第 3 章　链　　表

3.1　单链表

单链表是最简单的链表结构，同时也是最基本的链表结构，理解了单链表及其相关概念，其他链表结构也就很容易理解了。

3.1.1　基本概念

一个链表由若干链表结点链接而成。在单链表中，每个链表结点包括两个域：用来存储数据的数据域和用来链接下一个链表结点的指针域。为了访问到链表中的结点，每个链表有一个相关联的指针，该指针指向链表中的第一个结点，这个指针称为头指针。

例 3-1　给定线性表 A＝{31，27，59，40，58}，采用链接存储，则对应的链表如图 3.1 所示。

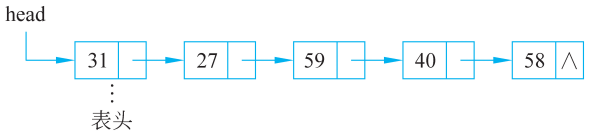

图 3.1　单链表

例 3-1 中，head 为头指针；用来存储数据 31 的结点是链表的第一个结点，称为表头；用来存储数据 58 的结点是链表的最后一个结点，称为表尾，其指针域为空(NULL)。

链表中通过头指针 head 可以访问到表头结点，通过前一个结点的指针可以访问到后一个结点，当前结点指针为空时，说明已经到达表尾结点。与顺序存储相比，无论是从链表中删除一个元素还是往链表中插入一个元素都要方便得多，不涉及任何结点的移动。图 3.2 和图 3.3 分别给出了从链表中删除一个结点以及往链表中插入一个结点的处理过程。

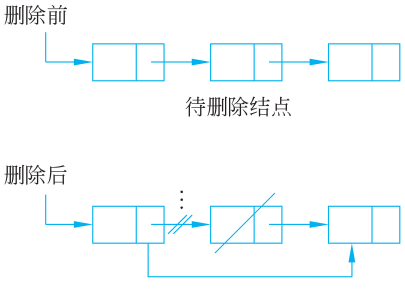

图 3.2　从链表中删除一个结点

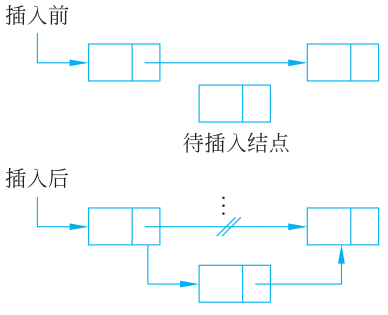

图 3.3　往链表中插入一个结点

不含结点的链表称为空链表，此时，head 为空（NULL）。通过判断 head 是否等于 NULL，可以确定一个链表是否为空链表。当在链表中插入一个结点时，首先要判断链表是否为空链表，如果是空链表，则修改 head 使之直接指向要插入的结点即可；否则，由 head 指向的表头开始寻找到插入位置，找到后，将要插入的结点插入链表中。

为了插入方便，实际应用中可以采用附加头结点的方式组织链表，此时，头指针所指向的结点不是表头，而是一个特殊的结点，这个结点不用来存储任何数据。因此，当链表为空链表时，头指针不是空指针，于是，在对链表进行操作时，无须判断头指针是否为空。这个附加的结点称为附加头结点，附加头结点的指针所指向的结点才是表头结点。请注意附加头结点与表头结点两者之间的差别。

例 3-2 对于例 3-1 中给出的一组数据，采用附加头结点的链表进行存储时，对应的链表如图 3.4 所示。

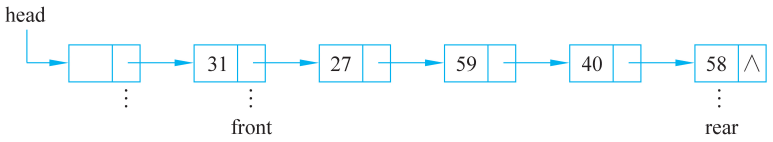

图 3.4 附加头结点的单链表

在本章后续部分的讨论中，将根据实际需要不加说明地选择使用上述两种单链表之一，请读者注意附加头结点的单链表的优点。

3.1.2 单链表结点结构

单链表结点结构的定义中包含了数据和指针这两个基本要素，关于单链表结点的常用操作包括一个对结点进行初始化处理的方法、一个在结点后插入元素的方法以及一个删除当前结点的后继结点的方法。对于每个结点，主要的操作都是与它的后继结点直接相关。单链表结点及操作方法的描述在头文件 node.h 中，实现细节在文件 node.c 中。

