循环链表

循环链表是一种特殊的单向链表，最后一个结点的后继指向是头结点。两者的不同之处是结束的标志不同：单向链表的结束标志是最后一个结点指向NULL，循环链表的结束标志是最后一个结点指向头结点。

可以令头指针指向最后一个结点，或新增加一个尾指针一直指向最后一个结点。可以使得查找链表的首元结点和最后一个结点都很方便。

```
//首先定义链表的结点。
typedef struct LLN{
    int Item;
    struct LLN *pNext;
}Linklist;
```

头结点初始化：

```
void Init_LinkList(Linklist *Head){
    Head=(Linklist *)malloc(sizeof(Linklist));
    Head->Item=0;
    Head->pNext=Head;
}
```

链表的创建

1.头插法：

```
void CreateLinkList(Linklist *Head,int *Array,int DataNum){
    int i;
    for(i=0;i<DataNum;i++){
        Linklist *CurrentNode=(Linklist *)malloc(sizeof(Linklist));
        CurrentNode->Item=Array[i];
        CurrentNode->pNext=Head->pNext;
        Head->pNext=CurrentNode;
    }
}
```

每次循环都创建一个新的结点，新的结点的后继指向Head->pNext是上一个循环结束时Head指向的结点，也就是上一个循环创建的结点。循环结束后，每个结点按创建顺序逆序排列。

2.尾插法：

```
void CreateLinkList(Linklist *Head,int *Array,int DataNum){
    int i;
    Linklist *pHead=Head;
    for(i=0;i<DataNum;i++){
        Linklist *CurrentNode=(Linklist *)malloc(sizeof(Linklist));
        CurrentNode->Item=Array[i];
        pHead->pNext=CurrentNode;
        pHead=CurrentNode;
    }
    pHead->pNext=Head;
}
```

和头插法相反，这个是正序排列。每个循环结束后pHead为当前循环新建的结点，进入下一个循环后，前一个循环的结点指向新建的结点。所有循环结束后，再将最后一个结点指向头结点即可。

大部分操作都差不多，贴一下代码：

```
//链表置空
Linklist *DestroyList(Linklist *HeadNode){
    Linklist *CurrentNode=HeadNode->pNext;
    while(CurrentNode!=HeadNode){
        Linklist *NextNode=CurrentNode->pNext;
        free(CurrentNode);
        CurrentNode=NextNode;
    }
    HeadNode->pNext=HeadNode;
    return HeadNode;
}
//输出链表
void PrintLink(Linklist *HeadNode){
    if(HeadNode->pNext==HeadNode){
        printf("NULL\n");
        return;
    }
    Linklist *CurrentNode=HeadNode->pNext;
    while(CurrentNode!=HeadNode){
        printf("%d ",CurrentNode->Item);
        CurrentNode=CurrentNode->pNext;
    }
    printf("\n");
}
//查找指定数据，返回第一个符合要求的结点
//查找的次数可能是1次，2次...n次，所以期望为（1+2+...+n）/n=（n+1）/2次
Linklist * SearchItem(Linklist *Head,int Data){
    Linklist *CurrentNode=Head->pNext;
    int i=1;
    while((CurrentNode!=Head)&&(CurrentNode->Item!=Data)){
        CurrentNode=CurrentNode->pNext;
        i++;
    }
    if(CurrentNode->Item==Data){
        printf("Position of %d is %d\n",Data,i);
        return CurrentNode;
    }
    else{
        return NULL;
    }
}
//删除指定位置结点
Linklist * DeleteList(Linklist *HeadNode,int Index){
    int i=0;
    Linklist *CurrentNode=HeadNode;
    while(i<Index-1){
        if(CurrentNode==NULL){
            printf("Invalid delete location");
            return HeadNode;
        }
        CurrentNode=CurrentNode->pNext;
        i++;
```

```
    }
    Linklist *NodeToDelete=CurrentNode->pNext;
    CurrentNode->pNext=NodeToDelete->pNext;
    free(NodeToDelete);
    return HeadNode;
}
//剩下的也都差不多，看几个例子：
```

1.约瑟夫问题

N个人围成一圈顺序编号为1、2...N，从1号开始按1、2、3...顺序报数，报m的人退出，其余的人再从1、2、3开始报数，报m的人再退出，以此类推。 请按退出顺序输出每个退出人的原序号。

