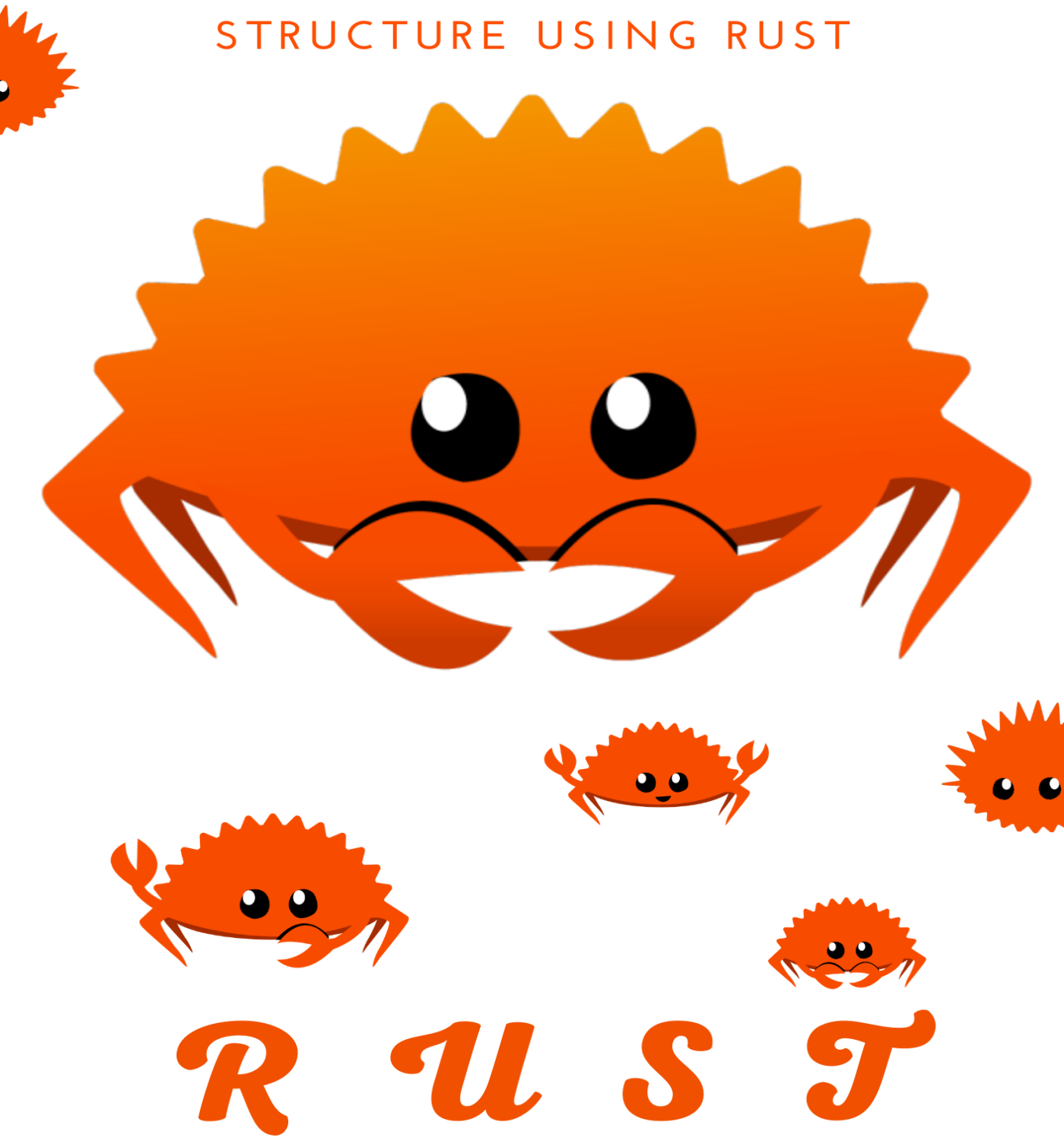
FOR OPEN SOURCE COMMUNITY

PROBLEM-SOLVING WITH ALGORITHMS AND DATA STRUCTURE USING RUST



Problem-solving with algorithms and data structures using Rust

Shieber

2021.02.11

序

晶体管的出现引发了集成电路和芯片革命，人类有了中央处理器，大容量存储器和便捷的通讯设施。Multics [1] 的失败催生了 Unix [2]，而后出现的 Linux 内核 [3] 及发行版 [4] 则将开源与网络技术紧密结合起来，使得信息技术得到飞速发展。技术的进步为社会提供了实践想法的平台和工具，社会的进步则创造了新的需求和激情，继而又推动技术再进步。尽管计算机世界的上层诞生了互联网、区块链、云计算、物联网，但在底层，其基本原理保持不变。变化的特性往往有不变的底层原理作为支撑，就像各种功能的蛋白质也只是几十种氨基酸组成的肽链的曲折变化一样。计算机世界的底层是基本硬件及其抽象数据类型（数据结构）和操作（算法）的组合，这是变化的上层技术取得进步的根本。

不管是普通计算机，超级计算机亦或将来的量子计算机 [5]，其功能都要建立在某种数据结构和算法的抽象之上。数据结构和算法作为抽象数据类型在计算机科学中具有重要地位，是完成各种计算任务的核心工具。本书重点关注抽象数据类型的设计、实现和使用。通过学习设计抽象数据类型有助于编程实现，而通过编程实现则可加深对抽象数据类型的理解。

本书的算法实现往往不是最优或最通用的工程实现，因为工程实现代码非常冗长，让人抓不住重点，这对于学习原理是有害的。本书代码对不同问题采取不同简化措施，有的直接用泛型，有的则使用具体类型。这些实现方法一是为简化代码，二是确保每段代码都可单独编译通过并得到运行结果。注意，本书使用的 Rust 版本为 1.58。

基本要求

虽然理解抽象数据类型概念不涉及到具体的形式，但代码实现却必须要考虑具体的形式，这就要求本书读者具有一定的 Rust 基础。虽然每个人对 Rust 的熟悉程度和编码习惯有所不同，但基本要求却是相通的。要阅读本书，读者最好具有以下能力和兴趣：

能使用 Rust 实现完整的程序，包括使用 Cargo、rustc、test 等。

能使用基本的数据类型和结构，包括结构体、枚举体、循环、匹配等。

能使用 Rust 泛型，生命周期、所有权系统、指针、非安全代码、宏等。

能使用内置库（crate）、外部库，会设置 Cargo.toml。

如果你不会这些，那么可以翻到第一章末尾，从推荐的学习材料里找一些书籍和资料来学习 Rust 这门语言，之后再回到本书，从头开始学习。本书代码均按照章节和名称保存在 github code 和 gitee code 仓库，欢迎下载使用及指出错误。

全书结构

为让读者对全书结构有较好的把握以及便于高级用户选择阅读的章节，下面将全书内容做一个总结。全书分为九章，前两章介绍计算机科学的概念以及算法分析，是整本书的基础。第二到第六章是简单数据结构和算法的设计和实现。第七和八章是较复杂的树及图数据结构，树和图是许多大型软件的底层实现，这两章是基于前几章的更高级主题。最后一章是利用前面所学内容进行的实战项目，通过实战将所学数据结构和算法用于解决实际问题。当然，全书的章节不是死板的，你可以选择先学某些章节再学其他内容。但总的来说，前两章要先看，中间的章节及最后的实战也建议按顺序阅读。

第一章：计算机科学。通过学习计算科学定义和概念能指导个人如何分析问题。其中包括如何建立抽象数据类型模型，设计、实现算法及结果检验。抽象数据类型是对数据操作的逻辑描述，隐藏了实现细节，有助于更好地抽象出问题本质。本章的最后是 Rust 基础知识回顾和学习资源总结。

第二章：算法分析。算法分析是理解程序执行时间和空间性能的方法。由算法分析能得到算法执行效率，比如时间、内存消耗。大 O 分析法是算法分析的一种标准方法。

第三章：基本数据结构。计算机是个线性的系统，对于内存来说也不例外。基本数据结构保存在内存中，所以也是线性的。Rust 中基于这种线性内存模型建立的基本数据结构有数组、切片、Vec 以及衍生的栈、队列等。本章学习用 Vec 这种基本数据结构来实现栈、队列、双端队列、链表。

第四章：递归。递归是一种算法技巧，是迭代的另一种形式，必须满足递归三定律。尾递归是对递归的一种优化。动态规划是一类高效算法的代表，通常利用递归或迭代来实现。

第五章：查找。查找算法用于在某类数据集合中找到某个元素或判断元素是否存在，是使用最广泛的算法。根据数据集合是否排序可以粗略地分为顺序查找和非顺序查找。非顺序查找算法包括二分查找和哈希查找等算法。

第六章：排序。前一章的顺序查找算法要求数据有序，而数据一般是无序的，所以需要排序算法。常见的排序算法有十类，包括冒泡排序、快速排序、插入排序、希尔排序、归并排序、选择排序、堆排序、桶排序、计数排序、基数排序。蒂姆排序算法是结合归并和插入排序的排序算法，效率非常高，已经是 Java、Python、Rust 等语言的默认排序算法。

第七章：树。计算机是线性系统，但在线性系统上，通过适当的方法也能构造出非线性的数据结构。树，正是一种非线性数据结构，它通过指针或引用来指向子树。树中最基础的是二叉树，由它衍生了二叉堆、二叉查找树、平衡二叉树、八叉树。当然，还有 B 树、B+ 树、红黑树等更复杂的树。

第八章：图。树的连接从根到子，且数量少。如果将这些限制取消，那么就得到了图数据结构。图是一种解决复杂问题的非线性数据结构，连接方向可有可无，没有父子结点的区别。虽然是非线性数据结构，但其存储形式也是线性的，包括邻接表和邻接矩阵。图用于处理含有大量结点和连接关系的问题，例如网络流量、交通流量、路径搜索等问题。

第九章：实战。在前面八章的基础上，本章通过运用所学知识来解决实际问题，实现一些有用的数据结构和算法。包括距离算法、字典树、过滤器、缓存淘汰算法、一致性哈希算法以及区块链。通过这些实战项目，读者定能加深对数据结构的认识并提升 Rust 编码水平。

致谢

高效、安全以及便捷的工程管理工具使得 Rust 成为了一门非常优秀的语言，也是未来可能替代部分 C/C++ 工作的最佳语言（目前 Rust 正逐步加入 Linux 内核）。市面上已经出版了许多 Rust 书籍，但关于算法的书籍要么是作者没看到，要么就是没有。因为没有 Rust 算法书籍，所以作者自己在学习过程中也不断碰壁。在这样的情况下，一部简单便捷的 Rust 书籍对新人学习算法和数据结构来说必然大有帮助。经过一段时间的资料查阅 [6]、思考、整理并结合自己的学习经历，作者完成了这本书。虽然 Rust 学习曲线陡峭，但只要找准方向，有好的资源，就一定能学好，希望本书能做点微小的贡献。

编写这本书主要是为了学习推广 Rust，以及回馈整个开源社区，是开源社区的各种资源让作者学习和成长。感谢 PingCap 开发的 TiDB 以及其运营的开源社区和线上课程。感谢 Rust 语言中文社区的 Mike Tang 等成员对 Rust 会议的组织、社区的建设维护。感谢令胡壹冲在 Bilibili 弹幕网分享的 Rust 学习视频。感谢张汉东等前辈对 Rust 语言的推广，包括他写的优秀书籍《Rust 编程之道》以及 RustMagazine 中文月刊。当然还要感谢 Mozilla、AWS、Facebook、谷歌、微软、华为公司为 Rust 设立基金会 [7]，正是这个基金会使得作者断定 Rust 的未来一片光明，还缺少学习资源，也才有了去写本书的动力。

最后，要感谢电子科技大学提供的学习资源和环境，感谢导师和 KC404 的众位师兄弟姐妹的关心和帮助。在这里，作者学到了各种技术、文化，得到了成长，更找准了人生前行的方向。

Shieber 成都

目录

序 **1**

1 计算机科学 **7**

1.1 本章目标 7
1.2 快速开始 7
1.3 什么是计算机科学 7
1.4 什么是编程 9
1.5 为什么学习数据结构 9
1.6 为什么学习算法 10
1.7 Rust 基础 10
1.7.1 安装 Rust 10
1.7.2 Rust 工具链 11
1.7.3 Rust 回顾 11
1.7.4 Rust 学习资料 12
1.8 总结 13

2 算法分析 **14**

2.1 本章目标 14
2.2 什么是算法分析 14
2.3 大 O 分析法 17
2.4 乱序字符串检查 19
2.4.1 穷举法 19
2.4.2 检查法 20
2.4.3 排序和比较法 21
2.4.4 计数和比较法 22
2.5 Rust 数据结构的性能 24
2.5.1 标量和复合类型 24
2.5.2 集合类型 25
2.6 总结 26

3 基本数据结构 **27**

3.1 本章目标 27
3.2 线性数据结构 27
3.3 栈 28
3.3.1 栈的抽象数据类型 . . . 29
3.3.2 Rust 实现栈 29
3.3.3 括号匹配 31
3.3.4 进制转换 36
3.3.5 前中后缀表达式 39
3.3.6 中缀转前后缀表达式 . . 41
3.4 队列 47
3.4.1 队列的抽象数据类型 . . 48
3.4.2 Rust 实现队列 49
3.4.3 烫手山芋 50
3.5 双端队列 52
3.5.1 双端队列的抽象数据类型 53
3.5.2 Rust 实现双端队列 . . . 54
3.5.3 回文检测 56
3.6 链表 57
3.6.1 链表的抽象数据类型 . . 58
3.6.2 Rust 实现链表 59
3.6.3 链表栈 64
3.7 Vec 66
3.7.1 Vec 的抽象数据类型 . . 66
3.7.2 Rust 实现 Vec 67
3.8 总结 71

4 递归 72
4.1 本章目标 72
4.2 什么是递归 72
4.2.1 递归三定律 74
4.2.2 到任意进制的转换 . . . 75
4.2.3 汉诺塔 77
4.3 尾递归 79
4.3.1 递归和迭代 80
4.4 动态规划 81
4.4.1 什么是动态规划 84
4.4.2 动态规划与递归 87
4.5 总结 88

5 查找 89
5.1 本章目标 89
5.2 查找 89
5.3 顺序查找 90
5.3.1 Rust 实现顺序查找 . . . 90
5.3.2 顺序查找复杂度 92
5.4 二分查找 94
5.4.1 Rust 实现二分查找 . . . 94
5.4.2 二分查找复杂度 97
5.4.3 内插查找 97
5.4.4 指数查找 99
5.5 哈希查找 100
5.5.1 哈希函数 101
5.5.2 解决冲突 103
5.5.3 Rust 实现 HashMap . . 105
5.5.4 HashMap 复杂度 109
5.6 总结 109

6 排序 110
6.1 本章目标 110
6.2 什么是排序 110
6.3 冒泡排序 111
6.4 快速排序 118
6.5 插入排序 120
6.6 希尔排序 123
6.7 归并排序 125
6.8 选择排序 127
6.9 堆排序 129
6.10 桶排序 132
6.11 计数排序 135
6.12 基数排序 137
6.13 蒂姆排序 139
6.14 总结 140

7 树 142
7.1 本章目标 142
7.2 什么是树 142
7.2.1 树的定义 145
7.2.2 树的表示 146
7.2.3 分析树 150
7.2.4 树的遍历 151
7.3 基于二叉堆的优先队列 155
7.3.1 二叉堆定义 155
7.3.2 Rust 实现二叉堆 156
7.3.3 二叉堆分析 167
7.4 二叉查找树 167
7.4.1 二叉查找树操作 167
7.4.2 Rust 实现二叉查找树 . 168
7.4.3 二叉查找树分析 176
7.5 平衡二叉树 177
7.5.1 AVL 平衡二叉树 177
7.5.2 Rust 实现平衡二叉树 . 178
7.5.3 平衡二叉树分析 187
7.6 总结 188

8 图 189
8.1 本章目标 189
8.2 什么是图 189
8.2.1 图定义 190
8.3 图的存储形式 190
8.3.1 邻接矩阵 191
8.3.2 邻接表 191
8.4 图的抽象数据类型 192
8.5 图的实现 193
8.5.1 图解决字梯问题 201

8.6 广度优先搜索 204
8.6.1 实现广度优先搜索 . . . 204
8.6.2 广度优先搜索分析 . . . 209
8.6.3 骑士之旅 210
8.6.4 图解决骑士之旅 210
8.7 深度优先搜索 213
8.7.1 实现深度优先搜索 . . . 214
8.7.2 深度优先搜索分析 . . . 218
8.7.3 拓扑排序 218
8.8 强连通分量 220
8.9 最短路径问题 222
8.9.1 Dijkstra 算法 223
8.9.2 实现 Dijkstra 算法 . . . 224
8.9.3 Dijkstra 算法分析 . . . 227
8.10 总结 227

9 实战 228

9.1 本章目标 228
9.2 编辑距离 228
9.2.1 汉明距离 228
9.2.2 莱文斯坦距离 230
9.3 字典树 235
9.4 过滤器 238
9.4.1 布隆过滤器 238
9.4.2 布谷鸟过滤器 242
9.5 缓存淘汰算法 LRU 248
9.6 一致性哈希算法 254
9.7 Base58 编码 259
9.8 区块链 266
9.8.1 区块链及比特币原理 . . 266
9.8.2 基础区块链 267
9.8.3 工作量证明 272
9.8.4 区块链存储 275
9.8.5 交易 279
9.8.6 账户 282
9.8.7 梅根哈希 285
9.8.8 矿工及挖矿 287
9.8.9 比特币奖励 293
9.8.10 回顾 294
9.9 总结 295

Chapter 1

计算机科学

1.1 本章目标

了解计算机科学的思想
了解抽象数据类型的概念
回顾 Rust 编程语言基础知识

1.2 快速开始

在计算机技术领域，每个人都要花相当多时间来学习该领域的基础知识，希望有足够的能力把问题弄清楚并想出解决方案。但是对有些问题，你发现要编写代码却很困难。问题的复杂性和解决方案的复杂性往往会掩盖与解决问题过程相关的思想。一个问题，往往存在多种解决方案，每种方案都由其问题陈述结构和逻辑所限定。然而，你可能会将解决 A 问题的陈述结构和解决 B 问题的逻辑结合起来，然后自己给自己制造麻烦。本章首先回顾计算机科学、算法和数据结构，特别是研究这些主题的原因，并希望借此帮助我们看清解决问题的陈述结构和逻辑。其次，本章还回顾了 Rust 编程语言，给出了一些学习资料。

1.3 什么是计算机科学

计算机科学往往难以定义，这可能是由于在名称中使用了“计算机”一词。如你所知，计算机科学不仅仅是对计算机的研究，虽然计算机作为一个工具在其中发挥着最为重要的作用，但它只是个工具。计算机科学是对问题、解决方案及产生方案的过程的研究。给定某个问题，计算机科学家的目标是开发一个算法，一系列指令列表，用于解决可能出现的该类问题的任何实例。遵循这套算法，在有限的时间内就能解决类似问题。计算机科学可以认为是对算法的研究，但必须认识到，某些问题可能没有解决的算法。这些问题可能是 NPC 问题 [8]，目前不能解决，但对其的研究却是很重要的，因为解决这些难题意味着技术的突破。就像“歌德

巴赫猜想[9]”一样，单是对其的研究就发展出不少工具。或许可以这么定义计算机科学：一门研究可解决问题方案和不可解决问题思想的科学。

在描述问题和解决方案时，如果存在算法能解决这个问题，那么就称该问题是可计算的。计算机科学的另一个定义是“针对那些可计算和不可计算的问题，研究是不是存在解决方案”。你会注意到“计算机”一词根本没有出现在此定义中。解决方案独立于机器，是一整套思想，和是否用计算机无关。

计算机科学涉及问题解决过程的本身，也就是抽象。抽象使人类能够脱离物理视角而采用逻辑视角来观察问题并思考解决方案。假设你开车上学或上班。作为老司机，你为了让汽车载你到目的地，会和汽车有些互动。你进入汽车，插入钥匙，点火，换挡，制动，加速和转向。从抽象的角度来说，你看到的是汽车的逻辑面，你正在使用汽车设计师提供的功能将你从一个位置运输到另一个位置，这些功能有时也被称为接口。另一方面，汽车修理师则有一个截然不同的视角。他不仅知道如何开车，还知道汽车内部所有必要的细节。他了解发动机是如何工作的，变速箱如何变速，温度如何控制，雨刷如何转动等等。这些问题都属于物理面，细节都发生在“引擎盖下”。

普通用户使用计算机也是基于这样的视角。大多数人使用计算机写文档、收发邮件、看视频、浏览新闻、听音乐等，但用户并不知道让这些程序工作的细节。他们从逻辑或用户视角看计算机。计算机科学家、程序员、技术支持人员和系统管理员看计算机的角度则截然不同，他们必须知道操作系统工作细节，如何配置网络协议，以及如何编写各种控制功能脚本，他们必须能够控制计算机的底层。

这两个例子的共同点就是用户态的抽象，有时也称为客户端。用户不需知道细节，只要知道接口工作方式就能与底层沟通。比如 Rust 的数学计算函数 sin，你可以直接使用。

```
1 // sin_function.rs
2 fn main() {
3     let x: f32 = 2.0;
4     let y = x.sin();
5     println!("sin(2) is {y}"); // 0.9092974
6 }
```

这就是抽象。我们不一定知道如何计算正弦值，但只要知道函数是什么以及如何使用它就行了。如果正确地输入，就可以相信该函数将返回正确的结果。我们知道一定有人实现了计算正弦的算法，但细节我们不知道，所以这种情况也被称为“黑箱”。只要简单地描述下接口：函数名称、参数、返回值，我们就能够使用，细节都隐藏在黑箱里面，如图（1.1）所示。

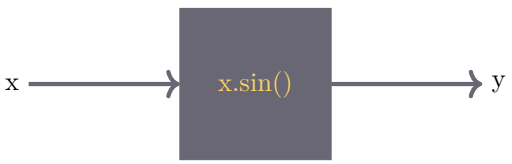


图 1.1: sin 函数

1.4 什么是编程

编程是将算法（解决方案）编码为计算机指令的过程。虽然有许多编程语言和不同类型的计算机存在，但第一步是需要有解决方案，没有算法就没有程序。计算机科学不是研究编程，然而编程是计算机科学家的重要能力。编程通常是为解决方案创建的表现形式，是问题及解决思路的陈述，这种陈述主要提供给计算机，其实编程就是梳理自己头脑中问题的陈述结构的过程。

算法描述了依据问题实际数据所产生的解决方案和产生预期结果所需的一套步骤。编程语言必须提供一种表示方法来表示过程和数据。为此，编程语言必须提供控制方法和各种数据类型。控制方法允许以简洁而明确的方式表示算法步骤。至少，算法需要执行顺序处理、决策选择和重复迭代。只要语言提供这些基本语句，它就可用于算法表示。

计算机中的所有数据项都以二进制形式表示。为了赋予二进制形式数据具体含义，就需要有数据类型。数据类型提供了对二进制数据的解释方法和呈现形式。数据类型是对物理世界的抽象，用于表示问题所涉及的实体。这些底层的数据类型（有时称为原始数据类型）为算法开发提供了基础。例如，大多数编程语言提供整数、浮点数数据类型。内存中的二进制数据可以解释为整数、浮点数，并且和现实世界的数字（例如-3、2.5）相对应。此外，数据类型还描述了数据可能存在的操作。对于数字，诸如加减乘除法的操作是最基本的。通常遇到的困难是问题及其解决方案非常复杂，编程语言提供的简单结构和数据类型虽然足以表示复杂的解决方案，但通常不便于使用。要控制这种复杂性，需要采用更合理的数据管理方式（数据结构）和操作流程（算法）。

1.5 为什么学习数据结构

为了管理问题的复杂性和获取解决问题的具体步骤，计算机科学家通过抽象以使自己能够专注于大问题而不会迷失在细节中。通过创建问题域模型，计算机能够更有效地解决问题。这些模型允许以更加一致的方式描述算法要处理的数据。

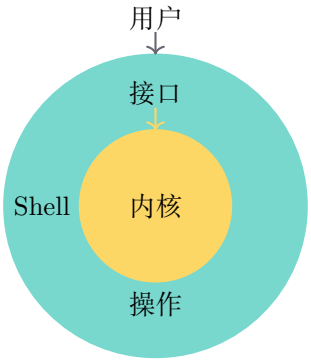


图 1.2: 系统层级图

先前，我们将解决方案抽象称为隐藏特定细节的过程，以允许用户或客户端从高层使用。

现在转向类似的思想，即对数据的抽象。抽象数据类型（Abstract Data Type，ADT）是对如何查看数据和操作数据的逻辑描述。这意味着只用关心数据表示什么，而不关心它最终形式。通过提供这种级别的抽象，就给数据创建了一个封装，隐藏了实现细节。上图展示了抽象数据类型是什么以及如何操作。用户与接口的交互是抽象的操作，用户和 shell 是抽象数据类型。

抽象数据类型的实现要求从物理视图使用原始数据类型来构建新的数据类型，我们又称其为数据结构。通常有许多不同的方法来实现抽象数据类型，但不同的实现要有相同的物理视图，允许程序员在不改变交互方式的情况下改变实现细节，用户则继续专注于问题。

1.6 为什么学习算法

在计算机世界中，使用一个更快，或者占用更少内存的算法是我们的目标，因为这具有很多显而易见的好处。在最坏的情况下，可能有一个难以处理的问题，没有什么算法在可预期的时间内能给出答案。但重要的是能够区分具有解决方案的问题和不具有解决方案的问题，以及存在解决方案但需要大量时间或其他资源的问题。作为计算机科学家需要一遍又一遍比较，然后决定某个方案是否是一个好的方案并决定采用的最终方案。

通过看别人解决问题来学习是一种高效的学习方式。通过接触不同问题的解决方案，看不同的算法设计如何帮助我们解决具有挑战性的问题。通过思考各种不同的算法，我们能发现其核心思想，并开发出一套具有普适性的算法，以便下一次出现类似的问题时能够很好地解决。同样的问题，不同人给出的算法实现常彼此不同。就像前面看到的计算 sin 的例子，完全可能存在许多种不同的实现版本。如果一种算法可以使用比另一种更少的资源，比如另一个算法可能需要 10 倍的时间来返回结果。那么即使两个算法都能完成计算 sin 函数，但时间少的显然更好。

1.7 Rust 基础

1.7.1 安装 Rust

Mac OS，Linux 等类 Unix 系统请使用如下命令安装 Rust，Windows 下的安装方式请读者到官网查看。

$ curl –proto '=https' –tlsv1.2 -sSf https://sh.rustup.rs | sh

安装好后还需设置环境变量，让系统能够找到 rustc 编译器等工具的位置。对于 Linux 来说，将如下三行加入 ~/.bashrc 末尾，注意修改 usrname 为自己用户名。

```
# Rust 语言环境变量
export RUSTPATH=/home/usrname/.cargo/bin
export PATH=$PATH:$RUSTPATH
```

保存退出后再执行 source ~/.bashrc 就可以了。如果嫌麻烦，可以到本书源码第一章下载 install_rust.sh，执行该脚本就可以完成安装。

1.7.2 Rust 工具链

上面的指令会安装 rustup 这个工具来管理 Rust 工具链。Rust 工具链包括编译器 rustc、项目管理工具 cargo、管理工具 rustup、文档工具 rustdoc、格式化工具 rustfmt、调试工具 rust-gdb。

平时写写简单代码可以使用 rustc 编译，但涉及大项目时还是需要使用 cargo 工具来管理，这是一个非常好的工具，尤其是管理过 C++ 工程的程序员肯定会欢呼这个工具的伟大。cargo 工具集项目构建、测试、编译、发布于一体。当然，cargo 内部是调用 rustc 来编译的，本书项目多不复杂，所以大部分时候也使用 rustc 编译器而不是 Cargo。

rustup 管理着 Rust 工具的安装、升级、卸载。注意，Rust 语言包括稳定版和 nightly 版。nightly 版包括最新特性，更新也更快，首次安装时默认是没有的，可用如下命令安装。

$ rustup default nightly

安装好后可用 rustup 查看。

$ rustup tooolchain list

stable-x86_64-unknown-linux-gnu

nightly-x86_64-unknown-linux-gnu (default)

要切换回 stable 版使用如下命令。

$ rustup default stable

1.7.3 Rust 回顾

Rust 是一门类似 C/C++ 的底层编程语言，这意味着你在那两门语言中学习的很多概念都能用于帮助理解 Rust。然而 Rust 也提出了自己独到的见解，特别是可变、所有权、借用、生命周期这几大概念，是 Rust 的优点，也是其难点。

```
1  // rust_basic.rs
2  fn sum_of_val(nums: &[i32], num: i32) -> i32 {
3      let mut sum: i32 = 0;
4      for n in nums {
5          sum += n;
6      }
7      sum + num
8  }
9
10 fn main() {
11     let num = 10;
12     let nums = [1,2,3,4,5,6,7,8];
13     let sum = sum_of_val(&nums, num);
14     println!("sum is {sum}");
15 }
```

上述代码表明 Rust 需要 main 函数，展示了 println 的格式化输出，使用 fn 来定义下划线风格的函数以及用 -> 表示返回值，最后一行用无分号来表示返回（return）。

下面是 Rust 目前在用的（未来可能增加）关键字，共 39 个，还是比较多的，所以学习要困难一些，特别注意 Self 和 self。

```
1  Self      enum      match     super
2  as        extern    mod       trait
3  async     false     move      true
4  await     fn        mut       type
5  break     for       pub       union
6  const     if        ref       unsafe
7  continue  impl      return    use
8  crate     in        self      where
9  dyn       let       static    while
10 else      loop      struct
```

1.7.4 Rust 学习资料

书籍文档

入门：《Rust 程序设计语言》、《深入浅出 Rust》、《Rust 编程之道》、《通过例子学 Rust》、《Rust Primer》、《Rust Cookbook》、《Rust in Action》、《Rust 语言圣经》

进阶：《Cargo 教程》、《Rustlings》、《通过链表学 Rust》、《Rust 设计模式》

高阶：《rustc 手册》、《Rust 宏小册》、《Rust 死灵书》、《Rust 异步编程》

特定领域

Wasm：https://wasmer.io、https://wasmtime.dev、https://wasmedge.org

HTTP/3: https://github.com/cloudflare/quiche

coreutils: https://github.com/uutils/coreutils

算法：https://github.com/TheAlgorithms/Rust

游戏：https://github.com/bevyengine/bevy

工具：https://github.com/rustdesk/rustdesk

区块链：https://github.com/w3f/polkadot

数据库：https://github.com/tikv、https://github.com/tensorbase/tensorbase

编译器：https://github.com/rust-lang/rustc_codegen_gcc

操作系统：https://github.com/Rust-for-Linux、https://github.com/rcore-os

Web 前端：https://github.com/yewstack/yew、https://github.com/denoland/deno

Web 后端：https://actix.rs/、https://github.com/tokio-rs/axum、https://github.com/poem-web/poem

资源网站

Rust 官网：https://www.rust-lang.org
Rust 源码：https://github.com/rust-lang/rust
Rust 文档：https://doc.rust-lang.org/stable
Rust 参考：https://doc.rust-lang.org/reference
Rust 杂志：https://rustmagazine.github.io/rust_magazine_2021
Rust 库/箱：https://crates.io、https://lib.rs
Rust 中文社区：https://rustcc.cn
Rust 乐酷论坛：https://learnku.com/rust
Rust LeetCode：https://rustgym.com/leetcode
Awesome Rust：https://github.com/rust-unofficial/awesome-rust
Rust Cheat Sheet：https://cheats.rs
芽之家：https://budshome.com/books.html
令狐壹冲：https://space.bilibili.com/485433391

1.8 总结

本章介绍了计算机科学的思想和抽象数据类型的概念，明确了算法和数据结构的定义和作用。其次，回顾了 Rust 基础知识并总结了部分学习资源。

Chapter 2

算法分析

2.1 本章目标

理解算法分析的重要性
能够使用大 O 符号分析算法执行时间
理解 Rust 数组等数据结构的大 O 分析结果
理解 Rust 数据结构的实现是如何影响算法分析的
学习对简单的 Rust 程序做性能基准测试

2.2 什么是算法分析

正如我们在第一章中所说，算法是一个通用的，解决某种问题的指令列表。它是用于解决一类问题任何实例的方法，给定特定输入会产生期望的结果。另一方面，程序是使用某种编程语言编码的算法。由于程序员知识水平各异且所使用的编程语言各有不同，存在描述相同算法的不同程序。

一个普遍的现象是，刚接触计算机的学生会将自己的程序和其他人的相比较。你可能注意到，这些程序看起来很相似。那么当两个看起来不同的程序解决同样的问题时，哪一个更好呢? 要探讨这种差异，请参考如下的函数 sum_of_n，该函数计算前 n 个整数的和。

```
1 // sum_of_n.rs
2 fn sum_of_n(n: i32) -> i32 {
3     let mut sum: i32 = 0;
4     for i in 1..=n {
5         sum += i;
6     }
7     sum
8 }
```

该算法使用初始值为 0 的累加器变量。然后迭代 n 个整数，将每个值依次加到累加器。现在再看看下面的函数，你可以看到这个函数本质上和前一个函数在做同样的事情。不直观的原因在于编码习惯不好，代码中没有使用良好的标识符名称，所以代码不易读，且在迭代步骤中使用了一个额外的赋值语句，这个语句其实是不必要的。

```
// foo.rs
fn foo(tom: i32) -> i32 {
    let mut fred = 0;
    for bill in 1..=tom {
        let barney = bill;
        fred = fred + barney;
    }
    fred
}
```

先前提出了一个的问题：哪个函数更好？答案其实取决于读者的评价标准。如果你关注可读性，函数 sum_of_n 肯定比 foo 好。你可能已经在各种介绍编程的书或课程中看到过类似例子，其目的之一就是帮助你编写易于阅读和理解的程序。然而，在本书中，我们对算法本身的陈述更感兴趣（干净的写法当然重要，但那不属于算法和数据结构的知识）。

算法分析是基于算法使用的资源量来进行比较的。说一个算法比另一个算法好就在于它在使用资源方面更有效率，或者说使用了更少的资源。从这个角度来看，上面两个函数看起来很相似，它们都使用基本相同的算法来解决求和问题。在资源计算这点上，重要的是要找准真正用于计算的资源。评价算法使用资源往往要从时间和空间两方面来看。

算法使用的空间指的是内存消耗，解决方案所需的内存通常由问题本身的规模和性质决定。但有时，部分算法会有一些特殊的空间需求，这种情况下需要非常仔细地审查。

算法使用的时间就是指算法执行所有步骤经过的时间，这种评价方式也被称为算法的执行时间，可以通过基准分析来测量函数 sum_of_n 的执行时间。

在 Rust 中，可以记录函数运行前后的系统时间来计算代码运行时间。在 std::time 中有获取系统时间的 SystemTime 函数，它可在被调用时返回系统时间并在之后给出经过的时间。通过在开始和结束的时候调用该函数，就可以得到函数执行时间。

```
// static_func_call.rs
use std::time::SystemTime;
fn sum_of_n(n: i64) -> i64 {
    let mut sum: i64 = 0;
    for i in 1..=n {
        sum += i;
    }
    sum
}
```

```
10
11 fn main() {
12     for _i in 0..5 {
13         let now = SystemTime::now();
14         let _sum = sum_of_n(500000);
15         let duration = now.elapsed().unwrap();
16         let time = duration.as_millis();
17         println!("func used {time} ms");
18     }
19 }
```

执行这个函数 5 次，每次计算前 500,000 个整数的和，得到了如下结果:

func used 10 ms

func used 6 ms

func used 6 ms

func used 6 ms

func used 6 ms

我们发现时间是相当一致的，执行这个函数平均需要 6 毫秒。第一执行耗时 10 毫秒是因为函数要初始化准备，而后面四次执行不需要，此时执行得到的耗时才是比较准确的，这也是为什么需要执行多次。如果我们运行计算前 1,000,000 个整数的和，其结果如下:

func used 17 ms

func used 12 ms

func used 12 ms

func used 12 ms

func used 12 ms

可以看到，第一次的耗时还是更长，后面的时间都一样，且恰好是计算前 500,000 个整数耗时的二倍，这说明算法的执行时间和计算规模成正比。现在考虑如下的函数，它也是计算前 n 个整数的和，只是不同于上一个 sum_of_n 函数的思路，它利用数学公式 $\sum_{i=0}^{n} = \frac{n(n+1)}{2}$ 来计算，效率更高。

```
1 // static_func_call2.rs
2 fn sum_of_n2(n: i64) -> i64 {
3     n * (n + 1) / 2
4 }
```

修改 static_func_call.rs 中的 sum_of_n 函数，然后再做同样的基准测试，使用 3 个不同的 n （100,000、500,000、1,000,000），每个计算 5 次取均值，得到了如下结果:

func used 1396 ns

func used 1313 ns

func used 1341 ns

在这个输出中有两点要重点关注，首先上面记录的执行时间是纳秒，比之前任何例子都短，这 3 个计算时间都在 0.0013 毫秒左右，和上面的 6 毫秒可是差着几个数量级。其次是执行时间和 n 无关，n 增大了，但计算时间不变，看起来此时计算几乎不受 n 的影响。

这个基准测试告诉我们，使用迭代的解决方案 sum_of_n 做了更多的工作，因为一些程序步骤被重复执行，这是它需要更长时间的原因。此外，迭代方案执行所需时间随着 n 递增。另外要意识到，如果在不同计算机上或者使用不用的编程语言运行类似函数，得到的结果也不同，你的电脑计算值可能不是 1341 纳秒（我使用的电脑是联想拯救者 R7000P，CPU 是 16 核的 AMD R7-4800H）。如果使用老旧的计算机，则需要更长时间才能执行完 sum_of_n2。

我们需要一个更好的方法来描述这些算法的执行时间。基准测试计算的是程序执行的实际时间，但它并不真正地提供给我们一个有用的度量，因为执行时间取决于特定的机器，且毫秒，纳秒间也涉及到数量级转换。我们希望有一个独立于所使用的程序或计算机的度量，这个度量能独立地判断算法，并且可以用于比较不同算法实现的效率，就像加速度这个度量值能明确指出速度每秒的改变值一样。在算法分析领域，大 O 分析法就是一种很好度量方法。

2.3 大 O 分析法

要独立于任何特定程序或计算机来表征算法的效率（复杂度），重要的是要量化算法所需操作的数量。对于先前的求和算法，一个比较好的度量单位是对执行语句进行计数。在 sum_of_n 中，赋值语句的计数为 1 (the_sum = 0 的次数)，the_sum += i 计数为 n。

使用函数 T 表示总的次数：T(n)=1 + n 。参数 n 通常称为问题的规模，T(n) 是解决问题规模为 n 的问题所花费的时间。在上面的求和函数中，使用 n 来表示问题大小是有意义的。我们可以说对 100,000 个整数求和比对 1000 个整数求和的规模大，因此所需时间也更长。我们的目标是表示出算法的执行时间是如何相对问题规模的大小而改变的。

将这种分析技术进一步扩展，确定操作步骤所有数量不如确定 T(n) 最主要的部分来得重要。换句话说，当问题规模变大时，T(n) 函数某些部分的分量会远远超过其他部分。函数的数量级表示随着 n 增加而增加最快的那些部分。数量级通常用大 O 符号表示，写作 O(f(n))，用于表示对计算中的实际步数的近似。函数 f(n) 表示 T(n) 最主要部分。

在上述示例中，T(n) = n + 1。当 n 变大时，常数 1 对于最终结果变得越来越不重要。如果我们找的是 T(n) 的近似值，可以删除 1，运行时间就是 O(T(n)) = O(n + 1) = O(n)。要注意，1 对于 T(n) 肯定是重要的，但当 n 变大时，有没有它 O(n) 近似都是准确的。比如对于 $T(n) = n^3 + 1$，n 为 1 时，T(n) = 2，如果此时舍掉 1 就不合理，因为这样就相当于丢掉了一半的运行时间。但是当 n 等于 10 时，T(n) = 1001，此时 1 已经不重要了，即便舍掉了，T(n) = 1000 依然是一个很准确的指标。

假设对于一些算法，确定的步数是 $T(n) = 5n^2 + 27n + 1005$。当 n 很小时，例如 1 或 2 ，常数 1005 似乎是函数的主要部分。然而，随着 n 变大，n^2 这项变得越来越重要。事实上，当 n 很大时，其他两项在最终结果中所起的作用变得不重要。当 n 变大时，为了近似 T(n)，我们可以忽略其他项，只关注 $5n^2$ 。系数 5 也变得不重要。我们说 T(n) 具有的复杂度数量级为 n^2，或者 $O(n^2)$ 。

但有时算法的复杂度取决于数据的确切值，而不是问题规模的大小。对于这类算法，需要根据最佳情况，最坏情况或平均情况来表征它们的性能。最坏情况是指导致算法性能特别差的特定数据集。而相同的算法，不同数据集可能具有非常不同的性能。大多数情况下，算法执行效率处在最坏和最优两个极端之间（平均情况）。对于程序员而言，重要的是了解这些区别，使它们不被某一个特定的情况误导。

在学习算法时，一些常见的数量级函数将会反复出现，见下表。为了确定这些函数中哪个是最主要的部分，我们需要看到当 n 变大时它们相互关系如何。

表 2.1: 不同数量级的函数

T(n)	O(1)	O(logn)	O(n)	O(nlogn)	$O(n^2)$	$O(n^3)$	$O(2^n)$
性能	常数	对数	线性	线性对数	平方指数	立方指数	幂指数

下图画出了各种函数的增长情况，当 n 很小时，函数彼此间不能很好的定义。很难判断哪个是主导的。随着 n 的增长，就有一个很明确的关系，很容易看出它们之间的大小关系。注意在 $n = 10$ 时，2^n 会大于 n^3，图中没有画出完整的情况。

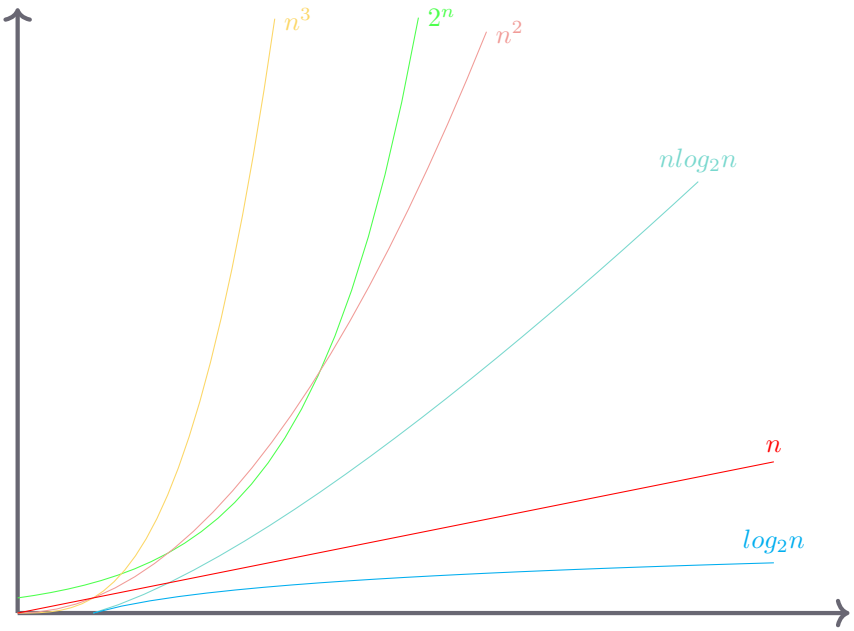


图 2.1: 复杂度曲线

通过上图可以得出这些不同数量级的区别，在一般情况下（$n > 10$），存在 $O(2^n) > O(n^3) > O(n^2) > O(nlogn) > O(n) > O(logn) > O(1)$。这对于我们设计算法很有帮助，因为对于每个算法，我们都能计算其复杂度，如果得到类似 $O(2^n)$ 复杂度的算法，我们知道这一定不实用，然后可以对其进行优化以取得更好的性能。

下面的代码只做了些加减乘除，但我们可以用它来试试新学的算法复杂度分析。

```
fn main() {
    let a = 1;
    let b = 2;
    let c = 3;
    let n = 1000000;

    for i in 0..n {
        for j in 0..n {
            let x = i * i;
            let y = j * j;
            let z = i * j;
         }
    }

    for k in 0..n {
        let w = a*b + 45;
        let v = b*b;
    }

    let d = 33;
}
```

分析上述代码的 T(n)，分配操作数 a，b，c，n 的时间为常数 4。第二项是 $3n^2$，因为嵌套迭代，有三个语句执行 n^2 次。第三项是 2n, 有两个语句迭代执行了 n 次。最后，第四项是常数 1，表示最终赋值语句 d = 33。最后得出 $T(n) = 4 + 3n^2 + 2n + 1 = 3n^2 + 2n + 5$，查看指数，可以看到 n^2 项是最显著的，因此这个代码段是 O(n^2)。当 n 增大时，其他项及主项上的系数都可以忽略。

2.4 乱序字符串检查

一个展示不同数量级复杂度的例子是乱序字符串检查算法。乱序字符串是指一个字符串只是另一个字符串的重新排列。例如，'heart' 和'earth' 就是乱序字符串，'rust' 和'trus' 也是。为简单起见，假设所讨论的两个字符串具有相等的长度，且都由 26 个小写字母集合组成。我们的目标是写一个函数，它将两个字符串做为参数并返回它们是不是乱序字符串。

2.4.1 穷举法

解决乱序字符串最笨的方法是暴力穷举法，把每种情况都列举出来。首先可以生成 s1 的所有乱序字符串列表，然后查看这些列表里是否有一个和 s2 相同。这种方法特别费资源，即

费时间又费内存。当 s1 生成所有可能的字符串时，第一个位置有 n 种可能，第二个位置有 n-1 种，第三个位置有 n-3 种...。总数为 $n \times (n-1) \times (n-2)...3 \times 2 \times 1$，即 n!。虽然一些字符串可能是重复的，程序也不可能提前知道，所以会生成 n! 个字符串。

如果 s1 有 20 个字符长，则将有 20! = 2,432,902,008,176,640,000 个字符串产生。如果每秒处理一种可能的字符串，那么需要 77,146,816,596 年才能过完整个列表。事实证明，n! 比 n^2 增长还快，所以暴力穷举法这种解决方案是不行的，在任何学习及真正的软件项目里都不可能使用这种方法，当然，了解它的存在并尽力避免这种情况却是很有必要的。

2.4.2 检查法

乱序字符串问题的第二种解法是检查第一个字符串中的字符是否出现在第二个字符串中。如果检测到每个字符都存在，那这两个字符串一定是乱序。可以通过用' ' 替换字符来了解一个字符是否完成检查。详细代码请参见源码中 anagram_solution2.rs。

```
1  // anagram_solution2.rs
2
3  fn anagram_solution2(s1: &str, s2: &str) -> bool {
4      // 字符串长度不同，一定不是乱序字符串
5      if s1.len() != s2.len() { return false; }
6
7      // s1 和 s2 中的字符分别加入 alist, blist
8      let mut alist = Vec::new();
9      let mut blist = Vec::new();
10     for c in s1.chars() { alist.push(c); }
11     for c in s2.chars() { blist.push(c); }
12
13     let mut pos1: usize = 0; // pos1、pos2 索引字符
14     let mut ok = true; // 乱序字符串标示、控制循环进程
15     while pos1 < s1.len() && ok {
16         let mut pos2: usize = 0;
17
18         // found 标示字符是否在 s2 中
19         let mut found = false;
20         while pos2 < blist.len() && !found {
21             if alist[pos1] == blist[pos2] {
22                 found = true;
23             } else {
24                 pos2 += 1;
25             }
26         }
```

```
27
28          // 某字符存在于 s2 中，将其替换成 ' ' 避免再次比较
29          if found {
30              blist[pos2] = ' ';
31          } else {
32              ok = false;
33          }
34
35          // 处理 s1 中的下一个字符
36          pos1 += 1;
37      }
38
39      ok
40  }
41
42  fn main() {
43      let s1 = "rust";
44      let s2 = "trus";
45      let result: bool = anagram_solution2(&s1, &s2);
46      println!("s1 and s2 is anagram: {result}");
47  }
```

分析这个算法，注意到 s1 的每个字符都会在 s2 中进行最多 n 次迭代检查。blist 中的 n 个位置将被访问一次以匹配来自 s1 的字符。总的访问次数可以写成 1 到 n 的整数和。

$$\begin{aligned}\sum_{i=1}^{n} i &= \frac{n(n+1)}{2} \\ &= \frac{1}{2}n^2 + \frac{1}{2}n\end{aligned} \tag{2.1}$$

当 n 变大，n^2 这项占据主导，1/2 可以忽略。所以这个算法复杂度为 $O(n^2)$。暴力法是 n!，而检查法一下就降到了 $O(n^2)$

2.4.3 排序和比较法

乱序字符串的另一个解决方案是利用这样一个事实：即使 s1，s2 不同，它们都是由完全相同的字符组成的。所以可以按照字母顺序从 a 到 z 排列每个字符串，如果排列后的两个字符串相同，那这两个字符串就是乱序字符串。

```
1  // anagram_solution3.rs
2
3  fn anagram_solution3(s1: &str, s2: &str) -> bool {
```

```
4       if s1.len() != s2.len() { return false; }
5
6       // s1 和 s2 中的字符分别加入 alist、blist 并排序
7       let mut alist = Vec::new();
8       let mut blist = Vec::new();
9       for c in s1.chars() { alist.push(c); }
10      for c in s2.chars() { blist.push(c); }
11      alist.sort(); blist.sort();
12
13      // 逐个比较排序的集合，任何字符不匹配就退出循环
14      let mut pos: usize = 0;
15      let mut matched = true;
16      while pos < alist.len() && matched {
17          if alist[pos] == blist[pos] {
18              pos += 1;
19          } else {
20              matched = false;
21          }
22      }
23
24      matched
25  }
26
27  fn main() {
28      let s1 = "rust";
29      let s2 = "trus";
30      let result: bool = anagram_solution3(s1, s2);
31      println!("s1 and s2 is anagram: {result}");
32  }
```

乍一看，只有一个 while 循环，所以应该是 O(n)。可调用排序函数 sort() 也是有成本的。其复杂度通常是 $O(n^2)$ 或 $O(nlogn)$，所以该算法复杂度跟排序算法属于同样数量级。

2.4.4 计数和比较法

上面的方法中，总是要创建 alist 和 blist，这非常费内存。当 s1 和 s2 比较短时，alist 和 blist 还算合适，但若是 s1 和 s2 达到百万字符呢？这时 alist 和 blist 就非常大。通过分析可知，s1 和 s2 只含 26 个小写字母，所以用两个长度为 26 的列表，统计各个字符出现的频次就可以了。每遇到一个字符，就增加该字符在列表对应位置的计数。最后如果两个列表

计数一样，则字符串为乱序字符串。

```rust
// anagram_solution4.rs

fn anagram_solution4(s1: &str, s2: &str) -> bool {
    if s1.len() != s2.len() { return false; }

    // 大小为 26 的集合，用于将字符映射为 ASCII 值
    let mut c1 = [0; 26];
    let mut c2 = [0; 26];
    for c in s1.chars() {
        let pos = (c as usize) - 97; // 97 为 a 的 ASCII 值
        c1[pos] += 1;
    }
    for c in s2.chars() {
        let pos = (c as usize) - 97;
        c2[pos] += 1;
    }

    // 逐个比较 ascii 值
    let mut pos = 0;
    let mut ok = true;
    while pos < 26 && ok {
        if c1[pos] == c2[pos] {
            pos += 1;
        } else {
            ok = false;
        }
    }

    ok
}

fn main() {
    let s1 = "rust";
    let s2 = "trus";
    let result: bool = anagram_solution4(s1, s2);
    println!("s1 and s2 is anagram: {result}");
}
```

这个方案也存在多个迭代，但和前面的解法不一样，首先迭代不是嵌套的，其次第三个迭代，比较两个计数列表，只需要 26 次，因为只有 26 个小写字母。所以 $T(n) = 2n + 26$，即 $O(n)$。这是一个线性复杂度的算法，其空间和时间复杂度都比较优秀。当然，s1 和 s2 也可能比较短，用不到 26 个字符，这样该算法也牺牲了部分存储空间。很多情况下，你需要在空间和时间之间做出权衡，要思考你的算法应对的真实场景，然后再决定采用哪种算法。

2.5 Rust 数据结构的性能

2.5.1 标量和复合类型

本节的目标是探讨 Rust 内置的各种基本数据类性的大 O 性能，重要的是了解这些数据结构的效率，因为它们是 Rust 中最基础和最核心的模块，其他所有复杂的数据结构都由它们构建而成。

在 Rust 中，每个值都属于某一数据类型，这告诉 Rust 编译器它被指定为何种数据，以便明确数据存储和操作方式。Rust 里面有两大类基础数据类型：标量和复合类型。

标量类型代表一个单独的值，复合类型是标量类型的组合。Rust 中有四种基本的标量类型：整型、浮点型、布尔型、字符型；有两种复合类型：元组、数组。标量类型都是最基本的和内存结合最紧密的原生类型，运算效率非常高，可以视为 O(1)，而复合类型则复杂一些，复杂度随其数据规模而变化。

```
fn main() {
    let a: i8 = -2;
    let b: f32 = 2.34;
    let c: bool = true;
    let d: char = 'a';

    let x: (i32, f64, u8) = (200, 5.32, 1);
    let xi32 = x.0;
    let xf64 = x.1;
    let xu8  = x.2;
}
```

元组是将多个各种类型的值组合成一个复合类型的数据结构。元组长度固定，一旦声明，其长度不能增大或缩小。元组的索引从 0 开始，直接用 “.” 号获取值。

数组一旦声明，长度也不能增减，但与元组不同的是，数组中每个元素的类型必须相同。数组非常有用，基于数组的切片其实更灵活，所以也更常用。

```
fn main() {
    let months = ["January", "February", "March", "April",
                  "May", "June", "July", "August", "September",
                  "October", "November", "December"];
```

```
5       let first_month = months[0]
6       let halfyear = &months[..6];
7
8       let mut monthsv = Vec::new();
9       for month in months { monthsv.push(month); }
10  }
```

Rust 里的其他数据类型都是由标量和复合类型构成的集合类型，如 Vec、HashMap 等。Vec 类型是标准库提供的一个允许增加和缩小长度的类似数组的集合类型。能用数组的地方都可用 Vec，所以当不知道该用哪个时，用 Vec 不算错，而且还更有扩展性。

2.5.2 集合类型

Rust 的集合类型是基于标量和复合类型构造的，其中又分为线性和非线性两类。线性的集合类型有：String、Vec、VecDeque、LinkedList，而非线性集合类型的有：HashMap、BTreeMap、HashSet、BTreeSet、BinaryHeap。这些线性和非线性集合类型多涉及到索引、增、删操作，对应的复杂度多是 O(1)、O(n) 等。具体性能如以下两表。由表可见 Rust 实现的集合数据类型都是非常高效的，复杂度最高也就是 O(n)。

表 2.2: 线性集合类型的性能

Type	get	insert	remove	append	split_off
Vec	O(1)	O(n-i)	O(n-i)	O(m)	O(n-i)
VecDeque	O(1)	O(min(i,n-i))	O(min(i,n-i))	O(m)	O(min(i,n-i))
LinkedList	O(min(i,n-i))	O(min(i,n-i))	O(min(i,n-i))	O(1)	O(min(i,n-i))

表 2.3: 非线性集合类型的性能

Type	get	insert	remove	predecessor	append
HashMap	O(1)	O(1)	O(1)	N/A	N/A
HashSet	O(1)	O(1)	O(1)	N/A	N/A
BTreeMap	O(logn)	O(logn)	O(logn)	O(logn)	O(n+m)
BTreeSet	O(logn)	O(logn)	O(logn)	O(logn)	O(n+m)
Type	push	pop	peek	peek_mut	append
BinaryHeap	O(1)	O(1ogn)	O(1)	O(1)	O(n+m)

Rust 实现的 String 底层基于数组，所以 String 同数组一样不可更改。若要使用字符串中部分字符则可使用 &str，&str 实际是基于 String 数组的切片，便于索引。Vec 则类似于 Python 中的列表，基于分配在堆上的数组。VecDeque 扩展了 Vec，支持在序列两端插入数据，所以是一个双端队列。LinkedList 是链表，当需要一个未知大小的 Vec 时可以采用。

Rust 实现的 HashMap 类似于 Python 的字典，BTreeMap 则是 B 树，其节点上包含数据和指针，多用于实现数据库，文件系统等需要存储内容的地方。Rust 实现的 HashSet 和 BTreeSet 类似于 Python 的集合，都用于记录出现过的值。HashSet 底层采用的是 HashMap，而 BTreeSet 底层则采用的 BTreeMap。BinaryHeap 类似优先队列，存储一堆元素，可在任何时候提取出最大的值。

2.6 总结

本章学习了算法复杂度分析的大 O 分析法：计算代码执行的步数，并取其最大数量级。接着学习了 Rust 实现的基本数据类型和集合数据类型的复杂度，通过对比学习，可知 Rust 内置的标量、复合以及集合数据类型都非常高效，基于这些集合类型来实现自定义的数据结构也就更容易做到高效实用。

Chapter 3

基本数据结构

3.1 本章目标

理解抽象数据类型 Vec、栈、队列、双端队列
能够使用 Rust 实现堆栈、队列、双端队列
了解基本线性数据结构实现的性能
了解前缀、中缀和后缀表达式格式
使用栈来实现后缀表达式并计算值
使用栈将中缀表达式转换为后缀表达式
能够识别问题该使用栈、队列还是双端队列
能够使用节点和引用将抽象数据类型实现为链表
能够比较自己实现链表与 Rust 自带的 Vec 的性能

3.2 线性数据结构

数组、栈、队列、双端队列这一类数据结构都是保存数据的容器，数据项之间的顺序由添加或删除的顺序决定，一旦数据项被添加，它相对于前后元素一直保持位置不变，诸如此类的数据结构被称为线性数据结构。线性数据结构有两端，被称为“左”和“右”，某些情况也称为“前”和“后”，当然，也可以称为顶部和底部，具体的名字不重要，重要的是这种命名展现出的位置关系表明了数据组织方式就是线性的。这种线性特性和内存紧密相关，因为内存便是一种线性硬件，由此也可以看出软件和硬件是如何关联在一起的。

区分不同线性数据结构的方法是看其添加和移除数据项的方式，特别是添加和移除项的位置。例如一些数据结构只允许从一端添加项，另一些则允许从另一端移除项，还有的数据结构允许两端操作数据项。这些变种及其组合形式产生了许多计算机科学领域最有用的数据结构，它们出现在各种算法中，并用于执行各种实际且重要的任务。

3.3 栈

栈就是一种特别有用的数据结构，可用于函数调用、网页数据记录等。栈是一个项的有序集合，其中添加移除新项总发生在同一端，这一端称为顶部，与之相对的端称为底部。栈的底部很重要，因为在栈中靠近底部的项是存储时间最长的，最近添加的项是会最先被移除的。这种排序原则有时被称为后进先出（Last In Fisrt Out，LIFO）或者先进后出（Fisrt In Last Out, FILO），所以较新的项靠近顶部，较旧的项靠近底部。

栈的例子很常见，工地堆的砖，桌上的书堆，餐厅堆叠的盘子都是栈的物理模型。要拿到最下面的砖、书、盘子，只有先把上面的一个个拿走。下图是栈的示意图，其中保存着一些概念名称（计算机是量子力学理论的衍生物）。

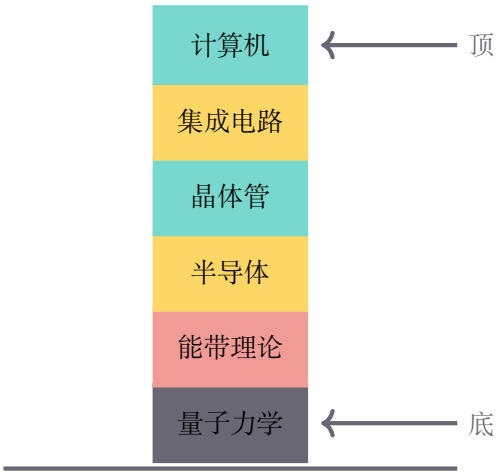


图 3.1: 栈

要理解栈的作用，最好的方式是观察栈的形成和清空。假设从一个干净的桌面开始，现在把书一本本叠起来，你就在构造一个栈。考虑下移除一本书会发生什么。移除的顺序跟刚刚放置的顺序相反。栈之所以重要是因为它能反转项的顺序，插入跟删除顺序相反。下图展现了数据对象创建和删除的过程，注意观察数据的顺序。

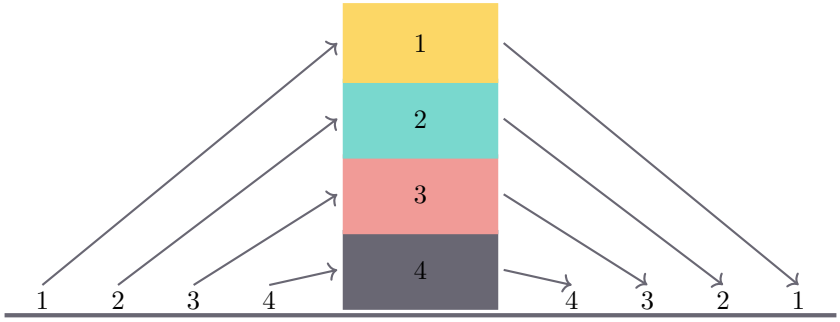


图 3.2: 栈逆反数据

这种反转的属性特别有用，你可以想到使用计算机的时候所碰到的例子。例如，每个浏览器都有返回按钮，当你浏览网页时，这些网页就被放在一个栈中，你当前查看的网页始终在顶部，第一次查看的网页在底部。如果你按返回键，将按相反的顺序回到刚才的页面。由这个例子也可看出数据结构的重要性，对某些功能，选好数据结构，能让事情简单不少。

3.3.1 栈的抽象数据类型

栈的抽象数据类型由其结构和操作定义。如上所述，栈被构造为项的有序集合，其中项被添加和移除的位置称为顶部。栈的一些操作如下。

new() 创建一个空栈，它不需要参数，返回一个空栈。

push(item) 将数据项 item 添加到栈顶，它需要 item 做参数，不返回任何内容。

pop() 从栈中删除顶部数据项，它不需要参数，返回数据项，栈被修改。

peek() 从栈返回顶部数据项，但不会删除它，不需要参数，不修改栈。

is_empty() 测试栈是否为空，不需要参数，返回布尔值。

size() 返回栈中数据项的数量，不需要参数，返回一个 usize 型整数。

假设 s 是已创建的空栈，此处用 [] 表示，下表展示了栈操作后的结果，栈顶在右边。

表 3.1: 栈操作及结果

栈操作	栈当前值	操作返回值
s.is_empty()	[]	true
s.push(1)	[1]	
s.push(2)	[1,2]	
s.peek()	[1,2]	2
s.push(3)	[1,2,3]	
s.size()	[1,2,3]	3
s.is_empty()	[1,2,3]	false
s.push(4)	[1,2,3,4]	
s.push(5)	[1,2,3,4,5]	
s.size()	[1,2,3,4,5]	5
s.pop()	[1,2,3,4]	5
s.pop()	[1,2,3]	4
s.size()	[1,2,3]	3

3.3.2 Rust 实现栈

前文已定义了栈的抽象数据类型，现在使用 Rust 来实现栈，抽象数据类型的实现又称为数据结构。在 Rust 中，抽象数据类型的实现多选择创建新的结构体 struct，栈操作实现为结构体的函数。此外，为了实现作为元素集合的栈，使用由 Rust 自带的基本数据结构对实现栈及其操作大有帮助。

这里使用 Vec 这种集合容器来作为栈的底层实现，因为 Rust 中的 Vec 提供了有序集合机制和一组操作方法，只需要选定 Vec 的哪一端是栈顶部就可以实现其他操作了。以下栈实现假定 Vec 的尾部将保存栈的顶部元素，随着栈增长，新项将被添加到 Vec 末尾，因为不知道插入数据类型，所以采用了泛型数据类型 T。

```
// stack.rs

#[derive(Debug)]
struct Stack<T> {
    top: usize,      // 栈顶
    data: Vec<T>,    // 栈数据容器
}

impl<T> Stack<T> {
    fn new() -> Self {
        Stack {
            top: 0,
            data: Vec::new()
        }
    }

    fn push(&mut self, val: T) {
        self.data.push(val); // 数据保存在 Vec 末尾
        self.top += 1;
    }

    fn pop(&mut self) -> Option<T> {
        if self.top == 0 { return None; }
        self.top -= 1; // 栈顶减 1 后再弹出数据
        self.data.pop()
    }

    fn peek(&self) -> Option<&T> { // 数据不能移动，只能返回引用
        if self.top == 0 { return None; }
        self.data.get(self.top - 1)
    }

    fn is_empty(&self) -> bool {
        0 == self.top
```

```
35      }
36
37      fn size(&self) -> usize {
38          self.top // 栈顶恰好就是栈中元素个数
39      }
40  }
41
42  fn main() {
43      let mut s = Stack::new();
44      s.push(1); s.push(2); s.push(4);
45      println!("top {:?}, size {}",s.peek().unwrap(), s.size());
46      println!("pop {:?}, size {}",s.pop().unwrap(), s.size());
47      println!("is_empty:{}, stack:{:?}", s.is_empty(), s);
48  }
```

