



第3章 栈和队列

3.1 栈

3.2 栈应用举例

3.3 栈与递归函数

3.4 队列

3.5 队列应用举例

3.6 小结

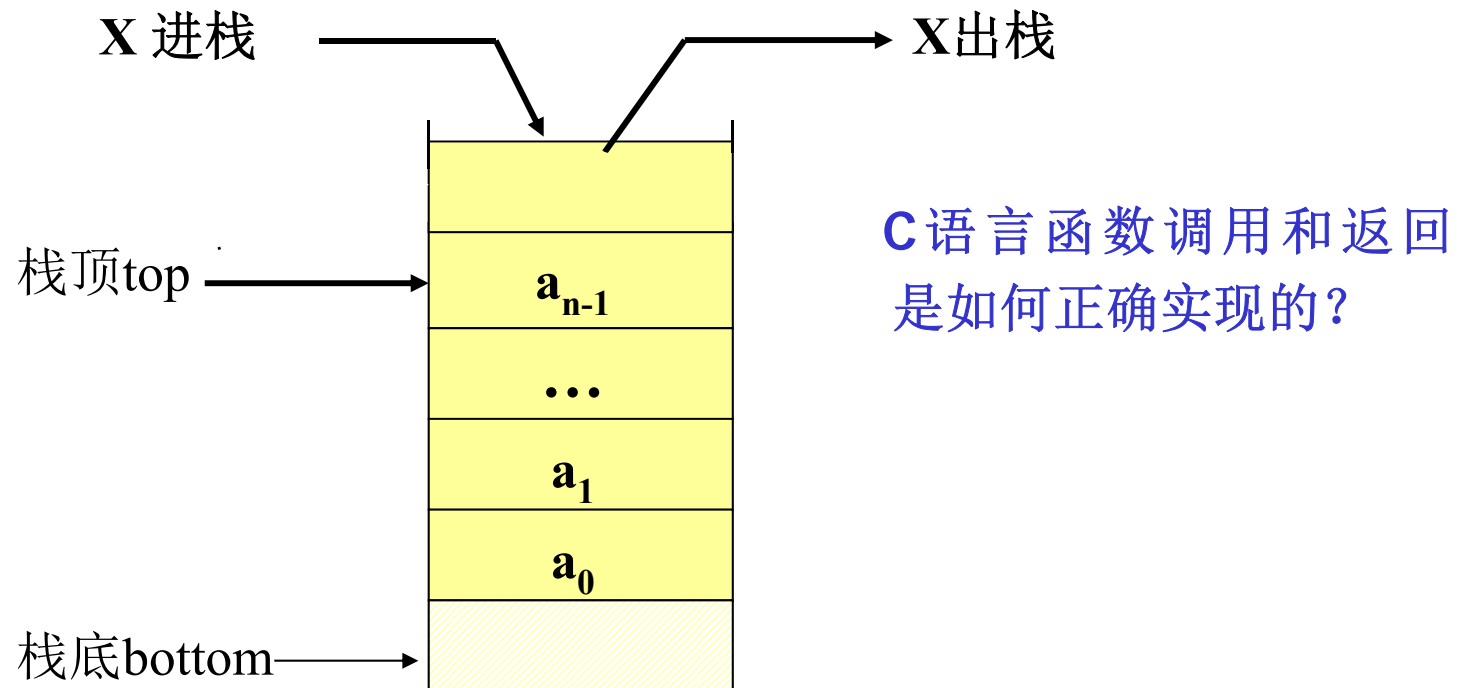
3.1 栈

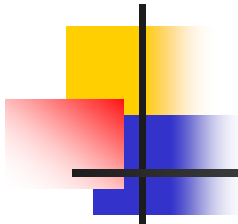
1. 栈的定义及其基本操作

定义：栈（stack）在逻辑上是一种线性表，记为

$$S = (a_0, a_1, \dots, a_{n-1})。$$

对栈S的操作（插入、删除等）限定在表的一端进行





栈的定义及其基本操作

栈的特点：

✓ 后进先出（**Last In First Out, LIFO**）

若元素进栈顺序为 a_0, a_1, \dots, a_{n-1} ，则出栈顺序是 $a_{n-1}, a_{n-2}, \dots, a_0$ ，即后进栈的元素先出栈，故栈可称作“后进先出”的线性表。

✓ 栈顶是唯一的出入口

当栈中没有元素时，称“栈空”。若栈的存储空间已满，再作进栈操作时称“栈满溢出”。

✓ 栈顶的位置随着进栈/出栈操作而发生变化



栈的定义及其基本操作

栈的抽象数据类型:

ADT Stack{

数据元素集: $D=\{a_i \mid a_i \in \text{datatype}, i=0,1,2, \dots, n-1, n \geq 0\}$

数据关系集: $R=\{ \langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D, 0 \leq i \leq n-2 \}$

约定 a_{n-1} 为栈顶元素

基本操作集: P

StackInit(&S)

操作结果: 创建一个空栈S。

ClearStack(&S)

初始条件: 栈S存在。

操作结果: 将S清为空栈。

EmptyStack(S)

初始条件: 栈S存在。

操作结果: 若S为空栈, 则返回TRUE(或返回1), 否则返回FLASE(或返回0)。



栈的定义及其基本操作

Push(&S, e)

初始条件：栈S存在且未滿。

操作结果：插入数据元素e，使之成为新栈顶元素。

Pop(&S)

初始条件：栈S存在且非空。

操作结果：删除S的栈顶元素并返回其值。

GetTop(S)

初始条件：栈S存在且非空。

操作结果：返回栈顶元素的值。

.

} **ADT Stack**;

3.1 栈

2. 栈的顺序存储结构

(1) 顺序栈的描述

```
#define MAXSIZE 64          //栈最大容量
typedef struct {
    datatype data[MAXSIZE]; //栈的存储空间
    int top;                 //栈顶指针（或游标）
} sqstack,*sqstack;        //顺序栈说明符
```

若说明 sqstack S;

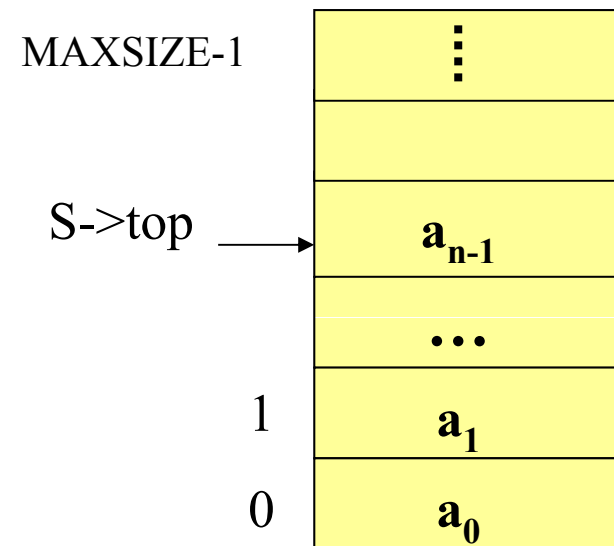
S = (sqstack)malloc(sizeof(sqstack));

则S指向一个顺序栈，如右图所示。

栈顶元素 a_{n-1} 写作：S->data [S->top]

栈空时S->top == -1

栈满时S->top == MAXSIZE - 1





栈的顺序存储结构

(2) 顺序栈基本操作的实现

- 1) void ClearStack (sqslink s) //置栈空
{ s->top = -1; }
- 2) int EmptyStack (sqslink s) //判断栈空
{
 if (s->top < 0) return 1; //栈空返回1
 else return 0; //栈非空返回0
}
- 3) int Push(sqslink s, datatype x) //x进栈
{
 if (s->top >= MAXSIZE - 1) return -1; //栈满溢出
 s->top++; s->data[s->top] = x; //x进栈
 return 0; //成功
}



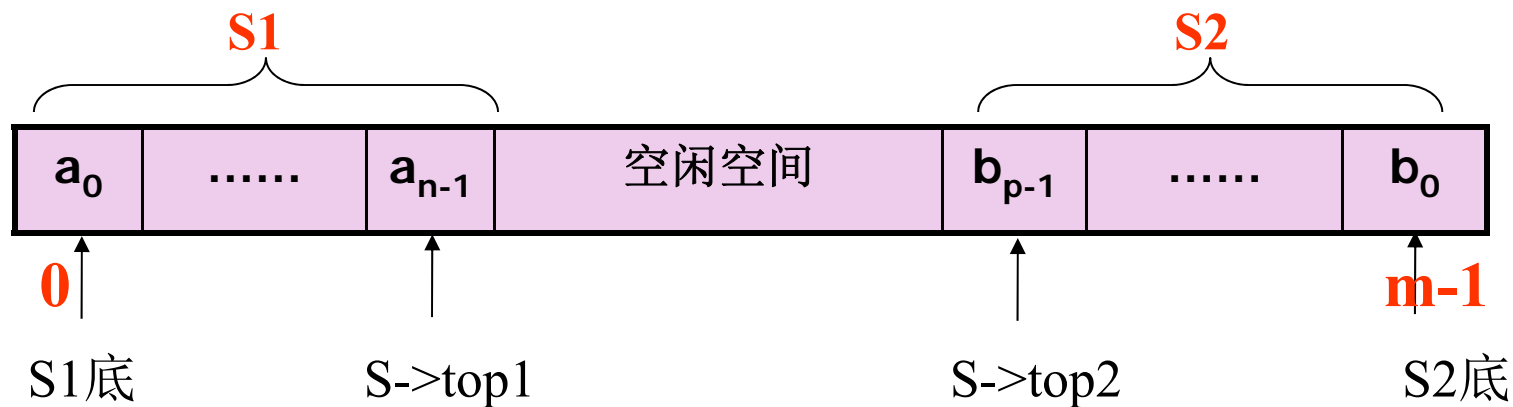
栈的顺序存储结构

```
4) int Pop(sqslink s, datatype *px) //出栈
{
    if (EmptyStack(s)) return -1; //栈空, 返回-1
    *px = s->data[s->top];
    s->top--;
    return 0; //成功
} // 为简单起见, 出栈操作常写成 x = Pop(s)

5) int GetTop(sqslink s, datatype *px)
{
    if (EmptyStack(s)) return -1; //栈空, 返回-1
    *px = s->data[s->top];
    return 0; //成功
}
```


