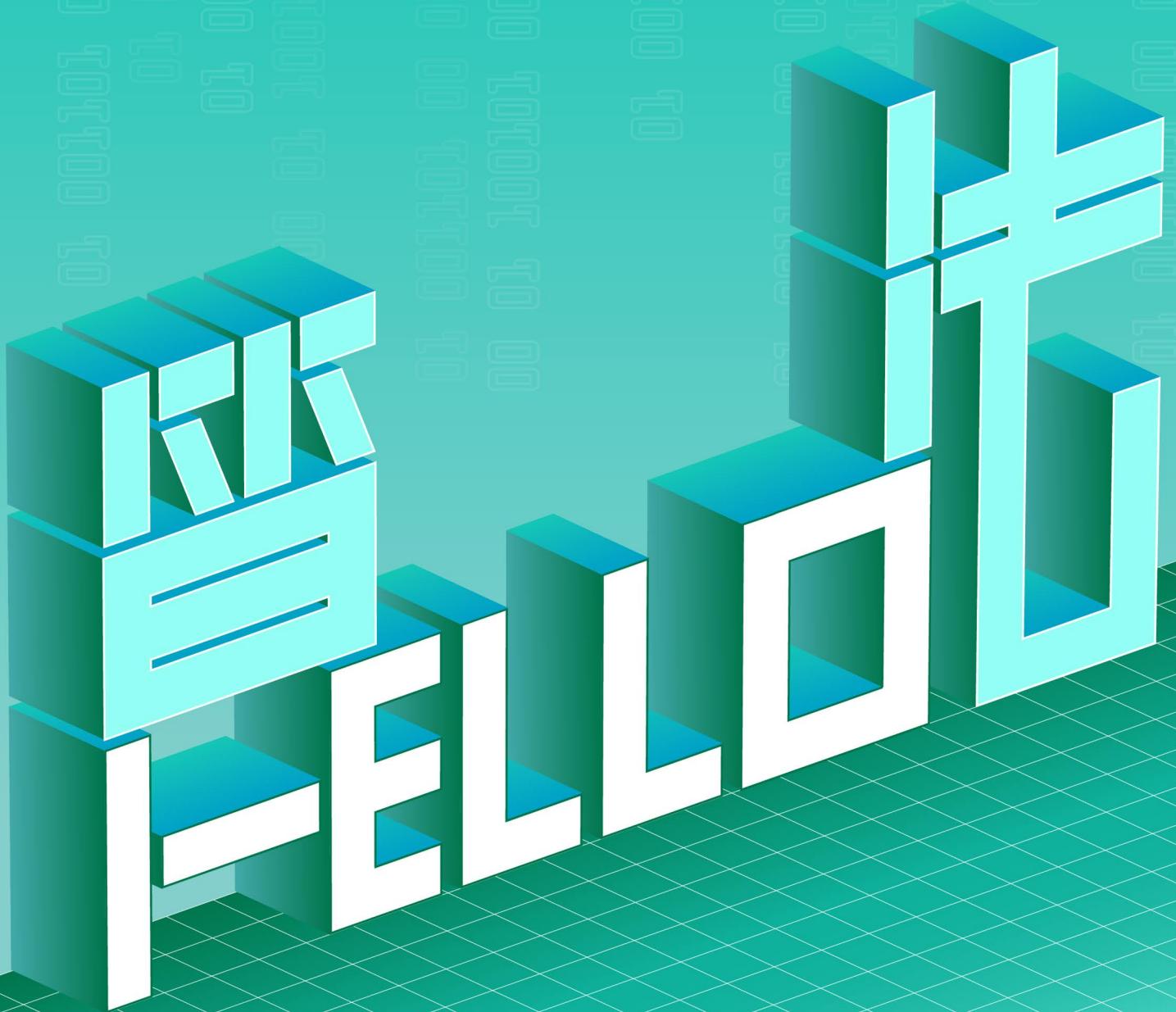


I-ELLO I-ELLO I-ELLO I-ELLO I-ELLO I-ELLO I-ELLO

> Hello 算法

动画图解、能运行、可提问的数据结构与算法入门教程

作者：靳宇栋 (Krahets)



Hello 算法

C++ 语言版

靳宇栋 (Krahets)



Release 1.0.0b1
2023-03-01

序

两年前，我在力扣上分享了《剑指 Offer》系列题解，受到了很多小伙伴的喜爱与支持。在此期间，我也回复了许多读者的评论问题，遇到最多的问题是“如何入门学习算法”。我渐渐也对这个问题好奇了起来。

两眼一抹黑地刷题应该是最受欢迎的方式，简单粗暴且有效。然而，刷题就如同玩“扫雷”游戏，自学能力强的同学能够顺利地将地雷逐个排掉，而基础不足的同学很可能被炸的满头是包，并在受挫中步步退缩。通读教材书籍也是常用方法，但对于面向求职的同学来说，毕业季、投递简历、应付笔面试已经占用大部分精力，厚重的书本也因此成为巨大的挑战。

如果你也有上述烦恼，那么很幸运这本书找到了你。本书是我对于该问题给出的答案，虽然不一定正确，但至少代表一次积极的尝试。这本书虽然不足以让你直接拿到 Offer，但会引导你探索数据结构与算法的“知识地图”，带你了解不同“地雷”的形状大小和分布位置，让你掌握各种“排雷方法”。有了这些本领，相信你可以更加得心应手地刷题与阅读文献，逐步搭建起完整的知识体系。

书内的代码配有可一键运行的源文件，托管在 github.com/krahets/hello-algo 仓库。动画在 PDF 内的展示效果有限，可前往 hello-algo.com 网页版获得更好的阅读体验。

致谢

本书在开源社区的群策群力下逐步成长，感谢每一位撰稿人，是他们的无私奉献让这本书变得更好，他们是（按照 GitHub 自动生成的顺序）：krahets, justin-tse, sjinzh, Reanon, nuomi1, Gonglja, S-N-O-R-L-A-X, danielsss, RiverTwilight, msk397, gyt95, zhuoqinyue, FangYuan33, mingXta, Xia-Sang, guowei-gong, GN-Yu, JoseHung, IsChristina, pengchzn, qualifier1024, Cathay-Chen, what-is-me, L-Super, Slone123c, mgisr, xBLACKICE, longranger2, xiongsp, WSL0809, Wonderdch, a16su, JeffersonHuang, xjr7670, MolDuM, XC-Zero, DullSword, iron-irax, huawuque404, 4yDX3906, ZJKung, xb534, Guanngxu, siqyka, ZnYang2018, beintentional, luluxia, GaochaoZhu, weibk, dshlstarr, ShiMaRing, fbigm, Aesperero, iStig, YuelinXin, szu17dmy, hezhizhen, fanchenggang, Keynman, youshaoXG, lipusheng, Javesun99, tao363, czruby, gltianwen, liuxjerry, yabo083.

本书的代码审阅工作由 justin-tse, krahets, nuomi1, Reanon, sjinzh 完成，感谢他们的辛勤付出！

推荐语

“一本通俗易懂的数据结构与算法入门书，引导读者手脑并用地学习，强烈推荐算法初学者阅读。”

——邓俊辉，清华大学计算机系教授

“如果我当年学数据结构与算法的时候有《Hello 算法》，学起来应该会简单 10 倍！”

——李沐，亚马逊资深首席科学家

目 录

0. 写在前面	1
0.1. 关于本书	1
0.2. 如何使用本书	3
0.3. 小结	7
1. 引言	8
1.1. 算法无处不在	8
1.2. 算法是什么	9
1.3. 小结	11
2. 复杂度分析	12
2.1. 算法效率评估	12
2.2. 时间复杂度	13
2.3. 空间复杂度	27
2.4. 权衡时间与空间	33
2.5. 小结	35
3. 数据结构简介	37
3.1. 数据与内存	37
3.2. 数据结构分类	41
3.3. 小结	42
4. 数组与链表	44
4.1. 数组	44
4.2. 链表	48
4.3. 列表	53
4.4. 小结	58
5. 栈与队列	59
5.1. 栈	59
5.2. 队列	65
5.3. 双向队列	72
5.4. 小结	81
6. 散列表	82
6.1. 哈希表	82
6.2. 哈希冲突	87
6.3. 小结	90
7. 树	91
7.1. 二叉树	91
7.2. 二叉树遍历	99
7.3. 二叉搜索树	102
7.4. AVL 树 *	110
7.5. 小结	121
8. 堆	123
8.1. 堆	123
8.2. 建堆操作 *	130
8.3. 小结	132

9. 图	134
9.1. 图	134
9.2. 图基础操作	138
9.3. 图的遍历	145
9.4. 小结	152
10. 查找算法	153
10.1. 线性查找	153
10.2. 二分查找	154
10.3. 哈希查找	158
10.4. 小结	160
11. 排序算法	162
11.1. 排序简介	162
11.2. 冒泡排序	164
11.3. 插入排序	167
11.4. 快速排序	170
11.5. 归并排序	175
11.6. 小结	179
12. 附录	180
12.1. 编程环境安装	180
12.2. 一起参与创作	181

0. 写在前面

0.1. 关于本书

本项目致力于构建一本开源免费、新手友好的数据结构与算法入门书。

- 全书采用动画图解，结构化地讲解数据结构与算法知识，内容清晰易懂、学习曲线平滑；
- 算法源代码皆可一键运行，支持 Java, C++, Python, Go, JS, TS, C#, Swift, Zig 等语言；
- 鼓励读者在章节讨论区互帮互助、共同进步，提问与评论一般能在两日内得到回复；

0.1.1. 读者对象

如果您是「算法初学者」，完全没有接触过算法，或者已经有少量刷题，对数据结构与算法有朦胧的理解，在会与不会之间反复横跳，那么这本书就是为你而写！

如果您是「算法老手」，已经积累一定刷题量，接触过大多数题型，那么本书可以帮助你回顾与梳理算法知识体系，仓库源代码可以被当作“刷题工具库”或“算法字典”来使用。

如果您是「算法大佬」，希望可以得到你的宝贵意见建议，或者[一起参与创作](#)。



前置条件

您需要至少具备任一语言的编程基础，能够阅读和编写简单代码。

0.1.2. 内容结构

本书主要内容有：

- **复杂度分析：**数据结构与算法的评价维度、算法效率的评估方法。时间复杂度、空间复杂度，包括推算方法、常见类型、示例等。
- **数据结构：**常用的基本数据类型，数据在内存中的存储方式、数据结构分类方法。数组、链表、栈、队列、散列表、树、堆、图等数据结构，内容包括定义、优劣势、常用操作、常见类型、典型应用、实现方法等。
- **算法：**查找算法、排序算法、搜索与回溯、动态规划、分治算法，内容包括定义、使用场景、优劣势、时空效率、实现方法、示例题目等。

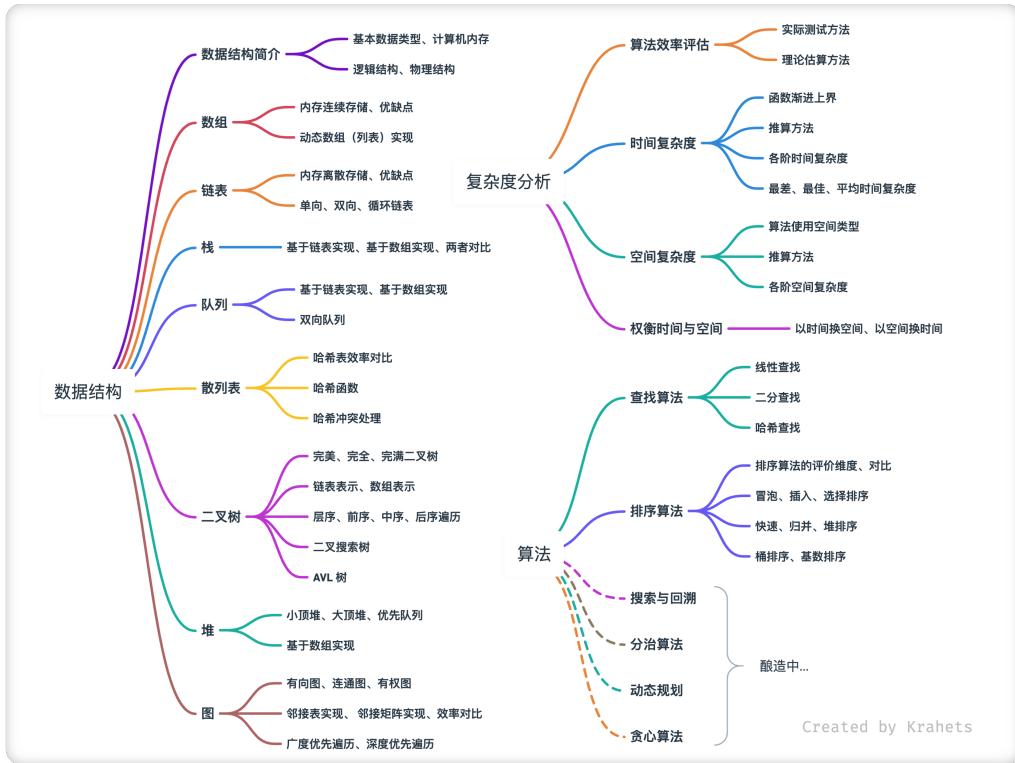


Figure 0-1. Hello 算法内容结构

0.1.3. 致谢

本书的成书过程中，我获得了许多人的帮助，包括但不限于：

- 感谢我的导师李博，在小酌畅谈时您告诉我“觉得应该做就去做”，坚定了我写这本书的决心。
- 感谢我的女朋友泡泡担任本书的首位读者，从算法小白的视角提出了许多建议，使这本书更加适合初学者来阅读。
- 感谢腾宝、琦宝、飞宝为本书起了个好听又有梗名字，直接唤起我最初敲下第一行代码“Hello World!”的回忆。
- 感谢苏潼为本书设计了封面和 LOGO，在我的强迫症下前后多次帮忙修改，谢谢你的耐心。
- 感谢 @squidfunk 给出的写作排版建议，以及优秀开源项目 [Material-for-MkDocs](#)。

本书鼓励“手脑并用”的学习方式，在这点上受到了《动手学深度学习》很大影响，也在此向各位同学强烈推荐这本著作，包括[中文版](#)、[英文版](#)、[李沐老师 bilibili 主页](#)。

在写作过程中，我阅读了许多数据结构与算法的教材与文章，这些著作为本书作出了很好的榜样，保证了本书内容的正确性与质量，感谢前辈们的精彩创作！

感谢父母，你们一贯的支持与鼓励给了我自由度来做这些有趣的事。

0.2. 如何使用本书

建议通读本节内容，以获取最佳阅读体验。

0.2.1. 算法学习路线

总体上看，我认为可将学习数据结构与算法的过程分为三个阶段。

1. **算法入门**。熟悉各种数据结构的特点、用法，学习各种算法的原理、流程、用途、效率等。
2. **刷算法题**。可以先从热门题单开刷，推荐剑指 Offer、LeetCode Hot 100，先积累至少 100 道题量，熟悉大多数的算法问题。刚开始刷题时，“遗忘”是最大的困扰点，但这是很正常的，请不要担心。学习中有一种概念叫“周期性回顾”，同一道题隔段时间做一次，在重复 3 轮以上后，往往就能牢记于心了。
3. **搭建知识体系**。在学习方面，可以阅读算法专栏文章、解题框架、算法教材，不断地丰富知识体系。在刷题方面，可以开始采用进阶刷题方案，例如按专题分类、一题多解、一解多题等，相关刷题心得可以在各个社区中找到。

作为一本入门教程，本书内容主要对应“第一阶段”，致力于帮助你更高效地开展第二、三阶段的学习。

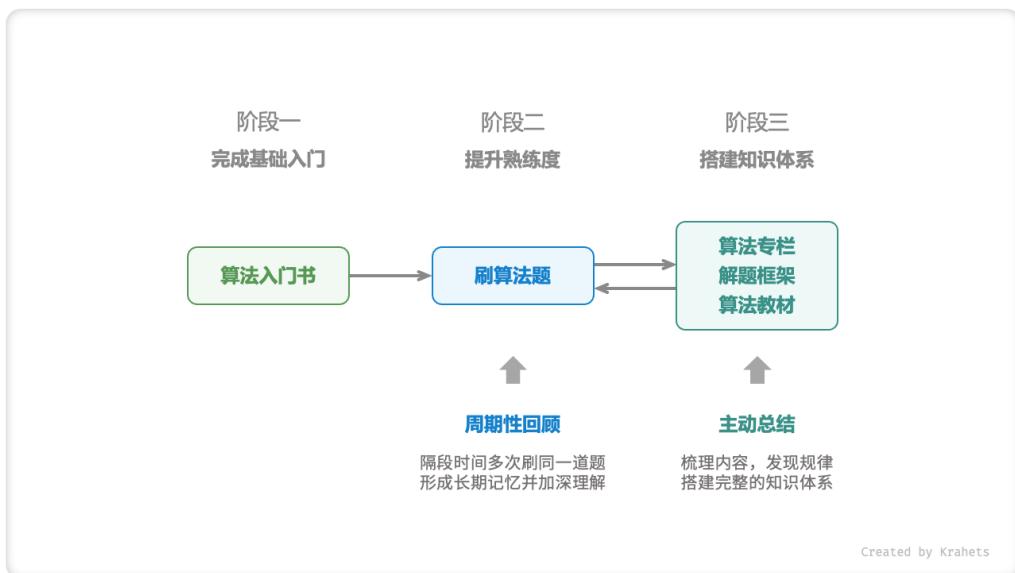


Figure 0-2. 算法学习路线

0.2.2. 行文风格约定

标题后标注 * 是的选读章节，内容相对较难。如果你的时间有限，建议可以先跳过。

文章中的重要名词会用「括号」标注，例如「数组 Array」。建议记住这些名词，包括英文翻译，以便后续阅读文献时使用。

重点内容、总起句、总结句会被**加粗**，此类文字值得特别关注。

专有名词和有特指含义的词句会使用“双引号”标注，以避免歧义。

本书部分放弃了编程语言的注释规范，以换取更加紧凑的内容排版。注释主要分为三种类型：标题注释、内容注释、多行注释。

```
/* 标题注释，用于标注函数、类、测试样例等 */

// 内容注释，用于详解代码

/**  
 * 多行  
 * 注释  
 */
```

0.2.3. 在动画图解中高效学习

视频和图片相比于文字的信息密度和结构化程度更高，更容易理解。在本书中，**知识重难点会主要以动画、图解的形式呈现**，而文字的作用则是作为动画和图的解释与补充。

阅读本书时，若发现某段内容提供了动画或图解，**建议你以图为主线**，将文字内容（一般在图的上方）对齐到图中内容，综合来理解。

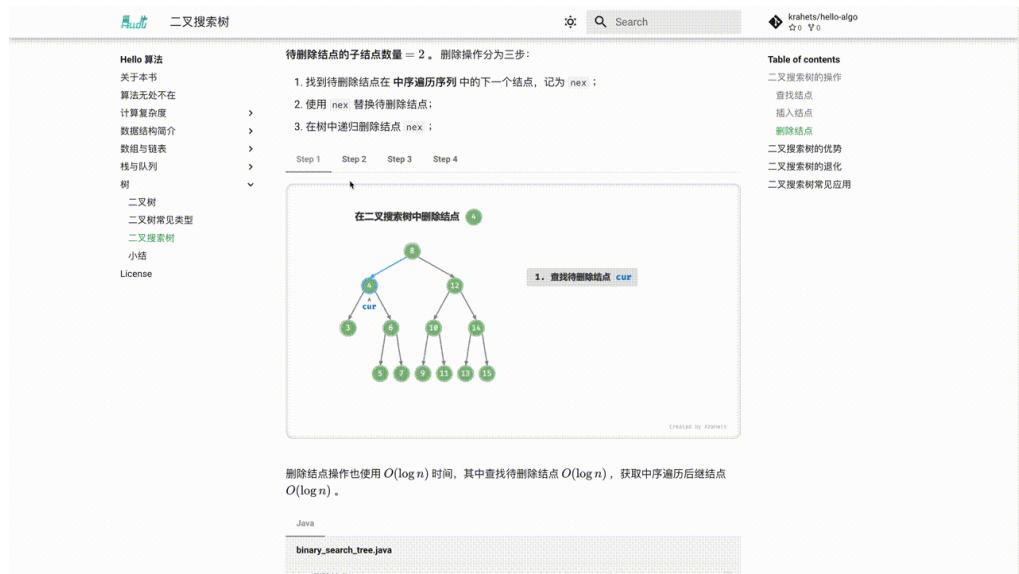


Figure 0-3. 动画图解示例

0.2.4. 在代码实践中加深理解

本书的配套代码托管在[GitHub 仓库](#)，源代码包含详细注释，配有测试样例，可以直接运行。

- 若学习时间紧张，建议至少将所有代码通读并运行一遍。
- 若时间允许，强烈建议对照着代码自己敲一遍。相比于读代码，写代码的过程往往能带来新的收获。

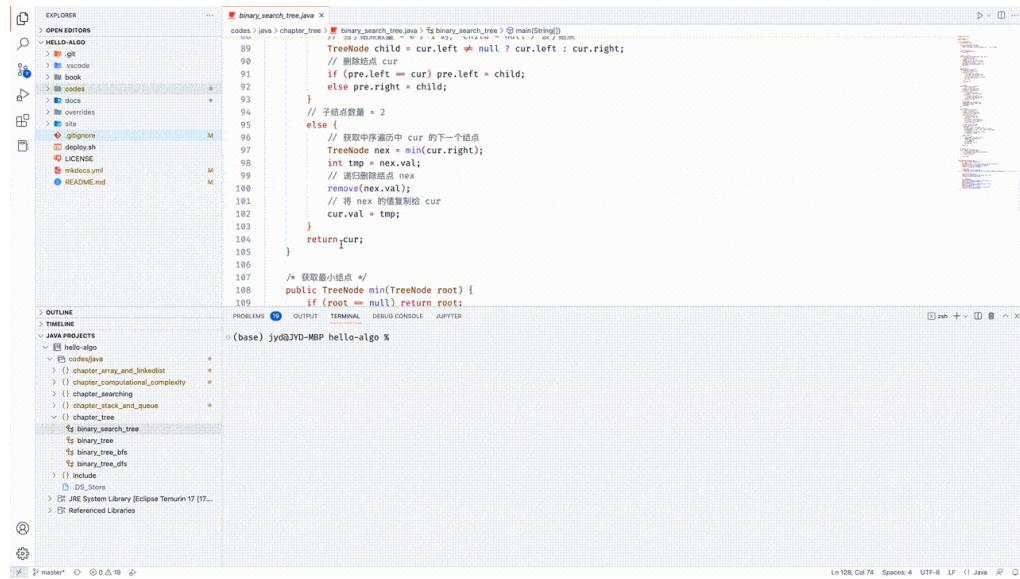


Figure 0-4. 运行代码示例

第一步：安装本地编程环境。参照[附录教程](#)，如果已有可直接跳过。

第二步：下载代码仓。如果已经安装 Git，可以通过命令行来克隆代码仓。

```
git clone https://github.com/krahets/hello-algo.git
```

当然，你也可以点击“Download ZIP”直接下载代码压缩包，本地解压即可。

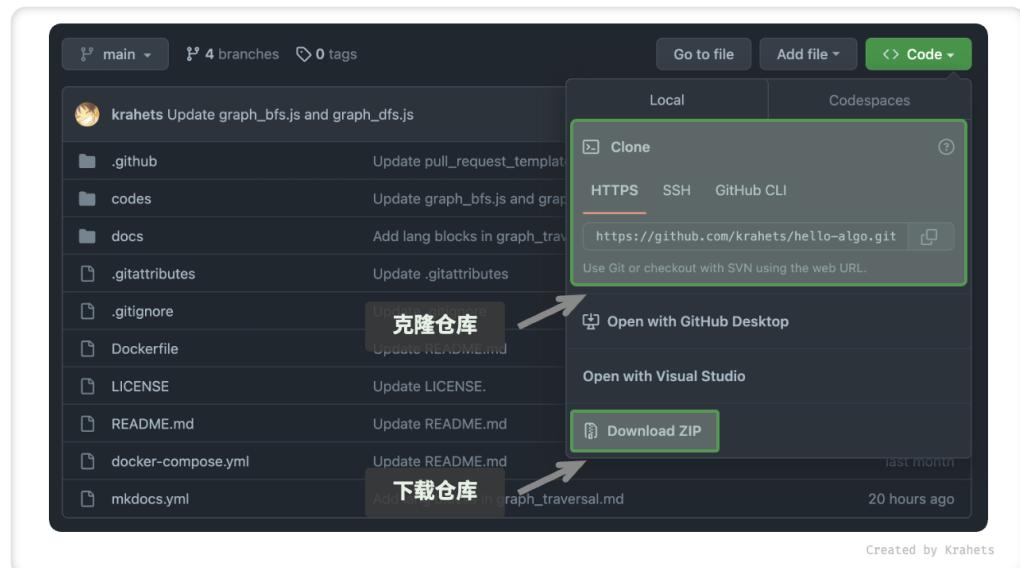


Figure 0-5. 克隆仓库与下载代码

第三步：运行源代码。若代码块的顶部标有文件名称，则可在仓库 codes 文件夹中找到对应的 源代码文件。源代码文件可以帮助你省去不必要的调试时间，将精力集中在学习内容上。



Figure 0-6. 代码块与对应的源代码文件

0.2.5. 在提问讨论中共同成长

阅读本书时，请不要“惯着”那些弄不明白的知识点。欢迎在评论区留下你的问题，小伙伴们和我都会给予解答，您一般 2 日内会得到回复。

同时，也希望你可以多花时间逛逛评论区。一方面，可以看看大家遇到了什么问题，反过来查漏补缺，这往往可以引起更加深度的思考。另一方面，也希望你可以慷慨地解答小伙伴们的问题、分享自己的见解，大家互相学习与进步！

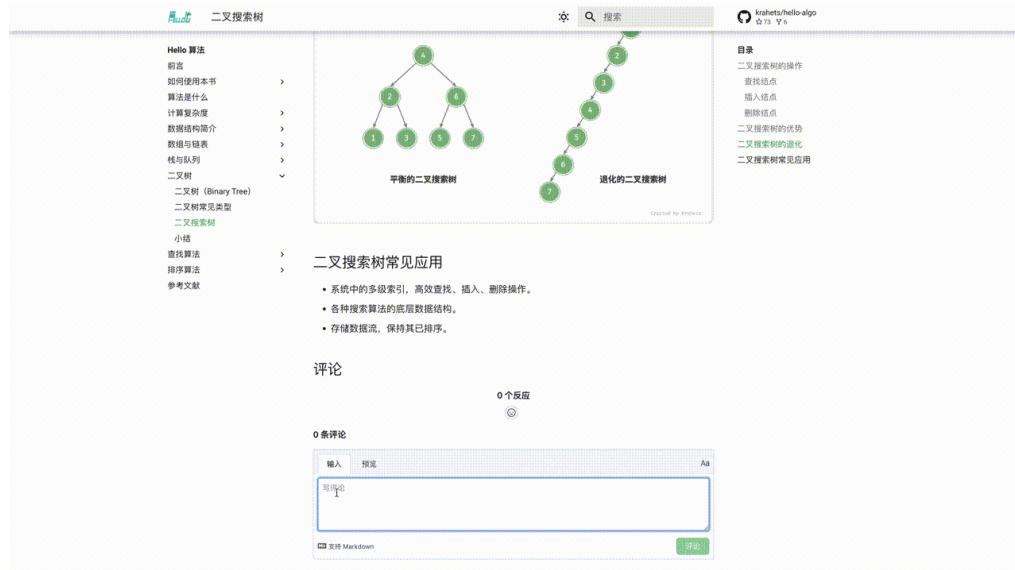


Figure 0-7. 评论区示例

0.3. 小结

- 本书主要面向算法初学者。对于已经有一定积累的同学，这本书可以帮助你系统回顾算法知识，源代码可被当作“刷题工具库”来使用。
- 书中内容主要分为复杂度分析、数据结构、算法三部分，覆盖了该领域的大部分主题。
- 对于算法小白，在初学阶段阅读一本入门书是非常有必要的，可以少走许多弯路。
- 书内的动画和图解往往介绍的是重点和难点知识，在阅读时应该多加关注。
- 实践是学习编程的最佳方式，强烈推荐运行源代码，动手敲代码。
- 本书提供了讨论区，遇到疑惑可以随时提问。

1. 引言

1.1. 算法无处不在

听到“算法”这个词，我们一般会联想到数学。但实际上，大多数算法并不包含复杂的数学，而更像是在考察基本逻辑，而这些逻辑在我们日常生活中处处可见。

在正式介绍算法之前，我想告诉你一件有趣的事：其实，**你在过去已经学会了很多算法，并且已经习惯将它们应用到日常生活中**。接下来，我将介绍两个具体例子来佐证。

例一：拼积木。一套积木，除了有许多部件之外，还会附送详细的拼装说明书。我们按照说明书上一步步操作，即可拼出复杂的积木模型。

如果从数据结构与算法的角度看，大大小小的「积木」就是数据结构，而「拼装说明书」上的一系列步骤就是算法。

例二：查字典。在字典中，每个汉字都有一个对应的拼音，而字典是按照拼音的英文字母表顺序排列的。假设需要在字典中查询任意一个拼音首字母为 *r* 的字，一般我们会这样做：

1. 打开字典大致一半页数的位置，查看此页的首字母是什么（假设为 *m*）；
2. 由于在英文字母表中 *r* 在 *m* 的后面，因此应排除字典前半部分，查找范围仅剩后半部分；
3. 循环执行步骤 1-2，直到找到拼音首字母为 *r* 的页码时终止。





Figure 1-1. 查字典步骤

查字典这个小学生的标配技能，实际上就是大名鼎鼎的「二分查找」。从数据结构角度，我们可以将字典看作是一个已排序的「数组」；而从算法角度，我们可将上述查字典的一系列指令看作是「二分查找」算法。

小到烹饪一道菜、大到星际航行，几乎所有问题的解决都离不开算法。计算机的出现，使我们可以通过编程将数据结构存储在内存中，也可以编写代码来调用 CPU, GPU 执行算法，从而将生活中的问题搬运到计算机中，更加高效地解决各式各样的复杂问题。



读到这里，如果你感到对数据结构、算法、数组、二分查找等此类概念一知半解，那么就太好了！因为这正是本书存在的价值，接下来，本书将会一步步地引导你进入数据结构与算法的知识殿堂。

1.2. 算法是什么

1.2.1. 算法定义

「算法 Algorithm」是在有限时间内解决特定问题的一组指令或操作步骤。算法具有以下特性：

- 问题是明确的，需要拥有明确的输入和输出定义。
- 解具有确定性，即给定相同输入时，输出一定相同。
- 具有可行性，可在有限步骤、有限时间、有限内存空间下完成。
- 独立于编程语言，即可用多种语言实现。

1.2.2. 数据结构定义

「数据结构 Data Structure」是在计算机中组织与存储数据的方式。为了提高数据存储和操作性能，数据结构的设计原则有：

- 空间占用尽可能小，节省计算机内存。
- 数据操作尽量快，包括数据访问、添加、删除、更新等。

- 提供简洁的数据表示和逻辑信息，以便算法高效运行。

数据结构的设计是一个充满权衡的过程，这意味着如果获得某方面的优势，则往往需要在另一方面做出妥协。例如，链表相对于数组，数据添加删除操作更加方便，但牺牲了数据的访问速度；图相对于链表，提供了更多的逻辑信息，但需要占用更多的内存空间。

1.2.3. 数据结构与算法的关系

「数据结构」与「算法」是高度相关、紧密嵌合的，体现在：

- 数据结构是算法的底座。数据结构为算法提供结构化存储的数据，以及操作数据的对应方法。
- 算法是数据结构发挥的舞台。数据结构仅存储数据信息，结合算法才可解决特定问题。
- 算法有对应最优的数据结构。给定算法，一般可基于不同的数据结构实现，而最终执行效率往往相差很大。



Figure 1-2. 数据结构与算法的关系

如果将「LEGO 乐高」类比到「数据结构与算法」，那么可以得到下表所示的对应关系。

数据结构与算法 LEGO 乐高	
输入数据	未拼装的积木
数据结构	积木组织形式，包括形状、大小、连接方式等
算法	把积木拼成目标形态的一系列操作步骤
输出数据	积木模型



约定俗成的简称

在实际讨论中，我们通常会将「数据结构与算法」直接简称为「算法」。例如，我们熟称的 LeetCode 算法题目，实际上同时考察了数据结构和算法两部分知识。

1.3. 小结

- 算法在生活中随处可见，并不高深莫测。我们已经不知不觉地学习到许多“算法”，用于解决生活中大大小小的问题。
- “查字典”的原理和二分查找算法一致。二分体现分而治之的重要算法思想。
- 算法是在有限时间内解决特定问题的一组指令或操作步骤，数据结构是在计算机中组织与存储数据的方式。
- 数据结构与算法两者紧密联系。数据结构是算法的底座，算法是发挥数据结构的舞台。
- 乐高积木对应数据，积木形状和连接形式对应数据结构，拼装积木的流程步骤对应算法。

2. 复杂度分析

2.1. 算法效率评估

2.1.1. 算法评价维度

在开始学习算法之前，我们首先要想清楚算法的设计目标是什么，或者说，如何来评判算法的好与坏。整体上看，我们设计算法时追求两个层面的目标。

1. **找到问题解法**。算法需要能够在规定的输入范围内，可靠地求得问题的正确解。
2. **寻求最优解法**。同一个问题可能存在多种解法，而我们希望算法效率尽可能的高。

换言之，在可以解决问题的前提下，算法效率则是主要评价维度，包括：

- **时间效率**，即算法的运行速度的快慢。
- **空间效率**，即算法占用的内存空间大小。

数据结构与算法追求“运行速度快、占用内存少”，而如何去评价算法效率则是非常重要的问题，因为只有知道如何评价算法，才能去做算法之间的对比分析，以及优化算法设计。

2.1.2. 效率评估方法

实际测试

假设我们现在有算法 A 和 算法 B，都能够解决同一问题，现在需要对比两个算法之间的效率。我们能够想到的最直接的方式，就是找一台计算机，把两个算法都完整跑一遍，并监控记录运行时间和内存占用情况。这种评估方式能够反映真实情况，但是也存在很大的硬伤。

难以排除测试环境的干扰因素。硬件配置会影响到算法的性能表现。例如，在某台计算机中，算法 A 比算法 B 运行时间更短；但换到另一台配置不同的计算机中，可能会得到相反的测试结果。这意味着我们需要在各种机器上展开测试，而这是不现实的。

