7.4 链表

1. 链表概述

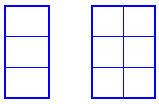
复习：

变量：通过变量定义引入。程序执行时，不能显式地用语句随意生成变量和消去变量，并且变量之间的关系也不能由变量本身的成分来表达。

数组 { 顺序存储结构
逻辑关系上相邻的两个元素在物理位置上也相邻
随机存取

数组作为存放同类数据的集合，给我们在程序设计时带来很多的方便，增加了灵活性。但数组也同样存在一些弊病。

数组的致命弱点：



(1)在对数组进行插入或删除操作时，需移动大量数组元素。

(2)数组的长度是固定的而且必须预先定义，因此数组的长度难以缩放，对长度变化较大的数据对象要预先按最大空间分配，使存储空间不能得到充分利用.

为避免大量结点的移动，我们希望构造动态的数组，随时可以调整数组的大小，以满足不同问题的需要。链表就是我们需要的动态数组。它是在程序的执行过程中根据需要有数据存储就向系统要求申请存储空间，决不构成对存储区的浪费。我们介绍线性数链式存储结构，简称为链表(Linked List)。

而按需要用随时生成和消去数据对象，数据之间的关系也能由程序随时设定和改变，这种实现方法称为动态方法，如链表。

什么是链表

由数据对象中的某个成员表示某个元素之后是哪一个元素。链表的元素个数可按需要任意增删，
链表元素的后继元素也能由元素自己指定。
具有链表这样两项特性的数据结构称为动态数据结构，链表是最简单的动态数据结构。
在链表上，可以非常方便地增加一个表元（或称元素、结点）或删除一个表元。图7.3(a)列出了只有3个表元的单链表结构。

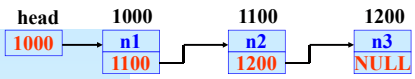


图7.3(a)单向链表结构示意图

head：链表的“头指针”变量，存放的是地址，指向链表的第一个元素（或称表元、结点）。
链表元素：分成两部分，存放实际的数据和后继表元指针。

链表一环扣一环地将多个表元链接在一起，即头指针head指向第1个表元，第1个表元又指向第2个表元，……，直到链表的最后一个表元。我们一般称链表的第1个表元为“首表元”，链表的最后一个表元为“尾表元”（尾表元的后继指针一般为“NULL”，表示链表不再有后继表元）。

当链表中没有任何一个表元时称为空链表，此时链表头指针为“NULL”，如图7.5(b)所示。

图7.3(b)空链表示意图

图7.5所示的链表结构是单向的，即每个表元只存储它的后继表元的位置，而不存储它的前驱表元的位置。这种形式的链表称为单链表。在单链表中，一个表元的后继表元位置存储在它所包含的某个指针成员中，单链表的每个表元在内存中的存放位置是可以任意的。如果某个算法要对链表作某种处理，必须从链表的首表元开始、顺序处理链表的表元。

常见的链表还有循环链表，在循环链表中，链表尾表元的后继指针指向链表的首表元。如下面图所示。

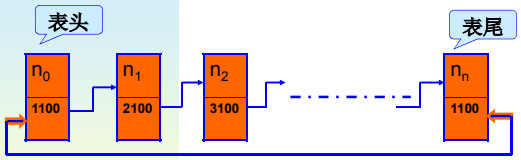


图7.3(c)循环链表结构示意图

链表结构的特点:

(1)数据对象的个数及其相互关系可以按需要改变.
(2)存储空间是程序根据需要在程序运行过程中向系统申请获得.
(3)不要求逻辑上相邻的元素在物理位置上也相邻.
(4)没有顺序存储结构所具有的弱点.

链表是结构的一个非常重要的应用，在实现计算机各种算法中应用非常多。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其后继结点的地址（或位置）信息，这个信息称为指针(pointer)或链(link)。这两部分组成了链表中的结点结构：

链表的组成

链表由有限多个节点（或者称结点）组成，每个结点的成员由两部分组成，一个是数据成员n，另一个是指针成员next，用一个结点的指针成员 next 指向下一个结点，各个结点逐个相连，从而构成一个链表。该链表中每一个结点都属于struct 的一个类型，其成员n存放的是数据信息,而成员next则存放下一结点的地址。链表的元素个数可按需要任意增删，链表元素的后继元素也能由元素自己指定。

具有链表这样两项特性的数据结构称为动态数据结构，链表是最简单的动态数据结构。

在某些应用中，要求链表的每一个表元都能方便地知道它的前驱表元和后继表元，这种链表的表元需包含两个指针，分别指向它的前驱表元和后继表元，这种链表称为双向链表。如图7. 4所示。在双向链表中，插入或删除一个表元比单向链表更为方便，但结构也较为复杂。限于篇幅，在这一节中仅介绍单链表的基本操作。

