# 数据结构和算法

# 第2章

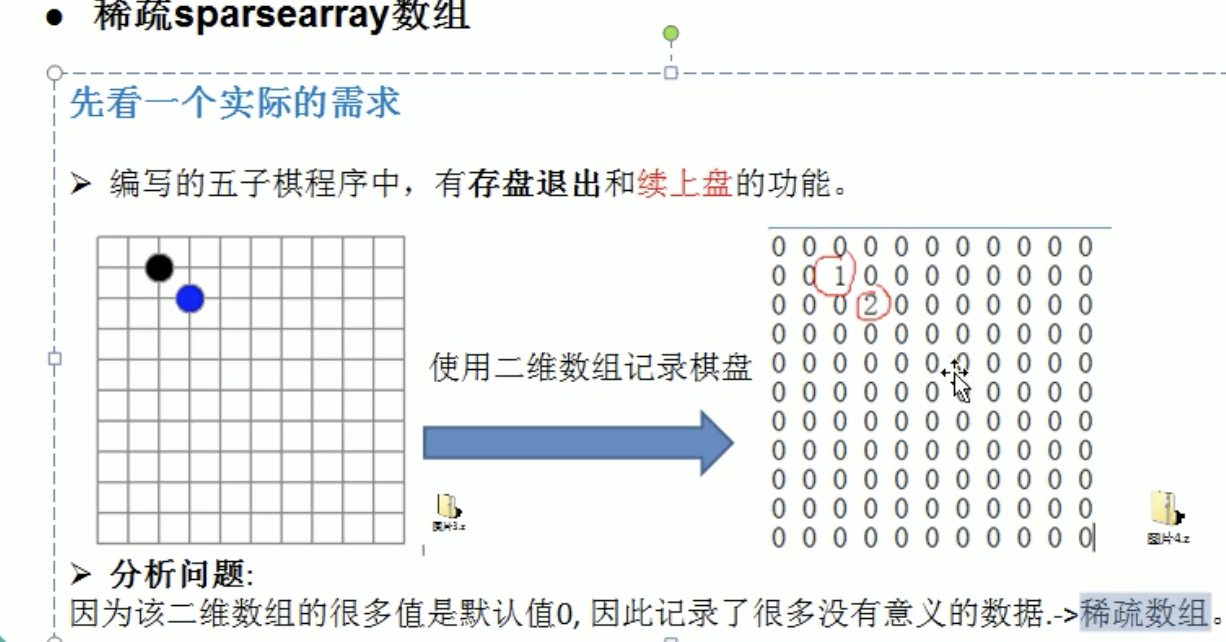
线性结构和非线性结构

线性结构：

1. 数据元素之间存在一对一的线性关系
2. 顺序存储结构和链式存储结构，顺序存储的线程表称为顺序表，顺序表的存储元素是连续的
3. 链式存储的线性表为链表，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及响铃的元素的地址信息
4. 线性表常见：数组、对列、链表和栈

非线性结构：二维数组，多为数组，广义表，树结构，图结构

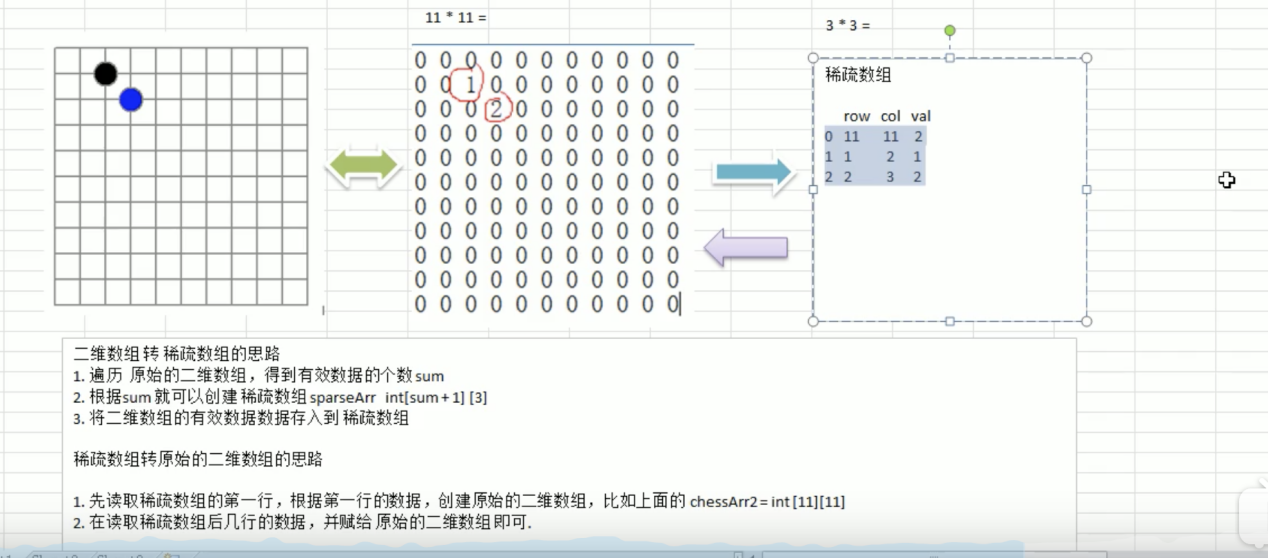
## 2.1 稀疏数组



当一个数组中大部分为0，或者为同一个值的数组时，可以使用稀疏数组来保存该数组

稀疏数组的处理方法是：

1. 记录数组一共有几行几列，有多少个不同的值
2. 把具有不同的值的元素的行列及值记录在一个小规模的数组中，从而缩小程序的规模



二维数组转稀疏数组：

1. 遍历 原始的二维数组，得到有效数据的个数sum
2. 根据sum就可以创建稀疏数组spaseArr int[sum+1][3]
3. 将二维数组的有效数据存入到稀疏数组