展开完整测试非常耗费资源。随着输入数据量的大小变化，算法会呈现出不同的效率表现。比如，有可能输入数据量较小时，算法 A 运行时间短于算法 B，而在输入数据量较大时，测试结果截然相反。因此，若想要达到具有说服力的对比结果，那么需要输入各种体量数据，这样的测试需要占用大量计算资源。

理论估算

既然实际测试具有很大的局限性，那么我们是否可以仅通过一些计算，就获知算法的效率水平呢？答案是肯定的，我们将此估算方法称为「复杂度分析 Complexity Analysis」或「渐近复杂度分析 Asymptotic Complexity Analysis」。

复杂度分析评估的是算法运行效率随着输入数据量增多时的增长趋势。这句话有些拗口，我们可以将其分为三个重点来理解：

- “算法运行效率”可分为“运行时间”和“占用空间”，进而可将复杂度分为「时间复杂度 Time Complexity」和「空间复杂度 Space Complexity」。
- “随着输入数据量增多时”代表复杂度与输入数据量有关，反映算法运行效率与输入数据量之间的关系；
- “增长趋势”表示复杂度分析不关心算法具体使用了多少时间或占用了多少空间，而是给出一种“趋势性分析”；

复杂度分析克服了实际测试方法的弊端。一是独立于测试环境，分析结果适用于所有运行平台。二是可以体现不同数据量下的算法效率，尤其是可以反映大数据量下的算法性能。

如果感觉对复杂度分析的概念一知半解，无需担心，后续章节会展开介绍。

2.1.3. 复杂度分析重要性

复杂度分析给出一把评价算法效率的“标尺”，告诉我们执行某个算法需要多少时间和空间资源，也让我们可以开展不同算法之间的效率对比。

复杂度是个数学概念，对于初学者可能比较抽象，学习难度相对较高。从这个角度出发，其并不适合作为第一章内容。但是，当我们讨论某个数据结构或者算法的特点时，难以避免需要分析它的运行速度和空间使用情况。因此，在展开学习数据结构与算法之前，建议读者先对复杂度建立起初步的了解，并且能够完成简单案例的复杂度分析。

2.2. 时间复杂度

2.2.1. 统计算法运行时间

运行时间能够直观且准确地体现出算法的效率水平。如果我们想要准确预估一段代码的运行时间，该如何做呢？

1. 首先需要确定运行平台，包括硬件配置、编程语言、系统环境等，这些都会影响到代码的运行效率。
2. 评估各种计算操作的所需运行时间，例如加法操作`+`需要 1 ns，乘法操作`*`需要 10 ns，打印操作需要 5 ns 等。
3. 根据代码统计所有计算操作的数量，并将所有操作的执行时间求和，即可得到运行时间。

例如以下代码，输入数据大小为 n ，根据以上方法，可以得到算法运行时间为 $6n + 12$ ns。

$$1 + 1 + 10 + (1 + 5) \times n = 6n + 12$$

```
// 在某运行平台下
void algorithm(int n) {
    int a = 2; // 1 ns
    a = a + 1; // 1 ns
    a = a * 2; // 10 ns
    // 循环 n 次
    for (int i = 0; i < n; i++) { // 1 ns，每轮都要执行 i++
        // ...
    }
}
```

```
    cout << 0 << endl;      // 5 ns
}
}
```

但实际上，**统计算法的运行时间既不合理也不现实**。首先，我们不希望预估时间和运行平台绑定，毕竟算法需要跑在各式各样的平台之上。其次，我们很难获知每一种操作的运行时间，这为预估过程带来了极大的难度。

2.2.2. 统计时间增长趋势

「时间复杂度分析」采取了不同的做法，其统计的不是算法运行时间，而是**算法运行时间随着数据量变大时的增长趋势**。

“时间增长趋势”这个概念比较抽象，我们借助一个例子来理解。设输入数据大小为 n ，给定三个算法 A, B, C。

- 算法 A 只有 1 个打印操作，算法运行时间不随着 n 增大而增长。我们称此算法的时间复杂度为「常数阶」。
- 算法 B 中的打印操作需要循环 n 次，算法运行时间随着 n 增大成线性增长。此算法的时间复杂度被称为「线性阶」。
- 算法 C 中的打印操作需要循环 1000000 次，但运行时间仍与输入数据大小 n 无关。因此 C 的时间复杂度和 A 相同，仍为「常数阶」。

```
// 算法 A 时间复杂度：常数阶
void algorithm_A(int n) {
    cout << 0 << endl;
}

// 算法 B 时间复杂度：线性阶
void algorithm_B(int n) {
    for (int i = 0; i < n; i++) {
        cout << 0 << endl;
    }
}

// 算法 C 时间复杂度：常数阶
void algorithm_C(int n) {
    for (int i = 0; i < 1000000; i++) {
        cout << 0 << endl;
    }
}
```

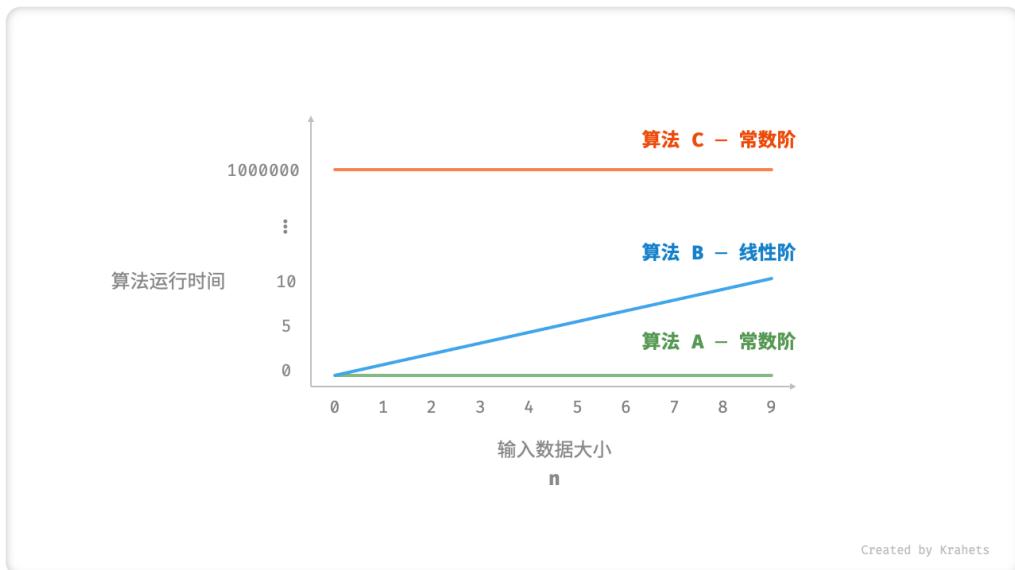


Figure 2-1. 算法 A, B, C 的时间增长趋势

相比直接统计算法运行时间，时间复杂度分析的做法有什么好处呢？以及有什么不足？

时间复杂度可以有效评估算法效率。算法 B 运行时间的增长是线性的，在 $n > 1$ 时慢于算法 A，在 $n > 1000000$ 时慢于算法 C。实质上，只要输入数据大小 n 足够大，复杂度为「常数阶」的算法一定优于「线性阶」的算法，这也正是时间增长趋势的含义。

时间复杂度的推算方法更加简便。在时间复杂度分析中，我们可以将统计「计算操作的运行时间」简化为统计「计算操作的数量」，这是因为，无论是运行平台还是计算操作类型，都与算法运行时间的增长趋势无关。因而，我们可以简单地将所有计算操作的执行时间统一看作是相同的“单位时间”，这样的简化做法大大降低了估算难度。

时间复杂度也存在一定的局限性。比如，虽然算法 A 和 C 的时间复杂度相同，但是实际的运行时间有非常大的差别。再比如，虽然算法 B 比 C 的时间复杂度要更高，但在输入数据大小 n 比较小时，算法 B 是要明显优于算法 C 的。对于以上情况，我们很难仅凭时间复杂度来判定算法效率高低。然而，即使存在这些问题，复杂度分析仍然是评判算法效率的最有效且常用的方法。

2.2.3. 函数渐近上界

设算法「计算操作数量」为 $T(n)$ ，其是一个关于输入数据大小 n 的函数。例如，以下算法的操作数量为

$$T(n) = 3 + 2n$$

```
void algorithm(int n) {
    int a = 1; // +1
    a = a + 1; // +1
    a = a * 2; // +1
    // 循环 n 次
}
```

```

for (int i = 0; i < n; i++) { // +1 (每轮都执行 i++)
    cout << 0 << endl; // +1
}
}

```

$T(n)$ 是一个一次函数，说明时间增长趋势是线性的，因此易得时间复杂度是线性阶。

我们将线性阶的时间复杂度记为 $O(n)$ ，这个数学符号被称为「大 O 记号 Big- O Notation」，代表函数 $T(n)$ 的「渐近上界 asymptotic upper bound」。

我们要推算时间复杂度，本质上是在计算「操作数量函数 $T(n)$ 」的渐近上界。下面我们先来看看函数渐近上界的数学定义。



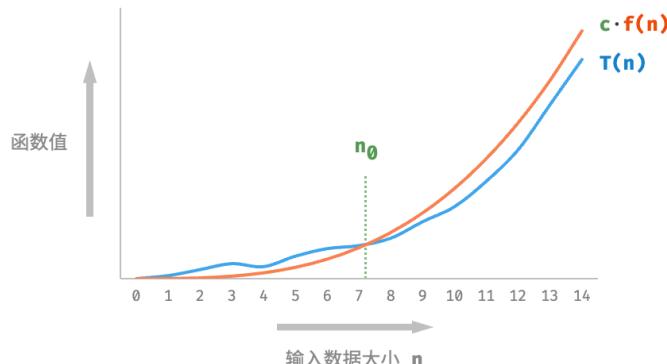
函数渐近上界

若存在正实数 c 和实数 n_0 ，使得对于所有的 $n > n_0$ ，均有

$$T(n) \leq c \cdot f(n)$$

则可认为 $f(n)$ 给出了 $T(n)$ 的一个渐近上界，记为

$$T(n) = O(f(n))$$



当 $n > n_0$ 时，始终有 $c \cdot f(n) \geq T(n)$ ，
因此可认为 $f(n)$ 是 $T(n)$ 的一个渐进上界，
记为 $T(n) = O(f(n))$

Created by Krahets

Figure 2-2. 函数的渐近上界

本质上看，计算渐近上界就是在找一个函数 $f(n)$ ，使得在 n 趋向于无穷大时， $T(n)$ 和 $f(n)$ 处于相同的增长级别（仅相差一个常数项 c 的倍数）。



渐近上界的数学味儿有点重，如果你感觉没有完全理解，无需担心，因为在实际使用中我们只需要会推算即可，数学意义可以慢慢领悟。

2.2.4. 推算方法

推算出 $f(n)$ 后，我们就得到时间复杂度 $O(f(n))$ 。那么，如何来确定渐近上界 $f(n)$ 呢？总体分为两步，首先「统计操作数量」，然后「判断渐近上界」。

1) 统计操作数量

对着代码，从上到下一行一行地计数即可。然而，由于上述 $c \cdot f(n)$ 中的常数项 c 可以取任意大小，因此操作数量 $T(n)$ 中的各种系数、常数项都可以被忽略。根据此原则，可以总结出以下计数偷懒技巧：

1. 跳过数量与 n 无关的操作。因为他们都是 $T(n)$ 中的常数项，对时间复杂度不产生影响。
2. 省略所有系数。例如，循环 $2n$ 次、 $5n + 1$ 次、……，都可以化简记为 n 次，因为 n 前面的系数对时间复杂度也不产生影响。
3. 循环嵌套时使用乘法。总操作数量等于外层循环和内层循环操作数量之积，每一层循环依然可以分别套用上述 1. 和 2. 技巧。

以下示例展示了使用上述技巧前、后的统计结果。

$$\begin{aligned} T(n) &= 2n(n + 1) + (5n + 1) + 2 && \text{完整统计 (-|||)} \\ &= 2n^2 + 7n + 3 \\ T(n) &= n^2 + n && \text{偷懒统计 (o.O)} \end{aligned}$$

最终，两者都能推出相同的时间复杂度结果，即 $O(n^2)$ 。

```
void algorithm(int n) {
    int a = 1; // +0 (技巧 1)
    a = a + n; // +0 (技巧 1)
    // +n (技巧 2)
    for (int i = 0; i < 5 * n + 1; i++) {
        cout << 0 << endl;
    }
    // +n*n (技巧 3)
    for (int i = 0; i < 2 * n; i++) {
        for (int j = 0; j < n + 1; j++) {
            cout << 0 << endl;
        }
    }
}
```

2) 判断渐近上界

时间复杂度由多项式 $T(n)$ 中最高阶的项来决定。这是因为在 n 趋于无穷大时，最高阶的项将处于主导作用，其它项的影响都可以被忽略。

以下表格给出了一些例子，其中有一些夸张的值，是想要向大家强调 系数无法撼动阶数 这一结论。在 n 趋于无穷大时，这些常数都是“浮云”。

操作数量 $T(n)$	时间复杂度 $O(f(n))$
100000	$O(1)$
$3n + 2$	$O(n)$
$2n^2 + 3n + 2$	$O(n^2)$
$n^3 + 10000n^2$	$O(n^3)$
$2^n + 10000n^{10000}$	$O(2^n)$

2.2.5. 常见类型

设输入数据大小为 n ，常见的时间复杂度类型有（从低到高排列）

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

常数阶 < 对数阶 < 线性阶 < 线性对数阶 < 平方阶 < 指数阶 < 阶乘阶

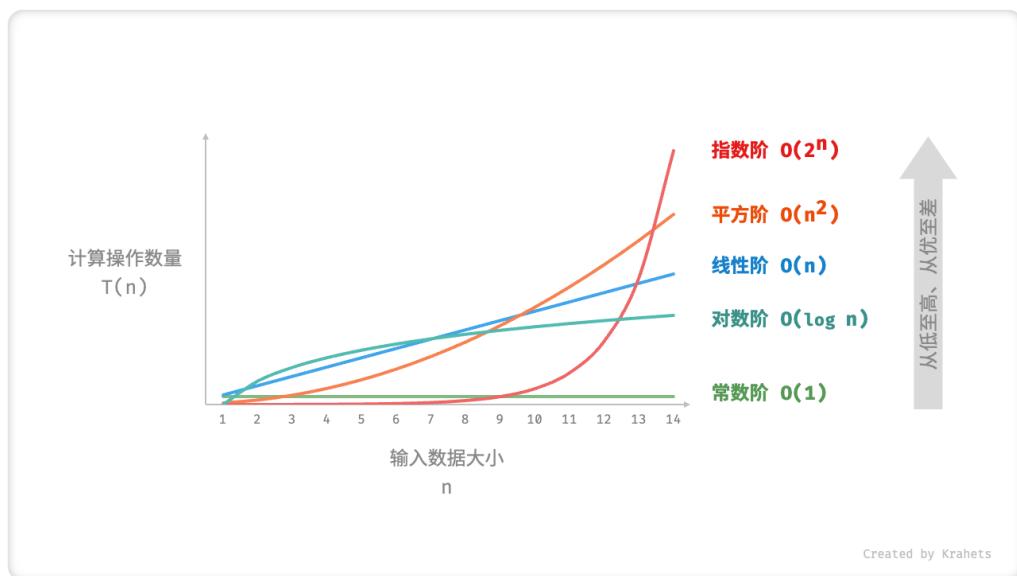


Figure 2-3. 时间复杂度的常见类型



部分示例代码需要一些前置知识，包括数组、递归算法等。如果遇到看不懂的地方无需担心，可以在学习完后面章节后再来复习，现阶段先聚焦在理解时间复杂度含义和推算方法上。

常数阶 $O(1)$

常数阶的操作数量与输入数据大小 n 无关，即不随着 n 的变化而变化。

对于以下算法，无论操作数量 `size` 有多大，只要与数据大小 n 无关，时间复杂度就仍为 $O(1)$ 。

```
// === File: time_complexity.cpp ===
/* 常数阶 */
int constant(int n) {
    int count = 0;
    int size = 100000;
    for (int i = 0; i < size; i++)
        count++;
    return count;
}
```

线性阶 $O(n)$

线性阶的操作数量相对输入数据大小成线性级别增长。线性阶常出现于单层循环。

```
// === File: time_complexity.cpp ===
/* 线性阶 */
int linear(int n) {
    int count = 0;
    for (int i = 0; i < n; i++)
        count++;
    return count;
}
```

「遍历数组」和「遍历链表」等操作，时间复杂度都为 $O(n)$ ，其中 n 为数组或链表的长度。



数据大小 n 是根据输入数据的类型来确定的。比如，在上述示例中，我们直接将 n 看作输入数据大小；以下遍历数组示例中，数据大小 n 为数组的长度。

```
// === File: time_complexity.cpp ===
/* 线性阶（遍历数组） */
int arrayTraversal(vector<int>& nums) {
```

```
int count = 0;
// 循环次数与数组长度成正比
for (int num : nums) {
    count++;
}
return count;
}
```

平方阶 $O(n^2)$

平方阶的操作数量相对输入数据大小成平方级别增长。平方阶常出现于嵌套循环，外层循环和内层循环都为 $O(n)$ ，总体为 $O(n^2)$ 。

```
// === File: time_complexity.cpp ===
/* 平方阶 */
int quadratic(int n) {
    int count = 0;
    // 循环次数与数组长度成平方关系
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            count++;
        }
    }
    return count;
}
```



Figure 2-4. 常数阶、线性阶、平方阶的时间复杂度

以「冒泡排序」为例，外层循环 $n - 1$ 次，内层循环 $n - 1, n - 2, \dots, 2, 1$ 次，平均为 $\frac{n}{2}$ 次，因此时间复杂度为 $O(n^2)$ 。

$$O((n-1)\frac{n}{2}) = O(n^2)$$

```
// === File: time_complexity.cpp ===
/* 平方阶（冒泡排序） */
int bubbleSort(vector<int>& nums) {
    int count = 0; // 计数器
    // 外循环：待排序元素数量为 n-1, n-2, ..., 1
    for (int i = nums.size() - 1; i > 0; i--) {
        // 内循环：冒泡操作
        for (int j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // 交换 nums[j] 与 nums[j + 1]
                int tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
                count += 3; // 元素交换包含 3 个单元操作
            }
        }
    }
    return count;
}
```

指数阶 $O(2^n)$



生物学科中的“细胞分裂”即是指数阶增长：初始状态为 1 个细胞，分裂一轮后为 2 个，分裂两轮后为 4 个，……，分裂 n 轮后有 2^n 个细胞。

指数阶增长得非常快，在实际应用中一般是不能被接受的。若一个问题使用「暴力枚举」求解的时间复杂度是 $O(2^n)$ ，那么一般都需要使用「动态规划」或「贪心算法」等算法来求解。

```
// === File: time_complexity.cpp ===
/* 指数阶（循环实现） */
int exponential(int n) {
    int count = 0, base = 1;
    // cell 每轮一分为二，形成数列 1, 2, 4, 8, ..., 2^(n-1)
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < base; j++) {
            count++;
        }
        base *= 2;
    }
}
```

```

    }
    base *= 2;
}
// count = 1 + 2 + 4 + 8 + .. + 2^(n-1) = 2^n - 1
return count;
}

```

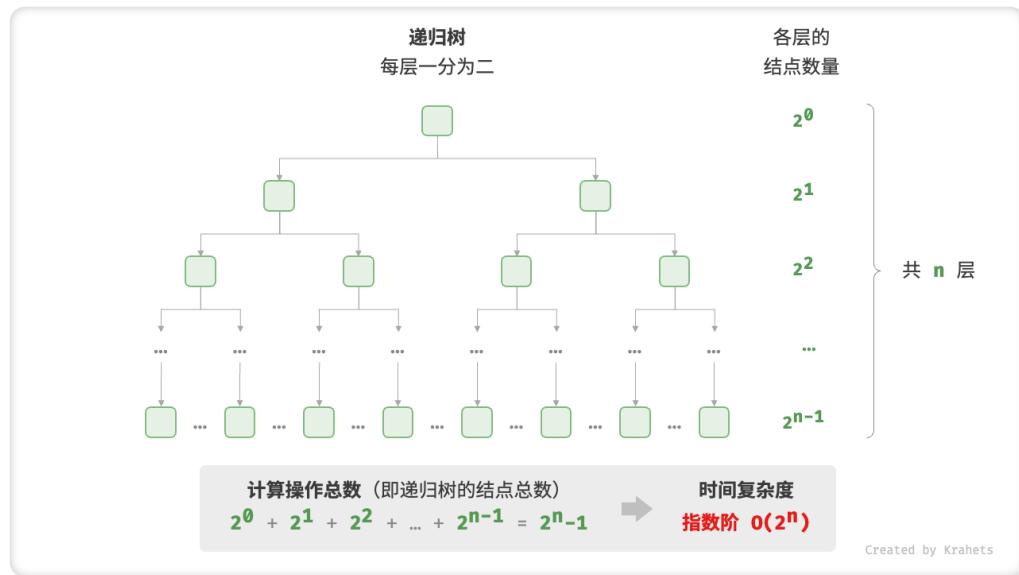


Figure 2-5. 指数阶的时间复杂度

在实际算法中，指数阶常出现于递归函数。例如以下代码，不断地一分为二，分裂 n 次后停止。

```

// === File: time_complexity.cpp ===
/* 指数阶 (递归实现) */
int expRecur(int n) {
    if (n == 1) return 1;
    return expRecur(n - 1) + expRecur(n - 1) + 1;
}

```

对数阶 $O(\log n)$

对数阶与指数阶正好相反，后者反映“每轮增加到两倍的情况”，而前者反映“每轮缩减到一半的情况”。对数阶仅次于常数阶，时间增长得很慢，是理想的时间复杂度。

对数阶常出现于「二分查找」和「分治算法」中，体现“一分为多”、“化繁为简”的算法思想。

设输入数据大小为 n ，由于每轮缩减到一半，因此循环次数是 $\log_2 n$ ，即 2^n 的反函数。

```
// === File: time_complexity.cpp ===
/* 对数阶（循环实现） */
int logarithmic(float n) {
    int count = 0;
    while (n > 1) {
        n = n / 2;
        count++;
    }
    return count;
}
```

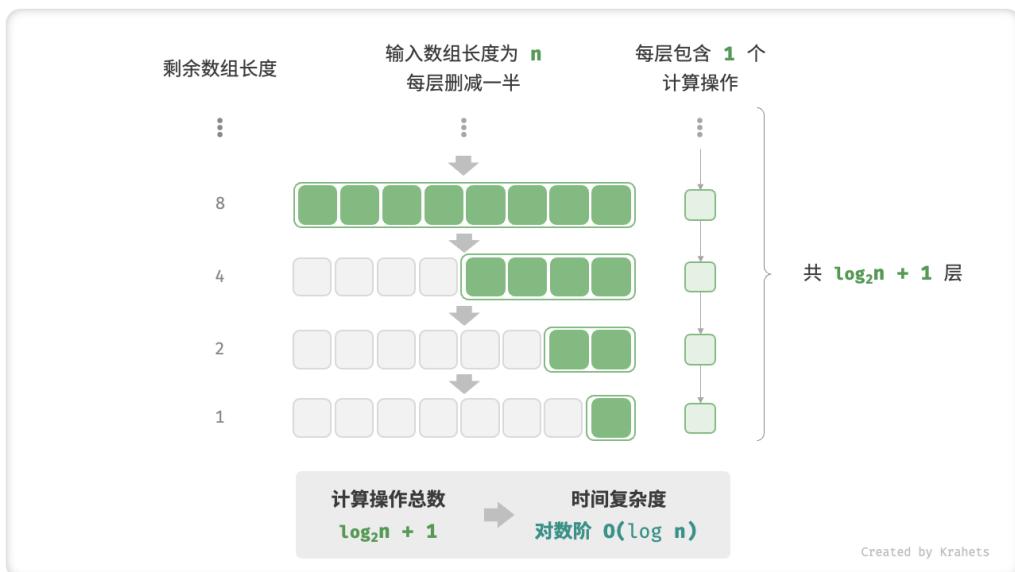


Figure 2-6. 对数阶的时间复杂度

与指数阶类似，对数阶也常出现于递归函数。以下代码形成了一个高度为 $\log_2 n$ 的递归树。

```
// === File: time_complexity.cpp ===
/* 对数阶（递归实现） */
int logRecur(float n) {
    if (n <= 1) return 0;
    return logRecur(n / 2) + 1;
}
```

线性对数阶 $O(n \log n)$

线性对数阶常出现于嵌套循环中，两层循环的时间复杂度分别为 $O(\log n)$ 和 $O(n)$ 。

主流排序算法的时间复杂度都是 $O(n \log n)$ ，例如快速排序、归并排序、堆排序等。

```
// === File: time_complexity.cpp ===
/* 线性对数阶 */
int linearLogRecur(float n) {
    if (n <= 1) return 1;
    int count = linearLogRecur(n / 2) +
                linearLogRecur(n / 2);
    for (int i = 0; i < n; i++) {
        count++;
    }
    return count;
}
```

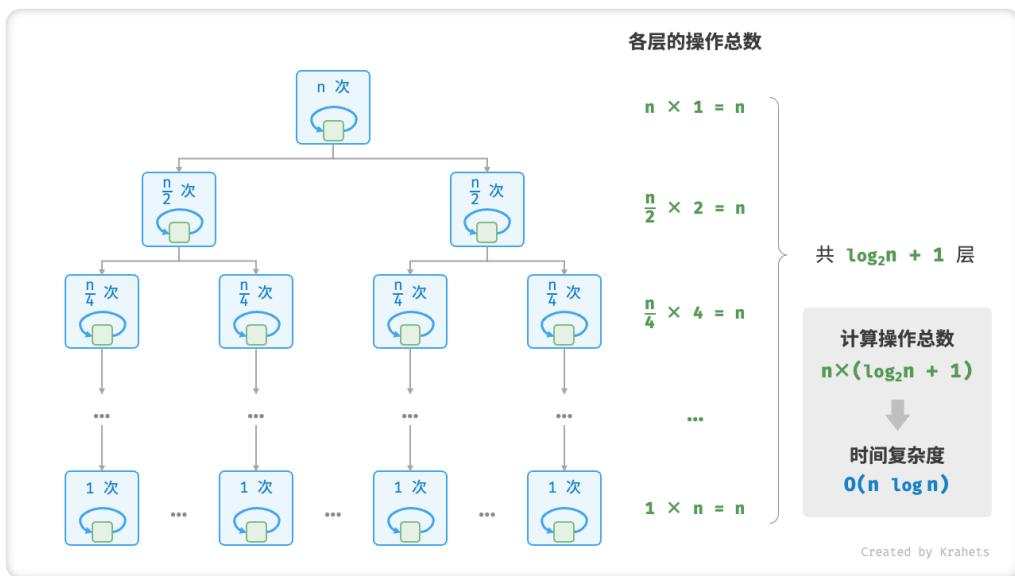


Figure 2-7. 线性对数阶的时间复杂度

阶乘阶 $O(n!)$

阶乘阶对应数学上的「全排列」。即给定 n 个互不重复的元素，求其所有可能的排列方案，则方案数量为

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$$

阶乘常使用递归实现。例如以下代码，第一层分裂出 n 个，第二层分裂出 $n - 1$ 个，……，直至到第 n 层时终止分裂。

```
// === File: time_complexity.cpp ===
/* 阶乘阶（递归实现） */
int factorialRecur(int n) {
    if (n == 0) return 1;
    int count = 0;
```

```
// 从 1 个分裂出 n 个
for (int i = 0; i < n; i++) {
    count += factorialRecur(n - 1);
}
return count;
}
```

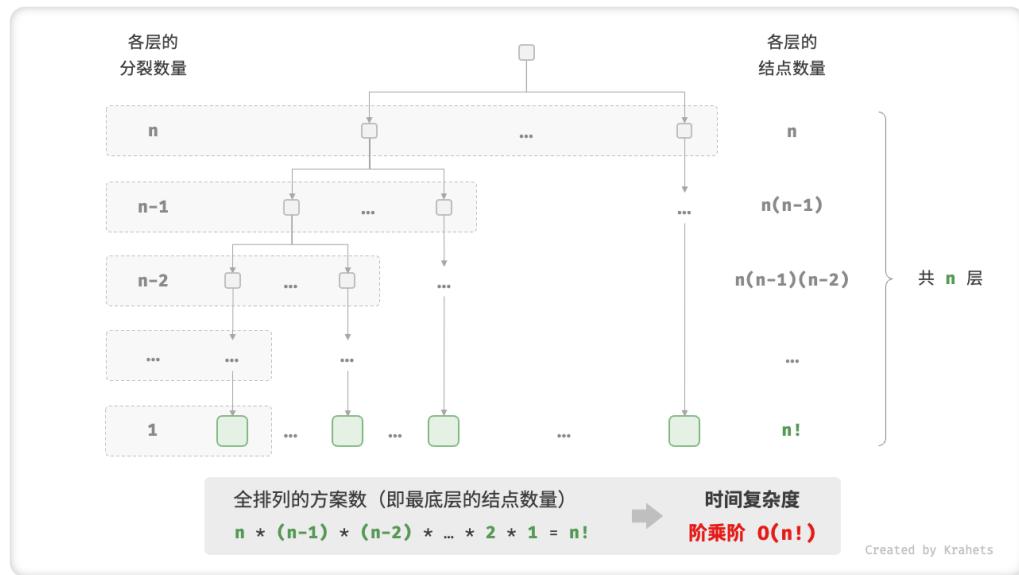


Figure 2-8. 阶乘阶的时间复杂度

2.2.6. 最差、最佳、平均时间复杂度

某些算法的时间复杂度不是恒定的，而是与输入数据的分布有关。举一个例子，输入一个长度为 n 数组 `nums`，其中 `nums` 由从 1 至 n 的数字组成，但元素顺序是随机打乱的；算法的任务是返回元素 1 的索引。我们可以得出以下结论：

- 当 `nums = [?, ?, ..., 1]`，即当末尾元素是 1 时，则需完整遍历数组，此时达到 **最差时间复杂度** $O(n)$ ；
- 当 `nums = [1, ?, ?, ...]`，即当首个数字为 1 时，无论数组多长都不需要继续遍历，此时达到 **最佳时间复杂度** $\Omega(1)$ ；

「函数渐近上界」使用大 O 记号表示，代表「最差时间复杂度」。与之对应，「函数渐近下界」用 Ω 记号 (Omega Notation) 来表示，代表「最佳时间复杂度」。

```
// === File: worst_best_time_complexity.cpp ===
/* 生成一个数组，元素为 { 1, 2, ..., n }，顺序被打乱 */
vector<int> randomNumbers(int n) {
    vector<int> nums(n);
    // 生成数组 nums = { 1, 2, 3, ..., n }
```

```
for (int i = 0; i < n; i++) {
    nums[i] = i + 1;
}
// 使用系统时间生成随机种子
unsigned seed = chrono::system_clock::now().time_since_epoch().count();
// 随机打乱数组元素
shuffle(nums.begin(), nums.end(), default_random_engine(seed));
return nums;
}

/* 查找数组 nums 中数字 1 所在索引 */
int findOne(vector<int>& nums) {
    for (int i = 0; i < nums.size(); i++) {
        // 当元素 1 在数组头部时，达到最佳时间复杂度 O(1)
        // 当元素 1 在数组尾部时，达到最差时间复杂度 O(n)
        if (nums[i] == 1)
            return i;
    }
    return -1;
}
```



我们在实际应用中很少使用「最佳时间复杂度」，因为往往只有很小概率下才能达到，会带来一定的误导性。反之，「最差时间复杂度」最为实用，因为它给出了一个“效率安全值”，让我们可以放心地使用算法。

从上述示例可以看出，最差或最佳时间复杂度只出现在“特殊分布的数据”中，这些情况的出现概率往往很小，因此并不能最真实地反映算法运行效率。相对地，「平均时间复杂度」可以体现算法在随机输入数据下的运行效率，用 Θ 记号（Theta Notation）来表示。

对于部分算法，我们可以简单地推算出随机数据分布下的平均情况。比如上述示例，由于输入数组是被打乱的，因此元素 1 出现在任意索引的概率都是相等的，那么算法的平均循环次数则是数组长度的一半 $\frac{n}{2}$ ，平均时间复杂度为 $\Theta(\frac{n}{2}) = \Theta(n)$ 。

但在实际应用中，尤其是较为复杂的算法，计算平均时间复杂度比较困难，因为很难简便地分析出在数据分布下的整体数学期望。这种情况下，我们一般使用最差时间复杂度来作为算法效率的评判标准。



为什么很少看到 Θ 符号？

实际中我们经常使用「大 O 符号」来表示「平均复杂度」，这样严格意义上来说是不规范的。这可能是因为 O 符号实在是太朗朗上口了。如果在本书和其他资料中看到类似 平均时间复杂度 $O(n)$ 的表述，请你直接理解为 $\Theta(n)$ 即可。

2.3. 空间复杂度

「空间复杂度 Space Complexity」统计 算法使用内存空间随着数据量变大时的增长趋势。这个概念与时间复杂度很类似。

2.3.1. 算法相关空间

算法运行中，使用的内存空间主要有以下几种：

- 「输入空间」用于存储算法的输入数据；
- 「暂存空间」用于存储算法运行中的变量、对象、函数上下文等数据；
- 「输出空间」用于存储算法的输出数据；



通常情况下，空间复杂度统计范围是「暂存空间」 + 「输出空间」。

暂存空间可分为三个部分：

- 「暂存数据」用于保存算法运行中的各种 **常量、变量、对象** 等。
- 「栈帧空间」用于保存调用函数的上下文数据。系统每次调用函数都会在栈的顶部创建一个栈帧，函数返回时，栈帧空间会被释放。
- 「指令空间」用于保存编译后的程序指令，在实际统计中一般忽略不计。



Figure 2-9. 算法使用的相关空间

```
/* 结构体 */
struct Node {
```

```

int val;
Node *next;
Node(int x) : val(x), next(nullptr) {}

};

/* 函数 */
int func() {
    // do something...
    return 0;
}

int algorithm(int n) {          // 输入数据
    const int a = 0;            // 暂存数据（常量）
    int b = 0;                 // 暂存数据（变量）
    Node* node = new Node(0);   // 暂存数据（对象）
    int c = func();            // 栈帧空间（调用函数）
    return a + b + c;          // 输出数据
}

```

2.3.2. 推算方法

空间复杂度的推算方法和时间复杂度总体类似，只是从统计“计算操作数量”变为统计“使用空间大小”。与时间复杂度不同的是，**我们一般只关注「最差空间复杂度」**。这是因为内存空间是一个硬性要求，我们必须保证在所有输入数据下都有足够的内存空间预留。

