



数据结构笔记

作者: KoSaking

组织: 计合二刺螈根据地

时间: 2023 年 2 月 19 日

版本: 5.0.0

教材: 严蔚敏、吴伟民《数据结构 (C 语言版)》



天鹅们也是如此，在看不见的水下拼命打腿前进。 —— 《轻音少女》

目 录

1	绪论	1	5.7.2 森林与二叉树的转换	42
1.1	逻辑结构	1	5.8 哈夫曼树	43
1.2	存储结构	1	5.8.1 哈夫曼树的概念	43
1.3	抽象数据类型	2	5.8.2 哈夫曼编码	43
1.4	算法	3	6 图	44
1.4.1	算法设计的要求	4	6.1 图的基本概念	44
1.4.2	算法效率的度量	4	6.2 图的存储结构	45
2	线性表	7	6.2.1 邻接矩阵	45
2.1	线性表的类型定义	7	6.2.2 邻接表	46
2.2	线性表的顺序表示和实现	8	6.3 图的遍历	49
2.3	线性表的链式表示和实现	12	6.4 最小生成树	51
2.4	循环链表	14	6.4.1 Prim 算法	52
2.5	双链表与循环双链表	15	6.4.2 Kruskal 算法	53
2.6	综合：一元多项式的表示及相加	16	6.5 拓扑排序	54
3	栈和队列	17	6.6 关键路径	55
3.1	顺序栈	17	6.7 Dijkstra 算法	56
3.2	链栈	19	6.8 Floyd 算法	57
3.3	顺序队列与循环队列	20	7 查找	59
3.4	链队列	23	7.1 顺序查找	59
3.5	递归	25	7.2 二分查找	60
4	数组	27	7.3 索引查找	61
4.1	数组的定义与存储	27	7.4 二叉排序树	61
4.2	矩阵的压缩存储	28	7.5 AVL 树	63
4.2.1	特殊矩阵	28	7.6 B- 树和 B+ 树	65
4.2.1.1	对称矩阵	28	7.6.1 B- 树	65
4.2.1.2	三对角矩阵	29	7.6.2 B+ 树	65
4.2.2	稀疏矩阵	29	7.7 哈希表	66
4.3	广义表	31	8 内部排序	68
5	树和二叉树	33	8.1 插入排序	68
5.1	树的定义与基本术语	33	8.2 希尔排序	69
5.2	二叉树	34	8.3 冒泡排序	70
5.3	二叉树的存储结构	34	8.4 快速排序	71
5.4	二叉树的遍历	35	8.5 简单选择排序	71
5.5	与二叉树相关的递归算法设计	37	8.6 堆排序	72
5.6	线索二叉树	40	8.7 归并排序	73
5.7	树与森林	41	8.8 基数排序	74
5.7.1	树的存储结构	41	A 代码说明	75

第 1 章 绪论

在正式学习数据结构之前，我们需要了解一些基本概念。

数据是对客观事物的符号表示，在计算机科学中是指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，包括文字、表格、图像等。

数据的基本单位是**数据元素**，也称为结构体、**结点**、**记录**等。含有大量记录的线性表称为**文件**。

数据对象是具有相同类型的数据元素的集合，在数据结构中除特别指定外数据通常都是数据对象。

有些数据元素是由若干个**数据项**（或称**域**、**字段**、**属性**等）组成的，数据项是数据的不可分割的最小单位。

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。这些数据元素不是孤立存在的，而是有着某种关系，这种关系构成了某种结构。

其中，我们关注的重点在于数据结构。数据结构包括**逻辑结构**和**存储结构**两类，下面将会详细介绍。

1.1 逻辑结构

逻辑结构是数据元素之间的逻辑关系的整体，是数据结构在用户面前呈现的形式，其主要指数据元素之间的相邻关系，可分为以下四类。

1. **集合**：包含的所有数据元素同属于一个集合，是最松散的关系。
2. **线性结构**：包含的数据元素之间存在一对一的关系。
3. **树形结构**：包含的数据元素之间存在一对多的关系。
4. **图状结构**：包含的数据元素之间存在多对多的关系，也成为网状结构。

数据的逻辑结构可以采用二元组的方式进行描述，表示如下：

$$\begin{cases} S = (D, R) \\ D = \{d_i | 1 \leq i \leq n\} \\ R = \{r_j | 1 \leq j \leq m\} \end{cases} \quad (1.1)$$

其中， D 是数据元素的有限集合，即 D 是由有限个数据元素所构成的集合； R 是 D 上的关系的有限集合，即 R 是由有限个关系 r_j 所构成的集合； r_j 是指 $D \rightarrow D$ 的关系。

对于线性结构，第一个数据元素称为**开始数据元素**，最后一个数据元素称为**尾元素**，在中间的每一个元素都有一个前驱元素和一个后继元素。

注 每个关系 r_j 用有序偶对集合表示，一个有序偶对表示两个元素的关系，用尖括号表示有向关系，用圆括号表示无向关系。

1.2 存储结构

存储结构指的数据元素及其关系在计算机存储器中的存储方式（或称**映像**），也称为数据的**物理结构**。由于数据元素之间的关系在计算机中有**顺序映像**和**非顺序映像**两类，故存储结构可分为**顺序存储结构**和**链式存储结构**两类。

1. **顺序存储结构**：采用一组连续的存储单元存放所有的数据元素，而逻辑上相邻的元素，其存储单元也相邻，即顺序存储结构将数据的逻辑结构直接映射到存储结构。
2. **链式存储结构**：每个结点单独存储，无需占用一整块存储空间，但为了表示结点之间的关系，需要给每个结点附加指针字段，用于存放相邻结点在内存中的存储地址。

除了上述两种存储结构，若我们视 C 语言为一个执行 C 指令和 C 数据类型的虚拟处理器，则之后讨论的存储结构是数据结构在 C 虚拟处理器中的表示，称为**虚拟存储结构**。

逻辑结构和存储结构不是相互独立的两个概念：同一种逻辑结构可以有多种存储结构，在不同的存储结构中，实现同一种运算的算法可能不同；对于数据的运算而言，逻辑结构针对的是运算的定义，存储结构针对的是运算的实现；将逻辑结构映射为存储结构时，我们需要存储逻辑结构中的所有元素和元素之间的关系。

注 顺序存储结构可实现对各数据元素的随机存取，即顺序存储结构具有随机存取特性；在链式存储结构中，所有结点的地址不一定连续，一个数据元素的所有数据项占用一片连续空间，并通过增加指针域来表示元素之间的逻辑关系。

1.3 抽象数据类型

一般地，我们称插入、删除、查询等操作为**数据运算**，其可分为**运算定义**和**运算实现**两类。

1. 运算定义：确定运算的功能，是抽象的。
2. 运算实现：在存储结构上确定对应运算实现的算法，是具体的。

数据类型是一组性质相同的值的集合和定义在此集合上的一组操作的总称，其为高级程序设计语言中的一个基本概念，用于描述变量的存储方式，可分为以下两类。

1. **原子类型**：原子类型的值是不可分解的，如整型、实型、字符型、枚举类型、指针类型和空类型等。
 2. **结构类型**：结构类型的值是可分解的，且其成分可以是结构的，也可以是非结构的，如数组、结构体等。
- 由数据类型的概念，我们可引出**抽象数据类型**的定义。

定义 1.1 (抽象数据类型)

抽象数据类型 (Abstract Data Type, ADT) 指的是一个数学模型以及定义在此数学模型上的一组操作。ADT 的定义仅取决于其中的一组逻辑特性，即无论内部结构如何变化，只要它的数学特性不变，就不会影响外部的使用。

如定义 1.1 所述，ADT 的定义由一个值域和定义在该值域上的一组操作组成，若按其值的不同特性，可分为以下三类。

1. **原子类型**：属于原子类型的变量的值是不可分解的，但有时也有必要定义新的原子数据类型。
2. **固定聚合类型**：属于固定聚合类型的变量的值由确定数目的成分按某种结构组成，如复数。
3. **可变聚合类型**：相对于固定聚合类型而言，属于可变聚合类型的变量的值中成分数目不确定，如结构体。

显然，固定聚合类型和可变聚合类型可统称为结构类型。

与数据结构的形式定义类似，ADT 可以用三元组表示，即

$$(D, S, P) \quad (1.2)$$

其中， D 是数据对象， S 是 D 上的关系集， P 是对 D 的基本操作集。

我们也可以采用格式定义来描述 ADT：

```

1  ADT 抽象数据类型名
2  {
3      数据对象:<数据对象的定义>
4      数据关系:<数据关系的定义>
5      基本操作:<基本操作的定义>
6  }ADT 抽象数据类型名

```

其中，数据对象和数据关系的定义用伪码描述，基本操作的定义格式为

```

1  基本操作名(参数表)
2      初始条件:<初始条件描述>
3      操作结果:<操作结果描述>

```

在定义一个 ADT 时, 我们需要给出其名称和各运算名称及其功能描述, 且 ADT 需要通过固有数据类型 (高级编程语言中已实现的数据类型) 来实现。ADT 对一个求解问题从逻辑上进行了准确的定义, 故 ADT 由数据逻辑结构和运算定义两部分组成。

多形数据类型是指其值的成分不确定的数据类型。

例题 1.1 定义单个集合的抽象数据类型 ASet, 其中所有元素为正整数, 包含创建一个集合、输出一个集合和判断一个元素是否属于某集合的基本运算, 并在此基础上再定义两个集合运算的抽象数据类型 BSet, 包含集合的并集、交集和差集运算。

解 由题意有

```

1  ADT ASet
2  {
3      数据对象:D={d[i]|0<=i&&i<=n,n为正整数}
4      数据关系:R={(d[i],d[j])|0<=i&&i<=n&&0<=j&&j<=n,i与j都为正整数但i!=j}
5      基本操作:
6          void cset(&s,a,n)
7              初始条件:数组a已存在.
8              操作结果:由含n个元素的数组a创建一个集合s.
9          void dispset(s)
10             初始条件:集合s已存在.
11             操作结果:输出集合s.
12         int inset(s,e)
13             初始条件:集合s已存在.
14             操作结果:若元素e属于集合s,则返回1,否则返回0.
15     }ADT ASet
16     ADT BSet
17     {
18         数据对象:D={属于ASet的s[i]|i>=0&&i<=n,n为正整数}
19         数据关系:R={(s[i],s[j])|0<=i&&i<=n&&0<=j&&j<=n,i与j都为正整数但i!=j}
20         基本操作:
21             void add(s[1],s[2],s[3])
22                 初始条件:集合s[1]与s[2]已存在.
23                 操作结果:返回s[1]与s[2]的并集s[3].
24             void intersection(s[1],s[2],s[3])
25                 初始条件:集合s[1]与s[2]已存在.
26                 操作结果:返回s[1]与s[2]的交集s[3].
27             void sub(s[1],s[2],s[3])
28                 初始条件:集合s[1]与s[2]已存在.
29                 操作结果:返回s[1]与s[2]的差集s[3].
30     }ADT BSet

```

1.4 算法

算法是对特定问题求解步骤的一般描述, 它是指令的有限序列, 其中每条指令表示一个或多个操作。

性质 算法具有以下特性。

1. 有穷性: 一个算法必须对任何合法的输入值总是在执行有限步之后结束, 且每一步都在有限时间内完成。
2. 确定性: 算法中的每一条指令必须有确切的含义, 不会产生二义性。
3. 可行性: 算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

4. 输入：一个算法有零个或多个输入。
5. 输出：一个算法有一个或多个输出。

描述算法的方式有很多种，如采用自然语言或某种高级语言，也可以采用混合形式。通常我们采用 C/C++ 语言来描述算法。

1.4.1 算法设计的要求

日常生活中，人们总会在保证不出差错的情况下，尽可能高效率地完成各项任务，算法也是如此，故我们在设计算法的时候，需要满足以下要求。

1. **正确性**：算法应当满足具体问题的需求，这是最重要也是最基本的标准。
2. **可读性**：算法的逻辑必须是清晰的、简单的和结构化的。
3. **健壮性**：当输入数据非法时，算法也能够适当地作出反应或进行处理，而不会产生奇怪的输出结果。
4. **效率与低存储量需求**：算法要在追求高效率的同时尽可能地保证较低的存储量，效率与存储量与问题的规模有关。

1.4.2 算法效率的度量

计算机资源主要包括**计算时间**和**内存空间**，而算法分析就是分析算法占用计算机资源的情况，故我们在进行算法分析的时候主要分析的是算法的**时间复杂度**和**空间复杂度**。需要注意的一点是，算法分析的目的不是分析算法是否正确或是否容易阅读，主要是考察算法的时间和空间效率，以求改进算法或对不同的算法进行比较。

通常有两种衡量算法效率的方法，一是**事后统计的方法**，二是**事前分析估算的方法**。其中，由于事后统计法必须先运行依据算法编制的程序，且所得时间的统计量依赖于计算机的硬件、软件等环境因素，有时容易掩盖算法本身的优劣，故大多数情况下我们会选择事前分析估算的方法。

一个用高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

1. 依据的算法选用何种策略；
2. 问题的规模；
3. 书写程序的语言，对于同一个算法，实现语言的级别越高，执行效率越低；
4. 编译程序所产生的机器代码的质量；
5. 机器执行指令的速度。

若不考虑上述与计算机硬件和软件有关的因素，可以认为一个特定算法的“运行工作量”的大小只依赖于问题的规模，后者通常用整数量 n 表示。

一个算法是由**控制结构**（顺序、分支和循环）和**原操作**（固有数据类型的操作）构成的，而算法时间取决于两者的综合效果。我们定义一个语句的**频度**为该语句在算法中被重复执行的次数，并将算法中所有语句的频度之和记作 $f(n)$ ，其为问题规模 n 的函数。当 n 趋于无穷大时， $f(n)$ 的数量级称为**渐进时间复杂度**，简称为**时间复杂度**，记作 $T(n) = O(f(n))$ 。

下面给出时间复杂度的严格定义。

定义 1.2 (时间复杂度)

若存在常数 $c \neq 0$ 与足够大的常数 n_0 ，使得

$$\lim_{n \rightarrow n_0} \frac{|f(n)|}{|g(n)|} = c \quad (1.3)$$

成立，则此时我们为 $f(n)$ 找到了一个上界 $g(n)$ ，并用 $O(g(n))$ 表示 $f(n)$ 的数量级，称之为时间复杂度，记作 $f(n) = O(g(n))$ 。



时间复杂度表现了随着问题规模 n 的增大，算法执行时间的增长率与 $f(n)$ 的增长率是相同的。

一般地，对于以多项式形式表示的 $f(n)$ ，我们可以利用类似 k 阶无穷小的性质来得到其时间复杂度，故有以下定理成立：

定理 1.1 (多项式函数的时间复杂度的计算公式)

若 $f(n) = \sum_{i=0}^m a_i n^i$ 是一个 m 次多项式, 则 $T(n) = O(n^m)$ 。



定理 1.1 表明了算法中基本运算语句的频度 $f(n)$ 与 $T(n)$ 是同数量级的。通常情况下, 我们常用最深层循环内的语句中的原操作的执行频度来表示该算法的时间复杂度。

一个没有循环的算法中基本运算次数与问题规模 n 无关, 称为**常量阶**, 记作 $O(1)$; 一个只有单循环的算法中基本运算次数与问题规模 n 的增长呈线性增长关系, 称为**线性阶**, 记作 $O(n)$; 除此之外, 常见的时间复杂度还有**平方阶** $O(n^2)$ 、**立方阶** $O(n^3)$ 、**对数阶** $O(\log_2 n)$ 和**指数阶** $O(2^n)$ 等。

不同数量级对应的值存在如下关系:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) \quad (1.4)$$

其中 $O(\log n)$ 即为对数阶, 因为在实际计算中底数的值不重要, 可以忽略。

一个上机执行的程序除了需要存储空间来寄存本身所用的指令、常数、变量和输入数据外, 也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。当我们对算法进行存储空间分析时, 只考察局部变量所占空间。类似于时间复杂度, 算法的临时空间即**空间复杂度**通常也作为问题规模 n 的函数, 并以数量级的形式给出, 记作 $S(n) = O(g(n))$ 。

若算法所需临时空间相对于输入数据量而言是常数, 则称此算法为**原地工作**或**就地工作**。若所需临时空间依赖于特定的输入, 则通常按最坏情况来考虑。

例题 1.2 某算法代码如下:

```
1  int max(int a[],int n)
2  {
3      int i,maxi=0;
4      for(i=1;i<=n;i++)
5          if(a[i]>a[maxi]) maxi=i;
6      return a[maxi];
7  }
```

试分析该算法的空间复杂度。

解 由于函数体内分配的变量空间为临时空间, 不计形参占用的空间, 故仅计算变量 i 和 $maxi$ 的空间即可, 空间复杂度为 $O(1)$ 。

例题 1.3 某算法代码如下:

```
1  void MATMulti(int n,int a[][N],int b[][N],int c[][N])
2  {
3      int i,j,k;
4      for(i=0;i<n;i++)
5          for(j=0;j<n;j++)
6              {
7                  c[i][j]=0;
8                  for(k=1;k<=n;k++) c[i][j]+=a[i][k]*b[k][j];
9              }
10 }
```

试求该算法的时间复杂度。

解 由最深层循环内语句中原操作的执行频度可得时间复杂度为 $O(n^3)$ 。

例题 1.4 试求函数 $h(n) = n^{1.5} + 5000n \log_2 n$ 的时间复杂度。

解 注意到当 n 趋于无穷大时, 有

$$\sqrt{n} > \log_2 n \quad (1.5)$$

故 $h(n) = n^{1.5} + 5000n \log_2 n = O(n^{1.5})$ 。

例题 1.5 某算法代码如下:

```
1      x=2;
2      while(x<n/2) x*=2;
```

设 n 为描述问题规模的非负整数, 试求该算法的时间复杂度。

解 该算法中基本语句为 $x*=2$, 设其执行时间为 $T(n)$, 则有 $2^{T(n)} < n/2$, 即 $T(n) < \log_2 n = O(\log_2 n)$ 。

例题 1.6 某算法代码如下:

```
1      int fact(int n)
2      {
3          if(n<=1) return 1;
4          else return n*fact(n-1);
5      }
```

试求该算法的时间复杂度。

解 本算法为递归算法, 设其执行时间为 $T(n)$, 则有

$$\begin{cases} T(n) = 1, & n = 1 \\ T(n) = T(n-1) + 1, & n > 1 \end{cases} \quad (1.6)$$

故 $T(n) = T(n-1) + 1 = [T(n-2) + 1] + 1 = \cdots = T(1) + (n-1) = n$, 即时间复杂度为 $O(n)$ 。

第2章 线性表

2.1 线性表的类型定义

一个线性表是由 $n \geq 0$ 个相同类型的数据元素组成的有限序列，记作 $L = (a_1, a_2, \dots, a_n)$ ，其中 n 称为线性表的长度，当 $n = 0$ 时称此线性表为空表。

若将线性表写为 $L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ ，则称 a_1 为线性起点， a_n 为线性终点； i 称为下标，表示数据元素 a_i 在线性表中的位序；对于其中的任意一项 a_i ，称 a_{i-1} 为其直接前驱， a_{i+1} 为其直接后继。

线性表是最常用且最简单的一种数据结构，其逻辑特征有以下几点：

1. 若至少含一个元素，则只有唯一的起始元素和唯一的尾元素；
2. 除了起始元素外，其他元素有且只有一个前驱元素；
3. 除了尾元素外，其他元素有且只有一个后继元素；
4. 线性表中的每个元素都有唯一的逻辑序号；
5. 同一个线性表中可以存在多个值相同的元素，但它们的序号互不相同。

线性表的 ADT 定义如下。

```
1  ADT List
2  {
3      数据对象:D={a[i] | a[i]属于集合ElemSet, 0<=i&&i<=n}
4      数据关系:R1={<a[i-1], a[i]> | a[i-1]与a[i]均属于D, 2<=i&&i<=n}
5      基本操作:
6          InitList(&L)
7              操作结果:构造一个空线性表L.
8          DestroyList(&L)
9              初始条件:线性表L已存在.
10             操作结果:销毁线性表L.
11         ListLength(L)
12             初始条件:线性表L已存在.
13             操作结果:返回L中数据元素个数.
14         GetElem(L, i, &e)
15             初始条件:线性表L已存在, i>=1且i<=ListLength(L).
16             操作结果:用e返回L中第i个数据元素的值.
17         LocateElem(L, e, compare())
18             初始条件:线性表L已存在, compare()是数据元素判定函数.
19             操作结果:返回L中第i个与e满足关系compare()的数据元素的位序, 若不存在则返回0.
20         ListInsert(&L, i, e)
21             初始条件:线性表L已存在, i>=1且i<=ListLength(L)+1.
22             操作结果:在L中第i个位置之前插入新的数据元素e, L的长度加1.
23         ListDelete(&L, i, &e)
24             初始条件:线性表L已存在且非空, i>=1且i<=ListLength(L).
25             操作结果:删除L的第i个数据元素, 并用e返回其值, L的长度减1.
26         DispList(L)
27             初始条件:线性表L已存在.
28             操作结果:按前后次序输出线性表L的所有元素值.
29         //...
30     }ADT List
```

例题 2.1 用两个线性表 LA 和 LB 表示两个集合 A 和 B ，现要求一个新集合 $A = A \cup B$ ，试写出该过程的算法。

解 扩大 LA ，遍历 LB ，将存在于 LB 但不存在于 LA 的数据元素插入 LA 即可。

```

1  void union(List &La,List Lb)
2  {
3      int La_len=ListLength(La),Lb_len=ListLength(Lb);
4      for(int i=1;i<=Lb_len;i++)
5      {
6          GetElem(Lb,i,e);
7          if(!LocateElem(La,e,equal)) ListInsert(La,++La_len,e);
8      }
9  }

```

例题 2.2 已知线性表 LA 和 LB 中的数据元素按值非递减有序排列，现要求将 LA 和 LB 归并为一个新的线性表 LC ，且 LC 中的数据元素仍按值非递减有序排列，试写出该过程的算法。

解 依次扫描 LA 和 LB 的元素，比较当前元素的值，将较小值的元素赋给 LC ，如此循环，直到其中一个线性表扫描完毕，再将仍有元素的线性表的余下部分直接赋给 LC 即可。

```

1  void MergeList(List La,List Lb,List &Lc)
2  {
3      InitList(Lc);
4      int i=j=1,k=0;
5      int La_len=ListLength(La),Lb_len=ListLength(Lb);
6      while(i<=La_len&&j<=Lb_len)
7      {
8          GetElem(La,i,a[i]);GetElem(Lb,j,b[j]);
9          if(a[i]<=b[j]) {ListInsert(Lc,++k,a[i]);i++;}
10         else {ListInsert(Lc,++k,b[j]);j++;}
11     }
12     while(i<=La_len)
13     {
14         GetElem(La,i++,a[i]);
15         ListInsert(Lc,++k,a[i]);
16     }
17     while(j<=Lb_len)
18     {
19         GetElem(Lb,j++,b[j]);
20         ListInsert(Lc,++k,b[j]);
21     }
22 }

```

2.2 线性表的顺序表示和实现

若我们将线性表中的元素相继存放在一个连续的存储空间中，就构成了**顺序表**。顺序表是线性表的顺序存储方式，可利用一维数组描述存储结构，其中元素的逻辑顺序与物理顺序一致，可按下标直接存取。

设线性表的每个元素需占用 l 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置，则线性表中第 $i+1$ 个数据元素的存储位置 $LOC(a_{i+1})$ 和第 i 个数据元素的存储位置 $LOC(a_i)$ 之间满足下列关系：

$$LOC(a_{i+1}) = LOC(a_i) + l \quad (2.1)$$

一般地, 线性表的第 i 个数据元素 a_i 的存储位置为

$$LOC(a_i) = LOC(a_1) + (i - 1) \cdot l \quad (2.2)$$

其中 $LOC(a_1)$ 为线性表的起始位置或基地址。

我们可以通过动态分配和静态分配两种方式来构建线性表, 代码如下:

```

1 //动态分配顺序存储结构
2 #define LIST_INIT_SIZE 100//线性表存储空间的初始分配量
3 #define LIST_INCREMENT 10//线性表存储空间的分配增量
4 typedef struct
5 {
6     ElemType *elem;//存储空间基地址
7     int length;//当前长度
8     int listsize;//当前分配的存储容量,以sizeof(ElemType)为单位
9 }SqList;
10 //静态分配顺序存储结构
11 #define MaxSize 100
12 typedef struct
13 {
14     ElemType data[MaxSize];//存放顺序表的元素
15     int length;//顺序表的实际长度
16 }SqList;//顺序表类型

