链表

链表中包含很多（或0个）结点，每个结点中包含一个数据结构和一个指针。数据结构用于存放各种类型的数据，指针用来指向下一个结点存储的地址，一般称为next。因为每个结点都是链表类型，所以每个指针也要定义为链表类型的指针。

链表的结点

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

typedef用于为复杂的声明定义简单的别名。以上的代码完成了两个操作：

1.定义了一个名为LLN的结构体变量。（这个操作无论有没有typedef都可以完成）

2.typedef struct LLN Linklist;为结构体变量LLN起了一个新名字。之后就可以用Linklist来定义变量。（如果没有typedef就必须要用struct LLN定义）

如果用以下写法会出现问题：

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

因为声明的pNext类型是Linklist，但Linklist这个数据类型本身还没有建立完成，所以编译器会报错。

正确的写法：

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

```
typedef struct LLN *pNode;
struct LLN{
    int Item;
    pNode pNext;//不用加*
};
```

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

//typedef通常比#define更好，例如：

```
#define pStr1 char*
typedef char* pStr2;
pStr1 s1,s2;//s1定义为字符型指针变量，而s2定义为字符型变量
//改为pStr1 s1,*s2;就和下一行一个效果
pStr2 s3,s4;//s3s4都定义为字符型指针变量
```

因为#define是字符串替换，而typedef是给一个类型起新名字。

多次使用时，使用define会多次分配内存空间，内存消耗多，而typedef不会。

结点的初始化

在没有任何输入的情况下对链表进行初始化，作用是生成一个链表的头指针，以便后续的函数调用。

首先要定义一个头指针，用来保存即将创建的链表。需要在函数内定义并申请一个结点的空间。在函数的结尾将这个结点作为新建链表的头指针返回给主调函数。

```
Linklist* List_init(){
    Linklist *HeadNode=(Linklist *)malloc(sizeof(Linklist));
    if(HeadNode==NULL){
        printf("Insufficient space cache");
        return HeadNode;
    }
    HeadNode->Item=0;
    HeadNode->pNext=NULL;
    return HeadNode;
}
//比如说这个函数的返回值就是链表的头指针。除非内存不足，否则头指针不为空。
//HeadNode所指向的结点就是链表的头结点。头结点中的pNext指向首元结点。
```

关于头结点和头指针的区别：

链表的头结点放置在第一个有效元素结点（首元结点）之前，其数据域一般无意义。有了头结点后，再第一个结点前插入结点或删除第一个结点的操作，就与其他结点统一了。但头结点不是必须的。

若链表有头结点，则链表的头指针是指向头结点的指针。若没有则链表的头指针指向第一个结点。头指针是链表的必要元素，无论链表是否为空，头指针都不为空。链表的名字就是头指针的名字。

完整链表的创建

创建链表就是将既定的数据按照链表的结构进行存储。例如：使用数组对链表赋值。

创建链表需要给定的头指针和需要初始化的数据序列（数组的头指针，数组的大小）作为输入参数。

先将首元结点用数组的第一个元素初始化，再在首元结点之后创建新的链表结点，赋值数组中的其余元素。

```
void CreateList(Linklist *HeadNode,int *Array,int DataNum){
    int i=0;
    Linklist *CurrentNode=(Linklist *)HeadNode;//没有头结点的情况，即头指针指向首元结点。
    //HeadNode->pNext=(Linklist *)malloc(sizeof(Linklist));//有头结点的情况，先为头结点中的pNext申请一块内存。
    //Linklist *CurrentNode=(Linklist *)HeadNode->pNext;//CurrentNode为指向首元结点的指针，即pNext。
    for(i=0;i<DataNum;i++){
        CurrentNode->Item=Array[i];
        if(i<DataNum-1){//新建结点的前提是上一个结点不是最后一个，这样可以避免结点的浪费。
```

```
            CurrentNode->pNext=(Linklist *)malloc(sizeof(Linklist));//pNext是指向下一个结点的指针。
            CurrentNode=CurrentNode->pNext;//进入下一个循环，为下一个结点赋值。
        }
    }
    CurrentNode->pNext=NULL;//最后一个结点中的pNext为空指针。
}
```

指针变量CurrentNode用来表示当前结点的指针，最初指向首元结点的位置。然后根据数组的大小进行循环赋值，每次赋值完成后要重新申请一个结点的空间，用来存放下一个结点的内容，并且要将CurrentNode指向新生成的结点。