栈的顺序存储结构

栈的共享:



x进栈 S_1 : if ($S->top2 - S->top1 \geq 2$) { $S->top1++$; $S->data[S->top1]=x$;}

栈 S_1 出栈: if ($S->top1 == -1$) return(NULL); else { $S->top1--$; return ($S->data[S->top1+1]$);}

x进栈 S_2 : if ($S->top2 - S->top1 \geq 2$) { $S->top2--$; $S->data[S->top2]=x$;}

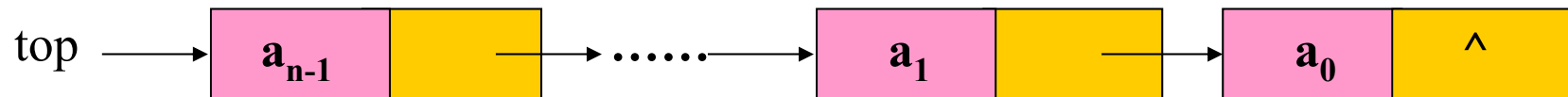
栈 S_2 出栈: if ($S->top2 == m$) return(NULL); else { $S->top2++$; return($S->data[S->top2-1]$);}

3.1 栈

2. 栈的链式存储结构

(1) 链式栈的描述

```
typedef struct node {  
    datatype data ;    //存储一个栈元素  
    struct node *next ; //后继指针  
} snode,*slink;
```



top为栈顶指针，后进栈节点的指针next指向先进栈节点，出栈时每次取top所指节点，满足栈的LIFO原则。

链式栈的基本操作如何实现？ **ClearStack, Push, Pop, EmptyStack**



栈的链式存储结构

(2) 链式栈基本操作的实现

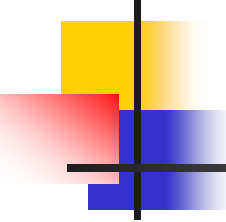
- 1) Lclearstack(slink *ptop) //置栈空, 略去了栈非空时释放空间的操作
 { *ptop = NULL; } //调用方式: **Lclearstack(&top);**
- 2) int Lemptystack(slink top) //判断栈空否
 { if (top == NULL) return 1;
 else return 0;
 }
- 3) Lpush(slink *ptop, datatype x) //进栈
 { slink p = (slink)malloc(sizeof(snode)); //生成进栈p节点
 p->data = x;
 p->next = *ptop; *ptop = p; //p节点作为新的栈顶链入
 } //调用方式: **Lpush(&top, x);**



栈的链式存储结构

```
4) int Lpop(slink *ptop, datatype *px)  //出栈
{ if (Lemptystack(*ptop)) return -1; //栈空返回
  *px = (*ptop)->data; //取栈顶元素
  slink p = *ptop;
  *ptop = (*ptop)->next; //重置栈顶指针
  free(p); return 0; //成功
} // 调用方式: Lpop(&top, &x);
```

```
5) int Lgettop(slink top, datatype *px)
{ if (Lemptystack(top)) return -1; //栈空返回
  *px = top->data; //取栈顶元素
  return 0; //成功
}
```



3.1 栈

3. 顺序栈与链式栈的比较

- ✓ 时间复杂度：均为 $O(1)$
- ✓ 顺序栈长度固定，浪费空间
- ✓ 链式栈有结构性开销
- ✓ 可以用数组实现2个栈，每个栈从各自的端点向中间延伸



3.2 栈应用举例

1. 数制转换

10进制正整数N与d进制数的基数d之间满足关系：

$$N_{10} = \underbrace{(N/d)}_{\text{整除}} * d + \underbrace{N \% d}_{\text{取模 (取余数)}}$$

10进制正整数N如何转换为二进制数？

10进制正整数N转换为d进制数的方法：

除以d取余数（直到商为0），逆序排列。



数制转换

算法描述:

```
int x = N;
ClearStack(S); //置栈空
while (x > 0) {
    push(S, x % d); //当前x%d进栈
    x = x/d;
}
while (!EmptyStack(S)) {
    x = Pop(S);
    printf("%d", x);
}
```



3.2 栈应用举例

2. 表达式括号匹配检验（行编辑处理）

设待检验的表达式存入字符型数组E[n]中，如：

E[0]	E[1]	E[2]	E[3]	E[4]	E[5]	E[6]	E[7]	E[8]	E[9]	E[10]	E[11]	E[12]
A	[(I	-	2)	*	4]	+	3	#

表达式括号不匹配：

([])] ; (([]) ; (] 或 [).



表达式括号匹配 — 算法描述

```
int bdsxgs(char E[n]) //括号匹配算法
{ int i=0; char x; stype S;
  Clearstack(S);
  while (E[i]!='#')
  { if (E[i]=='(' || E[i]=='[') push (S, E[i]); // (, [ 进栈
    if (E[i]==')' || E[i]==']')
    { if (Emptystack(S)) return(0);           //不匹配,返回0
      else {x=pop(S);                         //出栈, x为相应左括号
            if (x=='(' && E[i]==']' || x=='[' && E[i]==')')
              return(0);}                     //不匹配返回0
    }
    i++;}
  if (Emptystack(S)) return(1);               //括号匹配, 返回1
  else return(0); }                          //不匹配返回0
}
```



3.2 栈应用举例

3.表达式求值

(1) 表达式的形式

1) 中缀表达式: $\langle \text{操作数1} \rangle \langle \text{操作符} \rangle \langle \text{操作数2} \rangle$

如 $A + B$

2) 后缀表达式: $\langle \text{操作数1} \rangle \langle \text{操作数2} \rangle \langle \text{操作符} \rangle$, 或称逆波兰式

如 $A B +$

3) 前缀表达式: $\langle \text{操作符} \rangle \langle \text{操作数1} \rangle \langle \text{操作数2} \rangle$, 或称波兰式

如 $+ A B$

例 中缀表达式: $A + (B - C / D) * F$

后缀表达式: $A B C D / - F * +$

前缀表达式: $+ A * - B / C D F$



表达式求值

(2) 后缀表达式求值

后缀表达式计算不存在优先级问题。例如

$A B C D / - F * +$

如何计算？

方法：

从左到右扫描，当遇到操作符时，对其左边的2个操作数进行计算，以结果取代之，继续。



表达式求值

算法思路:

设置一个栈S;

从左到右依次扫描表达式中的各分量x:

若x是操作数: $\text{Push}(S, x)$;

若x是操作符: $a = \text{Pop}(S)$; $b = \text{Pop}(S)$; $\text{Push}(b \times a \text{ 的计算结果})$;

继续, 直到表达式扫描完毕;

最后, 栈顶就是表达式的值。



表达式求值

(3) 中缀表达式到后缀表达式的转换

如何转换?

$A + B$

当扫描到+时记录下来, 得 $AB+$

$A + B - C$

当扫描到+时记录下来;

当扫描到-时, -的优先级不大于+, 说明+应先计算, 则得 $AB+$, 最后得 $AB+C-$

$A + B * C$

当扫描到+时记录下来;

当扫描到*时, 仍然只能先记录下来, 因为不知道后面有无优先级更高的, 最后得 $ABC*+$

采用什么数据结构记录操作符? 栈!



中缀表达式到后缀表达式的转换

再如：

$A + B - C * D$

$A + (B - C / D) * F$

转换方法要点：

- ✓ 中缀表达式与后缀表达式操作数出现的次序是相同的
- ✓ 需要考虑操作符的优先级，优先级高的先转换
- ✓ 从左到右扫描表达式，将遇到的操作数输出
- ✓ 设置一个栈存放操作符
- ✓ 遇到的第1个操作符进栈
- ✓ 在扫描过程中，将遇到的操作符与栈顶进行优先级比较
- ✓ 栈中的操作符满足条件：后进栈的优先级高于先进栈的
- ✓ 当扫描完毕时，若栈非空，则将栈中操作符依次出栈输出



中缀表达式到后缀表达式的转换

方法:

从左到右扫描表达式, 设置一个栈s存放操作符;

对于遇到的每个分量x, 分以下几种情况处理:

1) x = 操作数: 输出x;

2) x = '(': x进栈;

3) x = 操作符 (非括号):

```
while (1) {  
    if (EmptyStack(s)) break;  
    y = GetTop(s); if (y = '(') break;  
    if (y 优先级 < x) break;  
    y出栈并输出;  
}
```

x进栈;

