



中国地质大学 计算机学院  
China University of Geosciences



# 数据结构

## 第二章 线性表[1] 顺序表

任课老师：郭艳

数据结构课程组

计算机学院

中国地质大学（武汉）2020年秋

# 上节课要点回顾

- 数据结构 = 数据的逻辑结构 + 数据的存储结构 + 数据的运算
- 程序 = 算法 + 数据结构
- **P5**: 数据结构的核心技术是分解与抽象。
- **P8**: 抽象数据类型的特征是使用与实现分离, 实行封装和信息隐蔽。  
抽象数据类型是大型软件构造的模块, 每个抽象数据类型的实现放在头文件中。
- **P10**: 类的定义体现了抽象数据类型的思想。
- 算法的时间效率分析——时间复杂度  
$$T(n)=O(f(n))$$

# 第二次课

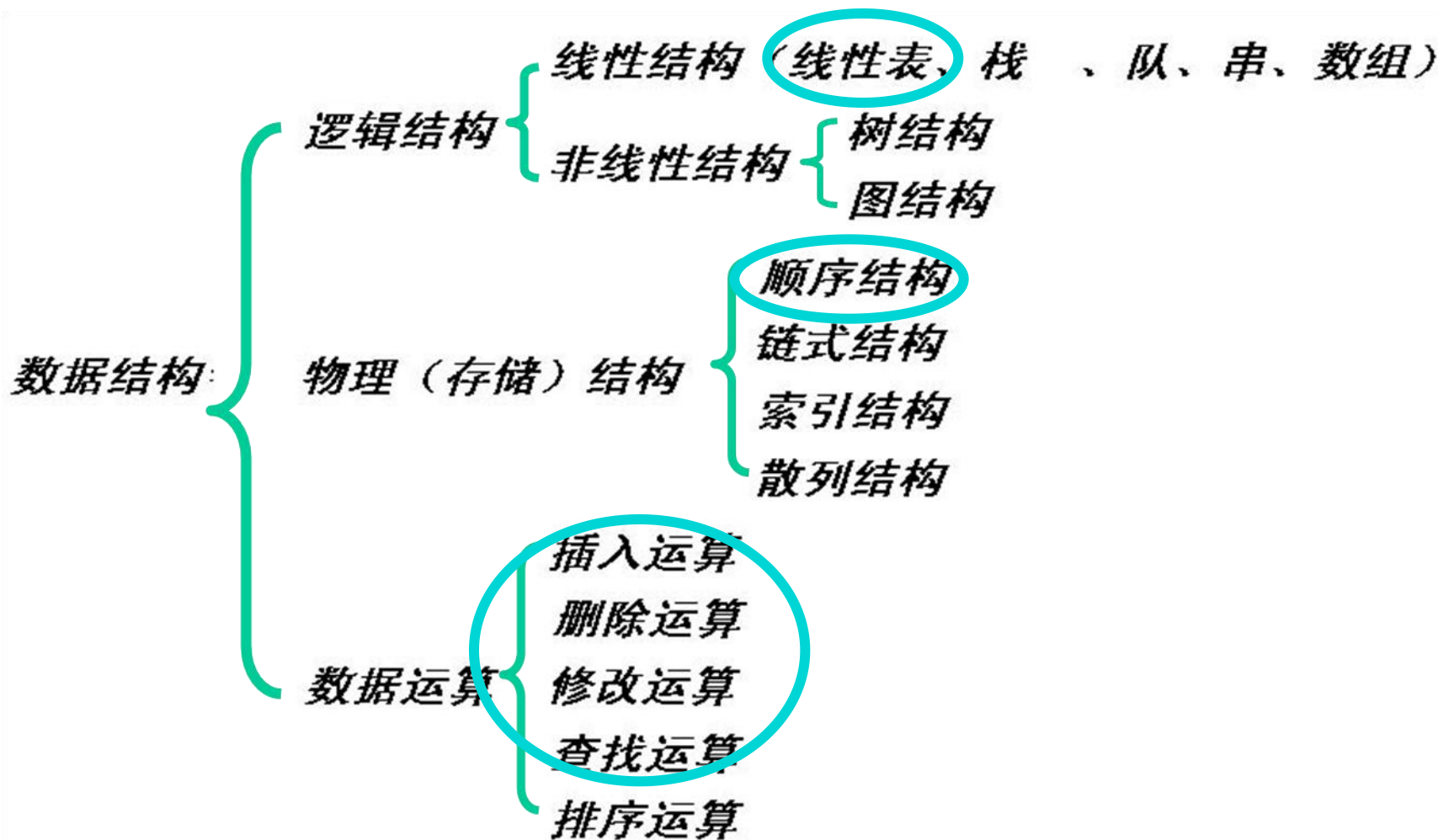
阅读：

殷人坤，第**43-52**页

练习：

作业**2**

# 数据结构课程内容



# Chapter2 线性表

2.1 线性表抽象数据类型

2.2 线性表的顺序表示和实现——顺序表

2.3 线性表的链式表示和实现——单链表

2.4 线性链表的其它变形

2.5 单链表的应用：多项式及其应用（自学）

2.6 静态链表（自学）

## 2.1 线性表抽象数据类型

### ● 什么是线性表?

例1.  $\langle 0, 1, 2, 3, \dots, 9 \rangle$

例2.  $\langle A, B, C, \dots, Z \rangle$

例3. 学生操行等级表

姓名	学号	性别	年龄	操行等级
张三	00101	女	19	优
李四	00102	男	20	良
王五	00103	男	20	优
⋮	⋮	⋮	⋮	⋮

**定义:** 一个线性表是 $n(n \geq 0)$  个相同类型数据元素的有限序列, 可以在该结构中任意位置插入和删除数据元素。

其中,  $n$ 为线性表的长度,

$n=0$ , 为空表;

$n>0$ , 非空表, 表示为

$a_i, 1 \leq i \leq n$  ( 其中 $a_i$ 是抽象的数据元素 )

或  $L = \langle a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n \rangle$

表头

数据元素

$a_i$ 的直接前趋

$a_i$ 的直接后继

表尾

下标, 是元素的序号,  
表示元素在表中的位置

$n$ 为元素总个数,  
即表长



# 线性表的抽象数据类型

## ADT LinearList{

Objects:  $n(n \geq 0)$ 个原子表项的一个有限序列。每个表项的数据类型为**T**。

Function:

Create( )

创建一个空线性表

int Length( )

求表长函数

int Search(T& x )

搜索函数，返回x表项位置

int Locate(int i)

定位函数，返回第i个表项位置

bool getData(int i, T& x)

取第i个表项的值，通过x返回，函数返回成功标志

bool setData(int i, T& x)

用x修改第i个表项内容，函数返回成功标志

bool Insert(int i, T& x)

插入x在表中第i个表项之**后**，函数返回成功标志

bool Remove(int i, T& x)

删除表中第i个表项，通过x返回被删表项值，  
函数返回成功标志

bool IsEmpty( )

判断表空否

bool IsFull( )

判断表满否

void CopyList(List<T> & L)

将表L复制到当前的表中

void Sort( )

对当前的表排序，将线性表转换为有序表

}//LinearList



# 线性表的抽象基类 (P44程序2. 2)



// “LinearList .h” 定义线性表类

```
template <class T> //使用模板实现不依赖于数据元素类型
class LinearList {
public: //其它类实例可以请求该类的实例执行共有函数
    LinearList();           //构造函数，对象建立时自动调用
    ~LinearList();          //析构函数，撤销对象时自动调用
    //纯虚函数在派生类中必须被重定义，抽象类不能生成实例
    //关键字const指明函数不可以修改实际引用参数的值或返回值
    virtual int Size() const = 0;           //获取最大表项个数...
    virtual int Length() const = 0;         //求表实际长度
    virtual int Search(T& x) const = 0;      //搜索
    virtual int Locate(int i) const = 0;     //定位
    virtual bool getData(int i, T& x) const = 0; //取值
    virtual void setData(int i, T& x) = 0;   //赋值
};
```



```
virtual bool Insert(int i, T& x) = 0;           //插入
virtual bool Remove(int i, T& x) = 0;          //删除
virtual bool IsEmpty() const = 0;              //判表空
virtual bool IsFull() const = 0;               //判表满
virtual void Sort() = 0;                       //排序
virtual void input() = 0;                      //输入...
virtual void output() = 0;                    //输出...
/*virtual LinearList<T>* operator=(LinearList<T>&
L)=0;*/
};
```

线性表的存储表示有两种：顺序存储方式和链表存储方式。

## 2.2 顺序表

- 顺序存储结构

- ◆ 定义：用一组地址连续<sup>连续</sup>的存储单元依次存放线性表中的各个数据元素。
- ◆ 特点：逻辑结构上相邻的元素其物理位置也相邻。

- 顺序表 (Sequential List)

