问题求解与实践
——数据结构的基本概念

主讲教师：陈雨亭、沈艳艳

数据和数据结构

- **数据**（Data）是能够被计算机存储、加工的对象，是信息的表达形式
- 组成数据的基本单位称为**数据元素**（Data Element）。也称为**元素**（Element）、**结点**(Node)或**记录**(Record)。

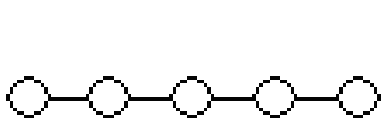
数据的逻辑结构

- **逻辑结构**
  - 是数据的组织形式，用来表示数据元素之间的逻辑关系，即数据元素之间的关联方式或相邻关系
- 逻辑结构的组成
  - 数据元素的集合，可以用D表示
  - 元素之间的前趋后继关系的集合，用R表示
- 所以数据结构可以表示为：
  - DS=（D，R）

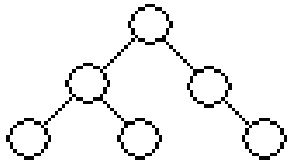
三种基本的逻辑结构

- **线性结构**
  - 数据元素之间存在着一对一的前趋后继关系
- **树形结构**
  - 只有一个处在最高层次的数据元素没有前趋，这个数据元素称为根结点，其他每个数据元素都有并且仅有一个前趋，而每个数据元素的后继则没有个数的限制
- **图结构**
  - 每一个数据元素都可以有任意多个前趋和后继，任何两个结点之间都可能邻接

三种基本的逻辑结构



线性结构



树形结构

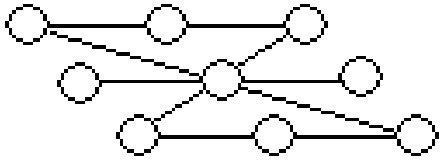


图 结构

数据的物理结构

- 数据元素及其关系在存储器中的存放形式称为数据的**物理结构**，也称为存储结构
- 物理结构的分类：
  - **顺序存储结构**
    - 元素按某种顺序存储在连续的存储单元中，存储位置间的关系反映了元素间的逻辑关系
  - **链式存储结构**
    - 元素存放在不一定连续的存储单元中。通过在元素中附加信息来表示与其相关的一个或多个其他元素的物理地址来建立元素间逻辑关系

数据的物理结构

- 物理结构的分类
  - **索引存储结构**
    - 将数据元素排成一个序列，每个元素E_i在序列中对应的位置i，称为元素的索引，在存储时，建立附加的索引表，索引表中的第i个值就是第i个元素E_i的存储地址
  - **散列存储结构**
    - 每个数据元素均匀存放在存储区中，在数据元素和其在存储器中的存储位置之间建立一个映象关系F（即函数）。根据这个映象关系可以得到它的存储地址A
- 一种逻辑结构可以采用任何一种存储结构来实现，但不同的物理结构要影响到对数据的操作

数据的运算

- **数据的运算**是对数据元素进行的某种操作，例如
  - 改变元素的个数（增加或删除）
  - 改变元素的顺序
  - 改变元素之间的关系
  - 浏览每个元素（遍历）
  - 检索符合某个条件的数据元素

问题求解与实践
——线性表

主讲教师：陈雨亭、沈艳艳

线性表

- **线性表**（Linear List）
  - 是由有限个相同类型的数据元素组成的有序序列，一般记作（$a_1,a_2,...,a_n$）
- 特点
  - 除了a_1和a_n之外，任意元素a_i都有一个直接前趋a_{i-1}和一个直接后继a_{i+1}
  - a_1无前趋
  - a_n无后继
- 表的长度：线性表中数据元素的个数
- 空表：元素个数为0的表