node.h 中的内容：

```
#ifndef _NODE_H
#define _NODE_H
#include <stdio.h>
#include <stdlib.h>

struct node
{
    struct node * next;                          /* next 为指向下一个结点的指针 */
    ElementType data;
};
typedef struct node Node;

void InitNode(Node *, ElementType item, Node *ptr);      /* 初始化单链表结点 */
void InsertAfter(Node *, Node *);                        /* 插入一个单链表结点 */
Node * DeleteAfter(Node *);                              /* 删除一个单链表结点 */
```

```
Node * GetNode(ElementType item, Node * ptr);          /* 创建一个单链表结点 */
void  FreeNode(Node * );                               /* 释放一个单链表结点 */

#endif
```

node.c 中的内容：

算法 3.1 单链表结点及操作。

```
#include "node.h"

void InitNode(Node * N, ElementType item, Node * ptr)
{
    N->data=item;
    N->next=ptr;
}

void InsertAfter(Node * N, Node * p)
{
    p->next=N->next;          /* 将当前结点的后继结点连接到结点 p 之后 */
    N->next=p;                /* 将结点 p 作为当前结点的后继结点 */
}

Node * DeleteAfter(Node * P)
{
    Node * ptr=P->next;       /* 保存当前结点的后继结点 */

    if (ptr==NULL)
        return NULL;          /* 若没有后继结点,则返回空指针 */
    P->next=ptr->next;        /* 当前结点指向其原来的后继结点,即 ptr 的后继 */
    return ptr;               /* 返回指向被删除结点的指针 */
}

Node * GetNode(ElementType item, Node * nextPtr)
{
    Node * newNode;

    /* 申请单链表结点空间 */
    newNode=(Node * ) malloc(sizeof(Node));
    if (newNode==NULL)
    {
        printf("Memory allocation failure!\n");
        exit(1);
    }
    InitNode(newNode, item, nextPtr);
    return newNode;
}

void FreeNode(Node * ptr)
{
    if (!ptr)                 /* 若 ptr 为空,则给出相应提示并返回 */
    {
```

```
        printf("FreeNode:invalid node pointer!\n");
        return;
    }
    free(ptr);          /*释放结点占用的内存空间*/
}
```

请读者注意，在插入结点和删除结点这两个方法的实现过程中，都多次涉及修改指针的操作，这些操作是不能够随意调换顺序的，建议写链表的算法前先画出相应的链表结构图。

3.1.3 单链表结构

利用单链表结点结构可以构建单链表结构。显然，单链表结构中必须包含基本的数据成员表头指针 front 和表尾指针 rear。此外，为了处理方便，对单链表完整的描述还包括单链表结点个数 size、当前结点位置 position、当前结点指针 currPtr、指向当前结点前驱的指针 prevPtr。单链表是动态存储数据结构，因此，对单链表的操作应该包含申请和释放单链表中结点所需空间的方法，以及有关插入、删除、访问、修改、移动当前结点指针、获取单链表信息等方法。

下面的文件 linkedlist.h 中给出了单链表结构的类型说明及相应的操作方法，其实现细节在文件 linkedlist.c 中。

文件 linkedlist.h 中的内容：

```
#ifndef _LINKEDLIST_H
#define _LINKEDLIST_H
enum boolean {FALSE, TRUE};
typedef enum boolean Bool;

struct linkedList
{
    Node *front, *rear;                 /*指向表头和表尾的指针*/
    Node *prevPtr, *currPtr;            /*用于访问数据、插入和删除结点的指针*/
    int size;                           /*表中的结点数*/
    int position;                       /*表中当前结点位置计数*/
};
typedef struct linkedList LinkedList;

Node *GetNode(ElementType item, Node *ptr);    /*申请结点空间的函数*/
void FreeNode(Node *p);                        /*释放结点空间的函数*/
void InitLinkedList(LinkedList *);             /*初始化函数*/
Bool IsEmpty(LinkedList *);                    /*判断表是否为空*/
int NextLNode(LinkedList *);                   /*求当前结点后继的函数*/
int SetPosition(LinkedList *, int pos);        /*重定位当前结点*/
void InsertAt(LinkedList *, ElementType item);    /*在当前结点处插入结点的函数*/
void InsertLAfter(LinkedList *, ElementType item); /*在当前结点后插入结点的函数*/
void DeleteAt(LinkedList *);                   /*删除当前结点的函数*/
void DeleteLAfter(LinkedList *);               /*删除当前结点后继的函数*/
```

```
ElementType GetData(LinkedList *);                    /*修改和访问数据的函数*/
void SetData(LinkedList *, ElementType item);
void Clear(LinkedList *);                             /*清空链表的函数*/

#endif
```

