数据结构

3　栈和队列

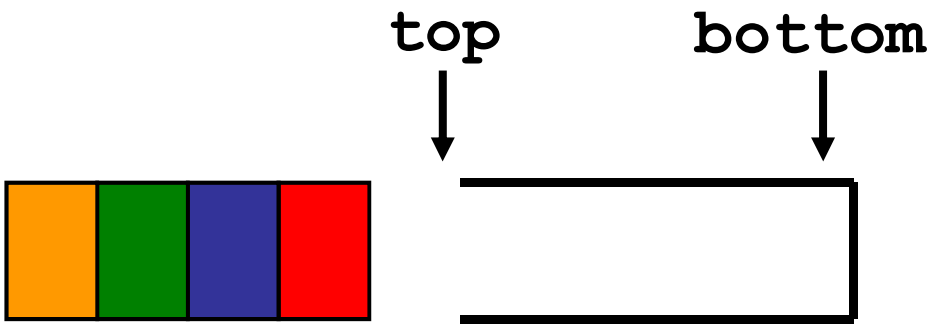
http://hwdong.com 董洪伟

主要内容

- 栈的类型定义
- 栈的表示
  - 顺序表示
  - 链表表示
- 栈的应用
  - 括号匹配
  - 走迷宫
  - 表达式计算
- 栈和递归
- 队列的类型定义
- 队列的表示
  - 链表表示
  - 顺序表示：循环队列
- 队列的应用
  - 农夫过河问题

栈的类型定义

- 定义
  - 只允许在同一端删除、同一端插入的线性表
  - 允许插入、删除的一端叫做栈顶(top)，另一端叫做栈底(bottom)
- 特性
  - 先进后出(FILO, First In Last Out)



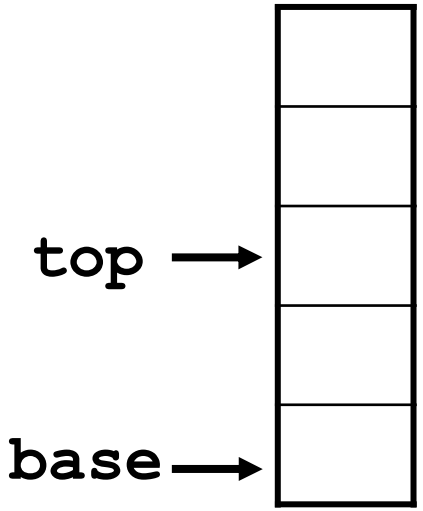
栈的类型定义

```
ADT Stack{
   数据对象:具有线形关系的一组数据
   操作:
      bool Push(e);   //入栈
      bool Pop(&e);   //出栈
      bool Top(&e);   //取栈顶
      bool IsEmpty(); //空吗?
      bool Clear();   //清空
}
```

栈的表示：顺序表示

- 顺序表示：即用数组来实现
  - 用数组存放堆栈中的数据
  - 再施加LIFO的访问限制：插入、删除只能从一端进行

```
typedef struct{
    SElemType *base;
    SElemType *top;
    int       stacksize;
} SqStack;
```



栈的表示：顺序表示

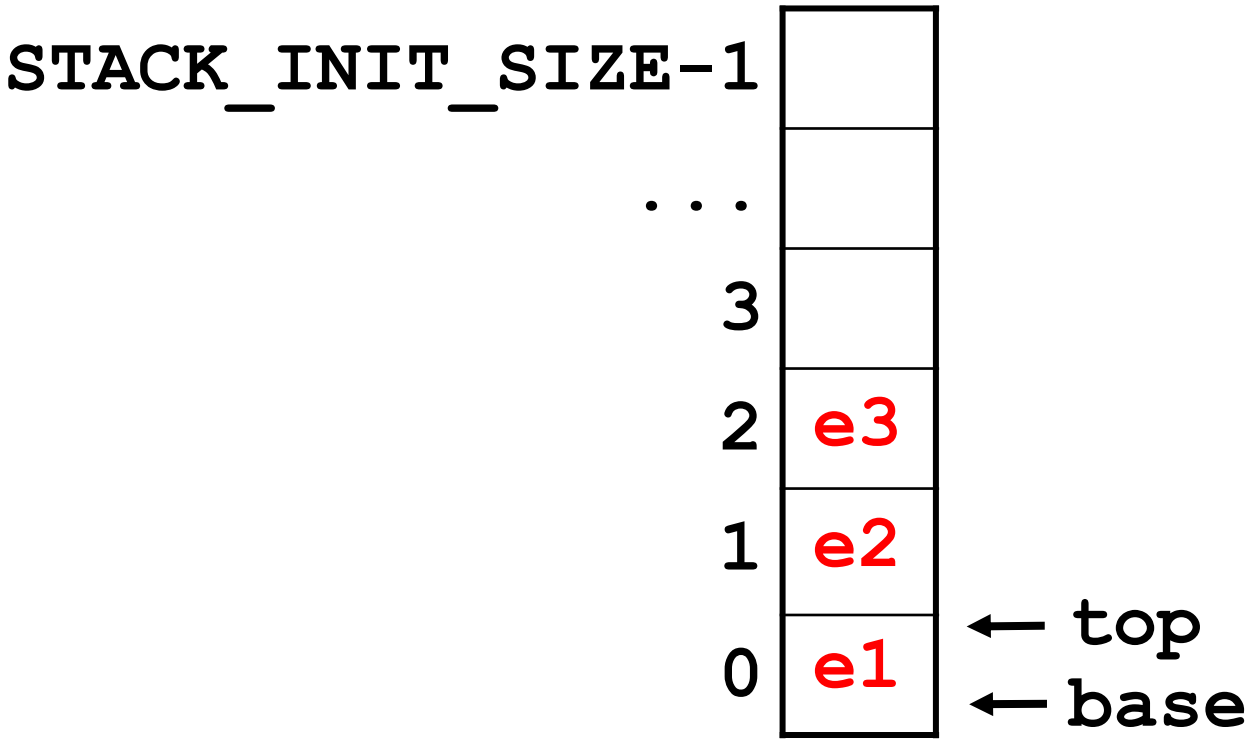
- 初始化

```
int InitStack(SqStack &S)
{
  //分配空间
  S.base = (SElemType*) malloc
     (STACK_INIT_SIZE * sizeof(SElemType));
  if(!S.base)  return NoMemory;
  S.top       = S.base;                 //设置指针
  S.stacksize = STACK_INIT_SIZE;  //设置大小
  return OK;
}
```

- 插入

```
int Push(SqStack &S, SElemType e){
      //若空间不够，重新分配
      if(S.top - S.base >= S.stacksize)  {
            S.base = (SElemType *)realloc
                  (S.base, (S.stacksize +
                        STACKINCREMENT)*
                        sizeof(SElemType));
         if(!S.base) return NoMemory;
           S.top = S.base + S.stacksize;
         S.stacksize += STACKINCREMENT;
      }
      *S.top ++ = e; //插入数据
      return OK;
}
```

```
int Push(SqStack &S, SElemType e){
      if(S.top - s.base >= S.stacksize)
      {     ...   }     //重新分配空间（略）
      *S.top ++ = e; return OK;
}
```

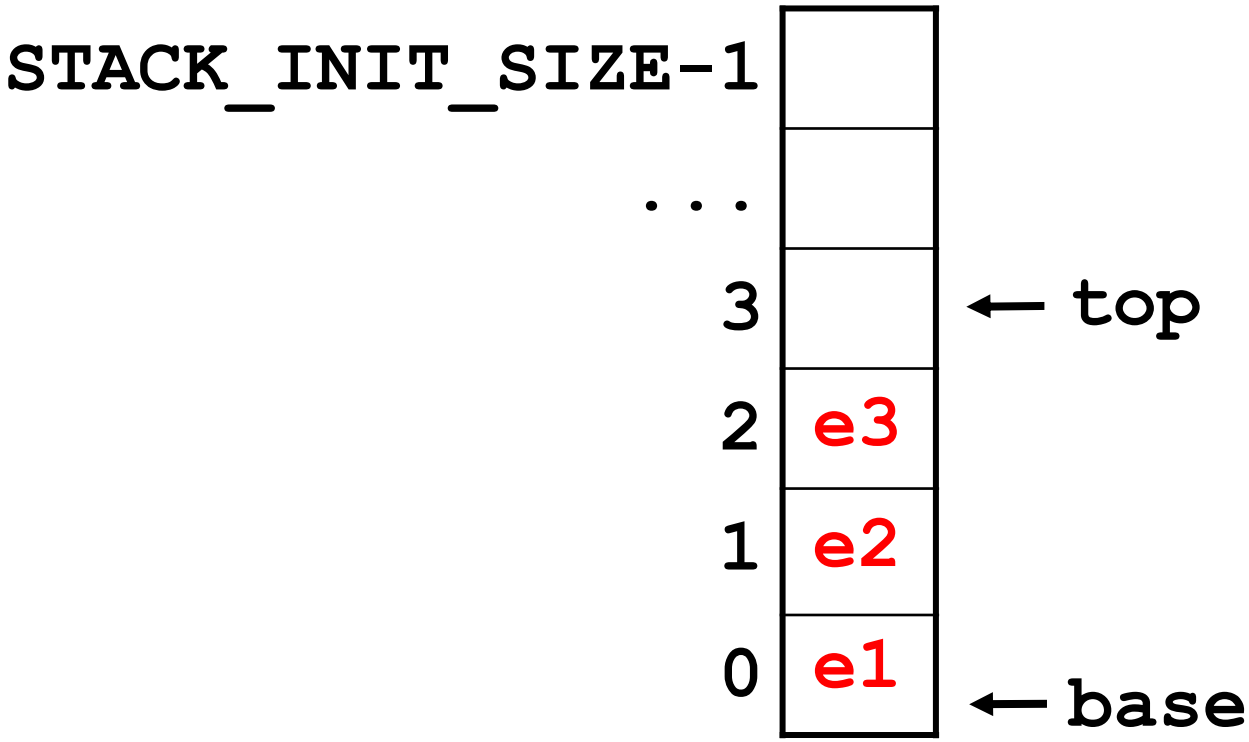


栈的表示：顺序表示

- 删除

```
int Pop(SqStack &S, SElemType &e)
{
    if(S.top == S.base)   //空栈
        return ERROR;
    e = * --S.top;        //出栈
    return OK;
}
```

```
int Pop(SqStack &S, SElemType &e){
      if(S.top == S.base)   //空栈
            return ERROR;
      e = * --S.top;        //出栈
      return OK;
}
```



栈的表示：链式表示

- 链式表示
  - 使用链表来实现
  - 栈不就是线性表 + LIFO限制么？
  - 参照线性表的链式表示

栈的应用

- 栈的应用
  - 颠倒元素顺序
    - 数制转换
  - 记录“历史信息”
    - 括号匹配的检验
    - 行编辑程序
    - 走迷宫
    - 表达式计算

栈的应用：数制转换

$$N = a_k d^k + a_{k-1} d^{k-1} + \ldots + a_1 d^1 + a_0$$

十进制数N转换为d进制数的转换，原理：

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

商　　余数

(其中：div 为整除运算，mod 为求余运算)

例如：$(1348)_{10} = (2504)_8$，其运算过程如下：

N	N div 8	N mod 8
1348	168	4 (个位)
168	21	0 (十位)
21	2	5 (百位)
2	0	2 (千位)

栈的应用：数制转换

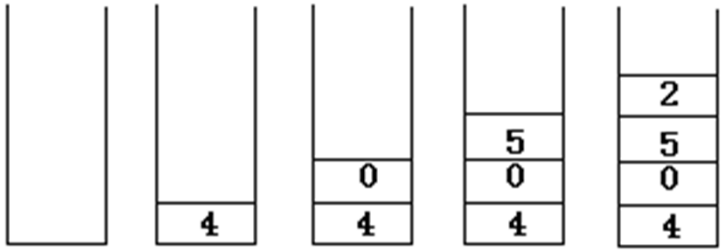
输入：任意一个非负十进制整数

输出：与其等值的八进制数。

由于输出每位数字(2、5、0、4)与得到每位数字（4、0、5、2）的次序正好相反，因此，可以用栈保持依次得到的各位(个、十、百、...)。

栈的应用：数制转换

```
void conversion () {
// 输入非负十进制整数，输出对应的八进制数
  SqStack  S; int N, int e;
  InitStack(S); // 构造空栈
  scanf ("%d",&N);
  while (N) {
     Push(S, N % 8);    N = N/8;
  }
  while (!StackEmpty(S)) {
     Pop(S,e);
     printf ( "%d\n", e );
  }
} // conversion
```



入栈过程

栈的应用：括号匹配检测

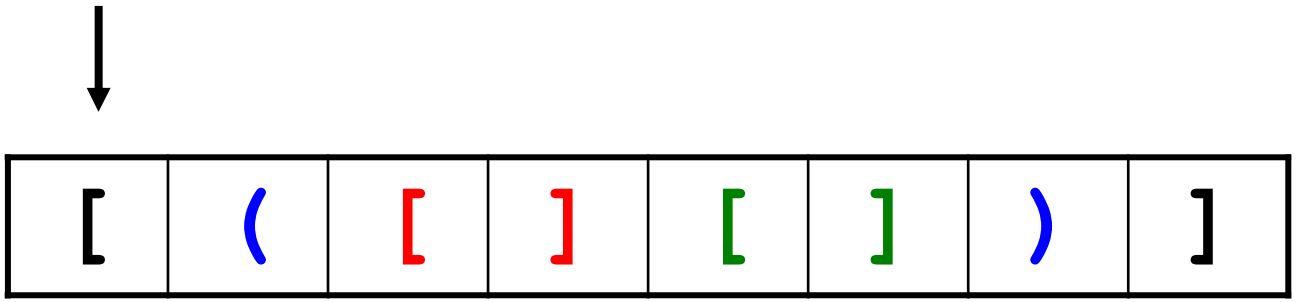
- 问题
  - 括号、引号等符号是成对出现的，必须相互匹配
  - 设计一个算法，自动检测输入的字符串中的括号是否匹配
  - 比如：
    - {}[([][])]匹配
    - [(]), (()]都不匹配

栈的应用：括号匹配检测

- 括号的匹配规则
  - 从里向外开始
  - 左括号应当和最近的右括号匹配
  - [([ ] [ ])]

栈的应用：括号匹配检测

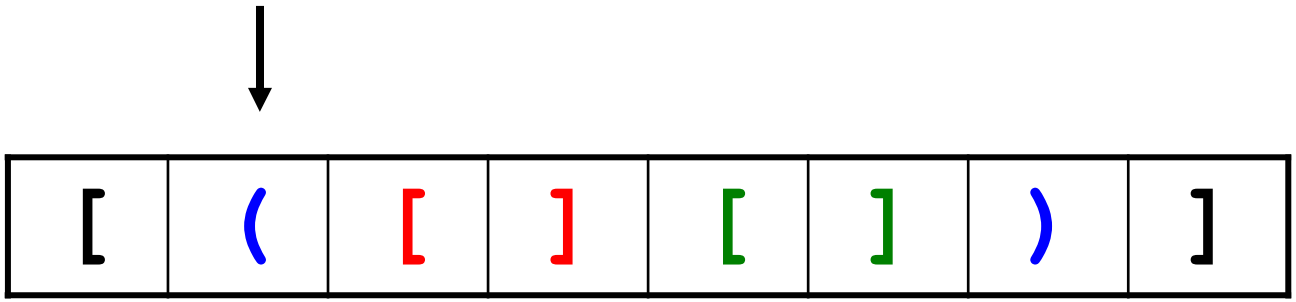
- 思考
  - 从左向右扫描字符串



当前是[，期待一个]

栈的应用：括号匹配检测

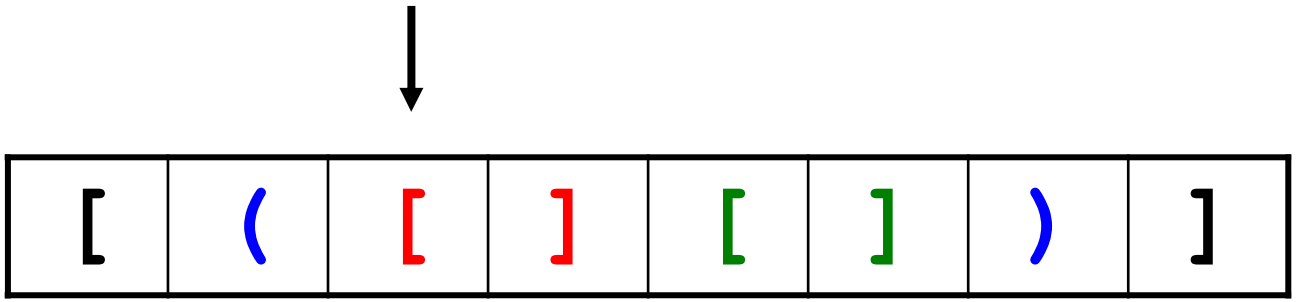
- 思考
  - 从左向右扫描字符串



当前是(，和刚才的[不匹配，说明相匹配的符号还在右边，继续扫描

栈的应用：括号匹配检测

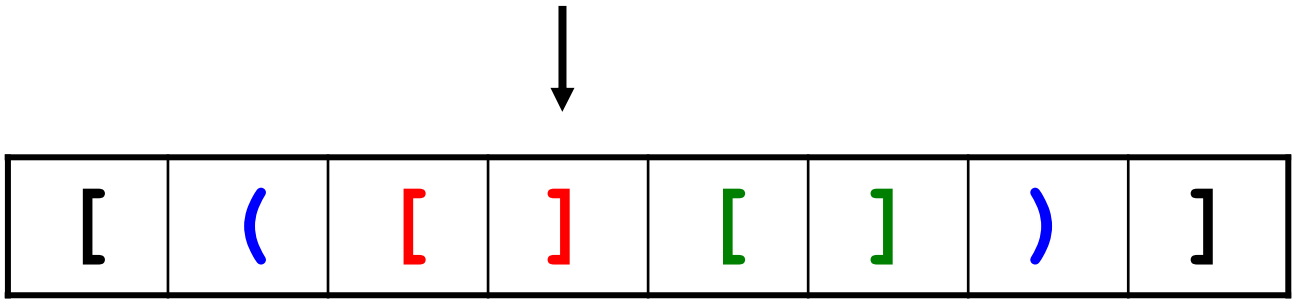
- 思考
  - 从左向右扫描字符串



当前是[，和刚才的(不匹配，说明相匹配的符号还在右边，继续扫描

栈的应用：括号匹配检测

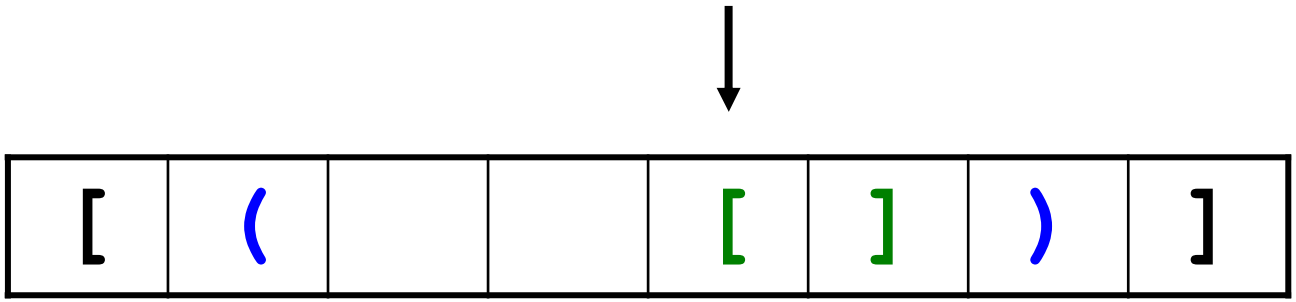
- 思考
  - 从左向右扫描字符串



当前是]，和刚才的[正好一对，
可以从字符串中“删去”不考虑了

栈的应用：括号匹配检测

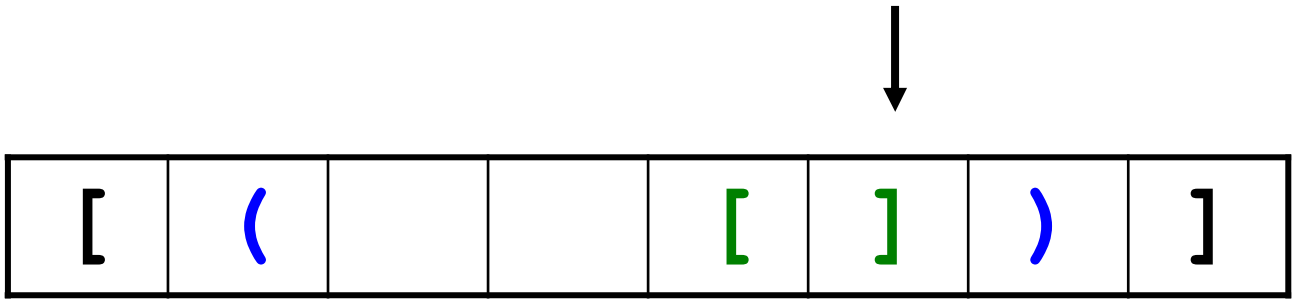
- 思考
  - 从左向右扫描字符串



当前是[，目前最近的一个是(，
不匹配，继续扫描

栈的应用：括号匹配检测

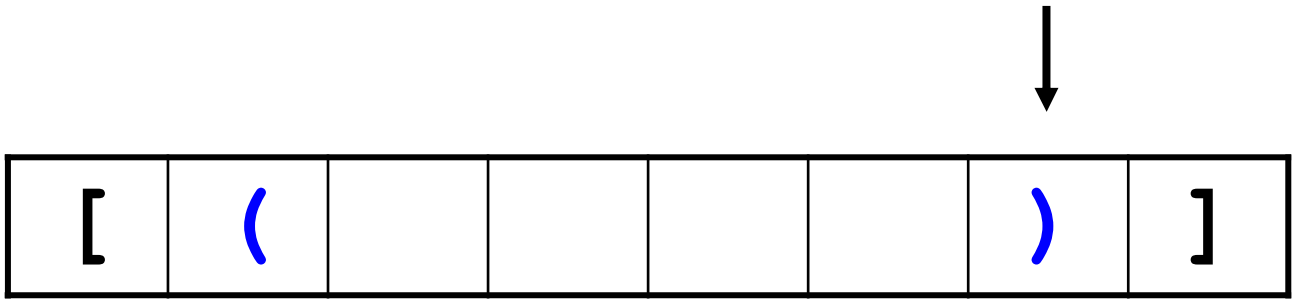
- 思考
  - 从左向右扫描字符串



当前是]，和刚才的[正好一对，
可以从字符串中“删去”不考虑了

栈的应用：括号匹配检测

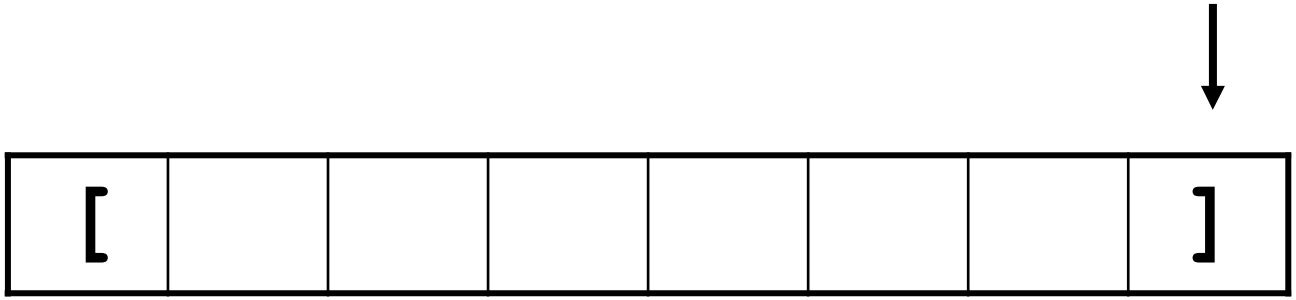
- 思考
  - 从左向右扫描字符串



当前是)，目前最近的一个是(，正好一对，可以从字符串中“删去”不考虑了

栈的应用：括号匹配检测

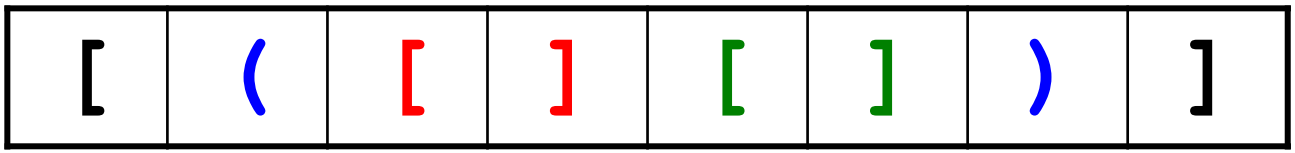
- 思考
  - 从左向右扫描字符串



当前是]，目前最近的一个是[，正好一对，可以从字符串中“删去”不考虑了，此时左右的括号都匹配成功

栈的应用：括号匹配检测

- 发现规律
  - 当扫描到当前字符的时候，需要知道已经扫描过的字符中，哪一个离它最近
  - 因此希望有一个工具，能够记录扫描的历史，这样可以方便的得到最近的上一次访问的字符



栈的应用：括号匹配检测

- 栈“记录历史”的特性
  - 人的记忆：
    - 越早发生的事情越难回忆
    - 越迟发生的事情越容易回忆
  - 栈的先进后出
    - 越早压入的元素越晚弹出
    - 越迟压入的元素越早弹出
  - 因此很自然的想到利用栈来模拟记忆

