1.import java.util.Scanner;

public class Test {

public static void main(String[] args) {

double[] myList;

int n = 100;

myList = new double[n];

Scanner input = new Scanner(System.in);

System.out.print("Enter "+myList.length+" doubles: ");

double sum = 0;

for (int i = 0; i < myList.length; i++) {

myList[i] = input.nextDouble();

sum += myList[i];

}

double avg = sum / myList.length;

int counter = 0;

for (double v : myList)

if (v > avg)

counter++;

System.out.println(counter

+" are greater than the average "+avg);

}

}

2.1 声明（Declaration）

首先需要声明数组变量，声明的语法主要有两种。

dataType[] arrayRefVar;// 首选的方法

dataType arrayRefVar[];// 效果相同，但不是首选方法

2.2 创建（Creation）

数组变量的创建使用new操作符来进行。

arrayRefVar = new dataType[arraySize];

这里使用new操作符创建了数组后又将新创建的数组的引用赋值给变量arrayRefVar。这里注意是引用，即它的内存地址，后面我们会详细叙述这一点。

数组变量的声明和创建可以用一条语句完成，如下。

dataType[] arrayRefVar = new dataType[arraySize];

所以我们的示例如下。

int[] array1;

array1 = new int[10];

int[] array2 = new int[10]

我们也可以用下面这种方式创建数组。

dataType[] arrayRefVar = {value0, value1, ..., valuek};

示例如下。

double[] myList = {1.9, 2.9, 3.4, 3.5};

它等价于下列代码。

double[] myList = new double[4];

myList[0] = 1.9;

myList[1] = 2.9;

myList[2] = 3.4;

myList[3] = 3.5;

当然下列代码也是和上面等价的。

double[] myList = new double[]{1.9, 2.9, 3.4, 3.5};

2.4.1 移动数组里的值

如果我们想将数组里的值进行移动。

向左移动，第一个元素移动至最后一个的代码如下。

int temp = myList[0];

for (int i = 1; i < myList.length; i++) {

myList[i - 1] = myList[i];

}

myList[myList.length - 1] = temp;

2.4.2 加强for循环（Enhanced for Loop）

JDK 1.5 引进了一种新的循环类型，被称为For-Each循环或者加强型循环，它能在不使用下标的情况下遍历数组。

语法格式如下。

for(type element: array){

}

例如下面的代码可以输出数组里的所有元素。

for(double value: myList){

System.out.println(value);

}

它与下面的for循环代码效果是一样的。

for (int i = 0; i < myList.length; i++) {

System.out.println(myList[i]);

}

加强for循环主要用于顺序读取数组或集合中的元素，而不适用于需要索引来访问元素的场景，因此也无法修改数组或集合中的元素。就比如前面的将数组里的元素移动的操作无法进行。再比如下面的代码。

int[] myList = new int[]{1, 2, 3, 4, 5};

for (int number : myList) {

number = number \* 2;

}

for (int number : myList) {

System.out.println(number);

}

//输出的结果还是1 2 3 4 5

我们可以理解成for-each循环中的number时原来数组的副本，因此对number的修改不会影响原数组myList中的元素。

2.4.3 多维数组

多维数组可以看成是数组的数组，比如二维数组就是一个特殊的一维数组，其每一个元素都是一个一维数组，格式如下。

type[][] typeName = new type[typeLength1][typeLength2];

type 可以为基本数据类型和复合数据类型（类和对象，将在后面介绍），typeLength1 和 typeLength2 必须为正整数，typeLength1 为行数，typeLength2 为列数。

例如下面的代码。

int[][] a = new int[2][3];

这个代码中创建了一个二维数组a，可以看成是一个两行三列的数组。

对二维数组中的每个元素，引用方式如下。

arrayName[index1][index2]

我们可以对二维数组的每一维分开分配不同长度的空间，如下所示。

String[][] s = new String[2][];

s[0] = new String[2];

s[1] = new String[3];

s[0][0] = new String("Good");

s[0][1] = new String("Luck");

s[1][0] = new String("to");

s[1][1] = new String("you");

s[1][2] = new String("!");

这串代码中，我们先创建了一个包含两个元素的二维数组s，其中每个元素是本身也是一个String数组。s[0]是第一行，它是一个长度为2的String数组。s[1]是第一行，它是一个长度为3的String数组。后面再分别给每个元素赋值。其中我们就用前面说的引用方式对元素进行了访问。

数组是引用类型（Reference Types）变量，而不是基础数据类型（Primitive Types）变量。即变量存储的是对象的引用（内存地址），而不是对象本身。

两者的区别可以看下面两段代码。

int a = 10;

int b = a;

a = 20;

System.out.println(b); // 这里print的结果是10，而不是20

int[] array1 = {1, 2, 3};

int[] array2 = array1;

array1[0] = 10;

System.out.println(array2[0]);// 这里print结果是10，因为array2指向了array1的地址

因此如果array2原来有值，在array2 = array1;后array2原理的值就会被丢弃，要想复制一个数组可以使用下面的几种方法。

方法一：使用循环。

int[] sourceArray={2, 3, 1, 5, 10};

int[] targetArray=new int[sourceArray.length];

for (int i = 0; i < sourceArray.length; i++){

targetArray[i] = sourceArray[i];

}

方法二：使用arraycopy方法

System.arraycopy(sourceArray, src\_pos, targetArray, tar\_pos, length);

其中sourceArray是被复制的数组。

src\_pos是被复制的数组的起始位置，表示从该位置开始复制数据。

targetArray是目标数组，表示要被复制到这个数组中。

tar\_pos是目标数组中的起始位置，表示从该位置开始粘贴数据。

length表示要复制的元素数量。

因此实践起来如下。

int[] sourceArray={2, 3, 1, 5, 10};

int[] targetArray=new int[sourceArray.length];

System.arraycopy(sourceArray, 0, targetArray, 0, sourceArray.length);

2.6.2 方法返回数组

前面我们使用void关键字，即该方法执行后不返回任何结果，现在我们尝试返回一个数组。

下面的示例展示了如何将一个数组翻转。

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];

}

return result;

}

这里for循环中使用了i和j两个参数，当然你也可以使用一个参数去解决。

public static int[] reverse(int[] list) {

int[] result = new int[list.length];

for (int i = 0; i < list.length; i++) {

result[result.length - 1 - i] = list[i];

}

return result;

}

当然这样做返回了一个新的数组，这么做不会对原来的数组进行修改。如果需要我们对原数组进行翻转，我们可以利用前面说的知识，示例如下。

public static void reverse(int[] list) {

int temp;

for (int i = 0, j = list.length- 1; i < list.length/2; i++, j--) {

temp = list[j];

list[j] = list[i];

list[i] = temp;

}

}

同理一个参数的示例如下。

public static void reverse(int[] list) {

int temp;

for (int i = 0; i < list.length/2; i++) {

temp = list[list.length - 1 - i];

list[list.length - 1 - i] = list[i];

list[i] = temp;

}

}

2.6.3 在数组中查询某元素

现在我们需要查找数组里的某个元素，返回第一次出现的这个元素的index值，如果没有这个元素，我们便返回-1。

一个很简单的思路便是我们一个个去对照，如果找到便返回这里的index值，如果没有找到便返回-1。我们叫这种方式为线性搜索（Linear Search）。

public static int linearSearch(int[] list, int key) {

for (int i = 0; i < list.length; i++){

if (key == list[i]){

return i;

}

}

return -1;

}

二分搜索（Binary Search），二分搜索的前提是这个数组里的元素是提前排序好的，下面假设这个数组是从低到高排序的。先将目标元素与中间的元素进行对比，如果相等，那就返回该元素的index，如果目标元素小于中间元素，则去前半部分中继续寻找，否则就去后半部分中继续寻找。然后再不断重复这个过程，如果搜寻完还是没有找到，那就返回-1。

代码如下。

public static int binarySearch(int[] list, int key) {

int low = 0;

int high = list.length- 1;

int mid;

while (high >= low) {

mid = (low + high) / 2;

if (key < list[mid]){

high = mid - 1;

}

else if (key == list[mid]){

return mid;

}

else{

low = mid + 1;

}

}

return -1;

}

它的时间复杂度是O ( l o g n )

##### **选择排序（Selection Sort）**

这个方法是将剩余元素中的最小（或最大）元素，存放到序列的起始位置，直到全部元素被排序。

public static void selectionSort(double[] list) {

for (int i = 0; i < list.length; i++) {

// Find the minimum in the list[i..list.length-1]

double currentMin = list[i];

int currentMinIndex = i;

for (int j = i + 1; j < list.length; j++) {

if (currentMin > list[j]) {

currentMin = list[j];

currentMinIndex = j;

}

}

// Swap list[i] with list[currentMinIndex] if necessary;

if (currentMinIndex != i) {

list[currentMinIndex] = list[i];

list[i] = currentMin;

}

}

}

##### **插入排序（Insertion Sort）**

这个方法是不断构件有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

public static void insertionSort(int[] a){

for(int i=1; i<a.length; i++){

int temp = a[i];

if(temp < a[i-1]) {

int j;

for(j = i-1; j >= 0; j--){

if(temp < a[j]){

a[j+1] = a[j];

}

else{

break;

}

a[j+1] = temp;

}

}

}

}

同理Java也有内置的插入排序方法再java.util.Arrays类中。示例如下。

double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};

java.util.Arrays.sort(numbers);

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};

java.util.Arrays.sort(chars);

Classes（类）

类（Class）是 Java 中实现面向对象编程（Object-Oriented Programming，OOP）的核心概念之一，它是Java中面向对象编程最好的体现，它为创建对象提供了模板或蓝图。

在面向对象编程中，我们使用对象表示现实世界中的一个实体，它可以从具有共同属性（properties）的类/对象模板中被明确识别出来。在Java中，我们使用类来定义相同类型对象。类描述了一组属性（数据字段）和行为（方法），用于创建对象。

一个对象有自己的状态（state）和行为（behavior）。

1.状态是由一些数据字段（或者说属性）机器当前值组成。

2.行为是由一组实例方法定义。

3.1 类的组成

因此Java的类由三部分组成。

1.非静态/实例变量（Instance Variables）：用于定义对象的数据字段。这些变量是每个对象独有的，存储对象的状态。

2.非静态/实例方法（Instance Methods）：用于定义对象的行为。实例方法是对象可以执行的操作，这些方法只能通过类的实例（对象）来调用，而不能通过类本身直接调用。

3.构造方法（Constructors）：一种特殊类型的方法，用于创建类的实例（对象）。构造方法在创建对象时被调用，用于初始化对象的状态。

3.3 静态（Static）与非静态（non-static）变量

静态变量（Static variables）是类级别的变量，属于类本身，而不是类的某个特定实例（对象），因此它在整个类的所有实例之间共享，对静态变量的修改也会影响到所有实例。

例如对于前面的例子中，我们可以添加一个静态变量用于记录这里目前有多少人。

public class Human {

// 静态变量

static int count = 0;

// 实例变量（数据字段）

int age;

// 构造方法

public Human() {

this.age = 0;

count++; // 每创建一个实例，计数增加1

}

public Human(int age) {

this.age = age;

count++; // 每创建一个实例，计数增加1

}

// 实例方法

public void addAge(int years) {

age += years;

System.out.println("Age updated to: " + age);

}

// 静态方法用于获取当前人类数量

public static int getCount() {

return count;

}

}

public class Main {

public static void main(String[] args) {

Human Peter = new Human(30); // 创建Human类的实例Peter

Peter.addAge(5); // 正确调用：通过实例（对象）调用实例方法

// Human.addAge(5); // 错误调用：尝试通过类直接调用实例方法

System.out.println("Number of humans: " + Human.getCount()); // 输出当前人类数量，结果为1

Human Alice = new Human();

Alice.addAge(5);

System.out.println("Number of humans: " + Human.getCount()); // 输出当前人类数量，结果为2

}

}

我们也可以在一个圆的类中，用静态变量以定义圆周率以供圆里的每个对象使用，而且像这种我们不会修改的常量，可以再使用final关键字以防止被修改。如下所示。

static final double PI = 3.1415926;

上面的示例中，我们还创建了一个名为getCount的静态方法。静态变量（也称为类变量）和静态方法属于类本身，而不是类的某个特定实例（对象）。因此访问静态变量或调用静态方法时，使用类名，而不是实例名。

System.out.println("Number of humans: " + Human.getCount());

因此这里是Human.getCount()。

✅ 一、什么是静态（static）的概念？

静态变量（static variable）：属于类（class），而不是属于某个具体的对象（instance）。

静态方法（static method）：只能访问类的静态成员（变量/方法），不依赖于对象。

换句话说，不需要创建对象就可以通过类名访问静态内容。

✅ 二、静态变量有什么用？

🧮 1. 用于所有实例共享的数据

静态变量不会因创建新对象而重新生成，而是在类加载时只初始化一次，并且所有对象共享这一个值。

📌 示例：统计有多少个 Human 被创建

java

复制

编辑

static int count = 0;

每次创建一个 Human 实例，构造函数里执行 count++。

所有对象共享这个变量 → count 是全局计数器。

运行结果：

java

复制

编辑

Human Peter = new Human(30);

Human Alice = new Human();

System.out.println(Human.getCount()); // 输出 2

无论你有多少个对象，count 始终是它们共享的变量。

✅ 三、静态方法的作用

💡 静态方法是不依赖实例的函数，通常用于：

访问或操作静态变量

提供工具方法或通用功能（比如 Math.sqrt()）

java

复制

编辑

public static int getCount() {

return count;

}

🔎 访问方式：

java

复制

编辑

Human.getCount(); // ✅ 正确

Peter.getCount(); // 虽然编译不报错，但不推荐这么用

✅ 四、静态常量（static final）

🎯 例如圆周率：

java

复制

编辑

static final double PI = 3.1415926;

static → 所有对象共享，不用每次都存一份

final → 常量，不能修改

✅ 五、静态的使用总结

特性 静态变量 静态方法

属于谁 类本身 类本身

是否共享 所有对象共享 所有对象共享

访问方式 类名.变量 类名.方法

是否需要实例化对象 ❌ 不需要 ❌ 不需要

能否访问非静态成员 ❌ 不能 ❌ 不能

✅ 六、静态有什么用（总结一句话）

静态关键字static让变量或方法属于类，而不是对象本身，从而在多个对象之间共享数据或行为。

3.4 默认值与值传递问题

前面我们说过Java会给数据字段分配默认值。

对于基础数据类型（Primitive Data Types）：

byte：默认值为 0

short：默认值为 0

int：默认值为 0

long：默认值为 0L

float：默认值为 0.0f

double：默认值为 0.0d

char：默认值为 \u0000（即空字符）

boolean：默认值为 false

其余的默认值为null

注意如果一个类的实例变量是另一个类的实例，那么这个变量会继承或使用那个类中定义的默认值。

需要注意的是Java 不会给方法内部的局部变量分配默认值。

比如下面的代码会提示你尚未给变量进行初始化。

public class Test {

public static void main(String[] args) {

int x; // x has no default value

String y; // y has no default value

System.out.println("x is " + x);

System.out.println("y is " + y);

}

}

由于类也属于一种引用类型。因此我们要是想复制一个对象和前面的数组一样（它们都是引用类型，或者详细点说数组是java.lang.Object类的子类，即java.lang.reflect.Array类的实例），不能直接使用c1 = c2这种方式。

3.6 对象数组

对象数组是引用变量的数组，类似于之前见过的多维数组。我们前面说过声明数组时的type可以是复合数据类型，即这里的类和对象。

对象数组意味着数组中的每个元素都是指向某个对象的引用。

例如我们可以创建一个Human类型的对象数组humanArray，如下所示。

Human[] humanArray = new Human[2];

humanArray[0] = new Human();

humanArray[1] = new Human(5);





1. 原始类型（Primitive Types）—— 值传递（复制数据）

Java 里一共有 8 种原始类型（primitive types）：

byte, short, int, long, float, double, char, boolean

当你把一个原始类型的变量赋值给另一个变量时，实际上是“复制了一份值”：

int a = 5;

int b = a; // b 会复制 a 的值（5），此后 a、b 互不影响

b = 10;

System.out.println(a); // 仍然输出 5

同样，如果把原始类型当作方法参数传进去，也只是传递值的副本（call by value）。方法内部改变副本，并不会影响到外部原来变量的值。

特点：

直接存储在栈（stack）或寄存器里（也可以说只是存储在局部变量表里），并不指向堆区的对象。

赋值或方法调用时，复制的是“数据本身”；改变一个变量，另一个变量不受影响。

2. 引用类型（Reference Types）—— 复制引用（指向同一个对象）

除了上面那 8 种原始类型，其他所有类型都是引用类型，包括：

类（class）、接口（interface）的实例，例如你写的 Example obj = new Example();

数组（int[]、String[]、Object[] 等）

枚举类型（enum）也是引用类型

注解类型（@interface）也算引用类型

这些类型底层都是在\*\*堆（heap）\*\*上申请一块内存给对象存放，而你在栈上（或者说在局部变量表里）存的是一个“引用”（reference），指向堆上那块内存里的数据。

Example obj1 = new Example(); // obj1 在栈里存的是一个引用，指向堆里 new Example() 的对象

obj1.x = 5;

// 下面这句不是创建一个新对象，而是复制引用：

Example obj2 = obj1;

// 现在 obj2 和 obj1 都指向同一个堆中 Example 实例

obj2.x = 10;

// 既然 obj2 和 obj1 指向同一个实例，修改 obj2.x 也就等同于修改 obj1.x

System.out.println(obj1.x); // 输出 10

赋值时（obj2 = obj1）只会复制“引用”这一份值，不会新建对象。

对象内部字段（field）或状态一起被共享，只要其中一个引用修改了对象的状态，另一个引用再去访问，就会看到修改后的值。

3. 什么时候是“直接修改”（值类型），什么时候是“引用指向同一个对象”？

如果变量声明的是原始类型（int、double、boolean 等）：

赋值、作为方法参数传递，都是“复制值”；因此修改一个，不会影响另一个。

示例：

int a = 100;

int b = a; // b = 100，此后 a 和 b 完全独立

b = 200;

System.out.println(a); // 还是 100

如果变量声明的是引用类型（new 出来的对象或者数组）：

赋值时复制的是引用，指向同一个堆中实例；方法参数传递的也是引用的拷贝，但拷贝后仍指向同一个对象。

因此如果对象是可变的，你在一个引用上改它的内部状态，其他引用都能立刻“看到”这个变化。

示例：

// 数组也是引用类型

int[] arr1 = {1, 2, 3};

int[] arr2 = arr1; // arr2 和 arr1 指向同一个数组对象

arr2[0] = 99;

System.out.println(arr1[0]); // 输出 99

要注意：Java 里没有所谓的“值类型对象”（value object），所有用 new 创建的都是引用类型；但有些对象本身是“不可变的”——比如 String、Integer（包装类）、LocalDate 等。这些“不可变对象”看起来像“值类型”，因为你改不了它们内部的状态，只能通过重新赋值得到新对象。例如：

String s1 = "hello";

String s2 = s1; // s2、s1 指向同一个 String 对象

s2 = s2 + " world"; // 其实是新建了一个字符串 "hello world"，s2 才指向新对象

System.out.println(s1); // 仍然是 "hello"

尽管 String 是引用类型，但它自己内部不可变，所以看上去“修改”时反而不会影响另一个引用。

4. 方法调用时的传递行为

Java 的方法参数传递一律都是“按值传递”（pass-by-value），不管你传什么类型。但“值”的含义对原始类型和引用类型不一样：

对于原始类型 int x，传进去的是 x 本身的数值，方法里改不会影响外面。

对于引用类型 Example obj，传进去的是 obj 保存的“引用值（一个指针/地址）”，方法里如果你用这个引用去修改对象内部字段，则会影响外面传入的那个对象。但如果你在方法里做 obj = new Example()、重新赋一个新对象给局部参数，那只是改变了方法内部这个局部引用，外部引用并不受影响。

举个对比：

public class Test {

static void modifyPrimitive(int n) {

n = 99; // 只是修改 n 的局部拷贝，不影响外部

}

static void modifyObjectField(Point p) {

p.x = 999; // 通过引用修改堆中对象的字段，外部也会看到变化

}

static void reassignObject(Point p) {

p = new Point(123, 456);

// 这时只是改变了方法内部 p 的指向，外部传进来的引用不受影响

}

public static void main(String[] args) {

int a = 5;

modifyPrimitive(a);

System.out.println(a); // 仍然是 5

Point pt = new Point(1, 1);

modifyObjectField(pt);

System.out.println(pt.x); // 999，确实被改了

reassignObject(pt);

System.out.println(pt.x); // 999，不会变成 456，因为方法内部的 p = new Point(...) 只改变了局部引用

}

}

5. 常见的“什么时候是值传递、什么时候是引用传递”的总结

直接使用 int、boolean、double… 这类原始类型时：

赋值、传参都是复制“值”本身，互不影响。

使用 new 创建的对象（包括数组）时：

赋值、传参复制的是“引用”（地址）。多个引用指向同一个堆内存，修改一个引用访问的对象状态，会被其他引用“看到”。

某些不可变类（如 String、包装类 Integer/Long、LocalDate 等）：

虽然它们技术上也是引用类型，但由于内部状态不可变，所以看上去像值类型：一旦创建就不能再修改，修改操作（比如 s = s.concat("x")）实际会创建新对象，原来引用并不会被“就地改变”。

小结

“引用类型”什么时候生效？

只要你在堆上 new 了一个对象（或数组），它就是引用类型。赋值时复制的是指向那块堆内存的引用，也就是“多个变量/参数指向同一个实例”。因此改变这个实例的字段，会被所有指向它的引用看到。

“直接修改”什么时候生效？

只针对原始类型变量（int、char、boolean…）。赋值/传参时复制的是“数值本身”，改一个不会影响另一个。

理解了这两类——“原始类型（值复制）” vs “引用类型（引用复制）”——之后，就能明白：

为什么 Example obj2 = obj1; obj2.x = 10; 会影响到 obj1.x，因为 obj1、obj2 都指向同一块堆内存。

而如果是 int a = 5; int b = a; b = 10;，则互不干扰。

希望这样能帮助你更清楚地分辨 Java 中哪些情况“是引用传递”，哪些情况“是值复制”。如果还有更具体的场景（比如某个类型不确定，或者自定义类/数组传参等），也可以接着讨论。

所以8大基本类型和其包装类的修改不会影响到传参前的值对吗

ChatGPT 说：

是的，基本类型（byte/short/int/long/float/double/char/boolean）和它们的包装类（Byte/Short/Integer/Long/Float/Double/Character/Boolean）在作为方法参数传递时，都不会改变调用方传入的原值。原因如下：

基本类型（primitive）一定是值传递

void f(int x) {

x = 100; // 只是修改了 f 方法内部的局部拷贝，不影响外面传进来的那个 int 变量

}

public static void main(String[] args) {

int a = 5;

f(a);

System.out.println(a); // 仍然输出 5

}

无论你在方法里怎么改 x，都只是改了「拷贝」的那份数据，调用处的 a 完全不受影响。

包装类（Wrapper Class）本身是引用类型，但它是不可变（immutable）对象

像 Integer、Double、Boolean 这些包装类型，虽然变量存的是“引用”，但它们内部的值一旦创建就不能修改（没有可以“就地改变内部状态”的方法）。

所以即使你把一个 Integer 传给方法，方法里如果做了 i = i + 1;，实际上是新建了一个 Integer 对象并让方法的局部引用指向它，调用方的原引用并不会被“就地改动”。

void g(Integer x) {

x = x + 10; // 这里相当于新生成一个 Integer 对象，再让 g 方法内部的 x 指向它

}

public static void main(String[] args) {

Integer a = 5;

g(a);

System.out.println(a); // 仍然输出 5

}

方法里不能改变原来的 Integer 对象（因为它没有类似 setValue(...) 这样的可变操作），只能“重新赋新对象给局部变量”。因此，传参前的那份引用在方法外部依然保持原样。

二！！！！！

1. Immutable Objects and Classes（不可变对象和不可变类）

不可变对象（Immutable object）是指一旦创建，其内容就无法被修改的对象。

不可变类（Immutable class）是指如果一个类的所有对象都是不可变的，那么这个类被称为不可变类。

不可变类的例子如下。

public class Circle {

private double radius;

public Circle() { }

public Circle(double radius) {

this.radius = radius;

}

public double getRadius() {

return radius;

}

}

为什么说这里Circle类是不可变类呢，因为radius字段被声明为private，这意味着外部代码无法直接访问或修改它，而类中没有提供setRadius方法来允许外部代码修改radius的值。一旦对象被创建，radius的值就无法被改变，所以radius的值只能在对象创建时通过构造函数设置，一旦设置就无法修改。

但仅仅将所有字段设为private，并且没有提供修改字段的方法（如set方法），并不足以使一个类成为不可变类，还有其他因素需要考虑。例如，如果类中包含可变对象的引用，那么即使字段是private，对象的状态仍然可能被外部代码修改。

1.1 不可变类的三个条件

因此一个不可变类必须满足以下三个条件：

1.所有字段必须是私有的（private）：这是为了隐藏内部状态，防止外部代码直接访问或修改字段。

2.不提供任何修改字段的方法（如set方法）：这样可以确保字段的值在对象创建后不会被改变。

3.不提供任何返回可变字段引用的访问器方法（如get方法）：如果类中包含可变对象的引用（如数组、集合或其他可变类的实例），那么返回这些引用可能会导致外部代码修改对象的状态。例如，如果类中有一个ArrayList字段，提供一个get方法返回这个ArrayList的引用，外部代码就可以通过这个引用修改列表的内容，从而破坏不可变性。

1.2 变量的作用域（Scope of Variables）

1.2.1 局部变量

对于局部变量来说，其作用域从其声明开始，一直持续到包含该变量的代码块（block）结束。代码块是指由花括号 {} 包围的一段代码，例如方法体、循环体或条件语句体。

当然局部变量在使用前必须显式初始化，否则编译器会报错，因为局部变量没有默认值。

1.2.2 类变量

类变量（包括实例变量和静态变量）可以在类的任何位置声明，通常放在类的顶部，与方法分开。

实例变量和静态变量的作用域是整个类。这意味着它们可以在类的任何方法中被访问。再次复习一遍，实例变量属于对象实例，而静态变量属于类本身。

1.3 this关键字

这里再次复习一遍this关键字。this关键字是一个引用，它指向当前对象本身。也就是说，它指向调用该方法或访问该字段的实例。

我们上次说了当局部变量（如方法参数或局部变量）与类的实例字段同名时，局部变量会“隐藏”实例字段。在这种情况下，可以使用this关键字来明确引用类的实例字段。

如下所示。

public class Foo {

private int i = 5; // 实例变量，每个Foo对象都有独立的i值

private static double k = 0; // 静态变量，所有Foo对象共享同一个k值

void setI(int i) { // 实例方法，用于设置实例变量i的值

this.i = i; // 使用this关键字来明确引用当前对象的实例变量i

}

static void setK(double k) { // 静态方法，用于设置静态变量k的值

Foo.k = k; // 直接使用类名Foo来引用静态变量k

}

}

这里i是实例变量，因此可以使用this关键字，而k是静态变量，属于类本身，所以对象都共享同一个静态变量，因此不能使用this关键字，而是使用类名。

其实this关键字还可以用于 调用同一个类的另一个构造函数，作为其第一条语句。这称为构造函数重载（Constructor Overloading）。

如下所示。

public class Person {

private String name;

private int age;

// 无参构造函数

public Person() {

this("Unknown", 0); // 调用另一个构造函数

}

// 带两个参数的构造函数

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public void printInfo() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

下面的代码同时体现了这两点。

public class Circle {

private double radius;

// 带参数的构造函数

public Circle(double radius) {

this.radius = radius; // 使用this关键字来引用当前对象的实例变量radius

}

// 无参构造函数

public Circle() {

this(1.0); // 使用this关键字调用带参数的构造函数

}

// 计算圆的面积的方法

public double getArea() {

return this.radius \* this.radius \* Math.PI; // 使用this关键字来明确引用当前对象的实例变量radius

}

}

1.4 类抽象和封装（Class Abstraction and Encapsulation）

类抽象是指将类的实现细节与类的使用分离。这通常通过提供类的公共接口（API）来实现，而隐藏类的内部实现细节。

通过抽象，类的创建者可以定义类的行为和功能，即提供类的描述和公共方法，这些方法定义了类可以执行的操作，而不必让使用者了解类的具体实现方式。这使得使用者可以专注于如何使用类，通过这些方法与类交互，而不需要知道类内部的具体实现。

而不让用户知道类内部是如何实现的，这也就是封装（encapsulated），将数据字段设置为private，只开放一些公共接口（API）。

封装是指将对象的状态（属性）和行为（方法）捆绑在一起，并隐藏对象的内部实现细节。这通常是通过将类的字段设置为私有（private）并提供公共方法（如getter和setter）来访问和修改这些字段来实现的。

2. String类

前面我们说过String其实也是一个类，因此我们创建一个新的字符串对象可以使用new关键字，格式如下。

String newString = new String(stringLiteral);

例子如下。

String message = new String("Welcome to Java");

Java提供了一种更简便的方式来创建字符串，即直接使用字符串字面量。  
即上面的例子可以简便结果如下。

String message = "Welcome to Java";

2.2 String类是不可变类

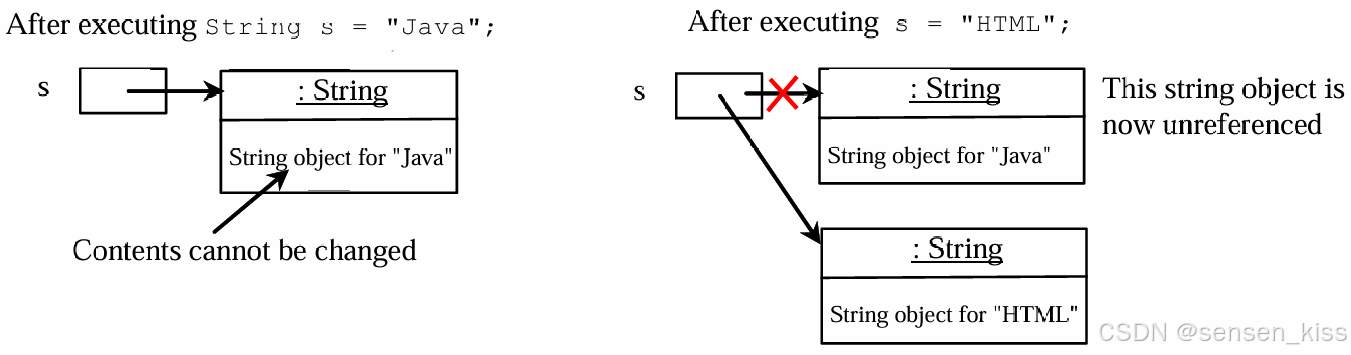
String类也是前文说的不可变类。

String s = "Java";

s = "HTML";

System.out.println(s); // 输出的结果是HTML

这里虽然输出的结果是HTML，但其实是代码并没有修改原来的字符串对象 “Java”，而是让变量 s 引用了一个新的字符串对象，该对象表示字符串"HTML"。其实是和前面的渐变方式创建字符串一样，s = "HTML"时创建了一个新的字符串并将其引用赋值给了s，而原来的字符串对象"Java"仍然存在于字符串池中，但由于没有变量引用它，因此它现在是一个未被引用的对象。



2.3.1 比较字符串

1.equals(Object obj): boolean 检查调用此方法的字符串是否与参数字符串内容相同。它是值比较，而不是引用比较。

如下所示。

String s1 = new String("Welcome");

String s2 = "Welcome";

if (s1.equals(s2)){ // true

// s1和s2有一样的值

}

if (s1 == s2) { // false

// s1和s2的引用值不同

}

这里使用两种方法创建创建了字符串，前者是在字符串池中创建，后者是在堆中创建，因此引用的对象不同，但引用的对象的内容是相同的。

下面的代码结果不同。

String s1 = "Welcome";

String s2 = "Welcome";

if (s1.equals(s2)){ // true

// s1 and s2 have the same contents

}

if (s1 == s2) { // true

// s1 and s2 have the same reference

}

因为字符串内部化，所以现在 s1 和 s2 指向同一个对象。

2.compareTo(String str): int 将调用此方法的字符串与参数字符串按字典顺序进行比较，并返回一个整数。如果调用字符串在参数字符串之前，返回负数；如果相等，返回0；如果调用字符串在参数字符串之后，返回正数。

字典顺序是比较字符串时常用的方法，它基于字符的Unicode值进行比较。示例如下。

String s1 = new String("Welcome");

String s2 = new String("Welcome");

if (s1.compareTo(s2) > 0) {

// s1 is greater than s2

} else if (s1.compareTo(s2) == 0) {

// s1 and s2 have the same contents

} else{

// s1 is less than s2

}

这里代码会运行中间的部分。

更多的比较字符串的方法如下：

1.equalsIgnoreCase(String str): boolean 比较调用此方法的字符串对象是否与参数字符串对象str的内容相同，忽略大小写。返回true如果内容相同（不区分大小写），否则返回false。

2.compareToIgnoreCase(String str): int 与compareTo相同，但是比较时忽略大小写。

3.regionMatches(int toffset, String str, int offset, int len): boolean 检查调用此方法的字符串对象从toffset开始的特定区域是否与参数字符串对象str从offset开始的特定区域匹配。返回true如果区域匹配，否则返回false。

4.regionMatches(Boolean ignoreCase，int toffset, String str, int offset, Int len): boolean 与上一个方法相同，但是可以指定比较是否忽略大小写。返回true如果区域匹配（根据ignoreCase参数决定是否忽略大小写），否则返回false。

5.startsWith(String prefix): boolean 检查调用此方法的字符串对象是否以指定的前缀字符串prefix开始。返回true如果字符串以指定前缀开始，否则返回false。

6.endsWith(String suffix): boolean 检查调用此方法的字符串对象是否以指定的后缀字符串suffix结束。返回true如果字符串以指定后缀结束，否则返回false。

2.3.2 获取字符串长度

length(): int 返回字符串的长度，即字符的数量。

2.3.3 检索字符串中的单个字符

charAt(int index): char 这个方法返回指定索引处的字符。索引是从 0 开始的，所以第一个字符的索引是 0，第二个字符的索引是 1，依此类推。

注意这里charAt返回的是 char 而不是 string 类型。

下图展示了上面两个方法的使用。

2.3.4 字符串连接

concat(String str): String将指定字符串连接到调用此方法的字符串的结尾。

2.3.5 子字符串

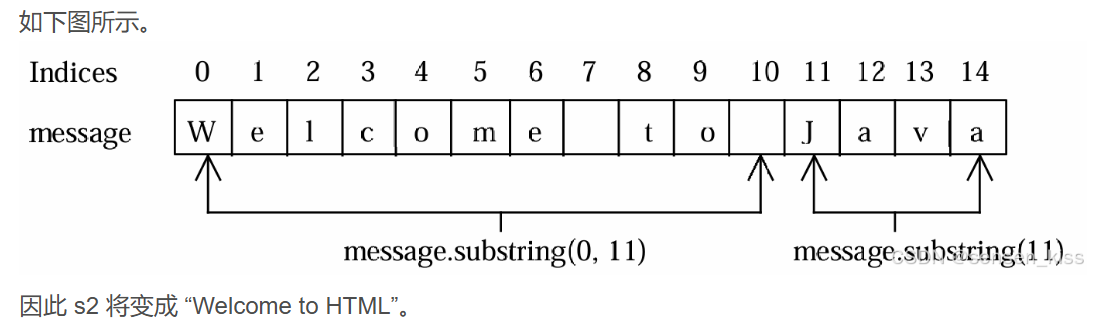
1.substring(int beginIndex): String 从beginIndex开始截取到字符串结尾的子字符串。

2.substring(int beginIndex, int endIndex): String 从beginIndex开始截取到endIndex之前的子字符串。也就是包含 beginIndex，不包含 endIndex。

例子如下。

String s1 = "Welcome to Java";

String s2 = s1.substring(0, 11) + "HTML";



2.3.6 在字符串中查找字符或子字符串

1.indexOf(char ch): int返回字符 ch 在字符串中第一次出现的索引。如果未找到该字符，则返回 -1。

2.indexOf(char ch, int fromIndex): int 返回字符 ch 在字符串中从 fromIndex 之后第一次出现的索引。如果未找到该字符，则返回 -1。

3.indexOf(String str)：int 返回指定子字符串在此字符串中第一次出现的索引。如果未找到该字符，则返回 -1。

4.indexOf(String str,int formIndex): int 返回子字符串 str 在字符串中从 fromIndex 之后第一次出现的索引。如果未找到该子字符串，则返回 -1。

5.lastIndexOf(char ch): int 返回字符 ch 在字符串中最后一次出现的索引。如果未找到该字符，则返回 -1。

6.lastIndexOf(char ch, int fromIndex): int 返回字符 ch 在字符串中从 fromIndex 之前最后一次出现的索引。如果未找到该字符，则返回 -1。

7.lastIndexOf((String str)：int 返回子字符串 s 在字符串中最后一次出现的索引。如果未找到该子字符串，则返回 -1。

8.lastIndexOf(String str,int formIndex): int 返回子字符串 s 在字符串中从 fromIndex 之前最后一次出现的索引。如果未找到该子字符串，则返回 -1。

2.3.7 通过模式匹配、替换和分割

1.toLowerCase(): String 返回一个新的字符串，该字符串是调用此方法的字符串转换为小写形式的结果。

2.toUpperCase(): String 返回一个新的字符串，该字符串是调用此方法的字符串转换为大写形式的结果。

3.trim(): String 返回一个新的字符串，该字符串是调用此方法的字符串去除两端空白字符后的结果。

4.replace(char oldChar, char newChar): String 返回一个新的字符串，该字符串是调用此方法的字符串中所有匹配的字符 oldChar 被替换为 newChar 后的结果。

5.replaceFirst(String oldString, String newString): String 返回一个新的字符串，该字符串是调用此方法的字符串中第一个匹配的子字符串 oldString 被替换为 newString 后的结果。

6.replaceAll(String oldString, String newString): String 返回一个新的字符串，该字符串是调用此方法的字符串中所有匹配的子字符串 oldString 被替换为 newString 后的结果。

7.split(Stirng delimiter): String[] 返回一个字符串数组，该数组包含调用此方法的字符串按照指定分隔符 delimiter 分割后的子字符串。

例子如下：

1.“Hello”.toLowerCase() 返回 “hello”。

2.“hello”.toUpperCase() 返回 “HELLO”。

3." Hello World “.trim() 返回 “Hello World”。

4.“Hello World”.replace(‘l’, ‘r’) 返回 “Hemmo World”。

5.“Hello World”.replaceFirst(“World”, “Java”) 返回 “Hello Java”。

6.“Hello World”.replaceAll(“l”, “AB”) 返回 “HeABABo World”。

7.“one,two,three”.split(”,") 返回 [“one”, “two”, “three”]。

2.3.7.1 使用正则表达式作为参数

在Java中，String 类的 replaceAll、replaceFirst、split 和 matches 方法可以接受正则表达式作为参数来指定搜索模式。

示例如下。

String s = "a+b$#c".replaceAll("[$+#]", "NNN");

System.out.println(s);

[$+#] 是一个字符集（character class），它匹配字符集中的任意一个字符。在这个例子中，字符集包含 $、+ 和 # 三个字符。每当在字符串中找到这些字符中的任意一个时，它都会被替换为 “NNN”。

因此最后的结果是"aNNNbNNNNNc"。

相关的方法如下。

1.matches(String regex): Boolean 返回一个布尔值，指示调用它的字符串是否匹配给定的正则表达式模式。

2.replaceAll(String regex, String replacement): String 返回一个新字符串，该字符串是调用它的字符串中所有匹配的子字符串被替换为指定的替换字符串后的结果。

3.replaceFirst(String regex, String replacement): String 返回一个新字符串，该字符串是调用它的字符串中第一个匹配的子字符串被替换为指定的替换字符串后的结果。

4.split(String regex): String[] 返回一个字符串数组，该数组包含调用它的字符串按照给定的正则表达式模式分割后的子字符串数组。

看ppt

#### **2.3.8 将字符和数字值转换为字符串：**

1.valueOf() 系列方法：将不同类型的数据（如基本数据类型、对象等）转换为字符串。  
2.toString() 方法：许多类都重写了此方法，以提供其实例的字符串表示。

2.4 StringBuilder 和 StringBuffer 类

StringBuilder 和 StringBuffer 类，它们是 String 类的替代品，提供了更灵活的字符串操作能力。

StringBuilder 和 StringBuffer 类提供了与 String 类似的方法，用于创建和操作字符串，这些类可以用于任何需要字符串的场景。

StringBuffer 是同步的（synchronized），意味着它是线程安全的。同步机制确保了同一时刻只有一个线程可以执行 StringBuffer 的方法，从而避免了多线程同时修改同一个 StringBuffer 对象时可能出现的竞争条件。

StringBuilder 是非同步的（non-synchronized），这意味着它不是线程安全的。在多线程环境中，如果多个线程同时调用 StringBuilder 的方法，可能会导致数据不一致或其他问题。

StringBuilder 和 StringBuffer 提供了一系列方法，如 append()、insert() 和 delete() 等，可以在不创建新字符串对象的情况下修改字符串内容。

与 String 类不同，String 是不可变的（immutable），一旦创建，其内容不能被修改。任何对 String 的修改都会生成一个新的 String 对象。

2.4.1 构造函数

1.StringBuilder() 这个构造函数创建了一个空的StringBuilder对象，其初始容量为16个字符。这意味着StringBuilder对象一开始有足够的空间来存储16个字符，如果添加的字符超过这个容量，它会自动扩展。

2.StringBuilder(int capacity) 这个构造函数允许你指定创建的StringBuilder对象的初始容量。你可以传递一个整数值capacity来设置初始容量，这将决定StringBuilder对象一开始可以存储多少个字符。

3.StringBuilder(String s) 这个构造函数创建了一个StringBuilder对象，并用指定的字符串s来初始化它。传递给构造函数的字符串s将被用作StringBuilder对象的初始内容，初始容量与传入的字符串 s 的长度是一致的。

2.4.2 相关方法

1.append(char[] data): StringBuilder 将一个字符数组追加到这个字符串构建器中。

2.append(char[] data, int offset , int len): StringBuilder 将字符数组data中从offset开始的len个字符追加到这个字符串构建器中。

3.append(PrimitiveType v): StringBuilder 将一个基本类型值（如int、double等）追加为字符串形式到这个字符串构建器中。

4.append(String s):StringBuilder 将一个字符串追加到这个字符串构建器中。

5.delete(int startIndex, int endIndex): StringBuilder 删除从startIndex到endIndex（不包含endIndex）的字符。

6.deleteCharAt(int index): StringBuilder 删除指定索引处的字符。

7.insert(int index, char[] data, int offset, int len): StringBuilder 在指定索引处插入字符数组data中从offset开始的len个字符。

8.insert(int offset, char[] data): StringBuilder 在指定位置offset插入字符数组。

9.insert(int offset, PrimitiveType v): StringBuilder 在指定位置插入一个基本类型值（如int、double等）转换成的字符串。

10.insert(int offset, String s): StringBuilder 在指定位置插入一个字符串。

11.replace(int startIndex, int endIndex, String s): StringBuilder 用指定的字符串s替换从startIndex到endIndex（不包含endIndex）的字符。

12.reverse(): StringBuilder 反转字符串构建器中的字符顺序。

13.setCharAt(int index,char ch): void 设置指定索引处的新字符。

2.5 Character类

Character类和String类比较像，但是String类代表一个不可变的字符序列，Character类代表一个单独的字符，其是可变的。

2.5.1 相关方法

1.Character(char value) 使用指定的字符值构造一个字符对象。

2.charValue(): char 返回此对象中的字符值。

3.compareTo(Character anotherCharacter): int 将此字符与另一个字符对象进行比较。如果此字符在字母顺序中位于参数字符之前，则返回一个负整数；如果位于之后，则返回一个正整数；如果两个字符相等，则返回0。

4.equals(Character anotherCharacter): boolean 如果此字符与另一个字符相等，则返回true。

5.isDigit(char ch): boolean 如果指定的字符是数字，则返回true。

6.isLetter(char ch): boolean 如果指定的字符是字母，则返回true。

7.isLetterOrDigit(char ch): boolean 如果指定的字符是字母或数字，则返回true。

8.isLowerCase(char ch): boolean 如果字符是小写字母，则返回true。

9.isUpperCase(char ch): boolean 如果字符是大写字母，则返回true。

10.toLowerCase(char ch): char 返回指定字符的小写形式。

11.toUpperCase(char ch): char 返回指定字符的大写形式。

面向对象编程的三个核心概念：内聚性（Coherence）、职责分离（Separating responsibilities）和重用性（Reuse）。这些原则有助于设计清晰、灵活且可维护的类。

1.内聚性（Coherence）：

内聚性指的是一个类应该描述一个单一的实体（entity），并且该实体的所有属性和行为都应该与该实体的概念紧密相关。一个类应该“做一件事并做好”，这意味着类中的属性和方法都与类的主要目的相关联。内聚性好的类通常更容易理解和维护，因为它们具有明确和一致的职责。

2.职责分离（Separating responsibilities）：

职责分离是指将具有多个职责的单一实体分解为多个类，每个类负责一部分职责。

当一个类承担了过多的职责时，这可能导致类变得复杂和难以管理。通过将不同的职责分配给不同的类，可以简化每个类的设计，提高代码的可读性和可维护性。

3.重用性（Reuse）：

重用性是指设计类时考虑到类的可重用性，即在不同的上下文中可以多次使用同一个类。通过创建可重用的类，可以避免代码重复，减少开发时间和维护成本。重用性也有助于提高代码的一致性，因为相同的功能在不同的地方使用相同的实现。

除了这三个核心思想以外，还有一些标准编程风格和命名的约定，些约定有助于编写清晰、一致且易于理解的代码。

1.遵循标准的Java编程风格和命名约定：

编写代码时应该遵循Java社区广泛接受的编程风格和命名规则，例如变量、方法和类名应该清晰、简洁且具有描述性。

即为类、数据字段和方法选择有意义的名称。

类名、字段名和方法名应该能够清楚地表达其作用和用途，以便于其他开发者理解代码的功能和行为。

2.将数据声明放在构造函数之前，构造函数放在方法之前：

类的字段声明应该在构造函数之前，这样可以让字段的初始化在对象创建时立即进行。

构造函数应该在其他方法之前，因为构造函数用于初始化对象，而其他方法通常依赖于对象的初始化状态。

3.尽可能提供一个公共的无参构造函数，并重写equals方法和toString方法（返回一个字符串）：

提供一个无参的公共构造函数可以让类的实例更容易被创建，即使没有特定的初始化参数。

重写equals方法可以确保类的实例可以根据其内容进行比较，而不仅仅是引用比较。

重写toString方法可以提供类的实例的字符串表示，这对于调试和日志记录非常有用。

3.1 继承（Inheritance）

前面我们说的是类设计中基础的一些概念，以及我们在上一章中介绍了我们该怎么想着去创造一个类。现在我们回到最初需要类的动机上，在现实世界中，许多实体（如圆形、矩形和三角形）具有共同的特征和行为（例如，颜色、是否填充、创建日期等），而这些共同的特征和行为可以在代码中表示为共享的数据字段和方法。

下面我们介绍类的一个进阶知识——继承（Inheritance）。

继承是一种机制，允许一个子类（sub-class）扩展另一个父类（super-class）。

子类继承父类的属性和方法，从而可以重用已有的代码，而不需要重新编写。继承可以帮助我们设计和实现类，使得代码结构更加清晰，易于理解和扩展。父类可以提供一个通用的框架，而子类可以专注于特定的实现细节。

我们一般用关键字 abstract 声明抽象类或抽象方法，当一个类被声明为abstract时，它不能被实例化，即你不能创建这个类的对象。

抽象类通常用作其他类的基类，提供通用的属性和方法，这些属性和方法可以在子类中被具体实现。

抽象类可以包含抽象方法和具体方法。

如果一个类继承自抽象类，它必须实现抽象类中的所有抽象方法，除非该子类也被声明为抽象。

抽象方法是不包含方法体（即方法的实现部分）的方法，而抽象方法必须在抽象类中声明。

抽象方法用于定义方法的规范或接口，具体的实现由继承该抽象类的子类提供。

因此我们根据前面的例子，我们可以创建一个图形的父类，它定义了所有图形共有的属性和方法。然后我们再创建对应的具体的图形类，这些类继承父类。

public abstract class GeometricObject {

private String color = "white";

private boolean filled;

private java.util.Date dateCreated;

protected GeometricObject() {

dateCreated = new java.util.Date();

}

protected GeometricObject(String color, boolean filled) {

this();

this.color = color;

this.filled = filled;

}

public String getColor() { return color; }

public void setColor(String color) { this.color = color; }

public boolean isFilled() { return filled; }

public void setFilled(boolean filled) { this.filled = filled; }

public java.util.Date getDateCreated() { return dateCreated; }

public String toString() {

return "color: " + color + ", filled: " + filled +

", created on " + dateCreated;

}

/\*\* Abstract method getArea \*/

public abstract double getArea();

/\*\* Abstract method getPerimeter \*/

public abstract double getPerimeter();

}

创建完GeometricObject类后，再创建Circle类和Rectangle类继承GeometricObject类。

public class Circle extends GeometricObject {

private double radius;

public Circle() { }

public Circle(double radius) {

this.radius = radius;

}

public double getRadius() {

return radius;

}

public void setRadius(double radius) {

this.radius = radius;

}

public String toString() {

return "Circle with radius is " + radius + ", " + super.toString();

}

public double getArea() {

return radius \* radius \* Math.PI;

}

public double getPerimeter() {

return 2 \* radius \* Math.PI;

}

public double getDiameter() {

return 2 \* radius;

}

}

public class Rectangle extends GeometricObject {

private double width;

private double height;

public Rectangle() {

// super();

}

public Rectangle(double width, double height) {

this();

this.width = width;

this.height = height;

}

public Rectangle(double width, double height, String color, boolean filled) {

super(color, filled);

this.width = width;

this.height = height;

}

public double getWidth() { return width; }

public void setWidth(double width) { this.width = width; }

public double getHeight() { return height; }

public void setHeight(double height) { this.height = height; }

public double getArea() {

return width \* height;

}

public double getPerimeter() {

return 2 \* (width + height);

}

}

public static void main(String[] args) {

// Declare and initialize two geometric objects

GeometricObject geoObject1 = new Circle(5);

GeometricObject geoObject2 = new Rectangle(5, 3);

// Display circle

displayGeometricObject(geoObject1);

// Display rectangle

displayGeometricObject(geoObject2);

System.out.println("The two objects have the same area? " + equalArea(geoObject1, geoObject2));

}

/\*\* A method for displaying a geometric object \*/

public static void displayGeometricObject(GeometricObject object) {

System.out.println(object); // object.toString()

System.out.println("The area is " + object.getArea());

System.out.println("The perimeter is " + object.getPerimeter());

}

/\*\* A method for comparing the areas of two geometric objects \*/

public static boolean equalArea(GeometricObject object1, GeometricObject object2) {

return object1.getArea() == object2.getArea();

}

这里测试的时候我们geoObject1和geoObject2两个对象声明的时候用的类型是GeometricObject，这里使用的是父类的引用类型来引用子类的实例，这便是多态性（polymorphism），后面我们会再详细介绍多态性。

3.1.1 声明子类

前面的代码中我们先声明了一个父类，再声明了两个子类。我们声明子类的时候，它自动继承其父类的所有属性（字段）和方法。这意味着子类可以使用父类中定义的所有公共（public）和受保护（protected）属性和方法，而无需重新编写这些代码。

而子类可以添加自己的属性，这些属性在父类中不存在，这些新属性可以满足子类特有的需求，而不影响父类。（例如Circle类里面的radius，Rectangle类里的width）。

子类还可以定义自己的方法，这些方法在父类中不存在，这些新方法可以提供新的功能，或者对现有功能进行扩展。（例如Circle类里面的getRadius()，Rectangle类里面的setWidth()）

子类可以重写（Override）父类的方法，以提供特定于子类的行为。

重写方法时，子类的方法签名必须与父类中的方法签名相同。通过重写，子类可以改变父类方法的行为，使其更适合子类的需求。（例如Circle类和Rectangle类里的getArea()和getPerimeter()）

