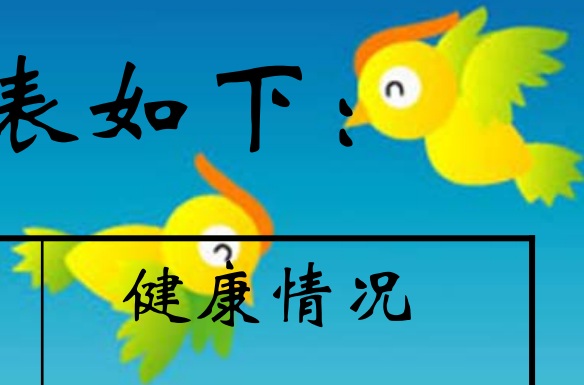


第二章 线性表



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
.....			

如何实现该系统？

系统分析



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

线性表的逻辑结构及其基本操作

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

形式化定义：

Linearlist = (D, R)

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

$R = \{N \mid N = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D_0, i = 1, 2, \Lambda, n-1 \}$

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 $a_i (2 \leq i \leq n-1)$ 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。

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， $i=1,2,\dots,n$ $n \geq 0$ 。

结构关系：对所有的数据元素 a_i ($i=1,2,\dots,n-1$) 存在次序关系 $\langle a_i, a_{i+1} \rangle$ ， 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。

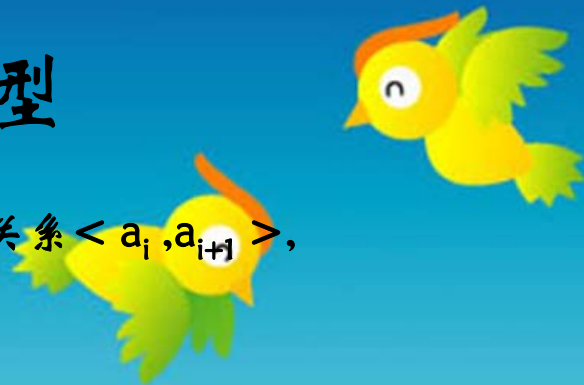
Locate(L,x) 定位操作。

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.操作的实现

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。

0	a_1
1	a_2
...	...
$i-1$	a_i
i	a_{i+1}
$n-1$	a_n
...	
...	
$\text{maxlen}-1$	

存储地址的计算



- 问题：假设线性表的每个元素需占用 d 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置 $LOC(a_1)$ ，那么线性表中第 i 个数据元素的存储位置 $LOC(a_i)=?$

0	a_1
1	a_2
...	...
$i-1$	a_i
i	a_{i+1}
$n-1$	a_n
...	
...	
$maxlen-1$	

存储地址的计算



- 由于 $LOC(a_i)$ 与第 $i-1$ 个数据元素的存储位置 $LOC(a_{i-1})$ 之间满足下列关系:

$$LOC(a_i) = LOC(a_{i-1}) + d$$

- 因此线性表的第 i 个数据元素 a_i 的存储位置为:

$$LOC(a_i) = LOC(a_1) + (i-1) * d$$

顺序存储类



- 存储空间：用数组实现

- 属性：.....

操作：.....

0	a_1
1	a_2
...	...
$i-1$	a_i
i	a_{i+1}
$n-1$	a_n
...	
...	
$\text{maxlen}-1$	

结构类型的描述



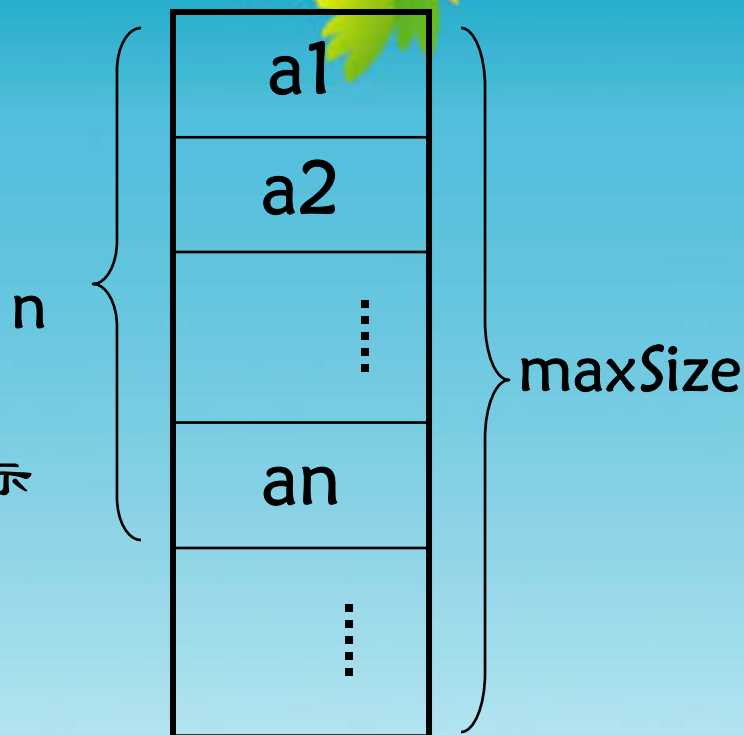
```
#define maxSize 100  
//const int maxSize=最大容量;
```

```
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

data[0..maxSize-1]



- 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
```




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 == NULL)        //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 == 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; }
```

