

# 数据结构与算法



主讲教师：刘峤

# 第4章 栈和队列

# 第3章 内容提要

---

∞ **栈的性质**

∞ **栈的实现**

∞ 栈的应用（自学）

∞ **队列的性质**

∞ **队列的实现**

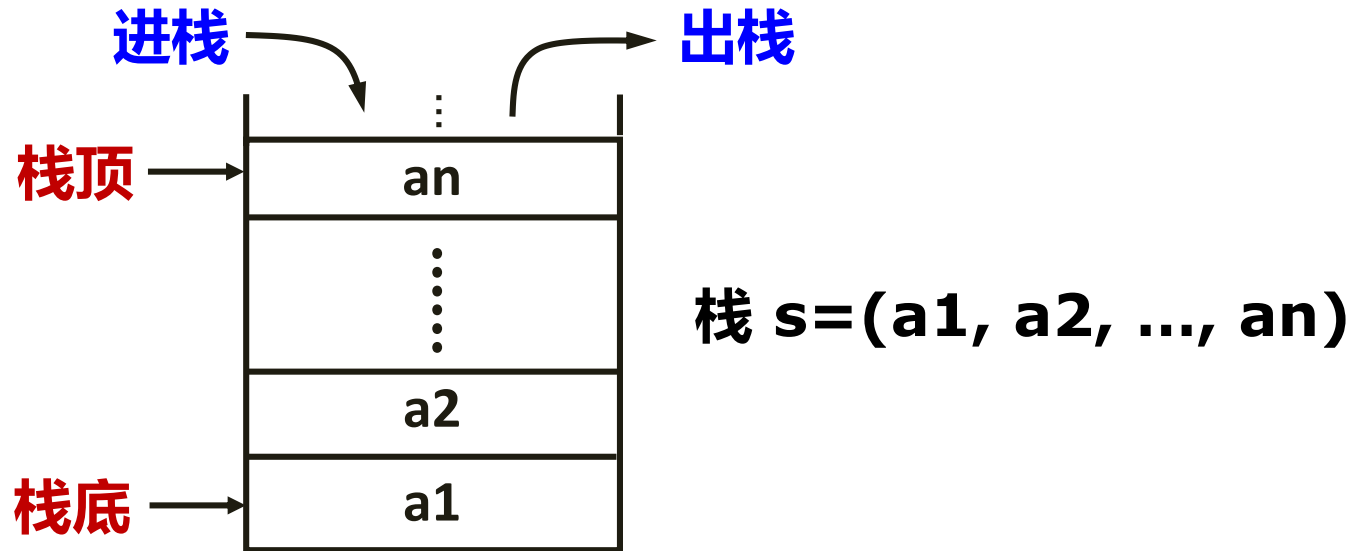
∞ 队列的应用（自学）



**栈**

**(Stack)**

# 栈 (stack)



## 栈的定义和特点

- 定义：限定仅在表的一端进行插入或删除操作的线性表
  - 允许插入和删除的一端称为栈顶，另一端称为栈底
  - 不含元素的空表称空栈
- 特点：后进先出 (**LIFO**) 或 先进后出 (FILO)

# 栈 (stack)

## ❧ 栈的实现方式有两种

- 第一种方式：顺序栈
  - 采用顺序存储结构（顺序表）实现
- 第二种方式：链栈
  - 采用链式存储结构（链表）实现

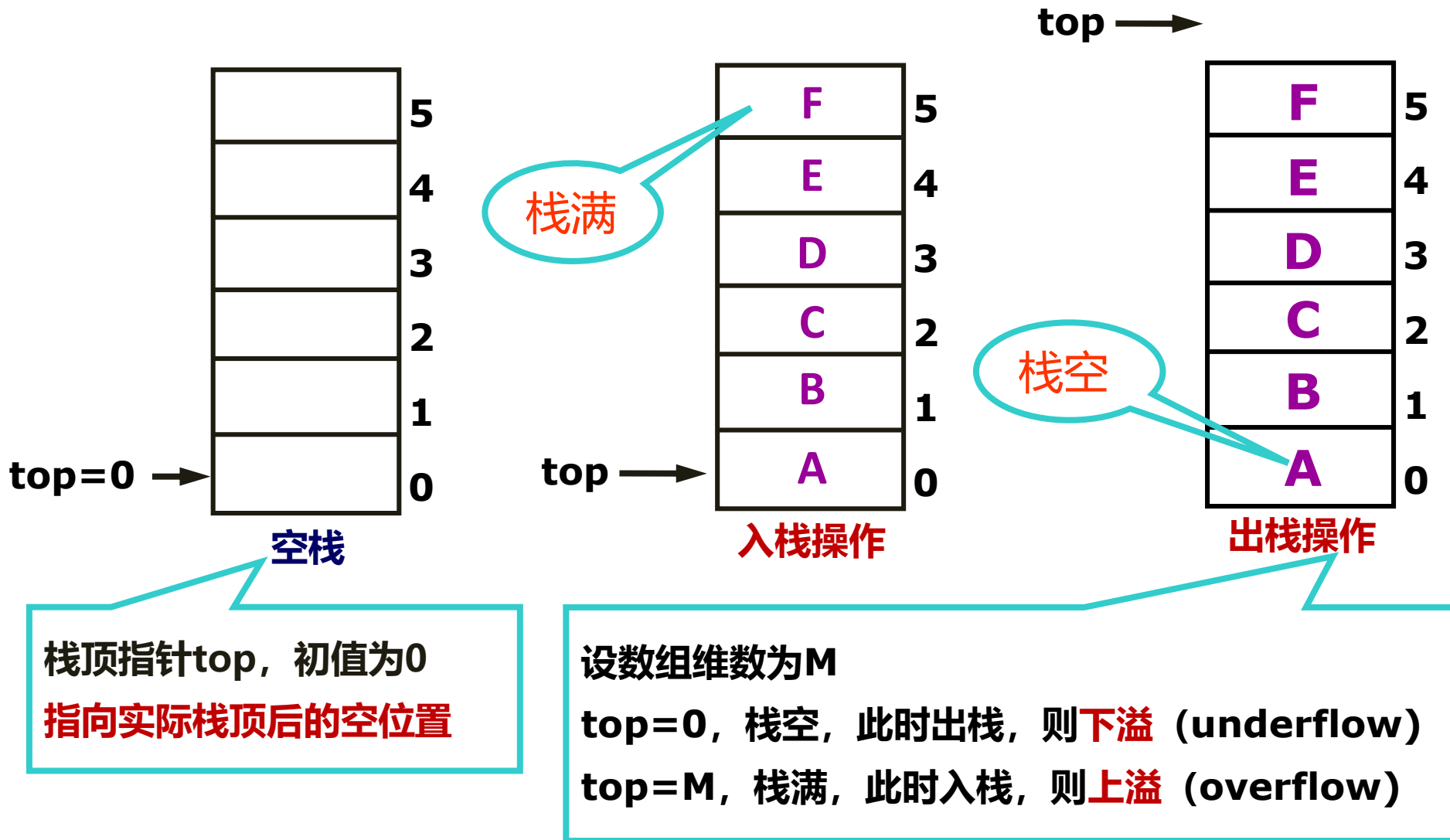
## ❧ 栈的主要操作

- 初始化：init\_stack(S)
- 进栈：push\_stack(S, e)
- 退栈：pop\_stack(S)
- 查看栈顶：top\_stack(S)
- 判栈空，判栈满