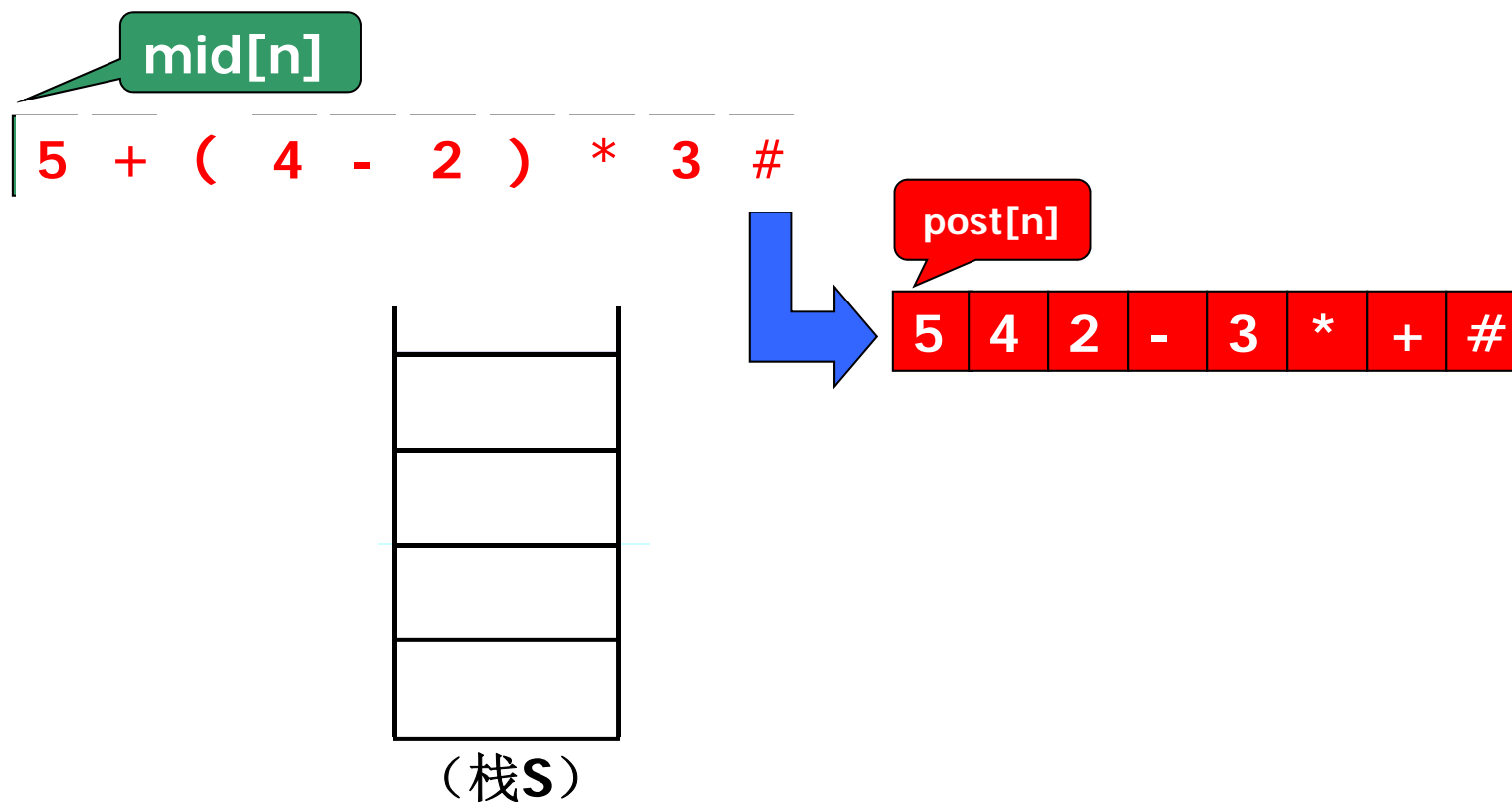
4) x = ')': //之前进栈的 '(' 与 ')' 之间的操作符应先计算!

反复出栈, 输出出栈的操作符, 直到遇 '(', 退掉;

当扫描完毕时, 若栈非空, 则将栈中内容依次出栈输出;

中缀表达式到后缀表达式的转换

例：设中缀表达式已存入数组mid[n]中，相应后缀表达式存入数组post[n]中。



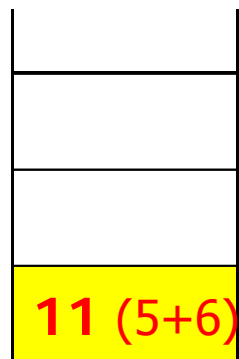
后缀表达式求值

对后缀表达式 $\text{post}[n] = "5\ 4\ 2\ -\ 3\ *\ +\ \#"$ 的求值，栈S的变化状态如图

post[n]

5	4	2	-	3	*	+	= 11
---	---	---	---	---	---	---	------

(结果)



(栈S)



3.3 栈与递归函数

1. 递归的定义和递归函数

(1) 递归定义

对于定义：<定义对象> = <定义描述>

(左部) (右部)

若定义的右部又出现定义的左部形式，则称为递归定义



递归的定义和递归函数

例 n 的阶乘定义:

$$n! = \begin{cases} 1 & \text{当 } n = 0 \text{ 时} \\ n * (n-1)! & \text{当 } n > 0 \text{ 时} \end{cases}$$

例 Fibonacci数列定义:

$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{当 } n = 0 \text{ 时} \\ 1 & \text{当 } n = 1 \text{ 时} \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) & \text{其他} \end{cases}$$

例 Ackermam 函数定义:

$$\text{Ack}(m, n) = \begin{cases} n + 1 & \text{当 } m = 0 \text{ 时} \\ \text{Ack}(m-1, 1) & \text{当 } n = 0 \text{ 时} \\ \text{Ack}(m-1, \text{Ack}(m, n-1)) & \text{其他} \end{cases}$$



递归的定义和递归函数

(2) 递归函数（或过程）

直接或间接调用自己

- ✓ 直接递归
- ✓ 间接递归

```
#include <stdio.h>
int f(int n);
int g(int n);
int main(){
    int i;
    system("cls");
    for(i=1;i<6;i++)
        printf("f=%d,g=%d\n",f(i),g(i));
    system("pause");
    return 0;
}
int f(int n)
{
    if(n==1)return 1;
    else
        return n+g(n-1);
}

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



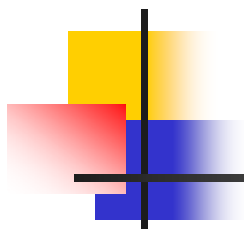
递归的定义和递归函数

(3) 递归的基本思想

- ✓ 一个大问题分解为几个子问题，其中的某些子问题与原问题相同，只是规模比原问题小；
- ✓ 随着问题的不断分解，一定存在一个最小问题，可以直接解决，这便是递归出口（即终止条件）。

说明：

- ✓ 对于本身就是递归定义的问题，用递归最容易实现。用非递归反而较难。
- ✓ 任何循环均可以递归的方式来实现。当然，与循环一样，递归函数也必须存在一个终止条件。
- ✓ 递归过程（函数）设计的关键：保证除了出口参数外，每次调用都不破坏以前调用时所用到的参数和中间结果。



递归的定义和递归函数

例:

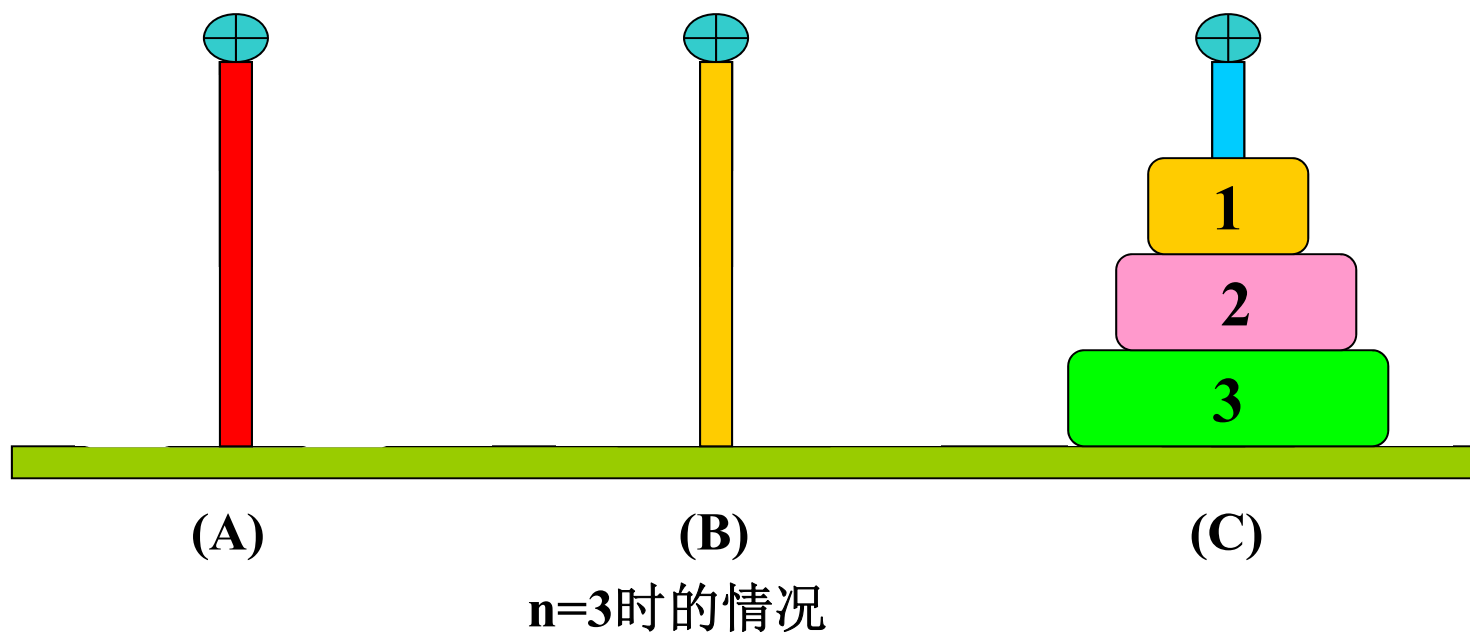
```
int Fact (int n)  //求n! (只能用于小规模n, 这里只是为了说明递归)
{
    if (n == 0) return 1;
    return n*Fact(n-1);
}
```

```
int Ack (int m, int n)  //求Ackermam函数
{
    if (m == 0) return n+1;
    else if (n == 0) return Ack (m-1,1);
    else return  ACK (m-1, Ack (m, n-1));
}
```

递归的定义和递归函数

例 3.10 hanoi塔问题：设有A、B、C三塔（或柱子）， $n(n \geq 1)$ 个直径不同的盘子依次从小到大编号为1, 2, ..., n 存放于A塔中。

- 要求将A塔上的1到 n 号盘移到C塔，B塔作为辅助塔。
- 移动规则：每次只能移动一个盘从一塔到另一塔，且任何时候编号大的盘不能在编号小的盘之上。



1->C, 2->B, 1->B, 3->C, 1->A, 2->C, 1->C



hanoi塔 问题

如何分析设计这类递归问题的算法？

1) 最小问题: $n=1$ 。可直接解决（从A移到C即可）

2) 否则 ($n > 1$) :

- ① 将A上的1到 ($n-1$) 号盘移到B, C为辅助塔; ①和③与原问题相同, 只是规模小。
- ② 将A上第 n 号盘移至C;
- ③ 将B上1到($n-1$)号盘移到C, A为辅助塔。

