算法 读书笔记

目录

1.3.3.1 结点记录

1.3.3.9 队列的实现

1.3 总结（1.3.4 综述）

背包、队列和栈

在研究一个新的应用领域时，识别目标并使用数据抽象解决问题的步骤

1.3 答疑

Q1 Java 标准库中有栈和队列吗?

Q2 为什么不实现一个单独的 Collection 数据类型并实现添加元素、删除最近插入的元素、删除最早插入的元素、删除随机元素、迭代、返回集合元素数量和其他我们可能需要的方法？这样我们就能在一个类中实现所有这些方法并可以应用于各种用例。

前言

作为教材

- 这些算法一般都巧妙奇特，20 行左右的代码就足以表达。
- 学过一门计算机方面的先导课程就足矣，只要熟悉一门现代编程语言并熟知现代计算机系统，就都能够阅读本书。
- 本书涉及的内容是任何准备主修计算机科学、……、等专业的学生应了解的基础知识，……

背景介绍

- ……而本书则是专门为大学一、二年级学生设计的一学期教材，也是最新的基础入门书或从业者的参考书。

第一章 基础

- 算法：有限、确定、有效的并适合用计算机实现的解决问题的方法
- 数据结构：便于算法操作的组织数据的方法
- 数据结构是算法的副产品或是结果，简单的算法也会产生复杂的数据结构，复杂的算法也许只需要简单的数据结构。
- 算法分析：为一项任务选择最合适的算法而进行的分析
- 学习算法是非常有趣和令人激动的，因为这是一个历久弥新的领域（我们学习的绝大多数算法都还不到“五十岁”……）

1.1 基础编程模型

- 我们把描述和实现算法所用到的语言特性、软件库和操作系统特性总称为**基础编程模型**

1.1.4 简便记法

- 程序有很多写法，我们追求清晰、优雅和高效的代码

1.1.5.1 创建并初始化数组

- 在代码中使用数组时，一定要依次声明、创建并初始化数组。忽略了其中的任何一步都是很常见的

编程错误。

1.1.6.1 静态方法

- 方法需要**参数**（某种数据类型的值）并根据参数计算出某种数据类型的**返回值**（例如数学函数的结果）或者产生某种**副作用**（例如打印一个值）
- 典型静态方法的实现
  - 判断一个数是否是素数

```
public static boolean isPrime(int N)
{
if (N < 2) return false;
for (int i = 2; i*i <= N; i++)
  if (N % i == 0) return false;
return true;
}
```

  - 计算平方根（牛顿迭代法）

```
public static double sqrt(double c)
{
if (c < 0) return Double.NaN;
double err = 1e-15;
double t = c;
while (Math.abs(t - c/t) > err * t)
  t = (c/t + t) / 2.0;
return t;
}
```

    - 牛顿法（Newton's method） 求 a 的 m 次方根

$$x_{n+1} = x_n - \frac{x_n}{m}(1 - ax_n^{-m})$$

当 $m = 2$ 时，则：

$$\begin{aligned} x_{n+1} &= x_n - \frac{x_n}{2}(1 - ax_n^{-2}) \\ &= \frac{x_n}{2} + \frac{ax_n^{-1}}{2} \\ &= (a/x_n + x_n)/2 \end{aligned}$$

1.1.6.4 递归

- 递归代码比相应的非递归代码更加简洁优雅、易懂
- 编写递归代码时最重要的有以下三点
  - 递归总有一个**最简单的情况**——方法的第一条语句总是一个包含 `return` 的条件语句

    - 递归调用总是尝试解决一个**规模更小**的子问题，这样递归才能收敛到最简单的情况
    - 递归调用的父问题和尝试解决的子问题之间不应该有**交集**
- 坚持这些原则能写出清晰、正确且容易评估性能的程序
- 使用递归的另一个原因：可以使用数学模型来估计程序的性能
- 练习 1.1.27：**二项分布**。估计用以下代码计算 `binomial(100, 50, 0.25)` 将会产生的递归调用次数：

```
public static double binomial(int N, int k, double p)
{
  if (N == 0 && k == 0) return 1.0;
  if (N < 0 || k < 0) return 0.0;
  return (1.0 - p)*binomial(N-1, k, p) + p*binomial(N-1, k-1, p);
}
```

