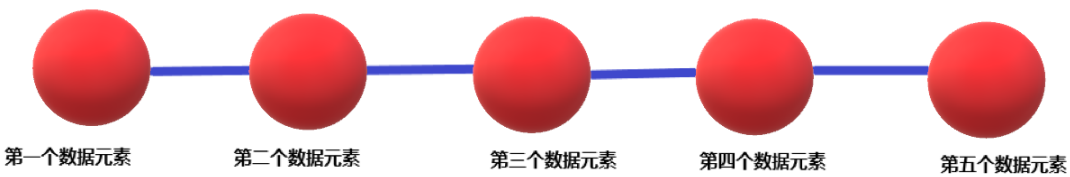
接着复习408基础知识，今天是数据结构与算法的线性表。

1、基本概念

线性表是具有**相同特性数据元素的一个有限序列**。（别的教材中也写作：由n（n>=0）个数据特性相同的元素构成的有限序列称为线性表）

- 相同特性：指的是**数据元素的数据类型相同。**
  - 数据元素都是一个数据类型。要么都是整型的（数组a[4]={1,2,3,4};），要么都是同一种结构体类型的，姓名+电话号码就可以构成一个结构体类型（struct phonebook{char name[100];int phonenumber; };）
- 有限：表中的数据元素个数为n（也叫做线性表的长度，n>=0），是有限个元素。当线性表长度n=0时，此时线性表是一个空表。
- 序列：数学上，序列是被排成一列的对象（或事件）；这样每个元素不是在其他元素之前，就是在其他元素之后；元素之间的顺序非常重要。除第一个数据元素和最后一个数据元素之外的每个数据元素有则仅有一个前驱数据元素和一个后继数据元素，第一个数据元素只有一个后继的数据元素，最后一个数据元素只有一个前驱的数据元素



总结线性表的特点如下：

（1）表中元素个数有限。

（2）表中元素具有逻辑上的顺序性，表中元素有其先后次序。

（3）表中元素都是数据元素，每个元素都是单个元素。

（4）表中元素的数据类型都相同，这意味着每个元素占有相同大小的存储空间。

（5）表中元素具有抽象性，即仅讨论元素间的逻辑关系，而不考虑元素究竟表示什么内容。

注： 线性表是一种逻辑结构，表示元素之间一对一的**相邻关系。**顺序表和链表是指存储结构，两者属于不同层面的概念。

2、线性表的存储结构

(1) 顺序表示和实现

线性表的顺序表示指的是用一组地址连续的存储单元依次存储线性表的数据元素，这种表示也称作线性表的顺序存储结构或顺序映像，同时，**顺序存储的线性表被称为顺序表**。（数组）

