**§2.3线性表的链式存储结构**

**链式存储结构：动态存储过程。根据所需要的元素（结点）随时申请存储空间（不需要连续的空间），即逻辑上连续的元素（结点）不要求物理（存储）上也连续，它是通过所谓的“链”拉起来。**

**§2.3.1 单链表**

**设有一线性表，元素分别是a1、a2和a3，存储情形如下：**

|  |  |  |
| --- | --- | --- |
| **存放地址** | **结点** | **链接地址** |
| **1000** |  | **1090** |
| **…………** | **……** | **……..** |
| **head🡪1040** |  | **1000** |
| **…………** | **………** | **…………** |
| **1090** |  | **NULL（0）** |

**每个元素都包含两个域（数据元素信息和指向链表中另一个结点的指针）。即，**

|  |  |
| --- | --- |
| **data数据信息** | **next下一结点的指针** |

**此时称各元素为“结点”，由一个“链”拉起来，称为链表。由于指针是向一个单方向指，所以称作“单链表”（后面还会有前后两个方向的指针指向的链表称为“双链表”）。为形象起见，将前面的存储方式画成下述情况（要了解图形的真实含义）。即，**

**head🡪**

a1

a2

a3

**其中：**

**head称为头指针（head放置头结点的起始地址）；**

**a1所在的结点称为头结点；**

**a3所在的结点称为尾结点（其指针域为空，即NULL，用斜线表示）。**

**单链表的头结点（a1）没有前驱，尾结点（a3）没有后继，而结点（a2）只有唯一的前驱和唯一的后继结点，所以符合“线性”特点。**

**单链表的结点定义如下：**

**typedef int ElemType;**

**typedef struct Node**

**{**

**ElemType data;**

**struct Node \*next;// 递归定义**

**}LNode,\*LinkList;**

**其中：LNode为结构体类型,而LinkList为结构体指针类型**

**LNode \*head,\*p,\*L;//头指针和指针**

**或LinkList head,p,L;**

**当定义完成之后，结点的存储是靠库函数malloc向系统申请。**

**如果申请成功该函数的返回值为这块内存的起始地址（指针），否则返回的值为NULL（空指针）。由于该函数申请指向的目标是void指针类型，所以对具体指向的目标为非void指针类型时要进行“指针类型的强制转换”。**

**库函数malloc的一般使用形式：**

**p=(指针类型的强制转换)malloc(申请存储的大小);**

**如果无“强制转换”部分，p所指向的类型默认为void型。即p=(void)malloc(申请存储的大小);**

**与其“存储申请”函数malloc对应的“存储释放”库函数free的功能是，释放掉指针指向的目标，将该存储还给系统。**

**库函数free的一般使用形式：**

**free(p);把p指向的目标释放后还给系统。**

**当申请一个结点时，其中“强制转换”可以由三种方式表达：**

**p=(LinkList)malloc(sizeof(LNode));**

**或p=(LNode \*)malloc(sizeof(LNode));**

**或p=(struct Node \*)malloc(sizeof(**

**struct Node));**

**申请成功后的状态：**

**p🡪**

**对结点的数据域和指针域的表示：**

|  |  |
| --- | --- |
| **p🡪data** | **p🡪next** |

**为方便起见，专门为单链表设置一个专用“头结点”作为标志，它的数据域无意义，而指针域指向链表的第一个结点。于是出现“带”和“不带”头结点的单链表。**

**●“带”头结点的单链表**

**L🡪**

a1

a3

a2

Ø

**●“不带”头结点的单链表**

**L🡪**

a3

a2

a1

**注意：L称为头指针(有时也用head)，操作链表时它总是指向头结点（千万不能丢），操作链表时，要另设辅助指针协助。**

**§2.3.2 单链表的实现和典型操作**

**单链表的实现和典型操作可有很多。只讨论：创建（拉链），遍历（访问，走链），插入和删除；**

**以上操作需要相关一些指针，除“头指针”必须外，还需要一个或几个辅助指针。在操作过程中要知道哪个指针不变，哪个指针要变。于是有下面的提醒**

**赋值的提醒：B=A;//A值不会改变，B值会改变；赋值后A和B相等（二者指向同一目标）。**

**§2.3.2.1单链表的创建（拉链）**

**1.创建带“头结点”单链表**

**（1）从链尾向链头创建（反向创建，即“头插法”）**

**由于每次创建的新结点为当前链表实际的头结点，所以称为“头插法”。**

**L🡪**

Ø

**L🡪**

Ø

a0

**L🡪**

a0 

Ø

a1

**L🡪**

a0 

Ø

a1

a2

a0 

Ø

a1

**操作需要“头指针”和一个辅助指针。即**

**LNode \*L,\*p;**

**第一步：创建专用的“头结点”（其数据域不用理睬）**

**L=(LNode \*)malloc(sizeof(LNode));**

**L🡪next=NULL;**

**L🡪**

Ø

