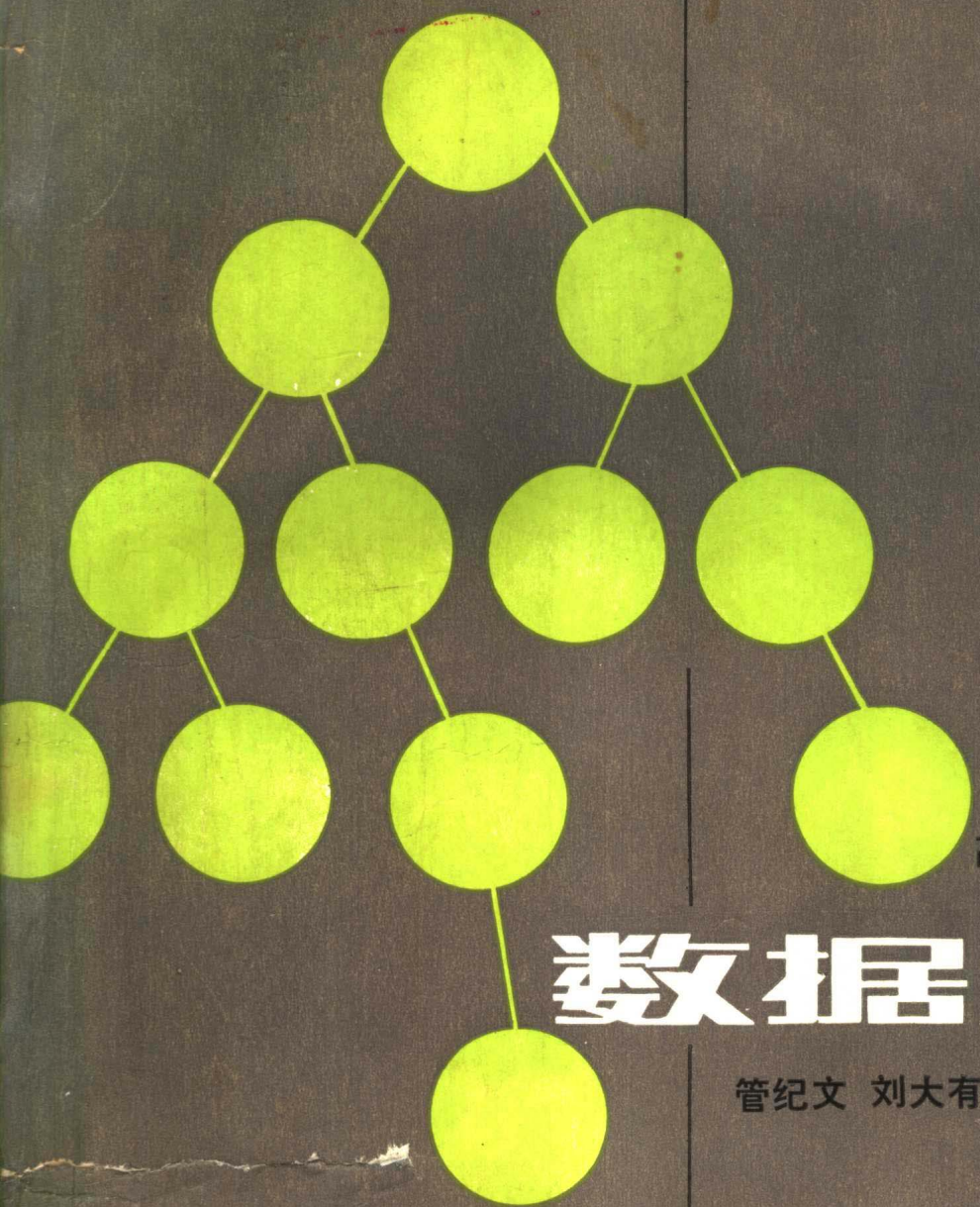
SHUJU JIEGOU



高等学校试用教材

数据结构

管纪文 刘大有

高等教育出版社

高等学校试用教材

数 据 结 构

管纪文 刘大有

高等教育出版社

内 容 提 要

本书是根据教育部颁布的高等院校计算机软件专业数据结构课程教学大纲编写的教材.

本书系统地介绍了数据结构的概念和内容. 主要包括: 线性表, 数组的存储和字符串的运算, 树形结构, 图, 排序与查找操作, 文件的结构和组织方式. 本书概念清楚, 内容丰富. 对于每一种数据结构技术以及所采取的运算和操作, 都给出了一种或几种不同的算法, 且对大多数算法给出了相应的 PASCAL 程序. 为了培养学生的运算能力, 书中还给出了许多相应的例题和习题.

本书可作为高等院校计算机软件专业的教材或参考书, 也可供广大从事研究和使用计算机的科技人员学习参考

本书是由北京大学王攻本、沈琼华同志和南京大学徐洁馨同志审阅的.

