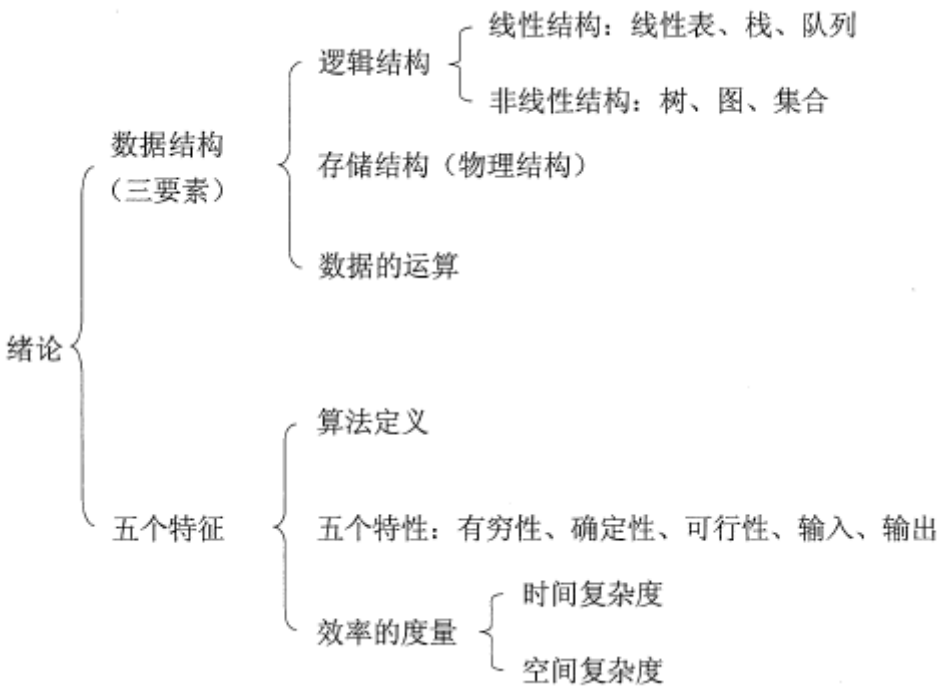
数据结构

第一章绪论



1.1 时间复杂度

一个语句的频度是指该语句在算法中被**重复执行**的次数。算法中所有语句的频度之和记为T(n), 它是该算法问题规模n 的函数，时间复杂度主要分析T(n) 的数量级。

（算法中基本运算（最深层循环内的语句）的频度与T(n) 同数量级，因此通常采用算法中基本运算的频度f(n)来分析算法的时间复杂度。

取f(n) 中随n 增长最快的项，将其系数置为1 作为时间复杂度的度量。例如， f(n) = an3 + bn2 + cn 的时向复杂度为O(n3)

算法的时间复杂度不仅依赖于**问题的规模n**, 也取决于**待输入数据的性质**）

1.2 空间复杂度

算法的空间复杂度S(n)定义为该算法所**耗费的存储空间**，它是问题规模n 的函数。S(n) = O(g(n))
（一个程序在执行时除需要存储空间来存放本身所用的指令、常数、变量和输入数据外，还需要一些对数据进行操作的工作单元和存储一些为实现计算所需信息的辅助空间。若输入数据所占空间只取决于问题本身，和算法无关，则只需分析除输入和程序之外的额外空间。
算法原地工作是指算法所需的辅助空间为常量，即O(1) 。）

1.3 数的存储结构

存储结构是指数据结构在**计算机**中的表示，也称**物理结构**，主要有以下4种：

顺序存储。

把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来体现。

【优点】是可以实现随机存取，每个元素占用最少的存储空间；

【缺点】是只能使用相邻的一整块存储单元，因此可能产生较多的外部碎片。

链式存储。

不要求逻辑上相邻的元素在物理位置上也相邻，**借助**指示元素存储地址的**指针**来表示元素之间的逻辑关系。

【优点】是不会出现碎片现象，能充分利用所有存储单元；

【缺点】是每个元素因存储指针而占用额外的存储空间，且只能实现顺序存取。

索引存储。

在存储元素信息的同时，还建立附加的索引表。索引表中的每项称为索引项，索引项的一般形式是（关键字，地址）。

【优点】是检索速度快；

【缺点】是附加的索引表额外占用存储空间。另外，增加和删除数据时也要修改索引表，因而会花费较多的时间。

散列存储。

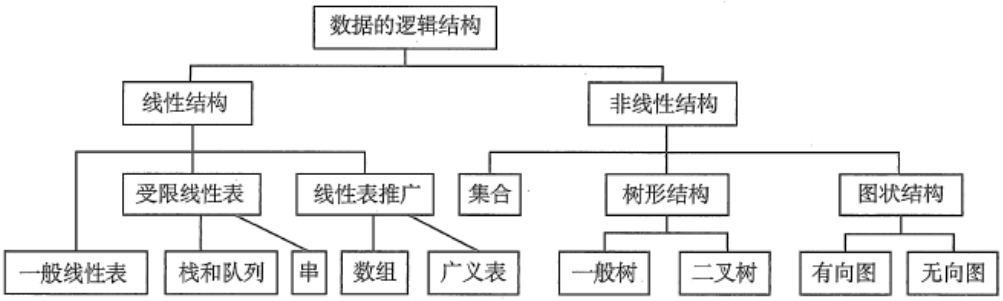
根据元素的关键字直接计算出该元素的存储地址，又称哈希(Hash) 存储。

【优点】是检索、增加和删除结点的操作都很快；

【缺点】是若散列函数不好，则可能出现元素存储单元的冲突，而解决冲突会增加时间和空间开销。

1.4 数的逻辑结构

指的是**数据元素**之间**逻辑关系**，与数的存储结构无关，是独立于计算机的，以下是分类图。



1.5 用循环比递归的效率高吗?

循环和递归两者是可以互换的，不能决定性的说循环的效率比递归高。

递归

【优点】：代码简洁清晰，容易检查正确性；

【缺点】：当递归调用的次数较多时，要增加额外的堆栈处理，有可能产生堆栈溢出的情况，对执行效率有一定的影响。

循环

【优点】：结构简单，速度快；

【缺点】：它并不能解决全部问题，有的问题适合于用递归来解决不适合用循环。

1.6 贪心算法和动态规划以及分治法的区别?

贪心算法

就是做出在当前看来是最好的结果，它不从整体上加以考虑，也就是**局部最优解。**贪心算法从上往下，从顶部一步一步最优，得到最后的结果，它不能保证全局最优解，与贪心策略的选择有关。

动态规划

是把问题**分解成子问题**，这些子问题可能有重复，可以记录下前面子问题的结果防止重复计算。动态规划解决子问题，前一个子问题的解对后一个子问题产生一定的影响。在求解子问题的过程中**保留哪些有可能得到最优的局部解，丢弃其他局部解**，直到解决最后一个问题时也就是初始问题的解。动态规划是从下到上，一步一步找到**全局最优解。**（各子问题**重叠**）

分治法（divide-and-conquer）

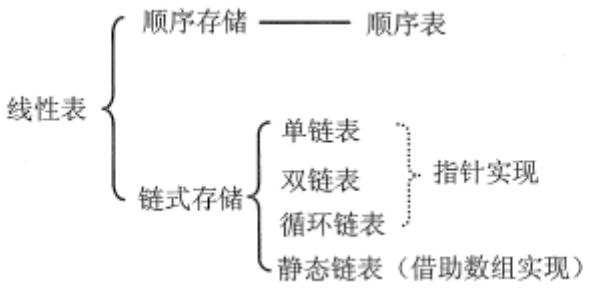
将原问题划分成**n个规模较小而结构与原问题相似的子问题**；递归地解决这些子问题，然后再合并其结果，就得到原问题的解。（各子问题**独立**）

分治模式在每一层递归上都有三个步骤：

分解（Divide）：将原问题分解成一系列子问题；
解决（conquer）：递归地解各个子问题。若子问题足够小，则直接求解；
合并（Combine）：将子问题的结果合并成原问题的解。

例如归并排序。

第二章线性表



2.1 顺序表和链表的比较

1.存取（读写）方式

顺序表可以顺序存取，也可以随机存取

链表只能从表头顺序存取元素。

2.逻辑结构与物理结构
采用顺序存储时，逻辑上相邻的元素，对应的物理存储位置也相邻。

采用链式存储时，逻辑上相邻的元素，物理存储位置则不一定相邻，对应的逻辑关系是通过指针链接来表示的。

3.查找、插入和删除操作
对于**按值查找**，顺序表无序时，两者的时间复杂度均为O(n); 顺序表有序时，可采用折半查找，此时的时间复杂度为O(log2n) 。