```
};
```

查找



(1)按序号查找

(2)按值查找

按值查找



0	a_1
1	a_2
...	...
$i-1$	a_i
i	a_{i+1}
$n-1$	a_n
...	
...	
$\text{maxlen}-1$	

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 ($1 \leq i \leq n+1$) 个位置上，插入一个新结点 el ，使长度为 n 的线性表

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

变成长度为 $n+1$ 的线性表

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

单元编号(序号)	内容	单元编号(序号)	内容	
0	a_1	0	a_1	
1	a_2	1	a_2	
...	
$i-1$	a_i	$i-1$	el	←插入位置
i	a_{i+1}	i	a_i	
		
$n-1$	a_n	$n-1$	a_{n-1}	
...		n	a_n	
...		...		
$maxlen-1$		$maxlen-1$		
	插入前		插入后	

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 $1 \leq i \leq n+1$ ，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 e_i 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 ( $1 \leq i \leq n+1$ ) 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 ($1 \leq i \leq n$) 结点删除，使长度为 n 的线性表： $(a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 变成长度为 $n-1$ 的线性表 $(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

单元编号 (序号)	内容	← 删除元素	单元编号 (序号)	内容
0	a_1		0	a_1
1	a_2		1	a_2
...
$i-1$	a_i		$i-1$	a_{i+1}
i	a_{i+1}		i	a_{i+2}
...
...	...		$n-2$	a_n
$n-1$	a_n		...	
...			...	
$\text{maxlen}-1$			$\text{maxlen}-1$	
删除前			删除后	

删除算法的步骤为：



(1) 检查 i 是否满足条件 $1 \leq i \leq n$, 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 ( $1 \leq i \leq n$ ) 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



半”运算

半”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 = 0;  
    while ( i < n1 ) {  
        x = LA.getData(i);    //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, 不能预先进行

如何解决?

例、学生健康情况登记表如下：



姓 名	学 号	年 龄	健康情况
王小林			健康
陈 红		20	一般
刘建平	790633	男 21	健康
张立立	790634	男	
.....

用新方法实现该系统？

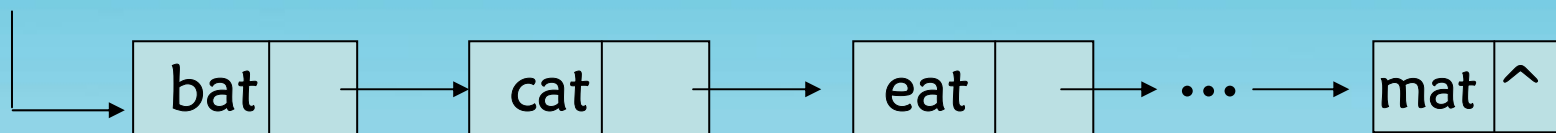
链式结构

线性表的链式存储和实现



线性表:(bat, cat, eat, fat, hat, jat, lat, mat)

first



链式结构中结点结构的抽象



data	link
-------------	-------------

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



data

link

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

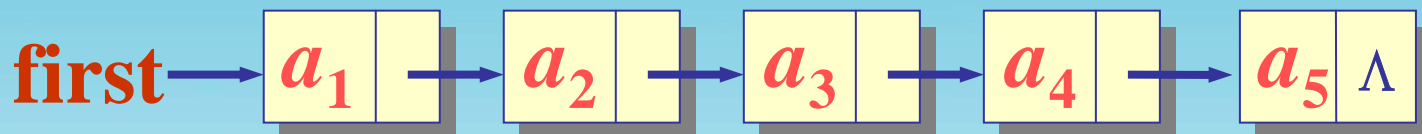
单链表 (Singly Linked List)



- 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)

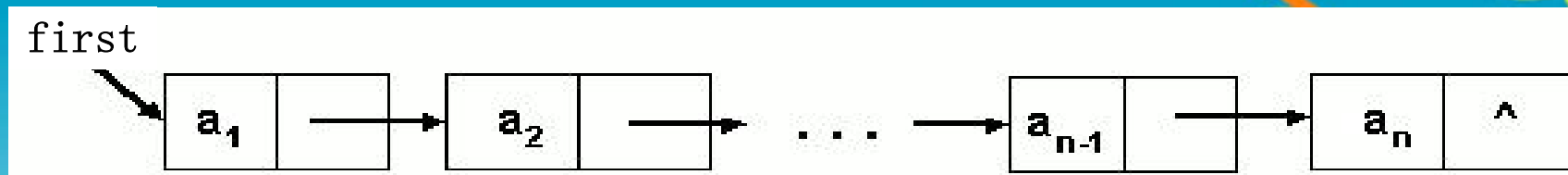


(5) 空指针 NULL或0

`p->link=NULL;`

`q=NULL;`

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>:: HLinkedList (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;  
  }  
}
```



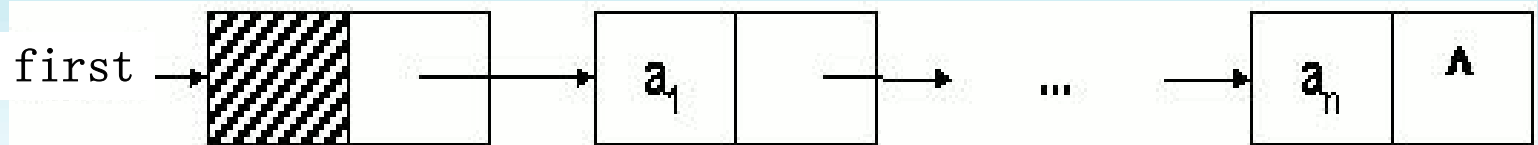
该算法的缺陷是不如头插入算法那么简洁

尾插入建链改进算法

```
template <class T>  
List<T>:: RLinkList (int n )
```

引入表头结点

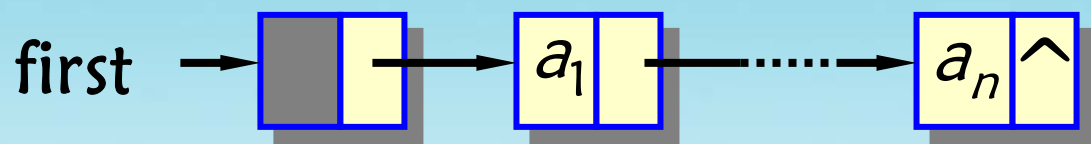
```
{  
    first=tail=new LinkNode <T>();  
    for(i=0;i<n;i++)  
    { p=new Node<T>();  cin>>p->data;  
      p->link=0;  
      tail->link=p;  
      tail=p;  
    }  
}
```



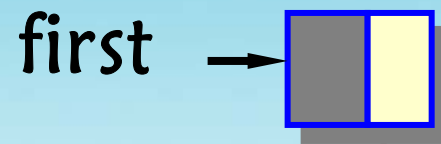
带表头结点的单链表



- 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 = NULL; }                 //构造函数
    LinkNode(T item, LinkNode<T> * ptr = NULL)
        { 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



```
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 == NULL ? 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 != NULL )
    {
        //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。



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 != NULL )
    {    //逐个结点检测
        p = p->link; count++;
    }
    return count;
}
```

