

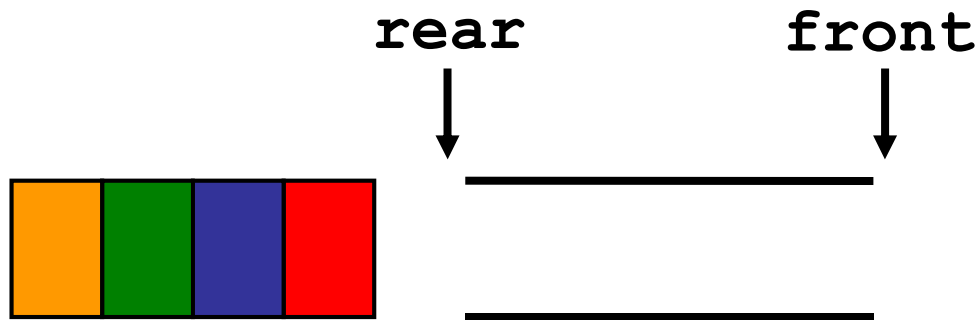
队列的类型定义

- 定义

- 队列是必须在一端删除 (队头**front**) , 在另一端插入 (队尾**rear**) 的线性表

- 特性

- 先进先出 (FIFO, First In First Out)



队列的类型定义

数据对象: 具有线形关系的一组数据

操作:

bool EnQueue(Queue &Q, ElemType e); //入队

bool OutQueue(Queue &Q, ElemType &e); //出队

bool GetFront(Queue Q, ElemType &e); //取队头

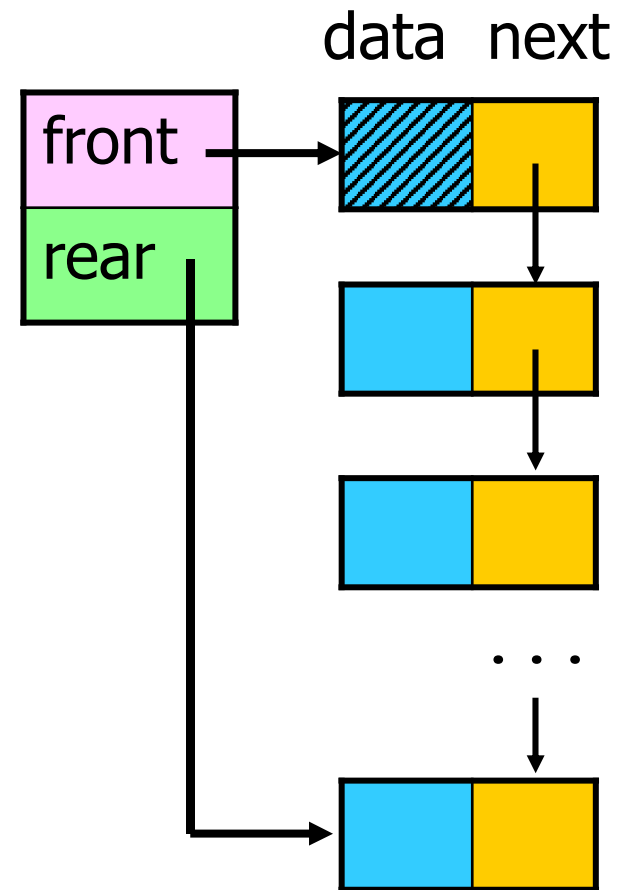
bool IsEmpty(Queue Q); //空吗?

bool Clear(Queue &Q); //清空

队列的表示：链式表示

- 链式表示和实现

```
typedef struct qnode{  
    ElemType data;  
    struct qnode *next;  
}QNode;  
  
typedef struct{  
    QNode* front;  
    QNode* rear;  
}LinkQueue;
```



队列的表示：链式表示

- 初始化

```
bool InitQueue(LinkQueue& Q) {  
    Q.front = Q.rear =  
        (QNode*)malloc(sizeof(QNode));  
    if(!Q.front) return false;  
    Q.front->next = NULL;  
    return true;  
}
```

队列的表示：链式表示

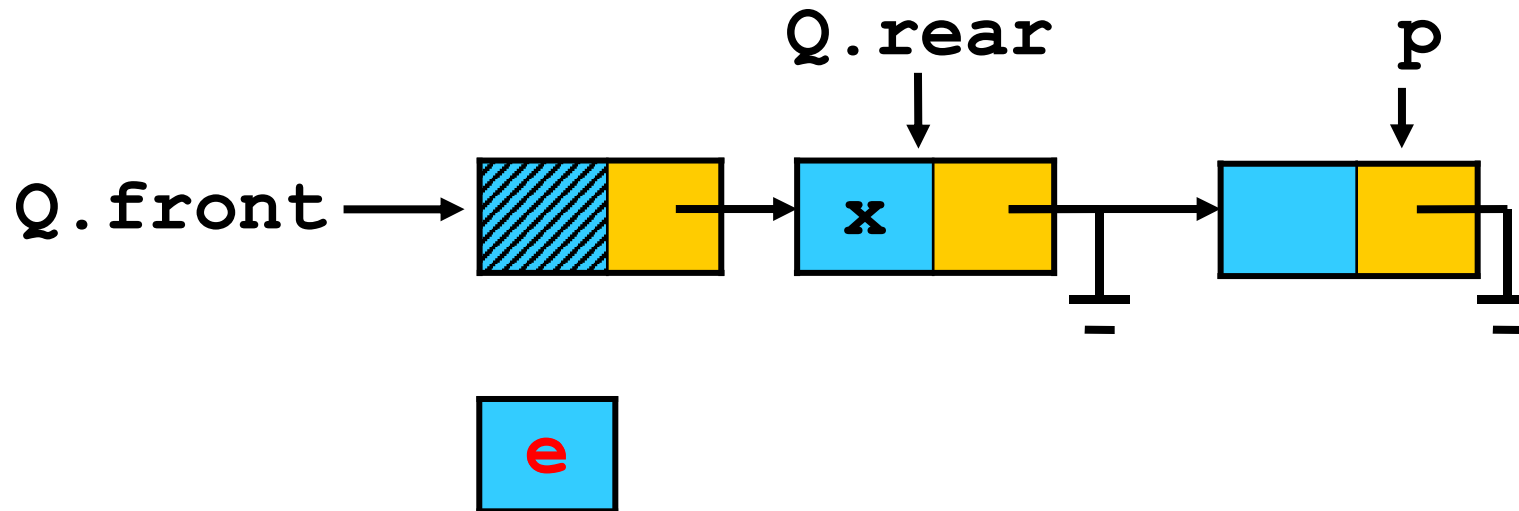
- 入队

```
bool EnQueue(LinkQueue& Q, ElemType
e) {
    QNode *p = (QNode *)
                malloc(sizeof(QNode));
    if(!p)    return false;
    p->data = e;    p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return true;
}
```

```

bool EnQueue(LinkQueue& Q, ElemType e) {
    QNode *p = (QNode *)malloc(sizeof(QNode));
    if(!p)    return false;
    p->data = e; p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return true;
}