```

由于顺序表采用数组存放元素, 而数组具有随机存取特性, 故顺序表也具有随机存取特性。

在顺序表中有以下几种基本操作。

1. 初始化线性表算法: 将顺序表 L 的长度置为 0。

```

1 Status Init_Sq(SqList &L)
2 {
3     L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
4     if(!L.elem) exit(OVERFLOW);//申请空间失败则报错
5     L.length=0;
6     L.listsize=LIST_INIT_SIZE;
7     return 1;
8 }

```

2. 销毁线性表算法: 由于顺序表 L 的内存空间是经动态分配得到的, 不需要时应主动释放其空间。

```

1 Status DestroyList(SqList &L)
2 {
3     free(L.elem);
4     L.elem=NULL;
5     L.length=0;
6     L.listsize=0;//释放空间后对相关参数进行修改的语句十分重要,不能忽略
7     return 1;
8 }

```

3. 求线性表长度算法: 返回顺序表 L 的长度, 时间复杂度为 $O(1)$ 。

```

1 int GetLength(SqList L) {return L.length;}

```

4. 求线性表中第 i 个元素算法：位序 i 无效时返回 0，否则返回 1，并用引用型形参 e 返回第 i 个元素的值。

```

1  Status GetElem(SqList L,int i,ElemType &e)
2  {
3      if(i<1||i>L.length) return 0;
4      e=*(L.elem+i-1);
5      return 1;
6  }

```

5. 按值查找算法：在顺序表 L 中找第一个值为 e 的元素，找到后返回其位序，否则返回 0。

```

1  int LocateElem(SqList L,ElemType e)
2  {
3      ElemType *p;
4      int i=1;
5      p=L.elem;
6      while(i<=L.length&&(*p++!=e)) i++;
7      if(i<=L.length) return i;
8      else return 0;
9  }

```

6. 插入算法：将新元素 e 插入到顺序表 L 中位序为 i 的位置，当 i 无效时返回 0，否则将从 $L.elem[i-1]$ 到 $L.elem[L.length-1]$ 之间的所有元素全部后移一个位置，在 $L.elem[i-1]$ 处插入 e ，顺序表长度加 1，并返回 1。

```

1  Status ListInsert_Sq(SqList &L,int i,ElemType e)
2  {
3      ElemType *p;
4      if(i<1||i>L.length+1) return 0;//i值不合法
5      if(L.length>=L.listsize)//当前存储空间已满,增加容量
6      {
7          ElemType *newbase=
8              (ElemType*)realloc(L.elem,(L.listsize+LIST_INCREMENT)*sizeof(ElemType));
9          if(!newbase) exit(OVERFLOW);//存储分配失败
10         L.elem=newbase;//新基地址
11         L.listsize+=LIST_INCREMENT;//增加存储容量
12     }
13     ElemType *q=&(L.elem[i-1]);//q为插入位置
14     for(p=&(L.elem[L.length-1]);p>=q;p--) *(p+1)=*p;//插入位置及之后的元素后移
15     *q=e;//插入e
16     L.length++;
17     return 1;
18 }

```

注 该算法最好的情况为当 $i = n + 1$ 时移动次数为 0，最坏的情况为当 $i = 1$ 时移动次数为 n 。从平均情况出发，若在位置 i 插入新元素 e ，则需要将从 a_i 到 a_n 之间的所有元素全部往后移一个位置，此时移动次数为 $n - i + 1$ 。

设在等概率 $p_i = \frac{1}{n+1}$ 下移动的平均次数为

$$\sum_{i=1}^{n+1} p_i(n-i+1) = \sum_{i=1}^{n+1} \frac{1}{n+1}(n-i+1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} \cdot \frac{n(n+1)}{2} = \frac{n}{2} \quad (2.3)$$

可以看出, 插入算法 `ListInsert_Sq(L,i,e)` 的平均时间复杂度为 $O(n)$ 。

7. 删除算法: 删除 L 中位序为 i 的元素, 当 i 无效时返回 0, 否则将从 $L.elem[i]$ 到 $L.elem[length-1]$ 之间的所有元素全部前移一个位置, 顺序表长度减 1, 并返回 1。

```

1  Status ListDelete_Sq(SqList &L,int i,ElemType &e)
2  {
3      ElemType *p,*q;
4      if(i<1||i>L.length) return 0;//i值不合法
5      p=&(L.elem[i-1]); //p为被删除元素的位置
6      e=*p; //被删除元素的值赋给e
7      q=L.elem+L.length-1; //表尾元素的位置
8      for(++p;p<=q;p++) *(p-1)=*p; //被删除元素之后的元素前移
9      L.length--;
10     return 1;
11 }

```

注 该算法最好的情况为当 $i = n$ 时移动次数为 0, 最坏的情况为当 $i = 1$ 时移动次数为 $n - 1$ 。从平均情况出发, 若删除位置为 i 的元素 a_i , 则需要将从 a_{i+1} 到 a_n 之间的所有元素全部往前移一个位置, 此时移动次数为 $n - i$ 。

设在等概率 $p_i = \frac{1}{n}$ 下移动的平均次数为

$$\sum_{i=1}^n p_i(n-i) = \sum_{i=1}^n \frac{1}{n}(n-i) = \frac{1}{n} \sum_{i=1}^n (n-i) = \frac{1}{n} \cdot \frac{n(n-1)}{2} = \frac{n-1}{2} \quad (2.4)$$

可以看出, 删除算法 `ListDelete_Sq(L,i,e)` 的平均时间复杂度为 $O(n)$ 。

例题 2.3 已知线性表 (a_1, a_2, \dots, a_n) 采用顺序表 L 存储, 且每个元素都是互不相等的整数。试设计一个将所有奇数移到所有偶数前面的算法, 要求用时最短且辅助空间最少。

解 令 $i = 1, j = n - 1$, 在顺序表 L 中从左向右找到偶数 $L.elem[i]$, 从右向左找到奇数 $L.elem[j]$, 并交换两者, 循环此过程直到 i 与 j 相等为止。

```

1  void Move(SqList &L)
2  {
3      int i=0,j=L.length-1;
4      while(i<j)
5      {
6          while(L.elem[i]%2==1) i++;
7          while(L.elem[j]%2==0) j--;
8          if(i<j) swap(L.elem[i],L.elem[j]);
9      }
10 }

```

例题 2.4 已知一个整数线性表采用顺序表 L 存储, 试设计一个算法以删除其中所有值为负数的元素。

解 可以采用“整体重建顺序表”的思路来设计算法。

```

1  void DeleteMinus(SqList &L)
2  {
3      int k=0;
4      for(int i=0;i<L.length;i++) {if(L.elem[i]>=0) L.elem[k++]=L.elem[i];}
5      L.length=k;
6  }

```

例题 2.5 设将 $n > 1$ 个整数存放在一维数组 R 中, 试设计一个算法将 R 中的整数序列循环左移 $0 < p < n$ 个位置, 即将 R 中的数据序列 $(X_0, X_1, \dots, X_{n-1})$ 变换为 $(X_p, X_{p+1}, \dots, X_{n-1}, X_0, X_1, \dots, X_{p-1})$, 并分析其时间复杂度和空间复杂度。

解 设 $R = (X_0, X_1, \dots, X_{n-1})$, 记 $A = (X_0, X_1, \dots, X_{p-1})$, $B = (X_p, X_{p+1}, \dots, X_{n-1})$, 可以设计一个逆置算法 `reverse()` 用于原地逆置数组, 则 A 逆置后为 $A' = (X_{p-1}, \dots, X_1, X_0)$, B 逆置后为 $B' = (X_{n-1}, \dots, X_{p+1}, X_p)$, 此时 $A'B' = (X_{p-1}, \dots, X_1, X_0, X_{n-1}, \dots, X_{p+1}, X_p)$, 将其逆置即可得 $(X_p, X_{p+1}, \dots, X_{n-1}, X_0, X_1, \dots, X_{p-1})$ 。

算法代码如下:

```

1  void reverse(int R[],int m,int n)//逆置数组
2  {
3      int tmp;
4      for(int i=0;i<(n-m+1)/2;i++) tmp=R[m+i],R[m+i]=R[n-i],R[n-i]=tmp;//将R[m+i]与R[n-i]交换
5  }
6  int ListReverse(int R[],int n,int p)
7  {
8      if(p<=0||p>=n) return 0;
9      else
10     {
11         reverse(R,0,p-1);
12         reverse(R,p,n-1);
13         reverse(R,0,n-1);
14         return 1;
15     }
16 }
```

不难看出算法 `reverse(R,m,n)` 的时间复杂度为 $O(n-m)$, 故算法 `ListReverse(R,n,p)` 的时间复杂度为 $O(p) + O(n-p) + O(n) = O(n)$; 由于算法 `ListReverse(R,n,p)` 只定义了少数变量, 故空间复杂度为 $O(1)$ 。

2.3 线性表的链式表示和实现

除了顺序表, 我们还可以用**单链表**方式来存储线性表, 即用一个指针来表示结点间的逻辑关系。单链表的一个存储结点分为**数据域**(data)和**指针域**(next), 后者又称为**链域**。其中, 数据域用于存储线性表的一个数据元素, 即在单链表中一个结点存放一个数据元素; 而指针域用于存储一个指针, 该指针指向后继元素对应的结点, 即单链表中结点的指针用于表示后继关系。

单链表具有以下特点:

1. 每个数据元素由结点构成, 且按线性结构成链存储;
2. 结点可以连续存储, 也可以不连续存储;
3. 结点的物理顺序与逻辑顺序可以不一致。

在单链表中, 第一个元素结点称为**首结点**, 而最后一个元素结点称为**尾结点**, 其指针域为空, 可方便后续扩充。此外, 单链表还可分为**带头结点**和**不带头结点**两种类型, 在许多情况下前者可以简化运算的实现过程。

设数据元素的类型为 `ElemType`, 则单链表的结点类型声明如下:

```

1  typedef struct node
2  {
3      ElemType data;
4      struct node *next;
5  }LNode,*LinkList;
```


在单链表中有以下几种基本操作（省略部分已学算法）。

1. 销毁单链表算法：利用 `free` 函数释放所有结点的空间。

```

1  Status DestroyList(LinkList &L)
2  {
3      LinkList p;
4      while(L)
5      {
6          p=L->next;
7          free(L);
8          L=p;
9      }
10     return 1;
11 }
```

2. 在第 i 个结点位置插入数据值为 e 的算法：先在单链表 L 中寻找第 $i-1$ 个结点，若未找到则返回 0，否则由 p 指向该结点，并创建一个以 e 为数据值的新结点 s ，将其插入到 p 所指结点之后。

```

1  Status ListInsert_L(LinkList &L,int i,ElemType e)
2  {
3      LinkList p,s;
4      p=L;
5      int j=0;
6      while(p&&j<i-1) p=p->next,j++;//寻找第i-1个结点
7      if(!p||j>i-1) return 0;
8      s=(LinkList)malloc(sizeof(LNode));//生成新结点
9      s->data=e;
10     s->next=p->next;//插入L中
11     p->next=s;
12     return 1;
13 }
```

3. 删除第 i 个结点算法：先在单链表 L 中寻找第 $i-1$ 个结点，若未找到则返回 0，否则由 p 指向该结点，再让 q 指向后继结点；若 q 所指结点为空则返回 0，否则删除 q 结点并释放其占用的空间。

```

1  Status ListDelete_L(LinkList &L,int i,ElemType &e)
2  {
3      LinkList p,q;
4      p=L;
5      int j=0;
6      while(p->next&&j<i-1) p=p->next,j++;//寻找第i个结点，并令p指向其前驱
7      if(!(p->next)||j>i-1) return 0;//删除位置不合理
8      q=p->next;
9      p->next=q->next;
10     e=q->data;//在L中删除第i个元素，并由e返回其数据值
11     free(q);//删除并释放结点
12     return 1;
13 }
```

例题 2.6 试设计一个算法将一个至少含两个数据结点单链表 L 中所有的结点逆置，并分析其时间复杂度。

解 先将单链表 L 拆为两部分，一部分是只含头结点 L 的空表，另一部分是由 p 指向首结点的单链表，然后利用头插法将单链表中的结点依次插入 L 中，这样就可以达到逆置的目的。

```

1  void ListReverse(LNode *&L)
2  {
3      LNode *p=L->next,*q;
4      L->next=NULL;
5      while(p!=NULL)
6      {
7          q=p->next;
8          p->next=L->next;
9          L->next=p;
10         p=q;
11     }
12 }

```

例题 2.7 已知一个带头结点的单链表 L ，试设计一个算法，要求在不改变链表的情况下查找倒数第 $k \in \mathbb{N}_+$ 个位置上的结点，若查找成功，则输出该结点数据域的值并返回 1，否则只返回 0。

解 定义两个指针 p 和 q ，初始时均指向首结点，并令 p 指针沿链表移动。当 p 指针移动到第 k 个结点时， q 指针与 p 指针同步移动；当 p 指针移动到尾结点时， q 指针所指结点即为倒数第 k 个结点。

```

1  int Searchk(LinkList L,int k)
2  {
3      LinkList p,q;
4      int cnt=0;
5      p=q=L->next;
6      while(p!=NULL&&cnt<k) cnt++,p=p->next;
7      if(p==NULL) return 0;
8      else
9      {
10         while(p!=NULL) q=q->next,p=p->next;
11         printf("%d",q->data);
12         return 1;
13     }
14 }

```

2.4 循环链表

循环单链表是单链表的变形，其中尾结点的 `next` 指针指向的是单链表的前端，而为简化操作，我们常在循环单链表中加入头结点。

循环单链表具有以下特点：

1. 判空条件为 `L->next==L`;
2. 只要知道表中某一结点的地址，便可搜寻到其他所有结点的地址；
3. 没有一个结点的指针域为空。

例题 2.8 有 n 个人围成一圈，并给他们从 1 开始编号。现规定从编号为 1 的人开始报数，数到第 m 个人出列，然后从出列的下一个人重新开始报数，再数到第 m 个人出列……如此循环反复，直到所有人都出列，试编写程序求出该过程的出列序列。

解 此问题称为约瑟夫问题，在这里我们可以利用不带头结点的循环单链表来解决。

```

1     typedef struct node//定义结点类型
2     {
3         int num;
4         struct node *next;
5     }Child;
6     void CreateList(Child *&L,int n)//建立有n个结点的循环单链表
7     {
8         Child *p,*tc;//tc指向新建循环单链表的尾结点
9         L=(Child*)malloc(sizeof(Child));
10        L->num=1;
11        tc=L;
12        for(int i=2;i<=n;i++)
13        {
14            p=(Child*)malloc(sizeof(Child));
15            p->num=i;
16            tc->next=p;
17            tc=p;
18        }
19        tc->next=L;
20    }
21    void Joseph(int n,int m)
22    {
23        Child *L,*p,*q;
24        CreateList(L,n);
25        for(int i=1;i<=n;i++)
26        {
27            int j=1;
28            p=L;
29            while(j<m-1) j++,p=p->next;
30            q=p->next;
31            printf("%d ",q->num);
32            p->next=q->next;
33            free(q);
34            L=p->next;
35        }
36    }

```

2.5 双链表与循环双链表

双向链表是指在前驱及后继方向都能遍历的线性链表，简称**双链表**。

与单链表相比，双链表结点除了有 **data** 域和 **next** 域外，还有一个 **prior** 域，其中 **next** 域存放指向后继的指针，**prior** 域存放指向前驱的指针，即双链表结点有一个数据域与两个指针域。

从操作角度而言，双链表克服了单链表只能单向遍历的缺点，使用更加方便；但从空间角度而言，由于多了 **prior** 域，在结点中 **data** 域占比减少，使得空间利用率下降。

在双链表中， $p == p \rightarrow next \rightarrow prior$ 与 $p == p \rightarrow prior \rightarrow next$ 恒成立。

设数据元素的类型为 `ElemType`，则双链表的结点类型声明如下：

```
1  typedef struct DuLNode
2  {
3      ElemType data;
4      DuLNode *prior,*next;
5  }DuLNode,DuLinkList;
```

同理地，我们可以将双链表分为非循环双链表与双向循环链表，后者简称为循环双链表。在一般情况下，我们描述的循环双链表均指带头结点的循环双链表。

有关双链表的基本操作和单链表类似，在此略去不表。

2.6 综合：一元多项式的表示及相加

在本节中，我们将利用线性表的相关知识来解决一元多项式的表示及相加问题。

众所周知，一个一元多项式 $P_n(x)$ 可按升幂写成

$$P_n(x) = p_0 + p_1x + p_2x^2 + \cdots + p_nx^n \quad (2.5)$$

其由 $n+1$ 个系数唯一确定，故我们可以在计算机中用一个线性表来表示它：

$$P = (p_0, p_1, p_2, \cdots, p_n) \quad (2.6)$$

此时再设有线性表表示为 $Q = (q_0, q_1, q_2, \cdots, q_m)$ 的一元多项式 $Q_m(x)$ ，其中 $m < n$ ，则 $P_n(x)$ 与 $Q_m(x)$ 相加的结果 $R_n(x) = P_n(x) + Q_m(x)$ 用线性表可表示为

$$R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \cdots, p_m + q_m, p_{m+1}, \cdots, p_n) \quad (2.7)$$

然而，在处理形如 $S(x) = 1 + 3x^{10000} + 2x^{20000}$ 这样的指数较高且各项指数间隔较大的一元多项式时，再用 (2.6) 式表达 $S(x)$ 就显然不合适了。我们可以考虑用另一种方式来表示一元多项式：

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \cdots + p_mx^{e_m} \quad (2.8)$$

其中 p_i 是指数为 e_i 的项的非零系数，且有 $0 \leq e_1 < e_2 < \cdots < e_m = n$ ，此时便可以用线性表

$$((p_1, e_1), (p_2, e_2), \cdots, (p_m, e_m)) \quad (2.9)$$

来唯一确定多项式 $P_n(x)$ 。通常情况下我们采用链式结构来存储一元多项式并进行相关操作。

两个一元多项式相乘的算法可以利用两个一元多项式相加的算法来实现。设 $A(x)$ 和 $B(x)$ 为满足 (2.8) 式的多项式，则

$$\begin{aligned} M(x) &= A(x) \times B(x) \\ &= A(x) \times [b_1x^{e_1} + b_2x^{e_2} + \cdots + b_nx^{e_n}] \\ &= \sum_{i=1}^n b_i A(x) x^{e_i} \end{aligned} \quad (2.10)$$

第3章 栈和队列

3.1 顺序栈

栈是一种特殊的线性表，其插入和删除运算仅限定在线性表的某一段进行，不能在表中间和另一端进行。栈的插入操作称为**进栈**（push），删除操作称为**出栈**（pop），且定义允许进行插入和删除的一端为**栈顶**，另一端为**栈底**。处于栈顶位置的数据元素称为**栈顶元素**，不含任何数据元素的栈称为**空栈**。

设有栈 $S = (a_1, a_2, \dots, a_n)$ ，根据定义，元素的进栈顺序为 a_1, a_2, \dots, a_n ，而出栈顺序为 a_n, \dots, a_2, a_1 ，故栈称为**后进先出**（last in first out）线性表，简称为**LIFO**结构。

栈的ADT定义如下：

```
1  ADT Stack
2  {
3      数据对象:D={a[i] | a[i]属于集合ElemSet, 0<=i&&i<=n}
4      数据关系:R1={<a[i-1], a[i]> | a[i-1]与a[i]均属于D, 2<=i&&i<=n}
5          约定a[n]端为栈顶, a[1]端为栈底。
6      基本操作:
7          InitStack(&S)
8              操作结果:构造一个空栈S.
9          DestroyStack(&S)
10             初始条件:栈S已存在.
11             操作结果:栈S被销毁.
12          ClearStack(&S)
13             初始条件:栈S已存在.
14             操作结果:将S清为空栈.
15          StackEmpty(S)
16             初始条件:栈S已存在.
17             操作结果:若栈S为空栈,则返回TRUE,否则返回FALSE.
18          StackLength(S)
19             初始条件:栈S已存在.
20             操作结果:返回S的元素个数,即栈的长度.
21          GetTop(S, &e)
22             初始条件:栈S已存在且非空.
23             操作结果:用e返回S的栈顶元素.
24          Push(&S, e)
25             初始条件:栈S已存在.
26             操作结果:插入元素e为新的栈顶元素.
27          Pop(&S, &e)
28             初始条件:栈S已存在且非空.
29             操作结果:删除S的栈顶元素,并用e返回其值.
30          StackTraverse(S, visit())
31             初始条件:栈S已存在且非空.
32             操作结果:从栈底到栈顶依次对S的每个数据元素调用函数visit(),一旦visit()失败则操作失效.
33  }
```

栈的顺序存储结构称为**顺序栈**，其通常由一个一维数组 data 和一个记录栈顶元素位置的变量 top 组成，并将栈底置于数组下标较小的一端。

顺序栈的类型声明如下：

```

1  #define STACK_INIT_SIZE 100//存储空间初始分配量
2  #define STACK_INCREMENT 10//存储空间分配增量
3  typedef struct
4  {
5      ElemType *base;//栈底指针,在对栈进行操作时固定不变
6      ElemType *top;//栈顶指针
7      int stacksize;//栈空间大小
8  }SqStack;

```

其中我们规定 `top` 指针指向数组中栈顶的下一个存储位置，并用 `top==base` 表示栈空。

在顺序栈中有以下几种基本操作。

1. 初始化栈算法：通过动态申请得到所需的存储空间，并初始化相关参数。

```

1  void InitStack(SqStack &S)
2  {
3      if(!(S.base=(ElemType*)malloc(STACK_INIT_SIZE*sizeof(ElemType)))) exit(OVERFLOW);
4      S.top=S.base;
5      S.stacksize=STACK_INIT_SIZE;
6  }

```

2. 销毁栈算法：由于顺序栈的内存空间是经动态申请得到的，在不需要时应主动释放其空间。

```

1  void DestroyStack(SqStack &S)
2  {
3      free(S.base);
4      S.top=S.base=NULL,S.stacksize=0;
5  }

```

3. 进栈算法：主要判断相关操作是否会导致数据溢出。

```

1  void Push(SqStack &S,ElemType e)
2  {
3      if(S.top-S.base>=S.stacksize)//栈满,扩大存储空间
4      {
5          S.base=(ElemType*)realloc(S.base,(S.stacksize+STACK_INCREMENT)*sizeof(ElemType));
6          if(!S.base) exit(OVERFLOW);
7          S.top=S.base+S.stacksize,S.stacksize+=STACK_INCREMENT;
8      }
9      *(S.top)++=e;
10 }

```

4. 出栈算法：主要判断是否在空栈的情况下进行出栈操作。

```

1  Status Pop(SqStack &S,ElemType &e)
2  {
3      if(S.top==S.base) return 0;//若为空栈则直接返回0
4      e=*--S.top;
5      return 1;
6  }

```


5. 取栈顶元素算法：通过 `top` 指针得到栈顶元素，注意栈长没有变化。

```

1   Status GetTop(SqStack &S, ElemType &e)
2   {
3       if(S.top>S.base) {e=*(S.top-1);return 1;}
4       else return 0;
5   }

```

6. 判断栈空算法：若栈为空则返回 `TRUE`，否则返回 `FALSE`。

```

1   Status StackEmpty(SqStack S)
2   {
3       if(S.top==S.base) return TRUE;
4       else return FALSE;
5   }

```

3.2 链栈

栈的链式存储结构称为**链栈**，通常用不带头结点的单链表实现，其中规定尾结点为栈底，首结点为栈顶，且当 `top==NULL` 时为链栈为空。与顺序栈有所不同，链栈不考虑栈满情况，即在一般情况下我们可以认为链栈的栈长为无穷大。

设数据元素的类型为 `ElemType`，则链栈的结点类型声明如下：

```

1   typedef struct node
2   {
3       ElemType data;
4       struct node *next;
5   }LinkStack;

