第1章 绪 论


初识数据结构和算法

掌握数据结构(data structure)和算法(algorithm)是程序员的内功,架构搭得再好,使用的技术再新,如果没有好的数据结构设计和算法,系统也会出问题甚至崩溃,细节决定成败,练好内功非常重要。数据结构和算法是计算机专业的一门基础课程,学习数据结构,对于初级程序员、软件设计师、系统架构设计师和追求细节的开发人员都有必要。

为了方便学习数据结构和算法,推荐一个由旧金山大学计算机科学专业的大卫·加勒提供的学习网站“Data Structure Visualizations”(“数据结构可视化”,网址将在本书的数字资源中提供)。该网站将数据结构和算法用可视化的方式展现出来,方便学习者理解其中的原理。虽然是英文网站,但大多是容易理解的术语,其中涵盖了平时常见的数据结构和算法(见图 1.1)。

Data Structure Visualizations

About
Algorithms
F.A.Q
Known Bugs / Feature Requests
Java Version
Flash Version
Create Your Own / Source Code
Contact

David Galles
Computer Science
University of San Francisco

Currently, we have visualizations for the following data structures and algorithms:

- Basics
  - Stack: Array Implementation
  - Stack: Linked List Implementation
  - Queues: Array Implementation
  - Queues: Linked List Implementation
  - Lists: Array Implementation (available in java version)
  - Lists: Linked List Implementation (available in java version)
- Recursion
  - Factorial
  - Reversing a String
  - N-Queens Problem
- Indexing
  - Binary and Linear Search (of sorted list)
  - Binary Search Trees
  - AVL Trees (Balanced binary search trees)
  - Red-Black Trees
  - Splay Trees
  - Open Hash Tables (Closed Addressing)
  - Closed Hash Tables (Open Addressing)
  - Closed Hash Tables, using buckets
  - Trie (Prefix Tree, 26-ary Tree)
  - Radix Tree (Compact Trie)
  - Ternary Search Tree (Trie with BST of children)
  - B Trees
  - B+ Trees
- Sorting
  - Comparison Sorting
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Shell Sort
    - Merge Sort
    - Quck Sort
  - Bucket Sort
  - Counting Sort
  - Radix Sort
  - Heap Sort
- Heap-like Data Structures
  - Heaps
  - Binomial Queues
  - Fibonacci Heaps
  - Leftist Heaps
  - Skew Heaps

图 1.1 数据结构和算法学习网站部分截图

1.1 初识数据结构和算法

1.1.1 学习数据结构和算法的必要性

（1）互联网软件特点是高并发、高性能、高扩展、高可用、海量数据。开发互联网软件必须掌握数据结构和算法。

（2）掌握数据结构和算法能够更好地使用类库。

（3）掌握数据结构和算法是对编程精益求精的追求。

（4）掌握数据结构和算法是面试大公司的必备技能。

1.1.2 数据结构和算法的关系

程序＝数据结构＋算法

（1）数据结构是算法的基础，要学好算法，必须把数据结构学到位。学好数据结构可以为学习算法打好基础。

（2）算法是程序的灵魂。优秀的程序可以在海量数据计算时，依然保持高速计算。

（3）数据结构指的是“一组数据的存储结构”，算法指的是“操作数据的一组方法”。

（4）数据结构是为算法服务的，算法是要作用在特定的数据结构上。

从分析问题的角度去厘清数据结构和算法之间的关系。通常每个程序问题的解决都要经过以下两个步骤。

步骤 1：分析问题，从问题中提取出有价值的数据，并将其存储。

步骤 2：对存储的数据进行处理，最终得出问题的答案。

数据结构负责实现步骤 1，即解决数据的存储问题。针对数据的不同逻辑结构和物理结构，可以选出最优的数据存储结构来存储数据。

步骤 2 属于算法的职责范围。算法从字面意思来理解，即解决问题的方法。评价一个算法的好坏，取决于在解决相同问题的前提下哪种算法的效率最高。这里的效率指的就是处理数据、分析数据的能力。