**第二步：创建第一个结点（即当前的头结点，实际也是尾结点）**

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

**p🡪data=a[0];**

**p🡪next=L🡪next;//相当于p🡪next=NULL;**

**L🡪next=p;**

**p**

**L🡪**

Ø

a0

**第三步：创建第二个结点（实际是倒数第二个结点，也是当前的头结点）**

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

**p🡪data=a[1];**

**p🡪next=L🡪next;**

**L🡪next=p;**

**p**

**L🡪**

a0 

Ø

a1

**………**

**创建n个结点，数据由数组提供**

**void LinkListCreat1(LinkList L,ElemType a[],**

**int n)**

**{//L为头指针，p为协助指针**

**int i;**

**LinkList p;**

**L=(LNode \*)malloc(sizeof(LNode));**

**if(L==NULL){ printf(“申请空间失败！”);exit(0);}**

**L🡪next=NULL; // 生成“带”头的结点（对数据域不予理睬）**

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

**{**

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

**//生成新结点（首次时建立尾结点）**

**if(p==NULL)**

**{printf(“申请空间失败！”);exit(0);}**

**p🡪data=a[i];**

**p🡪next=L🡪next;**

**L🡪next=p;**

**}**

**}**

**时间复杂度：O(n)**

**(2)从链头向链尾创建（正向创建，即“尾插法”）**

**由于每次创建的新结点为当前链表的尾结点，所以称为“尾插法”。**

**L🡪**

Ø

**L🡪**

a0

Ø

**L🡪**

a0

Ø

a1

**L🡪**

a0

a2 

Ø

a1

**操作需要“头指针”和两个辅助指针。即**

**LNode \*L,\*p,\*r;**

**第一步：创建专用的“头结点”（其数据域不予理睬）**

**L🡪**

Ø

**L=(LNode \*)malloc(sizeof(LNode));**

**L🡪next=NULL;**

**第二步：创建第一个结点（尾结点也是实际的头结点）**

**p**

**L🡪**

Ø

a0

**r**

**r=L; p=(LNode \*)malloc(sizeof(LNode));**

**p🡪data=a[0];**

**p🡪next=NULL;**

**r🡪next=p;**

**r=p;**

**p**

**L🡪**

Ø

a0

**r**

**第三步：创建第二个尾结点**

**p**

**L🡪**

Ø

a1

a0

**r**

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

**p🡪data=a[1];**

**p🡪next=NULL;**

**r🡪next=p;**

**r=p;**

**p**

**L🡪**

Ø

a1

a0

**r**

**………………………**

**说明：创建n个结点，数据由数组提供**

**void LinkListCreat2(LinkList L,ElemType a[],int n)**

**{//L为头指针，p,r为协助指针**

**int i;**

**LinkList p,r;**

**L=(LNode \*)malloc(sizeof(LNode));**

**if(L==NULL){ printf(“申请空间失败！”);exit(0);}**

**L🡪next=NULL;**

**// 生成“带”头的结点（对数据域不予理睬）**

**r=L;**

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

**{**

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

**if(p==NULL)**

**{printf(“申请空间失败！”);exit(0);}**

**p🡪data=a[i];**

**p🡪next=NULL;**

**r🡪next=p;**

**r=p;**

**}**

**}**

**时间复杂度：O(n)**

**§2.3.2.2单链表的遍历（visit,又称访问，周游和走链）**

**单链表只能从头指针所指的位置开始顺着指针方向进行遍历各结点，不具有随机（从任一结点）和反向遍历的特点。**

**●对于“带”头结点的单链表，由于头结点的数据域无意义，所以遍历时应将其跳过，即p=L🡪next**

**L🡪**

a1

a3

a2

Ø

**p**

**●“不带”头结点的单链表，遍历开始为p=L**

**L🡪**

a3

a2

a1