我们前面的测试代码中，我们声明了两个GeometricObject类的对象geoObject1和geoObject2，我们可以调用getArea()方法，这里其实使用的是子类Circle和Rectangle中的方法，这同样因为多态性，但我们不能直接调用子类的方法，例如我们无法使用geoObject2调用getWidth()，因为在编译时，geoObject2的静态类型是GeometricObject，而不是Rectangle。为了调用getWidth()方法，需要将geoObject2向下转型为Rectangle类型才可以调用。

3.1.2 构造函数

超类的构造函数不会被继承。子类需要显式或隐式地调用超类的构造函数。

1.显式调用：

使用super关键字和超类构造函数的参数来显式调用超类的构造函数。

2.隐式调用：

如果没有显式使用super关键字，那么超类的无参构造函数（即不接受任何参数的构造函数）会在子类构造函数的第一行自动被调用，除非使用this关键字调用了另一个构造函数。在这种情况下，链条中最后一个构造函数将调用超类的构造函数。（这里的链条指的是一个构造函数使用this关键字调用了另一个构造函数，这种调用形成了一个构造函数链，其中每个构造函数可以调用同一个类中的其他构造函数。）

示例如下。

public A(args) {

super(); // 显式调用超类的无参构造函数

// some statements

}

public A(args) {

// 隐式调用超类的无参构造函数

// some statements

}

这两种写法等效。

3.2 super关键字

super关键字我们前面就使用过，它具体的功能是什么呢？

super关键字用于引用当前对象的父类。在Java中，每个类都有一个直接的父类（除了 Object 类，它是所有类的根类）。

它具体的作用主要是以下两点。

1.构造函数链式调用：

super关键字用于在子类的构造函数中调用父类的构造函数，这称为构造函数链式调用（Constructor Chaining）。这确保了父类的初始化逻辑在子类对象创建时被执行。

2.调用超类方法：

当子类重写了父类的方法时，可以使用super关键来调用被重写（Override）的父类方法。这允许访问在子类中被隐藏的父类方法。

3.2.1 构造函数链式调用（Constructor Chaining）

前面说了一种链条关系，但这里更多是强调构造一个类的实例需要沿着继承链调用超类的构造函数。它们其实都想表达的是当多个构造函数在一起形成了一个构造函数链时，这个链可以确保对象在完全构造之前，所有的初始化逻辑都按顺序执行。

例子如下。

public class Faculty extends Employee {

public static void main(String[] args) {

new Faculty();

}

public Faculty() {

System.out.println("(4) Faculty's no-arg constructor is invoked");

}

}

class Employee extends Person {

public Employee() {

this("(2) Invoke Employee’s overloaded constructor");

System.out.println("(3) Employee's no-arg constructor is invoked");

}

public Employee(String s) {

System.out.println(s);

}

}

class Person {

public Person() {

System.out.println("(1) Person's no-arg constructor is invoked");

}

}

这里先是创建了一个Person父类，然后Employee子类继承Person类下面有一个无参数构造类，它会调用另一个构造函数，最后是Faculty子类它继承Employee类。

因此我们使用new Faculty()会调用Faculty类的无参构造参数，然后因为继承关系，它隐式调用super()，因此调用了Employee类的无参构造函数，然后这个函数又显式调用了另一个带有String参数的构造函数，然后又是因为继承关系，所以它又隐式调用super()，因此调用了Person类的无参构造函数。这个链就结束了，因此链的最后一个最先执行，也就是先执行Person类的无参构造函数。

最后结果如下。

(1) Person's no-arg constructor is invoked

(2) Invoke Employee’s overloaded constructor

(3) Employee's no-arg constructor is invoked

(4) Faculty's no-arg constructor is invoked

3.2.2 调用超类方法

前面的Circle类就是一个例子。

public String toString() {

return "Circle with radius " + radius + ", " + super.toString();

}

这里是重写父类里的toString()方法，因此现在使用super关键字调用被隐藏的父类方法。

我们这里顺便再详细说一下方法重写（Method overriding）的概念。方法重写是指在子类中修改父类中已定义的方法的实现。这是多态性的一种表现，允许子类根据需要改变从父类继承来的方法的行为。

在子类中，方法重写通过使用与父类中被重写的方法相同的签名（方法名和参数列表）来实现。

可以使用 @Override 注解来显式声明一个方法是重写父类中的方法。这个注解是可选的，但使用它可以提高代码的可读性和减少出错的机会。

例如Circle类的这个toString()方法就是重写了父类GeometricObject里的toString()方法。

3.3 Obejct类

前面提到在Java中，java.lang.Object 是所有类的根类（超类）。

因此这意味着每个Java类都直接或间接地继承自Object类。当你定义一个类时，如果没有显式指定它继承自哪个类，那么它默认继承自java.lang.Object类。

Object类提供了一些通用的方法，这些方法对所有对象都是可用的，例如 equals(), hashCode(), toString() 等。

下面的代码是等价的。

public class GeometricObject {

// some statements

}

public class GeometricObject extends Object {

// some statements

}

3.3.1 toString()方法

toString()方法用于返回对象的字符串表示形式。

Object 类的默认实现返回一个字符串，该字符串由对象的类名、@符号和一个数字组成，这个数字是对象的哈希码（hash code）的无符号十六进制表示。

示例如下。

class Loan {

// Loan 类的具体实现

}

public class Main {

public static void main(String[] args) {

Loan loan = new Loan();

System.out.println(loan.toString());

}

}

输出的结果可能类似于Loan@10f87f48其中Loan是对象的类名，@10f87f48是对象的哈希码。这种默认格式不是非常用户友好或信息丰富，因为它主要包含技术细节（类名和哈希码）。因此我们应该重写一个toString()方法从而返回一个更有信息量、更友好的字符串表示形式。

3.3.2 方法重载（Overloading）和方法重写（Overriding）的对比

我们前面说了方法重写（Overriding），比如重写toString()方法。方法重写是指在子类中覆盖父类中具有相同签名（即方法名和参数列表相同）的方法。它需要方法签名相同，包括返回类型和参数列表。

public abstract class GeometricObject {

public abstract double getArea();

}

class Circle extends GeometricObject {

@Override

public double getArea() {

return Math.PI \* radius \* radius;

}

}

方法重载（Overriding）是指在同一个类中可以创建多个同名的方法，但这些方法必须具有不同的参数列表（即参数的数量或类型不同）。

这是编译时多态性，即在同一个类中，根据传入参数的不同，可以调用同名但参数不同的方法。

方法重载不要求方法具有相同的返回类型，参数列表的不同足以区分重载的方法。

示例如下。

public class Overloading {

public static int max(int num1, int num2) {

if (num1 > num2) return num1;

return num2;

}

public static double max(double num1, double num2) {

if (num1 > num2) return num1;

return num2;

}

public static void main(String[] args) {

System.out.println(max(1, 2)); // 调用第一个max方法，参数类型为int

System.out.println(max(1, 2.3)); // 调用第二个max方法，参数类型为double

}

}

3.3.2.1 方法匹配（Method Matching）和方法绑定（Method Binding）

这两个的实现靠的是Java的两个不同机制，方法匹配（Method Matching）和方法绑定（Method Binding）。

1.方法匹配（Method Matching）：当一个类中存在多个同名但参数列表不同的方法时，即当遇到重载方法时，编译器在编译时根据方法的参数类型、参数数量和参数顺序来确定调用哪个方法。

例如，如果一个类中有两个名为doSomething的方法，一个接受int参数，另一个接受double参数，那么编译器会根据调用时提供的参数类型来决定使用哪个方法。

2.方法绑定（Method Binding）：当子类重写了父类中的方法时，即当遇到重写方法时，Java虚拟机（JVM）在运行时动态绑定到最具体的重写方法实现。

这意味着即使引用变量的类型是父类类型，实际执行的也是子类中重写的方法。

例如，如果Child类重写了Parent类的doSomething方法，那么即使有一个 Parent类型的引用指向Child对象，调用doSomething方法时也会执行Child类中的实现。

3.3.3 多态性（Polymorphism）、动态绑定（Dynamic Binding）以及泛型编程（Generic Programming）

1.多态性（Polymorphism）：

多态性是指子类型的对象可以在需要父类型的地方使用。在Java中，这意味着一个对象可以被视为它的任何父类的实例。

2.动态绑定（Dynamic Binding）：

动态绑定是指Java虚拟机（JVM）在运行时动态决定使用哪个方法实现。当一个方法被调用时，JVM会根据传入参数的实际类型（而不是引用类型）来决定调用哪个方法。

所以方法匹配（Method Matching）和方法绑定（Method Binding）都是Java中动态绑定（Dynamic Binding）的一部分，但它们发生在不同的阶段。

3.泛型编程（Generic Programming）：

我们在之前的数据结构课中详细学过这一机制。

泛型编程是Java提供的一种机制，允许类和方法在不知道具体类型的情况下被定义和使用。泛型参数在实例化时指定，这允许编写出更通用、更灵活的代码。

泛型类和接口可以使用类型参数（如T、E等），这些参数在实例化类或接口时确定。

下面的代码体现了前两个概念。

public class PolymorphismDemo {

public static void main(String[] args) {

m(new GraduateStudent());

m(new Student());

m(new Person());

m(new Object());

}

public static void m(Object x) {

System.out.println(x.toString());

}

}

class GraduateStudent extends Student {

// ...

}

class Student extends Person {

public String toString() {

return "Student";

}

}

class Person /\* extends Object\*/ {

public String toString() {

return "Person";

}

}

输出的结果如下。

Student

Student

Person

java.lang.Object@12345678

最后一行的哈希码可能不一样。

这里的多态性体现在m方法接受一个Object类型的参数，这意味着它可以接收任何类型的对象，包括GraduateStudent、Student、Person和Object。

动态绑定是指在运行时根据对象的实际类型来调用相应的方法。在这个例子中，toString()方法在Object类中定义，并被Person、Student和GraduateStudent类重写。当m方法被调用时，JVM根据传入对象的实际类型来决定调用哪个toString()方法。

下面的代码体现了泛型编程。

public class Box<T> {

private T t;

public Box(T t) {

this.t = t;

}

public T get() {

return t;

}

}

在这个泛型类Box中，T是一个类型参数，你可以在实例化Box类时指定具体类型，如Box<String>或Box<Integer>。

3.3.3.1 动态绑定（Dynamic Binding）

假设有一个对象 o 是类 C1 的实例（o = new C1()），其中 C1 是 C2 的子类，C2 是 C3 的子类，依此类推，直到 Cn-1 是 Cn 的子类。

Cn 是最一般的类（例如 Object），而 C1 是最具体的类（即 o 的具体类型）。

当对象 o 调用方法 m 时，Java虚拟机（JVM）会按照继承层次结构从 C1 开始向上搜索方法 m 的实现。

搜索顺序是从最具体的类（C1）到最一般的类（Cn，例如 Object）。

一旦找到第一个匹配的方法实现，搜索停止，并调用该方法。

比如前面说的这个例子。

public class PolymorphismDemo {

public static void main(String[] args) {

m(new GraduateStudent());

m(new Student());

m(new Person());

m(new Object());

}

public static void m(Object x) {

System.out.println(x.toString());

}

}

class GraduateStudent extends Student {

// ...

}

class Student extends Person {

public String toString() {

return "Student";

}

}

class Person extends Object {

public String toString() {

return "Person";

}

}

当 m(new GraduateStudent()) 被调用时，GraduateStudent 对象的实际类型是 GraduateStudent，但由于没有自己的 toString() 方法，它将使用 Student 类的 toString() 方法，输出 “Student”。

当 m(new Student()) 被调用时，Student 对象的实际类型是 Student，输出 “Student”。

当 m(new Person()) 被调用时，Person 对象的实际类型是 Person，输出 “Person”。

当 m(new Object()) 被调用时，Object 对象的实际类型是 Object，输出 “java.lang.Object@12345678”（Object 类的默认 toString() 方法返回值）。

3.3.4.1 隐式转换（Implicit Casting）

在特定情况下，Java编译器会自动将一种类型的值转换为另一种类型，而不需要开发者显式地编写转换代码。这种转换通常发生在基本数据类型（primitive types）之间，以及对象和它们的包装类（wrapper classes）之间。

在这里是当一个子类的对象被赋值给父类类型的变量时，会自动进行类型转换。这种转换是隐式的，不需要显式地进行类型转换。

因此向上转型（Upcasting）都是安全的，因此都是隐式的。

例如前面例子中的一下代码。

m(new Student());

等价于下面代码

Object o = new Student(); // 隐式转换

m(o);

Object o = new Student(); 将 Student 类型的对象赋值给 Object 类型的变量 o，这时发生了隐式转换，因为 Student 是 Object 的子类，所以 Student 类的对象可以被视为 Object 类型。

下面的代码是常见的隐式转换。

int i = 1;

double a = i; // 隐式转换：int 到 double

3.3.4.2 显式转换（Explicit Casting）

显式类型转换是开发者明确指示编译器进行特定类型转换的操作。这通常在需要向下转型（Downcasting）时使用，即从父类类型转换为子类类型。

但是需要注意的是向下转型（Downcasting）并不总是安全的。向下转型需要显式进行，并且只有在确信对象是目标类型时才应该进行。

例如前面的代码左右颠倒就无法运行，如下所示。

Student b = o;

这里会有编译错误。因为如果 o 的类型不是 Student 或 Student 的子类，那么尝试将 o 赋值给 Student 类型的变量 b 将导致编译失败。

Java 是一种静态类型语言（变量类型声明后，其类型在整个程序的生命周期内都是固定的，相反的非静态类型语言比如Python里一个变量的类型就是可以改变的），编译器在编译时需要知道所有变量和对象的确切类型。如果 o 可能不是 Student 类型，直接赋值给 Student 类型的变量可能导致类型不匹配的问题。

当你需要将一个对象从父类类型转换为子类类型时，你必须使用显式类型转换来告诉编译器对象的实际类型。这是因为编译器在编译时只知道变量的静态类型，而不知道对象的实际运行时类型。

显式类型转换使用圆括号 () 语法，其中包含目标类型和要转换的对象。

因此上面的代码正确的命令如下。

Student b = (Student)o;

这行代码显式地告诉编译器 o 实际上是 Student 类型的对象。即使 o 被声明为 Object 类型，编译器现在也知道 o 是 Student 类型，因此可以安全地将 o 赋值给 Student 类型的变量 b。

这种转换是必要的，因为如果 o 不是 Student 类型，直接赋值将导致运行时错误（ClassCastException）。

这与基础数据类型的转换相似，如下。

int i = (int)1.23;

这行代码将 double 类型的值 1.23 显式转换为 int 类型。这种转换是必要的，因为 int 类型不能直接接受 double 类型的值。

这种转换确保了类型安全，防止了可能的数据精度丢失或范围问题。

由于向下转型可能导致类型不匹配和运行时错误。在进行向下转型之前，通常使用 instanceof 操作符来检查对象的实际类型，以确保转型的安全性。

instanceof是一个二元操作符，用于检查对象是否是某个类或其子类的实例。如果检查结果为true，说明对象是该类的实例；否则为false。

示例如下。

Object myObject = new Student();

...

if (myObject instanceof Student) {

System.out.println("The student GPA is " + ((Student)myObject).getGPA());

}

这里先创建了一个Student对象，并将其赋值给Object类型的变量myObject。然后使用instanceof操作符检查myObject是否是Student类的实例。如果是，就将myObject强制转换为Student类型，并调用getGPA()方法获取学生的GPA值，然后打印出来。

下面给出了一个前面圆形和长方形类的例子。

public class CastingDemo{

public static void main(String[] args){

Object object1 = new Circle(1);

Object object2 = new Rectangle(1, 1);

displayObject(object1);

displayObject(object2);

}

public static void displayObject(Object object) {

if (object instanceof Circle) {

System.out.println("The circle radius is " + ((Circle)object).getRadius());

System.out.println("The circle diameter is " + ((Circle)object).getDiameter());

} else if (object instanceof Rectangle) {

System.out.println("The rectangle width is " + ((Rectangle)object).getWidth());

}

}

}

在 instanceof 操作符的帮助下判断对象的类型，从而进行类型转换以访问特定类的方法，这是一种安全的方法，可以避免类型转换错误



3.3.3 equals()方法

equals()方法用于比较两个对象的内容是否相等。在Java中，Object类提供了equals()方法的默认实现，该实现仅比较两个对象的引用是否相同（即是否是同一个对象）。

默认实现如下。

public boolean equals(Object obj) {

return (this == obj);

}

这里只是简单地比较参数对象是否和this对象是一个对象。

因此下面的代码展示了Circle类方法重写的equals()方法，这个方法比较了两个Circle对象的半径是否相等。

public boolean equals(Object o) {

if (o instanceof Circle) {

return radius == ((Circle)o).radius;

} else {

return false;

}

}

如果o是Circle类的实例，那么将其强制转换为Circle类型，并比较当前对象的半径（radius）和传入对象的半径是否相等。如果半径相等，返回true；否则，返回false。如果传入的对象不是Circle类的实例，直接返回false。

3.5 泛型编程（Generic programming）

泛型编程（Generic programming）是一种编程范式，它允许编写出可以对不同类型的数据进行操作的代码，而不需要指定具体的数据类型。

我们现在对前面的知识进行一个总结，多态（Polymorphism）允许使用一个接口来处理不同类型的对象。在Java等语言中，多态通常通过继承和方法重写来实现。

多态包含两部分：

1.方法的多态性：如果一个方法的参数类型被声明为一个超类（例如Object），那么你可以向这个方法传递任何该超类的子类对象（例如Student或String）。

2.动态绑定：在运行时，会根据实际传递的对象类型来决定调用哪个具体的方法实现。这意味着同一个方法调用可以有不同的行为，这取决于调用它的对象的实际类型。

而泛型编程是多态的一种扩展，它允许方法或类在定义时不指定具体的数据类型，而是在调用时指定。这样可以编写出更加通用和可重用的代码。

定义泛型方法或类时，可以使用类型参数（例如）来表示方法或类可以处理的任何类型。泛型编程还提供了类型安全泛型提供了编译时类型检查，这意味着如果代码中存在类型不匹配的问题，编译器会在编译时就报错，而不是在运行时。

在数据结构中，泛型编程和多态的结合非常有用，因为它们允许你创建可以处理任何类型元素的通用数据结构，如列表、集合、映射等。

其的优势有以下三点：

1.通用数据结构：例如，你可以创建一个泛型列表List<T>，它可以存储任何类型的元素，如List<Student>或List<String>。

2.类型安全：泛型列表在编译时检查元素类型，确保你只能添加和检索正确类型的元素，从而避免类型转换错误。

3.代码重用：泛型数据结构可以在不同的上下文中重复使用，而不需要为每种数据类型编写专门的代码。

3.6 ArrayList类

我们前面说array时，我们说array的大小是固定的，那有没有没有固定大小的数组呢？有的，下面介绍的ArrayList类正是一个可变的数组，可以储存任意数量的对象，它同时也是前面说的泛型编程的一个体现，java.util.ArrayList<T>，其中<T>是一个类型参数，可以在创建ArrayList实例时指定具体的类型。

例如，你可以创建一个ArrayList来存储String类型的对象，代码如下。

ArrayList<String> stringList = new ArrayList<String>();

public class TestArrayList {

public static void main(String[] args) {

// Generics: eliminates warnings

java.util.ArrayList<String> citylist = new java.util.ArrayList<String>();

citylist.add("London"); citylist.add("New York"); citylist.add("Paris");

citylist.add("Toronto"); citylist.add("Hong Kong");

System.out.println("List size? " + citylist.size());

System.out.println("Is Toronto in the list? " +

citylist.contains("Toronto"));

System.out.println("The location of New York in the list? " +

citylist.indexOf("New York"));

System.out.println("Is the list empty? " + citylist.isEmpty()); // false

citylist.add(2, "Beijing");

citylist.remove("Toronto");

for (int i = 0; i < citylist.size(); i++)

System.out.print(citylist.get(i) + " ");

System.out.println();

// Create a list to store two circles

java.util.ArrayList<Circle> list = new java.util.ArrayList<Circle>();

list.add(new Circle(2));

list.add(new Circle(3));

System.out.println(list.get(0).getArea()); // no casting needed

}

}

使用泛型确保了列表中只能存储指定类型的元素（String和Circle）尝试添加其他类型的元素会导致编译错误，从而在编译时就避免了潜在的问题。



3.7 访问修饰符

我们再复习一遍访问修饰符。

1.private：

这是最严格的访问级别。

private成员只能被定义它们的类本身访问。

不能在同一个包中的其他类或子类中访问，除非通过公共或受保护的方法。

2.默认（default）：

当没有指定访问修饰符时，成员具有包级访问权限（也称为默认访问级别）。

默认情况下，成员可以被同一个包中的任何其他类访问，但不能被子类访问，除非子类也在同一个包中。

3.protected：

protected成员可以被定义它们的类本身、同一个包中的任何其他类以及不同包中的子类访问。

这意味着即使子类不在同一个包中，它们也可以访问protected成员。

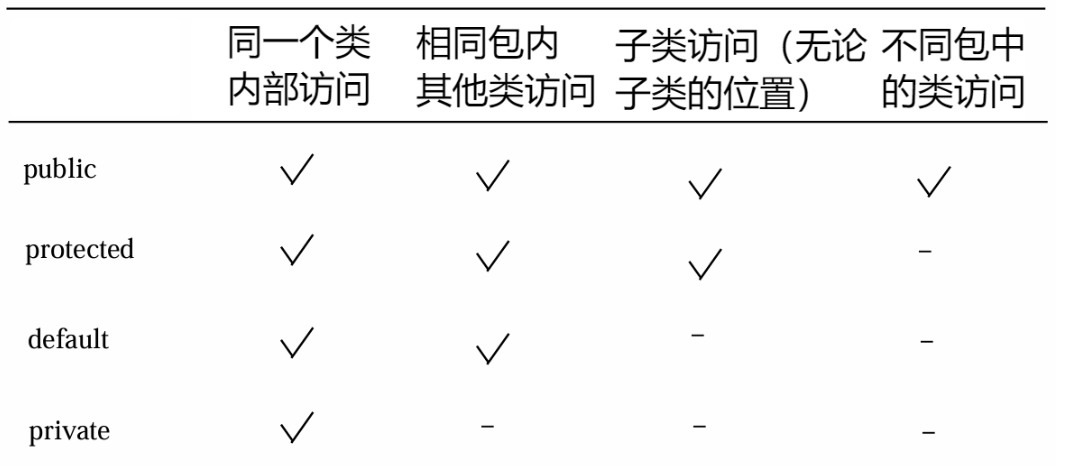
4.public：

这是最宽松的访问级别。

public成员可以被任何其他类访问，无论它们是否在同一个包中，也无论它们是否是子类。

因此可见性顺序从最严格到最宽松的顺序是：private < 默认 < protected < public。

区别如下图所示。



关于方法重写（Override）和访问修饰符有一个规则：子类在覆盖父类中的方法时，不能降低该方法的访问级别。

但是子类可以将提高该方法的访问级别。

如果父类中的一个方法是受保护的（protected），子类在覆盖这个方法时，可以将其访问级别改为public。这是因为public的访问级别比protected更宽松，允许更多的代码访问这个方法。

注意这里private无法被子类提高访问等级，因为子类不能直接访问这个方法，更不能重写（Override）它或改变其访问修饰符。这条规则的核心是，子类在覆盖父类中的方法时，不能将访问级别设置得比父类中的方法更严格。换句话说，子类不能“削弱”方法的访问权限。

例如，如果一个方法在父类中被定义为public，那么在子类中覆盖这个方法时，它也必须是public。子类不能将其改为protected或private。

这个规则的目的是为了保持或增加方法的可见性，以确保子类不会意外地限制对这些方法的访问。这有助于维护代码的兼容性和可预测性。

通过不允许降低访问级别，可以确保任何依赖于父类方法访问级别的代码，在切换到子类时仍然能够正常工作。

3.7.1 方法重写（Override）的规则

1.可访问性：

只有可访问的实例方法才能被子类覆盖。这意味着子类只能覆盖父类中具有default或public或protected访问级别的方法。

2.私有方法：

private方法不能被子类覆盖，因为它在定义它的类外部是不可见的。子类无法访问父类中的private方法，因此也就无法覆盖它。

3.完全不相关的私有方法：

如果子类中的一个方法在父类中被定义为private，那么这两个方法实际上是完全不相关的。子类中的方法并不是对父类中方法的覆盖，而是一个新的方法。

3.7.2 静态（static）方法的规则

1.继承：

静态方法可以被子类继承。这意味着子类可以使用父类中的静态方法。

2.不能覆盖：

静态方法不能被子类覆盖。这是因为静态方法属于类本身，而不是类的实例。因此，子类不能提供一个与父类中静态方法具有相同签名的新实现。

3.隐藏（Hide）：

如果子类中重新定义了一个与父类中静态方法具有相同名称和参数的静态方法，那么父类中的静态方法会被隐藏。这意味着在子类中调用该方法时，将执行子类中的定义，而不是父类中的静态方法。

3.7.3 final修饰符

当一个变量被声明为final，意味着这个变量是一个常量，它的值在初始化后不能被改变。

例如：final static double PI = 3.14159;声明了一个名为PI的常量，它是一个静态的final变量，表示圆周率π的值，并且这个值在程序运行期间是不可更改的。

当一个方法被声明为final，意味着这个方法不能被子类覆盖（override）。

子类不能提供覆盖该方法的新实现，因此final方法确保了方法的行为在所有子类中都是一致的。

当一个类被声明为final，意味着这个类不能被其他类继承（extend）。

没有其他类可以成为final类的子类，因此final类限制了类的扩展，确保了类的设计不会被其他开发者修改。



下面说法中错误的是哪个？

Abstract classes can have constructors, but cannot be instantiated

All methods in an abstract class must be abstract

Abstract methods do not have a body and must be implemented by subclasses

Abstract methods must be declared inside an abstract class

A subclass that does not implement/override all abstract methods from its abstract superclass must be declared abstract

错误的是2，抽象类中的方法不必须都是抽象的，它们可以有具体实现。

抽象类可以有构造函数，但是不能被实例化。

抽象方法没有方法体，必须由子类实现。

抽象方法必须在抽象类中声明。

如果子类没有实现/覆盖其抽象超类中的所有抽象方法，则该子类也必须声明为抽象

Override-overload



下列关于equals()方法说法错误的是哪一个？

Method signature is public boolean equals(Object obj)

Returns true if obj is not null

Checks if this and obj refer to the same object

Uses instanceof to check if obj is of the same type

Compares each field relevant to equality after casting obj

错误的是2，因为equals()方法的目的是判断两个对象是否相等，即它们的内容是否相同，不是null完全不足以说明。

这是equals()方法的正确签名。

这是Object类中equals()方法的默认行为，但通常在重写时会进行更深入的内容比较。

在使用instanceof检查之后，通常会将对象转换为适当的类型，并比较内容。

这是equals()方法重写时的常见做法，即在确认对象类型相同后，比较每个与等价性相关的字段

三！！！！！！！！！

抽象方法是定义在抽象类中的，它们不能存在于普通类中。

它只包含方法签名，没有方法体。它使用 abstract 关键字声明。

在一个非抽象的子类中，如果它继承自一个抽象的父类，那么它必须实现父类中的所有抽象方法，也就是前面的Circle类和Rectangle类必须实现getArea()和getPerimeter()方法，这样可以保证多态性的一致性。

因此对于这样的代码。

abstract class A {

abstract void m();

}

class B extends A {

void m() {

// 方法实现

}}

class B 就必须提供抽象方法 m() 的具体实现。

1.2 抽象类的抽象子类

一个抽象子类继承自另一个抽象超类时，这个子类既可以实现从抽象超类继承的所有抽象方法，也可以将实现抽象方法的责任推迟到非抽象子类。

代码如下。

abstract class A {

abstract void m();

}

abstract class B extends A {

void m() {

// 方法实现

}

}

class C extends B {

// 继承 B 的实现，不需要再次实现 m()

}

第二种情况。代码如下。

abstract class A {

abstract void m();

}

abstract class B extends A {

// 没有实现 m() 方法，所以 B 也必须是抽象类

}

public class C extends B {

void m() {

// 提供了 m() 方法的具体实现

}

}

1.3 非抽象父类的子类可以是抽象类

子类可以是抽象的，即使其超类是具体的。

例如，Object 类是具体的，但子类 GeometricObject 是抽象的。

同理，我们其实子类可以覆盖具体超类中的方法，将其定义为抽象方法。这样做有两个目的：

1.可以让其强制其的子类实现该方法。

2.将超累中的方法实现在子类中无效。如果超类中的方法实现对于子类来说不适用或无效，子类可以通过将其重新定义为抽象方法来避免使用超类的实现。

1.4 抽象类

我们现在回到抽象类上，首先我们需要弄清楚一点：我们可以定义不包含抽象方法的抽象类，但可以包含具体方法和属性。

这种抽象类用作定义新子类的基础类，通过继承这种抽象类，子类可以继承基类中的属性和方法，同时也可以添加或修改自己的属性和方法。

抽象类不能直接被实例化，即不能使用 new 关键字来创建抽象类的对象。

所以下面的代码会导致编译错误。

GeometricObject o = new GeometricObject();

尽管抽象类不能被直接实例化，但仍然可以在抽象类中定义构造函数。这些构造函数在子类的构造过程中被调用，这个过程称为构造函数链（constructor chaining）。

例如，GeometricObject 的构造函数被 Circle 和 Rectangle 类的构造函数调用。

1.4.1 抽象类可以作为数据类型

在声明变量时，可以使用抽象类作为数据类型。例如，可以声明一个 GeometricObject 类型的变量 c，然后将其初始化为 Circle 类型的对象。

GeometricObject c = new Circle(2);

在这个例子中，变量 c 被声明为 GeometricObject 类型，但它引用的是一个 Circle 对象。

因此我们可以创建一个数组，其元素类型为抽象类。例如，可以创建一个 GeometricObject 类型的数组 geo，该数组可以包含多个几何对象。

GeometricObject[] geo = new GeometricObject[10];

在数组的元素被具体对象初始化之前，它们都是 null。直到将它们指向具体的对象，如下所示。

geo[0] = new Circle();

geo[1] = new Rectangle();

2. 接口（Interface）

我们现在说接口，接口与前面的抽象类和抽象方法的组合有些类似，但是其有一些不同。

首先，接口是一种类似于类的结构，但它只包含抽象方法和常量（在 Java 8 及以后版本中，接口也可以包含默认方法和静态方法）。

接口的主要目的是为对象指定行为。通过定义一组方法，接口可以描述一个对象应该能够执行哪些操作。

接口可以用来定义对象的契约或能力，例如，可以定义一个接口来指定对象是否可比较（Comparable）、可食用（Edible）、可克隆（Cloneable）等。

虽然接口和抽象类都可以包含抽象方法，但接口的意图更侧重于定义对象的行为和能力，而不是提供部分实现。

与抽象类不同，一个类可以实现多个接口。这种特性称为多重继承，它允许类继承多个接口中定义的行为。这种灵活性使得接口成为定义行为规范的强大工具，因为一个类可以同时遵守多个接口的规范。

所以接口的主要作用更多是规定其的子类需要遵守什么样的规范，提供一个通用的实现框架。

因此其的代码框架如下。

public interface InterfaceName {

// constant declarations;

// method signatures;

}

下面给出一个例子，我们先规定一个可食用的接口，这个接口会规定一个 hotToEat() 的抽象方法。

public interface Edible {

public abstract String howToEat();

}

所以当有子类实现这个接口的时候就需要提供 howToEat() 方法的具体实现，返回一个描述如何食用这个子类的字符串。

class Chicken extends Animal implements Edible {

public String howToEat() {

return "Chicken: Fry it";

}

}

下面展示了一个较为复杂的例子。

interface Edible {

public abstract String howToEat();

}

abstract class Animal { }

class Chicken extends Animal implements Edible {

public String howToEat() {

return "Chicken: Fry it";

}

}

class Tiger extends Animal {

// Does not extend Edible

}

abstract class Fruit implements Edible { }

class Apple extends Fruit {

public String howToEat() {

return "Apple: Make apple cider";

}

}

class Orange extends Fruit {

public String howToEat() {

return "Orange: Make orange juice";

}

}

public class TestEdible {

public static void main(String[] args) {

Object[] objects = {new Tiger(), new Chicken(), new Apple()};

for (int i = 0; i < objects.length; i++) {

if (objects[i] instanceof Edible) {

System.out.println(((Edible) objects[i]).howToEat());

}

}

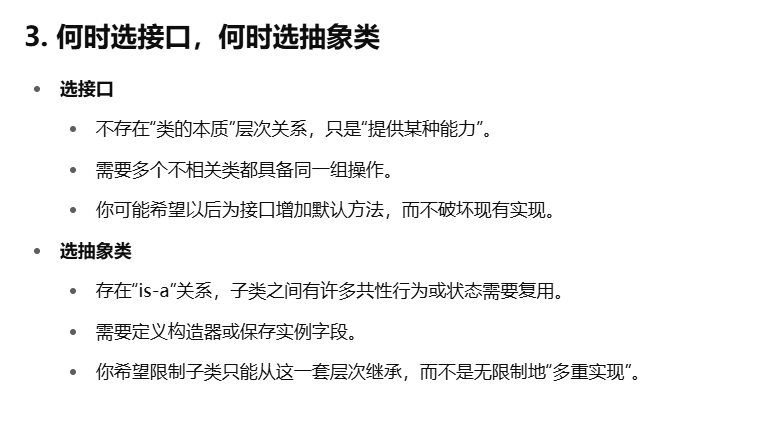
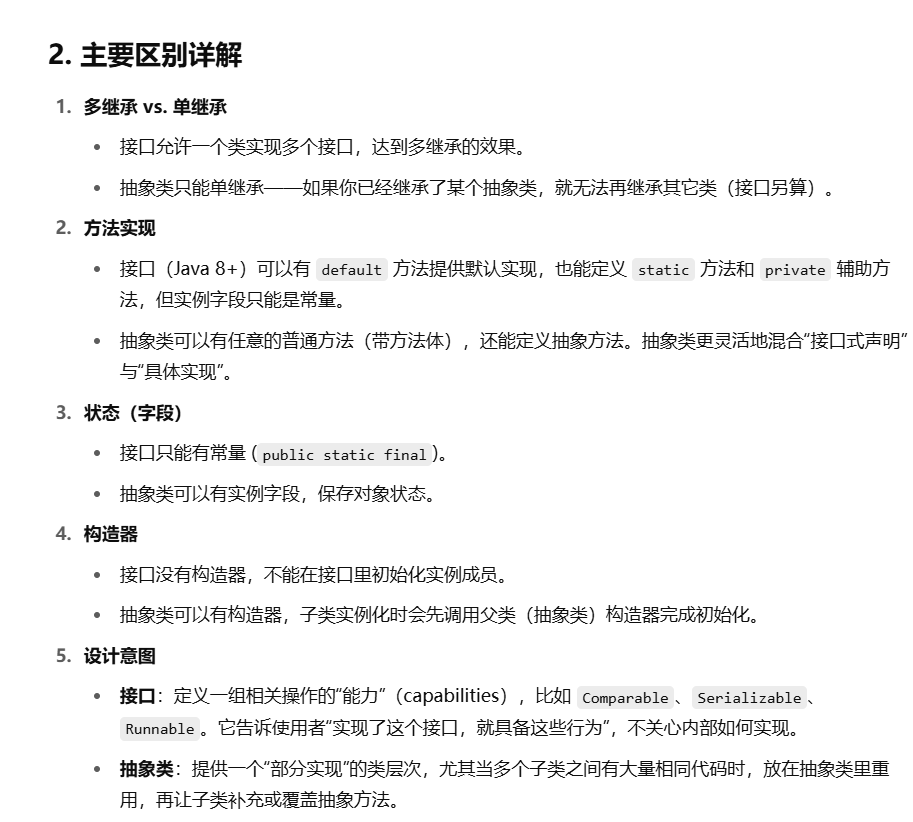
}

}

这里和前面类似，当一个抽象子类实现了一个接口时，它既可以具体实现继承的抽象方法，也可以将实现的责任交给自己的子类。

这里 Fruit 是一个抽象类，它实现了 Edible 接口。Fruit 类没有具体实现 howToEat() 方法，而是将实现的责任交给了它的子类 Apple 和 Orange。

这里还使用了 instanceof 来检查一个对象是否实现了特定的接口。



小结

接口 强调“能做什么”，适合定义契约和能力；支持多继承，但只能有常量与 default/static 方法。

抽象类 强调“是什么”，适合抽取子类间的公共状态和行为；支持实例字段、构造器和任意方法实现，但单继承。

2.1 可省略的修饰符

接口中的所有数据字段默认是 public、static 和 final 的。

接口中的所有方法默认是 public 和 abstract 的。

因此对于这些默认的修饰符，我们都是可以省略的。

因此如下所示。

public interface T1 {

public static final int K = 1;

public abstract void p();

}

这个代码等价于下面的代码。

public interface T1 {

int K = 1;

void p();

}

接口中定义的常量（即 public static final 类型的变量）可以通过 InterfaceName.CONSTANT\_NAME 的方式访问。

所以这里可以通过 T1.K 访问 T1 接口的常量 K。

2.2 接口的特征

在 Java 中，接口被当作一种特殊的类来处理。

每个接口在编译后都会生成一个独立的字节码文件（.class 文件），就像普通类一样。

接口不能实例化，类似于抽象类，不能使用 new 关键字来创建接口的实例。

接口可以像抽象类一样用作变量的数据类型。

所以接口也可以作为类型转换（casting）的结果。

如下面代码所示。

public class TestInterfaces {

public static void main(String[] args) {

Edible edible;

// 使用接口作为变量的数据类型

edible = new Apple();

edible.howToEat();

edible = new Orange();

edible.howToEat();

// 使用接口作为类型转换的结果

Object obj = new Apple();

Edible edibleObj = (Edible) obj;

edibleObj.howToEat();

}

}

2.3 Comparable 接口

Comparable 接口定义在 java.lang 包中，并且被 Arrays.sort 使用从而进行排序。

package java.lang;

public interface Comparable {

int compareTo(Object o);

}

2.3.1 实现 Comparable 接口

我们现在可以写一个通用的 max 方法，用于找出两个对象中的最大值。

有两种方案，一种是接受 Comparable 接口的对象作为参数，另一种是接受Object 类型的对象作为参数，但这种方案需要通过类型转换将 Object 类型的对象转换为 Comparable 类型，再使用 compareTo 方法进行比较。

方案一：

public class Max {

public static Comparable max(Comparable o1, Comparable o2) {

if (o1.compareTo(o2) > 0)

return o1;

else

return o2;

}

}

方案二：

public class Max {

public static Object max(Object o1, Object o2) {

if (((Comparable)o1).compareTo(o2) > 0)

return o1;

else

return o2;

}

}

需要注意的是，第二个版本仍需要保证传入的对象实际上是 Comparable 类型的，否则在运行时会抛出 ClassCastException。

还需要注意第一个方法返回的是 Comparable 类型，第二个返回的是 Object 类型，所以使用返回值时，可能需要进行类型转换以匹配调用方法时的期望类型。

就像我们前面说的，我们需要保证传入的对象实际上是 Comparable 类型的，所以我们现在无法使用 max 方法去找到两个 Rectangle 实例中较大的一个，因为 Rectangle 类没有实现 Comparable 接口。

我们可以定义一个新的 Rectangle 类，命名为 ComparableRectangle，并让它实现 Comparable 接口。这样，ComparableRectangle 类的实例就可以进行比较了。

代码如下。

public class ComparableRectangle extends Rectangle implements Comparable<ComparableRectangle> {

/\*\* Construct a ComparableRectangle with specified properties \*/

public ComparableRectangle(double width, double height) {

super(width, height);

}

/\*\* Implement the compareTo method defined in Comparable \*/

@Override

public int compareTo(object o) {

if (getArea() > ((ComparableRectangle)o).getArea())

return 1;

else if (getArea() < ((ComparableRectangle)o).getArea())

return -1;

else

return 0;

}

public static void main(String[] args) {

ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);

ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);

System.out.println(Max.max(rectangle1, rectangle2));

}

}

2.4 Cloneable 接口

Cloneable 是一个空接口，它不包含任何常量（constants）或方法（methods）。它的主要作用是作为一个标记（marker），向编译器（compiler）和 Java 虚拟机（JVM）表明实现该接口的类具有某些理想的属性。

package java.lang;

public interface Cloneable {

}

当一个类实现了 Cloneable 接口，这意味着这个类被标记为“可克隆”（cloneable）。

在 Java 中，Object 类（所有类的根类）定义了一个 clone() 方法，这个方法可以用来创建当前对象的一个副本（shallow copy）。如果一个类实现了 Cloneable 接口，那么它的实例可以使用 clone() 方法进行克隆。

当然开发者可以在自己的类中重写（override）Object 类的 clone() 方法，以提供自定义的克隆行为。

例如我们前面提到的 Calendar 类就实现了 Cloneable，这意味着 Calendar 对象可以被克隆。

Calendar calendar = new GregorianCalendar(2022, 1, 1);

Calendar calendarCopy = (Calendar)(calendar.clone());

System.out.println("calendar == calendarCopy is " + (calendar == calendarCopy));

这里需要显式转换的原因是 clone() 方法在 Object 类中被定义，所以返回的是 Object 类型，以确保兼容性。

而上面代码的结果如下。

calendar == calendarCopy is false。

因为克隆操作创建了一个新的对象，但它与原始对象在内存中有不同的引用（地址），所以这里直接比较引用的结果是false。

如果我们使用equals()方法进行比较，我们就会获得true。

System.out.println("calendar.equals(calendarCopy) is" + calendar.equals(calendarCopy));

结果如下。

calendar.equals(calendarCopy) is true

2.4.1 实现 Cloneable 接口

我们前面说 clone() 方法是在 Object 类下的，那我们如果如果尝试克隆一个没有实现 Cloneable 接口的类的对象会发生什么呢？

结果将会抛出 CloneNotSupportedException 异常，从而防止未经允许的对象被克隆。

Object 类中的 clone() 方法用于创建当前对象的一个新实例，并且初始化新对象的所有字段，使其内容与原对象的相应字段完全相同，就像通过赋值操作（使用一种称为反射的技术）一样。

需要注意的是，引用数据字段的内容不会被克隆，即如果对象中包含对其他对象的引用，克隆后的新对象将与原对象共享这些引用。

再次强调一下clone() 方法返回一个 Object 类型的对象，因此使用返回值时需要注意类型转换。

我们可以通过重写 Object 类的 clone() 方法来创建自定义的克隆行为。

现在给出一些实现 Cloneable 的例子。

public class SomethingCloneable implements Cloneable {

public boolean equals(Object o) {

return true;

}

public static void main(String[] args) throws CloneNotSupportedException {

SomethingCloneable s1 = new SomethingCloneable();

SomethingCloneable s2 = (SomethingCloneable) s1.clone();

System.out.println("s1 == s2 is " + (s1 == s2));

// 应该输出 false，因为 s1 和 s2 是不同的对象引用

System.out.println("s1.equals(s2) is " + s1.equals(s2));

// 应该输出 true，因为 equals 方法总是返回 true

}

}

这里或许可能会想 SomethingCloneable 没有继承 Object 类，怎么能使用 clone() 方法的？

在 Java 中，所有的类都隐式地继承自 Object 类，除非显式地继承自另一个类，所以能使用 clone() 方法。

public class House implements Cloneable, Comparable<House> {

private int id;

private double area;

private java.util.Date whenBuilt;

public House(int id, double area) {

this.id = id;

this.area = area;

this.whenBuilt = new java.util.Date();

}

public int getId() {

return id;

}

public double getArea() {

return area;

}

public java.util.Date getWhenBuilt() {

return whenBuilt;

}

// Override the clone method defined in the Object class

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException ex) {

return null;

}

}

// Implement the compareTo method defined in Comparable

public int compareTo(Object o) {

if (this.area > ((House)o).area)

return 1;

else if (this.area < ((House)o).area)

return -1;

else

return 0;

}

}

这里的 House 类就实现了两个接口，并且重写了来自 Object 类的 clone() 方法。

2.4.2 浅拷贝（Shallow Copy）和深拷贝（Deep Copy）

对于前面的代码，如果我们使用 clone() 方法。

House house1 = new House(1, 1750.50);

House house2 = (House)(house1.clone());

由于这里 whenBuilt 是引用类型，所以这里克隆直接复制引用本身，而不复制引用的对象，所以现在 house1 和 house2 共享同一个 Date 对象， 因此如果修改 house 1 的 whenBuilt 字段的任何修改都会影响 house2 的 whenBuilt 字段，反之亦然。

这就是浅拷贝（Shallow Copy），如果字段是引用类型（如 Date 对象），则只复制引用本身，而不复制引用的对象。

而在在深拷贝（Deep Copy）中，引用类型字段会被递归地复制。这意味着不仅复制引用，还复制引用的对象。

这样，house1 和 house2 将拥有独立的 Date 对象，修改一个不会影响到另一个。

我们将 House 类里重写的 clone 方法中加入对引用类型的克隆。

@Override

public Object clone() {

try {

House h = (House) super.clone(); // Perform a shallow copy

h.whenBuilt = (Date) whenBuilt.clone(); // Perform a deep copy for Date object

return h;

} catch (CloneNotSupportedException ex) {

return null;

}

}

2.4.3 接口和抽象类的对比

2.4.3.1 组成部分

变量（Variables）：

接口：接口中的所有变量都必须是 public、static 和 final 的常量。这意味着接口中定义的变量是全局常量，不能被修改。

抽象类：抽象类可以有普通的变量，即实例变量或类变量，这些变量可以被修改。

构造函数（Constructors）：

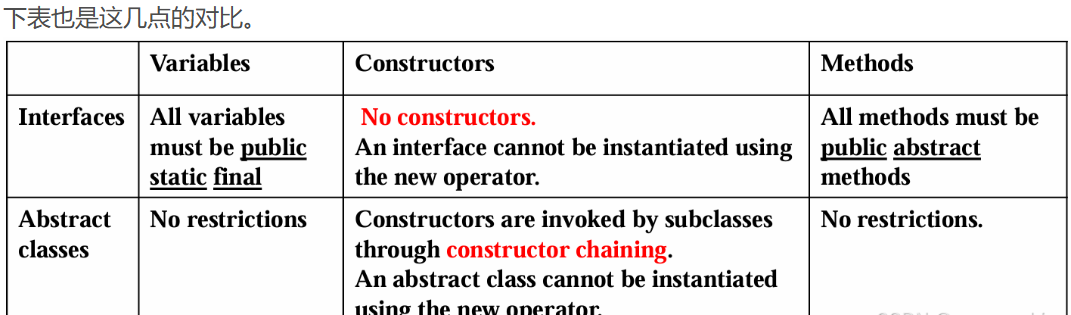
接口：接口没有构造函数。由于接口不能被实例化，它不需要构造函数。

抽象类：抽象类有构造函数，这些构造函数通过子类构造链（constructor chaining）被子类调用。尽管抽象类不能直接实例化，但子类可以创建抽象类的实例。

方法（Methods）：

接口：接口中的每个方法都必须是 public 和 abstract 的，即它们只定义了方法签名，没有实现。接口不能包含具体方法的实现。

抽象类：抽象类可以包含具体方法（即有实现的方法）。抽象类中的方法可以是 public、protected 或 private，并且可以包含方法体（实现）。



2.4.3.2 继承（Inheritance）

在继承（Inheritance）上，接口又有以下几种特性：

接口可以扩展任意数量的其他接口：

一个接口可以继承（或称为“扩展”）任意数量的其他接口。这意味着接口可以组合多个接口中定义的行为，从而为实现类提供更多的功能。

接口没有根：

与类继承体系不同，接口没有根接口。类继承体系中，所有类最终都继承自 Object 类，但接口可以独立存在，不需要继承自其他接口。

类可以实现任意数量的接口：

一个类可以实现（或称为“实现”）任意数量的接口。这允许类具有多个接口定义的行为，从而支持多重继承，这是 Java 实现多重继承的一种方式。

如图所示。

由于第三点所以当一个类实现了两个或多个接口，而这些接口中存在冲突的信息时，就会发生接口冲突。

突的信息可能包括：

两个相同的常量（constants）具有不同的值。例如，两个接口都定义了一个名为 MAX\_VALUE 的常量，但赋予了不同的数值。

两个具有相同签名（signature）但返回类型（return type）不同的方法。例如，两个接口都有一个名为 getValue 的方法，但一个返回 int 类型，另一个返回 double 类型。

为了解决这种冲突，类需要明确指定它将使用哪个接口的常量或方法。例如，类可以重新定义一个新的常量或方法来解决冲突，如下所示。

public class MyClass implements InterfaceA, InterfaceB {

// 使用 InterfaceA 的 MAX\_VALUE

int valueA = InterfaceA.MAX\_VALUE;

// 使用 InterfaceB 的 MAX\_VALUE

int valueB = InterfaceB.MAX\_VALUE;

// 实现两个接口的方法

public int getValueA() {

return InterfaceA.MAX\_VALUE;

}

public double getValueB() {

return InterfaceB.MAX\_VALUE;

}

}

2.4.3.3 什么时候该使用类/接口呢？

强关系（Strong）：

当存在一个清晰的“是一个”（is-a）关系时，应该使用类继承（class inheritance）。

这种关系描述了一种父子关系，例如，一个学生是一个人，这表示学生继承自人。

在这种情况下，类继承是合适的，因为它允许子类继承父类的所有属性和方法，并可以添加或修改这些属性和方法。

弱关系（Weak）：

当存在一种“具有”（has-a 或 is-kind-of）关系时，表明对象具有某种属性或行为。

例如，所有的字符串（String）都是可比较的（Comparable），所有的日期（Date）也都是可比较的。

在这种情况下，接口是合适的选择，因为接口允许类声明它们实现了某种行为或属性，而不需要继承自其他类。

使用接口来绕过单继承限制：

在 Java 中，类不能直接继承自多个类（即 Java 不支持多重继承），但可以实现多个接口。

接口允许类实现多个行为或属性，从而绕过 Java 的单继承限制，实现类似多重继承的效果。所以我们可以使用接口来绕过单继承限制

3. 包装类（Wrapper Classes）

包装类（Wrapper Classes）是用于将基本数据类型（primitive data types）封装为对象。

Java 中的基本数据类型（如 int, double, boolean 等）不是对象，它们是存储在栈上的简单值。因此它们提供更好的性能，因为它们直接存储值，没有对象开销。

包装类是这些基本数据类型的类版本，它们是对象，可以存储在堆上。比如数据结构（如 ArrayList）期望对象作为元素，而不是直接存储基本数据类型的值，我们就可以使用包装类。

包装类包括：Boolean, Character, Short, Byte, Integer, Long, Float, Double。

从这里我们发现这些类还都实现了 Comparable 接口，所以它们的实例可以通过 compareTo 方法进行比较。

而且包装类的对象一旦创建，其值就不能被改变。这种不可变性使得包装类的对象在多线程环境中更安全，因为它们的状态不会意外地被修改。

每个包装类还都重写了 Object 类中定义的 toString 和 equals 方法。

3.1 Number 类

每个数值包装类（如 Integer、Double、Float 等）都继承自抽象的 Number 类。

Number 类是一个抽象类，它定义了一组方法，这些方法用于将包装类的对象转换为对应的基本数据类型值。

Number 类包含一组方法，如 doubleValue()、floatValue()、intValue()、longValue()、shortValue() 和 byteValue()，这些方法的目的是将对象转换为基本数据类型值。

这些方法中，doubleValue()、floatValue()、intValue() 和 longValue() 是抽象方法，需要在每个具体的包装类中实现。

byteValue() 和 shortValue() 方法不是抽象的，它们直接返回对象值的 byte 和 short 类型，即 (byte)intValue() 和 (short)intValue()。

每个具体的数值包装类（如 Integer、Double、Float 等）都实现了 Number 类中的抽象方法。

3.4 静态方法和解析方法

数值包装类（如 Double 和 Integer）提供了一个名为 valueOf 的静态方法，该方法用于根据指定的字符串创建一个新的对象，并将其初始化为字符串所表示的数值。

例子如下。

Double doubleObject = Double.valueOf("12.4");

Integer integerObject = Integer.valueOf("12");

