启航计算机考研专业课系列

计算机考研专业课——
数据结构一本通
(考点详解+习题全解)

李　红　刘财政　主编

清华大学出版社
北　京

内 容 简 介

本书严格按照全国硕士研究生入学考试计算机学科专业基础综合大纲进行编写，内容涵盖线性表、树和二叉树、图、查找、排序等大纲要求的知识点，并以图表和代码的形式对考点进行讲解，注释清晰易懂。本书精选历年的408考题及部分名校试题进行详细讲解，帮助考生学练结合，提高考生的学习效率。

本书可作为学生参加计算机专业硕士研究生入学考试的辅导用书，也可作为计算机及相关专业的学生学习“数据结构”的教材。

图书在版编目（CIP）数据

计算机考研专业课. 数据结构一本通：考点详解+习题全解 / 李红，刘财政主编. —北京：清华大学出版社，2019
（启航计算机考研专业课系列）
ISBN 978-7-302-52708-4

I. ①计… II. ①李… ②刘… III. ①电子计算机－研究生－入学考试－题解 ②数据结构－研究生－入学考试－题解 IV. ①TP3-44

中国版本图书馆 CIP 数据核字（2019）第 062358 号

责任编辑：袁金敏
封面设计：刘新新
责任校对：焦丽丽
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张：**22.25　　**字　数：**487 千字
版　　次：2019 年 7 月第 1 版　　**印　次：**2019 年 7 月第 1 次印刷
定　　价：69.00 元

产品编号：083244-01

前　言

“数据结构”是全国硕士研究生入学考试科目——计算机学科专业综合的考查科目之一，是研究“非数值计算的程序设计问题中计算机的操作对象及其关系和操作等的学科”，是介于数学、计算机硬件和计算机软件三者之间的一门核心课程，同时也是计算机科学与技术专业的专业基础课。课程内容不仅是一般程序设计（特别是非数值计算的程序设计）的基础，同时也是设计和实现编译程序、操作系统、数据库系统及其他系统程序和应用程序的重要基础。

“数据结构”主要研究数据及数据之间的关系、数据的存储以及具有关系的数据集上的操作。主要涉及三种关系：一对一关系（线性表）、一对多关系（树和二叉树）、多对多关系（图）。常见操作有创建、查找、删除、排序等。考生在复习时首先应熟练掌握理论内容，单纯的刷题无法解决基础内容的欠缺问题，也无法应对考场上不曾谋面的试题。然后再通过练习题及真题检验对课程知识的掌握程度。

本书分为两部分，第一部分是考点详解，第二部分为习题精讲。考点详解部分首先是导学部分，主要介绍本科目考试大纲和本书各章节知识点分布。第 1 章为绪论，主要介绍数据结构和算法的基本概念。第 2 章为线性表，主要介绍线性结构的基本概念、算法及实现，以及特殊线性表（栈和队列）、特殊矩阵等。第 3 章介绍树及二叉树的相关概念及算法。第 4 章介绍图的定义、算法。第 5 章介绍各种查找算法及其分析。第 6 章介绍排序算法及其特点。习题精讲部分主要收集全国硕士研究生入学考试计算机学科专业基础综合（专业课代码 408，本书文中统称 408）及各高校科研院所的历年真题，通过真题使考生零距离感受考题形式和答题思路，通过做题，熟练、灵活地掌握理论内容，进一步加强分析题目、求解问题的思维训练。

本书的知识详解部分尽可能多地给出基本操作的伪码实现、注释、算法的流程分析等，帮助考生深入理解算法本质，为解题打下坚实基础。此外，我们还为本书配备了丰富的视频讲解，扫描每章和图书封底的二维码即可观看。

备考过程中，考生需注意复习方法。首先应全局把握本书内容，熟悉本书特点、重点章节等。然后进行系统学习和总结，熟练掌握各知识点。最后通过历年真题分析巩固各知识点并了解各知识点的出题方式和考查频率。

备考过程漫长辛苦，考生应注意学习方法，提高复习效率，不搞消耗战，不做过多重复题，以不变应万变。毫无头绪时不妨归本还原，静下心来认真研究基本概念和算法，或许能打开解题思路。

编者

2019 年 1 月

目　录

第 0 章　导学

0.1　学 习 目 标

C 语言基础

要求考生比较系统地理解和掌握数据结构涉及的基本概念、原理和方法，能够综合运用所学原理和方法分析、判断、解决有关的理论问题和实际应用问题。

本书主要参考 408 考试大纲编写，学习目标具体如下。

（1）掌握数据结构的基本概念、基本原理和基本方法。

（2）掌握数据的逻辑结构、存储结构及基本操作的实现，能够对算法进行基本的时间复杂度与空间复杂度的分析。

（3）能够运用数据结构基本原理和方法进行问题的分析与求解，具备采用 C 或 C++ 语言设计与实现算法的能力。

0.2　大　　纲

常用算法思想

本书内容基于 408 考试大纲。

第 1 章绪论：基本概念，数据结构，算法。

第 2 章线性表：概述，线性表的存储，线性表的应用，栈、队列和数组，特殊矩阵的压缩存储。

第 3 章树与二叉树：树的基本概念，二叉树，树、森林，树与二叉树的应用。

第 4 章图：图的基本概念，图的存储及基本操作，图的遍历，最小（代价）生成树，关键路径。

第 5 章查找：查找的基本概念，顺序查找法，分块查找法，折半查找法，B 树及其基本操作、B+树的基本概念，散列（Hash）表，字符串模式匹配，查找算法的分析及应用。

第 6 章排序：排序的基本概念，插入排序，起泡排序，简单选择排序，希尔排序，快速排序，堆排序，二路归并排序，基数排序，外部排序，各种排序算法的比较，排序算法的应用。

0.3　本书知识结构

本书知识结构如图 0.1 所示。其中：加粗框中的内容需要重点理解并掌握。排序部

分仅列出重点内容——内部排序，其他见具体章节。

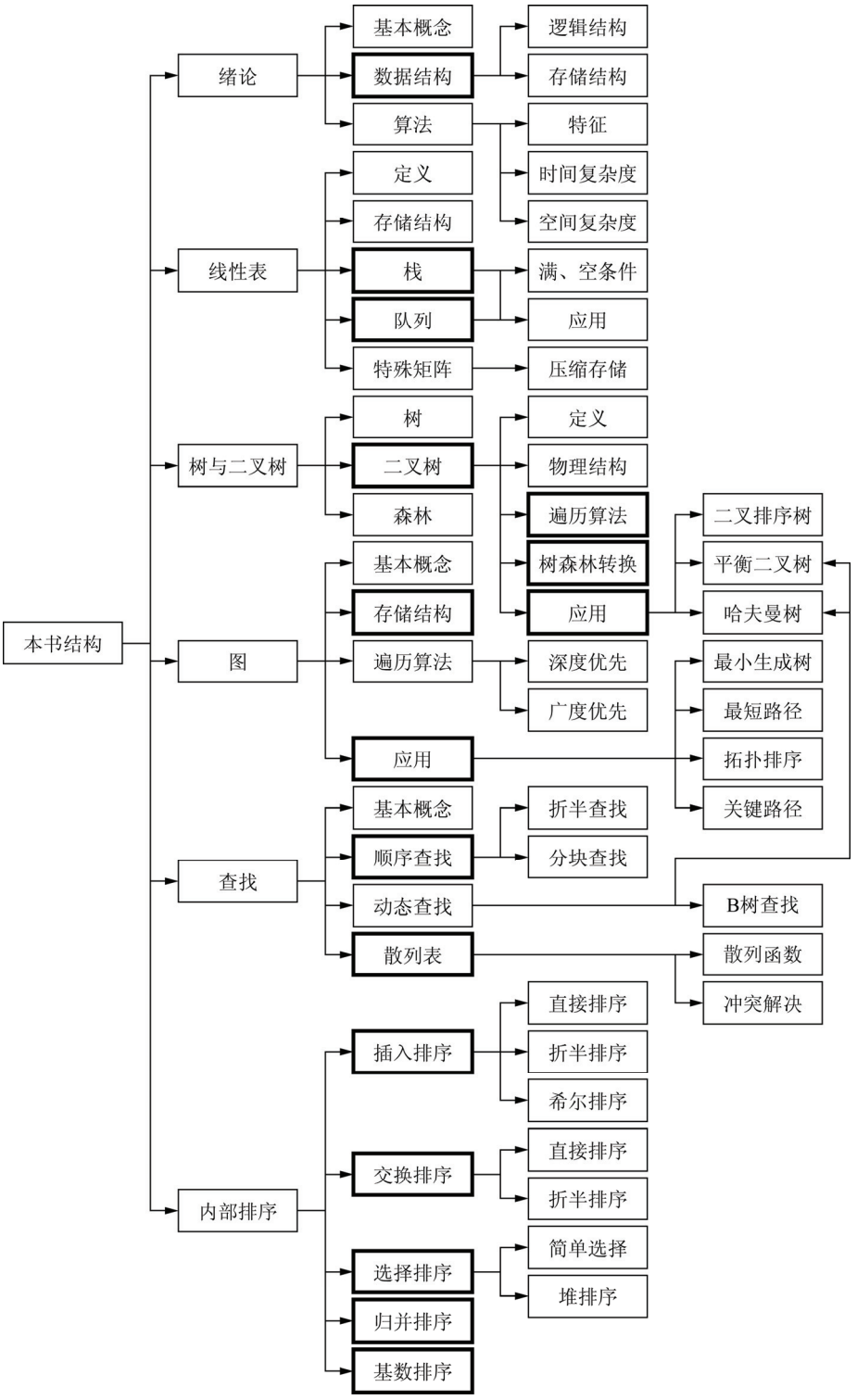


图 0.1　本书知识结构

第 1 章　绪论

本章学习目标

- 了解数据、数据元素、数据项等基本概念。
- 深入理解数据结构的含义、逻辑结构和存储结构的概念。
- 理解不同逻辑结构的本质区别。
- 掌握不同存储结构的实质。
- 理解算法的概念。
- 掌握算法的特征。
- 能够分析各种算法的时间复杂度和空间复杂度。

1.1　本 章 导 学

绪论

1.1.1　知识结构

本章知识结构如图 1.1 所示，加粗框中的内容需要重点理解并掌握。

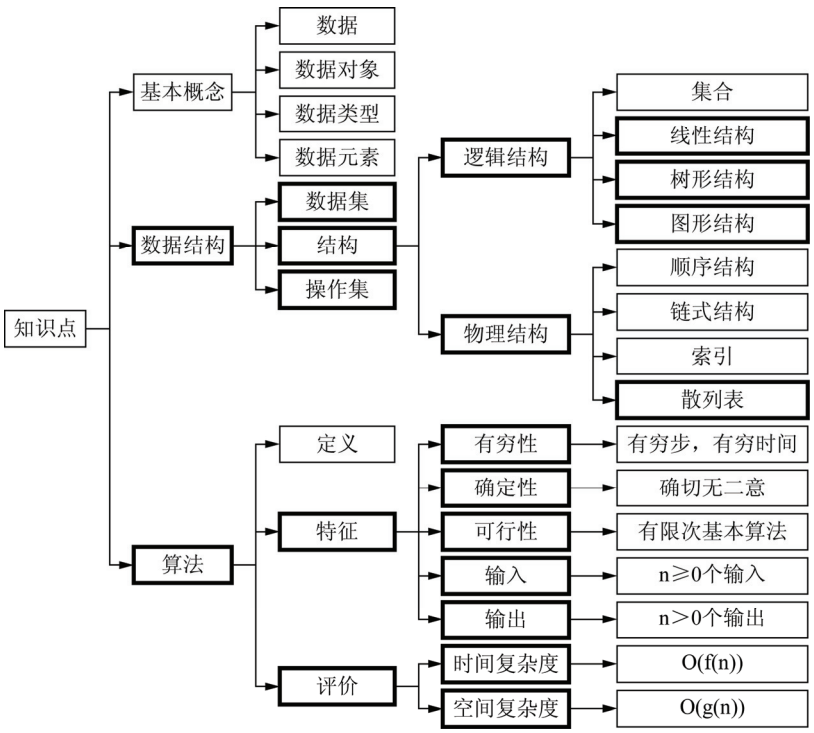


图 1.1　本章知识结构

1.1.2 命题特点

1. 命题规律

（1）本章是历年各高校硕士研究生招生考试的基本考查内容，命题形式既有客观题也有主观题。

（2）本章既可单独命题，也可与后续章节联合命题。

（3）顺序存储结构、链式存储结构的操作细节易出客观题。

（4）链表操作的综合应用易出主观题。

2. 考查趋势

本章在全国硕士研究生入学考试中的重要性近年不会改变，主观题型、客观题型出现的概率极大。特别注意顺序存储结构与查找、排序两章联合命题，以及链式存储结构的综合应用。

1.2 基 本 概 念

（1）数据：客观事物的符号表示，指所有能输入到计算机中并被计算机程序处理的符号。

（2）数据元素：又称结点，是数据的基本单位，在计算机中通常作为一个整体进行处理。该概念根据具体问题进行具体界定，外延可大可小。

（3）数据项：又称属性，指数据中具有独立意义的、不可分割的最小单位。

注意，数据由多个数据元素组成，一个数据元素又包含若干数据项。

（4）数据对象：性质相同的数据元素的集合。

（5）数据类型：一组性质相同的值集合以及定义在该集合上的一组操作的总称。

（6）抽象数据类型：Abstract Data Type，简称 ADT，指一个数学模型及定义在该模型上的一组操作。

（7）逻辑结构：数据元素之间固有的逻辑关系。该关系与计算机无关，是人的思维层面对现实世界数据之间关系的理解。

（8）存储结构：逻辑结构在计算机中的表示，也称物理结构，不同存储结构对数据处理效率具有较大影响。

（9）数据结构：相互之间具有特定关系的数据元素的集合，包含数据集、结构（逻辑结构、物理结构）及施加其上的操作集。

（10）算法：解决特定问题的有限指令序列。

1.3 数 据 结 构

1.3.1 定义

数据结构指相互之间存在一种或多种关系的数据元素的集合。数据元素不是孤立存在的，它们之间存在着这样或那样的关系，数据元素之间的关系称为结构，包含逻辑结构和存储结构两个层面，以及数据集上的一组操作。

其中，逻辑结构是数据结构的抽象，存储结构是数据结构的实现，两者综合起来建立数据元素之间的结构关系。

数据结构注重数据元素之间的相互关系与组织方式、运算及规则，不涉及数据元素的具体内容。

数据结构的形式化定义有如下三种常见形式。

（1）二元组：Data_Structures = (D, R)，其中，D 是数据元素的有限集，R 是 D 上关系的有限集。

【例 1-1】 有一种数据结构 A=(D,R)，其中：

D={1,2,3,4,5,6,7,8,9,10}，R={r}， R={<1,2>, <2,3>, <3,4>, <4,5>, <5,6> , <6,7>, <7,8>, <8,9>, <9,10>}。

（2）图形：用○表示一个元素，用横线或箭头连接两个○表示元素之间的关系，例如：

（3）抽象数据类型：用三元组(D, S, P)表示，其中 D 是数据对象的集合，S 是 D 上关系的集合，P 是 D 的基本操作集合，形式如下。

```
ADT 抽象数据类型名 {
    数据对象:〈数据对象的定义〉
    数据关系:〈数据关系的定义〉
    基本操作:〈基本操作的定义〉
} ADT 抽象数据类型名;
```

例如：抽象数据类型：矩形

① 矩形 ADT 的定义。

```
ADT Rectangle {
数据对象:  length;    // 非负实数，矩形的长
           width;     // 非负实数，矩形的宽
数据关系:  无
基本操作:
           init( &R, length，width );  // 将矩形 R 的长和宽分别初始化为 length 和
                                       // width
           area(R);                    // 返回矩形 R 的面积
           circumference(R);           // 返回矩形 R 的周长
} ADT Rectangle;
```

② 矩形 ADT 的表示。

```
// 定义矩形的存储结构
typedef struct
{
    float length;          // 矩形的长
    float width;           // 矩形的宽
} Rectangle;
// 定义矩形的基本操作
bool  init(Rectangle &R, float l, float w);
float  area(Rectangle R);
float  circumference(Rectangle R);
```

③ 矩形 ADT 的实现。

```
bool init(Rectangle &R, float l, float w)
{
    if( l>0 && w>0 ){
        R.length=l;
        R.width=w;
        return true;
    }
    else
        return false;
}
float area(Rectangle R)
{
    return R.length*R.width;
}
float circumference(Rectangle R)
{
    return 2*(R.length+R.width);
}
```

抽象数据类型是近年来计算机科学领域提出的最重要概念之一，集中体现了程序设计的一些最基本的原则，其特点具体如下。

- 数据抽象与信息隐藏。
  - 一个抽象数据类型确定了一个数学模型，并将模型的实现细节加以隐藏。
  - 定义了一组运算，并将运算的实现过程隐藏起来。
- 模块化。
- 继承性。
- 封装与复用。

1.3.2 逻辑结构

数据元素之间的逻辑关系，可以用一个数据元素的集合和定义在此集合上的若干关系来表示。根据数据元素之间逻辑关系的不同，数据元素之间的逻辑结构一般分为以下 4 种基本类型。

1. 集合

数据元素之间最松散的一种结构，仅具有属于同一集合这种关系，集合中无重复元

素，如图 1.2 所示。本书不讨论这种逻辑结构。

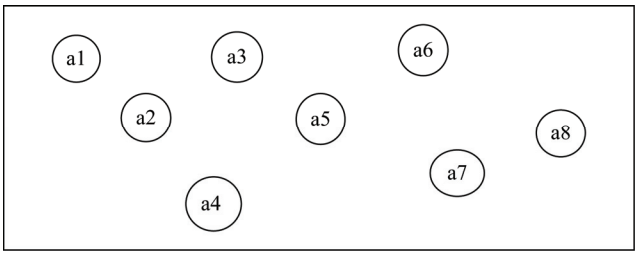


图 1.2　集合

2．线性结构

数据元素之间具有 1∶1 关系，是简单、常见的逻辑结构，如图 1.3 所示。

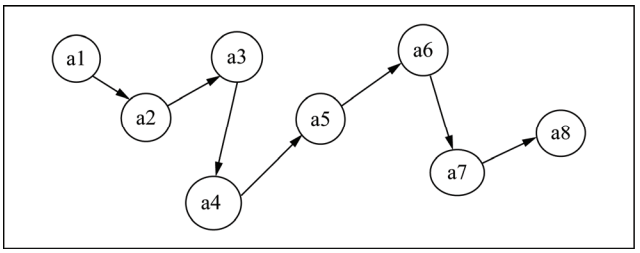


图 1.3　线性结构

3．树形结构

数据元素之间具有 1∶n 关系，为常见逻辑结构，如图 1.4 所示。

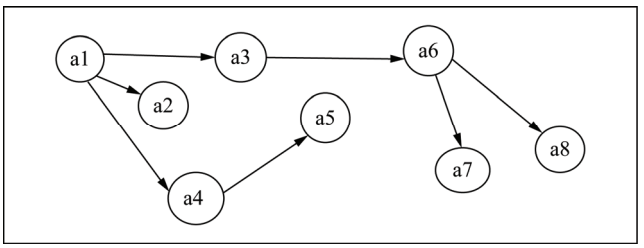


图 1.4　树形结构

4．图形结构

数据元素之间具有 m∶n 关系，为常见逻辑结构，如图 1.5 所示。

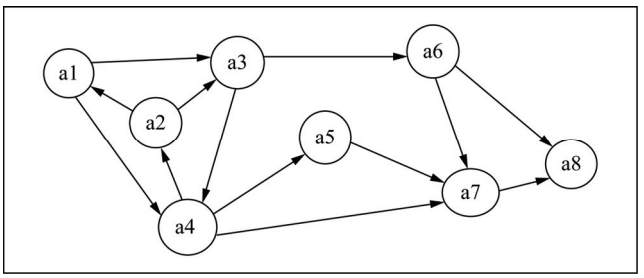


图 1.5　图形结构

线性结构为特殊树形结构，树形结构为特殊图形结构。

1.3.3 存储结构

存储结构是逻辑结构在计算机中的表示和实现。根据数据元素在计算机内部的存储方式不同，数据元素在内存中的物理结构一般分为以下 4 种基本类型。

1．顺序结构

逻辑结构中相邻的数据元素存储在计算机连续的内存单元中，即逻辑关系通过存储单元之间的相对位置表示。图 1.3 的线性结构对应的顺序存储结构如图 1.6 所示。

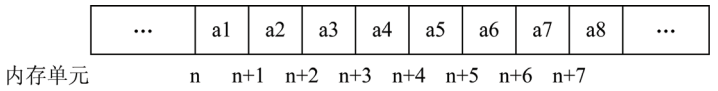


图 1.6　顺序存储结构

此结构的优点包括：

- 不需要额外空间存储逻辑关系，节省存储空间。
- 支持随机访问。

此结构的缺点包括：

- 为确保顺序存储方式的特性，进行数据元素的插入和删除操作需要移动部分甚至全部数据元素，改变其存储位置。
- 存储所有数据元素需要连续的存储空间。

2．链式结构

采用链式结构，逻辑结构中相邻的数据元素可以存储在计算机中不连续的内存单元中，逻辑关系通过数据元素中附加的指针域表示。图 1.3 的线性结构对应的链式存储结构如图 1.7 所示。

