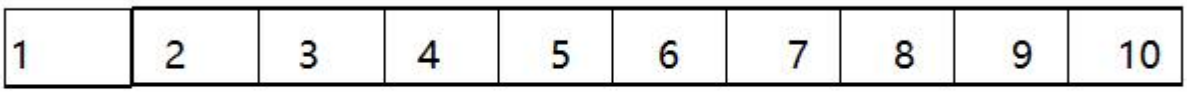
顺序表

> 顺序表的各项操作我采用静态的方式讲解，因为我觉得如果需要支持动态扩容的线性结构，那么链表会是一个更好的选择，所以我通常使用顺序表结构只使用静态的就足够了。

概述

顺序表可以分为静态顺序表和动态顺序表两种，静态顺序表不可以扩容，一旦创建，它的容量就写死了，是多大就是多大，而动态顺序表则可以扩容。通常情况下，动态顺序表使用价值比较好一些，但是这也不一定，因为有时候我们确认了某些结构的空间不需要改变，那么用静态顺序表就合适一些。

顺序表在内存中的存储方式是连续的：



如上图所示，顺序表中的每个元素都是相邻，由于它是连续存储在一大块空间上，所以就有一个显而易见的好处：

支持随机存取。

什么是随机存取呢？

我用数组来举例子，一个数组：`int arr[10]`,随机存取就是我们可以通过下标的方式直接读写某一个元素。比方说，我们想要读数组第三个元素的值，那么直接`arr[2]`就可以了。这个特点后面讲链表的时候我会做一个对比。

定义节点

首先明确，这个节点类型内都封装了哪些数据：

- 需要存储的元素
- 已存储元素的个数

第一个是毫无疑问需要包含的，原因不赘述；第二个是后面遍历顺序表的时候，需要有一个循环的上限，这个上限就是当前顺序表有效元素的个数。

```
#define SIZE 10 //定义顺序表的容量，方便以后更改 (1)
typedef int DataType; //将顺序表中元素类型进行逻辑抽象 (2)

typedef struct SeqList{
    DataType _table[SIZE]; //这里我用数组实现顺序表的底层结构
    int _size; //顺序表的有效元素个数
}SeqList;
```

我解释一下上面代码：

（1）：为什么这里要这么做呢？

这是因为以后如果想要更改顺序表的容量，直接修改这里定义的`SIZE`值就可以了

（2）：这个是不是感觉看的迷迷糊糊的？

`typedef`的作用是将一个数据类型重命名为另一个名称，这里我就是给`int`类型重起了个名字叫`DataType`。

那么当前的顺序表中的元素就是`DataType`类型的，`DataType`实际上是`int`类型的，那后面我要是想让顺序表存储`char`类型的元素，就不再需要重新定义节点，我只需要将`DataType`修改为`char`的重命名即可：

```
typedef char DataType;
```

这样就提高了程序的**适用性**。

顺序表初始化

将顺序表中的元素都初始化为0即可。

```
void InitSeqList(SeqList* Seq){
    assert(pSeq); //检测参数的合法性
    //遍历顺序表
    for (Seq->_size = 0; Seq->_size < SIZE, Seq->_size++){
        Seq->_table[Seq->_size] = 0;
    }
    Seq->_size = 0;//将顺序表有效元素个数置零
}
```

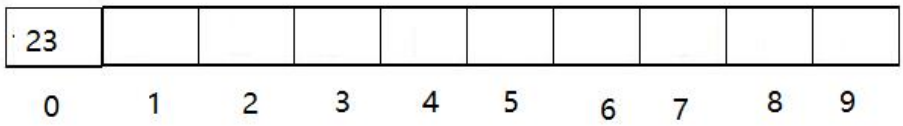
元素添加

如果顺序表未满，则将指定元素添加到顺序表的末尾，然后顺序表的容量+1即可。

```
void AddSeqList(SeqList* Seq, DataType elem){
    assert(Seq);
    if(Seq->_size < SIZE){
        //由于顺序表的_size是当前有效元素个数，而数组是以0为下标开始存储，当数组中有一个元素时
        //有效元素的个数为1，下一次存储的时候，下标就是1。所以每一次存储新元素时，_size的值刚好
        //就是存储位置的下标。
        Seq->_table[Seq->_size] = elem;
    }
    Seq->_size++;
}
```

还不清楚的可以看下图：

当前顺序表存储了一个元素23，下一次存储的时候需要存储到下标为1的位置，而_size的值刚好就是1



_size = 1

元素删除

删除某个元素的步骤：

- 在顺序表中找到这个元素，如果没找到，则不进行下面步骤
- 将这个元素后面的元素移动到这个元素的位置，然后重复进行这个过程，直到顺序表末尾

我画图说明一下这个步骤：


```
void DelSeqList(SeqList* Seq, DataType elem){
    assert(Seq);
    int pos = 0;
    int i = 0;
    //遍历顺序表，在顺序表中查找要删除的元素的位置
    for(i = 0; i < SeqList->_size; ++i){
        if(Seq->_table[i] == elem){
            pos = i; //记录要删除元素的位置
            break;
        }
    }
    //如果在顺序表中找到了要删除的元素（默认顺序表中所有元素不重复）
    if(i != SeqList->_size){
        for(int j = pos; j < SeqList->_size; ++j){
            SeqList->_table[j] = SeqList->_table[j + 1];
        }
        Seq->_size--;
    }
}
```

元素排序

> 排序函数调用了 SelectSort 函数，即底层采用了选择排序算法对顺序表中的元素进行排序，那么问题来了：
>
> 为什么我不直接将SortSeqList直接写成选择排序，而是单独写了一个选择排序函数，然后在SortSeqList函数中调用选择排序函数？（为了让读者先思考思考，答案文末揭晓）

```
void swap(int *x, int *y)
{
    *x = *x ^ *y;
    *y = *x ^ *y;
```

```
    *x = *x ^ *y;
}
//升序选择排序算法
void SelectSort(int array[], int size){
    for (int i = 0; i < size; ++i){
        for (int j = i + 1; j < size; ++j){
            if (array[j] < array[i]){
                swap(&array[i], &array[j]);
            }
        }//end for(j)
    }//end for(i)
}//end func

void SortSeqList(SeqList* Seq){
    assert(Seq);
    SelectSort(Seq->_table, Seq->_size);
}
```

清空顺序表

```
void ClearSeqList(SeqList* Seq){
    assert(Seq);
    Seq->_size = 0; //将顺序表有效元素个数置0，就相当清空了顺序表
}
```

> 答案揭晓：
>
> 之所以不直接将排序函数写成具体的排序算法，而是在**顺序表排序函数**中调用**选择排序函数**，是为了实现 **解耦和**（敲重点！）。
>
> 什么是解耦和呢？
>
> 这里就不引用百度的答案了，太书面化。一个软件必定要由多个模块构成，解耦和的意思就是尽量降低这些模块之间的相互依赖，避免在以后想要修改某一个模块而导致牵一发而动全身。
>
> 这里就是一个例子，如果以后软件开发者发现了一种更高效的排序算法，那么保持接口（即函数的返回值和参数的含义）不变，他就可以直接将SelectSort替换成另一种排序函数名即可，而不用大刀阔斧的修改顺序表的代码。
>
> 这在软件开发过程中是一个非常重要的方法，读者一定要细细品味~