最差空间复杂度中的“最差”有两层含义，分别为输入数据的最差分布、算法运行中的最差时间点。

- **以最差输入数据为准**。当 $n < 10$ 时，空间复杂度为 $O(1)$ ；但是当 $n > 10$ 时，初始化的数组 `nums` 使用 $O(n)$ 空间；因此最差空间复杂度为 $O(n)$ ；
- **以算法运行过程中的峰值内存为准**。程序在执行最后一行之前，使用 $O(1)$ 空间；当初始化数组 `nums` 时，程序使用 $O(n)$ 空间；因此最差空间复杂度为 $O(n)$ ；

```

void algorithm(int n) {
    int a = 0;                  // O(1)
    vector<int> b(10000);      // O(1)
    if (n > 10)
        vector<int> nums(n); // O(n)
}

```

在递归函数中，需要注意统计栈帧空间。例如函数 `loop()`，在循环中调用了 n 次 `function()`，每轮中的 `function()` 都返回并释放了栈帧空间，因此空间复杂度仍为 $O(1)$ 。而递归函数 `recur()` 在运行中会同时存在 n 个未返回的 `recur()`，从而使用 $O(n)$ 的栈帧空间。

```

int func() {
    // do something
    return 0;
}

/* 循环 O(1) */
void loop(int n) {
    for (int i = 0; i < n; i++) {
        func();
    }
}

/* 递归 O(n) */
void recur(int n) {
    if (n == 1) return;
    return recur(n - 1);
}

```

2.3.3. 常见类型

设输入数据大小为 n ，常见的空间复杂度类型有（从低到高排列）

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

常数阶 < 对数阶 < 线性阶 < 平方阶 < 指数阶

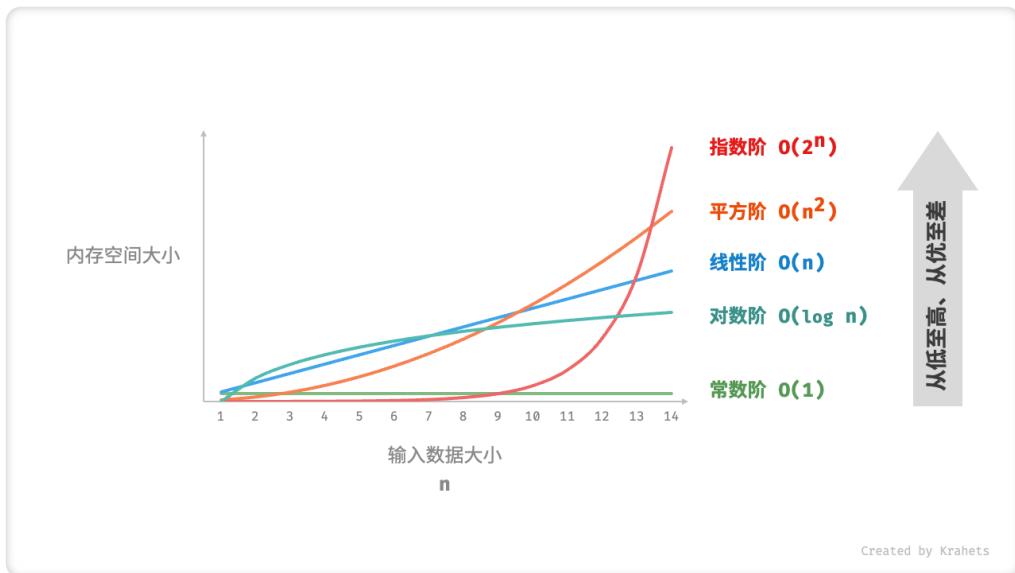


Figure 2-10. 空间复杂度的常见类型



部分示例代码需要一些前置知识，包括数组、链表、二叉树、递归算法等。如果遇到看不懂的地方无需担心，可以在学习完后面章节后再来复习，现阶段先聚焦在理解空间复杂度含义和推算方法上。

常数阶 $O(1)$

常数阶常见于数量与输入数据大小 n 无关的常量、变量、对象。

需要注意的是，在循环中初始化变量或调用函数而占用的内存，在进入下一循环后就会被释放，即不会累积占用空间，空间复杂度仍为 $O(1)$ 。

```
// === File: space_complexity.cpp ===
/* 常数阶 */
void constant(int n) {
    // 常量、变量、对象占用 O(1) 空间
    const int a = 0;
    int b = 0;
    vector<int> nums(10000);
    ListNode node(0);
    // 循环中的变量占用 O(1) 空间
    for (int i = 0; i < n; i++) {
        int c = 0;
    }
    // 循环中的函数占用 O(1) 空间
    for (int i = 0; i < n; i++) {
        func();
    }
}
```

线性阶 $O(n)$

线性阶常见于元素数量与 n 成正比的数组、链表、栈、队列等。

```
// === File: space_complexity.cpp ===
/* 线性阶 */
void linear(int n) {
    // 长度为 n 的数组占用 O(n) 空间
    vector<int> nums(n);
    // 长度为 n 的列表占用 O(n) 空间
    vector<ListNode> nodes;
    for (int i = 0; i < n; i++) {
        nodes.push_back(ListNode(i));
    }
}
```

```
// 长度为 n 的哈希表占用 O(n) 空间
unordered_map<int, string> map;
for (int i = 0; i < n; i++) {
    map[i] = to_string(i);
}
}
```

以下递归函数会同时存在 n 个未返回的 `algorithm()` 函数，使用 $O(n)$ 大小的栈帧空间。

```
// === File: space_complexity.cpp ===
/* 线性阶（递归实现） */
void linearRecur(int n) {
    cout << " 递归 n = " << n << endl;
    if (n == 1) return;
    linearRecur(n - 1);
}
```

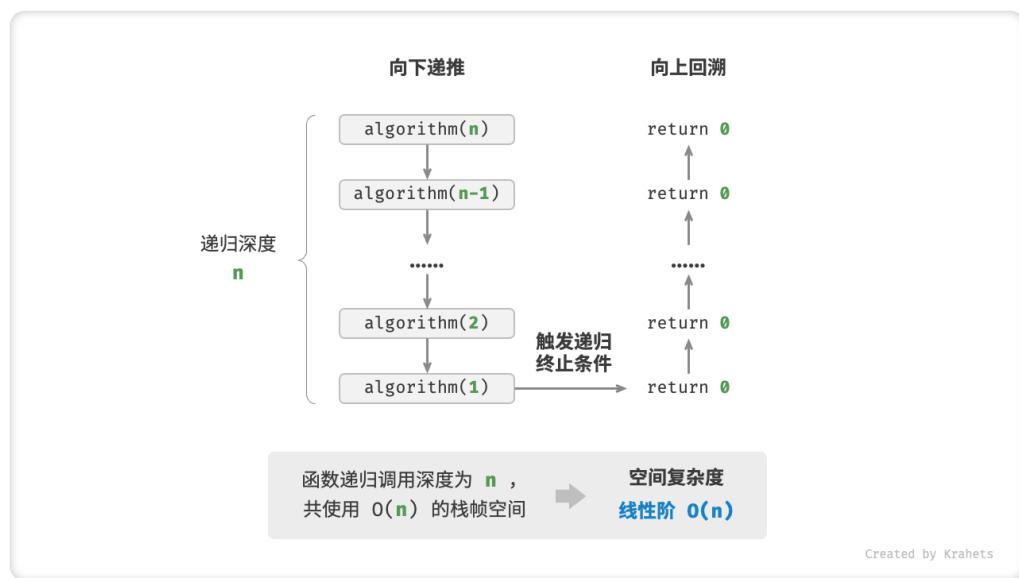


Figure 2-11. 递归函数产生的线性阶空间复杂度

平方阶 $O(n^2)$

平方阶常见于元素数量与 n 成平方关系的矩阵、图。

```
// === File: space_complexity.cpp ===
/* 平方阶 */
void quadratic(int n) {
    // 二维列表占用  $O(n^2)$  空间
    vector<vector<int>> numMatrix;
```

```

for (int i = 0; i < n; i++) {
    vector<int> tmp;
    for (int j = 0; j < n; j++) {
        tmp.push_back(0);
    }
    numMatrix.push_back(tmp);
}
}

```

在以下递归函数中，同时存在 n 个未返回的 `algorithm()`，并且每个函数中都初始化了一个数组，长度分别为 $n, n - 1, n - 2, \dots, 2, 1$ ，平均长度为 $\frac{n}{2}$ ，因此总体使用 $O(n^2)$ 空间。

```

// === File: space_complexity.cpp ===
/* 平方阶（递归实现） */
int quadraticRecur(int n) {
    if (n <= 0) return 0;
    vector<int> nums(n);
    cout << " 递归 n = " << n << " 中的 nums 长度 = " << nums.size() << endl;
    return quadraticRecur(n - 1);
}

```

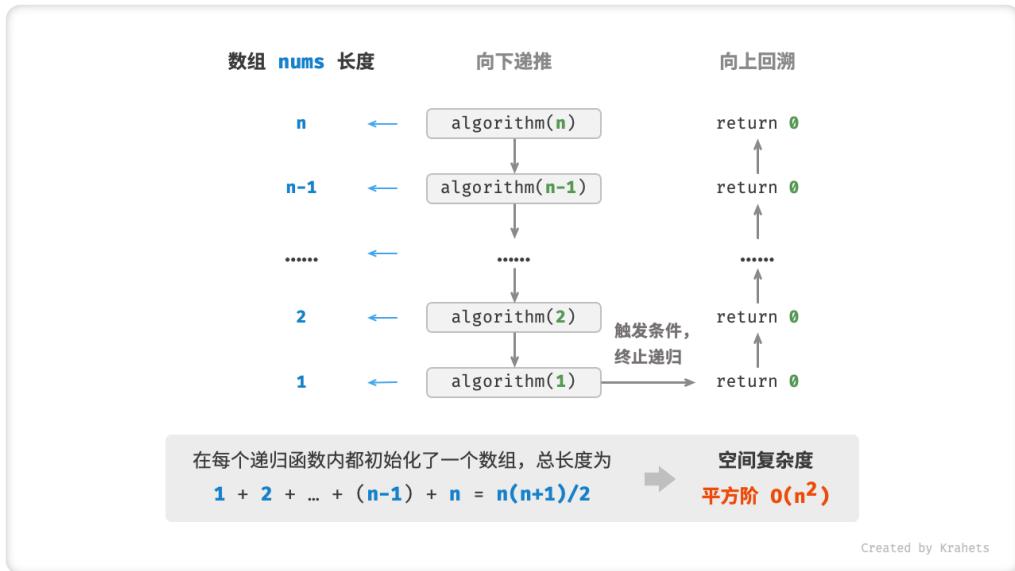


Figure 2-12. 递归函数产生的平方阶空间复杂度

指数阶 $O(2^n)$

指数阶常见于二叉树。高度为 n 的「满二叉树」的结点数量为 $2^n - 1$ ，使用 $O(2^n)$ 空间。

```
// === File: space_complexity.cpp ===
/* 指数阶 (建立满二叉树) */
TreeNode* buildTree(int n) {
    if (n == 0) return nullptr;
    TreeNode* root = new TreeNode(0);
    root->left = buildTree(n - 1);
    root->right = buildTree(n - 1);
    return root;
}
```

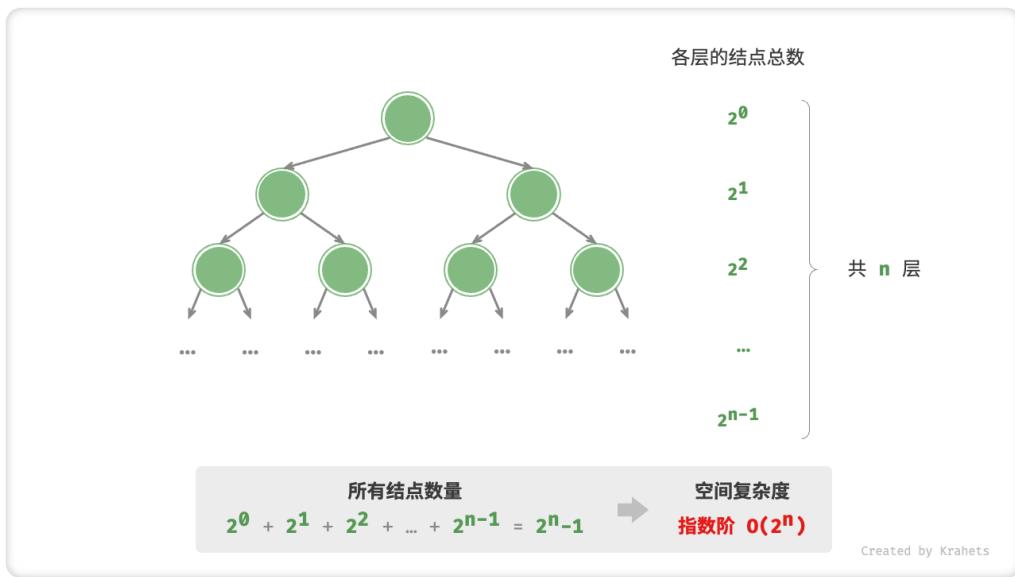


Figure 2-13. 满二叉树产生的指数阶空间复杂度

对数阶 $O(\log n)$

对数阶常见于分治算法、数据类型转换等。

例如「归并排序」，长度为 n 的数组可以形成高度为 $\log n$ 的递归树，因此空间复杂度为 $O(\log n)$ 。

再例如「数字转化为字符串」，输入任意正整数 n ，它的位数为 $\log_{10} n$ ，即对应字符串长度为 $\log_{10} n$ ，因此空间复杂度为 $O(\log_{10} n) = O(\log n)$ 。

2.4. 权衡时间与空间

理想情况下，我们希望算法的时间复杂度和空间复杂度都能够达到最优，而实际上，同时优化时间复杂度和空间复杂度是非常困难的。

降低时间复杂度，往往是以提升空间复杂度为代价的，反之亦然。我们把牺牲内存空间来提升算法运行速度的思路称为「以空间换时间」；反之，称之为「以时间换空间」。选择哪种思路取决于我们更看重哪个方面。

大多数情况下，时间都是比空间更宝贵的，只要空间复杂度不要太离谱、能接受就行，因此以空间换时间最为常用。

2.4.1. 示例题目 *

以 LeetCode 全站第一题 [两数之和](#) 为例。



两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`，请你在该数组中找出“和”为目标值 `target` 的那两个整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。

你可以按任意顺序返回答案。

「暴力枚举」和「辅助哈希表」分别对应 **空间最优** 和 **时间最优** 的两种解法。本着时间比空间更宝贵的原则，后者是本题的最佳解法。

方法一：暴力枚举

考虑直接遍历所有所有可能性。通过开启一个两层循环，判断两个整数的和是否为 `target`，若是则返回它俩的索引（即下标）即可。

```
// === File: leetcode_two_sum.cpp ===
/* 方法一：暴力枚举 */
vector<int> twoSumBruteForce(vector<int>& nums, int target) {
    int size = nums.size();
    // 两层循环，时间复杂度 O(n^2)
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (nums[i] + nums[j] == target)
                return {i, j};
        }
    }
    return {};
}
```

该方法的时间复杂度为 $O(N^2)$ ，空间复杂度为 $O(1)$ ，属于**时间换空间**。本方法时间复杂度较高，在大数据量下非常耗时。

方法二：辅助哈希表

考虑借助一个哈希表，`key` 为数组元素、`value` 为元素索引。循环遍历数组中的每个元素 `num`，并执行：

1. 判断数字 `target - num` 是否在哈希表中，若是则直接返回该两个元素的索引；
2. 将元素 `num` 和其索引添加进哈希表；

```
// === File: leetcode_two_sum.cpp ===
/* 方法二：辅助哈希表 */
vector<int> twoSumHashTable(vector<int>& nums, int target) {
    int size = nums.size();
    // 辅助哈希表，空间复杂度 O(n)
    unordered_map<int, int> dic;
    // 单层循环，时间复杂度 O(n)
    for (int i = 0; i < size; i++) {
        if (dic.find(target - nums[i]) != dic.end()) {
            return {dic[target - nums[i]], i};
        }
        dic.emplace(nums[i], i);
    }
    return {};
}
```

该方法的时间复杂度为 $O(N)$ ，空间复杂度为 $O(N)$ ，体现空间换时间。本方法虽然引入了额外空间使用，但时间和空间使用整体更加均衡，因此为本题最优解法。

2.5. 小结

算法效率评估

- 「时间效率」和「空间效率」是算法性能的两个重要的评价维度。
- 我们可以通过「实际测试」来评估算法效率，但难以排除测试环境的干扰，并且非常耗费计算资源。
- 「复杂度分析」克服了实际测试的弊端，分析结果适用于所有运行平台，并且可以体现不同数据大小下的算法效率。

时间复杂度

- 「时间复杂度」统计算法运行时间随着数据量变大时的增长趋势，可以有效评估算法效率，但在某些情况下可能失效，比如在输入数据量较小或时间复杂度相同时，无法精确对比算法效率的优劣性。
- 「最差时间复杂度」使用大 O 符号表示，即函数渐近上界，其反映当 n 趋于正无穷时， $T(n)$ 处于何种增长级别。
- 推算时间复杂度分为两步，首先统计计算操作数量，再判断渐近上界。
- 常见时间复杂度从小到大排列有 $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n), O(n!)$ 。
- 某些算法的时间复杂度不是恒定的，而是与输入数据的分布有关。时间复杂度分为「最差时间复杂度」和「最佳时间复杂度」，后者几乎不用，因为输入数据需要满足苛刻的条件才能达到最佳情况。
- 「平均时间复杂度」可以反映在随机数据输入下的算法效率，最贴合实际使用情况下的算法性能。计算平均时间复杂度需要统计输入数据的分布，以及综合后的数学期望。

空间复杂度

- 与时间复杂度的定义类似，「空间复杂度」统计算法占用空间随着数据量变大时的增长趋势。
- 算法运行中相关内存空间可分为输入空间、暂存空间、输出空间。通常情况下，输入空间不计入空间复杂度计算。暂存空间可分为指令空间、数据空间、栈帧空间，其中栈帧空间一般在递归函数中才会影响到空间复杂度。
- 我们一般只关心「最差空间复杂度」，即统计算法在「最差输入数据」和「最差运行时间点」下的空间复杂度。
- 常见空间复杂度从小到大排列有 $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$, $O(2^n)$ 。

3. 数据结构简介

3.1. 数据与内存

3.1.1. 基本数据类型

谈到计算机中的数据，我们能够想到文本、图片、视频、语音、3D 模型等等，这些数据虽然组织形式不同，但都是由各种基本数据类型构成的。

「基本数据类型」是 CPU 可以直接进行运算的类型，在算法中直接被使用。

- 「整数」根据不同的长度分为 byte, short, int, long，根据算法需求选用，即在满足取值范围的情况下尽量减小内存空间占用；
- 「浮点数」代表小数，根据长度分为 float, double，同样根据算法的实际需求选用；
- 「字符」在计算机中是以字符集的形式保存的，char 的值实际上是数字，代表字符集中的编号，计算机通过字符集查表来完成编号到字符的转换。占用空间与具体编程语言有关，通常为 2 bytes 或 1 byte；
- 「布尔」代表逻辑中的“是”与“否”，其占用空间需要根据编程语言确定，通常为 1 byte 或 1 bit；

类别	符号	占用空间	取值范围	默认值
整数	byte	1 byte	$-2^7 \sim 2^7 - 1 (-128 \sim 127)$	0
	short	2 bytes	$-2^{15} \sim 2^{15} - 1$	0
	int	4 bytes	$-2^{31} \sim 2^{31} - 1$	0
	long	8 bytes	$-2^{63} \sim 2^{63} - 1$	0
浮点数	float	4 bytes	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$	0.0 f
	double	8 bytes	$-1.7 \times 10^{308} \sim 1.7 \times 10^{308}$	0.0
字符	char	2 bytes / 1 byte	$0 \sim 2^{16} - 1$	0
布尔	bool	1 byte / 1 bit	true 或 false	false



以上表格中，加粗项在「算法题」中最为常用。此表格无需硬背，大致理解即可，需要时可以通过查表来回忆。

整数表示方式

整数的取值范围取决于变量使用的内存长度，即字节（或比特）数。在计算机中，1 字节 (byte) = 8 比特 (bit)，1 比特即 1 个二进制位。以 int 类型为例：

1. 整数类型 int 占用 4 bytes = 32 bits，因此可以表示 2^{32} 个不同的数字；
2. 将最高位看作符号位，0 代表正数，1 代表负数，从而可以表示 2^{31} 个正数和 2^{31} 个负数；
3. 当所有 bits 为 0 时代表数字 0，从零开始增大，可得最大正数为 $2^{31} - 1$ ；
4. 剩余 2^{31} 个数字全部用来表示负数，因此最小负数为 -2^{31} ；具体细节涉及到“源码、反码、补码”知识，有兴趣的同学可以查阅学习；

其它整数类型 byte, short, long 取值范围的计算方法与 int 类似，在此不再赘述。

浮点数表示方式 *



在本书中，标题后的 * 符号代表选读章节，如果你觉得理解困难，建议先跳过，等学完必读章节后续再单独攻克。

细心的你可能会疑惑：int 和 float 长度相同，都是 4 bytes，但为什么 float 的取值范围远大于 int？按说 float 需要表示小数，取值范围应该变小才对。

其实，这是因为浮点数 float 采用了不同的表示方式。IEEE 754 标准规定，32-bit 长度的 float 由以下部分构成：

- 符号位 S：占 1 bit；
- 指数位 E：占 8 bits；
- 分数位 N：占 24 bits，其中 23 位显式存储；

设 32-bit 二进制数的第 i 位为 b_i ，则 float 值的计算方法定义为

$$\text{val} = (-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

转化到十进制下的计算公式为

$$\text{val} = (-1)^S \times 2^{E-127} \times (1 + N)$$

其中各项的取值范围为

$$S \in \{0, 1\}, \quad E \in \{1, 2, \dots, 254\}$$

$$(1 + N) = (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) \subset [1, 2 - 2^{-23}]$$



Figure 3-1. IEEE 754 标准下的 float 表示方式

以上图为例， $S = 0$ ， $E = 124$ ， $N = 2^{-2} + 2^{-3} = 0.375$ ，易得

$$\mathbf{val} = (-1)^0 \times 2^{124-127} \times (1 + 0.375) = 0.171875$$

现在我们可以回答开始的问题：**float 的表示方式包含指数位，导致其取值范围远大于 int**。根据以上计算，float 可表示的最大正数为 $2^{254-127} \times (2 - 2^{-23}) \approx 3.4 \times 10^{38}$ ，切换符号位便可得到最小负数。

浮点数 float 虽然拓展了取值范围，但副作用是牺牲了精度。整数类型 int 将全部 32 位用于表示数字，数字是均匀分布的；而由于指数位的存在，浮点数 float 的数值越大，相邻两个数字之间的差值就会趋向越大。

进一步地，指数位 $E = 0$ 和 $E = 255$ 具有特殊含义，用于表示零、无穷大、NaN 等。

指数位 E	分数位 N = 0	分数位 N ≠ 0	计算公式
0	±0	次正规数	$(-1)^S \times 2^{-126} \times (0.N)$
1, 2, ..., 254	正规数	正规数	$(-1)^S \times 2^{(E-127)} \times (1.N)$
255	±∞	NaN	

特别地，次正规数显著提升了小数精度：

- 最小正正规数为 $2^{-126} \approx 1.18 \times 10^{-38}$ ；
- 最小正次正规数为 $2^{-126} \times 2^{-23} \approx 1.4 \times 10^{-45}$ ；

双精度 double 也采用类似 float 的表示方法，在此不再赘述。

基本数据类型与数据结构的关系

我们知道，**数据结构是在计算机中组织与存储数据的方式**，它的主语是“结构”，而不是“数据”。如果我们想要表示“一排数字”，自然想到使用「数组」数据结构。数组的存储方式可以表示数字的相邻关系、顺序关系，但至于其中存储的是整数 int，还是小数 float，或是字符 char，则与**所谓的数据的结构无关了**。

换言之，基本数据类型提供了数据的“内容类型”，而数据结构提供数据的“组织方式”。

```
/* 使用多种「基本数据类型」来初始化「数组」 */
int numbers[5];
float decimals[5];
char characters[5];
bool booleans[5];
```

3.1.2. 计算机内存

在计算机中，内存和硬盘是两种主要的存储硬件设备。「硬盘」主要用于长期存储数据，容量较大（通常可达到 TB 级别）、速度较慢。「内存」用于运行程序时暂存数据，速度较快，但容量较小（通常为 GB 级别）。

算法运行中，相关数据都被存储在内存中。下图展示了一个计算机内存条，其中每个黑色方块都包含一块内存空间。我们可以将内存想象成一个巨大的 Excel 表格，其中每个单元格都可以存储 1 byte 的数据，在算法运行时，所有数据都被存储在这些单元格中。

系统通过「内存地址 Memory Location」来访问目标内存位置的数据。计算机根据特定规则给表格中每个单元格编号，保证每块内存空间都有独立的内存地址。自此，程序便通过这些地址，访问内存中的数据。

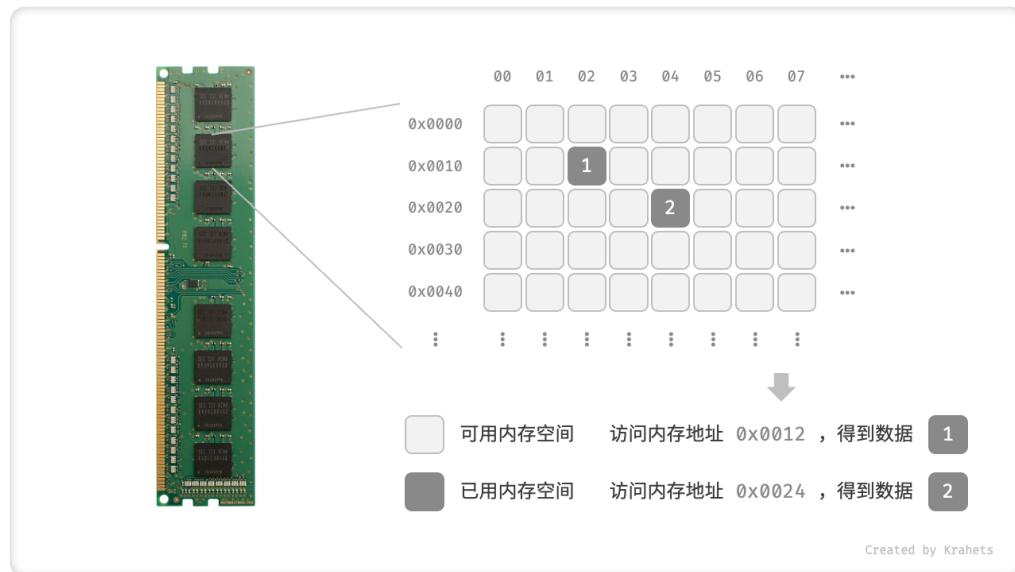


Figure 3-2. 内存条、内存空间、内存地址

内存资源是设计数据结构与算法的重要考虑因素。内存是所有程序的公共资源，当内存被某程序占用时，不能被其它程序同时使用。我们需要根据剩余内存资源的情况来设计算法。例如，若剩余内存空间有限，则要求算

法占用的峰值内存不能超过系统剩余内存；若运行的程序很多、缺少大块连续的内存空间，则要求选取的数据结构必须能够存储在离散的内存空间内。

3.2. 数据结构分类

数据结构主要可根据「逻辑结构」和「物理结构」两种角度进行分类。

3.2.1. 逻辑结构：线性与非线性

「逻辑结构」反映了数据之间的逻辑关系。数组和链表的数据按照顺序依次排列，反映了数据间的线性关系；树从顶至底按层级排列，反映了祖先与后代之间的派生关系；图由结点和边组成，反映了复杂网络关系。

我们一般将逻辑结构分为「线性」和「非线性」两种。“线性”这个概念很直观，即表明数据在逻辑关系上是排成一条线的；而如果数据之间的逻辑关系是非线性的（例如是网状或树状的），那么就是非线性数据结构。

- 线性数据结构：数组、链表、栈、队列、哈希表；
- 非线性数据结构：树、图、堆、哈希表；

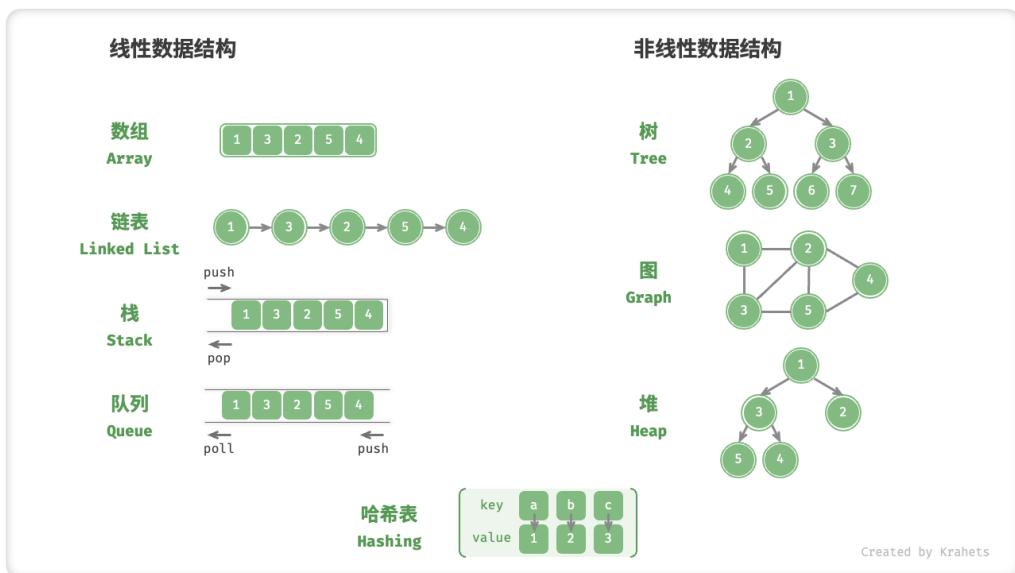


Figure 3-3. 线性与非线性数据结构

3.2.2. 物理结构：连续与离散



若感到阅读困难，建议先看完下个章节「数组与链表」，再回过头来理解物理结构的含义。

「物理结构」反映了数据在计算机内存中的存储方式。从本质上讲，分别是 **数组的连续空间存储** 和 **链表的离散空间存储**。物理结构从底层上决定了数据的访问、更新、增删等操作方法，在时间效率和空间效率方面呈现出此消彼长的特性。

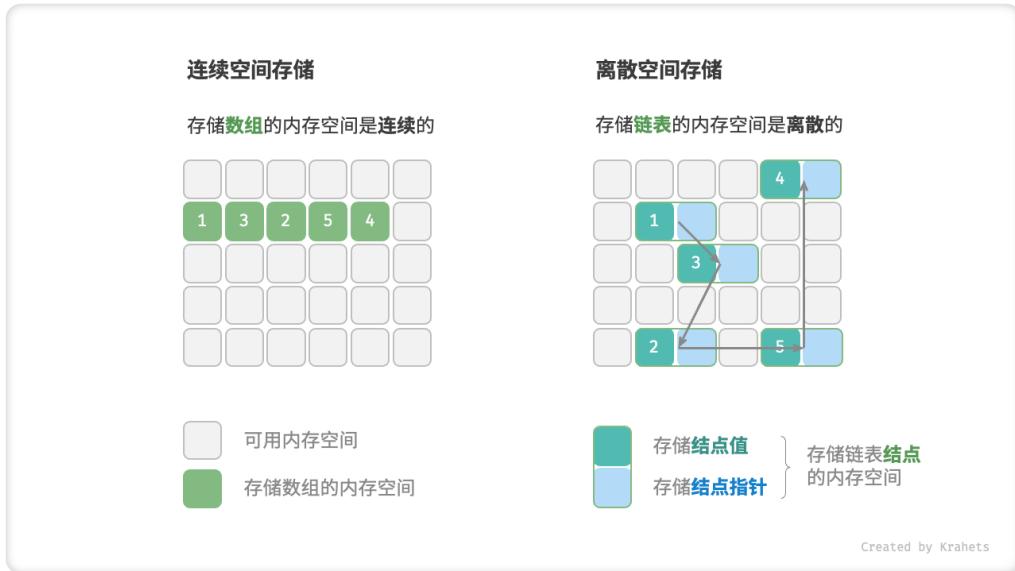


Figure 3-4. 连续空间存储与离散空间存储

所有数据结构都是基于数组、或链表、或两者组合实现的。例如栈和队列，既可以使用数组实现、也可以使用链表实现，而例如哈希表，其实现同时包含了数组和链表。

- 基于数组可实现：栈、队列、哈希表、树、堆、图、矩阵、张量（维度 ≥ 3 的数组）等；
- 基于链表可实现：栈、队列、哈希表、树、堆、图等；

基于数组实现的数据结构也被称为「静态数据结构」，这意味着该数据结构在被初始化后，长度不可变。相反地，基于链表实现的数据结构被称为「动态数据结构」，该数据结构在被初始化后，我们也可以在程序运行中修改其长度。



数组与链表是其他所有数据结构的“底层积木”，建议读者一定要多花些时间了解。

3.3. 小结

- 整数 byte, short, int, long、浮点数 float, double、字符 char、布尔 boolean 是计算机中的基本数据类型，占用空间的大小决定了它们的取值范围。
- 在程序运行时，数据存储在计算机的内存中。内存中每块空间都有独立的内存地址，程序是通过内存地址来访问数据的。
- 数据结构主要可以从逻辑结构和物理结构两个角度进行分类。逻辑结构反映了数据中元素之间的逻辑关系，物理结构反映了数据在计算机内存中的存储形式。

- 常见的逻辑结构有线性、树状、网状等。我们一般根据逻辑结构将数据结构分为线性（数组、链表、栈、队列）和非线性（树、图、堆）两种。根据实现方式的不同，哈希表可能是线性或非线性。
- 物理结构主要有两种，分别是连续空间存储（数组）和离散空间存储（链表），所有的数据结构都是由数组、或链表、或两者组合实现的。