对于**按序号查找**，顺序表支持随机访问，时间复杂度仅为0(1), 而链表的平均时间复杂度为O(n) 。

顺序表的插入、删除操作，平均需要移动**半个表长的元素。**

链表的插入、删除操作，只需**修改相关结点的指针域**即可。由于链表的每个结点都带有指针域，故而存储密度不够大。

4.空间分配

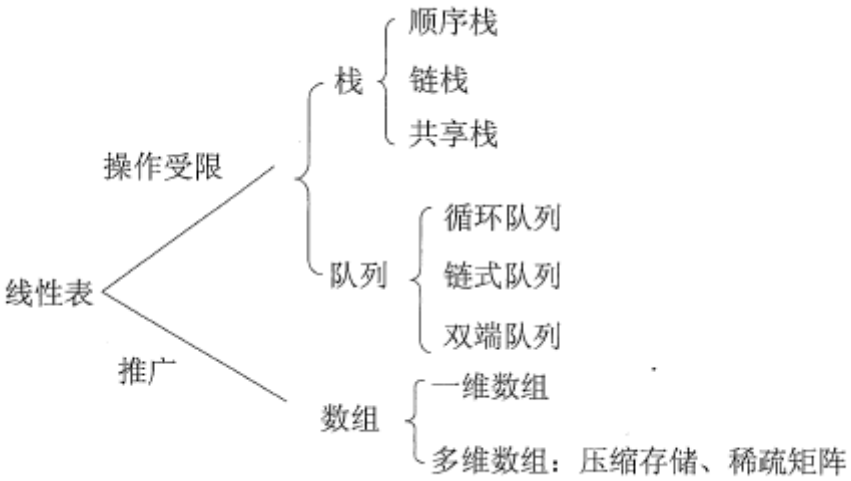
顺序存储在**静态存储分配**情形下，一旦存储空间装满就不能扩充，若再加入新元素，则会出现内存溢出，因此需要预先分配足够大的存储空间。预先分配过大，可能会导致顺序表后部大量闲置；预先分配过小，又会造成溢出。**动态存储分配**虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且若内存中没有更大块的连续存储空间，则会导致分配失败。

链式存储的结点空间只在**需要时申请分配**，只要内存有空间就可以分配，操作灵活、高效。

2.2 头指针和头结点的区别?

> **头指针**：是指向第一个节点存储位置的指针，具有标识作用，头指针是链表的必要元素，无论链表是否为空，头指针都存在。
> **头结点**：是放在第一个元素节点之前，便于在第一个元素节点之前进行插入和删除的操作，头结点不是链表的必须元素，可有可无，头结点的数据域也可以不存储任何信息。

第三章栈和队列



3.1 栈和队列的区别?

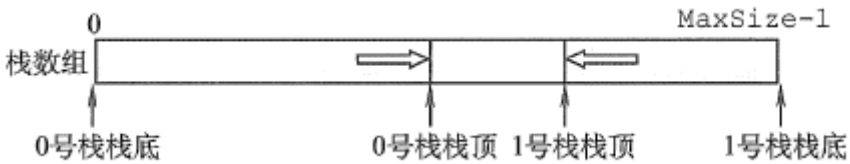
队列

允许在**一段进行插入另一端进行删除**的线性表。队列顾名思义就像排队一样，对于进入队列的元素按“**先进先出**”的规则处理，在表头进行删除在表尾进行插入。由于队列要进行频繁的插入和删除，一般为了高效，选择用**定长数组**来存储队列元素，在对队列进行操作之前要**判断队列是否为空或是否已满。**如果想要**动态长度也可以用链表**来存储队列，这时要记住队头和队尾指针的地址。

栈

只能在**表尾进行插入和删除**操作的线性表。对于插入到栈的元素按“**后进先出**”的规则处理，插入和删除操作都在栈顶进行，与队列类似一般用**定长数组**存储栈元素。由于进栈和出栈都是在栈顶进行，因此要有一个**size变量来记录当前栈的大小**，当进栈时size不能超过数组长度，size+1，出栈时栈不为空，size-1。

3.2 共享栈



利用**栈底位置相对不变**的特性，可以让两个顺序栈共享一个**一维数组**空间，将两个栈的**栈底**分别设置在共享空间的**两端**，两个**栈顶**向共享空间的**中间延伸。**这样能够更有效的利用存储空间，两个栈的空间相互调节，只有在**整个存储空间被占满时才发生上溢。**

3.3 如何区分循环队列是队空还是队满?

普通情况下，

循环队列队空和队满的判定条件是一样的，都是**Q.front == Q.rear。**

ps:队头指针指向第一个数；队尾指针指向最后一个数的下一个位置，即将要入队的位置。

> 方法一：**牺牲一个单元**来区分队空和队满，这个时候**(Q.rear+1)%MaxSize == Q.front**才是队满标志 。

> 方法二：类型中**增设表示元素个数的数据成员。**这样，队空的条件为Q.size == 0;队满的条件为Q.size == MaxSize。

3.4 栈在括号匹配中的算法思想

（1）出现的凡是“**左括号**”，则进栈；

（2）出现的是“**右括号**”，

首先检查**栈是否空**?

若栈空，则表明该“右括号”多余

否则和栈顶元素比较?

若相匹配，则栈顶“左括号出栈”

否则表明不匹配

（3）表达式**检验结束**时，

若栈空，则表明表达式中匹配正确

否则表明“左括号”有余；

3.5 栈在通过后缀表达式求值的算法思想?

顺序扫描表达式的每一项，然后根据它的类型做如下相应操作：

若该项是**操作数**，则将其压入栈中；

若该项是**操作符**,则连续从栈中退出两个操作数y 和x, 形成运算指令XY, 并将计算结果重新压入栈中。

当表达式的**所有项**都扫描并处理完后，栈顶存放的就是最后的计算结果。

表 3.1　后缀表达式 ABCD-*+EF/-求值的过程

步	扫描项	项类型	动　作	栈中内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	A B
4	C	操作数	进栈	A B C
5	D	操作数	进栈	A B C D
6	-	操作符	D、C 退栈，计算 C-D，结果 R_1 进栈	A B R_1
7	*	操作符	R_1、B 退栈，计算 B×R_1，结果 R_2 进栈	A R_2
8	+	操作符	R_2、A 退栈，计算 A+R_2，结果 R_3 进栈	R_3
9	E	操作数	进栈	R_3 E
10	F	操作数	进栈	R_3 E F
11	/	操作符	F、E 退栈，计算 E/F，结果 R_4 进栈	R_3 R_4
12	-	操作符	R_4、R_3 退栈，计算 R_3-R_4，结果 R_5 进栈	R_5

3.6 栈在递归中的应用?

递归是一种重要的程序设计方法。

简单地说，若在一个**函数、过程或数据结构**的定义中又**应用**了它**自身**，则这个函数、过程或数据结构称为是递归定义的，简称递归。
它通常把一个**大型的复杂问题**层层转化为一个**与原问题相似的规模较小的问题**来求解，

递归策略只需**少量的代码**就可以描述出解题过程所需要的**多次重复计算**，大大减少了程序的代码量。但在通常情况下，它的**效率**并**不是太高**。

将递归算法转换为非递归算法，通常需要借助**栈**来实现这种**转换**。

3.7 队列在层次遍历中的作用?

在信息处理中有一大类问题需要**逐层或逐行处理**。

这类问题的解决方法往往是在处理**当前层或当前行**时就对下**一层或下一行做预处理**，把处理顺序安排好，待当前层或当前行处理完毕，就可以处理下一层或下一行。使用队列是为了**保存下一步的处理顺序**。下面用二叉树层次遍历的例子，说明队列的应用。

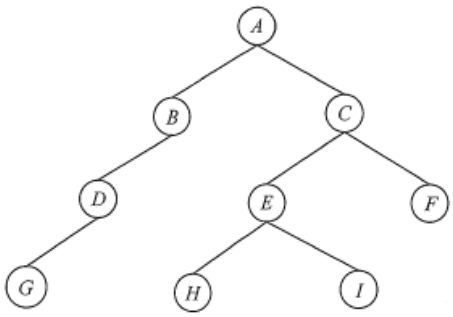


图 3.17　二叉树

序	说　明	队内	队　外
1	*A* 入	*A*	
2	*A* 出，*BC* 入	*BC*	*A*
3	*B* 出，*D* 入	*CD*	*AB*
4	*C* 出，*EF* 入	*DEF*	*ABC*
5	*D* 出，*G* 入	*EFG*	*ABCD*
6	*E* 出，*HI* 入	*FGHI*	*ABCDE*
7	*F* 出	*GHI*	*ABCDEF*
8	*GHI* 出		*ABCDEFGHI*

3.8 队列在计算机系统中的应用?

队列在计算机系统中的应用非常广泛，以下仅从两个方面来简述队列在计算机系统中的作用：

第一个方面是解决**主机与外部设备之间速度不匹配**的问题

第二个方面是解决由**多用户引起的资源竞争**问题。

> （对于第一个方面，仅以主机和打印机之间速度不匹配的问题为例做简要说明。主机输出数据给打印机打印，输出数据的速度比打印数据的速度要快得多，由于速度不匹配，若直接把输出的数据送给打印机打印显然是不行的。解决的方法是设置一个打印数据缓冲区，主机把要打印输出的数据依次写入这个缓冲区，写满后就暂停输出，转去做其他的事情。打印机就从缓冲区中按照

> 先进先出的原则依次取出数据并打印，打印完后再向主机发出请求。主机接到请求后再向缓冲区写入打印数据。这样做既保证了打印数据的正确，又使主机提高了效率。由此可见，打印数据缓冲区中所存储的数据就是一个队列。
>
> 对于第二个方面， CPU (即中央处理器，它包括运算器和控制器）资源的竞争就是一个典型的例子。在一个带有多终端的计算机系统上，有多个用户需要CPU 各自运行自己的程序，它们分别通过各自的终端向操作系统提出占用CPU 的请求。操作系统通常按照每个请求在时间上的先后顺序，把它们排成一个队列，每次把CPU 分配给队首请求的用户使用。当相应的程序运行结束或用完规定的时间间隔后，令其出队，再把CPU 分配给新的队首请求的用户使用。这样既能满足每个用户的请求，又使CPU 能够正常运行。）

