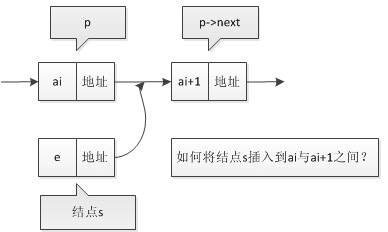
# 11线性表6

## 单链表的插入

我们先来看下单链表的插入。假设存储元素e的结点为s，要实现结点p、p->next和s之间逻辑关系的变化，大家参考下图思考一下：

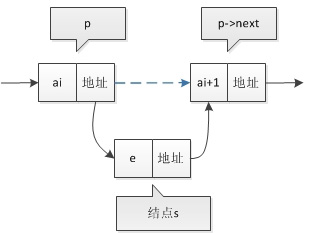


我们思考后发觉根本用不着惊动其他结点，只需要让s->next和p->next的指针做一点改变。

s->next = p->next;

p->next = s;

我们通过图片来解读一下这两句代码。



那么我们考虑一下大部分初学者最容易搞坏脑子的问题：这两句代码的顺序可不可以交换过来？

先p->next = s;

再s->next = p->next;

大家发现没有？如果先执行p->next的话会先被覆盖为s的地址，那么s->next = p->next其实就等于s->next = s了。

所以这两句是无论如何不能弄反的，这点初学者一定要注意咯~

单链表第i个数据插入结点的算法思路：

* + 声明一结点p指向链表头结点，初始化j从1开始；
  + 当j<1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；
  + 若到链表末尾p为空，则说明第i个元素不存在；
  + 否则查找成功，在系统中生成一个空结点s；
  + 将数据元素e赋值给s->data；
  + 单链表插入刚才两个标准语句；
  + 返回成功。

插入代码：ListInsert.c

/\* 初始条件：顺序线性表L已存在，1<=i<=ListLength(L) \*/

/\* 操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1 \*/

Status ListInsert(LinkList \*L, int i, ElemType e)

{

int j;

LinkList p, s;

p = \*L;

j = 1;

while( p && j<i ) // 用于寻找第i个结点

{

p = p->next;

j++;

}

if( !p || j>i )

{

return ERROR;

}

s = (LinkList)malloc(sizeof(Node));

s->data = e;

s->next = p->next;

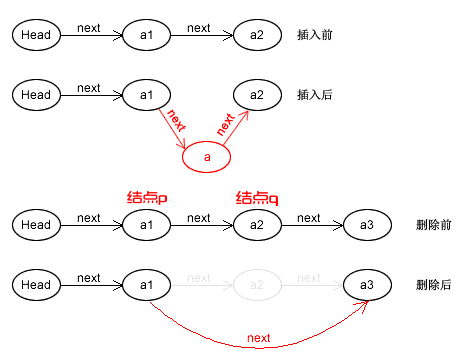
p->next = s;

return OK;

}

## 单链表的删除

现在我们再来看单链表的删除操作。



假设元素a2的结点为q，要实现结点q删除单链表的操作，其实就是将它的前继结点的指针绕过指向后继结点即可。

那我们所要做的，实际上就是一步：

可以这样：p->next = p->next->next;

也可以是：q=p->next; p->next=q->next;

那么我给大家提供算法的思路，由大家来写一下代码吧~

单链表第i个数据删除结点的算法思路：

* + 声明结点p指向链表第一个结点，初始化j=1；
  + 当j<1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累加1；
  + 若到链表末尾p为空，则说明第i个元素不存在；
  + 否则查找成功，将欲删除结点p->next赋值给q；
  + 单链表的删除标准语句p->next = q->next；
  + 将q结点中的数据赋值给e，作为返回；
  + 释放q结点。

## 效率PK

我们最后的环节是效率PK，我们发现无论是单链表插入还是删除算法，它们其实都是由两个部分组成：第一部分就是遍历查找第i个元素，第二部分就是实现插入和删除元素。

从整个算法来说，我们很容易可以推出它们的时间复杂度都是O(n)。

再详细点分析：如果在我们不知道第i个元素的指针位置，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个位置开始，插入连续10个元素，对于顺序存储结构意味着，每一次插入都需要移动n-i个位置，所以每次都是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O(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啦~

# 12线性表7

## 单链表的整表创建

对于顺序存储结构的线性表的整表创建，我们可以用数组的初始化来直观理解。

而单链表和顺序存储结构就不一样了，它不像顺序存储结构数据这么集中，它的数据可以是分散在内存各个角落的，他的增长也是动态的。

对于每个链表来说，它所占用空间的大小和位置是不需要预先分配划定的，可以根据系统的情况和实际的需求即时生成。

人生就要追求向单链表一样，灵活应变！

创建单链表的过程是一个动态生成链表的过程，从“空表”的初始状态起，依次建立各元素结点并逐个插入链表。

所以单链表整表创建的算法思路如下：

* 声明一结点p和计数器变量i；
* 初始化一空链表L；
* 让L的头结点的指针指向NULL，即建立一个带头结点的单链表；
* 循环实现后继结点的赋值和插入。

## 头插法建立单链表

头插法(栈即可由此实现)从一个空表开始，生成新结点，读取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，直到结束为止。

