《数据结构》

第二章　　线性表

目 录

2.0 案例导引

2.1 线性表的逻辑结构

2.2 线性表的顺序存储结构

2.3 线性表的链式存储结构

2.4 顺序表与链表的比较

2.5 案例实现——通讯录管理

知识目标：

- 理解线性表的逻辑结构特征
- 掌握顺序表的含义、特点、基本运算和相关算法分析
- 掌握链表的含义、特点、基本运算和相关算法分析
- 理解循环链表、双链表的逻辑结构特征及基本运算
- 理解顺序表和链表的比较

技能目标：

- 能应用顺序表的理论设计算法，解决实际问题
- 能应用链表的理论设计算法，解决实际问题

2.0 案例导引

案例：通讯录管理

编写一个用于通讯录管理的程序，实现对联系人信息的插入、删除、查找等功能。要求记录每位联系人的姓名、性别、手机号码、住宅号码和邮箱信息。

案例探析：

对于通讯录中的联系人需要管理其姓名、性别、电话、邮箱等信息，每位联系人所需要存储的信息类型是相同的，也就是说各个结点应该具有相同的结构。同时，各位联系人的信息记录之间按顺序排列，形成了线性结构。线性结构是最简单最常用的数据结构。

2.1 线性表的逻辑结构

2.1.1 线性表的定义

线性表（Linear List）：是由n个性质相同的数据元素组成的有限序列。表中数据元素的个数n定义为线性表的长度。n=0的表称为空表，即该线性表不包含任何数据元素。n>0时，线性表记为：

$$L=(a_1,a_2,a_3,\ldots,a_n)$$

其中a_i $(1\le i\le n)$称为表中的第i个数据元素，下标i表示该元素在线性表中的位置。任意一对相邻的数据元素a_{i-1}，a_i $(2\le i\le n)$存在着序偶关系（a_{i-1}，a_i)，且a_{i-1}称为a_i的直接前驱，a_i称为a_{i-1}的直接后继。

线性表的逻辑结构为：

●存在唯一的数据元素a_1，或称首结点，它没有直接前驱，只有一个直接后继；
●存在唯一的数据元素a_n，或称尾结点，它没有直接后继，只有一个直接前驱；
●除第一个结点（首结点）之外，集合中的每个数据元素均只有一个直接前驱；
●除最后一个结点（尾结点）之外，集合中每个数据元素均只有一个直接后继。

线性表中的数据元素之间是一对一的关系，数据元素可以是简单数据类型，如整型、字符型等，也可以是用户自定义的任何类型如记录类型等。

线性表是一种典型的线性结构，用二元组表示为：

$$S=(A,R)$$
$$A=\{a_1,a_2,\ \ldots,\ a_i,\ldots,a_n\ \}$$
$$R=\{< a_1,a_2>,< a_2,a_3>,\ldots,<a_{i-1},a_i>,< a_i,\ a_{i+1}>,\ldots,\ <a_{n-1},a_n>\}$$

线性表的逻辑结构如图2-1所示：

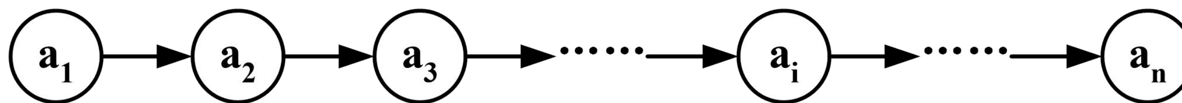


图2-1　线性表的逻辑结构图

在日常生活中有许多线性表的例子，比如一副扑克牌的点数可以表示为（2,3,4,5,6,7,8,9,10,J,Q,K,A）；再如图书目录、银行排队叫号等都是线性表的例子。

学生信息表就是一例典型的线性结构，如表2-1。

表2-1　　学生信息表

学号	**姓名**	**性别**	**电话号码**	**邮箱**
10001	李平	男	86000001	liping@126.com
10002	王芳	女	86000002	wangf@163.com
10003	吴冰	女	86000003	wubing@sina.com
10004	李清	男	86000004	liqing@yahoo.com
……	……	……	……	……

2.1.2 线性表的抽象数据类型定义

线性表的长度可以随着数据元素的插入、删除等操作而增加或减少，它是一种灵活的数据结构。线性表的抽象数据类型定义如下：

ADT List{

数据对象：D={ai|ai为DataType类型，1≤i≤n，n≥0}

/*DataType为自定义类型*/

数据关系：R={<ai-1,ai>|ai-1, ai∈D, 2≤i≤n }。在非空表中，除了首结点，每个结点都有且只有一个前驱结点；除了尾结点，每个结点都有且只有一个后继结点。

基本操作：

InitList(&L)：构造一个空的线性表L，即表的初始化。

ListLength(L)：求线性表L中的结点个数，即求表长。

GetNode(L,i)：取线性表L中的第i个结点，要求1≤i≤ListLength(L)。

LocateNode(L,x)：在L中查找值为x的结点，并返回该结点在L中的位置。若L中有多个结点的值和x相同，则返回 首次找到的结点位置；若L中没有结点的值为x，则返回一 个特殊值表示查找失败。

InsertList(&L,x,i)：在线性表L的第i个位置上插入一个值为x的新结点，使得原编号为i，i+1，…，n的结点变为编号为i+1，i+2，…，n+1的结点。这里1≤i≤n+1，n是原表L 的长度。插入操作成功后表L的长度加1。

DeleteList(&L,i)：删除线性表L的第i个结点，使得原编号为i+1，i+2，…，n的结点变成编号为i，i+1，…，n-1的结点。这里1≤i≤n，n是原表L的长度。删除操作成功后表L的长度减1。

}ADT List

2.2 线性表的顺序存储结构

线性表的顺序存储结构又称顺序表（Sequential list）。

2.2.1顺序表的结构

顺序表是用一组地址连续的存储单元依次存放线性表的数据元素，即保持元素同构且无缺项。

若每个数据元素占用c个存储单元，并以所占的第一个存储单元地址作为这个数据元素的存储位置，设表的最大长度为MaxSize，如图2-2，则表中任一元素ai的存储地址为：

$$LOC(a_i)=LOC(a_1)+(i-1)*c \qquad (1 \le i \le n)$$

顺序表中为相邻的元素ai和ai+1赋予相邻的存储位置LOC(ai)和LOC(ai+1)，即在线性表中逻辑关系相邻的数据元素在内存中的物理位置也是相邻的。对于这种存储方式，只要确定表头结点的首地址，线性表中任一数据元素都可以随机存取，所以顺序表是一种随机的存储结构。