```
void hanoi ( int n, char A, char B , char C) //hanoi塔算法
{
    if (n == 1) move (A, 1, C ); //移A上的一个盘到C
    else {
        hanoi(n-1, A, C, B);    //解决子问题①
        move (A, n, C);        //移A上的n 号盘到C
        hanoi (n-1, B, A, C);   //解决子问题③
    }
}
```




递归的定义和递归函数

许多问题都可以很容易用递归函数来设计。例如：

求 $1 + 2 + 3 + \dots + N$ 。

求N个数的最大值。

将N个数倒序。

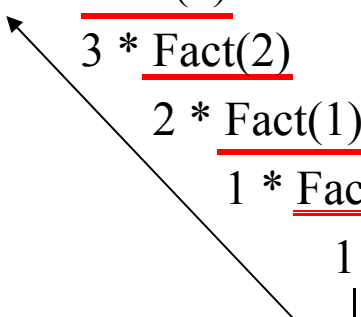


递归的定义和递归函数

2. 递归到非递归的转换

例 计算 $n!$ 。

以 $n = 4$ 为例，分析其计算过程。

$$\begin{array}{r} \text{Fact}(4) = 4 * \text{Fact}(3) \\ \quad \quad \quad \underline{3 * \text{Fact}(2)} \\ \quad \quad \quad \quad \underline{2 * \text{Fact}(1)} \\ \quad \quad \quad \quad \quad \underline{1 * \text{Fact}(0)} \\ \quad \quad \quad \quad \quad \quad 1 \end{array}$$


```
int Fact (int n) //求n!
{
    if (n == 0) return 1;
    else return n * Fact(n-1) ;
}
```

返回地址



两种求解思路：

(1) 从整个问题开始，当不能解决时先记下，不断分解，直到最小问题直接解决后，再逐层返回。

(2) 直接从求最小问题开始。



递归到非递归的转换

非递归：

(1) 不用栈，直接从最小问题（0!）开始求。

```
int Fact(int n)
{
    int f, i;
    f = 1; i = 0;
    while (i < n) {
        i++;
        f = f * i;
    }
    return f;
}
```



递归到非递归的转换

(2) 引入一个栈**S**，存放当前参数**n**。

思路：

当 $n > 0$ 时， n 进栈，然后 $n--$ ，继续，直到 $n = 0$ ，得 $f = 1$ ；

然后，依次出栈 $\rightarrow n$ ， $f = f * n$ ，直到栈空为止。

```
int Fact(int n)
{
    int f;
    ClearStack(S);
    while (n > 0) {
        Push(S, n); n--;
    }
    f = 1;
    while (!EmptyStack(S)) {
        f = f * Pop(S);
    }
    return f;
}
```



递归到非递归的转换

说明:

- ✓ 对于某些递归问题，非递归函数必须使用栈来实现；
- ✓ 理解函数调用与返回；
- ✓ 递归过程（函数）与非递归过程（函数）没有本质区别。

递归到非递归的转换

例 求Ackerman函数。

以Ack(2, 1)为例， $m=2, n=1$ 。

$Ack(2, 1) = 5$

```
int Ack (int m, int n) //求Ackermam函数
{
    if (m == 0) return n+1;
    else if (n == 0) return Ack (m-1,1) ;
    else return  ACK (m-1,Ack (m, n-1));
}
```

返回地址

Ack(2, 1)
Ack(1, Ack(2, 0))
Ack(1, 1)
Ack(0, Ack(1, 0))
Ack(0, 1)
2
3
Ack(1, 3)
Ack(0, Ack(1, 2))
Ack(0, Ack(1, 1))
Ack(0, Ack(1, 0))
Ack(0, 1)
2
3
4
5



递归到非递归的转换

非递归:

```
int Ack (int m, int n) //求Ackermam函数
{
    int f; ClearStack(S);
    while (1) {
        if (m == 0) {
            f = n + 1;
            if (EmptyStack(S)) break;
            m = Pop(S); n = f; }
        else if (n == 0) {
            m--; n = 1; }
        else {
            Push(S, m - 1); n--; }
    }
    return f;
}
```

3.4 队列

1. 队列的定义及其基本操作

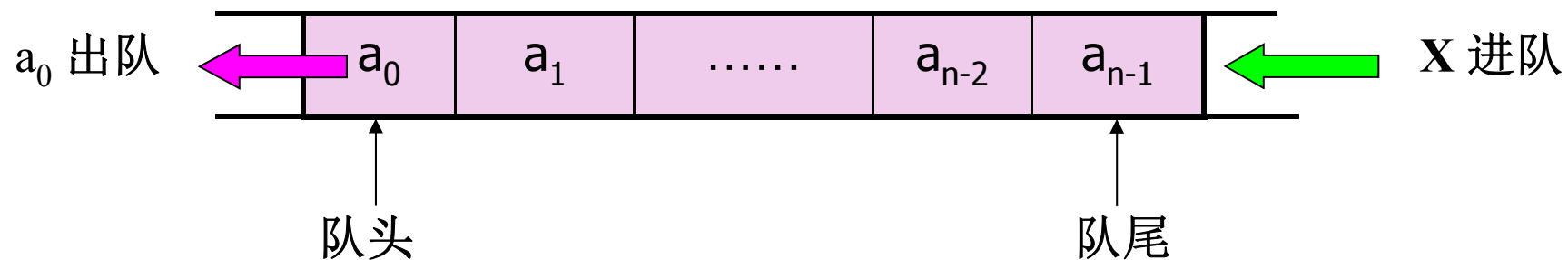
定义：队列（Queue）逻辑上也是一种线性表，记为

$$Q = (a_0, a_1, \dots, a_{n-1})$$

对队Q的操作（插入，删除）限定在表的两端进行。

队头：仅能删除（出队）

队尾：仅能插入（进队）





队列的定义及其基本操作

队列的特点：

✓ 先进先出（**First In First Out, FIFO**）

设元素进队顺序为 a_0, a_1, \dots, a_{n-1} ，则出队顺序也是 a_0, a_1, \dots, a_{n-1} ，即先进队的元素先出队，故队列可称为“**先进先出**”的线性表，

✓ 队头是唯一的出口，队尾是唯一的入口。

当队列中没有元素时，称“**队空**”。若队列存储空间已满，再作进队操作时，称“**队满溢出**”。



队列的定义及其基本操作

队列的抽象数据类型:

ADT Queue {

数据元素集: $D = \{a_i \mid a_i \in \text{datatype}, i=0,1,2, \dots, n-1, n \geq 0\}$

数据关系集: $R = \{ \langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D, 0 \leq i \leq n-2 \}$

约定 a_0 为队头元素, a_{n-1} 为队尾元素

基本操作集: P

QueueInit(&Q)

操作结果: 创建一个空队列Q。

ClearQueue(&Q)

初始条件: 队列Q已经存在。

操作结果: 清空队列。

QueueLength(Q)

初始条件: 队列Q已经存在。

操作结果: 返回队列Q的元素个数。



队列的定义及其基本操作

EmptyQueue(Q)

初始条件：队列Q已经存在。

操作结果：若Q为空队列，则返回TRUE，否则返回FLASE。

QueueFull(Q)

初始条件：队列Q已经存在。

操作结果：若Q为已满，则返回TRUE，否则返回FLASE。

EnQueue(&Q, e)

初始条件：队列Q已经存在且未滿。

操作结果：插入数据元素e，使之成为新队尾元素。

DeQueue(&Q)

初始条件：队列Q已经存在且非空。

操作结果：删除Q的队头元素，并返回其值。

GetHead(Q)

初始条件：队列Q已经存在且非空。

操作结果：返回队头元素的值。

.

}ADT Queue;



3.4 队列

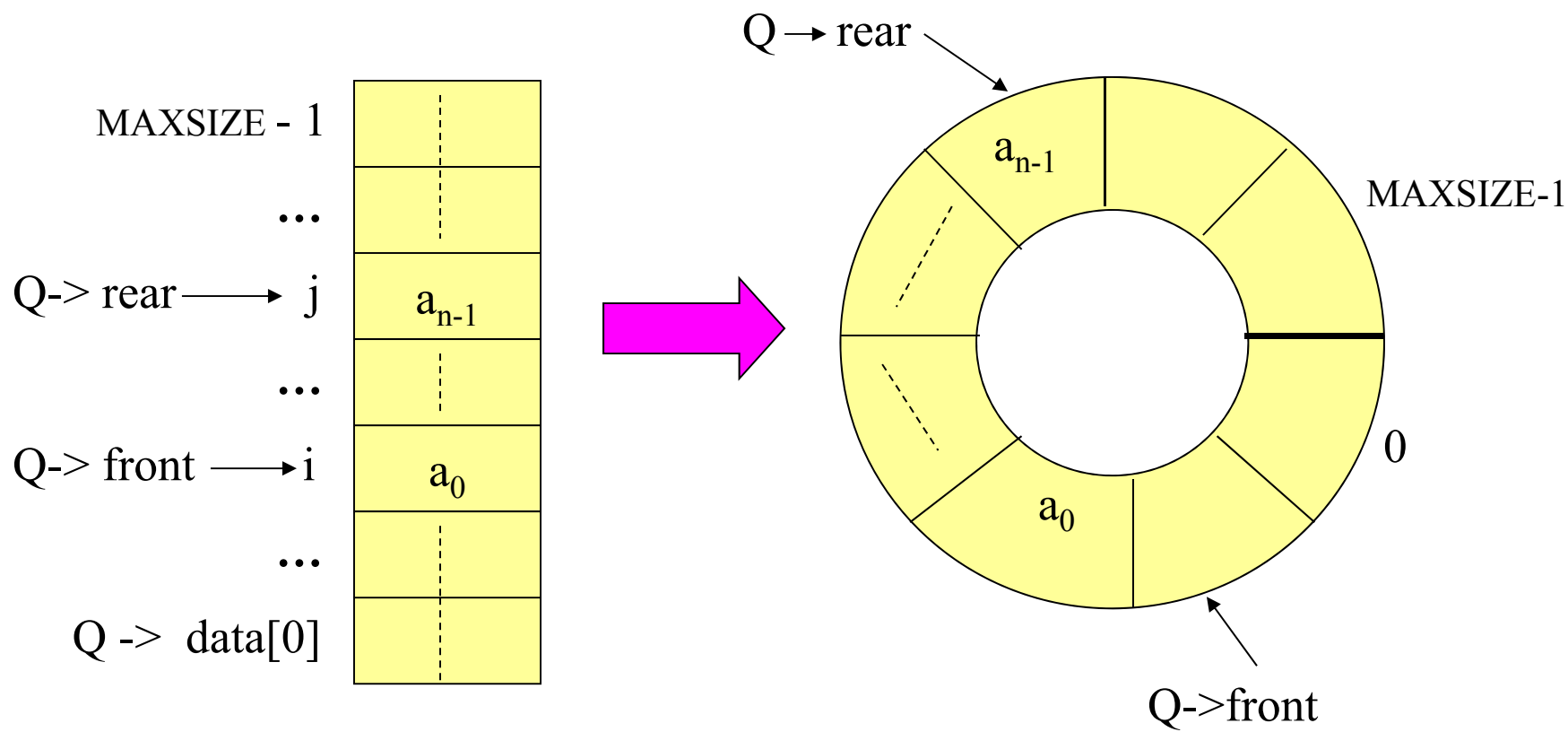
2. 队列的顺序存储结构

(1) 顺序队列的描述