通过静态方法我们能将数值字符串创建为对象，要是想获得对应的数据类型，我们可以使用解析方法，将数值字符串解析为相应的数值类型。

例子如下。

double d = Double.parseDouble("12.4");

int i = Integer.parseInt("12");

3.5 自动装箱（boxing）和拆箱（unboxing）

自 Java 1.5 版本起，Java 允许基本数据类型（primitive types）和包装类类型（wrapper class types）之间自动转换。

装箱（boxing）：当需要对象时，基本数据类型会自动转换为包装类型。

拆箱（unboxing）：当需要基本数据类型时，包装类型会自动转换为基本数据类型。

示例如下。

public class BoxingUnboxingExample {

public static void main(String[] args) {

// 装箱：基本数据类型转换为包装类类型

Integer[] intArray = {2, 4, 3};

System.out.println("intArray: " + java.util.Arrays.toString(intArray));

// 拆箱：包装类类型转换为基本数据类型

int n = intArray[0] + intArray[1] + intArray[2];

System.out.println("Sum: " + n);

}

}

Arrays 对象

在 Java 中，数组是对象，它们是 Object 类的实例。

我们可以使用 instanceof 来验证这一点。

new int[10] instanceof Object

结果是 true，所以 数组的确是 Object 类的实例。

如果 A 是 B 的子类，那么 A[] 的每个实例也是 B[] 的实例。

new GregorianCalendar[10] instanceof Calendar[]

new Calendar[10] instanceof Object[]

new Calendar[10] instanceof Object

这里的结果都是 true。

尽管我们可以将 int 数值直接赋值给 double 类型变量， 但是 int[] 和 double[] 是两种不兼容的类型，因为它们不是类。

所以我们无法将 int[] 数组赋值给 double[]，如下面的代码会造成编译错误。

double[] a = new int[10];

4.1 排序方法

java.util.Arrays 类提供了一个静态方法 sort，用于对对象数组进行排序。

这个方法接受一个对象数组作为参数，并使用 Comparable 接口来确定排序顺序。

下面的代码定义了一个 sort() 方法实现排序， 当然也可以直接使用 Arrays.sort() 方法进行排序。

public class GenericSort {

public static void main(String[] args) {

Integer[] intArray = {new Integer(2), new Integer(4), new Integer(3)};

sort(intArray); // 或者使用 Arrays.sort(intArray);

printList(intArray);

}

public static void sort(Object[] list) {

Object currentMax;

int currentMaxIndex;

for (int i = list.length - 1; i >= 1; i--) {

currentMax = list[i];

currentMaxIndex = i;

for (int j = i - 1; j >= 0; j--) {

if (((Comparable)currentMax).compareTo(list[j]) < 0) {

currentMax = list[j];

currentMaxIndex = j;

}

}

}

list[currentMaxIndex] = list[i];

list[i] = currentMax;

}

public static void printList(Object[] list) {

for (int i = 0; i < list.length; i++) {

System.out.print(list[i] + " ");

}

}

}

四！！！！

1. 泛型（Generics）

我们先看一段代码。

public class ShowUncheckedWarning {

public static void main(String[] args) {

java.util.ArrayList list = new java.util.ArrayList();

list.add("1");

Integer i = (Integer)(list.get(0)); // No Compiler Errors, but Runtime error because "1" is a String

}

}

这里的代码会报错，因为无法强行将 String 转换为 Integer，但这里是运行错误，而不是编译错误。但其实最好是出现编译错误，而不是运行错误！因为只有这样在代码部署前我们才能发现错误，不像运行错误可能会导致程序崩溃或行为异常，难以调试和修复。

我们可以使用泛型编程解决这个问题。

public class ShowCompilerError {

public static void main(String[] args) {

java.util.ArrayList<Integer> list = new java.util.ArrayList<Integer>();

// list.add("1"); // Compiler error on this line

list.add(new Integer(1)); // No error here

}

}

这里创建了一个泛型化的 ArrayList，指定了其存储的元素类型为 Integer。这意味着列表中只能存储 Integer 类型的对象。

如果尝试向列表中添加一个字符串 “1”。由于泛型指定了列表只能存储 Integer 类型的对象，因此编译器会报错，指出类型不匹配。

如果尝试列表中添加了一个 Integer 对象，符合泛型的要求，因此不会报错。

但是这里最后一行会有红色的报错，但是我们可以正常运行，这里会提示你这个语句已经不推荐使用将在未来的版本中被移除。

1.1 泛型的定义

泛型允许程序员定义类和方法时使用类型参数。这些类型参数可以在编译时被具体的类型替换，从而提供类型安全和代码重用。

假设我们定义了一个泛型栈类 Stack<T>，其中 T 是一个类型参数。使用这个泛型类，我们可以创建不同类型元素的栈对象。

1.创建一个用于存储字符串的栈。

示例如下。

Stack<String> stringStack = new Stack<>();

stringStack.push("Hello");

stringStack.push("World");

2.创建一个用于存储数字的栈。

示例如下。

Stack<Integer> integerStack = new Stack<>();

integerStack.push(1);

integerStack.push(2);

在这两个例子中，String 和 Integer 是具体的类型，它们替换了泛型类 Stack<\T> 中的类型参数 T。这样，你就可以创建专门用于存储字符串或整数的栈对象，而不需要为每种类型编写不同的类。

1.2 泛型的优点

能够在编译时检测错误，而不是在运行时。

泛型类或方法指定类或方法可以处理的对象的允许类型。

重用代码。例如，为一种特殊类型的数据结构编写单一实现，如泛型栈及其标准方法的单一实现。

使用泛型可以避免在代码中进行强制类型转换，使得代码更加清晰和易于维护。后面将阐述这一点。

当然最重要的优点是如果尝试使用不兼容的对象使用类或方法，将发生编译错误。正如前面例子展示的一样。

我们再看一个泛型的优势。

在JDK1.5之前在Comparable接口的定义中，它有一个compareTo方法，该方法接受一个Object类型的参数。

这种方式在编译时不会检查类型，因此可能会导致运行时错误（Runtime error）。

package java.lang;

public interface Comparable {

public int compareTo(Object o);

}

Comparable c = new Date();

System.out.println(c.compareTo("red"));

在JDK 1.5及以后版本中，Comparable接口被定义为泛型接口Comparable<T>。这种方式在编译时会检查类型，如果类型不匹配，编译器会报错（Compiler error），从而提高了代码的可靠性。

package java.lang;

public interface Comparable<T> {

public int compareTo(T o);

}

Comparable<Date> c = new Date();

System.out.println(c.compareTo("red"));

所以上面的代码运行的时候我们才知道错误，下面的代码我们运行之前就可以发现错误。

下图展示了JDK1.5版本之后的 ArrayList 类。

由于之前的版本，没有泛型，所以是使用Object类型来存储所有元素的，所以在从集合中取出元素时，往往需要进行类型转换，以确保取出的元素是期望的类型。

如下所示。

ArrayList list = new ArrayList();

list.add("Hello");

list.add(123);

Object obj = list.get(0); // 获取到的是Object类型

String str = (String) obj; // 需要进行类型转换

而现在就可以避免这一点。

ArrayList<String> list = new ArrayList<>();

list.add("Hello");

// list.add(123); // 这行代码会导致编译错误，因为123不是String类型

String str = list.get(0); // 直接获取String类型，无需类型转换

1.3 类型限制

泛型类型必须是引用类型（reference types），不能是基本数据类型（primitive types）。

泛型类型不能被替换为基本数据类型，如 int、double 或 char。

下列的代码会报错。

ArrayList<int> intList = new ArrayList<>(); // 这是错误的，会导致编译错误

正确做法是使用基本数据类型的包装类（wrapper types），如 Integer、Double 或 Character。

如下所示。

ArrayList<Integer> intList = new ArrayList<>();

得益于装箱（Boxing）机制，Java可以将基本数据类型自动转换为其对应的包装类。

如下所示。

intList.add(1); // Java自动将1装箱为：new Integer(1)

Java会自动将 1 装箱为 Integer 对象 new Integer(1)，然后添加到列表中。

因此当我们获取元素是，如果元素是包装类类型，不需要显式地进行类型转换。

如下所示。

ArrayList<Double> list = new ArrayList<Double>();

list.add(5.5); // 5.5自动装箱为new Double(5.5)

list.add(3.0); // 3.0自动装箱为new Double(3.0)

Double doubleObject = list.get(0); // 直接获取Double对象，无需类型转换

当然也支持拆箱（Unboxing）操作，从而将包装类自动转换为对应的基本数据类型。

如下所示。

double d = list.get(1); // 自动拆箱：自动转换为double

1.4.1 GenericStack 类

我们可以设计一个支持泛型的 Stack 类。

代码如下。

public class GenericStack<E> {

private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

public int getSize() {

return list.size();

}

public E peek() {

return list.get(getSize() - 1);

}

public void push(E o) {

list.add(o);

}

public E pop() {

E o = list.remove(getSize() - 1);

return o;

// OR just: return list.remove(getSize() - 1);

}

public boolean isEmpty() {

return list.isEmpty();

}

@Override

// Java annotation: also used at compile time to detect override errors

public String toString() {

return "stack: " + list.toString();

}

}

1.5 泛型静态方法的声明

在方法的声明中，泛型类型参数需要放在方法修饰符（如 public 和 static）之后，紧跟在方法名之前。

public class GenericMethods1 {

// 声明一个泛型静态方法

public static <E> void print(E[] list) {

for (int i = 0; i < list.length; i++) {

System.out.print(list[i] + " ");

}

System.out.println();

}

public static void main(String[] args) {

// 创建一个字符串数组

String[] s3 = {"Hello", "again"};

// 调用泛型方法，显式指定类型参数

GenericMethods1.<String>print(s3);

// 或者简单地调用泛型方法，编译器会自动推断类型参数

print(s3);

}

}

1.6 有界泛型（Bounded Generic Types）

有界泛型是指在定义泛型时，可以指定泛型类型参数的上界，即泛型类型参数必须是指定类或其子类的实例。

如前面我们经常使用的GeometricObject下面有Circle和Rectangle类。对这个例子我们有如下代码。

public class GenericMethods2 {

// 定义一个有界泛型方法，E 必须是 GeometricObject 或其子类的实例

public static <E extends GeometricObject> boolean equalArea(E object1, E object2) {

return object1.getArea() == object2.getArea();

}

public static void main(String[] args) {

Rectangle rectangle = new Rectangle(2, 2);

Circle circle = new Circle(2);

System.out.println("Same area? " + equalArea(rectangle, circle));

}

}

1.4.2 数组排序

基于上面一点我们可以有如下实践。

public class GenericSelectionSort {

public static <E extends Comparable<E>> void genericSelectionSort(E[] list) {

E currentMin;

int currentMinIndex;

for (int i = 0; i < list.length - 1; i++) {

// Find the minimum in the list[i...list.length-1]

currentMin = list[i];

currentMinIndex = i;

for (int j = i + 1; j < list.length; j++) {

if (currentMin.compareTo(list[j]) > 0) {

currentMin = list[j];

currentMinIndex = j;

}

}

// Swap list[i] with list[currentMinIndex] if necessary

if (currentMinIndex != i) {

list[currentMinIndex] = list[i];

list[i] = currentMin;

}

}

}

}

我们再看一下这个代码，其实这里的泛型类型参数 <E extends Comparable<E>>有两个含义：

E 必须是 Comparable 接口的实现者。

确保了在比较操作中使用的元素都是同一类型 E。

由于 Integer、Double、Character 和 String 等类都实现了 Comparable 接口，所以这些类的对象可以通过 compareTo 方法进行比较，因此可以使用 genericSelectionSort 方法对这些类的对象数组进行排序。

1.7 原始类型（Raw Type）和向后兼容性（Backward Compatibility）

在Java中，如果使用泛型类或接口时没有指定具体的类型参数，这种类型被称为原始类型（Raw Type）。原始类型允许Java程序与Java泛型出现之前的版本兼容，这便是向后兼容性（Backward Compatibility）。也就是说，即使在引入泛型之后的版本中，不使用泛型的旧代码仍然可以正常编译和运行。

在JDK 1.5及更高版本中，声明一个原始类型的变量大致等同于声明一个指定了Object类型参数的泛型变量。

如下面的两串代码是几乎等价的。

ArrayList list = new ArrayList();

ArrayList<Object> list = new ArrayList<Object>();

因此，所有在旧版本的JDK中编写的程序在新版本中仍然是可执行的。

当较旧的技术或程序能够无需修改或只需很少修改就能在更新的技术或系统中使用时，就称新技术具有向后兼容性。在Java的例子中，新版本通过引入原始类型的概念保持了对旧代码的兼容性。

例如Python 3.X版本在设计时没有保持与Python 2.X版本的向后兼容性，导致一些在Python 2中可以正常工作的代码在Python 3中不再适用。

在Python 2中，print 1 是有效的代码，会输出数字1。

但在Python 3中，这种语法不再有效，因为Python 3要求使用函数语法 print(1) 来进行输出。

1.7.1 原始类型的不安全性

由于之前的代码没有使用泛型，所以它们很可能会导致不安全性。

如下面的代码所示。

public class Unsafe {

// Return the maximum between two objects

public static Comparable max1(Comparable o1, Comparable o2) {

if (o1.compareTo(o2) > 0)

return o1;

else

return o2;

}

public static void main(String[] args) {

System.out.println(max1("Welcome", 23));

}

}

在这个例子中，Comparable o1 和 Comparable o2 是原始类型声明，所以这些原始类型在编译时被视为 Object 类型，而 Object 类型可以接受任何类型的对象。

但是在运行时会报错，因为字符串 “Welcome” 和整数 23 不是同一类型，不能直接进行比较。

因此我们回到前面我们所说的泛型的优点，我们利用泛型就可以解决这里的不安全性。

下面的代码修复了上面例子中的问题。

public class Safe {

// Return the maximum between two objects of the same type

public static <E extends Comparable<E>> E max2(E o1, E o2) {

if (o1.compareTo(o2) > 0)

return o1;

else

return o2;

}

public static void main(String[] args) {

// 下面的调用现在会导致编译错误，因为两个参数必须具有相同的类型

// System.out.println(max2("Welcome", 23));

// 正确的调用方式，使用相同类型的参数

System.out.println(max2("Welcome", "World"));

System.out.println(max2(1, 2));

}

}

通过这种方式，编译器会在编译时检查传递给 max2 方法的两个参数是否为同一类型，并且该类型是否实现了 Comparable 接口。从而避免了不安全性，将运行错误变成了编译错误。

1.8 通配符（Wildcards）

通配符（Wildcards）用于解决泛型中的一些类型限制问题。其可以允许我们在不知道具体类型的情况下，对一组类型进行操作。

Integer 是 Number 的子类，但是 GenericStack<Integer> 并不是 GenericStack<Number> 的子类。这意味着你不能将 GenericStack<Integer> 赋值给 GenericStack<Number> 类型的变量，即使 Integer 是 Number 的子类。

这里介绍通配符的三种类型：

无界通配符（Unbounded Wildcard）：? 表示任何类。例如，List<?> 表示一个可以包含任何类型元素的 List。

有界通配符（Bounded Wildcard）：? extends T 表示任何 T 的子类（包括 T 本身）。例如，List<? extends Number> 表示一个可以包含任何 Number 子类元素的 List。

下界通配符（Lower Bound Wildcard）：? super T 表示任何 T 的超类（包括 T 本身）。例如，List<? super Number> 表示一个可以接受任何 Number 类型及其超类型（如 Object）作为元素的 List。

所以我们可以在通配符的帮助下，让 GenericStack<Integer> 成为 GenericStack<? extends Number> 的子类，当然能这么做也是因为 Integer 是 Number 的一个子类。

这种关系允许我们能够编写更灵活的代码。

下面给出一些例子。

例1：

public class WildCardDemo1 {

/\*\*

\* It expects GenericStack<Number>

\* Find the maximum in a stack of numbers

\*/

public static double max(GenericStack<Number> stack) {

double max = stack.pop().doubleValue(); // initialize max

while (!stack.isEmpty()) {

double value = stack.pop().doubleValue();

if (value > max)

max = value;

}

return max;

}

public static void main(String[] args) {

GenericStack<Integer> intStack = new GenericStack<Integer>();

intStack.push(1); // 1 is autoboxed into new Integer(1)

intStack.push(2);

intStack.push(-2);

System.out.print("The max number is " + max(intStack));

// Compile Error: max cannot be applied to GenericStack<Integer>

}

}

在这里由于 GenericStack<Integer> 不是 GenericStack<Number> 的子类型，这将导致编译错误。

public class WildCardDemo1B {

/\*\*

\* Find the maximum in a stack of numbers

\*/

public static double max(GenericStack<? extends Number> stack) {

double max = stack.pop().doubleValue(); // initialize max

while (!stack.isEmpty()) {

double value = stack.pop().doubleValue();

if (value > max)

max = value;

}

return max;

}

public static void main(String[] args) {

GenericStack<Integer> intStack = new GenericStack<Integer>();

intStack.push(1); // 1 is autoboxed into new Integer(1)

intStack.push(2);

intStack.push(-2);

System.out.print("The max number is " + max(intStack));

}

}

在这个版本中，max 方法使用了有界通配符 <? extends Number>，这意味着它可以接受任何 Number 类型或其子类型的 GenericStack。因此，传递一个 GenericStack<Integer> 类型的参数给 max 方法是允许的，因为 Integer 是 Number 的子类。这解决了之前版本中的编译错误问题。

可以成功输出结果。

The max number is 2.0

例2：

public class WildCardDemo2 {

/\*\*

\* Print objects and empties the stack

\*/

public static void print(GenericStack<?> stack) {

while (!stack.isEmpty())

System.out.print(stack.pop() + " ");

}

public static void main(String[] args) {

GenericStack<Integer> intStack = new GenericStack<Integer>();

intStack.push(1); // 1 is autoboxed into new Integer(1)

intStack.push(2);

intStack.push(-2);

print(intStack);

}

}

结果如下。

-2 2 1

例3：

public class WildCardDemo3 {

// Add stack1 TO stack2: the type of elements in stack2 must be

// a SUPERTYPE of the type of elements in stack1

public static <T> void add(GenericStack<? extends T> stack1,

GenericStack<T> stack2) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

public static void print(GenericStack<?> stack) {

while (!stack.isEmpty()) {

System.out.print(stack.pop() + " ");

}

}

public static void main(String[] args) {

GenericStack<String> stack1 = new GenericStack<String>();

GenericStack<Object> stack2 = new GenericStack<Object>();

stack2.push("Java");

stack2.push(2);

stack1.push("Sun");

add(stack1, stack2);

print(stack2);

}

}

这里 add 方法将 stack1 中的所有元素添加到 stack2 中，要求 stack1 中元素的类型是 stack2 中元素类型的子类型或和 stack2 元素类型一致。

结果如下。

Sun 2 Java

1.9 类型擦除（Type Erasure）

类型擦除是Java实现泛型的一种机制。在这种机制下，泛型类型信息在编译时被用来检查类型安全，但在编译完成后，这些类型信息会被“擦除”，即泛型类型参数会被替换为它们的上界（对于无界泛型，上界是 Object）。

这种机制使得泛型代码能够与Java早期版本中使用的原始类型（如 ArrayList 而不是 ArrayList<String>）保持向后兼容性。

所以下面的代码是泛型代码。

ArrayList<String> list = new ArrayList<String>();

list.add("Oklahoma");

String state = list.get(0);

在编译完成后，这串代码等效于下面的代码。

ArrayList list = new ArrayList();

list.add("Oklahoma");

String state = (String)list.get(0);

下面出示更多的例子。

例1：

public static <E> void print(E[] list) {

for (int i = 0; i < list.length; i++) {

System.out.print(list[i] + " ");

}

System.out.println();

}

编译后的类型擦除后运行的版本如下。

public static void print(Object[] list) {

for (int i = 0; i < list.length; i++) {

System.out.print(list[i] + " ");

}

System.out.println();

}

例2：

public static <E extends GeometricObject> boolean equalArea(E object1, E object2) {

return object1.getArea() == object2.getArea();

}

编译后的类型擦除后运行的版本如下。

public static boolean equalArea(GeometricObject object1, GeometricObject object2) {

return object1.getArea() == object2.getArea();

}

1.9.2 泛型的限制

由于Java泛型在运行时进行类型擦除（Erasure），导致在使用泛型时存在的一些限制。

不能创建泛型类型的实例

问题：不能使用 new E() 这样的语法来创建泛型类型 E 的实例。例如，E object = new E(); 是错误的。

原因：new E() 这种实例化操作是在运行时执行的，但在运行时，泛型类型 E 的信息已经被擦除，不再可用。因此，编译器无法在运行时确定 E 的具体类型，这会导致编译错误。

不能创建泛型数组，如 new E[100]。例如，E[] elements = new E[capacity]; 是错误的。

原因：与创建泛型类型的实例类似，泛型数组的创建也是在运行时执行的。由于泛型类型信息在运行时不可用，编译器无法确定数组的具体类型，这同样会导致编译错误。

在静态方法、静态字段或初始化器中引用泛型类型参数是非法的。

原因：泛型类的所有实例在运行时共享相同的类，因此泛型类中的静态变量和方法也是共享的，而泛型类型参数在运行时是擦除的，因此无法确定具体的类型，这同样会导致编译错误。

如下面的代码都是报错的。

public class DataStructure<E> {

// 错误的静态方法声明

public static void m(E o1) { // Illegal

}

public static E o1; // Illegal

static {

E o2; // Illegal

}

}

不能声明一个泛型异常类，例如，public class MyException extends Exception { … }

如果允许泛型异常类，你可能会尝试捕获一个泛型异常，例如：

try {

…

} catch (MyException<T> ex) {

…

}

这种捕获语句在编译时是合法的，但在运行时会遇到问题。

在运行时，Java虚拟机（JVM）需要检查从 try 块中抛出的异常是否与 catch 块中指定的异常类型匹配。

由于泛型类型信息在运行时被擦除，JVM无法在运行时确定泛型异常的具体类型。这意味着JVM无法正确地匹配泛型异常类型，从而导致运行时错误。

五！！！！！！！

1.1 Java集合框架（Java Collections Framework）

Java集合框架提供了多种数据结构，可以高效地组织和操作数据。

Java集合框架支持两种类型的容器：

Collection：用于存储一系列元素的容器。

其还包含了大量子接口：

List：存储有序的元素集合，允许重复元素。例如：ArrayList、LinkedList。

Set：存储不重复的元素集合。例如：HashSet、TreeSet。

Stack：存储对象的集合，后进先出（LIFO）的方式处理对象。

Queue：存储对象的集合，先进先出（FIFO）的方式处理对象。

PriorityQueue：存储对象的集合，按照优先级顺序处理对象。

Map：用于存储键值对的容器（在Python中称为字典）。

所有在Java集合框架中定义的接口和类都被组织在 java.util 包中。

Java集合框架的设计是使用接口、抽象类和具体类的一个优秀示例。

接口定义了集合框架的通用API。它们规定了类必须实现的方法，以确保不同集合类之间的互操作性。例如，Collection、List、Set、Map 等接口定义了相应的操作。

抽象类提供了接口的部分实现。这些类实现了接口中的一些方法，并可能添加一些额外的方法或字段，以支持更具体的功能。提供一个部分实现接口的抽象类使用户能够更方便地编写专门化容器的代码。

AbstractCollection 是一个便利抽象类，它提供了 Collection 接口的部分实现。例如，它提供了一些基本操作的实现，如 size()、isEmpty()、contains() 等。

具体类使用具体的数据结构来实现接口。这些类提供了接口方法的具体实现，以及可能的额外功能。例如，ArrayList、LinkedList、HashSet、HashMap 等都是具体类，它们实现了相应的接口，并使用了不同的数据结构来优化性能。

下图展示了Java集合框架（Java Collections Framework）中接口、抽象类和具体类之间层次结构。

Collection 接口是操作对象集合的根接口，即它是所有单列集合（如 List 和 Set）的父接口。

而AbstractCollection 是一个抽象类，它实现了 Collection 接口中的大部分方法，除了 add、size 和 iterator 方法。

Iterable 接口是一个标记接口，它有一个 iterator() 方法，用于返回一个 Iterator 对象。Iterator 接口提供了一种方法来顺序访问集合中的每个元素，而不需要暴露集合的底层表示。它定义了 hasNext()、next() 和 remove() 方法，用于遍历集合并可选地从集合中移除元素。因此我们可以使用for-each 循环通过 Iterator 或者数组索引来遍历元素。

下图展示了Iterable、Collection 和 Iterator 这三个接口的UML图。

下面代码演示了上面的一些方法。

import java.util.\*;

public class TestCollection {

public static void main(String[] args) {

ArrayList<String> collection1 = new ArrayList<>();

collection1.add("New York"); // add

collection1.add("Atlanta");

collection1.add("Dallas");

collection1.add("Madison");

System.out.println("A list of cities in collection1:");

System.out.println(collection1);

// the Collection interface's contains method

System.out.println("\nIs Dallas in collection1? " + collection1.contains("Dallas")); // contains

// the Collection interface's remove method

collection1.remove("Dallas"); // remove

// the Collection interface's size method

System.out.println("\n" + collection1.size() + // size

" cities are in collection1 now");

Collection<String> collection2 = new ArrayList<>();

collection2.add("Seattle");

collection2.add("Portland");

System.out.println("\nA list of cities in collection2:");

System.out.println(collection2);

ArrayList<String> c1 = (ArrayList<String>) (collection1.clone()); // clone

c1.addAll(collection2); // addAll

System.out.println("\nCities in collection1 or collection2:");

System.out.println(c1);

c1 = (ArrayList<String>) (collection1.clone());

c1.retainAll(collection2); // retainAll

System.out.println("\nCities in collection2:");

System.out.println(c1);

c1 = (ArrayList<String>) (collection1.clone());

c1.removeAll(collection2); // removeAll

System.out.println("\nCities in collection1, but not in 2: ");

System.out.println(c1);

}

}

结果如下

A list of cities in collection1:

[New York, Atlanta, Dallas, Madison]

Is Dallas in collection1? true

3 cities are in collection1 now

A list of cities in collection2:

[Seattle, Portland]

Cities in collection1 or collection2:

[New York, Atlanta, Madison, Seattle, Portland]

Cities in collection2:

[]

Cities in collection1, but not in 2:

[New York, Atlanta, Madison]

Java集合框架中的所有具体类（如 ArrayList、HashSet 等）通常实现了 java.lang.Cloneable 和 java.io.Serializable 接口。但是，java.util.PriorityQueue 类没有实现 Cloneable 接口。

在Java集合框架中，有些方法在 Collection 接口中定义，但在具体子类中可能无法实现。例如，对于只读集合（如由 Collections.unmodifiableCollection 方法返回的集合），你不能添加或删除元素。

当一个方法在当前上下文中不被支持时，可以通过抛出 UnsupportedOperationException 来明确告知调用者该操作不被支持。这是一种明确的错误处理方式，让调用者知道他们尝试执行了一个不被允许的操作。

示例如下。

public void someMethod() {

throw new UnsupportedOperationException("Method not supported");

}

1.1.1 迭代器（Iterators）

在Java集合框架中，所有的集合类都实现了 Iterable 接口，这意味着它们都可以用迭代器进行遍历。

迭代器模式允许你顺序访问数据结构中的元素，而不需要暴露数据结构的内部存储细节。

我们可以使用增强型for循环（也称为for-each循环）遍历集合或数组。

示例如下。

for(String element : collection) {

System.out.print(element + " ");

}

Collection 接口继承自 Iterable 接口，从而通过使用 Iterable 接口中的 iterator() 方法来获取一个集合的迭代器对象。这个 Iterator 对象提供了遍历集合的方法，如 hasNext()、next() 和 remove()。

下面的代码展示了如何使用Java集合框架中的 Iterator 来遍历 Collection 集合中的元素。

import java.util.\*;

public class TestIterator {

public static void main(String[] args) {

Collection<String> collection = new ArrayList<>();

collection.add("New York");

collection.add("Atlanta");

collection.add("Dallas");

collection.add("Madison");

Iterator<String> iterator = collection.iterator();

while (iterator.hasNext()) {

System.out.print(iterator.next().toUpperCase() + " ");

}

System.out.println();

}

}

结果如下。

NEW YORK ATLANTA DALLAS MADISON

1

2. 列表（List）

下图展示了与 List 相关的接口、抽象类和具体类之间的层次结构。

List 接口表示的集合会按照元素插入的顺序来存储它们。这意味着元素在列表中有一个明确的顺序。

在 List 集合中，用户可以在添加元素时指定位置（索引），也可以在访问元素时通过索引来获取。

用户可以通过索引访问元素。

List 接口允许集合中的元素有相同的值，即允许重复元素。

Java集合框架中有两个具体类：ArrayList 和 LinkedList：

ArrayList：基于动态数组实现的列表，它提供了快速的随机访问。当需要频繁访问列表中的元素时，ArrayList 是一个好的选择。

LinkedList：基于链表实现的列表，它提供了快速的插入和删除操作。当需要在列表中频繁地添加或删除元素时，LinkedList 是一个更好的选择。

它们都可以根据需要动态地增加或减少其容量。如果你的应用场景中只需要访问元素，而不需要频繁地插入或删除元素，那么使用数组可能是更高效的选择。数组提供了更快的随机访问能力，因为数组的索引访问是 O(1) 的时间复杂度，而 ArrayList 和 LinkedList 的随机访问时间复杂度分别是 O(1) 和 O(n)。

下图展示了 List 接口的UML图。

下图展示了 ListIterator 接口的UML图。

listIterator() 和 listIterator(startIndex)这两个方法返回 ListIterator 的实例，用于遍历列表。

ListIterator 接口扩展了 Iterator 接口，提供了双向遍历列表的能力，并且可以在遍历过程中向列表中添加元素。

nextIndex() 方法返回迭代器中下一个元素的索引。

previousIndex() 方法返回迭代器中上一个元素的索引。

add(E element) 方法将指定的元素插入到列表中，位置是在迭代器的当前位置之前，即下一个通过 next() 方法返回的元素之前。

2.1 ArrayList 类

下图展示了关于 ArrayList 类与 Collection 和 List 接口之间的继承关系。

ArrayList 是基于数组实现的，能够动态调整大小以适应添加和删除操作。

在插入新元素之前，需要将指定索引i ii之后的所有元素向右移动一位，以为新元素腾出空间。插入操作会使得 ArrayList 的大小增加1。

下图展示了这个过程。

2.2 LinkedList 类

下图展示了关于 LinkedList 类与 Collection 和 List 接口之间的继承关系。

LinkedList 由一系列节点组成，每个节点包含一个元素/值（element）和指向下一个节点的引用（next）。

从链表的头节点（head）开始，通过每个节点的 next 引用，可以遍历到链表的尾节点（tail），尾节点的 next 引用为 null。

下列代码定义了一个泛型节点类 Node，其中 E 表示节点可以存储任何类型的数据。

class Node<E> {

E element;

Node<E> next;

public Node(E o) {

element = o;

}

}

2.3 对比

因此我们需要根据我们的应用场景和性能要求选择 ArrayList 或者 LinkedList 来存储和管理数据。如果我们需要经常通过索引访问，则可以选择 ArrayList ，如果是需要很多的插入和删除操作，则可以选择 LinkedList。

下面的代码展示了如何使用 ArrayList 和 LinkedList。

import java.util.\*;

public class TestArrayAndLinkedList {

public static void main(String[] args) {

List<Integer> arrayList = new ArrayList<>();

arrayList.add(1); // 1 is autoboxed to new Integer(1)

arrayList.add(2);

arrayList.add(3);

arrayList.add(1);

arrayList.add(4);

arrayList.add(0, 10);

arrayList.add(3, 30);

System.out.println("A list of integers in the array list:");

System.out.println(arrayList);

LinkedList<Object> linkedList = new LinkedList<>(arrayList);

linkedList.add(1, "red");

linkedList.removeLast();

linkedList.addFirst("green");

System.out.println("Display the linked list backward with index:");

for (int i = linkedList.size() - 1; i >= 0; i--) {

System.out.print(linkedList.get(i) + " ");

}

System.out.println();

System.out.println("Display the linked list forward:");

ListIterator<Object> listIterator = linkedList.listIterator();

while (listIterator.hasNext()) {

System.out.print(listIterator.next() + " ");

}

System.out.println();

System.out.println("Display the linked list backward:");

listIterator = linkedList.listIterator(linkedList.size());

while (listIterator.hasPrevious()) {

System.out.print(listIterator.previous() + " ");

}

}

}

可以自己在ide上尝试下最后的结果。

A list of integers in the array list:

[10, 1, 2, 30, 3, 1, 4]

Display the linked list backward with index:

1 3 30 2 1 red 10 green

Display the linked list forward:

green 10 red 1 2 30 3 1

Display the linked list backward:

1 3 30 2 1 red 10 green

Process finished with exit code 0

get(i) 方法可以在 LinkedList 中使用，但它是一个更耗时的操作，因为它需要从头开始遍历到指定索引 i 的位置才能找到元素。对于 ArrayList 来说，get(i) 方法是高效的，因为它是基于数组实现的，可以直接通过索引访问元素。

为了更高效地遍历列表，推荐使用迭代器或增强型for循环。这两种方法都不需要像 get(i) 那样从头遍历到指定索引，而是直接访问每个元素。

也就是使用下面的代码。

for (listElementType s : list) {

process s;

}

代替原本基础的 for 循环和 get(i) 方法。

for (int i = 0; i < list.size(); i++) {

process list.get(i);

}

2.4 静态方法

Collections 类提供了一组静态方法，用于对集合或列表执行各种操作。这些方法可以对集合或列表进行排序、搜索、修改等。

对于集合来说：

max 和 min 方法用于找出集合中的最大和最小元素。

disjoint 方法用于找出两个集合的不相交部分，即属于其中一个集合但不属于另一个集合的元素。

frequency 方法用于计算某个元素在集合中出现的次数。

对于列表来说：

sort 方法用于对列表进行排序。

binarySearch 方法用于在已排序的列表中进行二分查找。

reverse 方法用于反转列表中元素的顺序。

shuffle 方法用于打乱列表中的元素顺序。

copy 方法用于将一个列表的内容复制到另一个列表。

fill 方法用于用指定的元素填充列表。

对于 sort 方法，有两个重载版本。

static <T extends Comparable<? super T>> void sort(List<T> list)：这个版本的 sort 方法使用 Comparable 接口中的 compareTo 方法对列表进行排序。这意味着列表中的元素必须是 Comparable 的实例，或者其类必须实现了 Comparable 接口。

static <T> void sort(List<T> list, Comparator<T> c)：这个版本的 sort 方法使用 Comparator 接口中的 compare 方法对列表进行排序。这允许你定义自定义排序规则，即使元素不是 Comparable 的实例。

下图展示了 Collection 类的 UML 图。

还有一些其他的有用的静态方法如下：

rotate(List list, int distance)：

这个方法将列表中的所有元素按照指定的距离进行旋转。如果距离为正数，则元素向右旋转；如果距离为负数，则元素向左旋转。例如，Collections.rotate(list, 2) 将列表中的元素向右移动两个位置。

replaceAll(List list, Object oldVal, Object newVal)：

这个方法将列表中所有指定的旧值（oldVal）替换为新值（newVal）。这会影响列表中的所有匹配的元素。

indexOfSubList(List source, List target)：

这个方法返回源列表（source）中第一个与目标列表（target）相等的子列表的索引。如果找不到匹配的子列表，则返回 -1。

lastIndexOfSubList(List source, List target)：

这个方法返回源列表中最后一个与目标列表相等的子列表的索引。如果找不到匹配的子列表，则返回 -1。

swap(List list, int i, int j)：

这个方法交换列表中指定位置（i 和 j）的元素。这两个位置的元素将被互换。

addAll(Collection<? super T> c, T… elements)：

这个方法将指定数组中的所有元素添加到指定的集合（c）中。这允许你一次性向集合中添加多个元素。

2.4.1 sort() 方法

我们现在一个个查看这些关于列表的方法。

下面的代码展示了如何使用 Collections.sort() 对列表进行排序，这个排序是升序排序（自然排序），如果想要获得降序排序（逆自然顺序），可以使用 Collections.sort() 方法和 Collections.reverseOrder() 方法对列表进行降序排序。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class TestCollectionsSort {

public static void main(String[] args) {

List<String> list = Arrays.asList("red", "green", "blue");

Collections.sort(list);

System.out.println(list); // 输出: [blue, green, red]

Collections.sort(list, Collections.reverseOrder());

System.out.println(list); // 输出: [red, green, blue]

}

}

2.4.2 binarySearch() 方法

这个方法会使用我们在算法种学习的二分查找去搜索对应的键。因此它要求我们的列表是升序排列的。如果要查找的键不在列表中，binarySearch 方法会返回一个负数，该数是键应该插入的位置（插入点 + 1）。

示例代码如下。

import java.util.Arrays;

import java.util.Collections;

public class TestBinarySearch {

public static void main(String[] args) {

List<Integer> list1 = Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);

System.out.println("(1) Index: " + Collections.binarySearch(list1, 7)); // 2

System.out.println("(2) Index: " + Collections.binarySearch(list1, 9)); // -4

List<String> list2 = Arrays.asList("blue", "green", "red");

System.out.println("(3) Index: " + Collections.binarySearch(list2, "red")); // 2

System.out.println("(4) Index: " + Collections.binarySearch(list2, "cyan")); // -2

}

}

2.4.3 reverse() 方法

这个方法可以反转列表中元素的顺序。

示例代码如下。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class TestCollectionsReverse {

public static void main(String[] args) {

List<String> list = Arrays.asList("yellow", "red", "green", "blue");

Collections.reverse(list);

System.out.println(list); // 输出: [blue, green, red, yellow]

}

}

2.4.4 shuffle() 方法

这里的洗牌算法可以打乱列表中的元素顺序。

此外还可以使用 shuffle(List, Random) 方法，它允许使用一个特定的 Random 对象来随机重新排序列表中的元素。使用特定的 Random 对象来打乱另一个列表，如列表一样，Random 对象一样那么结果是一样的。

示例代码如下。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

import java.util.Random;

public class TestCollectionsShuffle {

public static void main(String[] args) {

List<String> list = Arrays.asList("yellow", "red", "green", "blue");

// 使用默认的随机数生成器来打乱列表

Collections.shuffle(list);

System.out.println(list);

// 创建一个Random对象

Random rnd = new Random();

// 使用指定的Random对象来打乱列表

List<String> anotherList = Arrays.asList("yellow", "red", "green", "blue");

Collections.shuffle(anotherList, rnd);

System.out.println(anotherList);

}

}

下面的代码展示了 Random 对象的种子相同，两个列表以相同的随机顺序被打乱。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

import java.util.Random;

public class SameShuffle {

public static void main(String[] args) {

List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");

List<String> list2 = Arrays.asList("Y", "R", "G", "B");

Collections.shuffle(list1, new Random(20));

Collections.shuffle(list2, new Random(20));

System.out.println(list1);

System.out.println(list2);

}

}

2.4.5 copy(dest, src) 方法

这个方法将源列表（src）中的所有元素的引用复制到目标列表（dest）中，保持相同的索引。

它实现的是浅拷贝操作，意味着它只复制元素的引用，而不是元素本身。如果源列表中的元素是可变对象，那么目标列表中的对应元素将引用同一个对象。

下面的代码展示了其的使用。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class TestCollectionsCopy {

public static void main(String[] args) {

List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");

List<String> list2 = Arrays.asList("white", "black");

Collections.copy(list1, list2);

System.out.println(list1); // 输出: [white, black, green, blue]

}

}

如果我们将这两个列表的顺序修改一下，即目标列表（destination list）比源列表（source list）小，将会抛出运行时错误（IndexOutOfBoundsException）。

示例代码如下。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class TestCollectionsCopy {

public static void main(String[] args) {

List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");

List<String> list1 = Arrays.asList("white", "black");

Collections.copy(list1, list2);

}

}

2.4.6 Arrays.asList() 方法

这个方法用于根据提供的变量长度参数列表创建一个新的 List。

这个方法返回一个 List 引用，指向 Arrays 类内部定义的一个类对象。这个内部类实际上是 ArrayList，但它是一个包装器（wrapper），用于将数组转换为列表。

以下是示例代码。

import java.util.Arrays;

import java.util.List;

public class TestArrayList {

public static void main(String[] args) {

List<String> list1 = Arrays.asList("red", "green", "blue");

List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);

System.out.println(list1); // 输出: [red, green, blue]

System.out.println(list2); // 输出: [10, 20, 30, 40, 50]

}

}

2.4.7 nCopies(int n, Object o) 方法

这个方法会创建一个包含指定对象 o 的 n 个副本的列表。

使用 nCopies 方法创建的列表是不可变的，这意味着你不能向列表中添加或移除元素。

由于列表中的所有元素都是同一个对象的副本，它们都指向相同的内存地址。

返回的列表类型是 Collections$CopiesList，这是 java.util.Collections 类的一个内部类，它实现了 List 接口。

示例代码如下。

import java.util.Arrays;

import java.util.Calendar;

import java.util.Collections;

import java.util.List;

public class TestCollectionsNCopies {

public static void main(String[] args) {

List<Calendar> list1 = Collections.nCopies(3, new GregorianCalendar(2022, Calendar.JANUARY, 1));

System.out.println(list1);

}

}

下面的代码展示了由于修改一个对象所以影响列表中的所有对象。

import java.util.\*;

public class Test {

public static void main(String[] args) {

List<GregorianCalendar> list1 = Collections.nCopies(3, new GregorianCalendar(2022, Calendar.JANUARY, 1));

list1.get(0).set(Calendar.YEAR, 2024);

for (GregorianCalendar g : list1) {

System.out.println(g.get(Calendar.YEAR));

}

}

}

2.4.8 fill(List list, Object o) 方法

这个方法将列表 list 中的所有元素替换为指定的元素 o。

示例代码如下。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class TestCollectionsFill {

public static void main(String[] args) {

List<String> list = Arrays.asList("red", "green", "blue");

Collections.fill(list, "black");

System.out.println(list); // 输出: [black, black, black]

}

}

2.4.9 max(), min() 方法

现在看关于 Collection 的方法。

Collections.max() 方法用于找出集合中的最大元素。它使用元素的 compareTo 方法来确定最大值。

Collections.min() 方法用于找出集合中的最小元素。它同样使用元素的 compareTo 方法来确定最小值。

示例代码如下。

import java.util.Arrays;

import java.util.Collections;

import java.util.List;

public class TestCollectionsMaxMin {

public static void main(String[] args) {

List<String> collection = Arrays.asList("red", "green", "blue");

System.out.println(Collections.max(collection)); // 输出: red

System.out.println(Collections.min(collection)); // 输出: blue

}

}

2.4.10 disjoint(collection1, collection2) 方法

这个方法用于检查两个集合是否有共同的元素。如果有共同元素，方法返回 false；如果没有共同元素，方法返回 true。

示例代码如下。

import java.util.Arrays;

import java.util.Collection;

import java.util.Collections;

public class TestCollectionsDisjoint {

public static void main(String[] args) {

Collection<String> collection1 = Arrays.asList("red", "cyan");

Collection<String> collection2 = Arrays.asList("red", "blue");

Collection<String> collection3 = Arrays.asList("pink", "tan");

System.out.println(Collections.disjoint(collection1, collection2)); // 输出: false

System.out.println(Collections.disjoint(collection1, collection3)); // 输出: true

}

}

2.4.11 frequency(collection, element) 方法

这个方法用于计算集合中某个元素出现的次数。

示例代码如下。

import java.util.Arrays;

import java.util.Collection;

import java.util.Collections;

public class TestCollectionsFrequency {

public static void main(String[] args) {

Collection<String> collection = Arrays.asList("red", "cyan", "red");

System.out.println(Collections.frequency(collection, "red")); // 输出: 2

}

}

更多例子。

import java.util.Arrays;

import java.util.Collection;

import java.util.Collections;

public class TestCollectionsFrequency {

public static void main(String[] args) {

Collection<String> collection = Arrays.asList(new String("red"), "cyan", new String("red"), "red");

System.out.println(Collections.frequency(collection, "red")); // 输出: 3

}

}

“red” 被创建了三次（每次都是一个新的对象），但是由于 equals 方法用于比较字符串的内容，所以 “red” 被正确地计算为出现了三次。

同理如下。

import java.util.Arrays;

import java.util.Collection;

import java.util.Collections;

public class TestCollectionsFrequency {

public static void main(String[] args) {

Collection<String> collection = Arrays.asList("red", "cyan", "red");

System.out.println(Collections.frequency(collection, new String("red"))); // 输出: 2

}

}

3. Vector 和 Stack 类

这两个类在Java 2之前就已设计，在集合框架设计后，这些类被重新设计以适应Java集合框架，但为了兼容性，保留了它们的旧方法。

Vector 和 Stack 类在现代Java集合框架中不常用（因为它们的方法不是线程安全的，且功能已被 ArrayList、LinkedList 等类取代），但它们在集合框架中仍然占有一席之地，特别是为了保持向后兼容性。

3.1 Vector 类

Vector 类在功能上与 ArrayList 类似，但主要区别在于 Vector 类包含了同步（synchronized）方法，用于访问和修改向量（vector）。

同步方法可以防止数据损坏，当一个向量被两个或更多线程同时访问和修改时，同步机制确保了数据的一致性和完整性。

到目前为止讨论的所有类都不是同步的。

对于许多不需要同步的应用，使用 ArrayList 比使用 Vector 更高效。这是因为 ArrayList 不包含额外的同步开销。

addElement(Object element) 方法（从Java2保留的方法）与 add(Object element) 方法相同，唯一的区别在于 addElement 方法是同步的。

Vector 类的UML图如下。

3.2 Stack 类

Stack 类表示一个后进先出（LIFO, Last-In-First-Out）的栈对象集合。

在 Stack 中，所有元素的访问、检索和移除操作都只能从栈的顶部进行。

相关方法如下：

push(o: E) 方法用于在栈顶添加新元素。

peek() 方法用于获取栈顶元素，但不移除它。

pop() 方法用于移除栈顶元素，并返回该元素。

Stack 类是 Vector 类的子类，这意味着它继承了 Vector 类的所有属性和方法，并在此基础上增加了栈特定的功能。

empty() 方法（从Java2保留的方法）与 isEmpty() 方法相同，都用于检查栈是否为空。

UML图如下。

4. 队列（Queues）和优先队列（Priority Queues）

队列是一种先进先出（FIFO, First-In-First-Out）的数据结构。

元素被添加到队列的末尾，并且从队列的开头被移除。

相关方法如下：

offer(o: E) 方法：这个方法用于向队列中添加一个元素。与 Collection 接口中的 add 方法相似，但对于队列来说，offer 方法是首选的方法。

poll 和 remove 方法：这两个方法用于从队列中移除元素。poll() 方法在队列为空时返回 null，而 remove() 方法在队列为空时抛出异常。

peek 和 element 方法：

这两个方法用于查看队列中的元素而不移除它。peek() 方法在队列为空时返回 null，而 element() 方法在队列为空时抛出异常。

在优先队列中，元素被分配优先级。当访问元素时，具有最高优先级的元素首先被移除。

4.1 队列（Queue）

Queue 接口继承自 Collection 接口，它包含了 Collection 接口的所有方法，并且添加了一些额外的操作。

其UML图如下。

LinkedList 类不仅可以作为列表使用，还可以作为队列使用，因为它实现了 Queue 和 Deque 接口。

下图展示了这些接口和类之间的继承关系。

Queue 接口支持在队列的两端插入和移除元素。而Deque 是 “double-ended queue” 的缩写，它扩展了 Queue 接口，增加了在队列两端插入和移除元素的方法。

Deque 接口定义了 addFirst(e)、removeFirst()、addLast(e)、removeLast()、getFirst() 和 getLast() 方法，即允许在队列的头部和尾部进行操作。

下面给出一个示例代码。

import java.util.LinkedList;

import java.util.Queue;

public class TestQueue {

public static void main(String[] args) {

Queue<String> queue = new LinkedList<>();

queue.offer("Oklahoma");

queue.offer("Indiana");

queue.offer("Georgia");

queue.offer("Texas");

while (queue.size() > 0) {

System.out.print(queue.remove() + " ");

}

}

}

它主要展示了 offer 方法的使用。

4.2 PriorityQueue 类

默认情况下，PriorityQueue 使用元素的自然排序（natural ordering）来确定优先级。这意味着元素必须实现 Comparable 接口，并且优先级是根据 compareTo 方法的结果来确定的。

值最小的元素被赋予最高的优先级，因此会首先从队列中移除。

如果有多个元素具有相同的最高优先级，这些元素之间的顺序是任意的。

可以通过 PriorityQueue 构造函数指定一个 Comparator 来定义元素的排序规则。

下图展示了其的UML图。

下面给出示例代码。

import java.util.\*;

public class PriorityQueueDemo {

public static void main(String[] args) {

PriorityQueue<String> queue1 = new PriorityQueue<>();

queue1.offer("Oklahoma");

queue1.offer("Indiana");

queue1.offer("Georgia");

queue1.offer("Texas");

System.out.println("Priority queue using Comparable:");

while (queue1.size() > 0) {

System.out.print(queue1.remove() + " ");

} // 输出顺序：Georgia Indiana Oklahoma Texas

PriorityQueue<String> queue2 = new PriorityQueue<>(4, Collections.reverseOrder());

queue2.offer("Oklahoma");

queue2.offer("Indiana");

queue2.offer("Georgia");

queue2.offer("Texas");

System.out.println("\nPriority queue using Comparator:");

while (queue2.size() > 0) {

System.out.print(queue2.remove() + " ");

} // 输出顺序：Texas Oklahoma Indiana Georgia

}

}

六！！！！！！！！！！！！！！！！！（SET 集合 MAP 映射）

1. 集合（Set）

Set 接口继承自 Collection 接口，这意味着 Set 接口继承了 Collection 接口的所有方法和行为。

尽管 Set 接口继承自 Collection，但它并没有添加任何新的方法或常量。它主要是对集合的行为进行了额外的约束。

Set 接口的主要特征是它不允许集合中有重复的元素。也就是说，集合中的任意两个元素 e1 和 e2 不能同时存在，使得 e1.equals(e2) 为 true。

我们可以使用 HashSet、LinkedHashSet 或 TreeSet 之一来创建一个 Set 集合。这些类都是 Set 接口的具体实现。

实现 Set 接口的具体类必须确保不能向集合中添加重复的元素。

HashSet 和 LinkedHashSet 都是基于哈希（hash）表实现的。它们使用对象的 hashCode() 方法来计算哈希值，以及使用 equals() 方法来比较对象是否相等。

TreeSet 是基于红黑树（一种自平衡的二叉搜索树）实现的。它使用元素的自然顺序（natural ordering）来维护元素。这意味着元素必须实现 Comparable 接口，或者在创建 TreeSet 时提供一个 Comparator 来定义元素的排序规则。

1.1 HashSet 类

HashSet 是 Set 接口的一个具体实现，这意味着它提供了 Set 接口定义的所有方法的实现，并且确保集合中的元素是唯一的（没有重复元素）。

1.1.1 构造函数

我们可以使用 HashSet 类的无参构造函数来创建一个空的 HashSet 实例。

代码如下。

Set<Srting> set = new HashSet<>();

第一个菱形操作符（<>）被称为类型参数或泛型类型。它指定了HashSet将存储的元素类型。

在这个例子中，HashSet被指定为存储String类型的对象。

第二个菱形操作符（<>）中，编译器会根据上下文推断泛型类型，这通常与第一个菱形操作符中指定的类型相同（仅在简单情况下）。

括号()用于调用HashSet类的构造函数。在这个例子中，调用的是无参构造函数，它创建了一个空的集合。

我们也可以使用将一个List创建为一个 HashSet。

代码如下。

List<String> list = Arrays.asList("Apple");

HashSet<String> hashSet = new HashSet<>(list);

这里第一个菱形操作符（diamond operation）内是 String， 因为列表的类型是 List<String>。而这里括号()接受列表的参数用于 HashSet 的构造函数。

下面再介绍一种构造方式，这个方法可以指定 HashSet 的初始容量，以及它还可以接受一个负载因子。负载因子的范围是0.0到1.0，它衡量了在增加容量之前，集合可以被填充的程度。当集合的大小达到初始容量乘以负载因子时，容量会翻倍。

示例代码如下。

// 创建一个具有指定初始容量的HashSet

int initialCapacity = 16;

HashSet<String> hashSet1 = new HashSet<>(initialCapacity);

// 创建一个具有指定初始容量和负载因子的HashSet

float loadFactor = 0.75f;

HashSet<String> hashSet2 = new HashSet<>(initialCapacity, loadFactor);

因此这里初始容量是16，负载因子是0.75，当集合的大小达到12（160.75=12）时，容量会翻倍到32（162）。

1.1.2 相关方法

HashSet，LinkedHashSet，TreeSet 都实现了 Collection 接口和抽象类 AbstractSet，所以他们都可以使用add(),remove()等方法。

1.1.2.1 add() 方法

这个方法可以向集合里添加元素。

示例如下。

public class TestMethodsInCollection {

public static void main(String[] args) {

// 创建 set1

java.util.Set<String> set1 = new java.util.HashSet<>();

// 向 set1 添加字符串

set1.add("London");

set1.add("Paris");

set1.add("New York");

set1.add("San Francisco");

set1.add("Beijing");

// 打印 set1 及其大小

System.out.println("set1 is " + set1);

System.out.println(set1.size() + " elements in set1");

}

}

如果我们尝试在代码里多添加一个重复元素，示例如下。

public class TestMethodsInCollection {

public static void main(String[] args) {

// 创建 set1

java.util.Set<String> set1 = new java.util.HashSet<>();

// 向 set1 添加字符串

set1.add("London");

set1.add("Paris");

set1.add("New York");

set1.add("San Francisco");

set1.add("Beijing");

set1.add("New York");

// 打印 set1 及其大小

System.out.println("set1 is " + set1);

System.out.println(set1.size() + " elements in set1");

}

}

我们会发现最后的结果不变，因为 HashSet 是非重复的，因为它通过哈希表来存储元素，哈希表的键是唯一的。当尝试添加一个已经存在的元素时，HashSet会检查该元素是否已经存在于集合中。如果存在，HashSet不会再次添加该元素，从而确保集合中的元素是非重复的。

我们可以点击 String 类名，然后按F4键，来查看 String 类的源代码。

我们可以按Ctrl + F12快速找到相关方法。

我们可以发现 HashSet 利用 String 类的 equals() 和 hashCode() 方法来检查重复元素。

因此我们可以尝试下列代码。

import java.util.Objects;

import java.util.HashSet;

import java.util.Set;

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + '}';

}

}

public class PersonTest {

public static void main(String[] args) {

// 在另一个类中使用 Person 类

Set<Person> set1 = new HashSet<>();

set1.add(new Person("John", 19));

set1.add(new Person("Mary", 20));

set1.add(new Person("John", 19));

System.out.println(set1);

}

}

由于 HashSet 是基于哈希表实现的，它使用对象的 hashCode() 和 equals() 方法来确定对象的唯一性。在Java中，如果没有重写这些方法，hashCode() 默认返回对象的内存地址的哈希码，而 equals() 默认只比较对象的引用。

在这个例子中，尽管有两个 Person 对象具有相同的 name 和 age，但由于没有重写 hashCode() 和 equals() 方法，这两个对象被视为不同的对象。因此，HashSet 会将它们都添加到集合中。

下面的代码就指挥打印出两个结果。

import java.util.Objects;

import java.util.HashSet;

import java.util.Set;

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Person person = (Person) o;

return age == person.age && Objects.equals(name, person.name);

}

@Override

public int hashCode() {

return Objects.hash(name, age);

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + '}';

}

}

public class PersonTest {

public static void main(String[] args) {

// 在另一个类中使用 Person 类

Set<Person> set1 = new HashSet<>();

set1.add(new Person("John", 19));

set1.add(new Person("Mary", 20));

set1.add(new Person("John", 19));

System.out.println(set1);

}

}

1.1.2.2 增强for循环（for-each循环）

Collection 接口扩展了 Iterable 接口，这意味着实现了 Collection 接口的类（如 HashSet）也实现了 Iterable 接口。

因此我们可以使用增强for循环遍历集合中的每个元素。

示例如下。

import java.util.HashSet;

import java.util.Set;

public class Main {

public static void main(String[] args) {

Set<String> set = new HashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Cherry");

// 使用增强型for循环遍历HashSet

for (String fruit: set) {

System.out.println(fruit.toLowerCase());

}

}

}

HashSet是无序的，这意味着它不保证元素的存储顺序，并且没有像数组那样的索引（即你不能通过下标[i]来访问元素）。增强型for循环允许你方便地遍历HashSet中的所有元素，而不需要关心元素的存储顺序或使用索引。

1.1.2.3 forEach() 方法

我们也可以使用 forEach() 方法循环遍历集合中的每个元素。

forEach()是Iterable接口中的一个默认方法，用于对集合中的每个元素执行指定的操作。

语法为Set.forEach(Consumer<? super E> action)。

常用方法为set.forEach(e -> System.out.print())。

这里e是传递给lambda表达式的参数，表示当前集合中的一个元素。

->是lambda表达式中的箭头，用于分隔参数和函数体。

因此代码如下。

set.forEach(e -> System.out.print(e.toLowerCase() + " "));

1.1.2.4 其他常用方法

下面的代码展示了其他常用方法的使用。

import java.util.HashSet;

import java.util.Set;

public class HashSetDemo {

public static void main(String[] args) {

Set<String> set1 = new HashSet<>();

set1.add("London");

set1.add("Paris");

set1.add("New York");

set1.add("San Francisco");

set1.add("Beijing");

Set<String> set2 = new HashSet<>();

set2.add("New York");

set2.add("London");

set2.add("Taipei");

// remove(): Delete a string from set1

set1.remove("London");

System.out.println("\nset1 is " + set1);

// size(): The size of the set

System.out.println(set1.size() + " elements in set1");

// contains(): If the set contains a certain element, return T/F

System.out.println("\nIs Taipei in set2? " + set2.contains("Taipei"));

// addAll(): add the elements in set1 and set2 together. NO DUPLICATION!