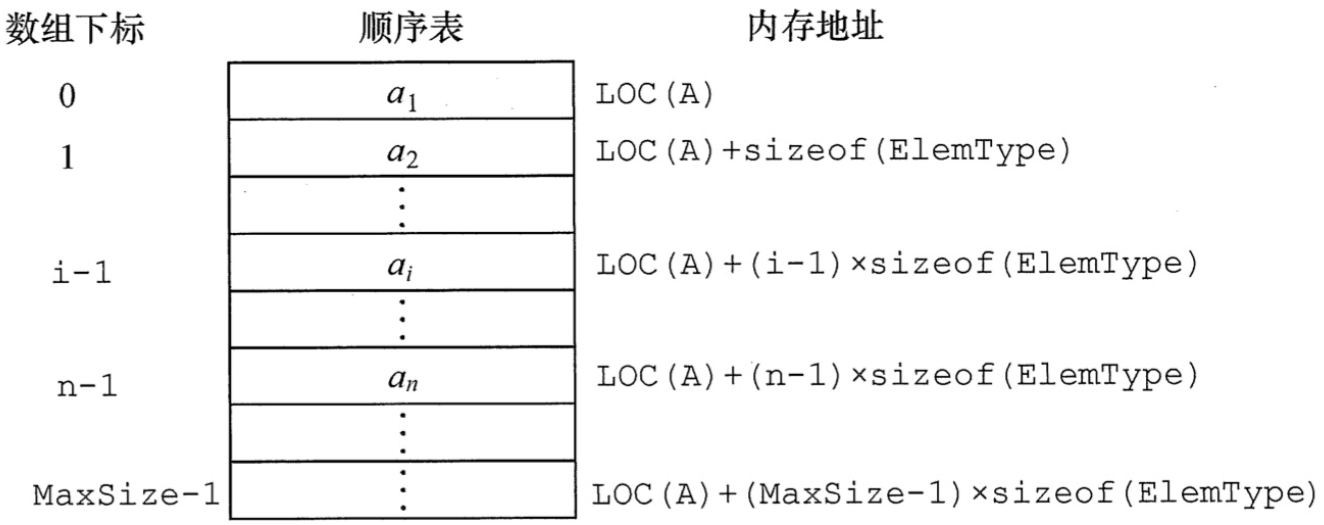


图 2.1　线性表的顺序存储结构


```
#define InitSize 100
typedef struct{
    ElemType data[MaxSize];
    int length;
}SqList;
```

一维数组可以是静态分配的，也可以是动态分配的。在静态分配时，由于数组的大小和空间事先已经固定，一旦空间占满，再加入新的数据将会产生溢出，进而导致程序崩溃。

在动态分配时，存储数组的空间是在程序执行过程中通过动态存储分配语句分配的，一旦数据空间占满，就另外开辟一块更大的存储空间，用以替换原来的存储空间，从而达到扩充存储数组空间的目的，而不需要为线性表一次性地划分所有空间。

```
#define InitSize 100
typedef struct{
    Elemtype *data;
    int MaxSize,length;
}SqList;
```

动态分配不是链式存储，它同样属于顺序存储结构，物理结构没有变化，依然是随机存取方式，只是分配的空间大小可以在运行时决定。

顺序表最主要的特点是**随机访问**，即通过首地址和元素序号可在时间 `O(1)` 内找到指定的元素。顺序表的存储密度高，每个结点只存储数据元素。顺序表逻辑上相邻的元素物理上也相邻，所以插入和删除操作需要移动大量元素。

一些简单的时间复杂度分析，这里就不分析了，在没有利用任何算法（如：二分）的前提下，任何操作都是 `O(n)` 级别的。

（2）线性表的链式表示

顺序表可以随时存取表中的任意一个元素，它的存储位置可以用一个简单直观的公式表示，但插入和删除的操作需要移动大量元素。

而链式存储线性表时，不需要使用地址连续的存储单元，即不要求逻辑上相邻的元素在物理位置上也相邻，它通过“链”建立起数据元素之间的逻辑关系，因此插入和删除操作不需要移动元素，而只需要修改指针，但也会失去顺序表可随机存取的优点。

线性表的链式存储又称单链表，它是指通过一组任意的存储单元来存储线性表中的数据元素。为了建立数据元素之间的线性关系，对每个链表结点，除存放元素自身的信息外，**还需要存放一个指向其后继的指针。**