栈的应用：括号匹配检测

- 算法思想

准备一个栈，用于存放扫描遇到的左括号

从左向后扫描每一个字符{

如果遇到的是左括号，则入栈

如果遇到的是右括号，则

把栈顶字符和当前字符比较

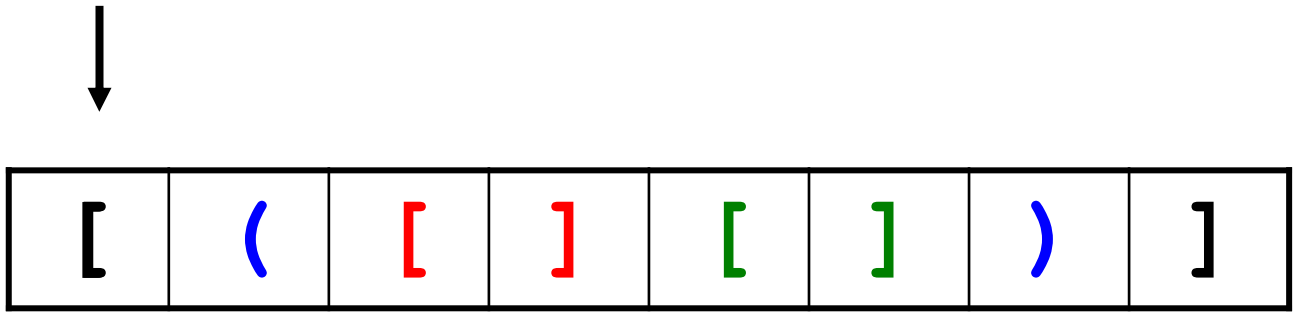
若匹配，则弹出栈顶字符，继续向前扫描

若不匹配，程序返回不匹配标志

}

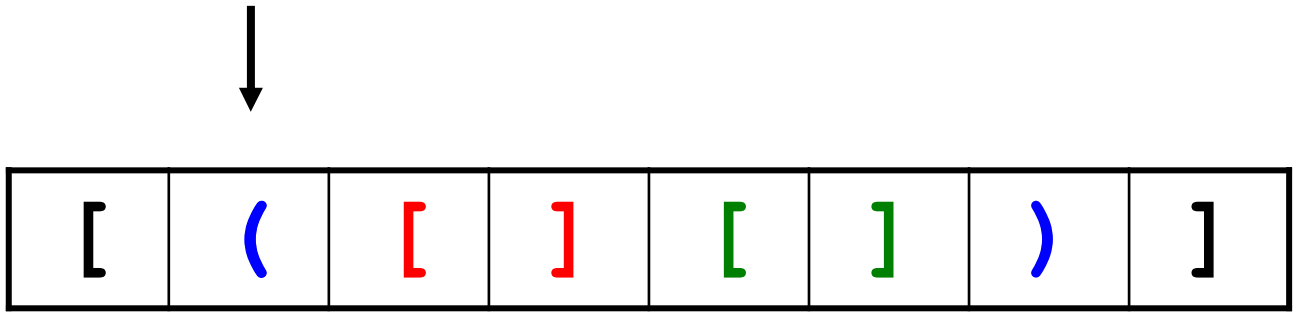
当所有字符都扫描完毕，栈应当为空

栈的应用：括号匹配检测



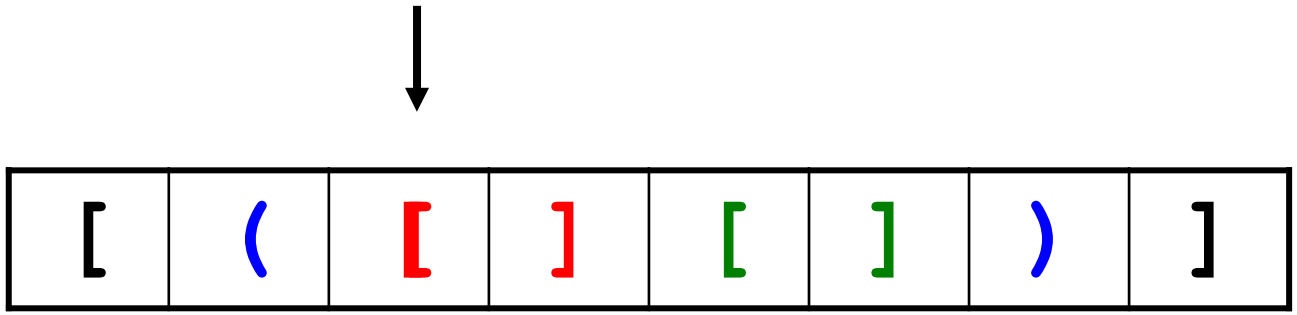
栈为空，
当前字符直接入栈

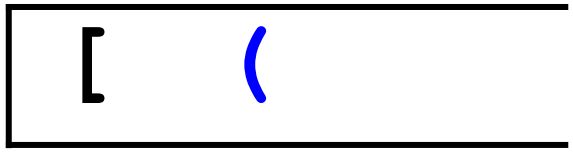
栈的应用：括号匹配检测



栈顶字符和当前字符不匹配，
当前字符入栈

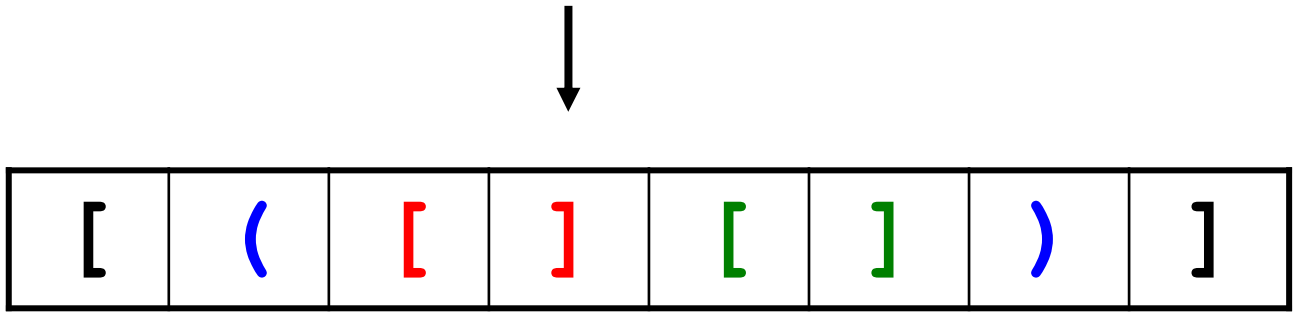
栈的应用：括号匹配检测

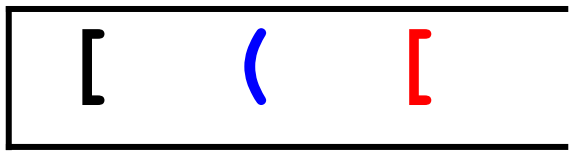




栈顶字符和当前字符不匹配，
当前字符入栈

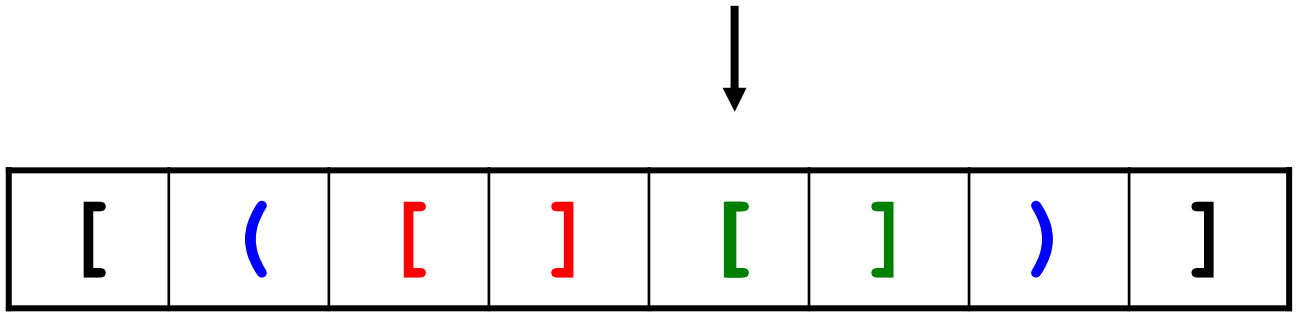
栈的应用：括号匹配检测

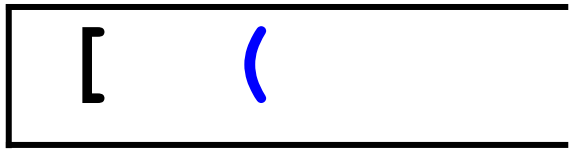




栈顶字符和当前字符匹配，
弹出栈顶字符

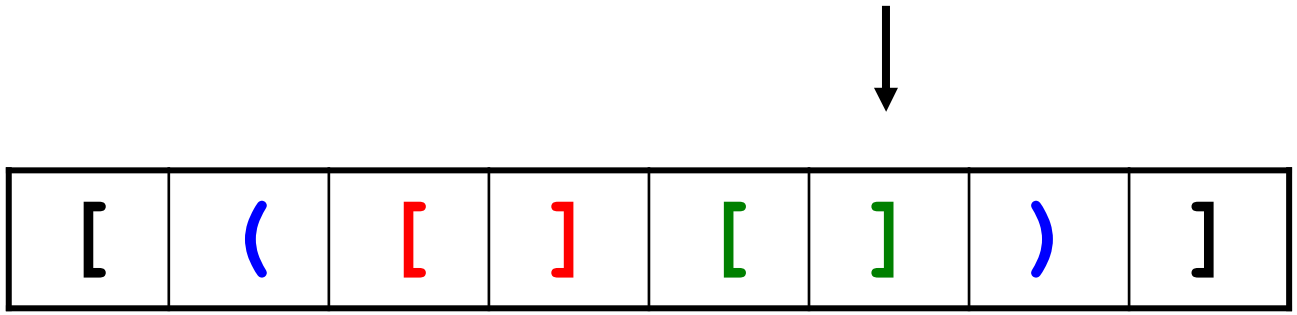
栈的应用：括号匹配检测

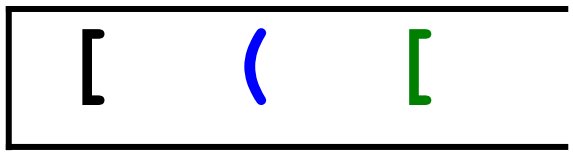




栈顶字符和当前字符不匹配，
当前字符入栈

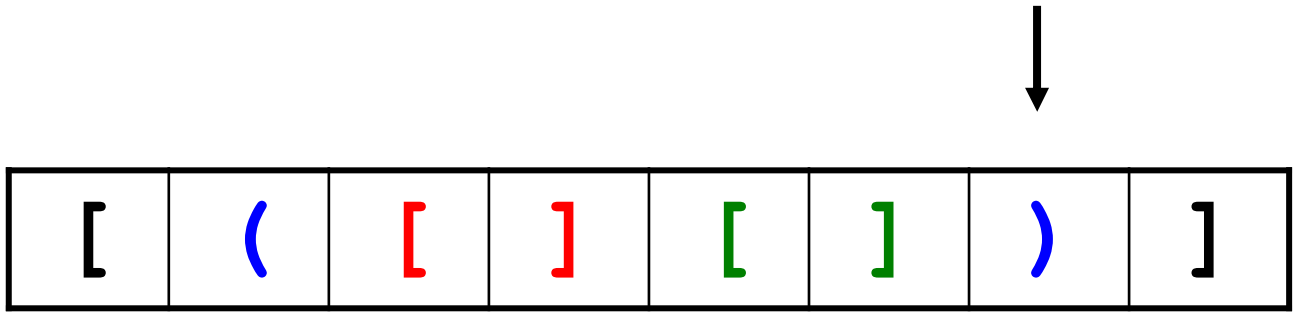
栈的应用：括号匹配检测

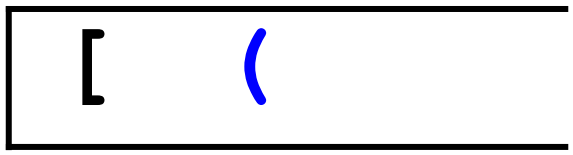




栈顶字符和当前字符匹配，
弹出栈顶字符

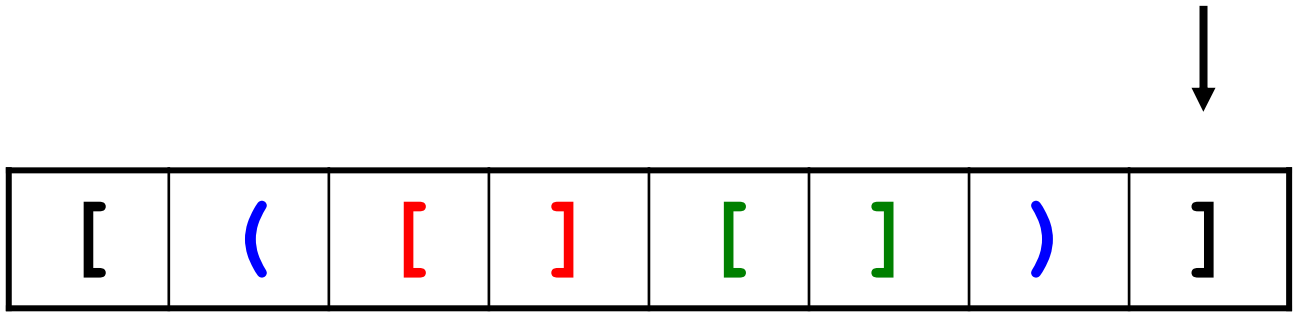
栈的应用：括号匹配检测





栈顶字符和当前字符匹配，
弹出栈顶字符

栈的应用：括号匹配检测



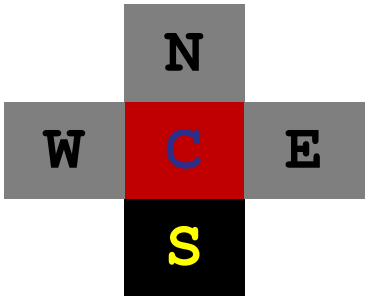
栈顶字符和当前字符匹配，
弹出栈顶字符

栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；

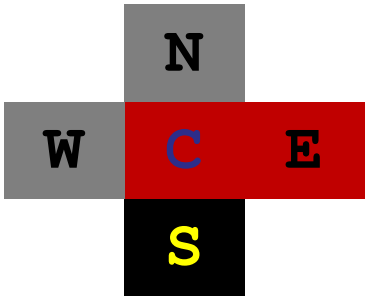
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



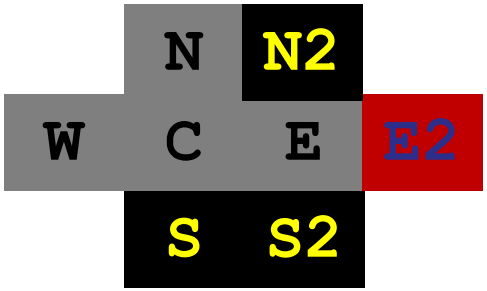
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



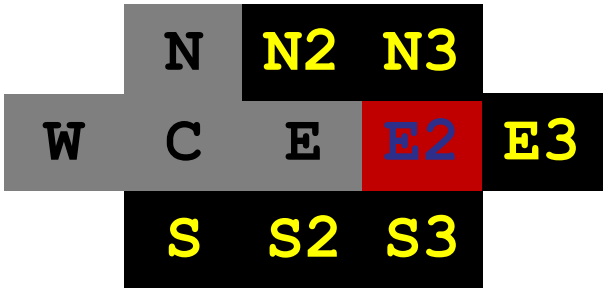
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



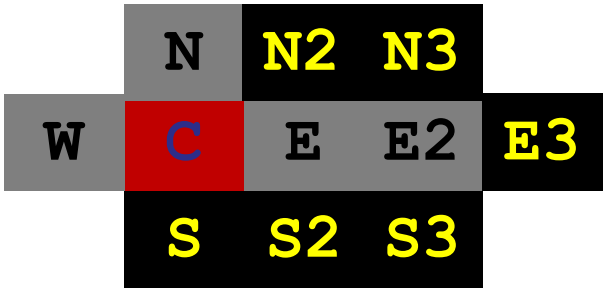
栈的应用：走迷宫

- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；

  - 如当前地点不可通（四周堵死），则退回到路径上的前一个地点；

栈的应用：走迷宫

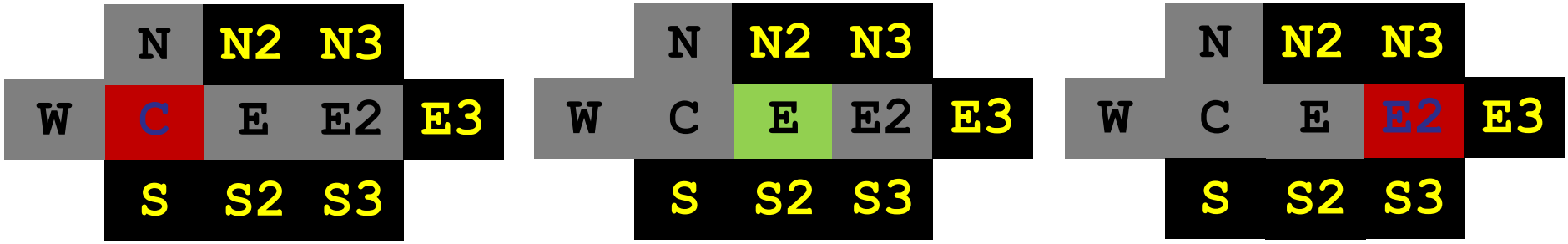
- 是一个探索过程
  - 从入口出发，当到达一个地点时，需要探索从该点按某个方向可到达的下一个地点，如可到达，则前进到新的地点；



  - 如当前地点不可通（四周堵死），则退回到路径上的前一个地点；

栈的应用：走迷宫

- 是一个探索过程
  - 如当前地点不可通（四周堵死或走过），则退回到路径上的前一个地点；



  - 需要保存走过的路径，按后进先退的过程回退！

栈的应用：走迷宫

- 再次应用栈来记录历史
  - 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
  - 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

- 算法

入口为当前位置

```
do{
    if当前位置可通{
        if当前是出口，结束
        else 当前位置入栈，向可前进方向前进
    }
    else{
        while(栈不空但栈顶位置的四周均不通)
            弹出栈顶
        if栈不空且栈顶位置还有其它方向未探索
            走向栈顶位置的顺时针下一位置
    }
}while(栈不空);
```

栈的应用：表达式求值

对含+、-、*、/、()的表达式进行求值，如

7+（4-2）*3-10/5

四则运算规则

（1） 先乘除、后加减

（2） 先左后右

（3） 先括号内后括号外

表达式的开头和结尾虚设#构成整个表达式的括号。

上述表达式变为

#7+（4-2）*3-10/5#

运算符和界限符统称为算符，其集合记为OP.有：

+、-、*、/、(、)、#

表达式求值：算符优先关系

【例】　　#3*(2+4)-8/2#

【分析】

依次读入操作数和运算符，考虑运算符的优先级别，在读入运算符curOP时，如下一个运算符nextOP 不比curOP优先，则可以用curOP计算;如下一个运算符nextOP比curOP优先，则先保存curOP，待nextOP运算完才能计算curOP 。依次类推。

例如“-8+2”，　-优先于+，可算- ，再算+

再如“-8/2”，/优先于-，应该先算/，再算-

表达式求值：算符优先关系表

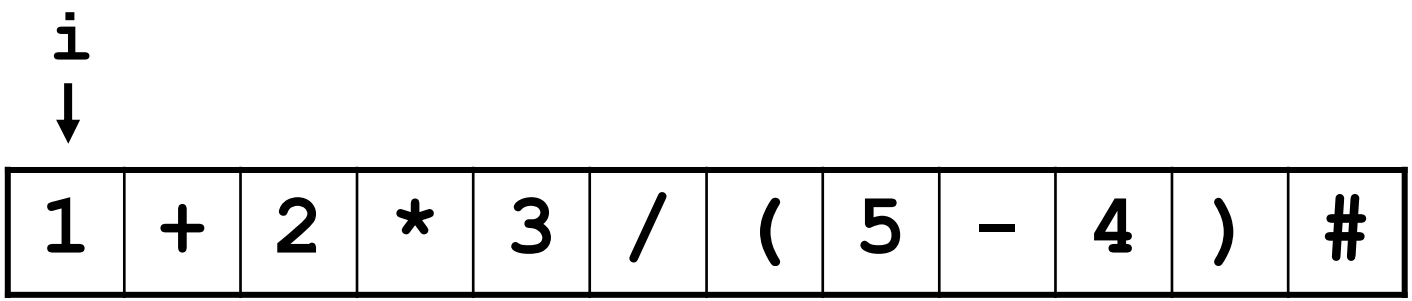
设表达式中算符 θ 1 出现在算符 θ 2 前，则两者优先关系：

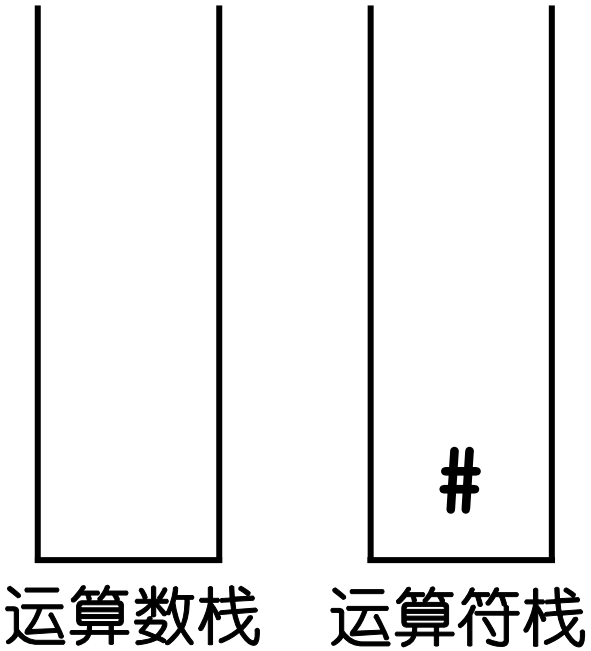
θ_1 \ θ_2	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	无
)	>	>	>	>	无	>	>
#	<	<	<	<	<	无	=

