数据结构（大话数据结构）

2018年6月5日

14:56

**第一章 数据结构绪论**

**数据结构是相互之间存在一种或多种特定关系的数据元素的集合。**

If you give someone a program，you will frustrate them for a day; if you teach them how to program, you will frustrate them for a life time.（如果你交给某人一个程序，你将折磨他一整天；如果你教某人如何编写程序，你将折磨他一辈子。）

重要概念：**抽象数据类型**

**数据**：是描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合。数据不仅仅包括整型实型等数值类型，还包括字符及声音、图像、视频等非数值类型。

**数据元素**：是组成数据的、有一定意义的基本单位，在计算机中通常作为整体处理。也被称为记录。

**数据项**：一个数据元素可以由若干个数据项组成。比如人这样的数据元素，可以有眼、耳、鼻、嘴、手、脚这些数据项，也可以有姓名、年龄、性别、出生地址、联系电话等数据项，具体有哪些数据项，要视你做的系统来决定。

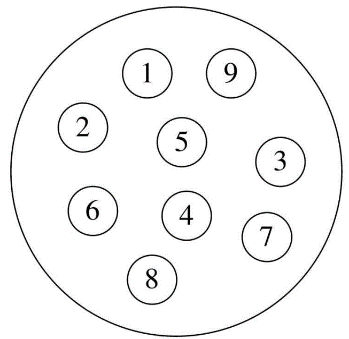
数据项是数据不可分割的最小单位。在数据结构这门课程中，我们把数据项定义为最小单位，是有助于我们更好地解决问题。所以，记住了，数据项是数据的最小单位。但真正讨论问题时，数据元素才是数据结构中建立数据模型的着眼点。

**数据对象**：是性质相同的数据元素的集合，是数据的子集。

按照视点的不同，我们把数据结构分为**逻辑结构**和**物理结构**。

**1）逻辑结构**：是指数据对象中数据元素之间的相互关系。其实这也是我们今后最需要关注的问题。逻辑结构分为以下四种：

1、**集合结构**：集合结构中的数据元素除了同属于一个集合外，它们之间没有其他关系。



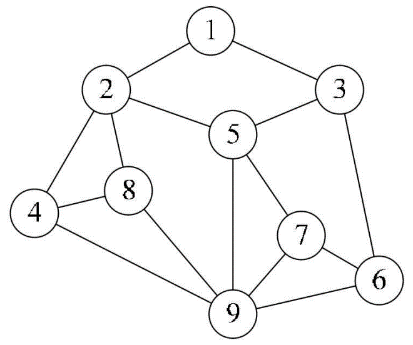
2、**线性结构**：线性结构中的数据元素之间是一对一的关系。

1 
2 
3 
4 
5 
6 

3、**树形结构**：树形结构中的数据元素之间存在一种一对多的层次关系。

F 
G 
H 
ı 

4、**图形结构**：图形结构的数据元素是多对多的关系。



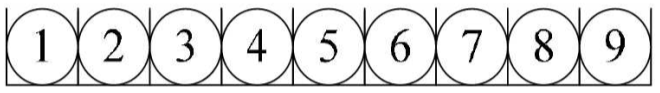
说完了**逻辑结构**，我们再来说说数据的**物理结构**（很多书中也叫做**存储结构**，你只要在理解上把它们当一回事就可以了。

**2）物理结构**：是指数据的逻辑结构在计算机中的存储形式

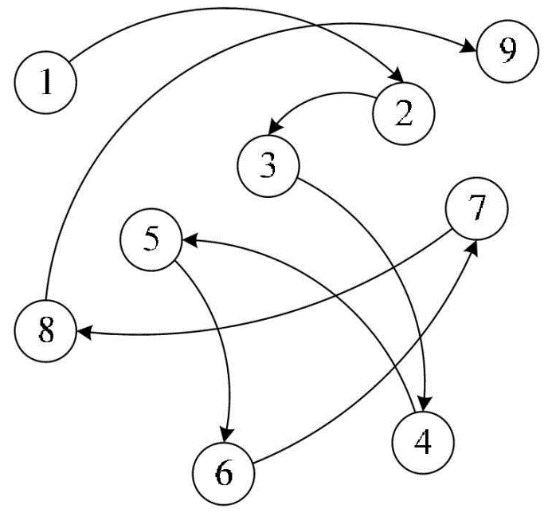
数据是数据元素的集合，那么根据物理结构的定义，实际上就是如何把数据元素存储到计算机的存储器中。存储器主要是针对内存而言的，像硬盘、软盘、光盘等外部存储器的数据组织通常用文件结构来描述。

