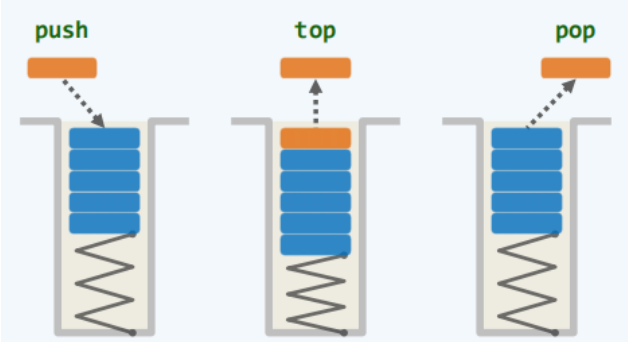
栈

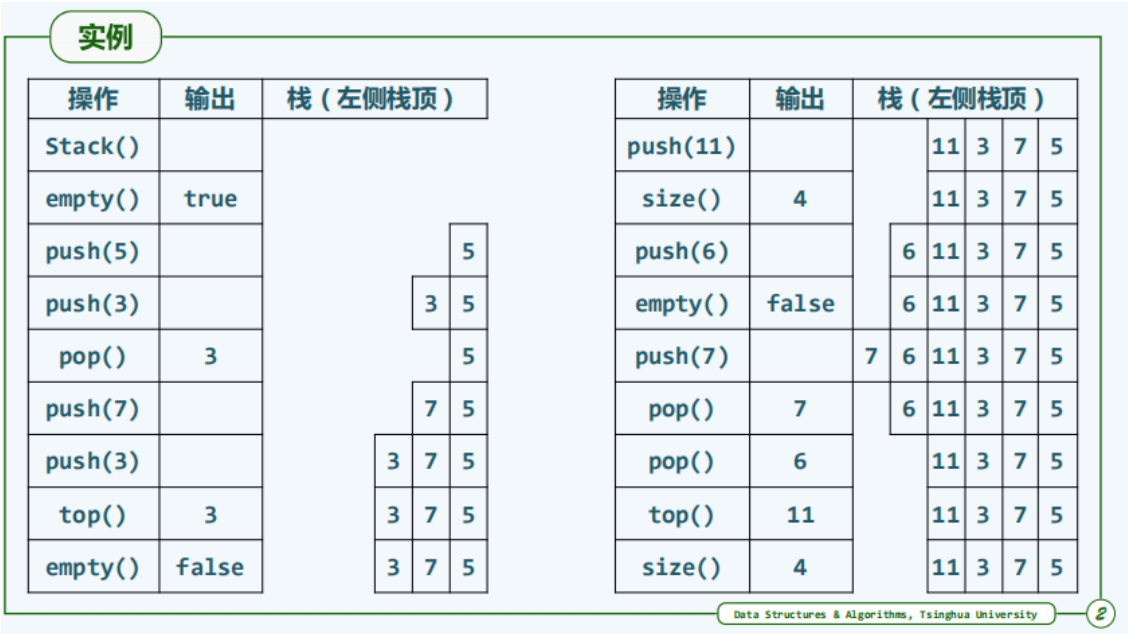
什么是栈

也是由一系列元素组成的线性结构，但任何时候只能访问其中特定位置的数，比如序列的末尾，而其他位置的元素禁止访问。通常将允许访问的末端元素叫做栈顶，而另一端称为栈底（也叫盲端）。这种结构的特点是先进后出（FILO）或后进先出（LIFO）。



栈的接口

1. push()：插入一个元素，但只能将其插入在顶部
2. pop()：取出一个元素，但只能取出顶部的元素
3. top()：只查看顶部元素的值，而不将其pop
4. size()/empty()/Stack()



栈的实现

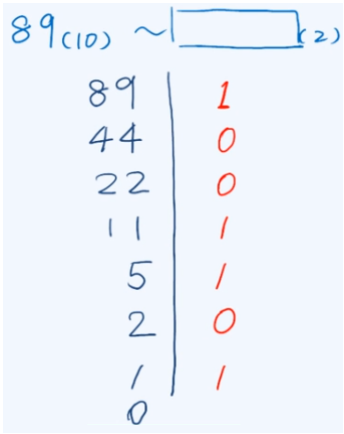
- 栈属于一种受限的序列，故可直接基于向量或列表派生，如下通过public直接继承自向量模板。
- size()和empty()直接取自于向量模板，push、pop、top接口内部也是直接通过向量本身的接口实现。
- 这里将向量的末端作为可访问的栈顶，而向量的首端作为不可访问的盲端，这使得所有栈的接口都只需要花费O(1)时间。如果将其颠倒过来，则所有栈的接口的复杂度将提升到O(n)，因为这里所使用到的向量接口的复杂度均取决于当前元素的后继长度，越靠近末端，复杂度越低。

```
❖template <typename T> class Stack: public Vector<T> { //由向量派生的栈模板类
  public: //size()、empty()以及其它开放接口均可直接沿用
     void push( T const & e ) { insert( e ); }  //入栈
     T pop() { return remove( size() - 1 ); } //出栈
     T & top() { return (*this)[ size() - 1 ]; } //取顶
  }; //以向量首/末端为栈底/顶——颠倒过来呢？
```