优先级相等的只有：（、）；　#、#

表达式求值：#3*(2+4)-8/2#的计算

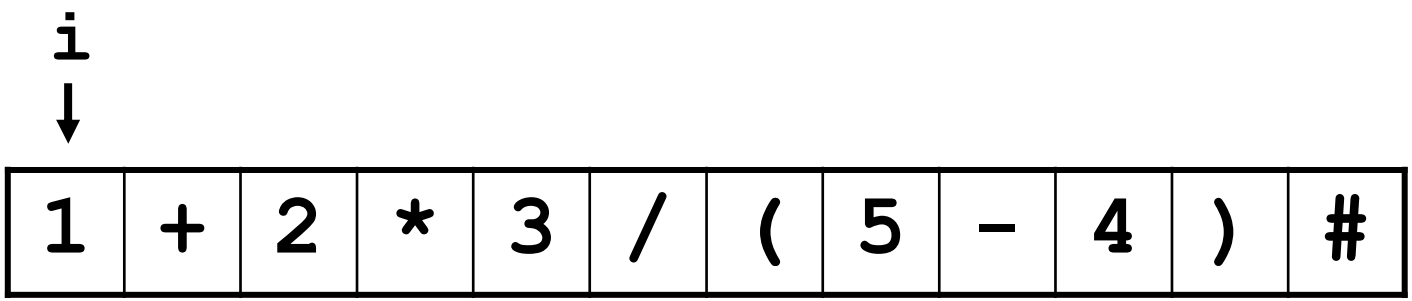
- 例

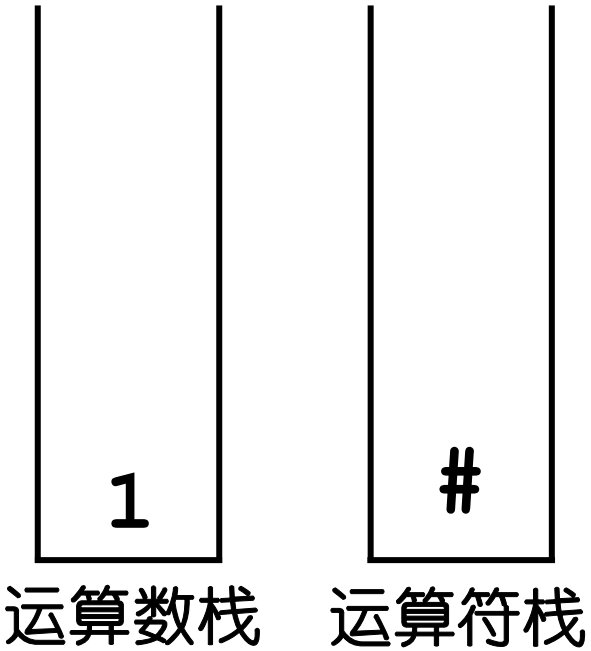




表达式求值：#3*(2+4)-8/2#的计算

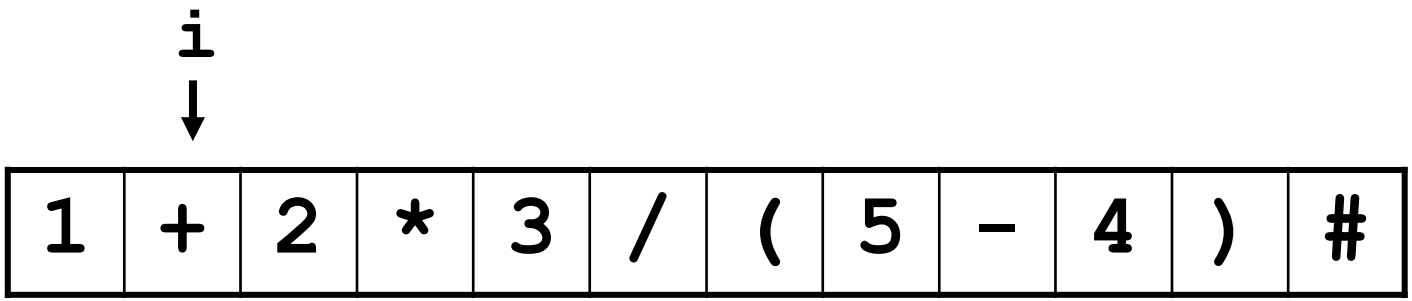
- 例

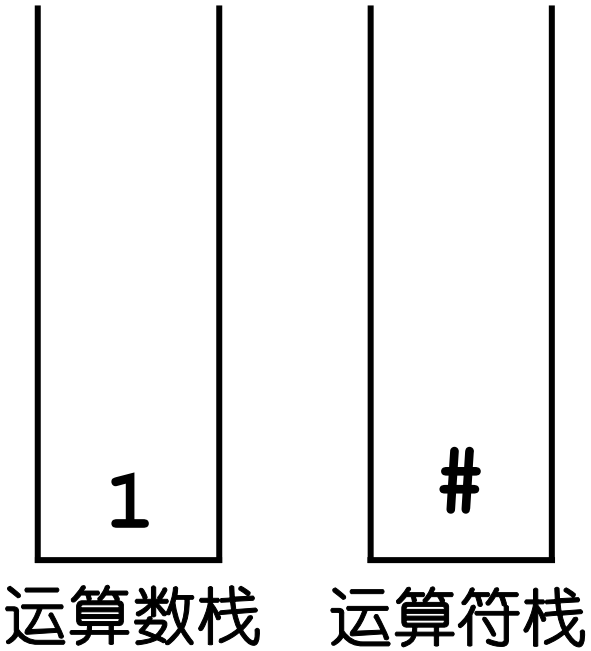




表达式求值：#3*(2+4)-8/2#的计算

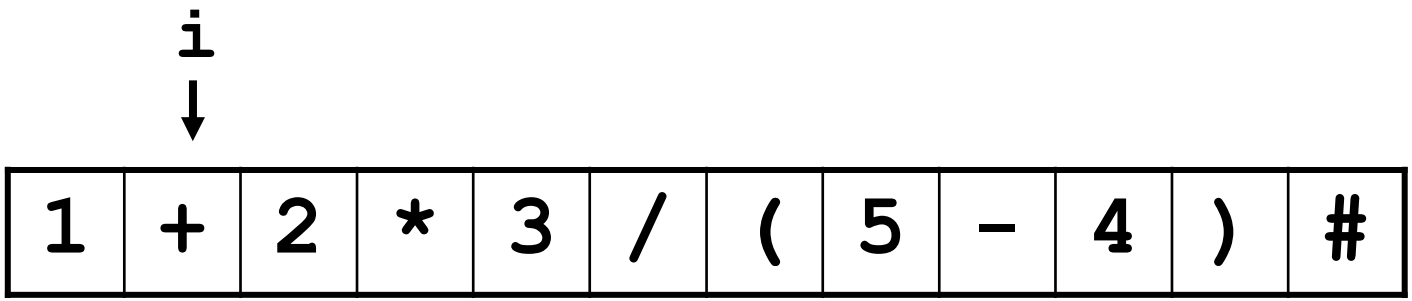
- 例

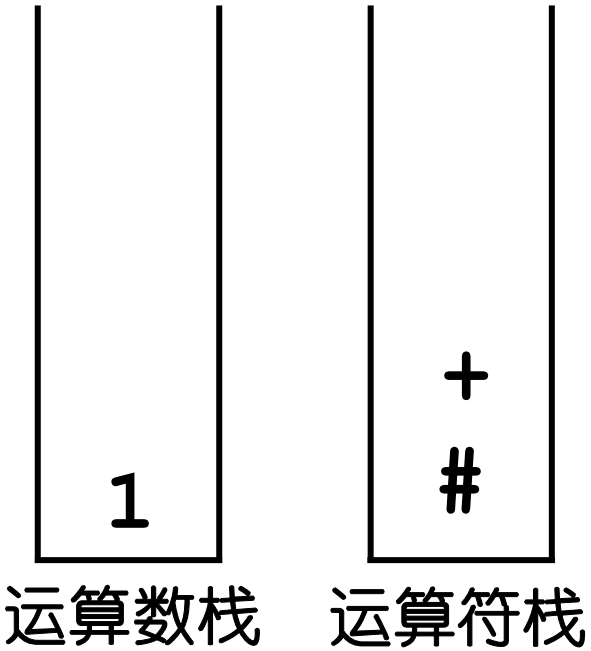




表达式求值：#3*(2+4)-8/2#的计算

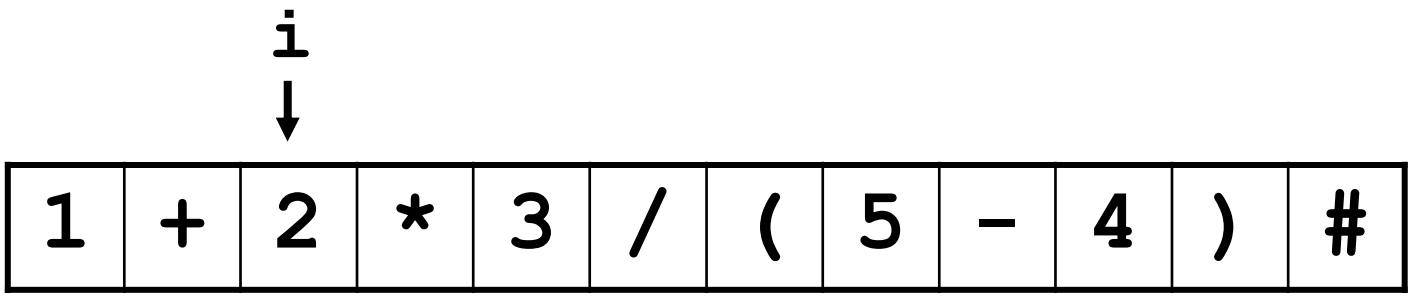
- 例

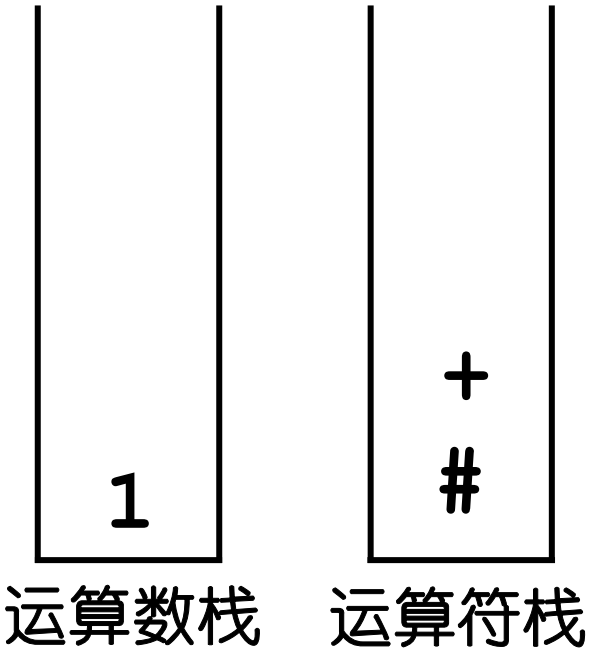




表达式求值：#3*(2+4)-8/2#的计算

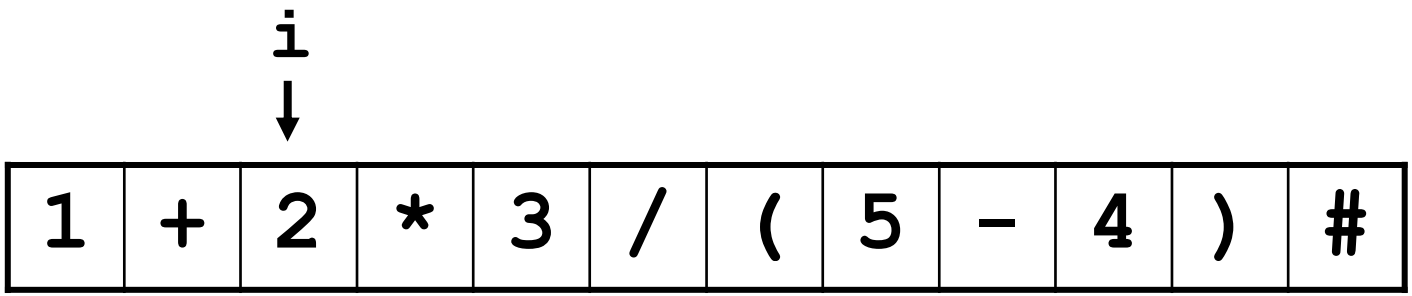
- 例

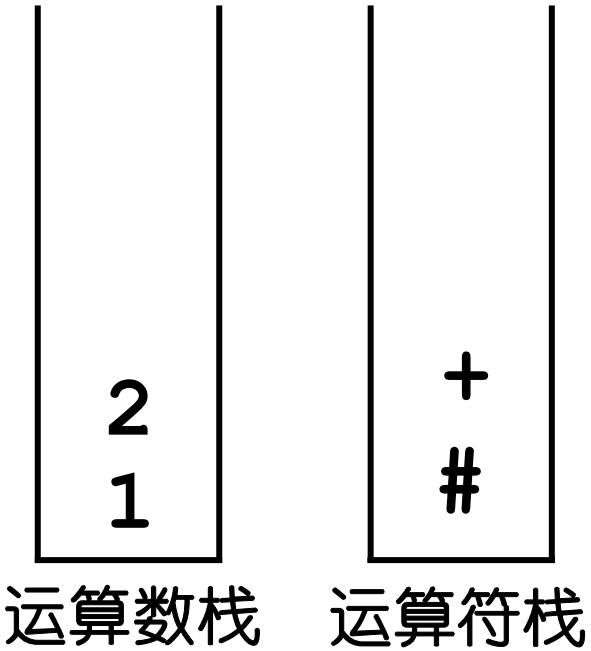




表达式求值：#3*(2+4)-8/2#的计算

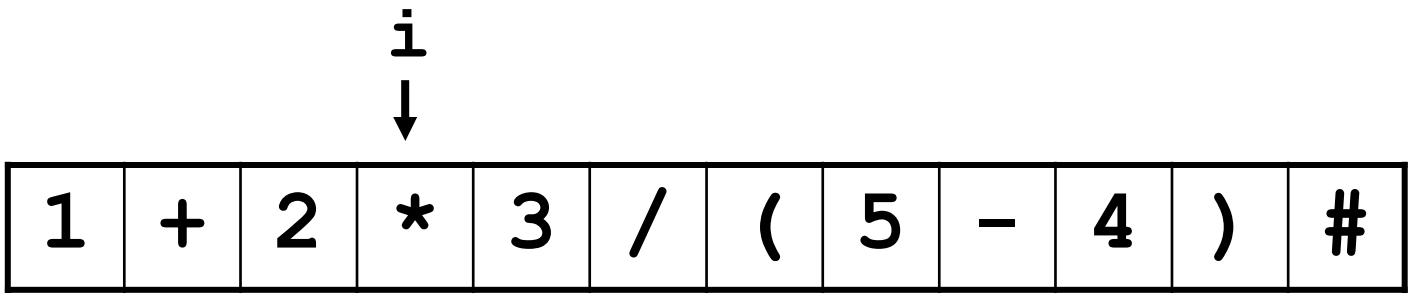
- 例

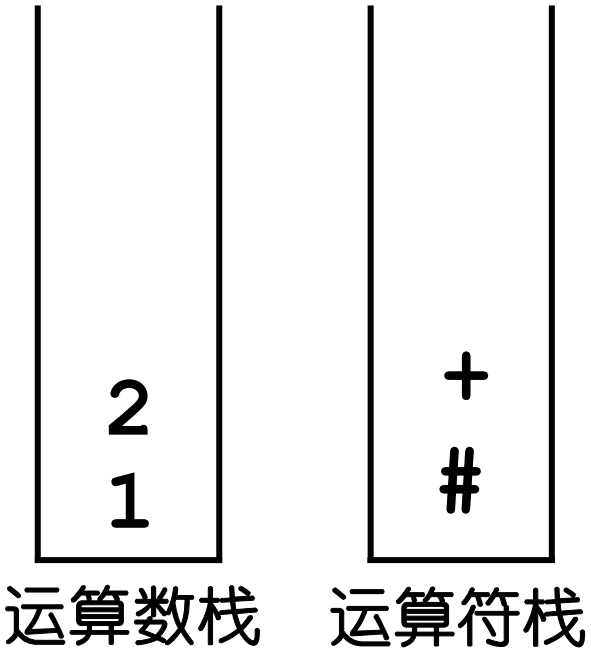




表达式求值：#3*(2+4)-8/2#的计算

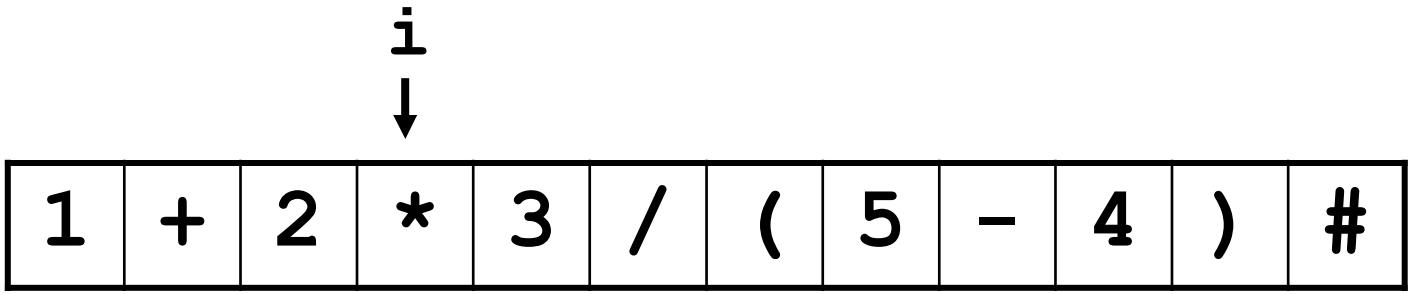
- 例

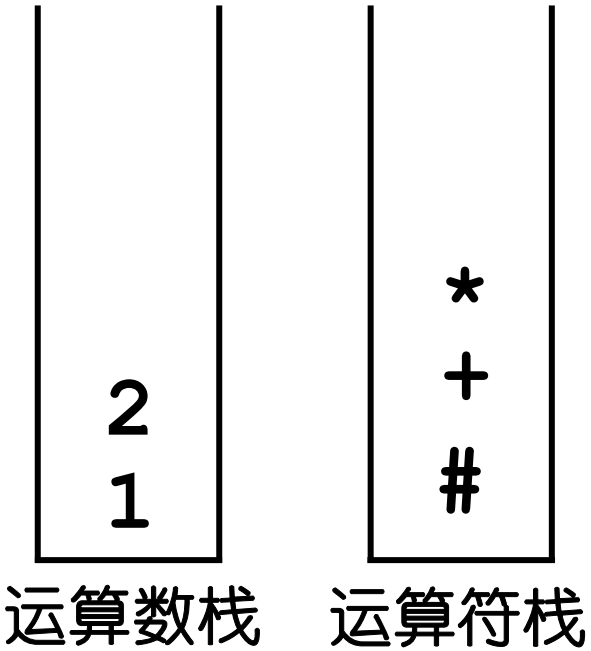




表达式求值：#3*(2+4)-8/2#的计算

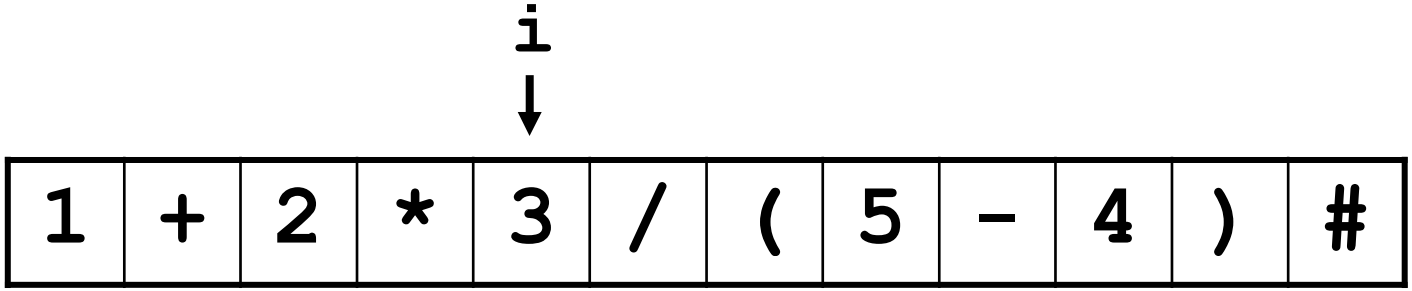
- 例

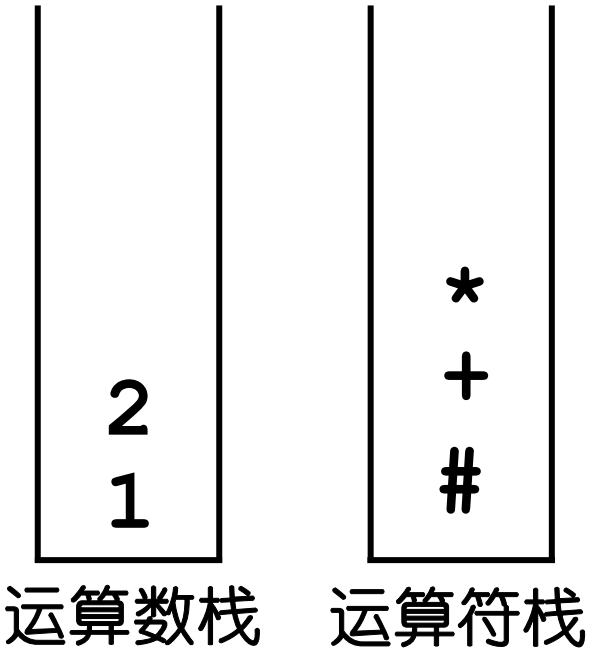




表达式求值：#3*(2+4)-8/2#的计算

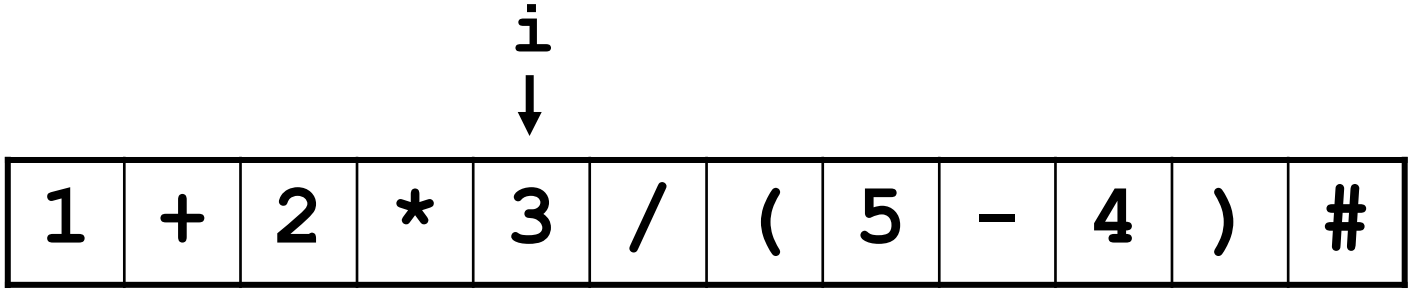
- 例

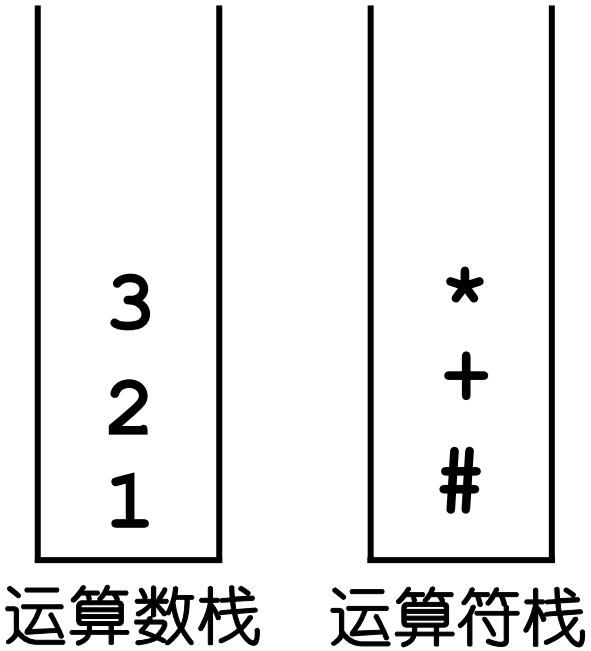




表达式求值：#3*(2+4)-8/2#的计算

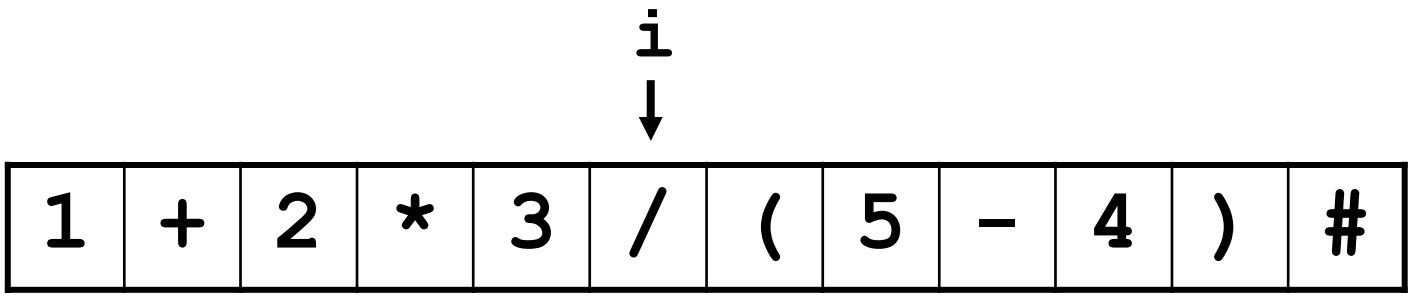
- 例

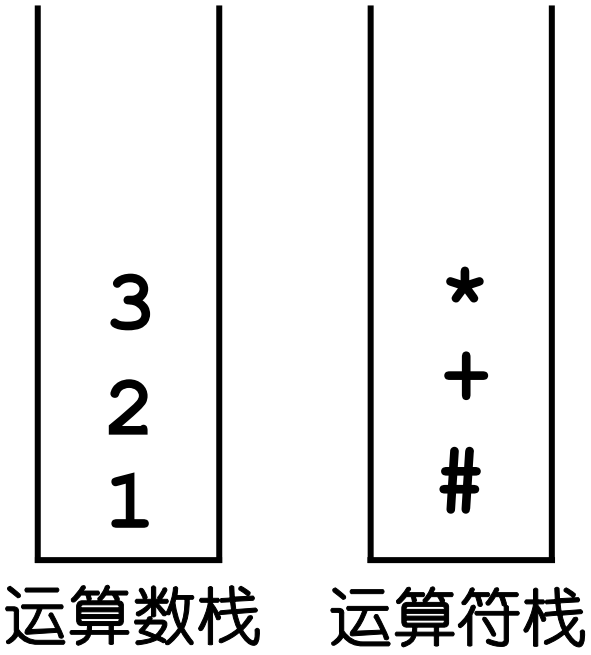




表达式求值：#3*(2+4)-8/2#的计算

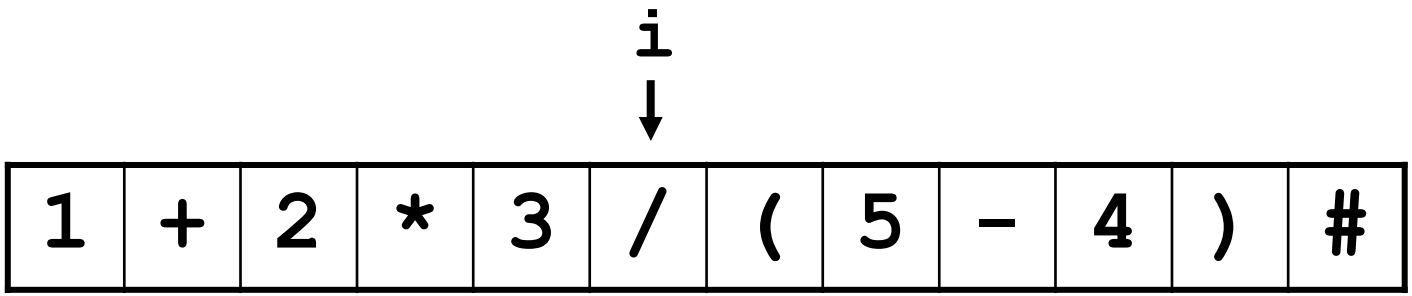
- 例

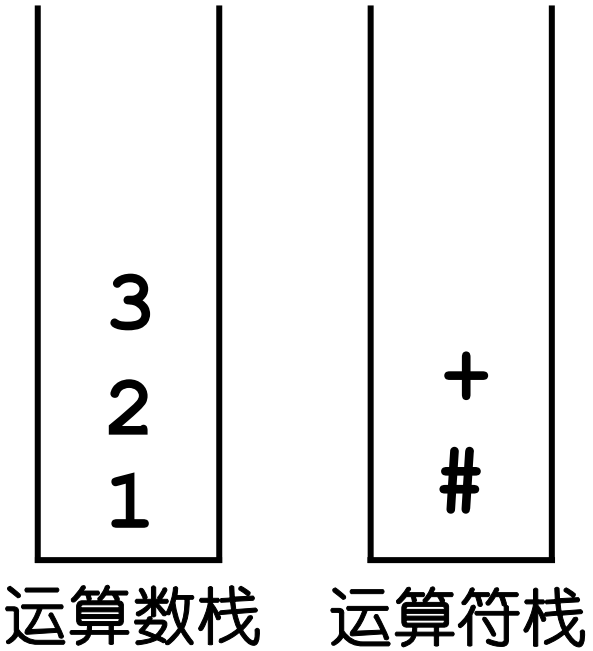




表达式求值：#3*(2+4)-8/2#的计算

- 例

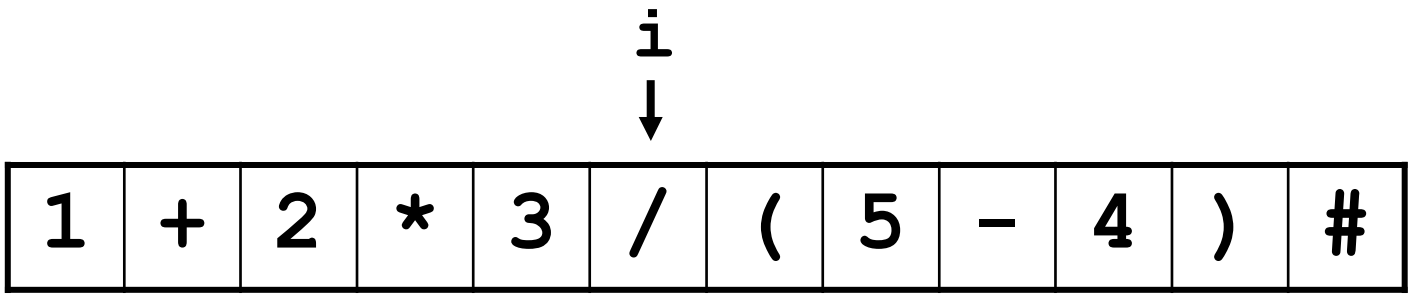


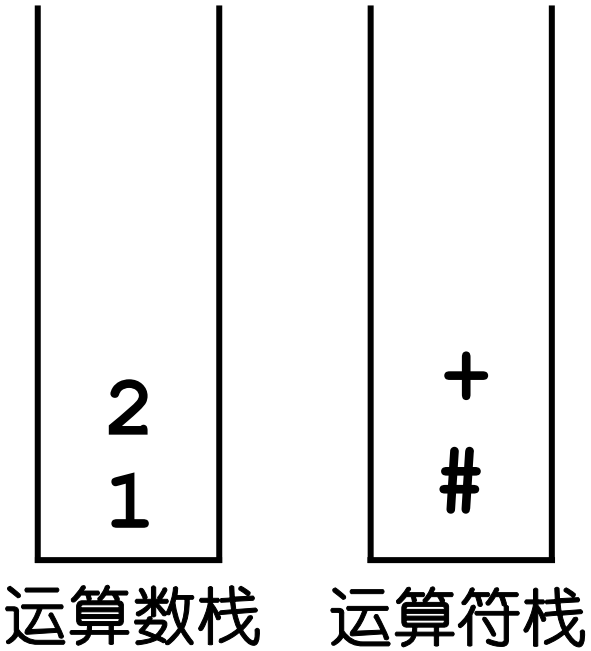


*

表达式求值：#3*(2+4)-8/2#的计算

- 例

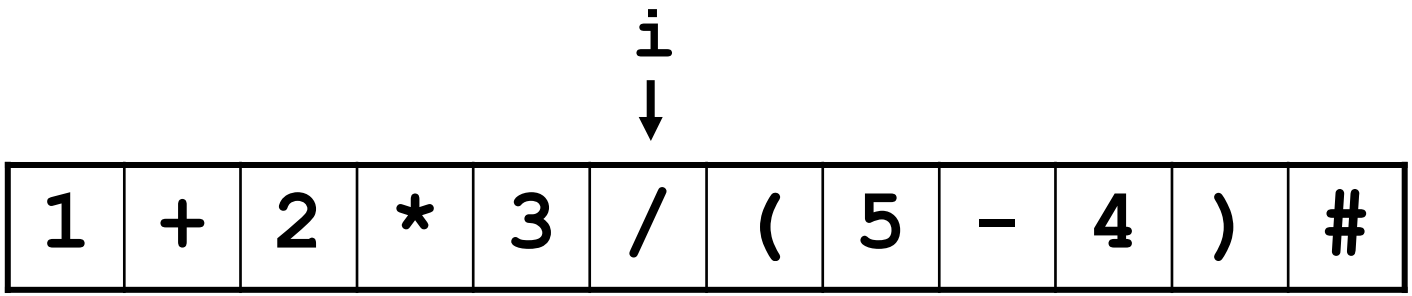


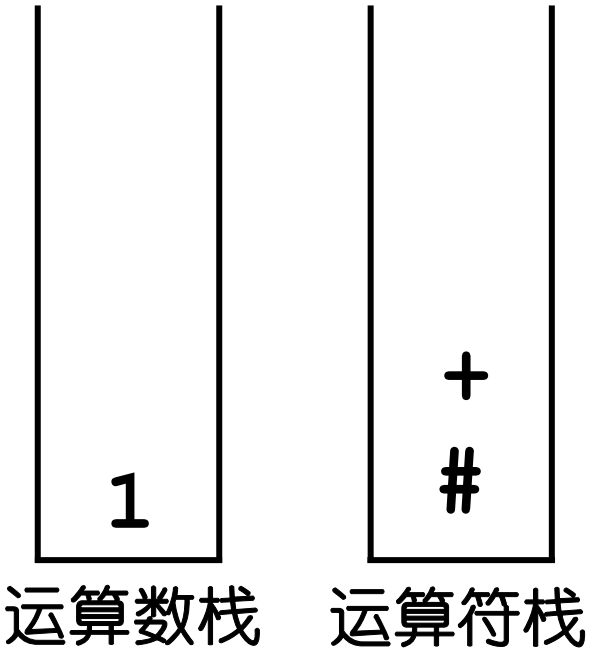


* 3

