

第四章 栈和队列

基本内容：

- 栈

- 栈的实现

- 栈的应用

函数和递归调用

迷宫问题求解

- 队列

- 队列的实现

- 队列的应用

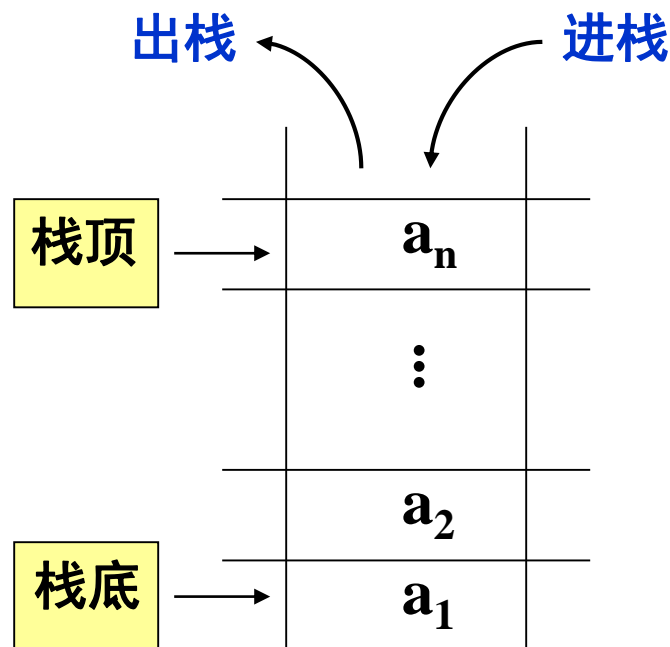
农夫过河问题求解

- ① 栈和队列是操作受限的线性表。这种操作限制主要体现在插入、删除操作的限制，普通的线性表的插入、删除可以在任何位置，而栈、队列的插入、删除只能在表头、表尾进行。
- ② 这种限制加快了插入、删除的速度（效率）
- ③ 栈、队列在系统软件和应用软件设计中应用广泛。

4.1 栈及其基本运算

1. 基本概念

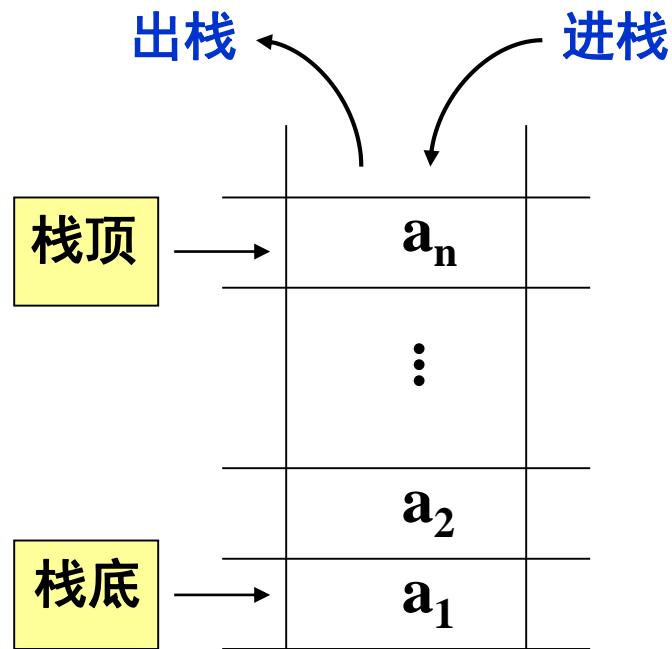
- **栈**：是限定在**表尾**进行插入和删除操作的线性表；插入、删除在表的同一端进行。
- **栈顶**：表尾端，元素从这端进行插入和删除。
- **栈底**：表头端。



- **栈的修改原则**：按**后进先出**（Last In First Out, LIFO）的原则进行插入、删除运算。
- **栈的插入**：进栈
- **栈的删除**：出栈

2. 栈的基本运算

- 创建一个空栈；
- 判断栈是否为空栈；
- 往栈中插入一个元素；
- 从栈中删除一个元素；
- 取栈顶元素的值。



4.2 栈的表示和实现

1. 顺序表示

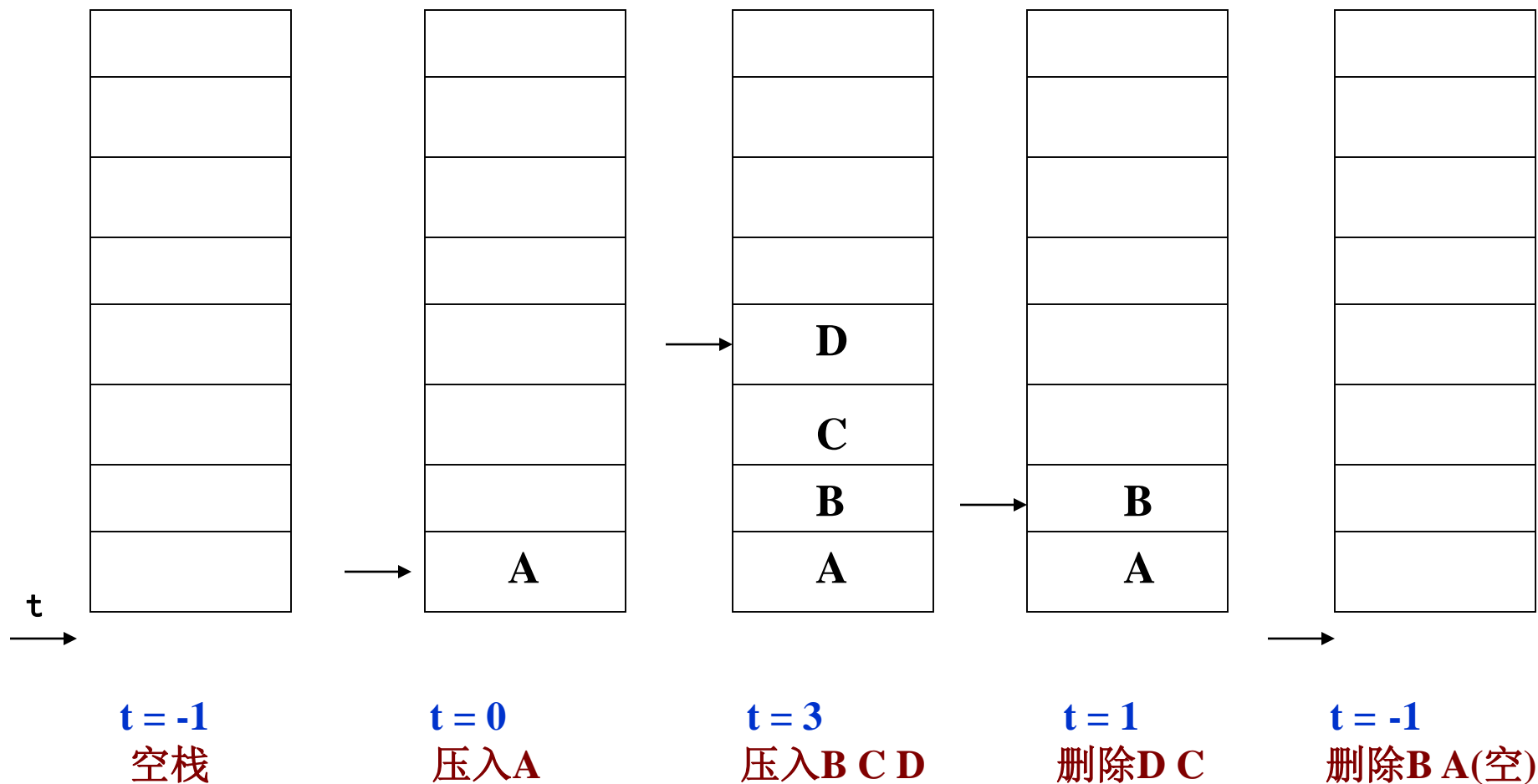
顺序栈类型定义:

```
#define MAXNUM 1000 /* 最大容量 */
struct SeqStack
{
    ElemType s[MAXNUM];
    int      t;      /* 指示栈顶位置，空栈=-1 */
}
typedef struct SeqStack, *PSeqStack;
PSeqStack pastack;
```

顺序实现

pastack->t: 栈顶指示变量
pastack->s: 存放栈元素的数组
pastack->s[pastack->t]: 栈顶元素

栈的插入、删除



栈满条件: $t = \text{MAXNUM} - 1$

栈空条件: $t = -1$

栈上溢条件: $t \geq \text{MAXNUM}$

栈下溢条件: $t < -1$

2. 链接表示

单链表结点结构

```
struct Node
{
    ElemType info;          /* 信息 */
    struct Node *link;      /* 下一个结点 */
};

typedef struct Node *PStackNode;
```