文件 linkedlist.c 中的内容：

算法 3.2 单链表中的运算。

```
#include "linkedlist.h"
/*单链表初始化函数(建立一个空链表)*/
void InitLinkedList(LinkedList *L)
{
    L->front=NULL;
    L->rear=NULL;
    L->prevPtr=NULL;
    L->currPtr=NULL;
    L->size=0;
    L->position=-1;
}

/*判断表是否为空的函数*/
Bool IsEmpty(LinkedList *L)
{
    return L->size?FALSE:TRUE;
}

/*将后继结点设置为当前结点的函数*/
int NextLNode(LinkedList *L)
{
    /*若当前结点存在,则将其后继结点设置为当前结点*/
    if (L->position >=0 && L->position<L->size)
    {
        L->position++;
        L->prevPtr=L->currPtr;
        L->currPtr=(L->currPtr)->next;
    }
    else L->position=L->size;                         /*否则将当前位置设为表尾后*/
    return L->position;                               /*返回新位置*/
}

/*重置链表当前结点位置的函数*/
int SetPosition(LinkedList *L, int pos)
{
    int k;
    if (!L->size) return-1;                           /*若链表为空,则返回*/
    if(pos<0||pos >L->size-1)                         /*若位置越界,则返回*/
    {
        printf("position error");
        return-1;
    }
```

```
    L->currPtr=L->front;                          /*寻找对应结点*/
    L->prevPtr=NULL;
    L->position=0;

    for (k=0; k<pos; k++)
    {
        L->position++;
        L->prevPtr=L->currPtr;
        L->currPtr=(L->currPtr)->next;
    }
    return L->position;                           /*返回当前结点位置*/
}

/*链表中在当前结点处插入新结点的函数*/
void InsertAt(LinkedList *L, ElementType item)
{
    Node *newNode;
    if (!L->size)
    {
        newNode=GetNode(item, L->front);          /*在空表中插入*/
        L->front=L->rear=newNode;
        L->position=0;
    }
    else if (!L->prevPtr)
    {
        newNode=GetNode(item, L->front);          /*在表头结点处插入*/
        L->front=newNode;
    }
    else
    {
        newNode=GetNode(item, L->currPtr);        /*在链表的中间位置插入*/
        InsertAfter(L->prevPtr,newNode);
    }

    L->size++;                                    /*增加链表的大小*/
    L->currPtr=newNode;                           /*新插入的结点为当前结点*/
}

/*链表中在当前结点后插入新结点的函数*/
void InsertLAfter(LinkedList *L, ElementType item)
{
    Node *newNode;
    if (!L->size)
    {
        newNode=GetNode(item, NULL);              /*在空表中插入*/
        L->front=L->rear=newNode;
        L->position=0;
    }
    else if (L->currPtr==L->rear||!L->currPtr)
    {
        newNode=GetNode(item, NULL);              /*在表尾结点后插入*/
        InsertAfter(L->rear, newNode);
```

```
            L->prevPtr=L->rear;
            L->rear=newNode;
            L->position=L->size;
        }
        else
        {
            newNode=GetNode(item, (L->currPtr)->next);  /*在链表的中间位置插入*/
            InsertAfter(L->currPtr, newNode);
            L->prevPtr=L->currPtr;
            L->position++;
        }
        L->size++;                                      /*增加链表的大小*/
        L->currPtr=newNode;                             /*新插入的结点为当前结点*/
    }
```