# 顺序栈

# 栈的存储结构：顺序栈

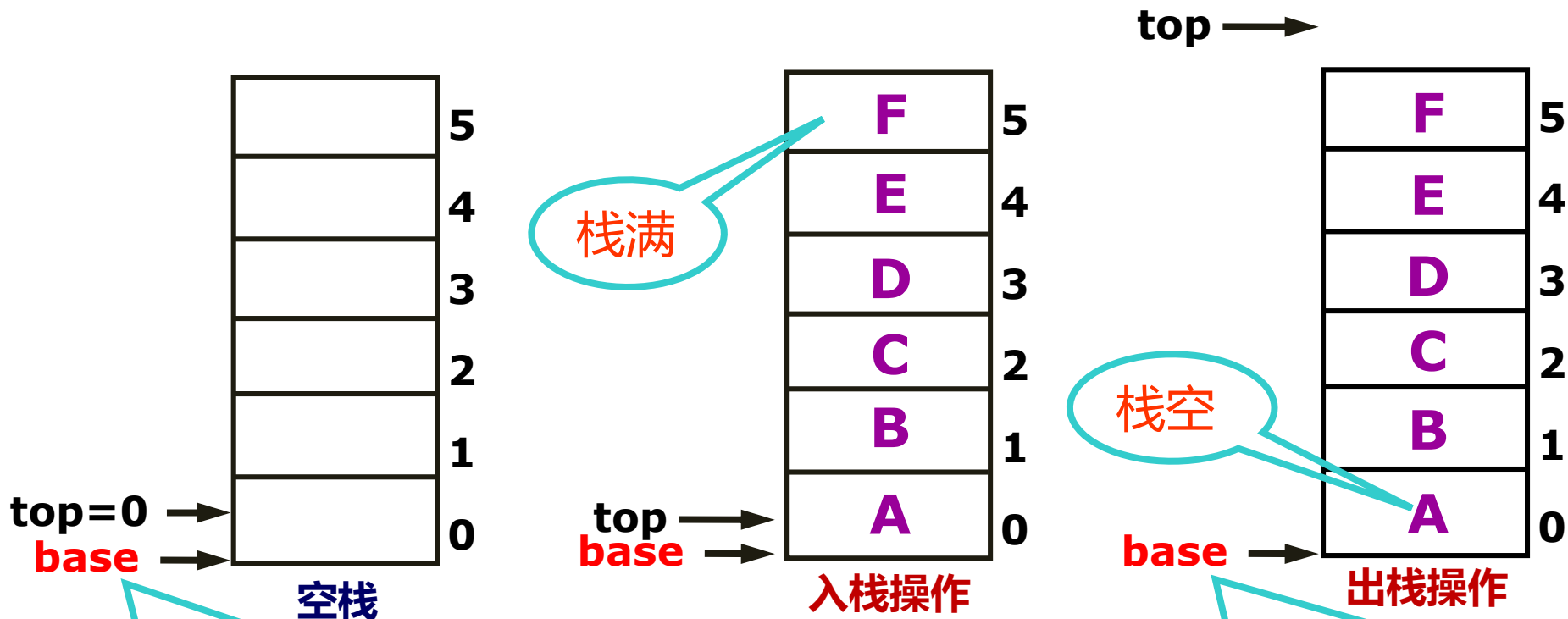
☞ 顺序栈采用一维数组实现





# 栈的存储结构：顺序栈

采用一维数组实现：为了方便处理，设置一个base指针



- 栈底指针： **base**
  - 始终指向栈底的位置
- 栈顶指针： **top**
  - 指向实际栈顶后的空位置
  - 初值为  $\text{top} = \text{base}$

- 设数组维数为  $M$
- $\text{top} - \text{base} == 0$ ，栈空
  - 此时出栈，则下溢 (underflow)
- $\text{top} - \text{base} == M$ ，栈满
  - 此时入栈，则上溢 (overflow)

# 顺序栈的数据结构定义

## ❧ 静态分配存储空间

```
typedef struct stack{  
    int top;                // 存放栈顶位置  
    ElemType data[MAXSIZE]; // 存放数据元素  
}TStack;
```

## ❧ 动态分配存储空间

```
typedef struct stack{  
    int top, len;           // 存放栈顶位置和栈的容量  
    ElemType *pdata;       // 指向数据元素存储空间的指针  
}TStack, *PStack;
```



# 顺序栈的基本操作

## ☞ 栈的初始化：动态分配存储空间

```
PStack init_stack (int n){  
    PStack ps = NULL; ElemType *p; // 指向栈底的指针  
    ps = (PStack) malloc (sizeof(TStack)); // 判空略  
    p = (ElemType*) malloc (sizeof(ElemType)*n);  
    if (p){  
        ps->pdata = p;  
        ps->top = 0;  
        ps->len = n;  
    } else{printf("分配空间失败! \n");}  
    return ps;  
}
```

# 顺序栈的基本操作

∞ 栈的销毁：释放为栈动态分配的存储空间

```
void destroy_stack (PStack ps) {  
    if(ps->pdata) { // 首先判断栈的存储空间是否存在  
        free (ps->pdata); ps->pdata = NULL;  
        ps->len = 0; ps->top = 0;  
    }  
}
```

∞ 栈的清空

```
void clear_stack (PStack ps){  
  
    ps->top = 0;  
  
}
```



# 顺序栈的基本操作

## ❧ 栈的判空函数

```
bool isempty_stack (PStack ps){  
    return (ps -> top <= 0);  
}
```

## ❧ 栈的判满函数

```
bool isfull_stack (PStack ps){  
    return (ps -> top >= ps->len );  
}
```

# 顺序栈的基本操作

☞ 入栈：插入元素item为新的栈顶元素

```
int push_stack(PStack ps, ElemType e) {  
    if( isfull_stack (ps) ){  
        return 1;  
    }  
    else{  
        ps->pdata[ps->top] = e;  
        ps->top++;  
    }  
    return 0;  
}
```

# 顺序栈的基本操作

∞ 出栈：取出栈顶元素，通过函数参数item返回

```
int pop_stack(PStack ps, ElemType *item) {  
    int status = 0;  
    if( isempty_stack(ps) ){  
        status = -1;  
    }  
    else{  
        (ps->top)--;  
        *item = ps->pdata[ps->top];  
    }  
    return status;  
}
```