```



队列的表示：链式表示

- 出队

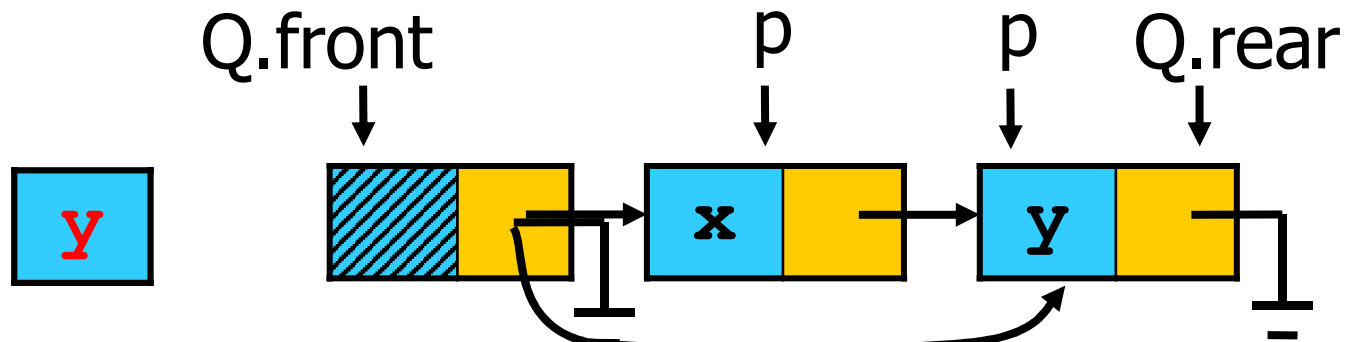
```
bool DeQueue(LinkQueue& Q)
{
    if (Q.front == Q.rear)    return false;
    QNode *p = Q.front->next;
    Q.front->next = p->next;
    if (Q.rear == p)    Q.rear = Q.front;
    free(p);
    return true;
}
```

```

bool DeQueue(LinkQueue& Q) {
    if(Q.front == Q.rear)           return false;
    p = Q.front->next;

    Q.front->next = p->next;
    if(Q.rear == p)                Q.rear = Q.front;
    free(p);                        最后一个元素
    return true;
}

```



队列的表示：顺序表示

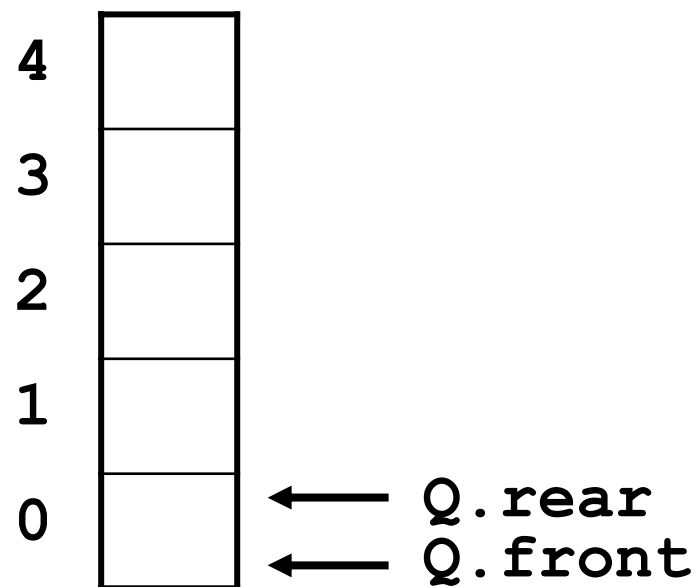
- 顺序表示和实现

- 用数组存储数据

- 初始化: **front = rear = 0**

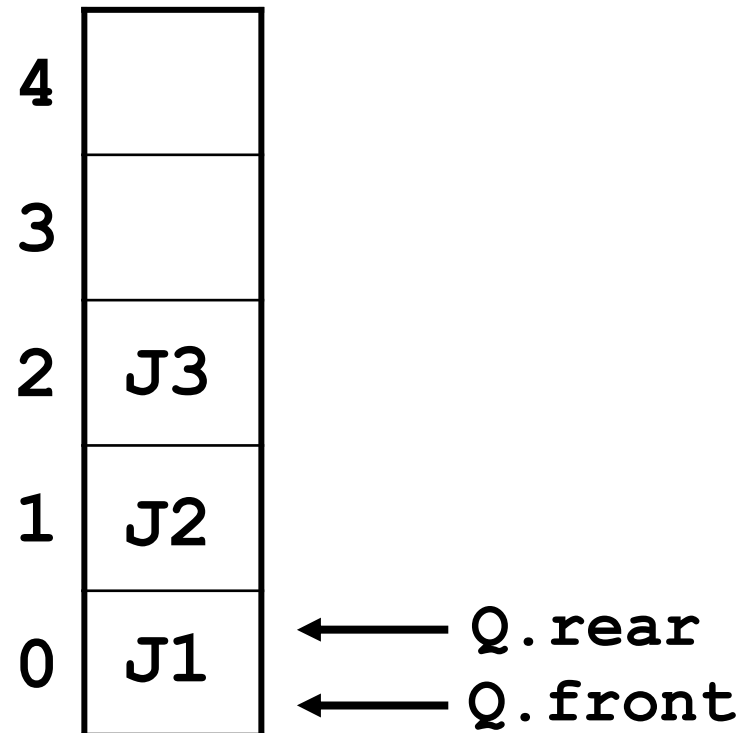
- 入队: **rear ++**

- 出队: **front ++**



队列的表示：顺序表示

- 插入和删除



队列的表示：顺序表示

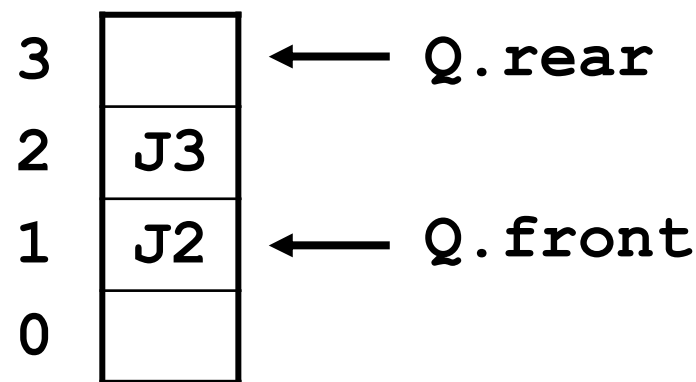
- 指针的指法

- **front**指向队头元素
- **rear**指向队尾元素的后一个单元

- 其实也可以如此错开

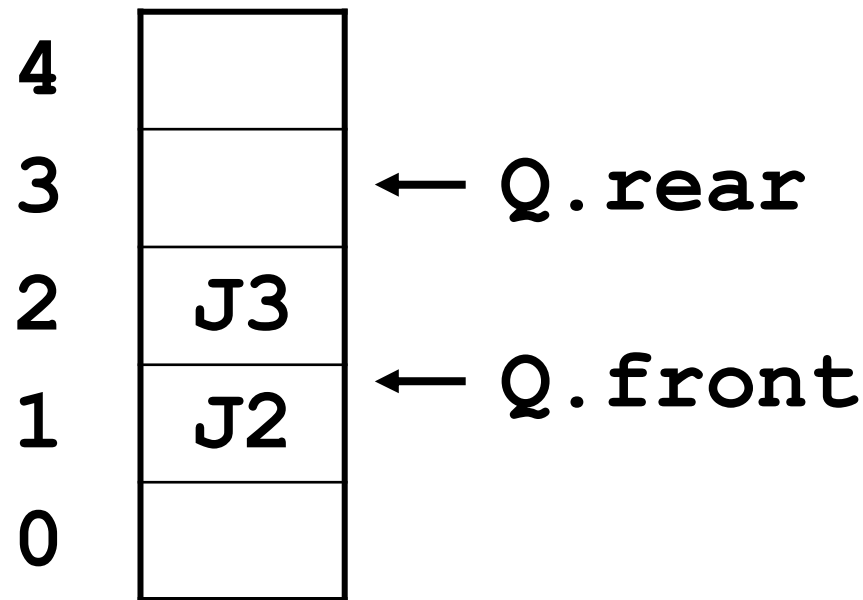
- **front**指向队头元素的前一个单元
- **rear**指向队尾元素

