**洒水a**

**数据结构与算法之——线性表**

**本教程为本人亲自整理，结合大学及各种习题资料汇总而成**

**版权所有，转载请注明出处，谢谢合作！**

****安伟超

达内时代科技集团有限公司

天津天大中心教学部

[anwc@tedu.cn](mailto:anwc@tedu.cn)

17600945626

最后修改：2019.6.4

# 线性表的基本概念

## 线性表的定义

线性表（linear list）是数据结构的一种，一个线性表是n个具有相同特性的数据元素的有限序列。数据元素是一个抽象的符号，其具体含义在不同的情况下一般不同。线性表在数学概念中，其本质是一个集合。

## 线性表的表示

一般地，我们用如下的公式来广义的表达一个线性表：

线性表的长度：序列中所含元素的个数——n，n≥0

空表：当n=0时，表示线性表是一个空表，即表中不包含任何元素。

前驱：是的前驱，2≤i≤n

后继：是的后继，1≤i≤n-1

表头元素：表中第一个元素

表尾元素：最后一个元素

## 线性表的二元组表示法

L = (D, R)

D = { | ∈ElemType, i=1,2,…,n, n≧0 } //ElemType为类型标识符

R = {r}

r = { | , ∈D, i=2,3,…,n }

## 线性表的抽象数据类型

ADT List {  
 数据对象：D = {ai | ai∈ElemType, i=1,2,…,n, n≧0 } //ElemType为类型标识符  
 数据关系：R = {<ai-1, ai> | ai-1, ai∈D, i=2,3,…,n }  
 数据操作：  
 （1）初始化线性表InitList(&L)：构造一个空的线性表L  
 （2）销毁线性表DestroyList(&L)：释放线性表L占用的内存空间  
 （3）判线性表是否为空表ListEmpty(L)：若L为空表，则返回真，否则返回假  
 （4）求线性表的长度ListLength(L)：返回L中元素个数  
 （5）输出线性表DispList(L):当线性表L不为空时，顺序显示L中各节点的值域  
 （6）求线性表L中指定位置的某个数据元素GetElem(L,i,&e)：用e返回L中第 i 个元素的值  
 （7）查找元素LocateElem(L,e)：返回线性表L中第1个与e相等的序号，找不到返回0  
 （8）插入元素ListInsert(&L, i, &e)：在线性表L中的第i个位置插入元素e；  
 （9）删除元素ListDelete(&L, i, &e)：在线性表L中删除第i个元素，有e返回删除的值；  
}

## 线性表相关算法举例

算法2.1：已知线性表LA和LB中的数据元素按值非递减有序排列，现要求将LA和LB归并为一个新的线性表LC，且LC中的数据元素仍然按值非递减有序排列。写出算法过程。

from random import random  
from math import floor  
from copy import copy  
def makeList():  
 l = []  
 while True:  
 num = floor(random()\*10 + 1)  
 l.append(num)  
 if len(l) == 10:  
 break  
 return l  
  
def bubble\_sort(l):  
 for i in range(len(l) - 1):  
 for j in range(len(l) - 1 - i):  
 if l[j] > l[j + 1]:  
 l[j], l[j + 1] = l[j + 1], l[j]  
 return l  
  
  
def merge\_sort(l1,l2):  
 l1\_copy = copy(l1)  
 l2\_copy = copy(l2)  
 l3 = []  
 while l1\_copy and l2\_copy:  
 if l1\_copy[0] < l2\_copy[0]:  
 l3.append(l1\_copy[0])  
 del l1\_copy[0]  
 else:  
 l3.append(l2\_copy[0])  
 del l2\_copy[0]  
 if l1\_copy:  
 l3.extend(l1\_copy)  
 if l2\_copy:  
 l3.extend(l2\_copy)  
 return l3  
  
  
  
def test():  
 list1 = bubble\_sort(makeList())  
 list2 = bubble\_sort(makeList())  
 res = merge\_sort(list1,list2)  
 print(**'有序序列l1：%s'**%list1)  
 print(**'有序序列l2：%s'**%list2)  
 print(**'归并序列13：%s'**%res)  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 test()

# 线性表的顺序存储结构

## 定义

线性表的顺序存储是指，用一组地址连续的存储单元依次存储线性表的数据元素。

## 线性表的顺序存储原理

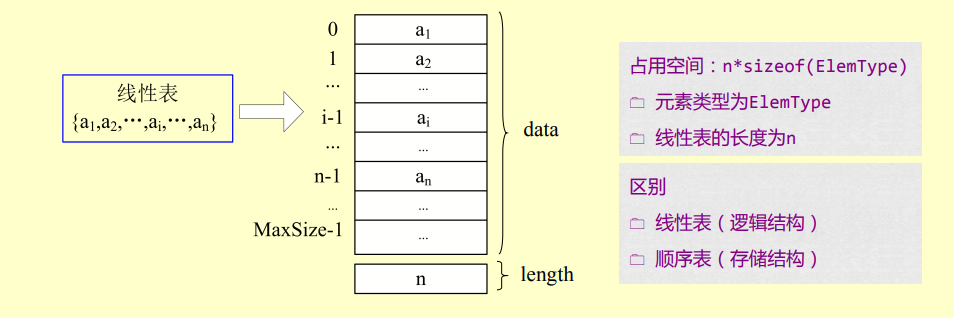
假设线性表的每个元素需占用那个l个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第个数据元素的存储位置和第个数据元素之间满足下列关系：

一般来说，线性表的第个数据元素的存储位置为：

上式中的是线性表的第一个数据元素的存储位置，通常称做线性表的起始位置或基地址。

线性表的每一个数据元素的存储位置都和线性表的起始位置相差一个和数据元素在线性表中的位序成正比的常数。由此，只要确定了存储线性表的起始位置，线性表中任一数据元素都可以随机存取。所以线性表的顺序存储结构是一种随机存取的存储结构。