课后练习

请同学们编程实现

含有base指针的顺序栈

请思考：顺序栈有什么缺陷，如何解决？



# 顺序栈小结

∞ 顺序栈的实现可以采用静态分配或动态分配

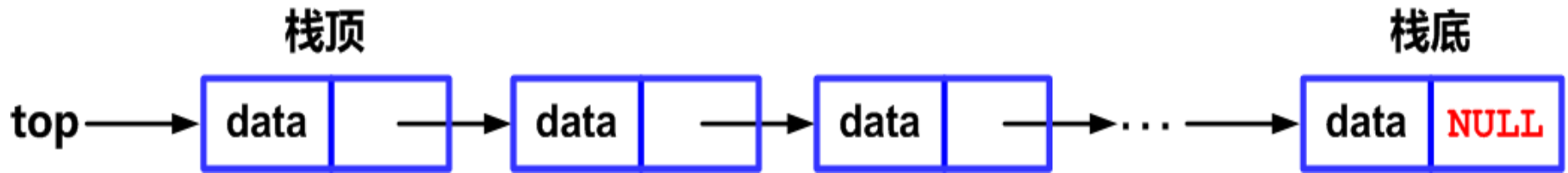
∞ **请思考：顺序栈有什么缺点，如何解决？**

- 顺序栈需要预先分配空间
  - 如果数组预分配空间太小，容易造成溢出
  - 如果数组预分配空间太大，又会浪费空间
- 解决办法：可用单链表来实现栈



# 链栈

# 链栈的数据结构



## 链栈结点的定义

```
typedef struct node {  
    ElemType data;  
    struct node *next;  
} TNode, *PStack;
```



# 链栈的基本操作

## ❧ 链栈的初始化操作

```
PStack ps = NULL; // 声明一个指向栈顶的指针
```

## ❧ 链栈的判空操作

```
bool isempty_stack(PStack ps){  
    return ps == NULL;  
}
```

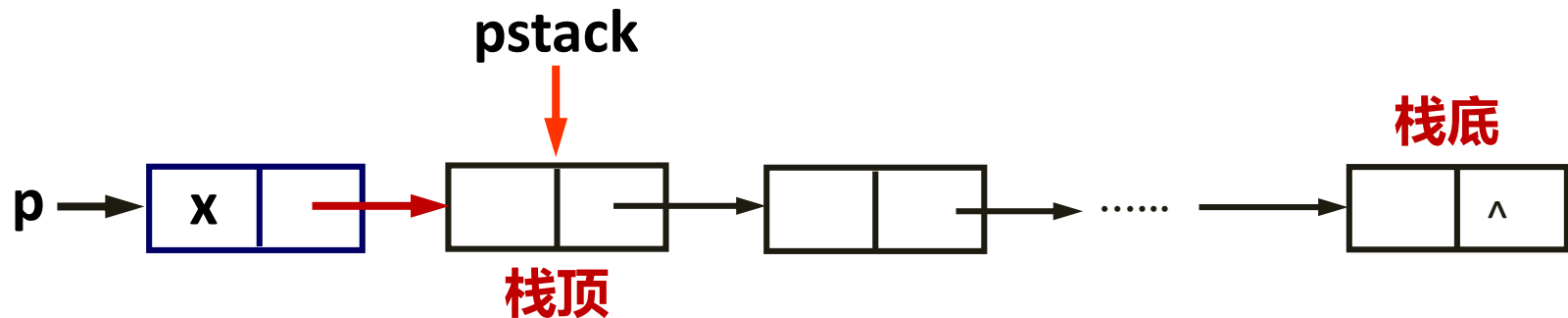
# 链栈的基本操作

## ☞ 链栈的清空操作（即：销毁操作）

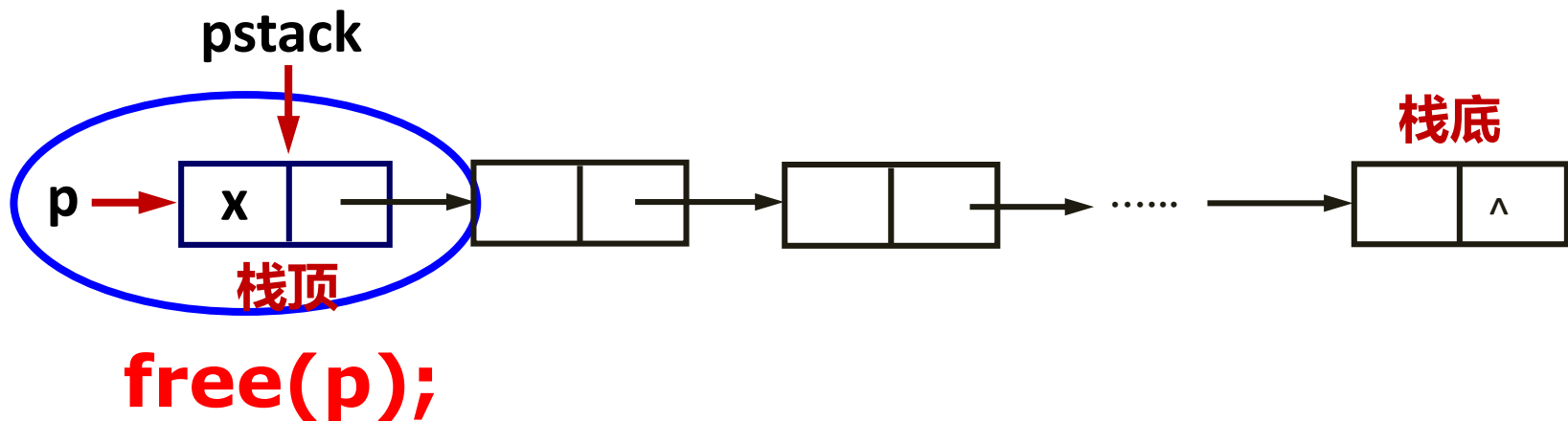
```
void clear_stack(PStack *pps) {  
    if(!(*pps)) return;  
    PStack p = *pps; PStack ps = p;  
    while (ps){  
        ps = ps->next;  
        free(p);  
        p = ps;  
    }  
    *pps = NULL;  
}
```

# 链栈的基本操作

## ❧ 入栈算法



## ❧ 出栈算法



# 链栈的基本操作

❧ 链栈的入栈操作：插入元素item为新的栈顶元素

```
void push_stack(PStack *pps, ElemType e){  
    PStack p = (PStack)malloc(sizeof(TNode));  
    if (p){  
        p->data = e; p->next = NULL;  
    } else{  
        printf("动态分配内存失败"); exit(0);  
    }  
    p->next = *pps; // 将新结点插入到表头位置  
    *pps = p;       // 修改栈顶指针  
}
```

# 链栈的基本操作

❧ 链栈的出栈操作：弹出栈顶元素(通过指针参数item返回)

```
ElemType pop_stack(PStack *pps) {  
    PStack p; ElemType e;  
    // 注意：出栈前一定要判空，对空栈执行出栈操作会导致下溢  
    if (isempty_stack(*pps)) {  
        printf("栈空! \n"); exit(0);  
    }  
    else {  
        p = *pps;           // p指向栈顶元素  
        *pps = p->next;     // 栈顶指针下移  
        e = p->data;         // 保存当前栈顶元素值  
        free(p);            // 删除当前栈顶元素  
    }  
    return e;  
}
```




# 栈的应用：数制转换

1. 数制转换：十进制数  $N$  与其他  $b$  进制数的转换

- 辗转相除法： $N = (N / b) * b + N \% b$

- 例如： $(1348)_{10} = (2504)_8$ ，运算过程如下：

$N$	$N/8$	$N\%8$
1348	168	4
168	21	0
21	2	5
2	0	2





# 栈的应用：数制转换

// 输入任意非负十进制整数，打印输出等值的b进制数

```
void dec2base(int n, int b) {  
    int d; PStack ps = NULL;  
    while ( n ){    // 辗转相除  
        push_stack (&ps, n%b);  
        n = n / b;  
    }  
    while ( ! isempty_stack (ps) ){  
        d = pop_stack(&ps);  
        printf("%d", digit);  
    }  
    n = (n / b)*b + n % b  
}
```



# 栈的应用：函数的递归调用

❧ 什么是递归 (recursion) ?

- 程序调用自身的编程技巧称为递归
- 例如 $n!$ 的定义就是递归的： $n! = n \times (n - 1)!$

❧ 为什么要用递归?

- 将一个大型的复杂问题转化为
- 一些与原问题相似的规模较小的问题来进行求解

❧ 递归程序的要素?

- 递归调用：问题得到简化
- 程序出口：结束（返回）条件

# 栈的应用：函数的递归调用

编写函数求  $n!$

```
int factorial(int n) {  
    if (n <= 1) } 递归的终止条件  
        return 1;  
    else  
        return n * factorial(n - 1); 函数的递归调用  
}
```

是同一个问题，但问题规模被缩减

为了解递归的工作原理，我们来跟踪 **factorial(4)** 的执行

```

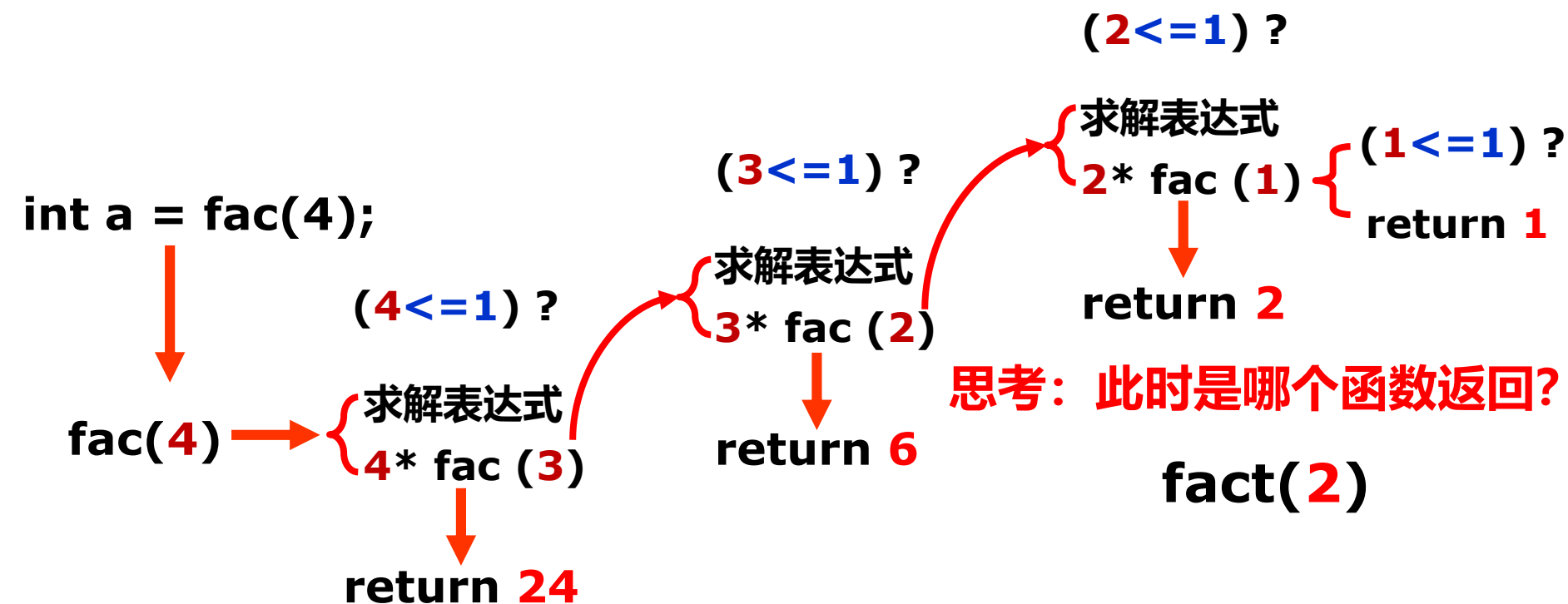
int main(void){
    int a = fact(4);
    printf("fact(4) = %d", a);
    return 0;
}