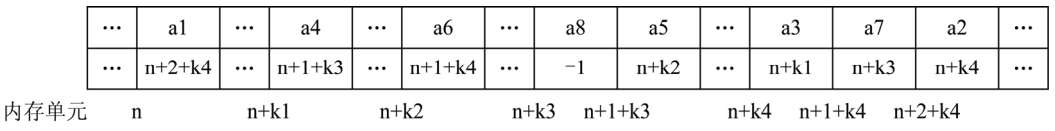


图 1.7　链式存储结构

head=n;　// head 为头指针，表示第一个元素的存储位置

此结构的优点包括：

- 进行数据元素的插入和删除操作仅需修改相关数据元素的指针域，无须移动其他数据元素，无须改变其存储位置。
- 修改数据效率高。
- 存储所有数据元素不需要连续的存储空间。

此结构的缺点包括：

- 需要额外空间——通过指针域存储逻辑关系，单个数据元素和整体数据的存储空

间增大。

- 不支持随机访问。

3．索引结构

索引结构通过索引表指示数据元素的存储位置，索引表由索引项构成。索引项的一般形式为（关键字，地址），其中关键字是能够唯一标识数据元素的数据项，地址表示数据元素在内存的存储位置。图 1.3 的线性结构对应的索引存储结构如图 1.8 所示，表 1.1 为相应的索引表，ai.key 为 ai 的关键字，i∈[1,8]。

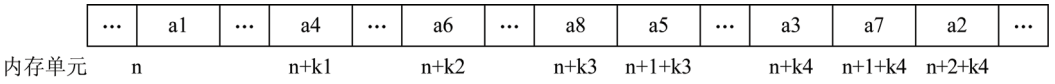


图 1.8　索引存储结构

表 1.1　索引表

关键字	地址	关键字	地址
a1.key	n	a5.key	n+1+k3
a2.key	n+2+k4	a6.key	n+ k2
a3.key	n+k4	a7.key	n+1+k4
a4.key	n+k1	a8.key	n+k3

此结构的优点包括：

- 进行数据元素的插入和删除操作仅需修改索引表中相关数据元素的存储地址，无须移动数据元素。
- 修改数据效率高。
- 支持随机访问。
- 存储所有数据元素不需要连续的存储空间。

此结构的缺点包括：

- 需要额外存储空间——通过索引表存储逻辑关系。
- 需要额外时间——对索引表进行维护。

4．散列结构

数据元素的存储单元地址由散列（Hash）函数根据其关键字计算得出，后续章节会有详细介绍。图 1.3 的线性结构对应的散列存储结构如图 1.9 所示，假设 ai.key 为 ai 的关键字，a1.key=5，a2.key=15，a3.key=9，a4.key=20，a5.key=6，a6.key=16，a7.key=18，a8.key=12，散列函数为 H(ai.key)= ai.key % 18，%为取余运算。则：

H(a1.key)= 5 % 18=5　　H(a2.key)= 15 % 18=15

H(a3.key)= 9 % 18=9　　H(a4.key)= 20 % 18=2

H(a5.key)= 6 % 18=6　　H(a6.key)= 16 % 18=16

H(a7.key)= 18 % 18=0　　H(a8.key)= 12 % 18=12

存储如下：

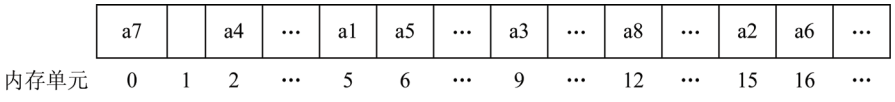


图 1.9　散列存储结构

此结构的优点包括：

- 进行数据元素的插入和删除操作仅需通过散列函数根据关键字计算该数据元素的存储位置，无须移动数据元素。
- 查找速度快。
- 存储所有数据元素不需要连续的存储空间。

此结构的缺点包括：

- 需要好的散列函数——数据元素计算得到的存储地址尽可能“散”，无重复。
- 计算得到的不同数据元素的存储地址难以完全避免冲突，所以需要有效的处理冲突的方法。

1.4　算　　法

1.4.1　定义

通俗讲，算法是一种解题方法，是解决某特定现实问题的一个过程或一种策略。

严格讲，算法是对特定问题求解步骤的一种描述，是一个有穷的规则集合，这些规则为解决某一特定任务规定了一个有序的运算序列。也可以说算法是指令的有限序列，其中每一条指令表示一个或多个操作。

计算机对数据的处理可分为数值数据处理和非数值数据处理两种，其中数值处理主要进行算术运算，非数值处理主要进行查找、排序、插入、删除等运算。

描述算法的主要方法有自然语言、程序设计语言（或类程序设计语言）、流程图（包括传统流程图和 N-S 结构图）、伪语言和 PAD 图。

1.4.2　特征

算法必须满足以下五个重要特性。

（1）有穷性：一个算法必须能在执行有穷步之后正常结束，且每一步都在有穷时间内完成，即算法必须在有限时间内完成，不能无穷循环。

（2）确定性：算法的每一步必须有确切的含义，无二义性，即输入相同数据必须得到同样的输出结果。

（3）可行性：算法描述的所有操作均可通过执行有限次已实现的基本运算实现。

（4）输入：一个算法应该具有 0 个或多个输入，这些输入取自某特定的数据对象集合。

（5）输出：一个算法应该具有一个或多个输出，这些输出和输入之间存在某种特定的关系。

1.4.3　算法和程序

算法是对特定问题求解步骤的一种描述，与所用计算机和所选编程语言无关；程序是算法在计算机中的实现，与所用计算机和所选编程语言有关。

程序=算法+数据结构。

程序不一定满足有穷性（如操作系统在用户未使用前一直处于踏步等待状态，直到新的用户事件出现，而算法必须满足有穷性。

程序中的指令必须是机器可执行的，不能有语法错，而算法则无此限制，算法可以通过程序表述，而程序不能用算法代替。

算法和程序都是表达解决问题方法的逻辑步骤，但算法独立于具体的计算机，与具体的编程语言无关，而程序正好相反。

程序是算法，但算法不一定是程序。

1.4.4　评价

一个好算法一般应该具有如下特点。

（1）正确性：对于任何输入数据都能得出满足要求的结果。

（2）可读性：易于阅读和理解。

（3）健壮性：对非法输入的恰当处理，使系统能在非法输入下给出适当反馈或正常退出。

（4）高效性：时间效率高，空间消耗少。

其中需要重点分析高效性，即在问题规模 n 下对算法运行时所花费的时间 T、所占用的存储空间 S 的评价。

1. 时间复杂度

（1）时间复杂度是算法对应程序在机器中运执行时所需的时间。

算法的执行时间=Σ算法中基本操作执行次数×该基本操作执行时间。

算法的时间复杂度计算相当烦琐，一般没必要精确地计算出算法的时间复杂度，只要大致计算出相应的数量级 (Order) 即可，T(n)=O(f(n))。

（2）时间复杂度的影响因素包括如下几点：

① 问题规模。

② 算法实现所用的编程语言。

③ 编译程序生成的目标代码质量。

④ 硬件速度。

（3）常见时间复杂度包括：

① 常量阶：T(n)=O(1)。

```
x=x+1;
```

② 线性阶：T(n)=O(n)。

```
for (i=1; i<= n; i++)
   x=x+1;
```

③ 平方阶：T(n)=O(n^2)。

```
for (i=1; i<= n; i++)
   for (j=1;j<= n; j++)
      x=x+1;
for (i=0; i<n-1; i++)
   for (j=i+1; j<n; j++)
      if (a[i] < a[j])
         m=a[i], a[i]=a[j], a[j]=m;
```

时间复杂度以最大语句频度 if 语句的频度 n^2 来估算，不考虑算法中其他语句频度：

$$(n-1)+(n-2)+\cdots+1=n(n-1)/2=(n^2-n)/2$$

$$T(n)= O(n^2)$$

④ 立方阶：T(n)=O(n^3)。

```
for (i=1; i<= n; i++)
   for (j=1; j<= n; j++)
      for (k=1; k<= n; k++)
         n++;
```

（4）不同时间复杂度的比较。

① 多项式级：O(1)<O(n)。

② 对数级：O($\log^n$)<O($n\log^n$)。

③ 平方立方级：O(n^2)<O(n^3)。

④ 指数级：O(2^n)<O(n^n)。

O(1)< O($\log^n$)<O(n)<O($n\log^n$)<O(n^2)<O(n^3)<O(2^n)<O(n^n)。

指数级的时间复杂度非常高，多需转换为多项式级时间复杂度。

2．空间复杂度

空间复杂度是算法对应程序在机器执行过程中所占用的临时存储空间，包含三部分：

（1）算法本身所占用的存储空间。

（2）输入数据所占用的存储空间。

（3）算法在运行过程中临时占用的存储空间。

其中前两部分和算法无关，所以算法的空间复杂度一般仅考虑第三部分。

与时间复杂度一样，空间复杂度也用数量级（Order）表示，即 S(n)=O(g(n))。

1.5 本章小结

本章主要介绍数据结构的基本概念，重点如下：

（1）理解常见逻辑结构的概念和区别。

（2）深入理解不同存储结构的特点及适用场合。

（3）熟练掌握算法的概念、特征和评价方法。

第 2 章　线性表

本章学习目标

- 理解线性表的基本概念和特点。
- 掌握两类存储结构的本质。
- 熟练掌握两类存储结构的操作细节，尤其是单链表、单循环链表、双链表、双循环链表在操作上的差异。
- 掌握本章内容和后续查找、排序等章节的结合出题。

2.1　本 章 导 学

线性表

2.1.1　知识结构

本章知识结构如图 2.1 所示，加粗框中的内容需要考生重点理解并掌握。

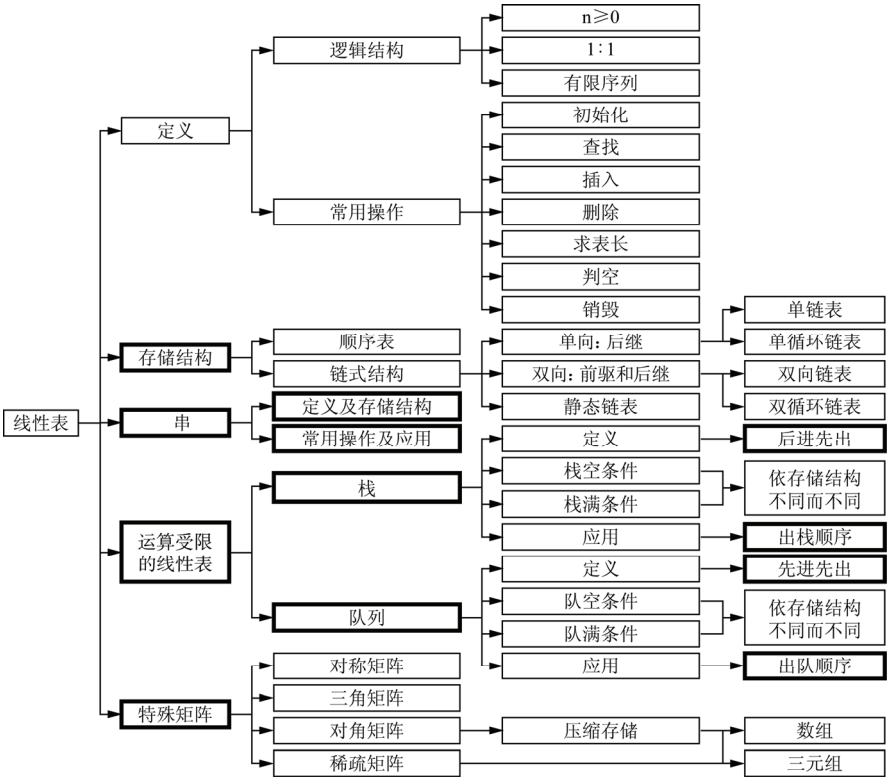


图 2.1　本章知识结构

2.1.2　命题特点

1．命题规律

（1）本章为各高校硕士研究生招生考试的重点考查内容，既有客观题又有主观题。

（2）本章内容既可单独命题，也可与后续章节联合命题。

（3）顺序存储结构、链式存储结构的操作细节易出客观题，链表操作的综合应用易出主观题。

2．考核趋势

本章在各高校硕士研究生入学考试中占据重要地位，其基本概念和原理对于后续章节具有重要参考作用，主观、客观题目均易出。

考生需特别注意以下内容。

（1）顺序存储结构与查找、排序两章联合命题。

（2）链式存储结构的综合应用。

（3）栈、队列的特点及应用。

（4）串的特点、存储及应用。

（5）特殊矩阵的存储及操作。

2.2　线性表概述

线性表是最基本、最常用的数据结构，表中各元素的逻辑关系具有 1∶1 的串行特性，编程实现时根据实际应用场景可以采用不同的存储结构，使逻辑上相邻的元素对应的存储位置未必相邻，增加了应用的灵活性。

2.2.1　定义

1．概念

具有相同数据类型的 n（n≥0）个数据元素的有限、有序序列称为线性表，n 为表长，n=0 时称为空表。

以下 2 点需特别注意：

（1）“有限”指的是任何线性表包含的元素个数是有限的。

（2）“有序”指的是逻辑上的先后顺序，而非存储空间的前后位置。

2．非空线性表的特点

（1）同一线性表中的所有数据元素具有相同的数据类型。

（2）线性表中的数据元素个数有限；

（3）数据元素之间总体满足 1∶1 的逻辑关系。

（4）存在唯一被称为“第一个”的元素（表头），该元素只有后继，没有前驱。

（5）存在唯一被称为“最后一个”的元素（表尾），该元素只有前驱，没有后继。

（6）除第一个元素外，其他元素均只有唯一的前驱。

（7）除最后一个元素外，其他元素均只有唯一的后继。

（8）线性表的抽象数据类型 ADT 描述

```
ADT list{
     数据对象 D: D={ ai| ai|∈eleSet, i=1,2,…,n, n≥0}
     数据关系 R: R={< ai-1, ai > | | ai-1, ai∈D, i=2,…,n, n≥0}
     基本操作 P:
       void initList(*L);              // 初始化线性表，构造空线性表，表长为 0
       void insertList (*L,i,e);       // 在线性表的第 i 个元素之前插入一个元素 e
       unsigned  listLength(L);        // 求线性表的长度
       eleType  getElem1(L,i);         // 获取线性表的第 i 个元素
       void getElem2(L,i,*e);          // 获取线性表的第 i 个元素，放入 e
       unsigned  locateElem(L,e);      // 返回 e 在线性表中的位置
       void listDelete(*L,i,*e);       // 删除线性表的第 i 个元素,将其值存放于变量 e
       void printList (L,visit());     // 遍历线性表
       int  emptyList (L);             // 线性表判空
       void destroyList(*L);           // 销毁线性表
} ADT list
```

3．应用场合

线性表是最简单、最常用的数据结构。对于一个实际应用问题，如果待处理的数据元素性质相同，且任意数据元素之间具有串行关系，就可以将其抽象为线性表。例如学生成绩、销售数据等。

2.2.2 基本操作

基本操作指常用、必要的操作，其他操作可以由基本操作实现。同一基本操作在不同存储结构下的实现方法不同。

（1）初始化线性表：initList(*L)，构造空线性表，表长为 0，但顺序表的初始化还需为线性表申请到初始存储空间。返回值——新构建的线性表由地址形式的形参带回调用函数，函数返回值类型为空类型 void。

（2）在线性表的第 i 个元素之前插入一个元素 e： insertList (*L,i,e)，对线性表的改变通过地址形式的形参 L 返回，函数返回值类型为空类型 void。

（3）求线性表的长度：listLength(L)，由函数中的返回语句返回线性表当前的数据元素个数，形参为值参，函数返回值类型为整型 int 或无符号类型 unsigned。

（4）获取线性表的第 i 个元素： 即按照逻辑位置查找线性表 L 的第 i 个数据元素，返回值可通过函数中的返回语句返回，比如 getElem1(L,i)，此时函数返回值类型应该和数据元素类型一致；也可通过地址形式的形参带回调用函数，比如 getElem2(L,i,*e)，此时函数返回值类型为空类型 void。

（5）确定元素 e 是线性表的第几个元素：locateElem(L,e)，即按值查找线性表中是否存在值为 e 的数据元素。若存在，返回其逻辑位置，否则返回 0。函数返回值类型为无符号类型 unsigned。

（6）删除线性表的第 i 个元素，并将其值存放于变量 e 中：listDelete(*L,i,*e)，对线性表的改变通过地址形式的形参 L 和 e 返回，函数返回值类型为空类型 void。

（7）遍历线性表：printList(L,visit())，以 visit()方式输出线性表 L 中的所有元素，函数返回值类型为空类型 void。

（8）线性表判空：emptyList (L)，判断线性表 L 中的数据元素个数是否为 0，或者表长是否为 0，函数返回值类型为布尔型，C 语言中可用整型 int 替代，若线性表为空，返回非零值（真），否则返回 0（假）。

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，函数返回值类型为空类型 void。

2.3　线性表存储结构及操作实现

2.3.1　顺序表

1．概念

顺序表即线性表的顺序存储方式，通常用一组地址连续的内存单元（数组）依次存储线性表的所有元素，即逻辑上相邻的数据元素存储在相邻的物理空间中，物理位置关系反映了逻辑关系。

2．存储形式

假设某线性表为 L={D,R}，D={a_i}，i∈[1,n]，R={<a_{i-1},a_i>}，其中 a_{i-1} 称为 a_i 的直接前驱，a_i 称为 a_{i-1} 的直接后继，a_0 无直接前驱，a_n 无直接后继。假设每个数据元素占 1 个单元，则其顺序表存储形式如图 2.2 所示。

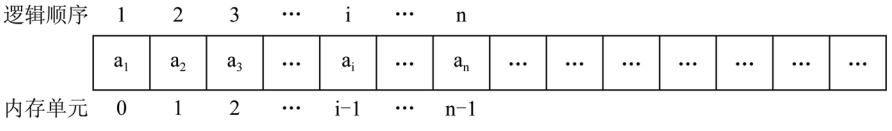


图 2.2　顺序表存储形式

3．定义

顺序表的定义以数组为主体，按照数组大小是否可变分为静态分配和动态分配两种形式。假设数据元素的类型为 eleType。

1）静态分配

编译时已确定数组的大小和位置，程序运行期间不可改变。

```
#define maxSize 1000          // 顺序表的最大长度
typedef  struct {             // 定义结构体类型
   eleType Selem[maxSize];    // 线性表存储于数组 Selem，一次性申请 maxSize 个数
                              // 据元素所需的存储空间 maxSize*sizeof(eleType)
                              // 个存储单元
  unsigned  length;           // 顺序表的当前长度，增删元素时需同步进行加减操作，
                              // length 取值范围为 0～maxSize-1
}Slist;                       // 静态分配空间的顺序表的类型名为 Slist
```

注意：

- 若 length 已等于 maxSize，则不可进行插入操作。
- 若 length 已等于 0，则不可进行删除操作。

2）动态分配

初始化程序执行期间通过 malloc 函数为数组申请空间，程序运行期间若空间不够，可通过 realloc 函数在保留原存储值的前提下为数组重新申请更大的连续存储空间。

```
#define initSize 1000   // 顺序表的初始长度
#define incSize 500     // 增大顺序表存储空间时，每次的增长值
typedef  struct {       // 定义结构体类型
   eleType *Delem;      // 线性表存储于指向数组的指针 Delem，当前该数组并不存
                        // 在，程序运行期间需要申请 initSize*sizeof(eleType)
                        // 个存储单元
   unsigned  length;    // 顺序表的当前长度，增删元素时需同步进行加减操作，
                        // length 取值为不超过当前申请存储单元个数的无符号数
}Dlist;                 // 动态分配空间的顺序表的类型名为 Dlist
```

注意：

- 初始化时，为顺序表申请 initSize 个数据元素所需的连续存储空间，首地址存放于指针变量 Delem。
- 若 length 已等于 maxSize，进行插入操作前需执行 realloc 函数，这样既保留原存储值，又能为数组重新申请更大的连续存储空间，每次增长 500 个数据元素所占的存储空间量。
- 若 length 已等于 0，则不可进行删除操作。

4．基本操作实现

1）初始化线性表：initList(*L)

（1）静态分配代码如下。

```
void initList(SList *L){
    L->Selem=(eleType *)malloc(maxSize *sizeof(eleType));
     if(!L->Selem)                  // 没有分配成功
       exit(OVERFLOW);              // 退出程序，提示溢出
    L->length=0;
     return;
}
```

（2）动态分配代码如下。

```
void initList(DList *L){
     L->Delem=(eleType *)malloc(initSize *sizeof(eleType));
     if(!L->Delem)                    // 没有分配成功
        exit(OVERFLOW);               // 退出程序，提示溢出
     L->length=0;
     return;
}
```

2）在线性表 L 的第 i 个元素(逻辑位置)之前插入一个元素 e：insertList (*L,i,e)

（1）静态分配代码如下。

```
void insertList(SList *L,unsigned i, eleType e){
    if(L->length==maxSize)           // 存储空间已满
       exit(OVERFLOW);               // 退出程序，提示溢出
    if(i<1 || i>L->length)           // 非法逻辑位置
       exit(ERROR);                  // 退出程序，提示位置出错
     for(unsigned j=L->length-1;j>=i-1;j--)
         L->Selem[j+1]= L->Selem[j]; // 从表尾开始到插入位置，数据元素依次后
                                     // 移一个位置
     L->Selem[i-1]=e;                // e 插入到线性表的第 i 个位置
     L->length++;                    // 表长加 1
     return;
}
```