// hashCode() and equals() are called

set1.addAll(set2);

System.out.println("\nAfter adding set2 to set1, set1 is " + set1);

// removeAll(): removing the elements in set 2 from set1

set1.removeAll(set2);

System.out.println("\nAfter removing set2 from set1, set1 is " + set1);

// retainAll(): What is the printed result in this case?

set1.retainAll(set2);

System.out.println("\nAfter retaining common elements in set1 and set2, set1 is " + set1);

}

}

这里只稍微说明一下这里的retainAll()方法是保留set1中与set2共有的元素，但是前一步set1移除了所有在set2中存在的元素，所以最后一行运行的结果没有任何元素。

1.2 LinkedHastSet 类

LinkedHashSet 是 HashSet 的一个子类，它继承了 HashSet 的特性，并添加了一个链表实现以支持元素的顺序。

LinkedHashSet 维护元素的插入顺序，即元素可以按照它们被添加到集合中的顺序被检索和迭代。

LinkedHashSet 在很多方面与 HashSet 非常相似。之前提到的 HashSet 的创建方式和方法（如添加元素、删除元素等）同样适用于 LinkedHashSet。

LinkedHashSet 和 HashSet 的一个显著区别是，LinkedHashSet 中的元素可以按照它们被插入的顺序被检索。这意味着如果你需要保持元素的添加顺序，LinkedHashSet 是一个比 HashSet 更好的选择。

示例代码如下。

import java.util.\*;

public class TestLinkedHashSet {

public static void main(String[] args) {

// Create a linked hash set

Set<String> set = new LinkedHashSet<>();

// Add strings to the set

set.add("London");

set.add("Paris");

set.add("New York");

set.add("San Francisco");

set.add("Beijing");

set.add("New York");

System.out.println(set);

// Display the elements in the hash set

for (String element : set) {

System.out.print(element.toLowerCase() + " ");

}

}

}

这里打印出来的结果与添加的顺序一致。

1.3 TreeSet 类

TreeSet 是一个具体的类，它实现了 SortedSet 和 NavigableSet 接口。

SortedSet 是 Set 接口的一个子接口，它保证了集合中的元素是有序的。

这意味着当你遍历 SortedSet 时，元素会按照某种排序顺序（通常是自然排序或构造时指定的比较器）返回。

NavigableSet 接口扩展了 SortedSet 接口，提供了导航方法，用于在有序集合中查找特定元素或一组元素。

这些导航方法包括：

lower(E e)：返回集合中小于给定元素的最大元素。

floor(E e)：返回集合中小于或等于给定元素的最大元素。

higher(E e)：返回集合中大于给定元素的最小元素。

ceiling(E e)：返回集合中大于或等于给定元素的最小元素。

1.3.1 构造方法

我们可以创建一个空的 TreeSet，它将根据其元素的自然顺序（natural ordering）进行升序排序。

示例代码如下。

TreeSet<String> treeSet = new TreeSet<>();

由于 TreeSet 实现了 SortedSet 接口，它会自动对元素进行排序。

我们也可以从其他集合（如 List）创建一个 TreeSet，新集合将根据元素的自然顺序进行排序。

TreeSet<String> treeSet = new TreeSet<>(list);

我们也可以使用自定义比较器创建一个 TreeSet，以定义元素的排序顺序。

TreeSet<String> treeSet = new TreeSet<>(Comparator.reverseOrder());

如果我们已经有一个根据特定规则排序的 SortedSet，你可以创建一个新的 TreeSet，它包含相同的元素并保持相同的排序顺序。

相关的UML图如下。

1.3.2 相关方法

1.3.2.1 add() 方法

类似于 HashSet，TreeSet 也不会添加重复的元素。但TreeSet 不使用 hashCode() 和 equals() 方法来检查重复元素，而是使用 Java 标准库中 String 和 Integer 等包装类提供的内置 compareTo() 方法。

HashSet 使用哈希表（Hashing）作为底层数据结构，而 TreeSet 使用树（Tree）结构，具体来说是红黑树。

这种底层数据结构的不同导致了 HashSet 和 TreeSet 在处理元素时的不同行为，尤其是在元素排序和查找方面。

示例代码如下。

import java.util.TreeSet;

public class TreeSetTest {

public static void main(String[] args) {

// 创建一个树形集合

TreeSet<String> treeSet = new TreeSet<>();

// 添加元素

treeSet.add("Apple");

treeSet.add("Banana");

treeSet.add("Cherry");

treeSet.add("Apple"); // 重复元素，不会被添加

treeSet.add("Date");

treeSet.add("Banana"); // 重复元素，不会被添加

// 打印集合内容

System.out.println("TreeSet contents: " + treeSet);

}

}

1.3.2.2 其他常用方法

作为 SortedSet ，其的常用方法如下：

first() 方法：

返回集合中的第一个元素，即按照排序顺序最小的元素。

last() 方法：

返回集合中的最后一个元素，即按照排序顺序最大的元素。

headSet() 方法：

返回一个视图（原始集合的一个子集），包含所有小于指定元素的集合元素。

tailSet() 方法：

返回一个视图（原始集合的一个子集），包含所有大于（或等于）指定元素的集合元素。

示例代码如下。

import java.util.\*;

public class TestTreeSet {

public static void main(String[] args) {

// 创建一个树形集合

TreeSet<String> set = new TreeSet<>();

set.add("London");

set.add("Paris");

set.add("New York");

set.add("San Francisco");

set.add("Beijing");

// 打印排序后的树形集合

System.out.println("Sorted tree set: " + set);

// 使用 SortedSet 接口中的方法

System.out.println("first(): " + set.first());

System.out.println("last(): " + set.last());

System.out.println("headSet(\"New York\"): " + set.headSet("New York"));

System.out.println("tailSet(\"New York\"): " + set.tailSet("New York"));

}

}

这里给出一个更简单的例子。

TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));

SortedSet<Integer> headSet = numbers.headSet(5);

在这个例子中，headSet 将包含所有小于 5 的元素，即 [1, 2, 3, 4]。

作为 NavigableSet ，其的常用方法如下：

lower()：

返回集合中小于给定元素的最大元素。

higher()：

返回集合中大于给定元素的最小元素。

floor()：

返回集合中小于或等于给定元素的最大元素。

ceiling()：

返回集合中大于或等于给定元素的最小元素。

pollFirst()：

检索并移除集合中的第一个（最小）元素。

pollLast()：

检索并移除集合中的最后一个（最大）元素。

示例代码如下。

TreeSet<Integer> treeSet1 = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));

// 使用 NavigableSet 接口中的方法

System.out.println("lower(5): " + treeSet1.lower(5));

System.out.println("higher(5): " + treeSet1.higher(5));

System.out.println("floor(5): " + treeSet1.floor(5));

System.out.println("ceiling(5): " + treeSet1.ceiling(5));

System.out.println("pollFirst(): " + treeSet1.pollFirst());

System.out.println("pollLast(): " + treeSet1.pollLast());

System.out.println("New tree set: " + treeSet1);

1.4 Set 和 List 的性能对比

import java.util.\*;

public class SetListPerformanceTest {

static final int N = 50000;

public static long getTestTime(Collection<Integer> c) {

long startTime = System.currentTimeMillis();

for (int i = 0; i < N; i++)

c.contains((int)(Math.random() \* 2 \* N));

return System.currentTimeMillis() - startTime;

}

public static long getRemoveTime(Collection<Integer> c) {

long startTime = System.currentTimeMillis();

for (int i = 0; i < N; i++)

c.remove(i);

return System.currentTimeMillis() - startTime;

}

public static void main(String[] args) {

// Add numbers 0, 1, 2, ..., N - 1 to an array list

List<Integer> list = new ArrayList<>();

for (int i = 0; i < N; i++)

list.add(i);

Collections.shuffle(list); // Shuffle the array list

// Create a hash set, and test its performance

Collection<Integer> set1 = new HashSet<>(list);

System.out.println("Member test time for hash set is " + getTestTime(set1) + " milliseconds");

System.out.println("Remove element time for hash set is " + getRemoveTime(set1) + " milliseconds");

// Create a linked hash set, and test its performance

Collection<Integer> set2 = new LinkedHashSet<>(list);

System.out.println("Member test time for linked hash set is " + getTestTime(set2) + " milliseconds");

System.out.println("Remove element time for linked hash set is " + getRemoveTime(set2) + " milliseconds");

// Create a tree set, and test its performance

Collection<Integer> set3 = new TreeSet<>(list);

System.out.println("Member test time for tree set is " + getTestTime(set3) + " milliseconds");

System.out.println("Remove element time for tree set is " + getRemoveTime(set3) + " milliseconds");

// Create an array list, and test its performance

Collection<Integer> list1 = new ArrayList<>(list);

System.out.println("Member test time for array list is " + getTestTime(list1) + " milliseconds");

System.out.println("Remove element time for array list is " + getRemoveTime(list1) + " milliseconds");

// Create a linked list, and test its performance

Collection<Integer> list2 = new LinkedList<>(list);

System.out.println("Member test time for linked list is " + getTestTime(list2) + " milliseconds");

System.out.println("Remove element time for linked list is " + getRemoveTime(list2) + " milliseconds");

}

}

HashSet 和 LinkedHashSet 在查找和删除操作上比 TreeSet 更快，因为它们使用哈希表实现，而 TreeSet 使用红黑树实现，这需要额外的时间开销。同样，ArrayList 和 LinkedList 在查找操作上也有所不同，ArrayList 使用数组实现，查找时间复杂度为 O(n)，而 LinkedList 使用链表实现，查找时间复杂度也为 O(n)，但删除操作的时间复杂度不同。

因此综合来看，Set 比 List 更适合存储不重复的元素。List 允许通过索引访问元素。Set 不支持索引访问，因为 Set 中的元素是无序的。要遍历 Set 中的所有元素，可以使用增强型for循环（for-each loop）或迭代器（iterator）。

1.5 总结

HashSet、LinkedHashSet 和 TreeSet 都是Java中 Set 接口的实现，这意味着它们都不允许存储重复元素。

HashSet：

排序：它不保证迭代的顺序。

内部结构：由哈希表支持。

LinkedHashSet：

排序：维护一个贯穿所有条目的双向链表，这定义了迭代的顺序，通常是元素被插入到集合中的顺序（插入顺序）。

内部结构：由哈希表支持，并通过一个链表贯穿。

TreeSet：

排序：保证元素将按照元素的自然顺序或在集合创建时提供的比较器进行升序排序。

内部结构：由树（通常是红黑树）支持。

2. 映射（Map）

与 Set 类似， Java提供了三种主要的 Map 实现：HashMap、LinkedHashMap 和 TreeMap。

这些 Map 实现需要确保键值对（key-value pairs）的唯一性。

HashMap 和 LinkedHashMap 使用 hashCode() 和 equals() 方法来检查键的唯一性。

TreeMap 使用 compareTo() 方法或在创建时提供的 Comparator 来检查键的唯一性。

TreeMap 除了实现 Map 接口外，还实现了 SortedMap 和 NavigableMap 接口。

下图展示了相关的UML图。

2.1 构造方法

可以使用泛型参数创建新的 HashMap、LinkedHashMap 或 TreeMap 实例。

构造函数的形式为Map<? extends K, ? extends V> 的参数，其中 K 是键的类型，V 是值的类型。

示例代码如下。

import java.util.HashMap;

import java.util.Map;

public class HashMapExample {

public static void main(String[] args) {

// 创建一个包含整数键和字符串值的 Map

Map<Integer, String> smallMap = new HashMap<>();

smallMap.put(10, "Ten");

smallMap.put(20, "Twenty");

// 使用接受另一个 Map 的构造函数创建一个新的 HashMap

Map<Number, Object> largerMap = new HashMap<>(smallMap);

System.out.println("Contents of largerMap: " + largerMap);

}

}

这里接受的参数中的键值对类型是Integer, String，它们分别是Number, Object的子类，因此可以。

当然也可以就是Integer, String。

如下。

import java.util.HashMap;

import java.util.Map;

public class HashMapExample {

public static void main(String[] args) {

// 创建一个包含整数键和字符串值的 Map

Map<Integer, String> smallMap = new HashMap<>();

smallMap.put(10, "Ten");

smallMap.put(20, "Twenty");

// 使用接受另一个 Map 的构造函数创建一个新的 HashMap

Map<Integer, String> largerMap = new HashMap<>(smallMap);

System.out.println("Contents of largerMap: " + largerMap);

}

}

LinkedHashMap 是 HashMap 的一个扩展，它使用链表实现，支持按插入顺序检索元素。

LinkedHashMap 的构造函数有三个参数：

initialCapacity：初始容量，整数类型。

loadFactor：负载因子，浮点数类型。

accessOrder：访问顺序，布尔类型。如果设置为 true，则 LinkedHashMap 会按照元素最后被访问的顺序排列元素，这意味着在每次访问（读或写）元素后，该元素会被移动到链表的末尾，从而使得最近访问的元素位于链表的尾部。如果设置为 false（默认值），LinkedHashMap 会按照元素的插入顺序来排列元素。即元素会按照它们被添加到映射中的顺序进行排序。

import java.util.\*;

public class LinkedHashMapExample {

public static void main(String[] args) {

// 创建一个 LinkedHashMap

Map<String, Integer> linkedHashMap = new LinkedHashMap<>(16, 0.75f,true);

linkedHashMap.put("Smith", 30);

linkedHashMap.put("Anderson", 31);

linkedHashMap.put("Lewis", 29);

linkedHashMap.put("Cook", 29);

// 在访问任何元素之前显示地图

System.out.println("\nDisplay before any access");

System.out.println(linkedHashMap);

// 访问 Lewis 以获取他的年龄

System.out.println("\nThe age for Lewis is " + linkedHashMap.get("Lewis"));

// 在访问一个元素之后显示地图

System.out.println("After an element is accessed the entries in LinkedHashMap are\n");

System.out.println(linkedHashMap);

}

}

比如这里本来的顺序是插入的顺序，但是我们中途访问了 Lewis，所以导致现在 Lewis 出现在了最后。

2.2 相关方法

HashMap 是无序的，类似于 HashSet。

TreeMap 是有序的，按照键的自然顺序或指定的比较器进行排序。

LinkedHashMap 可以按插入顺序排序，也可以按访问顺序（accessOrder: true）排序。

常用方法如下：

get()：返回与指定键关联的值。

forEach()：对 Map 中的每个条目执行给定的操作，直到所有条目都被处理或操作抛出异常。用法和前面的Set的类似。

import java.util.\*;

public class TestMap {

public static void main(String[] args) {

// 创建一个 HashMap

Map<String, Integer> hashMap = new HashMap<>();

hashMap.put("Smith", 30);

hashMap.put("Anderson", 31);

hashMap.put("Lewis", 29);

hashMap.put("Cook", 29);

System.out.println("Display entries in HashMap: " + hashMap);

// 从 HashMap 创建一个 TreeMap

Map<String, Integer> treeMap = new TreeMap<>(hashMap);

System.out.println("Display entries in ascending order of key:");

System.out.println(treeMap);

// 创建一个 LinkedHashMap

Map<String, Integer> linkedHashMap = new LinkedHashMap<>(10, 0.75f, true);

linkedHashMap.put("Smith", 30);

linkedHashMap.put("Anderson", 31);

linkedHashMap.put("Lewis", 29);

linkedHashMap.put("Cook", 29);

// 在访问任何元素之前显示映射

System.out.println("\nDisplay before any access");

System.out.println(linkedHashMap);

// 访问 Lewis 以获取他的年龄

System.out.println("\nThe age for Lewis is " + linkedHashMap.get("Lewis"));

// 在访问一个元素之后显示映射

System.out.println("After an element is accessed the entries in LinkedHashMap are\n");

System.out.println(linkedHashMap);

// 使用 forEach 显示每个条目，带有名称和年龄

System.out.println("\nNames and ages are ");

treeMap.forEach((name, age) -> System.out.println(name + ": " + age + " "));

}

}

下面的代码展示了HashMap里的一些方法。

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import java.util.Collection;

public class HashMaps4Fun {

public static void main(String[] args) {

// 创建一个新的 HashMap

Map<String, Integer> hashMap = new HashMap<>();

// 向 map 中放入一些键值对

hashMap.put("One", 1);

hashMap.put("Two", 2);

hashMap.put("Three", 3);

// 测试 containsKey 方法

System.out.println("Does hashMap contain 'Two'? " + hashMap.containsKey("Two"));

// 测试 containsValue 方法

System.out.println("Does hashMap contain value '3'? " + hashMap.containsValue(3));

// 测试 entrySet 方法

Set<Map.Entry<String, Integer>> entries = hashMap.entrySet();

System.out.println("Entry set: " + entries);

}

}

下面的代码展示了TreeMap里的一些方法。

import java.util.TreeMap;

public class TreeMaps4Fun {

public static void main(String[] args) {

// 创建一个 TreeMap 并添加一些条目

TreeMap<String, Integer> treeMap = new TreeMap<>();

treeMap.put("Apple", 3);

treeMap.put("Banana", 5);

treeMap.put("Date", 4);

treeMap.put("Elderberry", 1);

// 测试 SortedMap 方法

System.out.println("First key: " + treeMap.firstKey());

System.out.println("Last key: " + treeMap.lastKey());

System.out.println("HeadMap (keys less than 'Date'): " + treeMap.headMap("Date"));

System.out.println("TailMap (keys greater than or equal to 'Date'): " + treeMap.tailMap("Date"));

// 测试 NavigableMap 方法

System.out.println("Lower key than 'Cherry': " + treeMap.lowerEntry("Cherry"));

System.out.println("Floor key of 'Cherry': " + treeMap.floorEntry("Cherry"));

System.out.println("Ceiling key of 'Cherry': " + treeMap.ceilingEntry("Cherry"));

System.out.println("Higher key than 'Cherry': " + treeMap.higherEntry("Cherry"));

// 轮询条目

System.out.println("Poll first entry: " + treeMap.pollFirstEntry());

System.out.println("Poll last entry: " + treeMap.pollLastEntry());

System.out.println("TreeMap after polling: " + treeMap);

}

}

七！！！！！！！！！！！！！！！（算法 ALGORITHM）

1.1 算法分析

1.1 时间复杂度

如果我们使用执行时间来衡量算法性能，其具有局限性，因为执行时间依赖于具体输入和系统环境。

所以我们使用大 O 符号来衡量算法效率。大 O 符号不关心算法在特定输入数据或特定计算机硬件上的具体执行时间，而是关注算法随输入规模变化时的增长趋势。

时间复杂度描述了算法的执行时间如何随着输入规模 n 的增加而变化。使用大 O 符号描述时间复杂度时，需要注意两点：

忽略乘法常数：常数因子不会影响增长趋势。

忽略非主导项：当 n 充分大时，主导项（如 n²）的增长速度远远超过次要项（如 n），非主导项对整体增长率影响微乎其微。

常数时间复杂度 O(1)

如果一个算法的执行时间与输入规模无关（无论输入多大，执行时间都保持相同），则称其具有常数时间复杂度，用 O(1) 表示。例如：

java

复制

编辑

// 随机访问数组元素

int x = array[123456];

访问任意元素只需根据“数组起始地址 + 索引 × 元素大小”计算出内存地址，这个过程与数组长度无关，因此是 O(1)。

1.1.2 空间复杂度

空间复杂度是指算法在运行过程中所占用的内存空间的大小，包括：

输入数据本身所占空间

算法运行时额外需要的空间

同样可以使用大 O 符号描述空间复杂度。

如果空间复杂度是 O(n)，意味着所需内存与输入规模 n 成线性关系。

如果空间复杂度是 O(1)，表示所需内存空间固定，不随 n 变化。

顺便复习：等比数列求和公式

对于等比数列 a₀, a₁, a₂, …, aₙ，公比 q≠1，有

  Sₙ = a₀ + a₁ + … + aₙ = (a₀ (1 – qⁿ⁺¹)) / (1 – q) = (aⁿ⁺¹ – 1) / (a – 1) （如果首项是 a₀=1、公比 a）

1.1.3 具体分析

下面使用大 O 符号来分析不同结构和常见算法的时间复杂度。

1.1.3.1 Repetition（重复结构）

单层循环

java

复制

编辑

for (int i = 1; i <= n; i++) {

k = k + 5;

}

该循环执行 n 次常数操作，总时间复杂度为 O(n)。

性能测试示例：

java

复制

编辑

public class PerformanceTest {

public static void main(String[] args) {

getTime(1\_000\_000);

getTime(10\_000\_000);

getTime(100\_000\_000);

getTime(1\_000\_000\_000);

}

public static void getTime(long n) {

long startTime = System.currentTimeMillis();

long k = 0;

for (int i = 1; i <= n; i++) {

k = k + 5;

}

long endTime = System.currentTimeMillis();

System.out.println("Execution time for n = " + n

+ " is " + (endTime - startTime) + " milliseconds");

}

}

测试结果（仅示例，结果与机器性能有关）：

n = 1,000,000 执行时间 ≈ 60 毫秒

n = 10,000,000 执行时间 ≈ 600 毫秒

n = 100,000,000 执行时间 ≈ 6,000 毫秒

n = 1,000,000,000 执行时间 ≈ 60,000 毫秒

可以看出：当 n 增加 10 倍时，执行时间大致也增加 10 倍，符合 O(n) 的线性增长。

双重嵌套循环

java

复制

编辑

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++) {

k = k + i + j;

}

}

外层循环 n 次，内层循环 n 次，总共执行约 n × n = n² 次操作，因此时间复杂度为 O(n²)。

同时，可以通过求和验证：

T(n) = c + 2c + 3c + … + n c = c·n(n + 1) / 2 = (c/2)·n² + (c/2)·n = O(n²)。

内循环为常数次数的嵌套

java

复制

编辑

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= 20; j++) {

k = k + i + j;

}

}

外层循环 n 次，内层始终 20 次，总操作数约为 20·n，即 O(n)。常数 20 可忽略。

1.1.3.2 Sequence（序列结构）

java

复制

编辑

for (int i = 1; i <= n; i++) {

k = k + 5; // O(n)

}

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= 20; j++) {

k = k + i + j; // O(n) × 20 = O(n)

}

}

序列结构是先执行第一个 for（O(n)），然后再执行第二个 for（O(n)）。两者相加仍是 O(n + n) = O(n)。

1.1.3.3 Selection（选择结构）

java

复制

编辑

if (list.contains(e))

System.out.println(e);

else

for (Object t : list)

System.out.println(t);

list.contains(e) 最坏情况下需遍历整个列表，复杂度 O(n)。

如果 contains 返回 false，则进入 else，再次遍历列表并输出，O(n)。

最坏情况下，总时间 T(n) = O(n) + O(n) = O(2n) = O(n)。

1.1.3.4 Logarithm（对数结构）

幂运算的线性算法

java

复制

编辑

result = 1;

for (int i = 1; i <= n; i++) {

result \*= a; // 计算 a^n，共执行 n 次相乘

}

时间复杂度为 O(n)。

指数翻倍优化（对数时间）

假设 n = 2ᵏ，则 k = log₂ n。可以按如下方式计算 aⁿ：

java

复制

编辑

result = a;

for (int i = 1; i <= k; i++) {

result = result \* result; // 每次将指数翻倍

}

循环执行 k 次，即 O(log n)。

二分查找（经典对数时间算法）

java

复制

编辑

public static int binarySearch(int[] list, int key) {

int low = 0;

int high = list.length - 1;

while (high >= low) {

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; // 未找到

}

每次迭代将搜索范围减半，最多需要 log₂(n) 次迭代，因此时间复杂度为 O(log n)。

对数时间算法在处理大规模数据时特别高效，因为随着输入规模增长，所需额外时间增长非常缓慢。

1.1.3.5 二次算法（Quadratic Algorithm）

选择排序（Selection Sort）

java

复制

编辑

public static void selectionSort(double[] list) {

for (int i = 0; i < list.length; i++) {

double currentMin = list[i];

int currentMinIndex = i;

for (int j = i + 1; j < list.length; j++) {

if (currentMin > list[j]) {

currentMin = list[j];

currentMinIndex = j;

}

}

if (currentMinIndex != i) {

list[currentMinIndex] = list[i];

list[i] = currentMin;

}

}

}

比较次数是 (n – 1) + (n – 2) + … + 1 = n(n – 1)/2 = O(n²)。

交换和其它操作也是 O(n²) 级别，综合来看总时间复杂度仍为 O(n²)。

插入排序（Insertion Sort）

java

复制

编辑

public static void insertionSort(int[] list) {

for (int i = 1; i < list.length; i++) {

int currentElement = list[i];

int k;

for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {

list[k + 1] = list[k];

}

list[k + 1] = currentElement;

}

}

最坏情况下（每次都要把元素插入到最前面），第 i 步需要 i 次比较与移动，

总共比较和移动次数约为 2 × ((n – 1) + (n – 2) + … + 1) = n(n – 1)，因此为 O(n²)。

对于时间复杂度为 O(n²) 的算法（称为二次算法），当输入规模翻倍时，执行时间大约增加 4 倍。

1.1.3.6 多项式时间复杂度（Polynomial Time Complexity）

如果算法的运行时间可以被输入规模 n 的某个常数次幂多项式表达式所上界（即 T(n) = O(nᵏ)，k 为常数），则称为多项式时间算法。

P 类（Polynomial Time）：

可以在确定性图灵机上以多项式时间解决的决策问题集合。

NP 类（Nondeterministic Polynomial Time）：

可以在非确定性图灵机上以多项式时间解决的决策问题集合。若一个问题属于 NP，则对于任意给定的“候选解”，可以在多项式时间内验证其正确性。

NP-难（NP-hard）：

与 NP 中最难问题一样难或更难的问题。

NP-完全（NP-complete）：

同时属于 NP 且 NP-难的问题。即：

该问题属于 NP，

所有 NP 问题都可以在多项式时间内归约到该问题。

目前尚未证明 P 是否等于 NP（P = NP 问题）。

常见的 NP-完全问题示例：

布尔可满足性问题（SAT）

背包问题（Knapsack）

哈密顿路径（Hamiltonian Path）

旅行商问题（Traveling Salesman Problem, TSP）

图着色（Graph Coloring）

子图同构（Subgraph Isomorphism）

子集和（Subset Sum）

团问题（Clique）

顶点覆盖（Vertex Cover）

独立集（Independent Set）

支配集（Dominating Set）

1.1.3.6.1 3-SAT 问题

3-SAT 是 SAT 问题的一个特例，公式以合取范式（CNF）给出，且每个子句最多包含 3 个文字，例如：

scss

复制

编辑

(l₁ ∨ l₂ ∨ l₃) ∧ (l₄ ∨ l₅ ∨ l₆) ∧ … ∧ (l\_{n–2} ∨ l\_{n–1} ∨ l\_n)

其中，每个 lᵢ 是布尔变量或它的取反。3-SAT 是 NP-完全问题。

1.1.3.7 指数算法（Exponential Algorithm）

汉诺塔问题（Towers of Hanoi）

规则：

有 n 个盘子，编号 1（最小）到 n（最大），全部初始放在塔 A。

目标是将所有盘子移动到塔 C，过程中只能移动顶端的盘子，且不能把大盘子放在小盘子上面。

递归解法会导致移动次数满足：

r

复制

编辑

T(n) = 2·T(n – 1) + 1, T(1) = 1

=> T(n) = 2ⁿ – 1

因此时间复杂度为 O(2ⁿ)。

java

复制

编辑

import java.util.Scanner;

public class TowersOfHanoi {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Enter number of disks: ");

int n = input.nextInt();

System.out.println("The moves are:");

moveDisks(n, 'A', 'B', 'C');

}

public static void moveDisks(int n, char fromTower, char toTower, char auxTower) {

if (n == 1) {

System.out.println("Move disk 1 from " + fromTower + " to " + toTower);

} else {

moveDisks(n - 1, fromTower, auxTower, toTower);

System.out.println("Move disk " + n + " from " + fromTower + " to " + toTower);

moveDisks(n - 1, auxTower, toTower, fromTower);

}

}

}

若每秒移动一个盘子：

n = 32 时需要 2³² – 1 秒 ≈ 136 年；

n = 64 时需要 2⁶⁴ – 1 秒 ≈ 5,850 亿年。

1.1.3.8 递归算法（Recursive Algorithm）

斐波那契数列（Fibonacci）

朴素递归版本

java

复制

编辑

public static int fib(int index) {

if (index == 0) // 基本情况

return 0;

else if (index == 1) // 基本情况

return 1;

else

return fib(index - 1) + fib(index - 2);

}

其递归调用次数大致满足 T(n) = T(n–1) + T(n–2) + c，因此时间复杂度为 O(2ⁿ)。随着 n 增加，递归调用次数呈指数增长。

带“步骤计数”演示代码：

java

复制

编辑

import java.util.Scanner;

public class Fibonacci2 {

static int steps = 0;

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Enter the Fibonacci index: ");

int n = input.nextInt();

System.out.println("fib(" + n + ") = " + fib(n));

System.out.println("steps: " + steps);

}

public static int fib(int index) {

steps++;

if (index == 0)

return 0;

else if (index == 1)

return 1;

else

return fib(index - 1) + fib(index - 2);

}

}

动态规划优化版本（非递归）

java

复制

编辑

import java.util.Scanner;

public class Fibonacci3 {

static int steps = 0;

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Enter the Fibonacci index: ");

int n = input.nextInt();

System.out.println("fib(" + n + ") = " + fib(n));

System.out.println("steps: " + steps);

}

public static int fib(int n) {

if (n == 0)

return 0;

else if (n == 1 || n == 2)

return 1;

int f0 = 0; // fib(0)

int f1 = 1; // fib(1)

int f2 = 1; // fib(2)

steps = 2; // 已知 fib(1)、fib(2)

for (int i = 3; i <= n; i++) {

steps++;

f0 = f1;

f1 = f2;

f2 = f0 + f1;

}

return f2;

}

}

动态规划只需一次循环，时间复杂度变为 O(n)，大大提升了效率。

1.1.3.9 通过递推关系计算复杂度

在许多递归算法中，可以通过写出递推式（如 T(n) = 2·T(n/2) + O(n)）来推导时间复杂度。具体例子可参考归并排序、快速排序等经典算法，其复杂度通常由主定理（Master Theorem）分析得出。

1.1.3.10 时间复杂度的比较

scss

复制

编辑

O(1) < O(log n) < O(n) < O(n log n) < O(n²) < O(n³) < O(2ⁿ)

从左到右随着 n 增大，增长速度依次变快。

1.2 算法设计

算法设计一般包含以下步骤：

问题定义（Problem definition）

开发模型（Model construction）

算法规范（Specification）

设计算法（Design）

检查算法正确性（Correctness proof / Verification）

算法分析（Complexity analysis）

算法实现（Implementation）

程序测试（Testing）

文档编写（Documentation）

1.2.1 算法设计技术（Algorithm Techniques）

常见的设计技术包括：

暴力搜索（Brute-force or Exhaustive Search）

尝试所有可能的解法，以找到最优解。

分治法（Divide and Conquer）

将大问题分解为更小的子问题，递归求解后合并结果。例如归并排序、快速排序、最近点对问题等。

动态规划（Dynamic Programming）

通过存储子问题的结果（记忆化或自底向上）来避免重复计算，典型例子是斐波那契数列、背包问题、最长公共子序列等。

贪心算法（Greedy Algorithms）

在每一步选择局部最优解，希望最终得到全局最优。典型例子：活动选择、最短路径（Dijkstra）、最小生成树（Kruskal、Prim）等。但贪心不保证在所有情况下都能找到最优解，仅用于适合贪心策略的问题。

回溯算法（Backtracking）

通过试探所有可能的选择路径，当某条路径不满足约束时“回溯”到上一层继续尝试。例如八皇后问题、数独、子集划分等。

1.2.1.1 动态规划 vs 递归

递归：直接从顶层问题出发，递归地求解子问题，但子问题可能会重复计算。

动态规划：保存已经计算过的子问题结果，避免重复计算，从而提升效率。

1.2.2 最大公约数（Greatest Common Divisor, GCD）

暴力搜索版本

java

复制

编辑

public static int gcd(int m, int n) {

int gcd = 1;

for (int k = 2; k <= m && k <= n; k++) {

if (m % k == 0 && n % k == 0)

gcd = k;

}

return gcd;

}

时间复杂度为 O(min(m, n))，约 O(n)。

从较小数向下寻找

java

复制

编辑

public static int gcd(int m, int n) {

int gcd = 1;

for (int k = Math.min(m, n); k >= 1; k--) {

if (m % k == 0 && n % k == 0) {

gcd = k;

break;

}

}

return gcd;

}

虽然在找到第一个公约数后即可终止，但最坏情况下仍需 O(n) 次判断。

从 n/2 开始向下

java

复制

编辑

public static int gcd(int m, int n) {

if (m == n) return m;

int gcd = 1;

for (int k = n / 2; k >= 1; k--) {

if (m % k == 0 && n % k == 0) {

gcd = k;

break;

}

}

return gcd;

}

最坏时间复杂度依旧 O(n)。

1.2.2.1 欧几里得算法（Euclid’s Algorithm）

欧几里得算法利用了“若 p 为 m、n 的公约数，则 p 也为 n 和 m mod n 的公约数”这一性质。

java

复制

编辑

public static int gcd(int m, int n) {

if (m % n == 0)

return n;

else

return gcd(n, m % n);

}

假设 m ≥ n，那么 m mod n < m/2；下一次调用时参数规模至少减半。递归深度约为 O(log n)，因此时间复杂度为 O(log n)。

完整示例：

java

复制

编辑

import java.util.Scanner;

public class GCD {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Enter two numbers: ");

int n1 = input.nextInt();

int n2 = input.nextInt();

System.out.println("gcd(" + n1 + ", " + n2 + ") = " + gcd(n1, n2));

}

public static int gcd(int m, int n) {

if (m % n == 0)

return n;

else

return gcd(n, m % n);

}

}

1.2.3 寻找素数（Finding Prime Numbers）

暴力搜索（到 n – 1）

对每个 i，从 2 到 i – 1 试除，看是否能整除 i。这种方法时间复杂度为 O(n²)。

试除到 √n（所有数）

对每个 i，只需要测试 2 到 √i 是否能整除。由于 √i ≤ √n，最坏情况下总判断约 ∑\_{i=2}^{n} √i = O(n√n)。

试除到 √n（仅素数）

首先生成小于等于 √n 的素数表 P，然后用 P 中的所有素数去试除 i，这样可省掉非素数的试除操作，时间复杂度约 O(n√n / log n)。

1.2.3.1 埃拉托斯特筛法（Sieve of Eratosthenes）

创建一个长度为 n+1 的布尔数组primes[]，初始全部置为 true（假设都为素数）。

从最小素数 2 开始，如果 primes[k] 为 true，则将 k 的所有倍数标记为 false。

重复直到 k > √n。

最终 primes[i] == true 的 i 即为素数。

java

复制

编辑

import java.util.Scanner;

public class SieveOfEratosthenes {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Find all prime numbers <= n, enter n: ");

int n = input.nextInt();

boolean[] primes = new boolean[n + 1];

for (int i = 0; i <= n; i++) {

primes[i] = true;

}

for (int k = 2; k <= n / k; k++) {

if (primes[k]) {

for (int i = k \* k; i <= n; i += k) {

primes[i] = false;

}

}

}

final int NUMBER\_PER\_LINE = 10;

int count = 0;

for (int i = 2; i <= n; i++) {

if (primes[i]) {

count++;

if (count % NUMBER\_PER\_LINE == 0) {

System.out.printf("%7d\n", i);

} else {

System.out.printf("%7d ", i);

}

}

}

System.out.println("\n" + count + " prime(s) less than or equal to " + n);

}

}

时间复杂度分析

对于每个素数 k ≤ √n，需要将 k²、k² + k、k² + 2k … k·⌊n/k⌋ 标记为非素数。总工作量近似

css

复制

编辑

∑\_{p prime ≤ √n} ⌊n / p⌋

由于小于等于 √n 的素数约有 π(√n) ≈ √n / log(√n)，因此粗略上界为 O(n · π(√n)) ≈ O(n·√n / log n)。但实际效率远高于这个松散上界，通常认为筛法几乎接近 O(n log log n)。

1.2.4 “最近点对”问题（Closest-Pair Problem）

暴力算法

对所有点对 i < j，计算距离并取最小值。n 个点有 n(n – 1)/2 对，时间复杂度为 O(n²)。

分治算法（O(n log n)）

先按 x 坐标对所有点排序，O(n log n)。

递归将点集分成左右两半，分别求最近点对，记距离为 d₁、d₂，令 d = min(d₁, d₂)。

构造跨中线的“窄带”：所有点的 x 坐标位于 [mid\_x – d, mid\_x + d] 内，按 y 坐标排序，仅需检查每个点与后面最多 7 个点的距离。总体合并步骤 O(n)。

递归关系：T(n) = 2·T(n/2) + O(n)，解得 T(n) = O(n log n)。

以下是核心思路示意伪代码：

java

复制

编辑

double closestPair(Point[] ptsSortedByX, Point[] ptsSortedByY) {

if (n <= 3) {

// 直接计算暴力最近点对

}

int mid = n / 2;

Point midPoint = ptsSortedByX[mid];

// 分配左右两半的点集，并分别按 y 排序

double d1 = closestPair(leftPtsX, leftPtsY);

double d2 = closestPair(rightPtsX, rightPtsY);

double d = Math.min(d1, d2);

List<Point> strip = new ArrayList<>();

for (Point p : ptsSortedByY) {

if (Math.abs(p.x - midPoint.x) < d) {

strip.add(p);

}

}

// 仅需在 strip 中对每个点检查后面最多 7 个点的距离

double dStrip = stripClosest(strip, d);

return Math.min(d, dStrip);

}

1.2.5 八皇后问题（Eight Queens Problem）

目标：在 8×8 国际象棋棋盘上放置 8 个皇后，使得任意两个皇后不在同一行、同一列或同一对角线上。

回溯算法思路：

从第 0 行开始，尝试将皇后放在第 0 行的某一列。

若放置不会与前面已放置的皇后冲突，则递归进入下一行。

若当前行所有列都无法放置，则回溯到上一行，尝试该行的下一个位置。

若成功在第 7 行放置皇后，则找到一个解。

java

复制

编辑

public class NQueens {

static final int SIZE = 8;

private int[] queens = new int[SIZE]; // queens[i] = j 表示第 i 行的皇后放在第 j 列

public static void main(String[] args) {

NQueens nq = new NQueens();

if (nq.search(0)) {

nq.printSolution();

} else {

System.out.println("No solution found.");

}

}

// 尝试在 row 行放置皇后

private boolean search(int row) {

if (row == SIZE) {

return true; // 所有行已成功放置

}

for (int col = 0; col < SIZE; col++) {

if (isValid(row, col)) {

queens[row] = col;

if (search(row + 1)) {

return true;

}

// 否则回溯，尝试下一个 col

}

}

return false; // 无法在该行放置皇后，回溯

}

// 判断 (row, col) 放置皇后是否与之前放置的冲突

private boolean isValid(int row, int col) {

for (int i = 0; i < row; i++) {

int placedCol = queens[i];

// 同列或同对角线

if (placedCol == col || Math.abs(placedCol - col) == Math.abs(i - row)) {

return false;

}

}

return true;

}

private void printSolution() {

for (int i = 0; i < SIZE; i++) {

System.out.print("[");

for (int j = 0; j < SIZE; j++) {

if (queens[i] == j) System.out.print("Q ");

else System.out.print(". ");

}

System.out.println("]");

}

}

}

时间复杂度最坏情况下接近 O(n!)，但由于回溯早期会剪枝，8 皇后实际运行很快。

1.2.6 凸包（Convex Hull）

给定一组平面点，凸包是能包围所有点的最小凸多边形。应用于模式识别、图像处理等领域。

1.2.6.1 礼品包装算法（Gift-Wrapping / Jarvis March）

在所有点中选取 x 坐标最小（或最右且最低）的点 h₀，加入凸包列表 H。

令当前点 t₀ = h₀，从所有其他点中选取相对于 t₀ 所在直线最“左侧”的点 t₁，加入 H。

令 t₀ = t₁，重复步骤 2，直到再次回到 h₀ 为止。

判断点 p₂ = (x₂, y₂) 是否在从 p₀ = (x₀, y₀) 到 p₁ = (x₁, y₁) 的直线左侧，可计算叉乘：

复制

编辑

(x₁ – x₀)·(y₂ – y₀) – (x₂ – x₀)·(y₁ – y₀) > 0 → 在左侧

= 0 → 在直线上

< 0 → 在右侧

每一轮寻找最左点时要遍历所有其它点，若凸包顶点数为 h，总共有 h 轮，每轮 O(n)，总体 O(h·n)。

最坏情况 h = n（所有点都在凸包上），时间复杂度 O(n²)。

1.2.6.2 Graham 扫描算法（Graham Scan）

从点集中选取 y 坐标最低（若并列则 x 最大）的点 p₀ 作为基准点。

以 p₀ 为中心，对其余 n – 1 个点按与 p₀ 连线的极角从小到大排序（若极角相同，则保留距离 p₀ 最近的一个）。排序 O(n log n)。

初始化栈 H，依次将 p₀, p₁, p₂ 推入栈中。

对 i 从 3 到 n – 1：令 t₁, t₂ 为栈顶两个元素。（t₂ 次栈顶，t₁ 顶）

若 pᵢ 在从 t₂ 到 t₁ 的直线左侧（判断同叉乘），则将 pᵢ 推入栈中（说明拐向正确）。

否则弹出栈顶元素，继续与新的栈顶比较，直到满足左侧为止。

最后栈中剩余的点按顺序即为凸包顶点。

总时间复杂度 = 排序 O(n log n) + 单调栈处理 O(n) = O(n log n)。

1.3 复杂度类与比较

O(1) < O(log n) < O(n) < O(n log n) < O(n²) < O(n³) < O(2ⁿ) < O(nⁿ)

O(1)：常数时间，最优。

O(log n)：对数时间，如二分查找。

O(n)：线性时间，如简单循环。

O(n log n)：典型的排序算法（归并、快速、堆等）。

O(n²)：嵌套两层循环，如选择排序、插入排序（最坏情形）。

O(n³)：三重嵌套循环，少见于小规模问题。

O(2ⁿ)：指数时间，典型的“穷举”回溯或递归无剪枝算法。

O(nⁿ)：极度不可行，几乎不可能在现实中处理。

八！！！！！！！！！！！！！（SORTING 排序算法）

1. 排序算法

我们之前的学习已经对所有排序方法都有了了解，这里稍微复习一下。

1.1 冒泡排序（Bubble sort）

冒泡排序是一种简单的排序算法，它重复地遍历待排序的列表，比较每一对相邻的元素，如果它们的顺序错误就交换它们。每次遍历都会让最大的元素“冒泡”到列表的末尾。这个过程会重复进行，直到整个列表有序。

代码如下。

public class BubbleSort {

public static void bubbleSort(int[] list) {

for (int k = 1; k < list.length; k++) { // 外层循环控制趟数

for (int i = 0; i < list.length - k; i++) { // 内层循环比较相邻元素

if (list[i] > list[i + 1]) { // 如果当前元素大于后面的元素

// 交换它们

int temp = list[i];

list[i] = list[i + 1];

list[i + 1] = temp;

}

}

}

}

public static void main(String[] args) {

int[] list = {64, 34, 25, 12, 22, 11, 90};

bubbleSort(list); // 调用冒泡排序

System.out.println("Sorted list:");

for (int num : list) {

System.out.print(num + " ");

}

}

}

如果列表几乎已经排序好了，或者已经完全排序好了，传统的冒泡排序算法仍然会进行不必要的遍历和比较，这会浪费计算资源。

下面的代码会引入一个布尔变量 needNextPass 来检测数组是否可能已经排序好了，从而避免不必要的遍历。

public class BubbleSort {

public static void bubbleSort(int[] list) {

boolean needNextPass = true;

for (int k = 1; k < list.length && needNextPass; k++) {

needNextPass = false; // 假设这一轮不需要再排序

for (int i = 0; i < list.length - k; i++) {

if (list[i] > list[i + 1]) {

int temp = list[i];

list[i] = list[i + 1];

list[i + 1] = temp;

needNextPass = true; // 发生了交换，说明还需要继续排序

}

}

}

}

public static void main(String[] args) {

int[] list = {1, 2, 3, 4};

bubbleSort(list);

System.out.println("Sorted list:");

for (int num : list) {

System.out.print(num + " ");

}

}

}

冒泡排序的时间复杂度再最佳情况下是O ( n ) O(n)O(n)，在最坏情况下（数组完全逆序）是O ( n 2 ) O(n^2)O(n

2

)。

1.2 归并排序（Merge Sort）

归并排序使用分治法（Divide and Conquer）策略，因此详细步骤如下：

分解（Divide）：

将数组分成两半。如果数组的长度是奇数，那么可以将中间的元素单独处理，或者将其归入左边或右边的子数组。

这个过程是递归进行的，即对每个子数组继续进行分解，直到每个子数组只包含一个元素。由于只有一个元素的数组自然是有序的，所以这是递归的终止条件。

解决（Conquer）：

对每个子数组进行排序。由于子数组是由原始数组分解而来，且每个子数组只包含一个元素，所以这一步实际上是在递归地对子数组进行排序。

合并（Combine）：

将排序好的子数组合并成一个有序数组。这一步是归并排序的核心，需要将两个已排序的子数组合并成一个有序数组。

合并过程是通过比较两个子数组的首元素，将较小的元素放入新数组中，然后从相应的子数组中移除该元素，重复这个过程直到两个子数组中的元素都被合并到新数组中。

代码如下。

public class MergeSortTest {

public static void mergeSort(int[] list) {

if (list.length > 1) {

// 分解第一个子数组

int[] firstHalf = new int[list.length / 2];

System.arraycopy(list, 0, firstHalf, 0, list.length / 2);

mergeSort(firstHalf); // 递归排序第一个子数组

// 分解第二个子数组

int secondHalfLength = list.length - list.length / 2;

int[] secondHalf = new int[secondHalfLength];

System.arraycopy(list, list.length / 2, secondHalf, 0, secondHalfLength);

mergeSort(secondHalf); // 递归排序第二个子数组

// 合并两个子数组到原数组

merge(firstHalf, secondHalf, list);

}

}

public static void merge(int[] list1, int[] list2, int[] list) {

int current1 = 0; // 当前索引在 list1 中

int current2 = 0; // 当前索引在 list2 中

int current3 = 0; // 当前索引在 list 中

// 合并两个子数组

while (current1 < list1.length && current2 < list2.length) {

if (list1[current1] < list2[current2]) {

list[current3++] = list1[current1++];