栈的应用

进制转换

这是一个逆序输出问题：将十进制数转换成n进制。



把十进制数除以n，所得商作为下一次除法的被除数，余数作为结果，然后把新的被除数除以n，所得商作为下一次除法的被除数，余数作为结果，以此类推，直到商为0。此时将各个结果逆序输出即为最终的结果，为了便于输出，可以在计算过程中将结果压栈保存，然后在计算完成后弹栈输出。

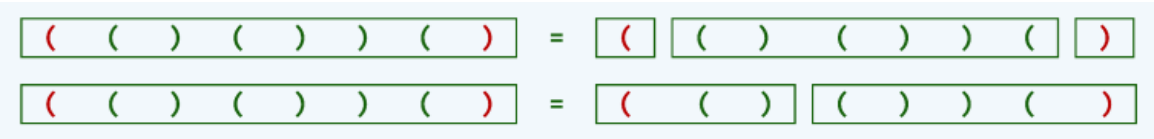
```
  // S为保存结果的栈，n为要转换的数，base为要转换的进制
❖void convert( Stack<char> & S, __int64 n, int base ) {
    static char digit[] = //新进制下的数位符号，可视base取值范围适当扩充
       { '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F' };
    while (n > 0) { //由低到高，逐一计算出新进制下的各数位
       S.push( digit[n % base] ); //余数（对应的数位）入栈
       n /= base; //n更新为其对base的除商
    }
  }
❖main() {
    Stack<char> S; convert(S, n, base); //用栈记录转换得到的各数位
    while ( !S.empty() ) printf( "%c", S.pop() ); //逆序输出
  }
```

括号匹配

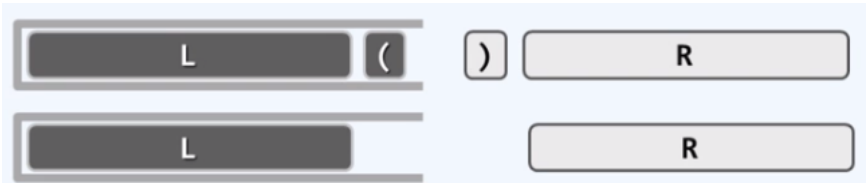
这是一个递归嵌套问题：判断一个表达式的括号是否成对出现。

```
( a [ i - 1 ] [ j + 1 ] ) + a [ i + 1 ] [ j - 1 ] ) * 2        //失配
( a [ i - 1 ] [ j + 1 ]   + a [ i + 1 ] [ j - 1 ] ) * 2        //匹配
```

这个问题不能通过减治和分治来完成，比如以下情况进行减治和分治都会得到错误的结果（为了方便，预先扫描消去了除括号外的符号）：



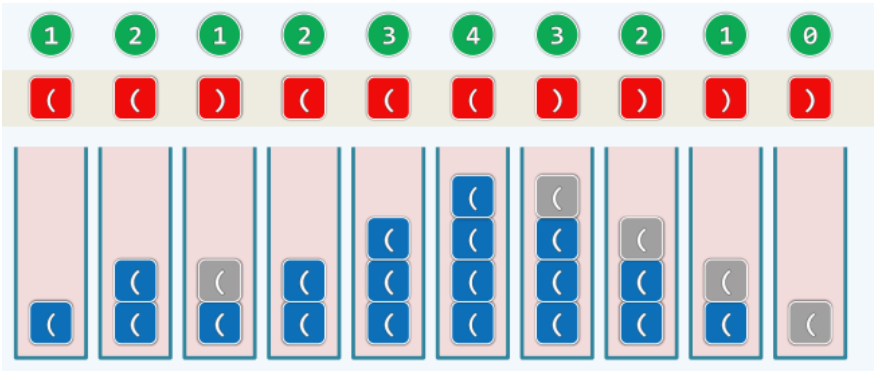
正确的思路应是消去紧邻的括号对，这样消除后并不会影响全局的判断。但是仅仅消除紧邻的括号对并不能在一次扫描后就做出判断，必须不断地扫描并消除紧邻的括号对，直到最后表达式里没有括号才能说明括号匹配。这时就是栈发挥作用的时候了，即在遇到左括号时将其入栈，而在遇到右括号时将左括号出栈，只要栈提前变空或最后栈非空就说明括号不匹配（最后栈空说明匹配）。这不仅符合正确的思路，而且仅需要一次扫描。



