**Java数据结构**

### 一、算法引导

1. 在字符串1中查找字符串2第一次出现的位置。【indexOf()】

KMP算法（部分匹配表）

1. 8皇后问题（92中解法）

回溯算法

1. 马踏棋盘算法

DFS + 贪心算法

### 二、数据结构

1. 数据结构包括线性结构和非线性结构：

* 线性结构：

1. 数据元素之间存在一对一的线性关系。
2. 线性结构有两种不同的存储结构：顺序存储结构和链式存储结构。顺序存储的线性表称作顺序表，顺序表中的存储元素是连续的（地址是连续的）；链式存储的线性表称为链表，链表中的存储元素不一定是连续的，元素节点中存放数据元素以及相邻元素的地址信息。
3. 线性结构常见的有：数组、链表、栈、队列。

* 非线性结构：

多维数组、广义表、树、图

1. 稀疏数组（SparseArray）

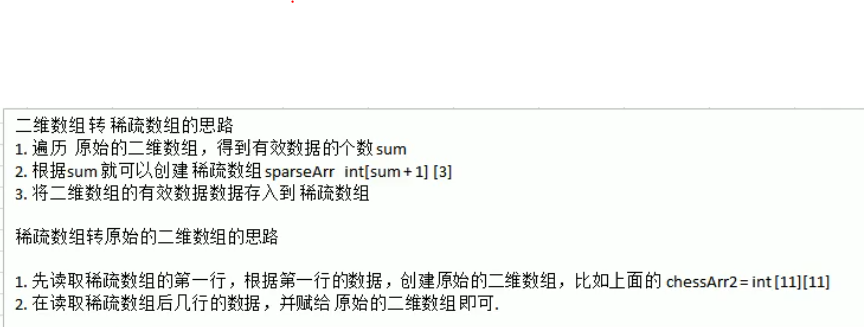
当一个数组中大部分元素为0，或者为同一值的数组时，可是使用稀疏数组来保存。

* 处理方法：

1. 记录数组一共有**几行几列**，有**多少个不同的值**。
2. 把具有不同值的元素的行列及值记录在一个小规模的数组（稀疏数组）中。



* 转化过程：



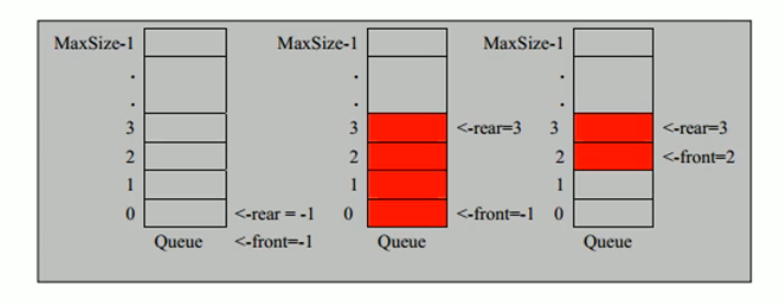
* 代码实现

|  |
| --- |
| **public class** SparseArray {   */\*\*  \* 二维数组转稀疏数组  \** ***@param array*** *\** ***@return*** *\*/* **public static int**[][] toSparseArray(**int**[][] array) {  **int** rowNumber = array.**length**;  **int** colNumber = array[0].**length**;  *// 1. 得到非 0 数据个数* **int** sum = 0;  **for** (**int** i = 0; i < rowNumber; i++) {  **for** (**int** j = 0; j < colNumber; j++) {  **if** (array[i][j] != 0) {  sum++;  }  }  }  *// 2. 构建稀疏数组* **int**[][]sparseArr = **new int**[sum + 1][3];  *// 3. 将数据存入稀疏数组* sparseArr[0][0] = rowNumber;  sparseArr[0][1] = colNumber;  sparseArr[0][2] = sum;   **int** count = 0;  **for** (**int** i = 0; i < rowNumber; i++) {  **for** (**int** j = 0; j < colNumber; j++) {  **if** (array[i][j] != 0) {  count++;  sparseArr[count][0] = i;  sparseArr[count][1] = j;  sparseArr[count][2] = array[i][j];  }  }  }  **return** sparseArr;  }   */\*\*  \* 稀疏数组转二维数组  \** ***@param sparseArr*** *\** ***@return*** *\*/* **public static int**[][] toOrdinaryArray(**int**[][] sparseArr) {  **if** (sparseArr == **null** && sparseArr[0].**length** != 3) {  **return null**;  }   **int**[][]array = **new int**[sparseArr[0][0]][sparseArr[0][1]];   **for** (**int** i = 1; i < sparseArr.**length**; i++) {  array[sparseArr[i][0]][sparseArr[i][1]] = sparseArr[i][2];  }  **return** array;  } } |