## 线性表的顺序存储示意图



## 使用Python实现线性表的顺序存储——列表

class List(object):  
 def \_\_init\_\_(self,size=20):  
 self.size = size  
 self.items = [None] \* size  
  
 def \_\_getitem\_\_(self,index):  
 return self.items[index]  
  
 def \_\_setitem\_\_(self,index,value):  
 self.items[index] = value  
  
 def \_\_len\_\_(self):  
 return self.size  
  
 def clear(self,value=None):  
 for i in range(len(self.size)):  
 self.items[i] = value  
  
 def \_\_iter\_\_(self):  
 for item in self.items:  
 yield item  
  
def test():  
 size = 10  
 l = List(size)  
 l[0] = 1  
 assert l[0] == 1  
  
 l.clear()  
 assert len(l) is None  
   
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 test()

## 线性表的顺序存储基本运算的实现

### 初始化线性表

运算实现的功能：构造一个空的线性表

方法：

1. 分配空间
2. 将length成员设置为0
3. 算法实现(C):

void InitList(SqList \*&L) {//指针的引用

L=(SqList \*)malloc(sizeof(SqList)); //赋值号左边L是一个SqList \*指针，指向名为SqList的结构体；赋值号右边，malloc(sizeof(SqList))是分配一块大小为sizeof(SqList)的内存，并返回首地址并赋值给左边的L指针，(SqList \*)表示把这个地址强制转化为SqlList \*的指针。

L->length=0;

}

1. 算法实现(Python)：

class List(object):  
 def \_\_init\_\_(self,size):  
 self.size = size  
 self.items = 0

时间复杂度：O(1)

### 销毁线性表

运算实现的功能：释放线性表L占用的内存空间

方法：调用free函数(C) & 调用del函数(Python)

算法实现(C)：

void DestroyList(SqList \*&L) {

free(L);

}

算法实现(Python)：

del(l)

时间复杂度：O(1)

### 判断是否为空表

运算实现的功能：返回一个值表示L是否为空表。若为空表，返回True，否则返回False

方法：运用获取长度的方法进行判断

算法实现(C):

bool ListEmpty(SqList \*L) {

return(L->length==0);

}

算法实现(Python):

def isEmpty(l):  
 if len(l) == 0:  
 return True  
 else:  
 return False

时间复杂度：O(1)

### 获得线性表的长度

运算实现的功能：返回顺序表l的长度

方法：即返回length成员的值即可

算法实现(C):

int ListLength(SqList \*L) {

return(L->length);

}

算法实现(Python):

class List(object):

def \_\_len\_\_(self):

return self.size

时间复杂度：O(1)

### 输出线性表

运算实现的功能：该运算当线性表L不为空时，顺序显示L中各个元素的值

方法：遍历线性表并顺序输出每一个元素即可。

算法实现(C):

void DispList(SqList \*L) {

int i;

if (ListEmpty(L))

return;

for (i=0; ilength; i++)

printf("%d ",L->data[i]);

printf("\n");

}

算法实现(Python):

class List(object):  
 def printf(self):  
 for i in range(len(self.size)):  
 print(i)

时间复杂度：O(1)或O(n)

### 求某个数据元素值

运算实现的功能：返回L中的第i个元素的值，存放在变量中

方法：索引取值

算法实现(C):

bool GetElem(SqList \*L, int i, ElemType &e) {

if (iL->length)

return false;

e=L->data[i-1];

return true;

}

算法实现(Python):

class List(object):  
 def \_\_getitem\_\_(self,index):  
 return self.items[index]  
 self.size = size

时间复杂度：O(1)

### 按元素值查找

运算实现的功能：查找第1个值域与e相等的元素的逻辑位序。若这样的元素不存在，则返回值为0

方法：匹配值相等的元素，返回其索引值

算法实现(C):

int LocateElem(SqList \*L, ElemType e) {

int i=0;

while (ilength && L->data[i]!=e)

i++;

if (i>=L->length)

return 0;

else

return i+1;

}

算法实现(Python):

class List(object):  
 def \_\_Locateitem\_\_(self,value):  
 if self.items[i] == value:  
 return i

时间复杂度：O(1)或O(n)

### 插入数据元素

#### 运算实现的功能

在顺序表L的第i（1≤i≤ListLength(L)+1） 个位置上插入新的元素e。

#### 方法

1.将顺序表原来第i个元素及以后元素均后移一个位置

2.腾出一个空位置插入新元素

3.最后顺序表长度增

#### 算法实现

C语言：

bool ListInsert(SqList \*&L, int i, ElemType e) {

int j;

if (iL->length+1)

return false;

i--; //将逻辑序号转化为物理序号

for (j=L->length; j>i; j--) //元素后移

L->data[j]=L->data[j-1];

L->data[i]=e; //腾开的位置插入元素e

L->length++; //顺序表长度增1

return true; //成功插入返回true

}

Python：

def listInsert(l):  
 l.insert(i)

#### 时间复杂度分析

如上代码：

bool ListInsert(SqList \*&L, int i, ElemType e) {

int j;

if (iL->length+1)

return false;

i--; //将逻辑序号转化为物理序号

**for (j=L->length; j>i; j--) //元素后移**

**L->data[j]=L->data[j-1];**

L->data[i]=e; //腾开的位置插入元素e

L->length++; //顺序表长度增1

return true; //成功插入返回true

}

从代码中可以看到，算法核心逻辑为for循环内部，那么该for循环会有以下情况可能出现：

最佳情况：移动0次

最差情况：移动n次

基于此得初步结论：移动次数与插入位置有关

设在第i个位置上插入元素的概率为,在长度为n的线性表中插入一个元素是所需的移动元素的平均次数为:

所以，插入操作的时间复杂度为：

### 删除数据元素

运算实现的功能：删除顺序表L的第i个元素 （1≤i≤ListLength(L)）

方法：

1. 将线性表第i个元素以后元素均向 前移动一个位置，覆盖了原来的第 i个元素，达到删除该元素的目的
2. 最后顺序表长度减1

算法实现(C):

bool ListDelete(SqList \*&L,int i,ElemType &e) {

int j;

if (iL->length)

return false;

i--;

e=L->data[i];

for (j=i; jlength-1; j++)

L->data[j]=L->data[j+1];

L->length--;

return true;

}

算法实现(Python):

l.remove(i)