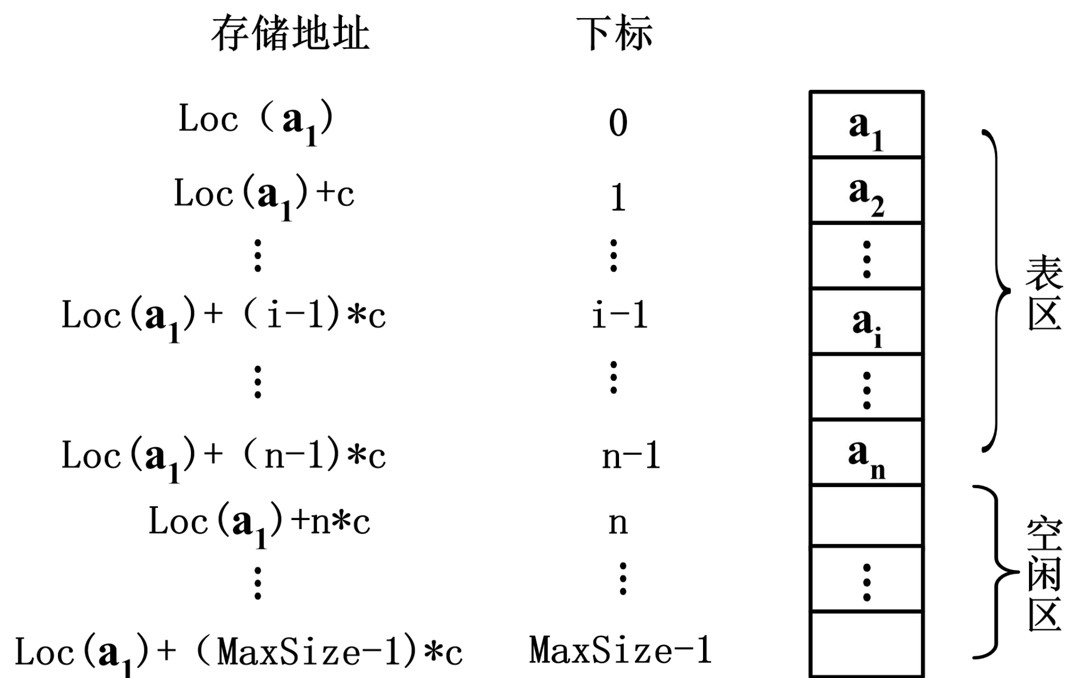


图2-2 顺序表逻辑结构图

2.2.2 顺序表上实现的基本运算

在C语言中，可以采用结构体类型来定义顺序表类型，如下：

```
/*顺序表的定义：*/
#define MaxSize 80    /*表空间大小应根据实际需要设定，这里假设为80 */
typedef int DataType;  /* DataType为数据元素的类型，可以是任何类型*/
typedef struct
{
  DataType data[MaxSize];        /* data[]用于存放表结点 */
  int length;                    /* 当前的表长度 */
}SeqList;
```

顺序表的存储结构如图2-3。

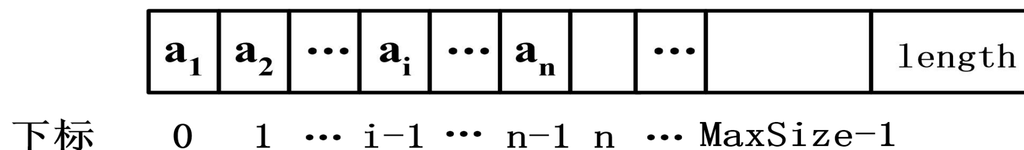


图2-3　顺序表存储结构图

1. 建表

输入给定的数组元素作为线性表的数据元素，将其传入顺序表中，将传入的元素个数作为顺序表的长度建立顺序表。

【算法2-1】：

```
void CreateList(SeqList *L,DataType a[],int n)  /* 建立顺序表 */
{
  int i;
  if(n>MaxSize)
  {
    printf("overflow");                    /* 如果n大于MaxSize，出现上溢 */
    exit(0);
  }
  for(i=0;i<n;i++)                         /* 为线性表的元素赋值 */
    L->data[i]=a[i];
  L->length=n;                             /* 设置表中元素个数 */
}
```

该算法的问题规模是表的长度n，基本语句是for循环中执行元素赋值的语句，故时间复杂度为O(n)。

在日常工作中，根据实际情况可以设计多种建表方式，比如通过键盘输入数据或通过文件读取数据等等。下面为读者提供从键盘输入数据建立顺序表的算法：

【算法2-2】：

```
void CreateListB(SeqList *L)    /*从键盘输入数据，建立顺序表*/
{
  int i,value;
  i=0;
  printf("请输入数据，输入9999时结束：\n");
  scanf("value=%d",&value);
```

```
    while(value!=9999)
    {
      if(i>MaxSize-1)
      {
         printf("overflow");          /*如果i大于MaxSize-1，出现上溢 */
         exit(0);
      }
      L->data[i]=value;                    /*为线性表的元素赋值 */
      i++;
      scanf("value=%d",&value);
    }
    L->length=i;                        /*设置表中元素个数 */
}
```

2.插入

线性表的插入运算是指在线性表的第i个(1≤i≤n+1)位置上，插入一个新元素x，使长度为n的线性表（$a_1,a_2, \ldots, a_i,\ldots,a_n$）变成长度为n+1的线性表（$a_1,a_2, \ldots, a_{i-1}, x, a_i,\ldots,a_n$）。

顺序表中要保持元素同构且无缺项。因此在表中进行插入运算时，必须将表尾结点至待插入位置的结点依次后移，空出第i个位置，如图2-4 (b)图（由内存的特性知，此时该物理单元的内容仍为a_i，记为（a_i）），然后将新元素插入该位置，完成插入运算。仅当在原表尾结点后插入新结点时，即插入位置为i=n+1时，无需移动结点。