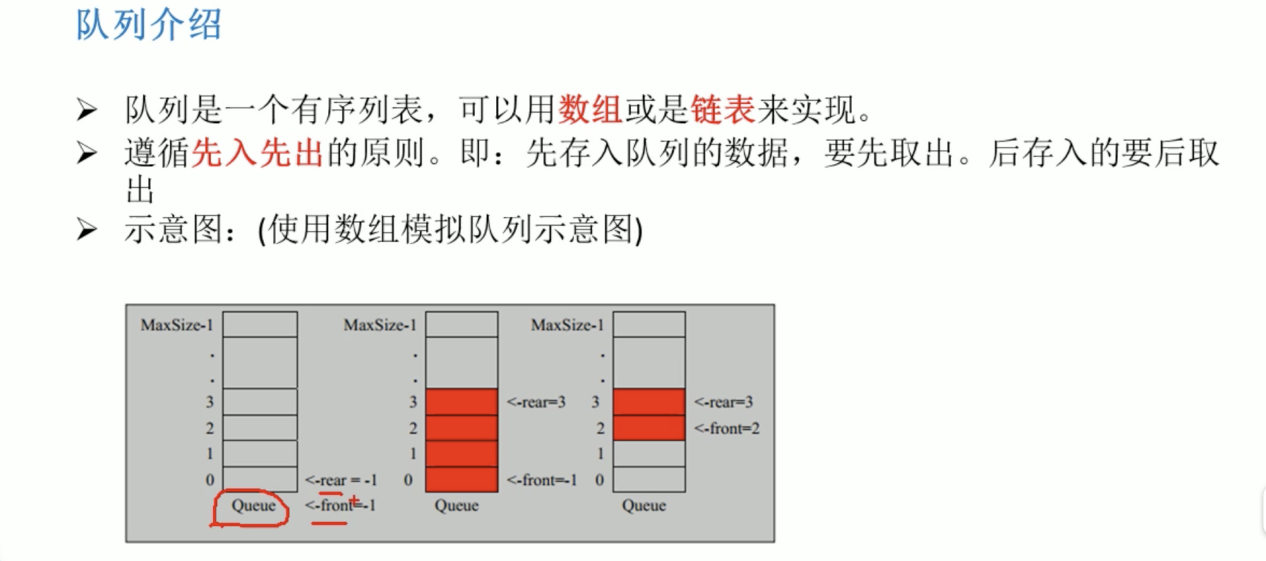
稀疏数组转原始的二维数组的思路：

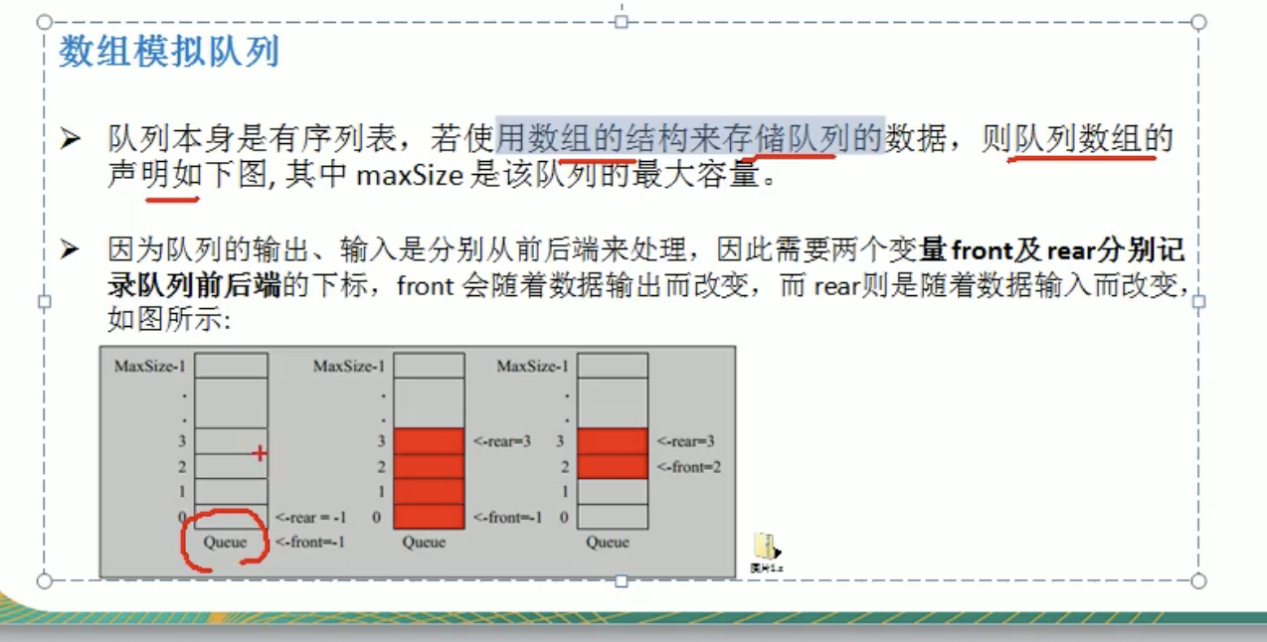
1. 先读取稀疏数组的第一行，根据第一行的数据，创建原始的二维数组，比如：
2. 在读取稀疏数组后几行的数据。并赋给原始二维数组即可

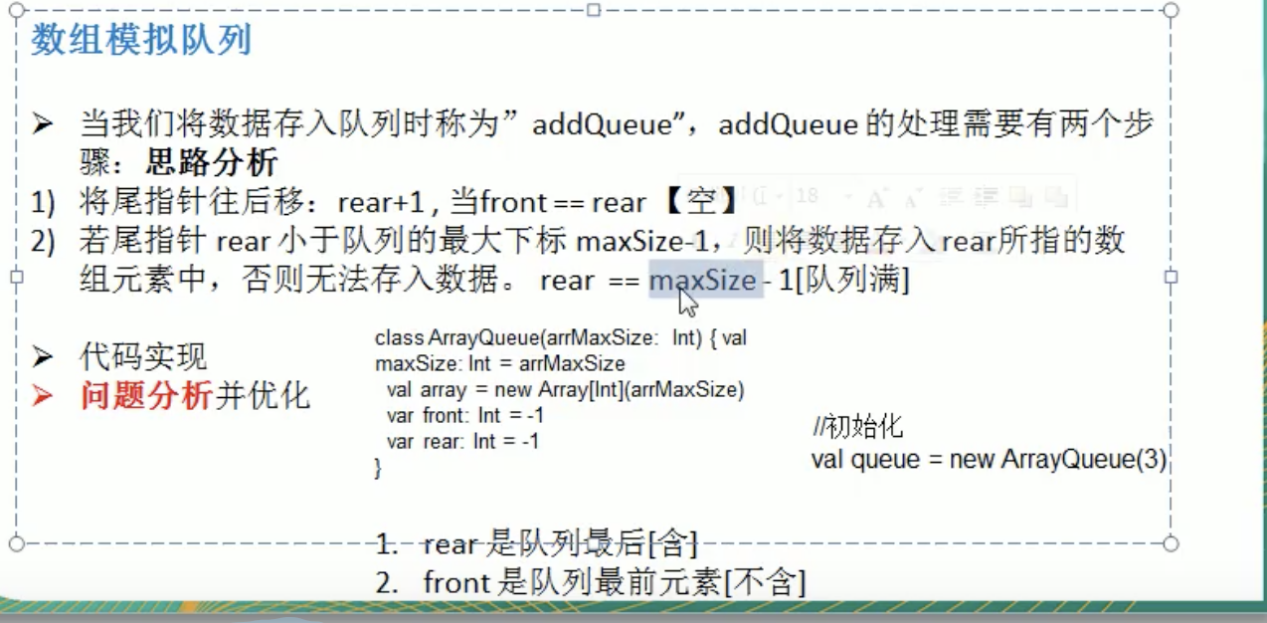
稀疏数组的简单使用

|  |
| --- |
| package com.datastructure.sparsearray;  public class SparseArray {    public static void main(String[] args) {  // 创建一个原始的二维数组11\*11  // 0: 没有棋子，1 表示黑子 2表示篮子   int chessArr1[][] = new int[11][11];   chessArr1[1][2] = 1;  chessArr1[2][3] = 2;   System.*out*.println("原始的二维数组");  for (int[] row : chessArr1) {  for (int data : row) {  System.*out*.printf("%d\t", data);  }  System.*out*.println();  }   // 将二维数组 转稀疏数组的思路  // 1.先遍历二位数组 得到非0数据的个数   int sum = 0;  for (int i = 0; i < 11; i++) {   for (int j = 0; j < 11; j++) {  if (chessArr1[i][j] != 0) {  sum++;  }  }  }  System.*out*.println("sum=" + sum);   // 创建稀疏数组  int sparseArray[][] = new int[sum + 1][3];  // 给稀疏数组赋值  sparseArray[0][0] = 11;  sparseArray[0][1] = 11;  sparseArray[0][2] = sum;   int count = 0;  for (int i = 0; i < 11; i++) {  for (int j = 0; j < 11; j++) {  if (chessArr1[i][j] != 0) {  count++;  sparseArray[count][0] = i;  sparseArray[count][1] = j;  sparseArray[count][2] = chessArr1[i][j];  }  }  }   System.*out*.println();   System.*out*.println("得到的稀疏数组为");   for (int i = 0; i < sparseArray.length; i++) {  System.*out*.printf("%d\t%d\t%d\t\n", sparseArray[i][0], sparseArray[i][1], sparseArray[i][2]);  }   // 将稀疏数组恢复成二维数组   System.*out*.println("将稀疏数组恢复原始数组");  int chessArr2[][] = new int[sparseArray[0][0]][sparseArray[0][1]];   for (int i = 1; i < sparseArray.length; i++) {  chessArr2[sparseArray[i][0]][sparseArray[i][1]] = sparseArray[i][2];  }   for (int[] row : chessArr2) {  for (int data : row) {  System.*out*.printf("%d\t", data);  }  System.*out*.println();  }   }   } |

