数据结构

Ch3 栈/队列

计算机学院（国家示范性软件学院）

第3章 栈、队列

- 3.1 栈（定义与实现：顺序栈、链栈）
- 3.2 栈的应用举例
- 3.3 栈与递归的实现
- 3.4 队列（定义与实现：循环队列、链队）
- 3.5 本章知识点小结

线性表、栈、队列的比较

线性表允许在表内任一位置进行插入和删

队列只允许在表尾一端进行插入，在表头一端进行删除。

计中被广泛使用的两种线性数据结构，它们的特

性，栈必须按“后进先出”的规则进行操作，

出”的规则进行操作。和线性表相比，它们的

的约束和限定，故又称为限定性的线性表结构。

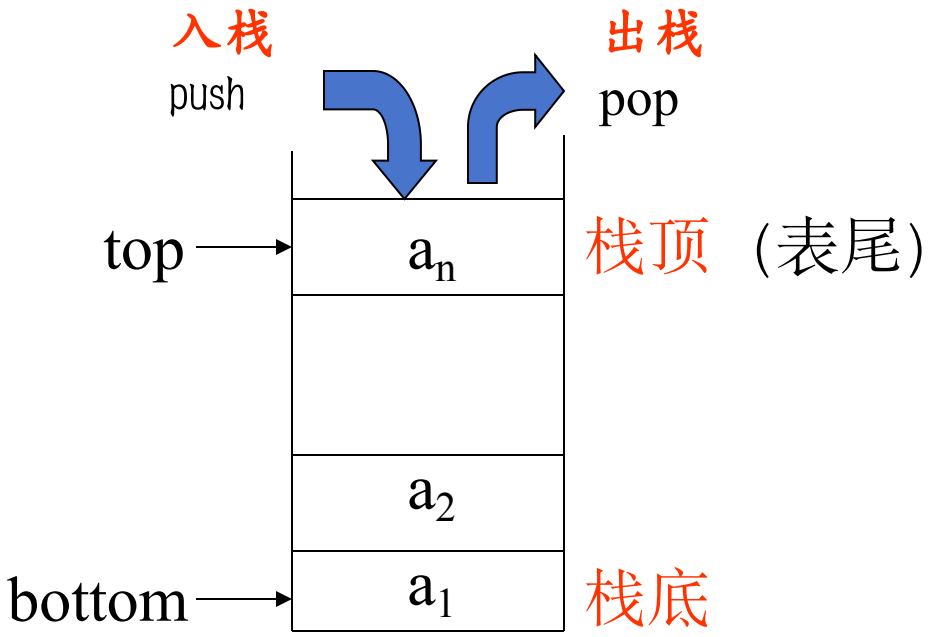
的插入和删除操作对比如下：

	插 入	删 除
线性表：	Insert(L,i,x)	Delete(L,i)
	(1≤i≤n+1)	(1≤i≤n)
栈：	Insert(L,n+1,x)	Delete(L,n)
队列：	Insert(L,n+1,x)	Delete(L,1)