在C语言中，数组下标从0开始依次存放数据元素，其完整插入过程如图2-4。

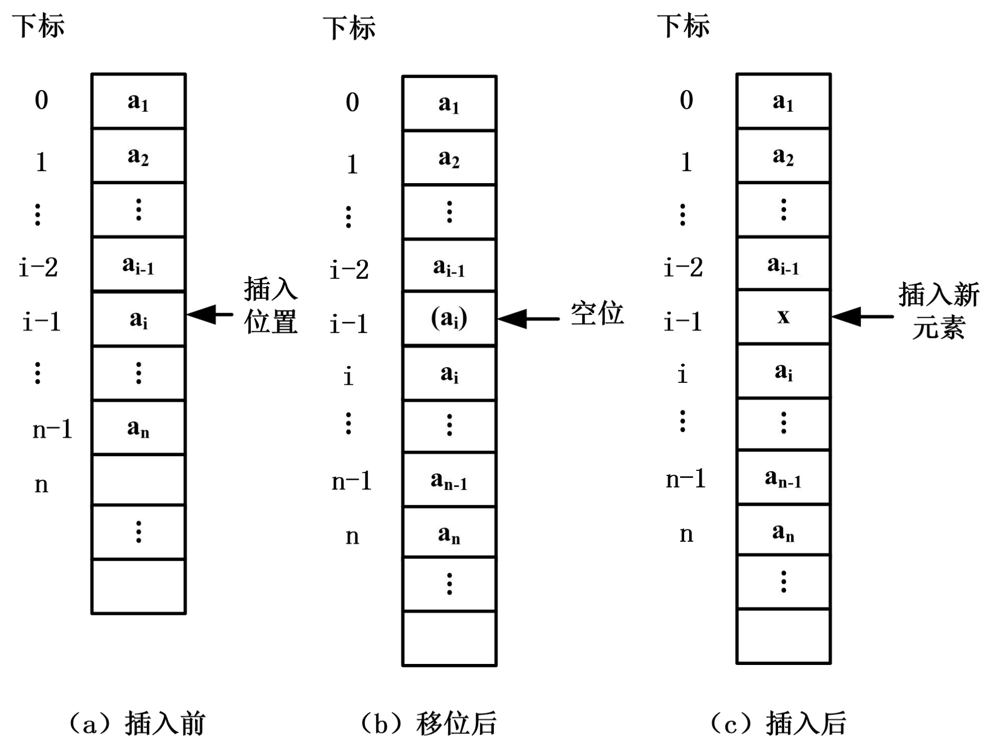


图2-4　顺序表插入运算示意图

【算法2-3】：

```
void InsertList(SeqList *L,DataType x,int i)
/*将新结点x插入L所指的顺序表的第i个结点的位置上*/
{
    int j;
    if (L->length==MaxSize)         /*表空间已满，不能插入元素，退出运行 */
    {
      printf("表空间已满，不能插入元素，退出运行。");
      exit(0);
    }
    if (i<1 || i>L->length+1)         /*插入位置错误，退出运行 */
    {
      printf("插入位置非法");
      exit(0);
     }
   for (j=L->length-1;j>=i-1;j--)       /*从表尾结点至第i个结点依次后移 */
     L->data[j+1]=L->data[j];
   L->data[i-1]=x;                     /*新元素赋值 */
   L->length++;                        /*表长加1 */
}
```

p_i

该算法的问题规模是表的长度n，基本语句是for循环中执行元素后移的语句。当i=1时，即新插入的元素为表头结点，需要移动表中所有的元素，元素后移语句将执行n次，这是最坏的情况，时间复杂度为O(n)；当i=n+1时，即新插入的结点为表尾结点，元素不需要执行后移，这是最好的情况，时间复杂度为O(1)。表长为n的线性表中，在第i个位置插入一个新元素，元素后移语句的执行次数为n-i+1。假设i是一个随机值，即在多次插入运算中取值分布是均匀的，则概率p_i为$\frac{1}{n+1}$，需要移动的元素平均次数为

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

也就是说，在顺序表上实现插入操作，等概率情况下，平均要移动表中一半的数据元素，算法的平均时间复杂度为O(n)。

3.删除

线性表的删除运算是指将线性表的第i个(1≤i≤n)位置上的元素删除，使长度为n的线性表（$a_1,a_2, \ldots, a_i,\ldots,a_n$）变成长度为n-1的线性表（$a_1,a_2, \ldots, a_{i-1}, a_{i+1},\ldots,a_n$）。

顺序表中应保持元素同构且无缺项。在表中进行删除运算时，将待删除结点之后的结点至表尾结点依次前移，顺次覆盖前一个位置的元素，实现删除第i个元素的操作。仅当删除原表尾结点时，即删除位置为i=n时，无需移动结点。其删除过程如图2-5。

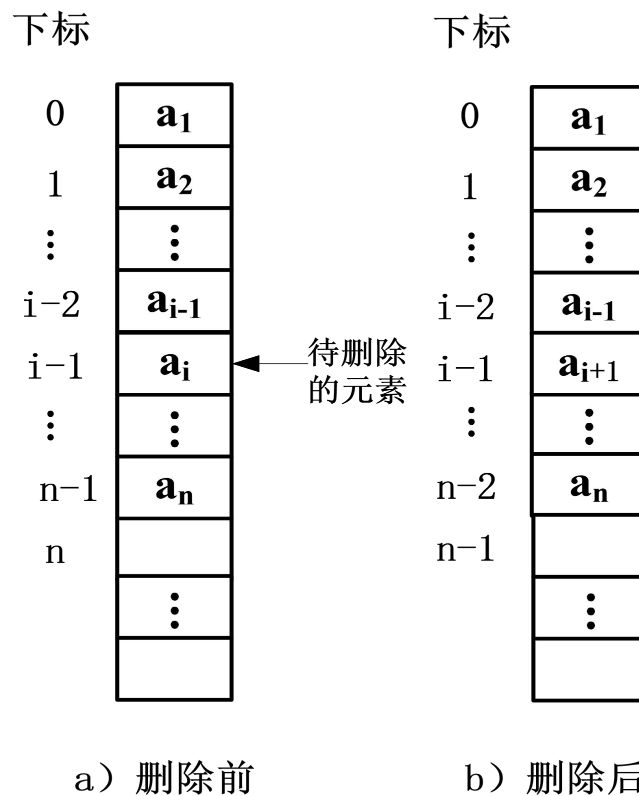


图2-5 顺序表删除运算示意图

【算法2-4】：

```
void DeleteList(SeqList *L,int i)      /*从L所指的顺序表中删除第i个结点*/
{
    int j;
    if (L->length==0)
    {
      printf("线性表为空，退出运行\n");
      exit(0);
    }
    if (i<1 || i>L->length)
    {
      printf("删除位置非法\n");
      exit(0);
     }
    for (j=i;j<=L->length-1;j++)
      L->data[j-1]=L->data[j];         /*将自第i个元素之后的所有元素向前移动*/
    L->length--;                        /*表长减1*/
}
```