3.3.3 括号匹配

上面实现了栈数据结构，下面利用栈来解决真正的计算机问题。第一个是括号匹配问题，任何人都见过如下这种计算值的算术表达式。

$$(5+6)\times(7+8)/(4+3)$$

Lisp 语言是一种非常依赖括号的语言，比如下面这个 multiply 函数。

```
1  (defun multiply(n)
2      (* n n))
```

这里关注点不在数字而在括号，因为括号更改了操作优先级，限定了语言的语义，非常重要。若括号不完整，那么整个式子就是错的。这对于人来说是再好懂不过了，可是计算机又如何知道呢？可见，计算机必然检测了括号是否匹配，并根据情况报错。

上面这两个例子中，括号都必须以成对匹配的形式出现。括号匹配意味着每个开始符号具有相应的结束符号，并且括号正确嵌套，这样计算机才能正确处理。

考虑下面正确匹配的括号字符串:

(()()()())

((((()))))

(()((())()))

以及这些不匹配的括号:

(((((((()

()))

(()()((

这些括号表达式省去了包含的具体值，只保留了括号自身，其实程序中经常出现这种括号及嵌套的情况。

区分括号匹配对于计算机程序来说是十分重要的，只有这样才能决定下一步操作。具有挑战的是如何编写一个算法能够从左到右读取一串符号，并决定括号是否平衡。要解决这个问题，需要对括号及其匹配有比较深入地了解。从左到右处理符号时，最近的左开始括号‘(’必须与下一个右关闭符号‘)’相匹配（如图3.3)。此外，处理的第一个左开始括号必须等待直到其匹配最后一个右关闭括号。结束括号以相反的顺序匹配开始括号，从内到外匹配，这是一个可以用栈来解决的问题。

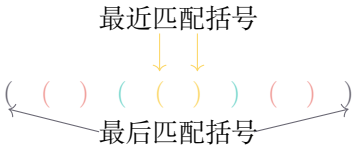


图 3.3: 括号匹配

一旦采用栈来保存括号，则算法的具体实现就很简单了，因为栈的操作无非就是出入栈和判断而已。从空栈开始，从左到右处理括号字符串。如果一个符号是一个左开始符号，将其入栈，如果是结束符号，则弹出栈顶元素，并开始匹配这两个符号。如果恰好是左右匹配的，那么就继续处理下一个括号，直到字符串处理完成。最后，当所有符号都被处理后，栈应该是空的。只要不为空，就说明有括号不匹配。下面是 Rust 实现的括号匹配程序。

```
1  // par_checker1.rs
2
3  fn par_checker1(par: &str) -> bool {
4      let mut char_list = Vec::new();
5      for c in par.chars() {
6          char_list.push(c);
7      }
8
9      let mut index = 0;
10     let mut balance = true; // 括号是否匹配(平衡)标示
11     let mut stack = Stack::new(); // 使用前面实现的栈
12     while index < char_list.len() && balance {
13         let c = char_list[index];
14
15         if '(' == c { // 如果为开符号，入栈
16             stack.push(c);
17         } else { // 如果为闭符号，判断栈是否为空
18             if stack.is_empty() {
```

```
19                  balance = false; // 为空则不平衡
20              } else {
21                  let _r = stack.pop();
22              }
23          }
24          index += 1;
25      }
26
27      // 平衡且栈为空，括号表达式才是匹配的
28      balance && stack.is_empty()
29  }
30
31  fn main() {
32      let sa = "()(())";
33      let sb = "()(()(";
34      let res1 = par_checker1(sa);
35      let res2 = par_checker1(sb);
36      println!("sa balanced:{res1}, sb balanced:{res2}");
37  }
```

当然，括号不仅限于 ()，还有 {} 及 []，下文将这三种都称为括号。上面显示的匹配括号问题非常简单，只有 (和) 这两个符号，但其实常用的括号有三种，分别是 ()、[]、{}。匹配和嵌套不同种类的左开始和右结束括号的情况经常发生。例如，在 Rust 中，方括号 [] 用于索引，花括号 {} 用于格式化输出，() 用于函数参数，元组，数学表达式等。只要每个符号都能保持自己的左开始和右结束关系，就可以混合嵌套符号。

{{([][])}()}
[[{{(())}}]]
[][][]({})

上面这些括号都匹配，每个开始符号都有对应的结束符号，而且符号类型也匹配。相反下面这些字符串符号就不匹配。

(}[]
(((()]))
[{()]

前面的那个括号检查程序 par_checker1 只能检测 ()，要处理三种括号，需要对其进行扩展。算法流程依旧不变，每个左开始括号被压入栈中，等待匹配的右结束括号出现。出现结束括号时，此时要做的是检查括号的类型是否匹配。如果两个括号不匹配，则字符串不匹配。如果整个字符串都被处理完并且栈为空，则字符串括号匹配。

为了检查括号类型是否匹配，我们新增了一个符号类型检测函数 par_match()，它可以同时检测常用的三种括号。

```rust
// par_checker2.rs

// 同时检测多种开闭符号是否匹配
fn par_match(open: char, close: char) -> bool {
    let opens = "([{";
    let closers = ")]}";
    opens.find(open) == closers.find(close)
}

fn par_checker2(par: &str) -> bool {
    let mut char_list = Vec::new();
    for c in par.chars() {
        char_list.push(c);
    }

    let mut index = 0;
    let mut balance = true;
    let mut stack = Stack::new();
    while index < char_list.len() && balance {
        let c = char_list[index];

        // 同时判断三种开符号
        if '(' == c || '[' == c || '{' == c {
            stack.push(c);
        } else {
            if stack.is_empty() {
                balance = false;
            } else {
                // 比较当前括号和栈顶括号是否匹配
                let top = stack.pop().unwrap();
                if !par_match(top, c) {
                    balance = false;
                }
            }
        }
        index += 1;
    }

```

```
39      balance && stack.is_empty()
40  }
41
42  fn main() {
43      let sa = "(){}[]";
44      let sb = "(){)[}";
45      let res1 = par_checker2(sa);
46      let res2 = par_checker2(sb);
47      println!("sa balanced:{res1}, sb balanced:{res2}");
48  }
```

现在我们的代码能处理多种括号匹配的问题，但是如果出现像下面这种字符串，其中含有其他字符，那么上面的程序就又不能处理了。

(a+b)(c*d)func()

这个问题看起来复杂，因为似乎要处理各种字符。但实际上，问题还是括号匹配检测，所以非括号不用处理，直接跳过。程序处理字符时，上面的字符串中非括号自动忽略，只剩下括号，相当于字符串：()()()。可见问题和原来一样，只需修改部分代码就能检测包含任意字符的字符串是否匹配。下面的代码是修改 par_checker2.rs 后得到的 par_checker3.rs。

```
1   // par_checker3.rs
2
3   fn par_checker3(par: &str) -> bool {
4       let mut char_list = Vec::new();
5       for c in par.chars() {
6           char_list.push(c);
7       }
8
9       let mut index = 0;
10      let mut balance = true;
11      let mut stack = Stack::new();
12      while index < char_list.len() && balance {
13          let c = char_list[index];
14
15          // 开符号入栈
16          if '(' == c || '[' == c || '{' == c {
17              stack.push(c);
18          }
19
20          // 闭符号则判断是否平衡
```

```
21          if ')' == c || ']' == c || '}' == c {
22              if stack.is_empty() {
23                  balance = false;
24              } else {
25                  let top = stack.pop().unwrap();
26                  if !par_match(top, c) {
27                      balance = false;
28                  }
29              }
30          }
31
32          // 非括号直接跳过
33          index += 1;
34      }
35
36      balance && stack.is_empty()
37  }
38
39  fn main() {
40      let sa = "(2+3){func}[abc]";
41      let sb = "(2+3)*(3-1";
42      let res1 = par_checker3(sa);
43      let res2 = par_checker3(sb);
44      println!("sa balanced:{res1}, sb balanced:{res2}");
45  }
```

这个例子表明栈是重要的数据结构，任何嵌套符号匹配问题都可以用栈处理。

3.3.4 进制转换

对你来说二进制应该很熟悉。二进制是计算机世界的底座，是计算机世界底层真正通用的数据格式，因为存储在计算机内的所有值都是以 0 和 1 的电压形式存储的。如果没有能力在二进制数和普通字符之间转换，与计算机之间的交互将非常困难。整数值是常见的数据形式，一直用于计算机程序和计算。我们在数学课上学习过，当然是学的十进制表示。十进制 233(10) 以及对应的二进制表示 11101001(2) 分别解释为：

$$\begin{aligned} 2\times 10^2+3\times 10^1+3\times 10^0 &= 233 \\ 1\times 2^7+1\times 2^6+1\times 2^5+0\times 2^4+1\times 2^3+0\times 2^2+0\times 2^1+1\times 2^0 &= 233 \end{aligned} \tag{3.1}$$

将整数值转换为二进制最简单的方法是“除 2”算法，它用栈来跟踪二进制结果的数字。

除 2 算法假定从大于 0 的整数开始。不断迭代地将十进制数除以 2，并跟踪余数。第一个除以 2 的余数说明了这个值是偶数还是奇数。偶数的余数为 0，奇数余数为 1。在迭代除 2 的过程中将这些余数记录下来，就得到了二进制数字序列，第一个余数实际上是序列中的最后一个数字。见下图，数字是反转的，第一次除法得到的余数放在栈底，出栈所有数字得到的表示就是原十进制数字的二进制表示。很明显，栈是解决这个问题的关键。

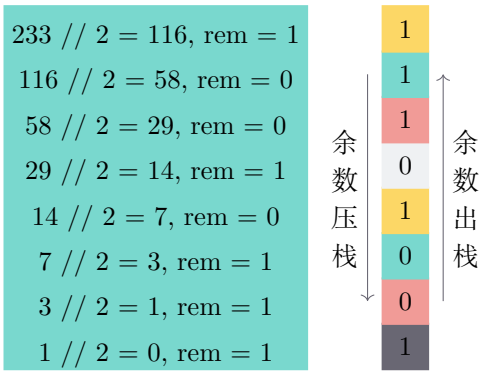


图 3.4: 除二法

下面的 Rust 代码实现了除 2 算法，函数 divideBy2 传入一个十进制的参数，并重复除以 2。第 9 行使用内置的模运算符% 来提取余数，第 10 行将余数加入栈中。当除到 0 后程序结束并构造返回一个二进制字符串。

```
1  // divide_by_two.rs
2
3  fn divide_by_two(mut dec_num: u32) -> String {
4      // 用栈来保存余数 rem
5      let mut rem_stack = Stack::new();
6
7      // 余数 rem 入栈
8      while dec_num > 0 {
9          let rem = dec_num % 2;
10         rem_stack.push(rem);
11         dec_num /= 2;
12     }
13
14     // 栈中元素出栈组成字符串
15     let mut bin_str = "".to_string();
16     while !rem_stack.is_empty() {
17         let rem = rem_stack.pop().unwrap().to_string();
18         bin_str += &rem;
```

```
19      }
20
21      bin_str
22  }
23
24  fn main() {
25      let bin_str: String = divide_by_two(10);
26      println!("10 is b{bin_str}");
27  }
```

这个用于十进制到二进制转换的算法可以很容易地扩展到执行任何进制数间的转换。在计算机科学中，通常会使用不同的进制数，其中最常见的是二进制，八进制和十六进制。十进制 233 对应的八进制为 351(8)，十六进制为 e9(16)。

可以修改 divideBy2 函数，使它不仅能接受十进制参数，还能接受预定转换的基数。除 2 的概念被替换成更通用的除基数。在下面展示的是一个名为 base_converter 函数。采用十进制数和 2 到 16 之间的任何基数作为参数。余数部分仍然入栈，直到被转换的值为 0。有一个问题是，超过 10 的进制，比如 16 进制，它的余数必然会出现大于 10 的数，为了简化字符显示，最好将大于 10 的余数显示为单个字符，我们选择 A-F 分别表示 10-15，当然也可以用小写形式的 a-f，或者其他的字符序列如 u-z，U-Z。

```
1  // base_converter.rs
2
3  fn base_converter(mut dec_num: u32, base: u32) -> String {
4      // digits 对应各种余数的字符形式，尤其是 10 - 15
5      let digits = ['0', '1', '2', '3', '4', '5', '6', '7',
6                    '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'];
7      let mut rem_stack = Stack::new();
8
9      // 余数入栈
10     while dec_num > 0 {
11         let rem = dec_num % base;
12         rem_stack.push(rem);
13         dec_num /= base;
14     }
15
16     // 余数出栈并取对应字符来拼接成字符串
17     let mut base_str = "".to_string();
18     while !rem_stack.is_empty() {
19         let rem = rem_stack.pop().unwrap() as usize;
```

```
20            base_str += &digits[rem].to_string();
21        }
22
23        base_str
24    }
25
26    fn main() {
27        let bin_str: String = base_converter(10, 2);
28        let hex_str: String = base_converter(43, 16);
29        println!("10 is b{bin_str}, 43 is x{hex_str}");
30    }
```

3.3.5 前中后缀表达式

编写一个算术表达式，如 B * C ，表达式形式使你能正确理解它。在这种情况下，你知道是 B 乘 C，操作符是乘法运算符 * 。这种类型的符号称为中缀表达式，因为运算符处于它处理的两个操作数 A 和 B 中间，而且读法 “A 乘 B” 和表达式的顺序一致，自然好理解。

再看另外一个中缀表达式的例子，A + B * C，此时运算符 + 和 * 处于操作数之间。这里的问题是，如何区分运算符分别作用于哪个运算数上呢？到底是 + 作用于 A 和 B，还是 * 作用于 B 和 C？当然，你肯定觉得这很简单，当然是 * 作用于 B 和 C，然后结果再和 A 相加。这是因为你知道每个运算符都有优先级，优先级较高的运算符在优先级较低的运算符之前使用。唯一改变顺序的是括号的存在，算术运算符的优先顺序是将乘法和除法置于加法和减法之前。如果出现具有相等优先级的两个运算符，则使用从左到右的顺序排序或关联。

任何学过基础数学知识的都知道这些。对于中缀表达式 A + B * C，用人脑算的时候，你的眼睛会自动移到后面两个数上，最后再计算加法。实际上，你可能没意识到，自己的大脑已经添加了括号并划分好了计算顺序：(A + (B * C))。

可是，计算机并没有知识，它只能按照规则，按照顺序来处理这种表达式。而我们平时使用计算机计算时并没有出错，说明计算机内部一定有某种规则（算法）使得它能正确计算。

借助上面的思路，似乎保证计算机不会对操作顺序产生混淆的方法就是创建一个完全括号表达式，这种类型的表达式对每个运算符都使用一对括号。括号指示着操作的顺序。可问题是，计算机是从左到右处理数据，类似 (A + (B * C)) 这种完全括号表达式，计算机如何跳到内部括号计算乘法然后再跳到外部括号计算加法呢？人脑能跳着计算是因为人有智能，知道判断，而计算机是死脑筋，对人看起来简单的任务，直接让它处理也是非常困难的。

所以计算机采用的一定是某种不同于人脑计算的模式。完全括号表达式将操作符和操作数混合在一起，这种模式对计算机很困难，所以可以考虑将操作符号移动到操作数外面，将操作符和操作数分开，这样计算机拿到操作符再去取操作数，计算的结果就作为当前值，再参与后面的运算，直到完成整个运算表达式的计算。

可以将中缀表达式 A + B 的 “+” 号移动出来，既可以放前面也可以放后面，所以得到

的分别是 + A B 和 A B + 这样看起来有点儿怪的表达式。这两种不同的表达式分别称为前缀表达式和后缀表达式，同中缀表达式可以区分开来。前缀表达式要求所有运算符在处理的两个操作数之前，后缀表达式则要求其操作符在相应的操作数之后，下面是更多这类表达式的例子。

表 3.2: 前中后缀表达式

中缀表达式	前缀表达式	后缀表达式
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +
(A + B) * C	* + A B C	A B + C *
A + B - C	- + A B C	A B + C -
A * B + C	+ * A B C	A B * C +
A * B / C	/ * A B C	A B * C /

可以看到，这个表达式还挺复杂的，尤其加不加括号，得到表达式很不一样。对于这种表达式需要读者习惯，要能分析并知道正确的写法。

注意到上面第三个中缀表达式里明明有括号，但前缀和后缀表达式中并没有括号。这是因为前缀和后缀表达式已经将计算逻辑表达得很明确了，没有模棱两可的地方，计算机按照这种表达式计算不会出错。只有中缀才需要括号，前缀和后缀表达式的操作顺序完全由操作符的顺序决定，所以不用括号。下表是一些更复杂的表达式。

表 3.3: 复杂的前中后缀表达式

中缀表达式	前缀表达式	后缀表达式
A + B * C - D	- + A * B C D	A B C * + D -
A * B - C / D	- * A B / C D	A B * C D / -
A + B + C + D	+ + + A B C D	A B + C + D +
(A + B) * (C - D)	* + A B - C D	A B + C D - *

有了前缀和后缀表达式，计算看起来更复杂了。比如 A + B * C 的前缀表达式 + A * B C ，这怎么计算呢? A + B * C 要先算 B * C 再计算加法，可是 + A * B C 的乘号还是在内部，仍旧无法计算。要是能将乘号和加号颠倒顺序就好了。前面我们学习过栈具有颠倒顺序的功能，所以可以采用两个栈，一个保存操作符号，一个保存操作数，直接按照从左到右的顺序将其分别入栈，结果如图（3.5)。

计算时先将操作符号出栈，然后将两个操作数出栈，此时用操作符计算这两个操作数，结果再入栈，如图（3.6)。接着再重复这个计算步骤，直到操作符号栈空，此时弹出操作数栈顶数据，这个值就是整个表达式的计算结果。可以看到，这个计算和完全括号表达式计算是一样的，而且计算机不用处理括号，只用出入栈，非常高效。

若是后缀表达式，也用栈来计算，且只用一个栈。比如 A + B * C 的后缀表达式 A B C * +，先将 A B C 入栈，接着发现 * 号，弹出两个操作数 B C，计算得到结果 BC，再将

其入栈，接着遇到 + 号，弹出两个操作数 A 和 BC，计算得到 A + BC。可见不论前缀还是后缀表达式，都能用栈快速地计算出来。

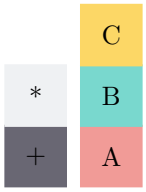


图 3.5: 栈保存表达式

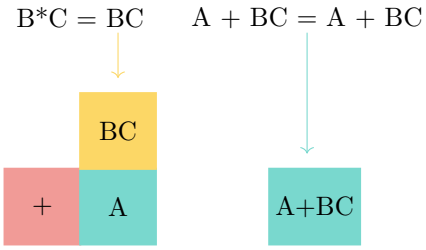


图 3.6: 栈计算表达式

3.3.6 中缀转前后缀表达式

上面的计算过程说明了中缀表达式需要转换为前缀或后缀表达式后计算才高效。所以第一步是，如何得到前后缀表达式？一种方法是采用完全括号表达式。

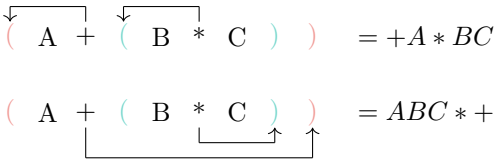


图 3.7: 中缀表达式转前后缀表达式

A + B * C 可写成 (A + (B * C))，表示乘法优先于加法。仔细观察可以看到每个括号对还表示操作数对的开始和结束。(A + (B * C)) 内部表大式是 (B * C)，如将乘法符号移到左括号位置并删除该左括号和配对的右括号，实际上就已经将子表达式转换为了前缀符号。如果加法运算符也被移动到其相应的左括号位置并删除匹配的右括号，则将得到完整的前缀表达式 + A * B C。如将符号向右移动到右括号位置，并删除对应的左括号，则可得到后缀表达式。

为了转换表达式，无论是转换为前缀还是后缀表达式，都要先根据操作的顺序把表达式转换成完全括号表达式。然后再将括号内运算符移动到左或右括号的位置。一个更复杂的例

子是 (A + B) * C - (D + E) / (F + G)，下图显示了如何将其转换为前缀或后缀表达式，对人来说此结果非常复杂，但计算机却能很好地处理。

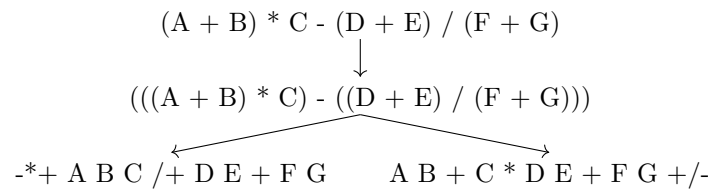


图 3.8: 中缀表达式转前后缀表达式

然而，得到完全括号表达式本身就很困难，而且移动字符再删除字符涉及修改字符串，所以这种方法还不够通用。仔细考察转换的过程，比如 (A + B) * C 转换为 A B + C *，如果不看操作符，则操作数 A B C 还保持着原来的相对位置，只有操作符在改变位置。既然这样，将操作符单独处理更方便。遇到操作数不改变位置，直接保留，遇到操作符才处理。可是操作符有优先级，往往会反转顺序。反转顺序是栈的特点，所以可以用栈来保存操作符。

(A + B) * C 这个中缀表达式，从左到右先看到 + 号，由于括号它的优先级高于 * 号。遇到左括号时，表示高优先级的运算符将出现，所以保存它，该操作符需要等到相应的右括号出现以表示其位置。当右括号出现时，可以从栈中弹出操作符。当从左到右扫描中缀表达式时，用栈来保留运算符，栈顶将始终是最近保存的运算符。每当读取到新的运算符时，需要将其与在栈上的运算符（如果有的话）比较优先级并决定是否弹出。

假设中缀表达式是一个由空格分隔的标记字符串，操作符只有 + - * /，以及左右括号 ()。操作数用字符 A，B，C 表示。下图展示了将 A * B + C * D 转换为后缀表达式的过程，最下面一行是后缀表达式。

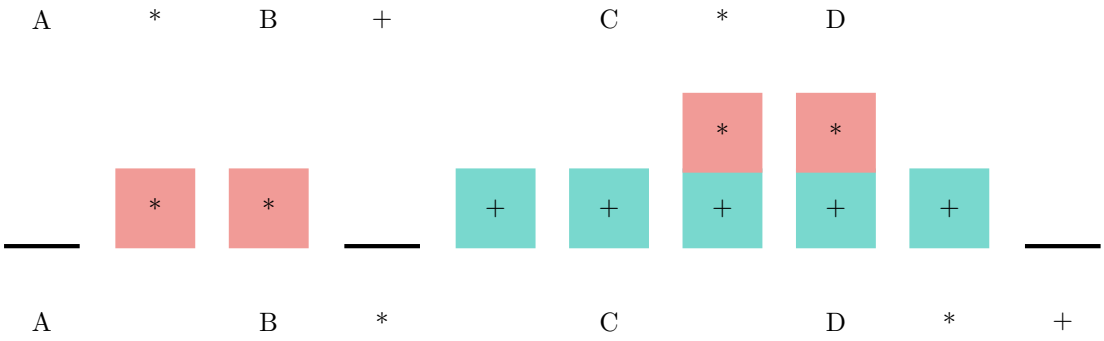


图 3.9: 栈构造后缀表达式

上述转换的具体步骤如下：

1. 创建一个名为 op_stack 的空栈以保存运算符。给输出创建一个空列表 postfix。
2. 通过使用字符串方法拆分将输入的中缀字符串转换为标记列表 src_str。
3. 从左到右扫描标记列表。

如果标记是操作数，将其附加到输出列表的末尾。

如果标记是左括号，将其压到 op_stack 上。

如果标记是右括号，则弹出 op_stack，直到删除相应左括号，将运算符加入 postfix。

如果标记是运算符 + - * /，则压入 op_stack。但先弹出 op_stack 中更高或相等优先级的运算符到 postfix。

4. 当输入处理完后，检查 op_stack，仍在栈上的运算符都可弹出到 postfix。

Algorithm 3.1: 中缀表达式转后缀表达式算法

```
Input: 中缀表达式字符串
Output: 后缀表达式字符串
创建 op_stack 栈保存操作符
创建 postfix 列表保存后缀表达式字符串
将中缀表达式字符串转换为列表 src_str
for c ∈ src_str do
    if c ∈ "A − Z" then
        postfix.append(c)
    else if c =='(' then
        op_stack.push(c)
    else if c ∈ "+ − ∗ /" then
        while op_stack.peek() prior to c do
            postfix.append(op_stack.pop())
        end
        op_stack.push(c)
    else if c ==')' then
        while op_stack.peek() != '(' do
            postfix.append(op_stack.pop())
        end
    end
end
while !op_stack.is_empty() do
    postfix.append(op_stack.pop())
end
return ' '.join(postfix)
```

为了在 Rust 中编写这个后缀表达式转换算法，我们使用了一个名为 prec 的 HashMap 来保存操作符优先级，它将每个运算符映射为一个整数，用于与其他运算符优先级进行比较。左括号赋予最低的优先级，这样，与其进行比较的任何运算符都具有更高的优先级。操作符只有 +-*/，操作数定义为任何大写字符 A-Z 或数字 0-9。第一个 if 判断用于检测括号是否

匹配，是前面已经学习过的内容。

```
// infix_to_postfix.rs

use std::collections::HashMap;

fn infix_to_postfix(infix: &str) -> Option<String> {
    // 括号匹配检验
    if !par_checker3(infix) { return None; }

    // 设置各个符号的优先级
    let mut prec = HashMap::new();
    prec.insert("(", 1); prec.insert(")", 1);
    prec.insert("+", 2); prec.insert("-", 2);
    prec.insert("*", 3); prec.insert("/", 3);

    // ops 保存操作符号、postfix 保存后缀表达式
    let mut op_stack = Stack::new();
    let mut postfix = Vec::new();
    for token in infix.split_whitespace() {
        // 0 - 9  和 A-Z 范围字符入栈
        if ("A" <= token && token <= "Z") ||
            ("0" <= token && token <= "9") {
            postfix.push(token);
        } else if "(" == token  {
            // 遇到开符号，将操作符入栈
            op_stack.push(token);
        } else if ")" == token  {
            // 遇到闭符号，将操作数入栈
            let mut top = op_stack.pop().unwrap();
            while top != "(" {
                postfix.push(top);
                top = op_stack.pop().unwrap();
            }
        } else {
            // 比较符号优先级来决定操作符是否加入 postfix
            while (!op_stack.is_empty()) &&
                (prec[op_stack.peek().unwrap()] >= prec[token]) {
                postfix.push(op_stack.pop().unwrap());
```

```
38                 }
39
40                 op_stack.push(token);
41             }
42         }
43
44         // 剩下的操作数入栈
45         while !op_stack.is_empty() {
46             postfix.push(op_stack.pop().unwrap())
47         }
48
49         // 出栈并组成字符串
50         let mut postfix_str = "".to_string();
51         for c in postfix {
52             postfix_str += &c.to_string();
53             postfix_str += " ";
54         }
55
56         Some(postfix_str)
57     }
58
59     fn main() {
60         let infix = "( A + B ) * ( C + D )";
61         let postfix = infix_to_postfix(infix);
62         match postfix {
63             Some(val) => {
64                 println!("infix: {infix} -> postfix: {val}");
65             },
66             None => {
67                 println!("{infix} is not a corret infix string");
68             },
69         }
70     }
```

计算后缀表达式的方法前面已经陈述过，但要注意 - , / 号，这个操作符不像 + 和 * 操作符。- 和 / 运算符要考虑操作数的顺序，A / B 和 B / A，A - B 和 B - A 是完全不同的，不能像 +，* 那样简单的处理。

假设后缀表达式是一个由空格分隔的标记字符串，运算符为 +-*/，操作数为整数，输出也是一个整数。下面是计算后缀表达式的算法步骤。

1. 创建一个名为 op_stack 的空栈。

2. 拆分字符串为符号列表。

3. 从左到右扫描符号列表。

如果符号是操作数，将其从字符串转换为整数，并将值压到 op_stack。如果符号是运算符，弹出 op_stack 两次。第一次弹出的是第二个操作数，第二次弹出的是第一个操作数。执行算术运算后，将结果压到操作数栈中。

4. 当输入的表达式被完全处理后，结果就在栈上，弹出 op_stack 得到最终运算值。

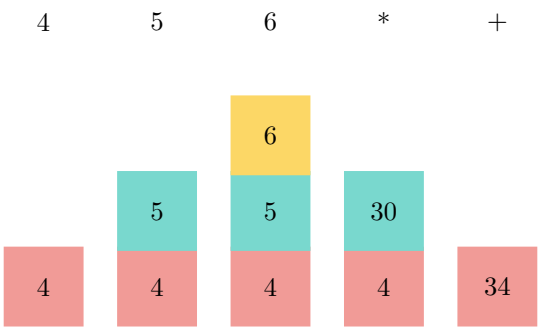


图 3.10: 用栈来计算后缀表达式

具体计算的代码如下所示。

```
1  //postfix_eval.rs
2
3  fn postfix_eval(postfix: &str) -> Option<i32> {
4      // 少于五个字符，不是有效的后缀表达式，因为表达式
5      // 至少两个操作数加一个操作符，还需要两个空格隔开
6      if postfix.len() < 5 { return None; }
7
8      let mut op_stack = Stack::new();
9      for token in postfix.split_whitespace() {
10         if "0" <= token  && token <= "9" {
11             op_stack.push(token.parse::<i32>().unwrap());
12         } else {
13             // 对于减法和除法，顺序有要求
14             // 所以先出栈的是第二个操作数
15             let op2 = op_stack.pop().unwrap();
16             let op1 = op_stack.pop().unwrap();
17             let res = do_calc(token, op1, op2);
18             op_stack.push(res);
19         }
```

```
20      }
21
22      Some(op_stack.pop().unwrap())
23  }
24
25  // 执行四则数学运算
26  fn do_calc(op: &str, op1: i32, op2: i32) -> i32 {
27      if "+" == op  {
28          op1 + op2
29      } else if "-" == op {
30          op1 - op2
31      } else if "*" == op {
32          op1 * op2
33      } else {
34          if 0 == op2 {
35              panic!("ZeroDivisionError: Invalid operation!");
36          }
37          op1 / op2
38      }
39  }
40
41  fn main() {
42      let postfix = "1 2 + 1 2 + *";
43      let res = postfix_eval(postfix);
44      match res {
45          Some(val) => println!("res is {val}"),
46          None => println!("{postfix} isn't a valid postfix"),
47      }
48  }
```

3.4 队列

队列是项的有序结合，其中添加新项的一端称为队尾，移除项的一端称为队首。当一个元素从队尾进入队列后，会一直向队首移动，直到它成为下一个需要移除的元素为止。最近添加的元素必须在队尾等待，集合中存活时间最长的元素在队首，因为它经历了从队尾到队首的移动。这种排序称为先进先出（First In First Out, FIFO），同栈的 LIFO 相反。

队列其实在生活中也很常见，单是从其名称也可见其普遍性。春运时火车站大排长龙等待进站，在公交车站排队上车，自助餐厅排队取餐等等都是队列。队列的行为是有限制的，因

为它只有一个入口，一个出口，不能插队，也不能提前离开，只有等待一定的时间才能到前面。

操作系统也使用队列这种数据结构，它使用多个不同的队列来控制进程。调度算法通常基于尽可能快地执行程序和尽可能多地服务用户的排队算法来决定下一步做什么。还有一种现象，有时敲击键盘，屏幕上出现的字符会有延迟，这是由于计算机在那一刻在做其他工作，按键内容被放置在类似队列的缓冲器中，使得它们最终可以正确地显示在屏幕上。下图展示了一个简单的 Rust 数字队列。

图 3.11: 队列

3.4.1 队列的抽象数据类型

如上所述，队列被构造成在队尾添加项的有序集合，并能从队首移除数据项。队列保持 FIFO 排序属性，其抽象数据类型由以下结构和操作定义。

new() 创建一个新队列，不需要参数，返回一个空队列。

enqueue(item) 将新项添加到队尾，需要 item 作为参数，不返回任何内容。

dequeue() 从队首移除项，不需要参数，返回 item，队列被修改。

is_empty() 检查队列是否为空，不需要参数，返回布尔值。

size() 返回队列中的项数，不需要参数，返回一个整数。

假设 q 是已经创建的空队列，下表展示了队列各种操作后的结果，左边为队首。

表 3.4: 队列操作

队列操作	队列当前值	操作返回值
q.is_empty()	[]	true
q.enqueue(1)	[1]	
q.enqueue(2)	[1,2]	
q.enqueue(3)	[1,2,3]	
q.enqueue(5)	[1,2,3,5]	
q.dequeue()	[2,3,5]	1
q.size()	[2,3,5]	3
q.is_empty()	[2,3,5]	false
q.enqueue(4)	[2,3,5,4]	
q.enqueue(6)	[2,3,5,4,6]	
q.dequeue()	[3,5,4,6]	2
q.size()	[3,5,4,6]	4

3.4.2 Rust 实现队列

前文已定义了队列的抽象数据类型，现在使用 Rust 来实现队列。和前面栈类似，这种线性的数据集合用 Vec 就可以，稍微限制下对 Vec 加入和移除元素的用法就能实现队列。我们选择 Vec 的左端作为队尾，右端作为队首，这样移除数据的复杂度是 O(1)，加入数据的复杂度为 O(n)。为了防止队列无限增长，添加了一个 cap 参数用于控制队列长度。

```
1  // queue.rs
2
3  // 队列定义
4  #[derive(Debug)]
5  struct Queue<T> {
6      cap: usize,    // 容量
7      data: Vec<T>, // 数据容器
8  }
9
10 impl<T> Queue<T> {
11     fn new(size: usize) -> Self {
12         Queue {
13             data: Vec::with_capacity(size),
14             cap: size,
15         }
16     }
17
18     // 判断是否有剩余空间、有则数据加入队列
19     fn enqueue(&mut self, val: T) -> Result<(), String> {
20         if Self::size(&self) == self.cap {
21             return Err("No space available".to_string());
22         }
23         self.data.insert(0, val);
24
25         Ok(())
26     }
27
28     // 数据出队
29     fn dequeue(&mut self) -> Option<T> {
30         if Self::size(&self) > 0 {
31             self.data.pop()
32         } else {
33             None
```

```
34              }
35          }
36
37          fn is_empty(&self) -> bool {
38              0 == Self::size(&self)
39          }
40
41          fn size(&self) -> usize {
42              self.data.len()
43          }
44      }
45
46      fn main() {
47          let mut q = Queue::new(3);
48          let _r1 = q.enqueue(1);
49          let _r2 = q.enqueue(2);
50          let _r3 = q.enqueue(3);
51          if let Err(error) = q.enqueue(4) {
52              println!("Enqueue error: {error}");
53          }
54
55          if let Some(data) = q.dequeue() {
56              println!("data: {data}");
57          } else {
58              println!("empty queue");
59          }
60
61          println!("size: {}, empty: {}", q.size(), q.is_empty());
62          println!("content: {:?}", q);
63      }
```

3.4.3 烫手山芋

队列的典型应用是模拟需要以 FIFO 方式管理数据的真实场景。一个例子是烫手山芋游戏，在这个游戏中（见图3.12），孩子们围成一个圈，并尽可能快地将一个山芋递给旁边的孩子。在某一个时刻，停止传递动作，有山芋的孩子从圈中移除，继续游戏直到剩下最后一个孩子。这个游戏使得孩子们尽可能快的把山芋传递出去，就像山芋很烫手一样，所以这个游戏叫烫手山芋。

这个游戏相当于著名的约瑟夫问题，一世纪历史学家弗拉维奥 · 约瑟夫斯讲述的传奇故事。他和 39 个战友被罗马军队包围在洞中，他们决定宁愿死，也不成为罗马人的奴隶。他们围成一个圈，其中一人被指定为第一个人，顺时针报数到第七人，就将他杀死，直到剩下一人。约瑟夫斯是一个数学家，他立即想出了应该坐到哪个位置才能成为最后一人。最后，他成了最后一人，加入了罗马的一方。这个故事有各种不同的版本，有些说是每次报数到第三个人，有人说允许最后一个人逃跑。无论如何，两个故事思想是一样的，都可用队列来模拟。程序将输入多个人名代表孩子，一个 num 常量用于设定报数到第几人。

假设拿山芋的孩子始终在队列的前面。当拿到山芋的时候，这个孩子将先出队再入队，把自己放在队列的最后，这相当于他把山芋传递给了下一个孩子，而那个孩子必须处于队首，所以他自己出队，让出了队首的位置，自己加入了队尾。经过 num 次的出队入队后，前面的孩子被永久移除。接着另一个周期开始，继续此过程，直到只剩下一个名字。

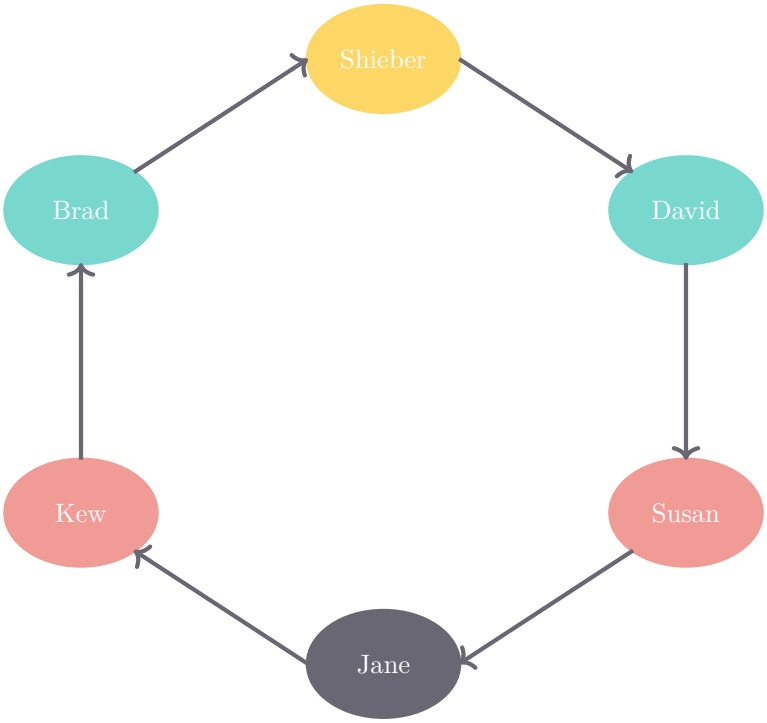


图 3.12: 烫手山芋

通过上面的分析可以得到如下烫手山芋游戏的队列模型。

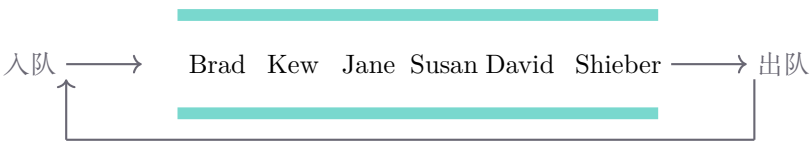


图 3.13: 模拟烫手山芋

上面两幅图表示的出入队列模型十分清楚，下面按照物理模型来实现烫手山芋游戏。

```rust
// hot_potato.rs

fn hot_potato(names: Vec<&str>, num: usize) -> &str {
    // 初始化队列、名字入队
    let mut q = Queue::new(names.len());
    for name in names {
        let _rm = q.enqueue(name);
    }

    while q.size() > 1 {
        // 出入栈名字，相当于传递山芋
        for _i in 0..num {
            let name = q.dequeue().unwrap();
            let _rm = q.enqueue(name);
        }

        // 出入栈达到 num 次，删除一人
        let _rm = q.dequeue();
    }

    q.dequeue().unwrap()
}

fn main() {
    let name = vec!["Shieber","David","Susan","Jane","Kew","Brad"];
    let rem = hot_potato(name, 8);
    println!("The left person is {rem}");
}
```

请注意，在实现中，计数值 8 大于队列中人数 6。但这不存在问题，因为队列像是一个圈，到尾后会重新回到首部，直到达到计数值，所以最终总是有个人会出队。

3.5 双端队列

deque 又称为双端队列，是与队列类似的项的有序集合。它有两个端部，首端和尾端。deque 不同于 queue 的地方是添加和删除项是非限制性的，可以在首端尾后端添加项，同样也可以从任一端移除项。在某种意义上，这种混合线性结构提供了栈和队列的所有功能。

即使 deque 拥有栈和队列的许多特性，但它不需要像那些数据结构那样强制的 LIFO 和 FIFO 排序，这取决于如何添加和删除数据。你把它当栈用，它就是栈，当队列用就是队列，但一般来说，deque 就是 deque，不要当成栈或队列使用，不同的数据结构都有其独特性，是为了不同的计算目的而设计的。下图展示了一个 deque。

图 3.14: 双端队列

3.5.1 双端队列的抽象数据类型

如上所述，deque 被构造为项的有序集合，其中项从首部或尾部的任一端添加和移除，deque 抽象数据类型由以下结构和操作定义。

new() 创建一个新的 deque，不需要参数，返回空的 deque。

add_front(item) 将新项 item 添加到 deque 首部，需要 item 参数，不返回任何内容。

add_rear(item) 将新项 item 添加到 deque 尾部，需要 item 参数，不返回任何内容。

remove_front() 从 deque 中删除首项，不需要参数，返回 item。deque 被修改。

remove_rear() 从 deque 中删除尾项，不需要参数，返回 item，deque 被修改。

is_empty() 测试 deque 是否为空，不需要参数，返回布尔值。

size() 返回 deque 中的项数，不需要参数，返回一个整数。

假设 d 是已经创建的空 deque，下表展示了一系列操作后的结果。注意，首部在右端。将 item 移入和移出时，注意跟踪前后内容，因为两端修改使得结果看起来有些混乱。

表 3.5: 双端队列操作

双端队列操作	双端队列当前值	操作返回值
d.is_empty()	[]	true
d.add_rear(1)	[1]	
d.add_rear(2)	[2,1]	
d.add_front(3)	[2,1,3]	
d.add_front(4)	[2,1,3,4]	
d.size()	[2,1,3,4]	4
d.is_empty()	[2,1,3,4]	false
d.remove_rear()	[1,3,4]	2
d.remove_front()	[1,3]	4
d.size()	[1,3]	2

3.5.2 Rust 实现双端队列

前文定义了双端队列的抽象数据类型，现在使用 Rust 来实现双端队列。和前面队列类似，这种线性的数据集合用 Vec 就可以。选择 Vec 的左端作为队尾，右端作为队首。为防止队列无限增长，添加了一个 cap 参数用于控制双端队列长度。

```
1  // deque.rs
2
3  // 双端队列
4  #[derive(Debug)]
5  struct Deque<T> {
6      cap: usize,    // 容量
7      data: Vec<T>, // 数据容器
8  }
9
10 impl<T> Deque<T> {
11     fn new(cap: usize) -> Self {
12         Deque {
13             cap: cap,
14             data: Vec::with_capacity(size),
15         }
16     }
17
18     // Vec 末尾为队首
19     fn add_front(&mut self, val: T) -> Result<(), String> {
20         if Self::size(&self) == self.cap {
21             return Err("No space available".to_string());
22         }
23         self.data.push(val);
24
25         Ok(())
26     }
27
28     // Vec 首部为队尾
29     fn add_rear(&mut self, val: T) -> Result<(), String> {
30         if Self::size(&self) == self.cap {
31             return Err("No space available".to_string());
32         }
33         self.data.insert(0, val);
34
```

```
35          Ok(())
36      }
37
38      // 从队首移除数据
39      fn remove_front(&mut self) -> Option<T> {
40          if Self::size(&self) > 0 {
41              self.data.pop()
42          } else {
43              None
44          }
45      }
46
47      // 从队尾移除数据
48      fn remove_rear(&mut self) -> Option<T> {
49          if Self::size(&self) > 0 {
50              Some(self.data.remove(0))
51          } else {
52              None
53          }
54      }
55
56      fn is_empty(&self) -> bool {
57          0 == Self::size(&self)
58      }
59
60      fn size(&self) -> usize {
61          self.data.len()
62      }
63  }
64
65  fn main() {
66      let mut d = Deque::new(4);
67      let _r1 = d.add_front(1); let _r2 = d.add_front(2);
68      let _r3 = d.add_rear(3);  let _r4 = d.add_rear(4);
69      if let Err(error) = d.add_front(5) {
70          println!("add_front error: {error}");
71      }
72
```

```
73      if let Some(data) = d.remove_rear() {
74          println!("data: {data}");
75      } else {
76          println!("empty queue");
77      }
78
79      println!("size: {}, is_empty: {}", d.size(), d.is_empty());
80      println!("content: {:?}", d);
81  }
```

可见，Deque 同 Queue 和 Stack 都有相似之处，感觉像是两者的合集一样。具体的实现是可以商榷的，哪端是首，哪端是尾要依据情况而定。

3.5.3 回文检测

回文是一个字符串，距离首尾两端相同位置处的字符相同。例如 radar，sos，rustsur 这类字符串。可以写一个算法来检查一个字符串是否是回文。一种方式是用队列 Queue，将字符串入队，然后字符再出队。此时出队的字符和原字符串的倒序相比较，如果相等就继续出队比较下一个，直到完成。这种方式很直观，但是费内存。检查回文的另一种方案是使用 Deque（如下图）。

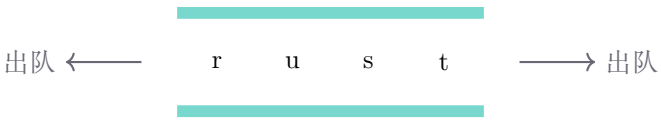


图 3.15: 回文检测

从左到右处理字符串，将每个字符添加到 Deque 尾部，此时 deque 的首部保存字符串的第一个字符，尾部保存最后一个字符。此时可以直接利用 Deque 的双边出队特性，删除首尾字符并比较，只有当首尾字符相等时才继续。如果可以持续匹配首尾字符，最终要么用完字符，留下空队列，要么留出大小为 1 的队列。在这两种情况下，字符串均是回文，这取决于原始字符串的长度是偶数还是奇数。下面是回文检查的完整代码。

```
1  // pal_checker.rs
2
3  fn pal_checker(pal: &str) -> bool {
4      let mut d = Deque::new(pal.len());
5      for c in pal.chars() {
6          let _r = d.add_rear(c);
7      }
8
```

```
9       let mut is_pal = true;
10      while d.size() > 1 && is_pal {
11          let head = d.remove_front();
12          let tail = d.remove_rear();
13          if head != tail { // 比较首尾字符，若不同则非回文
14              is_pal = false;
15          }
16      }
17
18      is_pal
19  }
20
21  fn main() {
22      let pal = "rustsur";
23      let is_pal = pal_checker(pal);
24      println!("{pal} is palindrome string: {is_pal}");
25  }
```

3.6 链表

有序的数据项集合能保证数据的相对位置，可以高效地索引。数组和链表都能做到将数据有序地收集起来并保存在相对的位置，所以数组和链表都可用于实现有序数据类型，比如 Rust 默认实现的 Vec 就是用的数组这种有序集合。当然，本节研究链表，我们要先实现链表再用其来实现其他的有序数据类型。

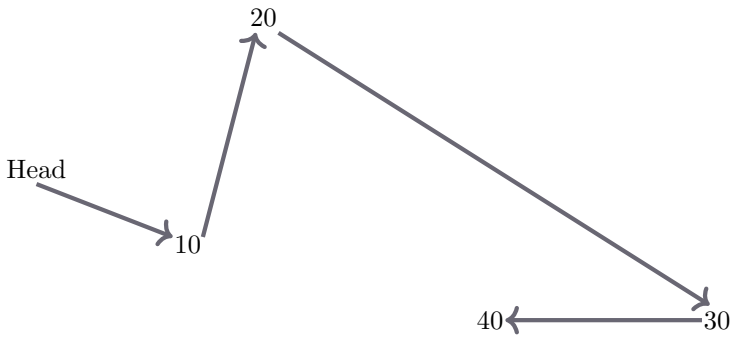


图 3.16: 链表数据关系

数组是一片连续的内存，且增删元素涉及到内存复制和移动等操作，非常耗时。链表不要求元素保存在连续的内存中。如上图所示的项集合，这些值随机放置，只要每个项中都有

下一项的位置，则项的位置可以通过简单地从一个项到下一个项的链接来表示，用不着像数组那样去分配整块内存，这样效率会更高。

必须明确地指定链表的第一项位置，一旦知道第一项在哪里，就可以知道第二项在哪里，直到整个链表结束。链表对外提供的通常是链表的头（Head），类似地，最后一个项要设置下一个项为空（None）或称为接地。

3.6.1 链表的抽象数据类型

链表的抽象数据类型由其结构和操作定义。如上所述，链表被构造为节点项的有序集合，可以从头节点遍历整个链表数据。链表结构很简单，只有一个头节点引用。下面是链表节点 Node 的具体的定义和操作。

new() 创建一个新的头节点用于指向 Node，不需要参数，返回指针。

push(item) 添加一个新的 Node，需要 item 参数，返回 Node。

pop() 删除链表头节点，不需要参数，返回 Node。

peek() 返回链表头节点，不需要参数，返回对值的引用。

peek_mut() 返回链表头节点，不需要参数，返回对值的可变引用。

into_iter() 改变链表为可迭代形式，不需要参数。

iter() 返回链表不可变迭代形式，链表不变，不需要参数。

iter_mut() 返回链表可变迭代形式，链表不变，不需要参数。

into_iter() 返回链表头节点，不需要参数，返回对值的可变引用。

is_empty() 返回当前链表是否为空，不需要参数，返回布尔值。

size() 计算链表的长度，不需要参数，返回整数值。

假设 l 是已经创建的空链表，下表展示了链表操作序列后的结果，左端为头节点，注意 Link<num> 表示指向 num 所在节点的地址。

表 3.6: 链表操作

链表操作	链表当前值	操作返回值
l.is_empty()	[None->None]	true
l.push(1)	[1->None]	
l.push(2)	[2->1->None]	
l.push(3)	[3->2->1->None]	
l.peek()	[3->2->1->None]	Link<3>
l.pop()	[2->1->None]	3
l.size()	[2->1->None]	2
l.push(4)	[4->2->1->None]	
l.peek_mut()	[4->3->2->None]	mut Link<4>
l.iter()	[4->3->2->None]	[4,3,2]
l.is_empty()	[4->3->2->None]	false
l.into_iter()	[None->None]	[4,3,2]

3.6.2 Rust 实现链表

链表中的每项都可抽象成一个节点 Node，节点保存了数据项和下一项位置。当然，节点还提供获取和修改数据项的方法。如下图所示，Node 包含数据和下一结点的地址。

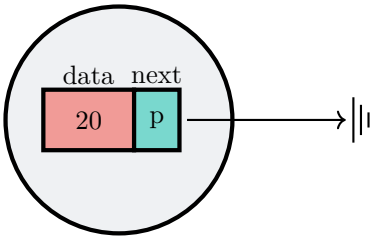


图 3.17: 链表节点

Rust 中的 None 将在 Node 和链表中发挥重要作用，None 地址代表没有下一个节点。请注意在 new 函数中，最初创建的节点 next 被设置为 None，这被称为接地节点，将 None 显式的分配给 next 是个好主意，这避免了 C++ 等语言中容易出现的悬荡指针。

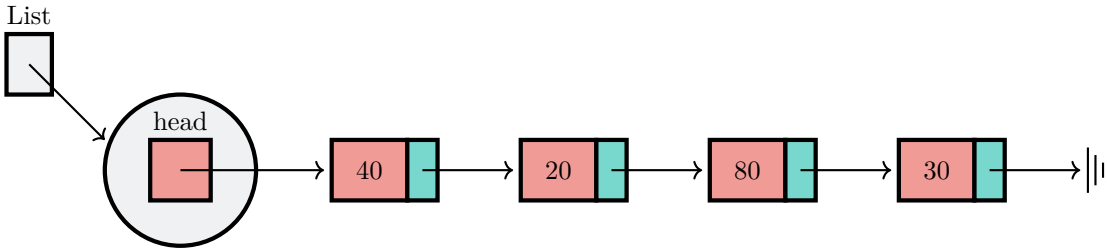


图 3.18: 链表 List

下面是链表的实现代码，为了对链表实现迭代功能，我们添加了 IntoIter、Iter、IterMut 三个结构体，分别完成三种迭代功能，InotIter 是将链表整个转换为可迭代类型，Iter 只迭代，不修改链表，IterMut 也是迭代，但可以修改链表结点。Drop 类似析构函数。

```
1  // linked_list.rs
2  // 节点连接用 Box 指针（大小确定），因为确定大小才能分配内存
3  type Link<T> = Option<Box<Node<T>>>;
4
5  // 链表定义
6  pub struct List<T> {
7      size: usize,    // 链表节点数
8      head: Link<T>, // 头节点
9  }
10
11 // 链表节点
```

```
12 struct Node<T> {
13     elem: T,        // 数据
14     next: Link<T>, // 下一个节点链接
15 }
16
17 impl<T> List <T> {
18     pub fn new() -> Self {
19         List { size: 0, head: None }
20     }
21
22     pub fn is_empty(&self) -> bool {
23         0 == self.size
24     }
25
26     pub fn size(&self) -> usize {
27         self.size
28     }
29
30     // 新节点总是加到头部
31     pub fn push(&mut self, elem: T) {
32         let node = Box::new(Node {
33             elem: elem,
34             next: self.head.take(),
35         });
36         self.head = Some(node);
37         self.size += 1;
38     }
39
40     // take 会取出数据并留下空位
41     pub fn pop(&mut self) -> Option<T> {
42         self.head.take().map(|node| {
43             self.head = node.next;
44             self.size -= 1;
45             node.elem
46         })
47     }
48
49     // peek 不改变值，只能是引用
```

```
50      pub fn peek(&self) -> Option<&T> {
51          self.head.as_ref().map(|node| &node.elem )
52      }
53
54      // peek_mut 可改变值, 是可变引用
55      pub fn peek_mut(&mut self) -> Option<&mut T> {
56          self.head.as_mut().map(|node| &mut node.elem )
57      }
58
59      // 以下是实现的三种迭代功能
60      // into_iter: 链表改变, 成为迭代器
61      // iter: 链表不变, 只得到不可变迭代器
62      // iter_mut: 链表不变, 得到可变迭代器
63      pub fn into_iter(self) -> IntoIter<T> {
64          IntoIter(self)
65      }
66
67      pub fn iter(&self) -> Iter<T> {
68          Iter { next: self.head.as_deref() }
69      }
70
71      pub fn iter_mut(&mut self) -> IterMut<T> {
72          IterMut { next: self.head.as_deref_mut() }
73      }
74  }
75
76  // 实现三种迭代功能
77  pub struct IntoIter<T>(List<T>);
78  impl<T> Iterator for IntoIter<T> {
79      type Item = T;
80      fn next(&mut self) -> Option<Self::Item> {
81          self.0.pop()
82      }
83  }
84
85  pub struct Iter<'a, T: 'a> { next: Option<&'a Node<T>> }
86  impl<'a, T> Iterator for Iter<'a, T> {
87      type Item = &'a T;
```

```
88      fn next(&mut self) -> Option<Self::Item> {
89          self.next.map(|node| {
90              self.next = node.next.as_deref();
91              &node.elem
92          })
93      }
94  }
95
96  pub struct IterMut<'a, T: 'a> { next: Option<&'a mut Node<T>> }
97  impl<'a, T> Iterator for IterMut<'a, T> {
98      type Item = &'a mut T;
99      fn next(&mut self) -> Option<Self::Item> {
100         self.next.take().map(|node| {
101             self.next = node.next.as_deref_mut();
102             &mut node.elem
103         })
104     }
105 }
106
107 // 为链表实现自定义 Drop
108 impl<T> Drop for List<T> {
109     fn drop(&mut self) {
110         let mut link = self.head.take();
111         while let Some(mut node) = link {
112             link = node.next.take();
113         }
114     }
115 }
116
117 fn main() {
118     fn basics() {
119         let mut list = List::new();
120         list.push(1); list.push(2); list.push(3);
121         assert_eq!(list.pop(), Some(3));
122         assert_eq!(list.peek(), Some(&2));
123         assert_eq!(list.peek_mut(), Some(&mut 2));
124         list.peek_mut().map(|val| {
125             *val = 4;
```

```
126             });
127             assert_eq!(list.peek(), Some(&4));
128             println!("basics test Ok!");
129         }
130 
131         fn into_iter() {
132             let mut list = List::new();
133             list.push(1); list.push(2); list.push(3);
134             let mut iter = list.into_iter();
135             assert_eq!(iter.next(), Some(3));
136             assert_eq!(iter.next(), Some(2));
137             assert_eq!(iter.next(), Some(1));
138             assert_eq!(iter.next(), None);
139             println!("into_iter test Ok!");
140         }
141 
142         fn iter() {
143             let mut list = List::new();
144             list.push(1); list.push(2); list.push(3);
145             let mut iter = list.iter();
146             assert_eq!(iter.next(), Some(&3));
147             assert_eq!(iter.next(), Some(&2));
148             assert_eq!(iter.next(), Some(&1));
149             assert_eq!(iter.next(), None);
150             println!("iter test Ok!");
151         }
152 
153         fn iter_mut() {
154             let mut list = List::new();
155             list.push(1); list.push(2); list.push(3);
156             let mut iter = list.iter_mut();
157             assert_eq!(iter.next(), Some(&mut 3));
158             assert_eq!(iter.next(), Some(&mut 2));
159             assert_eq!(iter.next(), Some(&mut 1));
160             assert_eq!(iter.next(), None);
161             println!("iter_mut test Ok!");
162         }
163 
```

```
164         basics();
165         into_iter();
166         iter();
167         iter_mut();
168     }
```

3.6.3 链表栈

前面使用 Vec 实现了栈，其实还可用链表来实现栈，因为都是线性数据结构。假设链表的头部将保存栈顶部元素，随着栈增长，新项将被添加到链表头部，实现如下。

注意，push 和 pop 函数会改变链表的结点，所以使用了 take 函数来取出结点值。

```
1   // list_stack.rs
2   // 链表节点
3   #[derive(Debug, Clone)]
4   struct Node<T> {
5       data: T,
6       next: Link<T>,
7   }
8
9   type Link<T> = Option<Box<Node<T>>>;
10
11  impl<T> Node<T> {
12      fn new(data: T) -> Self {
13          Node {
14              data: data, next: None, // 初始化时无下一链接
15          }
16      }
17  }
18
19  // 链表栈
20  #[derive(Debug, Clone)]
21  struct Stack<T> {
22      size: usize,
23      top: Link<T>,  // 栈顶控制整个栈
24  }
25
26  impl<T: Clone> Stack<T> {
27      fn new() -> Self {
```

```
28          Stack { size: 0, top: None }
29      }
30
31      // take 取出 top 中节点, 留下空位, 所以可以回填节点
32      fn push(&mut self, val: T) {
33          let mut node = Node::new(val);
34          node.next = self.top.take();
35          self.top = Some(Box::new(node));
36          self.size += 1;
37      }
38
39      fn pop(&mut self) -> Option<T> {
40          self.top.take().map(|node| {
41              let node = *node;
42              self.top = node.next;
43              node.data
44          })
45      }
46
47      // as_ref 将 top 节点转为引用对象
48      fn peek(&self) -> Option<T> {
49          self.top.as_ref().map(|node| {
50              &node.data
51          })
52      }
53
54      fn size(&self) -> usize {
55          self.size
56      }
57
58      fn is_empty(&self) -> bool {
59          0 == self.size
60      }
61  }
62
63  fn main() {
64      let mut s = Stack::new();
65      s.push(1); s.push(2); s.push(4);
```

```
66        println!("top {:?}, size {}",s.peek().unwrap(), s.size());
67        println!("pop {:?}, size {}",s.pop().unwrap(), s.size());
68        println!("is_empty:{}, stack:{:?}", s.is_empty(), s);
69    }
```

3.7 Vec

在对基本数据结构的讨论中，我们使用 Vec 这一基础数据类型来实现了栈，队列等多种抽象数据类型。Vec 是一个强大但简单的数据容器，提供了数据收集机制和各种各样的操作，这也是反复使用它来作为底层数据结构的原因。Vec 类似于 Python 中的 List，使用非常方便。然而，不是所有的编程语言都包括 Vec，或者说不是所有的数据类型都适合你。在某些情况下，Vec 或类似的数据容器必须由程序员单独实现。

3.7.1 Vec 的抽象数据类型

如上所述，Vec 是项的集合，其中每个项保持相对于其他项的相对位置。下面给出了 Vec 抽象数据类型的各种操作。

new() 创建一个新的 Vec，不需要参数，返回一个空 Vec。

push(item) 将新项添加到 Vec 末尾，需要 item 参数，不返回任何内容。

pop() 删除 Vec 中的末尾项，不需要参数，返回删除的项。

insert(pos,item) 在 Vec 的 pos 处插入新项，需要 pos 和 item 参数，不返回任何内容。

remove(index) 从列表中删除第 index 项，需要 index 作为索引，返回删除项。

find(item) 在 Vec 中检查 item 项是否存在，需要 item 参数，返回一个布尔值。

is_empty() 检查 Vec 是否为空，不需要参数，返回布尔值。

size() 计算 Vec 的项数，不需要参数，返回一个整数。

假设 v 是已经创建的空 Vec，下表展示了 Vec 不同操作后的结果，其中左侧为首部。

表 3.7: Vec 操作

Vec 操作	Vec 当前值	操作返回值
v.is_empty()	[]	true
v.push(1)	[1]	
v.push(2)	[1,2]	
v.size()	[1,2]	2
v.pop()	[1]	2
v.push(5)	[1,5]	
v.find(4)	[1,5]	false
v.insert(0,8)	[8,1,5]	
v.remove(0)	[8,1,5]	8

3.7.2 Rust 实现 Vec

如上所述，Vec 将用一组链表节点来构建，每个节点通过显式引用链接到下一个节点。只要知道在哪里找到第一个节点，之后的每项都可以通过连续获取下一个链接找到。

考虑到引用在 Vec 中的作用，Vec 必须保持对第一个节点的引用。创建如图（3.19）所示的链表，None 用于表示链表不引用任何内容。链表的头指代列表的第一节点，该节点保存下一个节点的地址。要注意到 Vec 本身不包含任何节点对象，相反，它只包含对链表结构中第一个节点的引用。

那么，如何将新项加入链表呢? 加到首部还是尾部呢? 链表结构只提供了一个入口点，即链表头部。所有其他节点只能通过访问第一个节点，然后跟随下一个链接到达。这意味着添加新节点的最简单的地方就在链表的头部，换句话说，将新项作为链表的第一项，现有项链接到这个新项后面。