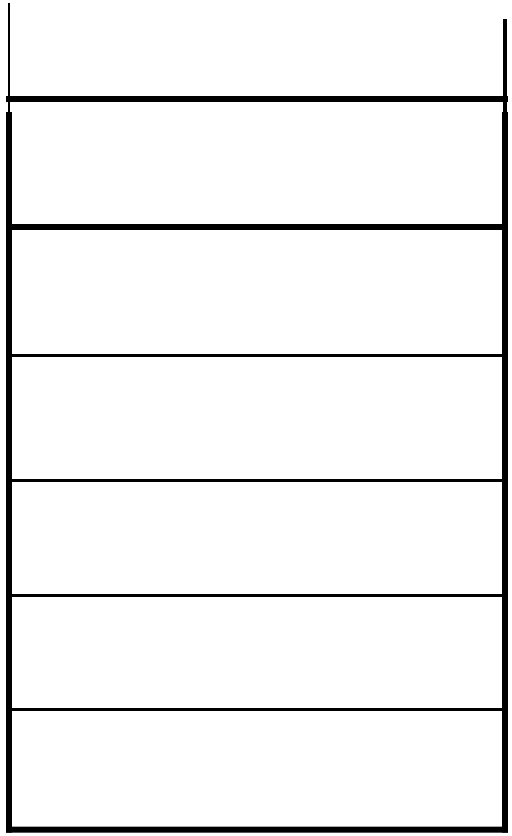
3.1 栈

3.1.1 栈的定义

栈 是一种特殊的线性表，限定插入和删除操作只能在表尾进行。具有后进先出(LIFO—Last In First Out)的特点。

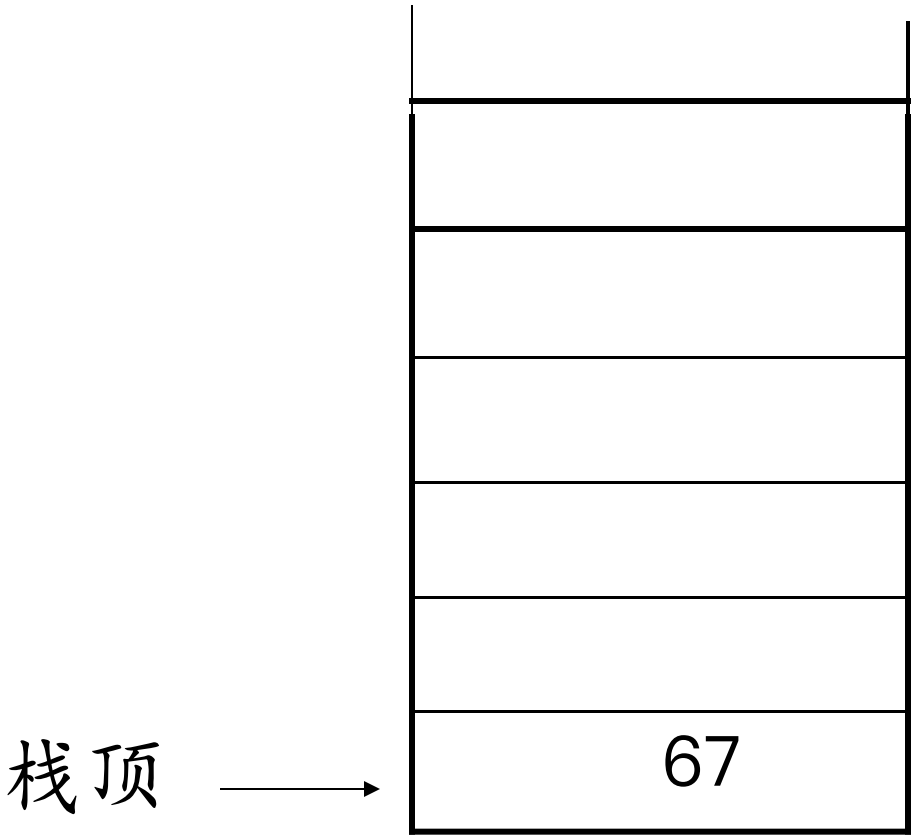


进栈和出栈

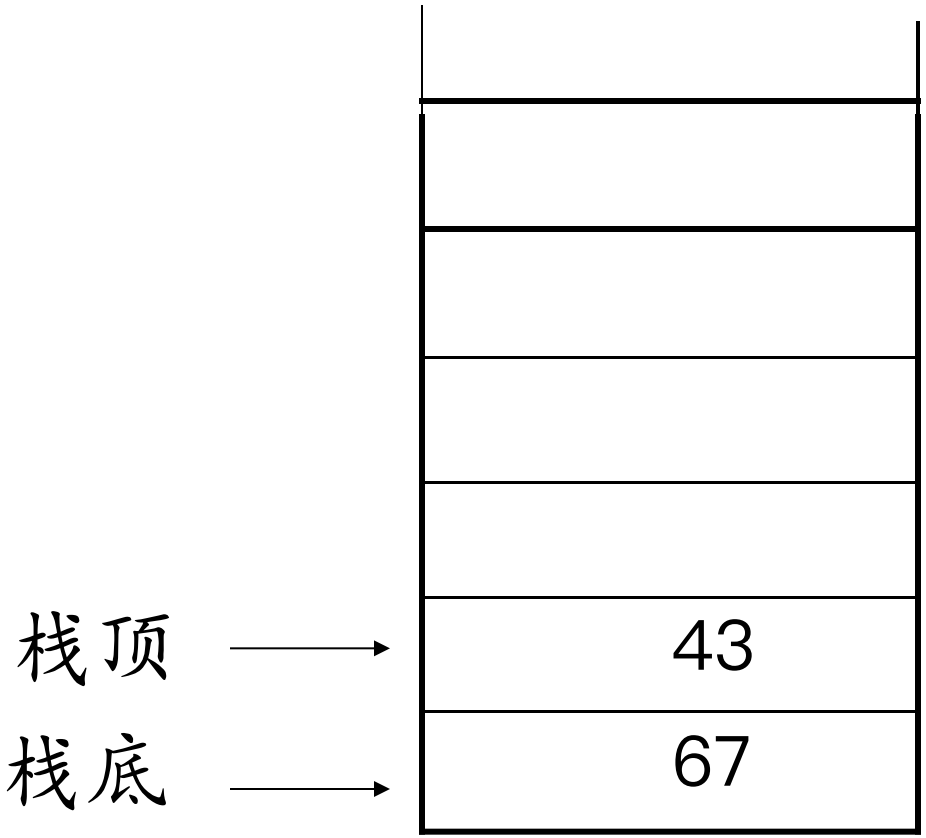


空栈

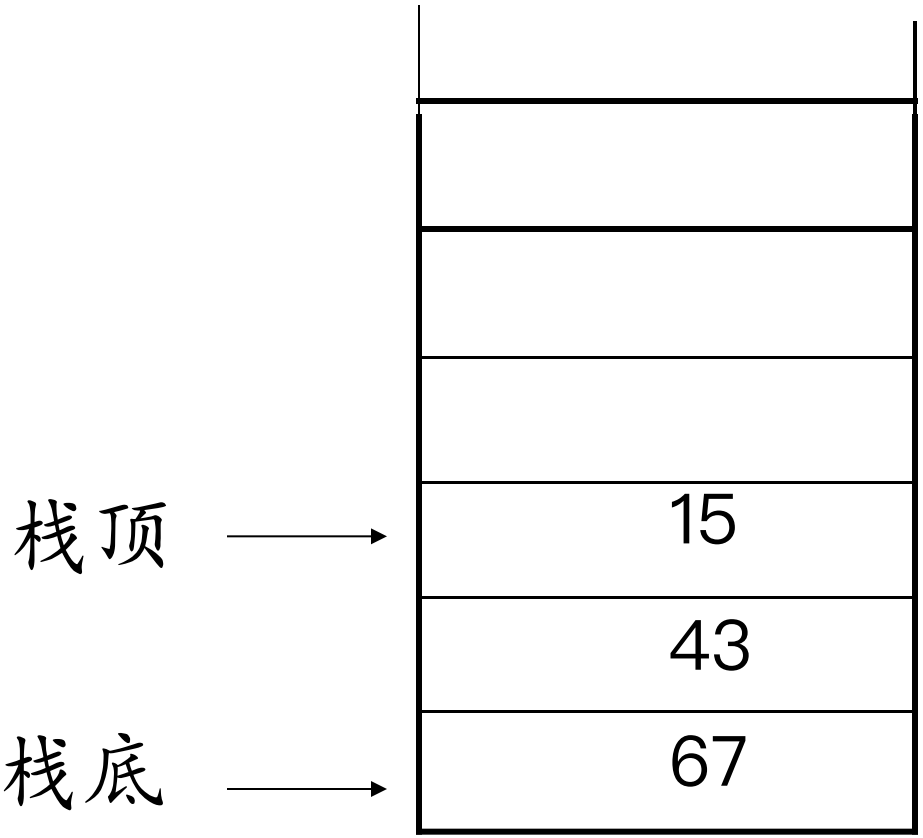
进栈和出栈



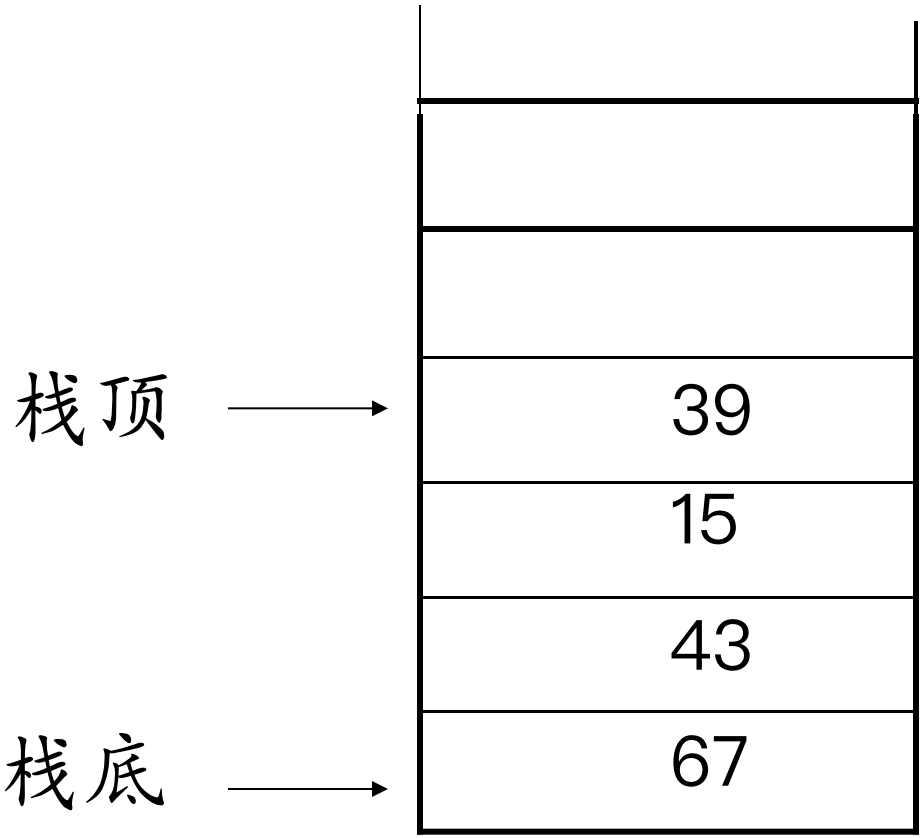
进栈和出栈



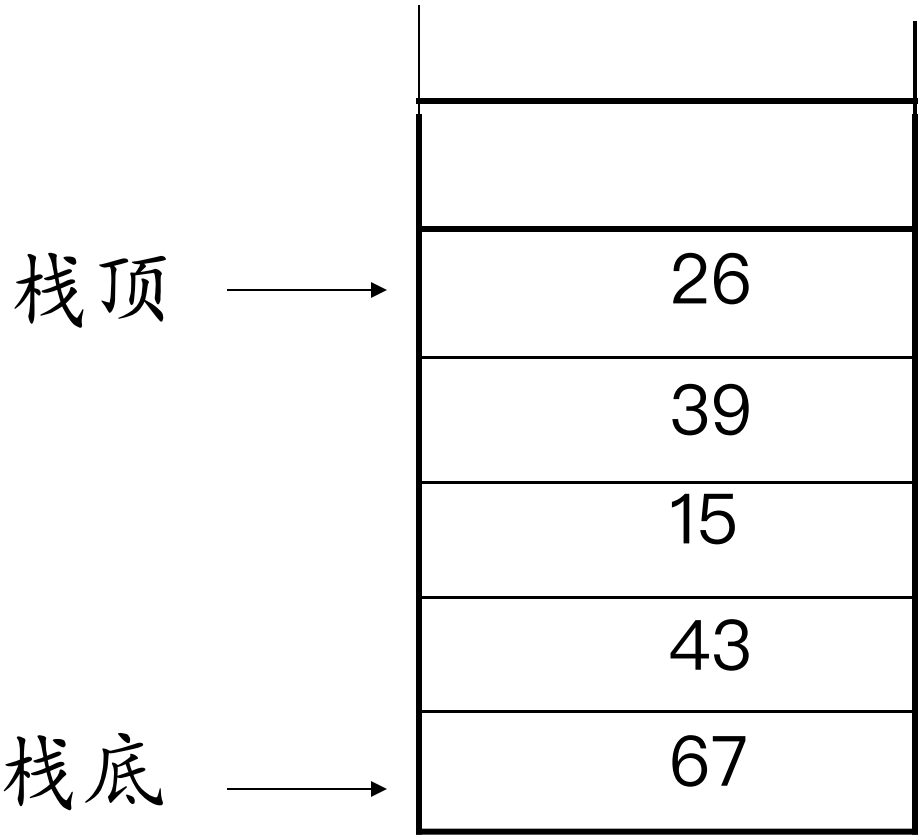
进栈和出栈



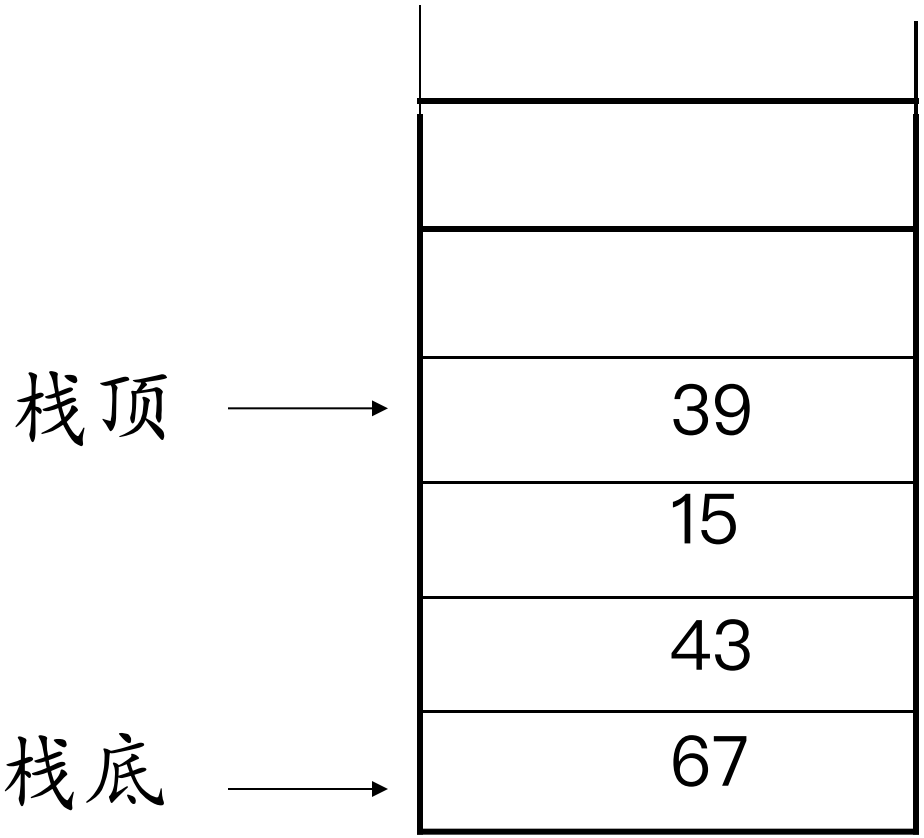
进栈和出栈



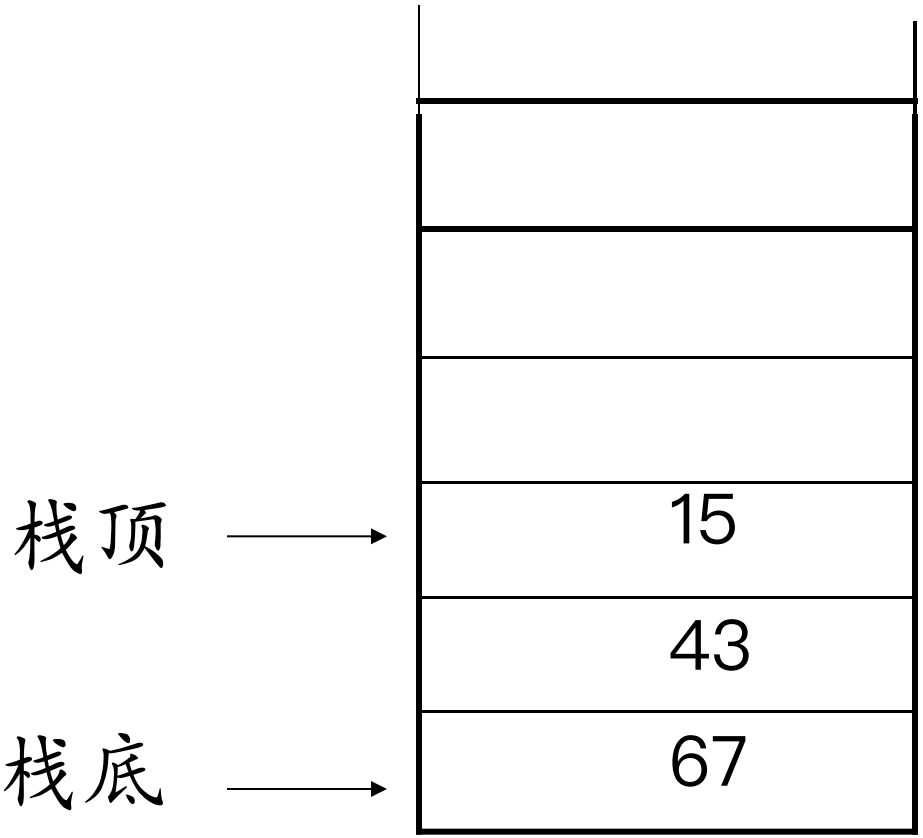
进栈和出栈



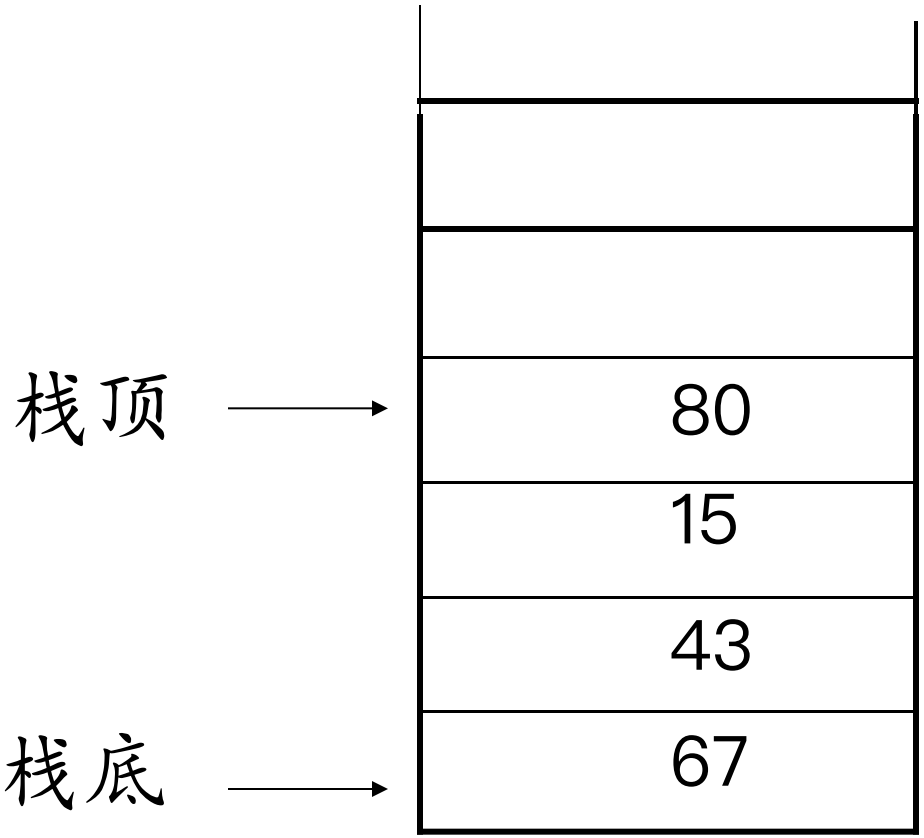
进栈和出栈



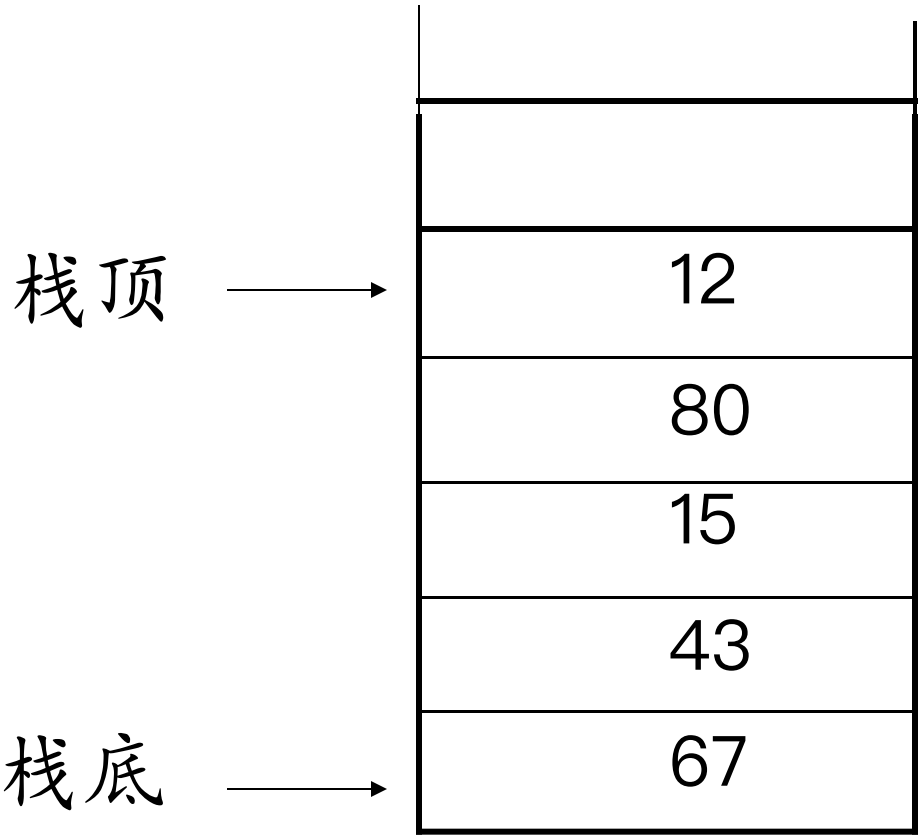
进栈和出栈



进栈和出栈



进栈和出栈



有5个元素，其入栈次序为：A，B，C，D，E，在各种可能的出栈次序中，以元素C，D最先出栈（即C第一个且D第二个出栈）的次序有哪几个？

CDEBA，CDBEA，CDBAE

栈的抽象数据类型定义

- 其ADT定义如下：

ADT Stack {

数据对象：$D=\{\ a_i \mid a_i \in ElemSet,\ i=1,2,...,n,\ n\geqslant 0\ \}$

数据关系：$R1=\{\ <a_{i-1},\ a_i>\mid a_{i-1},\ a_i \in D,\ i=2,...,n\ \}$ 约定a_n端为栈顶，a_1端为栈底。