　　该算法的问题规模是表长n，基本语句为for循环中元素前移的语句。当i=1时，即删除表头结点，需要移动表中除表头结点外所有的元素，这是最坏的情况，时间复杂度为O(n)；当i=n时，即删除表尾结点，元素不需要移动，这是最好的情况，时间复杂度为O(1)。在表长为n的线性表中，删除第i个元素，元素前移语句的执行次数为n-i。假设i是一个随机值，即在多次删除运算中取值分布是均匀的，则概率p_i为$\frac{1}{n}$，需要移动的元素平均次数为

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

也就是说，在顺序表上实现删除操作，等概率情况下，平均要移动表中大约一半的数据元素，算法的平均时间复杂度为O(n)。

4.查找

查找运算分按值查找和按位查找。

(1)按值查找

在顺序表中实现按值查找操作，需要对顺序表中的元素按照顺序依次进行比较，如果查找成功，则返回元素的序号（注意：不是元素的下标）；否则，返回0。

【算法2-5】：

```
int LocateList(SeqList L,DataType x)            /*顺序表按值查找*/
{
   int i=0;
   while (i<L.length && L.data[i]!=x)
      ++i;                             /*如果查找成功，下标为i的元素等于x*/
   if (i<L.length)
      return i+1;                      /*返回找到元素的序号i+1*/
   else
     return 0;                         /*找不到返回0*/
}
```

该算法的问题规模是表长n，基本语句为while循环中元素比较的语句。如果顺序表的最后一个元素为要找的x，就需要从第一个元素开始，比较n个元素，这是最坏的情况，时间复杂度为O(n)。如果顺序表的第一个元素就是x，算法只要比较一次就可以，这是最好的情况，时间复杂度为O(1)。等概率情况下，平均要比较n/2个元素，该算法的平均时间复杂度为O(n)。

本算法按顺序表从前向后进行查找，也可以修改算法实现从后向前查找。当实际项目中数据量比较大，经常使用的数据在表的后半部分时，适合采取从后向前查找的算法。

(2)按位查找

根据顺序表的随机查找的特性，按位查找只需返回相应位置的数据元素即可。

【算法2-6】：

```
DataType GetNode (SeqList L,int i)/*顺序表按位查找*/
{
  if(i<1||i>L.length)
  {
    printf("查找位置非法");
    exit(0);
  }
  else
    return L.data[i-1];                /*返回找到的值*/
}
```

显然，按位查找算法的时间复杂度为O(1)。

2.3 线性表的链式存储结构

顺序表的特点是利用数据元素在物理位置上的邻接关系来表示结点间的逻辑关系，这样顺序表就不需要为表示结点间的逻辑关系而增加额外的空间，同时也可以直接存取表中的任一元素。但它也带有相应的缺点：

●插入和删除操作需要移动大量的结点。

●表的容量难以预先确定。在为长度变化较大的线性表预先分配空间时，只能按照最大空间需求分配，造成空间利用率低。

●造成存储空间的"碎片"。因为顺序表存储要求占用连续的存储空间，即使空闲单元总数超过了表的容量，如果不连续，也无法使用。

鉴于顺序表的这些不足，我们考虑线性表的链式存储结构。

2.3.1 链表的结构

线性表的链式存储结构简称链表（Linked List），是用一组任意的存储单元存储该线性表中的各个数据元素，存储单元可以连续，也可以不连续。因此，链表中数据元素的逻辑次序和物理次序不一定相同。为了能体现元素间的逻辑顺序，每个结点除了存储数据元素的信息外，还要存储其后继元素所在的地址信息。一个链表结点由两个域构成：存储数据元素信息的域称为数据域(data)；存储直接后继存储位置的域称为指针域(next)。

图2-6 单链表的结点结构

链表通过每个结点中的指针域将线性表的各个结点按逻辑顺序链接在一起。若链表的每个结点中只有一个指针域，这种链表称为单链表（Single Linked List），结点结构见图2-6。线性表（a_1,a_2, a_3,a_4）的链式存储结构见图2-7 (a)，但这种方法表示单链表很不方便，而且用户也没必要关心线性表中每个数据元素的实际内存地址，因此通常采用图2-7(b)的形式表示单链表的逻辑关系。

头指针 地址

head 0450

地址	
	…
0100	a_2
	0200
0200	a_3
	0600
	…
0450	a_1
	0100
	…
0600	a_4
	∧
	…

(a)

head → a_1 → a_2 → a_3 → a_4 ∧

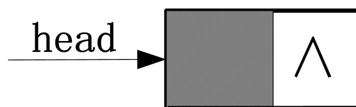
(b)

图2-7 线性表（a_1,a_2,a_3,a_4）的链式存储示意图

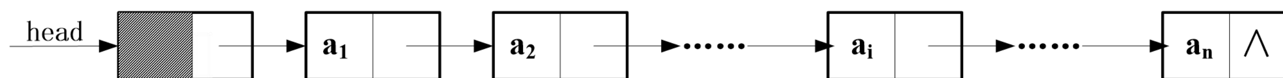
链表的存取要从头指针head开始，头指针指示链表中第一个结点（称为首结点）的存储位置；链表的最后一个结点被称为尾结点，由于其没有直接后继，因此尾结点的指针域为空（NULL），用“∧”表示。线性表（$a_1,a_2, \ldots, a_i,\ldots,a_n$）的单链表示意图如图2-8。

图2-8 单链表示意图

从示意图中可以看到，除首结点外单链表中每个结点的存储地址存放在其前驱结点的指针域中。由于首结点与其他结点的存储地址存放位置不同，因此结点的处理就有所不同。为了方便操作，在单链表第一个结点之前附加一个同结构的结点，称为头结点（front）或表头结点。头结点的数据域可以空闲，也可以用来存储如线性表长度等附加信息，但实际使用时要考虑结点数据域的数据类型设置。增加了头结点的单链表，链表头指针在空表和非空表中的处理就统一了，链表的首结点也不必进行特殊处理了。带头结点的单链表如图2-9所示。本书中，如无特殊说明，单链表的案例均采用带头结点的单链表。



(a)带头结点的空链表



(b)带头结点的非空链表
图2-9 带头结点的单链表示意图

2.3.2 单链表上实现的基本运算

在C语言中，单链表可以定义如下：

```
/*单链表的定义：*/
typedef int DataType;
 /* DataType为数据元素的类型，可以是任何类型*/
typedef struct node    /* 结点类型定义 */
{
  DataType data;    /* 结点的数据域 */
  struct node *next;  /* 结点的指针域 */
}ListNode;
typedef ListNode *LinkList;
```

