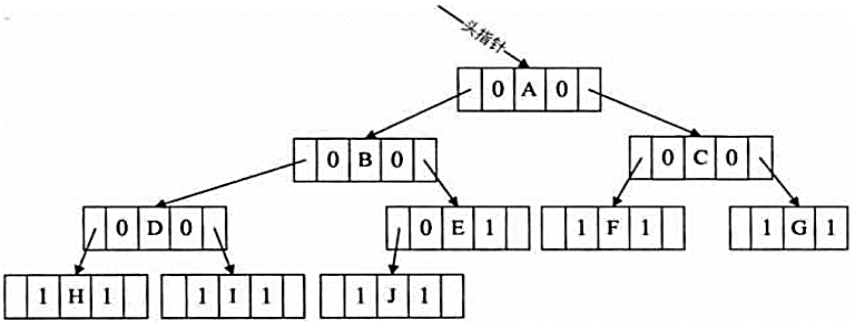
通过上图（空心箭头实线为前驱，虚线黑箭头为后继），可以看出，其实线索二叉树，等于是把一棵二叉树转变成了一个双向链表；所以我们对二叉树以某种次序遍历使其变为线索二叉树的过程称做是线索化。

但是，我们并不知道某一结点的 lchild 是指向它的左孩子还是指向前驱，所以需要一个区分标致；因此，我们在每个结点再增设两个标志域 ltag 和 rtag，这两个 tag 只是存放 0 或 1 数字的布尔型变量，其占用的内存空间要小于像 lchild 和 rchild 的指针变量，结点结构如下：

lchild	ltag	data	rtag	rchild

ltag 为 0 时指向该结点的左孩子，为 1 时指向该结点的前驱

rtag 为 0 时指向该结点的右孩子，为 1 时指向该结点的后继

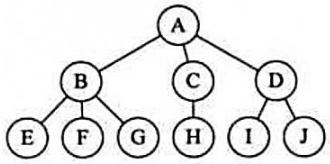


如果所用的二叉树需经常遍历或查找结点时需要某种遍历序列中的前驱和后继，那么采用线索二叉链表的存储结构就是非常不错的选择。

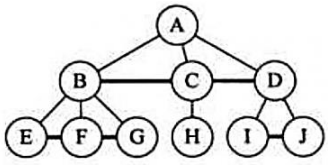
16、树转换为二叉树

- 加线，在所有兄弟结点之间加一条连线

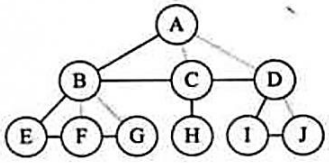
- 去线，对树中每个结点，只保留它与第一个孩子结点的连线，删除它与其他孩子结点之间的连线
- 层次调整，以树的根结点为轴心，将整棵树顺时针旋转一定的角度，使之结构层次分明，注意第一个孩子是二叉树结点的左孩子，兄弟转换过来的孩子是结点的右孩子



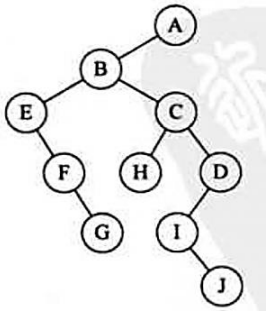
树



步骤1：给兄弟加线



步骤2：给除长子外的孩子去线

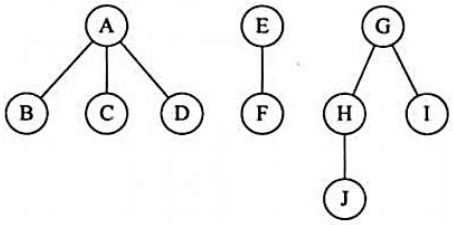


步骤3：层次调整

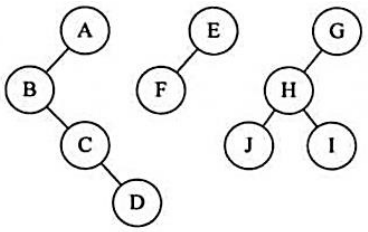
17、森林转换为二叉树

森林相当于每一棵树都是兄弟，先把每个树转换为二叉树

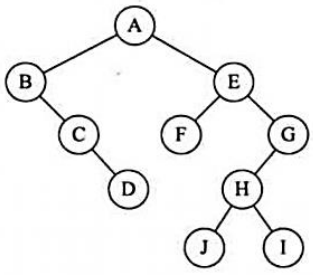
第一棵二叉树不动，从第二棵二叉树开始，以此把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，用线连接起来，当所有的二叉树连接起来后就得到了由森林转换来的二叉树



拥有三棵树的森林



步骤1:森林中每棵树转换为二叉树



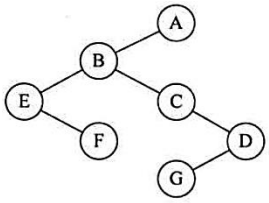
步骤2：将所有二叉树转换为一棵二叉树

18、二叉树转换为树

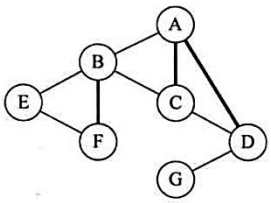
加线，若某结点的左孩子结点存在，则将这个左孩子的右孩子结点、右孩子的右孩子结点、右孩子的右孩子的右孩子结点……哈，反正就是左孩子的 n 个右孩子结点都作为此结点的孩子，将该结点与这些右孩子结点用线连接起来

去线，删除原二叉树中所有结点与其右孩子结点的连线

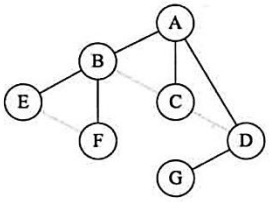
层次调整，使之结构层次分明



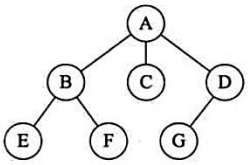
二叉树



步骤1：加线



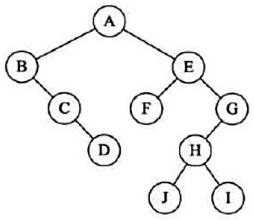
步骤2：去线



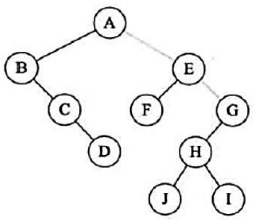
步骤3：层次调整

19、二叉树转换为森林

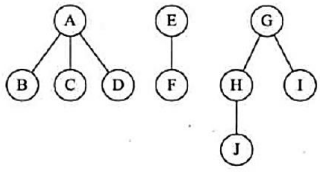
判断一棵二叉树能够转换成一棵树还是森林，标准很简单，那就是只要看这棵二叉树的根结点有没有右孩子，有就是森林，没有就是一棵树。从根结点开始，若右孩子存在，则把与右孩子结点的连线删除，再查看分离后的二叉树，若右孩子存在，则连线删除……，直到所有右孩子连线都删除为止，得到分离的二叉树，再将每棵分离后的二叉树转换为树即可