（2）动态分配代码如下。

```
void insertList(DList *L,unsigned i, eleType e){
    if(L->length==initSize){         // 存储空间已满
      eleType *p;
      p=(eleType*)realloc(L->Delem,(initSize+incSize) *sizeof(eleType));
                                     // 重新申请(initSize+incSize)*sizeof-
                                     // (eleType) 大小的存储空间，L->Delem
                                     // 中的 L->length 个数据元素复制过来，新
                                     // 空间首地址为 p
       if(!p)                        // 没有分配成功
          exit(OVERFLOW);            // 退出程序，提示溢出
       L->Delem=p;                   // L->Delem 指向新申请到的存储空间
      L->length+= incSize;           // 表长修改为新的存储空间可存放数据元
                                     // 素个数
    }
    if(i<1 || i>L->length)           // 非法逻辑位置
       exit(ERROR);                  // 退出程序，提示位置出错
     for(unsigned j=L->length-1;j>=i-1;j--)
         L->Delem[j+1]= L->Delem[j]; // 从表尾开始到插入位置，数据元素依次后
                                     // 移一个位置
     L->Delem[i-1]=e;                // e 插入到线性表的第 i 个位置
     return;
}
```

3）求线性表的长度：listLength(L)

（1）静态分配代码如下。

```
unsigned  listLength (SList L){
        return  L.length;
}
```

（2）动态分配代码如下。

```
unsigned  listLength(DList L){
          return  L.length;
}
```

4）获取线性表的第 i 个元素：getElem1(L,i)或 getElem2(L,i,*e)

（1）静态分配代码如下。

```
eleType  getElem1(SList L,unsigned i) {  // i 为逻辑位置,取值范围为1～L.length
    if(i<1 || i>L.length)                // 非法逻辑位置
       exit(ERROR);                      // 退出程序，提示位置出错
    return L.Selem[i];
}
```

或

```
void getElem2(SList L,unsigned i, eleType *e) {
                                         // i 为逻辑位置,取值范围为1～L.length
    if(i<1 || i>L.length)                // 非法逻辑位置
       exit(ERROR);                      // 退出程序，提示位置出错
    *e=L.Selem[i];
    return;
}
```

（2）动态分配代码如下。

```
eleType  getElem1(DList L,unsigned i) {  // i 为逻辑位置,取值范围为1～L.length
    if(i<1 || i>L.length)                // 非法逻辑位置
       exit(ERROR);                      // 退出程序，提示位置出错
    return L.Delem[i];
}
```

或

```
void getElem2(DList L,unsigned i, eleType *e) {
                                         // i 为逻辑位置,取值范围为1～L.length
    if(i<1 || i>L.length)                // 非法逻辑位置
       exit(ERROR);                      // 退出程序，提示位置出错
    *e=L.Delem[i];
    return;
}
```

5）确定元素 e 是线性表的第几个元素(逻辑位置)：locateElem(L,e)

（1）静态分配代码如下。

```
unsigned  locateElem(SList L, eleType e) {
```

```
    for(int i=0;i<L.length-1;i++)
         if(L.Selem[i]==e)
        break;
      if(i!=L.length)
  return i+1;
        else
       return 0;
}
```

（2）动态分配代码如下。

```
unsigned  locateElem(DList L, eleType e) {
    for(int i=0;i<L.length-1;i++)
         if(L.Delem[i]==e)
         break;
    if(i!=L.length)
       return i+1;
        else
       return 0;
}
```

6）删除线性表的第 i 个元素，将其放在变量 e 中：listDelete(*L,i,*e)

（1）静态分配代码如下。

```
void listDelete(SList *L, unsigned i, eleType *e){
    if(L->length==0)         // 空表不能删除
       exit(EMPTY);          // 退出程序，提示空表
    if(i<1 || i>L->length)   // 非法逻辑位置
       exit(ERROR);          // 退出程序，提示位置出错
     *e=L->Selem[i-1]);      // 线性表的第 i 个元素存放于变量 e
     for(eleType *p=L->Selem[i]);p>=L->Selem[L->length-1];p++)
                             // 从第 i 个数据元素开始，依次将后面的元素前移一个位置
        *(p-1)=*p;
    --L->length;             // 表长减 1
     return;
}
```

（2）动态分配代码如下。

```
void listDelete(DList *L, unsigned i, eleType *e){
    if(L->length==0)         // 空表不能删除
       exit(EMPTY);          // 退出程序，提示空表
    if(i<1 || i>L->length)   // 非法逻辑位置
       exit(ERROR);          // 退出程序，提示位置出错
     *e=L->Delem[i-1]);      // 线性表的第 i 个元素存放于变量 e
     for(eleType *p=L->Delem[i]);p>=L->Delem[L->length-1];p++)
                             // 从第 i 个数据元素开始，依次将后面的元素前移一个位置
        *(p-1)=*p;
    --L->length;             // 表长减 1
     return;
}
```

7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值

（1）静态分配代码如下。

```
void printList(SList L){
    for(int i=0;i<L.length-1;i++)
       printf(L.Selem[i]);
    return;
}
```

（2）动态分配代码如下。

```
void printList(DList L){
    for(int i=0;i<L.length-1;i++)
       printf(L.Delem[i]);
    return;
}
```

8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0

（1）静态分配代码如下。

```
int emptyList (SList L){
    return(!L.length);
}
```

（2）动态分配代码如下。

```
int emptyList (DList L){
    return(!L.length);
}
```

9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间

（1）静态分配代码如下。

```
void destroyList(SList *L){
     if(!L->Selem)       // 没有可销毁的内容
        exit(ERROR);
     free(L->Selem);
     L->length=0;
}
```

（2）动态分配代码如下。

```
void destroyList(DList *L){
     if(!L->Delem)       // 没有可销毁的内容
        exit(ERROR);
     free(L->Delem);
     L->length=0;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

5．应用场合

顺序表主要应用在数据量不大，且很少有插入、删除操作的场合。例如，学生基本信息。

2.3.2　链表

1．概念

链表是线性表的链式存储方式，线性表中每个数据元素单独申请和释放存储空间，逻辑相邻的元素其存储单元不必相邻，其特点如下。

（1）线性表的链式存储结构中，所有数据元素所占用的存储空间可以连续，也可以不连续。

（2）单个数据元素的存储单元不仅要存储数据元素的值（值域），还要存储数据元素之间的关系（指针域）。

（3）通过指针域反映逻辑关系。

（4）插入删除操作通过修改指针域完成，不用移动元素。

（5）查找操作需要遍历整个链表，无法实现随机查找。

（6）带头结点的链表，其头结点的值域可用来存放链表中数据元素个数（eleType 为数值型）、特殊值（例如最大值）等，也可不用。本书部分内容用其保存链表中数据元素的个数。

根据存储空间是整体申请还是插入操作时才申请，链表分为动态链表和静态链表。动态链表为进行插入操作时，为待插入元素申请所需的存储空间。静态链表为事先申请约定大小的内存空间，每个数据元素的指针域存放其前驱或后继的地址，所以逻辑上相邻的元素不一定放在相邻的位置上。

无论是动态链表，还是静态链表，都可构造单向链表或双向链表。

线性表的单向链表中每个数据元素一般包含一个指针域，用来存放该元素的直接后继的地址。线性表的双向链表中每个数据元素一般包含两个指针域，分别用来存放该元素的直接前驱和直接后继的地址。

无论单向链表，还是双向链表，均可通过首元素和末元素的指针域构造循环链表。无论单向链表，还是双向链表，均可包含头结点，即不存储任何数据元素，仅代表链表的结点。此时，空链表包含 1 个结点。可利用头结点存储链表的特有信息，例如元素个数、最大元素、最小元素等。

考生要重点掌握动态链表的非循环单向链表（单向链表）、单向循环链表、非循环双向链表（双向链表）、双向循环链表，以及静态链表的非循环静态链表（静态链表）。

2．单向链表

1）单向链表的存储结构

```
typedef  struct  Node{
     eleType      data;
     struct  Node  *next;
}LNode, *linkList;
linkList  L;  // 定义类型为 linkList 的指针变量 L，标识整个链表，指向“第一个结点”
```

不带头结点的单向链表如图 2.3 所示，带头结点的单向链表如图 2.4 所示。

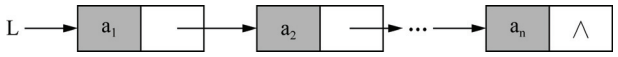


图 2.3　不带头结点的单向链表

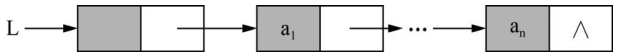


图 2.4　带头结点的单向链表

2）单向链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(linkList L){
    L=NULL;                          // NULL 为空指针，可记为∧
    return;
}
```

② 带头结点的代码如下。

```
void initList2(linkList L){
    L=( LNode *)malloc(sizeof(LNode));
    if(!L)                           // 没有分配成功
        exit(OVERFLOW);              // 退出程序，提示溢出
    L->data=0;                       // 可以利用头结点的值域存放表长，不是必须
    L->next=NULL;                    // NULL 为空指针，可记为∧
    return;
}
```

（2）建立单链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(linkList L){    // 表头插入
    initList1(L);
    LNode *p;                        // p 为待插入链表的结点
    eleType x;                       // x 为待插入链表的结点的值
    scanf(&x);                       // 读取第一个元素的值
    while (x!=EOF){                  // EOF 为用户自定义的结束标记值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=L;
        L=p;
        scanf(&x);                   // 读取下一个元素的值
    }
    return;
 }
```

或

```
 void creatList12(linkList L){   // 表尾插入
    initList1(L);
    LNode *p;                        // p 为待插入链表的结点
    LNode *tail;                     // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
```

```
    scanf(&x);                                  // 读取第一个元素的值
    if (x==EOF)                                 // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=( LNode *)malloc(sizeof(LNode));          // 建立含一个结点的链表
    p->data=x;
    p->next=NULL;
    L=p;
    tail=L;
    while (x!=EOF){                             // EOF 为用户自定义的结束标记值
        scanf(&x);                              // 读取下一个元素的值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=NULL;                           // 也可用 p->next= tail->next
        tail->next=p;                           // p 链接至当前表尾 tail 之后
        tail=p;                                 // p 为新的表尾 tail
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(linkList L){    // 表头插入
    initList2(L);
    LNode *p;                    // p 为待插入链表的结点
    eleType x;                   // x 为待插入链表的结点的值
    scanf(&x);                   // 读取第一个元素的值
    while (x!=EOF){              // EOF 为用户自定义的结束标记值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=L->next;
        L->next =p;
        L->data++;               // 不是必须，可以利用头结点的值域存放表长
        scanf(&x);               // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(linkList L){    // 表尾插入
    initList2(L);
    LNode *p;                    // p 为待插入链表的结点
    LNode *tail=L;               // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                   // 读取第一个元素的值
    while (x!=EOF){              // EOF 为用户自定义的结束标记值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=NULL;            // 也可用 p->next= tail->next
        tail->next=p;            // p 链接至当前表尾 tail 之后
        tail=p;                  // p 为新的表尾 tail
        L->data++;               // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);               // 读取下一个元素的值
    }
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

- 不带头结点的代码如下。

```
LNode *getElem11 (linkList L,unsigned i){    // i 为逻辑位置
    LNode *p=L;                              // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
        return p;
 }
```

或

```
void getElem21(linkList L,unsigned i, LNode *e){    // i 为逻辑位置
    LNode *p=L;                              // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    e=p;
    return;
}
```

- 带头结点的代码如下。

```
LNode *getElem12 (linkList L,unsigned i){    // i 为逻辑位置
    LNode *p=L->next;                        // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    return p;
}
```

其中 while 循环部分也可改为

```
if(i<1 || i>L->data)
    return NULL
while(j!=i)
    p=p->next, j++;
```

或

```
void getElem22(linkList L,unsigned i, LNode *e){    // i 为逻辑位置
    LNode *p=L->next;                        // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
```

```
    }
    e=p;
    return;
}
```

其中 while 循环部分也可改为

```
if(i<1 || i>L->data)
    e=NULL;
while(j!=i)
    p=p->next, j++;
```

② 获取特定值的结点：getElem*(L,x)。

- 不带头结点的代码如下。

```
LNode *getElem3 (linkList L, eleType x){
    LNode *p=L;                                  // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

- 带头结点的代码如下。

```
LNode *getElem4 (linkList L,unsigned i){     // i 为逻辑位置
    LNode *p=L->next;                        // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素（逻辑位置）插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(linkList L, unsigned i, eleType x){  // 以不带头结点链表为例
       LNode *p,*q;                    // q 指向插入位置的前驱
       q=getElem11(L, i-1);            // 带头结点链表改为 q=getElem12(L,i-1)
       if(q==NULL)                     // i 值过大
           exit(ERROR);
       p=( LNode *)malloc(sizeof(LNode));
       p->data=x;
       p->next=q->next;
       q->next=p;
       // 带头结点链表可增加语句 L->data++来修正表长，不是必须
       return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(linkList L, eleType x){         // 以不带头结点链表为例
       LNode *p;
       p=( LNode *)malloc(sizeof(LNode));
       p->data=x;
       p->next=L;                      // 带头结点链表改为 p->next=L->next
```

```
        L=p;                                    // 带头结点链表改为 L->next=p
        // 带头结点链表可增加语句 L->data++来修正表长，不是必须
        return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(linkList L){
    LNode *p=L;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(linkList L){
    LNode *p=L->next;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void listDelete1(linkList L,unsigned i){     // i 为逻辑位置
    LNode *p,*q;
    p=getElem11(L,i-1);                 // p 为待删结点的前驱结点
                                        // 带头结点链表改为 p=getElem21(L,i-1)
    if(p==NULL || p->next==NULL)        // 不存在第 i 个结点
        exit(ERROR);
    q=p->next;                          // q 为待删结点
    p->next=q->next;                    // q 脱离链表
    free(q);                            // 释放 q 所占存储空间
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

- 表中重复元素值的代码如下。

```
void listDelete2(linkList L, eleType x){
    LNode *p,*q;
    p=L;              // 带头结点链表改为 p=L->next，p 最终指向待删结点的前驱结点
```

```
    if(p->data==x){  // 第一个结点的值为 x
        L=L->next;   // 带头结点链表改为 L->next=L->next->next
        free(p);
                     // 带头结点链表增加 L->data--，不是必须
        return;
    }
    q=p->next;
    while(q!=NULL && q->data!=x) // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q==NULL)                  // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;             // q 脱离链表
    free(q);                     // 释放 q 所占存储空间
                                 // 带头结点链表增加 L->data--，不是必须
    return;
}
```

- 表中有重复元素值，删除所有目标元素的代码如下。

```
void listDelete2(linkList L, eleType x){
    LNode *p,*q;
    p=L;             // 带头结点链表改为 p=L->next，p 最终指向待删结点的前驱结点
    if(p->data==x){  // 第一个结点的值为 x
        L=L->next;   // 带头结点链表改为 L->next =L->next ->next
        free(p);
                             // 带头结点链表增加语句 L->data--，不是必须
        return;
    }
    q=p->next;
    while(q!=NULL){          // 定位待删结点及其前驱
        while(q!=NULL && q->data!=x){
            p=q;
            q=p->next;
        }
        if(q==NULL)          // 不存在值为 x 的结点
            exit(ERROR);
        else{                // 找到 1 个值为 x 的结点
            p->next=q->next; // q 脱离链表
            free(q);         // 释放 q 所占存储空间
                             // 带头结点链表增加语句 L->data--，不是必须
        }
        q=p->next;           // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void printList(linkList L){
    LNode *p;
    p=L;                 // 带头结点链表改为 p=L->next，p 指向待访问结点
    while(p!=NULL){
        printf(p->data);
        p=p->next;
```

```
    }
    return;
}
```

（8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(linkList L){
    if(L==NULL)           // 带头结点链表改为 if(L->next==NULL)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点链表算法类似，下面代码以不带头结点链表为例。

```
void destroyList(linkList L){
    LNode *p,*q;
    p=L;                  // 带头结点链表改为 p=L->next，p 指向待删结点
    while(p!=NULL){
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

3）单向链表的应用场合

单向链表不必占用一块连续的内存空间。在查找、取值等静态操作比较少，增加、删除操作比较多的情况下，单向链表应用较多。

3．单向循环链表

单向循环链表的最后一个结点的后继为第一个结点。

1）单向循环链表的存储结构

```
Typedef struct RNode{
     eleType data;
     struct RNode  *next;
}RLNode, *RlinkList;
RlinkList RLtail;
```

定义类型为 RlinkList 的指针变量 RLtail，标识整个链表，指向“最后一个结点”，其后继为头结点。对于循环链表来讲，定义尾结点既能表示最后一个结点，又能标识头结点，实现基本操作的算法比较方便。

不带头结点的单向链表如图 2.5 所示，带头结点的单向链表如图 2.6 所示。

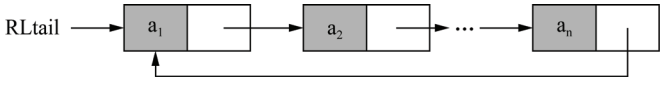


图 2.5　不带头结点的单向循环链表

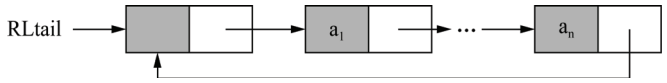


图 2.6　带头结点的单向循环链表

2）单向循环链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(RlinkList RLtail){
    RLtail=NULL;                        // NULL 为空指针，可记为∧
    return;
}
```

② 带头结点的代码如下。

```
void initList2(RlinkList RLtail){
    RLtail=(RLNode *)malloc(sizeof(RLNode));
    if(!RLtail)                         // 没有分配成功
        exit(OVERFLOW);                 // 退出程序，提示溢出
    RLtail->next=RLtail;
    RLtail->data=0;                     // 可以利用头结点的值域存放表长，不是必须
    return;
}
```

（2）建立单向循环链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(RlinkList RLtail){ // 表头插入
    initList1(RLtail);
    RLNode *p;                          // p 为待插入链表的结点
    eleType x;                          // x 为待插入链表的结点的值
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    RLtail=(RLNode *)malloc(sizeof(RLNode));
    RLtail->data=x;
    RLtail->next=RLtail;                // RLtail->next 为表头结点
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(RLNode *)malloc(sizeof(RLNode));
        p->data=x;
        p->next=RLtail->next;           // RLtail->next 为表头结点
        RLtail->next =p;
        scanf(&x);                      // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList12(RlinkList RLtail){ // 表尾插入
    initList1(RLtail);
    RLNode *p;                          // p 为待插入链表的结点
```

```
    eleType x;
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
       exit(EMPTY);
    p=(RLNode *)malloc(sizeof(RLNode)); // 建立含一个结点的链表
    p->data=x;
    p->next=p;
    RLtail=p;
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
       p=(RLNode *)malloc(sizeof(RLNode));
       p->data=x;
       p->next=RLtail->next;            // p 指向表头 RLtail->next，为新的表尾
       RLtail->next=p;                  // p 链接至当前表尾 RLtail 之后
       RLtail=p;                        // p 为新的表尾 RLtail
       scanf(&x);                       // 读取下一个元素的值
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(RlinkList RLtail){ // 表头插入
    initList2(RLtail);
    RLNode *p;                          // p 为待插入链表的结点
    eleType x;                          // x 为待插入链表的结点的值
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
       exit(EMPTY);
    p=(RLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->next=RLtail->next;
    RLtail=p;                           // RLtail 指向第一个插入的结点，即表尾
    RLtail->next->data++;               // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
       p=(RLNode *)malloc(sizeof(RLNode));
       p->data=x;
       p->next=RLtail->next->next;
       RLtail->next->next =p;
       RLtail->next->data++;            // 不是必须，可利用头结点的值域存放表长
       scanf(&x);                       // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(RlinkList RL){     // 表尾插入
    initList2(RL);
    RLNode *p;                          // p 为待插入链表的结点
    RLNode *tail=RL;                    // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
```

```
        exit(EMPTY);
    p=(RLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->next=RLtail->next;
    RLtail=p;                          // RLtail 指向第一个插入的结点，即表尾
    RLtail->next->data++;              // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                         // 读取第二个元素的值
    while (x!=EOF){                    // EOF 为用户自定义的结束标记值
        p=(RLNode *)malloc(sizeof(RLNode));
        p->data=x;
        p->next=RLtail->next;          // p 指向表头
        RLtail->next=p;                // p 链接至当前表尾 RLtail 之后
        RLtail=p;                      // p 为新的表尾 RLtail
        L->data++;                     // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);                     // 读取下一个元素的值
    }
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

- 不带头结点的代码如下。

```
RLNode *getElem11(RlinkList RLtail,unsigned i){          // i 为逻辑位置
    RLNode *p=RLtail->next;                    // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        return p;
    else
        return NULL;
}
```

或

```
void getElem21(RlinkList RLtail,unsigned i, RLNode *e){  // i 为逻辑位置
    RLNode *p=RLtail->next;                    // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        e=p;
    else
        e=NULL;
    return;
}
```

- 带头结点的代码如下。

```
RLNode *getElem12(RlinkList RLtail,unsigned i){     // i 为逻辑位置
    RLNode *p=RLtail->next->next;            // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next) {
        p=p->next;
        j++;
    }
    if(i==j)
        return p;
    else
        return NULL;
}
```

查找部分的代码也可改为

```
if(i<1 || i> RLtail->next ->data)
    return NULL;
while(j!=i)
    p=p->next, j++;
return p;
```

或

```
RLNode *getElem22(RlinkList RLtail,unsigned i, RLNode *e) // i 为逻辑位置
{
    RLNode *p=RLtail->next->next;        // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        e=p;
    else
        e=NULL;
    return;
}
```

查找部分的代码也可改为

```
if(i<1 || i> RLtail->next ->data)
    e=NULL;
while(j!=i)
    p=p->next, j++;
    e=p;
return;
```