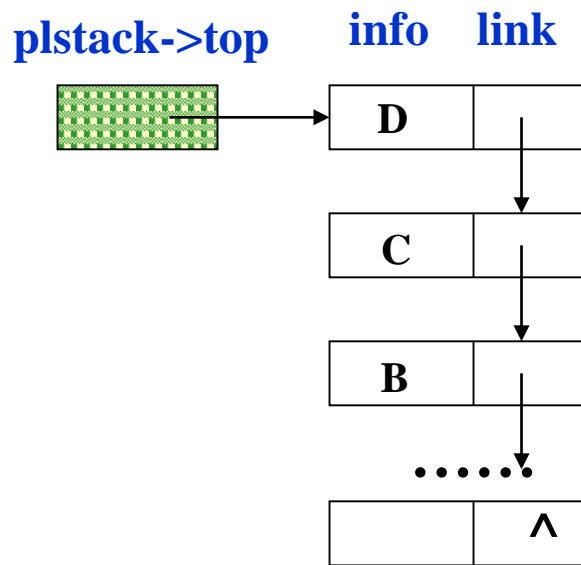
链接栈类型定义

```
struct LinkStack
{
    PStackNode top;        /* 栈顶指针，指向栈顶结点 */
};

typedef struct LinkStack *PLinkStack;

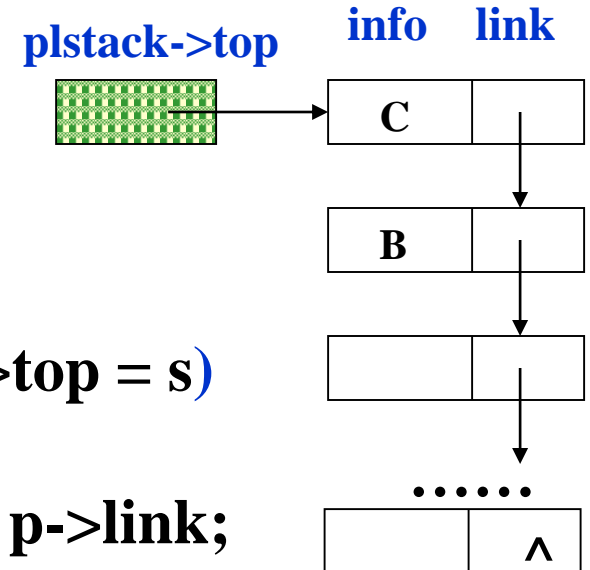
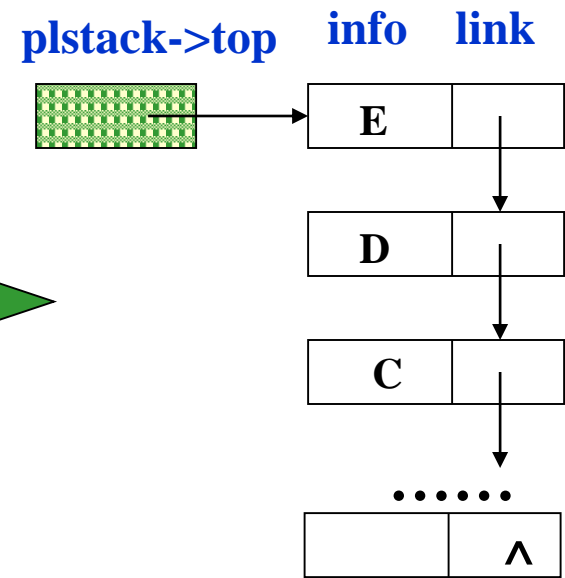
PLinkStack plstack;
```

链接实现



插入E

删除D



栈空条件: `plstack->top = NULL`

下溢条件: `if (plstack->top == NULL)`
还要继续删除

栈顶元素: `plstack->top`

压入元素: 插入s (新的栈顶)

(`s->link = plstack->top`, `plstack->top = s`)

弹出元素: 删除栈顶元素

(`p = plstack->top`; `plstack->top = p->link`;
`free(p)`;)

4.3 栈的应用举例

- 函数调用过程
- 递归
- 递归函数到非递归函数的转换
- 迷宫问题
- 表达式计算
- 数制转换

1. 函数调用的过程

调用：

- 1) 将所有的实参、返回地址传递给被调用函数保存
- 2) 为被调用函数的局部变量分配存储区
- 3) 将控制转移到被调用函数入口。

返回：

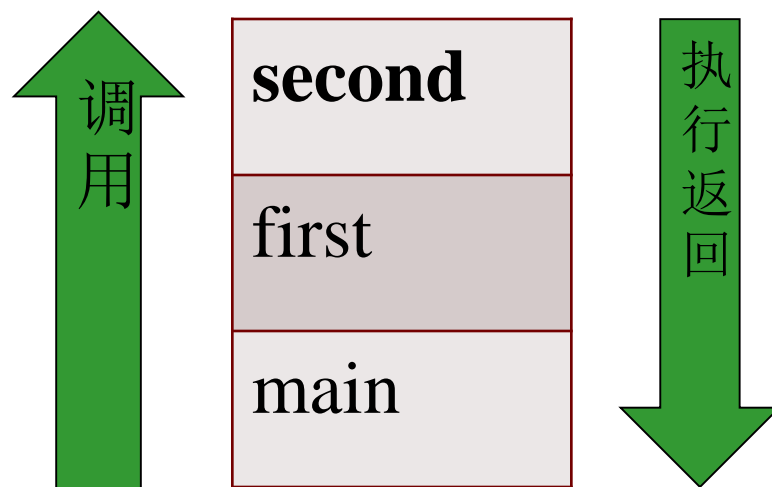
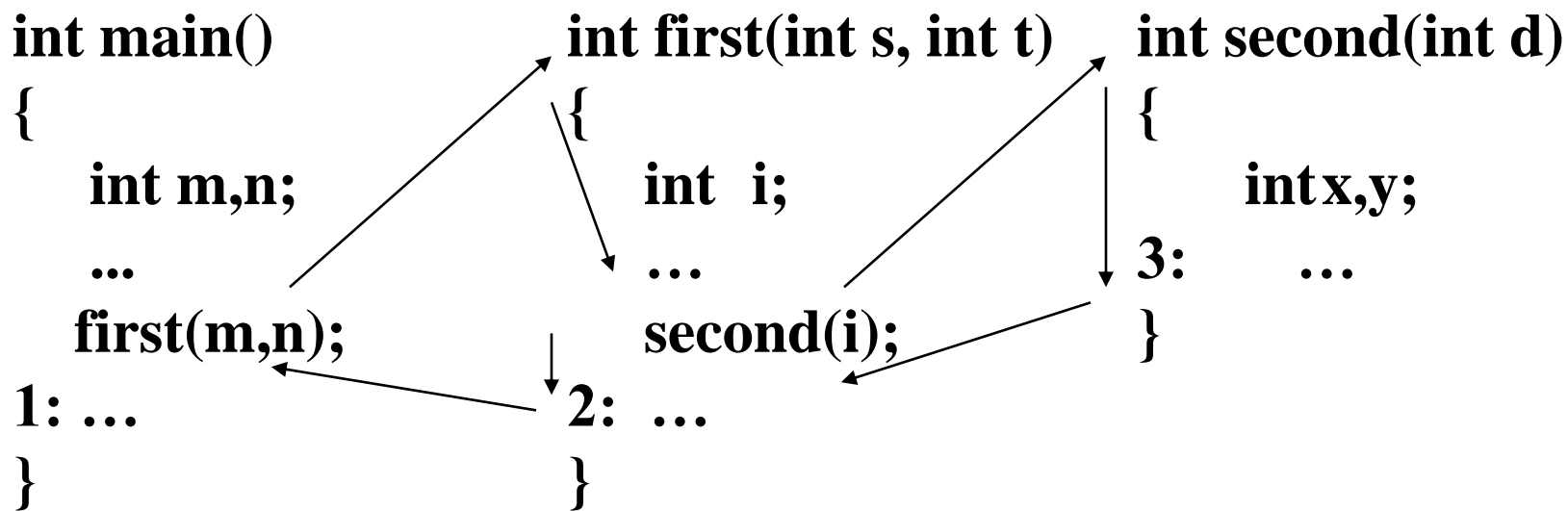
- 1) 保存被调用函数的计算结果。
- 2) 释放被调用函数的局部变量数据区
- 3) 按被调用函数保存的返回地址将控制转移到调用函数

多个函数嵌套调用时，按照“后调用先返回”的原则进行，如下所示：

```
int main()
{
    int m,n;
    ...
    first(m,n);
1: ...
}

int first(int s, int t)
{
    int i;
    ...
    second(i);
2: ...
}

int second(int d)
{
    int x,y;
    3: ...
}
```



