数据结构与算法

计算机学院

朱晨阳 副教授

zhuchenyang07@nudt.edu.cn

线性表

- 线性表是n个数据元素$d_0,d_1,d_2,\ldots,d_{n-1}$组成的有限序列。

线性表=(D, S)

$D=\{d_0,d_1,d_2,\ldots,d_{n-1}\}$

$S=\{<d_0,d_1>,<d_1,d_2>,<d_2,d_3>,\ldots,<d_{n-2},d_{n-1}>\}$

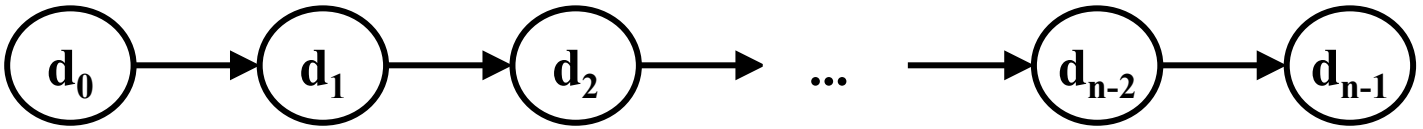
d_0为开始结点(首结点)，无前驱，有唯一后继

d_{n-1}为终端结点(尾结点)，无后继，有唯一前驱

$d_i(0<i<n-1)$有唯一前驱d_{i-1}和唯一后继d_{i+1}。

线性表逻辑结构图

- 线性表的逻辑结构图



线性表的存储

- 线性表的存储方式可以采用顺序存储(第2章)、链式存储(第3章)或散列存储(第6章)，必要时可为数据元素建立索引表，进行索引存储(第6章)。

线性表的运算

- 线性表的运算主要包括
  - 初始化，生成一个空表；
  - 判断表是否为空；
  - 判断表是否已满；
  - 求表长，返回表结点个数；
  - 取表中第i个结点；
  - 查找表中值为x的结点；
  - 在第i个位置上插入一个新结点；
  - 删除表中的第i个结点。

给出线性表的抽象数据类型

ADT LinearList

Data: _____

Operations:

_____

…

…

ADT LinearList

Data: $D=\{d_0,d_1,d_2,\ldots,d_{n-1}\}$; $R=\{<d_0,d_1>,\ldots\}$

Operations:

InitList	{ }
DestroyList	释放线性表所占内存
ListEmpty	true|false
ListFull	true|false
ListLength	|D|
GetElem(i)	d_i
LocateElem(x)	$i(d_0,\ldots,d_{i-1}\neq x,d_i=x) \mid -1(d_0,\ldots,d_{n-1}\neq x)$
InsertElem(i,x)	$\{d_0,d_1,\ldots,d_{i-1},x,d_i,\ldots,d_{n-1}\}$
DeleteElem(i)	$\{d_0,d_1,\ldots,d_{i-1},d_{i+1},\ldots,d_{n-1}\}$