## 2.2 对列



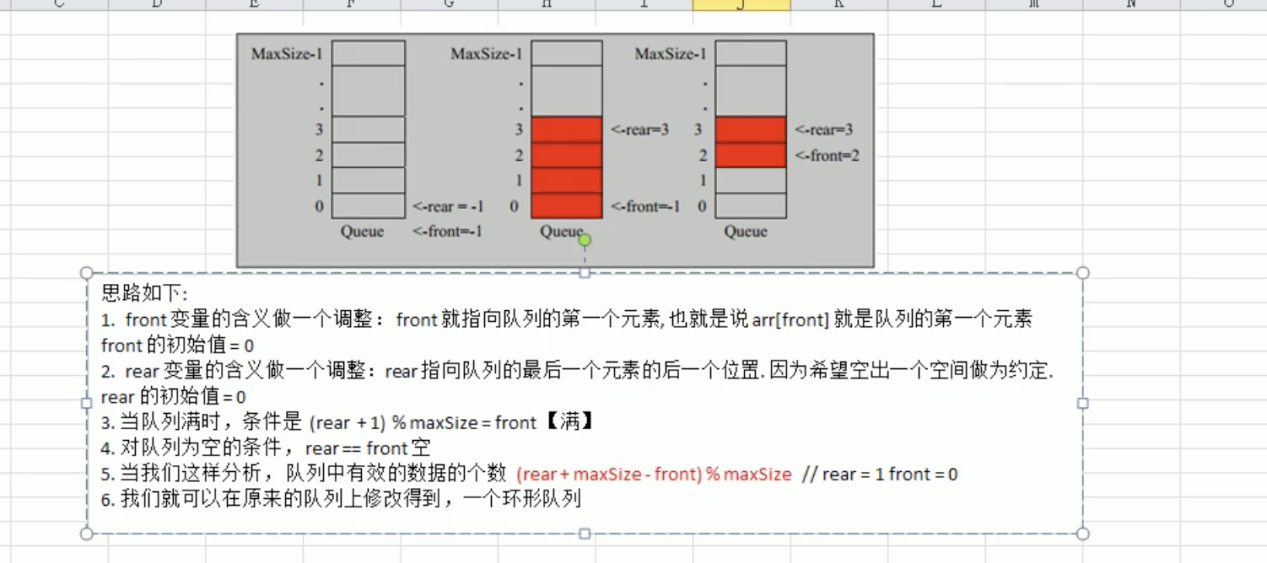




对列的数组实现

|  |
| --- |
| package com.datastructure.queue;  import java.util.Scanner;  public class ArrayQueueDemo {   public static void main(String[] args) {  ArrayQueue arrayQueue = new ArrayQueue(3);   char key = ' ';  Scanner scanner = new Scanner(System.*in*);   boolean loop = true;  while (loop) {  System.*out*.println("s(show):显示对列");  System.*out*.println("e(exit):退出程序");  System.*out*.println("a(add):添加数据到对列");  System.*out*.println("g(get):从对列取出数据");  System.*out*.println("h(head):查看对列头部数据");   key = scanner.next().charAt(0);   switch (key) {  case 's':  arrayQueue.showQueue();  break;  case 'a':  System.*out*.println("输出一个数");  int value = scanner.nextInt();  arrayQueue.addQueue(value);  break;  case 'g':  try {  int res = arrayQueue.getQueue();  System.*out*.printf("取出数据是%d\n", res);  } catch (Exception e) {  System.*out*.println(e.getMessage());  }  break;  case 'h':  try {  int res = arrayQueue.headQueue();  System.*out*.printf("对列头数据是%d\n", res);  } catch (Exception e) {  System.*out*.println(e.getMessage());  }  arrayQueue.showQueue();  break;  case 'e':  scanner.close();  loop = false;  break;  }  }  System.*out*.println("程序退出");  } }  // 使用数组模拟对列--编写一个ArrayQueue类  class ArrayQueue {   private int maxSize;// 数组最大值   //  private int front; // 头   private int rear; // 尾   private int[] arr;   // 创建对列  public ArrayQueue(int arrMaxSize) {  maxSize = arrMaxSize;  arr = new int[maxSize];  // 指向对列头部的前一个位置  front = -1;  rear = -1;  }   // 判断对列满  public boolean isFull() {  return rear == maxSize - 1;  }   // 判断对列空  public boolean isEmpty() {  return front == rear;  }   //添加数据到对列  public void addQueue(int n) {  if (isFull()) {  System.*out*.println("对列满，不能添加");  return;  }  rear++;  arr[rear] = n;  }   // 获取对列的数据  public int getQueue() {  // 判断对列是否空  if (isEmpty()) {  throw new RuntimeException("对列为空");  }   front++;// front后移动  return arr[front];   }   public void showQueue() {  if (isEmpty()) {  System.*out*.println("对列空");  return;  }   for (int i = 0; i < arr.length; i++) {  System.*out*.printf("arr[%d]=%d\n", i, arr[i]);  }  }   public int headQueue() {  if (isEmpty()) {  throw new RuntimeException("对列为空");  }  return arr[front + 1];  }   } |