- 为什么如此呢？



队列的表示：顺序表示

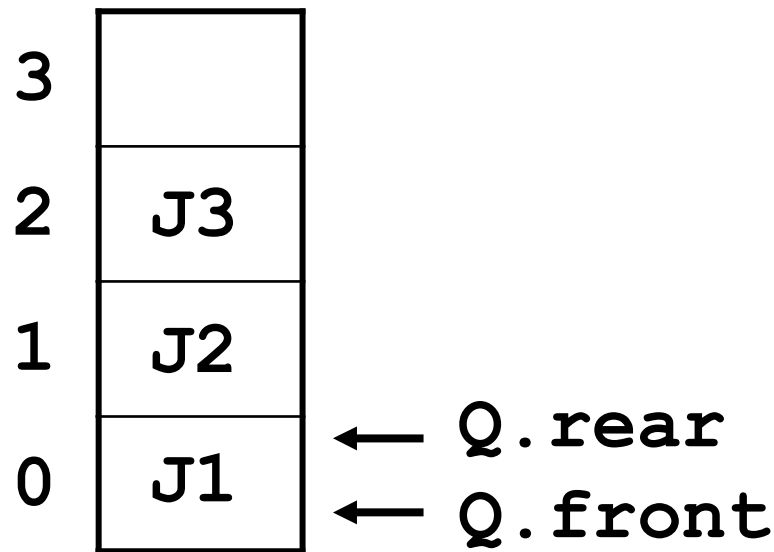
- 原因1：方便计算元素个数
 - 元素的个数 = `rear-front`
 - 而不用 = `rear-front+1`



队列的表示：顺序表示

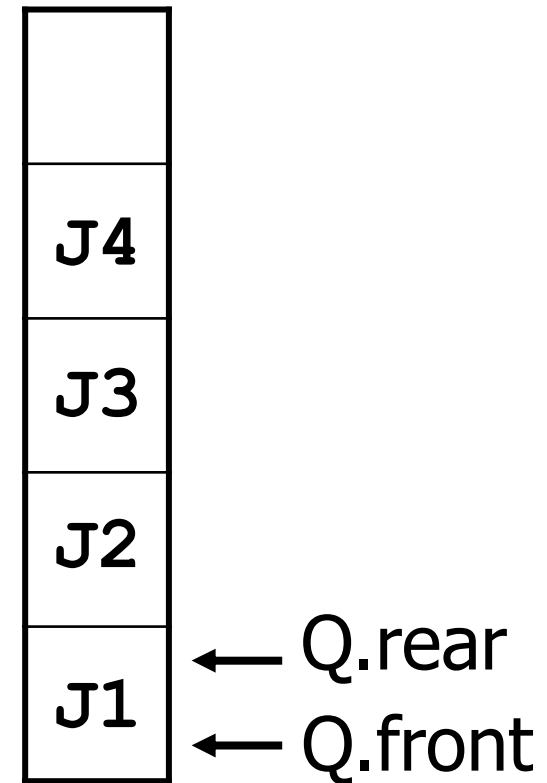
- 原因2：便于判别空队列

- 如果不错开，则空队列和只有一个元素时，都是 $\mathbf{front=rear}$ ，难以区别
- 或者空队列有可能有两种情况



队列的表示：顺序表示

- 线性队列的缺点
 - 不论是插入还是删除元素
 - **front**和**rear**都只是++
- 导致：
 - 数组空间有限，**rear**总有一天会达到数组顶端
 - **front**之前空间再不会被用到