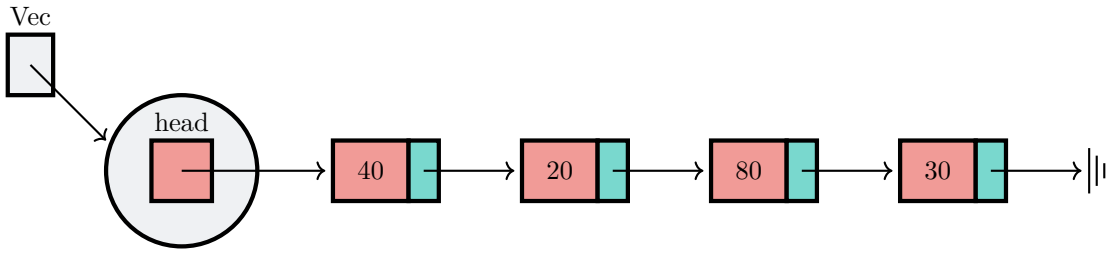


图 3.19: 链表节点组成的的 Vec

我们实现的 Vec 是无序的，若要实现有序 Vec 只需添加数据项比较函数。下面实现的 LVec，仅实现了 Rust Vec 中的部分功能，包括 new、insert、pop、remove、push、append 等方法，print_lvec 用于打印 LVec 数据项。

```
1  // lvec.rs
2  use std::fmt::Debug;
3
4  // 节点
5  #[derive(Debug)]
6  struct Node<T> {
7      elem: T,
8      next: Link<T>,
9  }
10
11 type Link<T> = Option<Box<Node<T>>>;
12
13 impl<T> Node<T> {
14     fn new(elem: T) -> Node<T> {
15         Node { elem: elem, next: None }
```

```
16      }
17  }
18
19  // 链表 Vec
20  #[derive(Debug)]
21  struct LVec<T> {
22      size: usize,
23      head: Link<T>,
24  }
25
26  impl<T: Copy + Debug> LVec<T> {
27      fn new() -> Self {
28          LVec { size: 0, head: None }
29      }
30
31      fn clear(&mut self) {
32          self.size = 0;
33          self.head = None;
34      }
35
36      fn len(&self) -> usize {
37          self.size
38      }
39
40      fn is_empty(&self) -> bool {
41          0 == self.size
42      }
43
44      fn push(&mut self, elem: T) {
45          let node = Node::new(elem);
46
47          if self.is_empty() {
48              self.head = Some(Box::new(node));
49          } else {
50              let mut curr = self.head.as_mut().unwrap();
51
52              // 找到最后一个节点
53              for _i in 0..self.size-1 {
```

```
54                  curr = curr.next.as_mut().unwrap();
55              }
56
57              // 最后一个节点后插入新数据
58              curr.next = Some(Box::new(node));
59          }
60
61          self.size += 1;
62      }
63
64      // 栈末尾加入数据
65      fn append(&mut self, other: &mut Self) {
66          while let Some(node) = other.head.as_mut().take() {
67              self.push(node.elem);
68              other.head = node.next.take();
69          }
70          other.clear();
71      }
72
73      fn insert(&mut self, mut index: usize, elem: T) {
74          if index >= self.size {
75              index = self.size;
76          }
77
78          // 分三种情况插入新节点
79          let mut node = Node::new(elem);
80          if self.is_empty() { // LVec 为空, 直接插入
81              self.head = Some(Box::new(node));
82          } else if index == 0 { // 插入链表首部
83              node.next = self.head.take();
84              self.head = Some(Box::new(node));
85          } else { // 插入链表中间
86              let mut curr = self.head.as_mut().unwrap();
87              for _i in 0..index - 1 { // 找到插入位置
88                  curr = curr.next.as_mut().unwrap();
89              }
90              node.next = curr.next.take();
91              curr.next = Some(Box::new(node));
```

```
92              }
93              self.size += 1;
94          }
95
96          fn pop(&mut self) -> Option<T> {
97              self.remove(self.size - 1)
98          }
99
100         fn remove(&mut self, index: usize) -> Option<T> {
101             if index >= self.size { return None; }
102
103             // 分两种情况删除节点，首节点删除最好处理
104             let mut node;
105             if 0 == index {
106                 node = self.head.take().unwrap();
107                 self.head = node.next.take();
108             } else { // 非首节点需要找到待删除点，并处理前后链接
109                 let mut curr = self.head.as_mut().unwrap();
110                 for _i in 0..index-1 {
111                     curr = curr.next.as_mut().unwrap();
112                 }
113                 node = curr.next.take().unwrap();
114                 curr.next = node.next.take();
115             }
116             self.size -= 1;
117
118             Some(node.elem)
119         }
120
121         // 打印 LVec，当然也可以实现 ToString 特性并用 println 打印
122         fn print_lvec(&self) {
123             let mut curr = self.head.as_ref();
124             while let Some(node) = curr {
125                 println!("lvec val: {:#?}", node.elem);
126                 curr = node.next.as_ref();
127             }
128         }
129     }
```

```
130
131 fn main() {
132     let mut lvec: LVec<i32> = LVec::new();
133     lvec.push(10); lvec.push(11);
134     lvec.push(12); lvec.push(13);
135     lvec.insert(0,9);
136
137     let mut lvec2: LVec<i32> = LVec::new();
138     lvec2.insert(0, 8);
139     lvec2.append(&mut lvec);
140     println!("lvec2 len: {}", lvec2.len());
141     lvec2.print_lvec();
142
143     let res1 = lvec2.pop();
144     let res2 = lvec2.remove(0);
145     println!("pop {:#?}", res1.unwrap());
146     println!("remove {:#?}", res2.unwrap());
147     lvec2.print_lvec();
148 }
```

LVec 是具有 n 个节点的链表，insert，push，pop，remove 等都需要遍历结点，虽然平均来说可能只需要遍历节点的一半，但总体上都是 O(n)，因为在最坏的情况下，都要处理链表中的每个节点。

3.8 总结

本章主要学习了栈、队列、双端队列、链表、Vec 这些线性数据结构。栈是维持后进先出（LIFO）排序的数据结构，其基本操作是 push，pop，is_empty。栈对于设计计算解析表达式算法非常有用，栈可以提供反转特性，在操作系统函数调用，网页保存方面非常有用。前缀，中缀和后缀表达式都是表达式，可以用栈来处理，但计算机不用中缀表达式。队列是维护先进先出（FIFO）排序的简单数据结构，其基本操作是 enqueue，dequeue，is_empty，队列在系统任务调度方面很实用，可以帮助构建定时仿真。双端队列是允许类似栈和队列的混合行为的数据结构，其基本操作是 is_empty，add_front，add_rear，remove_front，remove_rear。链表是项的集合，其中每个项保存在相对位置。链表的实现本身就能保持逻辑顺序，不需要物理顺序存储。修改链表头是一种特殊情况。Vec 是 Rust 自带的数据容器，默认实现是用的动态数组，本章使用的是链表。

Chapter 4

递归

4.1 本章目标

要理解简单的递归解决方案
学习如何用递归写出程序
理解和应用递归三个定律
将递归理解为一种迭代形式
将问题公式化地实现成递归
了解计算机如何实现递归

4.2 什么是递归

递归是一种解决问题的方法，通过将问题分解为更小的子问题，直到得到一个足够小的基本问题。这个基本问题可以被很简单地解决，再通过合并基本问题的结果得到大问题的结果。因为这些基本问题是类似的，可以重用解决方案，所以递归涉及到函数调用自身。递归允许你编写非常优雅的解决方案解决看起来可能很难的问题。

举个简单的例子。假设你想计算整数 [2,1,7,4,5] 的总和，最直观的就是用一个加法累加器，逐个将值与之相加，具体代码如下。

```
1 fn nums_sum(nums: Vec<i32>) -> i32 {
2     let mut sum = 0;
3     for num in nums {
4         sum += num;
5     }
6
7     sum
8 }
```

现在假设一种编程语言没有 while 循环或 for 循环，那么上面的代码就不成立了，此时你又将如何计算该整数列表的总和呢？要解决这个问题得换个角度思考问题，当解决大问题（数列求和）困难时，要考虑使用小问题去替代。加法是两个操作数和一个加法符号组合的运算逻辑，这是任何复杂加法的基本问题。所以如果能将数列求和分解为一个个小的加法和，则总是能解决这个求和问题的。

构造小加法需要使用你小学学过的知识，构造完全括号表达式，当然这种括号表达式有多种形式。

$$
\begin{aligned}
2+1+7+4+5 &= ((((2+1)+7)+4)+5) \\
sum &= (((3+7)+4)+5) \\
sum &= ((10+4)+5) \\
sum &= (14+5) \\
sum &= 19 \\
&= (2+(1+(7+(4+5)))) \\
sum &= (2+(1+(7+9))) \\
sum &= (2+(1+16)) \\
sum &= (2+17) \\
sum &= 19
\end{aligned}
\tag{4.1}
$$

上面的式子，右侧两种括号表达式都是正确的。运用括号优先，内部优先的规则，那么上述括号表达式就是一个一个的小加法，完全可以不用 While 或 For 循环来模拟实现上面这种括号表达式。

观察以 sum 开头的计算式，从下往上看，先是 19，接着是 (2 + 17)，然后是 (2 + (1 + 16))。可以发现，总和是第一项和右端剩下项的和，而右端剩下项又可以分解为它的第一项和右端剩下项的和。用数学表达式就是：

$$Sum(nums) = First(nums) + Sum(restR(nums))$$

这是括号表达式中的第二种表示法的计算方式，当然还有第一种括号表达式的计算方式，具体如下：

$$Sum(nums) = Last(nums) + Sum(restL(nums))$$

方程式中，First(nums) 返回数列第一个元素，restR(nums) 返回除第一个元素之外的所有右端数字项。Last(nums) 返回数列最后一个元素，restL(nums) 返回除最后一个元素外的所有左端数字项。

用 Rust 递归实现的两种表达式计算方法如下。nums_sum1 使用 nums[0] 加剩下的项来求和，而 nums_sum2 使用最后一项和前面所有项来求和，两实现的效率几乎相等。

```rust
// nums_sum12.rs

fn nums_sum1(nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        nums[0]
    } else {
        let first = nums[0];
        first + nums_sum1(&nums[1..])
    }
}

fn nums_sum2(nums: &[i32]) -> i32 {
    if 1 == nums.len() {
        nums[0]
    } else {
        let last = nums[nums.len() - 1];
        nums_sum2(&nums[..nums.len() - 1]) + last
    }
}

fn main() {
    let nums = [2,1,7,4,5];
    let sum1 = nums_sum1(&nums);
    let sum2 = nums_sum2(&nums);
    println!("sum1 is {sum1}, sum2 is {sum2}");
}
```

代码中关键处是 if 和 else 语句及其形式。if 1 == nums.len() 检查是至关重要的，因为这是函数的转折点，这里返回数字。else 语句中调用自身，实现了类似逐层解括号并计算值的效果，这也是被称之为递归的原因。递归函数总会调用自身，直到到达基本情况。

4.2.1 递归三定律

通过分析上面的代码可以看出，所有递归算法必须遵从三个基本规律:

```
递归算法必须具有基本情况
递归算法必须向基本情况靠近
递归算法必须以递归方式调用自身
```

第一条就是算法停止的情况，此处是 if 1 == nums.len() 子句，第二条意味着问题的分解，在 else 中，我们返回了数字，然后使用除返回数字项的集合再次计算，这减小了原来的 nums 中需要计算的元素个数，可见总会有 nums.len() == 1 的时候，所以 else 子句确实在向基本情况靠近。第三条，调用自身，也是在 else 子句实现的。要注意，调用自身不是循环，这里也没有 while 和 for 语句。综上，我们的求和递归算法是符合这三个定律的。

下图展示了上面的递归调用函数处理过程，一系列方框即函数调用关系图，每次递归都是解决一个小问题，直到小问题达到基本情况不能再分解，最后再回溯这些中间计算值来求大问题的结果。

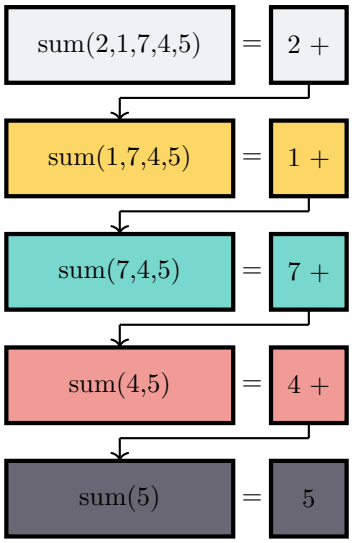


图 4.1: 递归求值

4.2.2 到任意进制的转换

前面在栈那一节实现了整数转换为二进制和十六进制字符串。例如，将整数 10 转换为二进制字符串 1010。其实使用递归来实现进制的转换也是可以的。

要用递归设计一个进制转换算法，遵循递归三定律将是必须的，也是算法的设计指南。

```
1 将原始数字简化为一系列单个数字。
2 使用查找法将单个数字转换为字符。
3 将单个字符连接在一起以形成最终结果。
```

改变数字状态并向基本情况靠近的方法是采用除法，用数字除以基数，当数字小于基数时停止运算，返回结果。比如 10 进制数 996，以 10 为进制，看转换后的结果。先整除 10，余数 6，商 99。余数小于 10，可以直接求得其字符 “6”，商 99 小于 996，在向基本情况靠近。接着再递归调用自身，把 99 除以 10 商 9 余 9，余数小于 10，转换为字符 “9”，继续计算再得到一个 “9”，最终得到 996 的 10 进制字符串 “996”。一旦能解决 10 进制，那么 2-16

进制就都没有问题了。

下面是整数到任意进制（2-16 进制）字符串的转换算法，BASESTR 中保存着不同数字对应的字符形式，大于 10 的数字用字符 A-F 来表示。

```
// num2str_rec.rs

const BASESTR: [&str; 16] = ["0","1","2","3","4","5","6","7",
                             "8","9","A","B","C","D","E","F"];

fn num2str_rec(num: i32, base: i32) -> String {
    if num < base {
        BASESTR[num as usize].to_string()
    } else {
        // 余数加在末尾
        num2str_rec(num/base, base) +
            BASESTR[(num % base) as usize]
    }
}

fn main() {
    let num = 100;
    let sb = num2str_rec(num,2);
    let so = num2str_rec(num,8);
    let sh = num2str_rec(num,16);
    println!("{num} is b{sb}, o{so}, x{sh}");
}
```

前面我们用栈实现了数字到任意进制的转换，这里用递归再次实现了类似功能，这说明栈和递归是有关系的。实际上可以把递归看成是栈，只是这个栈是由编译器为我们隐式调用的，代码中只用了递归，但编译器使用了栈来保存数据。

上面的递归代码改用栈来实现的话，应该是下面这样的代码，和递归实现的代码结构非常相似。

```
// num2str_stk.rs

fn num2str_stk(mut num: i32, base: i32) -> String {
    let digits: [&str; 16] = ["0","1","2","3","4","5","6","7",
                              "8","9","A","B","C","D","E","F"];

    let mut rem_stack = Stack::new();
```

```
8       while num > 0 {
9           if num < base {
10              rem_stack.push(num); // 不超过 base 直接入栈
11          } else { // 超过 base 余数入栈
12              rem_stack.push(num % base);
13          }
14          num /= base;
15      }
16
17      // 出栈余数并组成字符串
18      let mut numstr = "".to_string();
19      while !rem_stack.is_empty() {
20          numstr += digits[rem_stack.pop().unwrap() as usize];
21      }
22
23      numstr
24  }
25
26  fn main() {
27      let num = 100;
28      let sb = num2str_stk(100, 2);
29      let so = num2str_stk(100, 8);
30      let sh = num2str_stk(100, 16);
31      println!("{num} is b{sb}, o{so}, x{sh}");
32  }
```

4.2.3 汉诺塔

汉诺塔是由法国数学家爱德华 · 卢卡斯在 1883 年发明的。他的灵感来自一个传说。一个印度教寺庙，将谜题交给年轻的牧师。在开始的时候，牧师们被给予三根杆和 64 个金碟，每个盘比它下面一个小一点。他们的任务是将所有 64 个盘子从三个杆中一个转移到另一个。但是移动是有限制的，一次只能移动一个盘子，且大盘子不能放在小的盘子上面，换句话说，小盘子始终在上，下面的盘子更大。牧师门日夜不停，每秒钟移动一块盘子。当他们完成工作时，传说，寺庙会变成灰尘，世界将消失。

实际上移动 64 个盘子的汉诺塔所需的时间是 $2^{64} - 1 = 18446744073709551,615$ s = 5850 亿年，超过了已知宇宙存在的时间 138 亿年。

图4.2展示了盘从第一杆移动到第三杆的示例。请注意，如规则指定，每个杆上的盘子都被堆叠起来，以使较小的盘子始终位于较大盘的顶部，这看起来类似一个栈。如果你以前没

有玩儿过这个，你现在可以尝试下。不需要盘子，一堆砖，书或纸都可以。你可以试试是不是真的那么费时。

如何用递归解决这个问题呢？首先回想递归三定律，找出基本情况是什么？假设有一个汉诺塔，有左中右三根杆，五个盘子在左杆上。如果你已经知道如何将四个盘子移动到中杆上，那么可以轻松地将最底部的盘子移动到右杆，然后再将四个盘子从中杆移动到右杆。但是如果不知道如何移动四个盘子到中杆怎么办？这时可以假设你知道如何移动三个盘子到右杆，那么很容易将第四个盘子移动到中杆，并将右杆上盘子移动到中杆盘子的顶部。但是还不知道如何移动三个盘子呢？这时再假设知道何将两个盘子移动到中杆，接着将第三个盘子移动到右杆，然后再移动两个盘子到它的顶部？可是两个盘子的移动仍然不知道，所以再假设你知道移动一个盘子到右。这看起来就像是基本情况。实际上，上面的描述虽然绕得很，但这个过程其实就是移动盘子的过程的抽象，最基本的情况就是移动一个盘子。

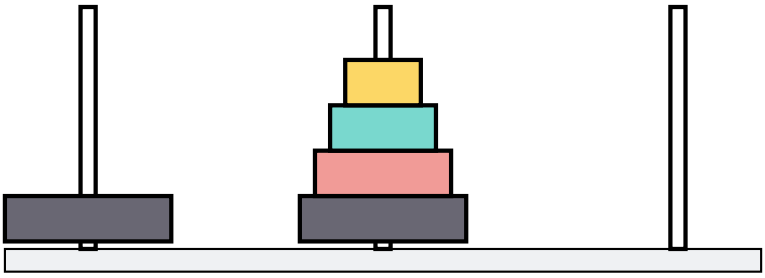


图 4.2: 汉诺塔

将上面的操作过程抽象描述整理为以下算法。

```
1 借助目标杆将 height - 1 个盘子移动到中间杆
2 将最后一个盘子移动到目标杆
3 借助起始杆将 height-1 个盘子从中间杆移动到目标杆。
```

只要遵守移动规则，较大的盘子保留在栈的底部，可以使用递归三定律来处理任何更多的盘子。最简单的汉诺塔就是只有一个盘子的塔，在这种情况下，只需要将盘子移动到其最终目的地就可以了，这就是基本情况。此外，上述算法在步骤 1 和 3 中减小了汉诺塔的高度，使汉诺塔趋向基本情况。下面是使用递归解决汉诺塔问题的 Rust 代码，总共也没几行代码。

```
1 // move2tower.rs
2
3 // p: pole 杆
4 fn move2tower(height: u32, src_p: &str,
5                des_p: &str, mid_p: &str) {
6     if height >= 1 {
7         move2tower(height - 1, src_p, mid_p, des_p);
8         println!("moving disk from {src_p} to {des_p}");
9         move2tower(height - 1, mid_p, des_p, src_p);
```

```
10      }
11  }
12
13  fn main() {
14      move2tower(1, "A", "B", "C");
15      move2tower(2, "A", "B", "C");
16      move2tower(3, "A", "B", "C");
17      move2tower(4, "A", "B", "C");
18  }
```

你可以用几张纸和三根笔来模拟盘在汉诺塔上的移动过程，并按照 height 等于 1，2，3，4 时 move2tower 的输出去移动纸，看看最终是否能将所有纸移动到某一个杆上。

4.3 尾递归

前面的递归计算称为普通递归，它需要在计算过程中保存值，如代码中的 first 和 last。我们知道，函数调用参数会保存在栈上，如果递归调用过多，栈会非常深。而内存又是有限的，所以递归存在爆栈的情况。如果这些中间值不单独处理，而是直接传递给递归函数作为参数，在参数传递时先计算，那么栈的内存消耗不会猛增，代码也会更简洁。因为所有参数都放到递归函数里，且函数的最后一行一定是递归，所以又叫尾递归。

尾递归就是把当前的运算结果如 first，last 等变量处理过后再直接当成递归函数的参数用于下次调用，深层递归函数所面对的参数是前面多个子问题的和，这个和形式上看起来该越来越复杂，但因为参数的和可以先求出来，又相当于减小了问题的规模，优化了算法，所有又叫尾递归优化。所以上面的普通递归版 nums 求和代码可以改成下面的尾递归形式。

```
1   // nums_sum34.rs
2
3   fn nums_sum3(sum: i32, nums: &[i32]) -> i32 {
4       if 1 == nums.len() {
5           sum + nums[0]
6       } else {
7           // 使用 sum 来接收中间计算值
8           nums_sum3(sum + nums[0], &nums[1..])
9       }
10  }
11
12  fn nums_sum4(sum: i32, nums: &[i32]) -> i32 {
13      if 1 == nums.len() {
14          sum + nums[0]
```

```
15      } else {
16          nums_sum4(sum + nums[nums.len() - 1],
17                    &nums[..nums.len() - 1])
18      }
19  }
20
21  fn main() {
22      let nums = [2,1,7,4,5];
23      let sum1 = nums_sum3(0, &nums);
24      let sum2 = nums_sum4(0, &nums);
25      println!("sum1 is {sum1}, sum2 is {sum2}");
26  }
```

尾递归代码是简洁了，但是看起来更不好懂了。因为子问题的结果需要直接当成参数输入下一次递归调用。从上面的代码也可以看出，函数最后一行才是递归调用，且参数是相加的形式，较为复杂。所以具体的递归程序实现要看程序员个人，如果尾递归不会爆栈，而且自己能写得清晰明了，那么使用尾递归就没有任何问题。

4.3.1 递归和迭代

计算 [2,1,7,4,5] 这个数列，我们采用了循环迭代、递归、尾递归三种代码，可见递归和迭代能实现相同的目的。那么递归和迭代的有没有什么关系呢?

递归：用来描述以自相似方法重复事务的过程，在数学和计算机科学中，指的是在函数定义中使用函数自身的方法。

迭代：重复反馈过程的活动，每一次迭代的结果会作为下一次迭代的初始值。

递归调用展开的话，是一个类似树的结构。从字面意思可以理解为重复递推和回溯的过程，当递推到达底部时就会开始回溯，其过程相当于树的深度优先遍历。迭代是一个环结构，从初始状态开始，每次迭代都遍历这个环，并更新状态，多次迭代直到结束状态。所有的迭代都可以转换为递归，但递归不一定可以转换成迭代。毕竟环改成树一定可以，但树改成环却未必能行。

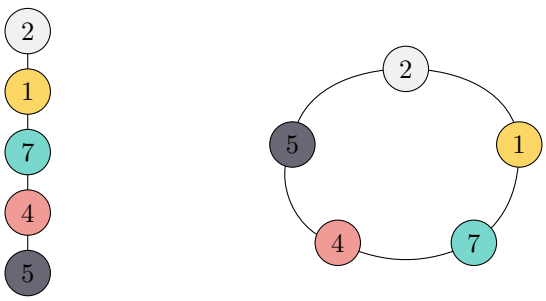


图 4.3: 递归和迭代

4.4 动态规划

计算机科学中有许多求最值的问题，例如滴滴顺风车要规划两个地点的最短路线，要找到最适合的几段路，或找到满足某些标准的最小道路集，这对于实现双碳目标、节约能源以及提升乘客体验都是至关重要的。本节的目标是向你展示几种求最值问题的不同解决方案，并表明动态规划是解决该类最优化问题的好办法。

优化问题的一个典型例子是使用最少的纸币/硬币来找零，这在地铁购票，自动售货机上很常见。我们希望每个交易返回最少的纸币张数，比如找零六元这笔交易，直接给一张五元加一张一元的共两张纸币，而不是向顾客吐出六张一元纸币。

现在的问题是：怎么从一个总任务，比如找零六元，规划出该怎么返回不同面值的纸币。最直观的就是从最大额的纸币开始找零，先尽可能使用大额纸币，然后再去找下一个小一点的纸币，并尽可能多的使用它们，直到完成找零。这种方法被称为贪婪方法，因为总是试图尽快解决大问题。

使用中国，美国货币时，这套贪婪法工作正常。但假设你决定在埃尔博尼亚部署自动贩卖机，除了通常的 1，5，10，25 分硬币，他们还有一个 21 分硬币。在这种情况下，贪婪法找不到找零 63 分的最佳解决方案。随着 21 分硬币的加入，贪婪法仍然会找到一个解决方案，但却有六个硬币（25x2, 10x1, 1x3），然而最佳答案是三个 21 分。

如果采用递归方法，那么找零问题能很好地解决。首先要找准基本问题，那就是找零一个币，面额不定但是恰好等于要找零的金额，因为刚好找零只用一个币，数量是除零之外的最小值了。如果金额不匹配，可以设置多个选项。为便于读者理解，假如使用中国纸币来找零，有 1，5，10，20, 50 元的纸币。对于用一元来找零的情况，找零纸币数量等于一加上总找零金额减去一元后所需的找零纸币数，对于 5 元的，是一加上原始金额减去五元后金额所需的纸币数量，如果是 10 元，则是一加上总金额减去十元后所需的找零数量等等。因此，对原始金额找零纸币数量可以根据下式计算:

$$numCoins(amount) = \begin{cases} 1 + numCoins(amount - 1) \\ 1 + numCoins(amount - 5) \\ 1 + numCoins(amount - 10) \\ 1 + numCoins(amount - 20) \\ 1 + numCoins(amount - 50) \end{cases} \tag{4.2}$$

numCoins 计算找零纸币数量，amount 为找零金额。按照找零纸币的不同面额，找零任务分为了五种情况。具体算法如下，在第 3 行，先检查基本情况，也就是说，当前找零额度是否和某个纸币面值等额。如果没有等额的纸币则递归调用小于找零额的不同面额的情况，此时问题规模减小了。注意，递归调用前先加 1，说明计算了当前正在使用的一张面额的纸币，因为要求的就是纸币数量。

```
1 // rec_mc1.rs
2
```

```
3  fn rec_mc1(cashes: &[u32], amount: u32) -> u32 {
4      // 全用 1 元纸币时的最少找零纸币数
5      let mut min_cashes = amount;
6
7      if cashes.contains(&amount) {
8          return 1;
9      } else {
10         // 提取符合条件的币种 (找零的币值肯定要小于找零值)
11         for c in cashes.iter()
12                         .filter(|&c| *c <= amount)
13                         .collect::<Vec<&u32>>() {
14
15             // amount 减去 c, 表示使用了一张面额为 c 的纸币
16             // 所以要加 1
17             let num_cashes = 1 + rec_mc1(&cashes, amount - c);
18
19             // num_cashes 若比 min_cashes 小则更新
20             if num_cashes < min_cashes {
21                 min_cashes = num_cashes;
22             }
23         }
24     }
25
26     min_cashes
27 }
28
29 fn main() {
30     // cashes 保存各种面额的纸币
31     let cashes = [1,5,10,20,50];
32     let amount = 31u32;
33     let cashes_num = rec_mc1(&cashes, amount);
34     println!("need refund {cashes_num} cashes");
35 }
```

你可以将 31 改成 81，然后你发现程序没有结果返回。实际上，程序需要非常多的递归调用来找到 4 张纸币：[50，20, 10, 1] 的组合。要理解递归方法中的缺陷，可以看看下面这个函数递归调用。其中用了很多重复计算来找到正确的支付组合。程序运行中计算了大量没用的组合，然后才能返回唯一正确的答案。图中的下划线加数字表示这个找零金额出现的次数，比如 11_2 表明找零 11 元这个任务第二次出现了。

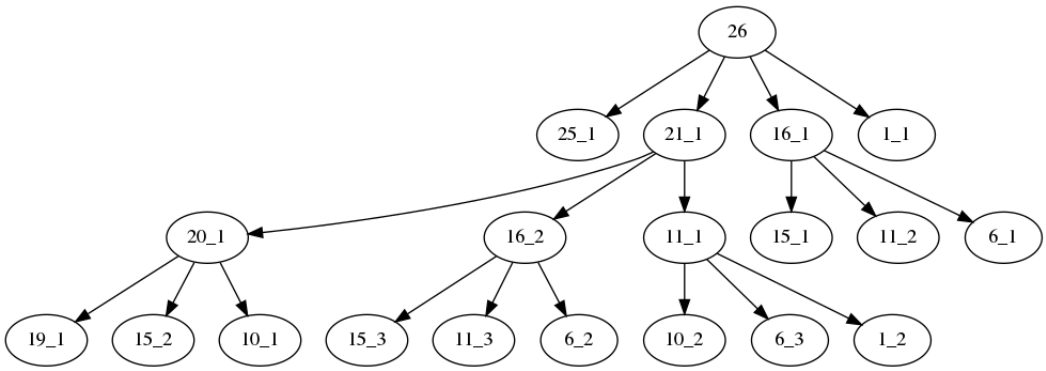


图 4.4: 递归求解找零任务

要减少程序的工作量，关键是要记住过去已经计算过的结果，这样可以避免重复计算。一个简单的解决方案是将当前最小数量纸币值存储在切片中。然后在计算新的最小值之前，首先查切片，看看结果是否存在。如果已经有结果了就直接使用切片中的值，而不是重新计算。这是算法设计中经常出现的用空间换时间的例子。

```
1  // rc_mc2.rs
2
3  fn rec_mc2(cashes: &[u32],
4             amount: u32,
5             min_cashes: &mut [u32]) -> u32 {
6      // 全用 1 元纸币的最小找零纸币数量
7      let mut min_cashe_num = amount;
8
9      if cashes.contains(&amount) {
10         // 收集和当前待找零值相同的币种
11         min_cashes[amount as usize] = 1;
12         return 1;
13     } else if min_cashes[amount as usize] > 0 {
14         // 找零值 amount 有最小找零纸币数，直接返回
15         return min_cashes[amount as usize];
16     } else {
17         for c in cashes.iter()
18                        .filter(|c| *(*c) <= amount)
19                        .collect::<Vec<&u32>>() {
20             let cashe_num = 1 + rec_mc2(cashes,
21                                         amount - c,
22                                         min_cashes);
```

```
23
24                  // 更新最小找零纸币数
25                  if cashe_num < min_cashe_num {
26                      min_cashe_num = cashe_num;
27                      min_cashes[amount as usize] = min_cashe_num;
28                  }
29              }
30          }
31
32          min_cashe_num
33      }
34
35      fn main() {
36          let amount = 81u32;
37          let cashes: [u32; 5] = [1,5,10,20,50];
38          let mut min_cashes: [u32; 82] = [0; 82];
39          let cashe_num = rec_mc2(&cashes, amount, &mut min_cashes);
40          println!("need refund {cashe_num} cashes");
41      }
```

新的 rec_mc 的计算就没有那么耗时了，因为使用了变量 min_cashes 来保存中间值。本节是讲动态规划，然而这两个程序都是递归而非动态规划，第二个程序只是在递归中保存了中间值，是一种记忆手段或者缓存。

4.4.1 什么是动态规划

动态规划（dynamic programming，DP）是运筹学的一个分支，是求解决策过程最优化的数学方法。上面的找零就属于最优化问题，求的是最小纸币数量，数量就是优化目标。

动态规划是对某类问题的解决方法，重点在于如何鉴定一类问题是动态规划可解的而不是纠结用什么解决方法。动态规划中状态是非常重要的，它是计算的关键，通过状态的转移来实现问题的求解。当尝试使用动态规划解决问题时，其实就是要思考如何将这个问题表达成状态以及如何在状态间转移。

前文的贪婪算法是从大到小去凑值，这种算法很笨。而动态规划总是假设当前已取得最好结果，再依据此结果去推导下一步行动。递归法将大问题分解为小问题，调用自身。而动态规划从小问题推导到大问题，推导过程的中间值要缓存起来，这个推导过程称为状态转移。比如找零问题，动态规划先求出找零一元所需要纸币数并保存，那么两元找零问题等于两个一元找零问题，计算得到的值保存起来，接着是三元找零问题，等于两元找零加一元找零，此时查表可得到具体值。通过这种从小到大的步骤，可以逐步构建出任何金额的找零问题。

对上面的找零问题，如果用动态规划，那么需要三个参数：可用纸币列表 cs，找零金额

amount, 一个包含各个金额所需最小找零纸币数量的列表 min_cs。当函数完成计算时, 列表内将包含从零到找零值的所有金额所需的最小找零纸币数量。下面是实现的算法，可以看到动态规划使用了迭代。

```
// dp_rec_mc.rs

fn dp_rec_mc(cashes: &[u32], amount: u32,
             min_cashes: &mut [u32]) -> u32 {
    // 动态收集从 1 到 amount 的最小找零币值数量
    // 然后从小到大凑出找零纸币数量
    for denm in 1..=amount {
        let mut min_cashe_num = denm;
        for c in cashes.iter()
                       .filter(|&c| *c <= denm)
                       .collect::<Vec<&u32>>() {
            let index = (denm - c) as usize;

            let cashe_num = 1 + min_cashes[index];
            if cashe_num < min_cashe_num {
                min_cashe_num = cashe_num;
            }
        }
        min_cashes[denm as usize] = min_cashe_num;
    }

    // 因为收集了各个值的最小找零纸币数，所以直接返回
    min_cashes[amount as usize]
}

fn main() {
    let amount = 81u32;
    let cashes = [1,5,10,20,50];
    let mut min_cashes: [u32; 82] = [0; 82];
    let cash_num = dp_rec_mc(&cashes,amount,&mut min_cashes);
    println!("Refund for ${amount} need {cash_num} cashes");
}
```

动态规划代码是迭代，比递归代码简洁不少，不像前两个递归版本算法，它减少了栈的使用。但要意识到，能为一个问题写递归解决方案并不意味着它就是最好的的解决方案。

虽然上面的动态规划算法找出了所需纸币最小数量，但它不显示到底是哪些面额的纸币。如果希望得到具体的面额，可以扩展该算法，使之记住具体使用的纸币面额及其数量。为此需要添加一个记录使用纸币的表 cashes_used。只需记住为每个金额添加它所需的最后一张纸币的金额到该列表，然后不断地在列表中找前一个金额的最后一张纸币，直到结束。

```
// dp_rc_mc_show.rs

// 使用 cashes_used 收集使用过的各面额纸币
fn dp_rec_mc_show(cashes: &[u32], amount: u32,
    min_cashes: &mut [u32], cashes_used: &mut [u32]) -> u32 {
    for denm in 1..=amount {
        let mut min_cashe_num = denm ;
        let mut used_cashe = 1; // 最小面额是 1 元
        for c in cashes.iter()
                        .filter(|&c| *c <= denm)
                        .collect::<Vec<&u32>>() {
            let index = (denm - c) as usize;
            let cashe_num = 1 + min_cashes[index];
            if cashe_num < min_cashe_num {
                min_cashe_num = cashe_num;
                used_cashe = *c;
            }
        }

        // 更新各金额对应的最小纸币数
        min_cashes[denm as usize] = min_cashe_num;
        cashes_used[denm as usize] = used_cashe;
    }

    min_cashes[amount as usize]
}

// 打印输出各面额纸币
fn print_cashes(cashes_used: &[u32], mut amount: u32) {
    while amount > 0 {
        let curr = cashes_used[amount as usize];
        println!("${curr}");
        amount -= curr;
    }
```

```
35 }
36
37 fn main() {
38     let amount = 81u32; let cashes = [1,5,10,20,50];
39     let mut min_cashes: [u32; 82] = [0; 82];
40     let mut cashes_used: [u32; 82] = [0; 82];
41     let cs_num = dp_rec_mc_show(&cashes, amount,
42                                 &mut min_cashes,
43                                 &mut cashes_used);
44     println!("Refund for ${amount} need {cs_num} cashes:");
45     print_cashes(&cashes_used, amount);
46 }
```

4.4.2 动态规划与递归

递归是一种调用自身，通过分解大问题为小问题以解决问题的技术，而动态规划则是一种利用小问题解决大问题的技术。递归费栈，容易爆内存。动态规划则不好找准转移规则和起始条件，而这两点又是必须的，所以动态规划好用，不好理解，代码很简单，理解很费劲儿。同样的问题，有时递归和动态规划都能解决，比如斐波那契数列问题，用两者都能解决。

```
1  // dp_rec.rs
2
3  fn fibnacci_rec(n: u32) -> u32 {
4      if n == 1 || n == 2 {
5          return 1;
6      } else {
7          fibnacci(n-1) + fibnacci(n-2)
8      }
9  }
10
11 fn fibnacci_dp(n: u32) -> u32 {
12     // 只用两个位置来保存值，节约内存
13     let mut dp = [1, 1];
14
15     for i in 2..=n {
16         let idx1 = (i % 2) as usize;
17         let idx2 = ((i - 1) % 2) as usize;
18         let idx3 = ((i - 2) % 2) as usize;
19         dp[idx1] = dp[idx2] + dp[idx3];
```

```
20        }
21
22        dp[((n-1) % 2) as usize]
23    }
24
25    fn main() {
26        println!("fib 10: {}", fibnacci(10));
27        println!("fib 10: {}", fibnacci_dp(10));
28    }
```

注意，能用递归解决的，用动态规划不一定都能解决。因为这两者本身就是不同的方法，动态规划需要满足的条件，递归时不一定能满足。这一点一定牢记，不要为了动态规划而动态规划。

4.5 总结

在本章中我们讨论了递归算法和迭代算法。所有递归算法都必须要满足三定律，递归在某些情况下可以代替迭代，但迭代不一定能替代递归。递归算法通常可以自然地映射到所解决的问题的表达式，看起来很直观简洁。递归并不总是好的方案，有时递归解决方案可能比迭代算法在计算上更昂贵。尾递归是递归的优化形式，能一定程度上减少栈资源使用。动态规划可用于解决最优化问题，通过小问题逐步构建大问题，而递归是通过分解大问题为小问题来逐步解决。

Chapter 5

查找

5.1 本章目标

能够实现顺序查找，二分查找
理解哈希作为查找技术的思想
使用 Vec 实现 HashMap 抽象数据类型

5.2 查找

在实现了多种数据结构（栈，队列，链表）后，现在开始利用这些数据结构来解决一些实际问题，即查找和排序问题。查找和排序是计算机科学的重要内容，大量的软件和算法都是围绕这两个任务来开展的。回忆你自己使用过的各种软件的查找功能，应该不陌生。比如 Word 文档里你用查找功能来找到某些字符，在 google 浏览器搜索栏你键入要搜索的内容，然后搜索引擎返回查找到的数据。搜索和查找是一个意思，本书不加区分。

查找是在项集合中找到特定项的过程，查找通常对于项是否存在返回 true 或 false，有时它也返回项的位置。在 Rust 中，有一个非常简单的方法来查询一个项是否在集合中，那就是 contains() 函数。该函数字面理解是是否包含某数据的意思，但其实就是查找（search, find）。

```
1 fn main() {
2     let data = vec![1,2,3,4,5];
3     if data.contains(&3) {
4         println!("Yes");
5     } else {
6         println!("No");
7     }
8 }
```

这种查找算法很容易写，Vec 的 contains 函数操作替我们完成了查找工作。查找算法不只有一种，事实上有很多不同的方法来进行查找，包括顺序查找，二分查找，哈希查找。我们感兴趣的是这些不同的查找算法如何工作以及它们的性能、复杂度如何。

5.3 顺序查找

当数据项存储在诸如 Vec，数组，切片这样的集合中时，数据具有线性关系，因为每个数据项都存储在相对于其他数据项的位置。在切片中，这些相对位置是数据项的索引值。由于索引值是有序的，可以按顺序访问，所以这样的数据结构也是线性的。回想前面学习的栈，队列，链表都是线性的。基于这种和物理世界相同的线性逻辑，一种很自然的查找技术就是线性查找，或者叫顺序查找。

下图展示了这种查找的工作原理。查找从切片中的第一个项目开始，按照顺序从一个项移动到另一个项，直到找到目标所在项或遍历完整个切片。如果遍历完整个切片后还没找到，则说明待查找的项不存在。

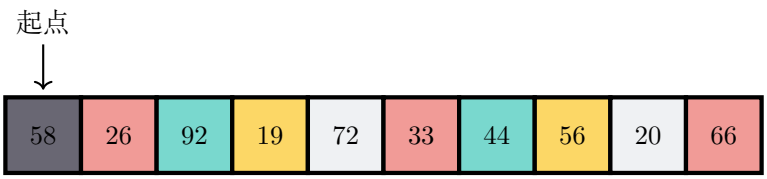


图 5.1: 顺序查找

5.3.1 Rust 实现顺序查找

下面是 Rust 实现的线性查找代码，整个程序非常直观。

```
1  // sequential_search.rs
2
3  fn sequential_search(nums: &[i32], num: i32) -> bool {
4      let mut pos = 0;
5      let mut found = false;
6
7      // found 表示是否找到
8      // pos 在索引范围内且未找到就继续循环
9      while pos < nums.len() && !found {
10         if num == nums[pos] {
11             found = true;
12         } else {
13             pos += 1;
```

```
14          }
15      }
16
17      found
18  }
19
20  fn main() {
21      let num = 8;
22      let nums = [9,3,7,4,1,6,2,8,5];
23      let found = sequential_search(&nums, num);
24      println!("{num} is in nums: {found}");
25  }
```

当然，顺序查找也可以返回查找项的具体位置，如果没有就返回 None，所以返回值类型我们使用的 Option。

```
1   // sequential_search_pos.rs
2
3   fn sequential_search_pos(nums: &[i32], num: i32)
4       -> Option<usize> {
5       let mut pos: usize = 0;
6       let mut found = false;
7
8       while pos < nums.len() && !found {
9           if num == nums[pos] {
10              found = true;
11          } else {
12              pos += 1;
13          }
14      }
15
16      if found {
17          Some(pos)
18      } else {
19          None
20      }
21  }
22
23  fn main() {
```

```
24      let num = 8;
25      let nums = [9,3,7,4,1,6,2,8,5];
26      match sequential_search_pos(&nums, num) {
27          Some(pos) => println!("index of {num} is {pos}"),
28          None => println!("{num} is not in nums"),
29      }
30  }
```

5.3.2 顺序查找复杂度

为了分析顺序查找算法复杂度，需要设定一个基本计算单位。对于查找来说，比较操作是主要的操作，所以统计比较的次数是最重要的。数据项是随机放置，无序的，每次比较都有可能找到目标项。从概率论角度来说，数据项在集合中任何位置的概率是一样的。

如果目标项不在集合中，知道这个结果的唯一方法是将目标与集合中所有数据项进行比较。如果有 n 个项，则顺序查找需要 n 次比较，此时的复杂度是 O(n)。如果目标项在集合中，那么复杂度是多少呢？还是 O(n) 吗？

这种情况下，分析不如前一种情况那么简单。实际上有三种不同的可能。最好的情况下，目标就在集合开始处，只需要比较一次就找到目标了，此时复杂度是 O(1)，最差的情况是目标在最后，要比较 n 次才能知道，此时复杂度为 O(n)，除此之外，目标分布在中间，且任何位置的概率相同。此时的复杂度有 n - 2 种可能，目标在第二位，则复杂度为 O(2)，直到 O(n-1)。综合来看，当目标项在集合中时，查找可能比较的次数在 1 至 n 次，其复杂度平均来说等于所有可能的复杂度之和除以总的次数。

$$\sum_{i=1}^{n} O(i)/n = O(n/2) = O(n) \tag{5.1}$$

当 n 很大时，1/2 可以不考虑，可见随机序列的顺序查找复杂度就是 O(n)。假设数据项是无序的，得到的复杂度是 O(n)，如果数据不是随机放置而是有序的，是否查找性能要好一些呢？假设数据集合按升序排列，且假设目标项存在于集合中，且在 n 个位置上的概率依旧相同。如果目标项不存在，则可以通过一些技巧来加快查找的速度。下图展示了这个过程。假如查找目标项 50。此时，比较按顺序进行，直到 56。此时，可以确定的是后面一定没有目标值 50 了，因为是升序排序，后面的项比 54 还大，所以不会有 50，算法停止查找。

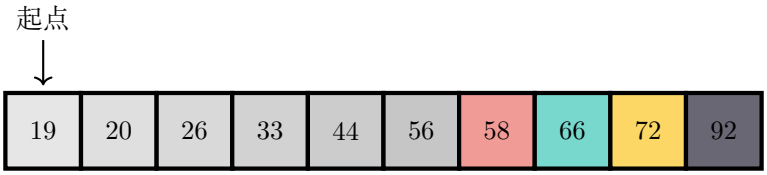


图 5.2: 有序集合顺序查找

下面是用在已排序数据集上的顺序查找算法，通过设置 stop 变量来控制查找超出范围时立即停止查找以节约时间，算是对算法进行了一次优化。

```rust
// ordered_sequential_search.rs

fn ordered_sequential_search(nums: &[i32], num: i32) -> bool {
    let mut pos = 0;
    let mut found = false;
    let mut stop = false; // 控制遇到有序数据时退出

    while pos < nums.len() && !found && !stop {
        if num == nums[pos] {
            found = true;
        } else if num < nums[pos] {
            stop = true; // 数据有序，退出
        } else {
            pos += 1;
        }
    }

    found
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let num = 44;
    let found = ordered_sequential_search(&nums, num);
    println!("{num} is in nums: {found}");

    let num = 49;
    let found = ordered_sequential_search(&nums, num);
    println!("{num} is in nums: {found}");
}
```

因为数据排序好了，所以数据项不在集合中时，如果小于第一项，那么比较一次就知道结果了，最差是比较 n 次，平均比较 n/2 次，复杂度还是 O(n)。但是这个 O(n) 比无序的查找好，因为大多数情况的查找符合平均情况，而平均情况的复杂度，有序数据集查找能提升一倍速度。可见，排序对于提升查找复杂度很有帮助，所以排序也是计算机科学里的重要议

题，我们将在下一章中学习各种排序算法。综合无序和有序数据集合的顺序查找，可得下表。

表 5.1: 顺序查找复杂度

情况	最少比较次数	平均比较次数	最多比较次数	查找类型
目标存在	1	$\frac{n}{2}$	n	无序查找
目标不存在	n	n	n	无序查找
目标存在	1	$\frac{n}{2}$	n	有序查找
目标不存在	1	$\frac{n}{2}$	n	有序查找

5.4 二分查找

有序数据集合对于查找算法是很有利的。在有序集合中顺序查找时，当与第一个项进行比较，如果第一项不是要查找的，则还有 n-1 项待比较。当遇到超过范围的值时，可以停止查找，综合来看这种有序查找速度还是比较慢。对于排好序的数据集有没有更快的查找算法呢? 当然有，那就是二分查找，这是一个非常重要的查找算法，下面来仔细分析并用 Rust 实现二分查找算法。

5.4.1 Rust 实现二分查找

其实二分查找的意思体现在其名字上了，说白了就是把数据集分成两部分来查找，通过 low，mid，high 来控制查找的范围。从中间项 mid 开始，而不是按顺序查找。如果中间项是正在寻找的项，则完成了查找。如果它不是，可以使用排序集合的有序性质来消除一半剩余项。如果正在查找的项大于中间项，就可以消除中间项以及比中间项小的一半元素，也就是第一项到中间项都可以不用去比较了。因为目标项大于中间值，那么不管是否在集合中，那肯定不会在前一半。相反，若是目标项小于中间项，则后面的一半数据就都可以不用比较了。去除了一半数据后在剩下的一半数据项里再查看其中间项，重复上述的比较和省略过程，最终得到结果。这个查找速度是比较快的，而且也非常形象，所以叫二分查找。

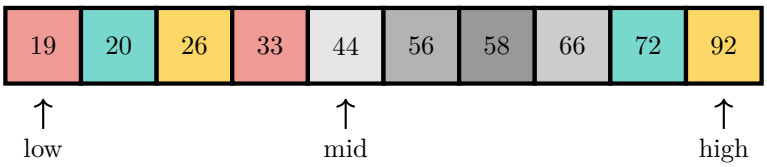


图 5.3: 二分查找

二分查找示意图如上图。初始时 low 和 high 分居最左和最右。比如你要查 60，那么先和中间值比较，大于 44，则 low 移动到 56 处，mid 移动到 66。此时和 66 比较，发现小于 66，所以 high 移动到 58，mid 移动到 56 和 low 重合了。比较发现大于 56，所以 low 移动到 58，mid 也移动到 58，发现大于 58。此时 low, high, mid 三者在一个位置。接着 low 移

动到 66，发现下标大于 high 了，不满足条件，查找停止，退出。根据上面的描述和示意图，可以用 Rust 写出如下的二分查找代码。

```rust
// binary_search.rs
fn binary_search1(nums: &[i32], num: i32) -> bool {
    let mut low = 0;
    let mut high = nums.len() - 1;
    let mut found = false;

    // 注意是 <= 不是 <
    while low <= high && !found {
        let mid: usize = (low + high) >> 1;

        // 若 low + high 可能溢出，可转换为减法
        //let mid: usize = low + ((high - low) >> 1);

        if num == nums[mid] {
            found = true;
        } else if num < nums[mid] {
            high = mid - 1; // num 小于中间值，省去后半部数据
        } else {
            low = mid + 1;  // num 大于等于中间值，省去前半部数据
        }
    }

    found
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let target = 3;
    let found = binary_search1(&nums, target);
    println!("{target} is in nums: {found}");

    let target = 63;
    let found = binary_search1(&nums, target);
    println!("{target} is in nums: {found}");
}
```

二分法其实是把大问题分解成小问题，采取的是分而治之策略。前面我们学习了分解大问题为小问题用递归来解决，所以二分法和递归是否有相似，二分法是否能用递归实现呢?

我们发现二分查找时，找到或没找到是最终的结果，是一个基本情况。而二分法不断减小问题的尺度，不断向基本情况靠近，且二分法是在不断重复自身步骤。所以二分法满足递归三定律，所以可以用递归实现二分法，具体实现如下。

```rust
// binary_search.rs

fn binary_search2(nums: &[i32], num: i32) -> bool {
    // 基本情况1: 项不存在
    if 0 == nums.len() { return false; }

    let mid: usize = nums.len() >> 1;

    // 基本情况2: 项存在
    if num == nums[mid] {
        return true;
    } else if num < nums[mid] {
        // 减小问题规模
        return binary_search2(&nums[..mid], num);
    } else {
        return binary_search2(&nums[mid+1..], num);
    }
}

fn main() {
    let nums = [1,3,8,10,15,32,44,48,50,55,60,62,64];

    let num = 3;
    let found = binary_search2(&nums, num);
    println!("{num} is in nums: {found}");

    let num = 63;
    let found = binary_search2(&nums, num);
    println!("{num} is in nums: {found}");
}
```

递归实现的查找涉及数据集合的切片，注意再查找时 mid 项需要舍去，所以使用了 mid + 1。递归算法总是涉及栈的使用，有爆栈风险，一般来说二分查找最好用迭代法来解决。

5.4.2 二分查找复杂度

二分查找算法最好的情况是中间项就是目标，此时复杂度为 O(1)。二分查找每次比较能消除一半的剩余项，可以计算最多的比较次数得到该算法的最差复杂度。第一次比较后剩 $n/2$，第二次比较后剩余 $n/4$，直到 $n/8$，$n/16$，$n/2^i$ 等等。当 $n/2^i = 1$ 时，二分结束。所以

$$\begin{aligned} \frac{n}{2^i} &= 1 \\ i &= log_2(n) \end{aligned} \tag{5.2}$$

所以二分查找算法最多比较 $log_2(n)$ 次，复杂度也就是 $O(log_2(n))$，这是一个比 O(n) 算法还要优秀的算法。但要注意，在上述的递归版实现中，默认的栈使用是会消耗内存的，复杂度不如迭代版。

二分查找看起来很好，但如果 n 很小，就不值得去排序再使用二分，此时直接使用顺序查找可能复杂度还更好。此外，对于很大数据集，对其排序很耗时和耗内存，那么直接采用顺序查找复杂度可能也更好。好在实际项目中大量的数据集即不多也不少，非常适合二分查找，这也是我们花大量篇幅来阐述二分查找算法的原因。

5.4.3 内插查找

内插查找是一种二分查找的变形，适合在排序数据中进行查找。如果数据是均分的，则可以使用内插查找快速逼近待搜索区域，从而提高效率。

内插查找不是像二分查找算法中那样直接使用中值来定界，而是通过插值算法找到上下界。回忆中学学过的线性内插法，给定直线两点 (x_0, y_0) 和 (x_1, y_2)，可以求出 $[x_0, x_1]$ 范围内任意点 x 对应的值 y 或者任意 y 对应的 x.

$$\begin{aligned} \frac{y - y_0}{x - x_0} &= \frac{y_1 - y_0}{x_1 - x_0} \\ x &= \frac{(y - y_0)(x_1 - x_0)}{y_1 - y_0} + x_0 \end{aligned} \tag{5.3}$$

比如要在 [1,9,10,15,16,17,19,23,27,28,29,30,32,35] 这个已排序的 14 个元素的集合中查找到元素 27，那么可以将索引当做 x 轴，元素值当做 y 轴。可知 $x_0 = 0, x_1 = 13$，而 $y_0 = 1, y_1 = 35$。所以可以计算 $y = 27$ 对应的 x 值。

$$\begin{aligned} x &= \frac{(27 - 1)(13 - 0)}{35 - 1} + 0 \\ x &= 9 \end{aligned} \tag{5.4}$$

查看 nums[9] 发现值为 28，大于 27，所以将 28 当做上界。28 的下标为 9，所以搜索 [0,8] 范围内的元素，继续执行插值算法。

$$\begin{aligned} x &= \frac{(27 - 1)(8 - 0)}{27 - 1} + 0 \\ x &= 8 \end{aligned} \tag{5.5}$$

查看 nums[8] 发现值恰为 27，找到目标，算法停止。具体实现代码如下。

```rust
// interpolation_search.rs

fn interpolation_search(nums: &[i32], target: i32) -> bool {
    if nums.is_empty() { return false; }

    let mut low  = 0usize;
    let mut high = nums.len() - 1;
    loop {
        let low_val  = nums[low];
        let high_val = nums[high];

        if high <= low || target < low_val
            || target > high_val {
            break;
        }

        // 计算插值位置
        let offset = (target - low_val)*(high - low) as i32
                   / (high_val - low_val);
        let interpolant = low + offset as usize;

        // 更新上下界 high、low
        if nums[interpolant] > target {
            high = interpolant - 1;
        } else if nums[interpolant] < target {
            low = interpolant + 1;
        } else {
            break;
        }
    }

    // 判断最终确定的上界是否是 target
    if target == nums[high] {
        true
    } else {
        false
    }
```

```
38 }
39
40 fn main() {
41     let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
42     let target = 27;
43     let found = interpolation_search(&nums, target);
44     println!("{target} is in nums: {found}");
45 }
```

仔细分析代码可以发现，除了 interpolant 的计算方式不同外，其他的和二分查找算法几乎一样。内插查找算法在数据均分时复杂度是 $O(loglog(n))$，具体证明较复杂，请看此论文 [10] 了解，最差和平均复杂度均是 $O(n)$。

5.4.4 指数查找

指数查找是另一种二分查找的变体，它划分中值的方法不是使用平均或插值而是用指数函数来估计，这样可以快速找到上界，加快查找，该算法适合已排序且无边界的数据。算法查找过程中不断比较 $2^0, 2^1, 2^2, 2^k$ 位置上的值和目标值的关系，进而确定搜索区域，之后在该区域内使用二分查找算法查找。

假设要在 [2,3,4,6,7,8,10,13,15,19,20,22,23,24,28] 这个 15 个元素已排序集合中查找 22，那么首先查看 $2^0 = 1$ 位置上的数字是否超过 22，得到 $3 < 22$，所以继续查找 $2^1, 2^2, 2^3$ 位置处元素，发现对应的值 4, 7, 15 均小于 22。继续查看 $16 = 2^4$ 处的值，可是 16 大于集合元素个数，超出范围了，所以查找上界就是最后一个索引 14。

下面是实现的指数搜索代码。注意 14 行的下界是 high 的一半，此处用的是移位操作。我们能找到一个上界，那么说明前一次访问处一定小于待查找的值，作为下界是合理的。当然用 0 作下界也可以，但是效率就低了。

```
1  // exponential_search.rs
2
3  fn exponential_search(nums: &[i32], target: i32) -> bool {
4      let size = nums.len();
5      if size == 0 { return false; }
6
7      // 逐步找到上界
8      let mut high = 1usize;
9      while high < size && nums[high] < target {
10         high <<= 1;
11     }
12
13     // 上界的一半一定可以作为下界
```

```
14        let low = high >> 1;
15
16        // 使用前面实现的二分查找
17        binary_search(&nums[low..size.min(high+1)], target)
18    }
19
20    fn main() {
21        let nums = [1,9,10,15,16,17,19,23,27,28,29,30,32,35];
22        let target = 27;
23        let found = exponential_search(&nums, target);
24        println!("{target} is in nums: {found}");
25    }
```

通过分析发现，指数查找分为两部分，第一部分是找到上界用于划分区间，第二部分是二分查找。划分区间的复杂度和查找目标 i 相关，其复杂度为 $O(logi)$，而二分查找时复杂度为 $O(logn)$，n 为查找区间长度，在这里可以知道区间长度为 $high - low = 2^{logi} - 2^{logi-1} = 2^{logi-1}$，所以复杂度为 $O(log(2^{logi-1})) = O(logi)$，最后总的复杂度为 $O(logi+logi) = O(logi)$

5.5 哈希查找

前面的查找算法都是利用项在集合中相对于彼此存储的位置信息来进行查找。通过排序集合，可以使用二分查找在对数时间内查找到数据项。这些数据项在集合中的位置信息由集合的有序性质提供，查找算法无从得知，所以要不断比较。

如果我们的算法能对不同的项保存地址有先验知识，那么查找时就不用依次比较，而是可以直接获取。这种通过数据项直接获得其保存地址的方法称为哈希查找 (Hash Search)，是一种复杂度为 O(1) 的查找算法，也就是最快的查找算法。

为了做到这一点，当在集合中查找项时，项地址要首先存在，所以我们得提供一个保存项且获取地址方便的数据结构，这就是哈希表 (散列表)。哈希表以容易找到数据项的方式存储着数据项，每个数据项位置通常称为一个槽 (地址)。这些槽可以从 0 开始命名，当然也可以是其他的数字。一旦选定首个槽的命名，那么接着所有的槽名都相应的加一。最初，哈希表不包含项，因此每个槽都为空。可以通过 Vec 来实现一个哈希表，每个元素初始化为 None。下图展示了大小 m = 11 的哈希表，换句话说，在表中有 m 个槽，命名为 0 到 10。

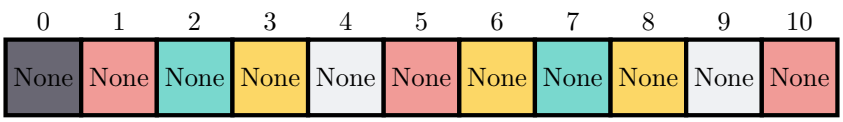
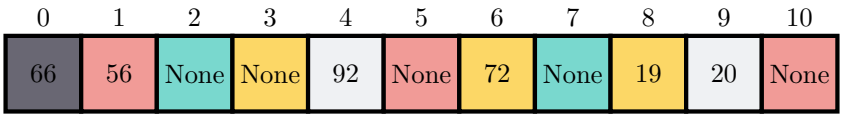


图 5.4: 哈希表

数据项及其在哈希表中所属槽之间的映射关系被称为 hash 函数或散列函数。hash 函数

接收集合中的任何项，并返回具体的槽名，这个动作称为哈希或散列。假设有整数项 [24, 61, 84, 41, 56, 31]，和一个容量为 11 的哈希槽，那么只要将每个项输入到哈希函数，就能得到它们在哈希表中的位置。一种简单的哈希函数就是求余，因为任何数对 11 求余，余数一定在 11 以内，也就是在 11 个槽范围内，这能保证数据总是有槽来存放。

$$hash(item) = item\%11$$



一旦计算了哈希值，就可以将项插入到指定位置的槽中，如上图所示。注意，11 个插槽中的 6 个现在已被占用，此时哈希表的负载可用占用的槽除以总槽数，该比值称为负载因子，用 λ 表示，其计算方式是 $\lambda =$ 项数/表大小，此处就是 $\lambda = 6/11$。这个负载因子可以作为评估指标，尤其是程序需要保存很多项时。若是负载因子太大，那么剩下的位置就不够，所以可以根据负载因子控制是否要扩容。在 Rust，Go 等语言中都是通过这种机制来扩容的，负载因子超过一个阈值，哈希表就开始扩容，为后面插入数据作准备。

从图中可见哈希表保存的数据不是有序的，相反，它是无序的，而且非常乱。我们有哈希函数，不管它怎么乱，都可以通过计算哈希值获取数据项的槽。比如要查询 66 是否存在，那么通过哈希计算，得到 hash(66) = 0，查看槽 0，发现为 66，所以 66 存在。该查找操作复杂度为 O(1)，因为只用在恒定时间算出槽位置并查看。

哈希表及哈希查找非常优秀，但也要注意冲突。假如现在加入 99，那么 hash(99) = 0，而槽 0 处是 66，不等于 99，此时哈希槽出现冲突。冲突必须解决，不然哈希表就无法使用。

5.5.1 哈希函数

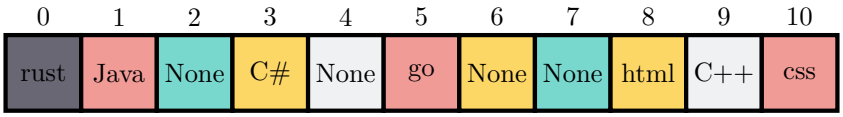
上节使用的哈希函数是直接对项求余，使得余数在一定范围内。可见一个算法只要能根据项求得一个在一定范围内的数，那么这个算法就可以看成是哈希函数。通过对哈希函数的改进，我们能减小冲突的概率，实现一个可用的哈希表。

第一种改进方法是分组求和法，它将项划分为相等大小的块，（最后一块可能不等），然后将这些项加起来再求余。例如，如果数据项是电话号码 316-545-0134，可以将号码分成两位数，不足补 0。那么可以得到 [31,65,45,01,34]，将其求和得 176，再对 11 求余得到哈希值（槽）为 0。当然，号码也可以分成三位数，甚至反转数字，最后再求余。哈希函数的种类非常多，因为只要能得出一个值再求余就行，而这个值可以采用各种方法得到。

分组求和法可以求哈希值，平法取中法也可以，这是另一种哈希算法。首先对数据项求平方，然后提取平方的中间部分作为值去求余。比如数字 36，平方为 1296，取中间部分 29，求余得到 hash(29) = 7，所以 36 应该保存在槽 7 处。

如果保存字符串，还可以基于字符的 ascii 值求余。字符串 “rust” 包含四个字符，其 ascii 值分别为 [114，117，115，116]，求和得到 462，求哈希的 hash(462) = 0。当然这 rust 字符

串看起来很巧，居然就是 114 - 117。我们可以选择其他的字符串试试，比如 Java，其 ascii 值为 [74,97,118,97]，求和得 386，求哈希 hash(386) = 1，所以 Java 该保存在槽 1，如下图。



ASCII 哈希函数具体如下。

```rust
// hash.rs

fn hash1(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for c in astr.chars() {
        sum += c as usize;
    }

    sum % size
}

fn main() {
    let size = 11;
    let s1 = "rust"; let s2 = "Rust";
    let p1 = hash1(s1, size);
    let p2 = hash1(s2, size);
    println!("{s1} in slot {p1}, {s2} in slot {p2}");
}
```

使用这个函数时，冲突比较严重，所以可以稍微修改一下。比如使用 “rust” 中不同字符的位置为权重，考虑将其 ascii 值乘以其位置权重。

$$\begin{aligned} hash(rust) &= (0*114 + 1*117 + 2*115 + 3*116)\%11 \\ &= 695\%11 \\ &= 2 \end{aligned} \tag{5.6}$$

当然，下标不一定从 0 开始，从 1 开始比较好，因为从 0 开始，那么第一个字符对总和没有影响。具体计算和代码如下。

$$\begin{aligned} hash(rust) &= (1*114 + 2*117 + 3*115 + 4*116)\%11 \\ &= 1157\%11 \\ &= 2 \end{aligned} \tag{5.7}$$

```rust
// hash.rs

fn hash2(astr: &str,  size: usize) -> usize {
    let mut sum = 0;
    for (i, c) in astr.chars().enumerate() {
        sum += (i + 1) * (c as usize);
    }
    sum % size
}
```

重要的是，哈希函数必须非常高效，以免其耗时成为主要部分。如果哈希函数太复杂，甚至比顺序或二分查找还耗时，这将打破哈希表的 O(1) 复杂度，那就得不偿失了。

5.5.2 解决冲突

前面我们都没有处理哈希冲突的问题，比如上面以位置为权重的 hash 函数，若下标从 0 开始，则 “rust” 和 “Rust” 字符串会发生冲突。当两项散列到同一个槽时，必须以某种方法将第二项放入哈希表中，这个过程称为冲突解决。

若哈希函数是完美的，则永远不会发生冲突。然而，由于内存有限且真实情况复杂，完美的哈希表不存在。解决冲突的一种直观方法就是查找哈希表，尝试找到下一个空槽来保存冲突项。最简单的方法是从原哈希冲突处开始，以顺序方式移动槽，直到遇到第一个空槽。注意，遇到末尾后可以循环从头开始查找。这种冲突解决方法被称为开放寻址法，具体是线性探测法，它试图在哈希表中线性地探测到下一个空槽。下图是槽为 11 的哈希表。

0	1	2	3	4	5	6	7	8	9	10
None	56	24	None	None	None	61	84	41	31	None

当我们尝试插入 35 时，其位置应为槽 2，可我们发现槽 2 存在 24，所以此时从槽 2 开始查找空槽，发现槽 3 空的，所以在此处插入 35。

0	1	2	3	4	5	6	7	8	9	10
None	56	24	35	None	None	61	84	41	31	None

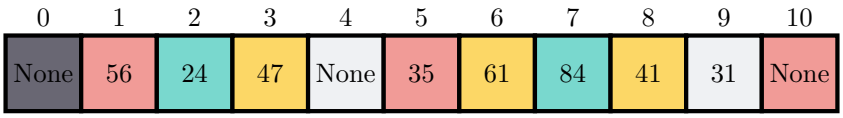
再插入 47，发现槽 3 有 35，所以查找下一个空槽，发现槽 4 为空，所以插入 47。

0	1	2	3	4	5	6	7	8	9	10
None	56	24	35	47	None	61	84	41	31	None

一旦使用开放寻址法建立了哈希表，后面就必须遵循相同的方法来查找项。假设想查找项 56，计算哈希为 1 ，查询表发现正是 56，所以返回 true。如果查找 35，计算哈希得 2，查表发现是 24，不是 35，此时不能返回 false。因为可能发生过冲突，所以要顺序查找，直到找到 35 或空槽或循环一圈再回到 24 时才能停止并返回结果。

线性查找的缺点是数据项聚集。项在表中聚集，这意味着如果在相同哈希槽处发生多次冲突，则将通过线性探测来填充多个后续槽，结果原本该插入这些槽的值被迫插入其他地方，而这种顺序查找非常费时，复杂度就不是 O(1)。比如 35 和 47 连续冲突。

