

2018年10月31日

栈是一种后进先出(LIFO)的数据结构，如同擦书本一样，最先放的书本是最后才会拿到：

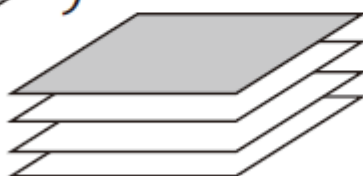
a stack of documents



`push(`



`)`



new (gray) one goes on top

`push(`

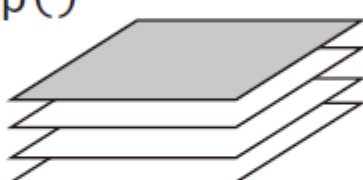


`)`



new (black) one goes on top

`= pop()`



remove the black one from the top

`= pop()`



remove the gray one from the top

栈的数组实现

```

1 public class ArrayStack<Item> implements Iterable<Item> {
2     private Item[] a = (Item[]) new Object[1];
3     private int N;
4
5     public boolean isEmpty() {
6         return N == 0;
7     }
8
9     public int size() {
10        return N;
11    }
12
13    public void push(Item item) {
14        if (N == a.length) {
15            resize(2 * a.length);
16        }
17        a[N++] = item;
18    }
19
20    public Item pop() {
21        if (isEmpty()) {
22            throw new NoSuchElementException("Stack underflow");
23        }
24        Item item = a[--N];
25        a[N] = null;
26        if (N > 0 && N == a.length / 4) {
27            resize(a.length / 2);
28        }
29        return item;
30    }
31
32    private void resize(int max) {
33        Item[] temp = (Item[]) new Object[max];
34        for (int i = 0; i < N; i++) {
35            temp[i] = a[i];
36        }
37        a = temp;
38    }
39
40    @Override
41    public Iterator<Item> iterator() {
42        return new ReverseArrayIterator();

```

```
43     }
44
45     private class ReverseArrayIterator implements Iterator<Item> {
46         private int i = N;
47
48         @Override
49         public boolean hasNext() {
50             return i > 0;
51         }
52
53         @Override
54         public Item next() {
55             if (!hasNext()) {
56                 throw new NoSuchElementException();
57             }
58             return a[--i];
59         }
60
61         @Override
62         public void remove() {
63
64         }
65     }
66 }
```

以上的实现中当栈中容量与数组容量相等时，会进行扩容；当栈中容量为数组容量四分之一时，会进行缩容，具体操作如下图：

push()	pop()	N	a.length	a[]							
				0	1	2	3	4	5	6	7
		0	1	null							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	null				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	null	null	null
-	to	4	8	to	be	or	not	null	null	null	null
be		5	8	to	be	or	not	be	null	null	null
-	be	4	8	to	be	or	not	null	null	null	null
-	not	3	8	to	be	or	null	null	null	null	null
that		4	8	to	be	or	that	null	null	null	null
-	that	3	8	to	be	or	null	null	null	null	null
-	or	2	4	to	be	null	null				
-	be	1	2	to	null						
is		2	2	to	is						

- **最好时间复杂度**：最理想的情况下，当前栈中元素数量比数组的容量小，此时就直接执行代码块 `a[N++] = item;`，即此时的时间复杂度为 $O(1)$ 。
- **最坏时间复杂度**：最糟糕的情况下，当前栈中元素数量与数组的容量相等，此时就要执行 `resize` 方法进行扩容了，进入循环体，执行 `N` 次复制操作，此时的时间复杂度为 $O(N)$ 。
- **平均时间复杂度**：
 - 当栈中元素小于数组容量时，此时进行压栈就有 `N` 种情况，且每种情况的时间复杂度为 $O(1)$ ；当栈中元素与数组容量相等时，此时进行压栈就只有一种情况了，要进行扩容操作，这种情况的时间复杂度为 $O(N)$ ；则总共有 `N+1` 中情况，对其取平均值：

$$\frac{1 + 1 + 1 + \dots + 1 + N}{N + 1} = \frac{2N}{N + 1}$$
 在大 O 标记法中，可以省略系数与低阶项，所以其平均时间复杂度为 $O(1)$
 - 下面使用概率来分析，由于有 `N+1` 中情况，每种情况的发生概率为 $\frac{1}{N+1}$ ，则其平均时间复杂度为：

$$1 \times \frac{1}{N+1} + 1 \times \frac{1}{N+1} + \dots + 1 \times \frac{1}{N+1} + N \times \frac{1}{N+1} = O(1)$$
- **均摊时间复杂度**：根据上述代码，每出现一次扩容操作时，即此时压栈的时间复杂度为 $O(N)$ ，那么后面的 `N` 次压栈操作的时间复杂度均为 $O(1)$ ，前后是连贯的，因此将 $O(N)$ 平摊到前 `N` 次上，得出均摊时间复杂度为 $O(1)$ 。