```

```

int fact (int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact (n - 1);
}

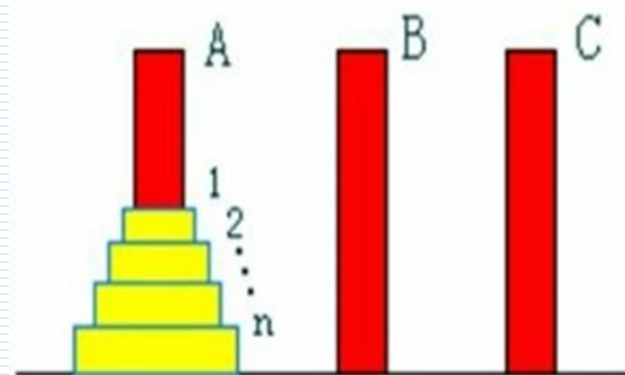
```



# Hanoi Tower (汉诺塔) 问题

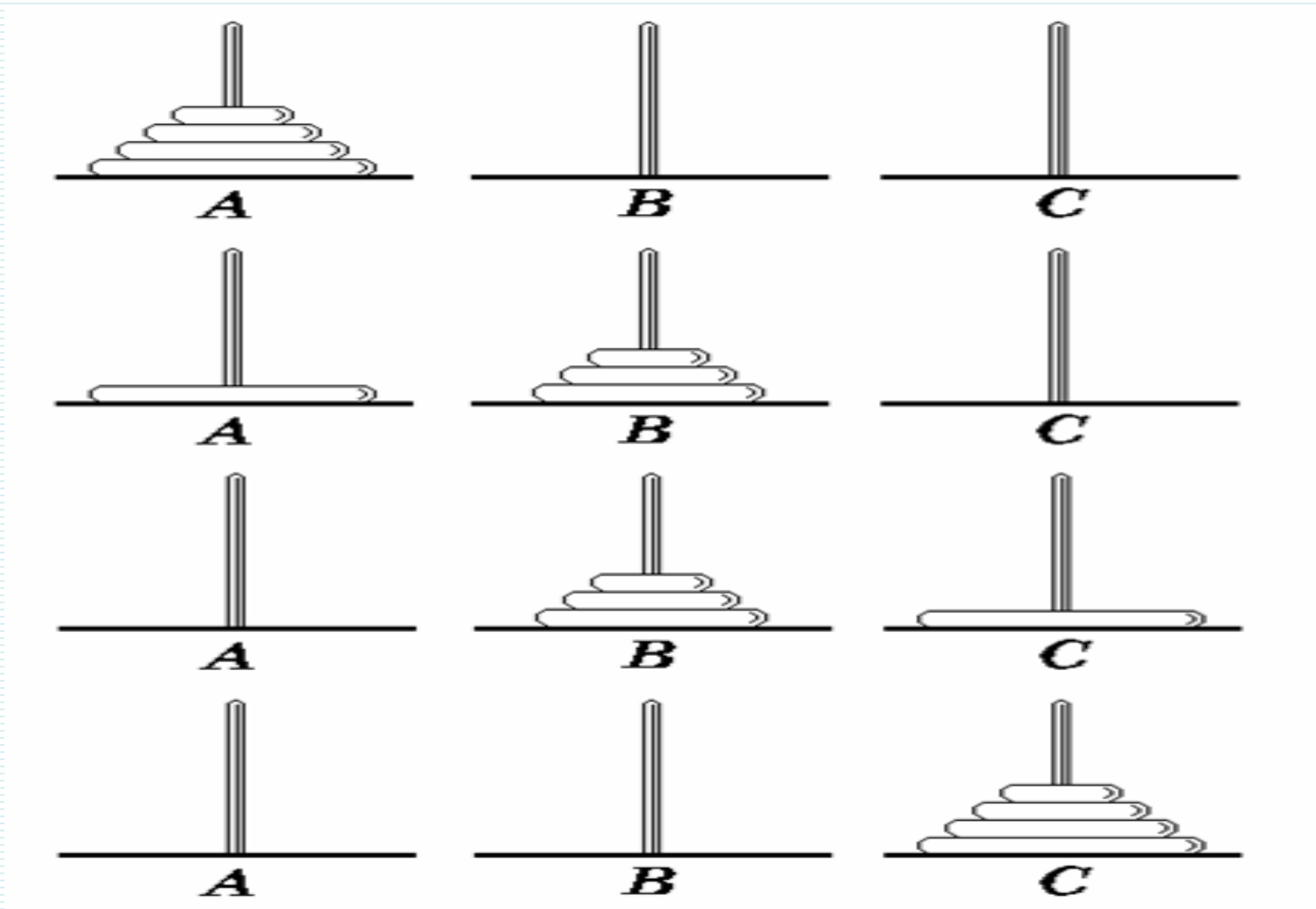
问题描述：设有A、B、C3个塔座

- 在塔座A上有一叠共 $n$ 个圆盘
  - 自上而下，由小到大地叠在一起
  - 自上而下依次编号为 $1, 2, \dots, n$
- 问题：要求将塔座A上的圆盘全部移到塔座C上，仍按同样顺序叠置。在移动圆盘时遵守以下规则：
  - 每次只允许移动1个圆盘
  - 任何时刻都不允许将较大的圆盘压在较小的圆盘之上
  - 在规则1和2的前提下，可将圆盘移至任何一塔座上



# 用递归技术求解汉诺塔问题

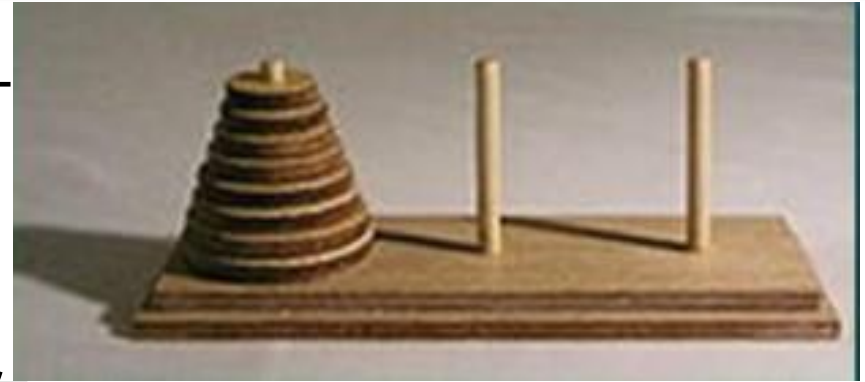
∞ 将4个盘子从A移至C，以B为辅助



# 汉诺塔问题的规模

一般的Hanoi塔玩具不超过8片

- 如果 $n=8$ ，需移动255次
- 如果 $n=10$ ，需移动1023次
- 如果 $n=20$ ，需移动1048575次（超过一百万次）
- 如果 $n=64$ ，需移动 $2^{64}-1$ 次
  - 如果每秒移动一块圆盘，需5845.54亿年
  - 按照宇宙大爆炸理论推测，宇宙的年龄也仅为137亿年



# 用递归技术求解汉诺塔问题

## ∞ 算法设计思路

- 当 $n=1$ 时，问题可以直接求解，一步完成
- 当 $n>1$ 时，分三步完成：
  - 将 $n-1$ 个较小盘子设法移动到辅助塔座
    - 构造出一个比原问题规模小1的问题
  - 将最大的盘子从原塔座一步移至目标塔座
  - 将 $n-1$ 个较小的盘子设法从辅助塔座移至目标塔座
    - 仍然是比原问题规模小1的问题