4. 数组与链表

4.1. 数组

「数组 Array」是一种将 **相同类型元素** 存储在 **连续内存空间** 的数据结构，将元素在数组中的位置称为元素的「索引 Index」。

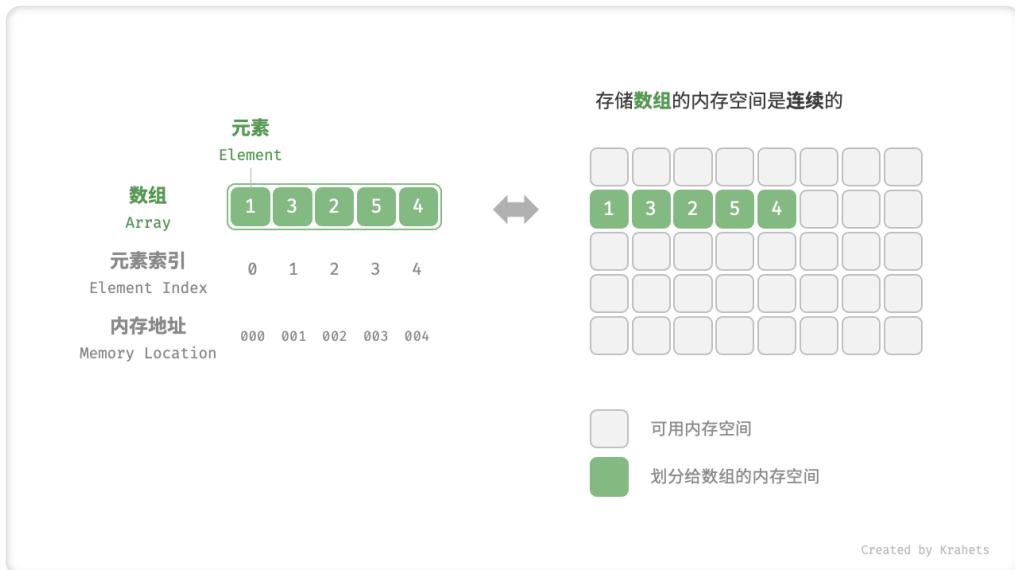


Figure 4-1. 数组定义与存储方式



观察上图，我们发现 **数组首元素的索引为 0**。你可能会想，这并不符合日常习惯，首个元素的索引为什么不是 1 呢，这不是更加自然吗？我认同你的想法，但请先记住这个设定，后面讲内存地址计算时，我会尝试解答这个问题。

数组初始化。一般会用到无初始值、给定初始值两种写法，可根据需求选取。在不给定初始值的情况下，一般所有元素会被初始化为默认值 0。

```
// === File: array.cpp ===
/* 初始化数组 */
// 存储在栈上
int arr[5];
int nums[5] { 1, 3, 2, 5, 4 };
// 存储在堆上
int* arr1 = new int[5];
int* nums1 = new int[5] { 1, 3, 2, 5, 4 };
```

4.1.1. 数组优点

在数组中访问元素非常高效。这是因为在数组中，计算元素的内存地址非常容易。给定数组首个元素的地址、和一个元素的索引，利用以下公式可以直接计算得到该元素的内存地址，从而直接访问此元素。



Figure 4-2. 数组元素的内存地址计算

```
# 元素内存地址 = 数组内存地址 + 元素长度 * 元素索引
elementAddr = firstElementAddr + elementLength * elementIndex
```

为什么数组元素索引从 0 开始编号？根据地址计算公式，索引本质上表示的是内存地址偏移量，首个元素的地址偏移量是 0，那么索引是 0 也就很自然了。

访问元素的高效性带来了许多便利。例如，我们可以在 $O(1)$ 时间内随机获取一个数组中的元素。

```
// === File: array.cpp ===
/* 随机返回一个数组元素 */
int randomAccess(int* nums, int size) {
    // 在区间 [0, size) 中随机抽取一个数字
    int randomIndex = rand() % size;
    // 获取并返回随机元素
    int randomNum = nums[randomIndex];
    return randomNum;
}
```

4.1.2. 数组缺点

数组在初始化后长度不可变。由于系统无法保证数组之后的内存空间是可用的，因此数组长度无法扩展。而若希望扩容数组，则需新建一个数组，然后把原数组元素依次拷贝到新数组，在数组很大的情况下，这是非常耗时的。

```
// === File: array.cpp ===
/* 扩展数组长度 */
int* extend(int* nums, int size, int enlarge) {
    // 初始化一个扩展长度后的数组
    int* res = new int[size + enlarge];
    // 将原数组中的所有元素复制到新数组
    for (int i = 0; i < size; i++) {
        res[i] = nums[i];
    }
    // 释放内存
    delete[] nums;
    // 返回扩展后的新数组
    return res;
}
```

数组中插入或删除元素效率低下。如果我们想要在数组中间插入一个元素，由于数组元素在内存中是“紧挨着的”，它们之间没有空间再放任何数据。因此，我们不得不将此索引之后的所有元素都向后移动一位，然后再把元素赋值给该索引。



Figure 4-3. 数组插入元素

```
// === File: array.cpp ===
/* 在数组的索引 index 处插入元素 num */
void insert(int* nums, int size, int num, int index) {
    // 把索引 index 以及之后的所有元素向后移动一位
    for (int i = size - 1; i > index; i--) {
        nums[i] = nums[i - 1];
    }
    // 将 num 赋给 index 处元素
```

```

    nums[index] = num;
}

```

删除元素也是类似，如果我们想要删除索引 i 处的元素，则需要把索引 i 之后的元素都向前移动一位。值得注意的是，删除元素后，原先末尾的元素变得“无意义”了，我们无需特意去修改它。

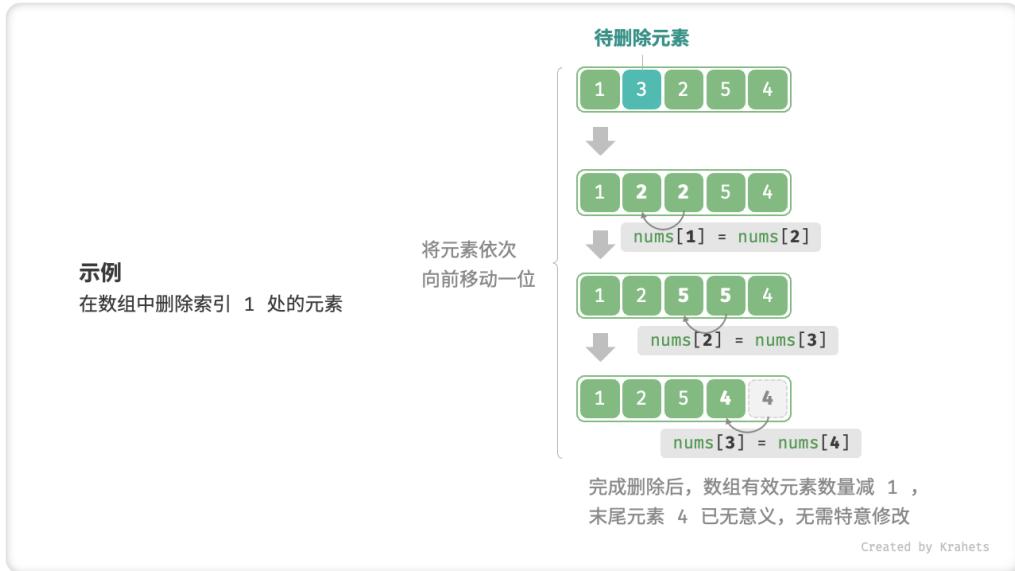


Figure 4-4. 数组删除元素

```

// === File: array.cpp ===
/* 删除索引 index 处元素 */
void remove(int* nums, int size, int index) {
    // 把索引 index 之后的所有元素向前移动一位
    for (int i = index; i < size - 1; i++) {
        nums[i] = nums[i + 1];
    }
}

```

总结来看，数组的插入与删除操作有以下缺点：

- **时间复杂度高**：数组的插入和删除的平均时间复杂度均为 $O(N)$ ，其中 N 为数组长度。
- **丢失元素**：由于数组的长度不可变，因此在插入元素后，超出数组长度范围的元素会被丢失。
- **内存浪费**：我们一般会初始化一个比较长的数组，只用前面一部分，这样在插入数据时，丢失的末尾元素都是我们不关心的，但这样做同时也会造成内存空间的浪费。

4.1.3. 数组常用操作

数组遍历。以下介绍两种常用的遍历方法。

```
// === File: array.cpp ===
/* 遍历数组 */
void traverse(int* nums, int size) {
    int count = 0;
    // 通过索引遍历数组
    for (int i = 0; i < size; i++) {
        count++;
    }
}
```

数组查找。通过遍历数组，查找数组内的指定元素，并输出对应索引。

```
// === File: array.cpp ===
/* 在数组中查找指定元素 */
int find(int* nums, int size, int target) {
    for (int i = 0; i < size; i++) {
        if (nums[i] == target)
            return i;
    }
    return -1;
}
```

4.1.4. 数组典型应用

随机访问。如果我们想要随机抽取一些样本，那么可以用数组存储，并生成一个随机序列，根据索引实现样本的随机抽取。

二分查找。例如前文查字典的例子，我们可以将字典中的所有字按照拼音顺序存储在数组中，然后使用与日常查纸质字典相同的“翻开中间，排除一半”的方式，来实现一个查电子字典的算法。

深度学习。神经网络中大量使用了向量、矩阵、张量之间的线性代数运算，这些数据都是以数组的形式构建的。数组是神经网络编程中最常使用的数据结构。

4.2. 链表



引言

内存空间是所有程序的公共资源，排除已占用的内存，空闲内存往往是散落在内存各处的。我们知道，存储数组需要内存空间连续，当我们需要申请一个很大的数组时，系统不一定存在这么大的连续内存空间。而链表则更加灵活，不需要内存是连续的，只要剩余内存空间大小够用即可。

「链表 Linked List」是一种线性数据结构，其中每个元素都是单独的对象，各个元素（一般称为结点）之间通过指针连接。由于结点中记录了连接关系，因此链表的存储方式相比于数组更加灵活，系统不必保证内存地址的连续性。

链表的「结点 Node」包含两项数据，一是结点「值 Value」，二是指向下一结点的「指针 Pointer」（或称「引用 Reference」）。

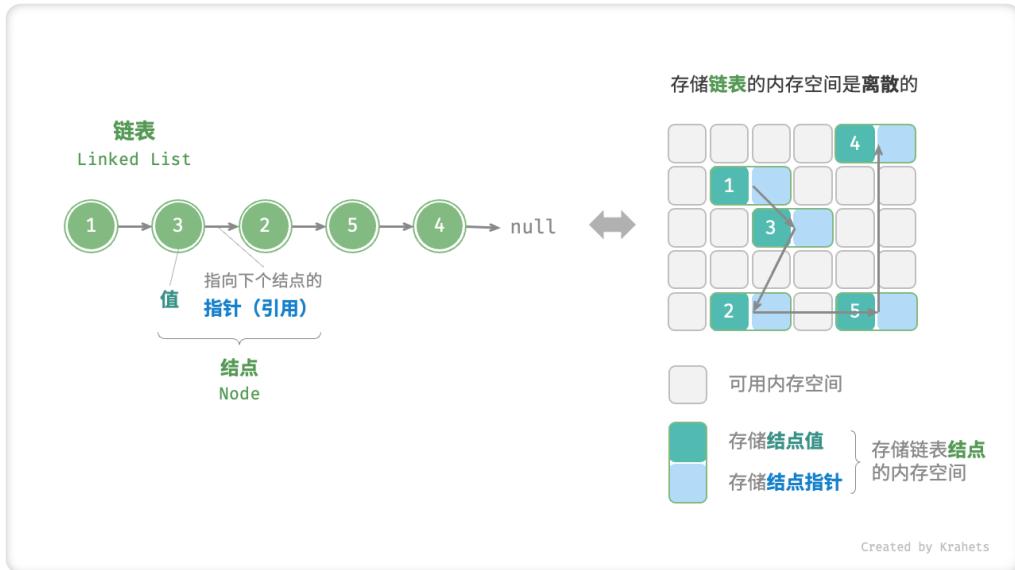


Figure 4-5. 链表定义与存储方式

```
/* 链表结点结构体 */
struct ListNode {
    int val;          // 结点值
    ListNode *next;  // 指向下一结点的指针 (引用)
    ListNode(int x) : val(x), next(nullptr) {} // 构造函数
};
```

尾结点指向什么？ 我们一般将链表的最后一个结点称为「尾结点」，其指向的是「空」，在 Java / C++ / Python 中分别记为 `null` / `nullptr` / `None`。在不引起歧义下，本书都使用 `null` 来表示空。

链表初始化方法。 建立链表分为两步，第一步是初始化各个结点对象，第二步是构建引用指向关系。完成后，即可以从链表的首个结点（即头结点）出发，访问其余所有的结点。



我们通常将头结点当作链表的代称，例如头结点 `head` 和链表 `head` 实际上是同义的。

```
// === File: linked_list.cpp ===
/* 初始化链表 1 -> 3 -> 2 -> 5 -> 4 */
// 初始化各个结点
ListNode* n0 = new ListNode(1);
```

```

ListNode* n1 = new ListNode(3);
ListNode* n2 = new ListNode(2);
ListNode* n3 = new ListNode(5);
ListNode* n4 = new ListNode(4);
// 构建引用指向
n0->next = n1;
n1->next = n2;
n2->next = n3;
n3->next = n4;

```

4.2.1. 链表优点

在链表中，插入与删除结点的操作效率高。比如，如果我们想在链表中间的两个结点 n_0 , n_1 之间插入一个新结点 P ，我们只需要改变两个结点指针即可，时间复杂度为 $O(1)$ ，相比数组的插入操作高效很多。

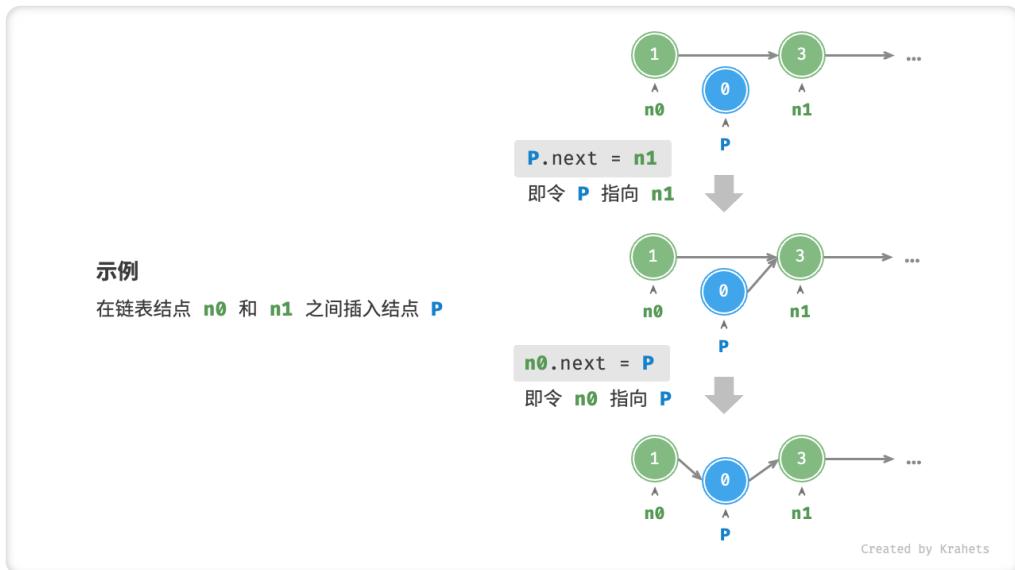


Figure 4-6. 链表插入结点

```

// === File: linked_list.cpp ===
/* 在链表的结点  $n_0$  之后插入结点  $P$  */
void insert(ListNode* n0, ListNode* P) {
    ListNode* n1 = n0->next;
    P->next = n1;
    n0->next = P;
}

```

在链表中删除结点也很方便，只需要改变一个结点指针即可。如下图所示，虽然在完成删除后结点 P 仍然指向 n_1 ，但实际上 P 已经不属此链表了，因为遍历此链表是无法访问到 P 的。

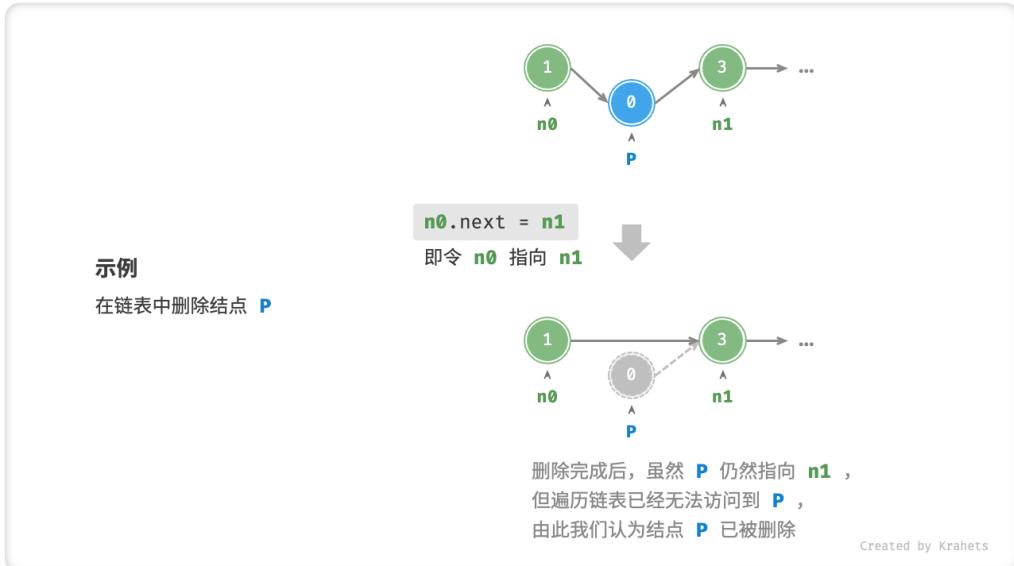


Figure 4-7. 链表删除结点

```
// === File: linked_list.cpp ===
/* 删除链表的结点 n0 之后的首个结点 */
void remove(ListNode* n0) {
    if (n0->next == nullptr)
        return;
    // n0 -> P -> n1
    ListNode* P = n0->next;
    ListNode* n1 = P->next;
    n0->next = n1;
    // 释放内存
    delete P;
}
```

4.2.2. 链表缺点

链表访问结点效率低。上节提到，数组可以在 $O(1)$ 时间下访问任意元素，但链表无法直接访问任意结点。这是因为计算机需要从头结点出发，一个一个地向后遍历到目标结点。例如，倘若想要访问链表索引为 `index` (即第 `index + 1` 个) 的结点，那么需要 `index` 次访问操作。

```
// === File: linked_list.cpp ===
/* 访问链表中索引为 index 的结点 */
ListNode* access(ListNode* head, int index) {
    for (int i = 0; i < index; i++) {
        if (head == nullptr)
            return nullptr;
        head = head->next;
    }
}
```

```
    }
    return head;
}
```

链表的内存占用多。链表以结点为单位，每个结点除了保存值外，还需额外保存指针（引用）。这意味着同样数据量下，链表比数组需要占用更多内存空间。

4.2.3. 链表常用操作

遍历链表查找。遍历链表，查找链表内值为 `target` 的结点，输出结点在链表中的索引。

```
// === File: linked_list.cpp ===
/* 在链表中查找值为 target 的首个结点 */
int find(ListNode* head, int target) {
    int index = 0;
    while (head != nullptr) {
        if (head->val == target)
            return index;
        head = head->next;
        index++;
    }
    return -1;
}
```

4.2.4. 常见链表类型

单向链表。即上述介绍的普通链表。单向链表的结点有「值」和指向下一结点的「指针（引用）」两项数据。我们将首个结点称为头结点，尾结点指向 `null`。

环形链表。如果我们令单向链表的尾结点指向头结点（即首尾相接），则得到一个环形链表。在环形链表中，我们可以将任意结点看作是头结点。

双向链表。单向链表仅记录了一个方向的指针（引用），在双向链表的结点定义中，同时有指向下一结点（后继结点）和上一结点（前驱结点）的「指针（引用）」。双向链表相对于单向链表更加灵活，即可以朝两个方向遍历链表，但也需要占用更多的内存空间。

```
/* 双向链表结点结构体 */
struct ListNode {
    int val;           // 结点值
    ListNode *next;   // 指向后继结点的指针（引用）
    ListNode *prev;   // 指向前驱结点的指针（引用）
    ListNode(int x) : val(x), next(nullptr), prev(nullptr) {} // 构造函数
};
```

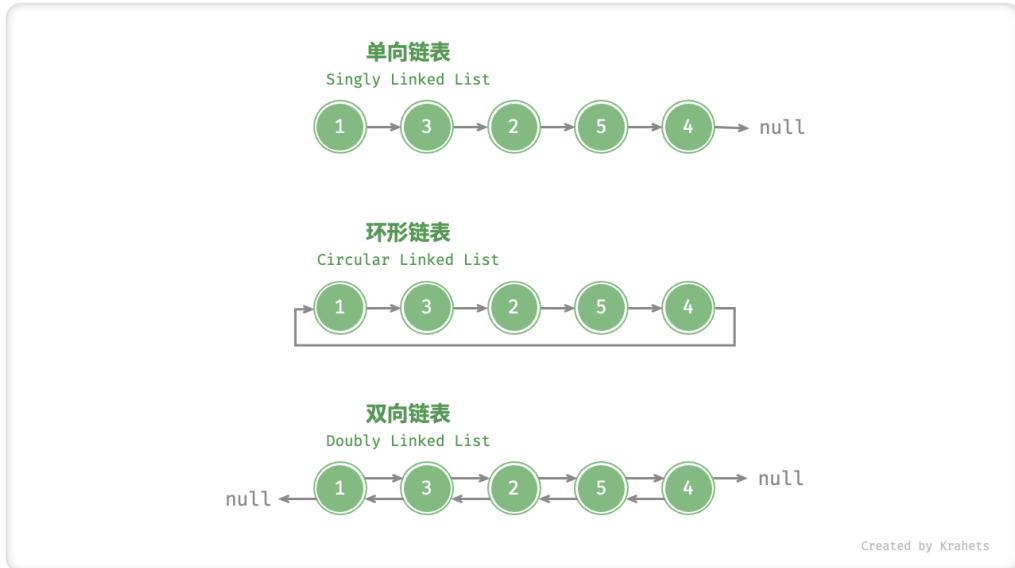


Figure 4-8. 常见链表种类

4.3. 列表

由于长度不可变，数组的实用性大大降低。在很多情况下，我们事先并不知道会输入多少数据，这就为数组长度的选择带来了很大困难。长度选小了，需要在添加数据中频繁地扩容数组；长度选大了，又造成内存空间的浪费。

为了解决此问题，诞生了一种被称为「列表 List」的数据结构。列表可以被理解为长度可变的数组，因此也常被称为「动态数组 Dynamic Array」。列表基于数组实现，继承了数组的优点，同时还可以在程序运行中实时扩容。在列表中，我们可以自由地添加元素，而不用担心超过容量限制。

4.3.1. 列表常用操作

初始化列表。 我们通常会使用到“无初始值”和“有初始值”的两种初始化方法。

```
// === File: list.cpp ===
/* 初始化列表 */
// 需注意，C++ 中 vector 即是本文描述的 list
// 无初始值
vector<int> list1;
// 有初始值
vector<int> list = { 1, 3, 2, 5, 4 };
```

访问与更新元素。 列表的底层数据结构是数组，因此可以在 $O(1)$ 时间内访问与更新元素，效率很高。

```
// === File: list.cpp ===
/* 访问元素 */
int num = list[1]; // 访问索引 1 处的元素

/* 更新元素 */
list[1] = 0; // 将索引 1 处的元素更新为 0
```

在列表中添加、插入、删除元素。相对于数组，列表可以自由地添加与删除元素。在列表尾部添加元素的时间复杂度为 $O(1)$ ，但是插入与删除元素的效率仍与数组一样低，时间复杂度为 $O(N)$ 。

```
// === File: list.cpp ===
/* 清空列表 */
list.clear();

/* 尾部添加元素 */
list.push_back(1);
list.push_back(3);
list.push_back(2);
list.push_back(5);
list.push_back(4);

/* 中间插入元素 */
list.insert(list.begin() + 3, 6); // 在索引 3 处插入数字 6

/* 删除元素 */
list.erase(list.begin() + 3); // 删除索引 3 处的元素
```

遍历列表。与数组一样，列表可以使用索引遍历，也可以使用 `for-each` 直接遍历。

```
// === File: list.cpp ===
/* 通过索引遍历列表 */
int count = 0;
for (int i = 0; i < list.size(); i++) {
    count++;
}

/* 直接遍历列表元素 */
count = 0;
for (int n : list) {
    count++;
}
```

拼接两个列表。再创建一个新列表 `list1`，我们可以将其中一个列表拼接到另一个的尾部。

```
// === File: list.cpp ===
/* 拼接两个列表 */
vector<int> list1 = { 6, 8, 7, 10, 9 };
// 将列表 list1 拼接到 list 之后
list.insert(list.end(), list1.begin(), list1.end());
```

排序列表。排序也是常用的方法之一，完成列表排序后，我们就可以使用在数组类算法题中经常考察的「二分查找」和「双指针」算法了。

```
// === File: list.cpp ===
/* 排序列表 */
sort(list.begin(), list.end()); // 排序后，列表元素从小到大排列
```

4.3.2. 列表简易实现 *

为了帮助加深对列表的理解，我们在此提供一个列表的简易版本的实现。需要关注三个核心点：

- **初始容量：**选取一个合理的数组的初始容量 `initialCapacity`。在本示例中，我们选择 10 作为初始容量。
- **数量记录：**需要声明一个变量 `size`，用来记录列表当前有多少个元素，并随着元素插入与删除实时更新。根据此变量，可以定位列表的尾部，以及判断是否需要扩容。
- **扩容机制：**插入元素有可能导致超出列表容量，此时需要扩容列表，方法是建立一个更大的数组来替换当前数组。需要给定一个扩容倍数 `extendRatio`，在本示例中，我们规定每次将数组扩容至之前的 2 倍。

本示例是为了帮助读者对如何实现列表产生直观的认识。实际编程语言中，列表的实现远比以下代码复杂且标准，感兴趣的读者可以查阅源码学习。

```
// === File: my_list.cpp ===
/* 列表类简易实现 */
class MyList {
private:
    int* nums; // 数组（存储列表元素）
    int numsCapacity = 10; // 列表容量
    int numsSize = 0; // 列表长度（即当前元素数量）
    int extendRatio = 2; // 每次列表扩容的倍数

public:
    /* 构造方法 */
    MyList() {
        nums = new int[numsCapacity];
    }

    /* 析构方法 */
    ~MyList() {
        delete[] nums;
    }
}
```

```
}

/* 获取列表长度（即当前元素数量）*/
int size() {
    return numsSize;
}

/* 获取列表容量 */
int capacity() {
    return numsCapacity;
}

/* 访问元素 */
int get(int index) {
    // 索引如果越界则抛出异常，下同
    if (index < 0 || index >= size())
        throw out_of_range("索引越界");
    return nums[index];
}

/* 更新元素 */
void set(int index, int num) {
    if (index < 0 || index >= size())
        throw out_of_range("索引越界");
    nums[index] = num;
}

/* 尾部添加元素 */
void add(int num) {
    // 元素数量超出容量时，触发扩容机制
    if (size() == capacity())
        extendCapacity();
    nums[size()] = num;
    // 更新元素数量
    numsSize++;
}

/* 中间插入元素 */
void insert(int index, int num) {
    if (index < 0 || index >= size())
        throw out_of_range("索引越界");
    // 元素数量超出容量时，触发扩容机制
    if (size() == capacity())
        extendCapacity();
    // 索引 i 以及之后的元素都向后移动一位
    for (int j = size() - 1; j >= index; j--) {
```

```
        nums[j + 1] = nums[j];
    }
    nums[index] = num;
    // 更新元素数量
    numsSize++;
}

/* 删除元素 */
int remove(int index) {
    if (index < 0 || index >= size())
        throw out_of_range("索引越界");
    int num = nums[index];
    // 索引 i 之后的元素都向前移动一位
    for (int j = index; j < size() - 1; j++) {
        nums[j] = nums[j + 1];
    }
    // 更新元素数量
    numsSize--;
    // 返回被删除元素
    return num;
}

/* 列表扩容 */
void extendCapacity() {
    // 新建一个长度为 size * extendRatio 的数组，并将原数组拷贝到新数组
    int newCapacity = capacity() * extendRatio;
    int* tmp = nums;
    nums = new int[newCapacity];
    // 将原数组中的所有元素复制到新数组
    for (int i = 0; i < size(); i++) {
        nums[i] = tmp[i];
    }
    // 释放内存
    delete[] tmp;
    numsCapacity = newCapacity;
}

/* 将列表转换为 Vector 用于打印 */
vector<int> toVector() {
    // 仅转换有效长度范围内的列表元素
    vector<int> vec(size());
    for (int i = 0; i < size(); i++) {
        vec[i] = nums[i];
    }
    return vec;
}
```

```
};
```

4.4. 小结

- 数组和链表是两种基本数据结构，代表了数据在计算机内存中的两种存储方式，即连续空间存储和离散空间存储。两者的特点呈现出此消彼长的关系。
- 数组支持随机访问、内存空间占用小；但插入与删除元素效率低，且初始化后长度不可变。
- 链表可通过更改指针实现高效的结点插入与删除，并且可以灵活地修改长度；但结点访问效率低、占用内存多。常见的链表类型有单向链表、循环链表、双向链表。
- 列表又称动态数组，是基于数组实现的一种数据结构，其保存了数组的优势，且可以灵活改变长度。列表的出现大大提升了数组的实用性，但副作用是会造成部分内存空间浪费。
- 下表总结对比了数组与链表的各项特性。

	数组	链表
存储方式	连续内存空间	离散内存空间
数据结构长度	长度不可变	长度可变
内存使用率	占用内存少、缓存局部性好	占用内存多
优势操作	随机访问	插入、删除



缓存局部性的简单解释

在计算机中，数据读写速度排序是“硬盘 < 内存 < CPU 缓存”。当我们访问数组元素时，计算机不仅会加载它，还会缓存其周围的其它数据，从而借助高速缓存来提升后续操作的执行速度。链表则不然，计算机只能挨个地缓存各个结点，这样的多次“搬运”降低了整体效率。

- 下表对比了数组与链表的各种操作效率。

操作	数组	链表
访问元素	$O(1)$	$O(N)$
添加元素	$O(N)$	$O(1)$
删除元素	$O(N)$	$O(1)$

5. 栈与队列

5.1. 栈

「栈 Stack」是一种遵循「先入后出 first in, last out」数据操作规则的线性数据结构。我们可以将栈类比为放在桌面上的一摞盘子，如果需要拿出底部的盘子，则需要先将上面的盘子依次取出。

“盘子”是一种形象比喻，我们将盘子替换为任意一种元素（例如整数、字符、对象等），就得到了栈数据结构。

我们将这一摞元素的顶部称为「栈顶」，将底部称为「栈底」，将把元素添加到栈顶的操作称为「入栈」，将删除栈顶元素的操作称为「出栈」。

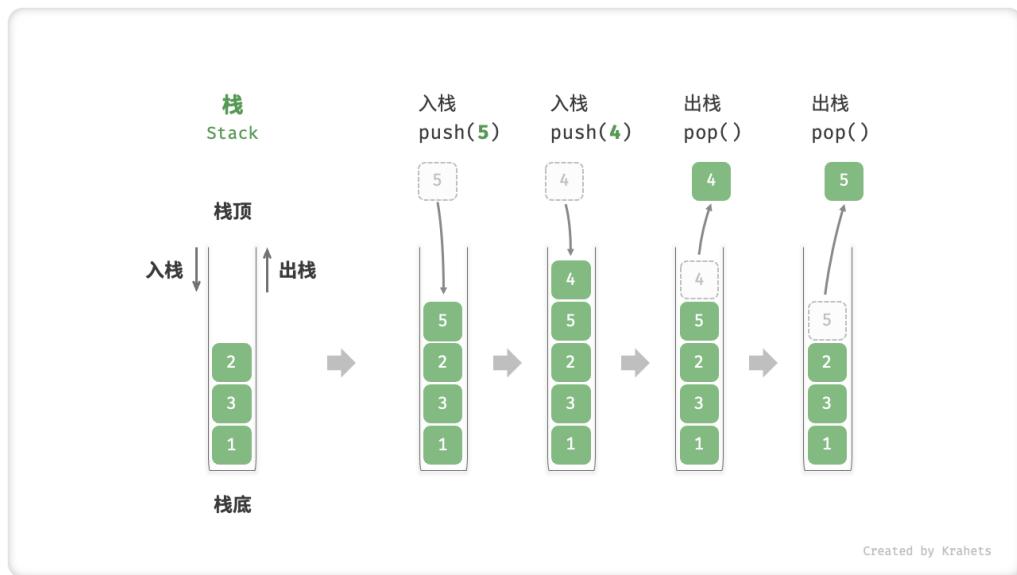


Figure 5-1. 栈的先入后出规则

5.1.1. 栈常用操作

栈的常用操作见下表（方法命名以 Java 为例）。

方法	描述	时间复杂度
push()	元素入栈（添加至栈顶）	$O(1)$
pop()	栈顶元素出栈	$O(1)$
peek()	访问栈顶元素	$O(1)$
size()	获取栈的长度	$O(1)$

方法	描述	时间复杂度
isEmpty()	判断栈是否为空	$O(1)$

我们可以直接使用编程语言实现好的栈类。某些语言并未专门提供栈类，但我们可以直接把该语言的「数组」或「链表」看作栈来使用，并通过“脑补”来屏蔽无关操作。

```
// === File: stack.cpp ===
/* 初始化栈 */
stack<int> stack;

/* 元素入栈 */
stack.push(1);
stack.push(3);
stack.push(2);
stack.push(5);
stack.push(4);

/* 访问栈顶元素 */
int top = stack.top();

/* 元素出栈 */
stack.pop();

/* 获取栈的长度 */
int size = stack.size();

/* 判断是否为空 */
bool empty = stack.empty();
```

5.1.2. 栈的实现

为了更加清晰地了解栈的运行机制，接下来我们来自动手实现一个栈类。

栈规定元素是先入后出的，因此我们只能在栈顶添加或删除元素。然而，数组或链表都可以在任意位置添加删除元素，因此 **栈可被看作是一种受约束的数组或链表**。换言之，我们可以“屏蔽”数组或链表的部分无关操作，使之对外的表现逻辑符合栈的规定即可。