1. 队列

队列是一个有序列表，可以用数组或者链表实现，遵循先入先出的原则。

* 使用数组实现普通队列：

【第三个图只有一个元素】

**front指向队列头的前一个位置，rear指向队列尾部。**

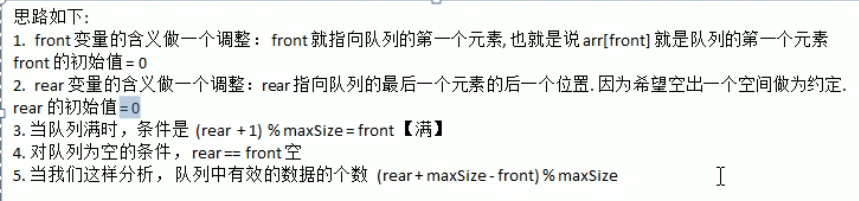
rear随数据的输入而改变，front随着数据的输出而改变。

当rear = maxSize – 1，表示队列已满。

当front = rear时，表示队列为空。

* 使用数组实现环形队列

改变front和rear含义



【此时少用了一个空间，实际只能放入maxSize – 1个数据。

rear指向的空间被预留出来，一直不放数据。】

|  |
| --- |
| **public class** ArrayCircleQueue {  **private int maxSize**;  **private int front**;  **private int rear**;  **private int**[] **arr**;   */\*\*  \* 创建队列  \*/* **public** ArrayCircleQueue(**int** arrMaxSize) {  **maxSize** = arrMaxSize;  **arr** = **new int**[**maxSize**];  **front** = 0; *// 指向队列头* **rear** = 0; *// 指向队列尾部的后一个位置* }   */\*\*  \* 判断队列是否满  \*/* **public boolean** isFull() {  **return** (**rear** + 1) % **maxSize** == **front**;  }   */\*\*  \* 判断队列是否为空  \*/* **public boolean** isEmpty() {  **return rear** == **front**;  }   */\*\*  \* 数据入队列  \*/* **public void** addQueue(**int** n) {  **if** (isFull()) {  System.***out***.println(**"队列已满"**);  **return**;  }  **arr**[**rear**] = n;  **rear** = (**rear** + 1) % **maxSize**;  }   */\*\*  \* 数据出队列  \*/* **public int** getQueue() {  **if** (isEmpty()) {  **throw new** RuntimeException(**"队列为空"**);  }  **int** result = **arr**[**front**];  **front** = (**front** + 1) % **maxSize**;  **return** result;  }   */\*\*  \* 显示队列头部  \** ***@return*** *\*/* **public int** headQueue() {  **if** (isEmpty()) {  **throw new** RuntimeException(**"队列为空"**);  }  **return arr**[**front**];  }   */\*\*  \* 显示队列  \*/* **public void** showQueue() {  **if** (isEmpty()) {  System.***out***.println(**"队列为空"**);  **return**;  }  **for** (**int** i = **front**; i < **front** + size(); i++) {  *// i is possibly over maxSize* System.***out***.printf(**"arr[%d]=%d\n"**, i % **maxSize**, **arr**[i % **maxSize**]);  }  }   */\*\*  \* 求出当前队列有效数据的个数  \*/* **public int** size() {  **return** (**rear** + **maxSize** - **front**) % **maxSize**;  } } |

1. 单链表（Linked List）

链表是以节点的方式来存储的，每个节点包含一个data域，一个指针next域。链表的各个节点不一定是连续存储。

链表分带头结点和不带头结点的链表。

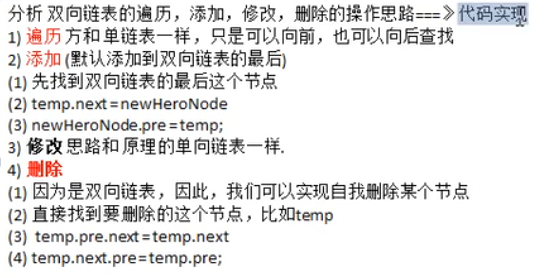
【小技巧：当需要找到所需要的节点的前一个结点时，使用temp = head；当不需要找到前一个节点时，使用temp = head.next更方便易懂】

|  |
| --- |
| **public class** LinkedList<T> {  **private** Node<T> **head** = **new** Node<>(**null**);   */\*\*  \* 尾插法  \*/* **public void** tailAdd(T data) {  Node<T> newNode = **new** Node<>(data);  newNode.**next** = **null**;  Node temp = **head**;  **while** (**true**) {  **if** (temp.**next** == **null**) **break**;  temp = temp.**next**;  }  temp.**next** = newNode;  }   */\*\*  \* 头插法  \*/* **public void** headAdd(T data) {  Node<T> newNode = **new** Node<>(data);  newNode.**next** = **head**.**next**;  **head**.**next** = newNode;  }   **public void** update(**int** position, T data) {  **if** (position < 1) {  System.***out***.printf(**"参数 position = %d 不合法\n"**, position);  **return**;  }  **int** count = 1;  Node temp = **head**.**next**;  **if** (temp == **null**) {  System.***out***.println(**"链表为空"**);  **return**;  }  **boolean** flag = **false**;  **while** (**true**) {  **if** (temp == **null**) **break**;  **if** (count == position) {  flag = **true**;  **break**;  }  temp = temp.**next**;  count++;  }  **if** (flag) {  temp.**data** = data;  } **else** {  System.***out***.printf(**"参数position(%d) > 链表总长(%d)\n"**, position, count - 1);  }  }   **public void** delete(**int** position) {  **if** (position < 1) {  System.***out***.printf(**"参数 position = %d 不合法\n"**, position);  **return**;  }  *// 判断不判断链表是否为空均可* **int** count = 0;  Node temp = **head**;  **boolean** flag = **false**;  **while** (**true**) {  **if** (temp.**next** == **null**) **break**;  **if** (count + 1 == position) {  flag = **true**;  **break**;  }  temp = temp.**next**;  count++;  }  **if** (flag) {  temp.**next** = temp.**next**.**next**;  } **else** {  System.***out***.printf(**"参数position(%d) > 链表总长(%d)\n"**, position, count);  }  }   **public void** list() {  Node temp = **head**.**next**;  **if** (temp == **null**) {  System.***out***.println(**"链表为空"**);  **return**;  }  **while** (**true**) {  **if** (temp == **null**) **break**;  System.***out***.println(temp);  temp = temp.**next**;  }  }   **private class** Node<T> {  **private** T **data**;  **private** Node **next**;   **public** Node(T data) {  **this**.**data** = data;  }   @Override  **public** String toString() {  **return "Node{"** +  **"data="** + **data** +  **'}'**;  }  } } |

