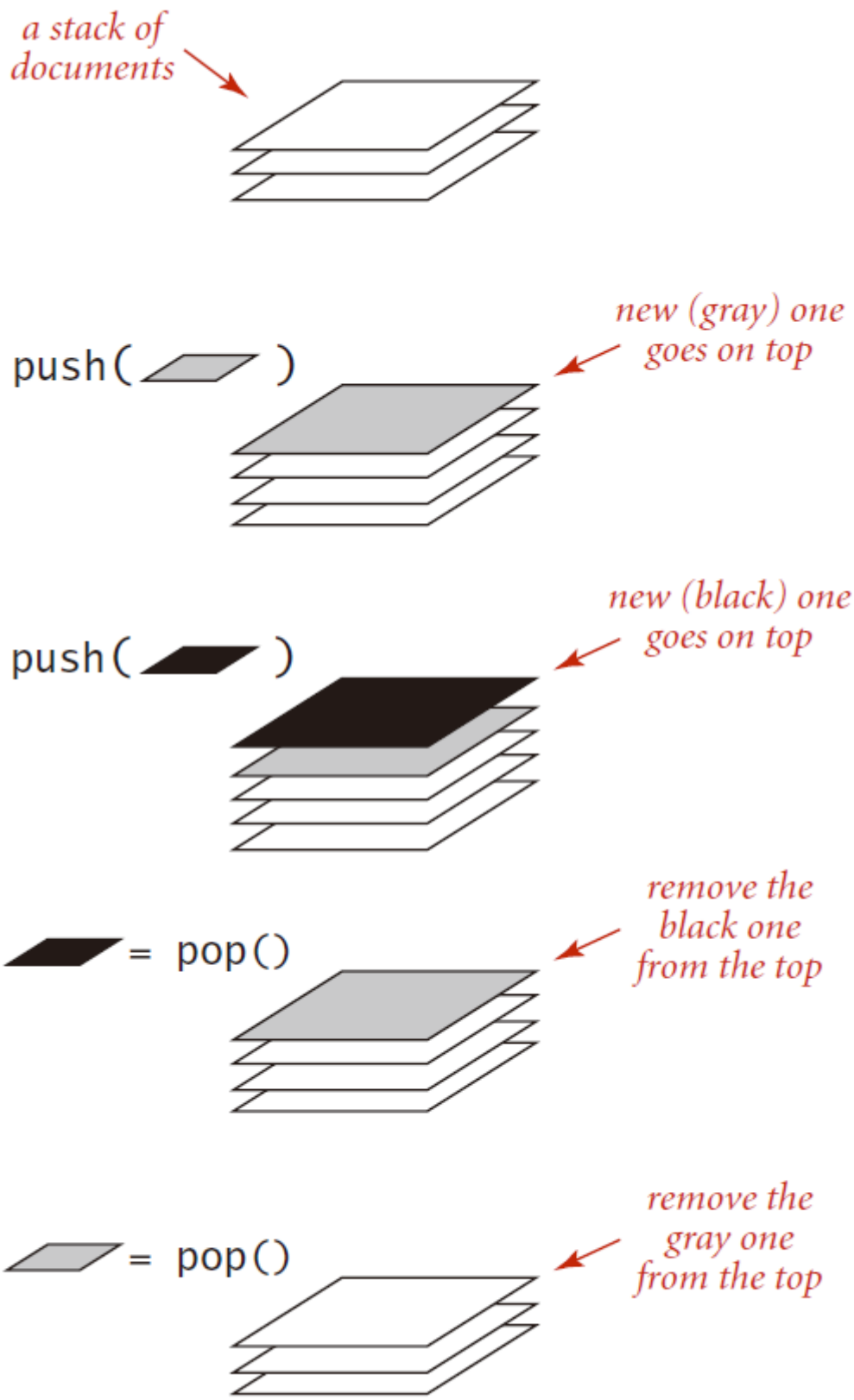
> 2018年10月31日

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



# 栈的数组实现

```java
public class ArrayStack<Item> implements Iterable<Item> {
    private Item[] a = (Item[]) new Object[1];
    private int N;

    public boolean isEmpty() {
        return N == 0;
    }

    public int size() {
        return N;
    }

    public void push(Item item) {
        if (N == a.length) {
            resize(2 * a.length);
        }
        a[N++] = item;
    }

    public Item pop() {
        if (isEmpty()) {
            throw new NoSuchElementException("Stack underflow");
        }
        Item item = a[--N];
        a[N] = null;
        if (N > 0 && N == a.length / 4) {
            resize(a.length / 2);
        }
        return item;
    }

    private void resize(int max) {
        Item[] temp = (Item[]) new Object[max];
        for (int i = 0; i < N; i++) {
            temp[i] = a[i];
        }
        a = temp;
    }

    @Override
    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i = N;

        @Override
        public boolean hasNext() {
            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：
    $\frac{1+1+1+\ldots+1+N}{N+1}=\frac{2N}{N+1}$ 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：$1 \times \frac{1}{N+1} + 1 \times \frac{1}{N+1} + \ldots + 1 \times \frac{1}{N+1} + N \times \frac{1}{N+1} = O(1)$