ex: 假设D={31,27,59,40,58}，设计其存储结构，以方便实现这些运算

线性表的实现

- 采用顺序存储的线性表 （2.1.2节，P.15）

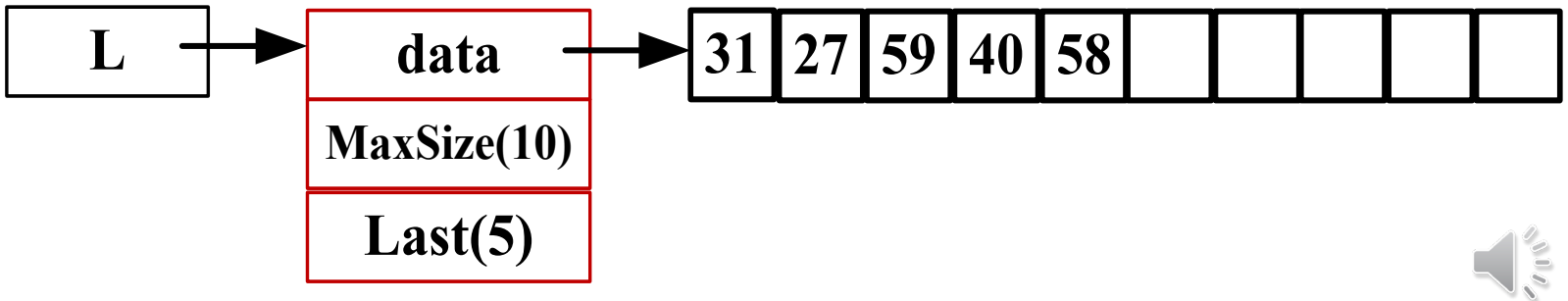
```
/*  linearList.h  */
typedef int ElementType;
enum boolean {FALSE, TRUE};
typedef enum boolean Bool;
struct linearList
{
        ElementType *data;
        int MaxSize;  //能存储的最大元素个数
        int Last;     //线性表当前元素个数
};
typedef struct linearList LinearList;
```

ex: 假设D={31,27,59,40,58}，画出其存储结构

线性表的实现

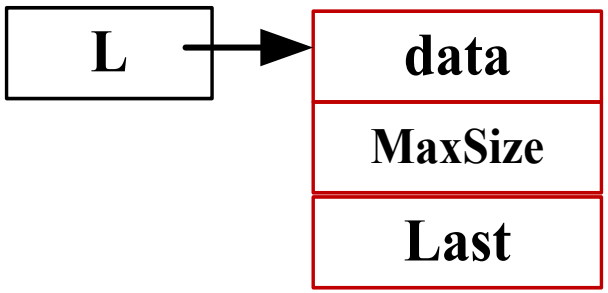
- 采用顺序存储的线性表 （2.1.2节，P.15）

```
/*  linearList.h  */
typedef int ElementType;
enum boolean {FALSE, TRUE};
typedef enum boolean Bool;
struct linearList
{
        ElementType *data;
        int MaxSize;  //能存储的最大元素个数
        int Last;     //线性表当前元素个数
};
typedef struct linearList LinearList;
```



```
void InitList(LinearList *, int sz);
void FreeList(LinearList *);
Bool ListEmpty(LinearList *);
Bool ListFull(LinearList *);
int ListLength(LinearList *);
ElementType GetElem(LinearList *, int i);
int LocateElem(LinearList *, ElementType x);
Bool InsertElem(LinearList *, ElementType x, int i);
Bool DeleteElem(LinearList *,int i);
```

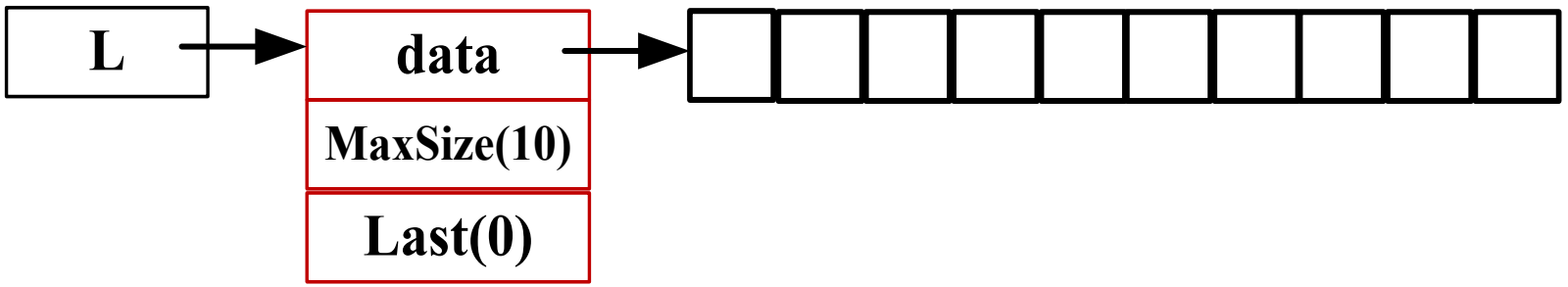
```
/*  linearList.c */
void InitList(LinearList *L, int sz)
{       if (sz>0)       {
                L->MaxSize = sz;
                L->Last = 0;
                L->data = (ElementType *) malloc
                (sizeof(ElementType)*L->MaxSize);
        }
}
```