```

在链栈中有以下几种基本操作。

1. 初始化栈算法：将栈置空。

```

1   void InitStack(LinkStack *&top) {top=NULL;}

```

2. 销毁栈算法：利用 `free` 函数释放所有结点的空间。

```

1   void DestroyStack(LinkStack *&top)
2   {
3       LinkStack *pre=top,*p;
4       if(pre==NULL) return;//若为空栈则不进行任何操作
5       p=pre->next;
6       while(p!=NULL)
7       {
8           free(pre);//释放pre指针所指的结点
9           pre=p;
10          p=p->next;//pre与p同步后移
11      }
12      free(pre);//释放尾结点
13  }

```

3. 进栈算法：创建一个数据域值为 x 的新结点，并将该结点插入到 top 结点之前作为栈顶结点。

```

1  int Push(LinkStack *&top, ElemType x)
2  {
3      LinkStack *p;
4      p=(LinkStack*)malloc(sizeof(LinkStack)); //创建结点p用于存放x
5      p->data=x;
6      p->next=top; //插入结点p作为栈顶顶点
7      top=p;
8      return 1;
9  }

```

4. 出栈算法：将栈顶结点的数据域值赋给 x 后删除结点。

```

1  int Pop(LinkStack *&top, ElemType &x)
2  {
3      LinkStack *p;
4      if(top==NULL) return 0; //若为空栈则直接返回0
5      else
6      {
7          p=top; //p指向栈顶结点
8          x=p->data; //取栈顶元素x
9          top=p->next; //删除结点p
10         free(p); //释放结点p
11         return 1;
12     }
13 }

```

5. 取栈顶元素算法：将栈顶结点的数据域值赋给 x 。

```

1  int GetTop(LinkStack *top, ElemType &x)
2  {
3      if(top==NULL) return 0;
4      else {x=top->data; return 1;}
5  }

```

6. 判断栈空算法：若栈为空则返回 1，否则返回 0。

```

1  int StackEmpty(LinkStack *top)
2  {
3      if(top==NULL) return 1;
4      else return 0;
5  }

```

3.3 顺序队列与循环队列

与栈类似，队列也是一种运算受限的线性表，不同的是，其插入限定在表的某一端进行，删除限定在表的另一端进行，是一种先进先出（first in first out, **FIFO**）的结构。

队列的插入操作称为入队，并称允许插入的一端称为队尾；删除操作称为出队，并称允许删除的一端为队头。设有队列 $q = (a_1, a_2, \dots, a_n)$ ，则称 a_1 为队头元素， a_n 为队尾元素。

队列的 ADT 定义如下：

```

1  ADT Queue
2  {
3      数据对象:D={a[i] | a[i] 属于集合ElemSet, 0<=i&&i<=n}
4      数据关系:R1={<a[i-1], a[i]> | a[i-1] 与 a[i] 均属于D, 2<=i&&i<=n}
5          约定a[1]端为队头,a[n]端为队尾.
6      基本操作:
7          InitQueue(&Q)
8              操作结果:构造一个空队列.
9          DestroyQueue(&Q)
10             初始条件:队列Q已存在.
11             操作结果:队列Q被销毁.
12          ClearQueue(&Q)
13             初始条件:队列Q已存在.
14             操作结果:将Q清为空队列.
15          QueueEmpty(Q)
16             初始条件:队列Q已存在.
17             操作结果:若Q为空队列,则返回TRUE,否则返回0.
18          QueueLength(Q)
19             初始条件:队列Q已存在.
20             操作结果:返回Q的元素个数,即队列的长度.
21          GetHead(Q,&e)
22             初始条件:Q为非空队列.
23             操作结果:用e返回Q的队头元素.
24          EnQueue(&Q,e)
25             初始条件:队列Q已存在.
26             操作结果:插入元素e为新的队尾元素.
27          DeQueue(&Q,&e)
28             初始条件:Q为非空队列.
29             操作结果:删除Q的队头元素,并用e返回其值.
30          QueueTraverse(Q,visit())
31             初始条件:队列Q已存在且非空.
32             操作结果:从队头到队尾依次对Q的每个数据元素调用函数visit(),一旦visit()失败则操作失效.
33  }ADT Queue

```

队列的顺序存储结构简称为**顺序队列**，其由一个存储队列元素的一维数组和两个分别指示队头和队尾位置的队头指针 `front` 和队尾指针 `rear` 组成，并规定当 `front==rear` 时队列为空。通常，我们在初始化队列时，令 `front=rear=0`；在非空队列中，`front` 始终指向队头元素，而 `rear` 指向队尾元素的下一个存储位置。

顺序队列的类型声明如下：

```

1  #define MaxSize 20//指定队列的容量
2  typedef struct
3  {
4      ElemType data[MaxSize]; //存储队列元素
5      int front, rear; //队头和队尾指针
6  }SqQueue;

```

若我们将顺序队列中队满的条件设置为 `rear==MaxSize`，会导致所谓“假溢出”的现象。为解决这个问题，我们可以把存储队列元素的表从逻辑上看成一个环，这种环形的表称为**循环队列**或**环形队列**。

采用循环队列后，有

```
1   front=(front+1)%MaxSize;//队头指针加1
2   rear=(rear+1)%MaxSize;//队尾指针加1
3   front==rear;//队空条件
4   front==(rear+1)%MaxSize;//队满条件
```

由队满条件可看出其最多只能容纳 $\text{MaxSize}-1$ 个元素，且队长为 $(\text{rear}-\text{front}+\text{MaxSize})\% \text{MaxSize}$ 。

循环队列的类型声明如下：

```
1   #define MAXQSIZE 100
2   typedef struct
3   {
4       ElemType *base;
5       int front,rear;
6   }SqQueue;
```

在循环队列中有以下几种基本操作。

1. 初始化队列算法：利用队空条件初始化队列。

```
1   void InitQueue(SqQueue &Q)
2   {
3       Q.base=(QElemType*)malloc(MAXQSIZE*sizeof(QElemType));
4       if(!Q.base) exit(OVERFLOW);//存储分配失败
5       Q.front=Q.rear=0;
6   }
```

2. 销毁队列算法：由于循环队列的内存空间是经动态申请得到的，在不需要时应主动释放其空间。

```
1   void DestroyQueue(SqQueue &Q)
2   {
3       if(Q.base) free(Q.base);
4       Q.base=NULL,Q.front=Q.rear=0;
5   }
```

3. 入队算法：判断是否队满，若未满足则在 rear 位置存放元素 e，并令 rear 循环加 1。

```
1   Status EnQueue(SqQueue &Q,QElemType e)
2   {
3       if((Q.rear+1)%MAXQSIZE==Q.front) return 0;//队列已满
4       Q.base[Q.rear]=e,Q.rear=(Q.rear+1)%MAXQSIZE;
5       return 1;
6   }
```

4. 出队算法：判断是否队空，若不空则将 front 位置的元素赋给 e，并令 front 循环加 1。

```
1   Status DeQueue(SqQueue &Q,QElemType &e)
2   {
3       if(Q.front==Q.rear) return 0;
4       e=Q.base[Q.front],Q.front=(Q.front+1)%MAXQSIZE;
5       return 1;
6   }
```

5. 取队头元素算法：判断是否队空，若不空则将 front 位置的元素赋给 e。

```

1  Status GetHead(SqQueue Q, QElemType &e)
2  {
3      if(Q.front==Q.rear) return 0;
4      e=Q.base[Q.front];
5      return 1;
6  }

```

6. 判断队空算法：若队空则返回 TRUE，否则返回 FALSE。

```

1  Status QueueEmpty(SqQueue Q)
2  {
3      if(Q.front==Q.rear) return TRUE;
4      else return FALSE;
5  }

```

例题 3.1 若用一个大小为 6 的数组实现循环队列，队头指针 front 指向队头元素的前一个位置，队尾指针 rear 指向队尾元素的位置。已知当前 rear 和 front 的值分别为 0 和 3，求在队列中删除一个元素并插入两个元素后 rear 和 front 的值。

解 由题意可知 rear=0, front=3, MaxSize=6, 则有

$$\begin{cases} \text{front}=(\text{front}+1)\% \text{MaxSize}=4 \\ \text{rear}=(\text{rear}+2)\% \text{MaxSize}=2 \end{cases}$$

3.4 链队列

队列的链式存储结构称为**链队列**，通常用带有队头指针 front 和队尾指针 rear 的单链表实现，其中 front 指向头结点，rear 指向队尾结点，且规定 Q.front->next==NULL 时为链队列为空。链队列不考虑队满情况。

设数据元素的类型为 ElemType，则链队列的结点类型声明如下：

```

1  typedef struct QNode//数据结点
2  {
3      ElemType data;//存放队中元素
4      struct QNode *next;//指向后继结点
5  }QNode,*QueuePtr;//数据结点类型
6  typedef struct//链队列结点
7  {
8      QNode *front;//队头指针
9      QNode *rear;//队尾指针
10 }LinkQueue;//链队列结点类型

```

在链队列中有以下几种基本操作。

1. 初始化队列算法：创建链队列头结点，其指针域为 NULL，并令 front 和 rear 指向头结点。

```

1  void InitQueue(LinkQueue &Q)
2  {
3      if(!(Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode)))) exit(OVERFLOW);
4      Q.front->next=NULL;
5  }

```

2. 销毁队列算法：依次释放链队列的结点。

```

1 void DestroyQueue(LinkQueue &Q)
2 {
3     while(Q.front) {Q.rear=Q.front->next;free(Q.front);Q.front=Q.rear;}
4 }

```

3. 入队算法：创建一个数据域值为 e 的新结点，将其插入到链队列的末端，并令 rear 指向该结点。

```

1 void EnQueue(LinkQueue &Q,QElemType e)
2 {
3     QueuePtr p;
4     if(!(p=(QueuePtr)malloc(sizeof(QNode)))) exit(OVERFLOW); //存储分配失败
5     p->data=e;
6     p->next=NULL;
7     Q.rear->next=p;
8     Q.rear=p;
9 }

```

4. 出队算法：主要考虑特殊情况。

```

1 Status DeQueue(LinkQueue &Q,QElemType &e)
2 {
3     QueuePtr p;
4     if(Q.front==Q.rear) return 0;
5     p=Q.front->next;
6     e=p->data;
7     Q.front->next=p->next;
8     if(Q.rear==p) Q.rear=Q.front; //重要,不能忽略
9     free(p);
10    return 1;
11 }

```

5. 取队头元素算法：将头结点的数据域值赋给 e。

```

1 Status GetHead(LinkQueue Q,QElemType &e)
2 {
3     QueuePtr p;
4     if(Q.front==Q.rear) return 0; //队空
5     p=Q.front->next;
6     e=p->data;
7     return 1;
8 }

```

6. 判断队空算法：若队空则返回 TRUE，否则返回 FALSE。

```

1 Status QueueEmpty(LinkQueue Q)
2 {
3     if(Q.front->next==NULL) return TRUE;
4     else return FALSE;
5 }

```


3.5 递归

在定义一个过程或函数时出现调用本过程或本函数的成分称为**递归**。若调用自身则称为**直接递归**；若过程或函数 p 调用过程或函数 q ，而 q 又调用 p ，则称为**间接递归**。如果一个递归过程或递归函数中递归调用语句是最后一条执行语句，则称这种递归调用为**尾递归**。一般情况下，当遇到定义递归（如阶乘、斐波那契数列等）、数据结构递归（如单链表结点类型的定义）和问题递归（如汉诺塔问题）时，通常采用递归的方法解决。

递归模型是递归算法的抽象，其反映了一个问题的递归结构，如下面所展示的阶乘算法的递归模型：

$$f(0) = 1 \quad (3.1)$$

$$f(n) = n \cdot f(n-1), \quad n > 0 \quad (3.2)$$

其中，(3.1) 式给出了递归的**终止条件**，(3.2) 式给出了 $f(n)$ 的值与 $f(n-1)$ 的值之间的关系。我们称 (3.1) 式为**递归出口**，(3.2) 式为**递归体**。一般地，一个递归模型是由递归出口和递归体两部分组成的，前者确定递归何时结束，后者确定递归求解时的递推关系。

主程序第一次调用递归函数称为**外部调用**，递归函数每次递归调用自身称为**内部调用**，二者返回调用其函数的地址不同，故需要用栈保存，这种栈称为**递归工作栈**。在递归工作栈中，每一次递归调用需要为函数中使用的参数、局部变量等另外分配存储空间，每个函数的工作空间互不干扰，回到上层还可恢复上层原先的值。每层递归调用需分配的空间形成**递归的工作记录**，即递归的工作记录包括局部变量、返回地址和参数，而后进入递归工作栈中。

我们在求递归模型时一般遵循以下步骤：

1. 对原问题进行分析，假设出合理的“较小问题”；
2. 假设“较小问题”是可解的，在此基础上确定“原问题”的解；
3. 确定一个特定情况的解，并将此作为递归出口。

例题 3.2 试编写一个递归算法，正向显示以 L 为首结点指针的单链表的所有结点值。

解 由题，可得递归算法如下：

```
1 void traverse(LNode *L)
2 {
3     if(L==NULL) return;
4     printf("%d ",L->data);traverse(L->next);
5 }
```

例题 3.3 试编写一个递归算法，反向显示以 L 为首结点指针的单链表的所有结点值。

解 由题，可得递归算法如下：

```
1 void traverseR(LNode *L)
2 {
3     if(L==NULL) return;
4     traverseR(L->next);printf("%d ",L->data);
5 }
```

例题 3.4 试编写一个递归算法，释放以 L 为首结点指针的单链表中的所有结点。

解 由题，可得递归算法如下：

```
1 void Destroy(LNode *L)
2 {
3     if(L==NULL) return;
4     Destroy(L->next);free(L);
5 }
```

例题 3.5 试编写一个递归算法，删除以 L 为首结点指针的单链表中值为 x 的第一个结点。

解 由题，可得递归算法如下：

```
1 void del(LNode *&L, ElemType x)
2 {
3     LNode *t;
4     if(L==NULL) return;
5     if(L->data==x)
6     {
7         t=L;
8         L=L->next;
9         free(t);
10        return;
11    }
12    else del(L->next, x);
13 }
```

例题 3.6 试编写一个递归算法，删除以 L 为首结点指针的单链表中值为 x 的所有结点。

解 由题，可得递归算法如下：

```
1 void delall(LNode *&L, ElemType x)
2 {
3     LNode *t;
4     if(L==NULL) return;
5     if(L->data==x)
6     {
7         t=L;
8         L=L->next;
9         free(t);
10        delall(L, x);
11    }
12    else delall(L->next, x);
13 }
```

第4章 数组

4.1 数组的定义与存储

数组是一种特殊的数据结构，具体表现如下。

1. 数组是一种存储结构，是语言内建的数据类型，其可以成为多种数据结构的存储表示，仅有按下标读与写两种操作。
2. 数组同时也是一种逻辑结构，用于问题的解决，具备查找、定位、插入、删除等操作。
3. 数据元素为不可再分割的单元元素的一维数组（向量）是线性结构，而数据元素为数组的多维数组是非线性结构。

数组是相同类型的数据元素的集合，其中，一维数组的每个数据元素都是一个序对，由下标和值组成，而多维数组的特点是每一个数据元素可以有多个直接前驱和多个直接后继，且数据元素的下标一般具有固定的下界和上界。在高级语言中，一维数组只能按元素的下标直接存取数组元素的值。

对于一维数组，我们设每个数据元素占据相等的 l 个存储单元，起始存储地址为 a ，则第 i 号数据元素的存储地址 $LOC(A[i])$ 为

$$LOC(A[i]) = \begin{cases} a, & i = 0 \\ LOC(A[i-1]) + l = a + il, & i > 0 \end{cases}$$

二维数组 $A[n][m]$ 可视为由 n 个行向量组成的向量，也可视为由 m 个列向量组成的向量，其类型可以定义为分量类型为一维数组类型的一维数组类型，声明如下：

```
1 typedef ElemType array2[n][m];
2 //等价于：
3 //typedef ElemType array1[m];
4 //typedef array1 array2[n];
```

一个 n 行 m 列的二维数组 A 可以看作是每个数据元素都是相同类型的一维数组的一维数组，即

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} = [A_1, A_2, \cdots, A_n]^T$$

其中有 $A_i = [a_{i1}, a_{i2}, \cdots, a_{in}]$, $1 \leq i \leq n$ 。

与之前线性表的构建方式相同，我们也可通过静态定义和动态存储分配两种方式建立一个二维数组。

```
1 //静态定义
2 #define n 30
3 int A[n][m], B[100][60];
4 //动态存储分配
5 int **A, n=10, m=6;
6 *A=(int**)malloc(n*sizeof(int*));
7 for(int i=0; i<n; i++) A[i]=(int*)malloc(m*sizeof(int));
```

若一个二维数组不再使用，则需要通过动态回收的方式销毁该数组：

```
1 for(int i=0; i<n; i++) free(A[i]);
2 free(A);
```

现给定一个二维数组 $A[n][m]$ ，我们有行优先存放与列优先存放两种方式对其进行存储。

1. 行优先存放：设数组起始存储地址为 a ，每个元素占用 l 个存储单元，则 $A[i][j]$ 的存储位置为

$$LOC(A[i][j]) = a + (i \cdot m + j) \cdot l \quad (4.1)$$

2. 列优先存放：设数组起始存储地址为 a ，每个元素占用 l 个存储单元，则 $A[i][j]$ 的存储位置为

$$LOC(A[i][j]) = a + (j \cdot n + i) \cdot l \quad (4.2)$$

同理，若起始存储位置为 a 的三维数组 $A[m_1][m_2][m_3]$ 按行优先存放的方式进行存储，则 $A[i][j][k]$ 的存储位置为

$$LOC(A[i][j][k]) = a + (i \cdot m_2 \cdot m_3 + j \cdot m_3 + k) \cdot l \quad (4.3)$$

联立 (4.1) 式与 (4.3) 式进行数学归纳，可推得起始存储位置为 a 的 n 维数组 $A[m_1][m_2] \cdots [m_n]$ 在按行优先存放的方式进行存储时， $A[i_1][i_2] \cdots [i_n]$ 的存储位置为

$$LOC(A[i_1][i_2] \cdots [i_n]) = a + \left(\sum_{j=1}^{n-1} i_j \cdot \prod_{k=j+1}^n m_k + i_n \right) \cdot l \quad (4.4)$$

4.2 矩阵的压缩存储

4.2.1 特殊矩阵

特殊矩阵是指非零元素或零元素的分布具有一定规律的矩阵。特殊矩阵的压缩存储主要是针对阶数较高的特殊矩阵，对于如零元素和对称元素等具有规律的元素不进行存储，以便节省空间。

本小节主要讨论对称矩阵和三对角矩阵两种特殊矩阵。

4.2.1.1 对称矩阵

设有一个 $n \times n$ 的矩阵 A ，若在矩阵中有

$$a_{ij} = a_{ji}, \quad 1 \leq i, j \leq n$$

则称矩阵 A 为对称矩阵。

根据对称矩阵的结构，不难发现，若仅存储矩阵的对角线及对角线以上或以下的元素，可以完全对应该矩阵，我们称此方法为对称矩阵的压缩存储。

以存放下三角矩阵为例，将其按行存放在一个一维数组 sa 中，可得 sa 共有 $\frac{n(n+1)}{2}$ 个元素，此时各元素在 sa 中的位置有规律可循。

1. 若 $i \geq j$ ，数据元素 $A[i][j]$ 在 sa 中的存放位置为 $1 + 2 + \cdots + i + j = \frac{i(i+1)}{2} + j$;
2. 若 $i < j$ ，数据元素 $A[i][j]$ 没有存放在 sa 中，但其对称元素 $A[j][i]$ 的存放位置为 $\frac{j(j+1)}{2} + i$ 。

相反地，若已知某矩阵元素位于 sa 的第 k 个位置，可寻找满足 $\frac{i(i+1)}{2} \leq k < \frac{(i+1)(i+2)}{2}$ 的 i 作为该元素的行号，再由 k 与 i 联立求解 $j = k - \frac{i(i+1)}{2}$ 得该元素的列号。

	0	1	2	3	4	5	6	7	8	...	$n(n+1)/2 - 1$
sa	a_{00}	a_{10}	a_{11}	a_{20}	a_{21}	a_{22}	a_{30}	a_{31}	a_{32}	...	$a_{(n-1)(n-1)}$

表 4.1: 对称矩阵在一维数组 sa 中的存储

注 与教材不同，此处遵循 C 语言的风格，规定起始存储位置为 $sa[0]$ ，故在已知 k 待求 i 与 j 时，上述公式与教材中给出的公式有较小的差异。

4.2.1.2 三对角矩阵

设有一个 $n \times n$ 的矩阵 A , 若 A 满足

$$A = \begin{bmatrix} a_{00} & a_{01} & 0 & 0 & \cdots & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & \cdots & 0 & 0 & 0 \\ 0 & a_{21} & a_{22} & a_{23} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{(n-2)(n-3)} & a_{(n-2)(n-2)} & a_{(n-2)(n-1)} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} \end{bmatrix}$$

则称矩阵 A 为三对角矩阵。

由定义可看出, 三对角矩阵 A 共有 $3n - 2$ 个非零元素。在一维数组 sa 中 $A[i][j]$ 在第 i 行前的有 $3i - 1$ 个非零元素, 在第 j 列前的有 $j - i + 1$ 个, 故 $A[i][j]$ 在 sa 中的存储位置为 $2i + j$ 。

4.2.2 稀疏矩阵

设有一个 $m \times n$ 的矩阵 A , 其中有 s 个非零元素, 若 $s \ll mn$, 则称 A 为稀疏矩阵。需要注意的一点是, 稀疏矩阵中非零元素较少, 且其分布无规律可循, 基于此原因, 我们不能再用类似于存储特殊矩阵的方法对其进行存储。

定义 4.1 (稀疏因子)

设矩阵 $A_{m \times n}$ 中有 s 个非零元素, 称 $\delta = \frac{s}{mn}$ 为 A 的稀疏因子。



笔记 通常而言, 当 $\delta \leq 0.05$ 时, 我们认为该矩阵为稀疏矩阵。

在存储稀疏矩阵时, 为节省存储空间, 应只存储非零元素。考虑到非零元素的分布无规律, 故在存储时还需同时记录其行号与列号。每个三元组 $(i, j, a[i][j])$ 唯一确定了矩阵 A 的一个非零元素, 因此稀疏矩阵可由表示非零元素的一系列三元组及其行列数唯一确定。

若把稀疏矩阵的三元组线性表按顺序存储结构进行存储, 则称其为稀疏矩阵的**三元组顺序表**。在三元组顺序表中以行为主序, 所有三元组按行号递增的顺序排列, 而行号相等的按列号递增的顺序排序。我们用变量 tu 表示三元组的个数, 用 mu 和 nu 分别表示稀疏矩阵的行数与列数。

稀疏矩阵的三元组顺序表存储结构表示如下:

```
1  #define MaxSize 100
2  typedef int ElemType;
3  typedef struct//三元组定义
4  {
5      int i,j;
6      ElemType e;
7  }Triple;
8  typedef struct//稀疏矩阵结构定义
9  {
10     int mu,nu,tu;
11     Triple data[MaxSize];
12 }TSMatrix;
```

由三元组顺序表可以求出其对应稀疏矩阵的转置: 设矩阵列数为 nu , 先对原三元组顺序表进行 nu 次扫描, 其中第 k 次扫描寻找所有列号为 k 的项, 再交换其行号与列号, 并顺次存储于转置矩阵的三元组顺序表中。若

设矩阵有 tu 个非零元素，则上述二重循环的时间复杂度为 $O(nu \times tu)$ ，代码如下：

```

1  void TransposeSMatrix(TSMatrix M,TSMatrix &T)
2  {
3      T.mu=M.nu,T.nu=M.mu,T.tu=M.tu;//相关参数赋值
4      if(T.tu)
5      {
6          int q=1;
7          for(int k=1;k<=M.nu;k++)
8              for(int p=1;p<=M.tu;p++)
9                  if(M.data[p].j==k)
10                     {
11                         T.data[q].i=M.data[p].j;
12                         T.data[q].j=M.data[p].i;
13                         T.data[q].e=M.data[p].e;
14                         q++;
15                     }
16      }
17  }

