数据结构

第一章：基本概念

基本概念和术语

数据（Data）所有能输入到计算机中并被计算机程序处理的符号的总称，加工原料。

数据元素（Data Element）数据的基本单位，通常作为一个整体考虑和处理。

数据项（Data Item）数据元素的可能组成单位。

数据对象（Data Object）Data的一个子集，是性质相同的Data Element的集合。

数据结构（Data Structure）相互之间存在一定特定关系的Data Element的集合

*****四种基本结构：1.集合 2.线性 3.树形 4.图状或网状

（集合是其中最松散的一种结构）

在计算机中的映像称为数据的物理结构，又称存储结构 包括1.数据元素的映像2.关系的映像

计算机中的概念

位（bit）信息最小单位，二进制数的一位。

元素（Element）、节点（Node）若干为组合起来的一个位串表示的数据元素

数据域（Data Field）数据元素由若干数据项组成时，位串对应于各数据项的子位串成为数据域

***元素或节点可看作Data Element在计算机的映像

计算机中数据元素之间的关系：

1.顺序映像：借助元素在存储器中的相对位置来表示数据元素的逻辑关系

2.非顺序映像：借助指示元素存储地址的指针（Pointer）表示数据之间的逻辑关系

由此得到的两种不同的存储结构：

1.顺序存储结构 2.链式存储结构

*****算法的设计取决于选定的数据结构，而算法的实现取决于采用的存储结构

数据类型：

1.数据类型（Data Type）是一个值的集合和定义在这个值集上的一组操作的总称

分类：1.非结构的原子类型，原子类型值是不可分解的。

如：C语言中基本类型（整型、实型、字符型、枚举型）、指针类型、空类型

2.结构类型：由若干成分按某种结构组成的，可分解。

2.抽象数据类型（Abstract Data Type简称ADT）一个值域以及定义在该值域上的一组操作。

*****两者实质上是一个概念，“抽象”的意义在于数据类型的数学抽象特性。

不过抽象数据类型的范畴更广

按值域的不同特性区分：

1.原子类型(Atomic Data Type):属原子类型，一般很少出现，毕竟原子类型自身就足够

2.固定聚合类型（Fixed-aggregate Data Type）值域由确定数目的成分按照某种结构组成

3.可变聚合类型（Variable-Aggregate Data Type）和2比较构成值的成分数目不确定

后两种类型可统称为结构类型

抽象数据类型可用三元组表示（D,S,P）

D:数据对象 S:数据对象的关系集 P:对数据对象的基本操作集

3.多型数据类型（Polymorphic Data Type） 其值的成分不确定的数据类型

*******算法和算法分析**

算法（Alogorithm）是对特定问题求解步骤的一种描述

特性：

1.有穷性：一个算法必须总是在执行有穷步结束，别每步时间是有穷

2.确定性：每条指令必须有确切含义

3.可行性：描述的操作都是可行可实现的

4.输入

5.输出

设计要求：

1.正确性（Correctness）

2.可读性（Readability）

3.健壮性（Robustness）

4.效率与低存储量需求

效率度量：

**时间复杂度

算法由控制结构（顺序、分支、循环三种）和原操作（固有数据类型的操作）构成

所以算法时间取决于两者的综合效果。

通常做法：选一种属于基本操作的原操作，以该基本操作重复执行的次数作为时间量度

栗子：

```
for(i=1;i<=n;i++){
 for(j=i;j<=n;j++){
  c[i,j]=0;
  for(k=1;k<=n;k++){
   c[i,j]+=a[i,k]*b[k,j];
  }
 }
}
```

执行时间与该基本操作（即乘法）重复执行的次数n^3成正比，T(n)=O(n^3)

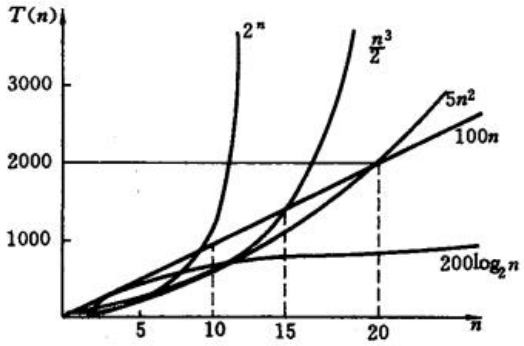


图 1.7　常见函数的增长率

*时间度量记作

T（n）=O（f（n））

*它称作算法的渐进时间复杂度（Asymptotic Time Complexity）,简称时间复杂度

栗子：

```
(a){++x;s=0;}
(b)for(i=1;i<=n;++i){++x;s+=x;}
(c)for(j=1;j<=n;++j){
 for(k=1;k<=n;++k){++x;s+=x;}
  }
```

含基本操作“x+1”的频度分别为1，n和n^2，则三个时间复杂度分别为

O(1),O(n),O(n^2),分别称为常量阶、线性阶、平方阶。

还有例如：对数阶O（log n）和指数阶O（2^n）等

***尽量用多项式阶O（n^k）的算法，而少用指数阶算法

第二章：线性表

线性结构特点：

1.存在惟一一个被称作“第一个”的数据元素；

2.存在惟一一个被称作“最后一个”的数据元素；

3.除第一个之外，集合中每个数据元素均只有一个前驱；

4.除最后一个之外，集合中每个数据元素均只有一个后继

（翻译成人话：有始有终、鳞次栉比）

线性表（Linear-List）：最常用、最简单的数据结构。

构成：数据项（Item）组成 记录（Record）组成 文件（File）

线性表的顺序：用一组地址连续的存储单元依次存储线性表的数据元素

这种机制称作线性表的顺序存储结构或顺序映像（Sequential Mapping）

线性表插入操作：在某个元素之前插入一个元素时，要将后面的所有元素都向后移动一个位置

线性表删除操作：在某个元素之前删除一个元素时，要将后面的所有元素都向前移动一个位置

*****由此可见，时间主要耗费在移动元素上，移动元素的个数取决于插删元素的位置，尾部添加删除操作最省时间

若以线性表表示集合并进行集合的各种运算，应先对表中元素进行排序

线性表的链式表示和实现

链式的原因：解决线性表的弱点——插入或删除操作时需要移动大量元素。

在很多场合下它是线性表的首选存储结构

它的缺点：求长度的时候不如顺序型

线链式表：

特点：用一组任意的存储单元存储线性表的数据元素（这组单元可连可不连续）

存储：对于其中的元素来说，存储是除了本身的信息之外，还需要存储直接后继的存储位置。

节点（Node）元素自身+元素直接后继的存储位置

包括两个域：1.数据域：存储元素自身信息的域。

2.指针域：存储直接后继存储位置的域。其中的信息称作“指针”或“链”

链表：n个节点链接成的表，即线性表的链式存储结构。

由于此链表的每个节点中只包含一个指针域，故又称线性链表或单链表

存取：必须从头指针开始进行，头指针表示第一个节点的存储位置

***最后一个节点没有后继，所以最后一个节点的指针为NULL

栗子：

	存储地址	数据域	指针域
	1	LI	43
	7	QIAN	13
头指针 H	13	SUN	1
31	19	WANG	NULL
	25	WU	37
	31	ZHAO	7
	37	ZHENG	19
	43	ZHOU	25

