

第三章栈和队列

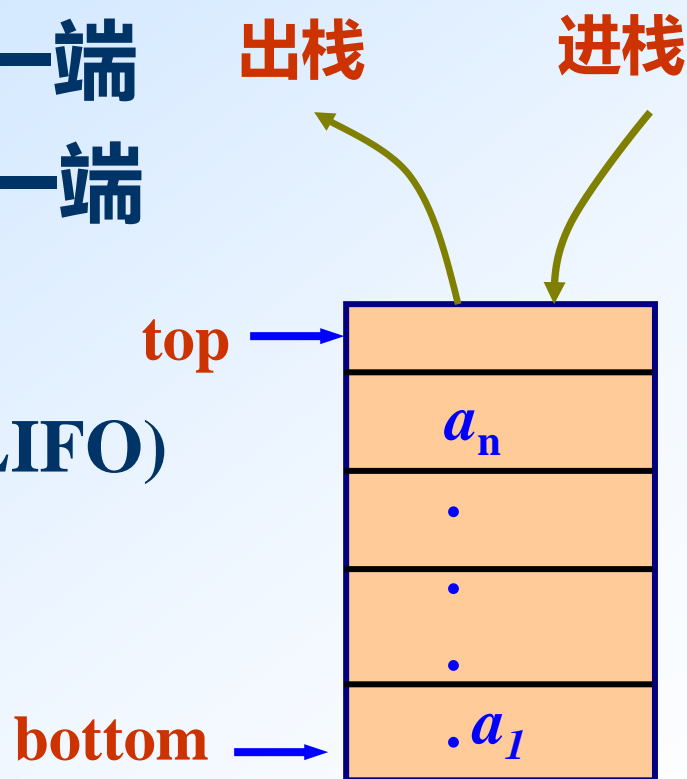
❖ 栈

❖ 递归

❖ 队列

栈 (Stack)

- **定义:**是限定仅在表尾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(top), 另一端称为栈底(bottom)
- **特点:** 后进先出 (LIFO)



栈的主要操作

ADT Stack {

//对象由数据类型为StackData的元素构成

void InitStack (stack *S); **//置空栈**

int StackEmpty (stack *S); **//判栈空否**

int StackFull (stack *S); **//判栈满否**

int Push (stack *S, StackData x); **//进栈**

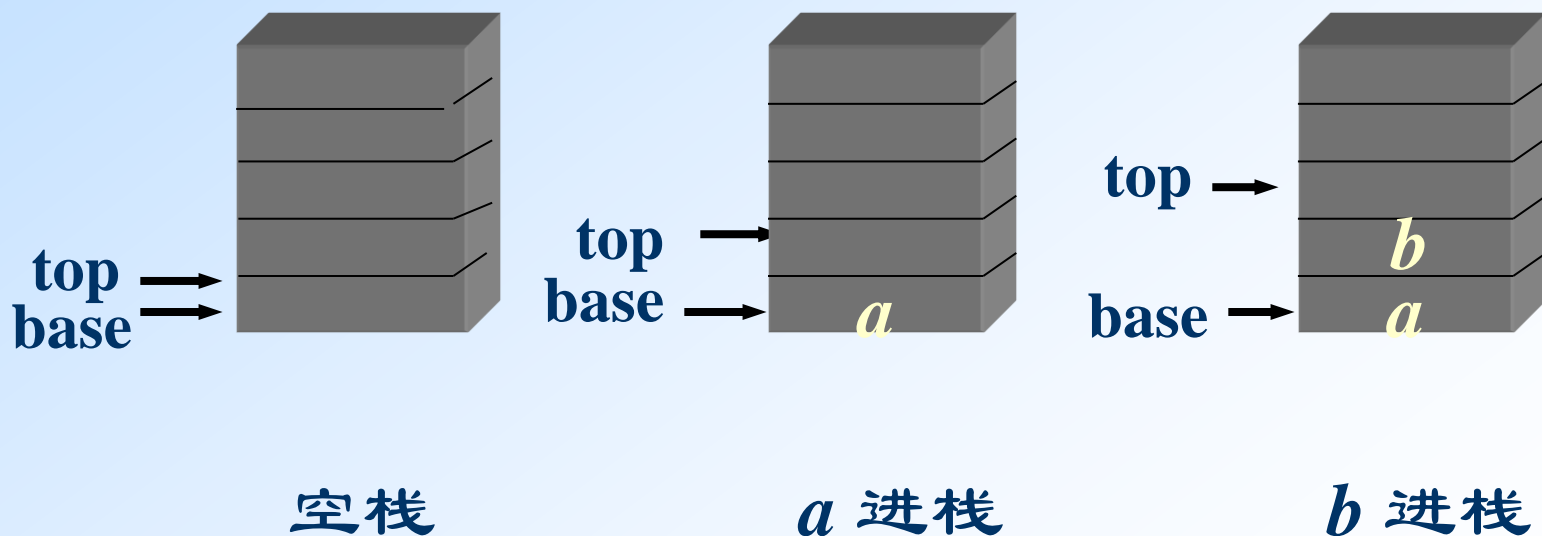
int Pop (stack *S, StackData *x); **//出栈**

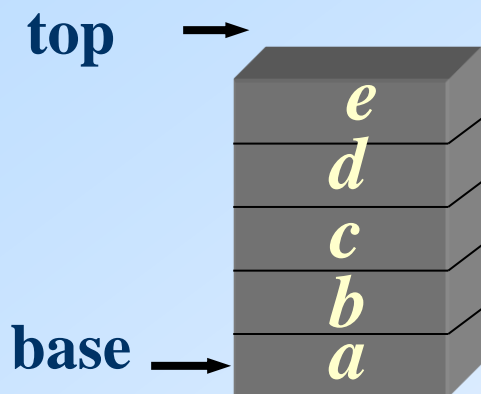
int GetTop (stack *S, StackData *x); **//取栈顶**

}

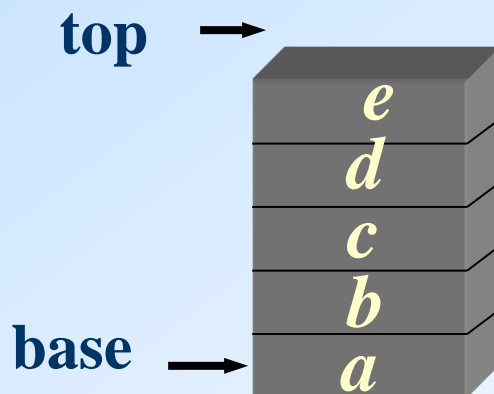
栈的表示和实现

- **顺序栈**：栈的顺序存储结构，利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，指针 top 指向栈顶元素在顺序栈中的下一个位置， $base$ 为栈底指针，指向栈底的位置。

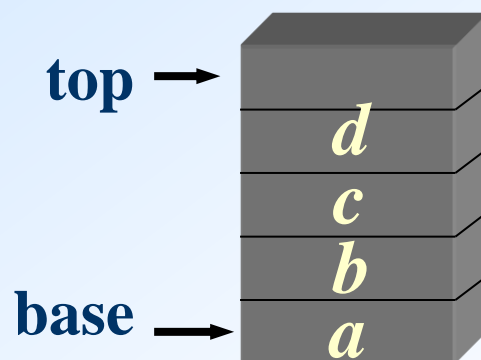




e 进栈



f 进栈溢出



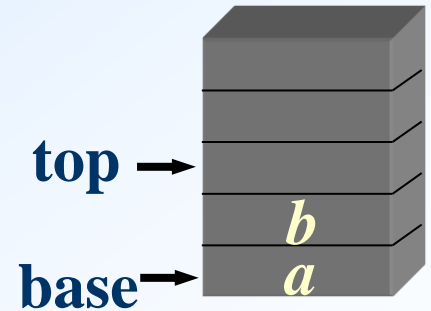
e 出栈

顺序栈的类型表示:

```
#define STACK_INIT_SIZE 100  
#define STACKINCREMENT 10
```

```
typedef char StackData;
```

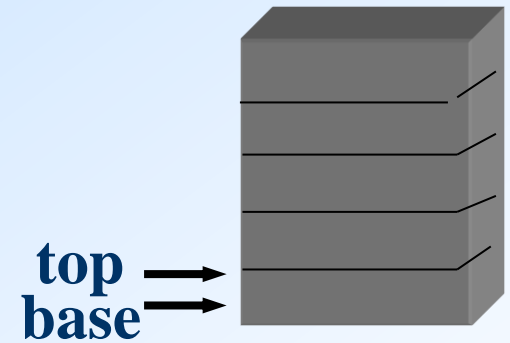
```
typedef struct {           //顺序栈定义  
    StackData *base; //栈底指针  
    StackData *top; //栈顶指针  
    int stacksize; //当前已分配的存储空间  
} SeqStack;
```



顺序栈的基本运算:

■ 判栈空

```
int StackEmpty (SeqStack *S) {  
    if( S->top == S->base ) return 1;  
    //判栈空,空则返回1  
    else return 0; //否则返回0  
}
```



■ 判栈满

```
int StackFull (SeqStack *S) {  
    if( S->top - S->base >= S->StackSize ) return 1;  
    //判栈满,满则返回1  
    else return 0; //否则返回0  
}
```

■初始化

```
void InitStack ( SeqStack *S) { //置空栈
```

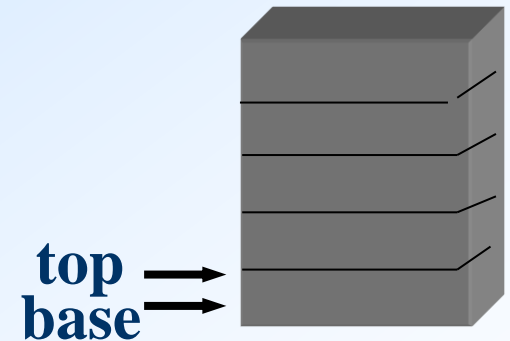
```
    S->base =( StackData *)malloc(STACK_INIT_SIZE *  
                                   sizeof(StackData));
```

```
    if (!S->base) exit;
```

```
    S->top = S->base ;
```

```
    S->stacksize= STACK_INIT_SIZE ;
```

```
}
```



空栈

```
main()
```

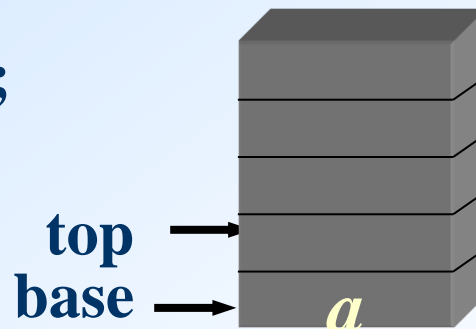
```
{ SeqStack *S;
```

```
  S=(SeqStack *)malloc(sizeof(SeqStack));
```

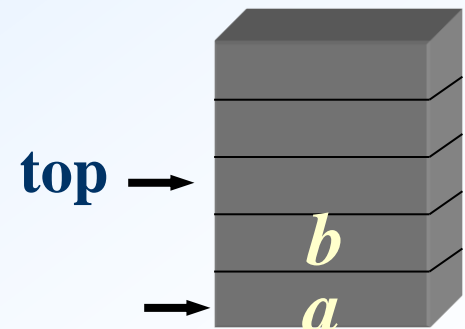
```
  InitStack (S);
```


■ 入栈

```
void Push (SeqStack *S, StackData x) {  
    //插入元素x为新的栈顶元素  
    if ( StackFull (S) ){  
        S->base =( StackData *)realloc(S->base ,  
                                         (S->stacksize+ STACKINCREMENT) *  
                                         sizeof(StackData)); //s->base可能改动  
        if(! S->base)exit(overflow);  
        S->top= S->base + S->stacksize;  
        S->stacksize+= STACKINCREMENT; //追加存储空间  
    }  
    *(S->top)=x;  
    (S->top) ++;  
}
```



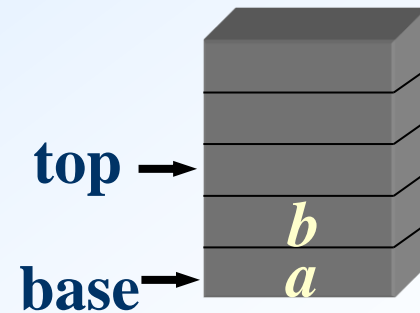
a 进栈



b 进栈

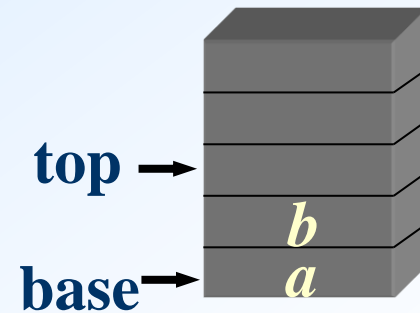
■ 取栈顶元素

```
int Gettop (SeqStack *S, StackData *x) {  
    //若栈空返回0, 否则栈顶元素读到x并返回1  
    if ( StackEmpty(S) ) return 0;  
    *x = *(S->top-1);  
    return 1;  
}
```



■ 出栈

```
Int Pop (SeqStack *S, StackData *x) {  
    //若栈空返回0, 否则栈顶元素退出到x并返回1  
    if ( StackEmpty(S) ) return 0;  
    (S->top) --;  
    *x = *(S->top);  
    return 1;  
}
```



1: 有一空栈, 现有ABCDE五个数据依次以PUSH指令放入栈中, 其中陆续执行了一些POP指令, 下列哪个选项为不可能的输出结果?

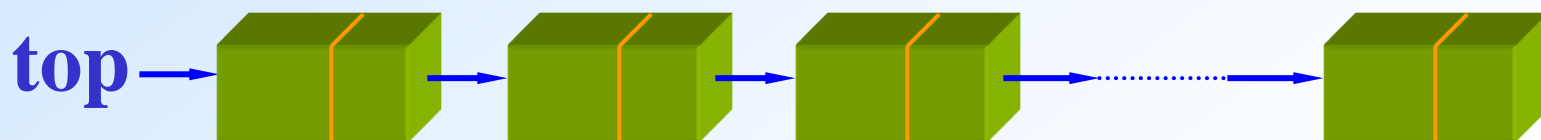
- (A) ABCDE (B) EDCBA
(C) EABCD (D) ABDEC

- 元素a,b,c,d,e依次进入初始为空的栈中。若元素进栈后可停留、可出栈，直到所有的元素都出栈，则所有可能的出栈序列中，以元素d开头的序列个数有几个？

- decba dceba dcbea dcbae

链式栈:栈的链接表示

- 链式栈无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链式栈的栈顶在链头
- 适合于多栈操作



链式栈 (LinkStack)的定义

```
typedef int StackData;  
typedef struct node {  
    StackData data;           //结点  
    struct node *link;       //链指针  
} StackNode;  
typedef struct {  
    StackNode *top;          //栈顶指针  
} LinkStack;
```

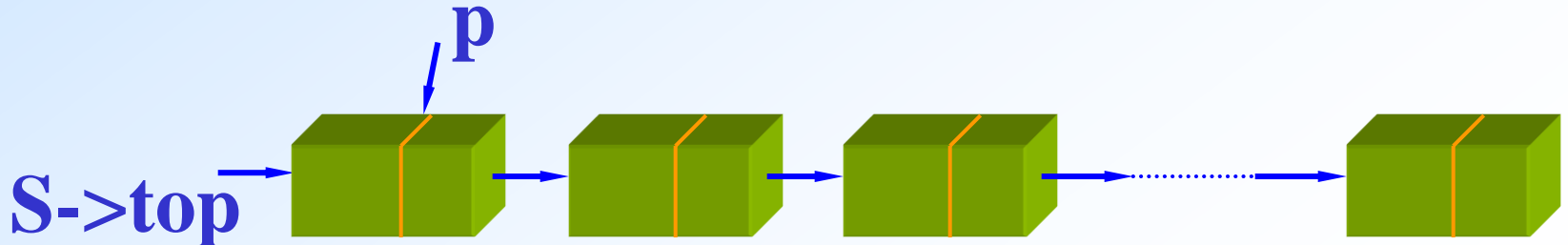
链式栈操作实现

■ 初始化

```
void InitStack ( LinkStack *S ) {  
    (*S).top = NULL; // 原代码S->top = NULL;  
}
```