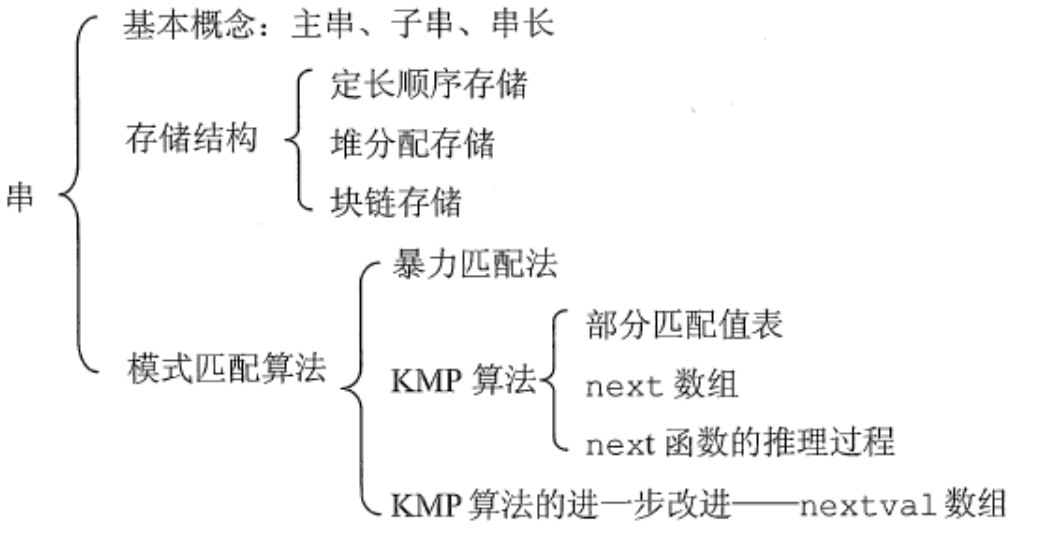
3.9 矩阵的压缩存储

数据结构中，提供针对某些**特殊矩阵**的压缩存储结构。这里所说的特殊矩阵，主要分为以下两类：

- 含有大量**相同数据元素**的矩阵，比如对称矩阵；
- 含有**大量 0 元素的矩阵**，比如稀疏矩阵、上（下）三角矩阵；

针对以上两类矩阵，数据结构的**压缩存储思想**是：矩阵中的**相同数据元素（包括元素 0）只存储一个。**

第四章串



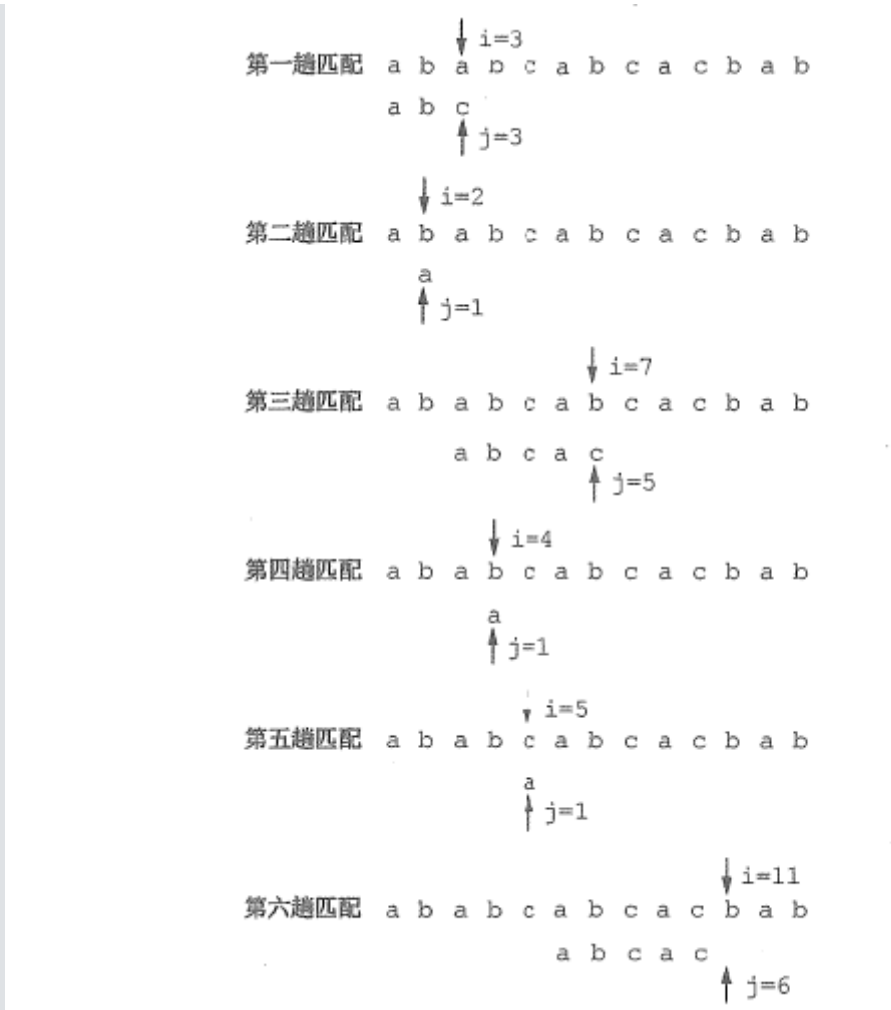
4.1 串的模式匹配

子串的定位操作通常称为串的模式匹配，求的是子串（常称模式串）在主串中的位置。

4.1.1暴力模式匹配算法

暴力模式匹配算法的思想是：从主串的第一个字符起，与子串的第一个字符比较，相等则继续比较；不等则从主串的下一个位置起，继续和子串开始比较，直到最后看是否匹配成功。

> 以下的子串为：‘abcac’:

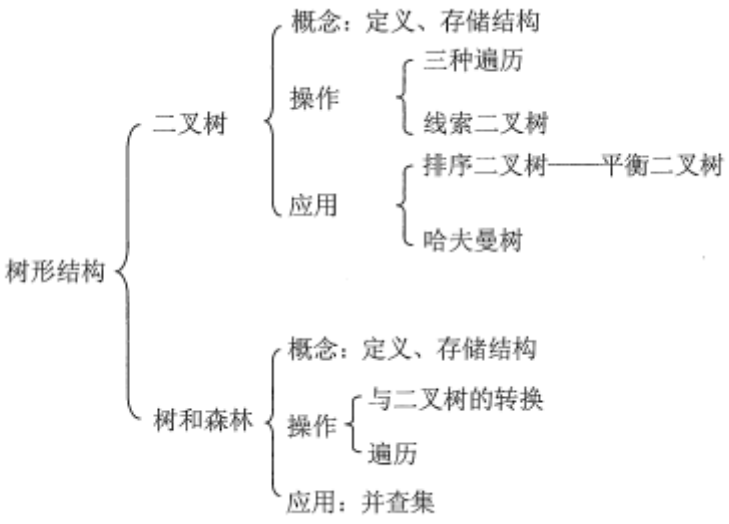


4.1.2 改进的模式匹配算法-----KMP算法

在**暴力匹配**中，每趟匹配失败都是模式后移一位再从头开始比较。而某趟已匹配相等的字符序列是模式的某个前缀，这种频繁的重复比较相当于模式串在不断地进行自我比较，这就是其低效率的根源。

因此，可以从分析模式本身的结构着手，**如果已匹配相等的前缀序列中有某个后缀正好是模式的前缀，那么就可以将模式向后滑动到与这些相等字符对齐的位置，主串i指针无须回溯，并继续从该位置开始进行比较。**而模式向后滑动位数的计算仅与模式本身的结构有关，与主串无关。

第五章树与二叉树



5.1 树与二叉树的相关概念?

树

是非线性结构，其元素之间有明显的层次关系。

在树的结构中，每个节点都只有一个前件称为**父节点**，没有前件的节点为树的**根节点**，简称为**树的根**；每个节点可以有多个后件称为**节点的子节点**，没有后件的节点称为**叶子节点**。

在树的结构中，一个节点所拥有的子节点个数称为该**节点的度**，树中最大的节点的度为**树的度**，树的最大的层次称为**树的深度**

二叉树

二叉树是另一种树形结构，其特点是每个结点**至多只有两棵子树**，并且二叉树的子树**有左右之分**，其次序不能任意颠倒。与树相似，二叉树也以递归的形式定义。二叉树是n (n >=0) 个结点的有限集合：

1)或者为**空二叉树**，即n=0 。
2)或者由**一个根结点和两个**互不相交的被称为根的**左子树和右子树组成**。左子树和右子树又分别是一棵二叉树。

二叉树是**有序树**，若将其左、右子树颠倒，则成为另一棵不同的二叉树。即使树中结点只有一棵子树，也要区分它是左子树还是右子树

满二叉树

满二叉树是指除了最后一层外其他节点均有两颗子树。

完全二叉树

完全二叉树是指除了最后一层外，其他任何一层的节点数均达到最大值，且最后一层也只是在最右侧缺少节点

二叉树的存储

二叉树可以用**链式**存储结构来存储，满二叉树和完全二叉树可以用**顺序**存储结构来存储

二叉树的遍历

二叉树有先序遍历（根左右），中序遍历（左根右）和后序遍历（左右根）；还有层次遍历，需要借助一个队列。

三种遍历算法中，递归遍历左、右子树的顺序都是固定的，只是访问根结点的顺序不同。不管采用哪种遍历算法，**每个结点都访问一次且仅访问一次**，故**时间复杂度都是O(n)** 。在递归遍历中，递归工作栈的栈深恰好为树的深度，所以在**最坏情况**下，二叉树是有n 个结点且深度为n 的单支树，遍历算法的**空间复杂度为O(n)** 。

5.2 如何由遍历序列构造一棵二叉树?