```

不难发现上述算法的时间复杂度过大，原因在于二重循环会多次访问已扫描过的项。我们给出快速转置的思想：对原矩阵 A 扫描一遍，按其中每一元素的列号，立即确定其在转置矩阵三元组顺序表中的位置。为实现此过程，我们需要建立两个辅助数组 $num[col]$ 与 $cpot[col]$ ，前者记录矩阵转置前各列非零元素个数，后者记录转置后各行第一个非零元素在转置矩阵三元组顺序表中开始存放的位置。该算法代码如下：

```

1  void FastTransposeSMatrix(TSMatrix M,TSMatrix &T)
2  {
3      int p,q,t,col,num[MaxSize],cpot[MaxSize];
4      T.mu=M.nu,T.nu=M.mu,T.tu=M.tu;//相关参数赋值
5      if(T.tu)
6      {
7          for(col=0;col<M.nu;col++) num[col]=0;//计数器初始化
8          for(t=0;t<M.tu;t++) ++num[M.data[t].j];//求M中每一列非零元素的个数
9          cpot[0]=0;
10         for(col=1;col<M.nu;col++) cpot[col]=cpot[col-1]+num[col-1];
11         for(p=0;p<M.tu;p++)
12         {
13             col=M.data[p].j;//当前元素在M中的列数
14             q=cpot[col];//当前的元素在T中的序号
15             T.data[q].i=M.data[p].j;
16             T.data[q].j=M.data[p].i;
17             T.data[q].e=M.data[p].e;
18             cpot[col]++;
19         }
20     }
21 }

```

通过计算可以得出该算法的时间复杂度为 $O(nu + tu)$ ，优于普通的转置算法。

为便于随机存取任意一行的非零元素，需知道每一行的第一个非零元素在三元组顺序表中的位置，故可以

将指示行信息的辅助数组 `rpos` 固定在稀疏矩阵的存储结构中，称此种“带行链接信息”的三元组顺序表为**行逻辑链接顺序表**，其存储结构表示如下：

```

1  #define MAXRC 20
2  typedef struct
3  {
4      Triple data[MaxSize];
5      int rpos[MAXRC];
6      int mu,nu,tu;
7  }RLSMatrix;

```

在执行稀疏矩阵相关的运算操作时，使用三元组顺序表会有诸多不便之处，故考虑采用链式存储结构对稀疏矩阵进行存储。在该存储结构中，一个结点除了数据域 (i, j, e) 之外，还包含了两个指针域 `right` 和 `down`，前者指向同一行中的下一个元素，后者指向同一列中的下一个元素。由此整个矩阵构成了一个十字交叉的链表，称为**十字链表**，其存储结构表示如下：

```

1  typedef struct OLNode
2  {
3      int i,j;//该非零元素的行号与列号
4      ElemType e;//非零元素的值
5      struct OLNode *right,*down;//该非零元所在行与列的后继链域
6  }OLNode,*OLink;
7  struct CrossList
8  {
9      OLink *rhead,*chead;
10     int mu,nu,tu;
11 };


```

4.3 广义表

广义表是 $n > 0$ 个元素的有限序列，记作


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

其中 α_i 为原子项或广义表， n 为广义表长度。当 LS 非空时称 α_1 为**表头**，而其余元素组成的子表 $(\alpha_2, \dots, \alpha_n)$ 称为**表尾**。由定义可看出，对于任何一个非空广义表，其表头可能是原子或广义表，但其表尾一定是广义表。

 **笔记** 对于广义表 $B = (a, (e))$ ，其长度为 2，表头为 a ，表尾为 $((e))$ ；对于广义表 $E = (a, E) = (a, (a, (a, \dots)))$ ，其长度为 2，表头为 a ，表尾为 (E) 。

广义表具有以下结构特点：

1. 广义表中的数据元素有相对次序；
2. 广义表的长度为最外层所包含的元素个数；
3. 广义表的深度为所包含的括弧的重数；
4. 广义表可以共享；
5. 广义表可以是一个递归的表，其长度为有限值，但深度为无限值。

 **笔记** 原子的深度为 0，空表的深度为 1。

由于广义表 $LS = (\alpha_1, \alpha_2, \dots, \alpha_n)$ 中的数据元素可以具有不同的结构，因此通常采用链式存储结构表示。

在头尾存储结构中，每个元素都用一个结点表示，共有两种结点：一种是针对子表的表结点，内含标志域 `tag`（为1）和指针域 `hp`（指向表头）、`tp`（指向表尾）；另一种是针对原子的原子结点，内含标志域 `tag`（为0）和原子值域 `atom`。

头尾存储结构的结点类型声明如下：

```
1  enum ElemTag{ATOM,LIST}; //ATOM==0为原子,LIST==1为子表
2  typedef struct GLNode
3  {
4      ElemTag tag; //公共部分,用于区分原子结点和表结点
5      union //原子结点和表结点的联合部分
6      {
7          AtomType atom; //atom是原子结点的值域,AtomType由用户定义
8          struct
9          {
10             struct GLNode *hp,tp;
11             }ptr; //ptr是表结点的指针域,ptr.hp和ptr.tp分别指向表头和表尾
12     };
13 }*GList,GLNode;
```

在扩展线性链表存储结构中，每个元素用一个结点表示，共有两种结点：一种是针对子表的表结点，和头尾存储结构的表结点完全一致；另一种是针对原子的原子结点，在头尾存储结构的原子结点的基础之上新增了表尾指针 `tp`。

扩展线性链表存储结构的结点类型声明如下：

```
1  enum ElemTag{ATOM,LIST}; //ATOM==0为原子,LIST==1为子表
2  typedef struct GLNode1
3  {
4      ElemTag tag; //公共部分,用于区分原子结点和表结点
5      union //原子结点和表结点的联合部分
6      {
7          AtomType atom; //原子结点的值域
8          struct GLNode1 *hp; //表结点的表头指针
9      };
10     struct GLNode1 *tp; //相当于线性链表的next指针,指向下一个结点
11 }*GList1,GLNode1;
```

第5章 树和二叉树

5.1 树的定义与基本术语

本节我们将以文字形式给出树的逻辑结构。

1. 数据对象 D : D 为具有相同特性的数据元素的集合。
2. 数据关系 R : 若 $D = \emptyset$ 则称为空树, 否则在 D 中存在唯一的称为**根**的数据元素 **root**, 且当 $n > 1$ 时, 其余结点可分为 $m > 0$ 个互不相交的有限集 T_1, T_2, \dots, T_m , 其中每个集合本身又是一棵符合本定义树, 称为根的**子树**。

从数据结构角度而言, 树包含 $n \geq 0$ 个结点, 且当 $n = 0$ 时为**空树**。非空树的定义为

$$T = (D, R) \quad (5.1)$$

其中, D 为树中结点的有限集合, 关系 R 满足以下条件:

1. 有且仅有一个结点 $d_0 \in D$, 其对于关系 R 而言没有前驱结点, 称为树的**根结点**;
2. 除根结点 d_0 外, D 中的每个结点有且仅有一个前驱结点, 但可以有多个后继结点;
3. D 中可以有多个**终端结点**, 后者又称为**叶子**。

显然, 树的结构定义是递归的。通常, 我们可以用**树形表示法**、**文氏图表示法** (即嵌套集合)、**凹入表示法**和**广义表表示法**来表示树。

与树相关的基本术语众多, 下面将逐个列举阐述。

1. **结点**: 树的结点包含一个数据元素及若干指向其子树的分支。
2. **结点的度**: 结点拥有的子树数。
3. **非终端结点**: 度不为 0 的结点, 又称为**分支结点**。
4. **树的度**: 树内各结点的度的最大值。
5. **孩子**: 结点的子树的根。
6. **双亲**: 结点即为其孩子的双亲。
7. **兄弟**: 同一个双亲的孩子之间互称为兄弟。
8. **祖先**: 从根到该结点所经分支上的所有结点 (不包含自身)。
9. **子孙**: 以某结点为根的子树中的任一结点都称为该结点的子孙。
10. **层次**: 从根开始定义, 根处于第一层, 根的孩子处于第二层, 以此类推。
11. **堂兄弟**: 双亲在同一层的互为堂兄弟。
12. **深度**: 树中结点的最大层次, 又称为**高度**。
13. **有序树与无序树**: 如果将树中结点的各子树看成从左至右是有次序的, 则称该树为有序树, 否则为无序树。
14. **森林**: $m \geq 0$ 棵互不相交的树的集合。

在有序树中, 最左边的子树的根称为**第一个孩子**, 最右边的称为**最后一个孩子**。对树中每个结点而言, 其子树的集合即为森林, 利用这一点, 我们也可以通过森林与树相互递归的定义来描述树。

定义 5.1 (树的逻辑结构的另一种描述)

任何一棵树都是一个二元组 $Tree = (root, F)$ 。

1. $root$ 是数据元素, 即树的根结点。
2. $F = (T_1, T_2, \dots, T_m)$ 是 $m \geq 0$ 棵树的森林, 其中 $T_i = (r_i, F_i)$ 为根 $root$ 的第 i 棵子树, 且当 $m \neq 0$ 时在根与其子树森林之间有

$$RF = \{ \langle root, r_i \rangle \mid i = 1, 2, \dots, m, m > 0 \} \quad (5.2)$$

借由此我们可以得到森林和树与二叉树之间转换的递归定义。

5.2 二叉树

二叉树是另一种树形结构，其特点为每个结点至多只有两棵子树，且子树有左右之分，不能交换次序。一棵二叉树或者是空树，或者是由一个根结点和两棵互不相交的分别称为根结点的**左子树**和**右子树**所组成的非空树，且左右子树同样是二叉树。在此特别强调，二叉树与度为2的树是不同的：度为2的树至少有3个结点，而二叉树可以有0个结点；度为2的树不区分子树次序，而二叉树必须明确指出子树的次序。

性质 二叉树具有以下性质。

1. 在二叉树的第 $i \geq 1$ 层上至多有 2^{i-1} 个结点。
2. 深度为 $k \geq 1$ 的二叉树至多有 $2^k - 1$ 个结点。
3. 对任何一棵二叉树 T ，如果其叶子数为 n_0 ，度为2的结点数为 n_2 ，则 $n_0 = n_2 + 1$ 。

深度为 k 且有 $2^k - 1$ 个结点的二叉树称为**满二叉树**。我们可以对满二叉树的结点进行连续编号，规定编号从根结点开始，自上而下，从左至右。根据这种编号方式，对于深度为 k 的有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从1到 n 的结点一一对应时，称之为**完全二叉树**。换言之，若设二叉树的深度为 k ，除第 k 层外的各层结点数都达到最大个数，而第 k 层从右向左连续缺若干结点，此为完全二叉树。

性质 完全二叉树具有以下性质。

1. 具有 $n \geq 0$ 个结点的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$ 。
2. 若完全二叉树中有 n_0 个叶子，则其总结点数为 $n = 2n_0$ 或 $n = 2n_0 - 1$ 。
3. 若完全二叉树的结点数为奇数，则其没有度为1的结点；若为偶数，则其有一个度为1的结点。
4. 若对一棵有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 $1 \leq i \leq n$ 有

(a). 若 $i = 1$ ，则结点 i 为树的根结点，即结点 i 无双亲，否则结点 i 的双亲为结点 $\left\lfloor \frac{i}{2} \right\rfloor$ ；

(b). 若 $2i \leq n$ ，则结点 i 的左孩子为结点 $2i$ ，否则结点 i 没有左孩子；

(c). 若 $2i + 1 \leq n$ ，则结点 i 的右孩子为结点 $2i + 1$ ，否则结点 i 没有右孩子。

例题 5.1 一棵二叉树中总结点个数为200，其中单分支结点个数为19，求其叶子个数。

解 由题意有 $n = 200$ ， $n_1 = 19$ ，联立 $n = n_0 + n_1 + n_2$ 与 $n_0 = n_2 + 1$ 两式可得 $n_0 = 91$ ，即叶子个数为91。

例题 5.2 一棵完全二叉树中总结点个数为200，求其叶子个数。

解 由于200为偶数，设叶子个数为 n_0 ，则有 $2n_0 = 200$ ，故 $n_0 = 100$ ，即叶子个数为100。

5.3 二叉树的存储结构

与线性表类似，我们可以用顺序存储结构与链式存储结构来存储一棵二叉树。

顺序存储一棵二叉树，即用一组连续的存储单元存放二叉树中的结点。由二叉树的性质可知，对于满二叉树或完全二叉树，树中结点层序编号可以唯一地反映出结点之间的逻辑关系，故可以用一维数组按自上而下、从左至右的顺序存储树中的所有结点值，并通过数组元素的下标关系反映满二叉树或完全二叉树中结点之间的逻辑关系。显然，上述方式不能存储一般二叉树，但我们可以借鉴其思想，通过添加空结点的方式将一般二叉树转化为完全二叉树，再经编号进行顺序存储。

二叉树的顺序存储结构表示如下：

```
1  #define MAX_TREE_SIZE 100//二叉树的最大结点数
2  typedef TElemType SqBiTree[MAX_TREE_SIZE]; //0号单元存储根结点
3  //TElemType为二叉树中结点的数据值类型
4  SqBiTree bt;
```

通常来说，满二叉树和完全二叉树采用顺序存储结构较为合适，而对于一般二叉树，若其形态接近于完全二叉树，不需要增加过多空结点，也可采用顺序存储结构进行存储。但是，对于其他二叉树如**单支树**（即链形树），若再采用顺序存储结构，空间利用率会极低，此时便要考虑采用链式存储结构进行存储。

我们可以用**二叉链表**和**三叉链表**来表示一般的二叉树。在二叉链表中，结点分为 **data**、**lchild** 和 **rchild** 三个域，其中 **lchild** 域存放指向该结点的左孩子的指针，**rchild** 域存放指向该结点的右孩子的指针；而在三叉链表中，结点在二叉链表的基础上新增了 **parent** 域，以存放指向该结点的双亲的指针。

设二叉树有 n 个结点，则其有 $2n$ 个指针域、 $n-1$ 个非空指针域与 $n+1$ 个空指针域。

二叉树的二叉链式存储结构表示如下：

```

1  typedef struct BiTNode
2  {
3      TElemType data;
4      struct BiTNode *lchild,*rchild;
5  }BiTNode,*BiTree;

```

5.4 二叉树的遍历

二叉树的遍历是指按一定的次序不重复地访问树中的所有结点，即每个结点恰好只被访问一次。实际上，遍历就是把二叉树中的结点依据某种次序排列成线性序列。在二叉树中，遍历是最基本的运算，同时也是其他运算的基础。

二叉树常用的遍历有**前序遍历**、**中序遍历**与**后序遍历**，三者的区别仅在于访问根结点的顺序不同。下面将给出三种遍历的算法。

```

1  void PreOrder(BiTNode *bt)//前序遍历
2  {
3      if(bt!=NULL)
4      {
5          printf("%d ",bt->data);
6          PreOrder(bt->lchild);
7          PreOrder(bt->rchild);
8      }
9  }
10 void InOrder(BiTNode *bt)//中序遍历
11 {
12     if(bt!=NULL)
13     {
14         InOrder(bt->lchild);
15         printf("%d ",bt->data);
16         InOrder(bt->rchild);
17     }
18 }
19 void PostOrder(BiTNode *bt)//后序遍历
20 {
21     if(bt!=NULL)
22     {
23         PostOrder(bt->lchild);
24         PostOrder(bt->rchild);
25         printf("%d ",bt->data);
26     }
27 }

```

上述代码均采用递归思想书写，下面将会给出三种遍历的非递归算法。

1. 先序遍历：利用栈 LIFO 的原理，用栈保存指向根结点的指针，而后依次将右孩子与左孩子入栈，或从根结点开始，沿左孩子方向边访问边入栈，当左孩子为空时转向右孩子重复此过程。

```

1  void PreOrder1(BiTNode *T)//第一种先序遍历算法
2  {
3      BiTNode *p;
4      SqStack *st;//定义栈指针st
5      InitStack(st);//初始化栈
6      if(T!=NULL)
7      {
8          Push(st,T);//根结点入栈
9          while(!StackEmpty(st))//栈不为空时循环
10         {
11             Pop(st,p);//将栈顶元素赋给p后删除栈顶元素，并访问p
12             printf("%c ",p->data);
13             if(p->rchild!=NULL) Push(st,p->rchild);//当p有右孩子时将后者入栈
14             if(p->lchild!=NULL) Push(st,p->lchild);//当p有左孩子时将后者入栈
15         }
16     }
17 }
18 void PreOrder2(BiTNode *T)//第二种先序遍历算法
19 {
20     BiTNode *p=T;
21     SqStack *st;//定义栈指针st
22     InitStack(st);//初始化栈
23     while(!StackEmpty(st)||p!=NULL)
24     {
25         while(p!=NULL)//访问结点p及其所有左下结点并入栈
26         {
27             printf("%c ",p->data);//访问结点p
28             Push(st,p);//结点p入栈
29             p=p->lchild;//移动到p的左孩子
30         }
31         if(!StackEmpty(st))//栈非空
32         {
33             Pop(st,p);//将栈顶元素赋给p后删除栈顶元素
34             p=p->rchild;//转向处理p的右孩子
35         }
36     }
37 }

```

2. 中序遍历：由第二种先序遍历算法改进而得，区别在于其结点在出栈时才进行访问，在此仅展示部分代码。

```

1  while(!StackEmpty(st)||p!=NULL)
2  {
3      while(p!=NULL) {Push(st,p);p=p->lchild;}
4      if(!StackEmpty(st)) {Pop(st,p);printf("%c ",p->data);p=p->rchild;}
5  }

```

3. 后序遍历：由于后序遍历的特殊性，在用栈存储结点时需要分清当前根结点是从左孩子还是从右孩子返回，故引入辅助指针 r 指向最近访问过的结点。

```

1  void PostOrder(BiTNode *T)
2  {
3      BiTNode *p=T,*r=NULL;
4      SqStack *st;//定义栈指针st
5      InitStack(st);//初始化栈
6      while(!StackEmpty(st)||p!=NULL)
7      {
8          if(p!=NULL) {Push(st,p);p=p->lchild;}//使p指向最左边的结点
9          else
10         {
11             GetTop(st,p);//取栈顶结点
12             if(p->rchild&& p->rchild!=r)//若右孩子存在且未被访问过
13             {
14                 p=p->rchild;//转向右孩子
15                 Push(st,p);//右孩子入栈
16                 p=p->lchild;//使p走到当前子树最左边的结点
17             }
18             else
19             {
20                 Pop(st,p);
21                 printf("%c ",p->data);//出栈并进行访问
22                 r=p;//记录最近访问的结点
23                 p=NULL;//重置p指针
24             }
25         }
26     }
27 }

```

5.5 与二叉树相关的递归算法设计

通常而言，递归算法设计是从递归数据结构的基本递归运算入手的。对于以二叉链表为存储结构的二叉树而言，其基本递归运算就是求一个结点 p 的左子树和右子树，且所求二者一定也为二叉树。

有关二叉树的递归求解问题思考步骤如下：

1. 对于二叉树 T ，设 $f(T)$ 为求解的原问题， $f(T \rightarrow \text{lchild})$ 与 $f(T \rightarrow \text{rchild})$ 为“较小问题”；
2. 设“较小问题”可求，解出 $f(T)$ 、 $f(T \rightarrow \text{lchild})$ 与 $f(T \rightarrow \text{rchild})$ 之间的关系，从而得到递归体；
3. 考虑 T 为空树或只有一个结点的情况，从而得到递归出口。

例题 5.3 设二叉树 T 中所有结点值为整数，采用二叉链表存储结构，求 T 所有结点值之和。

解 由题，可得递归算法如下：

```

1  int Sum(BiTNode *T)
2  {
3      if(T==NULL) return 0;
4      else return (T->data+Sum(T->lchild)+Sum(T->rchild));
5  }

```

例题 5.4 试设计销毁二叉树 T 的算法。

解 由题，可得递归算法如下：

```

1  void DestroyBTree(BiTreeNode *&T)
2  {
3      if(T!=NULL) {DestroyBTree(T->lchild);DestroyBTree(T->rchild);free(T);}
4      else return;
5  }

```

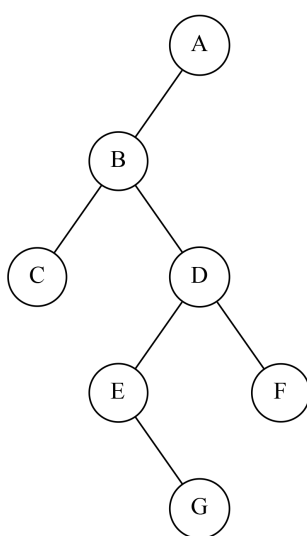
例题 5.5 给定一串字符，其中“#”代表空结点，要求设计算法构建以该字符串为先序遍历序列的二叉树，如输入字符串“ABC##DE#G##F###”，构建出如图 5.1(a) 所示的二叉树。

解 为能构建出二叉树，我们必须向二叉树中添加以“#”为值的空结点，并将其作为递归出口。每读入一个值，就为其建立一个结点以作为子树的根，其地址通过函数的引用型参数 T 直接链接到作为实际参数的指针中，随后分别对根的左右子树进行递归，直到读入“#”停止递归过程。

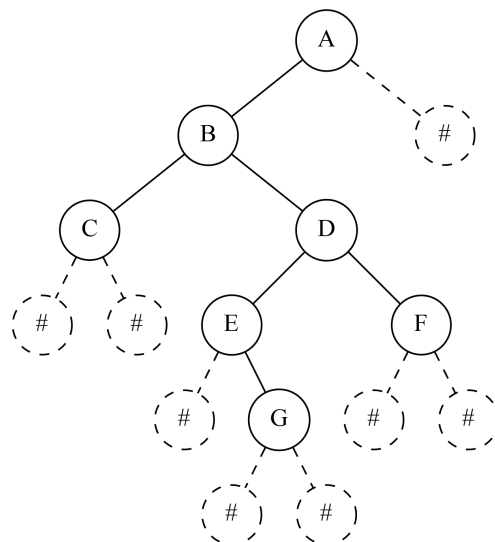
```

1  void CreateBiTree(BiTree &T)
2  {
3      TElemType ch;
4      scanf("%c",&ch);
5      if(ch=='#') T=NULL;//空树
6      else
7      {
8          T=(BiTree)malloc(sizeof(BiTreeNode));
9          if(!T) exit(OVERFLOW);//若申请空间失败则报错
10         T->data=ch;//生成根结点
11         CreateBiTree(T->lchild);//构造左子树
12         CreateBiTree(T->rchild);//构造右子树
13     }
14 }

```



(a) 题意说明图



(b) 加入空结点后的题意说明图

图 5.1: 例题 5.5 相关图

例题 5.6 试设计求二叉树深度的算法。

解 由题, 可得递归算法如下:

```

1  int bt_Height(BiTNode *T)
2  {
3      if(T==NULL) return 0; //空树
4      else if(T->lchild==NULL&&T->rchild==NULL) return 1; //只有一个结点
5      else return max(bt_Height(T->lchild),bt_Height(T->rchild))+1; //左右子树的深度取最大值后加1
6      /*下列代码亦可
7          int lh,rh;
8          if(T==NULL) return 0;
9          else
10         {
11             lh=bt_Height(T->lchild);
12             rh=bt_Height(T->rchild);
13             return (lh>rh)?(lh+1):(rh+1);
14         }
15     */
16 }
```

例题 5.7 试设计求二叉树结点数的算法。

解 由题, 可得递归算法如下:

```

1  int NodeCount(BiTNode *T)
2  {
3      int lnum,rnum;
4      if(T==NULL) return 0;
5      else
6      {
7          lnum=NodeCount(T->lchild);
8          rnum=NodeCount(T->rchild);
9          return (lnum+rnum+1);
10     }
11 }
```

例题 5.8 试设计求二叉树叶子数的算法。

解 由题, 可得递归算法如下:

```

1  int LeafCount(BiTNode *T)
2  {
3      int lnum,rnum;
4      if(T==NULL) return 0;
5      else if(T->lchild==NULL&&T->rchild==NULL) return 1;
6      else
7      {
8          lnum=LeafCount(T->lchild),rnum=LeafCount(T->rchild);
9          return (lnum+rnum);
10     }
11 }
```

例题 5.9 试设计计算二叉树中单分支结点个数的算法。

解 由题，可得递归算法如下：

```

1  int DegreeOne(BiTNode *T)
2  {
3      int lnum,rnum,n;
4      if(T==NULL) return 0;
5      else
6      {
7          if (T->lchild==NULL&&T->rchild!=NULL
8              ||T->lchild!=NULL&&T->rchild==NULL) n=1;
9          else n=0;
10     }
11     lnum=DegreeOne(T->lchild);
12     rnum=DegreeOne(T->rchild);
13     return (lnum+rnum+n);
14 }
```