■ 入栈

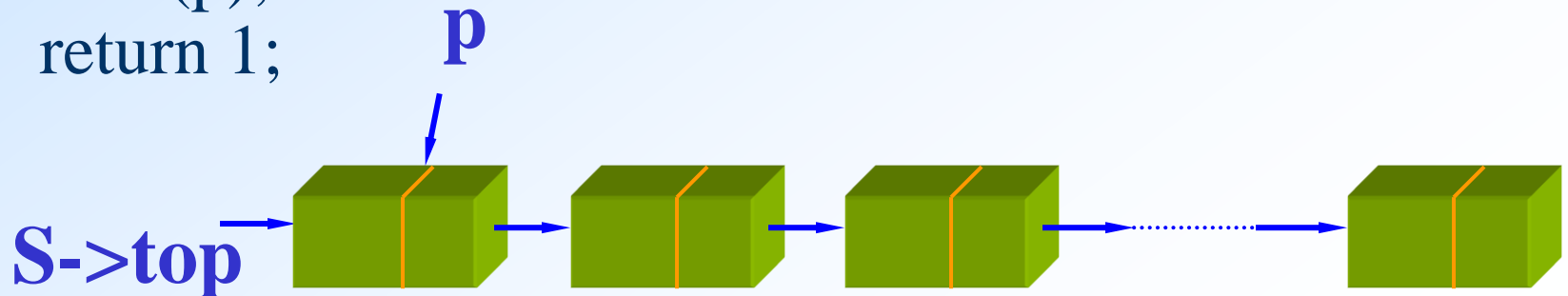
```
int Push ( LinkStack *S, StackData x ) {  
    StackNode *p;  
    p = ( StackNode * ) malloc ( sizeof ( StackNode ) );  
    p->data = x; p->link = (*S).top ;  
    (*S).top = p; return 1;  
}
```



- `int StackEmpty (LinkStack S) { //判栈空`
- `if(S.top == NULL) return 1;`
- `else return 0;`
- `}`

- **//出栈**

```
int Pop ( LinkStack *S, StackData *x ) {
    StackNode * p
    if ( StackEmpty ((*S) ) return 0;
    p= (*S).top ;
    (*S).top = p->link;
    * x = p->data;
    free (p);
    return 1;
}
```



■ 取栈顶

```
int GetTop ( LinkStack *S, StackData *x ) {  
    if ( StackEmpty (S) ) return 0;  
    * x = (*S).top->data;  
    return 1;  
}
```

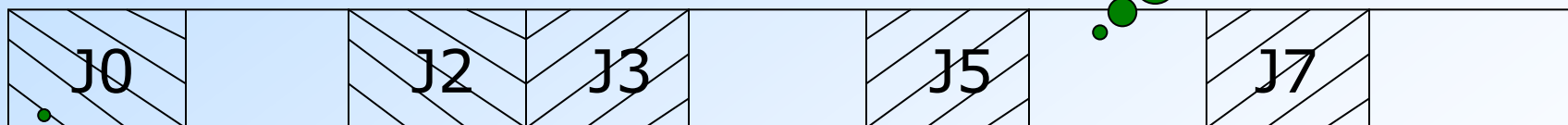
- **存储管理是操作系统的重要组成部分，它负责管理计算机系统的存储器。**
- **动态存储管理**的基本问题是系统如何应用户提出的“请求”分配内存？又如何收回那些用户不再使用而释放的内存以备新的“请求”产生时重新进行分配。动态存储管理中的一些常用技术，包括可利用空间表及分配方法、边界标识法、无用单元的收集和压缩存储等内容。

动态存储管理的基本问题是系统如何应用户提出的“请求”分配内存？又如何收回那些用户不再使用而释放的内存以备新的“新求”产生时重新进行分配？

在单用户操作系统中，整个内存空间被划分成两个区域：系统区和用户区，系统区供系统程序使用，用户区供单一的用户程序所使用。当计算机采用了多道程序设计技术后，需要在主存储器中同时存放多个作业的程序，而这些程序在主存储器中的位置此时不能由程序员自己来确定，否则将出现多道程序竞争同一存储空间的情况。



(A) 系统运行初期



可利用空间块
或空闲块

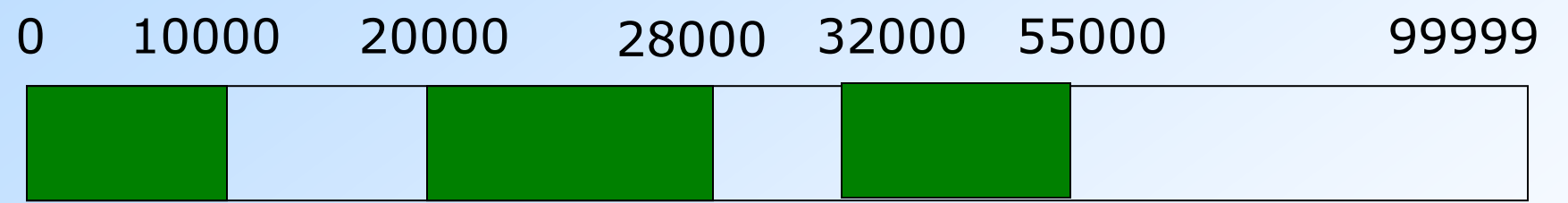
占用块

(B) 系统运行若干时间以后

通常有两种做法：一种策略是系统继续从高地址的空闲块中进行分配，而不理会已分配给用户的内存是否已空闲，直到分配无法进行

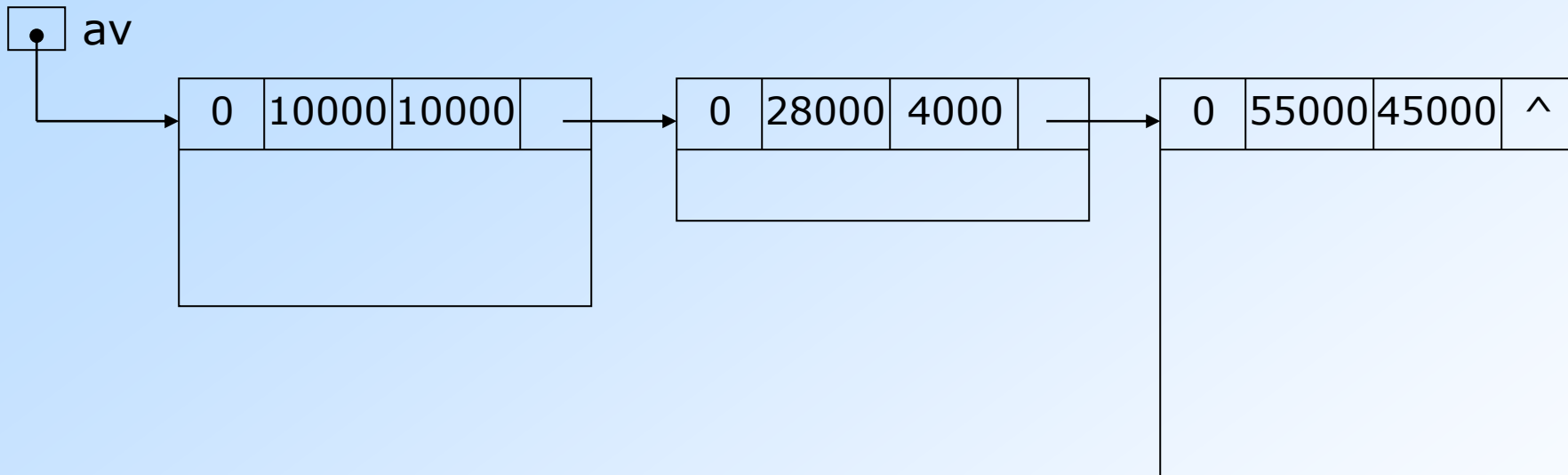
另一种策略是用户程序一旦运行结束，便将它所占内存区释放成为空闲块，同时，每当新的用户请求分配内存时，系统需要巡视整个内存区中所有空闲块，并从中找出一个“合适”的空闲块分配之。

为了实现这种分配策略，系统需建立一张记录所有空闲块的**可利用空间表**。此表的结构可以是目录表也可以是链表。如图所示为某系统运行过程中的内存状态及其两种结构的可利用空间表。



起始地址 内存块大小 使用情况

10000	10000	空闲
28000	4000	空闲
55000	45000	空闲



(c) 链表

可利用空间表及分配方法

操作系统既可借助目录表结构也可借助链表结构实现动态存储分配，本节将对采用链表的情况进行讨论。

根据系统运行的不同情况，可利用空间表可以有三种不同的结构形式：

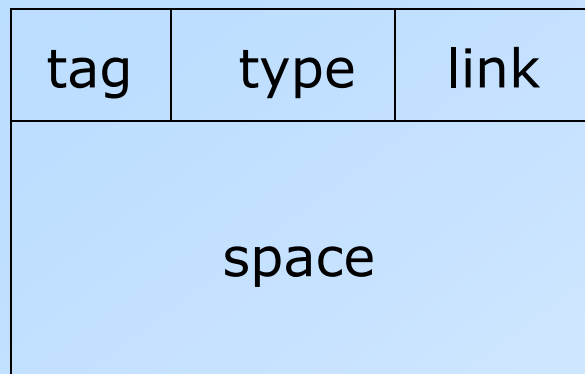
第一种情况是系统运行期间所有用户请求分配的**存储量大小相同**。对此类系统，可以在系统开始运行时将内存的用户区域按所需大小分割成若干大小相同的块，然后用指针链接成一个可利用空间表。

由于表中结点大小相同，所以在分配时无需查找，只要将第一个结点分配给用户即可；同样，当用户程序释放内存时，系统只需将用户释放的空闲块插入在表头即可。这种情况下的可利用空间表实质上是一个链栈，对应的存储管理方式在操作系统中称为“固定分区管理”。

第二种情况是系统运行期间用户请求分配的存储量有若干大小的固定规格。

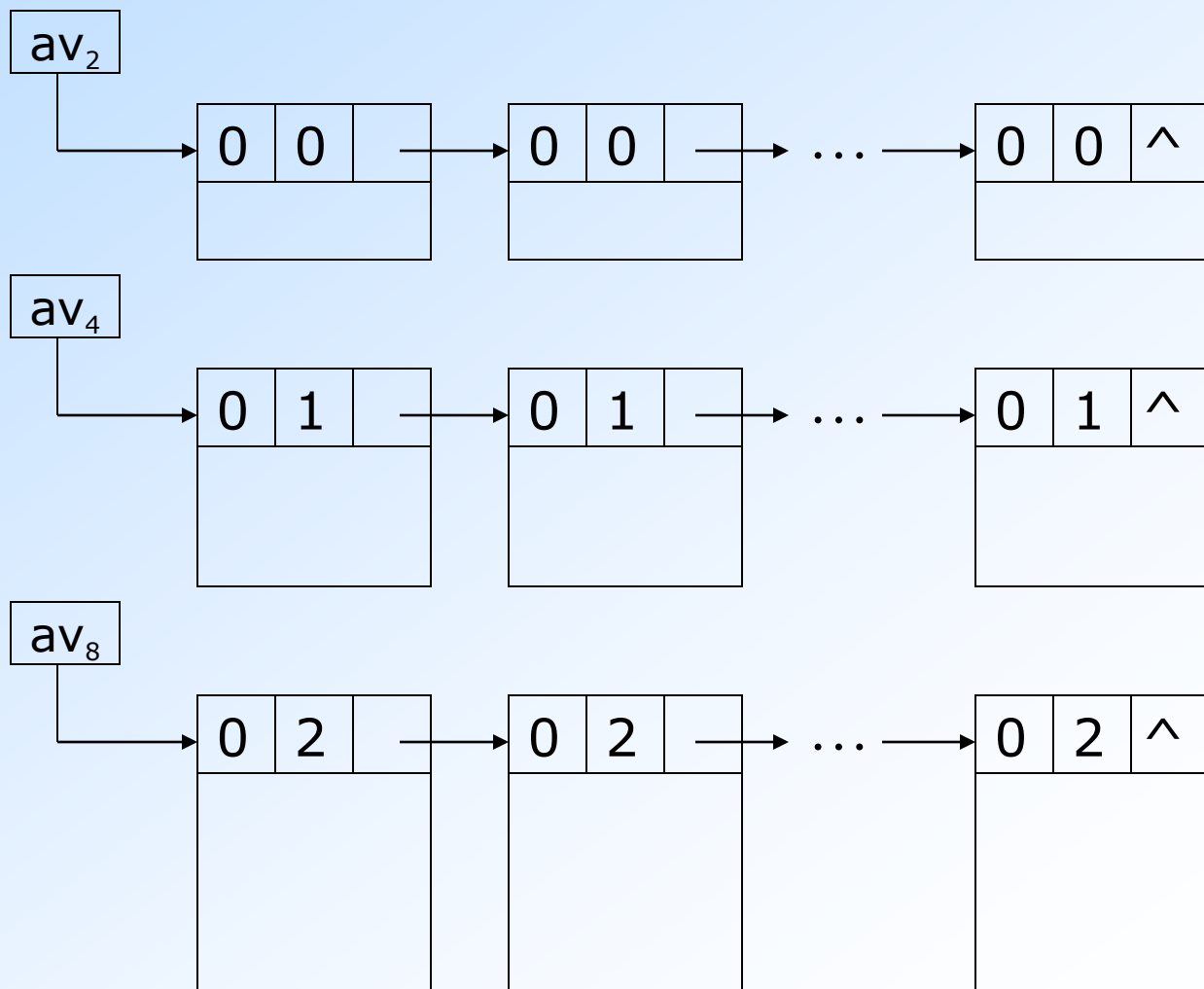
对此系统，可将用户存储空间分割成不同规格的若干块，并将大小相同的空闲块组织在同一个可利用空间表中，即同一链表中的结点大小相同。

例:



tag = $\begin{cases} 0 & \text{空闲块} \\ 1 & \text{占用块} \end{cases}$

type = $\begin{cases} 0 & \text{结点大小为2KB} \\ 1 & \text{结点大小为4KB} \\ 2 & \text{结点大小为8KB} \end{cases}$



第三种情况是系统在运行期间分配给用户的内存块大小不固定，可以随请求而变。此时，可利用空间表中的结点即空闲块的大小也是随意的。通常，操作系统中的可利用空间表属于这种类型，这种存储管理实际上就是**操作系统中的可变分区管理方法**。系统初始状态下，整个内存空间是一个空闲块，即可利用空间表中只有一个大小为整个内存区的结点，随着分配和回收的进行，可利用空间表中的结点大小和个数也随之而变。

由于链表中结点大小不同，结点的结构可包含四个域，即：标志域（tag），用于区分此块是否为空闲块、大小域（size），用于指示空闲块的存储量、链域（link），用于指示可利用空间链表中的下一个结点、存储空间域（space），它是一个大小为size的连续存储空间。

tag	size	link
space		

$\text{tag} = \begin{cases} 0 & \text{空闲块} \\ 1 & \text{占用块} \end{cases}$

由于可利用空间表中的结点大小不同，因此相应的分配与回收过程较为复杂。假设某用户需大小为 n 的内存，而可利用空间表中仅有一块大小为 $m \geq n$ 的空闲块，则只需将其中大小为 n 的一部分分配给申请的用户，同时将剩余大小为 $m-n$ 的部分作为一个结点留在链表中即可。当可利用空间表中存在多个空间大小不小于 n 的空闲块时，一般可采用以下三种不同的分配策略。

(1) **最先适应分配算法**，这种方法又称为首次适配法。每次分配时，总是顺序查找可利用空间链表，找到第一个能满足长度要求的空闲区为止。分割这个找到的未分配区，一部分分配给作业，另一部分仍为空闲区。

(2) **最优适应分配算法**。这种分配算法每次从空闲区中挑选一个能满足作业要求的最小分区，这样可保证不去分割一个更大的区域，使装入大作业比较容易得到满足。

(3) **最坏适应分配算法**。最坏适应分配算法总是挑选一个能满足作业要求的最大的空闲区分割给作业使用，这样可使剩下的空闲区不至于太小，这种算法对中、小作业是有利的。