高等学校试用教材

数 据 结 构

管纪文　刘人有

*

高等教育出版社出版

新华书店北京发行所发行

河北省香河县印刷厂印装

*

开本 787×1092　1/16　印张 18.5　字数 400,000

1985 年10月第 1 版　1985 年 10 月第 1 次印刷

印数　00,001—18,700

书号　13010·01130　定价 3.10 元

目　录

第一章 绪 论

§1 引 言

计算机程序加工处理的对象，或称数据，通常是一些信息表．在大多数情况下，这些表并非是无组织的信息元素集团；恰恰相反，它们包含的数据元素之间有着重要的结构关系．

就简单的形式而言，这种表可以是一个线性表．在更复杂的情况下，这种表可以是一个二维的直至多维的数组，或是具有等级和分枝关系的树结构，以至象人脑那样极其错综复杂的多链接结构．

为了有效地使用计算机，设计出高效、准确的程序，首先就需要对数据的性质和数据元素间的关系进行深入地研究．换言之，必须对数据内部的结构关系，以及在计算机内如何表示和操作这种结构的技术，有透彻的了解．

本书主要阐述数据结构及其运算(即操作)的原理和技术．全书共分八章．第一章主要讲述数据结构和算法两个基本概念．第二章至第五章讲述了各种基本数据结构的特点、存储方法和基本运算．在这些研究的过程中，我们还将给出若干重要例子，来说明这些技术的各种各样的应用．第六章讲述排序操作．排序又叫整检、整序、分类．就是把数据元素(如记录)按关键词整序，比如排成递增次序．第七章讲述查找操作．对于给定的关键词，找出含有该关键词的那个记录．这章还阐述插入和删除运算．第八章主要讲述数据在外存上的组织方法及文件的基本结构．

为了便于上机实习，对书中的大多数算法都给出了 PASCAL 语言程序．

§2 数据结构概念

本书是阐述与研究数据结构的，因此必须回答“什么是数据结构？”这一问题．为了弄清楚数据结构这一概念，首先要明确数据这一术语的含义．直观地说，数据系指一些事实，或指一批数，或者指一个符号集合，等等．我们把组成数据的“事实”，“数值”，或“符号”等称之为数据元素．数据元素是组成数据的基本单位．数据是计算机程序使用和加工的“原料”．例如，一个简单的数值计算程序所使用的数据是一些实数或整数，一个编译程序使用和加工的数据是源程序．又如，一个能修改自身的计算机程序把自身也作为其加工的对象(数据)．

数据结构概念与数据概念不同，若想描述一个数据结构，不仅要描述数据，而且还要描述诸数据元素(它们组成了数据)之间的相互关系．这些相互关系可用一组能施于诸数据元素的运算及关于这些运算的规则来描述．让我们来看一个简单的数据结构，整数数据结构 I^*．为描述 I^*，首先必须给出组成 I^* 的全部元素的集合 $I(I=\{0, \pm1, \pm2, \cdots\})$ 和 I 中元素间的关系集合 RE；其次要规定 I^* 的运算集合 P，比方说 P 是算术四则运算 +，−，*，/(注意符号“/”表示的除运

算，要求商数和余数都是整数）之集合；最后，还要定义 P 中诸运算的运算规则（如 *，/优先于＋，－进行，等等）集合 RU. 因此，我们说整数数据结构 I* 是由 I，RE，P 和 RU 组成的. 应强调的一是点，迄今为止数据结构概念还没有一个统一的定义. 本书实际上把数据结构概念与数据的逻辑结构概念等同起来，而把数据的存储结构仅仅看成是数据的逻辑结构在计算机存储器中的存储分配方式. 这着重说明本书更侧重于数据的逻辑结构.

§3 算　法

在计算机内，数据表现为一个个记录，也叫元素或节点，在图中又称为顶点等. 每个节点由计算机内存中的一个或多个相继的字组成，这些字又分成一些字段. 最简单的情况是：一个节点仅由一个字组成，并且该节点又只含一个字段. 举例来说，我们可以用两个相继的计算机字来表示一张扑克牌，两个相继的字分成五个字段（书中也常常把字段称之为域）：

+	TAG	SUIT	RANK	NEXT
+	TITLE			

(1)

其中 $TAG=\begin{cases}1, & \text{当这张牌背向上时}\\ 0, & \text{当这张牌面向上时}\end{cases}$　$SUIT=\begin{cases}1, & \text{牌花色＝梅花}\\ 2, & \text{牌花色＝方块}\\ 3, & \text{牌花色＝红心}\\ 4, & \text{牌花色＝黑桃}\end{cases}$

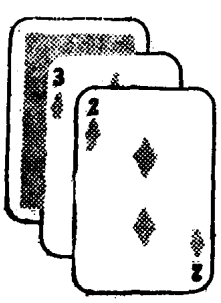
RANK＝1，2，…，13　表示这张扑克牌的面值；

NEXT　是在这叠纸牌中，紧接本张牌的下张牌（所对应的节点）的首地址；

TITLE　是本张牌的助记的五字符名称.