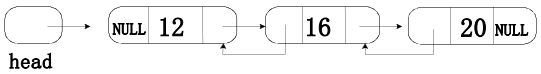


图7. 4 双链表结构

2. 动态变量

学完数组后,对于在编写程序时就必须决定数组大小一直觉得是件憾事，毕竟在很多时候,程序必须视使用者的需求而决定要定义多大的内存,在程序执行时再决定数组的大小. 动态数组和链表就是在程序的执行的过程中根据需要随时向系统申请存储空间，以满足不同问题的需要,不会造成存储空间的浪费。

在 C中提供了一些内存管理函数，这些内存管理函数可以按需要动态地分配内存空间，也可把不再使用的空间回收待用，即动态的生成变量或释放变量,这种变量称为动态变量,它为有效地利用内存资源提供了很好的手段。

动态数据结构中的数据对象是一种结构变量，有一些成员存储数据信息，还有存储指针的成员。最简单情况是结构包含有指向与自己同样结构的指针。例如

```
struct intNode  /* 整数链表表元类型 */
{ int value ;    /* 存放整数 */
  intNode *next ; /* 存放后继表元的指针 */
};
```

在结构类型intNode中，有两个成员，value和指向后继表元的指针成员next。其中指针成员next能指向的结构类型就是正在定义的intNode。这种结构又称为自引用结构。利用指针成员next，可把一个intNode类型的结构与另一个同类型的结构链接起来，用于描述动态数据结构中两数据对象之间的关系。在这种有成员引用自身的结构类型定义中，允许引用自身的成员只能是指针类型。

动态数据结构中的变量称为动态变量，动态变量能由程序调用库函数，显式地随意生成和释放(消去)。

在C语言中，有malloc和calloc两个库函数用于生成动态变量，另有一个库函数free用于释放动态变量。

(1) 分配内存空间函数malloc

调用形式： void * malloc(size)

功能：

在内存的动态存储区中分配一块长度为“size”字节的连续空间。分配空间成功则函数的返回值是一个空类型指针，为该区域的首地址,否则返回值为NULL。

void *：指向任何类型的数据，在使用时，要进行强制类型转换。

"size"是一个无符号整数。

例1：char *pc;

pc=(char *)malloc(100); 表示分配100个字节的内存空间，并强制转换为字符数组类型，函数的返回值为指向该字符数组的指针，把该指针赋予指针变量pc。

例2：

```
intNode *p;
p=(intNode*)malloc(sizeof(intNode));
```

申请能存储类型为intNode的结构的存储空间，并将该结构指针存于指针变量p。动态变量没有名字，只能通过指针引用它。

14

```
void main( )
{       int n,i;
        printf("please input the n\n");
        scanf("%d",&n);
        int *p=(int *)malloc(n);
        for(i=0;i<n;i++)*(p+i)=i+1;
        for(i=0;i<n;i++)
                printf("%d\t",*(p+i));
        printf("\n");
}
```

复习前面我们学过的指针数组，

例如： short *p [3];

表示定义了一个指针数组pc，p的3个元素p[0]、p[1]、p[2]都是指针，将分别指向short类型的对象。

如果有以下语句

```
short c[3][4], *pc[3];
pc[0] = (short *)malloc(n1 * sizeof(short));
pc[1] = (short *)malloc(n2 * sizeof(short));
pc[2] = (short *)malloc(n3 * sizeof(short));
```

表示指针数组p可以分别从内存中申请3个动态数组，数组长度分别为n1，n2和n3。显然，n1，n2和n3可以相同，也可以不相同。如果n1，n2和n3不相等，即指针数组的3个指针元素指向数量不同的short对象。这里数组长度都为4.

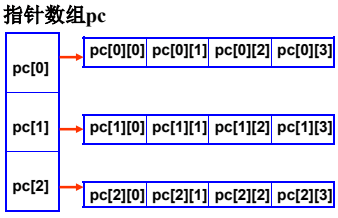
另外，还可以在使用完毕后释放这些单元，以后再指向不同数量的对象。

与二维数组c(静态数组)相比，每个动态数组pc[i]，其作用与静态数组c的作用基本相当，即可以用pc[i][j]来引用各动态数组的元素。但是多了3个指针变量，而且这3个动态数组的地址不一定连续。例如，c[0][3]与c[1][0]是连续存放的，而pc[0][3]与pc[1][0]的地址不相关。