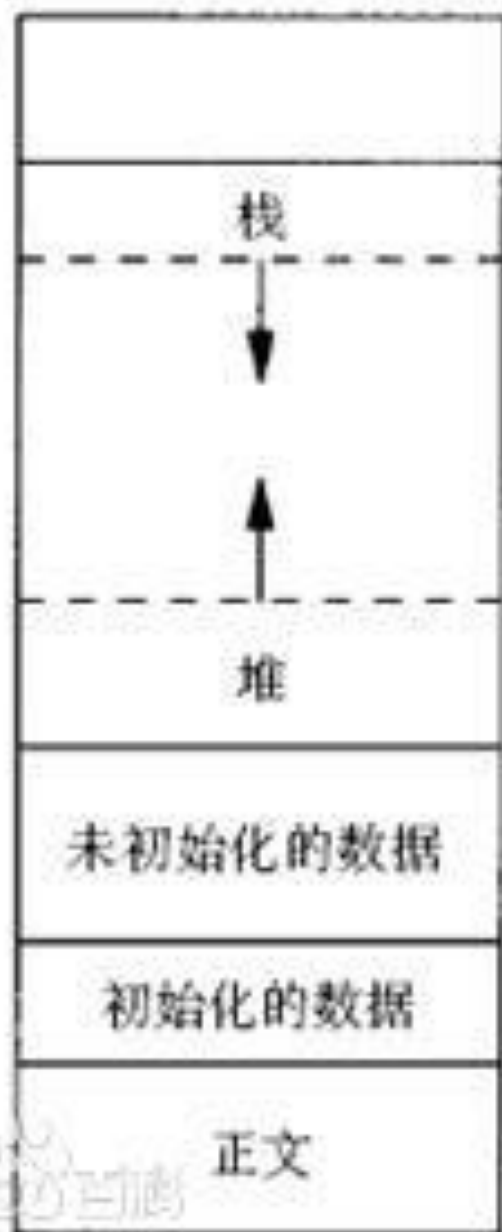
上述三种分配方法的选取一般需要考虑以下因素：用户的逻辑要求；请求分配量的大小分布；分配和释放的频率以及效率对系统的重要性等。

无论采用何种分配方法在进行回收系统空闲块时需要考虑“结点合并”的问题，即当系统在不断进行分配和回收的过程中，大的空闲块逐渐被分割成小的占用块，当用户程序将某一占用块释放重新成为空闲块时，如果将它作为一个独立的空闲块插入到链表中，将出现两个或多个地址相邻的空闲块作为几个结点独立放在可利用空间表中，显然这不利于以后出现的大容量作业的请求。为了更有效地利用内存，就要求系统在回收时应考虑将地址相邻的空闲块合并成尽可能大的结点。

由C/C++编译的程序占用的内存分为几个部分：

- 1、栈区 (stack) 由编译器自动分配释放，存放函数的参数名，局部变量的名等。其操作方式类似于数据结构中的栈。
- 2、堆区 (heap) 由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式类似于链表。
- 3、静态区 (static) 全局变量和局部静态变量的存储是放在一块的。程序结束后由系统释放。
- 4、文字常量区：常量字符串放在这里，程序结束后由系统释放。
- 5、程序代码区：存放函数体的二进制代码。

高地址



命令行参数和环境变量

栈

堆

未初始化的数据

初始化的数据

正文

由exec赋初值0

exec从程序文件中读到

低地址

8086/8088 的堆栈的组织

一、8086/8088的堆栈组织

1、堆栈是由SS指定的一段存储区域。

栈的最大空间是 64KB，栈的最大深度是 32K；

逻辑地址 SS: SP

二、8086/8088的堆栈操作

以16位二进制数进行操作。

- 地址最高的字单元叫作 “栈底 (Bottom) ” ；
- 入栈的数据从栈底开始逐个向地址低端存入；堆栈为空时SP指向栈底后第2个字节
- 堆栈的作用：
 - ①暂存数据
 - ②过程调用或处理中断时暂存断点信息。

8086/8088 的堆栈

PUSH — 压栈操作:

- ①将SP-1, 在指针单元中存储数据高字节
- ② 将SP-1, 在指针单元中存储数据低字节

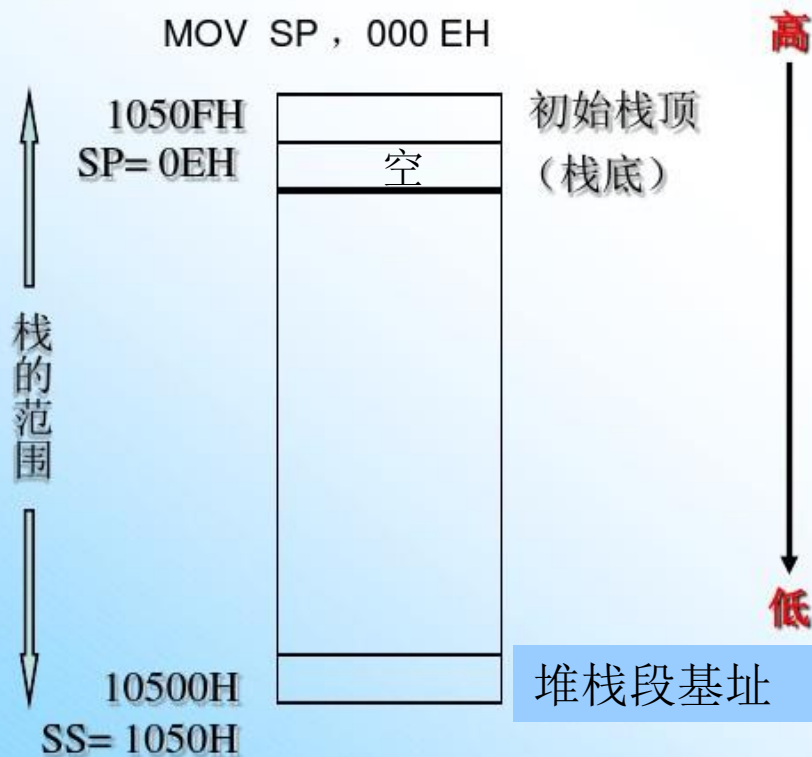
POP — 出栈操作:

- ①寄存器/字单元 $\leftarrow ((SP))$
将栈顶字单元内容送到指定的寄存器或字单元中。
 - 1)先将 (SP) 指针单元中的数据弹到低8位装置中。
 - 2)再将 (SP + 1) 指针单元中的数据弹到高8位装置中。
- ② (SP) + 2 \rightarrow SP
修改栈顶指针以指向新栈顶

8086/8088的堆栈：（向下生成）

8086通过赋值SS和SP建栈

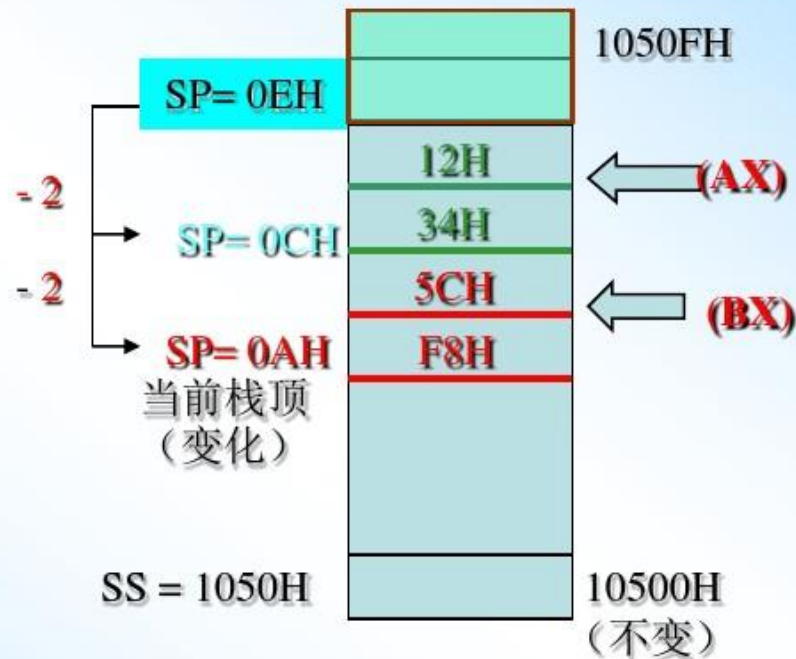
建栈 MOV AX, 1050H
 MOV SS, AX
 MOV SP, 000EH



进栈前SP和SS的值
SP=000EH SS=1050H

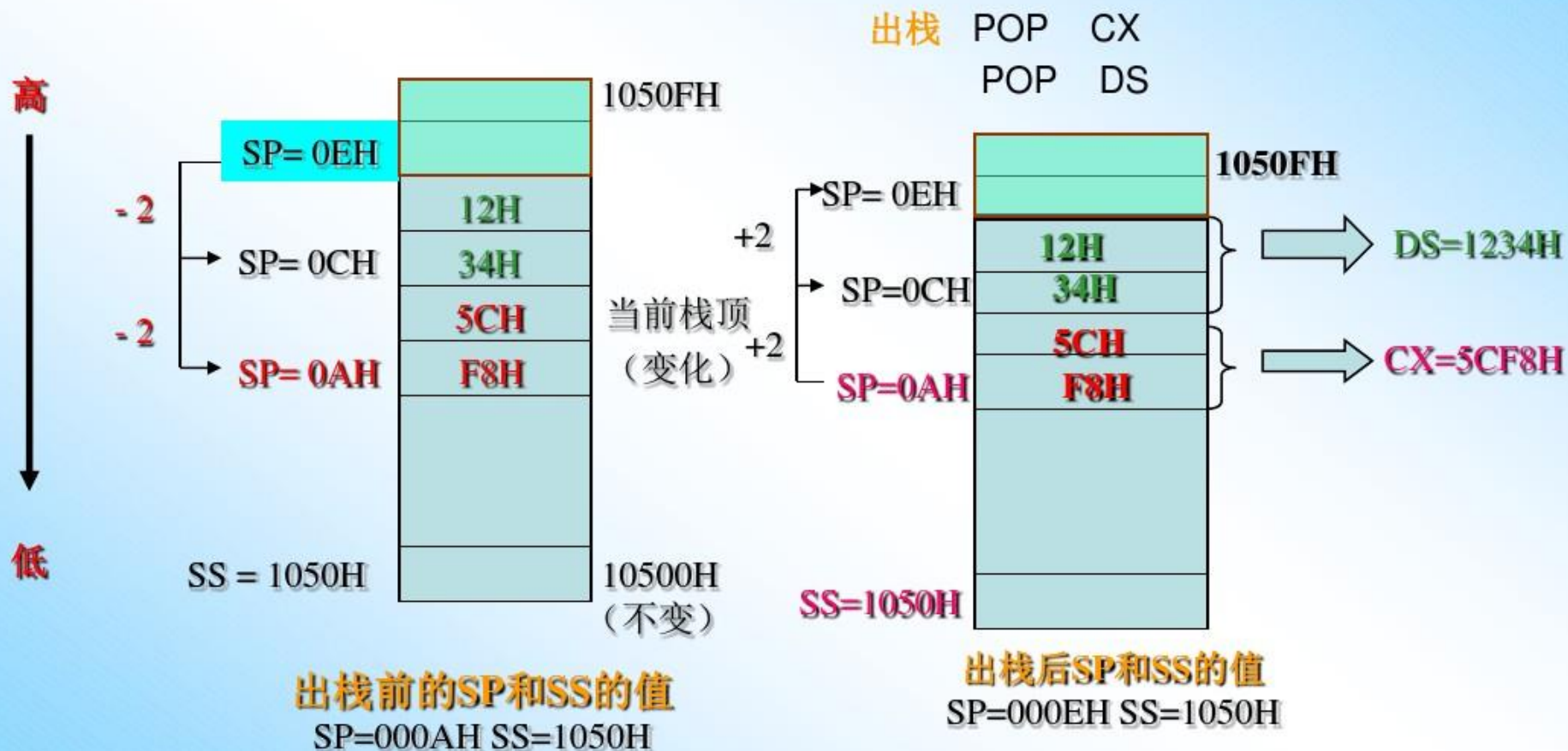
进栈操作：先SP减2，再内容进栈。

进栈 PUSH AX (设 AX=1234H)
 PUSH BX (设 BX=5CF8H)



进栈后的SP和SS的值
SP=000AH SS=1050H

出栈操作：先栈顶内容出栈，再修改SP，
使SP加2。（字操作）



栈的应用举例

- 数制转换
- 行编辑程序
- 迷宫求解
- 表达式求值

❖ 数制转换

$$N = (N \text{ div } d) \times d + N \text{ mod } d$$

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

	N	N div 8	N mod 8	
计算顺序 ↓	1348	168	4	↑ 输出顺序
	168	21	0	
	21	2	5	
	2	0	2	

```
main()
{ int N,e;
  SeqStack *S;
  S=(SeqStack *)malloc(sizeof(SeqStack));
  InitStack (S);

  printf("input the N ");
  scanf ("%d",&N);
  while (N) {
    Push(S, N % 8);
    N = N/8;
  }
  while (!StackEmpty(S)) {
    Pop(S,&e);
    printf ( "%d", e );
  }
```



```
#include <iostream>
#include <stack>
using namespace std;
```

```
void main()
{ stack<int> S;
  int N,e;
  cout<<"please input the data"<<endl;
  cin>>N;
  while (N) {
    S.push( N % 8);
    N = N/8;  }
  while (!S.empty()) {
    e=S.top();
    S.pop();
    cout<<e;    }
}
```

❖行编辑程序

在用户输入一行的过程中，允许 用户输入出差错，并在发现有误时可以及时更正。

设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区；并假设“#”为退格符，“@”为退行符。

假设从终端接受两行字符：

```
whli###ilr#e (s#*s)
```

```
outcha@putchar(*s=#++);
```

实际有效行为：

```
while (*s)
```

```
    putchar(*s++);
```

```
Void LineEdit(){
    InitStack(s);
    ch=getchar();
    while (ch != EOF) { //EOF为全文结束符
        while (ch != EOF && ch != '\n') {
            switch (ch) {
                case '#' : Pop(S, c); break;
                case '@': ClearStack(S); break;
                // 重置S为空栈
                default : Push(S, ch); break;
            }
            ch = getchar(); // 从终端接收下一个字符
        }
    }
```

将从栈底到栈顶的字符传送至调用过程的数据区;

ClearStack(S); // 重置S为空栈

if (ch != EOF) ch = getchar();

}

DestroyStack(s);

}

通常用的是“穷举求解”的方法

[illegible]

❖ 迷宫路径算法的基本思想

- 若当前位置“可通”，则纳入路径，继续前进；
- 若当前位置“不可通”，则后退，换方向继续探索；
- 若四周“均无通路”，则将当前位置从路径中删除出去。

设定当前位置的初值为入口位置;

do {

 若当前位置可通,

 则 {将当前位置插入栈顶;

 若该位置是出口位置, 则算法结束;

 否则切换当前位置的东邻方块为新的当前位置; }

否则 {若栈不空且栈顶位置尚有其他方向未被探索,

 则设定 新的当前位置为: 沿顺时针方向旋转找到的栈顶位置的下一相邻块;

 否则 {删去栈顶位置; **//栈不空但栈顶四周均不通,**
 若栈不空, 则重新测试新的栈顶位置, 直至找到一个可通的相邻块,或出栈至栈空;

}

} while (栈不空) ;

若栈不空且栈顶位置尚有其他方向未被探索，
则设定新的当前位置为：沿顺时针方向旋转

找到的栈顶位置的下一相邻块；

若栈不空但栈顶位置的四周均不可通，
则 {

 删去栈顶位置； // 从路径中删去该通道块

 若栈不空，则重新测试新的栈顶位置，
 直至找到一个可通的相邻块或出栈至栈空；

}

表达式求值

限于二元运算符的表达式定义:

表达式 ::= (操作数) + (运算符) + (操作数)

