

数据结构

Data Structure

2017年秋季学期
刘鹏远

线性结构之 栈 队列

栈

队列

栈与队列的应用

栈与递归

栈(Stack)

栈

概念

ADT定义

顺序存储结构(顺序栈)

操作实现

链式存储结构(链栈)

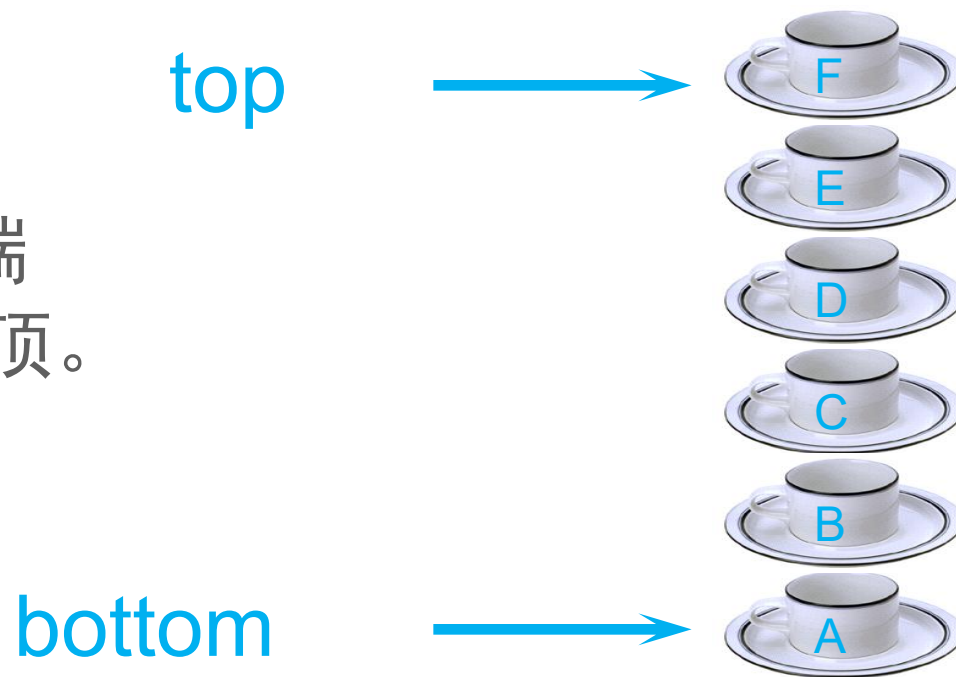
操作实现

概念

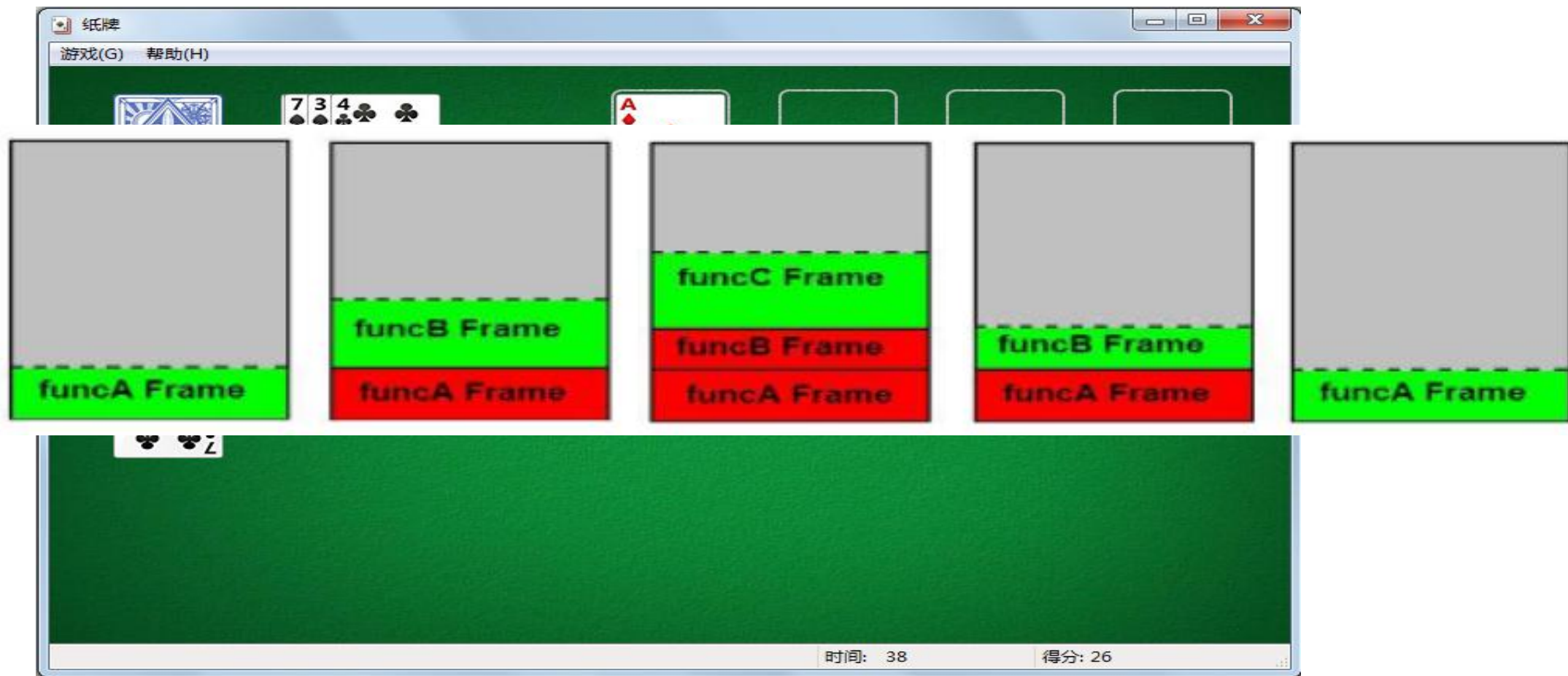
- 栈(Stack): 是限制在表的一端进行插入和删除操作的线性表。
- 逻辑结构是线性结构
- 对特定操作进行限定，其他方面均与线性表类似，因此称为限定性线性表