处理数据项聚集的一种方式是扩展开放寻址技术，发生冲突时不是顺序查找下一个开放槽，而是跳过若干个槽，从而更均匀地分散引起冲突的项。比如加入 35 时，发生冲突，那么从此处开始，查看第三个槽，也就是每次隔三个槽来查看，这样就将冲突分散开了。此时再插入 47，就没有冲突了，这种方式有一定效果，能缓解数据聚集。



在冲突后寻找另一个槽的过程叫重哈希（重散列, rehash），其计算方法如下：

$$rehash(pos) = (pos + n)\%size$$

要注意，跳过的大小必须使得表中的所有槽最终都能被访问。为确保这一点，建议表大小是素数，这也为什么示例中要使用 11。

解决冲突的另一种方法是拉链法，也就是说对每个冲突的位置，我们设置一个链表来保存数据项，如图（5.5）。查找时，发现冲突后就再到链上顺序查找，复杂度为 $O(n)$。当然，冲突链上的数据可以排序，然后再借助二分查找，这样哈希表复杂度为 $O(log_2(n))$。如果拉链太长，还可以将链改成树，这样其结构会更加稳定。拉链法是许多编程语言内置的哈希表数据结构解决冲突的默认实现。

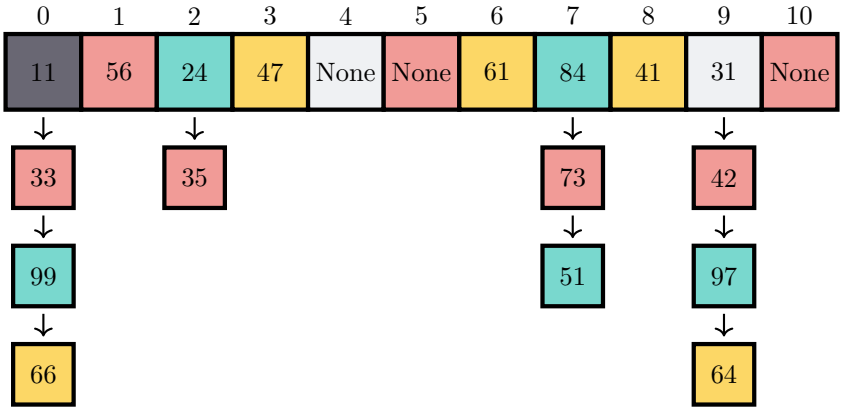


图 5.5: 拉链法解决冲突

5.5.3 Rust 实现 HashMap

Rust 集合类型中最有用的是 HashMap，它是一种关联数据类型，可以在其中存储键值对。键用于查找位置，因为数据位置不定，这种查找类似在地图（map）上查找一样，所以这种数据结构称为 HashMap。

HashMap 的抽象数据类型定义如下。该结构是键值间关联的无序集合，其中的键都是唯一的，键值间存在一对一的关系。

new() 创建一个新的 HashMap，不需要参数，返回一个空的 HashMap 集合。

insert(k,v) 向 HashMap 中添加一个新的键值对，需要参数 k、v，如果键存在，那么用新值 v 替换旧值。

remove(k) 从 HashMap 中删除某个 k，需要参数 k，返回 k 对应的 v。

get(k) 给定键 k，返回存储在 HashMap 中的值 v（可能为空 None），需要参数 k。

contains(k) 如果键 k 存在，则返回 true，需要参数 k。

len() 返回存储在 HashMap 中的键值对数量，不需要参数。

这里采用两个 Vec (slot、data) 来分别保存键和值，data 保存数据，slot 保存键，下标从 1 开始，0 为槽中默认值，HashMap 由一个 struct 封装。

```
// hashmap.rs

// 用 slot 保存位置, data 保存数据
#[derive(Debug, Clone, PartialEq)]
struct HashMap <T> {
    size: usize,
    slot: Vec<usize>,
    data: Vec<T>,
}
```

重哈希 rehash 设置为加 1 的线性探索方法。初始大小固定为 11，当然也可以是其他素数值，比如 13，17，19，23，29 等。下面是 HashMap 的实现代码。

```
// hashmap.rs

impl<T: Clone + PartialEq + Default> HashMap<T> {
    fn new() -> Self {
        // 初始化 slot 和 data
        let size = 11usize;
        let mut slot = Vec::with_capacity(size);
        let mut data = Vec::with_capacity(size);
        for _i in 0..size {
            slot.push(0);
            data.push(Default::default());
```

```
12          }
13
14          HashMap { size, slot, data }
15      }
16
17      fn hash(&self, key: usize) -> usize {
18          key % self.size
19      }
20
21      fn rehash(&self, pos: usize) -> usize {
22          (pos + 1) % self.size
23      }
24
25      fn insert(&mut self, key: usize, value: T) {
26          if 0 == key { panic!("Error: key mut > 0"); }
27
28          let pos = self.hash(key);
29          if 0 == self.slot[pos] { // 槽无数据直接插入
30              self.slot[pos] = key;
31              self.data[pos] = value;
32          } else { // 插入槽有数据再找下一个可行的位置
33              let mut next = self.rehash(pos);
34              while 0 != self.slot[next]
35                  && key != self.slot[next] {
36                  next = self.rehash(next);
37              }
38
39              // 在找到的槽插入数据
40              if 0 == self.slot[next] {
41                  self.slot[next] = key;
42                  self.data[next] = value;
43              } else {
44                  self.data[next] = value;
45              }
46          }
47      }
48
49      fn remove(&mut self, key: usize) -> Option<T> {
```

```
50          if 0 == key { panic!("Error: key mut > 0"); }
51
52          let pos = self.hash(key);
53          if 0 == self.slot[pos] { // 位中无数据, 返回 None
54              None
55          } else if key == self.slot[pos] {
56              self.slot[pos] = 0; // 更新 slot 和 data
57              let data = Some(self.data[pos].clone());
58              self.data[pos] = Default::default();
59
60              data
61          } else {
62              let mut data: Option<T> = None;
63              let mut stop = false;
64              let mut found = false;
65              let mut curr = pos;
66
67              while 0 != self.slot[curr] && !found && !stop {
68                  if key == self.slot[curr] { // 找到了值, 删除数据
69                      found = true;
70                      self.slot[curr] = 0;
71                      data = Some(self.data[curr].clone());
72                      self.data[curr] = Default::default();
73                  } else {
74                      curr = self.rehash(curr);
75                      // 再哈希回到最初位置, 说明找了一圈还没有
76                      if curr == pos { stop = true; }
77                  }
78              }
79
80              data
81          }
82      }
83
84      fn get(&self, key: usize) -> Option<&T> {
85          if 0 == key { panic!("Error: key mut > 0"); }
86
87          // 计算数据位置
```

```
88          let pos = self.hash(key);
89          let mut data: Option<&T> = None;
90          let mut stop = false;
91          let mut found = false;
92          let mut curr = pos;
93
94          // 循环查找数据
95          while 0 != self.slot[curr] && !found && !stop {
96              if key == self.slot[curr] {
97                  found = true;
98                  data = self.data.get(curr);
99              } else {
100                 curr = self.rehash(curr);
101
102                 // 再哈希回到了最初位置，说明找了一圈还没有
103                 if curr == pos {
104                     stop = true;
105                 }
106             }
107         }
108
109         data
110     }
111
112     fn contains(&self, key: usize) -> bool {
113         if 0 == key { panic!("Error: key mut > 0"); }
114         self.slot.contains(&key)
115     }
116
117     fn len(&self) -> usize {
118         let mut length = 0;
119         for d in self.slot.iter() {
120             if &0 != d  { // 槽数据不为 0，表示有数据，len + 1
121                 length += 1;
122             }
123         }
124
125         length
```

```
126         }
127     }
128
129     fn main() {
130         let mut hmap = HashMap::new();
131         hmap.insert(10,"cat");
132         hmap.insert(2,"dog");
133         hmap.insert(3,"tiget");
134
135         println!("HashMap size {:?}", hmap.len());
136         println!("HashMap contains key 2: {}", hmap.contains(2));
137         println!("HashMap key 3: {:?}", hmap.get(3));
138         println!("HashMap remove key 3: {:?}", hmap.remove(3));
139         println!("HashMap remove key 3: {:?}", hmap.remove(3));
140     }
```

5.5.4 HashMap 复杂度

最好的情况下，哈希表提供 O(1) 的查找复杂度。然而，由于冲突，查找过程中比较的数量通常会变动。这种冲突越剧烈，则性能越差。一种好的评估指标是负载因子 λ。如果负载因子小，则碰撞的机会低，这意味着项更可能在它们所属的槽中。如果负载因子大，意味着表快填满了，则存在越来越多的冲突。这意味着冲突解决更复杂，需要更多的比较来找到一个空槽。即便使用拉链法，冲突增加意味着链上的项数增加，则查找的时候链上查找花费时间占主要部分。

每次查找的结果是成功或不成功。对于使用线性探测的开放寻址法进行的成功查找，平均比较次数大约为 $\frac{1+\frac{1}{1-\lambda}}{2}$，不成功时查找次数大约为 $\frac{1+(\frac{1}{1-\lambda})^2}{2}$ 。即便使用拉链法，对于成功的情况，平均比较数目是 $1+\lambda/2$，查找不成功，则是 λ 次比较。总体来说哈希表的查找复杂度为 $O(\lambda)$ 左右。

5.6 总结

本章我们学习了查找算法，包括顺序查找，二分查找和哈希查找。顺序查找是最简单和直观的查找算法，其复杂度为 O(n)，二分查找算法每次都去掉一半数据，速度比较快，但要求数据集有序，复杂度为 $O(log_2(n))$。基于二分查找衍生了类似内插查找和指数查找这样的算法，它们适合的数据分布类型不同。哈希查找是利用 HashMap 实现的一种 O(1) 查找算法，要注意的是哈希表容易冲突，需要采取合理措施解决冲突，比如开放寻址法、拉链法。学习查找算法的过程中我们发现排序对于查找算法加快速度很有帮助，所以下一章我们来学习排序算法。

Chapter 6

排序

6.1 本章目标

学习了解排序思想
能用 Rust 实现十大基本排序算法

6.2 什么是排序

排序是以某种顺序在集合中放置元素的过程。例如，斗地主时大家拿到自己的牌都会抽来抽去的将各种牌按顺序或花色放到一起，这样便于思考和出牌。一堆散乱的单词可以按字母顺序或长度排序。中国的城市则可按人口、种族、地区排序。

排序是计算机科学的一个重要研究领域，大量领域内前辈对排序作出了杰出的贡献，推动了计算机科学的发展。我们已经看过许多能够从排序数据集合中获益的算法，包括顺序查找和二分算法等，这也说明了排序的重要性。

与查找算法一样，排序算法的效率与处理的项数有关。对于小集合，复杂的排序方法开销太高。另一方面，对于大的数据集，简单的算法性能又太差。在本章中，我们将讨论多种排序算法，并对它们的运行性能进行比较。

在分析特定算法之前，首先要知道排序设计的核心操作是比较。因为序列、顺序是靠比较得出来的，这和人的幸福感一样，靠比较才知道自己有多幸福。比较就是查看哪个更大，哪个更小。其次，如果比较发现顺序不对，就涉及数据位置交换。交换是一种昂贵的操作，交换的总次数对于评估算法的效率很关键。前面我们学习过大 O 分析法，本章的所有算法都要用大 O 分析法来分析性能，这是评价排序算法最直观的指标。

此外，排序还存在稳定与否的问题。例如 [1,4,9,8,5,5,2,3,7,6] 中，有两个 5，那么不同的排序可能会将两个 5 交换顺序，虽然它们最终是挨着的，但已经破坏了原有的次序关系。当然，对于纯数字是无所谓的，但若是这些数字是某个数据结构的某个键，例如下面这样的结构，包含个人信息，虽然其 amount 参数都是 5，但显然两个人的姓名和年龄都不同，贸然排序会改变次序。

```
people {
    amount: 5,
    name: 张三,
    age: 20,
}

people {
    amount: 5,
    name: 王五,
    age: 24,
}
```

张三如果原本在前，排序后排到后面了，这就会出现问题，尤其是有的算法依赖序列的稳定性，比如扣款这样的操作。所以评价排序算法除了时间空间复杂度，还要看稳定性。

对集合的排序，存在各种各样的排序算法。但有十类排序算法是各种排序算法的基础，它们是：冒泡排序，快速排序，选择排序，堆排序，插入排序，希尔排序，归并排序，计数排序，桶排序，基数排序。

除此之外，基于十类基础算法还衍生了许多改进的算法，本文选择了部分进行讲解，包括鸡尾酒排序、梳子排序、二分插入排序、Flash 排序、蒂姆排序。

6.3 冒泡排序

冒泡排序需要多次遍历集合，它比较相邻的项并交换那些无序的项。每次遍历列表都将最大的值放在其正确的位置。这就类似烧开水时，壶底的水泡往上冒的过程，这也是它叫冒泡排序的原因。

下一页的图（6.1）展示了冒泡排序的第一次遍历，阴影项正在比较是否乱序。如果在列表中有 n 项，则第一遍有 n-1 次项比较。在第二次遍历数据的时候，数据集中的最大值已经在正确的位置，剩下的 n-1 个数据还需要排序，这意味着将有 n-2 次比较。由于每次通过将下一个最大值放置在适当位置，所需的遍历的总数将是 n-1。在完成 n-1 轮遍历比较后，最小项肯定在正确的位置，不需要进一步处理。

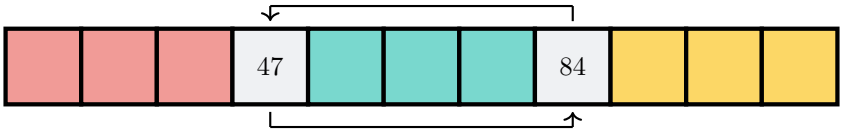
仔细看对角线，可以发现是最大值，而且它在不断往最右侧移动。冒泡排序涉及频繁的交换操作（swap），交换是比较中常用的辅助操作，在 Rust 中 Vec 数据结构就默认实现了 swap 函数，当然你也可以实现如下的交换操作。

```
// swap
let temp = data[i];
data[i] = data[j];
data[j] = temp;
```

84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24
84	66	92	56	44	31	72	19	24
84	66	56	92	44	31	72	19	24
84	66	56	44	92	31	72	19	24
84	66	56	44	31	92	72	19	24
84	66	56	44	31	72	92	19	24
84	66	56	44	31	72	19	92	24
84	66	56	44	31	72	19	24	92

图 6.1: 冒泡排序

有些语言，如 Python，可以不用临时变量便直接交换值，如：data[i], data[j] = data[j], data[i]，这是语言实现的特性，其内部还是使用了变量，只不过是同时操作两个。



结合上所述，我们可以用 Vec 来实现冒泡排序。注意，为简化算法设计，本章中所有待排序集合里保存的都是数字。

```rust
// bubble_sort.rs

fn bubble_sort1(nums: &mut [i32]) {
    if nums.len() < 2 {
        return;
    }

    for i in 1..nums.len() {
        for j in 0..nums.len()-i {
            if nums[j] > nums[j+1] {
```

```
11                      nums.swap(j, j+1);
12                  }
13              }
14          }
15      }
16
17      fn main() {
18          let mut nums = [54,26,93,17,77,31,44,55,20];
19          bubble_sort1(&mut nums);
20          println!("sorted nums: {:?}", nums);
21      }
```

除用 for 循环外，也可用 while 循环实现冒泡排序。这两程序均在 bubble_sort.rs 中。

```
1   // bubble_sort.rs
2
3   fn bubble_sort2(nums: &mut [i32]) {
4       let mut len = nums.len() - 1;
5       while len > 0 {
6           for i in 0..len {
7               if nums[i] > nums[i+1] {
8                   nums.swap(i, i+1);
9               }
10          }
11          len -= 1;
12      }
13  }
14
15  fn main() {
16      let mut nums = [54,26,93,17,77,31,44,55,20];
17      bubble_sort2(&mut nums);
18      println!("sorted nums: {:?}", nums);
19  }
```

注意，不管项在初始集合中如何排列，算法都将进行 n-1 轮遍历以排序 n 个数字。第一轮要比较 n - 1 次，第二轮 n - 2 次，直到 1 次。总的比较次数为 1 + 2 +..., n - 1，为前 n 个整数之和 $\frac{n^2}{2} + \frac{n}{2}$，所以性能为 $O(n^2)$。

上面的冒泡算法都实现了排序，但仔细分析发现，即便序列已经排好序了，算法也要不断的比较，只是不交换值。一个有序的集合就不应该再排序了。修改上面算法，添加一个 compare 变量来控制是否继续比较，在遇到已排序集合时直接退出。

```rust
// bubble_sort.rs
fn bubble_sort3(nums: &mut [i32]) {
    let mut compare = true;
    let mut len = nums.len() - 1;

    while len > 0 && compare {
        compare = false;
        for i in 0..len {
            if nums[i] > nums[i+1] {
                nums.swap(i, i+1);
                compare = true; // 数据无序，还需继续比较
            }
        }
        len -= 1;
    }
}
```

冒泡排序是从第一个数开始，依次往后比较，相邻的元素两两比较，根据大小来交换元素的位置。这个过程中，元素是单向交换的，也就是说只有从左往右交换。那么是否可以再从右到左来个冒泡排序呢？从左到右是升序排序，如果从右到左采取降序排序，那么这种双向排序法也一定能完成排序。这种排序称为鸡尾酒排序，是冒泡排序的一种变体排序。鸡尾酒稍微优化了冒泡排序，其复杂度还是 $O(n^2)$，若序列已经排序，则接近 $O(n)$。

```rust
// cocktail_sort.rs
fn cocktail_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }

    // bubble 控制是否继续冒泡
    let mut bubble = true;
    let len = nums.len();
    for i in 0..(len >> 1) {
        if bubble {
            bubble = false;

            // 从左到右冒泡
            for j in i..(len - i - 1) {
                if nums[j] > nums[j+1] {
                    nums.swap(j, j+1);
                    bubble = true
```

```
17                  }
18              }
19
20              // 从右到左冒泡
21              for j in (i+1..=(len - i - 1)).rev() {
22                  if nums[j] < nums[j-1] {
23                      nums.swap(j-1, j);
24                      bubble = true
25                  }
26              }
27          } else {
28              break;
29          }
30      }
31  }
32
33  fn main() {
34      let mut nums = [1,3,2,8,3,6,4,9,5,10,6,7];
35      cocktail_sort(&mut nums);
36      println!("sorted nums {:?}", nums);
37  }
```

在冒泡排序中，只会比较数组中相邻的项，比较间距为 1。一种称为梳排序的算法提出比较间距可以大于 1。梳排序中，开始比较间距设定为数组长度，并在循环中以固定的比率递减，通常递减率为 1.3，该数字是原作者通过实验得到的最有效递减率。因为乘法计算中耗时比除法短两个时钟周期，所以间距计算会取递减率的倒数与数组长度相乘，即 0.8。当间距为 1 时，梳排序就退化成了冒泡排序。梳排序通过尽量把逆序的数字往前移动并保证当前间隔内的数有序，类似梳子理顺头发，间隔则类似梳子梳齿间隙。梳排序时间复杂度是 O(nlogn)，空间复杂度为 O(1)，属于不稳定的排序算法。

```
1  // comb_sort.rs
2  fn comb_sort(nums: &mut [i32]) {
3      if nums.len() <= 1 { return; }
4
5      let mut i;
6      let mut gap: usize = nums.len();
7
8      // 大致排序，数据基本有序
9      while gap > 0 {
```

```
10          gap = (gap as f32 * 0.8) as usize;
11          i = gap;
12          while i < nums.len() {
13              if nums[i-gap] > nums[i] {
14                  nums.swap(i-gap, i);
15              }
16              i += 1;
17          }
18      }
19
20      // 细致调节部分无序数据, exchange 控制是否继续交换数据
21      let mut exchange = true;
22      while cnt > 0 {
23          exchange = false;
24          i = 0;
25          while i < nums.len() - 1 {
26              if nums[i] > nums[i+1] {
27                  nums.swap(i, i+1);
28                  exchange = true;
29              }
30              i += 1;
31          }
32      }
33  }
34
35  fn main() {
36      let mut nums = [1,2,8,3,4,9,5,6,7];
37      comb_sort(&mut nums);
38      println!("sorted nums {:?}", nums);
39  }
```

冒泡排序还有一个问题，就是它需要合理的安排好边界下标值如 i、j、i+1、j+1，一点都不能错。下面是 2021 年新发表的一种不需要处理边界下标值的排序算法 [11]，非常直观，乍一看以为是冒泡排序，但它实际类似插入排序。看起来是降序排序，实际是升序排序。

```
1  // CantBelieveItCanSort.rs
2
3  fn cbic_sort1(nums: &mut [i32]) {
4      for i in 0..nums.len() {
```

```
5           for j in 0..nums.len() {
6               if nums[i] < nums[j] {
7                   nums.swap(i, j);
8               }
9           }
10      }
11  }
12
13  fn main() {
14      let mut nums = [54,32,99,18,75,31,43,56,21,22];
15      cbic_sort1(&mut nums);
16      println!("sorted nums {:?}", nums);
17  }
```

当然，也可以优化此排序算法，像下面这样。

```
1   // CantBelieveItCanSort.rs
2
3   fn cbic_sort2(nums: &mut [i32]) {
4       if nums.len() < 2 {
5           return;
6       }
7
8       for i in 1..nums.len() {
9           for j in 0..i {
10              if nums[i] < nums[j] {
11                  nums.swap(i, j);
12              }
13          }
14      }
15  }
16
17  fn main() {
18      let mut nums = [54,32,99,18,75,31,43,56,21,22];
19      cbic_sort2(&mut nums);
20      println!("sorted nums {:?}", nums);
21  }
```

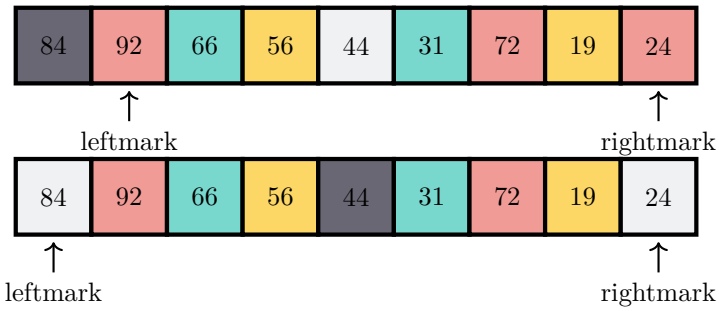
这个排序算法是直觉上最像冒泡定义的排序算法，然而它并不是冒泡排序算法。读者若要实现降序排序，只需要改变小于符号为大于符号就行。

6.4 快速排序

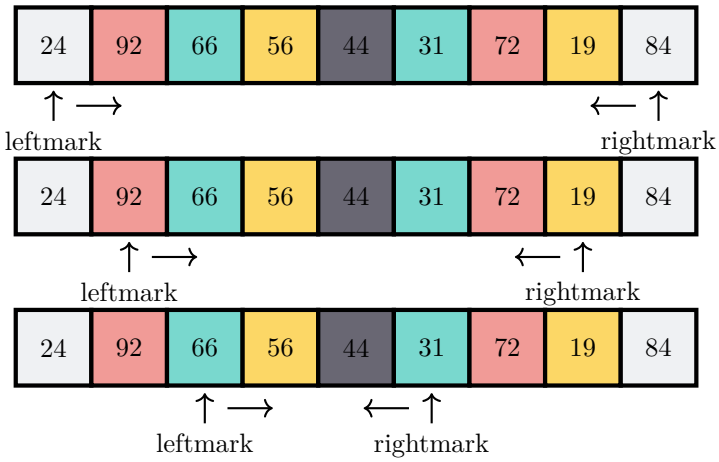
快速排序和冒泡排序有相似之处，应该说快速排序是冒泡排序的升级版。快速排序使用分而治之的策略来加快排序速度，这又和二分思想、递归思想有些类似。

快速排序只有两个步骤，一是选择中枢值，二是分区排序。首先在集合中选择某个值作为中枢，其作用是帮助拆分集合。注意中枢值不一定要选集合中间位置的值，中枢值应该是在最终排序集合中处于中间或靠近中间的值，这样排序速度才快。有很多不同的方法用于选择中枢值，本文只为说明原理，不考虑算法优化，所以直接选择第一项作中枢值。下图选择了 84 作为中枢值，实际上，排好序后，84 并不在中间，而是在倒数第二位。56 才是在中间。所以如果能选到 56 作中枢值将会非常高效。

选择好中枢值后 (灰色值)，需要再设置两个标记，用于比较。这两个标记称为左标记和右标记，分别处于集合中除中枢值的最左和最右端。下图展示了左右标记的位置。



可以看到，左右标记尽可能远离，处于左右两个极端。分区的目标是移动相对于中枢值 (44) 错位的项，通过比较左右标记和中枢处的值，交换小值到左标记处，大值到右标记处。



首先右移左标记，直到找到一个大于等于中枢值的值。然后左移递减右标记，直到找到小于等于中枢值的值。如果左标记值大于右标记值就交换值。此处 84 和 24 恰好满足条件，所以直接交换值。然后重复该过程。直到左右标记相互越过对方。比较越过后左右标记值，若右小于左，则将此时右标记值和中枢值交换，右标记值作为分裂点，将集合分为左右两个区间，然后在左右区间递归调用快速排序，直到最后完成排序。

可以看到，只要左右两侧分别执行快速排序，最终就能完成排序。如果集合长度小于或等于一，则它已经排序，直接退出。具体实现如下，我们为分区设置了专门的分区函数 partition，算法中取第一项做中枢值。

```
1  // quick_sort.rs
2
3  fn quick_sort1(nums: &mut [i32], low: usize, high: usize) {
4      if low < high {
5          let split = partition(nums, low, high);
6          if split > 1 { // 防止越界和语法错误 (split <= 1 的情形)
7              quick_sort1(nums, low, split - 1);
8          }
9          quick_sort1(nums, split + 1, high);
10     }
11 }
12
13 fn partition(nums: &mut [i32], low: usize, high: usize)
14     -> usize {
15     let mut lm = low;  // 左标记
16     let mut rm = high; // 右标记
17     loop {
18         // 左标记不断右移
19         while lm <= rm && nums[lm] <= nums[low] {
20             lm += 1;
21         }
22         // 右标记不断右移
23         while lm <= rm && nums[rm] >= nums[low] {
24             rm -= 1;
25         }
26
27         // 左标记越过右标记时退出并交换左右标记数据
28         if lm > rm {
29             break;
30         } else {
```

```
31                  nums.swap(lm, rm);
32              }
33          }
34          nums.swap(low, rm);
35
36          rm
37      }
38
39      fn main() {
40          let mut nums = [54,26,93,17,77,31,44,55,20];
41          let len = nums.len();
42          quick_sort1(&mut nums, 0, (len - 1) as isize);
43          println!("sorted nums: {:?}", nums);
44      }
```

对于长度为 n 的集合，如果分区总在中间，则会再出现 $log_2(n)$ 个分区。为找到分割点，需要针对中枢值检查 n 项中的每一个，复杂度为 $nlog_2(n)$ 。最坏的情况下，分裂点可能不在中间，非常偏左或右。此时会不断的对 1 和 n-1 项重复排序 n 次，复杂度为 $O(n^2)$。

快速排序需要递归，深度过深性能会下降。内观排序克服了这个缺点，其在递归深度超过 $log(n)$ 后转为堆排序。在数量少（$n < 20$）时，则转为插入排序。这种多个排序混合而成的排序能在常规数据集上实现快速排序的高性能，又能在最坏情况下仍保持 O(nlog(n)) 的性能，内观排序是 C++ 的内置排序算法。此外，对快速排序分析发现，它总是将待排序数组分成两个区域来排序。如果待排序集合有大量重复元素，则快速排序会重复比较，造成性能浪费。解决方法是将数组分成三区排序，把重复元素放到第三个区域，排序时只对另外两个区域排序。选择重复数据作为 pivot，小于 pivot 的放到左区，大于 pivot 的放到右区，等于的放到中区。然后对左右区域递归调用三区快速排序。

6.5 插入排序

插入排序，就像它的名字暗示的一样，是通过插入数据项来实现排序。尽管仍然性能是 $O(n^2)$，但其工作方式略有不同。它始终在数据集的较低位置处维护一个有序的子序列，然后将新项插入子序列，使得子序列扩大，最终实现集合排序。

假设开始的子序列只有一项，位置为 0。在下次遍历时，对于项 1 至 n-1，将其与第一项比较，如果小于该项，则将其插入到该项前。如果大于该项，则增长子序列，使长度加一。接着重复比较过程，在剩余的未排序项里取数据来比较。结果是要么插入子序列某个位置，要么增长子序列，最终得到排好序的集合。插入排序的具体实现和图示如下。

```
1  // insertion_sort.rs
2
```

```rust
fn insertion_sort(nums: &mut [i32]) {
    for i in 1..nums.len() {
        let mut pos = i;
        let curr = nums[i];

        while pos > 0 && curr < nums[pos-1] {
            nums[pos] = nums[pos-1]; // 向后移动数据
            pos -= 1;
        }
        nums[pos] = curr; // 插入数据
    }
}

fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21];
    insertion_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
}
```

84	92	66	56	44	31	72	19	24	假设 84 已排序
84	92	66	56	44	31	72	19	24	仍旧有序
66	84	92	56	44	31	72	19	24	插入 66
56	66	84	92	44	31	72	19	24	插入 56
44	56	66	84	92	31	72	19	24	插入 44
31	44	56	66	84	92	72	19	24	插入 31
31	44	56	66	72	84	92	19	24	插入 72
19	31	44	56	66	72	84	92	24	插入 19
19	24	31	44	56	66	72	84	92	插入 24

图 6.2: 插入排序

插入排序总是需要和前面已排序的元素逐个比较来找到具体位置，而第五章我们学习了二分查找法，可以快速找到元素在已排序子序列中的位置，所以可以利用二分法来加快插入排序的速度。具体的改动就是在插入时，用二分法找到数据该插入的位置，然后直接移动数据到相应位置。

```rust
// bin_insertion_sort.rs

fn bin_insertion_sort(nums: &mut [i32]) {
    let mut temp;
    let mut left;
    let mut mid;
    let mut right;

    for i in 1..nums.len() {
        left = 0;        // 已排序数组左右边界
        right = i - 1;
        temp = nums[i]; // 待排序数据

        // 二分法找到 temp 的位置
        while left <= right {
            mid = (left + right) >> 1;
            if temp < nums[mid] {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        // 将数据后移，留出空位
        for j in (left..=i-1).rev() {
            nums.swap(j, j+1);
        }

        // 将 temp 插入空位
        if left != i {
            nums[left] = temp;
        }
    }
}
```

```
35
36 fn main() {
37     let mut nums = [1,3,2,8,6,4,9,7,5,10];
38     bin_insertion_sort(&mut nums);
39     println!("sorted nums {:?}", nums);
40 }
```

6.6 希尔排序

希尔排序，也称递减递增排序。它将原始集合分为多个较小的子集合，然后对每个集合运用插入排序。选择子集合的方式是希尔排序的关键。希尔排序不是将集合均匀拆分为连续项的子列表，而是隔几个项选择一个项加入子集合，隔开的距离称为增量 gap。

84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24
84	92	66	56	44	31	72	19	24

这可以通过上图来加以理解。该集合有九项，如果使用三为增量，则总共会有三个子集合，每个子集合三项，颜色一样。这些隔开的元素可以看成是连接在一起的，这样可以通过插入排序来对同颜色元素进行排序。

完成排序后，总的集合还无序，如下图。虽然总集合无序，但这个集合并非完全无序，可以看到，同种颜色的子集合是有序的。只要再对整个集合进行插入排序，则很快就能将集合完全排序。可以发现，此时的插入排序移动次数非常少，因为挨着的几个项都处于自己所在子集合的有序位置，那么这些挨着的项也几乎有序，所以只用少量插入次数就能完成排序。

56	19	24	72	44	31	84	92	66

希尔排序中，增量是关键，也是其特征。可以使用不同的增量，但增量为几，那么子集合就有几个。下面是希尔排序的实现，通过不断调整 gap 值，实现排序。

```
1 // shell_sort.rs
2
3 fn shell_sort(nums: &mut [i32]) {
4     // 插入排序函数(内部), 数据相隔距离为 gap
5     fn ist_sort(nums: &mut [i32], start: usize, gap: usize) {
6         let mut i = start + gap;
```

```
7
8           while i < nums.len() {
9               let mut pos = i;
10              let curr = nums[pos];
11              while pos >= gap && curr < nums[pos - gap] {
12                  nums[pos] = nums[pos - gap];
13                  pos -= gap;
14              }
15
16              nums[pos] = curr;
17              i += gap;
18          }
19      }
20
21      // 每次让 gap 减少一半直到 1
22      let mut gap  = nums.len() / 2;
23      while gap > 0 {
24          for start in 0..gap {
25              ist_sort(nums, start, gap);
26          }
27          gap /= 2;
28      }
29  }
30
31  fn main() {
32      let mut nums = [54,32,99,18,75,31,43,56,21,22];
33      shell_sort(&mut nums);
34      println!("sorted nums: {:?}", nums);
35  }
```

乍一看，希尔排序并不比插入排序更好，因为它最后一步还是执行了完整的插入排序。然而，希尔排序分割子序列对其排序后，最后一次插入排序进行的插入操作就非常少了，因为该集合已经被较早的增量插入排序预排序了。换句话说，随着 gap 值向 1 靠拢，整个集合都比上一次更有序，这使得总的排序非常高效。

希尔排序的复杂度分析稍微复杂一些，但其大致分布在 $O(n)$ 到 $O(n^2)$ 之间。改变 gap 的值，使其按照 $2^k - 1$ 变化 (1, 3, 7, 15, 31)，那么其复杂度大概在 $O(n^{1.5})$ 左右，也是非常快的。当然，前面对插入排序可以用二分法改进，那么对希尔排序也可以使用二分法改进，具体思路同前面一样，只是下标处理不是连续的，要计算上 gap 值。

6.7 归并排序

现在转向使用分而治之策略作为提高排序算法性能的另外一种方法，归并排序。归并排序和快速排序都是一种分而治之的递归算法，通过不断将列表折半来进行排序。如果集合为空或只有一个项，则按基本情况进行排序。如果有多项，则分割集合，并递归调用两个区间的归并排序。一旦对这两个区间排序完成，就执行合并操作。合并是获取两个子排序集合并将它们组合成单个排序新集合的过程。因为归并排序是一种结合递归和合并操作的排序，所以名字就叫归并排序，如下图所示。

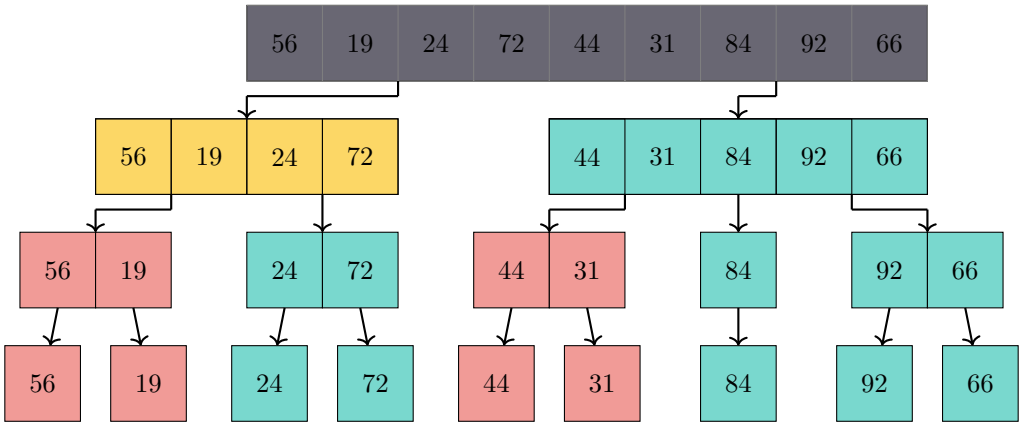


图 6.3: 归并排序分解

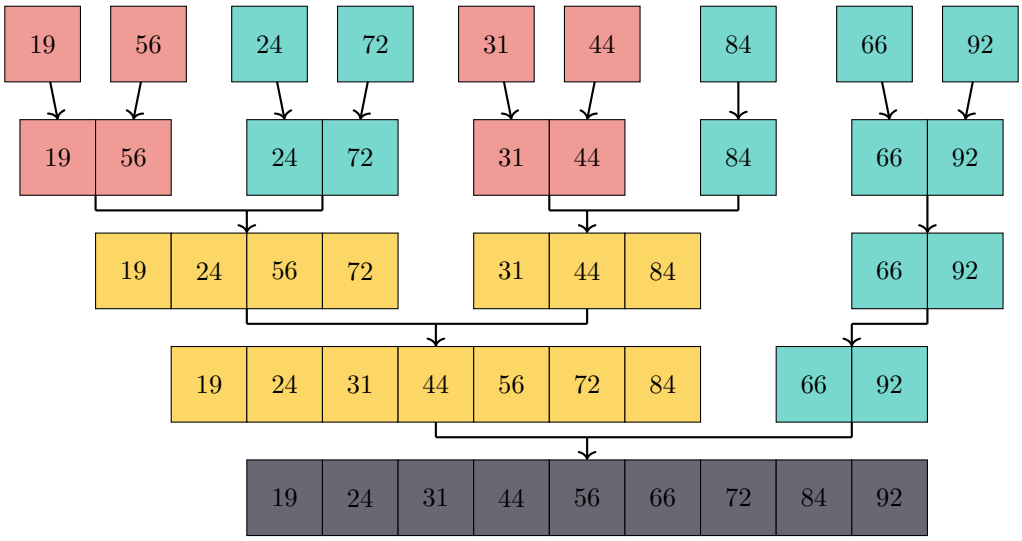


图 6.4: 归并排序合并

归并排序时递归分解集合到只有两个元素或一个元素的基本情况，便于直接比较。分解后是合并过程，首先对基本情况的最小子序列进行排序，接着开始两两合并，直到完成集合

排序，如上图所示。归并排序分割集合时，可能不是均分，因为数据不一定是偶数，但最多相差一个元素，不影响性能。合并操作其实很简单，因为每一次合并时，子序列都已经排好序，只需要逐个比较，先取小的，最后组合的序列一定也是有序的。下面是归并排序的实现代码。

```rust
// merge_sort.rs

fn merge_sort(nums: &mut [i32]) {
    if nums.len() > 1 {
        let mid = nums.len() >> 1;
        merge_sort(&mut nums[..mid]); // 排序前半部分
        merge_sort(&mut nums[mid..]); // 排序后半部分
        merge(nums, mid); // 合并排序结果
    }
}

fn merge(nums: &mut [i32], mid: usize) {
    let mut i = 0;    // 标记前半部分数据
    let mut k = mid; // 标记后半部分数据
    let mut temp = Vec::new();

    for _j in 0..nums.len() {
        if k == nums.len() || i == mid {
            break;
        }

        // 数据放到临时集合 temp
        if nums[i] < nums[k] {
            temp.push(nums[i]);
            i += 1;
        } else {
            temp.push(nums[k]);
            k += 1;
        }
    }

    // 合并的两部分数据长度大概率不一样长
    // 所以要将未处理完集合的数据全部加入
    if i < mid && k == nums.len() {
```

```
35            for j in i..mid {
36                temp.push(nums[j]);
37            }
38        } else if i == mid && k < nums.len() {
39            for j in k..nums.len() {
40                temp.push(nums[j]);
41            }
42        }
43
44        // temp 数据放回 nums, 完成排序
45        for j in 0..nums.len() {
46            nums[j] = temp[j];
47        }
48    }
49
50    fn main() {
51        let mut nums = [54,32,99,22,18,75,31,43,56,21];
52        merge_sort(&mut nums);
53        println!("sorted nums: {:?}", nums);
54    }
```

为了分析归并排序的性能，首先将集合分成两部分。在二分查找一节已经学习过，总共需要分 $log_2(n)$ 次。第二个过程是合并，集合中的每个项最终将被放置在排好序的列表上，对于 n 个数据，最多就放 n 次。递归和合并两个操作是结合在一起的，所以归并排序是一种性能为 $O(nlog_2(n))$ 的算法。

归并排序空间复杂度为 O(n)，这是比较高的，自然的想法就是减少空间使用。我们知道插入排序空间复杂度是 O(1)，所以可以利用插入排序来优化归并排序。当长度小于某阈值时直接调用插入排序，大于阈值时采用归并排序，这种算法叫插入归并排序，在一定程度上优化了归并排序算法。

6.8 选择排序

选择排序是对冒泡排序的改进，每次遍历集合只做一次交换。为做到这一点，选择排序在遍历时只寻找最大值的下标，并在完成遍历后，将该最大项交换到正确的位置。与冒泡排序一样，在第一次遍历后，集合的最大项在最后一个位置；第二遍后，次个值在倒数第二位，遍历 n-1 次才会排序完 n 个项。下图展示了整个排序过程。每次遍历时，选择未排序的最大的项，然后放置在适当位置。

选择排序与冒泡排序具有相同数量的比较，因此性能也是 $O(n^2)$。然而，由于选择排序每轮只进行一次数据交换，所以比冒泡排序更快，下面是其实现代码。

```rust
// selection_sort.rs
fn selection_sort(nums: &mut Vec<i32>) {
    let mut left = nums.len() - 1; // 待排序数据下标
    while left > 0 {
        let mut pos_max = 0;
        for i in 1..=left {
            if nums[i] > nums[pos_max] {
                pos_max = i; // 选择当前轮次最大值下标
            }
        }
        // 数据交换，完成一个数据的排序，待排序数据量减 1
        nums.swap(left, pos_max);
        left -= 1;
    }
}
fn main() {
    let mut nums = vec![54,32,99,18,75,31,43,56,21,22];
    selection_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
}
```

下图是选择排序的示意图，值被挑出来放到正确位置，而不像冒泡排序那样交换。

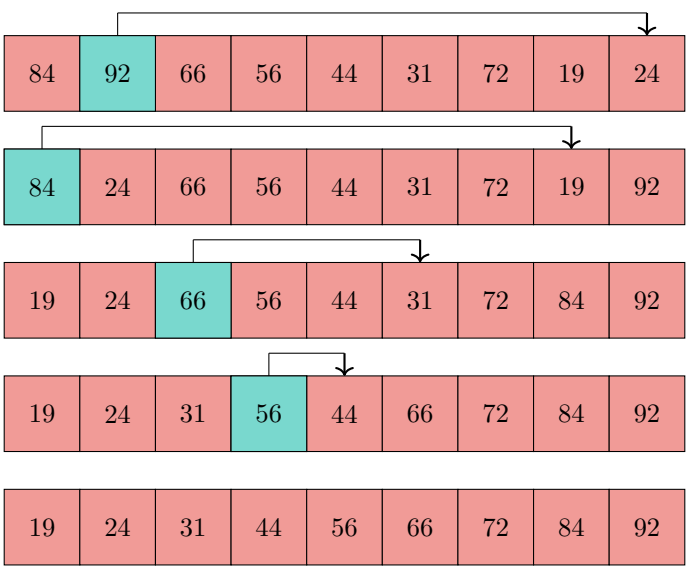


图 6.5: 选择排序

可以看到，即使冒泡和选择排序复杂度一样，仍然有优化点。这对于今后学习、研发很有帮助，要在看起来很复杂的问题中找出优化点，尽可能地优化它。前面我们学习鸡尾酒排序可以双向同时排序，此处的选择排序也可以实现双向排序。这种改进的选择排序的复杂度和常规的选择排序是一样的，唯一改变的是复杂度的系数，具体请读者自己思考实现。

6.9 堆排序

前面章节学习了栈，队列这些线性数据结构并用这些数据结构实现了各种算法。除此之外，计算机里还有非线性的数据结构，其中一种是堆。堆是一种非线性的完全二叉树，具有以下性质：每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆；每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆，如图（6.6）所示是一个小顶堆。

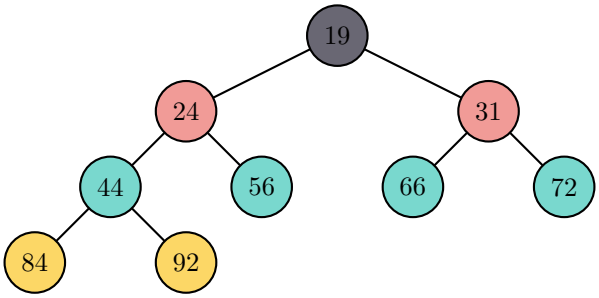


图 6.6: 小顶堆

堆排序是利用堆数据结构设计的一种排序算法，是一种选择排序，通过不断选择顶元素到末尾，然后再重建堆实现排序。它的最坏、最好、平均时间复杂度均为 $O(nlog_2(n))$，它是不稳定排序。虽然在第七章才开始学习树，但这里稍微了解一下有助于理解堆的性质。通过下面的图可以看到，这种堆类似具有多个连接的链表。如果对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就像下图一样。其中第一位，下标为 0，此处用 0 来占位。

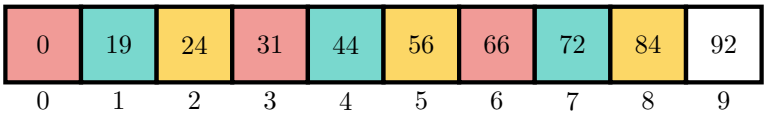


图 6.7: 小顶堆数组表示

可见，我们不一定非得用链表，用 Vec 或者数组都能表示堆，其实堆就其意义上来说，也类似一堆东西聚合在一起，所以数组或 Vec 表示的堆比之二叉树结构更贴近堆这个词汇的本身意义。注意此处我们的下标从 1 开始，这样可以将左右子节点表示为 arr[2i] 和 arr[2i+1]。

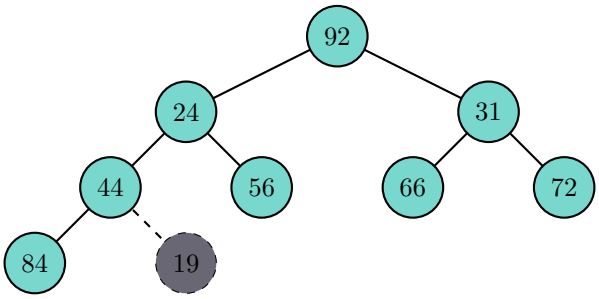
借助数组表示，按照二叉树的节点关系，堆满足如下的定义：

大顶堆：arr[i] >= arr[2i] 且 arr[i] >= arr[2i+1]

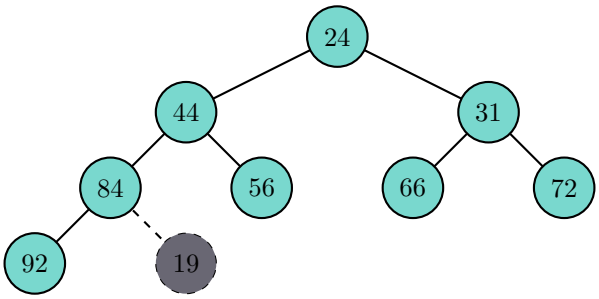
小顶堆：arr[i] <= arr[2i] 且 arr[i] <= arr[2i+1]

堆排序的基本思想是：将待排序序列构造成一个小顶堆，此时，整个序列的最小值就是堆顶根节点。将其与末尾元素进行交换，此时末尾就为最小值。这个最小值不再计算到堆内，那么再将剩余的 n - 1 个元素重新构造成一个堆，这样会得到一个新的最小值。此时将该最小值再次交换到新堆的末尾，这样就有了两个排序的值。重复这个过程，直到得到一个有序序列。当然，小顶堆得到的是降序排序，大顶堆得到的才是升序排序。

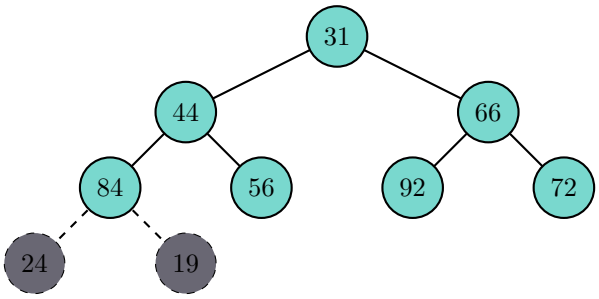
为便于读者理解堆排序过程，可借助图来演示其排序过程。下图中灰色为最小元素，是从顶部和 92 替换到了此处，不再计入堆内。交换后，92 位于堆顶，堆不再是小顶堆，所以交换后需要重新构建小顶堆，使得最小值 24 在堆顶。之后再交换 24 和堆中最后一个元素，则倒数第二个将变为灰色。注意虚线表示该元素已经不属于堆了，重建堆和交换时都不再处理它。继续交换下去，则灰色排序的子序列从最后一层倒序着逐步填满整个堆，最终实现排序，此时序列是从大到小逆序排序。



下面是重新构建小顶堆后 24 位于堆顶时堆的情况。



下面时交换堆顶元素及重建堆后的情况，此时 92 已经交换到右侧，31 是堆中最小元素。



有了上面的图示，可以实现如下堆排序。

```rust
// heap_sort.rs

macro_rules! parent { // 计算父节点下标宏
    ($child:ident) => {
        $child >> 1
    };
}
macro_rules! left_child {// 计算左子节点下标宏
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}
macro_rules! right_child {// 计算右子节点下标宏
    ($parent:ident) => {
        ($parent + 1) << 1
    };
}

fn heap_sort(nums: &mut [i32]) {
    if nums.len() <= 1 { return; }

    let len = nums.len();
    let last_parent = parent!(len);
    for i in (0..=last_parent).rev() {
        move_down(nums, i); // 第一次建小顶堆
    }

    for end in (1..nums.len()).rev() {
        nums.swap(0, end);
        move_down(&mut nums[..end], 0); // 重建堆
    }
}

// 大的数据项下移
fn move_down(nums: &mut [i32], mut parent: usize) {
    let last = nums.len() - 1;
    loop {
        let left = left_child!(parent);
```

```
39          let right = right_child!(parent);
40          if left > last { break; }
41
42          // right <= last 确保存在右子节点
43          let child = if right <= last
44                      && nums[left] < nums[right] {
45              right
46          } else {
47              left
48          };
49
50          // 子节点大于父节点，交换数据
51          if nums[child] > nums[parent] {
52              nums.swap(parent, child);
53          }
54
55          更新父字关系
56          parent = child;
57      }
58  }
59
60  fn main() {
61      let mut nums = [54,32,99,18,75,31,43,56,21,22];
62      heap_sort(&mut nums);
63      println!("sorted nums: {:?}", nums);
64  }
```

堆就是二叉树，其复杂度是 $O(nlog_2(n))$。这里我们使用了宏 parent!，left_child!，right_child! 来获取节点下标，当然也可以用函数来实现，但这里就当作复习 rust 基础知识。堆排序和选择排序有些类似，都是在集合中选择出最值并放到适当位置。

6.10 桶排序

前面学习的排序多涉及到比较操作，其实还有一些排序不用比较，只要按照数学规律就能自动映射数据到正确位置。这类非比较算法主要是桶排序，计数排序，基数排序。

非比较排序通过确定每个元素之前有多少个元素存在来排序。比如对集合 nums，计算 nums[i] 之前有多少个元素，则唯一确定了 nums[i] 在排序集合中的位置。非比较排序只要确定每个元素之前的已有的元素个数即可，所以一次遍历即可完成排序，时间复杂度 O(n)。

虽然非比较排序时间复杂度低，但由于非比较排序需要占用额外空间来确定位置，所以

对数据规模和数据分布有一定的要求。因为不是所有数据都适合这类排序，数据本身必须包含可索引的信息用于确定位置。而比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况，非比较排序只适合特殊数据（尤其是数字）的排序。

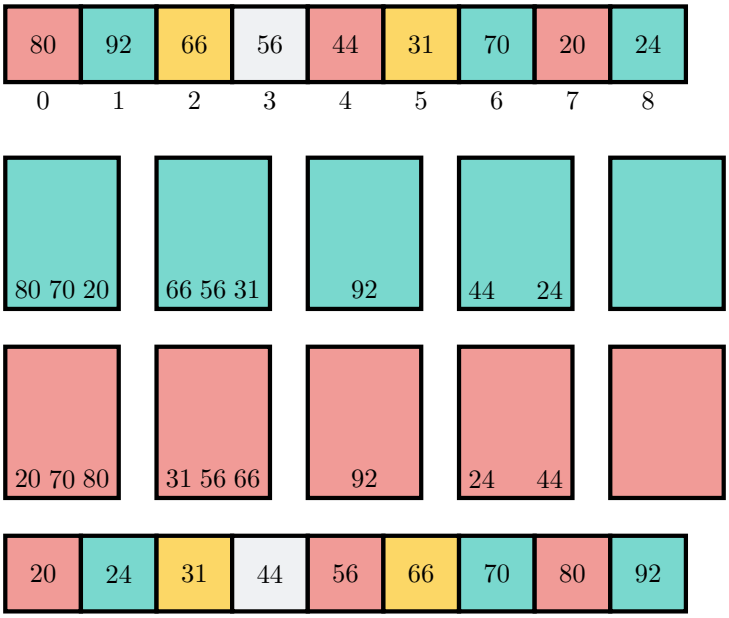
第一种非比较排序是桶排序，桶和哈希表中槽的概念是类似的，只是槽只能装一个元素，而桶可以装若干元素。槽用于保存元素，桶用于排序元素，桶排序基本思路是：

```
1 第一步，将待排序元素划分到不同的桶，先遍历求出 maxV 和 minV，
2 设桶个数为 k ，则把区间 [minV，maxV] 均匀划分成 k 个区间，
3 每个区间就是一个桶，将序列中的元素分配到各自的桶（求余法）。
4 第二步，对每个桶内的元素进行排序，排序算法可用任意排序算法。
5 第三步，将各个桶中的有序元素合并成一个大的有序集合。
```

在 Rust 里，可以定义桶为一个结构体，包含哈希函数和数据集合。

```
1 // bucket_sort.rs
2
3 // hasher 是一个函数，计算时传入
4 // values 是数据容器，保存数据
5 struct Bucket<H, T> {
6     hasher: H,
7     values: Vec<T>,
8 }
```

下图是桶排序示意图，先分散数据到桶，然后桶内排序并将各桶数据合并得到有序集合。



下面是桶排序的实现。

```
// bucket_sort.rs

impl<H, T> Bucket<H, T> {
    fn new(hasher: H, value: T) -> Bucket<H, T> {
        Bucket { hasher: hasher, values: vec![value] }
    }
}

// 桶排序, Debug 特性是为了打印 T
fn bucket_sort<H, T, F>(nums: &mut [T], hasher: F)
    where H: Ord, T: Ord + Clone + Debug, F: Fn(&T) -> H {
    let mut buckets: Vec<Bucket<H, T>> = Vec::new();

    for val in nums.iter() {
        let hasher = hasher(&val);

        // 对桶中数据二分搜索并排序
        match buckets.binary_search_by(|bct|
                                       bct.hasher.cmp(&hasher)) {
            Ok(idx) => buckets[idx].values.push(val.clone()),
            Err(idx) => buckets.insert(idx, Bucket::new(hasher,
                                                        val.clone())),
        }
    }

    // 拆桶, 将所有排序数据融合到一个 Vec
    let ret = buckets.into_iter().flat_map(|mut bucket| {
        bucket.values.sort();
        bucket.values
    }).collect::<Vec<T>>();

    nums.clone_from_slice(&ret);
}

fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    bucket_sort(&mut nums, |t| t / 5);
```

```
38         println!("sorted nums: {:?}", nums);
39     }
```

桶排序实现要复杂些，因为需要先实现桶的结构，然后再基于此结构实现排序算法。这里数据放到各个桶的依据是对 5 求余，当然可以是其他值，那么桶个数就要相应改变。求余方法 hasher 采用的是闭包函数。

假设数据是均匀分布的，则每个桶的元素平均个数为 n/k。假设选择用快速排序对每个桶内的元素进行排序，那么每次排序的时间复杂度为 O(n/klog(n/k)) 。总的时间复杂度为 O(n)+O(k)O(n/klog(n/k)) = O(n+nlog(n/k)) = O(n+nlogn-nlogk)。当 k 接近于 n 时，桶排序的时间复杂度就可以认为是 O(n) 。即桶越多，时间效率就越高，而桶越多，空间就越大，越费内存，可见这是用空间换时间。

桶排序的一个缺点是桶的数量太多。比如待排序数组 [1,100,20,9,4,8,50]，按照桶排序算法会创建 100 个桶，然而大部分桶用不上，造成了空间浪费。

FlashSort 是一种优化的桶排序，它的思路很简单，就是减少桶的数量。对于待排序数组 [1,100,20,9,4,8,50]，先找到最大最小值 A_{max}, A_{min}，然后用这两个值来估算大概需要的桶数量。思路是这样的，如果按照桶排序计算出的桶个数大于待排序元素个数，那么就通过 m = f*n 来计算桶数，f 是个小数，比如 f = 0.2。元素入桶的规则如下：

$$K(A_i) = 1 + int((m-1)\frac{A_i - A_{min}}{A_{max} - A_{min}})$$

假设有 n 个元素，每个桶平均有 n/m 个元素，对每个桶使用插入排序，总复杂度为 $O(\frac{n^2}{m})$。原作者通过实验发现，m = 0.42 n 时，性能最优，为 O(n)。当 m = 0.1n 时，只要 n > 80，这个算法就比快速排序快，当 n = 10000 时，快 2 倍。当 m = 0.2n 时，比 m = 0.1n 还快 15%。就算 m 只有 0.05n，当 n>200 是也要显著地比快速排序快。

6.11 计数排序

第二种非比较排序是计数排序，计数排序是桶排序的特殊情况，它的桶就只处理同种数据，所以比较费空间，基本思路是：

```
1 第一步，初始化长度为 maxV − minV + 1 的计数器集合，值全为 0，
2 其中 maxV为待排序集合的最大值，minV 为最小值。
3 第二步，扫描待排序集合，以当前值减 minV 作下标，并对计数器中
4 此下标的计数加 1。
5 第三步，扫描一遍计数器集合，按顺序把值写回原集合，完成排序。
```

举个例子，nums=[0,7,1,7,3,1,5,8,4,4,5]，首先遍历 nums 获取最小值和最大值，maxV=8，minV=0，于是初始化一个长度为 8 - 0 + 1 的计数器集合 counter。

$$[0, 0, 0, 0, 0, 0, 0, 0, 0]$$

接着扫描 nums，计算当前值减 minV 作为下标，如扫描到 0，则下标为 0 - 0 = 0，所以 counter 下标 0 处值加 1。counter 此时为

$$[1,0,0,0,0,0,0,0,0]$$

接着扫描到 7，下标为 7 - 0 = 7 ，所以对应位置加一，counter = [1,0,0,0,0,0,0,1,0]。继续扫描，最终 counter 为

$$[1,2,0,1,2,2,0,2,1]$$

有了 counter，那么遍历 counter 时只要某下标处数字不为 0，则将对应下标值写入 nums，counter 中值减一。比如 counter 第一个位置 0 处为 1，说明 nums 中有一个 0，此时写入 nums，继续，下标 1 处值为 2，说明 nums 中有两个 1，写入 nums。最终 nums 为

$$[0,1,1,3,4,4,5,5,7,7,8]$$

此时集合已经有序，且排序过程中不涉及比较、交换等操作，所以速度快。

```rust
// counting_sort.rs

fn counting_sort(nums: &mut [usize]) {
    if nums.len() <= 1 { return; }

    // 桶数量为 nums 中最大值加 1，保证数据都有桶放
    let max_bkt_num = nums.iter().max().unwrap() + 1;
    let mut counter = vec![0; max_bkt_num];
    for &v in nums.iter() {
        counter[v] += 1; // 将数据标记到桶
    }

    // 数据写回原 nums 切片
    let mut j = 0;
    for i in 0..max_bkt_num {
        while counter[i] > 0 {
            nums[j] = i;
            counter[i] -= 1;
            j += 1;
        }
    }
}

```

```
fn main() {
    let mut nums = [54,32,99,18,75,31,43,56,21,22];
    counting_sort(&mut nums);
    println!("sorted nums: {:?}", nums);
}
```

6.12 基数排序

第三种非比较排序是基数排序，它利用正数的进制规律来排序，基本是收集分配这样一个思路，其思想具体如下。

```
第一步，找到 nums 中最大值，得到位数，将数据统一为相同位数，
不够补零。
第二步，从最低位开始，依次进行稳定排序，收集，再排序高位，
直到排序完成。
```

举个例子，有一个整数序列，[1,134,532,45,36,346,999,102]。下面是排序过程，见图 (6.8)。第零次排序，首先找到最大值 999，三位数，所以要进行个位，十位，百位三轮排序。补 0 得到最左侧集合。第一轮对个位排序，第二轮对十位排序，第三轮对百位排序。可以看到红色位就是当前轮次排序的位，该位上的数字都是有序的。

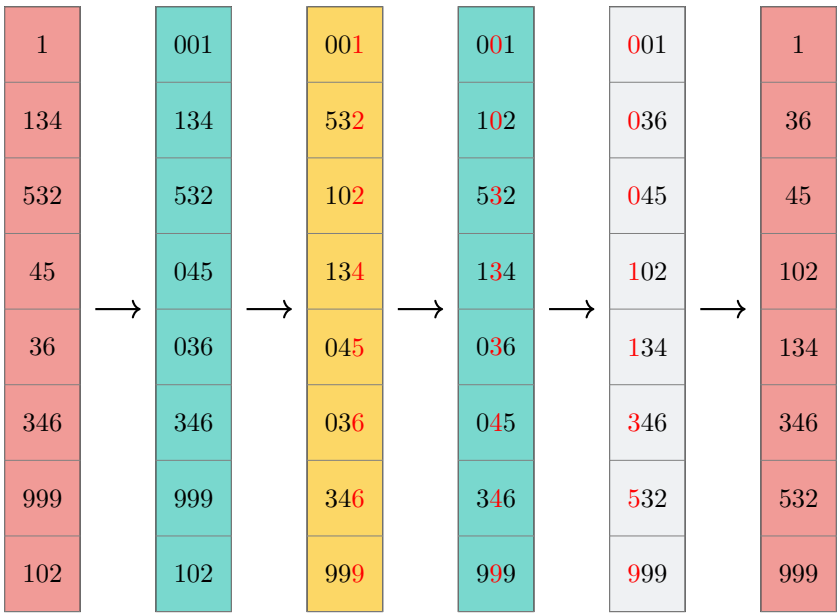


图 6.8: 计数排序

下面是基数排序的具体实现。

```rust
// radix_sort.rs

fn radix_sort(nums: &mut [usize]) {
    if nums.len() <= 1 { return; }

    // 找到最大的数，它的位最多
    let max_num = match nums.iter().max() {
        Some(&x) => x,
        None => return,
    };

    // 找最接近且 >= nums 长度的 2 的次幂值作为桶大小，如：
    // 最接近且 >= 10 的 2 的次幂值是 2^4 = 16
    // 最接近且 >= 17 的 2 的次幂值是 2^5 = 32
    let radix = nums.len().next_power_of_two();

    // digit 代表小于某个位对应桶的所有数
    // 个、十、百、千分别为在 1, 2, 3, 4 位
    // 起始从个位开始，所以是 1
    let mut digit = 1;
    while digit <= max_num {
        // index_of 计算数据在桶中哪个位置
        let index_of = |x| x / digit % radix;

        // 计数器
        let mut counter = vec![0; radix];
        for &x in nums.iter() {
            counter[index_of(x)] += 1;
        }

        for i in 1..radix {
            counter[i] += counter[i-1];
        }

        // 排序
        for &x in nums.to_owned().iter().rev() {
            counter[index_of(x)] -= 1;
            nums[counter[index_of(x)]] = x;
```

```
39              }
40
41              // 跨越桶
42              digit *= radix;
43          }
44      }
45
46      fn main() {
47          let mut nums = [54,32,99,18,75,31,43,56,21,22];
48          radix_sort(&mut nums);
49          println!("sorted nums: {:?}", nums);
50      }
```

为什么同一数位的排序要用稳定排序？因为稳定排序能将上一次排序的成果保留下来。例如十位数的排序过程能保留个位数的排序成果，百位数的排序过程能保留十位数的排序成果。能不能用 2 进制？能。可以把待排序序列中的每个整数都看成是 01 组成的二进制数值。这样任意一个非负整数序列都可以用基数排序算法解决？假设待排序序列中最大整数 64 位，则时间复杂度为 O(64n)，此时 64 也很重要，不可忽视。

既然任意一个非负整数序列都可以在线性时间内完成排序，那么基于比较排序的算法有什么意义呢？基于比较的排序算法，时间复杂度是 O(nlogn)，看起来比 O(64n) 慢，但其实不是，O(nlogn) 只有当序列非常长时 logn 才会达到 64，所以 64 这个系数太大了，基于比较的算法还是更快的。当使用 2 进制时，k=2 最小，位数最多，时间复杂度 O(nd) 会变大，空间复杂度 O(n+k) 会变小。当用最大值作为基数时，k=maxV 最大，位数最小，此时时间复杂度 O(nd) 变小，但是空间复杂度 O(n+k) 会急剧增大，此时基数排序退化成了计数排序。