```
#define MAXSIZE 64 //队列最大容量  
typedef struct {  
    datatype data[MAXSIZE]; //队列的存储空间  
    int front, rear;          //队头、队尾指针  
} queue, *sqlink;
```

队列的顺序存储结构

队头、队尾的位置以及如何变化？



循环队列

队列的顺序存储结构

设 $Q \rightarrow \text{front}$ 指向队头的前驱

$Q \rightarrow \text{rear}$ 指向队尾

初始化: $Q \rightarrow \text{front} = Q \rightarrow \text{rear} = 0$

进队:

$Q \rightarrow \text{rear} = (Q \rightarrow \text{rear} + 1) \% \text{MAXSIZE};$

$Q \rightarrow \text{data}[Q \rightarrow \text{rear}] = e;$

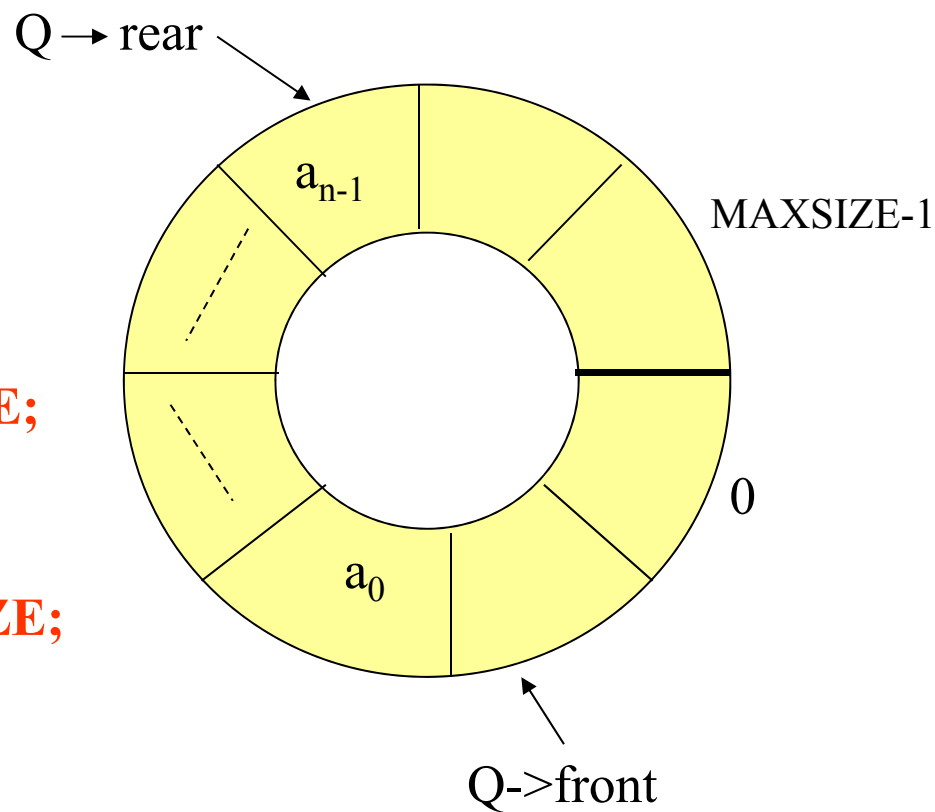
出队:

$Q \rightarrow \text{front} = (Q \rightarrow \text{front} + 1) \% \text{MAXSIZE};$

$e = Q \rightarrow \text{data}[Q \rightarrow \text{front}];$

如何判断队空/队满?

$Q \rightarrow \text{front} == Q \rightarrow \text{rear} ?$ 无法区分队空/队满!



队列的顺序存储结构

解决办法:

1) 设置1个布尔变量表示队空/队满。

当出队使得队头追上队尾, 则队空

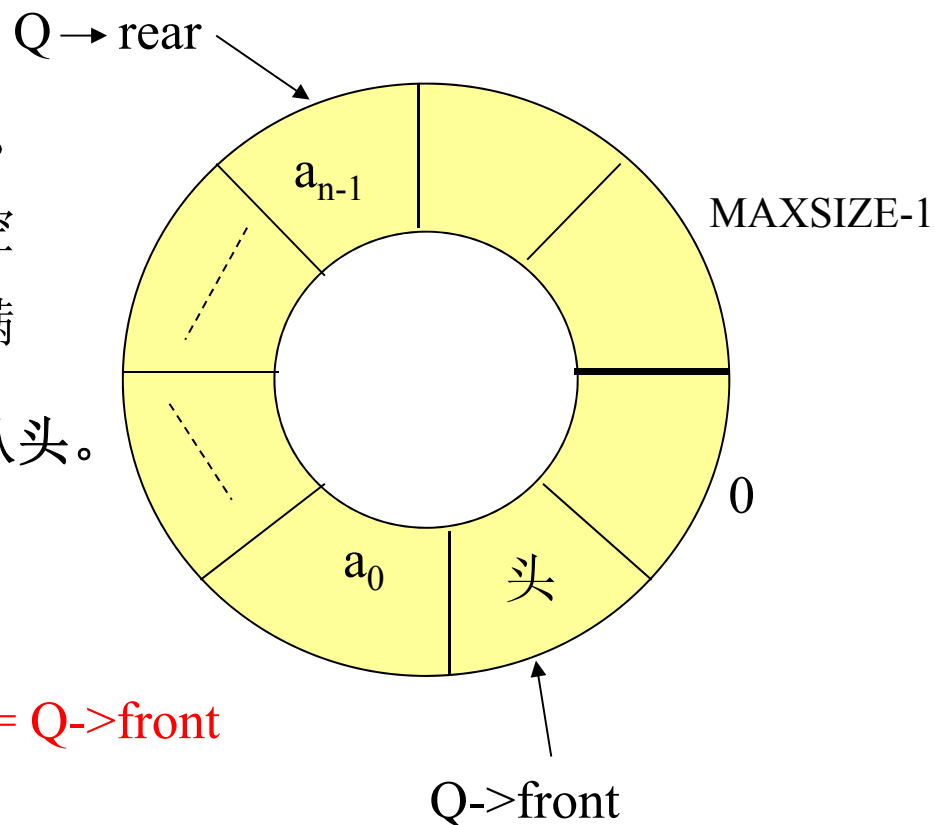
当进队使得队尾追上队头, 则队满

2) 牺牲1个单元, 使队尾不会追上队头。

即n个单元最多放n-1个元素

队空: $Q \rightarrow \text{front} == Q \rightarrow \text{rear}$

队满: $(Q \rightarrow \text{rear} + 1) \% \text{MAXSIZE} == Q \rightarrow \text{front}$





队列的顺序存储结构

(2) 循环队列基本操作的实现

```
1) void ClearQueue(squlink Q) //置队空
{
    Q->front = Q->rear = 0;
}
```

```
2) int EmptyQueue(squlink Q) //判断队空
{
    return (Q->front == Q->rear);
}
```

```
3) int QueueLength(squlink Q) //求队Q中当前元素个数
{
    int i = (Q->rear - Q->front + MAXSIZE) % MAXSIZE;
    return i;
}
```

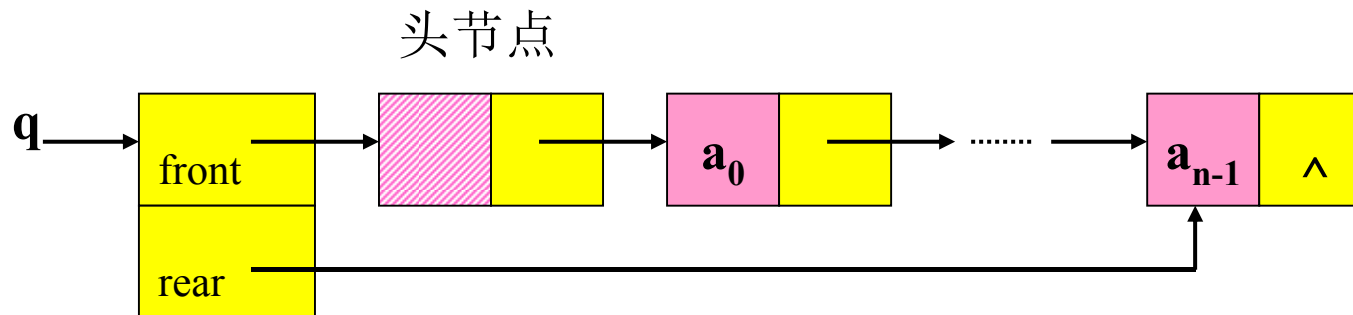



```
{
    if ((Q->rear + 1) % MAXSIZE == Q->front)
        return -1; //队满
    Q->rear = (Q->rear + 1) % MAXSIZE;
    Q->data[Q->rear] = e;
    return 0; //成功
}
```