算法实现如下（假设预先扫描消去了除括号外的符号）：

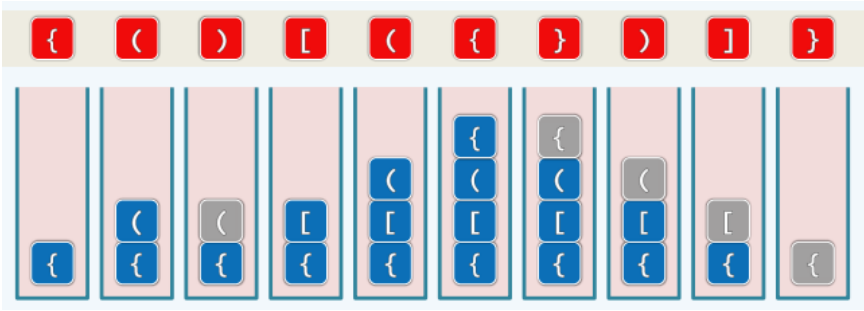
```
bool paren( const char exp[], int lo, int hi ) { //exp[lo, hi)
                                  // 扫描区间
   Stack<char> S; //使用栈记录已发现但尚未匹配的左括号

   for ( int i = lo; i < hi; i++ ) //逐一检查当前字符

      if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：则进栈

      else if ( ! S.empty() ) S.pop(); //遇右括号：若栈非空，则弹出左括号

      else return false; //否则（遇右括号时栈已空），必不匹配
                                                  // 说明左括号缺失

   return S.empty(); //最终，栈空当且仅当匹配
         // 若非空，则不匹配，且说明右括号缺失
}
```

观察以下实例可以发现：其实使用一个计数器足以实现栈的作用，只要计数器变成负数或最后非零就说明不匹配，最后为零则匹配。



但使用计数器只能应对一种类型的括号的情形，比如判断表达式 " [( ]) "，计数器则会认为其是匹配的。因此当遇到多种类型的括号时，就只能使用栈来进行匹配判断了，而且这种匹配判断还能进行拓展，不限于括号的类型和数目，比如HTML标签。