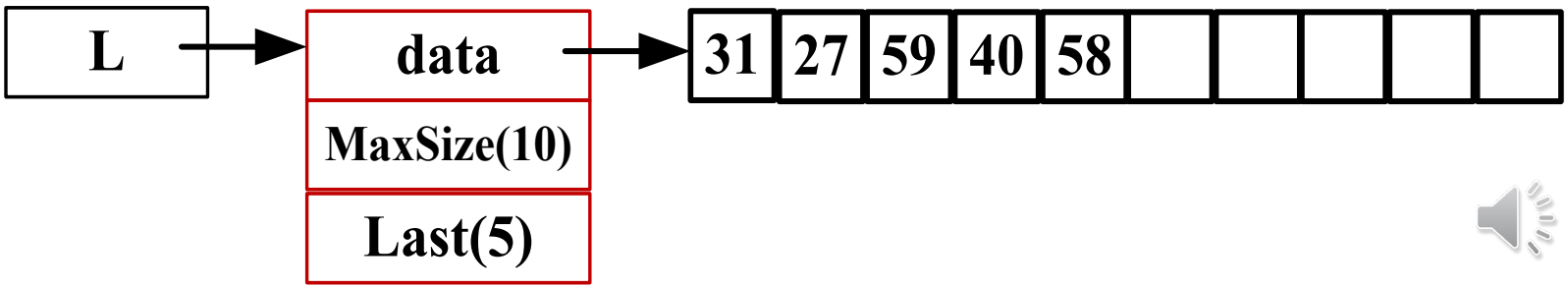
InitList(L,10)
画出结果图

```
/*  linearList.c */
void InitList(LinearList *L, int sz)
{       if (sz>0)       {
                L->MaxSize = sz;
                L->Last = 0;
                L->data = (ElementType *) malloc
                (sizeof(ElementType)*L->MaxSize);
        }
}
```

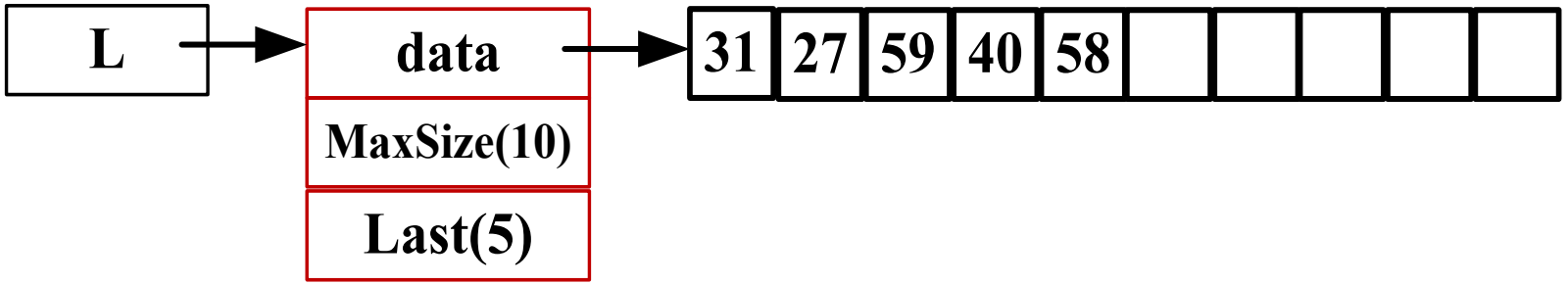
InitList(L,10)结果图



```
void FreeList(LinearList *L)
{       free(L->data);     }
Bool ListEmpty(LinearList *L)
{       return (L->Last <= 0) ? TRUE : FALSE;     }
Bool ListFull(LinearList *L)
{       return (L->Last >= L->MaxSize)?TRUE:FALSE;     }
int ListLength(LinearList *L)
{       return L->Last;     }
```



```
ElementType GetElem(LinearList *L, int i)
/*{d0,d1,...,dn-1} → di*/
{
        return (i<0 || i>=L->Last) ? NULL : L->data[i];
    /*此处取值不成功时返回NULL不妥*/
}
```



int LocateElem(LinearList *L, ElementType x)