如果改成用一个函数，则如下所示：

```
int main()
```

```
{
```

```
    int m,n;
```

```
    ...
```

```
    {
```

```
        int    i;
```

```
        ...
```

```
        {
```

```
            int x,y;
```

```
            3: ...
```

```
        }
```

```
        2: ...
```

```
    }
```

```
    1: ...
```

```
}
```

main

first

second

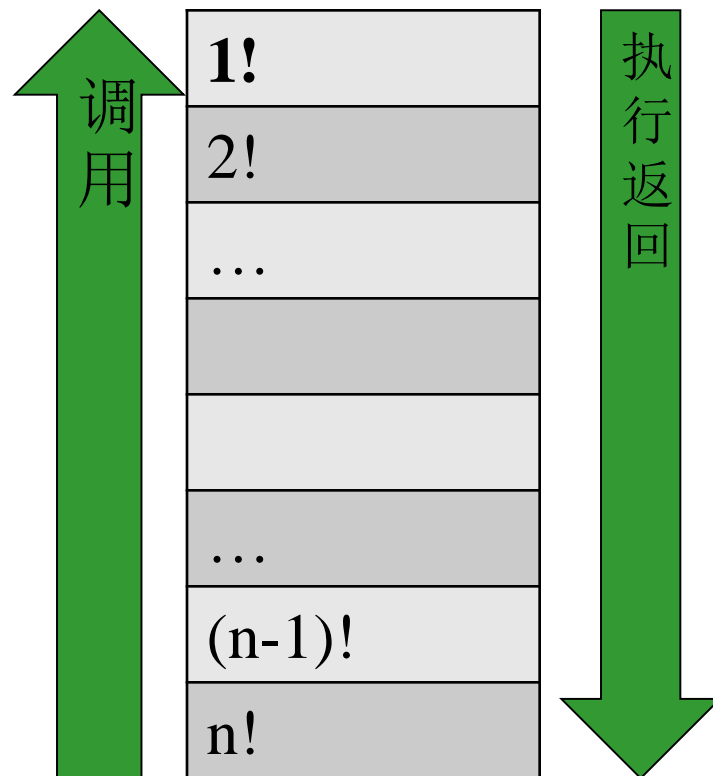
2. 递归的概念及递归调用过程

用自身的简单情况来定义自己的方式，称为“递归定义”（函数自己调用自己）。

(1) 阶乘： $\text{if } n=0, \text{ then } n! = 1$
 $\text{else } n! = n(n-1)!$

递归实现：

```
int factorial(int n)
{
    if(n == 1) return 1;
    else return(n*factorial(n-1));
}
```



(2) 简化的背包问题*

设有一个背包可以放入的物品重量为 s ，现有 n 件物品，重量分别为 w_1, w_2, \dots, w_n 。问能否从这 n 件物品中选择若干件放入此背包，使得放入的重量之和正好为 s 。

A. 分析问题，得到数学模型

$$\text{knap}(s, n) = \begin{cases} 1 & \text{当 } s = 0 \\ 0 & \text{当 } s < 0 \\ 0 & \text{当 } s > 0 \text{ 且 } n < 1 \\ \text{knap}(s, n-1) \text{ 或 } \text{knap}(s - w_n, n-1) & \text{当 } s > 0 \text{ 且 } n \geq 1 \end{cases}$$

不选择任何物品

无法实现

无物品可选

选择的一组物品中不包含 w_n

选择的一组物品中包含 w_n

B. 设计算法：递归算法

C. 程序设计：

```
int knap(int s,int n)
{
    if ( s == 0 )                return 1;
    else if ((s<0)||((s>0)&&(n<1))) return 0;
    else if ( knap(s - w[n-1],n - 1)==1 )
    {
        printf("result: n=%d ,w[%d]=%d \n",n,n-1,w[n-1]);
        return 1;
    }
    else return ( knap(s,n - 1) );
}
```

3. 递归函数到非递归函数的转换

(1) 阶乘的非递归计算

```
int nfact( int n )
{
    int          res;
    PSeqStack    st;  /* 使用顺序存储结构实现的栈 */
    st = CreateEmptyStack( );
    while (n>0)        /* 按照调用次序, 压栈 */
    {
        Push(st,n);
        n = n - 1;
    }
    res = 1;           /* 按照调用的反次序, 退栈 */
    while (! IsStackEmpty(st))
    {
        res = res * GetTop(st);
        Pop(st);
    }
    return ( res );
}
```

(2) 背包问题的递归函数到非递归函数的转换

设计栈，栈中的每个结点包含以下四个字段：参数s, n, 返回地址r和返回值k。

结点结构如下：

```
struct NodeBag      /* 栈中元素的定义 */
{
    int    s , n ;   /* 函数参数 */
    int    r ;       /* 返回地址，值为1,2,3 */
    int    k ;       /* 返回值 */
} DataType;
```


返回地址:

由于knap算法中有两处要递归调用knap算法，所以返回地址一共有三种情况：

- a) 计算 $\text{knap}(s_0, n_0)$ 完毕，返回到调用本函数的其它函数；
- b) 计算 $\text{knap}(s-w[n-1], n-1)$ 完毕，返回到本调用函数中继续计算；
- c) 计算 $\text{knap}(s, n-1)$ 完毕，返回到本调用函数继续计算。

为区分三种返回， r 分别用1, 2, 3表示。

转换的做法按以下规律进行：

凡调用语句knap(s1 , n1)均代换成：□

(1) **x.s = s1 ;**

x.n = n1 ;

x.r = 返回地址编号；

(2) **Push(st, x);**

(3) **goto 递归入口。** □

将调用返回统一处理成：□

(1) **x = Pop(st);**

(2) **根据x.r的值,进行相应处理：**

x.r = 1 , 返回；

x.r = 2 , 继续处理1；

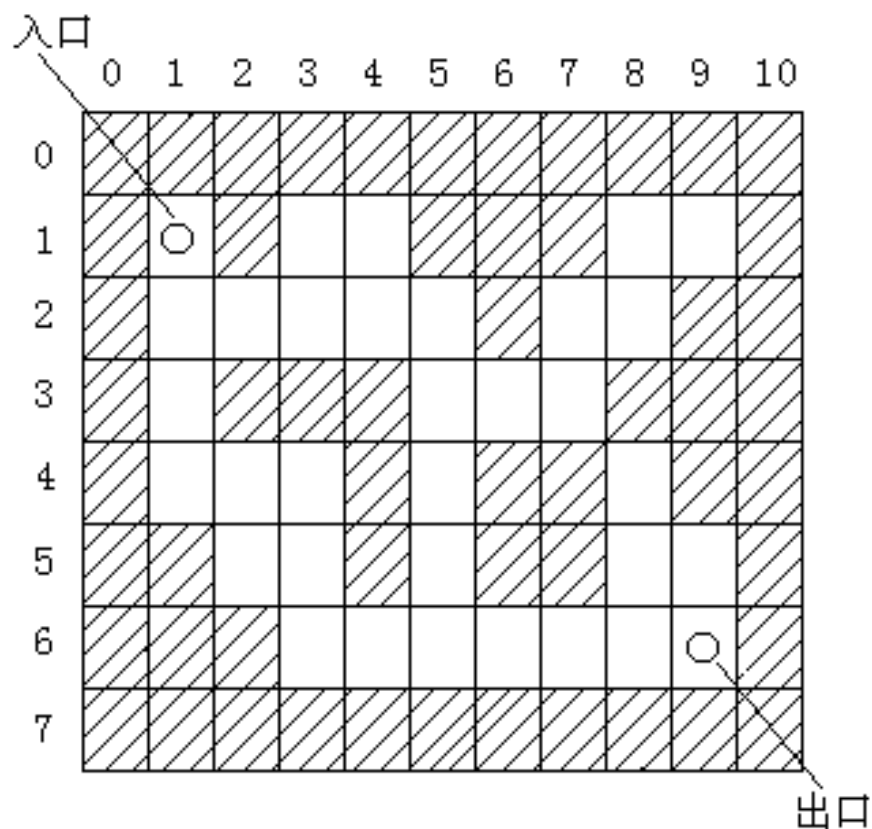
x.r = 3 , 继续处理2；

为便于对照，原递归算法改写成如下所示：

```
BOOL knap(int s,int n)  
{  
    if( s == 0 )  
        return TRUE;  
    else if( (s<0) || ((s>0) && (n<1)))  
        return FALSE;  
    else if(knap(s-w[n-1], n-1) == TRUE)  
    {    printf("result: n=%d ,w=%d\n",n, w[n-1]);  
        return TRUE;  
    }  
    else return(knap(s, n-1));  
}
```