用顺序存储结构存储的线性表简称为顺序表。

存储地址      内 存      元素序号

b	a <sub>1</sub>	0
b+1	a <sub>2</sub>	1
b+2	a <sub>3</sub>	2
⋮	⋮	⋮
b+i	a <sub>i</sub>	i
⋮	⋮	⋮
b+(n-1)	a <sub>n</sub>	n-1

若每个数据元素  
只占一个存储单元，  
则元素a<sub>i</sub>的地址为：  
 $Loc(a_i) = Loc(a_1) + i - 1$

数组的存储结构图示

若每个数据元素占  $l = \text{sizeof}(\mathbf{T})$  个存储单元，则有：

$$Loc(a_i) = Loc(a_1) + (i-1) * l \quad (1 \leq i \leq n)$$

因此，顺序表是一种随机存取的存储结构。



# 顺序表模板类 (SeqList<T>) 的定义 (P46程序2.5)

```
#include <iostream>           //“SeqList.h”实现顺序表类
#include <cstdlib>
#include <cassert>
using namespace std;
#include "LinearList.h"
const int defaultSize = 100;
template <class T> class SeqList: public LinearList<T> {
protected:    //仅允许该类的成员函数或友元函数存取或调用
    T *data;    //地址连续的数据空间，存放数组
    int maxSize;    //最大可容纳表项的项数
    int last;    //当前已存表项的最后位置(从0开始)
    void reSize(int newSize);    //改变data数组空间大小
```



public:

**SeqList(int sz = defaultSize);**

//构造函数

**SeqList(SeqList<T>& L);**

//复制构造函数

**~SeqList() {delete [ ] data;}**

//析构函数

**int Size() const {return maxSize;}**

//求表最大容量

**int Length() const {return last+1;}**

//计算表长度

**int Search(T& x) const;**

//搜索x在表中位置，函数返回表项序号

**int Locate(int i) const;**

//定位第 i 个表项，函数返回表项序号

**bool Insert(int i, T& x);**

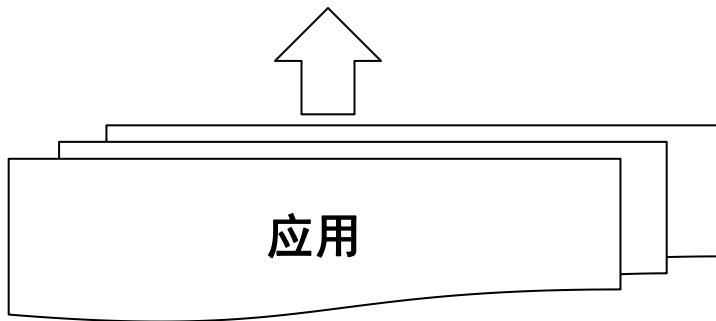
//插入x在第i个表项之后

**bool Remove(int i, T& x);**

//删除第i个表项



```
bool getData(int i, T &x) const; //取第i个表项的值
void setData(int i, T &x);      //用x修改第i个表项的值
bool IsEmpty()const
    //判断表空否，空返回true，否则返回false
bool IsFull()const
    //判断表满否，满返回true，否则返回false
void Sort(); //排序
void input();    //输入
void output();   //输出
//SeqList<T>* operator=(SeqList<T> &L); //表整体赋值，略
};
```





# 【顺序表的构造函数与复制构造函数】



P47 程序2.6(1, 2)

```
template <class T>
```

```
SeqList<T>::SeqList(int sz) {
```

```
    if (sz > 0) { maxSize = sz;          last = -1;
```

```
        data = new T[maxSize]; //动态创建顺序表存储数组
```

```
        if (data == NULL) //动态分配失败
```

```
            { cerr << "存储分配错误！" << endl; exit(1); } }
```

```
}
```

```
template <class T>
```

```
SeqList<T>::SeqList ( SeqList<T>& L ) { T value;
```

```
    maxSize = L.Size(); last = L.Length()-1;
```

```
    data = new T[maxSize]; //动态创建存储数组
```

```
    if (data == NULL) //动态分配失败
```

```
        {cerr << "存储分配错误！" << endl; exit(1);}
```

```
    for (int i = 1; i <= last+1; i++) //传送各个表项
```

```
        { L.getData(i,value); data[i-1]=value; }
```

```
}//时间复杂度？
```

# SeqListTest.cpp文件实现顺序表类测试

```
typedef int DataType;           //定义具体的数据元素类型
```

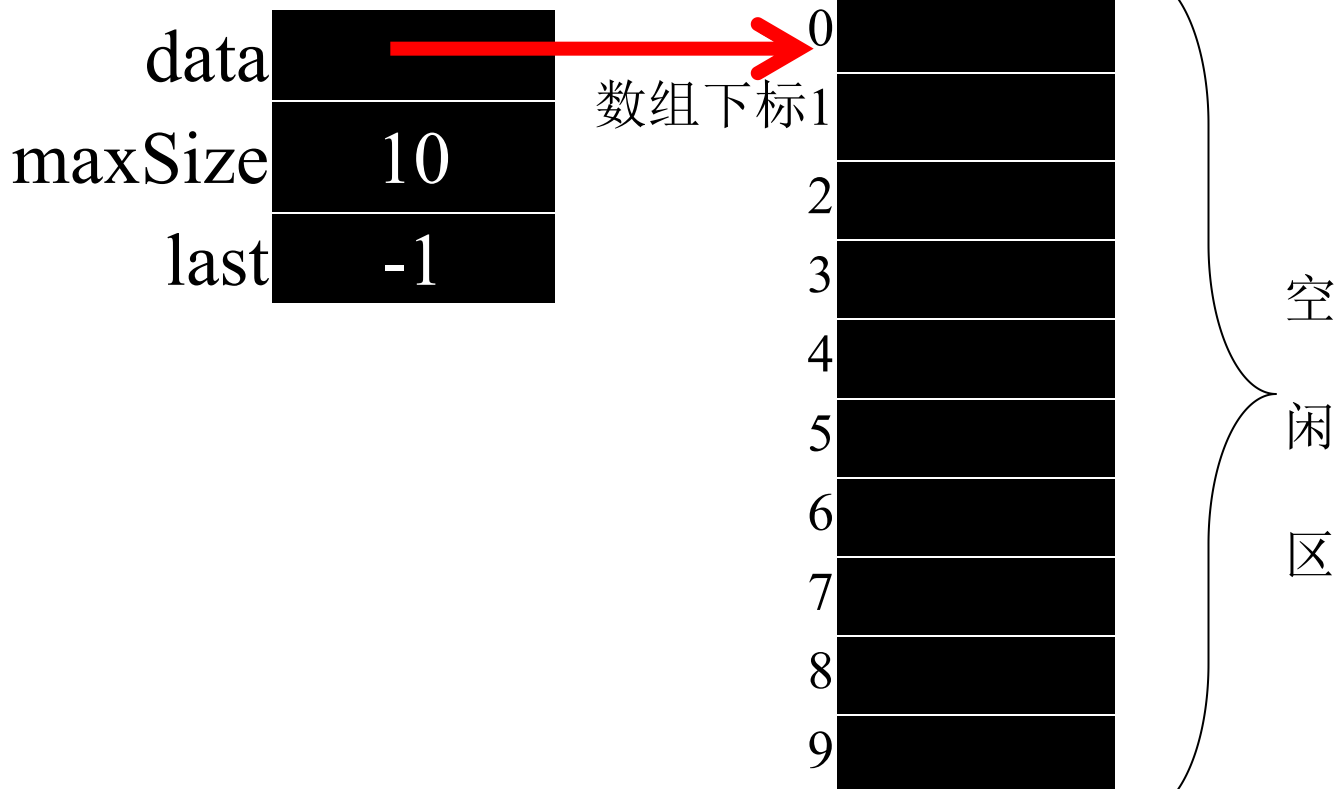
```
#include "SeqList.h"
```

```
main( )
```

```
{ SeqList<DataType> myList(10);/*创建一最大表长为10的顺序表，运  
行SeqList构造函数，动态创建一地址连续空间*/
```

```
.....
```

```
}  
/*调用myList的  
析构函数*/
```