一个节点的地址，就叫做到该节点的链接、指针或对该节点的访问等，就是指该节点的首地址. 在上例中，NEXT 就是到下张牌的链接. 按上例中每张牌的表示格式，我们可以这样来表示一叠纸牌：

一叠实际的纸牌



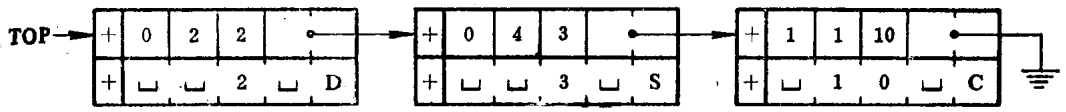
计算机表示

100	+	1	1	10	A
101	+	␣	1	0	␣ C
386	+	0	4	3	100
387	+	␣	␣	3	␣ S
242	+	0	2	2	386
243	+	␣	␣	2	␣ D

最底下的那张牌由首地址是单元 100 的节点表示；中间那张牌由首地址是单元 386 的节点表示，

它的 NEXT＝100 指出了紧接它底下的牌是“最底下的”那张牌；顶上那张由首地址是单元 242 的节点表示，它的 NEXT＝386 指出了紧接它底下的牌是中间的那张．最底下那张牌的节点的字段 NEXT＝Λ，表示它的底下已没有牌．

链接是表示复杂结构的基本手段．通常我们也可以用箭头来表达链接．上例中的链接关系即可简便地表示成



(2)

这里，实际的地址 242，386 和 100 都已不再出现，因为就数据的结构（即数据间的相互关系）而言，它们归根到底是非本质的（本质在于其间如何联系，而不在于实际的地址为何）．空链接 Λ 这里已画成“接地”．在上列表达形式中，我们还加进了链接变量 TOP，也称为指针变量，即是计算机程序之内的一个变量，其值是一个链接或指针．现在，TOP 就指向顶上的那张牌．

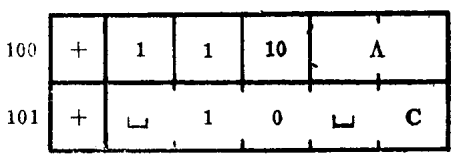
要访问节点内的诸字段，办法很简单，就是先给出该字段的名称，然后在圆括号中写出指向该节点的链接．例如，按格式(1)和表示形式(2)我们有

$$\mathrm{TITLE(TOP)}=\text{“␣␣2␣D”}$$

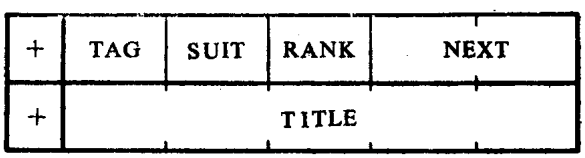
$$\mathrm{SUIT(TOP)}=2$$

$$\mathrm{RANK(NEXT(TOP))}=3$$

再如，由



和格式



我们有 RANK(100)＝10，TITLE(100)＝“␣␣10␣C”．

有了访问节点和其中的字段的办法，就能够设计出各种各样的算法来．现在举两个例子．

算法 A（在一叠扑克牌的顶上再放上一张面向上的新牌） 假定 NEWCARD 是一个链接变量，其值是一个指向新牌的指针．

A 1. [把新牌放在原叠顶上] 置 NEXT(NEWCARD)←TOP（新牌的下张牌是原叠最顶上的那张牌）．

A 2. [新牌成新叠之顶] 置 TOP←NEWCARD（让 TOP 保持指向叠顶）．

A 3. [新牌面向上] 置 TAG(TOP)←0（记下新牌是面向上的）．

算法 B(计叠中的牌数)

B1. [初始化] 置 N←0, X←TOP(N 是整数值变量，计叠中牌数；X 是链接变量，由顶而底查这叠牌).

B2. [已到底?] 若 X=Λ, 则算法告终, N 就是叠中的牌数.

B3. [查下张牌] 置 N←N+1, X←NEXT(X), 并返回步骤 B2(若未到底，再查下张牌，边查边数，一直查到底).

这里，我们来讲一下算法. 算法就是一项解决问题的办法. 在本书中，算法就是解决数据结构问题的办法. 本书中的结果，都集中地以算法形式给出. 其实，设计计算机程序，就是要在计算机上实现某种算法. 算法是用自然语言编写的程序(实际上，算法通常是用自然语言、数学语言和约定的符号语言来书写的)，程序就是用计算机所能接受的语言编写的算法. 因此，对于计算机程序设计而言，最基本的是算法.