所以，数据结构和算法是解决编程问题必不可少的两个步骤。毋庸置疑，数据结构不仅有用，更是每个程序员必须掌握的基本功。

1.1.3 数据结构和算法的主要学习内容

本书所讲解的数据结构和算法的主要内容如图 1.2 和图 1.3 所示。

1.2 数 据 结 构

1.2.1 数据结构的概念

数据结构是计算机存储、组织数据的方式，它是相互之间存在一种或多种特定关系的数据元素的集合。数据结构是研究数据组织方式的学科。简单来说，数据结构是用来存储数

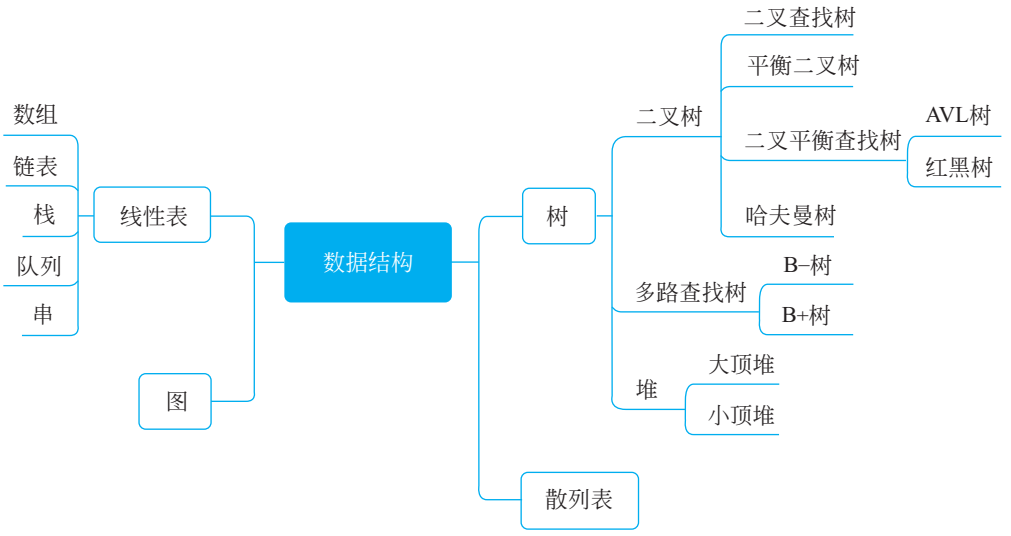

图 1.2 数据结构的学习内容

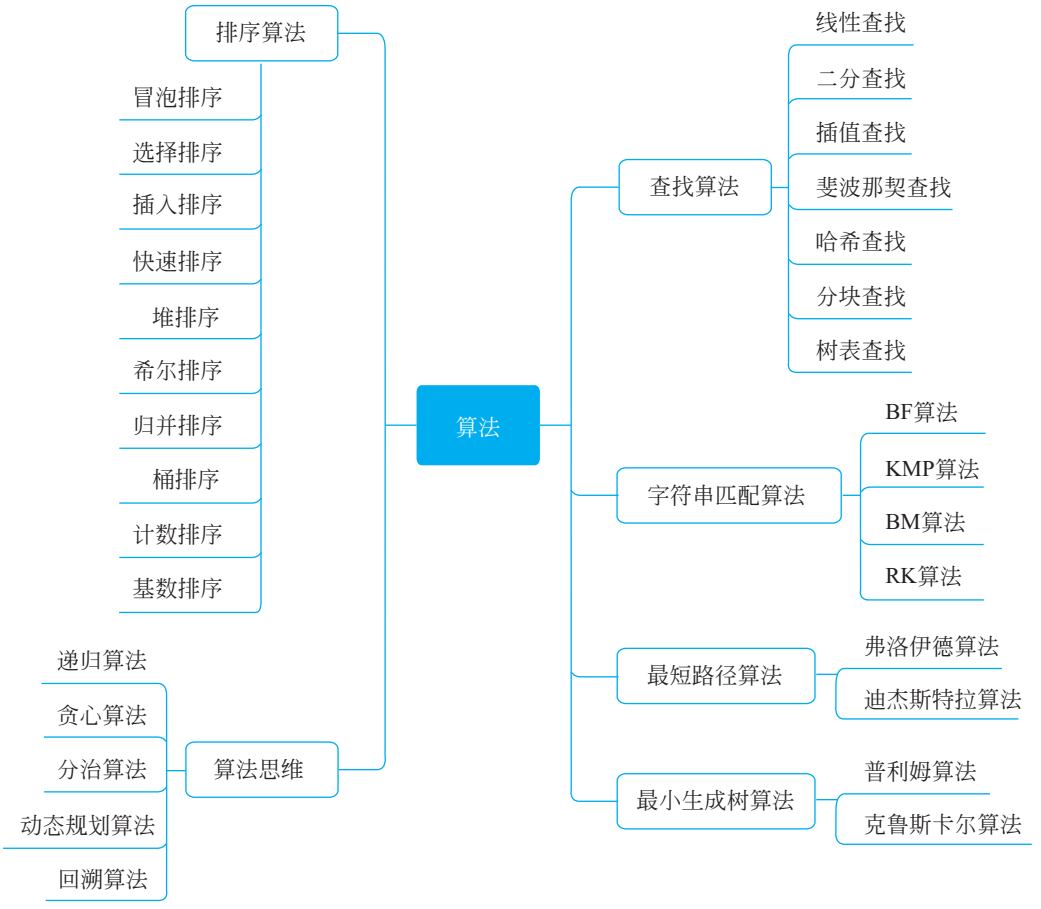

图 1.3 算法的学习内容

据的，而且是在内存中存储。有了编程语言就有了数据结构，学好数据结构可以编写出高效的代码，精心选择的数据结构可以带来最优效率的算法。

数据结构这门课程能教会我们如何存储具有复杂关系的数据，才更有利于后期对数据的再利用。

例如，图书馆的图书如何摆放，才能既方便用户查阅又方便管理员管理？

（1）图书随便摆放？

（2）图书按照书名的字母顺序摆放？

（3）图书按照类别＋书名的字母顺序摆放？

（4）图书类别的划分粒度该多大？

解决问题的效率，跟数据的组织方式有关，也跟空间的利用效率有关，还跟算法的巧妙程度有关。这也是我们必须学习数据结构和算法的原因。

1.2.2 数据结构的分类

数据结构可以按照逻辑结构与物理结构来分类。数据的逻辑结构是对数据元素之间逻辑关系的描述；数据的物理结构，又叫作存储结构，是对数据元素在计算机内存中存储情况的表示。逻辑结构与物理结构是数据结构的两个密切相关的概念，同一种逻辑结构可以对应不同的物理结构。算法的设计取决于数据的逻辑结构，算法的实现依赖于数据的存储结构。