**p**

**操作需要“头指针”和一个辅助指针。即**

**LNode \*L,\*p;或LinkList L,p;**

**void LinkListvisit(LinkList L)**

**{//L为头指针，p为协助指针**

**LinkList p=L🡪next;(不带头结点的p=L;)**

**while(p!=NULL)**

**{printf("%5d",p🡪data);p=p🡪next;}**

**}**

**时间复杂度：O(n)**

**遍历分析1：若不用协助指针变量，表达繁琐**

**L🡪**

a3

a2

a1

**L🡪data（a1）**

**L🡪next🡪data（a2）**

**L🡪next🡪next🡪data（a3）**

**L🡪next🡪next🡪next=NULL**

**遍历分析2：用协助指针变量p，使用p=p🡪next表达简洁**

**述评：**

**●“头插法”对于三个结点的链表创建的次序是a0,a1,a2,而遍历的次序则是a2,a1,a0,从而形成“先进后出”或“后进先出”的结构（即创建与遍历次序恰好相反），这种数据结构被称为“栈”结构；**

**●“尾插法”对于三个结点的链表创建和遍历的次序均为a0,a1,a2,从而形成“先进先出”或“后进后出”的结构（即创建与遍历次序恰好一致），这种数据结构被称为“队列”结构。**

**以上两种数据结构下一章将作详细介绍。**

**§2.3.2.3“不带头结点”单链表的插入操作**

**插入分为三种情形：表头插入、表中间插入和表尾插入**

**操作需要“头指针”和两个或三个辅助指针。即**

**LNode \*L,\*p,\*q,\*r;**

**使用3个还是4个指针，越多可能某些语句可能简练，反之可能复杂些。**

**下面看看结点插入的三种情况：**

**（1）在表头插入结点（形成新的表头）**

**插入前链表的状态：**

**L🡪**

a3

a2

a1

**申请到一个结点，准备插入到表头：**

**q🡪**

a0

**插入后链表的状态：**

**L🡪**

a3

a2

a1

a0

**q**

**第1步：q=(LinkList)malloc(sizeof(Lnode));**

**第2步：q🡪data=a0;**

**第3步：q🡪next=L;**

**第4步：L=q;**

**(2)在表中间插入结点**

**第一种方案：使用4个指针变量**

**LNode \*L,\*p,\*q,\*r;**

**插入前链表的状态：**

**r p**

**L🡪**

a3

a2

a1

**q🡪**

a0

**申请到一个结点a0，准备插入到结点a2和a3之间。显然，a0要插入到r所指结点之后（后插）和p所指的结点之前（前插）：**

**插入后链表的状态：**

**r q p**

**L🡪**

a3

a0

a2

a1

**第1步：q=(LinkList)malloc(sizeof(Lnode));**

**第2步：q🡪data=a0;**

**第3步：q🡪next=p;**

**第4步：r🡪next=q;**

**第二种方案：使用3个指针变量**

**LNode \*L,\*p,\*q;**

**前例若不用r指针变量，依下列情形对p进行后插。**

**插入前链表的状态：**

**p**

**L🡪**

a3

a2

a1

**申请到一个结点，准备插入到结点a2和a3之间：**

**q🡪**

a0

**插入后链表的状态：**

**p q**

**L🡪**

a3

a0

a2

a1

**第1步：q=(LinkList)malloc(sizeof(Lnode));**

**第2步：q🡪data=a0;**

**第3步：q🡪next=p🡪next;（语句复杂些）**

**第4步：p🡪next=q;**

**思考：上例如果对p进行前插是否可行？进一步讨论结点插入。**

**说明：在单链表上插入结点，必须知道被插结点的前驱指针是否能明确表示出来，以上所举之例三种情形的前驱结点的指针分别是L,r和p。若下列情形对p进行前插，被插结点的前驱指针（红色指针）不好表达，所以这种插法不易进行，因此不要采用。如：**

**p**

**L🡪**

a3

a2

a1

**q🡪**

a0

**第1步：q=(LinkList)malloc(sizeof(Lnode));**

**第2步：q🡪data=a0;**

**第3步：q🡪next=p;**

**第4步： =q;**

**(3)在表尾插入结点（形成新的表尾）**

**插入前链表的状态：**

**p**

**L🡪**

a3

a2

a1

**申请到一个结点，准备插入到表尾：**

**q🡪**

a0

**插入后链表的状态：**

**p q**

**L🡪**

a0

a3

a2

a1

**第1步：q=(LinkList)malloc(sizeof(Lnode));**

**第2步：q🡪data=a0;**

**第3步：q🡪next=NULL;**

**第4步：p🡪next=q;**

**下面是单链表插入的程序：**

**void LinkListInsert(LinkList L,**

**LinkList p,ElemType e)**

**{ //后插入**

**LinkList q;**

**q=(LNode \*)malloc(sizeof(LNode));**

**if(q==NULL)**

**{printf(“申请空间失败！”);exit(0);}**

**q🡪data=e;**

**if(p==L){q🡪next=L;L=q;exit(0);}**

**pre=L;**

**while(pre!=NULL&&pre🡪next!=p)// 找p的前驱**

**pre=pre🡪next;**

**q🡪next=pre🡪next;**

**pre🡪next=q;**

**}**

**时间复杂度：O(n)**

**§2.3.2.4“不带头结点”单链表结点的删除**

1. **头结点的删除**

