第二章　线性表

例、学生健康情况登记表如下：

姓　名	学　号	性　别	年龄	健康情况
王小林	790631	男	18	健康
陈　红	790632	女	20	一般
刘建平	790633	男	21	健康
张立立	790634	男	17	一般
……	……	……	……	……

系统分析

- (1)逻辑结构的分析
- (2)系统操作(功能)的分析
- (3)存储结构
- (4)系统实现

逻辑结构的分析

例、学生健康情况登记表如下：

姓　名	学　号	性　别	年龄	健康情况
王小林	790631	男	18	健康
陈　红	790632	女	20	一般
刘建平	790633	男	21	健康
张立立	790634	男	17	一般
……	……	……	……	……

线性表的逻辑结构及其基本操作

线性表是n(n>=0)个相同类型数据元素$a_0, a_1, \ldots, a_{n-1}$构成的有限序列。

形式化定义：

Linearlist = (D, R)

其中：

$$D=\{a_i \mid a_i \in D_0, i=0,1,\cdots,n-1, n>0\}$$

$$R=\{N\}|, N=\{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D_0, i=1,2,\cdots,n-1\}$$

D_0为某个数据对象的集合

N为线性表长度

- 线性表的逻辑特征是：
  - 在非空的线性表，有且仅有一个开始结点a_1，它没有直接前趋，而仅有一个直接后继a_2；
  - 有且仅有一个终端结点a_n，它没有直接后继，而仅有一个直接前趋a_{n-1}；
  - 其余的内部结点$a_i(2 \leqq i \leqq n-1)$都有且仅有一个直接前趋a_{i-1}和一个直接后继a_{i+1}。

  线性表是一种典型的线性结构。
- 数据的运算是定义在逻辑结构上的，而运算的具体实现则是在存储结构上进行的。

系统操作(功能)的分析

系统操作(功能)的分析

- 1------新建学生健康表
- 2------向学生健康表插入学生信息
- 3------在健康表删除指定学生的信息
- 4------为某个学生修改身体状况信息（按学号操作）
- 5------按学生的学号排序并显示结果
- 6------在健康表中查询学生信息（按学生学号来进行查找）
- 7------在屏幕中输出全部学生信息
- 8------从文件中读取所有学生健康表信息
- 9------向文件写入所有学生健康表信息
- 10-----退出

线性表抽象数据类型

数据元素：a_i同属于一个数据元素类，i=1,2,……,n n≧0。

结构关系：对所有的数据元素a_i（i=1,2,……,n－1）存在次序关系＜a_i ,a_{i+1} ＞，
a_1无前驱，a_n无后继。

基本操作：对线性表可执行以下的基本操作

Initiate(L) 构造一个空的线性表L。

Length(L) 求长度。

Empty(L) 判空表。

Full(L) 判表满。

Clear(L) 清空操作。

Get(L,i) 取元素。

Locate(L,x) 定位操作。

Prior(L,data) 求前驱。

link(L,data) 求后继。

Insert(L,i,b) 插入操作（前插）。

Delete(L,i) 删除操作。

线性表的抽象类

```
template <class T >
class LinearList {
public:
   LinearList();                                  //构造函数
   ～LinearList();                               //析构函数
   virtual int Size() const = 0;  //求表最大体积
   virtual int Length() const = 0;        //求表长度
   virtual int Search(T x) const = 0;          //搜索
   virtual int Locate(int i) const = 0;        //定位
   virtual T* getData(int i) 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;  //复制
};
```

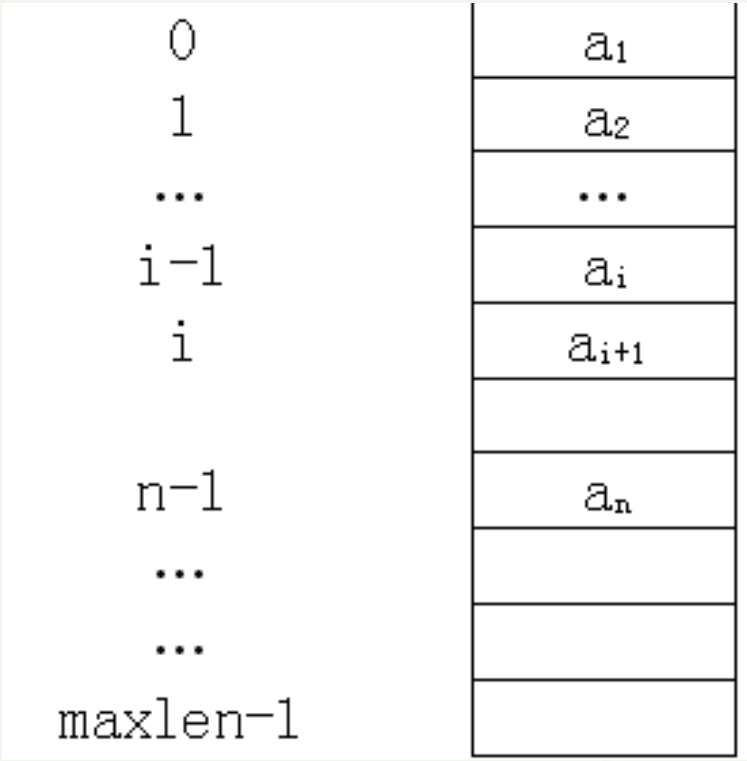
线性表在计算机中的实现

- 1.存储结构
- 2.系统操作(功能)的实现

线性表的顺序存储结构

- 顺序存储

定义：把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称顺序表。

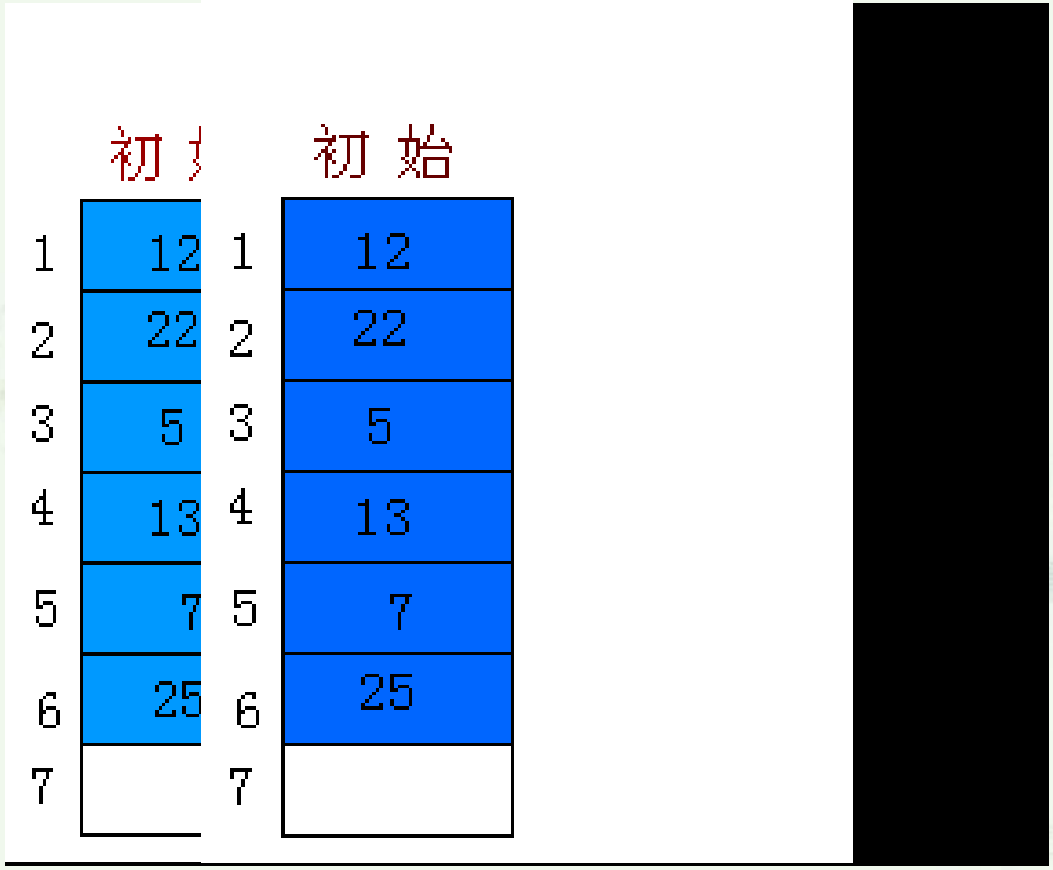


存储结构的可行性分析

- 系统操作(功能)实现过程的模拟
  - 插入
  - 删除
  - 查找
  - 排序
  - …

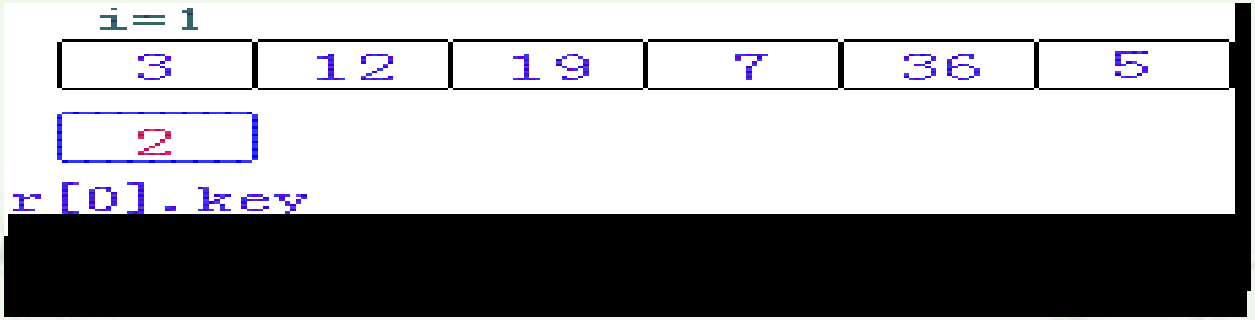
实现过程的模拟

- 删除操作



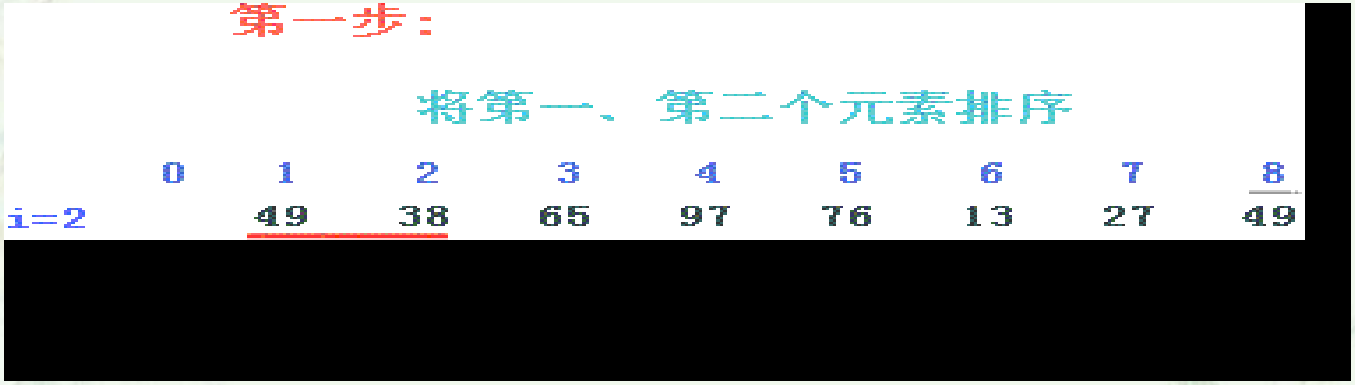
实现过程的模拟

- 查找操作



实现过程的模拟

- 排序操作



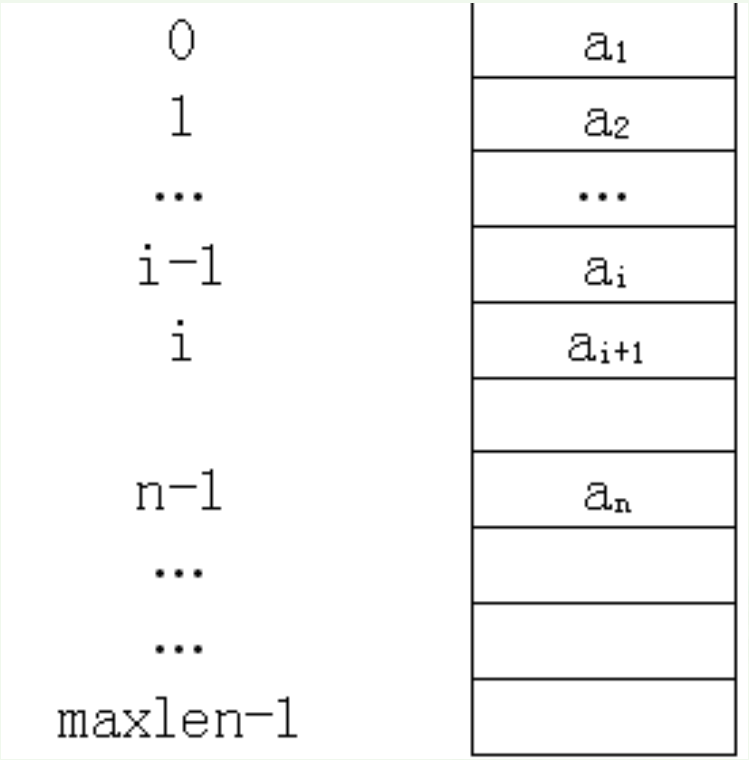
线性表在计算机中的实现

- 1.存储结构
- 2.系统操作(功能)的实现

顺序存储类

- 存储空间：用数组实现
- 属性：....

