

Lecture 8: 队列基本概念

8.1 队列的定义

定义: #队列

描述: 只允许一端进行插入, 在另一端删除的线性表;

解释

- 特点: 先进先出
- 重要术语:
 - 队头 (Front)
 - 队尾 (Rear)
 - 空队列

基本操作

- `InitQueue(&Q)`: 初始化队列, 构造一个空队列 `Q`
- `DestroyQueue(&Q)`: 销毁队列。销毁并释放队列 `Q` 所占用的内存空间
- `EnQueue(&Q, x)`: 入队, 若队列 `Q` 未滿, 将 `x` 加入, 使之成为新的队尾
- `DeQueue(&Q, &x)`: 出队, 若队列 `Q` 非空, 删除队头元素, 并用 `x` 返回
- `GetHead(Q, &x)`: 读队头元素, 若队列 `Q` 非空, 则将对头元素赋值给 `x`

总结

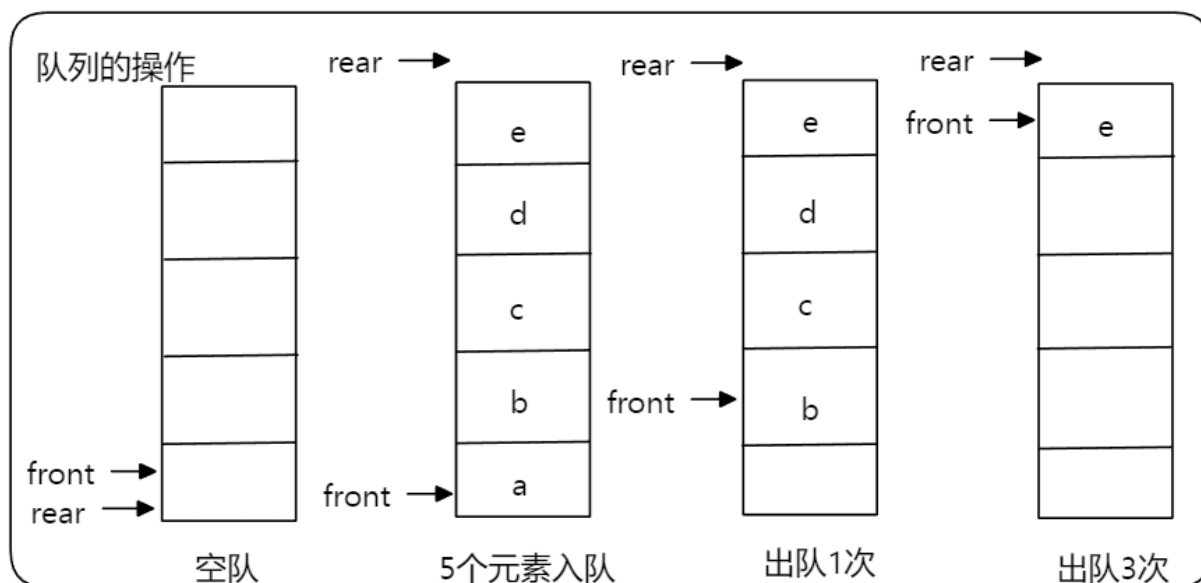
- 创, 销: `InitQueue` `DestroyQueue`
- 增, 删: `EnQueue` `DeQueue`
- 查: `GetHead`
- `QueueEmpty (Q)`: 判断队列是否空, 若队列 `Q` 为空返回 `true`, 否则返回 `false`

8.2 队列的顺序实现

8.2.1 队列的顺序存储

基本概念

- 图示



- 初始时:
 - `Q.front = Q.rear = 0`
- 进队操作:
 - 队不满时, 先送值到队尾元素, 再将队尾指针加 1;
- 出队操作:
 - 队不空时, 先取队头元素值, 再将队头指针加 1;