- 不匹配：弹出左括号和当前右括号不同、栈提前变空、栈最后非空；
- 匹配：最后栈空。

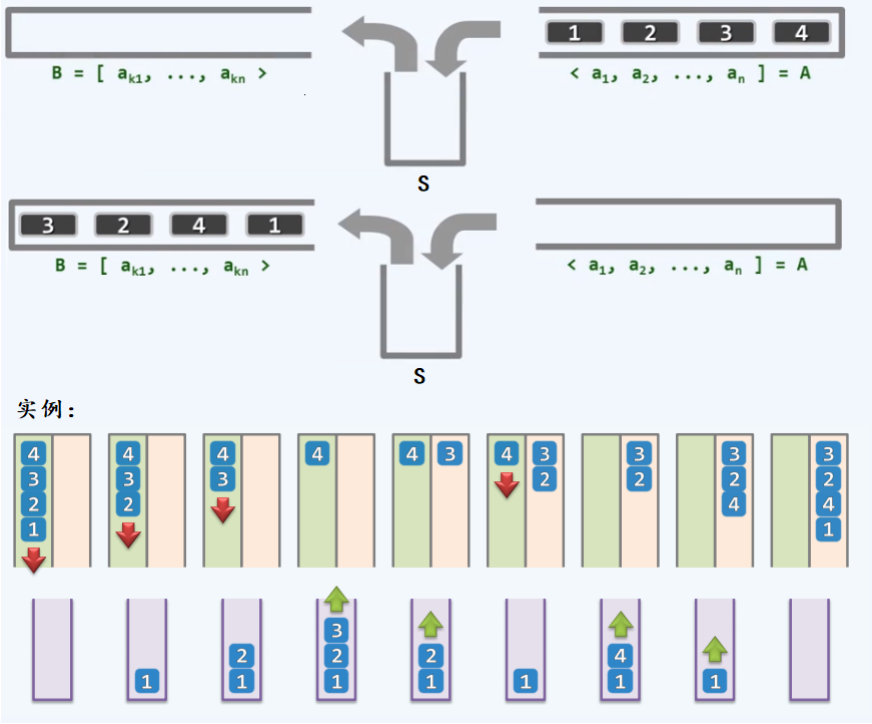


栈混洗

假设有三个栈A、B、S，栈A存在一个序列，对其只允许两种操作：

- S.push(A.pop())：将A的栈顶元素推入S中；
- B.push(S.pop())：将S的栈顶元素推入B中。

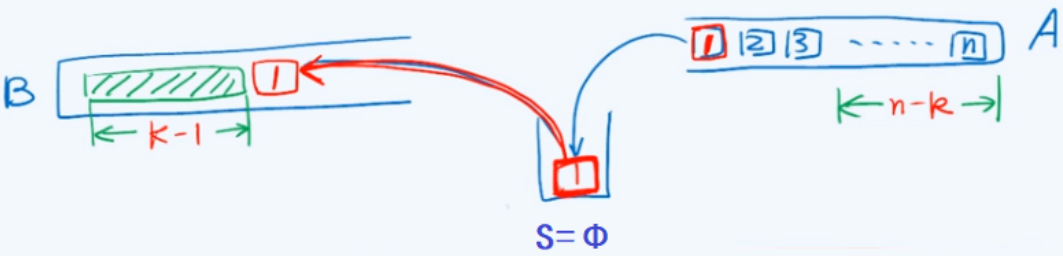
不断经过这两种操作，使得栈A中的序列以新的次序存放在栈B中，这个过程就称之为A的一个栈混洗，即按照某种规则将栈中的序列重新排列。



栈混洗总数

问题：一个含有n个元素的序列，经过栈混洗后可以总共产生多少种序列（记为SP(n)）？

1. 已知一个序列的全排序总数为 n! ，也就是说 SP(n) 必然小于或等于 n! 。
2. 对于栈A的第一个元素作为第k个元素插入到栈B的情况，其栈混洗个数为SP(K-1)×SP(n-k)。由于k取值为1~n，因此将k的每种取值下的栈混洗个数相加就是整个序列的栈混洗总数SP(n)：



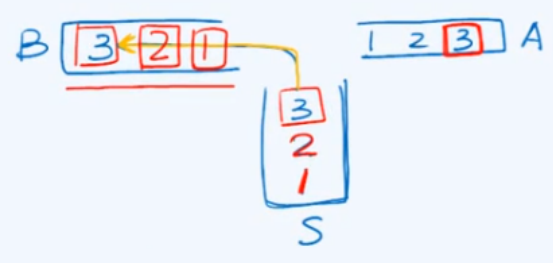
$$SP(n) = \sum_{k=1}^{n} SP(k-1) \cdot SP(n-k)$$

$$= catalan(n) = \frac{(2n)!}{(n+1)! \cdot n!} \quad \text{（其中，SP(1)=1）}$$

判断栈混洗序列

问题：对于指定输入序列<1，2，3，...，n]，判断其任一排列是否为栈混洗。

1. 假如输入序列为<1，2，3]，此时它的栈混洗个数为5种，而全排列有6种，其中少掉的一种就是[3，1，2>，因为3要作为第一个元素插入，1和2就必须先存在于中转栈S中，此时无论如何都只能按照3->2->1的顺序插入到栈B中，因此不可能出现[3，1，2>的情况。