数据元素的存储结构形式有两种：**顺序存储**和**链式存储**。

**顺序存储结构**：是把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的。



**链式存储结构**：是把数据元素存放在任意的存储单元里，这组存储单元可以是连续的，也可以是不连续的。数据元素的存储关系并不能反映其逻辑关系，因此需要用一个指针存放数据元素的地址，这样通过地址就可以找到相关联数据元素的位置。



**抽象数据类型**

**数据类型**：是指一组性质相同的值的集合及定义在此集合上的一些操作的总称。

在C语言中，按照取值的不同，数据类型可以分为两类：

**原子类型**：是不可以再分解的基本类型，包括整型、实型、字符型等。

**结构类型**：由若干个类型组合而成，是可以再分解的。

eg：整型数组是由若干整型数据组成的。

抽象是指抽取出事物具有的普遍性的本质。它是抽出问题的特征而忽略非本质的细节，是对具体事物的一个概括。抽象是种思考问题的方式，它隐藏了繁杂的细节，只保留实现目标所必需的信息。

我们对已有的数据类型进行抽象，就有了抽象数据类型。

**抽象数据类型**（Abstract Data Type，ADT）：是指一个数学模型及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何表示和实现无关比如刚才的例子，各个计算机，不管是大型机、小型机、PC平板电脑、PDA，甚至智能手机都拥有“整数”类型，也需要整数间的运算，那么整型其实就是一个抽象数据类型，尽管它在上面提到的这些在不同计算机中实现方法上可能不一样，但由于其定义的数学特性相同，在计算机编程者看来，它们都是相同的。因此，“抽象”的意义在于数据类型的数学抽象特性。不仅仅指已定义的还可以指计算机编程者自己定义的数据类型。eg：point数据类型 （坐标定义）

为了便于在之后的讲解中对抽象数据类型进行规范的描述，我们给出了描述抽象数据类型的标准格式：

ADT

抽象数据类型名

Data

数据元素之间逻辑关系的定义

Operation

操作1

初始条件

操作结果描述

操作2

……

操作n

……

endADT

**总结回顾：**

数 据 
数 据 对 象 
数 据 元 素 
数 据 项 1 数 据 项 2 
数 掘 元 素 
数 据 项 1 数 巛 唢 2 
数 据 元 素 
数 据 项 1 数 撫 唢 2 
数 据 元 素 
数 据 项 1 数 据 2 

今天首先用我一个不争气的学生为例子，说明数据结构很重要。接着讲了数据结构的起源，说白了，就是一老外，觉得编程这玩意儿不弄得复杂点，不能证明他厉害，所以推出“数据结构”这一课程，让所有学编程的人“享受它带来的乐趣”或者“体验被折磨后无尽的烦恼”。

然后是数据结构的定义：

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。

同样是结构，从不同角度来讨论会有不同的分类：

逻 辑 结 构 
集 合 结 构 
· 线 性 结 构 
· 树 形 结 构 
· 图 形 结 构 
物 理 结 构 
· 顺 序 存 储 结 构 
． 链 接 存 储 结 构 

之后，我们还介绍了抽象数据类型及它的描述方法。

**结尾语**

最后，我想对那些已经开始自学数据结构的同学说，可能你们会困惑、不懂、不理解、不会应用，甚至不知所云。可实际上，无论学什么，都是要努力才可以学到真东西。只有真正掌握技术的人，才有可能去享用它。如果你中途放弃了，之前所有的努力和付出都会变得没有价值。学会游泳难吗？掌握英语口语难吗？可能是难，但在掌握了的人眼里，这根本不算什么，“就那么回事呀”。只要你相信自己一定可以学得会、学得好，**既然无数人已经掌握了，你凭什么不行**。

最终的结果一定是，你对着别人很牛地说：“数据结构—一就那么回事。”

哎，我如此口干舌燥地投众位所好，怎么还有人打瞌睡呢？罢了罢了，下课。

**第二章**

**算法**是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。

为了解决某个或某类问题，需要把指令表示成一定的操作序列，操作序列包括一组操作，每一个操作都完成特定的功能，这就是算法了。

**算法具有五个基本特性：输入、输出、有穷性、确定性和可行性。**

**输入输出：**

输入和输出特性比较容易理解， 算法**具有零个或多个输入**。尽管对于绝大多数算法来说，输入参数都是必要的，但对于个别情况，如打印“hello world！”这样的代码，不需要任何输入参数，因此算法的输入可以是零个。**算法至少有一个或多个输出**，算法是一定需要输出的，不需要输出，你用这个算法干吗？输出的形式可以是打印输出，也可以是返回一个或多个值等。