插入链表结点

1.在链表尾部插入：新建一个链表结构，将原来的链表尾指针指向这个新建的结构。

2.在链表中间插入：（指定结点前或指定结点后，原理相同。以指定结点后为例）

链表中的两个结点A1和A2，要在A1后插入一个新的结点。首先新建一个结点叫NodeToInsert，然后将NodeToInsert中的pNext指向A2，并将A1中的pNext指向NodeToInsert。以上操作顺序不能改变，若先将A1中的pNext指向NodeToInsert，则失去了A2的寻址方式。（在A1的pNext指向别的地方之前，要把A2的地址存下，或赋值给NodeToInsert中的pNext。）

插入指定结点后：

```
Linklist * InsertList(Linklist *HeadNode,int LocateIndex,int InData){//头指针，待插入位置，待插入数据。
    //此处的待插入位置指的是插在第LocateIndex个结点之后。
    int i=1;//HeadNode指向首元结点。
    //若HeadNode指向头结点，i=0。
    Linklist *CurrentNode=(Linklist *)HeadNode;
    while(i<LocateIndex){
        if(CurrentNode==NULL){
            printf("Invalid insertion position");
            return HeadNode;//插入位置无效。
        }
        CurrentNode=CurrentNode->pNext;
        i++;
    }//此时CurrentNode指向第LocateIndex个结点。
    Linklist *NodeToInsert=(Linklist *)malloc(sizeof(Linklist));
    if(NodeToInsert==NULL){
        printf("Insufficient space cache");
        return HeadNode;//空间缓存不足。
    }
    NodeToInsert->Item=InData;//插入的数值。
    NodeToInsert->pNext=CurrentNode->pNext;//NodeToInsert中的pNext指向下一个结点。
    CurrentNode->pNext=NodeToInsert;//上一个结点中的pNext指向NodeToInsert。
    return HeadNode;//插入成功。
}
```

删除链表结点

可以删除指定位置的结点或指定元素的结点。

先锁定待删除的结点的位置，将该结点的后置结点链接到前置结点的指针处，即令前置结点的pNext指向后置结点。则待删除结点就从原来的链表中脱离出去了。注意删除后的结点要用free函数释放，否则会造成内存泄漏。

删除指定位置的结点：

```
Linklist * DeleteList(Linklist *HeadNode,int Index,int *DataToDelete){//头指针，待删除位置，传出删除的数据。
    int i=1;//HeadNode指向首元结点。
    //若HeadNode指向头结点，i=0。
    Linklist *CurrentNode=HeadNode;
    while(i<Index-1){
        if(CurrentNode==NULL){
            printf("Invalid delete location");
            return HeadNode;//删除位置无效。
        }
        CurrentNode=CurrentNode->pNext;
        i++;
    }//此时CurrentNode指向待删除位置的前一个结点。
    Linklist *NodeToDelete=CurrentNode->pNext;//记录待删除结点的指针，便于释放。
    *DataToDelete=NodeToDelete->Item;//可以记录被删除的数据。
    CurrentNode->pNext=NodeToDelete->pNext;//前置结点的pNext指向后置结点。
    free(NodeToDelete);
    return HeadNode;
}
```

如果想删除的结点是首元结点，就传入指向头结点的HeadNode，令i=0即可。

删除指定元素的结点：

```
Linklist * DeleteList(Linklist *HeadNode,int DataToDel){//头指针，指定的删除元素。
    //此处HeadNode指向首元结点，可能出现首元结点就需要删除的情况。
    Linklist *pNode=HeadNode;
    //如果HeadNode指向头结点，则令pNode=HeadNode->pNext。
    while((pNode!=NULL)&&(pNode->Item==DataToDel)){
        Linklist *DelNode=pNode;
        pNode=pNode->pNext;
        free(DelNode);
    }//这样就保证了DataToDel开头的结点全部被删除.
    if(pNode==NULL){
        printf("All deleted");
        return NULL;
    }//链表被删空了，说明链表里的元素全是DataToDel。
    Linklist *PreNode=pNode;//当前结点中的数据一定不是DataToDel。
    Linklist *CurrentNode=pNode->pNext;//从下一个开始判断。
    while(CurrentNode!=NULL){//判断到结束为止，若是指定元素则删除，不是则判断下一个。
        if(CurrentNode->Item==DataToDel){
            Linklist *DelNode=CurrentNode;//记录被删除的节点。
            PreNode->pNext=CurrentNode->pNext;//把前置结点与后置结点相连。
            CurrentNode=CurrentNode->pNext;//当前结点向后移一位。
            free(DelNode);
        }
        else{
            PreNode=PreNode->pNext;
            CurrentNode=CurrentNode->pNext;
        }
    }
    return HeadNode;
}
```