二叉树



步骤1：寻找右孩子去线



步骤2：将分离的二叉树转换成树

20、树的遍历

一种是先根遍历树，即先访问树的根结点，然后依次先根遍历根的每棵子树

另一种是后根遍历，即先依次后根遍历每棵子树，然后再访问根结点

21、森林的遍历

前序遍历：先访问森林中第一棵树的根结点，然后再依次先跟遍历根的每棵子树，再依次用同样方式遍历除去第一棵树的剩余树构成的森林

后序遍历：是先访问森林中第一棵树，后根遍历的方式遍历每棵子树，然后再访问根结点，再依次同样方式遍历除去第一棵树的剩余树构成的森林

森林的前序遍历和二叉树的前序遍历结果相同，森林的后序遍历和二叉树中的中序遍历结果相同

22、赫夫曼树算法描述

- 根据给定的 n 个权值{w1，w2，…wn}构成 n 棵二叉树的集合 F={T1，T2，…Tn},其中每个二叉树 Ti 中只有一个带权为 wi 的根结点，其左右子树均为空。
- 在 F 中选择两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。
- 在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。
- 重复 2 和 3，直到 F 只含一棵树为止。这棵树便是赫夫曼树。

23、赫夫曼编码

一般地，设需要编码的字符集为{d1，d2，…dn}，各个字符在电文中出现的次数或频率集合为{w1，w2，…wn}，以 d1，d2，…dn 作为叶子结点，以 w1，w2，…wn 作为相应叶子结点的权值来构造一棵赫夫曼树。规定赫夫曼树的左分支代表 0，右分支代表 1，则从根结点到叶子结点所经过的路径 分支组成的 0 和 1 的序列便为该结点对应字符的编码，这就是赫夫曼编码

六、图

1、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（vertex）是图 G 中顶点的集合，E（edge）是图 G 中边的集合。其中 V 有穷非空，E 可以是空的

2、关于图的一些定义

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），用无需偶对（vi，vj）来表示，如果图中任意两个顶点之间的边都是无向边，则图是无向图（undirected graphs）

有向边：若从顶点 vi 到 vj 的边有方向，则称为有向边，或弧（Arc），用有序对偶<vi,vj>表示

无向边用小括号“（）”表示，而有向边则是用尖括号“<>”表示

在图中，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**

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 个定点的无向完全图有 n*（n-1）/2 条边

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**

有很少条边或弧的图称为**稀疏图**，反之称为**稠密图**；这里稀疏和稠密是模糊的概念，是相对而言的

有些图的边或弧具有与它相关的数字，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 G=（V，{E}）和 G'=（V'，{E'}），如果 V'⊆V 且 E'⊆E，则称 G'为 G 的**子图**（Subgraph）

无向图中如果两定点之间存在边，则这两个点是邻接点（adjacent），该边依附（incident）与这两个定点。顶点的度（degree）是该点相关联的边的数目，记为 TD（v）

有向图中如果有<v，v’>，那么 v’邻接自 v，进入 v 的边数量为入度（InDegree），记为 ID(v)，从 v 出发的边数量为出度（OutDegree），记为 OD(v)，v 的度为 TD(v)= ID(v)+OD(v)

图中从一个点到另一个点的路径（Path）是一个顶点序列，路径的长度是路过的边的数目。

图中顶点间存在路径，两顶点存在路径则说明是连通的，如果路径最终回到起始点则称为环，当中不重复叫简单路径。若任意两顶点都是连通的，则图就是连通图，有向则称强连通图。图中有子图，若子图极大连通则就是连通分量，有向的则称强连通分量。

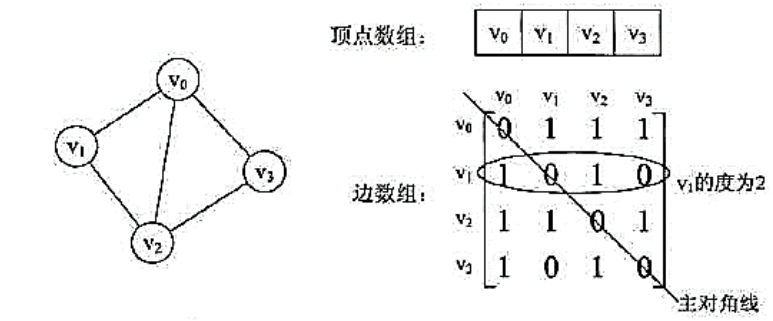
无向图中连通且 n 个顶点 n-1 条边叫生成树。有向图中一顶点入度为 0 其余顶点入度为 1 的叫有向树。一个有向图由若干棵有向树构成生成森林

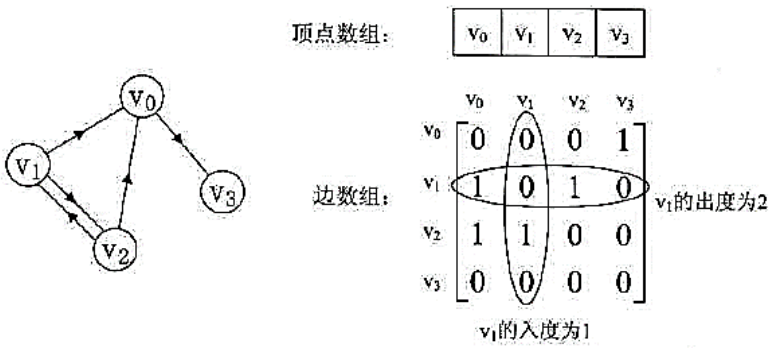
3、图的邻接矩阵（顺序存储）

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

$$arc[i][j] = \begin{cases} 1, 若(v_i,v_j) \in E 或 < v_i,v_j > \in E \\ 0, 反之 \end{cases}$$

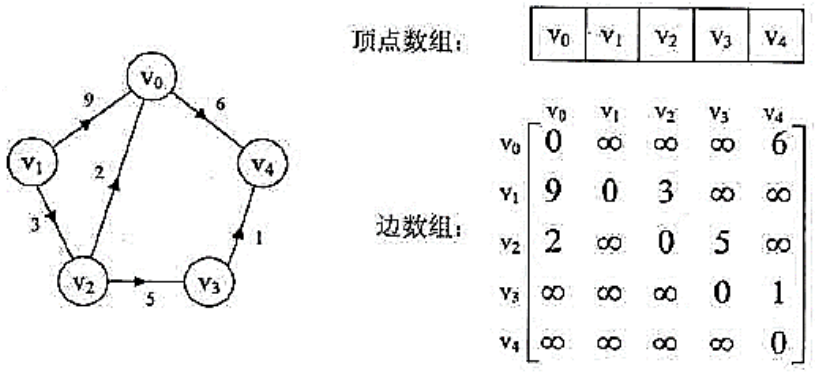
我们来看一个实例，图 7-4-2 的左图就是一个无向图。





n 个顶点和 e 条边的无向网图的创建，时间复杂度为 O（n+n^2+e），其中对邻接矩阵的初始化需要耗费 O（n^2）的时间

有向网：



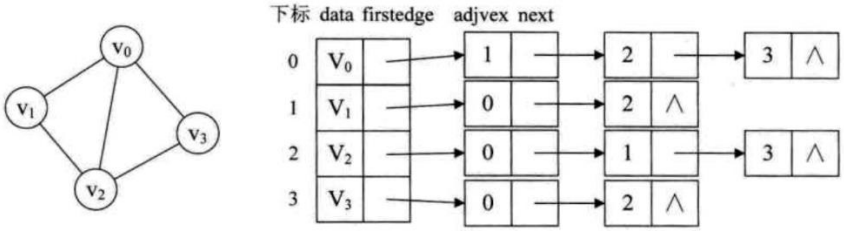
邻接矩阵对于边数相对于顶点较少的图（稀疏图），是极为浪费空间的，所以出现邻接表

4、邻接表

数组与链表相结合的存储方法称为邻接表（adjacent list）

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。另外，对于顶点数组中，每个数据元素还需要存储指向第一个邻接点的指针，以便于查找该顶点的边信息

