第5章 链　表

本章学习目标

- 掌握单向链表的创建和增加、删除、修改、查询等功能的实现。
- 掌握双向链表的创建和增加、删除、修改、查询等功能的实现。
- 理解内核链表的技术原理。
- 掌握内核链表在用户程序中的应用。

链表是一种常见的数据结构，由一系列结点组成，结点可以在程序运行时通过动态存储分配进行创建。每个结点至少包括两个域：数据域和指针域。数据域用于存储数据，指针域用于存储指向下一个结点的地址，建立与下一个结点的联系。各个结点通过指针链接构成一个链式结构，即链表。链表有多种类型：单向链表、双向链表和循环链表等。


扫一扫

5.1 单向链表

5.1.1 单链表结构与链表结点类型

单向链表简称单链表，其特点是链表的链接方向是单向的。本节介绍的单链表包含头结点，因此链表中的结点分为两种：头结点和一般结点（也称数据结点）。这两种结点的类型一样，都是某种结构体类型，但是头结点的数据域一般不使用，一般结点的数据域存放具体的用户数据。对单链表的访问要从头结点开始顺序进行。一般结点的第一个结点称为首结点，最后一个结点称为尾结点。头结点由头指针指向。尾结点的指针域为 NULL。

带头结点的单链表如图 5.1 所示。

单链表结点的类型定义如下。

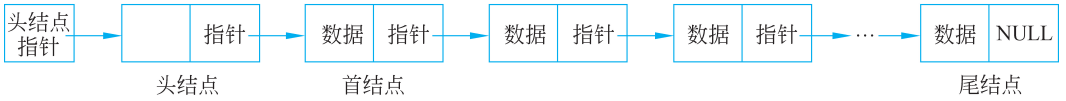

图 5.1　带头结点的单链表

```
#define ElemType int
typedef struct slnode{
  ElemType data;        //数据域
  struct slnode *next;  //指针域
}SLNode, *PSLNode;
```

数据域的数据类型为宏定义 ElemType，可根据实际需求定义 ElemType 的具体数据类型，比如定义为 int 型。指针域使用了结构体的自引用，也就是在结构体内部包含了指向自身类型的指针成员。使用 typedef 关键字将 struct slnode 类型定义为 SLNode，将 struct slnode * 类型定义为 PSLNode。使用 typedef 可以给已有类型起一个简短、易记、意义明确的新名字。后续可以使用 SLNode 定义结构体变量，使用 PSLNode 定义指向 SLNode 结构体变量的指针。