图 2.5　线性链表示例

可以看出，各元素的逻辑关系由节点中的指针指示，所以逻辑上两元素存储的物理位置不要求紧邻

****由此，这种存储结构为非顺序映像或链式映像；

*****单链表由头指针唯一确定，在C语言中可用“结构指针”来描述。

头结点：在单链表的第一个节点之前附设的一个节点

他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类的附加信息

他的指针域存储指向第一个节点的指针

链表中的查找操作：修改前一节点的指针域为插入节点存储位置，要插入节点的指针域指向后一节点

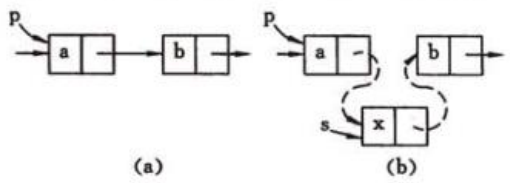


图 2.8　在单链表中插入结点时指针变化状况
(a) 插入前；(b) 插入后

链表中的删除操作：找到被删结点指针域（即后一节点的指向）并将它加入到到前一节点指针域

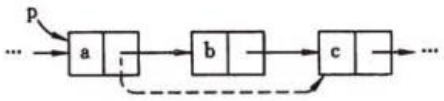


图 2.9　在单链表中删除结点时指针变化状况

**由此可见，首尾删插时间消耗相近

*****单链表是一种动态结构，每个链表占用的空间应需求即时生成。

建立线性表的链式存储结构的过程就是一个动态生成链表的过程。

用数组链表的方法便于在不设‘指针’类型的高级程序设计语言中使用链表结构。

数组的一个分量表示一个节点，同时用游标（指示器cur）代替指针指示节点在数组中的相对位置。

数组的[0]分量可看作头结点，其指针域表示第一个节点。

而且删插操作不需移动元素，仅需修改指针，这是链式存储结构的主要优点。

为了和指针性描述的线性链表区别，用数组描述的链表称作静态链表

循环链表（Circular Linked List）

特点：表中最后一个节点的指针与指向头结点，整个链表结成一个环。

双向链表

双向链表的节点中有两个指针域，一个指向直接后继，一个指向直接前驱

双向链表也可以有循环链表

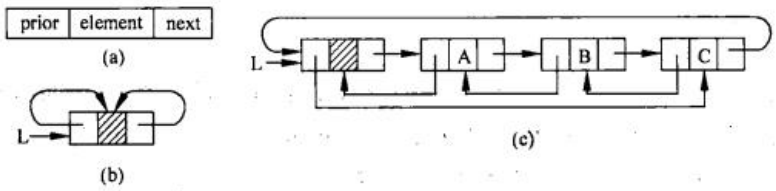


图 2.14　双向链表示例

（a）结点结构；（b）空的双向循环链表；（c）非空的双向循环链表

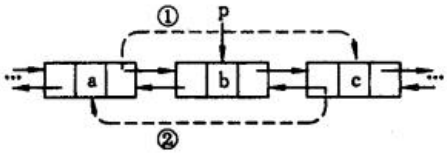


图 2.15　在双向链表中删除结点时指针变化状况

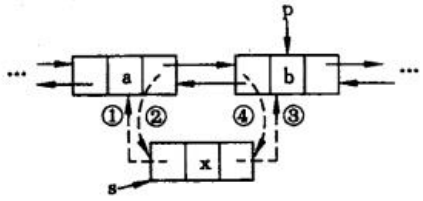


图 2.16　在双向链表中插入一个结点时指针的变化状况

第三章：栈和队列

**从数据结构角度看，栈和队列也是线性表，属于抽象数据类型

**但特殊性在于，栈和队列的基本操作时线性表操作的子集，操作受限，也因此成为限定性的数据结构

栈（Stack）是限定仅在表尾进行插入或删除操作的线性表（后进先出）

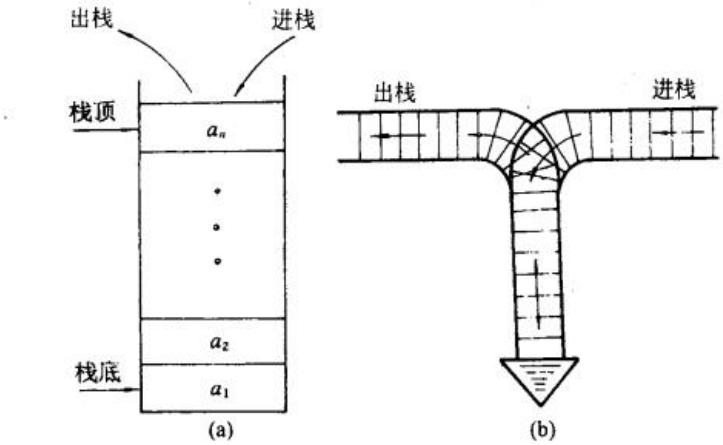


图 3.1　栈

（a）栈的示意图；（b）用铁路调度站表示栈

名词：栈顶（Top）栈底（bottom）空栈

栈底指针（bace）顺序栈中始终在栈底，为NULL时表示栈结构不存在

栈顶指针（top）初值指向栈底，每加新元素，指针top增一；删除时减一

栈的存储：

一般来说，在初始化设空栈时不应限定栈的最大容量。

一般合理做法：先分配个基本容量，在应用过程中不够再扩大

常量：STACK_INIT_SIZE(存储空间初始分配量)

STACKINCREMENT(存储空间分配增量)

有顺序栈和链栈两种

应用：1.进制转换，计算过程由低位到高位，输出是需要反过来，使用栈正好

2.括号匹配

3.行编辑程序的输入缓冲区

4.表达式按照算符优先法求职

5.实现递归，栗子：Hanoi塔问题

汇编程序设计中，主程序与子程序之间的连接及信息交换用的就是栈

高级语言编制的程序中，调用函数和被调用函数之间的连接及信息交换也用栈

*****在一个函数的运行期间调用另一个函数时，在运行被调用函数前，系统要做的是：

1.将所有实参、返回地址等信息传递给被调用函数

2.为被调用函数的局部变量分配存储区

3.将控制转移到被调函数的入口

*****而从被调用函数返回调用函数之前，也应完成：

1.保存被调函数的计算结果

2.释放被调函数的数据区

3.依照被调函数保存的返回地址将控制转移到调用函数

*****如果多个函数构成嵌套调用时，按照后调先回的原则

*****系统将整个程序运行时所需的数据空间安排在一个栈中，每当调用一个函数时，就为他在栈顶分配一个存储区，每当从一个函数退出时，就释放他的存储区，则当前正运行的函数的数据去必须在栈顶

队列（Queue）

队列是一种先进先出的线性表

名词：队头（front）队尾（rear）

典型栗子：操作系统中的作业排队（不知道是个啥- -#）

额外补充：双端队列（Deque）

限定插入和删除操作在表的两端进行的线性表

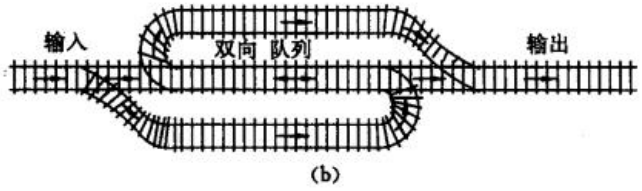


图 3.9　双端队列示意图

（a）双端队列；（b）铁道转轨网

ps.看起来似乎比栈和队列灵活，实际开发中没卵用

链队列

用链表表示的队列简称为链队列（妹的废话）

一个练队列需要两个分别只是队头和队尾的指针（即头指针和尾指针）才能唯一确定

循环队列

在C语言中不能用动态分配的一维数组来实现循环队列

如果软件中有社循环队列，则必须为他设定一个最大队列长度，拿不到最大长度则宜采用链队列

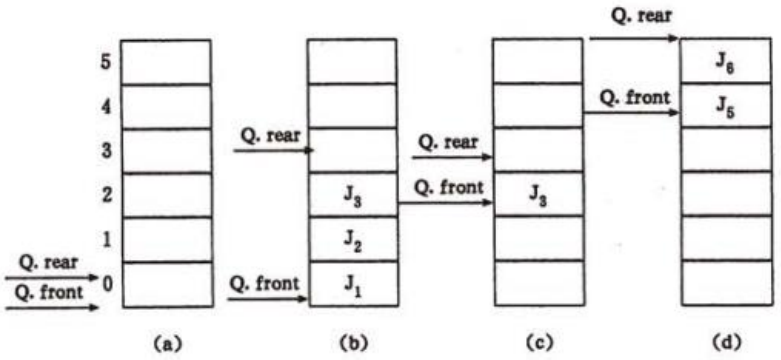


图 3.12　头、尾指针和队列中元素之间的关系

(a) 空队列；(b) J_1、J_2 和 J_3 相继入队列；(c) J_1 和 J_2 相继被删除；

(d) J_4、J_5 和 J_6 相继插入队列之后 J_3 及 J_4 被删除

离散事件

按时间发生的先后顺序进行处理，这种模拟程序称作事件驱动模拟

第四章：串

就是**字符串（string）**

汇编和语言的编译程序中，源程序和目标程序都是字符串数据

计算机上的非数值处理的对象基本上是字符串数据

上古年代字符串是作为常量输入输出，后来才产生的字符串处理

基本术语：字符串的长度、空串（null string）、子串、主串、空格串（blank string 不是空串）

当且仅当这两个串的值相等，则称这两个串是相等的

串的逻辑结构和线性表极为相似，但基本操作却有很大差别

存储方式：

定长顺序存储：串的实际长度可在预设范围内随意，超过的串值会被舍去，称之为“截断”