**有穷性：**

指算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每一个步骤在可接受的时间内完成。

eg：现实中经常会写出死循环的代码，这就是不满足有穷性。

**确定性：**

算法的每一步骤都具有确定的含义，不会出现二义性。算法在一定条件下，只有一条执行路径，相同的输入只能有唯一的输出结果。算法的每个步骤被精确定义而无歧义。

**可行性：**

算法的每一步都必须是可行的，也就是说，每一步都能够通过执行有限次数完成。可行性意味着算法可以转换为程序上机运行，并得到正确的结果。

**算法设计的要求**

**1、正确性**：算法的正确性是指算法至少应该具有输入、输出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案。

但是算法的“正确”通常在用法上有很大的差别，大体分为以下四个层次。

1.算法程序没有语法错误。

2.算法程序对于合法的输入数据能够产生满足要求的输出结果。

3.算法程序对于非法的输入数据能够得出满足规格说明的结果。

4.算法程序对于精心选择的，甚至刁难的测试数据都有满足要求的输出。

**2、可读性**：算法设计的另一目的是为了便于阅读、理解和交流。

**3、健壮性**：当输入数据不合法时，算法也能做出相关处理，而不是产生异常或莫名其妙的结果。

**4、设计算法应该尽量满足时间效率高和存储量低的需求。**

**算法效率的度量方法**

1、事后统计法（不采纳）

这种方法主要是通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定算法效率的高低。

2、事前分析估算方法

在计算机程序编制前，依据统计方法对算法进行估算。

经过分析，我们发现，一个用高级程序语言编写的程序在计算机上运行时所消耗的时间取决于下列因素：

1.算法采用的策略、方法。

2.编译产生的代码质量。

3.问题的输入规模。

4.机器执行指令的速度。

第1条当然是算法好坏的根本，第2条要由软件来支持，第4条要看硬件性能。也就是说，抛开这些与计算机硬件、软件有关的因素，一个程序的运行时间，依赖于算法的好坏和问题的输入规模。所谓问题输入规模是指输入量的多少。

Sum — 
for 
pråntf ( 
int sum = 
sum— (I 
pråntf ( 
100; 
100; 
• n/2; 
/•åkfrl 
/•åkfil ;k•,' 

显然，第-种算法，执行了1+（n+1）+n+1 次=2n+3次；而第二种算法，是1+1+1=3次。事实上两个算法的第-条和最后一条语句是一样的，所以我们关注的代码其实是中间的那部分，我们把循环看作一一个整体，忽略头尾循环判断的开销，那么这两个算法其实就是n次与1次的差距。算法好坏显而易见。

最终，在分析程序的运行时间时，最重要的是把程序看成是独立于程序设计语言的算法或一系列步骤。

11=100 
ΙΙ- 1,000 
η = ΙΟ,ΟΟΟ 
20 000 
2 000 
200 οοο οοο 
20 οοο οοο 
2 οοο οοο οοο οοο 
Η ( 311+1 ) 
3Ι)0Ι 
30 
3 οοο 
( 2η2+3η+1 ) 
20 301 
2 003 001 
200 030 
20 
200 οοο 3000 

这组数据应该就看得很清楚。当n的值越来越大时，你会发现，3n+1已经没法和2n^2的结果相比较，最终几乎可以忽略不计。也就是说，随着n值变得非常大以后，算法G其实已经很趋近于算法I。于是我们可以得到这样-一个结论，判断一个算法的效率时，函数中的常数和其他次要项常常可以忽略，而更应该关注主项（最高阶项）的阶数。

判断一个算法好不好，我们只通过少量的数据是不能做出准确判断的。根据刚才的几个样例，我们发现，如果我们可以对比这几个算法的关键执行次数函数的渐近增长性，基本就可以分析出：某个算法，随着n的增大，它会越来越优于另一算法，或者越来越差于另一算法。这其实就是事前估算方法的理论依据，通过算法时间复杂度来估算算法时间效率。

**算法时间复杂度**

O(n),O(1),O(n^2),O(logn)