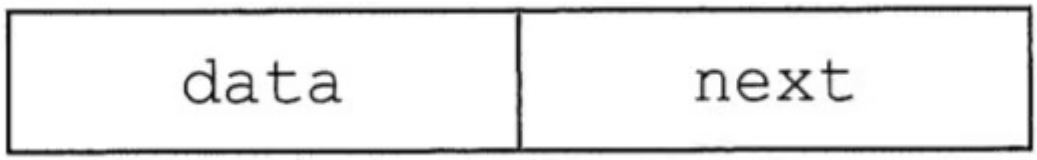


图 2.3　单链表结点结构

其中data为数据域，存放数据元素；next为指针域，存放其后继结点的地址。

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
```

利用单链表可以解决顺序表需要大量连续存储单元的缺点，但单链表附加指针域，也存在浪费存储空间的缺点。由于**单链表的元素离散的分布在存储空间**中，所以单链表是非随机存取的存储结构，即不能直接找到表中某个特定的结点。查找某个特定的结点时，需要从表头开始遍历，依次查找。

通常用头指针来标识一个单链表，如单链表L，头指针为NULL时，表示一个空表。

为了操作上的方便，在单链表第一个结点之前附加一个结点，称为头结点，头结点的数据域可以不设任何信息，也可以记录表长等信息。头结点的指针域指向线性表的第一个元素结点，如图所示：

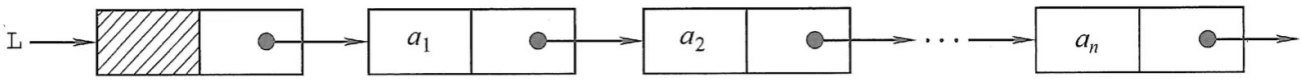


图 2.4　带头结点的单链表

> 头结点和头指针区别：
>
> > **头指针**
> >
> > > - 头指针是指链表指向第一个结点的指针。**若链表有头结点，则是指向头结点的指针。**
> > > - 头指针具有标识作用，所以头指针冠以链表的名字（指针变量的名字）
> > > - 无论链表是否为空，头指针均不为空 。
> > > - 头指针是链表的必要元素。
> >
> > **头节点**
> >
> > > - 头结点是为了操作的统一和方便而设立的，放在第一个元素的结点之前，其数据域一般无意义（但也可以用来存放链表的长度）。
> > > - 有了头结点，对在第一元素结点前插入结点和删除第一结点起操作与其它结点的操作就统一了。
> > > - 头结点不一定是链表的必要元素。

- 头插法创建链表（赋值）：

```
LinkList List_HeadInsert(LinkList &L){
    LNode *s;
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    scanf("%d",&x);
    while(x != 999999){
        s = (LNode*)malloc(sizeof(LNode));
        s -> data = x;
        s -> next = L -> next;
        L -> next = x;
        scanf("%d",&x);
    }
    return L;
}
```

采用头插法建立单链表时，读入数据的顺序与生成的链表中的元素的顺序是相反的。每个结点插入的时间为`O(1)`，设单链表长为`n`，则总时间复杂度为`O(n)`。

- 尾插法创建链表：

头插法建立单链表的算法虽然简单，单生成的链表中结点的次序和输入数据的顺序不一致。若希望两者次序一致，可以采用尾插法。该方法将新结点插入法到当前链表的尾表，为此必须增加一个尾指针r，使其始终指向当前链表的尾结点。

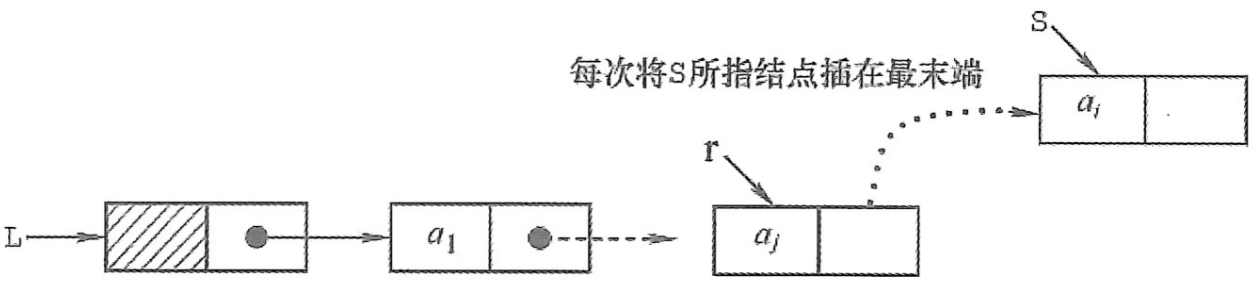


图 2.6　尾插法建立单链表