1)由二叉树的**先序序列和中序序列**可以唯一地确定一棵二叉树。

在先序遍历序列中，第一个结点一定是二叉树的根结点；

而在中序遍历中，根结点必然将中序序列分割成两个子序列，前一个子序列是根结点的**左子树的中序序列**，后一个子序列是根结点的**右子树的中序序列**。根据这两个子序列，在**先序序列**中找到对应的左子序列和右子序列。在先序序列中，左子序列的第一个结点是左子树的根结点，右子序列的第一个结点是右子树的根结点。如此递归地进行下去，便能唯一地确定这棵二叉树。

2)由二叉树的**后序序列和中序序列**也可以唯一地确定一棵二叉树。
因为后序序列的最后一个结点就如同先序序列的第一个结点，可以将中序序列分割成两个子序列，然后采用类似的方法递归地进行划分，进而得到一棵二叉树。

3）由二叉树的**层序序列和中序序列**也可以唯一地确定一棵二叉树。需要注意的是，若只知道二叉树的先序序列和后序序列，则无法唯一确定一棵二叉树。

5.3 线索二叉树的概念?

对于**n个结点**的二叉树，在二叉链存储结构中有**n+1个空链域**，利用这些空链域存放在某种遍历次序下该结点的**前驱结点和后继结点**的指针，这些指针称为**线索**，加上线索的二叉树称为**线索二叉树。**

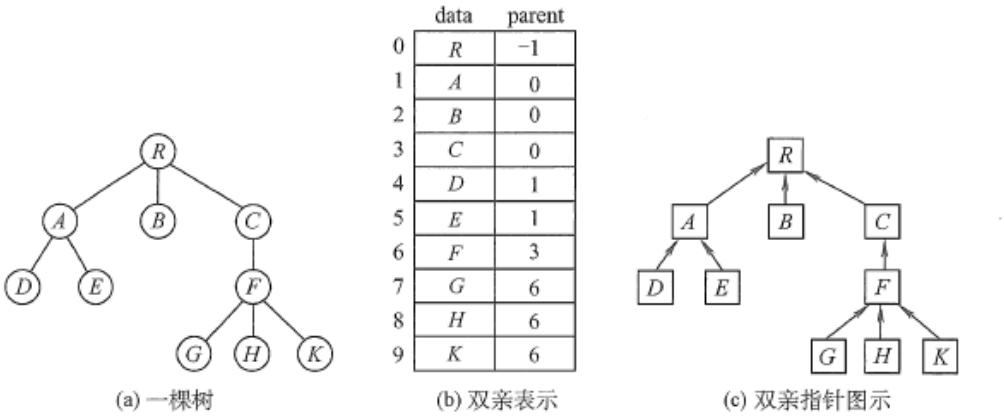
这种加上了线索的二叉链表称为**线索链表**，相应的二叉树称为**线索二叉树**(Threaded BinaryTree)。根据线索性质的不同，线索二叉树可分为**前序**线索二叉树、**中序**线索二叉树和**后序**线索二叉树三种。

注意：线索链表解决了**无法直接找到该结点在某种遍历序列中的前驱和后继结点**的问题，解决了二叉链表找左、右孩子困难的问题。

5.4 树的存储结构?

5.4.1 双亲表示法

这种存储方式采用**一组连续空间**来存储每个结点，同时在每个结点中增设一个**伪指针**，指示其**双亲结点在数组中的位置。**

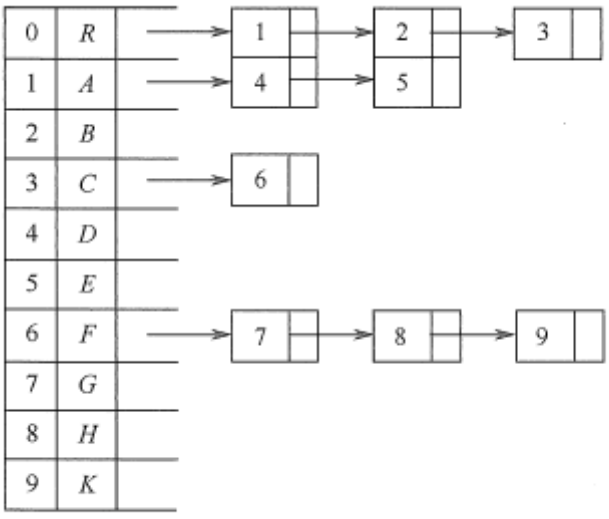


(a) 一棵树　(b) 双亲表示　(c) 双亲指针图示

该存储结构利用了每个结点（根结点除外）只有唯一双亲的性质，可以很快得到每个结点的双亲结点，但求结点的孩子时需要遍历整个结构。

5.4.2 孩子表示法

孩子表示法是将**每个结点**的**孩子结点**都用**单链表**链接起来形成一个线性结构，此时n 个结点就有n 个孩子链表（叶子结点的孩子链表为空表）

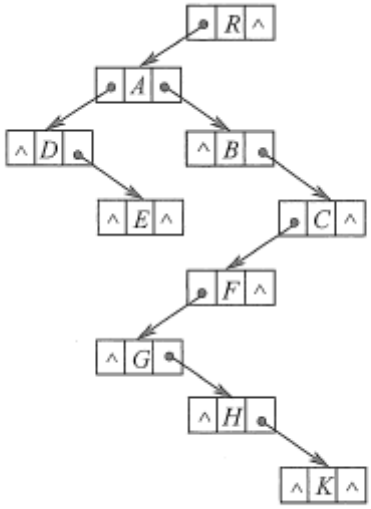


(a) 孩子表示法

这种存储方式寻找子女的操作非常直接，而寻找双亲的操作需要遍历n个结点中孩子链表指针域所指向的n个孩子链表。

5.4.3 孩子兄弟表示法

孩子兄弟表示法又称**二叉树表示法**，即以**二叉链表**作为树的存储结构。孩子兄弟表示法使**每个结点**包括三部分内容：**结点值、指向结点第一个孩子结点的指针，及指向结点下一个兄弟结点的指针**（沿此域可以找到结点的所有兄弟结点）



(b) 孩子兄弟表示法

这种存储表示法比较灵活，其最大的优点是可以方便地实现**树转换为二叉树**的操作，易于查找结点的孩子等，但缺点是从当前结点查找其双亲结点比较麻烦。若为每个结点增设一个parent域指向其父结点，则查找结点的父结点也很方便。

5.5 二叉排序树

5.5.1 二叉排序树的定义

二叉排序树（也称二叉查找树）或者是一棵空树，或者是具有下列特性的二叉树：

若左子树非空，则左子树上所有结点的值均小于根结点的值。
若右子树非空，则右子树上所有结点的值均大于根结点的值。
左、右子树也分别是一棵二叉排序树。
根据二叉排序树的定义，**左子树结点值 < 根结点值 < 右子树结点值**，所以对二叉排序树进行**中序遍历**，可以得到一个**递增的有序序列。**

5.5.2 二叉排序树的查找

二叉排序树的查找是从**根节点开始**的，延某个分支**逐层向下比较**的过程。若二叉树非空，先将给定值与根结点的关键字比较，若相等，则查找成功；若不等，如果小于根结点的关键字，则在根结点的左子树上查找，否则在根的右子树上查找。这显然是一个递归的过程。

5.6 平衡二叉树

为避免树的高度增长过快，降低二叉排序树的性能，规定在插入和删除二叉树结点时，要保证**任意结点**的**左、右子树高度差的绝对值不超过1**, 将这样的二叉树称为**平衡二叉树**(Balanced Binary Tree), 简称平衡树。

定义结点左子树与右子树的高度差为该结点的**平衡因子**，则平衡二叉树结点的平衡因子的值只可能是-1、0 或1 。因此，平衡二叉树可定义为或者是一棵空树，或者是具有下列性质的二叉树：它的左子树和右子树都是平衡二叉树，且左子树和右子树的高度差的绝对值不超过1 。

5.7 哈夫曼树和哈夫曼编码

5.7.1 哈夫曼树的定义

1．哈夫曼树的定义

在许多应用中，树中结点常常被赋予一个表示某种意义的数值，称为该结点的权。从树的根到任意结点的路径长度（经过的边数）与该结点上权值的乘积，称为该结点的带权路径长度。树中所有叶结点的带权路径长度之和称为该树的带权路径长度，记为

$$\mathrm{WPL}=\sum_{i=1}^{n} w_i l_i$$

式中，w_i是第 i 个叶结点所带的权值，l_i是该叶结点到根结点的路径长度。

在含有 n 个带权叶结点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为哈夫曼树，也称最优二叉树。例如，图 5.34 中的 3 棵二叉树都有 4 个叶子结点 a, b, c, d，分别带权 7, 5, 2, 4，它们的带权路径长度分别为