8.2.2 代码定义

代码定义

```
#define MaxSize 10    //定义队列中元素的最大个数
typedef struct{
    ElemType data[MaxSize]; //用静态数组存放队列元素
    int front, rear;        //队头指针和队尾指针
}SqQueue;
```

初始化

```
//初始化队列
void InitQueue(SqQueue &Q){
    //初始时, 队头, 队尾指针指向0
    Q.rear = Q.front = 0;
}
```

判空

- 空: `Q.rear == Q.front`

```
//判断队列是否为空
bool QueueEmpty(SqQueue Q){
    if(Q.rear == Q.front) //队空条件
        return true;
    else
```

```
        return false;
    }
}
```

8.2.3 基本操作

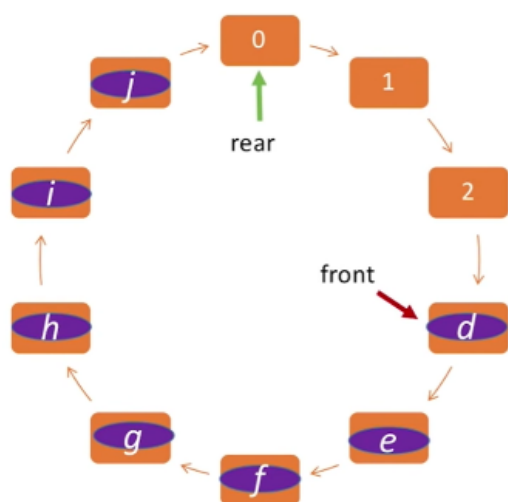
入队

```
//入队
bool EnQueue(SqQueue &Q, ElemType x){
    if(队列已满)
        return false;    //队满则报错
    Q.data[Q.rear]=x;    //将x插入队尾
    Q.rear = Q.rear + 1; //队尾指针后移
    return true;
}
```

8.2.4 循环队列

基本概念

- 图示:



- 核心操作:
- 入队: $Q.rear = (Q.rear + 1) \% MaxSize;$
- 出队: $Q.front = (Q.front + 1) \% MaxSize;$
- 长度: $(rear + MaxSize - front) \% MaxSize;$
- 队满: $(Q.rear + 1) \% MaxSize == Q.front;$

定义: 循环队列中的元素入队操作

```
//循环队列
Q.data[Q.rear] = x;    //新元素插入队尾
Q.rear = (Q.rear+1)%MaxSize; //队尾指针加1取模
```

入队

```
//循环队列的入队操作
bool EnQueue(SqQueue &Q, ElemType x){
    if((Q.rear+1)%MaxSize == Q.front) //根据rear的后一个位置判断
        return false; //队满则报错
    Q.data[Q.rear]=x; //将x插入队尾
    Q.rear = (Q.rear + 1)%MaxSize; //队尾指针后移
    return true;
}
```

- 注意: 判断队列空的存储条件是 `rear==front` , 所以要牺牲一个存储单元;

出队

```
//出队(删除一个队头元素, 并用x返回)
bool DeQueue(SqQueue &Q, ElemType &x){
    if(Q.rear == Q.front) //判断队空
        return false; //队空报错
    x=Q.data[Q.front];
    Q.front = (Q.front + 1)%MaxSize; //队头指针后移
    return true;
}
```

获取队头元素

```
//获取对头元素的值, 用x返回
bool GetHead(SqQueue Q, ElemType &x){
    //队空报错
    if(Q.rear == Q.front)
        return false;
    x = Q.data[Q.front];
    return true;
}
```

确定队列中数据元素个数的方法

- $(rear + MaxSize - front) \text{Mod} \cdot MaxSize$
- 加上 `MaxSize` 的意义: 确保数据一定可以取到正数, 不会出现因为 `rear` 已经走完了一圈, 在 `front` 前面时, 出现的计算值小于 0 情况;