1. 建表

链表的建立有以下两种方式：

(1)头插法建表

头插法建立链表是将新生成的结点插入到现有链表首结点之前，无头结点的单链表和带头结点的单链表头插法算法如下。

【算法2-7】：

```
LinkList CreateListF1(void)        /*用头插法建无头结点的单链表*/
{
  int value;
  LinkList head=NULL;                    /*设置头指针*/
  ListNode *p;                           /*p用于指向新结点*/
  printf("输入9999，结束输入！\n");
  printf("请输入数据值：\n");
  scanf("%d",&value);                    /*输入数据值*/
```

```
  while (value!=9999)
  {
   p=(ListNode *)malloc(sizeof(ListNode));     /*生成新结点*/
   if(!p)                                      /*如果新结点申请不成功，退出*/
     exit(-1);
   p->data=value;                              /*为新结点赋值*/
   p->next=head;                               /*新结点的指针指向原有链表首结点*/
   head=p;                                     /*头指针指向新结点*/
   printf("请输入数据值：\n");
   scanf("%d",&value);
  }
 return head;                                  /*返回头指针*/
}
```

【算法2-8】：

```
LinkList CreateListF2(void)        /*用头插法建带头结点的单链表*/
{
  int value;
  LinkList head;                   /*设置头指针*/
  ListNode *p;                     /*p用于指向新结点*/
  head=(ListNode*)malloc(sizeof(ListNode));   /*初始化一个空链表*/
  head->next=NULL;
  printf("输入9999，结束输入！\n");
  printf("请输入数据值：\n");
  scanf("%d",&value);              /*输入数据值*/
```

```
while (value!=9999)
 {
   p=(ListNode *)malloc(sizeof(ListNode));  /*生成新结点*/
   if(!p)                              /*如果新结点申请不成功，退出*/
     exit(-1);
   p->data=value;                      /*为新结点赋值*/
   p->next=head->next;                 /*新结点的指针指向原有链表首结点*/
   head->next=p;                       /*头结点的指针指向新结点*/
   printf("请输入数据值：\n");
   scanf("%d",&value);
 }
 return head;                          /*返回头指针*/
}
```

以带头结点单链表头插法为例，具体操作过程如图2-10。

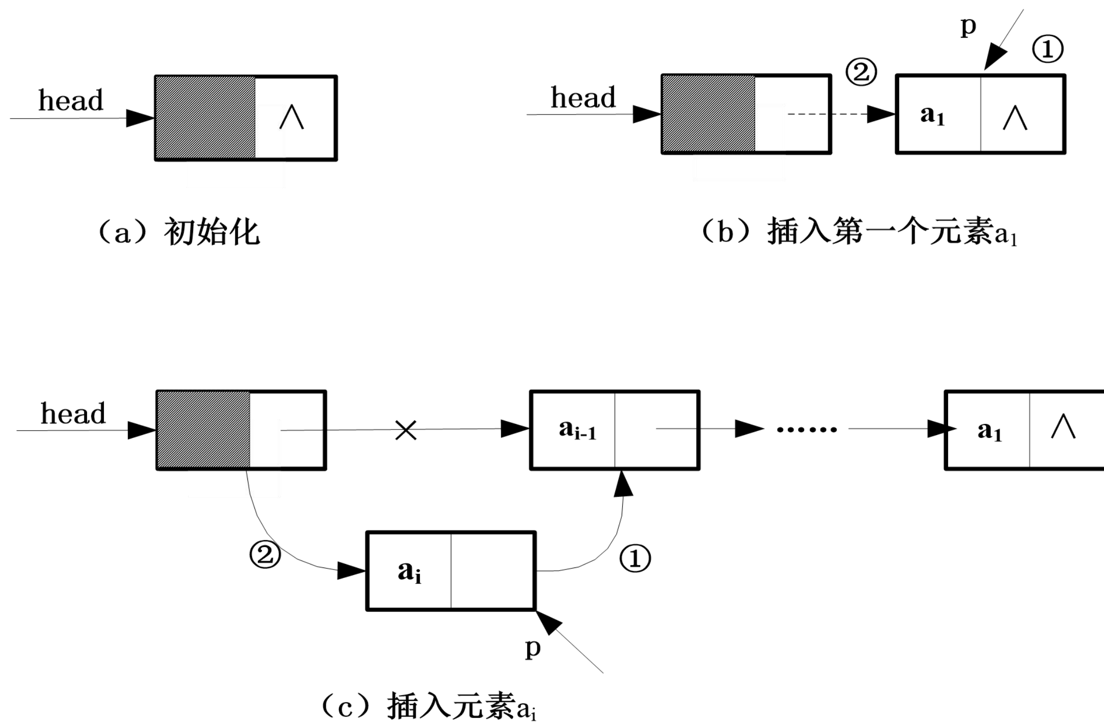


图2-10 带头结点单链表头插法示意图

(2)尾插法建表

尾插法建立链表是将新生成的结点链接到现有表尾结点之后，无头结点的单链表和带头结点的单链表尾插法算法如下。

【算法2-9】：

```
LinkList CreateListR1(void)          /*用尾插法建无头结点的单链表*/
{
  int value;
  LinkList head=NULL;                /*设置头指针*/
  ListNode *p,*r;                    /*p用于指向新结点，r用于指向表尾结点
   */
  r=NULL;
  printf("输入9999，结束输入！\n");
  printf("请输入数据值：\n");
  scanf("%d",&value);                /*输入数据值*/
```

```
 while (value!=9999)
{
  p=(ListNode *)malloc(sizeof(ListNode));   /*生成新结点*/
  if(!p)                       /*如果新结点申请不成功，退出*/
    exit(-1);
  p->data=value;                    /*为新结点赋值*/
  if (head==NULL)
      head=p;                       /*新结点插入空表*/
  else
      r->next=p;                    /*新结点接在表尾*/
  r=p;                              /*r指向新结点*/
  printf("请输入数据值：\n");
  scanf("%d",&value);
}
if(r!=NULL)
  r->next=NULL;                     /*表尾结点指针置空*/
return head;                        /*返回头指针*/
}
```

【算法2-10】：

```
LinkList CreateListR2(void)          /*用尾插法建带头结点的单链表*/
{
   int value;
   LinkList head;                              /*设置头指针*/
   ListNode *p,*r;                    /*p用于指向新结点，r用于指向表尾结点
*/
   head=(ListNode*)malloc(sizeof(ListNode));      /*生成头结点*/
   r=head;
   printf("输入9999，结束输入！ \n");
   printf("请输入数据值：\n");
   scanf("%d",&value);                          /*输入数据值*/
```

```
    while (value!=9999)
    {
        p=(ListNode *)malloc(sizeof(ListNode));    /*生成新结点*/
        if(!p)                         /*如果新结点申请不成功，退出*/
          exit(-1);
        p->data=value;                       /*为新结点赋值*/
        r->next=p;                           /*新结点接在表尾*/
        r=p;                                 /*r指向新结点*/
        printf("请输入数据值：\n");
        scanf("%d",&value);
    }
    r->next=NULL;                            /*表尾结点指针置空*/
    return head;                             /*返回头指针*/
}
```