注意上述算法中调用的 InsertAfter 函数都可改为指针运算的调用，如 InsertAfter(L→rear，newNode)可改为 L→rear→next＝newNode。

```
    /*链表中删除当前结点的函数*/
    void DeleteAt(LinkedList *L)
    {
        Node *oldNode;
        if (!L->currPtr)          /*若表为空或已到表尾之后,则给出错误提示并返回*/
        {
            printf("DeleteAt: current position is invalid!\n");
            return;
        }
        if (!L->prevPtr)
        {
            oldNode=L->front;                 /*删除的是表头结点*/
            L->front=(L->currPtr)->next;
        }
        else oldNode=DeleteAfter(L->prevPtr); /*删除的是表中结点*/
        if (oldNode==L->rear)
        {
            L->rear=L->prevPtr;               /*删除的是表尾结点,则修改表尾指针*/
        }
        L->currPtr=oldNode->next;             /*后继结点作为新的当前结点*/
        FreeNode(oldNode);                    /*释放原当前结点*/
        L->size--;                            /*链表大小减1*/
    }

    /*链表中删除当前结点后继的函数*/
    void DeleteLAfter(LinkedList *L)
    {
        Node *oldNode;
        if (!L->currPtr||L->currPtr==L->rear)
                                  /*若表为空或已到表尾,则给出错误提示并返回*/
        {
            printf("DeleteAfter:  current position is invalid!\n");
            return;
```

```
    }
    oldNode=DeleteAfter(L->currPtr);
                                    /*保存被删除结点的指针并从链表中删除该结点*/
    if (oldNode==L->rear)
        L->rear=L->currPtr;     /*删除的是表尾结点*/
    FreeNode(oldNode);          /*释放被删除结点*/
    L->size--;                  /*链表大小减 1*/
}

/*链表中获取当前结点数据的函数*/
ElementType GetData(LinkedList *L)
{
    if (!L->size||!L->currPtr) /*若表为空或已经到达表尾之后,则出错*/
    {
        printf("Data:  current node does not exist!\n");  /*给出出错信息并退出*/
        exit(1);
    }
    return L->currPtr->data;
}

/*链表中修改当前结点数据的函数*/
void SetData(LinkedList *L, ElementType item)
{
    if (!L->size||!L->currPtr)          /*若表为空或已经到达表尾之后,则出错*/
    {
        printf("Data:  current node does not exist!\n");
        exit(1);
    }
    L->currPtr->data=item;              /*修改当前结点的值*/
}

/*链表中清空链表的函数*/
void Clear(LinkedList *L)
{
    Node *currNode=L->front, *nextNode;
    while (currNode)
    {
        nextNode=currNode->next;        /*保存后继结点指针*/
        FreeNode(currNode);             /*释放当前结点*/
        currNode=nextNode;              /*原后继结点成为当前结点*/
    }
    L->front=L->rear=L->prevPtr=L->currPtr=NULL;
    L->size=0;
    L->position=-1;                     /*修改空链表数据*/
}
```

请注意，单链表结构中究竟包括哪些具体数据以及哪些具体方法并不是确定不变的，读者可以根据实际需要定义自己的单链表结构和相应的操作方法。例如，可以定义一个合并两个链表的方法，也可以定义一个具有附加表头结点的单链表结构。

例 3-3　对于例 3-1 中给出的一组数据，采用单链表进行存储，其对应的存储结构如图 3.5 所示。

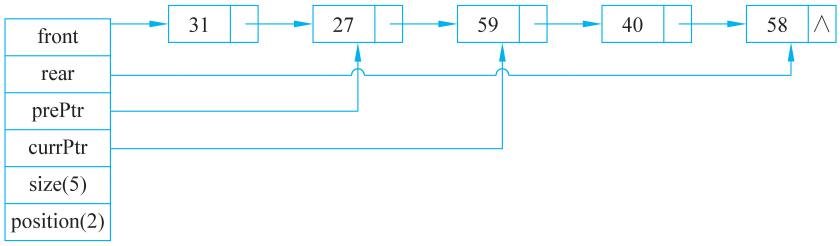

图 3.5 一个实际的单链表结构

单链表应用举例如下。

例 3-4 创建一个长度为 10 的单链表，该单链表的表目为随机产生的整型数，试编写算法实现由键盘输入的整数值，由此生成长度为 10 的单链表并检索该单链表中有相同整数值的表目个数。

解决这个问题的关键是建立单链表和在单链表中移动以达到检索单链表的目的。下面用两种方法建立算法。方法一是通过单链表结点结构建立 Nodelib 库，调用 Nodelib 库中的函数实现单链表的建立和访问。方法二是通过已建好的单链表结构中的方法实现单链表的建立和访问。