表达式求值：#3*(2+4)-8/2#的计算

- 例

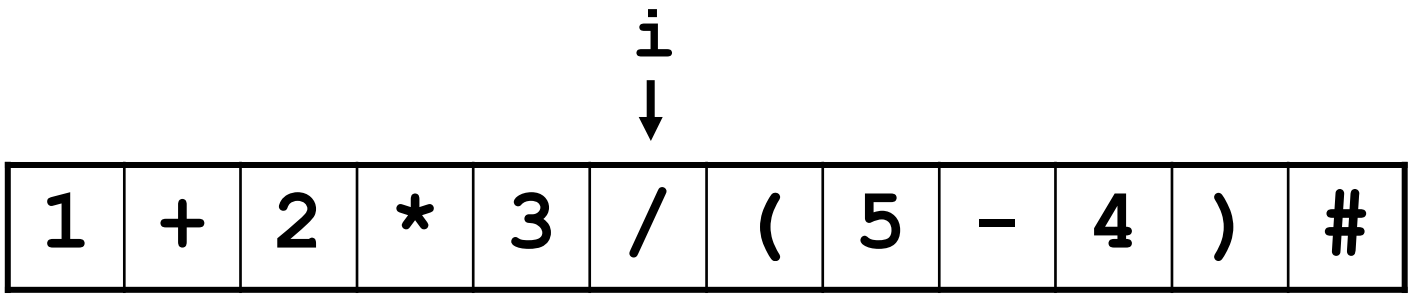


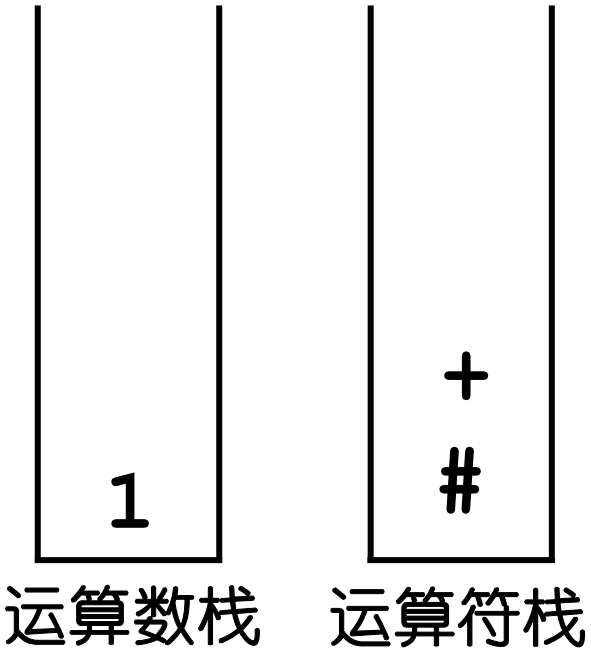


2 * 3

表达式求值：#3*(2+4)-8/2#的计算

- 例

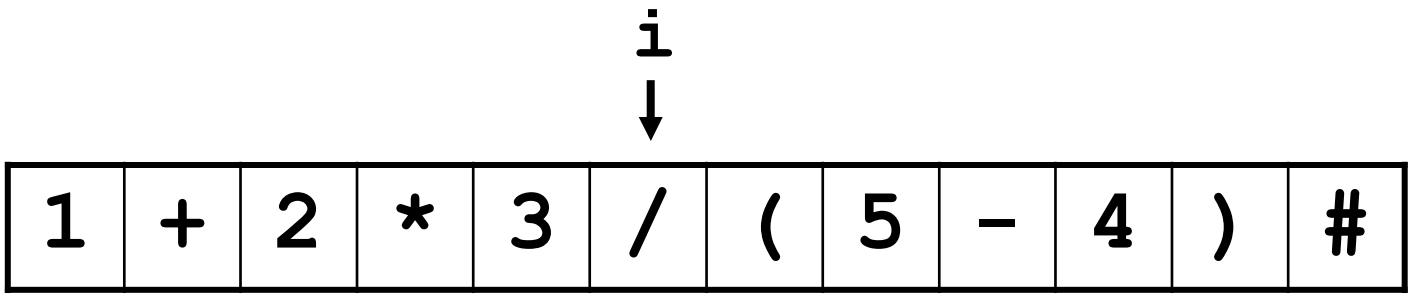


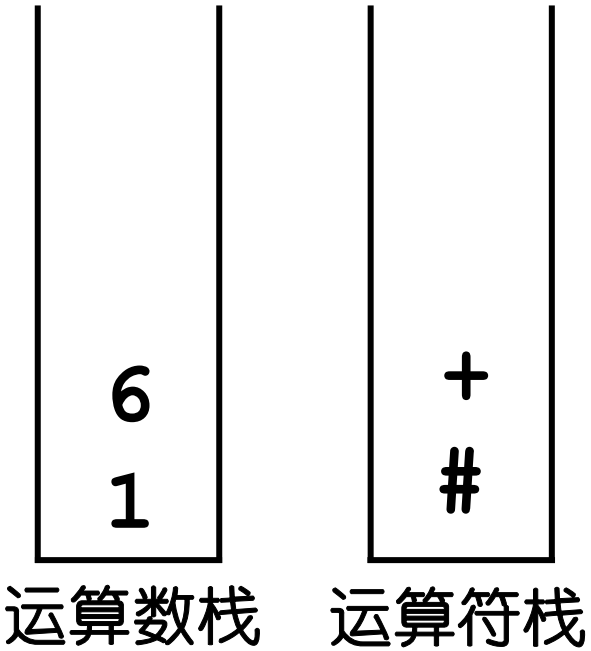


2 * 3 = 6

表达式求值：#3*(2+4)-8/2#的计算

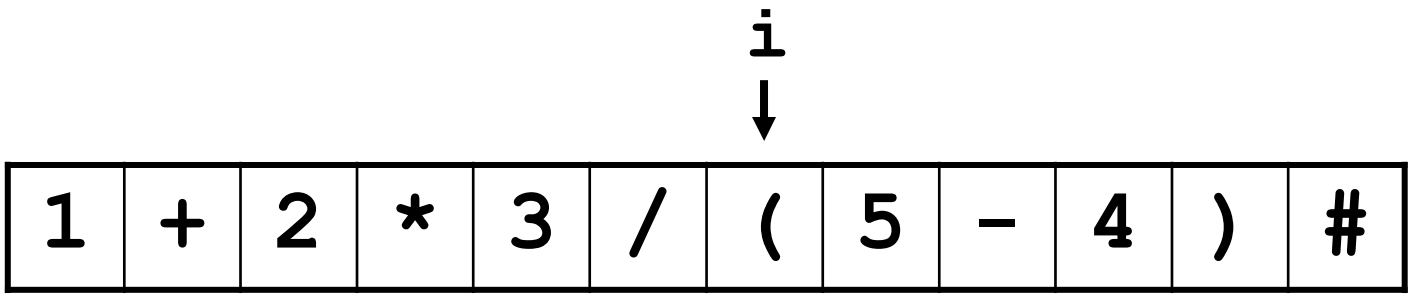
- 例

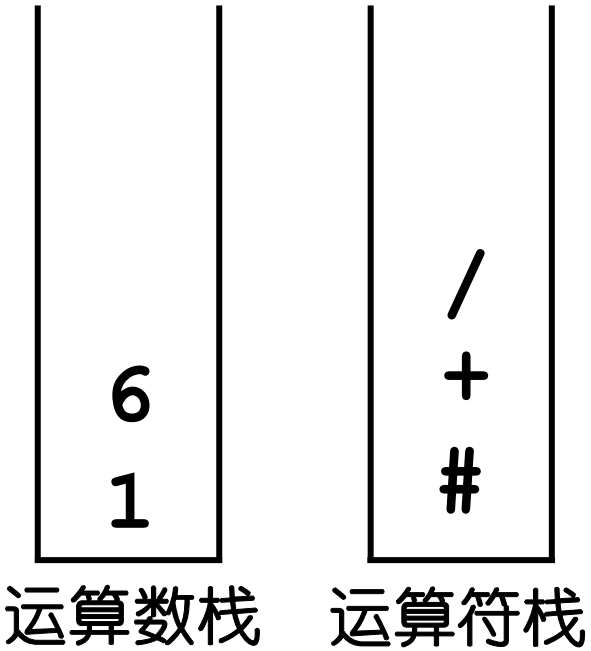




表达式求值：#3*(2+4)-8/2#的计算

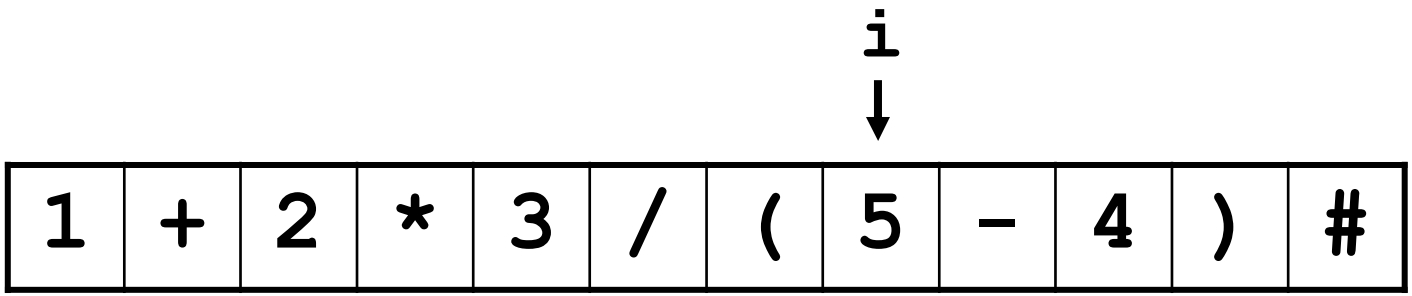
- 例

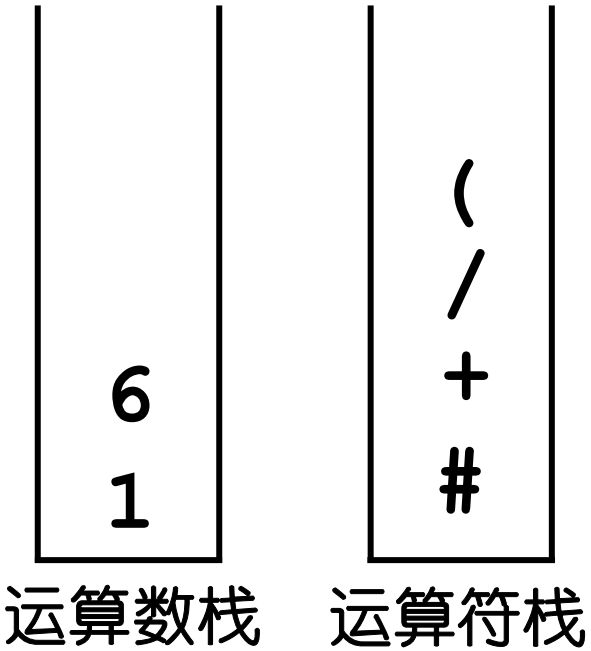




表达式求值：#3*(2+4)-8/2#的计算

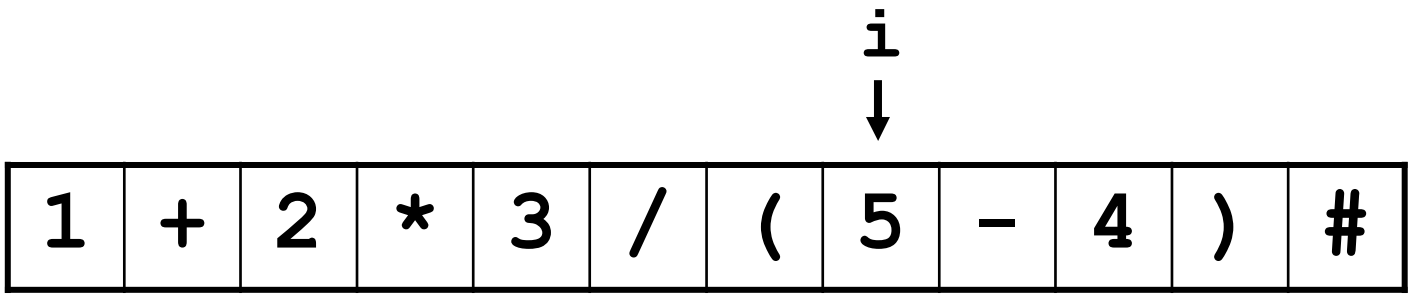
- 例

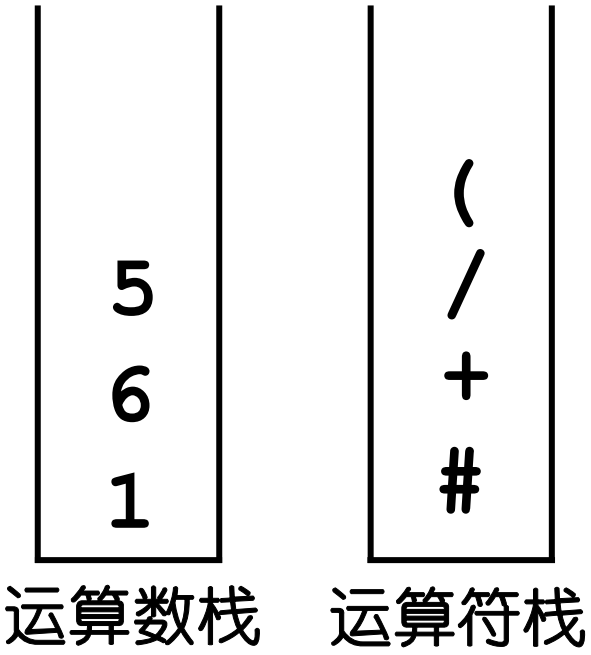




表达式求值：#3*(2+4)-8/2#的计算

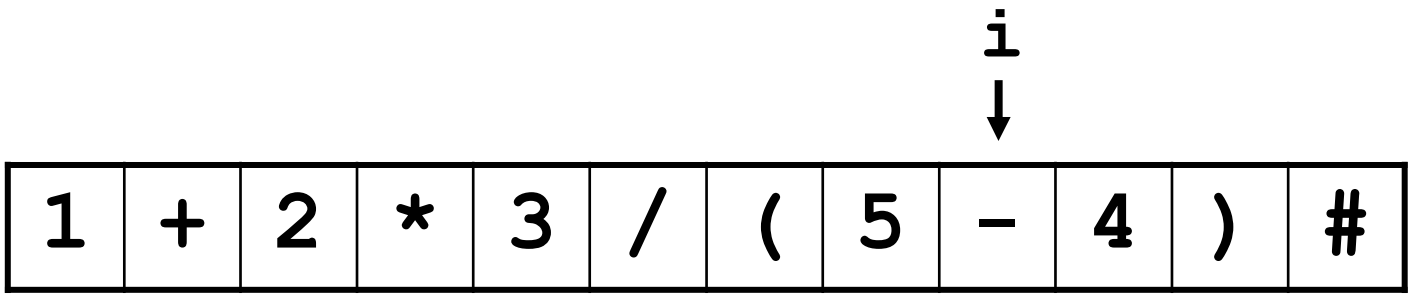
- 例

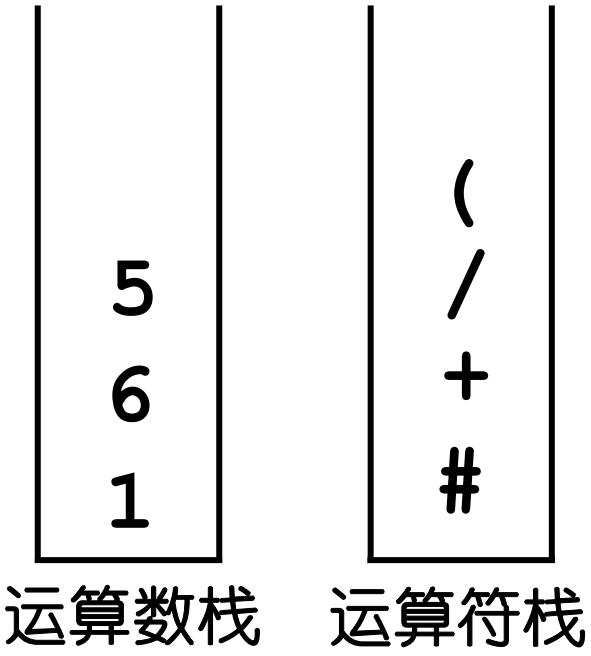




表达式求值：#3*(2+4)-8/2#的计算

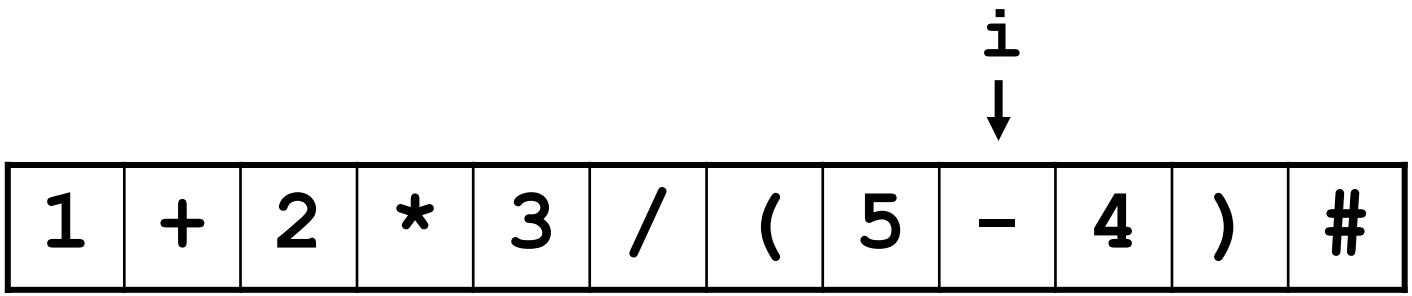
- 例

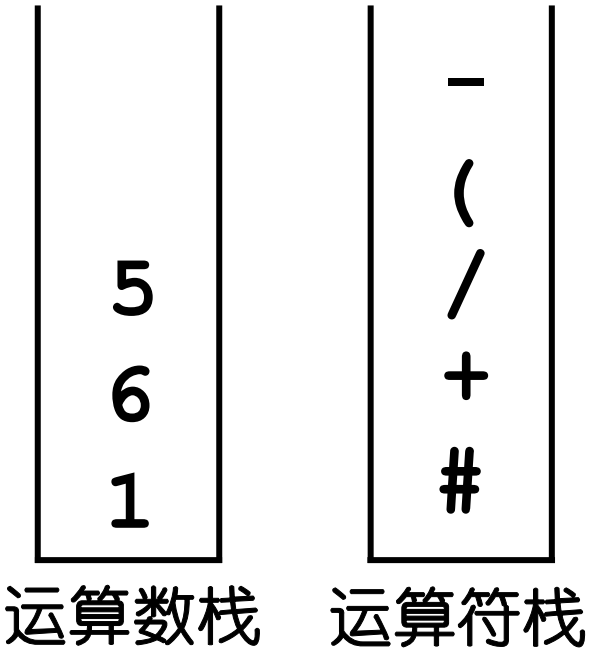




表达式求值：#3*(2+4)-8/2#的计算

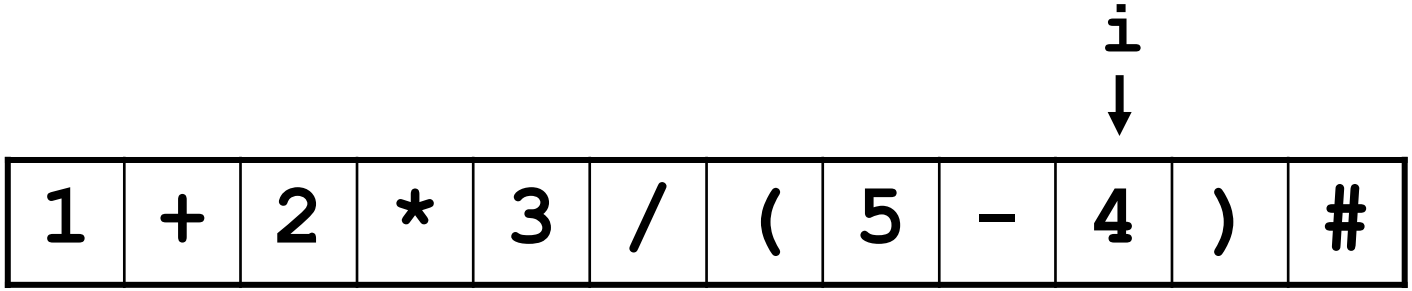
- 例

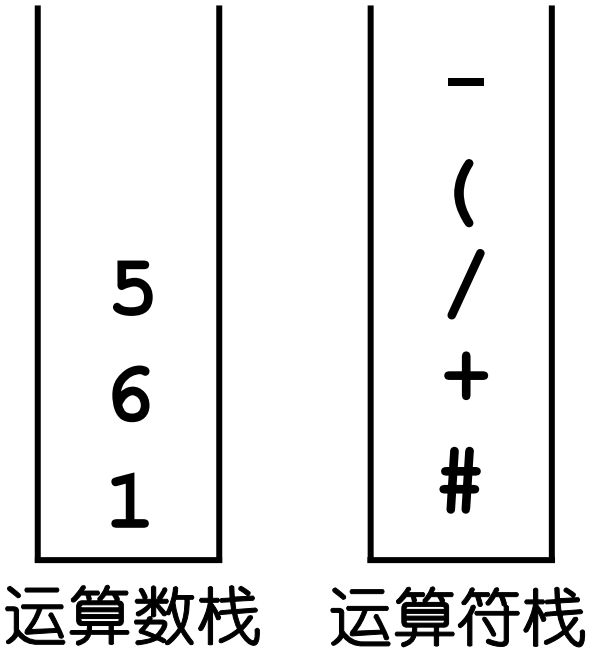




表达式求值：#3*(2+4)-8/2#的计算

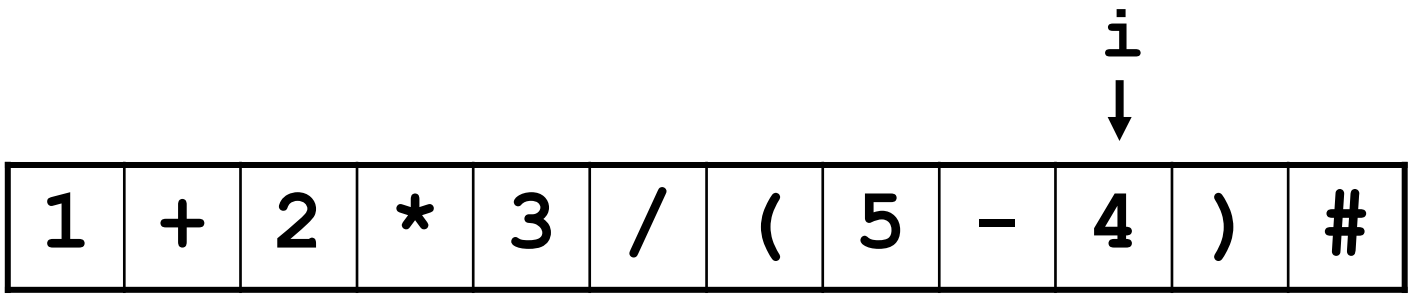
- 例

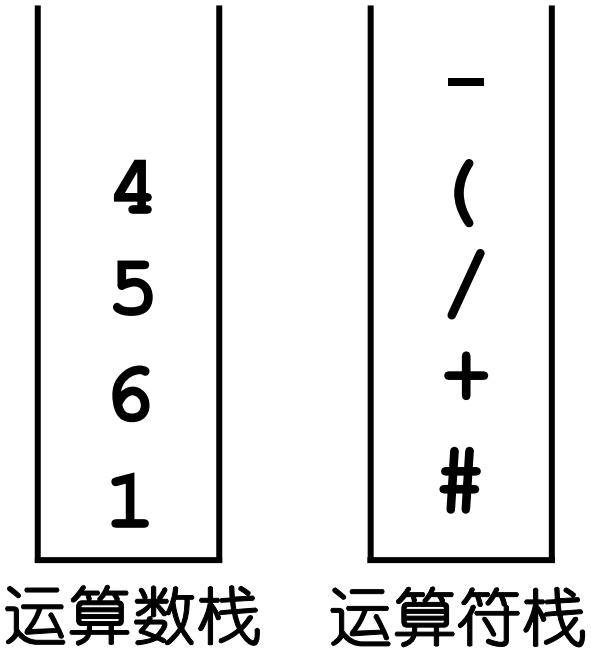




表达式求值：#3*(2+4)-8/2#的计算

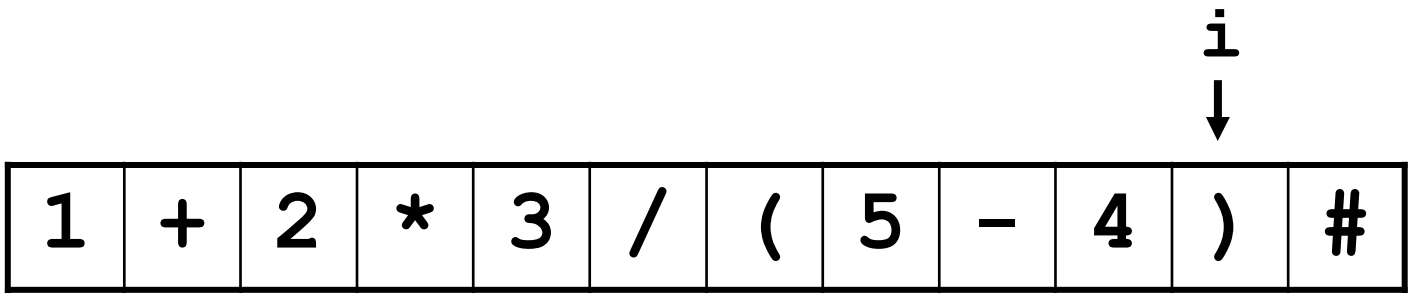
- 例

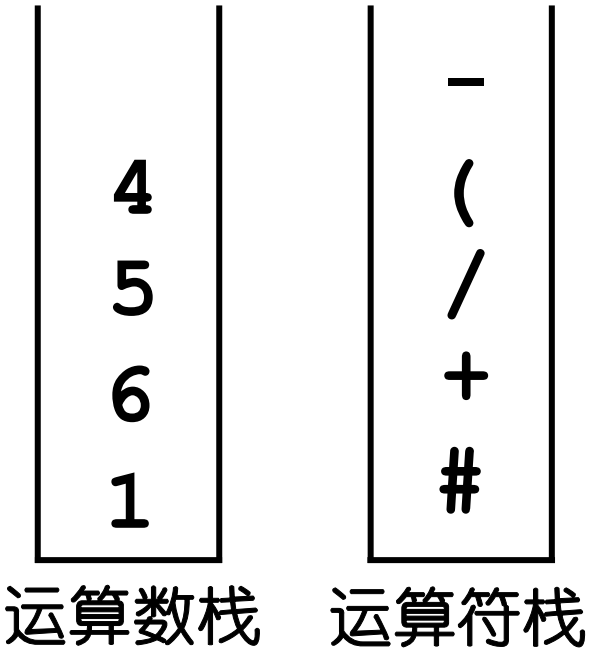




表达式求值：#3*(2+4)-8/2#的计算

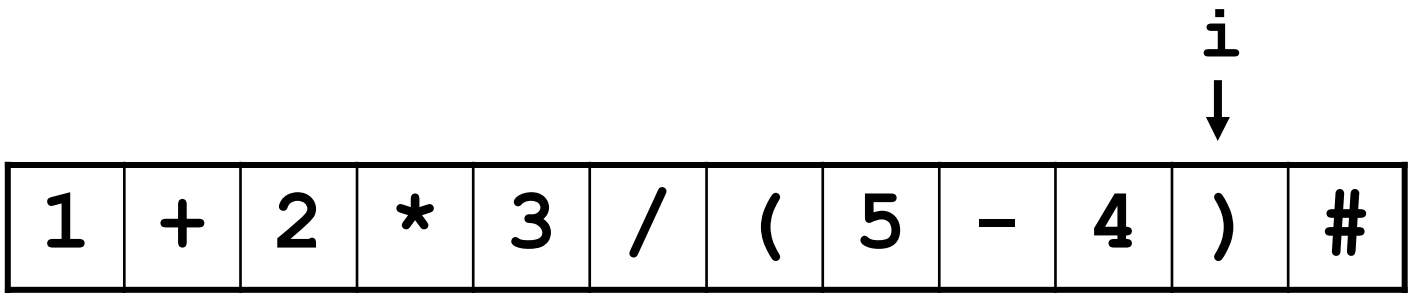
- 例

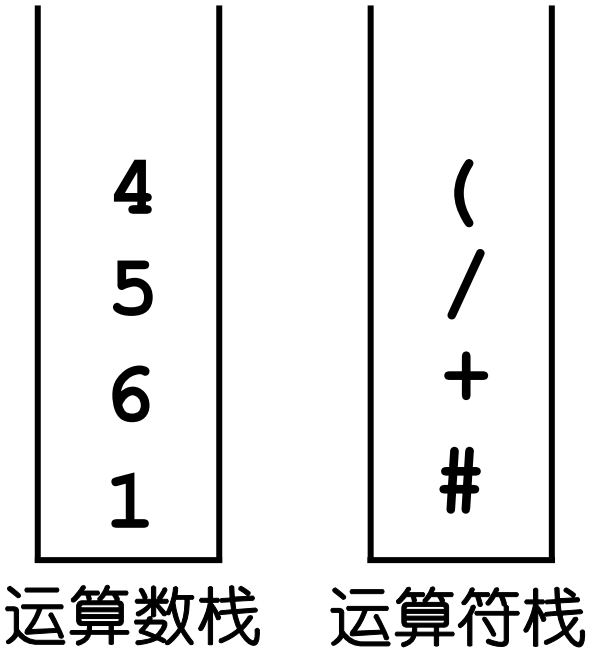




表达式求值：#3*(2+4)-8/2#的计算

- 例

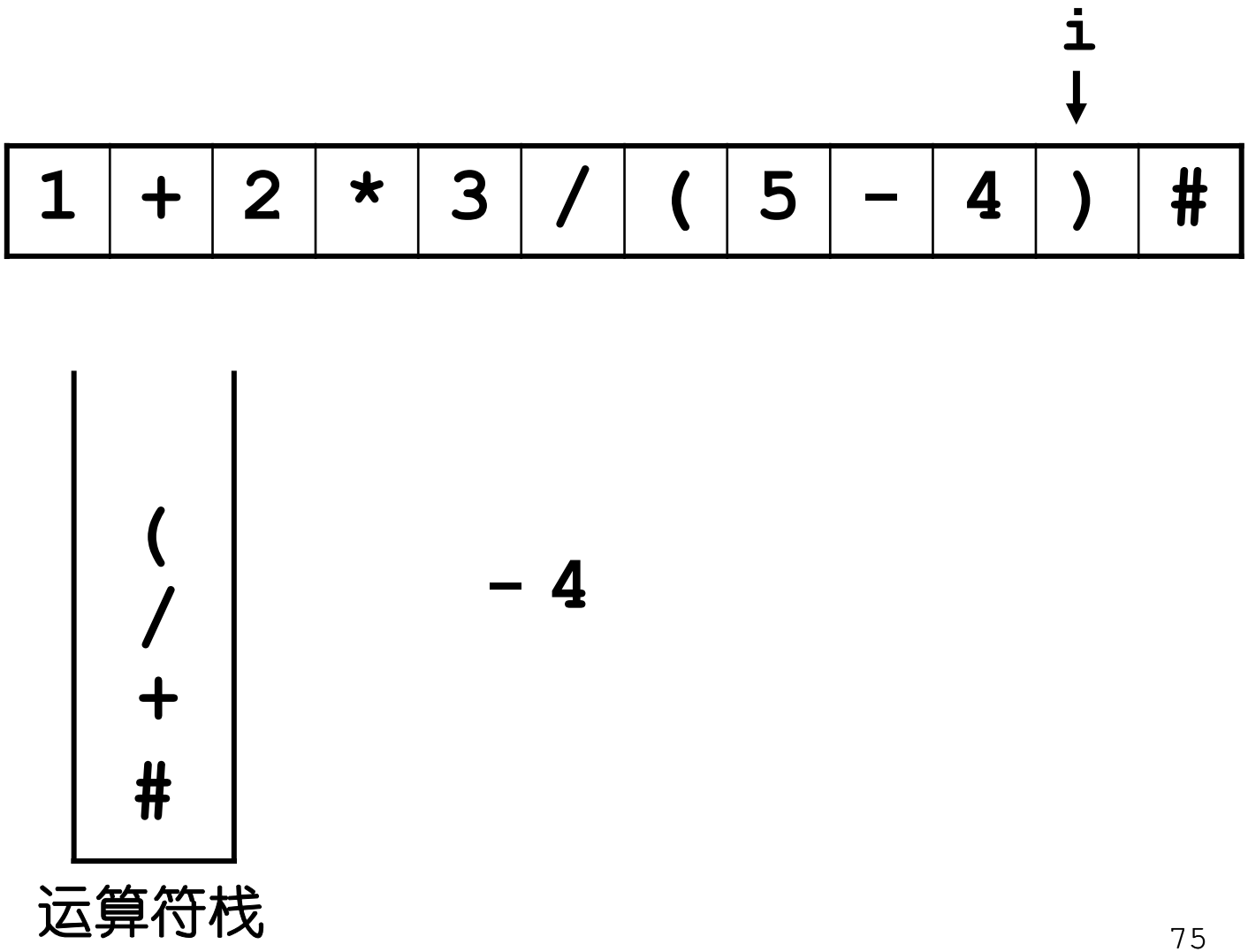




-

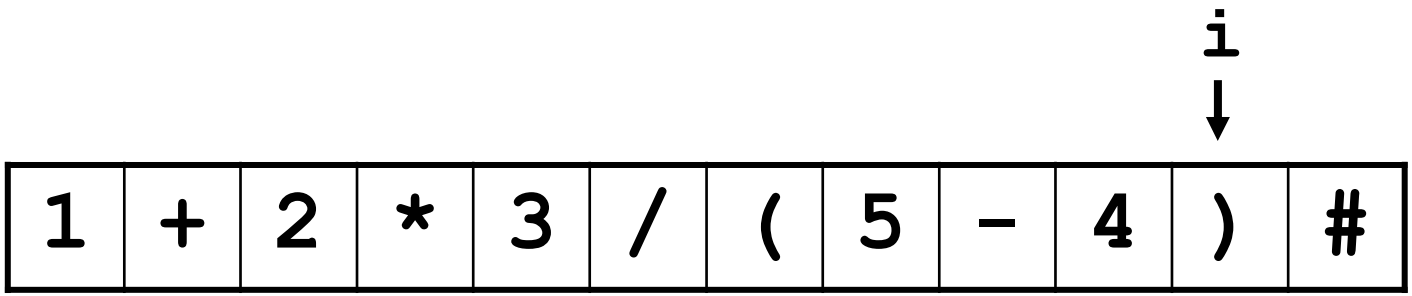
表达式求值：#3*(2+4)-8/2#的计算

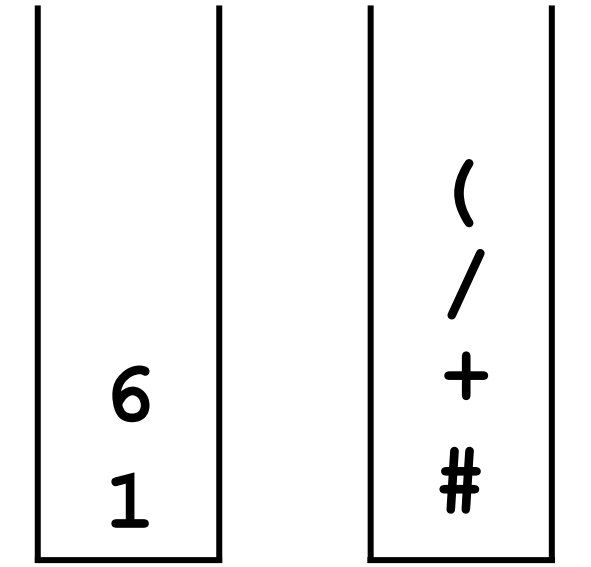
- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例