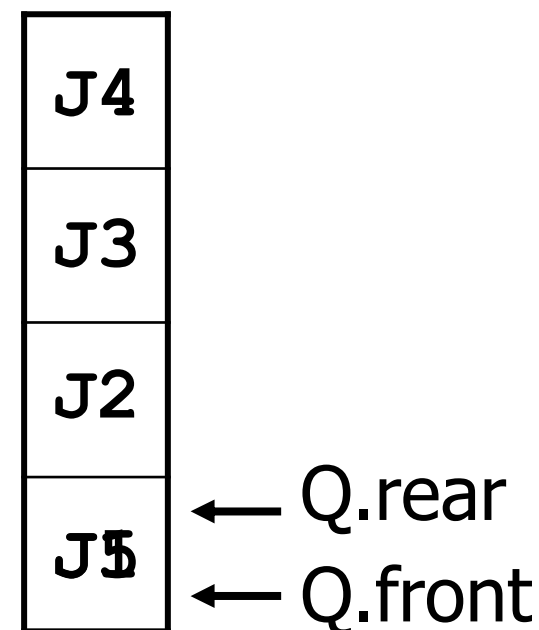
队列的表示：顺序表示

- 思考

- 既然上面不够，下面浪费
- 何不利用下面的空间？

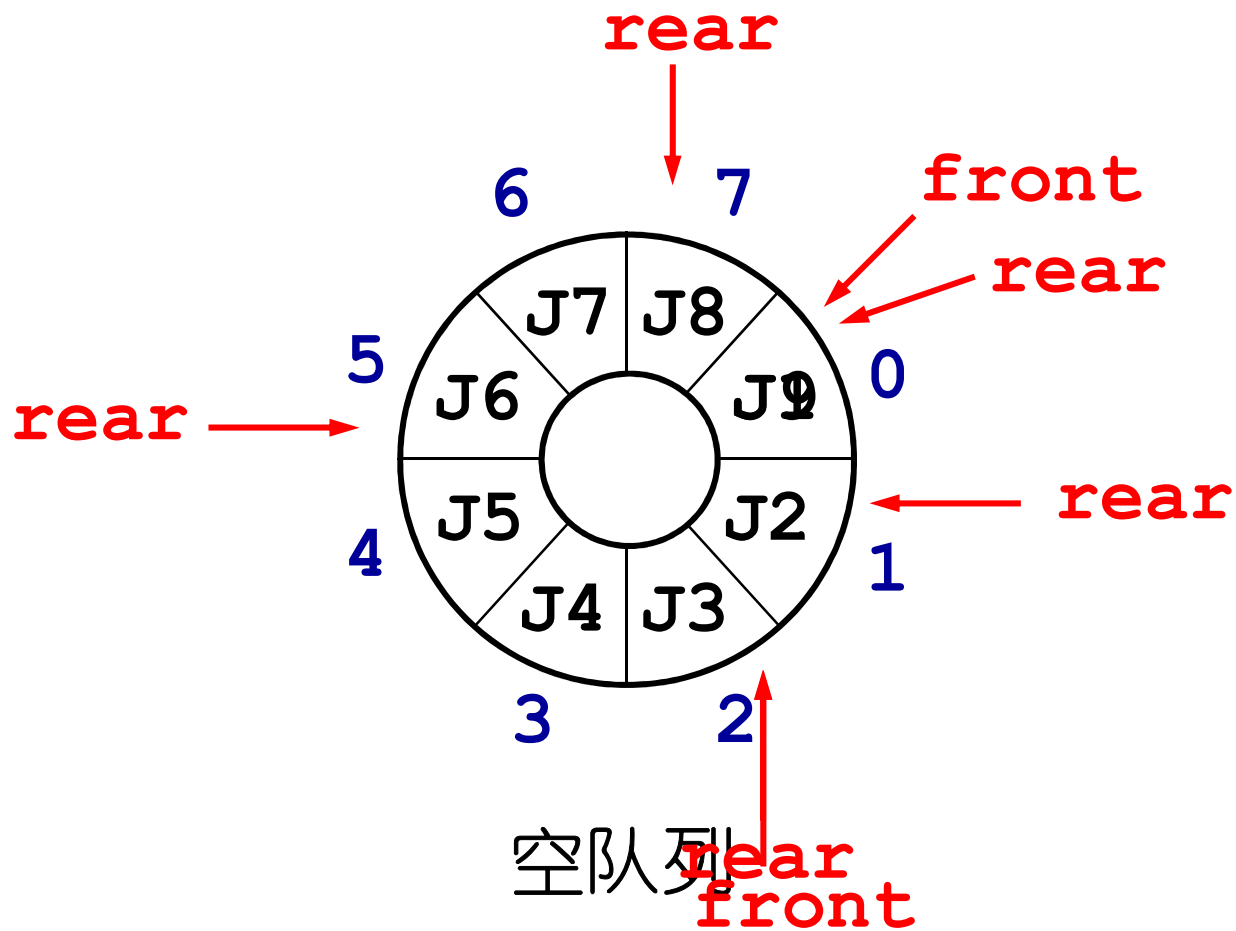
- 循环队列

- 当**rear**向上走到顶的时候，重新返回到最下端（如果下面有空闲单元）



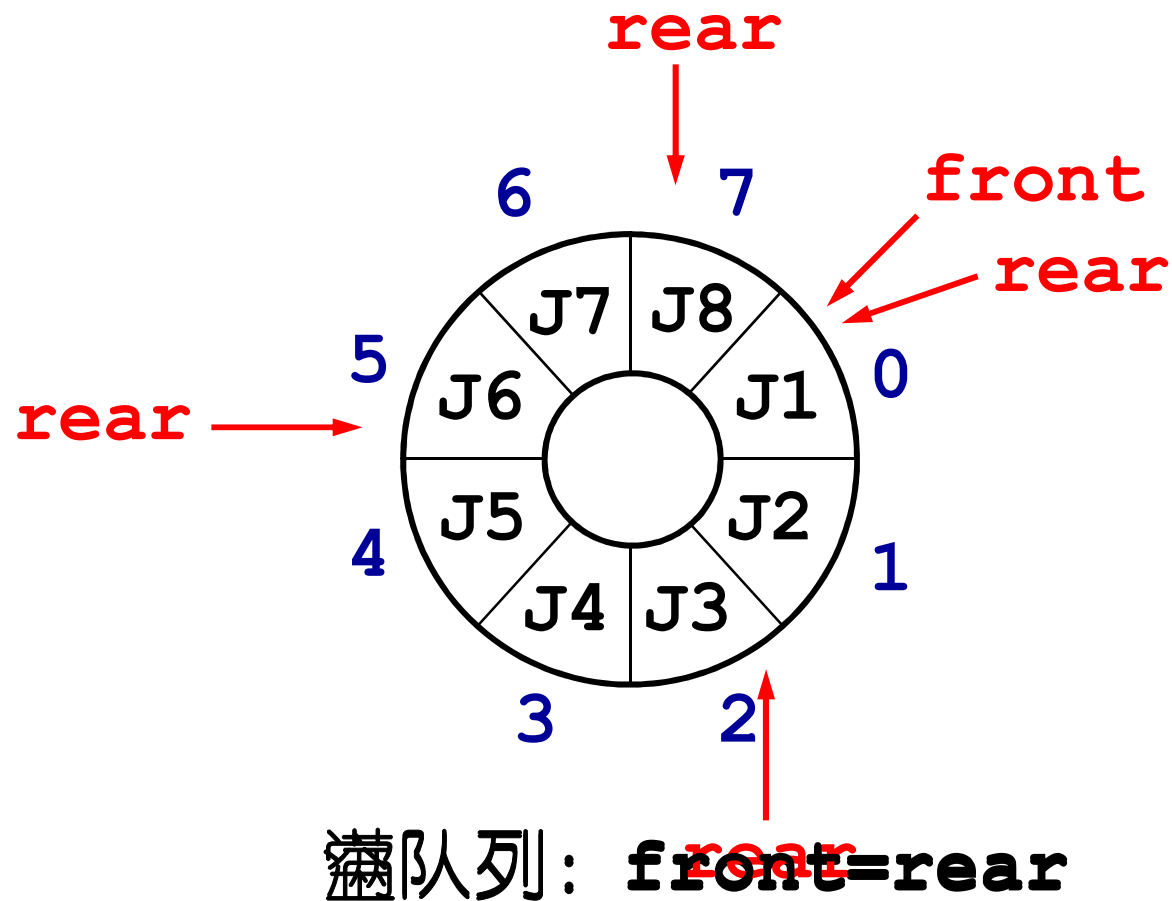
队列的表示：顺序表示

- 循环队列



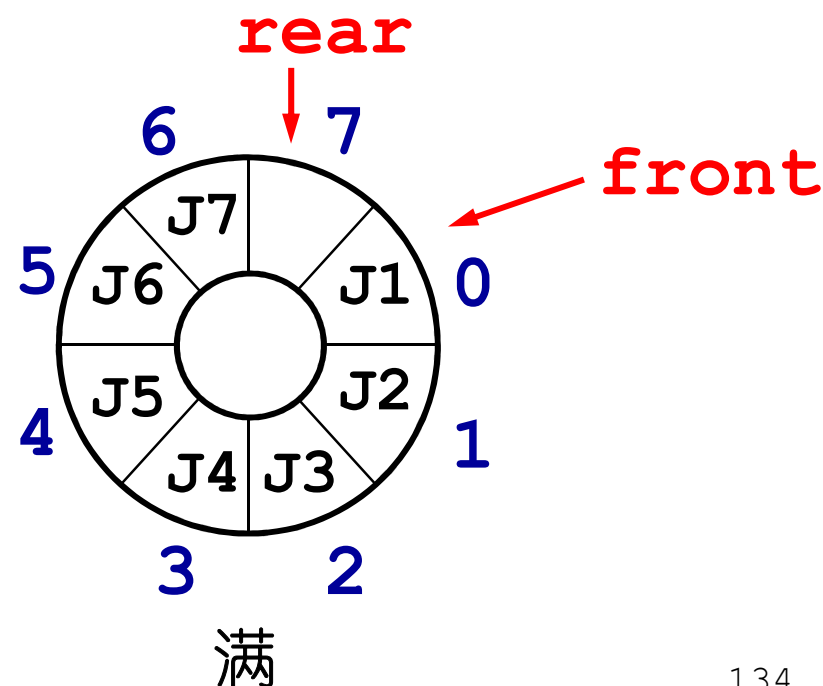
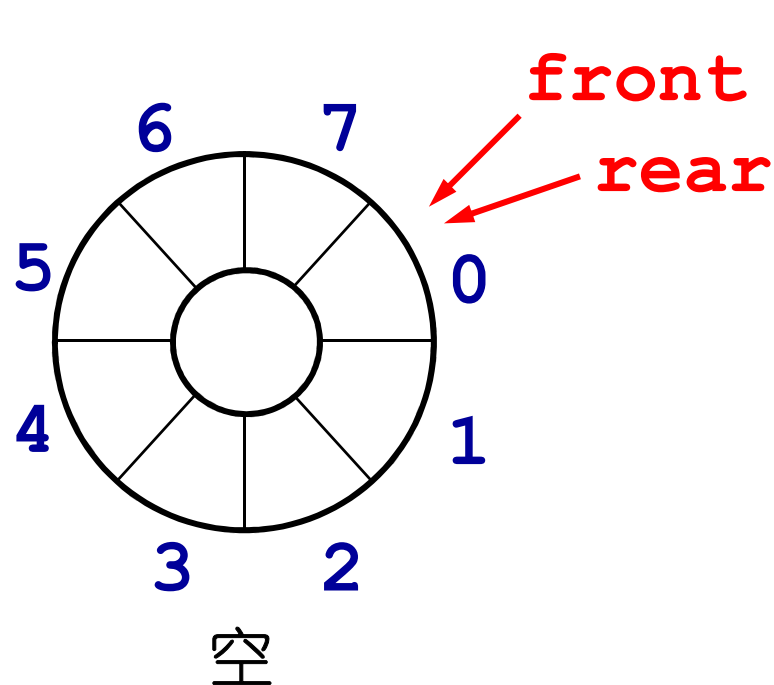
队列的表示：顺序表示