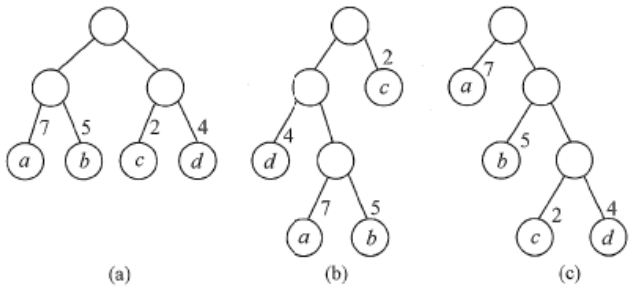


图 5.34　具有不同带权长度的二叉树

(a) WPL = 7×2 + 5×2 + 2×2 + 4×2 = 36。

(b) WPL = 4×2 + 7×3 + 5×3 + 2×1 = 46。

(c) WPL = 7×1 + 5×2 + 2×3 + 4×3 = 35。

其中，图 5.34(c)树的 WPL 最小。可以验证，它恰好为哈夫曼树。

5.7.2 哈夫曼树的构造

给定n 个权值分别为W1,W2…， Wn 的结点，构造哈夫曼树的**算法描述**如下：

1. 将这n 个结点分别作为**n 棵仅含一个结点的二叉树**，构成**森林F** 。
2. 构造一个**新结点**，从F 中选取**两棵根结点权值最小的树**作为新结点的左、右子树，并且将**新结点的权值**置为左、右子树上根结点的**权值之和。**
3. 从F 中**删除**刚才选出的**两棵树**，同时将**新得到的树加入**F 中。
   重复步骤2) 和3)'直至F 中只剩下一棵树为止。

从上述构造过程中可以看出哈夫曼树具有如下**特点**：

1. 每个初始结点最终都成为叶结点，且权值越小的结点到根结点的路径长度越大。
2. 构造过程中共新建了n — 1 个结点（双分支结点），因此哈夫曼树的结点总数为2n -1 。
3. 每次构造都选择2 棵树作为新结点的孩子，因此哈夫曼树中不存在度为1 的结点。

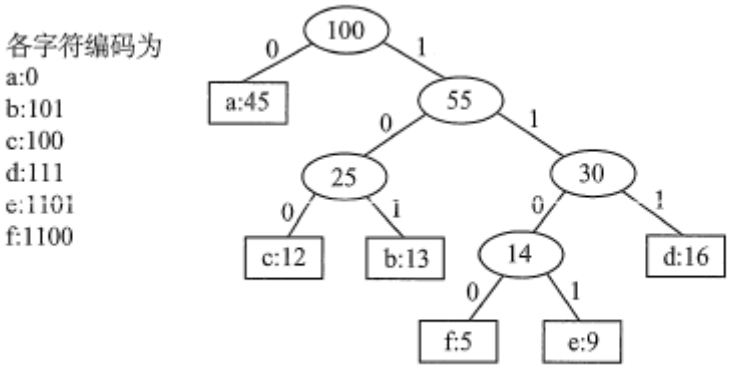
5.7.3 哈夫曼编码

在数据通信中，若对每个字符用**相等长度的二进制位**表示，称这种编码方式为**固定长度编码。**
若允许对不同字符用**不等长的二进制位**表示，则这种编码方式称为**可变长度编码。**

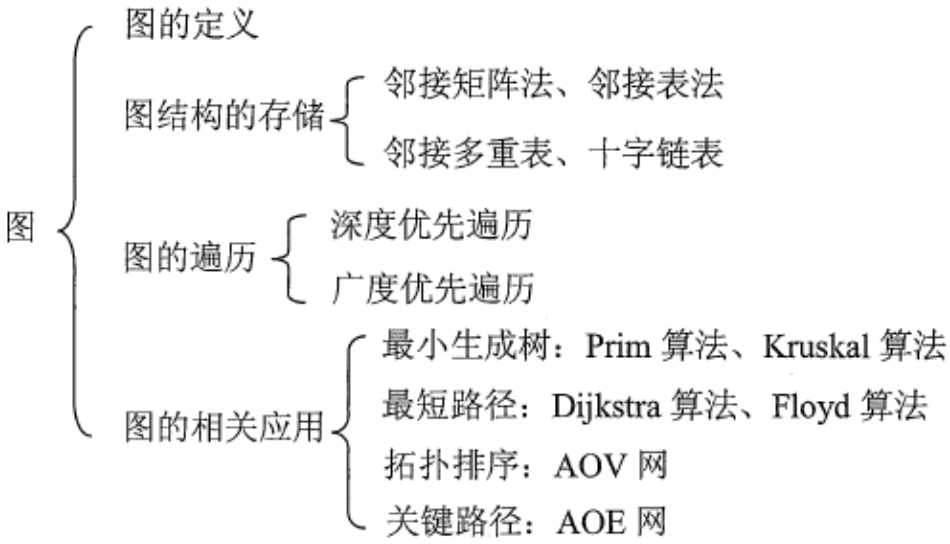
可变长度编码比固定长度编码要好得多，其**特点**是对**频率高**的字符赋以**短编码**，而对**频率较低**的字符则赋以**较长一些的编码**，从而可以使字符的平均编码长度减短，起到**压缩数据**的效果。

哈夫曼编码是一种被广泛应用而且非常有效的数据压缩编码。若**没有一个编码是另一个编码的前缀**，则称这样的编码为**前缀编码。**

由哈夫曼树得到哈夫曼编码是很自然的过程。首先，将每个出现的字符当作一个独立的结点，其权值为它出现的频度（或次数），构造出对应的哈夫曼树。显然，所有字符结点都出现在叶结点中。我们可将字符的编码解释为**从根至该字符的路径上边标记的序列**，其中**边标记为0** 表示"转向**左孩子**"，**标记为1** 表示"转向**右孩子**"。



第六章图



6.1 图的一些相关定义（连通分量再查一下）

简单图： 含平行边的图称为多重图，既不含平行边也不包含自环的图称为简单图。

无向图： 顶点间的边为无序对

有向图： 顶点间的弧为有序对。

网： 每条边或弧具有与之相关数值。

度： 无向图中，与顶点v相关联的边或弧的数目称为顶点v的度。有向图中，入v的弧数目成为入度，出v的弧数量称为出度。度=出度+入度。

完全图： 具有n个顶点和n(n-1)/2条边的无向图，必定是连通图。

有向完全图： 具有n个顶点和n(n-1)条边的有向图，每两个顶点之间都有两条方向相反的边连接的图。

回路或环： 第一个顶点和最后一个顶点相同的路径。

简单回路或简单环： 除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

连通： 顶点v至v' 之间有路径存在

连通图： 无向图图 G 的任意两点之间都是连通的，则称G是连通图。

连通分量： 极大连通子图，子图中包含的顶点个数极大

强连通图： 有向图G的任意两点之间都是连通的，则称G是强连通图。各个顶点间均互相有向可达。

强连通分量： 极大连通子图

一个无向图 G=(V,E) 是连通的，则：|E|>=|V|-1，而反之不成立。

一个有向图 G=(V,E) 是是强连通图，则：|E|>=|V|，而反之不成立。

没有回路的无向图是连通的当且仅当它是树，即等价于：|E|=|V|-1。

子图： 一个图的结点集和边集都是另一个图的子集，称这个图为另一个图的子图

生成子图： 由一个图的全部顶点及连结这些顶点的部分边构成的连通图称为原图的生成子图。

生成树： 形状为树的生成子图，往往不唯一

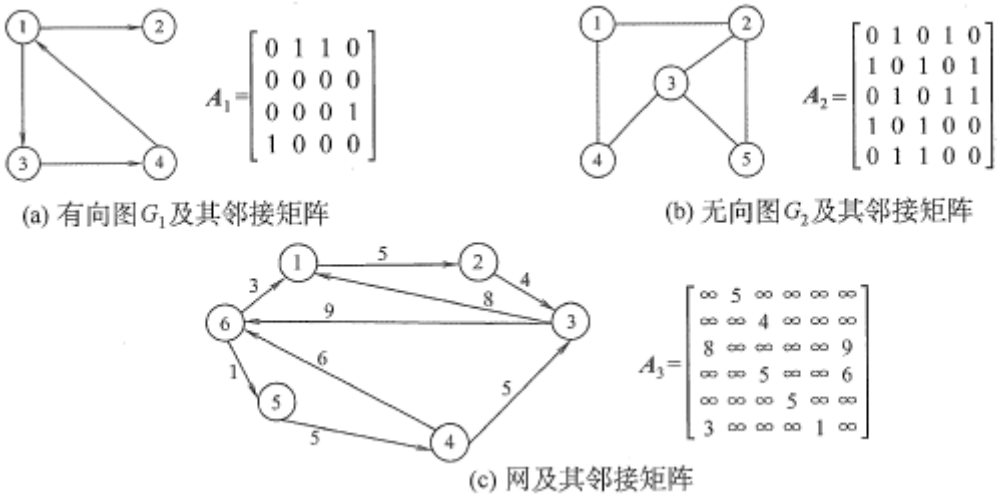
最小生成树： 极小连通子图。包含图的所有n个结点，但只含图中足以构成树的n-1条边。性质：有n个顶点的生成树必有n-1条边，如少则不连通，多则必成环。但有n-1条边不一定是树。

6.2 图的存储结构

6.2.1 邻接矩阵法

所谓邻接矩阵存储，是指用一个**一维数组**存储图中**顶点的信息**用一个**二维数组**存储图中**边的信息**（即各顶点之间的邻接关系），存储**顶点之间邻接关系**的二维数组称为**邻接矩阵**。