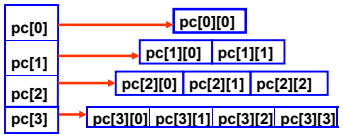
数组c

C[0][0]	C[0][1]	C[0][2]	C[0][3]
C[1][0]	C[1][1]	C[1][2]	C[1][3]
C[2][0]	C[2][1]	C[2][2]	C[2][3]

指针数组pc




```
/*利用动态申请存储空间,采用数组指针编程输出杨辉三角形 */
#include <stdlib.h>
#define N 10
void main()
{  int i, j, *pc[N], n;
   printf("Input n,<%d:\n",N);
   scanf("%d", &n);
  for(i=0; i<n; i++) /* 申请pc[i] */
     pc[i] = (int *)malloc((i+1)*sizeof(int)) ; /*pc[i]=pc[i-1]+i*/
```



```
    /* 生成所有数据 */
        for(i=0; i<n; i++)
        {
            pc[i][0] = pc[i][i] = 1;
            for(j=1; j<i; j++)
                pc[i][j] = pc[i-1][j-1] + pc[i-1][j];
        }
        /* 按格式输出*/
        for(i=0; i<n; i++)
        {
            printf("%*c", 2*n-2*i, ' ');
            for(j=0; j<=i; j++)
                printf("%4d", pc[i][j]);
            printf("\n");
        }
}
```

根据我们已经学过的关于指针和数组的知识，知道通过申请内存，二级指针可以指向一个指针，也可以指向一个指针数组。

例如：

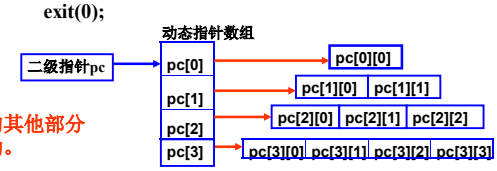
```
short **pe3, **pe4;
pe3 = (short **)malloc(sizeof(short *));
pe4 = (short **)malloc(3 * sizeof(short *));
```

对动态指针数组的指针元素pe4[0], pe4[1], pe4[2]，还可以进一步申请动态的一维数组。

因此，通过二级指针pe4的两次内存申请，得到的是动态的可变长度的二维数组，比静态的二维数组更加灵活，也有更多的应用。

```
/*利用动态申请存储空间,采用二级指针编程输出杨辉三角形*/
 #include <stdio.h>
 #include <malloc.h>
 void main()
 {      int i, j, **pc, n;
        printf("Input n:\t");
        scanf("%d", &n);
        if( !(pc = (int **)malloc(n*sizeof(int *))) )
            exit(0); // 申请pc的动态指针数组对象
        for(i=0; i<n; i++) //申请动态指针数组元素pc[i]的对象
            if( !(pc[i] = (int *)malloc((i+1)*sizeof(int))) )
                exit(0);
```

程序中的其他部分不作改动。



(2)分配内存空间函数 calloc

调用形式： void * calloc(n,size)

功能：

在内存的动态存储区中分配n个长度为size个字节的连续空间，分配空间成功则返回值是一个空类型指针，指向该分配域的起始地址。否则返回值为NULL.

例如：申请分配能存储100个整数的存储块。

```
p = (int *) malloc(100 * sizeof(int));
q = (int *) calloc(100, sizeof(int));
```

(3)释 放内存空间函数free

调用形式： free(void *p);

功能：

释放由p所指向的存储区。

说明：

p的值必须是调用函数malloc()或calloc()申请到的连续存储空间中的地址。

释放p所指向的一块内存空间，p是一个任意类型的指针变量，它指向被释放区域的首地址。被释放区应是由malloc或calloc函数所分配的区域。

以上三个函数中，形参size和n为unsigned类型，p,q为指针类型.

函数malloc()和calloc()返回的是系统分配的连续存储空间的首地址，其类型为void *的（即无类型的指针值，程序应将返回值强制转换成某种特定的指针类型），程序将这返回值赋给某个指针变量，然后用该指针变量间接引用存储空间的相应成分。

使用以上三个函数,必须在程序的头部写上:

```
#include <malloc.h>
或  #include <stdlib.h>
```

例2：分配一块区域，输入一个学生数据。

```
void main(){
   struct stu{  int num;char *name;
     char sex;float score;
   } *ps;
   ps=(struct stu*)malloc(sizeof(struct stu));
   ps->num=102;
   ps->name="Zhang ping";
   ps->sex='M';
   ps->score=62.5;
   printf("Number=%d\nName=%s\n",
         ps->num,ps->name);
   printf("Sex=%c\nScore=%f\n",ps->sex,
         ps->score);
  free(ps);
}
```