想解决截断产生的某些问题，可以用堆分配存储

堆分配存储：仍以一组地址连续的存储单元存串值字符序列，但它们的存储空间是在程序执行过程动态分配而得

堆分配特点：既有顺序存储结构的特点，处理方便，在操作中对串长又没有任何限制，更灵活。因此在串处理的应用程序中也常被选用

在C语言中就存在一个称作为堆的自由存储区，并由C语言的动态分配函数malloc()和free()来管理

块链存储：和线性表的链式存储结构相类似

在联接操作等方面有一定方便之处，但不如前两者灵活，占用存储量大且操作复杂

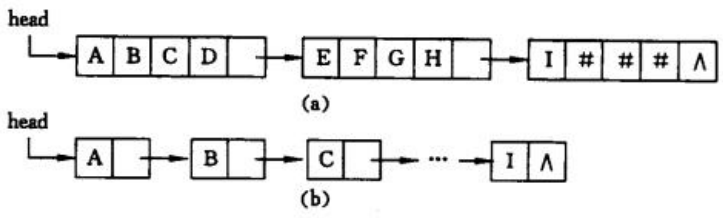


图 4.2　串值的链表存储方式

(a) 结点大小为 4 的链表；(b) 结点大小为 1 的链表

串值不必建立双向链表

存储密度：

$$存储密度=\frac{串值所占的存储位}{实际分配的存储位}$$

串的字符集的大小也是一个衡量效率的重要因素

第五章：数组和广义表

数组（array）

一旦被定义，它的维数和维界就不再改变，**一般不做插入或删除操作**。

一旦建立了数组，结构中的数据元素个数与元素之间的关系就**不在发生变动**。

因此采用**顺序存储结构**表示数组

随机存储结构：拥有计算各个元素存储位置的时间相等，所以存取数组中任意元素的时间也相等的特性存储结构

矩阵的压缩存储（矩阵大学没听课……抽空看看）

用高级语言编制程序时，都是用二维数组来存储矩阵元。

压缩存储：未解决阶数很高的矩阵的空间存储问题。为多个至相同的元只分配一个存储空间，对零元不分配空间。

特殊矩阵：至相同的元素或者零元素在矩阵中的分布有一定规律。

稀疏矩阵：非零元较零元少，且分布没有一定规律。这类矩阵的压缩存储比特殊矩阵复杂

***一个稀疏矩阵的转置矩阵仍然是个稀疏矩阵

***两个稀疏矩阵相乘的乘积不一定是稀疏矩阵

三元组顺序表：又称有序的双下标法，特点是非零元在表中按行序有序存储，因此便于进行依行顺序处理的矩阵运算，然而，若需按行号存取某一行的非零元，则需从头开始查找。

十字链表：当矩阵的非零元个数和位置在操作过程中变化较大时，就不宜采用顺序存储结构来表示三元组的线性表，这时采用链式存储机构表示更恰当。

十字链表表述：每个非零元既是某个行链表中的一个结点，又是某个列链表中的一个结点，整个矩阵构成了一个十字交叉的链表，故而得名十字链表。可用两个分别存储行链表的头指针和列链表的头指针的一维数组表示

