> 一研为定

@[toc]

第五次 直播 双向链表

双链表结点的定义：

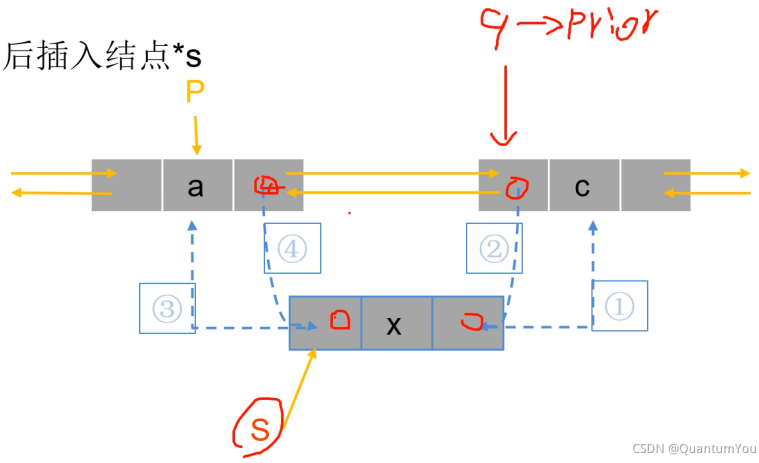
```
typedef struct DNode{          //双链表结点类型
    ElemType data;             //数据域
    struct DNode *prior;       //前驱指针
    struct DNode *next;        //后继指针
}DNode, *DinkList;
```

CSDN @QuantumYou

核心：**注意双向链表的插入次序** ①②③④ 标识

在双链表中p所指的结点之后插入结点*s

1、s->next=p->next
2、p->next->prior=s;
3、s->prior=p;
4、p->next=s;



CSDN @QuantumYou

注意：赋值语句的解读　eg: `p->next = s ;` 的意思为将 s 的值赋给 `p->next`

双向链表的增删改查

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

typedef int ElemType;
typedef struct DNode{
    ElemType data;
    struct DNode *prior,*next;//前驱，后继
}DNode,*DLinkList;
//双向链表头插法
DLinkList Dlist_head_insert(DLinkList &DL)
{
    DNode *s;int x;
    DL=(DLinkList)malloc(sizeof(DNode));//带头结点的链表,不带头结点
    DL->next=NULL;
    DL->prior=NULL;
    scanf("%d",&x);//从标准输入读取数据
    //3 4 5 6 7 9999
```

```
    while(x!=9999){
        s=(DLinkList)malloc(sizeof(DNode));//申请一个空间空间，强制类型转换
        s->data=x;
        s->next=DL->next;
        if(DL->next!=NULL)//插入第一个结点时，不需要这一步操作
        {
            DL->next->prior=s;
        }
        s->prior=DL;
        DL->next=s;
        scanf("%d",&x);//读取标准输入
    }
    return DL;
}
//双向链表尾插法
DLinkList Dlist_tail_insert(DLinkList &DL)
{
    int x;
    DL=(DLinkList)malloc(sizeof(DNode));//带头节点的链表
    DNode *s,*r=DL;
    DL->prior=NULL;
    //3 4 5 6 7 9999
    scanf("%d",&x);
    while(x!=9999){
        s=(DNode*)malloc(sizeof(DNode));
        s->data=x;
        r->next=s;
        s->prior=r;
        r=s;//r指向新的表尾结点
        scanf("%d",&x);
    }
    r->next=NULL;//尾结点的next指针赋值为NULL
    return DL;
}
//按序号查找结点值
DNode *GetElem(DLinkList DL,int i)
{
    int j=1;
    DNode *p=DL->next;
    if(i==0)
        return DL;
    if(i<1)
        return NULL;
    while(p&&j<i)
    {
        p=p->next;
        j++;
    }
    return p;
}
//新结点插入第i个位置
bool DListFrontInsert(DLinkList DL,int i,ElemType e)
{
    DLinkList p=GetElem(DL,i-1);
    if(NULL==p)
    {
        return false;
    }
```

```
    DLinkList s=(DLinkList)malloc(sizeof(DNode));//为新插入的结点申请空间
    s->data=e;
    s->next=p->next;
    p->next->prior=s;
    s->prior=p;
    p->next=s;
    return true;
}
//删除第i个结点
bool DListDelete(DLinkList DL,int i)
{
    DLinkList p=GetElem(DL,i-1);
    if(NULL==p)
    {
        return false;
    }
    DLinkList q;
    q=p->next;
    if(q==NULL)//删除的元素不存在
        return false;
    p->next=q->next;//断链