本例中，定义了结构stu，和stu类型指针变量ps。然后分配一块stu内存区，并把首地址赋予ps，使ps指向该区域。再以ps为指向结构的指针变量对各成员赋值，并用printf输出各成员值。最后用free函数释放ps指向的内存空间。整个程序包含了申请内存空间、使用内存空间、释放内存空间三个步骤，实现存储空间的动态分配。

调用free函数的注意事项

1）只能释放原先申请到的内存单元

```
int b[2], c, *p, *pi=b, *pj=&c;  p = (int *)calloc(3, sizeof(int));
```

由于由p指向的是动态存储单元(存储空间)，因此可以释放p的目标。由于pi指向的存储空间是数组b，pj指向的存储空间是变量c，因此不能释放pi和pj的目标。

2）必须将原先申请到的内存单元一次性释放

例如，用pk指向了申请到的n个连续存储单元。则在使用free函数时，不能只释放m(m<n)个单元，而是必须将这n个单元一起释放。例如：

```
short *pk, *pl;
pk = (short*)calloc(3, sizeof(short)); pl = pk + 1;
```

则只允许一次性释放由pk指向的3个连续的动态存储单元，而不能只释放由pl指向的2个连续的动态存储单元。即

```
free(pk); /* 合法 */
free(pl); /* 非法 */
```


3）由free函数释放单元的归属

虽然，我们通常把释放某个指针指向的动态存储单元简称为释放某个指针，但是使用free函数释放的意义表示只是将这些单元的使用权归还给系统，而不是释放指针变量本身。

假定用指针pm申请动态存储单元，而且获得了目标的地址，则当使用free(pm)语句时，仅仅表示将pm指向的单元归还给系统，而并不是释放指针本身，也就是说指针pm的存储单元还在。此后，可能pm还保持其原先目标的地址，但是已经失去它的目标，再也不能用pm的值访问所指向的单元，否则将造成非法访问内存的致命错误。

例如：

```
short *pm;
pm = (short*)malloc(sizeof(short)); /* 假定pm获得地址 */
*pm = 5; /* 给pm的目标赋值 */
free(pm); /* 释放pm的目标 */
*pm++; 或 *pm = 8; /* pm已经失去目标，非法 */
```


3. 单链表上的基本操作

- 建立空链表
- 生成指定值的新表元
- 遍历链表(输出链表)
- 插入一个新表元
- 查找一个表元
- 删除一个表元

(1)建立空链表

链表的建立过程是从空链表(没有链表表元)出发，逐渐插入链表新表元的过程。要使以变量head为头指针的链表为空链表，让head取NULL值即可。

语句：head = NULL;

说明：建立以head为链表头指针的空链表。

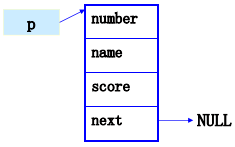
(2)生成指定值的新表元

链表的建立过程：

2.1 .定义链表数据结构；

下面以学生单链表为例，说明各种操作的实现算法。

```
struct stuS     /* 学生链表表元结构类型 */
{
     int  number;    //学号
     char name[20];//姓名
     int  score;     //成绩
     stuS *next; //指向后继表元的指针
};
stuS *head,*p,*p1,*p2;
/* head是链表的头指针，
其余是工作指针 */
```



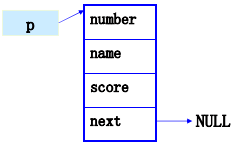
2.2 输入有效数据，重复以下操作：

a) 利用malloc( )函数向系统申请分配一个节点p。
b) 若是空表，将新节点连接到表头；若是非空表，将新节点接到表尾；

p=(stuS *)malloc(sizeof(stuS));//p指向申请到的存储块

函数调用malloc()的实参表达式sizeof(stuS)是单链表上一个表元所需的字节数。malloc()函数的返回值是一个新的动态变量的开始地址，返回值作强制类型转换(stuS *)，使返回的(void *)无类型指针转换成struct stuS * 类型的指针，并将该指针赋给指针变量p。就可使用p->number和p->next间接引用动态变量的成员number和next。


```
scanf("%d%s%d", &p->number, p->name, &p->score);
/* 输入并存储本表元的值 */
p->next = NULL; /* 后继表元地址为空 */
```



注意：链表的头指针是非常重要的参数，因为对链表的输出和查找都要从链表的头开始，所以链表创建成功后，要返回一个链表头节点的地址，即头指针。

【例7.4】编写创建一个学生表元的函数。

设学生的学号、姓名和成绩由函数的形参指定，函数返回新表元的指针。函数首先调用malloc()申请新表元的空间，空间所需的字节数为sizeof(stuS)。然后将形参值存入新表元的相应成员中。最后，函数返回新表元的指针。