② 获取特定值的结点：getElem*(L,x)，下面代码以不带头结点的循环链表为例，带头结点的算法类似。

```
RLNode *getElem3 (RlinkList RLtail, eleType x){
    RLNode *p=RLtail->next;
    // p 为遍历链表元素结点的指针，带头结点链表改为 RLNode *p=RLtail->next->next
    while(p->data!=x && p->next!=RLtail->next)
```

```
        p=p->next;
    if(p->data==x)
        return p;
    else
        return NULL;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素(逻辑位置)插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(RlinkList RLtail, unsigned i, eleType x){
            // 以不带头结点链表为例
    RLNode *p,*q;
    q=getElem11(RLtail, i);
            // 带头结点链表改为 q=getElem12(RLtail->next,i-1);
    if(q==NULL)
        exit(ERROR);
    p=(RLNode *)malloc(sizeofR(LNode));
    p->data=x;
    p->next=q->next;
    q->next=p;
            // 带头结点链表可增加语句 RLtail->next->data++来修正表长，不是必须
    return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(RlinkList RLtail, eleType x){ // 以不带头结点链表为例
    RLNode *p;
    p=(RLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->next=RLtail->next;
    // 带头结点链表改为 p->next=RLtail->next ->next
    RLtail->next=p;       // 带头结点链表改为 RLtail->next->next=p
    // 带头结点链表可增加语句 RLtail->next->data++来修正表长，不是必须
    return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(RlinkList RLtail){
    RLNode *p;
    unsigned i=1;
    if(RLtail==NULL)      // 空表
        return 0;
    p= RLtail->next;
    while(p->next!= RLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(RlinkList RLtail){
    RLNode *p=RLtail->next;
    unsigned i=1;
    if(p->next==p)    // 空表
        return 0;
    while(p->next!=RLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，以不带头结点为例。

```
void listDelete1(RlinkList RLtail,unsigned i){  // i 为逻辑位置
    RLNode *p,*q;
    p=getElem11(RLtail,i-1);                    // p 为待删结点的前驱结点
    // 带头结点链表改为 p=getElem21(RLtail,i-1)
    q= getElem11(RLtail,i);                     // q 为待删结点
    // 带头结点链表改为 q=getElem21(RLtail,i)
    if(p==NULL || q==NULL)                      // 不存在第 i 个结点
        exit(ERROR);
    p->next=q->next;                            // q 脱离链表
    free(q);                                    // 释放 q 所占存储空间
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，以不带头结点为例。

- 表中无重复元素值的情况代码如下：

```
void listDelete2(RlinkList RLtail, eleType x){
    RLNode *p,*q;
    p= RLtail->next;      // p 初值指向首元素结点，最终指向待删结点的前驱
                          // 带头结点链表改为 p= RLtail->next ->next
    if(p->data==x){       // 第一个结点的值为 x
        RLtail->next=RLtail->next->next;          // 删除首元素结点
                          // 带头结点链表改为
                          // RLtail->next->next=RLtail->next->next->next->next;
        free(p);
                          // 带头结点链表增加语句 RLtail->next->data--，不是必须
        return;
    }
    q=p->next;
    while(q!= RLtail->next && q->data!=x)         // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q== RLtail->next)                          // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;                              // q 脱离链表
```

```
    free(q);                // 释放 q 所占存储空间
                            // 带头结点链表增加语句 RLtail->next->data--,不是必须
    return;
}
```

- 表中有重复元素，删除表中所有目标元素的代码如下：

```
void listDelete2(RlinkList RLtail, eleType x){
    RLNode *p,*q;
    p= RLtail->next;        // p 初值指向首元素结点，最终指向待删结点的前驱
                            // 带头结点链表改为 p=RLtail->next->next
    if(p->data==x){         // 第一个结点的值为 x
        RLtail->next=RLtail->next->next; // 删除首元素结点
                            // 带头结点链表改为
                            // RLtail->next->next=RLtail->next->next->next
        free(p);
                            // 带头结点链表增加语句 RLtail->next->data--，不是必须
        return;
    }
                            // 第一个结点的值不是 x
    q=p->next;
    while(q!=RLtail->next){  // 定位待删结点及其前驱
        while(q!=RLtail->next && q->data!=x){
            p=q;
            q=p->next;
        }
        if(q==RLtail->next)   // 不存在值为 x 的结点
            exit(ERROR);
        else{                 // 找到一个值为 x 的结点
            p->next=q->next;  // q 脱离链表
            free(q);          // 释放 q 所占存储空间
                              // 带头结点链表可增加语句 RLtail->next->data--,
                              // 不是必须
        }
        q=p->next;            // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点链表算法类似，以不带头结点为例。

```
void printList(RlinkList RLtail){
    RLNode *p;
    if(RLtail==NULL)          // 空表则直接退出
                              // 带头结点链表改为 if(RLtail->next == RLtail)
        exit(ERROR);
    p= RLtail->next;          // p 指向待访问结点，初值为首结点
                              // 带头结点链表改为 p= RLtail->next ->next
    do{
        printf(p->data);
        p=p->next;
    } while(p!= RLtail->next)
    return;
}
```

（8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(RlinkList RLtail){
    if(RLtail==NULL)      // 空表，带头结点链表改为 if(RLtail->next == RLtail)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void destroyList(RlinkList RLtail){
    RLNode *p,*q;
    p=RLtail->next;       // p 指向待删结点
                          // 带头结点链表改为 p=RLtail->next->next
    while(p!=NULL){       // 带头结点链表改为 while(p!=p->next)
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

4．双向链表

1）双向链表的存储结构

```
typedef  struct  DNode{
     eleType      data;
     struct  DNode  *prior;      // 指向前驱
     struct  DNode  *next;       // 指向后继
 }DLNode, *DlinkList;
DlinkList DL;
```

定义类型为 DlinkList 的指针变量 DL，标识整个链表，指向“第一个结点”。

不带头结点的双向链表如图 2.7 所示，带头结点的双向链表如图 2.8 所示。

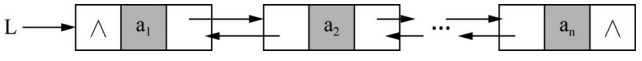


图 2.7　不带头结点的双向链表

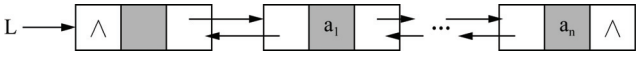


图 2.8　带头结点的双向链表

2）双向链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(DlinkList DL){
    DL=NULL;                  // NULL 为空指针，可记为∧
```

```
    return;
}
```

② 带头结点的代码如下。

```
void initList2(DlinkList DL){
    DL=(DLNode *)malloc(sizeof(DLNode));
    if(!DL)                          // 没有分配成功
        exit(OVERFLOW);              // 退出程序，提示溢出
    DL->data=0;                      // 可以利用头结点的值域存放表长，不是必须
    DL->prior=NULL;                  // NULL 为空指针，可记为∧
    DL->next=NULL;
    return;
}
```

（2）建立双链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(DlinkList DL){ // 表头插入
    initList1(DL);
    DLNode *p;                       // p 为待插入链表的结点
    eleType x;                       // x 为待插入链表的结点的值
    scanf(&x);                       // 读取第一个元素的值
    while (x!=EOF){                  // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
        p->data=x;
        p->prior=NULL;
        p->next=DL;
        if(DL!=NULL)
            DL->prior=p;
        DL=p;
        scanf(&x);                   // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList12(DlinkList DL){ // 表尾插入
    initList1(DL);
    DLNode *p;                       // p 为待插入链表的结点
    DLNode *tail;                    // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                       // 读取第一个元素的值
    if (x==EOF)                      // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(DLNode *)malloc(sizeof(DLNode)); // 建立含一个结点的链表
    p->data=x;
    p->prior=NULL;
    p->next=NULL;
    DL=p;
    tail=DL;
    scanf(&x);                       // 读取第二个元素的值
    while (x!=EOF){                  // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
```

```
        p->data=x;
        p->prior=tail;
        p->next=NULL;              // 也可以用 p->next= tail->next 语句
        tail->next=p;              // p 链接至当前表尾 tail 之后
        tail=p;                    // p 为新的表尾 tail
        scanf(&x);                 // 读取第一个元素的值
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(DlinkList DL){ // 表头插入
    initList2(DL);
    DLNode *p;                     // p 为待插入链表的结点
    eleType x;                     // x 为待插入链表的结点的值
    scanf(&x);                     // 读取第一个元素的值
    p=(DLNode *)malloc(sizeof(DLNode));
    p->data=x;
    p->prior=DL;
    p->next=NULL;
    DL->next=p;
    DL->data++;                    // 可以利用头结点的值域存放表长，不是必须
    scanf(&x);                     // 读取第二个元素的值
    while (x!=EOF){                // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
        p->data=x;
        p->prior=DL;
        p->next=DL->next;
        DL->next->prior=p;
        DL->next=p;
        DL->data++;                // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);                 // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(DlinkList DL){ // 表尾插入
    initList2(DL);
    DLNode *p;                     // p 为待插入链表的结点
    DLNode *tail=DL;               // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                     // 读取第一个元素的值
    while (x!=EOF){                // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
        p->data=x;
        p->prior=tail;
        p->next=NULL;              // 也可使用 p->next= tail->next 语句
        tail->next=p;              // p 链接至当前表尾 tail 之后
        tail=p;                    // p 为新的表尾 tail
        DL->data++;                // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);                 // 读取下一个元素的值
    }
```

```
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

- 不带头结点的代码如下。

```
DLNode *getElem11 (DlinkList DL,unsigned i){ // i 为逻辑位置
    DLNode *p=DL;                              // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    return p;
}
```

或

```
void getElem21(DlinkList DL,unsigned i, DLNode *e){ // i 为逻辑位置
    DLNode *p=DL;                              // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    e=p;
    return;
}
```

- 带头结点的代码如下。

```
DLNode *getElem12 (DlinkList DL,unsigned i){ // i 为逻辑位置
    DLNode *p=DL->next;                        // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    return p;
}
```

while 循环体部分的代码也可改为

```
if(i<1 || i>DL->data)
    return NULL;
while(j!=i)
    p=p->next, j++;
```

或

```
void getElem22(DlinkList DL,unsigned i, DLNode *e)      // i 为逻辑位置
{
    DLNode *p=DL->next;                       // p 为遍历链表元素结点的指针
```

```
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    e=p;
    return;
}
```

while 循环体部分的代码也可改为

```
if(i<1 || i>DL->data)
    e=NULL;
while(j!=i)
    p=p->next, j++;
```

② 获取特定值 x 的结点：getElem*(L,x)。

- 不带头结点的代码如下。

```
DLNode *getElem3(DlinkList DL, eleType x){
    DLNode *p=DL;                              // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

- 带头结点的代码如下。

```
DLNode *getElem4(DlinkList DL,unsigned i){  // i 为逻辑位置
    DLNode *p=DL->next;                        // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素（逻辑位置）插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(DlinkList L, unsigned i, eleType x){
                                    // 以不带头结点链表为例
    DLNode *p,*q;                   // q 指向插入位置的前驱
    q=getElem11(DL, i-1);           // 带头结点链表改为 q=getElem12(DL,i-1)
    if(q==NULL)
        exit(ERROR);
    p=(DLNode *)malloc(sizeof(DLNode));
    p->data=x;
    p->prior=q;
    p->next=q->next;
    q->next->prior=p;
    q->next=p;
    // 带头结点链表可增加语句 DL->data++来修正表长，不是必须
    return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(DlinkList DL, eleType x){  // 以不带头结点链表为例
    DLNode *p;
    p=(DLNode *)malloc(sizeof(DLNode));
    p->data=x;
    p->prior=NULL;                  // 带头结点链表改为 p->prior=DL
    p->next=DL;                     // 带头结点链表改为 p->next=DL->next
    DL->prior=p;                    // 带头结点链表改为 DL>next->prior=p
    DL=p;                           // 带头结点链表改为 DL->next=p
    // 带头结点链表可增加语句 DL->data++来修正表长，不是必须
    return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(DlinkList DL){
    DLNode *p=DL;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(DlinkList DL){
    DLNode *p=DL->next;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void listDelete1(DlinkList DL,unsigned i){      // i 为逻辑位置
    DLNode *p,*q;
    p=getElem11(DL,i-1);            // p 为待删结点的前驱结点
                                    // 带头结点链表改为 p=getElem21(DL,i-1)
    if(p==NULL || p->next==NULL) // 不存在第 i 个结点
        exit(ERROR);
    q=p->next;                      // q 为待删结点
    p->next=q->next;                // q 脱离链表
    q->next->prior=p
    free(q);                        // 释放 q 所占存储空间
                                    // 带头结点链表增加语句 DL->data--，不是必须
```

```
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，以不带头结点为例。

- 表中无重复元素值的情况代码如下。

```
void listDelete2(DlinkList DL, eleType x){
    DLNode *p,*q;
    p=DL;                          // 带头结点链表改为 p=DL->next，p 最终指向待
                                   // 删结点的前驱结点
    if(p->data==x){                // 第一个结点的值为 x
        DL=DL->next;               // 带头结点链表改为 DL->next=DL->next->next
        DL->prior=NULL;
        free(p);                   // 带头结点链表增加语句 DL->data--，不是必须
        return;
    }
    q=p->next;
    while(q!=NULL && q->data!=x) // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q==NULL)                    // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;               // q 脱离链表
    q->next->prior=p
    free(q);                       // 释放 q 所占存储空间
                                   // 带头结点链表增加语句 DL->data--，不是必须
    return;
}
```

- 表中有重复元素值，删除所有目标元素的代码如下。

```
void listDelete2(DlinkList DL, eleType x){
    DLNode *p,*q;
    p=DL;                          // 带头结点链表改为 p=DL->next，p 最终指向待
                                   // 删结点的前驱结点
    if(p->data==x){                // 第一个结点的值为 x
        DL=DL->next;               // 带头结点链表改为 DL->next=DL->next->next，
        DL->prior=NULL;
        free(p);
                                   // 带头结点链表增加语句 DL->data--，不是必须
    }
    q=p->next;
    while(q!=NULL){                // 定位待删结点及其前驱
        while(q!=NULL && q->data!=x){
            p=q;
            q=p->next;
        }
        if(q==NULL)                // 不存在值为 x 的结点
            exit(ERROR);
        else{                      // 找到一个值为 x 的结点
            p->next=q->next;       // q 脱离链表
            q->next->prior=p
            free(q);               // 释放 q 所占存储空间
                                   // 带头结点链表增加语句 L->data--，不是必须
```

```
        }
        q=p->next;              // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
void printList(DlinkList DL){
    DLNode *p;
    p=DL;                       // 带头结点的链表改为 p=DL->next，p 指向待访问结点
    while(p!=NULL){
        printf(p->data);
        p=p->next;
    }
    return;
}
```

（8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(DlinkList DL){
    if(DL==NULL)                // 带头结点的链表改为 if(DL->next==NULL)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
void destroyList(DlinkList DL){
    DLNode *p,*q;
    p=DL;                       // 带头结点的链表改为 p=DL->next，p 指向待删结点
    while(p!=NULL){
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

3）应用场合

相对单向链表，双向链表求前驱比较方便。

5．双向循环链表

双向循环链表的最后一个结点的后继为第一个结点，第一个结点的前驱为最后一个结点。

1）双向循环链表的存储结构

```
typedef  struct  DRNode{
    eleType data;
```

```
    struct  DRNode  *prior;
    struct  DRNode  *next;
}DRLNode, *DRlinkList;
DRlinkList DRLtail;
```

定义类型为 DRlinkList 的指针变量 DRLtail，标识整个链表，指向“最后一个结点”，其后继为头结点，头结点的后继为 DRLtail。对于循环链表来讲，定义尾结点既能表示最后一个结点，又能标识头结点，实现基本操作的算法比较简单。

不带头结点的双向循环链表如图 2.9 所示，带头结点的双向循环链表如图 2.10 所示。

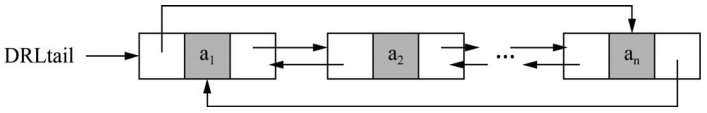


图 2.9　不带头结点的双向循环链表

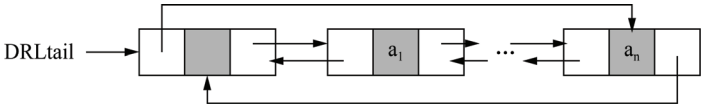


图 2.10　带头结点的双向循环链表

2）双向循环链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(DRlinkList DRLtail){
    DRLtail=NULL;                  // NULL 为空指针，可记为∧
    return;
}
```

② 带头结点的代码如下。

```
void initList2(DRlinkList DRLtail){
    DRLtail=(DRLNode *)malloc(sizeof(DRLNode));
    if(!DRLtail)                   // 没有分配成功
        exit(OVERFLOW);            // 退出程序，提示溢出
    RLtail->prior=RLtail;
    RLtail->next=RLtail;
    RLtail->data=0;                // 可以利用头结点的值域存放表长，不是必须
    return;
}
```

（2）建立双向循环链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(DRlinkList DRLtail){    // 表头插入
    initList1(DRLtail);
    DRLNode *p;                    // p 为待插入链表的结点
    eleType x;                     // x 为待插入链表的结点的值
    scanf(&x);                     // 读取第一个元素的值
    if (x==EOF)                    // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
```

```
    DRLtail=(DRLNode *)malloc(sizeof(DRLNode));
    RL-tail>data=x;
    RLtail->prior=RLtail;
    RLtail->next=RLtail;                    // RLtail->next 为表头结点
    scanf(&x);                              // 读取第二个元素的值
    while (x!=EOF){                         // EOF 为用户自定义的结束标记值
        p=(DRLNode *)malloc(sizeof(DRLNode));
        p->data=x;
        p->prior=DRLtail
        p->next=DRLtail->next;              // DRLtail->next 为表头结点
        DRLtail->next->prior=p;
        DRLtail->next=p;
        scanf(&x);                          // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList12(DRlinkList DRLtail){   // 表尾插入
    initList1(DRLtail);
    DRLNode *p;                         // p 为待插入链表的结点
    eleType x;
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(DRLNode *)malloc(sizeof(DRLNode)); // 建立含一个结点的链表
    p->data=x;
    p->prior=p;
    p->next=p;
    DRLtail=p;
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(DRLNode *)malloc(sizeof(DRLNode));
        p->data=x;
        p->prior=DRLtail;
        p->next=DRLtail->next;    // p 指向表头 DRLtail->next，为新的表尾
        DRLtail->next=p;          // p 链接至当前表尾 DRLtail 之后
        RLtail->next->prior=p;    // p 链接至当前表头 DRLtail->next 之前
        DRLtail=p;                // p 为新的表尾 DRLtail
        scanf(&x);                // 读取下一个元素的值
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(DRlinkList DRLtail){   // 表头插入
    initList2(DRLtail);
    DRLNode *p;                         // p 为待插入链表的结点
    eleType x;                          // x 为待插入链表的结点的值
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(DRLNode *)malloc(sizeof(DRLNode));
```

```
    p->data=x;
    p->prior=DRLtail;
    p->next=DRLtail;
    DRLtail->prior=p;
    DRLtail->next=p;
    DRLtail=p;                          // DRLtail 指向第一个插入的结点，即表尾
    DRLtail->next->data++;              // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
       p=(DRLNode *)malloc(sizeof(DRLNode));
       p->data=x;
       p->prior=DRLtail->next;          // p 的前驱为表头结点 DRLtail->next
       p->next=DRLtail->next->next;     // p 的后继为首元素结点 DRLtail->next
       DRLtail->next->next->prior=p;
       DRLtail->next->next =p;
       DRLtail->next->data++;           // 可利用头结点的值域存放表长，不是必须
       scanf(&x);                       // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(DRlinkList DRL){ // 表尾插入
    initList2(DRL);
    DRLNode *p;                     // p 为待插入链表的结点
    DRLNode *tail=DRL;              // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                      // 读取第一个元素的值
    if (x==EOF)                     // 如果第一个元素的值为结束标记，退出
       exit(EMPTY);
    p=(DRLNode *)malloc(sizeof(DRLNode));
    p->data=x;
    p->prior=DRLtail;
    p->next=DRLtail;
    DRLtail-> prior =p;
    DRLtail->next=p;
    DRLtail=p;                      // DRLtail 指向第一个插入的结点，即表尾
    DRLtail->next->data++;          // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                      // 读取第二个元素的值
    while (x!=EOF){                 // EOF 为用户自定义的结束标记值
       p=(DRLNode *)malloc(sizeof(DRLNode));
       p->data=x;
       p->prior= DRLtail;
       p->next=DRLtail->next;       // p 指向表头
       DRLtail->next=p;             // p 链接至当前表尾 DRLtail 之后
       DRLtail->next->prior=p;      // p 为表头 DRLtail->next 的前驱
       DRLtail=p;                   // p 为新的表尾 DRLtail
       L->data++;                   // 可以利用头结点的值域存放表长，不是必须
       scanf(&x);                   // 读取下一个元素的值
    }
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

- 不带头结点的代码如下。

```
DRLNode *getElem11(DRlinkList DRLtail,unsigned i){
                                                  // i 为逻辑位置
    DRLNode *p=DRLtail->next;                     // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=DRLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        return p;
    else
        return NULL;
}
```

或

```
void getElem21(DRlinkList DRLtail,unsigned i, DRLNode *e){
                                                  // i 为逻辑位置
    DRLNode *p=DRLtail->next;                     //  p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next){
        p=p->next;
           j++;
    }
    if(i==j)
        e=p;
    else
        e=NULL;
    return;
}
```

- 带头结点的实现可参考前面算法，利用头结点值域存放的元素个数修改程序。