以带头结点单链表尾插法为例，具体操作过程如图2-11。

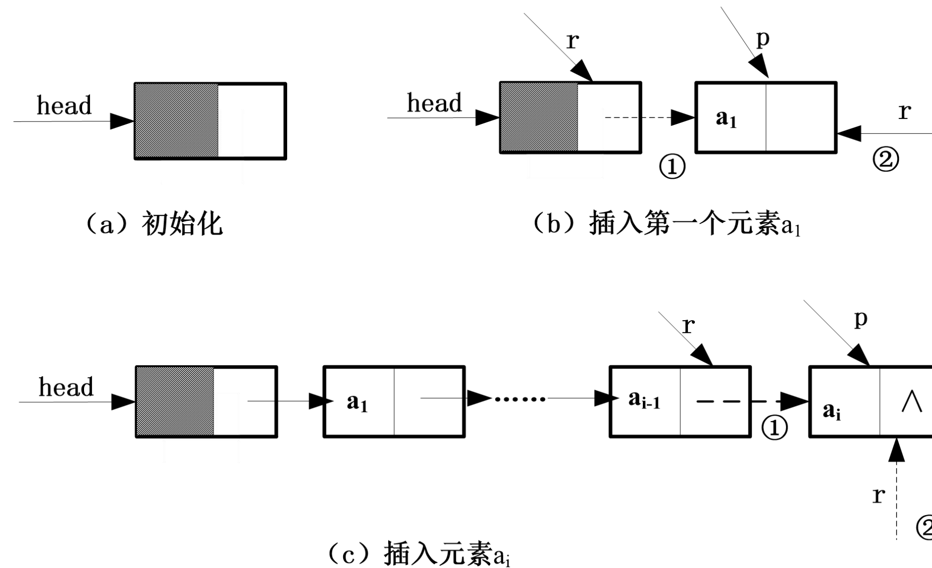


图2-11 带头结点单链表尾插法示意图

以上四种建表算法中，问题规模都取决于表长n，基本语句为while循环中新结点插入的语句。因此，算法的平均时间复杂度均为O(n)。

2.插入

将新元素x插入到链表中的数据元素a_{i-1} 和 a_i 之间，实现链表的插入运算。因此，首先要遍历单链表，找到a_{i-1}的存储地址r；然后生成新结点p，将其数据域赋值为x；结点p的next指针指向r的next；再将结点r的next指向p。具体步骤如图2-12所示。

（a）插入前

（b）生成新结点p，并赋值x

（c）插入结点p

（d）修改a_{i-1} 的指针

图2-12 单链表的插入

【算法2-11】：

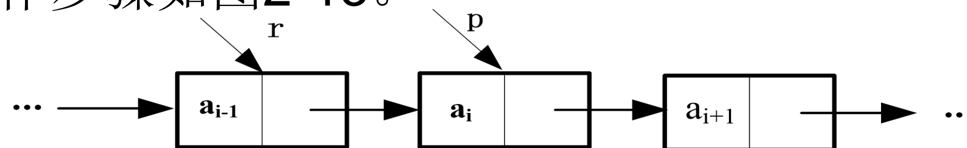
```
void InsertList(LinkList head,DataType x,int i)
/*将值为x的新结点插入到带头结点的单链表head的第i个结点的位置上*/
{
    int j;                          /*j用于记录当前结点位置*/
    ListNode *p,*r;
    r=head;
    j=0;
    while(r->next && j<i-1)                /*寻找第i-1个结点*/
    {
       r=r->next;
       j++;
    }
```

```
    if(j!=i-1)
    {
       printf("插入位置非法\n");
       exit(0);
    }
    p=(ListNode *)malloc(sizeof(ListNode));        /*生成新结点*/
    if(!p)                                         /*如果新结点申请不成功，退出*/
      exit(-1);
    p->data=x;                                     /* 为新结点p赋值x */
    p->next=r->next;                               /* 插入结点p */
    r->next=p;                                     /* 修改r的指针 */
}
```

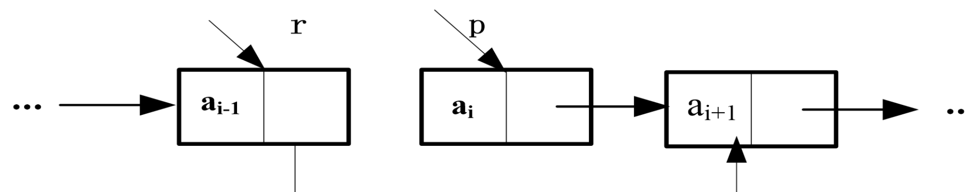
本算法的问题规模是表长n，基本语句为while循环中用于寻找第i-1个结点的语句。因此，算法的平均时间复杂度为O(n)。

3.删除

如果要删除单链表中的数据元素a_i，需要找到指向待删除元素的指针p及指向其直接前驱结点的指针r，再将r的next指针指向p的后继，释放结点p。操作步骤如图2-13。



（a）删除a_i前



（b）r的指针指向p的后继

（c）释放p

图2-13 单链表的删除

【算法2-12】：

```
void DeleteList(LinkList head,DataType x)/*删除带头结点的单链表中值等于x的结点*/
{
   ListNode *p,*r;                    /*p指向查找的结点，r指向p的前驱结点*/
   r=head;
   p=head->next;
   while(p && p->data!=x)             /*查找值等于x的结点*/
   {
     p=p->next;
     r=r->next;
    }
   if(p==NULL)                        /*如果p为空，查找失败*/
   {
     printf("待删除结点不存在！ \n");
     exit(0);
   }
   r->next=p->next;                   /*r的next指针指向p的后继 */
   free(p);                           /*释放结点p*/
}
```

本算法的问题规模是表长n，基本语句为while循环中用于查找值等于x的结点的语句。算法的平均时间复杂度为O(n)。

4.查找

单链表上的查找与顺序表的查找不同，不能实现随机查找，要找到某个元素，只能从表头开始查找，属于顺序查找。

（1）按值查找

在单链表中，每一个数据元素的存储位置都存放在其直接前驱结点的指针域中。因此，单链表的按值查找运算需要从表首结点开始，依次将表中结点的数据域与给定值比较，直到某个结点的数据域等于给定值，则查找成功，返回指向该结点的指针；若查过表尾仍未找到，则查找失败，返回NULL。