背包问题的非递归算法实现如下：

4. 迷宫问题



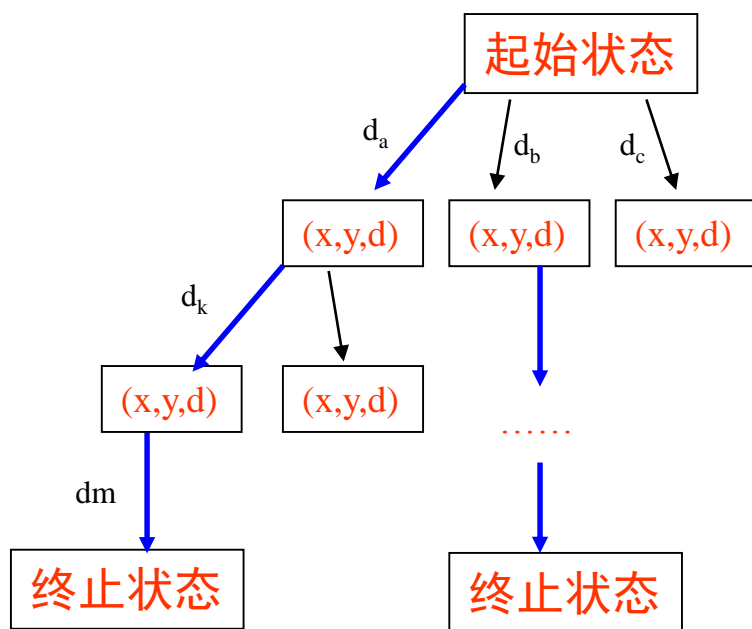
(a) 迷宫的图形表示

```
1 1 1 1 1 1 1 1 1 1 1
1 0 1 0 0 1 1 1 0---0 1
1 0-0-0-0-0 1 0---0 1 1
1 0 1 1 1 0---0---0 1 1 1
1 0 0 0 1 0 1 1 0 1 1
1 1 0 0 1 0 1 1 0 0 1
1 1 1 0 0 0-0-0-0-0 1
1 1 1 1 1 1 1 1 1 1
```

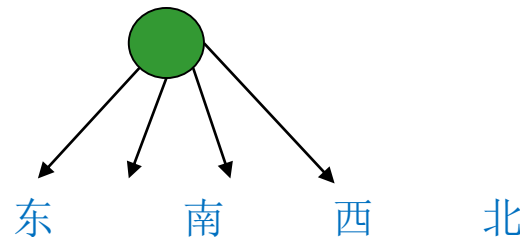
(b) 迷宫的二维数组表示

求解：

- (1) 数学模型：看成一个二维数组，如图(b)所示。
- (2) 算法设计：**回溯**算法：从入口出发，沿某一方向进行探索，若能走通，则继续向前走；否则沿原路返回一步，换一方向再进行探索，直到所有可能的通路都探索到为止。



中间是所有未探索过的、可通行的状态，可能存在多条路径。可探索时，记录分支信息，便于退回时继续其它分支探索。



深度优先的搜索策略！

探索过程：压栈过程

返回过程：弹栈过程

压栈信息：当前位置+下一步的方向

探索失败后返回，继续探索：

位置的下一个可能方向？

如无方向可走，继续弹栈。

如有方向可走，位置+方向压栈，继续探索。

直到栈空或找到出口为止。

mazeFrame()

{

创建一个（保存探索过程）的空栈

把入口位置压入栈

while（栈不空时）

{

取栈顶位置并设置为当前位置

while（当前位置存在试探可能：未试探的方向）

{

按照试探方向，计算试探位置

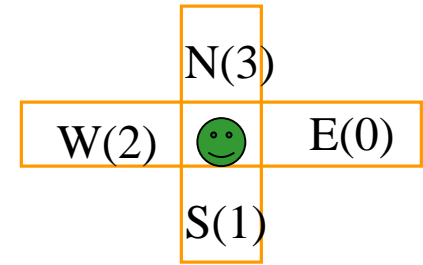
if（试探位置为出口） 打印结果，返回

if（试探位置为通道） 当前位置和试探方向进栈
并将试探位置设为当前位置

}

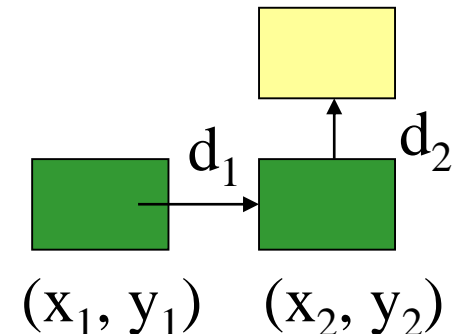
}

}



Direction[4][2]

0	1	0
1	0	1
2	-1	0
3	0	-1



(3) 程序设计

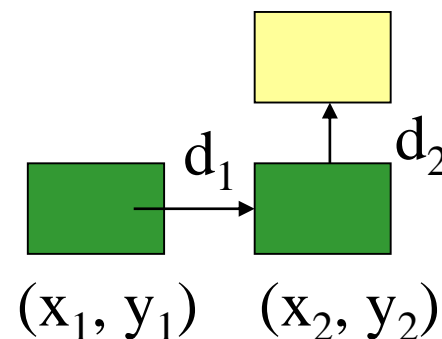
A. 为避免走回到已进入的点（包括已在当前路径上的点和曾经在当前路径上的点），凡是进入过的点都应做上特殊标记（原来只有0、1，可以2区分）。

B. 为记录**当前位置**及在该位置上所选的方向，设置一个栈，栈中每个元素包括三项，分别记录当前位置的行、列坐标及在该位置上所选的方向。如下：

```
struct NodeMaze  
{ int  x, y, d; } DataType;
```

C. 回溯用顺序栈实现

.....
(x_2, y_2, d_2)
(x_1, y_1, d_1)



5. 表达式计算

中缀表达式：运算符在操作数之间 $31 * (5 - 22) + 70$

后缀表达式：运算符在操作数后 $31\ 5\ 22\ -\ *\ 70\ +$

后缀表达式的求值：需要一个存放操作数的栈（碰到对应的运算符后才能输出）

过程：从左往右扫描表达式，

- 遇到操作数进栈；
- 遇到运算符时从栈中弹出两个操作数计算，并将计算的结果再压入栈。
- 扫描结束时，栈顶元素就是最后的结果。

$31\ 5\ 22\ -\ *\ 70\ +$

22
5
31

遇到-前，遇到-

-17
31

-527

遇到*

70
-527

70入栈

-457

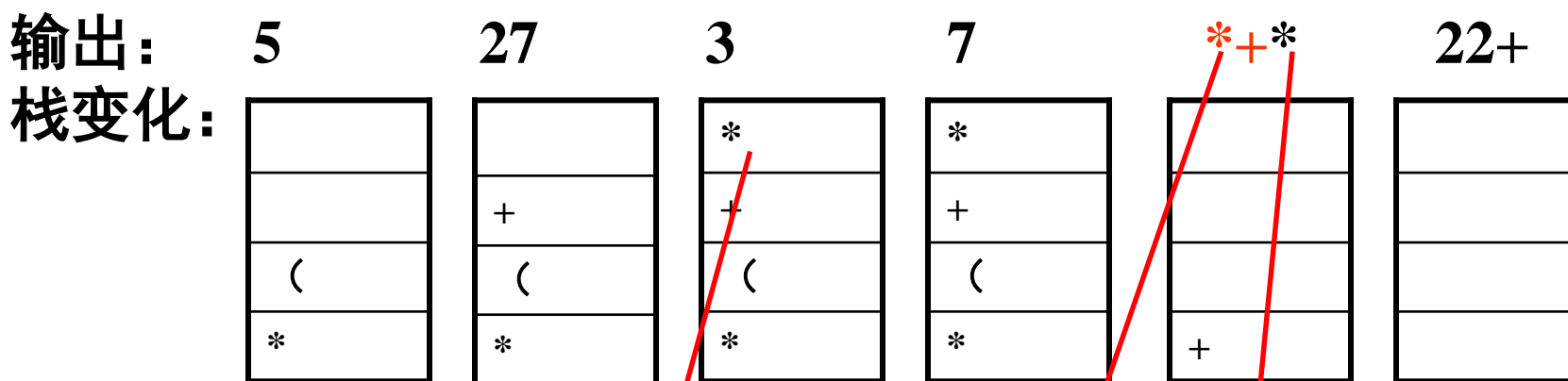
遇到+

中缀表达式到后缀表达式的转换：需要一个存放运算符的栈
(对应的操作数输出后才能输出)

过程：从左往右扫描表达式

- 遇到操作数：输出；
- 遇到运算符：【两个操作数输出后才输出】判断与栈顶运算符的优先关系，
高于：入栈；低于：弹栈输出栈顶运算符，压入当前运算符。【运算符优先级：四则运算】

