

CPT204

Lecture 1.1 - Arrays

一、数组基础：为什么需要数组？

核心问题引入

假设要读取 100 个数字并计算平均值，还要找出大于平均值的数字个数。如果为每个数字都创建一个变量，显然不现实。这时，**数组**就能派上用场，它可以用一个变量存储多个同类型的数据。

数组的定义与特点

- **定义**：数组是相同数据类型元素的集合，有固定的长度，通过索引（从 0 开始）访问元素。
- **内存存储**：数组在内存的堆区分配空间，变量存储的是数组的引用（内存地址）。

二、数组的声明与创建

1. 声明数组变量

- **语法**：

```
数据类型[] 数组名; // 推荐写法, 如 double[] myList;  
数据类型 数组名[]; // 允许但不推荐, 如 double myList[];
```

2. 创建数组（分配内存）

- **语法**：

```
数组名 = new 数据类型[长度]; // 如 myList = new double[10];
```

- **示例**：

```
double[] myList; // 声明  
myList = new double[10]; // 创建, 长度为 10
```

3. 声明与创建合并

```
double[] myList = new double[10]; // 一步完成
```

三、数组初始化

1. 默认值

- 数值型（如 `int`、`double`）：默认值为 `0`。
- 布尔型：默认值为 `false`。
- 引用类型（如 `String`）：默认值为 `null`。

2. 手动赋值

- 通过索引赋值：

```
myList[0] = 5.6; // 给第一个元素赋值  
myList[1] = 4.5; // 给第二个元素赋值
```

3. 快捷初始化（数组初始化器）

- 语法：

```
数据类型[] 数组名 = {值1, 值2, 值3, ...}; // 声明、创建、赋值一步完成
```

- 示例：

```
int[] scores = {85, 90, 78}; // 长度为 3
```

四、数组常用操作

1. 遍历数组

- 普通 `for` 循环（带索引）：

```
for (int i = 0; i < myList.length; i++) {  
    System.out.println(myList[i]); // length 是数组属性，获取长度  
}
```

- 增强 `for` 循环（`for-each`，无索引）：

```
for (double value : myList) { // 依次取出每个元素赋值给 value  
    System.out.println(value);  
}
```

```
}
```

2. 常见算法

- 求和：

```
double sum = 0;
for (double num : myList) {
    sum += num;
}
```

- 找最大值：

```
double max = myList[0]; // 假设第一个元素是最大值
for (int i = 1; i < myList.length; i++) {
    if (myList[i] > max) {
        max = myList[i]; // 发现更大值，更新 max
    }
}
```

- 数组复制：

- 错误做法（引用复制）：

```
int[] arr1 = {1, 2, 3};
int[] arr2 = arr1; // arr2 和 arr1 指向同一个数组，修改 arr2 会影响 arr1
```

- 正确做法（元素复制）：

```
int[] arr1 = {1, 2, 3};
int[] arr2 = new int[arr1.length];
for (int i = 0; i < arr1.length; i++) {
    arr2[i] = arr1[i]; // 逐个元素复制
}
```

五、数组作为方法参数

1. 传递数组（按值传递引用）

- Java 中传递数组时，实际传递的是数组的引用（内存地址），因此方法内对数组元素的修改会影响原始数组。
- 示例：

```
public static void changeArray(int[] arr) {
    arr[0] = 100; // 修改数组第一个元素
}

public static void main(String[] args) {
    int[] numbers = {1, 2, 3};
    changeArray(numbers); // 调用方法
    System.out.println(numbers[0]); // 输出 100 (原始数组被修改)
}
```

2. 返回数组

- 方法可以返回一个数组，例如反转数组：

```
public static int[] reverse(int[] list) {
    int[] result = new int[list.length];
    for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
        result[j] = list[i]; // 首尾元素交换
    }
    return result; // 返回新数组
}
```

六、搜索与排序算法

1. 线性搜索（适用于未排序数组）

- 原理：从数组第一个元素开始，逐个与目标值比较，直到找到或遍历完数组。
- 代码：

```
public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++) {
        if (list[i] == key) {
            return i; // 找到，返回索引
        }
    }
    return -1; // 未找到，返回 -1
}
```

2. 二分搜索（适用于已排序数组，效率更高）

- 原理：每次将数组分成两半，根据中间元素与目标值的大小关系，缩小搜索范围。
- 时间复杂度： $O(\log n)$ （最坏情况）。
- 代码：

```

public static int binarySearch(int[] list, int key) {
    int low = 0, high = list.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (key == list[mid]) return mid; // 找到
        else if (key < list[mid]) high = mid - 1; // 目标值在左半部分
        else low = mid + 1; // 目标值在右半部分
    }
    return -1; // 未找到
}

```

3. 选择排序（简单直观，效率较低）

- 原理：每次从剩余元素中找到最小值，与当前位置元素交换。
- 代码：

```

public static void selectionSort(int[] list) {
    for (int i = 0; i < list.length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < list.length; j++) {
            if (list[j] < list[minIndex]) {
                minIndex = j; // 更新最小值索引
            }
        }
        // 交换当前元素与最小值
        int temp = list[i];
        list[i] = list[minIndex];
        list[minIndex] = temp;
    }
}

```

七、练习与 Quiz

练习 1：数组初始化判断

下面哪个是正确的数组初始化方式？

- A. `int[] arr = new int {1, 2, 3};`
- B. `int[] arr = {1, 2, 3};`
- C. `int arr[3] = {1, 2, 3};`

答案：B（A 缺少长度；C 是 C/C++ 语法，Java 不支持）。

练习 2：增强 for 循环

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    num *= 2; // 修改的是副本，不影响原数组
}
System.out.println(numbers[2]); // 输出什么？
```

答案：3（增强 for 循环中无法通过 num 修改原数组元素）。

练习 3：二分搜索时间复杂度

二分搜索的最坏时间复杂度是？

A. $O(1)$ B. $O(n)$ C. $O(\log n)$ D. $O(n^2)$

答案：C（每次折半，时间复杂度为对数级）。

八、总结

- 重点知识：数组的声明与创建、初始化、遍历、作为方法参数传递、搜索与排序算法。
- 常见误区：
 - 数组长度不可变，创建后无法修改。
 - 直接赋值 `arr2 = arr1` 是引用复制，而非元素复制。
 - 二分搜索必须在已排序数组上使用。

Lecture 1.2 - Objects and Classes

一、核心概念：对象与类

1. 什么是对象？

- 定义：对象是现实世界中可唯一标识的实体，例如“一个圆形”“一个学生”。
- 特征：
 - 状态 (State)：用数据字段（属性）表示，如圆形的半径 `radius`。
 - 行为 (Behavior)：用方法（函数）表示，如计算面积的 `getArea()`。

2. 什么是类？

- 定义：类是对象的模板/蓝图，用于创建具有相同属性和方法的对象。
- 组成：
 - 数据字段 (Fields)：如 `private double radius;`（非静态，属于每个对象）。
 - 方法 (Methods)：如 `public double getArea()`（非静态方法需通过对象调用）。
 - 构造函数 (Constructors)：用于创建对象，名称必须与类名相同，无返回类型。

示例：Circle类

```
public class Circle {  
    private double radius = 1.0; // 数据字段（默认值1.0）  
  
    // 无参构造函数  
    public Circle() { }  
  
    // 有参构造函数  
    public Circle(double newRadius) {  
        radius = newRadius;  
    }  
  
    // 实例方法：计算面积  
    public double getArea() {  
        return radius * radius * 3.14159;  
    }  
}
```

二、创建对象与内存分析

1. 对象创建步骤

1. 声明引用变量： `Circle myCircle;`（此时变量值为 `null`，未指向任何对象）。
2. 实例化对象： `myCircle = new Circle(5.0);`（通过 `new` 关键字调用构造函数，分配内存）。

2. 内存示意图

```
myCircle → { radius: 5.0 } // 对象在堆内存中，引用变量指向对象
```

3. 访问对象成员

- 访问属性： `对象名.属性`（需注意权限，如 `private` 属性不可直接访问）。
- 调用方法： `对象名.方法名()`。

示例：

```
Circle c1 = new Circle(); // 调用无参构造，radius=1.0  
Circle c2 = new Circle(5.0); // 调用有参构造，radius=5.0  
System.out.println(c2.getArea()); // 输出：78.53975 (5²×π)
```

三、关键知识点解析

1. 静态（Static） vs 非静态（Non-static）

对比项	静态成员（类成员）	非静态成员（实例成员）
归属	属于类本身，所有对象共享	属于单个对象
访问方式	直接通过类名访问（如 <code>ClassName.x</code> ）	必须通过对象访问（如 <code>obj.x</code> ）
常见用途	工具方法（如 <code>Math.pow()</code> ）、全局计数器	对象的状态数据（如每个 <code>Circle</code> 的 <code>radius</code> ）

示例：静态变量 `numberOfObjects`

```
public class Circle {
    private static int numberOfObjects = 0; // 静态变量，记录创建的对象总数

    public Circle() {
        numberOfObjects++; // 每次创建对象时计数加1
    }

    public static int getNumberOfObjects() { // 静态方法，返回计数
        return numberOfObjects;
    }
}

// 使用方式：
System.out.println(Circle.getNumberOfObjects()); // 直接通过类名调用
```

2. 访问修饰符（Visibility Modifiers）

修饰符	作用范围	UML符号
<code>public</code>	任何类均可访问	+
<code>private</code>	仅当前类内部可访问	-
默认	同一包（package）内可访问	无符号

封装最佳实践：

- 将数据字段设为 `private`，通过访问器（getter）和修改器（setter）方法间接操作。

```
public class Circle {
    private double radius;

    // 访问器 (getter)
    public double getRadius() {
        return radius;
    }
}
```



```

// 修改器 (setter)
public void setRadius(double radius) {
    this.radius = radius; // `this`指代当前对象
}
}

```

3. 基本类型 vs 引用类型

类型	示例	赋值行为	内存存储
基本类型	<code>int i = 5;</code>	复制值（如 <code>j = i</code> 后， <code>j</code> 独立存储 5）	栈内存
引用类型	<code>Circle c = new Circle();</code>	复制引用（如 <code>c2 = c1</code> 后，两者指向同一对象）	引用在栈，对象在堆

注意：修改引用类型的属性会影响所有指向它的变量！

```

Circle c1 = new Circle(5);
Circle c2 = c1;           // c2引用c1的对象
c2.setRadius(10);         // c1和c2的radius都会变为10

```

四、UML类图快速入门

UML类图用于可视化类的结构，常见符号：

```

+-----+
|   Circle   |           // 类名
+-----+
| - radius: double |      // 私有属性 (-)
| + numberOfObjects: static int | // 静态属性 (下划线)
+-----+
| + Circle()      |       // 构造函数
| + Circle(radius: double) |
| + getArea(): double      | // 公共方法 (+)
+-----+

```

五、实战练习

练习1：定义学生类

需求：

- 定义 Student 类，包含私有字段 name (String)、age (int)、isScienceMajor (boolean)。
- 提供构造函数初始化 name 和 age，isScienceMajor 默认 false。
- 提供 getter 和 setter 方法。
- 在 main 方法中创建学生对象，设置专业为 true，并输出信息。

参考答案：

```
public class Student {
    private String name;
    private int age;
    private boolean isScienceMajor = false; // 默认值false

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // getter和setter
    public String getName() { return name; }
    public int getAge() { return age; }
    public boolean isScienceMajor() { return isScienceMajor; }
    public void setScienceMajor(boolean isScienceMajor) {
        this.isScienceMajor = isScienceMajor;
    }

    public static void main(String[] args) {
        Student stu = new Student("Alice", 20);
        stu.setScienceMajor(true);
        System.out.println(stu.getName() + "是科学专业吗? " +
            stu.isScienceMajor()); // 输出: Alice是科学专业吗? true
    }
}
```

练习2：静态方法应用

需求：在 Circle 类中添加静态方法 calculateArea(double radius)，无需创建对象即可计算面积。

参考答案：

```
public class Circle {
    // ...原有代码...

    // 静态方法：直接通过类名调用，计算任意半径的面积
    public static double calculateArea(double radius) {
        return radius * radius * 3.14159;
    }
}
```

```
}

// 使用示例：
double area = Circle.calculateArea(5); // 直接调用，无需创建对象
}
```

六、常见问题解答

1. 为什么构造函数没有返回类型？

构造函数的作用是初始化对象，不是“返回”对象，因此无需声明返回类型（包括 `void`）。

2. 静态变量什么时候初始化？

静态变量在类加载时初始化，早于对象创建，且仅初始化一次。

3. `this` 关键字的作用是什么？

`this` 指代当前对象，用于区分方法参数与类的字段（如 `this.radius = radius;`）。

Lecture 2.1 - Thinking in Objects

第一部分：核心概念

1. 不可变对象与类（Immutable Objects and Classes）

定义：对象一旦创建，内容不可修改，其类称为不可变类。

条件（缺一不可）：

1. 所有数据字段为 `private`
2. 无 `set` 方法（mutator）
3. 不返回可变对象的引用（如数组、自定义对象）

示例：不可变的 `Circle` 类

```
public class Circle {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    public double getRadius() { return radius; } // 仅有 getter, 无 setter
}
```

反例：可变的 `Student` 类

```
public class Student {
    private int id;
    private BirthDate birthDate; // BirthDate 是可变类
    public BirthDate getBirthDate() {
        return birthDate; // 返回可变对象引用，外部可通过该引用修改内部状态
    }
}
```

```
public class BirthDate {
    private int year;
    public void setYear(int newYear) { year = newYear; } // 有 setter, 可变
}
```

为什么可变?