线性表的运算

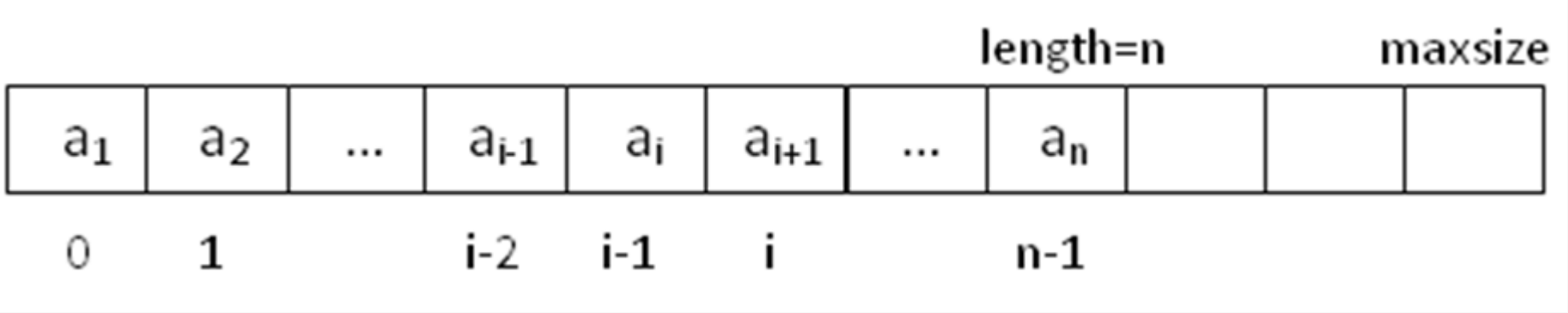
- 在线性表中，经常执行下列操作
  - 确定线性表是否为空
  - 确定线性表的长度
  - 查找某个元素
  - 删除第i个元素
  - 在第i个位置插入一个新元素

线性表的存储结构

- 采用顺序存储结构的称为**顺序表**
- 采用链式存储结构的称为**线性链表**

顺序表

- 顺序表中的数据元素按照逻辑顺序依次存放在一组连续的存储单元中
- 每个元素a_i的存储地址是该元素在表中位置i的线性函数



顺序表的定义

```
const int maxsize=200;          // 顺序表最大允许长度
struct SeqList {
   ElemType  data[maxsize];     // 顺序表存储数组的地址
   int length;                  // 顺序表当前长度
};
SeqList  List;
List.length=0;
```

在顺序表中插入元素

① 判断插入位置是否合理以及该表是否已满

② 从最后一个元素开始依次向前，将每个元素向后移动一个位置，一直到第i个元素为止

③ 向空出的第i个位置存入新元素x

④ 将线性表长度加1

在顺序表中插入元素

```
void Insert( SeqList *L, int i, ElemType x )
{
    if( i<1 || i>L->length+1 || L->length==maxsize )
        cout<<"插入位置错误或表满";
    else {
        for( int j=L->length-1; j>=i-1; j-- )
            L->data[j+1] = L->data[j];  // 元素依次向后移动
        L->data[i-1] = x;               // 向第 i 个位置存入新元素 x
        L->length++;                    // 表长度加一
     }
}
```

在顺序表中删除元素

① 判断要删除元素的位置的合理性

② 从第i+1个元素开始，依次向后直到最后一个元素为止，将每个元素向前移动一个位置。这时第i个元素已经被覆盖删除

③ 将线性表长度减1

在顺序表中删除元素

```
void Delete( SeqList *L, int i )
{
    if(i<1 || i>L->length )
        cout<<"表中没有第"<<i<<"个元素";
    else {
        for ( int j=i; j<=L->length-1; j++ )
            L->data[j-1] = L->data[j];
        L->length--;
    }
}
```

在顺序表中查找元素

- 数据元素是基本数据类型
  - 与数据元素**本身**进行对比
- 数据元素是结构体或类对象
  - 往往是与数据元素的**某个属性**比较

在顺序表中查找元素

```
int Find( SeqList *L, ElemType x )
{
    for( int i = 0; i<L->length; i++ )
    {
        if( L->data[i]==x )  return i+1;      //查找成功，返回元素位置

    }
    return 0;                          //查找失败, 返回 0
}
```

顺序存储的特点

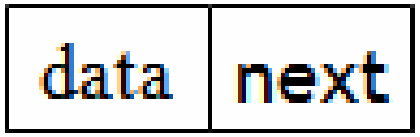
- **优点**
  - 不需要为元素间的逻辑关系增加额外的存储空间
  - 可以方便地随机存取顺序表中任意一个元素
- **缺点**
  - 元素进行插入和删除时都要进行大量元素的移动，因此，操作的效率较低
  - 占用连续的存储空间，存储空间的大小在初始化时就必须确定