现在，我们来解释算法的写法，也就是来约定算法的书写格式(即本书中绝大多数算法的书写格式).

1. 算法都有一个标识字母，如计叠中牌数的算法 B. 今后引用这个算法时，就写做算法 B 或算法 1.3 B，表示是第一章§3 的算法 B. 而算法中的每一步骤则记成 B1, B2, …, 等等.

2. 算法每一步骤开头都有带方括号的短句，用于综述该步骤的主要内容. 这样就可以直接得出一个框图，来帮助读者直观地了解这个算法. 例如算法 B 的框图是

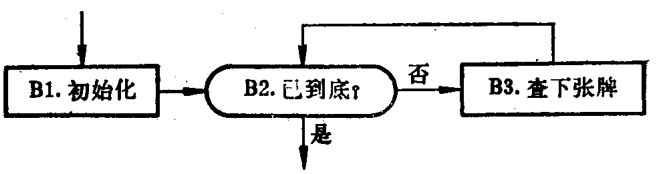


图 1.1 算法 B 的框图

框图中，方框表示执行框，左右两边为圆弧的框表示判断框.

3. 算法的每一步骤，在带方括号的短句之后，接着说明该步骤所要执行的动作或所要进行的判断. 有时为了帮助读者理解某一动作(操作)，还在该动作之后加了注释，注释的内容用圆括号括了起来，以便与算法部分相区别，注释完全不影响该步骤的执行.

4. 步骤 B1 和 B3 中的"←"是最重要的替代或赋值运算. "m←n"表示变量 m 的值换成了变量 n 的当前值，也就是把变量 n 的当前值赋给变量 m. 例如，n 增加 1 的运算，就可以表示成"n←n+1"(注意，不要写成"n→n+1"!).

5. 今后，我们约定把"n←r, m←r"写成"n←m←r". 而交换两个变量的值则写成"交换 m↔n"，这等价于"置 t←m, m←n, n←t".

6. 除非另有改变执行次序的说明，算法通常总是由编号最小的步骤，即步骤 1 处开始顺序执行. 如算法 B，执行时由步骤 B1 开始，到 B3 后即返回执行 B2. 在 B2 中，条件"若 X=Λ"是

执行该步骤动作的前提条件. 即,若 X≠Λ 则执行下一步骤,即 B3.

7. ▌表示本算法已写完.

8. 在算法中,我们还约定,把下标变量 v_j 和 a_{ij} 分别记成 v[j]和 a[i, j].

9. 在描述一个算法时,我们常常对两类全然不同的东西,都采用符号名称. 一类是变量的名称,如 TOP, NEWCARD, N, X 等等;另一类是字段的名称,如 TAG, NEXT 等等. 对这两类全然不同的东西,千万注意不要使其混淆.

在地址和内容之间,我们采用如下的记号:

a) CONTENTS(1000)表示单元 1000 中所存的内容. 一般地,若 L 为一链接变量,则 CONTENTS(L)就表示 L 所指的单元中的内容. CONTENTS(L) 有时也简记为(L). CONTENTS 总表示一个全字(全字长的字段)的内容.

b) 如果 V 是保存在某单元中的值,则 LOC(V)就表示该单元的地址. LOC(V)有时也简记为[V]. 于是,若 V 是一变量,其值占有内存的一个全字,则我们有

$$\text{CONTENTS(LOC(V))}=\text{V} \quad \text{或} \quad ([\text{V}])=\text{V}$$

另一方面

$$\text{LOC(CONTENTS(L))}=\text{L} \quad \text{或} \quad [(\text{L})]=\text{L}$$

这里 L 是一个链接变量,L≠Λ,而且内存中存有值 CONTENTS(L)的单元是唯一的,即只有单元 L 中存有 CONTENTS(L).

前面说过,一个节点可能占有多个计算机字,象 CONTENTS(L)则只表示一个字的内容. 有时,我们需要用到表示一个节点的变量,例如,我们可以写

$$\text{CARD}\leftarrow\text{NODE(TOP)} \tag{3}$$

在(3)中 CARD 就是一个表示整个节点(它往往由多个字段组成)的变量,而 NODE 是一个象 CONTENTS 一样的字段说明. 不同的是它表示整个节点所包括的全部字段,NODE(TOP)和 CARD 都表示一整个节点. 根据(1)和(2),它们各占有两个字. 如果一个节点中有 c 个字,则记号(3)就是 c 个赋值,即

CONTENTS(LOC(CARD)+j)←CONTENTS(TOP+j), $0\leqslant j<c$ 的缩写.

习　题

1. 在(2)中, SUIT(NEXT(TOP))=? NEXT(NEXT(NEXT(TOP)))=?