基于链表的实现

使用「链表」实现栈时，将链表的头结点看作栈顶，将尾结点看作栈底。

对于入栈操作，将元素插入到链表头部即可，这种结点添加方式被称为“头插法”。而对于出栈操作，则将头结点从链表中删除即可。

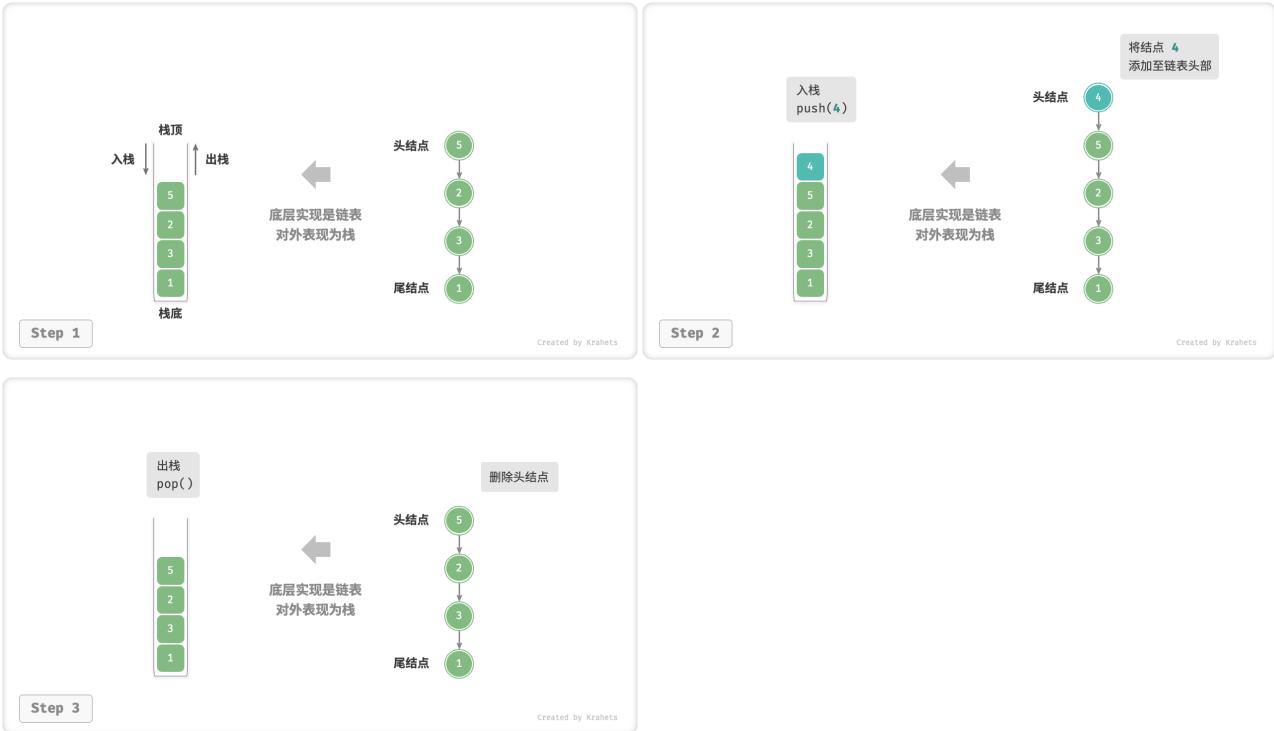


Figure 5-2. 基于链表实现栈的入栈出栈操作

以下是基于链表实现栈的示例代码。

```
// === File: linkedlist_stack.cpp ===
/* 基于链表实现的栈 */
class LinkedListStack {
private:
    ListNode* stackTop; // 将头结点作为栈顶
    int stkSize; // 栈的长度

public:
    LinkedListStack() {
        stackTop = nullptr;
        stkSize = 0;
    }

    ~LinkedListStack() {
        freeMemoryLinkedList(stackTop);
    }

    /* 获取栈的长度 */
    int size() {
        return stkSize;
    }
}
```

```
/* 判断栈是否为空 */
bool empty() {
    return size() == 0;
}

/* 入栈 */
void push(int num) {
    ListNode* node = new ListNode(num);
    node->next = stackTop;
    stackTop = node;
    stkSize++;
}

/* 出栈 */
void pop() {
    int num = top();
    ListNode *tmp = stackTop;
    stackTop = stackTop->next;
    // 释放内存
    delete tmp;
    stkSize--;
}

/* 访问栈顶元素 */
int top() {
    if (size() == 0)
        throw out_of_range(" 栈为空");
    return stackTop->val;
}

/* 将 List 转化为 Array 并返回 */
vector<int> toVector() {
    ListNode* node = stackTop;
    vector<int> res(size());
    for (int i = res.size() - 1; i >= 0; i--) {
        res[i] = node->val;
        node = node->next;
    }
    return res;
}
};
```

基于数组的实现

使用「数组」实现栈时，考虑将数组的尾部当作栈顶。这样设计下，「入栈」与「出栈」操作就对应在数组尾部「添加元素」与「删除元素」，时间复杂度都为 $O(1)$ 。

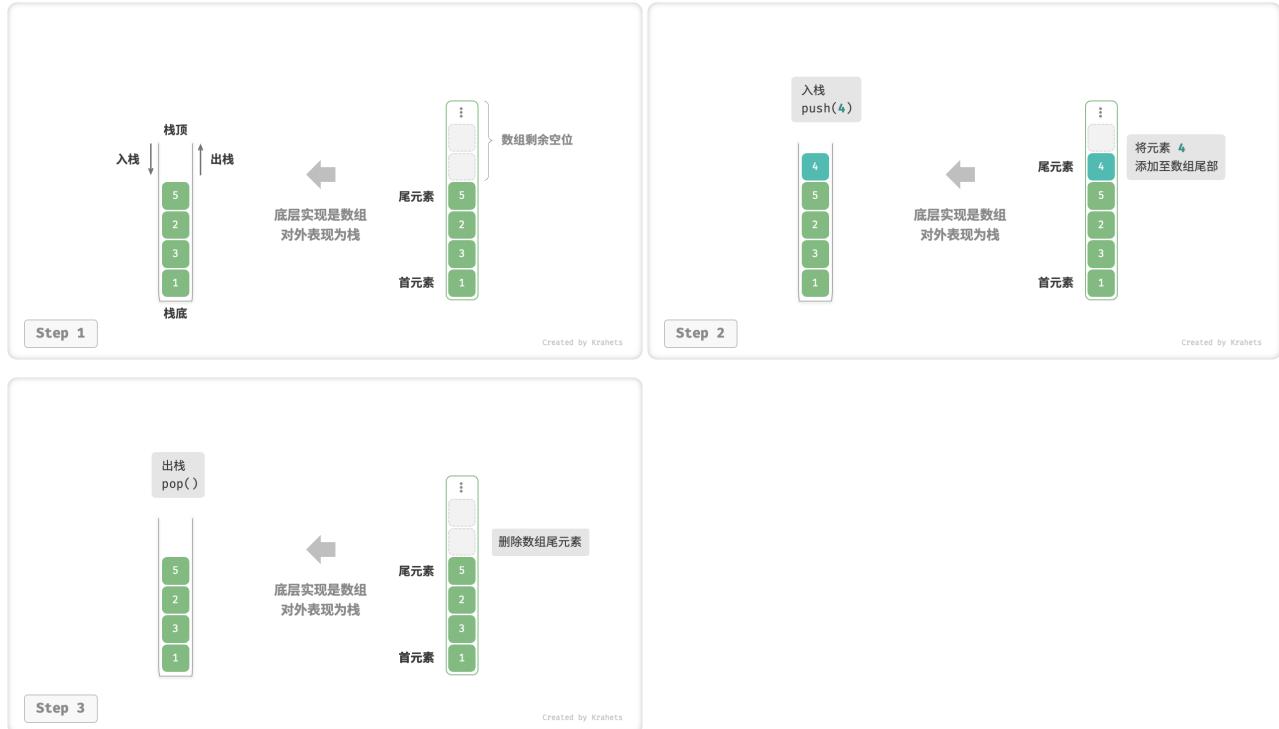


Figure 5-3. 基于数组实现栈的入栈出栈操作

由于入栈的元素可能是源源不断的，因此可以使用支持动态扩容的「列表」，这样就无需自行实现数组扩容了。以下是示例代码。

```
// === File: array_stack.cpp ===
/* 基于数组实现的栈 */
class ArrayStack {
private:
    vector<int> stack;

public:
    /* 获取栈的长度 */
    int size() {
        return stack.size();
    }

    /* 判断栈是否为空 */
    bool empty() {
        return stack.empty();
    }
}
```

```
}

/* 入栈 */
void push(int num) {
    stack.push_back(num);
}

/* 出栈 */
void pop() {
    int oldTop = top();
    stack.pop_back();
}

/* 访问栈顶元素 */
int top() {
    if(empty())
        throw out_of_range("栈为空");
    return stack.back();
}

/* 返回 Vector */
vector<int> toVector() {
    return stack;
}
};
```

5.1.3. 两种实现对比

支持操作

两种实现都支持栈定义中的各项操作，数组实现额外支持随机访问，但这已经超出栈的定义范畴，一般不会用到。

时间效率

在数组（列表）实现中，入栈与出栈操作都是在预先分配好的连续内存中操作，具有很好的缓存本地性，效率很好。然而，如果入栈时超出数组容量，则会触发扩容机制，那么该次入栈操作的时间复杂度为 $O(n)$ 。

在链表实现中，链表的扩容非常灵活，不存在上述数组扩容时变慢的问题。然而，入栈操作需要初始化结点对象并修改指针，因而效率不如数组。进一步地思考，如果入栈元素不是 `int` 而是结点对象，那么就可以省去初始化步骤，从而提升效率。

综上所述，当入栈与出栈操作的元素是基本数据类型（例如 `int`, `double`）时，则结论如下：

- 数组实现的栈在触发扩容时会变慢，但由于扩容是低频操作，因此 **总体效率更高**；

- 链表实现的栈可以提供 **更加稳定的效率表现**;

空间效率

在初始化列表时，系统会给列表分配“初始容量”，该容量可能超过我们的需求。并且扩容机制一般是按照特定倍率（比如 2 倍）进行扩容，扩容后的容量也可能超出我们的需求。因此，**数组实现栈会造成一定的空间浪费**。

当然，由于结点需要额外存储指针，因此 **链表结点比数组元素占用更大**。

综上，我们不能简单地确定哪种实现更加省内存，需要 case-by-case 地分析。

5.1.4. 栈典型应用

- **浏览器中的后退与前进、软件中的撤销与反撤销。**每当我们打开新的网页，浏览器就将上一个网页执行入栈，这样我们就可以通过「后退」操作来回到上一页面，后退操作实际上是在执行出栈。如果要同时支持后退和前进，那么则需要两个栈来配合实现。
- **程序内存管理。**每当调用函数时，系统就会在栈顶添加一个栈帧，用来记录函数的上下文信息。在递归函数中，向下递推会不断执行入栈，向上回溯阶段时出栈。

5.2. 队列

「队列 Queue」是一种遵循「先入先出 first in, first out」数据操作规则的线性数据结构。顾名思义，队列模拟的是排队现象，即外面的人不断加入队列尾部，而处于队列头部的人不断地离开。

我们将队列头部称为「队首」，队列尾部称为「队尾」，将把元素加入队尾的操作称为「入队」，删除队首元素的操作称为「出队」。

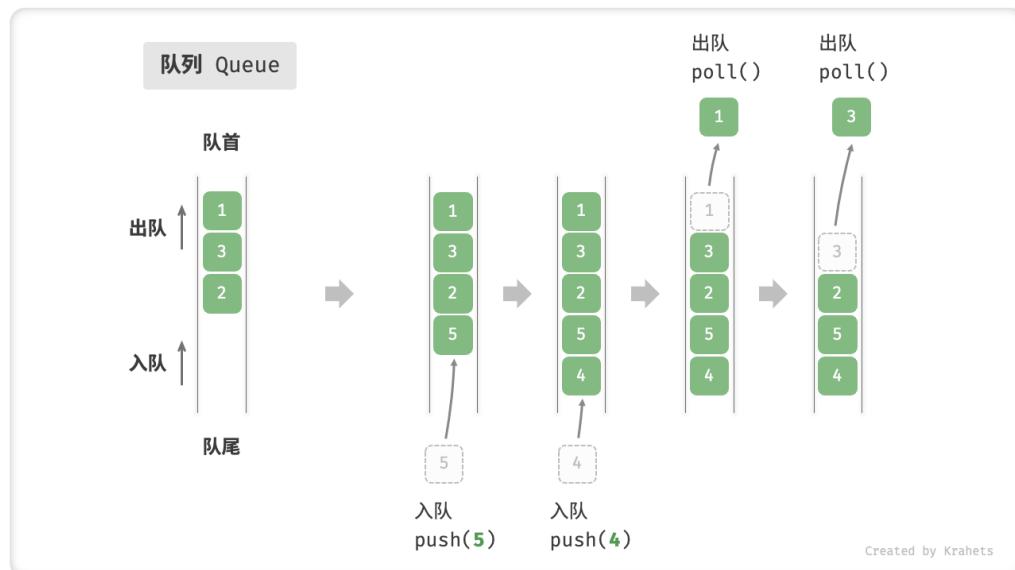


Figure 5-4. 队列的先入先出规则

5.2.1. 队列常用操作

队列的常用操作见下表，方法名需根据特定语言来确定。

方法名	描述	时间复杂度
push()	元素入队，即将元素添加至队尾	$O(1)$
poll()	队首元素出队	$O(1)$
front()	访问队首元素	$O(1)$
size()	获取队列的长度	$O(1)$
isEmpty()	判断队列是否为空	$O(1)$

我们可以直接使用编程语言实现好的队列类。

```
// === File: queue.cpp ===
/* 初始化队列 */
queue<int> queue;

/* 元素入队 */
queue.push(1);
queue.push(3);
queue.push(2);
queue.push(5);
queue.push(4);

/* 访问队首元素 */
int front = queue.front();

/* 元素出队 */
queue.pop();

/* 获取队列的长度 */
int size = queue.size();

/* 判断队列是否为空 */
bool empty = queue.empty();
```

5.2.2. 队列实现

队列需要一种可以在一端添加，并在另一端删除的数据结构，也可以使用链表或数组来实现。

基于链表的实现

我们将链表的「头结点」和「尾结点」分别看作是队首和队尾，并规定队尾只可添加结点，队首只可删除结点。

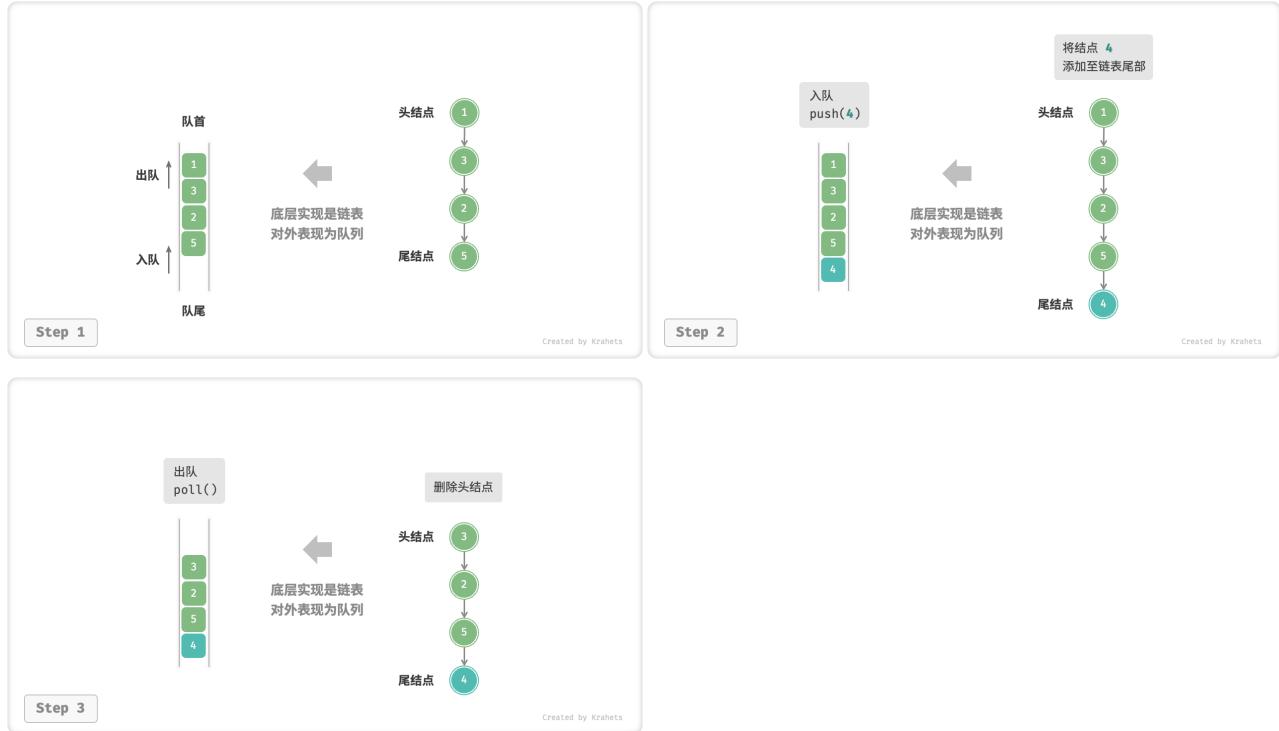


Figure 5-5. 基于链表实现队列的入队出队操作

以下是使用链表实现队列的示例代码。

```
// === File: linkedlist_queue.cpp ===
/* 基于链表实现的队列 */
class LinkedListQueue {
private:
    ListNode *front, *rear; // 头结点 front , 尾结点 rear
    int queSize;

public:
    LinkedListQueue() {
        front = nullptr;
        rear = nullptr;
        queSize = 0;
    }

    ~LinkedListQueue() {
        delete front;
    }
}
```

```
    delete rear;
}

/* 获取队列的长度 */
int size() {
    return queSize;
}

/* 判断队列是否为空 */
bool empty() {
    return queSize == 0;
}

/* 入队 */
void push(int num) {
    // 尾结点后添加 num
    ListNode* node = new ListNode(num);
    // 如果队列为空，则头、尾结点都指向该结点
    if (front == nullptr) {
        front = node;
        rear = node;
    }
    // 如果队列不为空，则将该结点添加到尾结点后
    else {
        rear->next = node;
        rear = node;
    }
    queSize++;
}

/* 出队 */
void poll() {
    int num = peek();
    // 删除头结点
    ListNode *tmp = front;
    front = front->next;
    // 释放内存
    delete tmp;
    queSize--;
}

/* 访问队首元素 */
int peek() {
    if (size() == 0)
        throw out_of_range("队列为空");
    return front->val;
```

```

}

/* 将链表转化为 Vector 并返回 */
vector<int> toVector() {
    ListNode* node = front;
    vector<int> res(size());
    for (int i = 0; i < res.size(); i++) {
        res[i] = node->val;
        node = node->next;
    }
    return res;
}
};

}

```

基于数组的实现

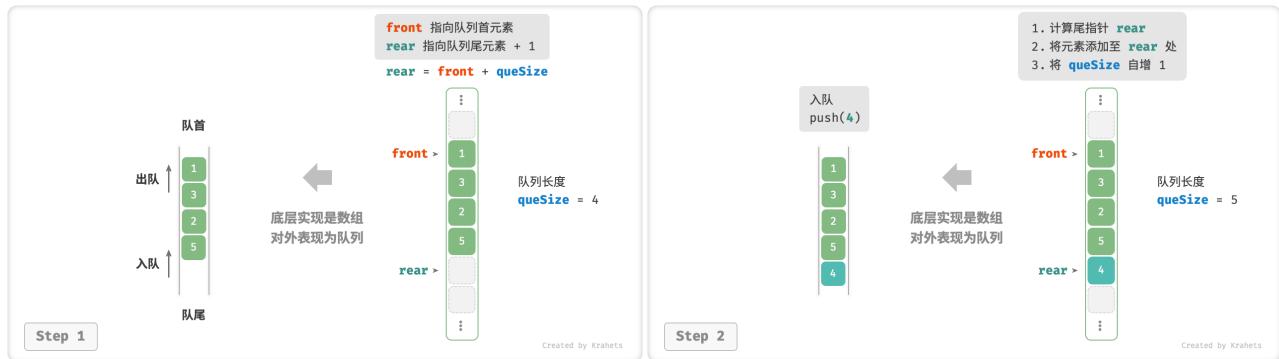
数组的删除首元素的时间复杂度为 $O(n)$ ，这会导致出队操作效率低下。然而，我们可以采取下述的巧妙方法来避免这个问题。

考虑借助一个变量 `front` 来指向队首元素的索引，并维护变量 `queSize` 来记录队列长度。我们定义 `rear = front + queSize`，该公式计算出来的 `rear` 指向“队尾元素索引 + 1”的位置。

在该设计下，数组中包含元素的有效区间为 `[front, rear - 1]`，进而

- 对于入队操作，将输入元素赋值给 `rear` 索引处，并将 `queSize` 自增 1 即可；
- 对于出队操作，仅需将 `front` 自增 1，并将 `queSize` 自减 1 即可；

观察发现，入队与出队操作都仅需单次操作即可完成，时间复杂度皆为 $O(1)$ 。



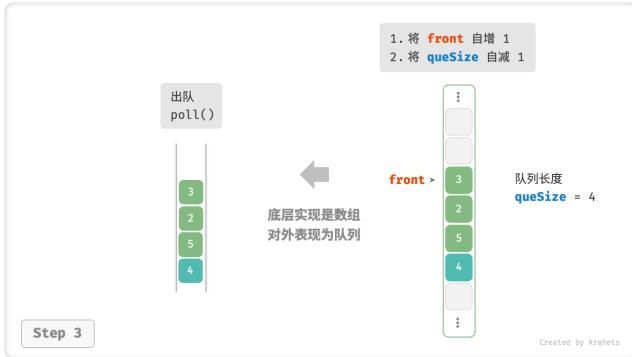


Figure 5-6. 基于数组实现队列的入队出队操作

细心的同学可能会发现一个问题：在不断入队与出队的过程中，`front` 和 `rear` 都在向右移动，在到达数组尾部后就无法继续移动了。为解决此问题，我们考虑将数组看作是首尾相接的，这样的数组被称为「环形数组」。

对于环形数组，我们需要令 `front` 或 `rear` 在越过数组尾部后，直接绕回到数组头部接续遍历。这种周期性规律可以通过「取余操作」来实现，详情请见以下代码。

```
// === File: array_queue.cpp ===
/* 基于环形数组实现的队列 */
class ArrayQueue {
private:
    int *nums;          // 用于存储队列元素的数组
    int front;          // 队首指针，指向队首元素
    int queSize;        // 队列长度
    int queCapacity;   // 队列容量

public:
    ArrayQueue(int capacity) {
        // 初始化数组
        nums = new int[capacity];
        queCapacity = capacity;
        front = queSize = 0;
    }

    ~ArrayQueue() {
        delete[] nums;
    }

    /* 获取队列的容量 */
    int capacity() {
        return queCapacity;
    }

    /* 获取队列的长度 */
    int size() {

```

```
    return queSize;
}

/* 判断队列是否为空 */
bool empty() {
    return size() == 0;
}

/* 入队 */
void push(int num) {
    if (queSize == queCapacity) {
        cout << "队列已满" << endl;
        return;
    }
    // 计算队尾指针，指向队尾索引 + 1
    // 通过取余操作，实现 rear 越过数组尾部后回到头部
    int rear = (front + queSize) % queCapacity;
    // 将 num 添加至队尾
    nums[rear] = num;
    queSize++;
}

/* 出队 */
void poll() {
    int num = peek();
    // 队首指针向后移动一位，若越过尾部则返回到数组头部
    front = (front + 1) % queCapacity;
    queSize--;
}

/* 访问队首元素 */
int peek() {
    if (empty())
        throw out_of_range("队列为空");
    return nums[front];
}

/* 将数组转化为 Vector 并返回 */
vector<int> toVector() {
    // 仅转换有效长度范围内的列表元素
    vector<int> arr(queSize);
    for (int i = 0, j = front; i < queSize; i++, j++) {
        arr[i] = nums[j % queCapacity];
    }
    return arr;
}
```

```
};
```

以上实现的队列仍存在局限性，即长度不可变。不过这个问题很容易解决，我们可以将数组替换为列表（即动态数组），从而引入扩容机制。有兴趣的同学可以尝试自行实现。

5.2.3. 两种实现对比

与栈的结论一致，在此不再赘述。

5.2.4. 队列典型应用

- **淘宝订单**。购物者下单后，订单就被加入到队列之中，随后系统再根据顺序依次处理队列中的订单。在双十一时，在短时间内会产生海量的订单，如何处理「高并发」则是工程师们需要重点思考的问题。
- **各种待办事项**。任何需要实现“先来后到”的功能，例如打印机的任务队列、餐厅的出餐队列等等。

5.3. 双向队列

对于队列，我们只能在头部删除或在尾部添加元素，而「双向队列 Deque」更加灵活，在其头部和尾部都能执行元素添加或删除操作。

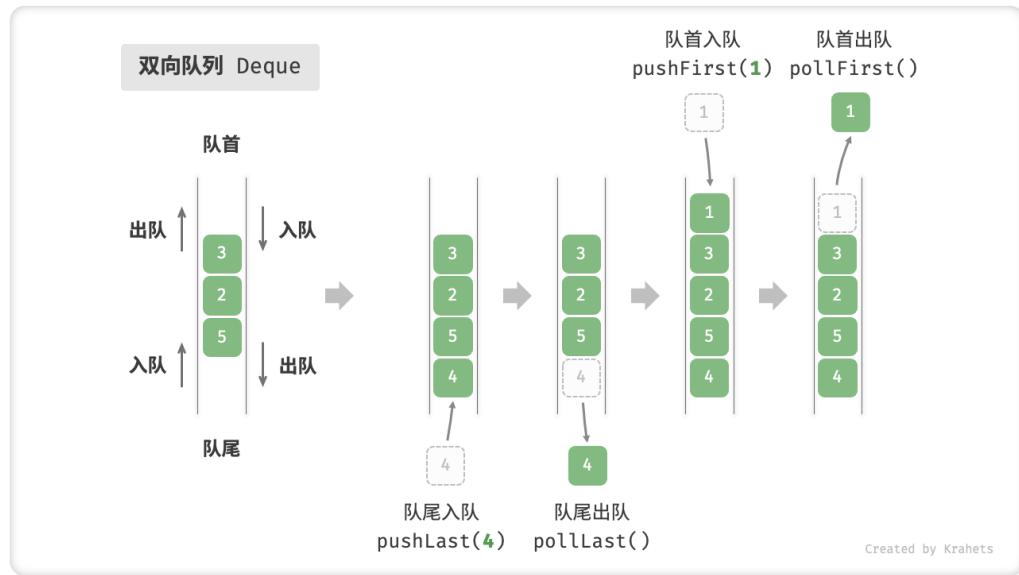


Figure 5-7. 双向队列的操作

5.3.1. 双向队列常用操作

双向队列的常用操作见下表，方法名需根据特定语言来确定。

方法名	描述	时间复杂度
pushFirst()	将元素添加至队首	$O(1)$
pushLast()	将元素添加至队尾	$O(1)$
pollFirst()	删除队首元素	$O(1)$
pollLast()	删除队尾元素	$O(1)$
peekFirst()	访问队首元素	$O(1)$
peekLast()	访问队尾元素	$O(1)$
size()	获取队列的长度	$O(1)$
isEmpty()	判断队列是否为空	$O(1)$

同样地，我们可以直接使用编程语言实现好的双向队列类。

```
// === File: deque.cpp ===
/* 初始化双向队列 */
deque<int> deque;

/* 元素入队 */
deque.push_back(2);    // 添加至队尾
deque.push_back(5);
deque.push_back(4);
deque.push_front(3);   // 添加至队首
deque.push_front(1);

/* 访问元素 */
int front = deque.front(); // 队首元素
int back = deque.back();   // 队尾元素

/* 元素出队 */
deque.pop_front(); // 队首元素出队
deque.pop_back(); // 队尾元素出队

/* 获取双向队列的长度 */
int size = deque.size();

/* 判断双向队列是否为空 */
bool empty = deque.empty();
```

5.3.2. 双向队列实现 *

与队列类似，双向队列同样可以使用链表或数组来实现。

基于双向链表的实现

回忆上节内容，由于可以方便地删除链表头结点（对应出队操作），以及在链表尾结点后添加新结点（对应入队操作），因此我们使用普通单向链表来实现队列。

而双向队列的头部和尾部都可以执行入队与出队操作，换言之，双向队列的操作是“首尾对称”的，也需要实现另一个对称方向的操作。因此，双向队列需要使用「双向链表」来实现。

我们将双向链表的头结点和尾结点分别看作双向队列的队首和队尾，并且实现在两端都能添加与删除结点。

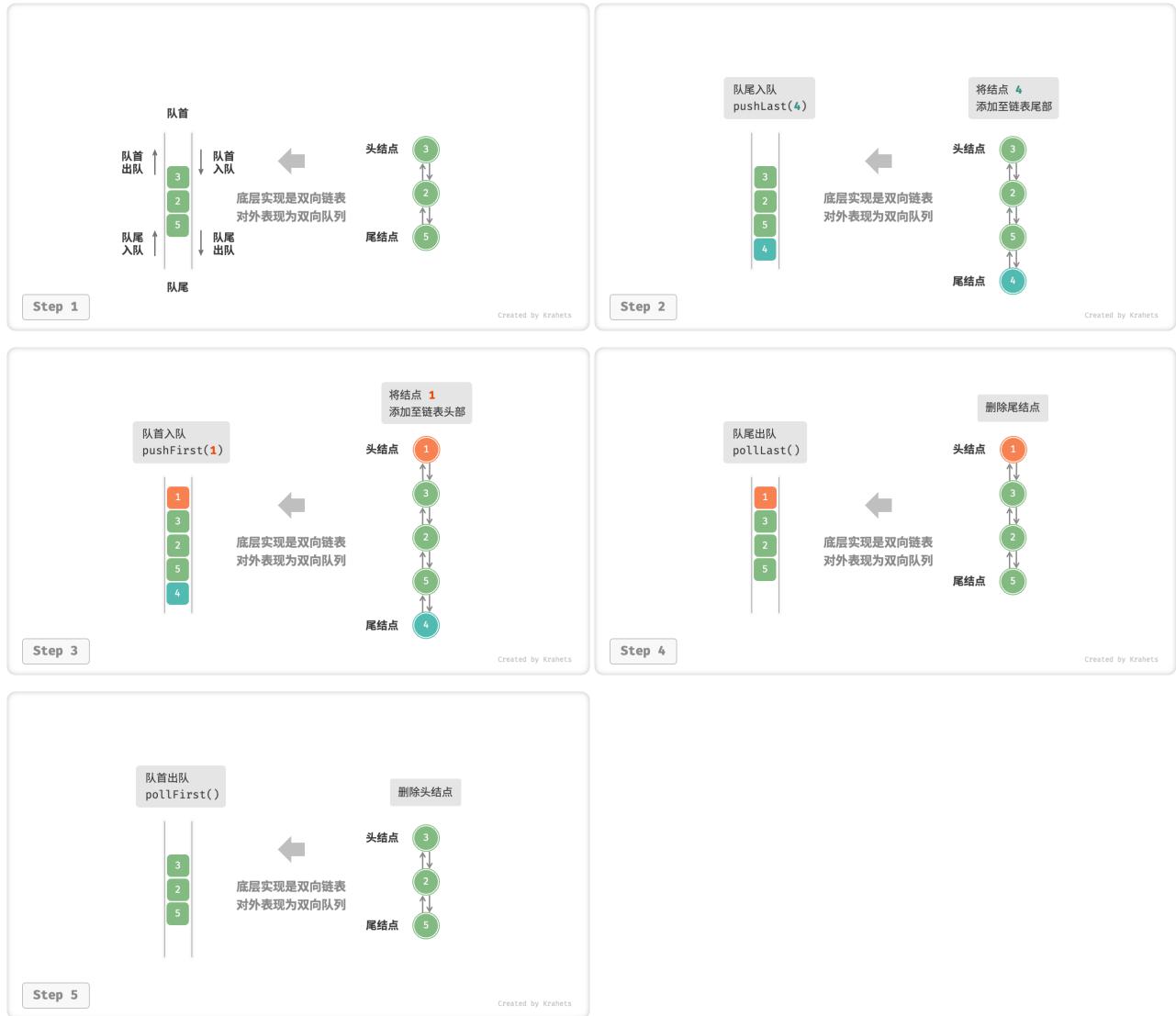


Figure 5-8. 基于链表实现双向队列的入队出队操作

以下是具体实现代码。

```
// === File: linkedlist_deque.cpp ===
/* 双向链表结点 */
```

```
struct DoublyListNode {
    int val;           // 结点值
    DoublyListNode *next; // 后继结点指针
    DoublyListNode *prev; // 前驱结点指针
    DoublyListNode(int val) : val(val), prev(nullptr), next(nullptr) {}
};

/* 基于双向链表实现的双向队列 */
class LinkedListDeque {
private:
    DoublyListNode *front, *rear; // 头结点 front , 尾结点 rear
    int queSize = 0;             // 双向队列的长度

public:
    /* 构造方法 */
    LinkedListDeque() : front(nullptr), rear(nullptr) {}

    /* 析构方法 */
    ~LinkedListDeque() {
        // 释放内存
        DoublyListNode *pre, *cur = front;
        while (cur != nullptr) {
            pre = cur;
            cur = cur->next;
            delete pre;
        }
    }

    /* 获取双向队列的长度 */
    int size() {
        return queSize;
    }

    /* 判断双向队列是否为空 */
    bool isEmpty() {
        return size() == 0;
    }

    /* 入队操作 */
    void push(int num, bool isFront) {
        DoublyListNode *node = new DoublyListNode(num);
        // 若链表为空, 则令 front, rear 都指向 node
        if (isEmpty())
            front = rear = node;
        // 队首入队操作
        else if (isFront) {
```

```
// 将 node 添加至链表头部
front->prev = node;
node->next = front;
front = node; // 更新头结点
// 队尾入队操作
} else {
    // 将 node 添加至链表尾部
    rear->next = node;
    node->prev = rear;
    rear = node; // 更新尾结点
}
queSize++; // 更新队列长度
}

/* 队首入队 */
void pushFirst(int num) {
    push(num, true);
}

/* 队尾入队 */
void pushLast(int num) {
    push(num, false);
}

/* 出队操作 */
int poll(bool isFront) {
    // 若队列为空，直接返回 -1
    if (isEmpty())
        return -1;
    int val;
    // 队首出队操作
    if (isFront) {
        val = front->val; // 暂存头结点值
        // 删除头结点
        DoublyListNode *fNext = front->next;
        if (fNext != nullptr) {
            fNext->prev = nullptr;
            front->next = nullptr;
        }
        front = fNext; // 更新头结点
    }
    // 队尾出队操作
} else {
    val = rear->val; // 暂存尾结点值
    // 删除尾结点
    DoublyListNode *rPrev = rear->prev;
    if (rPrev != nullptr) {
```

```
    rPrev->next = nullptr;
    rear->prev = nullptr;
}
rear = rPrev; // 更新尾结点
}
queSize--; // 更新队列长度
return val;
}

/* 队首出队 */
int pollFirst() {
    return poll(true);
}

/* 队尾出队 */
int pollLast() {
    return poll(false);
}

/* 访问队首元素 */
int peekFirst() {
    return isEmpty() ? -1 : front->val;
}

/* 访问队尾元素 */
int peekLast() {
    return isEmpty() ? -1 : rear->val;
}

/* 返回数组用于打印 */
vector<int> toVector() {
    DoublyListNode *node = front;
    vector<int> res(size());
    for (int i = 0; i < res.size(); i++) {
        res[i] = node->val;
        node = node->next;
    }
    return res;
}
};
```