定义在栈结构上的基本操作：

(1) 生成空栈操作　　InitStack(&S)

(2) 销毁栈　　DestroyStack(&S)

(3) 判栈是否为空函数　StackEmpty (S)

(4) 数据元素入栈操作　Push (&S,e)

(5) 数据元素出栈函数　Pop (&S,&e)

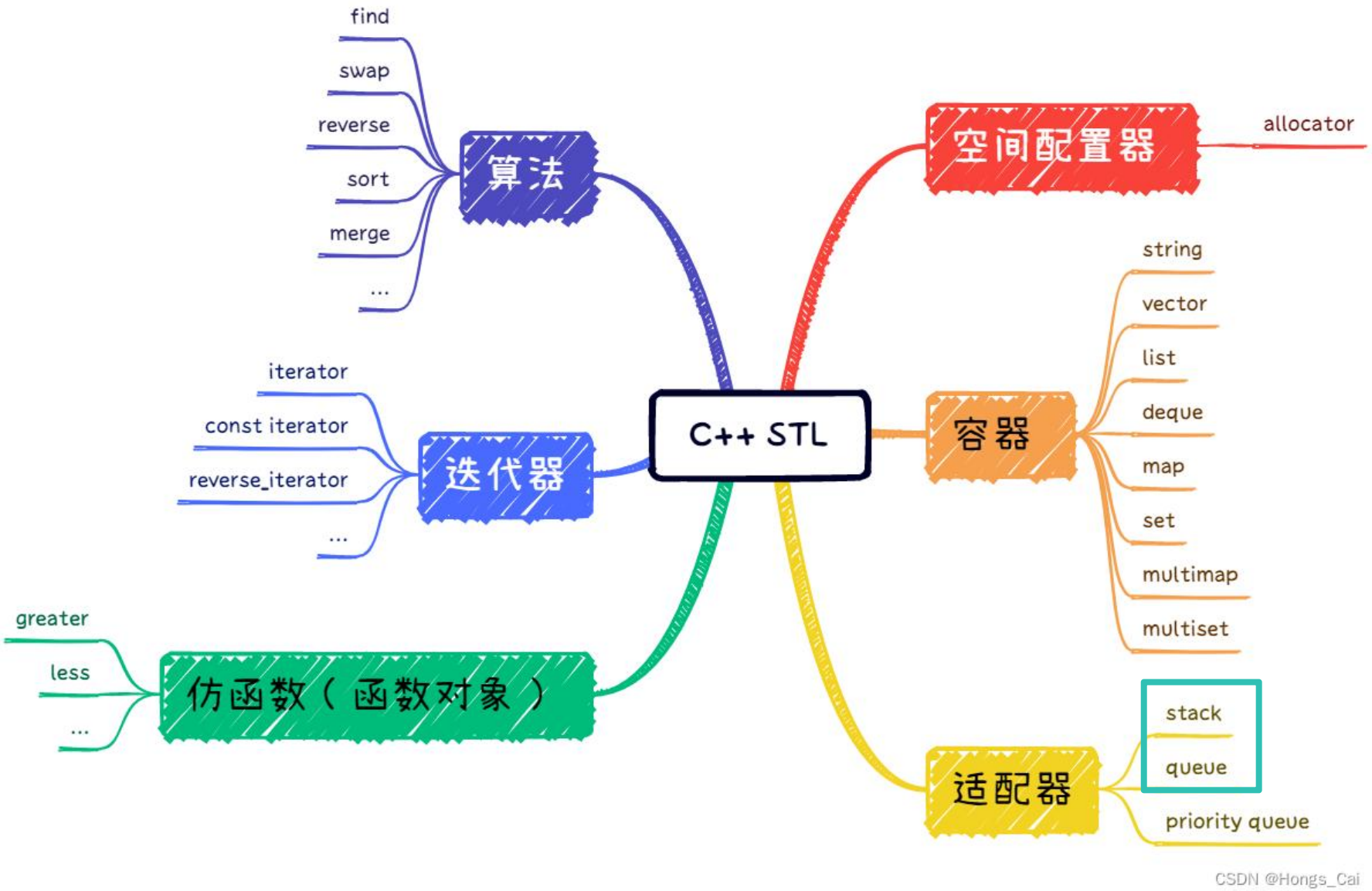
(6) 取栈顶元素函数　　GetTop(S,&e) //读栈顶元素，栈不变化

(7) 置栈空操作　　ClearStack(&S) //清空栈元素

(8) 求当前栈元素个数函数　StackLength(S)

(9) 遍历元素　　StackTraverse(S, visit())

C++ STL 标准模板库 （Standard Template Library）



顺序栈：利用顺序存储方式实现的栈

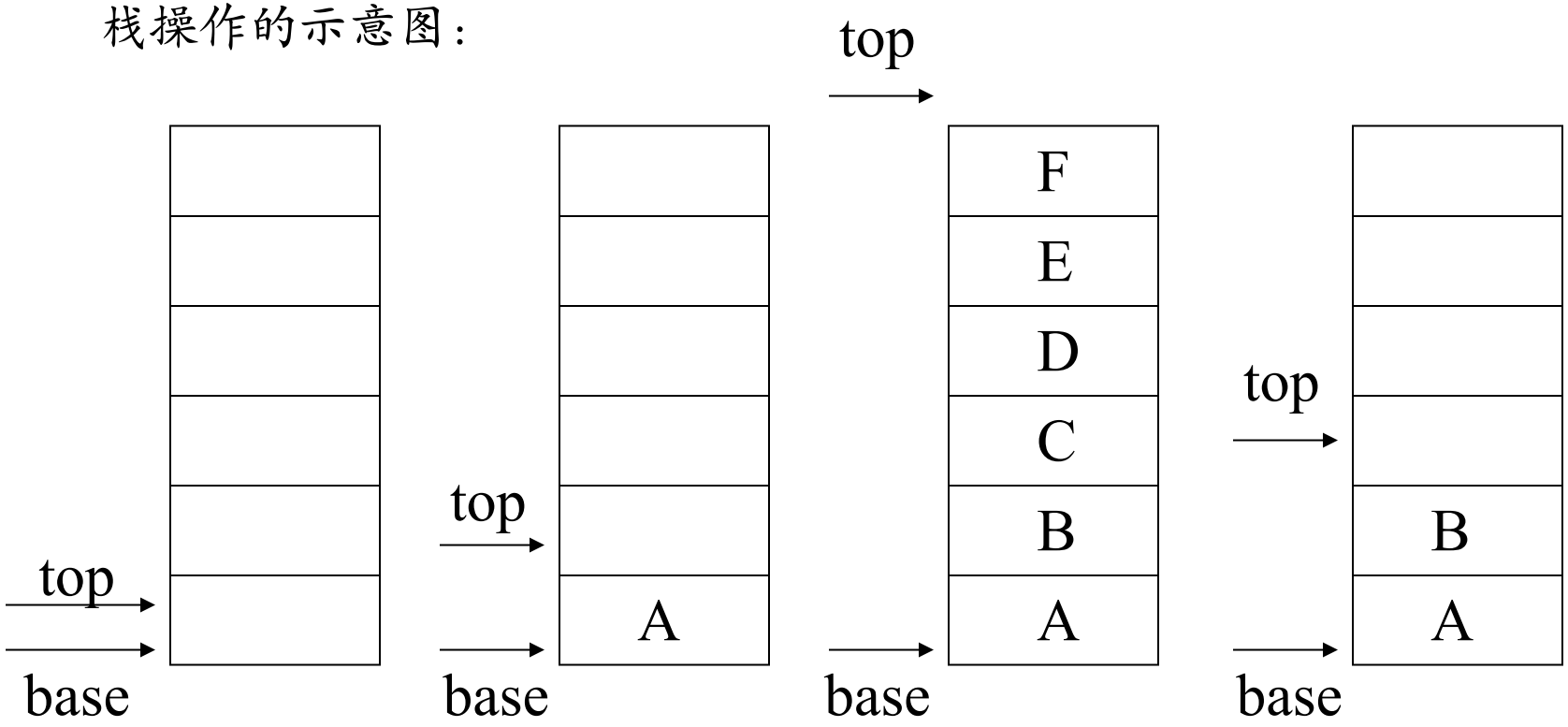
由于栈是运算受限的线性表，因此线性表的存储结构对栈也适用。

和顺序表相似，顺序栈的类型描述如下：

```
typedef  struct{
        SElemType  *base;//栈底指针
        SElemType  *top;//栈顶指针
        int  stacksize;//栈当前可使用的最大容量
}SqStack;
```

思考：为什么用栈顶指针top代替了顺序表中的length呢？

栈操作的示意图：



栈中的元素用一组连续的存储空间来存放的。栈底位置设置在存储空间的一个端点，而栈顶是随着插入和删除而变化的，非空栈中的栈顶指针top来指向栈顶元素的下一个位置。

好处：top-base就是栈中元素的数量，top==base时栈为空。

(1) 生成空栈

首先建立栈空间，然后初始化栈顶指针。

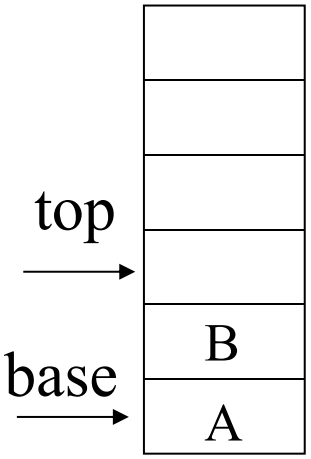
```
Status  InitStack( SqStack  &S)  {
    S.base = (SElemType *) malloc (STACK_INIT_SIEZ* sizeof( SElemType ));
    if(!S.base) exit (OVERFLOW);
    S.top = S.base;
    S.stacksize = STACK_INIT_SIZE;
    return  OK;
}
```

(2)判断是否为空栈

```
Status  StackEmpty ( S )
{
      if (S.top= = S.base)  return  TRUE;
     else  return  FALSE;
}
```

(3)入栈

```
Status  Push ( SqStack &S, Elemtype  e)
{
    if ( S.top-S.base >= S.stacksize )  {//栈满，追加存储空间
        S.base = (SElemType *) realloc(S.base, (S.stacksize +
                                STACKINCREMENT) * sizeof(SElemType));
       if(!S.base) exit(OVERFLOW);      /*存储分配失败*/
       S.top = S.base + S.stacksize;
      S.stacksize += STACKINCRMENT;
     }
     *S.top++ = e;
      return OK;
}//Push
```

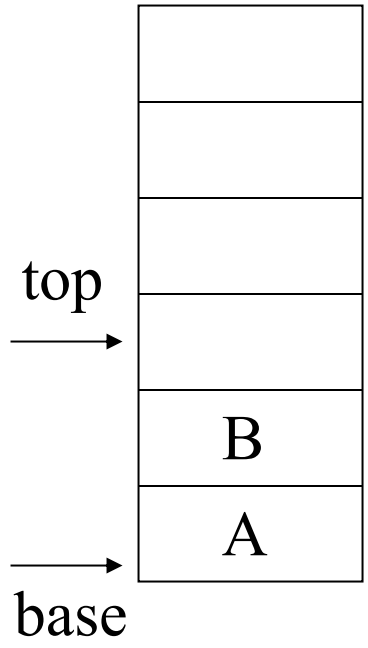


为什么需要这一句?