问题求解与实践
——线性链表

主讲教师：陈雨亭、沈艳艳

线性链表

- **线性链表**
  - 采用链式存储的线性表
- 存储特点：每个结点都分两部分
  - **数据域**：存储元素的值
  - **指针域**：存放直接前趋或直接后继元素的地址信息



链表的存储结构

	存储地址	数据域	指针域
	20H	a_2	80H
head		…	…
40H	40H		90H
		…	…
	80H	a_3	NULL
	90H	a_1	20H

链表的定义和初始化

```
struct LNode
{
   ElemType data;          // 数据域，ElemType代表某种数据类型
   struct LNode *next;     // 指针域
} ;


LNode*  head;              // 定义头指针
head = new LNode;          // 定义头结点
head->next = NULL;         // 头结点指针域为空
```

求单链表的长度

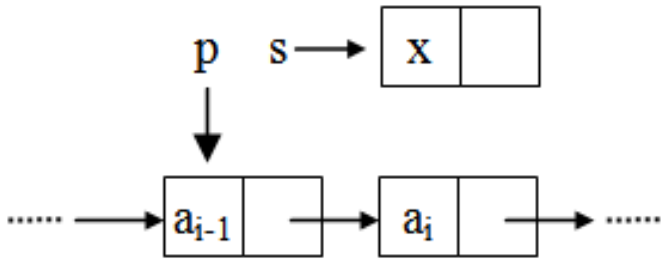
```
int Length( )
{
    LNode *p=head->next;        //p指向第一个元素所在结点
    int len=0;
    while( p!=NULL ) {          //逐个检测p结点存在与否
        len++;
        p=p->next;              //指针后移
    }
    return len;
}
```

在链表的第i个位置插入新的结点

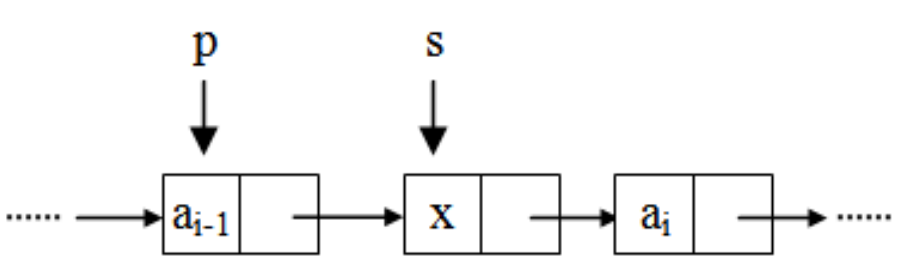
① 找到第i-1个结点的指针p。

② 建立新结点s，将新结点的指针域保存第i个结点的地址。

③ 第i-1个结点的指针域保存新结点的地址



(a) 插入前



(b) 插入后

在链表的第i个位置插入新的结点

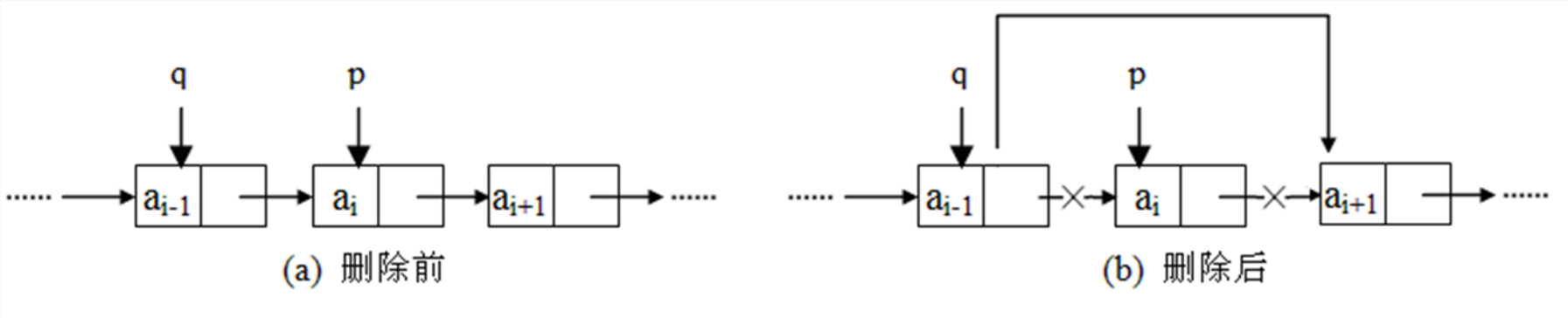
```
void Insert(LNode *head, int i, ElemType x)
{
        if(i<1) {
                cout<<"不存在第"<<i<<"个位置";
        } else {
                LNode *p=head;                          //p最终将指向第i-1个结点
                int k=0;                                //p目前指向第0个结点(头结点)
                while( p!=NULL&&k<i-1 )
                {
                        p=p->next;
                        k++;
                }
                if(p==NULL)
                        cout<< i<<"超出链表最大可插入位置";
                else {
                        LNode *s=new LNode;             //建立新结点s
                        s->data = x;
                        s->next=p->next;                //修改结点s的指针
                        p->next=s;                      //修改结点p的指针
                }
        }
}
```