- 如何区别空队列和满队列？



队列的表示：顺序表示

- 如何区别空队列和满队列？
 - 方法一：专门设置一个标记
 - 方法二：还剩一个单元时就算满



队列的表示：顺序表示

- 循环队列类型定义

```
typedef struct {  
    ElemType *base;  
    int front;  
    int rear;  
    int capacity;  
} SqQueue;
```

队列的表示：顺序表示

- 初始化

```
bool InitQueue(SqQueue &Q, int cap)
{
    Q.base = (ElemType*) malloc
        (cap*sizeof(ElemType));
    if(!Q.base) return false;
    Q.front = Q.rear = 0;
    Q.capacity = cap;
    return true;
}
```

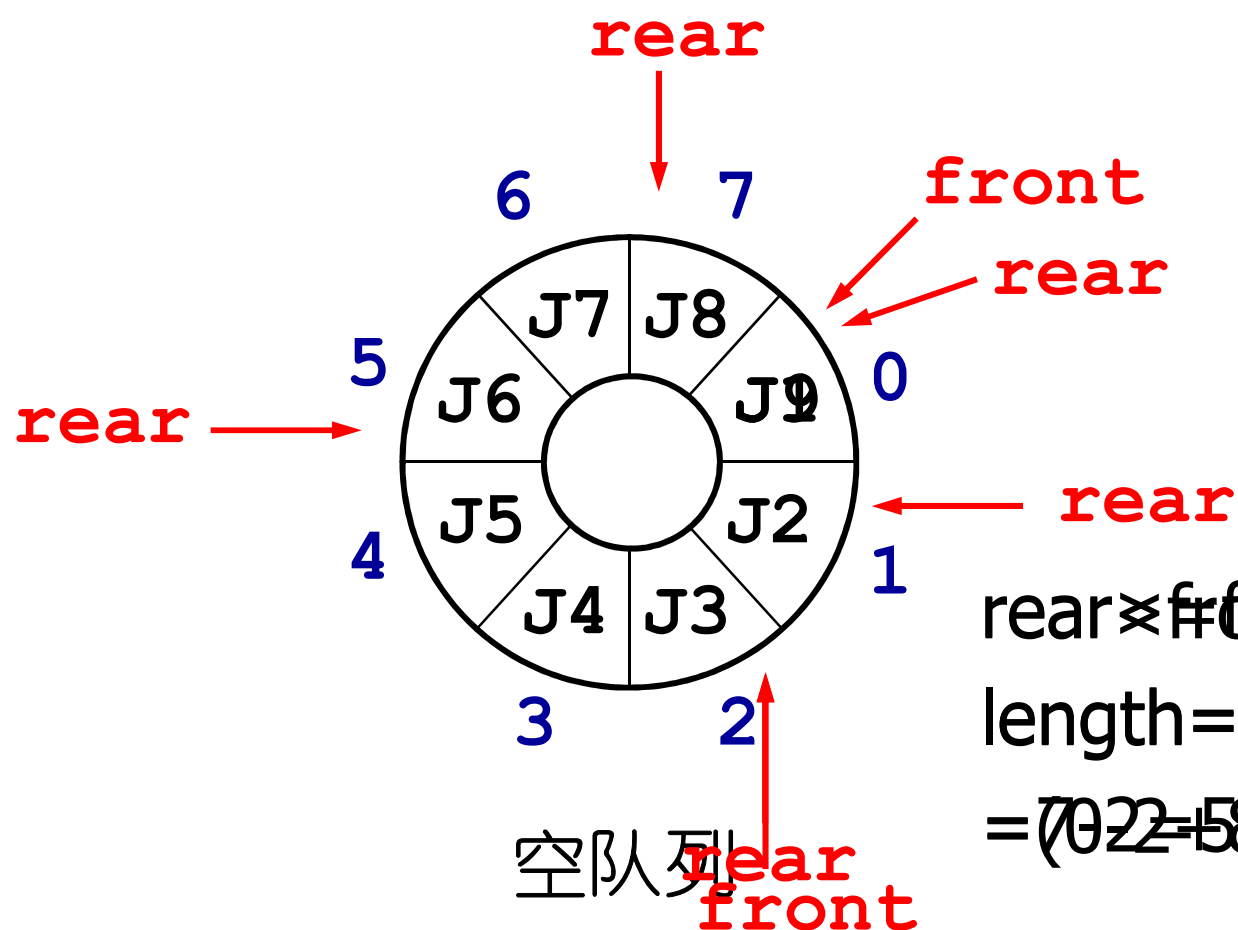
队列的表示：顺序表示

- 得到队列长度
 - 即元素个数

```
bool QueueLength (SqQueue Q)
{
    return
        (Q.rear - Q.front + Q.capacity)
        % Q.capacity;
}
```

队列的表示：顺序表示

- 得到队列长度



rear ≠ front 时:

$$\text{length} = (\text{rear} - \text{front} + 8) \% 8$$
$$= (0 - 22 + 58) \% 8 = 6$$

队列的表示：顺序表示

- 入队

```
bool EnQueue(SqQueue &Q, ElemType e) {  
    if ( (Q.rear+1)%Q.capacity== Q.front) {  
        return false;    //队列满  
    }  
    Q.base[Q.rear] = e;  
    Q.rear = (Q.rear + 1)%Q.capacity;  
    return true;  
}
```

队列的表示：顺序表示

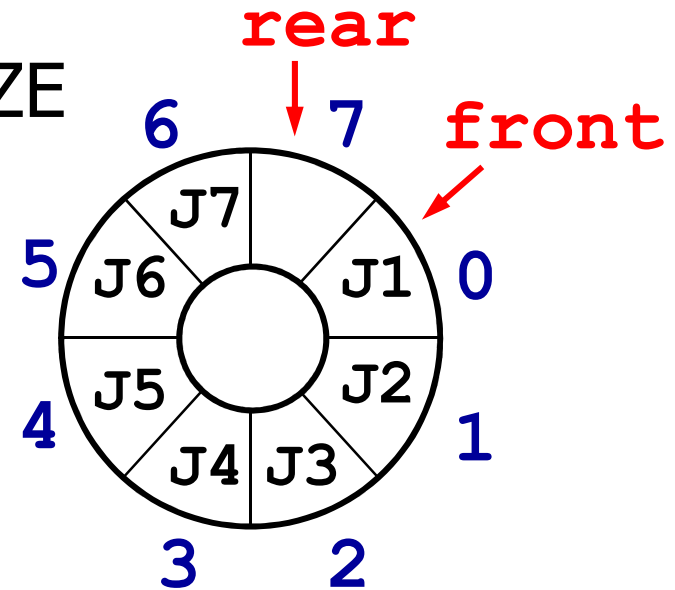
- 注意

- 满队列：**rear**下一个就是**front**

- $(\text{rear} + 1) \% \text{MAXQSIZE} == \text{front}$

- **rear**指向下一个单元

- $\text{rear} = (\text{rear} + 1) \% \text{MAXQSIZE}$



队列的表示：顺序表示

- 出队

```
bool DeQueue (SqQueue &Q) {  
    if (Q.front == Q.rear)  
        return false;    //空队列  
  
    Q.front = (Q.front+1)%Q.capacity;  
    return true;  
}
```

