# 06线性表1

## 感受线性表的存在

各位鱼油大家好，从今天开始小甲鱼带大家开始学习第一个数据结构：线性表

先让我我们通过图片来感受一下，线性表的特性吧。



大家姑且可以把这幅图想象为小甲鱼组织鱼油们外出野营。因为咱鱼油队伍强大，要小甲鱼记住每位鱼油并区分还真不是一件容易的事。

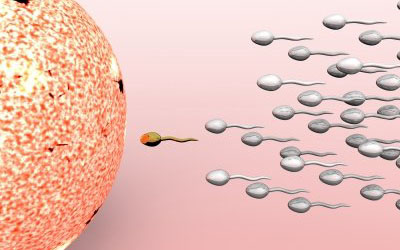
所以咱想到了一个办法，让大家按照一个约定排成一队，以后大家要怎么记住自己的位置呢？没错，大家只需要记住自己前边的鱼油即可。

小甲鱼也可以很快的清点人数，万一有人走丢，也能在最快时间内知道是谁不见了，因为只需要问哪位鱼油的“前驱”不见啦即可。

## 线性表的定义

线性表，从刚才小甲鱼的描述中我们可以很容易感受到是像排队一样，具有线一样性质的结构。

如果像下边这样，我们就不能说是线性表了：



线性表（List）：由零个（空表）或多个数据元素组成的有限序列。

这里需要强调几个关键的地方：

首先它是一个序列，也就是说元素之间是有个先来后到的，像刚才的小蝌蚪就没有顺序。

若元素存在多个，则第一个元素无前驱，而最后一个元素无后继，其他元素都有且只有一个前驱和后继。

另外，线性表强调是有限的，事实上无论计算机发展到多强大，它所处理的元素都是有限的。

如果用数学语言来进行定义，可如下：

若将线性表记为（a1,…,ai-1,ai,ai+1,…an）,则表中ai-1领先于ai,ai领先于ai+1,称ai-1是ai的直接前驱元素,ai+1是ai的直接后继元素。

所以线性表元素的个数n(n>=0)定义为线性表的长度，当n=0时，称为空表。

## 考题模拟

* 请问公司的组织架构是否属于线性关系？
* 分析：一般公司的总经理管理几个总监，每个总监管理几个经理，每个经理都有各自的下属和员工。
  + 那这样的组织架构是不是线性关系呢？
  + 当然不是啦，应该是树形结构！注意线性关系的条件是如果存在多个元素，则“第一个元素无前驱，而最后一个元素无后继，其他元素都**有且只有**一个前驱和后继。”
* 那么班级里同学之间的友谊呢？
* 当然也不是，应该是图的关系，因为每个人都会跟许多同学建立纯纯的友谊关系。
* 好，再来一题，那如果是情侣间的爱情关系呢？
* 哈，还是扯淡，这要是线性关系还哪里容得下第三者？！
* 最后一个问题，一个班级里的点名册，是不是线性表？ 嗯，这个就是了。

|  |  |  |  |
| --- | --- | --- | --- |
| **学号** | **姓名** | **性别** | **职位** |
| 1 | 黑夜 | 男 | 班长 |
| 2 | 迷途 | 男 | 副班长 |
| 3 | 喵咕 | 女 | 音乐课代表 |
| 4 | 秋舞斜阳 | 女 | 学生 |
| 5 | 小生我怕怕 | 男 | 小组长 |

