线性表的顺序表示

线性表的定义

线性表的顺序存储又称顺序表。它是用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。第1个元素存储在顺序表的起始位置，第i个元素的存储位置后面紧接着存储的是第i+1个元素，称i为元素ai在顺序表中的位序。因此，顺序表的特点是表中元素的逻辑顺序与存储的物理顺序相同。

假设顺序表L存储的起始位置为LOC(A)，sizeof(ElemType)是每个数据元素所占用存储空间的大小，则表L所对应的顺序存储如下图所示。

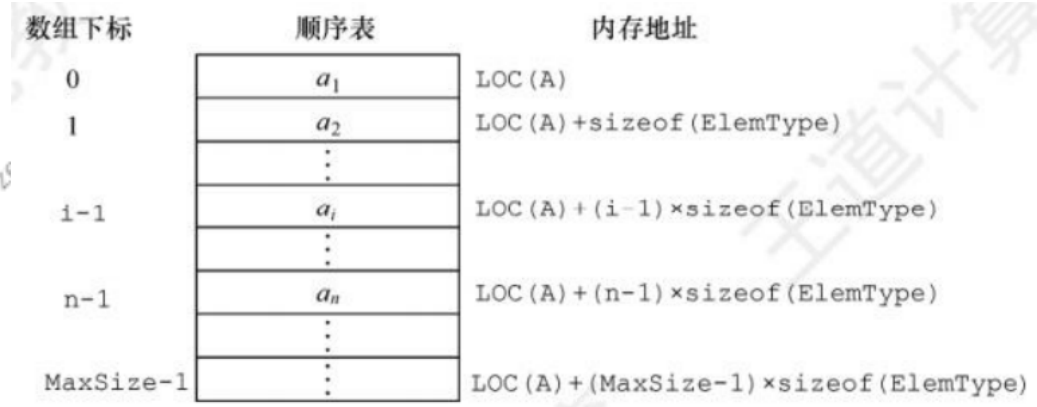

图 2.1　线性表的顺序存储结构

每个数据元素的存储位置都和顺序表的起始位置相差一个和该数据元素的位序成正比的常数，因此，顺序表中的任意一个数据元素都可以随机存取，所以线性表的顺序存储结构是一种随机存取的存储结构。通常用高级程序设计语言的数组来描述线性表的顺序存储结构。

线性表中元素的位序是从1开始的，而数组中元素的下标是从0开始的。

假定线性表的元素类型为ElemType，则静态分配的顺序表存储结构描述为

```
#define MaxSize 50//定义线性表的最大长度
typedef struct{
    ElemType data[MaxSize];//顺序表的元素
    int length;//顺序表的当前长度
}SqList;//顺序表的类型定义
```

一维数组可以是静态分配的，也可以是动态分配的。对数组进行静态分配时，因为数组的大小和空间事先已经固定，所以一旦空间占满，再加入新数据就会产生溢出，进而导致程序崩溃。

而在动态分配时，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，将原表中的元素全部拷贝到新空间，从而达到扩充数组存储空间的目的，而不需要为线性表一次性地划分所有空间。

动态分配的顺序表存储结构描述为

```
#define InitSize 100//表长度的初始定义
typedef struct{
    ElemType *data;//指示动态分配数组的指针
    int MaxSize, length;//数组的最大容量和当前个数
}SeqList;//动态分配数组顺序表的类型定义
```

C的初始动态分配语句为

```
L.data=(ElemType*)malloc(sizeof(ElemType)*InitSize);
```

C++的初始动态分配语句为

```
L.data=new ElemType[InitSize];
```

动态分配并不是链式存储，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动态决定。

优点	缺点
可进行随机访问，即可通过首地址和元素序号在O(1)时间内找到指定的元素	元素的插入和删除需要移动大量的元素，插入操作平均需要移动n/2个元素，删除操作平均需要移动(n-1)/2个元素
存储密度高，每个结点只存储数据元素	顺序存储分配需要一段连续的存储空间，不够灵活

顺序表上的基本操作的实现

这里仅讨论顺序表的初始化、插入、删除和按值查找，其他基本操作的算法都很简单。

在各种操作的实现中（包括严蔚敏老师撰写的教材），往往可以忽略边界条件判断、变量定义、内存分配不足等细节，即不要求代码具有可执行性，而重点在于算法的思想。

顺序表的初始化

静态分配和动态分配的顺序表的初始化操作是不同的。静态分配在声明一个顺序表时，就已为其分配了数组空间，因此初始化时只需将顺序表的当前长度设为0。

```
//SqList L;//生命一个顺序表
void InitList(SqList &L){
    L.length=0;//顺序表初始长度为0
}
```

动态分配的初始化为顺序表分配一个预定义大小的数组空间，并将顺序表的当前长度设为0。MaxSize指示顺序表当前分配的存储空间大小，一旦因插入元素而空间不足，就进行再分配。

```
void InitList(SeqList &L){
    L.data=(ElemType *)malloc(MaxSize*sizeof(ElemType));//分配存储空间
    L.length=0;//顺序表初始长度为0
    L.MaxSize=InitSize;//初始存储容量
}
```