注意：()特殊处理。碰到“(”，压栈；碰到“)”，弹出“(”上面的所有运算符【含“(”】。如：5*(27+3*7)+22



扫描：

*优先级
高于+

碰到)
输出*+(,

+优先级低于*

当扫描到左括号时立即推入栈，继续扫描直到出现右括号时才将留在栈中的这对左右括号之间的运算符逐一弹出输出。

十进制到m进制的转换1：

```
void main()
{   int v, m, i, j, result[1024];

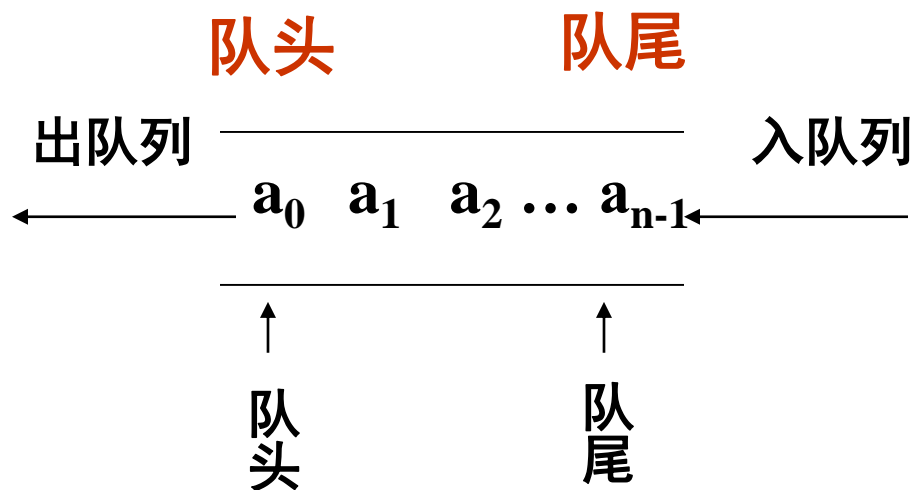
    //输入
    printf("输入进制:");
    scanf("%d", &m);
    printf("输入十进制数:");
    scanf("%d", &v);
    i = 0;
    while (v) //转换
    {   result[i++] = v%m; v /= m; }
    printf("输出结果:"); //输出
    for (j = i-1; j >= 0; j--)
        printf("%d ", result[j]);
    printf("\n");
}
```

十进制到m进制的转换2：

```
void main()
{   int v, m;
    pSeqStack *s = CreatStack();
    //输入
    printf("输入进制:");
    scanf("%d", &m);
    printf("输入十进制数:");
    scanf("%d", &v);
    while (v) //转换
    {   push(s, v%m); v /= m; }
    printf("输出结果:"); //输出
    while (!EmptyStack(s))
    {   v = pop(s); printf("%d ", v); }
    printf("\n");
    DestroyStack(s);
}
```

4.4 队列

1.定义: 是一种**先进先出**的线性表 (First In First Out, FIFO), 只允许在表的一端(队尾) 进行插入, 另一端 (队头) 进行删除。



a_0 先入队列, 接着 a_1, a_2, \dots, a_{n-1} 依次进入队列;
 a_0 先出队列, 后面的元素才能按照**先进先出**顺序出队列.

应用广泛: 作业、进程管理等

2. 队列的基本运算

- ✪ 创建一个空队列或将队列置成空队列；
- ✪ 判队列是否为空队列；
- ✪ 往队列中插入一个元素；
- ✪ 从队列中删除一个元素；
- ✪ 求队列头部元素的值。

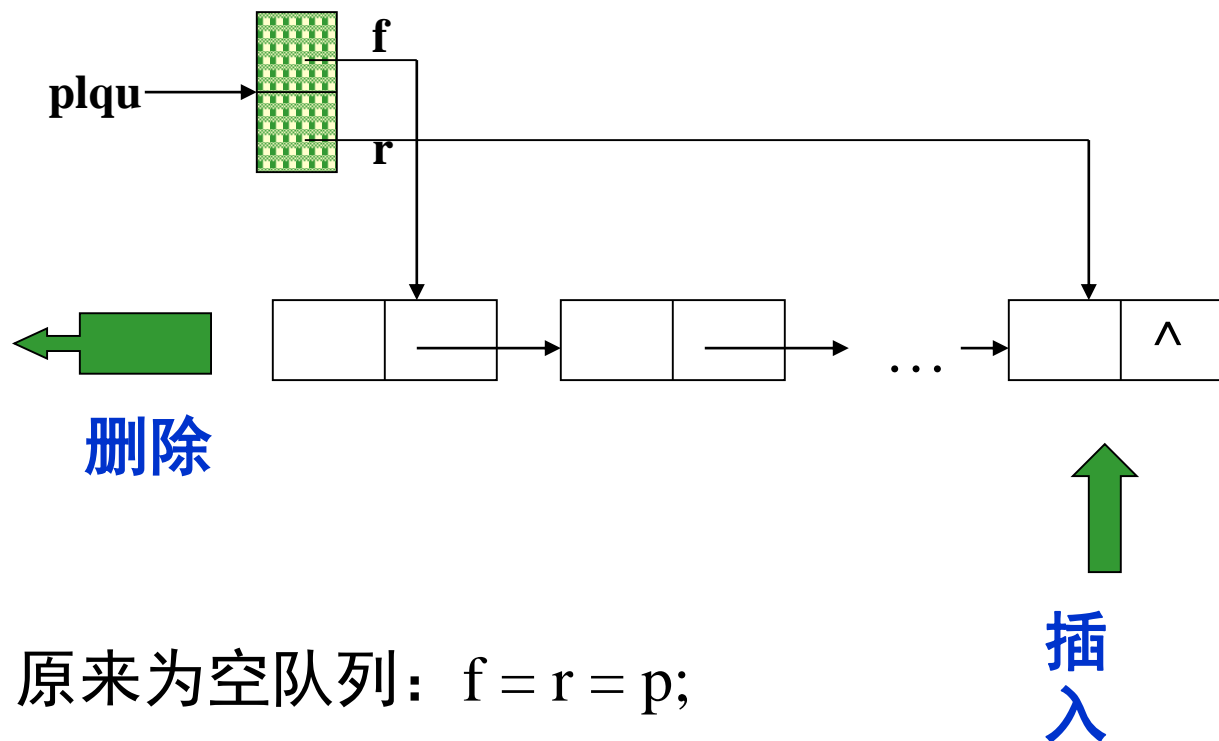
4.5 队列的实现

1. 队列的链式表示

```
struct QNode                /* 结点结构 */
{
    ElemType info;
    struct QNode    *link;
};
typedef struct QNode *PNode;
```

```
struct LinkQueue            /* 队列结构 */
{
    PNode f;
    PNode r;
};
typedef struct LinkQueue, *PLinkQueue;
```

队列的链式表示实现



插入: 原来为空队列: $f = r = p;$

否则: $r \rightarrow \text{link} = p; r = p;$

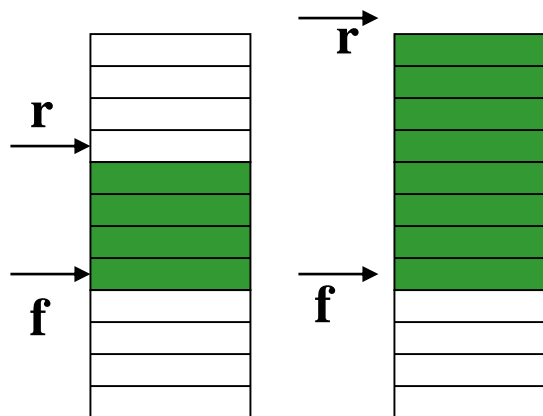
删除: $p = f; f = f \rightarrow \text{link}; \text{free}(p);$

2. 循环队列—队列的顺序表示和实现

(1) 普通顺序队列的缺陷

空间的浪费

(f指向队头, r指向队尾的下一个)



r达到顶, 不能再加入, 但实际上仍有空间。

==》假溢出

(2) 循环队列定义

```
struct SeqQueue    /* 顺序队列类型定义 */
```

```
{  
    ElemType    q[MAXNUM];  
    int         f, r;  
}
```

```
typedef struct SeqQueue *PSeqQueue;
```

队列满与队列空的判断问题【下一页】。

普通顺序队列:

f:将要删除的;

r:将要插入的;

Qu->q[qu->f]

Qu->q[qu->r]

空: ?

满: ?