推导大O阶：

1.用常数1取代运行时间中的所有加法常数。

2.在修改后的运行次数函数中，只保留最高阶项。

3.如果最高阶项存在且不是1，则去除与这个项相乘的常数。

得到的结果就是大O阶。

下 面 的 这 段 代 码 ， 时 间 复 杂 度 又 是 多 少 呢 ？ 
Count 
（ count < n ） 
0 时 间 复 杂 度 为 0 《 1 ， 的 程 序 步 驟 序 列 
由 于 每 次 count 乘 以 2 之 后 ， 就 距 离 n 更 近 了 一 分 。 也 就 是 说 ， 有 多 少 个 2 相 乘 后 
大 于 n ， 则 会 退 出 循 环 。 由 2、=n 得 到 x=bgzna 所 以 这 个 循 环 的 时 间 复 杂 度 为 0 gn 〕 。 

计算机生成了可选文字:
执行次数函数
12
2n十3
3n+2n+1
5k尽+20
2n+3nlog2n+19
6n42n+3n+4
表2．10．1
阶
0(1)
O囫
0仂2）
0№g
O（。g司
非正式术语
常数阶
线性阶
平方阶
对数阶
nlogzn阶
立方阶
指数阶
常用的时间复杂度所耗费的时间从小到大依次是．
0(1)<O(logn)<O(n)<O(nlogn)<<0(?)<001!)<O(nn)
我们前面已经谈到了0〔1〕常数阶、0gn〕对数阶、0〔n〕线性阶、0〔n2〕平方阶等，
至于0〔g我们将会在今后的课程中介绍，而像0〔113),过大的n都会使得结果变得冫
现实。同样指数阶0〔2n〕和阶乘阶0@〕等除非是很小的n值，否则哪怕n只是100，都
是噩梦般的运行时间。所以这种不切实际的算法时间复杂度，一般我们都不去讨论
匕。

通常，除非特别指定，我们提到的运行时间都是最坏情况的运行时间。

一般在没有特殊说明的情况下，都是指**最坏时间复杂度**。

通常，我们都使用“时间复杂度”来指运行时间的需求，使用“空间复杂度”指空间需求。当不用限定词地使用“复杂度”时，通常都是指时间复杂度。显然我们这本书重点要讲的还是算法的时间复杂度的问题。

算法的**空间复杂度**通过计算算法所需的存储空间实现，算法空间复杂度的计算公式记作：S(n)=O(f(n))，其中，n为问题的规模，f(n)为语句关于n所占存储空间的函数。

若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为O(1)。

也 许 你 就 可 以 深 刻 的 感 受 到 ， 愚 公 移 山 固 然 可 敬 ， 但 发 明 炸 药 和 推 士 机 ， 可 能 史 
加 实 在 和 聪 明 （ 如 图 2 ． 14 ． 1 所 示 ） 。 
愚 公 移 山 
箕 畚 运 
垦 壤 
叩 石 
现 代 移 山 
大 卡 车 运 
挖 掘 机 挖 
推 土 机 推 
烈 火 药 炸 
图 2 ． 凵 0 

**线性表**

线性表：零个或多个数据元素的有限序列。

除第一个元素外，每个元素有且仅有一个**直接前驱，**

除最后一个元素外，每个元素有且仅有一个**直接后驱。**

所以，**线性表的抽象数据类型**定义如下：

**ADT** 线性表(List)

**Data**

线性表的数据对象集合为{al，a2，......., an}

/\*每个元素的类型均为DataType。其中，除第一一个元素a1外，每一个元素有且只有一个直接前驱元素，除了最后一个元素an外，每一个元素有且只有一个直接后继元素。

数据元素之间的关系是一一对一的关系。\*/

**Operation**

InitList(&L) //初始化操作，建立一个空的线性表

ListEmpty(L) //判空操作，若空返回ture，否则返回false

ListLength(L) //求表长，返回线性表的长度，即L中数据元素的个数

GetElem(L, i, &e) //按位查找，用e获取第i位的元素的值

LocateElem(L, e) //按值查找操作

ListInsert(&L, i, e) //插入操作

ListDelete(&L, i, &e) //删除操作，并用e返回元素的值

PrintList(L)

DestoryList(&L) //删除并释放占用的空间

**线性表的顺序存储结构（顺序表）**

线性表的顺序存储的结构代码

顺序存储结构的 元素获取 插入 与 删除 见**List0301.cpp**

**数组长度**与**线性表长度**区别

数组的长度是存放线性表的存储空间的长度，存储分配后这个量一般是不变的。有个别同学可能会问，数组的大小一定不可以变吗？我怎么看到有书中谈到可以动态分配的一维数组。是的，般高级语言，比如C、VB、C++都可以用编程手段实现动态分配数组，不过这会带来性能上的损耗。