1. 数据结构按逻辑结构划分

数据结构按逻辑结构分为两大类：线性结构（linear structure）和非线性结构（nonlinear structure）。

常用的线性结构有线性表，包括一维数组、链表、栈、队列、串；常见的非线性结构有多维数组、树（二叉树、多路树、堆）、图、散列表。

2. 数据结构按存储结构划分

数据结构按照存储结构分为：顺序存储结构、链式存储结构、索引存储结构和散列存储结构四类。

1.2.3 线性数据结构的特征

线性数据结构有如下四个特征。

（1）集合中存在唯一的一个“第一个元素”。

（2）集合中存在唯一的一个“最后一个元素”。

（3）除最后一个元素之外，其他数据元素均只有一个“后继”。

（4）除第一元素之外，其他数据元素均只有一个“前驱”。

线性结构作为最常用的数据结构，其特点是数据元素之间存在一对一的线性关系。如集合$(a0,a1,a2,\cdots,an)$，$a0$ 为第一个元素，an 为最后一个元素，此集合即为一个线性结构的集合。

1.2.4 非线性数据结构的特征

非线性数据结构不具有线性数据结构的特征，元素之间的关系可以是一对多或多对多，

其逻辑特征是一个结点元素可能有多个直接前驱和多个直接后继。

1.3 算 法

1.3.1 算法的定义

数据结构是数据的存储结构，是数据对象在计算机中存储和组织的方式。数据对象不是孤立的，必定与一系列加在其上的操作相关联，而完成这些操作所用的方法就是算法(algorithm)。