## 抽象数据类型

因为下节课我们会讲线性表的抽象数据类型，所以这节课的末尾部分给大家介绍一下什么是抽象数据类型。

那在介绍抽象数据类型之前，我们应该了解数据类型的定义是什么？

数据类型：是指一组性质相同的值的集合及定义在此集合上的一些操作的总称。

例如很多编程语言的整型，浮点型，字符型这些指的就是数据类型。

当年那些设计计算机语言的人，为什么会考虑到数据类型呢？

比如，大家都需要住房子，也都希望房子越大越好。但显然，没有多少钱的话考虑房子是没啥意义的。

于是商品房就出现了各种各样的房型，有别墅的，有错层的，有单间的，甚至在北京还出现了胶囊公寓——只有两平方米的房间。

这样子就满足了大家的不同需求。

同样，在计算机中，内存也不是无限大的，你要计算入1+1=2这样的整型数字的加减乘除运算，显然不需要开辟很大的内存空间。

而如果要计算1.23456789+2.987654321这样带大量小数的，就需要开辟比较大的空间才存放的下。

于是计算机的研究者们就考虑，要对数据类型进行分类，分出多种数据类型来适合各种不同的计算条件差异。

例如在C语言中，按照取值的不同，数据类型可以分为两类：

原子类型：不可以再分解的基本类型，例如整型、浮点型、字符型等。

结构类型：由若干个类型组合而成，是可以再分解的，例如整型数组是由若干整型数据组成的。

抽象：是指抽取出事物具有的普遍性的本质。它要求抽出问题的特征而忽略非本质的细节，是对具体事物的一个概括。抽象是一种思考问题的方式，它隐藏了繁杂的细节。

我们对已有的数据类型进行抽象，就有了抽象数据类型。

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

抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关。

比如1+1=2这样一个操作，在不同CPU的处理上可能不一样，但由于其定义的数学特性相同，所以在计算机编程者看来，它们都是相同的。

“抽象”的意义在于数据类型的数学抽象特性。

而且，抽象数据类型不仅仅指那些已经定义并实现的数据类型，还可以是计算机编程者在设计软件程序时自己定义的数据类型。

例如一个3D游戏中，要定位角色的位置，那么总会出现x,y,z三个整型数据组合在一起的坐标。我们就可以定义一个point的抽象数据类型，它拥有x,y,z三个整型变量，这样我们就可以方便的对一个角色的位置进行操作。



为了便于在之后的讲解中对抽象数据类型进行规范的描述，我们给出了描述抽象数据类型的标准格式：

ADT 抽象数据类型名

Data

数据元素之间逻辑关系的定义

Operation

操作

End ADT

类似于类的定义

# 07线性表2

## 线性表的抽象数据类型

上节课我们讲到了线性表的定义，讲到了所谓抽象数据类型就是把数据类型和相关操作捆绑在一起。

那么我们接下来分析一下，线性表应该有什么样的相关操作呢？

还是回到小甲鱼组织大家春游的例子，小甲鱼把鱼油们按照规律安排成一队，并且是长期使用这样的顺序排队，大家只需要记住自己的前驱鱼油就可以了。那么这个考虑和安排的过程其实就是一个线性表的创建和初始化过程。

一开始小甲鱼没经验呀，把鱼油们按照名字第一个字母的规律排队，发现排到最后有的高有的矮，导致队伍很难看，于是让鱼油们解散重新按照从矮到高排。

上边这么描述的过程其实就是线性表重置为空表的操作过程，接着给大家描述下删除数据。

排好队后，尼玛，发现黑夜童鞋由于昨晚吃太多麻辣烫，今天闹肚子来不了春游。那么由原来排在黑夜后边的小生开始往前挪。

当然有删除数据就有插入数据，黑夜童鞋下午肚子康复了，又回来了。黑夜童鞋说记得是排在斜阳后边的，所以让从斜阳后边的小生开始往后退一个位置，黑夜就可以插回自己的位置上了。

噢，我们活动开始了不久，隔壁来了春田花花旅游团都是美眉啊，各种调戏有木有？然后小甲鱼被她们的导游投诉了有木有？！导游说你们队伍里第八个那个家伙老是调戏我们队里的小花，他叫什么名字？小甲鱼查了下名单说，噢，他叫迷途。

好吧，这就是根据位序得到数据元素的例子。

好了，剩下的还有很多大家可以自己想象。

我们给大家总结下线性表的抽象数据类型定义：

ADT 线性表（List）

Data

线性表的数据对象集合为{a1,a2,…,an}，每个元素的类型均为DataType。其中，除第一个元素a1外，每一个元素有且只有一个直接前驱元素，除了最后一个元素an外，每一个元素有且只有一个直接后继元素。数据元素之间的关系是一对一的关系。