时间复杂度分析：

# 线性表的链式存储结构

## 定义

线性表的链式存储是指：用一组任意的存储单元存储线性表的数据元素（这组存储单元可以是连续的，也可以是不连续的）

## 线性表的链式存储原理

为了表示每个数据元素与其直接后继数据元素之间的逻辑关系，对数据元素来说，除了存储其本身的信息之外，还需存储一个指示其直接后继的信息（即直接后继的存储位置）。这两部分信息组成的数据元素的存储映像，称为结点。它包括两个域：其中存储数据元素信息的域称为数据域；存储直接后继存储位置的域称为指针域。指针域中存储的信息称为指针或链。n个结点链结成一个链表，即为线性表：

的链式存储结构。又由于，此链式表的每个结点中只包含一个指针域，故又称线性链表或单链表。

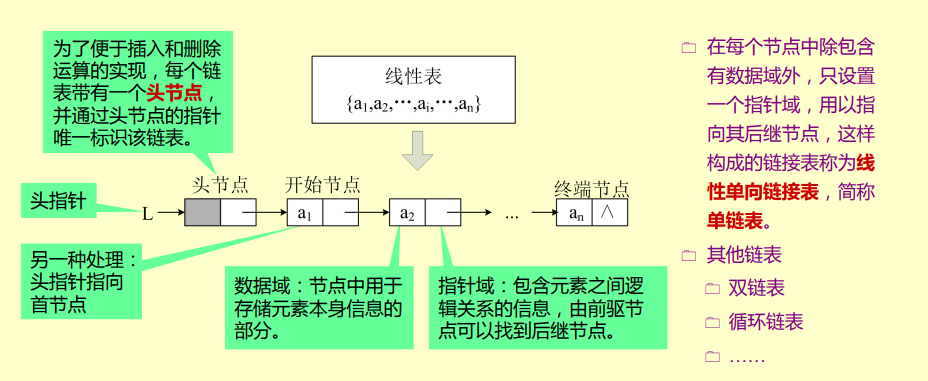
线性表的链式存储的存取必须从头指针开始进行，头指针指示链表中的第一个结点（即第一个数据元素的存储映像）的存储位置。同时，由于最后一个数据元素没有直接后继，则线性链表中最后一个结点的指针为空(NULL).

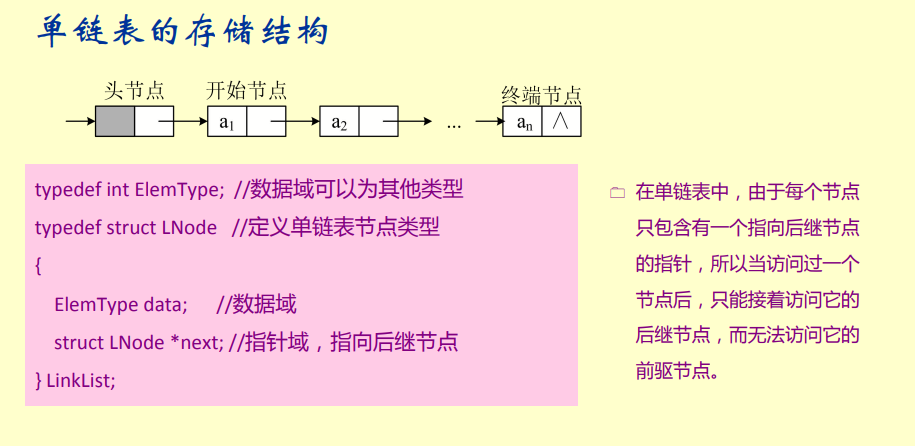
用线性链表表示线性表时，数据元素之间的逻辑关系是由结点中的指针指示的。换句话说，指针为数据元素之间的逻辑关系的映像，则逻辑上相邻的两个数据元素其存储的物理位置不要求紧邻，由此，这种存储结构为非顺序映像或链式映像。

若线性表为空表，则头结点的指针域为空(NULL).

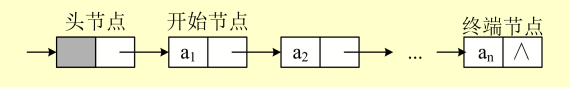
在单链表中，取得第i个数据元素必须从头指针出发寻找，因此，单链表是非随机存储的存储结构。

## 线性表的链式存储示意图





## 线性表的链式存储密度

 现有如下单链表：

现引入存储密度的概念

存储密度：节点数据本身所占的存储量和整个节点结构中所占的存储量之比。

存储密度的意义：存储密度越大，存储空间的利用率就越高。

比较：

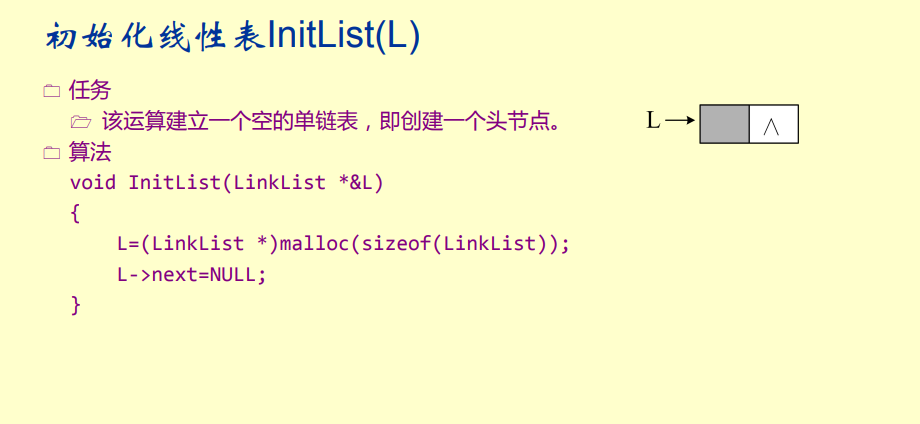
1. 顺序表的存储密度为1（若不考虑顺序表中的空闲区）
2. 链表的存储密度小于1
   1. 若单链表的节点数据均为整数，指针所占的空间和整数相同，则单链表的存储密度为50％
   2. 数据域所占空间越多，存储密度越高