循环对列

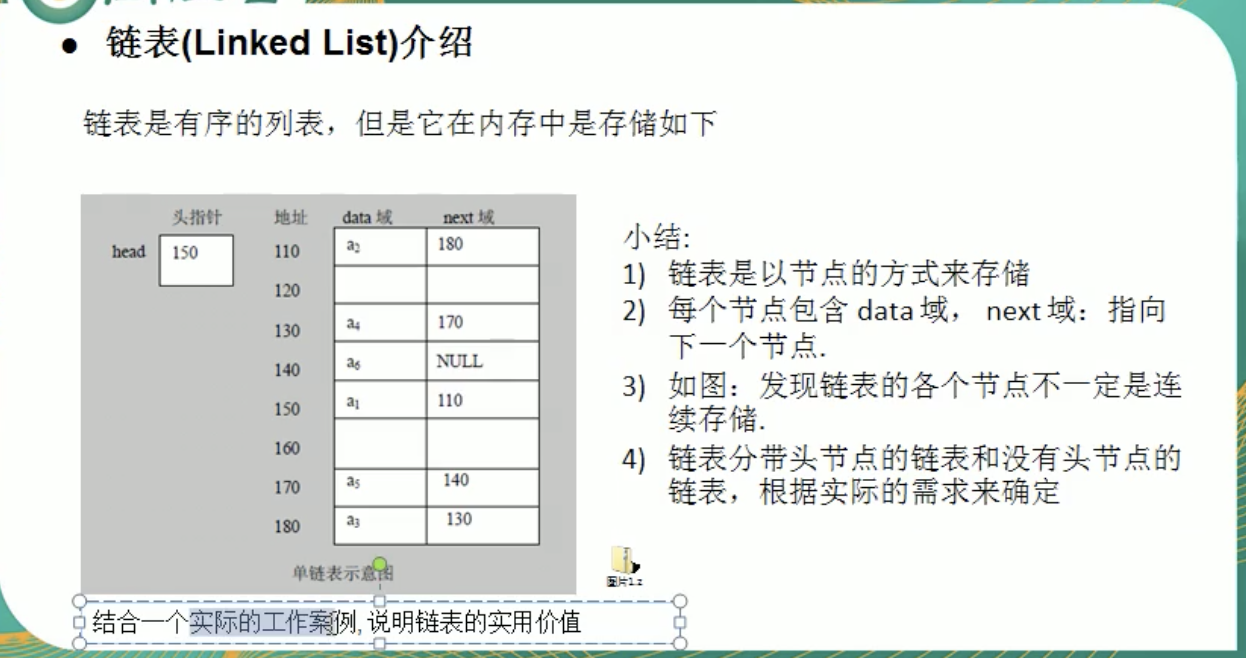


环形对列

|  |
| --- |
| package com.datastructure.queue;  import java.util.Scanner;  public class CircleArrayQueueDemo {   public static void main(String[] args) {  CircleArrayQueue arrayQueue = new CircleArrayQueue(3);   char key = ' ';  Scanner scanner = new Scanner(System.*in*);  boolean loop = true;  while (loop) {  System.*out*.println("s(show):显示对列");  System.*out*.println("e(exit):退出程序");  System.*out*.println("a(add):添加数据到对列");  System.*out*.println("g(get):从对列取出数据");  System.*out*.println("h(head):查看对列头部数据");   key = scanner.next().charAt(0);   switch (key) {  case 's':  arrayQueue.showQueue();  break;  case 'a':  System.*out*.println("输出一个数");  int value = scanner.nextInt();  arrayQueue.addQueue(value);  break;  case 'g':  try {  int res = arrayQueue.getQueue();  System.*out*.printf("取出数据是%d\n", res);  } catch (Exception e) {  System.*out*.println(e.getMessage());  }  break;  case 'h':  try {  int res = arrayQueue.headQueue();  System.*out*.printf("对列头数据是%d\n", res);  } catch (Exception e) {  System.*out*.println(e.getMessage());  }  arrayQueue.showQueue();  break;  case 'e':  scanner.close();  loop = false;  break;  }  }  System.*out*.println("程序退出");   }     }   // 使用数组模拟对列--编写一个ArrayQueue类  class CircleArrayQueue {   private int maxSize;// 数组最大值   // 指向对列的第一个元素，也就是arr[front]  private int front; // 头   // 指向对列的最后一个元素的后一个位置，希望空出一个位置  private int rear; // 尾   private int[] arr;   // 创建对列  public CircleArrayQueue(int arrMaxSize) {  maxSize = arrMaxSize;  arr = new int[maxSize];  }   // 判断对列满  public boolean isFull() {  return (rear + 1) % maxSize == front;  }   // 判断对列空  public boolean isEmpty() {  return front == rear;  }   //添加数据到对列  public void addQueue(int n) {  if (isFull()) {  System.*out*.println("对列满，不能添加");  return;  }  arr[rear] = n;  // 将rear后移，必须考虑取模  rear = (rear + 1) % maxSize;  }   // 获取对列的数据  public int getQueue() {  // 判断对列是否空  if (isEmpty()) {  throw new RuntimeException("对列为空");  }  // front是指向对列的第一个元素  // 1. 先把front对应的值保留到一个临时变量  // 2. 将front后移，考虑取模  // 3. 将临时保存的变量返回  int value = arr[front];  front = (front + 1) % front;  return value;  }   public void showQueue() {  if (isEmpty()) {  System.*out*.println("对列空");  return;  }  // 从front开始遍历，遍历多少个元素  //   for (int i = front; i < front + size(); i++) {  System.*out*.printf("arr[%d]=%d\n", i % maxSize, arr[i % maxSize]);  }  }   // 求出当前对列有效数据的个数  public int size() {  //  return (rear + maxSize - front) % maxSize;  }   public int headQueue() {  if (isEmpty()) {  throw new RuntimeException("对列为空");  }  return arr[front + 1];  } } |