```
LinkList List_TailInsert(LinkLiist &L){
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    //是由系统生产一个LNode型的结点，同时在该结点的其实位置赋值给变量指针s。
    LNode *s,*r = L;
    scanf("%d",&x);
    while(x!=999999){
        s=(LNode*)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s;
        scanf("%d",&x);
    }
    r->next=NULL;
    return L;
}
```

- 插入节点:

算法首先调用按序号查找算法 `GetElem(L, i - 1)`，查找第 `i - 1` 个结点。假设返回的第 `i - 1` 个结点为 `*p`，然后令新结点 `*s` 的指针域指向 `*p` 的后继结点，再令结点 `*p` 的指针域指向新插入的结点 `*s`。

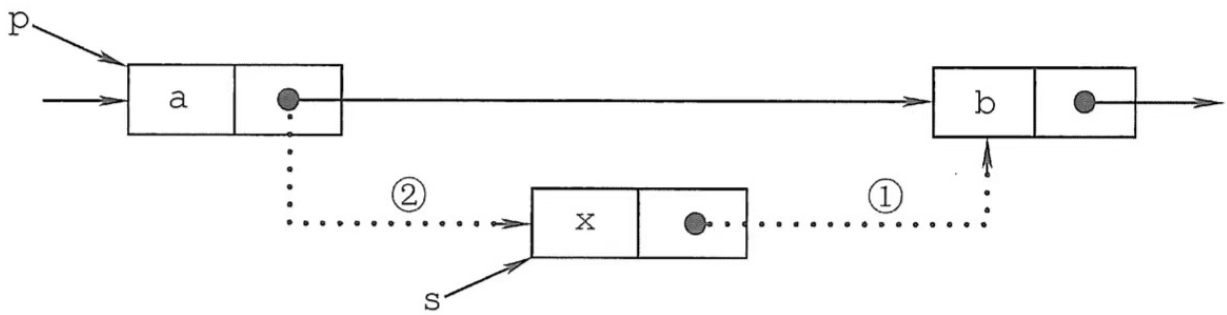


图 2.7　单链表的插入操作

① p=GetElem(L, i - 1);

② s->next=p->next;

③ p->next=s;

因为附设了一个指向表尾结点的指针，故时间复杂度和头插法的相同为 `O(n)`。

- 删除结点操作:

删除结点操作是将单链表的第i个结点删除。先检查删除位置的合法性，后查找表中第i-1个结点，即被删除结点的前驱结点，再将其删除。

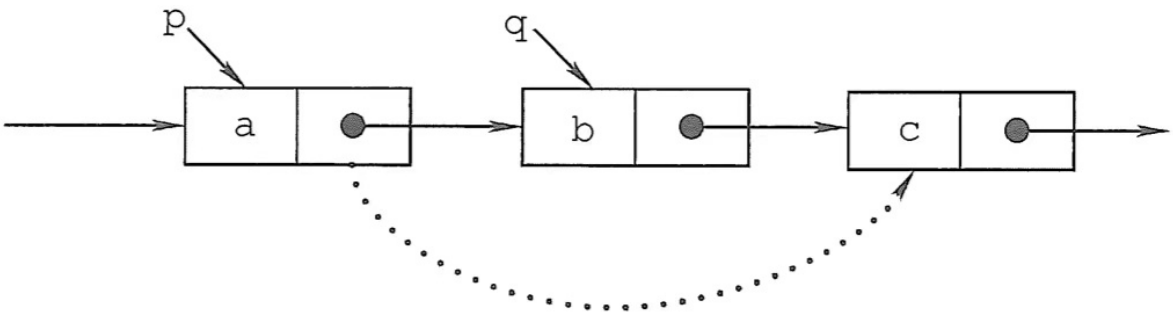


图 2.8　单链表结点的删除

假设结点`*p`为找到的被删结点的前驱结点，为实现这一操作后的逻辑关系的变化，仅需修改`*p`的指针域，即将`*p`的指针域next指向`*q`的下一结点。

```
p = GetElem(L, i - 1);
q = p -> next;
p -> next = q -> next;
free(q);
```

和插入算法一样，该算法的主要时间也耗费在查找操作上，时间复杂度为`O(n)`。

单链表结点中只有一个指向其后继的指针，使得单链表只能从头结点依次顺序地向后遍历。要访问某个结点的前驱结点（插入、删除操作时），只能从头开始遍历，访问后继结点的时间复杂度为`O(1)`，访问前驱结点的时间复杂度为`O(n)`。

为了克服单链表的缺点，引入双链表，双链表结点中有两个指针prior和next，分别指向其前驱结点和后继结点，如图所示。

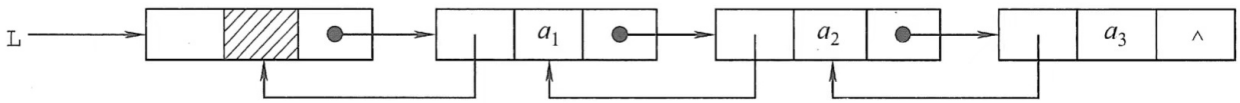


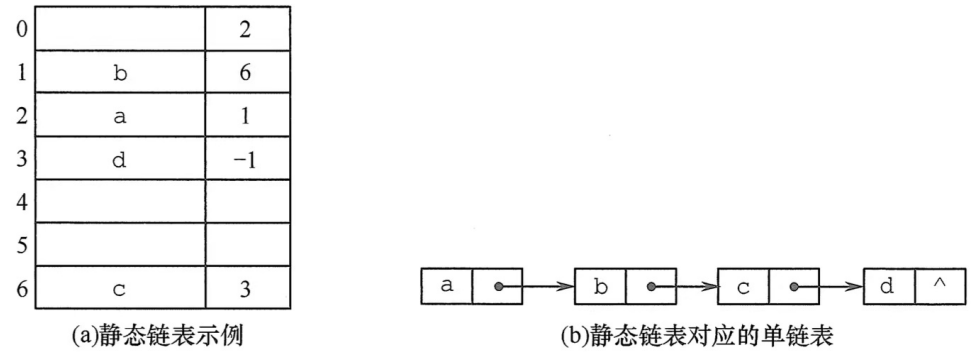
图 2.9　双链表示意图