名称	值	类型
myList	{data=0x00a1cc90 {-842150451} maxSize=100 last=-1 }	SeqList<int>
LinearList<int>	{...}	LinearList<int>
_vfptr	0x00deecb8 {Hw2-2017.exe!const SeqList<int>::vftable'} {0x00de110e {Hv	void **
[0]	0x00de110e {Hw2-2017.exe!SeqList<int>::Size(void)const }	void *
[1]	0x00de1415 {Hw2-2017.exe!SeqList<int>::Length(void)const }	void *
[2]	0x00de1014 {Hw2-2017.exe!SeqList<int>::Search(int &)const }	void *
[3]	0x00de11b3 {Hw2-2017.exe!SeqList<int>::Locate(int)const }	void *
[4]	0x00de14ec {Hw2-2017.exe!SeqList<int>::getData(int,int &)const }	void *
[5]	0x00de141f {Hw2-2017.exe!SeqList<int>::setData(int,int &)}	void *
[6]	0x00de125d {Hw2-2017.exe!SeqList<int>::Insert(int,int &)}	void *
[7]	0x00de1280 {Hw2-2017.exe!SeqList<int>::Remove(int,int &)}	void *
[8]	0x00de1249 {Hw2-2017.exe!SeqList<int>::IsEmpty(void)const }	void *
[9]	0x00de13bb {Hw2-2017.exe!SeqList<int>::IsFull(void)const }	void *
[10]	0x00de106e {Hw2-2017.exe!SeqList<int>::Sort(void)}	void *
[11]	0x00de151e {Hw2-2017.exe!SeqList<int>::input(void)}	void *
[12]	0x00de10cd {Hw2-2017.exe!SeqList<int>::output(void)}	void *
data	0x00a1cc90 {-842150451}	int *
	-842150451	int
maxSize	100	int
last	-1	int

## 【获取和设置数据元素】 P47 程序2.5中 自学

```
template <class T>
bool SeqList<T>::getData(int i, T &x) const
//取第i个表项的值 (1<=i<=表长)
{
    if (i > 0 && i <= last + 1) {
        x = data[i - 1];
        return true;
    }
    else return false;
} //时间复杂度?
```

```
template <class T>
void SeqList<T>::setData(int i, T &x)
//设置第i个表项的值
{
    if (i > 0 && i <= last + 1) data[i - 1] = x;
}
```

## 【判断表空/满否、定位函数（自学）】 P47 程序2.5中

```
template<class T>bool SeqList<T>::IsEmpty() const  
//判断表空否，空返回true，否则返回false  
{  
    return (last == -1)?true:false;}  
}
```

```
template<class T>bool SeqList<T>::IsFull() const  
//判断表满否，满返回true，否则返回false  
{  
    return (last == maxSize-1)?true:false;}  
}
```

```
template<class T>int SeqList<T>::Locate(int i) const  
//定位函数，程序2.7(2)  
{  
    if (i >= 1 && i <= last+1)        return i;  
    else    return 0;  
}
```

## 【顺序表的输入】 P49程序2.9(1) 自学

```
template<class T>void SeqList<T>::input()
{
    cout << "开始建立顺序表，请输入表中元素个数:";
    while (1)    {
        assert(cin >> last);
        last--;
        if (last < 0)  cout << "输入错误，需正整数\n";
        else if (last > maxSize-1)
            cout << "输入错误，需小于"<<maxSize<<"\n";
        else    break;    }
    cout << "\n输入表项： " << endl;
    for (int i = 0; i <= last; i++){
        cout << "#" << i+1 << ":";
        assert(cin >> data[i]);    }
}
```

## 【顺序表的输出】 P49程序2.9(2) 自学

```
template<class T>
void SeqList<T>::output()
{
    cout << "\nThe size of the list is:" << last+1 << endl;
    for (int i = 0; i <= last; i++)
        cout << "#" << i+1 << ":\t" << data[i] << endl;
}
```

# 【顺序表的搜索】 P48程序2.7(1)



```
template <class T>
```

```
int SeqList<T>::search(T& x) const {
```

```
//在表中顺序搜索与给定值 x 匹配的表项，找到则
```

```
//函数返回该表项是第几个元素，否则函数返回0
```

```
    for (int i = 0; i <= last; i++)           //顺序搜索
```

```
        if ( data[i] == x ) return i+1;
```

```
                                //表项序号和表项位置差1
```

```
    return 0;                       //搜索失败
```

```
}//时间复杂度？
```





# 顺序表搜索的性能分析

- 搜索成功的平均比较次数ACN(Average Comparing Number)

$$ACN = \sum_{i=0}^{n-1} p_i \times c_i$$

$p_i$  是搜索第  $i$  项的概率  
 $c_i$  是找到时的比较次数

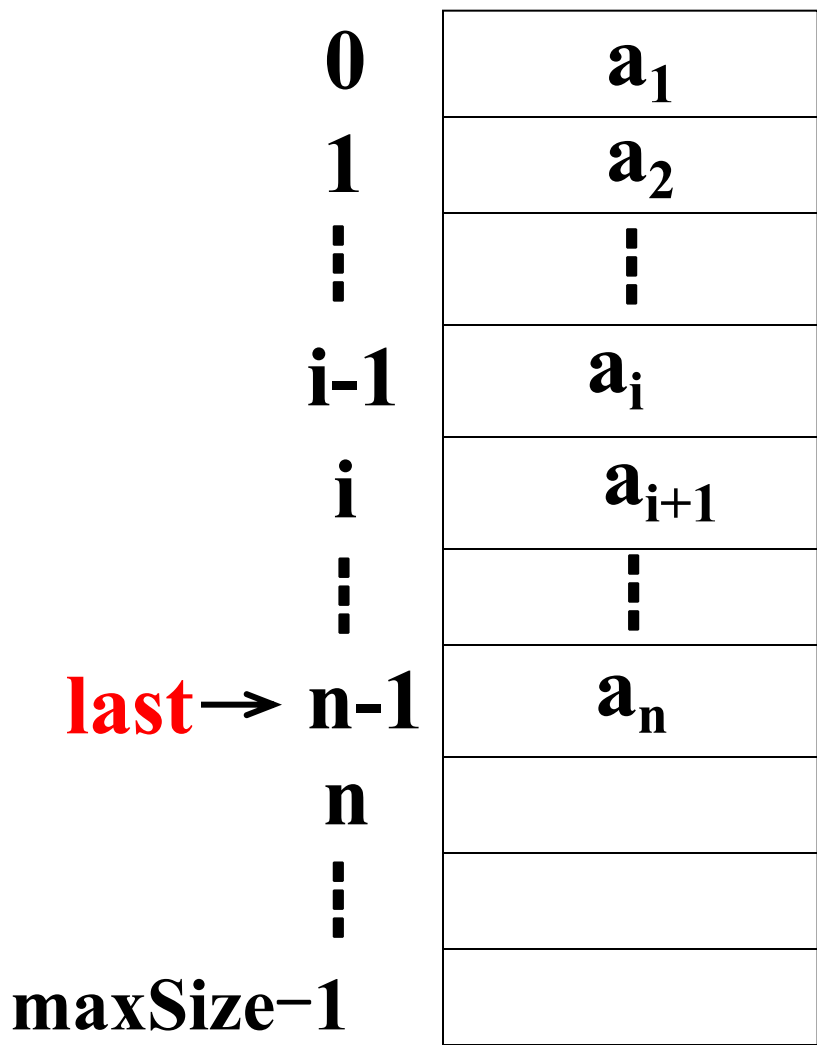
若搜索概率相等，则

$$\begin{aligned} ACN &= \frac{1}{n} \sum_{i=0}^{n-1} (i+1) = \frac{1}{n} (1 + 2 + \cdots + n) = \\ &= \frac{1}{n} * \frac{(1+n) * n}{2} = \frac{1+n}{2} \end{aligned}$$

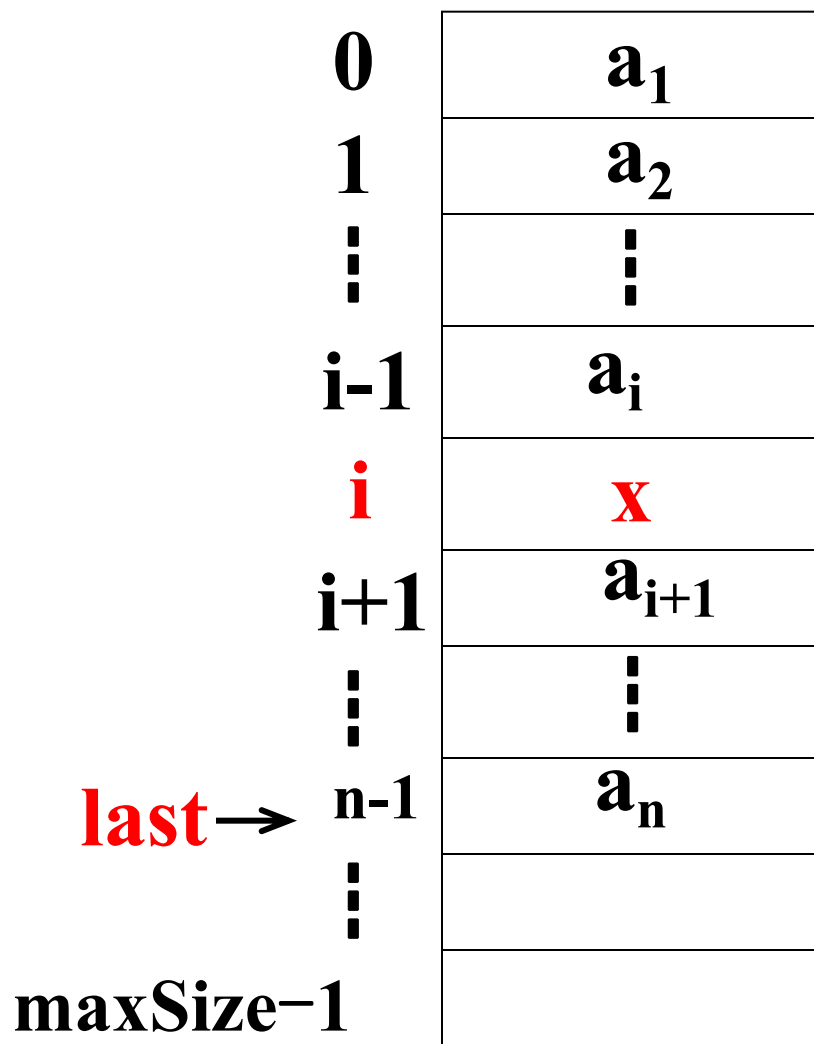
- 搜索不成功，数据比较  $n$  次

## 【顺序表的插入（后插）】

在表中的第 $i$ 个元素（ $1 \leq i \leq \text{表长}$ ）之后插入一个新的数据元素 $x$ ：使长度为 $n$ 的线性表  $(a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$  变成长度为 $n+1$ 的线性表  $(a_1, a_2, \dots, a_i, x, a_{i+1}, \dots, a_n)$ 。