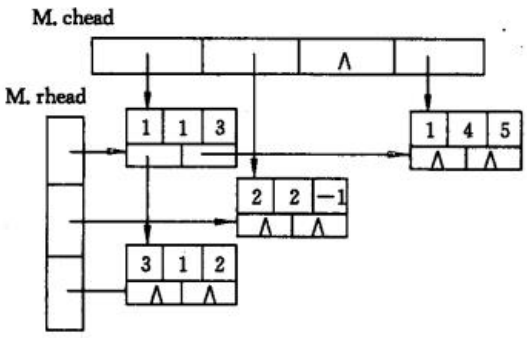


图 5.6　稀疏矩阵 M 的十字链表

广义表的定义

广义表是线性表的推广，也有人称其为列表（lists）广泛地用于人工智能等领域的表处理语言LISP语言，把广义表作为基本的数据结构，就连程序也表示为一系列的广义表

广义表一般记作

$$LS=(\alpha_1,\alpha_2,\cdots,\alpha_n)$$

LS 是广义表$(\alpha_1,\alpha_2,\cdots,\alpha_n)$的名称，$n$ 是它的长度。

这里面α既可以是单个元素，也可以是广义表，分别成为LS广义表的**原子**和**子表**

习惯上用大写字母表示广义表的名称，小写字母表示原子

第一个元素成为**表头（Head）**，其余元素都称作该表的**表尾（Tail）**

显然，广义表的定义是一个递归的定义，因为描述广义表时又用到了广义表的概念。

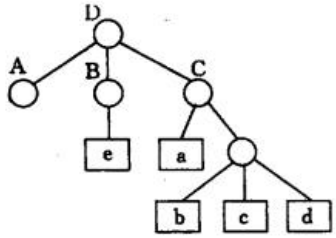


图 5.7　列表的图形表示

1. 列表的元素可以是子表，而子表的元素还可以是子表
2. 列表可为其他列表共享
3. 列表可以是一个递归的表，其列表也可以是其本身的一个字表

广义表的存储结构

由于元素可以具有不同结构，难用顺序存储结构表示，通常采用链式存储结构，每个数据元素可用一个结点表示。

表结点：用以表示列表

原子结点：用以表示原子

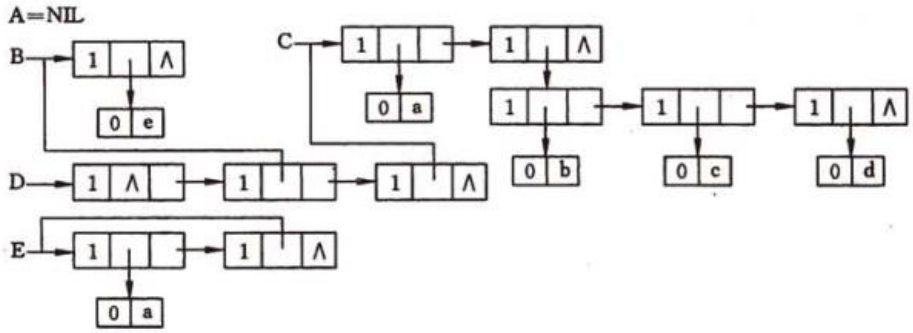


图 5.9　广义表的存储结构示例

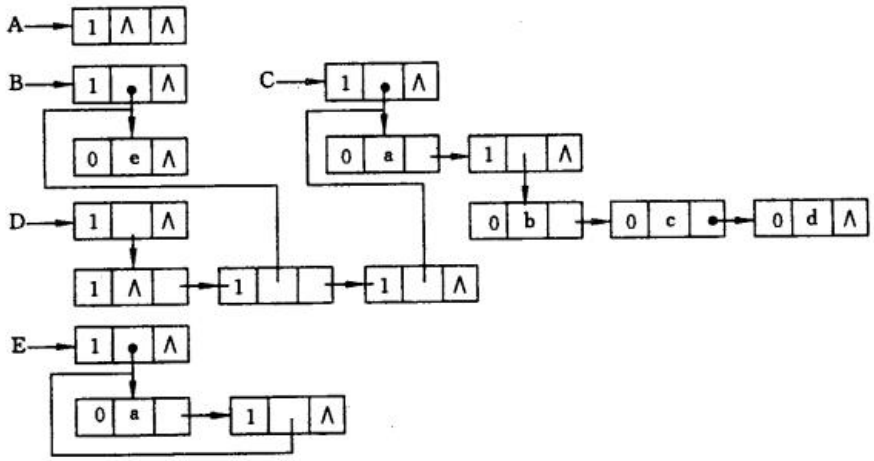


图 5.11　列表的另一种链表表示

m元多项式的表示

一般情况下使用的广义表既非是递归表，也不为其他表所共享。

可以这样理解，广义表中的一个数据元素可以是另一个广义表，一个m元多项式的表示就是广义表的这种应用的典型实例

一元多项式：用一个长度为m且每个数据元素有两个数据项的线性表来表示。

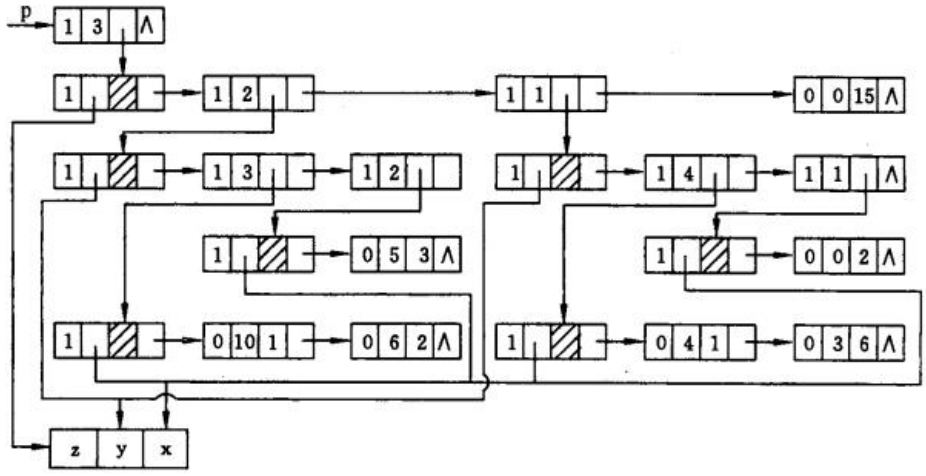


图 5.12　三元多项式 $p(x,y,z)$的存储结构示意图

第六章：树和二叉树

树形结构是一类重要的非线性数据结构，其中以树和二叉树最为常用。

树是以**分支关系**定义的层次结构，如族谱和组织结构，而在计算机领域中，编译程序中用树来表示源程序的语法结构。

在数据库系统中，树形结构也是信息的重要组织形式之一。

树（Tree）是n个结点的有限集。

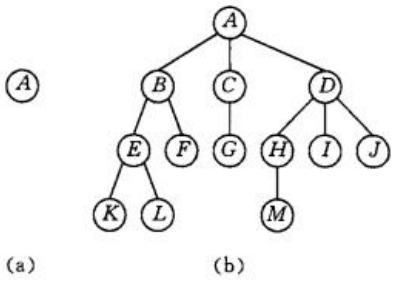


图 6.1　树的示例

(a) 只有根结点的树；(b) 一般的树

层次

1

2

3

4

1. 有且只有一个特定的成为**根（Root）**的结点

2. 每个集合本身又是一个树，称为根的**子树（SubTree）**

3. 树的结构定义是一个递归的定义，即在树的定义中又用到了树的概念

**术语：

度（Degree）结点拥有的子树数

叶子（Leaf）度为0的结点，又称终端结点

分支结点：度不为0的结点，又称非终端节点

孩子（Child）结点的子树的根

双亲（Parent）……

层次（Level）从根开始定义的，一层一层的

深度（Depth）树中结点的最大层次

有序树：树中结点的各子树看成从左至右是**有次序**的（即不能互换）

无序树：你懂的

森林（Forest）是m课互不相交的树的集合，对树中的每个结点而言，其子树的集合即为森林。由此也可以**森林和树相互递归的定义来表示树**

二叉树

二叉树（Binary Tree）特点是每个结点至多只有两棵子树，而且子树有左右之分且次序不能任意颠倒

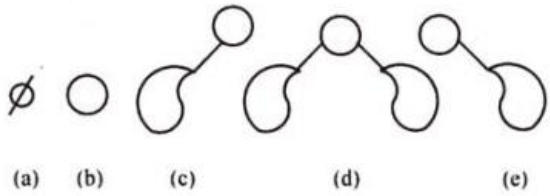


图 6.3　二叉树的 5 种基本形态

(a) 空二叉树；(b) 仅有根结点的二叉树；(c) 右子树为空的二叉树；

(d) 左、右子树均非空的二叉树；(e) 左子树为空的二叉树

二叉树的性质：

性质 1 在二叉树的第 i 层上至多有 2^{i-1} 个结点($i \geqslant 1$)。

性质 2 深度为 k 的二叉树至多有 2^k-1 个结点,($k \geqslant 1$)。

性质 3 对任何一棵二叉树 T,如果其终端结点数为 n_0,度为 2 的结点数为 n_2,则 $n_0=n_2+1$。

这图逼死强迫症啊- -##

满二叉树：一颗深度为k且有2^k -1个结点的二叉树

完全二叉树：深度为k，有n个结点的二叉树，当且仅当其每一个节点都与深度为k的满二叉树中编号为1至n的结点一一对应时就叫完全二叉树

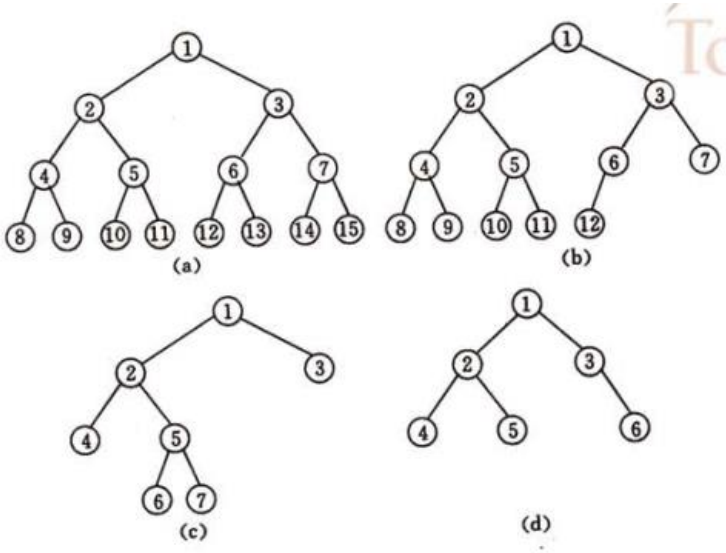


图 6.4 特殊形态的二叉树

(a) 满二叉树；(b) 完全二叉树；(c)和(d)非完全二叉树

二叉树的存储结构

1.顺序存储结构

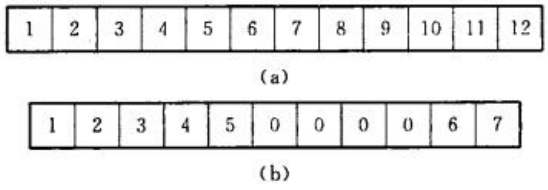


图 6.6 二叉树的顺序存储结构

(a) 完全二叉树；(b)一般二叉树

2.链式存储结构

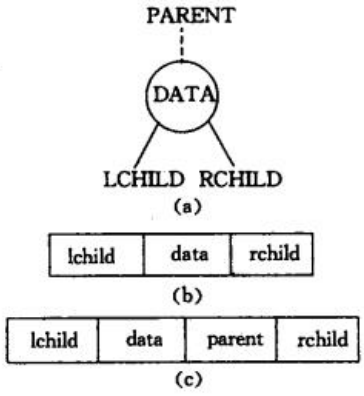


图 6.7　二叉树的结点及其存储结构

（a）二叉树的结点；

（b）含有两个指针域的结点结构；

（c）含有三个指针域的结点结构

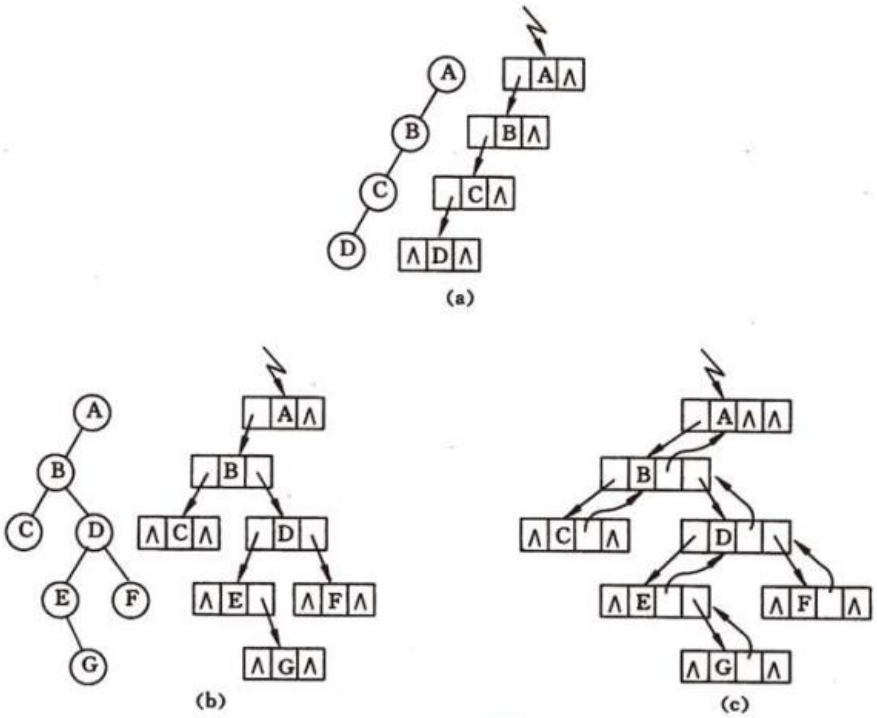


图 6.8　链表存储结构