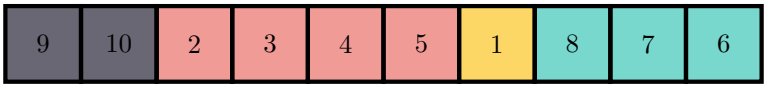
综合来看，三个非比较排序是相互有关系的。计数排序是桶排序的特殊情况，基数排序若采用最少的位来排，则此时也退化成计数排序。所以基数排序和计数排序都可以看作是桶排序，计数排序是桶取最大值时的桶排序，基数排序是每个数位上的桶排序，是多轮桶排序。当用最大值作基数时，基数排序退化成计数排序。桶排序适合元素分布均匀的场景，计数排序要求 maxV 和 minV 差距小，基数排序只能处理正数，也要求 maxV 和 minV 尽可能接近。所以，这三个排序只能排序少量的数据，最好总量小于 10000。

6.13 蒂姆排序

我们已经学习了十类排序算法，然而这些算法各有优缺点，不能很好的适合各种情况的排序。为此 Tim Peters 提出了结合多种排序的混合排序算法 TimSort。该排序算法高效，稳定且自适应数据分布，比大多数排序算法都优秀。

Tim Peters 在 2002 年提出了 TimSort，并首先在 Python 中实现为 sort 操作的默认算法，目前许多编程语言和平台都将 TimSort 或其改进版作为默认排序算法，包括 Java，Python，Rust，Android 平台。

TimSort 是一种混合的排序算法，结合了归并和插入排序，旨在更好地处理多种数据。现实中需要排序的数据通常有部分已经排好序了（包括逆序），如下图，同颜色数据是有序的。



Timsort 正是利用了这一特点来划分区块并进行排序。Timsort 称这些排好序的数据块为 run，可将其视为一个个分区，运算单元。在排序时，Timsort 迭代数据元素，将其放到不同的 run 里，同时针对这些 run，按规则合并至只剩一个，则这个仅剩的 run 即为排好序的结果。当然，为了设置合适的分区，TimSort 设置了 minrun 这个参数，即分区数据不能少于 minrun，如果小于 minrun，就利用插入排序扩充 run 到 minrun，然后再合并。

TimSort 根据排序数据集合大小产生 minrun，划分 $\frac{n}{minrun}$ 个 run，$\frac{n}{minrun}$ 小于等于 2 的次幂。若 run 的长度等于 64，则 minrun = 64，直接调用二分查找插入排序. 当集合元素个数大于 64 时，选择 [32, 64] 中某个值为 minrun。使得 $\frac{n}{minrun}$ 小于等于 2 的次幂。具体而言，选择集合长度的六个二进制位为 minrun，若剩余标志位不为 0，则 minrun 加 1。

a. 集合长度 189: 10111101，前六位 101111 = 47，剩余位 01，则 minrun = 48，n/minrun = 4。

b. 集合长度 976: 1111010000，前六位 111101 = 61，剩余位 0000，则 minrun = 61。

寻找 run，若长度小于 minrun，则调用插入排序扩充，将后面元素填充到 run。通过两两合并 run，直到只剩一个 run，则集合排序完成。

6.14 总结

本章学习了十类排序算法。冒泡及选择排序和插入排序是 $O(n^2)$ 算法，其余排序算法的复杂度多是 $O(nlog_2(n))$。选择排序是对冒泡排序的改进，希尔排序是对插入排序的改进，堆排序是对选择排序的改进，快速排序和归并排序均利用分而治之的思想。这些排序都是通过比较来排序，还有不需要比较只依靠数值规律而排序的算法，这类排序算法是非比较排序算法，分别有桶排序、计数排序、基数排序。它们的复杂度都是 O(n) 左右，适合少量数据排序。计数排序是特殊的桶排序，基数排序是多轮桶排序，基数排序可以退化成计数排序。除了基础排序算法，本章还学习了部分算法的改进版，尤其是蒂姆算法，是高效稳定的混合排序算法，其改进版已经是许多语言和平台的默认排序算法。下一页是十大排序算法的总结，读者可自行对照理解，以便加深印象。

表 6.1: 十大排序算法的时间和空间复杂度

序	排序算法	最差复杂度	最优复杂度	平均复杂度	空间复杂度	稳定性	综合类别
1	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	交换比较类
2	快速排序	$O(n^2)$	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	不稳定	交换比较类
3	选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	选择比较类
4	堆排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(1)$	不稳定	选择比较类
5	插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	插入比较类
6	希尔排序	$O(n^2)$	$O(n)$	$O(n^{1.3})$	$O(1)$	不稳定	插入比较类
7	归并排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$	稳定	分治比较类
8	计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定	非比较类
9	桶排序	$O(n^2)$	$O(n)$	$O(n+k)$	$O(n+k)$	稳定	非比较类
10	基数排序	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$	稳定	非比较类

Chapter 7

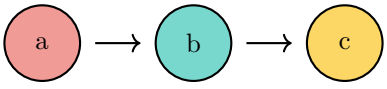
树

7.1 本章目标

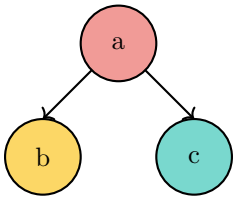
理解树及其使用方法
理解二叉树及平衡二叉树
用二叉堆实现优先级队列
实现二叉树及平衡二叉树

7.2 什么是树

前面章节我们学习了链表、栈、队列等数据结构，这些数据结构都是线性的，一个数据结点只能连接后面一项数据。

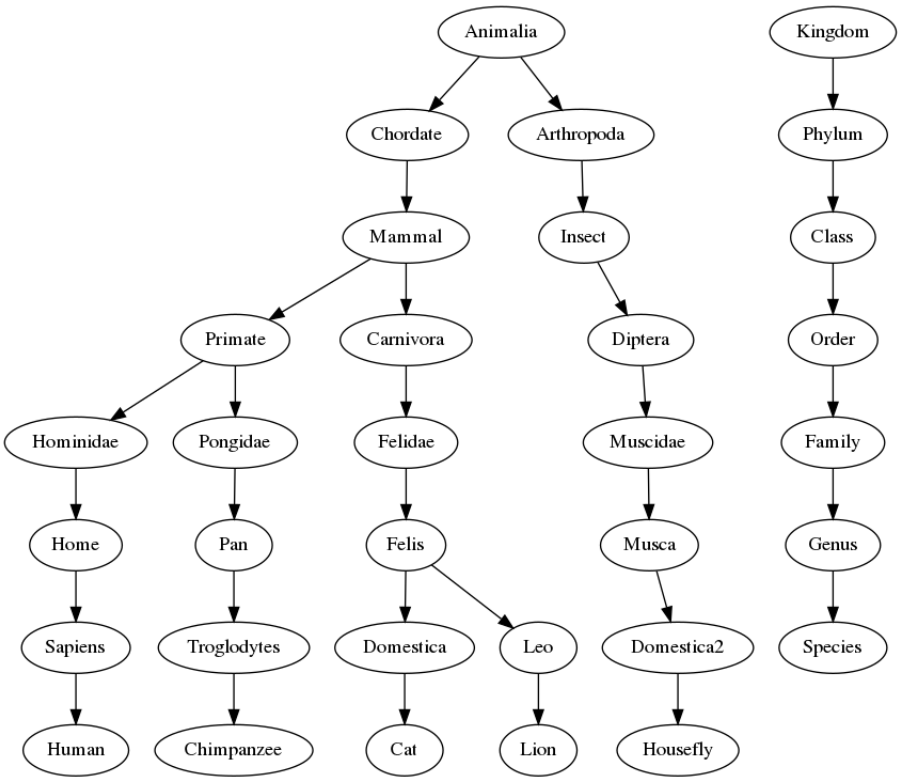


如果对线性数据进行拓展，为数据结点连接多项，那么就可以得到一种新的数据结构。



这种新的数据结构就像树一样，它有一个根，然后发散出了枝条和叶子，并且相互连接在一起。这种新的数据结构就称为树。自然界中的树和计算机科学中的树之间的区别在于树数据结构的根在顶部，叶在底部。树在计算机科学的许多领域中使用，包括操作系统，图形，数据库和计算机网络等。为简化行文，下文将树数据结构简称为树。

在开始研究树之前，先来看几个常见的树例子。第一个例子是生物学的分类树，如下图。从这幅图可以看到人具体所处的位置（左下侧），这对研究事物关系和性质非常有帮助。

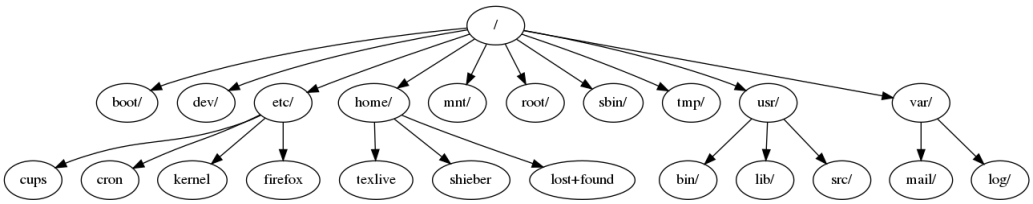


从这幅图中我们可以了解到树的属性。第一点，树是分层的。通过分层，树具有良好的层次结构，具体到这幅图就是“种、属、科、目、纲、门、界”七大层次。接近顶部的是抽象层次最高的事物，底层是最具体的事物。人种很具体，但动物界就太抽象了，包含所有动物，不仅限于人，还有虫、鱼、鸟、兽。从这棵树的根部开始，沿着箭头一直走到底部，会给出一条完整的路径，该路径表明了底层物种的全称。比如人只是简称，全称是“动物界-脊椎门-哺乳纲-灵长类目-人科-人属-智人种”。每一个生物都能在这棵生命大树上找到自己的位置，并显示出相对关系。

树的第二个属性是一个节点的所有子节点独立于另一个节点的子节点。智人种这个子节点不会属于昆虫纲及其子节点，这使得彼此之间关系明确，同时也意味着改变一个节点的子节点对其他节点没有影响。比如发现新的昆虫，这使得生命树又庞大了，但人科下面毫无变化，整个生物学知识的更新也不会涉及到人科。这种性质非常有用，尤其是树作为数据存储容器的时候，可以借助工具来修改某些结点上的数据而保持其他数据不变。

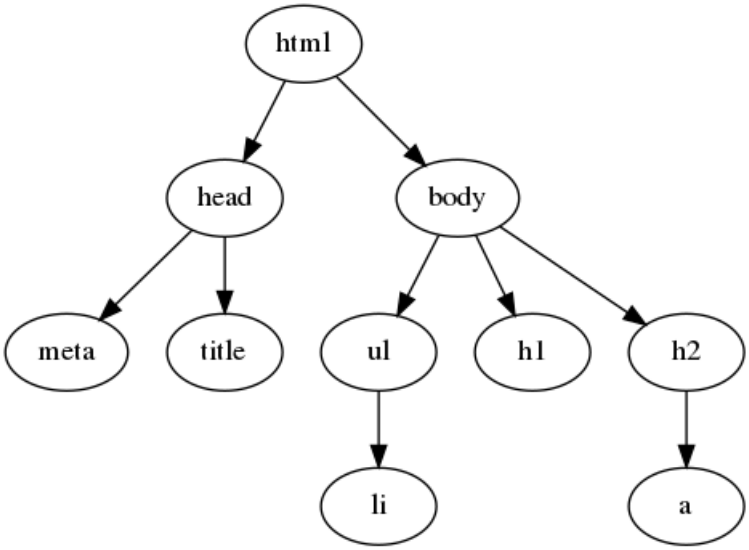
树的第三个属性是每个叶节点是唯一的。你可以从树的根到叶子节点找出一条唯一的路径，这种性质使得保存数据非常有效，既然路径唯一，那么就可以用来作为数据的存储路径。实际上，我们电脑里的文件系统就是通过改进的树来保存文件的。文件系统树与生物分类树有很多共同之处，从根目录到任何子目录的路径唯一标识了该子目录及其中的所有文件。如

果你使用类 Unix 操作系统，那么你应该熟悉类似 /root，/home/user，/etc 这样的路径，这些路径都是树上的节点，显然 / 是根节点，是树根。



还有一个使用树的例子是网页文件。网页是资源的集合，也具有树的层次结构。下面是 google.cn 的查找界面网页数据，可以看到这些 <> 括号标签也是有层次的。

```
<html lang="zh"><head><meta charset="utf-8">
  <head>
      <meta charset="utf-8">
      <title>Google</title>
      <style>
        html { background: #fff; margin: 0 1em; }
        body { font: .8125em/1.5 arial, sans-serif; }
      </style>
  </head>
  <body>
  <div>
    <a href="http://www.google.com.hk/webhp?hl=zh-CN">
      <img src="//www.google.cn/landing/cnexp/search.png"
        alt="Google" width="586" height="257">
    </a>
    <h1><a href="http://www.google.com.hk/webhp?hl=zh-CN">
        <strong id="target">google.com.hk</strong></a></h1>
    <p>请收藏我们的网址</p>
  </div>
  <ul>
    <li><a href="http://translate.google.cn/">翻译</a></li>
  </ul>
  <p id="footer">©2011 - <span>ICP证合字B2-20070004号</span>
  </p>
  </body>
</html>
```



7.2.1 树的定义

我们已经看了树的示例，现在来定义树的各种属性。

节点：节点是树的基本部分，它还有一个名称叫做“键”。节点也可以有附加信息，附加信息称为“有效载荷”。虽然有效载荷信息不是许多树算法的核心，但在利用树的应用中通常是关键的，比如树节点上存储时间，文件名等。

根：根是树中唯一没有传入边的节点，它处于顶层，所有的节点都可以从根找到，类似操作系统的 / 或 C 盘这样的概念。

边：边是树的另一个基本部分，又叫分支。边连接两个节点以保持之间存在的关系。每个节点，除根之外，都恰好有一个输入边和若干输出边。边就是路径，可以通过它来找到某个节点的具体位置。

路径：路径是由边连接节点的有序序列，它本身并不存在，是由其逻辑结构涌现出来的一种信息。比如 /home/user/files/sort.rs 就是一条路径，它标识了 sort.rs 这个文件的具体位置。

子节点：子节点是某个节点的下一级，所有子节点都源自同一个上层节点。比如上面的 sort.rs 就是 files/ 的子节点。子节点不唯一，可以存在零个、一个、多个子节点，这和人类社会的父子关系一样，所以名字也借用了人类的亲属关系称谓词。

父节点：父节点是所有下级节点的源，所有子节点都源自同一个父节点。比如 files/ 就是 sort.rs 的父节点。父节点唯一，这很好理解，一个孩子只可能有一个父亲。

子树：子树是由父节点和该父节点的所有后代组成的一组节点和边。因为树这种递归结构，从树中任意节点取出一部分，它的结构都仍然是树，这部分取出的内容就称为子树。

叶节点：叶节点是没有子节点的节点，处于最底层。

中间节点：中间节点有子节点，有父节点。

层数：节点 n 的层数为从根到该结点所经过的分支数目。根节点的层数为零，/home-

/user/files 中，files 的层数为 2。层数为零不代表没有层数，而是层数就是零，此零不是无，没有的意思，而是第零层。

高度：树的高度等于树中任何节点的最大层数。

有了这些基础知识就可以给出树的定义。

```
1 树具有一个根节点。
2 除根节点外，每个节点通过其他节点的边互相连接父和子节点（若有）。
3 从根遍历到任何节点的路径全局唯一。
```

下图为一个树结构，左右子节点为 lc 和 rc。因为树的结构是递归的，所以子树的结构和父树一致。

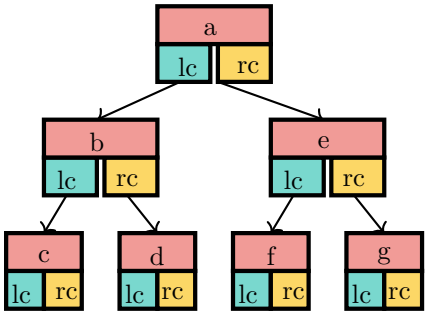


图 7.1: 树的节点表示

7.2.2 树的表示

树是一种非线性数据结构，然而计算机的存储硬件都是线性的。所以，计算机要表示树必然涉及用线性结构来表征非线性结构。一种表征方法是用数组来构成树，下面的 tree 就是通过数组来构造的。

```
1  tree = [ 'a',
2             ['b',
3               ['c',[],[]],
4               ['d',[],[]],
5             ],
6             ['e',
7               ['f',[],[]],
8               []
9             ],
10          ]
```

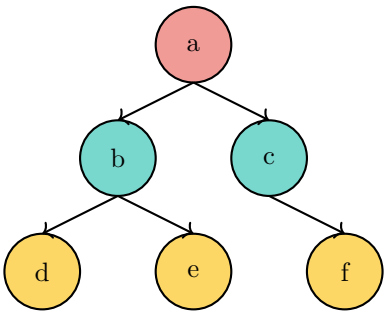
从这个数组结构可以看到树是如何保存在数组中的。我们知道数组在内存中是连续的，所以这个数在内存中也是连续的，可以利用数组的访问方式来获取数据。比如 tree[0] 就是'a',

左子树是 tree[1]，包含'b', 'c', 'd'，而该子树还是数组，继续使用数组访问方法，则 tree[1][0] 就是数据'c'。数组表示方法的一个好处是，表示子树的数组仍然遵循树的定义，结果是这整个结构本身是递归的，可以不断获取子树及元素。

```
println!("root {:?}", tree[0]);
println!("left subtree {:?}", tree[1]);
println!("right subtree {:?}", tree[2]);
```

用数组保存树虽然可行，但是嵌套太深，十分复杂。如果有十层，那么获取子树和元素非常麻烦，试想获取第四层的某个叶节点，你得用类似 tree[1][1][1][2] 这种形式访问，这对计算机和对人来说都太复杂了。所以用数组保存树，理论上可行，实际上不行。

另外一种可行的树保存方式是节点（Node）。回想链表一节的内容，我们发现链表节点和此处的节点概念是一致的，且链表里的链就是树里的边，就像下图。



这种结构看起来非常直观，且不用嵌套，避免了元素访问的麻烦。现在的关键是：如何定义树节点？有了根结点才能保存其子节点，一种可行的办法是使用 struct 定义节点。

```
// binary_tree.rs
use std::fmt::{Debug, Display}

// 子节点链接
type Link<T> = Option<Box<BinaryTree<T>>>;

// 二叉树定义
// key 保存数据
// left 和 right 保存左右子节点链接
#[derive(Debug, Clone)]
struct BinaryTree<T> {
    key: T,
    left: Link<T>,
    right: Link<T>,
}
```

key 中保存数据，left 和 right 保存左右子节点的地址，这样就可以通过访问地址获取子节点。可以通过插入函数来为根加入子节点。

```
// binary_tree.rs

impl<T: Clone> BinaryTree<T> {
    fn new(key: T) -> Self {
        BinaryTree { key: key, left: None, right: None }
    }

    // 新子节点作为根节点的左子节点
    fn insert_left_tree(&mut self, key: T) {
        if self.left.is_none() {
            let node = BinaryTree::new(key);
            self.left = Some(Box::new(node));
        } else {
            let node = BinaryTree::new(key);
            node.left = self.left.take();
            self.left = Some(Box::new(node));
        }
    }

    // 新子节点作为根节点的右子节点
    fn insert_right_tree(&mut self, key: T) {
        if self.right.is_none() {
            let node = BinaryTree::new(key);
            self.right = Some(Box::new(node));
        } else {
            let node = BinaryTree::new(key);
            node.right = self.right.take();
            self.right = Some(Box::new(node));
        }
    }
}
```

插入子节点时，必须考虑两种情况。第一种情况是节点没有子节点，此时直接插入就行。第二种情况是节点具有子节点，则先将子节点接到新节点的子节点位置，然后再将新节点作为根的子节点。

代码里 node = BinaryTree 可能让你感觉混淆，怎么又是节点又是树？实际上，我们定

义的 BinaryTree 看起来是一个节点，但多个节点链接起来后节点又成了树。当插入节点时，可以将整个子树看成一个节点，这样便于操作。但在使用时，它本身又是棵树，可以获取内部节点信息。最好的理解方法就是，节点是树的一部分，它本身也可用树来表示，来插入，来移动，因为树是递归的。但使用时，其内部结构很重要，所以写的是 BinaryTree 而不是 Node。

为获取二叉树节点数据，可以为其实现获取左右子节点以及根节点值的函数。

```
// binary_tree.rs

impl<T: Clone> BinaryTree<T> {
    // 获取左右子节点及根节点，注意使用了 clone
    fn get_left(&self) -> Link<T> {
        self.left.clone()
    }

    fn get_right(&self) -> Link<T> {
        self.right.clone()
    }

    fn get_key(&self) -> T {
        self.key.clone()
    }

    fn set_key(&mut self, key: T) {
        self.key = key;
    }
}
```

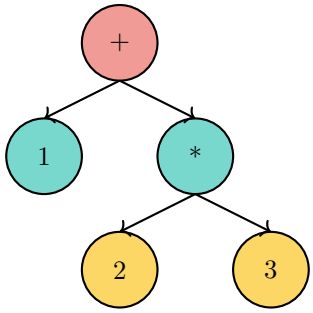
有了创建和操作二叉树的函数，让我们使用它来创建上面提到的二叉树吧。这个二叉树非常简单，只使用插入函数就能构造。

```
// binary_tree.rs

fn main() {
    let mut bt = BinaryTree::new('a');

    let root = bt.get_key();
    println!("root val is {:?}", root);

    let left = bt.get_left();
    println!("left child is {:#?}", left);
```

```
11      let right = bt.get_right();
12      println!("left child is {:#?}", right);
13
14      bt.insert_left_tree('b');
15      bt.insert_right_tree('e');
16
17      let left = bt.get_left();
18      println!("left child is {:#?}", left);
19      let right = bt.get_left();
20      println!("right child is {:#?}", right);
21  }
```

7.2.3 分析树

有了树的定义和操作函数，现在可以来思考树在保存数据时的工作原理，比如用树来存储类似 (1 + (2 * 3)) 这种式子。前面已经学过完全括号表达式，通过括号可以指示优先级。同样，由于有括号，所以可以指明符号保存的顺序。如果将算术表达式表示成树，那么应该类似下面这样的树结构。



那么，树是如何把数据保存进去的呢？根据树的结构，可以定义一些规则，然后按照规则把数据保存到节点上。定义规则如下:

```
1  若当前符号是 '(', 添加新节点作为左子节点, 并下降到左子节点。
2  若当前符号在 ['+','-','/','*'] 中, 将根值置为当前符号。
3      添加新右子节点, 下降到该子节点。
4  若当前符号是数字, 将根值置为该数字, 返回到父节点。
5  若当前符号是 ')', 则转到当前节点的父节点。
```

利用这套规则，我们可以将 (1 + (2 * 3)) 转换成上图所示的树。具体步骤如下，灰色表示当前节点。

a. 创建根节点。

b. 读取符号 (，创建新左子节点，下降到该节点。

c. 读取符号 1，将节点值置为 1，返回父节点。

d. 读取符号 +，将节点值置为 +，创建新右子节点，下降到该节点。

e. 读取符号 (，创建新左子节点，下降到该节点。

f. 读取符号 2，将节点值置为 2，返回父节点。

g. 读取符号 *，将节点值置为 *，创建新右子节点，下降到该节点。

h. 读取符号 3，将节点值置为 3，返回父节点。

i. 读取符号)，返回父节点。

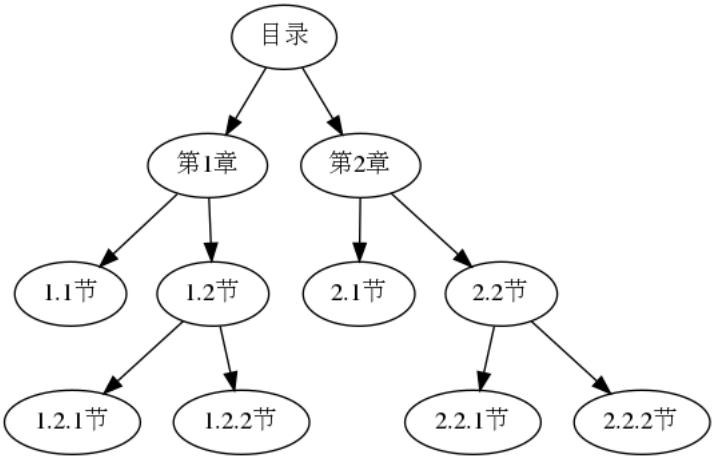
算术表达式保存需要关注子节点和父节点，然而我们没有实现获取父节点的功能，所以可以采用栈来保存当前节点的父节点，或者实现获取父节点的功能 get_parent()。

我们用树完成了算术表达式的二叉树保存，树能维持数据的结构信息。实际上，编程语言在编译时，也是用树来保存所有的代码并生成抽象语法树。通过分析语法树各部分功能，生成中间代码，优化，生成最终代码。如果你了解编译原理，那么这对你来说应该很熟悉。

7.2.4 树的遍历

保存数据的目的是为了更高效的使用数据，包括增加、删除、查找、修改。其中增删改的前提是要找到数据所在位置，对于树来说，是定位具体的节点。所以查找，访问或查找节点是首先要完成的功能。不同于线性数据结构，可以通过下标遍历所有数据。树是非线性结构，所以树的查找方法不同于栈、数组这类线性数据结构。

有三种常用的方法来访问树节点，这些方法间的差异主要是节点被访问的顺序。参照线性数据结构的遍历方法，我们也称树的节点访问方法为遍历。这三种遍历方法分别是前序遍历、中序遍历、后序遍历。首先用一个例子来说明这三种遍历。假如把这本书表示为树，那么目录是根，各章是其子节点，而小节是各章各自的子节点，依此类推。



想象自己读本书的顺序，是不是按照第一章第一节，第二节；第二章第一节，第二节这样的顺序？你可以看看各章节所在位置，并从目录开始按照看书的顺序在图中勾勒出阅读轨迹，这种顺序就是前序遍历 preorder。前序遍历从根节点开始，左子树继后，右子树最后。

算法遍历时，从根节点开始，递归调用左子树前序遍历。对于上面的树，preorder 得出的数据访问顺序和目录的线性顺序一致。先访问目录，然后第一章第一节，第二节... 一章完后，回到目录，选择第二章继续递归调用 preorder，访问第二章第一节，第二节...。

前序遍历算法理解起来不复杂，实现也很简单，利用递归就可以做到。下面是单独实现的函数，可以前序遍历树。

```
// 前序遍历： 外部实现
fn preorder<T: Clone + Debug>(bt: Link<T>) {
    if !bt.is_none() {
        println!("key is {:?}", bt.as_ref().unwrap().get_key());
        preorder(bt.as_ref().unwrap().get_left());
        preorder(bt.as_ref().unwrap().get_right());
    }
}
```

当然，也可以将遍历方法直接实现为 BinaryTree 的内部函数。

```
// binary_tree.rs

impl<T: Clone> BinaryTree<T> {
    // 前序遍历：内部实现
    fn preorder(&self) {
        println!("kes: {:?}", &self.key);
        if !self.left.is_none() {
            self.left.as_ref().unwrap().preorder();
        }
        if !self.right.is_none() {
            self.right.as_ref().unwrap().preorder();
        }
        // as_ref() 获取节点引用，因为打印不能更改节点
    }
}
```

后序遍历和前序遍历类似，它先查看左子树，然后右子树，最后根节点。

```
// binary_tree.rs 后序遍历：内外部实现

fn postorder(bt: Link<T>) {
    if !bt.is_none() {
        postorder(bt.as_ref().unwrap().get_left());
        postorder(bt.as_ref().unwrap().get_right());
```

```
7           println!("key is {:?}", bt.as_ref().unwrap().get_key());
8       }
9   }
10
11  impl<T: Clone> BinaryTree<T> {
12      fn postorder(&self) {
13          if !self.left.is_none() {
14              self.left.as_ref().unwrap().postorder();
15          }
16          if !self.right.is_none() {
17              self.right.as_ref().unwrap().postorder();
18          }
19          println!("kes is {:?}", &self.key);
20      }
21  }
```

现在回到前一节的算术表达式 (1 + (2 * 3))，我们用树保存了这个表达式，如果要计算它，总是需要先取到操作符和操作数，然后施加运算。要获得这三个值，需要正确的顺序。前序遍历会尽可能的从根节点开始，然而计算表达式要从叶节点数据开始，所以后序遍历才是正确的数据获取方法。通过先获取左右子节点值，再获取根节点操作符，可以做一次运算，并将该结果保存在运算符号所在位置，然后继续后序遍历，以先前计算的值为左节点，然后访问右子节点，直到计算出最终值。

最后一种遍历是中序遍历。中序遍历首先访问左子树，然后访问根，最后访问右子树。

```
1   // binary_tree.rs 中序遍历：内外部实现
2
3   fn inorder(bt: Link<T>) {
4       if !bt.is_none() {
5           inorder(bt.as_ref().unwrap().get_left());
6           println!("key is {:?}", bt.unwrap().get_key());
7           inorder(bt.as_ref().unwrap().get_right());
8       }
9   }
10
11  impl<T: Clone> BinaryTree<T> {
12      fn inorder(&self) {
13          if !self.left.is_none() {
14              self.left.as_ref().unwrap().inorder();
15          }
```

```
16            println!("kes is {:?}", &self.key);
17            if !self.right.is_none() {
18                self.right.as_ref().unwrap().inorder();
19            }
20        }
21    }
```

对保存算术表达式 (1 + (2 * 3)) 的树使用中序遍历可以得到原来的表达式 1 + 2 * 3。注意，因为树没有保存括号，恢复的表达式只是顺序正确，但优先级不一定对，可以修改中序遍历使得输出包含括号。

```
1  // binary_tree.rs
2
3  // 按照节点位置返回节点组成的字符串
4  fn get_exp<T: Clone + Debug + Display>(bt: Link<T>) -> Sring {
5      let mut exp = "".to_string();
6      if !bt.is_none() {
7          exp = "(".to_string() + &get_exp(bt.unwrap().get_left());
8          exp += &bt.as_ref().unwrap().get_key().to_string();
9          exp += &(get_exp(bt.as_ref().unwrap().get_right()) + ")");
10     }
11
12     exp
13 }
```

让我们简化三种遍历访问顺序的描述，前序遍历简化成“根左右”，表示先访问根，再访问左子树，最后访问右子树。综合三种遍历可以得出，根左右是前序遍历，左根右是中序遍历，左右根是后序遍历。实际上，还可以有根右左这种访问顺序，但这是前序遍历的镜像。因为左和右是相对的，所以左根右和右根左可以看成互为镜像。同理，右左根是后序遍历的镜像；右根左是中序遍历的镜像。

表 7.1: 前中后序遍历及其镜像遍历总结

序	遍历方法	遍历顺序	镜像遍历顺序	镜像遍历方法
1	前序遍历	根左右	根右左	前序镜像遍历
2	中序遍历	左根右	右根左	中序镜像遍历
3	后序遍历	左右根	右左根	后序镜像遍历
4	前序镜像遍历	根右左	根左右	前序遍历
5	中序镜像遍历	右根左	左根右	中序遍历
6	后序镜像遍历	右左根	左右根	后序遍历

7.3 基于二叉堆的优先队列

在前面的章节中，我们学习了队列这种先进先出的线性数据结构。队列的一个变种称为优先级队列，它的作用就像一个队列，你可以通过从队首出队数据项。然而，在优先级队列中，项的顺序不是按照从末尾加入的顺序，而是由数据项的优先级确定。最高优先级项在队列的首部，最先出队。因此，将项加入优先级队列时，如果新项的优先级够高，那么它会一直往队首移动，其实就是利用某个指标来排序，使得该项排到前面去。

优先级队列是很有用的一种数据结构，尤其对于涉及到优先级的事务，用这种队列管理就非常有效。比如，操作系统会调度各个进程，那么哪个进程该排在前面呢？这时优先级队列就非常有用了。通过某种算法，操作系统可以得到进程的优先级，然后据此排序。比如，你在用手机听音乐，同时还在浏览新闻，这时一个电话打过来，那么系统会将电话直接提到最高优先级，直接打断新闻浏览界面和音乐，展示一个来电呼叫界面。这就是利用优先级队列来管理的进程，直接赋予来电呼叫很高的优先级。

如果叫你来实现这个优先级队列，你会用什么办法呢？这种优先级队列一定是根据某种规则排序，把高优先级项排到最前面。然而，插入队列复杂度是 O(n)，且排序队列复杂度至少也有 O(nlogn)。要做得更快的话，可以采取用堆来排序。堆其实是一种完全二叉树，所以用来实现优先队列的堆又称为二叉堆。二叉堆允许在 O(logn) 时间内排队和出队，这对于高效的调度系统是非常重要的。

二叉堆是很有趣的一种结构，虽然它定义上是二叉树，但我们不必真的用链接的节点来实现二叉堆，相反我们采取前面提到过的用数组、切片或 Vec 这类线性数据结构来实现。只要我们的操作是按照二叉树的定义，那么线性数据结构一样可以当成非线性数据结构来用。其实真正的树在内存里也是线性放置的，因为内存本身就是线性的，只是使用时采取非线性方式。

二叉堆有两种常见的形态，一种是最小堆，或称小顶堆，最小的数据项在堆顶；另一种是最大堆，或称大顶堆，最大数据项在堆顶。不管大顶还是小顶堆，算法逻辑除了取大或取小外没有差别。

7.3.1 二叉堆定义

根据前面的描述，我们选择定义小顶二叉堆的基本操作如下。

new() 创建一个新的二叉堆，不需要参数，返回空堆。

push(k) 向堆添加一个新项，需要参数 k，不返回任何内容。

pop() 返回堆的最小项，从堆中删除该项，不需要参数，修改堆。

min() 返回堆的最小项，不需要参数，不修改堆。

size() 返回堆中的项数，不需要参数，返回数字。

is_empty() 返回堆状态，不需要参数，返回布尔值。

build(arr) 从数组或 vec 构建新堆，需要保存数据的参数 arr。

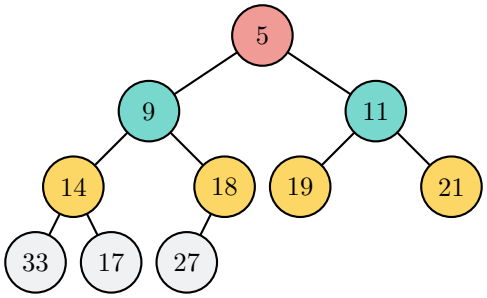
假设 h 是已经创建的二叉堆，下表展示了堆操作序列后的结果，堆顶在右边。此处将加入项的值作为优先级，越小则越优先，所以小的在右侧。

表 7.2: 二叉堆操作

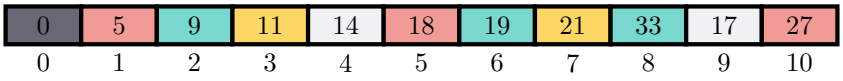
堆操作	堆当前值	操作返回值	堆操作	堆当前值	操作返回值
h.is_empty()	[]	true	h.is_empty()	[8,6,3]	false
h.push(8)	[8]		h.push(4)	[8,6,4,3]	
h.push(6)	[8,6]		h.min()	[8,6,4,3]	3
h.min()	[8,6]	6	h.pop()	[8,6,4]	3
h.push(3)	[8,6,3]		h.pop()	[8,6]	4
h.size()	[8,6,3]	3	h.build([1,2])	[8,6,2,1]	

7.3.2 Rust 实现二叉堆

为使二叉堆高效工作，可以利用其对数性质。二叉堆采用线性数据结构来保存，为保证性能，必须保持二叉堆平衡。平衡二叉堆在根的左右子树中具有大致相同数量的节点，它尽量将每个节点的左右子节点填满，最多有一个节点的子节点不满。



用 Vec 保存二叉堆，因父子节点处于线性数据结构中，所以父子节点的关系易于计算。一个节点若处于下标 p，则其左子节点在 2p，右子节点在 2p + 1 ，p 为从 1 开始的下标，下标为 0 处不放数据，直接置 0 占位。Vec [0,5,9,11,14,18,19,21,33,17,27]，其树表示如下。



可以看到 5 的下标 p 为 1，左子节点在下标 2*p = 2 处，此处值为 9，树结构图中 9 正好是 5 的左子节点。同样的，任意子节点的父节点位于下标 p/2 处，比如 9 的下标 p = 2，则父节点在 2/2 = 1 处，右子节点 11 的下标为 p = 3，则它的父节点在 3/2 = 1 处，除法值向下取整。综上，任意子节点的父节点计算只用一个计算表达式 p/2，而子节点的计算为 2p 和 2p + 1。前面我们定义过宏来计算父子节点下标，此处我们依然使用宏来计算。

```
1 // binary_heap.rs
2 macro_rules! parent { //计算父节点下标
3     ($child:ident) => {
4         $child >> 1
```

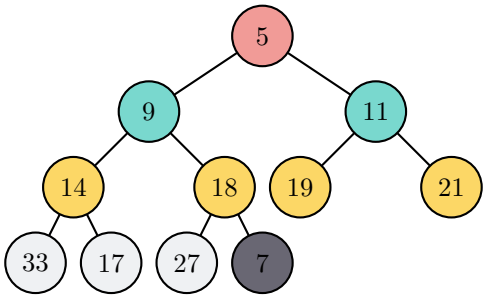
```
5       };
6   }
7   macro_rules! left_child { //计算左子节点下标
8       ($parent:ident) => {
9           $parent << 1
10      };
11  }
12  macro_rules! right_child { //计算右子节点下标
13      ($parent:ident) => {
14          ($parent << 1) + 1
15      };
16  }
```

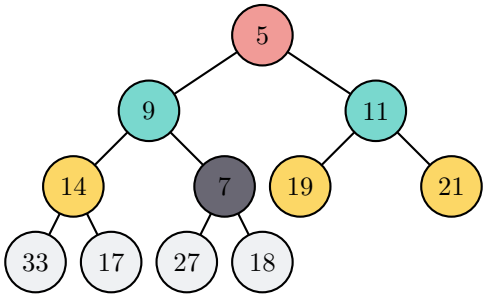
首先定义二叉堆。为跟踪堆大小情况，添加了一个表示数据项的字段 size，注意第一个数据 0 不算。初始化时，下标 0 处有数据但 size 也置为 0，此处保存的数据默认为 i32。

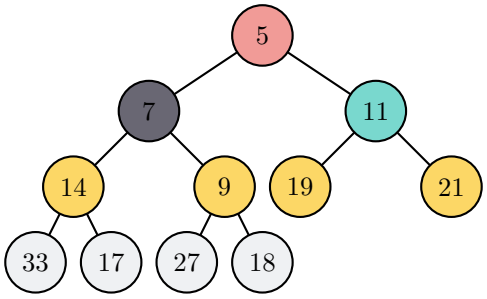
```
1   // binary_heap.rs
2
3   // 二叉堆定义
4   #[derive(Debug, Clone)]
5   struct BinaryHeap {
6       size: usize,      // 数据量
7       data: Vec<i32>, // 数据容器
8   }
9
10  impl BinaryHeap {
11      fn new() -> Self {
12          BinaryHeap {
13              size: 0, // vec 首位置 0，但不计入总数
14              data: vec![0]
15          }
16      }
17
18      fn len(&self)  -> usize {
19          self.size
20      }
21
22      fn is_empty(&self) -> bool {
23          0 == self.size
```

```
24          }
25
26          // 获取堆中最小数据
27          fn min(&self) -> Option<i32> {
28              if self.size == 0 {
29                  None
30              } else {
31                  // Some(self.data[1].clone()); 泛型数据用 clone
32                  Some(self.data[1])
33              }
34          }
35      }
```

有了空堆，可以加入数据项。在堆尾加入数据可能破坏平衡，所以需要向上移动到堆顶。







每加入一个数据，size 加一，然后从此开始往上移动 move_up （若需要）以维持平衡。

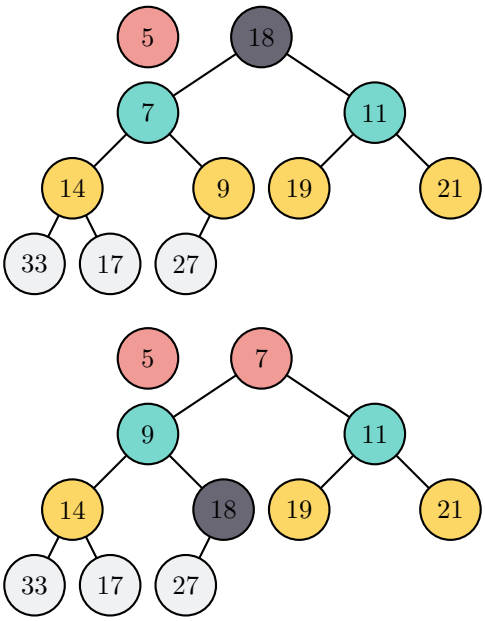
```
// binary_heap.rs

impl BinaryHeap {
    // 末尾添加数据，调整堆
    fn push(&mut self, val: i32) {
        self.data.push(val);
        self.size += 1;
        self.move_up(self.size);
    }

    // 小数据上冒
    fn move_up(&mut self, mut c: usize) {
        loop {
            let p = parent!(c);
            if p <= 0 { break; }

            if self.data[c] < self.data[p] {
                self.data.swap(c, p);
            }
            c = p;
        }
    }
}
```

假设要获取堆最小值，则要考虑三种情况：堆中无数据返回 None；堆中有一个数据，直接弹出；堆中有多项数据，交换堆顶和堆尾数据，调整堆，之后返回末尾最小值。下面实现了元素向下移动功能以维持平衡，min_child 用于找出最小子节点。

```
// binary_heap.rs

impl BinaryHeap {
    fn pop(&mut self) -> Option<i32> {
        if 0 == self.size { // 没数据，返回 None
            None
        } else if 1 == self.size {
            self.size -= 1; // 一个数据，比较好处理
            self.data.pop()
        } else { // 多个数据，先交换并弹出数据，再调整堆
```

```
11              self.data.swap(1, self.size);
12              let val = self.data.pop();
13              self.size -= 1;
14              self.move_down(1);
15              val
16          }
17      }
18
19      // 大数据下沉
20      fn move_down(&mut self, mut c: usize) {
21          loop {
22              let lc = left_child!(c);
23              if lc > self.size { break; }
24
25              let mc = self.min_child(c);
26              if self.data[c] > self.data[mc] {
27                  self.data.swap(c, mc);
28              }
29              c = mc;
30          }
31      }
32
33      // 最小子节点位置
34      fn min_child(&self, i: usize) -> usize {
35          let (lc, rc) = (left_child!(i), right_child!(i));
36          if rc > self.size {
37              lc
38          } else if self.data[lc] < self.data[rc] {
39              lc
40          } else {
41              rc
42          }
43      }
44  }
```

下面来看看删除堆中最小元素的过程，如下图。首先将堆顶部的元素拿出，并将最后一个元素移动到堆顶部。此时堆顶不是最小值，所以不满足堆定义，需要重新建堆。此时建堆需要将顶部元素其向下移动 move_down，通过节点下标计算宏来计算该和左还是右字节点交换，最终顶部元素 18 和 7 左子节点交换，重复这个过程，直到 18 到达某个节点。



除了一个个地 push 到堆中外，还可以通过对集合集中处理。比如 [5,4,3,1,2] 这个切片，可以一次性加入堆中，避免频繁调用 push 函数。假设原始堆中有数据 [6,7,8,9,10]，则加入切片数据有两种方式，一是保持原始数据，切片数据为新加，则最终堆为 [1,2,3,4,5,6,7,8,9,10]。二是删除原始数据，再添加切片元素，则最终堆为 [1,2,3,4,5]。

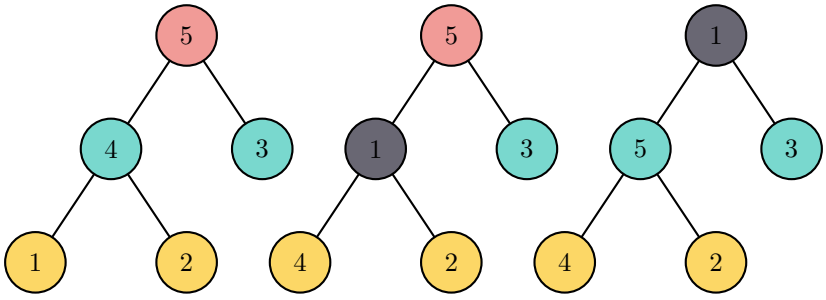


图 7.2: 删除数据并重新构建堆

为实现这两种功能，分别定义了 build_new 和 build_add 函数如下。

```
1  // binary_heap.rs
2
3  impl BinaryHeap {
4      // 构建新堆
5      fn build_new(&mut self, arr: &[i32]) {
6          // 删除原始数据
7          for _i in 0..self.size {
```

```
8                 let _rm = self.data.pop();
9             }
10
11            // 添加新数据
12            for &val in arr {
13                self.data.push(val);
14            }
15            self.size = arr.len();
16
17            // 调整小顶堆
18            let size = self.size;
19            let mut p = parent!(size);
20            while p > 0 {
21                self.move_down(p);
22                p -= 1;
23            }
24        }
25
26        // 切片数据逐个加入堆
27        fn build_add(&mut self, arr: &[i32]) {
28            for &val in arr {
29                self.push(val);
30            }
31        }
32    }
```

至此我们完成了二叉小顶堆的构建，整个过程理解起来应该非常简单。当然，你可以据此写出大顶堆的代码。二叉小顶堆的完整代码如下。

```
1   // binary_heap.rs 完整代码
2
3   // 计算节点下标的宏
4   macro_rules! parent {
5       ($child:ident) => {
6           $child >> 1
7       };
8   }
9
10  macro_rules! left_child {
```

```
    ($parent:ident) => {
        $parent << 1
    };
}

macro_rules! right_child {
    ($parent:ident) => {
        ($parent << 1) + 1
    };
}

// 定义
#[derive(Debug, Clone)]
struct BinaryHeap {
    size: usize,     // 数据量
    data: Vec<i32>, // 数据容器
}

impl BinaryHeap {
    fn new() -> Self {
        BinaryHeap {
            data: vec![0],
            size: 0,
        }
    }

    fn len(&self)  -> usize {
        self.size
    }

    fn is_empty(&self) -> bool {
        0 == self.size
    }

    // 获取堆中最小数据
    fn min(&self) -> Option<i32> {
        if self.size == 0 {
            None
```

```
49          } else {
50              Some(self.data[1])
51          }
52      }
53
54      // 末尾添加一个数据，调整堆
55      fn push(&mut self, val: i32) {
56          self.data.push(val);
57          self.size += 1;
58          self.move_up(self.size);
59      }
60
61      // 小数据上冒 c(child), p(parent)
62      fn move_up(&mut self, mut c: usize) {
63          loop {
64              // 计算当前节点的父节点
65              let p = parent!(c);
66              if p <= 0 { break; }
67
68              // 当前节点数据小于父节点数据，交换
69              if self.data[c] < self.data[p] {
70                  self.data.swap(c, p);
71              }
72
73              // 父节点成为当前节点
74              c = p;
75          }
76      }
77
78      fn pop(&mut self) -> Option<i32> {
79          if 0 == self.size { // 没数据，返回 None
80              None
81          } else if 1 == self.size {
82              self.size -= 1; // 一个数据，比较好处理
83              self.data.pop()
84          } else { // 多个数据，先交换并弹出数据，再调整堆
85              self.data.swap(1, self.size);
86              let val = self.data.pop();
```

```
87                 self.size -= 1;
88                 self.move_down(1);
89                 val
90             }
91         }
92
93         // 大数据下沉 l(left), r(right)
94         fn move_down(&mut self, mut c: usize) {
95             loop {
96                 // 计算当前节点的左子节点位置
97                 let lc = left_child!(c);
98                 if lc > self.size { break; }
99
100                // 计算当前节点的最小子节点位置
101                let mc = self.min_child(c);
102
103                // 当前节点数据大于最小子节点数据, 交换
104                if self.data[c] > self.data[mc] {
105                    self.data.swap(c, mc);
106                }
107
108                // 最小子节点成为当前节点
109                c = mc;
110            }
111        }
112
113        // 计算最小子节点位置
114        fn min_child(&self, c: usize) -> usize {
115            // 同时计算左右子节点位置
116            let (lc, rc) = (left_child!(c), right_child!(c));
117
118            // 1.如果右子节点位置超过 size, 表示只有左子节点
119            // 则左子节点就是最小子节点
120            // 2.否则, 同时存在左右子节点, 需具体判断左右子
121            // 节点数据大小, 然后返回最小的子节点位置
122            if rc > self.size {
123                lc
124            } else if self.data[lc] < self.data[rc] {
```

```
125                 lc
126             } else {
127                 rc
128             }
129         }
130
131         fn build_new(&mut self, arr: &[i32]) {
132             // 删除原始数据
133             for _i in 0..self.size {
134                 let _rm = self.data.pop();
135             }
136
137             // 添加新数据
138             for &val in arr {
139                 self.data.push(val);
140             }
141             self.size = arr.len();
142
143             // 调整堆, 使其为小顶堆
144             let sz = self.size;
145             let mut p = parent!(sz);
146             while p > 0 {
147                 self.move_down(p);
148                 p -= 1;
149             }
150         }
151
152         // 切片数据逐个加入堆
153         fn build_add(&mut self, arr: &[i32]) {
154             for &val in arr {
155                 self.push(val);
156             }
157         }
158     }
159
160     fn main() {
161         let mut bh = BinaryHeap::new();
162         let nums = [-1,0,2,3,4];
```

```
163         bh.push(10); bh.push(9);
164         bh.push(8); bh.push(7); bh.push(6);
165
166         bh.build_add(&nums);
167         println!("empty: {:?}", bh.is_empty());
168         println!("min: {:?}", bh.min());
169         println!("pop min: {:?}", bh.pop());
170
171         bh.build_new(&nums);
172         println!("size: {:?}", bh.len());
173         println!("pop min: {:?}", bh.pop());
174     }
```

7.3.3 二叉堆分析

二叉堆虽然是放到 Vec 里面，线性放置的，但其排序是按照树的方式来操作的。前面学习树时就分析过，树高度是 $O(nlog_2(n))$，而堆排序就是从树底层移到顶层，移动步骤为树的层数，所以排序复杂度应该是 $O(nlog_2(n))$。构建堆需要处理所有 n 项数据，所以复杂度是 O(n)。

7.4 二叉查找树

二叉堆用线性数据结构模拟树操作，但这只适合少量数据。一旦数据多了，数据复制移动非常耗时，所以本节来研究用节点实现树。我们定义的树只有两个子节点，这种树被称为二叉树。二叉树是最基本的树，许多树都是从二叉树衍生的。为了使用和分析二叉树，本节来研究一种用于查找的二叉树：二叉查找树。

我们通过学习树在查找任务上的性能来熟悉这种数据结构。前面学习 HashMap 时，是用键来获取值，二叉查找树类似 HashMap，也是用键值来存值。

7.4.1 二叉查找树操作

二叉查找树用于查找，所以可以定义二叉查找树如下的抽象数据类型。

new() 创建一棵新树，不需要参数，返回一个空树。

insert(k, v) 将数据 (k, v) 存储到树，需要 k 键，v 值，不返回任何内容。

search(&k) 在树中查找是否包含键 k，需要参数 &k，返回布尔值。

get(&k) 从树返回键 k 的值 v，但不会删除它，需要参数 &k。

max() 返回树中最大键 k 及其值 v，不需要参数。

min() 返回树中最小键 k 及其值 v，不需要参数。

len() 返回树中数据量，不需要参数，返回一个 usize 型整数。

is_empty() 测试树是否为空，不需要参数，返回布尔值。
iter() 返回树的迭代形式，不需要参数，不改变树。
preorder() 前序遍历，不需要参数，输出各个 k-v 值。
inorder() 中序遍历，不需要参数，输出各个 k-v 值。
postorder() 后序遍历，不需要参数，输出各个 k-v 值。
假设 t 是空二叉查找树，下表展示了各种操作后的结果，此处用元组来表示树。

表 7.3: 二叉查找树操作

二叉树操作	二叉树当前值	操作返回值
t.is_empty()	[]	true
t.insert(1,'a')	[(1,'a')]	
t.insert(2,'b')	[(1,'a'),(2,'b')]	
t.len()	[(1,'a'),(2,'b')]	2
t.get(&4)	[(1,'a'),(2,'b')]	None
t.get(&2)	[(1,'a'),(2,'b')]	Some('b')
t.min()	[(1,'a'),(2,'b')]	(Some(1), Some('a'))
t.max()	[(1,'a'),(2,'b')]	(Some(2), Some('b'))
t.search(2)	[(1,'a'),(2,'b')]	true
t.insert(2,'c')	[(1,'a'),(2,'c')]	

7.4.2 Rust 实现二叉查找树

不同于堆的左右子节点不考虑大小关系，二叉查找树左子节点键要小于父节点的键，右子节点的键要大于父节点键。也就是 left < parent < right 这个规律，其递归地适用于所有子树。

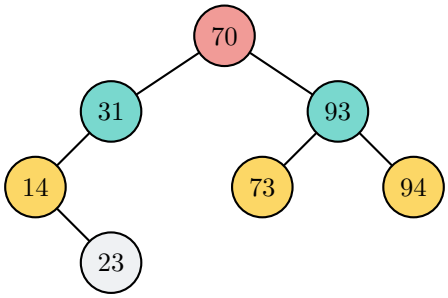


图 7.3: 二叉查找树

上图中，70 作为根节点，31 比 70 小，作为左节点，93 更大，作为右节点。接着插入 14，比 70 小，下降到 31 ，比 31 还小，则作为 31 的左节点，有其他数据插入同理，最终形成了二叉查找树。该树的中序遍历是 [14,23,31,70,73,93,94]，正是从小到大排序的数据，所以二

叉查找树也可以用来排序数据，只要使用中序遍历就能得到升序排序，使用中序遍历的镜像遍历法也就是按照“右根左”的顺序遍历就得到降序排序结果。为实现树，我们定义树为一个结构体 BST（BinarySearchTree)。按照抽象数据类型的定义，下面是实现的二叉查找树。

```
1  // bst.rs
2  use std::cmp::Ordering;
3  use std::ops::Deref;
4
5  // 二叉查找树子节点链接
6  type Link<T, U> = Option<Box<BST<T,U>>>;
7
8  // 二叉查找树定义
9  struct BST<T,U> {
10     key: Option<T>,
11     val: Option<U>,
12     left: Link<T,U>,
13     right: Link<T,U>,
14 }
15
16 impl<T,U> BST<T,U>
17 where T: Clone + Ord + std::fmt::Debug,
18       U: Clone + std::fmt::Debug {
19     fn new() -> Self {
20         BST {
21             key: None, val: None, left: None, right: None,
22         }
23     }
24
25     fn is_empty(&self) -> bool {
26         self.key.is_none()
27     }
28
29     fn len(&self) -> usize {
30         self.calc_len(0)
31     }
32
33     // 递归计算节点个数
34     fn calc_len(&self, mut i: usize) -> usize {
35         if self.key.is_none() {
```

```
36              return i;
37          }
38
39          // 当前节点加入总节点数 i
40          i += 1;
41
42          // 计算左右子节点数
43          if !self.left.is_none() {
44              i = self.left.as_ref().unwrap().calc_len(i);
45          }
46          if !self.right.is_none() {
47              i = self.right.as_ref().unwrap().calc_len(i);
48          }
49
50          i
51      }
52
53      // 前中后序遍历
54      fn preorder(&self) {
55          println!("key:{:#?},val:{:#?}",&self.key,&self.val);
56          match &self.left {
57              Some(node) => node.preorder(),
58              None => (),
59          }
60          match &self.right {
61              Some(node) => node.preorder(),
62              None => (),
63          }
64      }
65
66      fn inorder(&self) {
67          match &self.left {
68              Some(node) => node.inorder(),
69              None => (),
70          }
71          println!("key:{:#?},val:{:#?}",&self.key,&self.val);
72          match &self.right {
73              Some(node) => node.inorder(),
```

```
74                 None => (),
75             }
76         }
77
78         fn postorder(&self) {
79             match &self.left {
80                 Some(node) => node.postorder(),
81                 None => (),
82             }
83             match &self.right {
84                 Some(node) => node.postorder(),
85                 None => (),
86             }
87             println!("key:{:#?},val:{:#?}",&self.key,&self.val);
88         }
89
90         fn insert(&mut self, key: T, val: U) {
91             // 没数据直接插入
92             if self.key.is_none() {
93                 self.key = Some(key);
94                 self.val = Some(val);
95             } else {
96                 match &self.key {
97                     Some(k) => {
98                         // 存在 key, 更新 val
99                         if key == *k {
100                             self.val = Some(val);
101                             return;
102                         }
103
104                         // 未找到 key, 需要插入新节点
105                         // 先找到需要插入的子树
106                         let child = if key < *k {
107                             &mut self.left
108                         } else {
109                             &mut self.right
110                         };
111
```

```
112                         // 根据节点递归下去，直到插入
113                         match child {
114                             Some(ref mut node) => {
115                                 node.insert(key, val);
116                             },
117                             None => {
118                                 let mut node = BST::new();
119                                 node.insert(key, val);
120                                 *child = Some(Box::new(node));
121                             },
122                         }
123                     },
124                     None => (),
125                 }
126             }
127         }
128
129         fn search(&self, key: &T) -> bool {
130             match &self.key {
131                 Some(k) => {
132                     // 比较 key 值，并判断是否继续递归查找
133                     match k.cmp(&key) {
134                         Ordering::Equal => { true }, // 找到数据
135                         Ordering::Greater => { // 在左子树查找
136                             match &self.left {
137                                 Some(node) => node.search(key),
138                                 None => false,
139                             }
140                         },
141                         Ordering::Less => { // 在右子树查找
142                             match &self.right {
143                                 Some(node) => node.search(key),
144                                 None => false,
145                             }
146                         },
147                     }
148                 },
149                 None => false,
```

```
150             }
151         }
152
153         fn min(&self) -> (Option<&T>, Option<&U>) {
154             // 最小值一定在最左侧
155             match &self.left {
156                 Some(node) => node.min(),
157                 None => match &self.key {
158                     Some(key) => (Some(&key), self.val.as_ref()),
159                     None => (None, None),
160                 },
161             }
162         }
163
164         fn max(&self) -> (Option<&T>, Option<&U>) {
165             // 最大值一定在最右侧
166             match &self.right {
167                 Some(node) => node.max(),
168                 None => match &self.key {
169                     Some(key) => (Some(&key), self.val.as_ref()),
170                     None => (None, None),
171                 },
172             }
173         }
174
175         // 获取值，和查找流程相似
176         fn get(&self, key: &T) -> Option<&U> {
177             match &self.key {
178                 None => None,
179                 Some(k) => {
180                     match k.cmp(&key) {
181                         Ordering::Equal => self.val.as_ref(),
182                         Ordering::Greater => {
183                             match &self.left {
184                                 None => None,
185                                 Some(node) => node.get(key),
186                             }
187                         },
```

```
188                         Ordering::Less => {
189                             match &self.right {
190                                 None => None,
191                                 Some(node) => node.get(key),
192                             }
193                         },
194                     }
195                 },
196             }
197         }
198 }
199
200 fn main() {
201     let mut bst = BST::<i32,char>::new();
202     bst.insert(8,'e'); bst.insert(6,'c'); bst.insert(7,'d');
203     bst.insert(5,'b'); bst.insert(10,'g'); bst.insert(9,'f');
204     bst.insert(11,'h'); bst.insert(4,'a');
205
206     println!("empty: {:?}, len: {:?}", bst.is_empty(), bst.len());
207     println!("max: {:?}, min: {:?}", bst.max(), bst.min());
208     println!("key: 5, val: {:?}", bst.get(&5));
209     println!("5 in bst: {:?}", bst.search(&5));
210
211     println!("inorder: "); bst.inorder();
212     println!("preorder: "); bst.preorder();
213     println!("postorder: "); bst.postorder();
214 }
```

有了树就可以考虑插入问题了，通过使用 insert 函数向树中插入数据 76。如下图。

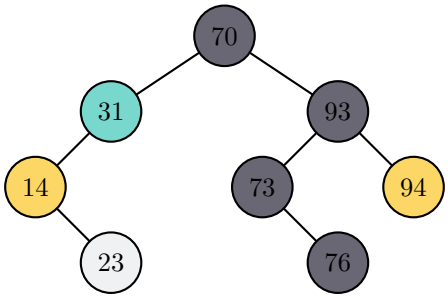


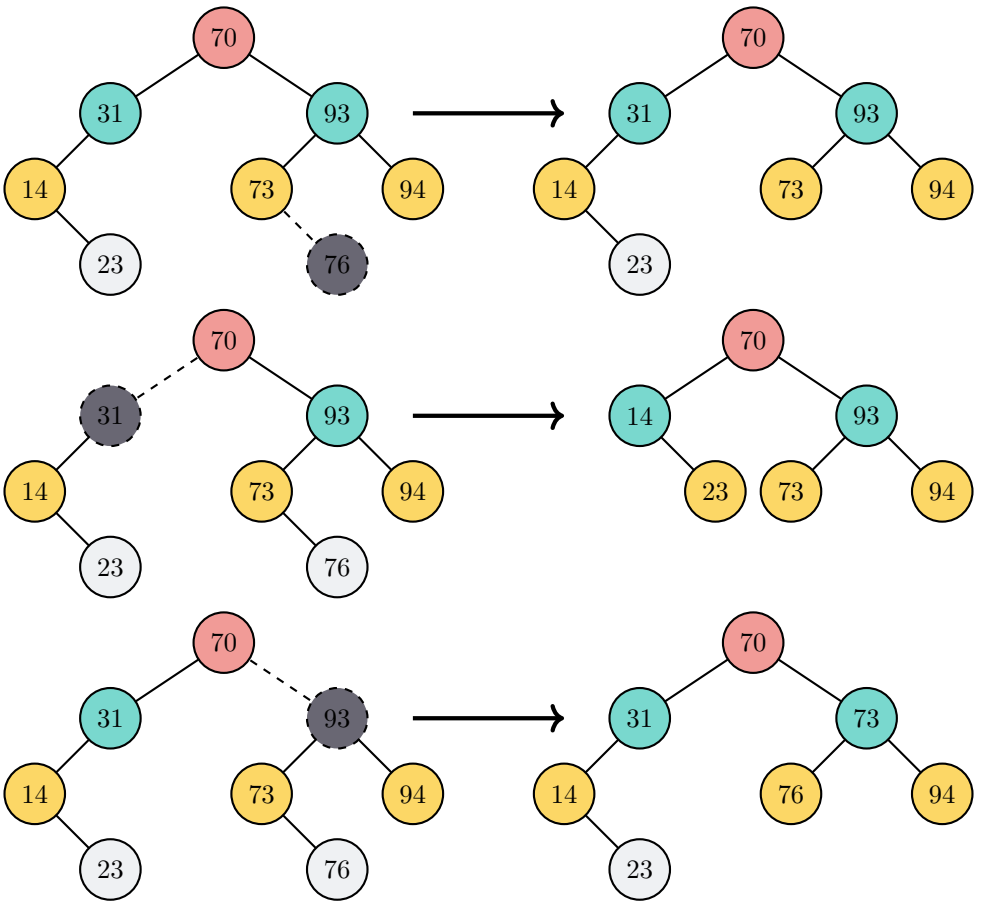
图 7.4: 二叉查找树插入数据

最后来看最复杂的操作：删除节点。删除一个键，首先要找到它。此时树可能存在三种情况：树没有节点、树有一根节点、树有若干节点。针对后两种情况，都需要检查要删除的节点是否存在。若该节点存在，则此时需要考虑该节点是否有子节点，又有三种情况：该节点是叶节点，该节点有一子节点，该节点有两个子节点。

表 7.4: k 节点删除状况

序号	树节点状况	k 子节点	删除方法	序号	树节点状况	k 子节点	删除方法
1	无节点	无	直接返回	4	有多个节点	有一个子节点	用子节点替换 k
2	有一根节点	无	直接删除	5	有多个节点	有两个子节点	用后继节点替换 k
3	有多个节点	无	直接删除				

在找到节点 k 的情况下，若它是叶节点，无子节点，则直接删除父节点对其的引用。若它有一个子节点，则修改父节点的引用直接指向子节点。若它有两个子节点，则找到右子树中的最小节点，该节点又叫后继节点，它是右子树中最小值，用该节点直接替换 k 就相当于删除了 k。当然该后继节点本身可能有零个或一个子节点，替换时也需要处理后继节点的父子引用关系。具体的情况如下面的图所示，虚线框为待删除节点 k，右侧为删除后的二叉树。



7.4.3 二叉查找树分析

我们终于完成了二叉查找树，现在可以来看看各个函数的时间复杂度了。对于三种遍历，因为要处理所有 n 个数据，所以复杂度一定是 O(n)。len() 也利用了前序遍历的方法来计算元素个数，所以也为 O(n)。

search 函数查找数据，因为它会不断和左右子节点比较，并根据比较结果选择一条分支，那么它最多走树中从根到叶节点的最长路径。根据二叉树的性质，我们知道，树的节点总数数为

$$2^0 + 2^1 + 2^i ... + 2^h = n \tag{7.1}$$

可得最大路径长度为 $log_2(n)$ 左右，所以 search 的性能为 $O(log_2(n))$。增删查改的基础是查，因为你需要定位元素位置才能继续处理。增删改都能在常量时间内完成，结果其时间复杂度只和查找有关。综上 search，insert，remove, get 的性能都是 $O(log_2(n))$，限制这些方法性能的因素是二叉树的高度 h。当然，如果插入的数据一直处于有序状态，那么树会退化成线性链表，此时 search，insert, remove 的性能均为 O(n)，如下图所示。