## 2.3 链表

### 2.3.1 单链表

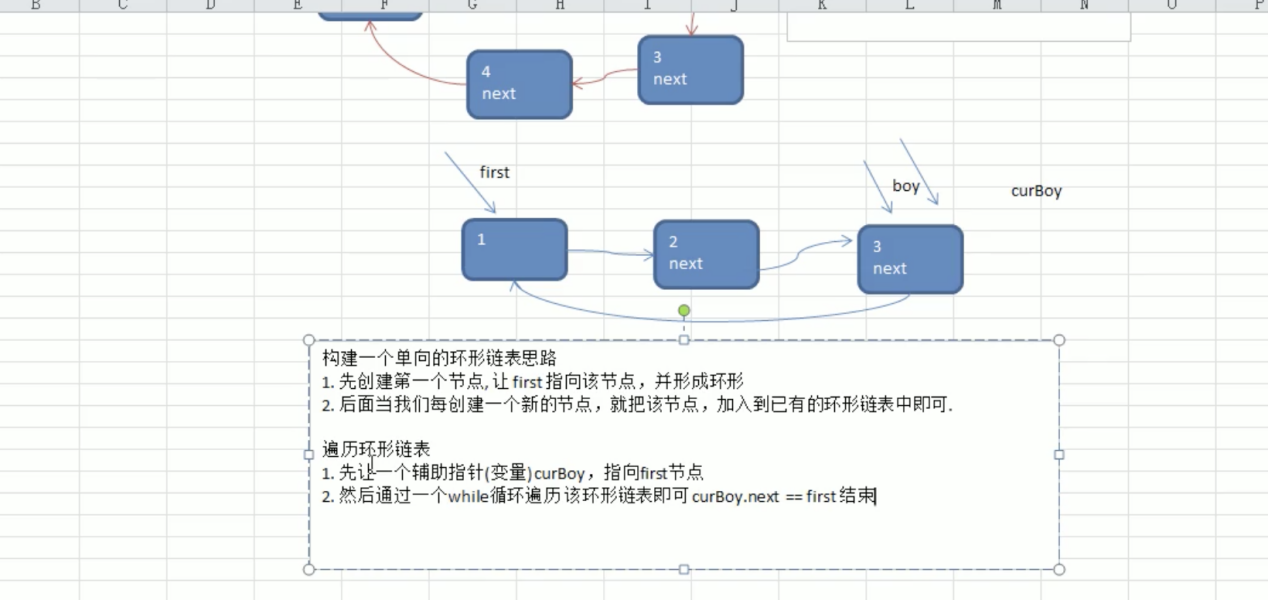


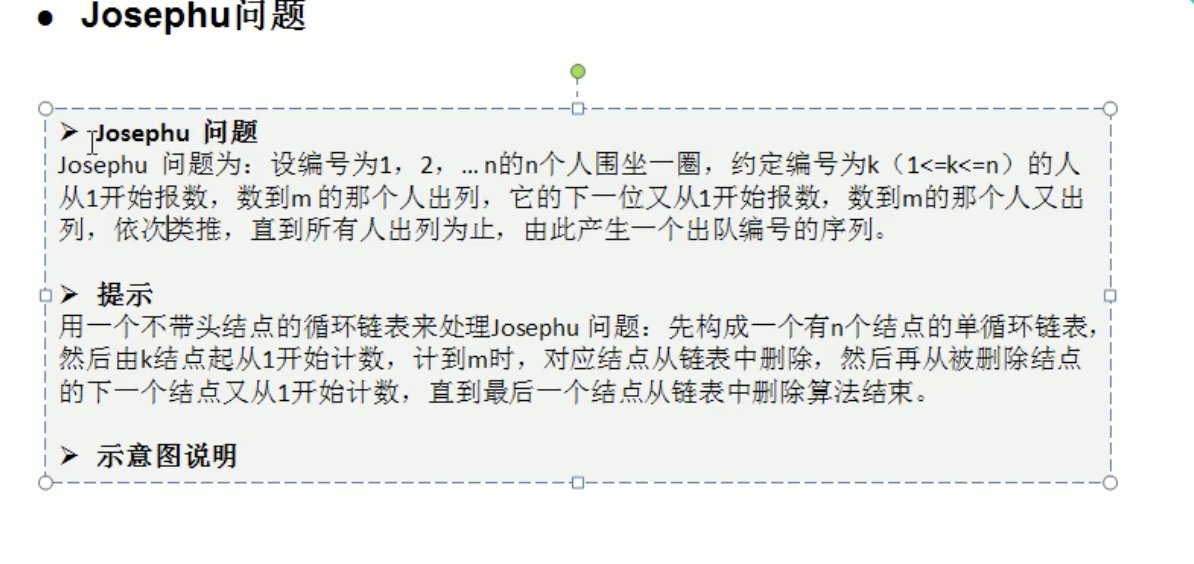
|  |
| --- |
| package com.datastructure.linkedlist;  public class SingleLinkedListDemo {   public static void main(String[] args) {    HeroNode hero1 = new HeroNode(1, "宋江", "及时雨");  HeroNode hero2 = new HeroNode(2, "卢俊义", "玉麒麟");  HeroNode hero3 = new HeroNode(3, "吴用", "智多星");  HeroNode hero4 = new HeroNode(4, "林冲", "豹子头");   SingleLinkedList list = new SingleLinkedList();  list.addByOrder(hero3);  list.addByOrder(hero1);  list.addByOrder(hero2);  list.addByOrder(hero4);    list.list();   }  }  // 定义SingleLinkedList class SingleLinkedList {   // 初始化头结点  private HeroNode head = new HeroNode(0, "", "");   // 添加及诶单到单向链表  // 1.当不考虑编号顺序时，  // 2.将最后这个节点的next指向新的节点  public void add(HeroNode heroNode) {  //  HeroNode temp = head;  // 遍历链表，找到最后  while (true) {  if (temp.next == null) {  break;  }  // 如果没有找到，将temp后移  temp = temp.next;  }   temp.next = heroNode;  }   // 根据排名将英雄插入到指定位置  // (如果哟这个排名，则添加失败，并给出提示)  public void addByOrder(HeroNode node) {   //因为是单链表，我们找的temp是位于添加位置的前一个结点，否则插入不了  HeroNode temp = head;  boolean flag = false;  while (true) {  if (temp.next == null) {  break;  }  if (temp.next.no > node.no) {  break;  } else if (temp.next.no == node.no) {  // 编号存在  flag = true;  break;  }   temp = temp.next;   }   if (flag) {  //不能添加  System.*out*.printf("准备插入的英雄的编号%d已经存在了，不能加入\n", node.no);  } else {  node.next = temp.next;  temp.next = node;  }   }   public void update(HeroNode node) {  if (head.next == null) {  System.*out*.println("空链表");  return;  }   // 找到需要修改的节点，根据no编号  HeroNode temp = head.next;  boolean flag = false;   while (true) {  if (temp == null) {  break;  }  if (temp.no == node.no) {  flag = true;  break;  }  temp = temp.next;   }  if (flag) {  temp.name = node.name;  temp.nickname = node.nickname;  } else {  System.*out*.printf("没有找到编号%d的节点，不能修改\n", node.no);  }  }    public void del(int no) {   HeroNode temp = head;  boolean flag = false;   while (true) {  if (temp.next == null) {  break;  }   if (temp.next.no == no) {  // 找到待删除结点的前一个结点  flag = true;  break;  }  temp = temp.next;   }  if (flag) {  temp.next = temp.next.next;  } else {  System.*out*.println("没有找到");  }   }    // 显示链表  public void list() {  // 空判断  if (head.next == null) {  System.*out*.println("链表为空");  return;  }  //  HeroNode temp = head.next;  while (true) {  if (temp == null) {  break;  }  System.*out*.println(temp);  temp = temp.next;  }  }   }  // 定义HeroNode，每个HeroNode对象就是一个节点 class HeroNode {  public int no;  public String name;  public String nickname;  public HeroNode next;   public HeroNode(int no, String name, String nickname) {  this.no = no;  this.name = name;  this.nickname = nickname;  }   @Override  public String toString() {  return "HeroNode{" +  "no=" + no +  ", name='" + name + '\'' +  ", nickname='" + nickname + '\'' +  '}';  } } |

### 2.3.2 双向链表

|  |
| --- |
| package com.datastructure.linkedlist;  public class DoubleLinkedListDemo {    public static void main(String[] args) {   HeroNode2 hero1 = new HeroNode2(1, "宋江", "及时雨");  HeroNode2 hero2 = new HeroNode2(2, "卢俊义", "玉麒麟");  HeroNode2 hero3 = new HeroNode2(3, "吴用", "智多星");  HeroNode2 hero4 = new HeroNode2(4, "林冲", "豹子头");   DoubleLinkedList list = new DoubleLinkedList();   list.add(hero1);  list.add(hero2);  list.add(hero3);  list.add(hero4);   list.list();  }  }  class DoubleLinkedList {   private HeroNode2 head = new HeroNode2(0, "", "");    // 显示链表  public void list() {  // 空判断  if (head.next == null) {  System.*out*.println("链表为空");  return;  }  //  HeroNode2 temp = head.next;  while (true) {  if (temp == null) {  break;  }  System.*out*.println(temp);  temp = temp.next;  }   }    public void add(HeroNode2 headNode) {  HeroNode2 temp = head;   while (true) {  if (temp.next == null) {  break;  }   temp = temp.next;  }  temp.next = headNode;  headNode.pre = temp;  }    public void update(HeroNode2 node) {  if (head.next == null) {  System.*out*.println("空链表");  return;  }   // 找到需要修改的节点，根据no编号  HeroNode2 temp = head.next;  boolean flag = false;   while (true) {  if (temp == null) {  break;  }  if (temp.no == node.no) {  flag = true;  break;  }  temp = temp.next;   }  if (flag) {  temp.name = node.name;  temp.nickname = node.nickname;  } else {  System.*out*.printf("没有找到编号%d的节点，不能修改\n", node.no);  }  }    public void del(int no) {   if (head.next == null) {  System.*out*.println("空");  return;  }   HeroNode2 temp = head;  boolean flag = false;   while (true) {  if (temp == null) {  break;  }   if (temp.no == no) {  // 找到待删除结点的前一个结点  flag = true;  break;  }  temp = temp.next;   }  if (flag) {  temp.pre.next = temp.next;  if (temp.next != null)  temp.next.pre = temp.pre;  temp.pre = null;  temp.next = null;    } else {  System.*out*.println("没有找到");  }   }   }  // 穿件  class HeroNode2 {  public int no;  public String name;  public String nickname;  public HeroNode2 next;  public HeroNode2 pre;   public HeroNode2(int no, String name, String nickname) {  this.no = no;  this.name = name;  this.nickname = nickname;  }   @Override  public String toString() {  return "HeroNode{" +  "no=" + no +  ", name='" + name + '\'' +  ", nickname='" + nickname + '\'' +  '}';  } } |