思路：构建一个没有头结点的循环链表。报数循环，每报到m就删除对应结点，下一个结点从1开始报数。直到只剩下一个人退出循环。

```
void Josephus(int N,int m,Linklist *Head){
    if(m==1){
        int j=0;
        for(j=0;j<N;j++){
            printf("%d ",a[i]);
        }
        return;
    }
    Linklist *CurrentNode=Head;
    while(CurrentNode->pNext!=CurrentNode){
        int i=1;
        while(i<m-1){
            CurrentNode=CurrentNode->pNext;
            i++;
        }
        Linklist *NodeToDelete=CurrentNode->pNext;
        printf("%d ",NodeToDelete->Item);
        CurrentNode->pNext=NodeToDelete->pNext;
        free(NodeToDelete);
        CurrentNode=CurrentNode->pNext;
    }
    printf("%d ",CurrentNode->Item);
}
```

也可以用递归的方法：

```
int Josephus(int N,int m){
    if(N==1){
        return N;
    }
    return (Josephus(N-1,m)+m-1)%N+1;
}
```

循环链表的完整代码如下：

```
#include <stdio.h>
#include <malloc.h>

typedef struct LLN{
    int Item;
```

```
    struct LLN *pNext;
}Linklist;

int a[65536];
void CreateLinkList(Linklist *Head,int *Array,int DataNum);
void Josephus(int N,int m,Linklist *Head);

int main(){
    int N,m,i;
    scanf("%d%d",&N,&m);
    for(i=0;i<N;i++){
        a[i]=i+1;
    }
    Linklist *Head=(Linklist *)malloc(sizeof(Linklist));
    CreateLinkList(Head,a,N);
    Josephus(N,m,Head);
    return 0;
}

//尾插法（无头结点）
void CreateLinkList(Linklist *Head,int *Array,int DataNum){
    int i=0;
    Linklist *pHead=Head;
    pHead->Item=Array[i];
    for(i=1;i<DataNum;i++){
        Linklist *CurrentNode=(Linklist *)malloc(sizeof(Linklist));
        CurrentNode->Item=Array[i];
        pHead->pNext=CurrentNode;
        pHead=CurrentNode;
    }
    pHead->pNext=Head;
}

void Josephus(int N,int m,Linklist *Head){
    if(m==1){
        int j=0;
        for(j=0;j<N;j++){
            printf("%d ",a[j]);
        }
        return;
    }
    Linklist *CurrentNode=Head;
    while(CurrentNode->pNext!=CurrentNode){
        int i=1;
        while(i<m-1){
            CurrentNode=CurrentNode->pNext;
            i++;
        }
        Linklist *NodeToDelete=CurrentNode->pNext;
        printf("%d ",NodeToDelete->Item);
        CurrentNode->pNext=NodeToDelete->pNext;
        free(NodeToDelete);
        CurrentNode=CurrentNode->pNext;
    }
    printf("%d ",CurrentNode->Item);
}
```

2.魔术师发牌

魔术师手中有A、2、3.....J、Q、K十三张黑桃扑克牌。在表演魔术前，魔术师已经将他们按照一定的顺序叠放好，并以有花色的一面朝下。魔术表演过程为：一开始，魔术师数1，然后把最上面的那张牌翻过来，是黑桃A；然后将其放到桌面上；第二次，魔术师数1、2；将第一张牌放到这些牌的最下面，将第二张牌翻转过来，正好是黑桃2；第三次，魔术师数1、2、3；将第1、2张牌依次放到这些牌的最下面，将第三张牌翻过来正好是黑桃3.....直到将所有的牌都翻出来为止。输出原来牌的顺序。（A=1，J=11，Q=12，K=13）

思路：创建一个不含头结点的循环链表，其中13个结点分别存放13张牌。第一个位置放A，下一个位置从1开始数，数到2的位置放2。之后再从1开始数，数到几号牌的位置放几号牌，依此循环即可。每个循环跳过的位置相当于被放到了牌堆的底部。