（a）单支树的二叉链表；（b）二叉链表；（c）三叉链表

遍历二叉树

以一定规则将二叉树终结点排列成一个线性序列，得到二叉树中结点的先序序列或中序序列或后序序列，这实质上是对一个非线性结构进行线性化操作，是每个结点（除首尾两个）在这些线性序列中有且仅有一个直接前驱和直接后继。

但是，当以二叉链表作为存储结构时，只能找到节点的左右child信息，而不能直接得到结点在任一序列中的前驱和后继信息，这种信息只有在遍历的动态过程中才能得到。

辣么问题来了，如何保存这种在遍历过程中得到的信息呢？

在每个节点上增加两个指针域fwd和bkwd，分别指示结点在依任一次序遍历时得到的前驱和后继信息。这样做就能使结构的存储密度大大降低。而另一方面，再有n个结点的二叉链表中必定存在n+1个空链域。这就有个想法，能否利用这些空链域来存放结点的前后信息。

试作如下规定：若结点有左子树，则其 lchild 域指示其左孩子，否则令 lchild 域指示其前驱；若结点有右子树，则其 rchild 域指示其右孩子，否则令 rchild 域指示其后继。为了避免混淆，尚需改变结点结构，增加两个标志域

lchild	LTag	data	RTag	rchild

其中：

$$LTag=\begin{cases}0 & \text{lchild 域指示结点的左孩子}\\ 1 & \text{lchild 域指示结点的前驱}\end{cases}$$

$$RTag=\begin{cases}0 & \text{rchild 域指示结点的右孩子}\\ 1 & \text{rchild 域指示结点的后继}\end{cases}$$

线索链表：以上这种结点结构构成的二叉链表作为二叉树的存储结构，叫做线索链表。

线索：指向结点前驱和后继的指针

线索二叉树（Threaded Binary Tree）：有线索的二叉树

线索化：对二叉树以某种次序便利使其变成线索二叉树的过程

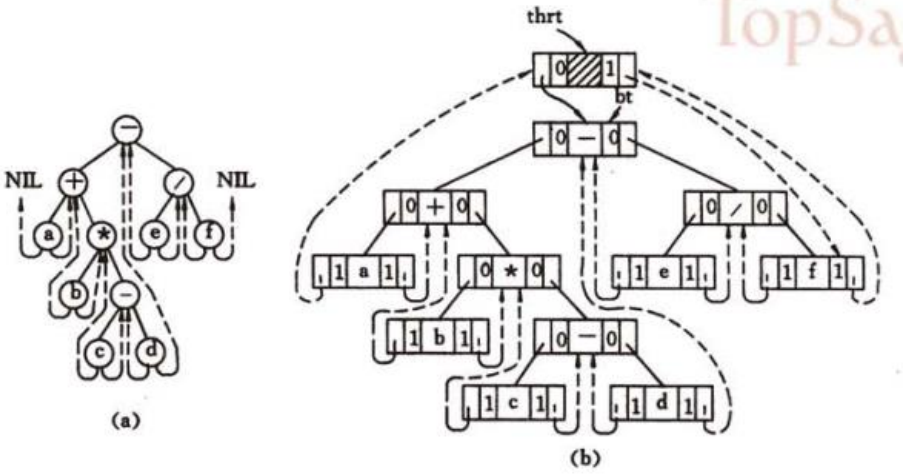


图 6.11　线索二叉树及其存储结构

(a) 中序线索二叉树；(b) 中序线索链表

树和森林的遍历

第一种：先根遍历树：先访问树的根节点，然后依次先根遍历根的每棵子树。

第二种：后根遍历树：先一次后根遍历每棵子树，然后访问根结点

赫尔曼树

赫尔曼（Huffman）树，又称最优树，是一类带权路径长度最短的树

路径：从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径

路径长度：路径上的分支数目

树的路径长度：从树根到每一结点的路径长度总和

数的带权路径长度：书中所有叶子节点的带权路径长度之和通常记作

$WPL=\sum_{k=1}^{n}w_k l_k$ 。　话说……书中哪里提到了带权？？？啥啊？？

如何构造赫尔曼树：

1.根据给定的n个权值｛w1,w2,w3,...,wn｝构成n棵二叉树的集合F=｛T1,T2,T3,....,Tn｝，其中每棵二叉树Ti中只有一个带权wi的根节点，其左右子树均空

2.在F中选取两棵根节点的权值最小的树作为左右子树构造一棵新的二叉树，且设置新二叉树的根结点的权值为其左、右子树上根节点的权值之和。

3.在F中删除这两棵树，同时将新得到的二叉树加入F中

4.重复2，3，直到F只含一棵树为止，剩下的这棵单身树就是赫尔曼树

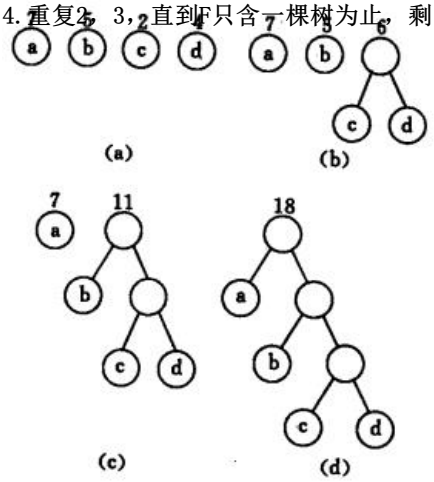


图 6.24　赫夫曼树的构造过程

赫夫曼编码

电报文字是专程二进制的字符串，长度都一样，排成串串没逗号也能识别。但为了让传值尽可能短，就想用短前缀代表常用字母，长前缀代替不常用的。辣么问题来了，长度不一样还没有逗号肯定是没办法准切实别的；

所以若要设计长短不等的代码，必须是每个字符的编码都不一样，这种编码称作前缀编码。

利用二叉树设计的二进制编码便成为赫尔曼编码

树的计数

称二叉树T和T’相似是指：两者都为空树或者两者均不为空树，且左右子树分别相似。

称二叉树T和T’等价是指：二者相似，且所有对应结点上的数据元素也要相同

第七章：图

图（Graph）是一种较线性表和树更为复杂的**数据结构**。

线性表：数据元素之间仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继

树形结构：数据元素之间有明显的层次关系，且每层上的数据元素可能和下一层中多个元素（即child）相关，但只能和上一层中一个元素（即parent）相关

图形结构：节点间的关系可以是任意的，图中的任意两个数据元素之间都可能相关。

ps.可看看“离散数学”中图的理论

在图中的数据元素通常称做**顶点**(Vertex)，V 是顶点的有穷非空集合；VR 是两个顶点之间的关系的集合。若$\langle v,w\rangle \in VR$，则$\langle v,w\rangle$表示从 v 到 w 的一条**弧**(Arc)，且称 v 为**弧尾**(Tail)或初始点(Initial node)，称 w 为**弧头**(Head)或终端点(Terminal node)，此时的图称为**有向图**(Digraph)。若$\langle v,w\rangle \in VR$ 必有$\langle w,v\rangle \in VR$，即 VR 是对称的，则以无序对(v,w)代替这两个有序对，表示 v 和 w 之间的一条**边**(Edge)，此时的图称为**无向图**(Undigraph)。例如图7.1(a)中 G_1 是有向图，定义此图的谓词 $P(v,w)$则表示从 v 到 w 的一条单向通路。