从链表中删除第i个结点

① 如果第i个结点存在，则找到第i个结点和第i-1个结点的指针p和q

② 将第i+1个结点的指针保存到第i-1个结点的指针域中，这样就可以将第i个结点从链表中断开

③ 释放第i个结点所占的空间



从单链表中删除第i个结点

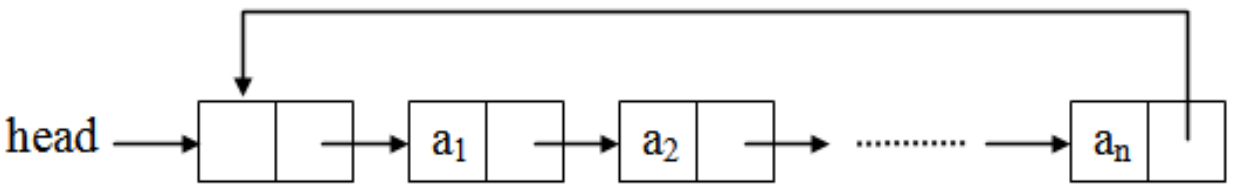
```
void Delete(LNode *head，int i )
{
        if(i<1)
                cout<<”不存在第”<<i<<”个元素”;
        else {
                LNode *p=head;                          //p指向头结点(第0个结点)
                LNode *q;                               //q和p最终分别指向第i-1和第i个结点
                int k=0;
                while( p!=NULL&&k<i )
                {
                        q=p;
                        p=p->next;
                        k++;
                }
                if(p==NULL)
                        cout<< i<<”超出链表长度”;
                else {
                        q->next=p->next;                //从链表中删除该结点
                        delete p;                       //释放结点p
                }
        }
}
```

查找链表中的结点

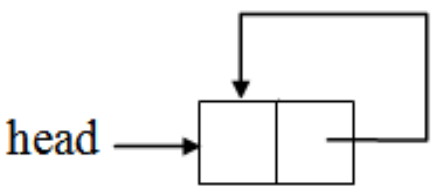
```
LNode* Find(LNode *head, ElemType x )
{
        LNode *p=head->next;          //p指向第一个元素所在结点
        while ( p!=NULL && p->data!=x )
                p = p->next;
        return p;
}
```

其它形式的链表

- **单向循环链表**
  - 将单链表尾结点的指针由NULL改为指向头结点，首尾连接形成一个环形，简称为**循环链表**



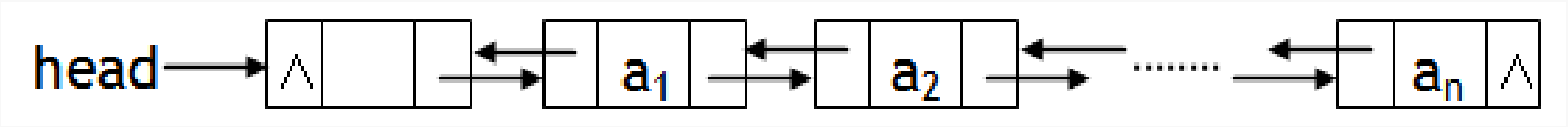
(a) 带有头节点的循环链表



(b) 空循环链表

其它形式的链表

- **双向链表**
  - 每个结点的指针域中再增加一个指针，使其指向该结点的直接前趋结点
  - 这样构成的链表中有两个不同方向的链



其它形式的链表

- **双向循环链表**
  - 将双向链表的头结点的前趋指针指向尾结点
  - 将尾结点的后继指针指向头结点