} else {

list[current3++] = list2[current2++];

}

}

// 复制剩余的元素（如果有的话）

while (current1 < list1.length) {

list[current3++] = list1[current1++];

}

while (current2 < list2.length) {

list[current3++] = list2[current2++];

}

}

public static void main(String[] args) {

int size = 100000;

int[] a = new int[size];

randomInitiate(a);

long startTime = System.currentTimeMillis();

mergeSort(a);

long endTime = System.currentTimeMillis();

System.out.println((endTime - startTime) + "ms");

}

private static void randomInitiate(int[] a) {

for (int i = 0; i < a.length; i++) {

a[i] = (int) (Math.random() \* a.length);

}

}

}

归并排序的递归关系式为：

T ( n ) = T ( n / 2 ) + T ( n / 2 ) + 2 n − 1 T(n)=T(n/2)+T(n/2)+2n−1T(n)=T(n/2)+T(n/2)+2n−1

其中：

第一个T ( n / 2 ) T(n/2)T(n/2)表示对数组的前半部分进行排序所需的时间。

第二个T ( n / 2 ) T(n/2)T(n/2) 表示对数组的后半部分进行排序所需的时间。

2 n − 1 2n−12n−1表示合并两个已排序子数组所需的时间，包括n − 1 n−1n−1次比较（用于比较两个子数组的元素）和n nn次移动（将每个元素放入临时数组）。

因此归并排序的时间复杂度为O ( n l o g n ) O(nlogn)O(nlogn)。

1.3 快速排序（Quick Sort）

快速排序也是使用分治法（Divide and Conquer）策略来实现排序。

算法从数组中选择一个元素，称为基准（pivot）。

将数组分成两部分：

第一部分（list1）包含所有小于或等于基准的元素。

第二部分（list2）包含所有大于基准的元素。

这个过程称为分区（partitioning）。

递归排序：

对第一部分（list1）和第二部分（list2）分别递归地应用快速排序算法，直到每个子数组只包含一个元素或为空。

代码如下。

public class QuickSortTest {

public static void quickSort(int[] list) {

quickSort(list, 0, list.length - 1);

}

public static void quickSort(int[] list, int first, int last) {

if (last > first) {

int pivotIndex = partition(list, first, last);

quickSort(list, first, pivotIndex - 1);

quickSort(list, pivotIndex + 1, last);

}

}

public static int partition(int[] list, int first, int last) {

int pivot = list[first]; // Choose the first element as pivot

int low = first + 1; // Index for forward search

int high = last; // Index for backward search

while (high > low) {

// Search forward from left

while (low <= high && list[low] <= pivot)

low++;

// Search backward from right

while (low <= high && list[high] > pivot)

high--;

// Swap two elements in the list

if (high > low) {

int temp = list[high];

list[high] = list[low];

list[low] = temp;

}

}

// Account for duplicated elements:

while (high > first && list[high] >= pivot)

high--;

// Swap pivot with list[high]

if (pivot > list[high]) {

list[first] = list[high];

list[high] = pivot;

return high;

} else {

return first;

}

}

public static void main(String[] args) {

int size = 100000;

int[] a = new int[size];

randomInitiate(a);

long startTime = System.currentTimeMillis();

quickSort(a);

long endTime = System.currentTimeMillis();

System.out.println((endTime - startTime) + "ms");

}

private static void randomInitiate(int[] a) {

for (int i = 0; i < a.length; i++)

a[i] = (int) (Math.random() \* a.length);

}

}

快速排序的递归关系式为：

T ( n ) = T ( n / 2 ) + T ( n / 2 ) + n T(n)=T(n/2)+T(n/2)+nT(n)=T(n/2)+T(n/2)+n

其中：

第一个T ( n / 2 ) T(n/2)T(n/2)表示对数组的前半部分进行排序所需的时间。

第二个T ( n / 2 ) T(n/2)T(n/2) 表示对数组的后半部分进行排序所需的时间。

n nn表示分区所需的时间。

因此快速排序的时间复杂度为O ( n l o g n ) O(nlogn)O(nlogn)。

然而在最坏情况下，每次分区都将数组分成一个非常大的子数组和一个空的子数组。如，假设数组的第一个元素是分区的基准：

[ 1 , 2 , 3 , 4 , 5 , … , n ] , s i z e = n [1,2,3,4,5,…,n], size=n[1,2,3,4,5,…,n],size=n

第一次分区，基准为1 11：

左侧：空，右侧：[ 2 , 3 , 4 , 5 , … , n ] [2, 3, 4, 5, …, n][2,3,4,5,…,n]，右侧子数组大小为n − 1 n−1n−1。

第二次分区，基准为2 22：

左侧：空，右侧：[ 3 , 4 , 5 , 6 , … , n ] [3, 4, 5, 6, …, n][3,4,5,6,…,n]，右侧子数组大小为n − 2 n−2n−2。

以此类推，直到子数组大小为1 11。

由于每次分区都将数组分成一个非常大的子数组和一个空的子数组，因此需要递归地对n − 1 n−1n−1个子数组进行分区操作。

每个分区操作的时间复杂度为O ( n ) O(n)O(n)，因此总的时间复杂度为O ( n 2 ) O(n^2)O(n

2

)。

1.4 堆排序（Heap Sort）

这个算法在本学期的算法课上详细介绍过。可以看这里的文章。

相关文章

在堆排序算法中，通常使用完全二叉树（Complete Binary Tree）来实现堆结构。完全二叉树是一种特殊的二叉树，其中除了最后一层外，每一层都是满的，并且最后一层的节点尽可能地集中在左侧。

堆排序算法通过维护一个堆结构来实现排序，堆结构有两种类型：

最大堆（Max Heap）：每个节点的值都大于或等于其子节点的值。

最小堆（Min Heap）：每个节点的值都小于或等于其子节点的值。

堆可以存储在 ArrayList 或数组中，前提是堆的大小是已知的。

对于数组中位置为i ii的节点，其左子节点的位置是2 i + 1 2i + 12i+1，右子节点的位置是2 i + 2 2i + 22i+2，父节点的位置是 ( i − 1 ) / 2 (i - 1) / 2(i−1)/2（注意使用整数除法）。

以位置为4 44的节点为例，其两个子节点的位置分别是2 ∗ 4 + 1 = 9 2 \* 4 + 1 = 92∗4+1=9和2 ∗ 4 + 2 = 10 2 \* 4 + 2 = 102∗4+2=10，其父节点的位置是（ 4 − 1 ） / 2 = 1 （4 - 1）/ 2 = 1（4−1）/2=1（注意这里使用的是整数除法，结果为1 11，而不是1.5 1.51.5）。

在堆排序中，通常使用最大堆来实现。算法的基本步骤如下：

将待排序的数组转换为最大堆。

将堆顶元素（最大值）与数组的最后一个元素交换，然后将数组缩小一个元素。

重新调整堆，使其满足最大堆的性质。

重复步骤 2 和 3，直到数组完全有序。

因此实现这个算法需要依赖于堆（Heap）这种数据结构。

其UML图如下。

代码如下。

import java.util.ArrayList;

public class Heap<E extends Comparable> {

private ArrayList<E> list = new ArrayList<E>();

// Create a default heap

public Heap() {

}

// Create a heap from an array of objects

public Heap(E[] objects) {

for (int i = 0; i < objects.length; i++)

add(objects[i]);

}

// Add a new object into the heap

public void add(E newObject) {

list.add(newObject); // Append to the end of the heap

int currentIndex = list.size() - 1; // The index of the last node

while (currentIndex > 0) {

int parentIndex = (currentIndex - 1) / 2;

// Swap if the current object is greater than its parent

if (list.get(currentIndex).compareTo(list.get(parentIndex)) > 0) {

E temp = list.get(currentIndex);

list.set(currentIndex, list.get(parentIndex));

list.set(parentIndex, temp);

} else

break; // the tree is a heap now

currentIndex = parentIndex;

}

}

// Remove the root from the heap

public E remove() {

if (list.size() == 0) return null;

E removedObject = list.get(0);

list.set(0, list.get(list.size() - 1));

list.remove(list.size() - 1);

int currentIndex = 0;

while (currentIndex < list.size()) {

int leftChildIndex = 2 \* currentIndex + 1;

int rightChildIndex = 2 \* currentIndex + 2;

// Find the maximum between two children

if (leftChildIndex >= list.size())

break; // The tree is a heap

int maxIndex = leftChildIndex;

if (rightChildIndex < list.size())

if (list.get(maxIndex).compareTo(list.get(rightChildIndex)) < 0)

maxIndex = rightChildIndex;

// Swap if the current node is less than the maximum

if (list.get(currentIndex).compareTo(list.get(maxIndex)) < 0) {

E temp = list.get(maxIndex);

list.set(maxIndex, list.get(currentIndex));

list.set(currentIndex, temp);

currentIndex = maxIndex;

} else

break; // The tree is a heap

}

return removedObject;

}

// Get the number of nodes in the tree

public int getSize() {

return list.size();

}

}

所以堆排序的完整代码如下。

import java.util.ArrayList;

public class HeapSort {

public static <E extends Comparable> void heapSort(E[] list) {

// Create a Heap of E

Heap<E> heap = new Heap<E>();

// Add elements to the heap

for (int i = 0; i < list.length; i++)

heap.add(list[i]);

// Remove the highest elements from the heap

// and store them in the list from end to start

for (int i = list.length - 1; i >= 0; i--)

list[i] = heap.remove();

}

/\*\* A test method \*/

public static void main(String[] args) {

Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};

heapSort(list);

for (int i = 0; i < list.length; i++)

System.out.print(list[i] + " ");

}

}

class Heap<E extends Comparable> {

private ArrayList<E> list = new ArrayList<E>();

// Create a default heap

public Heap() {

}

// Create a heap from an array of objects

public Heap(E[] objects) {

for (int i = 0; i < objects.length; i++)

add(objects[i]);

}

// Add a new object into the heap

public void add(E newObject) {

list.add(newObject); // Append to the end of the heap

int currentIndex = list.size() - 1; // The index of the last node

while (currentIndex > 0) {

int parentIndex = (currentIndex - 1) / 2;

// Swap if the current object is greater than its parent

if (list.get(currentIndex).compareTo(list.get(parentIndex)) > 0) {

E temp = list.get(currentIndex);

list.set(currentIndex, list.get(parentIndex));

list.set(parentIndex, temp);

} else

break; // the tree is a heap now

currentIndex = parentIndex;

}

}

// Remove the root from the heap

public E remove() {

if (list.size() == 0) return null;

E removedObject = list.get(0);

list.set(0, list.get(list.size() - 1));

list.remove(list.size() - 1);

int currentIndex = 0;

while (currentIndex < list.size()) {

int leftChildIndex = 2 \* currentIndex + 1;

int rightChildIndex = 2 \* currentIndex + 2;

// Find the maximum between two children

if (leftChildIndex >= list.size())

break; // The tree is a heap

int maxIndex = leftChildIndex;

if (rightChildIndex < list.size())

if (list.get(maxIndex).compareTo(list.get(rightChildIndex)) < 0)

maxIndex = rightChildIndex;

// Swap if the current node is less than the maximum

if (list.get(currentIndex).compareTo(list.get(maxIndex)) < 0) {

E temp = list.get(maxIndex);

list.set(maxIndex, list.get(currentIndex));

list.set(currentIndex, temp);

currentIndex = maxIndex;

} else

break; // The tree is a heap

}

return removedObject;

}

// Get the number of nodes in the tree

public int getSize() {

return list.size();

}

}

堆排序算法的时间复杂度为O ( n l o g n ) O(nlogn)O(nlogn)。

需要一个临时数组来合并两个子数组。这个临时数组的大小与原数组相同，因此归并排序的空间复杂度是 O(n)，其中 n 是数组的长度。

而堆排序不需要额外的数组空间，因此在空间效率上优于归并排序。

2. 在面向对象编程中终身学习

终身学习：指的是持续的、自愿的、自我激励的知识追求，目的是为了个人或职业发展。

在 OOP 领域的应用：终身学习鼓励开发者跟上不断发展的概念、技术和工具，以提高他们在设计、实现和维护软件系统方面的熟练程度。

终身学习的重要性如下：

不断演变的技术环境：

终身学习确保开发者能够跟上最新的最佳实践、框架和设计模式。

提高专业能力：

终身学习增强了你解决复杂问题、优化代码和设计健壮系统的能力。

适应新工具：

学习 OOP 中的新工具使你保持敏捷，并为你打开新的职业机会。

个人成长：

不断提高你的 OOP 技能不仅增强了你的技术专长，还发展了其他宝贵的技能，如批判性思维、解决问题和有效沟通。

2.1 记录和反思学习过程

我们可以记录和反思学习过程。

维护反思日志/记录：

记录每个学习阶段的关键收获。

记录在解决编程挑战后获得的见解和思考。

记录需要进一步澄清的问题或领域。

记录你已经工作过的代码片段和示例。

日志有助于巩固你的学习，并为未来的问题提供参考。

定期自我评估：

定期评估你与设定目标的进展。你是否掌握了基础知识，或者需要重新审视某些概念。

诚实地面对挑战。你是否在复杂的设计模式或高级 OOP 原则的实现上遇到困难？接下来专注于这些领域。

认可你的成就。完成课程或参与 OOP 项目是一个值得庆祝的重要里程碑。

加入社区和论坛：

加入像 StackOverflow、Reddit 、GitHub 或 CSDN 这样的专注于 OOP 的社区。

参与讨论，提问，并与他人分享你的知识。

2.2 EDI 原则

平等（Equality）：

在 OOP 中，平等意味着所有开发者，无论性别、种族、性取向或残疾状况，都有平等的职业机会和参与软件开发过程的能力。

多样性（Diversity）：

在 OOP 中，多样性通过汇集来自不同背景、经验和观点的人来丰富问题解决。这种多样性导致更创造性和创新性的解决方案来应对复杂的软件设计挑战。

包容性（Inclusion）：

在 OOP 中，包容性超越了代表性和平等；它强调创造一个每个人都感到尊重和有价值的环境。

将 EDI 融入 OOP 实践便是：

采用包容性编码实践：

在代码、注释和文档中使用包容性语言。避免可能无意中疏远或排除个体的术语或实践。

实施可访问的设计模式：

确保你构建的软件设计对广泛受众可访问。纳入可访问性功能，如可定制的 UI 选项、语音识别和高对比度主题。

鼓励跨多样化团队的协作：

积极寻求来自不同背景和经验的团队成员。这可能意味着促进科技领域的女性、支持 LGBTQ+ 个体，或与包括来自不同种族和文化的人的团队合作。

利用 EDI 聚焦资源：

利用专注于软件开发社区多样性和包容性的资源、课程和材料。这些资源帮助开发者更意识到编码实践中的潜在偏见，并提供减轻它们的策略。

2.3 人工智能辅助的管理工具进行软件开发

我们可以使用 JIRA 等 AI 辅助管理工具来规划和跟踪软件开发过程中的各种任务和问题。通过这些工具，团队可以更有效地管理项目进度，分配任务，跟踪问题，并确保项目按计划进行。这些工具不仅提高了团队的协作效率，还帮助团队成员更好地理解和执行项目需求。

九！！！！！！！！！！（图 GRAPH）

1. 图

1.1 图的基础知识

我们在算法和数学中都学习过图。

图是一种强大的工具，可以将复杂的现实世界问题抽象化，通过节点（vertices）和边（edges）来表示对象及其关系，从而更容易理解和解决问题。

其经常的应用是分析城市间的最短路径，更清晰地表达是寻找两个城市之间最少航班的问题，可以转化为在图中找到两个顶点之间的最短路径。也可以用于社交媒体分析（建模社交网络）、计算机芯片设计、搜索引擎算法。

图G = ( V , E ) G=(V,E)G=(V,E)

图G GG是由两个集合组成的数学结构：

V VV表示顶点集合（Vertices 或 Nodes），是一组离散的对象。

E EE表示边集合（Edges 或 Links），是一组连接顶点的连接关系。

无向图（Undirected Graph）是一种图，其中的边没有方向。在无向图中，边( x , y ) (x,y)(x,y)和边 ( y , x ) 边 (y,x)边(y,x)是相同的，因为它们表示的是同一个连接关系。

有向图（Directed Graph，也称为“有向图”或“Digraph”）是一种图，其中的边有方向。在有向图中，边( x , y ) (x,y)(x,y)和边( y , x ) (y,x)(y,x)是不同的，因为它们的方向不同。

如果两个顶点通过一条边连接，那么这两个顶点被称为相邻的（Adjacent）或邻居（Neighbors）。

连接两个顶点的边被称为与这两个顶点相关联的（Incident）。

如图中A AA和B BB是相邻的。

度（Degree）：一个顶点的度是指与该顶点相关联的边的数量。

对于无向图，顶点的度就是与它相连的边的数量。例如，如果一个顶点与三条边相连，那么它的度就是3。

对于有向图，顶点的度分为入度（In-degree）和出度（Out-degree）：

入度（In-degree）：指向该顶点的边的数量。

出度（Out-degree）：从该顶点出发的边的数量。

顶点的总度数是入度和出度之和。

下图给出了关于度的无向图例子。

完全图是一种图，其中每一对不同的顶点之间都有一条边直接相连。在完全图中，每个顶点的度都等于顶点总数减一（因为一个顶点不能与自己相连）。

如下图所示。

不完全图是一种图，其中至少存在一对顶点之间没有直接的边相连。

如下图所示，A AA、E EE之间没有直接的边相连。

在无权图（Unweighted）中，边没有与它们关联的权重（weight）。换句话说，图中的所有边都被认为具有相同的“成本”或“距离”。如前面的图都是无权图。

在加权图（Weighted）中，每条边都有一个与之关联的权重，这个权重可以代表成本、距离、时间或其他任何可以量化的度量。

如下图所示。

如果两个顶点之间存在两条或多条边，这些边被称为平行边（Parallel Edges）。

环（Loops）是一种特殊的边，它将一个顶点直接连接到它自己。

简单图（Simple Graph）是一种特殊的图，它不包含任何平行边或环。

下图不是一个简单图，因为他有平行边，也有环，而前面的图都是简单图。

闭合路径（Closed Path）是一种路径，其中所有顶点都恰好有两条边与之相关联（即每个顶点的度数为2）。

闭合路径是一条从某个顶点出发，经过一系列其他顶点后，最终回到起始顶点的路径。在闭合路径中，除了起始和结束顶点外，其他所有顶点都恰好连接两条边，这两条边分别指向路径的前后两个顶点。

环（Cycle）是一种特殊的闭合路径，它从同一个顶点开始并在同一个顶点结束。

环是一条起点和终点相同的路径，它不包含重复的边，除非起点和终点是同一个顶点。环的长度至少为3（即至少包含三个顶点），因为至少需要三个顶点才能形成一个环（一个顶点到自己的环没有实际意义）。

如果图中任意两个顶点之间都存在至少一条路径，那么这个图被称为连通图（Connected Graph）。

前面展示的图都是连通图。

树是一种特殊的连通图，它满足以下两个条件：

连通性：图中任意两个顶点之间都存在至少一条路径，即图是连通的。

无环性：图中不包含任何环，即不存在从某个顶点出发，沿着边走能够回到起始顶点的路径（除了顶点自身形成的自环，但树中不允许有自环）。

下图左侧不是树，因为有环存在，右侧是。

子图（Subgraph）是从一个更大的图（称为原图或超图）中通过选择一部分顶点和边而形成的新图。

如下图所示，这里红色的部分或者蓝色的部分都是整个图的子图。

1.2 图的表示。

图（Graphs）在计算机科学中常见的几种数据结构表示方法如下：

顶点可以用简单的数据结构（如数组、列表或哈希表）来表示，其中每个顶点都有一个唯一的标识符。

边的表示方式有很多：

边数组（Edge Array），使用数组来存储图中所有边。

边对象（Edge Objects），使用对象来表示每条边。

邻接矩阵（Adjacency Matrices），使用二维数组来表示图中顶点之间连接关系。

邻接表（Adjacency Lists），使用列表来表示图中每个顶点的邻接顶点或者邻接边。

1.2.1 顶点的表示

使用数组表示顶点的例子如下：

String[] vertices = {"Seattle", "San Francisco", "Los Angeles", "Denver", "Kansas City", /\* ... \*/};

1

使用列表表示顶点的例子如下：

List<String> vertices;

vertices.add("Seattle");

1

2

使用对象的方法也可以，例子如下：

public class City {

// 定义属性，例如城市名称

private String cityName;

// 定义构造函数

public City(String cityName) {

this.cityName = cityName;

}

// 定义获取城市名称的方法

public String getCityName() {

return cityName;

}

// 定义设置城市名称的方法

public void setCityName(String cityName) {

this.cityName = cityName;

}

}

// 创建City类的实例

City city0 = new City("Seattle");

City city1 = new City("San Francisco");

// 使用City类的实例创建数组

City[] vertices = {city0, city1, /\* ... \*/};

1.2.2 边的表示

1.2.2.1 边数组

我们可以使用使用二维数组表示边（Edge Array），例子如下。

int[][] edges = {

{0, 1}, {0, 3}, {0, 5}, // 从顶点 0 出发的边

{1, 0}, {1, 2}, {1, 3}, // 从顶点 1 出发的边

{2, 1}, {2, 3}, {2, 4}, {2, 10},

{3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},

{4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},

{5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},

{6, 5}, {6, 7}, // 后续边...

};

每对数字，例如 {0, 1}，表示从顶点 0 到顶点 1 存在一条边。

在无权图中，边没有权重，因此 {0, 1} 和 {1, 0} 表示的是同一条边。

1.2.2.2 边对象

我们可以使用边对象（Edge Objects）来表示图中的边，例子如下。

public class Edge {

// 定义边的两个端点

int u, v;

// 构造函数

public Edge(int u, int v) {

this.u = u;

this.v = v;

}

// getter 和 setter 方法

public int getU() {

return u;

}

public void setU(int u) {

this.u = u;

}

public int getV() {

return v;

}

public void setV(int v) {

this.v = v;

}

}

// 创建一个 ArrayList 来存储边对象

List<Edge> list = new ArrayList<>();

list.add(new Edge(0, 1));

list.add(new Edge(0, 3));

使用 ArrayList 存储边对象的一个主要优势是其动态性。你不需要事先知道图中边的数量，可以随着需要动态地添加边对象。

1.2.2.3 邻接矩阵

我们可以使用邻接矩阵表示边（Adjacency Matrix），例子如下。

int[][] adjacencyMatrix = {

{0, 1, 0, 1}, // 西雅图（Seattle）

{1, 0, 1, 1}, // 旧金山（San Francisco）

{0, 1, 0, 1}, // 洛杉矶（Los Angeles）

{1, 1, 1, 0}, // 丹佛（Denver）

};

矩阵中的每个元素[i,j]表示从顶点 i 到顶点 j 是否存在边。

1.2.2.4 邻接顶点列表（Adjacency Vertex List）

我们可以使用邻接顶点列表（Adjacency Vertex List）来表示图中的边，例子如下。

// 创建一个包含12个整数列表的数组，每个列表代表一个城市

List<Integer>[] neighbors = new List[12];

// 向每个城市的邻居列表中添加整数（即城市索引），表示与当前城市相连的邻近城市

neighbors[0].add(1); // 旧金山（San Francisco）与索引为1的城市相邻

neighbors[0].add(3); // 旧金山（San Francisco）与索引为3的城市相邻

neighbors[0].add(5); // 旧金山（San Francisco）与索引为5的城市相邻

neighbors[1].add(0); // 西雅图（Seattle）与索引为0的城市相邻

neighbors[1].add(2); // 西雅图（Seattle）与索引为2的城市相邻

neighbors[1].add(3); // 西雅图（Seattle）与索引为3的城市相邻

邻接顶点列表是一种使用数组来存储每个顶点的邻接顶点的表示方法。每个顶点对应一个列表，列表中包含与该顶点直接相连的所有顶点的索引。

1.2.2.5 邻接边列表（Adjacency Edge List）

与邻接顶点列表不同，邻接边列表存储的是边对象（Edge Objects）而不是顶点索引。

因此需要定义边对象。

public class Edge {

int u; // 边的起点顶点

int v; // 边的终点顶点

// 构造函数，用于初始化边的起点和终点

public Edge(int u, int v) {

this.u = u;

this.v = v;

}

}

然后使用邻接边列表表示边。

// 创建一个包含12个边对象列表的数组，每个列表代表一个顶点的邻接边

List<Edge>[] neighbors = new List[12];

// 向每个顶点的邻接边列表中添加边对象，表示与当前顶点相连的边

neighbors[0].add(new Edge(0, 1)); // 西雅图（Seattle）到旧金山（San Francisco）

neighbors[0].add(new Edge(0, 3)); // 西雅图（Seattle）到丹佛（Denver）

neighbors[0].add(new Edge(0, 5)); // 西雅图（Seattle）到芝加哥（Chicago）

1.3 对图建模

我们现在准备对图进行建模。我们会创建一个名为 Graph 的接口，该接口包含所有图的通用操作。然后是UnweightedGraph 和 WeightedGraph 类，它们实现了 Graph 接口。这些类定义了内部数据结构来存储图的信息，例如使用邻接表（Adjacency List）或邻接矩阵（Adjacency Matrix）来表示图的顶点和边。为了让用户能够从不同类型的输入初始化图，这些类提供了多个构造函数。这些类通过提供具体逻辑来实现 Graph 接口中声明的抽象方法。

代码如下。

public interface Graph<V> {

/\*\*

\* Return the number of vertices in the graph

\*/

public int getSize();

/\*\*

\* Return the vertices in the graph

\*/

public java.util.List<V> getVertices();

/\*\*

\* Return the object for the specified vertex index

\*/

public V getVertex(int index);

/\*\*

\* Return the index for the specified vertex object

\*/

public int getIndex(V v);

/\*\*

\* Return the neighbors of vertex with the specified index

\*/

public java.util.List<Integer> getNeighbors(int index);

/\*\*

\* Return the degree for a specified vertex

\*/

public int getDegree(int v);

/\*\*

\* Print the edges

\*/

public void printEdges();

/\*\*

\* Clear the graph

\*/

public void clear();

/\*\*

\* Add a vertex to the graph

\*/

public boolean addVertex(V vertex);

/\*\*

\* Add an edge (u, v) to the graph

\*/

public boolean addEdge(int u, int v);

/\*\*

\* Add an edge to the graph

\*/

public boolean addEdge(Edge e);

/\*\*

\* Remove a vertex v from the graph, return true if successful

\*/

public boolean remove(V v);

/\*\*

\* Remove an edge (u, v) from the graph

\*/

public boolean remove(int u, int v);

/\*\*

\* Obtain a depth-first search tree

\*/

public UnweightedGraph<V>.SearchTree dfs(int v);

/\*\*

\* Obtain a breadth-first search tree

\*/

public UnweightedGraph<V>.SearchTree bfs(int v);

}

UnweightedGraph类的代码如下。

import java.util.\*;

public class UnweightedGraph<V> implements Graph<V> {

protected List<V> vertices = new ArrayList<>(); // Store vertices

protected List<List<Edge>> neighbors

= new ArrayList<>(); // Adjacency lists

/\*\*

\* Construct an empty graph

\*/

public UnweightedGraph() {

}

/\*\*

\* Construct a graph from vertices and edges stored in arrays

\*/

public UnweightedGraph(V[] vertices, int[][] edges) {

for (int i = 0; i < vertices.length; i++)

addVertex(vertices[i]);

createAdjacencyLists(edges, vertices.length);

}

/\*\*

\* Construct a graph from vertices and edges stored in List

\*/

public UnweightedGraph(List<V> vertices, List<Edge> edges) {

for (int i = 0; i < vertices.size(); i++)

addVertex(vertices.get(i));

createAdjacencyLists(edges, vertices.size());

}

/\*\*

\* Construct a graph for integer vertices 0, 1, 2 and edge list

\*/

public UnweightedGraph(List<Edge> edges, int numberOfVertices) {

for (int i = 0; i < numberOfVertices; i++)

addVertex((V) (new Integer(i))); // vertices is {0, 1, ...}

createAdjacencyLists(edges, numberOfVertices);

}

/\*\*

\* Construct a graph from integer vertices 0, 1, and edge array

\*/

public UnweightedGraph(int[][] edges, int numberOfVertices) {

for (int i = 0; i < numberOfVertices; i++)

addVertex((V) (new Integer(i))); // vertices is {0, 1, ...}

createAdjacencyLists(edges, numberOfVertices);

}

/\*\*

\* Create adjacency lists for each vertex

\*/

private void createAdjacencyLists(

int[][] edges, int numberOfVertices) {

for (int i = 0; i < edges.length; i++) {

addEdge(edges[i][0], edges[i][1]);

}

}

/\*\*

\* Create adjacency lists for each vertex

\*/

private void createAdjacencyLists(

List<Edge> edges, int numberOfVertices) {

for (Edge edge : edges) {

addEdge(edge.u, edge.v);

}

}

@Override

/\*\* Return the number of vertices in the graph \*/

public int getSize() {

return vertices.size();

}

@Override

/\*\* Return the vertices in the graph \*/

public List<V> getVertices() {

return vertices;

}

@Override

/\*\* Return the object for the specified vertex \*/

public V getVertex(int index) {

return vertices.get(index);

}

@Override

/\*\* Return the index for the specified vertex object \*/

public int getIndex(V v) {

return vertices.indexOf(v);

}

@Override

/\*\* Return the neighbors of the specified vertex \*/

public List<Integer> getNeighbors(int index) {

List<Integer> result = new ArrayList<>();

for (Edge e : neighbors.get(index))

result.add(e.v);

return result;

}

@Override

/\*\* Return the degree for a specified vertex \*/

public int getDegree(int v) {

return neighbors.get(v).size();

}

@Override

/\*\* Print the edges \*/

public void printEdges() {

for (int u = 0; u < neighbors.size(); u++) {

System.out.print(getVertex(u) + " (" + u + "): ");

for (Edge e : neighbors.get(u)) {

System.out.print("(" + getVertex(e.u) + ", " +

getVertex(e.v) + ") ");

}

System.out.println();

}

}

@Override

/\*\* Clear the graph \*/

public void clear() {

vertices.clear();

neighbors.clear();

}

@Override

/\*\* Add a vertex to the graph \*/

public boolean addVertex(V vertex) {

if (!vertices.contains(vertex)) {

vertices.add(vertex);

neighbors.add(new ArrayList<Edge>());

return true;

} else {

return false;

}

}

@Override

/\*\* Add an edge to the graph \*/

public boolean addEdge(Edge e) {

if (e.u < 0 || e.u > getSize() - 1)

throw new IllegalArgumentException("No such index: " + e.u);

if (e.v < 0 || e.v > getSize() - 1)

throw new IllegalArgumentException("No such index: " + e.v);

if (!neighbors.get(e.u).contains(e)) {

neighbors.get(e.u).add(e);

return true;

} else {

return false;

}

}

@Override

/\*\* Add an edge to the graph \*/

public boolean addEdge(int u, int v) {

return addEdge(new Edge(u, v));

}

@Override

/\*\* Obtain a DFS tree starting from vertex u \*/

/\*\* To be discussed in Section 28.7 \*/

public SearchTree dfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

for (int i = 0; i < parent.length; i++)

parent[i] = -1; // Initialize parent[i] to -1

// Mark visited vertices

boolean[] isVisited = new boolean[vertices.size()];

// Recursively search

dfs(v, parent, searchOrder, isVisited);

// Return a search tree

return new SearchTree(v, parent, searchOrder);

}

/\*\*

\* Recursive method for DFS search

\*/

private void dfs(int v, int[] parent, List<Integer> searchOrder,

boolean[] isVisited) {

// Store the visited vertex

searchOrder.add(v);

isVisited[v] = true; // Vertex v visited

for (Edge e : neighbors.get(v)) { // Note that e.u is v

if (!isVisited[e.v]) { // e.v is w in Listing 28.8

parent[e.v] = v; // The parent of w is v

dfs(e.v, parent, searchOrder, isVisited); // Recursive search

}

}

}

@Override

/\*\* Starting bfs search from vertex v \*/

/\*\* To be discussed in Section 28.9 \*/

public SearchTree bfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

for (int i = 0; i < parent.length; i++)

parent[i] = -1; // Initialize parent[i] to -1

java.util.LinkedList<Integer> queue =

new java.util.LinkedList<>(); // list used as a queue

boolean[] isVisited = new boolean[vertices.size()];

queue.offer(v); // Enqueue v

isVisited[v] = true; // Mark it visited

while (!queue.isEmpty()) {

int u = queue.poll(); // Dequeue to u

searchOrder.add(u); // u searched

for (Edge e : neighbors.get(u)) { // Note that e.u is u

if (!isVisited[e.v]) { // e.v is w in Listing 28.11

queue.offer(e.v); // Enqueue w

parent[e.v] = u; // The parent of w is u

isVisited[e.v] = true; // Mark w visited

}

}

}

return new SearchTree(v, parent, searchOrder);

}

/\*\* Tree inner class inside the UnweightedGraph class \*/

/\*\*

\* To be discussed in Section 28.6

\*/

public class SearchTree {

private int root; // The root of the tree

private int[] parent; // Store the parent of each vertex

private List<Integer> searchOrder; // Store the search order

/\*\*

\* Construct a tree with root, parent, and searchOrder

\*/

public SearchTree(int root, int[] parent,

List<Integer> searchOrder) {

this.root = root;

this.parent = parent;

this.searchOrder = searchOrder;

}

/\*\*

\* Return the root of the tree

\*/

public int getRoot() {

return root;

}

/\*\*

\* Return the parent of vertex v

\*/

public int getParent(int v) {

return parent[v];

}

/\*\*

\* Return an array representing search order

\*/

public List<Integer> getSearchOrder() {

return searchOrder;

}

/\*\*

\* Return number of vertices found

\*/

public int getNumberOfVerticesFound() {

return searchOrder.size();

}

/\*\*

\* Return the path of vertices from a vertex to the root

\*/

public List<V> getPath(int index) {

ArrayList<V> path = new ArrayList<>();

do {

path.add(vertices.get(index));

index = parent[index];

}

while (index != -1);

return path;

}

/\*\*

\* Print a path from the root to vertex v

\*/

public void printPath(int index) {

List<V> path = getPath(index);

System.out.print("A path from " + vertices.get(root) + " to " +

vertices.get(index) + ": ");

for (int i = path.size() - 1; i >= 0; i--)

System.out.print(path.get(i) + " ");

}

/\*\*

\* Print the whole tree

\*/

public void printTree() {

System.out.println("Root is: " + vertices.get(root));

System.out.print("Edges: ");

for (int i = 0; i < parent.length; i++) {

if (parent[i] != -1) {

// Display an edge

System.out.print("(" + vertices.get(parent[i]) + ", " +

vertices.get(i) + ") ");

}

}

System.out.println();

}

}

@Override

/\*\* Remove vertex v and return true if successful \*/

public boolean remove(V v) {

return true; // Implementation left as an exercise

}

@Override

/\*\* Remove edge (u, v) and return true if successful \*/

public boolean remove(int u, int v) {

return true; // Implementation left as an exercise

}

}

WeightedGraph类的代码如下。

import java.util.\*;

public class WeightedGraph<V> extends UnweightedGraph<V> {

/\*\*

\* Construct an empty

\*/

public WeightedGraph() {

}

/\*\*

\* Construct a WeightedGraph from vertices and edged in arrays

\*/

public WeightedGraph(V[] vertices, int[][] edges) {

createWeightedGraph(java.util.Arrays.asList(vertices), edges);

}

/\*\*

\* Construct a WeightedGraph from vertices and edges in list

\*/

public WeightedGraph(int[][] edges, int numberOfVertices) {

List<V> vertices = new ArrayList<>();

for (int i = 0; i < numberOfVertices; i++)

vertices.add((V) (new Integer(i)));

createWeightedGraph(vertices, edges);

}

/\*\*

\* Construct a WeightedGraph for vertices 0, 1, 2 and edge list

\*/

public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {

createWeightedGraph(vertices, edges);

}

/\*\*

\* Construct a WeightedGraph from vertices 0, 1, and edge array

\*/

public WeightedGraph(List<WeightedEdge> edges,

int numberOfVertices) {

List<V> vertices = new ArrayList<>();

for (int i = 0; i < numberOfVertices; i++)

vertices.add((V) (new Integer(i)));

createWeightedGraph(vertices, edges);

}

/\*\*

\* Create adjacency lists from edge arrays

\*/

private void createWeightedGraph(List<V> vertices, int[][] edges) {

this.vertices = vertices;

for (int i = 0; i < vertices.size(); i++) {

neighbors.add(new ArrayList<Edge>()); // Create a list for vertices

}

for (int i = 0; i < edges.length; i++) {

neighbors.get(edges[i][0]).add(

new WeightedEdge(edges[i][0], edges[i][1], edges[i][2]));

}

}

/\*\*

\* Create adjacency lists from edge lists

\*/

private void createWeightedGraph(

List<V> vertices, List<WeightedEdge> edges) {

this.vertices = vertices;

for (int i = 0; i < vertices.size(); i++) {

neighbors.add(new ArrayList<Edge>()); // Create a list for vertices

}

for (WeightedEdge edge : edges) {

neighbors.get(edge.u).add(edge); // Add an edge into the list

}

}

/\*\*

\* Return the weight on the edge (u, v)

\*/

public double getWeight(int u, int v) throws Exception {

for (Edge edge : neighbors.get(u)) {

if (edge.v == v) {

return ((WeightedEdge) edge).weight;

}

}

throw new Exception("Edge does not exit");

}

/\*\*

\* Display edges with weights

\*/

public void printWeightedEdges() {

for (int i = 0; i < getSize(); i++) {

System.out.print(getVertex(i) + " (" + i + "): ");

for (Edge edge : neighbors.get(i)) {

System.out.print("(" + edge.u +

", " + edge.v + ", " + ((WeightedEdge) edge).weight + ") ");

}

System.out.println();

}

}

/\*\*

\* Add an edge (u, v, weight) to the graph.

\*/

public boolean addEdge(int u, int v, double weight) {

return addEdge(new WeightedEdge(u, v, weight));

}

/\*\*

\* Get a minimum spanning tree rooted at vertex 0

\*/

public MST getMinimumSpanningTree() {

return getMinimumSpanningTree(0);

}

/\*\*

\* Get a minimum spanning tree rooted at a specified vertex

\*/

public MST getMinimumSpanningTree(int startingVertex) {

// cost[v] stores the cost by adding v to the tree

double[] cost = new double[getSize()];

for (int i = 0; i < cost.length; i++) {

cost[i] = Double.POSITIVE\_INFINITY; // Initial cost

}

cost[startingVertex] = 0; // Cost of source is 0

int[] parent = new int[getSize()]; // Parent of a vertex

parent[startingVertex] = -1; // startingVertex is the root

double totalWeight = 0; // Total weight of the tree thus far

List<Integer> T = new ArrayList<>();

// Expand T

while (T.size() < getSize()) {

// Find smallest cost u in V - T

int u = -1; // Vertex to be determined

double currentMinCost = Double.POSITIVE\_INFINITY;

for (int i = 0; i < getSize(); i++) {

if (!T.contains(i) && cost[i] < currentMinCost) {

currentMinCost = cost[i];

u = i;

}

}

if (u == -1) break;

else T.add(u); // Add a new vertex to T

totalWeight += cost[u]; // Add cost[u] to the tree

// Adjust cost[v] for v that is adjacent to u and v in V - T

for (Edge e : neighbors.get(u)) {

if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge) e).weight) {

cost[e.v] = ((WeightedEdge) e).weight;

parent[e.v] = u;

}

}

}

return new MST(startingVertex, parent, T, totalWeight);

}

/\*\*

\* MST is an inner class in WeightedGraph

\*/

public class MST extends SearchTree {

private double totalWeight; // Total weight of all edges in the tree

public MST(int root, int[] parent, List<Integer> searchOrder,

double totalWeight) {

super(root, parent, searchOrder);

this.totalWeight = totalWeight;

}

public double getTotalWeight() {

return totalWeight;

}

}

/\*\*

\* Find single source shortest paths

\*/

public ShortestPathTree getShortestPath(int sourceVertex) {

// cost[v] stores the cost of the path from v to the source

double[] cost = new double[getSize()];

for (int i = 0; i < cost.length; i++) {

cost[i] = Double.POSITIVE\_INFINITY; // Initial cost set to infinity

}

cost[sourceVertex] = 0; // Cost of source is 0

// parent[v] stores the previous vertex of v in the path

int[] parent = new int[getSize()];

parent[sourceVertex] = -1; // The parent of source is set to -1

// T stores the vertices whose path found so far

List<Integer> T = new ArrayList<>();

// Expand T

while (T.size() < getSize()) {

// Find smallest cost v in V - T

int u = -1; // Vertex to be determined

double currentMinCost = Double.POSITIVE\_INFINITY;

for (int i = 0; i < getSize(); i++) {

if (!T.contains(i) && cost[i] < currentMinCost) {

currentMinCost = cost[i];

u = i;

}

}

if (u == -1) break;

else T.add(u); // Add a new vertex to T

// Adjust cost[v] for v that is adjacent to u and v in V - T

for (Edge e : neighbors.get(u)) {

if (!T.contains(e.v)

&& cost[e.v] > cost[u] + ((WeightedEdge) e).weight) {

cost[e.v] = cost[u] + ((WeightedEdge) e).weight;

parent[e.v] = u;

}

}

} // End of while

// Create a ShortestPathTree

return new ShortestPathTree(sourceVertex, parent, T, cost);

}

/\*\*

\* ShortestPathTree is an inner class in WeightedGraph

\*/

public class ShortestPathTree extends SearchTree {

private double[] cost; // cost[v] is the cost from v to source

/\*\*

\* Construct a path

\*/

public ShortestPathTree(int source, int[] parent,

List<Integer> searchOrder, double[] cost) {

super(source, parent, searchOrder);

this.cost = cost;

}

/\*\*

\* Return the cost for a path from the root to vertex v

\*/

public double getCost(int v) {

return cost[v];

}

/\*\*

\* Print paths from all vertices to the source

\*/

public void printAllPaths() {

System.out.println("All shortest paths from " +

vertices.get(getRoot()) + " are:");

for (int i = 0; i < cost.length; i++) {

printPath(i); // Print a path from i to the source

System.out.println("(cost: " + cost[i] + ")"); // Path cost

}

}

}

}

1.3.1 图遍历（以无权图为例）

这里面还涉及了其他的一些内容，比如图遍历算法。图遍历是指按照某种顺序访问图中的每个顶点，且每个顶点仅被访问一次的过程。

常见的两种方法是深度优先搜索（Depth-First Search, DFS）和广度优先搜索（Breadth-First Search, BFS），无论是 DFS 还是 BFS，遍历图的过程都会生成一个生成树（Spanning Tree），这是原图的一个子图，包含了图中的所有顶点。

1.3.1.1 深度优先搜索（Depth-First Search, DFS）

DFS 从图中的某个顶点开始，尽可能深地搜索图的分支，直到不能继续为止，然后回溯。

具体步骤如下：

访问当前顶点：标记当前顶点v vv为已访问。

遍历邻居：对于v vv的每个邻居w ww：

如果w ww未被访问：

在搜索树中将v vv设置为w ww的父顶点。

递归地访问w ww，这将在每次递归步骤中改变当前顶点。

伪代码如下。

Tree dfs(vertex v) {

// 标记当前顶点为已访问

visit(v);

// 遍历每个邻居

for each neighbor w of v {

// 如果这个邻居未被访问

if (w has not been visited) {

// 在搜索树中将 v 设置为 w 的父顶点

set v as the parent for w;

// 递归地访问 w

dfs(w);

}

}

}

它的时间复杂度为O ( ∣ E ∣ + ∣ V ∣ )

由于每个顶点恰好被访问一次，因此与顶点相关的操作总数是∣ V ∣ 。

由于每条边恰好被访问一次（在检查邻接顶点时），因此与边相关的操作总数是 ∣ E ∣ 。

因此，算法的总操作数是顶点操作数和边操作数之和，即∣ E ∣ + ∣ V ∣ 。

如何在Java中实现深度优先搜索（DFS）算法呢？

在Graph接口中我们定义了dfs(int) 方法，该方法用于执行深度优先搜索。

我们需要在UnweightedGraph 类实现这个方法。为此我们需要借助两个类，一个是Edge类，表示图中的边。一个是SeachTree类，可以作为容器存储和打印DFS遍历的结果。

UML如图所示。

在 UnweightedGraph 类中，私有递归 dfs 方法负责执行实际的DFS操作。

Edge 类表示边，支持DFS操作，通过u uu和v vv属性表示边的起点和终点。

具体代码如下。

public class Edge {

public int u;

public int v;

public Edge(int u, int v) {

this.u = u;

this.v = v;

}

public boolean equals(Object o) {

return u == ((Edge) o).u && v == ((Edge) o).v;

}

}

SearchTree 内部类作为容器，存储和打印DFS遍历的结果。

public class SearchTree {

private int root; // The root of the tree

private int[] parent; // Store the parent of each vertex

private List<Integer> searchOrder; // Store the search order

/\*\*

\* Construct a tree with root, parent, and searchOrder

\*/

public SearchTree(int root, int[] parent,

List<Integer> searchOrder) {

this.root = root;

this.parent = parent;

this.searchOrder = searchOrder;

}

/\*\*

\* Return the root of the tree

\*/

public int getRoot() {

return root;

}

/\*\*

\* Return the parent of vertex v

\*/

public int getParent(int v) {

return parent[v];

}

/\*\*

\* Return an array representing search order

\*/

public List<Integer> getSearchOrder() {

return searchOrder;

}

/\*\*

\* Return number of vertices found

\*/

public int getNumberOfVerticesFound() {

return searchOrder.size();

}

/\*\*

\* Return the path of vertices from a vertex to the root

\*/

public List<V> getPath(int index) {

ArrayList<V> path = new ArrayList<>();

do {

path.add(vertices.get(index));

index = parent[index];

}

while (index != -1);

return path;

}

/\*\*

\* Print a path from the root to vertex v

\*/

public void printPath(int index) {

List<V> path = getPath(index);

System.out.print("A path from " + vertices.get(root) + " to " +

vertices.get(index) + ": ");

for (int i = path.size() - 1; i >= 0; i--)

System.out.print(path.get(i) + " ");

}

/\*\*

\* Print the whole tree

\*/

public void printTree() {

System.out.println("Root is: " + vertices.get(root));

System.out.print("Edges: ");

for (int i = 0; i < parent.length; i++) {

if (parent[i] != -1) {

// Display an edge

System.out.print("(" + vertices.get(parent[i]) + ", " +

vertices.get(i) + ") ");

}

}

System.out.println();

}

}

SearchTree 类用于存储和打印深度优先搜索（DFS）遍历的结果。

包含三个属性：root（根节点）、parent（每个顶点的父节点数组）和 searchOrder（访问顶点的顺序列表）。

提供了一个构造方法来初始化这些属性，并提供了一个方法来获取树的节点等。

UnweightedGraph 类中的 dfs 方法如下。

// UnweightedGraph 类中的 dfs 方法

public SearchTree dfs(int v) {

// 一个列表，用于记录访问顶点的顺序

List<Integer> searchOrder = new ArrayList<>();

// 一个数组，用于存储DFS树中每个顶点的父节点

int[] parent = new int[vertices.size()];

for (int i = 0; i < parent.length; i++)

parent[i] = -1; // 初始化 parent 数组

// 一个布尔数组，用于标记顶点是否被访问过

boolean[] isVisited = new boolean[vertices.size()];

dfs(v, parent, searchOrder, isVisited);

// 返回一个新的 SearchTree 对象

return new SearchTree(v, parent, searchOrder);

}

// 递归的 dfs 方法

private void dfs(int v, int[] parent, List<Integer> searchOrder, boolean[] isVisited) {

// 标记顶点 v 为已访问

isVisited[v] = true;

// 将访问的顶点添加到 searchOrder 列表中

searchOrder.add(v);

// 遍历顶点 v 的所有邻居

for (Edge e : neighbors.get(v)) {

// 如果邻居 w 未被访问

if (!isVisited[e.v]) {

// 将 w 的父节点设置为 v

parent[e.v] = v;

// 递归地访问 w

dfs(e.v, parent, searchOrder, isVisited);

}

}

}

dfs(int v) 是一个公共方法，用于从顶点 v 开始执行DFS遍历，并返回一个 SearchTree 对象。

初始化 searchOrder 列表和 parent 数组，并调用私有的递归 dfs 方法。

返回一个新的 SearchTree 对象，包含遍历的结果。

而dfs(int v, int[] parent, List searchOrder, boolean[] isVisited) 是一个私有方法，负责执行实际的递归DFS遍历。

标记当前顶点 v 为已访问，并将其添加到 searchOrder 列表中。

遍历顶点 v 的所有邻居，如果邻居未被访问，则递归地访问该邻居，并更新 parent 数组。

测试代码如下。

public class TestDFS {

public static void main(String[] args) {

String[] vertices = {"Seattle", "San Francisco", "Los Angeles",

"Denver", "Kansas City", "Chicago", "Boston", "New York",

"Atlanta", "Miami", "Dallas", "Houston"};

int[][] edges = {

{0, 1}, {0, 3}, {0, 5},

{1, 0}, {1, 2}, {1, 3},

{2, 1}, {2, 3}, {2, 4}, {2, 10},

{3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},

{4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},

{5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},

{6, 5}, {6, 7},

{7, 4}, {7, 5}, {7, 6}, {7, 8},

{8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},

{9, 8}, {9, 11},

{10, 2}, {10, 4}, {10, 8}, {10, 11},

{11, 8}, {11, 9}, {11, 10}

};

Graph<String> graph = new UnweightedGraph<>(vertices, edges);

UnweightedGraph<String>.SearchTree dfs =

graph.dfs(graph.getIndex("Chicago"));

java.util.List<Integer> searchOrders = dfs.getSearchOrder();

System.out.println(dfs.getNumberOfVerticesFound() +

" vertices are searched in this DFS order:");

for (int i = 0; i < searchOrders.size(); i++)

System.out.print(graph.getVertex(searchOrders.get(i)) + " ");

System.out.println();

for (int i = 0; i < searchOrders.size(); i++)

if (dfs.getParent(i) != -1)

System.out.println("parent of " + graph.getVertex(i) +

" is " + graph.getVertex(dfs.getParent(i)));

}

}

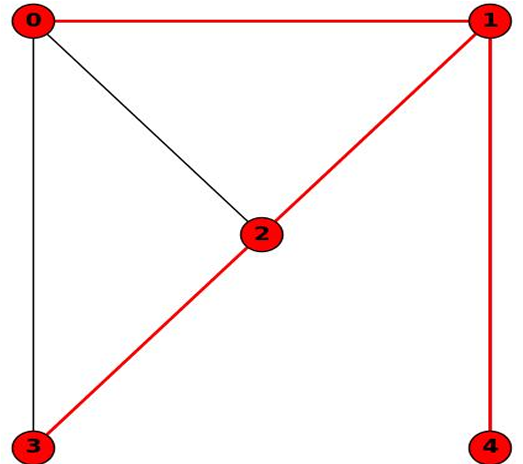
DFS的应用如下：

检测图是否连通：通过从任意顶点开始进行DFS遍历，如果遍历的顶点数量与图中顶点的总数相同，则图是连通的；否则，图不是连通的。

查找图中的所有连通分量：对于无向图，DFS可以用来找到所有的连通分量，即图中的每个连通子图。

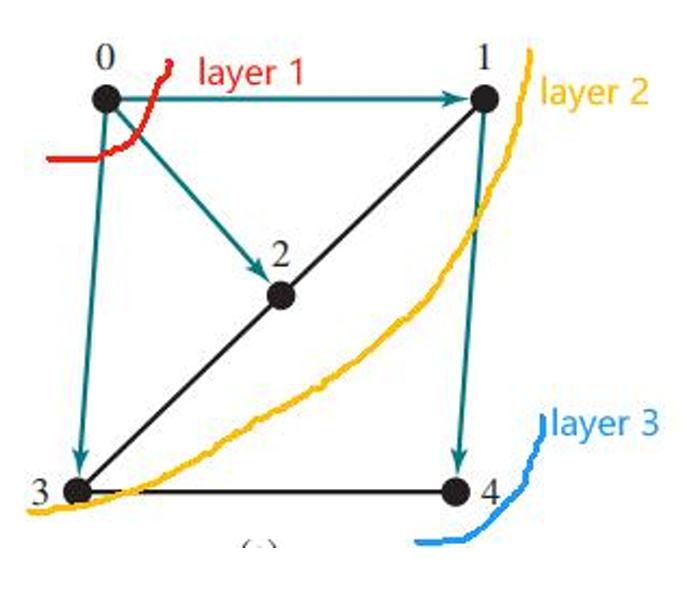
检测两个顶点之间是否存在路径，并找到它（不一定是最短路径）。

如图所示。这里0和2之间的最短路径是0-2，但是DFS的结果是0-1-2。



1.3.1.2 广度优先搜索（Breadth-First Search，简称BFS）

广度优先搜索按照从近到远的顺序（即层层向外扩展）访问图中的所有顶点。



算法的步骤如下：

创建一个空的队列，用于存储待访问的顶点。

将起始顶点 v 加入队列，并标记为已访问。

当队列不为空时，执行以下操作：

从队列中取出（出队）一个顶点，称为u uu。

遍历顶点u uu的所有邻居w ww：

如果邻居w ww未被访问：

将w ww加入队列。

将u uu设置为w ww的父顶点（在BFS树中）。

标记w ww为已访问。

伪代码如下：

bfs(vertex v) {

// 创建一个空队列用于存储待访问的顶点

create an empty queue for storing vertices to be visited;

// 将起始顶点 v 加入队列，并标记为已访问

add v into the queue;

mark v visited;

// 当队列不为空时，执行以下操作

while the queue is not empty {

// 从队列中取出一个顶点，称为 u

dequeue a vertex, say u, from the queue;

// 遍历顶点 u 的所有邻居 w

for each neighbor w of u {

// 如果 w 未被访问

if w has not been visited {

// 将 w 加入队列

add w into the queue;

// 将 u 设置为 w 的父顶点

set u as the parent for w;

// 标记 w 为已访问

mark w visited;

}

}

}

}

它的时间复杂度为O ( ∣ E ∣ + ∣ V ∣ ) ，与前者一样。

由于每个顶点恰好被访问一次，因此与顶点相关的操作总数是∣ V ∣。

由于每条边恰好被访问一次（在检查邻接顶点时），因此与边相关的操作总数是 ∣ E ∣ 。

因此，算法的总操作数是顶点操作数和边操作数之和，即∣ E ∣ + ∣ V ∣ 。

实现与牵着类似。

在 UnweightedGraph 类中，私有递归 bfs 方法负责执行实际的BFS操作。用Edge类表示边，支持 BFS 操作。SearchTree 内部类作为容器，存储和打印BFS遍历的结果。

相关的代码如下。

@Override

/\*\* Starting bfs search from vertex v \*/

public SearchTree bfs(int v) {

List<Integer> searchOrder = new ArrayList<>();

int[] parent = new int[vertices.size()];

for (int i = 0; i < parent.length; i++)

parent[i] = -1; // Initialize parent[i] to -1

java.util.LinkedList<Integer> queue =

new java.util.LinkedList<>(); // list used as a queue

boolean[] isVisited = new boolean[vertices.size()];

queue.offer(v); // Enqueue v

isVisited[v] = true; // Mark it visited

while (!queue.isEmpty()) {

int u = queue.poll(); // Dequeue to u

searchOrder.add(u); // u searched

for (Edge e : neighbors.get(u)) { // Note that e.u is u

if (!isVisited[e.v]) { // e.v is w in Listing 28.11

queue.offer(e.v); // Enqueue w

parent[e.v] = u; // The parent of w is u

isVisited[e.v] = true; // Mark w visited

}

}

}

return new SearchTree(v, parent, searchOrder);

}

测试代码如下。

public class TestBFS {

public static void main(String[] args) {

String[] vertices = {"Seattle", "San Francisco", "Los Angeles",

"Denver", "Kansas City", "Chicago", "Boston", "New York",

"Atlanta", "Miami", "Dallas", "Houston"};

int[][] edges = {

{0, 1}, {0, 3}, {0, 5},

{1, 0}, {1, 2}, {1, 3},

{2, 1}, {2, 3}, {2, 4}, {2, 10},

{3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},

{4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},

{5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},

{6, 5}, {6, 7},

{7, 4}, {7, 5}, {7, 6}, {7, 8},

{8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},

{9, 8}, {9, 11},

{10, 2}, {10, 4}, {10, 8}, {10, 11},

{11, 8}, {11, 9}, {11, 10}

};

Graph<String> graph = new UnweightedGraph<>(vertices, edges);

UnweightedGraph<String>.SearchTree bfs =

graph.bfs(graph.getIndex("Chicago"));

java.util.List<Integer> searchOrders = bfs.getSearchOrder();

System.out.println(bfs.getNumberOfVerticesFound() +

" vertices are searched in this order:");

for (int i = 0; i < searchOrders.size(); i++)

System.out.println(graph.getVertex(searchOrders.get(i)));

for (int i = 0; i < searchOrders.size(); i++)

if (bfs.getParent(i) != -1)

System.out.println("parent of " + graph.getVertex(i) +

" is " + graph.getVertex(bfs.getParent(i)));

}

}

BFS的应用有：

检测图是否连通：检测图中任意两个顶点之间是否存在路径，从而判断图是否连通。

也可以 通过BFS生成的树（生成树是包含图中所有顶点的无环连通子图）的大小是否与图中顶点的数量相同，可以作为图是否连通的一个指标。

用BFS确定两个特定顶点之间是否存在路径。

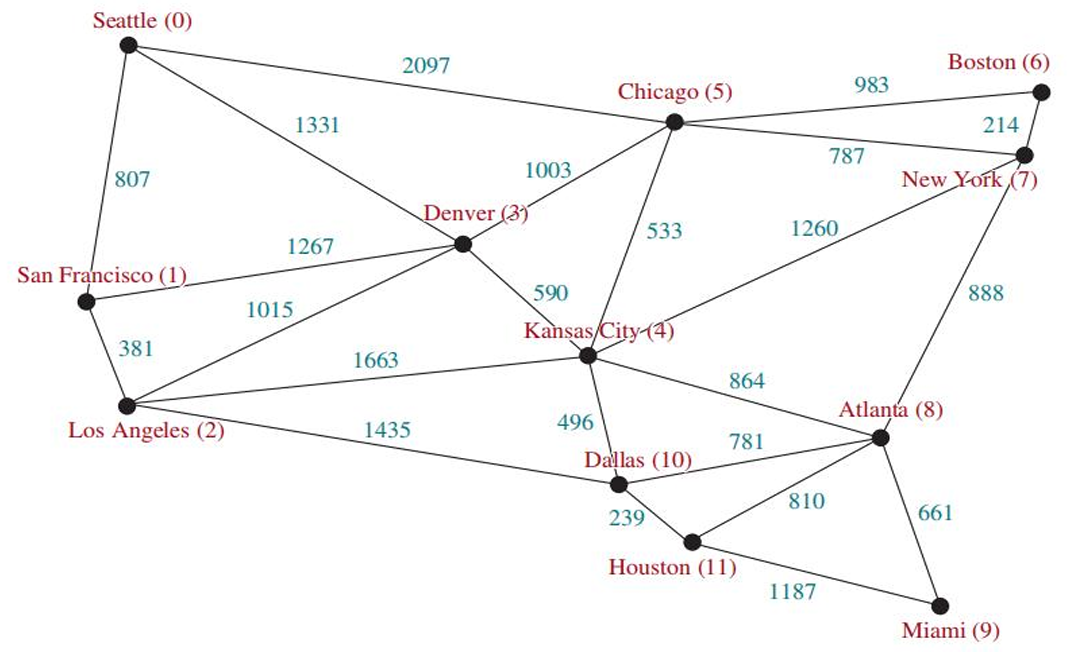
查找所有连通分量.