适合**稠密图**。



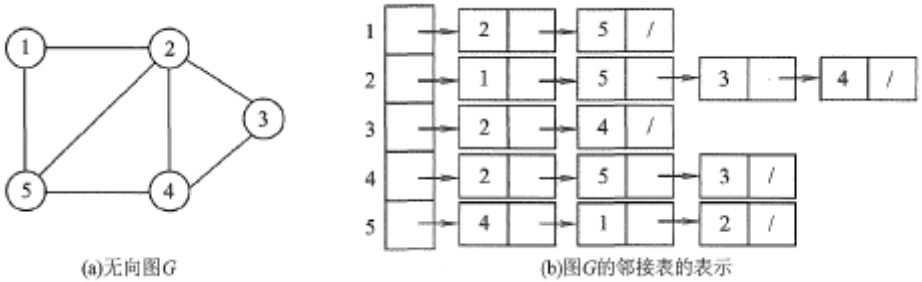
(a) 有向图G_1及其邻接矩阵　　(b) 无向图G_2及其邻接矩阵

(c) 网及其邻接矩阵

6.2.2 邻接表法

当一个图为**稀疏图**时，使用邻接矩阵法显然要浪费大量的存储空间，而图的邻接表法结合了**顺序存储**和**链式存储**方法，大大减少了这种不必要的浪费。

所谓**邻接表**，是指对图G 中的每个**顶点V建立一个单链表**，第i个单链表中的结点表示依附于顶点v, 的边（对于有向图则是以顶点v, 为尾的弧），这个单链表就称为顶点vi 的边表（对于有向图则称为出边表）。边表的头指针和顶点的数据信息采用顺序存储（称为顶点表），所以在邻接表中存在两种结点：**顶点表结点**和**边表结点**。



(a)无向图G　　(b)图G的邻接表的表示

6.2.3 十字链表法

十字链表法是**有向图**的一种**链式存储结构**。在十字链表中，对应于有向图中的每条弧有一个结点，对应于每个顶点也有一个结点。

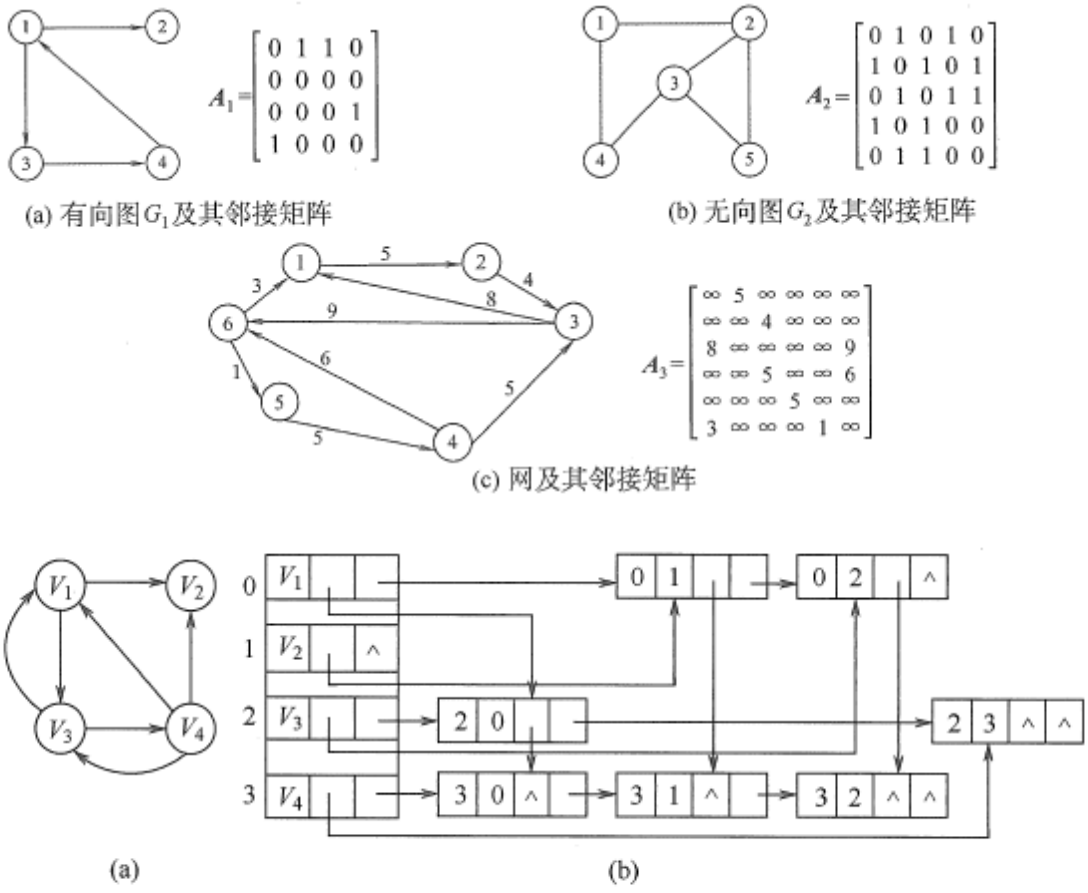


图 6.9　有向图的十字链表表示

6.2.4 邻接多重表

邻接多重表是**无向图**的另一种**链式**存储结构。

在邻接表中，容易求得顶点和边的各种信息，但在邻接表中求两个顶点之间是否存在边而对边执行删除等操作时，需要分别在两个顶点的边表中遍历，效率较低。与十字链表类似，在邻接多重表中，每条边用一个结点表示，每个顶点也用一个结点表示。

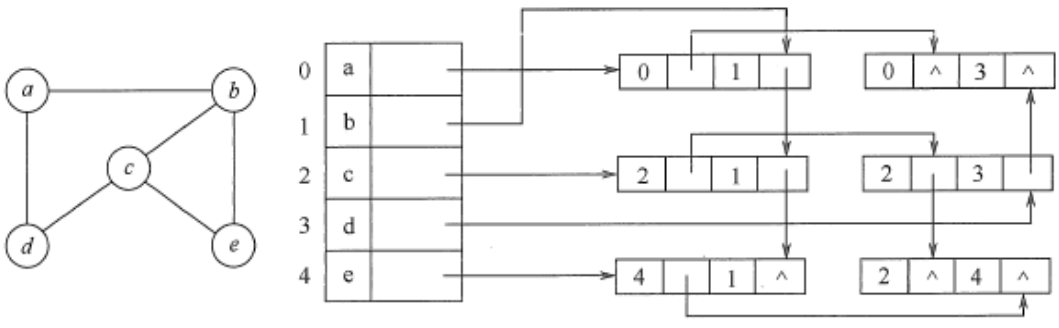
mark	ivex	ilink	jvex	jlink	info

其中，mark 为标志域，可用以标记该条边是否被搜索过；ivex 和 jvex 为该边依附的两个顶点在图中的位置；ilink 指向下一条依附于顶点 ivex 的边；jlink 指向下一条依附于顶点 jvex 的边，info 为指向和边相关的各种信息的指针域。

每个顶点也用一个结点表示，它由如下所示的两个域组成。

data	firstedge

其中，data 域存储该顶点的相关信息，firstedge 域指示第一条依附于该顶点的边。



6.3 图的遍历

图的遍历是指从图中的某一顶点出发，按照某种**搜索方法**沿着图中的边对图中的**所有顶点访问一次且仅访问一次。**

为避免同一顶点被访问多次，在遍历图的过程中，必须记下每个已访问过的顶点，为此可以设一个**辅助数组visited[]** 来标记顶点**是否被访问过。**图的遍历算法主要有两种：**广度**优先搜索和**深度**优先搜索。

6.3.1 广度优先搜索(Breadth-First-Search, BFS)

类似于二叉树的层序遍历算法。**基本思想**是：首先访问**起始顶点**V, 接着由V出发，依次访问V 的**各个未访问过的邻接顶点**W1, W2,... Wn, 然后依次访问W1, W2,..., Wn 的所有未被访问过的邻接顶点；再从这些访问过的顶点出发，访问它们所有未被访问过的邻接顶点，直至图中所有顶点都被访问过为止。若此时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为初始点，重复上述过程。Dijkstra源最短路径算法和Prim 最小生成树算法也应用了类似的思想。

6.3.2 深度优先搜索（Depth-First-Search, DFS)

基本思想如下：首先访问图中某一**起始顶点**V, 然后由v 出发，**访问与v 邻接且未被访问的任一顶点W1**，再访问与W1邻接且未被访问的任一顶点W2.....重复上述过程。当不能再继续向下访问时，**依次退回到最近被访问的顶点**，若它还有邻接顶点未被访问过，则从该点开始继续上述搜索过程，直至图中所有顶点均被访问过为止。

6.4 最小生成树和最短路径

最小生成树： 生成树中**边的权值(代价)之和最小**的树。

6.4.1迪杰斯特拉（dijkstra）算法

迪杰斯特拉算法是经典的**单源最短路径算法**，用于**求某一顶点到其他顶点的最短路径**

它的**特点**是以**起始点为中心层层向外扩展，直到扩展的终点为止**，迪杰斯特拉算法要求**边的权值不能为负权。**

6.4.2弗洛伊德（Floyd）算法

弗洛伊德算法是经典的**求任意顶点之间的最短路径**，其**边的权值可为负权**，该算法的**时间复杂度为O(N3),空间复杂度为O（N2）。**

6.4.3普里姆（prim）算法

用来求**最小生成树**，其**基本思想**为：从连通网络N={V,E}中某一顶点u0出发，选择与他**关联的最小权值的**边，将其顶点加入到顶点集S中，此后就从一个顶点在S集中，另一个顶点不在S集中的所有顶点中选择出权值最小的边，把对应顶点加入到S集中，**直到所有的顶点都加入到S集中为止。**