## 使用Python实现线性表的链式存储——链表

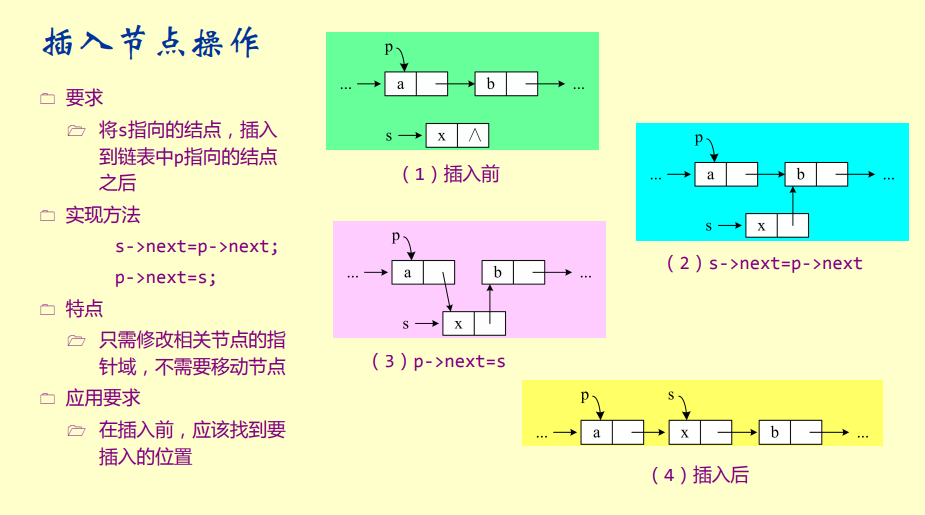
*"""  
链表程序实现  
重点代码  
  
思路分析  
1. 创建节点类，生成节点对象  
　　包含数据和下一个节点的引用  
2. 链表类，生成链表对象  
　　可以对链表进行数据操作  
"""  
  
# 节点类*class Node:  
 def \_\_init\_\_(self, data, next=None):  
 self.data = data  
 self.next = next  
  
  
*# 链表类*class Linklist:  
 *"""  
 建立链表模型  
 进行链表操作  
 """* def \_\_init\_\_(self):  
 *"""  
 初始化链表,生成一个头节点,表示链表开始节点  
 """* self.head = Node(None)  
  
 *# 初始添加一组链表节点* def init\_list(self, list\_):  
 p = self.head *# ｐ为移动变量* for i in list\_:  
 p.next = Node(i)  
 p = p.next *# ｐ向后移动一个节点  
  
 # 遍历链表* def show(self):  
 p = self.head.next *# 第一个有效节点* while p is not None:  
 print(p.data, end=**' '**)  
 p = p.next  
 print() *# 换行  
  
 # 获取链表长度* def get\_length(self):  
 n = 0  
 p = self.head  
 while p.next is not None:  
 n += 1  
 p = p.next  
 return n  
  
 *# 判断链表是否为空* def is\_empty(self):  
 if self.get\_length() == 0:  
 return True  
 else:  
 return False  
  
 *# 清空链表* def clear(self):  
 self.head.next = None  
  
 *#　尾部插入节点* def append(self,data):  
 node = Node(data) *#　生成新节点* p = self.head  
 while p.next is not None:  
 p = p.next  
 p.next = node  
  
 *#　选择位置插入节点* def insert(self,index,data):  
 if index < 0 or index > self.get\_length():  
 raise IndexError(**"index out of range"**)  
  
 *# 定义ｐ移动到插入位置的前一个* p = self.head  
 for i in range(index):  
 p = p.next  
  
 node = Node(data) *#　生成节点  
  
 # 插入* node.next = p.next  
 p.next = node  
  
 *# 删除节点* def delete(self,data):  
 p = self.head  
 while p.next and p.next.data != data:  
 p = p.next  
 *#　判断ｗｈｉｌｅ循环结束原因* if p.next is None:  
 raise ValueError(**"value is error"**)  
 else:  
 p.next = p.next.next  
  
 *# 获取节点值* def get\_item(self,index):  
 if index < 0 or index >= self.get\_length():  
 raise IndexError(**"index out of range"**)  
 p = self.head.next  
 for i in range(index):  
 p = p.next  
 return p.data  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 *# 链表对象　　  
 # link-->head  
 # link.head --> data == None  
 # link.head --> next == None* link = Linklist()  
 l = [1, 2, 3, 4, 5]  
 link.init\_list(l)  
 link.show() *#　遍历数据* print(**"Link length:"**,link.get\_length())  
 link.append(6)  
 link.insert(4,100)  
 link.show()  
 link.delete(3)  
 link.show()  
 print(link.get\_item(3))