  将已经计算过的值保存在数组中并给出一个更好的实现。

  - 解：将 $binomial(N,k,p)$ 产生的递归调用次数记为 $c(n,k)$，当 $k>0$ 时，则：

$$\begin{aligned} c(n,k) &= 1+c(n-1,k)+c(n-1,k-1) \\ &= 2+c(n-2,k)+c(n-2,k-1)+c(n-1,k-1) \\ &= \dots \\ &= n+c(n-n,k)+c(n-n,k-1)+\dots+c(n-1,k-1) \\ &= n+1+c(-1,k)+c(-1,k-1)+c(0,k-1)+\dots+c(n-1,k-1) \\ &= n+3+\sum_{i=0}^{n-1} c(i,k-1) \end{aligned}$$

    - 二项分布 设 X 为 n 重贝努利试验中成功的次数，则称 X 取值的概率分布为**二项分布**，记为 $b(n,p)$。
      - n 重贝努利试验的特点
        1. 重复进行 n 次相互独立的试验
        2. 每次试验只有成功或失败中的一个结果
        3. 每次出现成功的概率均为 p
    - $binomial(n,k,p)$ 表示事件 “n 重贝努利试验中成功出现 k 次” 的概率，则：对于其中任意一次贝努利试验，成功概率为 p，若其成功，则事件变为 $n-1$ 重贝努利试验中成功出现 $k-1$ 次；若其失败，则事件变为 $n-1$ 重贝努利试验中成功出现 k 次。故：

$$binomial(n,k,p) = p*binomial(n-1,k-1,p)+(1-p)*binomial(n-1,k,p)$$

      又 $n \geqslant k$，则需判断：$k<0$ 时，概率为 0；$n<k$ 时，概率为 0；$n=0,k=0$ 时，概率为 1。

1.1.7 API (Application programming interface)

- In computer programming, an application programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software.

- In general terms, it is a set of clearly defined methods of communication among various components. An API specification can take many forms, but often includes specifications for routines, data structures, object classes, variables, or remote calls.
- A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer. Documentation for the API usually is provided to facilitate usage and implementation.
- API 的目的是将调用和实现**分离**
  - API 是调用和实现之间的一份**契约**，它详细说明了每个方法的作用。实现的目标就是能够遵守这份契约。

1.1.9.5 重定向与管道

- 结合标准输入输出的重定向，将一个程序的输出重定向为另一个程序的输入叫做**管道**：

```
% java RandomSeq 1000 100.0 200.0 | java Average
```

这样做的影响非常深远，因为它突破了能够处理的输入输出流的长度限制。例如：即使计算机没有足够的空间来存储十亿个数，仍然可以将例子中的 1000 换成 1 000 000 000。输入输出流中字符串的增删动作发生的实际顺序取决于操作系统。

1.1 总结

1.1.6.5 基础编程模型

- **静态方法库**是定义在一个 Java 类中的一组静态方法
- Java 开发的基本模式是编写一个静态方法库（包含一个 `main()` 方法）来完成一个任务

1.1.6.6 模块化编程

- 这个模型的最重要之处在于通过静态方法库实现了模块化编程
  - 程序整体的代码量很大时，每次处理的模块大小仍然适中
  - 可以共享和重用代码而无需重新实现
  - 很容易用改进的实现替换老的实现
  - 可以为解决问题建立合适的抽象模型
  - 缩小调试范围

1.1.6.7 单元测试

- 将 `main()` 方法作为一个**开发用例**，也可以把它编成一个**测试用例**
- 用例复杂时，也可将它独立成一个模块

1.1.11 展望

- 本节描述了一个精巧而完整的编程模型，数十年来它一直在（并且现在仍在）为广大程序员服务。但现代编程技术已经更进一步。前进的这一步被称为**数据抽象**，有时也被称为**面向对象编程**。

- 数据抽象的主要思想是鼓励程序定义自己的**数据类型**（一系列值和对这些值的操作），而不仅仅是那些操作预定义的数据类型的静态方法。
  - 数据抽象允许通过模块化编程复用代码
  - 可以轻易构造多种所谓的**链式**数据结构，它们比数组更灵活，在许多情况下都是高效算法的基础
  - 借助数据抽象可以准确地定义所面对的算法问题

1.1 答疑

Q1 难道 Java 不应该自动检查溢出吗?

- 它们之所以被称为**原始**数据类型就是因为缺乏此类检查。

Q2 `Math.abs(-2147483648)` 的返回值是什么?

- 整数溢出，返回值是 `-2147483648`

Q3 负数的除法和余数的结果是什么?