  操作：....

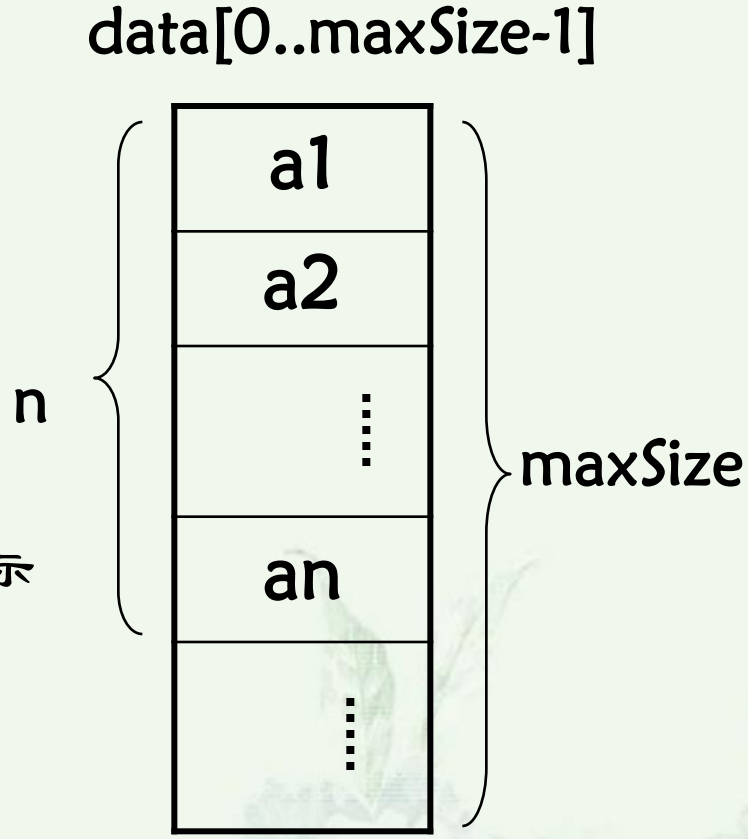


结构类型的描述

```
#define maxSize 100
//const int maxSize=最大容量；

Typedef  int T;
typedef struct {
    T data[maxSize]; //顺序表的静态存储表示
    int n;  //int last;
} SeqList;

typedef int T;
typedef struct {
    T *data;                    //顺序表的动态存储表示
    int maxSize;
    int  n;  // int last;
} SeqList;
```



SeqList La,Lb;
La.data[0]表示线性表的第一个元素，
La.n 则表示线性表的当前元素个数

- 顺序表上实现的基本操作

在顺序表存储结构中，很容易实现线性表的一些操作，如线性表的构造、第i个元素的访问。

注意：C语言中的数组下标从“0”开始，因此，若La是SeqList类型的顺序表，则表中第i个元素是La.data[i-1] 。

顺序表(SeqList)类的定义

```
const int defaultSize = 100;
template <class T>
class SeqList: public LinearList<T> {
protected:
    T *data;                  //存放数组
    int maxSize;              //最大可容纳表项的项数
    int last;                 //数组中最后一个元素的下标
    void reSize(int newSize);    //改变数组空间大小
```

P47

```
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 getData(int i, T & x);  //取第i个元素
    bool Insert(int i, T &x);                   //插入
    bool Remove(int i, T& x);               //删除
    ……
};
```

顺序表的构造函数

```
template <class T>
SeqList<T>::SeqList(int sz) {
    if (sz > 0) {
        maxSize = sz;  last = -1;
        data = new T[maxSize];      //创建表存储数组
        if (data == 0)              //动态分配失败
            { cerr << "存储分配错误！ " << endl;
              exit(1); }
    }
};
```

复制构造函数

```
template <class T>
SeqList<T>::SeqList ( SeqList<T>& L ) {
    T value;
    maxSize = L.Size();   last = L.Length()-1;
    data = new T[maxSize];        //创建存储数组
    if (data == 0)                //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;  }
};
```

查找

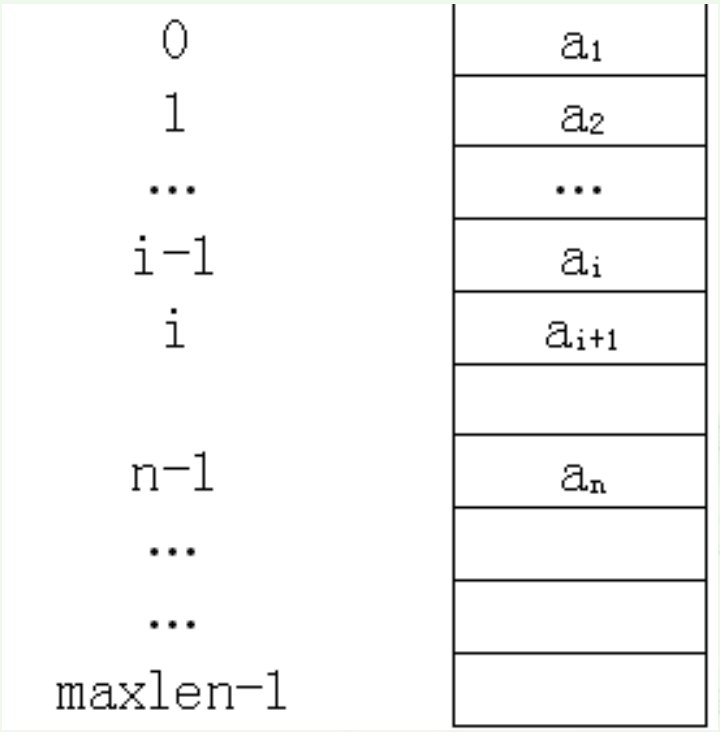
(1)按序号查找

(2)按值查找

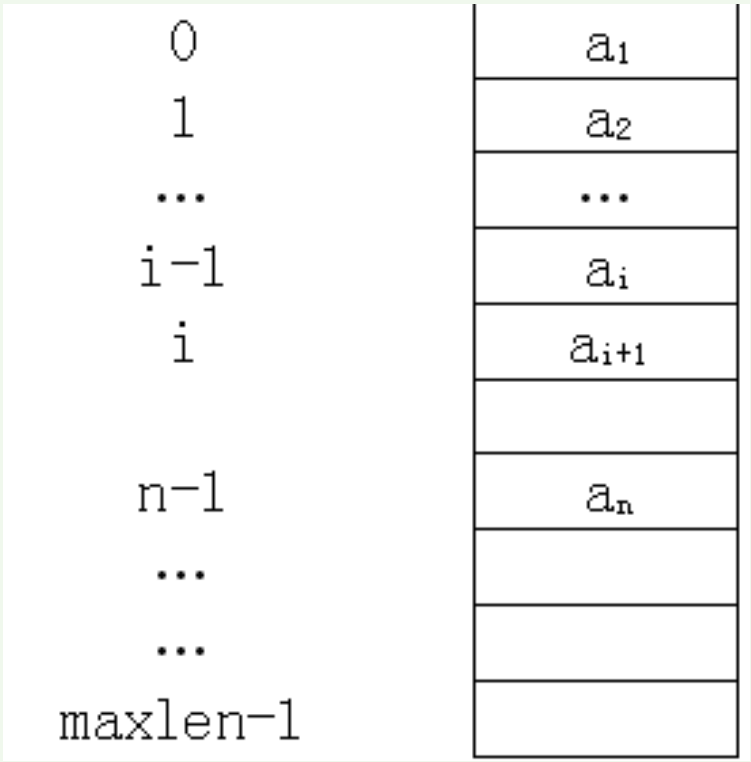
按序号查找

int SeqList<T>:: Locate(**int** i) **const**; // 1≦i≦n
//定位第 i 个表项，函数返回表项序号

bool SeqList<T>:: **getData(int i, T & x);** // 1≦i≦n
//取第i个元素



按值查找



顺序表的按值查找算法

```
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 = 1; i <=last+1; i++)                    //顺序搜索
        if ( data[i-1] == x ) return i;
                                //表项序号和表项位置差1
    return 0;                   //搜索失败
};
```

查找算法的分析

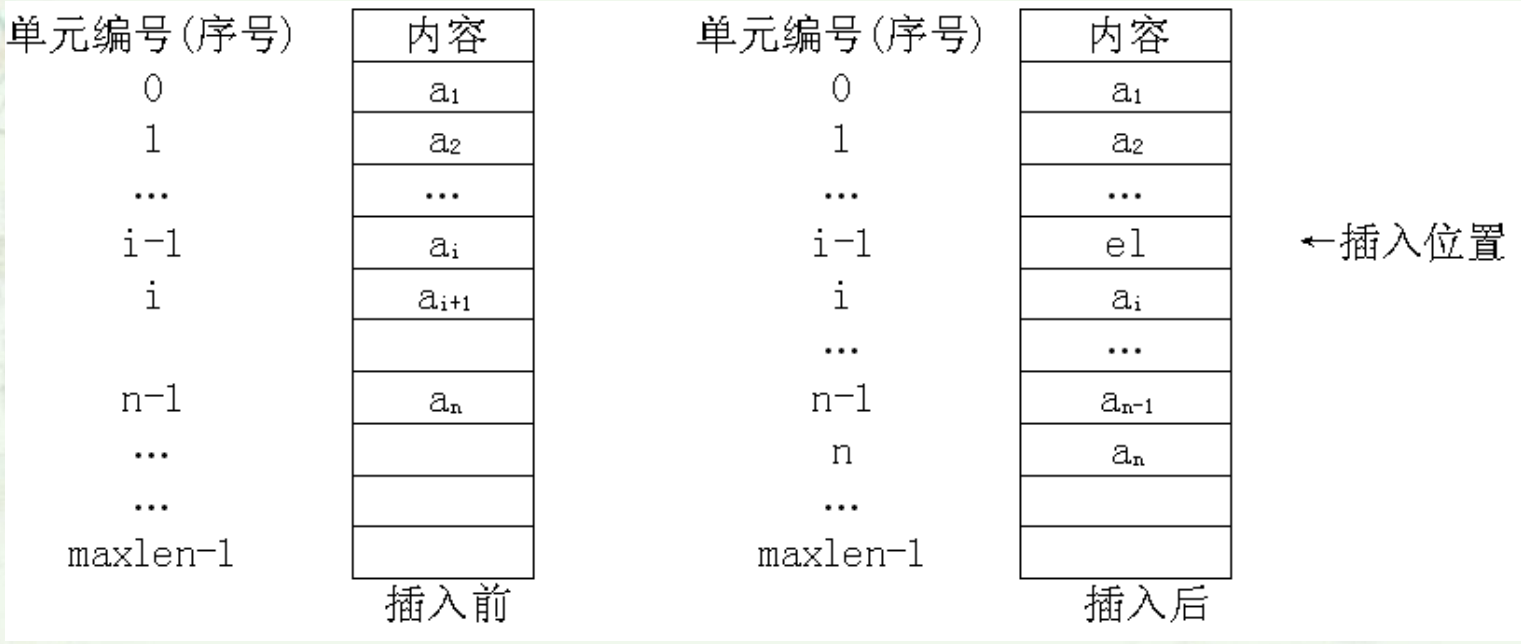
插入

线性表的插入运算是指在表的第i(1≦i≦n+1个位置上，插入一个新结点el，使长度为n的线性表

$(a_1, \ldots a_{i-1}, a_i, \ldots, a_n)$

变成长度为n+1的线性表

$(a_1, \ldots a_{i-1}, el, a_i, \ldots, a_n)$



说明：即在顺序表的第i-1个数据元素与第i个数据元素之间插入一个新的元素。

插入算法的步骤是：

(1)检查线性表是否还有剩余空间可以插入元素，若已满，则进行“溢出”的错误处理；

(2)检查i是否满足条件1≤i≤n+1，若不满足，则进行“位置不合法”的错误处理；

(3)将线性表的第i个元素以及其后面的所有元素均向后移动一个位置，以便腾出第i个空位置来存放新元素；

(4)将新元素el填入第i个空位置上；

(5)把线性表的长度增加1。

插入算法

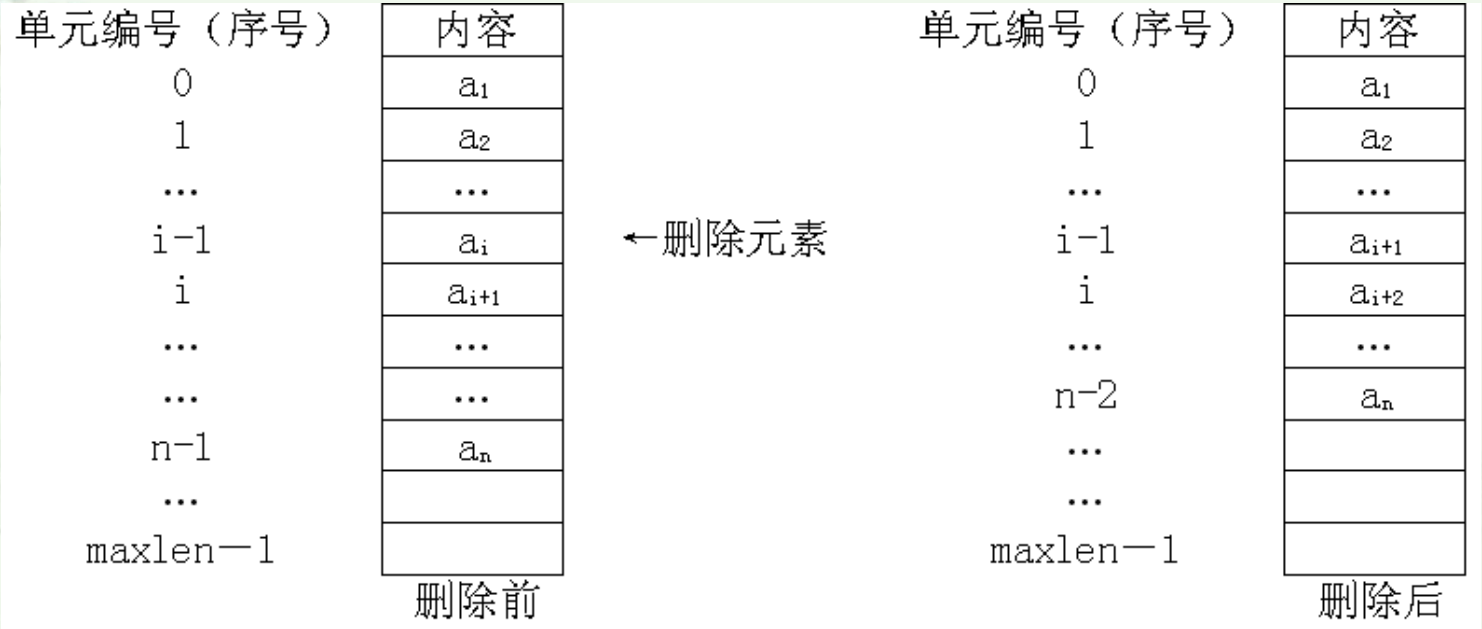
```
template <class T>
bool SeqList<T>::Insert (int i, T& x) {
//将新元素x插入到表中第i (1≤i≤n+1) 个表项位
//置。函数返回插入成功的信息
    if (last == maxSize-1) return false;        //表满
    if (i < 1 || i > Length()+1) return false;//参数i不合理
    for (int j = last+1; j >= i; j--)              //依次后移
        data[j] = data[j-1];
    data[i-1] = x;          //插入(第 i 表项在data[i-1]处)
    last++;   return true;                        //插入成功
};
```