5.6 线索二叉树

由二叉树的相关性质可知，对于有 n 个结点的二叉树，则在其二叉链表存储结构中有 $n+1$ 个空指针域。若我们利用这些空指针域存放在某种遍历次序下各结点的前驱结点与后继结点，则这些指针称为**线索**，并称加上线索的二叉树为**线索二叉树**。类似地，线索二叉树也分为**前序线索二叉树**、**中序线索二叉树**与**后序线索二叉树**。

为与一般的二叉树相区分，线索二叉树的存储结构在二叉链表的基础上新增了标志域 **LTag** 与 **RTag**，其中有

$$\begin{aligned}
 \text{LTag} &= \begin{cases} 0, & \text{表示 lchild 指向结点的左孩子} \\ 1, & \text{表示 lchild 指向结点的前驱结点，即 lchild 为线索} \end{cases} \\
 \text{RTag} &= \begin{cases} 0, & \text{表示 rchild 指向结点的右孩子} \\ 1, & \text{表示 rchild 指向结点的后继结点，即 rchild 为线索} \end{cases}
 \end{aligned}$$

线索二叉树的类型声明如下：

```

1  enum PointerTag {Link,Thread}; //Link==0为指针,Thread==1为线索
2  typedef struct BiThrNode
3  {
4      TElemType data;
5      struct BiThrNode *lchild,*rchild; //左右孩子指针
6      PointerTag LTag,RTag; //左右标志
7  }BiThrNode,*BiThrTree;
```

为方便操作，我们通常向线索二叉树中增加一个头结点。对于某种遍历序列，线索二叉树的首结点的 **lchild** 域与尾结点的 **rchild** 域指针指向头结点，而头结点的 **lchild** 域指针指向根结点，**rchild** 域指针指向尾结点，即有头结点的 **LTag** 为 0，**RTag** 为 1。

一般地，我们按如下算法来创建一棵线索二叉树，在此以中序线索化一棵二叉树为例，前序与后序同理：

- 中序遍历该二叉树，并在遍历过程中检查当前结点的左右孩子是否为空，若为空则将其置为指向前驱结点或后继结点的线索；
- 具体地，我们可以先创建一个头结点 `head`，在进行中序遍历过程中利用初始时指向头结点的全局指针 `pre` 以指向当前结点 `p` 的前驱结点。

```

1  BiThrNode *pre;//定义pre为全局变量
2  void InOrderThreading(BiThrTree &head,BiThrTree T)
3  {
4      if(!(head=(BiThrTree)malloc(sizeof(BiThrNode)))) exit(OVERFLOW);//申请头结点空间失败
5      head->LTag=0,
6      head->RTag=1;//建立头结点
7      head->rchild=head;//右指针回指自身
8      if(!T) head->lchild=head;//若为空树,则左指针回指自身
9      else
10     {
11         head->lchild=T,pre=head;
12         Thread(T);//中序线索化二叉树
13         pre->rchild=head,
14         pre->RTag=1,
15         head->rchild=pre;//最后一个结点线索化
16     }
17 }
18 void Thread(BiThrNode *&p)
19 {
20     if(p!=NULL)
21     {
22         Thread(p->lchild);//左子树线索化
23         if(p->lchild==NULL)
24             p->lchild=pre,p->LTag=1;//给结点p添加前驱线索
25         else p->LTag=0;
26         if(pre->rchild==NULL)
27             pre->rchild=p,pre->RTag=1;//给结点pre添加后继线索
28         else pre->RTag=0;
29         pre=p;
30         Thread(p->rchild);//右子树线索化
31     }
32 }

```

5.7 树与森林

5.7.1 树的存储结构

在大量应用中，人们曾使用多种形式的存储结构来表示树，下面将逐个列举阐述。

双亲存储结构是一种顺序存储结构，用一组连续空间存储树的所有结点，同时在每个结点中设置一个下标域以指示其双亲结点的位置，其中根结点的下标域为 -1 。这种存储结构利用了除根外的每个结点只有唯一双亲的性质，然而，若要查找某结点的所有孩子，则需遍历一次此结构。

双亲存储结构的类型声明如下：


```

1  #define MAX_TREE_SIZE 100
2  typedef struct PTreeNode
3  {
4      TElemType data;
5      int parent; // 双亲位置域
6  } PTreeNode;
7  typedef struct PTree
8  {
9      PTreeNode nodes[MAX_TREE_SIZE];
10     int r, n; // 根的位置与结点数
11 } PTree;

```

孩子表示法是建立在树中每个结点可能有多棵子树的事实之上的，使用了多重链表，每个结点由有个指针域，而其中每个指针指向一棵子树的根结点。对于链表中的结点，主要有两种设计方式。

1. **定长结点链表存储结构**：孩子链表存储结构可按树的度以设计结点的孩子结点的指针域个数；若拥有 n 个结点的树的度为 k ，则此链表中有 $n(k-1)+1$ 个空指针域。
2. **孩子链表存储结构**：将 n 个头指针组成一个线性表，采用顺序存储结构进行存储；同时将每个结点的孩子结点排列起来，视为一个线性表，且以单链表作为存储结构，`data` 域是每个结点在顺序表中的下标值，则 n 个结点有 n 个孩子链表，其中叶子结点的孩子链表为空表。

 **笔记** 与双亲存储结构不同，孩子表示法适用于涉及孩子结点的操作，但不适用于对双亲结点的操作。当然，我们可以将这两种存储方式合并为一种，即在孩子表示法的顺序表中扩充一列以存放各结点的双亲结点。

孩子兄弟表示法又称二叉树表示法或二叉链表表示法，其以二叉链表作为树的存储结构，链表中结点的两个链域分别指向该结点的第一个孩子结点和下一个兄弟结点，分别命名为 `firstchild` 域和 `nextsibling` 域。

孩子兄弟表示法的存储结构的类型声明如下：

```

1  typedef struct CSNode
2  {
3      TElemType data;
4      CSNode *firstchild, *nextsibling;
5  } CSNode, *CSTree;

```

5.7.2 森林与二叉树的转换

由于树和二叉树都可用二叉链表作为存储结构，则我们能以二叉链表为媒介导出树和二叉树之间的对应关系；换言之，给定一棵树，可以找到唯一的一棵二叉树与之对应，从物理结构上看二者的二叉链表是相同的，只是解释不同而已。

将一棵树转换成二叉树的过程如下：

1. 树中所有兄弟结点之间增加一条连线；
2. 对树中的每个结点，只保留其与第一个孩子结点之间的连线，并删除其与其他孩子结点之间的连线；
3. 以树的根结点为轴心，将整棵树顺时针旋转 45° ，使之结构层次分明。

将由两棵或两棵以上的树组成的森林转换成二叉树的过程如下：

1. 将森林中的每棵树转换成相应的二叉树；
2. 第一棵二叉树保持不变，从第二棵二叉树开始，依次将该二叉树的根结点作为前一棵二叉树根结点的右孩子结点，如此循环，当所有二叉树连在一起后，所得到的二叉树即由森林转换得到的二叉树。

注 若一棵二叉树的根结点的右孩子为空，则该二叉树由树转换而来，否则由森林转换而来；当一棵二叉树是由 m 棵树组成的森林转换得到的，则该二叉树的根结点有 $m-1$ 个右下孩子。

若一棵二叉树是由一棵树转换而来的，则该二叉树还原为树的过程如下：

1. 若某结点是其双亲结点的左孩子，则将该结点右链上的所有结点都与该结点的双亲结点用线连接起来；
2. 删除原二叉树中所有双亲结点与其右孩子之间的连线；
3. 整理由上述两步所得到的树，使之结构层次分明。

将由 m 棵树组成的森林转换得到的二叉树还原为森林的过程如下：

1. 删除二叉树根结点右链上所有结点之间的“双亲-右孩子”关系，将其分成若干个以右链上的结点为根结点的二叉树，设其为 bt_1, bt_2, \dots, bt_m ；
2. 分别将 bt_1, bt_2, \dots, bt_m 各自还原为一棵二叉树。

5.8 哈夫曼树

5.8.1 哈夫曼树的概念

设一棵二叉树具有 n 个带有权值的叶子结点，从根结点到各个叶子结点的路径长度与相应结点权值的乘积的和称为该二叉树的**带权路径长度**，记作

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

其中 w_i 为第 i 个叶子结点的权值， l_i 为第 i 个叶子结点的路径长度。

给定一组具有确定权值的叶子结点，可以构造出许多不同形态的二叉树。我们将其中具有最小带权路径长度的二叉树称为**哈夫曼树**，又称为**最优二叉树**。

构造一棵哈夫曼树的过程如下：

1. 通过给定的由 n 个权值组成的集合 $\{w_1, w_2, \dots, w_n\}$ 构造出 n 棵只有一个根结点的二叉树，从而得到一个二叉树集合 $F = \{T_1, T_2, \dots, T_n\}$ ；
2. 在 F 中选取根结点权值最小和次小的两棵二叉树 T_i, T_j 进行合并，即增加一个根结点，将 T_i, T_j 分别作为其左右子树，且该根结点的权值为其左右子树根结点权值之和；
3. 重复步骤 2，当 F 中只剩下一棵二叉树时，此二叉树即为所求的哈夫曼树。

5.8.2 哈夫曼编码

在传送电文时，我们希望文长尽可能地短。如果对每个字符设计长度不等的编码，且让电文中出现次数较多的字符采用尽可能短的编码，则传送电文的总长度便可减少。若要设计长短不等的编码，则必须满足任意一个字符的编码都不是另一个字符的编码的前缀，这种编码称为**前缀编码**。

利用哈夫曼树构造的用于通信的二进制编码称为**哈夫曼编码**，具体构造方式如下：

1. 用给定的若干字符的频度为权值生成哈夫曼树，并在叶子结点上注明对应的字符；
2. 树中从根到每个叶子都有一条路径，对路径上的各分支约定指向左子树根的分支表示为“0”码，指向右子树根的分支表示为“1”码，并取每条路径上的“0”和“1”的序列作为各叶子对应的字符的编码，即可构造出哈夫曼编码。

第6章 图

6.1 图的基本概念

在图形结构中，结点之间的关系可以是任意的，图中任意数据元素之间都可能相关。

定义 6.1 (图)

图 G 由两个集合 V 和 E 组成，通常记作 $G = (V, E)$ ，其中 V 是顶点的有限集合，记作 $V(G)$ ，而 E 是连接 V 中两个不同顶点的边的有限集合，记作 $E(G)$ 。

下面我们将介绍图中的一些基本概念。

对于一个图 G ，若边集 $E(G)$ 为无向边的集合，则称 G 为**无向图**，而若 $E(G)$ 为有向边的集合，则称 G 为**有向图**。在一个无向图中，若存在一条边 (i, j) ，则称顶点 i, j 为该边的两个**端点**，并称其互为**邻接点**，又称为**相邻点**；在一个有向图中，若存在一条边 $\langle i, j \rangle$ ，则称该边为顶点 i 的一条**出边**与顶点 j 的一条**入边**，且称顶点 i, j 分别为此边的**起始顶点**和**终止顶点**。对于边 $\langle i, j \rangle$ ，称顶点 i **邻接到**顶点 j 、顶点 j **邻接自**顶点 i 、边 $\langle i, j \rangle$ 与顶点 i, j **相关联**。


对于一个顶点 v ，我们将其**度**记作 $D(v)$ 。在一个无向图中，每个顶点的度定义为与其相关联的边的数目；在一个有向图中，每个顶点的度为**入度**与**出度**之和，前者是以该顶点为终点的入边数目，而后者是以该顶点为起点的出边数目。

定理 6.1 (边与度的关系)

若一个图中有 n 个顶点与 e 条边，顶点 v_i 的度为 d_i ($0 \leq i \leq n-1$)，则有

$$e = \frac{1}{2} \sum_{i=0}^{n-1} d_i$$

设有两个图 $G = (V, E)$ 与 $G' = (V', E')$ ，若 $V' \subseteq V$ 且 $E' \subseteq E$ ，则称 G' 为 G 的**子图**。

 **笔记** 对于一个图 $G = (V, E)$ ，若 $V' \subseteq V$ 且 $E' \subseteq E$ ，则有可能 (V', E') 不是一个图，即由 V 的子集与 E 的子集不一定能构造出 G 的子图。

对于一个有 n 个顶点的无向图，若其具有 $\frac{n(n-1)}{2}$ 条边，则称之为**完全无向图**；对于一个有 n 个顶点的有向图，若其具有 $n(n-1)$ 条边，则称之为**完全有向图**。

边数较少（即 $e \ll n \log_2 n$ ，其中 n 为顶点数）的图称为**稀疏图**，边数较多的图称为**稠密图**。

在图 G 中，从顶点 i 到顶点 j 的一条**路径**是一个顶点序列 $\{i = i_0, i_1, \dots, i_m = j\}$ 。若 G 为无向图，则有 $(i_{k-1}, i_k) \in E(G)$ ($1 \leq k \leq m$)；若 G 为有向图，则有 $\langle i_{k-1}, i_k \rangle \in E(G)$ ($1 \leq k \leq m$)，其中顶点 i 称为此路径的**起始点**，顶点 j 称为该路径的**结束点**。**路径长度**指的是一条路径上所经过的边的数目。若一条路径的顶点序列中没有出现重复的顶点，则称此路径为**简单路径**。

若一条路径上的起始点与结束点为同一个顶点，则称该路径为**回路**，又称为**环**；除起始点与结束点相同以外，其余顶点不重复出现的回路称为**简单回路**，又称为**简单环**。

在无向图 G 中，若在顶点 i 到顶点 j 之间存在路径，则称顶点 i 与顶点 j 是**连通**的。若 G 中任意两个顶点都是连通的，则称 G 为**连通图**，否则为非连通图。无向图的极大连通子图称为其**连通分量**。

在有向图 G 中，若任意两个顶点都是连通的，则称该图是**强连通图**。有向图的极大强连通子图称为其**强连通分量**。

在一个图中，每条边可以标上具有某种含义的数值，此数值称为该边的**权**。边上带权的图称为**带权图**，又称为**网**。

6.2 图的存储结构

6.2.1 邻接矩阵

定义 6.2 (邻接矩阵)

邻接矩阵是表示顶点之间相邻关系的矩阵；具体地，设 $G = (V, E)$ 是具有 n 个顶点的简单图，顶点编号依次为 $0, 1, \dots, n-1$ ，则 G 的邻接矩阵为如下定义的 n 阶方阵 A ：

1. 若 G 为不带权的图，则有

$$A[i][j] = \begin{cases} 1, & \begin{cases} (i, j) \in E(G) \vee (j, i) \in E(G), & \text{在无向图中} \\ \langle i, j \rangle \in E(G), & \text{在有向图中} \end{cases} \\ 0, & \text{其他} \end{cases}$$

2. 若 G 为带权图，则有

$$A[i][j] = \begin{cases} w_{ij}, & \begin{cases} (i, j) \in E(G) \vee (j, i) \in E(G), & \text{在无向图中} \\ \langle i, j \rangle \in E(G), & \text{在有向图中} \end{cases} \\ 0, & i = j \\ \infty, & \text{其他} \end{cases}$$



对于具有 n 个顶点和 e 条边的图采用邻接矩阵方式存储时所占用的空间为 $O(n^2)$ ，与边数无关。邻接矩阵非常适合存储稠密图，且非常容易判断任意两个顶点之间是否存在边。所有的图的邻接矩阵都是唯一的。

邻接矩阵的类型声明如下：

```

1  #define INF INT_MAX//用整型最大值代表无穷大
2  #define MAXVEX 30//图中最大顶点个数
3  typedef char VertexType[4];//定义VertexType 为字符串类型
4  typedef struct vertex
5  {
6      int adjvex;//顶点编号
7      VertexType data;//顶点信息
8  }VType;//顶点类型
9  typedef struct graph
10 {
11     int n,e;//n为实际顶点数,e为实际边数
12     VType vexs[MAXVEX];//点集
13     int edges[MAXVEX][MAXVEX];//边集
14 }MGraph;//图的邻接矩阵类型

```

在邻接矩阵中有以下两种基本算法。

1. 建立邻接矩阵算法：由邻接矩阵数组 A 、顶点数 n 和边数 e 建立图 G 的邻接矩阵存储结构。

```

1  void CreateGraph(MGraph &g,int A[][MAXVEX],int n,int e)
2  {
3      g.n=n,g.e=e;
4      for(int i=0;i<n;i++){for(int j=0;j<n;j++) g.edges[i][j]=A[i][j];}
5  }

```


2. 求顶点度算法：无向图和有向图的算法有所区别。

```

1    int Degree1(MGraph g,int v)
2    {
3        int d=0;
4        if(v<0||v>=g.n)
5            return -1;//顶点编号错误返回-1
6        for(int i=0;i<n;i++)//对第v行符合条件的值进行计数
7            if(g.edges[v][i]>0&&g.edges[v][i]<INF)
8                d++;
9        return d;
10   }
11   int Degree2(MGraph g,int v)
12   {
13       int d;
14       int d1=0,d2=0;
15       if(v<0||v>=g.n)
16           return -1;//顶点编号错误返回-1
17       for(int i=0;i<n;i++)//对第v行符合条件的值进行计数
18           if(g.edges[v][i]>0&&g.edges[v][i]<INF)
19               d1++;
20       for(int i=0;i<n;i++)//对第v列符合条件的值进行计数
21           if(g.edges[i][v]>0&&g.edges[i][v]<INF)
22               d2++;
23       d=d1+d2;
24       return d;
25   }

```

6.2.2 邻接表

邻接表是图的一种链式存储结构。在邻接表中，对图中每一个顶点建立一个单链表，把该顶点的所有相邻顶点串联起来，其中所有头指针构成一个数组，称为**表头结点数组**，用 adjlist 表示，第 i 个单链表 $\text{adjlist}[i]$ 中的结点表示依附于顶点 i 的边，即头指针数组元素的下标与顶点编号一致。

邻接表中的结点设计如下：

1. 表头结点分为 **data** 域和 **firstarc** 域，前者存放顶点的值，后者存放指向后续单链表的首结点的指针；
2. 单链表结点分为顶点域 **adjvex**、权值域 **weight** 和指针域 **nextarc**，其中
 - (a). **adjvex** 域存放该相邻顶点在头结点数组中的下标；
 - (b). **weight** 域存放对应边的权值；
 - (c). **nextarc** 域存放指向依附于顶点 i 的下一条边所对应的结点的指针。



笔记 对于无向图，每个顶点所对应的单链表的长度即为该顶点的度；对于有向图，每个顶点所对应的单链表的长度即为该顶点的出度。

有时，为求出有向图中每个顶点的入度，我们可以建立该有向图的**逆邻接表**，此时每个顶点所对应的单链表的长度即为该顶点的入度。

对于具有 n 个顶点和 e 条边的图采用邻接表方式存储时所占用的空间为 $O(n + e)$ 。邻接表非常适合存储稀疏图，且非常容易查找某一个顶点的所有相邻顶点。图的邻接表表示不是唯一的，因为在单链表中各相邻顶点的顺序可以有所不同。

邻接表的类型声明如下：

```

1  typedef char VertexType[10]; //VertexType为字符串类型
2  typedef struct edgenode
3  {
4      int adjvex; //相邻顶点编号
5      int weight; //边的权值
6      struct edgenode *nextarc; //下一条边所对应的顶点
7  }ArcNode; //每个顶点对应的单链表中边结点的类型
8  typedef struct vexnode
9  {
10     VertexType data; //存放顶点信息
11     ArcNode *firstarc; //指向第一条边所对应的结点
12 }VHeadNode; //单链表的头结点类型
13 typedef struct
14 {
15     int n,e;
16     VHeadNode adjlist[MAXVEX]; //单链表头结点数组
17 }ALGraph; //图的邻接表类型

```

在邻接表中有以下几种基本算法。

1. 建立邻接表算法：由邻接矩阵数组 A 、顶点数 n 和边数 e 建立图 G 的邻接表存储结构。

```

1  void CreateGraph(ALGraph *&G,int A[][MAXVEX],int n,int e)
2  {
3      ArcNode *p;
4      G=(ALGraph*)malloc(sizeof(ALGraph));
5      G->n=n;
6      G->e=e;
7      for(int i=0;i<G->n;i++)
8      {
9          G->adjlist[i].firstarc=NULL; //邻接表中所有头结点的指针域置空
10     }
11     for(int i=0;i<G->n;i++)
12     {
13         for(int j=G->n-1;j>=0;j--)
14         {
15             if(A[i][j]>0&&A[i][j]<INF) //存在一条边
16             {
17                 p=(ArcNode*)malloc(sizeof(ArcNode)); //创建结点p
18                 p->adjvex=j;
19                 p->weight=A[i][j];
20                 p->nextarc=G->adjlist[i].firstarc; //头插法
21                 G->adjlist[i].firstarc=p;
22             }
23         }
24     }
25 }

```

2. 销毁邻接表算法：在不需要使用邻接表时应用 `free` 函数释放其空间。

```

1  void DestroyGraph(ALGraph *&G)
2  {
3      ArcNode *pre,*p;
4      for(int i=0;i<G->n;i++)
5      {
6          pre=G->adjlist[i].firstarc;
7          if(pre!=NULL)
8          {
9              p=pre->nextarc;
10             while(p!=NULL) {free(pre);pre=p;p=p->nextarc;}
11             free(pre);
12         }
13     }
14     free(G);
15 }

```

3. 求顶点度算法：无向图和有向图的算法有所区别。

```

1  int Degree1(ALGraph *G,int v)
2  {
3      int d=0;
4      ArcNode *p;
5      if(v<0||v>=G->n) return -1;
6      p=G->adjlist[v].firstarc;
7      while(p!=NULL) {d++;p=p->nextarc;}
8      return d;
9  }
10 int Degree2(ALGraph *G,int v)
11 {
12     int d1=0,d2=0,d;
13     ArcNode *p;
14     if(v<0||v>=G->n)
15         return -1;
16     p=G->adjlist[v].firstarc;
17     while(p!=NULL) {d1++;p=p->nextarc;}
18     for(int i=0;i<G->n;i++)
19     {
20         p=G->adjlist[i].firstarc;
21         while(p!=NULL)
22         {
23             if(p->adjvex==v) d2++;
24             p=p->nextarc;
25         }
26     }
27     d=d1+d2;
28     return d;
29 }

```

4. 将邻接矩阵转换为邻接表算法：与建立邻接表算法基本相同。

```

1  void MatToAdj(MGraph g,ALGraph *&G)
2  {
3      ArcNode *p;
4      g=(ALGraph*)malloc(sizeof(ALGraph));
5      for(int i=0;i<g.n;i++)
6      {
7          G->adjlist[i].firstarc=NULL;
8          for(int j=g.n-1;j>=0;j--)
9          {
10             if(g.edges[i][j]!=0&&g.edges[i][j]!=INF)
11             {
12                 p=(ArcNode*)malloc(sizeof(ArcNode));
13                 p->adjvex=j;
14                 p->weight=g.edges[i][j];
15                 p->nextarc=G->adjlist[i].firstarc;
16                 G->adjlist[i].firstarc=p;
17             }
18         }
19     }
20     G->n=g.n,G->e=g.e;
21 }

```

6.3 图的遍历

给定一个图 $G = (V, E)$ 和其中的任意一顶点 v ，从顶点 v 出发，访问图 G 中的所有顶点并且每个顶点仅被访问一次，这一过程称为图的遍历。为避免同一顶点被访问多次，需要标记每一个已经访问过的顶点，为此，我们建立一个辅助数组 `visited[]`，用以标记顶点是否被访问过，初始值为 0，而一旦被访问则立即将其置为 1。

实现深度优先遍历（DFS）的递归算法如下：

```

1  visited[MAXVEX]={0}; //全局变量
2  void DFS(ALGraph *G,int v)
3  {
4      int w;
5      ArcNode *p;
6      printf("%d ",v); //访问顶点v
7      visited[v]=1;
8      p=G->adjlist[v].firstarc; //寻找v的第一个邻接点
9      while(p!=NULL) //寻找v的所有邻接点
10     {
11         w=p->adjvex; //v的邻接点w
12         if(visited[w]==0)
13             DFS(G,w); //若w未被访问过,则开始从w出发深度遍历
14         p=p->nextarc; //寻找v的下一个邻接点
15     }
16 }