在无权图中找到两个顶点之间的最短路径：因为它是按照边的数量来探索顶点的。即BFS按照离起始顶点的边数来逐层探索顶点，首先探索所有与起始顶点相邻的顶点（1条边），然后是这些顶点的邻居（2条边），以此类推。因此，当BFS第一次到达某个顶点时，它一定是通过最短的可能边数。

1.3.2 有权图

前面我们说的一直是无权图，现在我们说有权图的相关方法如何实现。

首先先有权图是每条边都被赋予了一个权重或者值的图。



所以现在我们的表示方式也要有所改变。

比如使用边数组（Edge Array）表示加权边的时候，需要加入权重。

int[][] edges = {

{0, 1, 2}, {0, 3, 8},

{1, 0, 2}, {1, 2, 7}, {1, 3, 3},

{2, 1, 7}, {2, 3, 4}, {2, 4, 5},

{3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},

{4, 2, 5}, {4, 3, 6}

};

同理，使用邻接矩阵（Adjacency Matrix）表示加权边。

Integer[][] adjacencyMatrix = {

{null, 2, null, 8, null},

{2, null, 1, 3, null},

{null, 1, null, 7, 3},

{null, 3, null, 4, 5},

{8, 3, 4, null, 6},

{null, null, 5, 6, null}

};

使用邻接表（Adjacency List）表示加权边。

import java.util.List;

import java.util.ArrayList;

// 一个包含5个列表的数组，每个列表打算存储加权边对象

List<WeightedEdge>[] list = new ArrayList[5];

// 加权边类

public class WeightedEdge implements Comparable<WeightedEdge> {

// 定义权重

public double weight;

// 使用权重构造加权边

public WeightedEdge(int u, int v, double weight) {

super(u, v);

this.weight = weight;

}

// 基于权重比较两个边

public int compareTo(WeightedEdge edge) {

// 如果当前边的权重大于另一个边，返回1，

// 如果相等，返回0，

// 如果小于，返回-1

}

}

UnweightedGraph类实现了Graph接口，而WeightedGraph类继承自 UnweightedGraph类，添加了处理加权图的额外能力。

1.3.3 最小生成树（Minimum Spanning Tree，简称MST）

在有权图的基础上，我们有最小生成树。

生成树是图G GG的一个连通子图，这个子图是一个树，包含了G GG中所有的顶点。

最小生成树是所有可能的生成树中，边的总权重最小的那一个。

应用如下：

网络设计：在设计网络时，最小生成树可以用来确定连接所有节点的最优方式，以最小化建设成本。

电路设计：在电路设计中，最小生成树可以用来优化电路的布局，减少材料的使用。

数据压缩：在某些数据压缩算法中，最小生成树可以用来减少数据冗余。

我们还需要知道的是一个图可能会有多个最小生成树。

1.3.3.1 Prim算法

算法步骤如下：

初始化：

让V VV表示图中顶点的集合。

让T TT表示生成树中的顶点集合。最初，将起始顶点添加到T TT。

构建生成树：

当T TT的大小小于V VV的大小时（即还有顶点未加入生成树），执行以下操作：

找到T TT中的顶点u uu和V − T V−TV−T（即不在生成树中的顶点集合）中的顶点v vv，使得边( u , v ) (u,v)(u,v)的权重最小。

将顶点v vv添加到T TT。

伪代码如下。

MST getMinimumSpanningTree(s) {

Let V denote the set of vertices in the graph;

Let T be a set for the vertices in the spanning tree;

Initially, add the starting vertex to T;

while (size of T < n) {

find u in T and v in V - T with the smallest weight on the edge (u, v), as shown in the figure;

add v to T;

}

}

代码如下。

/\*\*

\* Get a minimum spanning tree rooted at vertex 0

\*/

public MST getMinimumSpanningTree() {

return getMinimumSpanningTree(0);

}

/\*\*

\* Get a minimum spanning tree rooted at a specified vertex

\*/

public MST getMinimumSpanningTree(int startingVertex) {

// cost[v] stores the cost by adding v to the tree

double[] cost = new double[getSize()];

for (int i = 0; i < cost.length; i++) {

cost[i] = Double.POSITIVE\_INFINITY; // Initial cost

}

cost[startingVertex] = 0; // Cost of source is 0

int[] parent = new int[getSize()]; // Parent of a vertex

parent[startingVertex] = -1; // startingVertex is the root

double totalWeight = 0; // Total weight of the tree thus far

List<Integer> T = new ArrayList<>();

// Expand T

while (T.size() < getSize()) {

// Find smallest cost u in V - T

int u = -1; // Vertex to be determined

double currentMinCost = Double.POSITIVE\_INFINITY;

for (int i = 0; i < getSize(); i++) {

if (!T.contains(i) && cost[i] < currentMinCost) {

currentMinCost = cost[i];

u = i;

}

}

if (u == -1) break;

else T.add(u); // Add a new vertex to T

totalWeight += cost[u]; // Add cost[u] to the tree

// Adjust cost[v] for v that is adjacent to u and v in V - T

for (Edge e : neighbors.get(u)) {

if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge) e).weight) {

cost[e.v] = ((WeightedEdge) e).weight;

parent[e.v] = u;

}

}

}

return new MST(startingVertex, parent, T, totalWeight);

}

测试代码如下。

public class TestMinimumSpanningTree {

public static void main(String[] args) {

String[] vertices = {"Seattle", "San Francisco", "Los Angeles",

"Denver", "Kansas City", "Chicago", "Boston", "New York",

"Atlanta", "Miami", "Dallas", "Houston"};

int[][] edges = {

{0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},

{1, 0, 807}, {1, 2, 381}, {1, 3, 1267},

{2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},

{3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},

{3, 5, 1003},

{4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},

{4, 8, 864}, {4, 10, 496},

{5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},

{5, 6, 983}, {5, 7, 787},

{6, 5, 983}, {6, 7, 214},

{7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},

{8, 4, 864}, {8, 7, 888}, {8, 9, 661},

{8, 10, 781}, {8, 11, 810},

{9, 8, 661}, {9, 11, 1187},

{10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},

{11, 8, 810}, {11, 9, 1187}, {11, 10, 239}

};

WeightedGraph<String> graph1 =

new WeightedGraph<>(vertices, edges);

WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();

System.out.println("Total weight is " + tree1.getTotalWeight());

tree1.printTree();

edges = new int[][]{

{0, 1, 2}, {0, 3, 8},

{1, 0, 2}, {1, 2, 7}, {1, 3, 3},

{2, 1, 7}, {2, 3, 4}, {2, 4, 5},

{3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},

{4, 2, 5}, {4, 3, 6}

};

WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);

WeightedGraph<Integer>.MST tree2 =

graph2.getMinimumSpanningTree(1);

System.out.println("\nTotal weight is " + tree2.getTotalWeight());

tree2.printTree();

}

}

1.3.3.2 Dijkstra算法

最短路径问题是找到两个顶点之间的最短路径，即总权重（或成本）最小的路径。

算法步骤如下：

初始化：将起始顶点 s 到自身的最短路径距离设为0，到所有其他顶点的最短路径距离设为无穷大。

选择未访问顶点中距离最小的顶点：从起始顶点开始，选择未访问的顶点中，到起始顶点距离最小的顶点。

更新邻居顶点的距离：对于选定顶点的每个邻居顶点，计算从起始顶点经过选定顶点到邻居顶点的距离，如果这个距离小于当前记录的距离，则更新距离。

标记选定顶点为已访问：完成选定顶点的邻居顶点距离更新后，将其标记为已访问。

重复：重复步骤2-4，直到图中所有顶点都被访问。

伪代码如下。

ShortestPathTree getShortestPath(int s) {

Let T be a set that contains the vertices whose paths to s are known;

Set cost[s] = 0;

for (each vertex v in V) {

if (v != s) {

cost[v] = infinity;

}

}

while (size of T < n) {

Find u not in T with the smallest cost[u];

Add u to T;

for (each edge (u, v) in E) {

if (v not in T && cost[v] > cost[u] + w(u, v)) {

cost[v] = w(u, v) + cost[u];

parent[v] = u;

}

}

}

return new ShortestPathTree();

}

这个算法其实是找到顶点s ss到其他所有顶点的最短路径。

具体代码如下。

/\*\*

\* Find single source shortest paths

\*/

public ShortestPathTree getShortestPath(int sourceVertex) {

// cost[v] stores the cost of the path from v to the source

double[] cost = new double[getSize()];

for (int i = 0; i < cost.length; i++) {

cost[i] = Double.POSITIVE\_INFINITY; // Initial cost set to infinity

}

cost[sourceVertex] = 0; // Cost of source is 0

// parent[v] stores the previous vertex of v in the path

int[] parent = new int[getSize()];

parent[sourceVertex] = -1; // The parent of source is set to -1

// T stores the vertices whose path found so far

List<Integer> T = new ArrayList<>();

// Expand T

while (T.size() < getSize()) {

// Find smallest cost v in V - T

int u = -1; // Vertex to be determined

double currentMinCost = Double.POSITIVE\_INFINITY;

for (int i = 0; i < getSize(); i++) {

if (!T.contains(i) && cost[i] < currentMinCost) {

currentMinCost = cost[i];

u = i;

}

}

if (u == -1) break;

else T.add(u); // Add a new vertex to T

// Adjust cost[v] for v that is adjacent to u and v in V - T

for (Edge e : neighbors.get(u)) {

if (!T.contains(e.v)

&& cost[e.v] > cost[u] + ((WeightedEdge) e).weight) {

cost[e.v] = cost[u] + ((WeightedEdge) e).weight;

parent[e.v] = u;

}

}

} // End of while

// Create a ShortestPathTree

return new ShortestPathTree(sourceVertex, parent, T, cost);

}

测试代码如下。

public class TestShortestPath {

public static void main(String[] args) {

String[] vertices = {"Seattle", "San Francisco", "Los Angeles",

"Denver", "Kansas City", "Chicago", "Boston", "New York",

"Atlanta", "Miami", "Dallas", "Houston"};

int[][] edges = {

{0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},

{1, 0, 807}, {1, 2, 381}, {1, 3, 1267},

{2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},

{3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},

{3, 5, 1003},

{4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},

{4, 8, 864}, {4, 10, 496},

{5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},

{5, 6, 983}, {5, 7, 787},

{6, 5, 983}, {6, 7, 214},

{7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},

{8, 4, 864}, {8, 7, 888}, {8, 9, 661},

{8, 10, 781}, {8, 11, 810},

{9, 8, 661}, {9, 11, 1187},

{10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},

{11, 8, 810}, {11, 9, 1187}, {11, 10, 239}

};

WeightedGraph<String> graph1 =

new WeightedGraph<>(vertices, edges);

WeightedGraph<String>.ShortestPathTree tree1 =

graph1.getShortestPath(graph1.getIndex("Chicago"));

tree1.printAllPaths();

// Display shortest paths from Houston to Chicago

System.out.print("Shortest path from Houston to Chicago: ");

java.util.List<String> path

= tree1.getPath(graph1.getIndex("Houston"));

for (String s : path) {

System.out.print(s + " ");

}

edges = new int[][]{

{0, 1, 2}, {0, 3, 8},

{1, 0, 2}, {1, 2, 7}, {1, 3, 3},

{2, 1, 7}, {2, 3, 4}, {2, 4, 5},

{3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},

{4, 2, 5}, {4, 3, 6}

};

WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);

WeightedGraph<Integer>.ShortestPathTree tree2 =

graph2.getShortestPath(3);

System.out.println("\n");

tree2.printAllPaths();

}

}

1.3.3.3 Prim算法与Dijkstra算法的对比

Prim算法关注的是如何将新的顶点添加到生成树中，每次迭代都扩展生成树，直到包含所有顶点。

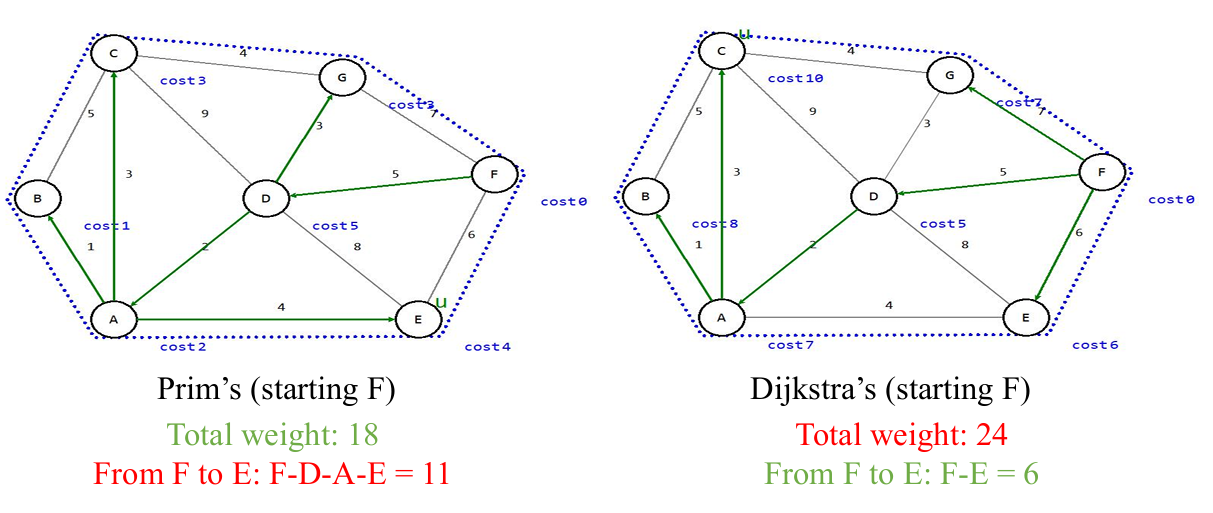
Dijkstra算法关注的是如何从起始顶点到达每个顶点的最短路径，每次迭代都更新到未访问顶点的最短路径。

所以Prim算法的目标是找到最小生成树。

而Dijkstra算法的目标是找到从指定的源顶点s ss到图中所有其他顶点的最短路径。

因此Dijkstra算法的生成树不是最小生成树。

如图所示。



两者的时间复杂度类似，前面的代码的时间复杂度都是O ( n 3 ) O(n^3)O(n

3

)，但是经过优化可以到O ( ( E + V ) l o g V ) O((E+V)logV)O((E+V)logV)。

十！！！！！！！！（树 TREES）

1.二叉树（Binary Trees）

二叉树是一种分层结构，它要么是空的，要么由一个元素（称为根节点）和两个不同的二叉树（称为左子树和右子树）组成。左子树和右子树本身也是二叉树，可以继续按照这种结构分解。

一个节点的左（右）子树的根节点被称为该节点的左（右）子节点。

没有孩子的节点称为叶子节点。

我们在学习数据结构的时候就知道如何用链式节点去表示二叉树，每个节点包含一个元素值，以及两个名为“left”和“right”的链接，分别指向左孩子和右孩子。

class TreeNode<E> {

// The value stored in this node (of generic type E)

E element;

// Reference to the left and right child nodes

TreeNode<E> left;

TreeNode<E> right;

// Constructor

// Initialize the node with a given value

public TreeNode(E o) {

element = o;

}

}

1.1 二叉搜索树（Binary Search Tree，简称BST）

二叉搜索树是一种特殊的二叉树。

这种二叉树有以下属性：

默认情况下，二叉搜索树中没有重复的元素。

对于树中的每个节点，其左子树中所有节点的值都小于该节点的值。对于树中的每个节点，其右子树中所有节点的值都大于该节点的值。

下面我们将对BST的各种操作进行简单讲解。

1.1.1 插入操作

插入操作的步骤如下：

如果树为空（root == null），直接创建一个新节点作为根节点。

如果树不为空，从根节点开始，通过比较待插入元素与当前节点的值，沿着树向下查找插入位置：

如果待插入元素小于当前节点的值，向左子树移动。

如果待插入元素大于当前节点的值，向右子树移动。

如果待插入元素与当前节点的值相等，说明元素已存在，返回 false。

找到合适的插入位置后，创建一个新节点，并将其链接到父节点的左孩子或右孩子上。

代码如下：

// 插入元素的方法

public boolean insert(E element) {

if (root == null) {

root = new TreeNode<>(element); // 如果树为空，直接创建根节点

} else {

// Locate the parent node

TreeNode<E> current = root, parent = null;

while (current != null) {

if (element.compareTo(current.element) < 0) { // 如果待插入元素小于当前节点的值

parent = current;

current = current.left; // 向左子树移动

} else if (element.compareTo(current.element) > 0) { // 如果待插入元素大于当前节点的值

parent = current;

current = current.right; // 向右子树移动

} else {

return false; // 如果元素已存在，返回false

}

}

// 插入新节点

if (element.compareTo(parent.element) < 0) {

parent.left = new TreeNode<>(element); // 插入到左子树

} else {

parent.right = new TreeNode<>(element); // 插入到右子树

}

}

return true; // 元素插入成功

}

1.1.2 搜索操作

搜索的步骤更为简单，因为插入的操作其实包含了搜索的逻辑。

步骤如下，

从根节点开始，根据目标元素与当前节点值的比较结果，决定向左子树还是右子树移动。

如果找到目标元素，返回 true。

如果遍历到叶子节点仍未找到目标元素，返回 false。

代码如下。

// 查找元素的方法

public boolean search(E element) {

// Start from the root

TreeNode<E> current = root;

while (current != null) {

if (element.compareTo(current.element) < 0) {

current = current.left; // Go left

} else if (element.compareTo(current.element) > 0) {

current = current.right; // Go right

} else {

// Element matches current.element

return true; // Element is found

}

}

return false; // Element is not in the tree

}

1.1.3 树的遍历（Tree Traversal）

树的遍历是访问树中每个节点恰好一次的过程。有几种不同的遍历方式：前序遍历、中序遍历、后序遍历、深度优先遍历和广度优先遍历。

1.1.3.1 前序遍历（Preorder Traversal）

首先访问当前节点，然后递归地访问当前节点的左子树，最后递归地访问当前节点的右子树。

下图展示了一个树。

对于这个树，它的前序遍历的结果是：60、55、45、57、100、67、107。

1.1.3.2 中序遍历（Inorder Traversal）

先递归地访问当前节点的左子树，然后访问当前节点本身，最后递归地访问当前节点的右子树。

对于前面的树，它的中序遍历的结果是：45、55、57、60、67、100、107。

1.1.3.3 后序遍历（Postorder Traversal）

先访问当前节点的左子树，然后访问当前节点的右子树，最后访问当前节点本身。

对于前面的树，它的后序遍历的结果是：45、57、55、67、107、100、60。

1.1.3.4 广度优先遍历（Breadth-First Traversal）

从根节点开始，逐层访问所有节点，先访问根节点，然后从左到右访问根节点的所有子节点，接着是子节点的所有子节点，依此类推。

对于前面的树，它的广度优先遍历的结果是：60、55、100、45、57、67、107。

1.1.3.5 深度优先遍历（Depth-First Traversal）

从根节点开始，沿着每个分支尽可能深地访问节点，直到到达叶子节点，然后再回溯到上一层节点，继续访问其他分支。

对于前面的树，它的深度优先遍历的结果是：60、55、45、57、100、67、107。

中序遍历和后序遍历的代码如下。

@Override

/\*\* Inorder traversal from the root \*/

public void inorder() {

inorder(root);

}

/\*\* Inorder traversal from a subtree \*/

protected void inorder(TreeNode<E> root) {

if (root == null) return;

inorder(root.left); // First visit the left subtree

System.out.print(root.element + " "); // Then visit the root

inorder(root.right); // Finally visit the right subtree

}

@Override

/\*\* Postorder traversal from the root \*/

public void postorder() {

postorder(root);

}

/\*\* Postorder traversal from a subtree \*/

protected void postorder(TreeNode<E> root) {

if (root == null) return;

postorder(root.left); // First visit the left subtree

postorder(root.right); // Then visit the right subtree

System.out.print(root.element + " "); // Finally visit the root

}

测试代码如下。

public class TestBST {

public static void main(String[] args) {

// Create a BST

BST<String> tree = new BST<>();

tree.insert("George");

tree.insert("Michael");

tree.insert("Tom");

tree.insert("Adam");

tree.insert("Jones");

tree.insert("Peter");

tree.insert("Daniel");

// Traverse tree

System.out.print("Inorder (sorted): ");

tree.inorder();

System.out.print("\nPostorder: ");

tree.postorder();

System.out.print("\nPreorder: ");

tree.preorder();

System.out.print("\nThe number of nodes is " + tree.getSize());

// Search for an element

System.out.print("\nIs Peter in the tree? " +

tree.search("Peter"));

// Get a path from the root to Peter

System.out.print("\nA path from the root to Peter is: ");

java.util.ArrayList<BST.TreeNode<String>> path

= tree.path("Peter");

for (int i = 0; path != null && i < path.size(); i++)

System.out.print(path.get(i).element + " ");

Integer[] numbers = {2, 4, 3, 1, 8, 5, 6, 7};

BST<Integer> intTree = new BST<>(numbers);

System.out.print("\nInorder (sorted): ");

intTree.inorder();

}

}

Tree 接口继承自 Collection 接口，扩展了树相关的操作方法，如后序遍历（postorder()）、获取树的大小（size()）、搜索（search(E)）等。

BST 类实现了 Tree 接口，提供了二叉搜索树的具体实现。

TreeNode 类是 BST 类的一个内部类，用于表示二叉搜索树中的节点。

TestBST 类包含了 main 方法，用于测试二叉搜索树的各种操作。

1.1.3.6 使用迭代器进行遍历

前面的方法通常只用于显示树中的元素。如果需要对二叉树中的元素进行处理（而不仅仅是显示它们），这些方法就不够用了。

为了解决这个问题我们可以使用迭代器（Iterator），因为它允许灵活和可定制的处理树元素，而不仅仅是显示它们。

Tree 接口扩展了 java.util.Collection 接口。由于 Collection 接口扩展了 java.lang.Iterable，因此 BST（二叉搜索树）也是 Iterable 的子类。

作为 Iterable 的子类，BST 可以直接定义一个迭代器类来实现 java.util.Iterator 接口。

代码如下。

@Override /\*\* Obtain an iterator. Use inorder. \*/

public java.util.Iterator<E> iterator() {

return new InorderIterator();

}

// Inner class InorderIterator

private class InorderIterator implements java.util.Iterator<E> {

// Store the elements in a list

private java.util.ArrayList<E> list =

new java.util.ArrayList<>();

private int current = 0; // Point to the current element in list

public InorderIterator() {

inorder(); // Traverse binary tree and store elements in list

}

/\*\* Inorder traversal from the root\*/

private void inorder() {

inorder(root);

}

/\*\* Inorder traversal from a subtree \*/

private void inorder(TreeNode<E> root) {

if (root == null) return;

inorder(root.left);

list.add(root.element);

inorder(root.right);

}

@Override /\*\* More elements for traversing? \*/

public boolean hasNext() {

if (current < list.size())

return true;

return false;

}

@Override /\*\* Get the current element and move to the next \*/

public E next() {

return list.get(current++);

}

@Override // Remove the element returned by the last next()

public void remove() {

if (current == 0) // next() has not been called yet

throw new IllegalStateException();

delete(list.get(--current));

list.clear(); // Clear the list

inorder(); // Rebuild the list

}

}

@Override /\*\* Remove all elements from the tree \*/

public void clear() {

root = null;

size = 0;

}

测试代码如下。

public class TestBSTWithIterator {

public static void main(String[] args) {

BST<String> tree = new BST<>();

tree.insert("George");

tree.insert("Michael");

tree.insert("Tom");

tree.insert("Adam");

tree.insert("Jones");

tree.insert("Peter");

tree.insert("Daniel");

for (String s: tree)

System.out.print(s.toUpperCase() + " ");

}

}

1.1.4 删除操作

删除操作也是与前面的插入操作类似，需要先找到要删除的节点.

步骤如下。

从根节点开始，沿着树向下遍历，根据BST的性质来找到目标节点。

此时 current 指向二叉树中包含要删除元素的节点，parent 指向 current 节点的父节点。 current 节点可能是 parent 节点的左孩子或右孩子。

情况1：如果 current 节点没有左孩子，那么删除操作相对简单。此时，有两种子情况：

如果 current 节点也没有右孩子（即 current 是一个叶子节点），那么可以直接从树中移除该节点，将其父节点的相应指针（左或右）设置为 null。

如果 current 节点有一个右孩子，那么可以用其右孩子替换 current 节点，然后更新 parent 节点的指针。

情况2：如果 current 节点有左孩子，删除操作会更复杂。此时，通常的做法是：

找到 current 节点的中序后继（即 current 节点右子树中值最小的节点，也就是右子树的最左节点）。

用中序后继的值替换 current 节点的值，然后删除中序后继节点。由于中序后继节点没有左孩子，这将简化为情况 1 的操作。

代码如下。

public boolean delete(E e) {

// Locate the node to be deleted and also locate its parent node

TreeNode<E> parent = null;

TreeNode<E> current = root;

while (current != null) {

if (c.compare(e, current.element) < 0) {

parent = current;

current = current.left;

}

else if (c.compare(e, current.element) > 0) {

parent = current;

current = current.right;

}

else

break; // Element is in the tree pointed at by current

}

if (current == null)

return false; // Element is not in the tree

// Case 1: current has no left child

if (current.left == null) {

// Connect the parent with the right child of the current node

if (parent == null) {

root = current.right;

}

else {

if (c.compare(e, parent.element) < 0)

parent.left = current.right;

else

parent.right = current.right;

}

}

else {

// Case 2: The current node has a left child

// Locate the rightmost node in the left subtree of

// the current node and also its parent

TreeNode<E> parentOfRightMost = current;

TreeNode<E> rightMost = current.left;

while (rightMost.right != null) {

parentOfRightMost = rightMost;

rightMost = rightMost.right; // Keep going to the right

}

// Replace the element in current by the element in rightMost

current.element = rightMost.element;

// Eliminate rightmost node

if (parentOfRightMost.right == rightMost)

parentOfRightMost.right = rightMost.left;

else

// Special case: parentOfRightMost == current

parentOfRightMost.left = rightMost.left;

}

size--; // Reduce the size of the tree

return true; // Element deleted successfully

}

测试代码如下。

package chapter25;

public class TestBSTDelete {

public static void main(String[] args) {

BST<String> tree = new BST<>();

tree.insert("George");

tree.insert("Michael");

tree.insert("Tom");

tree.insert("Adam");

tree.insert("Jones");

tree.insert("Peter");

tree.insert("Daniel");

printTree(tree);

System.out.println("\nAfter delete George:");

tree.delete("George");

printTree(tree);

System.out.println("\nAfter delete Adam:");

tree.delete("Adam");

printTree(tree);

System.out.println("\nAfter delete Michael:");

tree.delete("Michael");

printTree(tree);

}

public static void printTree(BST tree) {

// Traverse tree

System.out.print("Inorder (sorted): ");

tree.inorder();

System.out.print("\nPostorder: ");

tree.postorder();

System.out.print("\nPreorder: ");

tree.preorder();

System.out.print("\nThe number of nodes is " + tree.getSize());

System.out.println();

}

}

1.1.5 二叉树的时间复杂度

遍历操作的时间复杂度

中序遍历（Inorder Traversal）、前序遍历（Preorder Traversal） 和 后序遍历（Postorder Traversal）：时间复杂度为 O(n)，其中 n 是树中节点的数量。

这是因为在这些遍历中，每个节点恰好被访问一次。无论树的形状如何（完全平衡或完全不平衡），只要节点总数为 n，就需要进行 n 次访问。

搜索（Search）、插入（Insertion） 和 删除（Deletion）：

时间复杂度是树的高度。

树的高度是从根节点到最远叶子节点的最长路径上的节点数。

最坏情况下，树的高度是O ( n ) O(n)O(n)。

这种情况发生在树完全不平衡时，即树退化成链表形式，其中每个节点只有一个子节点（要么是左孩子，要么是右孩子）。

在这种情况下，搜索、插入和删除操作需要访问从根节点到目标节点的路径上的所有节点，导致时间复杂度为O ( n ) O(n)O(n)。

平衡二叉搜索树（如 AVL 树或红黑树）：通过保持树的平衡，可以确保树的高度尽可能小。

在平衡二叉搜索树中，树的高度通常是O ( l o g n ) O(log n)O(logn)，这显著提高了搜索、插入和删除操作的效率。

1.2 霍夫曼编码

在ASCII（美国信息交换标准代码）中，每个字符都用8位（1字节）来编码。例如，字符"M"被编码为二进制的01001101。

霍夫曼编码通过为更频繁出现的字符使用更少的位来压缩数据。

例如，单词"Mississippi"在ASCII编码中需要11字节（88位），但在霍夫曼编码中，如果根据字符出现的频率构建编码，可能只需要22位（约3字节）。

霍夫曼编码的编码是基于文本中字符的出现频率，使用一种特殊的二叉树——霍夫曼编码树来构建的。

在霍夫曼树中，每个内部节点（非叶子节点）都有两个子节点，分别代表向左和向右的边。

这些边被赋予二进制值，通常是0和1。左边缘通常被赋予值0，右边缘被赋予值1，但这个规则可以根据具体实现而变化。

在霍夫曼树中，每个字符都对应于树的一个叶子节点（树的最末端节点，没有子节点）。

叶子节点包含了要编码的字符和该字符在原始数据中的频率。

每个字符的编码是由从树的根节点到该字符对应叶子节点的路径上的边值（0或1）组成的。

路径上的每个边值按顺序连接起来，形成该字符的霍夫曼编码。

下图展示了一个例子。

构件霍夫曼树的过程使用贪心策略，详细步骤如下：

从一片森林（forest）开始，每棵树（tree）只包含一个节点（node），该节点代表一个字符。

节点的权重（weight）是字符在文本中的出现频率。

重复以下步骤，直到只剩下一棵树（即霍夫曼树）：

从森林中选择两棵权重最小的树。这通常通过优先队列（priority queue）实现，优先队列可以用堆（heap）数据结构来实现，以便于快速找到最小元素。

创建一个新节点作为这两棵树的父节点。

新树的权重是这两棵子树权重的和。

下图展示了这个过程。

相关的代码如下。

import java.util.Scanner;

public class HuffmanCode {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

System.out.print("Enter a text: ");

String text = input.nextLine();

int[] counts = getCharacterFrequency(text); // Count frequency

System.out.printf("%-15s%-15s%-15s%-15s\n",

"ASCII Code", "Character", "Frequency", "Code");

Tree tree = getHuffmanTree(counts); // Create a Huffman tree

String[] codes = getCode(tree.root); // Get codes

for (int i = 0; i < codes.length; i++)

if (counts[i] != 0) // (char)i is not in text if counts[i] is 0

System.out.printf("%-15d%-15s%-15d%-15s\n",

i, (char)i + "", counts[i], codes[i]);

}

/\*\* Get Huffman codes for the characters

\* This method is called once after a Huffman tree is built

\*/

public static String[] getCode(Tree.Node root) {

if (root == null) return null;

String[] codes = new String[2 \* 128];

assignCode(root, codes);

return codes;

}

/\* Recursively get codes to the leaf node \*/

private static void assignCode(Tree.Node root, String[] codes) {

if (root.left != null) {

root.left.code = root.code + "0";

assignCode(root.left, codes);

root.right.code = root.code + "1";

assignCode(root.right, codes);

}

else {

codes[(int)root.element] = root.code;

}

}

/\*\* Get a Huffman tree from the codes \*/

public static Tree getHuffmanTree(int[] counts) {

// Create a heap to hold trees

Heap<Tree> heap = new Heap<>(); // Defined in Listing 24.10

for (int i = 0; i < counts.length; i++) {

if (counts[i] > 0)

heap.add(new Tree(counts[i], (char)i)); // A leaf node tree

}

while (heap.getSize() > 1) {

Tree t1 = heap.remove(); // Remove the smallest weight tree

Tree t2 = heap.remove(); // Remove the next smallest weight

heap.add(new Tree(t1, t2)); // Combine two trees

}

return heap.remove(); // The final tree

}

/\*\* Get the frequency of the characters \*/

public static int[] getCharacterFrequency(String text) {

int[] counts = new int[256]; // 256 ASCII characters

for (int i = 0; i < text.length(); i++)

counts[(int)text.charAt(i)]++; // Count the character in text

return counts;

}

/\*\* Define a Huffman coding tree \*/

public static class Tree implements Comparable<Tree> {

Node root; // The root of the tree

/\*\* Create a tree with two subtrees \*/

public Tree(Tree t1, Tree t2) {

root = new Node();

root.left = t1.root;

root.right = t2.root;

root.weight = t1.root.weight + t2.root.weight;

}

/\*\* Create a tree containing a leaf node \*/

public Tree(int weight, char element) {

root = new Node(weight, element);

}

@Override /\*\* Compare trees based on their weights \*/

public int compareTo(Tree t) {

if (root.weight < t.root.weight) // Purposely reverse the order

return 1;

else if (root.weight == t.root.weight)

return 0;

else

return -1;

}

public class Node {

char element; // Stores the character for a leaf node

int weight; // weight of the subtree rooted at this node

Node left; // Reference to the left subtree

Node right; // Reference to the right subtree

String code = ""; // The code of this node from the root

/\*\* Create an empty node \*/

public Node() {

}

/\*\* Create a node with the specified weight and character \*/

public Node(int weight, char element) {

this.weight = weight;

this.element = element;

}

}

}

}

十一！！！！！！！！（AVL TREES AVL树）

1. AVL树

前一章说到关于二叉树操作的时间复杂度时，我们提到AVL树可以改善二叉搜索树（BST）的性能。

在二叉搜索树中，搜索、插入和删除操作的时间复杂度取决于树的高度。

在最坏的情况下，如果树退化成链表（即树的高度为O(n)），这些操作的时间复杂度也将是O ( n ) O(n)O(n)。

理论上，我们可以维护一个完美平衡的树（即完全二叉树），这样树的高度将是O ( l o g n ) O(log n)O(logn)，从而确保搜索、插入和删除操作的时间复杂度为O ( l o g n ) O(log n)O(logn)。

完全二叉树的每一层都完全填满，除了可能的最后一层，这一层从左到右填充。

但维护一个完全平衡的树需要在每次插入或删除操作后进行大量的重建和节点移动，这在实践中是非常昂贵的。

AVL树是一种自平衡二叉搜索树，其中每个节点的两个子树的高度最多相差1 11。

这种平衡条件确保了树的高度保持在O ( l o g n ) O(log n)O(logn)，从而保证了操作的时间复杂度为O ( l o g n ) O(log n)O(logn)。

与完全平衡的树相比，AVL树需要的平衡努力要少得多，因为它们不需要在每次操作后都进行大量的重建和节点移动。

AVL树通过在插入和删除操作中执行旋转来保持平衡，这些旋转操作相对简单且高效。

平衡因子（Balance Factor）：

平衡因子是一个节点的右子树高度减去左子树高度的结果。

如果一个节点的平衡因子是-1、0或1，则该节点被认为是平衡的，即平衡节点（Balanced Node）。

如果一个节点的平衡因子是-1，即右子树比左子树高1，那么这个节点被认为是左偏的，即左偏节点（Left-Heavy Node）。

如果一个节点的平衡因子是+1，即左子树比右子树高1，那么这个节点被认为是右偏的，即右偏节点（Right-Heavy Node）。

1.1 旋转操作

如果一个节点的平衡因子不再平衡，那么就需要通过旋转操作来重新平衡这个节点。

一共有四种可能的旋转操作：

LL旋转（左-左旋转，Left-Left Rotation）。

RR旋转（右-右旋转，Right-Right Rotation）。

LR旋转（左-右旋转，Left-Right Rotation）。

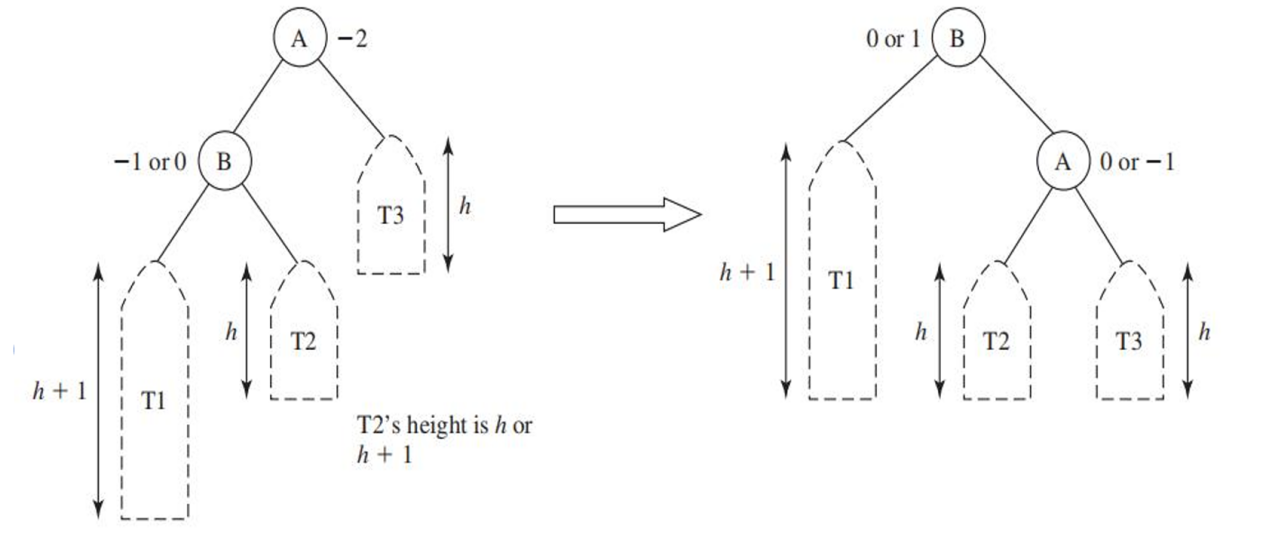
RL旋转（右-左旋转，Right-Left Rotation）。

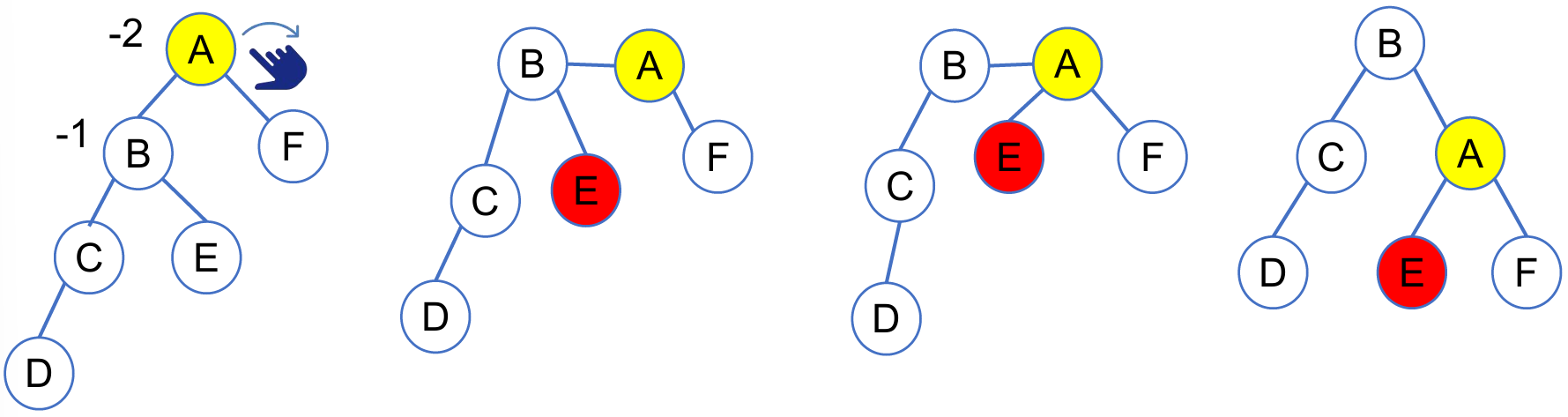
关于旋转的操作也可以查看算法文章里的讲解：AVL树讲解

1.1.1 LL旋转

当一个节点A的平衡因子为-2时，我们称该节点为左偏（left-heavy）。如果这个左偏的节点A的左子节点B的平衡因子为-1或0，那么节点A就处于LL不平衡状态（Left-Left Imbalance）。

为此我们使用LL旋转用来解决这种不平衡。





初始状态：节点A是不平衡的，其平衡因子为-2（左子树比右子树高2层），并且其左子节点B的平衡因子为-1或0（也是左偏或平衡）。

图中用黄色标记了节点A，用蓝色标记了节点B，用红色标记了节点C。

右旋：首先，将节点A向右旋转，使得原本不受旋转影响的节点F保持在原位置，仍然是节点A的右子节点。

在旋转过程中，节点B将取代节点A的位置，成为新的根节点。

调整位置：节点A在旋转后将向下移动，成为节点B的右子节点。

但是，节点B已经有一个右子节点E，这会导致冲突。

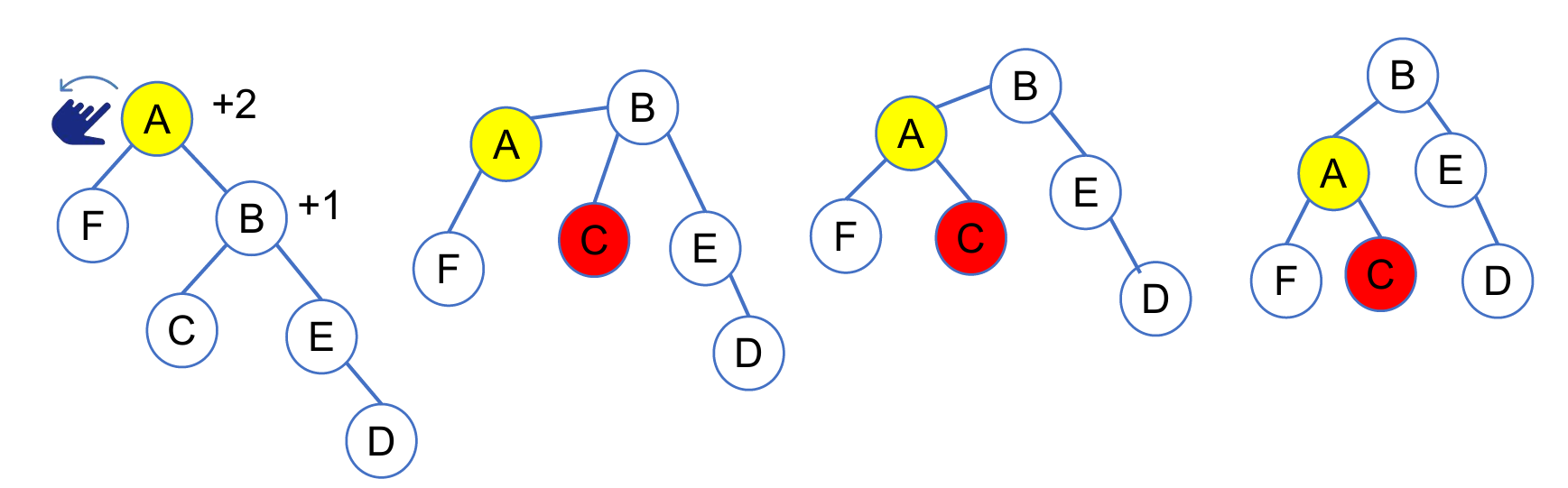
避免冲突：为了避免冲突，节点E将在旋转过程中被附加到节点A的左子节点位置。

这样，节点A现在有了左子节点E，而节点B成为了新的根节点，节点A是其右子节点。

1.1.2 RR旋转

当一个节点A的平衡因子为+2时，我们称该节点为右偏（right-heavy）。如果这个右偏的节点A的右子节点B的平衡因子为+1或0，那么节点A就处于RR不平衡状态（Right-Right Imbalance）。