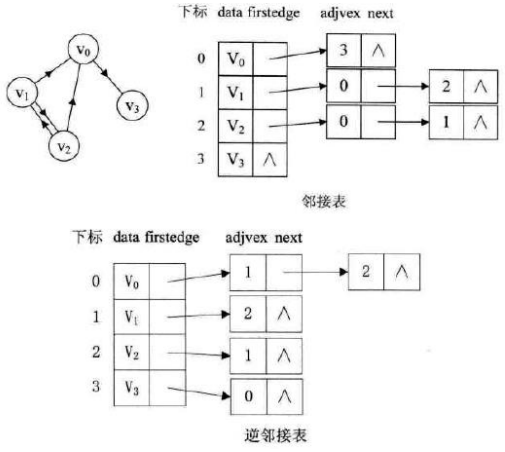
图中每个顶点 vi 的所有邻接点构成一个线性表，由于邻接点的个数不定，所以用单链表存储，无向图称为顶点 vi 的边表，有向图则称为顶点 vi 作为弧尾的出边表



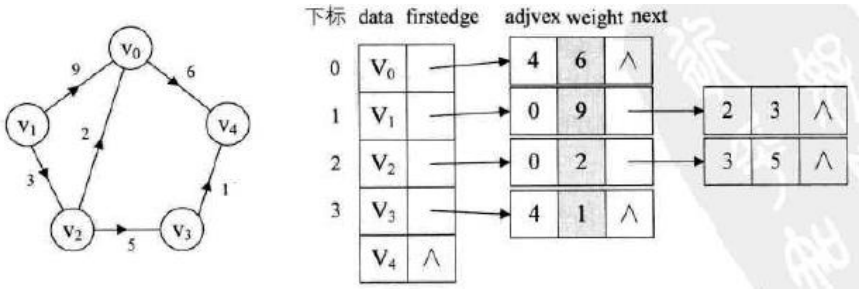
从图中我们知道，顶点表的各个结点由 **data** 和 **firstedge** 两个域表示，**data** 是数据域，存储顶点的信息，**firstedge** 是指针域，指向边表的第一个结点，即此顶点的第一个邻接点。边表结点由 **adjvex** 和 **next** 两个域组成。**adjvex** 是邻接点域，存储某顶点的邻接点在顶点表中的下标，**next** 则存储指向边表中下一个结点的指针。比如 v_1 顶点与 v_0、v_2 互为邻接点，则在 v_1 的边表中，**adjvex** 分别为 v_0 的 0 和 v_2 的 2。

这样的结构，对于我们要获得图的相关信息也是很方便的。比如我们要想知道某个顶点的度，就去查找这个顶点的边表中结点的个数。若要判断顶点 v_i 到 v_j 是否存在边，只需要测试顶点 v_i 的边表中 **adjvex** 是否存在结点 v_j 的下标 j 就行了。若求顶点的所有邻接点，其实就是对此顶点的边表进行遍历，得到的 **adjvex** 域对应的顶点就是邻接点。

若是有向图，邻接表结构是类似的，但我们是以顶点为弧尾来存储边表的，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，即对每个顶点 vi 都建立一个链接为 vi 为弧头的表**



对于带权值的网图，可以在边表结点定义中再**增加一个 weight 的数据域**，存储权值信息即可



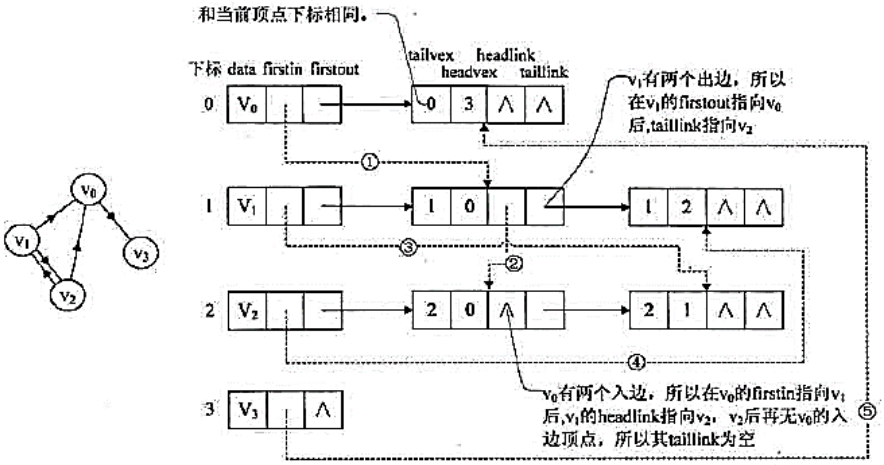
如果需要计算入度和出度都很方便的结构，可以采用十字链表：

十字链表顶点表结构：

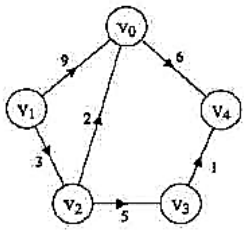
data	firstin	firstout

十字链表边表结点结构：

tailvex	headvex	headlink	taillink



边集数组：



顶点数组：

V0	V1	V2	V3	V4

边数组：

	begin	end	weight
edges[0]	0	4	6
edges[1]	1	0	9
edges[2]	1	2	3
edges[3]	2	3	5
edges[4]	3	4	1
edges[5]	2	0	2

图 7-4-14

定义的边数组结构如表 7-4-4 所示。

表 7-4-4

begin	end	weight

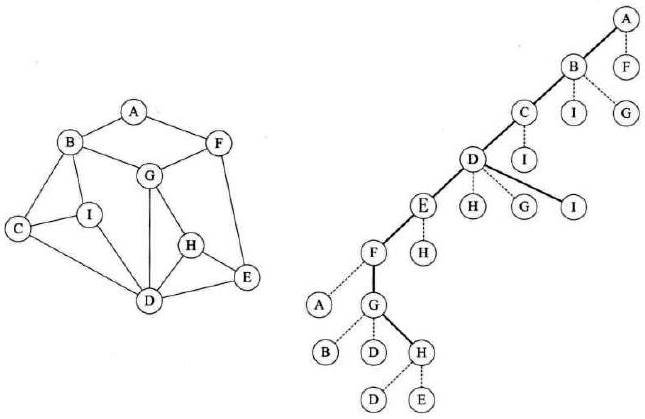
其中 begin 是存储起点下标，end 是存储终点下标，weight 是存储权值。

5、图的遍历

从图中某一顶点出发访遍图中其余顶点，且使每一个顶点仅被访问一次，这一过程就叫做图的遍历（Traversing Graph）

6、深度优先遍历（DFS）

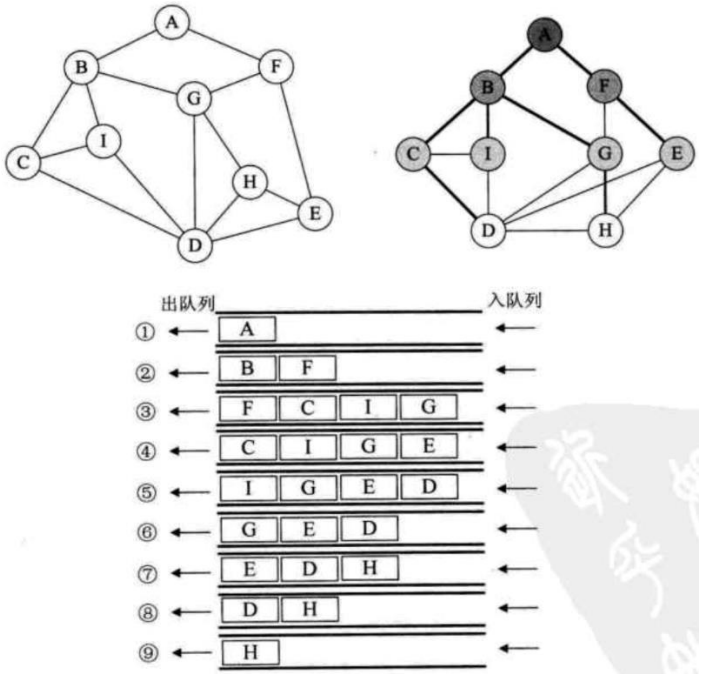
从图中某个顶点 v 出发，访问此顶点，然后从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到，若图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止：