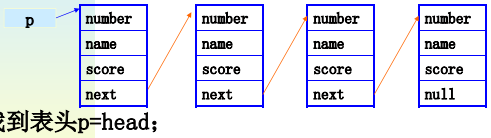
```
stuS *createStudent(int num,char *name,int s)
{
    stuS *p = (stuS *)malloc(sizeof(stuS));
    p->number = num;
    strcpy(p->name, name);
    p->score = s;
    p->next = NULL;
    return p;
}
```

```
void main()
{ stuS *p;
  p= createStudent(3, "zmy",89);
  printf(" %d,%s,%d\n",p->number,p->name,p->score);
}
```


(3)遍历链表

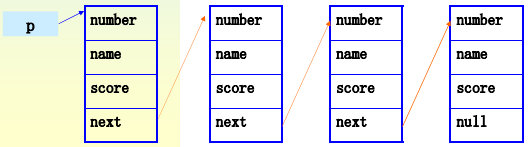
所谓遍历链表是顺序访问链表中的表元，对各表元作某种处理。首先找到链表头head，只有是非空链表才将节点的值输出，若是空表则立即退出输出过程；若是非空链表，则通过指针的不断向后移动逐个输出链表中各个节点的值，直到链表尾。

即：



① 找到表头p=head；
② 当链表非空p!=NULL时，重复以下操作；
a)输出节点p数据p->number, p->name,…, ；
b)跟踪p下一节点p=p->next；


```
struct stuS *p = head;
while(p!=NULL)
{
   printf(" %d, %s, %d\n", p->number,
   p->name, p->score);
   p = p->next;
}
```



【例7.5】编写遍历学生单链表的函数。

函数应设学生链表的头指针为形参。函数首先检查链表是否为空，如果是空链表，输出链表中没有学生字样后返回。否则，函数从链表的首表元开始，顺序输出每个学生的信息。

```
void travelStuLink(stuS *head)
{ stuS *p = head;
  if(head == NULL){
    printf ("\n目前在链表中没有学生\n");return;
    }
  printf ("\n目前在链表中的学生如下\n");
  while (p != NULL)// while (p)
  { printf("%d\t%s\t%d\n",
        p->number, p->name, p->score);
    p = p->next;  /* 准备访问下一个表元 */
  }
}
```

p++?

(4) 在链表中查找表元

- 查找表元的操作和链表的查找目的，链表的有序程度有关：
  - 简单查找：查找后只获取详细信息；
  - 动态查找：查找后进行链表修改；
  - 将查到的表元删除
  - 在查到的表元之前插入一个新表元
  - 无序查找：从头至尾查找；
  - 有序查找：可以利用有序信息。

● 无序单链表上的查找

无序单链表上的查找函数需设置两个形参：

●学生链表的头指针

●要寻找的学生的学号。

函数的寻找过程是一个循环，设循环工作变量为v。

- 查找关键字为num的程序片段：

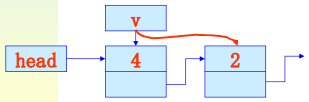
```
struct stuS *v = h;//v的初值是链表的头指针
while(v!=NULL && v->value!=num)//循环条件
    v = v->next;//没有找到指定学号的学生,让v指向下一个学生
```

寻找循环结束后，函数返回v。

如果找到，v指向找到的表元；如果未找到，v的值是NULL，表示链表中没有要找的学生。

【例7.6】编写无序学生单链表上的查找函数

```
stuS * searchSLink(stuS *h, int num)
{   stuS  *v = h;
    while (v != NULL && v->number != num)
          v = v->next;
    return  v;
}
```



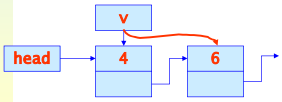
●有序单链表上的查找

如果单链表的表元是按表元值从小到大顺序链接的，在顺序考察链表表元的查找循环中，

当发现还有表元，且该表元的值比寻找值小时，应继续查找，准备去考察下一个表元。

反之，若指针为空，或当前的表元值大于要寻找的值，则查找结束。

查找结束后，仅当还有表元，并且当前表元的值与寻找值相等情况下，才算找到，函数返回当前表元的指针。否则，链表中没有要寻找的表元，函数应该返回NULL。



【例7.7】编写学号从小到大有序学生链表上查找指定学号的函数searchSOLink()。

有序单链表上的查找函数也需设置两个形参：

●学生链表的头指针

●要寻找的学生的学号。

```
stuS *searchSOLink(stuS *h, int key)
{ stuS  *v = h;  // v:从链表的首表元开始
  while ( v != NULL && v->number < key)
     //循环条件是有表元，且表元的值小于寻找值。
        v = v->next;
  return  v != NULL && v->number == key ? v : NULL;
 /*寻找循环结束后，有当前表元，且当前表元的值是要寻找的值，函数才返回当前表元的指针。否则，返回NULL。*/
}
```