栈的链表实现

```

1 public class ListStack<Item> implements Iterable<Item> {
2     private Node first;
3     private int N;

```

```

4
5     private class Node {
6         Item item;
7         Node next;
8     }
9
10    public void push(Item item) {
11        Node oldFirst = first;
12        first = new Node();
13        first.item = item;
14        first.next = oldFirst;
15        N++;
16    }
17
18    public Item pop() {
19        if (isEmpty()) {
20            throw new NoSuchElementException();
21        }
22        Item item = first.item;
23        first = first.next;
24        N--;
25        return item;
26    }
27
28    public Item peek() {
29        if (isEmpty()) {
30            throw new NoSuchElementException();
31        }
32        return first.item;
33    }
34
35    public boolean isEmpty() {
36        return first == null;
37    }
38
39    public int size() {
40        return N;
41    }
42
43    @Override
44    public Iterator<Item> iterator() {
45        return new ListIterator();

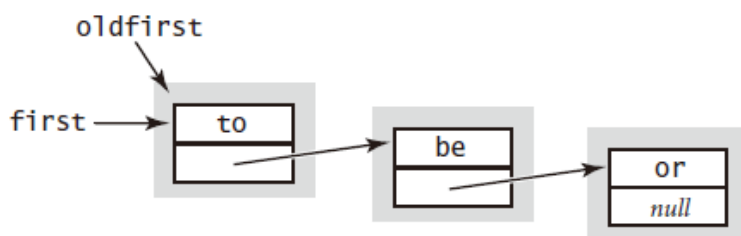
```

```
46     }
47
48     private class ListIterator implements Iterator<Item> {
49         private Node current = first;
50
51         @Override
52         public boolean hasNext() {
53             return current != null;
54         }
55
56         @Override
57         public Item next() {
58             Item item = current.item;
59             current = current.next;
60             return item;
61         }
62
63         @Override
64         public void remove() {
65
66         }
67     }
68 }
```

push 操作如图:

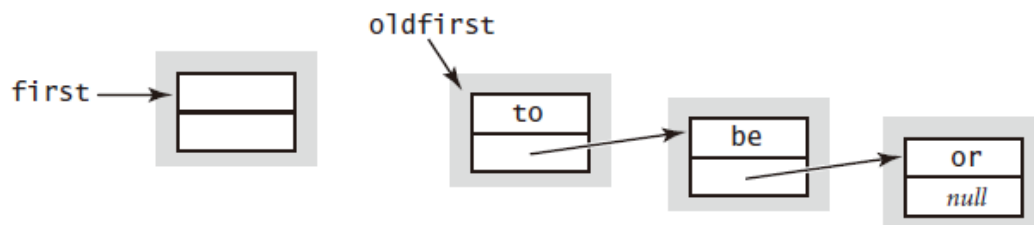
save a link to the list

```
Node oldfirst = first;
```



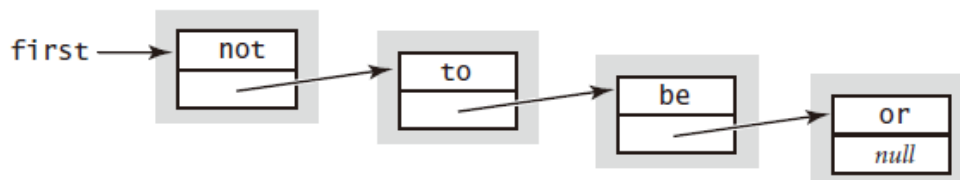
create a new node for the beginning

```
first = new Node();
```