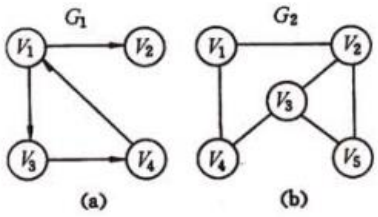


图 7.1　图的示例

(a) 有向图 G_1；(b) 无向图 G_2

$$G_1 = (V_1, \{A_1\})$$

其中：

$$V_1 = \{v_1, v_2, v_3, v_4\}$$

$$A_1 = \{\langle v_1, v_2\rangle, \langle v_1, v_3\rangle, \langle v_3, v_4\rangle, \langle v_4, v_1\rangle\}$$

图 7.1(b)中 G_2 为无向图。

$$G_2 = (V_2\{E_2\})$$

其中 $V_2 = \{v_1, v_2, v_3, v_4, v_5\}$

$$E_2 = \{(v_1, v_2), (v_1, v_4)(v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_5)\}$$

ps.迷之解释，全是看不懂的东西……这章先搁置，等境界高了再补上

第八章：动态存储管理

悠久的某个年代：程序员自己负责数据的存储管理

有了高级语言后：不需要直接和内存地址打交道，存储单元都由编辑变量（标识符）来表示，内存地址都是编译或执行进行分配

在不同的动态存储管理系统中，请求分配的内存量大小不同。

系统每次分配给用户都是一个地址连续的内存区，又称占用块。

不管什么系统，刚开工时内存区都是一个空闲块，在编译程序中管他叫‘堆’。

运行初期，内存区基本被分隔成两部分：**低地址区包含若干占用块；高地址区是一个空闲块**

而这也是下图情况出现的原因

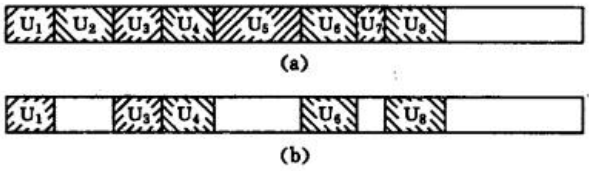


图 8.1　动态存储分配过程中的内存状态

(a) 系统运行初期；(b) 系统运行若干时间之后

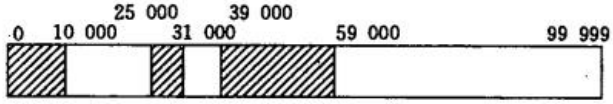
辣么此时又有新用户进入系统请求分配内存该肿么办？

第一种方案：

系统继续从高地址的休闲块中进行分配，而**不理会用过的低地址区**，直到高地址区没法满足分配时系统才会回头收拾空闲块，重新组织内存，将所有空闲块堆成一个大休闲块

第二种方案：

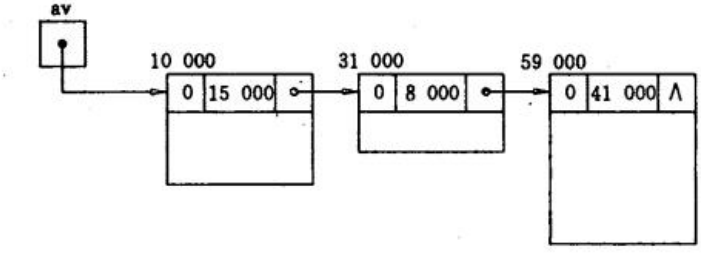
用户**一旦运行结束，便将它所占内存区释放成空闲块**，同时每当新用户请求分配内存时，系统需要巡视整个内存去所有空闲块，找到合适的空闲块配给他。针对这一方案，系统需建立一张**记录所有空闲块的‘可利用空间表’**，此表的结构可以是目录表，也可以是链表。



(a)

起始地址	内存块大小	使用情况
10 000	15 000	空闲
31 000	8 000	空闲
59 000	41 000	空闲

(b)



(c)

图 8.2　动态存储管理过程中的内存状态和可利用空间表

(a) 内存状态；(b) 目录表；(c) 链表

可利用空间表及分配方法

三种情况：

1.系统运行期间所有用户请求分配的存储量大小相同

做法：在系统开始时按需分割内存成大小相同的块，然后用指针链接成一个可利用空间表。

好处：内存大小相同，不用找合适的内存，每次用的时候把第一个拿走就可以。而用完后，空闲块在插入表头。

可见，这种表实质上是一个**链栈**。而且这是一种**最简单**的动态存储管理的方式。

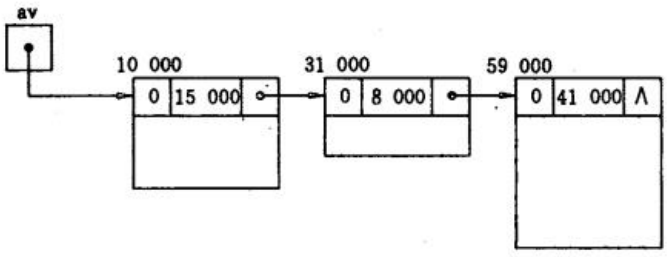
2.系统运行期间用户请求分配的存储量有若干种大小的规格

做法：按照结点大小来建立若干个可利用空间表，同一个表内结点大小相同，如下图。这样分配或回收时就根据自己的需求找对应的空间表。

这里有个特殊问题：当结点于请求相符的链表和结点更大的链表都是空的时，分配不能进行。可此时内存空间并不代表一定没有所需大小的连续空间**。这是由于在系统运行过程中，频繁出现小块的分配和回收，使得大结点列表中的空闲块被拿出去，用过后被切成沫沫插入到小结点的链表中。在这种情况下还想要让系统继续运行，就必须重新组织内存，即执行“存储紧缩”的操作。这个操作可以自行脑补。

3.系统在运行期间分配给用户的内存块的大小不固定，可以随请求而变。（通常的操作系统就是）

做法：因为可以随请求而变，可利用空间表的结点即空闲块的大小也是随意的。系统刚工作时整个内存是个空闲块，此时表中只有一个大小是整个内存区的结点，随着分配和回收的进行，表中结点大小和个数也随之变化，可参考下图



(c)

由于链表中结点大小不同，则结点的结构与前两种情况也不同，节点中除标志域和链域之外，尚需一个结点大小域（size）,用来指示休闲块的存储量。如下图，space域就是一个地址连续的内存空间

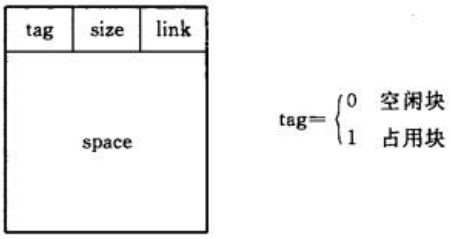


图 8.4 空闲块的大小随意的结点结构

至于分配时该如何分配？这有有三种不同的分配策略（WTF??）

（1）首次拟合法

从表头指针开始查表，将找到的第一个大小不小于n的空闲块的一部分分配给用户。可利用空间表本身既不按节点的初始地址有序，也不按结点的大小有序。再回收时，只要将释放的空闲块插入在表的表头即可。

（2）最佳拟合法

将可利用空间表中一个不小于n且最接近n的空闲块的一部分分配给用户（就是找大小最适合的那个）。则系统在分配前首先要对表从头到尾扫视一遍，然后从中找到一块不小于n且最接近n的空闲块进行分配。分配时，为避免每次分配都要扫表，通常预先设定表的结构按空间块的大小自小到大有序。

（3）最差拟合法

将可利用空间表中不小于n且是链表中最大的空闲块的一部分分配给用户。显然为节省时间，表结构空间块大小就要从大到小排。这样无需查找，直接从链表中删掉第一个结点，将它分给用户，用剩下的再插回到适当位置去。