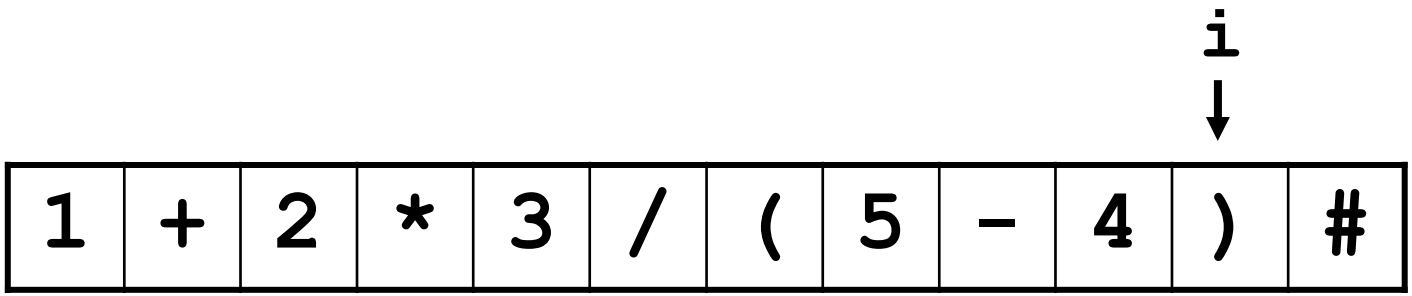


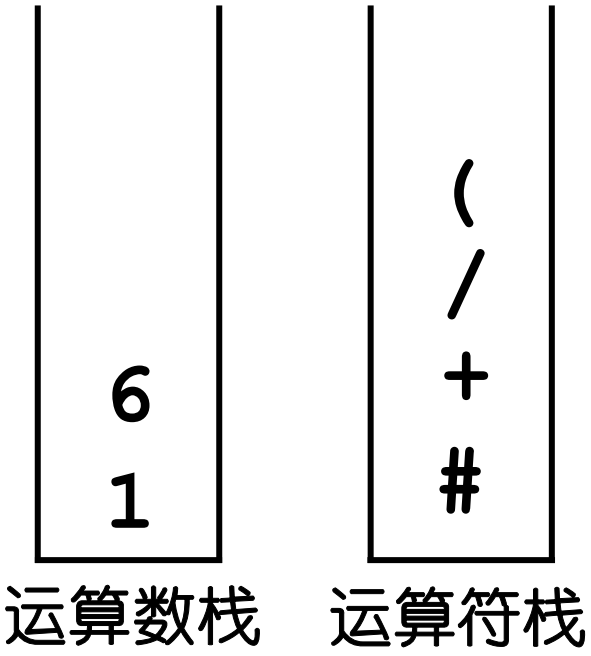


5 - 4

表达式求值：#3*(2+4)-8/2#的计算

- 例

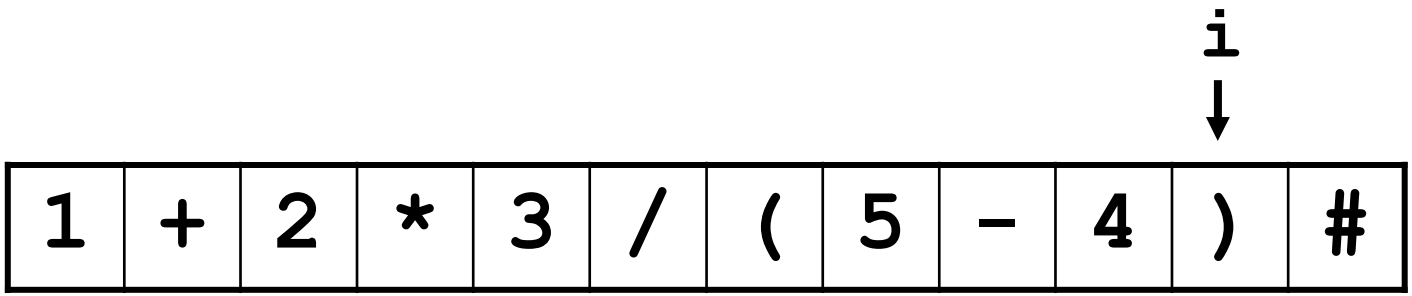


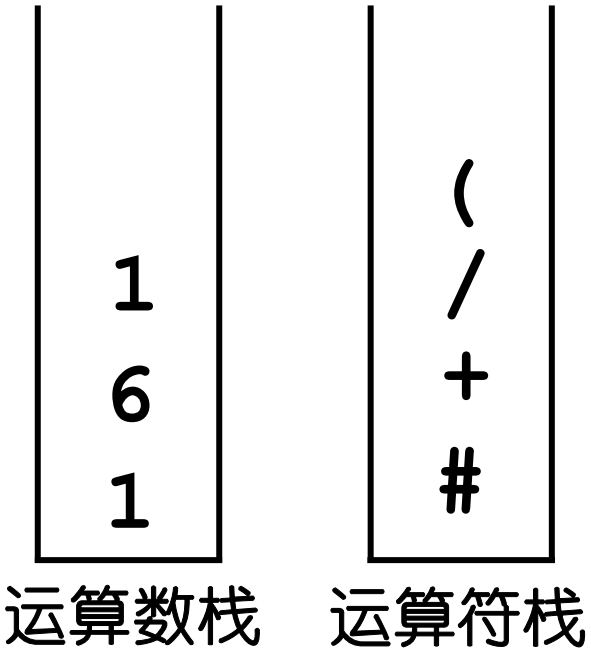


5 - 4 = 1

表达式求值：#3*(2+4)-8/2#的计算

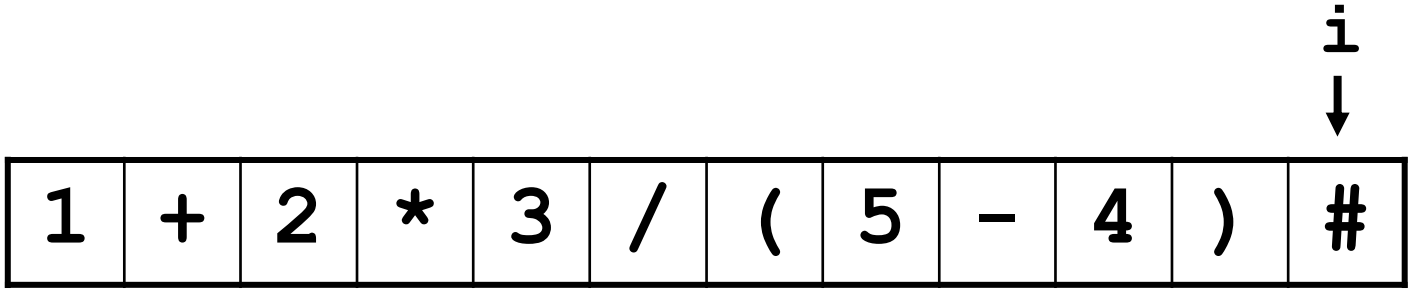
- 例

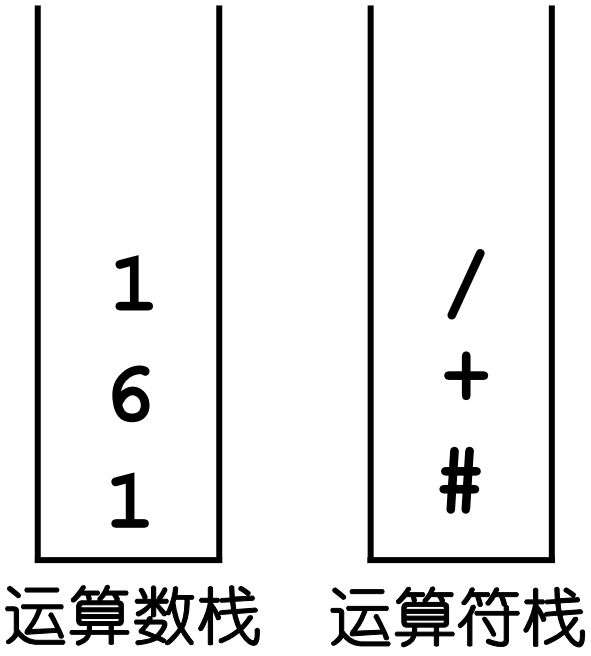




表达式求值：#3*(2+4)-8/2#的计算

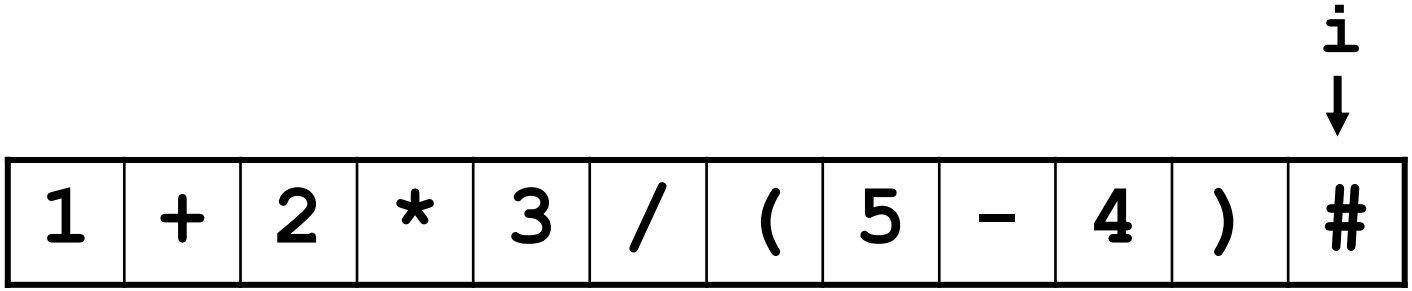
- 例

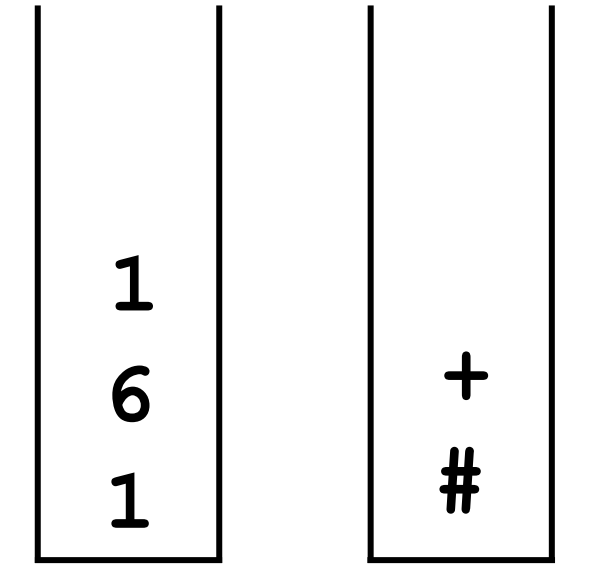




表达式求值：#3*(2+4)-8/2#的计算

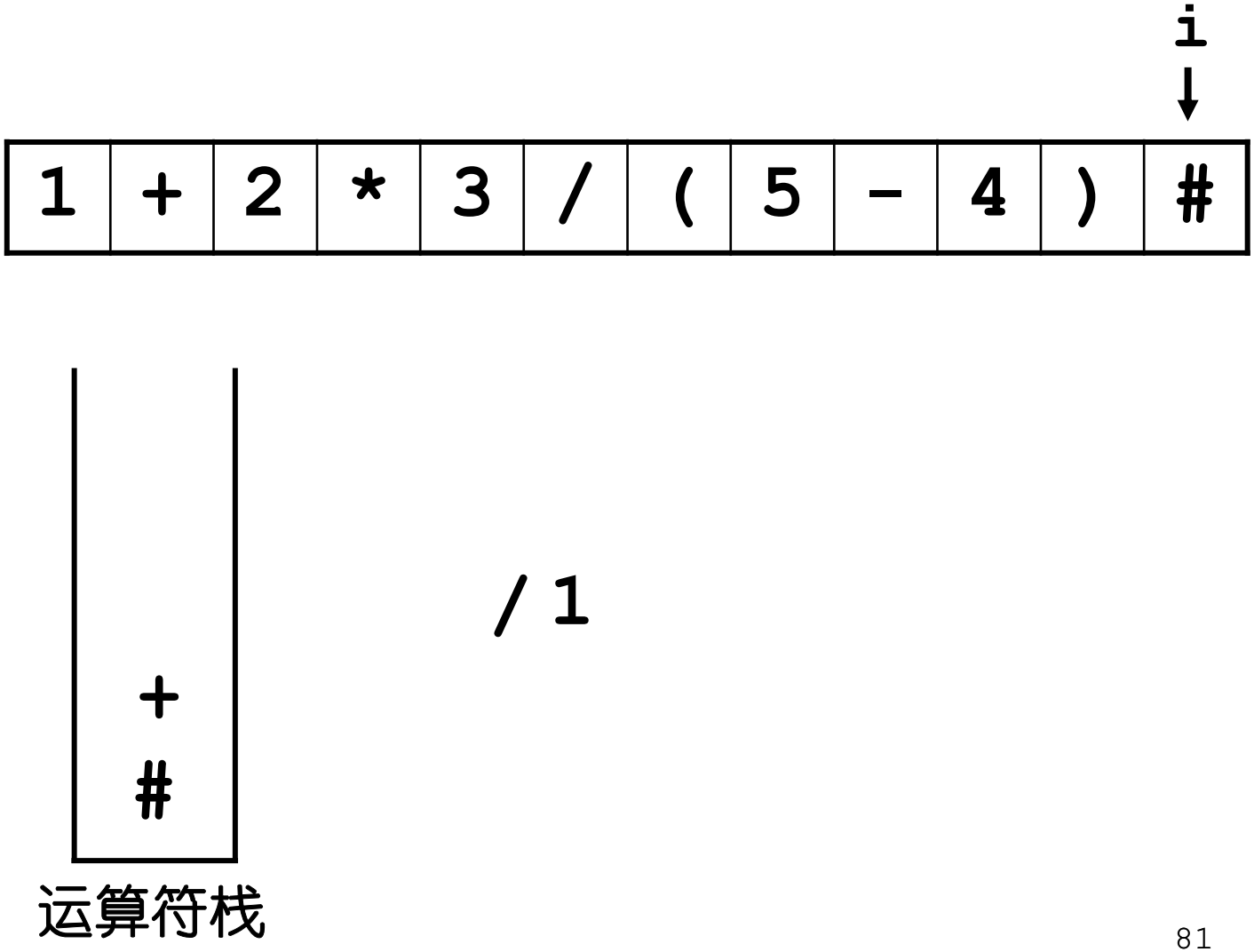
- 例





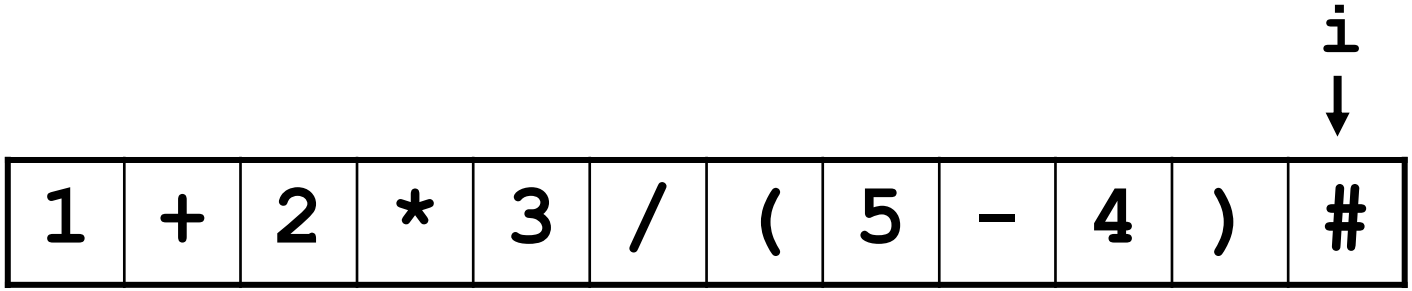
表达式求值：#3*(2+4)-8/2#的计算

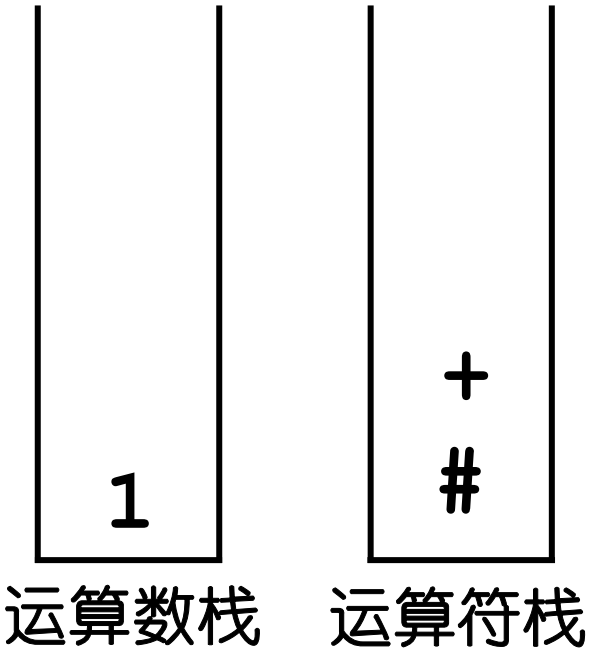
- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例