线性表的长度是线性表中数据元素的个数，随着线性表插入和删除操作的进行，这个量是变化的。

在任意时刻，线性表的长度应该小于等于数组的长度。

的 当 前 长 度 Leng 山 
下 标 ： 
i-2 
数 组 的 长 度 Ma 泛 e 
空 闲 空 间 

由于每个数据元素，不管它是整型、实型还是字符型，它都是需要占用一定的存储单元空间的。假设占用的是c个存储单元，那么线性表中第i+1个数据元素的存储位置和第i个数据元素的存储位置满足下列关系（LOC表示获得存储位置的函数）。

LOC(ai+1) = LOC(ai) + c

所以对于第i个数据元素ai的存储位置可以由a1推算得出：

LOC(ai) = LOC(a1) + (i-1)\*c

通过这个公式，你可以随时算出线性表中任意位置的地址，不管它是第一个还是最后一个，都是相同的时间。那么我们对每个线性表位置的存入或者取出数据，对于计算机来说都是相等的时间也就是一个常数，因此用我们算法中学到的时间复杂度的概念来说，它的存取时间性能为O(1)。我们通常把具有这一特点的存储结构称为随机存取结构。

插入和删除

至于平均的情况，由于元素插入到第i个位置，或删除第i个元素需要移动n-i个元素。根据概率原理，每个位置插入或删除元素的可能性是相同的，也就说位置靠前，移动元素多，位置靠后，移动元素少。最终平均移动次数和最中间的那个元素的移动次数相等，为（n-1）/2。

我们前面讨论过时间复杂度的推导，可以得出，平均时间复杂度还是O(n)。

这说明什么？线性表的顺序存储结构，在存、读数据时，不管是哪个位置，时间复杂度都是O(1)；而插入或删除时，时间复杂度都是O(n)。这就说明，它比较适合元素个数不太变化，而更多是存取数据的应用。当然，它的优缺点还不只这些.....

**线性表顺序存储结构的优缺点**

优 点 
· 无 须 为 表 示 表 中 元 素 之 
间 的 逻 辑 关 系 而 增 加 额 
外 的 存 储 窄 间 
· 可 以 快 速 地 存 取 表 中 任 
一 位 置 的 元 素 
缺 点 
· 插 入 和 删 除 操 作 需 要 移 
动 大 量 元 素 
· 当 线 性 表 长 度 变 化 较 大 
时 ， 难 以 确 定 存 储 空 间 
的 容 量 
造 成 存 储 空 间 的 “ 碎 片 ” 

**线性表的链式存储结构**

线性表的链式存储结构的特点是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。

因此，为了表示每个数据元素ai与其直接后继数据元素ai+1之间的逻辑关系，对数据元素ai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）。我们把**存储数据元素信息的域**称为**数据域**，把**存储直接后继位置的域**称为**指针域**。指针域中存储的信息称做**指针**或**链**。这两部分信息组成数据元素ai的存储映像，称为**结点**（Node）

n个结点（ai的存储映像）链结成一个链表，即为线性表（al，a2，……an)的链式存储结构，因为此链表的每个结点中只包含一个指针域所以叫做单链表。单链表正是通过每个结点的指针域将线性表的数据元素按其逻辑次序链接在一起，如图所示。

数 据 信 息 结 点 
指 针 
数 据 域 指 针 域 
地 址 05 開 

链表中第一个结点的存储位置叫做头指针，最后一个，当然就意味着直接后继不存在了，所以我们规定，线性链表的最后一个结点指针为“空”（通常用NULL或“^”符号表示，如图所示）

最 后 一 个 结 点 
头 指 针 
第 一 个 结 点 
07 圓 
山 
地 址 00 
an 
NULL 
指 针 指 向 窒 

有时，我们为了更加方便地对链表进行操作，会在单链表的第一个结点前附设一个结点，称为头结点。头结点的数据域可以不存储任何信息，谁叫它是第一个呢，有这个特权。也可以存储如线性表的长度等附加信息，头结点的指针域存储指向第一个结点的指针，如图3-6-4所示。