**使用辅助指针p**

**p**

**L🡪**

a3

a2

a1

**L=p🡪next;**

**p L**

a3

a2

a1

**此时p所指的结点已不属于链表。**

**free(p);**

**L🡪**

a3

a2

**如果不用辅助指针变量也可删除头结点a1，但其无法释放**

**L🡪**

a3

a2

a1

**L=L🡪next;**

**L**

a3

a2

a1

**(2)中间结点(含尾结点)的删除**

**●使用两个辅助指针变量删除p所指的结点，直观且语句表达较简练**

**r p**

**L🡪**

a3

a2

a1

**r🡪next=p🡪next;**

**r p**

**L🡪**

a3

a2

a1

**此时p所指的结点已不再链表中。**

**free(p);**

**r**

**L🡪**

a3

a1

**●如果只用一个辅助指针删除p所指的直接后继结点**

**p**

**L🡪**

a3

a2

a1

**p🡪next=p🡪next🡪next;**

**语句表达复杂且a2结点无法释放。**

**p**

**L🡪**

a3

a2

a1

**说明：在单链表上删除结点，必须知道前驱结点。**

**void LinkListDel(LinkList L,**

**ElemType e,LinkList p)**

**{**

**LinkList pre=L;**

**while(pre!=NULL&&pre🡪next🡪data!=e)**

**pre=pre🡪next;**

**p=pre🡪next;**

**if(p!=NULL)**

**{ if(p==L)L=p🡪next;**

**else pre🡪next=p🡪next;**

**free(p);**

**}**

**}**

**时间复杂度：O(n)**

**例p2-3.c单链表的创建（正生成即尾插法）、遍历、插入（前插和后插）和删除**

**#include"stdio.h"**

**typedef int ElemType;**

**typedef struct Node**

**{**

**ElemType data;**

**struct Node \*next;**

**}LNode,\*LinkList;**

**LinkList Creat()//创建**

**{**

**LinkList L,r,s;**

**ElemType value;**

**L=(LinkList)malloc(sizeof(LNode)); //建立头结点**

**L->next=NULL;**

**r=L;//尾指针r初值指向头结点**

**printf("生成单链表，输入int，-1作为结束： ");**

**scanf("%d",&value);**

**while(value!=-1)**

**{**

**s=(LinkList)malloc(sizeof(LNode));**

**//生成新结点s**

**s->data=value;//将新结点插入表尾**

**r->next=s;//尾指针r指向新的表尾**

**r=s;**

**scanf("%d",&value);//读入下一个结点值**

**}**

**r->next=NULL;//将尾结点的指针域置为空**

**return L;**

**}**

**void visit(LinkList L) //遍历**

**{**

**LinkList p=L->next;**

**while(p!=NULL)**

**{**

**printf("%5d",p->data);**

**p=p->next;**

**}**

**printf("\n\n");**

**}**

**void InsertBefore(LinkList L,int pos,**

**ElemType value)//前插**

**{//在第pos个结点前插入结点，其值为value**

**LinkList p=L->next,q=L,s;**

**//指针q指向p的前驱结点**

**int n=1;**

**while(p!=NULL&&n<pos)**

**//链表非空且未达到第pos个结点时**

**{**

**q=p;**

**p=p->next;//遍历下一个结点**

**n++;**

**}**

**if(p!=NULL)//找到第pos个结点**

**{**

**s=(LinkList)malloc(sizeof(LNode));**

**//申请新结点**

**s->data=value;**

**s->next=p;//将s插到p之前**

**q->next=s;**

**}**

**else printf("插入失败！\n");**

**}**

**void InsertAfter(LinkList L,int pos,**

**ElemType value)//后插**

**{//在第pos个结点后插入结点，其值为value**

**LinkList p=L->next,s;**

**int n=1;**

**while(p!=NULL&&n<pos)//链表非空且未达到第pos个结点时**

**{**

**p=p->next;//遍历下一个结点**

**n++;**

**}**

**if(p!