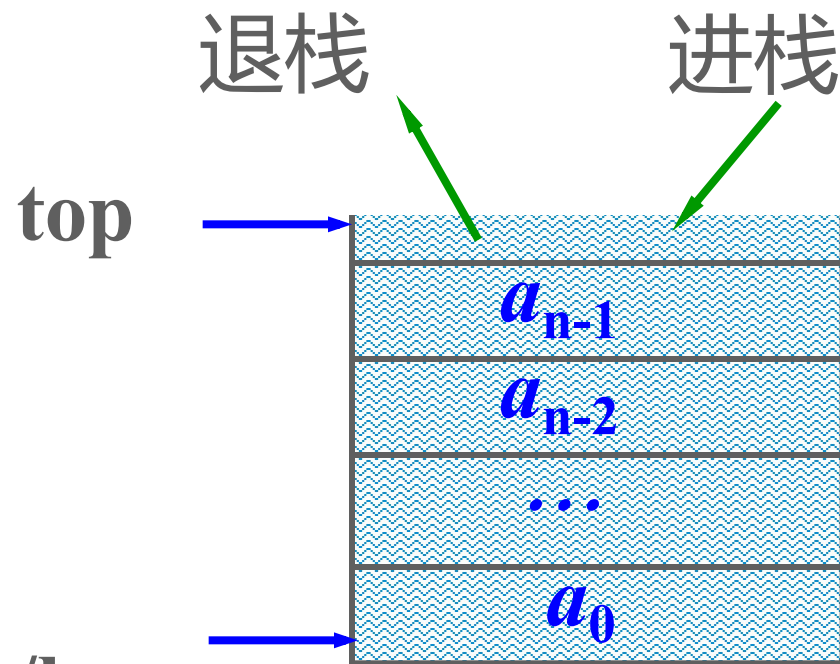
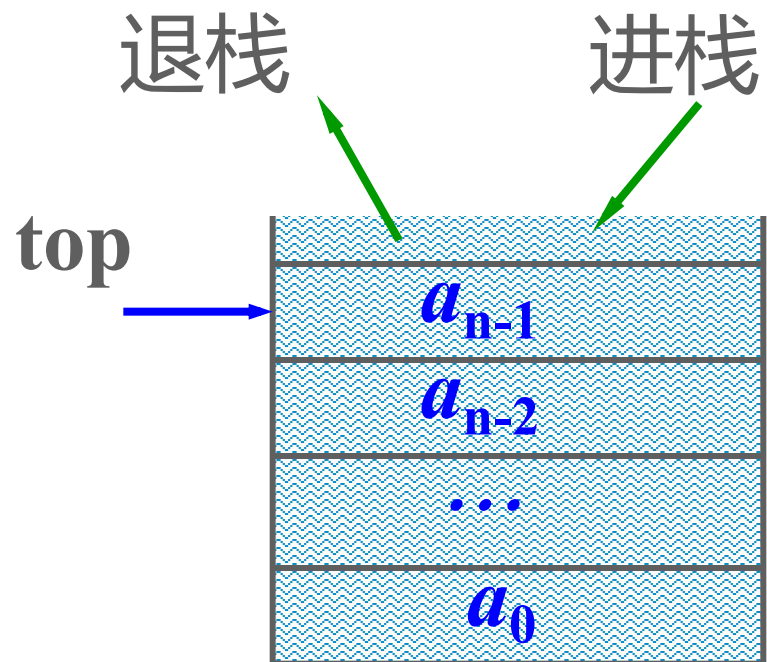
- 栈顶(Top):
允许进行插入、删除操作的一端
用栈顶指针/游标(top)来指示栈顶。