- **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

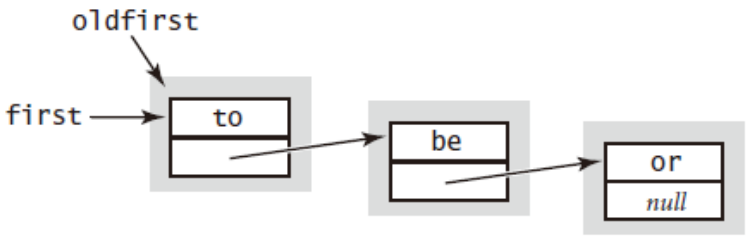
```
public class ListStack<Item> implements Iterable<Item> {
    private Node first;
    private int N;

    private class Node {
        Item item;
        Node next;
    }

    public void push(Item item) {
        Node oldFirst = first;
        first = new Node();
        first.item = item;
        first.next = oldFirst;
        N++;
    }

    public Item pop() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        Item item = first.item;
        first = first.next;
        N--;
        return item;
    }

    public Item peek() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        return first.item;
    }

    public boolean isEmpty() {
        return first == null;
    }

    public int size() {
        return N;
    }

    @Override
```

```
    public Iterator<Item> iterator() {
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Item> {
        private Node current = first;

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }

        @Override
        public void remove() {

        }
    }
}
```

`push`操作如图：

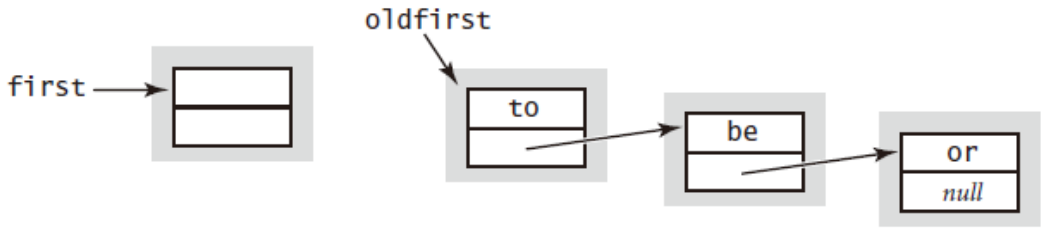
save a link to the list