【算法2-13】：

```
LinkList LocNode(LinkList head,DataType x)
/*在带头结点的单链表head中查找其值为x的结点*/
{
  ListNode *r=head->next;            /*设置比较的指针r从链表首结点开始*/
  while (r&&r->data!=x)             /*直到r为NULL或r->data等于x为止*/
    r=r->next;
  return r;
}
```

（2）按位查找

单链表的按位查找需要从表头结点开始，依次用给定的序号比较表中结点的序号，当查找成功时，返回结点的地址；否则，返回NULL。

【算法2-14】：

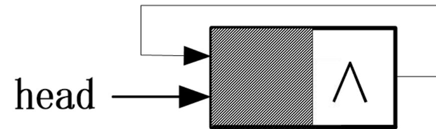
```
LinkList GetNode(LinkList head,int i)/*在带头结点的单链表head中查找第i个结点*/
{
  int j=0;                          /*设置计数器j，赋初值为0*/
  ListNode *r=head;
  while (r->next&&j<i)              /*直到r->next为NULL或j等于i为止*/
  {
    r=r->next;
    j++;
  }
  if (j==i)
    return r;
  else
    return NULL;
}
```

两种查找都需要从表头结点依次向后比较，因此问题规模均为表长n，时间复杂度均为O(n)。

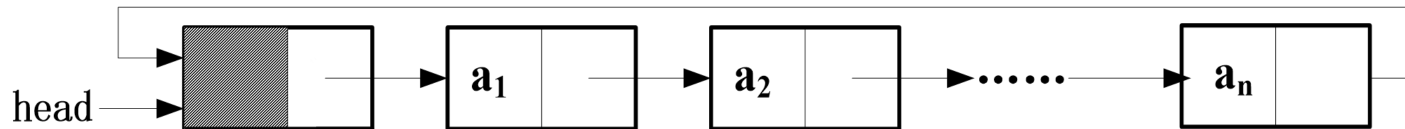
2.3.3 循环链表

将单链表中的最后一个结点的指针指向链表中第一个结点，使整个链表构成一个环形，这种链表称为单循环链表，简称循环链表（Circular Linked List）。

从循环链表中的任意一个结点出发都可以找到表中其他结点。为了使空表与非空表的处理统一，通常循环链表也附设一个头结点，如图2-14。有时，在单循环链表中只设指向尾结点的尾指针rear而不设头指针，这样对链表头结点和尾结点的操作都变得方便了。



(a)空循环链表



(b)非空循环链表

图2-14 循环链表示意图

循环链表的运算和单链表基本一样，差别在于：当需要从头到尾扫描整个链表时，是否达到表尾的条件不同。在单链表中找表尾结点要判断某结点链域值是否为“空”，在循环链表中找表尾结点则要判断某结点的链域值是否等于头指针。

在循环链表中，从表中任一结点p出发，都可以找到它的直接前驱结点。算法如下：

【算法2-15】：

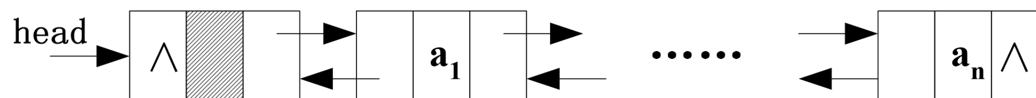
```
LinkList prior(ListNode *p)
/*求循环链表中任意一个结点的前驱结点*/
{
  ListNode *s;
  s=p->next;                    /*初始化s为p的直接后继*/
  while(s->next!=p)             /*当s的直接后继为p时，s是p的直接前驱*/
    s=s->next;
  return s;
}
```

显然，本算法的时间复杂度为O(n)。

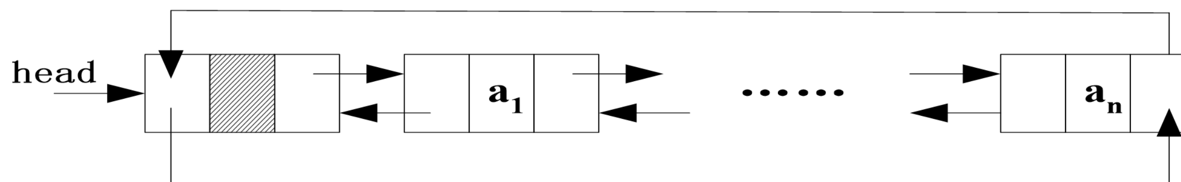
2.3.4 双链表

在单链表和循环链表中，数据元素的结点除数据域外，只有一个指向其直接后继的指针域，若要查找其前驱结点就需要遍历链表。为了解决这种单向性的问题，可以在单链表的结点中增加一个指向其直接前驱结点的指针域，这样有两种不同方向链的链表就称为双（向）链表（Double Linked List），其结点结构如图2-15 (a)。其中，data：数据域，存放数据元素；prior：前驱指针域，存放该结点的前驱结点地址；next：后继指针域，存放该结点的后继结点地址。给双链表加一表头结点成为带表头结点的双链表，如图2-15(b)。如果每条链都构成循环链表，就形成了双循环链表，如图2-15 (c)。

(a)双链表结点示意图



(b)带表头结点的双链表



(c)带表头结点的双循环链表

图2-15 双链表示意图

双链表和单链表相比，每一个结点增加了一个指针域，虽然多占用了空间，但它给数据运算带来了方便。在双链表中，如果只涉及单向的指针，其运算与单链表的算法一致。如果运算涉及两个方向的指针，由于双链表的对称结构，其插入和删除操作都很容易。双链表有一个重要的特点，若p是指向表中任一结点的指针，则有：

(p->next)->prior==(p->prior)->next==p

在C语言中，双链表可以定义如下：

```
typedef int DataType;  /* DataType为数据元素的类型，可以是任何类型的数据 */
typedef struct Dnode    /* 结点类型定义 */
{
  DataType data;      /* 结点的数据域 */
  struct Dnode *prior;  /* 结点的前驱指针域 */
  struct Dnode *next;  /* 结点的后继指针域 */
}DulListNode;
typedef DulListNode *DulLinkList;
```

1、插入

在双链表中结点s的后面插入新结点p，需要修改4个指针：

①p->prior=s;
②p->next=s->next;
③s->next->prior=p;
④s->next=p;

在涉及链表指针的操作中，应注意修改指针的顺序，以避免出现丢失后半段链表的情况。在修改第②和③步指针时，要用到s->next以指向s的后继结点，所以第④步的操作要在第②和③步指针修改完成后进行，而第②和③步的操作顺序可以互换。操作步骤如图2-16所示。