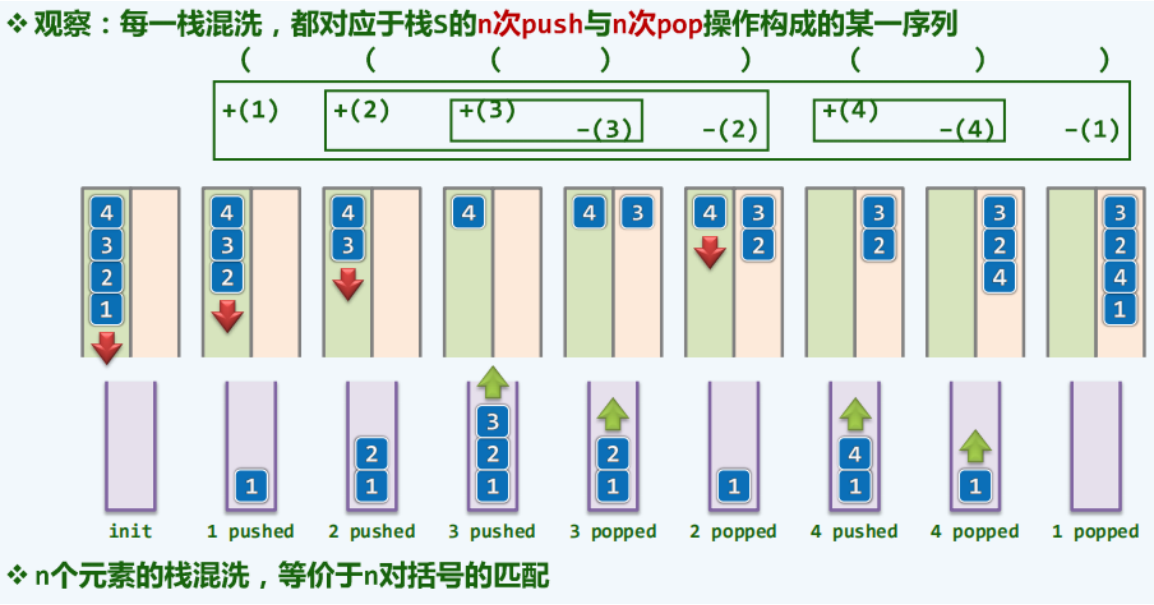


2. 对于序列中的任意3个元素，它们在栈混洗中的相对次序只与它们三个有关，而与其它元素无关。因此，对于序列中序号为 i<j<k 的三个元素，它们在栈混洗中的排列必不可能是[...，k，...，i，...，j，...>。也就是说，312模式是栈混洗所禁止的一种形式（简称禁形）。
3. 312模式是判断栈混洗序列的充要条件，即只要排列中存在312模式，则该排列就不是栈混洗；只要该排列不是栈混洗，则该排列中就不存在312模式。
4. 判断是否为栈混洗排列的算法：
    - 取任意三个元素(i，j，k)进行312模式的判断，复杂度为O(n^3)；
    - 取任意三个元素(i，j，j+1)进行312模式的判断，复杂度为O(n^2)；
    - 直接借助三个栈进行栈混洗模拟过程，复杂度为O(n)。这个过程其实和括号匹配的过程是相似的。

与括号匹配的关系

每一次栈混洗，都对应中转栈的n次push和n次pop操作构成的序列，如果将push操作看成是左括号，pop操作看成是右括号，就可以发现，这和括号匹配的过程是一样的。

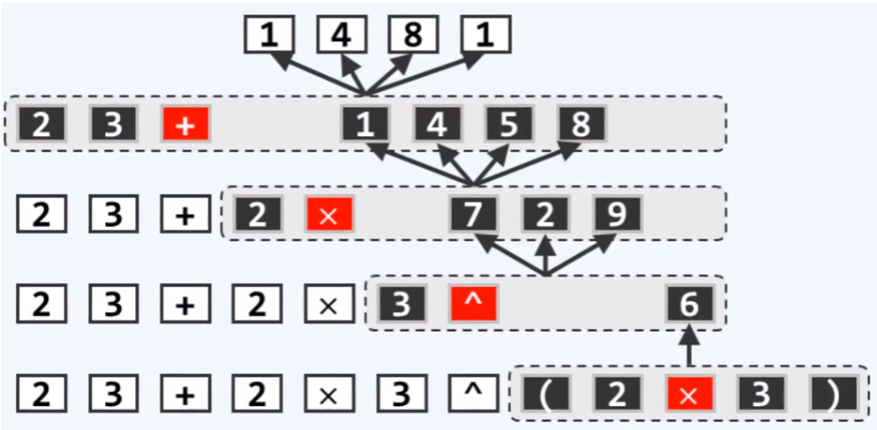
也就是说，一次对n对括号组成的合法表达式的括号匹配过程就对应n个元素的一次合法栈混洗过程，即合法的括号表达式和合法的栈混洗排列之间存在一一对应关系，n个元素的栈混洗总数就是n对括号可以构成的合法表达式的个数。