简单来说，就是把新加进的元素放在表头后的第一个位置：

先让新节点的next指向头节点之后

然后让表头的next指向新节点

嗯，用现实环境模拟的话就是插队的方法，始终让新结点插在第一的位置。

动画演示

头插法建立单链表

我们说好的代码呢？

<CreateListHead.c>

/\* 头插法建立单链表示例 \*/

void CreateListHead(LinkList \*L, int n)

{

LinkList p;

int i;

srand(time(0)); // 初始化随机数种子

\*L = (LinkList)malloc(sizeof(Node));

(\*L)->next = NULL;

for( i=0; i < n; i++ )

{

p = (LinkList)malloc(sizeof(Node)); // 生成新结点

p->data = rand()%100+1;

p->next = (\*L)->next;

(\*L)->next = p;

}

}

## 尾插法建立单链表

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。

就像现实社会我们鄙视插队不遵守纪律的孩子，那编程中我们也可以不这么干，我们可以把思维逆过来：把新结点都插入到最后，这种算法称之为尾插法。（小甲鱼给这个算法想到一个容易记住的艺名，叫“菊花”）

好，那我们接下来就结合欢乐的动画一起来理解理解菊花的内涵吧~

动画演示

尾插法建立单链表

我们说好的代码呢？

* + <CreateListTail.c>

/\* 尾插法建立单链表演示 \*/

void CreateListTail(LinkList \*L, int n)

{

LinkList p, r;

int i;

srand(time(0));

\*L = (LinkList)malloc(sizeof(Node));

r = \*L;

for( i=0; i < n; i++ )

{

p = (Node \*)malloc(sizeof(Node));

p->data = rand()%100+1;

r->next = p;

r = p; // 备注：初学者可能很难理解这句，重点解释。

}

r->next = NULL;

}

# 13线性表8

## 单链表的整表删除

当我们不打算使用这个单链表时，我们需要把它销毁（真狠，不要就给别人嘛，还销毁~）。

其实也就是在内存中将它释放掉（实际是告诉操作系统之前分配的内存空间不再使用，从而操作系统可以将这块内存空间交给其他的程序使用），以便于留出空间给其他程序或软件使用。

单链表整表删除的算法思路如下：

声明结点p和q；

将第一个结点赋值给p，下一结点赋值给q；

循环执行释放p和将q赋值给p的操作；

我们说好的代码呢？

ClearList.c

Status ClearList(LinkList \*L)

{

LinkList p, q;

p = (\*L)->next;

while(p)

{

q = p->next;

free(p);

p = q;

}

(\*L)->next = NULL;

return OK;

}

## 单链表的整表删除

这段算法代码里，常见的错误就是有同学会觉得q变量没有存在的必要，只需要在循环体内直接写free(p); p = p->next; 即可？

可这个世上没有无缘无故的爱，这样做会带来什么问题呢？

要知道p是一个结点，它除了有数据域，还有指针域。当我们做free(p);时候，其实是对它整个结点进行删除和内存释放的工作。而我们整表删除是需要一个个结点删除的，所以我们就需要q来记载p的下一个结点。这就好比\*\*\*\*\*\*。

## 单链表结构与顺序存储结构优缺点

我们分别从存储分配方式、时间性能、空间性能三方面来做对比。

**存储分配方式：**

顺序存储结构用一段连续的存储单元依次存储线性表的数据元素。

单链表采用链式存储结构，用一组任意的存储单元存放线性表的元素，相比顺序存储节约空间。

**时间性能：**

查找

顺序存储结构O(1)

单链表O(n)

插入和删除

顺序存储结构需要平均移动表长一半的元素，时间为O(n)

单链表在计算出某位置的指针后，插入和删除，无论多少元素时间仅为O(1)