插入元素前



插入元素后

**可插入位置：**  $-1 \leq i-1 \leq \text{last}$  即  $0 \leq i \leq \text{last}+1$

若  $i==0$ ，表中元素全部后移（特别慢）

若  $i==\text{last}+1$ ，无需移动（特别快）

插入操作的非法情况：

（1）若  $i < 0$  或  $i > \text{last}+1$ ，则非法，退出。

（2）表满时（ $\text{last} == \text{maxSize}-1$ ）不能插入。

0	$a_1$
1	$a_2$
$\vdots$	$\vdots$
$i-1$	$a_i$
$i$	$a_{i+1}$
$\vdots$	$\vdots$
$\text{last} \rightarrow n-1$	$a_n$
$n$	<b>X</b>
$\vdots$	
$\text{maxSize}-1$	

# 【顺序表的插入】

P49程序2.8(1)



```
template <class T>
bool SeqList<T>::Insert (int i, T& x) {
//将新元素x插入到表中第i 个表项( $0 \leq i \leq \text{last}+1$ ) 之后
    if (last == maxSize-1) return false;           //表满
    if (i < 0 || i > last+1) return false;         //参数i不合理
    for (int j = last; j >= i; j--)                //依次后移
        data[j+1] = data[j];
    data[i] = x;                                    //插入
    last++;
    return true;                                    //插入成功
} //时间复杂度 ?
```

# 插入算法的时间复杂度分析

## ——元素移动的次数 期望值

设在第 $i$ 个元素之后插入一个元素的概率为 $P_i$ ;

在第 $i$ 个元素之后进行插入时, 元素的移动次数为:

$$n-(i+1)+1=n-i;$$

可插入的位置 $i$ :  $0 \leq i \leq n$ , 共有 $n+1$ 个。

$\therefore$  平均移动次数为:

$$E_{is} = \sum_{i=0}^n p_i (n-i)$$

不失一般性，我们可以假定在线性表的任何位置上插入元素都是等概率的，则  $P_i = 1/(n+1)$ ，所以，

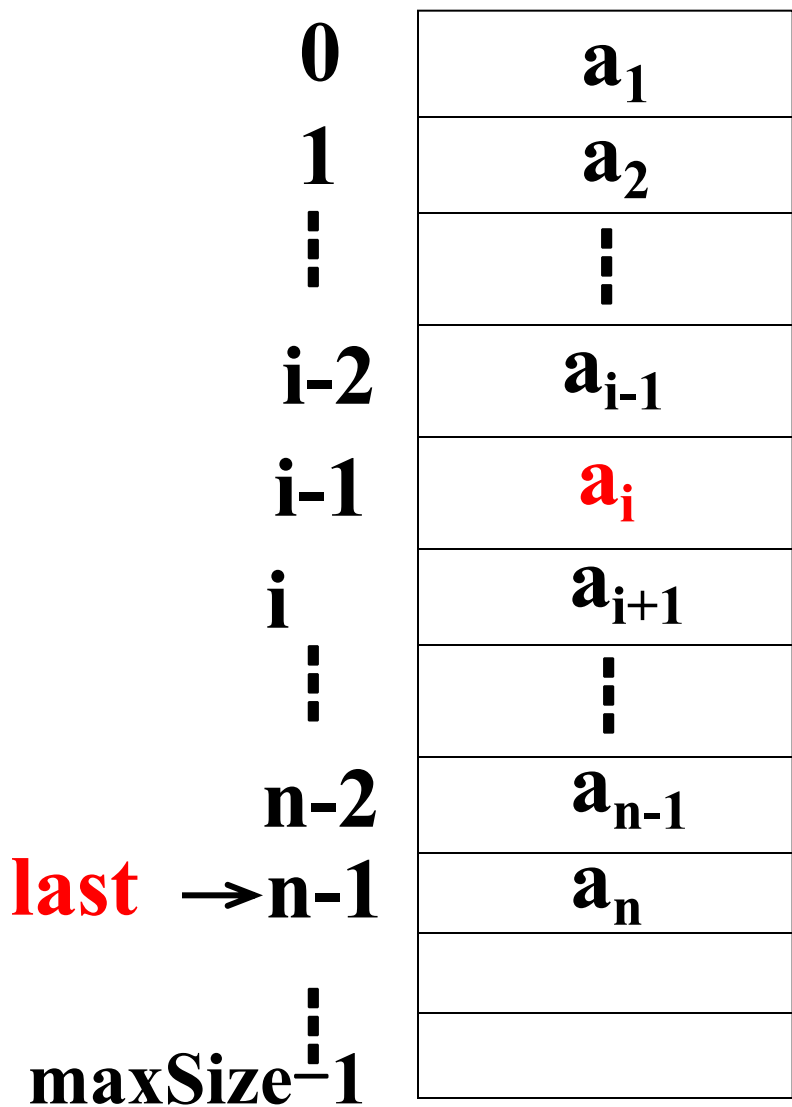
$$E_{is} = \frac{1}{n+1} \sum_{i=0}^n (n-i) = \frac{1}{n+1} \cdot \frac{(n+1)(n+0)}{2} = \frac{n}{2}$$

由上式可见，在顺序表中插入一个数据元素，平均约需移动表中一半元素。若表的长度为 $n$ ，则插入算法的时间复杂度为 $O(n)$ 。

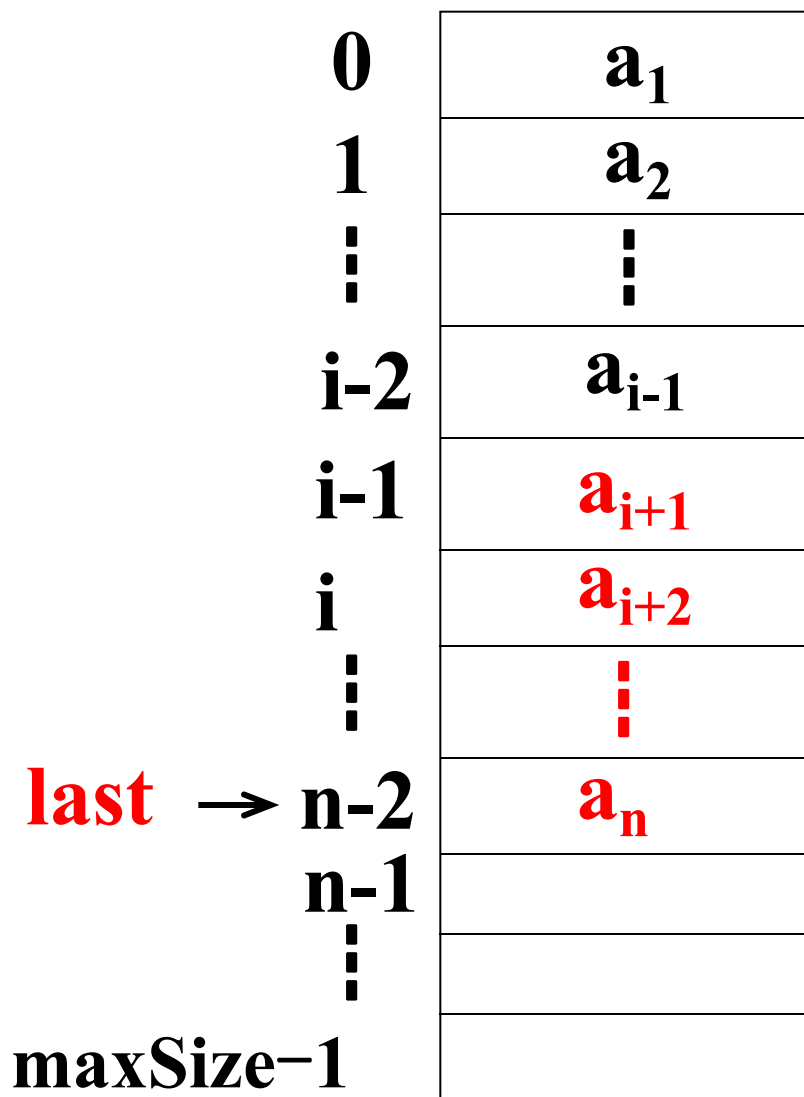
## ◆ 删除

线性表的删除运算是删除第 $i$ 个元素  
( $1 \leq i \leq \text{表长}$ )，使长度为 $n$ 的线性表  
( $a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$ )变成长度为 $n-1$ 的  
线性表 ( $a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n$ )。