```
Student student = new Student(...);
student.getBirthDate().setYear(2050); // 通过引用修改内部对象状态, 导致 Student
实际可变
```

练习题1:

判断以下类是否为不可变类? 为什么?

```
public class Book {
    private String title;
    private Author author; // Author 类有 public 的 name 字段
    public Book(String title, Author author) {
        this.title = title;
        this.author = author;
    }
    public Author getAuthor() { return author; }
}
public class Author {
    public String name; // 字段非 private
}
```

2. 变量作用域 (Scope of Variables)

类型	作用域	初始化规则
局部变量	从声明处到所在代码块结束	必须显式初始化后才能使用
实例变量	整个类范围	自动赋默认值 (如 int=0)
静态变量	整个类范围	自动赋默认值

示例:

```
public class ScopeDemo {
    int instanceVar; // 实例变量, 默认值 0
    static int staticVar; // 静态变量, 默认值 0

    public void method() {
        int localVar; // 局部变量, 未初始化不能使用!
        // System.out.println(localVar); // 编译错误!
        localVar = 10; // 必须先赋值
    }
}
```

```
}  
}
```

练习题2:

指出以下代码的错误:

```
public class Test {  
    public void printSum() {  
        int a = 10;  
        int b;  
        System.out.println(a + b); // 错误? 为什么?  
    }  
}
```

3. this 关键字 (The this Keyword)

作用:

1. 引用当前对象的成员: 当局部变量与成员变量同名时, 用 `this` 区分。

```
public class Person {  
    private String name;  
    public Person(String name) {  
        this.name = name; // this.name 指成员变量, name 指参数  
    }  
}
```

2. 在构造方法中调用重载的构造方法: 必须作为构造方法的第一条语句。

```
public class Circle {  
    private double radius;  
    public Circle() { this(1.0); } // 调用 Circle(double radius)  
    public Circle(double radius) { this.radius = radius; }  
}
```

注意: 静态方法中不能使用 `this` (静态方法属于类, 不属于对象)。

4. 类的抽象与封装 (Class Abstraction and Encapsulation)

核心思想:

- **抽象**: 对外暴露功能 (API), 隐藏实现细节 (如方法内部逻辑、数据存储方式)。
- **封装**: 通过 `private` 修饰符隐藏数据字段, 仅通过公共方法 (getter/setter) 访问。

示例: Loan 类的封装

```

public class Loan {
    private double annualInterestRate;
    private int numberOfYears;

    // 构造方法与公共方法 (API)
    public Loan(double rate, int years) {
        this.annualInterestRate = rate;
        this.numberOfYears = years;
    }
    public double getMonthlyPayment() {
        // 隐藏计算逻辑, 仅暴露结果
        double monthlyRate = annualInterestRate / 1200;
        return ...;
    }
}

```

优点:

- 数据安全 (防止非法修改)
- 易维护 (修改内部实现不影响外部调用)

第二部分：具体类设计实践

1. 设计 Loan 类

需求: 计算贷款的月还款额和总还款额。

属性: 年利率、贷款年限、贷款金额、创建日期

方法: 计算月还款 (getMonthlyPayment)、总还款 (getTotalPayment)

关键代码:

```

public double getMonthlyPayment() {
    double monthlyRate = annualInterestRate / 1200;
    double numerator = loanAmount * monthlyRate;
    double denominator = 1 - Math.pow(1 / (1 + monthlyRate), numberOfYears
* 12);
    return numerator / denominator;
}

```

练习题3:

修改 Loan 类, 添加一个 getDailyPayment() 方法, 计算每日还款额 (假设一年 365 天)。

2. 设计 BMI 类

需求: 计算身体质量指数 (BMI) 并判断健康状态。

公式:

$$\text{BMI} = \text{体重 (磅)} \times 0.45359237 / (\text{身高 (英寸)} \times 0.0254)^2$$

状态判断：

- <16：严重偏瘦
- 16~18：偏瘦
- 18~24：正常
- 24~29：超重
- ≥29：严重超重

关键代码：

```
public String getStatus() {  
    double bmi = getBMI();  
    if (bmi < 16) return "seriously underweight";  
    // ... 其他条件  
}
```

练习题4：

修改 BMI 类，增加一个构造方法，允许传入体重（公斤）和身高（米），并重载 getBMI() 方法。

第三部分：常用类详解

1. String 类（不可变性与常用方法）

不可变性：

```
String s = "Java";  
s = s + " HTML"; // 原 "Java" 对象不变，新建 "Java HTML" 对象
```

字符串比较：

方法	用途
<code>equals()</code>	比较内容是否相等（区分大小写）
<code>==</code>	比较引用是否指向同一对象
<code>equalsIgnoreCase()</code>	忽略大小写比较内容

示例：

```
String s1 = "Hello";  
String s2 = new String("Hello");
```

```
System.out.println(s1 == s2); // false (引用不同)
System.out.println(s1.equals(s2)); // true (内容相同)
```

常用方法：

- 截取子串： `substring(start, end)` （左闭右开）

```
"HelloWorld".substring(3, 7); // "loWo"
```

- 查找字符/子串： `indexOf("sub")` （返回首次出现的索引，未找到返回 -1）
- 替换： `replace("old", "new")` （返回新字符串，原字符串不变）

2. StringBuilder 与 StringBuffer

区别：

- `StringBuilder`：非线程安全，效率高（推荐单线程使用）
- `StringBuffer`：线程安全，效率低（适合多线程环境）

核心操作：

```
StringBuilder sb = new StringBuilder("Java");
sb.append(" HTML"); // 追加 → "Java HTML"
sb.insert(4, " and "); // 插入 → "Java and HTML"
sb.delete(0, 4); // 删除前 4 个字符 → " and HTML"
sb.reverse(); // 反转 → "MLTH and "
```

何时使用？：需要频繁修改字符串时（如循环拼接），用 `StringBuilder` 代替 `String`，避免生成大量临时对象。

3. 正则表达式 (Regular Expressions)

作用：用于字符串匹配、验证、替换、分割。

示例：

- 验证邮箱： `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
- 提取数字： `String[] nums = "a1b2c3".split("\\D");` // ["1", "2", "3"]
- 替换特殊字符： `"a$b#c".replaceAll("[\\$#]", "X");` // "aXbXc"

练习题5：

编写正则表达式，验证手机号（以 1 开头，第二位 3-9，共11位数字）。

第四部分：综合练习

设计 Course 类

需求：管理课程的学生列表，支持添加学生、获取学生列表、动态扩容。

属性：课程名、学生数组、学生人数

关键代码：

```
public class Course {
    private String courseName;
    private String[] students = new String[10]; // 初始容量 10
    private int numberOfStudents;

    public void addStudent(String student) {
        if (numberOfStudents >= students.length) {
            String[] newStudents = new String[students.length * 2];
            System.arraycopy(students, 0, newStudents, 0,
students.length);
            students = newStudents; // 扩容为原来的 2 倍
        }
        students[numberOfStudents++] = student;
    }
}
```

Lecture 2.2 - Inheritance and Polymorphism

一、继承的基本概念

1. 为什么需要继承？

- 动机：当多个类（如Circle、Rectangle）有共同属性（color、filled）和方法（getArea、getPerimeter）时，通过继承可以避免代码冗余。
 - 父类（超类）：提取公共属性和方法，如 GeometricObject。
 - 子类：继承父类，并添加特有的属性和方法，如 Circle 添加 radius 和 getDiameter。

2. 如何声明子类？

- 语法：使用 extends 关键字。

```
public class Circle extends GeometricObject {
    private double radius; // 子类特有的属性
    // 子类特有的方法
    public double getDiameter() { return 2 * radius; }
}
```

- 子类继承父类的哪些成员？
 - 继承：非 private 的属性和方法（如 color 是 private，需通过 getColor() 访问）。

- 不继承：构造方法（需显式或隐式调用父类构造方法）。

二、构造函数与 super 关键字

1. 构造函数链（Constructor Chaining）

- 规则：子类构造函数必须先调用父类构造函数（通过 `super()` 或 `this()` ），默认调用父类无参构造函数。

```
public class Rectangle extends GeometricObject {
    public Rectangle(double width, double height) {
        super(); // 隐式调用父类无参构造函数（可省略）
        this.width = width;
        this.height = height;
    }
    public Rectangle(double width, double height, String color, boolean
filled) {
        super(color, filled); // 显式调用父类有参构造函数
        this.width = width;
        this.height = height;
    }
}
```

- 示例分析：

```
class Person { public Person() { System.out.println("Person"); } }
class Employee extends Person { public Employee() { super();
System.out.println("Employee"); } }
class Faculty extends Employee { public Faculty() { super();
System.out.println("Faculty"); } }
// 执行 new Faculty() 时输出: Person → Employee → Faculty
```

2. super 关键字的作用

- 调用父类构造函数：必须作为子类构造函数的第一行。
- 调用父类被重写的方法：

```
public class Circle {
    @Override
    public String toString() {
        return "Circle radius: " + radius + ", " + super.toString(); //
调用父类的toString()
    }
}
```

三、方法重写（Override）与重载（Overload）

1. 方法重写（Override）

- 定义：子类重新实现父类的非 private、非 static、非 final 方法，方法签名必须完全一致。

```
public class GeometricObject {
    public abstract double getArea(); // 抽象方法，子类必须重写
}

public class Circle {
    @Override // 注解确保正确重写
    public double getArea() { return Math.PI * radius * radius; }
}
```

- 应用场景：多态的基础（后文会讲）。

2. 方法重载（Overload）

- 定义：同一类中，方法名相同但参数列表不同（类型、数量、顺序），与返回值无关。

```
public class Calculator {
    public int add(int a, int b) { return a + b; } // 重载1
    public double add(double a, double b) { return a + b; } // 重载2
}
```

3. 对比表格

特征	重写（Override）	重载（Overload）
发生范围	子类与父类之间	同一类中
方法签名	必须相同	必须不同（参数列表）
访问权限	子类方法不能比父类更严格（如父类 public，子类不能是 protected）	无限制
多态性	支持（动态绑定）	不支持（静态绑定）

练习1：判断以下代码是重写还是重载？

```
class A { public void m(int x) { } }
class B extends A { public void m(double x) { } } // 重载（参数类型不同）
class C extends A { public void m(int x) { } } // 重写（方法签名相同）
```

四、多态 (Polymorphism) 与动态绑定 (Dynamic Binding)

1. 多态的本质

- 定义：父类引用可以指向子类对象，运行时根据实际对象类型调用方法。

```
GeometricObject obj1 = new Circle(5); // 父类引用指向子类对象
GeometricObject obj2 = new Rectangle(5, 3);
```

- 优势：代码通用性强，如 `displayGeometricObject` 方法可接收任何 `GeometricObject` 子类对象。

```
public static void displayGeometricObject(GeometricObject obj) {
    System.out.println(obj.getArea()); // 动态绑定：根据obj实际类型调用
    Circle或Rectangle的getArea()
}
```

2. 动态绑定机制

- 过程：JVM在运行时根据对象的实际类型（而非引用类型）确定调用哪个方法。
 - 若对象是 `Circle`，调用 `Circle` 的 `getArea()`；若是 `Rectangle`，调用 `Rectangle` 的 `getArea()`。
- 示例：

```
class Person { public String toString() { return "Person"; } }
class Student extends Person { @Override public String toString() {
    return "Student"; } }
public static void main(String[] args) {
    Person p = new Student(); // 父类引用指向子类对象
    System.out.println(p.toString()); // 输出：Student（动态绑定）
}
```

五、类型转换与 `instanceof` 操作符

1. 向上转型 (Upcasting)

- 自动转换：子类对象 → 父类引用（安全，因为子类是父类的一种）。

```
Circle circle = new Circle(5);
GeometricObject geo = circle; // 向上转型（自动）
```

2. 向下转型 (Downcasting)

- **强制转换**：父类引用 → 子类对象（需确保引用实际指向子类对象，否则抛出 `ClassCastException`）。

```
GeometricObject geo = new Circle(5);
Circle circle = (Circle) geo; // 向下转型（安全，因为实际是Circle对象）
```

3. instanceof 操作符

- **作用**：在向下转型前检查对象类型，避免异常。

```
if (geo instanceof Circle) { // 先判断是否是Circle类型
    Circle circle = (Circle) geo;
    System.out.println(circle.getRadius());
}
```

六、Object 类的常用方法

1. toString() 方法

- **默认实现**：返回 类名@哈希码（如 `Circle@123456`），无实际意义。
- **重写建议**：返回对象属性的字符串表示。