- 表达式 a/b 的商会向 0 取整；a % b 的余数的定义是 $(a/b)*b+a\%b\equiv a$
- ：

$$\begin{aligned} \text{除法的定义：}\ a\div b=x\ldots y &\qquad (1)\\ x=a/b,\ y=a\%b &\qquad (2)\\ \text{由 (1) 得：}\ x*b+y\equiv a &\qquad (3)\\ \text{将 (2) 代入 (3)，得：}\ (a/b)*+a\%b\equiv a & \end{aligned}$$

- -14/3 = 14/-3 = -4; -14 % 3 = -2, 14 % -3 = 2

1.1 练习

1.1.6 下面这段程序会打印出什么?

```
int f = 0;
int g = 1;
for (int i = 0; i <= 15; i++)
{
  StdOuut.println(f);
  f = f + g;
  g = f - g;
}
```

- 解：
  1. 将斐波那契数列的定义：$f(i)+f(i+1)=f(i+2)\ (i\in N,\ f(0)=0,\ f(1)=1)$ 推广到 $i\in Z$，这样一来，可写成 ..., 2, -1, 1, 0, 1, 1, 2, ...
  2. 由 1.：程序中变量 `f` 和 `g` 的初始值可理解为 $f(0)$，$f(-1)$，记为 `f(i)`，`f(i-1) (i = 0)`

3. 迭代过程中 `f = f + g = f(i) + f(i-1) = f(i+1)`， `g = f - g = f(i+1) - f(i-1) = f(i)`
4. 迭代结果： `f = f(i+1), g = f(i)`
5. 故，打印出的是斐波那契数列的前 15 项

1.1.9 实现 `Integer.toBinaryString(N)`

- Solution:

```
String s = "";
for (int n = N; n > 0; n /= 2)
  s = (n % 2) + s
```

1.1.18 请看以下递归函数：

```
public static int mystery(int a, int b)
{
  if (b == 0)     return 0;
  if (b % 2 == 0) return mystery(a+a, b/2);
  return mystery(a+a, b/2) + a;
}
```

- `mystery(a, b)` 的计算结果是 $a * b$； 将代码中的 `+` 替换为 `*`， `0` 替换为 `1`，计算结果为 a^b

1.2 数据抽象

- **数据类型**指的是一组值和一组对这些值的操作的集合
- **对象**：保存了某个数据类型的值的实体
- **抽象数据类型**（ADT）
  - 用 Java 类来实现抽象数据类型 <=> 用一组静态方法实现一个函数库
  - 将数据和函数相关联，并隐藏数据的表示方式
  - 使用 ADT
    - 关注 API 描述的操作，而非数据的表示
  - 实现 ADT
    - 关注数据本身，而非对数据的各种操作
  - 在本书中通过 ADT
    - 以适用于各种用途的 API 形式准确地定义问题
    - 用 API 的实现描述算法和数据结构

1.2.1 使用抽象数据类型

- 要使用一种数据类型并不一定非得知道它是如何实现的

1.2.1.4 对象

- **对象**是能够承载数据类型的值的实体
- 三大重要特性：**状态**、**标识**和**行为**
  - **状态**：数据类型中的值
  - **标识**：将一个对象区别于另一个对象（可以认为是它在内存中的位置）
  - **行为**：数据类型的操作

1.2 答疑

Q1 指针是什么?

- 问得好。……但众所周知，指针的编程非常容易出错，因此需要精心设计指针类的操作以帮助程序员避免错误。Java 将这种观点发挥到了极致（许多主流编程语言的设计者也赞同这种做法）。在 Java 中，创建引用的方法**只有一种**（new），且改变引用的方法也**只有一种**（赋值语句）。

1.3 背包、队列和栈

1.3.2.2 泛型

- 创建泛型数组在 Java 中是不允许的，需要使用强制类型转换：`a = (Item[]) new Object[cap]`

1.3.2.3 调整数组大小

- 首先，实现一个方法将栈移动到另一个大小不同的数组中：

```
private void resize(int max)
{  // 将大小为 N < = max 的栈移动到一个新的大小为 max 的数组中
  Item[] temp = (Item[]) new Object[max];
  for (int i = 0; i < N; i++)
    temp[i] = a[i];
  a = temp;
}
```

- 在 `push()` 中，检查数组是否太小

```
public void push(Item item)
{  // 将元素压入栈顶
   if (N == a.length) resize(2*a.length);
   a[N++] = item;
}
```