5.1.2　创建单链表

可以采用头插法或尾插法创建单链表。头插法创建单链表如图 5.2 所示。尾插法将一个新结点链接到已有链表的尾部。尾插法创建单链表如图 5.3 所示。

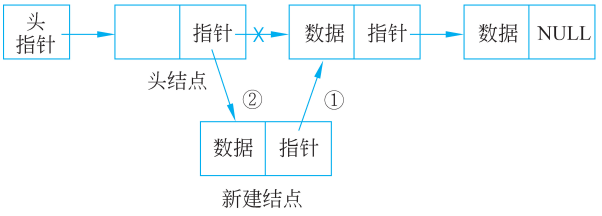

图 5.2　头插法创建单链表

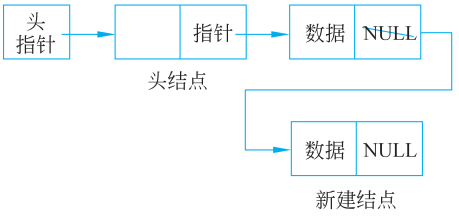

图 5.3　尾插法创建单链表

注意　本章所有源代码文件在本书配套资源的“src/第 5 章”目录中。

下面的 ListBuildH 和 ListBuildR 函数，都是根据含有 n 个元素的数组 a 创建带头结点的单链表。

1. 头插法创建单链表的函数 ListBuildH

```
#include <stdio.h>
#include <stdbool.h>
#include <malloc.h>
typedef struct{ //数据域类型，可以根据实际需求添加更多的结构体成员
  int data;
}DATA;
typedef struct slnode{
  DATA data;            //数据域
  struct slnode *next;  //指针域
}SLNode, *PSLNode;
PSLNode ListBuildH(DATA a[], int n){            //头插法创建单链表
  PSLNode L=(PSLNode)malloc(sizeof(SLNode)); L->next=NULL; //创建头结点
  for(int i=0; i<n; i++){                       //循环创建新结点
```

```
    PSLNode s=(PSLNode)malloc(sizeof(SLNode));  //为新结点分配存储空间
    s->data=a[i];               //将数组a中的元素值赋值给新结点s数据域
    s->next=L->next;            //新结点s的指针域指向首结点
    L->next=s;                  //头结点的指针域指向新结点s
  }
  return L;
}
```

2. 尾插法创建单链表的函数 ListBuildR

```
void ListBuildR(SLNode **L, DATA a[], int n){   //尾插法创建单链表
  PSLNode r=*L=(PSLNode)malloc(sizeof(SLNode)); //创建头结点
  for(int i=0; i<n; i++){                       //循环建立数据结点
    PSLNode s=(PSLNode)malloc(sizeof(SLNode));  //为新结点分配存储空间
    s->data=a[i];               //将数组a中的元素值赋值给新结点s数据域
    r->next=s;                  //尾结点的指针域指向新结点s
    r=s;                        //结点s成为新的尾结点，尾指针r始终指向尾结点
  }
  r->next=NULL;                 //尾结点的指针域置为NULL
} //由于要在本函数内修改传入的头指针L的值，因此第一个参数L为二级指针
```

5.1.3 插入结点

除头插法和尾插法插入结点外，更灵活的插入结点的方法是在指定位置插入结点。插入结点的示意图如图 5.4 所示。在单链表中插入结点的函数如下。

```
bool ListInsert(PSLNode L, int i, DATA e){ //在单链表L第i个位置插入值为e的结点
  int j=0;
  while(j<i-1 && L!=NULL){      //在单链表中查找第i-1个结点，由L指向
    L=L->next;
    j++;
  }
  if(L==NULL) return false;     //未找到第i-1个结点，返回false
  else{                         //找到第i-1个结点，插入新建结点并返回true
    PSLNode s=(PSLNode)malloc(sizeof(SLNode));  //为新结点分配存储空间
    s->data=e;          //新结点数据域被赋值为e
    s->next=L->next;    //新结点指针域指向L所指结点的下一个结点
    L->next=s;          //L所指向结点的指针域指向新结点
    return true;
  }
}
```