```
DRLNode *getElem12(DRlinkList DRLtail,unsigned i){
                                                  // i 为逻辑位置
    DRLNode *p=DRLtail->next->next;               // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=DRLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        return p;
    else
        return NULL;
}
```

或

```
DRLNode *getElem22(DRlinkList DRLtail,unsigned i, DRLNode *e){
```

```
                                                    // i 为逻辑位置
    DRLNode *p=DRLtail->next->next;                 // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=DRLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        e=p;
    else
        e=NULL;
    return;
}
```

② 获取特定值的结点：getElem*(L,x)，下面代码以不带头结点循环链表为例，带头结点的算法类似。

```
DRLNode *getElem3 (DRlinkList DRLtail, eleType x){
    DRLNode *p=DRLtail->next;     // p 为遍历链表元素结点的指针，带头结点链表
                                  // 改为 dRLNode *p=DRLtail->next->next
    while(p->data!=x && p->next!=DRLtail->next)
        p=p->next;
    if(p->data==x)
        return p;
    else
        return NULL;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素（逻辑位置）插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(DRlinkList DRLtail, unsigned i, eleType x){
                                  // 以不带头结点链表为例
    DRLNode *p,*q;
    q=getElem11(DRLtail, i-1);    // 带头结点链表改为 q=getElem12(DRLtail,i-1)
    if(q==NULL)
        exit(ERROR);
    p=(DRLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->prior=q;
    p->next=q->next;
    q->next->prior=p;
    q->next=p;
    // 带头结点链表可增加语句 DRLtail->next->data++来修正表长，不是必须
    return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(DRlinkList DRLtail, eleType x){ // 以不带头结点链表为例
    DRLNode *p;
    p=(DRLNode *)malloc(sizeof(DRLNode));
    p->data=x;
    p->prior=DRLtail;     // 带头结点链表改为 p->prior=DRLtail->next
```

```
    p->next=DRLtail->next;
                          // 带头结点链表改为 p->next=DRLtail->next ->next
    DRLtail->next->prior=p;
    DRLtail->next=p;      // 带头结点链表改为 DRLtail->next->next=p
            // 带头结点链表可增加语句 DRLtail->next->data++来修正表长，不是必须
    return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(DRlinkList DRLtail){
    DRLNode *p;
    unsigned i=1;
    if(DRLtail==NULL)             // 空表
        return 0;
    p=DRLtail->next;
    while(p->next!=DRLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(DRlinkList DRLtail){
    DRLNode *p=DRLtail->next;     // 可直接 return DRLtail->next->data
    unsigned i=1;
    if(p->next==p)                // 空表
        return 0;
    while(p->next!=DRLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void listDelete1(DRlinkList DRLtail,unsigned i){    // i 为逻辑位置
    DRLNode *p,*q;
    p=getElem11(DRLtail,i-1);     // p 为待删结点的前驱结点
                                  // 带头结点链表改为 p=getElem21(DRLtail,i-1)
    q= getElem11(DRLtail,i);      // q 为待删结点
                                  // 带头结点链表改为 q=getElem21(DRLtail,i);
    if(p==NULL || q==NULL)        // 不存在第 i 个结点
        exit(ERROR);
    p->next=q->next;              // q 脱离链表
    q->next->prior=p;
                                  // 带头结点链表可增加语句
                                  // DRLtail->next->data--，不是必须
```

```
    free(q);                          // 释放 q 所占存储空间
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

- 链表中无重复元素值的代码如下。

```
void listDelete2(DRlinkList DRLtail, eleType x){
    DRLNode *p,*q;
    p=DRLtail->next;                  // p 开始指向首元素结点，最终指向待删结点的前驱
                                      // 带头结点链表改为 p=DRLtail->next->next
    if(p->data==x){                   // 第一个结点的值为 x
        DRLtail->next=p->next;        // 删除首元素结点
        p->next->prior=DRLtail;
                        // 带头结点链表改为 DRLtail->next->next=p->next
                        //  p->next->prior =DRLtail->next
        free(p);
                        // 带头结点链表可增加语句 DRLtail->next->data--，不是必须
        return;
    }
                                      // 第一个结点的值不是 x
    q=p->next;
    while(q!=DRLtail->next && q->data!=x)    // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q==DRLtail->next)              // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;                  // q 脱离链表
    q->next->prior=p;
    free(q);                          // 释放 q 所占存储空间
                        // 带头结点链表增加 DRLtail->next->data--; 不是必须
    return;
}
```

- 表中有重复元素值，删除所有目标元素的代码如下。

```
void listDelete2(DRlinkList DRLtail, eleType x){
    DRLNode *p,*q;
    p=DRLtail->next;                  // p 开始指向首元素结点，最终指向待删结点的前驱
                                      // 带头结点链表改为 p=DRLtail->next->next
    if(p->data==x){                   // 第一个结点的值为 x
        DRLtail->next=p->next;        // 删除首元素结点
        p->next->prior=DRLtail;
                        // 带头结点链表改为 DRLtail->next->next=p->next;
                        // p->next->prior =DRLtail->next;
        free(p);
                        // 带头结点链表可增加语句 DRLtail->next->data--，不是必须
    }
    q=p->next;
    while(q!=DRLtail->next){          // 定位待删结点及其前驱
        while(q!=DRLtail->next && q->data!=x){
            p=q;
            q=p->next;
        }
```

```
        if(q==DRLtail->next)  // 不存在值为 x 的结点
            exit(ERROR);
        else{                 // 找到一个值为 x 的结点
            p->next=q->next;  // q 脱离链表
            q->next->prior=p;
            free(q);          // 释放 q 所占存储空间
                              // 带头结点链表可增加语句 DRLtail->next->data--,
                              // 不是必须
        }
        q=p->next;            // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
void printList(DRlinkList DRLtail){
    DRLNode *p;
    if(DRLtail==NULL)         // 空表即退出，带头结点链表改为 if(DRLtail->next
                              // ==DRLtail)
        exit(ERROR);
    p=DRLtail->next;          // p 指向待访问结点，初值为首结点
                              // 带头结点链表改为 p=DRLtail->next->next
    do{
        printf(p->data);
        p=p->next;
    } while(p!=DRLtail->next)
    return;
}
```

（8）线性表判空：emptyList(L)，为空返回非零值，非空返回 0，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(DRlinkList DRLtail){
    if(DRLtail==NULL)  //空表。带头结点链表改为 if(DRLtail->next ==DRLtail)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void destroyList(DRlinkList DRLtail){
    DRLNode *p,*q;
    p=DRLtail->next;          // p 指向待删结点
                              // 带头结点链表改为 p=DRLtail->next->next
    while(p!=NULL){           // 带头结点链表改为 while(p!=p->next)
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

3）应用场合

从任何一个结点开始均可以很方便地操作其前驱和后继结点。

6．静态链表

1）静态链表的存储结构

静态链表通过结构体数组存储，结构体包括两个成员，一个存放数据元素，另外一个存放该元素后继的地址，其存储示意图如图 2.11 所示，起始地址为 11，结束地址为 0。

…	a_3	a_7	a_5		a_8	a_1		a_2	a_6	a_4	…
…	10	5	9		0	8		1	2	3	6
地址： 0	1	2	3	4	5	6	7	8	9	10	11

图 2.11　静态链表存储结构示意图

存储结构描述如下。

```
#define maxSize 1000
typedef struct{
   eleType data;
   unsigned next;
}Node,SLink;
Slink SL[maxSize];
```

其中，SL[0]表示头结点，SL[0].data 可用来存放静态链表元素个数（eleType 为数值型）、特数值（例如最大值）等，也可不用，SL[0].next 指示第一个数据元素的存储位置。最后一个元素的 next 值为 0，表示静态链表结束。插入、删除操作通过修改元素的 next 值完成，无须移动数据元素。

2）静态链表的基本操作

（1）初始化：initSLink(*SL)，将 SL[0].next 设置为结束标记值 0，其他元素的 next 设置为-1，表示没有存放数据元素。

```
void initSLink(SLink *SL){
    SL[0].next=0;
                          // 如果 eleType 为数值型，也可加入语句 SL[0].data=0，
                          // 表示元素个数为 0，方便用户求表长
   for(unsigned i=1;i<maxSize)
      SL[i].next=-1;
   return;
}
```

（2）在线性表的第 i 个元素之前插入一个元素 x：insertSLink(*SL,i,x)，如果当前静态链表的存储空间没有用完，可进行插入操作，否则退出。算法流程如下。

- 遍历链表存储元素空间的所有 next 域，找到一个值为-1 的 SL[j].next，将 x 存入 SL[j].data。
- 通过 SL 的 next 域遍历静态链表，定位到第 i-1 个元素位置。
- 将 SL[j]插入到 SL[i-1]后面：SL[j].next=SL[i-1].next, SL[i-1].next=j。
- SL[0].data 如果存放元素个数的话，SL[0].data++。

```
void insertSLink(SLink *SL,unsigned i,eleType x){
    for(unsigned j=1;j<maxSize;j++)
        if(SL[j].next==-1)
            break;
    if(j==maxSize-1)
        exit(OVERFLOW);
    SL[j].data=x;
    unsigned h=0,k;                    // h 统计元素个数，k 表示元素的 next 值
    for(k=SL[0].next;h<i-1;k=SL[k].next)
           h++;
    SL[j].next=SL[k].next;             // k 为第 i-1 个元素的位置
    SL[k].next=j;
                                       // SL[0].data 如果存放元素个数的话，
                                       // SL[0].data++
     return;
}
```

（3）求线性表的长度 SListLength(SL)。

```
unsigned SListLength(SLink SL){      // unsigned 也可改为 int
    unsigned i=0,k;                  // i 统计元素个数，k 表示元素的 next 值
    for(k=SL[0].next;k!=0;k=SL[k].next)
        i++;
    return i;
                                     // SL[0].data 中如果存放元素个数的话，该
                                     // 算法仅需一条语句：return SL[0].data
}
```

（4）获取线性表的第 i 个元素：getElem1(SL,i)或 getElem2(SL,i,x)，i 为逻辑位置，取值范围 1～maxSize−1。

```
eleType getElem1(SLink SL, unsigned i){
    unsigned k,j=1;
    for(k=SL[0].next;j<i,k!=0;k=SL[k].next)
        j++;
    if(k==0)
        exit(ERROR);
      return SL[j].data;
  }
```

或

```
void getElem2(SLink SL, unsigned i, eleType *x){
    unsigned k,j=1;
    for(k=SL[0].next;j<i,k!=0;k=SL[k].next)
        j++;
    if(k==0)
        exit(ERROR);
      *x=SL[j].data;
      return;
  }
```

（5）确定 x 是线性表的第几个元素：locateElem(SL,x)。

```
unsigned locateElem(SLink *SL,eleType x){
    unsigned k,j=1;
    for(k=SL[0].next;k!=0,SL[k].data!=x;k=SL[k].next)
```

```
        j++;
    if(k==0)
          exit(ERROR);
      return j;
  }
```

（6）删除线性表的第 i 个元素：SListDelete(*L,i,*x)。

```
void SListDelete(SLink *SL,unsigned i,eleType* x){
    unsigned k,j=1;
    for(k=SL[0].next;j<i-1,k!=0;k=SL[k].next)     // 定位第 i-1 个元素
        j++;
    if(k==0)
        exit(ERROR);
      *x=SL[k].next.data;       // 待删元素值存放于 x
      j=SL[k].next.next;        // 待删元素后继位置 SL[k].next.next 存放于 j
      SL[k].next.next=-1;       // 待删元素后继位置置-1，释放其存储空间
      SL[k].next=j;             // 第 i-1 个元素的后继修改为已删元素的后继
      return;
  }
```

（7）遍历线性表：printSLink(SL)，假设遍历为输出元素的值。

```
void printSLink(SLink SL){
     for(unsigned k=SL[0].next;k!=0;k=SL[k].next)
         printf(SL[k].data);
     return;
  }
```

（8）线性表判空：emptySLink(*SL)。

```
int emptySLink(SLink *SL){
    return !SL[0].next; // SL[0].next=0 为空表，返回真；否则返回假
  }
```

3）应用场合

静态链表适用于没有指针类型的编程环境下需要使用链表的场合。

2.4 栈

2.4.1 定义

栈只能在一个固定端进行插入和删除操作的线性表，固定端称为栈顶，不能进行操作的一端称为栈底。

栈具有“后进先出”特性，即最后进入栈的元素最先出栈，是一种运算受限于一端的线性表。

当栈中没有任何元素时，称为栈空；栈的存储空间用完时，称为栈满。栈空栈满的

条件依赖不同存储结构。

入栈和出栈是栈的基本操作，通过修改栈顶指针完成，如图 2.12 所示。

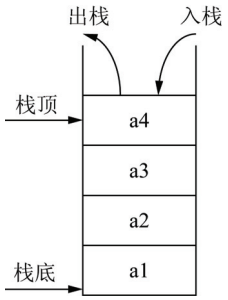


图 2.12　栈的基本操作示意图

栈的操作过程中需注意以下几点。

（1）栈底和栈顶为相对概念，当栈底的存储地址处于栈的小地址端，则入栈时栈顶指针增加，出栈时栈顶指针减小；反之，当栈底的存储地址处于栈的大地址端，则入栈时栈顶指针减小，出栈时栈顶指针增加。机器不同，约定不同。

（2）栈顶指针可能指向栈顶元素存放的位置，也可能指向新栈顶元素可以存放的位置。

（3）栈空、栈满的判定和存储结构的约定有关。

（4）数据元素的所有可能出栈顺序需要熟练掌握。

栈的抽象数据类型 ADT 描述如下。

```
ADT stack{
     数据对象 D: D={ ai| ai|∈eleSet, i=1,2,…,n, n≥0}
     数据关系 R: R={< ai-1, ai > | | ai-1, ai∈D, i=2,…,n, n≥0}
     约定 an 为栈顶，a1 为栈底。
     基本操作有如下几种。
          void initStack(*s)          // 初始化堆栈，构造空栈
          unsigned emptyStack(s)      // 判栈空，栈空返回 1，否则返回 0
          unsigned fullStack(s)       // 判栈满，栈满返回 1，否则返回 0
          void push(*s,x)             // 入栈，将 x 压入堆栈，形成新的栈顶
          void pop(*s,*x)             // 出栈，将栈顶元素出栈并存入 x，形成新栈顶
          void getTop(s,*x)           // 将栈顶元素存入 x，不出栈
} ADT stack
```

2.4.2　存储结构

1．顺序存储结构

1）顺序存储结构的基本原理

利用地址连续的存储空间依次存放从栈底到栈顶的所有数据元素，称为顺序栈，可通过一维数组实现，见图 2.13，图中假设栈空间为 4 个存储单元。

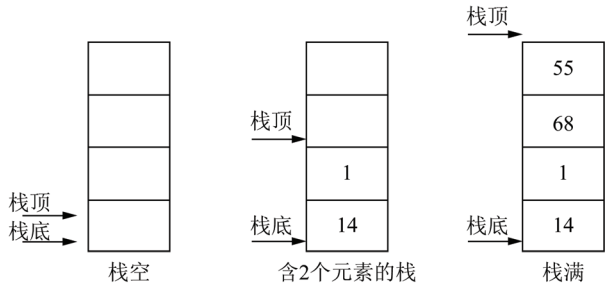


图 2.13　栈的顺序存储结构

顺序栈的描述如下。

```
#define maxStackSize 1000
typedef  struct
{
    eleType data[maxStackSize];     // 存放数据元素
    int top;                        // 指示栈顶位置
}SqStack;
```

2）顺序存储结构的基本操作

（1）初始化栈：initStack(*s)，约定入栈前先修改栈顶指针，所有栈顶指针初值为-1。

```
void initStack(SqStack *s){
    if(!(s=(SqStack *)malloc(sizeof(SqStack))))
        exit(ERROR);
    s->top=-1;
    return;
}
```

（2）判栈空：emptyStack(s)。

```
unsigned emptyStack(SqStack s){
    return s.top==-1?1:0;  // 当 s.top 为-1 时，返回 1，表示栈空为真；否则返回 0
}
```

（3）判栈满：fullStack(s)。

```
unsigned fullStack(SqStack s){
    return s.top== maxStackSize-1?1:0;
}
```

（4）入栈：push(*s,x)。

```
void push(SqStack*s,eleType x){
    if(fullStack(s))
        exit(OVERFLOW);
    s->top++;
    s[s->top]=x;
    return;
}
```

（5）出栈：pop(*s,*x)。

```
void pop(SqStack *s,eleType *x){
```

```
    if(emptyStack(s))
        exit(EMPTY);
    *x=s[s->top--];
    return;
}
```

（6）取栈顶元素：getTop(s,*x)。

```
void getTop (SqStack *s,eleType *x){
    if(emptyStack(s))
        exit(EMPTY);
    *x=s[s->top];
    return;
}
```

2．链式存储结构

1）链式存储结构的基本原理

如果栈中数据元素个数不确定，可以选用链式存储结构，元素入栈时为其申请存储空间，栈中各元素的存储单元地址不必连续，如图 2.14 所示。

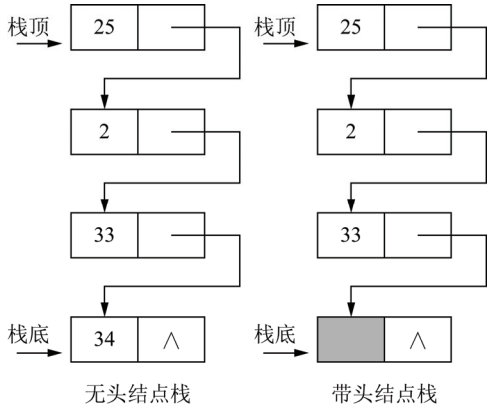


图 2.14　栈的链式存储结构

链栈的描述如下。

```
typedef  struct  Node
{
    eleType  data;           // 存放数据元素
    struct  Node  *next;     // 指示栈中当前元素后继的存放位置
}stackNode,*linkStack;
linkStack top;               // top 为栈顶指针，为表头结点，方便栈的操作实现
```

2）链式存储结构的基本操作，以不带头结点栈为例

（1）初始化栈：initStack(*top)，约定入栈前先修改栈顶指针，所有栈顶指针初值为−1。

```
void initStack(linkStack top){
    top=NULL;
    // 带头结点栈为 if(!(top=(stackNode *)malloc(sizeof(stackNode))))
    // exit(ERROR)
    return;
}
```

（2）判栈空：emptyStack(top)。

```
unsigned emptyStack(linkStack top){
    return !top; // 当 top 为 NULL 时，返回 1，表示栈空为真；否则返回 0
}
```

（3）入栈：push(*top,x)。

```
void push(linkStack top,eleType x){
    stackNode *s;
    s=(stackNode *)malloc(sizeof(stackNode));
    s->data=x;
    s->next=top;
    top=s;
    return;
}
```

（4）出栈：pop(*top,*x)。

```
void pop(linkStack top,eleType *x){
    stackNode *s;
    if(emptyStack(top))
        exit(EMPTY);
    *x=top->data;
    s=top;
    top=top->next;
    free(s);
    return;
}
```

（5）取栈顶元素：getTop(top,*x)。

```
void getTop (linkStack top,eleType *x){
    if(emptyStack(top))
        exit(EMPTY);
    *x=top->data;
    return;
}
```

2.4.3 应用

栈在编程中的用途很广，下面讲述常用的两种情况。

1．符号匹配

程序中经常用到一些成对出现的符号，例如括号，对其进行是否成对的检验是基本的编译问题，可利用栈的特点进行检测。每扫描到一个左括号，将其入栈，扫描到右括号，将其出栈。扫描结束，若栈非空，返回 0，说明左右括号个数不一致，否则返回 1，说明二者个数匹配。

（1）顺序栈算法描述如下。

```
unsigned  signMatch(){
    SqStack s;
    char ch,t;
```

```
    initStack(&s);
    while((ch=getchar()!=EOF)){
        if(ch==' (')
            push(&s,ch);
        if(ch==' ) '){
            if(emptyStack(s))
                return 0;
            pop(&s,&t);    // 也可以用 pop(&s,&ch)，这样可以省略变量 t
        }
    }
    if(emptyStack(s))
        return 1;
}
```

（2）链栈算法描述如下。

```
unsigned signMatch(){
    linkStack top;
    char ch;
    initStack(top);
    while((ch=getchar()!=EOF)){
        if(ch==' (')
            push(top,ch);
        if(ch==') '){
            if(emptyStack(top))
                return 0;
            pop(top,&t);    // 也可以用 pop(top,&ch)，这样可以省略变量 t
        }
    }
    if(emptyStack(top))
        return 1;
}
```

2．整数进制转换

以十进制转为八进制为例，其他进制的转换类似。例如将一个十进制数字 n 转换为八进制数，用 n 除以 8 取余数，所有余数倒序输出即为对应的八进制数。

（1）顺序栈算法描述如下。

```
void ten2Eight(unsigned n){
    SqStack s;
    unsigned x;
    initStack(&s);
    while(n){
        push(&s,n%8);
        n/=8;
    }
    while(!emptyStack(s)){
        pop(&s,&x);
        printf("%u",x);
    }
    return;
}
```

（2）链栈算法描述如下。

```
void ten2Eight(unsigned n){
```

```
    linkStack top;
    unsigned x;
    initStack(top);
    while(n){
        push(top,n%8);
        n/=8;
    }
    while(!emptyStack(top)){
        pop(top,&x);
        printf("%u",x);
    }
    return;
}
```