(5)在链表中插入新表元

在链表中插入新的表元是建立链表的主要操作之一。根据插入位置和风格的不同，可以有以下插入方法：

- 在首表元之前插入新表元；
- 在某指针所指的表元之后插入新表元；
- 在链表末尾插入新表元。

●在首表元之前插入新表元

在首表元之前插入新表元，有两个步骤：

1. 将原链表的首表元接在新表元的后面；
2. 然后修改链表的头指针，使之指向新的首表元：
   - p->next = head; /*新表元指向原首表元地址，使原首表元成为后继表元*/
   - head = p; //头指针指向新表元，使新表元成为首表元

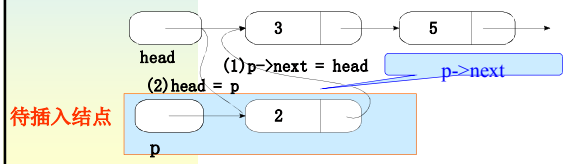
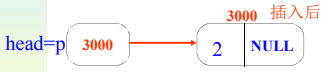


图7.5 在首表元之前插入新表元的实现代码和链表状态变化图

注意： 用指针p 指向新结点。并且假定新结点已经通过内存分配获得，即有p = (stuS *)malloc(sizeof(stuS));

若是空链表的前插入就是插入一个结点后使得空链表成为只有单个结点的链表。



① head=p;
② p->next=NULL;

●在指定表元之后插入新表元

- 对于单向链表，一个表元的前驱表元不是可以直接引用的，因此往往要求将新的表元插入在某个已知表元之后。设w为该已知表元，则w->next即为该表元的后继表元：
  - p->next = w->next;

  /*将w的后继表元连接在新表元p之后*/
  - w->next = p;

  /*将新表元连接在w之后*/

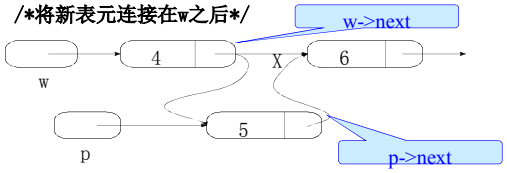


图7.6 在指定表元之后插入新表元的实现代码和链表状态变化图

【例7.8】编写在学生链表中，在指定学生之后插入一个新学生的函数。

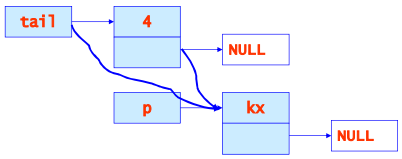
【解题思路】

函数设有一个表示学生链表头指针的形参head和指定学生学号的形参num。函数首先在链表中寻找指定学号的学生。如果链表中没有指定学号的学生，则不做插入工作，直接返回。如果有指定学号的学生，函数申请存储新学生信息的空间，输入新的学生信息，并完成插入工作。

```
void insert(stuS *head, int num)
{  stuS *w=head, *p;
   while (w != NULL && w->number != num)
       w = w->next;      /* 后移一个表元 */
   if (w == NULL)  {     /* 未找到 */
      printf ("学号 %d  没有找到! \n", num); return;
   }
   /* 找到，w指向找到的表元 */
   p=(stuS *)malloc(sizeof(stuS));//p指向新申请的表元
   printf ("请输入学号、姓名、成绩: ");
   scanf ("%d%s%d", &p->number, p->name, &p->score);
   p->next=w->next;/* *w的后继表元作为*p的后继表元*/
   w->next = p;   /* *p作为*w的后继表元 */
}
```

●在表尾插入新表元

- 如果某链表拥有尾指针tail，而新表元p正好需要插入在最后一个表元之后，则需要更新表尾指针tail：
  - tail->next = p;
  - tail = p;