1. 双向链表

单向链表查找方向只能是一个方向，并且单向链表不能自我删除。

* 思路分析

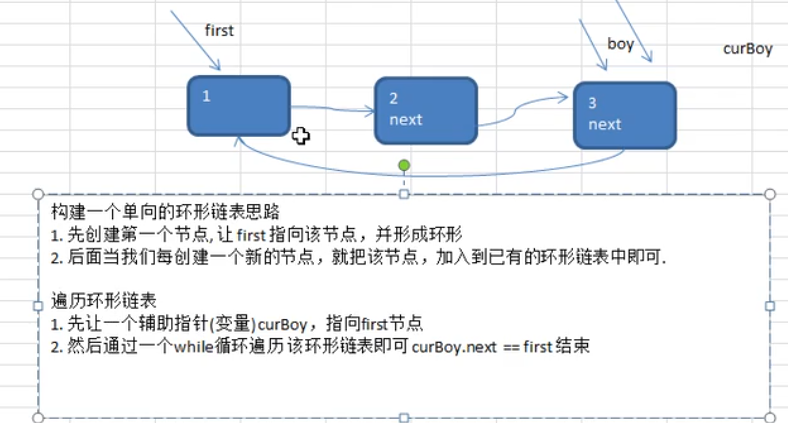


* 代码实现

|  |
| --- |
| **public class** DoubleLinkedList<T> {  **private** Node<T> **head** = **new** Node<>(); **public void** tailAdd(T data) {  ...  temp.**next** = newNode;  newNode.**pre** = temp;  }   */\*\*  \* 头插法  \*/* **public void** headAdd(T data) {  Node<T> newNode = **new** Node<>(data);  newNode.**next** = **head**.**next**;  newNode.**pre** = **head**;  **head**.**next** = newNode;  **if** (newNode.**next** != **null**) {  newNode.**next**.**pre** = newNode;  }   }   */\*\*  \* 删除节点  \** ***@param position*** *\*/* **public void** delete(**int** position) {  **if** (position < 1) {  System.***out***.printf(**"参数 position = %d 不合法\n"**, position);  **return**;  }  *// 判断不判断链表是否为空均可* **int** count = 0;  Node temp = **head**.**next**;  **boolean** flag = **false**;  **while** (**true**) {  **if** (temp == **null**) **break**;  count++;  **if** (count == position) {  flag = **true**;  **break**;  }  temp = temp.**next**;  }  **if** (flag) {  temp.**pre**.**next** = temp.**next**;  **if** (temp.**next** != **null**) { *// 如果是最后一个节点，就不需要执行* temp.**next**.**pre** = temp.**pre**;  }  } **else** {  System.***out***.printf(**"参数position(%d) > 链表总长(%d)\n"**, position, count);  }  }   */\*\*  \* 更新节点操作与单链表完全相同**\*/* **public void** update(**int** position, T data) {  ...  }   **private class** Node<T> {  **private** T **data**;  **private** Node **next**;  **private** Node **pre**;  ...  } } |

1. 单向环形链表（约瑟夫环）

维持一个first指针，作为队首指针。



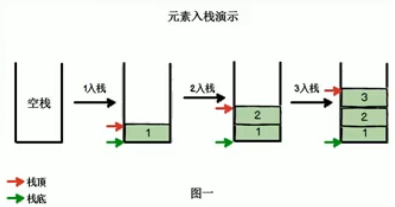
这里使用不带头结点的单向循环列表解决约瑟夫环问题。

|  |
| --- |
| */\*\*  \* 不带头指针的单向环形链表解决约瑟夫环问题  \*/* **class** CircleSingleLinkedList {  **private** Boy **first** = **null**;   **public void** create(**int** nums) {  **if** (nums < 1) {  System.***out***.println(**"nums 参数不正确"**);  **return**;  }   Boy curBoy = **null**; *// 辅助指针* **for** (**int** i = 1; i <= nums; i++) {  Boy boy = **new** Boy(i);  *// 不带头结点的单向循环指针* **if** (i == 1) {  **first** = boy;  **first**.**next** = **first**;  curBoy = **first**;  } **else** {  curBoy.**next** = boy;  boy.**next** = **first**;  curBoy = boy;  }  }  }   */\*\*  \* 遍历当前的环形裂变  \*/* **public void** show() {  **if** (**first** == **null**) {  System.***out***.println(**"链表为空"**);  **return**;  }  Boy temp = **first**;  **while** (**true**) {  System.***out***.printf(**"小孩的编号是 %d\n"**, temp.getNum());  **if** (temp.**next** == **first**) {  **break**;  }  temp = temp.**next**;  }  }   */\*\*  \* 打印小孩出圈顺序 解决约瑟夫环问题  \** ***@param startNo*** *开始数数的小孩数  \** ***@param count*** *数数出列个数  \** ***@param nums*** *开始圈中小孩数  \*/* **public void** count(**int** startNo, **int** count, **int** nums) {  **if** (**first** == **null** || startNo < 1 || startNo > nums) {  System.***out***.println(**"参数输入有误"**);  **return**;  }  *// 1. 初始时，辅助指针应该指向队列最后一个节点，即first的上一个节点* Boy helper = **first**;  **while** (**true**) {  **if** (helper.**next** == **first**) **break**;  helper = helper.**next**;  }   *// 2. 报数前，移动指针到startNo处，指针应该移动 startNo - 1 次* **for** (**int** i = 0; i < startNo - 1; i++) {  **first** = **first**.**next**;  helper = helper.**next**;  }   *// 3. 开始报数，指针移动 count - 1 次，然后出圈* **while** (**true**) {  **if** (helper == **first**) **break**;  **for** (**int** i = 0; i < count - 1; i++) {  **first** = **first**.**next**;  helper = helper.**next**;  }  System.***out***.printf(**"小孩 %d 出圈\n"**, **first**.getNum());  **first** = **first**.**next**;  helper.**next** = **first**;  }  System.***out***.printf(**"最后留在圈中的小孩编号是 %d\n"**, **first**.getNum());  } } |