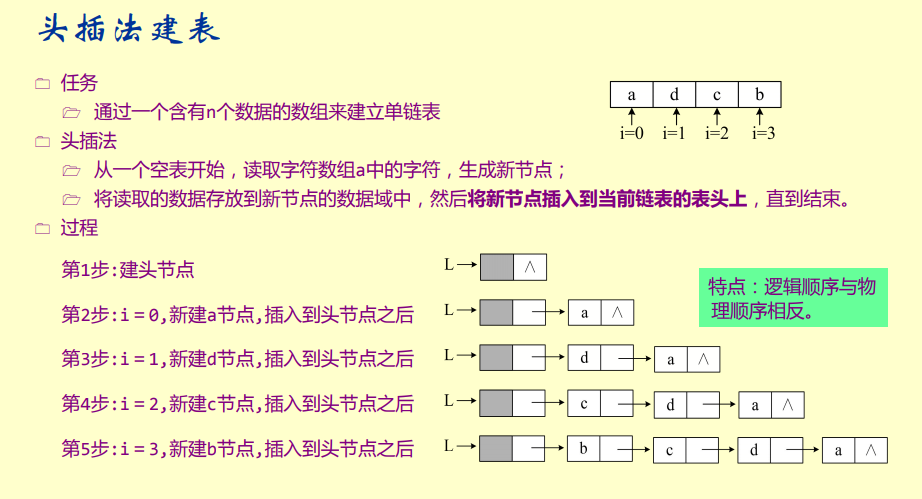
## 线性表的链式存储基本运算的实现

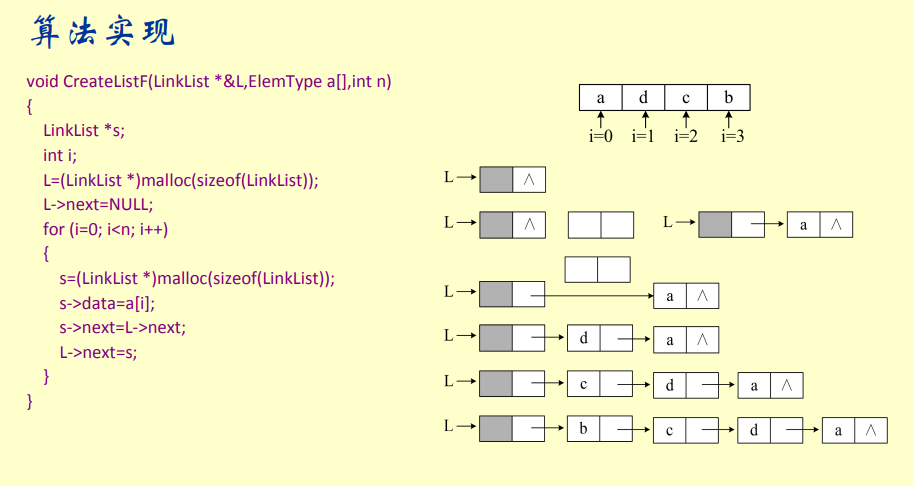
### 初始化线性表



### 插入结点操作

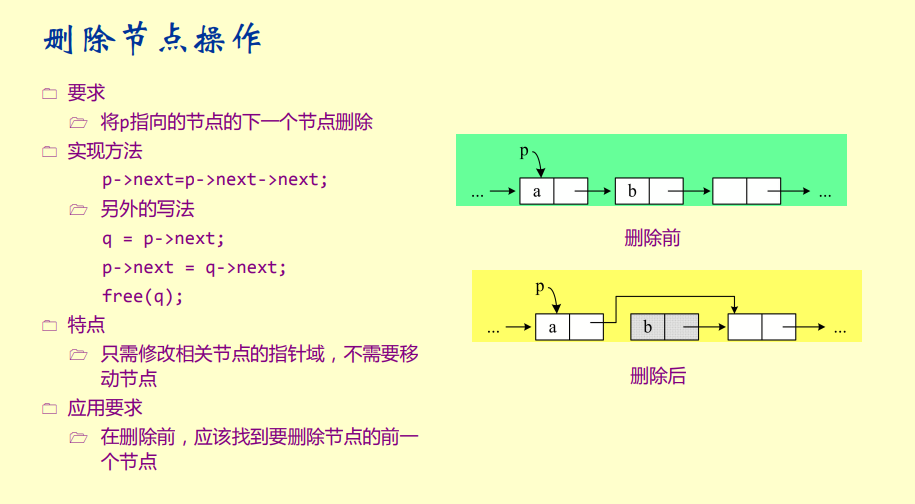




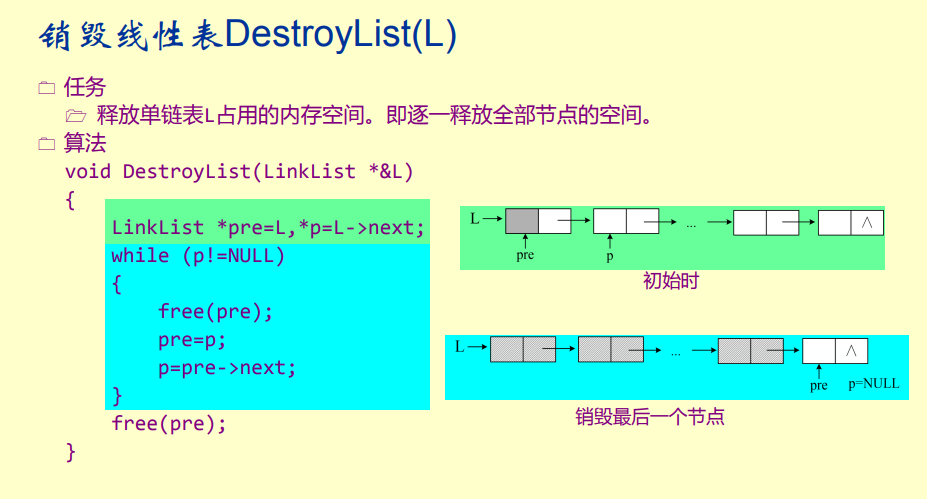




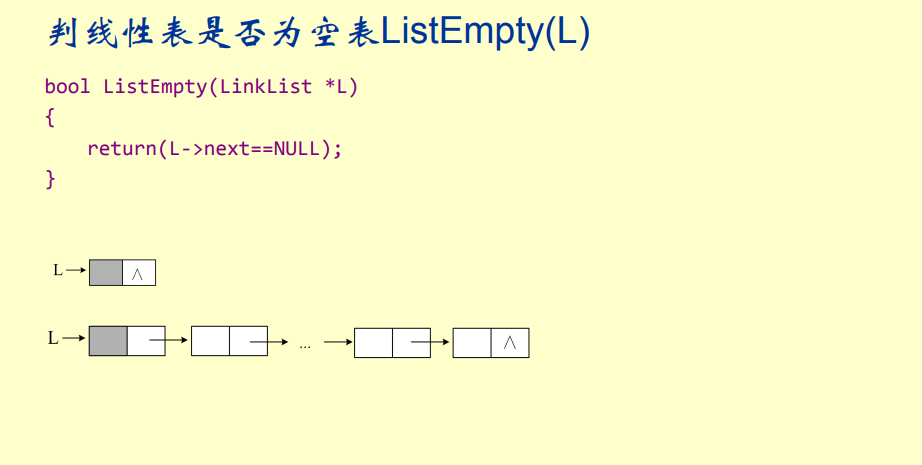
### 删除结点操作



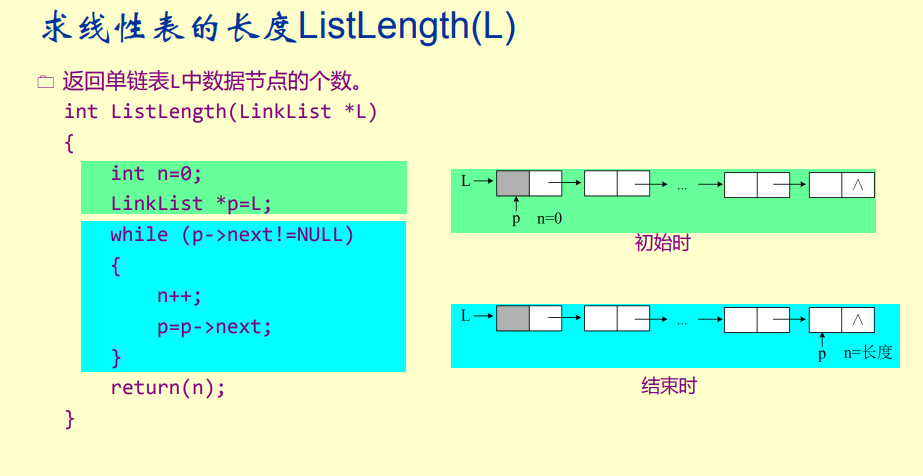