```
@Override
public String toString() {
    return "Circle[radius=" + radius + ", color=" + getColor() + "]";
}
```

2. equals() 方法

- **默认实现**：比较对象引用地址（`this == obj`），而非内容。
- **重写逻辑**：
 1. 判断是否为同一对象（`this == obj`）。
 2. 判断 `obj` 是否为 `null` 或类型是否匹配（用 `instanceof`）。
 3. 比较关键属性是否相等。

```
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;
    Circle other = (Circle) obj;
```

```
return Double.compare(other.radius, radius) == 0;
}
```

七、容器类：ArrayList 与自定义栈（MyStack）

1. ArrayList 的优势

- 动态扩容：无需指定初始大小，自动扩展。
- 泛型支持：避免类型转换警告，如 `ArrayList<Circle>` 只能存储 `Circle` 对象。

```
ArrayList<Circle> circles = new ArrayList<>();
circles.add(new Circle(2));
Circle c = circles.get(0); // 无需强制转换
```

2. 自定义栈（MyStack）

- 底层实现：用 `ArrayList` 存储元素，实现栈的 `push`（入栈）、`pop`（出栈）等操作。

```
public class MyStack {
    private ArrayList<Object> list = new ArrayList<>();
    public void push(Object o) { list.add(o); } // 入栈：添加到末尾
    public Object pop() { return list.remove(list.size() - 1); } // 出
    栈：移除末尾元素
}
```

八、访问修饰符：protected 与 final

1. protected 修饰符

- 作用：允许同一包内的类和子类（即使不同包）访问。
- 对比表格：

修饰符	同一类	同一包	子类（不同包）	其他包
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
默认	✓	✓	✗	✗
private	✓	✗	✗	✗

2. final 修饰符

- **final 变量**：常量，不可修改（如 `final static double PI = 3.14;`）。
- **final 方法**：不可被重写（防止子类修改核心逻辑）。
- **final 类**：不可被继承（如 `String` 类）。

九、课堂测验与巩固

测验1：继承与构造函数

```
class A { public A() { System.out.print("A"); } public A(int x) {  
System.out.print("B"); } }  
class B extends A { public B() { super(1); } }  
new B(); // 输出?
```

答案：B（子类构造函数调用 `super(1)`，执行父类 `A(int x)` 构造函数）。

测验2：重写与多态

```
class Animal { public void speak() { System.out.println("Animal"); } }  
class Dog extends Animal { @Override public void speak() {  
System.out.println("Woof"); } }  
public static void main(String[] args) {  
    Animal a = new Dog();  
    a.speak(); // 输出?  
}
```

答案：Woof（动态绑定，调用 `Dog` 的 `speak()`）。

Lecture 3 - Abstract Classes and Interfaces

一、核心知识点总结

1. 抽象类（Abstract Classes）

- **定义**：用 `abstract` 修饰的类，包含抽象方法（`abstract` 方法，无方法体）和具体方法。
 - 例：`GeometricObject` 是抽象类，包含 `getArea()`、`getPerimeter()` 抽象方法。
- **特点**：
 - 不能直接实例化（不能用 `new` 创建对象），但可作为父类被继承。
 - 非抽象子类必须实现父类的所有抽象方法。
 - 抽象类可包含构造方法，供子类通过 `super()` 调用。
- **用途**：定义通用模板，强制子类实现特定方法（如几何图形的面积计算）。

2. 接口（Interfaces）

- **定义**：用 `interface` 声明，仅包含 **抽象方法** 和 **常量**（默认 `public static final`）。
 - 例： `Edible` 接口定义 `howToEat()` 方法， `Comparable` 接口定义 `compareTo()` 方法。
- **特点**：
 - 类通过 `implements` 实现接口，需重写所有接口方法（默认 `public abstract`）。
 - 支持多继承（一个类可实现多个接口），弥补Java单继承限制。
 - 接口可继承其他接口（ `interface A extends B, C` ）。
- **用途**：定义行为规范（如“可比较”“可克隆”），解耦实现与接口。

3. 关键接口示例

- **Comparable 接口**：
 - 用于对象排序（如 `Arrays.sort()` ），需实现 `compareTo(Object o)` 方法。
 - 例：自定义 `ComparableRectangle` 类，通过面积比较大小。
- **Cloneable 接口**：
 - 标记接口（无方法），允许对象通过 `clone()` 方法克隆。
 - 注意： `Object.clone()` 是浅拷贝，深拷贝需手动重写 `clone()` 。

4. 抽象类 vs. 接口

对比维度	抽象类	接口
成员	可包含抽象方法、具体方法、变量	只能包含抽象方法和常量（ <code>public static final</code> ）
继承/实现	单继承（ <code>extends</code> ）	多实现（ <code>implements</code> ）
构造方法	有（供子类调用）	无
实例化	不能直接实例化	不能直接实例化
设计目的	定义类的模板或部分实现	定义行为规范（“能做什么”）

5. 包装类（Wrapper Classes）

- **作用**：将基本数据类型（如 `int`、`double` ）封装为对象，便于集合类使用。
- **常见类**： `Integer`、`Double`、`Boolean` 等，均继承自 `Number` 类。
- **特性**：
 - 不可变（对象创建后值不可改）。
 - 支持自动装箱/拆箱（JDK 1.5+）：


```
Integer num = 10; // 自动装箱 (int → Integer)
int n = num;      // 自动拆箱 (Integer → int)
```

- 常用方法：
 - 转换： `parseInt(String s)`、`doubleValue()`。
 - 比较： `compareTo()`（实现 `Comparable` 接口）。

6. 大数类（`BigInteger` & `BigDecimal`）

- 用途：处理超出基本类型范围的大整数（`BigInteger`）或高精度小数（`BigDecimal`）。
- 特点：
 - 不可变，方法返回新对象（如 `add()`、`multiply()`）。
 - `BigDecimal` 需指定精度和舍入模式：

```
BigDecimal result = a.divide(b, 20, BigDecimal.ROUND_HALF_UP);
```

7. 案例：`Rational`类（有理数）

- 功能：实现有理数的加减乘除、比较和类型转换（继承 `Number` 接口）。
- 关键点：
 - 重写 `compareTo()` 实现比较逻辑。
 - 使用 `gcd()` 方法约分，确保分母为正。

二、常见问题与解答

1. Q：抽象类可以没有抽象方法吗？

A：可以。抽象类的存在意义是禁止实例化，即使没有抽象方法，也可作为基类强制子类继承。

2. Q：接口能继承类吗？

A：不能。接口只能继承其他接口（`interface A extends B`），类通过 `implements` 实现接口。

3. Q：为什么包装类是不可变的？

A：为保证线程安全和哈希值稳定（如作为 `HashMap` 键时），包装类的内部状态不可修改。

4. Q：深拷贝和浅拷贝的实现区别？

A：浅拷贝直接复制引用（`super.clone()`），深拷贝需手动创建新对象并复制引用字段（如 `h.whenBuilt = (Date)whenBuilt.clone();`）。

Lecture 4 - Generics

一、泛型基础：为何需要泛型？

核心作用

泛型能够将类型检查的阶段提前到编译期，避免在运行时出现类型错误，同时还能实现代码的复用。下面通过一个示例来对比说明：

- 无泛型（存在风险）

```
ArrayList list = new ArrayList();  
list.add("1"); // 向列表中添加字符串  
Integer i = (Integer) list.get(0); // 运行时会抛出 ClassCastException 异常
```

存在的问题：在运行时才能发现类型不匹配的问题。

- 有泛型（更安全）

```
ArrayList<Integer> list = new ArrayList<>(); // JDK 1.7 及之后版本支持钻石语法  
list.add("1"); // 编译时就会报错，提示类型不匹配  
list.add(1); // 正确操作，自动装箱  
Integer i = list.get(0); // 无需进行强制类型转换
```

优势体现：在编译阶段就能捕获错误，代码的可读性也更强。

关键概念

泛型的本质是对类型进行参数化。就像定义一个“模板”，在使用的时候再指定具体的类型。例如：

- 泛型类模板： `class GenericStack<E>`
 - E 是类型参数（也被称为类型变量），在创建实例时，需要用具体的类型（如 `Integer`、`String`）来替换它。
- 实例化方式： `GenericStack<String> stack = new GenericStack<>();`

二、动手实践：定义泛型类与接口

案例：自定义泛型栈（GenericStack）

```
public class GenericStack<E> {  
    private ArrayList<E> list = new ArrayList<>(); // 使用 E 作为元素类型  
  
    public void push(E o) { list.add(o); } // 只能添加 E 类型的元素  
    public E pop() { return list.remove(list.size() - 1); } // 返回 E 类型的元素  
}
```

```
// 其他方法如 peek()、isEmpty() 等的返回类型或参数类型均为 E
}
```

使用方式：

```
GenericStack<Integer> intStack = new GenericStack<>();
intStack.push(10); // 自动装箱为 Integer
intStack.push(20);
System.out.println(intStack.pop()); // 输出：20（类型为 Integer）
```

练习 1

尝试定义一个泛型队列 `GenericQueue<E>`，要求包含 `enqueue(E element)`（入队）和 `dequeue()`（出队）方法。

三、深入探究：泛型方法与有界类型

1. 泛型方法

泛型方法可以独立于类而存在，在方法名前使用 `<T>` 来声明类型参数。

```
public static <E> void printArray(E[] array) { // 适用于任何类型的数组
    for (E element : array) {
        System.out.print(element + " ");
    }
}
// 使用示例
Integer[] intArray = {1, 2, 3};
printArray(intArray); // 输出：1 2 3
```

2. 有界类型参数（Bounded Type Parameters）

通过 `extends` 关键字来限制类型参数的范围，例如 `<E extends Comparable<E>>` 表示 `E` 必须实现 `Comparable` 接口。

案例：排序方法

```
public static <E extends Comparable<E>> void sort(E[] array) {
    // 使用 compareTo() 方法进行比较排序
    for (int i = 0; i < array.length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < array.length; j++) {
            if (array[j].compareTo(array[minIndex]) < 0) {
                minIndex = j;
            }
        }
        // 交换元素
    }
}
```

```

        E temp = array[i];
        array[i] = array[minIndex];
        array[minIndex] = temp;
    }
}
// 适用类型: Integer、String (因为它们都实现了 Comparable 接口)

```

练习 2

定义一个泛型方法 `max(E a, E b)`，用于返回两个可比较对象中的最大值，要求使用有界类型参数。

四、通配符 (Wildcards)：解决类型兼容性问题

通配符用于处理泛型类型之间的继承关系，常见的有以下三种：

通配符	含义
<code>?</code>	无界通配符，表示任意类型
<code>? extends T</code>	上界通配符，表示类型必须是 T 的子类（包括 T 本身）
<code>? super T</code>	下界通配符，表示类型必须是 T 的父类（包括 T 本身）

案例：计算栈中数字的最大值

```

public static double max(GenericStack<? extends Number> stack) { // 接受
    Number 及其子类 (如 Integer、Double)
    double max = stack.pop().doubleValue();
    while (!stack.isEmpty()) {
        double value = stack.pop().doubleValue();
        if (value > max) max = value;
    }
    return max;
}
// 使用示例
GenericStack<Integer> intStack = new GenericStack<>();
intStack.push(10);
intStack.push(20);
System.out.println(max(intStack)); // 输出: 20.0

```

注意事项

- **父子类型关系：**`GenericStack<Integer>` **不是** `GenericStack<Number>` 的子类，但 `GenericStack<Integer>` 是 `GenericStack<? extends Number>` 的子类。
- **写入限制：**使用 `? extends T` 时，无法向集合中写入元素（除了 `null`），因为类型不确定；使用 `? super T` 时，允许写入 `T` 及其子类元素。

练习 3

思考以下代码是否能编译通过，并解释原因：

```
GenericStack<? extends Number> stack = new GenericStack<Integer>();
stack.push(new Integer(1)); // 能否编译通过? NO
stack.push(new Number(1) {}); // 能否编译通过? NO
```

五、类型擦除（Type Erasure）：Java 泛型的实现原理

核心机制

- 编译器会在编译阶段移除泛型类型信息，将其替换为原始类型（如 E 替换为 Object）。
- 因此，泛型类型在运行时并不存在，例如：

```
System.out.println(new GenericStack<Integer>().getClass() == new
GenericStack<String>().getClass()); // 输出: true
```

这是因为擦除后两者的类型都是 `GenericStack`（原始类型）。

限制条件

由于类型擦除的存在，泛型有以下限制：

1. **不能实例化泛型类型**：`new E()` 是不允许的，因为运行时 E 会被擦除为 `Object`，无法确定具体类型。
2. **静态成员不能使用类型参数**：静态变量或方法属于类级别，而类型参数是实例级别的。
3. **数组操作受限**：不能创建泛型数组，如 `E[] array = new E[10];`，但可以使用 `ArrayList<E>`。

六、实战应用：设计泛型矩阵类

文档中提供了 `GenericMatrix<E>` 的示例，它是一个抽象类，通过泛型实现了矩阵的加法和乘法运算，具体的元素操作由子类（如 `IntegerMatrix`、`RationalMatrix`）来实现。

```
// 抽象泛型矩阵类
public abstract class GenericMatrix<E extends Number> {
    protected abstract E add(E a, E b); // 元素加法
    protected abstract E multiply(E a, E b); // 元素乘法
    protected abstract E zero(); // 返回零值

    public E[][] addMatrix(E[][] m1, E[][] m2) {
        // 实现矩阵加法，调用 add() 方法
    }
}
```