在计算机科学中，计算机程序通过一系列的数学步骤来解决问题，解决特定问题的步骤就是算法。算法是计算机科学的基石。

如果从编程的角度来描述，算法是一个有限指令集，接收一些输入(有些情况不需要输入)，产生输出，并一定在有限步骤之后终止。算法的每一条指令必须有充分明确的目标，不可以有歧义；每一条指令必须在计算机能处理的范围之内；每一条指令的描述不应依赖于任何一种计算机语言以及具体的实现手段。如 LRU(least recently used，最近最少使用)算法解决的就是当空间不够用时，应该淘汰谁的问题。这是一种策略，不是唯一的答案，所以算法无对错，只有优劣之分。

1.3.2 算法的作用

为一个任务找到最合适的算法，可以大大提升计算机的性能。算法可以在固定的硬件条件下提升系统的性能，如果没有算法，我们只能靠增加机器设备来提升系统性能，所以算法有助于系统优化。

学习算法需要掌握的知识：掌握一门编程语言，如 Java、C、C++、Python 等，理解数据结构(数组、链表、栈、堆、树、图等)的用法，且最好具有一定的数学功底。

1.3.3 学习算法前要掌握数据结构的原因

算法一般会有数据的输入，且一定会有输出，如 1～100 累加的算法。入参就是输入的数据，返回值就是输出的数据。往往有一些算法在执行之前，需要先整理数据，如把数据存起来。整理数据必然要涉及数据结构。数据提前整理好，算法可能就比较简单；数据比较杂乱，算法可能就比较复杂。

1.3.4 算法的特征

算法具有如下五个特征。

(1) 有穷性：一个算法必须总是在执行有限步数之后结束，且每一步都可在有限时间内完成。换言之，算法运行一定会结束，不会无限循环。

(2) 确定性：算法中的每一条指令必须有确切的含义，在理解时不会产生二义性。并且在任何条件下，算法只有唯一的一条执行路径，即对于相同的输入只能得出相同的输出。

(3) 可行性：一个算法中描述的操作都可以通过对基本运算的有限次执行来实现。

（4）输入：一个算法可以有零个或多个输入。有些算法的输入需要在执行过程中输入，有些算法则将输入嵌入算法之中。

（5）输出：一个算法必须有一个或多个输出，这些输出是算法对输入进行运算后的结果。

1.3.5 算法分析

在实际开发中，为了找到一个最优算法，需要反复进行复杂的数学分析，这就是算法分析。算法分析是对一个算法所需要的资源进行估算，这些资源包括计算机硬件、内存、通信宽带、运行时间等。

好的算法需要精心的设计。设计一个好的算法需要考虑达到几个目标：正确性、可读性、健壮性、效率与低存储量需求。算法的可读性是为了方便人的阅读与交流，有助于对算法的理解，晦涩难懂的算法容易隐藏错误，不利于调试和修改，也不容易被推广使用。算法的健壮性是指一个算法对不合理数据输入的反应能力和处理能力，也称为算法容错性。算法的效率指的是算法执行的时间。对同一个问题，如果有多个算法可以解决，执行时间短的算法效率就高。存储量需求指的是算法执行过程中所消耗的最大存储空间。

1.4 算法复杂度


算法复杂度

1.4.1 算法复杂度的概念

算法效率和算法存储空间是用算法复杂度来度量的。掌握和使用数据结构和算法的目的，归根到底是为了程序“快”和“省”。衡量代码的好坏，包括两个非常重要的指标：第一个是算法效率，即运行时间；第二个是存储量需求，即占用空间。对应的就是算法的两个复杂度：时间复杂度和空间复杂度。

时间复杂度表示计算机执行一段算法所需要的时长。对于计算机来说，解决同一个问题，所需时间越少的算法越优（不考虑空间问题），所以时间复杂度是衡量一个算法好坏的指标之一。由于运行环境的不同，代码的绝对执行时间是无法估计的，但是可以预估出代码的基本操作执行次数。平时我们写程序更关注时间复杂度，它是算法的基石。