队列的表示：顺序表示

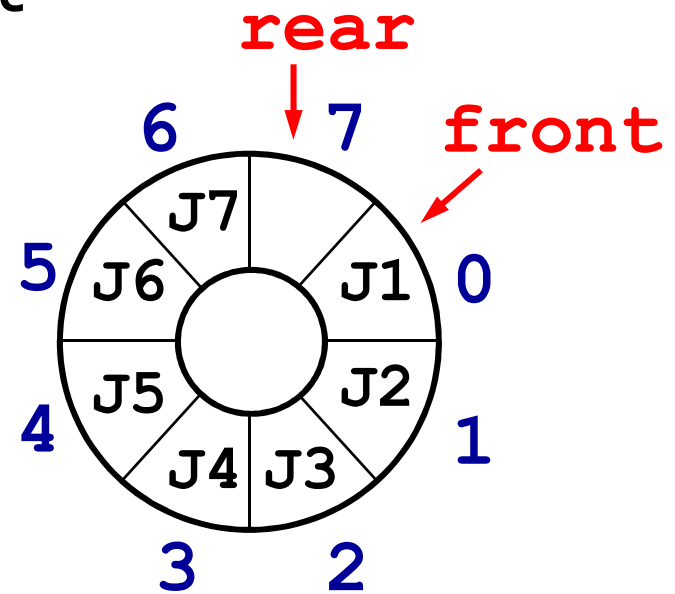
- 取队头

```
bool Front(SqQueue &Q, ElemType &e) {  
    if (Q.front == Q.rear)  
        return false;    //空队列  
    e = Q.base[Q.front];  
  
    return true;  
}
```

队列的表示：顺序表示

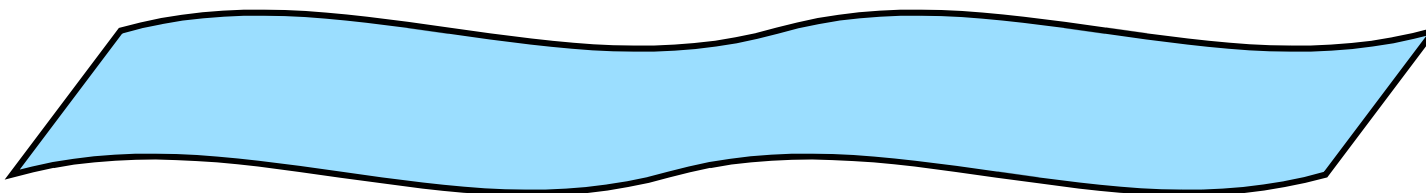
- 注意

- 空队列： **rear==front**
- **front**指向下一个单元
 - $\text{front} = (\text{front} + 1) \% \text{MAXQSIZE}$



农夫过河问题

- 一个农夫带着一只狼、一只羊和一棵白菜过河。如果没有农夫看管，则狼要吃羊，羊要吃白菜。但是船很小，只够农夫带一样东西过河。问农夫该如何解此难题？



安全与不安全状态

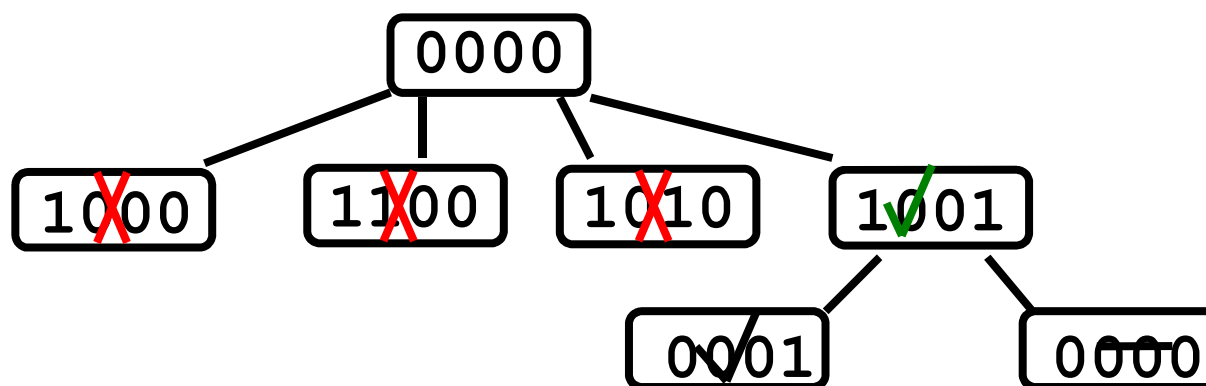
- 用四位二进制数顺序分别表示农夫、狼、白菜和羊的当前状态（位置）。用0表示在河的南岸，1表示在河的北岸。

例如整数5（其二进制表示为0101）表示农夫和白菜在河的南岸，而狼和羊在北岸。

安全状态与不安全状态：单独留下白菜和羊，或单独留下狼和羊在某一岸的状态是不安全的

求解：状态空间搜索

从初始状态0 (0000) 出发搜索可能正确的安全状态过渡序列，直到最终状态16 (1111)



广度优先搜索：在搜索过程中总是先考虑当前状态的所有状态，再进一步考虑更后面的各种情况。

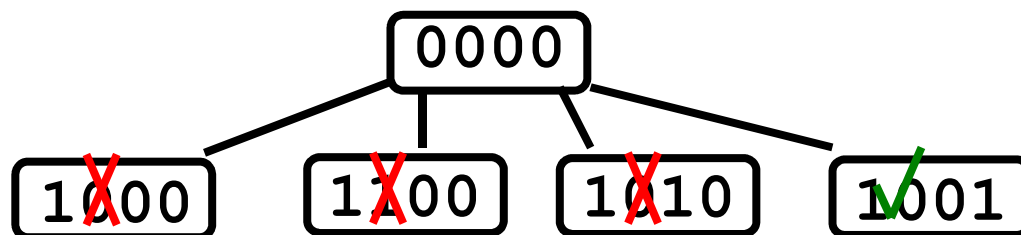
数据结构

- 1). 顺序表`route[]`: 状态`i`是否已被访问过, 若已被访问过则在这个顺序表元素中记入前驱状态值。
- `route[i] = -1`, 表示未访问过; 则 `route[i] = pre`, 表示状态`i`已经访问过, 且是从安全状态`pre`过渡来的。
 - 最后可以利用`route`顺序表元素的值建立起正确的状态路径。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