2. 给出一个类似于算法 A 的算法,它在一叠牌(可能为空)的底下放进一张背向上的新牌.

3. 写出一个 PASCAL 程序,它从顶上那张牌开始,打印出这叠牌当前内容的字母名称,每行打印一张牌,并把背向上的牌打上圆括号.

第二章 线 性 表

§1 线性表的定义、运算；堆栈、队列

现实数据中的结构信息是各种各样的. 但在计算机上，只能抽取我们所需要的某些方面. 这里，首先需要判定我们究竟要表示多少结构，并且怎样才能使每项信息都得以存取. 为此就需要知道，对于数据将施以什么操作.

一个线性表就是 n≥0 个节点 X[1], X[2], …, X[n]的集合，其结构仅涉及诸节点的线性相对位置：当 n>0 时，X[1]是头一个节点；当 1<k<n 时，第 k 个节点 X[k]在 X[k−1]之后，在 X[k+1]之前；而 X[n]是最后一个节点. 对于线性表，我们所要施行的操作，举例来说有：

1. 存取. 存取第 k 个节点，来检查和(或)更新其中字段的内容.
2. 插入. 就是在第 k 个节点前插入一个新节点.
3. 删除. 删除第 k 个节点.
4. 归并. 把多张线性表组合成一张统一的新表.
5. 分拆. 把一张表分拆成多张表.
6. 复制.
7. 计数. 确定表中的节点个数.
8. 排序. 把表中节点按某一字段整序.
9. 查找. 寻觅具有特定字段值的节点.

后两项操作，排序和查找，将在第六、七两章中讨论，本章只考虑前七项操作. 根据对这七项操作不同的侧重，线性表又分为各种不同的类型.

最常遇到的线性表，它们对其中前三项操作：插入、删除和存取，总是在线性表的首尾两端进行. 这样的结构主要有两类：堆栈就是一张这样的线性表，对于它，所有的插入和删除(以至几乎所有的存取)都是在表的一端进行的；队列也是一种这样的线性表，对它进行的所有插入都在表的一端进行，所有的删除(以至几乎所有的存取)都在表的另一端进行.

对于堆栈，我们总是取走当前在表中“最年轻的”节点，即最近新插入的节点，所以堆栈也称为 LIFO(后进先出)结构或下推表；反之，队列总是被取走“最老的”节点，故队列也叫做 FIFO(先进先出)结构. 这两类结构都有特殊的术语：新节点被插入一堆栈的顶上，称为压入堆栈；从栈顶取走节点，称为弹出堆栈；堆栈最底下的节点是最后可取出的节点，就是说仅当其上的所有节点全都被删除时，才能取走该节点. 对于队列，我们关注其前头(队头)和后尾(队尾). 新节点由队尾进入，并在其最终到达前头位置时才可被取走. 我们用

$$A \Leftarrow x$$

表示值 x 插入 A(压入 A 的顶上，当 A 为堆栈时；插入 A 的后尾，当 A 为队列时)；用

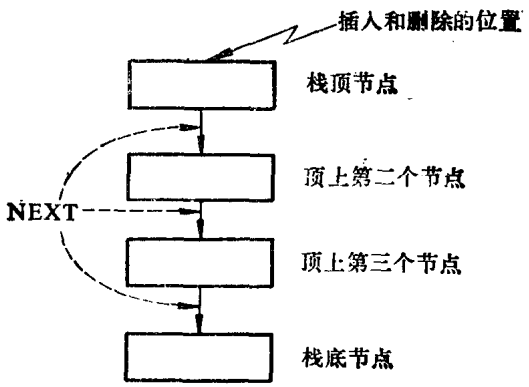
$$x \Leftarrow A$$

表示把变量 x 置成堆栈 A 之顶上(弹出)或队列 A 之前头的元素(或曰节点)所包含的信息，并从 A 中删除这个元素(但若 A 为空时，则此记号无意义).

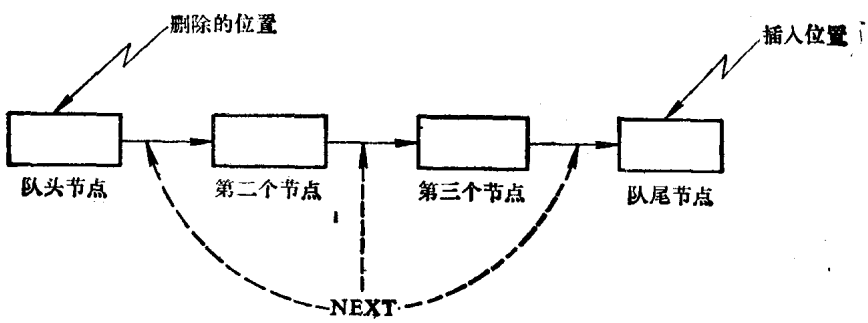
当 A 是一个非空的堆栈时，我们可以用

$$top(A)$$

表示栈顶上的元素(即节点). 堆栈和队列的插入和删除操作可由图 2.1 直观地示出.