### 销毁单链表



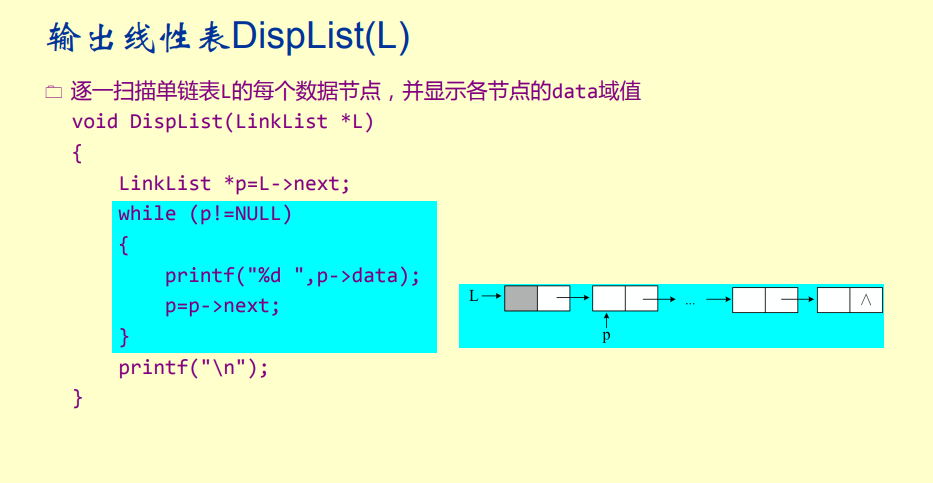
### 判断单链表是否为空



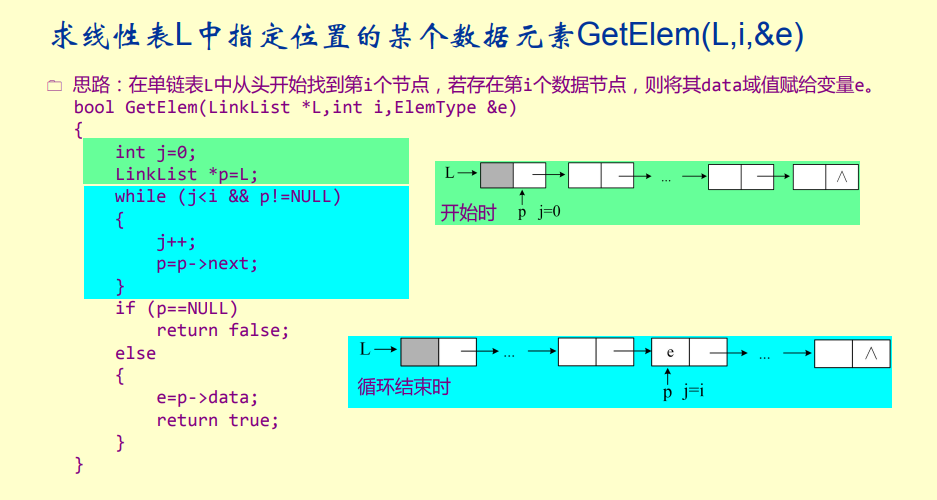
### 求单链表的长度



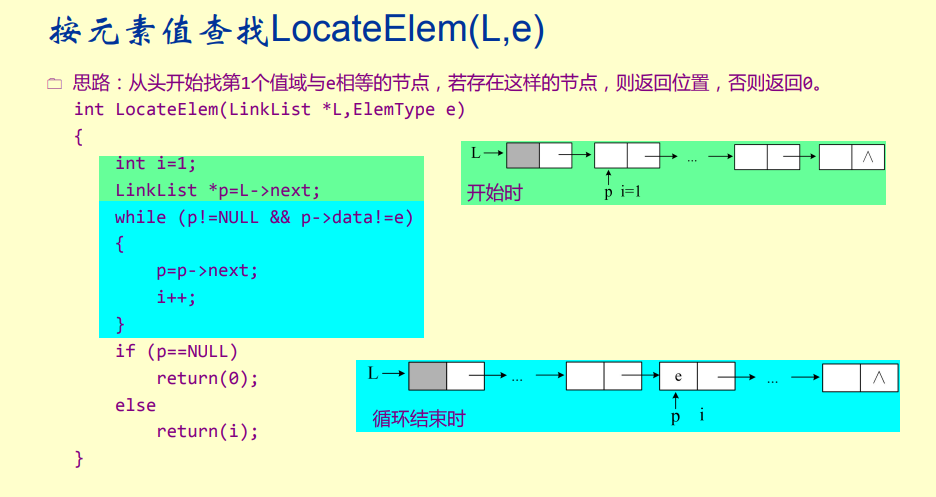
### 输出单链表



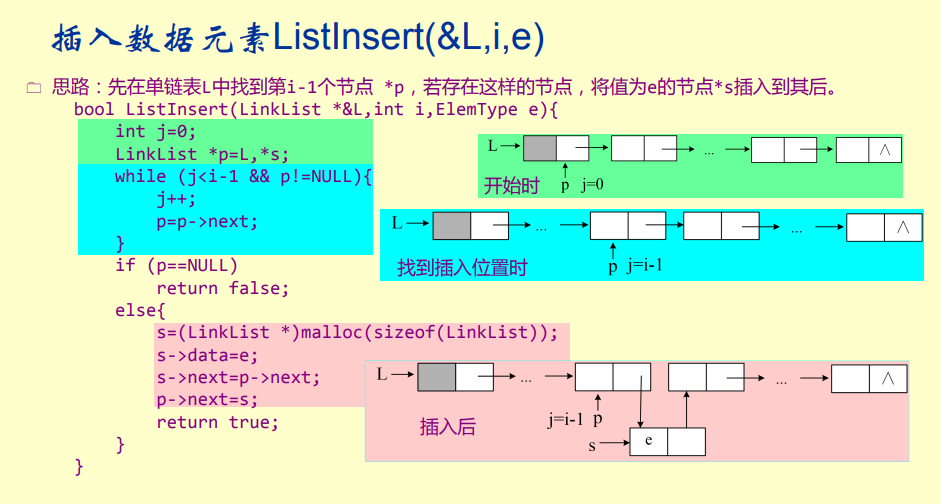
### 求线性表L中指定位置的某个数据元素



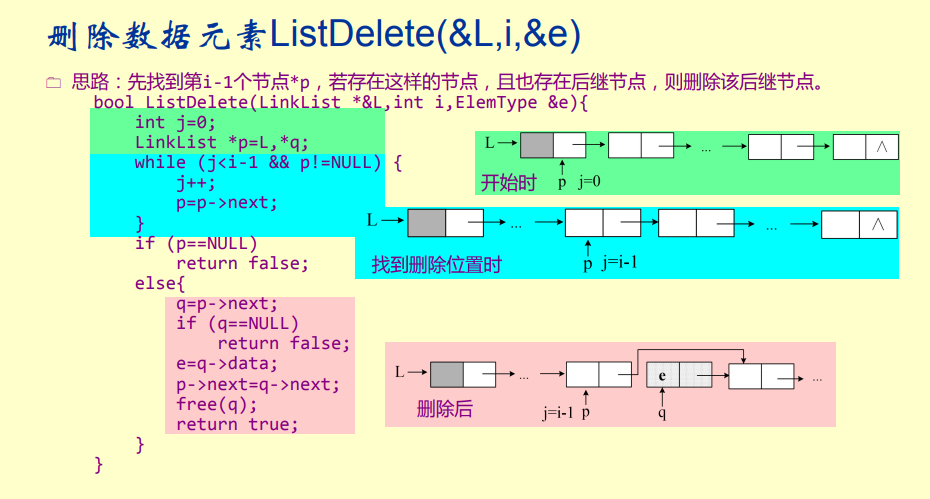
### 按元素值查找



### 插入数据元素



### 删除数据元素



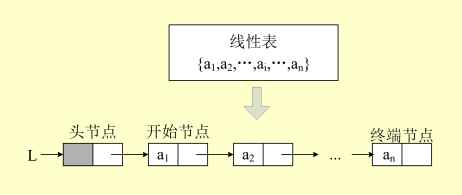
# 线性表的链式存储结构——双向链表

在此之前我们讨论的链式存储结构的结点中只有一个指示直接后继的指针域，由此，从某个结点出发只能顺着指针往后查询其他结点。若要查询结点的直接前驱元素，则需从表头指针出发。换句话说，在单链表中，向后查询的时间复杂度为O(1)，而向前查询的时间复杂度为O(n)。为克服单链表这种单向性的缺点，引入双向链表的概念。

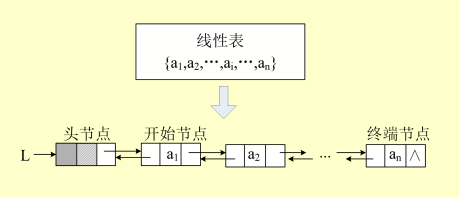
## 定义

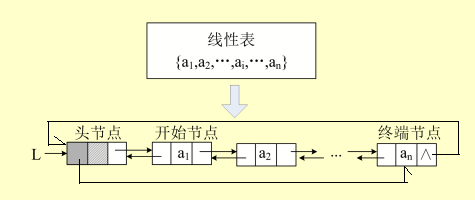
双向链表是指每个结点中有两个指针域，其一指向直接后继，另一指向直接前驱。双向链表又称为线性双向链表