基于数组的实现

与基于数组实现队列类似，我们也可以使用环形数组来实现双向队列。在实现队列的基础上，增加实现“队首入队”和“队尾出队”方法即可。

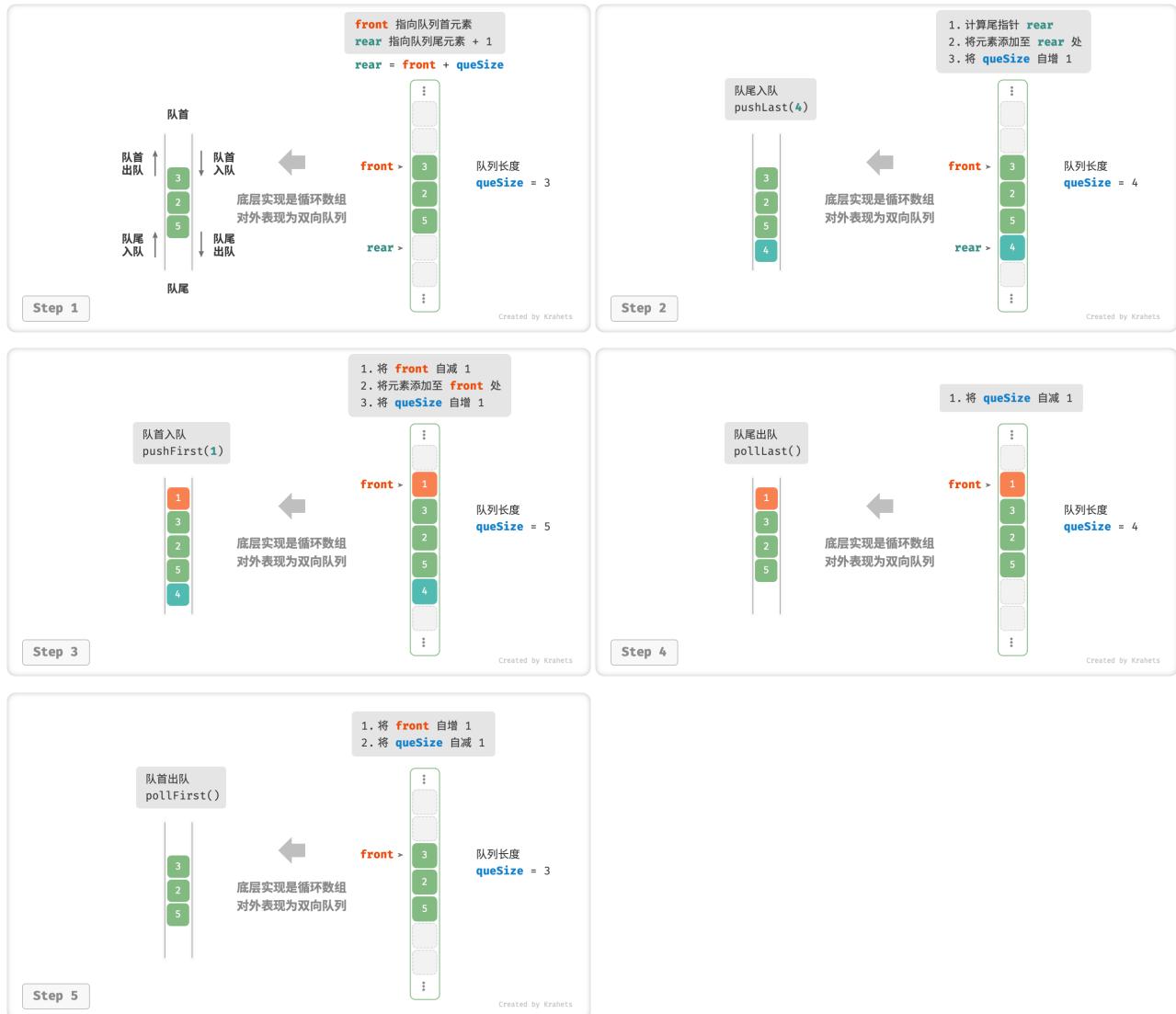


Figure 5-9. 基于数组实现双向队列的入队出队操作

以下是具体实现代码。

```
// === File: array_deque.cpp ===
/* 基于环形数组实现的双向队列 */
class ArrayDeque {
private:
    vector<int> nums; // 用于存储双向队列元素的数组
    int front; // 队首指针, 指向队首元素
    int queSize; // 双向队列长度

public:
    /* 构造方法 */
    ArrayDeque(int capacity) {
```

```
    nums.resize(capacity);
    front = queSize = 0;
}

/* 获取双向队列的容量 */
int capacity() {
    return nums.size();
}

/* 获取双向队列的长度 */
int size() {
    return queSize;
}

/* 判断双向队列是否为空 */
bool isEmpty() {
    return queSize == 0;
}

/* 计算环形数组索引 */
int index(int i) {
    // 通过取余操作实现数组首尾相连
    // 当 i 越过数组尾部后，回到头部
    // 当 i 越过数组头部后，回到尾部
    return (i + capacity()) % capacity();
}

/* 队首入队 */
void pushFirst(int num) {
    if (queSize == capacity()) {
        cout << " 双向队列已满" << endl;
        return;
    }
    // 队首指针向左移动一位
    // 通过取余操作，实现 front 越过数组头部后回到尾部
    front = index(front - 1);
    // 将 num 添加至队首
    nums[front] = num;
    queSize++;
}

/* 队尾入队 */
void pushLast(int num) {
    if (queSize == capacity()) {
        cout << " 双向队列已满" << endl;
        return;
    }
```

```
}

// 计算尾指针，指向队尾索引 + 1
int rear = index(front + queSize);
// 将 num 添加至队尾
nums[rear] = num;
queSize++;
}

/* 队首出队 */
int pollFirst() {
    int num = peekFirst();
    // 队首指针向后移动一位
    front = index(front + 1);
    queSize--;
    return num;
}

/* 队尾出队 */
int pollLast() {
    int num = peekLast();
    queSize--;
    return num;
}

/* 访问队首元素 */
int peekFirst() {
    if (isEmpty())
        throw out_of_range(" 双向队列为空");
    return nums[front];
}

/* 访问队尾元素 */
int peekLast() {
    if (isEmpty())
        throw out_of_range(" 双向队列为空");
    // 计算尾元素索引
    int last = index(front + queSize - 1);
    return nums[last];
}

/* 返回数组用于打印 */
vector<int> toVector() {
    // 仅转换有效长度范围内的列表元素
    vector<int> res(queSize);
    for (int i = 0, j = front; i < queSize; i++, j++) {
        res[i] = nums[index(j)];
    }
}
```

```
    }
    return res;
}
};
```

5.4. 小结

- 栈是一种遵循先入后出的数据结构，可以使用数组或链表实现。
- 在时间效率方面，栈的数组实现具有更好的平均效率，但扩容时会导致单次入栈操作的时间复杂度劣化至 $O(n)$ 。相对地，栈的链表实现具有更加稳定的效率表现。
- 在空间效率方面，栈的数组实现会造成一定空间浪费，然而链表结点比数组元素占用内存更大。
- 队列是一种遵循先入先出的数据结构，可以使用数组或链表实现。对于两种实现的时间效率与空间效率对比，与上述栈的结论相同。
- 双向队列的两端都可以添加与删除元素。

6. 散列表

6.1. 哈希表

哈希表通过建立「键 key」和「值 value」之间的映射，实现高效的元素查找。具体地，输入一个 key，在哈希表中查询并获取 value，时间复杂度为 $O(1)$ 。

例如，给定一个包含 n 个学生的数据库，每个学生有“姓名 name”和“学号 id”两项数据，希望实现一个查询功能：输入一个学号，返回对应的姓名，则可以使用哈希表实现。



Figure 6-1. 哈希表的抽象表示

6.1.1. 哈希表效率

除了哈希表之外，还可以使用以下数据结构来实现上述查询功能：

1. 无序数组：每个元素为 [学号, 姓名]；
2. 有序数组：将 1. 中的数组按照学号从小到大排序；
3. 链表：每个结点的值为 [学号, 姓名]；
4. 二叉搜索树：每个结点的值为 [学号, 姓名]，根据学号大小来构建树；

使用上述方法，各项操作的时间复杂度如下表所示（在此不做赘述，详解可见 [二叉搜索树章节](#)）。无论是查找元素、还是增删元素，哈希表的时间复杂度都是 $O(1)$ ，全面胜出！

	无序数组	有序数组	链表	二叉搜索树	哈希表
查找元素	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
插入元素	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
删除元素	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$

6.1.2. 哈希表常用操作

哈希表的基本操作包括 **初始化、查询操作、添加与删除键值对**。

```
// === File: hash_map.cpp ===
/* 初始化哈希表 */
unordered_map<int, string> map;

/* 添加操作 */
// 在哈希表中添加键值对 (key, value)
map[12836] = "小哈";
map[15937] = "小啰";
map[16750] = "小算";
map[13276] = "小法";
map[10583] = "小鸭";

/* 查询操作 */
// 向哈希表输入键 key , 得到值 value
string name = map[15937];

/* 删除操作 */
// 在哈希表中删除键值对 (key, value)
map.erase(10583);
```

遍历哈希表有三种方式，即 **遍历键值对、遍历键、遍历值**。

```
// === File: hash_map.cpp ===
/* 遍历哈希表 */
// 遍历键值对 key->value
for (auto kv: map) {
    cout << kv.first << " -> " << kv.second << endl;
}

// 单独遍历键 key
for (auto key: map) {
    cout << key.first << endl;
}

// 单独遍历值 value
```

```
for (auto val: map) {
    cout << val.second << endl;
}
```

6.1.3. 哈希函数

哈希表中存储元素的数据结构被称为「桶 Bucket」，底层实现可能是数组、链表、二叉树（红黑树），或是它们的组合。

最简单地，我们可以仅用一个「数组」来实现哈希表。首先，将所有 value 放入数组中，那么每个 value 在数组中都有唯一的「索引」。显然，访问 value 需要给定索引，而为了建立 key 和索引之间的映射关系，我们需要使用「哈希函数 Hash Function」。

设数组为 `bucket`，哈希函数为 `f(x)`，输入键为 `key`。那么获取 value 的步骤为：

1. 通过哈希函数计算出索引，即 `index = f(key)`；
2. 通过索引在数组中获取值，即 `value = bucket[index]`；

以上述学生数据 `key 学号 -> value 姓名` 为例，我们可以将「哈希函数」设计为

$$f(x) = x \% 100$$

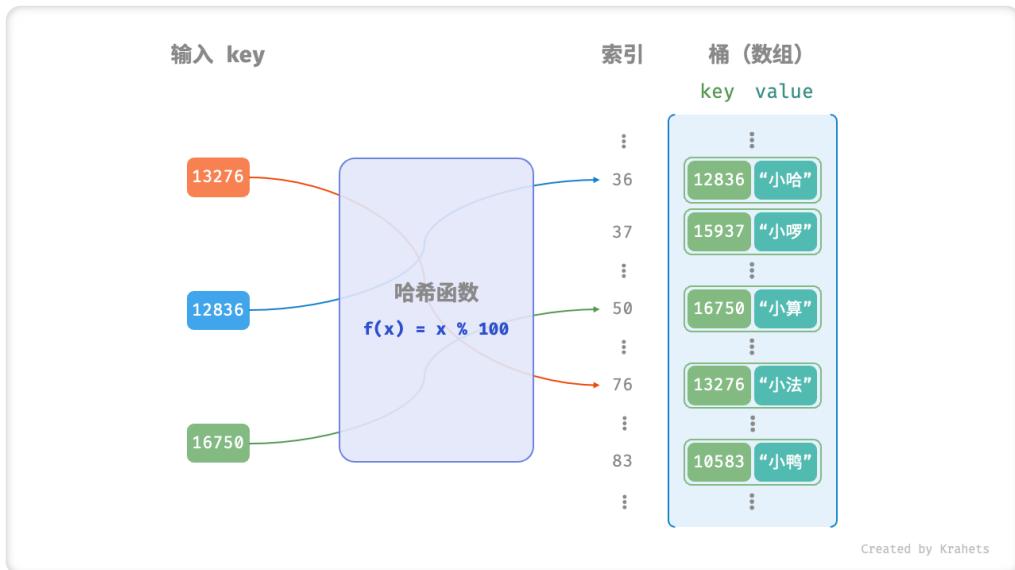


Figure 6-2. 简单哈希函数示例

```
// === File: array_hash_map.cpp ===
/* 键值对 int->String */
struct Entry {
public:
    int key;
```

```
string val;
Entry(int key, string val) {
    this->key = key;
    this->val = val;
}
};

/* 基于数组简易实现的哈希表 */
class ArrayHashMap {
private:
    vector<Entry*> bucket;
public:
    ArrayHashMap() {
        // 初始化一个长度为 100 的桶（数组）
        bucket = vector<Entry*>(100);
    }

    /* 哈希函数 */
    int hashFunc(int key) {
        int index = key % 100;
        return index;
    }

    /* 查询操作 */
    string get(int key) {
        int index = hashFunc(key);
        Entry* pair = bucket[index];
        if (pair == nullptr)
            return nullptr;
        return pair->val;
    }

    /* 添加操作 */
    void put(int key, string val) {
        Entry* pair = new Entry(key, val);
        int index = hashFunc(key);
        bucket[index] = pair;
    }

    /* 删除操作 */
    void remove(int key) {
        int index = hashFunc(key);
        // 置为 nullptr，代表删除
        bucket[index] = nullptr;
    }
}
```

```
/* 获取所有键值对 */
vector<Entry*> entrySet() {
    vector<Entry*> entrySet;
    for (Entry* pair: bucket) {
        if (pair != nullptr) {
            entrySet.push_back(pair);
        }
    }
    return entrySet;
}

/* 获取所有键 */
vector<int> keySet() {
    vector<int> keySet;
    for (Entry* pair: bucket) {
        if (pair != nullptr) {
            keySet.push_back(pair->key);
        }
    }
    return keySet;
}

/* 获取所有值 */
vector<string> valueSet() {
    vector<string> valueSet;
    for (Entry* pair: bucket) {
        if (pair != nullptr){
            valueSet.push_back(pair->val);
        }
    }
    return valueSet;
}

/* 打印哈希表 */
void print() {
    for (Entry* kv: entrySet()) {
        cout << kv->key << " -> " << kv->val << endl;
    }
}
};
```

6.1.4. 哈希冲突

细心的同学可能会发现，哈希函数 $f(x) = x \% 100$ 会在某些情况下失效。具体地，当输入的 key 后两位相同时，哈希函数的计算结果也相同，指向同一个 value。例如，分别查询两个学号 12836 和 20336，则有

$$f(12836) = f(20336) = 36$$

两个学号指向了同一个姓名，这明显是不对的，我们将这种现象称为「哈希冲突 Hash Collision」。如何避免哈希冲突的问题将被留在下章讨论。

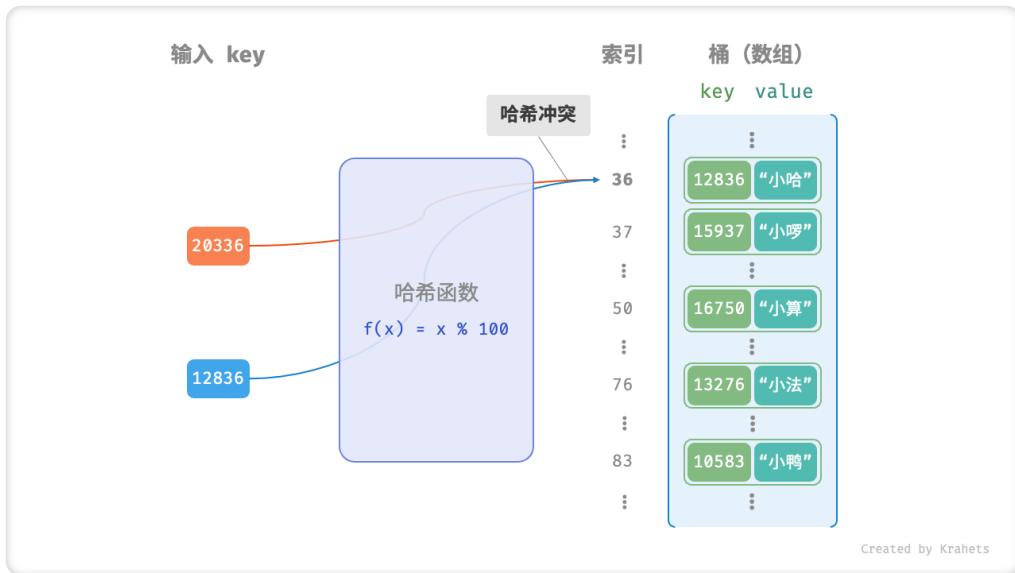


Figure 6-3. 哈希冲突示例

综上所述，一个优秀的「哈希函数」应该具备以下特性：

- 尽量少地发生哈希冲突；
- 时间复杂度 $O(1)$ ，计算尽可能高效；
- 空间使用率高，即“键值对占用空间 / 哈希表总占用空间”尽可能大；

6.2. 哈希冲突

理想情况下，哈希函数应该为每个输入产生唯一的输出，使得 key 和 value 一一对应。而实际上，往往存在向哈希函数输入不同的 key 而产生相同输出的情况，这种情况被称为「哈希冲突 Hash Collision」。哈希冲突会导致查询结果错误，从而严重影响哈希表的可用性。

那么，为什么会出现哈希冲突呢？本质上讲，由于哈希函数的输入空间往往远大于输出空间，因此不可避免地会出现多个输入产生相同输出的情况，即为哈希冲突。比如，输入空间是全体整数，输出空间是一个固定大小的桶（数组）的索引范围，那么必定会有多个整数同时映射到一个桶索引。

为了缓解哈希冲突，一方面，**我们可以通过哈希表扩容来减小冲突概率**。极端情况下，当输入空间和输出空间大小相等时，哈希表就等价于数组了，可谓“大力出奇迹”。

另一方面，**考虑通过优化哈希表的表示方式以缓解哈希冲突**，常见的方法有「链式地址」和「开放寻址」。

6.2.1. 哈希表扩容

「负载因子 Load Factor」定义为 哈希表中元素数量除以桶槽数量（即数组大小），代表哈希冲突的严重程度。

负载因子常用作哈希表扩容的触发条件。比如在 Java 中，当负载因子 > 0.75 时则触发扩容，将 HashMap 大小扩充至原先的 2 倍。

与数组扩容类似，哈希表扩容操作的开销很大，因为需要将所有键值对从原哈希表依次移动至新哈希表。

6.2.2. 链式地址

在原始哈希表中，桶内的每个地址只能存储一个元素（即键值对）。考虑将单个元素转化成一个链表，将所有冲突元素都存储在一个链表中。



Figure 6-4. 链式地址

链式地址下，哈希表操作方法为：

- **查询元素**：先将 key 输入到哈希函数得到桶内索引，即可访问链表头结点，再通过遍历链表查找对应 value。
- **添加元素**：先通过哈希函数访问链表头部，再将结点（即键值对）添加到链表头部即可。
- **删除元素**：同样先根据哈希函数结果访问链表头部，再遍历链表查找对应结点，删除之即可。

链式地址虽然解决了哈希冲突问题，但仍存在局限性，包括：

- 占用空间变大，因为链表或二叉树包含结点指针，相比于数组更加耗费内存空间；
- 查询效率降低，因为需要线性遍历链表来查找对应元素；

为了提升操作效率，可以把「链表」转化为「AVL 树」或「红黑树」，将查询操作的时间复杂度优化至 $O(\log n)$

◦

6.2.3. 开放寻址

「开放寻址」不引入额外数据结构，而是通过“多次探测”来解决哈希冲突。根据探测方法的不同，主要分为线性探测、平方探测、多次哈希。

线性探测

「线性探测」使用固定步长的线性查找来解决哈希冲突。

插入元素：如果出现哈希冲突，则从冲突位置向后线性遍历（步长一般取 1），直到找到一个空位，则将元素插入到该空位中。

查找元素：若出现哈希冲突，则使用相同步长执行线性查找，会遇到两种情况：

1. 找到对应元素，返回 value 即可；
2. 若遇到空位，则说明查找键值对不在哈希表中；

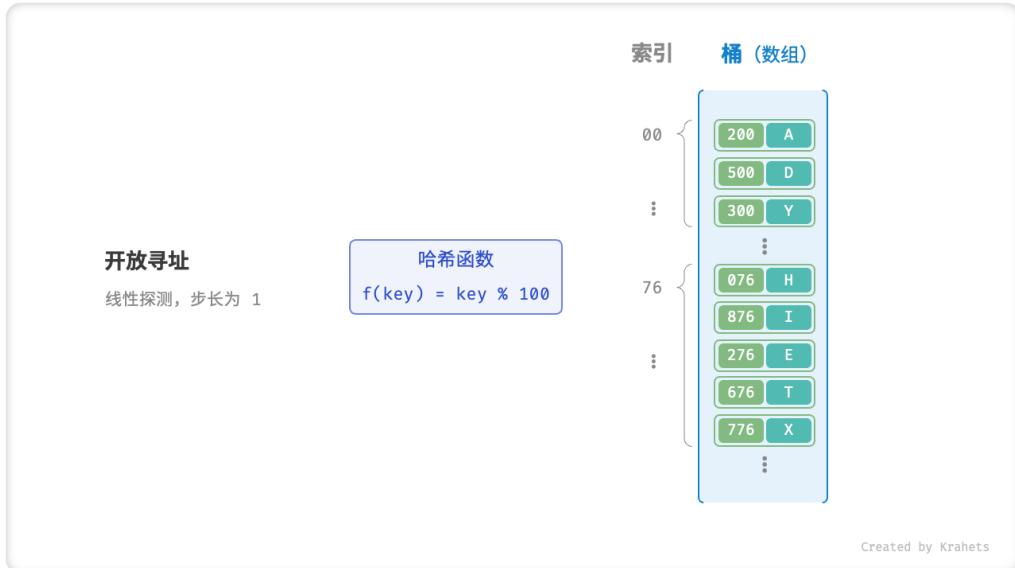


Figure 6-5. 线性探测

线性探测存在以下缺陷：

- **不能直接删除元素。**删除元素会导致桶内出现一个空位，在查找其他元素时，该空位有可能导致程序认为元素不存在（即上述第 2. 种情况）。因此需要借助一个标志位来标记删除元素。
- **容易产生聚集。**桶内被占用的连续位置越长，这些连续位置发生哈希冲突的可能性越大，从而进一步促进这一位置的“聚堆生长”，最终导致增删查改操作效率的劣化。

多次哈希

顾名思义，「多次哈希」的思路是使用多个哈希函数 $f_1(x), f_2(x), f_3(x), \dots$ 进行探测。

插入元素：若哈希函数 $f_1(x)$ 出现冲突，则尝试 $f_2(x)$ ，以此类推……直到找到空位后插入元素。

查找元素：以相同的哈希函数顺序查找，存在两种情况：

1. 找到目标元素，则返回之；
2. 到空位或已尝试所有哈希函数，说明哈希表中无此元素；

相比于「线性探测」，「多次哈希」方法更不容易产生聚集，代价是多个哈希函数增加了额外计算量。



工业界方案

Java 采用「链式地址」。在 JDK 1.8 之后，HashMap 内数组长度大于 64 时，长度大于 8 的链表会被转化为「红黑树」，以提升查找性能。

Python 采用「开放寻址」。字典 dict 使用伪随机数进行探测。

6.3. 小结

- 向哈希表中输入一个键 key，查询到值 value 的时间复杂度为 $O(1)$ ，非常高效。
- 哈希表的常用操作包括查询、添加与删除键值对、遍历键值对等。
- 哈希函数将 key 映射到桶（数组）索引，从而访问到对应的值 value。
- 两个不同的 key 经过哈希函数可能得到相同的桶索引，进而发生哈希冲突，导致查询错误。
- 缓解哈希冲突的途径有两种：哈希表扩容、优化哈希表的表示方式。
- 负载因子定义为哈希表中元素数量除以桶槽数量，体现哈希冲突的严重程度，常用作哈希表扩容的触发条件。与数组扩容的原理类似，哈希表扩容操作开销也很大。
- 链式地址考虑将单个元素转化成一个链表，将所有冲突元素都存储在一个链表中，从而解决哈希冲突。链表过长会导致查询效率变低，可以通过把链表转化为 AVL 树或红黑树来解决。
- 开放寻址通过多次探测来解决哈希冲突。线性探测使用固定步长，缺点是不能删除元素且容易产生聚集。多次哈希使用多个哈希函数进行探测，相对线性探测不容易产生聚集，代价是多个哈希函数增加了计算量。
- 在工业界中，Java 的 HashMap 采用链式地址、Python 的 Dict 采用开放寻址。

7. 树

7.1. 二叉树

「二叉树 Binary Tree」是一种非线性数据结构，代表着祖先与后代之间的派生关系，体现着“一分为二”的分治逻辑。类似于链表，二叉树也是以结点为单位存储的，结点包含「值」和两个「指针」。

```
/* 链表结点结构体 */
struct TreeNode {
    int val;           // 结点值
    TreeNode *left;    // 左子结点指针
    TreeNode *right;   // 右子结点指针
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

结点的两个指针分别指向「左子结点 Left Child Node」和「右子结点 Right Child Node」，并且称该结点为两个子结点的「父结点 Parent Node」。给定二叉树某结点，将左子结点以下的树称为该结点的「左子树 Left Subtree」，右子树同理。

除了叶结点外，每个结点都有子结点和子树。例如，若将下图的「结点 2」看作父结点，那么其左子结点和右子结点分别为「结点 4」和「结点 5」，左子树和右子树分别为「结点 4 及其以下结点形成的树」和「结点 5 及其以下结点形成的树」。

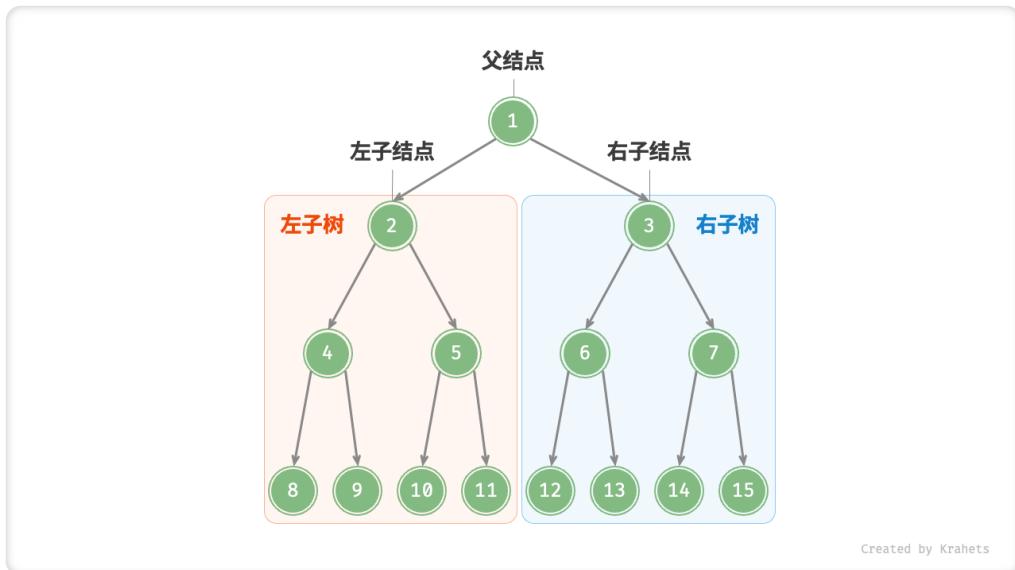


Figure 7-1. 父结点、子结点、子树

7.1.1. 二叉树常见术语

二叉树的术语较多，建议尽量理解并记住。后续可能遗忘，可以在需要使用时回来查看确认。

- 「根结点 Root Node」：二叉树最顶层的结点，其没有父结点；
- 「叶结点 Leaf Node」：没有子结点的结点，其两个指针都指向 null；
- 结点所处「层 Level」：从顶到底依次增加，根结点所处层为 1；
- 结点「度 Degree」：结点的子结点数量。二叉树中，度的范围是 0, 1, 2；
- 「边 Edge」：连接两个结点的边，即结点指针；
- 二叉树「高度」：二叉树中根结点到最远叶结点走过边的数量；
- 结点「深度 Depth」：根结点到该结点走过边的数量；
- 结点「高度 Height」：最远叶结点到该结点走过边的数量；

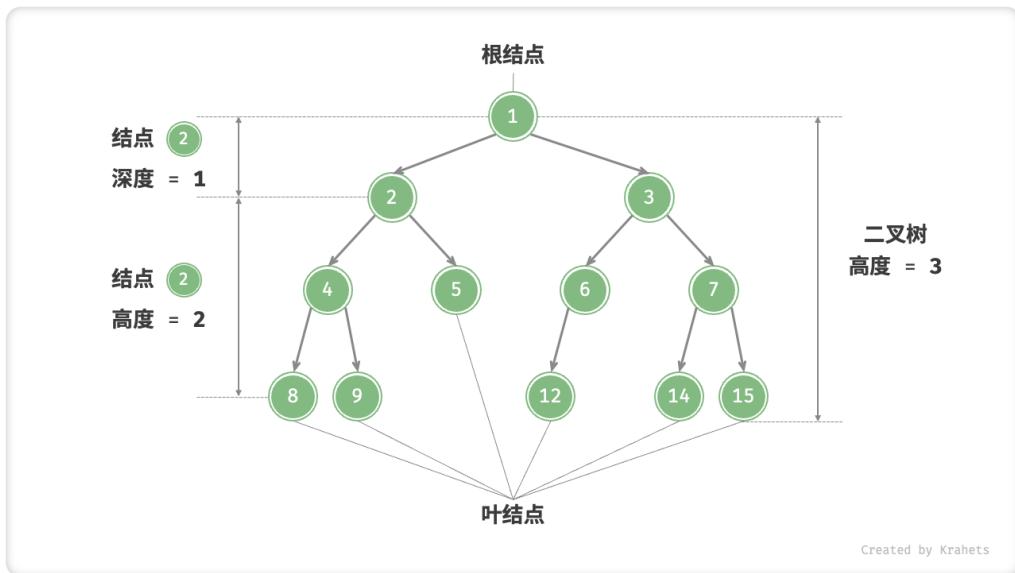


Figure 7-2. 二叉树的常用术语



高度与深度的定义

值得注意，我们通常将「高度」和「深度」定义为“走过边的数量”，而有些题目或教材会将其定义为“走过结点的数量”，此时高度或深度都需要 + 1。

7.1.2. 二叉树基本操作

初始化二叉树。与链表类似，先初始化结点，再构建引用指向（即指针）。

```
// === File: binary_tree.cpp ===
/* 初始化二叉树 */
// 初始化结点
```

```

TreeNode* n1 = new TreeNode(1);
TreeNode* n2 = new TreeNode(2);
TreeNode* n3 = new TreeNode(3);
TreeNode* n4 = new TreeNode(4);
TreeNode* n5 = new TreeNode(5);
// 构建引用指向 (即指针)
n1->left = n2;
n1->right = n3;
n2->left = n4;
n2->right = n5;

```

插入与删除结点。与链表类似，插入与删除结点都可以通过修改指针实现。

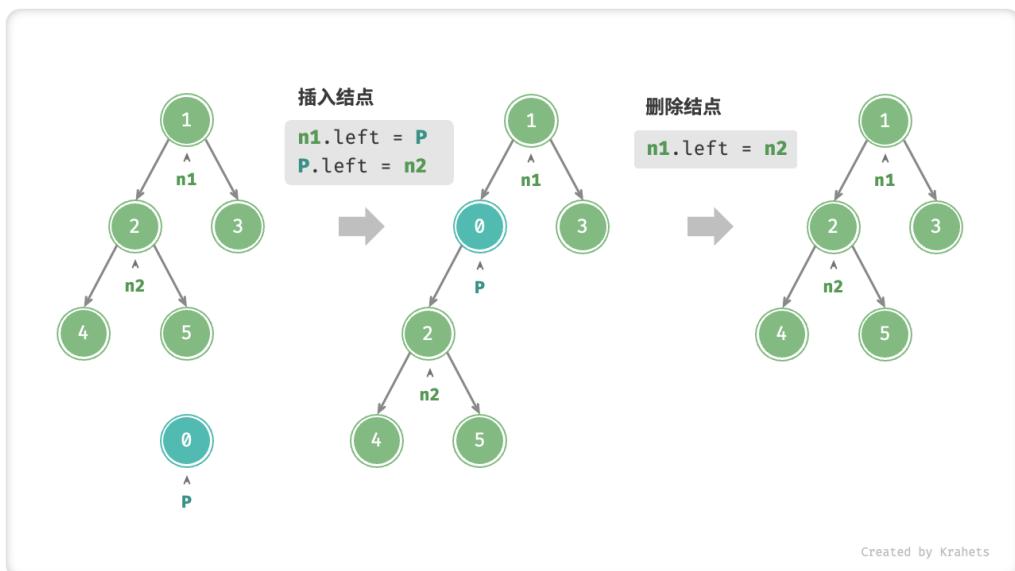


Figure 7-3. 在二叉树中插入与删除结点