```
// 整数矩阵子类
public class IntegerMatrix extends GenericMatrix<Integer> {
    @Override
    protected Integer add(Integer a, Integer b) { return a + b; }
    @Override
    protected Integer multiply(Integer a, Integer b) { return a * b; }
    @Override
    protected Integer zero() { return 0; }
}
```

练习 4

尝试为 `GenericMatrix` 添加一个 `RationalMatrix` 子类，用于处理有理数矩阵的运算（有理数类 `Rational` 已在文档中定义）。

七、常见问题与最佳实践

1. **原始类型（Raw Types）**：为了兼容旧代码，可以使用原始类型（如 `ArrayList`），但这会失去泛型的类型安全保障，不建议在新代码中使用。
2. **优先使用泛型集合**：避免使用 `ArrayList`，而应使用 `ArrayList<String>` 等参数化类型。
3. **合理使用通配符**：在需要处理类型层次结构时，使用通配符来提高代码的灵活性。

总结：知识图谱

泛型

- ├ 核心优势：编译时类型检查、代码复用
- ├ 基础语法：泛型类/接口 (`GenericStack<E>`)、泛型方法 (`<T> void print(T[] arr)`)
- ├ 类型参数：无界类型 (`E`)、有界类型 (`E extends Comparable<E>`)
- ├ 通配符：`?`、`? extends T`、`? super T`
- ├ 实现原理：类型擦除、原始类型
- └ 实战：自定义泛型数据结构、泛型算法（排序、矩阵运算）

Lecture 5 - Lists, Stacks, Queues, and Priority Queues

第一部分：Java集合框架概述

1. 集合框架的核心结构

Java集合框架（Java Collections Framework, JCF）是一组用于存储和操作数据的API，包含接口、抽象类和具体类。

- **接口**：定义操作规范（如 `Collection`、`List`、`Queue`）。

- **抽象类**：提供部分实现（如 `AbstractCollection`、`AbstractList`）。
- **具体类**：实现具体数据结构（如 `ArrayList`、`LinkedList`、`PriorityQueue`）。

2. 核心接口分类

- **单元素集合（Collection）**：存储独立元素，分为：
 - **有序列表（List）**：允许重复，有序（如 `ArrayList`、`LinkedList`）。
 - **集合（Set）**：不允许重复（如 `HashSet`、`TreeSet`）。
 - **队列（Queue）**：先进先出（FIFO）或优先级排序（如 `PriorityQueue`）。
- **键值对集合（Map）**：存储键值对（如 `HashMap`、`TreeMap`）。

第二部分：列表（List）

1. List接口特点

- **有序性**：元素按插入顺序存储，可通过索引访问。
- **允许重复**：可存储相同元素。
- **核心方法**：
 - `add(index, element)`：在指定位置插入元素。
 - `get(index)`：获取指定索引的元素。
 - `remove(index)`：删除指定索引的元素。
 - `size()`：获取元素个数。

2. 实现类：ArrayList vs. LinkedList

特性	ArrayList	LinkedList
数据结构	动态数组	双向链表
随机访问	快 ($O(1)$)	慢 ($O(n)$, 需遍历链表)
插入/删除	尾部快, 中间/头部慢 (需移动元素)	快 ($O(1)$, 仅需修改指针)
适用场景	频繁查询、较少增删	频繁增删、首尾操作

3. 示例代码：List基本操作

```
import java.util.ArrayList;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        // 创建ArrayList
        List<String> list = new ArrayList<>();

        // 添加元素
```

```

list.add("Apple");
list.add("Banana");
list.add(1, "Cherry"); // 在索引1处插入元素

// 遍历列表
System.out.println("遍历列表 (for-each) :");
for (String item : list) {
    System.out.print(item + " "); // 输出: Apple Cherry Banana
}

// 删除元素
list.remove("Banana");
System.out.println("\n删除后的列表: " + list); // 输出: [Apple,
Cherry]

// 获取元素
String firstItem = list.get(0);
System.out.println("第一个元素: " + firstItem); // 输出: Apple
}
}

```

第三部分：栈（Stack）

1. 栈的特点

- 后进先出（LIFO）：最后插入的元素最先取出。
- 核心方法：
 - push(element)：压入元素（栈顶）。
 - pop()：弹出栈顶元素（并删除）。
 - peek()：查看栈顶元素（不删除）。
 - empty()：判断栈是否为空。

2. 示例代码：栈的基本操作

```

import java.util.Stack;

public class StackDemo {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();

        // 压入元素
        stack.push("Java");
        stack.push("Python");
        stack.push("C++");

        // 弹出元素
        System.out.println("弹出元素: " + stack.pop()); // 输出: C++
    }
}

```



```

        System.out.println("栈顶元素: " + stack.peek()); // 输出: Python

        // 判断栈是否为空
        System.out.println("栈是否为空: " + stack.empty()); // 输出: false
    }
}

```

第四部分：队列（Queue）和优先队列（PriorityQueue）

1. 队列（Queue）

- 先进先出（FIFO）：元素从队尾插入，队头取出。
- 核心方法：
 - `offer(element)`：添加元素到队尾（推荐用此方法，避免异常）。
 - `poll()`：取出队头元素（队列为空时返回 `null`）。
 - `peek()`：查看队头元素（队列为空时返回 `null`）。

2. 优先队列（PriorityQueue）

- 按优先级排序：元素按自然顺序（`Comparable`）或自定义顺序（`Comparator`）排列，优先级高的元素先取出。
- 示例代码：优先队列排序

```

import java.util.PriorityQueue;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        // 默认按自然顺序（字符串字典序）
        PriorityQueue<String> queue = new PriorityQueue<>();
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");

        System.out.println("默认排序（字典序）：");
        while (!queue.isEmpty()) {
            System.out.print(queue.poll() + " "); // 输出: Georgia Indiana
            Oklahoma
        }

        // 自定义排序（逆序）
        PriorityQueue<String> reverseQueue = new PriorityQueue<>(
            (a, b) -> b.compareTo(a) // 匿名内部类实现Comparator
        );
        reverseQueue.offer("Oklahoma");
        reverseQueue.offer("Indiana");
        reverseQueue.offer("Georgia");
    }
}

```

```

        System.out.println("\n逆序排序:");
        while (!reverseQueue.isEmpty()) {
            System.out.print(reverseQueue.poll() + " "); // 输出: Oklahoma
Indiana Georgia
        }
    }
}

```

第五部分：集合工具类（Collections）

Collections 类提供大量静态方法，用于操作集合：

- 排序： `sort(list)`（自然顺序）、`sort(list, comparator)`（自定义顺序）。
- 搜索： `binarySearch(list, key)`（需先排序）。
- 其他： `reverse(list)`（反转列表）、`shuffle(list)`（打乱顺序）、`frequency(collection, element)`（统计元素出现次数）。

示例代码：排序与搜索

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsDemo {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>(Arrays.asList(3, 1, 4, 1, 5, 9));

        // 排序（自然顺序）
        Collections.sort(list);
        System.out.println("排序后: " + list); // 输出: [1, 1, 3, 4, 5, 9]

        // 搜索元素
        int index = Collections.binarySearch(list, 4);
        System.out.println("元素4的索引: " + index); // 输出: 3

        // 反转列表
        Collections.reverse(list);
        System.out.println("反转后: " + list); // 输出: [9, 5, 4, 3, 1, 1]
    }
}

```

第六部分：练习与思考

练习1：List性能对比

编写代码对比 ArrayList 和 LinkedList 在中间插入元素的性能差异，思考为什么会有这样的结果。

练习2：栈的应用

使用 Stack 实现一个简单的计算器，计算后缀表达式（如 "3 4 + 5 *"）。

Expression	Scan	Action	operandStack	operatorStack
(1 + 2)*4 - 3 ↑	(Phase 1.4		(
(1 + 2)*4 - 3 ↑	1	Phase 1.1	1	(
(1 + 2)*4 - 3 ↑	+	Phase 1.2	1	+
(1 + 2)*4 - 3 ↑	2	Phase 1.1	2 1	+
(1 + 2)*4 - 3 ↑)	Phase 1.5	3	
(1 + 2)*4 - 3 ↑	*	Phase 1.3	3	*
(1 + 2)*4 - 3 ↑	4	Phase 1.1	4 3	*
(1 + 2)*4 - 3 ↑	-	Phase 1.2	12	-
(1 + 2)*4 - 3 ↑	3	Phase 1.1	3 12	-
(1 + 2)*4 - 3 ↑	none	Phase 2	9	

等价于中缀表达式：(3 + 4) * 5 = 35

```
import java.util.Stack;

public class PostfixCalculator {

    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();
        String[] tokens = expression.split(" ");

        for (String token : tokens) {
            if (isOperator(token)) {
                int b = stack.pop(); // 注意：先出栈的是右操作数
                int a = stack.pop();
                int result = applyOperator(a, b, token);
                stack.push(result);
            } else {
                stack.push(Integer.parseInt(token));
            }
        }
    }
}
```

```

        return stack.pop();
    }

    private static boolean isOperator(String token) {
        return "+-*/".contains(token);
    }

    private static int applyOperator(int a, int b, String operator) {
        switch (operator) {
            case "+": return a + b;
            case "-": return a - b;
            case "*": return a * b;
            case "/": return a / b; // 假设输入合法, b ≠ 0
            default: throw new IllegalArgumentException("未知操作符: " +
operator);
        }
    }

    public static void main(String[] args) {
        String expr = "3 4 + 5 *";
        int result = evaluatePostfix(expr);
        System.out.println("结果: " + result); // 输出: 35
    }
}

```

练习3：优先队列自定义排序

定义一个 Student 类（包含姓名和成绩），使用 PriorityQueue 按成绩从高到低排序学生。

Lecture 6 - Sets and Maps

第一部分：Set（集合）

1. Set 基础概念

- 定义：Set 是一种不允许存储重复元素的集合，继承自 Collection 接口。
- 核心特性：
 - 无序性：元素存储顺序不确定（除了 LinkedHashSet 和 TreeSet）。
 - 唯一性：通过 equals() 方法判断元素是否重复，确保无重复元素。
- 实现类：
 - HashSet：基于哈希表实现，无序，查询速度快。
 - LinkedHashSet：继承自 HashSet，通过链表维护插入顺序，遍历时按插入顺序输出。
 - TreeSet：基于红黑树实现，元素有序（自然排序或自定义排序）。

2. HashSet 详解

2.1 创建 HashSet

```
// 空 HashSet
Set<String> set1 = new HashSet<>();

// 从现有集合创建 (如 List)
List<String> list = Arrays.asList("Apple", "Banana");
Set<String> set2 = new HashSet<>(list);

// 指定初始容量和负载因子 (默认容量16, 负载因子0.75)
Set<Integer> set3 = new HashSet<>(20, 0.8f);
```

2.2 添加元素 (add())

- 唯一性验证: 通过 hashCode() 和 equals() 判断重复。
 - 示例: 向 HashSet 中添加重复元素 (字符串会自动去重)。

```
Set<String> cities = new HashSet<>();
cities.add("London");
cities.add("Paris");
cities.add("New York");
cities.add("New York"); // 重复, 不会被添加
System.out.println(cities); // 输出: [London, Paris, New York] (顺序不确定)
```

2.3 遍历 HashSet

- 增强 for 循环:

```
for (String city : cities) {
    System.out.print(city + " ");
}
```

- forEach() 方法 (Lambda 表达式):

```
cities.forEach(city -> System.out.print(city.toLowerCase() + " "));
```

2.4 常用方法

方法	说明
<code>remove(obj)</code>	删除指定元素
<code>contains(obj)</code>	检查是否包含元素
<code>size()</code>	获取元素个数
<code>clear()</code>	清空集合

3. LinkedHashMap 详解

- 特点：保持元素的插入顺序，遍历时按插入顺序输出。
- 示例：

```
Set<String> linkedSet = new LinkedHashMap<>();
linkedSet.add("A");
linkedSet.add("B");
linkedSet.add("C");
System.out.println(linkedSet); // 输出: [A, B, C] (顺序与插入一致)
```

4. TreeSet 详解

- 特点：元素自动排序，支持自然排序（Comparable）或自定义排序（Comparator）。
- 示例：自然排序（字符串按字母顺序）

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("Z");
treeSet.add("A");
treeSet.add("B");
System.out.println(treeSet); // 输出: [A, B, Z] (升序排列)
```

- 自定义排序（Comparator）：

```
// 按字符串长度降序排列
Set<String> treeSet = new TreeSet<>
(Comparator.comparingInt(String::length).reversed());
treeSet.add("Apple"); // 5字母
treeSet.add("Banana"); // 6字母
treeSet.add("Cat"); // 3字母
System.out.println(treeSet); // 输出: [Banana, Apple, Cat]
```

5. Set vs List 性能对比

- 场景总结：

- **Set**: 适合存储唯一元素，查询效率高（尤其是 `HashSet`）。
- **List**: 适合需要索引访问的场景（如 `ArrayList`、`LinkedList`）。
- 性能测试示例（判断元素是否存在）：