6.4.4克鲁斯卡尔（kruskal）算法

用来求**最小生成树**，其**基本思想**为：设有一个有N个顶点的连通网络N={V,E},初始时建立一个只有N个顶点，没有边的非连通图T，T中每个顶点都看作是一个连通分支，从边集E中**选择出权值最小的边且该边的两个端点不在一个联通分支中**，则把该边加入到T中，否则就再重新选择一条权值最小的边，直到所有的顶点都在一个连通分支中为止。

6.5 AOV和AOE

无环的有向图称作有向无环图，简称 **DAG图** 。

拓扑排序： 由一个集合上的偏序得到该集合上的一个全序，得到的全序称为拓扑有序

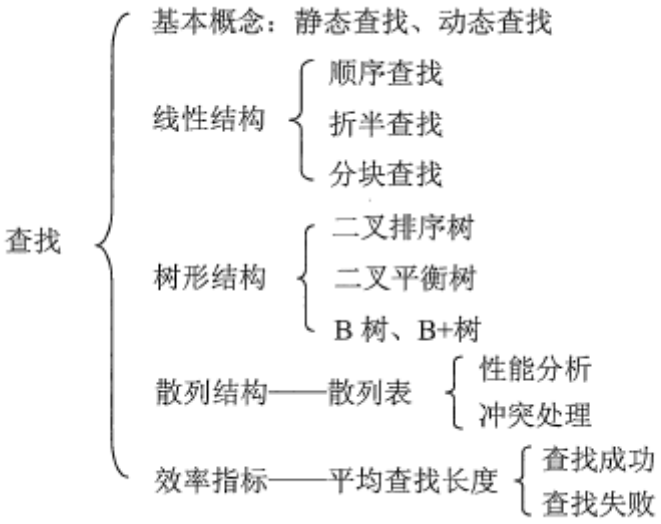
AOV网(Activity On Vertex)：用**顶点表示活动**，**边表示活动的优先关系**的有向图称为AOV网。AOV网中不允许有回路，因为回路这意味着某项活动以自己为先决条件。

拓扑有序序列：把AOV网络中各顶点按照它们相互之间的优先关系排列一个线性序列的过程。若vi是vj前驱，则vi一定在vj之前；对于没有优先关系的点，顺序任意。

AOE网（activity on edge）：用**边表示活动**，用于估算工程完成的时间。

工程完成的时间是从起始点到终止点的最长路径的长度。这条路径最长的路径称为**关键路径**

第七章查找



7.1 对各种查找方法的概括?

查找分为静态查找表和动态查找表；

静态查找表包括：顺序查找、折半查找、分块查找；

动态查找表包括：二叉排序树和平衡二叉树。

7.1.1顺序查找

把**待查关键字**key放入**哨兵位置**（i=0），再**从后往前**依次把表中元素和key比较，如果返回值为0则查找失败，表中没有这个key值，如果返回值为元素的位置i（i!=0）则查找成功，设置哨兵的位置是为了加快执行速度。**时间效率为O（n）**

其**特点**是：结构简单，对**顺序**结构和**链式**结构都适用，但是查找效率太低。

7.1.2折半查找

要求**查找表**为**顺序**存储结构并且**有序**，若关键字在表中，则返回关键字的位置，若关键字不在表中时停止查找的典型标志是：**查找范围的上界<=查找范围的下界。**

7.1.3分块查找

先把查找表分为**若干子表**，要求每个子表的元素都要比他后面的子表的元素小，也就是保证**块间是有序**的（但是子表内不一定有序），把各子表中的**最大关键字构成一张索引表**，表中还包含各子表的**起始地址。**

特点是：**块间有序，块内无序**，查找时**块间**进行**索引**查找，**块内**进行**顺序**查找。

7.1.4 二叉排序树

二叉排序树的定义为：或者是一棵空树，或者是一棵具有如下特点的树：如果该树有左子树，则其左子树的所有节点值小于根的值；若该树有右子树，则其右子树的所有节点值均大于根的值；其左右子树也分别为二叉排序树。在查找时可以进行动态的插入，插入节点要符合二叉排序树的定义，这也是动态查找和静态查找的区别，**静态查找不能进行动态插入。**

7.1.5平衡二叉树

平衡二叉树又称为AVL树，它或者是一棵空树或者具有如下特点：他的左子树和右子树的高度差的绝对值不能大于1，且他的左右子树也都是平衡二叉树。

平衡因子：是指左子树的高度减去右子树的高度，它的值只能为1,0，-1

如果再一个平衡二叉树中插入一个节点可能造成失衡，这时就要进行树结构的调整，即**平衡旋转。**

包括4中情况：

在**左**子树的**左**子树上插入节点时向**右**进行单向**旋**转；

在**右**子树的**右**子树上插入节点时向**左**进行单向**旋**转；

在**左**子树的**右**子树插入节点时先向**左**旋转再向**右**旋转；

在**右**子树的**左**子树插入节点时先向**右**旋转再向**左**旋转。

7.2 B树和B+树

7.2.1B 树

又称**多路平衡查找树**， B 树中所有结点的孩子个数的最大值称为**B 树的阶**，通常用m表示。一棵m 阶B树或为空树，或为满足如下特性的m 叉树：

1. 树中每个结点至多有m 棵子树，即**至多**含有**m-1** 个关键字。
2. 若根结点不是终端结点，则至少有两棵子树。
3. 除根结点外的所有非叶结点至少有「m/2] 棵子树，即**至少**含有**「m/2]- 1** 个关键字。
4. 所有的**叶结点**都出现在**同一层次**上，并且不带信息（可以视为外部结点或类似于折半查找判定树的查找失败结点，实际上这些结点不存在，指向这些结点的指针为空）。

B 树是所有结点的平衡因子均等于0 的多路平衡查找树。

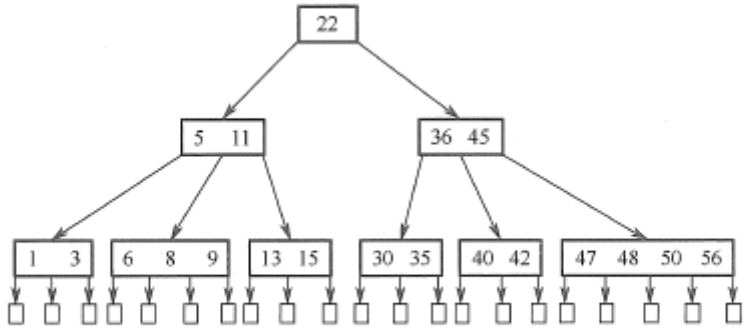


图 7.4　一棵 5 阶 B 树的实例

7.2.2 B+树

是因**数据库所**需而出现的一种B 树的变形树。

一棵m 阶的B+树需满足下列条件：

1. 每个分支结点最多有m 棵子树（孩子结点）。
2. 非叶根结点至少有两棵子树，其他每个分支结点至少有「m/2]棵子树。
3. 结点的子树个数与关键字个数相等。

4. 所有叶结点包含全部关键字及指向相应记录的指针，叶结点中将关键字按大小顺序排列，并且相邻叶结点按大小顺序相互链接起来。
5. 所有分支结点（可视为索引的索引）中仅包含它的各个子结点（即下一级的索引块）中关键字的最大值及指向其子结点的指针。

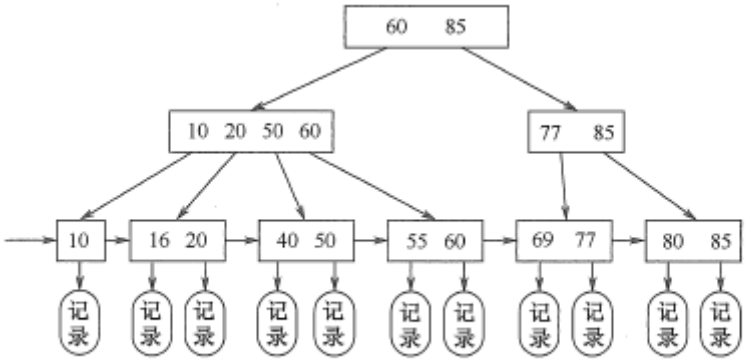


图 7.8　B+树结构示意图

7.2.3 m 阶的B+树与m 阶的B 树的主要差异

1. 在B+树中，具有n 个关键字的结点只含有n 棵子树，即**每个关键字对应一棵子树**；而在B 树中，具有**n 个关键字的结点含有n+1棵子树。**
2. 在B+树中，每个结点（非根内部结点）的关键字个数n 的范围是「m/2]<=n<= m (根结点：1<=n<=m); 在B 树中，每个结点（非根内部结点）的关键字个数n 的范围是「m/2]-1<=n<=m-1。
3. 在B+树中，叶结点包含信息，所有非叶结点仅起索引作用，非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。
4. 在B+树中，叶结点包含了全部关键字，即在非叶结点中出现的关键字也会出现在叶结点中；而在B树中，叶结点包含的关键字和其他结点包含的关键字是不重复的。

7.3 哈希表的概念

哈希表又称为散列表，是根据**关键字的值**直接进行访问的数据结构，即它通过把**关键码的值映射**到表中的一个**位置**以加快查找速度，其中映射函数叫做**散列函数**，存放记录的数组叫做**散列表。**