```
Node oldfirst = first;
```



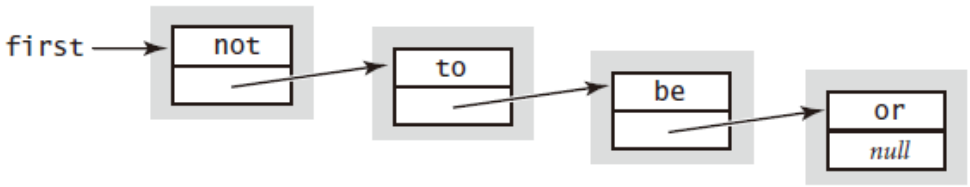

create a new node for the beginning

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

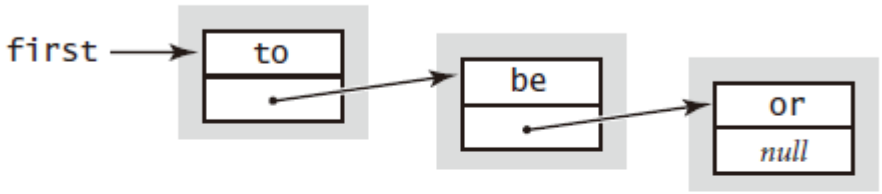



set the instance variables in the new node

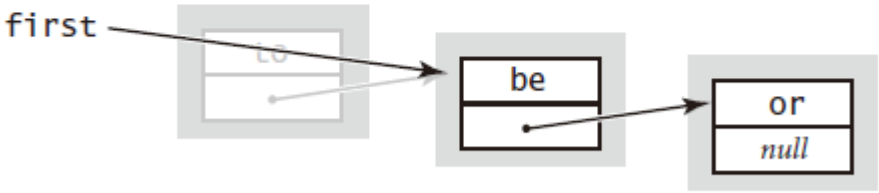
```
first.item = "not";
first.next = oldfirst;
```



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

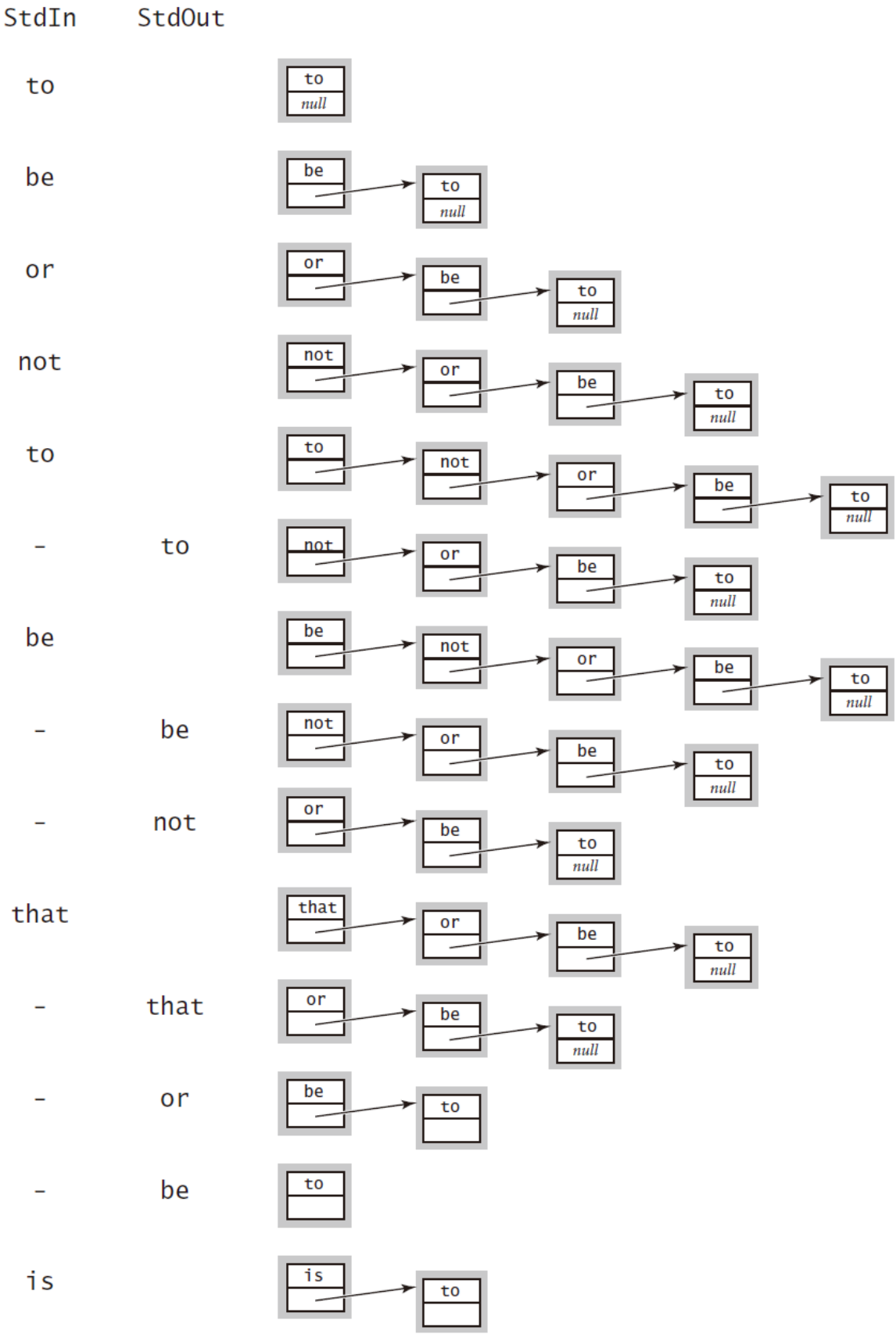



pop 操作如图：




**Removing the first node in a linked list**

整体操作如下：


StdIn StdOut
to
be
or
not
to
- to
be
- be
- not
that
- that
- or
- be
is
to
null
be
to
null
or
be
to
null
not
or
be
to
null
to
not
or
be
to
null
not
or
be
to
null
be
not
or
be
to
null
not
or
be
to
null
or
be
to
null
that
or
be
to
null
or
be
to
null
be
to
to
is
to

# 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。