# 汉诺塔问题的递归算法

```
void hanoi (int  n, int src, int tar, int aux){  
    if(n>0){  
        hanoi(n-1,  src,  aux, tar);  
        move(src, tar);  
        hanoi(n-1,  aux, tar, src);  
    }  
}
```

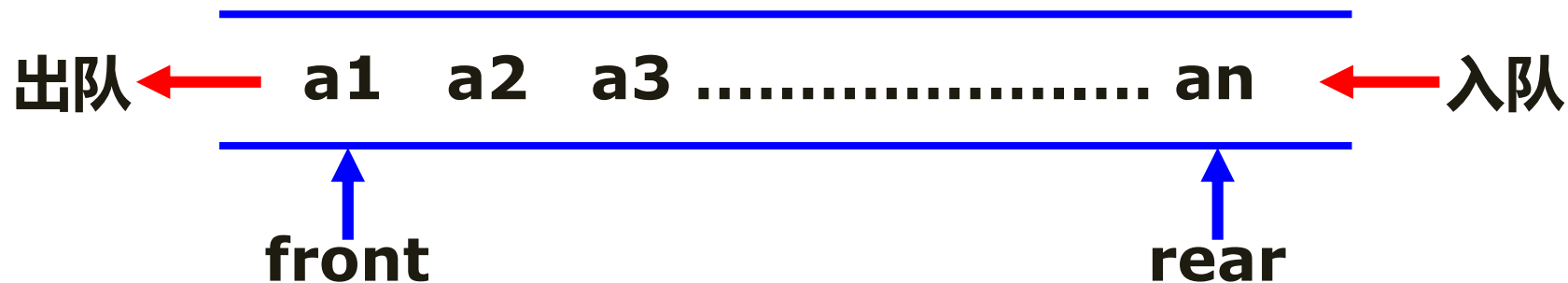
其中：hanoi(int n, int src, int tar, int aux) 表示将塔座src上的n个盘子移动到塔座tar，以塔座aux为辅助（auxiliary）



**队列**

**(queue)**

# 队列



队列 Queue =  $(a_1, a_2, \dots, a_n)$

∞ 队列：限定只能在表的一端插入，在另一端删除的线性表

- 队尾 (rear)：允许插入的一端
- 队头 (front)：允许删除的一端

∞ 队列的特点：先进先出 (**F**irst **I**n **F**irst **O**ut, **FIFO**)



# 队列

## ❧ 队列的实现方式有两种

- 第一种方式：顺序队列
  - 采用顺序存储结构（顺序表）实现
- 第二种方式：链队列
  - 采用链式存储结构（链表）实现

## ❧ 队列的主要操作

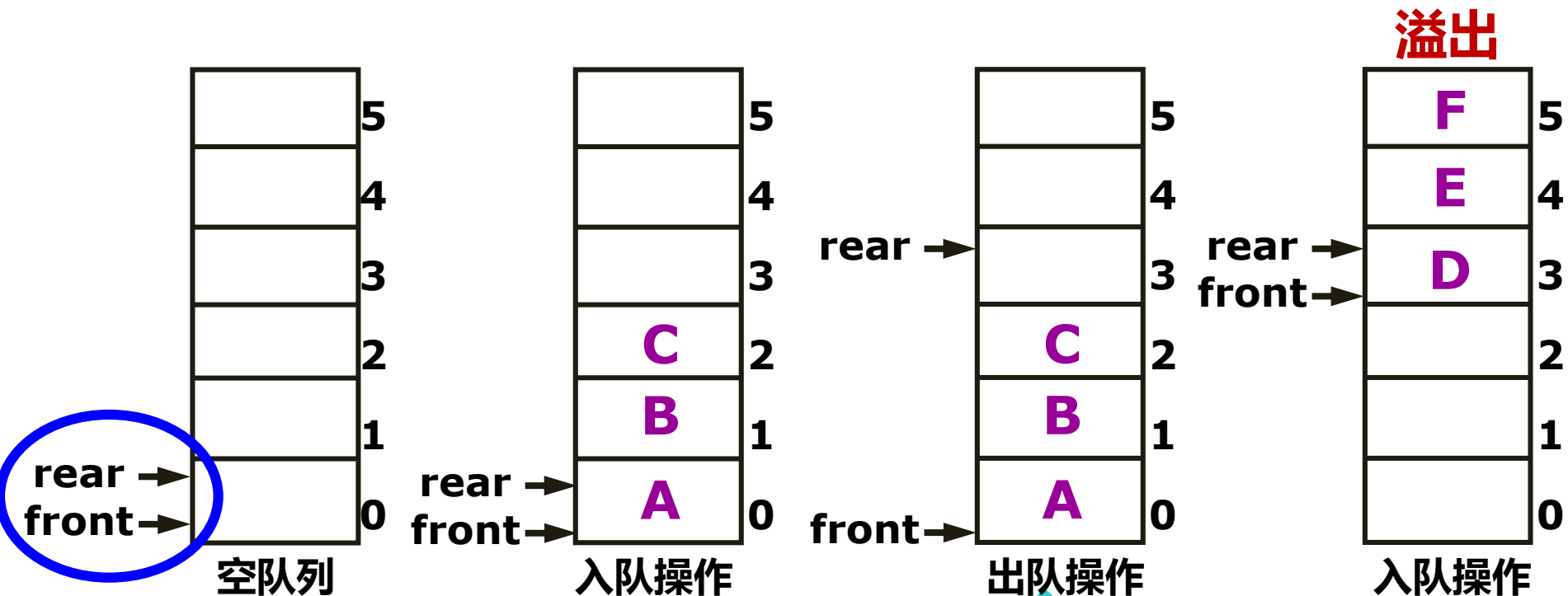
- 初始化：init\_queue(Q)
- 入队：enqueue(Q, e)
- 出队：dequeue(Q)
- 判队空，判队满（循环队列）

# **顺序队列**

## **(Sequential Queue)**

# 顺序队列的存储结构

☞ 顺序队列采用一维数组实现，设置头尾两个指针



**front:** 指示队首位置

**rear:** 指示队尾元素的下一个位置

**初值:** `front=rear=0`

**入队:** `Q[rear++] = x;`

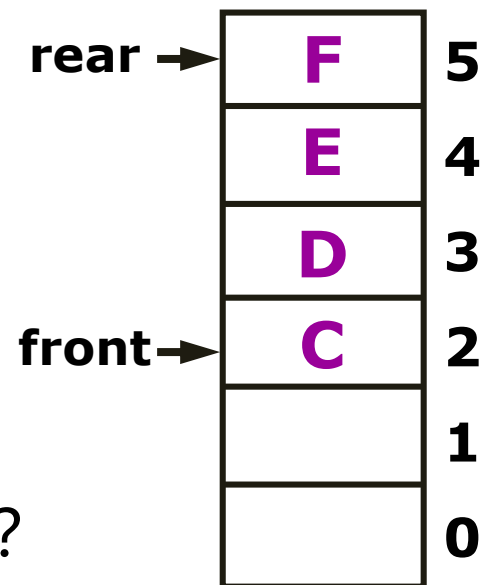
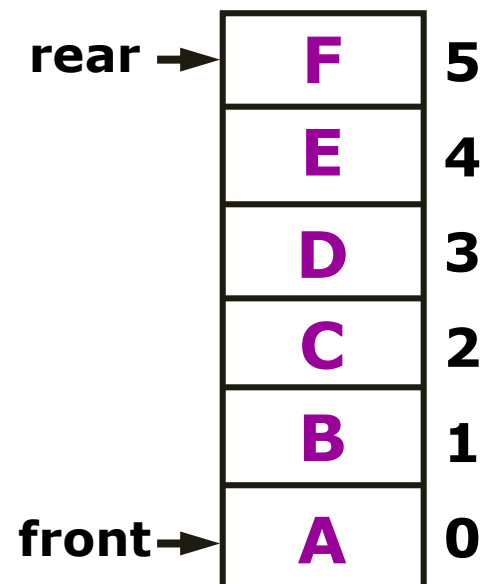
**出队:** `x = Q[front++];`