- 栈底(Bottom):
是固定端。



- 特点：后进先出LIFO (Last In First Out)/先进后出FILO (First In Last Out)





bottom/base

bottom/base

操作要实现push入（进）栈与pop出（退）栈，代替线性表中的insert与delete两个操作。push与pop其实也可以视为是特殊/限定的insert与delete操作。

互动：
请同学们讲几个现实中可
以形式化为栈的例子

栈的长度： 栈中元素个数

空栈： 栈的长度为0

栈顶元素： 栈中第一个元素

入(进)栈： 向栈中加入元素

出(退)栈： 从栈中删除元素

ADT定义

ADT Stack

{

数据对象: $D = \{ a_i | a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系: $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

初始化、入栈、出栈、清空、

判断栈是否空、栈长度、取栈顶元素

...

}// 参考教材

顺序存储结构

顺序存储

(动态数组)

```
typedef struct
{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sq_stack; //教材
```

(静态数组) :

```
typedef struct
{
    ElemType data[MAXSIZE];
    int top;
}sq_stack;
```

顺序存储结构

思考：顺序栈，数组两端应设哪端为栈顶哪端为栈底？

//操作

Status InitSqStack(SqStack*);

Status Pop(SqStack*,ElemType*);

Status Push(SqStack*,ElemType);

Status Clear(SqStack*);

Status Destory(SqStack*);

Status IsEmptySq(SqStack);

int Length(SqStack);

Status GetTop(SqStack,ElemType*);

操作实现

```
Status InitSqStack(SqStack *S){  
    S->base = (ElemType *)  
        malloc(sizeof(ElemType)*INIT_SIZE);  
    S->top = S->base;  
    S->stacksize = INIT_SIZE;  
    return OK;}
```

//该方式栈空为 $top = base$ ，栈满为 $top - base \geq stacksize$
//若采用静态数组顺序表，栈满为 $top \geq MAXSIZE$ 或
 $MAXSIZE - 1$ 。（初始 top 与 $base$ 指向0或-1）

操作实现

```
Status Push(SqStack *S, ElemType e){
```

```
    检测入栈条件（判断栈容量）
```

```
    *(S->top) = e;
```

```
    S->top++;
```

```
    return OK;
```

```
}O(1)  数组尾部插入
```

//画图，看栈顶元素与top指针的关系（重要）

操作实现

//入栈条件，判断是否栈满

```
if(S->top - S->base >= S->stacksize){  
    S->base = (ElemType *)realloc(S->base,  
        sizeof(ElemType)*(S->stacksize+INCREMENT));  
    if(!S->base)    exit(OVERFLOW);  
  
    S->top = S->base + S->stacksize;  
    S->stacksize += INCREMENT;  
}
```


操作实现

```
Status Pop(SqStack *S,  
ElemType *e)
```

```
{先检测出栈条件
```

```
    S->top--;
```

```
    *e = *(S->top);
```

```
    return OK;
```

```
}//O(1)
```

```
Status GetTop(SqStack S,  
ElemType *e)
```

```
{先检测边界条件
```

```
    *e = *(S.top-1);
```

```
    return OK;
```

```
}O (1)
```