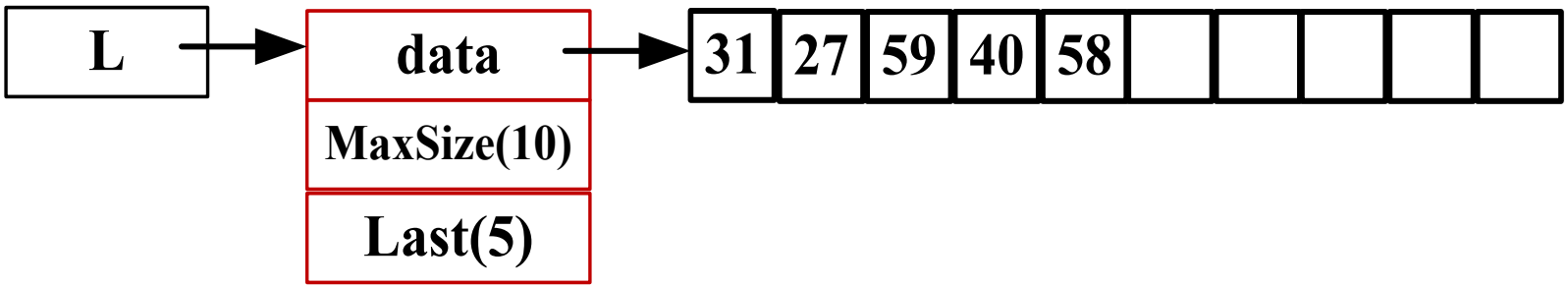
/*查找表中值为x的结点。若查找成功，则返回该结点的序号；否则返回-1。 若表中值为x的结点有多个，找到的是最前面的一个*/

/*{$d_0,d_1,\ldots,d_{n-1}$} →i($d_0,\ldots,d_{i-1}\neq x,d_i=x$) | -1($d_0,\ldots,d_{n-1}\neq x$)*/

{

????????????????

}



int LocateElem(LinearList *L, ElementType x)

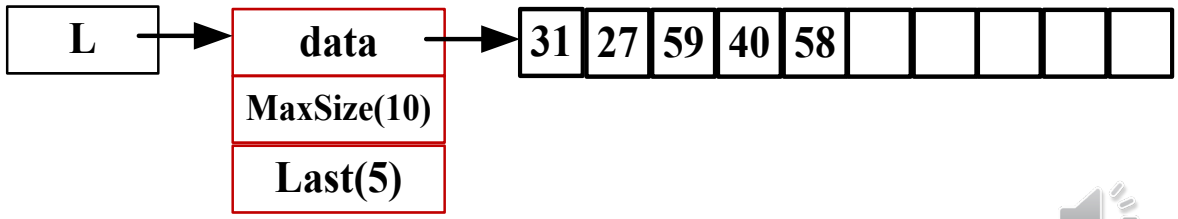
/*查找表中值为x的结点。若查找成功，则返回该结点的序号；否则返回-1。 若表中值为x的结点有多个，找到的是最前面的一个*/

/*$\{d_0,d_1,\ldots,d_{n-1}\} \rightarrow i(d_0,\ldots,d_{i-1}\neq x,d_i=x) \mid -1(d_0,\ldots,d_{n-1}\neq x)$*/