1.4.2 算法不同导致执行效率不同的案例

例 1：求 1＋2＋3＋…＋n 的和。

初级程序员和高级程序员的代码通常会采用普通算法和高斯算法来实现。如代码 1.1 所示。

【代码 1.1】

```
1  /**
2   * 算法的重要性
3   * 求 1+2+3+4+5+…+n 的结果
4   * 初级程序员和高级程序员会采用不用算法,运行效率不同
5   */
```

```
6  public class SumDemo1 {
7      public static void main(String[] args) {
8          double sum = 0;
9          //放在要检测的代码段前,取开始前的时间戳
10         Long startTime = System.currentTimeMillis();
11
12         //1.调用初级程序员的方法,分别运行第 12 行和第 16 行代码,观察两者差异
13         sum = sum1(100000000);
14
15         //2.调用高级程序员的方法
16         sum = sum2(100000000);
17
18         System.out.println(sum);
19         //放在要检测的代码段后,取结束后的时间戳
20         Long endTime = System.currentTimeMillis();
21         System.out.println("花费时间" + (endTime - startTime) + "ms");
22     }
23
24     //初级程序员的代码
25     //1 + 2 + 3 + 4 + 5 +…+ n
26     //时间复杂度:O(n)
27     public static double sum1(int n) {
28         double sum = 0;
29         for (int i = 1; i <= n; i++) {
30             sum += i;
31         }
32         return sum;
33     }
34
35     //高级程序员的代码
36     //1 + 2 + 3 + 4 + 5 +…+ n
37     //时间复杂度:O(1)
38     public static double sum2(int n) {
39         return n / 2 * (n + 1);
40     }
41 }
```

代码分析

在同一台机器上运行,相同条件下运行结果,初级程序员的算法需要 176ms,而高级程序员的算法仅耗时 2ms。

普通算法和高斯算法导致运行效率相差大的原因分析如下。

1) 普通算法的执行过程

(1) int i = 1 执行 1 次。

(2) $i \leqslant n$ 执行 $n+1$ 次(因为 for 循环执行的顺序,只有 i 大于 n 时才会停止循环,所以 $i=n+1$ 时,还会再判断一下 $i \leqslant n$,所以相比较而言会多执行一次)。

（3）i ++ 执行 n 次。

（4）sum += 1 执行 n 次。

（5）汇总以上步骤，上述代码执行 $1+n+1+n+n=3n+2$ 次。

（6）用极限思维，$n\to\infty$，$3n+2\to 3n\to n$，记作 O(n)。

（7）O(n)就是上述代码的时间复杂度。

2）高斯算法的执行过程

同样计算 1 加到 n，采用高斯算法就简单多了。上述代码只需要执行 1 次，没有循环。所以时间复杂度就是 O(1)。

O(1)和 O(n)的区别是什么呢？

当初级程序员代码和高级程序员代码中的变量 n 不断增大的时候，高斯算法消耗的时间基本不变，但是初级程序员代码的时间就会增加。

对于计算机来说，高斯算法求解 1 连续加到 n 的计算速度远远大于 for 循环的速度。速度越快，系统的性能就会越好。

例 2：求 $x+x^2+x^3+x^4+\cdots+x^n$ 的和。

初级程序员和高级程序员采用不同的算法，如代码 1.2 所示。

【代码 1.2】

```
/**
 * 算法的重要性
 * 求 x^1 + x^2 + x^3 + x^4 + … + x^n 的结果
 * 初级程序员和高级程序员会采用不用算法，运行效率不同
 */
public class SumDemo2 {
    public static void main(String[] args) {
        double sum = 0;
        //放在要检测的代码段前，取开始前的时间戳
        Long startTime = System.currentTimeMillis();

        //调用初级程序员的方法，分别运行第 13 行和第 16 行代码，观察其差异
        sum = sum3(2, 1000000);

        //调用高级程序员的方法
        sum = sum4(2, 1000000);

        System.out.println(sum);
        //放在要检测的代码段后，取结束后的时间戳
        Long endTime = System.currentTimeMillis();
        System.out.println("花费时间" + (endTime - startTime) + "ms");
    }

    //初级程序员的代码
    //x^1+x^2+x^3+x^4+…+x^n
    //时间复杂度：O(n)
    public static double sum3(int x, int n) {
        double sum = 0;
```

```
29          for (int i = 1; i <= n; i++) {
30              sum += Math.pow(x, i);
31          }
32          return sum;
33      }
34
35      //高级程序员的代码
36      //x^1+x^2+x^3+x^4+…+x^n
37      //时间复杂度:O(1)
38      /**
39       * s = x^1+x^2+x^3+x^4+…+x^n
40       * sx = x(x^1+x^2+x^3+x^4+…+x^n) //等式左右两侧同时乘以 x
41       * sx = x^2+x^3+x^4+…+x^(n+1))
42       * sx - s = x^(n+1)-x
43       * s = (x^(n+1)-x)/(x-1)
44       */
45      public static double sum4(int x, int n) {
46          return (Math.pow(x,n+1)-x)/(x-1);
47      }
48  }
```