获取链表长度

```
int GetListLength(Linklist *HeadNode){//此处HeadNode指向头结点。
    int ListLength=0;
    Linklist *CurrentNode=HeadNode->pNext;
    while(CurrentNode!=NULL){
        ListLength++;
        CurrentNode=CurrentNode->pNext;
    }
    return ListLength;
}
```

链表置空

```
Linklist *DestroyList(Linklist *HeadNode){//此处HeadNode指向头结点。
    Linklist *CurrentNode=HeadNode->pNext;
    while(CurrentNode!=NULL){
        Linklist *NextNode=CurrentNode->pNext;
        free(CurrentNode);
        CurrentNode=NextNode;
    }
    HeadNode->pNext=NULL;
    return HeadNode;
}
```

链表逆序

假设一个链表的头指针为Head，指向首元结点A1。含有五个结点A1,A2,A3,A4,A5。从A1开始向后遍历，将后面的每一个结点移动到Head之后，遍历到最后一个结点时即完成倒序。移动的步骤为：（以A2为例）

1.将A3链接到A1后面；H->1->3->4->5

2.将Head后面的整体链接到A2后面；2->1->3->4->5

3.将A2链接到Head后面。H->2->1->3->4->5

之后是A3：

1.将A4链接到A1后面；H->2->1->4->5

2.将Head后面的整体链接到A3后面；3->2->1->4->5

3.将A3链接到Head后面。H->3->2->1->4->5

对于An：1至n-1逆序排列，n+1之后正序排列。

先将An+1连在A1后面，再将An-1连在An后面，最后将An连在Head后面。

依次遍历即可完成逆序。

```
Linklist *ListRotate(Linklist *HeadNode){
    //此处HeadNode指向头结点。
    Linklist *Pre=(Linklist*)malloc(sizeof(Linklist));
    if(Pre==NULL){
        printf("Insufficient space cache");
        return HeadNode;//空间缓存不足。
    }
    Pre->pNext=HeadNode->pNext;//即为以上说明中的Head。
```

```
    Linklist *CurrentNode=HeadNode->pNext;//即为以上说明中的A1，在逆序过程中A1后面的每个结点依次移动到首元结点的位置。
    while(CurrentNode->pNext!=NULL){//CurrentNode->pNext==NULL时，说明A1已经到了最后一个，逆序完成。
        Linklist *Next=CurrentNode->pNext;
        CurrentNode->pNext=Next->pNext;
        Next->pNext=Pre->pNext;
        Pre->pNext=Next;
    }
    return Pre->pNext;
}
```

以上述说明为例，Pre为Head，CurrentNode为A1，Next为需要移动的结点。

1.A1与Next的后置结点相连；

2.Next与Pre的后置结点相连，即为将Head后面的整体链接到Next后面；

3.Pre与Next相连，即为将Next链接到Head后面。

循环结束后，Next从A1后面移到了最前面（Head的后置结点）。

进入下一个循环时，Next变为A1后面的新结点，相当于向后移动了一位。

当CurrentNode->pNext==NULL时，说明A1已经到了最后一个，逆序完成。

或者也可以采用递归的方法：

假设有一个函数，可以将链表逆序，则对于每一个链表可以进行分解：将链表的第一个结点与之后的部分分开。

递归的过程为：用函数将之后的部分逆序，再把第一个结点放在逆序链表的末尾。

递归的终止条件为链表中只剩下一个结点，就返回这个结点。

```
Linklist *ListRotate(Linklist *HeadNode){//此处HeadNode指向首元结点。
    Linklist *CurrentNode=HeadNode;
    if((CurrentNode==NULL)||(CurrentNode->pNext==NULL)){
        return CurrentNode;
    }
    Linklist *HeadOfReverse=ListRotate(CurrentNode->pNext);
    Linklist *LastNode=CurrentNode->pNext;
    LastNode->pNext=CurrentNode;
    CurrentNode->pNext=NULL;
    return HeadOfReverse;
}
```

完整的代码

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

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

void CreateList(Linklist *HeadNode,int *Array,int DataNum);
void PrintLink(Linklist *HeadNode);
```

```c
void PrintLink2(Linklist *HeadNode);
Linklist * InsertList1(Linklist *HeadNode,int LocateIndex,int InData);
Linklist * InsertList2(Linklist *HeadNode,int LocateIndex,int InData);
Linklist * DeleteList1(Linklist *HeadNode,int Index,int *DataToDelete);
Linklist * DeleteList2(Linklist *HeadNode,int DataToDel);
int GetListLength(Linklist *HeadNode);
Linklist *DestroyList(Linklist *HeadNode);
Linklist *ListRotate1(Linklist *HeadNode);
Linklist *ListRotate2(Linklist *HeadNode);

int main(){
    int a[10]={1,2,3,4,5,6,7,8,9,0};