++在后面，先 *S.top =e ， 然后S.top++

(4)出栈

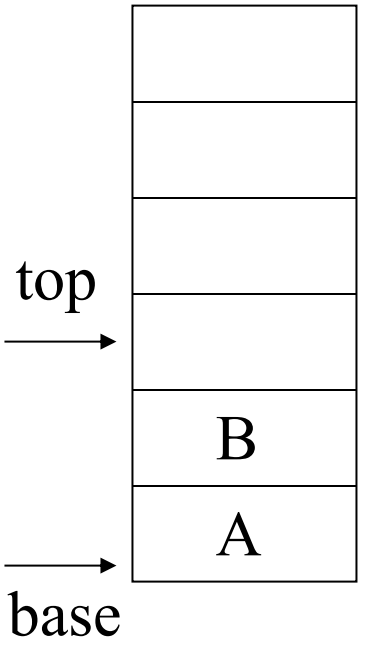
```
Status   Pop ( SqStack &S, Elemtype &e )
 {
      if( S.top = = S.base )
            return ERROR;
      e = *--(S.top);
       return  OK;
 }//Pop
```



--在前面，先 S.top 减1，然后e =*S.top

(5)取栈顶元素

```
Status  GetTop ( SqStack S, Elemtype &e )
{
    if( S.top = = S.base )    return ERROR;
     e = *(S.top-1);      return  OK;
}
```



注意是-1，不是--

说明：

1）对于顺序栈，入栈时，首先判栈是否满了，栈满的条件为：

S.top-S.base >= S.stacksize

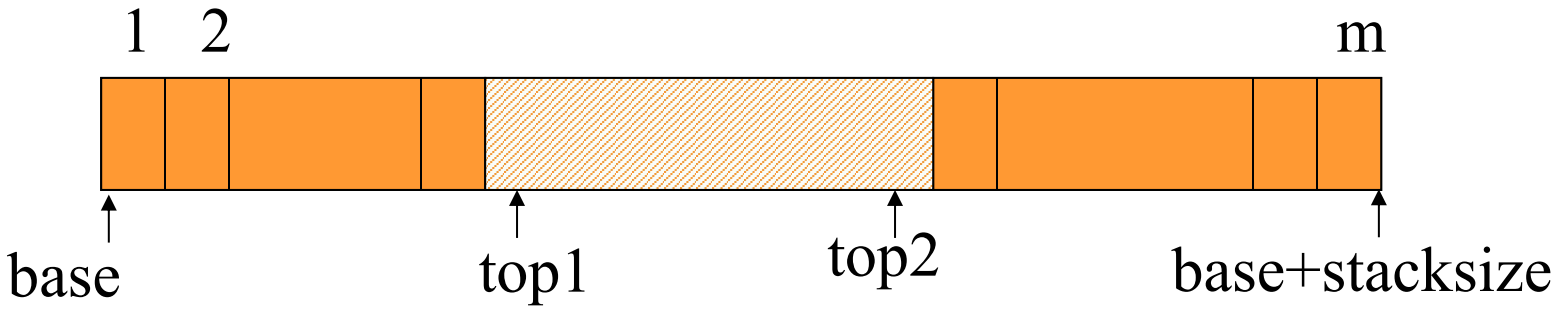
栈满时，不能入栈，否则出现空间溢出，这种现象称为上溢

2）出栈和读栈顶元素操作，先判栈是否为空，为空时不能出栈和读栈顶元素，否则产生错误。

两个栈共享一组地址连续的存储单元

【类型定义】 数组(栈空间) + 两个栈顶指示

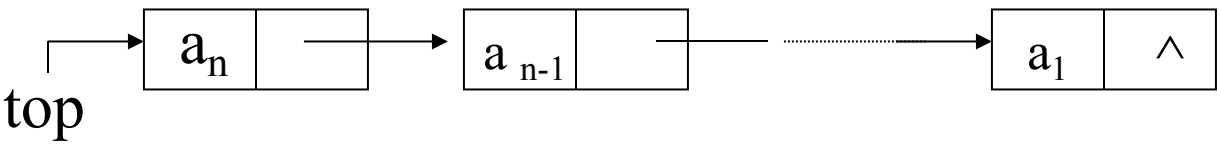
```
CONST   m=500{两栈的总允许容量};
typedef  struct  {
  Elemtype    * top1;
  Elemtype    * top2;
  Elemtype    * base;
  int    stacksize;
 }  SqStack;
```



2. 链栈

[类型定义]

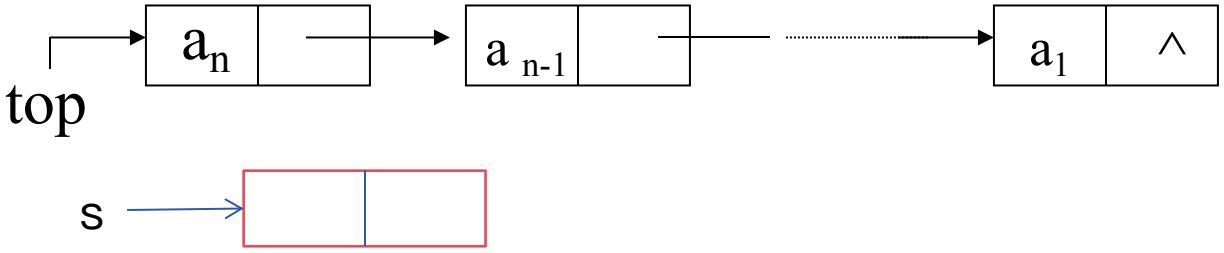
```
typedef  struct StackNode  {
    SElemtype     data;
    struct StackNode  *next;
}StackNode, *LinkStack;

LinkStack  top;
```



栈顶指针(链首指针) 指向表尾元素a_n， 和单链表相反

- 链栈不需要事先分配空间；
- 在进行入栈操作时不需要顾忌栈的空间是否已经被填满。

- 链栈的结点结构和单链表中的结点结构相同，**由于栈只在栈顶作插入和删除操作，因此链栈中不需要头结点，但要特别注意链栈中指针的方向是从栈顶指向栈底的，这正好和单链表是相反的。**



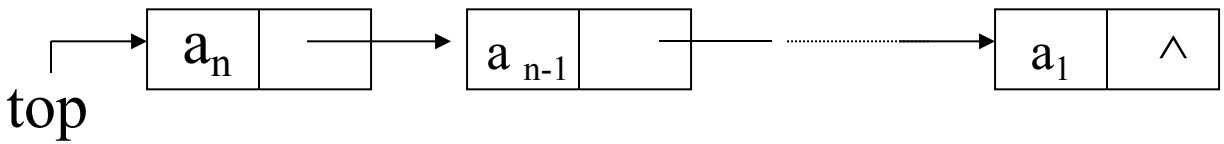
(1) 入栈

```
Status  Push_LinkStack  (LinkStack  &top, SElemtype e)
{
      s = (StackNode *)malloc(sizeof(StackNode));
      s->data = e;
      s->next = top;
      top = s;
      return OK;
   }
```

核心思路：创建一个结点，把结点插入到链栈的第一个位置

(2) 出栈

```
Status   Pop_LinkStack (LinkStack  &top, Elemtype  &e)
{
    if   (top = = NULL)   return ERROR;
    e = top->data;
    p = top;
    top = top->next;
    free (p);
    return  OK;
}
```



3.2 栈的应用举例

- 将十进制数N转换为r进制的数，其转换方法利用辗转相除法：以N=3467，r=8为例转换方法如下：

N	N/8（整除）	N%8（求余）	
3467	433	3	低
433	54	1	
54	6	6	
6	0	6	高

- 所以：$(3467)_{10} = (6613)_{8}$
- 将得到的余数依次放入栈中。

算法基本思想：

设栈s，当N>0时重复1），2）

1）若N≠0，则将N%r压入栈s中，执行2）；若N=0，将栈s的内容依次出栈，算法结束。

2）用N/r代替N

3.2.1 数制转换

栈的引入简化了程序设计问题，区分了关注层次，缩小了思考范围。

```
void conversion ()
{
// 对于输入的任意非负十进制整数，输出与其等值的八进制数
   InitStack(S);                  // 构造空栈
   scanf ( “%d” ,N);              // 输入一个十进制数
   while(N)
   {
      Push(S,N % 8);              // "余数"入栈
      N = N/8;                    // 非零"商"继续运算
   } // while
   while (!StackEmpty(S))
   {     // 和"求余"所得相逆的顺序输出八进制的各位数
      Pop(S,e);
      printf ( “ %d” ,e);
   } // while
} // conversion
```

算法3.1

3.2.2 括弧匹配检验

- 假设表达式中允许包含两种括号：圆括号和方括号，其嵌套的顺序随意，如（［］［］）或［（［］［］）］等为正确的匹配，［（］）或（［］（））或((()))均为错误的匹配。
- 要求检验一个给定表达式中的括弧是否正确匹配?
- 例如考虑下列括号序列：

  ［（［］［］）］

  1 2 3 4 5 6 7 8

3.2.2 括弧匹配检验

算法核心思路：

检查表达式中的字符，遇到左括号入栈，遇到右括号则出栈栈顶元素与其匹配，如果匹配成功则继续，否则退出

那么，什么样的情况是“不匹配”的情况呢？

1. 和栈顶的左括弧不相匹配；
2. 栈中并没有左括弧等在哪里；
3. 栈中还有左括弧没有等到和它相匹配的右括弧。

在以上分析的基础上就可以写出检验括弧匹配的算法了。

3.2.2 括弧匹配检验

算法注意事项:

- 1. “匹配”不是“相等”。 ()
- 2. 和栈顶元素进行比较的前提是栈不为空。(右括号还有，左括号没了)
- 3. “没有等到”即为栈不空的情况。因此在算法结束之前，需要判别栈是否已为空。 (左括号还有，右括号没了)
- 4. 别忘了使用栈之前一定要进行初始化。

3.2.5 表达式求值

表达式求值是程序设计语言编译中的一个基本问题，它的实现也用到栈。

表达式由操作数（operand）、运算符（operator）、界限符（delimiter）组成的有意义的式子。运算符从运算对象的个数上分，有单目运算符和双目运算符。在此仅讨论只含双目运算符的算术表达式。

例如： 3*2^（4+2*2-1*3）-5

每个双目运算符在两个运算量的中间的叫中缀表达式

设定运算符包括：+、-、*、/、%、^和（）

设定运算规则为：

1） 优先级（）→ ^ → ×、/、% → +、-；

2）有括号出现时先算括号内的，后算括号外的，多层括号，由内向外进行；

3） 乘方连续出现时先算最右面的。

3.2.5 表达式求值

(1)中缀表达式求值

如表达式“3*2^（4+2*2-1*3）-5”

正确的处理过程是：需要两个栈，操作数与运算结果栈s1和运算符栈s2.

自左向右扫描表达式的每一个字符，

- 若当前字符是操作数，则入栈s1，
- 如果是运算符时：
  - 若这个运算符比s2栈顶运算符高，则入栈s2，继续向后处理，
  - 若这个运算符比s2栈顶运算符低，则从栈s1出栈两个运算对象，从运算符栈s2出栈一个运算符进行运算，并将其结果入栈s1
- 继续处理当前字符，如处理完则处理下一字符，直到遇到结束符。

中缀表达式表达式 “3*2^（4+2*2-1*3）-5”求值过程中两个栈的状态情况如图所示。

为了使第一个运算符入栈，预设一个最低级运算符（

读字符	对象栈 s1	算符栈 s2	说明
3	3	(	3 入栈 s1
*	3	(*	*入栈 s2
2	3，2	(*	2 入栈 s1
^	3，2	(*^	^入栈 s2
(	3，2	(*^(	(入栈 s2
4	3，2，4	(*^(	4 入栈 s1
+	3，2，4	(*^(+	+入栈 s2
2	3，2，4，2	(*^(+	2 入栈 s1
*	3，2，4，2	(*^(+*	*入栈 s2
2	3，2，4，2，2	(*^(+*	2 入栈 s1
-	3，2，4，4	(*^(+	做 2*2=4，结果入栈 s1
	3，2，8	(*^(	做 4+4=8，结果入栈 s2
	3，2，8	(*^(-	-入栈 s2
1	3，2，8，1	(*^(-	1 入栈 s1
*	3，2，8，1	(*^(-*	*入栈 s2
3	3，2，8，1，3	(*^(-*	3 入栈 s1
)	3，2，8，3	(*^(-	做 1*3，结果 3 入栈 s1
	3，2，5	(*^(	做 8-3，结果 5 入栈 s2
	3，2，5	(*^	(出栈
-	3，32	(*	做 2^5，结果 32 入栈 s1
	96	(	做 3*32，结果 96 入栈 s1
	96	(-	-入栈 s2
5	96，5	(-	5 入栈 s1
结束符	91	(	做 96-5，结果 91 入栈 s1

中缀表达式表达式 “3*2^（4+2*2−1*3）−5”求值过程中两个栈的状态情况如图所示。

有些操作符在栈内外的优先级是不同的，

左括号在栈外时优先级最高，在栈内时优先级很低，仅高于栈外的右括号。

读字符	对象栈 s1	算符栈 s2	说明
3	3	(	3 入栈 s1
*	3	(*	* 入栈 s2
2	3，2	(*	2 入栈 s1
^	3，2	(*^	^ 入栈 s2
(	3，2	(*^(	(入栈 s2
4	3，2，4	(*^(	4 入栈 s1
+	3，2，4	(*^(+	+ 入栈 s2
2	3，2，4，2	(*^(+	2 入栈 s1
*	3，2，4，2	(*^(+*	* 入栈 s2
2	3，2，4，2，2	(*^(+*	2 入栈 s1
-	3，2，4，4	(*^(+	做 2*2=4，结果入栈 s1
	3，2，8	(*^(	做 4+4=8，结果入栈 s2
	3，2，8	(*^(-	- 入栈 s2
1	3，2，8，1	(*^(-	1 入栈 s1
*	3，2，8，1	(*^(-*	* 入栈 s2
3	3，2，8，1，3	(*^(-*	3 入栈 s1
)	3，2，8，3	(*^(-	做 1*3，结果 3 入栈 s1
	3，2，5	(*^(	做 8-3，结果 5 入栈 s2
	3，2，5	(*^	(出栈
-	3，32	(*	做 2^5，结果 32 入栈 s1
	96	(	做 3*32，结果 96 入栈 s1
	96	(-	- 入栈 s2
5	96，5	(-	5 入栈 s1
结束符	91	(	做 96-5，结果 91 入栈 s1