操作数 ::= 简单变量 | 表达式

简单变量 ::= 标识符 | 无符号整数

$\text{Exp} = S1 + OP + S2$

前缀表示法 $OP + \underline{S1} + \underline{S2}$

中缀表示法 $\underline{S1} + OP + \underline{S2}$

后缀表示法 $\underline{S1} + \underline{S2} + OP$

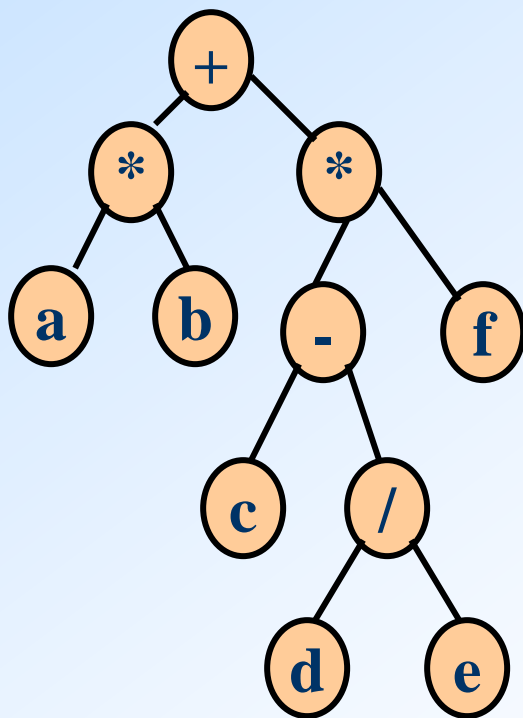
表达式标识方法

例如: $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$

前缀式: $+ \underline{\times a b} \underline{\times - c / d e f}$

中缀式: $\underline{a \times b} + \underline{c - d / e \times f}$

后缀式: $\underline{a b \times} \underline{c d e / - f \times} +$

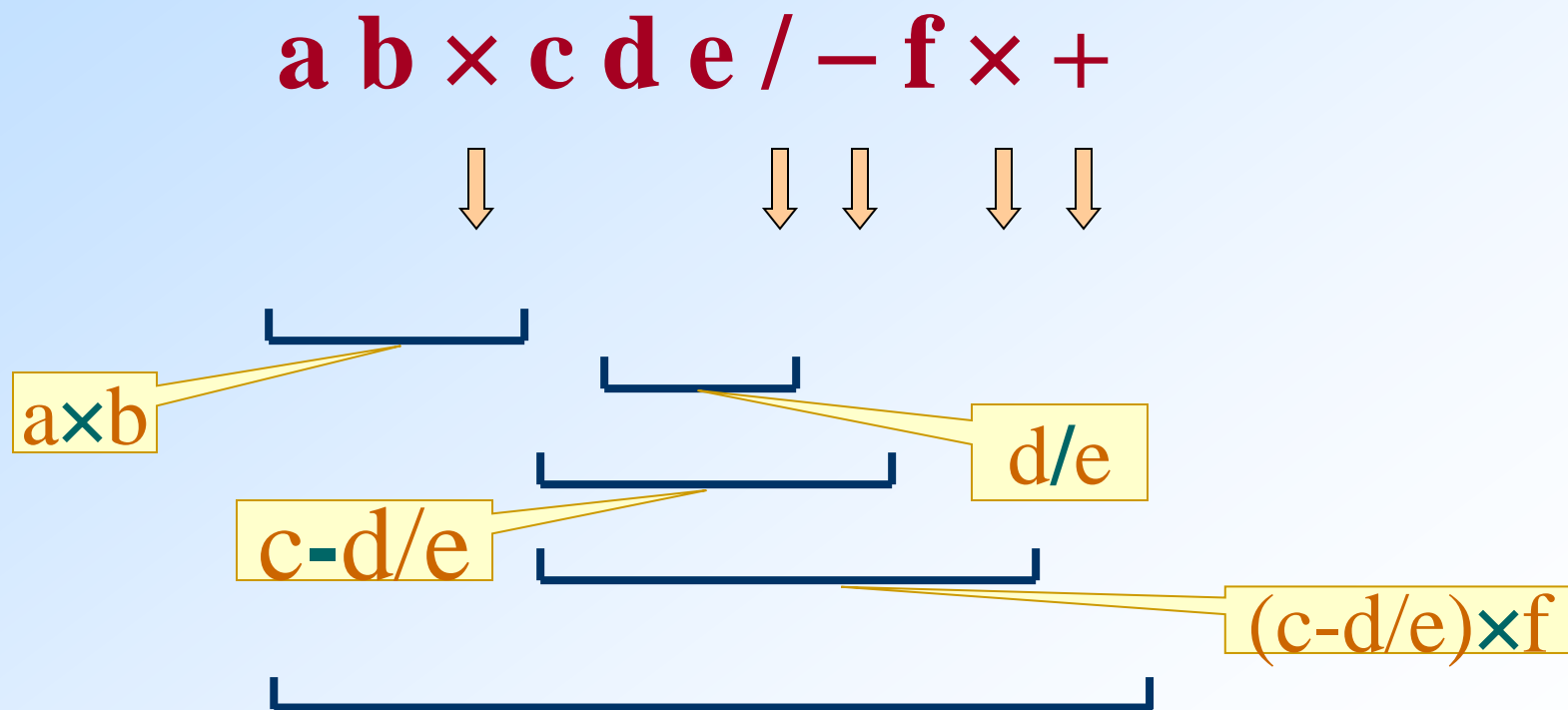


后缀表达式求值

- 建立一个栈S。从左到右读表达式，如果读到操作数就将它压入栈S中，
- 如果读到n元运算符(即需要参数个数为n的运算符)则取出由栈顶向下的n项按操作符运算，再将运算的结果代替原栈顶的n项，压入栈S中。如果后缀表达式未读完，则重复上面过程，最后输出栈顶的数值则为结束。

后缀表达式求值

例如：



中缀表达式 $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$
 $= \underline{2 \times 3} + \underline{(5 - 8 / 2) \times 6} = 12$

从原表达式求得后缀表达式

- 1) 设立暂存运算符的栈;
- 2) 设表达式的结束符为 “#”, 预设运算符栈的栈底为 “#”
- 3) 若当前字符是操作数, 则直接发送给后缀式;
- 4) 若当前运算符的优先数高于栈顶运算符, 则进栈;
- 5) 否则, 退出栈顶运算符发送给后缀式;
- 6) 若为 '(', 入栈;
- 7) 若为 ')', 则依次把栈中的运算符加入后缀表达式中, 直到出现 '(', 从栈中删除 '('

表达式标识方法

例如: $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$

中缀式: $\underline{a \times b} + \underline{(c - d / e) \times f}$

后缀式: $\underline{a \ b \times} \ \underline{c \ d \ e \ / - f \times} \ +$

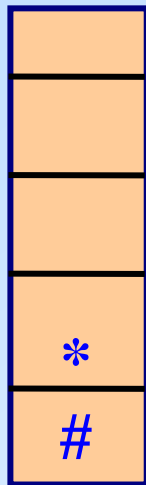
中缀表达式 $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$
 $= \underline{2 \times 3} + \underline{(5 - 8 / 2) \times 6} = 12$

2 3

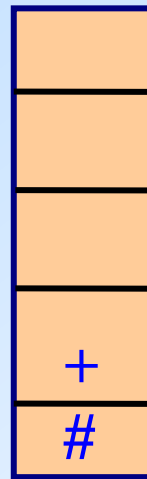
2 3 *

2 3 * 5

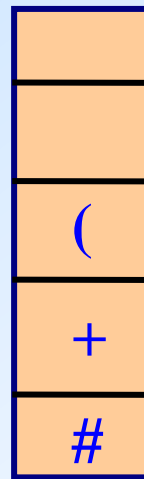
2 3 * 5 8



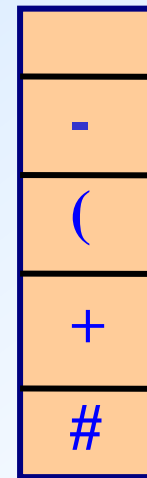
“+”



“(”



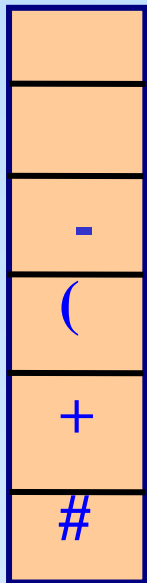
“-”



“/”

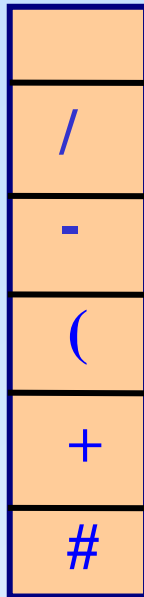
中缀表达式 $\text{Exp} = \underline{a \times b} + \underline{(c - d / e) \times f}$
 $= \underline{2 \times 3} + \underline{(5 - 8 / 2) \times 6} = 12$

2 3 * 5 8



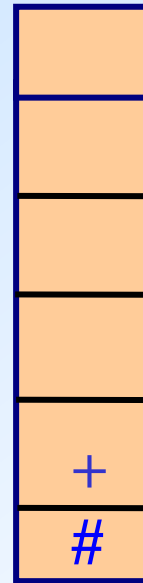
“/”

2 3 * 5 8 2

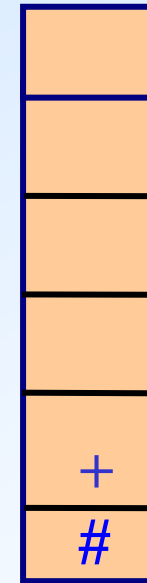


“)”

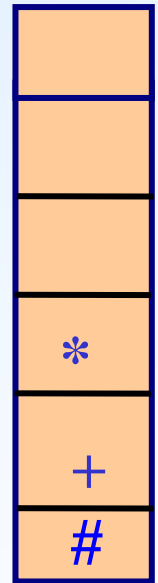
2 3 * 5 8 2 / -



2 3 * 5 8 2 / - 6 * +



“*”



“#”

用栈实现将中缀表达式

$8-(3+5)*(5-6/2)$ 转换成后缀表达式。

转换成后缀表达式 $835+562/-*-$

递归

- **定义** 若一个对象部分地包含它自己, 或用它自己给自己定义, 则称这个对象是递归的; 若一个过程直接地或间接地调用自己, 则称这个过程是递归的过程。
- **三种递归情况**
 - 定义是递归的
 - 数据结构是递归的
 - 问题的解法是递归的

定义是递归的

例1.阶乘函数

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

求解阶乘函数的递归算法

```
long Factorial ( long n ) {  
    if ( n == 0 ) return 1;  
    else return n * Factorial (n-1);  
}
```

求解阶乘 $n!$ 的过程



递归过程与递归工作栈

- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，退出时的次序正好相反：

递归调用

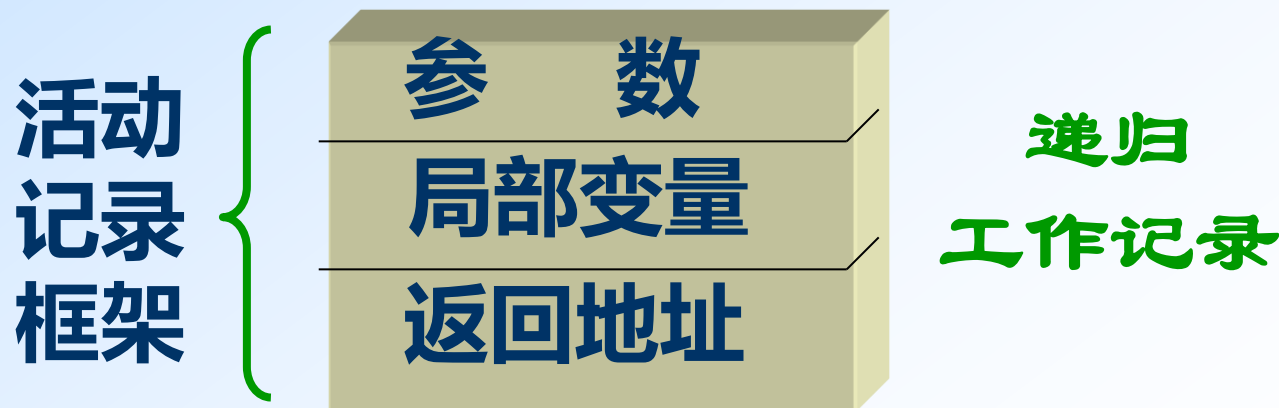
$n! \longrightarrow (n-1)! \longrightarrow (n-2)! \Rightarrow \dots \Rightarrow 1! \longrightarrow 0!=1$

返回次序

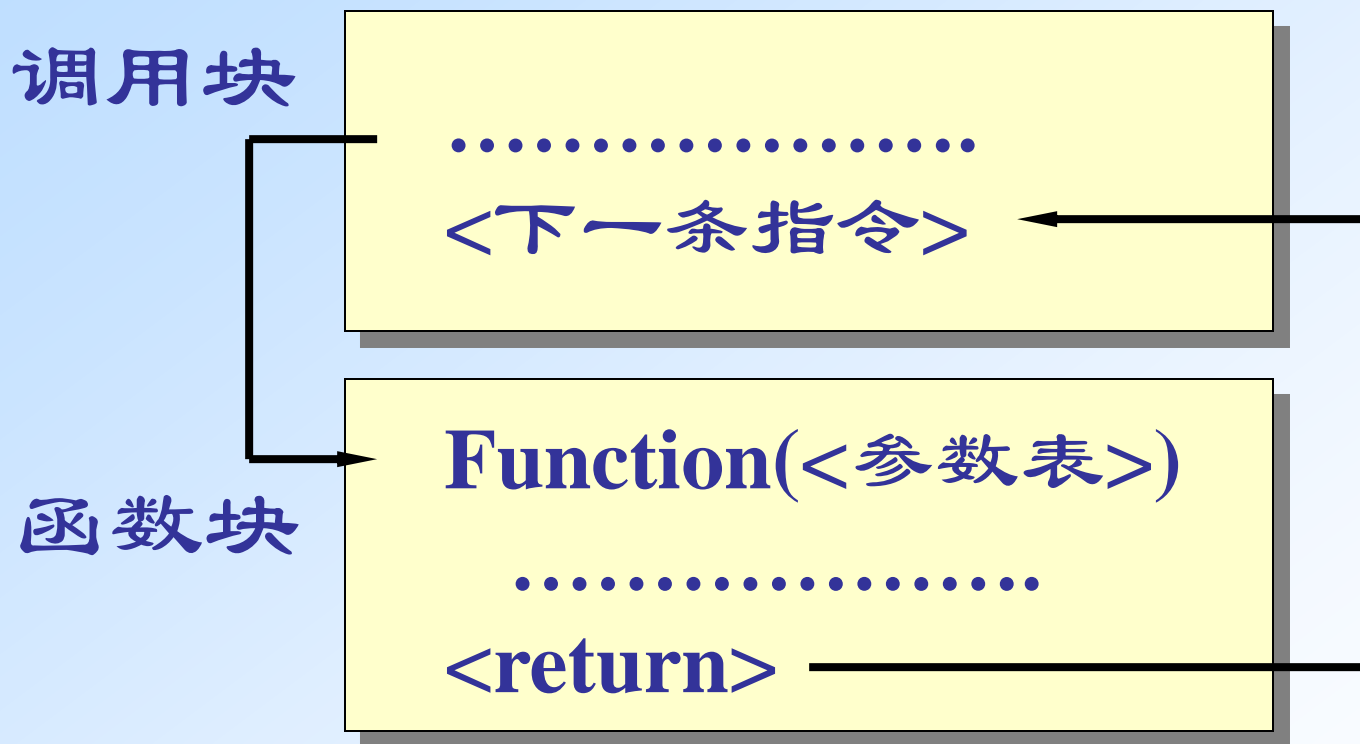
- 主程序第一次调用递归过程为外部调用；
- 递归过程每次递归调用自己为内部调用。

递归工作栈

- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。



函数递归时的活动记录

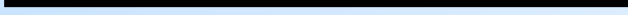


返回地址(下一条指令)

局部变量

参数

```
long Factorial ( long n ) {  
    int temp;  
    if ( n == 0 ) return 1;  
    else temp = n * Factorial (n-1);  
    return temp;  
}
```

RetLoc2 

```
void main ( ) {  
    int n;  
    n = Factorial (4);  
}
```

RetLoc1 

计算Fact时活动记录的内容

递归调用序列

参数 返回地址

返回时的指令

4 RetLoc1

RetLoc1 return $4*6$ //返回24

3 RetLoc2

RetLoc2 return $3*2$ //返回6

2 RetLoc2

RetLoc2 return $2*1$ //返回2

1 RetLoc2

RetLoc2 return $1*1$ //返回1

0 RetLoc2

RetLoc2 return 1 //返回1

思考：求算法的时间复杂度

$$\begin{aligned}T(n) &= T(n-1)+1 \\&= T(n-2)+1+1 \\&= T(1)+n+1 \\&= O(n)\end{aligned}$$