删除元素前



删除元素后

可删除位置：  $0 \leq i-1 \leq \text{last}$  即  $1 \leq i \leq \text{last}+1$

- 若  $i=1$ ，表中元素全部前移（特别慢）
- 若  $i=\text{last}+1$ ，无需移动（特别快）

删除操作的非法情况：

- （1）若  $i < 1$  或  $i > \text{last}+1$ ，则非法，退出。
- （2）若  $\text{last} = -1$ ，表空，不能删除，退出。

# 【顺序表的删除】

P49程序2.8(2)



```
template <class T>
bool SeqList<T> :: Remove ( int i, T& x ) {
    //在顺序表中删除第i个表项( $1 \leq i \leq \text{last}+1$ ), 通过引用型
    //参数x返回删除的元素值
    if ( last == -1) return false; //表空, 不能删除
    if ( i < 1 || i > last+1) return false; //参数i不合法
    x = data[i-1]; //存被删元素的值
    for(int j=i; j<=last; j++)
        data[j-1] = data[j]; //依次前移
    last--; //最后位置减1
    return true; //删除成功
} //时间复杂度?
```

## ■ 删除算法的时间复杂度分析

设删除第 $i$ 个元素的概率为 $q_i$ ;

删除第 $i$ 个元素时, 元素的移动次数为:

$$n-(i+1)+1=n-i;$$

可删除的位置 $i$ :  $1 \leq i \leq n$ , 共有 $n$ 个。

$\therefore$  平均移动次数为:

$$E_{de} = \sum_{i=1}^n q_i (n - i)$$

同样，我们假定在线性表的任何位置上删除元素都是等概率的，则 $q_i = 1/n$ ，所以，

$$E_{de} = \frac{1}{n} \sum_{i=1}^n (n-i) = \frac{1}{n} \cdot \frac{n(n-1+0)}{2} = \frac{n-1}{2}$$

由上式可见，在顺序表中删除一个数据元素，平均约需移动表中一半元素。若表的长度为 $n$ ，则删除算法的时间复杂度为 $O(n)$ 。

# 空间管理

## 静态空间管理

- 空间容量固定
- 弱点：不足以存放所有元素，尽管系统仍然有足够空间。
- 糟糕的是，一般的应用环境难以准确预测空间的需求量。需要可随实际需求动态调整容量，并同时保证高效率。

## 动态空间管理

- 在即将发生溢出时，适当地扩大数组的容量
- 如何扩容？
  - 直接在原有物理空间的基础上追加空间？这不现实。
  - 另行申请一个容量更大的数组，并将原数组中的成员集体搬迁至新的空间，此后方可顺利地插入新元素而不致溢出。原数组所占的空间需要及时释放并归还操作系统。

# 【顺序表改变数组空间大小函数】 P47 程序2.6(3)

```
template<class T>void SeqList<T>::reSize(int newSize)
{
    if ( last < maxSize-1) return; //尚未满员时不必扩容，懒惰策略
    if (newSize <= 0){
        cerr << "Invalid array index!" << endl;        return;    }
    if (newSize > maxSize ){
        T *newarray = new T[newSize];
        if (newarray == NULL){
            cerr << "Memory allocating error!" << endl;exit(1);}
        int n = last+1;
        T *srcptr = data;
        T *destptr = newarray;
        while (n-- )      *destptr++ = *srcptr++;
        delete [] data;
        data = newarray;
        maxSize = newSize;    }
} //时间复杂度？
```

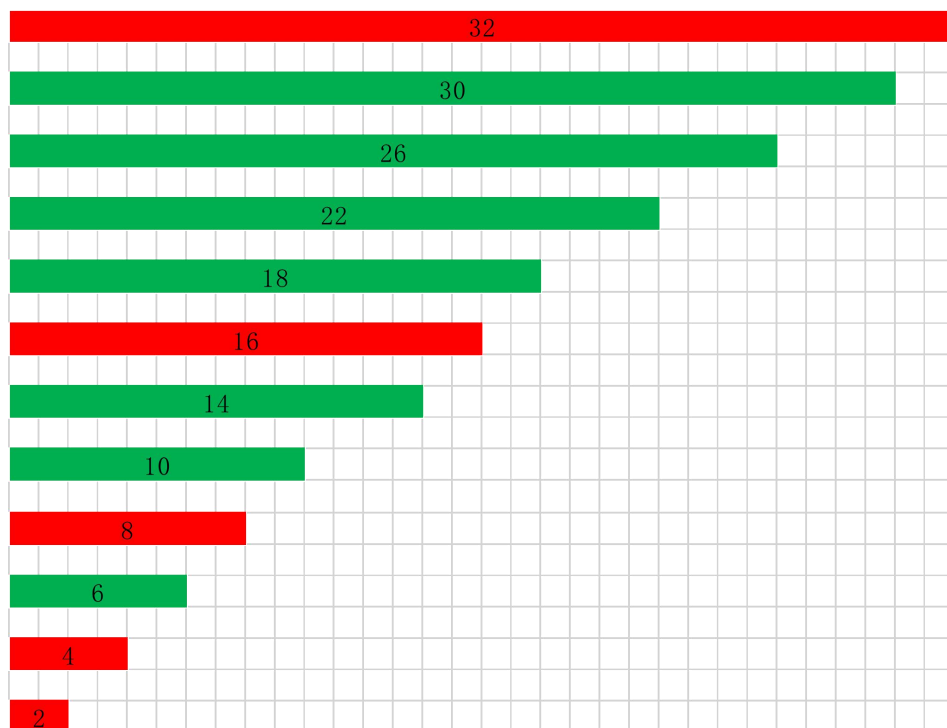
## ■ newSize多大合适？

- ◆ **递增策略**  $\text{reSize}(\text{maxSize}+I)$ : 设连续插入 $n=m*I$ 次，每经 $I$ 次插入就要扩容，总共要做 $m=n/I$ 次扩容（即第 $1, I+1, 2I+1, \dots, (m-1)I+1$ 次插入时），每次扩容中复制原数据的时间成本是 $0, I, 2I, \dots, (m-1)I$ （算术级数），因此总体消耗 $= I(m-1)*m/2 = O(n^2)$ ，每次扩容的分摊成本是 $I(m-1)/2 = O(n)$ ，每次插入的分摊成本是 $O(n)$ 。
- ◆ **倍增策略**  $\text{reSize}(2*\text{maxSize})$ : 设连续插入 $n=2^m$ 次，每经加倍次插入就要扩容，总共要做 $m=\log_2 n$ 次扩容（即第 $2, 4, 8, 16 \dots 2^m$ 次插入时），每次扩容中复制原数据的时间成本是 $1, 2, 4, 8, \dots, 2^{m-1}$ （几何级数），因此总体消耗 $= 1+2+4+8+\dots+2^{m-1} = 2^m - 1 = n - 1 = O(n)$ ，每次扩容的分摊成本是 $(2^m - 1)/m = O(n/\log_2 n)$ ，每次插入的分摊成本是 $O(1)$ 。



倍增策略 `reSize(2*maxSize)`

递增策略 `reSize(maxSize+4)`



	递增策略	倍增策略
平均扩容时间	$O(n)$	$O(n)$
累计扩容时间	$O(n^2)$	$O(n)$
分摊插入时间	$O(n)$	$O(1)$
装填因子 $=n/\text{maxSize}$	$\approx 100\%$	$>50\%$

选择：采用倍增策略

## ■ 插入操作修改

当溢出时进行扩容。 `if (last == maxSize-1) reSize(2*maxSize);`

# ● 顺序分配的优缺点

## ◆ 优点

- (1) 算法简单;
- (2) 存取元素方便, 时间固定, 时间代价为  $O(1)$  ;
- (3) 空间利用率高。

空间利用率 $d$  定义为:

$$d = \frac{\text{数据元素的值所需的存储量}}{\text{该数据元素所需的存储总量}}$$

## ◆ 缺点

(1) 静态存储分配需预先分配空间，按最大空间分配，空间利用不充分；动态存储分配可扩充表容量但耗时。

(2) 当存储密集装载时表现低效，通常应用于有相当多的存储空间空置。

(3) 插入和删除时间代价为  $O(n)$ 。

为克服缺点，引入另一种存储形式：

——**链式存储结构**

# 小结

- 顺序表数据结构及其实现 (**SeqList.h**)
- 顺序存储结构的优缺点

## ● 顺序表应用实例 (自学)

建立一个线性表，首先依次输入数据元素1，2，3，...，10，然后删除数据元素4，最后依次显示当前线性表中的数据元素。要求采用顺序表实现，假设该顺序表的数据元素个数在最坏情况下不会超过100个。