算法分析



查找操作



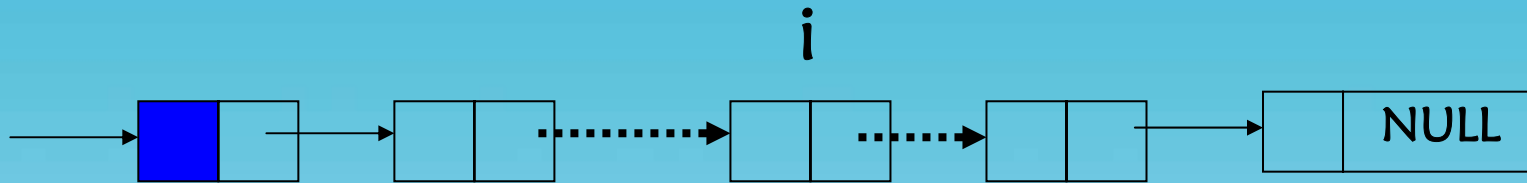
- 按序号查找——单链表找第 i 个元素——定位
- 按值查找——搜索

定位操作



`LinkNode<T> * locate(int i)`

功能：求单链表第*i*个结点。



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NULL。
```

```
    if (  $i < 0$  ) return NULL;           //i不合理
```

```
    LinkNode<T> * current = first; int k = 0;
```

```
    while ( current != NULL &&  $k < i$  )
```

```
        { current = current->link; k++; }
```

```
    return current;    //返回第 i 号结点地址或NULL
```

```
};
```

算法的分析



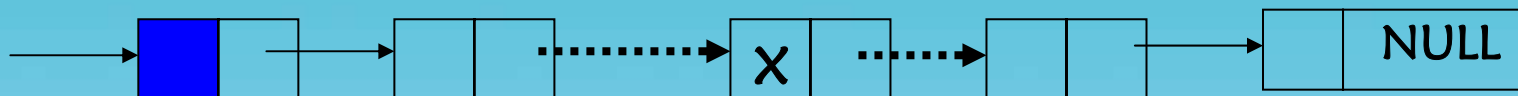
假设当前链表中的结点总数为 n 个：

- 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 $2 \cdot 1/n \cdot (1+2+\dots+n) = n+1$ ，所以该算法的平均时间复杂度为 $O(n)$ 。

搜索操作

$\text{LinkNode}\langle T \rangle * ::\text{Search}(T\ x)$

功能：在单链表中搜索值为x的结点。



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NULL。
```

```
    LinkNode<T> * current = first->link;
```

```
    while ( current != NULL && current->data != x )  
        current = current->link;
```

```
    //沿着链找含x结点
```

```
    return current;
```

```
};
```

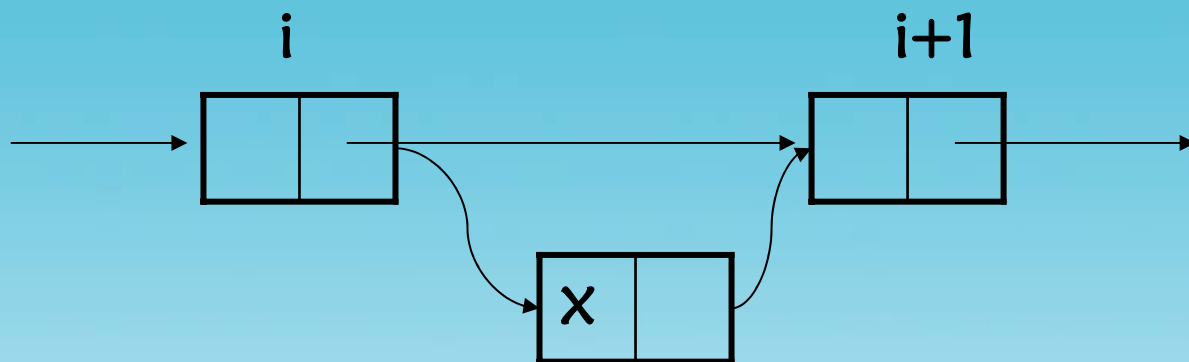

算法分析



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。

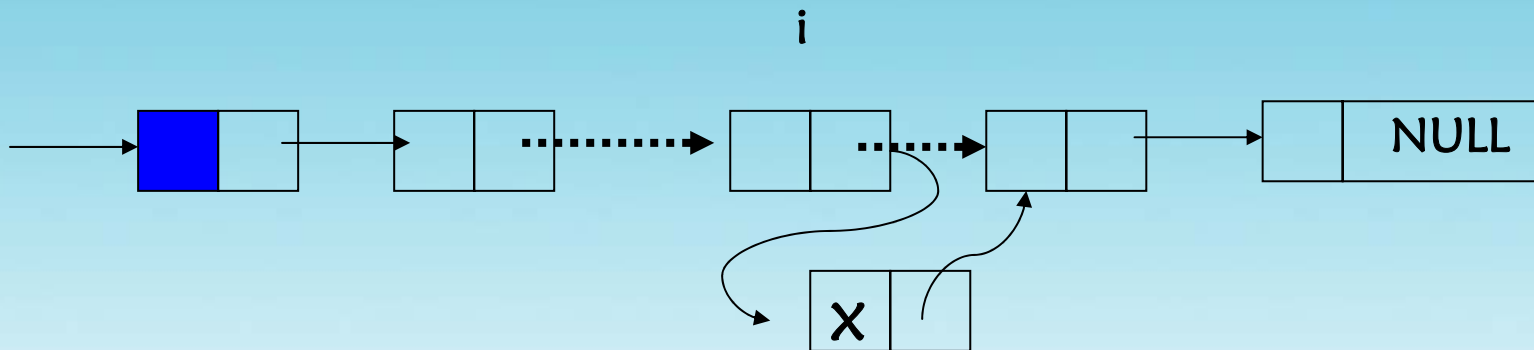
该函数的功能为：在带头结点的单链表中的第i个结点之后插入数据元素值为x的新结点。

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 == NULL) 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