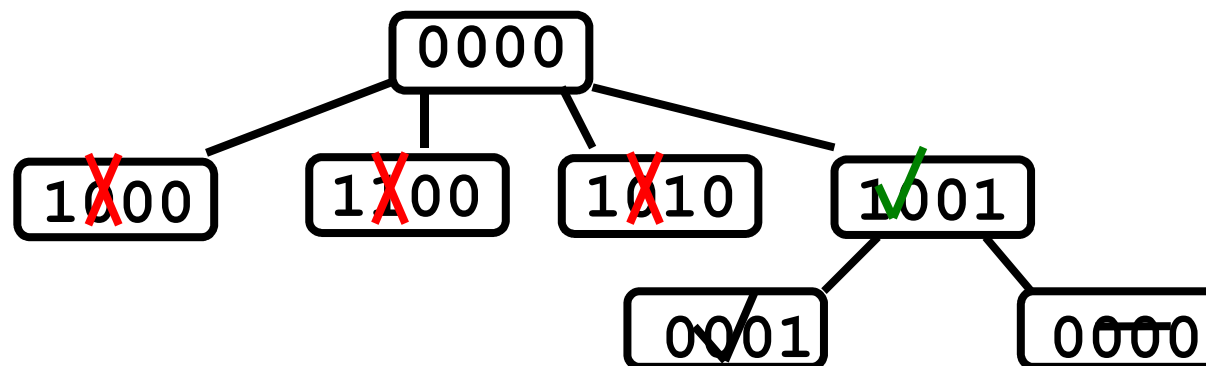
- 2) 整数队列`moveTo`: 每个元素表示可以安全到达的状态。

基于队列的状态遍历



0000															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

基于队列的状态遍历



0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

0000

0001

0010

0011

0100

0101

0110

0111

1000

1001

1010

1011

1100

1101

1110

1111

0

-1

-1

-1

-1

-1

-1

-1

-1

0

-1

-1

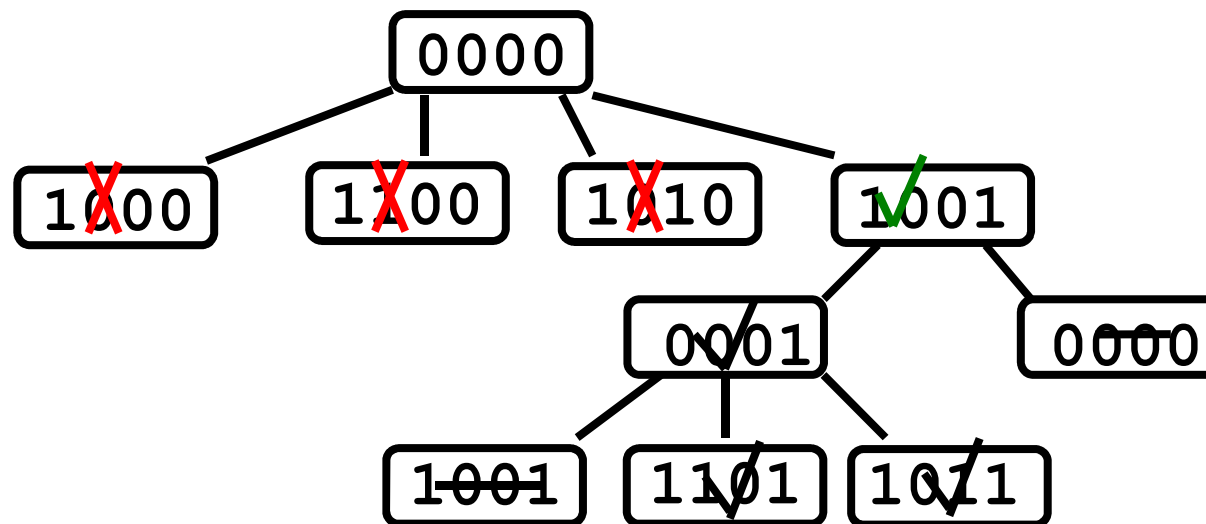
-1

-1

-1

-1

基于队列的状态遍历



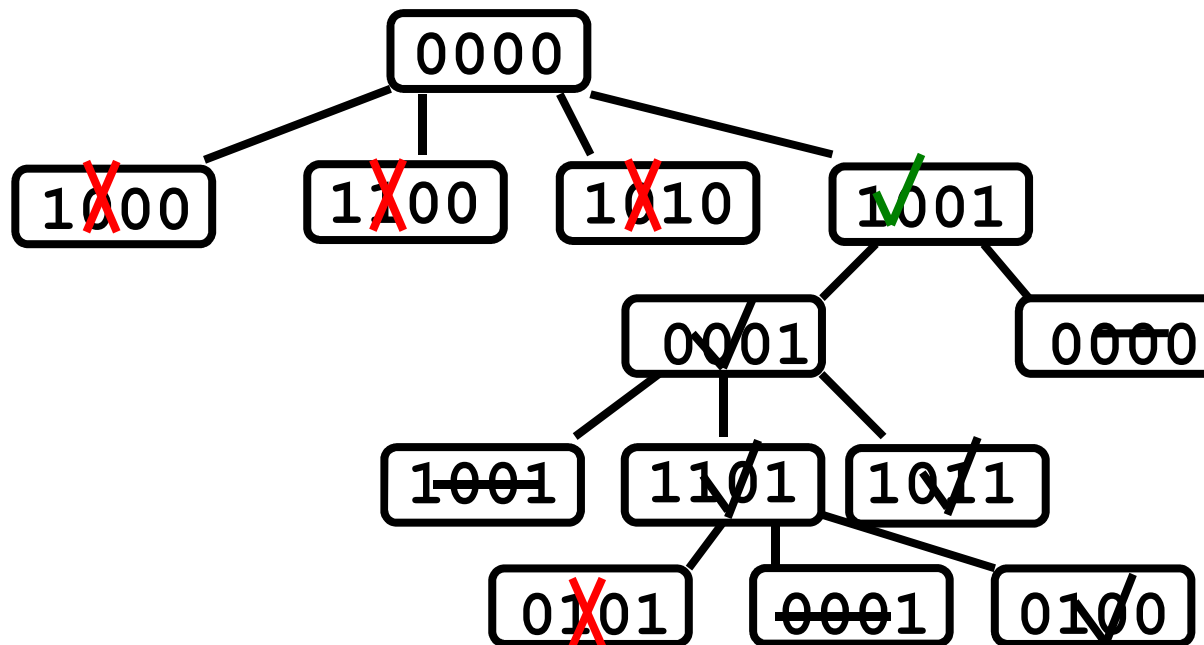
0001

01000100110111

0000000100100011001010111100110111101111

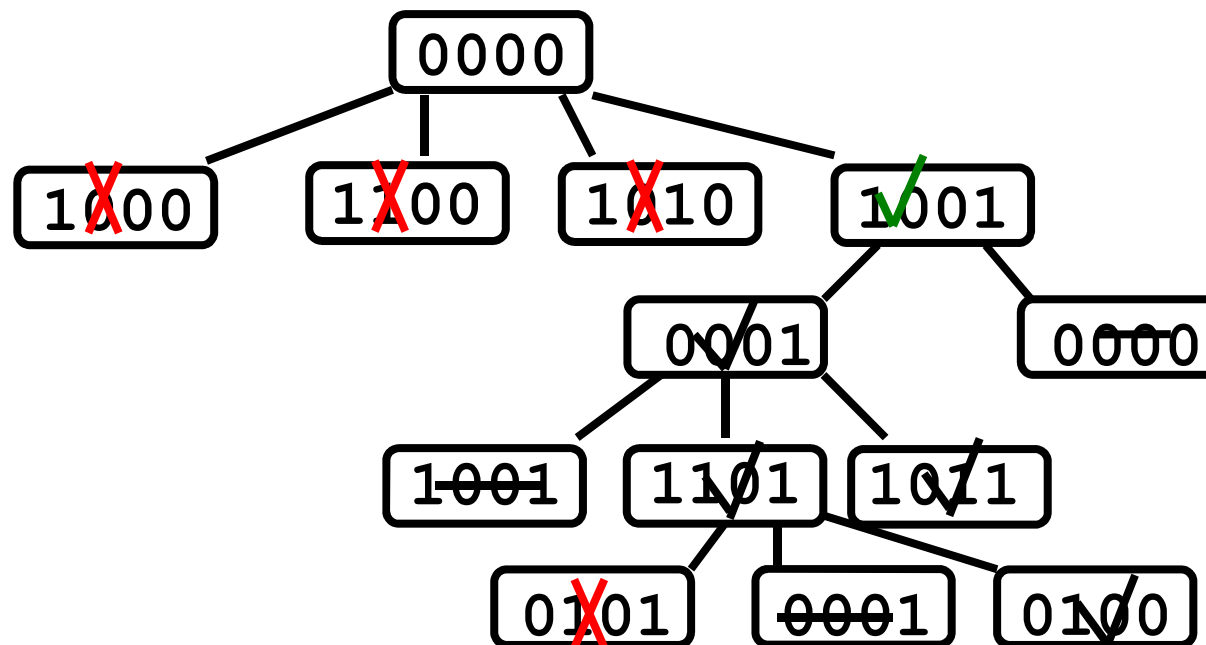
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	9	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1