方法一：利用单链表结点结构构建的 Nodelib 库，该库中实现了单链表的若干运算，可利用库中部分运算解决例 3-4 中的问题。Nodelib 库中的内容：

```
#ifndef NODE_LIBRARY
#define NODE_LIBRARY

enum appendNewline {noNewline,addNewline};
typedef enum appendNewline AppendNewline;

/*打印链表*/
void PrintList(Node *head, AppendNewline addnl)
{
    Node *currPtr=head;
    /*打印当前结点数据直至表尾*/
    while(currPtr !=NULL)
    {
        if(addnl==addNewline)
            printf("%d\n", currPtr->data);
        else
            printf("%d ", currPtr->data);
        /*指针调整指向其后继结点*/
        currPtr=currPtr->next;
    }
}

/*单链表中基于表目的查找*/
int Find(Node *head, ElementType item)
{
    Node *currPtr=head, *prevPtr=NULL;
    while(currPtr !=NULL)
```

```
    {
        if (currPtr->data==item)          /* 比较链表结点的数据域 */
        {
            return 1;
        }
        prevPtr=currPtr;
        currPtr=currPtr->next;
    }
    return 0;                             /* 查找失败 */
}

/* 在单链表的头部插入一个新的单链表结点 */
void InsertFront(Node **head, ElementType item)
{
    /* 申请新插入结点的空间 */
    Node *newNode=(Node *) malloc (sizeof(Node));
    newNode->data=item;
    newNode->next=*head;
    *head=newNode;
}

/* 删除单链表的头结点 */
void DeleteAt(Node **head)
{
    Node *p=*head;
    if (*head !=NULL)
    {
        *head=*head->next;
        free(p);
    }
    else
    printf("delete failure");

}

/* 删除单链表中值为 key 的结点 */
void Delete (Node **head, ElementType key)
{
    Node *currPtr=*head, *prevPtr=NULL;

    if (currPtr==NULL) return;
    while (currPtr !=NULL && currPtr->data !=key)
    {
        prevPtr=currPtr;
        currPtr=currPtr->next;
    }

    if (currPtr !=NULL)
    {
        if(prevPtr==NULL)
            *head=(*head)->next;
        else
```

```
            DeleteAfter(prevPtr);
        free(currPtr);
    }
}

/*在有序单链表中插入新结点*/
void InsertOrder(Node **head, ElementType item)
{
    Node *currPtr, *prevPtr, *newNode;

    prevPtr=NULL;
    currPtr=*head;
    while (currPtr !=NULL)
    {
       if (item<currPtr->data) break;
         prevPtr=currPtr;
         currPtr=currPtr->next;
    }
    if (prevPtr==NULL)
         InsertFront(head,item);
    else
    {
         newNode=GetNode(item, NULL);
         InsertAfter(prevPtr, newNode);
    }
}

/*删除整个单链表*/
void ClearList(Node **head)
{
    Node *currPtr, *nextPtr;
    currPtr=*head;
    while(currPtr !=NULL)
    {
         nextPtr=currPtr->next;
         free(currPtr);
         currPtr=nextPtr;
    }
   head=NULL;
}

#endif                                  /*NODE_LIBRARY*/
```

算法 3.3 用 nodelib 中的函数实现单链表的建立和查找。

```
typedef int ElementType;

#include <time.h>
#include "node.h"
#include "nodelib.h"

void main(void)
```

```
{
    Node *head=NULL, *currPtr;          /*初始时链表头指针 head 为 NULL*/
    int i, key, count=0, k;

     srand(time(NULL));
    for (i=0; i<10; i++)                /*依次从链表头部随机插入 10 个整数表目*/
    {
        k=1+rand()%10;
        InsertFront(&head, k);
    }

    printf("List:");                    /*显示链表*/
    PrintList(head,noNewline);
    printf("\n");

    printf("Enter a key:");             /*由键盘输入一个整数*/
    scanf("%d", &key);

    currPtr=head;                       /*遍历整个链表*/
    while (currPtr !=NULL)
    {
                                        /*统计链表中表目值等于 key 的结点个数*/
        if (currPtr->data==key)
            count++;
        currPtr=currPtr->next;
    }
    printf("The data value %d occurs %d times in the list\n", key, count);
}
```

上述算法的一次运行结果如下：

```
List:5 6 1 3 1 4 5 7 8 2
Enter a key:5
The data value 5 occurs 2 times in the list
```

方法二：用基于单链表结构的操作方法实现。与前面的基于单链表结点的操作方法不同，由于基于单链表的操作方法实现了对整个单链表的操作，利用这些方法可方便实现单链表的建立，单链表中基于表目元素值的查找、打印等。

算法 3.4 基于单链表结构的操作方法实现单链表的建立和查找。

```
typedef int ElementType;

#include "node.h"
#include "linkedlist.h"
#include <time.h>

void main(void)
{
    LinkedList *L=(LinkedList *) malloc (sizeof(LinkedList));     /*声明链表*/
    int i, key, count=0, k;

    srand(time(NULL));
```

```
    InitLinkedList(L);
    for (i=0; i<10; i++)                    /*随机插入10个表目为整数的链表结点*/
        InsertLAfter(L, 1+rand()%10);

    printf("List:");                        /*显示链表*/
    k=0;
    while(k<L->size))
    {
        SetPosition(L, k++);
        printf("%d ", GetData(L));
    }
    printf("\n");

    printf("Enter a key:");                 /*由键盘输入一个整数*/
    scanf("%d", &key);
    k=0;                                    /*遍历整个链表*/
    while (k<L->size)
    {
        SetPosition(L, k++);                /*统计链表中表目值等于key的结点个数*/
        if (GetData(L)==key)
        count++;
    }
    printf("The data value %d occurs %d times in the list\n", key, count);
}
```

上述算法的一次运行结果如下：

```
List:7 1 7 9 9 6 4 1 7 5
Enter a key: 5
The data value 5 occurs 1 times in the list
```

3.1.4 栈的单链表实现

用单链表结构存储的栈又称为链栈，可以用前面定义的单链表结构描述栈结构，在此存储模式下实现栈的运算，假设用单链表的表头代表栈顶，有关链栈结构声明及操作描述在文件 linkedstack.h 中。