图的深度优先遍历类似树的前序遍历

7、广度优先遍历（BFS）

图的广度优先遍历就类似于树的层序遍历，如下图所示：



8、图的两种遍历方式的比较

两者在时间复杂度上是一样的，不同之处仅仅在于对顶点访问的顺序不同，可见两者在全图遍历上是没有优劣之分的，只是视不同的情况选择不同的算法。

不过，深度优先更适合目标比较明确，以找到目标为主要目的的情况，而广度优先更适合在不断扩大遍历范围时找到相对最优解的情况

9、最小生成树

我们把构造连通网的最小代价生成树称为**最小生成树**（Minimum Cost Spanning Tree）

找连通网的最小生成树，经典的有两种算法，**普利姆算法**和**克鲁斯卡尔算法**
(https://www.cnblogs.com/biyeymyhjob/archive/2012/07/30/2615542.html)

10、普利姆（Prim）算法 时间复杂度 O(n^2)

算法思路：

以某顶点为起点，逐步找各顶点上最小权值的边来构建最小生成树

算法步骤

输入：一个加权连通图，其中顶点集合为 V，边集合为 E；

初始化：Vnew = {x}，其中 x 为集合 V 中的任一节点（起始点），Enew = {},为空；

重复下列操作，直到 Vnew = V：

在集合 E 中选取权值最小的边<u, v>，其中 u 为集合 Vnew 中的元素，而 v 不在 Vnew 集合当中，并且 v∈V（如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；

将 v 加入集合 Vnew 中，将<u, v>边加入集合 Enew 中；

输出：使用集合 Vnew 和 Enew 来描述所得到的最小生成树。

11、克鲁斯卡尔（Kruskal）算法

算法思路：

因为权值是在边上，所以直接去找最小权值的边来构建生成树，只不过构建时要考虑是否会形成环路而已

算法步骤：

先构造一个只含 n 个顶点、而边集为空的子图，把子图中各个顶点看成各棵树上的根结点，之后，从网的边集 E 中选取一条权值最小的边，若该条边的两个顶点分属不同的树，则将其加入子图，即把两棵树合成一棵树，反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之。依次类推，直到森林中只有一棵树，也即子图中含有 n-1 条边为止。

12、Prim 算法和 Kruskal 算法的对比

对比两个算法，克鲁斯卡尔算法主要是针对边来展开，边数少时效率会非常高，所以堆于稀疏图有很大的优势；而普利姆算法对于稠密图，即边数非常多的情况会更好一些。

13、最短路径

对于网图来说，最短路径，是指两顶点之间经过的边上权值之和最少的路径，并且我们称路径上的第一个顶点是源点，最后一个顶点是终点

主要有两种求最短路径的算法：迪杰斯特拉算法和

14、迪杰斯特拉（Dijkstra）算法

算法步骤：

G={V,E}

1. 初始时令 S={V0},T=V-S={其余顶点}，T 中顶点对应的距离值

若存在<V0,Vi>，d(V0,Vi)为<V0,Vi>弧上的权值

若不存在<V0,Vi>，d(V0,Vi)为∞

2. 从 T 中选取一个与 S 中顶点有关联边且权值最小的顶点 W，加入到 S 中

3. 对其余 T 中顶点的距离值进行修改：若加进 W 作中间顶点，从 V0 到 Vi 的距离值缩短，则修改此距离值

重复上述步骤 2、3，直到 S 中包含所有顶点，即 W=Vi 为止

15、弗洛伊德（Floyd）算法（动态规划）

算法步骤：

1，从任意一条单边路径开始。所有两点之间的距离是边的权，如果两点之间没有边相连，则权为无穷大。

2，对于每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从 u 到 w 再到 v 比已知的路径更短。如果是更新它。

把图用邻接矩阵 G 表示出来，如果从 Vi 到 Vj 有路可达，则 G[i][j]=d，d 表示该路的长度；否则 G[i][j]=无穷大。定义一个矩阵 D 用来记录所插入点的信息，D[i][j]表示从 Vi 到 Vj 需要经过的点，初始化 D[i][j]=j。把各个顶点插入图中，比较插点后的距离与原来的距离，G[i][j] = min(G[i][j], G[i][k]+G[k][j])，如果 G[i][j]的值变小，则 D[i][j]=k。在 G 中包含有两点之间最短道路的信息，而在 D 中则包含了最短通路径的信息。

比如，要寻找从 V5 到 V1 的路径。根据 D，假如 D(5,1)=3 则说明从 V5 到 V1 经过 V3，路径为{V5,V3,V1}，如果 D(5,3)=3，说明 V5 与 V3 直接相连，如果 D(3,1)=1，说明 V3 与 V1 直接相连。

16、拓扑排序（例如选课-先修课系统）

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，我们称为 **AOV 网**（Activity On Vertex Network）

拓扑序列：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**

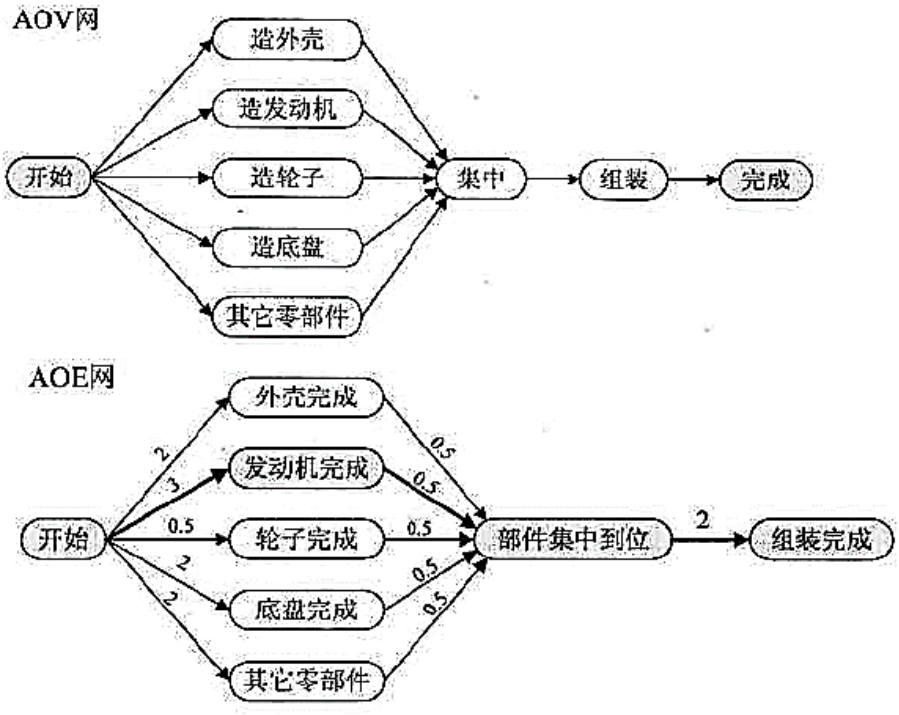
拓扑排序：其实就是对一个有向图构造拓扑序列的过程；构造时会有两个结果，如果此网的全部顶点都被输出，则说明它是不存在环（回路）的 AOV 网；如果输出顶点数少了，哪怕是少了一个，也说明这个网存在环（回路），不是 AOV 网。