```

// === File: binary_tree.cpp ===
/* 插入与删除结点 */
TreeNode* P = new TreeNode(0);
// 在 n1 -> n2 中间插入结点 P
n1->left = P;
P->left = n2;
// 删除结点 P
n1->left = n2;

```



插入结点会改变二叉树的原有逻辑结构，删除结点往往意味着删除了该结点的所有子树。因此，二叉树中的插入与删除一般都是由一套操作配合完成的，这样才能实现有意义的操作。

7.1.3. 常见二叉树类型

完美二叉树

「完美二叉树 Perfect Binary Tree」的所有层的结点都被完全填满。在完美二叉树中，所有结点的度 = 2；若树高度 = h ，则结点总数 = $2^{h+1} - 1$ ，呈标准的指数级关系，反映着自然界中常见的细胞分裂。



在中文社区中，完美二叉树常被称为「满二叉树」，请注意与完满二叉树区分。

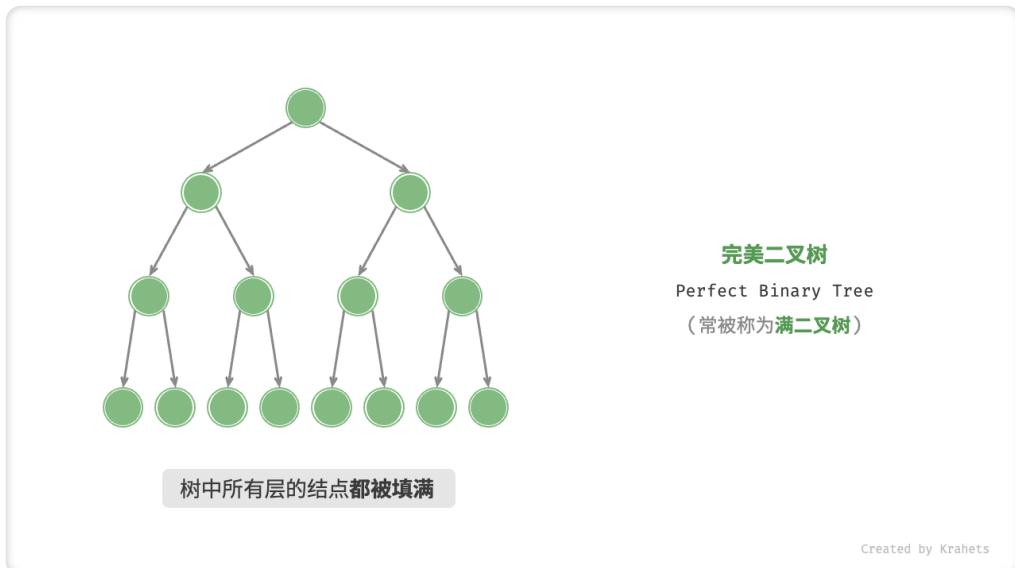


Figure 7-4. 完美二叉树

完全二叉树

「完全二叉树 Complete Binary Tree」只有最底层的结点未被填满，且最底层结点尽量靠左填充。

完全二叉树非常适合用数组来表示。如果按照层序遍历序列的顺序来存储，那么空结点 `null` 一定全部出现在序列的尾部，因此我们就可以不用存储这些 `null` 了。

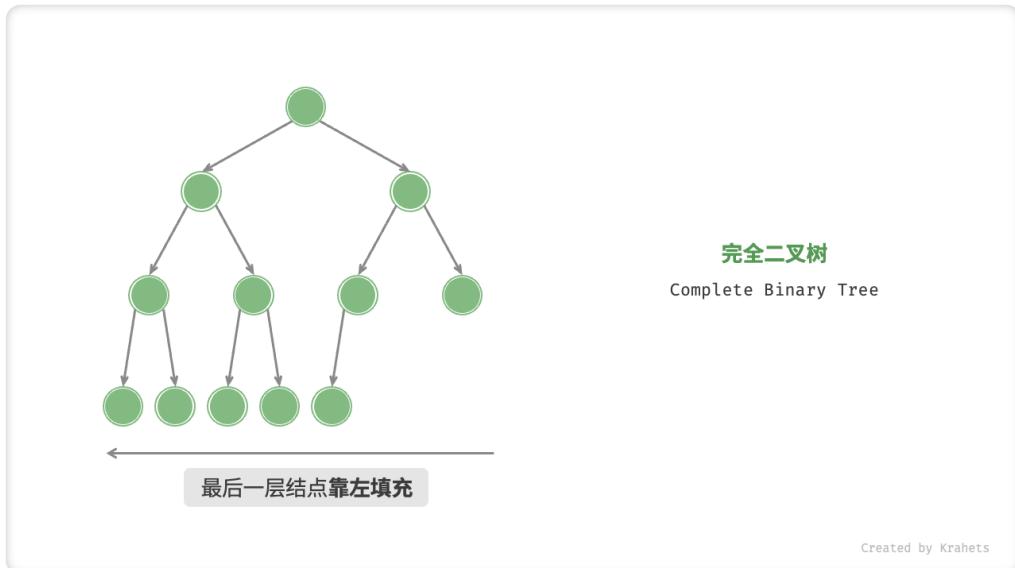


Figure 7-5. 完全二叉树

完满二叉树

「完满二叉树 Full Binary Tree」除了叶结点之外，其余所有结点都有两个子结点。

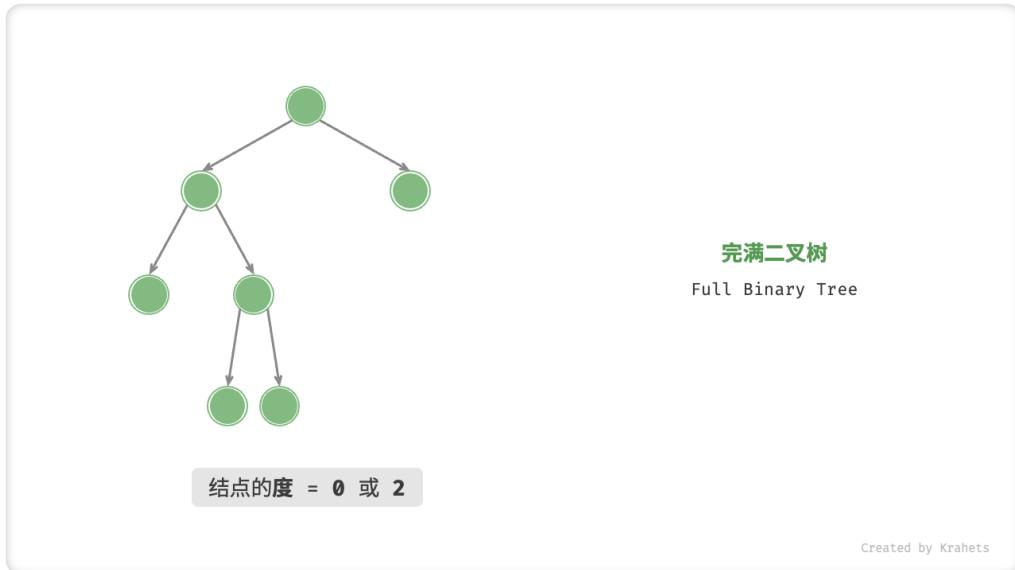


Figure 7-6. 完满二叉树

平衡二叉树

「平衡二叉树 Balanced Binary Tree」中任意结点的左子树和右子树的高度之差的绝对值 ≤ 1 。

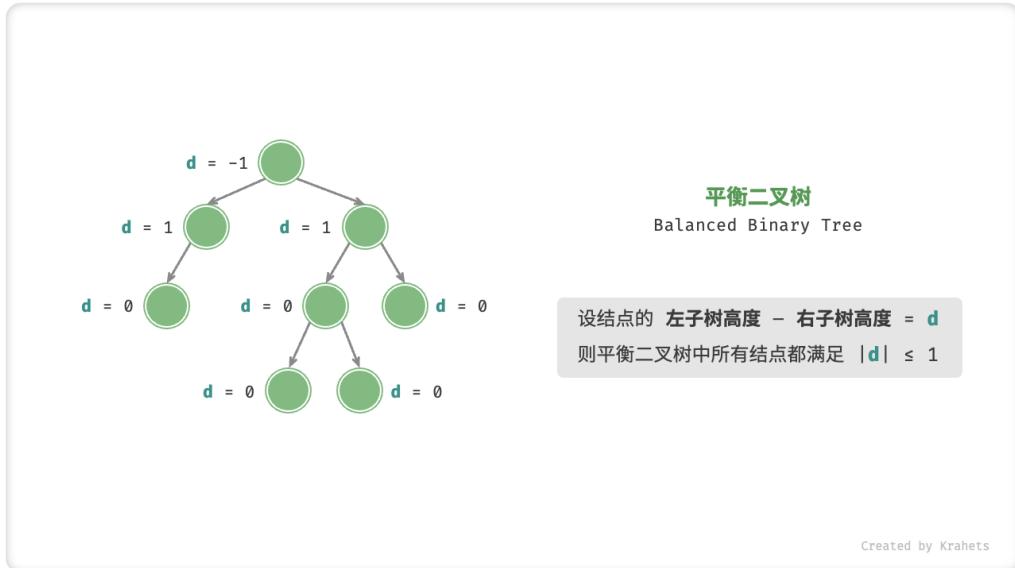


Figure 7-7. 平衡二叉树

7.1.4. 二叉树的退化

当二叉树的每层的结点都被填满时，达到「完美二叉树」；而当所有结点都偏向一边时，二叉树退化为「链表」。

- 完美二叉树是一个二叉树的“最佳状态”，可以完全发挥出二叉树“分治”的优势；
- 链表则是另一个极端，各项操作都变为线性操作，时间复杂度退化至 $O(n)$ ；

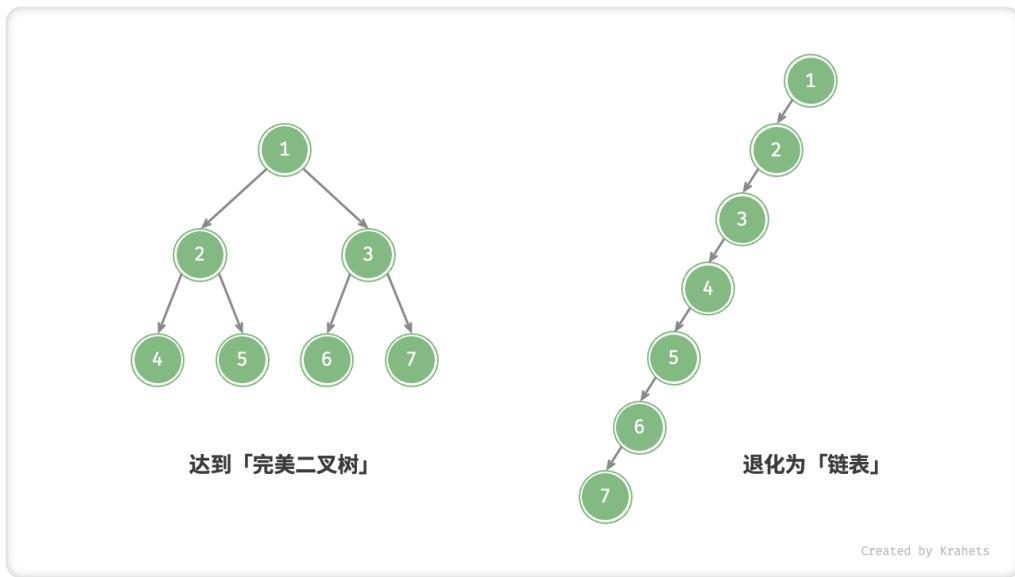


Figure 7-8. 二叉树的最佳与最差结构差情况

如下表所示，在最佳和最差结构下，二叉树的叶结点数量、结点总数、高度等达到极大或极小值。

	完美二叉树	链表
第 i 层的结点数量	2^{i-1}	1
树的高度为 h 时的叶结点数量	2^h	1
树的高度为 h 时的结点总数	$2^{h+1} - 1$	$h + 1$
树的结点总数为 n 时的高度	$\log_2(n + 1) - 1$	$n - 1$

7.1.5. 二叉树表示方式 *

我们一般使用二叉树的「链表表示」，即存储单位为结点 `TreeNode`，结点之间通过指针（引用）相连接。本文前述示例代码展示了二叉树在链表表示下的各项基本操作。

那能否可以用「数组表示」二叉树呢？答案是肯定的。先来分析一个简单案例，给定一个「完美二叉树」，将结点按照层序遍历的顺序编号（从 0 开始），那么可以推导得出父结点索引与子结点索引之间的「映射公式」：设结点的索引为 i ，则该结点的左子结点索引为 $2i + 1$ 、右子结点索引为 $2i + 2$ 。

本质上，映射公式的作用就是链表中的指针。对于层序遍历序列中的任意结点，我们都可以使用映射公式来访问子结点。因此，可以直接使用层序遍历序列（即数组）来表示完美二叉树。

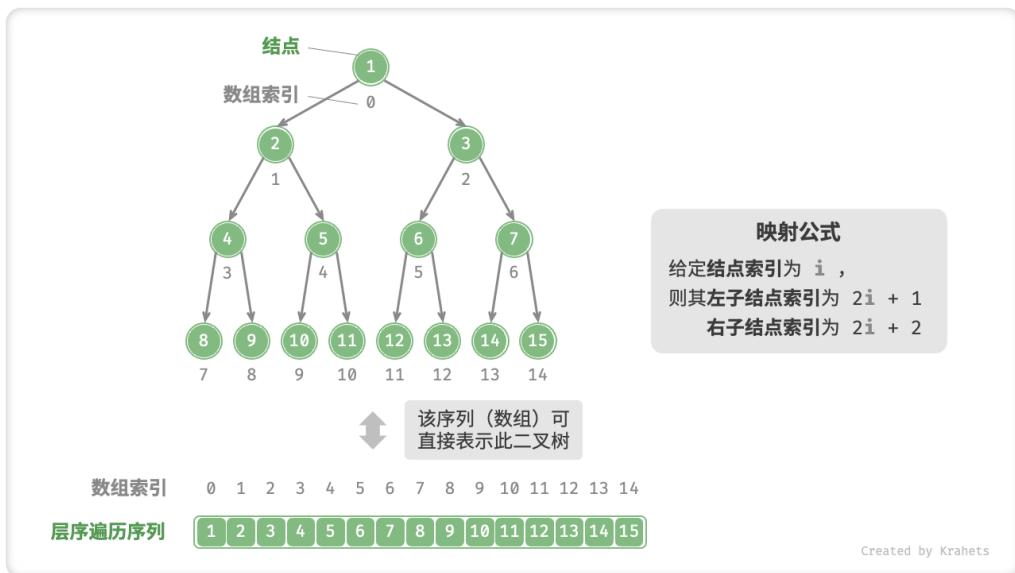


Figure 7-9. 完美二叉树的数组表示

然而，完美二叉树只是个例，二叉树中间层往往存在许多空结点（即 `null`），而层序遍历序列并不包含这些空结点，并且我们无法单凭序列来猜测空结点的数量和分布位置，即理论上存在许多种二叉树都符合该层序遍历序列。显然，这种情况无法使用数组来存储二叉树。

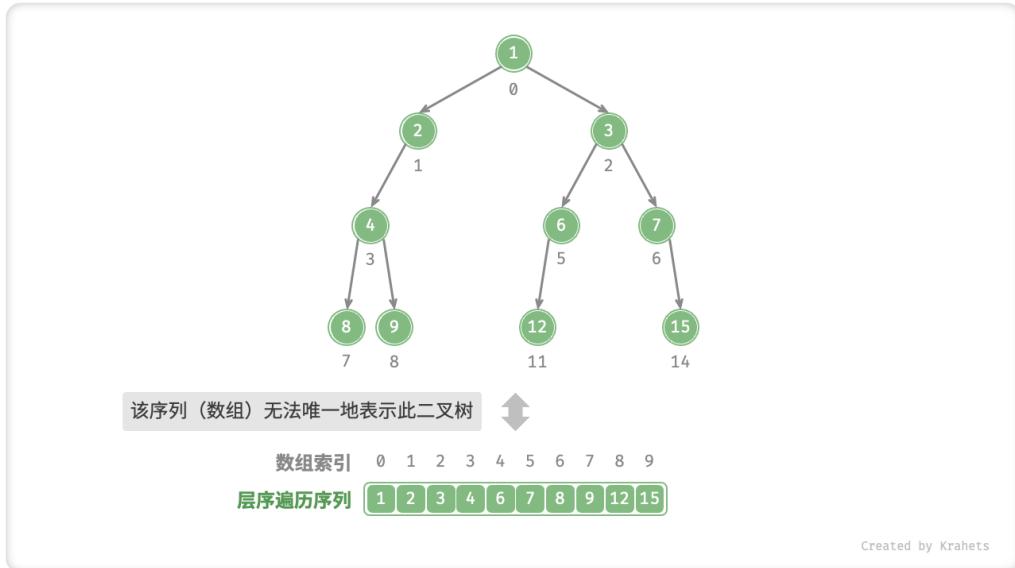


Figure 7-10. 给定数组对应多种二叉树可能性

为了解决此问题，考虑按照完美二叉树的形式来表示所有二叉树，即在序列中使用特殊符号来显式地表示“空位”。如下图所示，这样处理后，序列（数组）就可以唯一表示二叉树了。

```
/* 二叉树的数组表示 */
// 为了符合数据类型为 int，使用 int 最大值标记空位
// 该方法的使用前提是所有结点的值 = INT_MAX
vector<int> tree = { 1, 2, 3, 4, INT_MAX, 6, 7, 8, 9, INT_MAX, INT_MAX, 12, INT_MAX, INT_MAX, 15 };
```

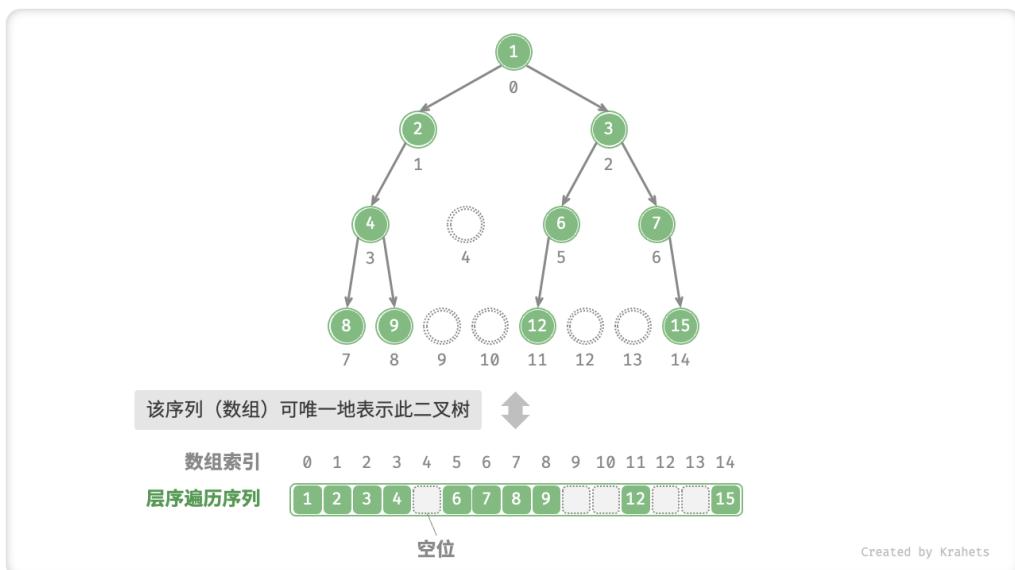


Figure 7-11. 任意类型二叉树的数组表示

回顾「完全二叉树」的定义，其只有最底层有空结点，并且最底层的结点尽量靠左，因而所有空结点都一定出

现在层序遍历序列的末尾。因为我们先验地确定了空位的位置，所以在使用数组表示完全二叉树时，可以省略存储“空位”。因此，完全二叉树非常适合使用数组来表示。

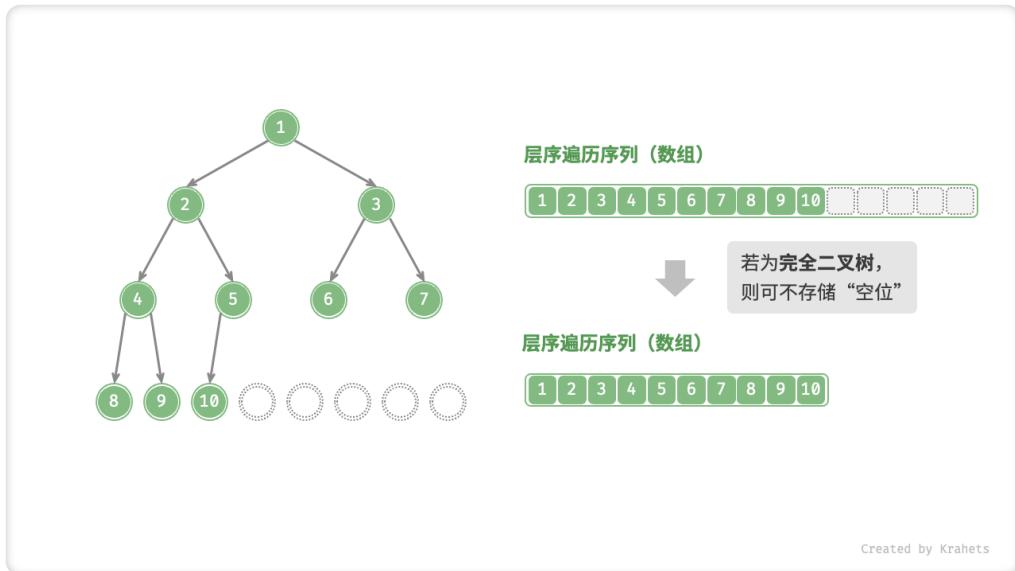


Figure 7-12. 完全二叉树的数组表示

数组表示有两个优点：一是不需要存储指针，节省空间；二是可以随机访问结点。然而，当二叉树中的“空位”很多时，数组中只包含很少结点的数据，空间利用率很低。

7.2. 二叉树遍历

从物理结构角度看，树是一种基于链表的数据结构，因此遍历方式也是通过指针（即引用）逐个遍历结点。同时，树还是一种非线性数据结构，这导致遍历树比遍历链表更加复杂，需要使用搜索算法来实现。

常见的二叉树遍历方式有层序遍历、前序遍历、中序遍历、后序遍历。

7.2.1. 层序遍历

「层序遍历 Level-Order Traversal」从顶到底、一层一层地遍历二叉树，并在每层中按照从左到右的顺序访问结点。

层序遍历本质上是「广度优先搜索 Breadth-First Traversal」，其体现着一种“一圈一圈向外”的层进遍历方式。

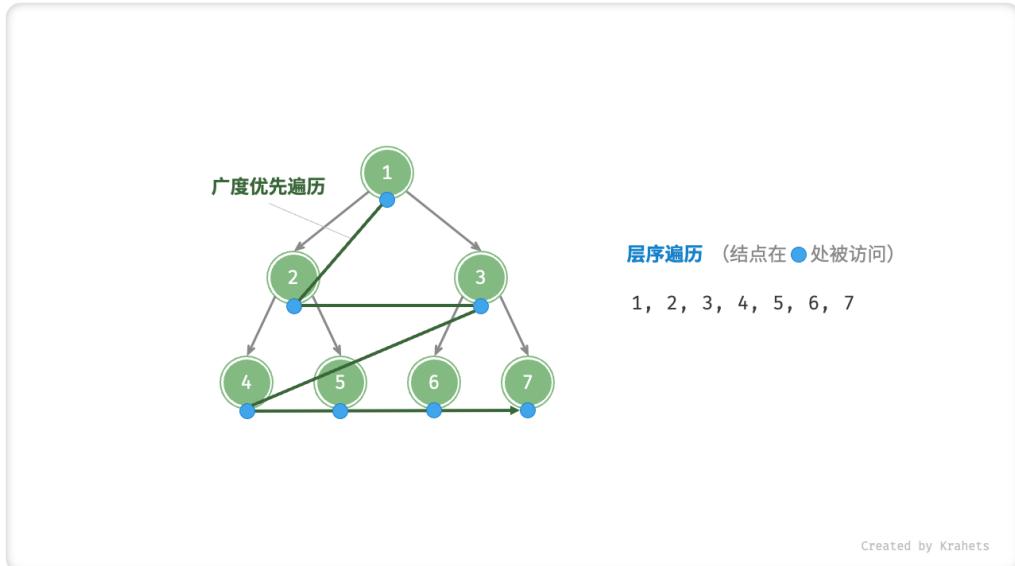


Figure 7-13. 二叉树的层序遍历

算法实现

广度优先遍历一般借助「队列」来实现。队列的规则是“先进先出”，广度优先遍历的规则是“一层层平推”，两者背后的思想是一致的。

```
// === File: binary_tree_bfs.cpp ===
/* 层序遍历 */
vector<int> levelOrder(TreeNode* root) {
    // 初始化队列，加入根结点
    queue<TreeNode*> queue;
    queue.push(root);
    // 初始化一个列表，用于保存遍历序列
    vector<int> vec;
    while (!queue.empty()) {
        TreeNode* node = queue.front();
        queue.pop();           // 队列出队
        vec.push_back(node->val); // 保存结点值
        if (node->left != nullptr)
            queue.push(node->left); // 左子结点入队
        if (node->right != nullptr)
            queue.push(node->right); // 右子结点入队
    }
    return vec;
}
```

复杂度分析

时间复杂度：所有结点被访问一次，使用 $O(n)$ 时间，其中 n 为结点数量。

空间复杂度：当为满二叉树时达到最差情况，遍历到底层前，队列中最多同时存在 $\frac{n+1}{2}$ 个结点，使用 $O(n)$ 空间。

7.2.2. 前序、中序、后序遍历

相对地，前、中、后序遍历皆属于「深度优先遍历 Depth-First Traversal」，其体现着一种“先走到尽头，再回头继续”的回溯遍历方式。

如下图所示，左侧是深度优先遍历的示意图，右上方是对应的递归实现代码。深度优先遍历就像是绕着整个二叉树的外围“走”一圈，走的过程中，在每个结点都会遇到三个位置，分别对应前序遍历、中序遍历、后序遍历。

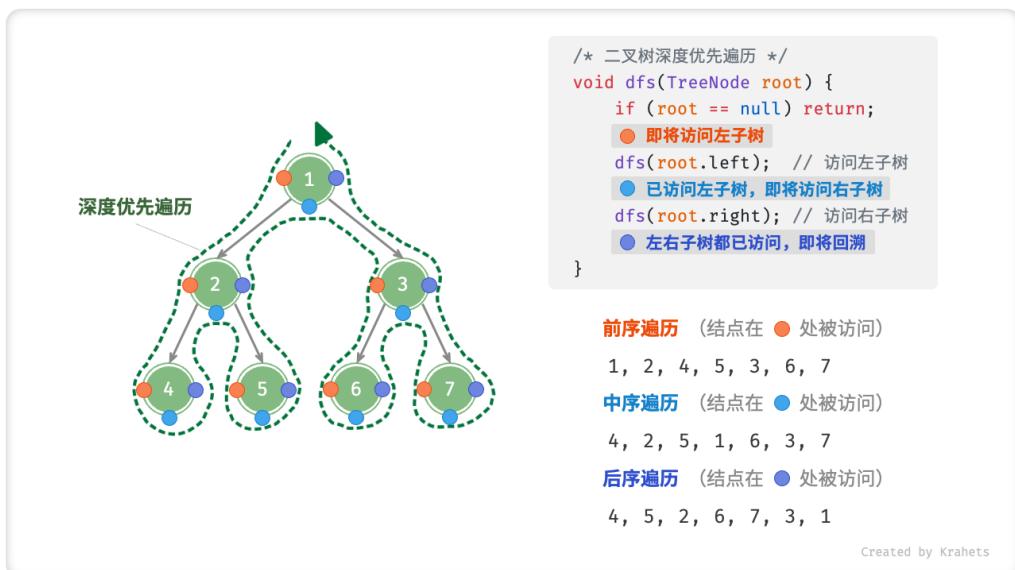


Figure 7-14. 二叉搜索树的前、中、后序遍历

位置	含义	此处访问结点时对应
橙色圆圈处	刚进入此结点，即将访问该结点的左子树	前序遍历 Pre-Order Traversal
蓝色圆圈处	已访问完左子树，即将访问右子树	中序遍历 In-Order Traversal
紫色圆圈处	已访问完左子树和右子树，即将返回	后序遍历 Post-Order Traversal

算法实现

```
// === File: binary_tree_dfs.cpp ===
/* 前序遍历 */
void preOrder(TreeNode* root) {
    if (root == nullptr) return;
    // 访问优先级：根结点 -> 左子树 -> 右子树
    vec.push_back(root->val);
    preOrder(root->left);
    preOrder(root->right);
}

/* 中序遍历 */
void inOrder(TreeNode* root) {
    if (root == nullptr) return;
    // 访问优先级：左子树 -> 根结点 -> 右子树
    inOrder(root->left);
    vec.push_back(root->val);
    inOrder(root->right);
}

/* 后序遍历 */
void postOrder(TreeNode* root) {
    if (root == nullptr) return;
    // 访问优先级：左子树 -> 右子树 -> 根结点
    postOrder(root->left);
    postOrder(root->right);
    vec.push_back(root->val);
}
```



使用循环一样可以实现前、中、后序遍历，但代码相对繁琐，有兴趣的同学可以自行实现。

复杂度分析

时间复杂度：所有结点被访问一次，使用 $O(n)$ 时间，其中 n 为结点数量。

空间复杂度：当树退化为链表时达到最差情况，递归深度达到 n ，系统使用 $O(n)$ 栈帧空间。

7.3. 二叉搜索树

「二叉搜索树 Binary Search Tree」满足以下条件：

1. 对于根结点，左子树中所有结点的值 $<$ 根结点的值 $<$ 右子树中所有结点的值；
2. 任意结点的左子树和右子树也是二叉搜索树，即也满足条件 1.；

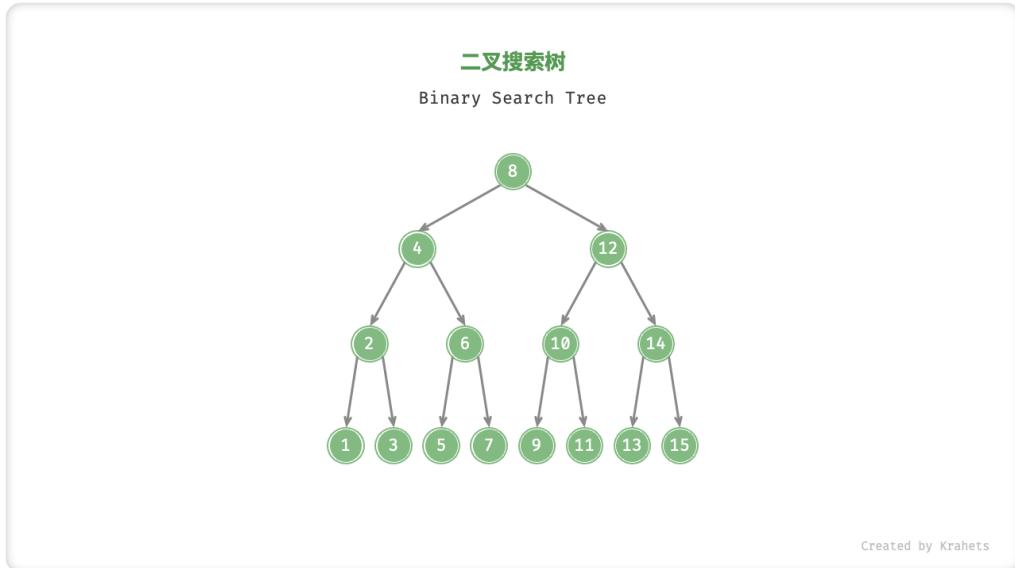


Figure 7-15. 二叉搜索树

7.3.1. 二叉搜索树的操作

查找结点

给定目标结点值 `num`，可以根据二叉搜索树的性质来查找。我们声明一个结点 `cur`，从二叉树的根结点 `root` 出发，循环比较结点值 `cur.val` 和 `num` 之间的大小关系

- 若 `cur.val < num`，说明目标结点在 `cur` 的右子树中，因此执行 `cur = cur.right`；
- 若 `cur.val > num`，说明目标结点在 `cur` 的左子树中，因此执行 `cur = cur.left`；
- 若 `cur.val = num`，说明找到目标结点，跳出循环并返回该结点即可；





Figure 7-16. 查找结点步骤

二叉搜索树的查找操作和二分查找算法如出一辙，也是在每轮排除一半情况。循环次数最多为二叉树的高度，当二叉树平衡时，使用 $O(\log n)$ 时间。

```
// === File: binary_search_tree.cpp ===
/* 查找结点 */
TreeNode* search(int num) {
    TreeNode* cur = root;
    // 循环查找，越过叶结点后跳出
    while (cur != nullptr) {
        // 目标结点在 cur 的右子树中
        if (cur->val < num) cur = cur->right;
        // 目标结点在 cur 的左子树中
        else if (cur->val > num) cur = cur->left;
        // 找到目标结点，跳出循环
        else break;
    }
    // 返回目标结点
    return cur;
}
```

插入结点

给定一个待插入元素 `num`，为了保持二叉搜索树“左子树 < 根结点 < 右子树”的性质，插入操作分为两步：

1. **查找插入位置：**与查找操作类似，我们从根结点出发，根据当前结点值和 `num` 的大小关系循环向下搜索，直到越过叶结点（遍历到 `null`）时跳出循环；
2. **在该位置插入结点：**初始化结点 `num`，将该结点放到 `null` 的位置；

二叉搜索树不允许存在重复结点，否则将会违背其定义。因此若待插入结点在树中已经存在，则不执行插入，直接返回即可。

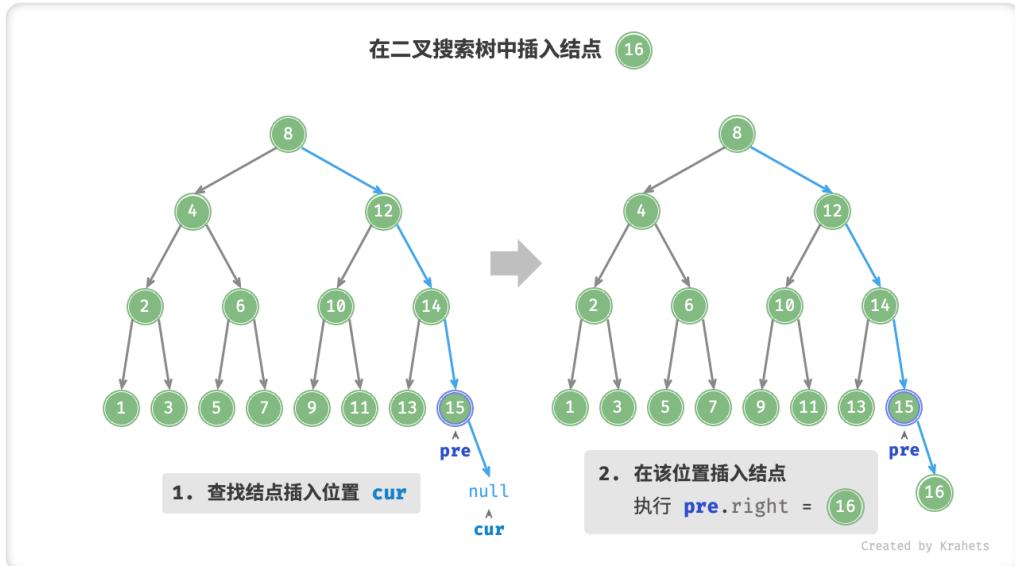


Figure 7-17. 在二叉搜索树中插入结点