文件 linkedstack.h 中的内容为：

```
struct linkedStack
{
    LinkedList *stack;                              /*存放栈元素的链表*/
};
typedef struct linkedStack LinkedStack;

void InitLinkedStack(LinkedStack *);                /*初始化函数*/
void Push(LinkedStack *, ElementType item);         /*操作栈的方法*/
ElementType Pop(LinkedStack *);                     /*弹栈*/
ElementType Top(LinkedStack *);                     /*取栈顶的表目*/
```

链栈的操作方法的实现细节在文件 linkedstack.c 中，因为栈的插入、删除运算始终在

栈的一端进行，因此，规定非空链栈的当前结点始终在 position＝0 的位置。

linkedstack.c 中的内容见算法 3.5。

算法 3.5 链栈运算。

```
#include "node.h"
#include "linkedlist.h"
#include "linkedstack.h"

/*链栈的初始化函数*/
void InitLinkedStack(LinkedStack *LS)
{
    LS->stack=(LinkedList *) malloc (sizeof(LinkedList));
    InitLinkedList(LS->stack);
}

/*链栈的入栈函数*/
void Push(LinkedStack *LS, ElementType item)
{
    InsertAt(LS->stack, item);
}

/*链栈的弹栈函数*/
ElementType Pop(LinkedStack *LS)
{
    ElementType tmpData;
    if (!LS->stack->size)
    {                                          /*栈空,给出出错信息并退出*/
        printf("Pop:  underflowed!\n");
        exit(1);
    }
    tmpData=GetData(LS->stack);                /*删除栈顶元素*/
    DeleteAt(LS->stack);
    return tmpData;                            /*返回栈顶数据*/
}

/*链栈的读栈顶函数*/
ElementType Top(LinkedStack *LS)
{
    if (!LS->stack->size)
    {                                          /*栈空,给出出错信息并退出*/
        printf("Top:  underflowed!\n");
        exit(1);
    }
    return GetData(LS->stack);                 /*返回栈顶数据*/
}
```

3.1.5 队列的单链表实现

用单链表结构存储的队列又称为链队列，与链栈类似，可以用前面定义的单链表结构

描述队列结构,在此存储方式上实现的队列的运算,有关链队列结构声明及操作描述在文件 linkedqueue.h 中,实现细节在 linkedqueue.c 中。

文件 linkedqueue.h 中的内容:

```
struct linkedQueue
{
    LinkedList * queue;
};
typedef struct linkedQueue LinkedQueue;

void InitLinkedQueue(LinkedQueue * );          /* 初始化一个链队列 */
void In(LinkedQueue * , ElementType item);     /* 在队列中插入一个新数据元素 */
ElementType Out(LinkedQueue * );               /* 在队列中删除一个数据元素 */
ElementType Front(LinkedQueue * );             /* 取队列的头部元素 */
void ClearLQ(LinkedQueue * );                  /* 清除队列中的数据元素 */
Bool IsEmptyLQ(LinkedQueue * );                /* 判队列是否为空 */
```

文件 linkedqueue.c 中的内容见算法 3.6。

算法 3.6 链队列运算。

```
#include "node.h"
#include "linkedlist.h"
#include "linkedqueue.h"

/* 初始化一个链队列 */
void InitLinkedQueue(LinkedQueue * LQ)
{
    LQ->queue=(LinkedList * ) malloc (sizeof(LinkedList));
    InitLinkedList(LQ->queue);
}

/* 在队列中插入一个新数据元素 */
void In(LinkedQueue * LQ, ElementType item)
{
    SetPosition(LQ->queue, (LQ->queue)->size-1);   /* 设置队尾元素为当前结点 */
    InsertLAfter(LQ->queue, item);                 /* 在队尾结点后插入新结点 */
}

/* 在队列中删除一个数据元素 */
ElementType Out(LinkedQueue * LQ)
{
    ElementType  tmpData;

    if (!(LQ->queue)->size)                        /* 若队列为空,提示出错信息 */
    {
        printf("Out:  underflowed!\n");
        exit(1);
    }

    SetPosition(LQ->queue, 0);                     /* 当前结点位置移到队列头 */
    tmpData=GetData(LQ->queue);                    /* 保存队列头结点的数据 */
```

```
    DeleteAt(LQ->queue);                    /*删除队列头结点*/

    return tmpData;                         /*返回被删除的队列头数据*/
}
```