• 分析算法的复杂度

这里的问题规模是表的长度，设它的值为n。

删除

线性表的删除运算是指将表的第i (1≦i≦n)结点删除，使长度为n的线性表：(a_1，…a_{i-1}，a_i，a_{i+1}…，a_n) 变成长度为n-1的线性表(a_1，…a_{i-1}，a_{i+1}，…，a_n)



删除算法的步骤为：

(1)检查i是否满足条件1≤i≤n，若不满足，则进行“位置不合法”的错误处理；

(2)将线性表中的第i个元素后面的所有元素均向前移动一个位置；

(3)把线性表的长度减少1。

删除算法

```
template <class T>
bool SeqList<T>::Remove (int i, T& x) {
//从表中删除第 i (1≤i≤n) 个表项，通过引用型参
//数 x 返回被删元素。函数返回删除成功信息
    if (last == -1) return false;              //表空
    if (i < 1 || i > last+1) return false;  //参数i不合理
    x = data[i−1];
    for (int j = i; j <= last; j++)      //依次前移，填补
        data[j−1] = data[j];
    last−−;   return true;
};
```

分析算法的时间复杂度

顺序表类的应用

P47

利用类的成员函数来解决问题

```
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 getData(int i, T & x);  //取第i个元素
    bool Insert(int i, T &x);                   //插入
    bool Remove(int i, T& x);               //删除
    ……
};
```

顺序表的应用：集合的“并”运算

顺序表的应用：集合的“并”运算

```
void Union ( SeqList<int>& LA,
                        SeqList<int>& LB ) {
    int n1 = LA.Length ( ), n2 = LB.Length ( );
    int i, k, x;
    for ( i = 1; i <= n2; i++ ) {
        LB.getData(i,x);              //在LB中取一元素
        k = LA.Search(x);             //在LA中搜索它
        if (k == 0)             //若在LA中未找到插入它
          { LA.Insert(n1+1, x);  n1++; }
                                //插入到第n个表项位置}
}
```

顺序表的应用：集合的“交”运算

顺序表的应用：集合的“交”运算

```
void Intersection ( SeqList<int> & LA,
                                SeqList<int> & LB ) {
   int n1 = LA.Length ( );
   int x, k, i = 1;
   while ( i <= n1 ) {
      LA.getData(i,x);      //在LA中取一元素
      k = LB.Search(x);     //在LB中搜索它
      if (k == 0)           //若在LB中未找到
         { LA.Remove(i, x); n1--; } //在LA中删除它
      else i++;             //未找到在A中删除它
   }
}
```

完整系统的运行逻辑

顺序表的优点：

✓无须为表示节点间的逻辑关系而增加额外的存储空间

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

✓可以方便的随机存取表中的任一节点

顺序表的缺点

✓插入和删除运算不方便

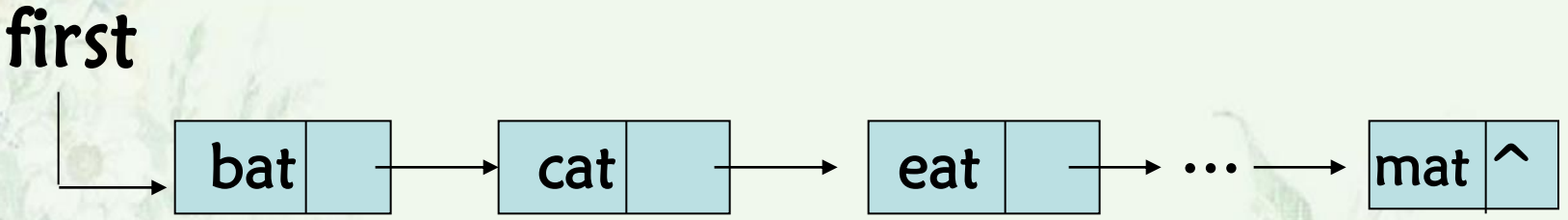
✓由于要求占用连续的存储空间，存储分配只能预先进行

例、学生健康情况登记表如下：

姓　名	学　号	性　别	年龄	健康情况
王小林	790631	男	18	健康
陈　红	790632	女	20	一般
刘建平	790633	男	21	健康
张立立	790634	男	17	一般
……	……	……	……	……

线性表的链式存储和实现

线性表:(bat，cat，eat，fat，hat，jat，lat，mat)

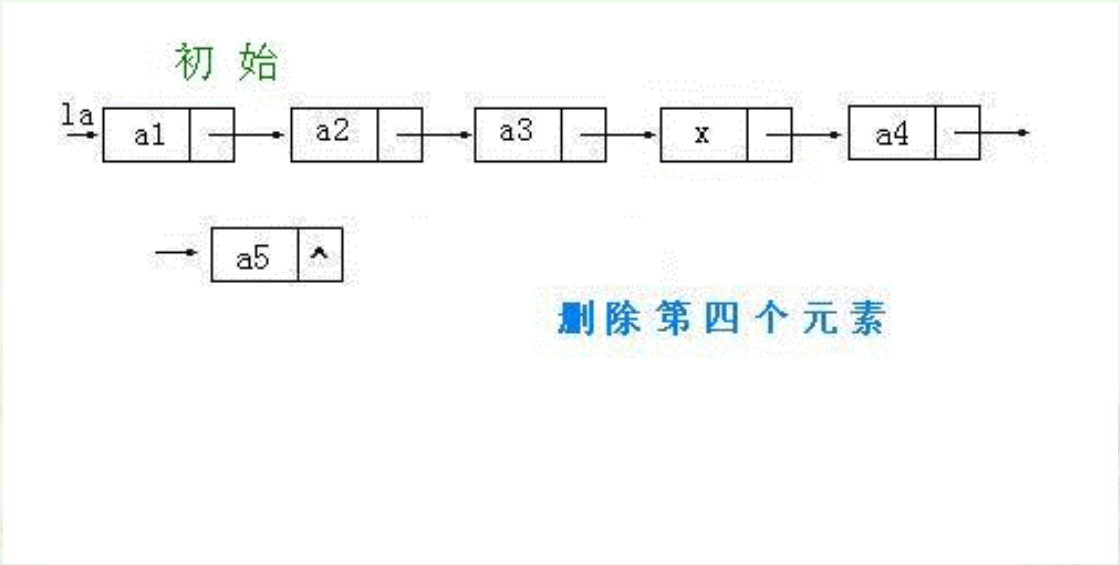


链表的可行性分析

- 系统操作(功能)实现过程的模拟
  - 插入
  - 删除
  - 查找
  - 排序
  - …

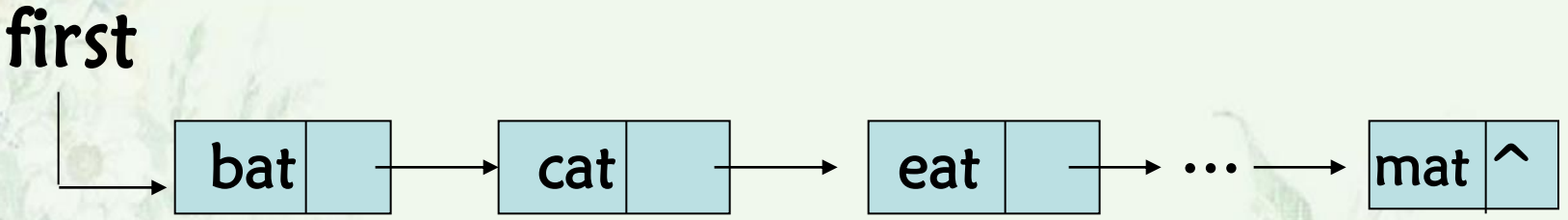
实现过程的模拟

- 删除操作

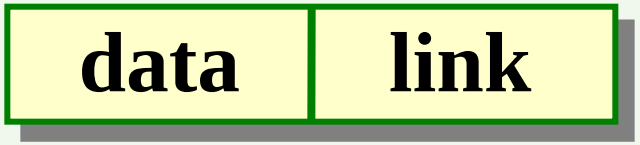


线性表的链式存储和实现

线性表:(bat，cat，eat，fat，hat，jat，lat，mat)



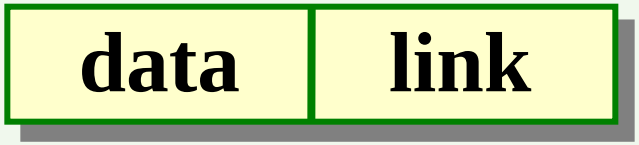
链式结构中结点结构的抽象



链表是指用一组任意的存储单元来依次存放线性表的结点，这组存储单元即可以是连续的，也可以是不连续的，甚至是随机分布在内存中的任意位置上的。

即链表中结点的逻辑次序和物理次序不一定相同。

C++语言实现链表结点的结构

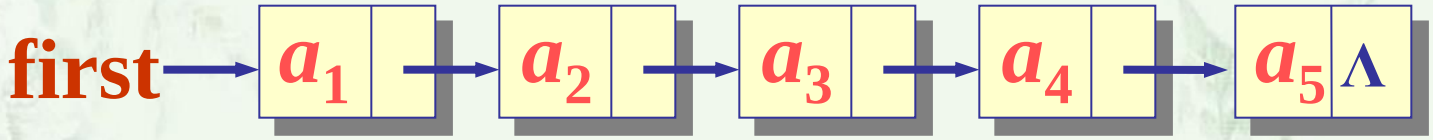



```
struct LinkNode {    //链表结点类
    int data;       // T data;
    LinkNode * link;
};
```

单链表 (Singly Linked List)

- 线性结构

  结点之间可以连续，可以不连续存储
  - 结点的逻辑顺序与物理顺序可以不一致
  - 表可扩充



单链表类的定义

- 属性：反映链表的重要特性(即链头)
- 操作：线性表操作的抽象及综合(查找、插入、删除等)

```
class List {
 //链表类, 直接使用链表结点类的数据和操作
 private:
    LinkNode *first;          //表头指针
   …..
 public:
    ….
 };
```

与链表有关的基本操作

(1)结点指针的定义　LinkNode<T> *p;

(2)结点存储空间的申请

new 方法

p=new LinkNode <T>( );

(3)空间的释放

delete方法

delete p

(4)结点中域的访问

p->data=x;

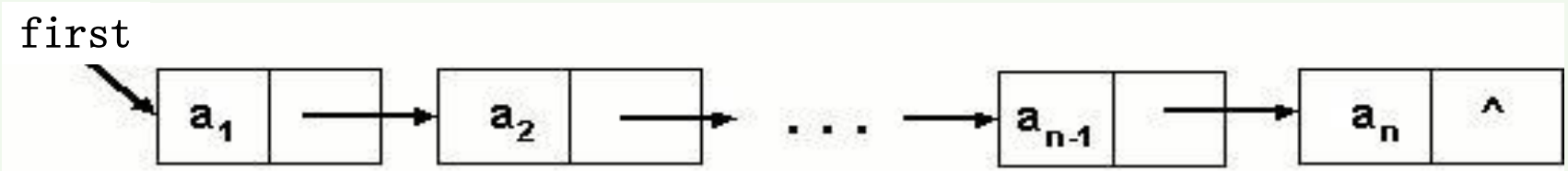
p->link=q; (q是结点指针)

(5)空指针 NULL或0

p->link=0;

q=0;

单链表的建立



1、头插入建表

该方法从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束为止。

```
template <class T>
List<T>:: HLinkList (int n)
{
    first=0;
    for (i=0;i<n;i++)
    {   p=new LinkNode <T>();
        cin>>p->data;
        p->link=first;
        first=p;
     }
}
```

单链表的建立

2、尾插入建表

头插法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。

该方法是将新结点插入到当前链表的表尾上，为此必须增加一个尾指针r，使其始终指向当前链表的尾结点。

```
template <class T>
List<T>:: RLinkList (int n  )
 {   first=0;tail=0;
     for(i=0;i<n;i++)
     {  p=new LinkNode <T>( );
        cin>>p–>data;
         p->link=0;
         if(first==0)
             first=p;
         else
             tail->link=p;
        tail=p;
     }
}
```