出栈，看栈顶两个操作，出栈/边界条件为判断是否为空：

```
if(S.top == S.base)    return ERROR;
```

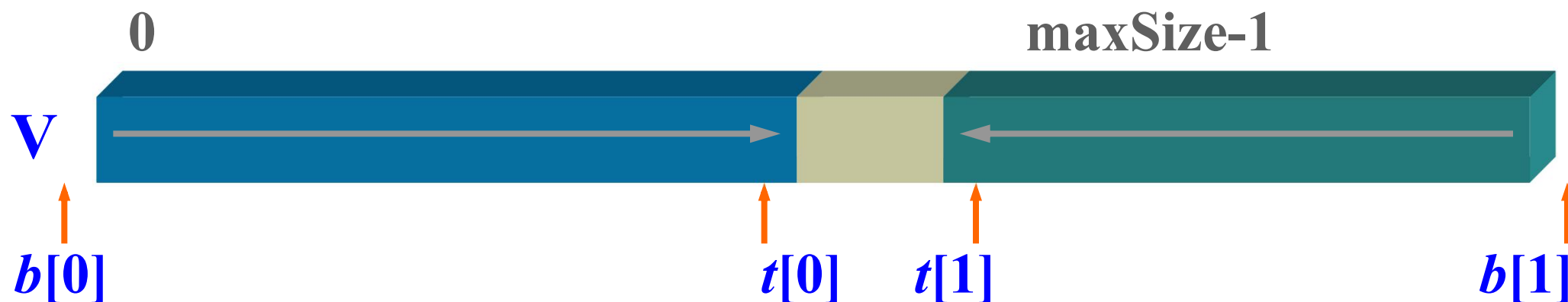
操作实现

数组描述缺陷：空间利用率低。

如同时有两个栈，可能均有不少未利用的存储空间。

优化：双栈共享一个栈空间
如何做？大家画图

操作实现



两个栈共享一个数组空间 $V[\text{maxSize}]$

设立栈顶指针数组 $t[2]$ 和栈底指针数组 $b[2]$

$t[i]$ 和 $b[i]$ 分别指示第 i 个栈的栈顶与栈底

初始 $t[0] = b[0] = -1$, $t[1] = b[1] = \text{maxSize}$

栈满条件: $t[0] + 1 == t[1]$ // 栈顶指针相遇

栈空条件: $t[0] = b[0]$ 或 $t[1] = b[1]$ // 退到栈底

思考：可否基于顺序表基本操作来实现顺序栈？如果可以，如何做？

链式存储结构

利用单链表

思考：top应该设置在单链表的哪端？

链式存储结构

TOP设为尾元素节点

Push(x)——Insert(L,n, x): $O(n)$

Pop(x)——Delete(L,n, x): $O(n)$

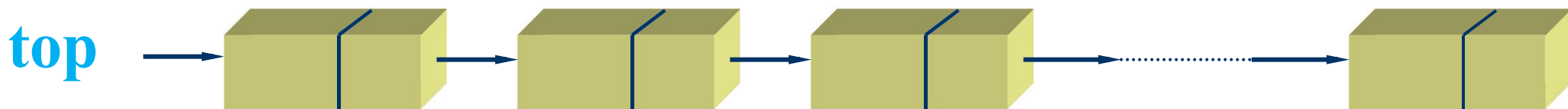
TOP设为首元素节点

Push(x)——Insert(L,1, x): $O(1)$

Pop(x)——Delete(L,1, x): $O(1)$

其实，主要是尾部删除元素麻烦

操作实现



```
typedef struct stack_node
{
    ElemType data ;
    struct stack_node *next ;
} stack_node, *link_stack ;
```

P47(教材省略)

带不带头均可。课堂实现**不带头结点的**，带头结点的自行实现。

```
Status Init ( link_stack* S ) {
    //栈初始化
    //指向NULL或头结点
    return OK;
} //无头的，主函数S=NULL
不可以吗？可以，但用init
主要是为了与其他形式的
各项操作一致。不用考虑
操作的具体实现
```

操作实现

```
Bool is_empty ( link_stack S ) { //判栈空否
    return S == NULL;
}
void Traverse(link_stack S)//遍历
{
    while(S){
        printf("%d\n",S->data);
        S=S->next;
    }
}
```


操作实现

```
void Push ( link_stack* S, ElemType e ) {    //进栈
    stack_node *new_node =
    (stack_node*)malloc(sizeof(stack_node));
    if(!new_node) exit(OVERFLOW);
    new_node->data = e; new_node->next = *S;
    *S=new_node;
    return OK;}
//栈顶指针S每次都变化指向
```

操作实现

```
Status pop (link_stack* S, ElemType* e ) { //退栈
    if(Is_empty(*S)) return ERROR;
    link_stack_node *p = *S; *S = (*S)->next; *e = p->data;
    free(p); return OK;           //释放原栈顶结点}
```

```
Status get_top( link_stack S, ElemType *e) { //看栈顶
    if ( Is_empty(S) ) return ERROR;
    *e = S->data; return OK;}
```

操作实现

其实，在实现单链表基础上：

PUSH 即在位置1插入

POP 即在位置1删除

对带头结点的栈，已经上机实现了带头的单链表，因此自行实现

栈的应用

队列之后再讲

队列 (Queue)

队列(Queue)