其他队列操作方法：取队列的头部元素，清除队列中的数据元素，求队列中的元素个数，判断队列是否为空的实现可借鉴单链表结构的实现方法类似实现，这些代码请读者自行完成。

上述队列的实现是借助单链表结构实现的。其优点是能方便调用单链表结构中的函数，模块化的程序设计思想能够很好地体现；缺点是因为反复调用 SetPosition 函数，链队列的插入、删除运算不能保证 $O(1)$。下面给出了一种直接基于单链表结点结构的链队列实现方法，能保证链队列的插入、删除运算的时间开销为 $O(1)$。

头文件 clnkqueue.h（循环链式队列，只用了尾结点指针）：

```
typedef int T;
struct LNode {
    T data;
    LNode* next;
};
typedef LNode CLnkQueue;
void CLQ_Free(CLnkQueue* q);
void CLQ_MakeEmpty(CLnkQueue** q);
BOOL CLQ_IsEmpty(CLnkQueue* q);
int CLQ_Length(CLnkQueue* q);
void CLQ_In(CLnkQueue** q, T x);
T CLQ_Out(CLnkQueue** q);
T CLQ_Head(CLnkQueue* q);
void CLQ_Print(CLnkQueue* q);
```

实现文件 clnkqueue.c 如下：

```
#include <stdio.h>
#include <stdlib.h>
#include "clnkqueue.h"

void CLQ_Free(CLnkQueue* q)                 /*q指向尾结点*/
{
    CLQ_MakeEmpty(q);
}

void CLQ_MakeEmpty(CLnkQueue** q)           /**q指向尾结点*/
{
    if (*q==NULL) return;
    LNode* node1=*q;
    LNode* node=*node1->next;
    free(node1);
    while (node!=*q) {
        LNode* next=node->next;
        free(node);
```

```
        node=next;
    }
    *q=NULL;
}

BOOL CLQ_IsEmpty(CLnkQueue* q)                    /*q指向尾结点*/
{
    return q==NULL;
}

int CLQ_Length(CLnkQueue* q)                      /*q指向尾结点*/
/*求队列长度*/
{
    if (q==NULL) return 0;
    int count=1;
    LNode* node=q->next;
    while(node!=q) {
        count++;
        node=node->next;
    }
    return count;
}

void CLQ_In(CLnkQueue** q, T x)                   /**q指向尾结点*/
/*入队列,新结点加入链表尾部*/
{
    LNode* node=(LNode*)malloc(sizeof(LNode));
    node->data=x;
    if(*q==NULL) {
        node->next=node;
    }
    else {
        node->next=*q->next;
        *q->next=node;
    }
    *q=node;
}

T CLQ_Out(CLnkQueue** q)                          /**q指向尾结点*/
/*出队列函数*/
{
    if (*q) {
        LNode* head=*q->next;
        T x=head->data;
        if (head==*q) *q=NULL;
        else *q->next=head->next;
        free(head);
        return x;
    }
    else {
        printf("CLQ_Out(): queue is empty\n");
```

```
            exit(1);
        }
    }

    T CLQ_Head(CLnkQueue * q)                    /*q指向尾结点*/
    /*获取队列头的函数*/
    {
        if (q)
            return q->next->data;
        else {
            printf("CLQ_Head(): queue is empty\n");
            exit(1);
        }
    }
    void CLQ_Print(CLnkQueue * q)                /*q指向尾结点*/
    /*打印队列的函数*/
    {
        if (q==NULL) {
            printf("The queue is empty. \n");
            return;
        }
        printf("The queue is: ");
        LNode * node=q;
        do {
            node=node->next;
            printf("%d  ", node->data);
        }while (node!=q);
        printf("\n");
    }
```

3.1.6 单链表的应用举例

作为单链表应用实例之一，下面讨论模拟打印缓冲池的实现。模拟打印缓冲池的工作原理如下：有一个待打印队列用来存储用户的打印作业，每个打印作业包括文件名、总打印页数、已打印页数3项信息；打印缓冲池接受用户的打印请求并将待打印的文件插入队列中，当打印机可用时，打印缓冲池从队列中读取打印作业并提交给打印机进行打印，一个请求中的全部打印页打印完毕，则将这个打印作业从队列中删除。为了方便编程实现，已打印页数的修改是这样进行的：已知打印机每分钟可打印的页数，系统每隔一个固定的时间段计算一次此时间段内已打印的页数，以此更新当前打印作业已经打印的页数。

打印请求信息可以定义成一个结构PrintJob，结构中含有待打印作业的文件名、总打印页数和已打印页数。

假设打印机以高达每分钟50页的速度连续打印。在打印机进行打印的同时，用户可以和系统进行交互，列出打印队列中的每个作业、加入新的打印作业、查看打印队列大小等。打印缓冲池每隔10秒以上计算一次打印的页数，为了计算时间段的时长，需要记录上次计算打印页数时的时刻。当计算出本次时间段内打印的页数后，对打印队列进行一