```
// HashSet 耗时约 10-50ms
long time = System.currentTimeMillis();
for (int i = 0; i < 100000; i++) {
    set.contains(i);
}
System.out.println("HashSet耗时: " + (System.currentTimeMillis() - time)
+ "ms");
```

第二部分：Map（映射）

1. Map 基础概念

- 定义：存储键值对（key-value）的集合，通过 key 快速查找 value。
- 核心特性：
 - 键唯一性：key 不能重复（通过 `equals()` 判断）。
 - 值可重复：value 可以重复。
- 实现类：
 - `HashMap`：无序，基于哈希表，查询速度快。
 - `LinkedHashMap`：保持插入顺序或访问顺序。
 - `TreeMap`：按键排序（自然排序或自定义排序）。

2. HashMap 详解

2.1 创建 HashMap

```
// 空 HashMap
Map<String, Integer> map1 = new HashMap<>();

// 初始化键值对
Map<String, Integer> map2 = new HashMap<>() {{
    put("Apple", 10);
    put("Banana", 20);
}};

// 从现有 Map 创建
Map<String, Integer> map3 = new HashMap<>(map2);
```

2.2 操作方法

- 添加/更新键值对：`put(key, value)`（若 key 已存在，覆盖原有 value）。

```
map1.put("Orange", 30); // 添加
map1.put("Apple", 15); // 更新 Apple 的值为 15
```

- 获取值：get(key)（若 key 不存在，返回 null）。

```
int value = map1.get("Apple"); // value = 15
```

- 遍历键值对：

```
// 遍历所有键
for (String key : map1.keySet()) {
    System.out.println("Key: " + key + ", Value: " + map1.get(key));
}

// 遍历所有键值对（推荐）
for (Map.Entry<String, Integer> entry : map1.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}

// 使用 forEach()
map1.forEach((key, value) -> System.out.println(key + ": " + value));
```

2.3 常用方法

方法	说明
remove(key)	删除指定键的键值对
containsKey(key)	检查是否存在指定键
size()	获取键值对个数
keySet()	获取所有键的 Set
values()	获取所有值的 Collection

3. LinkedHashMap 详解

- 特点：
 - 插入顺序：默认按插入顺序存储和遍历。
 - 访问顺序：设置 accessOrder = true 时，按最后访问顺序排序（最近最少访问 → 最近最多访问）。
- 示例：访问顺序


```
Map<String, Integer> linkedMap = new LinkedHashMap<>(16, 0.75f, true);
linkedMap.put("A", 1);
linkedMap.put("B", 2);
linkedMap.get("A"); // 访问键 "A"
System.out.println(linkedMap); // 输出: [B, A] ("A" 被移动到最后)
```

4. TreeMap 详解

- 特点：按键的自然排序或自定义排序（Comparator）排列。
- 示例：自然排序（字符串按字母顺序）

```
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("Z", 26);
treeMap.put("A", 1);
treeMap.put("B", 2);
System.out.println(treeMap); // 输出: {A=1, B=2, Z=26} (键升序)
```

- 自定义排序（按值降序）：

```
Map<String, Integer> treeMap = new TreeMap<>
(Comparator.comparingInt(Map.Entry::getValue).reversed());
treeMap.put("Apple", 15);
treeMap.put("Banana", 20);
treeMap.put("Cherry", 10);
System.out.println(treeMap); // 输出: {Banana=20, Apple=15, Cherry=10}
(值降序)
```

第三部分：关键练习与注意事项

1. 练习建议

1. Set 练习：

- 创建一个 HashSet，存储自定义对象 Person（包含姓名和年龄），观察去重效果。若去重失败，重写 equals() 和 hashCode() 方法。
- 使用 TreeSet 对整数列表进行排序，尝试自定义排序规则（如降序）。

2. Map 练习：

- 创建一个 HashMap，统计字符串中每个字符的出现次数（例如："abracadabra" → a:5, b:2, r:2, c:1, d:1）。
- 使用 LinkedHashMap 实现一个简单的LRU缓存（最近最少使用），设置 accessOrder = true，当容量超过限制时删除最旧的元素。

2. 注意事项

- **Set 去重**：自定义对象必须重写 equals() 和 hashCode()，否则 HashSet 无法正确去重。
- **Map 键的选择**：建议使用不可变对象（如 String、Integer）作为键，避免键值改变导致哈希冲突。
- **性能优化**：
 - HashSet / HashMap 适合高频查询和插入。
 - TreeSet / TreeMap 适合需要排序的场景，但性能略低于哈希结构。

总结：核心知识点速查表

类型	实现类	顺序性	去重机制	典型场景
Set	HashSet	无序	hashCode() + equals()	快速去重、唯一性校验
	LinkedHashSet	插入顺序	同上	需要保持插入顺序的场景
	TreeSet	自然/自定义排序	compareTo() / Comparator	排序集合、范围查询
Map	HashMap	无序	hashCode() + equals()	键值对快速查找
	LinkedHashMap	插入/访问顺序	同上	日志记录、LRU缓存
	TreeMap	自然/自定义排序	compareTo() / Comparator	按键排序的统计、范围查询

Lecture 8 - Developing Efficient Algorithms

第一讲：算法效率与大O表示法

1. 为什么需要分析算法效率？

- **问题**：同一任务可能有不同算法（如线性搜索 vs 二分搜索），如何比较它们的效率？
- **解决方案**：用大O表示法（Big O Notation）衡量算法的时间复杂度，关注输入规模增长时的性能变化趋势。

2. 大O表示法的核心思想

- **忽略常数和低阶项**：例如，100n和n/2的时间复杂度均为O(n)。
- **关注最坏情况**：分析算法在最坏输入下的表现（因为平均情况通常与最坏情况同阶）。

3. 常见时间复杂度排序

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

- 示例：
 - 常数时间 $O(1)$ ：数组索引访问。
 - 线性时间 $O(n)$ ：线性搜索。
 - 对数时间 $O(\log n)$ ：二分搜索。
 - 平方时间 $O(n^2)$ ：选择排序、插入排序。

第二讲：常见算法的时间复杂度分析

1. 线性搜索 vs 二分搜索

- 线性搜索：
 - 代码：

```
public static int linearSearch(int[] list, int key) {  
    for (int i = 0; i < list.length; i++) {  
        if (list[i] == key) return i;  
    }  
    return -1;  
}
```

- 复杂度： $O(n)$ （最坏情况需遍历所有元素）。
- 二分搜索：
 - 条件：数组必须有序。
 - 代码：

```
public static int binarySearch(int[] list, int key) {  
    int low = 0, high = list.length - 1;  
    while (low <= high) {  
        int mid = (low + high) / 2;  
        if (key < list[mid]) high = mid - 1;  
        else if (key == list[mid]) return mid;  
        else low = mid + 1;  
    }  
    return -1;  
}
```

- 复杂度： $O(\log n)$ （每次将问题规模减半）。

2. 选择排序

- 思想：每次从未排序部分选择最小元素，与未排序部分的第一个元素交换。
- 代码：

```

public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length; i++) {
        int minIndex = i;
        for (int j = i + 1; j < list.length; j++) {
            if (list[j] < list[minIndex]) minIndex = j;
        }
        double temp = list[i];
        list[i] = list[minIndex];
        list[minIndex] = temp;
    }
}

```

- 复杂度： $O(n^2)$ （双重循环）。

3. 插入排序

- 思想：将未排序元素逐个插入已排序部分的正确位置。
- 代码：

```

public static void insertionSort(int[] list) {
    for (int i = 1; i < list.length; i++) {
        int current = list[i];
        int j = i - 1;
        while (j >= 0 && list[j] > current) {
            list[j + 1] = list[j];
            j--;
        }
        list[j + 1] = current;
    }
}

```

- 复杂度： $O(n^2)$ （最坏情况下需移动所有元素）。

第三讲：递归与动态规划

1. 斐波那契数列的递归实现

- 递归公式： $fib(n) = fib(n - 1) + fib(n - 2)$ 。
- 代码：

```

public static int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

```

- 复杂度： $O(2^n)$ （重复计算大量子问题）。

2. 动态规划优化斐波那契数列

- 思想：用数组存储已计算的子问题结果，避免重复计算。
- 代码：

```
public static int fib(int n) {  
    if (n <= 1) return n;  
    int[] dp = new int[n + 1];  
    dp[0] = 0;  
    dp[1] = 1;  
    for (int i = 2; i <= n; i++) {  
        dp[i] = dp[i-1] + dp[i-2];  
    }  
    return dp[n];  
}
```

- 复杂度： $O(n)$ （线性时间）。

第四讲：分治法与回溯法

1. 最近点对问题（分治法）

- 步骤：
 1. 将点按x坐标排序。
 2. 递归求解左右两半的最近点对。
 3. 合并时检查中间带内的点对。
- 复杂度： $O(n \log n)$ （排序和递归合并）。

2. 八皇后问题（回溯法）

- 思想：逐行放置皇后，若当前位置冲突则回溯到上一行。
- 代码关键逻辑：

```
private boolean isValid(int row, int col) {  
    for (int i = 0; i < row; i++) {  
        if (queens[i] == col || Math.abs(row - i) == Math.abs(col -  
queens[i])) {  
            return false;  
        }  
    }  
    return true;  
}
```

- 复杂度： $O(n!)$ （理论上最坏情况，但通过剪枝优化后实际效率更高）。

第五讲：其他高效算法

1. 欧几里得算法（求GCD）

- 思想：利用 $\text{gcd}(m, n) = \text{gcd}(n, m \% n)$ 。
- 代码：

```
public static int gcd(int m, int n) {  
    if (n == 0) return m;  
    return gcd(n, m % n);  
}
```

- 复杂度： $O(\log n)$ 。

2. 筛法求素数（Sieve of Eratosthenes）

- 思想：标记非素数，从2开始筛去所有倍数。
- 代码：

```
public static boolean[] sieve(int n) {  
    boolean[] isPrime = new boolean[n + 1];  
    Arrays.fill(isPrime, true);  
    isPrime[0] = isPrime[1] = false;  
    for (int i = 2; i * i <= n; i++) {  
        if (isPrime[i]) {  
            for (int j = i * i; j <= n; j += i) {  
                isPrime[j] = false;  
            }  
        }  
    }  
    return isPrime;  
}
```

- 复杂度： $O(n \log \log n)$ 。

总结与练习

1. 大O表示法：分析算法复杂度的核心工具，关注主导项。
2. 排序算法：选择排序和插入排序均为 $O(n^2)$ ，适合小规模数据。
3. 递归与动态规划：动态规划通过存储子问题结果优化递归的指数复杂度。
4. 分治与回溯：分治法将问题分解为独立子问题，回溯法通过剪枝避免无效搜索。

练习题：

1. 分析以下代码的时间复杂度：

```
public static void printPairs(int[] arr) {  
    for (int i = 0; i < arr.length; i++) {  
        for (int j = i + 1; j < arr.length; j++) {  
            System.out.println(arr[i] + ", " + arr[j]);  
        }  
    }  
}
```

答案：

这段代码的时间复杂度为 $O(n^2)$ （ n 为数组长度）。

- 分析过程：

代码的核心是两层嵌套循环，用于打印数组中所有不重复的元素对（即每个元素对 $(arr[i], arr[j])$ 满足 $i < j$ ）。

1. 外层循环的迭代次数

外层循环的变量 i 从0遍历到 $arr.length - 1$ （共 n 次迭代， $n = arr.length$ ）。

2. 内层循环的迭代次数

对于外层循环的每一次迭代 i ，内层循环的变量 j 从 $i + 1$ 遍历到 $arr.length - 1$ 。因此：

- 当 $i = 0$ 时，内层循环执行 $n - 1$ 次（ $j = 1, 2, \dots, n - 1$ ）；
- 当 $i = 1$ 时，内层循环执行 $n - 2$ 次（ $j = 2, 3, \dots, n - 1$ ）；
- ...
- 当 $i = n - 2$ 时，内层循环执行1次（ $j = n - 1$ ）；
- 当 $i = n - 1$ 时，内层循环不执行（ $j = n$ 超出数组范围）。

3. 总操作次数

总操作次数为内层循环所有迭代次数的和：

$$\text{总次数} = (n - 1) + (n - 2) + \dots + 2 + 1 + 0 = \frac{n(n-1)}{2}$$

4. 时间复杂度结论

根据大 O 表示法的规则（忽略低阶项和常数系数）， $\frac{n(n-1)}{2}$ 可简化为 $O(n^2)$ 。

结论：该代码的时间复杂度为 $O(n^2)$ （平方阶）。

2. 用动态规划实现斐波那契数列，尝试优化空间复杂度。

答案：

动态规划实现斐波那契数列（空间优化版）

斐波那契数列的定义为：

$$F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2) \quad (n \geq 2)$$

传统动态规划方法需要使用数组存储中间结果（空间复杂度 $O(n)$ ），但通过观察可以发现，计算 $F(n)$ 仅需前两项 $F(n-1)$ 和 $F(n-2)$ ，因此可以用两个变量代替数组，将空间复杂度优化至 $O(1)$ 。

实现代码（Java）