### 2.3.3 循环链表





## 2.4 栈

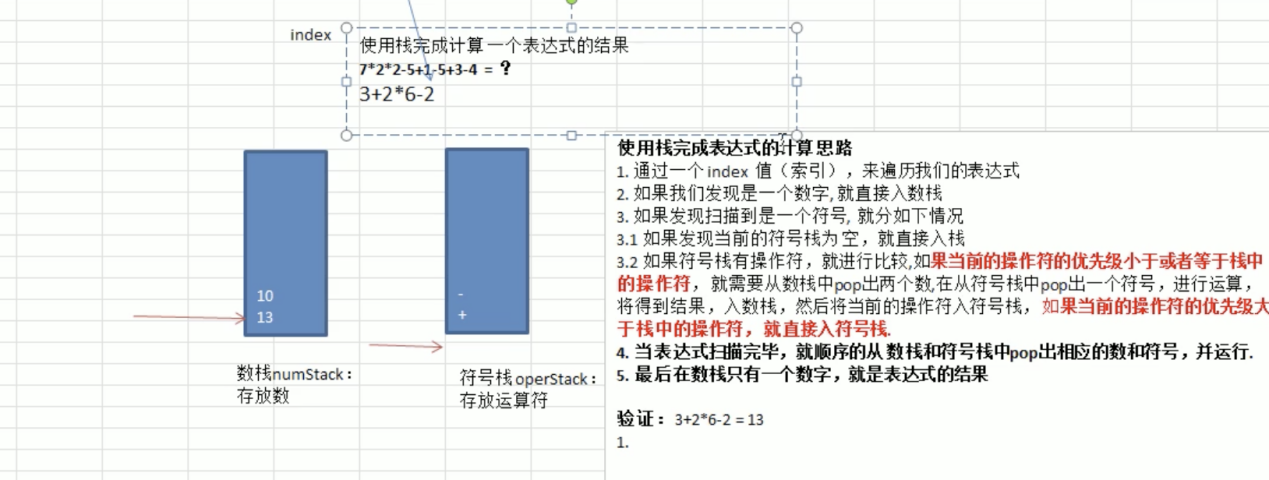
### 2.4.1 数组栈

|  |
| --- |
| package com.datastructure.stack;  import java.util.Scanner;  public class ArrayStackDemo {   public static void main(String[] args) {   ArrayStack stack = new ArrayStack(4);  String key = "";  boolean loop = true;  Scanner scanner = new Scanner(System.*in*);   while (loop){  System.*out*.println("show:表示显示栈");  System.*out*.println("exit:退出程序");  System.*out*.println("push:表示添加数据到栈(入栈)");  System.*out*.println("pop:表示从栈取出数据");  System.*out*.println("请你输入你的选者");   key = scanner.next();  switch (key){  case "show":  stack.list();  break;  case "push":  System.*out*.println("请输入一个数");  int value = scanner.nextInt();  stack.push(value);  break;  case "pop":  int pop = stack.pop();  System.*out*.println("出栈的数据是"+pop);  break;  case "exit":  scanner.close();  loop = false;  break;  }  }   } }   class ArrayStack {   private int maxSize; // 栈大小  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;  }   public boolean isEmpty(){  return top == -1;  }   public void push(int value){  if(isFull()){  System.*out*.println("满了");  return;  }   top++;  stack[top] = value;  }   public int pop(){  if(isEmpty()){  throw new RuntimeException("栈空了");  }   int value = stack[top];  top--;  return value;  }    public void list(){  if(isEmpty()){  System.*out*.println("栈空，没有数据");  return;  }   for (int i = top; i >= 0; i--) {  System.*out*.printf("stack[%d]=%d\n",stack[i]);  }   }                } |