**空队列:** `front == rear`

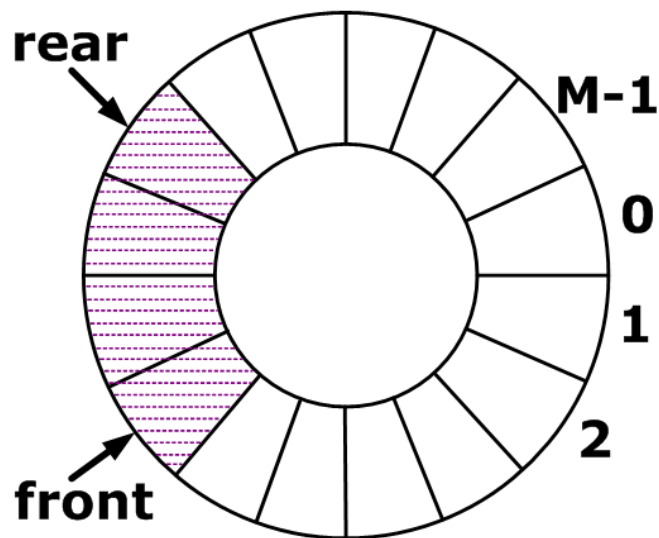
# 顺序队列的存储结构

## ❧ 顺序队列存在的问题

- 设数组维数为M
- **front==0, rear==M**
  - 如果此时有元素入队会怎样?
  - 队列发生: **溢出**
- **front≠0, rear==M**
  - 如果此时有元素入队会怎样?
  - 队列发生: **假溢出**
- 思考: 怎样利用front指针之前的空间?



# 顺序队列的存储结构



❧ 怎样利用front指针之前的空间？

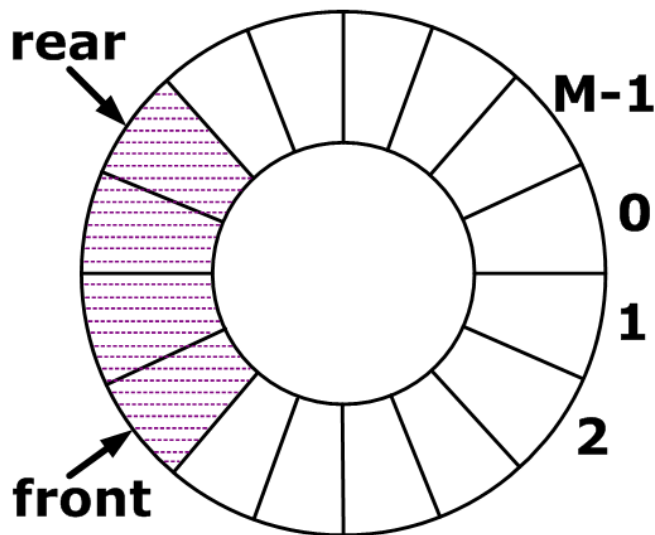
- 每次出队操作后剩余元素向下移动 ..... **浪费时间!**
- 采用循环队列
  - 将队列设想成环形：让 $Q[0]$ 接在 $Q[M-1]$ 之后
  - 若  $\text{rear} == M$ ，则令  $\text{rear} = 0$  ..... **思考：如何实现?**



# 循环队列 (Circular Queue)

∞ 循环队列的实现：利用“模”运算

- 入队：
  - $Q[\text{rear}] = x;$
  - $\text{rear} = (\text{rear} + 1) \% M;$
- 出队：
  - $x = Q[\text{front}];$
  - $\text{front} = (\text{front} + 1) \% M;$



∞ 问题：怎样判空和判满？

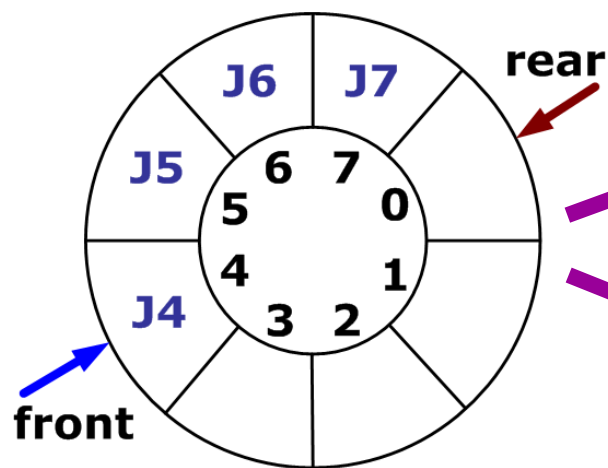


# 循环队列：队满和队空的判定条件

队空:  $\text{front} == \text{rear}$

队满:  $\text{front} == \text{rear} ?$

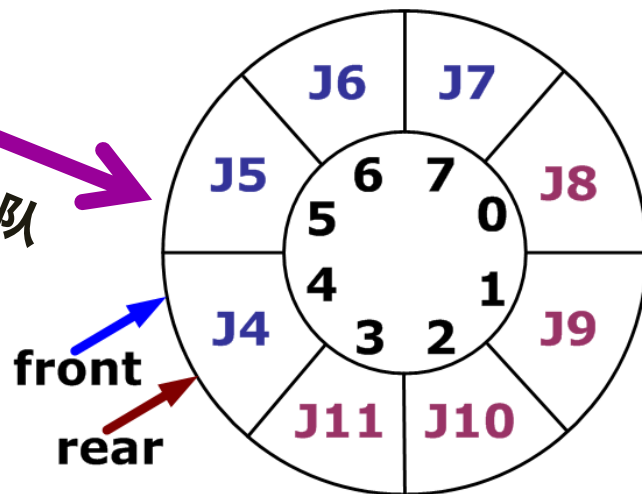
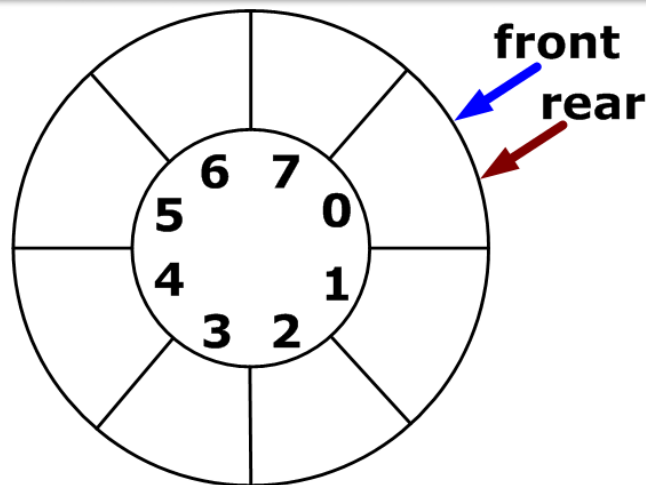
你们在suo啥?



初始状态

J4, J5, J6, J7出队

J8, J9, J10, J11入队



解决方案1: **设置一个标志变量**

队空:  $\text{flag} = 0$  队满:  $\text{flag} = 1$

解决方案2: **牺牲一个元素空间**

队空:  $\text{front} == \text{rear}$  队满:  $(\text{rear} + 1) \% M == \text{front}$

# 顺序队列的数据结构定义

## 静态分配存储空间

```
typedef struct {  
    int front, rear;           // 队头和队尾指针  
    ElemType data[MAXSIZE];  // 数据元素存储空间  
}TQue;
```

## 动态分配存储空间

```
typedef struct {  
    int front, rear, len;      // 队头和队尾指针，队列长度  
    ElemType *pdata;          // 指向数据元素存储空间的指针  
}TQue, *PQue;
```



# 循环队列的初始化

```
PQue init_queue(int n){
    PQue pqe; ElemType *p;
    pqe = (PQue)malloc(sizeof(TQue)); // 判空略
    p = (ElemType*)malloc(n*sizeof(ElemType));
    if(p){
        pqe->pdata = p;
        pqe->front = 0;
        pqe->rear = 0;
        pqe->len = n;
    }
    else {printf("申请空间失败! \n"); exit(0);}
    return pqe;
}
```

# 循环队列基本操作：销毁与清空

❧ 循环队列的销毁：释放为队列动态分配的存储空间

```
void destroy_queue(Pque *ppque) {  
    Pque pque = *ppque;  
    if(pque->pdata) {  
        free(pque->pdata); pque->pdata = NULL; }  
    if(pque){  
        free(pque); *ppque = NULL;}  
}
```