(略) 递归过程改为非递归过程

- **递归过程简洁、易编、易懂**
- **递归过程效率低，重复计算多**
- **改为非递归过程的目的是提高效率**
- **单向递归和尾递归可直接用迭代实现其非递归过程**
- **其他情形必须借助栈实现非递归过程**

例2.计算斐波那契数列函数Fib(n)的定义

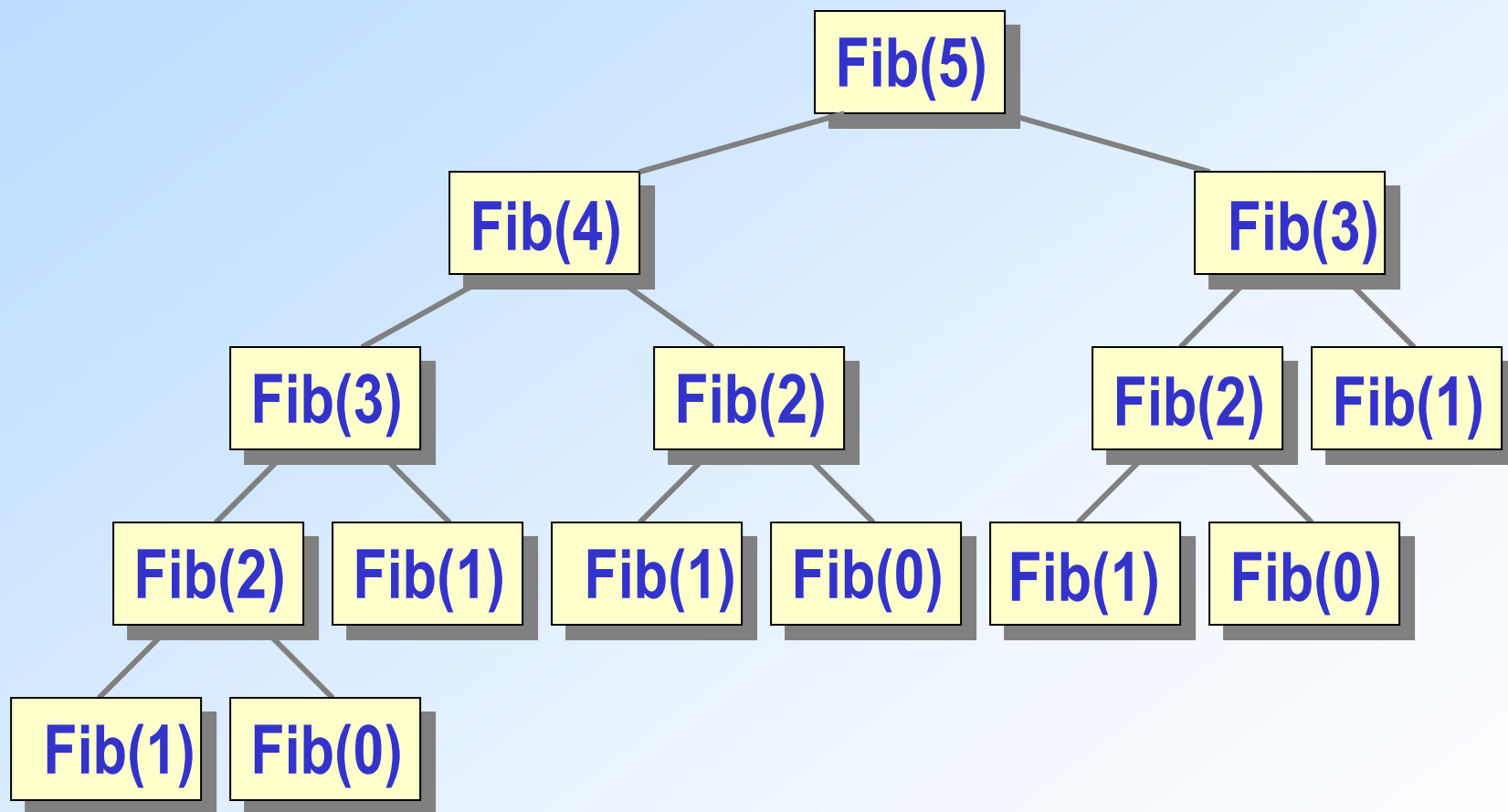
$$\text{Fib}(n) = \begin{cases} n, & n = 0, 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2), & n > 1 \end{cases}$$

如 $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5$

递归算法

```
long Fib ( long n ) {  
    if ( n <= 1 ) return n;  
    else return Fib (n-1) + Fib (n-2);  
}
```

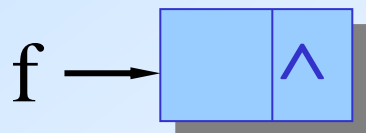
❖ 斐波那契数列的递归调用树



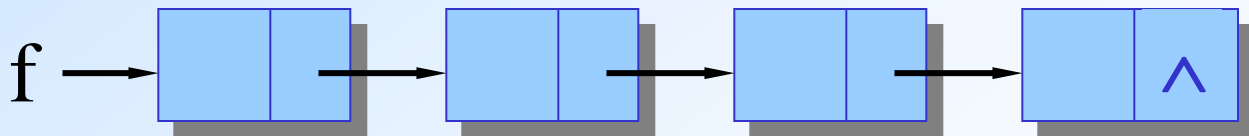
数据结构是递归的

例如单链表结构

只有一个结点的单链表



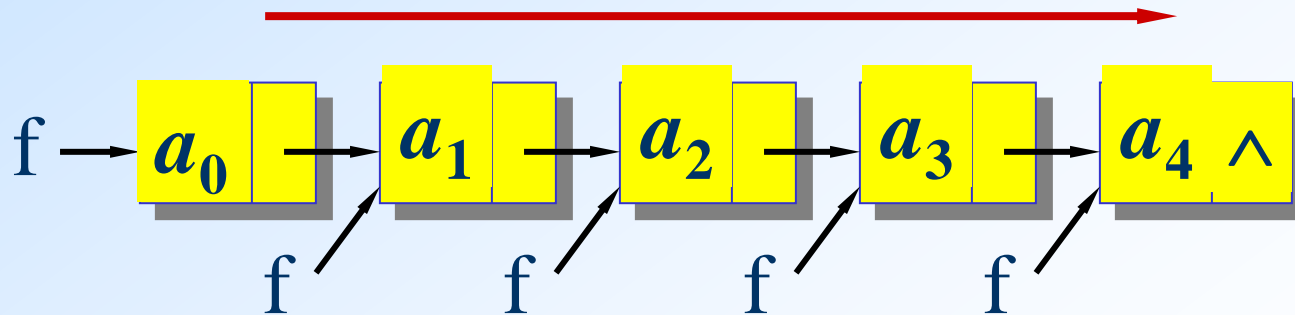
有若干结点的单链表



例1. 搜索链表最后一个结点并打印其数值

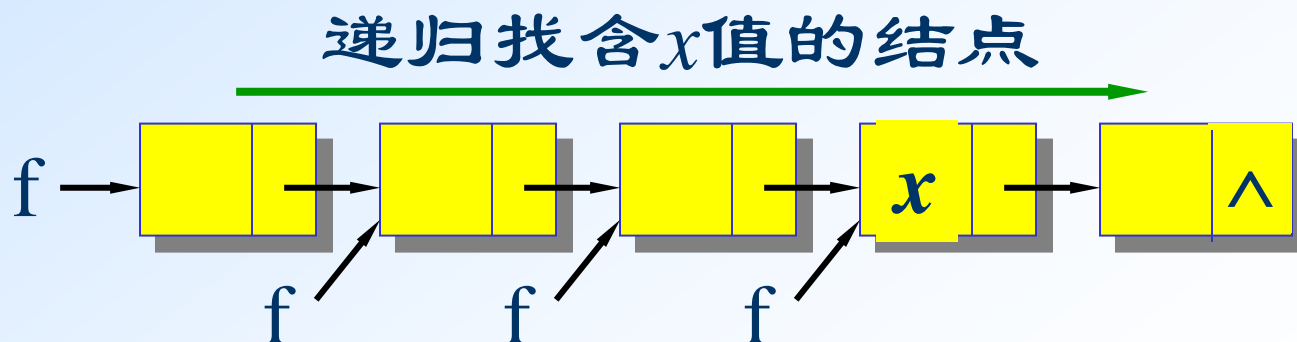
```
void Search ( ListNode *f ) {  
    if ( f ->next == NULL )  
        printf ("%d\n", f ->data );  
    else Search ( f ->next );  
}
```

递归找链尾



例2.在链表中寻找等于给定值的结点,并打印其数值

```
void Search ( ListNode *f, Elemtype x ) {  
    if ( f != NULL )  
        if ( f -> data == x )  
            printf ( “%d\n”, f -> data );  
        else Search ( f -> next, x );  
}
```



- **汉诺塔问题**是印度的一个古老的传说。开天辟地的神勃拉玛在一个庙里留下了三根金刚石的棒，第一根上面套着64个圆的金片，最大的一个在底下，其余一个比一个小，依次叠上去，庙里的众僧不倦地把它们一个个地从这根棒搬到另一根棒上，规定可利用中间的一根棒作为帮助，但每次只能搬一个，而且大的不能放在小的上面。
- 现在则演变为N阶的汉诺塔问题，三根金刚石棒变为X,Y,Z的塔座，在塔座X上插有N个直径大小不同，依次编号为1, 2, ..., n的圆盘，要求将X轴上的圆盘均移至塔座Z上，且按同样顺序排列。规则和古老传说一致。



❖ 问题的解法是递归的

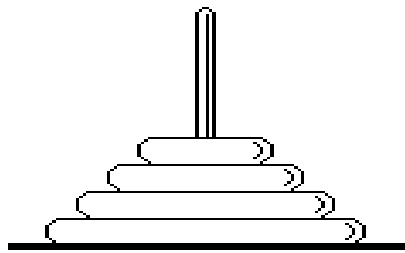
例如，汉诺塔(Tower of Hanoi)问题的解法：

如果 $n = 1$ ，则将这一个盘子直接从 A 柱移到 C 柱上。否则，执行以下三步：

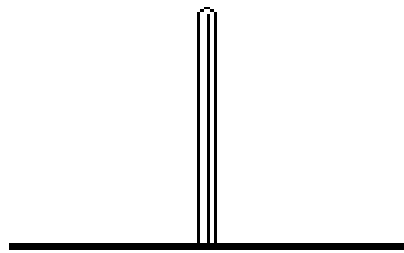
① 用 C 柱做过渡，将 A 柱上的 $(n-1)$ 个盘子移到 B 柱上：

② 将 A 柱上最后一个盘子直接移到 C 柱上；

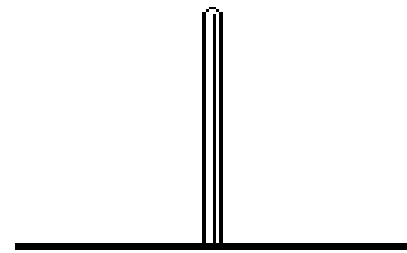
③ 用 A 柱做过渡，将 B 柱上的 $(n-1)$ 个盘子移到 C 柱上。



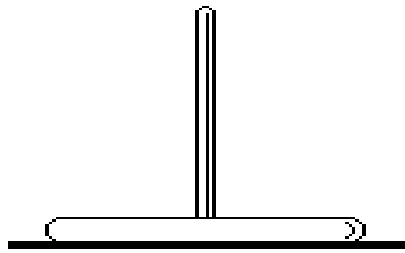
A



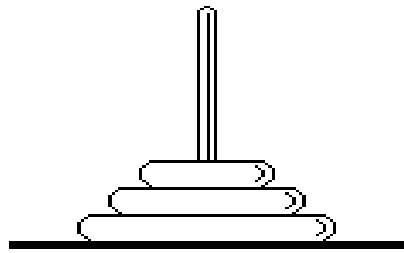
B



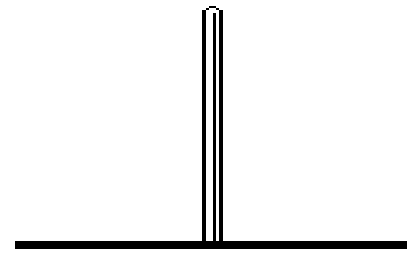
C



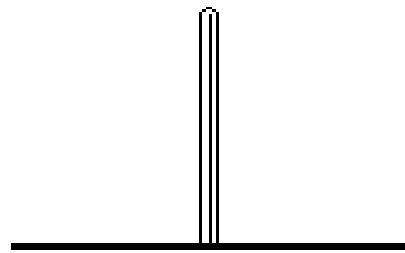
A



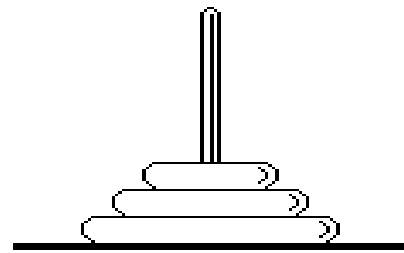
B



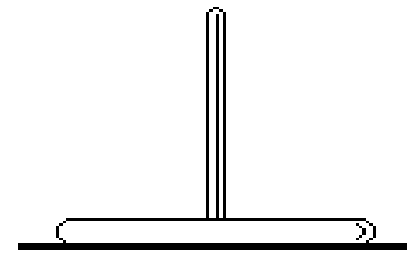
C



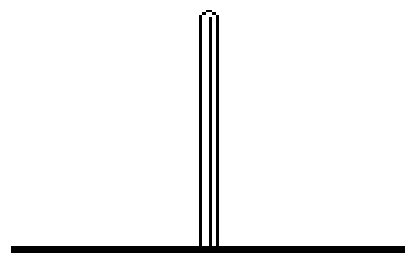
A



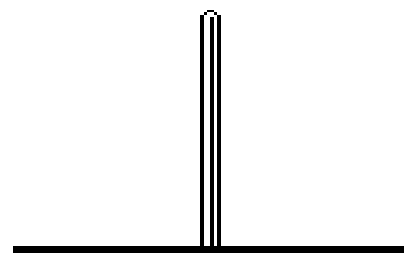
B



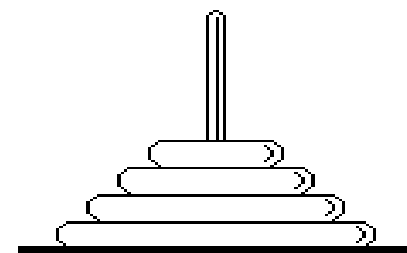
C



A



B



C

■ 算法

```
void Hanoi (int n, char X, char Y, char Z) {  
    //解决汉诺塔问题的算法  
    if ( n == 1 ) printf ( " %c→%c\n", X, Z);  
    else { Hanoi ( n-1, X, Z, Y );  
           printf ( " %c→%c\n",X, Z);  
           Hanoi ( n-1, Y, X, Z );  
        }  
}
```

```
void Hanoi (int n, char X, char Y, char Z) {
```

```
//解决汉诺塔问题的算法
```

```
if ( n == 1 ) printf ( " %c→%c\n", X, Z);
```

```
else { Hanoi ( n-1, X, Z, Y );
```

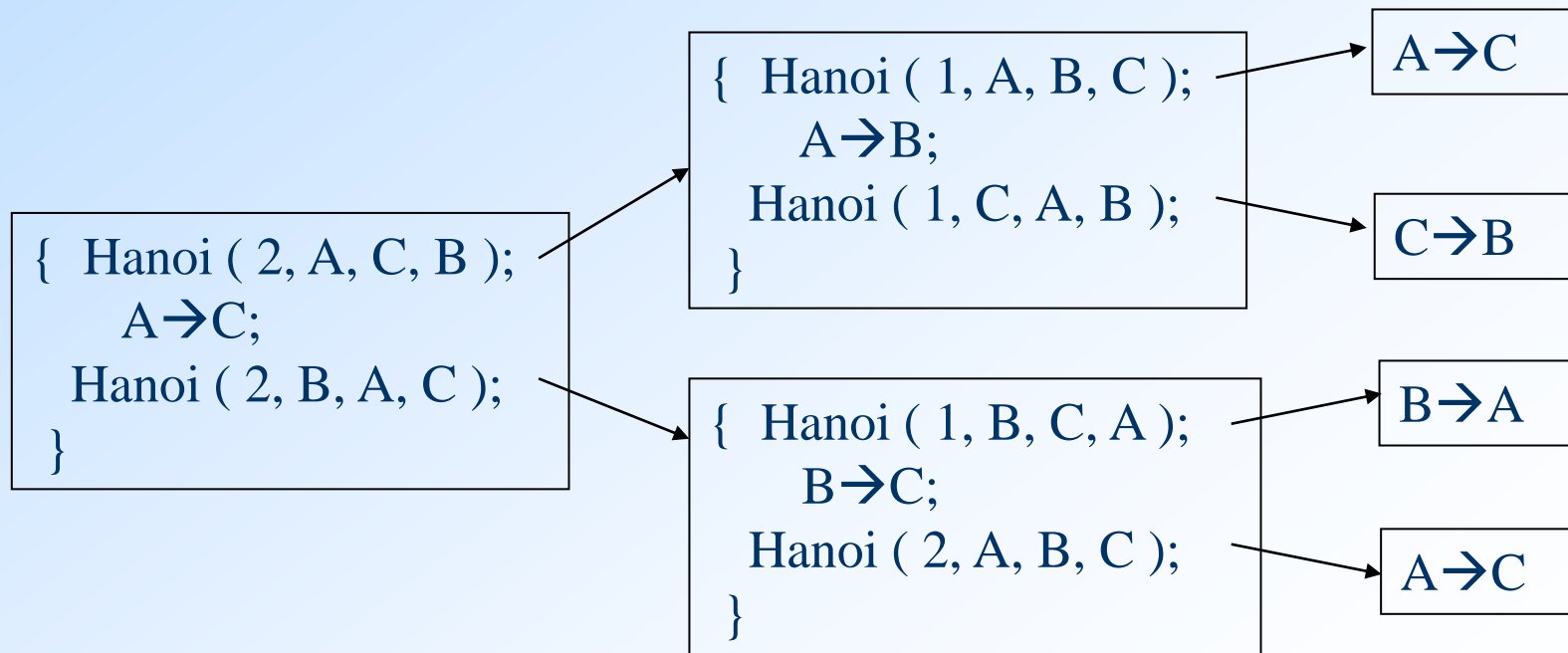
```
printf ( " %c→%c\n", X, Z);
```

```
Hanoi ( n-1, Y, X, Z );
```

```
}
```

```
}
```

例如：主程序调用Hanoi (3, 'A', 'B', 'C');



思考：汉诺塔问题中，若移动N个圆盘，
总共需要移动多少次？

$$2^n - 1$$

提示： $T(n) = 2T(n-1) + 1$

To iterate is human, to recurse divine.