```
    public static boolean isComplete(String str) {
        if (str.length() == 0) {
            return false;
        }
        Deque<Character> sta = new ArrayDeque<>();
        for (int i = 0; i < str.length(); i++) {
            if (str.charAt(i) == '[') {
                sta.push('[');
            }
            if (str.charAt(i) == '(') {
                sta.push('(');
            }
            if (str.charAt(i) == '{') {
                sta.push('{');
            }
            if (str.charAt(i) == ']') {
                if (sta.isEmpty()) {
                    return false;
                }
                if (sta.pop() != '[') {
                    return false;
                }
            }
            if (str.charAt(i) == ')') {
                if (sta.isEmpty()) {
                    return false;
                }
                if (sta.pop() != '(') {
                    return false;
                }
            }
            if (str.charAt(i) == '}') {
                if (sta.isEmpty()) {
                    return false;
                }
                if (sta.pop() != '{') {
                    return false;
                }
            }
        }
```

# 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：$6\ 5\ 2\ 3+8*+3+*$

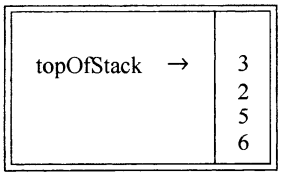
前缀表达式的表示形式为：$*6++5*+2\ 3\ 8\ 3$

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：

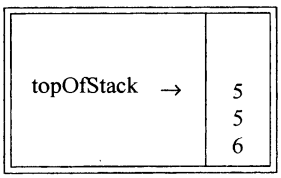
后缀表达式

6 5 2 3 + 8 * + 3 + *

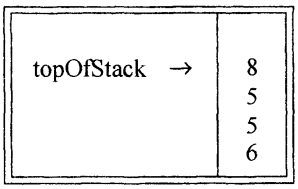
计算如下：前四个字符放入栈中，此时栈变成



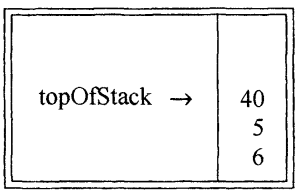

下面读到一个‘ + ’号，所以 3 和 2 从栈中弹出并且它们的和 5 被压入栈中



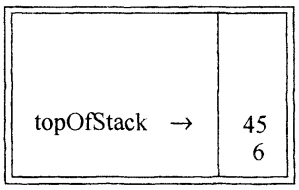

接着，8 进栈



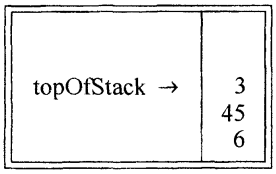

现在见到一个‘ * ’号，因此 8 和 5 弹出并且 5 * 8 = 40 进栈



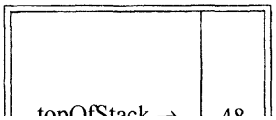

接着又见到一个‘ + ’号，因此 40 和 5 被弹出并且 5 + 40 = 45 进栈

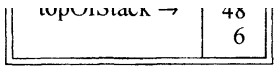



现在将 3 压入栈中

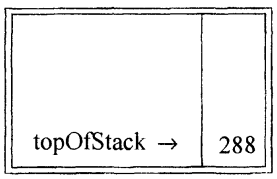



然后‘ + ’使得 3 和 45 从栈中弹出并将 45 + 3 = 48 压入栈中






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




```
    private static Pattern ISNUMBER = Pattern.compile("[0-9]+");