```
{
    if ( EmptyQueue(Q)) return -1;    //队空
    Q->front = (Q->front + 1 ) % MAXSIZE;
    *pe = Q->data[Q->front]);
    return 0;
}
```

3.4 队列

2. 队列的链式存储结构



(1) 链式队列的描述

```
typedef struct node { //节点类型
    datatype data;
    struct node *next;
}Qnode, *Qlink
```

```
typedef struct { //q节点类型
    Qnode *front, *rear;
} linkqueue;
```



队列的链式存储结构

(2) 链式队列基本操作的实现

1) void Lcreatqueue (linkqueue *q) //创建空队列

{

 q->front = q->rear = (Qlink) malloc(sizeof(Qnode)); //申请头节点

 q->front->next = NULL;

}

2) int Lemptyqueue (linkqueue *q) //判断队空

{

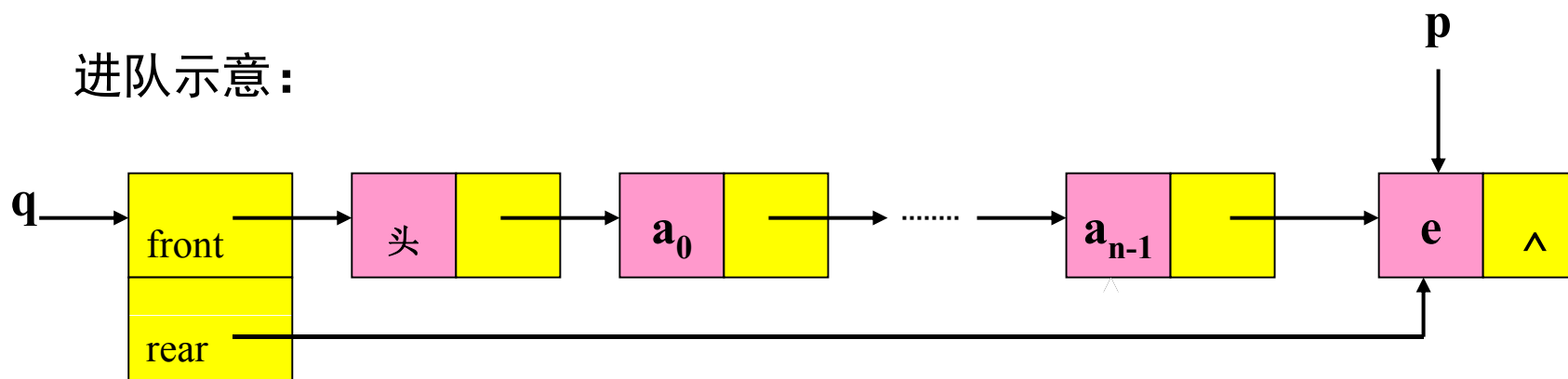
 return (q->front == q->rear);

}

队列的链式存储结构

```
3) void Lenqueue ( linkqueue *q, datatype e ) //元素e进队
{
    Qlink p = (Qlink) malloc ( (sizeof (Qnode) )); //申请进队节点
    p->data = e; p->next = NULL;
    q->rear->next = p;
    q->rear = p;
}
```

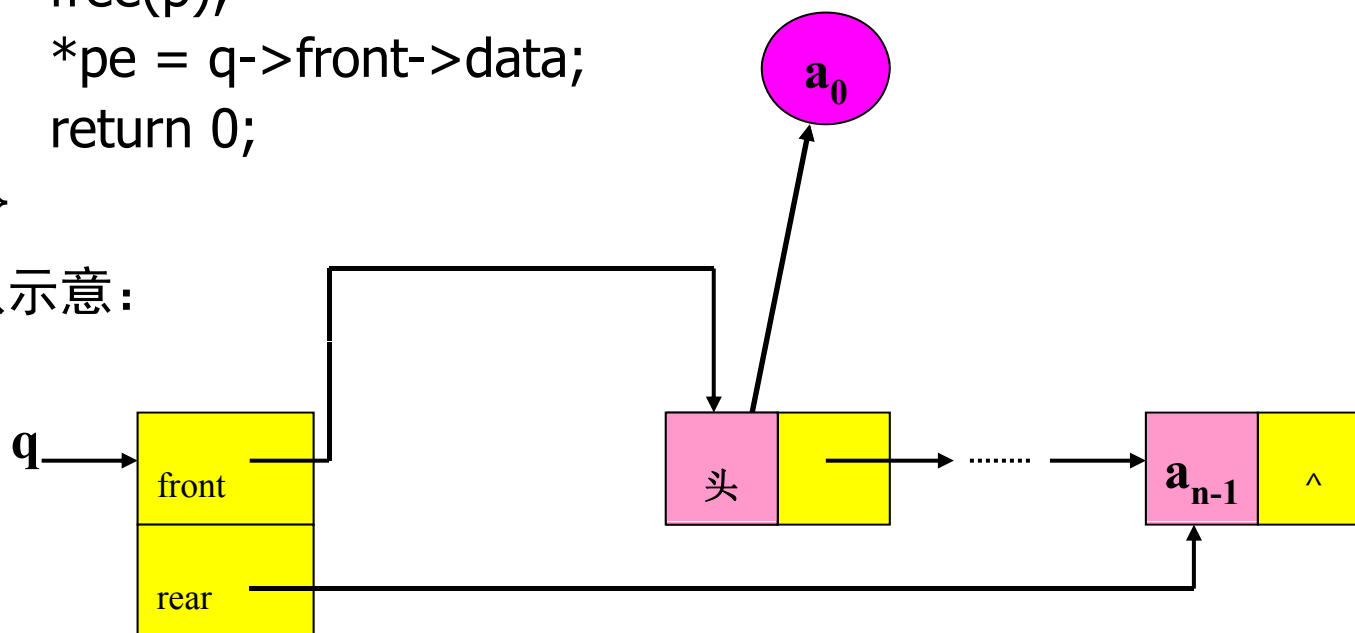
进队示意：



队列的链式存储结构

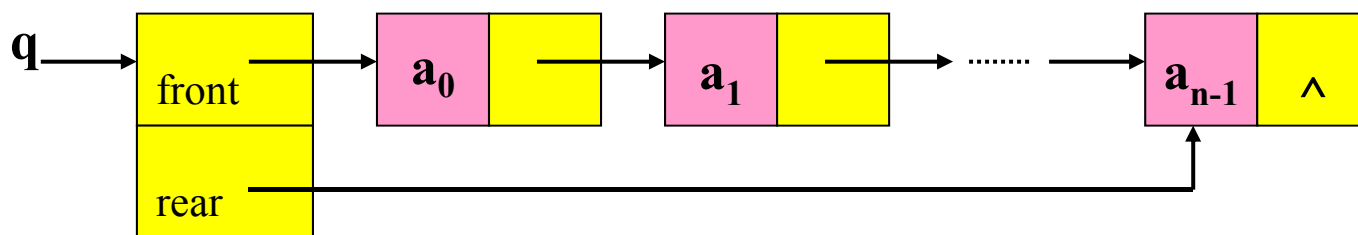
```
4) int Ldequeue(linkqueue *q, datatype *pe) //出队
{
    Qlink p;
    if ( Lemptyqueue (q)) return -1; //队空处理
    p = q->front ; q->front = p->next ;
    free(p);
    *pe = q->front->data;
    return 0;
}
```

出队示意:



队列的链式存储结构

如果链式队列采用不带头节点的单链表，如何实现？



1) void Lcreatqueue (linkqueue *q) //创建空队列

{

 q->front = q->rear = NULL;

}

2) int Lemptyqueue (linkqueue *q) //判断队空

{

 return (q->front == NULL);

}



队列的链式存储结构

```
3) int Lenqueue( linkqueue *q, datatype e ) //元素e进队
{
    Qlink p = (Qlink) malloc( (sizeof (Qnode) ) ); //申请进队节点
    if (p == NULL ) return -1; //为简化处理，许多算法略去了对p的检查
    p->data = e; p->next = NULL;
    if (q->rear) != NULL ) {
        q->rear->next = p; q->rear = p;
    } else {
        q->front = q->rear = p;
    }
    return 0;
}
```



队列的链式存储结构