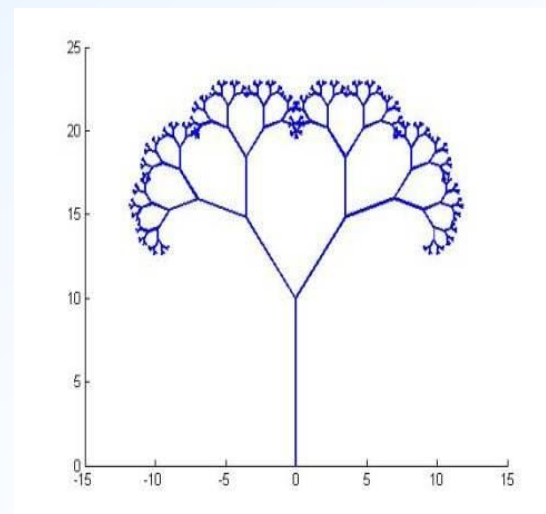
——L. Peter Deutsch

迭代乃人工，递归方神通。

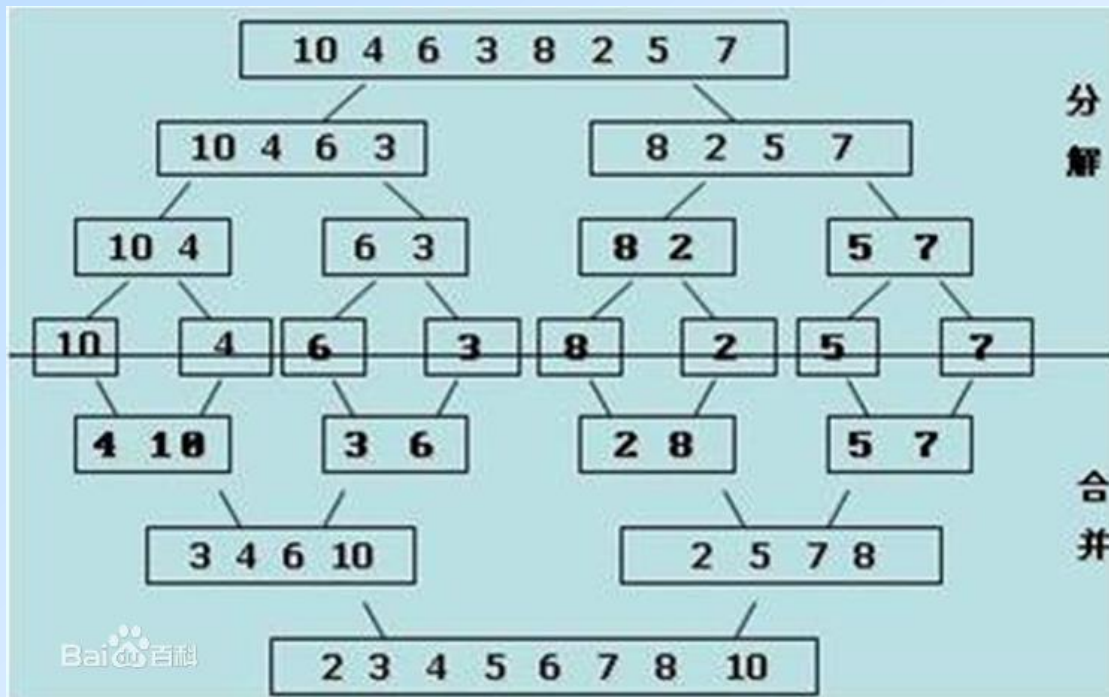
“迭代 (*iterate*) 者为人，递归 (*recurse*) 者为神。”

分形树：自相似递归图形

- 分形Fractal，是1975年由Mandelbrot开创的新学科.通常被定义为“一个粗糙或零碎的几何形状，可以分成数个部分，且每一部分都（至少近似地）是整体缩小后的形状”，即具有自相似的性质。
- 自然界中能找到众多具有分形性质的物体: 海岸线、山脉、闪电、云朵、雪花、树.分形是在不同尺度上都具有相似性的事物，一棵树的每个分叉和每条树枝，实际上都具有整棵树的外形，可以把树分解为三个部分：树干、左边的小树、右边的小树。



❖ 递归和分治



分治法

直接或间接的调用自身的算法称为递归算法。与分治法像一对孪生兄弟，经常同时应用在算法设计中，并由此产生许多高效的算法。

分治法的设计思想是：一个先自顶向下，再自底向上的过程。

分——将问题分解为规模更小的子问题；

治——将这些规模更小的子问题逐个击破；

合——将已解决的子问题合并，最终得出“母”问题的解；

分治法与递归的联系

由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。在分治策略中，由于子问题与原问题在结构和解法上的相似性，用分治方法解决的问题，大都采用了递归的形式。在各种排序方法中，如归并排序、堆排序、快速排序等。

分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

- 1) 该问题的规模缩小到一定的程度就可以容易地解决
- 2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。（递归思想）
- 3) 利用该问题分解出的子问题的解可以合并为该问题的解；
- 4) 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

第三条是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。

第四条特征涉及到分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

❖ 递归与回溯

n皇后问题

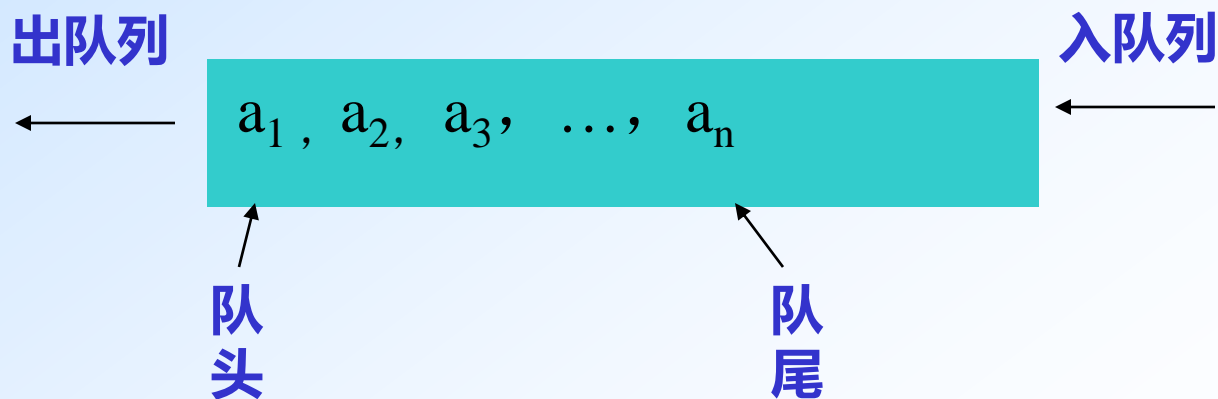
在 n 行 n 列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。n皇后问题是指找到这 n 个皇后的互不攻击的布局。

解题思路

- 安放第 i 行皇后时，需要在列的方向从 0 到 $n-1$ 试探 ($j = 0, \dots, n-1$)
- 在第 j 列安放一个皇后：
 - ◆ 如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。
 - ◆ 如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 $i+1$ 行皇后。

队列

- **定义:**只允许在表的一端进行插入，而在另一端删除元素的线性表。
- 在队列中，允许插入的一端叫队尾 (rear)
- ，允许删除的一端称为对头(front)。
- **特点:** 先进先出 (FIFO)



操作系统在管理和分配系统资源时，大量的应用了队列这种数据结构。

1) 队列在输入/ 输出管理中的应用

例如，当多个任务分配给打印机时，为了防止冲突，创建一个队列，把任务入队，按先入先出的原则处理任务。

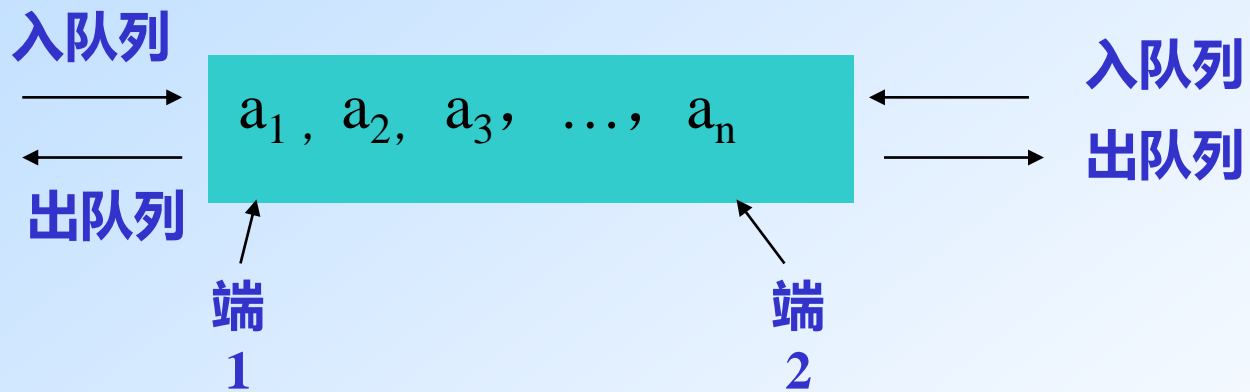
当多个用户要访问远程服务端的文件时，也用到队列，满足先来先服务的原则。

2) 对CPU 的分配管理

每个CPU都会维持一个运行队列，理想情况下，调度器会不断让队列中的进程运行。进程不是处在sleep状态就是run able状态。如果CPU过载，就会出现调度器跟不上系统的要求，导致可运行的进程会填满队列。队列愈大，程序执行时间就愈长。

有一个数学的分支，叫队列理论（queuing theory）。用来计算 预测用户在队中的等待时间，队的长度等等问题。答案取决于用户到达队列的频率，用户的任务的处理时间。

双端队列



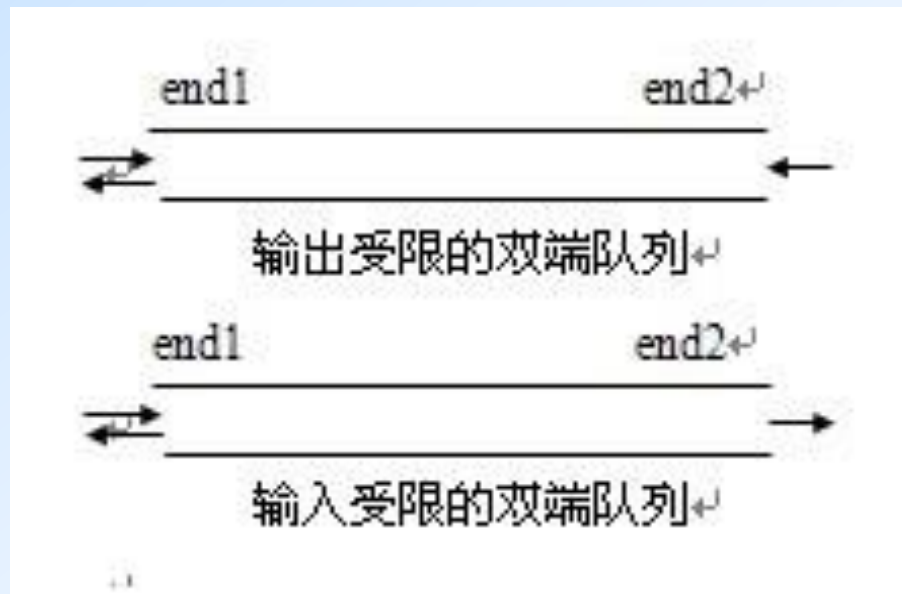
思考：若a,b,c,d元素进队，能否通过双端队列产生dacb的出队序列？

d a b c

受限的双端队列

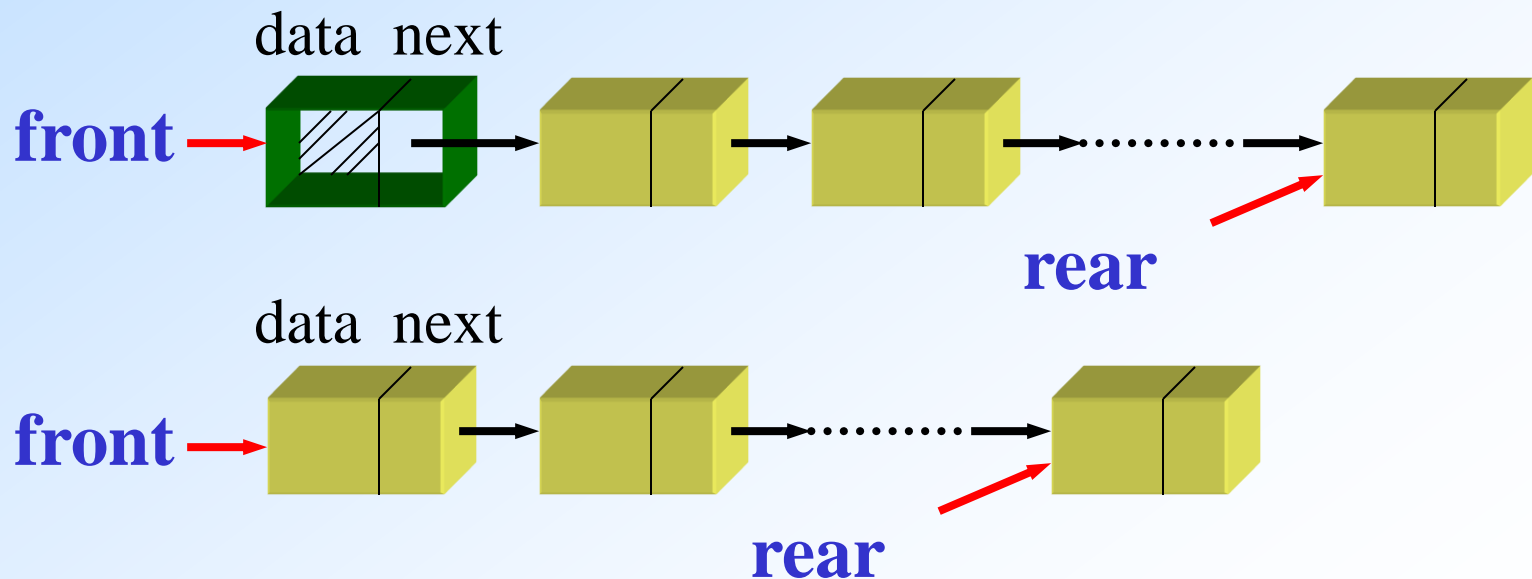
允许在一端进行插入和删除，但在另一端只允许插入的双端队列叫做输出受限的双端队列。