// 下面注意要进行判断
    if(q->next!=NULL)
    {
        q->next->prior=p;
    }
    free(q);//释放对应结点的空间
    return true;
}
//链表打印
void PrintDList(DLinkList DL)
{
    DL=DL->next;
    while(DL!=NULL)
    {
        printf("%3d",DL->data);
        DL=DL->next;
    }
    printf("\n");
}


//2.3.3 双链表增删查
int main()
{
    DLinkList DL;
    DLinkList search;
    Dlist_head_insert(DL);
    //Dlist_tail_insert(DL);
    //3 4 5 6 7 9999
    PrintDList(DL);
    search=GetElem(DL,2);
    if(search!=NULL)
    {
        printf("按序号查找成功\n");
        printf("%d\n",search->data);
    }
```

```
    DListFrontInsert(DL,3,99);
    PrintDList(DL);
    DListDelete(DL,2);
    PrintDList(DL);
    system("pause");
}
```

双向链表的删除

= 删除双链表中结点 `*p` 的后继结点 `*q`

1、`p->next=q->next ;`

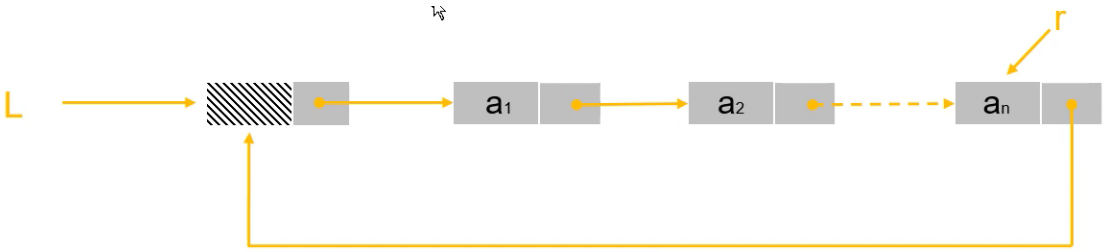
2、`q->next-> prior=p;`

3、`free(q) ;`

第五次 直播 其他链表

循环单链表

- 循环单链表与单链表的区别在于，表中最后一个结点的next指针不是NULL,而是指向头结点L,从而整个链表形成一个环



r->next=L;　　while(pcur->next!=L)

静态链表

- 静态链表是借助数组来描述线性表的链式存储结构，结构类型如下

```
#define Maxsize 50
typedef struct {
ElemType data ;
int next ;
) Slinklist[Maxsize] ;
```

0		2
1	b	6
2	a	1
3	d	-1
4		
5		
6	c	3

常见问题

- 在链表中插入第 i 个位置和删除第 i 个元素不需要用引用的原因在于：不需要改变头节点

第六次 直播 引用解析、栈与队列

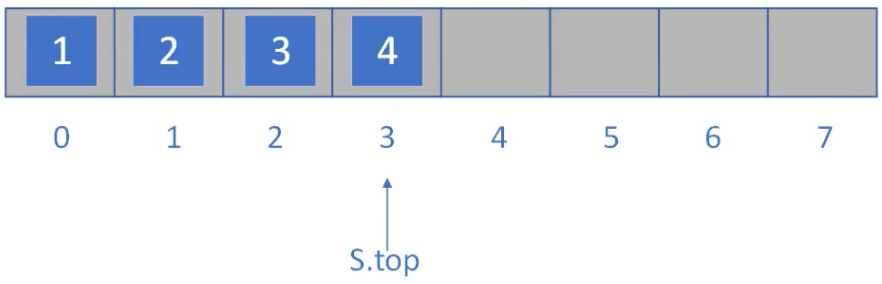
为什么我们需要在形参的地方使用引用?

- 在子函数中去给对应的形参赋值后，子函数结束，主函数中对应的实参就发生了变化，如果没有使用引用，那么在子函数中给形参赋值后，子函数结束，主函数中对应的实参不会变化的

栈的实现

顺序存储实现栈