```
public class Fibonacci {
    // 动态规划实现，空间复杂度优化为 O(1)
    public static long fibonacci(int n) {
        if (n == 0) return 0;    // 边界条件: F(0) = 0
        if (n == 1) return 1;    // 边界条件: F(1) = 1

        long prev1 = 0; // 保存 F(n-2) 的值 (初始为 F(0))
        long prev2 = 1; // 保存 F(n-1) 的值 (初始为 F(1))

        for (int i = 2; i <= n; i++) {
            long current = prev1 + prev2; // 计算 F(i) = F(i-1) + F(i-2)
            prev1 = prev2;                // 迭代更新 F(n-2) 为原 F(n-1)
            prev2 = current;               // 迭代更新 F(n-1) 为当前 F(i)
        }
        return prev2; // 最终 prev2 即为 F(n)
    }

    public static void main(String[] args) {
        for (int i = 0; i <= 10; i++) {
            System.out.printf("F(%d) = %d\n", i, fibonacci(i));
        }
    }
}
```

代码说明

1. 边界条件处理：

当 $n = 0$ 或 $n = 1$ 时，直接返回0或1，无需计算。

2. 变量初始化：

- `prev1` 初始化为 $F(0) = 0$ （对应 $F(n-2)$ ）；
- `prev2` 初始化为 $F(1) = 1$ （对应 $F(n-1)$ ）。

3. 迭代计算：

从 $i = 2$ 开始循环至 n ，每次计算当前项 $F(i) = \text{prev1} + \text{prev2}$ ，然后更新 `prev1` 和 `prev2` 为下一次迭代的前驱值。

4. 空间优化：

仅用两个变量保存前驱状态，避免了数组存储，空间复杂度从 $O(n)$ 优化至 $O(1)$ 。

复杂度分析

- 时间复杂度： $O(n)$ （仅需一次循环遍历到 n ）。
- 空间复杂度： $O(1)$ （仅用两个变量保存前驱状态）。

测试结果

运行 `main` 方法输出前10项斐波那契数：

```
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
```

Lecture 9 - Sorting

第一部分：排序算法基础

1. 什么是排序算法？

排序算法是将一组数据按特定顺序（如升序、降序）排列的方法。评价标准包括：

- 时间复杂度：算法运行所需时间（如 $O(n^2)$ 、 $O(n \log n)$ ）。
- 空间复杂度：算法占用的额外内存空间（如原地排序 $O(1)$ ）。
- 稳定性：相同元素的相对顺序在排序后是否保持不变。

2. 常见排序算法分类

算法名称	时间复杂度	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(1)$	稳定
归并排序	$O(n \log n)$	$O(n)$	稳定
快速排序	平均 $O(n \log n)$	$O(\log n)$	不稳定
堆排序	$O(n \log n)$	$O(1)$	不稳定

第二部分：冒泡排序（Bubble Sort）

1. 核心思想

通过反复比较相邻元素，将较大的元素逐步“冒泡”到数组末尾。

- 示例：排序 [4, 3, 2, 1]
 - 第1轮：比较 4↔3、4↔2、4↔1，最大元素4到位，数组变为 [3, 2, 1, 4]。
 - 第2轮：比较 3↔2、3↔1，次大元素3到位，数组变为 [2, 1, 3, 4]。
 - 第3轮：比较 2↔1，数组有序 [1, 2, 3, 4]。

2. 代码实现 (Java)

```
public static void bubbleSort(int[] list) {
    boolean needNextPass = true; // 优化：提前终止排序
    for (int k = 1; k < list.length && needNextPass; k++) {
        needNextPass = false;
        for (int i = 0; i < list.length - k; i++) {
            if (list[i] > list[i + 1]) {
                // 交换相邻元素
                int temp = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp;
                needNextPass = true; // 有交换，需继续下一轮
            }
        }
    }
}
```

3. 时间复杂度

- 最好情况（数组已排序）： $O(n)$ （仅需1轮遍历）。
- 最坏情况（完全逆序）： $O(n^2)$ （需 $n(n-1)/2$ 次比较）。

练习1

用冒泡排序对 [5, 1, 4, 2, 8] 排序，手动模拟每一轮过程。

第三部分：归并排序 (Merge Sort)

1. 核心思想（分治算法）

1. 分解：将数组分成两半，递归排序每一半。
2. 合并：将两个已排序的子数组合并成一个有序数组。

- 示例：排序 [5, 2, 9, 1]
 - 分解：[5, 2] 和 [9, 1] → 继续分解为 [5], [2], [9], [1]。
 - 合并：[2, 5] 和 [1, 9] → 最终合并为 [1, 2, 5, 9]。

2. 合并操作（双指针法）

```
private static void merge(int[] left, int[] right, int[] list) {
    int i = 0, j = 0, k = 0;
    while (i < left.length && j < right.length) {
        if (left[i] <= right[j]) {
            list[k++] = left[i++]; // 取较小元素放入结果数组
        } else {
            list[k++] = right[j++];
        }
    }
    // 复制剩余元素
    while (i < left.length) list[k++] = left[i++];
    while (j < right.length) list[k++] = right[j++];
}
```

3. 时间复杂度

- 每一层合并时间为 $O(n)$ ，递归深度为 $\log n$ ，总时间 $O(n \log n)$ 。
- 空间复杂度 $O(n)$ （需临时数组存储合并结果）。

练习2

归并排序的稳定性如何？为什么？

第四部分：快速排序（Quick Sort）

1. 核心思想（分治算法）

1. **选枢轴**：选择一个元素（如第一个元素）作为枢轴。
2. **分区**：将数组分为两部分，左半部分 \leq 枢轴，右半部分 $>$ 枢轴。
3. **递归排序**：对左右两部分递归应用快速排序。

- **示例**：排序 [5, 2, 9, 3, 8]，选5为枢轴
 - 分区后：[2, 3] 5 [9, 8] → 递归排序左右部分，最终得到 [2, 3, 5, 8, 9]。

2. 分区操作（双指针法）

```
private static int partition(int[] list, int first, int last) {
    int pivot = list[first]; // 枢轴
    int low = first + 1, high = last;
    while (high > low) {
        // 找左半部分第一个>枢轴的元素
        while (low <= high && list[low] <= pivot) low++;
        // 找右半部分第一个<=枢轴的元素
        while (low <= high && list[high] > pivot) high--;
    }
}
```

```

        // 交换元素
        if (high > low) swap(list, low, high);
    }
    // 将枢轴放到正确位置
    while (high > first && list[high] >= pivot) high--;
    if (pivot > list[high]) swap(list, first, high);
    return high; // 返回枢轴索引
}

```

3. 时间复杂度

- 平均情况： $O(n \log n)$ （枢轴平衡分区）。
- 最坏情况： $O(n^2)$ （枢轴每次为最小/最大值，如已排序数组）。

练习3

快速排序的空间复杂度为什么是 $O(\log n)$ ？最坏情况如何优化？

第五部分：二叉堆与堆排序（Heap Sort）

1. 二叉堆（最大堆）

- 定义：完全二叉树，每个节点 \geq 子节点（最大堆）。
- 数组表示：根节点下标0，左子节点 $2i+1$ ，右子节点 $2i+2$ ，父节点 $(i-1)/2$ 。
- 操作：
 - 上浮：插入元素后，与父节点比较并交换，直到堆性质满足。
 - 下沉：删除根节点后，与子节点比较并交换，重建堆。

2. 堆排序步骤

1. 建堆：将数组转换为最大堆（自底向上调整）。
2. 排序：重复删除根节点（最大值），放到数组末尾，重建堆。

- 示例：数组 `[10, 5, 3, 4, 1]`
 - 建堆后：`[10, 5, 3, 4, 1]`（根节点最大）。
 - 第1次删除10，数组变为 `[5, 4, 3, 1, 10]`，重建堆后根节点5。
 - 最终排序为 `[1, 3, 4, 5, 10]`。

3. 代码片段（建堆）

```

public static void heapSort(int[] array) {
    int n = array.length;
    // 建堆：从最后一个非叶子节点开始下沉
    for (int i = n/2 - 1; i >= 0; i--) {
        heapify(array, n, i);
    }
}

```

```

// 排序：删除根节点，重建堆
for (int i = n-1; i > 0; i--) {
    swap(array, 0, i); // 根节点（最大值）放到末尾
    heapify(array, i, 0); // 对前i个元素重建堆
}

private static void heapify(int[] array, int n, int i) {
    int largest = i;
    int left = 2*i + 1, right = 2*i + 2;
    if (left < n && array[left] > array[largest]) largest = left;
    if (right < n && array[right] > array[largest]) largest = right;
    if (largest != i) {
        swap(array, i, largest);
        heapify(array, n, largest); // 递归下沉
    }
}

```

4. 时间复杂度

- 建堆时间 $O(n)$ ，排序时间 $O(n \log n)$ ，总时间 $O(n \log n)$ 。

第六部分：总结与对比

算法	核心思想	最佳场景
冒泡排序	相邻交换	小规模数据，教育场景
归并排序	分治+合并	大规模数据，需稳定性
快速排序	分治+分区	通用场景，效率最高
堆排序	二叉堆+删除根	原地排序，内存敏感场景

课后任务

1. 编程练习：

- 实现冒泡排序，测试最好/最坏情况性能。
- 用归并排序对 [3, 1, 4, 1, 5, 9, 2, 6] 排序，输出每一步合并结果。

2. 思考问题：

- 为什么快速排序平均效率高于归并排序？
- 堆排序如何实现原地排序？

Lecture 10 - Graphs and Applications

第一讲：图的基本概念

1.1 什么是图？

- 定义：图是由 **顶点 (Vertices/Nodes)** 和 **边 (Edges)** 组成的结构，记作 $G = (V, E)$ 。
 - 顶点**：表示对象（如城市、用户、节点）。
 - 边**：表示对象之间的关系（如路线、连接、依赖）。
- 示例：
 - 社交网络：顶点是用户，边是“关注”关系（有向边）。
 - 城市地图：顶点是城市，边是道路（无向边，可能带距离权重）。

1.2 图的分类

分类维度	类型	特点
边的方向	无向图	边无方向（如 $A - B$ 与 $B - A$ 是同一条边）
	有向图	边有方向（如 $A \rightarrow B$ 表示从 A 到 B 的单向关系）
边的权重	无权图	边仅表示连接，无数值属性
	有权图	边带有权重（如距离、成本、时间）
特殊结构	简单图	无自环（顶点到自身的边）和并行边（两顶点间多条边）
	非简单图	允许自环或并行边
连通性	连通图	任意两顶点之间存在路径
	非连通图	存在至少两顶点无法通过路径连接

练习1：判断以下场景属于哪种图：

- 地铁线路图（站点连接，无方向）→ **无向无权图**
- 航班路线图（城市间单向航线，带飞行时间）→ **有向有权图**
- 化学分子结构（原子为顶点，化学键为边，无权重）→ **无向简单图**



1.3 关键术语

- 相邻顶点 (Adjacent Vertices)**：通过一条边直接连接的顶点（如 A 和 B 相邻）。
- 顶点的度 (Degree)**：
 - 无向图：顶点关联的边数（如顶点 A 有3条边，度为3）。
 - 有向图：分为 **入度 (In-Degree)** 和 **出度 (Out-Degree)**。
- 环 (Cycle)**：起点和终点相同的路径（如 $A \rightarrow B \rightarrow C \rightarrow A$ ）。
- 树 (Tree)**：连通且无环的无向图，边数为 $|V| - 1$ 。
- 生成树 (Spanning Tree)**：包含图中所有顶点的树（边数为 $|V| - 1$ ，连通无环）。

第二讲：图的表示方法

图的表示方法影响算法的效率，常见方法有两种：邻接矩阵 和 邻接表。



2.1 邻接矩阵 (Adjacency Matrix)

- 定义：用 $n \times n$ 矩阵 (n 为顶点数) 表示顶点间的连接关系。
 - 无权图：矩阵元素 $matrix[i][j] = 1$ 表示存在边，0 表示不存在。
 - 有权图：矩阵元素为边的权重，不存在的边用 ∞ 表示。
- 示例：无向无权图（顶点 $A(0), B(1), C(2)$ ，边 $A - B, A - C, B - C$ ）：
$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$
- 优缺点：
 -  优点：访问速度快（直接查矩阵）。
 -  缺点：存储空间大（稀疏图浪费空间）。

2.2 邻接表 (Adjacency List)

- 定义：每个顶点对应一个列表，存储其相邻顶点或边。
 - 无权图：列表存储相邻顶点索引（如 $0 : [1, 2]$ 表示顶点0相邻顶点1和2）。
 - 有权图：列表存储边对象（包含目标顶点和权重）。
- 实现方式 (Java示例)：

```
// 无权图邻接表
List<List<Integer>> adjList = new ArrayList<>();
adjList.add(Arrays.asList(1, 2)); // 顶点0的相邻顶点
adjList.add(Arrays.asList(0, 2)); // 顶点1的相邻顶点
adjList.add(Arrays.asList(0, 1)); // 顶点2的相邻顶点
```

- 优缺点：
 -  优点：节省空间（仅存储存在的边）。
 -  缺点：访问相邻顶点需遍历列表。

练习2：用邻接表表示有向有权图（顶点 $A(0) \rightarrow B(1)$ 权重5， $B(1) \rightarrow C(2)$ 权重3）：