**空间性能：**

顺序存储结构需要预分配存储空间，分大了，容易造成空间浪费，分小了，容易发生溢出。

单链表不需要分配存储空间，只要有就可以分配，元素个数也不受限制。

综上所述对比，我们得出一些经验性的结论：

若线性表需要频繁查找，很少进行插入和删除操作时，宜采用顺序存储结构。

若需要频繁插入和删除时，宜采用单链表结构。

比如说游戏开发中，对于用户注册的个人信息，除了注册时插入数据外，绝大多数情况都是读取，所以应该考虑用顺序存储结构。

而游戏中的玩家的武器或者装备列表，随着玩家的游戏过程中，可能会随时增加或删除，此时再用顺序存储就不太合适了，单链表结构就可以大展拳脚了。

当线性表中的元素个数变化较大或者根本不知道有多大时，最好用单链表结构，这样可以不需要考虑存储空间的大小问题。

而如果事先知道线性表的大致长度，比如一年12个月，一周就是星期一至星期日共七天，这种用顺序存储结构效率会高很多。

总之，线性表的顺序存储结构和单链表结构各有其优缺点，不能简单的说哪个好，哪个不好，需要根据实际情况，来综合平衡采用哪种数据结构更能满足和达到需求和性能。

最后，让我们通过欢乐的动画一起来回顾我们近期学过的知识吧~

# 14线性表9

## 静态链表

这一节课，我们试图通过静态链表的讲解来瞻仰古人的伟大！（似乎人总要挂了之后才能变得伟大~\_~）

神马是静态链表呢？又跟古人能有半毛钱关系？

地球人都知道C语言是个伟大的语言，他的魅力在于指针的灵活性，使得它可以非常容易地操作内存中的地址和数据，这比其他高级语言更加灵活方便。（面向对象使用对象引用机制间接地实现了指针的某些功能）

但是古人还木有C语言丫，木有JAVA丫，只有原始的Basic，Fortran等早期的编程语言，这些语言没有类似于C的指针功能，但是他们又想描述单链表，就没法实现了，怎么办呢？

真是不得不佩服他们的智慧，有人想出了用数组代替指针来描述单链表。大家能猜一猜他们是如何做到的么？

在讲解原理之前，先让大家知道这种用数组描述的链表叫做静态链表，这种描述方法叫做游标实现法。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 游标 | 5 | 2 | 3 | 4 | 0 | 6 | 7 | … | 1 |
| 数据 |  | A | C | D | E |  |  | … |  |
| 下标 | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **…** | **999** |

线性表的静态链表存储结构

#define MAXSIZE 1000

typedef struct

{

ElemType data; // 数据

int cur; // 游标（Cursor）

} Component, StaticLinkList[MAXSIZE];

对静态链表进行初始化相当于初始化数组：

Status InitList(StaticLinkList space)

{

int i;

for( i=0; i < MAXSIZE-1; i++ )

space[i].cur = i + 1;

space[MAXSIZE-1].cur = 0;

return OK;

}

小甲鱼备忘录：

我们对数组的第一个和最后一个元素做特殊处理，他们的data不存放数据。

我们通常把未使用的数组元素称为备用链表。

数组的第一个元素，即下标为0的那个元素的cur就存放备用链表的第一个结点的下标（相当于为尾节点）。

数组的最后一个元素，即下标为MAXSIZE-1的cur则存放第一个有数值的元素的下标，相当于单链表中的头结点作用。

静态链表的插入操作

有些喜欢思考的鱼油会说，你这不是挂羊头买鱼肉吗？还是数组丫，貌似没看出太多单链表的端倪~\_~

那我们接着从行为学的角度来剖析下，静态链表如何模拟单链表进行插入和删除的操作呢？

现在我们来看看如何实现元素的插入。静态链表中要解决的是：如何用静态模拟动态链表结构的存储空间分配，也就是需要的时候申请，不需要的时候释放。

我们前面说过，在动态链表中，结点的申请和释放分别借用C语言的malloc()和free()两个函数来实现。

在静态链表中，操作的是数组，不存在像动态链表的结点申请和释放的问题，所以我们需要自己实现这两个函数，才可以做到插入和删除操作。

为了辨明数组中哪些分量未被使用，解决的办法是将所有未被使用过的及已被删除的用游标链成一个备用链表，应该如何操作呢？请大家课后头脑风暴一下！

# 15线性表10

## 静态链表的插入操作

* 上节课我们说到，为了辨明数组中哪些分量未被使用，解决的方法是将所有未被使用过的及已被删除的分量用游标链成一个备用的链表。
* 每当进行插入时，便可以从备用链表上取得第一个结点作为待插入的新结点。可能这样说大家都还是难以理解吧？我们结合图片来讲解，这里我们假设要在A后边插入B：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 游标 | 6 | 5 | 3 | 4 | 0 | 2 | 7 | … | 1 |
| 数据 |  | A | C | D | E | B |  | … |  |
| 下标 | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **…** | **999** |

相信大家现在可以理解什么叫“一图胜千言”和“No pic you say a J8”的深刻内涵了吧！

那我们来谈谈代码，代码由两部分组成：

首先是获得空闲分量的下标：

int Malloc\_SLL(StaticLinkList space)

{

int i = space[0].cur;

if( space[0].cur )

space[0].cur = space[i].cur;// 把它的下一个分量用来作为备用

return i;

}

插入操作的实现代码如下：ListInsert.c

/\* 在静态链表L中第i个元素之前插入新的数据元素e \*/

Status ListInsert( StaticLinkList L, int i, ElemType e )

{

int j, k, l;

k = MAX\_SIZE - 1; // 数组的最后一个元素

if( i<1 || i>ListLength(L)+1 )

{

return ERROR;

}

j = Malloc\_SLL(L);

if( j )

{

L[j].data = e;

for( l=1; l <= i-1; l++ )

{

k = L[k].cur;

}

L[j].cur = L[k].cur;

L[k].cur = j;

return OK;

}

return ERROR;

}