17、拓扑排序算法

对 AOV 网进行拓扑排序的基本思路是：从 AOV 网中选择一个入度为 0 的顶点输出，然后删去此顶点，并删除以此顶点为尾的弧，继续重复此步骤，直到输出全部顶点或者 AOV 网中不存在入度为 0 的顶点为止。

18、关键路径（例如汽车各个部件分别生产-组装的活动）

在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，用边上的权值表示活动的持续时间，这种有向图的边表示活动的网，我们称之为 **AOE 网**（Activity On Edge Network）；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；正常情况下，AOE 网只有一个源点一个汇点



我们把路径上各个活动所持续的时间之和称为**路径长度**，从源点到汇点具有最大长度的路径叫**关键路径**，在关键路径上的活动叫**关键活动**

19、关键路径算法

原理：

我们只需要找到所有活动的最早开始时间和最晚开始时间，并且比较它们，如果相等就意味着此活动是关键活动，活动间的路径为关键路径。如果不等，则就不是。为此，我们需要定义如下几个参数：

事件的**最早发生时间 etv**（earliest time of vertex）：即顶点 Vk 的最早发生时间

事件的**最晚发生时间 ltv**（latest time of vertex）：即顶点 Vk 的最晚发生时间，也就是每个顶点对应的事件最晚需要开始的时间，超出此时间将会延误整个工期

活动的**最早开工时间 ete**（earliest time of edge）：即弧 ak 的最早发生时间

活动的**最晚开工时间 lte**（latest time of edge）：即弧 ak 的最晚发生时间，也就是不推迟工期的最晚开工时间

我们是由 1 和 2 可以求得 3 和 4，然后再根据 ete[k]是否与 lte[k]相等来判断 ak 是否是关键活动

20、总结

图的存储结构

邻接矩阵　　邻接表　　边集数组

十字链表　　邻接多重表

八、查找

1、查找概论

查找表（Search Table）是由同一类型的数据元素（或记录）构成的集合

关键字（Key）是数据元素中某个数据项的值，又称为**键值**

若此关键字可以唯一地标识一个记录，则称此关键字为**主关键字**（Primary Key）

那些可以识别多个数据元素（或记录）的关键字，我们称为**次关键字**（Secondary Key）

查找（Searching）就是根据给定的某个值，在查找表中确定一个其关键字等于给定值得数据元素（或记录）

2、查找表操作方式

分为两大种：静态查找表和动态查找表

静态查找表（Static Search Table）：只作查找操作的查找表。它的主要操作有：

- 查询某个“特定的”数据元素是否在查找表中
- 检索某个“特定的”数据元素和各种属性

动态查找表（Dynamic Search Table）：在查找过程中同时插入查找表中不存在的数据元素，或者从查找表中删除已经存在的某个数据元素。显然动态查找表的操作就是两个：

- 查找时插入数据元素
- 查找时删除数据元素

3、顺序查找

顺序查找（Sequential Search）又叫线性查找，是最基本的查找技术，它的查找过程是：

从表中第一个（或最后一个）记录开始，逐个进行记录的关键字和给定值比较，若某个记录的关键字和给定值相等，则查找成功，找到所查的记录；如果直到最后一个（或第一个）记录，其关键字和给定值比较都不等时，则表中没有所查的记录，查找不成功。时间复杂度 O（n）

(哨兵法，减少了每次和 n 的比较，效率大大提高)

```
int Sequential_Search(int *a,int n,int key)
{
    int i;
    for(i=1;i<=n;i++)
    {
        if (a[i]==key)
            return i;
    }
    return 0;
}
```

```
/* 有哨兵顺序查找 */
int Sequential_Search2(int *a,int n,int key)
{
    int i;
    a[0]=key;  /* 设置a[0]为关键字值，我们称之为“哨兵” */
    i=n;       /* 循环从数组尾部开始 */
    while(a[i]!=key)
    {
        i--;
    }
    return i;  /* 返回0则说明查找失败 */
}
```

4、二分查找

折半查找（Binary Search）技术，又称为二分查找。它的前提是线性表中的记录必须是关键码有序（通常从小到大有序），线性表必须采用顺序存储。折半查找的基本思想是：

在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键字相等，则查找成功；若给定值小于中间记录的关键字，则在中间记录的左半区继续查找；若给定值大于中间记录的关键字，则在中间记录的右半区继续查找。不断重复上述过程，直到查找成功，或所有查找区域无记录，查找失败为止。时间复杂度 O（logn）

5、插值查找

插值查找（Interpolation Search）是根据要查找的关键字 key 与查找表中最大最小记录的关键字比较后的查找方法，其核心就在于插值的计算公式(key-a[low])/(a[high]-a[low])

把二分法中的 mid =low + (high - low)/2

改为：mid = low + (high – low) * (key-a[low])/(a[high]-a[low])

对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好得多

6、斐波那契查找算法

算法核心：

当 key=a[mid]时，查找就成功

当 key<a[mid]时，新范围是第 low 个到第 mid-1 个，此时范围个数为 F[k-1]-1 个

当 key>a[mid]时，新范围是第 m+1 个到第 high 个，此时范围个数为 F[k-2]-1 个

7、三种查找算法的比较

折半查找是进行加法与除法运算(mid=(low+high)/2)，插值查找进行复杂的四则运算(mid=low+(high-low)*(key-a[low])/(a[high]-a[low]))，而斐波那契查找只是最简单加减法运算(mid=low+F[k-1]-1)，在海量数据的查找过程中，这种细微的差别可能会影响最终的查找效率