❧ 循环队列的清空

```
void clear_queue(PQue pque){  
    pque->front = 0;  
    pque->rear = 0;  
}
```

# 循环队列的判空与判满：牺牲单位元素空间

∞ 循环队列的判空函数

```
int isempty_queue(PQue pqe){  
    return (pqe->front == pqe->rear);  
}
```

∞ 循环队列的判满函数

```
int isfull_queue(PQue pqe){  
    int r = (pqe->rear + 1) % (pqe->len);  
    return (r == pqe->front);  
}
```



# 循环队列基本操作：元素入队

∞ 循环队列的入队算法：插入元素  $e$  为新的队尾元素

```
void en_queue(PQue pque, ElemType e){  
    int m = pque->len;  
    if (! isfull_queue(pque)){  
        pque->pdata[pque->rear] = e;  
        pque->rear = (pque->rear + 1) % m;  
    }  
    else printf("队列已满，入队操作无效\n");  
}
```

# 循环队列基本操作：元素出队

∞ 循环队列的出队算法：取出队首元素作为函数返回值

```
ElemType de_queue(PQue pqe){  
    int m= pqe->len; ElemType item;  
    if (! isempty_queue(pqe)){  
        item = pqe->pdata[pqe->front];  
        pqe->front = (pqe->front + 1) % m;  
    }  
    return item;  
}
```

} 含有标志位或计数器的循环队列：请同学们自己实现

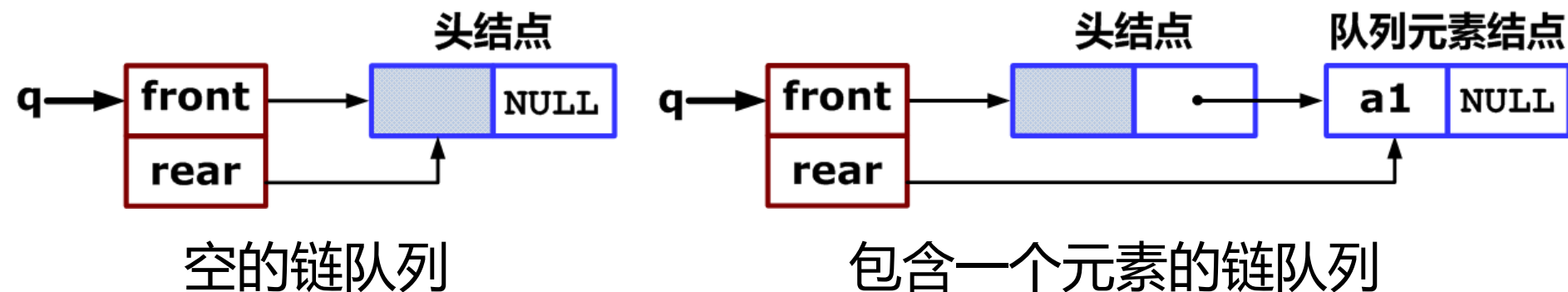




# 链队列

## (Linked Queue)

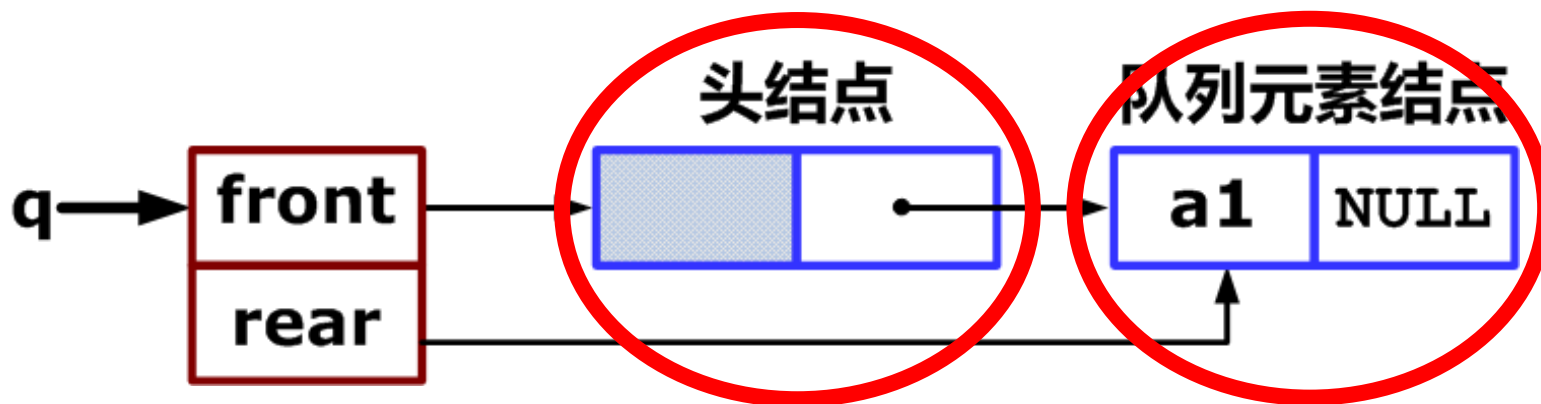
# 链队列的存储结构



问题：选择哪种链表作为队列的存储结构？

- 提示：单链表、循环单链表、双链表、循环双链表？
  - 不关心前驱结点：**采用单链表**
- 提示：是否需要头结点？ **保留头结点，以便处理空队列**
- 提示：队列是FIFO **增设一个尾指针，方便在队尾插入**
- 综上：采用含有2个指针，带头结点的单链表

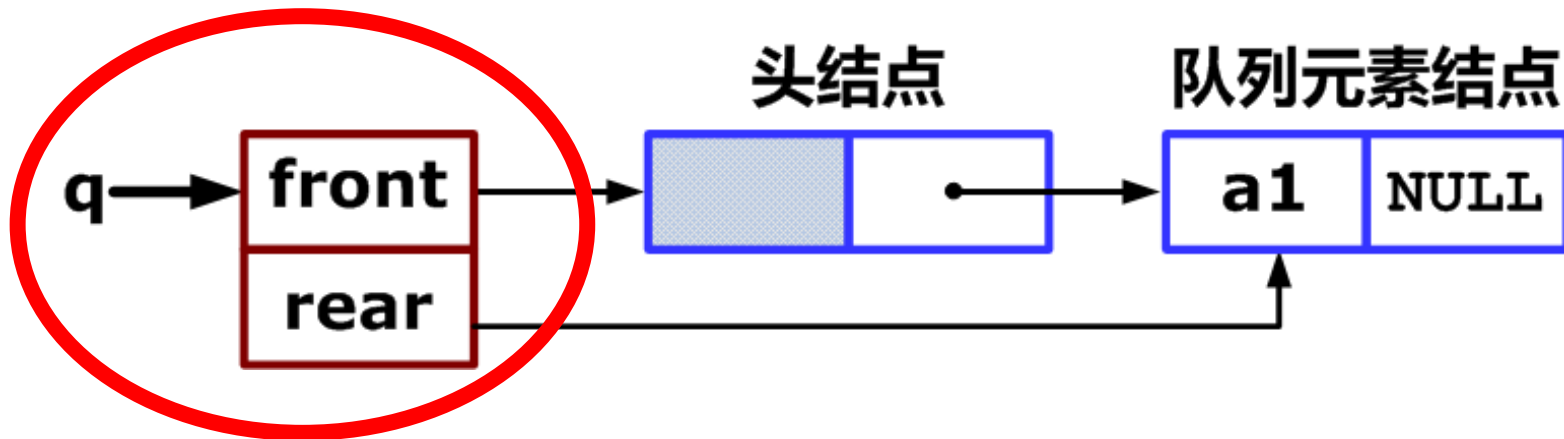
# 链队列的数据结构定义



链队列中的元素结点

```
typedef struct qnode {  
    ElemType data;           // 队列中结点的数据元素  
    struct qnode *next;      // 指向后继结点的指针  
} TNode, *PNode;
```

# 链队列的数据结构定义



链队列的数据结构

```
typedef struct{  
    TNode *front;    // 队头指针  
    TNode *rear;     // 队尾指针  
    int len;         // 队列长度  
} TQue, *PQue;
```