如果没有尾指针tail

设要插入的新表元由指针p所指，将p所指表元添加到链表末尾的操作可由以下3个操作步骤完成：

① 从头指针开始依次找到最后的表元，由指针w指向最后的表元。

② 将指针p所指表元成为指针w所指的后继表元，用代码 w->next = p实现。

③ 将新表元的后继表元设置为空，用代码 p->next = NULL 实现。

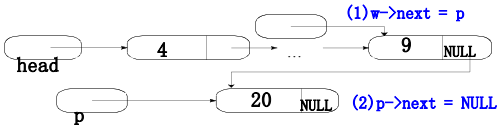


图7.7 在末表元之后插入新表元的实现代码和链表状态变化图

【例7.9】编写在学生链表的末尾接上一个新学生的函数。

函数设有一个表示学生链表头指针的形参head。函数首先申请存储学生信息的空间，并输入新的学生信息。函数在发现原链表是空时，简单地将新表元指针作为链表的头指针返回。否则，用循环寻找链表的末尾表元，然后将新表元接在末尾表元之后，返回原链表的头指针。

```
stuS *append(stuS *head)
{ stuS *w = head,*p;
  p=(stuS *)malloc(sizeof(stuS));//p2指新申请的表元
  printf ("请输入学号、姓名、成绩：");
  scanf ("%d%s%d", &p->number, p->name, &p->score);
  p->next = NULL; /* p为最后一个表元 */
  if (head == NULL) return p;   /* 原链表是空链表,
直接返回新表元指针 */
  while(w->next != NULL)//非空链表，顺序向后找尾表元
    w = w->next;/* 后移一个表元，直到w指向尾表元 */
  w->next = p;/* *p作为*w的后继表元 */
  return head;    /* 返回头指针 */
}
```

(6)从链表中删除表元

要将一表元从链表中删除，首先要在链表中查找该表元。若未找到，则不用做删除工作；在删除时还要考虑两种不同的情况：

1.删除链表头节点、要修改链表头指针.

2.删除的表元不是链表的首表元,要更改删除表元的前驱表元的后继指针。

设指针变量p1的初值指向首表元，通过循环，p1指向下一个表元（p1 = p1->next实现）；
指针变量p2总是指向p1的前驱表元（使用p2 = p1实现）。

53

●删除单链表的首表元

单链表的首表元删除后，首表元的后继表元成为新的首表元。假设从单链表删除下来的表元由指针变量p指向它。

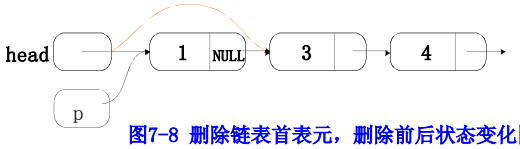


图7-8 删除链表首表元，删除前后状态变化图

```
p = head;/* 将首表元的指针保存到某指针变量*/
 if(p) {//如果有首表元,即单链表不为空表
   head = head->next;//头指针指向首表元的后继表元
   p->next = NULL;/* 删除表元与原单链表脱离联系*/
 }
```

●删除单链表中的某表元

首先必须查找表元，若找到，则删除，删除时必须考虑：若删除的是首表元，必须更改头指针；否则，必须更改前驱w表元的后继指针，因此需要2个工作指针。

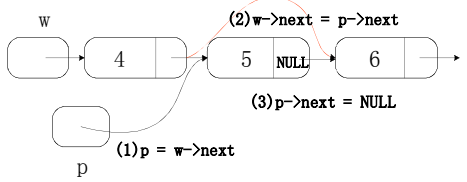


图7-9删除链表中w所指表元的后继表元，删除前后状态变化图

表元从链表删除后，对删除的表元可有两种不同的处理。或回收删除表元所占的存储空间，这可调用函数free(p)来实现；或将删除表元插入到其他链表中。

【例7.10】编写在学生链表中删除指定学号的表元的函数。

【解题思路】

删除特定的表元必须先做寻找的工作。在寻找过程中，必须记住当前表元的前驱表元，待寻找结束时，不仅找到了要删除的表元，同时得到删除表元的前驱表元。设以p为寻找循环的工作指针，它的初值是链表的头指针，循环条件是还有当前表元，且当前表元的学号不等于要寻找的学号。循环体由两条语句组成，让当前表元的指针保存于变量w(w = p)，当前表元指针指向下一个表元(p = p->next)。其中，先把当前表元指针保存于某个变量是必须的，否则当寻找循环结束时，没有当前表元的前驱表元指针就不能做删除操作。细节见以下函数定义。

```
stuS *delStu (stuS *head, int num)  /* num为要查找的学号 */
{ stuS *w, *p;
  if (head == NULL) return NULL;
  p = head;   /* 让p指向链表的首表元 */
  while (p && p->number != num) {
     w = p; p = p->next;  /* w指向前驱表元，p指向下一个表元 */
  }
  if (p) {/* 找到 */
     if(p == head) head = p->next;/*删除首表元，改链表头指针*/
     else  w->next = p->next; /* 修改前导表元的的后继指针 */
     printf ("学号 %d  已被删除\n", num);
     free (p);/* 释放被删表元的存储空间 */
  }
  else printf ("学号 %d  找不到! \n", num);
  return head;       /* 返回头指针 */
}
```

```
//以下程序是在单链表上操作的主程序的定义
#include <stdio.h>
#include <string.h>
#include <malloc.h>
struct stuS
{ int number;              /* 学号 */
  char name[20];           /* 姓名 */
  int  score;              /* 成绩 */
  stuS *next;
};        /* 其他函数应插在主函数之前 */
```

```
void main()
{ stuS *head=NULL;
  char *menu[ ]={"1  在链表末尾添加新表元",
                 "2  在指定表元之后插入新表元",
                 "3  显示链表中所有表元",
                 "4  删除链表中指定表元",""};
  int i, ans, number;
```

```
while (1) {
    printf ("\n请选择下列菜单命令：\n");
     for ( i=0; menu[i][0]!='\0'; i++)  /* 显示菜单 */
         printf ("\t%s\n", menu[i]);
     printf ("\t其他选择（非1～4），结束本程序运行！\n");
     scanf ("%d",&ans);
     switch (ans) {
        case 1: head= append (head);  break;
        case 2: printf ("请输入插入学生的学号：");
                scanf("%d", &number);
                insert (head, number);  break;
        case 3: travelStuLink(head); break;
        case 4: printf ("请输入要删除的学号：");
                scanf ("%d", &number);
                head = delStu(head, number);  break;
        default: return;
      }
   }
}
```