8、线性索引

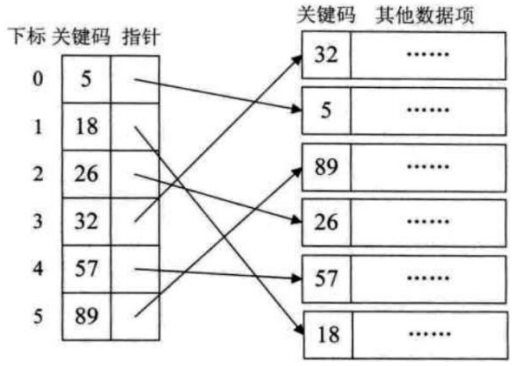
索引就是把一个关键字与它对应的记录相关联的过程

所谓**线性索引**就是将索引项集合组织为线性结构，也称为**索引表**

三种线性索引：稠密索引、分块索引和倒排索引

9、稠密索引

稠密索引是指在线性索引中，将数据集中的每个记录对应一个索引项



对于稠密索引这个索引表来说，索引项一定是按照关键码有序的排列

索引项有序也就意味着，我们要查找关键字时，可以用到折半、插值、斐波那契等有序查找算法，大大提高了效率

10、分块索引

分块有序，是把数据集的记录分成了若干块，并且这些块需要满足两个条件：

块内无序：当然如果能够让块内有序对查找来说更理想

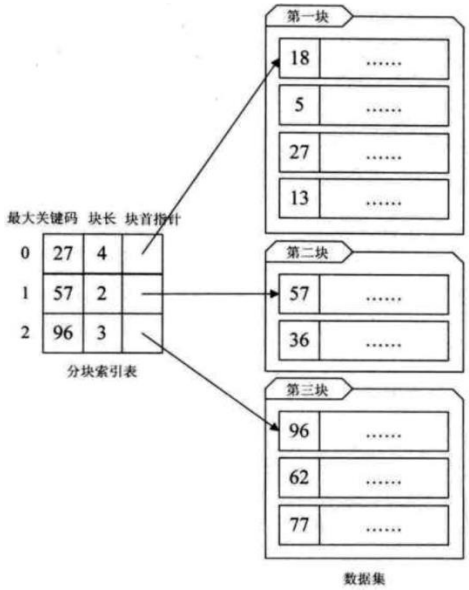
块间有序：只有块间有序，才有可能在查找时带来效率

分块索引的索引项结构分三个数据项：

最大关键码，它存储每一块中的最大关键字，这样的好处就是可以使得在它之后的下一块中的最小关键字也能比这一块最大的关键字要大

存储了**块中的记录个数**，以便于循环时使用

用于**指向块首数据元素的指针**，便于开始对这一块中记录进行遍历



分块索引表中查找的步骤：

在分块索引表中查找要查关键字所在的块，可以利用折半、插值等算法

根据块首指针找到响应的块，并在块中顺序查找关键码。因为块中可以是无序的，因此只顺序查找

11、倒排索引

记录号表存储具有相同次关键字的所有记录的记录号（可以是指向记录的指针或者是该记录的主关键字），这样的索引方法就是倒排索引（inverted index）

倒排索引的优点就是查找记录非常快，基本等于生成索引表后，查找时都不用去读取记录，就可以得到结果。但它的缺点是这个记录号不定长

12、二叉排序树

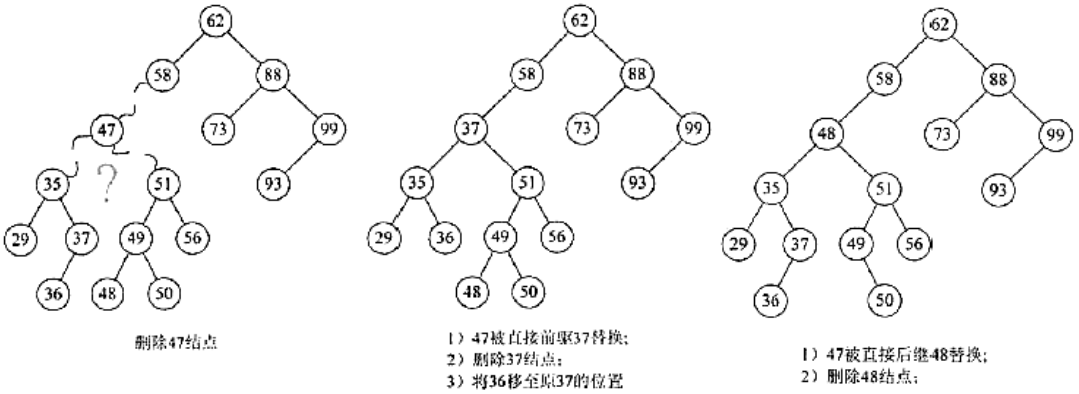
二叉排序树（Binary Sort Tree），又称为二叉查找树。它或者是一棵空树，或者是具有下列性质的二叉树。

特点：中序遍历是升序序列（但中序遍历是升序的不一定是 BST）

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结构的值
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值
- 它的左、右子树也分别为二叉排序树

如果我们希望对一个集合按二叉排序树查找，最好是把它构建成一棵平衡的二叉排序树。

在 BST 中删除一个有左右孩子的结点，可以找到离被删结点最近的结点，把这个点移过去，再做一些调整。如下图，可以选择 37 或 48



BST 查找时间复杂度 O（logn）但 BST 有可能是斜树（全在一边）此时等同于顺序查找（复杂度 O（n）），所以我们希望它是尽可能平衡的。

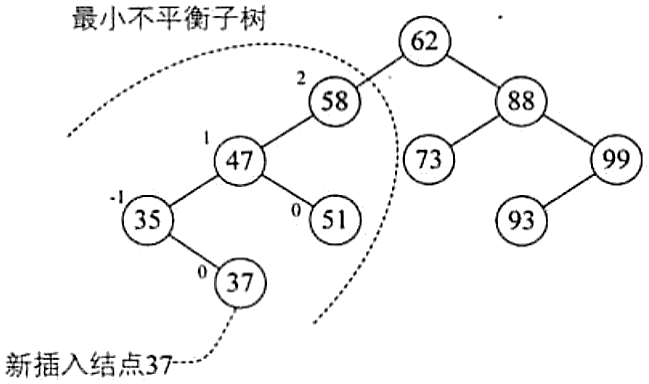
13、平衡二叉树（AVL 树）

平衡二叉树（Self-Balancing Binary Search Tree 或 Height-Balanced Binary Search Tree），是一种二叉排序树，其中每一个节点的左子树和右子树的高度差至多等于 1

我们将二叉树上结点的左子树深度减去右子树深度的值称为**平衡因子** BF（Balance Factor），那么平衡二叉树上所有结点的平衡因子只可能是-1，0，1

14、最小不平衡子树

距离插入结点最近的，且平衡因子的绝对值大于 1 的结点为根的子树，我们称为**最小不平衡子树**



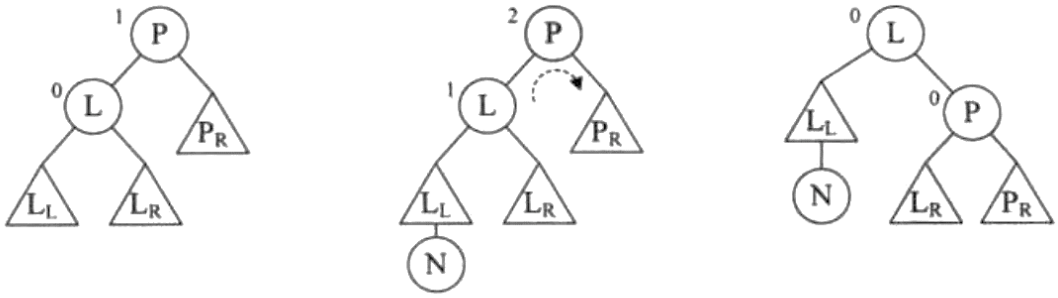
如上图所示，当新插入结点 37 时，距离它最近的平衡因子绝对值超过 1 的结点是 58（即它的左子树高度 2 减去右子树高度 0），所以从 58 开始以下的子树为最小不平衡子树

15、平衡二叉树实现算法

算法原理：

基本思想就是在构建二叉排序树的过程中，每当插入一个结点时，先检查是否因插入而破坏了树的平衡性，若是，则找出最小不平衡子树。在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行相应地旋转，使之成为新的平衡子树。

右旋操作：



左旋和右旋代码是对称的。AVL 的查找、插入和删除时间复杂度都是 O（logn）

16、多路查找树（B 树）

多路查找树（multi-way search tree），其每一个结点的孩子数可以多于两个，且每一个结点处可以存储多个元素。四种特殊形式：2-3 树，2-3-4 树，B 树，B+树