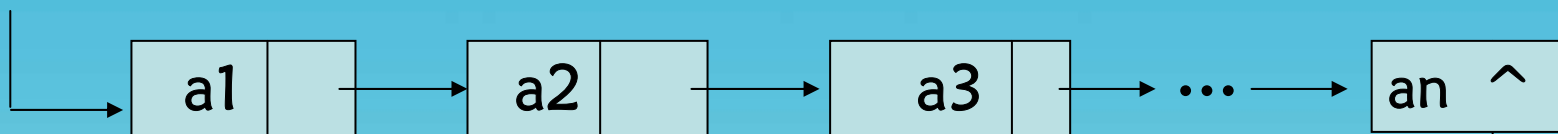


该算法的执行时间与插入点所对应的位置有关，假设当前链中的结点总数为 n 个且在各结点之前插入的概率相等，则平均时间复杂度为 $O(n)$ 。

是否可以改进？

改进算法：**互换法**

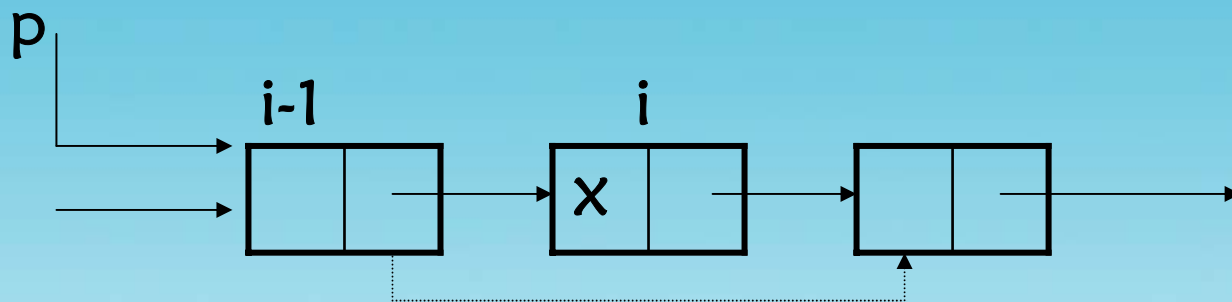
插入



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。



$p \rightarrow \text{link} = p \rightarrow \text{link} \rightarrow \text{link};$

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NULL，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 == NULL || current->link == NULL)
        return false;        //删除不成功
    LinkNode<T> *del = current->link;
    current->link = del->link;
    x = del->data;  delete del;
    return true;
};
```

算法分析



- 删除算法分析: $O(n)$
- 改进方案: 互换法

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 != NULL)
```

```
    {
```

```
        q = first;           //保存被删结点
```

```
        first = first->link; //从链上摘下该结点
```

```
        delete q;           //删除
```

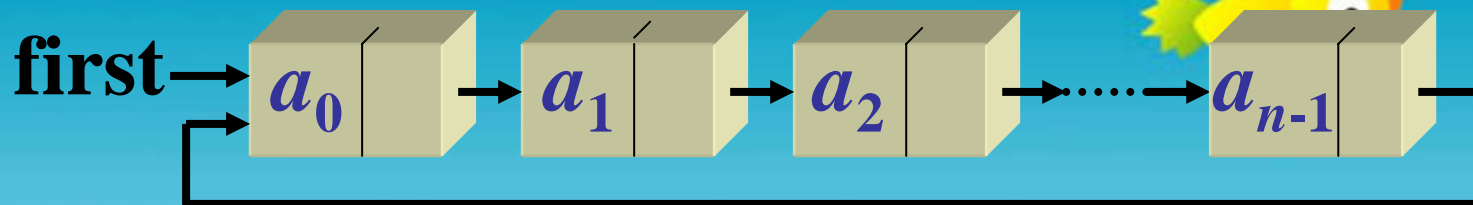
```
    }
```

```
};
```

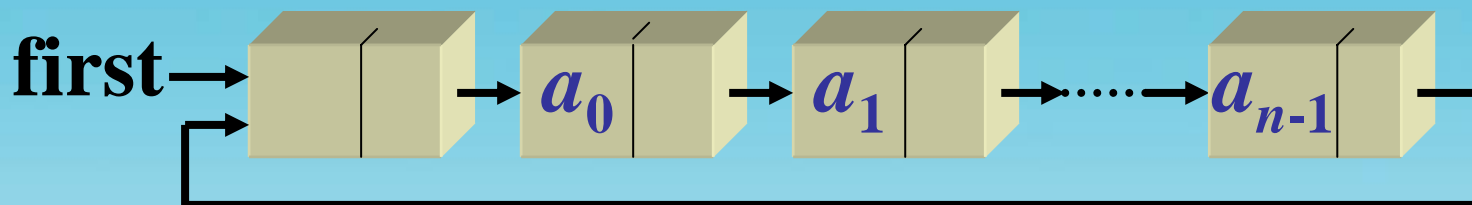
其他形式的链式结构



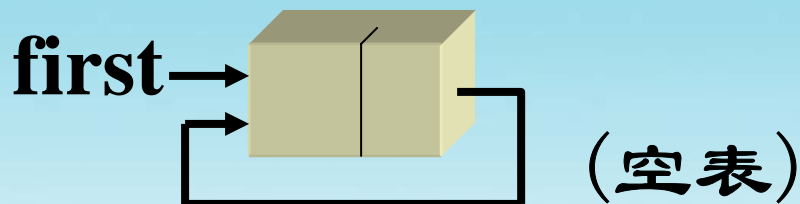
循环链表



• 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 NULL，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 {
    T data;
    CircLinkNode<T> *link;
    CircLinkNode ( CircLinkNode<T> * next =
        NULL ) { link = next; }
    CircLinkNode ( T d, CircLinkNode<T> * next =
        NULL ) { data = d; link = next; }
    bool Operator==(T x) { return data.key == x.key; }
    bool Operator!=(T x) { return data.key != x.key; }
};
```

//链表结点类定义