- 文件 **SeqList.h** ( P44—50 )
- 文件 **SeqListTest.cpp**

# SeqListTest.cpp

```
typedef int DataType;           //定义具体的元素类型，先定义再使用
```

```
#include "SeqList.h"
```

```
void main(void)
```

```
{  SeqList<DataType> myList;
```

```
    //定义顺序表类对象，最大表长为100
```

```
    int n = 10,i; DataType x;
```

```
    for(i = 1; i <=n; i++) //顺序插入10个元素
```

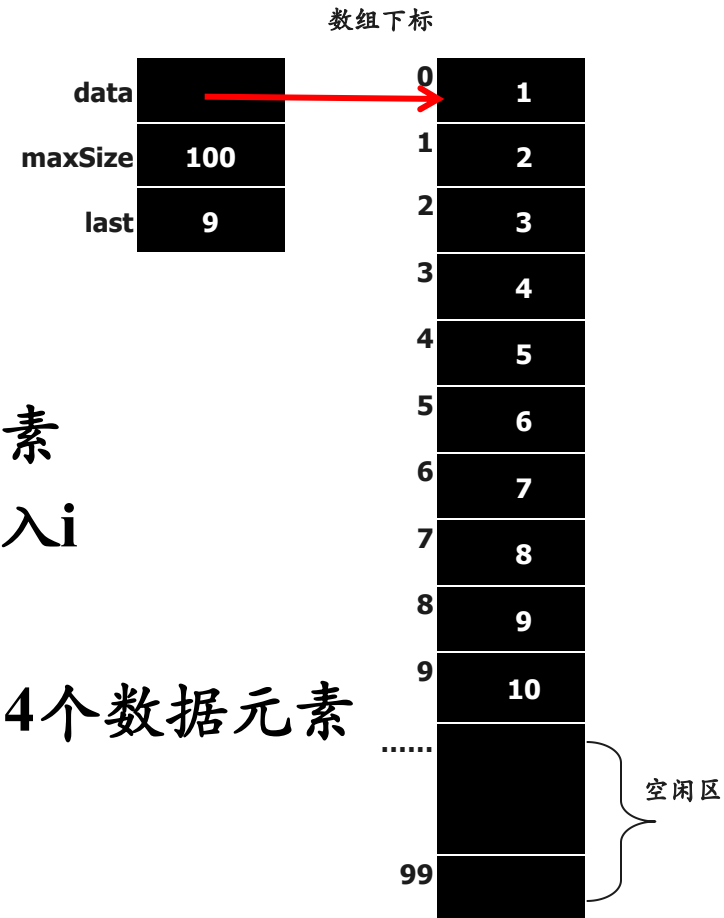
```
        myList.Insert(i-1,i); // 在表尾处插入i
```

```
    myList.output();
```

```
    myList.Remove(4,x); //删除myList中第4个数据元素
```

```
    myList.output();
```

```
}
```



myList的存储结构

## ● 例、有序表的归并(自学)

### ◆ 问题描述:

若线性表La和Lb中元素按值非递减有序排列，现要求将La、Lb归并为新表Lc，且Lc中元素也按值非递减有序排列。

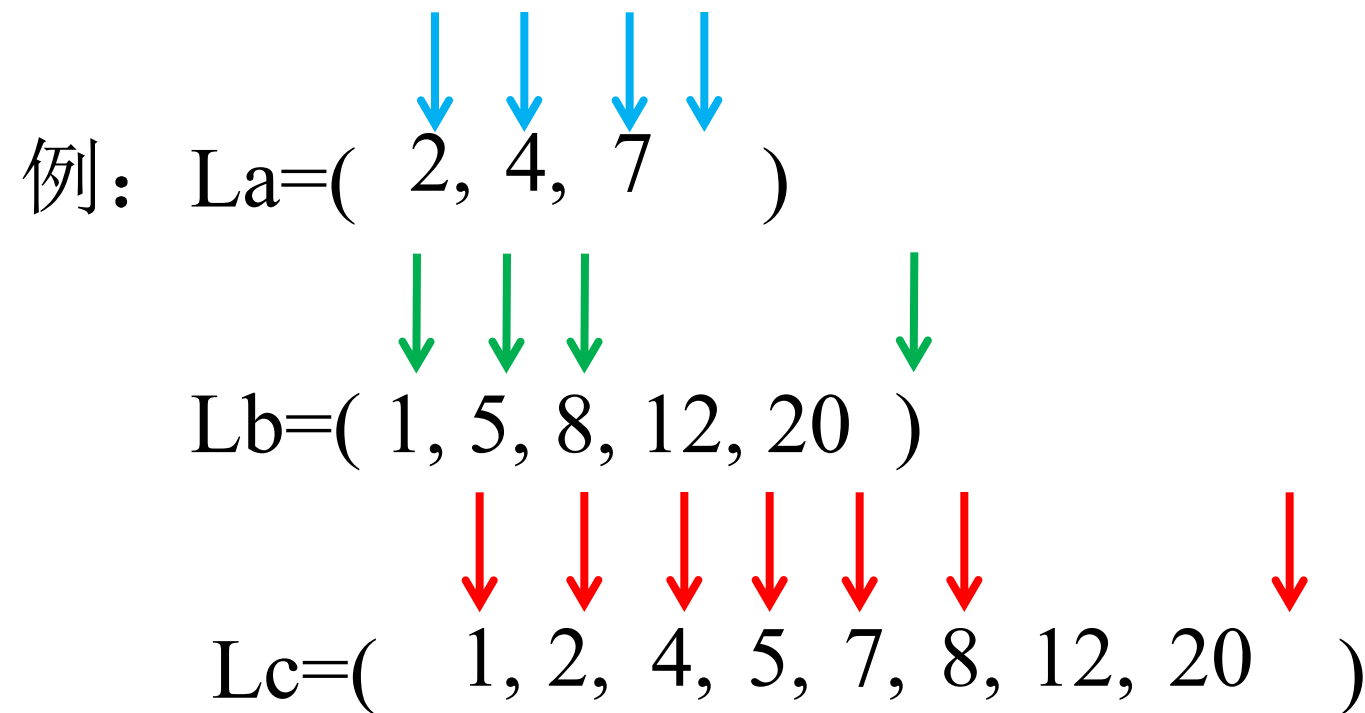
例：La=<2,4,7>    Lb=<1,5,8,12,20>

则 Lc=<1,2,4,5,7,8,12,20>

**思想：**先设一个空表Lc，然后将La或Lb中的元素逐个插入到Lc中即可。

**存储结构：**采用顺序存储结构

**实现：**设三个指针（实际是整型变量） $i, j, k$ 分别指向  $La, Lb$  和  $Lc$  中某个元素，初值分别为0, 0, 0，然后比较  $La, Lb$  中  $i, j$  所指的两个元素的大小，谁小就把谁插入到  $Lc$  表尾，即  $k$  所指的位置。





# 【有序表的归并算法】

```
template<class T>void SeqList<T>::Merge (const SeqList& La, const
SeqList& Lb)
```

/\*将两个非递减有序的顺序表La和Lb合并为表\*this, 并且顺序表  
\*this也按值非递减有序\*/

```
{ int i, j; T xa,xb;
```

```
    last=-1;    i=1; j=1;
```

```
while (i<=La.Length() && j<=Lb.Length())//两个表都有元素未比完
```

```
{    if (last == maxSize-1) reSize(2 * maxSize);
```

```
    La.getData(i,xa); Lb.getData(j,xb);
```

```
    if (xa <=xb )
```

```
    {        Insert(last+1,xa); i++;}
```

```
    else
```

```
    {        Insert(last+1,xb); j++;}
```

```
while (i<La.Size( ))    /*Lb表的元素已经比完*/
{
    if (last == maxSize-1) reSize(2 * maxSize);
    La.getData(i,xa);
    Insert(last+1,xa);
    i++;
}
while(j<Lb.Size( ))    /*La表的元素已经比完*/
{
    if (last == maxSize-1) reSize(2 * maxSize);
    Lb.getData(j,xb);
    Insert(last+1,xb);
    j++;
}
}
```