该算法的缺陷是不如头插入算法那么简洁

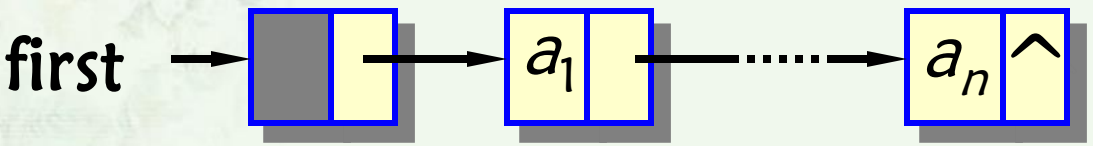
尾插入建链改进算法

```
template <class T>
List<T>:: RLinkList (int n  )
 {
   first=tail=new LinkNode <T>( ); first->link=0;
   for(i=0;i<n;i++)
   {  p=new Node<T>( );   cin>>p–>data;
      p->link=0;
      tail->link=p;
      tail=p;
    }
}
```

引入表头结点

带表头结点的单链表

- 表头结点位于表的最前端，本身不带数据，仅标志表头。
- 设置表头结点的目的是
  - 统一空表与非空表的操作
  - 简化链表操作的实现。



非空表

空表

单链表的模板类

- 类模板将类的数据成员和成员函数设计得更完整、更灵活。
- 类模板更易于复用。
- 在单链表的类模板定义中，增加了表头结点。

用模板定义的结点类

```
template <class T>
struct LinkNode {                    //链表结点类的定义
    T data;                          //数据域
    LinkNode<T> * link;              //链指针域
    LinkNode() { link = 0; }         //构造函数
    LinkNode(T item, LinkNode<T> * ptr = 0)
        { data = item;  link = ptr; }    //构造函数
    bool operator== (T x) { return data.key == x; }
        //重载函数，判相等
    bool operator != (T x) { return data.key != x; }
};
```

用模板定义的单链表类

P60

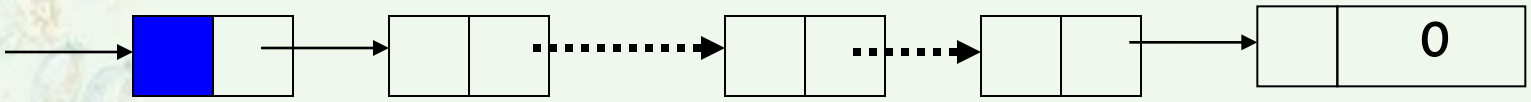
```
template <class T>
class List {
//单链表类定义, 不用继承也可实现
protected:
    LinkNode<T> * first;  //表头指针
public:
    List() { first = new LinkNode<T>; } //构造函数
    List(T x) { first = new LinkNode<T>(x); }
    List( List<T>& L);              //复制构造函数
    ~List(){ }                      //析构函数
    void makeEmpty();               //将链表置为空表
    int Length() const;             //计算链表的长度
```

```
    LinkNode<T> *Search(T x); //搜索含x元素
    LinkNode<T> *Locate(int i);          //定位第i个元素
    T *getData(int i);                   //取出第i元素值
    void setData(int i, T x);            //更新第i元素值
    bool Insert (int i, T x);           //在第i元素后插入
    bool Remove(int i, T& x);           //删除第i个元素
    bool IsEmpty() const                //判表空否
      { return first->link == 0 ? true : false; }
    LinkNode<T> *getFirst() const { return first; }
    void setFirst(LinkNode<T> *f ) { first = f; }
    void Sort();                        //排序
    void Print();   //输出整条链表的结点值
};
```

输出整条链表的结点值——遍历

void print();

功能：输出链表中的所有结点值。



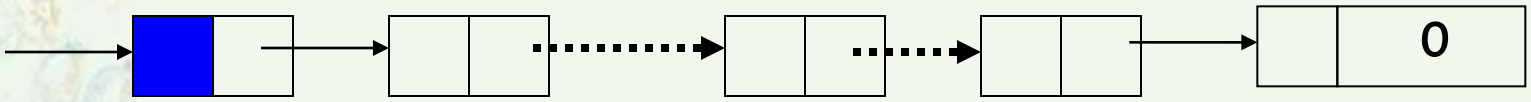
输出单链表所有结点值的算法

```
template <class T>
Void List<T> :: Print ( )
{
    LinkNode<T> *p = first->link;
    //检测指针 p 指示第1个结点
    while ( p != 0 )
     {     //逐个结点检测
         cout<<p->data; p = p->link;
     }
}
```

求长度操作

int length( );

功能：求单链表的长度。



改写输出单链表所有结点值的算法

```
template <class T>
int List<T> :: Length ( ) const
{
    LinkNode<T> *p = first->link;
    //检测指针 p 指示第1个结点
    int count=0;
    while ( p != 0 )
    {    //逐个结点检测
        count++;      p = p->link;
    }
    return count;
}
```

求单链表长度的算法

```
template <class T>
int List<T> :: Length ( ) const
{
    LinkNode<T> *p = first->link;
    //检测指针 p 指示第1个结点
    int count = 0;
    while ( p != 0 )
    {     //逐个结点检测
        count++;    p = p->link;
    }
    return count;
}
```

算法分析

查找操作

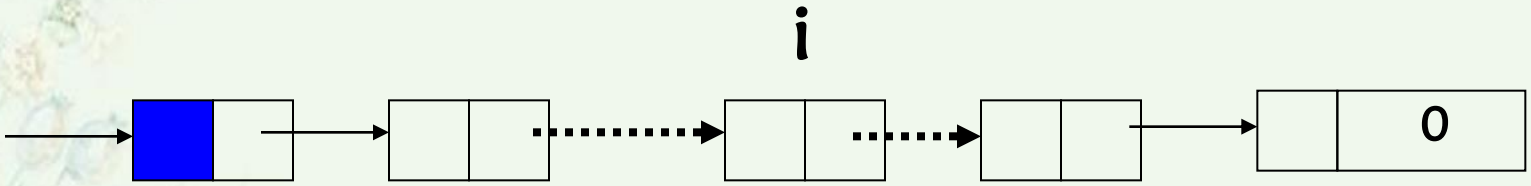
- 按序号查找——单链表找第i个元素——定位

- 按值查找——搜索

定位操作

LinkNode<T> * locate(int i)

功能：求单链表第i个结点。



改写求单链表长度的算法

```
template <class T>
LinkNode<T> * List<T>::Locate ( int i )
{
    LinkNode<T> *p = first->link;
    //检测指针 p 指示第1个结点
    int count = 0;
    while ( p != 0 )
    {    //逐个结点检测
        count++;   if(count==i) return p;
        p = p->link;
    }
   return 0;
}
```

单链表的定位算法

```
template <class T>
LinkNode<T> * List<T>::Locate ( int i ) {
//函数返回表中第 i 个元素的地址。若i < 0或 i 超
//出表中结点个数，则返回0。
    if (i < 0) return 0;                    //i不合理
    LinkNode<T> * current = first;  int k = 0;
    while ( current != 0 && k < i )
      { current = current->link;  k++; }
    return current;      //返回第 i 号结点地址或0
};
```

算法的分析

假设当前链表中的结点总数为n个：

1）最好情况：当然是要找的结点序号为1时，此时只需要比较2次即可。

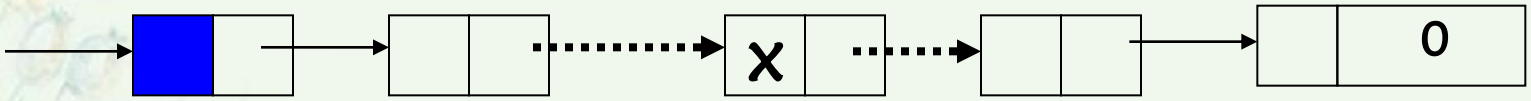
2）最坏情况：当要找的结点序号大于或等于n时，比较次数最多，共有2n次。

3）平均情况：假设在链表中查找各结点的概率相等（即等于1/n时），则其平均的比较次数为2*1/n*(1+2+...+n)=n+1,所以该算法的平均时间复杂度为O(n)。

搜索操作

LinkNode<T> * ::Search(T x)

功能：在单链表中搜索值为x的结点。



改写输出单链表所有结点值的算法

```
template <class T>
LinkNode<T> *  List<T>::Search(T x)
{
    LinkNode<T> *p = first->link;
    //检测指针 p 指示第1个结点
    while ( p != 0 )
    {     //逐个结点检测
        if (p->data==x) return p;
        p = p->link;
    }
    return 0;
}
```

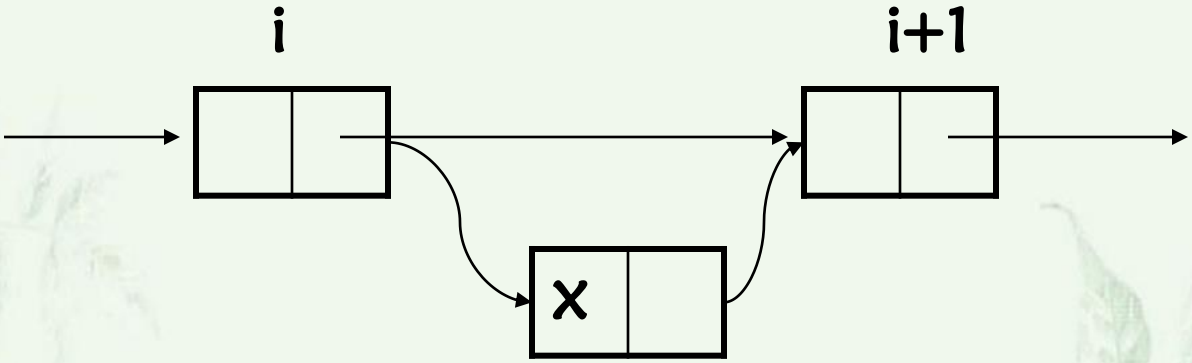
单链表的搜索算法

```
template <class T>
LinkNode<T> *  List<T>::Search(T x) {
//在表中搜索含数据x的结点, 搜索成功时函数返
//该结点地址; 否则返回0。
    LinkNode<T> * current = first->link;
    while ( current != 0 && current->data != x )
        current = current->link;
        //沿着链找含x结点
    return current;
};
```

算法分析

单链表插入操作

- 含义：在线性表L的第i个结点后插入一个指定元素值的新结点。



插入操作表示形式

bool List<T>**::**Insert (**int** i, T x)

参数i、x分别表示插入的位置与插入的元素。

该函数的功能为：在带头结点的单链表中的第i个结点之后插入数据元素值为x的新结点.

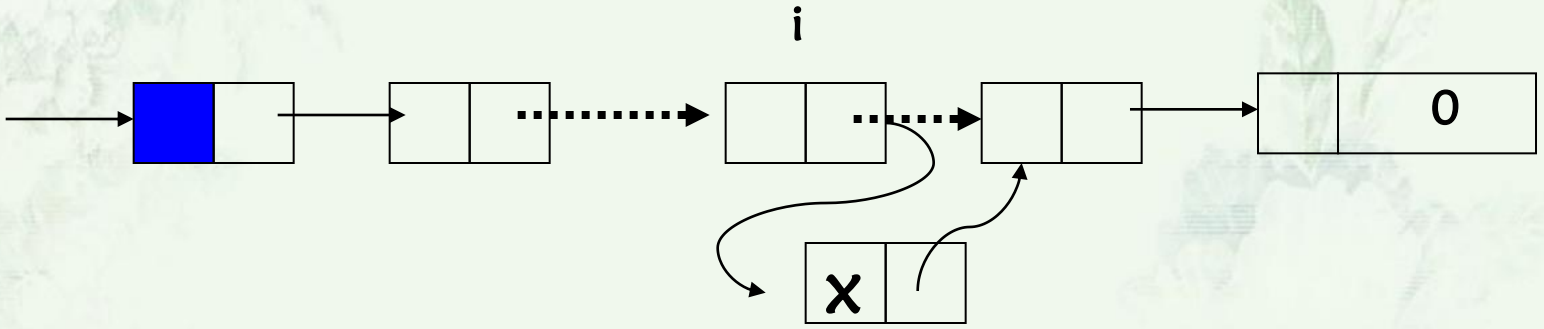
插入操作

- 处理过程：

(1) 寻找第i个结点，使指针p指向该结点；

(2) 若由于i不合理而找不到相应的结点，则输出信息，否则：

(3) 生成一个新结点s，并将s插入到结点p之后。



单链表的插入算法

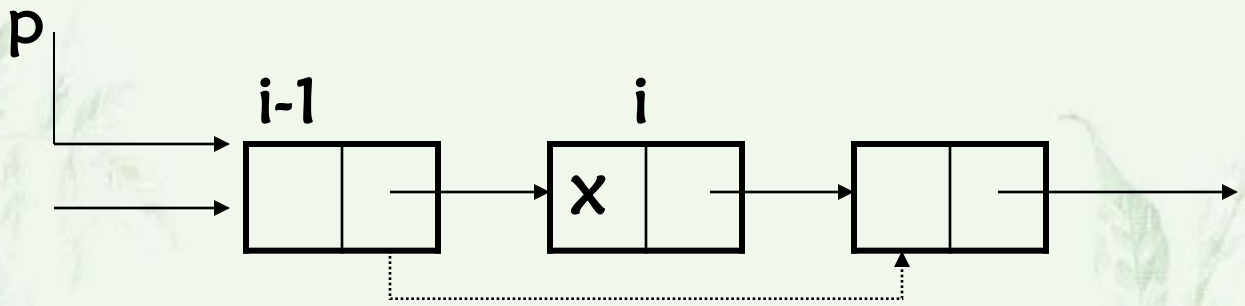
```
template <class T>
bool List<T>::Insert (int i, T x) {
//将新元素 x 插入在链表中第 i 个结点之后。
   LinkNode<T> *current = Locate(i);
   if (current == 0) return false;          //无插入位置
   LinkNode<T> *newNode=new  LinkNode<T>(x);
     //创建新结点
   newNode->link = current->link;          //链入
   current->link = newNode;
   return true;                                    //插入成功
};
```