(3) 循环队列实现

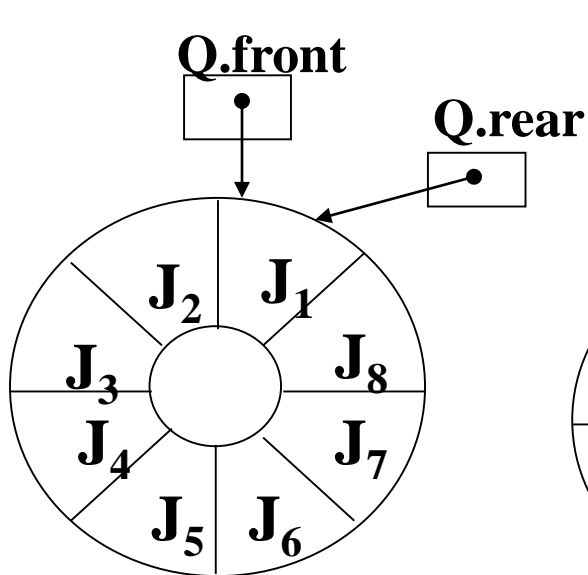


图1 队列满
(尾指针从后面追上头指针)

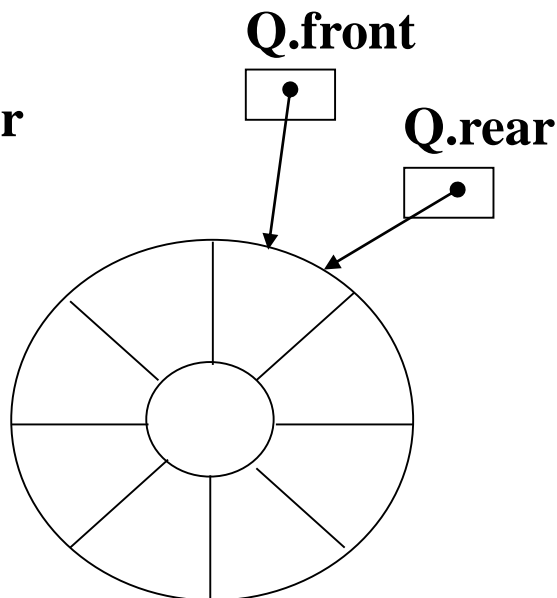


图2 队列空
(头指针追上尾指针)

空、满情况下：
队头 = 队尾
($Q.front = Q.rear$)
如何区分处理？

循环队列判空满条件：

- (1) 附设一个变量，当头尾相碰时判断头追上尾还是尾追上头。
- (2) 少用一个空间，当尾+1等于头时为满；当头等于尾时为空。

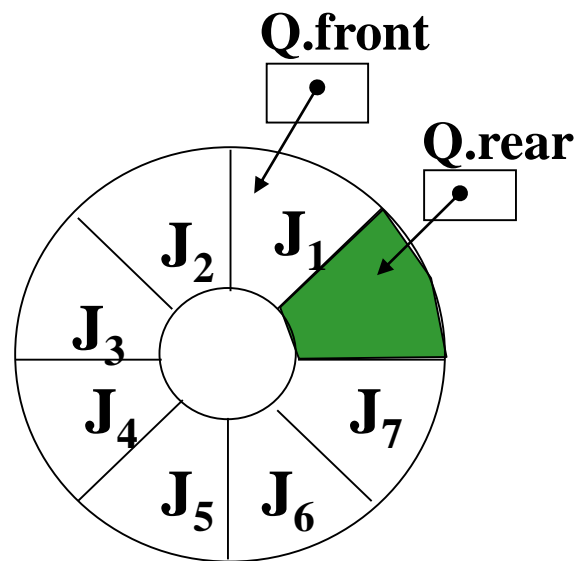


图3 策略2的队列满
判断 $Q.rear + 1 == Q.front$

int **status**; //出队列后: 1,入队列后: 2, 初始空队列: 1

Del:

f = (f+1)%n;

...

status = 1;

Add:

r = (r+1)%n

...

status = 2;

空判断: (f == r && status == 1)

满判断: (f == r && status == 2)

```
if (f == r)
{
    //空队列
    ;
}
```

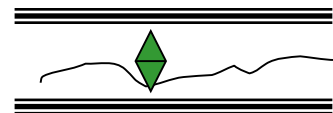
```
if ((r+1)%n == f)
{
    //满队列
    ;
}
```

4.6 队列的应用-农夫过河问题求解

农夫带一只狼、一只羊、一棵白菜过河，船小的每次只能带一样东西，由于狼能吃羊、羊能吃白菜，因此留下的两样东西不能任意组合。那么，农夫采用何种方案才能将所有东西不损害地运过河？



F+W+C+G



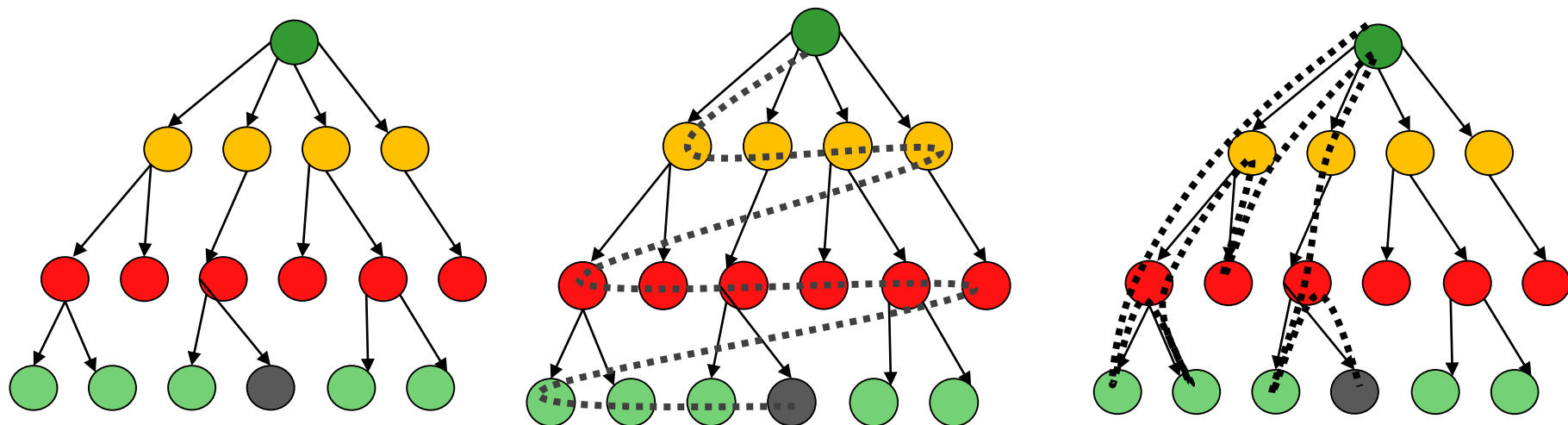
F+W+C+G

该问题的求解可以使用**试探法**，每一步都搜索所有可能的选择，对前一步合适的选择再考虑下一步的各种方案。

计算机求解可以有**两种不同的搜索策略**：

广度优先搜索：搜索该步的所有可能状态，再进一步考虑后面的各种情况；（队列应用）

深度优先搜索：沿某一状态走下去，不行再回头。（栈应用）

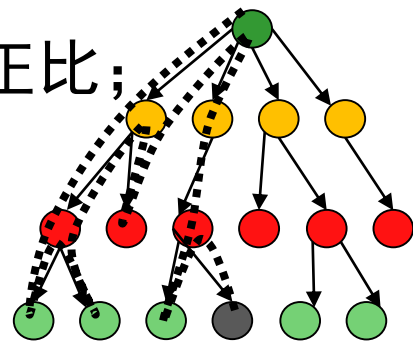


关于状态空间搜索问题

两种基本方式：深度优先搜索和广度优先搜索

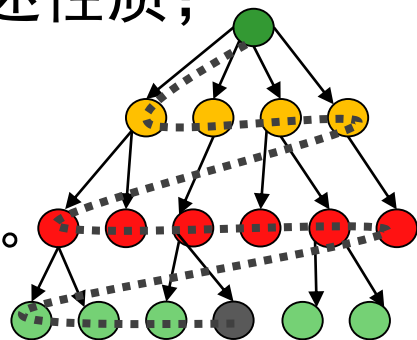
深度优先：在途径的每个分支点选一个分支前进，保留其它可能分支的信息，遇到死路时回溯。

- ① 用**栈**保留路途上的**分支信息**，可保证上述性质；
- ② 可能比较高效，需要保存的信息与路径长度成正比；
- ③ 可能陷入无穷路径而无法得到解。

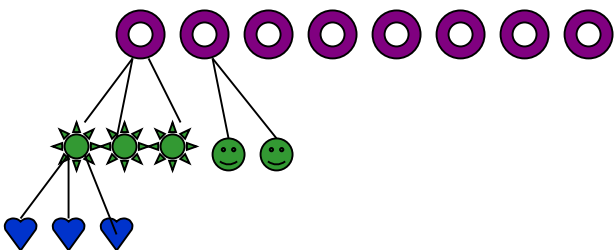
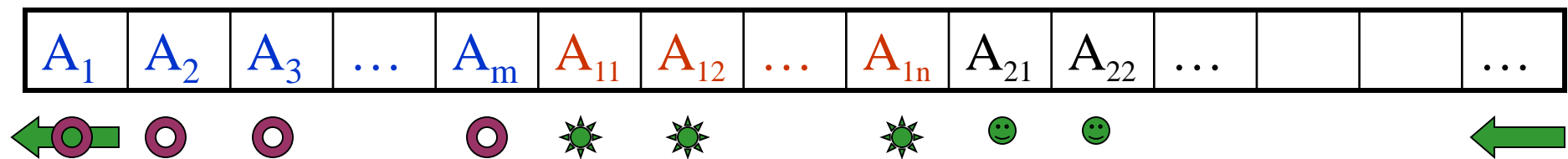