中缀表达式求值

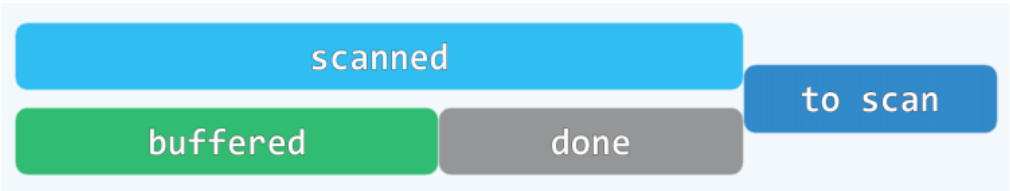
问题：给定语法正确的算术表达式，计算与之对应的数值。

直观的实现方案：从表达式中寻找能够优先计算的子表达式，然后将计算所得的数值替换掉该子表达式，从而减少整体表达式的运算符总数（减而治之），直到最后消除表达式中的所有运算符，得到最终的数值。



这个方案的实现难度在于如何定位当前可以优先计算的运算符，比如通过线性扫描一个表达式，当扫描到一个运算符时，我们无法判断该运算符是否可以计算，而且各运算符的计算次序也与扫描次序通常不一致，其原因就在于延迟缓冲，即仅根据表达式的前缀不足以确定各运算符的计算次序，只有获得足够的后续信息，才能确定其中哪些运算符可以执行。

解决方法就在于栈：我们可以利用栈来缓存当前无法判断是否可以优先计算的运算符，而对于一些局部就可以判断的运算符则马上进行计算。



以下是使用单个栈的实现实例：

问题及改进：用一个栈来保存数值和运算符显得比较混乱，更好的做法是使用两个栈分别保存数值和运算符，使得数值和运算符可以分别对待。

下面是以上讨论过程的算法实现：

1. 主算法：
   - 创建两个栈分别保存数值和运算符；
   - 首先将结束符"\0"存入运算符栈，作为铺垫；
   - 每次扫描一个字符，如果是数值则调用readNumber处理（将单位操作数入栈或连续的多位操作数组合后入栈）；如果是运算符则调用switch处理，但调用前先用orderBetween比较栈顶运算符和当前运算符的优先级；
   - 循环处理每个字符，直到运算符栈为空，返回数值栈中唯一的值，也就是表达式的计算结果。

```
❖ float evaluate( char* S, char* & RPN ) { //S保证语法正确
    Stack<float> opnd; Stack<char> optr; //运算数栈、运算栈
    optr.push('\0'); //铺垫
    while ( ! optr.empty() ) { //逐个处理各字符，直至运算符栈空
       if ( isdigit( *S ) ) //若为操作数（可能多位、小数），则
          readNumber( S, opnd ); //读入
       else //若为运算符，则视其与栈顶运算符之间优先级的高低
          switch( orderBetween( optr.top(), *S ) ) { /* 分别处理 */ }
    } //while
    return opnd.pop(); //弹出并返回最后的计算结果
 }
```

2. 栈顶运算符和当前运算符的优先级比较方式：通过预先制作优先级表，然后在算法中通过查表获得对应的优先级。orderBetween的作用就是查表并返回优先级。

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
   /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 运  ^ */  '>', '>', '>', '>', '>', '<', '<', '>', '>',
   /* 算  ! */  '>', '>', '>', '>', '>', '>', ' ', '>', '>',
   /* 符  ( */  '<', '<', '<', '<', '<', '<', '<', '=', ' ',
   /*  |  ) */  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
   /* -- \0 */  '<', '<', '<', '<', '<', '<', '<', ' ', '='
   //            +    -    *    /    ^    !    (    )    \0
   //             |-------------- 当前运算符 --------------|
};
```

3. switch函数的执行过程：

   - 当前运算符比栈顶运算符的优先级更大：将当前运算符入栈，转而处理下一个符号；
   - 当前运算符比栈顶运算符的优先级更小：取出栈顶运算符和操作数进行计算，将结果保存回数值栈中；
   - 当前运算符和栈顶运算符的优先级相等：说明栈顶运算符和当前运算符是括号或者结束符，此时说明括号内的表达式已经计算完毕，只需要将括号从栈中弹出即可。