5.1.4 删除结点

删除数据结点的示意图如图 5.5 所示。在单链表中删除数据结点的函数如下。

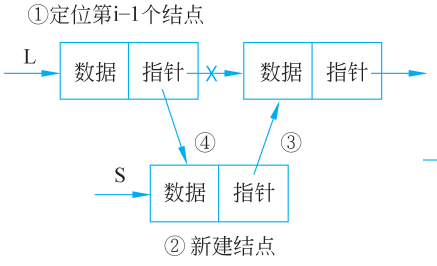

图 5.4 插入结点的示意图

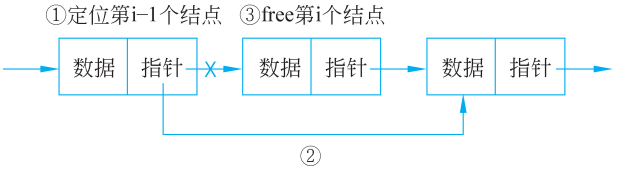

图 5.5 删除数据结点的示意图

```
bool ListDelete(PSLNode L, int i, DATA *pe){ //单链表L中若存在第i个结点，则将其删除，
  int j=0;                                   //并且将其数据域值赋给pe所指的变量
  while(j<i-1 && L!=NULL){      //在单链表中查找第i-1个结点，由L指向
    L=L->next;
    j++;
  }
  if(L==NULL) return false;     //未找到第i-1个结点，返回false
  else{                         //找到第i-1个结点L
    PSLNode s=L->next;          //s指向第i个结点
    if(s==NULL) return false;   //若不存在第i个结点，则返回false
    *pe=s->data;          //将要删除的结点的数据域值复制给pe所指的变量
    L->next=s->next;      //L所指向结点的指针域指向被删除结点s的下一个结点
    free(s);              //释放s结点，即成功删除第i个结点
    return true;          //返回true
  }
}
```

5.1.5 读取结点

在单链表中读取指定位置数据结点内容的函数如下。

```
bool ListGetElem(PSLNode L, int i, DATA *pe){   //读取单链表L中若存在第i个结点，
  int j=0;                                      //则将其数据域值赋给pe所指的变量
  while(j<i && L!=NULL){
    j++;
    L=L->next;
  }
  if(L==NULL) return false;     //未找到第i个结点，返回false
  else{                         //存在第i个结点，返回true
    *pe=L->data;                //将第i个结点的数据域值复制给pe所指的变量
    return true;
  }
}
```

5.1.6 查找结点

查找函数根据传入的 x 值，在单链表中进行遍历，查找 x 值所在的结点，然后返回该结点的地址。

```
bool ListEmpty(PSLNode L){      //判断链表是否为空表，无数据结点为空表
  if(!L){printf("链表无头结点\n"); return true;}
  return(L->next==NULL);        //若单链表L没有数据结点，则返回true，否则返回false
}
PSLNode ListFind(PSLNode L, DATA x){    //查找结点
  if(ListEmpty(L)){printf("链表为空\n"); return NULL;}
  L=L->next;                            //L指向第一个数据结点
  while(L){                             //存在数据结点
    if(L->data.data==x.data) return L;  //找到，返回x结点指针
    else L=L->next;
  }
  return NULL;                          //没有找到，返回NULL
}
```

5.1.7 打印单链表

打印单链表函数从第一个数据结点开始依次将所有数据结点中的数据域值输出。

```
int ListLength(PSLNode L){       //返回单链表L中数据结点的个数
  int i=0;                       //L指向头结点
  while(L->next!=NULL){          //统计数据结点个数
    i++;
    L=L->next;
  }
  return i;                      //返回结点个数
}
void ListOutput(PSLNode L){ //逐一扫描单链表L的每个数据结点，并显示各结点的data域值
  if(ListEmpty(L)){printf("链表为空\n"); return;}
  printf("链表数据结点个数为: %d\n", ListLength(L));
  int i=1;
  L=L->next;                     //L指向第1个数据结点
  while(L){                      //统计结点个数
    printf("第%d个结点的值: %d\n", i++, L->data.data);
    L=L->next;                   //L指向下一个数据结点
  }
}
```

5.1.8 逆转单链表

假设单链表数据结点的数据域值依次为 A、B、C、D，那么将单链表逆转后，数据结点的数据域值依次为 D、C、B、A。逆转单链表的函数如下。

```
void ListReverse(PSLNode L){     //将单链表L中的所有数据结点位置逆转
  if(ListEmpty(L)){printf("链表为空\n"); return;}
  PSLNode p=L->next, q;
  L->next=NULL;
  while(p!=NULL){        //while循环中采用头插法逆转所有数据结点位置
    q=p->next;
    p->next=L->next;
    L->next=p;
    p=q;
  }
}
```

5.1.9 构建单向循环链表

带头结点的单向循环链表示意图如图 5.6 所示。将单向循环链表构建为单循环链表的方法：让尾结点的指针域指向头结点。从单向循环链表中的任一结点出发均可找到链表中的其他结点。构建单向循环链表的函数如下。

```
void ListBuildC(PSLNode L){      //构建循环链表L
  if(!L){printf("链表无头结点\n"); return;}
  PSLNode s=L;
  while(s->next) s=s->next;      //s指向最后一个结点
  s->next=L;                     //最后一个结点的指针域指向头结点
}
```