○定义

- 队列是只允许在一端删除，在另一端插入的线性表
- 允许删除的一端叫做队头 (**front**)，允许插入的一端叫做队尾 (**rear**)。

○特性

- 先进先出 (**FIFO, First In First Out**)

队列与栈的**共性**在于它们都是限制了**操作位置**的线性表；
区别在于所限定的**操作位置**不同

互动：
请同学们讲几个现实中可
以形式化为队列的例子

概念

入队：队尾插入元素

出队：队头删除元素

队长：队列中元素个数

空队：队长为0

对头元素/队尾元素。其他元素除遍历外，用户一般不可见。

队列的ADT

ADT queue{

数据对象： $D = \{ a_i | a_i \in \text{ElemSet}, i=1, 2, \dots, n, n \geq 0 \}$

数据关系： $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

约定 a_1 端为队首， a_n 端为队尾。 基本操作：

init_queue(): 创建一个空队列；

is_empty(): 若队列为空，则返回true，否则返回false；

... ..

en_queue(x)： 入队---向队尾插入元素x；

de_queue(x)： 出队---删除队首元素x；

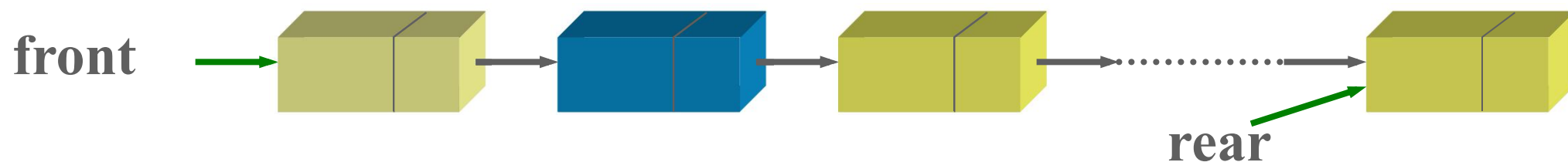
} ADT queue //P59页

存储结构

这回换个顺序，先考虑链式结构

哪端队头，哪端队尾合适？

队列的链式表示 — 链式队列



链式队列一般采用带尾指针的单链表（带不带头均可）存储队列元素，队头在链头，队尾在链尾。（因此插入删除均为 $O(1)$ ）

front指向队头元素，rear指向队尾元素

思考：判空？分有头和无头

队空条件为 $\text{front} == \text{NULL}$ (无头)， $\text{rear} == \text{front}$ (有头)。

链式队列在进队时一般无队满问题。

链式队列的结构定义

```
typedef struct LinkQueNode{    //节点定义
    ElemType data;
    struct LinkQueNode * next;
}LinkQueNode;
```

```
typedef struct LinkQueue{ //用首尾指针结构体来定义队列
    LinkQueNode *front;
    LinkQueNode *rear;
}LinkQueue;
```

链式队列的操作

```
Status InitLinkQueue(LinkQueue *);  
Status EnLinkQueue(LinkQueue*, ElemType);  
Status DeLinkQueue(LinkQueue*, ElemType *);  
Bool Is_empty (LinkQueue);  
Status GetTop(LinkQueue, ElemType *);  
void Traverse(LinkQueue);
```

```
Status InitLinkQueue(LinkQueue *Q){  
    Q->front = Q->rear = NULL;  
    return OK;  
}
```

```
Bool Is_empty(LinkQueue Q)  
{  
    return Q.front == NULL;  
}
```

```
Status get_front ( link_queue Q, ElemType *e )  
{  
    if (is_empty(Q)) return ERROR;  
    *e = Q.front->data;  
    return OK;  
}
```

//各项操作基本可参考不带头结点的单链表

```
Status EnLinkQueue(LinkQueue *Q, ElemType e)
```

```
LinkQueNode *p = (LinkQueNode *)malloc(sizeof(LinkQueNode));
```

```
if(!p) exit(OVERFLOW);
```

```
p->data = e;    p->next = NULL;
```

```
if(Is_empty(*Q))//空队列时
```

```
    Q->rear = Q->front = p;
```

```
else{//非空队列时
```

```
    Q->rear->next = p;
```

```
    Q->rear = p;
```

```
}
```

```
return OK;}
```



```
Status DeLinkQueue(LinkQueue *Q, ElemType *e)
{
    if ( Is_empty(*Q) ) return ERROR;
    LinkQueNode *p = Q->front;
    Q->front = Q->front->next;
    if ( Q->front == NULL ) Q->rear = NULL;
    *e = p->data;
    free(p);
    return OK;
}
```

队列的顺序表示

```
#define MAX_SIZE 50;
```

```
typedef struct {           //顺序队列的结构定义  
    ElemType queue_array[MAX_SIZE]; //队列存储数组  
    int rear, front;        //队尾与队头指针（游标）  
} sq_queue;
```

队列的顺序表示

```
#define MAX_SIZE 50;
```

```
typedef struct {           //顺序队列的结构定义  
    ElemType *elem;        //队列存储数组  
    int rear, front;        //队尾与队头指针  
} sq_queue;
```

两种结构体定义中为什么都没有queue_size 队列长度呢.... ?