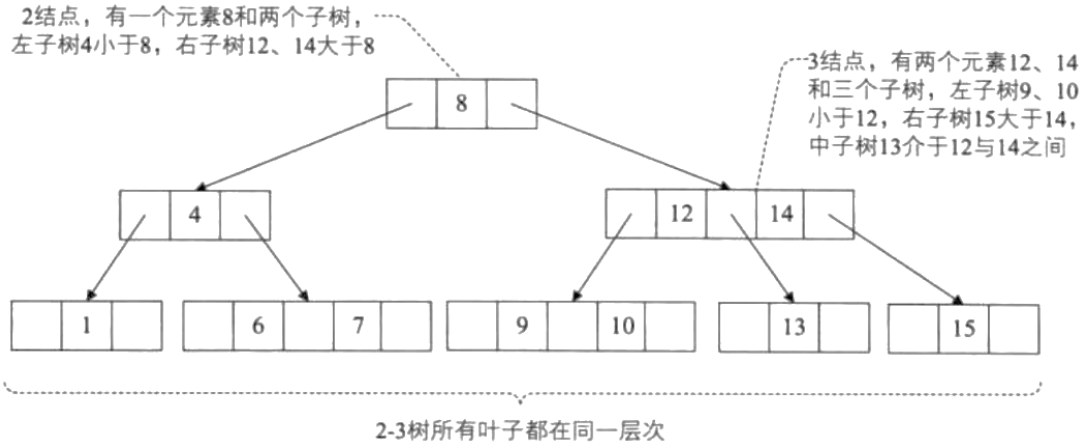
17、2-3 树

2-3 树是这样的一棵多路查找树：其中的每一个结点都具有两个孩子（我们称它为 2 结点）或三个孩子（我们称它为 3 结点）。

一个 2 结点包含一个元素和两个孩子（或没有孩子），且与二叉排序树类似

一个 3 结点包含一小一大两个元素和三个孩子（或没有孩子），左子树包含小于较小元素的元素，右子树包含大于较大元素的元素，中间子树包含介于两元素之间的元素

2-3 树中所有的叶子都在同一层次上



18、2-3 树的插入实现

可分为三种情况：

对于空树，插入一个 2 结点即可，这很容易理解

插入结点到一个 2 结点的叶子上。由于其本身就只有一个元素，所以只需要将其升级为 3 结点即可

要往 3 结点中插入一个新元素。因为 3 结点本身已经是 2-3 树的结点最大容量（已经有两个元素），因此就需要将其拆分，且将树中两元素或插入元素的三者中选择其一向上移动一层

19、2-3-4 树

就是 2-3 树的概念扩展，包括了 4 结点的使用。一个 4 结点包含小中大三个元素和四个孩子（或没有孩子），一个 4 结点要么没有孩子，要么具有 4 个孩子。如果某个 4 结点有孩子的话，从左到右按照由小到大的顺序排列

20、B 树

B 树（B-tree）是一种平衡的多路查找树，2-3 树和 2-3-4 树都是 B 树的特例。结点最大的孩子数目称为 **B 树的阶**（order），因此，2-3 树是 3 阶 B 树，2-3-4 树是 4 阶 B 树

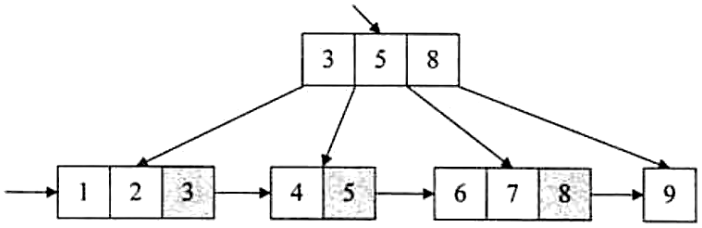
一个 m 阶的 B 树具有如下属性：

- 如果根结点不是叶节点，则其至少有两棵子树

- 每一个非根的分支结点都有 k-1 个元素和 k 个孩子，其中⌈m/2⌉≤k≤m。每一个叶子节点 n 都有 k-1 个元素，其中⌈m/2⌉≤k≤m
- 所有叶子结点都位于同一层次
- 所有分支结点包含下列信息数据（n,A0,K1,A1,K2,A2,...,Kn,An），其中：Ki(i=1,2,...,n)为关键字，且 Ki<Ki+1(i=1,2,...,n-1)；Ai(i=0,2,...,n)为指向子树根结点的指针，且指针 A(i-1)所指子树中所有结点的关键字均小于 Ki(i=1,2,...,n),An 所指子树中所有结点的关键字均大于 Kn，n·(⌈m/2⌉-1≤n≤m-1)为关键字的个数（或 n+1 为子树的个数）

21、B+树

在 B+树中，出现在分支结点中的元素会被当作它们在该分支结点位置的中序后继者（叶子节点）中再次列出。另外，每一个叶子结点都会保存一个指向后一叶子结点的指针



一棵 m 阶的 B+树和 m 阶的 B 树的差异在于：

有 n 棵子树的结点中包含有 n 个关键字

所有的叶子结点包含全部关键字的信息，及指向含这些关键字记录的指针，叶子结点本身依关键字的大小自小而大顺序链接

所有分支结点可以看成是索引，结点中仅含有其子树中的最大（或最小）关键字

如果我们是需要从最小关键字进行从小到大的顺序查找，我们就可以从最左侧的叶子结点出发，不经过分支结点，而是沿着指向下一叶子结点的指针就可遍历所有的关键字

22、散列表（哈希表）

散列技术是在记录的存储位置和它的关键字之间建立一个确定的对应关系 f，使得每个关键字 key 对应一个存储位置 f(key)，散列技术既是一种存储方法，也是一种查找方法

散列技术最适合的求解问题是查找与给定值相等的记录

f 称为散列函数，又称为**哈希（Hash）函数**

采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为**散列表或哈希表**（Hash table）

散列过程有两步：

在存储时，通过散列函数计算记录的散列地址，并按此散列地址存储该记录

当查找记录时，我们通过同样的散列函数计算记录的散列地址，按此散列地址访问该记录

23、散列函数构造方法

- 直接定址法
  取关键字的某个线性函数值为散列地址
- 数字分析法
  如果我们的关键字是位数较多的数字，可以对数字进行翻转、右环位移、左环位移、甚至前两数与后两数叠加等方法，合理地将关键字分配到散列表的各位置
- 平方取中法
  假设关键字是 1234，那么它的平方就是 1522756，再抽取中间的 3 位就是 227，用做散列地址。平方取中法比较适合于不知道关键字的分布，而位数又不是很大的情况
- 折叠法
  将关键字从左到右分割成位数相等的几部分，然后将这几部分叠加求和，并按散列表表长，取后几位作为散列地址.折叠法事先不需要知道关键字的分布，适合关键字位数较多的情况
  比如我们的关键字是 9876543210，散列表表长为三位，我们将它分为四组，
  987|654|321|0，然后将它们叠加求和 987+654+321+0=1962，再求后 3 位得到散列地址为 962
- 除留余数法（最常用）
  对关键字直接取模，也可在折叠、平方取中后再取模，对于散列表长为 m 的散列函数公式为：
  f(key)=key mod p(p≤m)
  根据前辈们的经验，若散列表表长为 m，通常 p 为小于或等于表长（最好接近 m）的最小质数或不包含小于 20 质因子的合数
- 随机数法
  选择一个随机数，取关键字的随机函数值为它的散列地址

24、采用不同的散列函数应该考虑的因素

- 计算散列地址所需的时间
- 关键字的长度
- 散列表的大小
- 关键字的分布情况
- 记录查找的频率

25、处理散列冲突的方法

开放定址法

一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入，公式是：

fi(key)=（f(key)+di）MOD m(di=1,2,3,……,m-1)