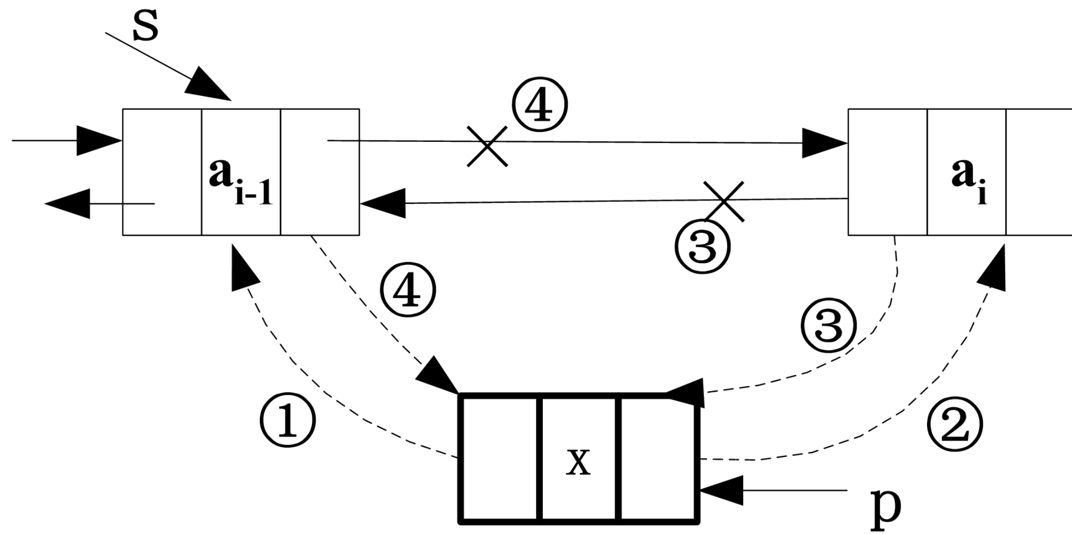


图2-16 双链表插入操作示意图

2、删除

在双链表中删除结点s，可以采用如下语句完成：

①(s->next)->prior=s->prior;

②(s->prior)->next=s->next;

③free(s);

第①和②步可以互换，如图2-17所示。

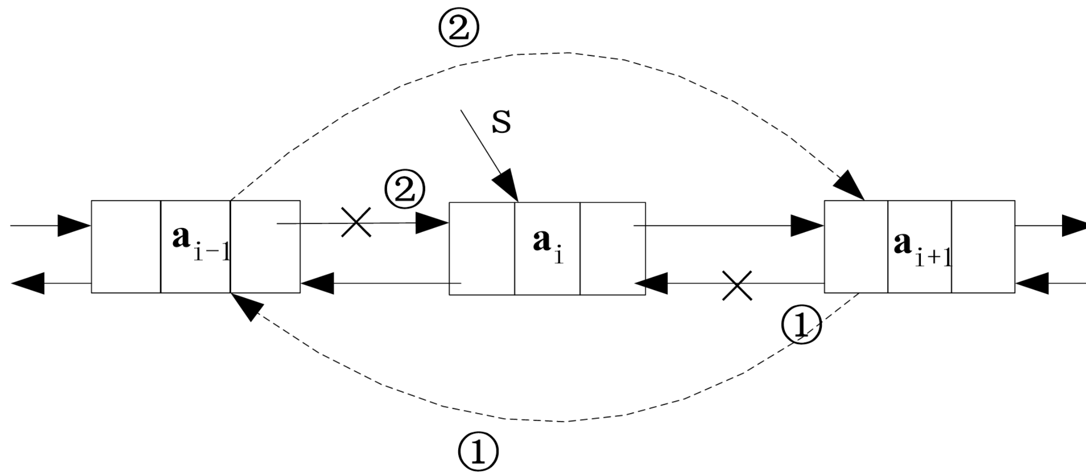


图2-17 双链表删除操作示意图

2.4 顺序表与链表的比较

1、考虑时间因素

顺序表的查找运算是随机操作，对表中任一结点都可以直接存取，而链表中的结点需要从头指针起沿着链表扫描才能找到。所以，当线性表的操作主要是进行查找，很少做插入和删除操作时，宜采用顺序表作为存储结构；对于频繁进行插入和删除的线性表，宜采用链表做存储结构；若表的插入和删除主要发生在表的首尾两端，则宜采用尾指针表示的单循环链表作为存储结构。

2、考虑空间因素

顺序表的存储空间是静态分配的，在程序执行之前必须明确定义其存储规模。如果估计太小可能造成空间溢出，估计太大将造成空间浪费。链表的存储空间是动态分配的，只要系统内存尚有空闲，就不会产生溢出。

当线性表的长度变化较大，难以估计其存储规模时，宜采用动态链表作为存储结构；当线性表的长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表作为存储结构。

存储密度（Storage Density）是指结点数据本身所占的存储量与整个结点结构所占的存储量之比。动态链表存储密度小于1，顺序表存储密度等于1。显然，存储密度越大，存储空间的利用率就越高。

3、考虑程序设计语言

从计算机程序语言看，绝大多数高级语言都提供数组类型，因此顺序表的实现相对简单一些。若无指针类型，可以采用静态链表的方法来模拟动态存储结构。

2.5 案例实现

2.5.2 案例实现1——用顺序表实现通讯录管理

见教材

2.5.3案例实现2——用链表实现通讯录管理

见教材

本章小结

1、 线性表是最基本最常用的数据结构。线性表的数据元素之间有一对一的对应关系。在非空表中，除了首结点，每个结点都有且只有一个前驱结点；除了尾结点外，每个结点都有且只有一个后继结点。线性表的存储结构通常选用顺序存储结构和链式存储结构。

2、 顺序表是用一组地址连续的存储单元依次存放线性表的数据元素，即保持元素同构且无缺项。顺序表是一种随机的存储结构。

3、 链表的特点是用一组任意的存储单元存储该线性表中的各个数据元素，存储单元可以连续，也可以不连续。一个单链表结点由数据域和指针域构成，利用指针表示数据元素间的逻辑关系。

4、 若线性表的操作主要是进行查找，很少做插入和删除操作时，以采用顺序表做存储结构为宜。对于频繁进行插入和删除的线性表，宜采用链表做存储结构。当线性表的长度变化较大，难以估计其存储规模时，以采用动态链表作为存储结构为好。当线性表的长度变化不大，易于事先确定其大小，为了节约存储空间，宜采用顺序表作为存储结构。