分析算法的时间复杂度

该算法的执行时间与插入点所对应的位置有关，假设当前链中的结点总数为n个且在各结点之前插入的概率相等，则平均时间复杂度为O(n)。

单链表删除操作

- 含义：线性链表的删除操作是指删除线性链表中的第i号结点。



p->link=p->link->link;

删除操作表示形式

bool List<T>**::**Remove (**int** i, T**&** x)

表示，其功能为在单链表中删除第i个结点并返回该结点中的元素值，该操作的处理过程为：

(1)寻找第i号结点，使指针p指向该结点的前驱结点。

(2)若由于i不合理而找不到相应的结点，则返回0，否则：

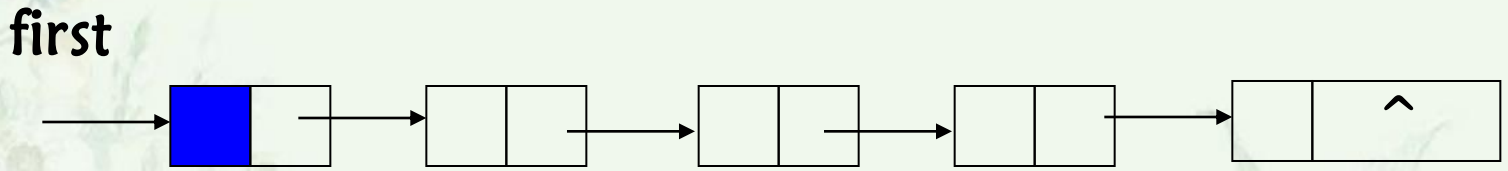
(3)改变p的指针域，使得第i号结点从链表中被删除，释放该结点并通过X带回该结点中的元素值。

单链表的删除算法

```
template <class T>
bool List<T>::Remove (int i, T& x ) {
//删除链表第i个元素,通过引用参数x返回元素值
    LinkNode<T> *current = Locate(i-1);
    if ( current == 0 || current->link == 0)
        return false;          //删除不成功
    LinkNode<T> *del = current->link;
    current->link = del->link;
    x = del->data;   delete del;
    return true;
};
```

析构函数

即意味着把当前链表中所有结点占用的存储空间进行释放。因此算法可描述如下：



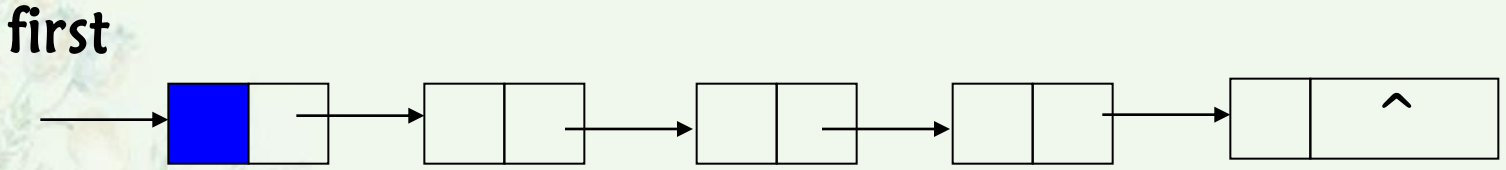
析构函数

即意味着把当前链表中所有结点占用的存储空间进行释放。因此算法可描述如下：

```
template <class T>

List<T>::~List() {
    LinkNode<T> *q;
    while (first != 0)
    {
        q = first;              //保存被删结点
        first = first->link;    //从链上摘下该结点
        delete q;               //删除
    }
};
```

其他形式的链式结构



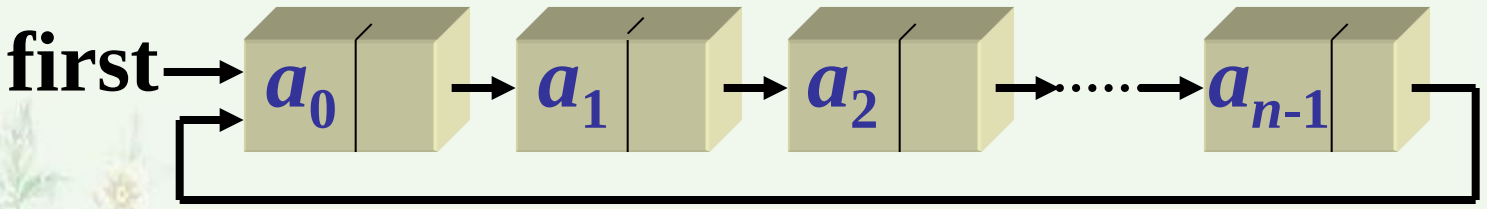
问题思考

例、在链表上实现将两个线性表

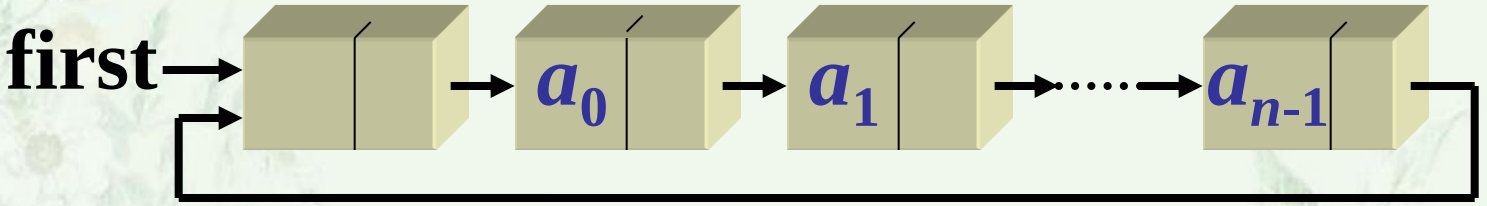
$(a_1, a_2, a_3, \ldots a_n)$和$(b_1, b_2, b3, \ldots b_n)$

链接成一个线性表的运算。

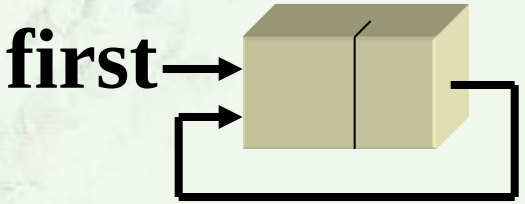
循环链表



- 带表头结点的循环链表



（非空表）



（空表）

- 循环链表是单链表的变形。
- 循环链表最后一个结点的 link或link指针不 为0，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入表头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。
- 实际中多采用尾指针表示单循环链表。

循环链表类的定义

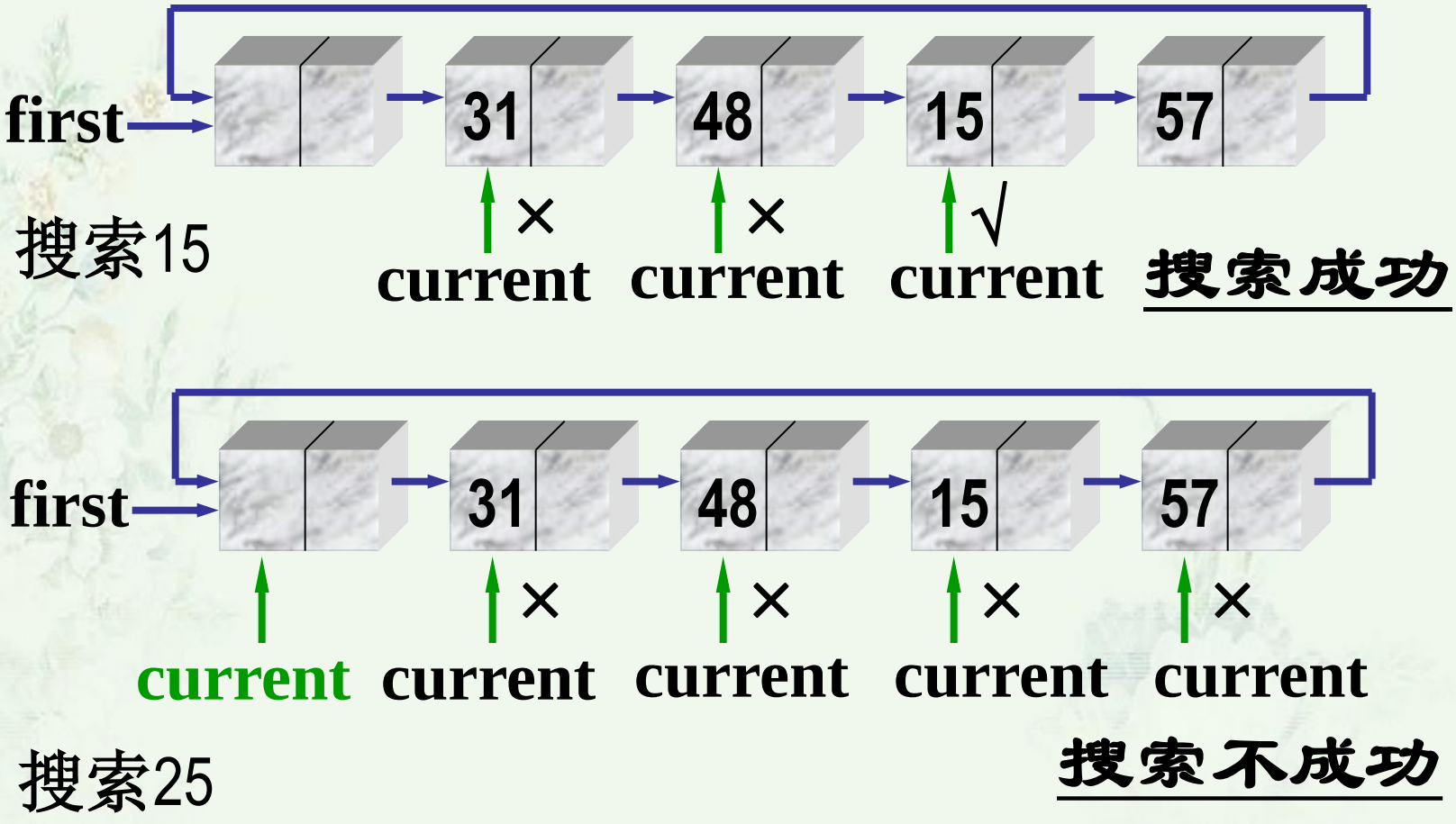
```
template <class T>
struct CircLinkNode {                    //链表结点类定义
    T data;
    CircLinkNode<T> *link;
    CircLinkNode ( CircLinkNode<T> * next =
          0 ) { link = next; }
    CircLinkNode ( T d, CircLinkNode<T> * next =
          0 ) { data = d;  link = next; }
    bool Operator==(T x) { return data.key == x.key; }
    bool Operator!=(T x) { return data.key != x.key; }
};
```

```
template <class T>      //链表类定义
class CircList {
private:
   CircLinkNode<T> *first, *last;  //头指针, 尾指针
public:
   CircList(const T x);                    //构造函数
   CircList(CircList<T>& L);       //复制构造函数
   ～CircList();                           //析构函数
   int Length() const;                     //计算链表长度
   bool IsEmpty() { return first->link == first; }
                                           //判表空否
   CircLinkNode<T> *getHead() const;
                                   //返回表头结点地址
```

```
    void setHead ( CircLinkNode<T> *p );
                                        //设置表头结点地址
    CircLinkNode<T> *Search ( T x ); //搜索
    CircLinkNode<T> *Locate ( int i );//定位
    T *getData ( int i );                          //提取
    void setData ( int i, T x );                   //修改
    bool Insert ( int i, T x );                    //插入
    bool Remove ( int i, T& x);                    //删除
};
```

- 循环链表与单链表的操作实现，最主要的不同就是扫描到链尾，遇到的不是0，而是表头。

循环链表的搜索算法


first
31
48
15
57
搜索15
×
×
√
current
current
current
搜索成功
first
31
48
15
57
current
current
current
current
current
搜索25
搜索不成功

循环链表的搜索算法

```
template <class T>
CircLinkNode<T> * CircList<T>::Search( T x )
{
//在链表中从头搜索其数据值为 x 的结点
    current = first->link;
    while ( current != first && current->data != x )
        current = current->link;
    return current;
}
```