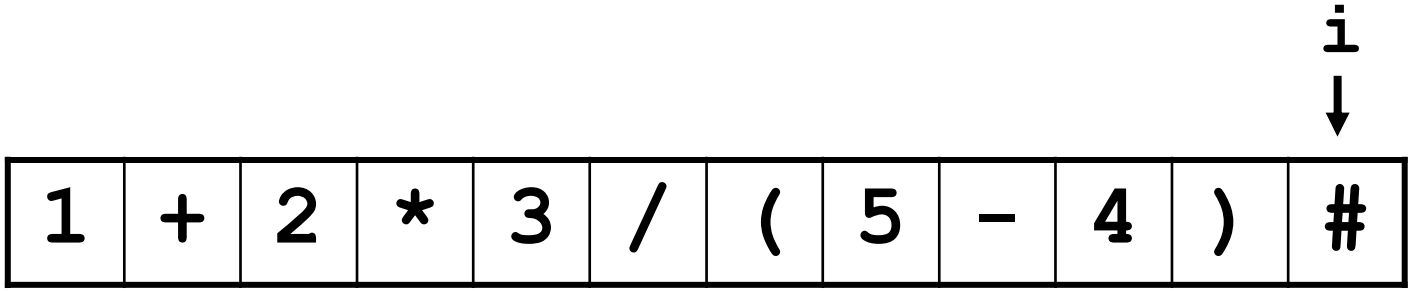


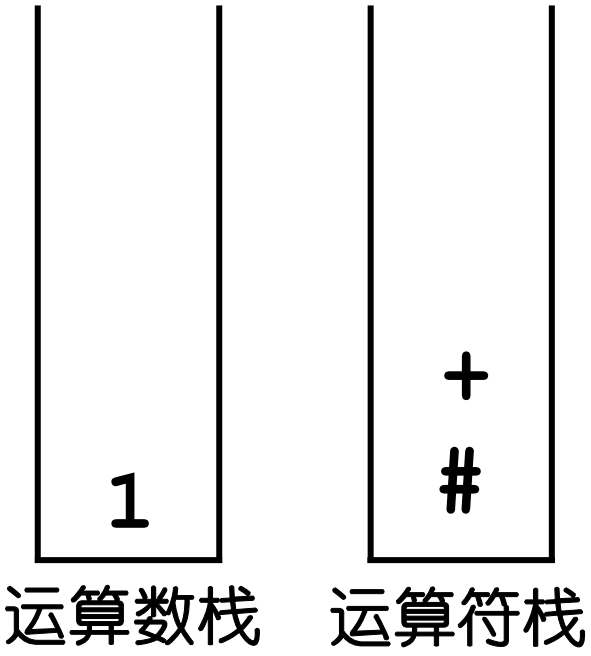


6 / 1

表达式求值：#3*(2+4)-8/2#的计算

- 例

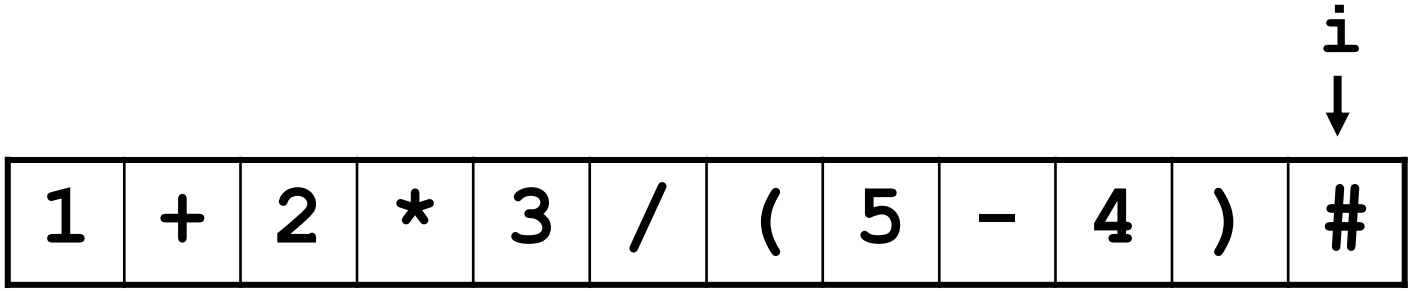


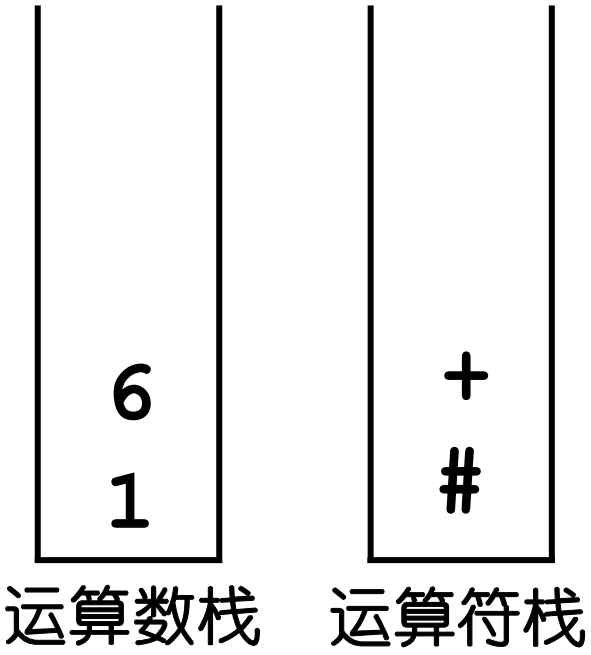


6 / 1 = 6

表达式求值：#3*(2+4)-8/2#的计算

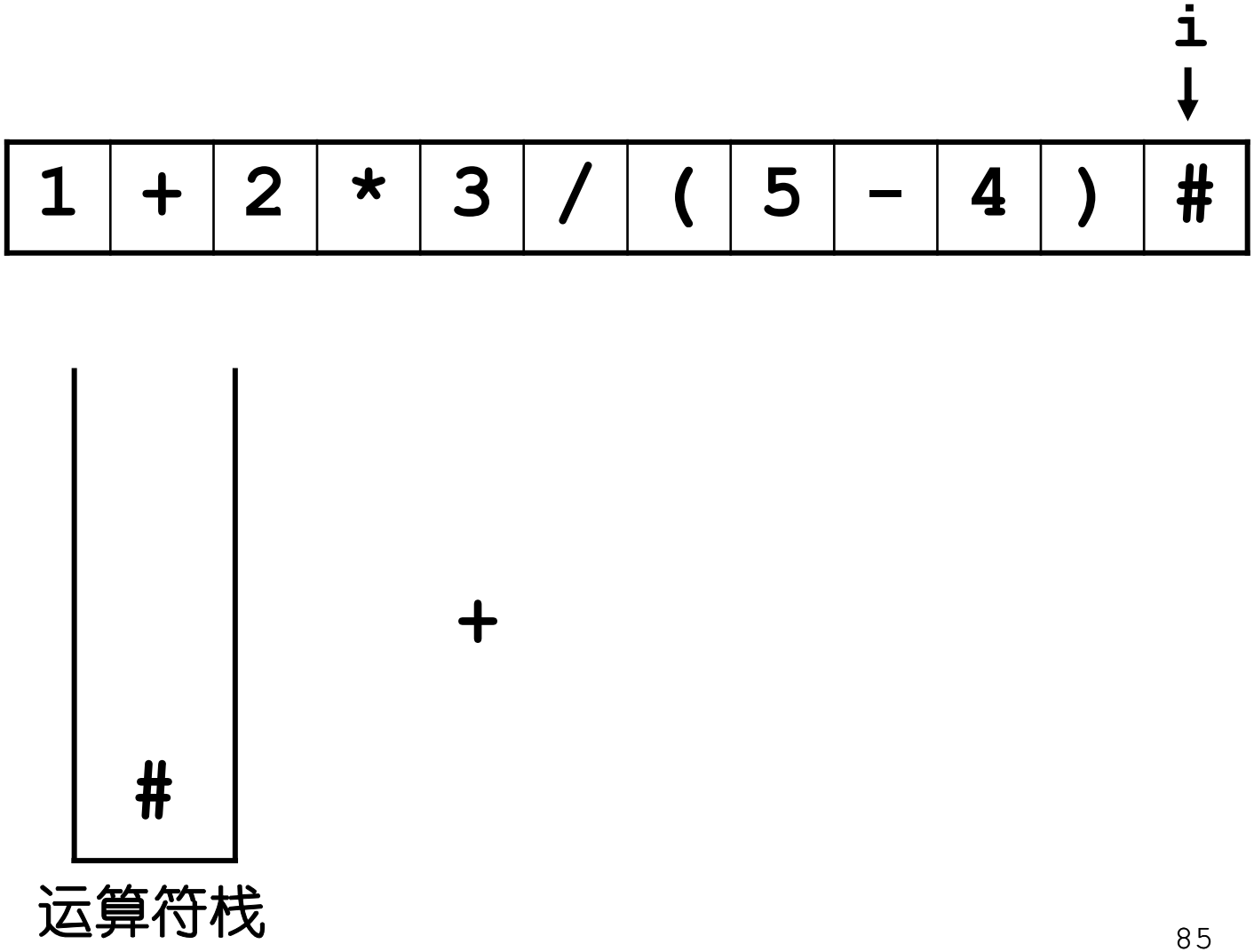
- 例





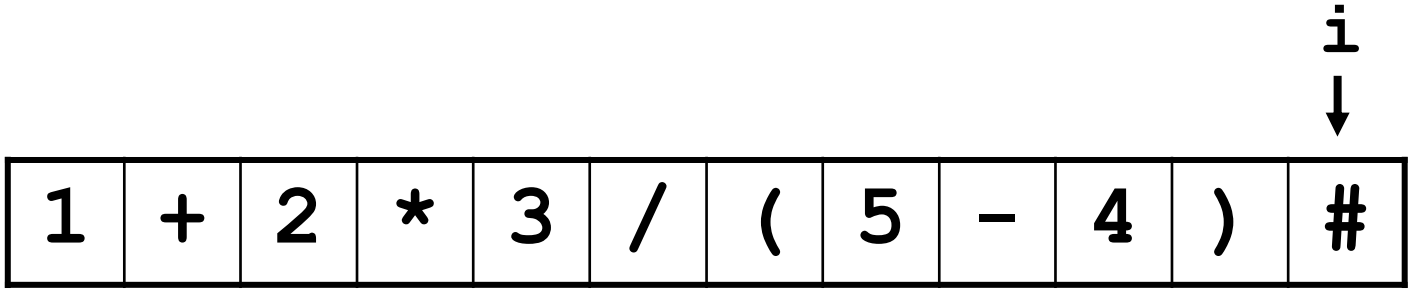
表达式求值：#3*(2+4)-8/2#的计算

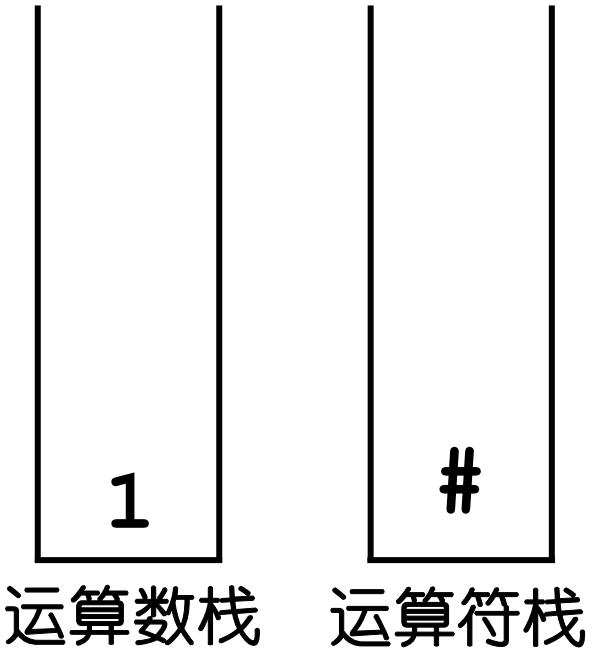
- 例



表达式求值：#3*(2+4)-8/2#的计算

- 例

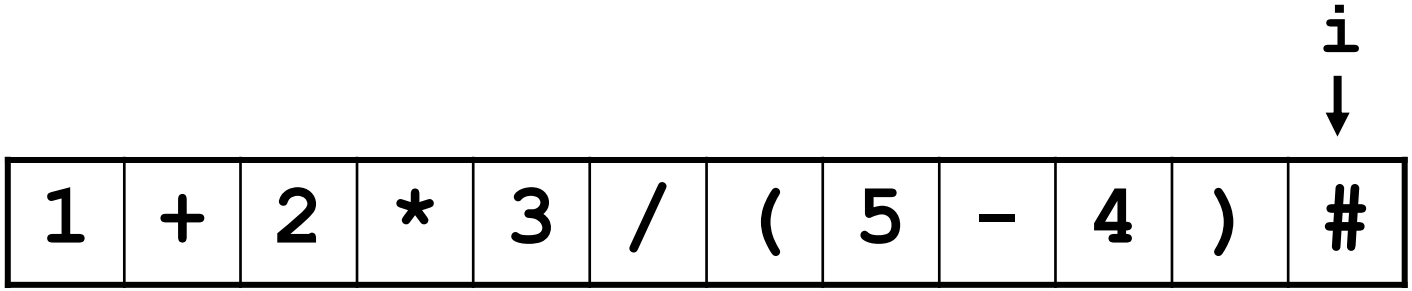


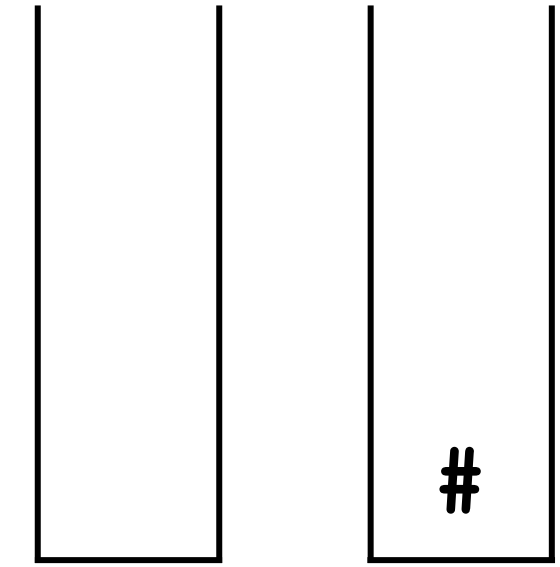


+ 6

表达式求值：#3*(2+4)-8/2#的计算

- 例

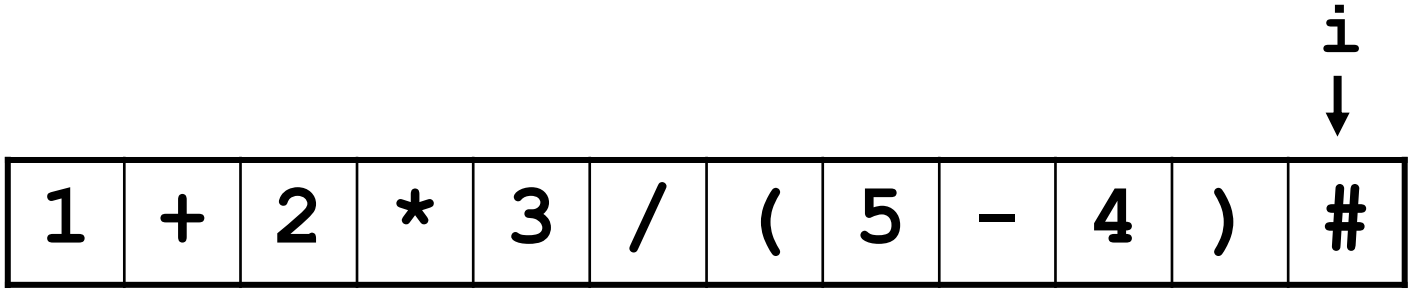


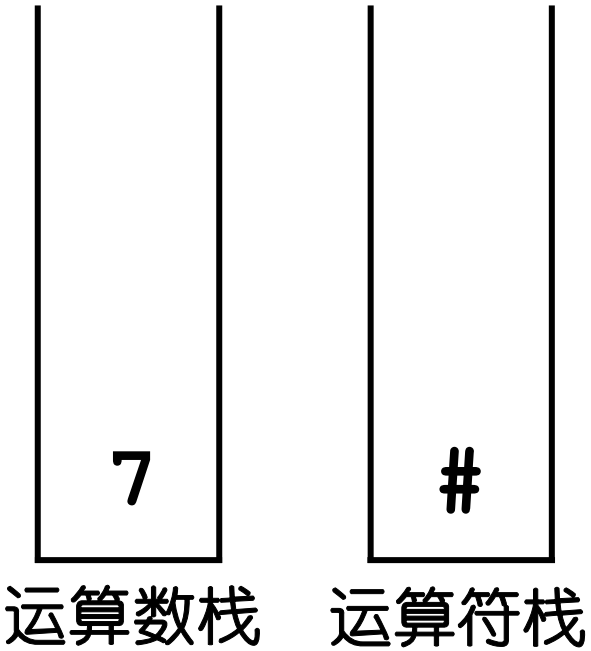


1 + 6 = 7

表达式求值：#3*(2+4)-8/2#的计算

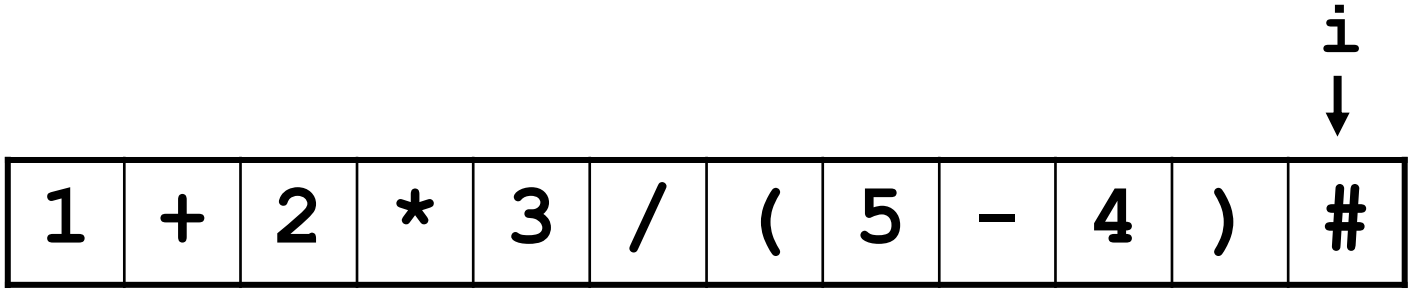
- 例

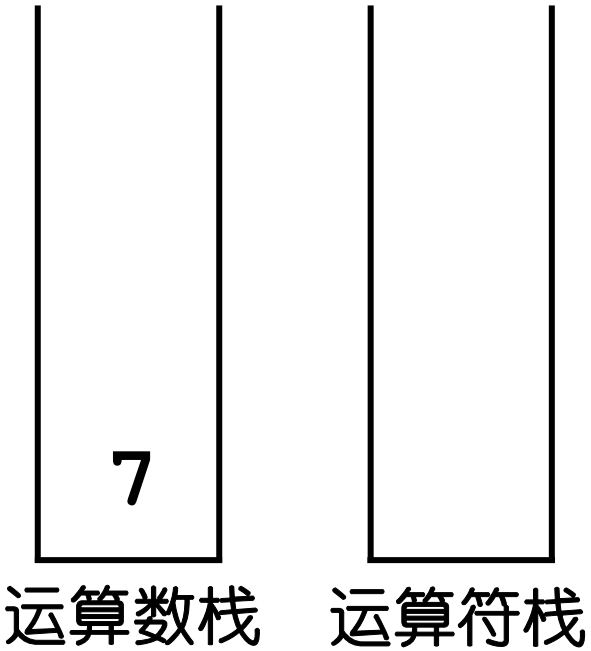




表达式求值：#3*(2+4)-8/2#的计算

- 例





表达式求值：算法过程

- 算法过程
  - 用两个栈分别存放运算符和运算数
  - 当前符号是运算数，直接入栈
  - 当前符号是运算符，且优先级更高，则入栈
  - 否则弹出栈顶元素运算，运算结果重新压入运算数栈

表达式求值：表达式的表示形式

- 表达式的表示
  - 中序表达式：运算符放在两个运算数中间
  - 前序表达式：运算符放在两个运算数之前
  - 后序表达式：运算符放在两个运算数之后
  - 例如：

中序	前序	后序
X+Y	+XY	XY+
X+Y*Z	+X*YZ	XYZ*+
(X+Y)*Z	*+XYZ	XY+Z*
a*(b/(c-d))+e	+*a/b-cde	abcd-/*e+

栈的应用：表达式的表示

- 前序表达式
  - 特点：
    - 运算符在运算数之前
    - 运算数顺序跟中序表达式相同
    - 运算符按照运算顺序的逆序排列

```
          3     2     1       4
中序: a * (b / (c - d)) + e

前序: + * a / b - cde
```

表达式求值：表达式的表示

- 后序表达式
  - 特点：
    - 运算符在运算数之前
    - 运算数顺序跟中序表达式相同
    - 运算符按照运算顺序排列

```
          3     2     1       4
中序: a * (b / (c - d)) + e

后序: abcd - / * e +
```

表达式求值：表达式的表示

- 手工计算方法：
  - 前序表达式
    - 取最后面的运算符
    - 取当前运算符后面的两个数作为运算数

    } 先找运算符，再找运算数
    - 运算结果放到原来运算符的位置上
    - 循环，直到所有的运算符都运算完毕
  - 后序表达式？同学们自己总结

*2+34
*27
14

234+*
27*
14

表达式求值：表达式的表示

- [练习] 计算下列表达式的值：
  - */+1*2+345-76

```
* / + 1 * 2 + 3 4 5 - 7 6
* / + 1 * 2 + 3 4 5 1
* / + 1 * 2 7 5 1
* / + 1 14 5 1
* / 15 5 1
* 3 1
```

表达式求值：表达式的表示

- 1234+*+5/76-*

1 2 3 4 + * + 5 / 7 6 - *

1 2 7 * + 5 / 7 6 - *

1 14 + 5 / 7 6 - *

15 5 / 7 6 - *

3 7 6 - *

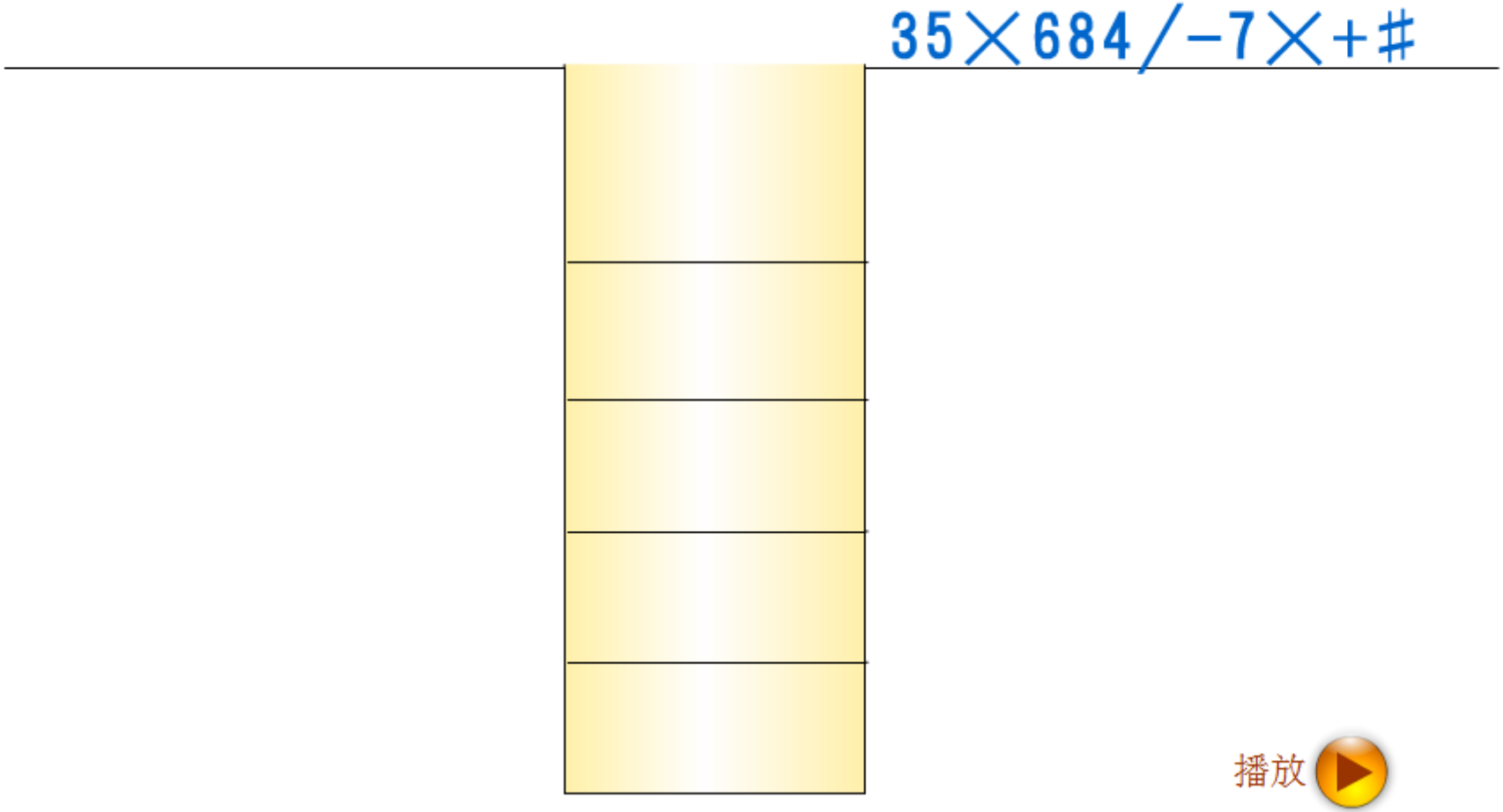
3 1 *

3

表达式求值：表达式的表示

- 后序表达式的计算机求解
  - 从左向右扫描每一个输入字符
  - 如果是运算数，入栈
  - 如果是运算符
    - 从栈中弹出所需的运算数
    - 运算
    - 结果压回栈
- 前序表达式与之类似

表达式求值：表达式的表示



表达式求值：表达式的转换

- 中序表达式-〉后序表达式
  - 运算数顺序不变
  - 运算数后的运算符的顺序可以由中序表达式的计算过程确定。

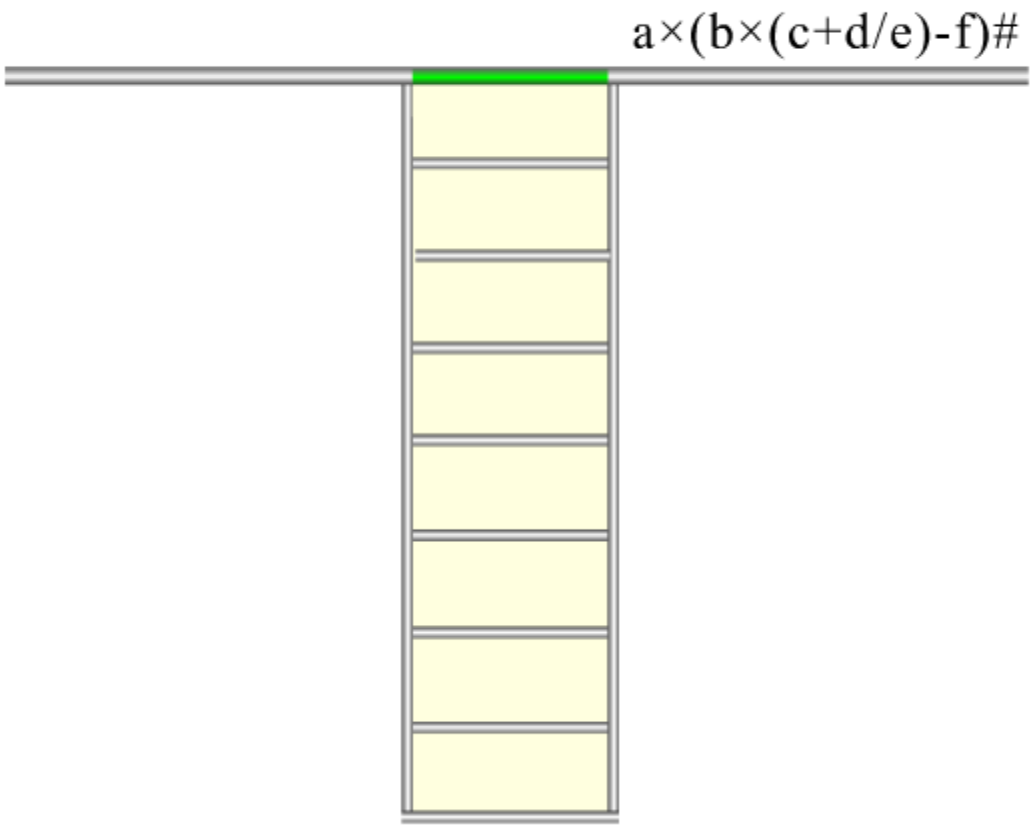
```
a*(b+c)
a*(bc+)
a*bc+
abc+*
```

栈的应用：表达式的转换

- 中序表达式的计算和中序转后序的区别：
  - 回顾中序表达式的计算算法：
    - 对于运算数：保持顺序不变
    - 对于运算符：
      - 优先于栈顶，则入栈
      - 栈顶运算符优先，计算该运算符
  - 因此类似的可以得出转换算法：
    - 对于运算数：直接输出
    - 对于运算符：
      - 优先于栈顶，则入栈
      - 栈顶运算符优先，输出该运算符

栈的应用：表达式的转换

表达式“a×(b×(c+d/e)-f)#”转换
成后缀式的演算过程如下所示：

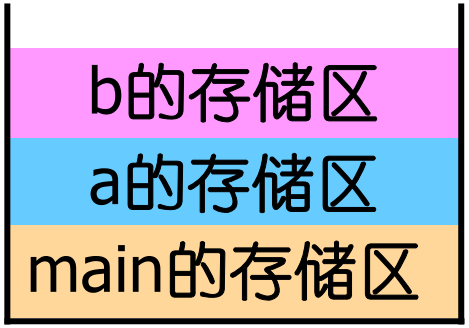


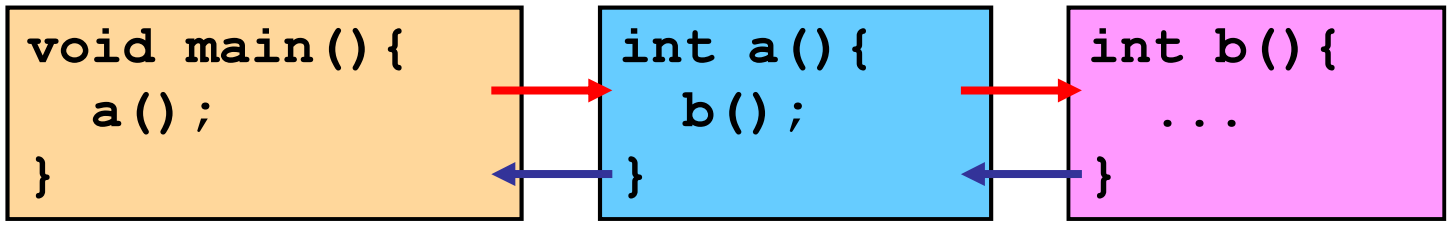
栈和递归的实现

- 函数调用与运行栈
  - 当一个函数在运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：
    - 将所有的参数、返回地址等信息传递给被调用函数
    - 为被调用函数的局部变量分配存储区
    - 将控制转移到被调用函数的入口
  - 而从被调用函数返回之前，应该完成：
    - 保存被调函数的计算结果
    - 释放被调函数的数据区
    - 依照被调函数保存的返回地址将控制转移到调用函数

栈和递归的实现

- 运行栈
  - 当多个函数嵌套调用时，由于函数的运行规则是：后调用先返回
  - 因此函数存储的管理通常实行“栈式管理”





栈和递归的实现

- 递归调用
  - 一个递归函数的运行过程类似于多个函数的嵌套调用
  - 差别仅仅在于“调用函数和被调用函数是同一个函数”
  - 运行栈中保存的都是同一个函数不同次调用时的信息

递归算法的应用

- 什么时候考虑用递归算法
  - 数据结构是递归定义的
    - 典型的比如二叉树
  - 问题直接递归定义的
    - 比如求阶乘、求Fibonacci级数等
  - 解题思路包含有递归规律的
    - 有一些问题并不是直接用递归定义的
    - 但是仔细分析可以发现其中有递归的规律
    - 用递归程序可以很简单的解决

递归算法的应用

- 编写递归程序的要点
  - (1) 把部分看成整体
    - 把整体的解决分成若干部分
    - 每个部分的解决方法和整体相同
    - 解决整体时假设部分已经解决
  - (2) 注意留递归出口
    - 如果不留会怎么样？

递归算法的应用

- 例1：求阶乘
  - 阶乘的定义如下

$$Fact(n)=\begin{cases}1 & n=1\\ n\times Fact(n-1) & n>1\end{cases}$$

  - 很显然是递归定义的，可以用递归程序解决

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

递归算法的应用

- 例2：求最大公约数
  - 算法：辗转相除法

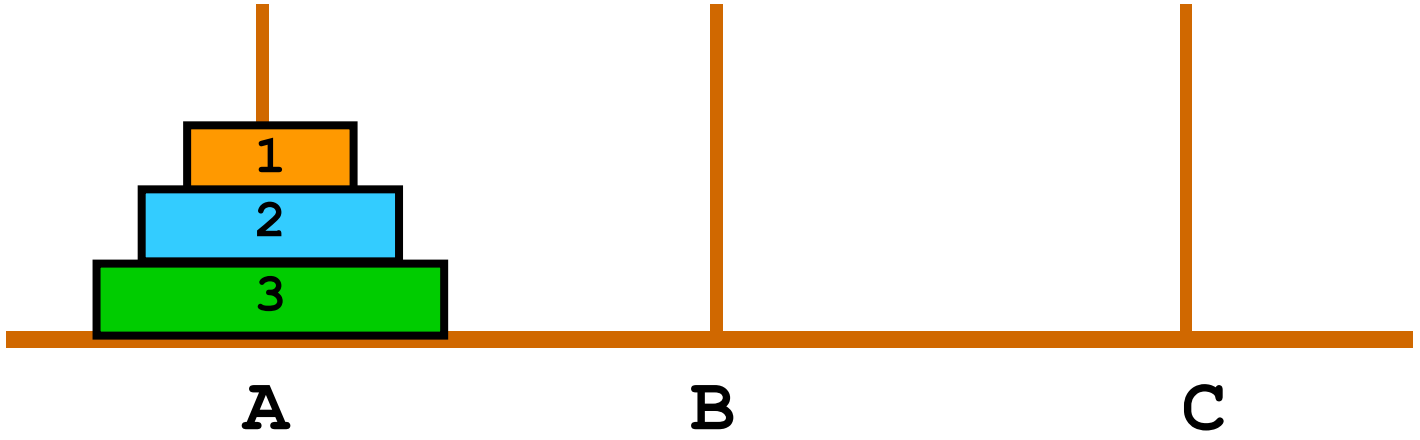
$$GCD(M, N) = \begin{cases} M & N = 0 \\ GCD(N, M\%N) & N > 0 \end{cases}$$

  - 比如：

```
GCD(72,27)  = GCD(27,18)
            = GCD(18,9)
            = GCD(9,0)
            = 9
```

递归算法的应用

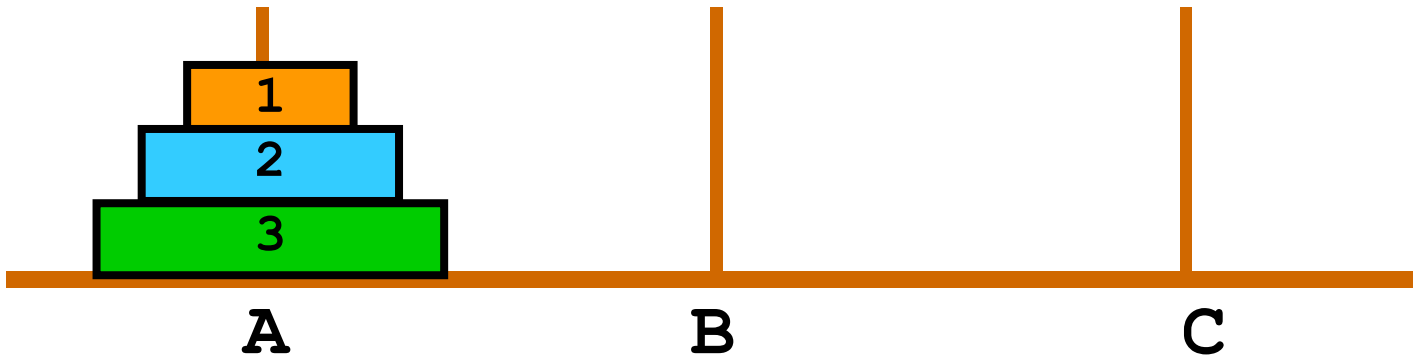
- 例3：汉诺塔



  - 每次只允许移动一个盘子
  - 必须保证小盘子在大盘子之上
  - 如何把所有的盘子从A移到C？

递归算法的应用

- 这个问题本身看不出有递归的特点
- 但是解决方法却可以采用递归的策略：
  - 我们把1、2看成一个整体，假设能够把1、2移到B（至于怎么移动这个整体，以后再说）
  - 这时3上面没有盘子，可以直接把3移到C
  - 最后再把1、2移到C



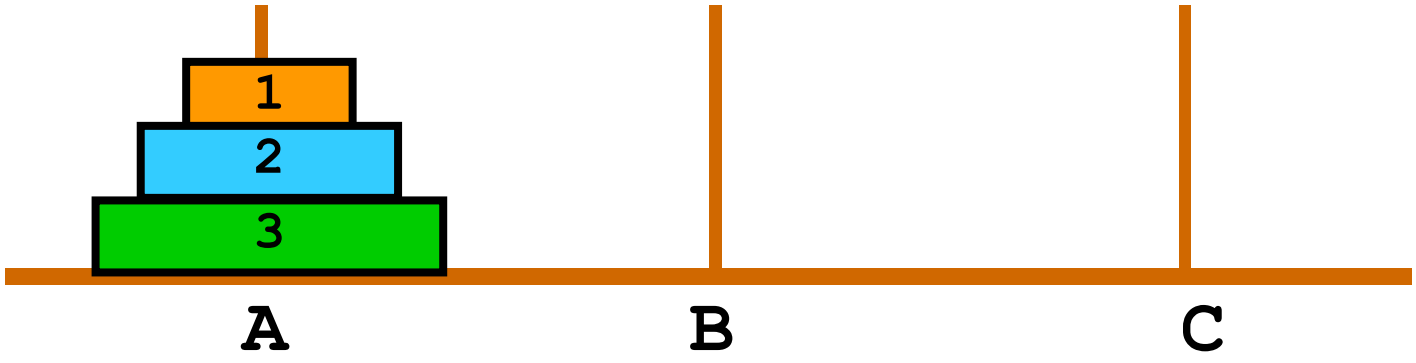
递归算法的应用

这样移动n个盘子的问题就简化为：

(1) 用C柱做过渡，将A柱上的n-1个盘子移到B上

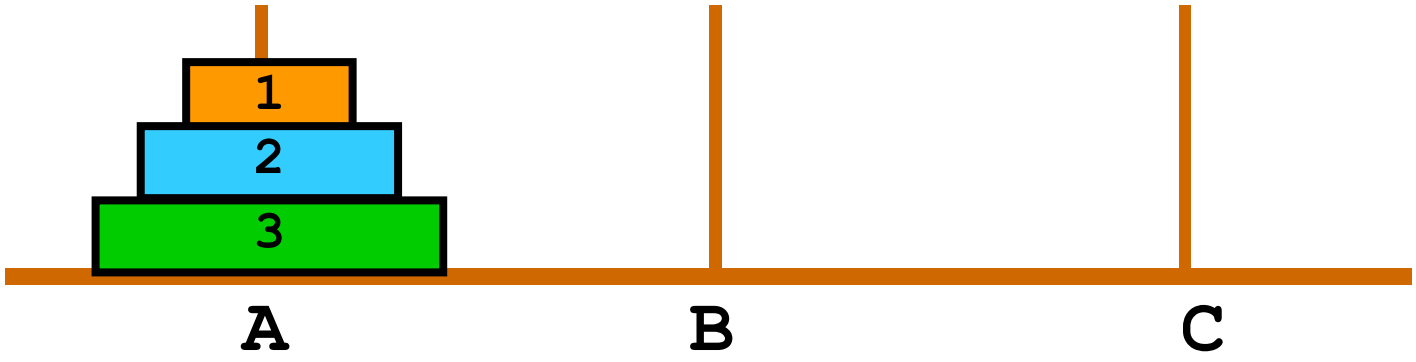
(2) 把A柱上最下面的盘子直接移到C柱上

(3) 用A柱做过渡，将B柱上的n-1个盘子移到C上



递归算法的应用

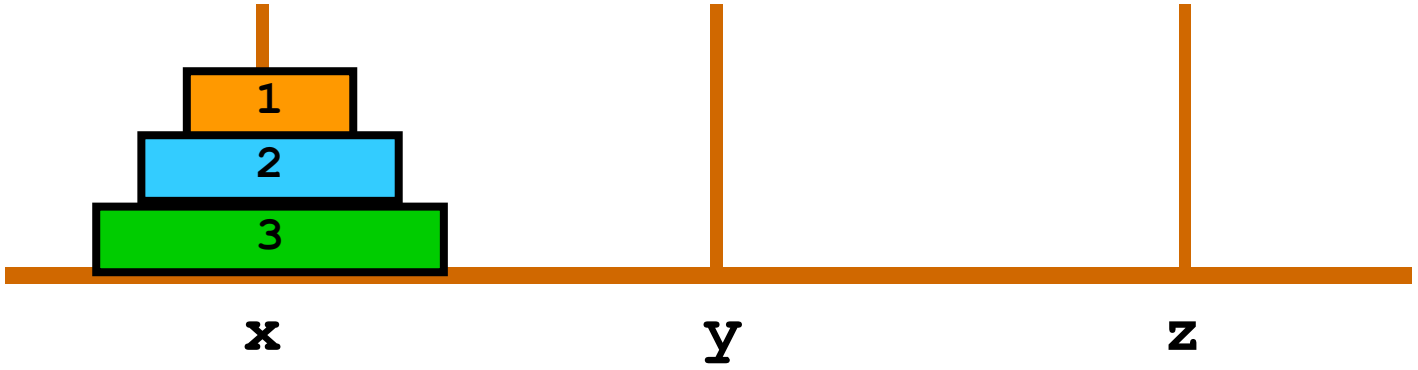
- 移动1，2个盘采用同样的方法
  - 先把上面的1个盘子移到一个过渡柱子上
  - 然后把最下面的1个盘子移到目标柱子上
  - 最后把上面的1个盘子移到目标柱子上