2.5 队　　列

2.5.1 定义

队列是只能在一个固定端进行插入操作、另外一端进行删除操作的线性表，能够进行插入操作的端称为队尾，能够进行删除操作的端称为队头。

队列具有“先进先出”特性，即最先进入队列的元素也最先出队，是一种运算受限的线性表。

当队列中没有任何元素时，称为队空；队列的存储空间用完时，称为队满。队空队满的条件依赖不同存储结构。

入队和出队是队列的基本操作，分别通过修改队尾指针和队头指针完成，如图 2.15 所示。

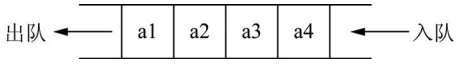


图 2.15　队列的基本操作示意图

队列的使用过程中要注意以下几点。

（1）队头指针可能指向队首元素存放的位置，也可能指向队首元素位置的前一个位置。

（2）队尾指针可能指向队尾元素存放的位置，也可能指向队尾元素位置的下一个位置。

（3）队空、队满的判定和存储结构的约定有关。

（4）和堆栈结合，数据元素的所有可能出栈、出队顺序需要熟练掌握。

队列的抽象数据类型 ADT 描述如下。

```
ADTquene{
    数据对象 D：D={ ai| ai|∈eleSet, i=1,2,…,n, n≥0}
    数据关系 R：R={< ai-1, ai > | | ai-1, ai∈D, i=2,…,n, n≥0}
    约定 a1 为队头，an 为队尾。
    基本操作如下。
        void initQuene(*q)            // 初始化队列，构造空队
```

```
        unsigned emptyQuene(*q)        // 判断队是否空，队空返回 1，否则返回 0
        unsigned fullQuene(*q)         // 判断队是否满，队满返回 1，否则返回 0
        void enQuene(*q,x)             // 入队，将 x 存放到队尾后面，形成新的队尾
        void deQuene(*q,*x)            // 出队，将队首元素出队并存入 x，形成新的队首
        unsigned lengthQuene(*q)       // 返回队列元素的个数
} ADT queen
```

2.5.2　存储结构

1．顺序存储结构

1）顺序存储结构的基本原理

利用地址连续的存储空间依次存放从队首到队尾的所有元素，称为顺序队列，可通过一维数组实现，见图 2.16，假设队列空间为 4 个存储单元。

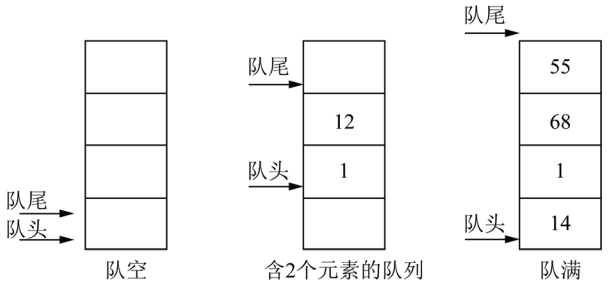


图 2.16　队列的顺序存储结构

顺序队列的描述如下。

```
#define maxQueneSize 1000
typedef struct
{
     eleType data[maxQueneSize];      // 存放数据元素
     int front;                       // 指示队头元素存放位置
     int rear;                        // 指示队尾元素存放位置的下一个位置
}SqQuene;
```

由于队列的基本操作入队和出队同方向修改队头指针和队尾指针，即同加或同减，随着入队、出队操作的随机进行，可能出现假溢出现象，即队尾已达最大值，无法入队，但队列中仍有空闲单元的情况。如图 2.17 所示，队列共占据 4 个存储单元，目前队列仅有一个元素 55，但是由于 rear 已经超出顺序队范围，无法进行入队操作。其中 base 为顺序队基址。

解决假溢出的办法是构建循环队列，如图 2.18 所示。入队时队尾指针的变化为

```
rear=(rear+1)% maxQueneSize;
```

出队时队头指针的变化为

```
front=(front+1)% maxQueneSize;
```

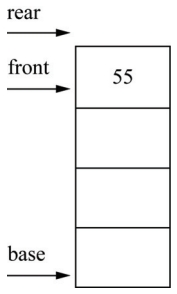


图 2.17　队列的假溢出现象

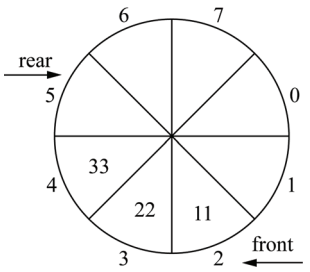


图 2.18　循环队列

2）循环队列的基本操作

（1）初始化循环队列：initQuene(*q)，约定入队前先修改队首指针，所以队首和队尾指针初值为 0。

```
void initQuene(SqQuene *q){
    if(!(q=(SqQuene *)malloc(sizeof(SqQuene))))
        exit(ERROR);
    q->front=q->rear=0;
    return;
}
```

（2）判循环队列空：emptyQuene(*q)。

```
unsigned emptyQuene(SqQuene q){
    if(q->rear==q->front)          // 循环队列为空的条件为 q->rear==q->front
        return 1;
    return 0;
}
```

（3）判循环队列满：fullQuene(*q)。

```
unsigned fullQuene(SqQuene *q){
    if((q->rear+1) % maxSize==q->front)
     // 循环队列满的条件为 (q->rear+1) % maxSize==q->front，即队尾追上队头
        return 1;
    return 0;
}
```

（4）入队：enQueue(*q,x)。

```
void enQueue(SqQueue *q,eleType x){
    if(fullQuene(q))
        exit(OVERFLOW);
    q->data[(q->rear) % maxQueueSize] =x;
    q->rear=(q->rear+1) % maxQueueSize;
    return;
}
```

（5）出队：deQueue(*q,*x)。

```
void deQueue(SqQueue *q,eleType *x){
    if(emptyQuene(q))
        exit(EMPTY);
    *x=q->data[q->front];
```

```
    q->front=(q-> front+1) % maxQueueSize;
    return;
}
```

（6）求队列长度：lengthQuene(*q)。

```
unsigned deQueue(SqQueue *q){
    return (q->rear-q->front+maxQueueSize) % maxQueueSize;
}
```

2．链式存储结构

1）链式存储结构的基本原理

如果队列中数据元素个数不确定，可以选用链式存储结构，元素入队时为其申请存储空间，出队时释放其所占存储单元，队中各元素的存储单元地址不必连续，如图 2.19 所示。

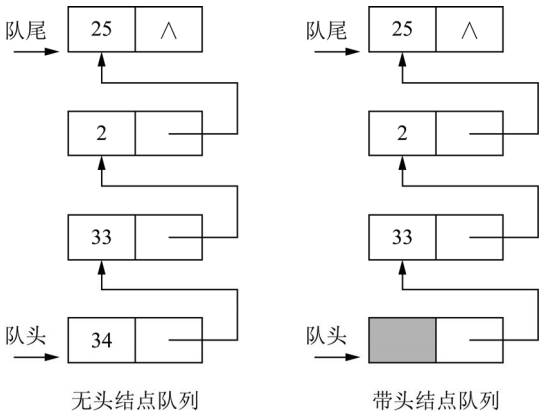


图 2.19　队列的链式存储结构

队列的链式存储结构描述如下。

```
typedef  struct  qNode{
    eleType  data;                  // 存放数据元素
    struct  qNode  *next;           // 指示队列中当前元素后继的存放位置
}queneNode;
typedef  struct {
    queneNode  *front;              // 存放队头指针
    queneNode  *rear;               // 存放队尾指针
}linkQuene;
```

2）链式存储结构的基本操作

（1）初始化队列：initQuene(*q)，创建带头结点的空队。

```
void initQuene(linkQuene *q){
    queneNode *s;
    if(!q=(linkQuene *)malloc(sizeof(linkQuene)))
        exit(ERROR);
    if(!s=(queneNode *)malloc(sizeof(queneNode)))
        exit(ERROR);
    s->next=NULL;
```

```
    q->front=q->rear=s;
    return;
}
```

（2）判队空：emptyQuene(*q)。

```
unsigned emptyQuene(linkQuene *q){
    return q->front==q->rear;
}
```

（3）入队：enQuene(*q,x)。

```
void enQuene ( linkQuene *q, eleType  x){
    queneNode *s;
    if(!s=(queneNode *)malloc(sizeof(queneNode)))
        exit(ERROR);
    s->data=x;
    s->next=NULL;
    q->rear->next=s;                    // s 链至队尾之后
    q->rear=s;                          // s 为新的队尾
    return;
}
```

（4）出队：deQuene(*q,*x)。

```
void deQuene ( linkQuene  *q, eleType  *x) {
    queneNode *s= q->front->next;
    if( emptyQuene (q) )
        exit(EMPTY);
    *x=s->data;
    q->front->next = s->next;
    free(s);
    if (q->front->next == NULL)         // 队空
        q->rear= q->front;
    return;
}
```

2.5.3 应用

下面以键盘输入来讲述队列的应用

键盘缓冲区用来暂存未处理的输入码，为“循环队列”结构。缓冲区满时，按键无效。

假设某程序包含两个进程，其中一个进程在屏幕上连续显示字符“Process A”，同时程序不断检测键盘是否有输入。如果有，则读入用户输入的字符并存入输入缓冲区。在用户输入过程中，输入的字符并不立即显示在屏幕上。输入“$”表示第一个进程结束。

另外一个进程从缓冲区中读取输入的字符并显示在屏幕上。

第二个进程结束后，程序又进入第一个进程，重新显示字符“Process A”，同时用户可以继续输入字符，直到输入“#”键，结束第一个进程，同时整个程序结束。

```
void simKeyBuffer(){                    // 模拟键盘的循环输入缓冲区
    char ch1,ch2;
```

```
    SqQueue  q;
    initQueue (&q);                    // 初始化队列
    for( ; ; ){
        for( ; ; ){                    // 第一个进程代码开始
            printf("Process A\n");
            if (hitKeyboard() ){       // 有按键
                ch1=readChar();        // 读入一个字符
                if ( fullQuene(q) ) {
                    printf("输入缓冲区已满，第一个进程被迫中止。\n");
                    break;             // 第一个进程非正常中止
                }
                enterQueue (&q,ch1);
            }
            if(ch1=='$' || ch1=='#')
                 break;                // 第一个进程正常结束
        }                              // 第一个进程代码结束
        while (!emptyQuene(q)) {       // 第二个进程代码开始
            deQueue (&q,&ch2);
            putchar(ch2);              // 按照输入顺序，输出输入缓冲区中的字符
        }
        if(ch1=='#')
              break;                   // 整个程序结束
        else
            ch1=' ';                   // ch1 赋值为非结束标记（$ 或 #），程序继续
    }
}
```

2.6 特 殊 矩 阵

矩阵是在科学工程计算中常见的数学模型之一，m 行 n 列排列有 m×n 个类型相同的数据元素。由于计算机的存储空间是线性的，所以一般程序设计语言通过一维数组存放二维的矩阵，有行优先（存完第 i 行的 n 个数据元素再存储第 i+1 行的 n 个数据元素）和列优先（存完第 i 列的 m 个数据元素再存储第 i+1 列的 m 个数据元素）两种存储方式。

矩阵通过二维数组 a[m][n]以行优先方式存放，其数组元素 a[i][j] 的地址计算公式如下：

```
a[i][j].addr= a[0][0].addr+(i*n+j)*sizeof(eleType) i∈[0,m-1]，j∈[0,n-1]
```

a[i][j]在一维存储空间的相对位置为第 i*n+j 个结点。

列优先方式存放时，数组元素 a[i][j] 的地址计算公式如下：

```
a[i][j].addr= a[0][0].addr+(j*m+i)*sizeof(eleType) i∈[0,m-1]，j∈[0,n-1]
```

a[i][j]在一维存储空间的相对位置为第 j*m+i 个结点。

当 m 和 n 较大时，需要占用大量的连续存储空间。但是实际应用中存在一些特殊的

矩阵，其中的数据元素值的分布有规律，或尽管矩阵包含数据元素非常多，但是其中不同值的元素（例如非 0 元素）很少。特殊矩阵存储时可为多个值相同的元素分配一个存储空间，对零元不分配空间，以降低矩阵对存储容量的需求。本节详细讲述特殊矩阵的存储方式及应用。

图 2.20 为示例矩阵，i 和 j 表示逻辑位置，i∈[1,m]，j∈[1,n]，m=n，图中 aij 对应的数组元素为 a[i-1][j-1]。其中，图 2.20（a）为一般矩阵；图 2.20（b）为对称矩阵，数据元素的值沿对角线对称出现；图 2.20（c）为上三角矩阵，对角线及其上方的元素值没有规律，但对角线以下所有数据元素具有相同的值；图 2.20（d）为下三角矩阵，对角线及其下方的元素值没有规律，但对角线以上所有数据元素具有相同的值；图 2.20（e）为对角矩阵，对角线及其同列最相邻的上下行两个元素值没有规律，其余数据元素具有相同的值；图 2.20（f）为稀疏矩阵，不同值的数据元素分布没有规律，但数量很少。

$$\begin{bmatrix} a11 & a12 & \cdots & a1n \\ a21 & a22 & \cdots & a2n \\ \cdots & \cdots & \cdots & \cdots \\ an1 & an2 & \cdots & ann \end{bmatrix}$$

（a）一般矩阵

$$\begin{bmatrix} a11 & a12 & \cdots & a1n \\ a12 & a22 & \cdots & a2n \\ \cdots & \cdots & \cdots & \cdots \\ a1n & a2n & \cdots & ann \end{bmatrix}$$

（b）对称矩阵

$$\begin{bmatrix} a11 & a12 & \cdots & a1n \\ m & a22 & \cdots & a2n \\ \cdots & \cdots & \cdots & \cdots \\ m & m & \cdots & ann \end{bmatrix}$$

（c）上三角矩阵

$$\begin{bmatrix} a11 & m & \cdots & m \\ a21 & a22 & \cdots & m \\ \cdots & \cdots & \cdots & \cdots \\ an1 & an2 & \cdots & ann \end{bmatrix}$$

（d）下三角矩阵

$$\begin{bmatrix} a11 & a12 & & \\ a12 & a22 & \cdots & \\ & \cdots & \cdots & an-1n \\ & & ann-1 & ann \end{bmatrix}$$

（e）对角矩阵

$$\begin{bmatrix} a11 & m & \cdots & m \\ a21 & m & \cdots & a2n \\ \cdots & \cdots & \cdots & \cdots \\ an1 & m & \cdots & m \end{bmatrix}$$

（f）稀疏矩阵

图 2.20　矩阵示意图

2.6.1　对称矩阵

n 阶对称矩阵，其数据元素满足 $a_{ij}=a_{ji}$，i, j∈[1,n]，存储于二维数组 a[n][n]，其对应数组元素为 a[k][h]，其中 k=i-1，h=j-1，(k,h∈[0,n-1])，压缩存储时对称的两个相同的值元素只存储一个，则第 i 行仅存储 i 个元素（假设存储下三角的数据元素），可以极大降低矩阵对存储空间的需求。n*n 个元素需要的存储结点（数组元素）数为：

$$1+2+\cdots+n=n(n+1)/2$$

图 2.21 为对称矩阵示意图，其中左图为矩阵的所有数据元素，右图为需要存储的数据元素。

$$\begin{bmatrix} a11 & a12 & \cdots & a1n \\ a21 & a22 & \cdots & a2n \\ \cdots & \cdots & \cdots & \cdots \\ a1n & a2n & \cdots & ann \end{bmatrix} \qquad \begin{bmatrix} a11 & & & \\ a12 & a22 & & \\ \cdots & \cdots & \cdots & \\ an1 & an2 & \cdots & ann \end{bmatrix}$$

图 2.21　对称矩阵示意图

正常存储需要 n^2 个结点，压缩存储后减少了将近一半的存储量。

对称矩阵的数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先的方式压缩存储，其对应数组元素 a[i][j]的地址计算公式如下（ i∈[0,n−1]，j∈[0,n−1] ）。

```
a[i][j].addr= a[0][0].addr+(i*(i+1)/2+j)*sizeof(eleType)      i≥j
a[i][j].addr= a[0][0].addr+(j*(j+1)/2+i)*sizeof(eleType)      i<j
```

i≥j 时，a[i][j]在一维存储空间的相对位置为第 i*(i+1)/2+j 个结点；i<j 时，a[i][j]在一维存储空间的相对位置为第 j*(j+1)/2+i 个结点。

列优先方式存放时的公式请考生自行推算。

2.6.2　三角矩阵

n 阶三角矩阵有上三角矩阵和下三角矩阵之分，分别表示主对角线以下或以上的元素值相同，其他数据元素不同的矩阵。图 2.22 为上三角矩阵示意图，其中左图为上三角矩阵，右图为上三角矩阵需要存储的数据元素，其中左下方 n(n−1)/2 个具有相同值 m 的数据元素在存储时仅占 1 个数组元素空间。

$$\begin{bmatrix} a11 & a12 & \cdots & a1n \\ m & a22 & \cdots & a2n \\ \cdots & \cdots & \cdots & \cdots \\ m & m & \cdots & ann \end{bmatrix} \qquad \begin{bmatrix} a11 & a12 & \cdots & a1n \\ & a22 & \cdots & a2n \\ & & \cdots & \cdots \\ & & & ann \end{bmatrix}$$

图 2.22　上三角矩阵示意图

图 2.23 为下三角矩阵示意图，其中左图为下三角矩阵，右图为下三角矩阵需要存储的数据元素，其中右上方 n(n−1)/2 个具有相同值 m 的数据元素在存储时仅占 1 个数组元素空间。

$$\begin{bmatrix} a11 & m & \cdots & m \\ a21 & a22 & \cdots & m \\ \cdots & \cdots & \cdots & \cdots \\ an1 & an2 & \cdots & ann \end{bmatrix} \qquad \begin{bmatrix} a11 & & & \\ a21 & a22 & & \\ \cdots & \cdots & \cdots & \\ an1 & an2 & \cdots & ann \end{bmatrix}$$

图 2.23　下三角矩阵示意图

n*n 个元素需要的存储结点即数组元素的个数为：

$$1+(1+2+\cdots+n)=1+n(n+1)/2$$

比对称矩阵的压缩存储多一个元素。正常存储需要 n^2 个结点，压缩存储减少了将近一半的存储量。

1．上三角矩阵存储数据元素对应数组下标的计算

上三角矩阵 a_{n*n} 的数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先方式压缩存储，其对应数组元素为 a[i][j]。

存储时，首先保存上三角的所有元素，最后保存等值元素 m。

（1）上三角矩阵（i≤j）的所有元素 a[i][j]的下标计算。

数组的第 0 行需要存储 n 个元素，第 1 行需要存储 n−1 个元素……第 i 行需要存储 n−i 个元素。

数组的第 i 行前面共 i−1 行，需要存储的数据元素个数如下：

$$n+(n-1)+(n-2)+\cdots+(n-i+1)= i*(2*n-i+1)/2$$

数组的第 i 行在 a[i][j]前面的数据元素有 j−i 个。即 a[i][j]之前需要存储的数据元素总数为 i*(2*n−i+1)/2+j−i，对应一维存储空间的数组下标范围为 0～i*(2*n−i+1)/2+j−i−1。所以 a[i][j] 在一维存储空间的数组下标为 i*(2*n−i+1)/2+j−i。其地址计算公式如下(i∈[0,n−1]，j∈[0,n−1])：

$$a[i][j].addr= a[0][0].addr+(i*(2*n-i+1)/2+j-i)*sizeof(eleType)\quad i\leqslant j$$

（2）矩阵右下方（i>j）等值元素 m 的下标计算。

上三角的数据元素个数同对称矩阵，为 n*(n+1)/2 个，对应一维存储空间的数组下标范围为 0～n*(n+1)/2−1。所以等值元素 m 的数组下标为 n*(n+1)/2。

$$a[i][j].addr= a[0][0].addr+(n*(n+1)/2)*sizeof(eleType)\ i>j$$

2．下三角矩阵存储数据元素对应数组下标的计算

下三角矩阵的数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先方式压缩存储，其对应数组元素为 a[i][j]。

存储时，首先保存下三角的所有元素，最后保存等值元素 m。

（1）下三角（i≥j）的所有元素对应的 a[i][j]的下标计算。

数组的第 0 行需要存储 1 个元素，第 1 行需要存储 2 个元素……第 i 行需要存储 i+1

个元素。

数组的第 i 行前面共 i−1 行，需要存储的数据元素个数如下：

$$1+2+\cdots+i= i*(i+1)/2$$

数组的第 i 行在 a[i][j]前面的数组元素有 j 个。即 a[i][j]之前需要存储的数据元素总数为 i*(i+1)/2+j，对应一维存储空间的数组下标范围为 0～i*(i+1)/2+j−1。所以 a[i][j] 在一维存储空间的数组下标为 i*(i+1)/2+j。其地址计算公式如下(i∈[0,n−1]，j∈[0,n−1])：

a[i][j].addr= a[0][0].addr+(i*(i+1)/2+j)*sizeof(eleType)　i≥j

（2）矩阵左上方（i<j）等值元素 m 的下标计算。

下三角的数据元素个数同对称矩阵，为 n*(n+1)/2 个，对应一维存储空间的数组下标范围为 0～n*(n+1)/2−1。所以等值元素 m 的数组下标为 n*(n+1)/2。

a[i][j].addr= a[0][0].addr+(n*(n+1)/2)*sizeof(eleType)　i<j

列优先方式存放时的公式请考生自行推算。

2.6.3　对角矩阵

对角矩阵的所有非零元集中在以主对角线为中心的带状区域，如图 2.24 所示。

$$\begin{bmatrix} a11 & a12 & & \\ a21 & a22 & \cdots & \\ & \cdots & \cdots & an-1n \\ & & ann-1 & ann \end{bmatrix}$$