- 在 `pop()` 中，首先删除栈顶元素，然后如果数组太大我们就将它的长度**减半**。只要稍加思考，你就明白正确的检测条件是栈大小是否小于**数组的四分之一**。（代码实现中判断的是“是否相等”，原因是：`private Item[] a = (Item[]) new Object[1]` ）

```
public Item pop()
{  // 从栈顶删除元素
   Item item = a[--N];
   a[N] = null;  // 避免对象游离
   if (N > 0 && N == a.length/4) resize(a.length/2);
   return item;
}
```

- 在这个实现中，栈永远不会溢出，使用率也永远不会低于四分之一（除非栈为空，那种情况下数组的大小为 1）

1.3.2.4 对象游离

- Java 的垃圾收集策略是回收所有无法被访问的对象的内存。例如，在 `pop()` 的实现中，被弹出的元素的引用仍然存在于数组中，虽然这个**孤儿**元素永远不会再被访问。保存一个不需要的对象的引用称为**游离**。这里将被弹出的数组元素的值设为 `null` 就能避免对象游离。这将覆盖无用的引用并使系统可以在用例使用完被弹出的元素后回收它的内存。

1.3.2.5 迭代

- 任意集合类数据类型的实现的最佳性能：
  - 每项操作的用时都与集合大小无关
  - 空间需求总是不超过集合大小乘以一个常数

1.3.3 链表

- **定义 链表**是一种递归的数据结构，它或者为空（ `null` ），或者是指向另一个**结点**（node）的引用，该结点含有一个泛型的元素和一个指向另一条链表的引用。

1.3.3.1 结点记录

- 使用**嵌套类**来定义结点的抽象数据类型：

```
private class Node
{
  Item item;
  Node next;
}
```

在需要使用 `Node` 类的类中定义它并将它标记为 `private` ，因为它不是为用例准备的

1.3.3.9 队列的实现

- 在结构化存储数据集时，**链表是数组的一种重要的替代方式**

1.3 总结（1.3.4 综述）

背包、队列和栈

- 深入理解这三种抽象数据类型是研究算法和数据结构的开始
  1. 这些数据类型是构造其他更高级的数据结构的基石
  2. 它们展示了数据结构和算法的关系 以及 同时满足多个有可能相互冲突的性能目标时所要面对的挑战
  3. 将要学习的若干算法的实现重点 就是需要 其中的抽象数据类型 能够 支持对对象集合的强大操作，这些实现正是起点

在研究一个新的应用领域时，识别目标并使用数据抽象解决问题的步骤

- 1. 定义 API
  2. 根据特定的应用场景开发用例代码
  3. 描述一种数据结构（一组值的表示），并在 API 所对应的抽象数据类型的实现中根据它定义类的**实例变量**
  4. 描述算法（实现一组操作的方式），并根据它实现类中的**实例方法**
  5. 分析算法的性能特点
- 最后一步常常能够决定哪种算法和实现才是解决现实应用问题的最佳选择

1.3 答疑

Q1 Java 标准库中有栈和队列吗?

- 有，也没有。Java 有一个内置的库，叫做 java.util.Stack，但你需要栈的时候请不要使用它。它新增了几个一般不属于栈的方法，例如获取第 i 个元素。它还允许从栈底添加元素（而非栈顶），所以它可以被当作队列使用！……java.util.Stack 的 API 是**宽接口**的一个典型例子，我们通常会极力避免出现这种情况。

Q2 为什么不实现一个单独的 Collection 数据类型并实现添加元素、删除最近插入的元素、删除最早插入的元素、删除随机元素、迭代、返回集合元素数量和其他我们可能需要的方法？这样我们就能在一个类中实现所有这些方法并可以应用于各种用例。

- 再强调一遍，这又是一个**宽接口**的例子。Java 在……中实现了类似的设计。避免使用它们的原因是这样无法保证高效实现所有这些方法。
- 坚持窄接口的原因
  1. 在本书中总是以 API 作为设计高效算法和数据结构的起点，而设计只含有几个操作的接口显然比设计含有许多操作的接口更简单
  2. 窄接口能够限制用例的行为，这将使用例更加易懂
     - 如果一段用例代码使用 `Stack<String>` ，而另一段用例代码使用 `Queue<Transaction>` ，我们就可以知道后进先出的访问顺序对于前者很重要，而先进先出的访问顺序对于后者很重要