一个队首指针front，指向队首

一个队尾指针rear，指向队尾

第一个考虑：

front与rear各处于顺序表的哪端较为合适？

首在前端，尾在后端

第二、出队方式的考虑与选择：

1、指针（游标）法

2、删除(这次是真删除)队首元素

$O(1)$ VS. $O(n)$

假溢出 VS. 无需队首指针，操作方便

第三、入队出队操作方式的考虑与选择：

- 1、先动元素再动指针(主流，按此讲授)
- 2、先动指针再加元素

先动元素再动指针

进队： 1、新元素按 rear 指示位置加入(赋值);
2、队尾指针进一 $\text{rear} = \text{rear} + 1$ 。

队尾指针指示实际队尾的下一位置。

出队： 1、下标为 front 的元素取出（保存）；
2、队头指针进一 $\text{front} = \text{front} + 1$ 。

队头指针指示实际队头的位置。

（元素实际不必要一定去删除）

先动指针再加元素

进队： 1、队尾指针进一 $\text{rear} = \text{rear} + 1$;
2、新元素按 rear 指示位置加入。

队尾指针指示实际队尾的位置。

出队： 1、队头指针进一 $\text{front} = \text{front} + 1$;
2、下标为 front 的元素取出（保存）。

队头指针指示实际队头的前一位置。

考虑入队出队下front、rear的变化：

1、初始化：front=rear=0。

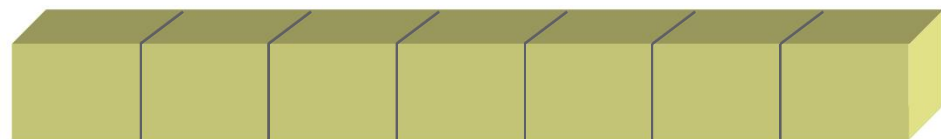
2、入队：将新元素插入rear所指的位置，rear++

3、出队：取出front所指的元素，front++

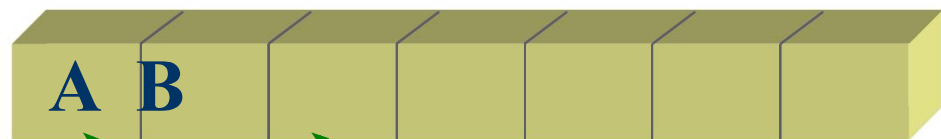
根据front、rear进行如下判断：

1、队列为空：front=rear

2、队满：rear>=MAX_SIZE



front rear 空队列



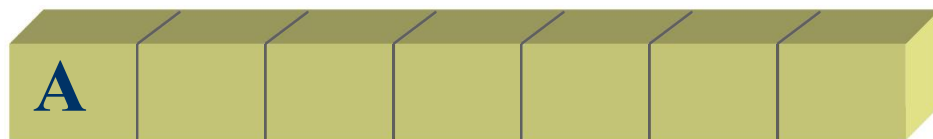
front rear B 进队



front rear A 退队



front rear E,F,G 进队



front rear A 进队



front rear C,D 进队



front rear B 退队



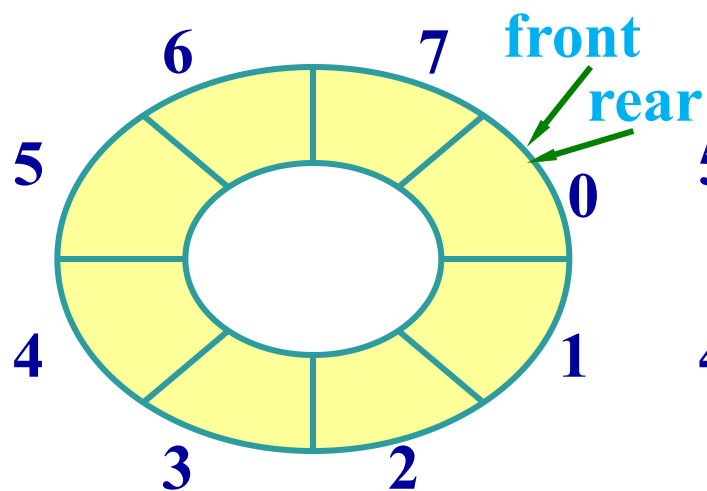
front rear H 进队,溢出