```
4) int Ldequeue(linkqueue *q, datatype *pe) //出队
{
    Qlink p;
    if ( Lemptyqueue (q)) return -1; //队空处理
    *pe = q->front->data;
    p = q->front ;
    q->front = q->front->next ;
    free(p);
    if (q->front == NULL ) q->rear = NULL;
    return 0;
}
```

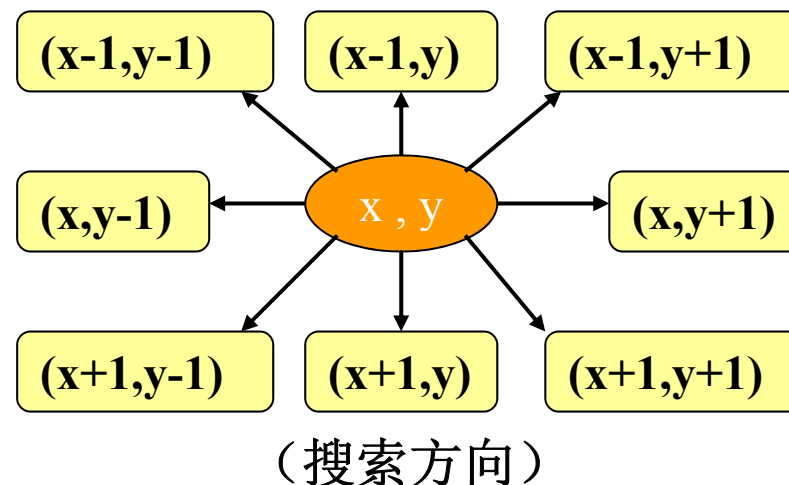

3.5 队列应用举例

1. 迷宫问题：设二维数组 $\text{maze}[m+2][n+2]$ 表示一迷宫，如 $m=5$ 、 $n=6$ 时的迷宫如下图所示。求从迷宫入口到出口的一条最短路径。

	0	1	2	3	4	5	6	7 (列下标)
0	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1
2	1	0	1	0	1	1	1	1
3	1	0	1	0	0	0	1	1
4	1	1	1	1	0	1	0	1
5	1	1	1	1	1	1	0	1
6	1	1	1	1	1	1	1	1

(行下标)

每一步怎么走？



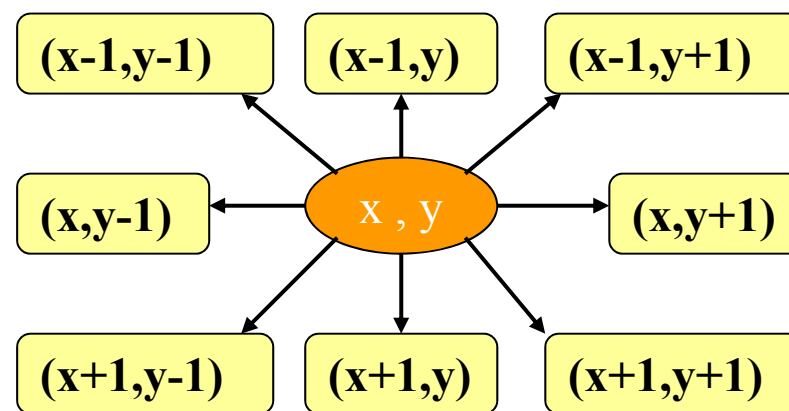
约定： $\text{maze}[1][1]=0$ 为迷宫入口， $\text{maze}[m][n]=0$ 为迷宫出口； $\text{maze}[x][y]$ 取值为0时表示路通，取1时表示路不通。迷宫外所围的一层为处理问题方便所加，其相应点全为1。

迷宫问题

0 1 2 3 4 5 6 7 (列下标)

0	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1
2	1	0	1	0	1	1	1	1
3	1	0	1	0	0	0	1	1
4	1	1	1	1	0	1	0	1
5	1	1	1	1	1	1	0	1
6	1	1	1	1	1	1	1	1

(行下标)

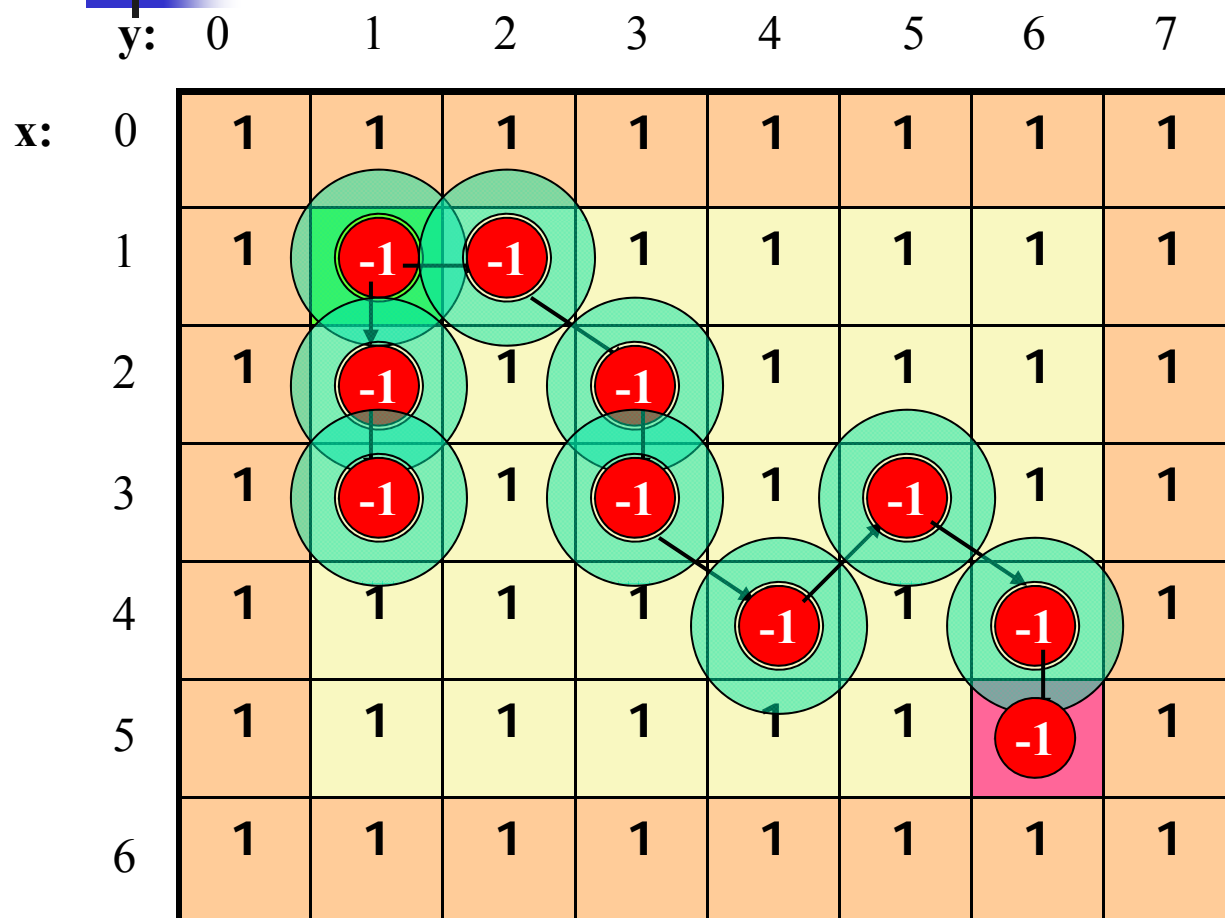


(搜索方向)

采用什么数据结构？如何处理？

- 1) 8个搜索方向的表示： 坐标增量表
- 2) 搜索过程如何记录？ 队列，避免重复搜索（标记），
记住来时路（队列元素定义）

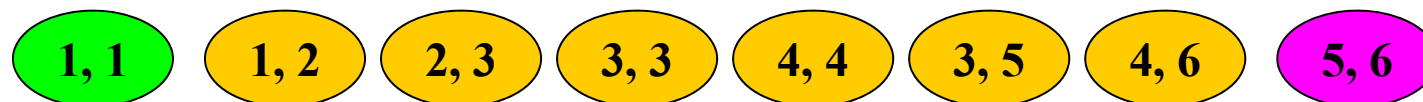
迷宫问题



队列，p表示前驱

序号	x	y	p
1	1	1	0
2	1	2	1
3	2	1	1
4	2	3	2
5	3	1	3
6	3	3	4
7	4	4	6
8	3	5	7
9	4	6	8