基于队列的状态遍历



												1101		1011	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	9	-1	-1	-1	-1	-1	-1	-1	0	-1	1	-1	1	-1	-1

基于队列的状态遍历



1011

0100

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	9	-1	-1	13	-1	-1	-1	-1	0	-1	1	-1	1	-1	-1

基于队列的状态遍历

Path: 15, 6, 14, 2, 11, 1, 9, 0

从初始状态0到最终状态15的动作序列为：

农夫把羊带到北岸；

农夫独自回到南岸；

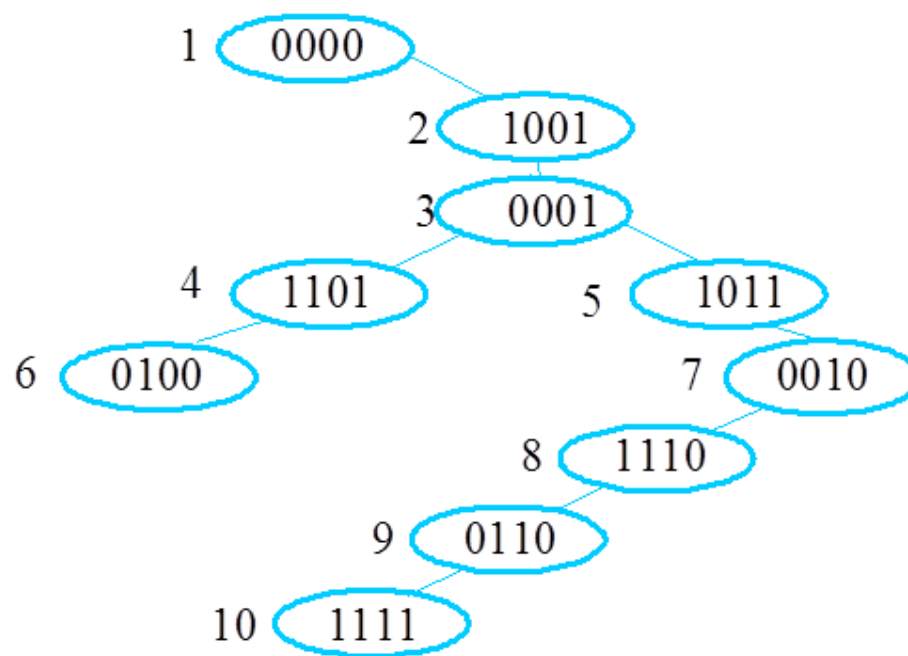
农夫把白菜带到北岸；

农夫带着羊返回南岸；

农夫把狼带到北岸；

农夫独自返回南岸；

农夫把羊带到北岸。



广度优先搜索的结果和顺序

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	9	11	-1	13	-1	14	-1	-1	0	-1	1	-1	1	2	6

角色状态

从四位二进制整数检测每个对象是否在北岸。

```
bool farmer(int location) {           1011
    return (0 != (location & 0x08));  1000
}
bool wolf(int location) {             1011
    return (0 != (location & 0x04));  0100
}
bool cabbage(int location) {          1011
    return (0 != (location & 0x02));  0010
}
bool goat(int location) {             1011
    return (0 != (location & 0x01));  0001
}
```

安全状态的判断

```
int safe(int location){ //状态安全则返回1
    if((goat(location)==cabbage(location))
        &&(goat(location)!=farmer(location)))
        return 0;    // 羊吃白菜

    if((goat(location)==wolf(location))
        &&(goat(location)!=farmer(location)))
        return 0;    // 狼吃羊

    return 1;        // 其他状态是安全的
}
```

农夫过河算法

准备一个数组**route**记录路径

准备一个队列**moveTo**记录多重选择

初始状态0 (0000) 加入**moveTo**和**route**

```
while (队列非空 && 还没到终止状态) {  
    state = DeQueue(moveTo) ; //出队当前状态  
    for (每个从state可以到达的状态newstate)  
        if (newstate安全且未访问过) {  
            EnQueue(moveTo, newstate) ;  
            route[newstate] = state;  
        }  
}  
输出结果
```


农夫过河算法

```
{初始化顺序表Route和安全队列moveTo;  
初始安全状态0 (0000) 入队列moveTo, Route[0] = 0,  
while (IsEmpty(moveTo) && (route[15]==-1)) {  
    DeQueue(moveTo, location); //出队当前安全状态  
    for (每个从location可以过渡到的状态newlocation)  
        //农夫(附带同侧物品) 移动  
        if (newlocation安全且未访问过)  
            EnQueue(moveTo, newlocation);  
}  
if (route[15] != -1) //已经到达终点了, 打印路径  
    for (location=15; location>=0;  
        location=route[location]) {  
        printf("The location is : %d\n", location);  
        if (location==0) return 1; }  
else printf("问题无解\n");  
}
```

```

int farmerProblem( ){
    int movers, i, location, newlocation;

    int route[16];          /*记录已考虑的状态路径*/
    for(i=0;i<16;i++)      route[i]=-1;
    SqQueue  moveTo;
    InitQueue(moveTo);
    EnQueue(moveTo, 0x00);

    route[0]=0;
    while(!QueueEmpty(moveTo) && (route[15]==-1)) {
        OutQueue(moveTo, location); /*得到现在的状态*/
        for(movers=1;movers<8;movers<=<1) {
            1) .....
        }
    }
    2) ...
}

```

```

for (movers=1;movers<8;movers<<=1) {
    /* 农夫总是在移动, 随农夫移动的也只能是在农夫同侧的东西 */
    if ((0!=(location & 0x08))
        ==(0!=(location & movers)))
    {
        newlocation=location^(0x08|movers);
        if (safe(newlocation)
            &&(route[newlocation]==-1) {
            route[newlocation]=location;
            EnQueue(moveTo,newlocation);
        }
    }
}
}

```

```
if(route[15]!=-1){ /* 打印出路径 */
    printf("The reverse path is : \n");

    for(location=15;location>=0;location=route[location])
    {
        printf("The location is :
                %d\n",location);
        if(location==0) return 1;
    }
}
else    printf("No solution.\n");
return 0;
}
```

队列的应用：农夫过河问题

- 思考

- 广度优先搜索换成深度优先搜索呢？
- 需要一个栈以回退到上级的状态，以便搜索上级状态的下一个子状态。