3.2.5 表达式求值

(2)后缀表达式

后缀表达式是运算符在运算对象之后，在后缀表达式中，不在引入括号，所有的计算按运算符出现的顺序，严格从左到右进行，而不用再考虑运算规则和级别。

中缀表达式表达式 “3*2^（4+2*2-1*3）-5”的后缀表达式为：

“3 2 4 2 2 * + 1 3 * - ^ * 5 -”

后缀表达式“32422*+13*—^*5—”，栈中状态变化情况：

当前字符	栈中数据	说明
3	3	3 入栈
2	3，2	2 入栈
4	3，2，4	4 入栈
2	3，2，4，2	2 入栈
2	3，2，4，2，2	2 入栈
*	3，2，4，4	计算 2 * 2，将结果 4 入栈
+	3，2，8	计算 4 + 4，将结果 8 入栈
1	3，2，8，1	1 入栈
3	3，2，8，1，3	3 入栈
*	3，2，8，3	计算 1 * 8，将结果 4 入栈
—	3，2，5	计算 8 — 5，将结果 5 入栈
^	3，32	计算 2^5，将结果 32 入栈
*	96	计算 3 *32，将结果 96 入栈
5	96，5	5 入栈
-	91	计算 96-5，结果入栈
结束符	空	结果出栈

3.2.5 表达式求值

(2) 后缀表达式

中缀表达式与后缀表达式的转换（快速手写）：

1、按照运算符的优先级对所有的运算单位加括号

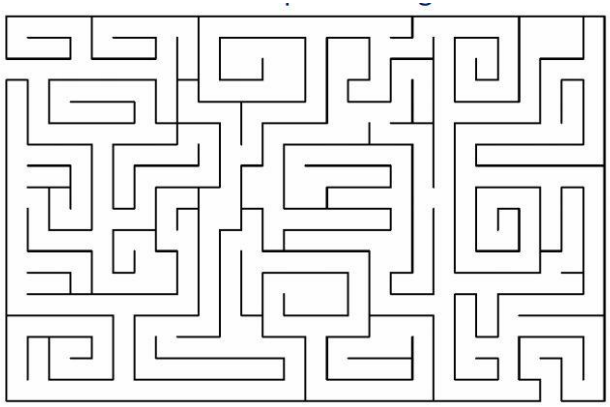
例如：a+b*c-d -→((a+(b*c))-d)

2、把运算符移动到括号的后面，然后去除括号，得后缀表达式

((a+(b*c))-d) → ((a(bc)*)+d)- → abc*+d-

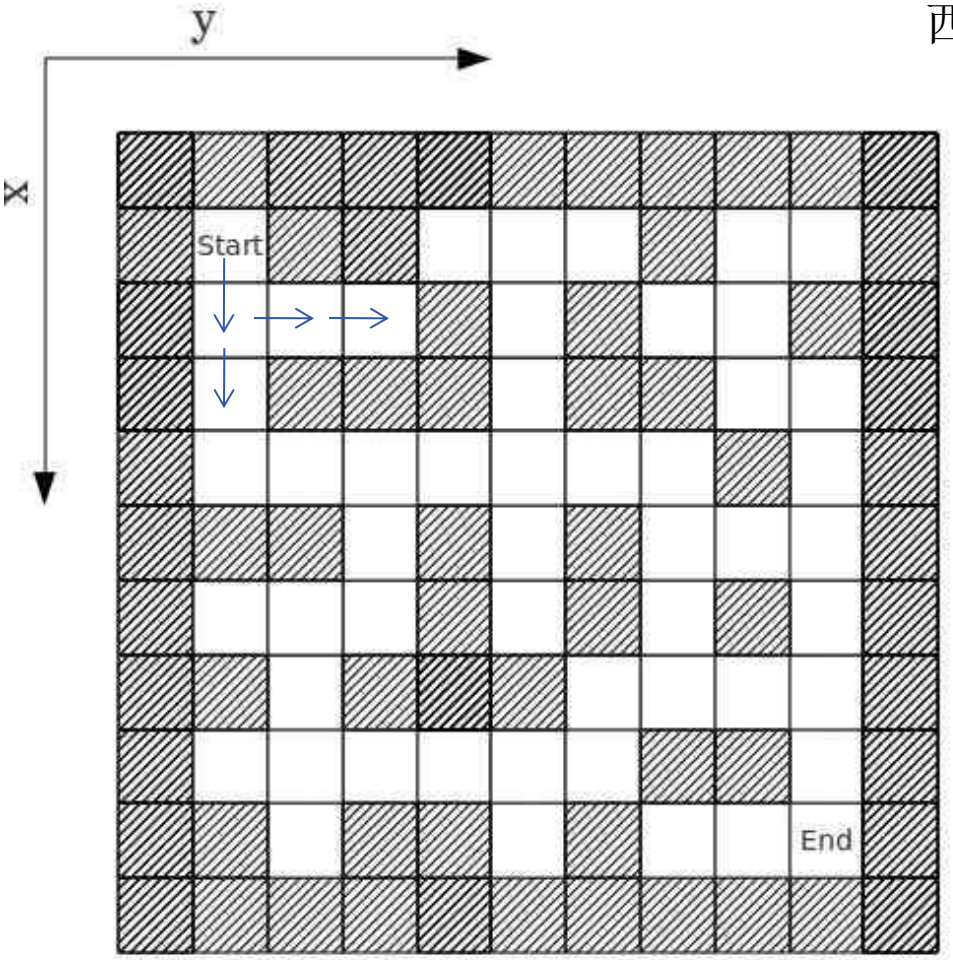
3.2.4 迷宫求解

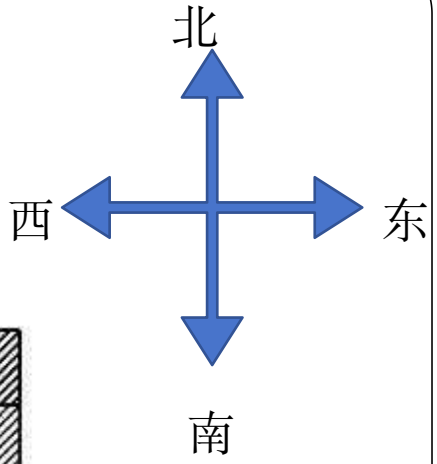
- 计算机解迷宫时，通常用的是“**穷举求解**”的方法，即从入口出发，顺某一方向向前探索，若能走通，则继续往前走；否则沿原路退回，换一个方向再继续探索，直至所有可能的通路都探索到为止,如果所有可能的通路都试探过，还是不能走到终点，那就说明该迷宫不存在从起点到终点的通道。



3.2.4 迷宫求解

为了实现在探索过程中能沿原路退回，需要用一个“后进先出”的结构即“栈”来保存从入口到当前位置的路径。





3.2.4 迷宫求解

- 从入口进入迷宫之后，不管在迷宫的哪一个位置上，都是先往东走，如果走得通就继续往东走，如果在某个位置上往东走不通的话，就依次试探往南、往西和往北方向，从一个走得通的方向继续往前直到出口为止；
- 如果在某个位置上四个方向都走不通的话，就退回到前一个位置，换一个方向再试，如果这个位置已经没有方向可试了就再退一步，如果所有已经走过的位置的四个方向都试探过了，一直退到起始点都没有走通，那就说明这个迷宫根本不通；
- 所谓“走不通”不单是指遇到“墙挡路”，还有“已经走过的路不能重复走第二次”，它包括“曾经走过而没有走通的路”。

显然为了保证在任何位置上都能沿原路退回，**需要用一个“后进先出”的结构即栈来保存从入口到当前位置的路径。**并且在走出出口之后，栈中保存的正是一条从入口到出口的路径。

3.3 栈与递归的实现

3.3 栈与递归的实现

递归是一个过程或函数直接或间接调用自身的一种方法，它可以把一个大型的问题层层转化为一个与原问题相似、但规模较小的问题来求解。

数学中阶乘的定义，n的阶乘可以如下表示：

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

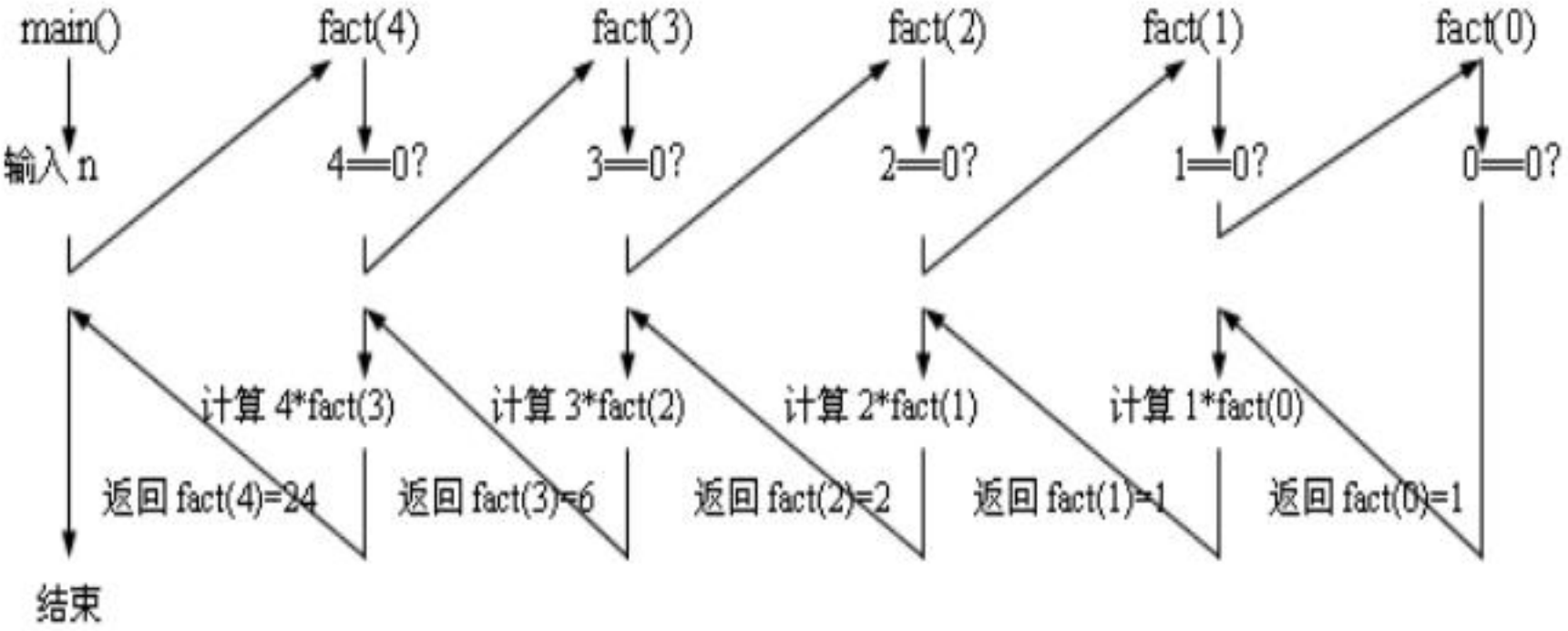
再如，斐波那契（Fibonacci）数列指的是这样一个数列：

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

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

long fibonacci(int n) {
    if (n == 0 || n == 1) {
        return n;
    }
     return fibonacci(n - 2) + fibonacci(n - 1);
}
```

**此算法复杂度为指数级，不能实用

递归求阶乘的问题，假设程序运行时，n＝4，那么程序的执行过程如下：



递归应用举例－汉诺塔问题

汉诺塔问题描述

有a、b、c三个底座，底座上面可以放盘子。初始时a座上有n个盘子，这些盘子大小各不相同，大盘子在下，小盘子在上，依次排列。要求将a座上n个盘子移至c座上，每次只能移动一个，并要求移动过程中保持小盘子在上，大盘子在下，可借助b座实现移动。编程序输出移动步骤

a　　b　　c

递归应用举例－汉诺塔问题

这个问题可用递归思想来分析，将n个盘子由a座移动到c座可分为如下三个过程：

- 先将a座上n-1个盘子借助c座移至b座；
- 再将a座上最下面一个盘子移至c座；
- 最后将b上n-1个盘子借助a移至c座。

上述过程是把移动n个盘子的问题转化为移动n-1个盘子的问题，按这种思路，再将移动n-1个盘子的问题转化为移动n-2个盘子的问题，……

可以用两个函数来描述上述移动过程：

- 从一个底座上借助某一个底座移动n个盘子到另一底座。
- 从一个底座上移动1个盘子到另一底座。

递归应用举例－汉诺塔问题