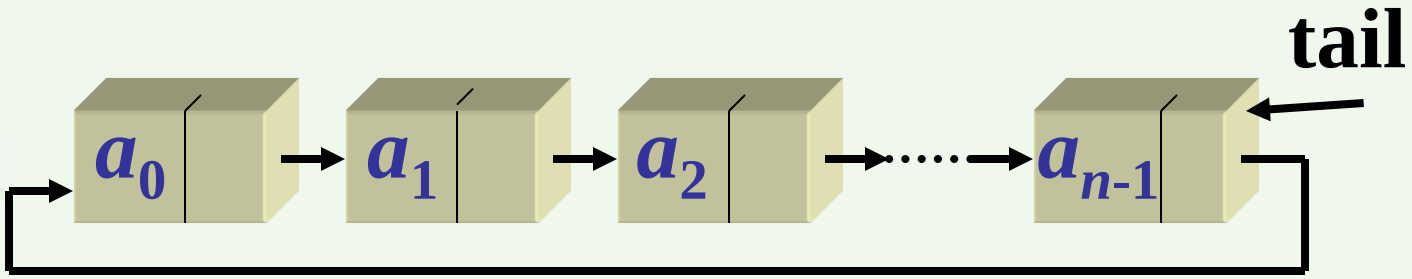
循环链表的优点

例、在链表上实现将两个线性表

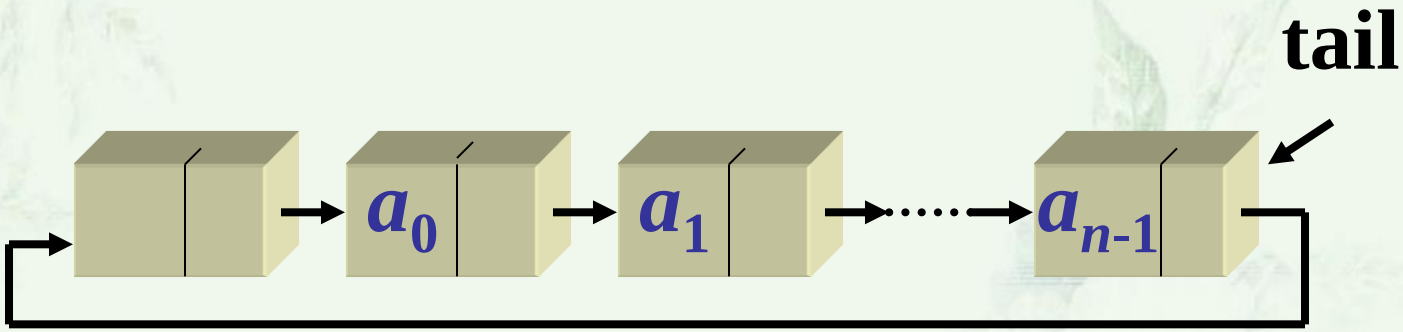
$(a_1, a_2, a_3, \ldots a_n)$和$(b_1, b_2, b3, \ldots b_n)$

链接成一个线性表的运算。

循环链表

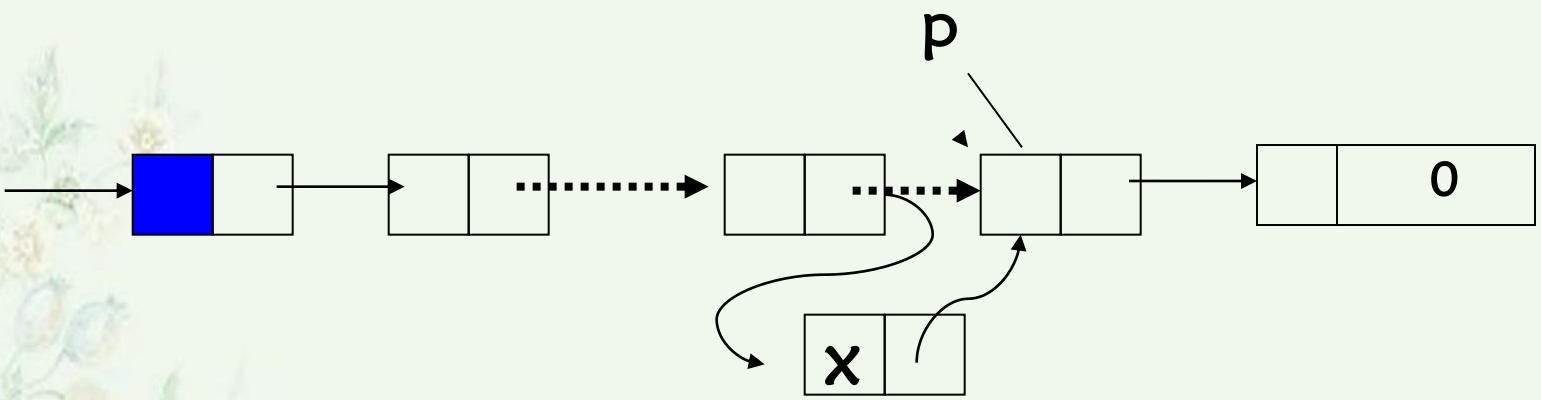


- 带表头结点的循环链表

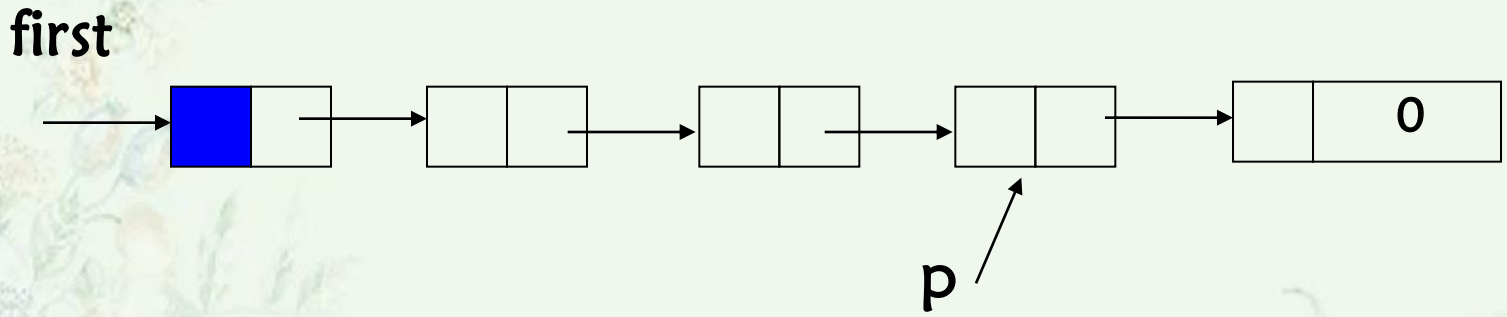


其他形式的链表

插入操作

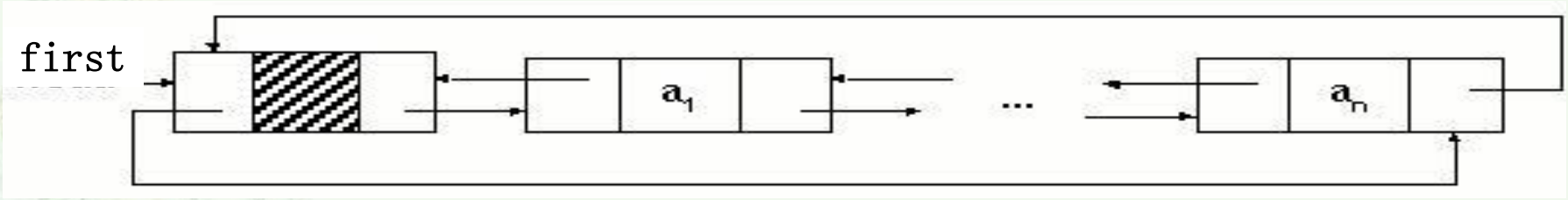


删除操作

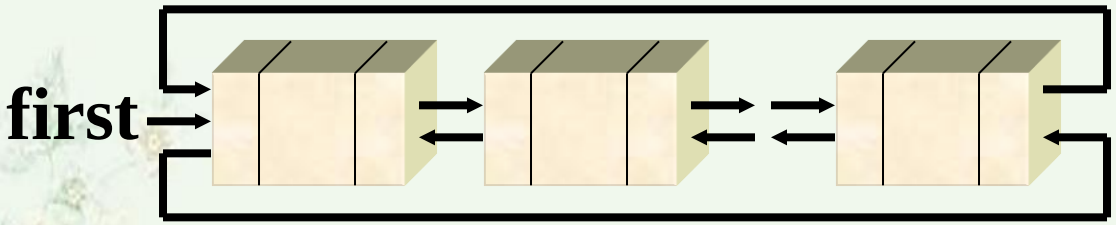


双向链表

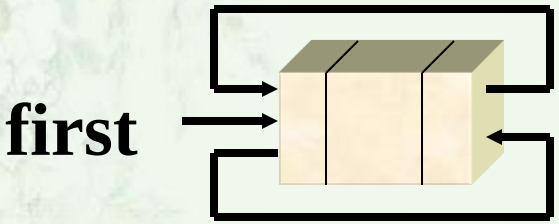
- 单链表及循环链表的缺陷



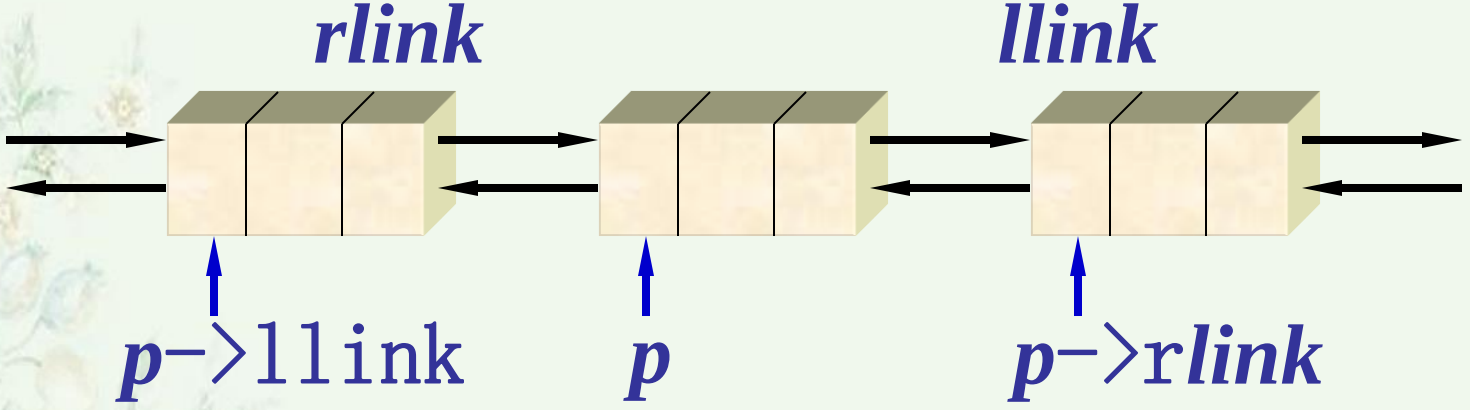
llink	data	rlink



非空表



空表



- 结点指向

p == p->llink->rlink == p->rlink->llink

双向循环链表类型定义

双向循环链表结点类的定义

```
template <class T>
struct DblNode {              //链表结点类定义
T data;                       //链表结点数据
DblNode<T> *lLink, *rLink;    //前驱、后继指针
DblNode ( DblNode<T> * l = 0,
   DblNode<T> * r = 0 )
     { lLink = l;  rLink = r; }        //构造函数
DblNode ( T value, DblNode<T> * l = 0, DblNode<T> *
   r = 0)
     { data = value;  lLink = l;  rLink = r; } //构造函数
};
```

双向循环链表类的定义

```
template <class T>
class DblList {                    //链表类定义
public:
DblList ( T uniqueVal ) {                  //构造函数
    first = new DblNode<T> (uniqueVal);
    first->rLink = first->lLink = first;
};
DblNode<T> *getFirst () const { return first; }
void setFirst ( DblNode<T> *ptr ) { first = ptr; }
DblNode<T> *Search ( T x, int d);
  //在链表中按d指示方向寻找等于给定值x的结点,
  //d=0按前驱方向,d≠0按后继方向
```

P70

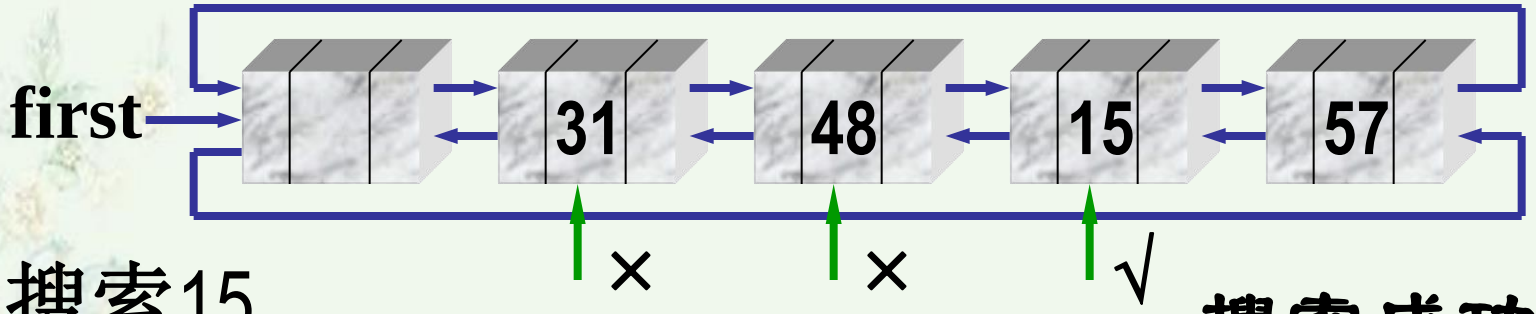
```
DblNode<T> *Locate ( int i, int d );
  //在链表中定位序号为i(≥0)的结点, d=0按前驱方
  //向,d≠0按后继方向
bool Insert ( int i, T x, int d );
  //在第i个结点后插入一个包含有值x的新结点,d=0
  //按前驱方向,d≠0按后继方向
bool Remove ( int i, T& x, int d );   //删除第i个结点
bool IsEmpty() { return first->rlink == first; }
  //判双链表空否
private:
    DblNode<T> *first;                  //表头指针
};
```