```

实现广度优先遍历（BFS）的算法如下：

```

1  void BFS(ALGraph *G,int vi)
2  {
3      int v,visited[MAXVEX]={0};
4      ArcNode *p;
5      int Q[MAXVEX],front=0,rear=0;//定义循环队列Q
6      printf("%d ",vi);//访问初始顶点
7      visited[vi]=1;
8      rear=(rear+1)%MAXVEX;
9      Q[rear]=vi;//初始顶点进队
10     while(front!=rear)//队不为空时
11     {
12         front=(front+1)%MAXVEX;
13         v=Q[front];//顶点v出队
14         p=G->adjlist[v].firstarc;//寻找v的第一个邻接点
15         while(p!=NULL)//寻找v的所有邻接点
16         {
17             if(visited[p->adjvex]==0)//如未被访问
18             {
19                 printf("%d ",p->adjvex);//访问该顶点
20                 visited[p->adjvex]=1;
21                 rear=(rear+1)%MAXVEX;
22                 Q[rear]=p->adjvex;//该顶点进队
23             }
24             p=p->nextarc;//寻找v的下一个邻接点
25         }
26     }
27 }
```

例题 6.1 假设无向图 G 采用邻接表存储，试设计一个算法，判断图 G 是否连通，若连通则返回 1，否则返回 0。

解 直接遍历一遍图 G ，根据数组 `visited[]` 中各值情况判断该图是否连通。

```

1  int Connect(ALGraph *G)
2  {
3      int flag=1;
4      DFS(G,0);
5      for(int i=0;i<G->n;i++)
6      {
7          if(visited[i]==0)
8          {
9              flag=0;
10             break;
11         }
12     }
13     return flag;
14 }
```

例题 6.2 假设图 G 采用邻接表存储，试设计一个算法判断顶点 u 到顶点 v 之间是否有简单路径。

解 直接遍历图 G ，通过数组 `visited[]` 观察顶点 u 与顶点 v 之间是否存在路径。

```

1  int HasaPath(ALGraph *G,int u,int v)
2  {
3      ArcNode *p;
4      int w;
5      visited[u]=1;
6      p=G->adjlist[u].firstarc;//p指向u的第一个邻接点
7      while(p!=NULL)
8      {
9          w=p->adjvex;
10         if(w==v) return 1;
11         if(visited[w]==0)//若顶点w未被访问过
12             if(HasaPath(G,w,v)==1) return 1;//从顶点w到顶点v进行深度优先遍历
13         p=p->nextarc;
14     }
15     return 0;
16 }
```

例题 6.3 假设图 G 采用邻接表存储，试设计一个算法输出从顶点 u 到顶点 v 之间的一条简单路径。

解 增加形参 `path` 和 `d`，前者存放从顶点 u 到顶点 v 的路径，后者表示 `path` 中的路径长度，初始值为 -1 。

```

1  void FindaPath(ALGraph *G,int u,int v,int path[],int d)
2  {
3      int w;
4      ArcNode *p;
5      visited[u]=1;
6      d++;//路径长度增加1
7      path[d]=u;//将顶点u加入路径中
8      if(u==v)//找到一条路径后输出
9      {
10         for(int i=0;i<=d;i++) printf("%d ",path[i]);
11         return;
12     }
13     p=G->adjlist[u].firstarc;
14     while(p!=NULL)
15     {
16         w=p->adjvex;
17         if(visited[w]==0) FindaPath(G,w,v,path,d);
18         p=p->nextarc;
19     }
20 }
```

6.4 最小生成树

在一个无向连通图 G 中，取其所有顶点和一部分边构成一个子图 G' ，若边集 $E(G')$ 中的边既将 G 中的所有顶点连通而又不形成回路，则称子图 G' 是 G 的一棵生成树。我们仍然可以通过遍历方式产生无向图的一棵

生成树，其中采用 DFS 产生的生成树称为深度优先生成树，采用 BFS 产生的生成树称为广度优先生成树。对于连通图，仅需要从任意一个顶点开始遍历就可以得到一棵生成树；而对于非连通图，则需要从每个连通分支的任意一个顶点开始遍历得到该图的生成森林。

由一个带权无向图可能会得到多棵生成树，我们将其中具有权之和最小的生成树称为该图的最小生成树。构造一个图的最小生成树有两种算法，一种是普林算法（Prim algorithm），另一种是克鲁斯卡尔算法（Kruskal algorithm）。

6.4.1 Prim 算法

Prim 算法的具体步骤如下。

1. 选定一个顶点 v_0 ，由其出发，令 $U = \{v_0\}$ ， $TE = \{\}$ 。
2. 首先寻找图中权值最小的边 (u, v) ，其中有 $u \in U$ ， $v \in V - U$ ，且子图不构成环，然后令 $U = U \cup \{v\}$ 及 $TE = TE \cup \{(u, v)\}$ 。
3. 重复步骤 2 直到 $U = V$ 为止，此时 T 即为最小生成树。

为实现 Prim 算法，需要建立两个辅助数组 `closest[]` 和 `lowcost[]`，其中 U 中的顶点 i 满足 `lowcost[i]` 为 0，而 $V - U$ 中的顶点 j 满足 `lowcost[j]` 大于 0。对于 $V - U$ 中的顶点 j ，我们记录其到 U 的一条最小边，其中 `closest[j]` 存放该边依附的在 U 中的顶点序号，`lowcost[j]` 存放该边的权值。

Prim 算法代码如下：

```

1  void Prim(MGraph g,int v)
2  {
3      int lowcost[MAXVEX],closest[MAXVEX],min;
4      for(int i=0;i<g.n;i++)//置初始值
5      {
6          lowcost[i]=g.edges[v][i];
7          closest[i]=v;
8      }
9      for(int i=1;i<g.n;i++)//构造n-1条边
10     {
11         min=INF,k=-1;
12         for(int j=0;j<g.n;j++)//在(V-U)中找到离U最近的顶点k
13             if(lowcost[j]!=0&&lowcost[j]<min)
14             {
15                 min=lowcost[j];
16                 k=j;//k为最近顶点的序号
17             }
18         lowcost[k]=0;//标记k已经加入U
19         for(int j=0;j<g.n;j++)//修正数组lowcost[]和closest[]
20             if(lowcost[j]!=0&&g.edges[k][j]<lowcost[j])
21             {
22                 lowcost[j]=g.edges[k][j];
23                 closest[j]=k;
24             }
25     }
26 }
```

从上面给出的代码可看出，Prim 算法的时间复杂度为 $O(n^2)$ 。除此以外，我们不难发现，由于与边无关，Prim 算法非常适合于由稠密图构造最小生成树的问题。

6.4.2 Kruskal 算法

Kruskal 算法的基本思想是在连通网中按权值的递增顺序构造一棵最小生成树，其具体步骤如下。

1. 设置初始值 $U = V$ 与 $TE = \{\}$ 。
2. 选取当前图 G 中未被选择的且权值最小的边 (i, j) ，若该边加入 TE 后会形成回路则跳过，不进行任何操作，否则令 $TE = TE \cup \{(i, j)\}$ 。
3. 重复步骤 2 直到 TE 中包含 $n - 1$ 条边为止。

为判断边 (i, j) 加入 TE 后是否会形成回路，需要建立一个辅助数组 `vset[]` 以判定两个顶点之间是否连通，其中数组元素 `vset[i]` 表示序号为 i 的顶点所属的连通子图的编号。对于边 (i, j) ，若 `vset[i]` 与 `vset[j]` 相等，则不选取该边；一旦选取，则将两个连通分量中所有顶点的 `vset` 值都置为 `vset[i]` 或 `vset[j]`。

首先我们需要定义一个存储边相关信息的结构体，包含边的起始顶点、终止顶点和权值：

```
1 typedef struct {int u,v,w;} Edge;
```

其次我们还需要对边集 E 进行排序，这里采用直接插入排序法：

```
1 void SortEdge(Edge E[],int e)
2 {
3     int j,k=0;Edge tmp;
4     for(int i=1;i<e;i++)
5     {
6         tmp=E[i],j=i-1;//从右向左在有序区E[0..i-1]寻找E[i]的插入位置
7         while(j>=0&&tmp.w<E[j].w) {E[j+1]=E[j],j--;}
8         E[j+1]=tmp;//在j+1处插入tmp
9     }
10 }
```

Kruskal 算法代码如下：

```
1 void Kruskal(MGraph g) {
2     int u1,v1,j,k=0,vset[MAXVEX];Edge E[MAXE];//MAXE为边集元素数量最大值
3     for(int i=0;i<g.n;i++)//存储图中所有边的相关信息
4         for(j=0;j<=i;j++)
5             if(g.edges[i][j]!=0&&g.edges[i][j]!=INF) {E[k].u=i,E[k].v=j,E[k].w=g.edges[i][j];k++;}
6     SortEdge(E,k);//对边进行排序
7     for(int i=0;i<g.n;i++) vset[i]=i;//辅助数组初始化
8     k=1,j=0;//k表示当前所构造的生成树的边的序号,j为E数组下标
9     while(k<g.n) {//生成的边数小于n时进行循环
10         u1=E[j].u,v1=E[j].v;//取一条边的起始顶点和终止顶点
11         if(vset[u1]!=vset[v1]) {//若两个顶点不属于同一个连通分支,则选取该边
12             k++;//生成的边数增加1
13             for(int i=0;i<g.n;i++) {if(vset[i]==vset[v1]) vset[i]=vset[u1];}//统一编号
14         }
15         j++;//搜索下一条边
16     }
17 }
```

上述算法并不是最优的，经改进可达到 $O(e \log e)$ 的时间复杂度，但一般仍认为 Kruskal 算法的时间复杂度为 $O(e \log e)$ 。除此以外，由于仅与边有关，Kruskal 算法非常适合于由稀疏图构造最小生成树的问题。

6.5 拓扑排序

无环的有向图称为**有向无环图**，简称 **DAG 图**，多用来表示一个工程或系统的流程图。假设用一个有向图表示一个工程的的施工图或程序的数据流图，其中每个顶点代表一个活动，边 $\langle i, j \rangle$ 表示活动 i 必须先于活动 j 进行，则将这种图称为 **AOV 网**。不难发现，AOV 网中不允许有回路，而为检查有向图中是否存在回路，则需要采用**拓扑排序**的方法。

在 AOV 网中，顶点序列 v_1, v_2, \dots, v_n 称为一个**拓扑序列**，当且仅当该顶点序列满足下列条件：若 $\langle i, j \rangle$ 是图中的边，则在序列中顶点 v_i 必须在顶点 v_j 之前。在一个有向图中，我们会将图中顶点排成一个线性序列，而对于没有限定次序关系的顶点，则可以人为加上任意的次序关系。

拓扑排序的具体步骤如下。

1. 从 AOV 网中选择一个没有前驱的顶点并输出。
2. 从 AOV 网中删去步骤 1 中的顶点，并且删去从该顶点出发的所有有向边。
3. 重复步骤 1 和步骤 2，直到网中不再存在没有前驱的顶点为止。

经过拓扑排序后，若图中所有顶点都包含在拓扑序列中，则说明该图不存在有向回路，否则存在。

拓扑排序算法代码如下：

```

1  typedef struct//对邻接表结构稍加修改,其余部分不变
2  {
3      VertexType data;//顶点信息
4      int count;//存放顶点入度
5      ArcNode *firstarc;//指向第一条边
6  }VNode;
7  void TopSort(ALGraph *G)
8  {
9      int S[MAXVEX],top=-1;//栈S的指针为top
10     ArcNode *p;
11     for(int i=0;i<G->n;i++)
12     {
13         G->adjlist[i].count=0;//入度置初始值为0
14         p=G->adjlist[i].firstarc;//求所有顶点的入度
15         while(p!=NULL) {G->adjlist[p->adjvex].count++;p=p->nextarc;}
16     }
17     for(int i=0;i<G->n;i++) {if(G->adjlist[i].count==0) S[++top]=i;}//将所有入度为0的顶点进栈
18     while(top>-1)
19     {
20         int i=S[top--];//顶点i出栈
21         printf("%d ",i);//输出该顶点
22         p=G->adjlist[i].firstarc;//寻找顶点i的第一个邻接点
23         while(p!=NULL)//将顶点i的所有出边邻接点的入度减1
24         {
25             int j=p->adjvex;
26             G->adjlist[j].count--;
27             if(G->adjlist[j].count==0) S[++top]=j;//将入度为0的邻接点进栈
28             p=p->nextarc;//寻找下一个邻接点
29         }
30     }
31 }
```


6.6 关键路径

若用 DAG 图描述工程的预计进度, 以顶点表示事件, 有向边表示活动, 边权值 $w(a_i)$ 表示完成活动 a_i 所需的时间, 则将这种图称为 **AOE 网**, 其中入度为 0 的顶点称为**源点**, 出度为 0 的顶点称为**汇点**。由定义可以看出, 整个工程完成的时间为从 AOE 网源点到汇点的最长路径, 我们将其称为**关键路径**, 而**关键活动**即为关键路径上的边。在一个 AOE 网中, 关键路径不止一条。

在求解关键活动之前, 需要引入几个概念。

1. **事件 v 的最早开始时间**: 指的是从源点 v_0 到顶点 v 的最长路径长度, 记作 $ve(v)$, 并规定源点 v_0 的最早开始时间为 0。
2. **事件 v 的最迟开始时间**: 在保证汇点 v_{n-1} 在 $ve(v_{n-1})$ 时刻完成的前提下, 允许 v 最迟开始的时间, 记作 $vl(v)$, 即 $vl(v)$ 的求解应从 $vl(v_{n-1}) = ve(v_{n-1})$ 开始反向递推。
3. **活动 a_i 的最早开始时间**: 该活动起始顶点 x 的最早开始时间, 即 $e(a_i) = ve(x)$ 。
4. **活动 a_i 的最迟开始时间**: 该活动终止顶点 y 的最迟开始时间与 $w(a_i)$ 之差, 即 $l(a_i) = vl(y) - w(a_i)$ 。

对于每个活动 a_i , 令 $d(a_i) = l(a_i) - e(a_i)$, 若 $d(a_i) = 0$, 则称 a_i 为**关键活动**。将所有满足上述条件的 a_i 按顺序排列起来, 即可得到一条关键路径。

例题 6.4 试求出图 6.1 的关键路径。

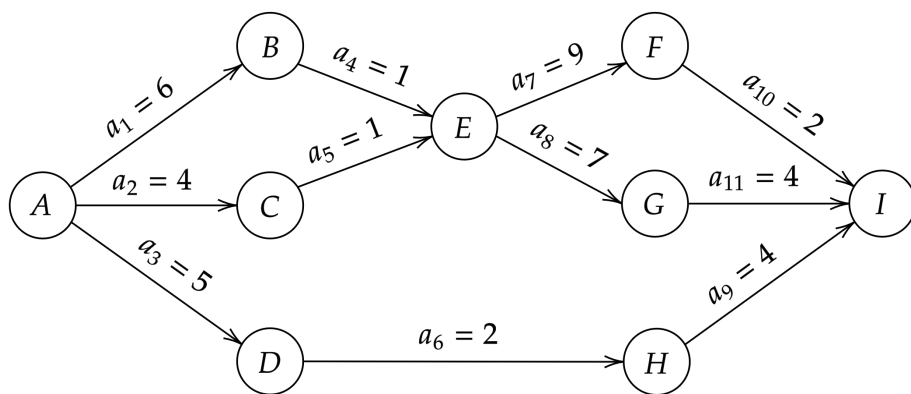


图 6.1: 例题 6.4 图

解 由图可产生一个拓扑序列 $ABCDEFGHI$, 并据此计算出各事件的最早开始时间, 结果如下:

$$ve(A) = 0, ve(B) = ve(A) + w(a_1) = 6, ve(C) = ve(A) + w(a_2) = 4, ve(D) = ve(A) + w(a_3) = 5,$$

$$ve(E) = \max\{ve(B) + w(a_4), ve(C) + w(a_5)\} = 7, ve(F) = ve(E) + w(a_7) = 16,$$

$$ve(G) = ve(E) + w(a_8) = 14, ve(H) = ve(D) + w(a_6) = 7,$$

$$ve(I) = \max\{ve(F) + w(a_{10}), ve(G) + w(a_{11}), ve(H) + w(a_9)\} = 18$$

再根据上述拓扑序列的反序 $IHG FEDCBA$ 计算各事件的最迟开始时间, 结果如下:

$$vl(I) = 18, vl(F) = vl(I) - w(a_{10}) = 16, vl(G) = vl(I) - w(a_{11}) = 14, vl(H) = vl(I) - w(a_9) = 14,$$

$$vl(E) = \min\{vl(F) - w(a_7), vl(G) - w(a_8)\} = 7, vl(D) = vl(H) - w(a_6) = 12,$$

$$vl(C) = vl(E) - w(a_5) = 6, vl(B) = vl(E) - w(a_4) = 6,$$

$$vl(A) = \min\{vl(B) - w(a_1), vl(C) - w(a_2), vl(D) - w(a_3)\} = 0$$

接着由 $e(a_i) = ve(x)$ 、 $l(a_i) = vl(y) - w(a_i)$ 与 $d(a_i) = l(a_i) - e(a_i)$ 联立求解, 可得满足 $d(a_i) = 0$ 的活动有 $a_1, a_4, a_7, a_8, a_{10}, a_{11}$, 即关键活动为 $a_1, a_4, a_7, a_8, a_{10}, a_{11}$, 故关键路径为 $ABEFI$ 和 $ABEGI$ 。

6.7 Dijkstra 算法

求带权有向图的最短路径问题分为两种情况, 第一种是求从一个顶点到其他各顶点的最短路径, 称为**单源最短路径问题**; 第二种是求每对顶点之间的最短路径, 称为**多源最短路径问题**。为解决单源最短路径问题, 迪杰斯特拉 (Dijkstra) 提出了“按路径长度的递增次序逐步产生最短路径”的算法: 先求出一条长度最短的路径, 再参照其求出长度次短的一条路径, 依此类推, 直到从源点到其他各顶点的最短路径全部被求出为止。我们将这种算法称为 **Dijkstra 算法**。

Dijkstra 算法的具体步骤如下。

1. 初始时, 顶点集 S 只包含源点 v , 即 $S = \{v\}$, 且顶点 v 到自身的距离为 0; 顶点集 $U = V - S$ 包含除 v 外的其他顶点, 且顶点 v 到 U 中顶点 i 的距离为边上的权值, 若边 $\langle v, i \rangle$ 不存在则置为 ∞ 。
2. 从 U 中选取一个顶点 u , 其为源点 v 到 U 中距离最小的一个顶点, 然后将顶点 u 加入 S 中, 此时该距离即为从源点 v 到顶点 u 的最短路径长度。
3. 以顶点 u 为新考虑的中间点, 修改源点 v 到 U 中各顶点 j 的距离。
4. 重复步骤 2 和步骤 3, 直到 S 包含所有的顶点为止, 即 U 为空。

为实现 Dijkstra 算法, 我们设置一个距离数组 $\text{dist}[]$, 其中第 i 项用以保存从源点 v 到顶点 i 的目前最短路径长度, 另设置一个数组 $\text{path}[]$, 其中第 j 项用以保存从源点到顶点 j 的最短路径, 实际上保存的是该最短路径中顶点 j 的前驱顶点。当求出最短路径后, 我们可以由 $\text{path}[j]$ 向前推出从源点到顶点 j 的最短路径。

Dijkstra 算法代码如下:

```

1  void Dijkstra(MGraph g,int v)
2  {
3      int dist[MAXVEX],path[MAXVEX],S[MAXVEX],mindis,u=0;
4      for(int i=0;i<g.n;i++)
5      {
6          dist[i]=g.edges[v][i],S[i]=0;//距离初始化并将S[]置空
7          if(g.edges[v][i]<INF) path[i]=v;//当源点v到顶点i有边时,置顶点i的前驱结点为v
8          else path[i]=-1;//否则置为-1
9      }
10     S[v]=1;//将源点v放入S中
11     for(int i=0;i<g.n-1;i++)//循环向S中添加n-1个顶点
12     {
13         mindis=INF;//mindis置最小长度初值
14         for(int j=0;j<g.n;j++)//选取不在S中且有最小距离的顶点u
15             if(S[j]==0&&dist[j]<mindis) {u=j;mindis=dist[j];}
16         S[u]=1;//将顶点u加入S中
17         for(int j=0;j<g.n;j++)//修改不在S中的顶点的距离
18             if(S[j]==0)
19                 if(g.edges[u][j]<INF&&dist[u]+g.edges[u][j]<dist[j]) {
20                     dist[j]=dist[u]+g.edges[u][j];
21                     path[j]=u;
22                 }
23     }
24 }
```

6.8 Floyd 算法

实际上, Dijkstra 算法可以同样可以解决多源最短路径问题, 时间复杂度为 $O(n^3)$, 而弗洛伊德 (Floyd) 提出了另外一种算法, 其时间复杂度仍为 $O(n^3)$, 但算法形式更为简明, 步骤更为清晰。

假设有向图 $G = (V, E)$ 用邻接矩阵 g 表示, 并设置二维数组 D 用于存放当前顶点之间的最短路径长度。Floyd 算法的基本思想是递推产生一个矩阵序列 $D_0, D_1, \dots, D_k, \dots, D_{n-1}$, 其中 $D_k[i][j]$ 表示从顶点 i 到顶点 j 的路径上所经过的顶点序号不大于 k 的最短路径长度, 即

$$\begin{cases} D_{-1}[i][j] = g.edges[i][j] \\ D_k[i][j] = \min\{D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j]\}, & 0 \leq k \leq n-1 \end{cases}$$

此外, 类似于 Dijkstra 算法, 我们使用二维数组 $path[][]$ 保存最短路径, 其与当前迭代次数有关, 即当迭代完毕后, $path[i][j]$ 存放着在从顶点 i 到顶点 j 的最短路径中 j 的前驱顶点。

Floyd 算法代码如下:

```

1  void Floyd(MGraph g)
2  {
3      int D[MAXVEX][MAXVEX];
4      int path[MAXVEX][MAXVEX];
5      for(int i=0; i<g.n; i++) //将数组D和path赋初值
6          for(int j=0; j<g.n; j++)
7              {
8                  D[i][j]=g.edges[i][j];
9                  if(i!=j&&g.edges[i][j]<INF) path[i][j]=g.edges[i][j]; //顶点i和顶点j之间有边时
10                 else path[i][j]=-1; //顶点i和顶点j之间没有边时
11             }
12     for(int k=0; k<g.n; k++)
13         for(int i=0; i<g.n; i++)
14             for(int j=0; j<g.n; j++)
15                 if(D[i][j]>D[i][k]+D[k][j]) //若找到更短的路径
16                     {
17                         D[i][j]=D[i][k]+D[k][j]; //修改路径长度
18                         path[i][j]=path[k][j]; //修改最短路径
19                     }
20 }
```

例题 6.5 给定如图 6.2 所示的 n 个村庄之间的交通图, 其中边上的权值表示道路的长度。现要在 n 个村庄中选定一个村庄建造一所医院, 试设计一个算法求此医院应建在哪所村庄可使得其到其他所有村庄的路径总和最少。

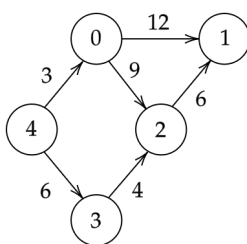


图 6.2: 例题 6.5 图

解 假设该图使用邻接矩阵 g 存储, 可以采用 Floyd 算法求出图中每对顶点之间的最短路径长度数组 D , 再累加每行的元素之和存放至数组 B 中, 其中 $B[i]$ 表示顶点 i 到其他所有顶点的最短路径长度之和, 最后求出 B 中的最小元素 $B[\text{minv}]$, 而 minv 即为所求的村庄序号。据此, 我们可求得最小元素为 $B[2] = B[3] = 29$, 即医院应建在村庄 2 或村庄 3。

第7章 查找

7.1 顺序查找

查找表是由同一类型的数据元素或记录构成的集合。若查找表只支持查询某个特定元素是否存在以及检索某个特定元素的各种属性，则称该查找表为**静态查找表**；若在查找过程中同时插入查询表中不存在的数据元素，或从查找表中删除已存在的某个数据元素，则称该查找表为**动态查找表**。