```
void move(int num, char frompeg, char topeg){
    printf("Move Disk %d from %c to peg %c\n", num, frompeg, topeg);
}
void Hanoi(int num, char startpeg, char finalpeg, char auxpeg){
    if(num == 1){
        move(1, startpeg, finalpeg)
        return;
    }
    Hanoi(num-1, startpeg, auxpeg, finalpeg);
    move(num, startpeg, finalpeg);
    Hanoi(num-1, auxpeg, finalpeg, startpeg);
}
int main(int argc, char *argv[]){
    int num;
    printf("Enter The Number of Disks on Peg A:");
    scanf("%d", &num);
    Hanoi(num, 'A', 'C', 'B'); //A起点，C终点，B辅助
}
```

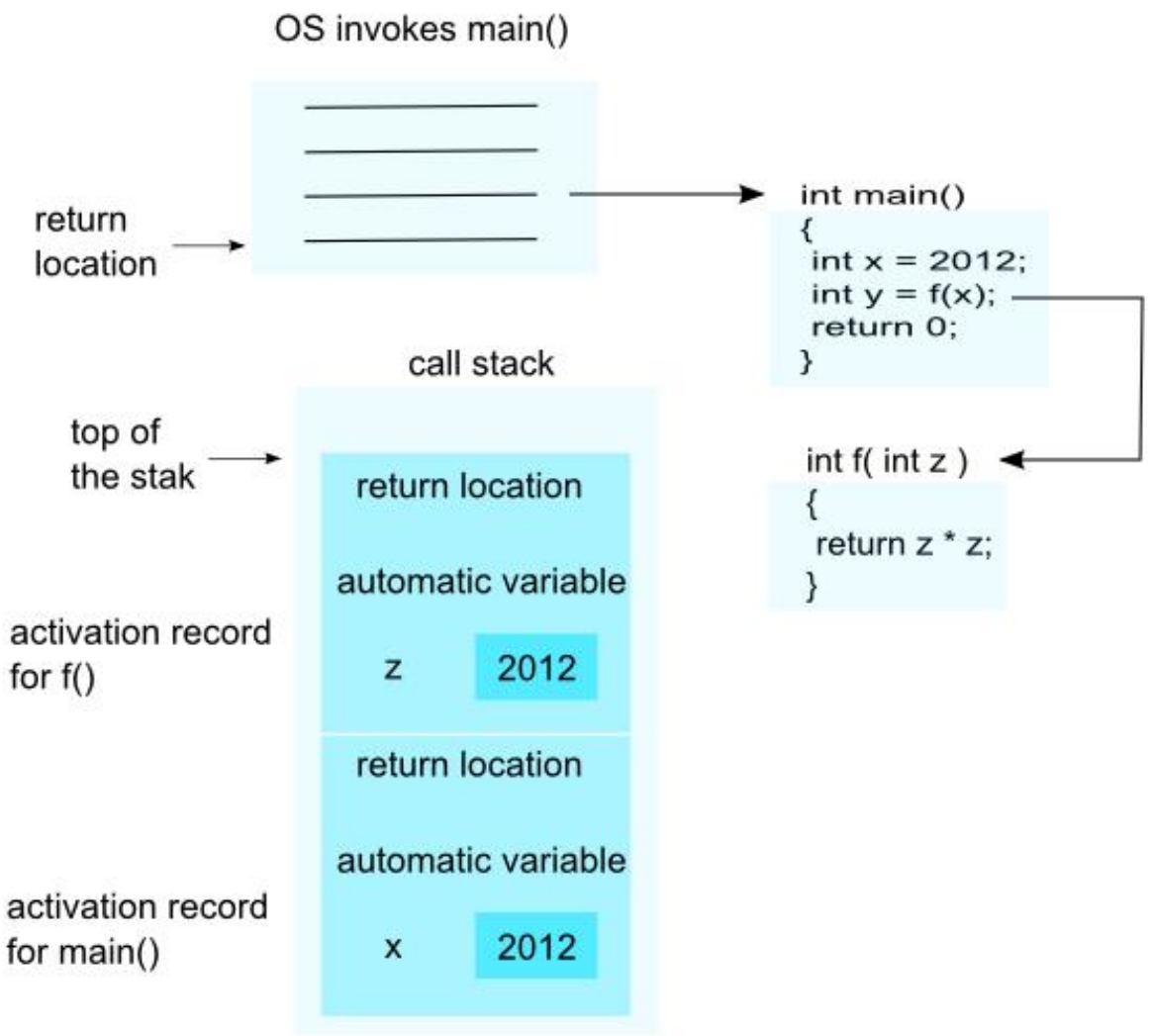
递归算法需要满足的三个条件：

1. 待求解问题的解可以分解为几个子问题的解；
2. 待求解问题与分解之后的子问题，只是数据规模不同，求解思路完全相同；
3. 存在递归终止条件

编写递归算法的正确思维方式是

- 假设子问题是已经解决的，不要陷入试图用人脑分解整个“递”与“归”过程的思维误区；
- 先写出递归终止条件，再写递推公式。

2. 递归调用实现的内部过程



2. 递归调用实现的内部过程

通常当一个函数调用另一个函数时，系统需完成三件事：

- 为被调用过程的局部变量分配存储区；
- 将所有的实参、返回地址等信息传递给被调用过程保存；
- 将控制转移到被调过程的入口。

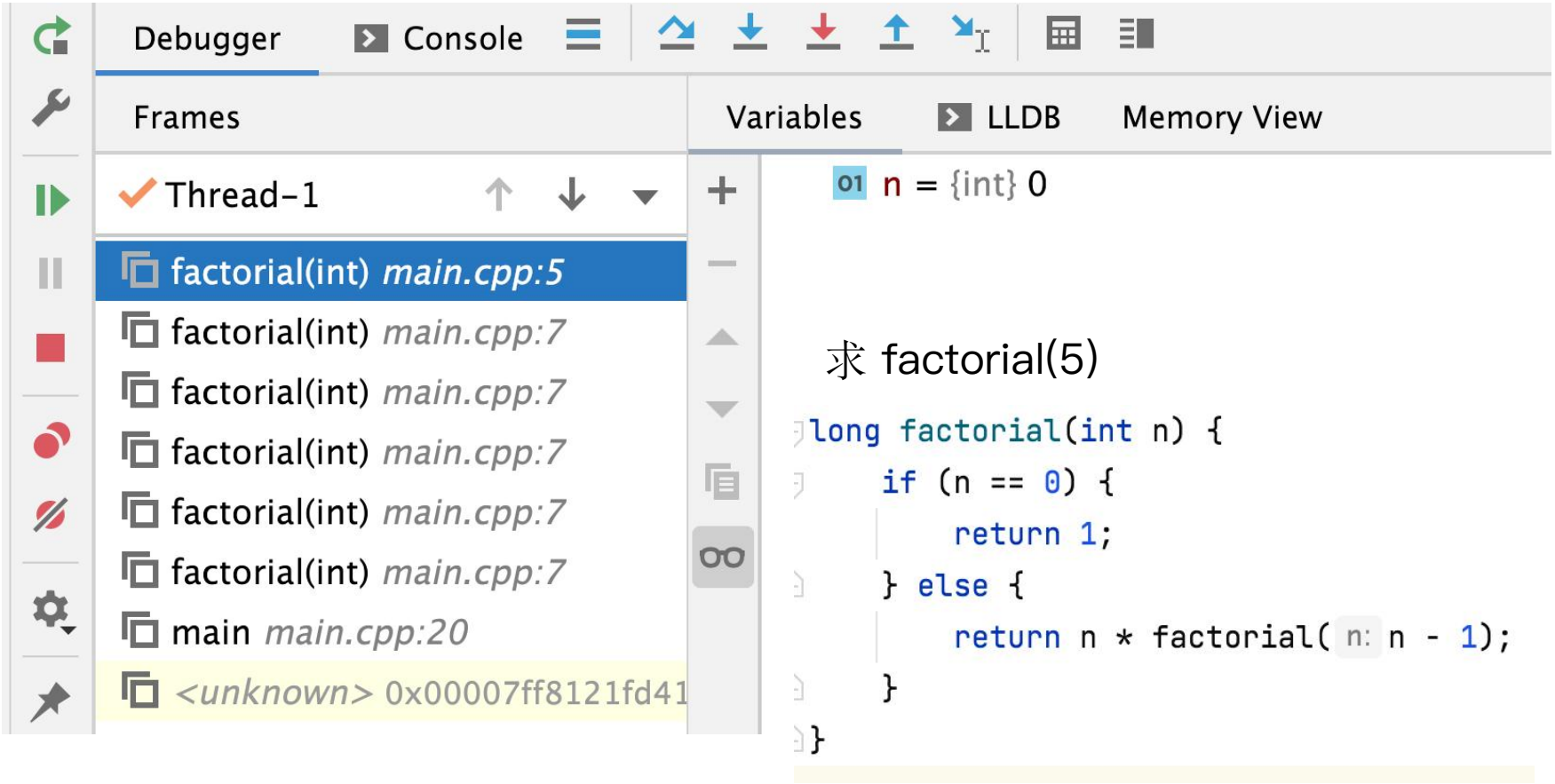
从被调用过程返回调用过程之前，系统也应完成三件工作：

- 保存被调过程的计算结果；
- 释放被调过程的数据区；
- 依照被调过程保存的返回地址将控制转移到调用过程。

在计算机中，是通过使用系统栈来完成上述操作的。

2. 递归调用实现的内部过程

一个递归函数的运行过程类似于多个函数的嵌套调用，只是调用函数和被调用函数是同一个函数.



递归过深导致系统的栈溢出问题

- 原因：每调用一个新的函数，都会将临时变量等封装为栈帧入栈，等函数执行完成后，再讲栈帧出栈。如果递归求解的数据规模很大，调用层次很深，可能塞满函数栈，导致栈溢出。
- 解决方法：
  - 限制递归调用的最大深度
  - 看是否能改写成尾递归
  - 将递归代码改写成非递归

递归过深导致系统的栈溢出问题

- 尾递归:

递归调用出现在函数中的最后一行，并且没有任何局部变量参与最后一行代码的计算。此时支持“尾递归优化”的编程语言就可以在执行尾递归代码时不进行入栈操作。示例

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

```
long factorial(int n,  int res) {
    if (n <= 1) {
        return res;
    }
    return factorial(n-1, n*res);
}