    Linklist *HeadNode=(Linklist *)malloc(sizeof(Linklist));
    HeadNode->Item=0;
    HeadNode->pNext=NULL;
    //HeadNode为头指针，指向头结点。

    CreateList(HeadNode,a,10);
    PrintLink(HeadNode);
    InsertList1(HeadNode,4,10);
    PrintLink(HeadNode);
    InsertList2(HeadNode,4,10);
    PrintLink(HeadNode);

    int *pDataToDel=(int *)malloc(sizeof(int));
    DeleteList1(HeadNode,4,pDataToDel);
    PrintLink(HeadNode);
    printf("The deleted data is %d\n",*pDataToDel);
    free(pDataToDel);
    printf("The length of the linked list is %d\n",GetListLength(HeadNode));

    DeleteList2(HeadNode,10);
    PrintLink(HeadNode);

    Linklist *HeadOfRev=ListRotate1(HeadNode);
    PrintLink2(HeadOfRev);
    Linklist *HeadOfRev2=ListRotate2(HeadOfRev);
    PrintLink2(HeadOfRev2);

    printf("DestroyList\n");
    DestroyList(HeadNode);
    PrintLink(HeadNode);
    return 0;
}

void CreateList(Linklist *HeadNode,int *Array,int DataNum){
    int i=0;
    HeadNode->pNext=(Linklist *)malloc(sizeof(Linklist));
    Linklist *CurrentNode=(Linklist *)HeadNode->pNext;
    for(i=0;i<DataNum;i++){
        CurrentNode->Item=Array[i];
        if(i<DataNum-1){
            CurrentNode->pNext=(Linklist *)malloc(sizeof(Linklist));
            CurrentNode=CurrentNode->pNext;
        }
    }
    CurrentNode->pNext=NULL;
```

```
}

void PrintLink(Linklist *HeadNode){//这个HeadNode指向头结点，底下的代码有小改动
    Linklist *CurrentNode=(Linklist *)HeadNode->pNext;
    if(CurrentNode==NULL){
        printf("NULL");
    }
    while(CurrentNode!=NULL){
        printf("%d ",CurrentNode->Item);
        CurrentNode=CurrentNode->pNext;
    }
    printf("\n");
}

void PrintLink2(Linklist *HeadNode){
    Linklist *CurrentNode=(Linklist *)HeadNode;
    if(CurrentNode==NULL){
        printf("NULL");
    }
    while(CurrentNode!=NULL){
        printf("%d ",CurrentNode->Item);
        CurrentNode=CurrentNode->pNext;
    }
    printf("\n");
}

Linklist * InsertList1(Linklist *HeadNode,int LocateIndex,int InData){//头指针，待插入位置，待插入数据。
    //此处的待插入位置指的是插在第LocateIndex个结点之后。
    int i=0;
    Linklist *CurrentNode=(Linklist *)HeadNode;
    while(i<LocateIndex){
        if(CurrentNode==NULL){
            printf("Invalid insertion position\n");
            return HeadNode;//插入位置无效。
        }
        CurrentNode=CurrentNode->pNext;
        i++;
    }//此时CurrentNode指向第LocateIndex个结点。
    Linklist *NodeToInsert=(Linklist *)malloc(sizeof(Linklist));
    if(NodeToInsert==NULL){
        printf("Insufficient space cache\n");
        return HeadNode;//空间缓存不足。
    }
    NodeToInsert->Item=InData;//插入的数值。
    NodeToInsert->pNext=CurrentNode->pNext;//NodeToInsert中的pNext指向下一个结点。
    CurrentNode->pNext=NodeToInsert;//上一个结点中的pNext指向NodeToInsert。
    return HeadNode;//插入成功。
}

Linklist * InsertList2(Linklist *HeadNode,int LocateIndex,int InData){//头指针，待插入位置，待插入数据。
    //此处的待插入位置指的是插在第LocateIndex个结点之前，即插入后的新结点是第LocateIndex个。
    int i=0;
    Linklist *CurrentNode=(Linklist *)HeadNode;
    while(i<LocateIndex-1){
        if(CurrentNode==NULL){
```