=NULL)//找到第pos个结点**

**{**

**s=(LinkList)malloc(sizeof(LNode));**

**//申请新结点**

**s->data=value;**

**s->next=p->next;//将s插到p之前**

**p->next=s;**

**}**

**else printf("插入失败！\n");**

**}**

**void Delete(LinkList L,int pos)//删除**

**{//删除单链表的第pos个结点**

**LinkList p=L->next,q=L;//指针q指向p的前驱结点**

**int n=1;**

**while(p!=NULL&&n<pos)//链表非空且未达到第pos个结点时**

**{**

**q=p;**

**p=p->next;//遍历下一个结点**

**n++;**

**}**

**if(p!=NULL)//找到第pos个结点**

**{**

**q->next=p->next;//删除p结点**

**free(p);//释放p所指的存储单元**

**}**

**else printf("删除失败！\n\n");**

**}**

**void main()**

**{**

**LinkList L;**

**int pos;**

**ElemType value;**

**L=Creat();//创建**

**printf("创建的链表是：\t");visit(L);//创建后的遍历**

**printf("前插输入位置int pos和插入值int value: ");**

**scanf("%d%d",&pos,&value);**

**InsertBefore(L,pos,value);//前插**

**printf("前插后的链表是:\t");visit(L);**

**//前插后的遍历**

**printf("后插输入位置int pos和插入值int value: ");**

**scanf("%d%d",&pos,&value);**

**InsertAfter(L,pos,value);//后插**

**printf("后插后的链表是:\t");visit(L);**

**//后插后的遍历**

**printf("删除输入位置int pos: ");scanf("%d",&pos);**

**Delete(L,pos);//删除**

**printf("删除后的链表是:\t");visit(L);**

**//删除后的遍历**

**}**

**例p2-4.c创建一个“带头结点”的单链表，消除连续的重复结点后将其反转（逆置）**

**创建单链表：**

35

Head

Ø

47

35

**消除重复结点：**

35

Head

Ø

47

**反转（逆置）：**

Head

Ø

35

47

**#include"stdio.h"**

**typedef int DataType;**

**typedef struct node**

**{**

**DataType data;**

**struct node \*next;**

**}LinkList;**

**LinkList \*CreatList()//用尾插法创建链表**

**{**

**LinkList \*head,\*r,\*s;**

**DataType value;**

**head=(LinkList\*)malloc(sizeof(LinkList));**

**//建立头结点**

**head->next=NULL;**

**r=head;//尾指针r初值指向头结点**

**printf("生成单链表，输入若干数据int，**

**-1作为结束： ");**

**scanf("%d",&value);**

**while(value!=-1)**

**{**

**s=(LinkList \*)malloc(sizeof(LinkList));**

**//生成新结点s**

**s->data=value;//将新结点插入表尾**

**r->next=s;//尾指针r指向新的表尾**

**r=s;**

**scanf("%d",&value);//读入下一个结点值**

**}**

**r->next=NULL;//将尾结点的指针域置为空**

**return head;**

**}**

**void PrnList(LinkList \*head) //遍历**

**{**

**LinkList \*p=head->next;**

**while(p!=NULL)**

**{**

**printf("%5d",p->data);**

**p=p->next;**

**}**

**printf("\n\n");**

**}**

**void DelNode(LinkList \*head)//删除重复的结点**

**{**

**LinkList \*p=head->next,\*q;**

**//p从第一个数据结点开始**

**while(p->next!=NULL)**

**{**

**q=p->next;//q指向p的后续结点**

**if(p->data==q->data)**

**//判断p和q的结点是否相同**

**{**

**p->next=q->next;//删除q所指的相同值结点**

**free(q);**

**}**

**else p=p->next;**

**}**

**}**

**void Invert(LinkList \*head)//逆置单链表**

**{**

**LinkList \*p=head->next,\*q;**

**//p从第一个数据结点开始**

**head->next=NULL;**

**while(p!