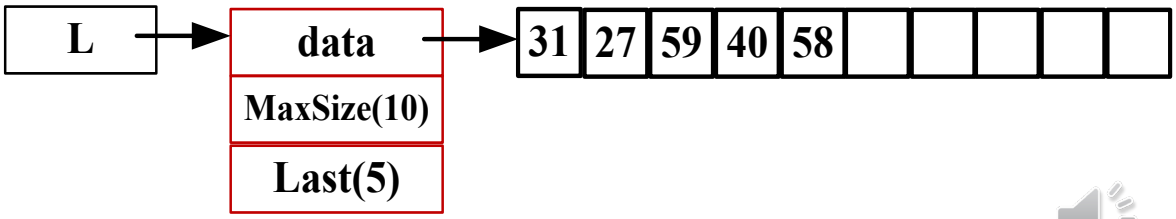
```
{
        int i;
        for (i = 0; i < L->Last; i++)
                if( L->data[i] == x) return i;/*查找成功*/
        return -1;                          /*查找失败*/
}
```

```
Bool InsertElem (LinearList *L, ElementType x, int i)
/*在表中第i个位置插入值为x的结点。
若插入成功，则返回TRUE；否则返回FALSE*/
/*{d0,d1,...,dn-1} → {d0,d1,...,di-1,x,di,...,dn-1}*/
{
        int j;
        if ( i < 0 || i > L->Last ||
        L->Last == L->MaxSize )
                return FALSE; /*插入位置不合理，插入失败*/
        else
        {

                ??????????????  /*后移*/
                ??????????????  /*插入*/
                ??????????????  /*表长增1*/
                return TRUE;
        }
}
```

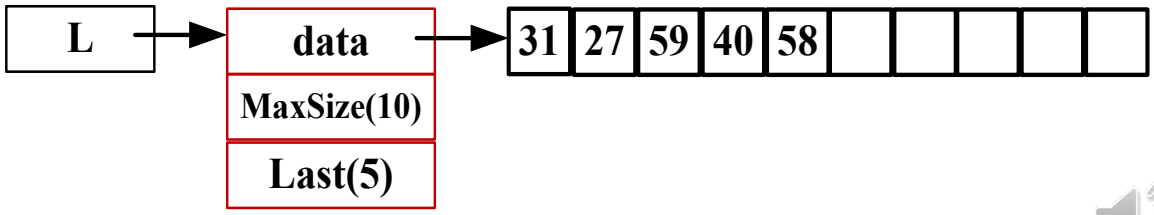


```
Bool InsertElem (LinearList *L, ElementType x, int i)
/*在表中第i个位置插入值为x的结点。
若插入成功，则返回TRUE；否则返回FALSE*/
/*{d0,d1,...,dn-1} → {d0,d1,...,di-1,x,di,...,dn-1}*/
{
        int j;
        if ( i < 0 || i > L->Last ||
        L->Last == L->MaxSize )
                return FALSE; /*插入位置不合理，插入失败*/
        else
        {
                for (j = L->Last-1; j >= i; j--)
                        L->data[j+1] = L->data[j];/*后移*/
                L->data[i] = x; /*插入*/
                L->Last++;      /*表长增1*/
                return TRUE;
        }
}
```

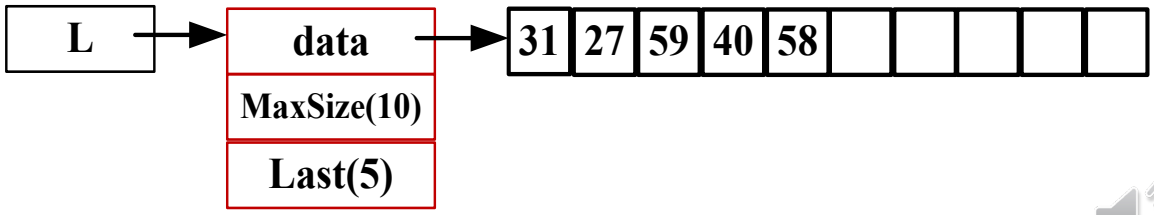


```
Bool DeleteElem(LinearList *L, int i)
/*删除表中第i个结点。若删除成功，则返回TRUE；否则返回FALSE*/
/*{d0,d1,...,dn-1} → {d0,d1,...,di-1,di+1,...,dn-1}*/
{
        int j;
        if (i < 0 || i >= L->Last || L->Last == 0 )
                return FALSE;/*第i个结点不存在，删除失败*/
        else
        {

                ?????????????   /*前移*/
                ?????????????   /*表长减1*/
                return TRUE;
        }
}
/*  end of LinearList.c */
```



```
Bool DeleteElem(LinearList *L, int i)
/*删除表中第i个结点。若删除成功，则返回TRUE；否则返回FALSE*/
/*{d0,d1,...,dn-1} → {d0,d1,...,di-1,di+1,...,dn-1}*/
{
        int j;
        if (i < 0 || i >= L->Last || L->Last == 0 )
                return FALSE;/*第i个结点不存在，删除失败*/
        else
        {
                for (j = i; j < L->Last-1; j++)
                        L->data[j] = L->data[j+1];/*前移*/
                L->Last--;                      /*表长减1*/
                return TRUE;
        }
}
/*  end of LinearList.c */
```