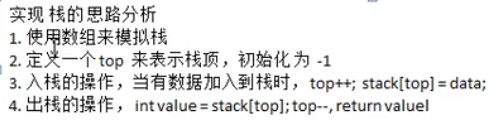
1. 栈（stack）

栈是一个**先入后出**（FILO）的有序列表。栈限制线性表中元素的插入和删除只能在线性表的同一端进行，允许插入和删除的一端叫栈顶，另外固定的一端称为栈底。

可以用来解决对于表达式的求解（7\*2\*2-5+1-5\*3-3）、子程序的跳入、表达式的转换（中缀表达式转后缀表达式）、DFS。



* 思路实现

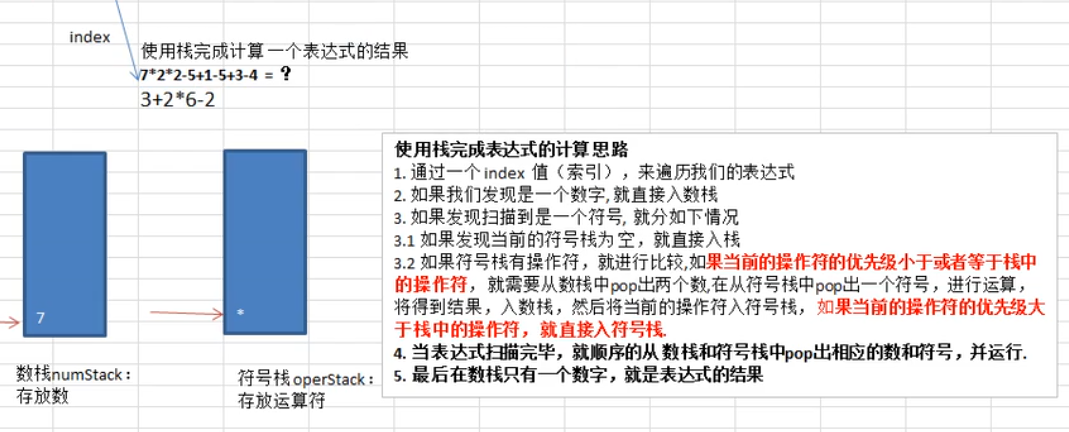


【当前栈顶指针指向的位置是最后一个入栈的数据的位置】

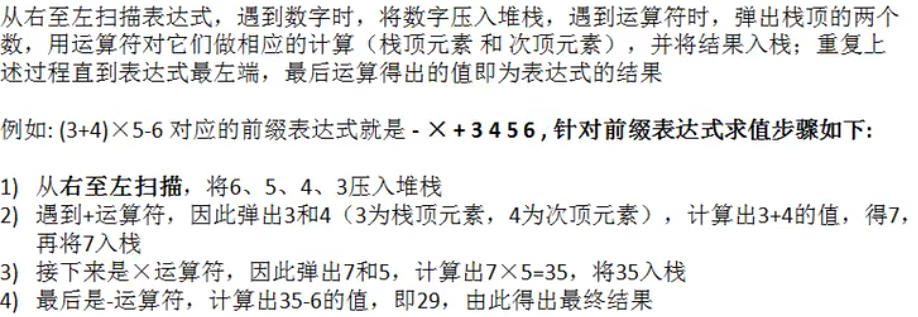
* 代码实现

|  |
| --- |
| **public class** ArrayStack {  **private int maxSize**;  **private int**[] **stack**;  **private int top** = -1; *// 栈顶* **public** ArrayStack(**int** maxSize) {  **this**.**maxSize** = maxSize;  **stack** = **new int**[**this**.**maxSize**];  }   **public boolean** isFull() {  **return top** == **maxSize** - 1;  }   **public boolean** isEmpty() {  **return top** == -1;  }   **public void** push(**int** value) {  **if** (isFull()) {  System.***out***.println(**"栈满"**);  **return**;  }  **top**++;  **stack**[**top**] = value;  }   **public int** pop() {  **if** (isEmpty()) {  **throw new** RuntimeException(**"栈空"**);  }  **int** value = **stack**[**top**];  **top**--;  **return** value;  }   **public void** list() {  **if** (isEmpty()) {  System.***out***.println(**"栈空"**);  **return**;  }  **for** (**int** i = **top**; i >= 0; i--) {  System.***out***.printf(**"stack[%d] = %d\n"**, i, **stack**[i]);  }  } } |