```
❖switch( orderBetween( optr.top(), *S ) ) {
    case '<': //栈顶运算符优先级更低
       optr.push( *S ); S++; break; //计算推迟，当前运算符进栈
    case '=': //优先级相等（当前运算符为右括号，或尾部哨兵'\0'）
       optr.pop(); S++; break; //脱括号并接收下一个字符
    case '>': { //栈顶运算符优先级更高，实施相应的计算，结果入栈
       char op = optr.pop(); //栈顶运算符出栈，执行对应的运算
       if ( '!' == op ) opnd.push( calcu( op, opnd.pop() ) ); //一元运算符
       else {   float pOpnd2 = opnd.pop(), pOpnd1 = opnd.pop(); //二元运算符
                opnd.push( calcu( pOpnd1, op, pOpnd2 ) ); //实施计算，结果入栈
       } //为何不直接：opnd.push( calcu( opnd.pop(), op, opnd.pop() ) )?
       break;
    } //case '>'
 } //switch
```

> 只要后一个运算符的优先级没有前一个高（小于或等于），则前一个运算符就可以直接计算。
>
> 结束符的作用就相当于括号，因此算法开始前先将其进行入栈，以和末尾的结束符产生呼应。

逆波兰表达式

这是一个栈式计算的应用：中缀表达式的计算需要考虑运算符之间的优先级，只有优先级高的运算符才能优先计算，而且除了运算符本身具有优先级外，还必须处理括号所强制指定的优先级关系。这种带有运算优先级的计算方式使得表达式的计算比较混乱，过程也显得复杂，因此就有了逆波兰表达式（RPN）。

逆波兰表达式也是由一串字符组成的式子，但其中不包含任何括号，也没有运算符之间的优先级限制。也就是说，逆波兰表达式中任何运算符的计算次序均决定于它自身在表达式中的出现顺序，即谁先出现，谁就先计算。理论上，任何中缀表达式都可以转化为逆波兰表达式（后缀式）。

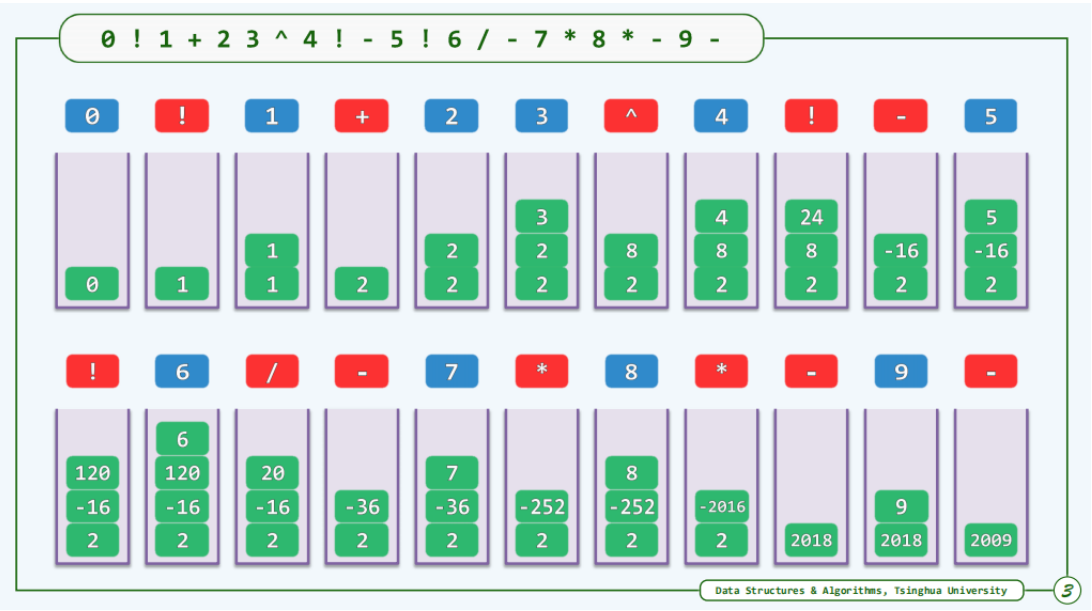
```
中缀表达式：  0 ! + 123 + 4 * ( 5 * 6 ! + 7 ! / 8 ) / 9
逆波兰表达式：0 ! 123 + 4 5 6 ! * 7 ! 8 / + * 9 / +
```

从例子中也可以看见，为了区分各个操作数，RPN也额外引入了一个起分隔作用的元字符（比如空格）。先看一个RPN的求值过程：

- 运算逻辑：

```
❖ 引入栈S //用以存放操作数
  逐个处理下一元素x
    if ( x是操作数 ) 将x压入S
    else //x是运算符（无需缓冲）
      从S中弹出x所需数目的操作数
      执行相应的计算，结果压入S //无需顾及优先级！
  返回栈顶
❖ 只要输入的RPN语法正确，此时的栈顶亦是栈底，对应于最终的计算结果
```

- 一个实例：



预处理：如何将一个中缀表达式转换为逆波兰表达式呢？

1. 手工转换