向量

- 向量是由同一种数据类型的数据元素组成的线性表。
- 一般采用数组来存储向量(顺序存储)。数组元素物理上的邻接关系隐式地体现了向量元素逻辑上地相邻关系。

地址	数组下标	
A	0	d_0
A+L	1	d_1
A+2*L	2	d_2
…	…	…
A+i*L	i	d_i
…	…	…
A+(n-1)*L	n-1	d_{n-1}

假设每个数据元素占用L个存储单元

向量

- C语言描述向量的顺序存储及基本操作

```
enum Bool {FALSE, TRUE};
struct Vector
{   ElementType* elements;
    int ArraySize; int VectorLength;
}
void GetArray(Vector *V)
{   V->elements=(ElementType*)malloc(sizeof(ElementType) *V->ArraySize);
    if(V->elements==NULL)
        printf("Memory Allocation Error!\n");
}
```

向量

```
void InitVector(Vector* V, int sz)
{   if(sz<=0) printf("Invalid Array Size\n");
    else
    {   V->ArraySize=sz;
        V->VectorLength=0;
        GetArray(V);
    }
}
ElementType GetNode(Vector* V, int i)
{
     return(i<0||i>=V->VectorLength)?NULL:V->elements[i];
}
```

向量

```
void FreeVector(Vector * V)
{   free(V->elements);
    V->ArraySize=0; V->VectorLength=0;
}

int Find(Vector* V, ElementType x)
{   int i;
    for(i=0; i<V->VectorLength; i++)
        if(V->elements[i]==x)  return i;
    return -1;
}
```

向量的插入操作

- 顺序存储的向量的插入操作：在向量的第i个位置上插入一个值为x的新结点。
  - 设向量元素个数为n，只有n<ArraySize且0≤i≤n时，插入操作才能进行。
  - 若0≤i<n，则插入时必须将元素d_i~d_{n-1}分别往后移动一个数据元素的位置；
  - 若i=n，则只须将新元素放在d_{n-1}之后。插入操作完成后，向量长度增加1。

```
Bool Insert(Vector* V, ElementType x, int i)
{   int j;
    if(V->VectorLength==V->ArraySize){
        printf("overflow\n");   return FALSE;
    }
    else if(i<0||i>V->VectorLength){
        printf("position error\n");  return FALSE;
    }
    else{
        for(j=V->VectorLength-1; j>=i; j--)
            V->elements[j+1]=V->elements[j]; //后移
      V->elements[i]=x;   //插入
        V->VectorLength++;  //向量长度增加1
        return TRUE;
    }
}
```

向量的插入操作

- 设向量长度为n, 则在位置i插入元素，有n-i个元素要移动位置。i的取值范围为[0,n]。假设在向量任何合法的位置上插入新元素的机会是均等的，则元素的平均移动次数为n/2。 平均时间复杂度为O(n)。

向量的删除操作

- 删除向量的位置i处的元素，元素d_{i+1}~d_{n-1}分别往前移动一个数据元素的位置。
- 删除位置i的元素要移动n-i-1个元素， i的取值有n种可能，每种取值概率均等时，平均移动次数为(n-1)/2。
- 向量的删除操作C语言实现