```
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
```

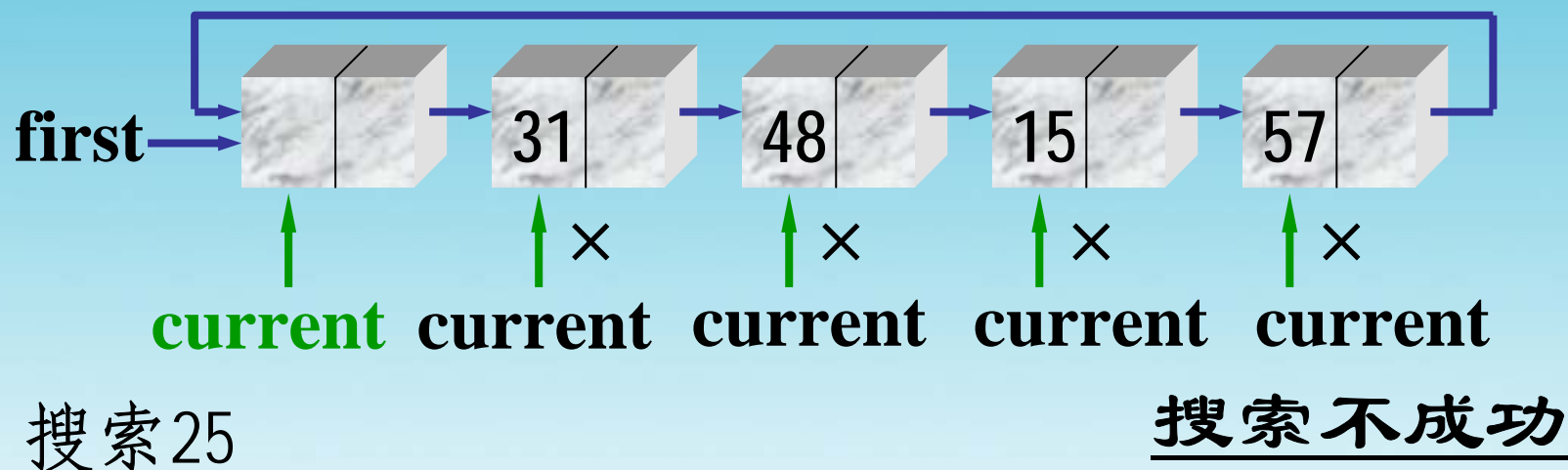
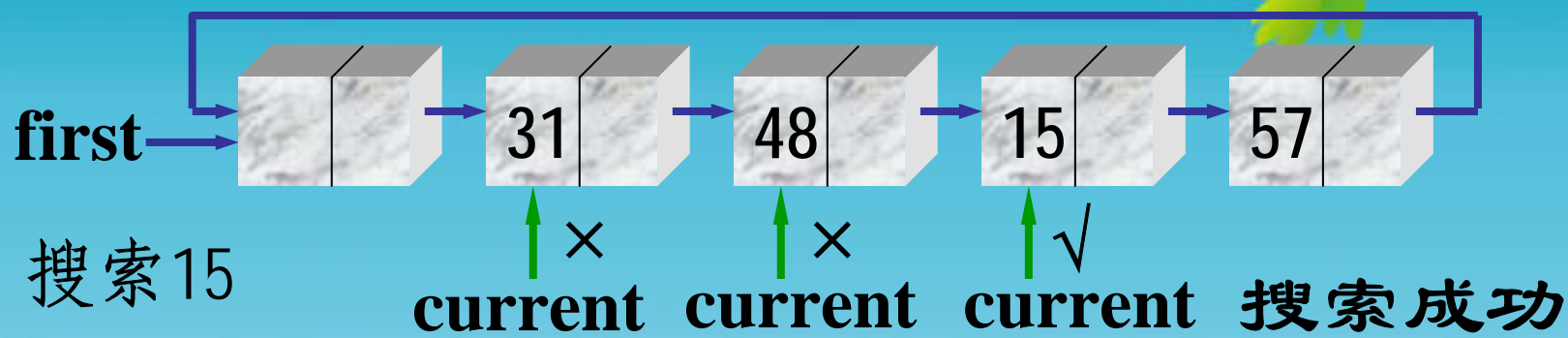
```
bool Insert ( int i, T x ); //插入
```

```
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NULL，而是表头。

循环链表的搜索算法



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

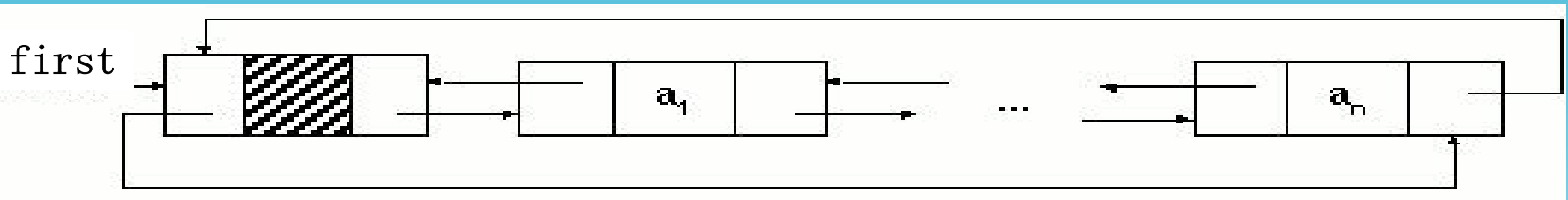
$(a_1, a_2, a_3, \dots a_n)$ 和 $(b_1, b_2, b_3, \dots b_n)$