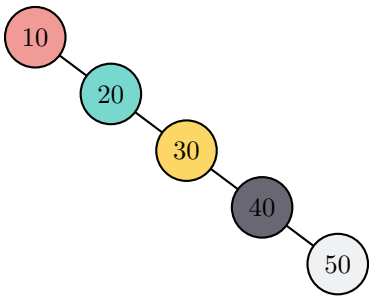


图 7.5: 二叉查找遭遇线性状况

二叉树具有 $O(log_2(n))$ 的时间复杂度，这是非常优秀的。因为树高 h 和 n 有关系。只要通过改变子节点数量，让二叉树衍生为多叉树，既每个节点保存多个子节点，比如 1000 个。这样能大幅度降低树高度，性能会更好。比较常见的多叉树是 B 树，B+ 树，它们的子节点都比较多，树很矮，查询非常快，广泛用于实现数据库和文件系统。

比如 MySQL 数据库底层就是用的 B+ 树来保存数据，它的节点是一个 16K 大的内存页。如果一条数据为 1k 大小，那么一个节点能存 16 条数据。如果用于中间层存储索引，键使用 bigint，8 字节，索引 6 字节，共 14 字节，则 16k 能存放 16 * 1024 / 14 = 1170 个索引。对于高度为 3 的 B+ 树，能存放的索引有 1170 * 1170 * 16 = 21902400 条，也就意味着能存储大概二千万条数据。而每次获取数据最多需要两次查询，因为高度为 3，几乎就是常量时间，这也是数据库查询速度快的原因。对数据库感兴趣的读者可以去阅读 MySQL 相关书籍了解更多内容。

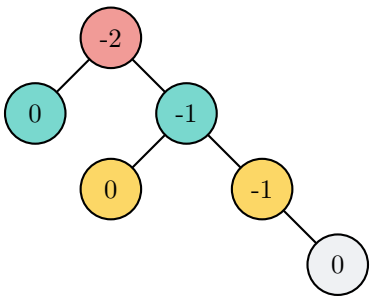
7.5 平衡二叉树

在前面一节中，我们学习并构建了一个二叉查找树，其性能在某些特殊情况可能降级到$O(n)$。比如树不平衡，一侧有非常多节点，而另一侧几乎没有，则其性能会退化，导致后续操作都非常低效。所以，构建一个平衡的二叉树是高效处理数据的前提。在本节中，我们将讨论一种平衡的二叉查找树，它能自动保持平衡状态。这种平衡的二叉查找树称为 AVL 树，名字来源于其发明人：G.M. Adelson-Velskii 和 E.M.Land。

AVL 树的是普通二叉查找树，唯一区别是树的操作执行方式。为实现 AVL 树，操作过程中需要跟踪树中节点的平衡因子。平衡因子是节点左右子树高度差，其定义为：

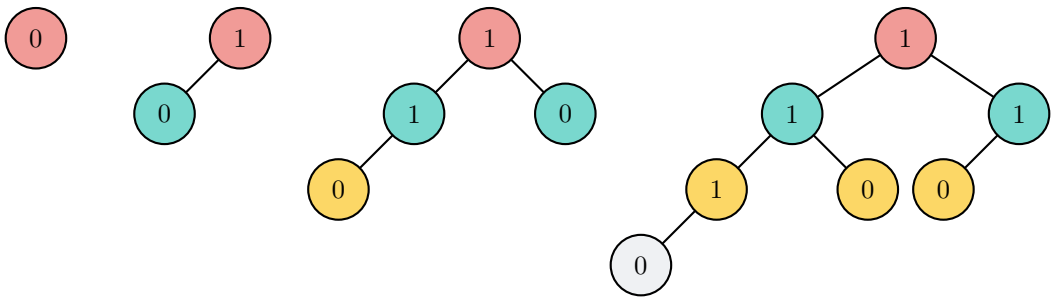
$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$

使用这个平衡因子定义，如果平衡因子大于零，则左子树重，如果平衡因子小于零，则右子树重；如果平衡因子是零，那么树是平衡的。为了实现高效的 AVL 树，可以将平衡因子 -1，0，1 三种情况均视为平衡，因为平衡因子为 1，-1 时，左右子树高度差只有 1，基本平衡。一旦树中节点平衡因子处于这个范围之外，比如 2，-2，则需要将树旋转使之维持平衡。下图展示了左右子树不平衡的情况，每个节点上标示的值就是该节点的平衡因子。



7.5.1 AVL 平衡二叉树

要得到平衡的树，则需要满足平衡因子要求。树只有三种情况，左重，平衡，右重。只要树满足左重或右重的情况下依然满足平衡因子为-1，0，1 的要求，则这样的左或右重树还是平衡的。考虑高度为 0，1，2，3 的树，下图是满足平衡因子条件的最不平衡左重树。



分析树中节点总数，可以发现对于高度为 0 的树，只有 1 个节点；对于高度为 1 的树，有 $1+1=2$ 个节点；对于高度为 2 的树则有 $1+1+2=4$ 个节点；对于高度为 3 的树，有 $1+2+4=7$ 个节点。综上，更一般地，可以看到高度为 h 的树中节点数量满足如下式：

$$N_h = 1 + N_{h-1} + N_{h-2}$$

这个公式看起来非常类似于斐波那契数列。给定树中节点数量，可以利用斐波那契公式来导出 AVL 树的高度公式。对于斐波那契数列，第 i 个斐波那契数由下式给出：

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \end{aligned} \tag{7.2}$$

这样，可以将 AVL 树高度和节点公式转化为下式：

$$N_h = F_{h+2} - 1$$

随着 i 增大，F_i/F_{i-1} 趋近于黄金比率 $\Phi = (1+\sqrt{5})/2$，所以可以使用 Φ 来表示 F_i，通过计算可得 $F_i = \frac{\Phi^i}{5}$，则：

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

通过取 2 为底的对数，可以求解 h。

$$\begin{aligned} log(N_h+1) &= log(\frac{\Phi^{h+2}}{\sqrt{5}}) \\ log(N_h+1) &= (h+2)log\Phi - \frac{1}{2}log5 \\ h &= \frac{log(N_h+1) - 2log\Phi + \frac{1}{2}log5}{log\Phi} \\ h &= 1.44log(N_h) \end{aligned} \tag{7.3}$$

所以，AVL 树的高度最高等于树中节点数目的对数值的 1.44 倍，忽略系数，则查找时的复杂度限制为 $O(logN)$，这还是非常高效的。

7.5.2 Rust 实现平衡二叉树

现在来看看如何插入新节点到 AVL 树。由于所有新节点将作为叶节点插入到树中，并且叶的平衡因子为 0，所以刚插入的新节点不用处理。但添加了新叶后，其父节点的平衡因子会改变，所以需要更新父节点的平衡因子。新插入的叶节点如何影响父节点的平衡因子取决于叶节点是左子节点还是右子节点。如果新节点是右子节点，则父节点的平衡因子将减 1；如果新节点是左子节点，则父节点的平衡因子将加 1。

这个关系可以应用到新节点的祖父节点，直到树根，这是一个递归过程。有两种情况不会更新平衡因子：（1）递归调用已到达树根。（2）父节点的平衡因子已调整为零，其祖先节点的平衡因子不会改变。

为了简洁，我们将 AVL 树实现为枚举体，其中 Null 表示空树，Tree 表示存在对树节点的引用。树节点 AvlNode 用于存放数据和左右子树及平衡因子。

```
// avl.rs

// Avl 树定义
#[derive(Debug)]
enum AvlTree<T> {
    Null,
    Tree(Box<AvlNode<T>>),
}

// Avl 树节点定义
#[derive(Debug)]
struct AvlNode<T> {
    val: T,
    left: AvlTree<T>,
    right: AvlTree<T>,
    bfactor: i8,
}
```

首先需要为 AVL 树添加插入节点的功能 insert，然而插入节点后又需要处理平衡因子，所以还需要添加一个再平衡函数 rebalance 用于更新平衡因子。为实现节点数据的比较，数据需要满足排序 Ord 特性，所以引入了 Ording 做比较。为更新值和计算树高还需要 replace 和 max 函数。

```
// avl.rs

use std::cmp::Ordering::*;
use std::cmp::max;
use std::mem::replace;
use AvlTree::*;

impl<T> AvlTree<T> where T : Ord {
    // 新树是空的
    fn new() -> AvlTree<T> {
        Null
```

```
12      }
13
14      fn insert(&mut self, val: T) -> (bool, bool) {
15          let ret = match *self {
16              // 没有节点，直接插入
17              Null => {
18                  let node = AvlNode {
19                      val: val,
20                      left: Null,
21                      right: Null,
22                      bfactor: 0,
23                  };
24                  *self = Tree(Box::new(node));
25
26                  (true, true)
27              },
28              Tree(ref mut node) => match node.val.cmp(&val) {
29                  // 比较节点值，再判断该从哪边插入
30                  // inserted 表示是否插入
31                  // deepened 表示是否加深
32                  Equal => (false, false), // 相等，无需插入
33                  Less => { // 比节点数据大，插入右边
34                      let (inserted, deepened) = node.right
35                                                     .insert(val);
36                      if deepened {
37                          let ret = match node.bfactor {
38                              -1 => (inserted, false),
39                               0 => (inserted, true),
40                               1 => (inserted, false),
41                               _ => unreachable!(),
42                          };
43                          node.bfactor += 1;
44
45                          ret
46                      } else {
47                          (inserted, deepened)
48                      }
49                  },
```

```
50                  Greater => { // 比节点数据小，插入左边
51                      let (inserted, deepened) = node.left
52                                                      .insert(val);
53                      if deepened {
54                          let ret = match node.bfactor {
55                              -1 => (inserted, false),
56                               0 => (inserted, true),
57                               1 => (inserted, false),
58                               _ => unreachable!(),
59                          };
60                          node.bfactor -= 1;
61
62                          ret
63                      } else {
64                          (inserted, deepened)
65                      }
66                  },
67              },
68          };
69          self.rebalance();
70
71          ret
72      }
73
74      // 调整各节点的平衡因子
75      fn rebalance(&mut self) {
76          match *self {
77              // 没数据，不用调整
78              Null => (),
79              Tree(_) => match self.node().bfactor {
80                  // 右子树重
81                  -2 => {
82                      let lbf = self.node().left.node().bfactor;
83                      if lbf == -1 || lbf == 0 {
84                          let (a, b) = if lbf == -1 {
85                              (0, 0)
86                          } else {
87                              (-1,1)
```

```
88                          };
89                          self.rotate_right(); // 不平衡，旋转
90                          self.node().right.node().bfactor = a;
91                          self.node().bfactor = b;
92                      } else if lbf == 1 {
93                          let (a, b) = match self.node()
94                                                  .left.node()
95                                                  .right.node()
96                                                  .bfactor {
97                              -1 => (1,0),
98                               0 => (0,0),
99                               1 => (0,-1),
100                              _ => unreachable!(),
101                         };
102
103                         // 先左旋再右旋
104                         self.node().left.rotate_left();
105                         self.rotate_right();
106                         self.node().right.node().bfactor = a;
107                         self.node().left.node().bfactor = b;
108                         self.node().bfactor = 0;
109                     } else {
110                         unreachable!()
111                     }
112                 },
113                 // 左子树重
114                 2 => {
115                     let rbf=self.node().right.node().bfactor;
116                     if rbf == 1 || rbf == 0 {
117                         let (a,b) = if rbf == 1 {
118                             (0,0)
119                         } else {
120                             (1,-1)
121                         };
122                         self.rotate_left();
123                         self.node().left.node().bfactor = a;
124                         self.node().bfactor = b;
125                     } else if rbf == -1 {
```

```
126                             let (a, b) = match self.node()
127                                                 .right.node()
128                                                 .left.node()
129                                                 .bfactor {
130                                  1 => (-1,0),
131                                  0 => (0,0),
132                                 -1 => (0,1),
133                                  _ => unreachable!(),
134                             };
135
136                             // 先右旋再左旋
137                             self.node().right.rotate_right();
138                             self.rotate_left();
139                             self.node().left.node().bfactor = a;
140                             self.node().right.node().bfactor = b;
141                             self.node().bfactor = 0;
142                         } else {
143                             unreachable!()
144                         }
145                     },
146                     _ => (),
147                 },
148             }
149         }
150     }
```

rebalance 函数完成了再平衡工作，insert 完成了平衡因子更新，这个过程是递归进行的。有效的再平衡是使 AVL 树在不牺牲性能的情况下正常工作的关键。为使 AVL 树恢复平衡，需要将树执行一次或多次旋转，可能是左旋转也可能是右旋转。

要执行左旋转，要执行的操作如下：

(1) 提升右孩子 (B) 为子树的根。

(2) 将旧根 (A) 移动为新根的左子节点。

(3) 如果新根 (B) 已经有一个左孩子，那么使它成为新左孩子 (A) 的右孩子。

要执行右旋转，要执行的操作如下：

(1) 提升左孩子 (B) 为子树的根。

(2) 将旧根 (A) 移动为新根的右子节点。

(3) 如果新根 (B) 已经有一个右孩子，那么使它成为新右孩子 (A) 的左孩子。

下图中的两棵树都不平衡，通过以 A 为根左右旋转得到了再平衡的右图。

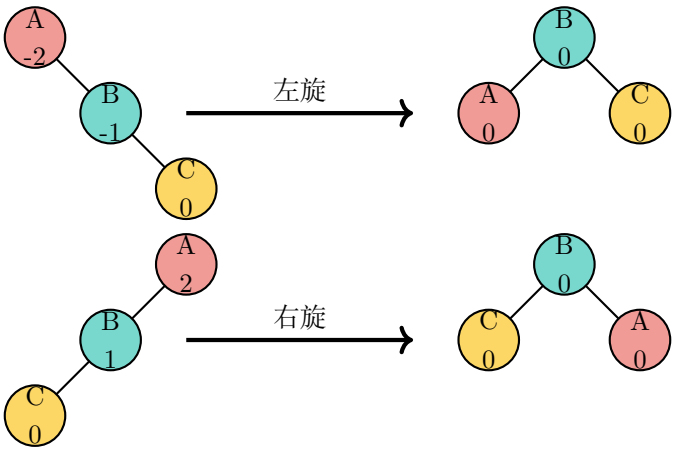
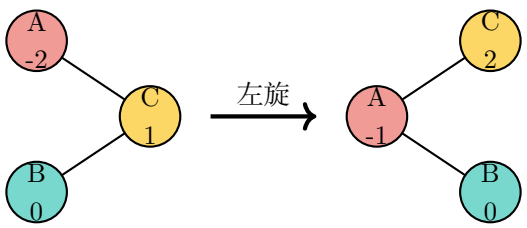


图 7.6: 不平衡树及旋转

知道了如何对子树进行左右旋转的规则，来试试将其运用到如下的这棵比较特殊的子树并进行旋转。



在左旋后，发现在另一方向还是失去了平衡。如果右旋纠正这种情况，则又回到最开始的情况。要纠正这个问题，必须使用新的旋转规则，具体如下：

（1）如子树需左旋使其平衡则首先检查右子节点平衡因子，若右孩子是重的，则对右孩子做右旋转，然后是再执行左旋转。

（2）如子树需右旋使其平衡则首先检查左子节点平衡因子，若左孩子是重的，则对左孩子做左旋转，然后是再执行右旋转。

可见子树旋转过程在概念上相当容易理解，但是代码实现非常复杂，因为首先需要按照正确的顺序移动节点，以便保留二叉查找树的所有属性。此外，还需要确保适当地更新所有的父指针。而 Rust 中移动和所有权机制的存在使得移动节点和更新指针关系颇为复杂，十分容易出错。了解了旋转的概念和工作原理后可以看看下面实现的旋转函数，包括右旋转和左旋转的代码。其中 node 和 left、right 函数用于获取节点和子树。

```
1 // avl.rs
2
3 impl<T> AvlTree<T> where T : Ord {
4     // 获取节点
```

```
5       fn node(&mut self) -> &mut AvlNode<T> {
6           match *self {
7               Null => panic!("Empty tree"),
8               Tree(ref mut n) => n,
9           }
10      }
11
12      // 获取左右子树
13      fn left_subtree(&mut self) -> &mut Self {
14          match *self {
15              Null => panic!("Empty tree"),
16              Tree(ref mut node) => &mut node.left,
17          }
18      }
19
20      fn right_subtree(&mut self) -> &mut Self {
21          match *self {
22              Null => panic!("Empty tree"),
23              Tree(ref mut node) => &mut node.right,
24          }
25      }
26
27      // 左右旋
28      fn rotate_left(&mut self) {
29          let mut v = replace(self, Null);
30          let mut right = replace(v.right_subtree(), Null);
31          let right_left = replace(right.left_subtree(), Null);
32          *v.right_subtree() = right_left;
33          *right.left_subtree() = v;
34          *self = right;
35      }
36
37      fn rotate_right(&mut self) {
38          let mut v = replace(self, Null);
39          let mut left = replace(v.left_subtree(), Null);
40          let left_right = replace(left.right_subtree(), Null);
41          *v.left() = left_right;
42          *left.right_subtree() = v;
```

```
43              *self = left;
44          }
45      }
```

通过旋转操作，我们始终维持着树的平衡。为获取树的节点数、节点值、树高，我们还需要实现 len、depth、value 等函数。

```
1   // avl.rs
2
3   impl<T> AvlTree<T> where T : Ord {
4       // 树节点数是左右子树节点数加根节点数，递归计算
5       fn len(&self) -> usize {
6           match *self {
7               Null => 0,
8               Tree(ref v) => 1 + v.left.len() + v.right.len(),
9           }
10      }
11
12      // 树深度是左右子树深度最大值 + 1，递归计算
13      fn depth(&self) -> usize {
14          match *self {
15              Null => 0,
16              Tree(ref v) => max(v.left.depth(),
17                                 v.right.depth()) + 1,
18          }
19      }
20
21      fn is_empty(&self) -> bool {
22          match *self {
23              Null => true,
24              _ => false,
25          }
26      }
27
28      // 数据查找
29      fn search(&self, val: &T) -> bool {
30          match *self {
31              Null => false,
32              Tree(ref v) => {
```

```
33                     match v.val.cmp(&val) {
34                         Equal => { true },
35                         Greater => {
36                             match &v.left {
37                                 Null => false,
38                                 _ => v.left.search(val),
39                             }
40                         },
41                         Less => {
42                             match &v.right {
43                                 Null => false,
44                                 _ => v.right.search(val),
45                             }
46                         },
47                     }
48                 },
49             }
50         }
51     }
52
53     fn main() {
54         let mut avl = AvlTree::new();
55         for i in 0..10 {
56             let (_r1, _r2) = avl.insert(i);
57         }
58         println!("empty: {}", avl.is_empty());
59         println!("length: {}", avl.len());
60         println!("depth: {}", avl.depth());
61         println!("9 in avl: {}", avl.search(&9));
62     }
```

7.5.3 平衡二叉树分析

AVL 平衡二叉树相比二叉树添加了左旋转和右旋转操作，这些旋转操作用于维持二叉树自身的平衡，这样它的其他各种操作性能就能维持在比较优秀的水平，使得其最差性能都是 $O(log_2(n))$。

7.6 总结

在本章中，我们学习了树这种高效的数据结构。树使得我们能编写许多有用和高效的算法，它广泛用于存储，网络等领域。本章我们用树执行了以下操作：

（1）解析和计算表达式。

（2）实现二叉树，二叉查找树，二叉平衡树、红黑树

（3）实现堆及优先队列。

在前面几章中，我们已经学习了可用于实现映射关系（Map）的几种抽象数据类型，包括有序表、散列表、二叉查找树和平衡二叉查找树，下面是它们的各种操作的最差性能。

表 7.5: 各种抽象数据结构的操作性能

操作	有序列表	哈希表	二叉查找树	平衡二叉树	红黑树
insert	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
search	$O(log(n))$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$
delete	$O(n)$	$O(1)$	$O(n)$	$O(log(n))$	$O(log(n))$

Chapter 8

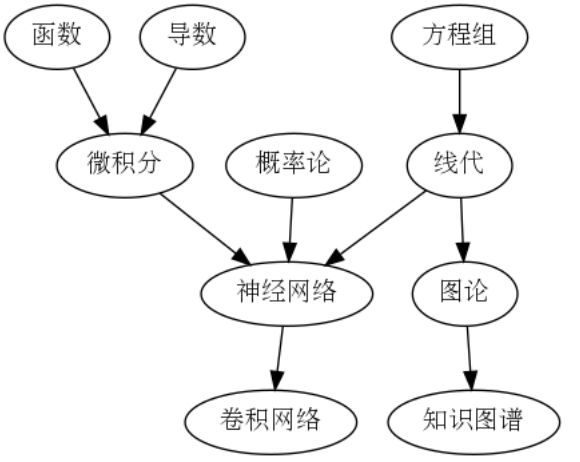
图

8.1 本章目标

了解图的概念及其使用方法
用 Rust 来实现图数据结构
使用图来解决各类现实问题

8.2 什么是图

上一章学习了树，尤其是二叉树。这一章来学习一种树的更普遍的形式：图。树可以看成是简化的图，或者精心挑选的图，因为它的节点关系是有规律的。树有根节点，但图没有，树有方向，从上到下，图的方向可有可无，树中无环，但图可以有环。图涉及点、边及点边关系，可以用来表示真实世界中存在的事物，包括航班图、网络连接、菜谱、课程规划等。下面就是一幅图，它包含多个节点，存在多个连接。



8.2.1 图定义

因为图 (graph) 是树更普遍的形式，所以图的定义是树定义的延伸。

顶点：也就是树中所说的节点，是图的元素，它有一个名称：键。一个顶点也可能有额外的信息：有效载荷。

边：图的另一个元素。边连接两个顶点，表明点之间的关系。边可以是单向的或双向的。如图中边都是单向的，称该图是有向图。

权重：是边的度量。用一个数值来表示从一个顶点到另一个顶点的距离、成本、时间、亲密度。

利用这些概念，可以定义图。图用 G 来表示，G = (V, E)。图中核心元素是点集合 V 和边集合 E。每两个不同点的组合 (v, w, q) 表示一条属于 E 的边 v-w，权重为 q。下图是带权重的有向图，其点集合为 V = (V0, V1, V2, V2, V4, V5)，边集合为 E = ((V0,V1,5),(V1,V2,5),(V2,V3,5),(V3,V4,5),(V4,V0,5),(V0,V5,5),(V5,V4,5),(V3,V5,5),(V5,V2,1))。

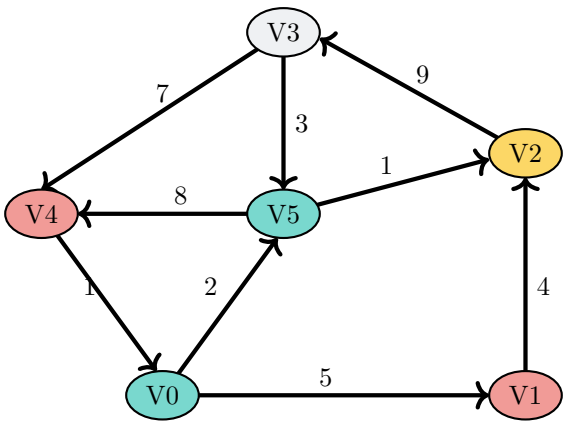


图 8.1: 图

除了定义的点、边、权重，还使用路径来表示各个点的先后顺序。路径就是用顶点序列来表示点连接的前后顺序，如 (v,w,x,y,z)。因为图中的点连接是任意的，所以可能出现有环图，例如图中 V5 -> V2 -> V3 -> V5。如果图中没有环，那么这种图称为无环图或 DAG 图。许多重要的问题都可以用 DAG 图来表示。

8.3 图的存储形式

计算机内存是线性的，图是非线性的，所以一定有要某种方法来保存图到线性的存储设备中。最常用的保存方法有两种，一种是邻接矩阵，一种是邻接表。

邻接矩阵采用二维矩阵存储图的节点和边及权重，对于 N 个点的图来说，需要 N^2 个空间。邻接表类似哈希表，通过对每个顶点开一条链来保存所有与之相关的点和边，其存储空间根据边的连接而定，一般情况下远远小于 N^2。

通过上述分析可以知道，计算机应该使用的是邻接表来存储图。

8.3.1 邻接矩阵

保存图最简单的方法就是用二维矩阵。在矩阵中，每行和每列表示图中的顶点。存储在行 v 和列 w 的交叉点处的值表示是顶点 v 到顶点 w 的边存在且权重为该值。当两个顶点通过边连接时，我们说它们是相邻的。下图就是邻接矩阵，存储的是图（8.1）的点和边。

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

图 8.2: 邻接矩阵

邻接矩阵简单、直观，对于小图，容易看出节点连接关系。观察矩阵可以发现大多数单元是空的，矩阵很稀疏。如果有 N 个顶点，那么矩阵所需的存储为 N^2，非常浪费内存。

8.3.2 邻接表

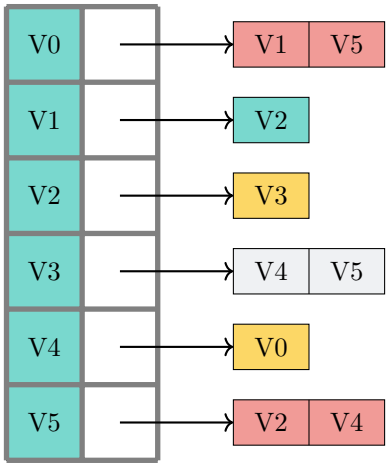


图 8.3: 邻接表

实现图高效保存的方法是使用邻接表，如上图。在邻接表中，使用数组来保存所有顶点的主列表，然后图中的每个主顶点维护一个连接到其他顶点的链表。这样，通过访问各个顶点的链接表，就能知道它链接多少点，这类似 HashMap 解决冲突时用的拉链法。

采用类似 HashMap 的结构来保存这些链接的点，主要是因为边带有权重，而数组只能保存顶点。用邻接表实现的图是紧凑的，没有内存浪费，这种结构保存也非常方便。从这里也看出了，基础数据结构的重要性，在这里使用到了前面章节实现过的 HashMap 。

8.4 图的抽象数据类型

有了图的基本定义，下面定义图的抽象数据结构。图中核心的元素是点和边，所以操作是围绕点和边来开展的。

new() 创建一个空图，不需要参数，返回空图。

add_vertex(v) 添加一个顶点，需要参数 v，无返回内容。

add_edge(fv,tv,w) 添加带权重的有向边，需要起点 fv 和终点 tv 及权重，无返回值。

get_vertex(vk) 在图中找到键为 vk 的点，需要参数 vk，发挥点。

get_vertices() 返回图中所有顶点的列表，不需要参数。

vert_nums() 返回图中顶点数，不需要参数。

edge_nums() 返回图中边数，不需要参数。

contains(vk) 判断点是否在图中，需要参数 vk，返回布尔值。

is_empty() 判断图是否为空，不需要参数，返回布尔值。

假设 g 是新创建的空图，下表展示了图各种操作后的结果。[] 装点，() 表示边，其中前两个值是点，第三个值是边权重，如 (1,5,2) 表示点 1 和点 5 间边权重为 2。

表 8.1: 图操作及其结果

图操作	图当前值	操作返回值
g.is_empty()	[]	true
g.add_vertex(1)	[1]	
g.add_vertex(5)	[1,5]	
g.add_edge(1,5,2)	[1,5,(1,5,2)]	
g.get_vertex(5)	[1,5,(1,5,2)]	5
g.get_vertex(4)	[1,5,(1,5,2)]	None
g.edge_nums()	[1,5,(1,5,2)]	1
g.vert_nums()	[1,5,(1,5,2)]	2
g.contains(1)	[1,5,(1,5,2)]	true
g.get_verteces()	[1,5,(1,5,2)]	[1,5]
g.add_vertex(7)	[1,5,7,(1,5,2)]	
g.add_vertex(9)	[1,5,7,9,(1,5,2)]	
g.add_edge(7,9,8)	[1,5,7,9,(1,5,2),(7,9,8)]	

8.5 图的实现

首先来实现基于邻接矩阵的图，然后再实现基于邻接表的图。根据图的定义和抽象数据类型，我们需要实现点（Vertex）和边（Edge），然后用矩阵来存储边关系。Rust 中二维的 Vec 可以用来构造矩阵。

```rust
// graph_matrix.rs

// 点定义
#[derive(Debug)]
struct Vertex<'a> {
    id: usize,
    name: &'a str,
}

impl Vertex<'_> {
    fn new(id: usize, name: &'static str) -> Self {
        Self { id, name }
    }
}
```

边只需要用布尔值表示是否存在即可，因为边是关系，不需要构造一个实体出来。

```rust
// graph_matrix.rs

// 边定义
#[derive(Debug, Clone)]
struct Edge {
    edge: bool, // 表示是否有边，并不需要构造一个边实体
}

impl Edge {
    fn new() -> Self {
        Self { edge: false }
    }

    fn set_edge() -> Self {
        Edge { edge: true }
    }
}
```

图（Graph）中实现边关系，并保存在二维的 Vec 中。

```rust
// graph_matrix.rs

// 图定义
#[derive(Debug)]
struct Graph {
    nodes: usize,
    graph: Vec<Vec<Edge>>, // 每个点的边放一个 vec
}

impl Graph {
    fn new(nodes: usize) -> Self {
        Self {
            nodes,
            graph: vec![vec![Edge::new(); nodes]; nodes],
        }
    }

    fn len(&self) -> usize {
        self.nodes
    }

    fn is_empty(&self) -> bool {
        0 == self.nodes
    }

    // 添加边，设置边属性为 true
    fn add_edge(&mut self, n1: &Vertex, n2: &Vertex) {
        if n1.id < self.nodes && n2.id < self.nodes {
            self.graph[n1.id][n2.id] = Edge::set_edge();
        } else {
            panic!("error");
        }
    }
}

fn main() {
    let mut g = Graph::new(4);
```

```
38      let n1 = Vertex::new(0, "n1");
39      let n2 = Vertex::new(1, "n2");
40      let n3 = Vertex::new(2, "n3");
41      let n4 = Vertex::new(3, "n4");
42
43      g.add_edge(&n1,&n2); g.add_edge(&n1,&n3);
44      g.add_edge(&n2,&n3); g.add_edge(&n2,&n4);
45      g.add_edge(&n3,&n4); g.add_edge(&n3,&n1);
46
47      println!("{:#?}", g);
48      println!("graph empth: {}", g.is_empty());
49      println!("graph nodes: {}", g.len());
50  }
```

下面再用 Rust 的 HashMap 来实现邻接表图。因为点是核心元素，边是点的关系，所以 Graph 还是将创建表示点元素的数据结构 Vertex。对 Vertex，需要的操作有新建点、获取点自身的值、添加邻接点、获取所有邻接点、获取到邻接点的权重。connects 变量用于保存所有邻接点。

```
1   // graph_adjlist.rs
2
3   use std::hash::Hash;
4   use std::collections::HashMap;
5
6   // 点定义
7   #[derive(Debug, Clone)]
8   struct Vertex<T> {
9       key: T,
10      connects: Vec<(T, i32)>, // 邻点集合
11  }
12
13  impl<T: Clone + PartialEq> Vertex<T> {
14      fn new(key: T) -> Self {
15          Self { key: key, connects: Vec::new() }
16      }
17
18      // 判断与当前点是否相邻
19      fn adjacent_key(&self, key: &T) -> bool {
20          for (nbr, _wt) in self.connects.iter() {
```

```
21              if nbr == key { return true; }
22          }
23
24          false
25      }
26
27      fn add_neighbor(&mut self, nbr: T, wt: i32) {
28          self.connects.push((nbr, wt));
29      }
30
31      // 获取相邻的点集合
32      fn get_connects(&self) -> Vec<&T> {
33          let mut connects = Vec::new();
34          for (nbr, _wt) in self.connects.iter() {
35              connects.push(nbr);
36          }
37
38          connects
39      }
40
41      // 返回到邻点的边权重
42      fn get_nbr_weight(&self, key: &T) -> &i32 {
43          for (nbr, wt) in self.connects.iter() {
44              if nbr == key {
45                  return wt;
46              }
47          }
48
49          &0
50      }
51  }
```

Graph 是实现的图数据结构，其中包含将顶点名称映射到顶点对象的 HashMap。

```
1  // graph_adjlist.rs
2
3  // 图定义
4  #[derive(Debug, Clone)]
5  struct Graph <T> {
```

```
6       vertnums: u32, // 点数
7       edgenums: u32, // 边数
8       vertices: HashMap<T, Vertex<T>>, // 点集合
9   }
10
11  impl<T: Hash + Eq + PartialEq + Clone> Graph<T> {
12      fn new() -> Self {
13          Self {
14              vertnums: 0,
15              edgenums: 0,
16              vertices: HashMap::<T, Vertex<T>>::new(),
17          }
18      }
19
20      fn is_empty(&self) -> bool {
21          0 == self.vertnums
22      }
23
24      fn vertex_num(&self) -> u32 {
25          self.vertnums
26      }
27
28      fn edge_num(&self) -> u32 {
29          self.edgenums
30      }
31
32      fn contains(&self, key: &T) -> bool {
33          for (nbr, _vertex) in self.vertices.iter() {
34              if nbr == key {
35                  return true;
36              }
37          }
38
39          false
40      }
41
42      fn add_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
43          let vertex = Vertex::new(key.clone());
```

```
44          self.vertnums += 1;
45          self.vertices.insert(key.clone(), vertex)
46      }
47
48      fn get_vertex(&self, key: &T) -> Option<&Vertex<T>> {
49          if let Some(vertex) = self.vertices.get(key) {
50              Some(&vertex)
51          } else {
52              None
53          }
54      }
55
56      // 获取所有节点的 key
57      fn vertex_keys(&self) -> Vec<T> {
58          let mut keys = Vec::new();
59          for key in self.vertices.keys() {
60              keys.push(key.clone());
61          }
62
63          keys
64      }
65
66      // 删除点 (同时要删除边)
67      fn remove_vertex(&mut self, key: &T) -> Option<Vertex<T>> {
68          let old_vertex = self.vertices.remove(key);
69          self.vertnums -= 1;
70
71          // 删除从当前点出发的边
72          self.edgenums -= old_vertex.clone()
73                                      .unwrap()
74                                      .get_connects()
75                                      .len() as u32;
76
77          // 删除到当前点的边
78          for vertex in self.vertex_keys() {
79              if let Some(vt) = self.vertices.get_mut(&vertex) {
80                  if vt.adjacent_key(key) {
81                      vt.connects.retain(|(k, _)| k != key);
```

```
                        self.edgenums -= 1;
                    }
                }
            }

            old_vertex
        }

        fn add_edge(&mut self, from: &T, to: &T, wt: i32) {
            // 若点不存在要先添加点
            if !self.contains(from) {
                let _fvert = self.add_vertex(from);
            }

            if !self.contains(to) {
                let _tvert = self.add_vertex(to);
            }

            // 添加边
            self.edgenums += 1;
            self.vertices.get_mut(from)
                        .unwrap()
                        .add_neighbor(to.clone(), wt);
        }

        // 判断两个点是否相邻
        fn adjacent(&self, from: &T, to: &T) -> bool {
            self.vertices.get(from).unwrap().adjacent_key(to)
        }
    }
```

使用 Graph 可以创建顶点 V0-V5，再据此创建边，HashMap 保存了整个图的点和边。

```
// graph_adjlist.rs

fn main() {
    let mut g = Graph::new();

    for i in 0..6 { g.add_vertex(&i); }
```

```
7       println!("graph empty: {}", g.is_empty());
8
9       let vertices = g.vertex_keys();
10      for vertex in vertices { println!("Vertex: {:#?}", vertex); }
11
12      g.add_edge(&0,&1,5); g.add_edge(&0,&5,2);
13      g.add_edge(&1,&2,4); g.add_edge(&2,&3,9);
14      g.add_edge(&3,&4,7); g.add_edge(&3,&5,3);
15      g.add_edge(&4,&0,1); g.add_edge(&4,&4,8);
16      println!("vert nums: {}", g.vertex_num());
17      println!("edge nums: {}", g.edge_num());
18      println!("contains 0: {}", g.contains(&0));
19
20      let vertex = g.get_vertex(&0).unwrap();
21      println!("key: {}, to nbr 1 weight: {}",
22               vertex.key, vertex.get_nbr_weight(&1));
23
24      let keys = vertex.get_connects();
25      for nbr in keys { println!("nighbor: {nbr}"); }
26
27      for (nbr, wt) in vertex.connects.iter() {
28          println!("0 nighbor: {nbr}, weight: {wt}");
29      }
30
31      let res = g.adjacent(&0, &1);
32      println!("0 adjacent to 1: {res}");
33      let res = g.adjacent(&3, &2);
34      println!("3 adjacent to 2: {res}");
35
36      let rm = g.remove_vertex(&0).unwrap();
37      println!("remove vertex: {}", rm.key);
38      println!("left vert nums: {}", g.vertex_num());
39      println!("left edge nums: {}", g.edge_num());
40      println!("contains 0: {}", g.contains(&0));
41  }
```

8.5.1 图解决字梯问题

有了图数据结构，就可以解决些实际问题了。一个问题是字梯难题，指将某个单词转换成另一个单词。比如将 FOOL 转换为单词 SAGE。在字梯中你通过逐个改变字母而使单词发生变化，但新的单词必须是存在的而不是任意单词。这个游戏是《爱丽丝梦游仙境》的作者刘易斯于 1878 年发明的。下面的单词序列显示了对上述问题的多种解决方案。

```
1  a      b      c      d      e      f      g
2  ————————————————————————————————————————————
3  FOOL   FOOL   FOOL   FOOL   FOOL   FOOL   FOOL
4  POOL   FOIL   FOIL   COOL   COOL   FOUL   FOUL
5  POLL   FAIL   FAIL   POOL   POOL   FOIL   FOIL
6  POLE   FALL   FALL   POLL   POLL   FALL   FALL
7  PALE   PALL   PALL   POLE   PALL   FALL   FALL
8  SALE   PALE   PALE   PALE   PALE   PALL   PALL
9  SAGE   PAGE   SALE   SALE   SALE   POLL   POLL
10        SAGE   SAGE   SAGE   SAGE   PALE   PALE
11                                    PAGE   SALE
12                                    SAGE   SAGE
```

可见转换的路径有多条。我们的目标是，通过使用图算法来找出从起始单词转换为结束单词的最小转换次数，如上图中 a 列。利用图首先将单词转换为顶点，然后将这些能前后变化得到的单词链接起来。最后使用图搜索算法来搜索最短路径。如果两个词只有一个字母不同，表明它们可以相互转换，则就创建这两个单词的有向边，结果如下图。

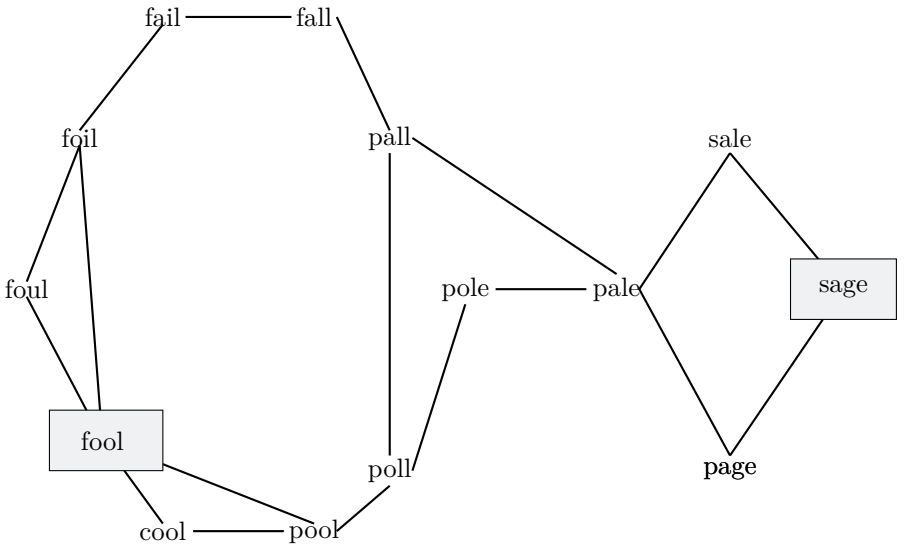


图 8.4: 字梯图

可以使用多种不同方法来创建这个问题的图模型。假设有一个长度相同的单词列表，首先可以在图中为列表中的每个单词创建一个顶点。为了弄清楚如何连接单词，必须比较列表中的每个单词。比较时看有多少字母是不同的，如果所讨论的两个单词只有一个字母不同，则可以在图中创建它们之间的边。对于小的单词列表，这种方法可以正常工作。然而如果使用大学四级单词表，假设有 3000 个单词，那么上述比较是 $O(n^2)$，需要比较九百万次。如果是六级词汇表，假设 5000 单词，则比较次数达到二千五百万。显然相互间能转换的单词不过几十个，但每次比较却要几千次，非常低效。

然而，要是换个思路，只看单词的字母位，对每个位置搜寻类似单词并放在一起，那么最终这些单词都能和该单词连接。比如 SOPE 去掉第一个字母，剩下 _OPE，那么凡是结尾是 OPE 的四个字母的单词都要和 SOPE 链接，将他们收集起来放到一个集合里面。接着是 S_PE，凡是符合这种模式的单词也收集起来。最终，符合模式的单词都收集了，如下图。

```
1 _OPE P_PE  PO_E  POP_
2 POPE POPE  POPE  POPE
3 ROPE PIPE  POLE  POPS
4 NOPE PAPE  PORE
5 HOPE       POSE
6 LOPE       POKE
7 COPE
```

可以使用 HashMap 来实现这种方案。上述集合保存了相同模式的单词，以这种模式作为 HashMap 的键，然后存储类似的单词集合。一旦建立了单词集合，就可以创建图。通过为 HashMap 中的每个单词创建一个顶点来开始图，然后与字典中相同键下找到的所有顶点间创建边。实现了图后，就可以完成字梯搜索任务。

```
1  // word_lader.rs
2
3  fn build_word_graph(words: &[String]) -> Graph {
4      let mut d: HashMap<String, Vec<String>> = HashMap::new();
5
6      for word in words {
7          for i in 0..word.len() {
8              let bucket = (word[..i].to_string() + "_")
9                          + &word[i+1..];
10
11             let wd = word.to_string();
12             if d.contains_key(&bucket) {
13                 d.get_mut(&bucket).unwra().push(wd);
14             } else {
15                 d.insert(bucket, vec![wd]);
```

```
16              }
17          }
18      }
19
20      let mut g = Graph::new();
21      for bucket in d.keys() {
22          for wd1 in &d[bucket] {
23              for wd2  in &d[bucket] {
24                  if wd1 != wd2 {
25                      g.add_edge(wd1,wd2, 1);
26                  }
27              }
28          }
29      }
30
31      g
32  }
33
34  fn world_lader(mut g: Graph<String>, mut start: Vertex<String>,
35                 end: Vertex<String>, len: usize) -> uu32 {
36      start.set_distance(0);
37      start.set_pred(None);
38
39      let mut vertex_queue = Queue::new(len);
40      let _r = vertex_queue.enqueue(start);
41
42      while vertex_queue.size() > 0 {
43          let mut currv = vertex_queue.dequeue.unwrap();
44          for nbr in currv.get_connects() {
45              let nbv = g.vertices.get_mut(nbr).unwrap();
46              if 0 == nbv.color {
47                  nbv.set_color(1);
48                  nbv.set_distance(currv.dist + 1);
49                  nbv.set_pred(Some(currv.key.clone()));
50                  let v = g.vertices.get(nbr).unwrap().clone();
51                  let _r = vertex_queue.enqueue(v);
52              }
53          }
```

```
54              currv.set_color(2);
55          }
56
57          g.vertices.get(&end.key).unwrap().dist
58      }
```

8.6 广度优先搜索

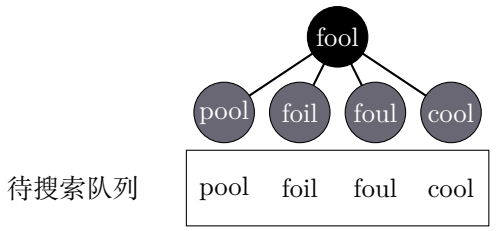
有了单词构建的图，接下来是如何查找到最短路径。不像线性数据结构的查找，图非线性，所以不能用传统的二分和线性查找算法来搜索最短路径。

图及研究图的专门理论-图论，都是非常复杂的。对图的研究也产生了各种优秀算法，比如用于搜索的算法有深度优先搜索和广度优先搜索。对于字梯问题，可以采用图的广度优先搜索来查找最短路径。注意，树是特殊的图，所以适合图的深度优先搜索和广度优先搜索也可用于在树中搜索。深度和广度优先搜索是搜索图的最简单算法，也是众多其他重要图算法的基础。

给定图 G 和起始顶点 s，广度优先搜索通过探索图中的边以找到 G 中的所有顶点，其中存在从 s 开始的路径。通过广度优先搜索，它找到和 s 相距为 1 的所有顶点，然后找到距离为 2 的所有顶点。

可以将这个过程看成是逐层搜索，先将直接联系的一层顶点放到队列里逐个搜索。可以用三种颜色来标志顶点的状态，初始时点均为白色，搜索时将与当前搜索点相连的点都置为灰色，逐个查看灰色点，一旦确定该点及连接点都不是目标值，则将当前点置为黑色，继续搜索灰色点连接的白色点，重复搜索过程。这似乎和某些语言的垃圾收集机制有些类似，比如 Go 的三色垃圾收集法。

为处理灰色点，可用一个队列来保存灰色点，然后出队灰色点并比较。为对点颜色进行设置和修改，需要对 Vertex 扩展，添加表示颜色的参数，与起始点的距离，前导节点。



8.6.1 实现广度优先搜索

BFS 从起始顶点 s 开始，将颜色设置为灰色，表明它正在被搜索。另外两个值，即距离和前导，对于起始顶点分别初始化为 0 和 None，最后将其放到一个队列中。下一步是开始系统地检查队列里的顶点，通过迭代邻接表来探索新节点。当检查邻接表上的节点时，要检查其颜色，如果是白色的，表明该顶点是未被搜索的。在搜索顶点时会出现如下四种情况:

```
顶点是新的，未被搜索，则将其着色为灰色。
顶点的前导被设置为当前接点。
到顶点的距离设置为当前距离加一。
顶点添加到队尾。
```

下面来实现广度优先搜索算法，首先定义一个图，此处只为实现广度优先算法，所以图可以简化。

```
// bfs.rs

use std::rc::Rc;
use std::cell::RefCell;

// 因为节点存在多个共享的链接，Box 不可共享，Rc 才可共享
// 又因为 Rc 不可变，所以使用具有内部可变性的 RefCell 包裹
type Link = Option<Rc<RefCell<Node>>>;

// 节点
struct Node {
    data: usize,
    next: Link,
}

impl Node {
    fn new(data: usize) -> Self {
        Self {
            data: data,
            next: None
        }
    }
}

// 图定义及实现
struct Graph {
    first: Link,
    last: Link,
}

impl Graph {
```

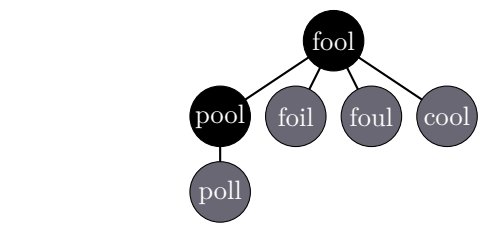
```
32      fn new() -> Self {
33          Self { first: None, last: None }
34      }
35
36      fn is_empty(&self) -> bool {
37          self.first.is_none()
38      }
39
40      fn get_first(&self) -> Link {
41          self.first.clone()
42      }
43
44      // 打印节点
45      fn print_node(&self) {
46          let mut curr = self.first.clone();
47          while let Some(val) = curr {
48              print!("[{}]", &val.borrow().data);
49              curr = val.borrow().next.clone();
50          }
51
52          print!("\n");
53      }
54
55      // 插入节点, RefCell 使用 borrow_mut 修改
56      fn insert(&mut self, data: usize) {
57          let node = Rc::new(RefCell::new(Node::new(data)));
58          if self.is_empty() {
59              self.first = Some(node.clone());
60              self.last = Some(node);
61          } else {
62              self.last.as_mut()
63                       .unwrap()
64                       .borrow_mut()
65                       .next = Some(node.clone());
66              self.last = Some(node);
67          }
68      }
69  }
```

下面是实现的广度优先搜索算法。

```
// bfs.rs

// 根据 data 构建图
fn create_graph(data: [[usize;2];20]) -> Vec<(Graph, usize)> {
    let mut arr: Vec<(Graph, usize)> = Vec::new();
    for _ in 0..9 {
        arr.push((Graph::new(), 0));
    }

    for i in 1..9 {
        for j in 0..data.len() {
            if data[j][0] == i {
                arr[i].0.insert(data[j][1]);
            }
        }
        print!("[{i}]->");
        arr[i].0.print_node();
    }

    arr
}

fn bfs(graph: Vec<(Graph, usize)>) {
    let mut gp = graph;
    let mut nodes = Vec::new();

    gp[1].1 = 1;
    let mut curr = gp[1].0.get_first().clone();

    // 打印图
    print!("{1}->");
    while let Some(val) = curr {
        nodes.push(val.borrow().data);
        curr = val.borrow().next.clone();
    }

    // 打印宽度优先图
```

```
38      loop {
39          if 0 == nodes.len() {
40              break;
41          } else {
42              let data = nodes.remove(0);
43              if 0 == gp[data].1 {
44                  gp[data].1 = 1;
45                  print!("{data}->");
46                  let mut curr = gp[data].0.get_first().clone();
47                  while let Some(val) = curr {
48                      nodes.push(val.borrow().data);
49                      curr = val.borrow().next.clone();
50                  }
51              }
52          }
53      }
54
55      println!();
56  }
57
58  fn main() {
59      let data = [[1,2],[2,1],[1,3],[3,1],[2,4],[4,2],[2,5],
60                  [5,2],[3,6],[6,3],[3,7],[7,3],[4,5],[5,4],
61                  [6,7],[7,6],[5,8],[8,5],[6,8],[8,6]];
62      let gp = create_graph(data);
63      bfs(gp);
64  }
```

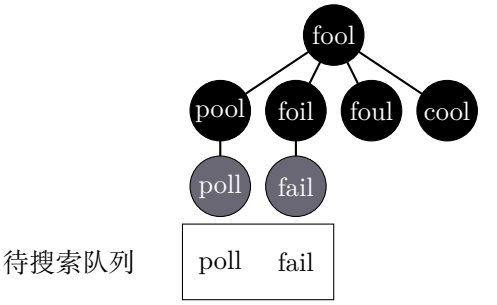
bfs 函数构造出如下图的广度优先搜索树。开始取所有与 fool 相邻的节点，并将它们添加到树中。相邻节点包括 pool，foil，foul，cool。然后这些节点会被添加到新节点的队列以进行搜索。



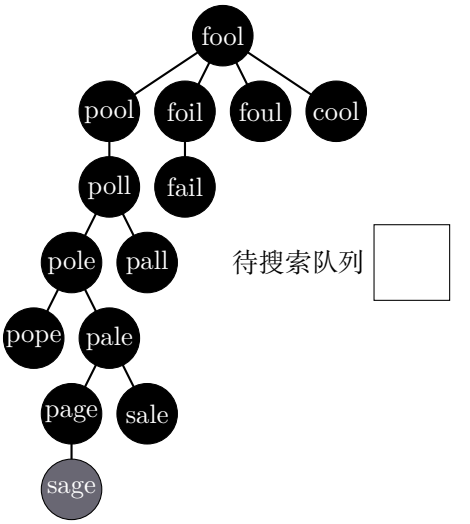
待搜索队列

foil	foul	cool	poll

在下一步骤中，bfs 从队列前面删除下一个节点 pool，并对其所有相邻节点重复该过程。当 bfs 检查节点 cool 时，发现它是灰色，说明有较短的路径到 cool。在检查 pool 时，又添加了新节点 poll。



队列上的下一个顶点是 foil，可以添加到树中的唯一新节点是 fail。当 bfs 继续处理队列时，接下来的两个节点都不向队列添加新点。



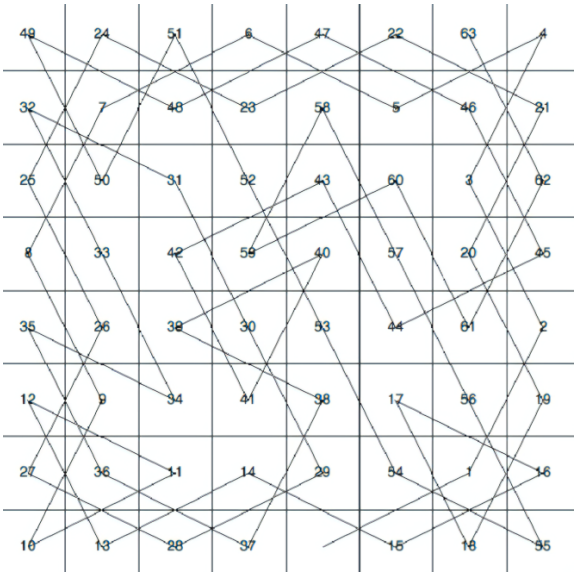
重复操作，直到完成图搜索。这个图看起来就像是一棵树。

8.6.2 广度优先搜索分析

bfs 对图中的每个顶点 V 最多执行一次 while 循环。因为顶点必须是白色，才能被检查和添加到队列，此操作过程性能为 O(v)。嵌套在 while 内部的 for 循环对图中的每个边执行最多一次，因为每个顶点最多被出队一次，并且仅当节点 u 出队时，才检查从节点 u 到节点 v 的边。所以 for 循环的性能为 O(E)，总的性能为 O(V+E)。

8.6.3 骑士之旅

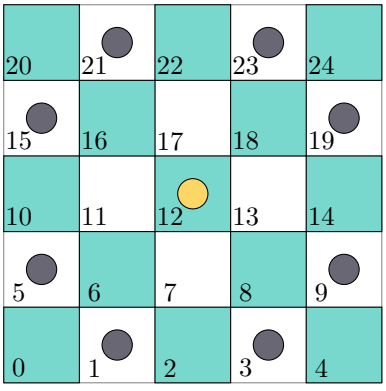
另一个可以用图来解决的问题是骑士之旅。骑士之旅是将棋盘上的棋子当骑士玩，目的是找到一系列的动作，让骑士访问板上的每个格一次，这样的序列被称为旅游。骑士之旅难题多年来一直吸引着象棋玩家、数学家和计算机科学家。一个 8×8 的棋盘可能的游览次数上限为 1.305×10^{35}，即有这么多条不重复路径满足到访问每个格子一次且不重复。

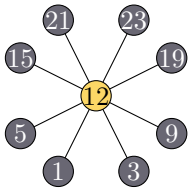


经过多年的研究，也发明了多种不同的算法来解决骑士旅游问题，图搜索是最容易理解的解决方案。图算法需要两个步骤来解决骑士之旅，一是如何表示骑士在棋盘上的动作，其次是查找长度为 $rows \times columns - 1$，的路径，-1 是因为自身所占格子不计入总数。

8.6.4 图解决骑士之旅

为将骑士旅游问题表示为图，可以将棋盘上的每个正方形表示为图中的一个点，骑士的每次合法移动（马走日）表示为图中的边，如下图，黄色处是骑士。





骑士能移动到的点

要构建一个 $n \times n$ 的完整图，可使用如下函数。该函数在整个板上进行一次遍历，在每个方块上为板上的位置创建一个移动列表，所有移动在图中转换为边。另一个辅助函数 pos_to_node_id 按照行和列将板上的位置转换为顶点。

```
// kinght_tour2.rs

fn kinght_graph(bdsize: u32) -> Graph {
    let mut g = Graph::new();

    for row in 0..bdsize {
        for col in 0..bdsize {
            let nodeid = pos_2_node_id(row,col,bdsize);
            let pos = gen_legal_move(row,col,bdsize);
            for e in pos {
                let nid = pos_2_node_id(e[0], e[1], bdsize);
                g.add_edge(nodeid, nid);
            }
        }
    }

    g
}

fn pos_2_nod_id(row: u32, col: u32, bdsize: u32) -> u32 {
    row * bdsize + col
}

fn gen_legal_move(x: u32, y: u32, bdsize) -> Vec<(u32, u32)> {
    let mut moves = vec![];

    // 马移动是坐标变化，共 8 个方向
    let move_offsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                        (1,-2),(1,2),(2,-1),(2,1)];
    for offset in move_offsets {
        let newx = x + offset[0];
        let newy = y + offset[1];
        if legal_coord(newx, bdsize)
            && legal_coord(newy, bdsize) {
            moves.push((newx,newy));
```

```
36              }
37          }
38          moves
39      }
40
41      fn legal_coord(x: u32, bdsize: u32) -> boool {
42          if x >= 0 && x < bdsize {
43              true
44          } else {
45              false
46          }
47      }
```

解决骑士旅游问题的搜索算法称为深度优先搜索。在前面讨论过广度优先搜索算法是尽可能广的在同一层搜索顶点，而深度优先搜索则是尽可能深地探索树的多层。可以通过设置多种不同的策略来利用深度优先搜索解决骑士之旅问题。第一种是通过明确禁止节点被访问多次来解决骑士之旅，第二种实现则更通用，允许在构建树时多次访问节点。深度优先搜索算法在找到死角（没有可移动的地方）时，它将回溯到上一个最深的点，继续探索。

```
1   // depth: 走过的路径长度， u: 起始点，path: 保存访问过的点
2   fn kinght_tour(depth: u32, path: &mut Vec,
3                  u: &mut Vec, limit: u32) -> bool {
4       u.set_color("gray");
5       path.push(u);
6       let mut done = false;
7       if depth < limit {
8           let nbrs = u.get_connects();
9           let mut i = 0;
10          while i < nbrs.len() && !done {
11              if nbrs[i].get_color() == "white" {
12                  done = kinght_tour(depth+1,path,nbrs[i],limit);
13              }
14              i += 1;
15          }
16          if !done {
17              let _rm = path.pop();
18              u.set_color("white");
19          }
20      } else {
```

```
21              done = true;
22          }
23
24          done
25      }
```

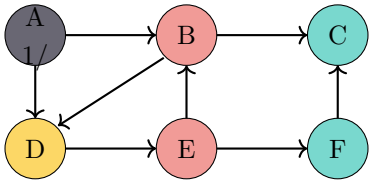
kinght_tour 是递归的深度优先搜索算法，若找到的路径等于 64，则返回，否则找寻下一个顶点来探索。深度优先搜索还使用颜色来跟踪图中的哪些顶点已被访问。未访问的顶点是白色的，访问的顶点是灰色的。如果已经探索了特定顶点的所有邻居，并且尚未达到 64 的目标长度，表明已经到达死胡同，此时必须回溯。当状态为 false 的 knightTour 返回时，会发生回溯。在广度优先搜索中，我们使用了一个队列来跟踪下一个要访问的顶点。由于深度优先搜索是递归的，所以有一个隐式的栈在帮助算法回溯。

8.7 深度优先搜索

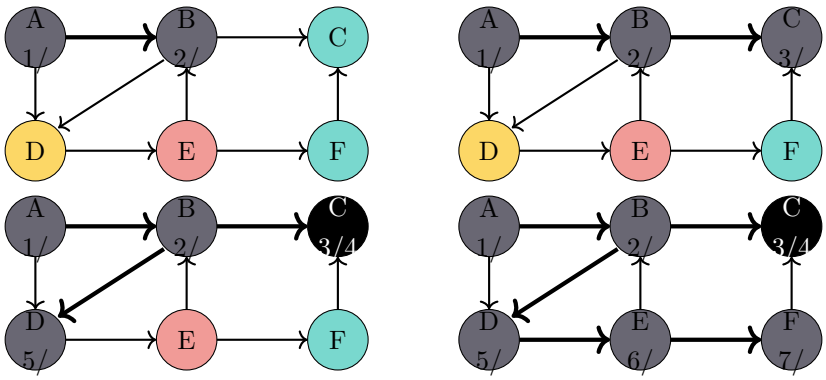
骑士之旅是深度优先搜索的特殊情况，其目的是创建一棵最深的树。更一般的深度优先搜索实际上是对有多个分支的图进行搜索。其目标是尽可能深地搜索，连接尽可能多的节点，并在必要时创建分支。

深度优先搜索可能创建多于一个树。当深度优先搜索算法创建一组树时，称之为深度优先森林。与广度优先搜索一样，深度优先搜索使用前导链接来构造树。此外，深度优先搜索将在顶点类中使用两个附加的变量。新变量是发现和结束时间。发现时间跟踪首次遇到顶点之前的步骤数，结束时间是顶点着色为黑色之前的步骤数。

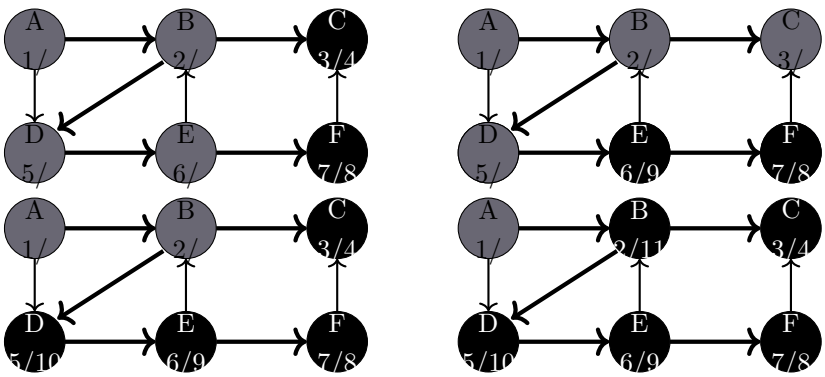
下图展示了一个小的图深度优先搜索算法。在这些图中，虚线指示检查的边，在边的另一端的节点已经被添加到深度优先树。搜索从图的顶点 A 开始。由于所有顶点在搜索开始时都是白色的，所以算法随机开始访问顶点 A。访问顶点的第一步是将颜色设置为灰色，这表示正在探索顶点，并且将发现时间设置为 1，由于顶点 A 具有两个相邻的顶点 (B，D)，因此每个顶点也需要被访问，我们就按字母顺序来访问相邻顶点。



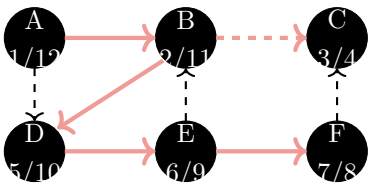
接下来访问顶点 B，设置颜色为灰色并且发现时间设置为 2。顶点 B 也与两个其他节点 (C，D) 相邻，因此接下来将访问节点 C。访问顶点 C 使我们到了树的一个分支末端，这意味着结束了对节点 C 的探索，因此可以将顶点着色为黑色，并将结束时间设置为 4。现在必须返回到顶点 B，继续探索与 B 相邻的节点。从 B 中探索的额外一个顶点是 D，接着访问 D。顶点 D 引导我们到顶点 E。顶点 E 具有两个相邻的顶点 B 和 F 。由于 B 已经是灰色的，所以算法识别出它不应该访问 B，继续探索的下一个顶点是 F。