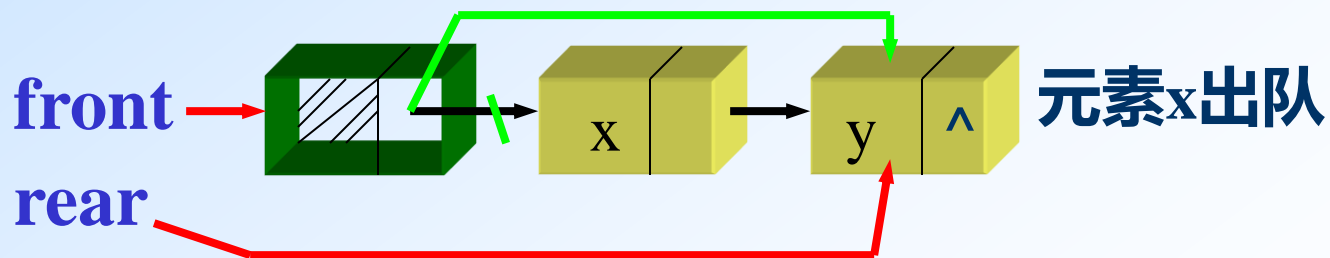
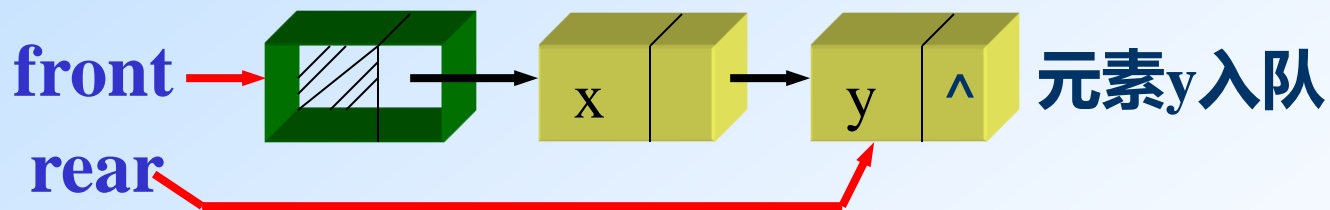
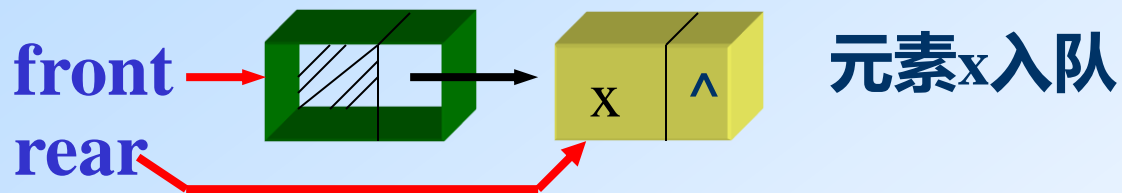
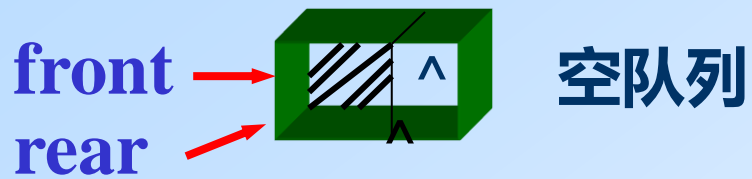
允许在一端进行插入和删除，但在另一端只允许删除的双端队列叫做输入受限的双端队列。



链队列：队列的链式表示

- 链队列中，有两个分别指示队头和队尾的指针。
- 链式队列在进队时无队满问题，但有队空问题。





链式队列的定义

```
typedef int QueueData;
```

```
typedef struct node {  
    QueueData data;  
    struct node *link;  
} QueueNode;
```

//队列结点数据
//结点链指针

```
typedef struct {  
    QueueNode *rear, *front;  
} LinkQueue;
```

链队列的主要操作

■ 初始化

```
void InitQueue ( LinkQueue *Q ) {
```

```
    Q->rear = Q->front = ( QueueNode * ) malloc  
        ( sizeof ( QueueNode ) );
```

```
    If (! Q->front ) exit;
```

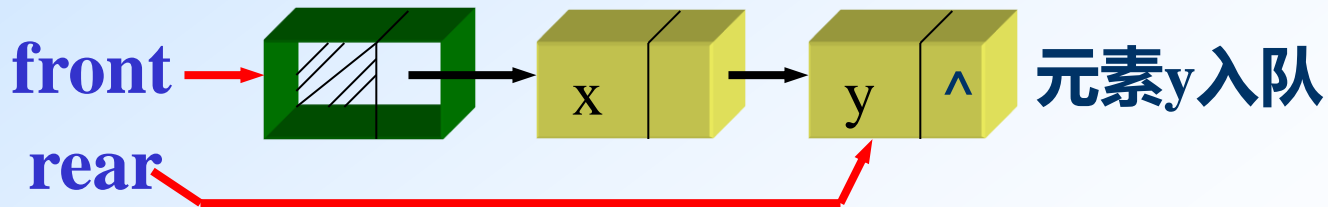
```
    Q->front ->link = NULL;  
}
```

■ 取队头元素

```
int GetFront ( LinkQueue *Q, QueueData *x ) {  
    if ( QueueEmpty (Q) ) return 0;  
    (*x) = Q->front ->link ->data; return 1;  
}
```

■ 入队

```
int EnQueue ( LinkQueue *Q, QueueData y ) {  
    QueueNode *p = ( QueueNode * ) malloc  
        ( sizeof ( QueueNode ) );  
    p->data = y; p->link = NULL;  
    Q->rear->link = p; //入队  
    Q->rear = p;  
    return 1;  
}
```



■ 出队

```
int DeQueue ( LinkQueue *Q, QueueData *x) {
```

```
//删去队头结点，并返回队头元素的值
```

```
    if (Q->rear == Q->front ) return 0;    //判队空
```

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

```
    *x = p->data; //保存队头的值
```

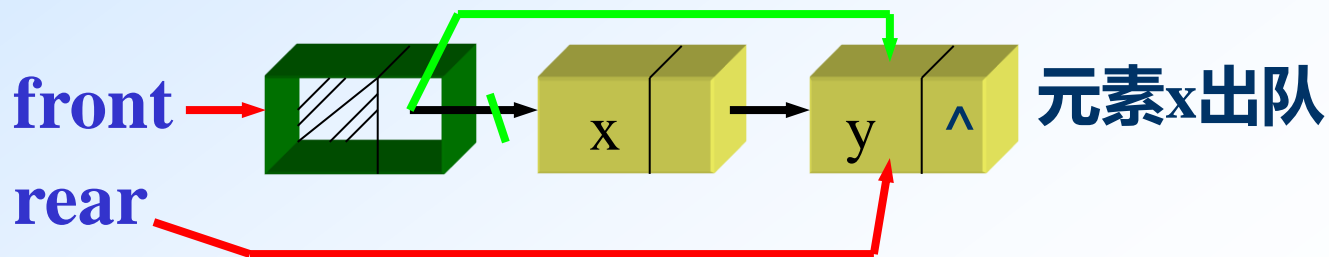
```
    Q->front ->link = p->link; //新队头
```

```
    if (Q->rear == p) Q->rear = Q->front
```

```
    free (p);
```

```
    return 1;
```

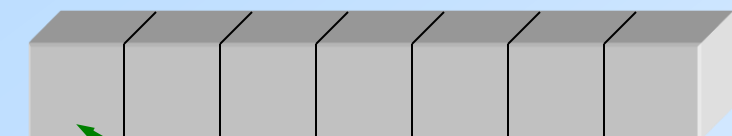
```
}
```



循环队列 (Circular Queue)

- **顺序队列——队列的顺序存储表示。** 用一组地址连续的存储单元依次存放从队列头到队列尾的元素，指针front和rear分别指示队头元素和队尾元素的位置。
- 插入新的队尾元素，尾指针增1， $\text{rear} = \text{rear} + 1$,
- 删除队头元素，头指针增1， $\text{front} = \text{front} + 1$,
- 因此，在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置。
- 队满时再进队将溢出
- 解决办法：将顺序队列臆造为一个环状的空间，形成循环(环形)队列

队列的进队和出队



front rear 空队列



front rear

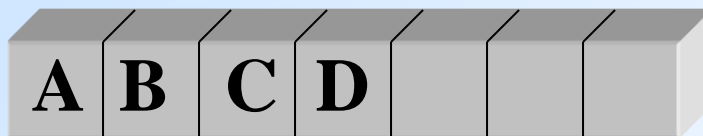
A,B出队



front

rear

H进队,溢出



front

rear

A,B,C,D进队



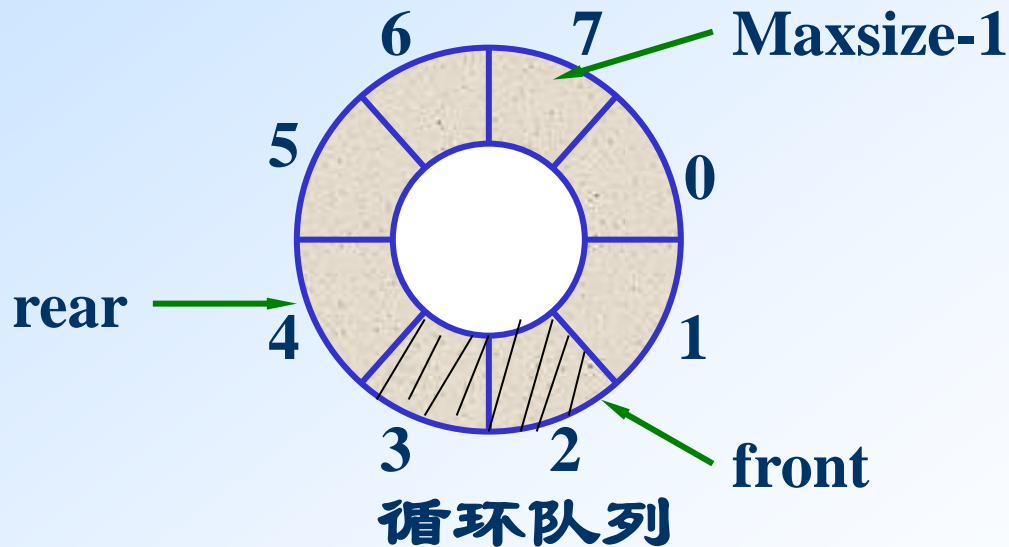
front

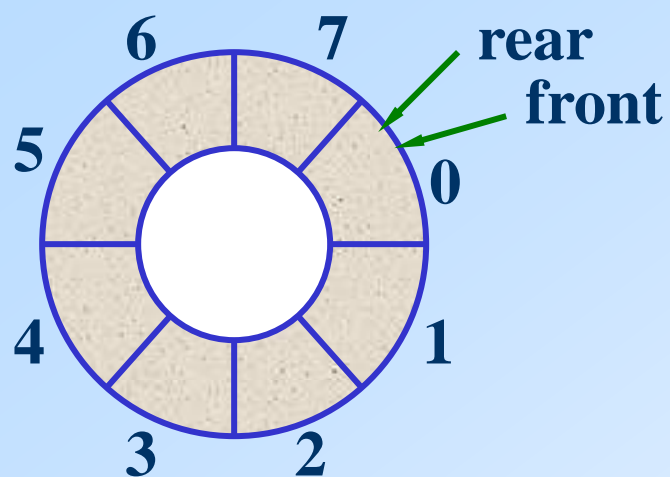
rear

E,F,G进队

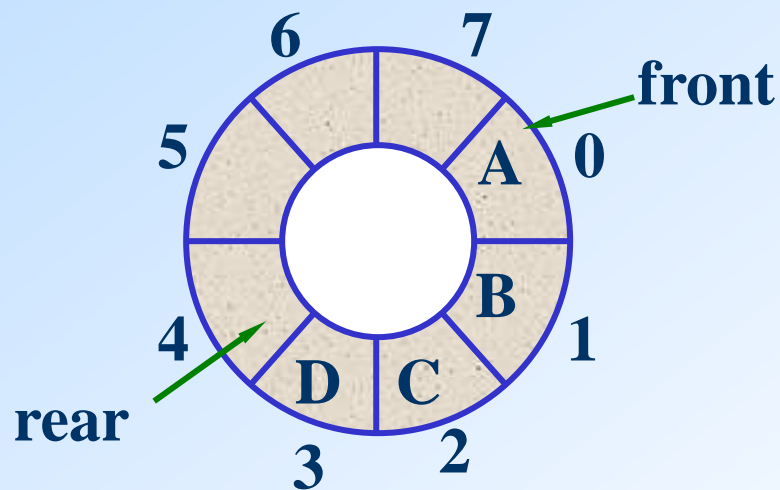
循环队列 (Circular Queue)

- 队头、队尾指针加1, 可用取模(余数)运算实现。
- 队头指针进1: $\text{front} = (\text{front} + 1) \% \text{maxsize};$
- 队尾指针进1: $\text{rear} = (\text{rear} + 1) \% \text{maxsize};$
- 队列初始化: $\text{front} = \text{rear} = 0;$
- 队空条件: $\text{front} == \text{rear};$
- 队满条件: $(\text{rear} + 1) \% \text{maxsize} == \text{front};$

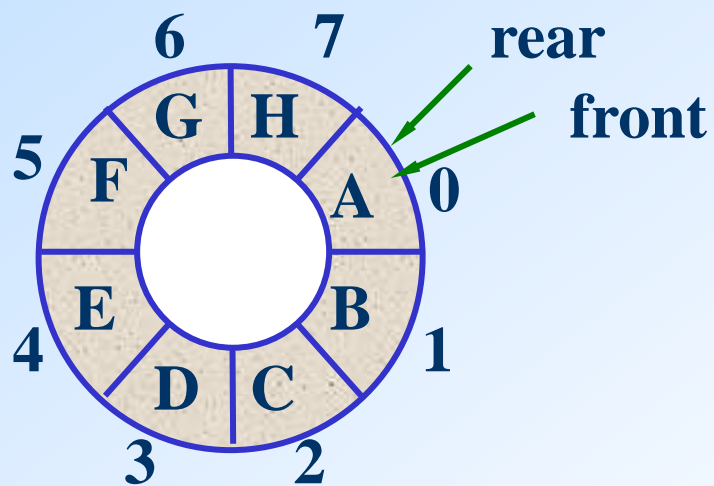




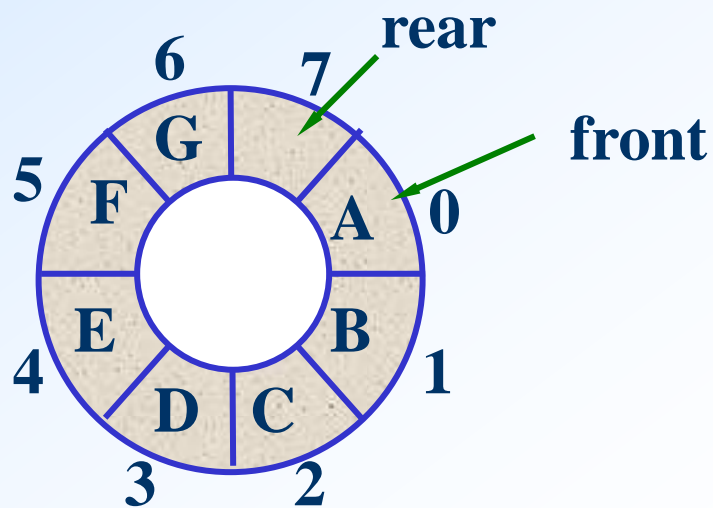
空队列



一般情况



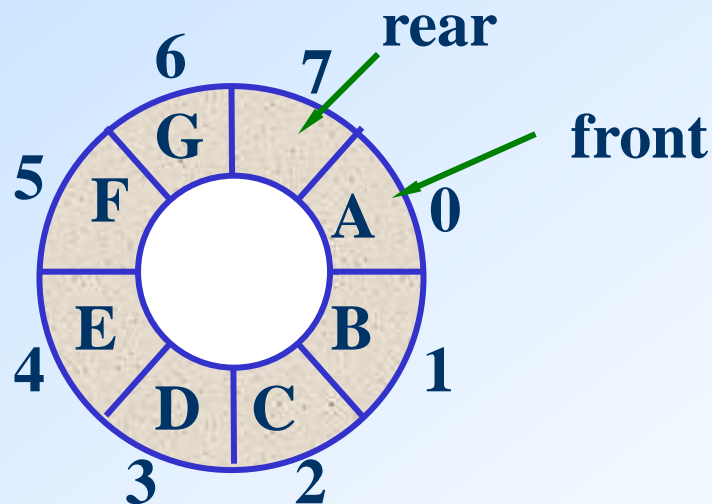
队满



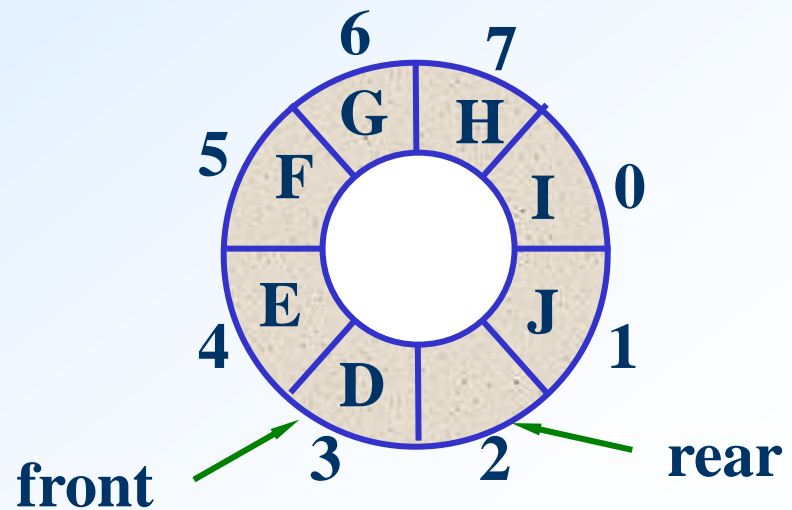
队满(正确)