    /**
     * 计算后缀表达式: 2 3 * 2 1 - / 3 4 1 - * +
     * 其中缀表达式: 2 * 3 / ( 2 - 1 ) + 3 * ( 4 - 1 )
     * 将数字压栈，一遇到运算符就将其取出运算，结果再压入栈
     */
    public static void evaluatePostFix() {
        String str = "2 3 * 2 1 - / 3 4 1 - * +";
        String[] strings = str.split("\\s+");
        Stack<Integer> sta = new Stack<>();
        for (String s : strings) {
            if (ISNUMBER.matcher(s).matches()) {
                sta.push(Integer.parseInt(s));
            } else {
                int n1 = sta.pop();
                int n2 = sta.pop();
                int n3 = 0;
                if (s.equals("+")) {
                    n3 = n2 + n1;
                } else if (s.equals("-")) {
                    n3 = n2 - n1;
                } else if (s.equals("*")) {
                    n3 = n2 * n1;
                } else if (s.equals("/")) {
                    n3 = n2 / n1;
                }
                sta.push(n3);
            }
        }
        System.out.println(sta.pop());
    }
```

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

```
    private static Pattern ISNUMBER = Pattern.compile("[0-9]+");
    /**
     * 计算前缀表达式: + / * 2 3 - 2 1 * 3 - 4 1
     * 其中缀表达式为: 2 * 3 / ( 2 - 1 ) + 3 * ( 4 - 1 )
     * 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
```

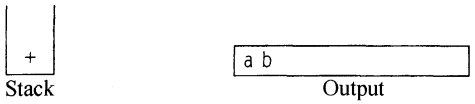
```java
     */
    public static void evaluatePreFix() {
        String str = "+ / * 2 3 - 2 1 * 3 - 4 1";
        String[] strings = str.split("\\s+");
        for (int i = 0; i < strings.length / 2; i++) {
            String temp = strings[i];
            strings[i] = strings[strings.length - i - 1];
            strings[strings.length - i - 1] = temp;
        }
        Stack<Integer> sta = new Stack<>();
        for (String s : strings) {
            if (ISNUMBER.matcher(s).matches()) {
                sta.push(Integer.parseInt(s));
            } else {
                int n1 = sta.pop();
                int n2 = sta.pop();
                int n3 = 0;
                if (s.equals("+")) {
                    n3 = n1 + n2;
                } else if (s.equals("-")) {
                    n3 = n1 - n2;
                } else if (s.equals("*")) {
                    n3 = n1 * n2;
                } else if (s.equals("/")) {
                    n3 = n1 / n2;
                }
                sta.push(n3);
            }
        }
        System.out.println(sta.pop());
    }
```

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

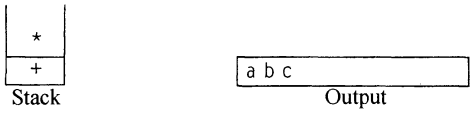
后缀表达式为： $abc*+de*f+g*+$

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：

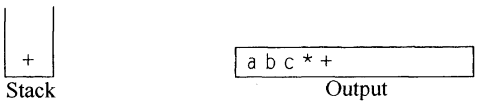



接着 * 号被读入。操作符栈的栈顶元素比 * 的优先级低，故没有输出且 * 进栈。接着，c 被读入并输出。至此，我们有

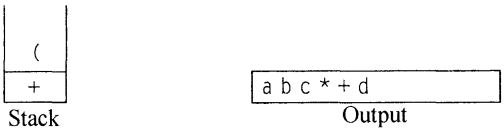



]

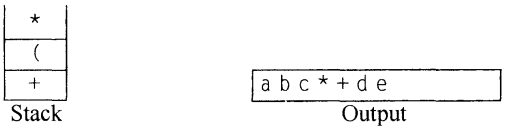
后面的符号是一个 + 号。检查一下栈我们发现，需要将 * 从栈弹出并把它放到输出中；弹出栈中剩下的 + 号，该算符不比刚刚遇到的 + 号优先级低而是有相同的优先级；然后，将刚刚遇到的 + 号压入栈中



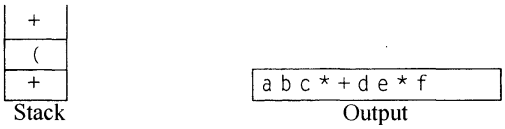

下一个被读到的符号是一个(，由于有最高的优先级，因此它被放进栈中。然后，d 读入并输出



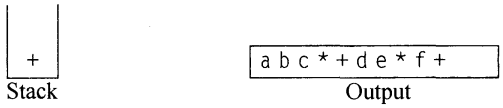

继续进行，我们又读到一个 * 。由于除非正在处理闭括号否则开括号不会从栈中弹出，因此没有输出。下一个是 e，它被读入并输出



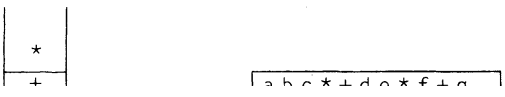

再往后读到的符号是 + 。我们将 * 弹出并输出，然后将 + 压入栈中。这以后，我们读到 f 并输出

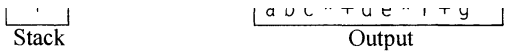



现在，我们读到一个)，因此将栈元素直到(弹出，我们将一个 + 号输出

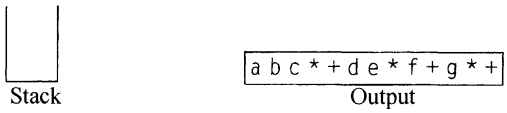



下面又读到一个 * ；该算符被压入栈中。然后，g 被读入并输出






现在输入为空，因此我们将栈中的符号全部弹出并输出，直到栈变成空栈




程序如下：