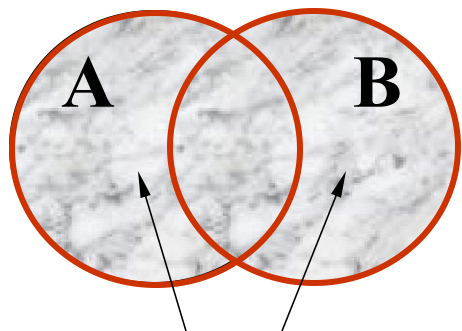
## 时间复杂度分析：

设 $m$ 、 $n$ 分别为 $L_a$ 、 $L_b$ 的长度，则该算法的时间复杂度为： $O(m+n)$ 。

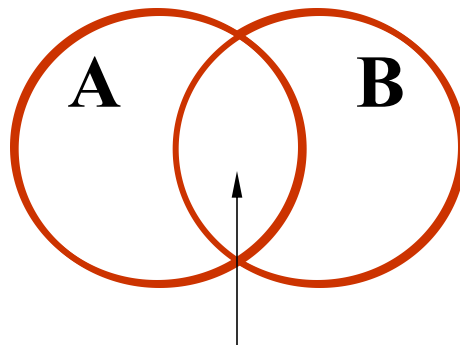
## 6.1 集合及其表示 (自学)

- 集合是成员(元素)的一个群集。集合中的成员可以是原子(单元元素)，也可以是集合。
- 集合的成员是无序的，且互不相同
- 数据结构中的集合
  - 单元元素通常是整数、字符、字符串或指针，且同一集合中所有成员具有相同的数据类型。
  - 为了高效，数据结构中的集合常写成一个序列。
  - 有的集合保存的是实际数据值，某些集合保存的是表示元素在集合与不在集合中的指示信息。
  - 允许元素重复出现的集合叫做多重集合。如果集合保存的是实际数据值，可给每个数据附加一个出现次数计数器就可以实现多重集合。

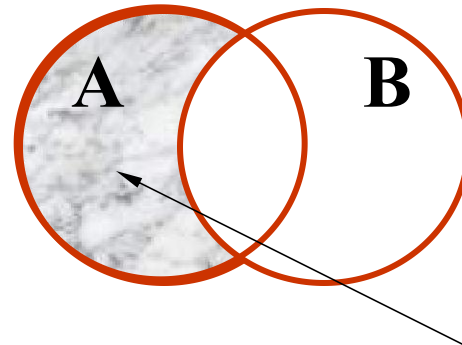
## ■ 集合的操作



**$A \cup B$  或  $A+B$**



**$A \cap B$  或  $A \times B$**



**$A-B$**

# 集合(*Set*)的抽象数据类型

**template <class T> *class Set* {** //P252 程序6.1

**public: Set ();**

**void makeEmpty ();** //置空集合

**bool addMember (const T x);** //加入新成员

**bool delMember (const T& x);** //删除老成员

**Set<T>& unionWith (const Set<T>& R);**//集合的并运算

**Set<T>& intersectionWith (const Set<T>& R);**//集合的交

**Set<T>& differenceFrom (const Set<T>& R);**//集合的差运

**bool Contains (const T x);** //判是否集合的成员

**bool Equal (const Set<T>& R);** //判集合是否相等

**bool subSet (const Set<T>& R);** //判是否集合的子集

**}**

## 6.1.2 用位向量实现集合抽象数据类型

- 现实中任一个有限集合都可以对应到 $\{0, 1, 2, \dots, n\}$ 的整数序列。当全集合是由有限的可枚举的成员组成的集合时，可建立全集合成员与整数 $0, 1, 2, \dots$ 的一一对应关系。
- 建立一个全集合 $\{0, 1, 2, \dots, n\}$ ，现实中的集合成为这个全集合的一个子集。当 $n$ 不大时，可用二进位 $(0, 1)$ 向量来表示全集合的子集合。
- 一个二进位只有两个取值：1或0，分别表示成员在集合与不在集合。
- 需要在getMember(x)和putMember(x,b)中考虑已知元素值x如何求出x在二进位向量中的位置。
- 如果有必要，可以从二进位向量直接映射回原来集合。

# 使用位向量(bit Vector)表示集合的类的定义 P253 程序6.2

```
#include <assert.h>
```

```
const int DefaultSize = 100;
```

```
template <class T>
```

```
class bitSet {
```

```
private:
```

```
    unsigned short * bitVector; //每个数组元素是个16位二进制数
```

```
    int vectorSize;             //16位二进制数的个数
```

```
    int setSize;                //全集合的成员个数
```

```
//Set实现全集合={1,2,.....setSize}及其子集合
```

```
/*例如，有一全集合有setSize=3000个成员，则vectorSize为
```

```
    (3000+15) /16=188，而 bitVector 是188个无符号短整数（16  
    位），具有188*16=3008位的连续空间，其中每位的值分别表示  
    相应成员在/不在集合Set对象中*/
```



**public:**

**bitSet** ( **int** sz = DefaultSize );

**~bitSet** ( ) { **delete** [ ] bitVector; }

**void** makeEmpty ( ) {**for** (**int** i = 0; i <vextorSize; i++ ) bitVector[i] =0;}

**unsigned short** **getMember** ( **const** **T** x );

//返回集合元素x是否在集合Set对象中

**void** putMember(**const** **T** x, **unsigned short** b);

//将值b(0或1)赋值给集合Set对象中的元素x

**int** addMember ( **const** **T** x );      //加入新成员

**int** delMember ( **const** **T** x );      //删除老成员

**bitSet**<**T**>& **operator** = (**const** **bitSet**<**T**> & R ); //赋值自R

**bitSet**<**T**>& **operator** + (**const** **bitSet**<**T**> & R ); //并

**bitSet**<**T**>& **operator** \* (**const** **bitSet**<**T**> & R ); //交

**bitSet**<**T**>& **operator** - (**const** **bitSet**<**T**> & R ); //差

**bool** Contains ( **const** **T** x );      //判x是否是集合成员

**bool** subSet ( **bitSet**<**T**> & R );      //判是否是R的子集

**bool** **operator** == ( **bitSet**<**T**> & R );      //判集合相等

};

## 使用示例

```

bitSet<int> s1, s2, s3, s4, s5; int index, equal;
for ( int k = 0; k < 10; k++ )    //集合赋值
{ s1.addMember( k ); s2.addMember( k +7 ); }
// s1 : { 0, 1, 2, ..., 9 }, s2 : { 7, 8, 9, ..., 16 }
s3 = s1+s2;    //求s1与s2的并 { 0, 1, ..., 16 }
s4 = s1*s2;    //求s1与s2的交 { 7, 8, 9 }
s5 = s1-s2;    //求s1与s2的差 { 0, 1, ..., 6 }
index = s1.subSet ( s4 ); //判断s1是否为s4子集
cout << index << endl;    //结果, index = 0
// s4 : { 7, 8, 9 }
equal = s1 == s2;          //集合s1与s2比较相等
cout << equal << endl;    //为0, 两集合不等

```

# 用位向量实现集合的部分操作实现

```
template <class T>  //P254 程序6.2(2)
bitSet<T> :: bitSet (int sz) : setSize(sz) {
    assert ( setSize > 0 );
    vectorSize=(setSize+15)>>4;  //例如(3000+15)/16=188
    bitVector = new unsigned int[vectorSize];
    assert ( bitVector != NULL );
    for ( int i = 0; i < vectorSize; i++ )
        bitVector[i] = 0;
}
```

//P254 程序6.3(1)

```
template <class T> //返回集合元素x是否在集合Set对象中
unsigned int bitSet<T> :: getMember ( const T x )
{ assert ( x >= 0 && x < setSize);
  int address = x/16, id = x%16;
  unsigned short v = bitVector[address];
  unsigned short tmp= v >> (15-id);
  return ( tmp %2 );
}/*例如，setSize=3000，x=2001，
```

address=2001/16=125（向下取整），

id=2001%16=1（bitVector中第125个无符号整数的第1位表示2001是否在集合Set对象中），

v 赋值自bitVector中188个无符号短整型数组组成的数组中读出的第125个无符号短整数(16位二进制)，

将v右移(15-1)=14位得到一个新的无符号短整数tmp，通过对2求余获得tmp最低位值（0或1）返回，该值即表示x是否在集合Set对象中

\*/

//P254 程序6.3(2)

```
template <class T> //将值b(0或1)赋值给集合Set对象中的元素x
unsigned int bitSet<T> :: putMember ( const T x,
unsigned int b ) { assert ( x >= 0 && x < setSize);
    int address = x/16,    id = x%16;
    unsigned short v = bitVector[address];
    tmp = v >> (15-id) ;
    v = v << (id+1) ;
    if ( tmp%2 == 0 && b == 1) tmp = tmp + 1;
    else if( tmp%2 == 1 && b == 0) tmp = tmp - 1;
    bitVector[address] = (tmp << (15-id)) | (v >> (id+1)) ;}
```

/\*例如，setSize=3000，x=2001,address=2001/16=125，id=2001%16=1，  
v 赋值自bitVector中188个无符号短整型数组成的数组中读出的第125个无符号短整数  
将v右移(15-1)=14位得到一个新的无符号短整数tmp，