Operation

InitList(\*L): 初始化操作，建立一个空的线性表L。

ListEmpty(L): 判断线性表是否为空表，若线性表为空，返回true，否则返回false。

ClearList(\*L): 将线性表清空。

GetElem(L,i,\*e): 将线性表L中的第i个位置元素值返回给e。

LocateElem(L,e): 在线性表L中查找与给定值e相等的元素，如果查找成功，返回该元素在表中序号表示成功；否则，返回0表示失败。

ListInsert(\*L,i,e): 在线性表L中第i个位置插入新元素e。

ListDelete(\*L,i,\*e): 删除线性表L中第i个位置元素，并用e返回其值。

ListLength(L): 返回线性表L的元素个数。

endADT

对于不同的应用，线性表的基本操作是不同的，上述操作是最基本的，对于实际问题中涉及的关于线性表的更复杂操作，完全可以用这些基本操作的组合来实现。

举个例子？好吧，比如要实现两个线性表A、B的并集操作，即要使得集合A=A∪B。

解释下“A=A∪B”：说白了，就是把存在集合B中但不存在集合A中的元素插入到A中即可。

其实仔细思考一下，我们只需要循环遍历集合B中的每个元素，判断当前元素是否存在A中，若不存在，则插入A中即可。

综合分析，我们需要运用到几个基本的操作组合即可：

ListLength(L);

GetElem(L,i,\*e);

LocateElem(L,e);

ListInsert(\*L,i,e);

参考实现代码段：union.c

// La表示A集合，Lb表示B集合。

void unionL(List \*La, list Lb)

{

int La\_len, Lb\_len, i;

ElemType e;

La\_len = ListLength(\*La);

Lb\_len = ListLength(Lb);

for( i=1; i <= Lb\_len; i++ ) ——伪代码

{

GetElem(Lb, i, &e);

if( !LocateElem(\*La, e) )

{

ListInsert(La, ++La\_len, e);

}

}

}

# 08线性表3

## 线性表的顺序存储结构

我们可以想象，线性表有两种物理存储结构：顺序存储结构和链式存储结构。

线性表的顺序存储结构，指的是用一段地址连续的存储单元**依次**存储线性表的数据元素。

线性表（a1,a2,…,an）的顺序存储如下：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **a1** | **a2** | **a3** | **a4** | **…** | **ai-1** | **ai** | **ai+1** | **…** | **an** |

上边的图片大家想到了什么？

是不是跟丫的数组一样一样的？！

物理上的存储方式事实上就是在内存中找个初始地址，然后通过占位的形式，把一定的内存空间给占了，然后把相同数据类型的数据元素依次放在这块空地中。

大家可能喜欢听小甲鱼拿现实的栗子来举例。

哈，就例如小甲鱼带着咱的鱼C旅游团去看电影，看那个叫3D\*蒲团有木有？！哎哟，去到电影院人山人海，但妹子的身影还是难以逃过小甲鱼的火眼精金，所以小甲鱼猛地带领众鱼油一把夺下了妹子旁边的一排位置。。。。。。

话说小甲鱼拿下第一个位置后，鱼油们果断依次坐下，这样子我们就建立了一个线性表。

由此可见，这小甲鱼夺下的第一个位置是非常关键的，如果妹子身旁的位置被其他人占据了，小甲鱼就不能代替大家问到妹子的联系方式了！

可现实中总会有些不按部就班来的，例如黑夜喜欢乱吃零食，所以他又拉肚子了。。。但是黑夜买票了，所以他的位置就算空着，也没有人会坐上去。这就是顺序存储结构的特性之一。

接下来看线性表顺序存储的结构代码：

#define MAXSIZE 20

typedef int ElemType;

typedef struct

{

ElemType data[MAXSIZE];

int length; // 线性表当前长度

} SqList;