链接成一个线性表的运算。

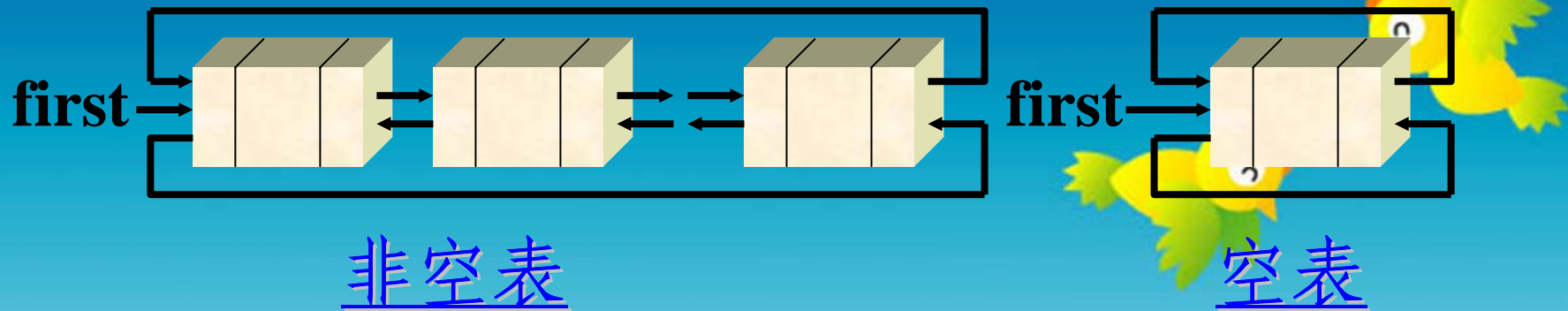
双向链表



- 单链表及循环链表的缺陷

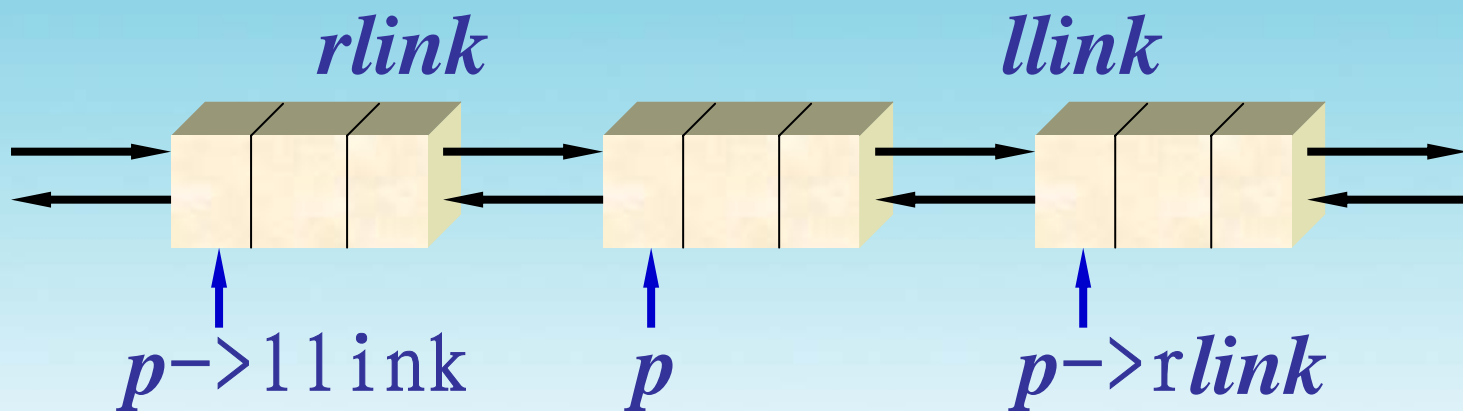


llink	data	rlink
-------	------	-------



- 结点指向

$$p == p \rightarrow \text{llink} \rightarrow \text{rlink} == p \rightarrow \text{rlink} \rightarrow \text{llink}$$



双向循环链表类型定义



llink	data	rlink
-------	------	-------

双向循环链表结点类的定义



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

```
    first->rLink = first->lLink = first;
```

```
};
```

```
DbllNode<T> *getFirst () const { return first; }
```

```
void setFirst ( DbllNode<T> *ptr ) { first = ptr; }
```

```
DbllNode<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
```



```
DbtNode<T> *Locate ( int i, int d );
```

```
//在链表中定位序号为i( $\geq 0$ )的结点, d=0按前驱方  
//向, d $\neq 0$ 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 $\neq 0$ 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:
```

```
DbtNode<T> *first;
```

```
//表头指针
```

```
};
```

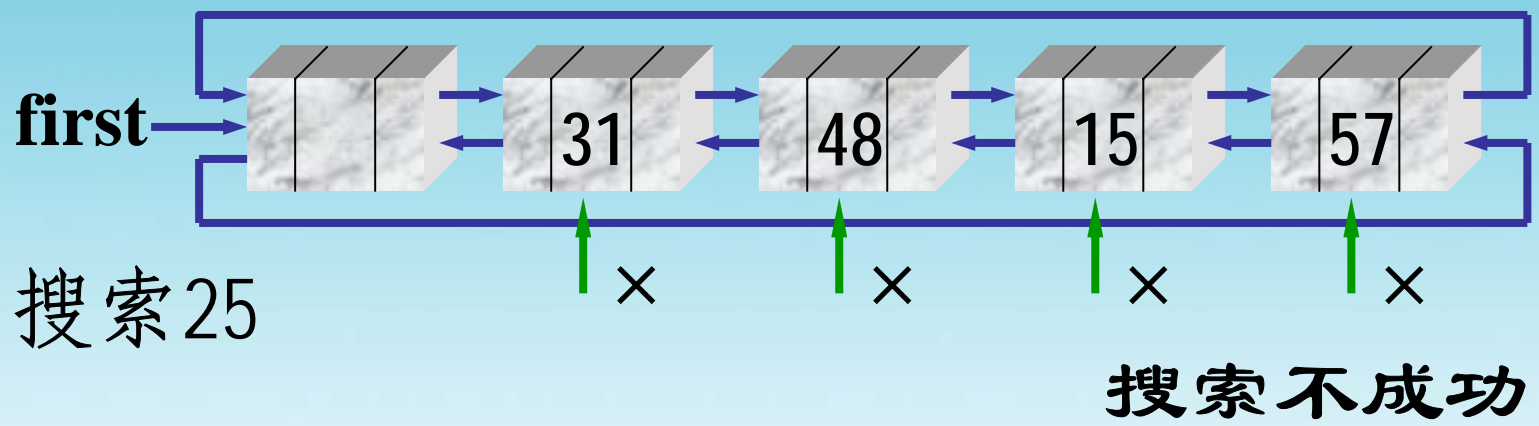
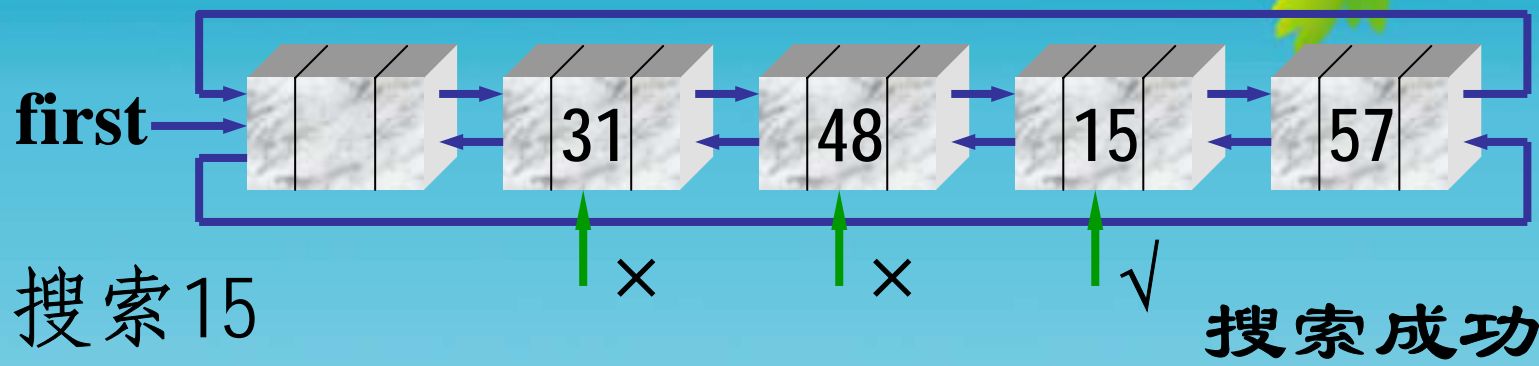
双向链表的操作特点:



“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



双向循环链表的搜索算法



```
template <class T>
```

```
DbtNode<T> *DbtList<T>::Search ( T x , int d ) {
```

```
//在双向循环链表中寻找其值等于x的结点。
```

```
    DbtNode<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 NULL; //搜索失败
```

```
};
```