向量的删除操作

```
Bool Remove(Vector *V, int i)
{   int j;
    if(V->VectorLength==0)
    {   printf("Vector is empty\n"); return FALSE; }
    else if(i<0 || i>V->VectorLength-1)
    {   printf("position error\n");  return FALSE; }
    else for(j=i; j<V->VectorLength-1; j++)
                V->elements[j]=V->elements[j+1];
    V->VectorLength--;
    return TRUE;
}
```

求集合的并和交

- 用向量表示集合，两个集合的并运算即将两个向量合并为一个向量，两个向量中相同的元素只保留一个；两个集合的交运算即提取两个向量的相同元素而组成一个向量。
- 并运算的C语言实现
- 交运算的C语言实现

```
Vector* Union(Vector * Va, Vector *Vb)
{   int m, n, i, k, j;  ElementType x;
    Vector * Vc=(Vector*)malloc(sizeof(Vector));
    n=Va->VectorLength;
    m=Vb->VectorLength;
    InitVector(Vc, m+n);
    j=0;
    for(i=0; i<n; i++) {x=GetNode(Va,i); Insert(Vc,x,j); j++;}
    for(i=0; i<m; i++)
    {   x=GetNode(Vb, i);  k=Find(Va, x);
        if(k==-1) {insert(Vc,x,j);  j++;}
    }
    return Vc;
}
```

```
Vector *Intersection(Vector* Va, Vector*Vb)
{   int m,n,i,k,j;  ElementType x;
    Vector* Vc=(Vector*)malloc(sizeof(Vector));
    n=Va->VectorLength;
    m=Vb->VectorLength;
    InitVector(Vc, (m>n)?n:m);
    i=0; j=0;
    while (i<m)
    {   x=GetNode(Vb, i);
        k=Find(Va, x);
        if(k!=-1) { Insert(Vc, x, j);  j++; }
        i++;
    }
    return Vc;
}
```

求集合的并和交

- 对用向量表示的集合Va、Vb合并到Vc中的并运算，与问题规模有关的操作是查找和插入。查找需要做比较，插入需要做移动。
- 设Va和Vb的结点个数分别为n和m，不妨假设n>m。并运算中，对Vb中每一结点，要在Va中查找是否有等值的结点，最好的情况是Va的前m个结点分别与Vb中的一个结点等值，这时需要的比较次数最少：

$$C_{\min} = \sum_{0}^{m-1}(i+1) = \frac{m(m+1)}{2}$$

求集合的并和交

- 并运算最坏的情况是Va中的任何结点都不与Vb中的结点等值，因而Vb中的每个结点经过与Va中的每个结点比较之后都会插入到Vc的最后，这时需要的比较次数为：

$$C_{\max} = \sum_{0}^{m-1} n = m \square n$$

- 并运算中每次都是在Vc的最后插入，最好和最坏情况下的移动次数分别是n和m+n。

求集合的并和交

- 交运算的比较操作

$$C_{\min} = \sum_{0}^{m-1}(i+1) = \frac{m(m+1)}{2}$$

$$C_{\max} = \sum_{0}^{m-1} n = m\square n$$

- 交运算的移动操作，最好和最坏情况下的移动次数分别是0和m

约瑟夫问题

- 设n个人围成一个圆圈，按一指定的方向，从第s个人开始报数，报数到m为止，报数为m的人出列，然后从下一个人开始重新报数，报数为m的人又出列，如此反复，直到所有的人都出列为止。求n个人出列顺序。
- 约瑟夫问题的C语言实现

约瑟夫问题

```
void Josephus(Vector* P, int n, int s, int m)
{   int k=1, i, s1=s, j, w;
    for(i=0; i<n; i++) { Insert(P, k, i);  k++; }
    for(j=n; j>=1; j--)
    {   s1=(s1+m-1)%j;
        if(s1==0) s1=j;
        w=GetNode(P, s1-1);
        Remove(P, s1-1);
        Insert(P, w, n-1);
    }
}
```