```
    /**
     * 利用正则表达式判断是否为整数
     * 以下只能匹配非负整数
     */
    private static Pattern ISNUMBER = Pattern.compile("[0-9]+");

    /**
     * 将中缀表达式转为后缀表达式
     * 结果为: 2 3 * 2 1 - / 3 4 1 - * +
     */
    public static void infixToPostfix() {
        String str = "2 * 3 / ( 2 - 1 ) + 3 * ( 4 - 1 )";
        String[] strings = str.split("\\s+");
        Stack<String> sta = new Stack<>();
        StringBuffer sb = new StringBuffer();
        for (String s : strings) {
            if (ISNUMBER.matcher(s).matches()) {
                sb.append(s + " ");
            } else {
                switch (s) {
                    case ")":
                        while (!sta.isEmpty() && (!"(".equals(sta.peek()))) {
                            sb.append(sta.pop() + " ");
                        }
                        sta.pop();
                        break;
                    case "(":
                        sta.push(s);
                        break;
                    case "^":
                        while (!sta.isEmpty() && (!("^".equals(sta.peek()) ||
                                "(".equals(sta.peek())))) {
                            sb.append(sta.pop());
                        }
                        sta.push(s);
                        break;
                    case "*":
                    case "/":
                        while (!sta.isEmpty() && (!"+".equals(sta.peek())) &&
                                (!"-".equals(sta.peek())) && (!"
("".equals(sta.peek()))) {
                            sb.append(sta.pop()+" ");
                        }
                        sta.push(s);
```

```
                        break;
                    case "+":
                    case "-":
                        while (!sta.isEmpty() && (!"(".equals(sta.peek()))) {
                            sb.append(sta.pop() + " ");
                        }
                        sta.push(s);
                        break;
                    default:
                }
            }
        }
        while (!sta.isEmpty()) {
            sb.append(sta.pop()+" ");
        }
        System.out.println(sb.toString());
    }
```

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：

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

        String[] strings = str.split("\\s+");
        for (int i = 0; i < strings.length / 2; i++) {
            String temp = strings[i];
            strings[i] = strings[strings.length - i - 1];
            strings[strings.length - i - 1] = temp;
        }
        System.out.println(Arrays.toString(strings));
        Stack<String> sta = new Stack<>();
        StringBuffer sb = new StringBuffer();
        for (String s : strings) {
            if (ISNUMBER.matcher(s).matches()) {
                sb.append(s + " ");
            } else {
                switch (s) {
                    case "(":
                        while (!sta.isEmpty() && (!")".equals(sta.peek()))) {
                            sb.append(sta.pop() + " ");
                        }
                        sta.pop();
                        break;
                    case ")":
                        sta.push(s);
                        break;
                    case "^":
                        while (!sta.isEmpty() && (!("^".equals(sta.peek()) ||
```

```
                                "(".equals(sta.peek()))) {
                            sb.append(sta.pop());
                        }
                        sta.push(s);
                        break;
                    case "*":
                    case "/":
                        sta.push(s);
                        break;
                    case "+":
                    case "-":
                        while (!sta.isEmpty() && (!")".equals(sta.peek()))) {
                            sb.append(sta.pop() + " ");
                        }
                        sta.push(s);
                        break;
                    default:
                }
            }
        }
        while (!sta.isEmpty()) {
            sb.append(sta.pop() + " ");
        }
        System.out.println(sb.reverse().toString());
    }
```