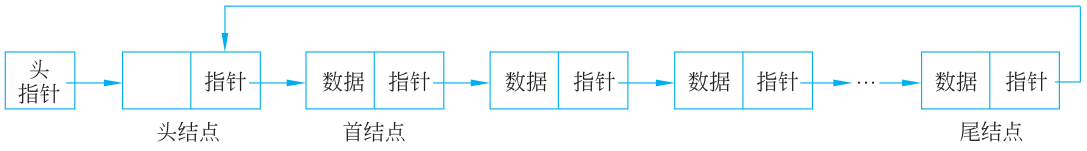

图 5.6 带头结点的单向循环链表示意图

5.1.10 销毁单链表

销毁链表就是将链表中的所有结点占据的内存空间释放掉。销毁链表的函数ListDestroy如下。如果调用ListDestroy函数销毁由指针L指向的链表后，需要将指针L置为NULL，否则指针L会成为野指针。

```
void ListDestroy(PSLNode L){     //销毁单链表L
  while(L){                      //逐一释放每个结点占用的内存空间
    PSLNode s=L;                 //s指向第1个结点
    PSLNode L=L->next;           //L指向下一个结点
    free(s);                     //释放结点s占用的内存空间
  }
}
```

5.1.11 主函数及测试结果

在main函数中通过对上面函数的调用创建、操作单链表。main函数的定义如下。

```
int ListLengthC(PSLNode L){      //返回单循环链表L中数据结点的个数
  int i=0; PSLNode p=L->next;    //L指向头结点
  while(p!=L){                   //统计数据结点个数
    i++; p=p->next;
  } return i;                    //返回结点个数
}
int main(void){
  DATA dat1[5]={ {11}, {12}, {13}, {14}, {15} };
  DATA dat2[5]={ {21}, {22}, {23}, {24}, {25} };
  PSLNode   L1=ListBuildH(dat1, 5);      ListOutput(L1); //头插法建表、输出
  PSLNode L2;  ListBuildR(&L2, dat2, 5); ListOutput(L2); //尾插法建表、输出
  DATA d1={100}; ListInsert(L1, 3, d1);  ListOutput(L1); //插入数据结点、输出
  DATA d2; ListDelete(L2, 2, &d2);  printf("d2=%d\n", d2.data); ListOutput(L2);
  DATA d3; ListGetElem(L2, 2, &d3); printf("d3=%d\n", d3.data);
  PSLNode p=ListFind(L1, d1);  printf("d1=%d\n", p->data.data);
  ListReverse(L1);  ListOutput(L1);     //逆转链表、输出
  ListBuildC(L2);  printf("单向循环链表数据结点个数为：%d\n", ListLengthC(L2));
  p=L2;                          //将链表L2构建为单向循环链表，下面的for循环将
  for(int i=1; i<11; i++){       //链表L2中的数据结点中数据域值输出两轮
    if(p!=L2) printf("%d ", p->data.data);
    if( !(i%(ListLengthC(L2)+1)) ) printf("\n");
    p=p->next;
  }
}
```

本小节的所有代码保存在slink.c文件中。如图5.7所示，编译、运行slink。

```
# gcc slink.c -o slink
# ./slink
链表数据结点个数为：5
第1个结点的值：15
第2个结点的值：14
第3个结点的值：13
第4个结点的值：12
第5个结点的值：11
链表数据结点个数为：5
第1个结点的值：21
第2个结点的值：22
第3个结点的值：23
第4个结点的值：24
第5个结点的值：25
链表数据结点个数为：6
第1个结点的值：15
第2个结点的值：14
第3个结点的值：100
第4个结点的值：13
第5个结点的值：12
第6个结点的值：11
d2=22
链表数据结点个数为：4
第1个结点的值：21
第2个结点的值：23
第3个结点的值：24
第4个结点的值：25
d3=23
d1=100
链表数据结点个数为：6
第1个结点的值：11
第2个结点的值：12
第3个结点的值：13
第4个结点的值：100
第5个结点的值：14
第6个结点的值：15
单向循环链表数据结点个数为：4
21 23 24 25
21 23 24 25
```