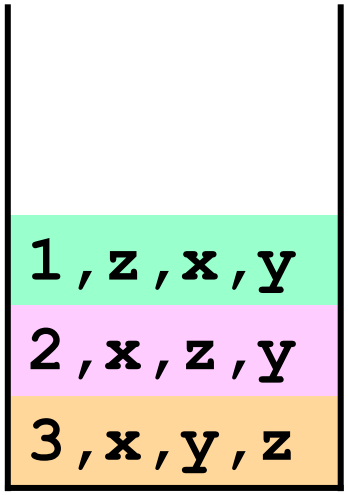
递归算法的应用

- 递归算法

```
void hanoi
    (int n, char x, char y, char z){
    if (n==1)    move(x, 1, z);
    else {
            hanoi(n-1, x, z, y);
            move(x, n, z);
            hanoi(n-1, y, x, z);
      }
}
```



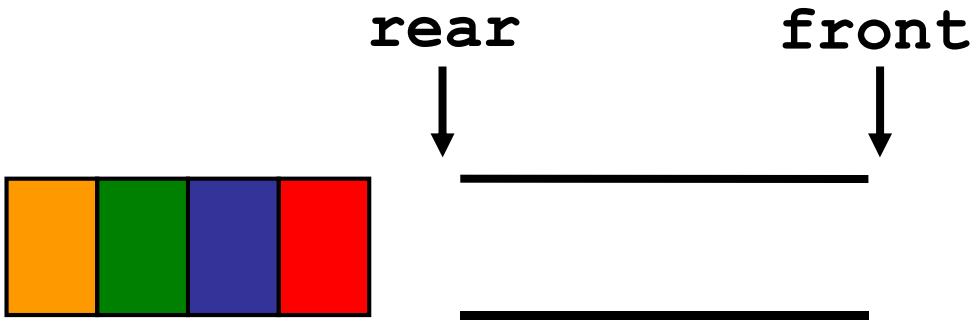

```
void hanoi
    (int n, char x, char y, char z){
    if (n==1)     move(x, 1, z);
    else {
        hanoi(n-1, x, z, y);
        move(x, n, z);
        hanoi(n-1, y, x, z);
    }
}
```



运行栈

队列的类型定义

- 定义
  - 队列是必须在一端删除(队头**front**)，在另一端插入(队尾**rear**)的线性表
- 特性
  - 先进先出(FIFO, First In First Out)



队列的类型定义

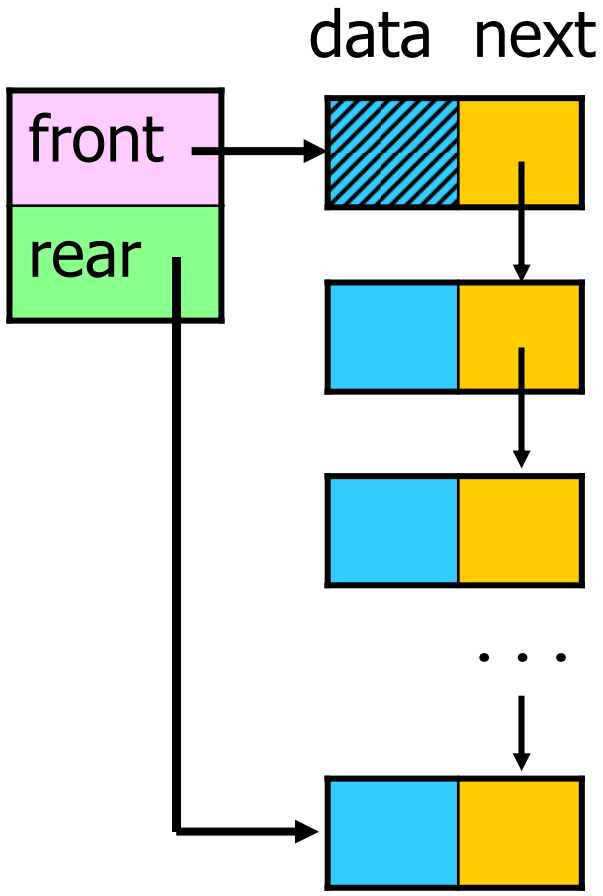
```
ADT Queue{
   数据对象:具有线形关系的一组数据
   操作:
      bool EnQueue(e);   //入队
      bool OutQueue(&e);   //出队
      bool GetFront(&e);   //取队头
      bool IsEmpty(); //空吗?
      bool Clear();   //清空
}
```

队列的表示：链式表示

- 链式表示和实现

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

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



队列的表示：链式表示

- 初始化

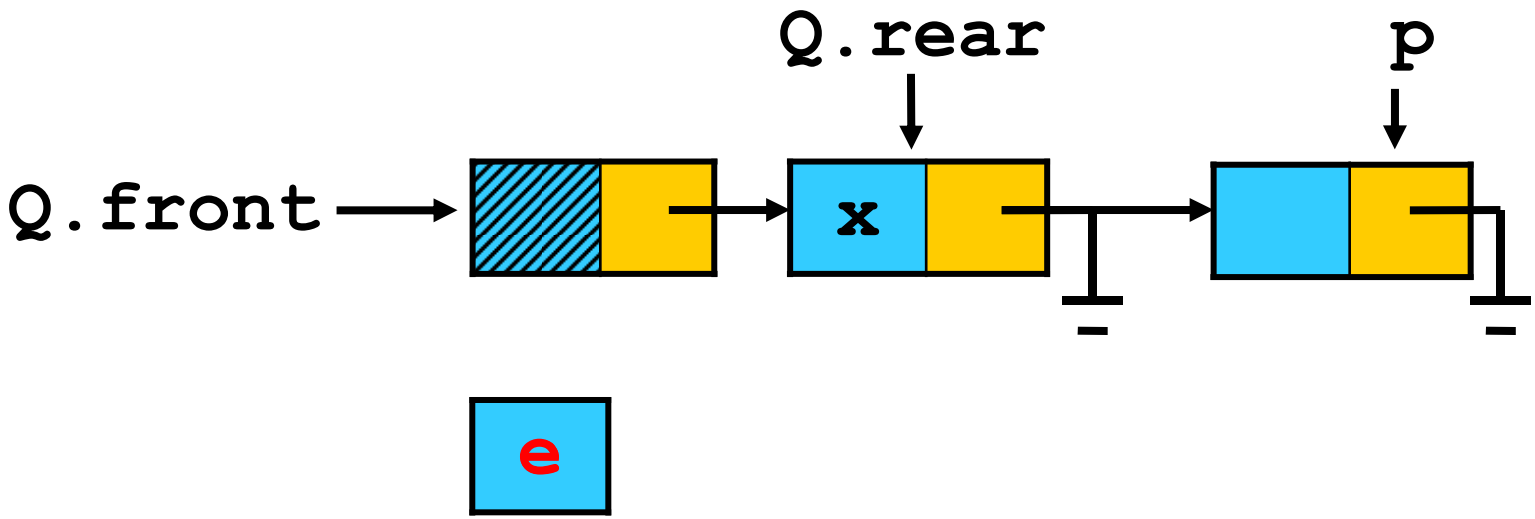
```
int InitQueue(LinkQueue& Q)
{
  Q.front = Q.rear =
  (QueuePtr)malloc(sizeof(QNode));
  if(!Q.front) return ERROR;
  Q.front->next = NULL;
  return OK;
}
```

队列的表示：链式表示

- 入队

```
int EnQueue(LinkQueue& Q,
                    QElemType e){
    p = (QueuePtr)malloc(QNode);
    if(!p)    return ERROR;
    p->data = e; p->next = NULL;
    Q.rear->next = p;
    Q.rear = p;
    return OK;
}
```

```
int EnQueue(LinkQueue& Q, QElemType e){
     p = (QueuePtr)malloc(QNode);
     if(!p)     return NoMemory;
     p->data = e;    p->next = NULL;
     Q.rear->next = p;
     Q.rear = p;
     return OK;
}
```



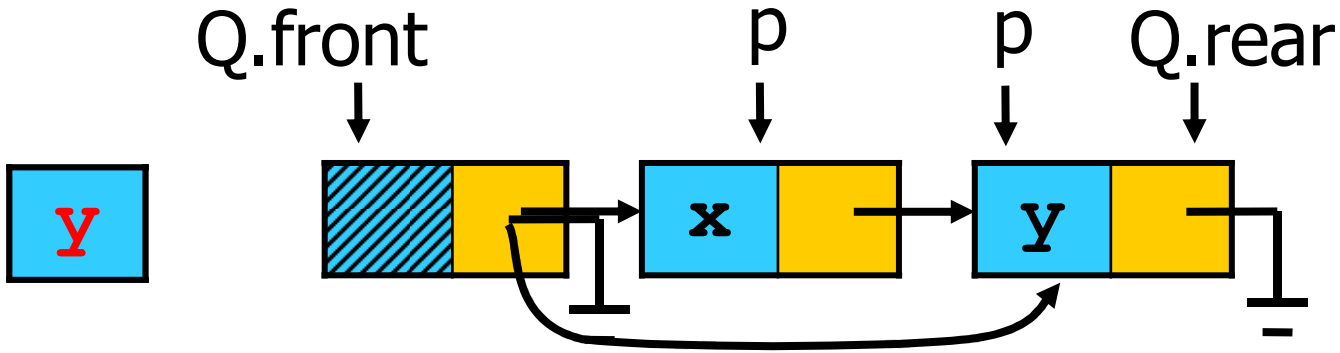
队列的表示：链式表示

- 出队

```
int DeQueue(LinkQueue& Q,
                    QElemType& e){
  if(Q.front = Q.rear)    return ERROR;
    p = Q.front->next;
    e = p->data;
    Q.front->next = p->next;
  if(Q.rear == p)    Q.rear = Q.front;
  free(p);
  return OK;
}
```

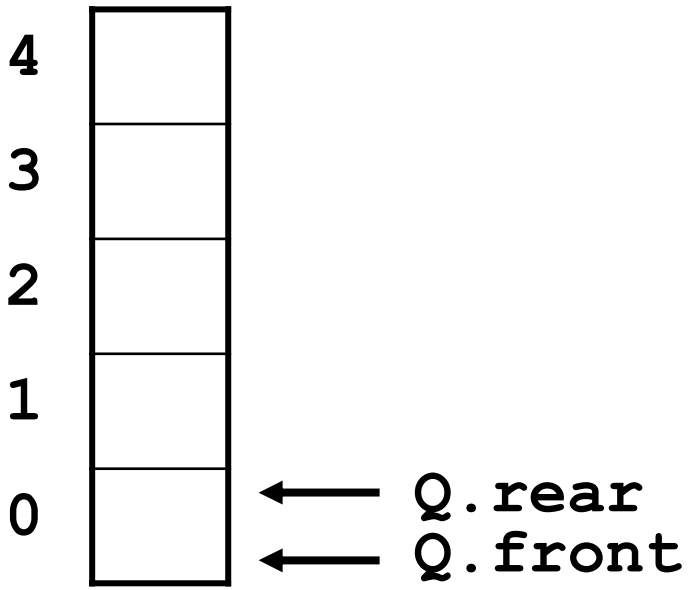
```
int DeQueue(LinkQueue& Q, QElemType& e){
      if(Q.front = Q.rear)return ERROR;
      p = Q.front->next;
      e = p->data;
      Q.front->next = p->next;
      if(Q.rear == p)      Q.rear = Q.front;
      free(p);
      return OK;
}
```

最后一个元素



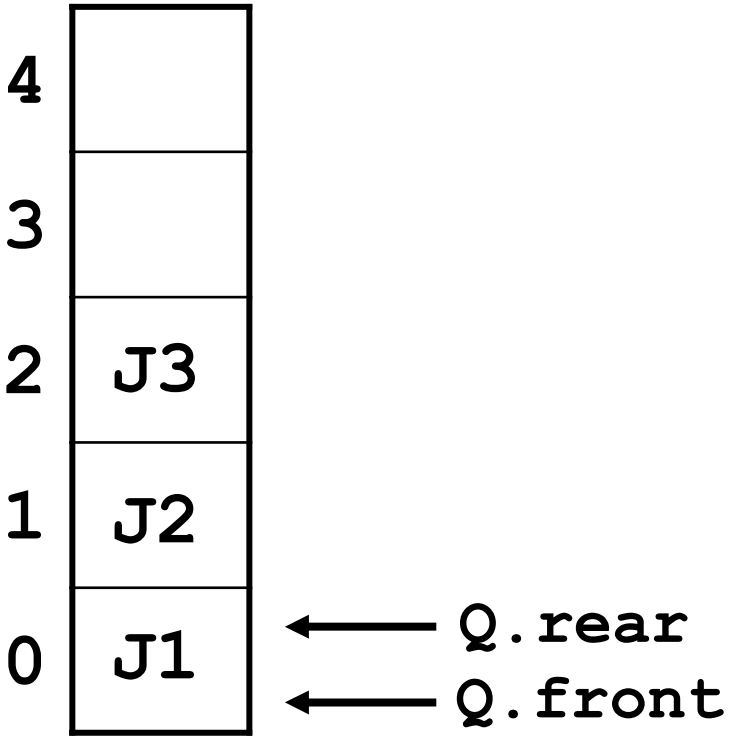
队列的表示：顺序表示

- 顺序表示和实现
  - 用数组存储数据
  - 初始化：**front = rear = 0**
  - 入队：**rear ++**
  - 出队：**front ++**



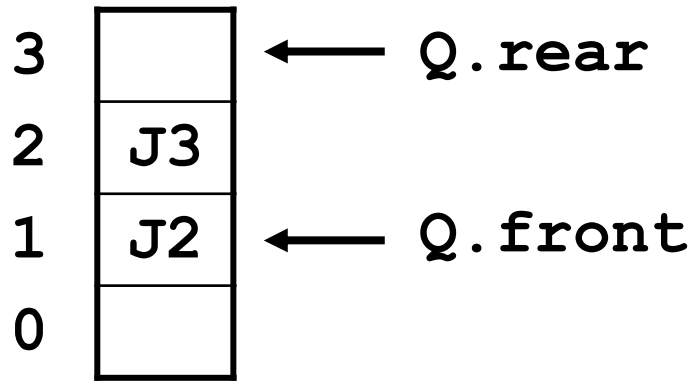
队列的表示：顺序表示

- 插入和删除



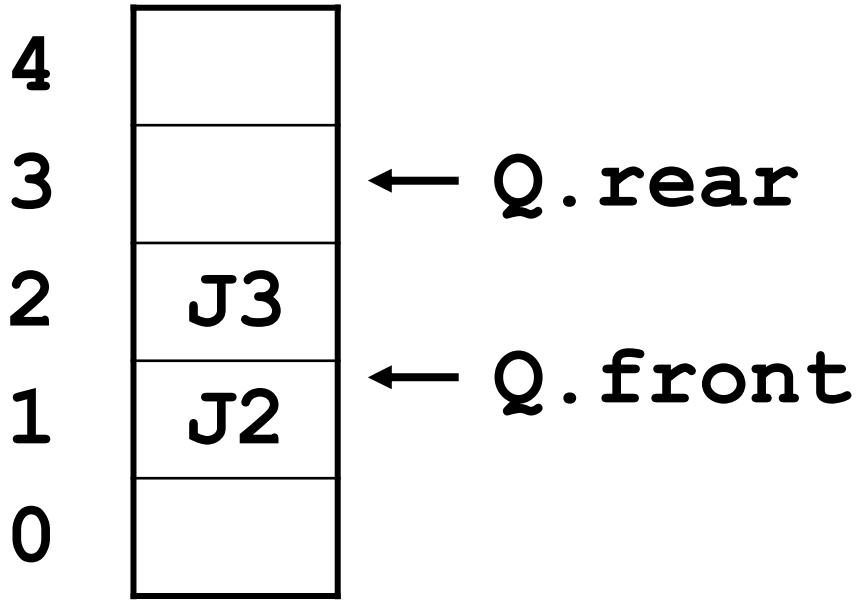
队列的表示：顺序表示

- 指针的指法
  - front指向队头元素
  - rear指向队尾元素的后一个单元
- 其实也可以如此错开
  - front指向队头元素的前一个单元
  - rear指向队尾元素
- 为什么如此呢？



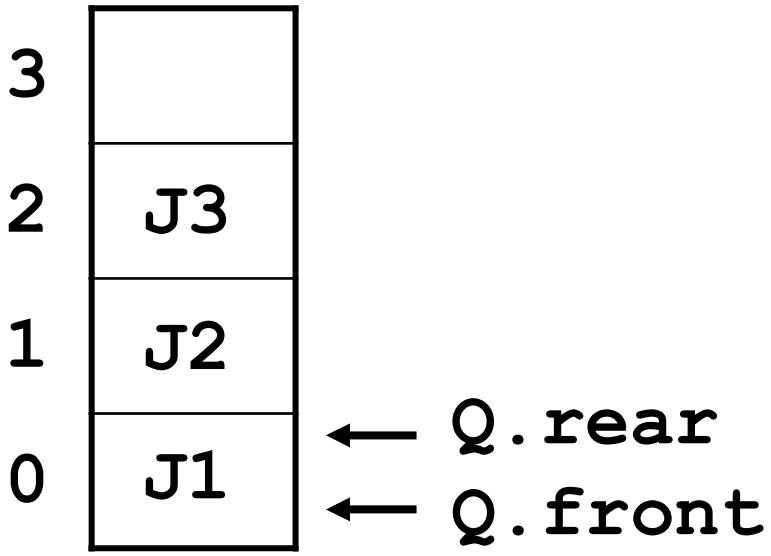
队列的表示：顺序表示

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



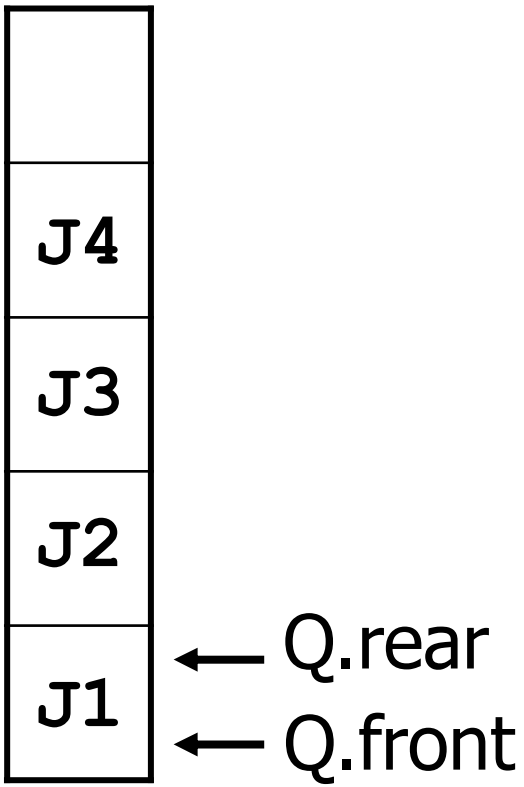
队列的表示：顺序表示

- 原因2：便于判别空队列
  - 如果不错开，则空队列和只有一个元素时，都是front=rear，难以区别
  - 或者空队列有可能有两种情况



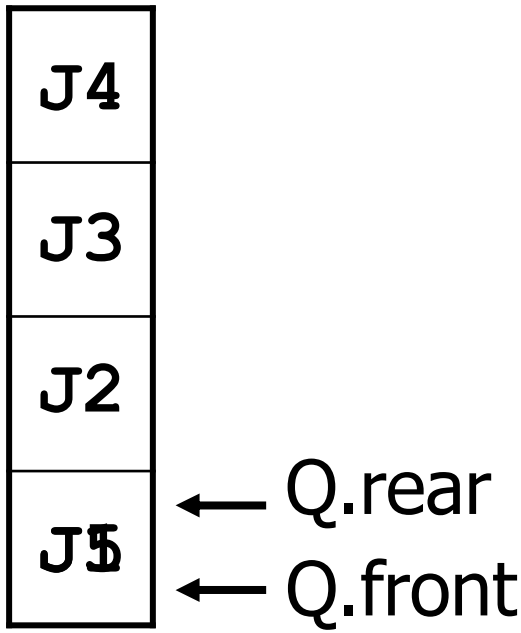
队列的表示：顺序表示

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



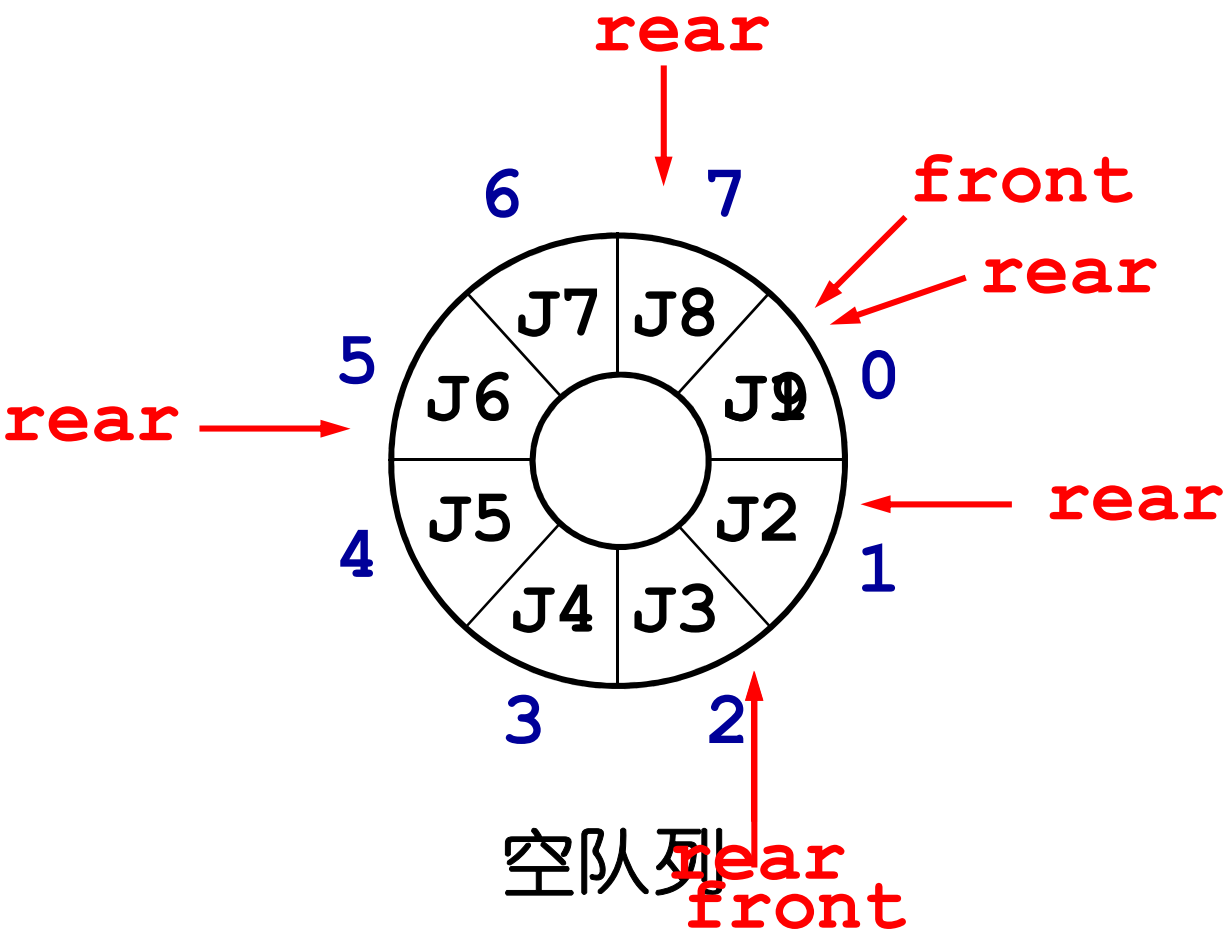
队列的表示：顺序表示

- 思考
  - 既然上面不够，下面浪费
  - 何不利用下面的空间？
- 循环队列
  - 当**rear**向上走到顶的时候，重新返回到最下端（如果下面有空闲单元）



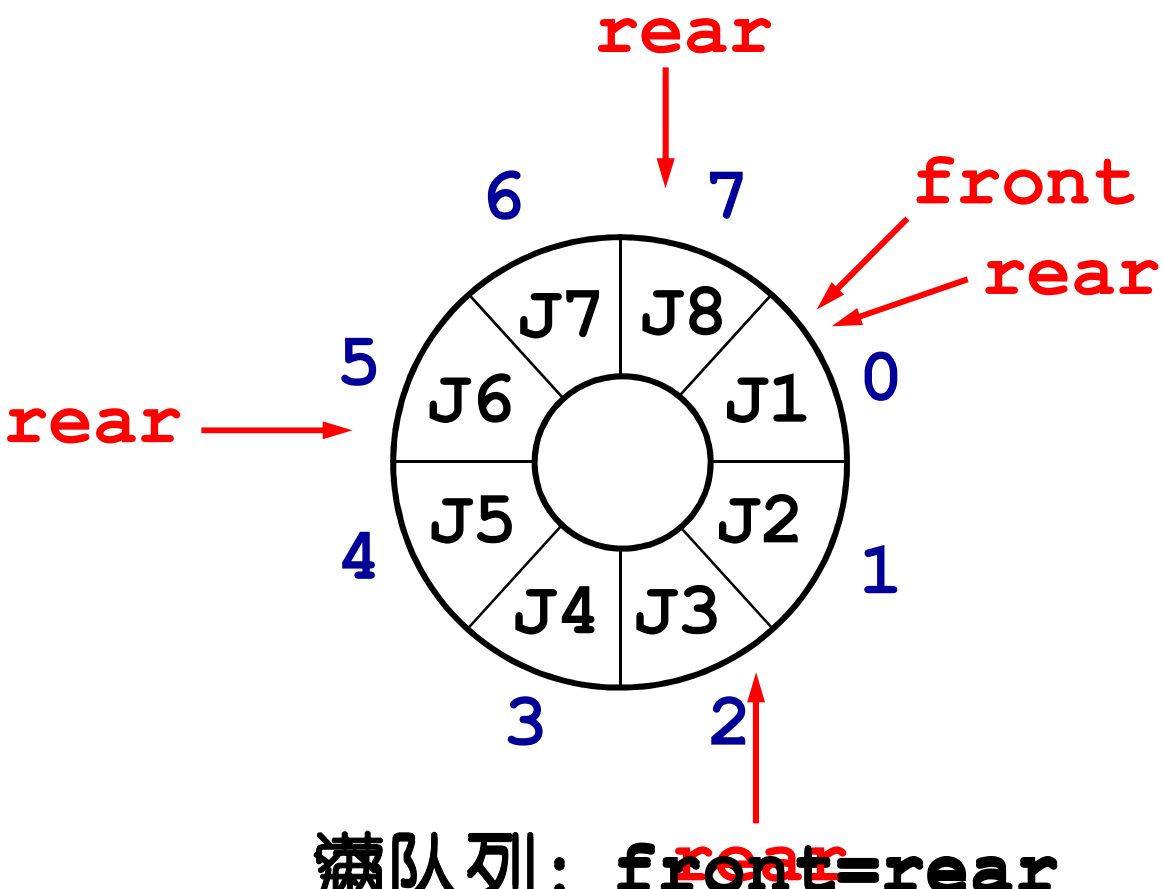
队列的表示：顺序表示

- 循环队列



队列的表示：顺序表示

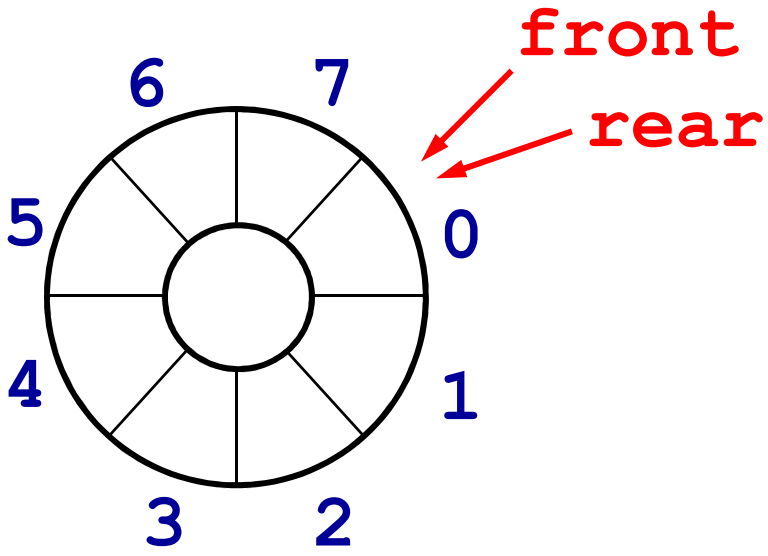
- 如何区别空队列和满队列？



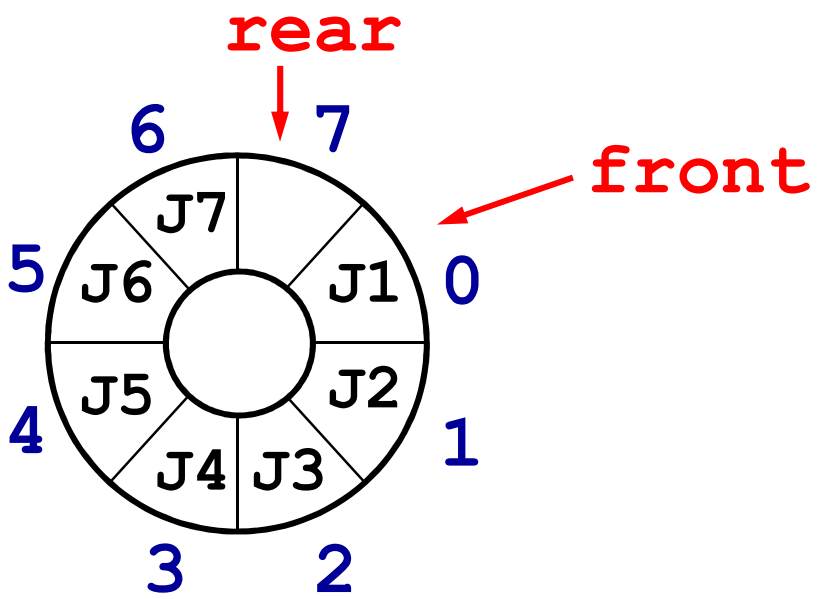
满队列：front=rear

队列的表示：顺序表示

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



空



满

队列的表示：顺序表示

- 循环队列类型定义

```
#define MAXQSIZE 100
typedef struct {
    QElemType *base;
    int front;
    int rear;
}SqQueue;
```