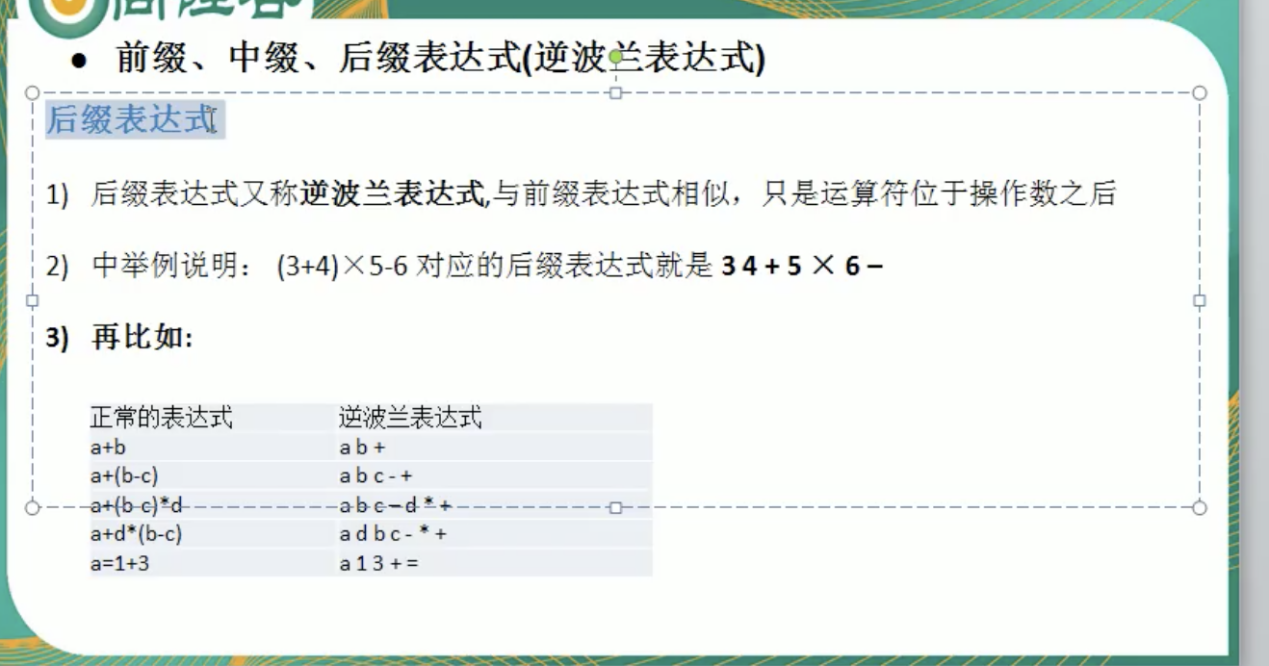
### 2.4.2 链表栈

|  |
| --- |
| package com.datastructure.stack;  import java.util.Stack;  public class SingleLinkedStackDemo {   public static void main(String[] args) {   SingleLinkedStack stack = new SingleLinkedStack(5);   stack.push(1);  stack.push(2);  stack.push(3);  stack.push(4);   stack.push(5);  stack.push(6);    }  }  class SingleLinkedStack {   private StackNode head = new StackNode();   private int maxSize;   private int top;   public SingleLinkedStack(int maxSize) {  this.maxSize = maxSize;  }   public boolean isFull() {  return top == maxSize;  }   public boolean isEmpty() {  return top == -1;  }   public void push(int value) {  if (isFull()) {  System.*out*.println("满了");  return;  }   StackNode next = head.next;   StackNode stackNode = new StackNode();  stackNode.value = value;  head.next = stackNode;  stackNode.next = next;  top++;  }   public int pop() {  if (isEmpty()) {  throw new RuntimeException("栈空了");  }   StackNode topStack = head.next;  head.next = topStack.next;  topStack.next = null;  top--;  return topStack.value;  }    public void list() {  if (isEmpty()) {  System.*out*.println("栈空，没有数据");  return;  }   StackNode temp = head.next;  while (temp != null) {  System.*out*.println(temp.value);  temp = temp.next;  }   }   }  class StackNode {   int value;   StackNode next;    @Override  public String toString() {  return "StackNode{" +  "value=" + value +  '}';  } } |

### 2.4.3 计算器



|  |
| --- |
| package com.datastructure.stack;  public class Calculator {   public static void main(String[] args) {   String expression = "30+20\*6-2";  // 创建连个栈   ArrayStack2 numStack = new ArrayStack2(10);  ArrayStack2 operStack = new ArrayStack2(10);    int index = 0;  int num1 = 0;  int num2 = 0;   int oper = 0;  int res = 0;  char ch = ' ';   String keepNum = "";   while (true) {  // 一次得到expression的每个字符。  ch = expression.charAt(index);   if (operStack.isOper(ch)) {  // 判断当前的复函栈是否为空  if (!operStack.isEmpty()) {  // 比较优先级  if (operStack.priority(ch) <= operStack.priority(operStack.peek())) {  num1 = numStack.pop();  num2 = numStack.pop();   res = numStack.cal(num1, num2, operStack.pop());  // 把运算结果入栈  numStack.push(res);  // 然后将当前的操作符号栈  operStack.push(ch);  } else {  // 如果当前的操作符的优先级栈中的操作符，就直接入符号栈  operStack.push(ch);  }  } else {  // 如果为空  operStack.push(ch);  }   } else {// 如果是数，则直接入数栈  // 多位数  keepNum +=ch;  if(index == expression.length() -1){  numStack.push(Integer.*parseInt*(keepNum));  }else{  char next = expression.charAt(index + 1);  if(operStack.isOper(next)){  numStack.push(Integer.*parseInt*(keepNum));  keepNum = "";  }  }    }   index++;  if (index >= expression.length()) {  break;  }   }    while (true) {  //如果符号栈为空，则计算到最后的结果，数栈中只有一个数字  if (operStack.isEmpty()) {  break;  }   num1 = numStack.pop();  num2 = numStack.pop();  oper = operStack.pop();   res = numStack.cal(num1, num2, oper);  numStack.push(res);    }   System.*out*.printf("表达式%s = %d", expression, numStack.pop());    }  }   class ArrayStack2 {   private int maxSize; // 栈大小  private int[] stack;  private int top = -1;// 栈顶    public ArrayStack2(int maxSize) {  this.maxSize = maxSize;  stack = new int[this.maxSize];  }   public boolean isFull() {  return top == maxSize;  }   public boolean isEmpty() {  return top == -1;  }   public void push(int value) {  if (isFull()) {  System.*out*.println("满了");  return;  }   top++;  stack[top] = value;  }   public int pop() {  if (isEmpty()) {  throw new RuntimeException("栈空了");  }   int value = stack[top];  top--;  return value;  }    public void list() {  if (isEmpty()) {  System.*out*.println("栈空，没有数据");  return;  }   for (int i = top; i >= 0; i--) {  System.*out*.printf("stack[%d]=%d\n", stack[i]);  }   }    // 返回与哪算符的优先级，优先级是程序员来确定，优先级使用数字表示  // 数字越大，则优先级就越高  public int priority(int oper) {  if (oper == '\*' || oper == '/') {  return 1;  } else if (oper == '+' || oper == '-') {  return 0;  } else {  return -1;  }  }   // 判断是不是一个运算符  public boolean isOper(char val) {  return val == '+' || val == '-' || val == '\*' || val == '/';  }   // 计算方法  public int cal(int num1, int num2, int oper) {  int res = 0;  switch (oper) {  case '+':  res = num1 + num2;  break;  case '-':  res = num2 - num1;  break;   case '\*':  res = num2 \* num1;  break;  case '/':  res = num2 / num1;  break;  }  return res;  }    public int peek() {  return stack[top];  } } |



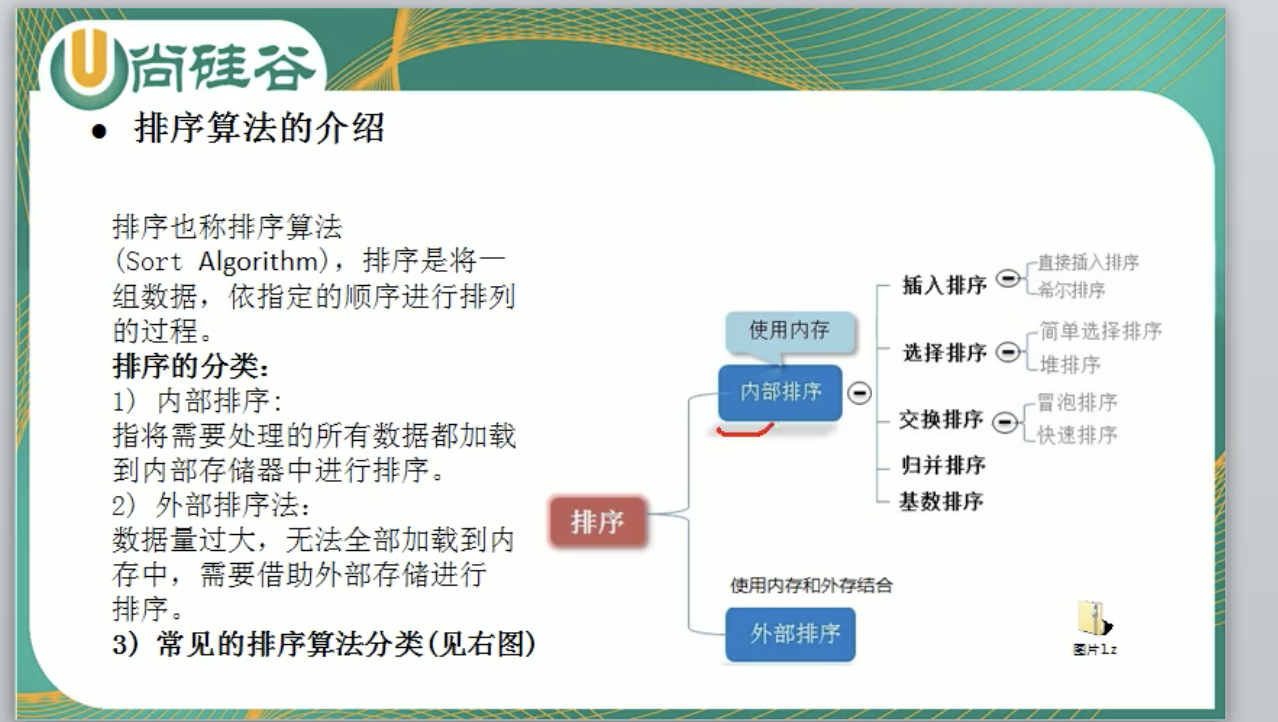
#### 2.4.3.2 逆波兰计算器(后缀)