//求5的阶乘
factorial(5,1)
```

尾递归

通过缓存已经计算过的值来避免重复计算，从而提高效率。

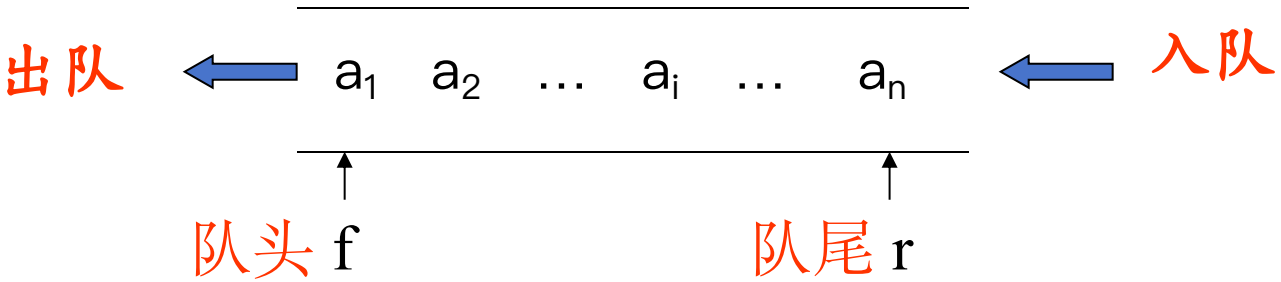
```cpp
// 记忆化递归实现 Fibonacci 数列
long fibonacci_2(long n, std::unordered_map<long, long>& memo) {
    if (n <= 1) {
        return n;
    }
    if (memo.find(n) != memo.end()) {
        return memo[n];
    }
    memo[n] = fibonacci_2( n: n - 1,  &: memo) + fibonacci_2( n: n - 2,  &: memo);
    return memo[n];
}
```

3.4 队列

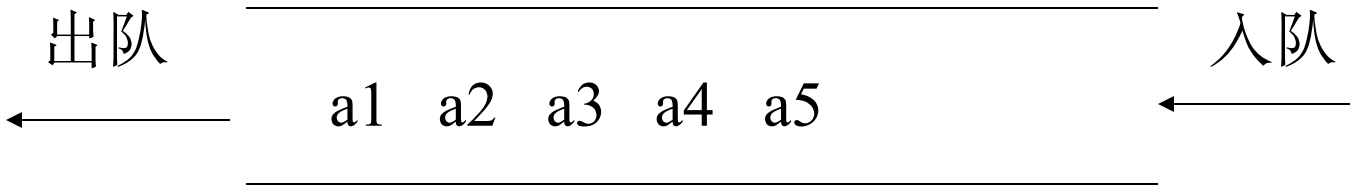
3.4 队列

3.4.1 队列的定义

队列是一种特殊的线性表，限定插入和删除操作分别在表的两端进行。具有**先进先出**(FIFO—First In First Out)的特点。



把允许插入的一端叫队尾(rear)，把允许删除的一端叫队头(front)。没有元素时称为空队列。



上图是一个有5个元素的队列，入队的顺序依次为a1,a2,a3,a4,a5，出队时的顺序将依然是a1,a2,a3,a4,a5。先进入队列的元素总是先离开队列。因此队列也称作先进先出(First In First Out)的线性表，简称FIFO表。

定义在队列结构上的基本运算

(1)构造空队列操作	InitQueue(&Q)
(2)销毁队列操作	DestroyQueue(&Q)
(2)判队空否函数	QueueEmpty (Q)
(3)元素入队操作	EnQueue(&Q,e)
(4)元素出队函数	DeQueue(&Q,&e)
(5)取队头元素函数	GetHead(Q,&e)
(6) 队列置空操作	ClearQueue(&Q)
(7)求队中元素个数函数	QueueLength(Q)

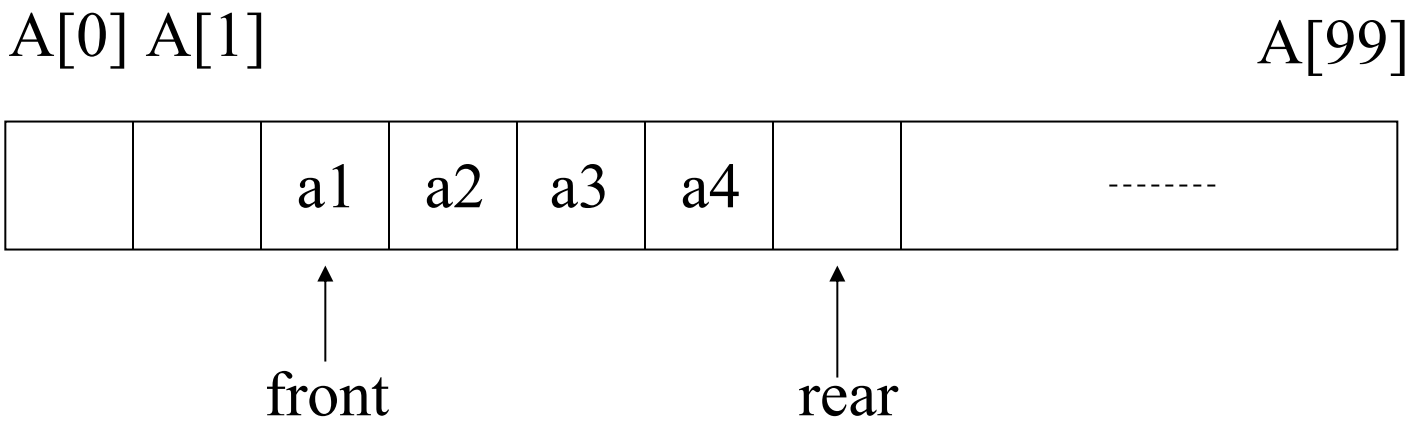
思考：可否用两个栈实现一个队列？如何实现？

3.4.2 队列的存储及运算实现

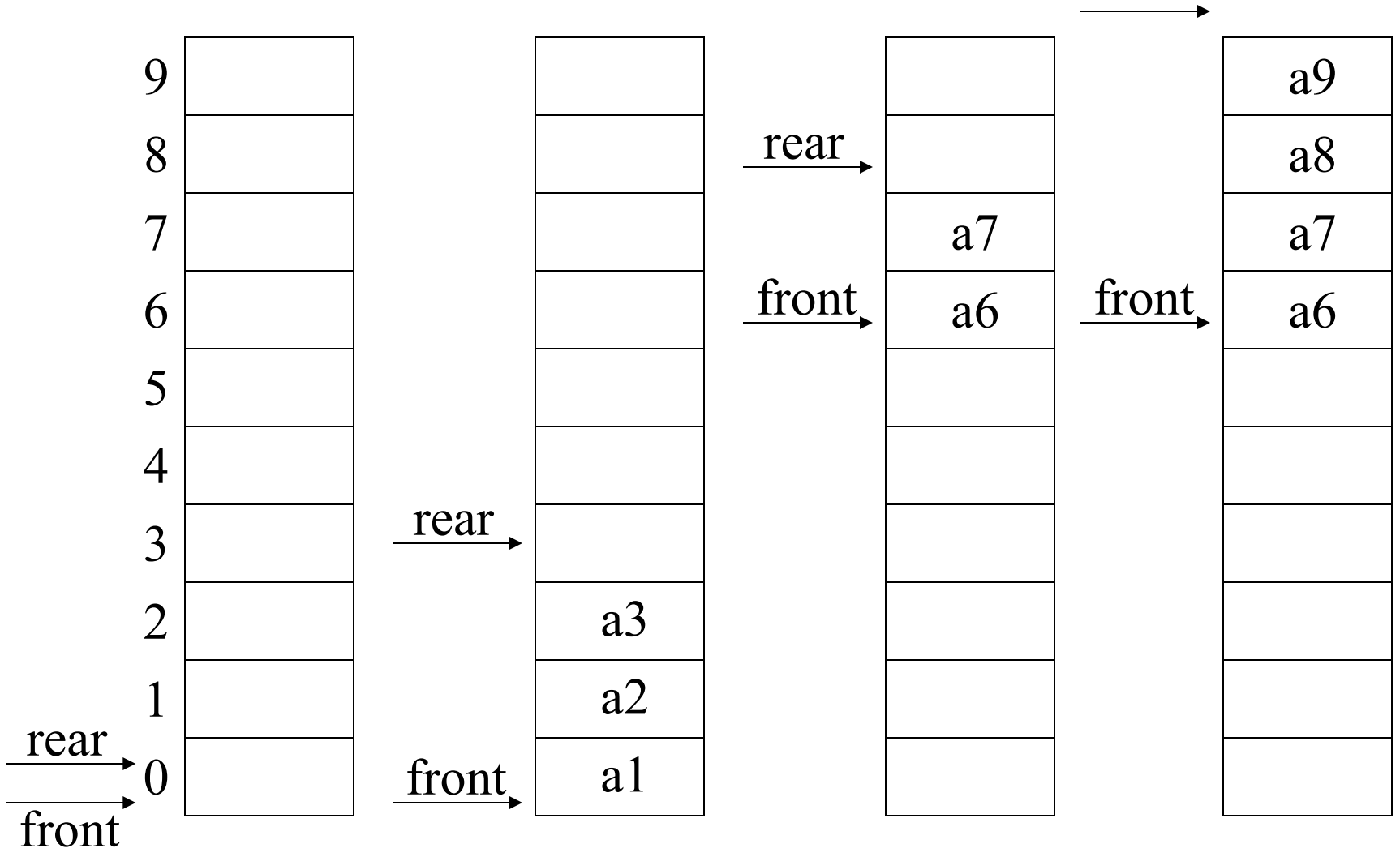
1.顺序队列（一般用法的顺序存储结构之缺陷）

•需附设两个指针front和rear分别指示队列头元素及队列尾元素的位置，为了描述方便，约定：

•空队列时front=rear=0；每当插入新的队列尾元素时，尾指针增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。**



- 随着入队出队的进行，会使整个队列整体向后移动，出现“假溢出”（最右图）

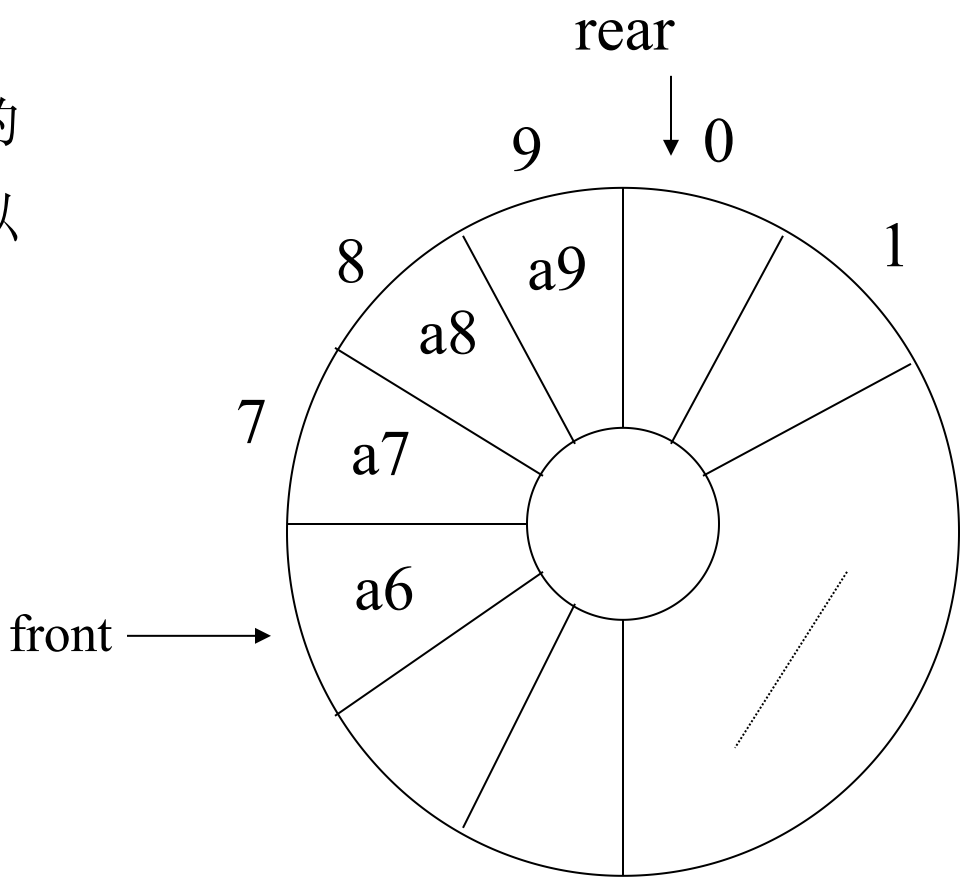


- 循环队列：
- 解决假溢出的方法：将队列的数据区看成头尾相接的循环结构，头尾指针的关系不变，将其称为“循环队列”，“循环队列”如下图所示。

只要队列元素个数小于总的可用空间，插入删除就可以一直进行下去

队尾插入一个元素时：
rear = (rear + 1) %10

队头删除一个元素时：
front = (front + 1) %10



- 从上图所示的循环队可以看出：
- 可见在队满和队空情况下都有：front==rear，这显然是必须要解决的一个问题。

方法之一是：

- 附设一个存储队中元素个数的变量如num，当num==0时队空，当num==MAXSIZE时为队满。
- 另一种方法是：
- 少用一个元素空间，当队尾指针加1就会从后面赶上队头指针，这种情况下队满的条件是：(rear+1) % MAXSIZE ==front，也能和空队区别开。

我们采用第二种方法。

```
typedef struct {//队列的顺序存储结构
    QElemType  *base;
    int  front;
    int  rear;
} SeQueue;

//初始化
Status  InitQueue ( SqQueue &Q)
{
   Q.base =(QElemType *) malloc( MAXQSIZE*sizeof(QElemtype) );
  if(!Q.base) exit (OVERFLOW);
  Q.front = Q.rear = 0;
   return OK;
}
```

注意类型是int不是指针

(2) 判队空

```
Status  QueueEmpty( SqQueue Q )
{
   if ( Q.front = = Q.rear )  return  TRUE;
   else   return  FALSE;
}
```

(3) 判队满

```
Status  QueueFull( SqQueue Q )
{
   if ( (Q.rear +1) % MAXQSIZE = = Q.front )  return  TRUE;
   else   return  FALSE;
}
```

(4) 求队长

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

(5) 入队

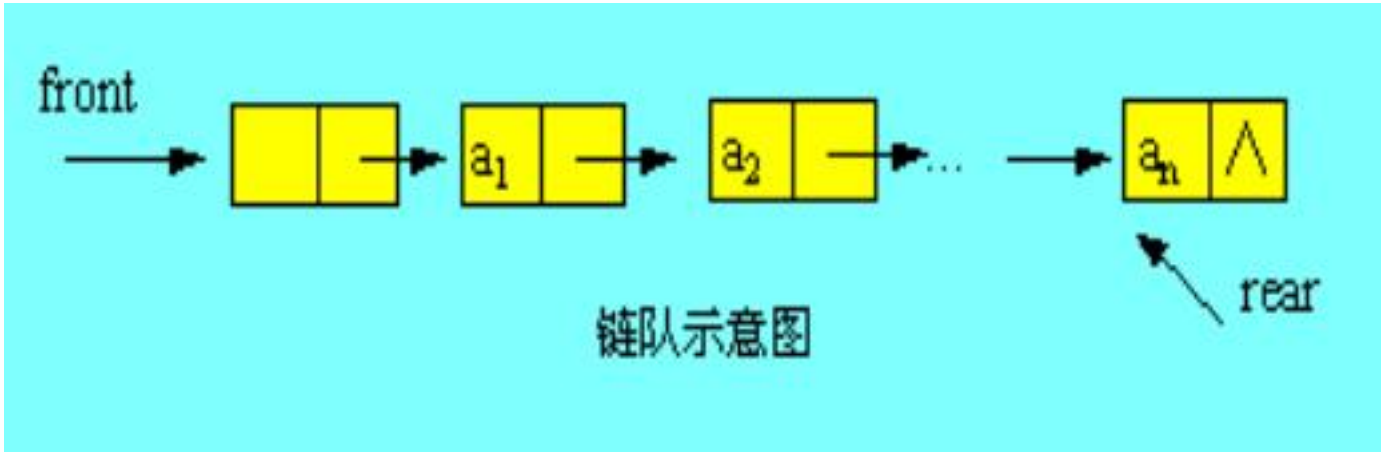
```
Statue   EnQueue ( SqQueue  &Q , Elemtype  e )
{
   if ( (Q.rear +1) % MAX = = Q.front )  return  ERROR;
    Q.base [Q.rear ]= e;
   Q.rear = (Q.rear +1) % MAXQSIZE;
   return  OK;
}
```

(6) 出队