(a) 堆栈



(b) 队列

图 2.1 两类最常用的线性表

§2 线性表的存储结构

要使线性表成为计算机程序使用和加工的对象，那就必须解决线性表在计算机中的表示问题. 本节将给出四种方案，它们是顺序分配、链接分配、循环链接结构和双重链接结构等.

2.1 线性表的顺序分配

顺序分配，即是把表中的节点一个接一个地放进顺序的内存单元，这是最简单而又最自然的表达线性表的方式.

假定每个节点都占据 c 个连续的内存单元，并设线性表之长度为M. 于是，我们有

$$LOC(X[j+1])=LOC(X[j])+c$$

从而可以推出

$LOC(X[j])=L_0+c*j$，对于 $1\leqslant j\leqslant M$ 其中 L_0 是基地址，即节点 X[0]的首地址. X[0]是人为设置的伪节点，也具有长度 c，但它不是线性表的成员. 下面图 2.2 示出一个例子：设有一个 M＝4 的线性表，其每个节点占两个连续的内存单元，基地址 $L_0=300$.

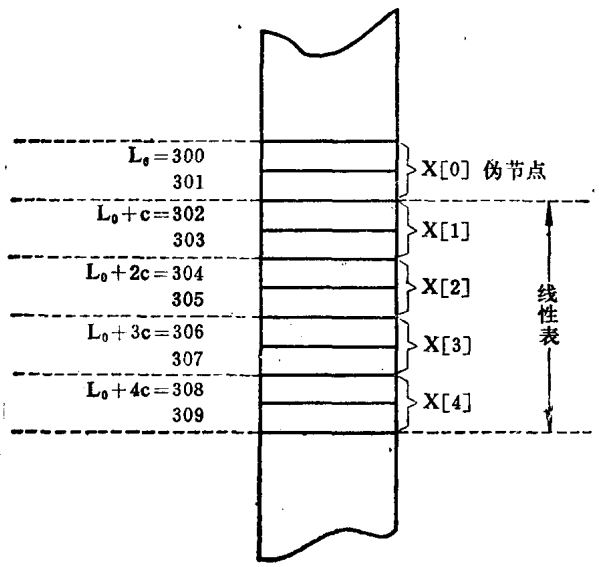


图 2.2

下面我们对这种最简单的结构作些讨论，看看它的优点和局限性.

对于处理堆栈而言，顺序分配十分方便. 实际上，只需要一个堆栈指针变量 T. 设堆栈包含 M个节点，当堆栈为空时，命 T＝0.

压入堆栈之操作 X⇐y 为

若 T＝M，则 overflow；否则置 T←T＋1，X[T]←y.

弹出堆栈的操作 y⇐X 为

若 T＝0，则 underflow；否则置 y←X[T]，T←T－1.

对于队列，我们总要关注其队头与队尾. 因此，需要使用两个指针变量 F 和 R，F 用来指向队头的前一个(节点)，R 指向队尾. 设队列共包含 M 个节点.

算法 QI(把节点变量 y 之值插入队列的尾部) 队列 X 之长不超过 M，初始置 F←R←0，一个空队已建立.

QI1. [上溢吗？] 若 R＝M，则输出上溢信息并终止本算法；否则，置 R←R＋1.

QI2. [插入节点] 置 X[R]←y.

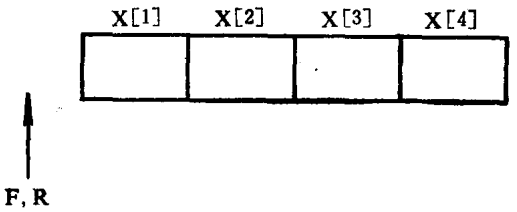
算法 QD(删除队头节点，并将其包含的信息赋给 y) 一个长度不超过 M 的队列 X 已经建立.

QD1. [下溢吗？] 若 F＝R＝0，则输出下溢信息并终止本算法.

QD2. [删除节点] 置F←F+1(F指向队头的前一个), y←X[F].

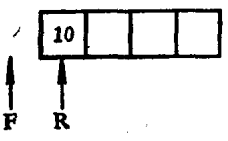
QD3. [队列为空？] 若F=R, 则F←R←0, 并终止本算法.

上述两个算法虽然简单，但却不能充分利用空间. 让我们用一个实例来说明这一结论. 假定有一包含四个节点之队列X, 初始X为空, 即F=R=0.