图5.7 编译、运行slink

5.2 双向链表

扫一扫

5.2.1 双链表结构与链表结点类型

双向链表简称双链表，其特点是链表的链接方向是双向的。本节介绍的双链表包含头结点，因此链表中的结点分为两种：头结点和一般结点（也称数据结点）。这两种结点的类型一样，都是某种结构体类型，但是头结点的数据域一般不使用，一般结点的数据域存放具体的用户数据。对双链表的访问要从头结点开始顺序进行。一般结点的第一个结点称为首结点，最后一个结点称为尾结点。头结点由头指针指向。双链表结点有两个指针域，分别称为前驱指针和后继指针。前驱指针指向当前结点前面的结点，后继指针指向当前结点后面的结点。头结点的前驱指针为 NULL，尾结点的后继指针为 NULL。从双链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。既可以从头到尾遍历，又可以从尾到头遍历。带头结点的双链表结构示意图如图 5.8 所示。

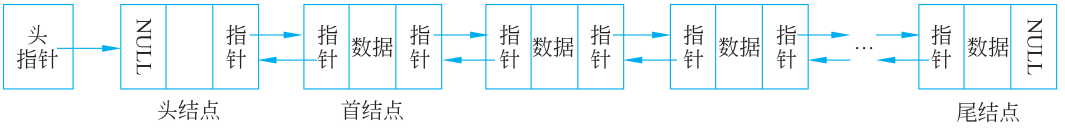

图 5.8 带头结点的双链表结构示意图

双链表结点的类型定义如下。

```
#define ElemType int
typedef struct dlnode{
   ElemType data;         //数据域
   struct dlnode *prior; //指针域，指向前驱结点
   struct dlnode *next;  //指针域，指向后继结点
}DLNode, *PDLNode;
```

数据域的数据类型为宏定义 ElemType，可根据实际需求定义 ElemType 的具体数据类型，比如定义为 int 型。两个指针域都使用了结构体的自引用，也就是在结构体内部包含了指向自身类型的指针成员。使用 typedef 关键字将 struct dlnode 类型定义为 DLNode，将 struct dlnode * 类型定义为 PDLNode。后续可以使用 DLNode 定义结构体变量，使用 PDLNode 定义指向 DLNode 结构体变量的指针。

5.2.2 创建双链表

可以采用头插法或尾插法创建双链表。头插法创建双链表的示意图如图 5.9 所示。尾插法将一个新结点链接到已有链表的尾部。尾插法创建双链表的示意图如图 5.10 所示。