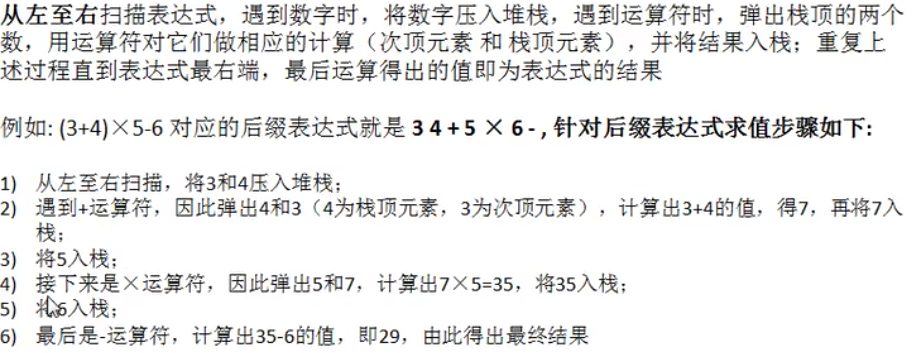
* 栈实现综合计算器（中缀表达式的计算）：



* 前缀表达式（波兰表达式）：运算符都位于操作数之前



* 后缀表达式（逆波兰表达式）

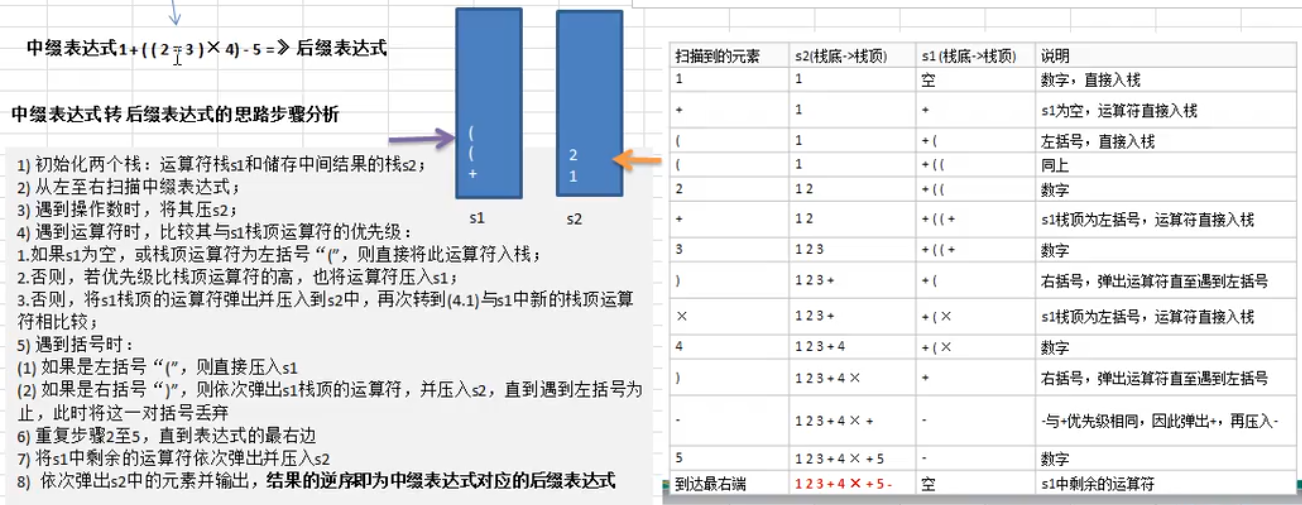


【注意：后缀表达式中，如果是减法，应该是次栈顶减栈顶；前缀表达式为栈顶减次栈顶】

|  |
| --- |
| **public class** PolandNotation {   **public static void** main(String[] args) {  *// 为简单起见，逆波兰表达式中间用括号隔开  // (30+4)\*5-6 => 30 4 + 5 \* 6 - res = 164* String expression = **"30 4 + 5 \* 6 -"**;  **int** res = *calculate*(*getListString*(expression));  System.***out***.printf(**"表达式 %s = %d\n"**, expression, res);  }   */\*\*  \* 差分逆波兰表达式，将其放入List中  \*/* **public static** List<String> getListString(String expression) {  String[] split = expression.split(**" "**);  **return new** ArrayList<>(Arrays.*asList*(split));  }   */\*\*  \* 完成对逆波兰表达式的遍历  \*/* **public static int** calculate(List<String> list) {  Stack<String> stack = **new** Stack<>();  **for** (String item : list) {  **if** (item.matches(**"\\d+"**)) { *// 如果是数字* stack.push(item);  } **else** {  *// 注意后缀表达式，此处应该为次栈顶 减 栈顶* **int** num2 = Integer.*parseInt*(stack.pop());  **int** num1 = Integer.*parseInt*(stack.pop());  **int** res = *cal*(num1, num2, item);  stack.push(String.*valueOf*(res));  }  }  **return** Integer.*parseInt*(stack.pop());  }   **private static int** cal(**int** num1, **int** num2, String oper) {  **int** res = 0;  **switch** (oper) {  **case "+"**:  res = num1 + num2;  **break**;  **case "-"**:  res = num1 - num2; *// 注意顺序* **break**;  **case "\*"**:  res = num1 \* num2;  **break**;  **case "/"**:  res = num1 / num2;  **break**;  }  **return** res;  } } |