为了避免判别是否是链表的首表元和修改链表头指针，这类程序中的单链表通常增设一个辅助的表元，它出现在链表有效表元的首表元之前，如图7-10所示。辅助的表元的值往往比较特殊，以便能与其他有效表元相区别。

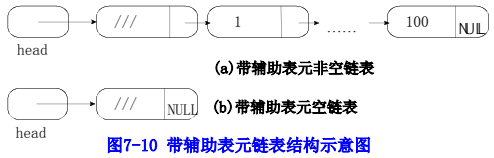


图7-10 带辅助表元链表结构示意图

在带辅助表元链表上的插入或删除表元的操作，因总是在辅助表元之后进行，不再需要判别是否是链表的首表元，也不会对链表头指针作修改。带辅助表元链表有的书上也称为带岗哨的链表。

例如：把新数据放在链表表首形成链表(后进先出)

```
#include <stdio.h>
#include <stdlib.h>
#define  LIST struct  link
LIST {  char data;
        LIST  *next;
     };
/*typedef struct link LIST;*/
```

62

```
void   CreatList(LIST *head, int n)
{
        LIST *p;
        p =(LIST *)malloc(sizeof(LIST));
        p->data=n;
        p->next=head->next;
        head->next=p;
}
void writelist(LIST  *head)
{    LIST *p=head->next,*t;
      while( p ){   printf("%4c",p->data);
                    t=p;   //保留当前地址
                    p=p->next;
                    free(t);//释放空间
                }
      printf("\n");
}
```

63

```
void main()
{
        LIST *head;char c;
        head =(LIST *)malloc(sizeof(LIST));
        head->next = NULL;
        printf("输入数据\n");
        while((c=getchar())!='\n')
            CreatList(head,c);
        writelist(head);printf("\n\n");
}
```

习题讲解：

64

```
//建立动态链表(先进先出)
#include <stdio.h>
#include <stdlib.h>
#define   LIST      struct  link
LIST {
            int data;
            LIST  *next;
        };
/*typedef struct link LIST;*/
LIST *ptr;
```

```
void create(int n)
{
        LIST *p;   int i=0;
        p=ptr =(LIST *)malloc(sizeof(LIST));
        printf("Please inpue data:\n");
        scanf("%d",&p->data);
        for( i = 1;i < n ;i++)
        {
                p->next=(LIST *)malloc(sizeof(LIST));
                p=p->next;
                scanf("%d",&p->data);
        }
                p->next=NULL;
}
```

```
void main()
{

    LIST *p;
    int n;
    printf("Enter n:\n");
    scanf("%d",&n);
    create(n);
    p=ptr;//p=ptr->next;带辅助表元
    for(;p;p=p->next)
        printf("%d\t",p->data);
    printf("\n");
}
```

```
//建立动态链表(后进先出)
 #include <stdio.h>
#include <stdlib.h>
#define  LIST  struct  link
LIST { int data;  LIST  *next;};
/*typedef struct link LIST;*/
LIST *ptr;
 void append(LIST *ptr,int c)
 {
        LIST *p;
        p =(LIST *)malloc(sizeof(LIST));
        p->data=c;
        p->next=ptr->next;
        ptr->next=p;
 }
```

```
void main()
{
        int  c;
        LIST *p, *ptr;
        int i;
        printf("Input Data.\n");
        ptr =(LIST *)malloc(sizeof(LIST));
        ptr->next=NULL;
        while((scanf("%d",&c))==1)
                append(ptr,c);
        p=ptr->next;
        for(i=0;p;p=p->next)
                printf("%c%c",p->data,++i%6?'\t':'\n');
        printf("\n");
}
```