确定判空, 判满的三种方法

1. 牺牲一个存储单元 (如上)
2. 增加 `size` 变量记录队列长度 (待补充)
3. 增加 `tag = 0/1` 用于标记最近的一次操作是出队/入队

```
const int MAXSIZE = 10;
typedef struct
{
    ElemType data[MAXSIZE];
```

```

int front,rear;
int size; //多设置一个字段，表示队列的长度，size=0是为空，size=maxsize是为满
//也可以使用一个bool的值：比如 1为满，0为空。1为每次成功的插入操作，0为每次成功的删除操作
}

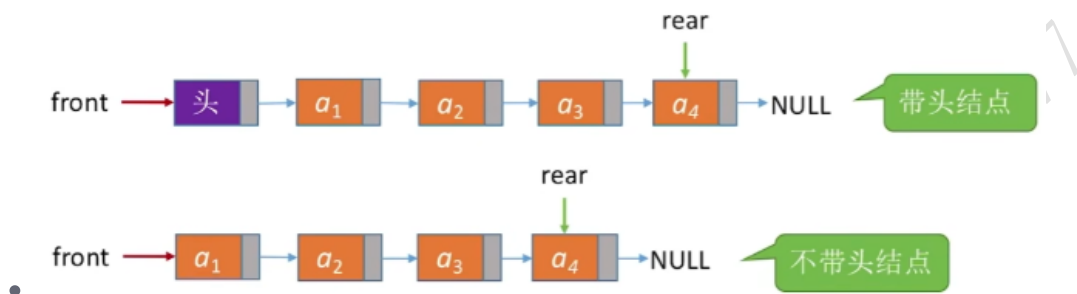
```

8.3 队列的链式实现

8.3.1 定义

基本概念

- 图示：队头朝队尾看



- 其中：链头是队头

代码定义

```

//队列的定义
typedef struct LinkNode{    //链式队列结点
    ElemType data;
    struct LinkNode *next;
}LinkNode;

typedef struct{             //链式队列
    LinkNode *front,*rear;  //队列的队头和队尾指针
}*LinkQueue;

```

初始化：带头结点

```

//初始化队列(带头结点)
void InitQueue(LinkQueue &Q){
    //初始时，front rear 都指向头结点
    Q.front = Q.rear = (LinkNode*)malloc(sizeof(LinkNode));
    Q.front->next=NULL;
}

//判断队列是否为空
bool IsEmpty(LinkQueue Q){
    if(Q.front == Q.rear)
        return true;
    else

```

```

        return false;
    }
}

```

初始化：不带头结点

```

//初始化队列(不带头结点)
void InitQueue(LinkQueue &Q){
    //初始时 front,rear 都指向NULL
    Q.front = NULL;
    Q.rear = NULL;
}

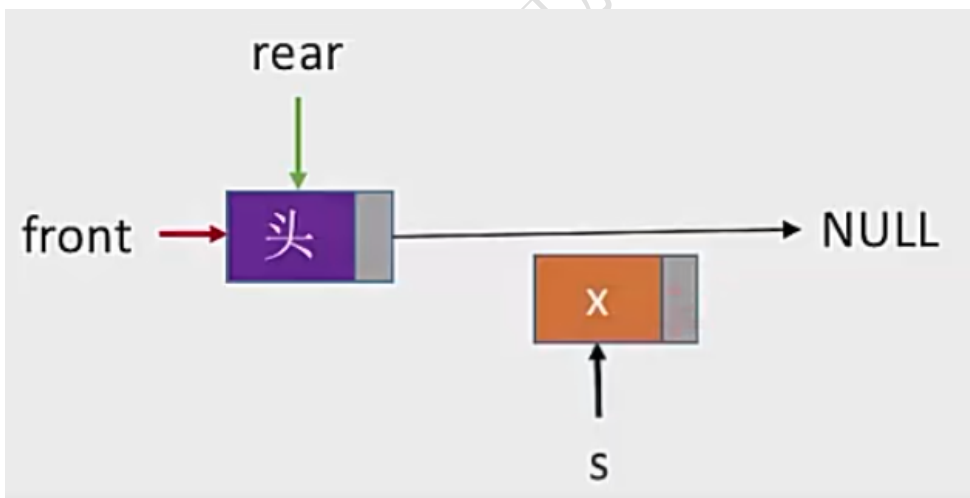
//判断队列是否为空(不带头结点)
bool IsEmpty(LinkQueue Q){
    if(Q.front == NULL) //也可以Q.rear == NULL
        return true;
    else
        return false;
}