图 2.24　对角矩阵示意图

对角矩阵的非零数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先方式压缩存储，其对应数组元素为 a[i][j]，其存储地址如下。

数组的第 0 行需要存储 2 个元素，第 1 行～第 n−2 行需要存储 3 个元素，第 n−1 行需要存储 2 个元素。所有需要存储的数据元素个数为 2+3*(n−2)+2=3n−2。

非 0 元素 a[i][j] 在一维存储空间的地址计算公式如下(i∈[0,n−1]，j∈[0,n−1]，i==j+1 或 i==j 或 i==j−1)：

a[i][j].addr= a[0][0].addr+ j*sizeof(eleType)　　i=0

a[i][j].addr= a[0][0].addr+(2+3*(n-2) +(j-(n-2)))*sizeof(eleType)

= a[0][0].addr+(2*n+j-2)*sizeof(eleType)　　i=n-1

a[i][j].addr= a[0][0].addr+(2*i+j)*sizeof(eleType)　　i!=0, i!=n-1

其中 i=j+1 表示主对角线下方的非零元素，为第 i 行的首个非零元素（i!=0）；i=j 表示主对角线上的非零元素；i=j−1 表示主对角线上方的非零元素，为第 i 行的最后一个非零元

素（i!= n−1）。

非零元素 a[i][j] 在一维存储空间的数组下标 k 的取值如下。

```
k=j;                    // i=0
k=2*n+j-2;              // i=n-1
k=2*i+j;                // i!=0, i!=n-1
```

2.6.4 稀疏矩阵

1．定义

一个 n×n 矩阵中，设非 0 元素的个数为 t，则 0 元素个数为 n×（n−t）。若 t 远小于 n*n−t），且非 0 元素的分布没规律，这样的矩阵称为稀疏矩阵，如图 2.25 所示。

$$\begin{bmatrix} a11 & \cdots & \cdots \\ \cdots & \cdots & a2n \\ an1 & \cdots & \cdots \end{bmatrix}$$

图 2.25　稀疏矩阵示意图

设 δ=t/(n*n)，一般当 δ≤0.05 时为稀疏矩阵，δ 为稀疏因子。

2．存储结构及应用

为合理利用存储空间，稀疏矩阵一般采用压缩存储。由于非 0 元素的分布没有规律，所以存储数据元素的同时需要存储其位置信息。常见的稀疏矩阵压缩存储方式有三元组和十字链表。

1）三元组

（1）三元组的概念

稀疏矩阵中的每个非 0 元素用三元组(i, j, aij)唯一确定，其中，i 表示元素 aij 所在的行号，j 表示元素 aij 所在的列号。所有非 0 元素的三元组存放于一个一维数组中。如图 2.26 所示，左图为一稀疏矩阵，右图为其三元组存储结构示意图，设三元组存放于数组 a 中。

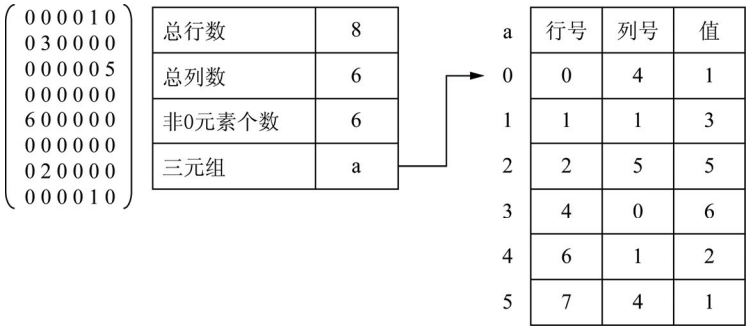


图 2.26　稀疏矩阵的三元组存储

（2）三元组的存储结构

稀疏矩阵的三元组存储需要定义两个类型，一个为三元组结点类型 spNode，存储非 0 元素的行号、列号以及值；另外一个为三元组表类型 spMatrix，存储稀疏矩阵的总行数、总列数、非 0 元素个数，以及对应的三元组。

```
#define spMAX  100
typedef  struct
  {  unsigned  i,j;       // 非 0 元素的行号 i、列号 j
     eleType  x;          // 非 0 元素的值
  }spNode;                // 三元组结点类型
typedef  struct
 {  unsigned  mu,nu,tu;   // 稀疏矩阵的总行数 mu、总列数 nu、非 0 元素个数 tu
    spNode  data[spMAX]; // 三元组
 } spMatrix;              // 三元组表存储类型
```

（3）三元组的基本操作

① 稀疏矩阵的三元组创建。

设稀疏矩阵为 a[M][N]，对应三元组表为 sqA，则创建 a[M][N]的三元组表 sqA 的算法描述如下。

```
#define M 10000
#define N 20000
void creatSpA( spMatrix *sqA, eleType a[M][N]){
   unsigned i,j;
   sqA->mu=M;
   sqA->nu=N;
   sqA->tu=0;
   for(i=0;i<M;i++)
       for(j=0;j<N;i++)
           if( a[i][j]!=0 ) {
               sqA->data[sqA->tu].i= i;
               sqA->data[sqA->tu].j= j;
               sqA->data[sqA->tu].x= a[i][j];
               sqA->tu++;
           }
}
```

② 稀疏矩阵的转置。

求稀疏矩阵 A 的转置矩阵 B，设 A 对应三元组表为 sqA，B 对应三元组表为 sqB。快速方法为先计算每个非零元转置后在三元组中的存储位置，然后遍历三元组，将非 0 元素放入转置后的位置。

- 计算每个非 0 元素转置后在三元组中的存储位置。

设 num[col]表示第 col 列非 0 元素的个数，cpot[col]表示第 col 列中第一个非 0 元素在转置矩阵对应的三元组中的位置。则：

```
cpot[0]=0
cpot[col]=cpot[col-1]+num[col-1]   col>0
```

算法描述如下。

```
unsigned col;
unsigned cpot[sqMAX]={0};
unsigned num[sqMAX]={0};
            // 第一次遍历稀疏矩阵 A 的三元组表，求每列的非 0 元素个数
for(unsigned k=0;k<sqA.tu;k++)
   num[ sqA.data[k].j ]++;
```

```
                // 第二次遍历稀疏矩阵 A 的三元组表，求每列的第一个非 0 元素在稀疏
                // 矩阵 B 的三元表中的位置
for(unsigned k=0;k<sqA.nu;k++)
    cpot[k]= cpot[k-1]+num[k-1];
```

- 遍历三元组，将非 0 元素放入转置后的位置。

```
sqB.mu=sqA.nu;
sqB.nu=sqA.mu;
sqB.tu=sqA.tu;
for(unsigned k=0;k<sqA.tu;k++){
    sqB.data[cpot[sqA.data[k].i]].j=sqA.data[k].i;
    sqB.data[cpot[sqA.data[k].i]].i=sqA.data[k].j;
    sqB.data[cpot[sqA.data[k].i]].x=sqA.data[k].x;
    cpot[i]++;
}
```

求稀疏矩阵 A 的转置矩阵 B 的算法如下。

```
void Atrans2B( spMatrix sqA, spMatrix *sqB ){
    unsigned col;
    unsigned cpot[sqMAX]={0};
    unsigned num[sqMAX]={0};
    for(unsigned k=0;k<sqA.tu;k++)
        num[sqA.data[k].j]++;
    for(unsigned k=0;k<sqA.nu;k++)
        cpot[k]= cpot[k-1]+num[k-1];
    sqB->mu=sqA.nu;
    sqB->nu=sqA.mu;
    sqB->tu=sqA.tu;
    for(unsigned k=0;k<sqA.tu;k++){
       sqB->data[cpot[sqA.data[k].i]].j=sqA.data[k].i;
       sqB->data[cpot[sqA.data[k].i]].i=sqA.data[k].j;
       sqB->data[cpot[sqA.data[k].i]].x=sqA.data[k].x;
       cpot[sqA.data[k].i]++;
    }
}
```

2）十字链表

（1）十字链表的概念

当稀疏矩阵中非 0 元素的个数和位置在操作过程中变化较大时，用十字链表作存储结构的算法实现具有较高的效率。每个非 0 元素用含 5 个域的结点表示，其中 i、j 和 x 的含义和三元组表示相同，分别表示该非 0 元素所在行号、列号和值；right 域用来链接同一行的下一个非 0 元素，down 域用来链接同列的下一个非 0 元素，如图 2.27 所示。

图 2.27　十字链表非零元结点结构

稀疏矩阵中同一行的非 0 元素通过向右的 right 指针链接为一个带表头结点的链表；

同一列的非 0 元素通过向下的 down 指针链接为一个带表头结点的链表。因此，每个非 0 元素既是第 i 行链表中的一个结点，又是第 j 列链表中的一个结点，类似十字交叉，故称十字链表。

十字链表结构如图 2.28 所示。

总行数 mu	总列数 nu	非 0 元素个数 tu
列数组指针 cHead	行数组指针 rHead	

图 2.28　十字链表结构

图 2.29 中，左图为一稀疏矩阵，右图为其十字链表存储结构示意图。

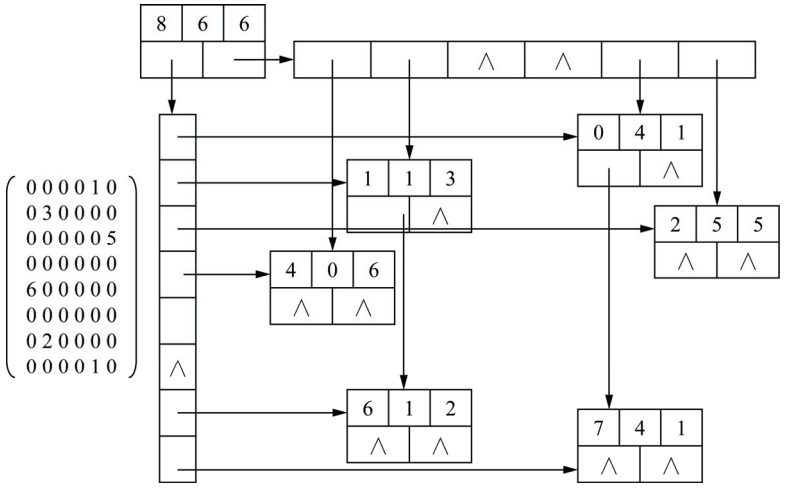


图 2.29　稀疏矩阵的十字链表存储

（2）十字链表的存储结构

稀疏矩阵的十字链表存储需要定义两个类型，一个为结点类型 croNode，存储非 0 元素的行号、列号、非 0 元素的值、同行下一个非 0 元素结点地址、同列下一个非 0 元素结点地址；另外一个为十字链表存储类型 croLinkList，存储稀疏矩阵的总行数、总列数、非 0 元素个数，以及分别指向行指针数组和列指针数组的指针。

```
typedef  struct cNode{
    unsigned  i,j;                  // 非 0 元素的行号 i、列号 j
    eleType  x;                     // 非 0 元素的值
    struct cNode *down, *right;
  }croNode, *cLink;                 // 十字链表结点类型
typedef  struct{
   unsigned  mu,nu,tu;              // 稀疏矩阵的总行数 mu、总列数 nu、非 0 元素个数 tu
   cLink  cHead[N], rHead[M];       // 指向行数组 rHead 和列数组 cHead
 }croLinkList;                      // 十字链表存储类型
```

2.7 串

串是数据元素为字符类型的特殊线性表，通常作为整体参与处理。

2.7.1 基本概念

（1）串：又称字符串，由用双引号括起的 0 个或多个字符组成的有限字符序列。如 str="$s_1s_2\cdots s_n$"，其中 s_i 为字符类型，str 称为串名，n 称为串长，n=0 称为空串。

（2）子串：字符串（主串）中任意多个连续字符所组成的子序列，串中第 1 个字符在主串中的位置，称为该子串在主串中的位置。

2.7.2 存储结构

串的存储结构有以下两种。

1．顺序存储

顺序存储又称顺序串，用一组地址连续的存储单元存储字符串的存储方式，通常高级语言选择用数组。定义如下。

```
#define  MAX_STR_LEN  1000           // 最大串长
typedef  struct{
   char  s[MAX_STR_LEN +1];          // 多申请一个空间存放字符串结束标记
   int    length;                    // 串的实际字符个数
}seqString;
```

顺序存储的优点是简单方便。缺点是 MAX_STR_LEN 尽可能定义得大一些，否则应用中容易益出，丢失有效字符。但过大的 MAX_STR_LEN 会导致存储空间浪费。

2．链式存储

链式存储是动态分配存储空间，定义如下。

```
#define  MAX_STR_LEN  1000           // 最大串长
typedef  struct{
       char *ch;
       int length;                   // 串的实际字符个数
}linkString;
定义变量：linkString  linkS;
申请空间：linkS.ch= (char *)malloc(MAX_STR_LEN+1);
释放空间：fee(linkS.ch);
```

2.7.3 基本操作

本节串的算法以链式存储结构为例。

1．初始化串

给字符串 str 申请空间，并赋初值。

```
void stringInit( linkString  *strD, int  len){
       strD->ch=(char *)malloc(len+1);  // 申请 len+1 个元素的存储空间
                                        // 第 0～len-1 个存储单元存放字符串，第 len
                                        // 个（最后一个）存储单元存放字符串结束
                                        // 标记'\0'
       if(strD->ch==NULL)               // 申请空间失败则提示出错并退出
          exit("申请空间失败，退出");
       for(int i=0;i<=len;i++)
          strD->ch[i]='\0';             // 将字符串的所有字符初始化为 ASCII 码
                                        // 值为 0 的字符
       strD->length=0;                  // 串长置为 0
}
```

2．求串长

求字符串 str 的长度，即串中包含的有效字符个数。

```
int  stringLength ( linkString  *str){
   return  str->length;                 // 返回串长度
}
```

3．拷贝赋值

将字符串 str 赋值给串 strD。

```
void stringCopy ( linkString  *strD,  linkString   *str ){
   int len=stringLength (str);          // 字符串 str 的长度赋给 len
   if( !len ){                          // 如果串长为 0，则字符串 strD 置空串
      strD->ch=NULL;
      strD->length=0;
   }else{                               // 如果字符串 str 不是空串，依次赋值
      strD->ch=(char *)malloc(len+1);    // 申请 len+1 个元素的存储空间
                                        // 第 0～len-1 个存储单元存放字符串,第 len
                                        // 个（最后一个）存储单元存放字符串结束
                                        // 标记'\0'
      if(strD->ch==NULL)                // 申请空间失败则提示出错并退出
         exit("申请空间失败，退出");
      for(int i=0;i<=len;i++)
         strD->ch[i]=str->ch[i];
      strD->length=len;                 // 修改串长
   }
}
```

4．插入

将字符串 strT 插入到串 strD 中 index 开始的位置。

```
void stringInsert( linkString  *strD,  linkString  *strT ,int  index ){
   if( index<0  || index> stringLength (strT)+1)  // 插入位置不合法
      exit("插入位置不合法，退出");
   int lenT=stringLength (strT);             // 字符串 strT 的长度赋给 lenT
   int lenD=stringLength (strD);             // 字符串 strD 的长度赋给 lenD
```

```
    int i;
    strD->ch=(char *)realloc(strD->ch, lenT+lenD+1);
                    // 申请首地址为 strD->ch、lenT+lenD+1 个数据元素所需的存储空间
                    // 多申请的 1 个单元存放字符串结束标记'\0'
                    //  strD 仍然保留原字符串
    if(strD->ch==NULL)                          // 申请空间失败则提示出错并退出
        exit("申请空间失败，退出");
    for(i=lenD;i>=index;i--)
                     // 原串位于区间 [lenD，index ]的字符依次后移 lenT 个字符位置
        strD->ch[i+ lenT]= strD->ch[i];
    for(i=0;i< lenT;i++)                        // 将 strT 所含字符串插入 strD 第
                                                // index～index+ lenT-1 区间
        strD->ch[i+index]= strT->ch[i];
    strD->length= lenT+ lenD;                   // 修改串长
}
```

5．删除若干字符

删除字符串 strD 中 index 位置开始的 n 个字符。

```
void stringDeleteN( linkString  *strD, int  index,int  n ){
    int lenD=stringLength (strD);         // 字符串 strD 的长度赋给 lenD
    if( index<0  || index>= lendD)        // 删除位置不合法
        exit("删除位置不合法，退出");
    if( index+n-1 > = lenD )              // 从 index 位置开始不够 n 个字符
        exit("从 index 位置开始不够 n 个字符，退出");
    for( int  i=index;i< index+n;i++)
                                          // 原串位于区间 [index，index+n-1]
                                          // 的 n 字符依次前移 n 个字符位置
        strD->ch[i-n]= strD->ch[i];
    strD->length= strD->length - n;       // 修改串长
}
```

6．串删除

删除字符串 strD，释放整个串空间。

```
void stringDelete( linkString  *strD ){
    if(strD->ch!=NULL ){
        free(strD->ch);
        strD-> length=0;
    }
}
```

7．串比较

按字典顺序比较字符串 strS 和 strD。

（1）strS > strD，返回 1。

（2）strS == strD，返回 0。

（3）strS < strD，返回−1。

```
int  stringCompare(linkString  strS[ ],linkString  strD[ ] ){
    for( int i=0; i< strS.length && i< strD.length; i++){
        if( strS.ch[i] == strD.ch[i] )
            continue;
```

```
        if( strS.ch[i] > strD.ch[i] )
            return 1;
        if( strS.ch[i] < strD.ch[i] )
            return -1;
    }
    if( strS.length == strD.length )
        return 0;
    if( strS. length > strD.length )
        return 1;
    if( strS.length < strD.length )
        return -1;
}
```

8．串连接

将字符串 strS 连接到串 strD 的后面。

```
void stringConcat( linkString  *strD, linkString  *strS ){
    int lenS=stringLength (strS);          // 字符串 strS 的长度赋给 lenS
    int lenD=stringLength (strD);          // 字符串 strD 的长度赋给 lenD
    strD->ch=(char *)realloc(strD->ch, lenS+lenD+1);
      // 申请首地址为 strD->ch、lenS+lenD+1 个数据元素所需的存储空间
      // 多申请的 1 个单元存放字符串结束标记'\0'
      // strD 仍然保留原字符串
    if(strD->ch==NULL)                     // 申请空间失败则提示出错并退出
       exit("申请空间失败，退出");
    for(i=0;i<= lenS;i++)                  // strS 的字符依次连接到串 strD 的后
                                           // 面，包括字符串结束标记
      strD->ch[i+ lenD]= strS->ch[i];
    strD->length= lenS+ lenD;              // 修改串长
}
```

2.7.4　模式匹配

子串 strT（模式）在主串 strS 中的定位称为串的模式匹配。功能为在 strS 中查找与子串 strT 完全匹配/相同的子串。若查找成功，则返回模式串 strT 的第一个字符在主串 strS 中出现的位置，否则返回−1。

1．一般匹配算法

1）算法思想

一般匹配算法的主要思想如下。

（1）主串 strS 的开始位置 i=0，长度为 lenS；strT 的开始位置 j=0 开始，长度为 lenT。

（2）通过 i++、j++依次匹配 strS 的第 i 个字符和 strT 的第 j 个字符。

（3）遇到第一个不匹配的字符时通过 i=i−j+1（上次开始位置的下一个位置）、j=0 进行回溯。

（4）重复步骤（2）、（3）继续匹配，直到 i==lenS - lenT 并且 strD->ch[i] != strT->ch[j]，或 j==lenT。

（5）若 i==lenS，匹配失败。

（6）若 j==lenT，匹配成功，返回 strT 的第一个字符在主串 strS 中出现的位置 i–j+1（即下标+1）。

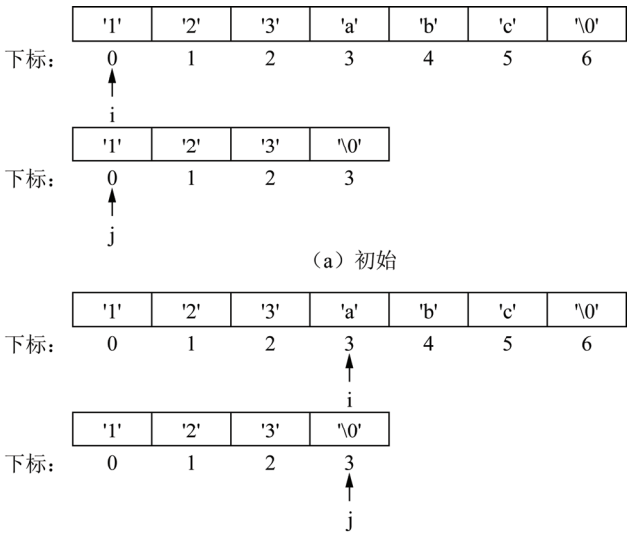
2）伪代码

```
void stringMatch( linkString  *strS, linkString  *strT ){
   int i=0,j=0;
   while( i<= strS->length - strT->length &&  j< strT->length )
      if( strD->ch[i] == strT->ch[j] )
         i++, j++;
      else if( i!= strS->length - strT->length )
            i=i-j+1, j=0;
         else
            return -1;
   if( j==strT->length )
      return  i-j+1; //返回 strT 的第一个字符在主串 strS 中的位置 i-j+1(即下标+1)
   else
      return  -1;
}
```