将v左移(1+1)=2位得到一个新的无符号短整数赋值给v

如果tmp的最低位值为0并且b值为1则tmp值最低位修改为1，否则如果tmp的最低位值为1  
并且b值为0，则tmp值最低位修改为0。

将tmp左移(15-1)=14位，将v右移(1+1)=2位，然后二者做或运算，赋值给bitVector中第125  
个无符号短整型数。\*/

```
template <class T> //集合加入新成员x P255 程序6.4(1)
bool bitSet<T> :: addMember ( const T x ) {
    assert ( x >= 0 && x < setSize );
    if ( getMember(x) == 0 )
        { putMember(x,1); return true; }
    return false;
}
```

```
template <class T> //删除集合老成员x P255 程序6.4(2)
bool bitSet<T> :: delMember ( const T x ) {
    assert ( x >= 0 && x < setSize );
    if ( getMember(x) == 1 )
        { putMember(x,0); return true; }
    return false;
}
```

```
template <class T>    //判x是否在集合Set对象中 P255 程序6.4(6)
bool bitSet<T> :: Contains ( const T x ) {
    assert ( x >= 0 && x < setSize );
    return (getMember(x) == 1) ? true : false;
};
```

```
template <class T>    //赋值自集合R
bitSet<T>& bitSet<T> :: operator = ( bitSet<T>&
R) { assert ( vectorSize == R.vectorSize );
    for ( int i = 0; i < vectorSize; i++ )
        bitVector[i] = R.bitVector[i];
    return *this;
};
```

```

template <class T>           //求集合的并 P255 程序6.4(3)
bitSet<T>& bitSet<T> :: operator + ( const bitSet<T>& R )
{
    assert ( vectorSize == R.vectorSize );
    bitSet tmp ( vectorSize );
    for ( int i = 0; i < vectorSize; i++ )
        tmp.bitVector[i] = bitVector[i] | R.bitVector[i];
    return tmp;
}

```

this	0	1	1	1	0	0	0	0	1	1	0
R	0	0	1	0	0	1	0	1	0	1	0
tmp	0	1	1	1	0	1	0	1	1	1	0



```

template <class T>           //求集合的交 P255 程序6.4(4)
bitSet<T>& bitSet<T> :: operator * ( const bitSet<T>& R )
{
    assert ( vectorSize == R.vectorSize );
    bitSet tmp ( vectorSize );
    for ( int i = 0; i < vectorSize; i++ )
        tmp.bitVector[i] = bitVector[i] & R.bitVector[i];
    return tmp;
}

```

this	0	1	1	1	0	0	0	0	1	1	0
R	0	0	1	0	0	1	0	1	0	1	0
tmp	0	0	1	0	0	0	0	0	0	1	0

```

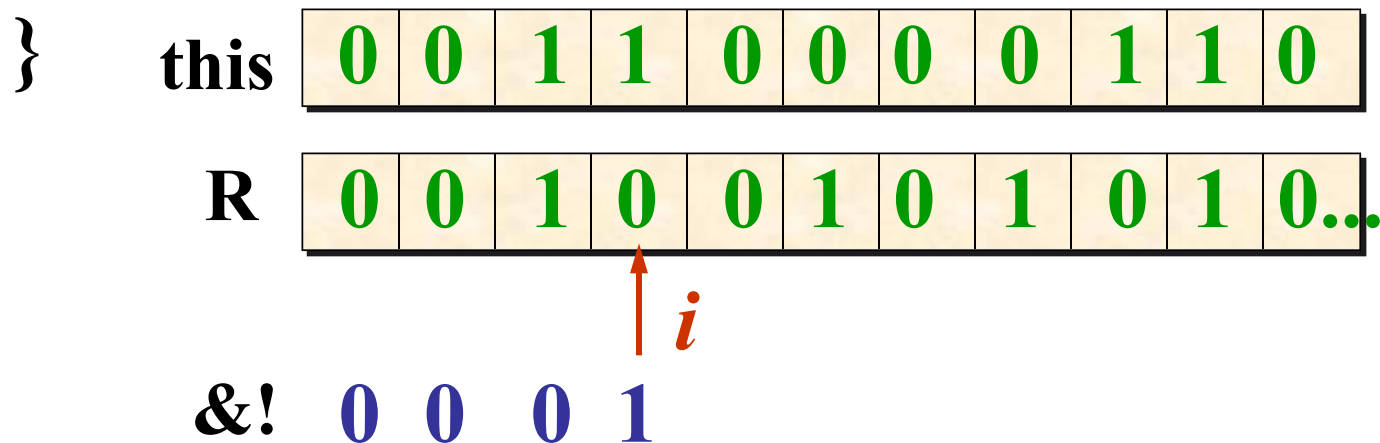
template <class T>           //求集合的差 P255 程序6.4(5)
bitSet<T>& bitSet<T> :: operator - ( const bitSet<T>& R )
{
    assert ( vectorSize == R.vectorSize );
    bitSet tmp ( vectorSize );
    for ( int i = 0; i < vectorSize; i++ )
        tmp.bitVector[i] = bitVector[i] & ! R.bitVector[i];
    return tmp;
}

```

this	0	1	1	1	0	0	0	0	1	1	0
R	0	0	1	0	0	1	0	1	0	1	0
tmp	0	1	0	1	0	0	0	0	1	0	0

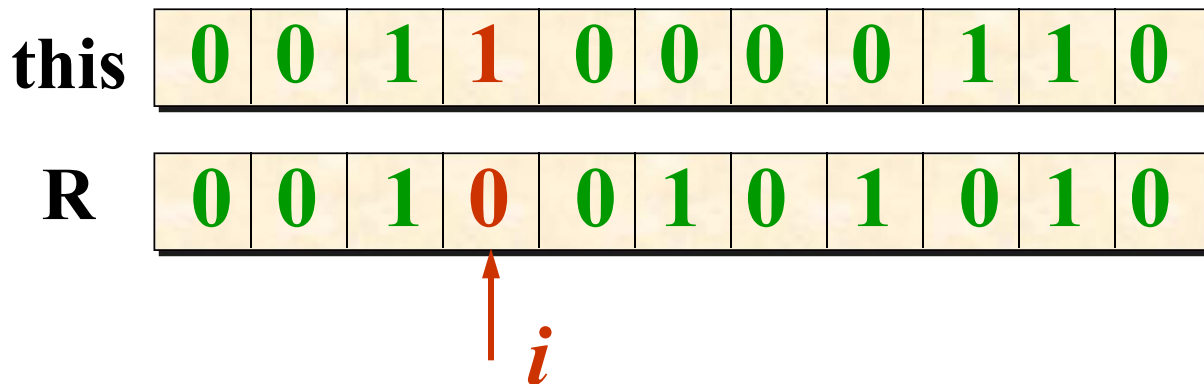
**template <class T>** //判是否是R的子集合 P255 程序6.4(7)

```
bool bitSet<T> :: subSet ( bitSet<T>& R ) {  
    assert ( setSize == R.setSize );  
    for ( int i = 0; i < vectorSize; i++)  
        if (bitVector[i] & ! R.bitVector[i])  
            return false;  
    return true;  
}
```



/\*思考： 算法仅从R的第0位起判子集，推广从R的其它位起判子集，如何修改算法\*/

```
template <class T> //判断与集合R相等P255 程序6.4(8)
bool bitSet<T> ::operator == ( bitSet<T>& R )
{ if ( vectorSize == R.vectorSize )
    return false;
  for ( int i = 0; i < vectorSize; i++)
    if ( bitVector[i] != R.bitVector[i] )
      return false;
  return true;
}
```



# 自学2

- 1、P47 程序2.6(2)SeqList()
- 2、P47 程序2.5中Size()、Length()、getData()、setData()、IsEmpty()、IsFull()
- 3、P48 程序2.7(2)Locate()
- 4、P49 程序2.9(1,2)Input()、Output()
- 5、P52 程序2.10 应用：顺序表实现集合的并交运算
- 6、P251-P257: 6.1.1-6.1.2 用位向量实现集合抽象数据类型

# 作业2

## 算法设计题：

2-1、有序顺序表的有序插入函数。

2-2、Uniquify( )函数实现剔除有序单链表中重复元素，要求单链表每个结点被访问一次，且仅被访问一次。算法要求尽可能高效，分析算法的时间复杂度和空间复杂度。例如有序线性表

$L = \langle 9, 9, 9, 9, 19, 19, 19, 19, 19, 20, 20, 20, 30, 30, 30, 30 \rangle$ , 执行Uniquify( )操作后  
 $L = \langle 9, 19, 20, 30 \rangle$ 。

# 要点回顾

- 线性表的抽象数据类型定义和抽象基类
- 顺序表
  - SeqList.h类定义和实现
  - 应用
    - 一方面设计并实现抽象数据类型，另一方面应用已实现的抽象数据类型设计应用软件系统
    - 算法书写规范
  - 顺序存储结构的优缺点