双向循环链表插入操作



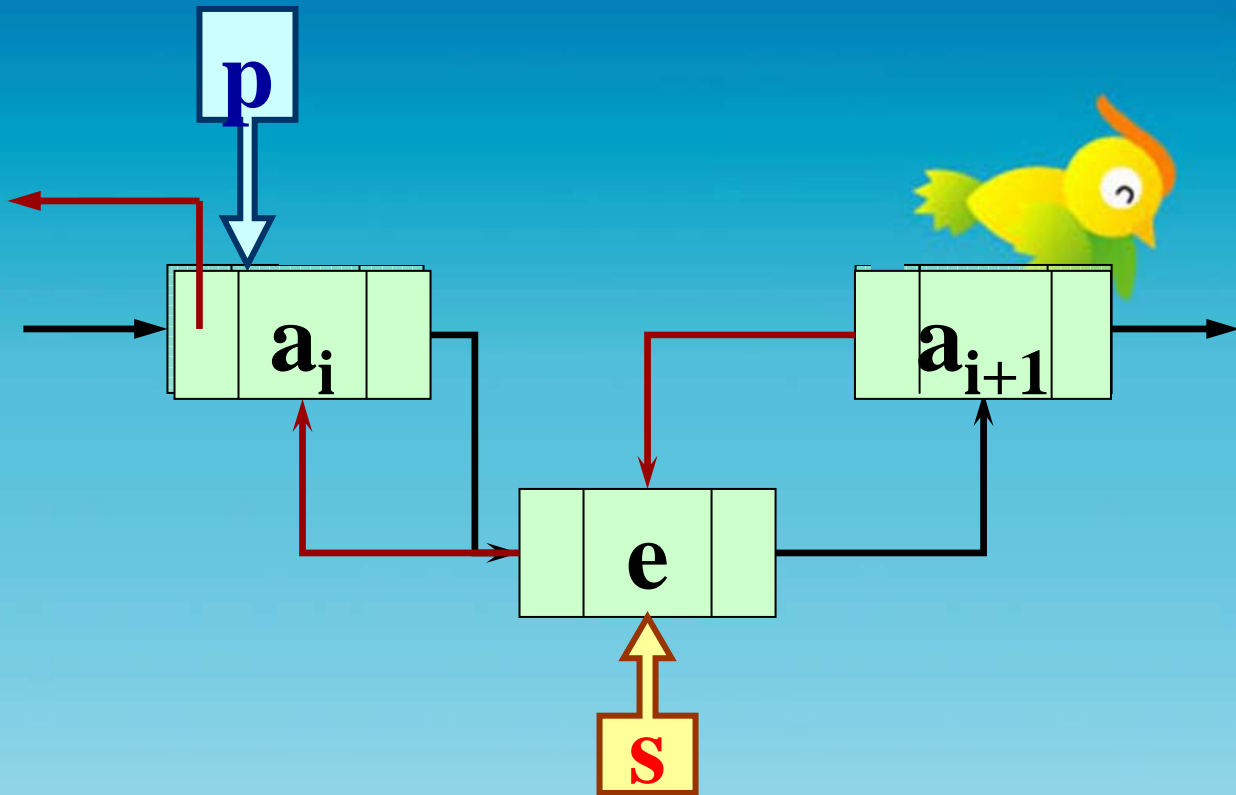
bool Dbllist<T>::Insert (int i, T x, int d)

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



$s \rightarrow rlink = p \rightarrow llink;$ $p \rightarrow rlink = s;$

$s \rightarrow rlink \rightarrow llink = s;$ $s \rightarrow llink = p;$

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。
```

```
    DbllNode<T> *current = Locate(i, d);
```

```
    //按d指示方向查找第i个结点
```

```
    if ( current == NULL ) return false; //插入失败
```

```
    DbllNode<T> *newNd = new DbllNode<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



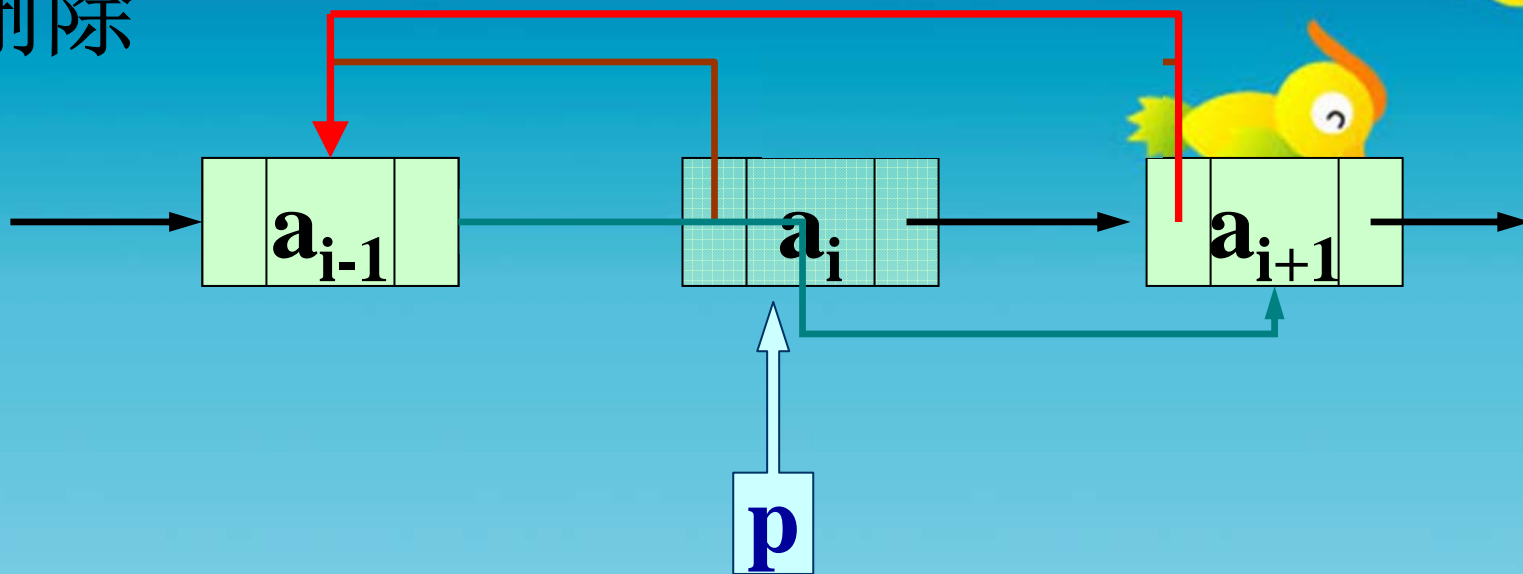
bool DblList<T>::Remove(int i, T& x, int d)