大家看到了，这里我们封装了一个结构，事实上就是对数组进行封装，增加了个当前长度的变量罢了。

总结下，顺序存储结构封装需要三个属性：

存储空间的起始位置，数组data，它的存储位置就是线性表存储空间的存储位置。

线性表的最大存储容量：数组的长度MaxSize。

线性表的当前长度：length。

注意，数组的长度与线性表的当前长度需要区分一下：数组的长度是存放线性表的存储空间的总长度，一般初始化后不变（动态改变的话会导致性能有所下降）。而线性表的当前长度是线性表中元素的个数，是会变化的。

## 地址计算方法

我们或许习惯了数组的从0开始的计算方法，所以导致小甲鱼像去电影院坐错位置被妹子怒骂无耻等尴尬事件不胜枚举。

线性表的定义充分考虑到很多军师级别领导的智商指数，所以决定从1开始回归正常思维。

假设ElemType占用的是c个存储单元（字节），那么线性表中第i+1个数据元素和第i个数据元素的存储位置的关系是（LOC表示获得存储位置的函数）：LOC(ai+1) = LOC(ai) + c

所以对于第i个数据元素ai的存储位置可以由a1推算得出：LOC(ai) = LOC(a1) + (i-1)\*c

结合下图来理解：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **元素** | **a1** | **a2** | **…** | **ai-1** | **ai** | **…** | **an** | **空闲空间** |
| 下标 | 0 | 1 | … | i-2 | i-1 | … | n-1 |  |

通过这个公式，我们可以随时计算出线性表中任意位置的地址，不管它是第一个还是最后一个，都是相同的时间。那么它的存储时间性能当然就为0(1)，我们通常称为随机存储结构。

## 获得元素操作

实现GetElem的具体操作，即将线性表L中的第i个位置元素值返回。就程序而言非常简单了，我们只需要把数组第i-1下标的值返回即可。

我们来学习下具体的代码：

getElem.c

#define OK 1

#define ERROR 0

#define TRUE 1

#define FALSE 0

typedef int Status;

// Status 是函数的类型，其值是函数结果状态代码，如OK等。

// 初始条件：顺序线性表L已存在，1 <= i <= ListLength(L)

// 操作结果：用e返回L中第i个数据元素的值。

Status GetElem(SqList L, int i, ElemType \*e)

{

if( L.length==0 || i<1 || i>L.length ) ——伪代码

{

return ERROR;

}

\*e = L.data[i-1];

return OK;

}

注意这里返回值类型Status是一个整型，约定返回1代表OK，返回0代表ERROR。今后还会出现，也是使用同样的约定，不再详述。

## 插入操作

刚才我们也谈到，线性表的顺序存储结构具有随机存储结构的特点，时间复杂度为O(1)。

大家现在来考虑下，如果我们要实现ListInsert(\*L, i, e)，即在线性表L中的第i个位置插入新元素e，代码应该如何写？

例子接着举，刚才小甲鱼在电影院成功的博得妹子的欢心，妹子表示愿意加入我们鱼C旅游团。大家狠兴奋丫有木有！因为大家都希望小甲鱼把妹子排到自己身边。但是我们要按我们定下的规则来排：按身高顺序。

所以插入算法的思路：

* + 如果插入位置不合理，抛出异常；
  + 如果线性表长度大于等于数组长度，则抛出异常或动态增加数组容量；
  + 从最后一个元素开始向前遍历到第i个位置，分别将它们都向后移动一个位置；
  + 将要插入元素填入位置i处；
  + 线性表长+1。

实现代码：

<ListInsert.c>

/\* 初始条件：顺序线性表L已存在，1<=i<=ListLength(L)。 \*/

/\* 操作结果：在L中第i个位置之前插入新的数据元素e，L长度+1。\*/

Status ListInsert(SqList \*L, int i, ElemType e)

{

int k;

if( L->length == MAXSIZE ) // 顺序线性表已经满了

{

return ERROR;

}

if( i<1 || i>L->length+1) // 当i不在范围内时

{

return ERROR;

}

if( i <= L->length ) // 若插入数据位置不在表尾

{

/\* 将要插入位置后数据元素向后移动一位 \*/

for( k=L->length-1; k >= i-1; k-- )

{

L->data[k+1] = L->data[k];

}

}

L->data[i-1] = e; // 将新元素插入

L->length++;

return OK;