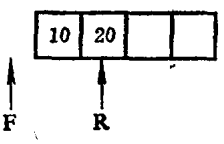


空队列

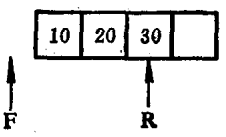
假定每一个节点都只包含一个字段(域), a, b, c, d和e都是代表这样节点的变量, 并且它们予先分别被赋了值10, 20, 30, 40和50. 先依序插入节点a, b, c和d, 图示如下:



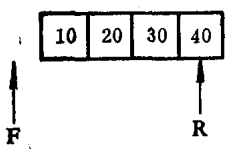
插入节点a



插入节点b

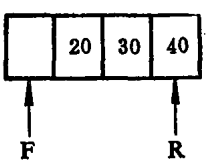


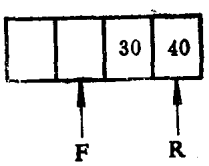
插入节点c

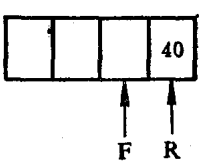


插入节点d

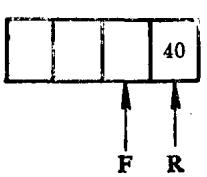
接着删除节点a, b和c:







最后插入节点e:



因此时R=M=4, 故若插入节点e将发生上溢!

我们从上例中看到, 当队列X尚有两个空位置(节点)时, 却不能插入一个新节点e. 这一情况, 一方面说明了上述结构是有缺陷的, 另一方面也提醒人们注意到上面的溢出是虚假的. 一种改进的办法是循环队列结构. 所谓循环队列就是一个以循环方式来安排M个节点X[1], X[2], …, X[M]的队列.

算法CQI(把节点y插入循环队列之尾部) 队列X由M个节点按循环方式组成, 开始时

(即该算法一次也未执行过的时候)指针 F 和 R 均被置成 1(F 仍指向队头的前一个节点，R 指向队尾节点)．本算法将把节点 y 插入到队列尾部，如果不发生溢出的话．

CQI1. [R=M？] 若 R=M，则置 R←1；否则，置 R←R+1．

CQI2. [上溢吗？] 若 R=F，则输出上溢信息，并终止本算法．

CQI3. [插入队尾] 置 X[R]←y．

算法CQD(删除循环队列头节点) 队列 X 由 M 个节点按循环方式组成，设 X 已经建立，指针变量 F 指向队头的前一个节点；R 指向队尾．如果不发生下溢，则本算法将删除队头节点，并将该节点赋给节点变量 y．

CQD1. [下溢吗？] 若 R=F，则输出下溢信息并终止本算法．

CQD2. [F=M？] 若 F=M，则置 F←1；否则 F←F+1．

CQD3. [删除队头节点] 置 y←X[F]．

下面是一个循环队列的例子．假定这个循环队列 X 由六个节点组成，在尚未执行插入或删除算法之前，空队列 X 已建立，有 F=R=1，如图 2.3 所示．

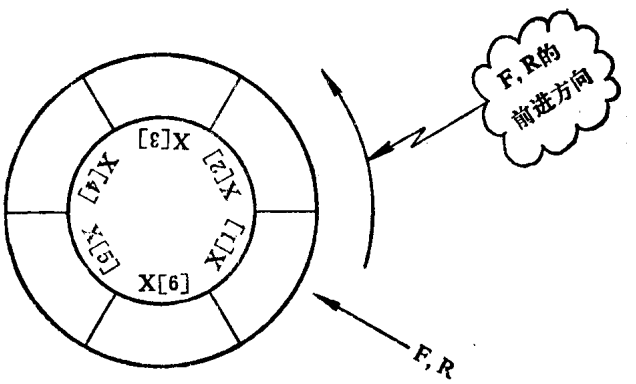
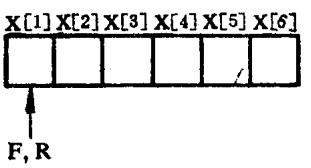


图 2.3

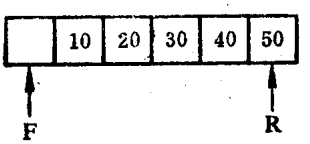
为方便计，我们在具体图示中，将圆环展成长条矩形．设 X 的每个节点都只包含一个字段(域)．a，b，c，d，e，f 和 g 都是代表 X 的那样节点的变量，并且它们予先都分别被赋予了值 10，20，30，40，50，60 和 70．

(i)



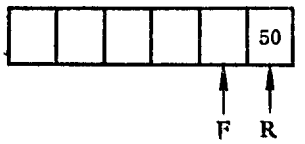
初始 X 为空队

(ii)



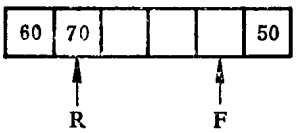
依序插入 a，b，c，d，e 之后

(iii)



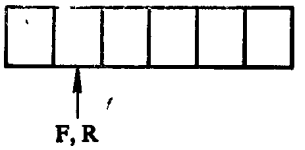
删除 a, b, c, d, 之后

(iv)



插入 f, g 之后

(v)



删除 e, f 和 g 之后

读者或许已注意到，无论在队列还是在循环队列中，总是约定 F 指向队头的前一个节点. 这样做的结果，花费了一个节点的空间，但加快了算法的执行速度. 当 M 稍大时，比如 M＝10 时，空间利用率达到 90%；当 M＝100 时，空间利用率达到 99%. 所以花费一个节点还是十分值得的. 循环队列由满变空时，有 F＝R，但不一定有 F＝R＝1.