下面的 DListBuildH 和 DListBuildR 函数，都是根据含有 n 个元素的数组 a 创建带头结点的双链表。

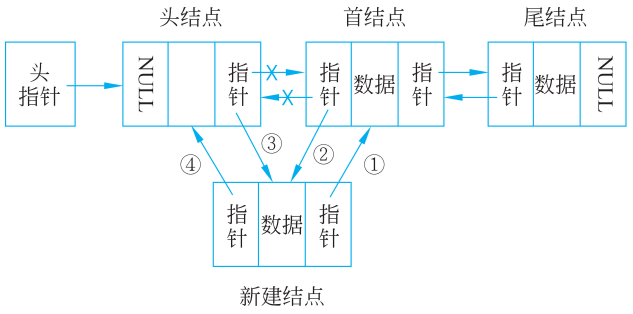

图 5.9　头插法创建双链表的示意图

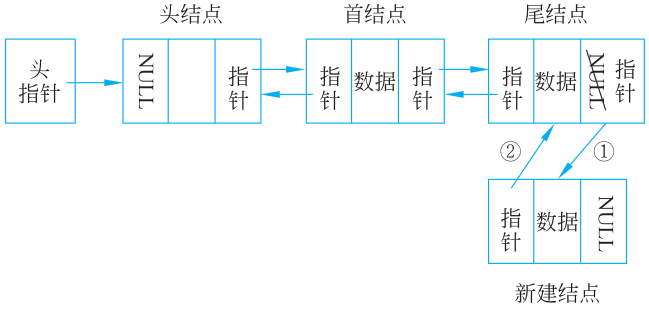

图 5.10　尾插法创建双链表的示意图

1. 头插法创建双链表的函数 DListBuildH

```
#include <stdio.h>
#include <stdbool.h>
#include <malloc.h>
typedef struct{ //数据域类型，可以根据实际需求添加更多的结构体成员
  int data;
}DATA;
typedef struct dlnode{
  DATA data;            //数据域
  struct dlnode *prior; //指针域，指向前驱结点
  struct dlnode *next;  //指针域，指向后继结点
}DLNode, *PDLNode;
PDLNode DListBuildH(DATA a[], int n){          //头插法创建双链表
  PDLNode L=(PDLNode)malloc(sizeof(DLNode)); L->next=NULL; //创建头结点
  for(int i=0; i<n; i++){                      //循环创建新结点
    PDLNode s=(PDLNode)malloc(sizeof(DLNode));  //为新结点分配存储空间
    s->data=a[i];               //将数组a中的元素值赋值给新结点s数据域
    s->next=L->next;            //新结点s的后继指针域指向首结点（或NULL）
    if(L->next!=NULL) L->next->prior=s; //若存在首结点，则让其前驱指针指向新结点
    L->next=s;                  //头结点的后继指针指向新结点s
    s->prior=L;                 //新结点的前驱指针指向头结点
  }
  return L;
}
```

2. 尾插法创建双链表的函数 DListBuildR

```
void DListBuildR(DLNode **L, DATA a[], int n){  //尾插法创建双链表
  PDLNode s, r=*L=(PDLNode)malloc(sizeof(DLNode)); //创建头结点
  for(int i=0; i<n; i++){                        //循环建立数据结点
    s=(PDLNode)malloc(sizeof(DLNode));           //为新结点分配存储空间
    s->data=a[i];                 //将数组a中的元素值赋值给新结点s数据域
    r->next=s;                    //尾结点的后继指针指向新结点s
    s->prior=r;                   //新结点s的前驱指针指向尾结点
    r=s;                          //结点s成为新的尾结点，尾指针r始终指向尾结点
  }
  r->next=NULL;                   //尾结点的后继指针域置为NULL
} //由于要在本函数内修改传入的头指针L的值，因此第一个参数L为二级指针
```

5.2.3 插入结点

除头插法和尾插法插入结点外，更灵活的插入结点的方法是在指定位置插入结点。插入结点的示意图如图 5.11 所示。在双链表中插入结点的函数如下。

```
bool DListInsert(PDLNode L, int i, DATA e){ //在双链表L第i个位置插入值为e的结点
  int j=0;
  while(j<i-1 && L!=NULL){      //在双链表中查找第i-1个结点，由L指向
    L=L->next;
    j++;
  }
  if(L==NULL) return false;     //未找到第i-1个结点，返回false
  else{                         //找到第i-1个结点，插入新建结点并返回true
    PDLNode s=(PDLNode)malloc(sizeof(DLNode));  //为新结点分配存储空间
    s->data=e;                  //新结点数据域被赋值为e
    s->next=L->next;            //新结点指针域指向L所指结点的下一个结点
    if(L->next!=NULL) L->next->prior=s; //若存在后继结点，则让其前驱指针指向新结点
    s->prior=L;                 //新结点s的前驱指针指向L所指向结点
    L->next=s;                  //L所指向结点的后继指针指向新结点
    return true;
  }
}
```