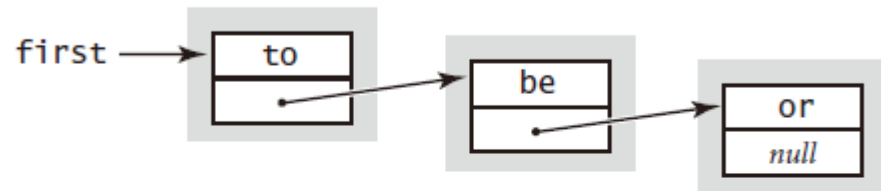


set the instance variables in the new node

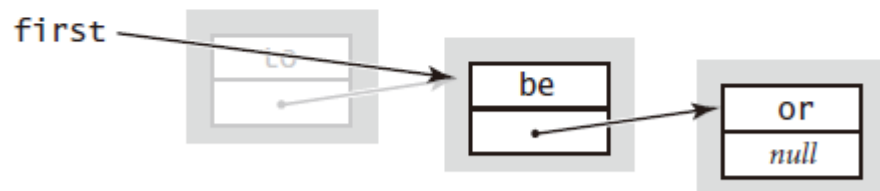
```
first.item = "not";  
first.next = oldfirst;
```




```
first = first.next;
```

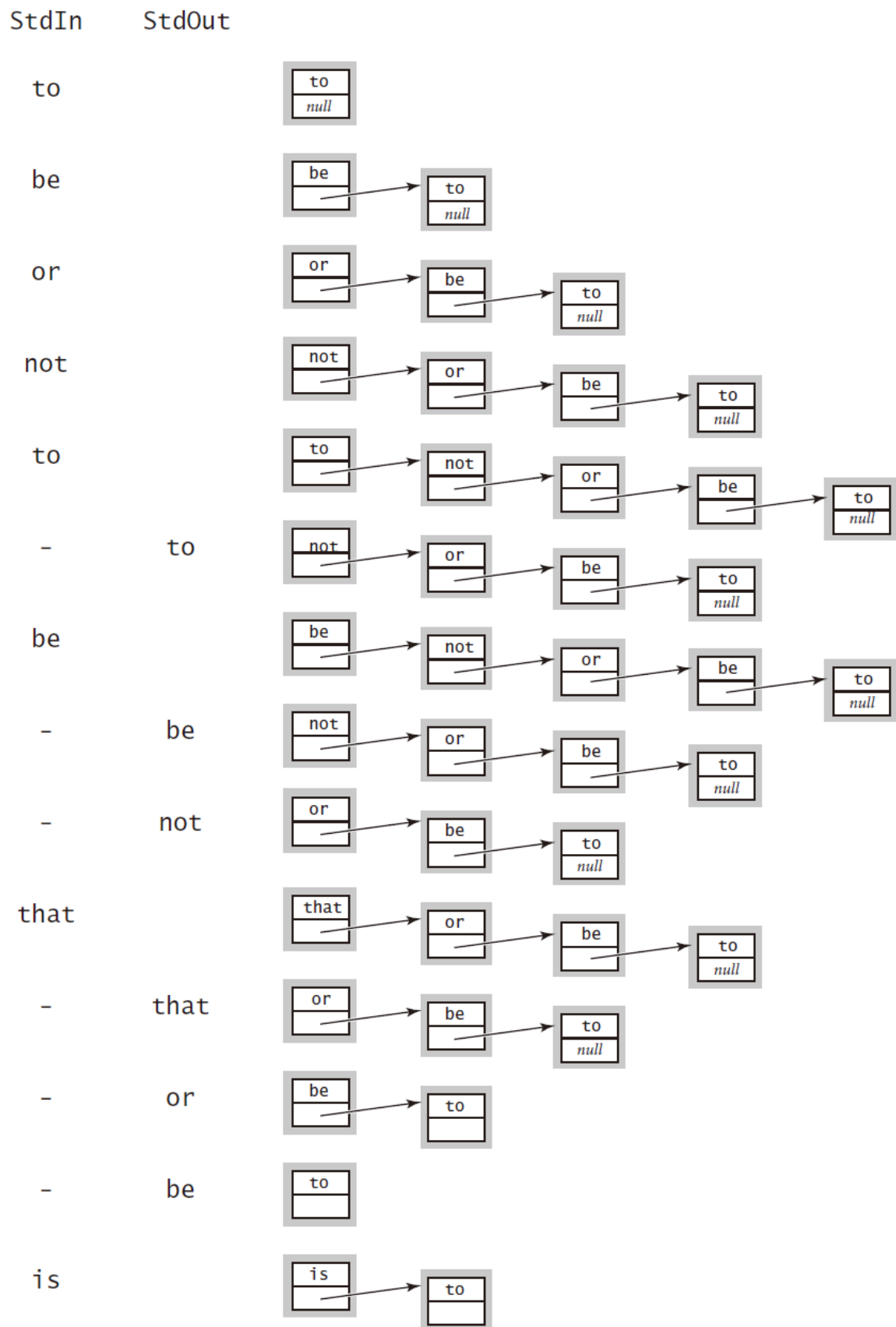


pop 操作如图：



Removing the first node in a linked list

整体操作如下：



栈的应用

1, 括号匹配

每个左括号必然对应其右括号，如 `[()]` 就是合法的，而 `[()]` 就是错误的；这个程序可以用栈来实现：

定义一个空栈，读入字符，如果字符是一个左括号，则将其压入栈中。如果字符是一个右括号，则当栈空时报错，否则，将栈顶元素弹出，如果弹出的字符不是对应的左括号，则报错。若全部字符读完后，栈不为空则报错。

```
1      public static boolean isComplete(String str) {
2          if (str.length() == 0) {
3              return false;
4          }
5          Deque<Character> sta = new ArrayDeque<>();
6          for (int i = 0; i < str.length(); i++) {
7              if (str.charAt(i) == '[') {
8                  sta.push('[');
9              }
10             if (str.charAt(i) == '(') {
11                 sta.push('(');
12             }
13             if (str.charAt(i) == '{') {
14                 sta.push('{');
15             }
16             if (str.charAt(i) == ']') {
17                 if (sta.isEmpty()) {
18                     return false;
19                 }
20                 if (sta.pop() != '[') {