```
❖ 例如：( 0 ! + 1 ) ^ ( 2 * 3 ! + 4 - 5 )
   1) 用括号显式地表示优先级
        { ( [ 0 ! ]  + 1 )  ^ ( [ ( 2 * [ 3 ! ]  )  + 4 ]  - 5 )  }
   2) 将运算符移到对应的右括号后
        { ( [ 0  ] !  1 ) +  ( [ ( 2  [ 3  ] ! ) *  4 ] +  5 ) - } ^
   3) 抹去所有括号
             0   !  1  +      2   3   !  *  4  +  5  -  ^
   4) 稍事整理，即得
             0 ! 1 + 2 3 ! * 4 + 5 - ^
```

2. 自动转换

   这个转换过程和中缀表达式的求值算法相似，仅仅改了两点：① 遇到操作数时原封不动地添加到RPN尾部；② 当栈顶运算符可以执行时，将其追加到RPN尾部。

```
❖ float evaluate( char* S, char* & RPN ) { //RPN转换
    /* ..................................... */
    while ( ! optr.empty() ) { //逐个处理各字符，直至运算符栈空
       if ( isdigit( * S ) ) //若当前字符为操作数，则直接
          { readNumber( S, opnd ); append( RPN, opnd.top() ); } //将其接入RPN
       else //若当前字符为运算符
          switch( orderBetween( optr.top(), *S ) ) {
             /* ..................................... */
             case '>': { //且可立即执行，则在执行相应计算的同时
                char op = optr.pop(); append( RPN, op ); //将其接入RPN
                /* ..................................... */
             } //case '>'
          /* ..................................... */
```

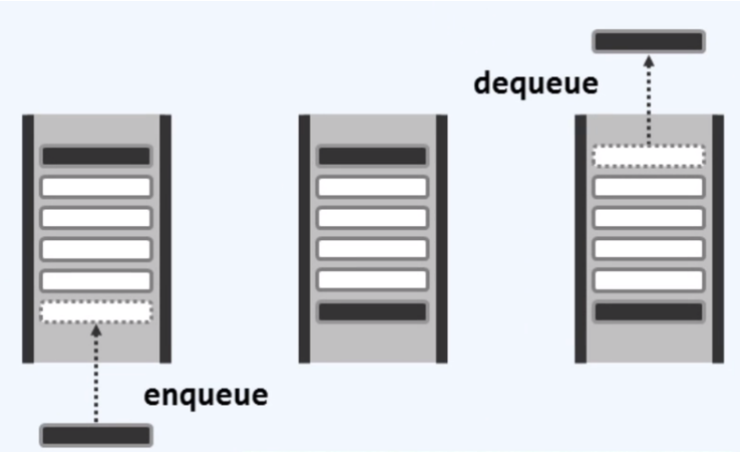
一句话总结

栈通常用于解决序列长度未知的问题中。

队列

什么是队列

也是一个线性的受限序列，只能在序列两端进行访问：只能在一端（队尾）插入元素，而在另一端（队头）获取元素，即一端进，另一端出。这种结构的特点就是先进先出（FIFO）或后进后出（LILO）。



队列的接口

❖ 只能在队尾插入（查询）：

- enqueue() / rear()

❖ 只能在队头删除（查询）：

- dequeue() / front()

队列的操作实例

实例

操作	输出	队列（右侧为队头）			
Queue()					
empty()	true				
enqueue(5)		5			
enqueue(3)		3	5		
dequeue()	5	3			
enqueue(7)		7	3		
enqueue(3)		3	7	3	
front()	3	3	7	3	
empty()	false	3	7	3	

操作	输出	队列（右侧为队头）				
enqueue(11)		11	3	7	3	
size()	4	11	3	7	3	
enqueue(6)		6	11	3	7	3
empty()	false	6	11	3	7	3
enqueue(7)		7	6	11	3	7	3
dequeue()	3	7	6	11	3	7	
dequeue()	7	7	6	11	3		
front()	3	7	6	11	3		
size()	4	7	6	11	3		

队列的实现

实现

- 队列既然属于序列的特例，故亦可直接基于向量或列表派生
- template <typename T> class Queue: public List<T> { //由列表派生的队列模板类

```
public: //size()与empty()直接沿用
   void enqueue( T const & e ) { insertAsLast( e ); } //入队
   T dequeue() { return remove( first() ); } //出队
   T & front() { return first()->data; } //队首
}; //以列表首/末端为队列头/尾——颠倒过来呢？
```

- 确认：如此实现的队列接口，均只需$O(1)$时间