```
// 有权图邻接表，使用自定义边类
class Edge {
    int to; // 目标顶点
    int weight; // 权重
    public Edge(int to, int weight) { this.to = to; this.weight = weight; }
}

List<List<Edge>> adjList = new ArrayList<>();
adjList.add(Arrays.asList(new Edge(1, 5))); // A(0) → B(1)
adjList.add(Arrays.asList(new Edge(2, 3))); // B(1) → C(2)
```

第三讲：图的遍历算法

遍历图的目的是访问每个顶点一次，常见算法有 **深度优先搜索（DFS）** 和 **广度优先搜索（BFS）**。

3.1 深度优先搜索（DFS）

- **核心思想**：从起点出发，尽可能深入访问相邻顶点，遇无法继续则回溯。
- **实现方式**：递归或栈（此处用递归）。
- **算法步骤**：
 1. 标记当前顶点为已访问。
 2. 递归访问所有未访问的相邻顶点。
- **示例（无向图 0 – 1 – 2 – 3）**：
 - 起点0 → 访问0 → 访问1 → 访问2 → 访问3（假设邻接顺序为0→1→2→3）。
- **代码框架（Java）**：

```
boolean[] visited; // 标记是否访问过
void dfs(int v) {
    visited[v] = true; // 标记当前顶点
    for (int neighbor : adjList.get(v)) { // 遍历相邻顶点
        if (!visited[neighbor]) {
            dfs(neighbor); // 递归访问
        }
    }
}
```

3.2 广度优先搜索（BFS）

- **核心思想**：从起点出发，逐层访问相邻顶点（类似“水波扩散”）。
- **实现方式**：队列（FIFO）。
- **算法步骤**：
 1. 将起点入队，标记为已访问。
 2. 取出队首顶点，访问其所有未访问的相邻顶点，标记并加入队列。
- **示例（无向图 0 – 1 – 2 – 3）**：
 - 起点0 → 队列：[0] → 访问0，相邻顶点1入队 → 队列：[1] → 访问1，相邻顶点2入队 → 队列：[2] → 访问2，相邻顶点3入队 → 访问3。
- **代码框架（Java）**：

```
boolean[] visited;
void bfs(int start) {
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(start);
    visited[start] = true;
```



```

while (!queue.isEmpty()) {
    int v = queue.poll(); // 取出队首顶点
    for (int neighbor : adjList.get(v)) {
        if (!visited[neighbor]) {
            visited[neighbor] = true;
            queue.offer(neighbor); // 相邻顶点入队
        }
    }
}
}

```

练习3：画出以下图的DFS和BFS遍历顺序（起点0）：

邻接表：0: [1, 3], 1: [0, 2], 2: [1], 3: [0, 4], 4: [3]

- **DFS顺序：** $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$
- **BFS顺序：** $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4$

第四讲：有权图与经典算法

4.1 最小生成树（MST）

- **定义：**连通有权图中，边权和最小的生成树。
- **应用场景：**构建最低成本的网络（如城市间管道铺设）。
- **算法：**
 - **Prim算法：**从顶点出发，每次选当前生成树与未加入顶点的最小边（适用于稠密图）。
 - **Kruskal算法：**从边出发，按权重排序后选最小边，用并查集避免成环（适用于稀疏图）。

4.2 最短路径算法（Dijkstra）

- **定义：**求单源顶点到其他所有顶点的最短路径（边权非负）。
- **应用场景：**导航系统（如最短驾车路线）。
- **核心思想：**维护每个顶点到源点的当前最短距离，逐步更新（贪心策略）。
- **示例：**源点0到顶点3的最短路径：

图：0→1（权重2），0→2（权重5），1→3（权重3），2→3（权重1）
 最短路径：0→2→3（总权重5+1=6）

Lecture 11 - Binary Search Trees

一、二叉树基础

1. 二叉树定义

- **结构**：要么为空，要么由根节点、左子树、右子树组成。
- **节点**：每个节点包含一个元素（`element`），以及左右子节点指针（`left / right`）。
- **术语**：
 - 叶子节点：没有子节点的节点（如示例中的45、57、67、107）。
 - 子树：节点的左/右分支形成的树。

2. 二叉树的表示（Java代码）

```
class TreeNode<E> {  
    E element;  
    TreeNode<E> left;  
    TreeNode<E> right;  
    public TreeNode(E o) { element = o; }  
}
```

- 每个节点是一个泛型类，可存储任意类型数据。

二、二叉搜索树（BST）核心性质

1. BST的特性

- **无重复元素**（默认）。
- **有序性**：
 - 左子树所有节点的值 **小于** 根节点的值。
 - 右子树所有节点的值 **大于** 根节点的值。
- **示例**：

```
      60  
     /  \  
    55   100  
   /  \  /  \  
  45  57 67 107
```

- $55 < 60 < 100$, $45 < 55 < 57$, $67 < 100 < 107$ ，符合BST规则。

2. BST的优势

- **高效搜索、插入、删除**（理想情况下时间复杂度为 $O(\log n)$ ）。

- 中序遍历有序：遍历结果为升序序列（如上述示例中序遍历结果：45, 55, 57, 60, 67, 100, 107）。

三、BST核心操作：插入元素

1. 插入逻辑

1. 树为空：直接创建根节点。
2. 树非空：
 - 用 `current` 和 `parent` 指针找到插入位置：
 - 若插入值 $<$ 当前节点值，移动到左子树；
 - 若插入值 $>$ 当前节点值，移动到右子树；
 - 若相等（重复），插入失败。
 - 找到 `parent` 后，根据值大小决定插入左或右子节点。

2. 代码实现（Java）

```
public boolean insert(E element) {
    if (root == null) {
        root = new TreeNode<>(element); // 空树时创建根节点
    } else {
        TreeNode<E> current = root, parent = null;
        while (current != null) {
            parent = current;
            if (element.compareTo(current.element) < 0) {
                current = current.left;
            } else if (element.compareTo(current.element) > 0) {
                current = current.right;
            } else {
                return false; // 重复元素，插入失败
            }
        }
        // 插入到parent的左或右子节点
        if (element.compareTo(parent.element) < 0) {
            parent.left = new TreeNode<>(element);
        } else {
            parent.right = new TreeNode<>(element);
        }
    }
    return true;
}
```

3. 示例：插入101到BST

- 步骤跟踪：
 1. 从根节点60开始， $101 > 60 \rightarrow$ 移动到右子节点100。

- 2. $101 > 100 \rightarrow$ 移动到右子节点107。
- 3. $101 < 107 \rightarrow$ `parent=107` , `current=null` , 插入到107的左子节点。
- 结果: 107的左子节点为101, 树结构保持BST性质。

四、BST核心操作：搜索元素

1. 搜索逻辑

- 从根节点开始, 逐层比较:
 - 若搜索值 $<$ 当前节点值, 搜索左子树;
 - 若搜索值 $>$ 当前节点值, 搜索右子树;
 - 相等则返回 `true` , 遍历完仍未找到则返回 `false` 。

2. 代码实现 (Java)

```
public boolean search(E element) {
    TreeNode<E> current = root;
    while (current != null) {
        if (element.compareTo(current.element) < 0) {
            current = current.left;
        } else if (element.compareTo(current.element) > 0) {
            current = current.right;
        } else {
            return true; // 找到元素
        }
    }
    return false; // 未找到元素
}
```

3. 示例：搜索57

- $60 \rightarrow 55$ (左子节点) $\rightarrow 57$ (右子节点) , 找到, 返回 `true` 。

五、树的遍历

1. 深度优先遍历 (递归实现)

- 前序遍历 (根 \rightarrow 左 \rightarrow 右) : 访问根节点, 递归遍历左子树, 递归遍历右子树。
- 中序遍历 (左 \rightarrow 根 \rightarrow 右) : BST中序遍历结果为升序序列 (重点!) 。
- 后序遍历 (左 \rightarrow 右 \rightarrow 根) : 常用于删除树前释放资源。

2. 代码示例 (中序遍历)

```
public void inorder() {
    inorder(root); // 从根节点开始
}
```

```

}

private void inorder(TreeNode<E> root) {
    if (root == null) return;
    inorder(root.left);    // 左子树
    System.out.print(root.element + " "); // 根节点
    inorder(root.right);   // 右子树
}

```

3. 广度优先遍历（层序遍历）

- 按层访问：根节点→第一层子节点→第二层子节点→...
- 实现**：使用队列，先入队根节点，每次出队一个节点，入队其左右子节点（若存在）。

好的！我们继续学习剩余内容，包括 **BST删除操作**、**迭代器实现**、**Huffman编码** 和 **时间复杂度分析**。

六、BST核心操作：删除元素

删除操作是BST中最复杂的操作，需分情况处理，确保删除后仍满足BST性质。

1. 准备工作：定位节点

- 用 `current` 指向待删除节点，`parent` 指向其父节点。
- 若 `current` 为 `null`，说明元素不存在，删除失败。

2. 情况1：当前节点无左子树

- 操作**：直接用当前节点的右子树替代当前节点。
- 示例**：删除节点10（父节点20）：

原树结构：	删除后结构：
20	20
/ \	/ \
10 40	40
\ / \	
16 30 80	

- 代码逻辑**：

```

if (current.left == null) {
    if (parent == null) { // 删除根节点
        root = current.right;
    } else if (current == parent.left) { // 当前节点是左子节点
        parent.left = current.right;
    } else { // 当前节点是右子节点

```

```

        parent.right = current.right;
    }
}

```

3. 情况2：当前节点有左子树

- **核心思想**：找到当前节点左子树中的 **最右节点 (rightMost)**，用其值替换当前节点的值，然后删除该最右节点（因其无右子树，转化为情况1）。
- **原因**：rightMost是左子树中最大的值，替换后不会破坏BST的有序性。
- **步骤**：
 1. 找到 current 左子树的最右节点 rightMost 及其父节点 parentOfRightMost。
 2. 用 rightMost 的值覆盖 current 的值。
 3. 删除 rightMost（若 rightMost 有左子树，连接到 parentOfRightMost 的右侧）。
- **示例**：删除节点20（左子树最右节点为16）：

原树结构：	替换后结构：
20	16
/ \	/ \
10 40	10 40
\ / \	
16 30 80	

- **代码逻辑**：

```

else { // 当前节点有左子树
    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;
    // 找到左子树的最右节点
    while (rightMost.right != null) {
        parentOfRightMost = rightMost;
        rightMost = rightMost.right;
    }
    // 用rightMost的值替换current的值
    current.element = rightMost.element;
    // 删除rightMost节点（其无右子树，可能有左子树）
    if (parentOfRightMost == current) { // rightMost是current的左子节点
        parentOfRightMost.left = rightMost.left;
    } else {
        parentOfRightMost.right = rightMost.left;
    }
}
}

```

七、使用迭代器遍历树

1. 为什么需要迭代器？

- 传统的 `inorder()`、`preorder()` 方法只能打印元素，而迭代器允许通过 `foreach` 循环灵活处理元素（如转换为大写、过滤等）。

2. 实现步骤（以中序遍历为例）

1. 定义迭代器类：实现 `java.util.Iterator` 接口。
2. 中序遍历存储元素：在迭代器构造函数中执行中序遍历，将元素存入列表 `list`。
3. 实现迭代器方法：
 - `hasNext()`：检查列表是否有剩余元素。
 - `next()`：返回当前元素并移动指针。
 - `remove()`：删除当前元素（需重新构建列表）。

3. 代码示例

```
public Iterator<E> iterator() {
    return new InorderIterator();
}

private class InorderIterator implements Iterator<E> {
    private ArrayList<E> list = new ArrayList<>();
    private int current = 0;

    public InorderIterator() {
        inorder(root); // 中序遍历填充list
    }

    private void inorder(TreeNode<E> root) {
        if (root == null) return;
        inorder(root.left);
        list.add(root.element);
        inorder(root.right);
    }

    @Override
    public boolean hasNext() {
        return current < list.size();
    }

    @Override
    public E next() {
        return list.get(current++);
    }

    @Override
    public void remove() {
```

```

        BST.this.delete(list.get(current)); // 删除元素
        list.clear(); // 清空列表并重新遍历
        inorder(root);
        current = 0;
    }
}

```

4. 使用示例

```

BST<String> tree = new BST<>();
tree.insert("A");
tree.insert("B");
tree.insert("C");
for (String s : tree) { // 自动调用迭代器
    System.out.print(s.toUpperCase() + " "); // 输出: A B C
}

```

八、Huffman编码：二叉树在数据压缩中的应用

1. 核心思想

- 用 频率高的字符分配短编码，频率低的分配长编码，减少整体编码长度。
- 通过构建Huffman树（带权路径长度最短的二叉树）实现。

2. 构建Huffman树的步骤（贪心算法）

1. 统计字符频率：例如，字符串"Mississippi"中，'i'和's'频率为4，'M'为1，'p'为2。
2. 创建初始森林：每个字符作为一棵单节点树，权重为频率。
3. 合并最小权重树：
 - 每次选取权重最小的两棵树，创建父节点，权重为两者之和。
 - 重复直到只剩一棵树（Huffman树）。

3. 生成字符编码

- 从根节点到叶子节点的路径中，左分支记为 0，右分支记为 1。
- 示例：