小总结：**一般来说，最佳拟合法适合请求分配的内存大小范围较广的系统。最差拟合法适合请求分配的内存大小范围较窄的系统；首次拟合法适用于系统实现不掌握运行时间可能出现的请求分配和释放信息的情况**

从时间上来看：首次拟合法分配时需要查表，回收时直接插在表头就行；最差拟合正好想反，分配时直接调表头的结点，用完在查表往里塞；**最佳拟合则最费时间**，因为分配释放都要查表。

所以选择时要考虑：用户的逻辑要求；请求分配量的大小分布；分配和释放的频率以及效率对系统的重要性等等等等

另外，回收空闲块是还需考虑一个“**结点合并**”的问题。因为系统不断拿还的过程中，大空闲块逐渐碎成小占用块，用完后也是把这些变小的块以结点身份插到表中，以致再出现大容量请求时，发现在座各位空闲块都是渣渣……为了更有效地利用内存，就要求系统回收时应考虑将地址相邻的空闲块合并成尽可能大的节点。换句话说，**回收空闲块时，首先检查地址与他相邻的内存是否是空闲块。**

边际标识法

友情提示：这个是为了解决上述的**可利用空间表及分配方法**的第三种情况**3.系统在运行期间分配给用户的内存块的大小不固定，可以随请求而变。（通常的操作系统就是）**

边际标识法（boundary tag method）是操作系统中用以进行动态分区分配的一种存储管理方法。

系统将所有的空闲块链接在一个双重循环链表结构的可利用空间表中；分配可按首次拟合法进行，当然最佳拟合也行。

此时系统特点是：在每个内存区的头部和底部两个边界上分别设有标识，用来表示这块儿是空闲块还是占有块，使得在回收空闲块时容易判别在物理位置上相邻的内存区域是否是空闲块，好用来拼个大的空闲块。

过程看下图

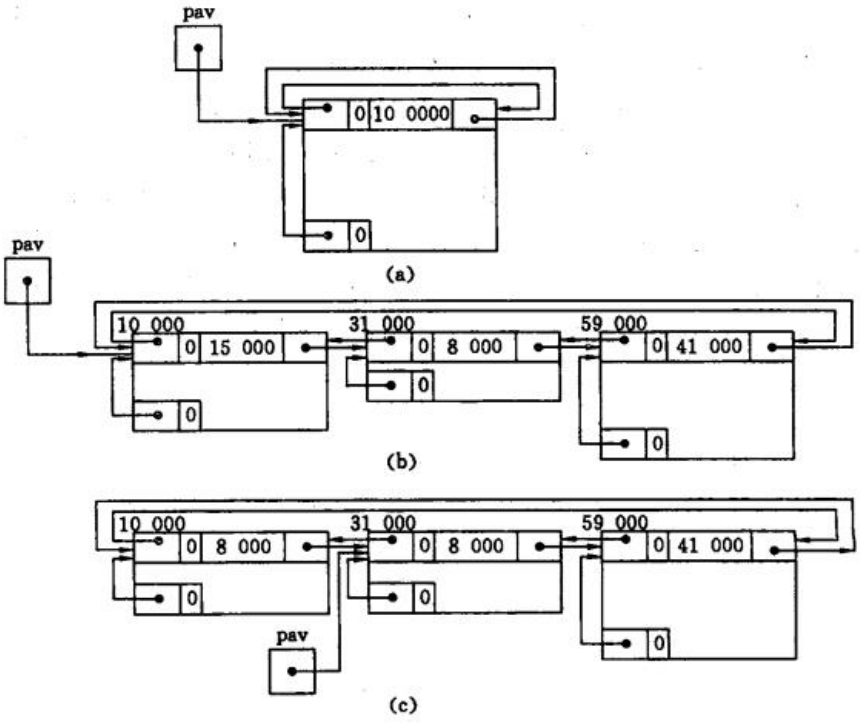


图 8.6　某系统的可利用空间表

(a) 初始状态；(b) 运行若干时间后的状态；(c) 进行再分配后的状态

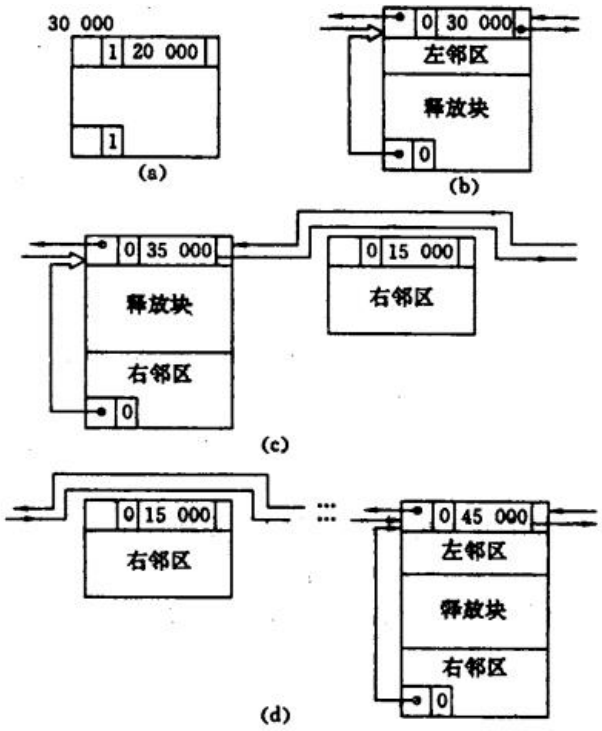


图 8.7　回收存储块后的可利用空间表

(a) 释放的存储块；　(b) 左邻区是空闲块的情况；

(c) 右邻区是空闲块的情况；　(d) 左、右邻区均是空闲块的情况

伙伴系统

伙伴系统（buddy system）是操作系统中用到的另一种动态存储管理方法。

它和边界标识法类似（哪来的？？），在用户提出申请时，分配一块大小“恰当”的内存去给用户；反之，在用户释放时即回收。而不同点在于：伙伴系统中，无论是占用块还是空闲块，大小均为2的K次幂。

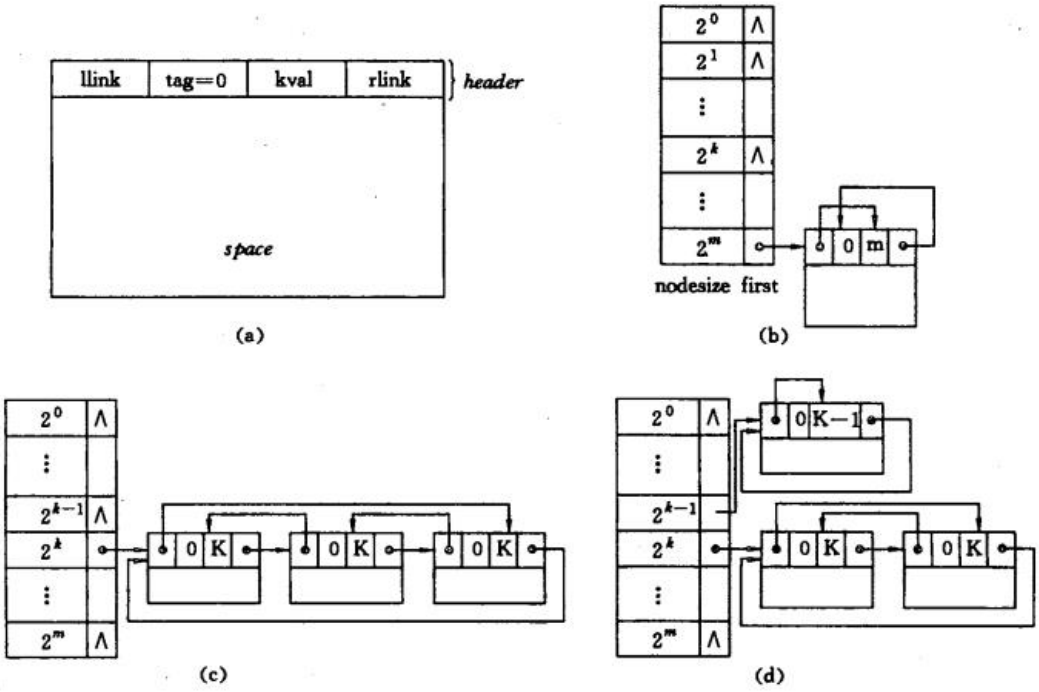


图 8.8　伙伴系统中的可利用空间表

(a) 空闲块的结点结构；(b) 表的初始状态；(c) 分配前的表；(d) 分配后的表

上述的问题都是如何使用可利用空间表来进行动态存储管理。

这个表的特点：在用户请求存储时进行分配；释放存储时进行回收，即系统是应用户的需求来进行存储分配和回收的。

也就是说，在这类存储管理系统中，**用户必须明确给出“请求”和“释放”的信息**。

而如果在多用户**分时并发**的操作系统中，当用户程序进入系统时请求分配存储区；反之，当用户程序执行完毕退出系统时释放所占存储。

栗子：在C语言中，用户通过malloc和free两个函数来请求分配和释放存储的。但有时会因为用户的疏漏或结构本身的原因致使系统在不恰当的时候或没有进行回收而产生“无用单元”或“悬挂访问”的问题。