5.2.4 删除结点

删除双链表中的结点时，遍历链表找到要删除结点的前一个结点，然后将其后面结点从双链表中删除即可。删除数据结点的示意图如图 5.12 所示。在双链表中删除数据结点的函数如下。

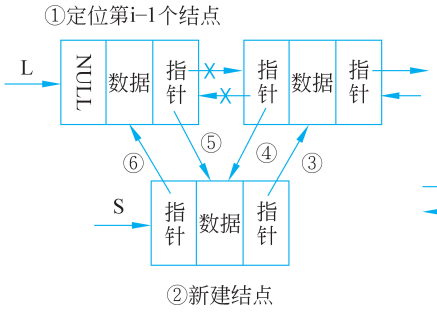

图 5.11 插入结点的示意图

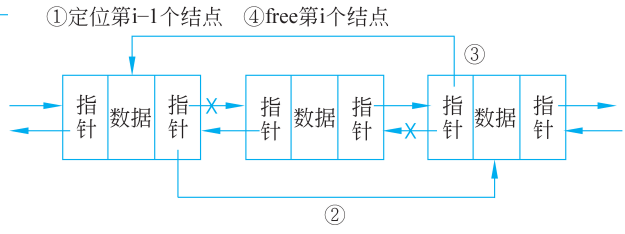

图 5.12 删除数据结点的示意图

```
bool DListDelete(PDLNode L, int i, DATA *pe){ //双链表L中若存在第i个结点，则将其删除，
  int j=0;                                    //并且将其数据域值赋给pe所指的变量
  while(j<i-1 && L!=NULL){      //在双链表中查找第i-1个结点，由L指向
    L=L->next;
    j++;
  }
  if(L==NULL) return false;     //未找到第i-1个结点，返回false
  else{                         //找到第i-1个结点L
    PDLNode s=L->next;          //s指向第i个结点
    if(s==NULL) return false;   //若不存在第i个结点，则返回false
    *pe=s->data;                //将要删除的结点的数据域值复制给pe所指的变量
    L->next=s->next;            //L所指向结点的指针域指向被删除结点s的下一个结点
    if(s->next!=NULL)           //若s结点存在后继结点，
      s->next->prior=L;         //则让后继结点的前驱指针指向L所指向结点
    free(s);                    //释放s结点，即成功删除第i个结点
    return true;                //返回true
  }
}
```

5.2.5 读取结点

在双链表中读取指定位置数据结点内容的函数如下。

```
bool DListGetElem(PDLNode L, int i, DATA *pe){  //读取双链表L中若存在第i个结点，
  int j=0;                                      //则将其数据域值赋给pe所指的变量
  while(j<i && L!=NULL){
    j++;
    L=L->next;
  }
  if(L==NULL) return false;     //未找到第i个结点，返回false
  else{                         //存在第i个结点，返回true
    *pe=L->data;                //将第i个结点的数据域值复制给pe所指的变量
    return true;
  }
}
```

5.2.6 查找结点

查找函数根据传入的 x 值，在双链表中进行遍历，查找 x 值所在的结点，然后返回该结点的地址。

```
bool DListEmpty(PDLNode L){     //判断双链表是否为空表，无数据结点为空表
  if(!L){printf("双链表无头结点\n"); return true;}
  return(L->next==L);           //若双链表L没有数据结点，则返回true，否则返回false
}
PDLNode DListFind(PDLNode L, DATA x){   //查找结点
  if(DListEmpty(L)){printf("双链表为空\n"); return NULL;}
  L=L->next;                            //L指向第一个数据结点
  while(L){                             //存在数据结点
    if(L->data.data==x.data) return L;  //找到，返回x结点指针
    else L=L->next;
  }
  return NULL;                          //没有找到，返回NULL
}
```

5.2.7 打印双链表

打印双链表函数从第一个数据结点开始依次将所有数据结点中的数据域值输出。