细心同学应该注意到了：front, rear总是++

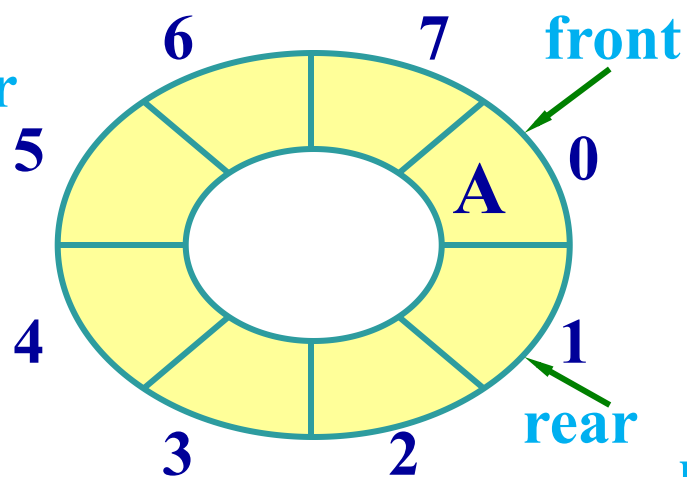
结果是造成：假溢出

顺序队列一般多为假溢出

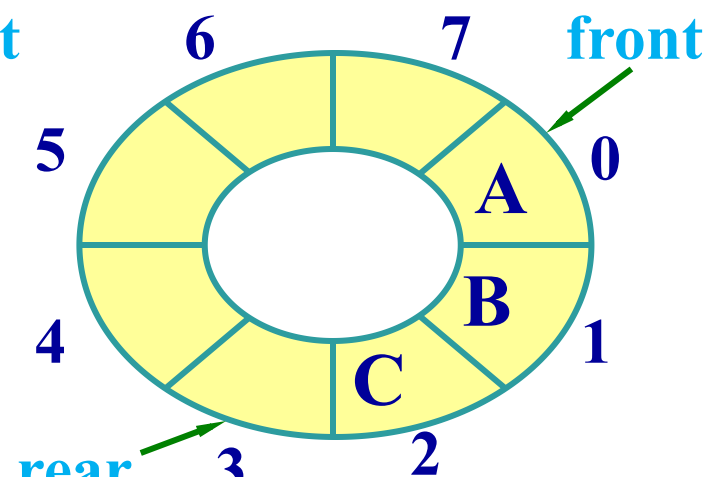
解决办法？



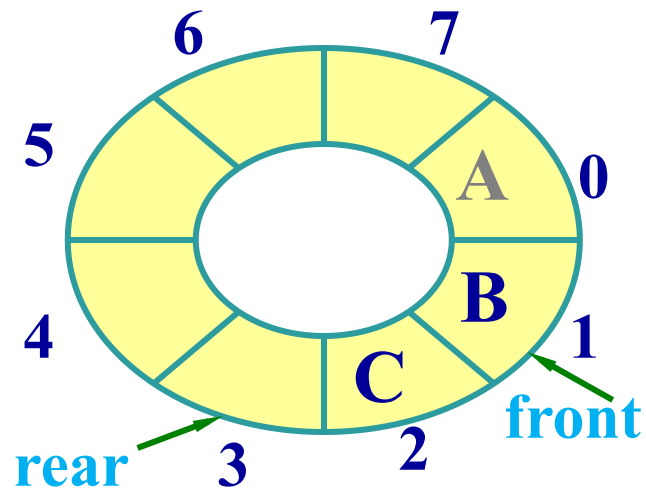
空队列



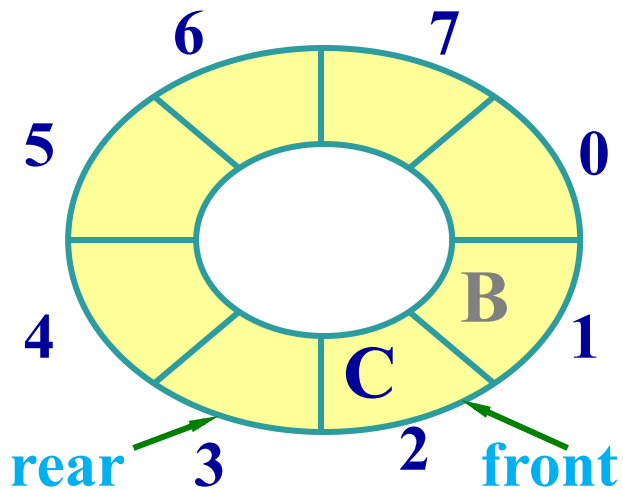
A 进队



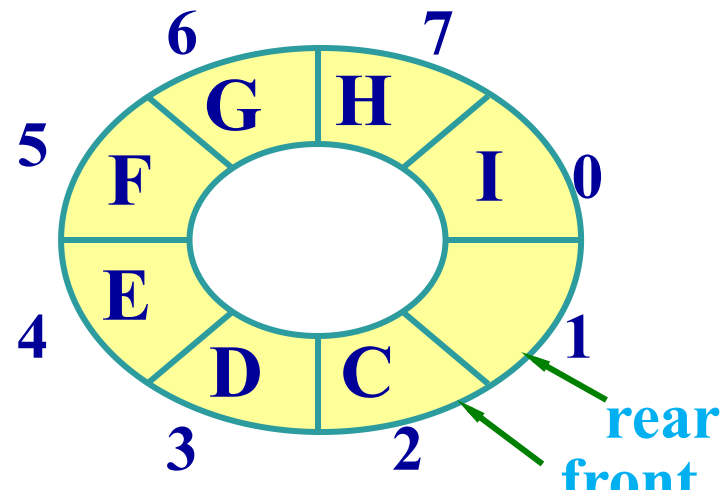
B, C 进队



A 出队



B 出队



D, E, F, G, H, I 进队

循环队列(Circular Queue)

队列存放数组被当作首尾相接的环形表处理。

队头、队尾指针加 1 时从 `queue_size-1` 直接进到 0，可用取模 (%) 运算实现（类似约瑟夫问题模拟）：

```
if (i+1==queue_size) i=0;  
else    i++ ;
```

其中： i 代表队首指针(front)或队尾指针(rear)

这两行用模运算可简化为：`i=(i+1)%queue_size`；

循环队列(Circular Queue)

队头指针进1: $\text{front} = (\text{front} + 1) \% \text{queue_size};$

队尾指针进1: $\text{rear} = (\text{rear} + 1) \% \text{queue_size};$

队列初始化: $\text{front} = \text{rear} = 0;$

队判空条件: $\text{front} == \text{rear};$

提问: 可否用 front 或 $\text{rear} == 0$ 作为判空条件? And why?

提问: 如何判满? 可否 $\text{front} == \text{rear}$?

队判满条件: $(\text{rear} + 1) \% \text{queue_size} == \text{front};$

注意, 进队和出队时指针都是顺时针前进。

循环队列操作的实现

```
Status init_queue ( sq_queue *Q ) { // 以下均以静态为例
```

```
    Q->rear = Q->front = 0;
```

```