由于查找的主要运算是关键字的比较，故通常将查找过程中和给定值进行比较的关键字个数的平均比较次数（也称为**平均查找长度**）作为衡量一个查找算法效率优劣的标准。平均查找长度 ASL 定义为

$$ASL = \sum_{i=1}^n p_i c_i$$

其中 n 为查找表中元素的个数， p_i 为查找表第 i 个元素的概率（一般地，除非特别指出，我们认为每个元素的查找概率均相等，即 $p_i = \frac{1}{n}$ ）， c_i 为找到第 i 个元素所需进行的比较次数。

本节往后涉及到的关键字类型和数据元素类型统一声明如下：

```
1 //关键字类型声明
2 typedef int KeyType;
3 typedef char KeyType;
4 typedef float KeyType;
5 //数据元素类型声明
6 typedef struct RecType
7 {
8     KeyType key; //关键字码
9     otherinfo; //其他域
10 }RecType;
11 //对数值型关键字的比较约定为如下的宏定义
12 #define EQ(a,b) ((a)==(b))
13 #define LT(a,b) ((a)<(b))
14 #define LQ(a,b) ((a)<=(b))
15 //对字符串型关键字的比较约定为如下的宏定义
16 #define EQ(a,b) (!strcmp((a),(b)))
17 #define LT(a,b) (strcmp((a),(b))<0)
18 #define LQ(a,b) (strcmp((a),(b))<=0)
```

静态查找表采用顺序存储结构，其数据元素类型定义如下：

```
1 typedef struct
2 {
3     ElemType *elem; //数据元素存储空间基址,建表时按实际长度分配,0号单元为空
4     int length; //表长
5 }SSTable;
```

顺序查找又称为线性查找，是一种最简单的查找方法：从表的一端开始顺序扫描顺序表，依次将扫描到的元素和给定值 k 比较，若相等则查找成功，而若直到扫描结束后仍未找到关键字与 k 相等的元素则查找失败。顺序查找算法如下：

```

1  int SeqSearch(SSTable ST,KeyType k) {
2      int i=0;
3      while(i<ST.length&&ST.elem[i]!=k) i++;
4      if(i==ST.length) return 0;//若未找到则返回0
5      else return i+1;//若找到则返回逻辑序号加1
6  }

```

对于含有 n 个元素的顺序表，设元素查找成功是等概率的，则易得第 $1 \leq i \leq n$ 个元素查找成功需要比较 i 次，即有 $c_i = i$ ，故查找成功时的平均查找长度为

$$ASL_{succ} = \sum_{i=1}^n p_i c_i = \sum_{i=1}^n \frac{1}{n} \cdot i = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2} = O(n)$$

而任何一次不成功的查找都需要和顺序表中 n 个元素都比较一次，故有 $ASL_{unsucc} = n$ 。

我们对上述顺序查找算法进行改进，省略对 i 是否越界的判断，具体代码如下：

```

1  int Search_Seq(SSTable ST,KeyType key) {
2      int i;
3      ST.elem[0].key=key;//哨兵
4      for(i=ST.length;!EQ(ST.elem[i].key,key);i--);
5      return i;
6  }

```

7.2 二分查找

二分查找要求顺序表中的元素是有序的，一般假设为升序，其基本思路易于理解：设 $\text{elem}[\text{low}..\text{high}]$ 是当前的查找区间，首先确定该区间的中点位置 $\text{mid} = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor$ ，再将待查的 key 值与 $\text{elem}[\text{mid}].\text{key}$ 值比较，则可得到三种结果，即

1. 若 $\text{elem}[\text{mid}].\text{key} = \text{key}$ ，则表示查找成功，并返回该元素的逻辑序号；
2. 若 $\text{elem}[\text{mid}].\text{key} > \text{key}$ ，则继续在 $\text{elem}[\text{low}..\text{mid}-1]$ 中查找；
3. 若 $\text{elem}[\text{mid}].\text{key} < \text{key}$ ，则继续在 $\text{elem}[\text{mid}+1..\text{high}]$ 中查找。

二分查找算法代码如下：

```

1  int Search_Bin(SSTable ST,KeyType key) {
2      int low=1,high=ST.length,mid;
3      while(low<=high) {
4          mid=(low+high)/2;
5          if(EQ(key,ST.elem[mid].key)) return mid;//找到待查值直接返回
6          else {
7              if(LT(key,ST.elem[mid].key)) high=mid-1;//继续在前半区间进行查找
8              else low=mid+1;//继续在后半区间进行查找
9          }
10     }
11     return 0;//顺序表中不存在待查元素
12 }

```

二分查找过程构成一棵判定树，其将当前查找区间的中间位置上的数据元素作为根，并将左子表和右子表的数据元素分别作为根的左子树和右子树。当二分查找中元素数量 n 较大时，可将整棵判定树近似视为满二叉树，其中所有的叶子结点位于同一层。若不考虑外部结点，则该二叉树的高度为 $h = \lceil \log_2(n+1) \rceil$ ，故有

$$ASL_{succ} = \sum_{i=1}^n p_i c_i = \frac{1}{n} \sum_{i=1}^n i \cdot 2^{i-1} = \frac{n+1}{n} \log_2(n+1) \approx \log_2 n$$

而对于任何一次不成功的查找都有 $ASL_{unsucc} = h = \lceil \log_2(n+1) \rceil$ 。

7.3 索引查找

一般地，索引存储结构需要在数据表基础上建立一个关于索引项的索引表，其结构为“(索引关键字, 该关键字记录在数据表中的相对地址)”，且索引关键字项有序排列。对无序的数据表建立索引表后，我们可以先在索引表中进行二分查找，得到待查关键字的相对地址，而后直接在数据表中根据此地址查找到该关键字。

若数据表可以分成若干块，每一块中的元素是无序的，但块与块之间的元素是有序的，即前一块中的最大（最小）关键字小于后一块的最小（最大）关键字，则可以采用**分块查找**的方式进行：索引表中的一项对应数据表中的一块，索引项由关键字域和链域组成，前者存放相应块的最大关键字，链域存放指向本块第一个元素的指针，而整个索引表按关键字值递增（递减）排列。由该过程不难看出，分块查找可分为两步：先对索引表进行二分查找，后在相应块中进行顺序查找。

若使用索引顺序查找的方法，当查找成功时，其平均查找长度为

$$ASL_{bs} = ASL_b + ASL_w$$

其中 ASL_b 为在索引表中查找块位置时的平均查找长度， ASL_w 为在块内查找元素位置时的平均查找长度。具体地，设将长度为 n 的表分为均等的 b 个块，每个块含有 s 个元素，则有 $b = \lceil \frac{n}{s} \rceil$ ；再设表中每个元素的查找概率相等，即每个块的查找概率为 $\frac{1}{b}$ ，块内各元素的查找概率为 $\frac{1}{s}$ 。若对索引表和块进行顺序查找，则查找成功时的平均查找长度为

$$ASL_{bs} = \frac{b+1}{2} + \frac{s+1}{2} = \frac{b+s}{2} + 1 = \frac{\frac{n}{s} + s}{2} + 1$$

易得当 $s = \sqrt{n}$ 时， ASL_{bs} 取极小值 $\sqrt{n} + 1$ 。

若对索引表进行二分查找，则查找成功时的平均查找长度为

$$ASL_{bs} = ASL_b + ASL_w \approx \log_2(b+1) - 1 + \frac{s+1}{2} \approx \log_2\left(\frac{n}{s} + 1\right) + \frac{s-1}{2}$$

综上两点，可以推出“分块查找的性能介于顺序查找与二分查找之间”的结论。

7.4 二叉排序树

二叉排序树又称为**二叉搜索树**，简称**BST**，其或为一棵空树，或为具有以下特性的二叉树：

1. 若左子树非空，则左子树上的所有结点的值均小于根结点的值；
2. 若右子树非空，则右子树上的所有结点的值均大于根结点的值；
3. 左右子树均为二叉排序树。

由定义可以看出，在二叉排序树中，关键字最小的结点处于根结点左子树最左下的位置，而关键字最大的结点处于根结点右子树最右下的位置。除此以外，我们不难发现，二叉排序树的中序序列是一个递增有序序列，这是一个非常重要的结论。

鉴于二叉排序树本质上也是一棵二叉树，故同样可以采用第五章中的二叉链表对其进行存储，其结点类型声明如下：

```

1  typedef struct
2  {
3      RecType data;
4      BiTNode *lchild,*rchild;
5  }BiTNode,*BiTree;

```

在二叉排序树中有以下几种基本操作。

1. 查找算法：在二叉排序树 T 中查找关键字为 key 的结点，若找到则返回该结点的指针，否则返回 $NULL$ 。

```

1  BiTree SearchBST(BiTree T,KeyType key)
2  {
3      if(!T||EQ(key,T->data.key)) return T;
4      else
5      {
6          if(LT(key,T->data.key)) return SearchBST(T->lchild,key);
7          else return SearchBST(T->rchild,key);
8      }
9  }

```

2. 插入结点算法：若待查关键字不存在于二叉排序树的任何结点中时，则需将包含该关键字的结点插入树中。

```

1  int SearchBST(BiTree &T,KeyType key,BiTree f,BiTree &p) {//改进算法
2      //若查找成功,则令指针p指向该结点,并返回1
3      //若查找失败,则令指针p指向查找路径上的最后一个结点,并返回0
4      //指针f指向T的双亲结点,其初始调用值为NULL
5      if(!T) {p=f;return 0;}
6      else {
7          if(EQ(key,T->data.key)) {p=T;return 1;}
8          else {
9              if(LT(key,T->data.key)) return SearchBST(T->lchild,key,T,p);
10             else return SearchBST(T->rchild,key,T,p);
11         }
12     }
13 }
14 int InsertBST(BiTree &T,RecType e) {
15     BiTree p,s;
16     if(!SearchBST(T,e.key,NULL,p)) {//查找不成功
17         s=(BiTree)malloc(sizeof(BiTNode));
18         s->data=e;s->lchild=s->rchild=NULL;
19         if(!p) T=s;//空树,此时s即为根结点
20         else {
21             if(LT(key,p->data.key)) p->lchild=s;
22             else p->rchild=s;
23         }
24         return 1;
25     }
26     else return 0;//树中已有关键字相同的结点,不再插入
27 }

```


3. 创建二叉排序树算法：由包含 n 个关键字的数组 $a[]$ 建立相应的二叉排序树，依次扫描数组 $a[]$ 的所有元素，并调用 $\text{InsertBST}()$ 函数将其插入到二叉排序树 T 中。

```

1 void CreateBST(BiTreeNode *&T,KeyType a[],int n)
2 {
3     T=NULL;
4     for(int i=0;i<n;i++) InsertBST(T,a[i]);
5 }

```

4. 删除结点算法：删除指针 p 指向的结点需要分三种情况讨论。

```

1 void Delete(BiTree &T)
2 {
3     BiTree q,s;
4     if(!p->rchild) {q=p;p=p->rchild;free(q);} //p的右子树为空,只需重接其左子树
5     else if(!p->lchild) {q=p;p=p->rchild;free(q);} //p的左子树为空,只需重接其右子树
6     else //p的左右子树均不为空
7     {
8         q=p;
9         s=p->lchild; //寻找p的左子树中的最大关键字结点,即待删结点的前驱结点
10        while(s->rchild) {q=s;s=s->rchild;} //用被删结点的前驱结点的值取代其值
11        p->data=s->data;
12        if(q!=p) q->rchild=s->lchild; //重接q的右子树
13        else q->lchild=s->lchild; //重接q的左子树
14        free(s);
15    }
16 }
17 int DeleteBST(BiTree &T,KeyType key) {
18     if(!T) return 0; //不存在关键字等于key的数据元素
19     else {
20         if(EQ(key,T->data.key)) Delete(T);
21         else {
22             if(LT(key,T->data.key)) DeleteBST(T->lchild,key);
23             else DeleteBST(T->rchild,key);
24         }
25         return 1;
26     }
27 }

```

含有 n 个结点的二叉排序树的平均查找长度与树的形态有关：当二叉排序树为单支树时，意味着先后插入的关键字有序，其平均查找长度为 $\frac{n+1}{2}$ ；当二叉排序树与二分查找的判定树形态相似时，其平均长度与 $\log_2 n$ 成正比。在随机情况下，二叉排序树的平均查找长度与 $\log_2 n$ 等数量级，但其出现的概率约为 53.5%。

7.5 AVL 树

平衡二叉树简称为 **AVL 树**。一棵 AVL 树或为一棵空树，或为具有以下特性的二叉树：

1. 任何一个结点的左子树与右子树高度之差的绝对值不超过 1；

2. 左右子树均为 AVL 树。

对于 AVL 树中的每个结点，我们设置一个平衡因子 (bf) 域，存放该结点左子树的高度与右子树的高度之差，取值范围为 $\{-1, 0, 1\}$ 。换言之，若一棵二叉树中所有结点的平衡因子的绝对值均小于等于 1，则该二叉树即为 AVL 树。

由于含有 n 个结点的 AVL 树中任何结点的左右子树的深度之差不超过 1，故其深度与 $\log_2 n$ 等数量级，据此可推出其平均查找长度与 $\log_2 n$ 等数量级。

若在一棵 AVL 树中插入一个新结点造成了不平衡状态，则必须调整树的结构使之重新平衡，对此通常有两种解决方案：一种是单旋转，包括 LL 旋转和 RR 旋转；另一种是双旋转，包括 LR 旋转和 RL 旋转。在插入一个新结点后，为保证正确性，我们需要从插入位置沿通向根结点的路径回溯，检查各结点的平衡因子是否满足 AVL 树的条件。

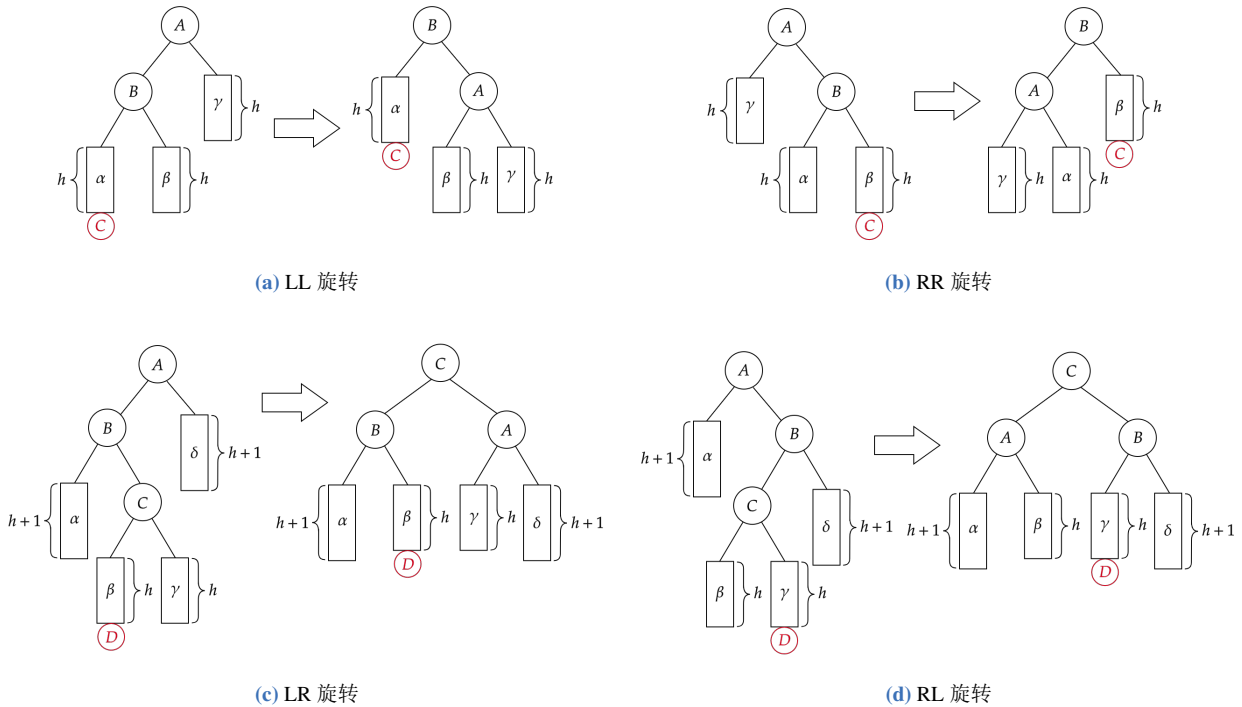


图 7.1: AVL 树结构的调整

在 AVL 树中进行查找的过程与在二叉排序树中进行查找的过程完全相同。为求 AVL 树的平均查找长度，我们构造一系列 AVL 树 T_1, T_2, \dots ，其中 T_h ($h = 1, 2, \dots$) 是高度为 h 且结点数最少的 AVL 树，容易推得 T_h 是由 T_{h-1} 和 T_{h-2} 分别作为其根的左右子树构造而来的。设 $N(h)$ 为 T_h 的结点数，则有以下关系成立：

$$N(h) = \begin{cases} 1, & h = 1 \\ 2, & h = 2 \\ N(h-1) + N(h-2) + 1, & h = 3, 4, \dots \end{cases}$$

不难发现该关系近似于斐波那契数列，比较二者可得 $N(h) = F(h+2) - 1$ 。由于斐波那契数渐进公式

$$F(h) = \frac{1}{\sqrt{5}} \varphi^h$$

其中 $\varphi = \frac{1+\sqrt{5}}{2}$ ，故有 $N(h) = \frac{1}{\sqrt{5}} \varphi^{h+2} - 1 \approx 2^h - 1$ ，即 $h \approx \log_2(N(h) + 1)$ 。

至此，我们可以得出结论：含有 n 个结点的 AVL 树的平均查找长度为 $O(\log_2 n)$ 。

7.6 B-树和B+树

7.6.1 B-树

B-树中所有结点的孩子结点最大值称为**B-树**的阶，用 m 表示，通常我们从查找效率考虑会要求 $m \geq 3$ 。一棵 m 阶 **B-树** 或为一棵空树，或为具有满足以下要求的 m 叉树：

1. 树中每个结点至多有 m 个孩子结点；
2. 若根结点不是叶子结点，则至少有两棵子树；
3. 除根结点外，所有内部结点至少有 $\lceil \frac{m}{2} \rceil$ 棵子树；
4. 树的所有外部结点都在同一层上，不带任何信息，且算入树的高度中；
5. 所有的内部结点均表示为

n	P_0	K_1	P_1	K_2	P_2	\cdots	K_n	P_n
-----	-------	-------	-------	-------	-------	----------	-------	-------

其中 K_i 为关键字，且 $K_i < K_{i+1}$ ； P_i 为指向子树根结点的指针，且指针 P_{i-1} 所指子树中所有结点的关键字均小于 K_i ， P_i 所指子树中所有结点的关键字均大于 K_i ($i = 1, 2, \dots, n$)； n 为该结点中的关键字个数，且满足 $\lceil \frac{m}{2} \rceil - 1 \leq n \leq m - 1$ 。

在 **B-树** 中查找给定关键字的方法与在二叉排序树类似，不同之处在于每个含 n 个关键字的结点确定的向下查找的路径是 $n + 1$ 路的，并且由于结点内的关键字序列是有序的，可以采用二分的方式进行查找。

当向 m 阶 **B-树** 中插入关键字 k 时，我们需要从根结点开始比较，类似于查找过程，寻找到合适的叶子结点 x 并将 k 插入之。特别注意的是，若 x 的关键字个数小于 $m - 1$ ，则直接有序插入 k ，否则进行分裂结点操作，即令含有中间位置关键字 K_s 的结点并入双亲结点（若不满足 **B-树** 定义则继续分裂），含有关键字 K_1, \dots, K_{s-1} 的结点成为其左孩子（局部），含有关键字 K_{s+1}, \dots, K_m 的结点成为其右孩子（局部）。

当给定一个关键字序列以构建一棵 m 阶的 **B-树** 时，我们需要从一棵空树开始，扫描所有的关键字，并采用上述 **B-树** 插入方式完成每一个关键字的插入。

当在 m 阶 **B-树** 中删除关键字 k 时，我们需要从根结点开始比较，类似于查找过程，寻找到包含 k 的结点 x 并将 k 删除。特别注意的是，结点 x 可能为非叶子结点，也可能为叶子结点，由此导致的删除方法有所不同。

1. 若 x 为非叶子结点，其中某个关键字 $K_i = k$ ，则以指针 P_i 所指子树中的最小关键字 $\min K$ 代替 K_i ($\min K$ 所在结点一定是叶子结点)，并在相应的叶子结点中删除 $\min K$ 。
2. 若 x 为叶子结点，则又可分为三种情况：
 - (a). 若从 x 中删除关键字 k 后，其关键字个数仍大于等于 $\min = \lceil \frac{m}{2} \rceil - 1$ ，则直接删除 k 即可；
 - (b). 若从 x 中删除关键字 k 后，其关键字个数小于 \min ，但其右兄弟（左兄弟）结点中的关键字个数大于 \min ，则删去 k 后从双亲结点处引入后者的最小（最大）关键字，并从兄弟结点处向双亲结点引入前者的最小（最大）关键字；
 - (c). 若从 x 中删除关键字 k 后，其关键字个数小于 \min ，且其两个相邻的兄弟结点的关键字个数都为 \min 个，则将 x 和其双亲结点中的一个关键字共同合并至其兄弟结点中。

注 若合并后双亲结点的关键字个数仍小于 \min ，则继续进行合并，直到根结点为止。

7.6.2 B+树

B+树 是 **B-树** 的一种变形，其形态近似于索引分块查找结构。一棵 m 阶 **B+树** 满足下列条件：

1. 每个分支结点至多有 m 棵子树；
2. 根结点或没有子树，或至少有两棵子树；
3. 除根结点外，其他每个分支结点至少有 $\lceil \frac{m}{2} \rceil$ 棵子树；

4. 拥有 n 棵子树的结点含有 n 个关键字；
5. 所有叶子结点包含全部关键字及指向相应记录的指针，且叶子结点按关键字大小顺序进行连接，即可以将每个叶子结点视为一个基本索引块，其指针不再指向另一索引块，而是直接指向数据文件中的记录；
6. 所有分支结点（可视为索引的索引）中仅包含其各个子结点（即下级索引的索引块）中的最大关键字及指向子结点的指针。



笔记 m 阶 B+ 树与 m 阶 B- 树具有以下差异。

1. 在 B+ 树中，具有 n 个关键字的结点含有 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，所有非叶子结点仅起到索引的作用，所有叶子结点包含全部关键字，而在 B- 树中，叶子结点包含的关键字与其他结点包含的关键字是不重复的。
3. B+ 树支持随机查找和顺序查找，而 B- 树仅支持随机查找。

7.7 哈希表

前面介绍的查找方式都是建立在“比较”基础之上的，即查找效率依赖于查找过程中所进行的比较次数，而我们希望不经过任何比较，通过一次存取便能得到待查记录。为此，需要在记录的存储位置和关键字之间建立一个确定的关系 $h(key)$ ，我们称 $h(key)$ 为**哈希函数**或**散列函数**，据此思想建立的表称为**哈希表**或**散列表**。

哈希表是除顺序存储结构、链式存储结构和索引存储结构之外的一种存储结构。对于给定的 n 个关键字，我们通过一个映射函数 h 生成每个关键字 key 的存储地址 $h(key)$ ，进而构建出一个大小为 m 的存储空间，此处 m 为哈希表的**长度**。由此可看出，若要查询某个关键字，则可以通过哈希函数直接在哈希表中定位至其地址，并返回相关的值，故利用哈希表查询的时间复杂度为 $O(1)$ 。

不过，在使用哈希表时可能会出现如部分关键字经哈希函数映射后得到的地址超过哈希表的长度 m 的情况，可以通过对哈希函数对 m 进行取模运算得到合理的地址；然而，若遇到**冲突**的情况，即对于两个关键字（称为**同义词**） k_i 和 k_j ，有 $k_i \neq k_j$ 且 $h(k_i) = h(k_j)$ ，则应最大程度地减少冲突。需要指出的是，哈希函数是一种压缩映射，冲突的出现不可避免，只能尽可能减少。

通常，在设计哈希表时需要考虑以下几点：

1. 哈希表的空间范围，即哈希函数的值域；
2. 合适的哈希函数，即对于所有可能的元素，其哈希函数值均在哈希表的地址空间范围内，且冲突出现的可能性尽可能最小；
3. 处理冲突的方法。