# 链队列的初始化

```
PQue init_queue(){
    PQue pqe; PNode p;
    pqe = (PQue)malloc(sizeof(TQue)); // 判空略
    p = (PNode)malloc(sizeof(TNode));
    if(p){
        p->next = NULL;
        pqe->front = p;
        pqe->rear = p;
        pqe->len = 0;
    }
    else {printf("申请空间失败! \n"); exit(0);}
    return pqe;
}
```

# 链队列的判空

---

```
int isempty_queue(PQue pque){  
    return (pque->front == pque->rear);  
}
```

# 链队列的基本操作：元素入队

```
void en_queue (PQue pqe, ElemType e) {  
    PNode p = (PNode)malloc(sizeof(TNode));  
    if(p){  
        p->data = e; p->next = NULL;  
        pqe->rear->next=p; // 元素入队  
        pqe->rear = p;      // 修改队尾指针  
    }  
    else{ printf("空间申请失败! \n"); exit(0); }  
}
```

# 链队列的基本操作：元素出队

```
ElemType de_queue(PQue pqe){
    PQue p; ElemType e;
    if(pqe->front == pqe->rear){
        printf("队列为空, 出队操作无效! \n"); exit(0);
    }
    p = pqe->front->next; // pqe->front指向头结点
    pqe->front->next = p->next; // 元素出队
    e = p->data; free(p);
    if(pqe->front->next == NULL) // 出队后队列为空
        pqe->rear = pqe->front;
    return e;
}
```



# 队列的应用：农夫过河问题

一个农夫带着一只狼、一只羊和一棵白菜来到河东岸准备渡河到西岸，但渡船很小，只能容纳农夫和其中一件物品，而且船只能由农夫来操作。问题：农夫在不单独留下羊和狼或者羊和白菜的情况下，怎样渡河？



# 队列的应用：农夫过河问题

- ☞ 提示：解题的第一步：用符号表示问题
  - 合理的符号表达会简化对问题的处理.....数据结构设计
- ☞ 提示：怎样表达任意时刻农夫、狼、白菜和羊的位置状态？
  - 提示：可以用 0 表示在河的东岸，1 表示在河的西岸
    - 可以用二进制数构成的**数组**来表达这四个对象的状态
    - $(dddd)_B \Leftrightarrow \{\text{Farmer, Wolf, Cabbage, Goat}\}$
- ☞ 提示：问题的初始状态和终止状态是怎样的？
  - Start: **0000**; Terminal: **1111**
- ☞ 思考：总共有多少种状态？ ..... 状态空间的维度与容量
  - 答案：维度**4**，取值范围 $\{0,1\}$ ，可能的状态共有 **$2^4$** 种



# 队列的应用：农夫过河问题

- 用 **0/1** 构成的**数组**来表达这四个对象的状态
  - $(dddd)_B \Leftrightarrow \{\text{Farmer, Wolf, Cabbage, Goat}\}$
- 问题的初始状态**S**: **0000**; 终止状态**T**: **1111** 共 $2^4$  种状态
- 原始问题转化为：在约束条件下**搜索**从状态**S**到状态**T**的路径
  - 约束1：狼和羊不能独处；羊和菜不能独处
  - 约束2：农夫必须每次均过河，且只能携带一样物品
- 思考：怎样搜索答案？
  - 问题1：怎样表达农夫的选择？
  - 问题2：怎样记录合法的状态转换路径？
  - 问题3：怎样驱动搜索过程？



# 队列的应用：农夫过河问题

∞ 问题1：怎样表达农夫的选择？

- 若当前状态为： $\{F, W, C, G\} = (1\ 0\ 1\ 1)_B$
- 设：农夫准备带羊过河，则渡河后状态？
- 答： $(0\ 0\ 1\ 0)_B \rightarrow \text{怎样用C语言表达?}$
- 提示：引入一个passenger变量  $pas = (0\ 0\ 0\ 1)_B$ 
  - 其中：1表示对应的成员参与本次渡河
  - $(0\ 0\ 1\ 0)_B = (1\ 0\ 1\ 1)_B \wedge (1\ 0\ 0\ 1)_B$
- 设农夫什么也不带，则渡河后状态？
  - $(0\ 0\ 1\ 1)_B = (1\ 0\ 1\ 1)_B \wedge (1\ 0\ 0\ 0)_B$
- 问：如果农夫想带狼过河呢？ **不可以！只能选同侧物品**

# 队列的应用：农夫过河问题

问题1：怎样表达农夫的选择？

- 怎样判断passenger是否合法（与农夫同侧）？
- 提示：关于乘客的约束有哪些？
  - 农夫必须每次均过河，且只能携带与之同侧的一样物品
- 若当前状态为：**state = (1 0 1 1)<sub>B</sub>**
- 思考：怎样判断哪些物品与农夫位于河岸的同一侧？
  - 提示：**a = state & 0X08** 农夫是否在河的西岸
  - 提示：**b = state & pas** 选定物品是否在河的西岸
  - 思考：当a与b同为真或同为假时？ 农夫与该物品位于同侧
- 此时令：**pas = pas | 0X08**

# 队列的应用：农夫过河问题

问题2：怎样记录合法的状态转换路径？

- 设置一个route数组用于记录路径间的转换途径

**int route[16] // 16种可能状态**

- 数组下标（十进制）与状态（二进制）1:1对应
- 数组填充原则：**route[m] = s**
  - **s：当前状态**（如{F, W, C, G} = **0001**  $\Leftrightarrow$  1）
  - **m：后续状态**（如{F, W, C, G} = **1011**  $\Leftrightarrow$  11）
- 问：route[11] = 1表示什么意思？
  - 答：状态11是由前驱状态1转换得到的
  - 相应的动作为：农夫携带卷心菜渡河

# 农夫过河问题：判断状态是否合法

// 判断当前状态s是否安全：若安全则返回1，否则返回0

```
int check(int s){  
    // 羊菜同岸且农夫不在场  
    if(((s & 1) != 0) == ((s & 2) != 0)) &&  
        (((s & 1) != 0) != ((s & 8) != 0)){  
        return 0; }  
    // 狼羊同岸且农夫不在场  
    if(((s & 1) != 0) == ((s & 4) != 0)) &&  
        (((s & 1) != 0) != ((s & 8) != 0)){  
        return 0; }  
    return 1;  
}
```

# 队列的应用：农夫过河问题

## 问题3：怎样驱动搜索过程？

- 提示：首先需要选择状态空间的存储结构：哪种线性表？
- 两种基本的搜索策略：深度优先搜索 vs 广度优先搜索
  - 深度优先搜索采用栈实现；广度优先搜索采用队列实现
- 对本问题而言栈和队列都可以：队列更符合解题直觉
  - 引入状态选择队列：sque
  - 用于保存搜索过程中农夫的选择可能导致的状态改变
  - 思考：怎样利用队列实现搜索？



# 农夫过河问题的算法流程

## ❧ 系统初始化

- route数组元素初始化值全部设置为-1
- 初始化状态选择队列sque：将状态0000入队（即数字0）

## ❧ 循环处理

- 处理队首状态的转换（农夫可能的选择）
- 如果渡河动作导致的状态结果安全，则：
  - 将渡河动作结束后得到的新状态加入队尾
  - 在route数组中新状态对应的位置记录当前状态
- 结束条件？ **状态选择队列为空，或者route[15] != -1**

# 农夫过河问题

```
void cross_river(){
    TQue squeue; int pas, after; int route[16]; .....
    for(i = 0; i < 16; i++) route[i] = -1;
    init_queue(&squeue, 17);
    en_queue(&squeue, 0); // 初始状态0入队
    while(!isempty(&squeue) && (route[15] == -1)){
        s = de_queue(&squeue);
        for(pas = 1; pas <= 8; pas <<= 1 ){
            if(((s & 8) != 0) == ((s & pas) != 0)){
                after = s ^ ( 8 | pas ); // 农夫每次需渡河
                if(check(after) && (route[after] == -1)){
                    route[after] = s;
                    en_queue(&squeue, after);
                }
            }
        }
    }
}
```



# 农夫过河问题的算法流程

## 结果输出

- 首先判断问题是否有解
  - 怎样判断? **route[15] != -1时有解**
- 如果有解, 怎样得到状态变迁的路径?
  - 可根据route数组元素的值得到状态路径
- 思考: 采取怎样的策略遍历route数组得到正确输出?
  - 循环初始条件: **state = 15**
  - 循环结束条件: **state == 0**
  - 循环变量更新: **state = route[state]**