插入操作

在顺序表L的第i(1<=i<=L.length+1)个位置插入新元素e。若i的输入不合法，则返回false，表示插入失败；否则，将第i个元素及其后的所有元素依次往后移动一个位置，腾出一个空位置插入新元素e，顺序表长度增加1，插入成功，返回true。

```
bool ListInsert(SqList &L,int i,ElemType e){
    if(i<1||i>L.length+1)//判断i的范围是否有效
        return false;
    if(L.length>=MaxSize)//当前存储空间已满，不能插入
        return false;
    for(int j=L.length;j>=i;j--)//将第i个元素及之后的元素后移
        L.data[j]=L.data[j-1];
    L.data[i-1]=e;//在位置i处放入e
    L.length++;//线性表长度加1
    return true;
}
```

区别顺序表的位序和数组下标。为何判断插入位置是否合法时if语句中用length+1，而移动元素的for语句中只用length?

最好情况：在表尾插入（即i=n+1），元素后移语句将不执行，时间复杂度为O(1)。

最坏情况：在表头插入（即i=1），元素后移语句将执行n次，时间复杂度为O(n)。

平均情况：假设pi(pi=1/(n+1))是在第i个位置上插入一个结点的概率，则在长度为n的线性表中插入一个结点时，所需移动结点的平均次数为

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

因此，顺序表插入算法的平均时间复杂度为O(n)。

删除操作

删除顺序表L中第i(1<=i<=L.length)个位置的元素，用引用变量e返回。若i的输入不合法，则返回false；否则，将被删元素赋给引用变量e，并将第i+1个元素及其后的所有元素依次往前移动一个位置，返回true。

```
bool ListDelete(SqList &L, int i, ElemType &e){
    if(i<1||i>L.length)//判断i的范围是否有效
        return false;
    e=L.data[i-1];//将被删除的元素赋值给e
    for(int j=i;j<L.length;j++)//将第i个位置后的元素前移
        L.data[j-1]=L.data[j];
    L.length--;//线性表长度减1
    return true;
}
```

最好情况：删除表尾元素（即i=n），无须移动元素，时间复杂度为O(1)。

最坏情况：删除表头元素（即i=1），需移动除表头元素外的所有元素，时间复杂度为O(n)。

平均情况：假设pi(pi=1/n)是删除第i个位置上结点的概率，则在长度为n的线性表中删除一个结点时，所需移动结点的平均次数为

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

因此，顺序表删除算法的平均时间复杂度为O(n)。

可见，顺序表中插入和删除操作的时间主要耗费在移动元素上，而移动元素的个数取决于插入和删除元素的位置。下图所示为一个顺序表在进行插入和删除操作前、后的状态，以及其数据元素在存储空间中的位置变化和表长变化。a中，将第4个至第7个元素从后往前依次后移一个位置，b中，将将第5个至第7个元素从前往后依次前移一个位置。

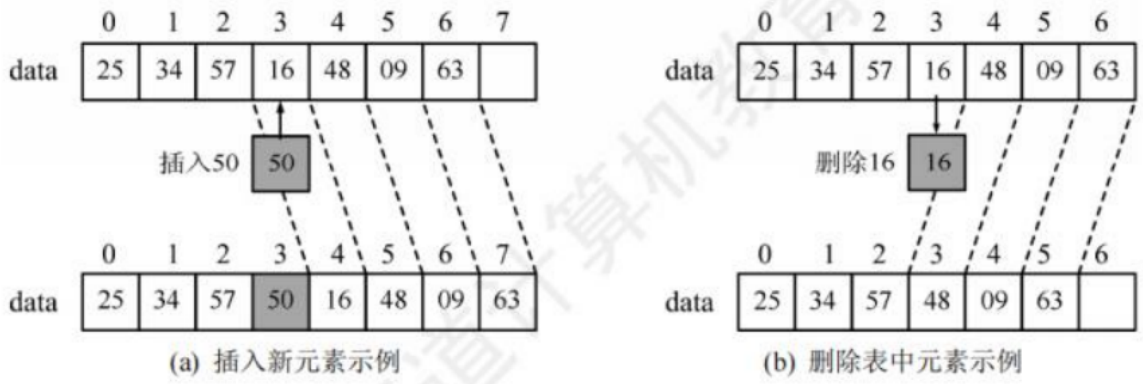

图 2.2　顺序表的插入和删除

按值查找（顺序查找）

在顺序表L中查找第一个元素值等于e的元素，并返回其位序。

```
int LocateElem(SqList L, ElemType e){
    int i;
    for(i=0;i<L.length;i++)
        if(L.data[i]==e)
            return i+1;//下标为i的元素值等于e，返回其位序i+1
    return 0;//推出循环，说明查找失败
}
```

最好情况：查找的元素就在表头，仅需比较一次，时间复杂度为O(1)。

最坏情况：查找的元素在表尾（或不存在）时，需要比较n次，时间复杂度为O(n)。

平均情况：假设pi(pi=1/n)是查找的元素在第i(1<=i<=L.length)个位置上的概率，则在长度为n的线性表中查找值为e的元素所需比较的平均次数为

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

因此，顺序表按值查找算法的平均时间复杂度为O(n)。

顺序表的按序号查找非常简单，即直接根据数组下标访问数组元素，其时间复杂度为O(1)。