广度优先：按与初始点的距离，在所有可能路径上一步步齐头并进，在每个点探索所有下一步可能到达的点。

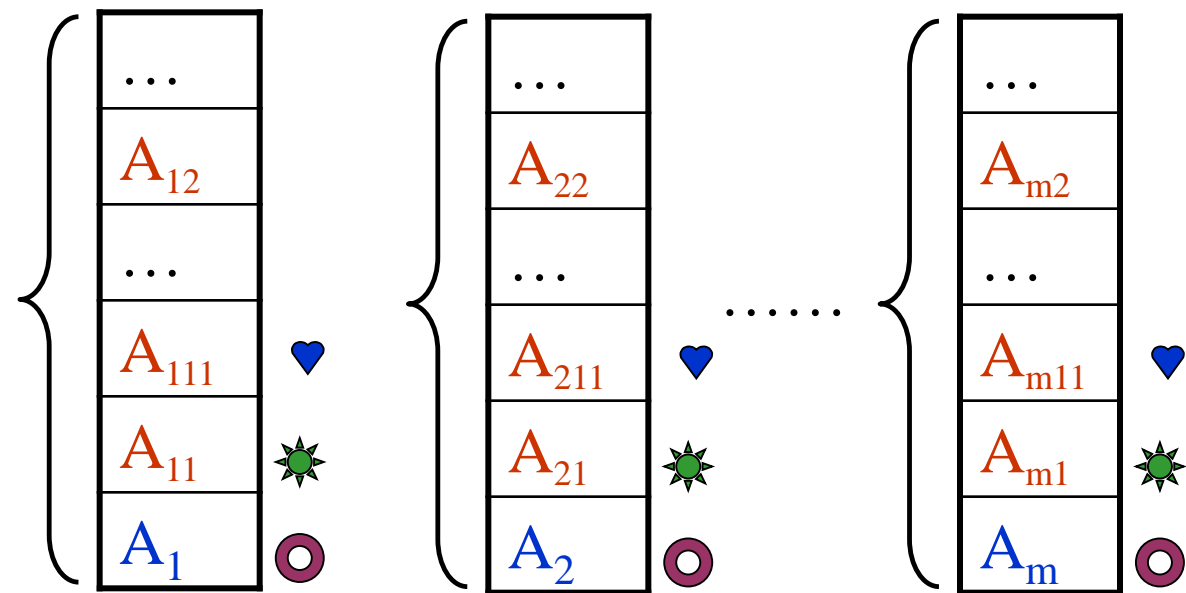
- ① 用**队列**保存还有探索可能的所有点，可保证上述性质；
- ② 可以保证找到的解是距离初始点**最近的解**；
- ③ 若路径长为 L ，点的平均分支数为 k ，要保存的信息与 k^L 成正比，可能膨胀很快（指数爆炸）。



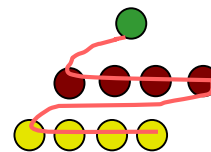
广度优先：(m种状态)



深度优先：(m种状态)



假定采用广度优先搜索解决农夫过河问题：

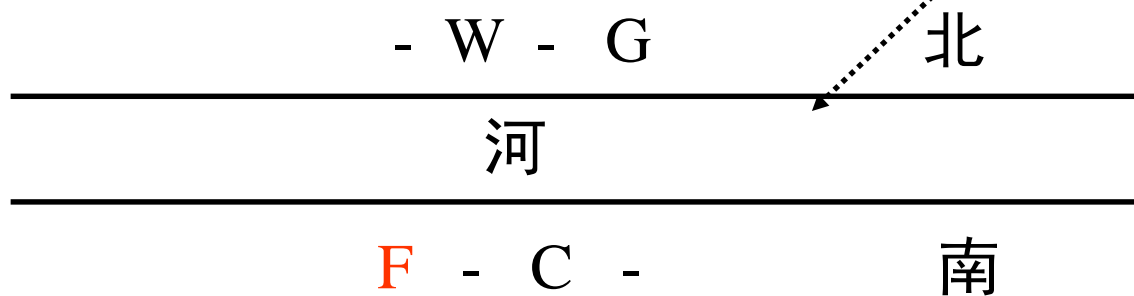


采用队列做辅助结构，把本层的所有状态都放在队列中，然后顺序取出对其分别处理，处理过程中再把下一层的所有可能状态放在队列中，……。由于队列的操作按照先进先出原则，因此只有前一层的所有状态都处理完后才能进入下一层。

描述方法：四个目标各采用一位表达，目标在河南：0，河北：1（假定按照农夫F、狼W、白菜C、羊G次序）。

任一时间的状态采用四位0-1码表示，如0101表示农夫、白菜在河南，而狼、羊在河北（此状态为不安全状态）

实现：



```

建立队列;
初始状态【0000】入队列;
while (队列非空时, 并且1111状态未到达)
{
    对头出列【获取一种状态】;
    考虑农夫过河动作导致的所有“安全的、尚未到达的”状态;
    【农夫过河, 农夫的状态改变; 农夫带的目标必须跟农夫在同岸】
    这些状态顺序入队列;
}

```

```

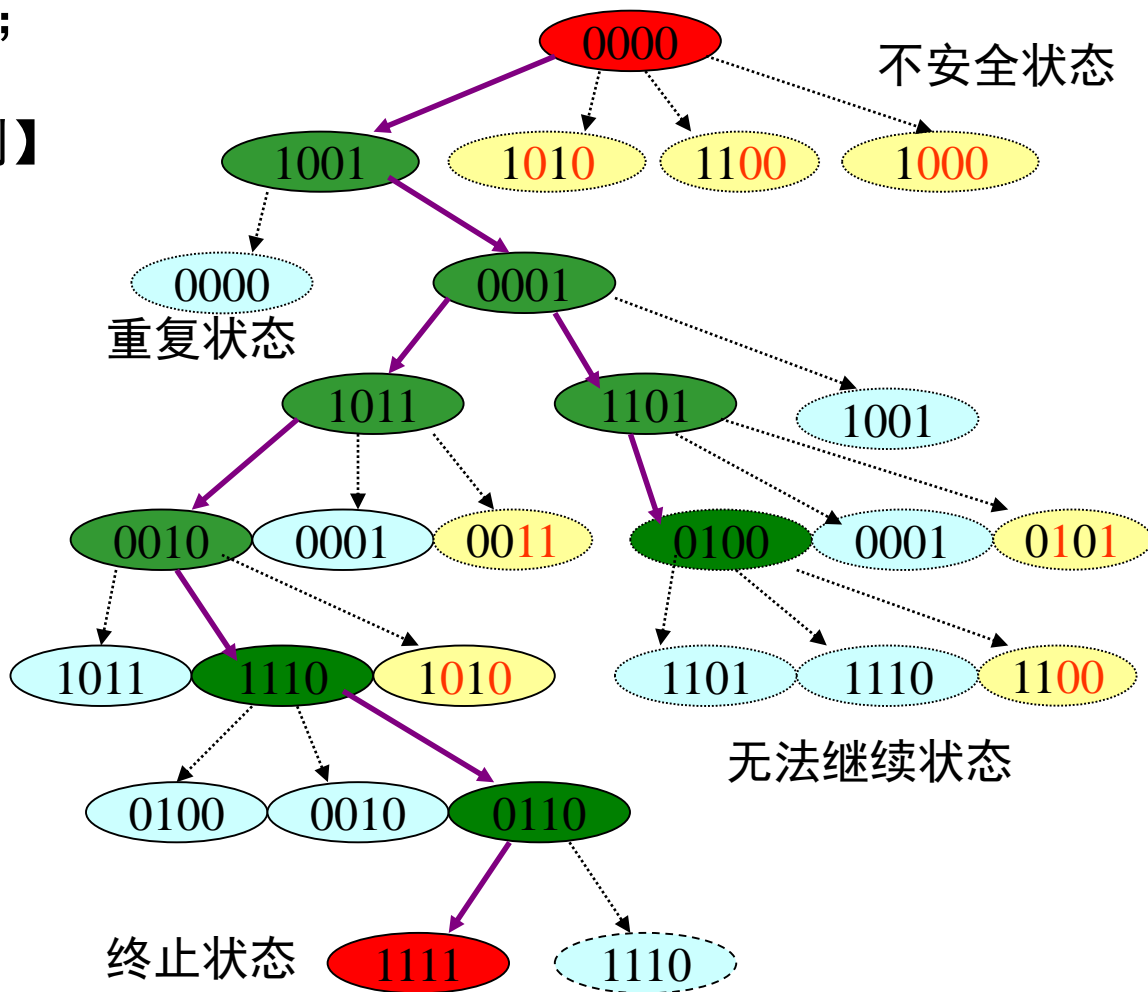
if (1111状态到达) 【打印序列】
else 【队列空, 无解】