10	5	6	9

最短路径:





迷宫问题

算法描述:

建立空队列Q; 将迷宫入口点进队;

置入口点在迷宫的值为-1; //表示已处理

CurrentPoint = Q队头元素; //入口点

while (CurrentPoint存在) {

 for (k = 0; k < 8; k++) { //依次处理8个搜索方向

 NextPoint = 相对CurrentPoint第k个方向的点;

 if (NextPoint可达) {

 将NextPoint进队; 置NextPoint在迷宫的值为-1;

 在队列Q中记录NextPoint是由CurrentPoint到达的;

 if (NextPoint是出口点) {

 根据队列Q中记录的信息输出路径; return 0; //存在路径

 }

 }

 }

 CurrentPoint = Q中CurrentPoint的后继;

}

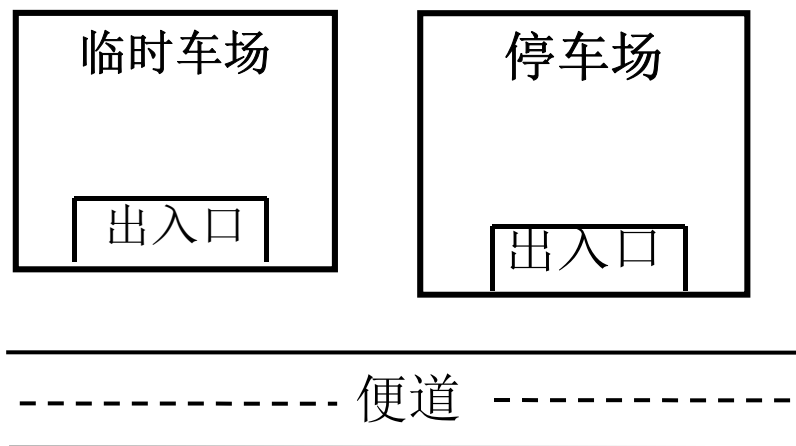
return -1; //路径不存在

3.5 队列应用举例

2. 离散事件模拟

停车场进出车事件模拟：

停车场可停放若干辆汽车，且只有一个出入口。汽车在停车场内按到达的先后顺序，依次由里向外排列。若停车场已满，后来的汽车只能在车场外的便道上排队等候。一旦停车场内有汽车开走，则便道上的第一辆车可开入。当停车场内某车要开走时，在它之后的车必须先退到临时车场为其让路，待汽车开走后，为它让路的车再按原次序进入停车场。



相关的数据结构：

停车场：栈

便道：队列

临时车场：栈



离散事件模拟

相关的事件及处理:

1) 车来:

if (停车场栈未满) 车进入停车场栈;

else 车进入便道队列;

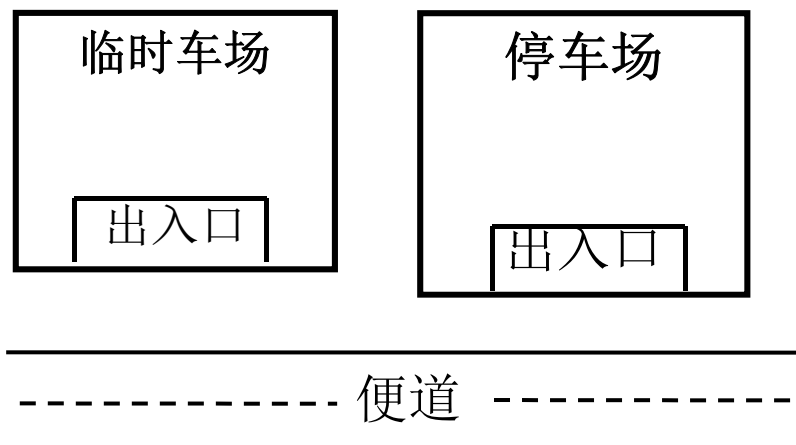
2) 车走:

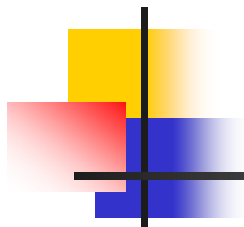
将停车场中该车后面的所有车依次出栈到临时车场栈;

该车出栈 (开走);

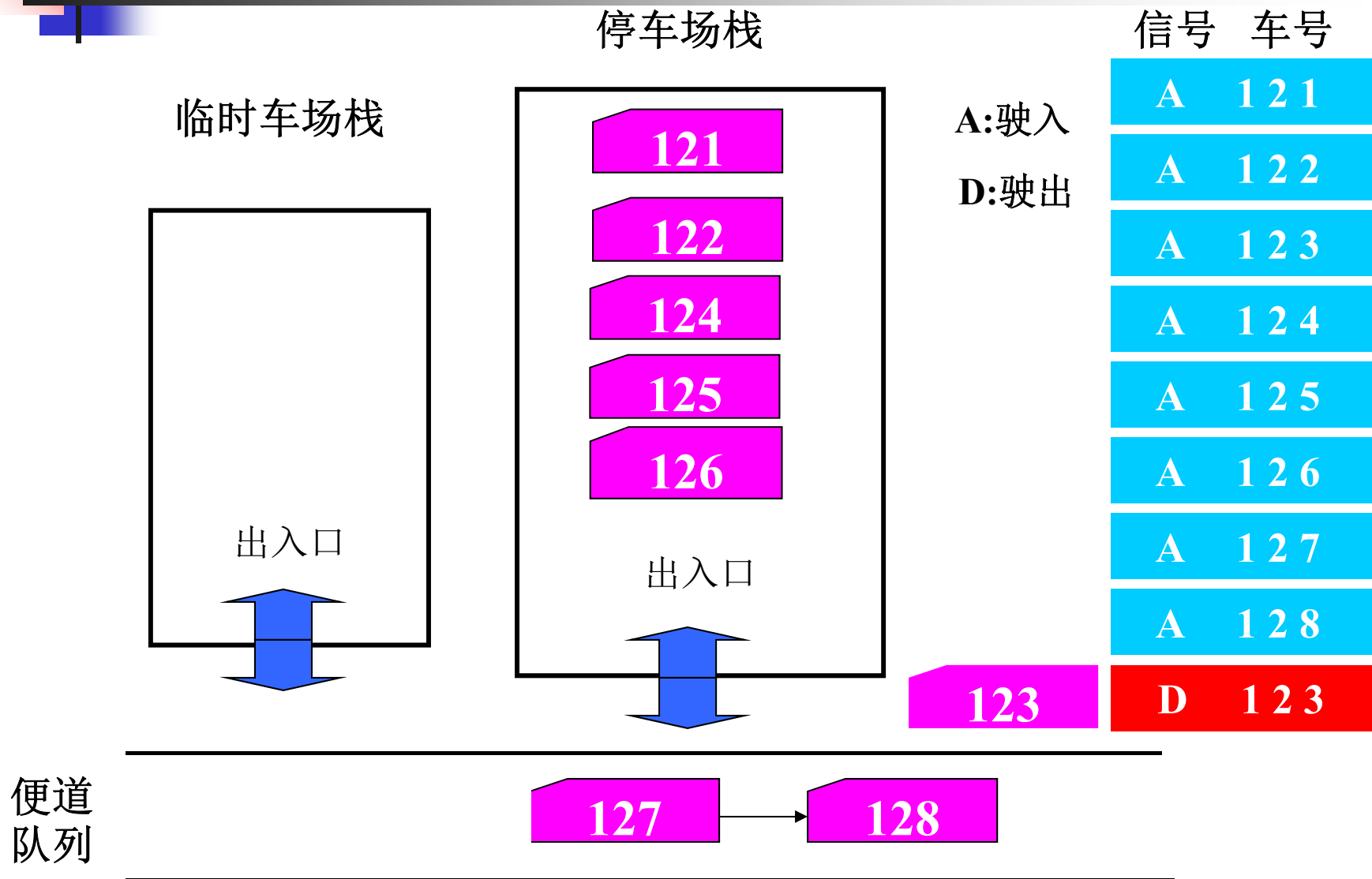
将临时车场栈的所有车依次出栈进入停车场栈;

if (便道队列非空) 队头车进入停车场栈;

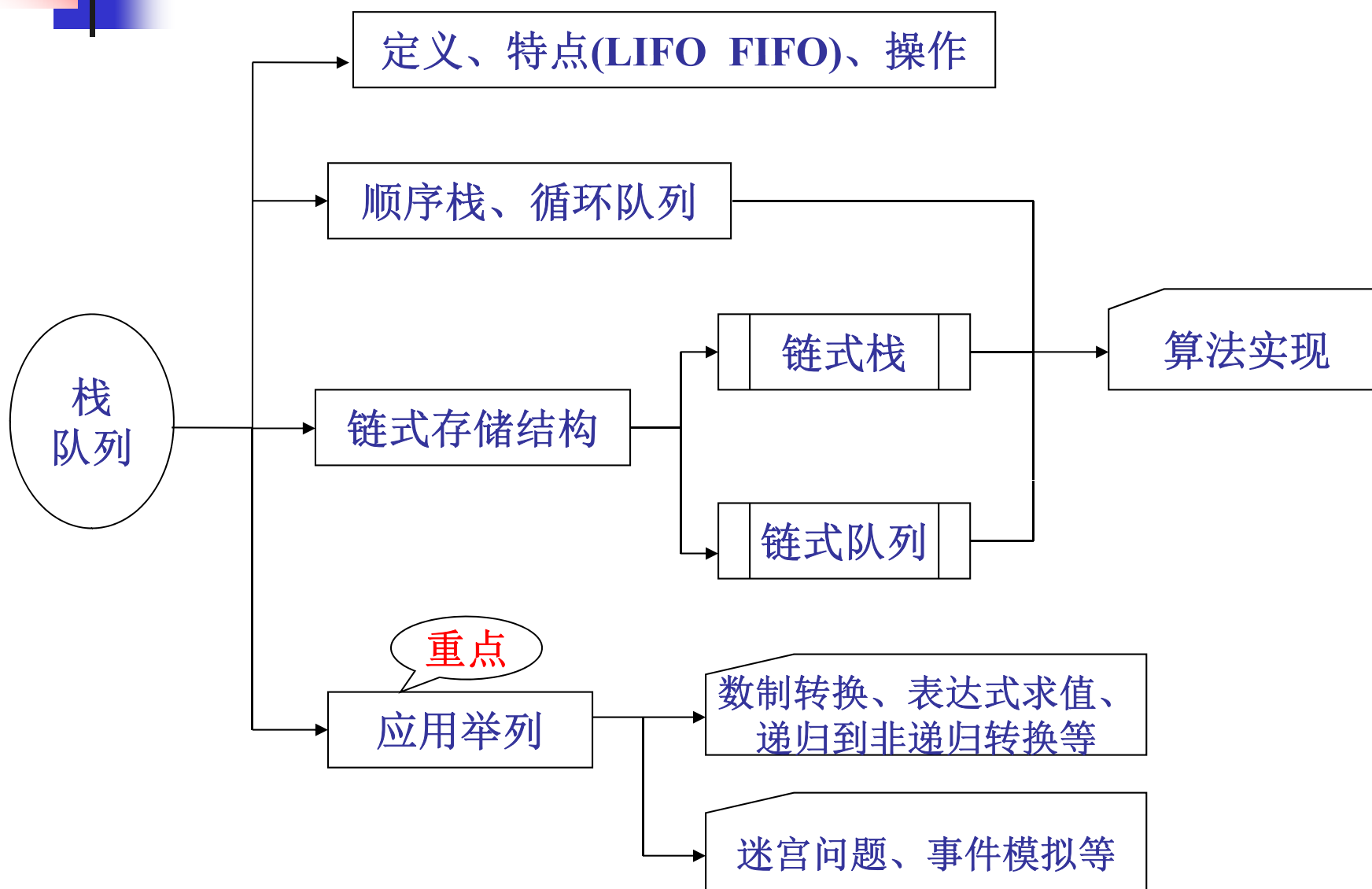




离散事件模拟



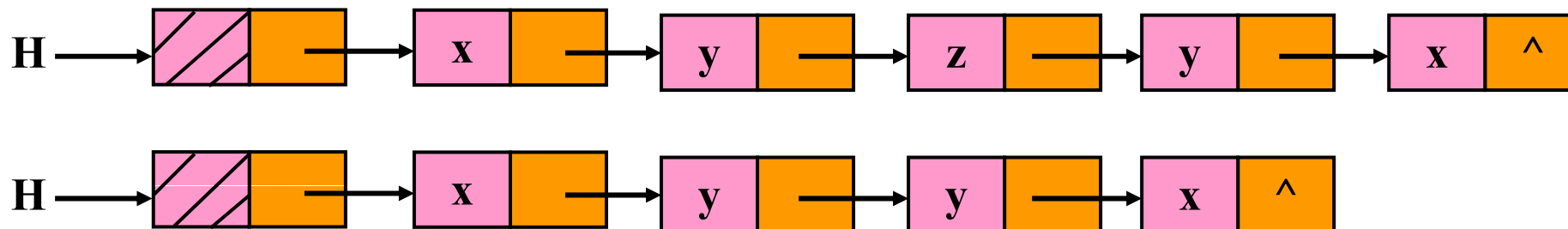
3.6 小结



第3章 作业

1. 设元素为1, 2, 3, 4, 5。进栈顺序约定：值小的元素先进栈，但在两次进栈之间，可作出栈运算。写出5个可以得到的出栈序列；5个不可以得到的出栈序列。

2. 设单链表：



为对称形式（表长= n ）。使用栈操作，写出判断表H是否对称的算法：
 $xyx(H)$ 。