```
typedef struct{
    Elemtype data[50];
    int top;
}SqStack;
SqStack S;
```



S.data[++S.top]=4;

元素入栈

x=S. data[S. top--]

元素出栈

- 入栈：Top 栈顶指针先加加 ； 出栈：Top 栈顶指针后减减

栈的基本操作

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

#define MaxSize 50
typedef int ElemType;
typedef struct {
    ElemType data[MaxSize];//数组
    int top;
}SqStack;

void InitStack(SqStack &S)
{
    S.top = -1;//代表栈为空
}

bool StackEmpty(SqStack S)
{
    if (-1 == S.top)
    {
        return true;
    }
    return false;
}
bool Push(SqStack& S, ElemType x)
{
    if (S.top == MaxSize - 1)
    {
        return false;//栈满了
    }
    S.data[++S.top] = x;
    return true;//返回true就是入栈成功
}
//获取栈顶元素
bool GetTop(SqStack S, ElemType &x)
{
    if (StackEmpty(S))//栈为空
```

```
    {
        return false;
    }
    x = S.data[S.top];
    return true;
}
bool Pop(SqStack& S, ElemType& x)
{
    if (StackEmpty(S))//栈为空
    {
        return false;
    }
    x = S.data[S.top--];//等价于x = S.data[S.top]；再做   S.top--;
    return true;
}
int main()
{
    SqStack S;
    bool flag;
    ElemType m;//存储拿出来的栈顶元素的
    InitStack(S);//初始化
    flag = StackEmpty(S);
    if (flag)
    {
        printf("栈是空的\n");
    }
    Push(S, 3);//入栈元素3
    Push(S, 4);//入栈元素4
    Push(S, 5);
    flag = GetTop(S, m);//获取栈顶元素,但是S.top值不变
    if (flag)
    {
        printf("获取栈顶元素为 %d\n", m);
    }
    flag = Pop(S, m);//弹出栈顶元素
    if (flag)
    {
        printf("弹出元素为 %d\n", m);
    }
    return 0;
}
```

拓展知识： 链表实现的栈是头部插入与头部删除

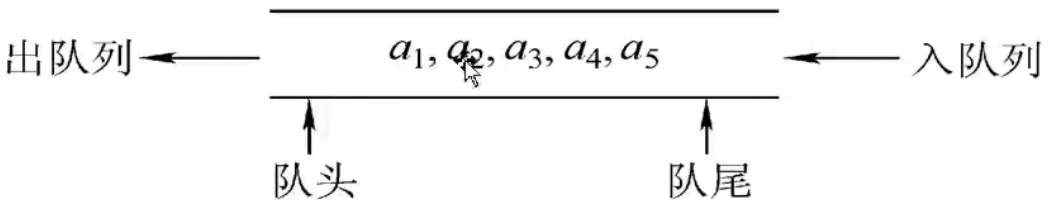
链式存储实现栈(相对不重要)

```
LiStack  Lhead=(LiStack)malloc(sizeof(struct Linknode))
Lhead->next=NULL;
LiStack  top=NULL;
```


```
top=(LiStack)malloc(sizeof(struct Linknode));
top->next=NULL;
top->data=1;
top->next=Lhead->next;
Lhead->next=top;
```


队列

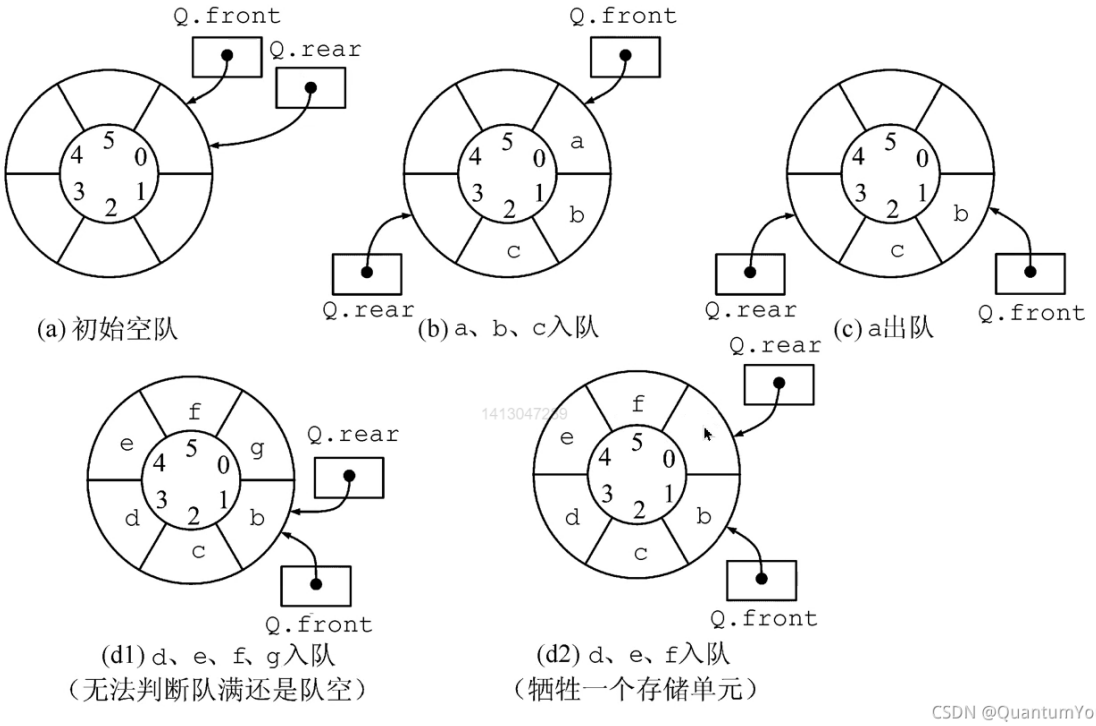
队头（Front）。允许删除的一端，又称队首。
队尾（Rear）。允许插入的一端。