```

      12 (根)
     /  \
    5    7
   / \  / \
  M p s i (频率: 1, 2, 4, 4)

```

编码：M=000, p=001, s=01, i=1

4. 代码实现（简化版）

```
// 构建Huffman树
public static HuffmanTree buildHuffmanTree(int[] frequencies) {
    PriorityQueue<HuffmanTree> queue = new PriorityQueue<>();
    // 初始化单节点树
    for (int i = 0; i < 256; i++) {
        if (frequencies[i] > 0) {
            queue.offer(new HuffmanTree(frequencies[i], (char) i));
        }
    }
    // 合并树
    while (queue.size() > 1) {
        HuffmanTree t1 = queue.poll();
        HuffmanTree t2 = queue.poll();
        HuffmanTree newTree = new HuffmanTree(t1, t2);
        queue.offer(newTree);
    }
    return queue.poll();
}

// 生成编码
public static String[] getHuffmanCodes(HuffmanTree root) {
    String[] codes = new String[256];
    generateCodes(root.root, "", codes);
    return codes;
}

private static void generateCodes(HuffmanNode node, String code, String[] codes) {
    if (node.left == null && node.right == null) { // 叶子节点
        codes[(int) node.element] = code;
        return;
    }
    generateCodes(node.left, code + "0", codes); // 左分支加0
    generateCodes(node.right, code + "1", codes); // 右分支加1
}
```

九、时间复杂度分析

操作	时间复杂度	说明
插入/搜索/删除	$O(h)$ (h 为树高)	最坏情况树退化为链表, $h = n$, $O(n)$; 理想情况平衡树, $h = \log n$, $O(\log n)$
遍历（前/中/后序）	$O(n)$	每个节点访问一次

十、课程总结

1. 核心知识点

- **BST性质**：左<根<右，中序遍历有序。
- **三大操作**：
 - 插入：找到合适位置，保持BST性质。
 - 搜索：逐层比较，高效查找。
 - 删除：分情况处理，维护BST结构。
- **遍历方式**：前序/中序/后序（递归）、层序（队列）。
- **迭代器**：通过中序遍历实现，支持灵活遍历。
- **Huffman编码**：利用二叉树压缩数据，贪心算法构建最优树。

2. 代码结构建议

- **TreeNode**：作为内部类，封装节点属性。
- **BST类**：实现插入、删除、搜索、遍历、迭代器等方法。
- **Huffman相关类**：独立实现树构建和编码生成。

3. 练习建议

1. **手动绘制**：删除BST中的节点，如删除示例树中的60，观察结构变化。
2. **编码实践**：实现Huffman编码的完整流程（编码+解码）。
3. **扩展思考**：如何优化BST的性能？（提示：平衡树如AVL、红黑树）

课后问题

1. 删除BST节点时，为什么选择左子树的最右节点替换，而不是右子树的最左节点？
2. Huffman树中，为什么合并最小权重的两棵树能得到最优编码？
3. 尝试用迭代方式实现前序遍历（非递归）。

Lecture 12 - AVL Trees

一、AVL树核心知识

1. 为什么需要AVL树？

- **问题**：普通二叉搜索树（BST）在最坏情况下高度为 $O(n)$ ，导致搜索、插入、删除的时间复杂度退化为 $O(n)$ 。
- **目标**：通过保持树的“平衡”，使高度近似 $O(\log n)$ ，从而保证操作的高效性。
- **平衡定义**：每个节点的左右子树高度差（平衡因子）绝对值不超过1。平衡因子 = 右子树高度 - 左子树高度。

2. AVL树的平衡操作（四种旋转）

当插入或删除节点导致某节点平衡因子为 ± 2 时，需要通过旋转重新平衡。以下是四种旋转的核心逻辑：

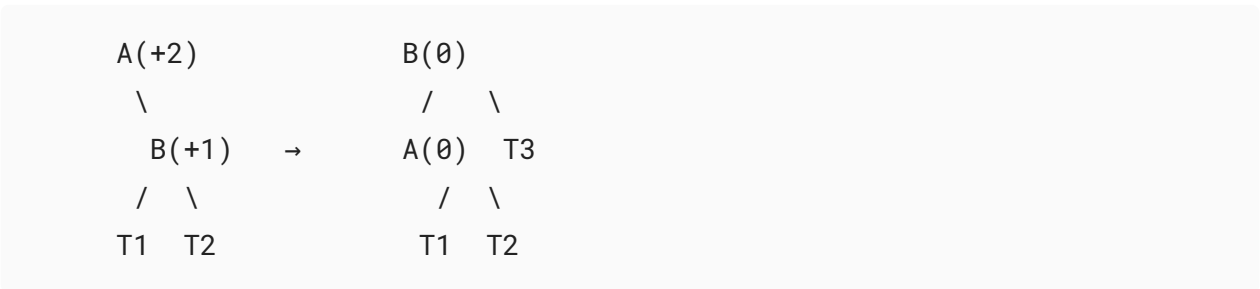
LL旋转（左左失衡，右单旋）

- **场景：**节点A左子树高度比右子树高2，且左子节点B是左重（平衡因子 ≤ 0 ）。
- **操作：**将B提升为新根，A变为B的右子节点，B原来的右子树T3变为A的左子树。
- **示例：**



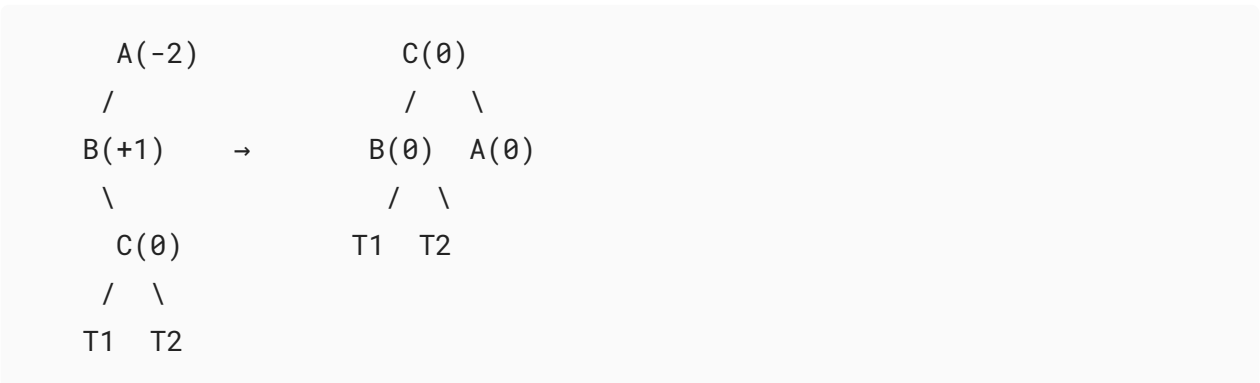
RR旋转（右右失衡，左单旋）

- **场景：**节点A右子树高度比左子树高2，且右子节点B是右重（平衡因子 ≥ 0 ）。
- **操作：**将B提升为新根，A变为B的左子节点，B原来的左子树T2变为A的右子树。
- **示例：**



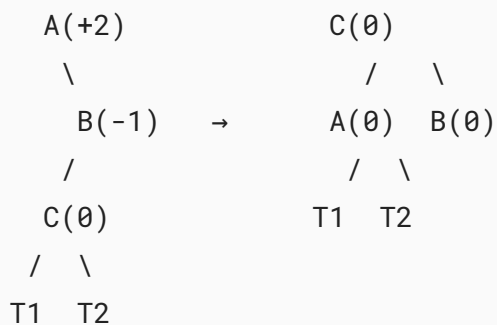
LR旋转（左右失衡，先左旋B再右旋A）

- **场景：**节点A左子树高度比右子树高2，且左子节点B是右重（平衡因子 $+1$ ）。
- **操作：**先对B进行左旋转，再对A进行右旋转，中间节点C成为新根。
- **示例：**



RL旋转（右左失衡，先右旋B再左旋A）

- 场景：节点A右子树高度比左子树高2，且右子节点B是左重（平衡因子-1）。
- 操作：先对B进行右旋转，再对A进行左旋转，中间节点C成为新根。
- 示例：



3. AVL树的实现关键点

- 节点设计：每个节点包含高度属性，用于计算平衡因子。
- 插入/删除流程1
 1. 按BST规则插入/删除节点。
 2. 从插入/删除节点向上遍历，更新路径上所有节点的高度。
 3. 检查每个节点的平衡因子，触发对应的旋转操作。
- 代码示例（插入后的平衡处理）：

```
private void balancePath(E e) {
    ArrayList<TreeNode<E>> path = path(e); // 获取插入路径
    for (int i = path.size() - 1; i >= 0; i--) {
        AVLTreeNode<E> A = (AVLTreeNode<E>) path.get(i);
        updateHeight(A); // 更新高度
        int bf = balanceFactor(A);
        if (bf == -2) { // 左重
            if (balanceFactor((AVLTreeNode<E>) A.left) <= 0) {
                balanceLL(A, parent); // LL旋转
            } else {
                balanceLR(A, parent); // LR旋转
            }
        } else if (bf == +2) { // 右重
            if (balanceFactor((AVLTreeNode<E>) A.right) >= 0) {
                balanceRR(A, parent); // RR旋转
            } else {
                balanceRL(A, parent); // RL旋转
            }
        }
    }
}
```

```
}  
}
```

4. 时间复杂度分析

- 树的高度为 $O(\log n)$ ，平衡操作（旋转）是常数时间，因此插入、删除、搜索的时间复杂度均为 $O(\log n)$ 。

二、哈希表核心知识

1. 什么是哈希表？

- 目标：通过哈希函数将键映射到数组索引，实现 $O(1)$ 平均时间复杂度的查找、插入、删除。
- 核心组件：
 - 哈希函数：将键转换为哈希码（整数），再压缩为数组索引（如 `hashCode % N`， N 为表大小）。
 - 碰撞处理：当不同键映射到同一索引时，通过开放寻址法或链地址法处理。

2. 哈希函数与哈希码

- 哈希码计算：
 - 基本类型：直接转换（如`int`类型直接作为哈希码）。
 - 长类型/双精度：通过高32位和低32位异或（XOR）避免信息丢失。
 - 字符串：通过多项式哈希（如 `hash = 31 * hash + charValue`）。
- 压缩方法：使用取模运算 `hashCode % N`， N 通常为质数以减少碰撞。

3. 碰撞处理方法

开放寻址法：在数组中寻找下一个可用位置。

- 线性探测：依次检查下一个位置（`(index + 1) % N`，`(index + 2) % N ...`），易形成“聚类”。
- 二次探测：检查 `(index + j2) % N` ($j=1,2,3...$)，减少聚类但可能导致“二次聚类”。
- 双重哈希：使用第二个哈希函数确定步长（如 `h'(key) = 7 - key % 7`），步长需与 N 互质。

链地址法（分离链接）：每个索引对应一个链表（或链表+红黑树，如Java 8后的HashMap）。

- 优点：处理碰撞简单，无负载因子上限。
- 缺点：链表过长时查找效率下降，需结合负载因子扩容。

4. 负载因子与重新哈希

- 负载因子： $\lambda = \text{元素数} / \text{表大小}$ ，开放寻址法通常需 $\lambda < 0.5$ ，链地址法 $\lambda < 0.9$ 。
- 重新哈希：当 λ 超过阈值时，增大表大小（如翻倍），重新计算所有键的索引，避免性能下降。

5. MyHashMap实现关键点

- 哈希函数优化：通过位移和异或（如Java的 supplementalHash ）让哈希码更均匀分布。
- 链地址法实现：使用数组存储链表，插入时检查键是否存在，存在则更新值，否则添加到链表尾部。
- 代码示例（哈希函数）：

```
private int hash(int hashCode) {  
    hashCode = (hashCode >>> 20) ^ (hashCode >>> 12); // 混合高位  
    return hashCode ^ (hashCode >>> 7) ^ (hashCode >>> 4); // 进一步混合  
}
```

三、学习建议

1. **AVL树部分：**
 - 手动绘制插入/删除示例（如文档中的插入25,20,5,34...的过程），观察旋转如何恢复平衡。
 - 编写旋转函数时，先理清节点指针的变化（父节点、左右子树的连接），再处理高度更新。
2. **哈希表部分：**
 - 对比不同碰撞处理方法的优缺点，思考为何Java的HashMap在链长较长时转为红黑树。
 - 实现MyHashMap时，注意负载因子的阈值设置和重新哈希的时机，避免频繁扩容。
3. **实践步骤：**
 - 先完成AVL树的节点定义、插入/删除的BST部分，再逐步添加平衡逻辑。
 - 哈希表从简单的链地址法开始，实现put、get、containsKey方法，再考虑扩容和性能优化。

四、课后小问题（检验理解）

1. AVL树中，LL旋转和LR旋转的区别是什么？什么场景下触发？
2. 哈希表中，为什么负载因子过高会影响性能？重新哈希的代价是什么？
3. 链地址法和开放寻址法在删除操作上有何不同？（提示：开放寻址法删除需标记“墓碑”）