=NULL)**

**{**

**q=p;//q指向当前的结点**

**p=p->next;**

**q->next=head->next;**

**//将q结点插入到头结点之后**

**head->next=q;**

**}**

**}**

**void main()**

**{**

**LinkList \*head;**

**head=CreatList();//创建**

**printf("创建的链表是:");PrnList(head);**

**//创建后的遍历**

**DelNode(head);//删除重复的结点**

**printf("删除重复结点后的链表是:");**

**PrnList(head);**

**Invert(head);//逆置单链表**

**printf("逆置后的链表是:");**

**PrnList(head);//逆置后的遍历**

**}**

**例p2-5.c 两个一元高次多项式相加**

**如：()+()=**

**=**

**qa🡪**

1 5

coef exp

**qb🡪**

2 4

coef exp

**qc🡪**

1 5

coef exp

**#include<stdio.h>**

**typedef struct node**

**{**

**int coef; //系数**

**int exp; //指数**

**struct node \*next;**

**}PolyList;//多项式链表的类型名**

**PolyList \*CreatList()//创建**

**{**

**PolyList \*head,\*r,\*s;**

**int co,ex;**

**head=(PolyList\*)malloc(sizeof(PolyList));**

**//建立头结点**

**head->next=NULL;**

**r=head;//尾指针r初值指向头结点**

**printf("成对输入多项式的系数和指数**

**(int co,ex)，最后系数和指数9999作为结束：");**

**scanf("%d%d",&co,&ex);**

**while(co!=9999&&ex!=9999)**

**{**

**s=(PolyList \*)malloc(sizeof(PolyList));**

**//生成新结点s**

**s🡪coef=co;//将新结点插入表尾**

**s🡪exp=ex;**

**r🡪next=s;//尾指针r指向新的表尾**

**r=s;**

**scanf("%d%d",&co,&ex);**

**//读入下一个结点值**

**}**

**r🡪next=NULL;//将尾结点的指针域置为空**

**return head;**

**}**

**void PrnList(PolyList \*head) //遍历**

**{**

**PolyList \*p=head🡪next;**

**int noden=0;**

**while(p!=NULL)**

**{**

**switch(++noden)**

**{**

**case 1: //多项式的首项**

**switch(p🡪exp)//指数为0，1，>1**

**{**

**case 0: printf("%d",p🡪coef);break;**

**case 1: switch(p🡪coef)**

**//系数为-1，1，其它**

**{**

**case -1: printf("-X");break;**

**case 1: printf("X");break;**

**default: printf("%dX",p->coef);**

**}**

**break;**

**default:switch(p🡪coef)**

**//系数为-1，1，其它**

**{**

**case -1:**

**printf("-X\*\*%d",p🡪exp);break;**

**case 1:**

**printf("X\*\*%d",p🡪exp);break;**

**default:**

**printf("%dX\*\*%d",p🡪coef,p🡪exp);**

**}**

**}**

**break;**

**default://多项式从第二项至末项**

**switch(p🡪exp)**

**{**

**case 0:**

**if(p🡪coef>0)**

**printf("+%d",p🡪coef);**

**else printf("%d",p🡪coef);**

**break;**

**case 1:**

**switch(p🡪coef)**

**{**

**case -1:printf("-X");break;**

**case 1: printf("+X");break;**

**default:if(p🡪coef>0)**

**printf("+%dX",p🡪coef);**

**else printf("%dX",p🡪coef);**

**}**

**break;**

**default:switch(p🡪coef)**