```
            printf("Invalid insertion position");
            return HeadNode;//插入位置无效。
        }
        CurrentNode=CurrentNode->pNext;
        i++;
    }//此时CurrentNode指向第LocateIndex-1个结点。
    Linklist *NodeToInsert=(Linklist *)malloc(sizeof(Linklist));
    if(NodeToInsert==NULL){
        printf("Insufficient space cache");
        return HeadNode;//空间缓存不足。
    }
    NodeToInsert->Item=InData;//插入的数值。
    NodeToInsert->pNext=CurrentNode->pNext;//NodeToInsert中的pNext指向第
LocateIndex个结点。
    CurrentNode->pNext=NodeToInsert;//第LocateIndex-1个结点中的pNext指向
NodeToInsert。
    return HeadNode;//插入成功。
}

Linklist * DeleteList1(Linklist *HeadNode,int Index,int *DataToDelete){//头指针，
待删除位置，传出删除的数据。
    int i=0;
    Linklist *CurrentNode=HeadNode;
    while(i<Index-1){
        if(CurrentNode==NULL){
            printf("Invalid delete location\n");
            return HeadNode;//删除位置无效。
        }
        CurrentNode=CurrentNode->pNext;
        i++;
    }//此时CurrentNode指向待删除位置的前一个结点。
    Linklist *NodeToDelete=CurrentNode->pNext;//记录待删除结点的指针，便于释放。
    *DataToDelete=NodeToDelete->Item;//可以记录被删除的数据。
    CurrentNode->pNext=NodeToDelete->pNext;//前置结点的pNext指向后置结点。
    free(NodeToDelete);
    return HeadNode;
}

Linklist * DeleteList2(Linklist *HeadNode,int DataToDel){//头指针，指定的删除元素。
    Linklist *pNode=HeadNode->pNext;
    while((pNode!=NULL)&&(pNode->Item==DataToDel)){
        Linklist *DelNode=pNode;
        pNode=pNode->pNext;
        free(DelNode);
    }//这样就保证了DataToDel开头的结点全部被删除.
    if(pNode==NULL){
        printf("All deleted");
        return NULL;
    }//链表被删空了，说明链表里的元素全是DataToDel。
    Linklist *PreNode=pNode;//当前结点中的数据一定不是DataToDel。
    Linklist *CurrentNode=pNode->pNext;//从下一个开始判断。
    while(CurrentNode!=NULL){//判断到结束为止，若是指定元素则删除，不是则判断下一个。
        if(CurrentNode->Item==DataToDel){
            Linklist *DelNode=CurrentNode;
            PreNode->pNext=CurrentNode->pNext;
            CurrentNode=CurrentNode->pNext;
            free(DelNode);
        }
```

```
        else{
            PreNode=PreNode->pNext;
            CurrentNode=CurrentNode->pNext;
        }
    }
    return HeadNode;
}

int GetListLength(Linklist *HeadNode){//此处HeadNode指向头结点。
    int ListLength=0;
    Linklist *CurrentNode=HeadNode->pNext;
    while(CurrentNode!=NULL){
        ListLength++;
        CurrentNode=CurrentNode->pNext;
    }
    return ListLength;
}

Linklist *DestroyList(Linklist *HeadNode){
    Linklist *CurrentNode=HeadNode->pNext;
    while(CurrentNode!=NULL){
        Linklist *NextNode=CurrentNode->pNext;
        free(CurrentNode);
        CurrentNode=NextNode;
    }
    HeadNode->pNext=NULL;
    return HeadNode;
}

Linklist *ListRotate1(Linklist *HeadNode){
    //此处HeadNode指向头结点。
    Linklist *Pre=(Linklist*)malloc(sizeof(Linklist));
    if(Pre==NULL){
        printf("Insufficient space cache");
        return HeadNode;//空间缓存不足。
    }
    Pre->pNext=HeadNode->pNext;//即为以上说明中的Head。
    Linklist *CurrentNode=HeadNode->pNext;//即为以上说明中的A1，在逆序过程中A1后面的每个结点依次移动到首元结点的位置。
    while(CurrentNode->pNext!=NULL){//CurrentNode->pNext==NULL时，说明A1已经到了最后一个，逆序完成。
        Linklist *Next=CurrentNode->pNext;
        CurrentNode->pNext=Next->pNext;
        Next->pNext=Pre->pNext;
        Pre->pNext=Next;
    }
    return Pre->pNext;
}

Linklist *ListRotate2(Linklist *HeadNode){
    //此处HeadNode指向首元结点。
    Linklist *CurrentNode=HeadNode;
    if((CurrentNode==NULL)||(CurrentNode->pNext==NULL)){
        return CurrentNode;
    }
    Linklist *HeadOfReverse=ListRotate2(CurrentNode->pNext);
    Linklist *LastNode=CurrentNode->pNext;
    LastNode->pNext=CurrentNode;
```

```
    CurrentNode->pNext=NULL;
    return HeadOfReverse;
}
```