```
Status  DeQueue ( SqQueue &Q ,  Elemtype  &e )
{
    if  (Q.front = = Q.rear )  return  ERROR;
    e = Q.base[Q.front ] ;
    Q.front = (Q.front +1) % MAXQSIZE;
    return  OK;
}
```

2.链队

链式存储的队称为链队。和链栈类似，用单链表来实现链队，根据队的FIFO原则，为了操作上的方便，我们分别需要一个头指针和尾指针，如图所示。

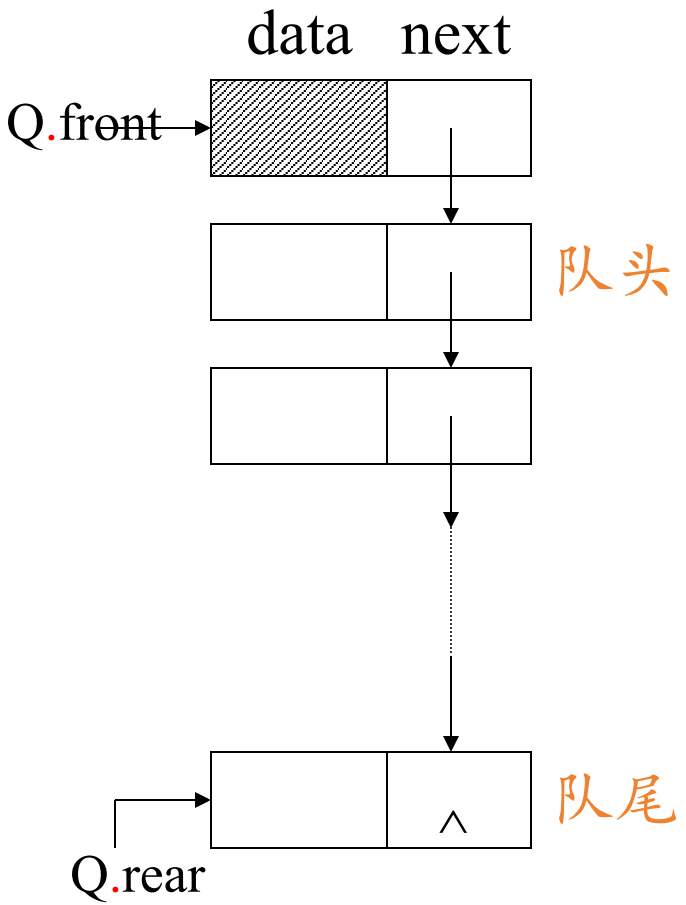


链队示意图

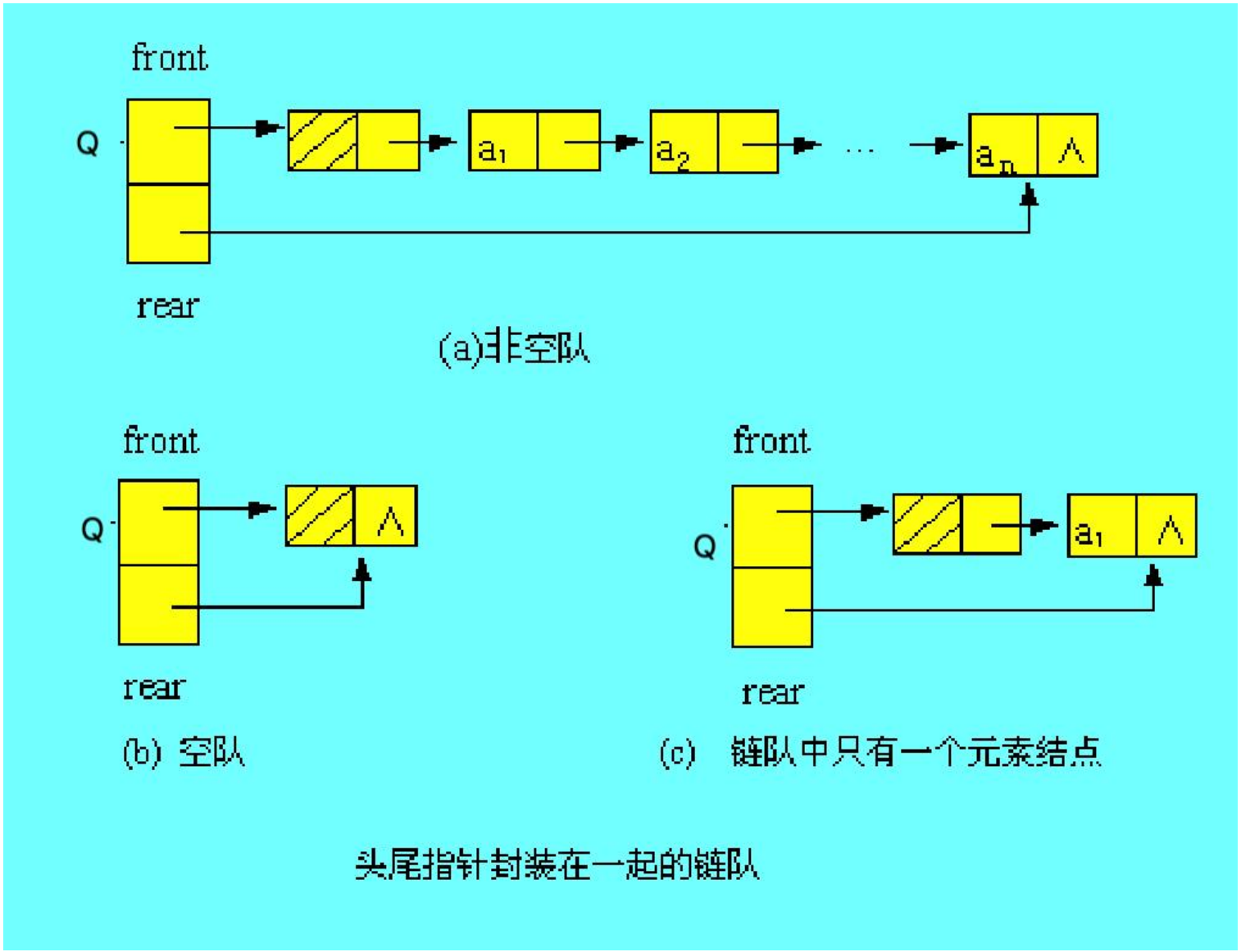
队列的链式存储结构以及操作的实现

[类型定义] 队头指针 + 队尾指针

```
typedef  struct QNode  {
    QElemtype     data;
    struct QNode  *next;
}  Qnode, *QueuePtr;

typedef struct  {
     QueuePtr front;
     QueuePtr rear;
}LinkQueue;
```



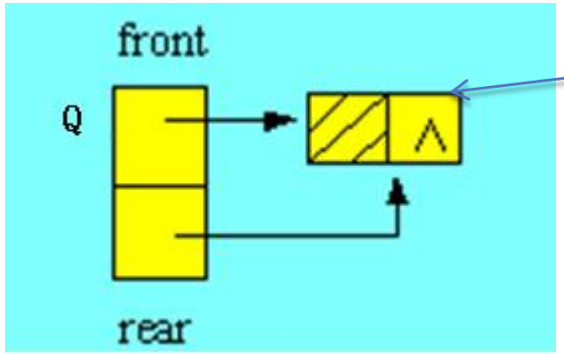
带头结点的链队如图所示：



队列的链式存储结构以及操作的实现

(1) 初始化链队

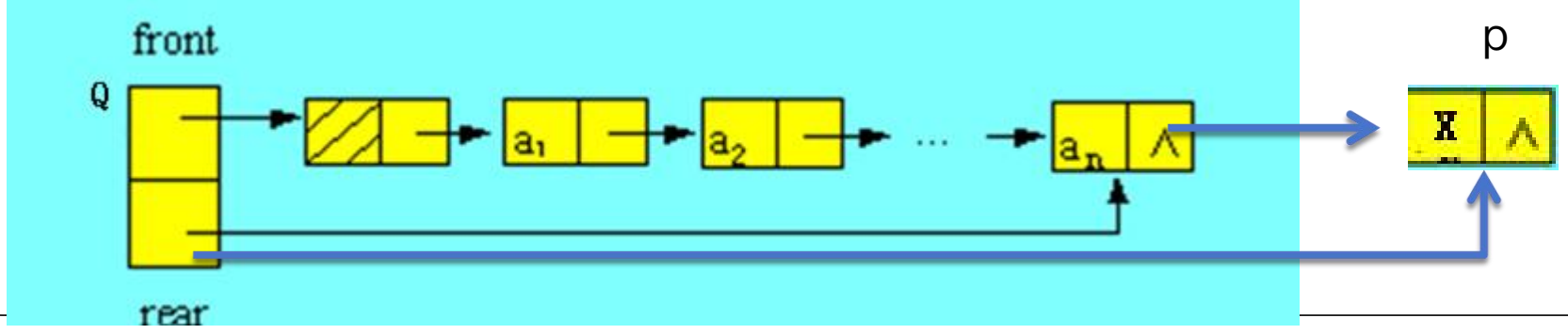
```
Status   InitQueue ( LinkQueue  &Q )
{// 链队带头结点
    Q.front = Q.rear = (QueuePtr)malloc(sizeof(Qnode)) ;
    if(!Q.front) exit (OVERFLOW);
   Q.front->next = NULL;
    return  OK;
}
```



头结点

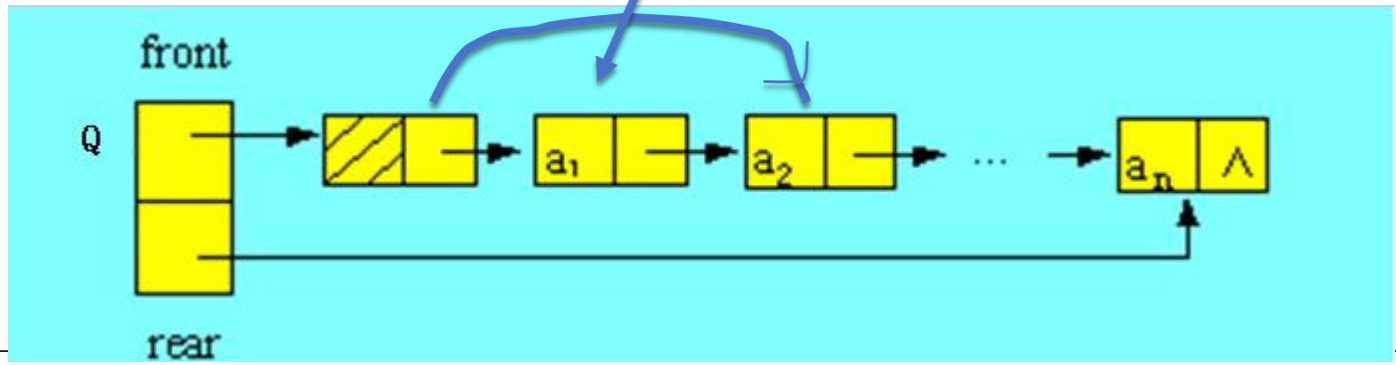
(2) 入链队

```
Status  EnQueue ( LinkQueue  &Q, QElemtype e )
{
        p = (QueuePtr) malloc ( sizeof ( QNode ) );
         if (!p)  exit (OVERFLOW);
        p->data = e;
        p->next = NULL;
        Q.rear->next = p;
        Q.rear = p;
        return OK;
 }
```



(3) 出链队

```
Status   DeQueue ( LinkQueue  &Q,  QElemtype  &e )
{   // 链队带头结点
    if  ( Q.front = = Q.rear )   return ERROR;  //如果为空
    p = Q.front->next;
    e = p->data;
    Q.front->next = p ->next;
     if(Q.rear == p)  Q.rear = Q. front; //如果出队的是最后一个元素
     free (p);
     return  OK;
}
```



本章知识点小结

- 堆栈的定义
- 顺序栈的实现
- 链式栈的实现
- 栈与递归
- 队列的定义
- 顺序队列、假上溢
- 循环队列的实现
- 链式存储队列的实现