这种解决冲突的开放定址法称为**线性探测法**

如果 di 改进为正负两类值，等于是可以双向寻找到可能的空位置，可以不让关键字都聚集在某一块区域。我们称这种方法为**二次探测法**

如果 di 采用随机函数计算得到，我们称之为**随机探测法**

再散列函数法

我们事先准备多个散列函数，每当发生散列地址冲突时，就换一个散列函数计算，相信总会有一个可以把冲突解决掉

链地址法

将所有关键字为同义词的记录存储在一个单链表红，我们称这种表尾同义词子表，在散列表中只存储所有同义词子表的头指针

公共溢出区法

凡是冲突的都将它们存储到溢出表中

九、排序

排序的稳定性：

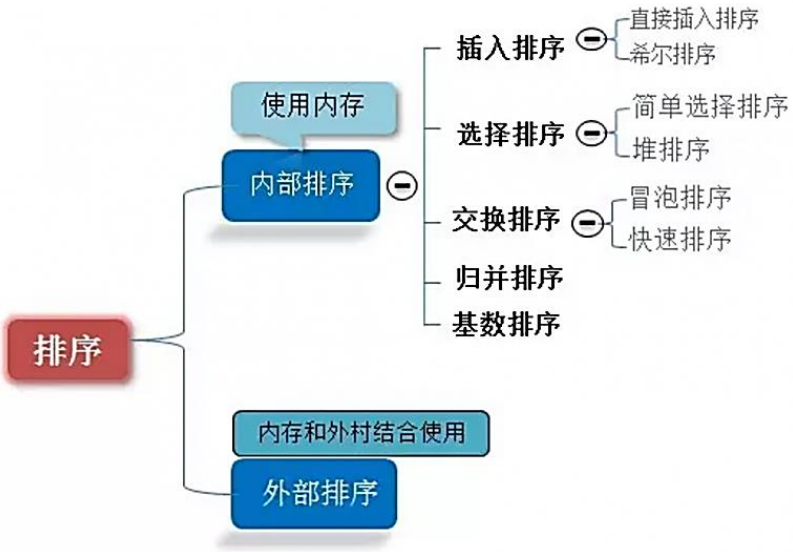
- 稳定：如果 a 原本在 b 前面，而 a=b，排序之后 a 仍然在 b 的前面；
- 不稳定：如果 a 原本在 b 的前面，而 a=b，排序之后 a 可能会出现在 b 的后面；

内排序与外排序：

- 内排序：所有排序操作都在内存中完成；
- 外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；排序算法对比：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

n：数据规模 k:“桶”的个数 In-place：占用常熟内存，不占额外内存 Out-place：占用额外内存



1.冒泡排序

（1）算法简介

冒泡排序（bubble Sort）它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

（2）算法具体描述

1. 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
3. 针对所有的元素重复以上的步骤，除了最后一个；
4. 重复步骤 1~3，直到排序完成。

   最佳情况：T(n)=O(n)=>当输入的数据已经是正序时

   最差情况：T(n)=O(n^2)=>当输入的数据是反序时

   平均情况：T(n)= O(n^2)

```
def bubble_sort(arr):
    count = len(arr)
    for i in range(0, count):
        for j in range(0, count-i-1):
            if arr[j] > arr[j+1]:
                temp = arr[j+1]
                arr[j+1] = arr[j]
                arr[j] = temp
    return arr
```

2.选择排序

（1）算法简介

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

（2）算法具体描述

1. 初始状态：无序区为 R[1..n]，有序区为空；
2. 第 i 趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为 R[1..i-1]和 R(i..n）。该趟排序从当前无序区中选出关键字最小的记录 R[k]，将它与无序区的第 1 个记录 R 交换，使 R[1..i]和 R[i+1..n)分别变为记录个数增加 1 个的新有序区和记录个数减少 1 个的新无序区；
3. n-1 趟结束，数组有序化了。

   最佳情况：T(n)= O(n^2)

   最差情况：T(n)= O(n^2)

   平均情况：T(n)= O(n^2)

```
def selection_sort(arr):
    count = len(arr)
    for i in range(0, count):
        minid = i
        for j in range(i+1, count):
            if arr[j] < arr[minid]:
                minid = j
        temp = arr[i]
        arr[i] = arr[minid]
        arr[minid] = temp
    return arr
```

3.插入排序

（1）算法简介

插入排序（Insertion-Sort）工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。实现上，通常采用 in-place 排序（只需用到 O(1)的额外空间），在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

（2）算法具体描述

1. 从第一个元素开始，该元素可以认为已经被排序；
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
4. 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置；
5. 将新元素插入到该位置后；
6. 重复步骤 2~5。

   最佳情况：T(n)=O(n)=>输入数组按升序排列

   最差情况：T(n)= O(n^2)=>输入数组按降序排列

   平均情况：T(n)= O(n^2)

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

4.希尔排序

（1）算法简介

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。

希尔排序的核心在于间隔序列的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。

（2）算法具体描述(**每次以一定步长(就是跳过等距的数)进行排序，直至步长为 1**)

1. 选择一个增量序列 t1，t2，…，tk，其中 ti>tj，tk=1；
2. 按增量序列个数 k，对序列进行 k 趟排序；
3. 每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

```
def shell_sort(arr):
    count = len(arr)
    gap = count/2
    while gap > 0:
        for i in range(gap, count):
            temp = arr[i]
            j = i
            while j >= gap and arr[j-gap] > temp:
                arr[j] = arr[j-gap]
                j -= gap
            arr[j] = temp
        gap = gap/2
    return arr
```

最佳情况：T(n)=O(nlog2n)

最差情况：T(n)=O(nlog2n)

平均情况：T(n)=O(nlogn)

5.归并排序

（1）算法简介

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为 2-路归并。

（2）算法具体描述

1. 把长度为 n 的输入序列分成两个长度为 n/2 的子序列；
2. 对这两个子序列分别采用归并排序；
3. 将两个排序好的子序列合并成一个最终的排序序列。

最佳情况：T(n)=O(n)

最差情况：T(n)=O(nlogn)

平均情况：T(n)=O(nlogn)

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)/2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left,right):
    l, r = 0, 0
    temp = []
    while l < len(left) and r <
len(right):
        if left[l] <= right[r]:
            temp.append(left[l])
            l += 1
        else:
            temp.append(right[r])
            r += 1
    while l < len(left):
        temp.append(left[l])
        l += 1
    while r < len(right):
        temp.append(right[r])
        r += 1
    return temp
```

6.快速排序

（1）算法简介

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，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。

（2）算法具体描述

1. 从数列中挑出一个元素，称为 “基准”（pivot）；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

   最佳情况：T(n)=O(nlogn)

   最差情况：T(n)=O(n^2)

   平均情况：T(n)=O(nlogn)

```
def quick_sort(list):
    #partition
    less = []
    pivotlist = []
    more = []
    if len(list) <= 1:
        return list
    pivot = list[0]
    for i in range(0, len(list)):
        if list[i] < pivot:
            less.append(list[i])
        elif list[i] == pivot:
            pivotlist.append(list[i])
        else:
            more.append(list[i])
    #recursion:
    less = quick_sort(less)
    more = quick_sort(more)
    return less + pivotlist + more
```

7.堆排序

（1）算法简介

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

（2）算法具体描述

1. 创建最大堆:将堆所有数据重新排序，使其成为最大堆
2. 最大堆调整:作用是保持最大堆的性质，是创建最大堆的核心子程序
3. 堆排序:移除位在第一个数据的根节点，并做最大堆调整的递归运算

   最佳情况：T(n)=O(nlogn)

   最差情况：T(n)=O(nlogn)

   平均情况：T(n)=O(nlogn)

```
def heap_sort(list):
    # 创建最大堆
    for start in range((len(list) - 2) // 2, -1, -1):
        sift_down(list, start, len(list) - 1)
    # 堆排序
    for end in range(len(list) - 1, 0, -1):
        list[0], list[end] = list[end], list[0]
        sift_down(list, 0, end - 1)
    return list
# 最大堆调整
def sift_down(lst, start, end):
    root = start
    while True:
        child = 2 * root + 1
        if child > end:
            break
        if child + 1 <= end and lst[child] < lst[child
+ 1]:
            child += 1
        if lst[root] < lst[child]:
            lst[root], lst[child] = lst[child], lst[root]
            root = child
        else:
            break
```