一个优秀的哈希函数能够使得到的哈希地址尽可能均匀地分布在 m 个存储单元中，同时使计算过程尽可能简单。根据关键字的结构和分布的不同，哈希函数有多种构造方法。

1. 以关键字 key 本身或关键字加上某个常数 c 为哈希地址的**直接定址法**，即 $h(key) = key + c$ 。该方法简单便捷，且不可能出现冲突，但不适用于关键字分布不连续的情况。
2. 以关键字 key 除以某个常数 $p \leq m$ 所得余数为哈希地址的**除留余数法**，即 $h(key) = key \bmod p$ ，其中 p 以素数为佳。
3. 提取关键字中取值较均匀的数字位作为哈希地址的**数字分析法**，适用于所有关键字值均已知情况，并需要对关键字中每位数的取值分布情况进行分析。

在哈希表中，冲突发生的可能性除了与所采用的哈希函数和解决冲突的方式有关外，还与**装填因子**有关。装填因子 α 是指哈希表中已存入的元素数 n 与哈希地址空间大小 m 的比值，即 $\alpha = \frac{n}{m}$ 。 α 越小，冲突发生的可能性就越小，但存储空间的利用率也会越低。通常，我们会采用下列方法降低冲突发生的可能性。

1. **开放定址法**：当冲突发生时，该方法形成某个探测序列，并按此序列逐个探测哈希表中的其他地址，直到找到给定的关键字或一个空地址为止，将发生冲突的记录存放至此地址中。哈希地址的计算公式为

$$H_i(key) = (H(key) + d_i) \bmod m, \quad i = 1, 2, \dots, k, k \leq m - 1$$

其中 $H(key)$ 为哈希函数, m 为哈希表长度, d_i 为第 i 次探测时的增量序列, $H_i(key)$ 为经第 i 次探测后得到的哈希地址。根据 d_i 取值的不同, 开放定址法又可分为线性探测法和二次探测再散列两种方法。

- (a). 线性探测法: 从发生冲突的地址 d 开始, 依次循环探测 d 的下一个地址直到找到一个空闲单元为止, 即

$$H_i(key) = (H(key) + d_i) \bmod m, \quad d_i = 1, 2, \dots, m-1$$

设 $d_0 = H(key)$, 则有 $d_i = (d_{i-1} + 1) \bmod m$ ($1 \leq i \leq m-1$)。该方法的优点在于只要哈希表未滿, 总能找到一个不冲突的哈希地址, 而缺点在于可能会出现二次聚集现象, 即每个产生冲突的记录会被散列到离冲突发生最近的地址中, 从而产生了更多的冲突机会。

- (b). 二次探测再散列: 该方法在线性探测法的基础上进行了改进, 会在冲突发生地址的前后处寻找空地址, 即

$$H_i(key) = (H(key) + d_i) \bmod m, \quad d_i = 1^2, -1^2, 2^2, -2^2, \dots, \pm k^2, k \leq \left\lfloor \frac{m}{2} \right\rfloor$$

设 $d_0 = H(key)$, 则有 $d_i = (d_0 \pm i^2) \bmod m$ ($1 \leq i \leq m-1$)。该方法可以避免二次聚集问题的出现, 缺点在于不能探测到哈希表上的所有单元, 但可以探测至少一半的单元。

2. 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中, 并用一维数组存放链表的头指针。在此方法中, 哈希表每个单元存放的不是记录, 而是相应同义词单链表的头指针。由于单链表可以插入多个结点, 故此时装填因子 α 可根据同义词的数量任意设置, 通常取 $\alpha = 1$ 。

例题 7.1 将关键字序列 (7,8,30,11,18,9,14) 散列存储至哈希表中, 该表的存储空间为一个下标从 0 开始的一维数组, 哈希函数为 $H(key) = (key \times 3) \bmod 7$, 处理冲突的方法为线性探测再散列法, 要求装填因子为 0.7。

1. 请画出所构造的哈希表。

2. 分别计算等概率情况下, 查找成功和查找不成功的平均查找长度。

解 由题意有 $n = 7$, $\alpha = 0.7$, 故哈希表长度为 $m = \frac{n}{\alpha} = 10$; 根据给定的哈希函数 $H(key) = (key \times 3) \bmod 7$ 有 $H(7) = 7 \times 3 \bmod 7 = 0$, 同理 $H(8) = 3$, $H(30) = 6$, $H(11) = 5$ 。由于 $H(18) = 18 \times 3 \bmod 7 = 5 = H(11)$ 产生冲突, 故利用线性探测法有 $d_1 = (5 + 1) \bmod 10 = 6 = H(30)$, $d_2 = (6 + 1) \bmod 10 = 7$, 可得 $H(18) = 7$, 同理 $H(9) = 6$, $H(14) = 1$ 。综上画出的哈希表如下:

下标	0	1	2	3	4	5	6	7	8	9
关键字	7	14		8		11	30	18	9	
成功时的探测次数	1	2		1		1	1	3	3	
不成功时的探测次数	3	2	1	2	1	5	4	3	2	1

表 7.1: 例题 7.1 所求的哈希表

由该哈希表可得

$$\begin{cases} ASL_{succ} = \frac{1+2+1+1+1+3+3}{7} \approx 1.71 \\ ASL_{unsucc} = \frac{3+2+1+2+1+5+4}{7} \approx 2.57 \end{cases}$$

注 在求 ASL_{unsucc} 时, 由于题目规定模数为 7, 故根据哈希函数得到的哈希地址均小于 7。



笔记 求哈希表的平均查找长度的公式如下。

1. 顺序表结构: $ASL_{succ} = \frac{\text{所有成功探测次数之和}}{\text{关键字个数}}$, $ASL_{unsucc} = \frac{\text{所有不成功探测次数之和}}{\min\{\text{哈希函数模数}, \text{哈希表长度}\}}$ 。
2. 链表结构: $ASL_{succ} = \frac{\sum_i (\text{第 } i \text{ 层结点个数} \times i)}{\text{关键字个数}}$, $ASL_{unsucc} = \frac{\sum_i (\text{仅含有 } i \text{ 个结点的单链表数} \times i)}{\text{哈希表长度}}$ 。

第8章 内部排序

8.1 插入排序

在排序过程中，若整个数据表都是存放在内存中进行处理，排序时不涉及数据的内外存交换，则称之为**内部排序**，否则称为**外部排序**。在内部排序算法中，我们根据是否基于关键字的比较将其分为基于比较的排序算法和不基于比较的排序算法，前者包括插入排序、交换排序、选择排序、归并排序等算法，后者包括基数排序等算法。当待排序记录的关键字均不相同，排序的结果是唯一的，否则不一定唯一。若在待排序的表中存在多个关键字相同的记录，经排序后其之间的相对次序保持不变，则称该算法是**稳定的**，否则称为是**不稳定的**。

在本章中，除基数排序采用单链表外，我们以顺序表作为待排序数据的存储结构，并规定记录类型如下：

```
1  #define MAXSIZE 20
2  typedef int KeyType;
3  typedef struct
4  {
5      KeyType key; //关键字项
6      InfoType otherinfo; //其他数据项
7  }RedType; //记录类型
8  typedef struct
9  {
10     RedType r[MAXSIZE+1]; //r[0]置空或作为哨兵
11     int length;
12 }SqList; //顺序表类型
```

插入排序的基本思路是每一轮将一个待排序的记录按关键字值的大小插入到已经排序的部分文件中的适当位置上，直至全部插入完成。主要的插入排序算法包括直接插入排序、折半插入排序和希尔排序。

直接插入排序算法代码如下：

```
1  void InsertSort(RedType R[], int n)
2  {
3      int j;
4      RedType tmp;
5      for(int i=1; i<n; i++) //初始时R[0]即为有序区,从R[1]开始排序
6      {
7          if(R[i-1].key>R[i].key)
8          {
9              tmp=R[i]; //取出无序区的第一个元素R[i]
10             j=i-1; //在R[0..i-1]中寻找R[i]的插入位置
11             do
12             {
13                 R[j+1]=R[j]; //将关键字大于tmp.key的元素后移
14                 j--; //继续向前比较
15             }while(j>=0&&R[j].key>tmp.key);
16             R[j+1]=tmp; //在j+1处插入R[i]
17         }
18     }
19 }
```


设置了哨兵的直接插入排序算法代码如下：

```

1 void InsertSort(SqList &L) {
2     for(int i=2;i<=L.length;i++)
3         if(LT(L.r[i].key,L.r[i-1].key)) {
4             L.r[0]=L.r[i];//令L.r[i]为哨兵
5             for(int j=i-1;LT(L.r[0].key,L.r[j].key);j--)
6                 L.r[j+1]=L.r[j];//记录后移
7             L.r[j+1]=L.r[0];//插入到正确位置
8         }
9     }

```

该算法最好的情况为关键字在其中顺序有序，此时比较次数为 $\sum_{i=1}^{n-1} 1 = n-1$ ，移动次数为 0；相反地，最坏

的情况为关键字在其中逆序有序，此时比较次数为 $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ ，移动次数为 $\sum_{i=1}^{n-1} (i+2) = \frac{(n-1)(n+4)}{2}$ 。

经上述分析，可得直接插入排序的时间复杂度为 $O(n^2)$ 。由于仅使用了少量变量，该算法的空间复杂度为 $O(1)$ ，且性能稳定。

若利用有序区的有序性，将待插入关键字通过二分查找方式在有序区中找到其位置并插入，则称该方法为**二分插入排序**。在 $R[0..i-1]$ 中查找并插入 $R[i]$ 的位置，二分查找的平均比较次数为 $\log_2(i+1)$ ，平均移动元素次数为 $\frac{i}{2} + 2$ ，故平均时间复杂度为 $\sum_{i=1}^{n-1} \left(\log_2(i+1) + \frac{i}{2} + 2 \right) = O(n^2)$ 。与直接插入排序相比，二分插入排序会将插入位置后的所有元素集中移动，而在平均性能上也优于直接插入排序。

8.2 希尔排序

希尔排序又称为**缩小增量排序**，其基本思路如下。

1. 先选择一个小于 n 的整数 d_1 作为第一个增量，将表的所有记录分为 d_1 个组，所有距离为 d_1 的倍数的记录放在同一个组中，在各组内进行直接插入排序。
2. 再取第二个增量 $d_2 < d_1$ ，重复步骤 1，直至所有的所有的增量 $d_t = 1 < d_{t-1} < \dots < d_2 < d_1$ ，即所有记录放在同一组中进行直接插入排序为止。

取 $d_1 = \frac{n}{2}$ ， $d_{i+1} = \left\lfloor \frac{d_i}{2} \right\rfloor$ 时的希尔排序算法代码如下：

```

1 void ShellSort(RedType R[],int n) {
2     int j,d=n/2;
3     RedType tmp;//增量置初值
4     while(d>0) {
5         for(int i=d;i<n;i++) {
6             tmp=R[i],j=i-d;//对所有位置距离为d的记录组采用直接插入排序
7             while(j>=0&&tmp.key<R[j].key) R[j+d]=R[j],j-=d;
8             R[j+d]=tmp;
9         }
10        d/=2;//缩小增量
11    }
12 }

```

留有暂存单元的、只进行一轮的希尔排序算法代码如下：

```

1  void ShellSort(SqlList &L,int dk)
2  {
3      for(int i=dk+1;i<=L.length;i++)
4          if(LT(L.r[i].key,L.r[i-dk].key))
5              {
6                  L.r[0]=L.r[i];
7                  for(int j=i-dk;j>0&&LT(L.r[0].key,L.r[j].key);j-=dk)
8                      L.r[j+dk]=L.r[j];
9                  L.r[j+dk]=L.r[0];
10             }
11 }

```

希尔排序是不稳定的，其时间复杂度为 $O(n^{1.3})$ ，空间复杂度为 $O(1)$ 。在希尔排序的增量序列中，最后一个增量必定为 1，且除该增量外其他所有增量两两互质。

8.3 冒泡排序

冒泡排序的基本思路如下。

1. 设待排序记录序列中的记录个数为 n ，并最多作 $n-1$ 轮排序。
2. 在第 i 轮排序中由后向前，顺次两两比较 $r[j-1].key$ 和 $r[j].key$ ，其中 $j = n-1, n-2, \dots, i$ 。
3. 若 $r[j-1].key > r[j].key$ ，则交换二者。

我们可以如此描述冒泡排序的过程：将序列分为有序区与无序区，对无序区进行一次从后往前的冒泡排序（对每对相邻的关键字进行关键字的比较），这样可将无序区中关键字最小的记录排序至第一位；再将该记录直接插入有序区的末端，并不断重复上述过程直至整个序列有序。

冒泡排序算法代码如下：

```

1  void BubbleSort(RedType R[],int n)
2  {
3      RedType tmp;
4      for(int i=0;i<n-1;i++)
5          {
6              int exchange=0;
7              for(int j=n-1;j>i;j--)
8                  if(R[j].key<R[j-1].key)
9                      {
10                         tmp=R[j];
11                         R[j]=R[j-1];
12                         R[j-1]=tmp;
13                         exchange=1;
14                     }
15              if(exchange==0) return; //若本轮未发生元素交换则直接结束算法
16          }
17 }

```

冒泡排序是一种稳定的算法，在最好的情况下时间复杂度为 $O(n)$ ，在最坏的情况下时间复杂度为 $O(n^2)$ ，故其平均时间复杂度为 $O(n^2)$ ；由于仅使用了少量变量，故其空间复杂度为 $O(1)$ 。

8.4 快速排序

快速排序的基本思路如下。

1. 将序列中的某一个记录（通常取第一个记录）的关键字作为基准。
2. 关键字小于基准的记录均移动至该记录之前，关键字大于基准的记录均移动至该记录之后。
3. 对基准前后的两个子序列递归地重复步骤 1 和步骤 2。

快速排序算法代码如下：

```

1  int Partition(SqlList &L,int low,int high)//进行一轮快速排序
2  {
3      KeyType pivotkey;
4      L.r[0]=L.r[low];
5      pivotkey=L.r[low].key;
6      while(low<high)
7      {
8          while(low<high&&L.r[high].key>=pivotkey) high--;
9          L.r[low]=L.r[high];
10         while(low<high&&L.r[low].key<=pivotkey) low++;
11         L.r[high]=L.r[low];
12     }
13     L.r[low]=L.r[0];
14     return low;
15 }
16 void QSort(SqlList &L,int low,int high)//对子序列递归地进行快速排序
17 {
18     int pivotkey;
19     if(low<high)
20     {
21         pivotkey=Partition(L,low,high);
22         QSort(L,low,pivotkey-1);
23         QSort(L,pivotkey+1,high);
24     }
25 }
26 void QuickSort(SqlList &L)
27 {
28     QSort(L,1,L.length);
29 }
```

快速排序是一种不稳定的算法，在最好的情况下时间复杂度为 $O(n \log_2 n)$ ，在最坏的情况下时间复杂度为 $O(n^2)$ ，故其平均时间复杂度为 $O(n \log_2 n)$ ，而空间复杂度为 $O(\log_2 n)$ 。

8.5 简单选择排序

简单选择排序的基本思路如下。

1. 在当前无序区 $r[i..n]$ 中选择具有最小关键字的记录。
2. 若该记录不是无序区中的第一个记录，则交换二者顺序。
3. 在无序区中删除该记录，并在新无序区 $r[i+1..n]$ 中重复步骤 1 和步骤 2，直至无序区仅含一个记录。

简单选择排序算法代码如下：

```

1  int SelectMinKey(Sqlist L,int i)//返回在L.r[i..L.length]中key最小的记录序号
2  {
3      KeyType min=L.r[i].key;
4      int k=i;
5      for(int j=i+1;j<=L.length;j++)
6      {
7          if(L.r[j].key<min)
8          {
9              k=j;
10             min=L.r[j].key;
11         }
12     }
13     return k;
14 }
15 void SelectSort(Sqlist &L)
16 {
17     RedType tmp;
18     for(int i=1;i<L.length;i++)
19     {
20         int j=SelectMinKey(L,i);
21         if(i!=j) {tmp=L.r[i],L.r[i]=L.r[j],L.r[j]=tmp;}
22     }
23 }

```

简单选择排序是一种不稳定的算法，在最好的情况与最坏的情况下时间复杂度均为 $O(n^2)$ ，故其平均时间复杂度为 $O(n^2)$ ，而空间复杂度为 $O(1)$ 。

8.6 堆排序

堆是满足下列性质的记录 $\{r_1, r_2, \dots, r_n\}$ ：

$$\begin{cases} r_i \leq r_{2i} \\ r_i \leq r_{2i+1} \end{cases} \quad (\text{小根堆})$$

$$\begin{cases} r_i \geq r_{2i} \\ r_i \geq r_{2i+1} \end{cases} \quad (\text{大根堆})$$

设待排序记录为 $r[i..n]$ ，并将其视为一棵完全二叉树的顺序存储结构。若每个结点的关键字均大于等于其所有孩子结点的关键字，则称该记录为大根堆；若每个结点的关键字均小于等于其所有孩子结点的关键字，则称该记录为小根堆。本节采用大根堆以完成堆排序。

堆排序的关键是构造堆，此处采用筛选算法建堆，即对一棵左右子树均为堆的完全二叉树进行调整根结点操作，使之仍然为一个堆。堆排序过程如下。

1. 从最后一个分支结点（序号为 $n/2$ ）开始到根结点（序号为 1）通过多次调用筛选算法建堆。
2. 将 $r[1]$ 与当前无序区最后一个记录交换位置，并在无序区中删除该记录。
3. 重复步骤 1 和步骤 2 直至无序区仅含一个记录。

堆排序算法代码如下：

```

1  typedef SqList HeapType;
2  void HeapAdjust(HeapType &H,int s,int m) {
3      //已知H.r[s..m]中记录的关键字除H.r[s].key外均为大根堆,调整H.r[s].key使得H.r[s..m]成为一个大根堆
4      RedType rc=H.r[s];
5      for(int j=2*s;j<=m;j*=2) { //沿key较大的孩子结点向下筛选
6          if(j<m&&LT(H.r[j].key,H.r[j+1].key)) j++; //j为key较大的记录的下标
7          if(!LT(rc.key,H.r[j].key)) break; //rc应插入在位置s上
8          H.r[s]=H.r[j];s=j;
9      }
10     H.r[s]=rc; //插入
11 }
12 void HeapSort(HeapType &H) {
13     RedType tmp;
14     for(int i=H.length/2;i>0;i--) HeapAdjust(H,i,H.length); //将H.r[1..H.length]调整为大根堆
15     for(int i=H.length;i>1;i--) { //将堆顶记录与当前未经排序子序列H.r[1..i]中最后一个记录进行交换
16         tmp=H.r[1],H.r[1]=H.r[i],H.r[i]=tmp;
17         HeapAdjust(H,1,i-1); //将H.r[1..i-1]重新调整为大根堆
18     }
19 }

```

堆排序是一种不稳定的算法，在最好的情况与最坏的情况下时间复杂度均为 $O(n \log_2 n)$ ，故其平均时间复杂度为 $O(n \log_2 n)$ ，而空间复杂度为 $O(1)$ 。

8.7 归并排序

归并排序算法代码如下：

```

1  void Merge(RedType SR[],RedType TR[],int i,int m,int n) {
2      //将有序的SR[i..m]与SR[m+1..n]归并为有序的TR[i..n]
3      int j,k,l;
4      for(j=m+1,k=i;i<=m&&j<=n;k++) //将SR中的记录由小到大并入TR
5          if(LQ(SR[i].key,SR[j].key)) TR[k]=SR[i++];
6          else TR[k]=SR[j++];
7      if(i<=m) {for(l=0;l<=m-i;l++) TR[k+l]=SR[i+l];} //将剩余的SR[i..m]复制到TR
8      if(j<=n) {for(l=0;l<=n-j;l++) TR[k+l]=SR[j+l];} //将剩余的SR[j..n]复制到TR
9  }
10 void MSort(RedType SR[],RedType TR1[],int s,int t) {
11     RedType TR2[MAXSIZE+1];
12     if(s==t) TR1[s]=SR[s];
13     else {
14         int m=(s+t)/2;
15         MSort(SR,TR2,s,m);MSort(SR,TR2,m+1,t);Merge(TR2,TR1,s,m,t);
16     }
17 }
18 void MergeSort(SqList &L) {MSort(L.r,L.r,1,L.length);}

```

每一轮归并排序的时间复杂度为 $O(n)$ ，总共需要进行 $\lceil \log_2 n \rceil$ 轮，故对 n 个记录进行归并排序的时间复杂度为 $O(n \log_2 n)$ 。归并排序是一种稳定的算法，在最好的情况与最坏的情况下时间复杂度均为 $O(n \log_2 n)$ ，故其平均时间复杂度为 $O(n \log_2 n)$ ，而空间复杂度为 $O(n)$ 。

8.8 基数排序

一般情况下，假定有一包含 n 个元素的序列 $\{elem_0, elem_1, \dots, elem_{n-1}\}$ ，且每个元素 $elem_i$ 都含有 d 个关键字 $(K_i^1, K_i^2, \dots, K_i^d)$ ，若对于序列中任意两个元素 $elem_i$ 与 $elem_j$ ($0 \leq i < j \leq n-1$)，都满足

$$(K_i^1, K_i^2, \dots, K_i^d) < (K_j^1, K_j^2, \dots, K_j^d)$$

则称序列对关键字 $(K_i^1, K_i^2, \dots, K_i^d)$ 有序，其中 K^1 称为最高位关键字， K^d 称为最低位关键字。

实现多关键字排序有两种常用的方法，一种是最高位优先 (MSD)，一种是最低位优先 (LSD)。

1. MSD：通常是一个递归的过程。

- (a). 先根据最高位关键字 K^1 排序，得到若干元素组，元素组中各元素都有相同关键字 K^1 。
- (b). 再分别对每组中元素根据关键字 K^2 进行排序，按 K^2 值的不同再分成若干个更小的元组，每个子组中的元素具有相同的 K^1 和 K^2 值。
- (c). 重复步骤 (b) 直至对关键字 K^d 完成排序为止。
- (d). 最后将所有子组中的元素依次连接起来，便可得到排序完毕的序列。

2. LSD：与最高位优先过程基本相同，区别仅在于 LSD 从最低位关键字 K^d 开始排序。

LSD 对每个关键字进行排序时不需要进行分组，而是让整个元素组都参与排序。LSD 和 MSD 也可以将单个关键字 K_i 视为一个子关键字组 $(K_i^1, K_i^2, \dots, K_i^d)$ 进行排序。

基数排序是典型的 LSD 排序方法，其将单关键字 K_i 视为一个 d 元组 $(K_i^1, K_i^2, \dots, K_i^d)$ ，其中每个分量 K_i^j 可同样视为一个关键字。若分量 K_i^j 有 $radix$ 种取值，则称 $radix$ 为基数。例如，关键字“984”可视为一个三元组 (9, 8, 4)，其中每一位都有 0, 1, \dots , 9 等 10 种取值，则该关键字的基数为 10。

链式基数排序过程如下。

1. 针对 d 元组中的每一位分量，将元素序列中的所有元素按 K_i^j 的取值分配到 $radix$ 个队列中。
2. 按各队列的顺序，依次将元素从队列中收集，此时所有元素即按 K_i^j 排序完毕。
3. 若对于所有元素的关键字 $\{key_1, key_2, \dots, key_n\}$ ，依次对各位的分量，令 $j = d, d-1, \dots, 1$ ，重复步骤 1 和步骤 2 逐轮排序，则可将所有元素按关键字由小到大完成排序。

各队列采用链式存储结构，且设置两个队列指针：

```
1  int front[radix]; // 指示队头
2  int rear[radix];  // 指示队尾
```

若每个关键字有 d 位，需重复执行 d 轮排序分配与收集操作，每轮对 n 个元素进行分配，对 rd 个队列进行收集，则总时间复杂度为 $O(d(n+rd))$ 。基数排序是一种稳定的算法，在最好的情况与最坏的情况下时间复杂度均为 $O(d(n+rd))$ ，而空间复杂度为 $O(rd)$ 。

附录 A 代码说明

本笔记采用 C 语言与类 C 语言书写算法代码与 ADT 定义伪码，在此作简要说明。

```
1  #define TRUE      1
2  #define FALSE     0
3  #define INFEASIBLE -1
4  #define OVERFLOW  -2
5  typedef int Status; //Status是函数的类型,其值是函数结果状态代码
6  exit(异常代码); //异常结束语句
```