代码分析

在同一台机器上运行,相同条件下运行结果,初级程序员的算法需要 398ms,而高级程序员的算法耗时 1ms。

例 3:求 a[0]+a[1]*x+a[2]*x^2+a[3]*x^3+…+a[n]*x^n 的和。

初级程序员和高级程序员采用不同的算法,如代码 1.3 所示。

【代码 1.3】

```
1   /**
2    * 算法的重要性
3    * 求 a[0] + a[1] * x^1 + a[2] * x^2 + a[3] * x^3 + … + a[n] * x^n 的结果
4    * 初级和高级程序员采用不用算法,运行效率不同
5    */
6   public class SumDemo3 {
7       public static void main(String[] args) {
8           double sum = 0;
9           //放在要检测的代码段前,取开始前的时间戳
10          Long startTime = System.currentTimeMillis();
11
12          //定义一个数组
13          double arr[] = new double[1000000];
14          //给数组动态赋值
15          for (int i = 0; i < arr.length; i++) {
16              arr[i] = i + 1;
17          }
18
19          //调用初级程序员的方法,分别运行第 20 行和第 23 行代码,观察其差异
```

```
20          sum = sum5(2, arr);
21
22          //调用高级程序员的方法
23          sum = sum6(2, arr);
24
25          System.out.println(sum);
26          //放在要检测的代码段后,取结束后的时间戳
27          Long endTime = System.currentTimeMillis();
28          System.out.println("花费时间" + (endTime - startTime) + "ms");
29      }
30
31      //初级程序员的代码
32      //a[0] + a[1] * x + a[2] * x^2 + a[3] * x^3 + ... + a[n] * x^n
33      //计算机运行加减法的速度比运行乘除法的速度快很多
34      //以下循环执行乘法的次数为:(n^2 + n)/2 次
35      //时间复杂度:O(n^2)
36      public static double sum5(int x, double[] arr) {
37          double sum = arr[0];
38          for (int i = 1; i <= arr.length - 1; i++) {
39              sum += arr[i] * Math.pow(x, i);
40          }
41          return sum;
42      }
43
44      //高级程序员的代码
45      //a[0] + a[1] * x + a[2] * x^2 + a[3] * x^3 + ... + a[n] * x^n
46      //a[0] + x(a[1] + x(...(a[n-1] + x * a[n]))...)
47      //时间复杂度:O(n)
48      public static double sum6(int x, double[] arr) {
49          double sum = arr[arr.length - 1];
50          for (int i = arr.length - 1; i > 0; i--) {
51              sum = arr[i - 1] + x * sum;
52          }
53          return sum;
54      }
55  }
```

代码分析

在同一台机器上运行，相同条件下运行结果，初级程序员的算法需要 404ms，而高级程序员的算法耗时 13ms。

1.4.3 算法的时间复杂度表示法

时间复杂度的表示法源于数学的极值问题。例如，一元二次函数 $f(x)=2x^2+2x+2$，当持续增大 x 的值，甚至 x 为无穷大时，$2x^2+2x+2$ 表达式中的 $2x+2$ 这一项就可以忽略不计，在极限思想里面，$2x^2$ 前面的系数 2 也可以省略。也就是说 $x\to\infty$ 时，$f(x)=2x^2+2x+2\approx x^2$。

所以对于以下两个表达式：$T(n)=O(2n+2)$ 和 $T(n)=O(2n^2+2n+3)$，当 n 很大时，