完整代码如下：

```
#include <stdio.h>
#include <malloc.h>

typedef struct LLN{
    int Item;
    struct LLN *pNext;
}Linklist;

int a[13];

void CreateLinkList(Linklist *Head,int *Array,int DataNum);
void PrintLink(Linklist *HeadNode);

int main(){
    Linklist *Head=(Linklist *)malloc(sizeof(Linklist));
    CreateLinkList(Head,a,13);//创建链表
    Linklist *CurrentNode=Head;
    int i,j;
    for(i=0;i<13;i++){
        j=0;
        while(j<i){//从第1个数到第i个，后移了i-1步。
            if(CurrentNode->Item==0){//当前结点没有放过牌，跳过后算向后移一步。
                CurrentNode=CurrentNode->pNext;
                j++;
            }
            else{//当前结点放过牌，跳过后不算向后移一步。
                CurrentNode=CurrentNode->pNext;
            }
        }
        while(CurrentNode->Item!=0){//判断跳完最后一步时，该位置是否已经放过牌。若放过，
跳到下一个没放过的为止。
            CurrentNode=CurrentNode->pNext;
        }
        CurrentNode->Item=i+1;
        CurrentNode=CurrentNode->pNext;//从下一个开始继续。为下一个循环的第1个。
    }
    PrintLink(Head);
    return 0;
}

//尾插法（无头结点）
void CreateLinkList(Linklist *Head,int *Array,int DataNum){
```

```
    int i=0;
    Linklist *pHead=Head;
    pHead->Item=Array[i];
    for(i=1;i<DataNum;i++){
        Linklist *CurrentNode=(Linklist *)malloc(sizeof(Linklist));
        CurrentNode->Item=Array[i];
        pHead->pNext=CurrentNode;
        pHead=CurrentNode;
    }
    pHead->pNext=Head;
}

void PrintLink(Linklist *HeadNode){
    Linklist *CurrentNode=(Linklist *)HeadNode;
    printf("%d ",CurrentNode->Item);
    CurrentNode=CurrentNode->pNext;
    while(CurrentNode!=HeadNode){
        printf("%d ",CurrentNode->Item);
        CurrentNode=CurrentNode->pNext;
    }
    printf("\n");
}
```

双向链表

双向链表的每个结点中有两个指针，分别指向前置结点和后置结点。首元结点中的指针一个指向后置结点，一个为空指针；最后一个结点中的指针一个指向前置结点，一个为空指针。所以从双向链表的任意一个结点都可以访问它的前置结点和后置结点。双向链表既可以从头到尾遍历，也可以从尾到头遍历。（之前的只有一个指针的叫单链表）

双向链表也可以首尾相连构成双向循环链表，首元结点中的指针一个指向后置结点，一个指向最后一个结点；最后一个结点中的指针一个指向前置结点，一个指向首元结点。

```
typedef struct LLN{
    struct LLN *pPre;
    int Item;
    struct LLN *pNext;
}Linklist;
```

结点的初始化：

```
void Init_LinkList(Linklist *Head){
    Head=(Linklist *)malloc(sizeof(Linklist));
    Head->Item=0;
    Head->pPre=NULL;
    Head->pNext=NULL;
}
```

链表的创建：

```
void CreateLinkList(Linklist *Head,int *Array,int DataNum){
    if(DataNum==1){
        Head=(Linklist *)malloc(sizeof(Linklist));
        Head->Item=Array[0];
        Head->pPre=NULL;//改成Head就是循环链表。
        Head->pNext=NULL;//改成Head就是循环链表。
```

```
    }
    else{
        Head=(Linklist *)malloc(sizeof(Linklist));
        Head->Item=Array[0];
        Head->pPre=NULL;
        Linklist *PreNode=Head;
        int i;
        for(i=1;i<DataNum;i++){
            Linklist *CurrentNode=(Linklist *)malloc(sizeof(Linklist));
            CurrentNode->Item=Array[i];
            CurrentNode->pPre=PreNode;
            PreNode->pNext=CurrentNode;
            PreNode=CurrentNode;
        }
        PreNode->pNext=NULL;
        //PreNode->pNext=Head;
        //Head->pPre=PreNode;
        //改成这两句就是循环链表。
    }
}
```

双向链表插入数据

1.添加至表头：将待添加结点与首元结点建立双层逻辑关系。

2.添加至表中：待添加结点先与后置结点建立双层逻辑关系；再将前置结点与待添加结点建立双层逻辑关系。

3.添加至表尾：将最后一个结点与待添加结点建立双层逻辑关系。

```
Linklist *InsertList(Linklist *Head,int LocateIndex,int InData){
    //以插入指定结点前为例。
    Linklist *NodeToInsert=(Linklist *)malloc(sizeof(Linklist));
    NodeToInsert->Item=InData;
    if(LocateIndex==1){
        NodeToInsert->pNext=Head;
        NodeToInsert->pPre=NULL;
        Head->pPre=NodeToInsert;
    }
    else{
        Linklist *CurrentNode=Head;
        int i;
        for(i=1;i<LocateIndex-1;i++){//待插入位置的前一个结点。
            CurrentNode=CurrentNode->pNext;
        }
        if(CurrentNode->pNext==NULL){
            CurrentNode->pNext=NodeToInsert;
            NodeToInsert->pPre=CurrentNode;
            NodeToInsert->pNext=NULL;
        }
        else{
            NodeToInsert->pNext=CurrentNode->pNext;
            (CurrentNode->pNext)->pPre=NodeToInsert;
            CurrentNode->pNext=NodeToInsert;
            NodeToInsert->pPre=CurrentNode;
        }
    }
    return Head;
```

```
}
```