7.4 哈希函数的构造方法

哈希函数的构造方法包括：直接定址法，除留余数法，数字分析法，平方取中法，折叠法，随机数法

（1）**直接定址法**：取关键字的某个**线性函数值**作为散列地址，H(key)=a*key+b。

（2）**除留余数法**：取关键字对**p取余**的值作为散列地址，其中p<m,即H(key)=key%p (p<m)。

（3）**数字分析法**：当**关键字的位数大于地址的位数**，对关键字的各位分布进行分析，选出**分布均匀的任意几位**作为散列的地址，适用于所有**关键字都已知**的情况。

（4）**平方取中法**：对关键字**求平方**，再取**结果中的中间几位**作为散列地址。

（5）**折叠法**：将关键字分为**位数相同**的几部分，然后取这几部分的**叠加和**作为散列地址。适用于**关键字位数较多**，且关键字中每一位上数字**分布大致均匀。**

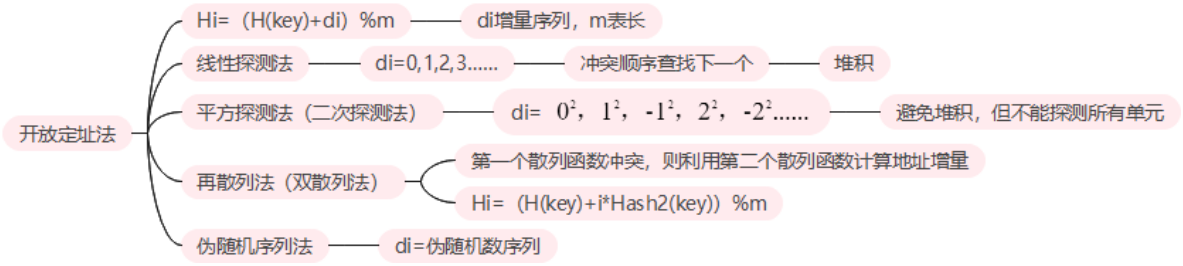
（6）**随机数法**：选择一个**随机函数**，把关键字的**随机函数值**作为散列地址。适合于**关键字的长度不相同**时。

7.5 哈希冲突的解决办法

哈希冲突的解决方法包括：**开放定址法**和**链接法。**

7.5.1 开放定址法

当冲突发生时，使用某种**探测技术**形成一个**探测序列**，然后沿此序列**逐个单元**查找，直到找到该关键字或者碰到一个开放的地址为止，探测到开放的地址表明该表中没有此关键字，若要插入，则探测到开放地址时可将新节点插入该地址单元。其中开放定址法包括：线性探查法，二次探查法，双重散列法



（1）**线性探查法**：

基本思想，探查时从地址d开始，首先探查T[d],在探查T[d+1]...直到查到T[m-1]，此后循环到T[0],T[1]...直到探测到T[d-1]为止。

（2）**二次探查法**：

基本思想，探查时从地址d开始，首先探查T[d],再探查T[d+12],T[d+22]...等，直到探查到有空余地址或者探查到T[d-1]为止，缺点是无法探查到整个散列空间。

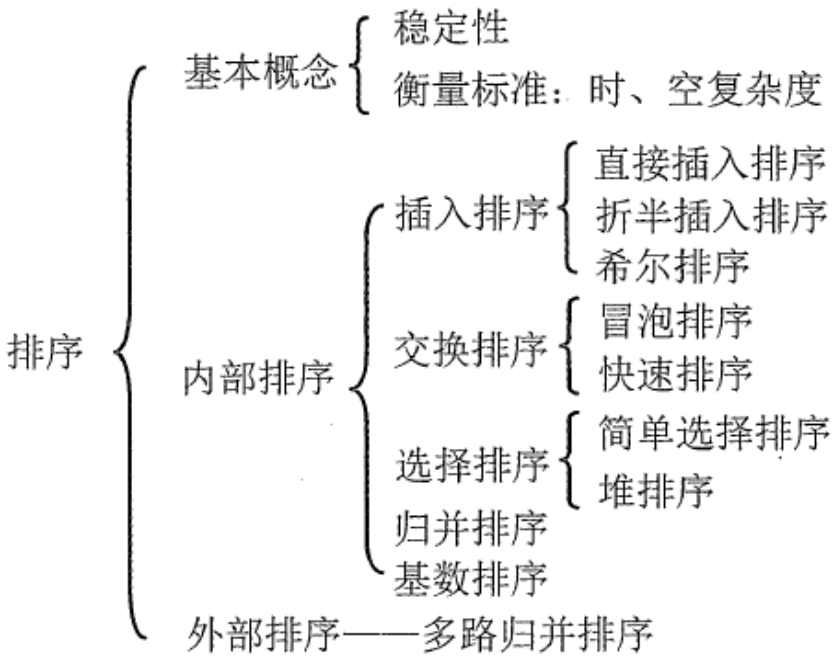
（3）**双重散列法**：

基本思想，使用两个散列函数来确定地址，探查时从地址d开始，首先探查T[d],再探查T[d+h1(d)],T[d+2*h1(d)]...

7.5.2 链接法

将所有**关键字为同义词**的节点链接在**同一个单链表**中，若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数组，凡是**散列地址为i的节点均插入到头指针为i的单链表**中。

第八章排序



8.1对各种内部排序的概括?

排序：是指把一个任意元素的序列排列成一个**按关键字key有序**的序列。

内部排序包括：插入排序、选择排序、交换排序、归并排序、基数排序。

其中**插入排序**包括：直接插入排序、折半插入排序、希尔排序；

选择排序包括：简单选择排序，堆排序；

交换排序包括：冒泡排序、快速排序。

（1）直接插入排序（稳定）

基本思想为：将序列分为有序部分和无序部分，从无序部分依次选择元素与有序部分比较找到合适的位置，将原来的元素往后移，将元素插入到相应位置上。

时间复杂度为：O（n^2）,空间复杂度为O（1）

（2）折半插入排序（稳定）

基本思想为：设置三个变量low high mid，令mid=(low+high)/2,若a[mid]>key,则令high=mid-1,否则令low=mid+1,直到low>high时停止循环，对序列中的每个元素做以上处理，找到合适位置将其他元素后移进行插入。

比较次数为O(nlog2n),但是因为要后移，因此**时间复杂度为O(n^2),空间复杂度为O(1)。** 优点是：比较次数大大减少。

（3）希尔排序（不稳定）

基本思想为：先将序列分为**若干个子序列**，对各子序列进行**直接插入排序**，等到序列基本有序时再对整个序列进行一次直接插入排序。

优点是：让关键字值小的元素能够很快移动到前面，且序列基本有序时进行直接插入排序时间效率会提升很多，**空间复杂度为O（1）。**

（4）简单选择排序（不稳定）

基本思想为：将序列分为**2部分**，每经过一趟就在**无序部分**找到一个**最小值**然后与**无序部分的第一个元素交换位置。**

优点是：实现起来特别简单，**缺点**是：每一趟只能确定一个元素的位置，时间效率低。

时间复杂度为O（n^2），空间复杂度为O（1）。

（5）堆排序（不稳定）

设有一个任意序列，k1,k2,...,kn，当满足下面特点时称之为堆：让此序列排列成**完全二叉树**，该树具有以下特点，该树中任意节点均大于或小于其左右孩子，此树的根节点为最大值或者最小值。

优点是：对大文件效率明显提高，但对小文件效率不明显。

时间复杂度为O(nlog2n),空间复杂度为O(1)。

（6）冒泡排序（稳定）

基本思路为：每一趟都将元素进行**两两比较**，并且按照"**前小后大**"的规则进行交换。

优点是：每一趟不仅能找到一个最大的元素放到序列后面，而且还把其他元素理顺，如果下一趟排序没有发生交换则可以提前结束排序。

时间复杂度为O（n^2）,空间复杂度为O（1）。

（7）快速排序（不稳定）

基本思路为：在序列中**任意选择一个元素**作为**中心**，比它**大**的元素一律**向后移动**，比它**小**的元素一律**向前移动**，形成左右两个子序列，再把子序列按上述操作进行调整，直到**所有的子序列中都只有一个元素**时序列即为有序。

优点是：每一趟不仅能确定一个元素，时间效率较高。

时间复杂度为O(nlog2n),空间复杂度为O（log2n）.

（8）归并排序（稳定）

基本思想为：把两个或者两个以上的有序表合并成一个新的有序表。**时间复杂度为O（nlogn）,空间复杂度和待排序的元素个数相同。**

（9）基数排序（稳定）

时间复杂度为：对于n个记录进行链式基数排序的时间复杂度为**O（d(n+r)）**,其中每一趟分配的时间复杂度为O(n),回收的时间复杂度为O（rd）。

8.2内部排序总结

算法种类	时间复杂度			空间复杂度	是否稳定
	最好情况	平均情况	最坏情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	是
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	否
希尔排序				$O(1)$	否
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
2 路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	是

直接插入排序、冒泡排序和简单选择排序是基本的排序方法，它们主要用于**元素个数n不是很大**(n < 10000) 的情形。

对于**中等规模**的元素序列(n <=1000), **希尔排序**是一种很好的选择。

对于**元素个数n 很大**的情况，可以采用**快排、堆排序、归并排序或基数排序**，其中快排和堆排序都是不稳定的，而归并排序和基数排序是稳定的排序算法。