次更新，删除已打印完的作业，或更新当前打印作业的已打印页数。

通过以上分析，可以确定打印缓冲池应包含的数据成员和打印缓冲池的驱动方法。有关模拟打印缓冲池的结构定义与实现方法描述在头文件 spooler.h 中，相应的实现部分在文件 spooler.c 中。

文件 spooler.h 中的内容：

```
#define PRINTSPEED        50          /* 每分钟打印页数 */
#define DELTATIME         10          /* 更新打印队列的最小时间间隔为 10 秒 */

/* 打印作业结构声明 */
struct PrintJob
{
    char filename[31];                /* 待打印作业的文件名,最长不超过 30 个字符 */
    int totalpages;                   /* 总打印页数 */
    int printedpages;                 /* 已打印页数 */
};

/* 模拟打印缓冲池的结构声明 */
struct spooler
{
    LinkedList *jobList;              /* 存放打印作业及状态的队列 */
     time_t lasttime;                 /* 打印作业的状态时间 */
};
typedef struct spooler Spooler;

/* 模拟打印缓冲池的驱动方法 */
void UpdateSpooler(Spooler *, int timedelay);   /* 更新打印缓冲池 */
void InitSpooler(Spooler *);                    /* 初始化打印缓冲池 */
void AddJob(Spooler *, ElementType *job);       /* 加入打印作业 */
void ListJob(Spooler *);                        /* 显示打印缓冲池的打印作业 */
int NumberOfJobs(Spooler *);                    /* 求打印缓冲池的作业数 */
```

与 spooler.h 对应的 spooler.c 文件中的主要内容见算法 3.7。

算法 3.7 打印缓冲池。

```
#define WAITTIME 60

void InitSpooler(Spooler *SP)                          /* 初始化打印缓冲池 */
{
    SP->jobList=(LinkedList *) malloc (sizeof(LinkedList));
    InitLinkedList(SP->jobList);
    time(&(SP->lasttime));
}

/* 设定每页所需打印时间, 删除打印完的作业并修改正在打印作业的未打印页数 */
void UpdateSpooler(Spooler *SP, int timedelay)
{
    struct PrintJob job;

    int printedpages,remainpages;
```

```
    printedpages=timedelay * PRINTSPEED/60;        /* 计算时间段内打印的页数 */
    SetPosition(SP->jobList, 0);                    /* 定位到队列头 */
    while (!IsEmpty(SP->jobList) && printedpages >0) /* 更新打印队列 */
     {
        job=GetData(SP->jobList);          /* 取打印队列中的当前被打印作业 */
        /* 打印机可打印当前打印作业的页数 */
        jobprintpages=min(job.totalpages-job.printedpages, printedpages);
        printedpages-=jobprintpages;    /* 打印机打印完当前作业后还能打印的页数 */
        if((job.printedpages+=jobprintpages) >=job.totalpages)
            DeleteAt(SP->jobList);         /* 当前作业打印完毕,删除 */
        else
            SetData(SP->jobList, job);     /* 当前作业未打印完,更新 */
    }
}

/* 修改缓冲池加入新打印作业；计算缓冲池下次修改事件的随机时间 */
void AddJob(Spooler * SP, ElementType * job)
{
    if(IsEmpty(SP->jobList))           /* 若打印队列为空,则重置时间段开始时刻 */
        time(&(SP->lasttime));
    job->printedpages=0;               /* 置已打印页数为 0 */
    SetPosition(SP->jobList, (SP->jobList)->size-1);   /* 定位到打印队尾 */
    InsertLAfter(SP->jobList, * job);                  /* 将当前作业插入队尾 */
}

void ListJob(Spooler * SP)             /* 显示打印缓冲池的打印作业 */
{
    struct PrintJob job;
    time_t currtime;
    int k;

    /* 判断更新打印队列的时间段是否已到 */
    if(difftime(time(&currtime), SP->lasttime)>=DELTATIME)
    {
        k=(int)difftime(currtime, SP->lasttime);
        UpdateSpooler(SP, k);                  /* 更新打印队列 */
        SP->lasttime=currtime;                 /* 重置时间段开始时刻 */
    }

    if(IsEmpty(SP->jobList))
        printf("Print queue is empty\n");
    else
    {
        printf("Current print job are:\n");      /* 输出每个作业的相关信息 */
        for(SetPosition(SP->jobList,0); (SP->jobList)->position<=(SP->
        jobList)->size-1;NextLNode(SP->jobList))
        {
            job=GetData(SP->jobList);
            printf("Job=%s\t ", job.filename);
            printf("TotalPages=%d\t ", job.totalpages);
            printf("PrintedPages=%d\n", job.printedpages);
```