为此我们使用RR旋转用来解决这种不平衡。



初始状态：节点A的平衡因子为+2，表示它是一个右偏（right-heavy）节点，其右子节点B的平衡因子为+1或0，表示B也是右偏或平衡的。

图中用黄色标记了节点A，用蓝色标记了节点B，用红色标记了节点C。

左旋：首先，将节点A向左旋转，使得原本不受旋转影响的节点F保持在原位置，仍然是节点A的左子节点。

调整位置：在旋转过程中，节点B将取代节点A的位置，成为新的根节点。

节点A在旋转后将向下移动，成为节点B的左子节点。

避免冲突：为了避免冲突，节点C（B的左子节点）将在旋转过程中被附加到节点A的右子节点位置。

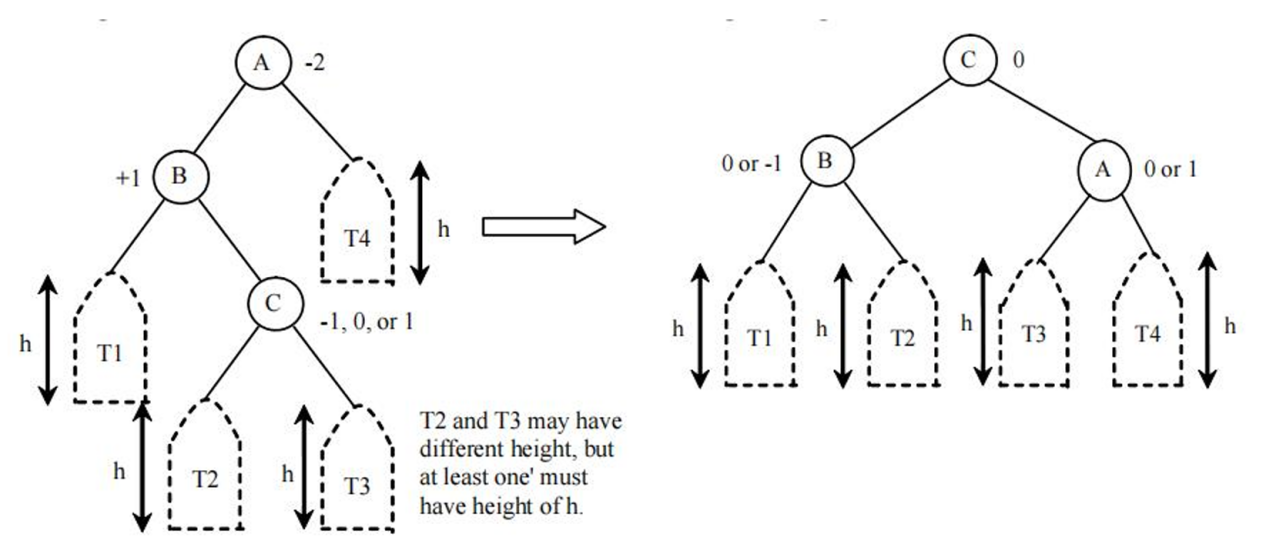
1.1.3 LR旋转

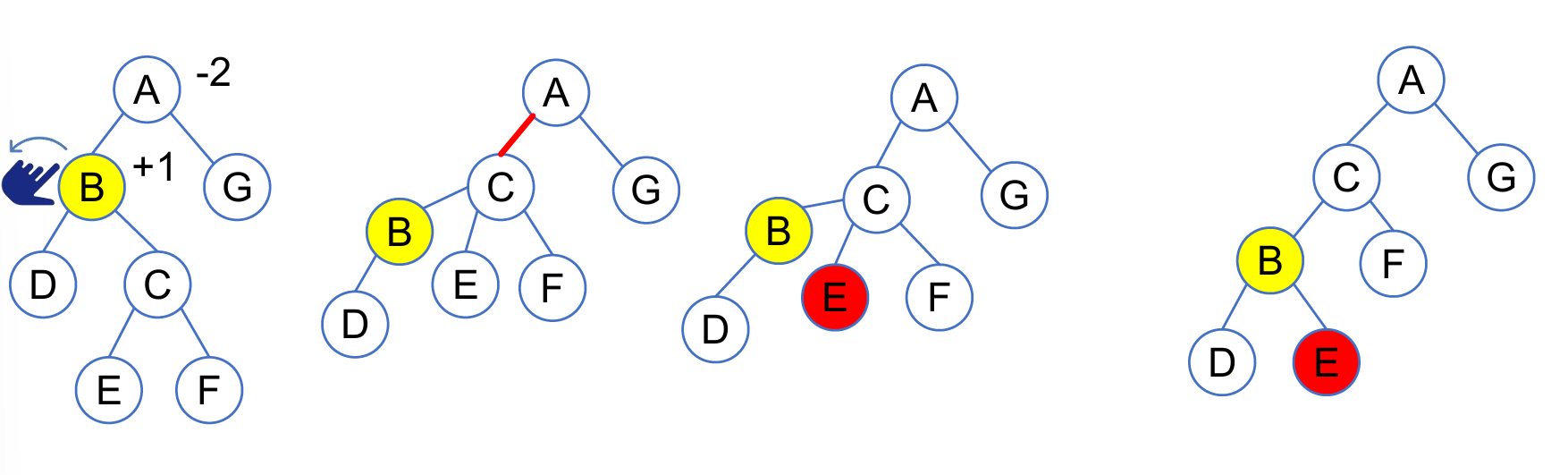
LR不平衡：节点A的平衡因子为-2，意味着它的左子树比右子树高2层。同时，节点A的左子节点B的平衡因子为+1，意味着B的左子树比右子树低1层。

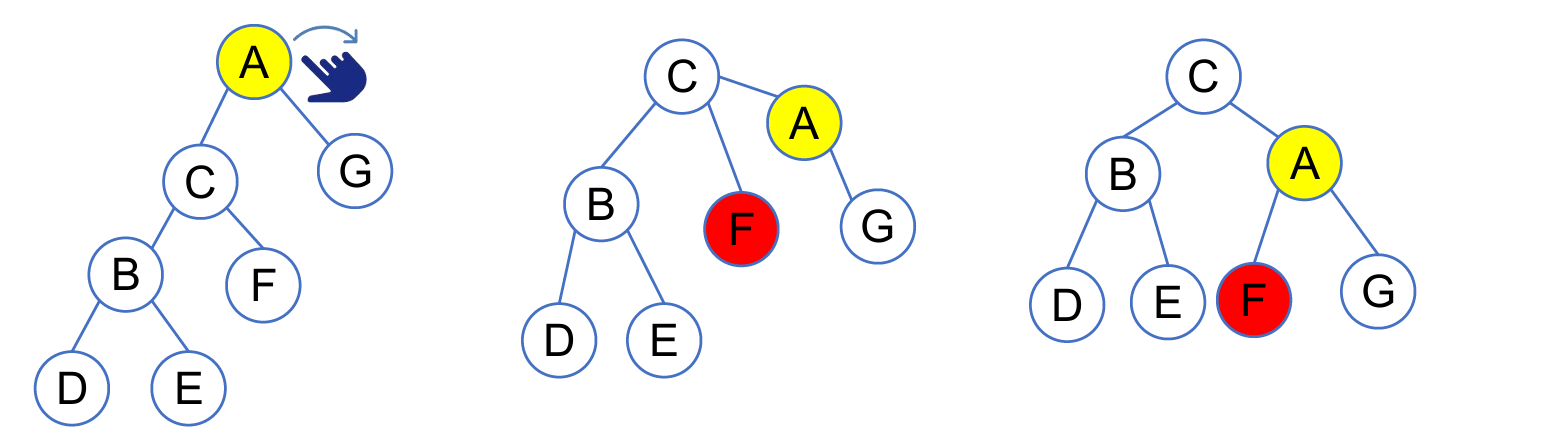
为此我们使用LR旋转用来解决这种不平衡。LR旋转是一种双旋转操作，用于解决LR不平衡。

首先对节点B进行左旋转，以减少B的右偏程度。这会使得B的右子节点成为新的子树的根，而B则成为这个新根的左子节点。

接着对节点A进行右旋转，以解决A的左偏问题。这会使得A的左子节点（现在已经是经过左旋转的B）成为新的子树的根，而A则成为这个新根的右子节点。







初始状态：节点A的平衡因子为-2，节点B的平衡因子为+1。

需要对A的左子树B进行左旋转，然后对A进行右旋转。

图中用黄色标记了节点A，用蓝色标记了节点B，用红色标记了节点C。

对B进行左旋转：左旋转A的左子树B，使得B与A断开连接。

C节点（B的右子节点）取代B的位置，成为A的左子节点。

B成为C的左子节点。

对A进行右旋转：由于B已经与A断开，并且C现在位于B原来的位置，我们需要对A进行右旋转。

在右旋转过程中，A会与它的右子节点F发生冲突。

避免冲突：为了避免冲突，将E（B的右子节点）附加到B作为右子节点，完成对B的旋转。

然后将F（A的右子节点）附加到A作为左子节点，完成对A的旋转。

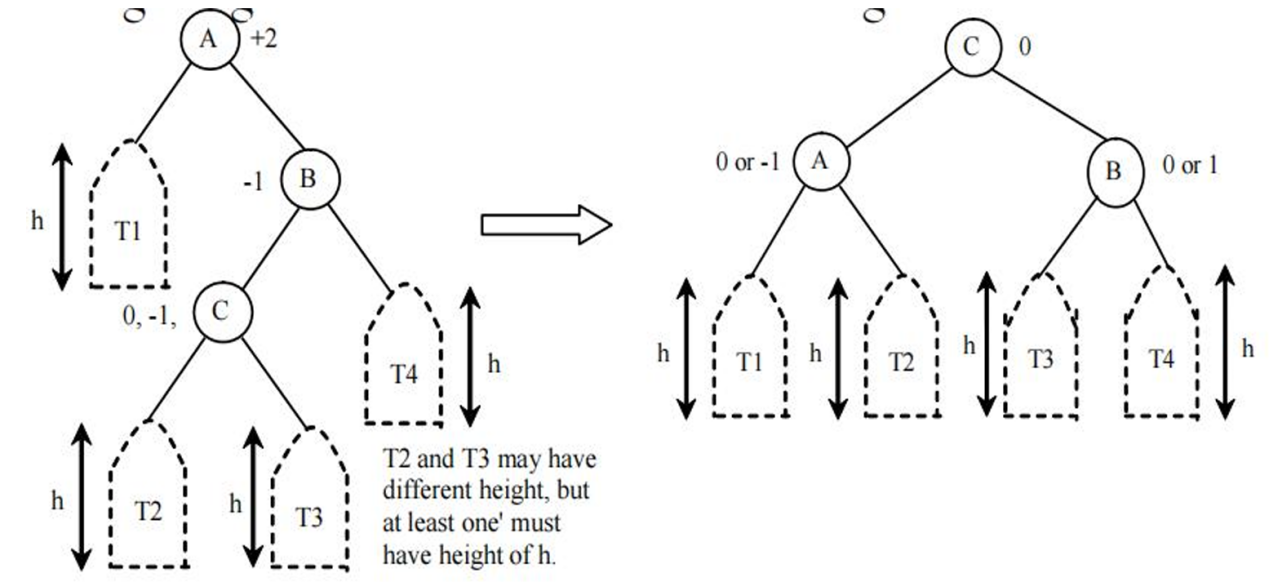
1.1.4 RL旋转

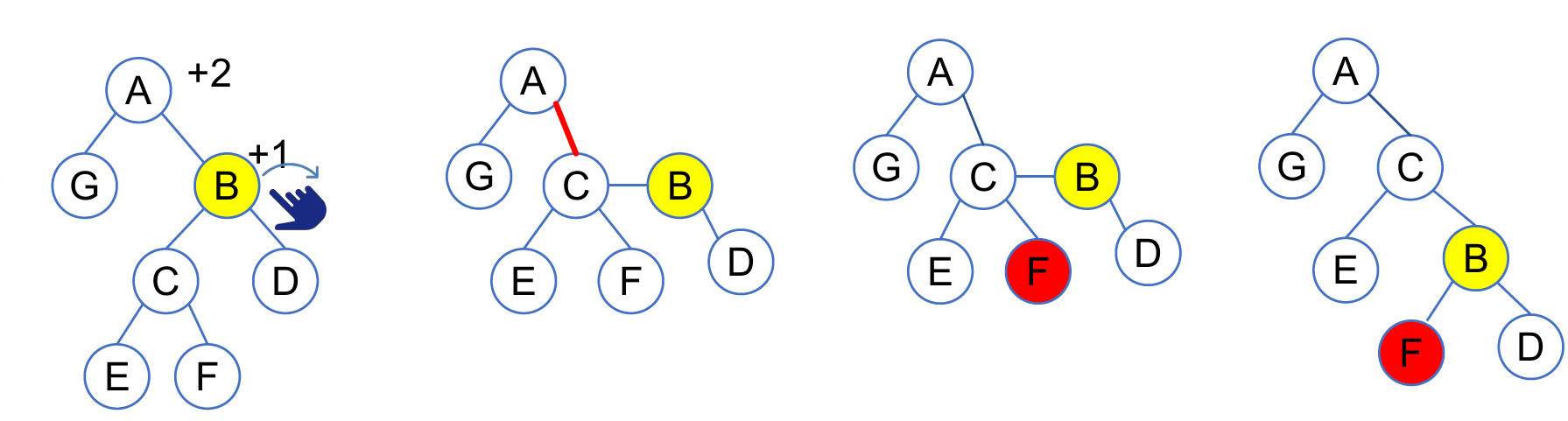
RL不平衡：节点A的平衡因子为+2，意味着它的右子树比左子树高2层。同时，节点A的右子节点B的平衡因子为-1，意味着B的右子树比左子树低1层。

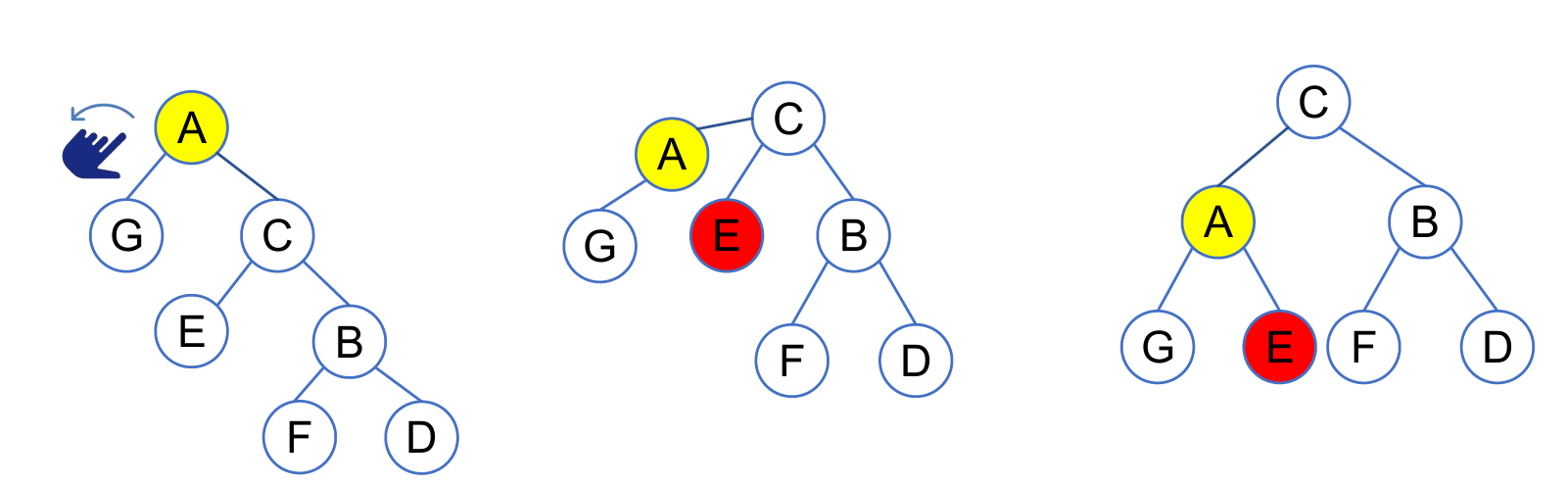
为此我们使用RL旋转用来解决这种不平衡。RL旋转也是一种双旋转操作。

首先对节点B进行右旋转，以减少B的左偏程度。这会使得B的左子节点成为新的子树的根，而B则成为这个新根的右子节点。

接着对节点A进行左旋转，以解决A的右偏问题。这会使得A的右子节点（现在已经是经过右旋转的B）成为新的子树的根，而A则成为这个新根的左子节点。







初始状态：节点A的平衡因子为+2，表示它是一个右偏（right-heavy）节点，其右子节点B的平衡因子为+1，表示B也是右偏的。

图中用黄色标记了节点A和B，用红色标记了节点C、E和F。

右旋转B：首先对节点B进行右旋转，使得B与A断开连接。

C节点（B的左子节点）取代B的位置，成为A的右子节点。

B成为C的右子节点。

左旋转A：接着对节点A进行左旋转，以解决A的右偏问题。

在左旋转过程中，A会与它的左子节点E发生冲突。

调整节点位置：为了避免冲突，将F（C的左子节点）附加到B作为左子节点，完成对B的旋转。

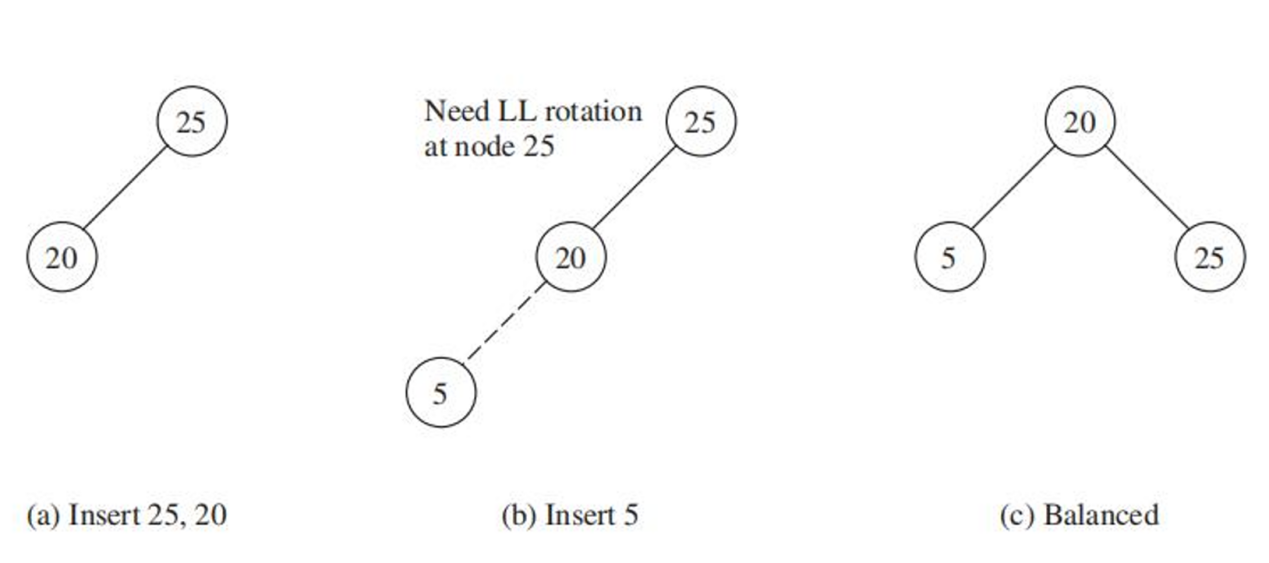
然后将E（A的左子节点）附加到A作为右子节点，完成对A的旋转。

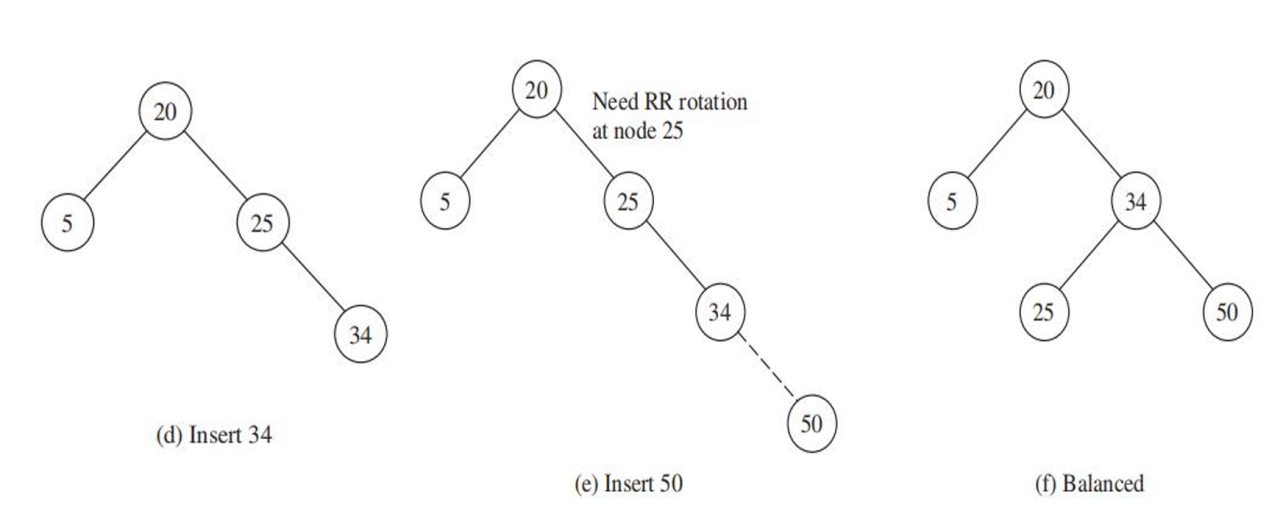
对于后两种复杂的旋转其实还是建议参考算法课上的AVL旋转操作。AVL树讲解

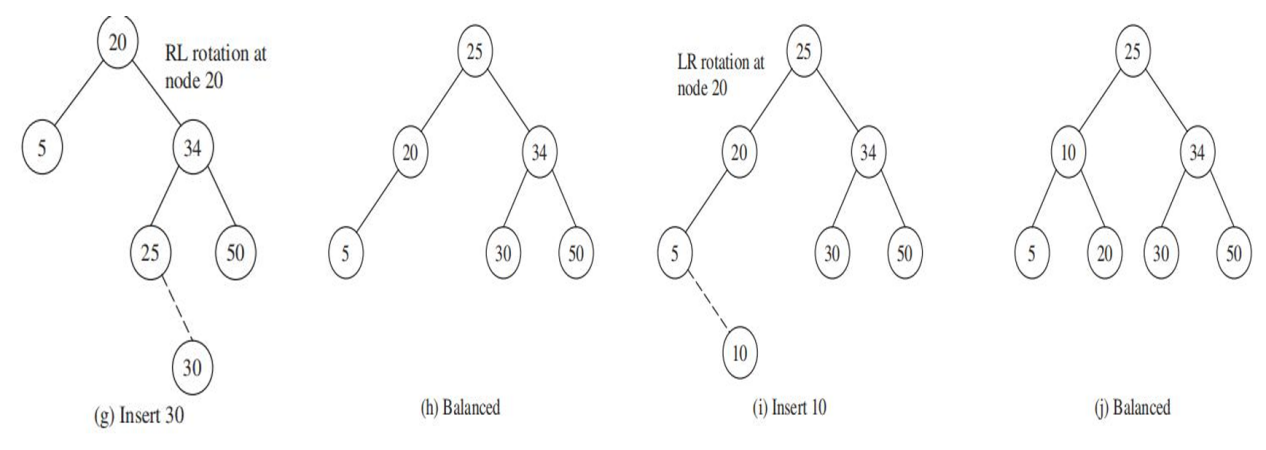
1.2 插入操作

插入每一个节点后，如果出现不平衡节点，我们就使用旋转操作保持AVL树的平衡。

下图展示了插入25，20，5，34，50，30，10的全过程。

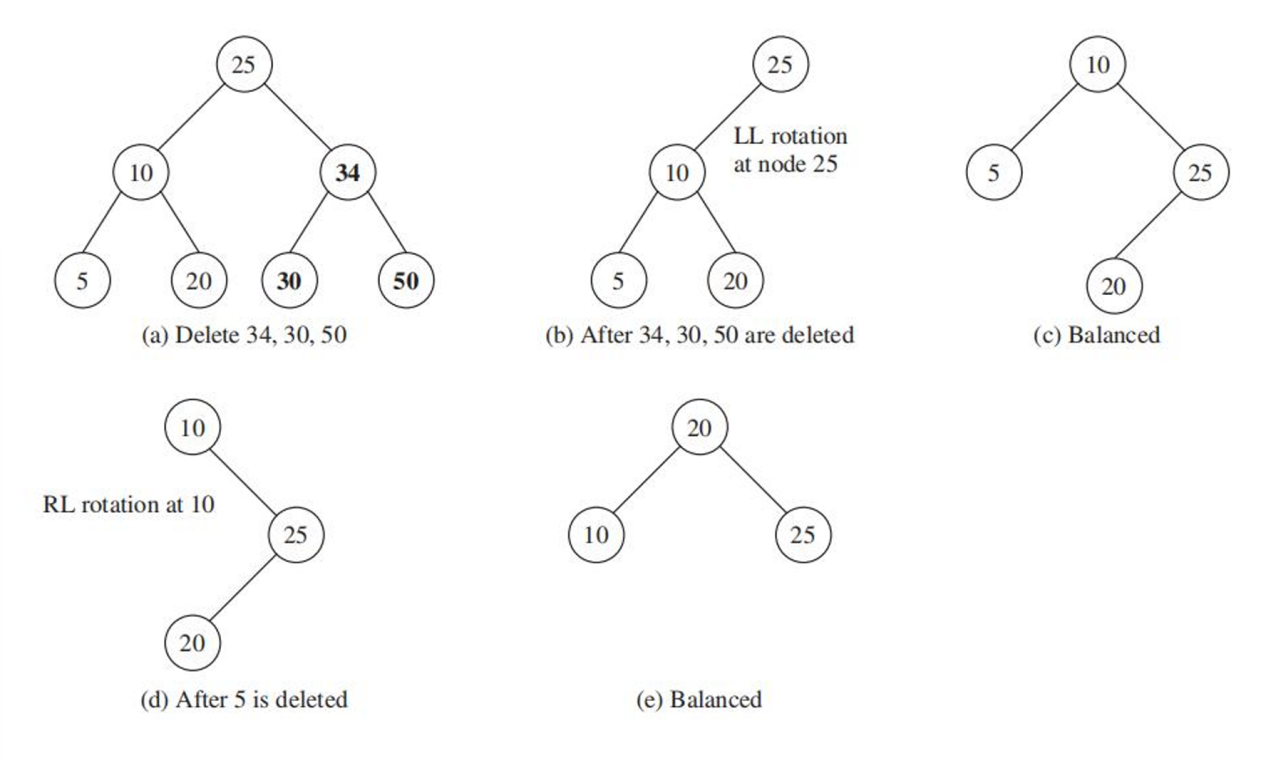






### **1.3 删除操作**

删除操作同理，当AVL树不平衡时，通过旋转操作让AVL树平衡。  
下面展示了删除34，30，50，5的全过程。



1.4 实现AVL树

AVL树同样还是一个二叉搜索树，所以我们可以让AVLTree类继承BST类。

代码如下。

public class AVLTree<E> extends BST<E> {

/\*\* Create an empty AVL tree using a natural comparator\*/

public AVLTree() { // super() is implicitly called

}

/\*\* Create a BST with a specified comparator \*/

public AVLTree(java.util.Comparator<E> c) {

super(c);

}

/\*\* Create an AVL tree from an array of objects \*/

public AVLTree(E[] objects) {

super(objects);

}

@Override /\*\* Override createNewNode to create an AVLTreeNode \*/

protected AVLTreeNode<E> createNewNode(E e) {

return new AVLTreeNode<E>(e);

}

@Override /\*\* Insert an element and rebalance if necessary \*/

public boolean insert(E e) {

boolean successful = super.insert(e);

if (!successful)

return false; // e is already in the tree

else {

balancePath(e); // Balance from e to the root if necessary

}

return true; // e is inserted

}

/\*\* Update the height of a specified node \*/

private void updateHeight(AVLTreeNode<E> node) {

if (node.left == null && node.right == null) // node is a leaf

node.height = 0;

else if (node.left == null) // node has no left subtree

node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;

else if (node.right == null) // node has no right subtree

node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;

else

node.height = 1 +

Math.max(((AVLTreeNode<E>)(node.right)).height,

((AVLTreeNode<E>)(node.left)).height);

}

/\*\* Balance the nodes in the path from the specified

\* node to the root if necessary

\*/

private void balancePath(E e) {

java.util.ArrayList<TreeNode<E>> path = path(e);

for (int i = path.size() - 1; i >= 0; i--) {

AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));

updateHeight(A);

AVLTreeNode<E> parentOfA = (A == root) ? null :

(AVLTreeNode<E>)(path.get(i - 1));

switch (balanceFactor(A)) {

case -2:

if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) {

balanceLL(A, parentOfA); // Perform LL rotation

}

else {

balanceLR(A, parentOfA); // Perform LR rotation

}

break;

case +2:

if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) {

balanceRR(A, parentOfA); // Perform RR rotation

}

else {

balanceRL(A, parentOfA); // Perform RL rotation

}

}

}

}

/\*\* Return the balance factor of the node \*/

private int balanceFactor(AVLTreeNode<E> node) {

if (node.right == null) // node has no right subtree

return -node.height;

else if (node.left == null) // node has no left subtree

return +node.height;

else

return ((AVLTreeNode<E>)node.right).height -

((AVLTreeNode<E>)node.left).height;

}

/\*\* Balance LL (see Figure 26.3) \*/

private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.left; // A is left-heavy and B is left-heavy

if (A == root) {

root = B;

}

else {

if (parentOfA.left == A) {

parentOfA.left = B;

}

else {

parentOfA.right = B;

}

}

A.left = B.right; // Make T2 the left subtree of A

B.right = A; // Make A the left child of B

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

}

/\*\* Balance LR (see Figure 26.5) \*/

private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.left; // A is left-heavy

TreeNode<E> C = B.right; // B is right-heavy

if (A == root) {

root = C;

}

else {

if (parentOfA.left == A) {

parentOfA.left = C;

}

else {

parentOfA.right = C;

}

}

A.left = C.right; // Make T3 the left subtree of A

B.right = C.left; // Make T2 the right subtree of B

C.left = B;

C.right = A;

// Adjust heights

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

updateHeight((AVLTreeNode<E>)C);

}

/\*\* Balance RR (see Figure 26.4) \*/

private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy

if (A == root) {

root = B;

}

else {

if (parentOfA.left == A) {

parentOfA.left = B;

}

else {

parentOfA.right = B;

}

}

A.right = B.left; // Make T2 the right subtree of A

B.left = A;

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

}

/\*\* Balance RL (see Figure 26.6) \*/

private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {

TreeNode<E> B = A.right; // A is right-heavy

TreeNode<E> C = B.left; // B is left-heavy

if (A == root) {

root = C;

}

else {

if (parentOfA.left == A) {

parentOfA.left = C;

}

else {

parentOfA.right = C;

}

}

A.right = C.left; // Make T2 the right subtree of A

B.left = C.right; // Make T3 the left subtree of B

C.left = A;

C.right = B;

// Adjust heights

updateHeight((AVLTreeNode<E>)A);

updateHeight((AVLTreeNode<E>)B);

updateHeight((AVLTreeNode<E>)C);

}

@Override /\*\* Delete an element from the binary tree.

\* Return true if the element is deleted successfully

\* Return false if the element is not in the tree \*/

public boolean delete(E element) {

if (root == null)

return false; // Element is not in the tree

// Locate the node to be deleted and also locate its parent node

TreeNode<E> parent = null;

TreeNode<E> current = root;

while (current != null) {

if (c.compare(element, current.element) < 0) {

parent = current;

current = current.left;

}

else if (c.compare(element, current.element) > 0) {

parent = current;

current = current.right;

}

else

break; // Element is in the tree pointed by current

}

if (current == null)

return false; // Element is not in the tree

// Case 1: current has no left children (See Figure 23.6)

if (current.left == null) {

// Connect the parent with the right child of the current node

if (parent == null) {

root = current.right;

}

else {

if (c.compare(element, parent.element) < 0)

parent.left = current.right;

else

parent.right = current.right;

// Balance the tree if necessary

balancePath(parent.element);

}

}

else {

// Case 2: The current node has a left child

// Locate the rightmost node in the left subtree of

// the current node and also its parent

TreeNode<E> parentOfRightMost = current;

TreeNode<E> rightMost = current.left;

while (rightMost.right != null) {

parentOfRightMost = rightMost;

rightMost = rightMost.right; // Keep going to the right

}

// Replace the element in current by the element in rightMost

current.element = rightMost.element;

// Eliminate rightmost node

if (parentOfRightMost.right == rightMost)

parentOfRightMost.right = rightMost.left;

else

// Special case: parentOfRightMost is current

parentOfRightMost.left = rightMost.left;

// Balance the tree if necessary

balancePath(parentOfRightMost.element);

}

size--;

return true; // Element inserted

}

}

测试代码如下。

package chapter26;

import chapter25.BST;

public class TestAVLTree {

public static void main(String[] args) {

// Create an AVL tree

AVLTree<Integer> tree = new AVLTree<>(new Integer[]{25,

20, 5});

System.out.print("After inserting 25, 20, 5:");

printTree(tree);

tree.insert(34);

tree.insert(50);

System.out.print("\nAfter inserting 34, 50:");

printTree(tree);

tree.insert(30);

System.out.print("\nAfter inserting 30");

printTree(tree);

tree.insert(10);

System.out.print("\nAfter inserting 10");

printTree(tree);

tree.delete(34);

tree.delete(30);

tree.delete(50);

System.out.print("\nAfter removing 34, 30, 50:");

printTree(tree);

tree.delete(5);

System.out.print("\nAfter removing 5:");

printTree(tree);

System.out.print("\nTraverse the elements in the tree: ");

for (int e: tree) {

System.out.print(e + " ");

}

}

public static void printTree(BST tree) {

// Traverse tree

System.out.print("\nInorder (sorted): ");

tree.inorder();

System.out.print("\nPostorder: ");

tree.postorder();

System.out.print("\nPreorder: ");

tree.preorder();

System.out.print("\nThe number of nodes is " + tree.getSize());

System.out.println();

}

}

1.3 时间复杂度分析

AVL树的时间复杂度分析表明，无论是搜索、插入还是删除操作，它们的时间复杂度都是O ( l o g n ) O(log n)O(logn)。这是因为：

AVL树的高度是O ( l o g n ) O(log n)O(logn)，这意味着从根节点到任何叶子节点的路径长度最多是O ( l o g n ) O(log n)O(logn)。

在balancePath()方法中，尽管需要遍历从新节点到根节点的路径，但每个节点上的操作（如更新高度、计算平衡因子和执行旋转）都是常数时间操作，不依赖于树的大小。

因此，AVL树能够保持高效的操作性能，即使在动态变化（插入和删除）的情况下。

2. 哈希（Hashing）

哈希在集合（Sets）和映射（Maps）中被使用。

如果不使用哈希，检查一个元素是否存在于集合中或从映射中检索一个值，可能需要遍历整个集合或映射。

哈希函数可以将每个元素或键映射到一个特定的索引，这样我们就可以跳转到元素或键应该在的位置。

哈希函数的输出通常称为哈希码（Hash Code）或哈希值（Hash Value）。

哈希的主要优势在于它提供了快速的访问速度。

平均情况下，我们可以在O(1)时间内搜索、插入和删除元素（不需要搜索和比较）。

2.1 哈希的实现

我们先说到数组，它允许通过索引（index）直接访问其元素。如果我们知道数组中某个元素的索引，那么我们可以在常数时间内O ( 1 ) O(1)O(1)访问或更新这个元素。

哈希表是一种使用数组来存储数据的实现方式，其中的数组被称为哈希表。哈希表通过哈希函数将键（key）映射到数组的索引，从而实现快速的数据访问。

哈希函数接受一个搜索键（Search Key）作为输入，并将其转换为一个整数值，这个整数值被称为哈希码（Hash Code）。

然后使用压缩（compressing）将输入数据映射到一个固定范围内的整数，这个整数通常用作哈希表（Hash Table）的索引。

下面的代码将得到一个哈希码。

Object o = new Object();

System.out.println(o.hashCode());

由于哈希码可能是一个非常大的数，而哈希表的大小通常是有限的（例如，我们通常不会有一个包含超过3亿个位置的哈希表），因此需要对哈希码进行压缩，以将其转换为哈希表的有效索引。

2.1 为基本数据类型生成哈希码

对于这些较小的整数类型（byte、short、char和int类型），可以直接转换为int类型来生成哈希码。

例子如下。

char a = ‘A’;

int charHash = (int) a;

对于float类型，可以使用Float.floatToIntBits(f)方法将其转换为原始的整数位表示形式，然后使用这个整数作为哈希码。

例子如下。

System.out.println(Float.floatToIntBits(1.23f))

对于long类型，由于其大小为64位，而int类型只能容纳32位，因此需要将long类型的高32位和低32位组合起来生成哈希码。

这可以通过使用XOR（异或）操作来完成，以确保结果适合int类型。

下面的代码演示了这一操作。

int hashCode = (int)(key ^ (key >>> 32));

例如，long value1 = 0xFFFFFFFFABCDEF00L;和long value2 = 0x12345678ABCDEF00L;直接转换为int类型都会得到ABCDEF00L，即直接丢失高32位的信息，只保留低32位的信息。

按照我们说的解决方案的结果如下。

double类型是64位的，可以使用Double.doubleToLongBits(value)方法将其转换为一个long类型值。这个方法返回一个表示double值的64位二进制补码表示形式。

所以现在使用使用与long类型相同的XOR操作来生成哈希码。

int hashCode = (int)(bits ^ (bits >>> 32))。

1

这确保了哈希码能够保留double值的高32位和低32位的信息。

对于String类型，生成哈希码的步骤如下：

初始值：初始化一个hashCode变量为0。

逐字符处理：对于字符串中的每个字符，执行以下操作：

将当前的hashCode值乘以一个固定的质数（例如31、33、41等）。

将当前字符的整数值加到结果上。

这个过程累积了每个字符的效果，使得哈希码对字符及其顺序都敏感。

int hashCode = 0;

for (int i = 0; i < length; i++) {

hashCode = 31 \* hashCode + charAt(i);

}

2.2 压缩（Compressing）

哈希码可能是一个大整数，超出了哈希表索引的范围。

例如，如果哈希表的大小是$11（索引从0 00到10 1010），哈希码可能是一个大整数，如366712642 366712642366712642。

我们需要将哈希码缩小以适应索引的范围。

假设哈希表的索引在0 00和N − 1 N-1N−1之间，其中N NN是哈希表的大小（容量），定义了可用于存储元素的槽数。

最常见的缩放整数的方法是使用模运算（modulus operator）：h ( h a s h C o d e ) = h a s h C o d e % N h(hashCode) = hashCode \% Nh(hashCode)=hashCode%N。

N NN是哈希表的大小，通常选择一个质数（例如11、13、31等）以确保索引均匀分布。

选择质数作为哈希表的大小有助于减少哈希冲突，因为质数在数学上具有某些理想的分布属性。

2.3 冲突处理（Handling Collisions）

当两个或多个键被哈希函数映射到哈希表的同一索引时，就会发生冲突。

例如哈希值是1 11和12 1212，而哈希表的大小是11 1111，所以12 1212被压缩成了1 11，这时候就发生了冲突。

处理冲突的两种主要方法为开放寻址（Open Addressing）和分离链接（Separate Chaining）。

开放寻址（Open Addressing）是在发生冲突时，寻找哈希表中的另一个开放位置的过程。

开放寻址有许多变体：

线性探测（Linear Probing）：从冲突的索引开始，按照固定步长（通常是1）顺序查找下一个开放位置。

二次探测（Quadratic Probing）：按照二次函数（如i^2）的序列查找开放位置。

双重哈希（Double Hashing）：使用第二个哈希函数来确定探测序列，通常提供比线性或二次探测更好的性能。

分离链接（Separate Chaining）是将所有具有相同哈希索引的条目放在同一个位置的链表中。

2.3.1 线性探测（Linear Probing）

如果在哈希表的hashTable[key % N]位置发生冲突，线性探测会检查下一个连续的位置hashTable[(key+1) % N]是否可用。

如果这个位置也不可用，继续检查hashTable[(key+2) % N]，依此类推，直到找到一个可用的单元格。

这里不直接+1而是还保留模运算的原因是确保索引值在哈希表的大小范围内。

下图展示了一个例子。

当需要从哈希表中删除一个条目时，首先搜索与键匹配的条目。

如果找到了匹配的条目，不是直接将其删除，而是在该位置放置一个特殊的标记，以表示该条目已被删除，但该位置仍然可以用于插入其他值。

这样做的原因是为了保持哈希表中元素的顺序，以便在后续的搜索、插入或删除操作中，可以通过线性探测策略找到正确的位置。

因此在哈希表中，每个单元格可以处于三种状态之一：被占用（occupied）、被标记（marked）、或为空（empty）。

被标记的单元格也是可用于插入新元素的。

线性探测导致了聚类问题：线性探测倾向于导致哈希表中连续的单元格被占用，这些连续占用的单元格形成的组被称为聚类或簇（cluster）。

当发生冲突时，新元素总是被放置在附近的空单元格中（例如k%N, (k+1)%N, (k+2)%N等）。

这种邻近放置的策略导致占用的单元格累积并形成聚类。

每个聚类实际上是一个探测序列，当检索、添加或删除条目时，都必须搜索这个序列。

这是线性探测的一个主要缺点，因为它增加了操作的复杂度和时间成本，特别是在聚类很长时。

例如我们对于前面的哈希表，再添加37 3737的时候会进行额外的操作，因为原来的单元格已被占用。

2.3.2 二次探测（Quadratic Probing）

二次探测不是简单地线性地查找下一个位置，而是按照二次函数的序列来查找。这个序列通常形式为( k e y + j 2 ) % N (key+j^2 )\%N(key+j

2

)%N，其中j jj是一个非负整数（0, 1, 2, 3, …），N NN是哈希表的大小。

例如，如果k e y keykey是哈希码计算出的索引，那么探测的序列将是k e y keykey，( k e y + 1 2 ) % N (key+1^2)\%N(key+1

2

)%N，( k e y + 2 2 ) % N (key+2^2 )\%N(key+2

2

)%N，( k e y + 3 2 ) % N (key+3^2 )\%N(key+3

2

)%N，依此类推。

所以像之前的例子，26 2626将会在第二步后直接添加。

2.3.3 双重哈希（Double Hashing）

双重哈希使用两个哈希函数来确定键在哈希表中的存储位置和冲突解决时的探测序列。

第一个哈希函数h ( k e y ) h(key)h(key)用于确定初始位置。

第二个哈希函数h ′ ( k e y ) h'(key)h

′

(key)用于确定探测序列的步长（step size），从而避免聚类（clustering）问题。

通常形式为( h ( k e y ) + j ∗ h ′ ( k e y ) ) % N (h(key)+j\*h′(key) )\%N(h(key)+j∗h′(key))%N, 例如，如果k e y keykey是哈希码计算出的索引，那么探测的序列将是h ( k e y ) + 0 ∗ h ′ ( k e y ) ) % N h(key)+0\*h′(key) )\%Nh(key)+0∗h′(key))%N，h ( k e y ) + 1 ∗ h ′ ( k e y ) ) % N h(key)+1\*h′(key) )\%Nh(key)+1∗h′(key))%N，h ( k e y ) + 2 ∗ h ′ ( k e y ) ) % N h(key)+2\*h′(key) )\%Nh(key)+2∗h′(key))%N，h ( k e y ) + 3 ∗ h ′ ( k e y ) ) % N h(key)+3\*h′(key) )\%Nh(key)+3∗h′(key))%N，依此类推。

下面给一个完整的例子。

现在第一个哈希函数是h ( k e y ) = k e y % 11 h(key)=key \% 11h(key)=key%11，第二个哈希函数是h ’ ( k e y ) = 7 − k e y % 7 h’(key)=7 - key \% 7h’(key)=7−key%7

对于k e y = 12 key=12key=12，初始位置是1 11,步长是2 22，探测序列将是1 , 3 , 5 , 7 , . . . 1,3,5,7,...1,3,5,7,...

这里需要注意h ’ ( k e y ) h’(key)h’(key)需要保证步长永远不是0 00并且与与哈希表的大小N NN互质，以确保探测序列均匀分布，从而避免了线性探测中连续键值导致的聚类问题。

例如这里是7 − k e y % 7 7 - key \% 77−key%7从而保证永远不是0 00，而7 77与哈希表的大小11 1111互质。因此这里也可以是h ’ ( k e y ) = 6 − k e y % 6 h’(key)=6 - key \% 6h’(key)=6−key%6

2.3.4 分离链接（Separate Chaining）

它将所有具有相同哈希索引的条目放置在同一个位置，而不是寻找新的位置。

在分离链接方案中，哈希表的每个位置被称为一个桶（bucket）。

桶是用来存储具有相同哈希索引的多个条目的容器。

桶可以使用数组、ArrayList或LinkedList等数据结构来实现。

例如，如果使用LinkedList，那么每个桶就是一个链表，链表的头部存储在哈希表的对应位置。

2.4 负载因子（Load Factor）和重新哈希（Rehashing）

负载因子λ λλ（通常表示为λ λλ）是哈希表中元素数量与哈希表大小（即桶的数量）之间的比率。

所以负载因子的计算公式为：λ = n / N λ= n/Nλ=n/N,其中n nn是哈希表中的元素数量，N NN是哈希表的总位置数。​

对于开放寻址（Open Addressing）方案，负载因子λ λλ的值在0 00到1 11之间。

如果哈希表为空，则λ = 0 λ=0λ=0。

如果哈希表满了（即所有位置都被占用），则λ = 1 λ=1λ=1。

对于分离链接（Separate Chaining）方案，负载因子λ λλ可以是任何值，因为它不直接影响哈希表的探测过程。分离链接方案通过链表来处理冲突，因此即使哈希表的某些位置满了，也可以通过链表来存储更多的元素。

当负载因子λ λλ增加时，冲突的概率也随之增加。

这是因为负载因子衡量了哈希表的填充程度，填充程度越高，冲突的可能性越大。

对于开放寻址（Open Addressing）方案，建议将负载因子维持在0.5 0.50.5以下。

对于分离链接（Separate Chaining）方案，负载因子可以维持在0.9 0.90.9以下。

当负载因子超过预设阈值时（例如，Java中的 HashMap 类通常设置为0.75），就需要增加哈希表的大小，并将所有条目重新加载到新的、更大的哈希表中，这个过程称为重新哈希（Rehashing）。

由于哈希表大小（N NN）已经改变，需要更新哈希函数以适应新的哈希表大小。

重新哈希是一个成本较高的操作，因为它涉及到重新计算哈希码和重新插入元素。所以为了减少重新哈希的频率，建议至少将哈希表大小加倍。

MyMap类的代码如下。

public interface MyMap<K, V> {

/\*\* Remove all of the entries from this map \*/

public void clear();

/\*\* Return true if the specified key is in the map \*/

public boolean containsKey(K key);

/\*\* Return true if this map contains the specified value \*/

public boolean containsValue(V value);

/\*\* Return a set of entries in the map \*/

public java.util.Set<Entry<K, V>> entrySet();

/\*\* Return the first value that matches the specified key \*/

public V get(K key);

/\*\* Return true if this map contains no entries \*/

public boolean isEmpty();

/\*\* Return a set consisting of the keys in this map \*/

public java.util.Set<K> keySet();

/\*\* Add an entry (key, value) into the map \*/

public V put(K key, V value);

/\*\* Remove the entries for the specified key \*/

public void remove(K key);

/\*\* Return the number of mappings in this map \*/

public int size();

/\*\* Return a set consisting of the values in this map \*/

public java.util.Set<V> values();

/\*\* Define inner class for Entry \*/

public static class Entry<K, V> {

K key;

V value;

public Entry(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() {

return key;

}

public V getValue() {

return value;

}

@Override

public String toString() {

return "[" + key + ", " + value + "]";

}

}

}

MyHashMap类的代码如下。

package org.example;

import java.util.LinkedList;

public class MyHashMap<K, V> implements MyMap<K, V> {

// Define the default hash table size. Must be a power of 2

private static int DEFAULT\_INITIAL\_CAPACITY = 4;

// Define the maximum hash table size. 1 << 30 is same as 2^30

private static int MAXIMUM\_CAPACITY = 1 << 30;

// Current hash table capacity. Capacity is a power of 2

private int capacity;

// Define default load factor

private static float DEFAULT\_MAX\_LOAD\_FACTOR = 0.75f;

// Specify a load factor used in the hash table

private float loadFactorThreshold;

// The number of entries in the map

private int size = 0;

// Hash table is an array with each cell that is a linked list

LinkedList<MyMap.Entry<K,V>>[] table;

/\*\* Construct a map with the default capacity and load factor \*/

public MyHashMap() {

this(DEFAULT\_INITIAL\_CAPACITY, DEFAULT\_MAX\_LOAD\_FACTOR);

}

/\*\* Construct a map with the specified initial capacity and

\* default load factor \*/

public MyHashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_MAX\_LOAD\_FACTOR);

}

/\*\* Construct a map with the specified initial capacity

\* and load factor \*/

public MyHashMap(int initialCapacity, float loadFactorThreshold) {

if (initialCapacity > MAXIMUM\_CAPACITY)

this.capacity = MAXIMUM\_CAPACITY;

else

this.capacity = trimToPowerOf2(initialCapacity);

this.loadFactorThreshold = loadFactorThreshold;

table = new LinkedList[capacity];

}

@Override /\*\* Remove all of the entries from this map \*/

public void clear() {

size = 0;

removeEntries();

}

@Override /\*\* Return true if the specified key is in the map \*/

public boolean containsKey(K key) {

if (get(key) != null)

return true;

else

return false;

}

@Override /\*\* Return true if this map contains the value \*/

public boolean containsValue(V value) {

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

LinkedList<Entry<K, V>> bucket = table[i];

for (Entry<K, V> entry: bucket)

if (entry.getValue().equals(value))

return true;

}

}

return false;

}

@Override /\*\* Return a set of entries in the map \*/

public java.util.Set<MyMap.Entry<K,V>> entrySet() {

java.util.Set<MyMap.Entry<K, V>> set =

new java.util.HashSet<>();

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

LinkedList<Entry<K, V>> bucket = table[i];

for (Entry<K, V> entry: bucket)

set.add(entry);

}

}

return set;

}

@Override /\*\* Return the value that matches the specified key \*/

public V get(K key) {

int bucketIndex = hash(key.hashCode());

if (table[bucketIndex] != null) {

LinkedList<Entry<K, V>> bucket = table[bucketIndex];

for (Entry<K, V> entry: bucket)

if (entry.getKey().equals(key))

return entry.getValue();

}

return null;

}

@Override /\*\* Return true if this map contains no entries \*/

public boolean isEmpty() {

return size == 0;

}

@Override /\*\* Return a set consisting of the keys in this map \*/

public java.util.Set<K> keySet() {

java.util.Set<K> set = new java.util.HashSet<K>();

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

LinkedList<Entry<K, V>> bucket = table[i];

for (Entry<K, V> entry: bucket)

set.add(entry.getKey());

}

}

return set;

}

@Override /\*\* Add an entry (key, value) into the map \*/

public V put(K key, V value) {

if (get(key) != null) { // The key is already in the map

int bucketIndex = hash(key.hashCode());

LinkedList<Entry<K, V>> bucket = table[bucketIndex];

for (Entry<K, V> entry: bucket)

if (entry.getKey().equals(key)) {

V oldValue = entry.getValue();

// Replace old value with new value

entry.value = value;

// Return the old value for the key

return oldValue;

}

}

// Check load factor

if (size >= capacity \* loadFactorThreshold) {

if (capacity == MAXIMUM\_CAPACITY)

throw new RuntimeException("Exceeding maximum capacity");

rehash();

}

int bucketIndex = hash(key.hashCode());

// Create a linked list for the bucket if it is not created

if (table[bucketIndex] == null) {

table[bucketIndex] = new LinkedList<Entry<K, V>>();

}

// Add a new entry (key, value) to hashTable[index]

table[bucketIndex].add(new MyMap.Entry<K, V>(key, value));

size++; // Increase size

return value;

}

@Override /\*\* Remove the entries for the specified key \*/

public void remove(K key) {

int bucketIndex = hash(key.hashCode());

// Remove the first entry that matches the key from a bucket

if (table[bucketIndex] != null) {

LinkedList<Entry<K, V>> bucket = table[bucketIndex];

for (Entry<K, V> entry: bucket)

if (entry.getKey().equals(key)) {

bucket.remove(entry);

size--; // Decrease size

break; // Remove just one entry that matches the key

}

}

}

@Override /\*\* Return the number of entries in this map \*/

public int size() {

return size;

}

@Override /\*\* Return a set consisting of the values in this map \*/

public java.util.Set<V> values() {

java.util.Set<V> set = new java.util.HashSet<>();

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

LinkedList<Entry<K, V>> bucket = table[i];

for (Entry<K, V> entry: bucket)

set.add(entry.getValue());

}

}

return set;

}

/\*\* Hash function \*/

private int hash(int hashCode) {

return supplementalHash(hashCode) & (capacity - 1);

}

/\*\* Ensure the hashing is evenly distributed \*/

private static int supplementalHash(int h) {

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

/\*\* Return a power of 2 for initialCapacity \*/

private int trimToPowerOf2(int initialCapacity) {

int capacity = 1;

while (capacity < initialCapacity) {

capacity <<= 1;

}

return capacity;

}

/\*\* Remove all entries from each bucket \*/

private void removeEntries() {

for (int i = 0; i < capacity; i++) {

if (table[i] != null) {

table[i].clear();

}

}

}

/\*\* Rehash the map \*/

private void rehash() {

java.util.Set<Entry<K, V>> set = entrySet(); // Get entries

capacity <<= 1; // Double capacity

table = new LinkedList[capacity]; // Create a new hash table

size = 0; // Reset size to 0

for (Entry<K, V> entry: set) {

put(entry.getKey(), entry.getValue()); // Store to new table

}

}

@Override

public String toString() {

StringBuilder builder = new StringBuilder("[");

for (int i = 0; i < capacity; i++) {

if (table[i] != null && table[i].size() > 0)

for (Entry<K, V> entry: table[i])

builder.append(entry);

}

builder.append("]");

return builder.toString();

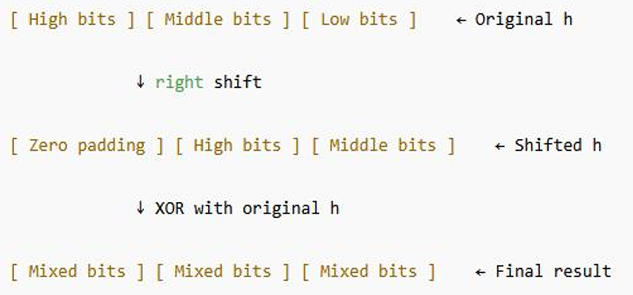
}

}

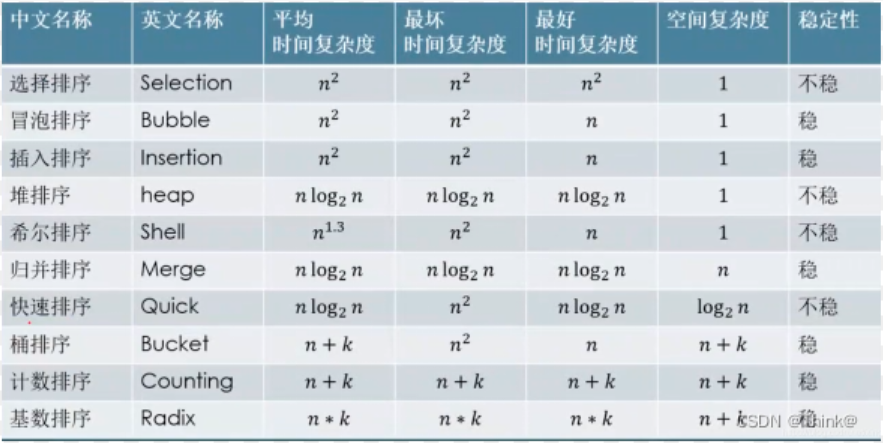
这里有一个补充哈希的操作，这是一种减少哈希冲突的技术，它通过将高位的哈希码移动到低位并与原始哈希码进行XOR（异或）操作，从而使得哈希码分布更加均匀。

这样也可以使得哈希码在哈希表中的分布更加均匀，减少聚类现象。

这个技术在代码中的方法是supplementalHash(int h)。



！！！！！！！！！！！！！！！！ FINAL !!!!!!!!!!!!!!!!!!!