21                     return false;
22                 }
23             }
24             if (str.charAt(i) == ')') {
25                 if (sta.isEmpty()) {
26                     return false;
27                 }
28                 if (sta.pop() != '(') {
29                     return false;
30                 }
31             }
32             if (str.charAt(i) == '}') {
```

```
33         if (sta.isEmpty()) {  
34             return false;  
35         }  
36         if (sta.pop() != '{') {  
37             return false;  
38         }  
39     }  
40 }
```

2, 后缀(逆波兰)表达式

中缀表达式就是我们常见的表达式，如： $6 * (5 + (2 + 3) * 8 + 3)$

而后缀表达式的表示形式为： $6\ 5\ 2\ 3\ +\ 8\ *\ +3\ +\ *$

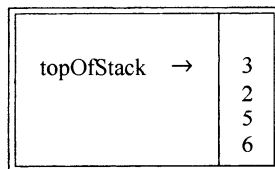
前缀表达式的表示形式为： $*6\ +\ +5\ *\ +2\ 3\ 8\ 3$

在计算机中，利用后缀表达式进行计算没有必要知道任何优先规则，使用后缀表达式计算的过程及其程序：

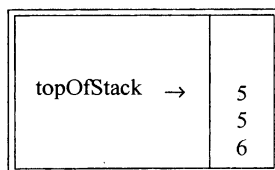
后缀表达式

6 5 2 3 + 8 * + 3 + *

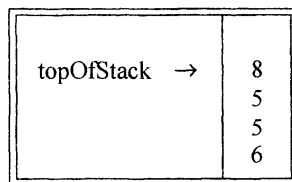
计算如下：前四个字符放入栈中，此时栈变成



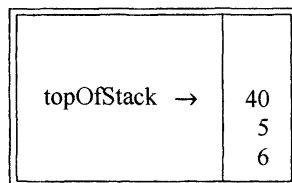
下面读到一个‘+’号，所以3和2从栈中弹出并且它们的和5被压入栈中



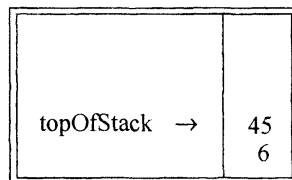
接着，8进栈



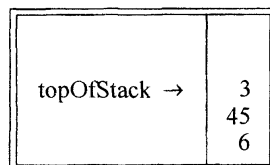
现在见到一个‘*’号，因此8和5弹出并且 $5 * 8 = 40$ 进栈



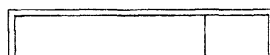
接着又见到一个‘+’号，因此40和5被弹出并且 $5 + 40 = 45$ 进栈