2.2 线性表的链接分配及循环链接结构

一、链接分配

在顺序的内存单元中保存线性表，可以采用更灵活的分配方案来代替顺序分配，例如链接分配. 这种分配要求每个节点都包含一个链接字段(域)，该字段之内容是指向表中下一个节点的指针. 为简便计，不妨假定每个节点都有两个字段：INFO 和 LINK. 一个节点的格式如下：

INFO	LINK

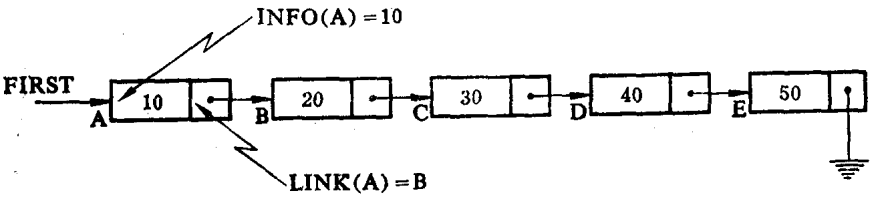
字段 INFO 中存放任意信息，譬如说是一些整数，字段 LINK 中存放指向表中下一个节点的指针. 顺序分配和链接分配的对比由图 2.4 给出.

顺序分配			链接分配			
地址	内容		地址	内	容	
L_0+c	10	X[1]	A	10	B	X[1]
L_0+2c	20	X[2]	B	20	C	X[2]
L_0+3c	30	X[3]	C	30	D	X[3]
L_0+4c	40	X[4]	D	40	E	X[4]
L_0+5c	50	X[5]	E	50	Λ	X[5]

图 2.4 顺序和链接分配对比图

其中 A, B, C, D 和 E 分别表示节点(链接分配)X[1], X[2], …, X[5]之首地址，Λ 表示空链

接. 图 2.4 的链接分配还可表成



FIRST 是一个指向链表中第一个节点的链接变量.

对比起来,顺序分配与链接结构的优缺点如下:

(1) 链接分配需要额外的空间供 LINK 字段使用,但链表的诸节点可灵活地散布在内存中(可用空间)各处.

(2) 链接分配便于做删除操作. 例如,如果我们要从上面的链接表中删去节点 3,则只需执行

$$LINK(B)\leftarrow D$$

但对于顺序分配,节点 3 后面的全部节点都必须向前移动,即需要执行

$$INFO(L_0+3C)\leftarrow INFO(L_0+4C)$$
$$INFO(L_0+4C)\leftarrow INFO(L_0+5C)$$

若例中的表 X 包含 100 个节点,则删去节点 3 就需要执行 97 次赋值操作,即节点 3 之后的全部 97 个节点均要前移一个位置. 然而,对链接分配却不然,无论表 X 多么长(即无论包含的节点怎样多),删去一个节点只需一次赋值操作,完全不象顺序分配那样,其操作次数与表 X 之长 M 成正比.

(3) 链接结构便于插入操作.

例如,若把一个字段 INFO 之值为 100首地址为 W 的新节点,插入上面的链表之节点 3 的前面,则只需执行

$$LINK(W)\leftarrow LINK(B)(即\ C),LINK(B)\leftarrow W$$

图 2.5 对链表的插入操作给出了直观说明.

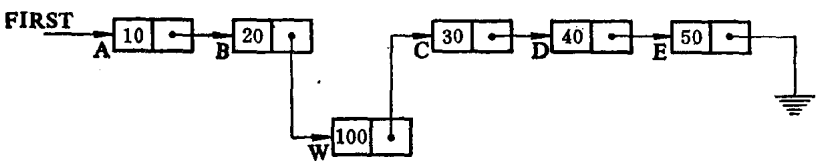


图 2.5 链表的插入操作

而对于顺序分配,则需要把节点 2 之后的节点全部都后移一个位置(一个节点). 与(2)中的一样,完成顺序分配之插入操作所要执行的赋值操作次数,与表 X 之长 M 成正比. 而链接分配之插入操作与表 X 之长 M 无关,即无论链表 X 是怎样长,完成插入一个新节点之操作只需执行两次赋值操作.

(4) 链接结构不便于随机存取. 设表 X 之长为 M(即包含 M 个节点),若读取第 k 个节点