**{**

**case -1:**

**printf("-X\*\*%d",p🡪exp);break;**

**case 1:**

**printf("+X\*\*%d",p🡪exp);break;**

**default: if(p🡪coef>0)**

**printf("+%dX\*\*%d",p🡪coef,p🡪exp);**

**else**

**printf("%dX\*\*%d",p🡪coef,p🡪exp);**

**}**

**}**

**}**

**p=p🡪next;**

**}**

**printf("\n\n");**

**}**

**PolyList \*PolyAdd(PolyList \*pa,PolyList \*pb)**

**//计算两个一元多项式相加**

**{**

**PolyList \*qa=pa🡪next,\*qb=pb🡪next,\*qc=pa,\*s;**

**//qa指向链表pa中的结点，初始时指向第一个结点**

**//qb指向链表pb中的结点，初始时指向第一个结点**

**//利用pa所指链表头结点，作为“和多项式”的头结点，**

**qc指向“和多项式”的尾结点**

**while(qa!=NULL&&qb!=NULL)**

**{**

**if(qa🡪exp>qb🡪exp)**

**{**

**qc🡪next=qa;**

**//将qa所指结点插入“和多项式”的尾部**

**qc=qa;**

**qa=qa🡪next;**

**}**

**else if(qa🡪exp<qb🡪exp)**

**{**

**qc🡪next=qb;**

**//将qb所指结点插入“和多项式”的尾部**

**qc=qb;**

**qb=qb🡪next;**

**}**

**else**

**{**

**qa🡪coef+=qb🡪coef;**

**//系数相加，结果存于qa所指结点的系数域**

**if(qa🡪coef!=0)**

**{**

**qc🡪next=qa;**

**//将qa所指结点插入“和多项式”的尾部**

**qc=qa;**

**qa=qa🡪next;**

**s=qb;**

**//删除qb所指结点，并释放存储单元**

**qb=qb🡪next;**

**free(s);**

**}**

**else //当系数为0**

**{**

**s=qa;**

**//删除qa所指结点，并释放存储单元**

**qa=qa🡪next;**

**free(s);**

**s=qb;**

**//删除qb所指结点，并释放存储单元**

**qb=qb🡪next;**

**free(s);**

**}**

**}**

**}**

**qc🡪next=NULL;**

**if(qa!=NULL)qc🡪next=qa;**

**//插入pa中剩余的结点**

**if(qb!=NULL)qc🡪next=qb;**

**//插入pb中剩余的结点**

**free(pb);//释放pb所指的头结点**

**return pa;**

**}**

**void main()**

**{**

**PolyList \*pa,\*pb;**

**printf("生成pa单链表");pa=CreatList();**

**printf("pa单链表: ");PrnList(pa);**

**printf("生成pb单链表");pb=CreatList();**

**printf("pb单链表: ");PrnList(pb);**

**pa=PolyAdd(pa,pb);//两个一元多项式相加**

**printf("pa+pb单链表: ");PrnList(pa);**

**}**

**2.3.3循环链表**

**一个单链表，尾结点的指针域为空指针（NULL），若将头指针置入其中，使得头尾相连，就构成单循环链表。**

**下列空表只要：h🡪next=h;非空表:p🡪next=h;**

h

Ø

47

35

h

Ø

p

**Joseph.c 约瑟夫问题-循环单链表之例**

**在海上海盗劫持一群船员（人数为约瑟夫环数），命令海员站成一圈儿，从某个人开始报数至某数（称为密码）将其推入海中，再开始重新报数，……，输出被害海员的序列，该故事称为约瑟夫问题。**

**#include<stdio.h>**

**typedef struct Node**

**{**

**int data;**

**struct Node \*next;**

**}Node;**

**Node \*Creat(int n);**

**void Joseph(Node \*rear,int m);**

**void main()**

**{**

**int m,n;**

**Node \*rear=NULL;**

**printf("输入约瑟夫环的长度: ");scanf("%d",&n);**

**printf("输入密码: ");scanf("%d",&m);**

**rear=Creat(n);**

**Joseph(rear,m);**

**}**

**Node \*Creat(int n)**

**{**

**int i;**

**Node \*rear=NULL,\*s;**

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