顶点 F 只有一个相邻的顶点 C，但由于 C 是黑色的，没有别的点可以探索，算法已经到达另一个分支的末尾。因此算法必须回溯并不断将访问过的点标及为黑色。



最终，算法将回溯到最初的点，并退出搜索，得到如下图所示访问路径。红线是访问过的路径，最深路径为 A->B->D->E->F。



8.7.1 实现深度优先搜索

结合前面的内容及图示，可以得到下面更通用深度优先搜索算法。

```
1 // dfs.rs
2 use std::rc::Rc;
3 use std::cell::RefCell;
```

```
4
5  // 链接
6  type Link = Option<Rc<RefCell<Node>>>;
7
8  // 节点
9  struct Node {
10     data: usize,
11     next: Link,
12 }
13
14 impl Node {
15     fn new(data: usize) -> Self {
16         Self { data: data, next: None }
17     }
18 }
19
20 // 图
21 struct Graph {
22     first: Link,
23     last: Link,
24 }
25
26 impl Graph {
27     fn new() -> Self {
28         Self { first: None, last: None }
29     }
30
31     fn is_empty(&self) -> bool {
32         self.first.is_none()
33     }
34
35     fn get_first(&self) -> Link {
36         self.first.clone()
37     }
38
39     fn print_node(&self) {
40         let mut curr = self.first.clone();
41         while let Some(val) = curr {
```

```
42              print!("[{}]", &val.borrow().data);
43              curr = val.borrow().next.clone();
44          }
45          print!("\n");
46      }
47
48      // 插入数据
49      fn insert(&mut self, data: usize) {
50          let node = Rc::new(RefCell::new(Node::new(data)));
51          if self.is_empty() {
52              self.first = Some(node.clone());
53              self.last = Some(node);
54          } else {
55              self.last.as_mut()
56                      .unwrap()
57                      .borrow_mut()
58                      .next = Some(node.clone());
59              self.last = Some(node);
60          }
61      }
62  }
63
64  // 构建图
65  fn create_graph(data: [[usize;2];20]) -> Vec<(Graph, usize)> {
66      let mut arr: Vec<(Graph, usize)> = Vec::new();
67      for _ in 0..9 {
68          arr.push((Graph::new(), 0));
69      }
70
71      for i in 1..9 {
72          for j in 0..data.len() {
73              if data[j][0] == i {
74                  arr[i].0.insert(data[j][1]);
75              }
76          }
77          print!("[{i}]->");
78          arr[i].0.print_node();
79      }
```

```
80
81      arr
82  }
83
84  fn dfs(graph: Vec<(Graph, usize)>) {
85      let mut gp = graph;
86      let mut nodes: Vec<usize> = Vec::new();
87      let mut temp: Vec<usize> = Vec::new();
88
89      gp[1].1 = 1;
90      let mut curr = gp[1].0.get_first().clone();
91
92      // 打印图
93      print!("{1}->");
94      while let Some(val) = curr {
95          nodes.insert(0,val.borrow().data);
96          curr = val.borrow().next.clone();
97      }
98
99      // 打印深度优先图
100     loop{
101         if 0 == nodes.len() {
102             break;
103         }else{
104             let data = nodes.pop().unwrap();
105             if 0 == gp[data].1 {
106                 gp[data].1 = 1;
107                 print!("{data}->");
108
109                 // 节点加入 temp
110                 let mut curr = gp[data].0.get_first().clone();
111                 while let Some(val) = curr {
112                     temp.push(val.borrow().data);
113                     curr = val.borrow().next.clone();
114                 }
115
116                 while !temp.is_empty(){
117                     nodes.push(temp.pop().unwrap());
```

```
118                     }
119                 }
120             }
121         }
122
123         println!("");
124     }
125
126     fn main() {
127         let data = [[1,2],[2,1],[1,3],[3,1],[2,4],[4,2],[2,5],
128                     [5,2],[3,6],[6,3],[3,7],[7,3],[4,5],[5,4],
129                     [6,7],[7,6],[5,8],[8,5],[6,8],[8,6]];
130
131         let gp = create_graph(data);
132         dfs(gp);
133     }
```

dfs 迭代所有白点，并使用 dfsvisit 来访问节点。dfsvisit 从一顶点开始，并尽可能深地探查所有相邻白色顶点，dfsvisit 和 dfs 算法几乎一样，除了 for 循环最后一行，dfsvisit 递归调用自己进行深度搜索，而 bfs 负责控制回溯后要访问的顶点。bfs 使用的的是队列，因为节点的访问是按先后顺序的，而 dfsvisit 使用栈来管理递归。通过这里可以看到栈、队列这些基础的数据结构在解决大问题中的作用。

8.7.2 深度优先搜索分析

深度优先搜索 dfs 中的循环都在 O(V) 中运行，可以不计入 dfsvisit 中的时间，因为它们对图中的每个顶点执行一次。在 dfsvisit 中，对当前顶点的邻接表中的每个边执行一次循环。由于只有当顶点为白色时，dfsvisit 才被递归调用，所以循环对图中的每个边执行最多一次，其复杂度为 O(E)。综上，深度优先搜索的总时间性能是 O(V+E)。

8.7.3 拓扑排序

图的出现使得我们可以把许多现实世界的问题抽象成图，并利用图的性质和算法来解决现实世界的大问题。让我们考虑做煎饼的问题，如图（8.5）所示。菜谱很简单：鸡蛋 1 个，煎饼粉 1 杯，1 汤匙油和 3/4 杯牛奶。要制作煎饼，你必须加热炉子，将所有的材料混合在一起，勺子搅拌。当开始冒泡，再把它们翻过来，直到底部变金黄色。在你吃煎饼之前，你可能还要热一些糖浆。制作煎饼的困难是不知道先做那一个步骤。从图中看，你可以先打开煎锅，或混合原材料。为了决定每个步骤的精确顺序，可以用算法来将图排序，这种排序称为拓扑排序。拓扑排序采用有向无环图，并且产生所有顶点的线性排序。定向非循环图可以用来指示事件的优先级，设定软件项目计划，产生数据库查询的优先图等。

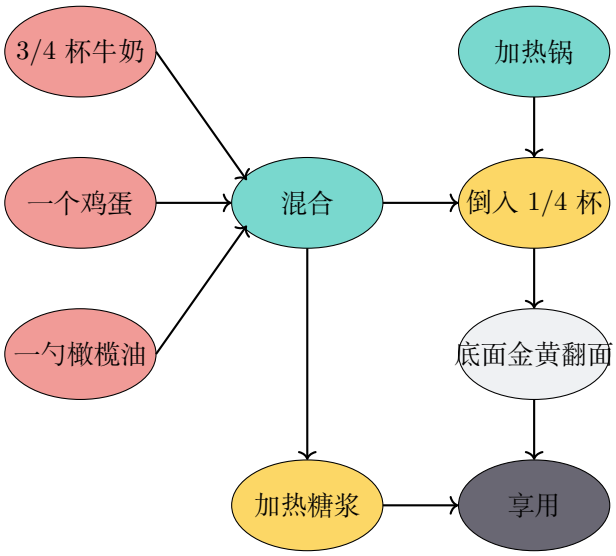
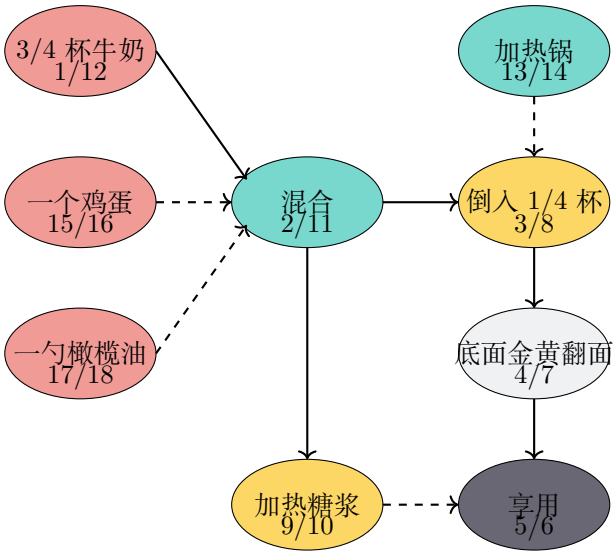


图 8.5: 制作煎饼

拓扑排序是深度优先搜索的一种改造，拓扑排序的算法如下:

```
1 对图 g 调用 dfs(), 用深度优先搜索的来计算每个顶点的结束时间。
2 以结束时间的递减顺序将顶点存储在列表中。
3 返回有序列表作为拓扑排序的结果。
```

拓扑排序可以将图转化成具有线性关系的图，如将上述菜谱将转化成如下深度优先森林及步骤关系图。



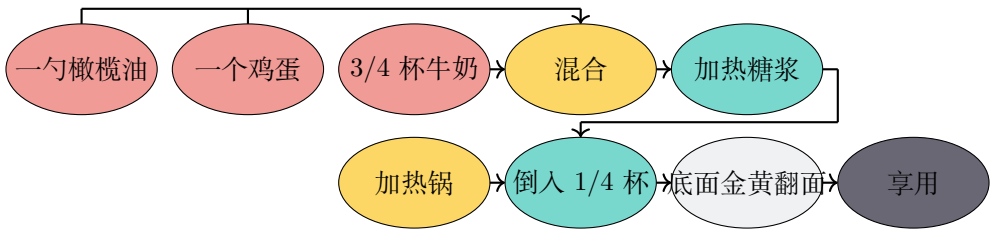


图 8.6: 制作煎饼的深度优先森林及步骤关系图

8.8 强连通分量

一些非常大的图，比如网页间产生的链接，也是图。百度，谷歌等搜索引擎存储的海量链接就是庞大的有向图。为将万维网变换为图，可将页面视为顶点，并将页面上的超链接作为顶点间连接的边。下图是 Google 主站点链接的其他站点，整个 Internet 络都在图中。

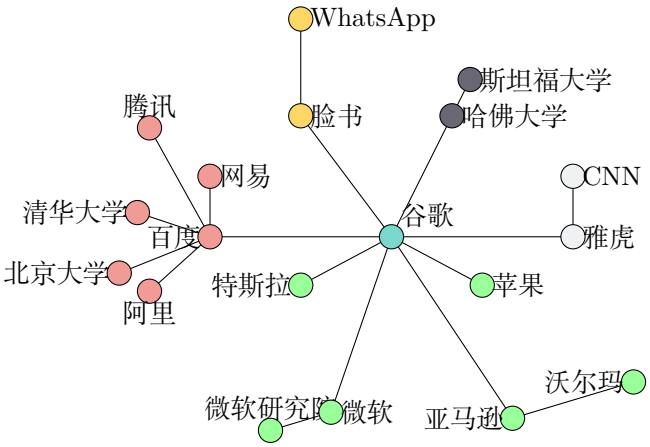


图 8.7: Internet 连接图

这类图有个明显的特点：某些节点链接特别多。如谷歌几乎和世界上任何网站连接起来，而有的节点却只有一个链接。也就是说，节点分区域聚集，高度互连。聚集的点区域称为连通区域，节点是强连通的。强连通是指在某个区域内，从任意节点可在有限路径内到达另一节点。为找到哪些节点组成强连通区域，可以用连通分量算法来计算。

强连通分量 C ∈ G，其中每个点 v, w ∈ G，且点 v 和 w 相互可达，图（8.8）是强连通分量图及其简化图，左侧三种颜色区域强连通。确定了强连通分量，就可以将其看成一个点以简化图。为获取强连通分量，需要对图调用深度优先搜索。首先将图连接反转 G^T，生成的图还是强连通的，且连通分量不变，如图（8.9）。计算强连通分量的算法步骤如下：

```
1 调用 dfs 为 G 计算每个点的结束时间
2 计算 G^T，并为图 G^T 调用 dfs，计算每个点结束时间
3 输出每个森林中每棵树顶点的标志组件
```

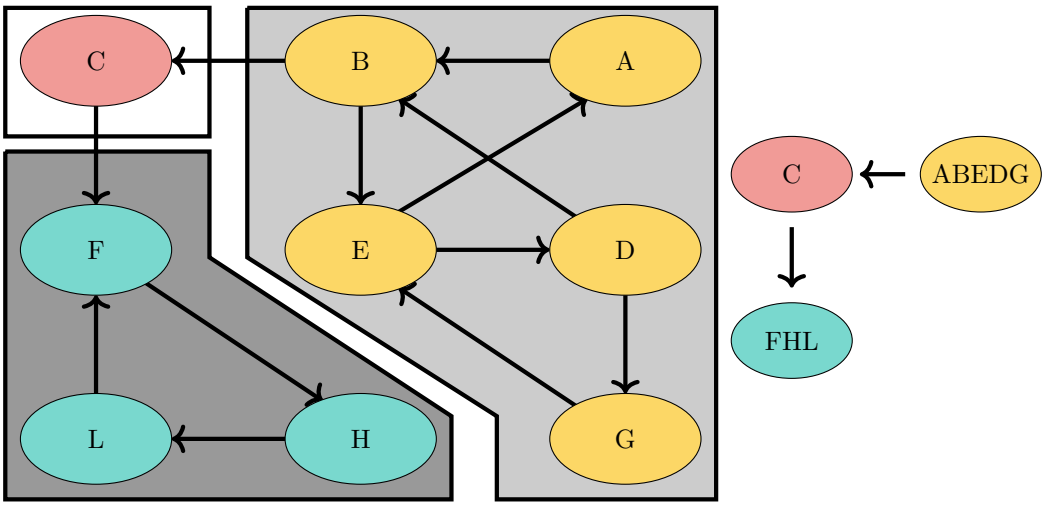


图 8.8: 强连同分量图（左），及其简化图（右）

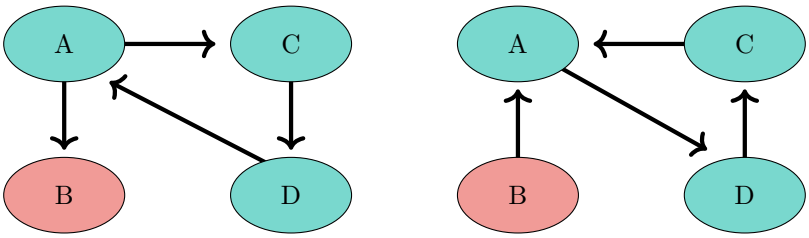
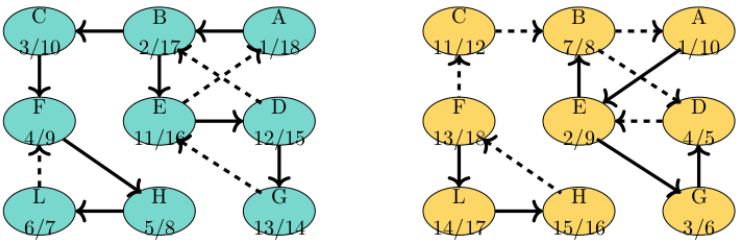
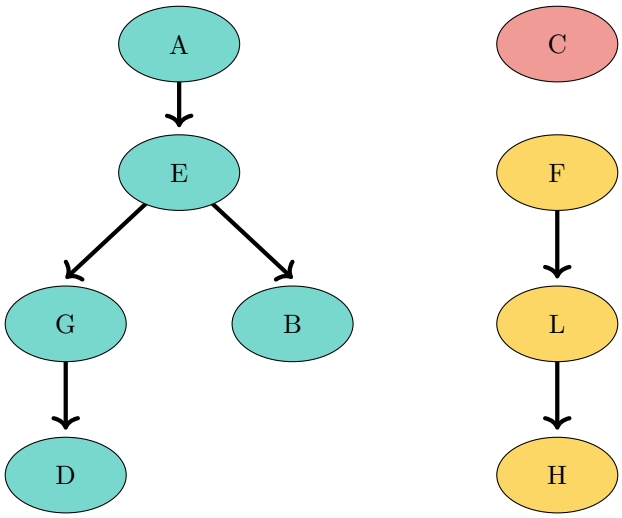


图 8.9: 图及其反转图

通过对连通图及其转置图使用深度优先搜索可得到如下连通图。实线表示连通区域内的搜索路径，虚线表示可连通路径，数字表示搜索的起止时间。无论是图还是其反转图，都能通过该算法准确得到连通区域。

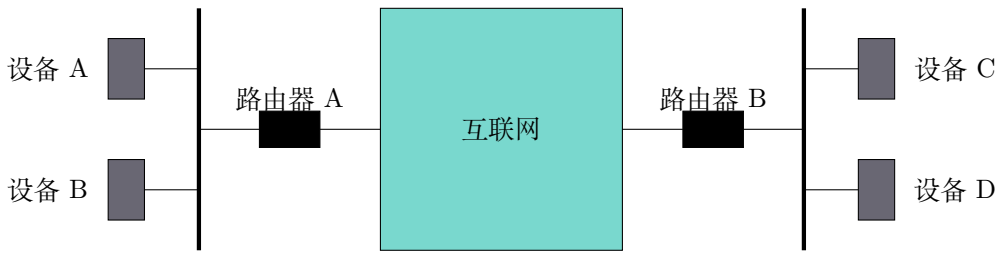


通过对连通图使用强连通分量算法可以得到三棵树，其实就是三个连通分量。在下图中可以看到连通区域是如何简化的。其中 C 是一个独立区域，虽然只有它自己一个节点。F，L，H 三点是一个连通区域，A，E，G，D，B 也是一个连通区域。通过强连通分量算法得到的三棵树就是三个连通区域，该算法将问题由节点层面转到了连通区域层面，极大地降低了问题难度，便于后续分析处理。



8.9 最短路径问题

上网看短视频，发邮件，或从校园外登陆录实验室计算机时，信息是由网络传输的。研究信息如何通过互联网从一台计算机流向另一台计算机是计算机网络中的一个大课题。



上图展示了 Internet 上的通信如何工作。使用浏览器从服务器请求网页时，请求必须通过局域网传输，并通过路由器传输到 Internet 上。该请求通过因特网传播，并最终到达服务器所在的局域网路由器，服务器返回的网页再通过相同的路由器回到您的浏览器。如果你的计算机支持 tracepath 命令，可以用它来查看你的电脑到某个链接的路径，比如如下追踪到达 xxx.cn 网站经过了 13 个路由器，其中第一二个是自己所在网络组的网关路由器。

```
1?: [LOCALHOST]                      pmtu 1500
1:  _gateway                          4.523 毫秒
1:  _gateway                          3.495 毫秒
2:  10.253.0.22                       2.981 毫秒
3:  无应答
4:  ???                               6.166 毫秒
5:  202.115.254.237                 558.609 毫秒
6:  无应答
7:  无应答
```

```
10   8:   101.4.117.54                    48.822 毫秒 asymm 16
11   9:   无应答
12  10:   101.4.112.37                    48.171 毫秒 asymm 14
13  11:   无应答
14  12:   101.4.114.74                    44.981 毫秒
15  13:   202.97.15.89                    49.560 毫秒
```

互联网上的每个路由器都连接到一个或多个路由器。因此，如果在一天的不同时间运行 tracepath，你看到的输出不一定一样。你很可能会看到信息在不同的时间不同，信息流经了不同的路由器。这是因为路由器之间的连接存在成本，同时还取决于网络流量情况。你可以将网络链接看成带有权重的图，连接会根据网络情况作调整。

我们的目标是找到具有最小总权重的路径，用于沿着该路径传送消息。这个问题类似于广度优先搜索解决的问题，这里关心的是路径的总权重，而不是路径条数。应当注意，如果所有权重相等，则问题是相同的。

8.9.1 Dijkstra 算法

研究网络图最短路径算法的前辈们提出了各种各样的算法，其中 Dijkstra 算法是搜索图中最短路径的好算法。Dijkstra 算法是一种贪心迭代算法，它为我们提供从一个特定起始节点到图中所有其他节点的最短路径，这有点类似于广度优先搜索。

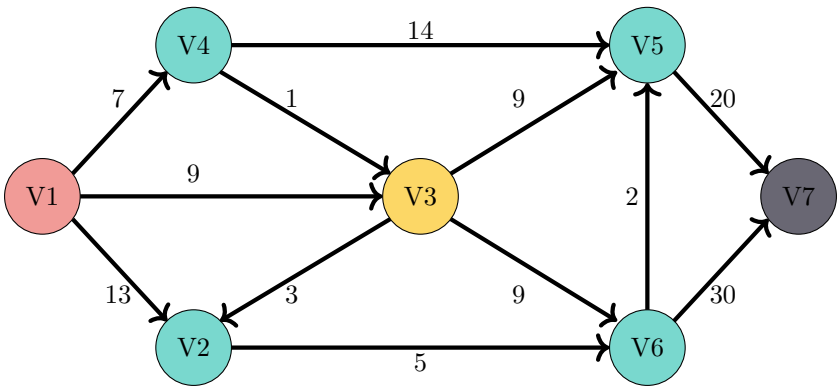


图 8.10: Dijkstra 图示

如上图，需要找到从 V1 到 V7 的最短路径，通过一定时间的探索，读者定能得出最短路径是 [V1->V4->V3->V2->V6->V5->V7]，最短距离为 38。如果通过算法来计算，就需要跟踪并计算各个距离并求和。

为跟踪从起始节点到每个目标节点的总距离，将使用图顶点中的 dist 实例变量。该实例变量将包含从开始到目标节点的路径总权重。Dijkstra 算法对图中的每个节点重复一次，在节点上迭代的顺序由优先级队列控制，而用于确定优先级队列中对象顺序的值便是 dist 值。首次创建节点时，dist 被设置为非常大的数。理论上，将 dist 设置为无穷大，但在实践中，

只将它设置为一个数字，大于任何真正的距离就可以了，比如光在一秒内行进的距离，也就是地球和月亮间的距离也是一个合理的值，因为地球上没有哪两个点的距离会达到这么大。

8.9.2 实现 Dijkstra 算法

Dijkstra 算法使用优先级队列处理顶点，用于保存一个键值对元组，值作为优先级别。

```
1  // dijkstra.rs
2
3  use std::cmp::Ordering;
4  use std::collections::{BinaryHeap, HashMap, HashSet};
5
6  // 点
7  #[derive(Debug, Copy, Clone, PartialEq, Eq, Hash)]
8  struct Vertex<'a> {
9      name: &'a str,
10 }
11
12 impl<'a> Vertex<'a> {
13     fn new(name: &'a str) -> Vertex<'a> {
14         Vertex { name }
15     }
16 }
17
18 // 访问过的点
19 #[derive(Debug)]
20 struct Visit<V> {
21     vertex: V,
22     distance: usize, // 距离
23 }
24
25 // 为 Visited 添加全序比较功能
26 impl<V> Ord for Visit<V> {
27     fn cmp(&self, other: &Self) -> Ordering {
28         other.distance.cmp(&self.distance)
29     }
30 }
31
32 impl<V> PartialOrd for Visit<V> {
33     fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
```

```
34          Some(self.cmp(other))
35      }
36  }
37
38  impl<V> Eq for Visit<V> {}
39
40  impl<V> PartialEq for Visit<V> {
41      fn eq(&self, other: &Self) -> bool {
42          self.distance.eq(&other.distance)
43      }
44  }
45
46  // 最短路径算法
47  fn dijkstra<'a>(
48      start: Vertex<'a>,
49      adj_list: &HashMap<Vertex<'a>, Vec<(Vertex<'a>, usize)>>
50      ) -> HashMap<Vertex<'a>, usize> {
51      let mut distances = HashMap::new();    // 距离
52      let mut visited = HashSet::new();      // 已访问过的点
53      let mut to_visit = BinaryHeap::new(); // 待访问的点
54
55      // 设置起始点和起始距离
56      distances.insert(start, 0);
57      to_visit.push(Visit {
58          vertex: start,
59          distance: 0,
60      });
61
62      while let Some(Visit {vertex, distance}) = to_visit.pop() {
63          // 已访问过该点, 继续下一个点
64          if !visited.insert(vertex) { continue; }
65
66          // 获取邻点
67          if let Some(neighbors) = adj_list.get(&vertex) {
68              for (neighbor, cost) in neighbors {
69                  let new_distance = distance + cost;
70                  let is_shorter = distances
71                      .get(&neighbor)
```

```
72                      .map_or(true, |&curr| new_distance < curr);
73
74                  // 若距离更近，则插入新距离和邻点
75                  if is_shorter {
76                      distances.insert(*neighbor, new_distance);
77                      to_visit.push(Visit {
78                          vertex: *neighbor,
79                          distance: new_distance,
80                      });
81                  }
82              }
83          }
84      }
85
86      distances
87  }
88
89  fn main() {
90      let s = Vertex::new("s");
91      let t = Vertex::new("t");
92      let x = Vertex::new("x");
93      let y = Vertex::new("y");
94      let z = Vertex::new("z");
95
96      let mut adj_list = HashMap::new();
97      adj_list.insert(s, vec![(t, 10), (y, 5)]);
98      adj_list.insert(t, vec![(y, 2), (x, 1)]);
99      adj_list.insert(x, vec![(z, 4)]);
100     adj_list.insert(y, vec![(t, 3), (x, 9), (z, 2)]);
101     adj_list.insert(z, vec![(s, 7), (x, 6)]);
102
103     let distances = dijkstra(s, &adj_list);
104
105     for (v, d) in &distances {
106         println!("{} to {}, min distance: {d}",
107                  s.name, v.name);
108     }
109
```

```
110         assert_eq!(distances.get(&t), Some(&8));
111         assert_eq!(distances.get(&s), Some(&0));
112         assert_eq!(distances.get(&y), Some(&5));
113         assert_eq!(distances.get(&x), Some(&9));
114         assert_eq!(distances.get(&z), Some(&7));
115     }
```

在互联网上使用 Dijkstra 算法的一个问题是，为了使算法运行，你必须有完整的网络的图表示。这意味着每个路由器都有完整的互联网中所有路由器的地图，实际上是根本不可能的。这意味着通过因特网路由器发送消息需要使用其他算法来找到最短路径。实际中网络信息传递使用的是距离矢量路由算法和状态路由算法，些类算法允许路由器在发送信息时发现对方路由器保存的网络图，这些图包含相互连通的节点信息。通过实时发现这样的方式获取网络图内容更高效，同时容量大大减小了。

8.9.3 Dijkstra 算法分析

最后，来看 Dijkstra 算法的时间复杂度。构建优先级队列需要 O(v)，一旦构造了队列，则对于每个顶点要执行一次 while 循环。因为顶点都在开始处添加，并且在那之后才被移除。在循环中每次调用 pop，需要 $O(logV)$ 时间。将该部分循环和对 pop 的调用取为 $O(VlogV)$。for 循环对于图中的每条边执行一次，在 for 循环中，对 decrease_key 的调用需要时间 $O(ElogV)$，因此，总的时间复杂度为 $O((V+E)logV)$。

8.10 总结

本章学习了图抽象数据类型，以及图的实现。图在网络，交通，计算机，人工智能知识图谱等领域非常有用。图使我们能够解决许多问题，只要可以将原始问题转换为图表示。图在以下领域有较好的应用。

```
1 强连通分量用于简化图。
2 深度优先搜索图的深分支。
3 拓扑排序用于理清复杂的图连接。
4 Dijkstra 用于搜索加权图的最短路径。
5 广度优先搜索用于搜索无加权图的最短路径。
```

Chapter 9

实战

9.1 本章目标

用 Rust 数据结构和算法来完成实战项目
学习并理解实战项目中的数据结构和算法

9.2 编辑距离

9.2.1 汉明距离

汉明距离（Hamming distance）是指两个相同长度的序列在相同位置上有多少个符号不同，对二进位序列来说就是“相异的比特数目”。汉明距离同时也是一种编辑距离，即将一个字符串转换成另一个字符串需要经过多少次替换操作。比如下图中，trust 转换为 rrost 只需要替换两个字符，所以汉明距离是 2。

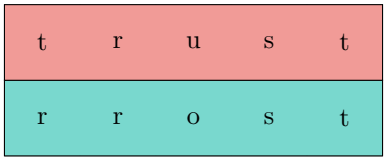


图 9.1: 字符串的汉明距离

汉明距离多用于编码中的错误更正，汉明码 [12] 中计算距离的算法即为汉明距离。为简化代码，我们将处理数字和处理字符的汉明距离算法分别实现。计算数字的汉明距离非常简单，因为数字可以用位运算直接比较异同，下面是计算数字汉明距离的代码。

```
1 // hamming_distance.rs
2 fn hamming_distance1(source: u64, target: u64) -> u32 {
3     let mut count = 0;
```

```
    let mut xor = source ^ target;

    // 异或取值
    while xor != 0 {
        count += xor & 1;
        xor >>= 1;
    }

    count as u32
}

fn main() {
    let source = 1;
    let target = 2;
    let distance = hamming_distance1(source, target);
    println!("the hamming distance is {distance}");
}
```

通过异或操作可以让数字 source 和 target 中相同位为 0，不同位为 1，如果结果不等于 0，则说明有不同的位，所以从最后一位逐步计算不同的位。xor 与 1 相与就能得到最后一位是 0 还是 1。每计算一位就要移除一位以便比较前面的比特位，所以加入了右移操作。当然，前面的实现需要自己计算二进制中的 1 个数，实际上 Rust 的数字自带一个 count_ones() 函数用于计算 1 的个数，所以上述代码可以化简成如下代码，非常简单。

```
// hamming_distance.rs

fn hamming_distance2(source: u64, target: u64) -> u32 {
    (source ^ target).count_ones()
}

fn main() {
    let source = 1;
    let target = 2;
    let distance = hamming_distance2(source, target);
    println!("the hamming distance is {distance}");
}
```

有了上面的基础，下面来实现字符版的汉明距离，当然，此时无法用位运算。

```
// hamming_distance.rs
fn hamming_distance_str(source: &str, target: &str) -> u32 {
```

```
    let mut count = 0;
    let mut source = source.chars();
    let mut target = target.chars();

    // 两字符串逐字符比较可能出现如下四种情况
    loop {
        match (source.next(), target.next()) {
            (Some(cs), Some(ct)) if cs != ct => count += 1,
            (Some(_), None) | (None, Some(_)) => panic!("Must
                                                have the same lenght"),
            (None, None) => break,
            _ => continue,
        }
    }

    count as u32
}

fn main() {
    let source = "abce";
    let target = "edcf";
    let distance = hamming_distance_str(source, target);
    println!("the hamming distance is {distance}");
}
```

字符版汉明距离算法还是接受 source 和 target 两个输入，然后用 chars 方法取出 Unicode 字符来比较。使用 Unicode 而非 ASCII 是因为字符可能不只有字母，还有中文、日文、韩文等其他字符。if c1 != c2 是在模式匹配之外，额外的条件检查，只有 source 和 target 都有下一个字符且两字符不相等时才会进入该匹配分支。若有任何一个字符是 None，另外一个是 Some，表示输入字串的长度不同，可直接返回。如果都没有下一个字符了，则结束。其他情况表示两个字符相同，则继续比较下一个字符。汉明距离需要计算所有的字符，所以时间复杂度为 O(n)，空间复杂度为 O(1)。

9.2.2 莱文斯坦距离

莱文斯坦距离又称编辑距离（Edit distance），是一种量化两字符串差异的算法，表示的是从一个字符串转换为另一个字符串最少需要多少次编辑操作。这些操作包括插入、删除、替换。编辑距离概念非常好理解，操作也简单，可用于简单的字符修正。比如用莱文斯坦距离算法来计算单词 sitting 和 kitten 的编辑距离，可以用如下步骤将 kitten 转换为 sitting。

（1）sitting -> kitting，替换 s 为 k。

（2）kitting -> kitteng，替换 i 为 e。

（3）kitteng -> kitten，删除 g。

因为处理了 3 次，所以编辑距离为 3。现在的问题是，如何证明 3 就是最少的编辑次数呢？因为两个字符串间的转换只有三种操作：删除，插入，替换，所以可以分类单独计算每种操作的次数，最后取最小值。

一种极端情况是从空字符串转换为某长度的字符串 s，此时的编辑距离很明显就是 s 的长度。比如从空字符串转换为 abc，那么需要插入三个字符，编辑距离就是 3。这同时也说明编辑距离的上限就是较长字符串的长度，用数学公式表达就是下式。

$$max(i,j) \quad if \;\; min(i,j) = 0 \tag{9.1}$$

min(i,j) = 0 表示没有公共子串，此时取最长字符串的长度。除了这种极端情况，还有可能出现三种操作，而每种操作都会使编辑距离加 1，所以可以分别计算三种操作得到的距离再取最小值，其中 edi 指编辑距离。

$$edi_{a,b}(i,j) = min \begin{cases} edi_{a,b}(i-1,j)+1 \\ edi_{a,b}(i,j-1)+1 \\ edi_{a,b}(i-1,j-1)+1_{a \neq b} \end{cases} \tag{9.2}$$

$edi_{a,b}(i-1,j)+1$ 表示从 a 到 b 要删除 1 个字符，编辑距离加 1；$edi_{a,b}(i,j-1)+1$ 表示从 a 到 b 要插入 1 个字符，编辑距离再加 1；$edi_{a,b}(i-1,j-1)+1_{a \neq b}$ 表示从 a 到 b 要替换 1 个字符，编辑距离加 1。注意 a 和 b 不等才替换，距离才加 1，相等就跳过。这些函数计算是递归定义的，其空间复杂度为 $O(3^{m+n-1})$，m 和 n 为字符串的长度。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1							
i	2							
t	3							
t	4							
e	5							
n	6							

前面我们学习过动态规划可以处理递归，所以此处采用动态规划算法。首先需要一个矩阵来存储各种操作后的编辑距离，最基本的情况就是从空字符串到不同长度的字符串所需的编辑距离，如上图所示。

接下来要计算字符 k 和 s 的编辑距离，分三种情况，如下图所示。

(1) 红色上方累积删除的编辑距离 1，加上删除操作，则编辑距离为 2。

(2) 红色左方累积插入的编辑距离 1，加上插入操作，则编辑距离为 2。

(3) 红色对角线累积替换的编辑距离 0，加上替换操作，则编辑距离为 1。

仔细观察，可以发现处理的数值都是下图中黄色区域的值，开始计算时选择左上角，通过对黄色区域的三个值进行计算，最后选择了结果中最小的值作为编辑距离填入红色处，得到了的新的编辑距离。下图中青色值为空字符串转换为当前长度字符串所需要的编辑次数，这些数值是最极端情况下的编辑距离。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1						
i	2							
t	3							
t	4							
e	5							
n	6							

根据上面的描述和图可以写出如下的算法。

```
1  // edit_distance.rs
2
3  use std::cmp::min;
4
5  fn edit_distance(source: &str, target: &str) -> usize {
6      // 极端情况：空字符串到字符串的转换
7      if source.is_empty() {
8          return target.len();
9      } else if target.is_empty() {
10         return source.len();
```

```
11      }
12
13      // 建立矩阵存储过程值
14      let source_c = source.chars().count();
15      let target_c = target.chars().count();
16      let mut distance = vec![vec![0;target_c+1]; source_c+1];
17      for i in 1..=source_c {
18          distance[i][0] = i;
19      }
20      for j in 1..=target_c {
21          distance[0][j] = j;
22      }
23
24      // 存储过程值，取增、删、改中的最小步骤数
25      for (i, cs) in source.chars().enumerate() {
26          for (j, ct) in target.chars().enumerate() {
27              let ins = distance[i+1][j] + 1;
28              let del = distance[i][j+1] + 1;
29              let sub = distance[i][j] + (cs != ct) as usize;
30              distance[i+1][j+1] = min(min(ins, del), sub);
31          }
32      }
33
34      // 返回最后一行最后一列的值
35      *distance.last().and_then(|d| d.last()).unwrap()
36  }
37
38  fn main() {
39      let source = "abce";
40      let target = "adcf";
41      let distance = edit_distance(source, target);
42      println!("the edit distance is {distance}");
43
44      let source = "bdfc";
45      let target = "adcf";
46      let distance = edit_distance(source, target);
47      println!("the edit distance is {distance}");
48  }
```

可通过逐步移动黄色区域来选择需要计算的三个值，再取计算结果中的最小值填入当前黄色区域的右下角，计算的最终结果如下图。

		s	i	t	t	i	n	g
	0	1	2	3	4	5	6	7
k	1	1	2	3	4	5	6	7
i	2	2	1	2	3	4	5	6
t	3	3	2	1	2	3	4	5
t	4	4	3	2	1	2	3	4
e	5	5	4	3	2	2	3	4
n	6	6	5	4	3	3	2	3

算完整个编辑距离矩阵后，最后一行最后一列的值就是编辑距离。仔细分析上面的图就会发现，整个矩阵是二维的，处理时要仔细使用下标。一种比较直观的方式是将矩阵的每一行拉出来放到数组中组成一个大数组，总量还是 m*n，但维度却小了。但是计算值只有最后一个值有用，而大量中间值浪费了太多内存。因为计算过程中只需要矩阵中左侧黄色区域的值，那么就可以再优化算法，在计算过程中可以反复利用一个数组来计算和保存值。将矩阵缩小为 m + 1 或 n + 1 长度的数组。经过优化的编辑距离算法代码如下。

```
1  // edit_distance.rs
2
3  use std::cmp::min;
4
5  fn edit_distance2(source: &str, target: &str) -> usize {
6      if source.is_empty() {
7          return target.len();
8      } else if target.is_empty() {
9          return source.len();
10     }
11
12     // distance 存储了到各种字符串的编辑距离
13     let target_c = target.chars().count();
14     let mut distances = (0..=target_c).collect::<Vec<_>>();
```

```
15      for (i, cs) in source.chars().enumerate() {
16          let mut substt = i;
17          distances[0] = substt + 1;
18          for (j, ct) in target.chars().enumerate() {
19              let dist = min(min(distances[j], distances[j+1])+1,
20                             substt + (cs != ct) as usize);
21              substt = distances[j+1];
22              distances[j+1] = dist;
23          }
24      }
25
26      // 最后一个距离值就是最终答案
27      distances.pop().unwrap()
28  }
29
30  fn main() {
31      let source = "abce";
32      let target = "adcf";
33      let distance = edit_distance2(source, target);
34      println!("the edit distance is {distance}");
35  }
```

优化过的编辑距离算法最坏时间复杂度为 O(mn)，最坏空间复杂度由矩阵的 O(mn) 降为 O(min(m,n))，这是非常大的进步。

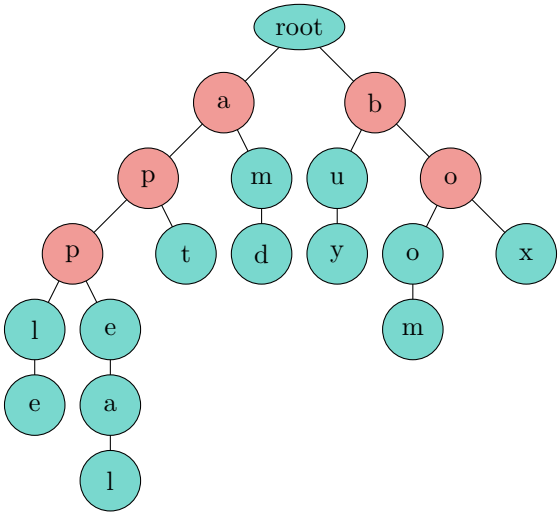
最后提一下微软的 Word 软件。Word 也有拼写检查功能，但不是用的编辑距离，而是用的散列表。它将常用的几十万单词存储到散列表，每输入一个单词就到散列表中查找，找不到就报错。散列表速度非常快，而几十万单词也就几 M 内存，所以非常高效。

9.3 字典树

Trie (音 try) 是一种树数据结构，又称为字典树，前缀树，用于检索某个单词或前缀是否存在于树中。Trie 应用广泛，包括打字预测，自动补全，拼写检查等。

平衡树和哈希表也能够用于搜索单词，为什么还需要 Trie 呢? 哈希表能在 O(1) 时间内找到单词，但却无法快速地找到具有同一前缀的全部单词或按字典序枚举出所有存储的单词。Trie 树优于哈希表的另一个点是，单词越多，哈希表就越大，这意味着可能出现大量冲突，时间复杂度可能增加到 O(n)。与哈希表相比，Trie 树存储多个具有相同前缀的单词时可以使用更少的空间，时间复杂度也只有 O(m)，m 为单词长度，而在平衡树中查找单词的时间复杂度为 O(m log(n)) 。

Trie 树结构如下，可以发现，存储单词只用处理 26 个字母就够了，而且同样前缀的单词共享前缀，节省了存储空间，比如 apple 和 appeal 共享 app，boom 和 box 共享 bo。



为实现 Trie，我们首先要抽象出结点 Node ，类似上图中的结点。Node 中保存子结点的引用和当前结点的状态。状态指此结点是否是单词结束（end），在查询时可用于判断单词是否结束。此外根结点（root）是 Trie 的入口，可以将其用于代表整个 Trie。

```
1  // trie.rs
2
3  // 字典树定义
4  #[derive(Default)]
5  struct Trie {
6      root: Node,
7  }
8
9  // 节点
10 #[derive(Default)]
11 struct Node {
12     end: bool,
13     children: [Option<Box<Node>>; 26], // 字符节点列表
14 }
15
16 impl Trie {
17     fn new() -> Self {
18         Self::default()
19     }
```

```
20
21      // 单词插入
22      fn insert(&mut self, word: &str) {
23          let mut node = &mut self.root;
24          // 逐个字符插入
25          for c in word.as_bytes() {
26              let index = (c - b'a') as usize;
27              let next = &mut node.children[index];
28              node = next.get_or_insert_with(Box::<Node>::default);
29          }
30          node.end = true;
31      }
32
33      fn search(&self, word: &str) -> bool {
34          self.word_node(word).map_or(false, |n| n.end)
35      }
36
37      // 判断是否存在以某个前缀开头的单词
38      fn start_with(&self, prefix: &str) -> bool {
39          self.word_node(prefix).is_some()
40      }
41
42      // 前缀字符串
43      // wps: word_prefix_string
44      fn word_node(&self, wps: &str) -> Option<&Node> {
45          let mut node = &self.root;
46          for c in wps.as_bytes() {
47              let index = (c - b'a') as usize;
48              match &node.children[index] {
49                  None => return None,
50                  Some(next) => node = next.as_ref(),
51              }
52          }
53          Some(node)
54      }
55  }
56
57  fn main() {
```

```
58      let mut trie = Trie::new();
59      trie.insert("box"); trie.insert("insert");
60      trie.insert("apple"); trie.insert("appeal");
61
62      let res1 = trie.search("apple");
63      let res2 = trie.search("apples");
64      let res3 = trie.start_with("ins");
65      let res4 = trie.start_with("ina");
66      println!("word 'apple' in Trie: {res1}");
67      println!("word 'apples' in Trie: {res2}");
68      println!("prefix 'ins' in Trie: {res3}");
69      println!("prefix 'ina' in Trie: {res4}");
70  }
```

9.4 过滤器

9.4.1 布隆过滤器

在软件开发中，常要判断一个元素是否在一个集合中。比如在字处理软件中，需要检查一个英语单词是否拼写正确（也就是判断它是否在已知的字典中，编辑距离一节已经讲过 Word 的拼写检查）；在 FBI，要快速判断一个嫌疑人名字是否在嫌疑名单上并给出 FBI Warning；在网络爬虫里，判断一个网址是否被访问过等等。最直接的方法就是将集合中全部的元素存在计算机中，遇到一个新元素时，将它和集合中的元素直接比较即可。一般来讲，计算机中的集合是用哈希表来存储的，其好处是快速准确，缺点是费空间。

当集合比较小时，这个问题不显著，但是当集合巨大时，哈希表存储效率低的问题就显现出来了。比如说，一个像雅虎或谷歌邮箱这样的电子邮件提供商，总是需要过滤垃圾邮件。一个办法就是记录下那些发垃圾邮件的地址，由于那些发送者不停注册新地址，将其都存起来则需要大量的网络服务器。如果用哈希表，存储一亿个邮件地址就需要 1.6G 左右的内存。将这些信息存入哈希表理论上是可行的，但哈希表存在负载因子，通常空间不能被用满。此外，如果数据集存储在远程服务器上，要在本地接受输入，而数据集非常大不可能一次性读进内存构建出哈希表，这时候也会存在问题。

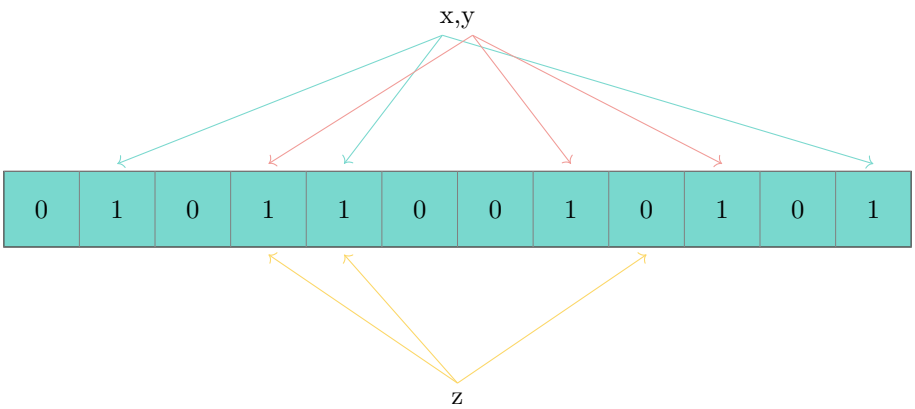
这时候就需要考虑类似布隆过滤器这样的数据结构。布隆过滤器由布隆（Burton Howard Bloom）在 1970 年提出，它由一个很长的二进制向量和一系列随机映射函数组成。布隆过滤器可用于检索一个元素是否在一个集合中，它的优点是空间效率和查询效率都远远超过一般的数据结构，缺点是有一定的识别误差率且删除较困难。布隆过滤器本质上是一种巧妙的概率型数据结构。

布隆过滤器包含一个能保存 n 个数据的二进制向量（位数组）和 k 个哈希函数。布隆过滤器支持插入和查询两种基本操作，但插入的值总数在设计时就定好了，所以针对不同问题，

要详细设计布隆过滤器的大小。

初始化布隆过滤器时，所有位置置 0。插入数据时，利用 k 个哈希函数计算数据在过滤器中的位置并将对应位置置 1。比如 k = 3 时，计算三个哈希值作为下标，并将对应值全部置为 1。查询时同样通过 k 个哈希函数产生 k 个哈希值作为索引，若所有索引对应的值皆为 1，则代表该值可能存在。

下图是三个值对应的情况，x 和 y 都在布隆过滤器中，而 z 因为最后一个哈希值为 0，所以一定不存在。想要体验布隆过滤器的可以到 bloomfilter 这个地址尝试。



已知布隆过滤器长度为 n，在可容忍的误差率为 ϵ 的情况下，此时最佳的存储个数为 m：

$$m = -\frac{nln\epsilon}{(ln2)^2} \tag{9.3}$$

而此时需要的哈希函数个数 k 为：

$$k = -\frac{ln\epsilon}{ln2} = log_2\epsilon \tag{9.4}$$

假如容忍的误差率 $\epsilon = 8\%$，那么 k = 3，k 越大代表误差率越大。在不改变容错率的情况下，可以组合迭代次数和两个基本哈希函数来模拟 k 个哈希函数。

$$g_i(x) = h_1(x) + ih_2(x) \tag{9.5}$$

为实现布隆过滤器，我们采用一个结构体来封装所需的全部结构，包括存储位的集合和哈希函数。因为只有 1 和 0 两种情况，我们将其转换为 true 或 false 并保存到 Vec 中。这样在判断的时候，值是布尔值，可以直接表示是否存在。因为布隆过滤器只判断值是否存在，或者说此前是否出现过并有所记录，所以它必定需要适合任意类型的数据，必然采用泛型。

```
1 // bloom_filter.rs
2
3 use std::collections::hash_map::DefaultHasher;
4
5 // 布隆过滤器
```

```
struct BloomFilter<T> {
    bits: Vec<bool>,                // 比特桶
    hash_fn_count: usize,           // 哈希函数个数
    hashers: [DefaultHasher; 2],    // 两个哈希函数
}
```

但是上述代码编译出错，因为泛型 T 并没有被哪个字段用了，那么编译器认为这是非法的。要让它编译通过，则需要使用 Rust 中的幽灵数据（PhantomData）来占位，假装使用了 T，但又不占内存，其实就是骗编译器，使它放过我们的代码。最后，为了尽可能多的支持存储的数据类型，对于编译期不定大小的数据我们也要支持，所以加上?Sized 特性让过滤器支持不定大小的数据。_phantom 前缀表示该字段不使用，占用为 0。但它带上了 T，所以可以骗过编译器。此外，我们使用两个随机的哈希函数来模拟 k 个哈希函数。

```
// bloom_filter.rs

use std::collections::hash_map::DefaultHasher;
use std::marker::PhantomData;

// 布隆过滤器
struct BloomFilter<T: ?Sized> {
    bits: Vec<bool>,
    hash_fn_count: usize,
    hashers: [DefaultHasher; 2],
    _phantom: PhantomData<T>,    // T 占位，欺骗编译器
}
```

为了实现过滤器的功能，我们还需要为其实现三个函数，分别是初始化函数 new，新增元素函数 insert，检测函数 contains。此外还有辅助函数，用于实现前面三个函数。new 函数需要根据容错率和大致的存储规模计算出 m 的大小并初始化过滤器。

```
// bloom_filter.rs

use std::hash::{BuildHasher, Hash, Hasher};
use std::collections::hash_map::RandomState;

impl<T: ?Sized + Hash> BloomFilter<T> {
    fn new(cap: usize, ert: f64) -> Self {
        let ln22 = std::f64::consts::LN_2.powf(2f64);
        // 计算比特桶大小和哈希函数个数
        let bits_count = -1f64 * cap as f64 * ert.ln() / ln22;
        let hash_fn_count = -1f64 * ert.log2();
```

```
12
13          // 随机哈希函数
14          let hashers = [
15              RandomState::new().build_hasher(),
16              RandomState::new().build_hasher(),
17          ];
18
19          Self {
20              bits: vec![false; bits_count.ceil() as usize],
21              hash_fn_count: hash_fn_count.ceil() as usize,
22              hashers: hashers,
23              _phantom: PhantomData,
24          }
25      }
26
27      // 按照 hash_fn_count 计算值并置比特桶相应位为 true
28      fn insert(&mut self, elem: &T) {
29          let hashes = self.make_hash(elem);
30          for fn_i in 0..self.hash_fn_count {
31              let index = self.get_index(hashes, fn_i as u64);
32              self.bits[index] = true;
33          }
34      }
35
36      // 数据查询
37      fn contains(&self, elem: &T) -> bool {
38          let hashes = self.make_hash(elem);
39          (0..self.hash_fn_count).all(|fn_i| {
40              let index = self.get_index(hashes, fn_i as u64);
41              self.bits[index]
42          })
43      }
44
45      // 计算哈希
46      fn make_hash(&self, elem: &T) -> (u64, u64) {
47          let hasher1 = &mut self.hashers[0].clone();
48          let hasher2 = &mut self.hashers[1].clone();
49
```

```
50            elem.hash(hasher1);
51            elem.hash(hasher2);
52
53            (hasher1.finish(), hasher2.finish())
54        }
55
56        // 获取比特桶某位下标
57        fn get_index(&self, (h1,h2): (u64,u64), fn_i: u64)
58            -> usize {
59            let ih2 = fn_i.wrapping_mul(h2);
60            let h1pih2 = h1.wrapping_add(ih2);
61            ( h1pih2 % self.bits.len() as u64) as usize
62        }
63    }
64
65    fn main() {
66        let mut bf = BloomFilter::new(100, 0.08);
67        (0..20).for_each(|i| bf.insert(&i));
68        let res1 = bf.contains(&2);
69        let res2 = bf.contains(&200);
70        println!("2 in bf: {res1}, 200 in bf: {res2}");
71    }
```

分析布隆过滤器可以发现，其空间复杂度为 O(m)，插入 insert 和检测 contains 的时间复杂度为 O(k)，因为 k 非常小，所以可以看成 O(1)。

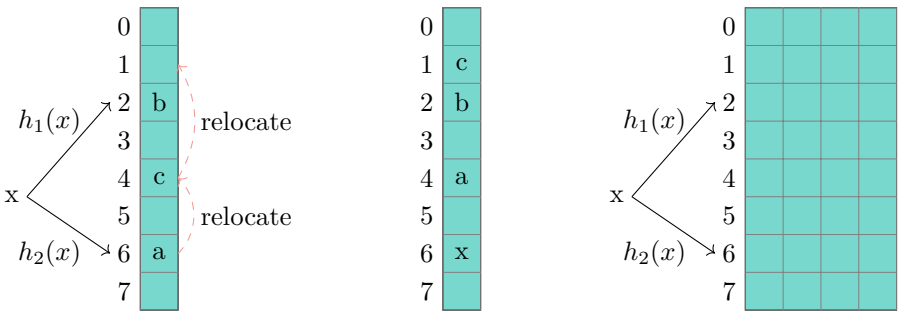
9.4.2 布谷鸟过滤器

前面实现的布隆过滤器容易实现，但也有许多缺点，第一是随着插入数据越多，误差率越来越大，第二是不能删除数据，最后一点是布隆过滤器随机存储，在具有 Cache 的 CPU 上性能不好，具体参见 CloudFlare 的博文《When Bloom filters don't bloom》。为解决布隆过滤器的缺点，布谷鸟过滤器 [13]（Cuckoo filter）应运而生。布谷鸟过滤器是改进的布隆过滤器，它的哈希函数是成对的，分别将数据映射到两个位置，一个是保存的位置，另一个是备用位置，用于处理碰撞。

布谷鸟过滤器名字来源于布谷鸟，也叫杜鹃，就是“庄生晓梦迷蝴蝶，望帝春心托杜鹃”里这个杜鹃。这种鸟有一种狡猾又贪婪的习性，它不自己筑巢，而是把蛋下到别种鸟的巢里，由别的鸟帮助孵化出自己的后代。它的幼鸟比别的鸟早出生，所以布谷幼鸟一出生就会拼命把未出生的其它鸟蛋挤出巢，以便今后独享养父母的食物，真真是鸠占鹊巢。借助生物学上的这一现象，布谷鸟过滤器处理碰撞的方法也是把原来位置上的元素踢走，不过被踢出去的

元素还比鸟蛋要幸运些，因为它还有一个备用位置可以安置，如果备用位置上还有人，再把它踢走，如此往复，直到被踢的次数达到一个上限，才确认哈希表已满。

布谷鸟过滤器里存储的元素不是 0 或 1，而是一定比特位的数据，又称为指纹。指纹长度由假阳性率 ϵ 决定，小的 ϵ 需要更长的指纹。布谷鸟过滤器基于布谷鸟哈希表，通过扩展为二维矩阵得到可以存储多个指纹的表，如下图。



布谷鸟哈希表插入 x 时，发现两个桶都有数据，则随机踢出 a 到 c，而后 c 移到最上面。布谷鸟过滤器则将桶扩展到 4 个，一个位置可以存储多个数据，支持插入，删除，查找。

在左侧图示的标准布谷鸟散列中，将新项插入到现有哈希表中需要方法来访问原始项，以便确定在需要时将其迁移并为新项腾出空间。然而，布谷鸟过滤器只存储指纹，因此没有办法重新散列原始项以找到其替代位置。为突破这个限制，可以利用一种称为部分键布谷鸟散列的技术来根据其指纹得到项的备用位置。对于项 x，通过散列函数计算两个候选桶的索引方式如下：

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \oplus hash(figureprint(x)) \end{aligned} \tag{9.6}$$

异或操作 $\oplus$ 确保 $h_1(x)$ 和 $h_2(x)$ 可以用同一个公式计算出来，这样就不用管 x 到底是什么，都可以用式（9.7）计算出备用桶的位置。

$$j = i \oplus hash(figureprint(x)) \tag{9.7}$$

查找方法很简单，利用式（9.6）计算出待查找元素的指纹和两个备用桶位置，然后读取两个桶，任何桶中有值与待查找元素的指纹相等则表示存在。删除方法也很简单，检查两个备用桶的值，如果有匹配的值，那么删除该桶中指纹的副本。同时要注意，删除前请确保插入了该项，否则可能把碰巧具有相同指纹的其他值删除了。

通过反复实验和测试，桶大小为 4 时性能非常优秀，甚至就是最佳值。布谷鸟过滤器具有以下四个主要优点：

（1）支持动态添加和删除项。

（2）比布隆过滤器更高的查找性能，即使当其接近满载。

（3）比其他的布隆过滤器诸如商过滤器等替代品更容易实现。

（4）在实际应用中，若假阳性率 ϵ 小于 3%，则其使用空间小于布隆过滤器。

表 9.1: 各种过滤器对比

过滤器类型	空间使用	哈希函数个数	删除功能
布隆过滤器	1	k	no
块布隆过滤器	1x	1	no
计数布隆过滤器	3x-4x	k	yes
d-left 计数布隆过滤器	1.5x-2x	d	yes
商数过滤器	1x-1.2x	≥ 1	yes
布谷鸟过滤器	$\leq 1x$	2	yes

下面来实现布谷鸟过滤器，前面虽然已经实现过布隆过滤器，但此处需要扩展到二维，所以要新增指纹结构体 FingerPrint，桶结构体 Bucket 用于存储指纹。因为涉及到随机获取，哈希操作等，所以代码中还使用了 Rng 和 Serde 等库。我们将 CuckooFilter 实现为一个 Rust 库（lib），其中 bucket.rs 包含指纹和桶的定义及操作，util.rs 包含计算指纹和桶索引的结构体 FaI，整个代码结构如下。

```
shieber@Kew:cuckoofilter/ tree
 .
 /Cargo.toml
 /src
    |- bucket.rs
    |- lib.rs
    |- util.rs

1 directory, 4 files
```

布谷鸟过滤器代码非常多，这里仅将 lib.rs 中列出，其他请参阅随书源码。

```
// lib.rs
mod bucket;
mod util;

use std::fmt;
use std::cmp::max;
use std::iter::repeat;
use std::error::Error;
use std::hash::{Hash, Hasher};
use std::marker::PhantomData;
use std::collections::hash_map::DefaultHasher;

```

```
13 // 序列化
14 use rand::Rng;
15 #[cfg(feature = "serde_support")]
16 use serde_derive::{Serialize, Deserialize};
17
18 use crate::util::FaI;
19 use crate::bucket::{Bucket, FingerPrint,
20                     BUCKET_SIZE, FIGERPRINT_SIZE};
21
22 const MAX_RELOCATION: u32 = 100;
23 const DEFAULT_CAPACITY: usize = (1 << 20) - 1;
24
25 // 错误处理
26 #[derive(Debug)]
27 enum CuckooError {
28     NotEnoughSpace,
29 }
30
31 // 添加打印输出功能
32 impl fmt::Display for CuckooError {
33     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
34         f.write_str("NotEnoughSpace")
35     }
36 }
37
38 impl Error for CuckooError {
39     fn description(&self) -> &str {
40         "Not enough space to save new element, operation failed!"
41     }
42 }
43
44 // 布谷鸟过滤器
45 struct CuckooFilter<H> {
46     buckets: Box<[Bucket]>,    // 桶
47     len: usize,                // 长度
48     _phantom: PhantomData<H>,
49 }
50
```

```
51 // 添加默认值功能
52 impl Default for CuckooFilter<DefaultHasher> {
53     fn default() -> Self {
54         Self::new()
55     }
56 }
57
58 impl CuckooFilter<DefaultHasher> {
59     fn new() -> Self {
60         Self::with_capacity(DEFAULT_CAPACITY)
61     }
62 }
63
64 impl<H: Hasher + Default> CuckooFilter<H> {
65     fn with_capacity(cap: usize) -> Self {
66         let capacity = max(1, cap.next_power_of_two()
67                               / BUCKET_SIZE);
68         Self {
69             buckets: repeat(Bucket::new()) // 构建 capacity 个 Bucket
70                     .take(capacity)
71                     .collect::<Vec<_>>(),
72                     .into_boxed_slice(),
73             len: 0,
74             _phantom: PhantomData,
75         }
76     }
77
78     fn try_insert<T: ?Sized + Hash>(&mut self, elem: &T)
79         -> Result<bool, CuckooError> {
80         if self.contains(elem) {
81             Ok(false)
82         } else {
83             self.insert(elem).map(|_| true)
84         }
85     }
86
87     fn insert<T: ?Sized + Hash>(&mut self, elem: &T)
88         -> Result<(), CuckooError> {
```

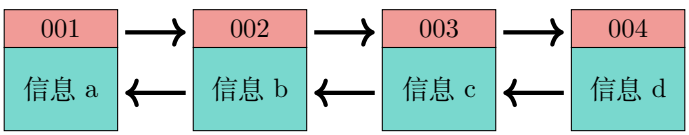
```
89          let fai = FaI::from_data::<_, H>(elem)
90          if self.put(fai.fp, fai.i1)
91              || self.put(fai.fp, fai.i2) {
92              return Ok(());
93          }
94
95          // 插入数据冲突, 重定位
96          let mut rng = rand::thread_rng();
97          let mut i = fai.random_index(&mut rng);
98          let mut fp = fai.fp;
99          for _ in 0..MAX_RELOCATION {
100             let other_fp;
101             {
102                 let loc = &mut self.buckets[i % self.len]
103                                 .buffer[rng.gen_range(0,
104                                         BUCKET_SIZE)];
105                 other_fp = *loc;
106                 *loc = fp;
107                 i = FaI::get_alt_index::<H>(other_fp, i);
108             }
109             if self.put(other_fp, i) {
110                 return Ok(());
111             }
112             fp = other_fp;
113         }
114
115         Err(CuckooError::NotEnoughSpace)
116     }
117
118     // 加入指纹
119     fn put(&mut self, fp: FingerPrint, i: usize) -> bool {
120         if self.buckets[i % self.len].insert(fp) {
121             self.len += 1;
122             true
123         } else {
124             false
125         }
126     }
```

```
127
128         fn remove(&mut self, fp: FingerPrint, i: usize) -> bool {
129             if self.buckets[i % self.len].delete(fp) {
130                 self.len -= 1;
131                 true
132             } else {
133                 false
134             }
135         }
136
137         fn contains<T: ?Sized + Hash>(&self, elem: &T) -> bool {
138             let FaI { fp, i1, i2 } = FaI::from_data::<_, H>(elem);
139             self.buckets[i1 % self.len]
140                 .get_fp_index(fp)
141                 .or_else(|| {
142                     self.buckets[i2 % self.len]
143                         .get_fp_index(fp)
144                  })
145                 .is_some()
146         }
147     }
```

从代码中可以看到，布谷鸟过滤器支持插入、删除、查询功能。

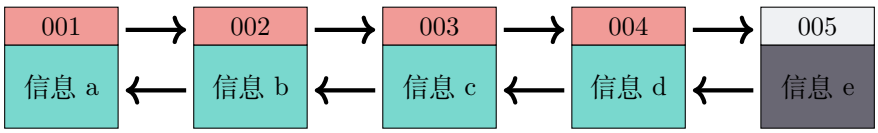
9.5 缓存淘汰算法 LRU

缓存淘汰算法或页面置换算法，是一种典型的内存管理算法，常用于虚拟页式存储，数据缓存。这种算法的原理是“如果数据最近被访问过，那么将来被访问的几率也更高”。对于在内存中但又不用的数据块，会根据哪些数据属于最近最少使用而将其移出内存，腾出空间。在这类淘汰算法中，LRU 很常用。LRU（Least recently used，最近最少使用）算法用于在存储有限的情况下，根据数据的访问记录来淘汰数据。假设使用哈希链表来缓存用户信息，容量为 5，目前缓存了 4 个用户信息，按时间顺序依次从右端插入。

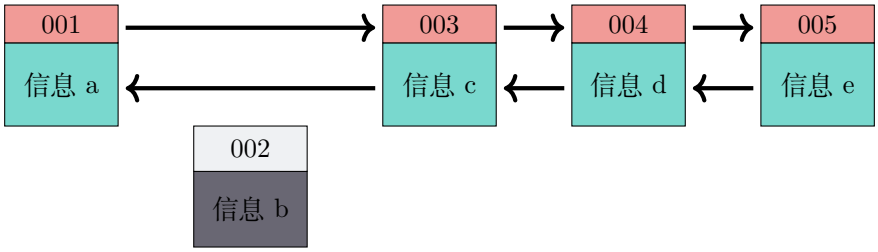


此时，业务方访问用户 5，由于哈希链表中没有用户 5 的数据，我们必须要从数据库中读取出来。为了后续访问方便，我们将其插入到缓存当中。这时候，链表中最右端是最新访

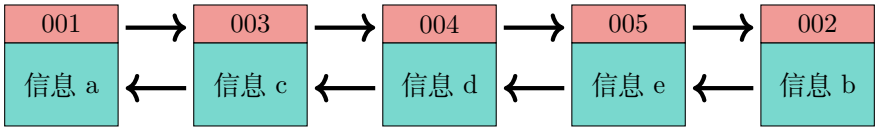
问到的用户 5，最左端是最近最少访问的用户 1。



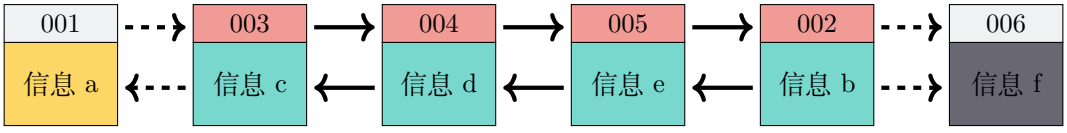
接下来，业务方访问用户 2，哈希链表中存在用户 2 的数据，我们怎么做呢？我们把用户 2 从它的前驱节点和后继节点之间移除，重新插入到链表最右端。这时候，链表中最右端变成了最新访问到的用户 2，最左端仍然是最近最少访问的用户 1。



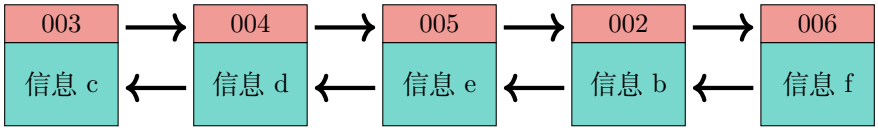
更新数据后，结果如下图。



后来业务方又访问了用户 6，而用户 6 在缓存里没有，需要插入到哈希链表。



但这时候缓存容量已达到上限，必须先删除最近最少访问的数据，那么位于哈希链表最左端的用户 1 就会被删除掉，然后再把用户 6 插入到最右端。



通过上述图示，相信你一定已经理解了 LRU 算法的原理。要实现它，就要从图中抽象出数据结构和操作来。基本上，LRU 需要管理插入数据的键（key），数据项（entry），前后指针。操作函数应当包含，插入（insert）、删除（remove）、查询（contains），此外还包含许多辅助函数。