双向链表的操作特点：

“查询”和单链表相同

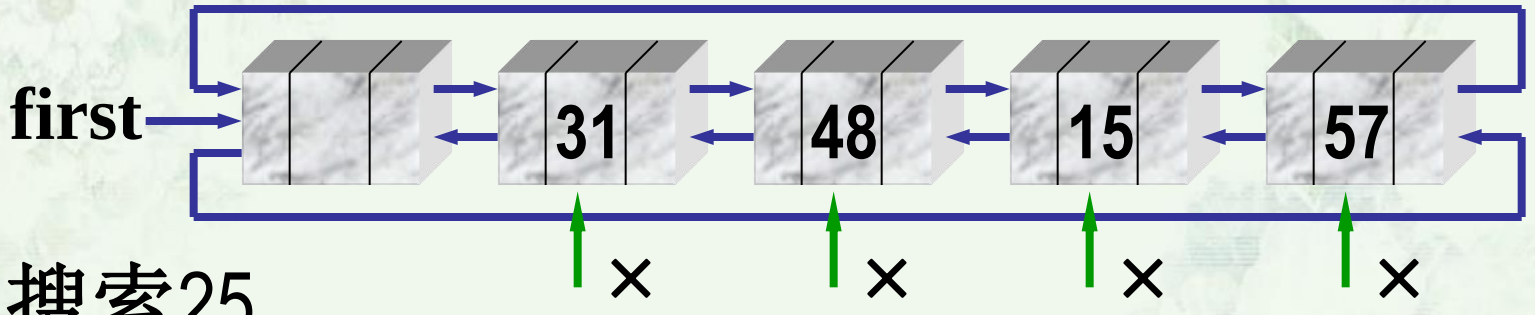
“建链”、“插入”和“删除”时需要同时修改两个方向上的指针。

双向循环链表的搜索



搜索15

搜索成功



搜索25

搜索不成功

双向循环链表的搜索算法

```
template <class T>
DblNode<T> *DblList<T>::Search ( T x , int d ) {
//在双向循环链表中寻找其值等于x的结点。
   DblNode<T> *current = (d == 0)?
        first->lLink : first->rLink;  //按d确定搜索方向
   while ( current != first && current->data != x )
        current = (d == 0) ?
            current->lLink : current->rLink;
   if ( current != first ) return current;          //搜索成功
   else return 0;                                   //搜索失败
};
```

双向循环链表插入操作

bool DblList<T>::Insert (int i, T x, int d)

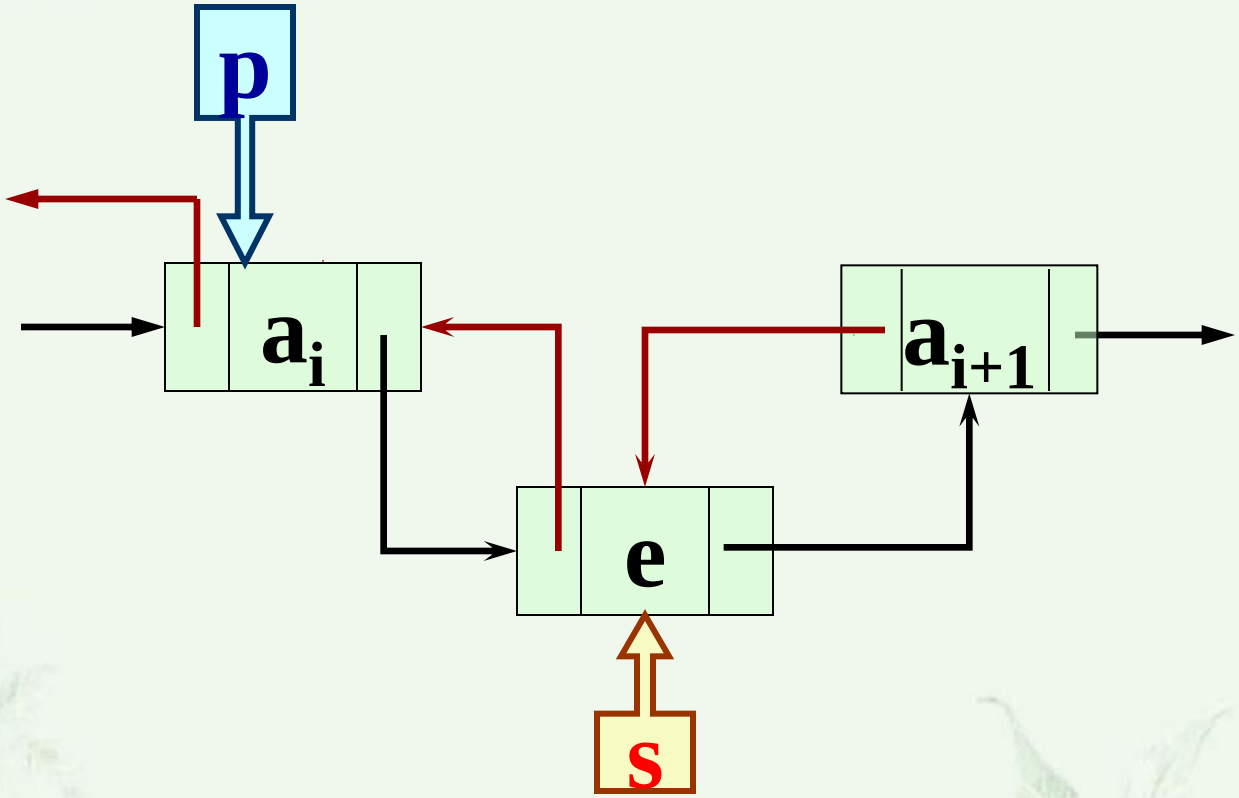
功能：在双向循环链表按d方向搜索第i个结点之后插入元素值为x的结点。

处理过程：

(1)由参数i和搜索方向d求得结点的指针p。

(2)若容许插入则生成一个元素值为x的新结点s，将由s所指向的结点插入到双向循环链表中的由p所指向的结点之后并返回true，否则返回false。

插入



s->rlink = p->rlink;　　**p->rlink = s;**

s->rlink->llink = s;　　**s->llink = p;**

双向循环链表的插入算法

```
template <class T>
bool DblList<T>::Insert ( int i, T x, int d ) {
//建立一个包含有值x的新结点，并将其按 d 指定的
//方向插入到第i个结点之后。
    DblNode<T> *current = Locate(i, d);
      //按d指示方向查找第i个结点
    if ( current == 0 ) return false;   //插入失败
    DblNode<T> *newNd = new DblNode<T>(x);
    if (d == 0) {     //前驱方向:插在第i个结点左侧
        newNd->lLink = current->lLink; //链入lLink链
        current->lLink = newNd;
```

```
        newNd->lLink->rLink = newNd; //链入rLink链
        newNd->rLink = current;
    } else {                    //后继方向:插在第i个结点后面
        newNd->rLink = current->rLink; //链入rLink链
        current->rLink = newNd;
        newNd->rLink->lLink = newNd;  //链入lLink链
        newNd->lLink = current;
    }
    return true;            //插入成功
};
```

双向循环链表删除操作

bool DblList<T>::Remove(int i, T& x, int d)

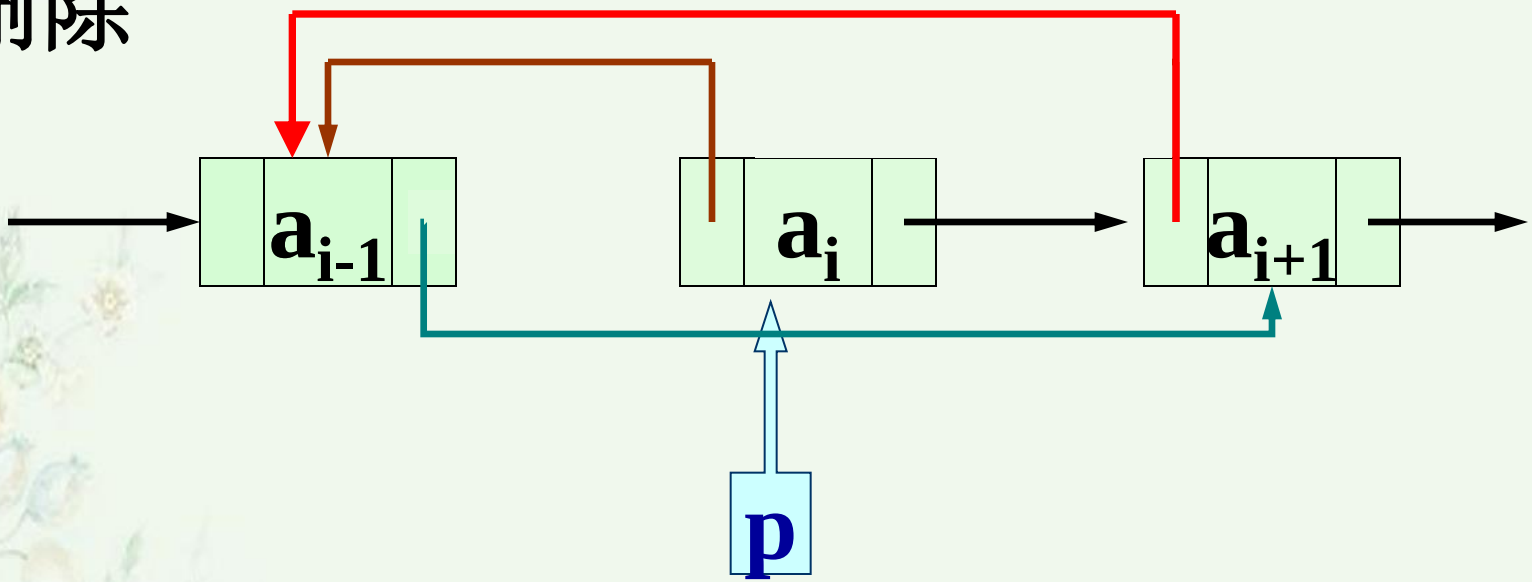
功能：删除双向循环链表中按d方向搜索的第i个结点并通过x带回该结点中的元素值。

处理过程：

(1)由参数i和搜索方向d求得结点的指针p。

(2)若第i个结点存在，则删除双向循环链表中的由p所指向的结点，释放该结点并带回该结点中的元素值否则返回0。

删除




```
p->llink->rlink = p->rlink;
p->rlink->llink = p->llink;
```

双向循环链表的删除算法

```
template <class T>
bool DblList<T>::Remove( int i, T& x, int d ) {
//在双向循环链表中按d所指方向删除第i个结点。
   DblNode<T> *current = Locate (i, d);
   if (current == 0) return false;          //删除失败
   current->rLink->lLink = current->lLink;
   current->lLink->rLink = current->rLink;
       //从lLink链和rLink链中摘下
   x = current->data;  delete current;          //删除
   return true;                                 //删除成功
};
```

链式存储结构的特点

链式存储结构的优点是：

（1）用指针来反映结点之间的逻辑关系，这样做插入、删除操作就只需修改相应结点的指针域即可完成，从而克服了顺序表中插入、删除需大量移动结点的缺点。

（2）结点空间可以动态申请和动态释放，这样就克服了顺序表中结点的最大数目需预先确定的缺点。

链式存储结构的特点

但链式存储结构同样也有不足：

（1）为了能够用指针来反映结点之间的逻辑关系，需要为每个结点额外增加相应的指针域，从而使结点的存储密度比顺序表中结点的存储密度要小。

（2）在链式存储结构中要查找某一结点，一般要从链头开始沿链进行扫描才能找到该结点，其平均时间复杂度为O(n)。因此，链式存储结构是一种非随机存储结构。

线性表的应用

- 例1:将一个线性表的元素逆置。

- （1）将一个顺序表中的元素逆置。

- （2）用单链表作存储结构，编写一个实现线性表中元素逆置的算法

P84(2.6)　　P86(2.17)

约瑟夫问题

所谓约瑟夫(Josephus)问题指的是假设有 n 个人围坐一圈，先由某个位置 start 的人站出来，并从后一个人开始报数，数到 m 的人就要站出来。然后从这个人的下一个人重新开始报数，再数到 m 的人站出来，依次重复下去，直到所有的人都站出来为止，则站出来的人的次序如何？

例如，当 n=8，m=4, start=4 时，出来的次序为 4，8，5，2，1，3，7，6 。

P68　P83(2.1)　P84(2.5)　P86(2.18)

思路分析

由于这n个人原坐的位置号分别为1，2，3，...,n，显然我们可以采用数组或链表来存储这n个位置号，当有人要站出来时，则将这个人的位置号输出并从该位置号序列中删除，如此反复上述过程，则可以把所有人的位置号输出并从序列中删去。

存储结构

顺序存储

- 操作的模拟

实例分析——顺序存储

数组下标	0	1	2	3	4	5	6	7
初始状态	1	2	3	4	5	6	7	8
第一个人出来后	1	2	3	5	6	7	8	
第二个人出来后	1	2	3	5	6	7		
第三个人出来后	1	2	3	6	7			
第四个人出来后	1	3	6	7				
第五个人出来后	3	6	7					
第六个人出来后	6	7						
第七个人出来后	6							
第八个人出来后								

数组下标	0	1	2	3	4	5	6	7
初始状态	1	2	3	4	5	6	7	8

1.设定起始位置：start=start-1; (3)

数组下标	0	1	2	3	4	5	6	7
初始状态	1	2	3	4	5	6	7	8

2.输出数据

3.元素往前移动：start+1 → n-1(最后元素下标)

数组下标	0	1	2	3	4	5	6	7
第一个人出来后	1	2	3	5	6	7	8	

4.往后开始报数：start=start+m-1 (3+4-1)

数组下标	0	1	2	3	4	5	6	7
第一个人出来后	1	2	3	5	6	7	8	

2.输出数据

3.元素往前移动：start+1 → n-2(最后元素下标)

4.往后开始报数：start=start+m-1 (6+4-1)