循环队列 (Circular Queue)

- 队头、队尾指针加1, 可用取模(余数)运算实现。
- 队头指针进1: $\text{front} = (\text{front} + 1) \% \text{maxsize};$
- 队尾指针进1: $\text{rear} = (\text{rear} + 1) \% \text{maxsize};$
- 队列初始化: $\text{front} = \text{rear} = 0;$
- 队空条件: $\text{front} == \text{rear};$
- 队满条件: $(\text{rear} + 1) \% \text{maxsize} == \text{front};$



队满(正确)



循环队列的类型定义

```
#define MAXSIZE 100  
typedef int QueueData
```

```
typedef struct{  
    QueueData *data;  
    int front;  
    int rear;  
} SeqQueue
```

循环队列操作的实现

■ 初始化队列

```
void InitQueue ( SeqQueue *Q ) {
```

```
//构造空队列
```

```
Q->data=(QueueData *)malloc(MAXSIZE  
    *sizeof(QueueData));
```

```
if(! Q->data)exit;
```

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

```
}
```

- **判队空**

```
int QueueEmpty ( SeqQueue *Q ) {  
    return Q->rear == Q->front;  
}
```

- **判队满**

```
int QueueFull ( SeqQueue *Q ) {  
    return (Q->rear+1) % MAXSIZE == Q->front;  
}
```

- **入队**

```
int EnQueue ( SeqQueue *Q, QueueData x ) {  
    if ( QueueFull (Q) ) return 0;  
    Q->data[Q->rear] = x;  
    Q->rear = ( Q->rear+1) % MAXSIZE;  
    return 1;  
}
```


■ 出队

```
int DeQueue ( SeqQueue *Q, QueueData *x ) {  
    if ( QueueEmpty (Q) ) return 0;  
    *x = Q->data[Q->front];  
    Q->front = ( Q->front+1) % MAXSIZE;  
    return 1;  
}
```

■ 取队头

```
int GetFront ( SeqQueue *Q, QueueData *x ) {  
    if ( QueueEmpty (Q) ) return 0;  
    * x = Q->data[(Q->front)];  
    return 1;  
}
```

- 练习：在循环队列中，充分运用第N个位置，则算法如何编写？

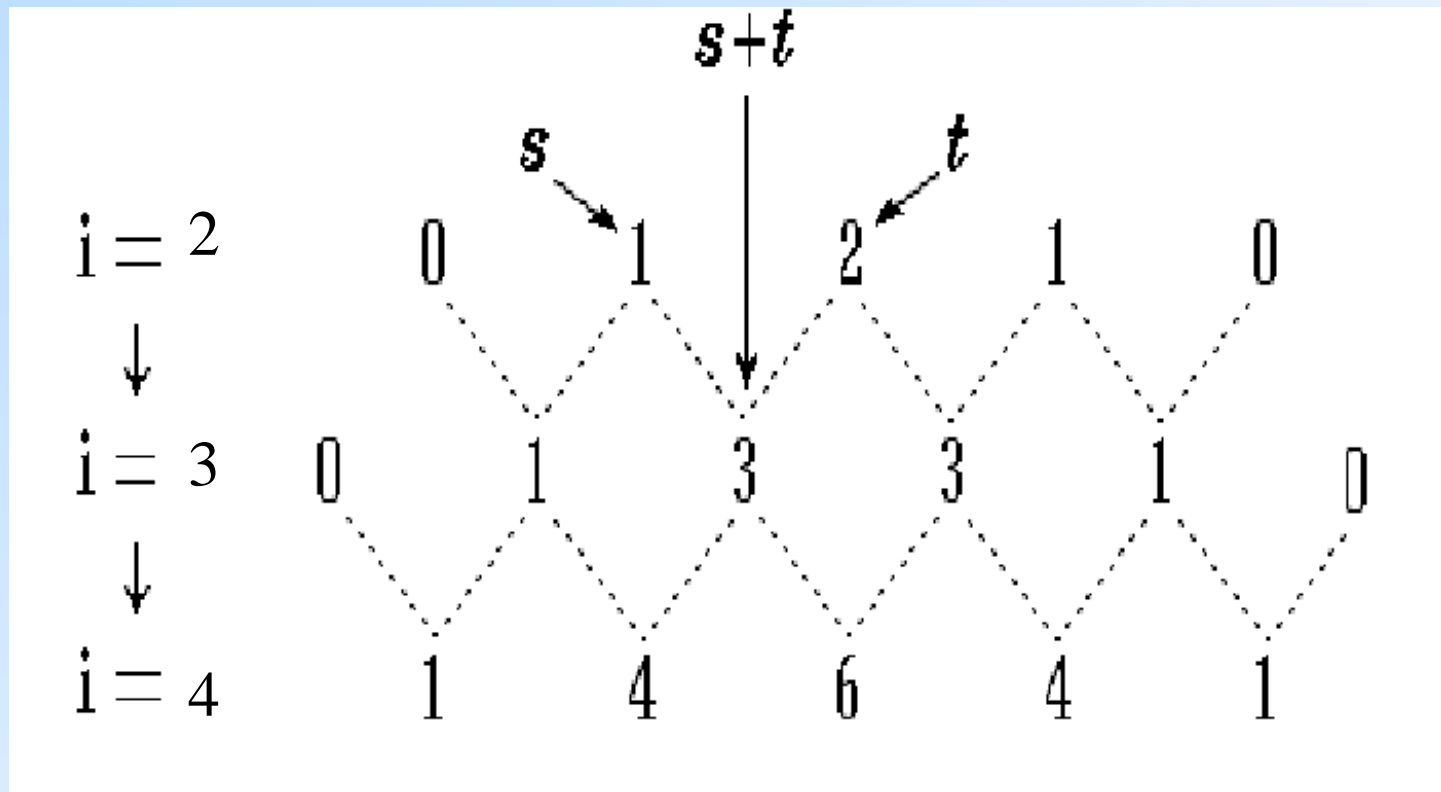
队列应用举例

—打印二项展开式 $(a + b)^i$ 的系数

杨辉三角形 (Pascal's triangle)

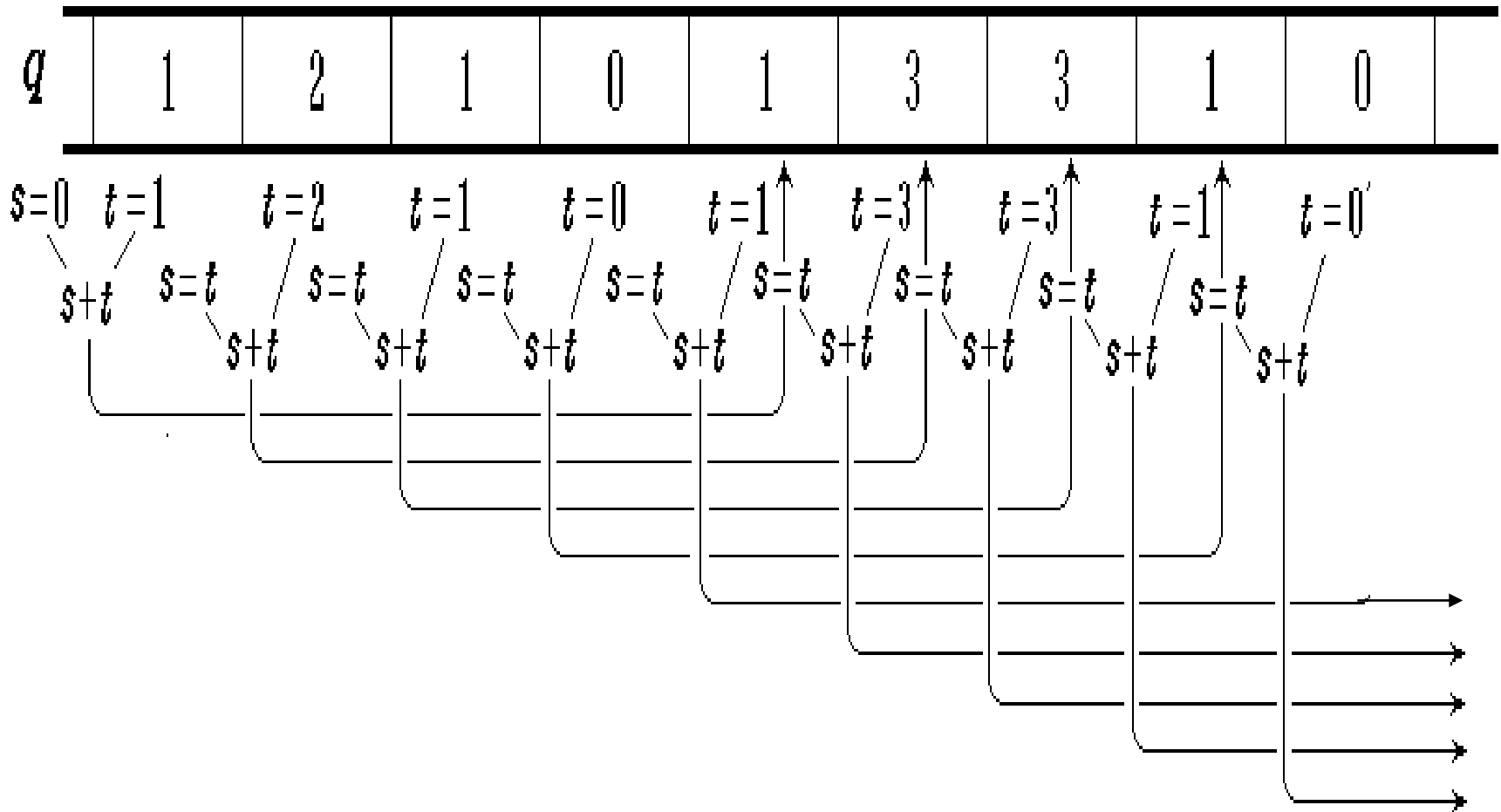
				1		1				$i =$ 1	
				1		2		1		2	
			1		3		3		1	3	
		1		4		6		4		4	
	1		5		10		10		5	5	
1		6		15		20		15		6	6

分析第 i 行元素与第 $i+1$ 行元素的关系



目的是从前一行的数据可以计算下一行的数据

从第 i 行数据计算并存放第 $i+1$ 行数据

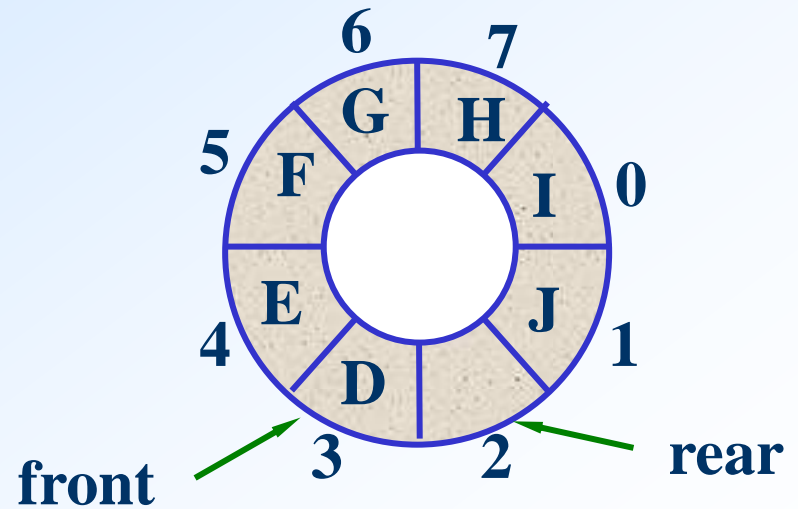
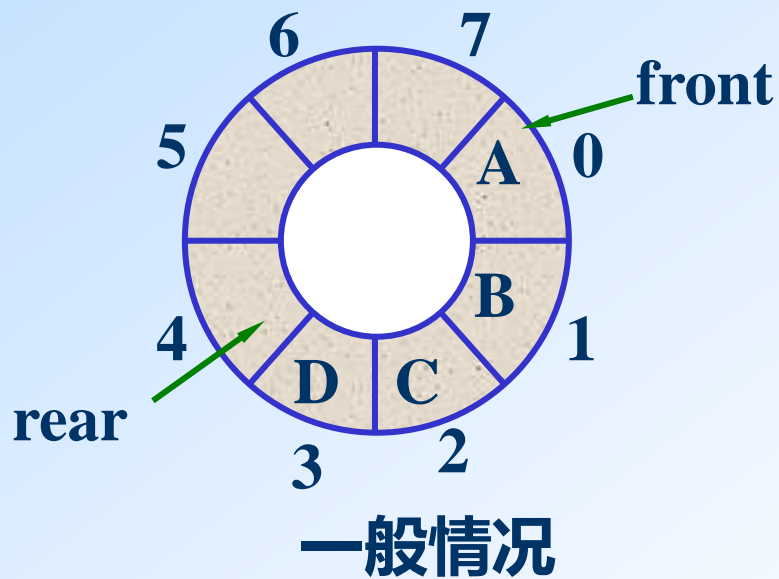


```
void YangHui ( int n )
{int s=0, t,i,j;
  SqQueue *q;
  q=(SqQueue *)malloc(sizeof(SqQueue));
  InitQueue(q);
  EnQueue (q, 1); EnQueue (q, 1); //第1行
  for( i=1; i<=n; i++ ) //逐行计算
  { printf(“\n”);
    EnQueue (q, 0);
    for ( j=1; j<=i+2; j++ ) //根据上行系数求下行系数
    { DeQueue (q,& t);
      EnQueue (q, s+t);
      s = t;
      if ( j != i+2 ) printf(“%3d”, s);          //不输出每行结尾的0
    }
  }
}
```

- 为解决计算机主机与打印机之间的速度不匹配问题，通常设计一个打印数据缓冲区。该缓冲区的逻辑结构应该是？

- 1、设环形队列中数组的下标是0~N-1，其头、尾指针分别是f和r，则其元素个数是？

$$(r-f+N)\%N$$



- 2、某队列允许在两端进行入队操作，但仅允许在一端进行出队操作，若a,b,c,d,e元素进队，则不可能得到的顺序是？

A) bacde B) dbace C) dbcae D) ecbad