## 双向链表的存储结构及图示

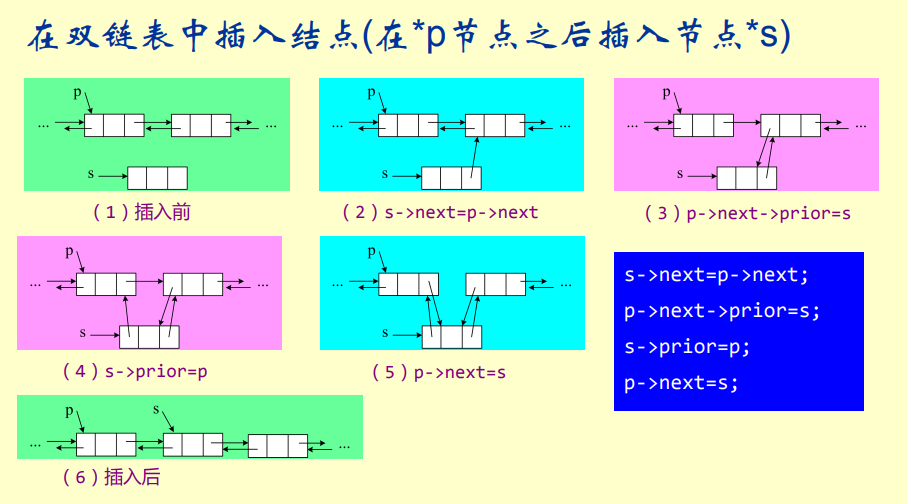
 单链表：

双向链表：

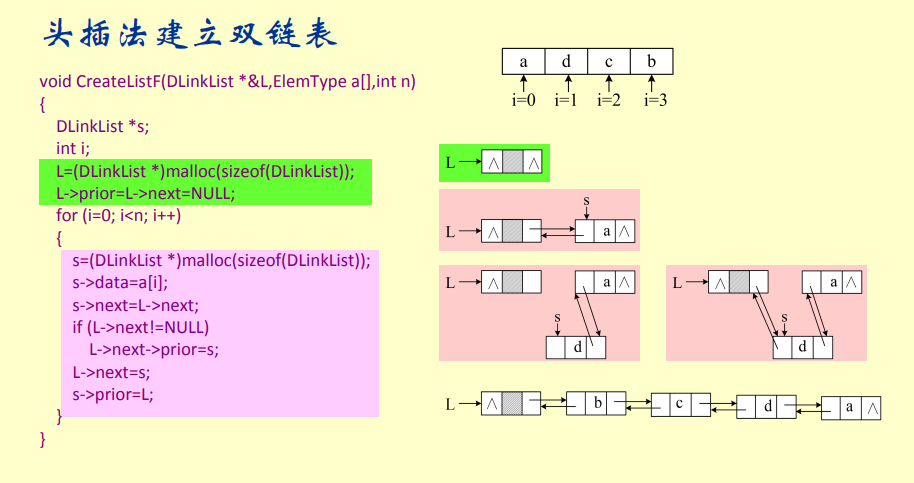


双向链表拓展——双向循环链表：

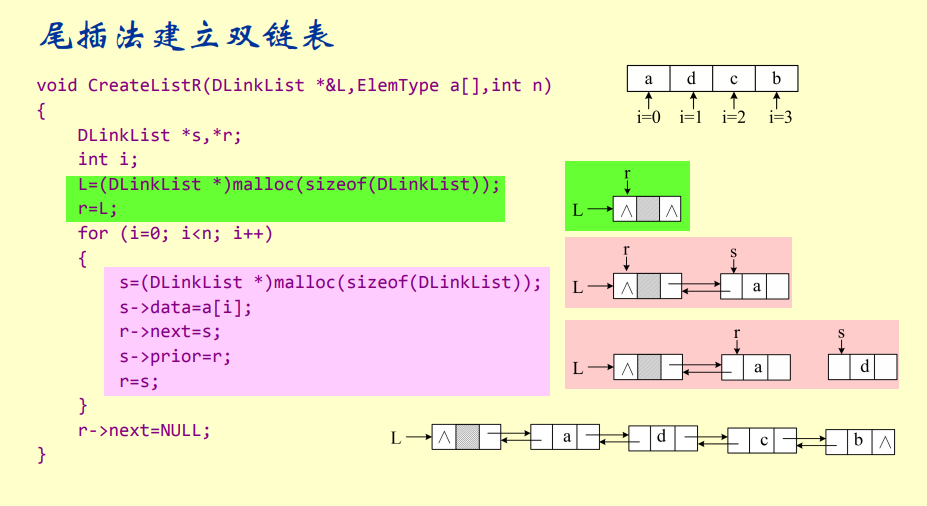
## 在双向链表中插入结点动态图解



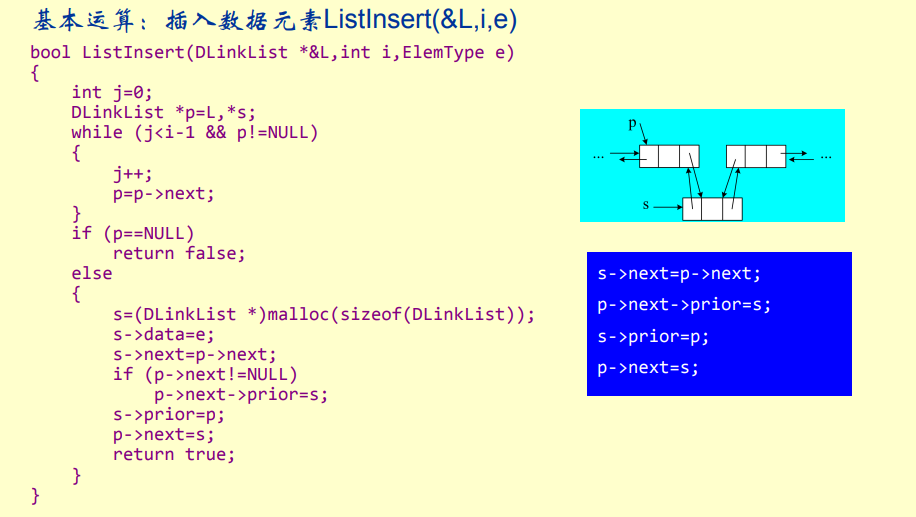
## 头插法建立双链表



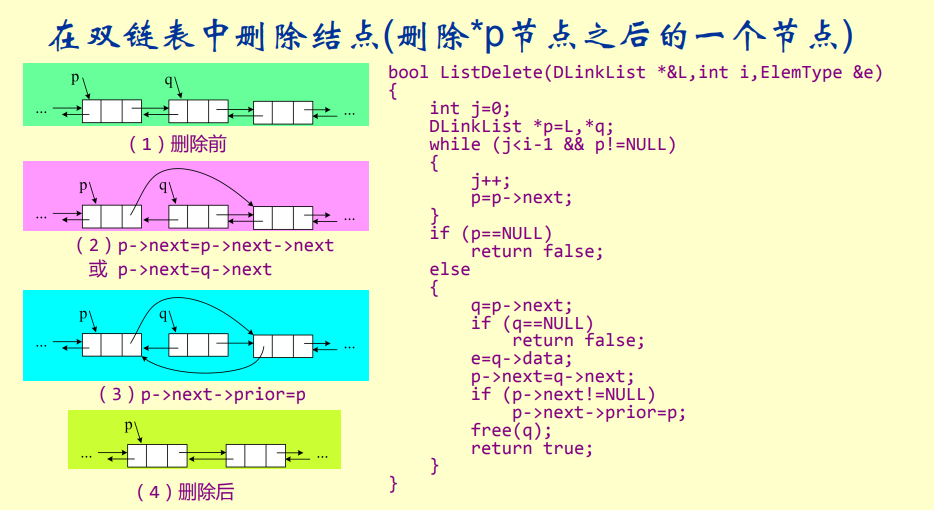
## 尾插法建立双链表



## 双向链表的基本运算：插入数据元素



## 双向链表的基本运算：在双向链表中删除结点



# 线性表的链式存储结构——循环链表

## 定义

循环链表是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个连表白形成一个环。由此，从表中任意结点出发均可找到表中的其他结点。

循环链表的操作和线性链表基本一致，差别仅在于算法中的循环条件不是p或p->next是否为空，而是他们是否等于头指针。

## 带头结点的单向循环链表和双向循环链表的存储结构及图示