* 中缀表达式转后缀表达式

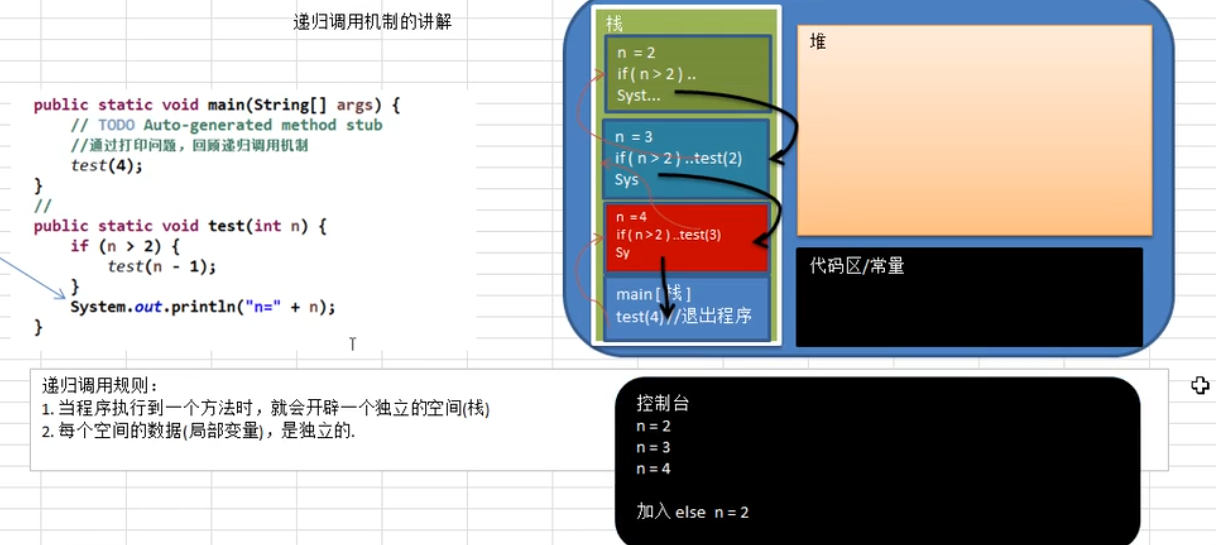
思路分析：



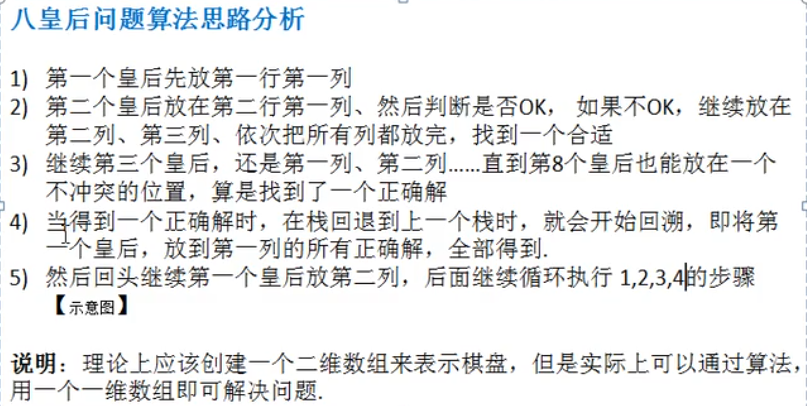
【因为 s2 栈中没有出栈操作，并且要求逆序输出，故可用List代替】

|  |
| --- |
| **public class** PolandNotation {   **public static void** main(String[] args) {  *// 为简单起见，逆波兰表达式中间用括号隔开  // (30+4)\*5-6 => 30 4 + 5 \* 6 - res = 164  // 1+((2+3)\*4)-5 => 1 2 3 + 4 \* + 5 - res = 16* String expression = **"1+((2+3)\*4)-5"**;  List<String> infixExpressionList = *toInfixExpressionList*(expression);  List<String> suffixExpressionList = *parseSuffixExpressionList*(infixExpressionList);  **int** res = *calculate*(suffixExpressionList);   System.***out***.printf(**"表达式 %s = %d\n"**, expression, res);  }    */\*\*  \* 将中缀表达式转为对应的中缀 List  \*/* **private static** List<String> toInfixExpressionList(String expression) {  List<String> list = **new** ArrayList<>();  **int** index = 0; *// 扫描的索引* String str = **""**;  **char** c = **' '**; **do** {  **if** (((c = expression.charAt(index)) < **'0'**) || ((c = expression.charAt(index)) > **'9'**)) { *// 符号* list.add(String.*valueOf*(c));  } **else** {  str += c;  **if** (index == expression.length() - 1) { *// 最后一个数字* list.add(str);  } **else if** (((c = expression.charAt(index + 1)) < **'0'**) || ((c = expression.charAt(index + 1)) > **'9'**)) {  list.add(str);  str = **""**;  }  }  index++;  } **while** (index < expression.length());  **return** list;  }    */\*\*  \* 将中缀表达式List转化为 后缀表达式List  \*/* **private static** List<String> parseSuffixExpressionList(List<String> list) {  Stack<String> s1 = **new** Stack<>();  *// 因为 s2 栈中没有出栈操作，并且要求逆序输出，故用List代替* List<String> s2 = **new** ArrayList<>();   **for** (String item : list) {  **if** (item.matches(**"\\d+"**)) {  s2.add(item);  } **else if** (**"("**.equals(item)) {  s1.push(item);  } **else if**(**")"**.equals(item)) {  **while** (!**"("**.equals(s1.peek())) {  s2.add(s1.pop());  }  s1.pop();  } **else** {  *// 当item的优先级小于栈顶的优先级时，s1出栈加入到s2  // 按照当前 ( 的优先级小于所有其他运算符, 故左括号不需要单独考虑* **while** (!s1.isEmpty() && Operation.*getPriority*(item) <= Operation.*getPriority*(s1.peek())) {  s2.add(s1.pop());  }  s1.push(item);  }  }  *// 将s1中的剩余运算符加入到s2* **while** (!s1.isEmpty()) {  s2.add(s1.pop());  }  **return** s2;  }   */\*\*  \* 操作符优先级类  \*/* **static class** Operation {  **private static int** *ADD* = 1;  **private static int** *SUB* = 1;  **private static int** *MUL* = 2;  **private static int** *DIV* = 2;   **public static int** getPriority(String operation) {  **int** res = 0;  **switch** (operation) {  **case "+"**:  res = *ADD*;  **break**;  **case "-"**:  res = *SUB*;  **break**;  **case "\*"**:  res = *MUL*;  **break**;  **case "/"**:  res = *DIV*;  **break**;  **default**:  **break**;  }  **return** res;  }  } } |