3）示例

（1）最好匹配情况。

如 strD->ch={"123abc"}，strT->ch={"123"}，匹配过程如图 2.30（a）和图 2.30（b）所示。



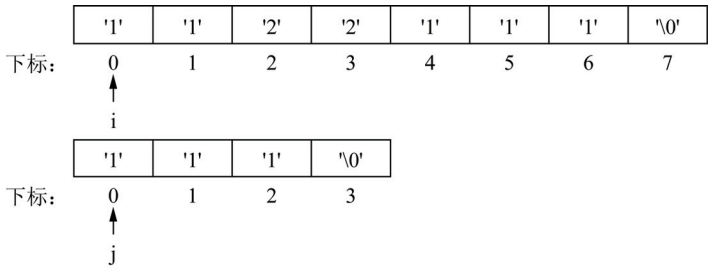
（a）初始

（b）第一趟结束，匹配成功

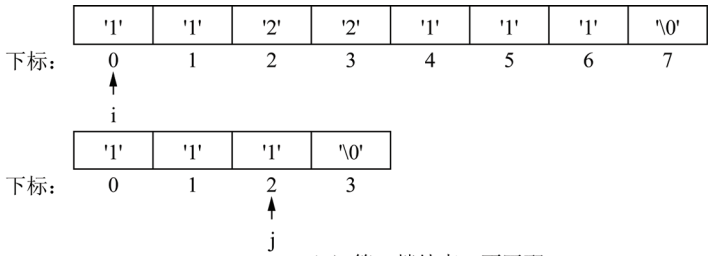
图 2.30　一般匹配算法最好情况示例

（2）最坏匹配情况。

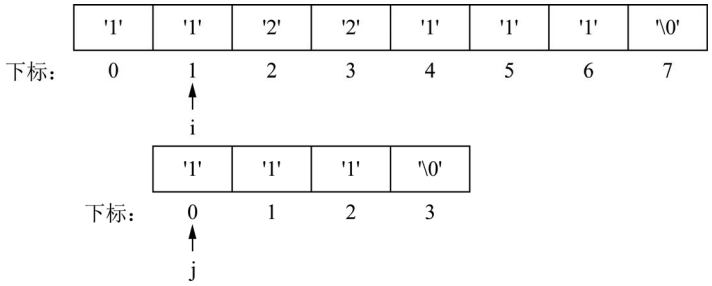
如 strD->ch={"1122111"}，strT->ch={"111"}，匹配过程如图 2.31（a）～图 2.31（h）所示。



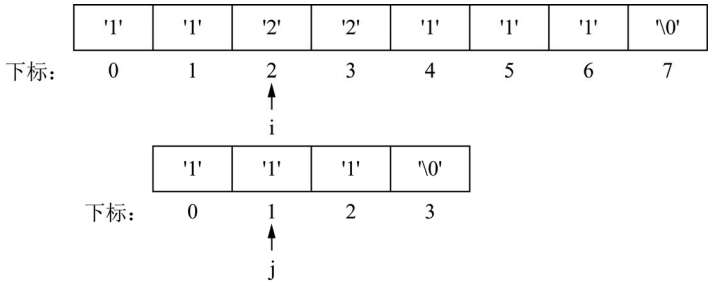
（a）初始



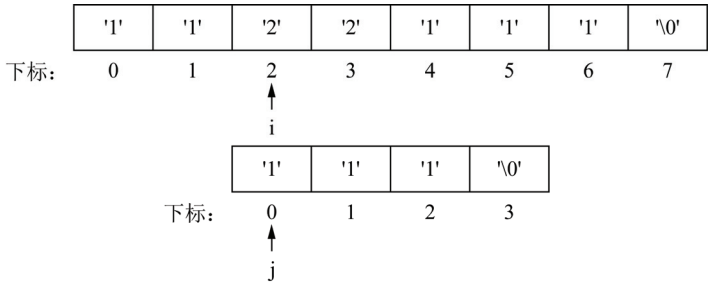
（b）第一趟结束，不匹配



（c）第一趟结束后回溯，第二趟开始

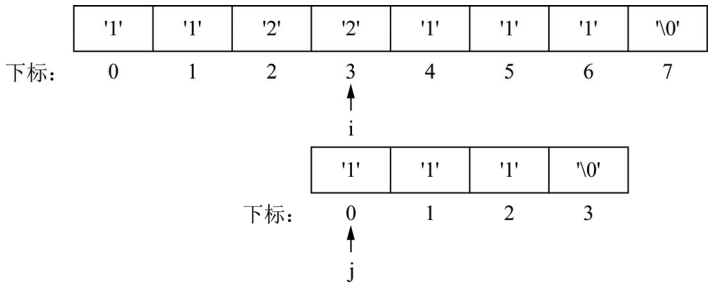


（d）第二趟结束，不匹配

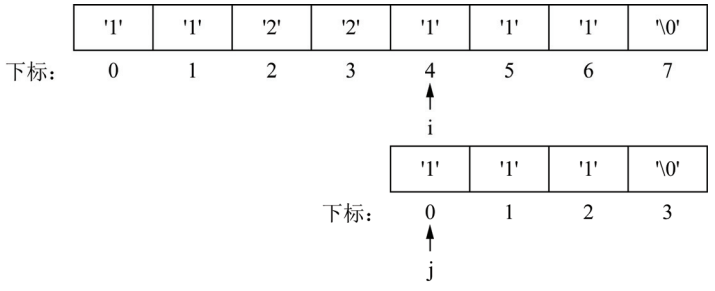


（e）第二趟结束后回溯，第三趟开始

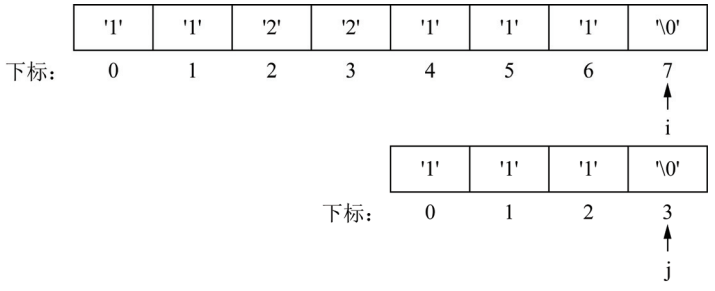
图 2.31　一般匹配算法最坏情况示例



（f）第三趟结束后回溯，第四趟开始



（g）第四趟结束，不匹配，回溯，第五趟开始

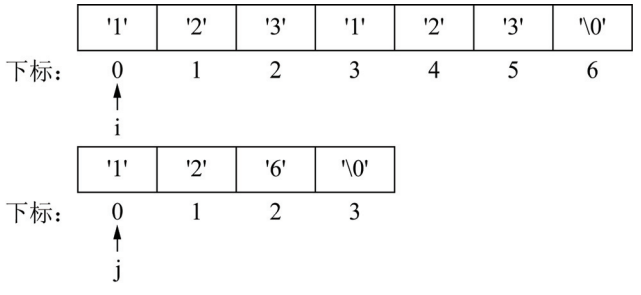


（h）第五趟结束，匹配成功

图 2.31（续）

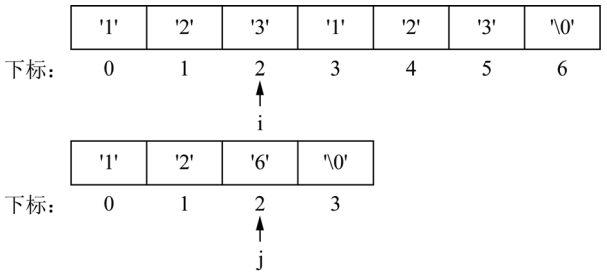
（3）不匹配情况。

如 strD->ch={"123123"}，strT->ch={"126"}，匹配过程如图 2.32（a）～2.32（f）所示。

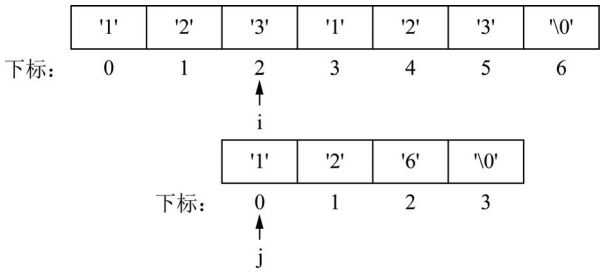


（a）初始

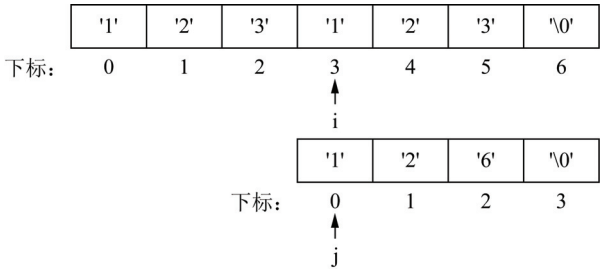
图 2.32　一般匹配算法不匹配情况示例



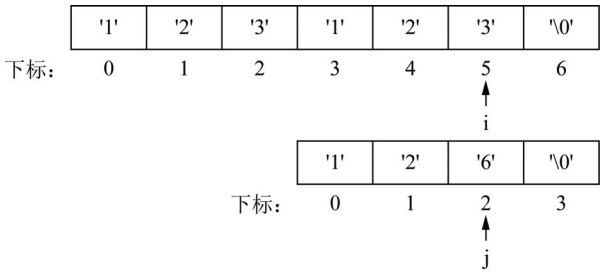
（b）第一趟结束，不匹配



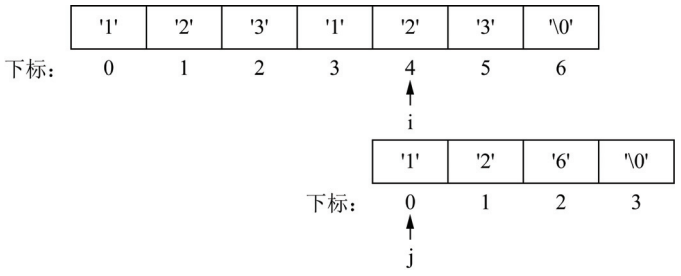
（c）回溯，第三趟开始，不匹配



（d）回溯，第四趟开始



（e）第四趟结束，不匹配



（f）回溯，第五趟开始，i 值过大，不匹配

图 2.32（续）

4）性能分析

图 2.30～图 2.32 给出了一般字符串匹配算法的两大类情况，说明如下。

（1）匹配成功的情况。

图 2.30 为匹配成功的最好情况，即主串的前 lenT 个字符即为 strT 中字符，故返回值为 str->ch T 的第一个字符在主串 strS->ch 中的位置 1（下标 0 加 1），这种情况的时间复杂度为 O(lenT)。

图 2.31 为匹配成功的最坏情况，即主串的前 k-1 个长度为 lenT-1 的字符均和 strT 中的前 lenT-1 个字符匹配，只有最后一个不匹配；并且主串最后的 lenT 个字符与 strT 匹配。故返回值为 strT 的第一个字符在主串 strS 中的位置 lenS-lenT+1（下标 lenS-lenT 加 1）。这种情况的时间复杂度为 O(lenS*lenT)。

（2）匹配不成功的情况

如图 2.32 所示，其中 i 的值为 4，lenT 的值为 3，i+lenT 的值为 7，主串 strS->ch 中不可能包含子串 strT->ch，算法可结束, 时间复杂度为 O(lenS*lenT)。

2．改进的匹配算法——KMP 算法

1）算法思想

分析一般匹配算法可发现，回溯操作部分可以加快，没有必要将主串指针回溯到上次比较开始位置的下一个位置。设子串 strT 包含的有效字符为 strT->ch[0], strT->ch[1], …, strT->ch[i] , …, strT->ch[lenT-1]。

（1）如果当前子串 strT 中待比较的字符为 strT->ch[k]，则 strT->ch[0]～strT->ch[i-1]已经匹配成功，即 strT->ch[0]～strT->ch[k-2]和 strS->ch[i-k]～strS->ch[i-2]对应位置字符相等。

（2）已经得到的匹配结果为 strT->ch[j-k]～strT->ch[j-2]和 strS->ch[i-k]～strS->ch[i-2]对应位置字符相等。

（3）由（1）和（2）可知，strT->ch[**0**]～strT->ch[k-2]和 strT->ch[**j-k**]～strT->ch[j-2]对应位置字符相等。

（4）一般匹配算法回溯时，i 和 j 均回退，其中 j 回退至 0，从第一个字符重复重叠继续比较，所以时间复杂度高。

（5）利用（3） 的结论，KMP 算法回溯时，i 不动，j 回退至 j-k，不仅使 i 不再回退，而且 j 回退缩短，可有效降低时间复杂度为 O(lens+lenT)。

2）KMP 算法的关键——next 值求解

（1）设 next[j]=k，则 next[j]表示当子串中第 j 个字符与主串中对应字符不匹配时，子串指示 j 回退到的下标。

（2）next 值计算

$$\text{next}[j]=\begin{cases}0: & j=1，第一个字符不匹配，直接将主串待比较字符后移一个\\ \max(k\mid 1<k<j \text{ 且 strT->ch[1]～strT->ch[k-1] 和 strT->ch[j-k+1]～strT->ch[j-1] 对应位置字符相等}), & \text{集合不空}\\ 1: & \text{其他}\end{cases}$$

如 strT ={"abaabcac" }的 next 值如下。

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

该方法可以优化，考生在熟悉该计算方法的基础上，可思考更好的求解方法。

3）伪代码

```
void getNext( linkString  *strT, int  next[]) {
    int  next[1]=0,j=1, k=0;
    while (j < strT->length) {
        if ( k== 0 || strT[j] ==strT[k] ){
            ++j;
            ++k;
            next[j] = k;
        } else
            k = next[k];
    }
}
int  KMPMatch( linkString  *strS, linkString  *strT) {
    int  i=1, j=1 ;
    int  next[]=(int *)malloc((strT->string+1)*sizeof(int));
    getNext(strT, next);
    while (i <= strS->length  &&  j <= strT->length) {
        if ( j== 0 || strS[i] ==strT[j] ){
            ++j;
            ++k;
        } else
            j = next[j];
    }
    if(j > strT->length )
        return  i- strT->length;
    else
        return 0;
}
```

2.8 综 合 应 用

下面举例说明线性表的实际应用。

2.8.1 两栈共享空间

由于栈只有一端的地址可以随出入栈操作动态变化，另外一端不变的特点，让两个栈共享一个连续空间，两个栈的栈底分别位于连续空间的两端，做入栈操作时栈顶向连

续空间的中间位置移动，出栈操作时栈顶向连续空间的两端移动，如图 2.33 所示。

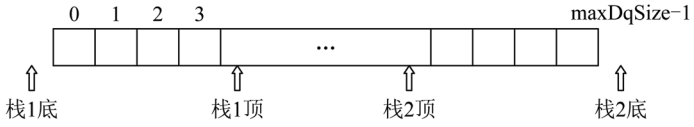


图 2.33　两栈共享空间示意图

两栈共享空间的优点如下。

（1）只要两个栈的数据元素个数之和不大于连续空间的总容量就可进行入栈操作。

（2）每个栈的栈满条件为两个栈顶相等。

（3）只要满足入栈条件，每个堆栈的最大长度可变。

伪代码的实现如下。

```
#define maxDqSize 1000
typedef struct{
   eleType Stack[maxDqSize];
   int top[2];                    // top[0]和 top[1]分别为两个栈的栈顶指示器
}DqStack;
void InitStack(DqStack *S) {
   S->top[0]=-1;
   S->top[1]= maxSize;
}
int push(DqStack *S, eleType x, int i){
   if(S->top[0]+1==S->top[1])   // 共享的栈空间已满
      return -1;
   switch(i){
      case 0: S->top[0]++;
           S->Stack[S->top[0]]=x;
           break;
      case 1: S->top[1]--;
           S->Stack[S->top[1]]=x;
           break;
      default: return -1;
   }
   return 1;
}
int pop(DqStack *S, eleType *x, int i){
   switch(i){
      case 0: if(S->top[0]==-1)
             return -1;
           *x=S->Stack[S->top[0]];
           S->top[0]--;
           break;
      case 1: if(S->top[1]==M)
             return -1;
           *x=S->Stack[S->top[1]];
           S->top[1]++;
           break;
      default: return -1;
   }
   return 1;
}
int emptyDqStack(DqStack *S){
   switch(i){
```

```
        case 0: return S->top[0]==-1;
        case 1: return S->top[1]== maxDqSize;
    }
    return 1;
}
int fullDqStack(DqStack *S){
    return S->top[0]+1==S->top[1];
}
```

2.8.2　多项式求和

多项式运算是线性表的典型应用，本节以多项式求和为例介绍其算法及其实现。数学中的多项式为如下表达式：

$$p = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

其中，p 称为 n 项多项式，其中 a_i 为系数，x 为自变量，i 为指数（$0 \leqslant i \leqslant n$）。本节以带头结点的链表存储多项式，链表中除头结点外的每个结点对应多项式的一项，存储该项的系数和指数，其结点结构定义如下。

```
typedef  struct poly {
    int  coef ;                 // 变量的系数
    int  exp ;                  // 变量的指数
    struct  poly  *next;        // 指到下一结点的指针
} Lpoly;
```

每个多项式由多个结点构成，高指数项（高次幂）的结点在链表头部，低指数项（低次幂）的结点在链表后部。A、B 两个多项式的链表结构如图 2.34 所示。

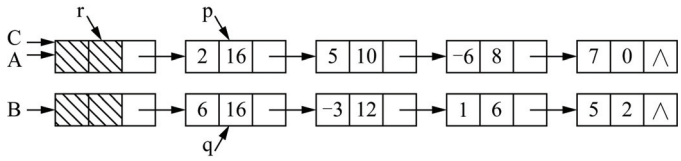


图 2.34　多项式求和初始化

进行加法运算，首先设置 p、q 两个指针变量分别指向 A、B 两个链表的第一个数据元素结点。然后对 p、q 两个结点的指数域进行比较，指数相同则系数相加，连入 C 链表；指数不同，则将指数较大的结点连入 C 链表。

1．具体算法如下

（1）设 p、q 分别指向 A、B 中某一结点，初值为相应链表的第一数据结点。

（2）C 为 A、B 和的最终链表表头指针，初值为 A（充分利用现有结点，节省存储资源）。

（3）比较 p、q 结点的指数域的值，进行相应操作，直到其中一个为空。具体操作如下。

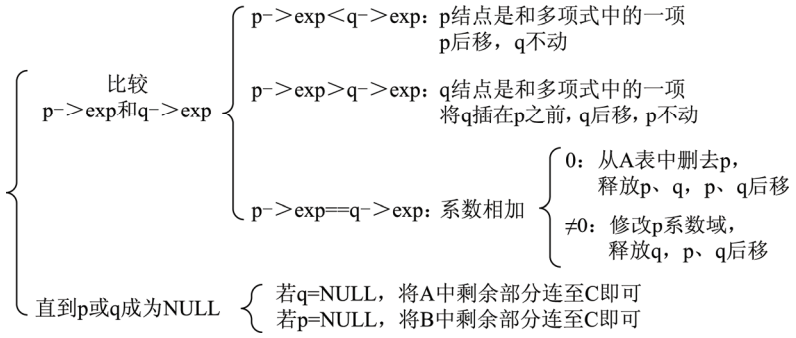


图 2.35 为多项式求和部分操作结果示意图，操作流程如下。

（1）r 指针后移一个结点指向 C 链表的第一个数据结点。

（2）r 结点的系数变为 2+6=8。

（3）p、q 指针各后移一个结点。

（4）比较 p、q 结点的指数域的值，将指数较大的 q 结点连入 C 链表。

（5）p 不动，q、r 各向前移动一步，此时链表状态如图 2.36 所示。

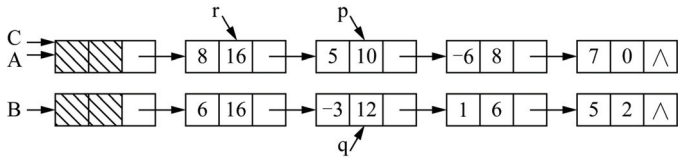


图 2.35　多项式求和第一步

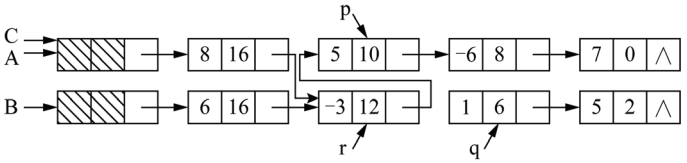


图 2.36　多项式求和第二步

算法伪代码如下：

```
Lpoly *add_poly(Lpoly *A, Lpoly *B) {
   p=A->next;
   q=B->next;
   r=A;
   C=A;
   while (p!=NULL) && (q!=NULL){ // 或者 while (!p && !q){
      if(p->exp==q->exp){
         x=p->coef+q->coef;
         if(x!=0) {
            p->coef=x;
            r=p;
         }else
            r->next=p->next;
         p=p->next;
         q=q->next;
      }else if(p->exp>q->exp) {
```

```
            r->next=p;
            r=p;
            p=p->next;
            } else {
                r->next=q;
                r=q;
                q=q->next;
            }
    }                                   // while 循环结环
    if(p==NULL)
            r->next=q;
    else
            r->next=p;
    return C;
}                                       // add_poly 函数结束
```

2.9 本 章 小 结

本章为理解其他章节内容的基础，也是历年考查重点。学习过程应注意：

（1）重点理解线性结构的本质。

（2）顺序存储结构和链式存储结构的特征。

（3）熟练掌握特殊线性表——栈和队列的特殊性。

（4）熟练掌握串的性质、存储结构特点、常用算法实现及应用。

（5）掌握数组元素的存储方式，会计算数组元素的存储单元地址。

（6）掌握特殊矩阵的特征和存储方式。

（7）熟练掌握常见算法的原理和各种存储结构下的伪码实现。