2023-2024 F

1. 填空题：变量类型

题目：

A \_\_\_\_\_ variable stores the actual values of primitive data types directly, while a \_\_\_\_\_ variable stores the address of an object in memory.

中文翻译：

A类型的变量直接存储基本数据类型的实际值，而B类型的变量则存储对象在内存中的地址。

答案：

A: primitive（基本类型）

B: reference（引用类型）

完整句子：

A primitive variable stores the actual values of primitive data types directly, while a reference variable stores the address of an object in memory.

中文解释：

基本类型变量直接存储值，比如 int, float，而引用类型变量（如 String, Object）存储的是内存地址（引用）。

2. 填空题：Java特性

题目：

2. \_\_\_\_\_\_\_\_\_\_ is a feature in Java where a subclass provides a specific implementation of a method that is already provided by its parent class, whereas \_\_\_\_\_\_\_\_\_\_ is a feature that allows a class to have more than one method having the same name, but with different parameter lists.

中文翻译：

\_\_\_\_\_\_\_\_\_\_ 是 Java 中一个特性，子类提供了父类已有方法的特定实现；而 \_\_\_\_\_\_\_\_\_\_ 是一个允许类中存在多个方法同名但参数不同的特性。

答案：

第一个空：Overriding（重写）

第二个空：Overloading（重载）

完整句子：

Overriding is a feature in Java where a subclass provides a specific implementation of a method that is already provided by its parent class, whereas Overloading is a feature that allows a class to have more than one method having the same name, but with different parameter lists.

中文解释：

重写（Overriding）：子类覆盖父类的方法，方法名、参数、返回类型都相同。

重载（Overloading）：方法名相同，但参数数量或类型不同。

3. 代码输出题

题目：

public class Test {

public static void main(String[] args) {

Object o1 = new Object();

Object o2 = new Object();

System.out.print((o1 == o2) + " and " + (o1.equals(o2)));

}

}

问题： 输出结果分别是什么？（填空）

答案：

第一个空：false

第二个空：false

完整输出：

false and false

中文解释：

o1 == o2 比较的是两个对象的引用地址，它们是两个不同的对象，所以是 false。

o1.equals(o2) 在没有重写 equals 方法的情况下，Object 的默认实现和 == 等价，也是比较地址，因此也是 false。



第 4 题：接口与抽象类

题目：

4. An \_\_\_\_\_ is a class-like construct that contains only constants, abstract methods, default methods, and static methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain \_\_\_\_\_.

中文翻译：

\_\_\_\_\_ 是一种类似类的结构，它只包含常量、抽象方法、默认方法和静态方法。在许多方面，接口类似于抽象类，但抽象类还可以包含 \_\_\_\_\_。

答案：

第一个空：interface

第二个空：instance variables and constructors

完整句子：

An interface is a class-like construct that contains only constants, abstract methods, default methods, and static methods. In many ways, an interface is similar to an abstract class, but an abstract class can contain instance variables and constructors.

中文解释：

接口（interface）不能有实例变量，也不能有构造方法。

抽象类（abstract class）可以有实例变量（fields）和构造方法（constructors），这使得它更像一个“未完成的类”。

第 5 题：输出结果

题目代码：

public class Main {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<String>();

list.add("NewYork");

ArrayList<String> list1 = (ArrayList<String>)(list.clone());

list.add("Atlanta");

list1.add("Dallas");

System.out.println(list1);

}

}

问题：

输出结果是：\_\_\_\_\_\_\_\_\_\_？

答案：

[NewYork, Dallas]

中文解释：

list 中添加了 "NewYork"。

然后用 clone() 创建了 list1，此时它复制了当前 list 的内容，变成了 ["NewYork"]。

后来 list 添加了 "Atlanta"，但是这不会影响 list1，因为 clone() 创建的是浅复制的新列表（列表对象本身不同，但元素是同一个引用）。

list1 再添加 "Dallas"，所以最终输出是 ["NewYork", "Dallas"]



第 6 题：编译错误原因

题目代码（有行号）：

1. public class Test {

2. public static void main(String[] args) {

3. Fruit[] fruits = { new Fruit(2), new Fruit(3), new Fruit(1) };

4. Arrays.sort(fruits);

5. }

6. }

class Fruit {

private double weight;

public Fruit(double weight) {

this.weight = weight;

}

}

问题：

这个程序会出现编译错误，因为 Fruit 类没有实现 Comparable 接口。请问是在哪一行出的问题？Line \_\_\_\_?

答案：

Line 4

解释：

Arrays.sort(fruits) 试图对 Fruit 类型数组排序。

排序必须依赖于元素的可比较性（Comparable 接口），但 Fruit 类没有实现 Comparable<Fruit> 接口。

所以编译器会报错：Fruit cannot be cast to Comparable。

✅ 如果想修复这个错误：

你需要让 Fruit 类实现 Comparable 接口并定义 compareTo() 方法，例如：

class Fruit implements Comparable<Fruit> {

private double weight;

public Fruit(double weight) {

this.weight = weight;

}

@Override

public int compareTo(Fruit other) {

return Double.compare(this.weight, other.weight);

}

}

第 7 题：泛型类与编译器检查

题目：

7. A \_\_\_\_\_ class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use a class or method with an incompatible object, the \_\_\_\_\_ will detect the error.

中文翻译：

一个 \_\_\_\_\_ 类或方法允许你指定它可以使用的对象类型。如果你尝试使用不兼容的类型，\_\_\_\_\_ 会检测出错误。

✅ 答案：

第一个空：generic

第二个空：compiler

📘 完整句子：

A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use a class or method with an incompatible object, the compiler will detect the error.

🧠 中文解释：

泛型类或方法（Generic）可以在编译时指定类型，避免类型转换错误。

编译器（Compiler）在编译期间检查类型安全，减少运行时错误。

第 8 题：泛型方法声明

题目代码：

public class GenericMethodDemo {

public static void main(String[] args) {

Integer[] integers = {1, 2, 3, 4, 5};

String[] strings = {"London", "Paris", "NewYork", "Austin"};

print(integers);

print(strings);

}

public static \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ {

for (int i = 0; i < list.length; i++)

System.out.print(list[i] + " ");

System.out.println();

}

}

✅ 答案：

<T> void print(T[] list)

📘 完整签名：

public static <T> void print(T[] list)

🧠 中文解释：

<T> 是泛型声明，代表这个方法是一个泛型方法。

T[] list 表示参数是一个元素类型为 T 的数组，T 可以是任何类型。

这样才能让一个方法同时打印 Integer[] 和 String[]。



第 9 题：集合类型性能对比

题目：

9. In Java collections, the \_\_\_\_\_ is efficient for retrieving elements and for adding and removing elements at the end of the list. On the other hand, a \_\_\_\_\_ is better suited for adding and removing elements at any position within the list, although locating specific elements for operations is not as efficient.

中文翻译：

在 Java 集合中，\_\_\_\_\_ 适合快速访问和在列表尾部添加/删除元素。而 \_\_\_\_\_ 更适合在列表中任意位置添加/删除元素，但定位具体元素的效率不高。

✅ 答案：

第一个空：ArrayList

第二个空：LinkedList

🧠 中文解释：

ArrayList：底层是数组，支持快速随机访问，但中间插入/删除比较慢。

LinkedList：底层是链表，插入/删除任意位置都快，但查找慢。

第 10 题：集合操作 remove()

题目条件：

list1: ["red", "yellow", "green"]

list2: ["red", "yellow", "blue"]

执行：list1.remove(list2)

❗ 注意：你这题的写法可能是 list1.removeAll(list2) 才对。因为 list.remove(Object) 只删除单个对象。我会分别解释。

✅ 正常情况 1：如果是 list1.removeAll(list2)：

作用：删除 list1 中所有在 list2 中也出现的元素

删除 "red" 和 "yellow"

list1 剩下 ["green"]

✅ 答案： ["green"]

🚫 异常情况 2：如果真的是 list1.remove(list2)（错误写法）

list2 是一个 List，不是 String，所以 list1.remove(list2) 会尝试删除整个 list2 这个对象，但失败。

list1 不变，仍是 ["red", "yellow", "green"]

🧠 建议答案：

如果你是考 removeAll：

✅ 答案：["green"]

✅ 第 11 题：栈与队列的处理顺序

题目：

11. \_\_\_\_\_ store objects that are processed in a last-in, first-out fashion.

\_\_\_\_\_ store objects that are processed in a first-in, first-out fashion.

中文翻译：

\_\_\_\_\_ 存储的对象遵循 后进先出（LIFO） 的顺序处理。

\_\_\_\_\_ 存储的对象遵循 先进先出（FIFO） 的顺序处理。

✅ 答案：

第一个空：Stack

第二个空：Queue

📘 英文解释：

Stack → Last In, First Out (LIFO)

Queue → First In, First Out (FIFO)

🧠 中文解释：

栈（Stack）：像书堆，最后放进去的最先被取出。

队列（Queue）：像排队买票，先来先走。

✅ 第 12 题：PriorityQueue 的输出顺序

📄 原始代码：

public class Test {

public static void main(String[] args) {

PriorityQueue<Integer> queue = new PriorityQueue<Integer>(

Arrays.asList(20, 40, 60, 10, 50, 30)

);

while (!queue.isEmpty())

System.out.print(queue.poll() + " ");

}

}

✅ 答案输出：

10 20 30 40 50 60

🧠 解释：

PriorityQueue 是一个 小顶堆：最小的元素有最高优先级。

poll() 方法每次取出当前队列中最小的元素。

所以元素按升序输出。

Java 中的 PriorityQueue（对于整数类型默认情况）会按自然升序对元素进行排序。poll() 方法检索并移除此队列的头部（最小的元素）。代码使用数字 20, 40, 60, 10, 50 和 30 初始化 PriorityQueue。while 循环重复地取出并打印最小的元素，直到队列为空。

✅ 第 13 题：LinkedHashSet 的输出顺序

📄 代码：

Set<String> set = new LinkedHashSet<>();

set.add("ABC");

set.add("FGH");

System.out.println(set);

✅ 答案输出：

[ABC, FGH]

🧠 中文解释：

LinkedHashSet 是一个有顺序的 Set，它保留了元素 插入的顺序。

所以虽然是 Set（不重复），但不像 HashSet 那样无序。

你加入的顺序是 "ABC" → "FGH"，所以输出顺序一样。

✅ 第 14 题：创建空的 HashSet

题目：

Complete the following code to create an empty hash set of integers:

Set<Integer> set = new \_\_\_\_\_\_;

✅ 答案：

new HashSet<>();

📘 完整代码：

Set<Integer> set = new HashSet<>();

🧠 中文解释：

Set 是接口，不能直接实例化。

用 HashSet 实现它，创建一个空的整数集合

第 15 题：图书馆系统中最佳数据结构

题目：

In a library management system where you need to frequently check the availability and information of books with unique identifiers (e.g., bookID), the most suitable data structure to store the books in the library would be a \_\_\_\_\_\_\_\_\_\_.

✅ 答案：

HashMap

🧠 中文解释：

图书ID（如 bookID）是唯一的，最适合用 key-value 结构存储。

HashMap<bookID, BookObject> 可以用 O(1) 的时间快速查找书籍信息。

✅ 第 16 题：算法性能分析方法

题目：

The \_\_\_\_\_ is a theoretical approach for analyzing the performance of an algorithm. It estimates how fast an algorithm’s execution time increases as the input size increases...

✅ 答案：

Big O notation

🧠 中文解释：

Big O 表示法（大 O 符号）用于估算算法的增长率。

举例：O(n)、O(log n)、O(n²)

✅ 第 17 题：常见时间复杂度术语

题目：

An algorithm with the \_\_\_\_\_ time complexity is called a linear time algorithm, and an algorithm with the \_\_\_\_\_ time complexity is called a logarithmic algorithm.

✅ 答案：

第一空：O(n)（线性）

第二空：O(log n)（对数）

✅ 第 18 题：方法的时间复杂度分析

题目代码：

public static void mD(int[] m) {

for (int i = 0; i < m.length; i++) {

System.out.print(m[i] + "");

}

}

✅ 答案：

O(n)

🧠 解释：

有一个 for 循环，运行 n 次（数组长度）。

每次执行一次打印语句，所以总操作是 n 次，时间复杂度是 O(n)。

✅ 第 19 题：最坏情况的时间复杂度

题目：

The worst-case time complexity for selection sort, insertion sort, bubble sort, and quicksort algorithms is \_\_\_\_\_.

✅ 答案：

O(n²)

📘 英文解释：

All of Selection Sort, Insertion Sort, and Bubble Sort have a worst-case time complexity of O(n²).

QuickSort's worst case also can be O(n²), especially when the pivot selection is poor (like always picking the smallest/largest element).

🧠 中文解释：

选择、插入、冒泡 排序算法在最坏情况下都要比较接近 n² 次。

快速排序在最坏情况下也会退化为 O(n²)，比如每次分割都很不均衡（例如已经排序好的数据）。

✅ 第 20 题：冒泡排序第一轮结果

题目：

Suppose a list is [12, 9, 15, 4, 20, 11]. After the first pass of bubble sort, the list becomes \_\_\_\_\_.

🧠 冒泡排序第一轮过程（两两比较并交换）：

初始：[12, 9, 15, 4, 20, 11]

比较过程如下：

12 vs 9 → 交换 → [9, 12, 15, 4, 20, 11]

12 vs 15 → 不换

15 vs 4 → 交换 → [9, 12, 4, 15, 20, 11]

15 vs 20 → 不换

20 vs 11 → 交换 → [9, 12, 4, 15, 11, 20]

✅ 答案：

[9, 12, 4, 15, 11, 20]

🧠 中文解释：

冒泡排序的第一轮会把最大值“冒泡”到最后，剩下的还要继续排。

第 21 题：最大堆的可能取值数量

题目大致结构是：

一个 最大堆（Max Heap） 的结构如下图：

13

/ \

? 10

/ \

8 9

我们要确定 ? 的位置可以填哪些值，才能仍然满足 Max Heap 的结构。

🧠 Max Heap 规则：

每个父节点的值必须大于等于它的子节点的值。

当前结构：

根节点是 13

? 是 13 的左孩子，它的子节点是 8 和 9

10 是 13 的右孩子

所以 ? 的值必须：

≤ 13（因为它是 13 的子节点）

≥ 9（因为它是 8 和 9 的父节点）

✅ 所以，? 的合法范围是：

9 ≤ ? ≤ 13

能取的整数值有：9, 10, 11, 12, 13 → 共 5 个值

✅ 答案：

5

✅ 第 22 题：图中多条边 & 完全图定义

题目：

In a graph, if two vertices are connected by two or more edges, these edges are called \_\_\_\_\_\_.

A \_\_\_\_\_\_ is the one in which every two pairs of distinct vertices are connected by an edge.

✅ 答案：

第一空：parallel edges（平行边）

第二空：complete graph（完全图）

🧠 中文解释：

平行边（Parallel edges）：两个顶点之间有多条边。

完全图（Complete graph）：每一对不同的顶点之间都直接连接一条边。

✅ 第 23 题：图的遍历方式

题目：

There are two popular ways to traverse a graph, namely \_\_\_\_\_ and \_\_\_\_\_.

✅ 答案：

Depth-First Search（DFS）

Breadth-First Search（BFS）

🧠 中文解释：

DFS（深度优先搜索）：尽可能“走到底”，用递归或栈实现。

BFS（广度优先搜索）：一层层展开，用队列实现。

✅ 第 24 题：最小生成树 vs 最短路径

题目：

In a network of cities connected by roads of varying lengths,

if you want to connect all the cities in the most cost-effective way possible, use the \_\_\_\_\_\_\_\_\_\_ algorithm.

if you want to find the shortest route from one specific city to another, use the \_\_\_\_\_\_\_\_\_\_ algorithm.

✅ 答案：

最小生成树： Prim’s algorithm（或 Kruskal’s algorithm）

最短路径： Dijkstra’s algorithm

✔️ 所以填法可以是：

Prim’s (或 Kruskal’s), Dijkstra’s

🧠 中文解释：

Prim / Kruskal：用于构造连接所有城市的最小花费网络（MST）

Dijkstra：从一个城市出发找最短路径到其它城市

✅ 第 25 题：平衡二叉搜索树的插入删除复杂度

题目：

The time complexity for inserting and deleting an element into a binary search tree are \_\_\_\_\_ and \_\_\_\_\_ respectively, assuming that the tree is balanced.

✅ 答案：

O(log n), O(log n)

🧠 中文解释：

在 平衡二叉搜索树（如 AVL 或红黑树） 中，每次查找路径长度是对数级别。

所以插入、删除的复杂度都是 O(log n)。

✅ 第 26 题：二叉树的遍历顺序

题目：

In a binary tree, a \_\_\_\_\_ traversal visits the root node first, followed by the left subtree, and then the right subtree.

Conversely, a \_\_\_\_\_ traversal visits the left subtree first, followed by the right subtree, and finally the root node.

✅ 答案：

preorder, postorder

📘 英文解释：

Preorder Traversal (先序遍历)：Root → Left → Right

Postorder Traversal (后序遍历)：Left → Right → Root

🧠 中文解释：

先序遍历：先访问根节点，然后左子树，最后右子树。

后序遍历：先左子树，再右子树，最后访问根节点。

✅ 第 27 题：AVL 树的平衡因子（Balance Factor）

题目：

In an AVL tree, a node is said to be \_\_\_\_\_ if its balance factor is -1, and it is said to be \_\_\_\_\_ if its balance factor is +1.

✅ 答案：

right-heavy, left-heavy

📘 英文解释：

Balance Factor = Height(left subtree) - Height(right subtree)

-1 表示 右子树高于左子树 → Right-heavy

+1 表示 左子树高于右子树 → Left-heavy

🧠 中文解释：

平衡因子 = 左子树高度 - 右子树高度

平衡因子为 -1：右边比较“重” → 叫做 right-heavy（右偏）

平衡因子为 +1：左边比较“重” → 叫做 left-heavy（左偏）

除了**右旋 (Right Rotation)**，AVL 树还有以下几种核心操作，它们主要用于在插入或删除节点后，维持树的平衡性：

**左旋 (Left Rotation)**：

* 1. **何时使用：** 当 AVL 树出现 **右-右 (RR) 不平衡** 时。这意味着某个节点的平衡因子为 -2，且不平衡是由其右子树的右侧插入或删除引起的。
  2. **过程：** 想象一个节点 A 的右子节点 B。左旋操作会使 B 成为新的子树根，A 成为 B 的左子节点。如果 B 原本有左子节点，该节点会成为 A 的右子节点。
  3. **例子：**

A (-2)

\

B (-1或0)

\

C

* 1. 左旋 A 后变为：

B

/ \

A C

**左右旋 (Left-Right Rotation / LR Rotation)**：

* 1. **何时使用：** 当 AVL 树出现 **左-右 (LR) 不平衡** 时。这意味着某个节点的平衡因子为 +2，且不平衡是由其左子树的右侧插入或删除引起的。
  2. **过程：** 这是一种**组合旋转**。首先对不平衡节点的**左子节点**执行**左旋**，这会将其转化为一个 LL 不平衡的情况。然后，对原始不平衡节点执行**右旋**。
  3. **例子：**

A (+2)

/

B (-1)

\

C

* + 1. 对 B 进行左旋：

A (+2)

/

C

\

B

* + 1. 对 A 进行右旋：

C

/ \

B A

**右左旋 (Right-Left Rotation / RL Rotation)**：

* 1. **何时使用：** 当 AVL 树出现 **右-左 (RL) 不平衡** 时。这意味着某个节点的平衡因子为 -2，且不平衡是由其右子树的左侧插入或删除引起的。
  2. **过程：** 这也是一种**组合旋转**。首先对不平衡节点的**右子节点**执行**右旋**，这会将其转化为一个 RR 不平衡的情况。然后，对原始不平衡节点执行**左旋**。
  3. **例子：**

A (-2)

\

B (+1)

/

C

* + 1. 对 B 进行右旋：

A (-2)

\

C

/

B

* + 1. 对 A 进行左旋：

C

/ \

A B

除了这些核心的旋转操作，AVL 树还涉及：

* **插入 (Insertion):** 将新节点插入到二叉搜索树的正确位置，然后从插入点向上回溯，检查并执行必要的旋转以保持平衡。
* **删除 (Deletion):** 从二叉搜索树中删除一个节点（可能涉及将节点替换为其中序前驱或后继），然后从删除点向上回溯，检查并执行必要的旋转以保持平衡。
* **搜索 (Search):** 这是所有二叉搜索树的基本操作，用于查找树中是否存在某个特定值。它不涉及平衡操作，时间复杂度为 O(log n)。

这些操作使得 AVL 树在各种场景下都能保持高效的 O(log n) 查找、插入和删除时间复杂度。

✅ 第 29 题：处理哈希冲突的方法

题目：

\_\_\_\_\_ and \_\_\_\_\_ methods are usually taken to deal with hashing collision where two different keys are mapped to the same index in a hash table.

✅ 答案：

chaining, open addressing

🧠 中文解释：

Chaining（链地址法）：

把同一个哈希位置上的多个元素用链表存起来。

每个桶可以有多个元素，发生冲突时加入链表。

Open Addressing（开放地址法）：

冲突时，在哈希表中寻找下一个空位插入，而不是使用链表。

这包括线性探测（linear probing）、二次探测（quadratic probing）等方式。

✅ 第 30 题：开放地址法中的具体探测方式

题目：

In hash tables, \_\_\_\_\_ probing resolves collisions by sequentially checking the next available slots,

while \_\_\_\_\_ probing resolves collisions by checking slots at increasing squared distances from the original hash.

✅ 答案：

linear, quadratic

🧠 中文解释：

Linear probing（线性探测）：

发生冲突后，往下一个格子 +1、+2、+3... 逐个试。

优点：实现简单；缺点：容易发生聚集（clustering）。

Quadratic probing（二次探测）：

探测间隔是平方数：+1², +2², +3²...

可以减少聚集，分布更平均。

23-24 R

第 1 题：变量类型（Variable Types）

题目：

In the following code, ‘number’ is a \_\_\_\_\_ variable as it stores the actual values of primitive data types directly, while ‘message’ is a \_\_\_\_\_ variable as it stores the address of an object in memory.

✅ 答案：

primitive, reference

📘 英文解释：

int number = 42; → number 是基本类型（primitive type），直接存储值。

String message = "Hello"; → message 是引用类型（reference type），存储对象地址。

🧠 中文解释：

number 是基本数据类型变量，直接存储值 → primitive

message 是引用类型变量，存储对象在内存中的地址 → reference

✅ 第 2 题：继承类的概念（Superclass & Subclass）

题目：

A \_\_\_\_\_\_ is a class from which other classes are derived, while a \_\_\_\_\_\_ is a class that is derived from another class.

✅ 答案：

superclass, subclass

📘 英文解释：

A superclass provides base functionality.

A subclass inherits and extends the superclass.

🧠 中文解释：

父类（superclass） 是其他类派生的基础类

子类（subclass） 是继承了父类的类

✅ 第 3 题：Java 中的相等性（Equality in Java）

题目：

In Java, \_\_\_\_\_\_\_\_\_\_ equality checks if two variables refer to the exact same object in memory, while \_\_\_\_\_\_\_\_\_\_ equality checks if two objects have the same state or content, as defined by the equals() method.

✅ 答案：

reference, logical

📘 英文解释：

Reference equality uses == to compare memory addresses.

Logical equality uses .equals() to compare values or content.

🧠 中文解释：

引用相等（reference equality） 用 == 比较内存地址是否相同

逻辑相等（logical equality） 用 .equals() 方法判断内容是否相等

✅ 第 4 题：接口与抽象类（Interface vs Abstract Class）

题目：

An \_\_\_\_\_\_ can have instance methods that implement a default behavior, while an \_\_\_\_\_\_ cannot have instance methods.

✅ 答案：

abstract class, interface

📘 英文解释：

Abstract classes can have method implementations.

Interfaces (prior to Java 8) cannot have any method body.

🧠 中文解释：

抽象类（abstract class） 可以包含已实现的方法

接口（interface）（Java 8 前）不能包含方法实现（Java 8 后可以有 default 方法）

✅ 第 5 题：ArrayList vs LinkedList

题目：

For scenarios requiring efficient random access through an index without frequent insertion or removal of elements except at the end of the list, the \_\_\_\_\_\_\_\_\_\_ class is more suitable. In contrast, for applications that frequently insert or delete elements at the beginning of the list, the \_\_\_\_\_\_\_\_\_\_ class is a better choice.

✅ 答案：

ArrayList, LinkedList

📘 英文解释：

ArrayList supports fast indexed access.

LinkedList is efficient for frequent insertions/deletions at the start.

🧠 中文解释：

如果你需要通过索引快速访问 → 使用 ArrayList

如果你经常在列表头部插入/删除元素 → 使用 LinkedList

第 6 题：接口实现错误（Interface Implementation Error）

题目：

In the program below, a compilation error occurs at line 4 because the Fruit class does not implement the \_\_\_\_\_\_\_\_\_\_ interface, which is required for the Arrays.sort() method to sort the array of Fruit objects.

public class Test {

public static void main(String[] args) {

Fruit[] fruits = {new Fruit(2), new Fruit(3), new Fruit(1)};

Arrays.sort(fruits); // Line 4

}

}

class Fruit {

private double weight;

public Fruit(double weight) {

this.weight = weight;

}

}

✅ 答案：

Comparable, Comparable

📘 英文解释：

The Arrays.sort() method requires the objects to implement the Comparable interface to define how they should be compared for sorting.

🧠 中文解释：

Arrays.sort() 方法要求对象实现 Comparable 接口，以便定义对象如何进行排序。

✅ 第 7 题：泛型错误（Generic Type Error）

题目：

The following statement would encounter a compilation error because a generic type must be a \_\_\_\_\_ type. One way to correct the statement is to change the type parameter “int” to \_\_\_\_\_.

ArrayList<int> intList = new ArrayList<>();

✅ 答案：

reference, Integer

📘 英文解释：

In Java, generics can only work with reference types, not primitive types. So, you should use Integer instead of int.

🧠 中文解释：

在 Java 中，泛型只能使用引用类型，不能使用基本类型。因此，应将 int 改为 Integer。

✅ 第 8 题：泛型方法（Generic Method）

题目：

The method header is left blank in the following code. Fill in the header using a generic method \_\_\_\_\_.

public class GenericMethodDemo {

public static void main(String[] args) {

Integer[] numbers = {10, 20, 30, 40, 50};

String[] cities = {"Tokyo", "London", "Paris", "New York"};

print(numbers);

print(cities);

}

public static \_\_\_\_\_\_\_\_ print(Object[] list) {

for (int i = 0; i < list.length; i++)

System.out.print(list[i] + " ");

System.out.println();

}

}

✅ 答案：

<T> void

📘 英文解释：

A generic method allows you to specify the type parameter <T>, which will be determined when the method is called. The void indicates that the method doesn't return anything.

🧠 中文解释：

泛型方法允许你使用 <T> 来指定类型参数，这个类型参数在方法调用时确定。void 表示该方法不返回任何值。

✅ 第 9 题：LinkedList 类型（LinkedList Type）

题目：

In the code below, ‘LinkedList’ is a \_\_\_\_\_\_ without specifying any type parameter, which implies it can hold elements of any type.

public class Example {

public static void main(String[] args) {

LinkedList list = new LinkedList();

list.add("World");

list.add(24);

list.add(2.71);

}

}

✅ 答案：

raw type

📘 英文解释：

A raw type refers to the use of a generic class or interface without specifying a type parameter. This can lead to unchecked warnings or errors.

🧠 中文解释：

原始类型（raw type） 是指使用泛型类或接口时没有指定类型参数，这可能会导致未经检查的警告或错误。

✅ 第 10 题：List 的特点

题目：

A list collection organizes elements in a \_\_\_\_\_ order, and permits specifying the position where an element is stored. Additionally, elements can be accessed by \_\_\_\_\_.

✅ 答案：

linear, index

📘 英文解释：

A list is a linear data structure that maintains the order of elements.

You can access elements by their index.

🧠 中文解释：

List 是一种线性结构，元素按插入顺序排列

元素可以通过索引访问

✅ 第 11 题：比较接口

题目：

The \_\_\_\_\_ interface defines the natural ordering of objects of each class that implements it, whereas the \_\_\_\_\_ interface defines a separate class to compare objects of another class.

✅ 答案：

Comparable, Comparator

📘 英文解释：

Comparable is implemented by the class itself to define natural ordering.

Comparator is a separate class used to define custom ordering.

🧠 中文解释：

Comparable 接口由类自身实现，用于定义自然排序

Comparator 是一个外部比较器类，用于自定义比较规则

✅ 第 12 题：Queue 的方法

题目：

The \_\_\_\_\_\_\_\_\_\_ method in the Queue interface retrieves and removes the head of this queue, or null if this queue is empty. The \_\_\_\_\_\_\_\_\_\_ method retrieves and removes the head and throws an exception if the queue is empty.

✅ 答案：

poll, remove

📘 英文解释：

poll() returns null if the queue is empty.

remove() throws NoSuchElementException if the queue is empty.

🧠 中文解释：

poll()：获取并删除队首元素，如果队列为空则返回 null

remove()：获取并删除队首元素，如果队列为空则抛出异常

✅ 第 13 题：addAll 方法效果

题目：

Suppose list1 is [1, 3, 5] and list2 is [2, 4, 6]. After list1.addAll(list2), list1 is \_\_\_\_\_.

✅ 答案：

[1, 3, 5, 2, 4, 6]

📘 英文解释：

addAll() appends all elements from one list to another.

🧠 中文解释：

addAll() 会将一个列表中的所有元素追加到另一个列表末尾 → 合并成 [1, 3, 5, 2, 4, 6]

✅ 第 14 题：Map 中的 key-value 替换

题目：

In the following code, the key "101" will correspond to the value “\_\_\_\_\_\_” after all the operations.

Map<String, String> map = new HashMap<>();

map.put("101", "Alice");

map.put("102", "Bob");

map.put("101", "Charlie");

✅ 答案：

Charlie

📘 英文解释：

Keys in a map must be unique; later values overwrite earlier ones.

🧠 中文解释：

Map 中键必须唯一，重复插入相同 key 会覆盖旧值 → "101" 对应 "Charlie"

✅ 第 15 题：获取 HashMap 中的值

题目：

The output of the following code is \_\_\_\_\_.

HashMap<String, Integer> map = new HashMap<>();

map.put("One", 1);

map.put("Two", 2);

map.put("Three", 3);

System.out.println(map.get("Two"));

✅ 答案：

2

📘 英文解释：

map.get("Two") returns the value associated with the key "Two".

🧠 中文解释：

通过 key 获取对应的值 → "Two" 对应的 value 是 2

✅ 第 16 题：选择排序（Selection Sort）

题目：

In the selection sort algorithm, the \_\_\_\_\_ element in the array is identified and exchanged with the \_\_\_\_\_ element in each iteration.

✅ 答案：

smallest, first unsorted

📘 英文解释：

Selection sort finds the smallest element from the unsorted part and swaps it with the first unsorted element.

🧠 中文解释：

每次选择最小元素，与未排序部分的第一个元素交换位置

✅ 第 17 题：算法复杂度分类

题目：

An algorithm with the

O(n) time complexity is called a \_\_\_\_\_ algorithm, and an algorithm with the

O(logn) time complexity is called a \_\_\_\_\_\_ algorithm.

✅ 答案：

linear, logarithmic

📘 英文解释：

O(n): The algorithm's runtime grows linearly with the input size → Linear algorithm.

O(logn): The algorithm's runtime grows logarithmically with the input size → Logarithmic algorithm.

🧠 中文解释：

线性算法（Linear Algorithm）：运行时间与输入规模成正比。

对数算法（Logarithmic Algorithm）：运行时间与输入规模的对数成正比。

✅ 第 18 题：解决问题的方法

题目：

\_\_\_\_\_\_ is a method for solving complex problems by breaking them down into simpler subproblems where these subproblems overlap, unlike in \_\_\_\_\_\_ method where subproblems are more independent.

✅ 答案：

Dynamic Programming, Divide and Conquer

📘 英文解释：

Dynamic Programming: Overlapping subproblems are solved optimally using memoization or tabulation.

Divide and Conquer: Subproblems are independent, solved separately, and their solutions are combined.

🧠 中文解释：

动态规划（Dynamic Programming）：将问题分解为重叠的子问题，使用记忆化或表格法优化解决。

分治法（Divide and Conquer）：将问题分解为互相独立的子问题，各自解决后再合并。

✅ 第 19 题：时间复杂度分析

题目：

Use the Big-O notation to estimate the time complexity of the following method:

public static void mD(int[] m) {

for (int i = 0; i < m.length; i++) {

for (int j = 0; j < m.length; j++) {

System.out.print(m[i] + "");

}

}

}

✅ 答案：

O(n^2)

📘 英文解释：

There are two nested loops, each iterating over the array of size n. Thus, the total number of iterations is

n×n, leading to O(n^ 2) time complexity.

🧠 中文解释：

两个嵌套循环，每个循环遍历数组 n 次，总共运行 n×n 次，因此时间复杂度为 O(n 2)。

✅ 第 20 题：冒泡排序第二轮结果

题目：

Suppose a list is [12, 9, 15, 4]. After the second pass of bubble sort, the list becomes \_\_\_\_\_.

✅ 答案：

[9, 12, 4, 15]

📘 英文解释：

First pass: The largest element 15 is moved to the end → [9, 12, 4, 15].

Second pass: The second-largest element 12 is moved before 15 → [9, 12, 4, 15]. The smaller elements start shifting toward the beginning.

🧠 中文解释：

第一轮：最大值 15 移到末尾 → [9, 12, 4, 15]。

第二轮：次大值 12 移动到正确位置 → [9, 12, 4, 15]，较小的元素逐渐向前移动。

✅ 第 22 题：完全图与不完全图

📌 题目：

A \_\_\_\_\_ graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge, while in an \_\_\_\_\_ graph, not every pair of vertices is connected by an edge.

✅ 答案：

complete, incomplete

📘 英文解释：

A complete graph has all possible edges between vertices.

An incomplete graph is missing at least one edge between a pair of vertices.

🧠 中文解释：

完全图：任意两个不同顶点之间都存在一条边。

不完全图：存在至少一对顶点之间没有边。

✅ 第 23 题：DFS 和 BFS

📌 题目：

\_\_\_\_\_ is a graph traversal method that visits all the vertices of a graph as deeply as possible before backtracking, while \_\_\_\_\_ visits all the vertices of a graph level by level.

✅ 答案：

Depth-First Search (DFS), Breadth-First Search (BFS)

📘 英文解释：

DFS explores as far as possible along each branch before backtracking.

BFS explores all neighbors at the current depth before moving deeper.

🧠 中文解释：

DFS（深度优先搜索）：尽可能深入遍历每个分支，再回溯。

BFS（广度优先搜索）：先访问当前层的所有邻居，再进入下一层。

✅ 第 24 题：Prim 与 Dijkstra

📌 题目：

\_\_\_\_\_ is an algorithm in graph theory where the goal is to connect all vertices with the least total weight, while \_\_\_\_\_ is an algorithm where the goal is to find the path with the least total weight between two vertices.

✅ 答案：

Prim's, Dijkstra's

📘 英文解释：

Prim's algorithm builds a Minimum Spanning Tree (MST).

Dijkstra's algorithm finds the shortest path from one vertex to another.

🧠 中文解释：

Prim算法：用于构建最小生成树，连接所有顶点使总权值最小。

Dijkstra算法：用于找出两个顶点之间最短路径。

✅ 第 25 题：前序与中序遍历

📌 题目：

In a binary tree, the \_\_\_\_\_ traversal visits the root node before its children, while the \_\_\_\_\_ traversal visits the left subtree, then the root, and finally the right subtree.

✅ 答案：

preorder, inorder

📘 英文解释：

Preorder: Root → Left → Right

Inorder: Left → Root → Right

🧠 中文解释：

前序遍历：先访问根节点，再访问左右子树。

中序遍历：先访问左子树，再访问根节点，最后访问右子树。

问题 26：树的遍历 (Tree Traversal)

给定以下树的后序遍历和广度优先遍历分别是 \_\_\_\_\_\_ 和 \_\_\_\_\_\_。

给定的树结构：

37

/ \

29 73

/ \ / \

17 31 59 97

1. 后序遍历 (Postorder Traversal)

定义： 后序遍历的顺序是：访问左子树 -> 访问右子树 -> 访问根节点。

我们递归地应用这个规则：

节点 37 (根节点):

访问左子树 (根节点 29):

节点 29:

访问左子树 (根节点 17):

节点 17: 这是一个叶子节点。没有左子节点，没有右子节点，然后访问 17。

输出：17

访问右子树 (根节点 31):

节点 31: 这是一个叶子节点。没有左子节点，没有右子节点，然后访问 31。

输出：31

然后访问 29。

输出：29

访问右子树 (根节点 73):

节点 73:

访问左子树 (根节点 59):

节点 59: 这是一个叶子节点。没有左子节点，没有右子节点，然后访问 59。

输出：59

访问右子树 (根节点 97):

节点 97: 这是一个叶子节点。没有左子节点，没有右子节点，然后访问 97。

输出：97

然后访问 73。

输出：73

然后访问 37。

输出：37

按顺序组合所有输出： 17, 31, 29, 59, 97, 73, 37

2. 广度优先遍历 (Breadth-First Traversal, BFS) / 层次遍历 (Level Order Traversal)

定义： 广度优先遍历是按层级访问节点，每层从左到右。我们通常使用队列来实现广度优先遍历。

第 0 层： 从根节点开始。

队列：[37]

输出：37

37 出队。将 37 的子节点（29, 73）入队。

队列：[29, 73]

第 1 层：

29 出队。将 29 的子节点（17, 31）入队。

输出：29

队列：[73, 17, 31]

73 出队。将 73 的子节点（59, 97）入队。

输出：73

队列：[17, 31, 59, 97]

第 2 层：

17 出队。没有子节点。

输出：17

队列：[31, 59, 97]

31 出队。没有子节点。

输出：31

队列：[59, 97]

59 出队。没有子节点。

输出：59

队列：[97]

97 出队。没有子节点。

输出：97

队列：[] (空了)

按顺序组合所有输出： 37, 29, 73, 17, 31, 59, 97

最终答案 (Final Answer)

给定树的后序遍历是 17, 31, 29, 59, 97, 73, 37，广度优先遍历是 37, 29, 73, 17, 31, 59, 97。

✅ 第 27 题：AVL 树的失衡类型

📌 题目：

In an AVL tree, a \_\_\_\_\_ imbalance occurs when a node is inserted into the right subtree of the right child of A, while a \_\_\_\_\_ imbalance occurs when a node is inserted into the left subtree of the left child of A.

✅ 答案：

right-right (RR), left-left (LL)

📘 英文解释：

RR imbalance happens when insertion occurs in the right subtree of the right child — resolved by a left rotation.

LL imbalance happens when insertion occurs in the left subtree of the left child — resolved by a right rotation.

🧠 中文解释：

右右失衡（RR）：当新节点插入某节点的右子节点的右子树中，需做一次左旋。

左左失衡（LL）：当新节点插入某节点的左子节点的左子树中，需做一次右旋。

✅ 第 29 题：哈希冲突与解决方案

📌 题目：

Collisions occur when two different keys produce the same \_\_\_\_\_. Besides open addressing, \_\_\_\_\_, where each hash table slot maintains a linked list of elements mapping to the same slot, is the other way to solve collisions.

✅ 答案：

hash value, chaining

📘 英文解释：

A collision happens when multiple keys hash to the same index.

Chaining solves this by storing all colliding keys in a linked list at that index.

🧠 中文解释：

当不同的键计算出相同的哈希值时，称为哈希冲突。

除了开放地址法，\*\*链地址法（chaining）\*\*是另一种解决方式：每个槽位存储一个链表，链表中保存哈希到该位置的所有元素。

✅ 第 30 题：双重散列（Double Hashing）

📌 题目：

\_\_\_\_\_ hashing uses a secondary hash function h′ on the keys to determine the increments, aiming to avoid the \_\_\_\_\_ problem.

✅ 答案：

Double, clustering

📘 英文解释：

Double hashing applies a second hash function to calculate the step size.

This avoids clustering, where multiple keys accumulate in adjacent slots.

🧠 中文解释：

双重散列法使用两个哈希函数，通过第二个函数决定探测步长，避免集中冲突。

\*\*聚集问题（clustering）\*\*是指冲突元素集中在某区域，导致查找效率降低。

总结：

Java知识结构图

├── 基础语法与数据类型

│ ├── 原始数据类型 & 引用类型

│ ├── 变量类型（基本类型变量 vs 引用类型变量）

│ ├── 方法重载与重写

├── 面向对象编程

│ ├── 类与对象

│ ├── 继承与多态

│ │ ├── extends、super关键字

│ │ ├── 重写（Override）

│ │ └── 方法重载（Overload）

│ ├── 抽象类 vs 接口

│ │ ├── 抽象方法与默认方法

│ │ ├── 接口的特性（只是声明，Java 8后有default和static方法）

│ └── 封装与访问修饰符

├── 泛型（Generics）

│ ├── 泛型类与泛型方法

│ ├── 类型擦除与通配符

│ └── 常见使用错误（如：ArrayList<int>非法，应使用包装类型）

├── 集合框架（Collections）

│ ├── List接口

│ │ ├── ArrayList（基于动态数组，随机访问快，尾部增删快）

│ │ └── LinkedList（链表结构，任意位置增删高效）

│ ├── Set接口

│ │ ├── HashSet（无序，基于哈希表）

│ │ └── LinkedHashSet（有序，维护插入顺序）

│ ├── Map接口

│ │ └── HashMap（键值对映射，键唯一）

│ ├── Queue接口

│ │ ├── PriorityQueue（优先队列，堆实现）

│ │ └── 常用方法：poll(), remove()

│ └── 迭代器和for-each循环

├── 哈希和冲突解决

│ ├── 哈希函数与哈希码

│ ├── 开放地址法（线性探测，平方探测）

│ └── 链地址法（链表冲突解决）

├── 算法复杂度与分析

│ ├── Big-O表示法（时间复杂度）

│ ├── 常见复杂度：O(1), O(log n), O(n), O(n log n), O(n^2)等

│ ├── 典型排序算法

│ │ ├── 选择排序、冒泡排序、插入排序（O(n^2)）

│ │ └── 快速排序（平均O(n log n)，最坏O(n^2)）

│ └── 动态规划与分治法区别

├── 数据结构

│ ├── 栈（LIFO）与队列（FIFO）

│ ├── 树结构

│ │ ├── 二叉树遍历（前序、中序、后序与层序）

│ │ ├── 二叉搜索树

│ │ └── 平衡树（AVL树，自平衡，旋转操作）

│ └── 图结构

│ ├── 有向无向图

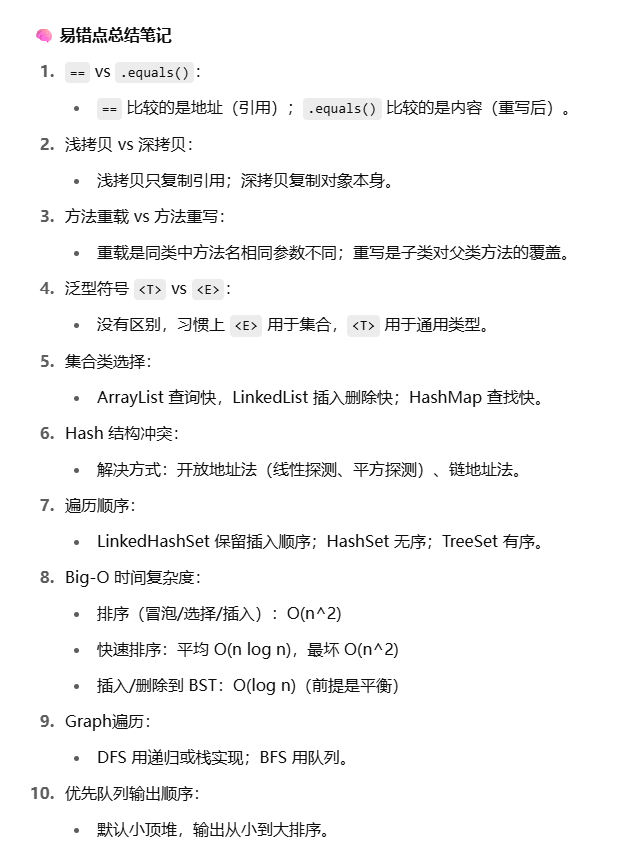
│ ├── 简单图与多重图

│ ├── 深度优先搜索（DFS）和广度优先搜索（BFS）

│ ├── 最小生成树算法（Kruskal，Prim）

│ └── 最短路径算法（Dijkstra）





🧠 易错点总结笔记

在您提供的题目中，以下是几个常见的易错点和需要特别注意的地方：

基本类型与引用类型的区别:

基本类型变量直接存储值（如 int number = 42; number 存储 42）。

引用类型变量存储的是对象的内存地址（如 String message = "Hello"; message 存储的是 "Hello" 字符串对象在内存中的地址）。

在判断相等时，== 用于比较基本类型的值，或比较引用类型变量是否指向同一个对象（即内存地址）。

equals() 方法（通常在引用类型中使用）用于比较对象的内容是否相等。对于 Object 类的 equals() 默认行为与 == 相同，但许多类（如 String, ArrayList 等）都重写了 equals() 方法来比较内容。

方法重写 (Overriding) 与 方法重载 (Overloading):

重写 (Overriding)：子类提供父类已有的方法的特定实现（相同的方法签名，不同的实现）。

重载 (Overloading)：同一个类中，有多个同名方法，但它们的参数列表不同。

抽象类 (Abstract Class) 与 接口 (Interface) 的区别:

抽象类可以包含：抽象方法、非抽象方法（带实现）、构造器、成员变量（包括实例变量）。一个类只能继承一个抽象类。

接口只能包含：常量（public static final）、抽象方法（public abstract）、Java 8 后的 default 方法和 static 方法。一个类可以实现多个接口。

关键差异: 抽象类可以有实例方法（有实现的行为），而接口不能有实例变量和实例方法（在 Java 8 之前）。

集合框架类的选择:

ArrayList: 基于数组实现，适合随机访问元素（通过索引），在列表末尾添加/删除高效。但在列表中间插入或删除元素效率较低，因为需要移动大量元素。

LinkedList: 基于链表实现，适合在列表的任何位置频繁插入或删除元素，因为只需要修改指针。但随机访问效率较低，需要从头开始遍历。

HashSet: 基于哈希表实现，不保证元素顺序，不允许重复元素，查找、添加、删除操作平均时间复杂度为 O(1)。

LinkedHashSet: 保持元素的插入顺序。

PriorityQueue: 基于堆实现，元素按照自然顺序或自定义比较器排序，poll() 方法总是移除优先级最高的元素。

HashMap: 存储键值对，通过键的哈希值快速查找值。

泛型 (Generics) 的使用:

类型参数必须是引用类型，不能是基本类型。例如，ArrayList<int> 是错误的，应该用 ArrayList<Integer>。

Comparable 和 Comparator 接口用于定义对象的比较逻辑。Comparable 定义“自然排序”，Comparator 定义“自定义排序”。

排序算法时间复杂度:

O(n^2)：选择排序、插入排序、冒泡排序（最坏情况）。

O(n log n)：快速排序（平均情况），归并排序，堆排序。

O(n)：线性时间算法。

O(log n)：对数时间算法（如二分查找）。

哈希冲突解决:

链式处理 (Chaining)：在每个哈希表槽位维护一个链表，存储所有哈希到该槽位的元素。

开放寻址 (Open Addressing)：

线性探测 (Linear Probing)：顺序查找下一个空槽。

二次探测 (Quadratic Probing)：以平方递增的距离查找空槽。

双重散列 (Double Hashing)：使用第二个哈希函数来决定探测步长。

哈希冲突产生的原因: 不同的键经过哈希函数计算后得到相同的索引。

图遍历算法:

深度优先搜索 (DFS)：尽可能深地探索每个分支。

广度优先搜索 (BFS)：层层遍历。

图算法应用:

最小生成树 (MST)（如 Prim 算法）：连接所有顶点且总权重最小。

最短路径 (Shortest Path)（如 Dijkstra 算法）：找到两点间权重最小的路径。

树遍历 (Binary Tree Traversal):

前序遍历 (Preorder): 根 -> 左 -> 右

中序遍历 (Inorder): 左 -> 根 -> 右 (二叉搜索树中序遍历结果为有序序列)

后序遍历 (Postorder): 左 -> 右 -> 根

广度优先遍历 (BFS): 层序遍历