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NULL。

删除



$p \rightarrow llink \rightarrow rlink = p \rightarrow rlink;$

$p \rightarrow rlink \rightarrow llink = p \rightarrow llink;$

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。
    DbllNode<T> *current = Locate (i, d);
    if (current == NULL) 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. 将一个元素插入到一个有序表中，并要求插入新元素后，表仍维持着有序。
- (1) 设线性表存储在数组 $A[1..arrsize]$ 的前 $elenum$ 个分量中，且递增有序。试编写一个算法：将 x 插入到线性表的适当位置上，以保持线性表的有序性，并且分析算法的时间复杂度。
- (2) 已知单链表 L 中的结点是按值非递减有序排列的，试编写一算法将值为 x 的结点插入到表 L 中，使得 L 仍然有序。



- 例2: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



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。

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



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 count , 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;
    count=1; start--;
    while ( count < 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-count ; j++ )
            A[j]=A[j+1]; // 把位置号前移
        start=( start+m-1 ) % ( n-count );
        count++;
    }
    cout<<A[0];
} // Josephus
```

链式存储结构



所需链表的特点

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; // 输出位置号
} // Josephus
```

• 习题解答提示



- 6. 已知一个单链表中的数据元素含有三类字符（即字母字符，数字字符和其它字符），试编写算法，构造三个循环链表，使每个循环链表中只含有同一类的字符，且利用原表中的结点空间作为这三个表的结点空间。
- 8. 已知指向一线性表第一个结点的指针为 $first$ ，试写一个算法删除链表中第 i 个结点开始的连续 k 个结点。



- 9.试编写一个算法，找出一个循环链表中的最小值。
- 16.已知线性表第一个结点的指针为first，使编写一个算法按递减次序打印各结点数据域中的内容（提示：反复执行在链表中找出最大值的结点，打印之后将其删除，直到链表为空为止）。

- 20. 试编写一个算法，判别一个线性表中的元素是否对称。
- 21. 已知线性表中的元素以值递增有序排列的整数，并以单链表作为存储结构，试编写一个高效的算法，删除表中所有值大于 mink 且小于 maxk 的元素（若表中存在这样的元素），并分析你的算法的时间复杂度（注意： mink 和 maxk 是给定的两个参数）





- 23. 假设有两个按元素值递增有序排列的线性表A和B，均以单链表作为存储结构，试编写算法将表A和表B合并成一个按元素值递减有序排列的线性表C，并要求利用原表（即表A和表B的）结点空间存放表C。
- 24. 已知A,B,C为三个递增有序的线性表，现要求对A表作如下运算：删除那些既在表B中出现又在表A中出现的元素，试分别以顺序存储结构和链式存储结构来实现上述算法。

向量（vector容器类）



- 向量（vector容器类）：
- `#include <vector>`，vector是一种动态数组，是基本数组的类模板。
- 支持随机访问迭代器，所有STL算法都能对vector操作。
- 随机访问时间为常数。在尾部添加速度很快，在中间插入慢。

例如：

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    vector<int> v; //一个存放int元素的向量，一开始里面没有元素
    v.push_back(2);
    v.push_back(4);
    v.push_back(3);
    v.push_back(1);
    vector<int>::const_iterator i; //常量迭代器
    for(i = v.begin(); i != v.end(); i++)
        cout << *i << ",";
    cout << endl;
    sort(v.begin(), v.end()); //排序处理
    for(i = v.begin(); i != v.end(); i++)
        cout << *i << ",";
}
```

输出：2, 4, 3, 1,
1, 2, 3, 4,



表 (List 容器类)



- List (`#include <list>`) 又叫链表, 是一种双线性列表, 只能顺序访问 (从前向后或者从后向前), list 的数据组织形式如下图。
- 与前面的容器类有一个明显的区别就是: 它不支持随机访问。要访问表中某个下标处的项需要从表头或表尾处 (接近该下标的一端) 开始循环。而且缺少下标运算符: `operator[]`。

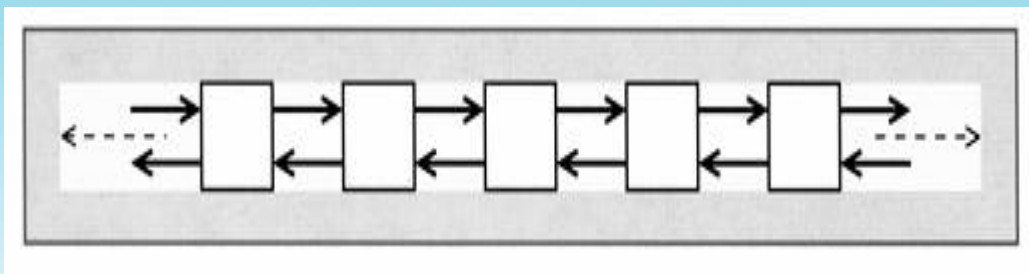


表 (List 容器类)



- 在任何位置插入删除都是常数时间，不支持随机存取。
除了具有所有顺序容器都有的成员函数以外，还支持8个成员函数：
 - push_front: 在前面插入
 - pop_front: 删除前面的元素
 - sort: 排序 (list 不支持STL 的算法sort)
 - remove: 删除和指定值相等的所有元素
 - unique: 删除所有和前一个元素相同的元素
 - merge: 合并两个链表，并清空被合并的那个
 - reverse: 颠倒链表
 - splice: 在指定位置前面插入另一链表中的一个或多个元素,并在另一链表中删除被插入的元素

使用例子



```
#include <iostream>
#include <string>
#include <list>
using namespace std;
void PrintIt(list<int> n)
{  for(list<int>::iterator iter=n.begin(); iter!=n.end(); ++iter)
    cout<<*iter<<" ";//用迭代器进行输出循环
}
int main()
{
    list<int> listn1,listn2;  //给listn1,listn2初始化
    listn1.push_back(123);    listn1.push_back(0);    listn1.push_back(34);
    listn1.push_back(1123);  //now listn1:123,0,34,1123
    listn2.push_back(100);
    listn2.push_back(12);  //now listn2:12,100
    listn1.sort();
    listn2.sort();  //给listn1和listn2排序
    //now listn1:0,34,123,1123    listn2:12,100
    PrintIt(listn1);
    cout<<endl;
    PrintIt(listn2);
    listn1.merge(listn2);  //合并两个排序列表后,listn1:0, 12, 34, 100, 123, 1123
    cout<<endl;
    PrintIt(listn1);
}
```