}

# 09线性表4

## 删除操作

* 所以删除算法的思路：
  + 如果删除位置不合理，抛出异常；
  + 取出删除元素；
  + 从删除元素位置开始遍历到最后一个元素位置，分别将它们都向前移动一个位置；
  + 表长-1。

实现代码：ListDelete.c

/\* 初始条件：顺序线性表L已存在，1<=i<=ListLength(L) \*/

/\* 操作结果：删除L的第i个数据元素，并用e返回其值，L的长度-1 \*/

Status ListDelete(SqList \*L, int i, ElemType \*e)

{

int k;

if( L->length == 0 )

{

return ERROR;

}

if( i<1 || i>L->length )

{

return ERROR;

}

\*e = L->data[i-1];

if( i < L->length )

{

for( k=i; k < L->length; k++ )

{

L->data[k-1] = L->data[k];

}

}

L->length--;

return OK;

}

现在我们分析一下，插入和删除的时间复杂度。

最好的情况：插入和删除操作刚好要求在最后一个位置操作，因为不需要移动任何元素，所以此时的时间复杂度为O(1)。

最坏的情况：如果要插入和删除的位置是第一个元素，那就意味着要移动所有的元素向后或者向前，所以这个时间复杂度为O(n)。

至于平均情况，就取中间值O((n-1)/2)。

按照前边游戏秘籍指导，平均情况复杂度简化后还是O(n)。

线性表顺序存储结构的优缺点

线性表的顺序存储结构，在存、读数据时，不管是哪个位置，时间复杂度都是O(1)。而在插入或删除时，时间复杂度都是O(n)。

这就说明，它比较适合元素个数比较稳定，不经常插入和删除元素，而更多的操作是存取数据的应用。

那我们接下来给大家简单总结下线性表的顺序存储结构的优缺点：

优点：

* + 无须为表示表中元素之间的逻辑关系而增加额外的存储空间。
  + 可以快速地存取表中任意位置的元素。

缺点：

* + 插入和删除操作需要移动大量元素。
  + 当线性表长度变化较大时，难以确定存储空间的容量。
  + 容易造成存储空间的“碎片”。

## 线性表的链式存储结构

前面我们讲的线性表的顺序存储结构，它最大的缺点就是插入和删除时需要移动大量元素，这显然就需要耗费时间。

那我们能不能针对这个缺陷或者说遗憾提出解决的方法呢？要解决这个问题，我们就得考虑一下导致这个问题的原因！

为什么当插入和删除时，就要移动大量的元素？

原因就在于相邻两元素的存储位置也具有邻居关系，它们在内存中的位置是紧挨着的，中间没有间隙，当然就无法快速插入和删除。

经过叽叽呱呱的讨论之后，我们派出几个童鞋跟大家分享一下思路。

A童鞋：让当中每个元素之间都留有一个空位置，这样要插入一个元素时，就不至于要移动了。可一个空位置如何解决多个相同位置插入数据的问题呢？所以这个想法显然不行。

B童鞋：那就让当中每个元素之间都留足够多的位置，根据实际情况制定空隙大小，比如每个元素间留10个空位。那么问题就显而易见了，造成资源的极大浪费，并且在同一个位置插入11次也不是不可能。

C童鞋：反正要在相邻元素间留多少空间都是有可能不够的，那不如干脆不要考虑相邻位置这个问题了。哪里有空位就放在哪里，记得在小甲鱼老湿的《零基础入门学习C语言》中讲到的指针刚好可以派上用场。每个元素多用一个位置来存放指向下一个元素的位置的指针。这样子从第一个元素可以找到第二个元素，第二个元素可以找到第三个元素，依此类推，所有的元素我们就都可以通过遍历而找到了。