如上分析，我们需要首先定义数据项和缓存的数据结构。我们用 HashMap 来存储键，用 Vec 来存储项，头尾指针则简化成了 Vec 中的下标。

```rust
// lru.rs

use std::collections::HashMap;

// LRU 上的元素项
struct Entry<K, V> {
    key: K,
    val: Option<V>,
    next: Option<usize>,
    prev: Option<usize>,
}

// LRU 缓存
struct LRUCache<K, V> {
    cap: usize,
    head: Option<usize>,
    tail: Option<usize>,
    map: HashMap<K, usize>,
    entries: Vec<Entry<K, V>>,
}
```

为了自定义缓存容量，我们将实现一个 with_capacity 函数，此外默认的 new 则将容量设置为 100。

```rust
use std::hash::Hash;
const CACHE_SIZE: usize = 100;

impl<K: Clone + Hash + Eq, V> LRUCache<K, V> {
    fn new() -> Self {
        Self::with_capacity(CACHE_SIZE)
    }

    fn with_capacity(cap: usize) -> Self {
        LRUCache {
            cap: cap,
            head: None,
            tail: None,
            map: HashMap::with_capacity(cap),
            entries: Vec::with_capacity(cap),
```

```
16            }
17        }
18    }
```

下面是插入函数，如果插入数据已存在，则直接更新值，那么插入新值后要将原始值返回。又因为插入值可能不存在，那么返回的原始值应该是 None，所以返回是 Option 类型。access 函数用于删除原始值并更新信息，ensure_room 用于在缓存达到容量时删除最少使用的数据。

```
1  // lru.rs
2
3  impl<K: Clone + Hash + Eq, V> LRUCache<K, V> {
4      fn insert(&mut self, key: K, val: V) -> Option<V> {
5          if self.map.contains_key(&key) { // 存在 key 就更新
6              self.access(&key);
7              let entry = &mut self.entries[self.head.unwrap()];
8              let old_val = entry.val.take();
9              entry.val = Some(val);
10             old_val
11         } else { // 不存在就插入
12             self.ensure_room();
13
14             // 更新原始头指针
15             let index = self.entries.len();
16             self.head.map(|e| {
17                 self.entries[e].prev = Some(index);
18             });
19
20             // 新的头结点
21             self.entries.push(Entry {
22                 key: key.clone(),
23                 val: Some(val),
24                 prev: None,
25                 next: self.head,
26             });
27             self.head = Some(index);
28             self.tail = self.tail.or(self.head);
29             self.map.insert(key, index);
30
```

```
31                  None
32              }
33          }
34
35          fn get(&mut self, key: &K) -> Option<&V> {
36              if self.contains(key) {
37                  self.access(key);
38              }
39
40              let entries = &self.entries;
41              self.map.get(key).and_then(move |i| {
42                  entries[*i].val.as_ref()
43              })
44          }
45
46          fn get_mut(&mut self, key: &K) -> Option<&mut V> {
47              if self.contains(key) {
48                  self.access(key);
49              }
50
51              let entries = &mut self.entries;
52              self.map.get(key).and_then(move |i| {
53                  entries[*i].val.as_mut()
54              })
55          }
56
57          fn contains(&mut self, key: &K) -> bool {
58              self.map.contains_key(key)
59          }
60
61          // 确保容量足够，满了就移除末尾的元素
62          fn ensure_room(&mut self) {
63              if self.cap == self.len() {
64                  self.remove_tail();
65              }
66          }
67
68          fn remove_tail(&mut self) {
```

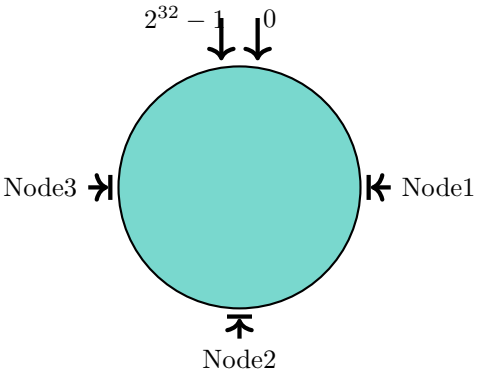
```
69          if let Some(index) = self.tail {
70              self.remove_from_list(index);
71              let key = &self.entries[index].key;
72              self.map.remove(key);
73          }
74          if self.tail.is_none() {
75              self.head = None;
76          }
77      }
78
79      // 获取某个 key 的值, 移除原来位置的值并在头部加入
80      fn access(&mut self, key: &K) {
81          let i = *self.map.get(key).unwrap();
82          self.remove_from_list(i);
83          self.head = Some(i);
84      }
85
86      fn remove(&mut self, key: &K) -> Option<V> {
87          self.map.remove(&key).map(|index| {
88                  self.remove_from_list(index);
89                  self.entries[index].val.take().unwrap()
90              })
91      }
92
93      fn remove_from_list(&mut self, i: usize) {
94          let (prev, next) = {
95              let entry = self.entries.get_mut(i).unwrap();
96              (entry.prev, entry.next)
97          };
98
99          match (prev, next) {
100             // 数据项在缓存中间
101             (Some(j), Some(k)) => {
102                 let head = &mut self.entries[j];
103                 head.next = next;
104                 let next = &mut self.entries[k];
105                 next.prev = prev;
106             },
```

```
107                 // 数据项在缓存末尾
108                 (Some(j), None) => {
109                     let head = &mut self.entries[j];
110                     head.next = None;
111                     self.tail = prev;
112                 },
113                 // 数据项在缓存头部
114                 _ => {
115                     if self.len() > 1 {
116                         let head = &mut self.entries[0];
117                         head.next = None;
118                         let next = &mut self.entries[1];
119                         next.prev = None;
120                     }
121                 },
122             }
123         }
124
125         fn len(&self) -> usize {
126             self.map.len()
127         }
128
129         fn is_empty(&self) -> bool {
130             self.map.is_empty()
131         }
132
133         fn is_full(&self) -> bool {
134             self.map.len() == self.cap
135         }
136     }
```

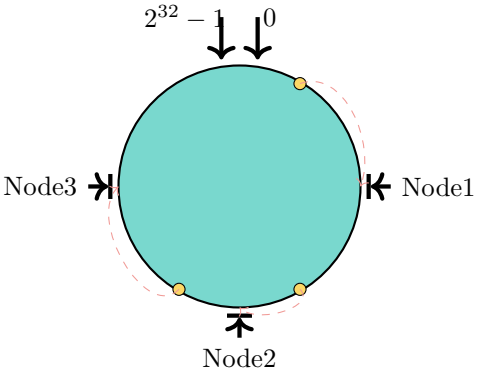
9.6 一致性哈希算法

假设用 Redis 来缓存图片，数据量小访问量也不大时，一台 Redis 就能搞定，最多用个主从就够了。然而数据量一旦变大，访问量也增加的时候，全部数据放在一台机器上不行，毕竟资源有限。这时候，往往会选择搭建集群，让数据分散存储到多台机器。比如 5 台机器，则图片对应的位置 index = hash(key) % 5。key 是和图片相关的某个指标。但若是要添加新机器或者有机器出现故障，那么 N 就会改变，上述计算的 index 就不对。一致性哈希算法的出

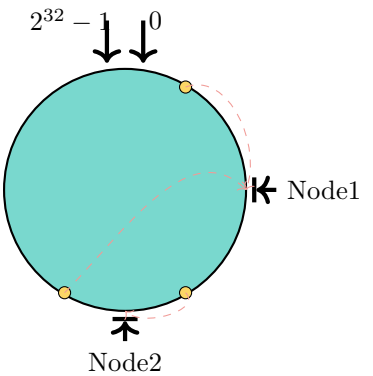
现就是为了解决这个问题，它以 0 为起点，在 $2^{32}-1$ 处停止，将这些点围成一个圆圈。让数据一定落在圆圈某个位置上。



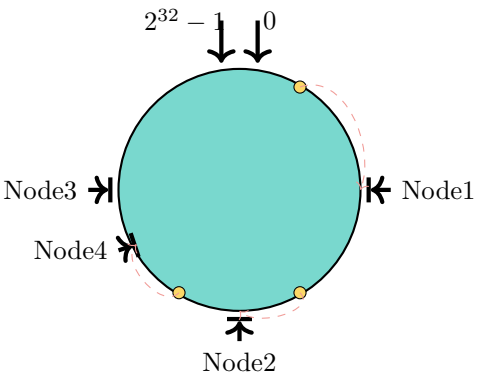
加入数据，则其哈希值会落在某段环上，将数据顺时针放到对应的结点可实现缓存。



现在假设结点 Node 3 宕机了，只有 Node2 到 Node3 间的数据项受到影响，它会转存到结点 Node1 上。



同样的，如果加入新的机器 Node 4，那么原本属于结点 Node 3 的数据会存储到 Node 4 上。



一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，有很好的容错性和可扩展性，这也是它能作到一致性的原因。

下面来考虑实现一致性哈希算法。如上分析，我们需要环（Ring）来存储结点，而结点（Node）代表机器，可以保存主机名（host），ip 和端口（port）为关键信息。

```
1  // conshash.rs
2  use std::fmt::Debug;
3  use std::clone::Clone;
4  use std::string::ToString;
5  use std::hash::{Hash, Hasher};
6  use std::collections::BTreeMap;
7  use std::collections::hash_map::DefaultHasher;
8
9  // 环上节点
10 #[derive(Clone, Debug)]
11 struct Node {
12     host: &'static str,
13     ip: &'static str,
14     port: u16,
15 }
16
17 // 为 Node 添加 to_string() 功能
18 impl ToString for Node {
19     fn to_string(&self) -> String {
20         format!("{}:{}", self.ip.to_string(),
21                          self.port.to_string())
22     }
23 }
24
```

```
// 环
struct Ring<T: Clone + ToString + Debug> {
    replicas: usize,          // 分区数
    ring: BTreeMap<u64, T>, // 保存数据的环
}
```

Ring 中的 replicas 是为防止结点聚集而导致数据也集中存储到少量结点上。对一个结点产生多个虚拟结点，那么这些结点会更均匀的分布到环上，就能解决结点聚集问题。

哈希计算可以采用标准库提供的默认哈希计算器，默认的结点是 10 个，可自定义创建结点数。对一致性哈希算法，我们至少还需要支持插入结点、删除结点、查询功能。当然，为了批量处理，插入和删除都可以实现批量处理版本。

```
const DEFAULT_REPLICAS: usize = 10;

// 哈希计算函数
fn hash<T: Hash>(val: &T) -> u64 {
    let mut hasher = DefaultHasher::new();
    val.hash(&mut hasher);
    hasher.finish()
}

impl<T> Ring<T> where T: Clone + ToString + Debug {
    fn new() -> Self {
        Self::with_capacity(DEFAULT_REPLICAS)
    }

    fn with_capacity(replicas: usize) -> Self {
        Ring {
            replicas: replicas,
            ring: BTreeMap::new(),
        }
    }

    // 批量插入结点
    fn add_multi(&mut self, nodes: &[T]) {
        if !nodes.is_empty() {
            for node in nodes.iter() { self.add(node); }
        }
    }
```

```
28
29      fn add(&mut self, node: &T) {
30          for i in 0..self.replicas {
31              let key = hash(&format!("{}{}", node.to_string(),
32                                                  i.to_string()));
33              self.ring.insert(key, node.clone());
34          }
35      }
36
37      // 批量删除结点
38      fn remove_multi(&mut self, nodes: &[T]) {
39          if !nodes.is_empty() {
40              for node in nodes.iter() { self.remove(node); }
41          }
42      }
43
44      fn remove(&mut self, node: &T) {
45          assert!(!self.ring.is_empty());
46          for i in 0..self.replicas {
47              let key = hash(&format!("{}{}", node.to_string(),
48                                                  i.to_string()));
49              self.ring.remove(&key);
50          }
51      }
52
53      // 查询结点
54      fn get(&self, key: u64) -> Option<&T> {
55          if self.ring.is_empty() { return None; }
56          let mut keys = self.ring.keys();
57          keys.find(|&k| k >= &key)
58              .and_then(|k| self.ring.get(k))
59              .or(keys.nth(0).and_then(|x| self.ring.get(x)))
60      }
61  }
62
63  fn main() {
64      let replica = 3;
65      let mut ring = Ring::with_capacity(replica);
```

```
66
67      let node = Node{ host: "localhost", ip: "127.0.0.1",port: 23 };
68      ring.add(&node);
69
70      for i in 0..replica {
71          let key = hash(&format!("{}{}", node.to_string(),
72                                          i.to_string()));
73          let res = ring.get(key);
74          assert_eq!(node.host, res.unwrap().host);
75      }
76
77      println!("{:?}", &node);
78      ring.remove(&node);
79  }
```

9.7 Base58 编码

Base58 和 Base64 一样，是一种编码算法。用于表示比特币钱包地址，由中本聪引入。Base58 在 Base64 的基础上删除了易引起歧义的（表9.2中红色）字符，包括 0（零）、O（大写 O）、I（大写 i）、l（小写 L），以及 + 和 / 字符，剩下的 58 个字符作为编码字符。这些字符既不容易认错，又避免了 / 等字符在复制时断行的问题。

表 9.2: Base 64 编码字符

编号	字符	编号	字符	编号	字符	编号	字符	编号	字符
0	0	13	D	26	Q	39	d	52	q
1	1	14	E	27	R	40	e	53	r
2	2	15	F	28	S	41	f	54	s
3	3	16	G	29	T	42	g	55	t
4	4	17	H	30	U	43	h	56	u
5	5	18	I	31	V	44	i	57	v
6	6	19	J	32	W	45	j	58	w
7	7	20	K	33	X	46	k	59	x
8	8	21	L	34	Y	47	l	60	y
9	9	22	M	35	Z	48	m	61	z
10	A	23	N	36	a	49	n	62	+
11	B	24	O	37	b	50	o	63	/
12	C	25	P	38	c	51	p		

Base58 的编码其实是大数进制转换，先将字符转换为 ASCII，然后转换为 10 进制，接着是 58 进制，最后按照编码表选择对应字符组成 Base58 编码字符串。因为涉及数的进制转换，所以效率比较低，其编码原理如算法（9.1）。

Algorithm 9.1: Base58 编码流程

Data: 原始字符串 s
Result: 编码后字符串 b58
初始化一个空字符串 b58 用于保存结果
for $c \in s$ **do**
    将 s 中字节 c 转换成 ASCII 值（256 进制）
    将 256 进制数字转换成 10 进制数字
    将 10 进制数字转换成 58 进制数字
    将 58 进制数字按照 Base58 字符表转换成对应字符
    将得到的字符加入 b58
end
返回编码后的字符串 b58

解码是个逆过程，同样的也是大数的进制间转换，先将其中 Base58 字符串中字符转换为 ASCII 值，然后再转到 10 进制，接着转到 256 进制，最后再转到 ASCII 字符。具体解码原理如算法（9.2）。

Algorithm 9.2: Base58 解码流程

Data: 编码后字符串 b58s
Result: 解码后字符串 s
初始化一个空字符串 s 用于保存结果
for $c \in b58$ **do**
    将 b58 中字节 c 转换成 ASCII 值（58 进制）
    将 58 进制数字转换成 10 进制数字
    将 10 进制数字转换成 256 进制数字
    将 256 进制数字按照 ASCII 表转换成对应字符
    将得到的字符加入 s
end
返回解码后的字符串 s

其实编码和解码就是两个空间的字符串转换，实际上更像是编码空间的映射。知道了 Base58 的编解码原理，下面来实现一个简易的 Base58 编解码器。首先是准备 Base58 的编码字符 ALPHABET 和编码转换表 BASE58_DIGITS_MAP。

```
1 // base58.rs
```

```
2
3  // base58 变码字符
4  const ALPHABET: &[u8;58] = b"123456789ABCDEFGHJKLMNPQRSTUVWXYZ
5  abcdefghijkmnopqrstuvwxyz";
6
7  // 进制映射关系
8  const BASE58_DIGITS_MAP: &'static [i8] = &[
9      -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
10     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
11     -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,
12     -1, 0, 1, 2, 3, 4, 5, 6, 7, 8,-1,-1,-1,-1,-1,-1,
13     -1, 9,10,11,12,13,14,15,16,-1,17,18,19,20,21,-1,
14     22,23,24,25,26,27,28,29,30,31,32,-1,-1,-1,-1,-1,
15     -1,33,34,35,36,37,38,39,40,41,42,43,-1,44,45,46,
16     47,48,49,50,51,52,53,54,55,56,57,-1,-1,-1,-1,-1,
17 ];
```

为了应对编解码可能出现的错误，我们为 Base58 编码实现了自定义的错误类型，用于处理字符非法和长度错误及其他情况。编码和解码我们实现成 str 类型的两个 trait：Encoder，Decoder，两者分别含有 encode 和 decode 方法，返回 String 和 Result<String, Err>。

```
1  // base58.rs
2
3  // 解码错误类新
4  #[derive(Debug, PartialEq)]
5  pub enum DecodeError {
6      Invalid,
7      InvalidLength,
8      InvalidCharacter(char, usize),
9  }
10
11 // 编解码 trait
12 pub trait Encoder {
13     fn encode(&self) -> String;
14 }
15
16 pub trait Decoder {
17     fn decode(&self) -> Result<String, DecodeError>;
18 }
```

定义了 trait，接下来就是分别实现 encode 和 decode 方法，具体原理如前所述。此处的 trait 是为 str 实现的，但内部计算用 u8 比较好，因为字符串中的字符可能包含多个 u8。

```
// base58.rs

// 实现 base58 编码
impl Encoder for str {
    fn encode(&self) -> String {
        let str_u8 = self.as_bytes();
        let zero_count = str_u8.iter()
                               .take_while(|&&x| x == 0)
                               .count();
        let size = (str_u8.len() - zero_count) * 138 / 100 + 1;

        // 字符进制转换
        let mut i = zero_count;
        let mut high = size - 1;
        let mut buffer = vec![0u8; size];
        while i < str_u8.len() {
            let mut j = size - 1;
            let mut carry = str_u8[i] as u32;

            while j > high || carry != 0 {
                carry += 256 * buffer[j] as u32;
                buffer[j] = (carry % 58) as u8;
                carry /= 58;

                if j  > 0 { j -= 1; }
            }

            i += 1;
            high = j;
        }

        // 处理多个前置 0
        let mut base58_str = String::new();
        for _ in 0..zero_count { // 处理多个前置 0
            base58_str.push('1');
        }
```

```
37
38          // 获取编码后的字符并拼接成字符串
39          let mut j = buffer.iter()
40                              .take_while(|x| **x == 0)
41                              .count();
42          while j < size {
43              base58_str.push(ALPHABET[buffer[j] as usize]
44                              as char);
45              j += 1;
46          }
47
48          base58_str
49      }
50  }
```

解码就是将 Base58 编码逆编码的过程，具体如下。

```
1   // base58.rs
2
3   // 实现 base58 解码
4   impl Decoder for str {
5       fn decode(&self) -> Result<String, DecodeError> {
6           let mut bin = [0u8; 132];
7           let mut out = [0u32; (132 + 3) / 4];
8           let bytesleft = (bin.len() % 4) as u8;
9           let zeromask = match bytesleft {
10              0 => 0u32,
11              _ => 0xffffffff << (bytesleft * 8),
12          };
13
14          let zero_count = self.chars()
15                              .take_while(|&x| x == '1')
16                              .count();
17          let mut i = zero_count;
18          let b58: Vec<u8> = self.bytes().collect();
19          while i < self.len() {
20              if (b58[i] & 0x80) != 0 {
21                  return Err(DecodeError::InvalidCharacter(
22                              b58[i] as char, i));
```

```
23              }
24
25              if BASE58_DIGITS_MAP[b58[i] as usize] == -1 {
26                  return Err(DecodeError::InvalidCharacter(
27                              b58[i] as char, i));
28              }
29
30              let mut j = out.len();
31              let mut c = BASE58_DIGITS_MAP[b58[i] as usize]
32                          as u64;
33              while j != 0 {
34                  j -= 1;
35                  let t = out[j] as u64 * 58 + c;
36                  c = (t & 0x3f00000000) >> 32;
37                  out[j] = (t & 0xffffffff) as u32;
38              }
39
40              if c != 0 {
41                  return Err(DecodeError::InvalidLength);
42              }
43
44              if (out[0] & zeromask) != 0 {
45                  return Err(DecodeError::InvalidLength);
46              }
47
48              i += 1;
49          }
50
51          let mut i = 1;
52          let mut j = 0;
53          bin[0] = match bytesleft {
54              3 => ((out[0] & 0xff0000) >> 16) as u8,
55              2 => ((out[0] & 0xff00) >> 8) as u8,
56              1 => {
57                  j = 1;
58                  (out[0] & 0xff) as u8
59              },
60              _ => {
```

```
61                  i = 0;
62                  bin[0]
63              }
64          };
65
66          while j < out.len() {
67              bin[i] = ((out[j] >> 0x18) & 0xff) as u8;
68              bin[i + 1] = ((out[j] >> 0x10) & 0xff) as u8;
69              bin[i + 2] = ((out[j] >> 8) & 0xff) as u8;
70              bin[i + 3] = ((out[j] >> 0) & 0xff) as u8;
71              i += 4;
72              j += 1;
73          }
74
75          let leading_zeros = bin.iter()
76                                  .take_while(|x| **x == 0)
77                                  .count();
78          let new_str = String::from_utf8(
79                                   bin[leading_zeros - zero_count..]
80                                   .to_vec());
81          match new_str {
82              Ok(res) => Ok(res),
83              Err(_) => Err(DecodeError::Invalid),
84          }
85      }
86  }
87
88  fn main() {
89      println!("{:#?}","abc".encode());
90      println!("{:#?}","ZiCa".decode().unwrap());
91      println!("{:#?}","我爱你iloveu".encode());
92      println!("{:#?}","7T5VrPqoBr9DeUXiUr2Fn".decode().unwrap());
93  }
```

至此，整个 Base58 算法就完成了。同样的，实现 Base32、Base36、Base62、Base64、Base85、Base92 等编解码算法也是用类似的方法，读者可自行思考并实现感兴趣的编码算法。本书之所以实现 Base58，是因为它在数字货币中使用非常普遍，这和下面要讲的区块链紧密相关。

9.8 区块链

互联网上的贸易，几乎都需要借助金融机构作为可资信赖的第三方来处理电子支付信息。虽然这类系统在绝大多数情况下都运作良好，但是这类系统仍然内生性地受制于“基于信用的模式”的弱点。我们无法实现完全不可逆的交易，因为金融机构总是不可避免地会出面协调争端。而金融中介的存在，也会增加交易的成本，并且限制了实际可行的最小交易规模，也限制了日常的小额支付交易。并且潜在的损失还在于，很多商品和服务本身是无法退货的，如果缺乏不可逆的支付手段，互联网的贸易就大大受限。因为有潜在的退款的可能，就需要交易双方拥有信任。而商家也必须提防自己的客户，因此会向客户索取完全不必要的个人信息。而实际的商业行为中，一定比例的欺诈性客户也被认为是不可避免的，相关损失计入了销售费用里。而在使用物理现金的情况下，这些销售费用和支付问题上的不确定性却是可以避免的，因为此时没有第三方信用中介的存在。所以，我们非常需要这样一种电子支付系统，它基于密码学原理而不基于信用，使得任何达成一致的双方，能够直接进行支付，从而不需要第三方中介的参与。杜绝回滚支付交易的可能，就可以保护特定的卖家免于欺诈；而对于想要保护买家的人来说，在此环境下设立通常的第三方担保机制也可谓轻松加愉快。在这篇论文中，我们将提出一种通过点对点分布式的时间戳服务器来生成依照时间前后排列并加以记录的电子交易证明，从而解决双重支付问题。只要诚实的节点所控制的计算能力的总和大于有合作关系的攻击者的计算能力的总和，该系统就是安全的

上面的这段话是比特币发明人中本聪在比特币白皮书 [14]《比特币：一种点对点电子现金系统》中的介绍，这回答了比特币发明的原因。其实更实际的问题是 2008 年全球陷入金融危机，通货膨胀，各国都遭到严重冲击。中本聪对这样的金融环境不满，他相信未来国家退出发行货币角色就可以规避通货膨胀。在此基础上，他结合自己的专业知识发明了比特币及区块链，其完整概念载于 10 页白皮书。

9.8.1 区块链及比特币原理

区块链和比特币是什么关系呢？近些年，媒体对区块链和比特币报到很多，但多是宏观的叙述，没有技术细节。其实区块链技术是利用链式数据结构来验证与存储数据、利用分布式节点共识算法来生成和更新数据、利用密码学来保证数据传输和访问安全、利用自动化脚本组成的智能合约来编程和操作数据的一种全新的分布式基础架构与计算范式。

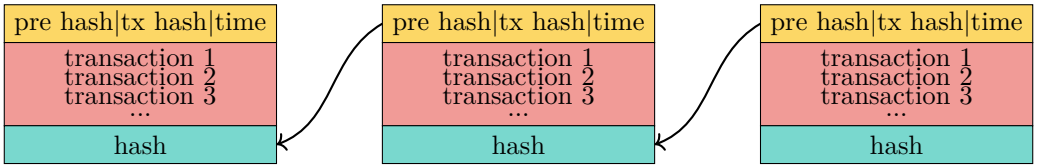
简单来说，区块链就是去中心化的分布式账本。去中心化，就是没有中心，或者说每个人都可以是中心，这是和传统的中心化方式不同的地方。分布式账本，意味着数据的存储不只是在每一个节点上，而是每一个节点会复制并共享整个账本的数据。

区块链是记录交易的账本，而记录交易是非常耗费资源的，所以在记录交易（打包）的过程中产生了奖励和手续费，这种奖励和手续费是一种数字货币，用于维持系统的运行。中本聪发明的区块链中的数字货币就是大名鼎鼎的比特币，它也是第一种数字货币。

所以可以看出，区块链是一个分布式的交易媒介，比特币交易的保障，是一种副产品，一种激励。区块链作为一个系统，它本身存在区块、区块链、交易、账户、矿工、手续费、奖励等组件。要实现区块链系统，就要从这些基本的组件开始，逐一实现。

9.8.2 基础区块链

一个简单的区块包含区块头、区块体、区块哈希。其中头包括前一个区块的哈希(pre_hash)、当前区块交易哈希（tx_hash)、区块打包时间（time)；区块体包含所有交易（transactions)；区块哈希（hash）是计算区块头和区块体得到的哈希值。区块及区块链结构如下图。



从上面的结构可以看出，哈希值是非常重要的，所以第一项工作，我们首先来实现哈希计算。一般来说，计算之前先将区块结构序列化，再计算哈希值更高效。

我们的基础区块链的第一个功能是实现序列化和哈希值计算，具体如下面的代码。?Sized 是为了处理不定大小的区块，因为交易可能多也可能少，数量不一。bincode 用于序列化，crypto 的 Sha3 用于计算哈希。为了方便查看，我们将所有哈希全转换成字符串。serialize 序列化后的数据是 &[u8] 类型，而 hash_str 获取该类型数据并返回字符串。这样我们就完成了数据结构序列化和哈希。

```
// serializer.rs
use bincode;
use serde::Serialize;
use crypto::digest::Digest;
use crypto::sha3::Sha3;

// 序列化数据
pub fn serialize<T: ?Sized>(value: &T) -> Vec<u8>
    where T: Serialize {
    bincode::serialize(value).unwrap()
}

// 计算 value 哈希值并以 String 形式返回
pub fn hash_str(value: &[u8]) -> String {
    let mut hasher = Sha3::sha3_256();
    hasher.input(value);
    hasher.result_str()
}
```

有了计算哈希的函数，区块里的 hash，pre_hash，tx_hash 就都能计算了，时间则可采用生成区块时的时间，只有交易 transaction 不定。为了简化问题，最开始就用字符串来模拟交易，通过将其放入 Vec 中来表示多笔交易。Rust 中可以用 struct 来表示区块和区块头。

```
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: String,
    pub hash: String,
}

pub struct BlockHeader {
    pub time: i64,
    pub pre_hash: String,
    pub txs_hash: String,
}
```

对于每个区块，首先要能新建。新建后，还需要更新区块的哈希值，区块的实现如下。

```
// block.rs
use std::thread;
use std::time::Duration;
use chrono::prelude::*;
use utils::serializer::{serialize, hash_str};
use serde::Serialize;

// 区块头
#[derive(Serialize, Debug, PartialEq, Eq)]
pub struct BlockHeader {
    pub time: i64,
    pub pre_hash: String,
    pub txs_hash: String,
}
// 区块
#[derive(Debug)]
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: String,
    pub hash: String,
}

impl Block {
    pub fn new(txs: String, pre_hash: String) -> Self {
        // 用延迟 3 秒来莫以挖矿
```

```
26          println!("Start mining .... ");
27          thread::sleep(Duration::from_secs(3));
28
29          // 准备时间、计算交易哈希值
30          let time = Utc::now().timestamp();
31          let txs_ser = serialize(&txs);
32          let txs_hash = hash_str(&txs_ser);
33          let mut block = Block {
34              header: BlockHeader {
35                  time: time,
36                  txs_hash: txs_hash,
37                  pre_hash: pre_hash,
38              },
39              tranxs: txs,
40              hash: "".to_string(),
41          };
42          block.set_hash();
43          println!("Produce a new block!\n");
44          block
45      }
46
47      // 计算并设置区块哈希值
48      fn set_hash(&mut self) {
49          let header = serialize(&(self.header));
50          self.hash = hash_str(&header);
51      }
52  }
```

有了区块，接下来就是链了。一条链需要保存多个区块，可以用 Vec 来存储。此外还要能产生第一个区块（创世区块）以及添加新区块。第一个区块没有 pre_hash，所以需要手动设置一个，我选择的是"Bitcoin hit $60000" 的 base64 值作为创世区块的 pre_hash。

```
1  // blockchain.rs
2  use crate::block::Block;
3
4  // 创世区块 pre_hash
5  const PRE_HASH: &str = "22caaf24ef0aea3522c13d133912d2b7
6                          22caaf24ef0aea3522c13d133912d2b7";
7  pub struct BlockChain {
```

```
8       pub blocks: Vec<Block>,
9   }
10
11  impl BlockChain {
12      pub fn new() -> Self {
13          BlockChain { blocks: vec![Self::genesis_block()] }
14      }
15
16      // 生成创世区块
17      fn genesis_block() -> Block {
18          Block::new("创始区块".to_string(),PRE_HASH.to_string())
19      }
20
21      // 添加区块，形成区块链
22      pub fn add_block(&mut self, data: String) {
23          // 获取前一个区块的哈希值
24          let pre_block = &self.blocks[self.blocks.len() - 1];
25          let pre_hash  = pre_block.hash.clone();
26          // 构建新区块并加入链
27          let new_block = Block::new(data, pre_hash);
28          self.blocks.push(new_block);
29      }
30
31      // 打印区块信息
32      pub fn block_info(&self) {
33          for b in self.blocks.iter() { println!("{:#?}", b); }
34      }
35  }
```

为了运行区块链，我们需要构造交易，用于生成区块。下面的 main 文件里采用字符串代表交易 tx，并在打包交易及结束后分别打印出信息。

```
1   // main.rs
2   use core::blockchain::BlockChain as BC;
3
4   fn main() {
5       println!("----------------Mine Info--------------------");
6
7       let mut bc = BC::new();
```

```
8         let tx = "0xabcd -> 0xabce: 5 btc".to_string();
9         bc.add_block(tx);
10        let tx = "0xabcd -> 0xabcf: 2.5 btc".to_string();
11        bc.add_block(String::from(tx));
12
13        println!("------------------Block Info--------------------");
14        bc.block_info();
15    }
```

这些代码要按照逻辑将其组织起来才能很好的工作，哈希计算放到 utils 下当工具使用，因为它本身和区块链没有关系，而 block 和 blockchain 需要放到 core 目录下，main 用来调用 core，实现区块的新建和添加。可以用 Cargo 来生成项目 blockchain，具体参见 github 上的 blockchain1 仓库或 gitee 上的 blockchain1 仓库。通过上述代码，我们实现了一个最基本的区块链项目，它能新建及添加区块，下面是运行的结果，包含挖矿信息、区块信息。

```
---------------------------------Mine Info---------------------------------------
Start mining ...
Produced a new block!

Start mining ...
Produced a new block!

Start mining ...
Produced a new block!
---------------------------------Block Info---------------------------------------
Block {
    header: BlockHeader {
        time: 1619011220,
        txs_hash: "b868068f9515f7f89a2a0d691508fb380af41166fd4834fee4969bed33b38839",
        pre_hash: "22caaf24ef0aea3522c13d133912d2b722caaf24ef0aea3522c13d133912d2b7",
    },
    tranxs: "创世区块",
    hash: "1215955b17955d31bbda7ba638d7ca240e3431d06fb0bb1a4f06fa21e6bf3ac5",
}
Block {
    header: BlockHeader {
        time: 1619011223,
        txs_hash: "84eeeb7be34240b4a5c45534fc0951ec8f375d93cd3a77d2f154fbdfdde080c1",
        pre_hash: "1215955b17955d31bbda7ba638d7ca240e3431d06fb0bb1a4f06fa21e6bf3ac5",
    },
    tranxs: "0xabcd -> 0xabce: 5 btc",
    hash: "9defaba787ac4034ac37fa516b2e9b5a325901585d198a73827be9036f2f18d2",
}
Block {
    header: BlockHeader {
        time: 1619011226,
        txs_hash: "ca51ee57941a2af26ae2fa02d05f6276f6c2c050535314d6393501b797b13438",
        pre_hash: "9defaba787ac4034ac37fa516b2e9b5a325901585d198a73827be9036f2f18d2",
    },
    tranxs: "0xabcd -> 0xabcf: 2.5 btc",
    hash: "80d05dfffaf6bf476651604de38f4f880013372b82f3286375abb5526e9523a1",
}
```

9.8.3 工作量证明

很明显，我们的区块链中并没有计算区块的函数，都是直接生成区块，这和实际中比特币十分钟才能产生一个是不同的。区块链系统通过维持一个计算任务的难度，使得产生区块稳定有序。计算任务又称工作量证明，通过要求矿工对区块计算一个哈希值并满足某个难度要求，若通过则加入链，并给予奖励。

我们将计算任务实现在 core 目录下的 pow.rs 里面。因为计算的哈希值往往较大，所以采用 U256 来表示。每打包一个区块就是要构造一个工作量任务 ProofOfWork。new 通过计算当前难度得到一个目标值，run 则计算值并和这个目标值比较，若不符合，则改变 nonce 值（加一），再计算哈希值，如此往复。

```
1  // pow.rs
2
3  use std::thread;
4  use std::time::Duration;
5  use crate::block::Block;
6  use utils::serializer::{serialize, hash_str, hash_u8};
7  use bigint::U256;
8
9  // nonce 最大值
10 const MAX_NONCE: u32 = 0x7FFFFFFF;
11
12 // 工作量证明任务
13 pub struct ProofOfWork {
14     target: U256,
15 }
16
17 impl ProofOfWork {
18     // 计算当前任务难度值
19     pub fn new(bits: u32) -> Self {
20         let (mant, expt) = {
21             let unshifted_expt = bits >> 24;
22             if unshifted_expt <= 3 {
23                 ((bits & 0xFFFFFF) >>
24                     (8 * (3-unshifted_expt as usize)), 0)
25             } else {
26                 (bits & 0xFFFFFF, 8 * ((bits >> 24) - 3))
27             }
28         };
29
```

```
30          if mant > 0x7FFFFF {
31              Self {
32                  target: Default::default(),
33              }
34          } else {
35              Self {
36                target:U256::from(mant as u64)<<(expt as usize),
37              }
38          }
39      }
40
41      // 开启工作量证明任务，即挖矿
42      pub fn run(&self, mut block: &mut Block) {
43          println!("Start mining .... ");
44          thread::sleep(Duration::from_secs(3));
45
46          let mut nonce: u32 = 0;
47          while nonce <= MAX_NONCE {
48              // 计算值
49              let hd_ser = Self::prepare_data(&mut block, nonce);
50              let mut hash_u: [u8; 32] = [0; 32];
51              hash_u8(&hd_ser, &mut hash_u);
52
53              // 判断值是否满足要求，满足则计算并设置区块哈希值
54              let hash_int = U256::from(hash_u);
55              if hash_int <= self.target {
56                  block.hash = hash_str(&hd_ser);
57                  println!("Produce a new block!\n");
58                  return;
59              }
60
61              nonce += 1;
62          }
63      }
64
65      // 准备区块头数据
66      fn prepare_data(block: &mut Block, nonce: u32) ->Vec<u8> {
67          block.header.nonce = nonce;
```

```
68          serialize(&(block.header))
69      }
70  }
```

因为 U256 类型是非常大的数字，所以需要从 [u8] 类型数据生成，这是使用 hash_u8 的原因。自然的，utils 下的 serializer 中需要实现 hash_u8 的函数。

```
1  // serializer.rs
2  // 省略前面内容 ....
3
4  // 计算 value 哈希值并传递给 out 参数
5  pub fn hash_u8(value: &[u8], mut out: &mut [u8]) {
6      let mut hasher = Sha3::sha3_256();
7      hasher.input(value);
8      hasher.result(&mut out);
9  }
```

为了使用 ProofOfWork，blockchain.rs 中相应代码也需要修改，其结果如下。下面只列出了新增和改变的内容。

```
1  // block.rs
2  // ...
3  use crate::pow::ProofOfWork;
4
5  #[derive(Serialize, Debug, PartialEq, Eq)]
6  pub struct BlockHeader {
7      pub nonce: u32,
8      pub bits: u32,
9      pub time: i64,
10     pub txs_hash: String,
11     pub pre_hash: String,
12 }
13
14 impl Block {
15     pub fn new(txs:String, pre_hash:String, bits:u32) ->Self {
16         // ....
17         let mut block = Block {
18             header: BlockHeader {
19                 time: time,
20                 txs_hash: txs_hash,
21                 pre_hash: pre_hash,
```

```
                bits: bits,
                nonce: 0,
            },
            tranxs: txs,
            hash: "".to_string(),
        };

        // 初始化挖矿任务并开始挖矿
        let pow = ProofOfWork::new(bits);
        pow.run(&mut block);

        block
    }
}
```

可以发现，此区块链项目是在基础区块链上逐个添加功能得到的。添加了工作量证明机制的完整区块链项目在 github 上的 blockchain2 仓库或 gitee 上的 blockchain2仓库。

9.8.4 区块链存储

前面我们实现了区块链，但每次运行后，上一次的链就没有了，所以可以考虑将链保存下来。保存链有很多种办法，这里我们选择 LevelDB [15] 来保存，当然用 RocketDB [16] 也可以。

LevelDB 是由谷歌的 Jeff Dean 和 Sanjay Ghemawat 开发并开源的键值存储数据库。为使用此数据库，我们还需要实现键的定义，具体可以实现到 utils 中。

```
// bkey.rs

use bigint::U256;
use db_key::Key;
use std::mem::transmute;

// 定义大 Key
#[derive(Debug, Copy, Clone, Eq, PartialEq, Ord, PartialOrd)]
pub struct BKey {
    pub val: U256,
}

impl Key for BKey {
    fn as_slice<T, F: Fn(&[u8]) -> T>(&self, func: F) -> T {
```

```
15          let val = unsafe {
16              transmute::<_, &[u8; 32]>(self)
17          };
18          func(val)
19      }
20
21      fn from_u8(key: &[u8]) -> Self {
22          assert!(key.len() == 32);
23          let mut res: [u8; 32] = [0; 32];
24
25          for (i, val) in key.iter().enumerate() {
26              res[i] = *val;
27          }
28
29          unsafe {
30              transmute::<[u8; 32], Self>(res)
31          }
32      }
33  }
```

而存储则作为核心功能放到 core 下面。

```
1   // bcdb.rs
2
3   use leveldb::kv::KV;
4   use leveldb::database::Database;
5   use leveldb::options::{Options, WriteOptions};
6   use utils::bkey;
7   use std::{env, fs};
8
9   pub struct BlockChainDb;
10
11  impl BlockChainDb {
12      // 新建并返回数据库
13      pub fn new(path: &str) -> Database<bkey::BKey> {
14          let mut dir = env::current_dir().unwrap();
15          dir.push(path);
16
17          let path_buf = dir.clone();
```

```
18          fs::create_dir_all(dir).unwrap();
19
20          let path = path_buf.as_path();
21          let mut opts = Options::new();
22          opts.create_if_missing = true;
23
24          let database = match Database::open(path, opts) {
25              Ok(db) => db,
26              Err(e) => panic!("Failed to open db: {:?}", e),
27          };
28
29          database
30      }
31
32      // 数据写入数据库
33      pub fn write_db(db: &mut Database<bkey::BKey>,
34                      key: bkey::BKey, val: &[u8]) {
35          let write_opts = WriteOptions::new();
36          match db.put(write_opts, key, &val) {
37              Ok(_) => (),
38              Err(e) => panic!("Failed to write block: {:?}", e),
39          }
40      }
41  }
```

这样，在 blockchain.rs 里就可以使用 bcdb.rs 中的存储函数来存储区块链。

```
1  // blockchain.rs
2  // ...
3  use utils::bkey::BKey;
4  use leveldb::database::Database;
5
6  const SAVE_DIR: &str = "bc_db";
7
8  pub struct BlockChain {
9      pub blocks: Vec<Block>,
10     curr_bits: u32,
11     blocks_db: Box<Database<BKey>>,
12 }
```

```
13
14 impl BlockChain {
15     pub fn new() -> Self {
16         let mut db = BlockChainDb::new(SAVE_DIR);
17         let genesis = Self::genesis_block();
18         BlockChain::write_block(&mut db, &genesis);
19         BlockChain::write_tail(&mut db, &genesis);
20         println!("New produced block saved!\n");
21
22         BlockChain {
23             blocks: vec![genesis],
24             curr_bits: CURR_BITS,
25             blocks_db: Box::new(db),
26         }
27     }
28
29     fn genesis_block() -> Block {
30         Block::new("创世区块".to_string(),
31                    PRE_HASH.to_string(), CURR_BITS)
32     }
33
34     pub fn add_block(&mut self, txs: String) {
35         let pre_block = &self.blocks[self.blocks.len() - 1];
36         let pre_hash  = pre_block.hash.clone();
37         let new_block = Block::new(txs,pre_hash,self.curr_bits);
38
39         // 数据写入库
40         Self::write_block(&mut (self.blocks_db), &new_block);
41         Self::write_tail(&mut (self.blocks_db), &new_block);
42
43         println!("New produced block saved!\n");
44         self.blocks.push(new_block);
45     }
46
47     fn write_block(db: &mut Database<BKey>, block: &Block) {
48         // 基于区块链的 header 生成 key
49         let header_ser = serialize(&(block.header));
50         let mut hash_u: [u8; 32] = [0; 32];
```

```
51            hash_u8(&header_ser, &mut hash_u);
52
53            let key = BKey{ val: U256::from(hash_u) };
54            let val = serialize(&block);
55            BlockChainDb::write_db(db, key, &val);
56        }
57
58        // 将区块哈希值作为尾巴写入
59        fn write_tail(mut db: &mut Database<BKey>, block:&Block) {
60            let key = BKey{ val: U256::from("tail".as_bytes()) };
61            let val = serialize(&(block.hash));
62            BlockChainDb::write_db(&mut db, key, &val);
63        }
64    }
```

添加了存储功能的项目在 github 上的 blockchain3 仓库或 gitee 上的 blockchain3 仓库。

9.8.5 交易

前面我们的交易都是用的字符串来代替，其实交易涉及交易双方、金额、手续费等信息，这些适合封装到结构体里面，其具体字段如下。

```
1  pub struct Transaction {
2      pub nonce: u64,
3      pub amount: u64,
4      pub fee: u64,
5      pub from: String,
6      pub to: String,
7      pub sign: String,
8      pub hash: String,
9  }
```

nonce 作为交易记录值，amount 和 fee 是金额和手续费，from 和 to 交易双方的地址，sign 标记一些具体信息，hash 是整个交易的哈希值。下面是完整的交易实现。

```
1  // transaction.rs
2  use serde::Serialize;
3  use utils::serializer::{serialize, hash_str};
4
5  // 交易体
6  #[derive(Serialize, Debug)]
```

```
pub struct Transaction {
    pub nonce: u64,
    pub amount: u64,
    pub fee: u64,
    pub from: String,
    pub to: String,
    pub sign: String,
    pub hash: String,
}

impl Transaction {
    pub fn new(from: String, to: String,
               amount: u64, fee: u64,
               nonce: u64, sign: String) -> Self
    {
        let mut tx = Transaction {
            nonce,
            amount,
            fee,
            from,
            to,
            sign,
            hash:"".to_string(),
        };
        tx.set_hash();

        tx
    }

    pub fn set_hash(&mut self) {
        let txs_ser = serialize(&self);
        self.hash = hash_str(&txs_ser);
    }
}
```

一旦有了交易结构体，那么可以在区块中使用，替代前面使用的字符串。

```
// block.rs
// ...
```

```
use crate::transaction::Transaction;

#[derive(Serialize, Debug)]
pub struct Block {
    pub header: BlockHeader,
    pub tranxs: Vec<Transaction>, // 改成了具体交易
    pub hash: String,
}

impl Block {
    pub fn new(txs: Vec<Transaction>, pre_hash: String,
               bits: u32) -> Self {
        // ...
    }
}
```

同样的，blockchain.rs 也需要修改其中的交易的类型。

```
// ...
use crate::transaction::Transaction;

impl BlockChain {
    fn genesis_block() -> Block {
        // ...
        let tx = Transaction::new(from, to, 0, 0, 0, sign);
    }

    pub fn add_block(&mut self, txs: Vec<Transaction>) {
        // ...
    }
}
```

最后，main 函数也得修改。

```
// ...
use core::transaction::Transaction;

fn main() {
    // ...
    let sign = format!("{} -> {}: 9 btc", from, to);
    let tx   = Transaction::new(from, to, 9, 1, 0, sign);
```

```
8
9       // ...
10      let sign = format!("{} -> {}: 6 btc", from, to);
11      let tx   = Transaction::new(from, to, 6, 1, 0, sign);
12
13      // ...
14  }
```

添加了交易功能的项目在 github 上的 blockchain4 仓库或 gitee 上的 blockchain4 仓库。

9.8.6 账户

前面的交易结构体包含 from 和 to 字段，表示交易的账户，然而实际上我们并没有账户。所以，本节来实现账户。账户要包含地址、余额，其次还可以包含姓名、哈希标志。

```
1  pub struct Account {
2      pub nonce: u64,
3      pub balance: u64,
4      pub name: String,
5      pub address: String,
6      pub hash: String,
7  }
```

有了账户，可以将交易（转款）实现为其函数，这样账户和交易就绑定在一起，比较好理解。为了模拟，假设初始时，每个账户有 100 个比特币（现实是 0）。

```
1  use crate::transaction::Transaction;
2  use utils::serializer::{serialize, hash_str};
3  use serde::Serialize;
4
5  // 账户
6  #[derive(Serialize, Debug, Eq, PartialEq, Clone)]
7  pub struct Account {
8      pub nonce: u64,
9      pub balance: u64,
10     pub name: String,
11     pub address: String,
12     pub hash: String,
13 }
14
15 impl Account {
```

```
16      pub fn new(address: String, name: String) -> Self {
17          let mut account = Account {
18              nonce: 0,
19              name: name,
20              balance: 100,
21              address: address,
22              hash: "".to_string(),
23          };
24          account.set_hash();
25
26          account
27      }
28
29      fn set_hash(&mut self) {
30          let data = serialize(&self);
31          self.hash = hash_str(&data);
32      }
33
34      // 交易, 此处就只是转移比特币
35      pub fn transfer_to(&mut self, to: &mut Self,
36          amount: u64, fee: u64) -> Result<Transaction, String>
37      {
38          if amount + fee > self.balance {
39              return Err("Error:not enough amount!".to_string());
40          }
41
42          self.balance -= amount;
43          self.balance -= fee;
44          self.nonce += 1;
45          self.set_hash();
46
47          to.balance += amount;
48          to.nonce += 1;
49          to.set_hash();
50          let sign = format!("{} -> {}: {} btc",
51                             self.address.clone(),
52                             to.address.clone(),
53                             amount);
```

```
54          let tx = Transaction::new(self.address.clone(),
55                                    to.address.clone(),
56                                    amount,fee,self.nonce,sign);
57          Ok(tx)
58      }
59
60      pub fn account_info(&self) {
61          println!("{:#?}", &self);
62      }
63  }
```

有了账户，那么 main 函数中的交易就可以用账户来处理。

```
1   // main.rs
2   // ..
3   use core::account::Account;
4
5   fn main() {
6       let user1 = "Kim".to_string();
7       let user2 = "Tom".to_string();
8       let user3 = "Jim".to_string();
9       let mut acnt1 = Account::new("0xabcd".to_string(), user1);
10      let mut acnt2 = Account::new("0xabce".to_string(), user2);
11      let mut acnt3 = Account::new("0xabcf".to_string(), user3);
12
13      println!("----------------Mine Info--------------------");
14      // ...
15
16      let res = acnt1.transfer_to(&mut user2, 9, 1);
17      match res {
18          Ok(tx) => bc.add_block(vec![tx]),
19          Err(e) => panic!("{}", e),
20      }
21
22      let res = acnt2.transfer_to(&mut user3, 6, 1);
23      match res {
24          Ok(tx) => bc.add_block(vec![tx]),
25          Err(e) => panic!("{}", e),
26      }
```

```
27
28      println!("---------------Account Info-------------------");
29      let users = vec![&acnt1, &acnt2, &acnt2];
30      for u in users {
31          u.account_info();
32      }
33
34      // ..
35  }
```

添加了账户功能的项目在 github 上的 blockchain5 仓库或 gitee 上的 blockchain5 仓库。

9.8.7 梅根哈希

前面我们分析过，tx_hash 是所有交易哈希两两计算得到的最终结果，但实际还没有实现，我们前面其实使用的就是对所有交易的哈希。本节来实现计算所有交易的哈希值：梅根哈希。

首先创建一个梅根树来放置所有哈希值，然后两两合并再求哈希值并放入梅根树中，最后得到唯一的一个哈希值作为 tx_hash 填入 header 。

```
1  impl Block {
2      pub fn new(txs: Vec<Transaction>, pre_hash: String,
3                 bits: u32) -> Self {
4          let time = Utc::now().timestamp();
5          let txs_hash = Self::merkle_hash_str(&txs);
6          // ...
7
8          let pow = ProofOfWork::new(bits);
9          pow.run(&mut block);
10
11         block
12     }
13
14     // 计算梅根哈希值
15     fn merkle_hash_str(txs: &Vec<Transaction>) -> String {
16         if txs.len() == 0 {
17             return "00000000".to_string();
18         }
19
20         let mut merkle_tree: Vec<String> = Vec::new();
```

```
21          for tx in txs {
22              merkle_tree.push(tx.hash.clone());
23          }
24
25          let mut j: u64 = 0;
26          let mut size = merkle_tree.len();
27          while size > 1 {
28              let mut i: u64 = 0;
29              let temp = size as u64;
30
31              while i < temp {
32                  let k = Self::min(i + 1, temp - 1);
33                  let idx1 = (j + i) as usize;
34                  let idx2 = (j + k) as usize;
35                  let hash1 = merkle_tree[idx1].clone();
36                  let hash2 = merkle_tree[idx2].clone();
37                  let merge = format!("{}-{}",hash1, hash2);
38                  let merge_ser = serialize(&merge);
39                  let merge_hash = hash_str(&merge_ser);
40                  merkle_tree.push(merge_hash);
41                  i += 2;
42              }
43
44              j += temp;
45              size = (size + 1) >> 1;
46          }
47
48          if merkle_tree.len() != 0 {
49              merkle_tree.pop().unwrap()
50          } else {
51              "00000000".to_string()
52          }
53      }
54
55      fn min(num1: u64, num2: u64) -> u64 {
56          if num1 >= num2 {
57              num2
58          } else {
```

```
59                  num1
60              }
61          }
62      }
```

添加了梅根哈希的项目在 github 上的 blockchain6 仓库或 gitee 上的 blockchain6 仓库。

9.8.8 矿工及挖矿

区块链的交易会通过打包来保存成区块，打包过程又被称为挖矿。可是前面的实现中，并没有矿工，挖矿也只是区块来调动的，这是不符合实际的，所以还需要设置矿工。有了用户账户、矿工，交易就可以通过矿工打包来存储区块到链上。一个矿工要包含自己的地址，用于接受比特币，一个账户余额，当然也可以加上名字。

```
1  pub struct Miner {
2      name: String,
3      balance: u64,
4      address: String,
5  }
```

矿工第一笔交易是 coinbase 交易，如果最后打包成功则获得挖矿奖励 50 比特币（随时间半衰）。其他的交易则陆续加入。然后进行工作量证明，开始挖矿。

```
1  // miner.rs
2  use crate::block::Block;
3  use crate::pow::ProofOfWork;
4  use crate::transaction::Transaction;
5
6  const MINER_NAME: &str = "anonymous";
7
8  // 矿工
9  #[derive(Debug, Clone)]
10 pub struct Miner {
11     name: String,
12     balance: u64,
13     address: String,
14 }
15
16 impl Miner {
17     pub fn new(address: String) -> Self {
18         Miner {
```

```
19              name: MINER_NAME.to_string(),
20              balance: 0,
21              address: address,
22          }
23      }
24
25      pub fn mine_block(&self, txs: &mut Vec<Transaction>,
26                        pre_hash: String, bits: u32) -> Block {
27          let from = "0x0000".to_string();
28          let to = self.address.clone();
29          let sign = format!("{} -> {}: 50 btc", from, to);
30          let coinbase = Transaction::new(from,to,0,0,0,sign);
31
32          // 加入 coinbase 交易和普通交易
33          let mut txs_2: Vec<Transaction> = Vec::new();
34          txs_2.push(coinbase);
35          txs_2.append(txs);
36
37          Self::mine_job(txs_2, pre_hash, bits)
38      }
39
40      // 挖矿任务-工作量证明
41      fn mine_job(txs: Vec<Transaction>, pre_hash: String,
42                  bits: u32) -> Block {
43          let mut block = Block::new(txs, pre_hash, bits);
44          let pow = ProofOfWork::new(bits);
45          pow.run(&mut block);
46
47          block
48      }
49
50      pub fn miner_info(&self) {
51          println!("{:#?}", &self);
52      }
53  }
```

有了矿工，我们还需要将其挖到的矿（区块）加入区块链中。一种直观的方法是将区块链和矿工放到一起组成挖矿任务，每当矿工挖到一个区块就立马添加到区块链中。我们的挖矿任务可以定义为结构体 Mine，内含 Miner 和 BlockChain 两个变量。

```
pub struct Mine {
    pub miner: Miner,
    pub blockchain: BlockChain,
}
```

这样 miner 挖到区块后就可以直接加入链。

```
// mine.rs
use crate::miner::Miner;
use crate::blockchain::BlockChain;
use crate::transaction::Transaction;

const MINER_ADDRESS: &str = "0x1b2d";

// 挖矿任务
pub struct Mine {
    pub miner: Miner,
    pub blockchain: BlockChain,
}

impl Mine {
    pub fn new() -> Self {
        Mine {
            blockchain: BlockChain::new(),
            miner: Miner::new(MINER_ADDRESS.to_string()),
        }
    }

    pub fn mining(&mut self, txs: &mut Vec<Transaction>) {
        // 准备 pre_hash 和难度值
        let pre_hash = self.blockchain.curr_hash.clone();
        let bits = self.blockchain.curr_bits.clone();
        // 核心代码点: 开始挖矿
        let block = self.miner.mine_block(txs,pre_hash,bits);
        // 区块保存
        self.blockchain.add_block(block);
    }
}
```

有了挖矿任务结构体 Mine，我们又可以修改 main 里的挖矿任务。同时要注意到，我们打印区块的信息移到了链里，增加了打印矿工信息的代码。

```
// main.rs
// ...
use core::mine::Mine;

fn main() {
    // ...
    println!("------------------Mine Info--------------------");
    let mut mine = Mine::new();

    let mut txs: Vec<Transaction> = Vec::new();
    let res = user1.transfer_to(&mut user2, 9, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    let res = user1.transfer_to(&mut user2, 5, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    mine.mining(&mut txs);

    let mut txs: Vec<Transaction> = Vec::new();
    let res = user2.transfer_to(&mut user3, 6, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    let res = user2.transfer_to(&mut user3, 3, 1);
    match res {
        Ok(tx) => txs.push(tx),
        Err(e) => panic!("{}", e),
    }
    mine.mining(&mut txs);

    println!("---------------Miner Info--------------------");
```

```
37      mine.miner.miner_info();
38
39      // ...
40
41      println!("----------------Block Info----------------------");
42      mine.blockchain.block_info();
43  }
```

因为有了这些改变，所以 blockchain.rs 也需要相应改变。

```
1   // blockchain.rs
2   // ...
3   use std::sync::Mutex;
4   use std::collections::HashMap;
5
6   pub struct BlockChain {
7       blocks_db: Box<Database<BKey>>,
8       blocks_index: Mutex<HashMap<String, Block>>,
9       pub gnes_hash: String,
10      pub curr_hash: String,
11      pub curr_bits: u32,
12  }
13
14  impl BlockChain {
15      pub fn new() -> Self {
16          // ...
17
18          let gene_block = genesis.clone();
19          let mut block_index = Mutex::new(HashMap::new());
20          Self::update_hmap(&mut block_index, gene_block);
21
22          // ....
23      }
24
25      fn genesis_block() -> Block {
26          println!("Start mining .... ");
27          let from = "0x0000".to_string();
28          let to   = "0x0000".to_string();
29          let sign = "创世区块".to_string();
```

```
30          let tx = Transaction::new(from, to, 0, 0, 0, sign);
31          let mut block  = Block::new(vec![tx],
32                                      PRE_HASH.to_string(), INIT_BITS);
33
34          let header_ser=ProofOfWork::prepare_data(&mut block,0);
35          block.hash = hash_str(&header_ser);
36          println!("Produced a new block!");
37
38          block
39      }
40
41      pub fn add_block(&mut self, block: Block) {
42          // ...
43          self.curr_hash = block.hash.clone();
44          self.curr_bits = block.header.bits.clone();
45          Self::update_hmap(&mut self.blocks_index, block);
46      }
47
48      fn update_hmap(hmap: &mut Mutex<HashMap<String, Block>>,
49                     block: Block) {
50          let mut hmap = hmap.lock().unwrap();
51          let hash = block.hash.clone();
52          hmap.insert(hash, block);
53      }
54
55      pub fn  block_info(&self) {
56          let mut hash = self.curr_hash.clone();
57          let hmap = self.blocks_index.lock().unwrap();
58          let mut blocks: Vec<Block> = Vec::new();
59
60          loop {
61              if let Some(b) = hmap.get(&hash) {
62                  blocks.push(b.clone());
63                  hash = b.header.pre_hash.clone();
64              } else {
65                  panic!("Error getting block");
66              }
67
```

```
68                  if hash == self.gnes_hash {
69                      if let Some(b) = hmap.get(&hash) {
70                          blocks.push(b.clone());
71                      }
72                      break;
73                  }
74              }
75              blocks.reverse();
76
77              for b in blocks {
78                  println!("{:#?}", b);
79              }
80          }
81      }
```

添加了矿工的项目在 github 上的 blockchain7 仓库或 gitee 上的 blockchain7 仓库。

9.8.9 比特币奖励

最后一步就是将矿工挖矿该得的收益支付给矿工，只需要修改少量代码就可以了。

```
1   impl Miner {
2       pub fn mine_block(&mut self,
3                         txs: &mut Vec<Transaction>,
4                         pre_hash: String,
5                         bits: u32) -> Block {
6           let mut fee = 0; // 挖矿手续费
7           for tx in txs.iter() {
8               fee += tx.fee.clone();
9           }
10          // ...
11          // 挖矿奖励，实际中会半衰 50、25、12.5
12          self.balance += 50;
13          self.balance += fee;
14
15          block
16      }
17  }
```

最终实现的完整项目在 github 上的 blockchain8 仓库或 gitee 上的 blockchain8 仓库。

9.8.10 回顾

现在得到了一个完整区块链项目，我们从基本的区块链开始逐步向其中添加了许多功能，最终的区块链实现了账户、矿工、挖矿、交易等功能，这些内容是对前面学习的数据结构的总复习。

下图是运行过程中输出的挖矿及区块链、矿工及用户信息，还可以看到用户和矿工的余额 balance，钱包地址 address 等信息。

```
--------------------------------Mine Info-------------------------------------
Start mining ...
Produced a new block!
New produced block saved!

Start mining ...
Produced a new block!
New produced block saved!

Start mining ...
Produced a new block!
New produced block saved!
--------------------------------Miner Info-------------------------------------
Miner {
        name: “anonymous”,
        balance: 204,
        address: “0x1b2d”,
}
--------------------------------Account Info-----------------------------------
Account {
        nonce: 2,
        name: “Kim”,
        balance: 84,
        address: “0xabcd”,
        hash: “19a1b86d52dc298fb8ec67080b3cfe4672cb640b79e9b4bfd43fcaacce40e618”,
}
Account {
        nonce: 4,
        name: “Tom”,
        balance: 103,
        address: “0xabce”,
        hash: “bcc52eda8d4b4ef78e2b7c9e5ed720141fe4a4e05646ccf5418ed3fbd2ea7541”,
}
Account {
        nonce: 2,
        name: “Jim”,
        balance: 109,
        address: “0xabcf”,
        hash: “3d6cb5ed0f318bf9c57bd5912937eb8561def6702f84835424de6bc2c37e24c8”,
}
```

下图是生成的一个区块，里面是交易信息，每条交易包含了转帐金额，手续费，交易双方账户。最后的 hash 值是整个区块的计算得到的哈希，也是下一个区块的 pre_hash。

```
Block {
    header: BlockHeader {
        nonce: 0,
        time: 1619008034,
        bits: 553713663,
        txs_hash: "20c4a3b94c760d6a9f0bd9b2cd0dcb683cd6ad57d8c8fd3287df2df2eb5c1d3c",
        pre_hash: "912fdc07d0b5ddb7da343a295f856acdd1d4b4df8d9f0d82e441aeb443df9c7e",
    },
    tranxs: [
        Transaction {
            nonce: 0,
            amount: 0,
            fee: 0,
            from: "0x0000",
            to: "0x1b2d",
            sign: "0x0000 -> 0x1b2d: 50 btc",
            hash: "9a1cd23a5b0ecb6326621ae93c218e8db4499283386ce6b6008d496d469364c2",
        },
        Transaction {
            nonce: 1,
            amount: 9,
            fee: 1,
            from: "0xabcd",
            to: "0xabce",
            sign: "0xabcd -> 0xabce: 9 btc",
            hash: "53d5cc1ac2b19555233eb2a9e8fdb62fb4a19387127f5f3b45b195edaa568305",
        },
        Transaction {
            nonce: 2,
            amount: 5,
            fee: 1,
            from: "0xabcd",
            to: "0xabce",
            sign: "0xabcd -> 0xabce: 5 btc",
            hash: "84fa895cb4f0d21d555e66366af23648987fd81da539d99cb4f6810558863c2d",
        },
    ],
    hash: "163e2ffcb1abc15c941f6cae4534a0383d001857092d31dd8a4b55e27a4da44e",
}
```

9.9 总结

本章的实践包含许多数据结构，而且都非常有用。我们学习了如何实现字典树，布隆和布谷鸟过滤器；懂得了汉明距离的原理，编辑距离需要动态规划来解决；缓存淘汰算法 LRU 和一致性哈希算法非常常用。最后逐步实现的区块链是最复杂的项目，它使用到了各种数据结构，算是全书综合复习。

本书行文至此，学习了各种数据结构，也写了非常多 Rust 代码，希望本书能对读者有所帮助并促进 Rust 在中国的发展。

参考文献

[1] Multicians. Multics. Website, 1995. https://www.multicians.org.

[2] The Open Group. Unix. Website, 1995. https://unix.org/what_is_unix.html.

[3] Linus. Linux 内核官网. Website, 1991. https://www.kernel.org.

[4] GNU. Gnu/linux. Website, 2010. https://www.gnu.org.

[5] Wikipedia. 量子计算机. Website, 2022. https://en.wikipedia.org/wiki/Quantum_computing.

[6] Bradley N. Miller and David L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle & Associates, US, 2011.

[7] Rust Foundation. Rust 基金会. Website, 2021. https://foundation.rust-lang.org/members/.

[8] Wikipedia. Np 完全问题. Website, 2021. https://zh.wikipedia.org/wiki/NP%E5%AE%8C%E5%85%A8.

[9] Wikipedia. 歌德巴赫猜想. Website, 2021. https://zh.wikipedia.org/zh-cn/%E5%93%A5%E5%BE%B7%E5%B7%B4%E8%B5%AB%E7%8C%9C%E6%83%B3.

[10] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logn search. *Commun. ACM*, 21(7):550–553, jul 1978.

[11] Stanley P. Y. Fung. Is this the simplest (and most surprising) sorting algorithm ever?, 2021.

[12] Wikipedia. 汉明码. Website, 2022. https://zh.wikipedia.org/zh-hans/%E6%B1%89%E6%98%8E%E7%A0%81.

[13] Bin Fan and David G Andarsen. Cuckoo filter: Practically better than bloom. Website, 2014. https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

[14] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Website, 2008. https://bitcoin.org/bitcoin.pdf.

[15] Google. Leveldb. Website, 2010. https://github.com/google/leveldb.

[16] Facebook. Rocksdb. Website, 2014. https://github.com/facebook/rocksdb.