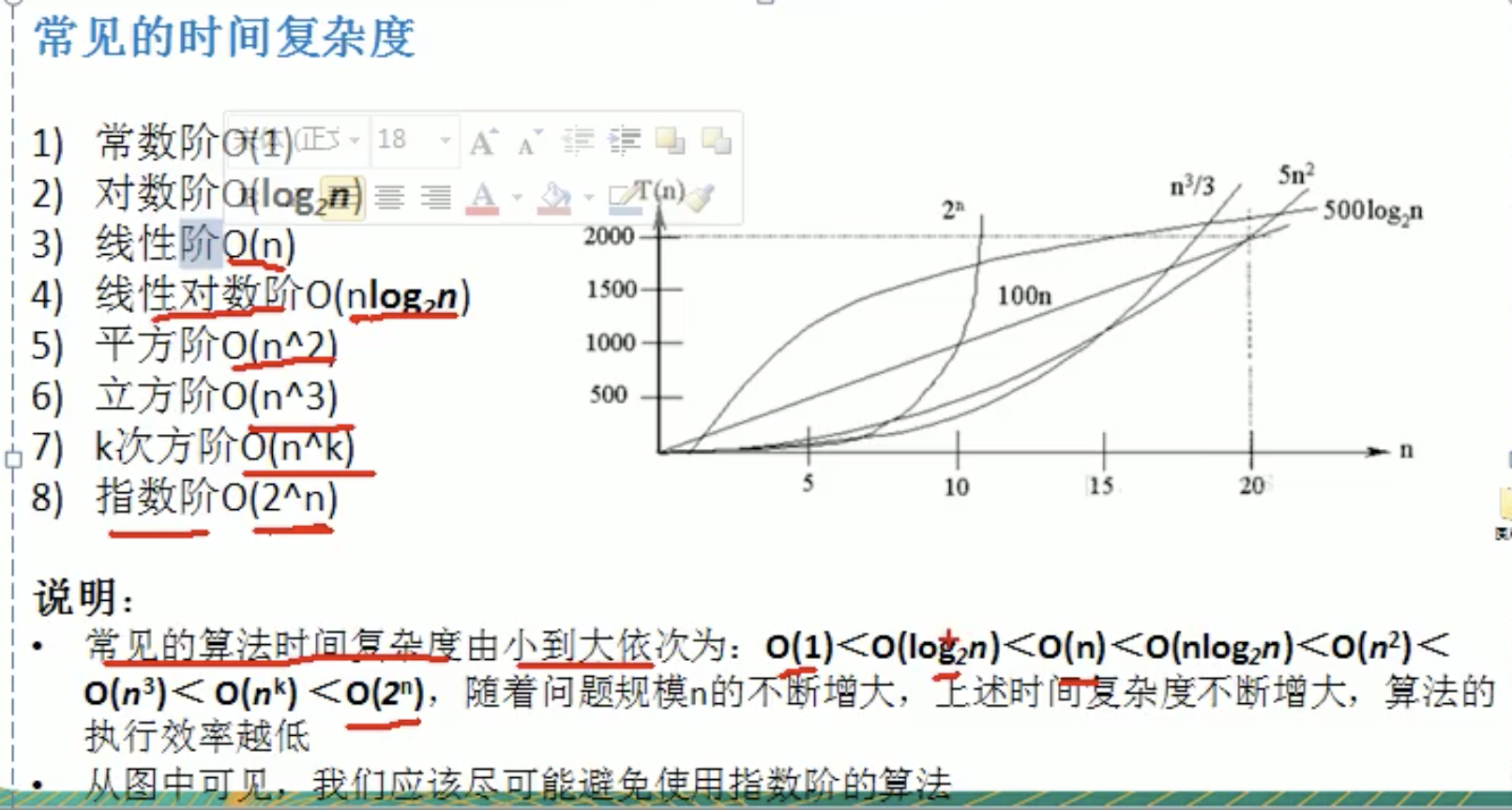
|  |
| --- |
| package com.datastructure.stack;  import java.util.ArrayList; import java.util.List; import java.util.Stack;  public class PolandNotation {   public static void main(String[] args) {   // 先定义逆波兰表达式  String suffixExpression = "3 4 + 5 \* 6 -";   // 思路  // 1.现将 "3 4 + 5 \* 6 - " => 放到ArrayList中  // 2.将ArrayList传递给一个方法，遍历配合栈 完成计算  //    List<String> list = *getListString*(suffixExpression);  System.*out*.println(list);   int res = *calculate*(list);  System.*out*.println(res);   }    //将一个逆波兰表达式，一次将数据和运算符方法到ArrayList中   public static List<String> getListString(String suffixExpression) {  // 将suffixExpression分割  String[] split = suffixExpression.split(" ");  List<String> list = new ArrayList<>();  for (String ele : split) {  list.add(ele);  }   return list;  }    public static int calculate(List<String> ls) {  // 创建栈，  Stack<String> stack = new Stack<>();    for (String item : ls) {  if (item.matches("\\d+")) {  // 直接入栈  stack.push(item);  } else {  // pop出两个数，并运算  int num2 = Integer.*parseInt*(stack.pop());  int num1 = Integer.*parseInt*(stack.pop());   int res = 0;  if (item.equals("+")) {  res = num1 + num2;  } else if (item.equals("-")) {  res = num1 - num2;   } else if (item.equals("\*")) {  res = num1 \* num2;  } else if (item.equals("/")) {  res = num1 / num2;  } else {  throw new RuntimeException("运算符有误");  }  // 结果入栈  stack.push(res + "");  }    }   // 最后留在stack中的数据就是运算结果  String pop = stack.pop();  return Integer.*parseInt*(pop);  }  } |

#### 2.4.3.3 中缀表达式转后缀

# 递归

# 排序







## 4.1 冒泡

相邻的元素进行比较，逆序则进行交换

3 9 -1 10 20

第一趟排序

1->3，9，-1，10，20

2->3，-1，9，10，20

3->3，-1，9，10，20

4->3，-1，9，10，20

第二趟排序

1->-1，3，9，10，20

2->-1，3，9，10，20

3->-1，3，9，10，20

第三趟

1->-1，3，9，10，20

2->-1，3，9，10，20

第四趟

1->-1，3，9，10，20

规则：

1. 一共进行数组的大小-1次大的循环
2. 每一堂排序的次数在逐渐的减少
3. 如果我们发现在某糖排序中，没有发生一次交换，可以提前结束冒泡排序

## 4.2 选择排序

101，34，119，1

第1趟排序：1，34，119，101，

第2趟排序：1，34，119，101，

第3趟排序：1，34，101，119，

说明：

1. 选择排序一共有数组大小-1轮排序
2. 每1轮排序，又是一个循环，循环的规则

2.1先假定当前这个数是最小数

2.2 然后和后面的每个数进行比较，如果发现有比当前数更小的数，就重新确定最小数，并得到下标

2.3 当遍历到数组的最后时，就得到本轮最小数和下标

2.4 交换代码

## 4.2 插入排序

## 4.4 希尔排序

# 