好，太棒了，这个想法灰常好！掌声鼓励！

## 线性表链式存储结构定义

哎呀，又要讲定义的时候了，往往啊，这个时候就是搬出概念砸脚的时候了。我们不能这样丫，我们的课堂应该是欢乐与傻笑并存，智慧与邪恶同在。

好，那我们从爱情动作片开始吧~

这个无论是爱情片还是动作片，我们都感觉它没有悬疑片那么吸引人。。。。。。

而悬疑片正是像链式存储那样一环套这一环，时而让你紧张，时而让你鸡冻！

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素，这组存储单元可以存在内存中未被占用的任意位置。

比起顺序存储结构每个数据元素只需要存储一个位置就可以了。现在链式存储结构中，除了要存储数据元素信息外，还要存储它的后继元素的存储地址（指针）。

也就是说除了存储其本身的信息外，还需存储一个指示其直接后继的存储位置的信息。

我们把存储数据元素信息的域称为数据域，把存储直接后继位置的域称为指针域。指针域中存储的信息称为指针或链。这两部分信息组成数据元素称为存储映像，称为结点(Node)。

n个结点链接成一个链表，即为线性表(a1, a2, a3, …, an)的链式存储结构。

因为此链表的每个结点中只包含一个指针域，所以叫做单链表。

No pic you say a J8…

单链表

* 对于线性表来说，总得有个头有个尾，链表也不例外。我们把链表中的第一个结点的存储位置叫做头指针，最后一个结点指针为空(NULL)。

10线性表5

头指针与头结点的异同

* 上节课我们提到了，头结点的数据域一般不存储任何信息，谁叫它是第一个呢，有这个特权。
* 拿个小旗子即可：



* 那有童鞋就疑惑了，既然头结点的数据域不存储任何信息，那么头指针和头结点又有何异同呢？
* 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，所以常用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。
* 头结点
  + 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须要素。

No pic you say a J8…

单链表存储结构

* 单链表图例：



* 空链表图例：



## 单链表存储结构

我们在C语言中可以用结构指针来描述单链表。

typedef struct Node

{

ElemType data; // 数据域

struct Node \*Next; // 指针域

} Node;

typedef struct Node \*LinkList; //取别名

我们看到结点由存放数据元素的数据域和存放后继结点地址的指针域组成。

假设p是指向线性表第i个元素的指针，则该结点ai的数据域我们可以用p->data的值是一个数据元素，结点ai的指针域可以用p->next来表示，p->next的值是一个指针。

那么p->next指向谁呢？当然指向第i+1个元素！也就是指向ai+1的指针。

问题：

如果p->data = ai，那么p->next->data = ?

答案：p->next->data = ai+1。

## 单链表的读取

在线性表的顺序存储结构中，我们要计算任意一个元素的存储位置是很容易的。

但在单链表中，由于第i个元素到底在哪？我们压根儿没办法一开始就知道，必须得从第一个结点开始挨个儿找。

因此，对于单链表实现获取第i个元素的数据的操作GetElem，在算法上相对要麻烦一些，大家不妨先思考一下。

获得链表第i个数据的算法思路：

声明一个结点p指向链表第一个结点，初始化j从1开始；

当j<i时，就遍历链表，让p的指针向后移动，不断指向一下结点，j+1；

若到链表末尾p为空，则说明第i个元素不存在；

否则查找成功，返回结点p的数据。

有了以上的思路提示，小甲鱼邀请大家再度进行头脑风暴：算法的C语言实现代码，

GetElem.c

/\* 初始条件：顺序线性表L已存在，1<=i<=ListLength(L) \*/

/\* 操作结果：用e返回L中第i个数据元素的值 \*/

Status GetElem( LinkList L, int i, ElemType \*e )

{

int j;

LinkList p;

p = L->next;

j = 1;

while( p && j<i )

{

p = p->next;

++j;

}

if( !p || j>i )

{

return ERROR;

}

\*e = p->data;

return OK;

}

说白了，就是从头开始找，直到第i个元素为止。

由于这个算法的时间复杂度取决于i的位置，当i=1时，则不需要遍历，而i=n时则遍历n-1次才可以。因此最坏情况的时间复杂度为O(n)。

由于单链表的结构中没有定义表长，所以不能实现知道要循环多少次，因此也就不方便使用for来控制循环。

其核心思想叫做“工作指针后移”，这其实也是很多算法的常用技术。