```
typedef struct DNode{
        ElemType data;
        struct DNode *prior,*next;
}DNode,*DLinkList;
```

双链表在单链表的结点中增加了一个指向其前驱的prior指针，因此，双链表中的按值查找和按位查找的操作与单链表的相同。 但双链表在插入和删除的操作的实现上，与单链表由着较大的不同，这是因为“链”变化时也需要对prior指针做出修改，其关键是保证在修改的过程中不断链。此外，双链表可以很方便的找到其前驱结点，因此，插入、删除操作的时间复杂度为`O(1)`。

静态链表借助数组来描述线性表的链式存储结构(链式向前星)，结点也有数据域data和指针域next，与前面说的链表中的指针不同的是，这里的指针是结点的相对地址（数组下标），又称**游标**。和顺序表一样，静态链表也要预先分配一块连续的内存空间。

静态链表和单链表的对应关系如图所示：



(a)静态链表示例　　(b)静态链表对应的单链表

图 2.14　静态链表存储示意图

静态链表以next==1作为其结束的标志。静态链表的插入、删除操作与动态链表的相同，只需要修改指针，而不需要移动元素。

总体来说，静态链表没有单链表使用起来方便，但在一些不支持指针的高级语言（如Basic）中，是一种非常巧妙的设计方法。

```
#define MaxSize 50
typedef struct{
    ElemType data;
    int next;
}SlinkList[MaxSize];
```

(3) 顺序表与链表的比较

- 存取（读写）方式：顺序表可以顺序存取，也可以随机存取，链表只能从表头顺序存取元素。
    - 例如在第 `i` 个位置上执行或存取的操作，顺序表仅需一次访问，而链表则需从表头开始依次访问i次。
- 逻辑结构与物理结构：采用顺序存储时，逻辑上相邻的元素，对应的物理存储位置也相邻。而采用链式存储时，逻辑上相邻的元素，物理位置则不一定相邻，对应的逻辑关系是通过指针链接来表示的。
- 查找和删除操作：
    - 对于按值查找，顺序表无序时，两者的时间复杂度均为 `O(n)`；顺续表有序时，可采用折半查找，此时的时间复杂度为 `O(logn)`。

- 对于按序号查找，顺序表支持随机访问，时间复杂度仅为`O(1)`，而链表的平均时间复杂度为`O(n)`。
- 顺序表的插入、删除操作，平均需要移动半个表长的元素。链表的插入、删除操作，只需修改相关结点的指针域即可。由于链表的而每个结点都带有指针域，故而存储密度不够大。

- 空间分配：
  - 顺序存储在静态存储分配情形下，一旦存储空间装满就不能扩充，若再加入新元素，则会出现内存溢出，因此需要预先分配足够大的存储空间。
  - 预先分配过大，可能会导致顺序表后部大量元素闲置；预先分配过小，又会造成溢出。动态存储分配虽然存储空间可以扩充，但需要移动大量元素，导致操作效率降低，而且若内存中没有更大块的连续存储空间，则会导致分配失败。
  - 链式存储的结点空间只在需要时申请分配，只要内存中有空间就可以分配，操作灵活、高效。