无用单元：用户不再使用而系统没有回收的结构和变量

悬挂访问：所释放的节点被再分配时指针仍继续访问之前那个，后果自行脑补

如何解决呢？

（1）使用访问计数器：在所有子表或广义表上增加一个表头结点，并设立一个计数域，他的值为指向该子表或广义表的指针数目。只有当该计数域的值为零时，此子表或广义表的结点才被释放

（2）收集无用单元：在程序运行过程中，对所有结点，不管是否还有用，都不回收，直到整个可利用空间表为空。此时才暂时中断执行程序，将所有当前不被使用的结点链接在一起，成为一个新的可利用空间表，然后程序继续进行。

辣么问题又来了……一般情况下是无法辨别哪些节点是当前未被使用的，而对于一个正在运行的程序，哪些节点在被使用是可以查到的，这就需要遍历之后过滤一下，剩下的就是没用上的结点了。

因此收集无用单元分两步：

第一步：对所有占用节点加上标志。可以设置成1

第二步：对整个可利用存储空间顺序扫描一遍，将所有标志为0的结点连接成一个新的可利用空间表。

第一步是在极其困难的条件（可利用存储几乎耗尽时）进行的，因此猿的精力就集中在**研究标志算法**上。

存储紧缩

在整个动态存储管理过程中，不论何时，可利用空间都是一个地址连续的存储区，在编译程序称之为“堆”，每次分配都是从这个可利用空间中划出一块。

具体实现方法：设一个指针，称之为堆指针，始终指向堆的最低（或最高）地址。申请存储块时就让指针移动相应存储单位，移动之前堆指针的值就是分配给用户的存储块的初始地址。

这种方法分配起来很简单，回收就麻烦了。由于系统的空闲块始终是一个地址连续的存储块，因此回收时必须将空闲块合并到整个堆上去才能重新使用，这才是“存储紧缩”的任务。

两种做法：

1. 一旦有用户释放存储块即进行回收紧缩
2. 在程序执行中不回收用户随时释放的空闲块，等到存储块不够或堆指针指向最高地址时才存储紧缩

具体步骤：

1. 计算占用块的新地址
2. 修改用户的初始变量表，以便在存储紧缩后用户程序还能接着进行
3. 检查每个占用块存储的数据，若有指向其他存储块的指针则需作相应更改。
4. 将所有占用块迁移到新地址去。
5. 将堆指针附以新值（即紧缩后的空闲存储区的最低地址）

*****由此可见，存储紧缩法比无用单元收集法更为复杂，前者不仅要传送数据，还要修改所有占用块的指针值，所以，存储紧缩也是一个系统操作，不到万不得已不要用

第九章：查找

查找表（Search Table）是由同一类型的数据元素构成的集合，由于集合是那种完全松散的关系，因此查找表是一种非常灵便的数据结构。

常进行的操作有：

（1）查询某个“特定的”数据元素是否在查找表中

（2）检索某个“特定的”数据元素的各种属性

（3）增

（4）删

动态查找表（Dynamic Search Table）在查找过程中改变了查找表内容（增删操作），那这类表就叫这个了。

关键字（Key）是数据元素中某个数据项的值，用它可以识别一个数据元素。

主关键字（Primary Key）此关键字作为唯一能识别该数据元素的标识

次关键字（Secondary Key）用来识别多个数据元素的标识

静态查找表

1.**顺序查找（Sequential Search）**

从表中最后一个记录开始挨个找

缺点:平均查找长度较大，当数据量大时查找效率低。

优点：算法简单且适应面广

2.**折半查找（Binary Search）** 有序表中查找

先确定个范围，然后逐步缩小范围

效率比顺序查找高，但**只适用于有序表**

哈希表

概念：根据设定的哈希函数H（key）和处理冲突的方法将一组关键字映像到一个有限的连续的地址集上，应以关键字在地址集中的“像”作为记录在表中的存储位置。

这一映像过程称为哈希造表或**散列**，所得存储位置称哈希地址或**散列地址**

在记录的存储位置和他的关键字之间建立一个确定的对应关系f，是关键字和结构中一个唯一的存储位置相对应。这个对应关系f就是**哈希（Hash）函数**

1.哈希函数是一个映像，设定很灵活，只要使得任何关键字由此得到的哈希函数值都落在表长允许范围内即可。

2.对不同关键字可能得到同一哈希地址的现象叫做**冲突（collision）**，具有相同函数值的关键字对该哈希函数来说称作**同义词（synonym）**

冲突

这种现象给建表带来了困难，而且冲突只能尽可能的少，却**不能完全避免**。因为哈希函数是从关键字集合到地址集合的映像。一般情况下关键字集合比较大，地址集合中的仅为哈希表中的地址值。

第十章：内部排序

排序（Sorting）将一个数据元素的任意序列，按照关键字有序的重新排列

由于排序数量不同，设计的存储器也不同，所以分个类

1.内部排序

待排序记录存放在计算机随机存储器中进行的排序过程

2.外部排序

待排序记录的数量很大，一直内存一次装不下，在排序过程中尚需对外存进行访问的排序过程

排序操作：

1.比较两个关键字的大小

2.将记录从一个位置移动到另一个位置

待排序的记录序列可有3中存储方式：

1.存放在地址连续的一组存储单元上，类似于线性表的顺序存储结构。

2.存放在静态链表中，记录之间的次序关系由指针指示，这样实现排序时不需移动，改指针就行。这种方法实现的排序叫（链）表排序

3.本身存储在一组地址连续的存储单元内，同时另设一个能指向各个记录存储位置的地址向量，排序时不改变序列本身，而是移动地址向量中这些记录的“地址”。待排序结束后，按照调整后的地址向量来调整记录的存储位置；这种方式实现的排序又称地址排序

内部排序的几类：插入排序、交换排序、选择排序、归并排序、计数排序

按工作量来区分：1.简单的排序方法，时间复杂度O（n^2）

2.先进的排序方法，时间复杂度O（nlogn）

3.基数排序，时间复杂度O（d•n）

插入排序

直接插入排序（Strainght Insertion Sort）最简单的排序，直接往里插。时间复杂度O（n^2）

折半插入排序 脑补

希尔排序

希尔排序（Shell’s Sort）又称“缩小增量排序”（Diminishing Increment Sort）

基本思想：先将待排序列分割成若干子序列分别直接插入排序，待整个序列的记录大概有序时，在全体来一次直接插入排序

快速排序

起泡排序（Bubble Sort）的一种改进，基本思想是：通过一趟排序将待排记录分割成独立的两部分，其中一部分的关键字均比另一部分的关键字小，则可分别多这两部分记录继续排序，以达到整个序列有序。

***小总结：时间上看，快排的平均性能由于之前各种排序方法。空间上看，之前的都只需要一个记录的附加空间即可，但快排需要一个栈空间来实现递归。

第十二章：文件

文件（file）是有大量性质相同的记录组成的集合，习惯上称存储在主存储器中的集合为表，存储在二级存储器上的记录集合为文件。

系统中的文件：仅是一堆的连续的字符序列，无结构、无解释。

数据库中的文件：带有结构的记录的集合。