特性是先进先出（**First In First Out，FIFO**）

循环队列

循环队列图示



为了区分队空还是队满的情况：

1、以牺牲一个单元区分队空和队满
2、类型中增设表示数据元素个数的数据单元
3、类型中增设tag 数据成员

入队

```
bool EnQueue(SqQueue &Q,ElemType x)
{
    if((Q.rear+1)%MaxSize==Q.front) //判断是否队满
        return false;
    Q.data[Q.rear]=x;//放入元素
    Q.rear=(Q.rear+1)%MaxSize;//改变队尾标记
    return true;
}
```


出队

```
bool DeQueue(SqQueue &Q,ElemType &x)
{
    if(Q.rear==Q.front)
    return false;
    x=Q.data[Q.front];//先进先出
    Q.front=(Q.front+1)%MaxSize;
    return true;
}
```


队列全局代码

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

#define MaxSize 5
typedef int ElemType;
typedef struct{
    ElemType data[MaxSize];//数组,存储MaxSize-1个元素
    int front,rear;//队列头 队列尾
}SqQueue;

void InitQueue(SqQueue &Q)
{
    Q.rear=Q.front=0;
}
//判空
bool isEmpty(SqQueue &Q)
{
    if(Q.rear==Q.front)//不需要为零
        return true;
    else
```

```
        return false;
}
//入队
bool EnQueue(SqQueue &Q,ElemType x)
{
    if((Q.rear+1)%MaxSize==Q.front) //判断是否队满
        return false;
    Q.data[Q.rear]=x;//3 4 5 6
    Q.rear=(Q.rear+1)%MaxSize;
    return true;
}
//出队
bool DeQueue(SqQueue &Q,ElemType &x)
{
    if(Q.rear==Q.front)
        return false;
    x=Q.data[Q.front];//先进先出
    Q.front=(Q.front+1)%MaxSize;
    return true;
}
//《王道C督学营》课程
//王道数据结构 3.2 循环队列
int main()
{
    SqQueue Q;
    bool ret;//存储返回值
    ElemType element;//存储出队元素
    InitQueue(Q);
    ret=isEmpty(Q);
    if(ret)
    {
        printf("队列为空\n");
    }else{
        printf("队列不为空\n");
    }
    EnQueue(Q,3);
    EnQueue(Q,4);
    EnQueue(Q,5);
    ret=EnQueue(Q,6);
    ret=EnQueue(Q,7);
    if(ret)
    {
        printf("入队成功\n");
    }else{
        printf("入队失败\n");
    }
    ret=DeQueue(Q,element);
    if(ret)
    {
        printf("出队成功,元素值为 %d\n",element);
    }else{
        printf("出队失败\n");
    }
    ret=DeQueue(Q,element);
    if(ret)
    {
        printf("出队成功,元素值为 %d\n",element);
    }else{
```

```
        printf("出队失败\n");
    }
    ret=EnQueue(Q,8);
    if(ret)
    {
        printf("入队成功\n");
    }else{
        printf("入队失败\n");
    }
    system("pause");
}
```