**rear->data=1;**

**rear->next=rear;**

**for(i=2;i<=n;i++)**

**{**

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

**s->data=i;**

**s->next=rear->next;**

**rear->next=s;**

**rear=s;**

**}**

**return rear;**

**}**

**void Joseph(Node \*rear,int m)**

**{**

**Node \*pre=rear,\*p=rear->next,\*q;**

**int count=1;**

**printf("出环的顺序是:\n");**

**while(p->next!=p)**

**if(count<m)**

**{**

**pre=p;p=p->next;count++;**

**}**

**else**

**{**

**printf("%4d",p->data);**

**q=p;**

**pre->next=p->next;**

**p=pre->next;**

**free(q);**

**count=1;**

**}**

**printf("%4d\n",p->data);**

**free(p);**

**}**

**2.3.4双向链表**

**一个结点若已有一个指向后继（next）的指针，再设置一个指向前驱（prior）的指针，称这样的链表为双向链表。显然不论顺着前驱或后继指针均可以遍历链表。**

**双向链表如果头尾相接，可以形成为双向循环链表，遍历更加方便。**

|  |  |  |
| --- | --- | --- |
| **prior** | **data** | **next** |

**空和非空的双向链表：**

Ø

A

M

Ø

**head**

**head**

**空和非空的双向循环链表：**

Ø

A

M

Ø

**head**

**head**

**定义：**

**typedef char ElemType;**

**typedef struct Node**

**{**

**ElemType data;**

**struct Node \*prior,\*next;**

**}DLNode,\*DLinkList;**

**DLNode \*head,\*p,\*q,\*r,\*s;**

**或 DLinkList head,p,q,r,s;**

**双向链表的建立、插入和删除**

**1. 双向链表的建立 r**

**(1)建立空表 head🡪**

Ø

**head=r=(DLNode \*)malloc(sizeof(DLNode));**

**r🡪next=r🡪prior=NULL;**

**(2)申请一个新结点s并将数据输入数据域**

**s=(DLNode \*)malloc(sizeof(DLNode));**

**s🡪data=’A’；**

**s🡪**

A

**(3)将新结点插入到链表表尾:**

**r🡪next=s; s🡪prior=r;**

**r s**

**head🡪**

A

Ø

**（4）更新尾结点指针r,如果到了最后的尾结点指针域为空：**

**r=s;r🡪next=NULL;**

**r s**

**head🡪**

A

Ø

**2．双向链表的插入**

**●在p所指的结点前面插入（前插法）**

**s=(DLNode \*)malloc(sizeof(DLNode));**

**s🡪data=’X’；**

**(1)s🡪prior=p🡪prior;**

**(2)s🡪next=p;**

**(3)p🡪prior🡪next=s;//原指针p🡪prior🡪next被修改**

**(4)p🡪prior=s; //原指针p🡪prior被修改**

M

N

p

X

s

1

2

3

4

**若结点再设置一个指针r指向结点M,上述插入的表达将会变得简练。**

**s=(DLNode \*)malloc(sizeof(DLNode));**

**s🡪data=’X’；**

**(1)s🡪prior=r;**

**(2)s🡪next=p;**

**(3)r🡪next=s;//原指针p🡪prior🡪next被修改**

**(4)p🡪prior=s; //原指针p🡪prior被修改**

M

N

p

X

s

1

2

3

4

r

**3．双向链表的删除**

**将p所指的结点删除（利用图示指针1和2的对称性）**

M

N

p

X

1

2

1. **printf(“%c”,p**🡪**data);**

**/\*保存被删结点的数据\*/**

**（2）p🡪prior🡪next=p🡪next;**

**（3）p🡪next🡪prior=p🡪prior; /\* 以上两条对称 \*/**

**（4）free(p);**

**2.4 顺序表和链表的比较**

**本章讨论了线性表的逻辑结构和两种存储结构：顺序表和链表**

**（1）顺序表**

**存储结构的优点：**

**●方法简单，可以用数组实现，容易理解；**

**●由于元素的逻辑关系与物理关系一致，所以不用额外的开销；**

**●按元素下标遍历即可。**

**存储结构的缺点：**

**●由于数组下标必须有范围，实际问题元素的个数不好确定，所以对范围要进行估计，估计过大造成浪费，估计过小造成溢出；**

**●在插入和删除过程中，引起数据大量移动，效率较低。**

**（2）链表**

**可以克服上述缺点，但需要增加操作指针，这增加了开销。**