双向链表删除数据

```
Linklist *DeleteList(Linklist *HeadNode,int Index){
    Linklist *CurrentNode=HeadNode;
    if(Index==1){
        HeadNode=HeadNode->pNext;
        HeadNode->pPre=NULL;
        free(CurrentNode);
    }
    else{
        int i=1;
        while(i<Index-1){
            CurrentNode=CurrentNode->pNext;
            i++;
        }//此时CurrentNode指向待删除位置的前一个结点。
        if((CurrentNode->pNext)->pNext==NULL){
            Linklist *NodeToDelete=CurrentNode->pNext;
            CurrentNode->pNext=NULL;
            free(NodeToDelete);
        }
        else{
            Linklist *NodeToDelete=CurrentNode->pNext;
            CurrentNode->pNext=NodeToDelete->pNext;
            (NodeToDelete->pNext)->pPre=CurrentNode;
            free(NodeToDelete);
        }
    }
    return HeadNode;
}
```

词频统计

统计一个英文文本文件中每个单词的出现次数，并将统计结果按单词字典序输出到屏幕上。单词为仅由字母组成的字符序列。包含大写字母的单词应将大写字母转换为小写字母后统计。按字典顺序输出单词及出现顺序。

思路：先读取文章，然后依次读取单词。单词以字母开头，读到第一个非字母结束。创建一个结构体存储单词的内容与频数。对于每一个单词，与已存储的单词依次比较，若有相同则频数+1，没有就复制到下一个结构体中。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct WFC{
    char word[100];
    int times;
}Word;

Word Vocab[1000];
char Article[1000],CurrentWord[100];
int WordCmp(int cnt,char *CurrentWord,int *Location);
int cmp(const void *p1,const void *p2);

int main(){
```

```
    //先读取文章。
    char ch=getchar();
    int i=0;
    while(ch!=EOF){
        if(ch>=65&&ch<=90){
            ch=ch+'a'-'A';
        }//大小写转换。
        Article[i]=ch;
        ch=getchar();
        i++;
    }

    //再读取单词。
    int L=strlen(Article),cnt=0,j,k;//j、k计数用 ， cnt表示已有的单词数量。
    for(i=0;i<L;i++){
        if(Article[i]>=97&&Article[i]<=122){//是字母。
            for(j=i;j<L;j++){
                if(Article[j]<97||Article[j]>122){//数到不是字母为止，确定单词长度。
                    break;
                }
            }//单词范围为i至j-1。
            memset(CurrentWord,'\0',sizeof(CurrentWord));
            for(k=i;k<j;k++){
                CurrentWord[k-i]=Article[k];
            }
            int *Location;
            if(WordCmp(cnt,CurrentWord,Location)==1){//单词已存在。
                Vocab[*Location].times++;
            }
            else{//单词不存在。
                Vocab[cnt].times=1;
                strcpy(Vocab[cnt].word,CurrentWord);
                cnt++;
            }
            i=j;
        }
        else{
            continue;
        }
    }

    //单词存储完毕后，按字典顺序排序。
    qsort(Vocab,cnt,sizeof(Word),cmp);
    //最后输出。
    for(i=0;i<cnt;i++){
        printf("%s %d\n",Vocab[i].word,Vocab[i].times);
    }
    return 0;
}

int WordCmp(int cnt,char *CurrentWord,int *Location){//单词已存在返回1，不存在返回0。
    if(cnt==0){
        return 0;
    }
    int i;
    for(i=0;i<cnt;i++){
        if(strcmp(Vocab[i].word,CurrentWord)==0){
            *Location=i;
```

```
            return 1;
        }
    }
    return 0;
}

int cmp(const void *p1,const void *p2){
    struct WFC *a=(struct WFC *)p1;
    struct WFC *b=(struct WFC *)p2;
    return strcmp(a->word,b->word);
}
```