队列的表示：顺序表示

- 初始化

```
int InitQueue(&SqQueue &Q)
{
  Q.base = (QElemType*) malloc
     (MAXQSIZE*sizeof(QElemType));
  if(!q.base) return ERROR;
  Q.front = Q.rear = 0;
  return OK;
}
```

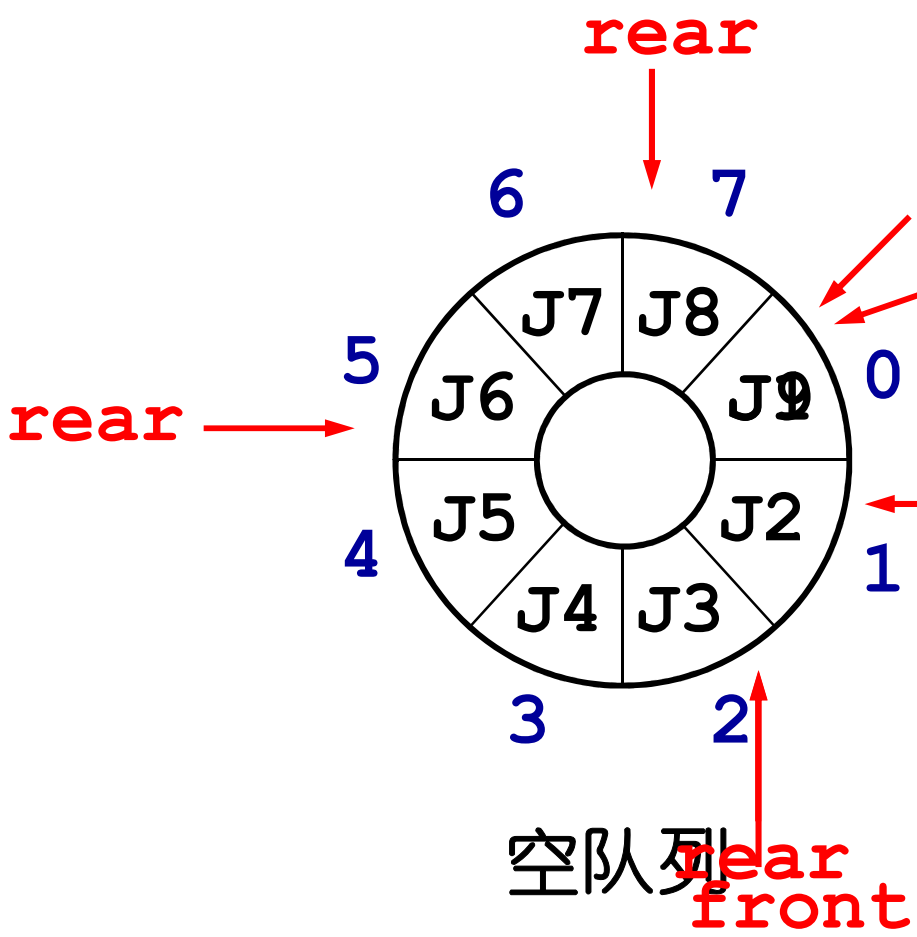
队列的表示：顺序表示

- 得到队列长度
  - 即元素个数

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

队列的表示：顺序表示

- 得到队列长度



rear≠front时：

length=(rear-front+8)%8

=(7-2+8)%8=6

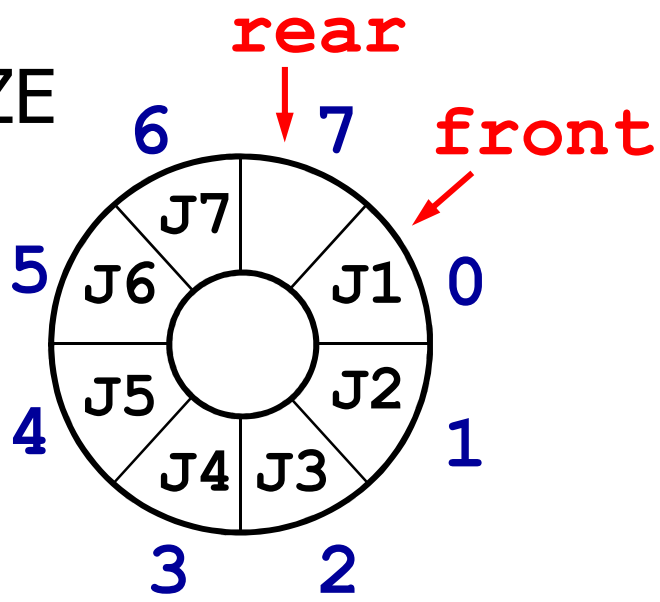
队列的表示：顺序表示

- 入队

```
int EnQueue(SqQueue &Q,
            QElemType e){
  if((Q.rear+1)% MAXSIZE == Q.front)
     return ERROR;  //满队列
  Q.base[Q.rear] = e;
  Q.rear = (Q.rear + 1) % MAXQSIZE;
  return OK;
}
```

队列的表示：顺序表示

- 注意
  - 满队列：**rear**下一个就是**front**
    - (rear+1) % MAXQSIZE == front
  - **rear**指向下一个单元
    - rear=(rear+1)%MAXQSIZE



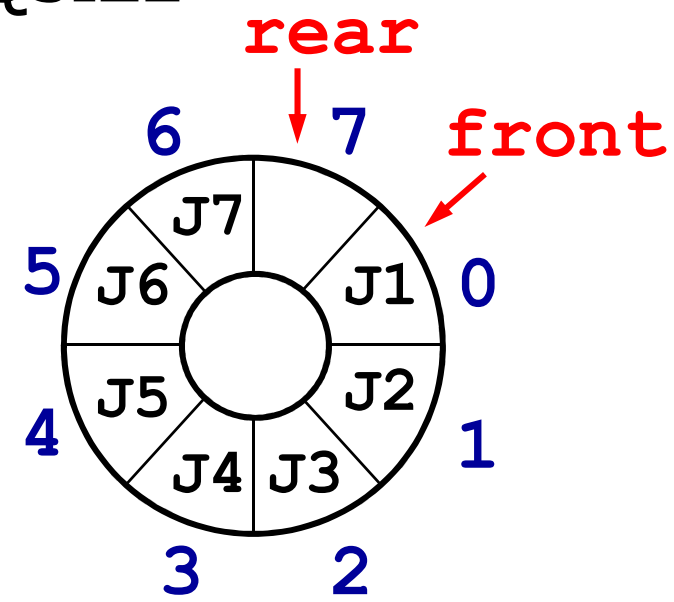
队列的表示：顺序表示

- 出队

```
int DeQueue(SqQueue &q,
                    QElemType &e){
  if(Q.front == Q.rear)
     return ERROR;    //空队列
  e = Q.base[Q.front];
  Q.front = (Q.front+1)% MAXQSIZE;
  return OK;
}
```

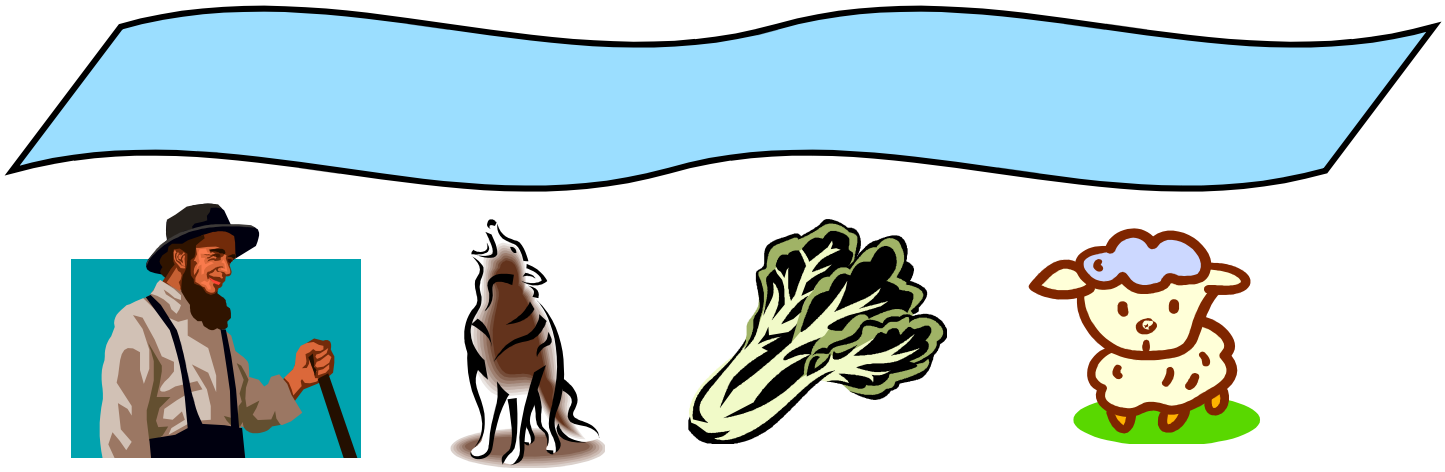
队列的表示：顺序表示

- 注意
  - 空队列：**rear==front**
  - **front**指向下一个单元
    - front = (front+1)%MAXQSIZE



队列的应用：农夫过河问题

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



队列的应用：农夫过河问题

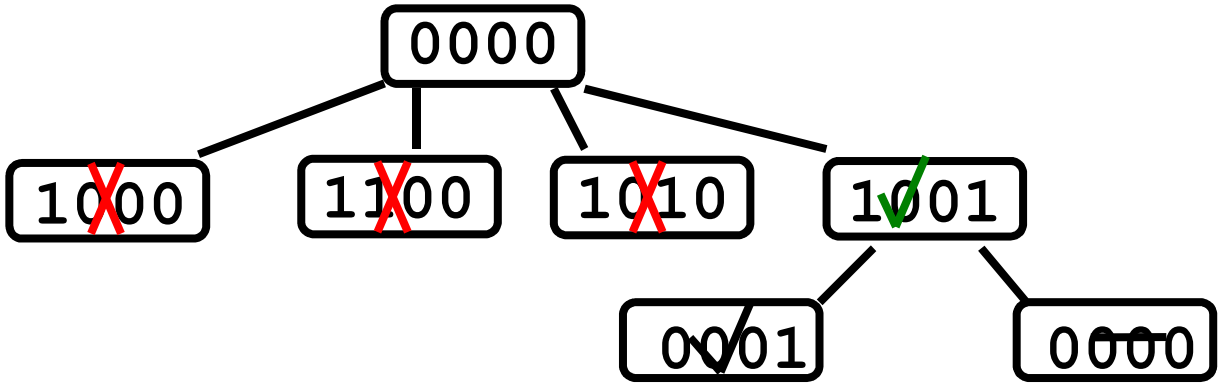
- 安全与不安全的状态

用四位二进制数顺序分别表示农夫、狼、白菜和羊的当前状态（位置）。用0表示在河的南岸，1表示在河的北岸。例如整数5(其二进制表示为0101) 表示农夫和白菜在河的南岸，而狼和羊在北岸。

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

农夫过河问题：状态空间搜索

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

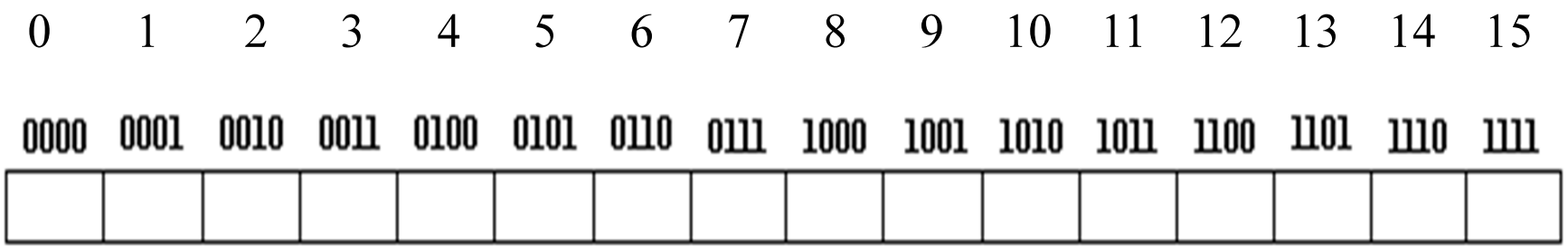


广度优先搜索：在搜索过程中总是先考虑当前状态的所有状态，再进一步考虑更后面的各种情况。

农夫过河问题：数据结构

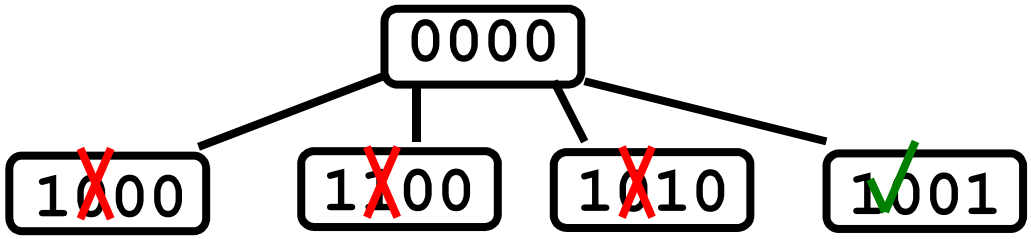
1).顺序表route[]：状态i是否已被访问过，若已被访问过则在这个顺序表元素中记入前驱状态值。

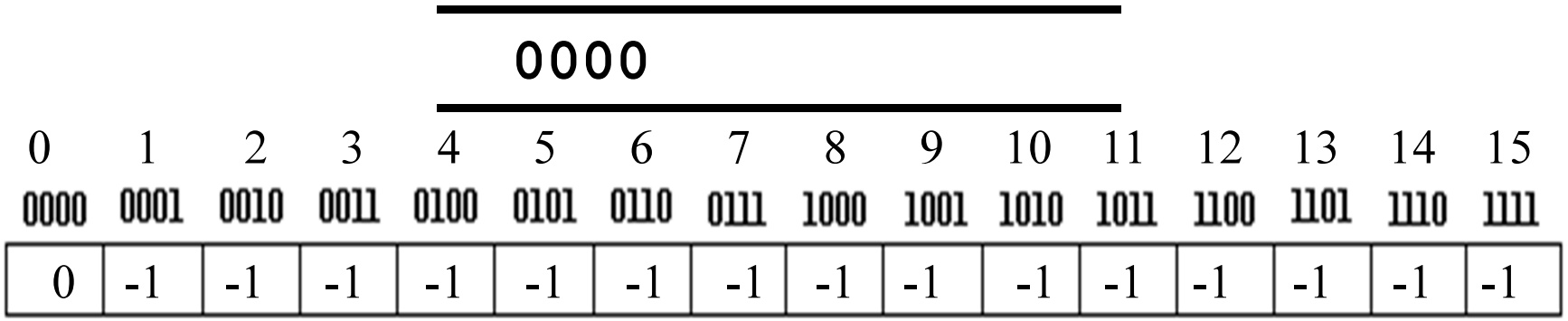
- route[i] = -1,表示未访问过;则 route[i] = pre,表示状态i已经访问过，且是从安全状态pre过渡来的。
- 最后可以利用route顺序表元素的值建立起正确的状态路径。



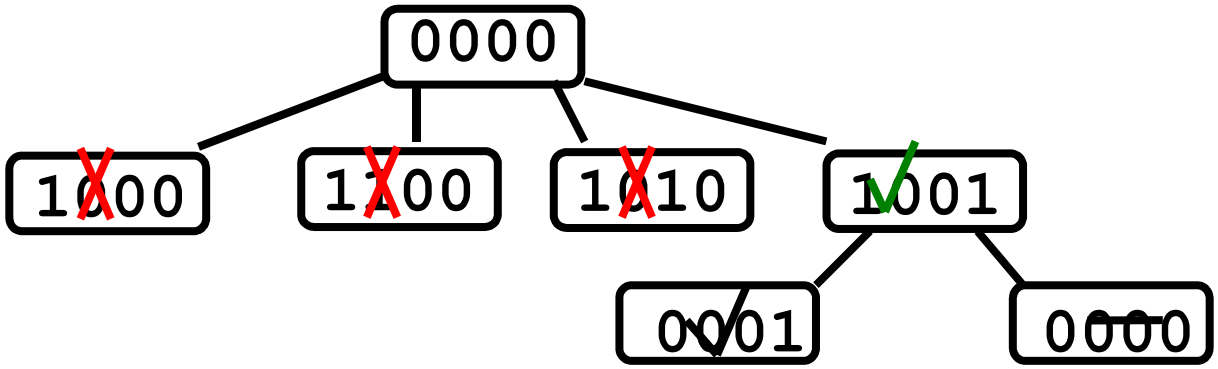
2)整数队列moveTo：每个元素表示可以安全到达的状态。

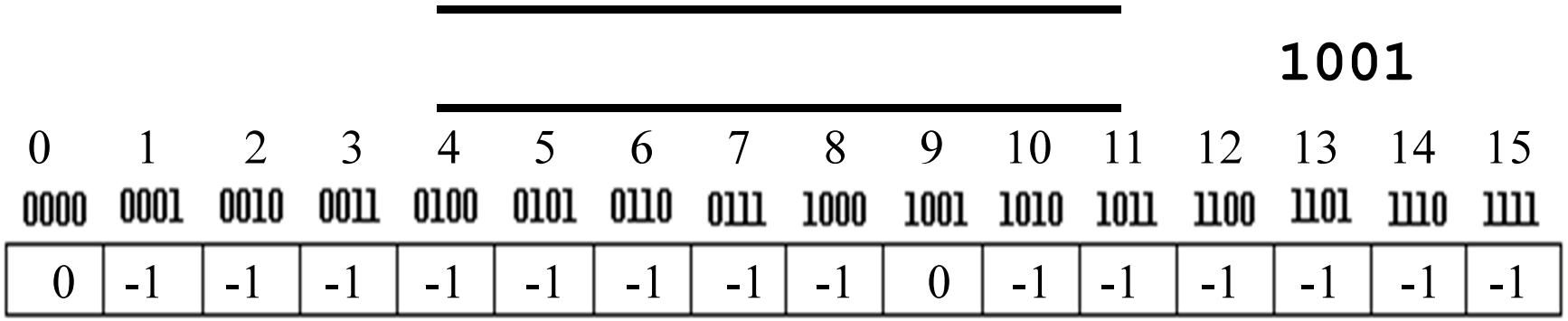
农夫过河问题：数据结构


0000
1000
1100
1010
1001

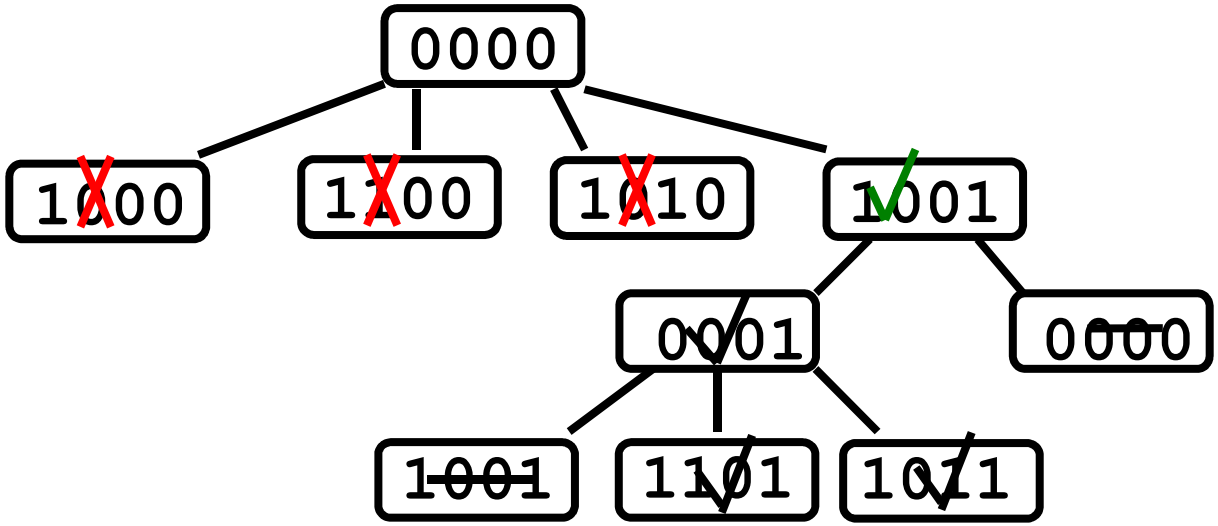

0000
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

农夫过河问题：数据结构


0000
1000
1100
1010
1001
0001
0000


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

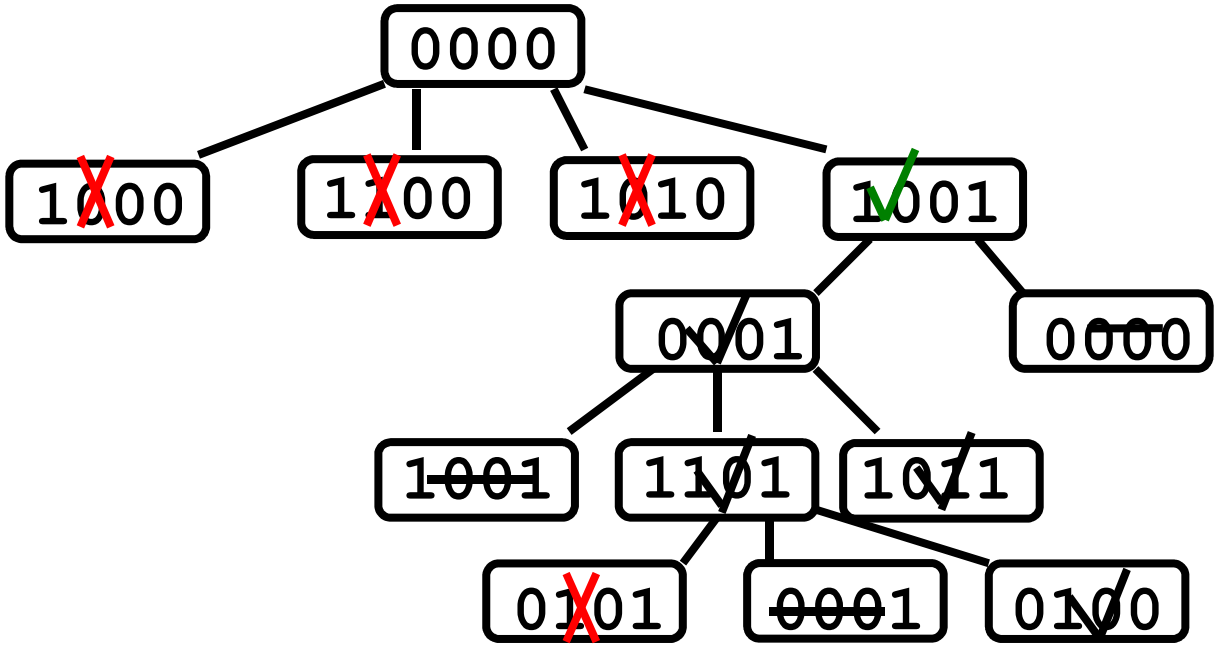
农夫过河问题：数据结构



0001

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

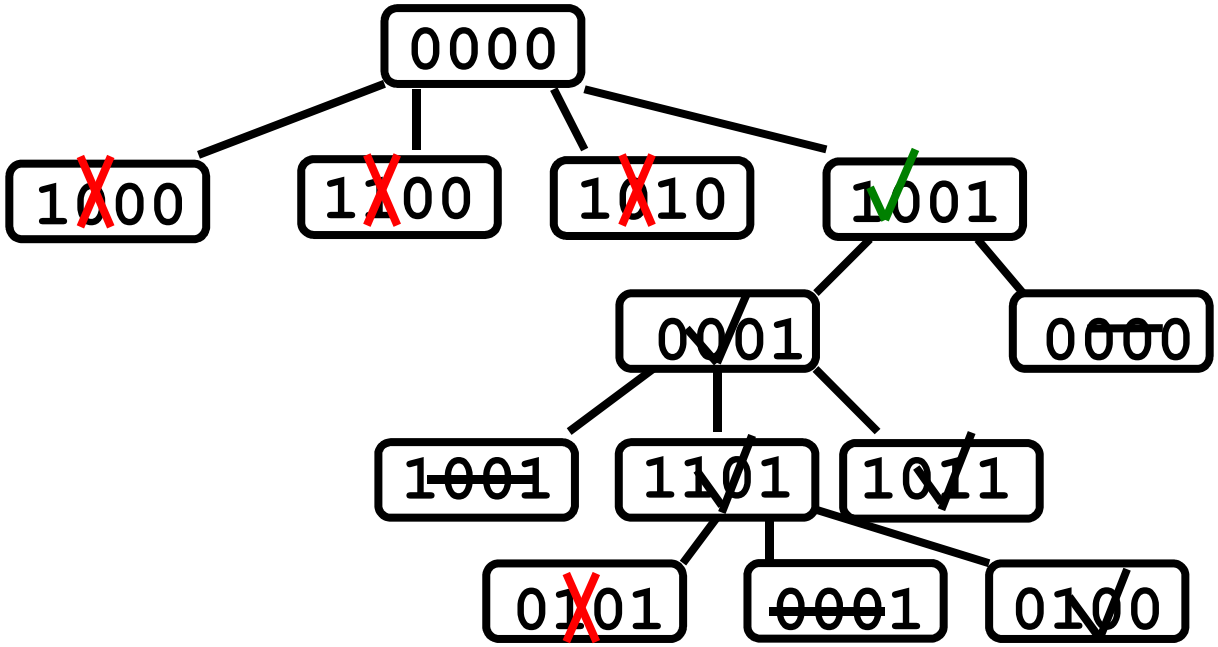
农夫过河问题：数据结构



1101 1011

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

农夫过河问题：数据结构



1011 0100

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

农夫过河问题：广度优先搜索

Path：15，6，14，2，11，1，9，0

从初始状态0到最终状态15的动作序列为：

农夫把羊带到北岸；

农夫独自回到南岸；

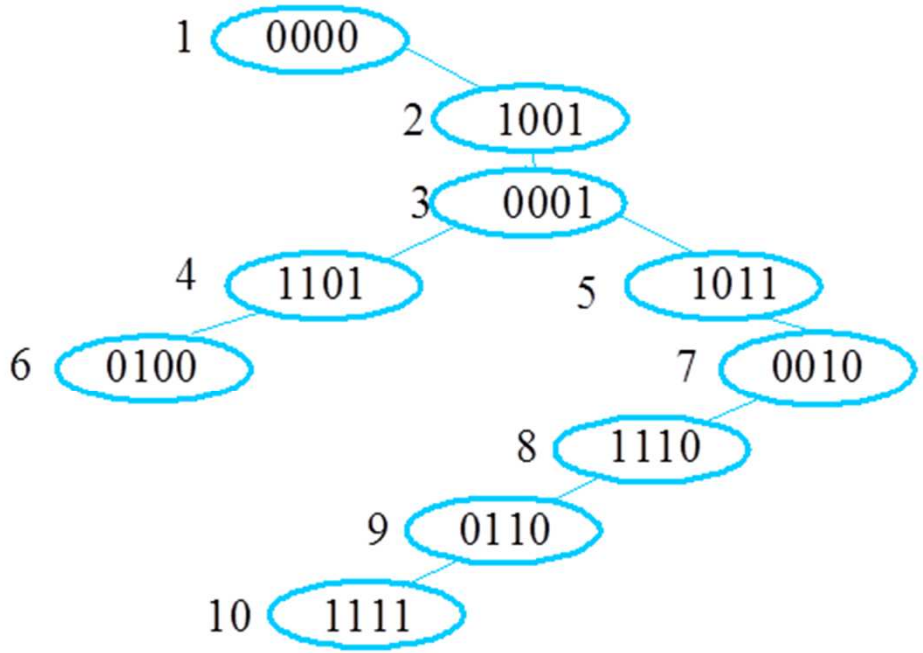
农夫把白菜带到北岸；

农夫带着羊返回南岸；

农夫把狼带到北岸；

农夫独自返回南岸；

农夫把羊带到北岸。



广度优先搜索的结果和顺序

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

农夫过河问题：角色状态

状态用四位二进制整数表示，从中可知每个对象是否已经在北岸。

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

农夫过河问题：安全状态的判断

```
int safe(int location){//状态安全则返回1
 if((goat(location)==cabbage(location))
   &&(goat(location)!=farmer(location)))
          return 0;  // 羊吃白菜
 if((goat(location)==wolf(location))
   &&(goat(location)!=farmer(location)))
          return 0;  // 狼吃羊
  return 1;         // 其他状态是安全的
}
```

农夫过河算法

```
准备一个数组route记录路径
准备一个队列moveto记录多重选择
初始状态0(0000)加入moveTo和route
while(队列非空 && 还没到终止状态){
    state = DeQueue(moveTo);   //出队当前状态
    for(每个从state可以到达的状态newstate)
        if(newstate安全且未访问过) {
            EnQueue(moveTo, newstate);
            newstate加入route
        }
}
输出结果
```

农夫过河算法

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

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

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

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

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

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

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

队列的应用：农夫过河问题

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