1. 递归（Recursion）



* 八皇后问题（92种）



* 穷举

|  |
| --- |
| */\*\*  \* 暴力解8皇后问题  \*/* **public class** Queue8 {   **int max** = 8;  **int**[] **array** = **new int**[**max**];  **static int** *count* = 0;   **public static void** main(String[] args) {  Queue8 queue8 = **new** Queue8();  queue8.place(0);  System.***out***.printf(**"一共有 %d 种解法"**, *count*);   }   **private void** print() {  **for** (**int** i = 0; i < **array**.**length**; i++) {  System.***out***.print(**array**[i] + **" "**);  }  System.***out***.println();  }   */\*\*  \* 检测是否冲突  \* array[i] = value 表示第 i+1个皇后放在第 value+1列  \** ***@param n*** *n表示第n个皇后  \** ***@return*** *true表示不冲突  \*/* **private boolean** judge(**int** n) {  **for** (**int** i = 0; i < n; i++) {  *// 同列 同一条斜线（|行差| = |列差|）* **if** (**array**[i] == **array**[n] || Math.*abs*(n - i) == Math.*abs*(**array**[n] - **array**[i])) {  **return false**;  }  }  **return true**;  }   */\*\*  \* 放置皇后方法  \** ***@param n*** *\*/* **private void** place(**int** n) {  **if** (n == **max**) { *// 相当于在放第9个皇后, 表示前8个皇后都放置好* print();  *count*++;  **return**;  }  *// 依次放入皇后，并判断是否冲突* **for** (**int** i = 0; i < **max**; i++) {  *// 将该皇后n放到第i个位置* **array**[n] = i;  **if** (judge(n)) {  place(n + 1);  }  }  } } |

1. 排序算法

排序是指将一组数据，按指定的顺序进行排序的过程。

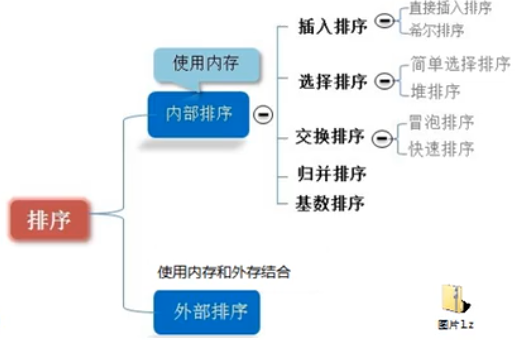
* 排序算法的分类：

1. 内部排序

指将需要处理的所有数据都加载到内部存储器中进行排序。

1. 外部排序

数据量过大，无法全部加载到内存中，需要借助外部存储进行排序。



【基数排序是桶排序的升级版】

* 时间复杂度

1. 时间频度：一个算法花费的时间与算法中语句的执行次数成正比。一个算法中的语句执行次数称为语句频度或时间频度，记为T(n)。
2. 常见时间复杂度

：

|  |
| --- |
| **int** i = 1; **while** (i < n) {  i = i \* 2; } |

：

|  |
| --- |
| **for** (**int** j = 0; j < n; j++) {  **int** i = 1;  **while** (i < n) {  i = i \* 2;  } } |

1. 排序算法的时间复杂度



* 冒泡排序

1. 基本思路：通过对待排序序列从前向后一次比较相邻元素的值，若发现逆序则交换，使值较大的元素逐渐从前移向后部，就像水底的气泡一样逐渐向上冒。
2. 优化：因为排序过程中，各元素不断接近自己的位置，如果一趟下来没有进行过交换，就说明序列有序，因此可以在排序过程中设置一个标志flag判断元素是否进行过交换，从而减少不必要的比较。
3. 规则：