数组下标	0	1	2	3	4	5	6	7
第二个人出来后	1	2	3	5	6	7		

数组下标	0	1	2	3	4	5	6	7
第二个人出来后	1	2	3	5	6	7		

2.输出数据

3.元素往前移动：start+1 → n-3(最后元素下标)

4.往后开始报数：start=start+m-1 (3+4-1)

数组下标	0	1	2	3	4	5	6	7
第三个人出来后	1	2	3	6	7			

- start=(start+m-1) % 剩余元素个数
- 需要记录输出元素个数 counter
- 剩余元素个数：n-counter
- 最后元素下标:n-couter-1

实例分析——顺序存储

数组下标	0	1	2	3	4	5	6	7
初始状态	1	2	3	4	5	6	7	8
第一个人出来后	1	2	3	5	6	7	8	
第二个人出来后	1	2	3	5	6	7		
第三个人出来后	1	2	3	6	7			
第四个人出来后	1	3	6	7				
第五个人出来后	3	6	7					
第六个人出来后	6	7						
第七个人出来后	6							
第八个人出来后								

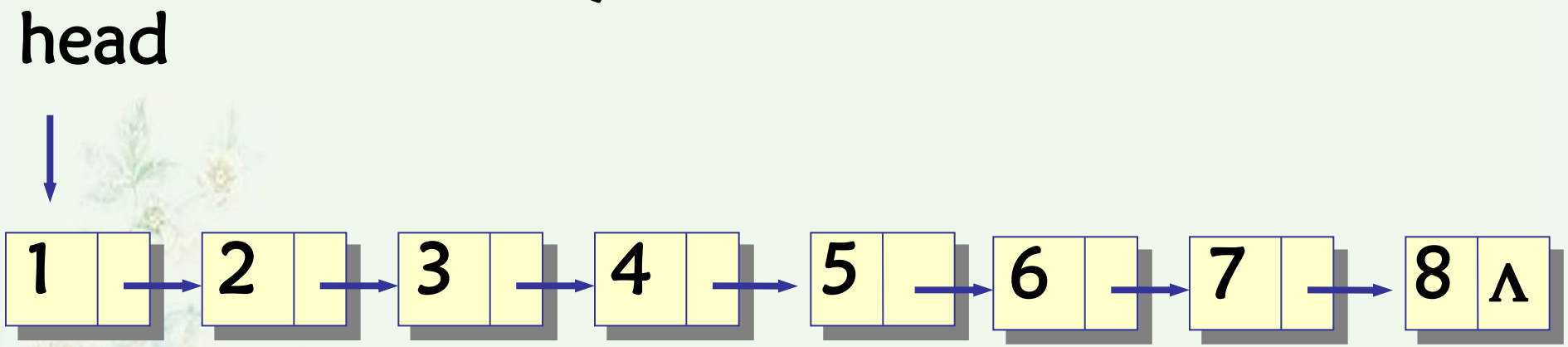
程序清单

```
void Josephus ( int n , int start , int m )
{
  int counter, j , *A=new int [n] ;
  for ( j=0 ; j<n ; j++ )  // 初始化，把各位置号存入数组中
     A[j]=j+1;
  counter=1; start--;
  while ( counter < n )  // 当前已站出来人的数目
  {
   cout<< A[start];  // 输出当前要站出来人的位置号
   for ( j=start ; j<n-counter ; j++ )
      A[j]=A[j+1];   // 把位置号前移
   start=( start+m-1) % ( n-counter );
   counter++;
  }
  cout<<A[0];
} // Josephus
```

链式存储结构

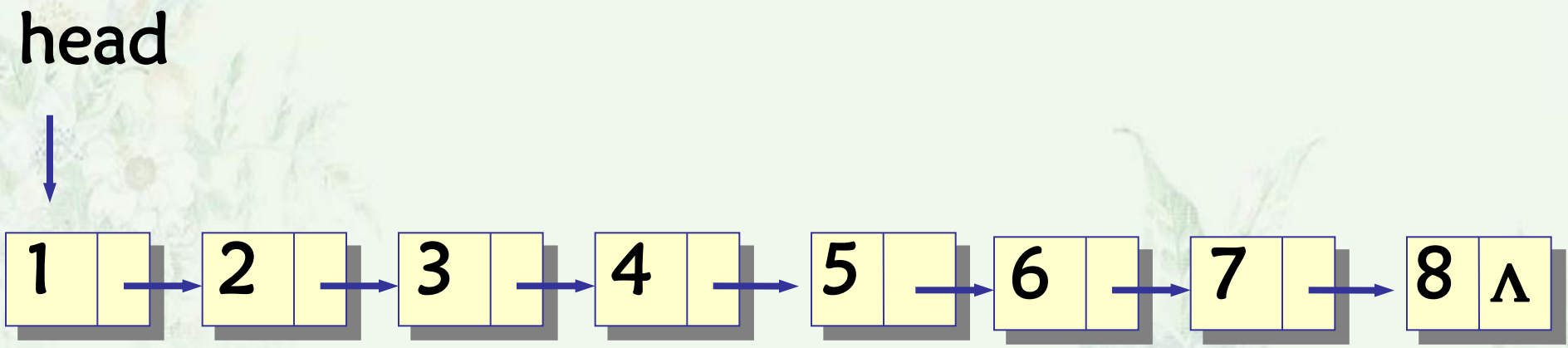
所需链表的特点

所需链表的特点

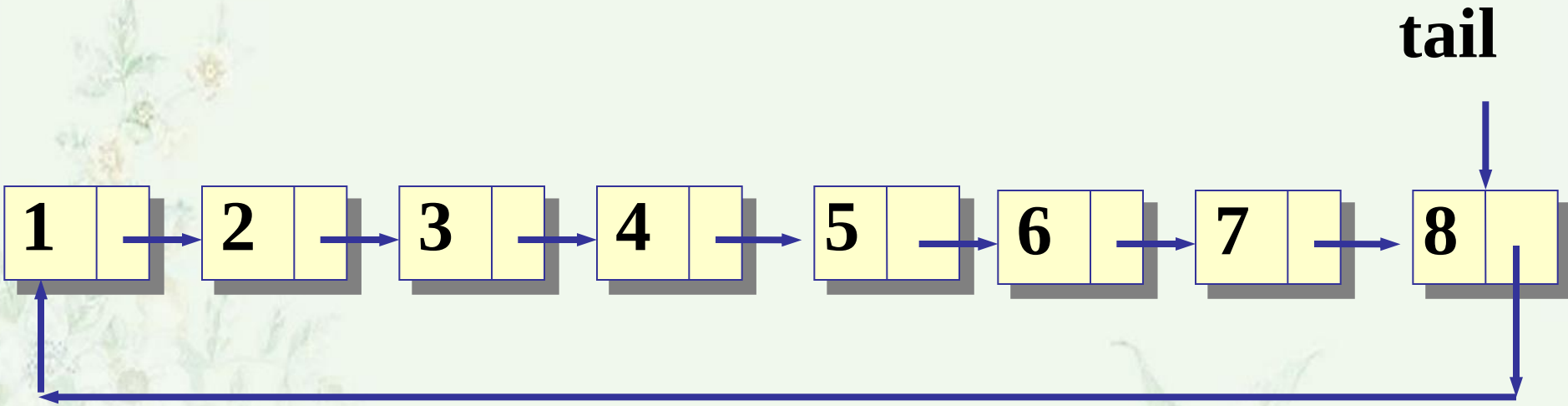


所需链表的特点

- 主要操作：搜索和删除



操作过程



算法的实现

- 链表中结点的结构只须有一个存放每个人位置号的整数数据域（data）和一个指向下一个结点的指针域（next）。
- 因此，问题解决的主要思路是：首先找到开始报数的结点p，接着从结点p开始沿链寻找其后的第m-1个结点，输出该结点的位置号后，删除该结点，然后再从该结点的下一个结点开始找其后的第m-1个结点，…，如此反复，直到所有的结点被删除为止。

算法步骤

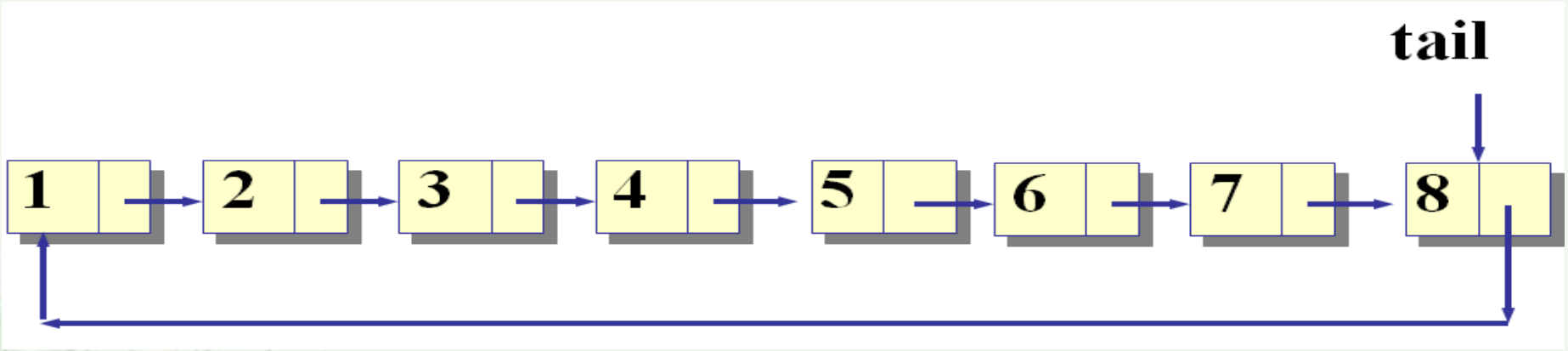
- （1）建立链表（循环链表）
  - 尾插入

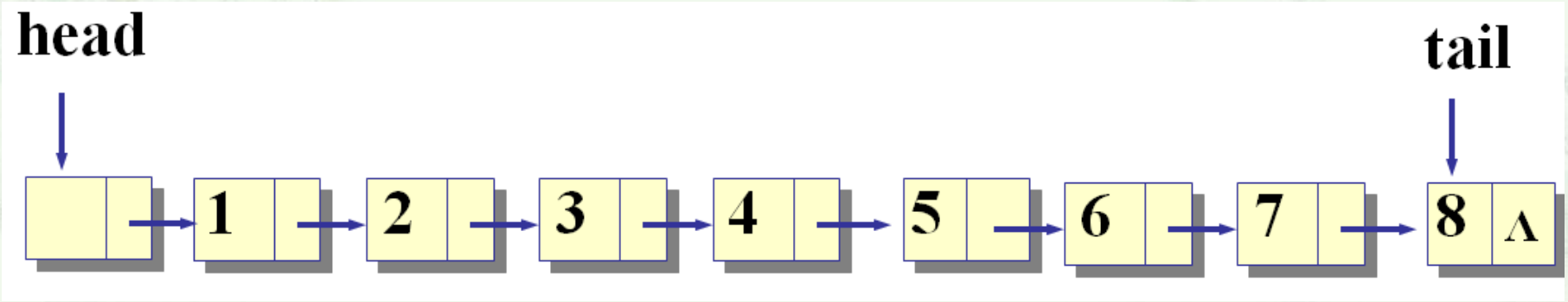
- （2）在循环链表中不断计数、输出并删除相应的结点。

算法 jose.cpp

```
void Josephus ( int n , int start , int m )
{
  LinkNode  *tail,*p , *ptr;      int i ;
  tail=CreateCir( n); // 生成以tail为尾指针的循环链表
   ptr=tail; p=tail->next;
   for ( i=1 ; i<start ; i++ )   //搜索到起始号
       {  ptr=p;    p=p->next;     }
   while ( ptr!=p)
     {
      cout<< p->data<<endl;  // 输出位置号
      ptr->next=p->next;  // 删除已输出的结点
      delete p ;
      p=ptr->next;
      for ( i=1 ; i<m ; i++ )   //搜索后面的第m个节点
        {  ptr=p;    p=p->next;     }
     }
    cout<< p->data<<endl; delete p; // 输出位置号
 }// Josephus
```

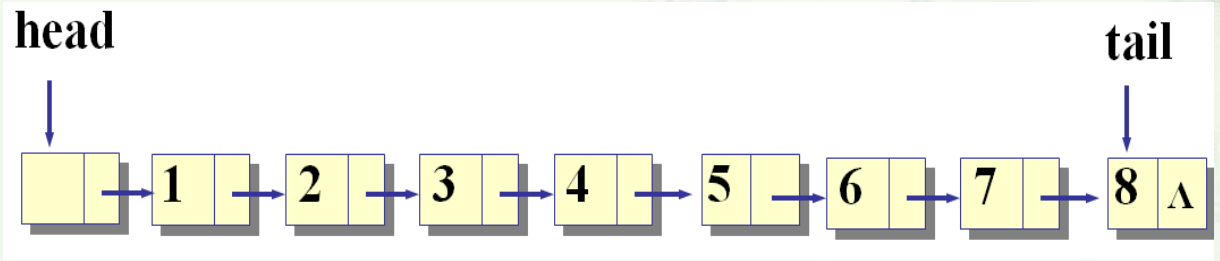
循环链表的建立





循环链表的建立

```
ListNode *Create2 ( int n )
{  ListNode *head,*tail;
   head=tail=new ListNode;   tail->next=0;
   for(j=0;j<n;j++)
    {  p=new ListNode;      cin>>p - >data;
       p->next=0;  tail->next=p;  tail=p;
    }
  tail->next=head->next;
  delete head;
 return tail;
}
```



数组中模拟链式结构

data	1	2	3	4	5	6	7	8
next	1	2	3	4	5	6	7	0
下标	0	1	2	3	4	5	6	7