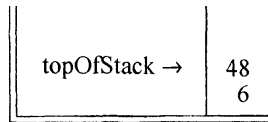


现在将3压入栈中

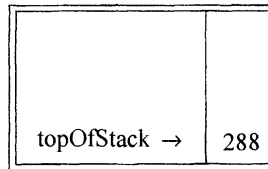


然后‘+’使得3和45从栈中弹出并将 $45 + 3 = 48$ 压入栈中





最后，遇到一个‘ * ’号，从栈中弹出 48 和 6；将结果 $6 * 48 = 288$ 压进栈中



```

1  private static Pattern ISNUMBER = Pattern.compile("[0-9]+");
2
3  /**
4   * 计算后缀表达式: 2 3 * 2 1 - / 3 4 1 - * +
5   * 其中缀表达式: 2 * 3 / ( 2 - 1 ) + 3 * ( 4 - 1 )
6   * 将数字压栈，一遇到运算符就将其取出运算，结果再压入栈
7   */
8  public static void evaluatePostFix() {
9      String str = "2 3 * 2 1 - / 3 4 1 - * +";
10     String[] strings = str.split("\\s+");
11     Stack<Integer> sta = new Stack<>();
12     for (String s : strings) {
13         if (ISNUMBER.matcher(s).matches()) {
14             sta.push(Integer.parseInt(s));
15         } else {
16             int n1 = sta.pop();
17             int n2 = sta.pop();
18             int n3 = 0;
19             if (s.equals("+")) {
20                 n3 = n2 + n1;
21             } else if (s.equals("-")) {
22                 n3 = n2 - n1;
23             } else if (s.equals("*")) {
24                 n3 = n2 * n1;
25             } else if (s.equals("/")) {
26                 n3 = n2 / n1;
27             }
28             sta.push(n3);
29         }
30     }
31     System.out.println(sta.pop());
32 }

```

前缀式求值是先将前缀式逆序，后通过后缀式求值的方法，求值。但要注意的是操作数的运算顺序是与后缀表达式相反的。举个例子：中缀表达式 $8-7$ ；后缀表达式为 $87-$ ，前缀表达式为 -87 ，前缀表达式逆序 $78-$ ，左操作数为 8 ，右操作数为 7

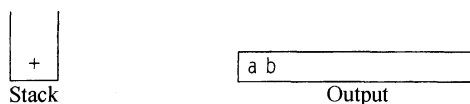
```
1 private static Pattern ISNUMBER = Pattern.compile("[0-9]+");
2 /**
3  * 计算前缀表达式: + / * 2 3 - 2 1 * 3 - 4 1
4  * 其中缀表达式为: 2 * 3 / ( 2 - 1 ) + 3 * ( 4 - 1 )
5  * 先将其反转, 将数字压入栈, 一遇到运算符就取出数字计算, 将计算结果压入栈
6  */
7 public static void evaluatePreFix() {
8     String str = "+ / * 2 3 - 2 1 * 3 - 4 1";
9     String[] strings = str.split("\\s+");
10    for (int i = 0; i < strings.length / 2; i++) {
11        String temp = strings[i];
12        strings[i] = strings[strings.length - i - 1];
13        strings[strings.length - i - 1] = temp;
14    }
15    Stack<Integer> sta = new Stack<>();
16    for (String s : strings) {
17        if (ISNUMBER.matcher(s).matches()) {
18            sta.push(Integer.parseInt(s));
19        } else {
20            int n1 = sta.pop();
21            int n2 = sta.pop();
22            int n3 = 0;
23            if (s.equals("+")) {
24                n3 = n1 + n2;
25            } else if (s.equals("-")) {
26                n3 = n1 - n2;
27            } else if (s.equals("*")) {
28                n3 = n1 * n2;
29            } else if (s.equals("/")) {
30                n3 = n1 / n2;
31            }
32            sta.push(n3);
33        }
34    }
35    System.out.println(sta.pop());
36 }
```

如何将中缀表达式转化为后缀表达式？ 假设中缀表达式： $a + b * c + (d * e + f) * g$

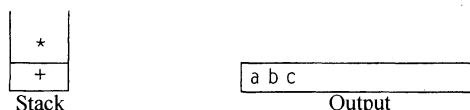
后缀表达式为： $abc * + de * f + g * +$

过程如下:

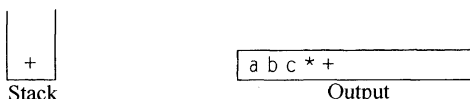
为了理解这种算法的运行机制,我们将把上面长的中缀表达式转换成后缀形式。首先,符号 a 被读入,于是它被传向输出。然后, '+' 被读入并被放入栈中。接下来 b 读入并流向输出。这一时刻的状态如下:



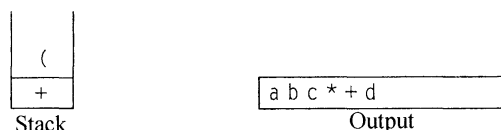
接着 * 号被读入。操作符栈的栈顶元素比 * 的优先级低,故没有输出且 * 进栈。接着, c 被读入并输出。至此,我们有