```

队列变化过程

0000		
1001		
0001		
1011	1101	
1101	0010	
0010	0100	
0010	1110	
1110		
0110		
1111		

0000
1001
0001
1011
0010
1110
0110
1111



问题变为： 从0000状态出发，寻找全部由安全状态构成的状态序列，以1111为最终目标。状态序列中每个状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达。（序列中不能出现重复状态）

如何由某种状态提取所有可能的、安全的、未走过的状态？

目标位置判断：

```
int farmer(int location)
{ return (location & 0x08); }
int wolf(int location)
{ return (location & 0x04); }
int cabbage(int location)
{ return (location & 0x02); }
int goat(int location)
{ return (location & 0x01); }
```

```
int mover_location(int location, int mover)
{ return (location & mover); }
```

安全状态判断(返回1:安全, 0:不安全)

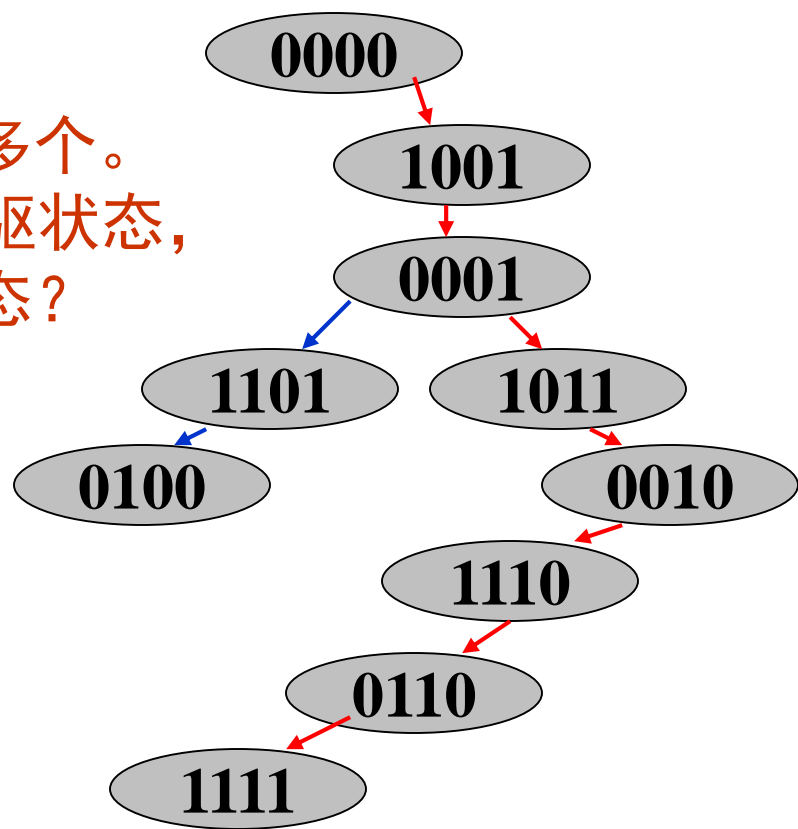
```
int safe(int location)
{
    if ((goat(location) == cabbage(location)) &&
        (goat(location) != farmer(location))) return(0); //羊吃白菜
    if ((goat(location) == wolf(location)) &&
        (goat(location) != farmer(location))) return(0); //狼吃羊
    return(1); //其它状态为安全
}
```


算法： moveTo顺序队列，存放可以安全到达的中间状态。

route顺序表，记录各个状态被访问过的情况（状态16种，因此使用包含16个元素的数组表示）：-1表示未被访问，否则记录前驱状态值的下标（表示该状态已经访问过，由前驱到达）。最后，（route[15]为非负值时）利用route建立正确的状态路径。

0000	-1
0001	9
0010	11
0011	
0100	
0101	
0110	14
0111	
1000	
1001	0
1010	
1011	1
1100	
1101	
1110	2
1111	6

前驱只有一个，后继可多个。
因此：route数组记录前驱状态，
即：此状态来自哪个状态？



实现： p109~112

结果： p113
15, 6, 14, 2, 11, 1, 9, 0

```

while (!IsEmptyQueue_seq(moveTo) && route[15] < 0)
{
    //得到现在的状态
    location = frontQueue_seq(moveTo);
    deQueue_seq(moveTo);
    //通过location状态由农夫过河引起的所有可能的安全且尚未走过的状态提取
    for (movers = 1; movers <= 8; movers << 1) //考虑所有可能的目标移动
    {
        //农夫总是在移动，随农夫移动的东西必须与农夫在同侧
        if (mover_location(location, movers) == mover_location(location, 0x08))
        {
            //新位置，农夫与要移动的东西同时过河【状态改变】
            newlocation = location ^ (0x08 | movers); //农夫+目标状态反转
            if (safe(newlocation) && route[newlocation] < 0)
            {
                //安全并且未访问过的状态【入队列】
                route[newlocation] = location;
                enQueue_seq(moveTo, newlocation);
            }
        }
    }
}

int mover_location(int location, int mover)
{ return (location & mover); }

```

假定采用深度优先搜索解决农夫过河问题：

采用栈做辅助结构，把某状态的所有后继“可行子状态”都放在栈中，如果某个状态无后继“可行子状态”且不是最终状态，退栈到上一层，进入上一层的其它“可行子状态”。如无，继续退栈，直到栈空或到达最终状态为止。

对于每个状态，只有其子状态都测试后才能进入下一可能状态。

描述方法：四个目标各采用一位（假定按照农夫、狼、白菜、羊次序），目标在河南位置：0，河北：1

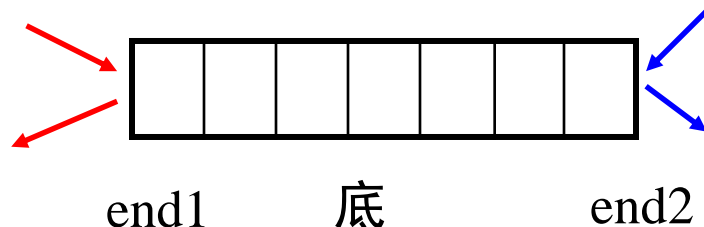
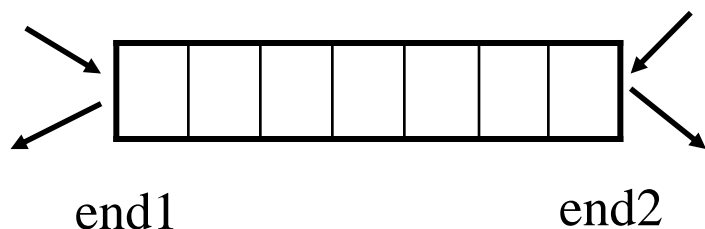
任一时间的状态采用四位0-1码表示，如0101表示农夫、白菜在河南，而狼、羊在河北（此状态为不安全状态）

实现：

4.7 限制存取点的表 *

1. 双端队列

双端队列是一种特殊的线性表，对它所有的插入和删除都限制在表的两端进行。

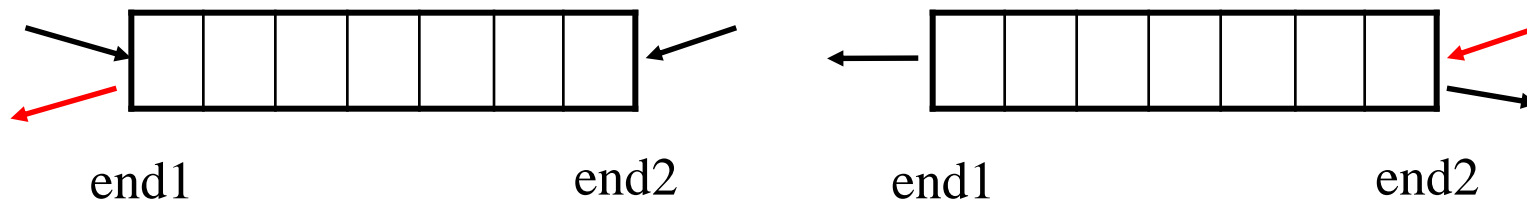


2. 双栈

双栈是一种加限制的双端队列，它规定从end1插入的元素只能从end1端删除，而从end2插入的元素只能从end2端删除。

3. 超队列

超队列是一种输出受限的双端队列，即删除限制在一端(例如end1)进行，而插入仍允许在两端进行。



4. 超栈

超栈是一种输入受限的双端队列，即插入限制在一端（例如end2）进行，而删除仍允许在两端进行。

小结

对于栈的顺序和链接表示以及在这两种表示方式下基本运算的实现应熟练掌握，特别要注意栈满和栈空的条件及描述；对于循环队列和链接队列及其基本运算应熟练掌握，特别要注意队列满和队列空的条件及描述。

作业

复习：P115： 1

书面作业： P115： 2、 7、 9

上机作业：

- 1) 迷宫问题的广度优先搜索求解；**
- 2) 农夫过河问题的深度优先搜索求解；**