    return OK;
```

```
}
```

```
Bool is_empty ( sq_queue Q ) {
```

```
    return Q.rear == Q.front;
```

```
}
```

```
int is_full ( sq_queue Q ) {  
    return (Q.rear+1) % queue_size == Q.front;  
}
```

//当进队速度快于出队速度，rear追上front，造成队满，为了区分队空条件，取当 $\text{rear}+1 == \text{front}$ ，以牺牲一个存储单元为代价。

```
Status en_queue ( sq_queue* Q, ElemType e ) {  
    if ( is_full(Q) ) return ERROR;  
    Q->queue_array[Q->rear] = e; //先操作元素  
    Q->rear = (Q->rear+1) % queue_size;  
    return OK;  
}
```



```
Status de_queue ( sq_queue* Q, ElemType *e ) {  
    if ( is_empty(Q) ) return ERROR;  
    e = Q->queue_array[Q->front];  
    Q->front = (Q->front+1) % queue_size;  
    return 1;  
}
```

```
Status get_front ( sq_queue Q, ElemType * e ) {  
    if ( is_empty(Q) ) return ERROR;  
    *e = Q.queue_array[Q.front];    return OK;  
}
```

问题：

如果用循环链表来构造循环队列
合理否？

下午上机：

- 1、顺序栈，链栈(不带头结点)，PPT13页标红操作实现
- 2、链队列(不带头结点)，PPT37页操作实现

主程序进行测试：

栈：压入1,3,5,7、看此时栈顶，遍历，弹出一个数字并打印之、看此时栈顶，压入2,4、将所有元素弹出并依次打印。

队列：入队1,3,5,7、看队首，遍历，出队一个数字并打印之、看此时队首，入队2,4、将所有元素出队并依次打印。

作业：1,2及

- 3、顺序队列(循环队列)，PPT37页操作实现