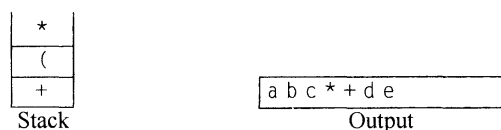
后面的符号是一个 + 号。检查一下栈我们发现,需要将 * 从栈弹出并把它放到输出中;弹出栈中剩下的 + 号,该算符不比刚刚遇到的 + 号优先级低而是有相同的优先级;然后,将刚刚遇到的 + 号压入栈中



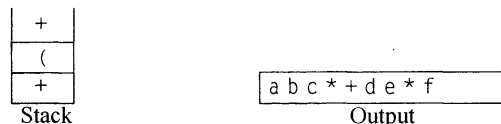
下一个被读到的符号是一个 (, 由于有最高的优先级,因此它被放进栈中。然后, d 读入并输出



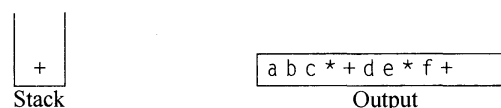
继续进行,我们又读到一个 *。由于除非正在处理闭括号否则开括号不会从栈中弹出,因此没有输出。下一个是 e, 它被读入并输出



再往后读到的符号是 +。我们将 * 弹出并输出,然后将 + 压入栈中。这以后,我们读到 f 并输出



现在,我们读到一个), 因此将栈元素直到(弹出,我们将一个 + 号输出



下面又读到一个 *; 该算符被压入栈中。然后, g 被读入并输出




```

32         "(" .equals(sta.peek())) {
33             sb.append(sta.pop());
34         }
35         sta.push(s);
36         break;
37         case "*":
38         case "/":
39             while (!sta.isEmpty() &&
40 ("+" .equals(sta.peek())) &&
41 ("-" .equals(sta.peek())) && (!"
42 "(" .equals(sta.peek())) {
43                 sb.append(sta.pop()+" ");
44             }
45             sta.push(s);
46             break;
47             case "+":
48             case "-":
49                 while (!sta.isEmpty() && (!"
50 "(" .equals(sta.peek())) {
51                     sb.append(sta.pop() + " ");
52                 }
53                 sta.push(s);
54                 break;
55             default:
56             }
57         }
58     }
59     while (!sta.isEmpty()) {
60         sb.append(sta.pop()+" ");
61     }
62     System.out.println(sb.toString());
63 }

```

将中缀表达式转为前缀表达式，要先将其反转，当栈中数据全部弹出后，再将其反转即可：

```

1  /**
2   * 将中序表达式转为前序表达式
3   * 首先将其反转，将运算符压栈，遇到（就弹出
4   * 结果为：+/*23-21*3-41
5   */
6  public static void infixToPrefix() {
7      String str = "2 * 3 / ( 2 - 1 ) + 3 * ( 4 - 1 )";

```

```

8
9     string[] strings = str.split("\\s+");
10    for (int i = 0; i < strings.length / 2; i++) {
11        string temp = strings[i];
12        strings[i] = strings[strings.length - i - 1];
13        strings[strings.length - i - 1] = temp;
14    }
15    System.out.println(Arrays.toString(strings));
16    Stack<String> sta = new Stack<>();
17    StringBuffer sb = new StringBuffer();
18    for (String s : strings) {
19        if (ISNUMBER.matcher(s).matches()) {
20            sb.append(s + " ");
21        } else {
22            switch (s) {
23                case "(":
24                    while (!sta.isEmpty() &&
25(!")".equals(sta.peek())) {
26                        sb.append(sta.pop() + " ");
27                    }
28                    sta.pop();
29                    break;
30                case ")":
31                    sta.push(s);
32                    break;
33                case "^":
34                    while (!sta.isEmpty() && (!
35("^".equals(sta.peek()) ||
36                    "(" .equals(sta.peek())))) {
37                        sb.append(sta.pop());
38                    }
39                    sta.push(s);
40                    break;
41                case "*":
42                case "/":
43                    sta.push(s);
44                    break;
45                case "+":
46                case "-":
47                    while (!sta.isEmpty() &&
48(!")".equals(sta.peek())) {
49                        sb.append(sta.pop() + " ");

```

```
47         }
48         sta.push(s);
49         break;
50     default:
51     }
52 }
53 }
54 while (!sta.isEmpty()) {
55     sb.append(sta.pop() + " ");
56 }
57 System.out.println(sb.reverse().toString());
58 }
```