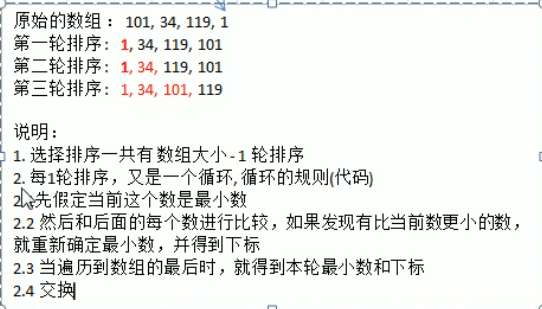
一共进行 n-1次循环；每一趟排序的次数在逐渐的减少。

1. 代码

|  |
| --- |
| **public static void** bubbleSort(**int**[] arr) {  **int** temp = 0;  **boolean** flag = **false**;  **for** (**int** i = 0; i < arr.**length** - 1; i++) {  **for** (**int** j = 0; j < arr.**length** - 1 - i; j++) {  **if** (arr[j] > arr[j + 1]) {  flag = **true**;  temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }   **if** (!flag) {  **break**;  } **else** { *// 重置flag，进行下次判断* flag = **false**;  }  } } |

* 选择排序

1. 选择排序是从欲排序的数据中，按指定的规则选出某一元素，再依规定交换位置后达到排序的目的。
2. 思路图解



1. 代码实现

|  |
| --- |
| **public static void** selectSort(**int**[] arr) {  **int** minIndex, min;  **for** (**int** i = 0; i < arr.**length** - 1; i++) {  minIndex = i;  min = arr[i];  **for** (**int** j = i + 1; j < arr.**length**; j++) {  **if** (min > arr[j]) {  minIndex = j;  min = arr[j];  }  }  **if** (minIndex != i) {  arr[minIndex] = arr[i];  arr[i] = min;  }  } } |

1. 选择排序速度 > 冒泡排序

* 插入排序

对欲排序的元素以插入的方式找寻该元素的适当位置，已达到排序的目的。

插入排序时，可采用交换法（速度较慢）和移动法。

代码实现：

|  |
| --- |
| *// 交换法*  **public static void** insertSortByChange(**int**[] arr) {  **int** temp;  **for** (**int** i = 1; i < arr.**length**; i++) { *// 遍历所有组  // 遍历各组中所有元素* **for** (**int** j = i - 1; j >= 0 ; j -= 1) { *// 步长为 5* **if** (arr[j] > arr[j + 1]) {  temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  } **else** {  **break**;  }  }  } } |
| *// 移动法*  **public static void** insertSortByMove(**int**[] arr) {  **int** insertVal, insertIndex;  **for** (**int** i = 1; i < arr.**length**; i++) {  insertVal = arr[i];  insertIndex = i - 1;  *// insertVal < arr[insertIndex] 表示还没有找到待插入的位置* **while** (insertIndex >= 0 && insertVal < arr[insertIndex]) {  arr[insertIndex + 1] = arr[insertIndex];  insertIndex--;  }  arr[insertIndex + 1] = insertVal;  } } |

* 希尔排序（缩小增量排序）

希尔排序是把记录按下标的一定**增量**分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰好分成一组，算法便终止。

1. 简单插入排序存在的问题：当较小的数位于较后面时，需要进行后移的次数明显增多，对效率有很大的影响。
2. 图解



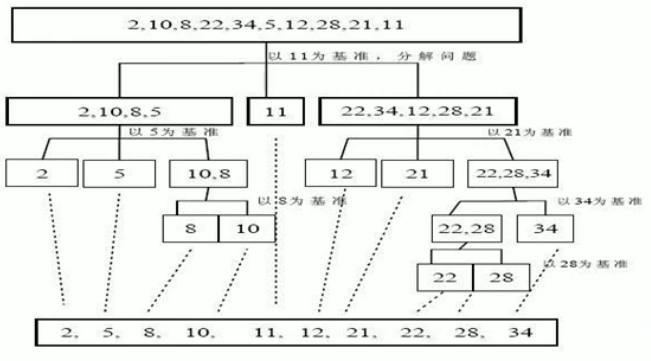
1. 代码实现

|  |
| --- |
| **public static void** shellSortByMove(**int**[] arr) {  **int** insertIndex, insertVal;  **for** (**int** gap = arr.**length** / 2; gap > 0; gap /= 2) {  **for** (**int** i = gap; i < arr.**length**; i++) {  insertVal = arr[i];  insertIndex = i - gap;  **while** (insertIndex >= 0 && arr[insertIndex] > insertVal) {  arr[insertIndex + gap] = arr[insertIndex];  insertIndex -= gap;  }  arr[insertIndex + gap] = insertVal;  }  } } |

* 快速排序

快速排序是对冒泡排序的一种改进。基本思路是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都小，然后按照此方法对这两部分数据进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

1. 图解



如果基准数选最左边，则要先从右进行选取；如果基准数在最右边，则要先从左进行选取。

1. 代码实现

|  |
| --- |
| **public static void** quickSort(**int**[] arr, **int** left, **int** right) {  **if** (left >= right) **return**;  *// 基准数* **int** base = arr[left];  **int** i = left, j = right;  **int** temp;  *// 如果基准数选最左边，则从先从右进行选择* **while** (i < j) {  **while** (arr[j] >= base && i <j) j--;  **while** (arr[i] <= base && i <j) i++;  **if** (i < j) {  temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  *// i和j相遇后* arr[left] = arr[i];  arr[i] = base;  *quickSort*(arr, left, i -1);  *quickSort*(arr, i + 1, right); } |

* 归并排序