头 结 点 
最 耵 一 个 点 
可 存 线 性 表 长 度 等 公 敖 
后 继 指 
一 个 结 点 
地 址 的 
指 针 指 同 

**头指针与头结点的异同**

头 指 针 
· 头 指 针 是 指 涟 表 指 向 第 一 个 结 
点 的 指 针 ， 若 链 表 有 头 结 点 ， 
则 是 指 向 头 结 点 的 指 针 
· 头 指 针 具 有 标 识 作 用 ， 所 以 常 
用 头 指 针 冠 以 链 表 的 名 字 
· 无 论 链 表 是 否 为 空 ， 头 指 针 均 
不 为 窄 。 头 指 针 是 链 表 的 必 要 
元 素 
头 结 点 
头 结 点 是 为 了 操 作 的 统 一 和 方 
便 而 设 立 的 ， 放 在 第 一 元 素 的 
结 点 之 前 ， 其 数 据 域 一 般 无 意 
义 （ 也 可 存 放 链 表 的 长 度 
· 有 了 头 结 点 ， 对 在 第 一 元 素 结 
点 前 插 入 结 点 和 删 除 第 一 结 点 ， 
其 操 作 与 其 它 结 点 的 操 作 就 统 
· 头 结 点 不 一 定 是 链 表 必 须 要 素 

**线性表链式存储结构代码描述**

若线性表为空表，则头结点的指针域为"空"。

头 结 点 后 继 指 
针 地 址 
N 凵 上 
表 示 仝 链 表 

**↑**这只是表示线性表中的数据元素及数据元素之间的逻辑关系。

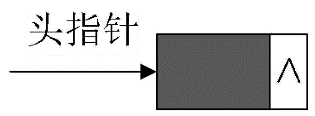
结 点 后 继 指 针 地 址 
头 指 针 
数 据 元 素 据 域 指 针 域 

**↑**改为此图较好

若带有头结点的单链表：



空链表表示方法：



单链表中，我们在C语言中可用结构指针来描述见Linklist0302.c

p->next 
ai+l 
P->data 
p->next->data 

**单链表的插入与删除**

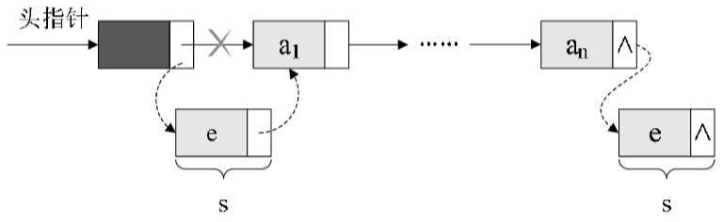
s->next = p->next；

p->next = S；

解读这两句代码，也就是说让p的后继结点改成s的后继结点，再把结点s变成p的后继结点（如图3-8-2所示）

s 
Jxau«-d 

对于单链表的表头和表尾的特殊情况，操作是相同的



单链表第i个数据**插入结点**的算法思路：

1.声明一指针p指向链表头结点，初始化j从1开始；

2.当j<i时，就遍历链表，让p的指针向后移动，不断指向下一结点，j累加1；

3.若到链表末尾p为空，则说明第i个结点不存在；

4.否则查找成功，在系统中生成一个空结点s；

5.将数据元素e赋值给s->data；

6.单链表的插入标准语句s->next=p->next；p->next=s；

7.返回成功。

**单链表的删除**

结 点 p 
结 点 q 或 
p->next 
结 点 q->next 或 
p->next->next 

q=p->next；

p->next=q->next；

解读这两句代码，也就是说把p的后继结点改成p的后继的后继结点。

1.声明一指针p指向链表头结点，初始化j从1开始；

2.当j<i时，就遍历链表，让p的指针向后移动，不断指向下一个结点，j++；

3.若到链表末尾p为空，则说明第i个结点不存在；

4.否则查找成功，将欲删除的结点p->next赋值给q；

5.单链表的删除标准语句p->next=q->next；

6.将q结点中的数据赋值给e，作为返回；

7.释放q结点；

8.返回成功。

从整个算法来说，我们很容易推导出：它们的时间复杂度都是O(n)。如果在我们不知道第i个结点的指针位置，单链表数据结构在插入和删除操作上，与线性表的顺序存储结构是没有太大优势的但如果，我们希望从第i个位置，插入10个结点，对于顺序存储结构意味着，每一次插入都需要移动n-i个结点，每次都是O(n)。

而单链表，我们只需要在第一次时，找到第i个位置的指针，此时为O(n)，接下来只是简单地通过赋值移动指针而已，时间复杂度都是0（1）。显然，对于插入或删除数据越频繁的操作，单链表的效率优势就越是明显。