```

8.3.2 基本操作

入队：带头结点

- 图示：



- 代码：

```

//新元素入队(带头结点)
void EnQueue(LinkQueue &Q, ElemType x){
    LinkNode *s = (LinkNode *)malloc(sizeof(LinkNode));
    s->data=x;
    s->next=NULL;
    Q.rear->next = s; //新结点插入到rear所指向的节点之后
    Q.rear=s;         //修改表尾指针为当前新插入的节点
}

```

入队：不带头结点

```

//新元素入队(不带头结点)
void EnQueue(LinkQueue &Q, ElemType x){
    LinkNode *s = (LinkNode *)malloc(sizeof(LinkNode ));
    s->data=x;
    s->next=NULL;

    //在空队列中插入第一个元素
    if(Q.front == NULL)
    {
        Q.front = s;    //修改对头队尾指针
        Q.rear = s;
    }
    else
    {
        Q.rear->next = s; //新结点插入到rear之后
        Q.rear=s;        //修改rear指针
    }
}

```

出队：带头节点

- 注意空队列状态: `Q.front == Q.rear`

```

//队头元素出队(不带头结点)
bool DeQueue(LinkQueue &Q, ElemType &x){
    if(Q.front == Q.rear)
        return false;    //空队
    LinkNode *p=Q.front->next; //因为带头节点，所以front不是当前队头元素，得使用
    Q.front->next得到;
    x=p->data;              //用变量x返回对头元素
    Q.front->next=p->next; //修改头结点的next指针，将其指向当前头元素的下一个元素，从而
    删除掉当前头元素（头元素不等于头节点）
    if(Q.rear == p)        //此次是最后一个结点出队
        Q.rear=Q.front;   //修改rear指针
    free(p);               //释放结点空间
    return true;
}

```

出队：不带头节点

- 注意空队列状态: `Q.front == NULL` 或者 `Q.rear == NULL`

```

//队头元素出队(不带头结点)
bool DeQueue(LinkQueue &Q, ElemType &x){
    if(Q.front == NULL)
        return false;
    LinkNode *p=Q.front; //P指向此次出队的结点
    x=p->data;            //用变量x返回对头元素
    Q.front=p->next;      //修改头结点的front指针
    if(Q.rear == p){      //此次是最后一个结点出队

```

```

    Q.front = NULL;    //front指向NULL
    Q.rear = NULL;     //rear指向NULL
}
free(p);
return true;
}

```

- 注意：链式存储一般是不会满的，而顺序存储是静态的会有存满**的情况；

8.4 双端队列

基本概念

- 双端队列
 - 可以从两端队列进行插入与删除；
 - 输入受限：只允许一端插入、两端删除；
 - 输出受限：只允许两端插入、一端删除；
- 图示：



- 考点：主要是已知输入队列，求输出队列；

分析：合法输入、输出序列

- 什么是输入序列：
 - 举例：输入顺序为 1234
 - 无论输出顺序如何，输入的顺序一定 1、2、3、4
- 什么是输出序列：
 - 举例：输出序列为 4321
 - 无论何时输入，输出的顺序一定是 4321
- 什么是输入受限的双端序列：
 - 图示：



- 比如输入为 1234 时，非法输出的序列：4213 和 4231

获取笔记持续更新可加微信：WandD2277