```
// === File: binary_search_tree.cpp ===
/* 插入结点 */
TreeNode* insert(int num) {
    // 若树为空，直接提前返回
    if (root == nullptr) return nullptr;
    TreeNode *cur = root, *pre = nullptr;
    // 循环查找，越过叶结点后跳出
    while (cur != nullptr) {
        // 找到重复结点，直接返回
        if (cur->val == num) return nullptr;
        pre = cur;
        // 插入位置在 cur 的右子树中
        if (cur->val < num) cur = cur->right;
        // 插入位置在 cur 的左子树中
        else cur = cur->left;
    }
    // 插入结点 val
    TreeNode* node = new TreeNode(num);
    if (pre->val < num) pre->right = node;
    else pre->left = node;
    return node;
}
```

为了插入结点，需要借助 **辅助结点 pre** 保存上一轮循环的结点，这样在遍历到 null 时，我们也可以获取到其父结点，从而完成结点插入操作。

与查找结点相同，插入结点使用 $O(\log n)$ 时间。

删除结点

与插入结点一样，我们需要在删除操作后维持二叉搜索树的“左子树 < 根结点 < 右子树”的性质。首先，我们需要在二叉树中执行查找操作，获取待删除结点。接下来，根据待删除结点的子结点数量，删除操作需要分为三种情况：

当待删除结点的子结点数量 = 0 时，表明待删除结点是叶结点，直接删除即可。

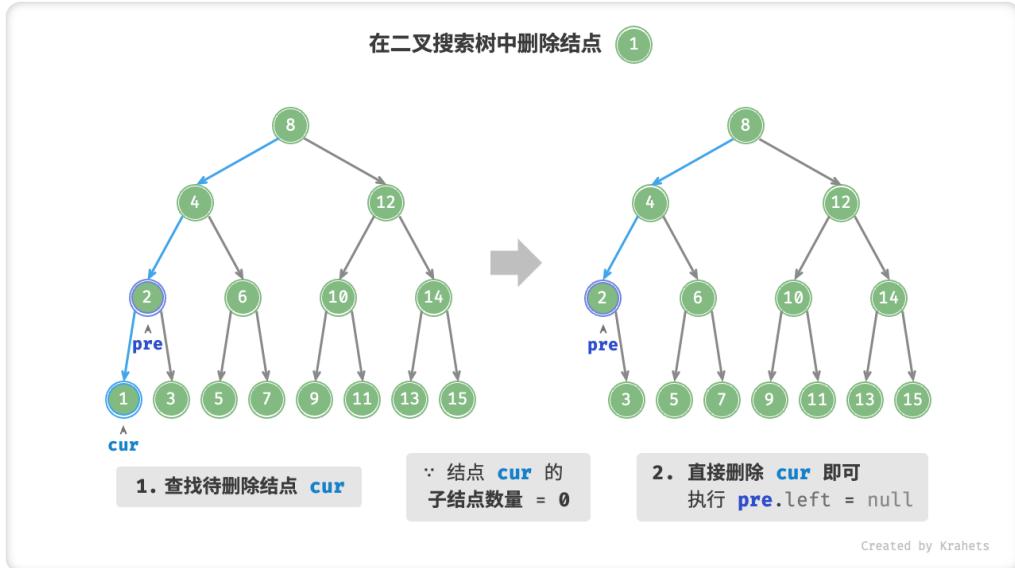


Figure 7-18. 在二叉搜索树中删除结点（度为 0）

当待删除结点的子结点数量 = 1 时，将待删除结点替换为其子结点即可。

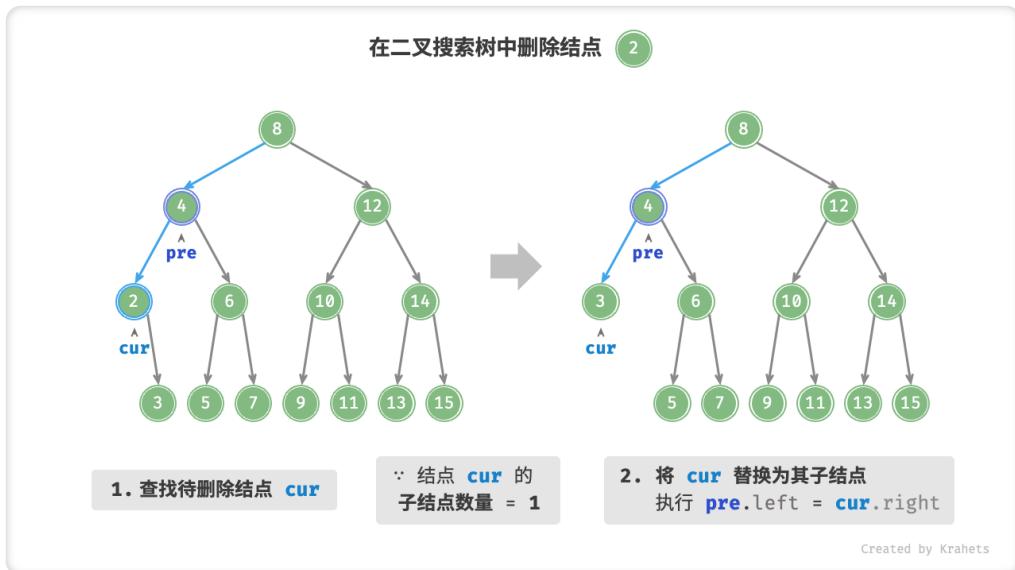


Figure 7-19. 在二叉搜索树中删除结点（度为 1）

当待删除结点的子结点数量 = 2 时，删除操作分为三步：

1. 找到待删除结点在 **中序遍历序列** 中的下一个结点，记为 `nex`；
2. 在树中递归删除结点 `nex`；
3. 使用 `nex` 替换待删除结点；

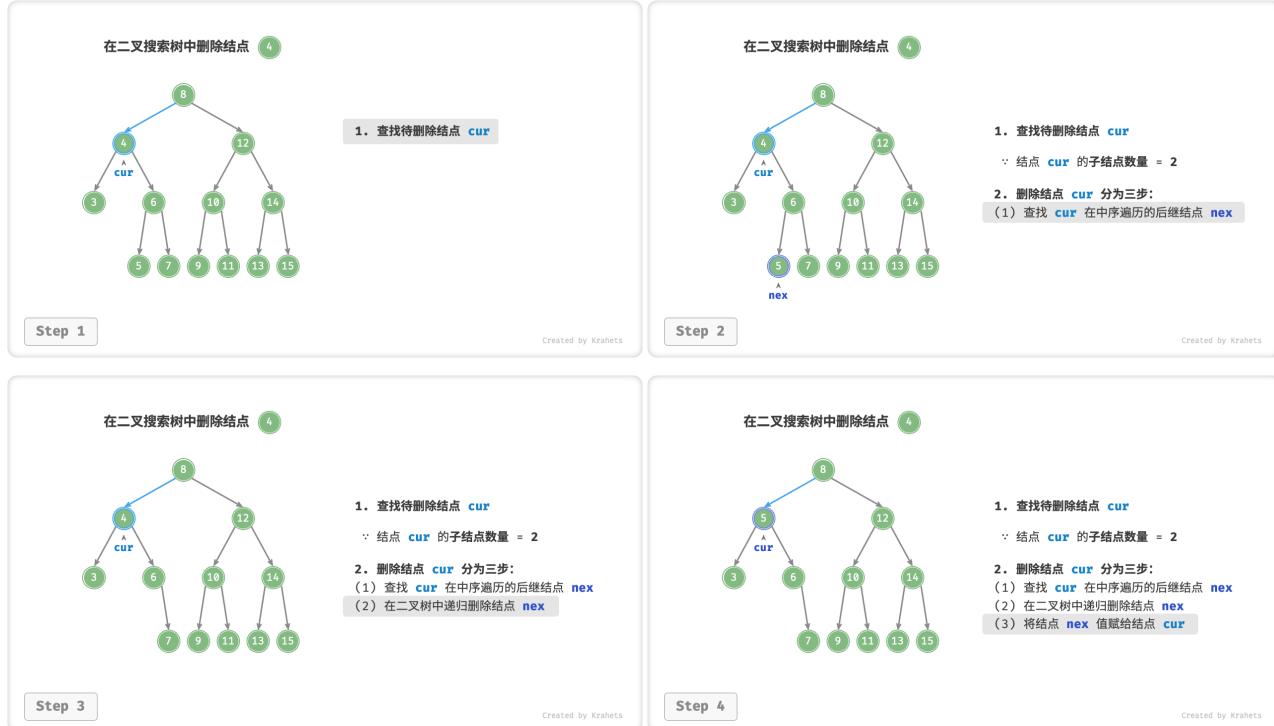


Figure 7-20. 删除结点（度为 2）步骤

删除结点操作也使用 $O(\log n)$ 时间，其中查找待删除结点 $O(\log n)$ ，获取中序遍历后继结点 $O(\log n)$ 。

```
// === File: binary_search_tree.cpp ===
/* 删除结点 */
TreeNode* remove(int num) {
    // 若树为空，直接提前返回
    if (root == nullptr) return nullptr;
    TreeNode *cur = root, *pre = nullptr;
    // 循环查找，越过叶结点后跳出
    while (cur != nullptr) {
        // 找到待删除结点，跳出循环
        if (cur->val == num) break;
        pre = cur;
        // 待删除结点在 cur 的右子树中
        if (cur->val < num) cur = cur->right;
        // 待删除结点在 cur 的左子树中
        else cur = cur->left;
    }
    // 若无待删除结点，则直接返回
}
```

```
if (cur == nullptr) return nullptr;
// 子结点数量 = 0 or 1
if (cur->left == nullptr || cur->right == nullptr) {
    // 当子结点数量 = 0 / 1 时, child = nullptr / 该子结点
    TreeNode* child = cur->left != nullptr ? cur->left : cur->right;
    // 删除结点 cur
    if (pre->left == cur) pre->left = child;
    else pre->right = child;
    // 释放内存
    delete cur;
}
// 子结点数量 = 2
else {
    // 获取中序遍历中 cur 的下一个结点
    TreeNode* nex = getInOrderNext(cur->right);
    int tmp = nex->val;
    // 递归删除结点 nex
    remove(nex->val);
    // 将 nex 的值复制给 cur
    cur->val = tmp;
}
return cur;
}

/* 获取中序遍历中的下一个结点（仅适用于 root 有左子结点的情况） */
TreeNode* getInOrderNext(TreeNode* root) {
    if (root == nullptr) return root;
    // 循环访问左子结点，直到叶结点时为最小结点，跳出
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}
```

排序

我们知道，「中序遍历」遵循“左 → 根 → 右”的遍历优先级，而二叉搜索树遵循“左子结点 < 根结点 < 右子结点”的大小关系。因此，在二叉搜索树中进行中序遍历时，总是会优先遍历下一个最小结点，从而得出一条重要性质：**二叉搜索树的中序遍历序列是升序的。**

借助中序遍历升序的性质，我们在二叉搜索树中获取有序数据仅需 $O(n)$ 时间，而无需额外排序，非常高效。

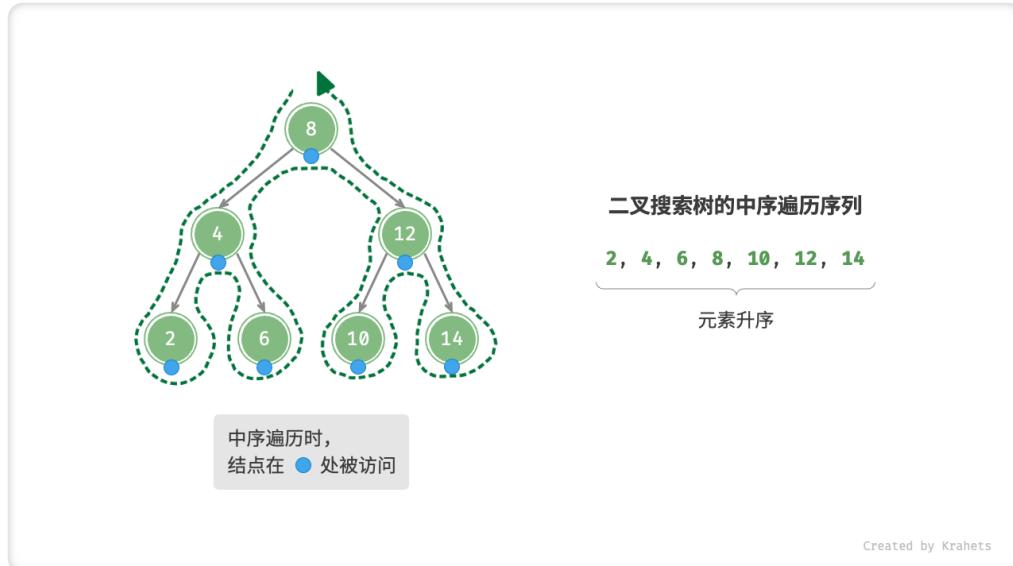


Figure 7-21. 二叉搜索树的中序遍历序列

7.3.2. 二叉搜索树的效率

假设给定 n 个数字，最常用的存储方式是「数组」，那么对于这串乱序的数字，常见操作的效率为：

- **查找元素**：由于数组是无序的，因此需要遍历数组来确定，使用 $O(n)$ 时间；
- **插入元素**：只需将元素添加至数组尾部即可，使用 $O(1)$ 时间；
- **删除元素**：先查找元素，使用 $O(n)$ 时间，再在数组中删除该元素，使用 $O(n)$ 时间；
- **获取最小 / 最大元素**：需要遍历数组来确定，使用 $O(n)$ 时间；

为了得到先验信息，我们也可以预先将数组元素进行排序，得到一个「排序数组」，此时操作效率为：

- **查找元素**：由于数组已排序，可以使用二分查找，平均使用 $O(\log n)$ 时间；
- **插入元素**：先查找插入位置，使用 $O(\log n)$ 时间，再插入到指定位置，使用 $O(n)$ 时间；
- **删除元素**：先查找元素，使用 $O(\log n)$ 时间，再在数组中删除该元素，使用 $O(n)$ 时间；
- **获取最小 / 最大元素**：数组头部和尾部元素即是最大和最小元素，使用 $O(1)$ 时间；

观察发现，无序数组和有序数组中的各项操作的时间复杂度是“偏科”的，即有的快有的慢；而二叉搜索树的各项操作的时间复杂度都是对数阶，在数据量 n 很大时有巨大优势。

	无序数组	有序数组	二叉搜索树
查找指定元素	$O(n)$	$O(\log n)$	$O(\log n)$
插入元素	$O(1)$	$O(n)$	$O(\log n)$
删除元素	$O(n)$	$O(n)$	$O(\log n)$
获取最小 / 最大元素	$O(n)$	$O(1)$	$O(\log n)$

7.3.3. 二叉搜索树的退化

理想情况下，我们希望二叉搜索树的是“左右平衡”的（详见「平衡二叉树」章节），此时可以在 $\log n$ 轮循环内查找任意结点。

如果我们动态地在二叉搜索树中插入与删除结点，则可能导致二叉树退化为链表，此时各种操作的时间复杂度也退化之 $O(n)$ 。



在实际应用中，如何保持二叉搜索树的平衡，也是一个需要考虑的问题。

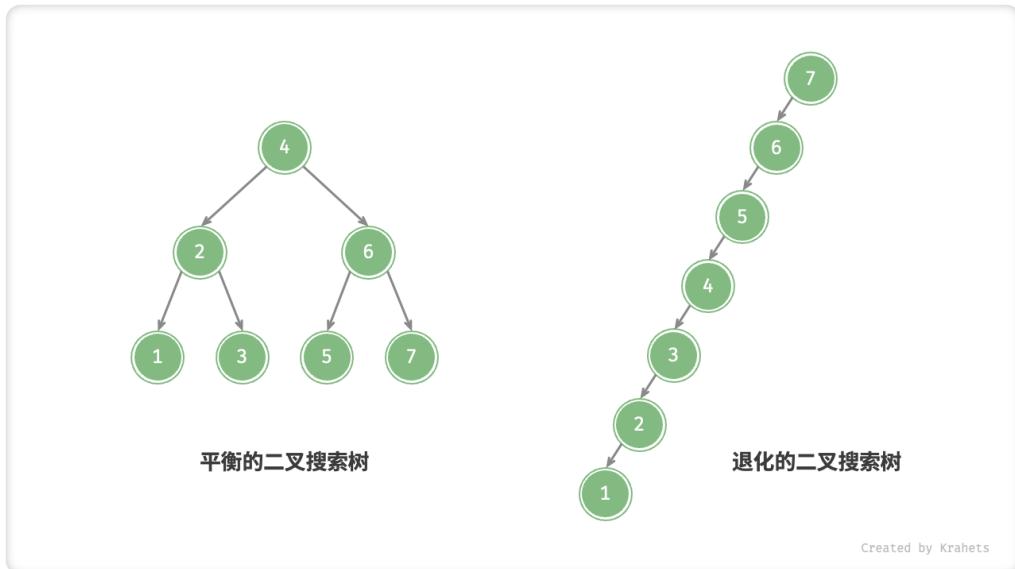


Figure 7-22. 二叉搜索树的平衡与退化

7.3.4. 二叉搜索树常见应用

- 系统中的多级索引，高效查找、插入、删除操作。
- 各种搜索算法的底层数据结构。
- 存储数据流，保持其已排序。

7.4. AVL 树 *

在「二叉搜索树」章节中提到，在进行多次插入与删除操作后，二叉搜索树可能会退化为链表。此时所有操作的时间复杂度都会由 $O(\log n)$ 劣化至 $O(n)$ 。

如下图所示，执行两步删除结点后，该二叉搜索树就会退化为链表。

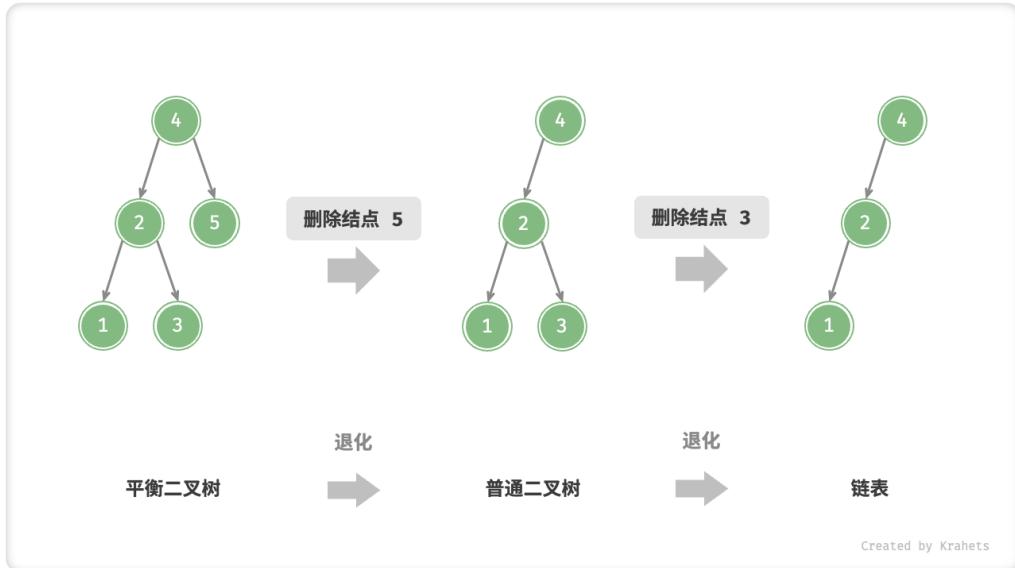


Figure 7-23. AVL 树在删除结点后发生退化

再比如，在以下完美二叉树中插入两个结点后，树严重向左偏斜，查找操作的时间复杂度也随之发生劣化。

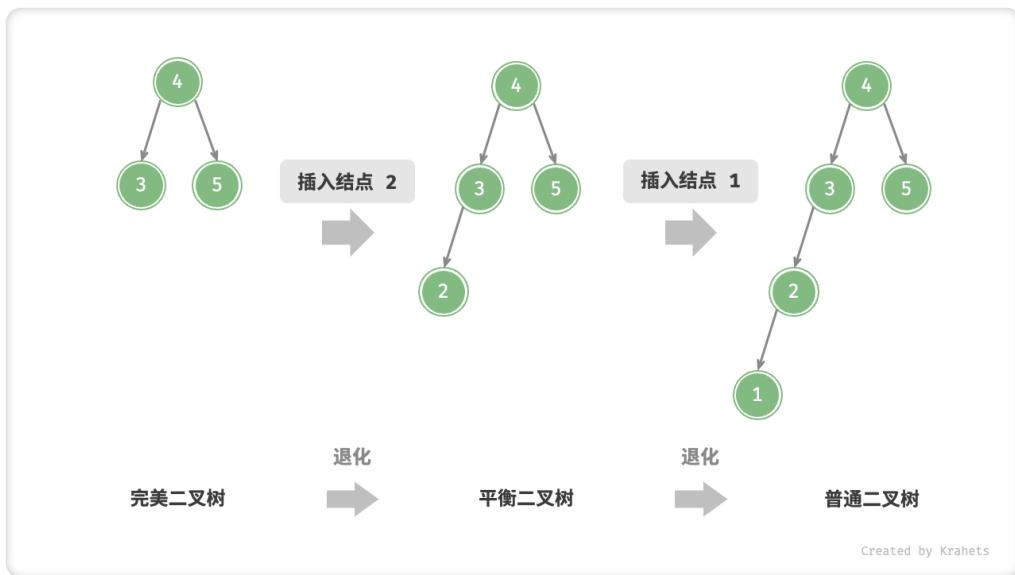


Figure 7-24. AVL 树在插入结点后发生退化

G. M. Adelson-Velsky 和 E. M. Landis 在其 1962 年发表的论文 “An algorithm for the organization of information” 中提出了「AVL 树」。论文中描述了一系列操作，使得在不断添加与删除结点后，AVL 树仍然不会发生退化，进而使得各种操作的时间复杂度均能保持在 $O(\log n)$ 级别。

换言之，在频繁增删查改的使用场景中，AVL 树可始终保持很高的数据增删查改效率，具有很好的应用价值。

7.4.1. AVL 树常见术语

「AVL 树」既是「二叉搜索树」又是「平衡二叉树」，同时满足这两种二叉树的所有性质，因此又被称为「平衡二叉搜索树」。

结点高度

在 AVL 树的操作中，需要获取结点「高度 Height」，所以给 AVL 树的结点类添加 `height` 变量。

```
/* AVL 树结点类 */
struct TreeNode {
    int val{};           // 结点值
    int height = 0;      // 结点高度
    TreeNode *left{};    // 左子结点
    TreeNode *right{};   // 右子结点
    TreeNode() = default;
    explicit TreeNode(int x) : val(x){}
};
```

「结点高度」是最远叶结点到该结点的距离，即走过的「边」的数量。需要特别注意，叶结点的高度为 0，空结点的高度为 -1。我们封装两个工具函数，分别用于获取与更新结点的高度。

```
// === File: avl_tree.cpp ===
/* 获取结点高度 */
int height(TreeNode* node) {
    // 空结点高度为 -1，叶结点高度为 0
    return node == nullptr ? -1 : node->height;
}

/* 更新结点高度 */
void updateHeight(TreeNode* node) {
    // 结点高度等于最高子树高度 + 1
    node->height = max(height(node->left), height(node->right)) + 1;
}
```

结点平衡因子

结点的「平衡因子 Balance Factor」是 **结点的左子树高度减去右子树高度**，并定义空结点的平衡因子为 0。同样地，我们将获取结点平衡因子封装成函数，以便后续使用。

```
// === File: avl_tree.cpp ===
/* 获取平衡因子 */
int balanceFactor(TreeNode* node) {
```

```
// 空结点平衡因子为 0
if (node == nullptr) return 0;
// 结点平衡因子 = 左子树高度 - 右子树高度
return height(node->left) - height(node->right);
}
```



设平衡因子为 f ，则一棵 AVL 树的任意结点的平衡因子皆满足 $-1 \leq f \leq 1$ 。

7.4.2. AVL 树旋转

AVL 树的独特之处在于「旋转 Rotation」的操作，其可在不影响二叉树中序遍历序列的前提下，使失衡结点重新恢复平衡。换言之，旋转操作既可以使树保持为「二叉搜索树」，也可以使树重新恢复为「平衡二叉树」。

我们将平衡因子的绝对值 > 1 的结点称为「失衡结点」。根据结点的失衡情况，旋转操作分为右旋、左旋、先右旋后左旋、先左旋后右旋，接下来我们一起来看看它们是如何操作的。

Case 1 - 右旋

如下图所示（结点下方为「平衡因子」），从底至顶看，二叉树中首个失衡结点是 **结点 3**。我们聚焦在以该失衡结点为根结点的子树上，将该结点记为 `node`，将其左子结点记为 `child`，执行「右旋」操作。完成右旋后，该子树已经恢复平衡，并且仍然为二叉搜索树。

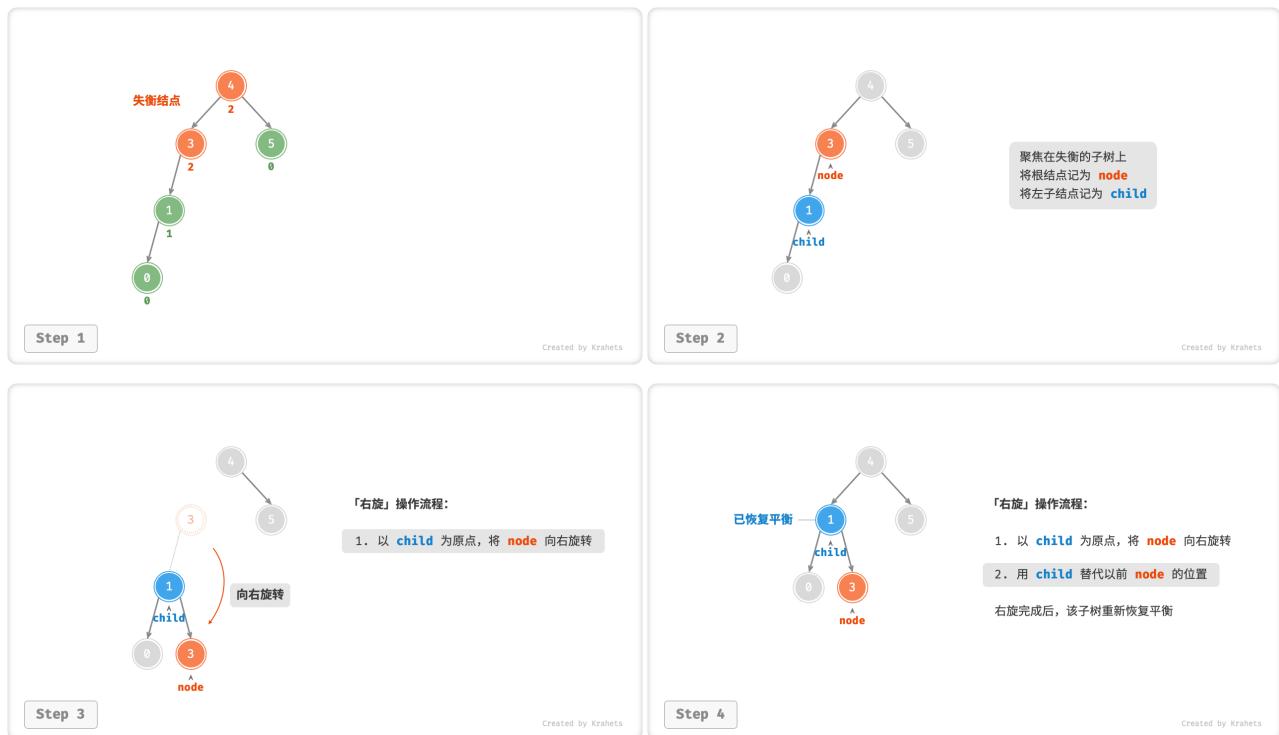


Figure 7-25. 右旋操作步骤

进而，如果结点 `child` 本身有右子结点（记为 `grandChild`），则需要在「右旋」中添加一步：将 `grandChild` 作为 `node` 的左子结点。

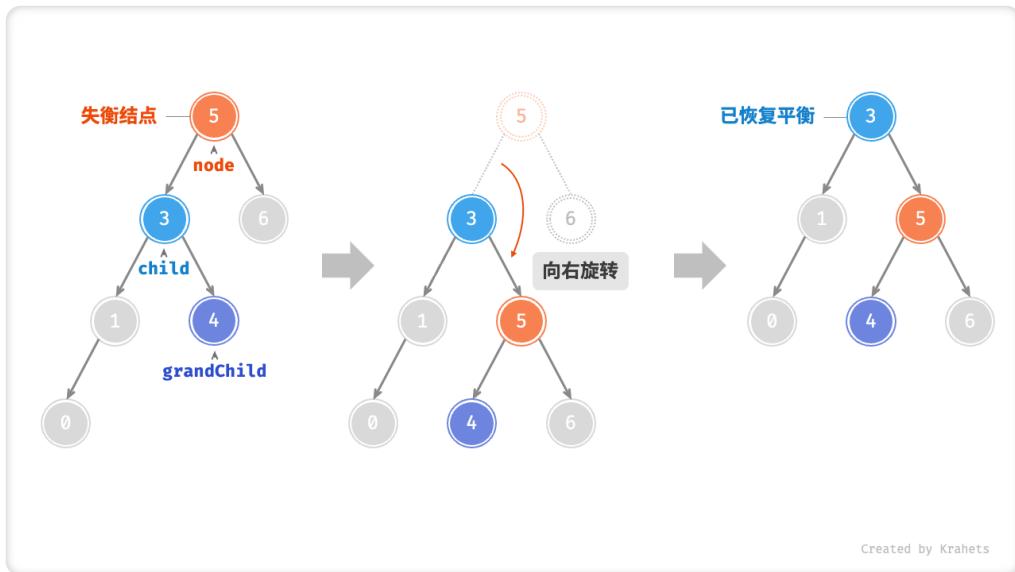


Figure 7-26. 有 grandChild 的右旋操作

“向右旋转” 是一种形象化的说法，实际需要通过修改结点指针实现，代码如下所示。

```
// === File: avl_tree.cpp ===
/* 右旋操作 */
TreeNode* rightRotate(TreeNode* node) {
    TreeNode* child = node->left;
    TreeNode* grandChild = child->right;
    // 以 child 为原点，将 node 向右旋转
    child->right = node;
    node->left = grandChild;
    // 更新结点高度
    updateHeight(node);
    updateHeight(child);
    // 返回旋转后子树的根结点
    return child;
}
```

Case 2 - 左旋

类似地，如果将取上述失衡二叉树的“镜像”，那么则需要「左旋」操作。

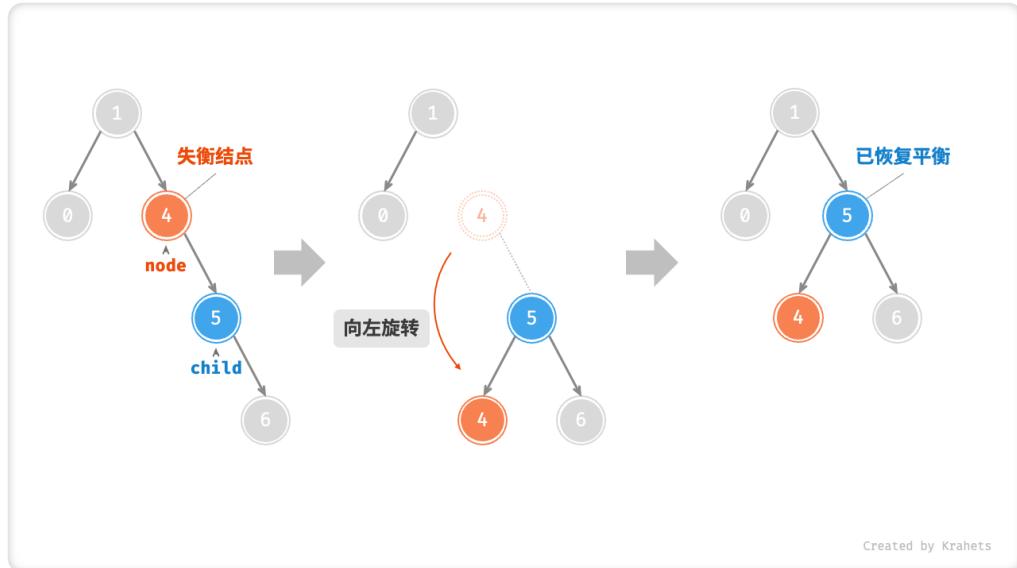
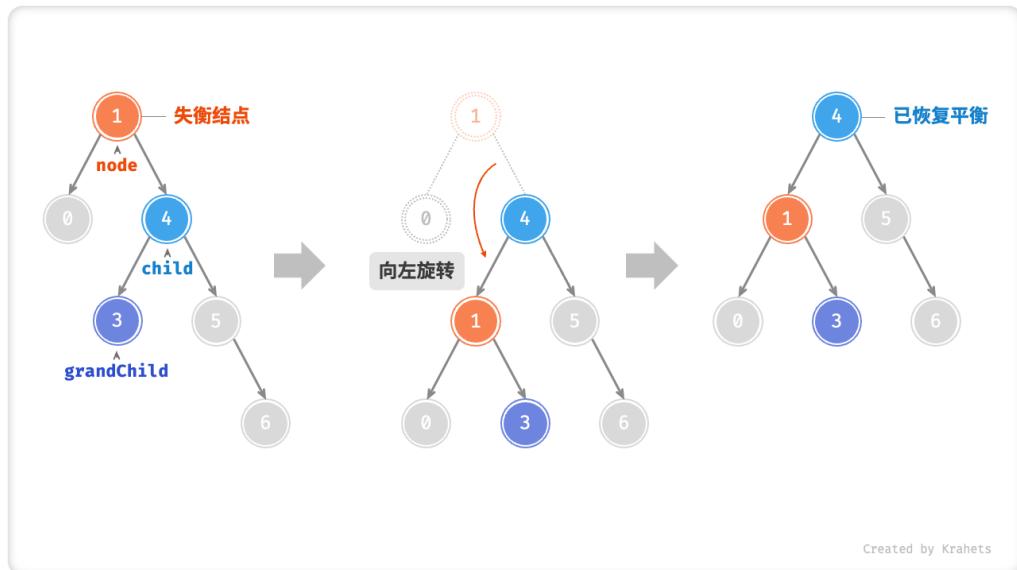


Figure 7-27. 左旋操作

同理，若结点 `child` 本身有左子结点（记为 `grandChild`），则需要在「左旋」中添加一步：将 `grandChild` 作为 `node` 的右子结点。

Figure 7-28. 有 `grandChild` 的左旋操作

观察发现，「左旋」和「右旋」操作是镜像对称的，两者对应解决的两种失衡情况也是对称的。根据对称性，我们可以很方便地从「右旋」推导出「左旋」。具体地，只需将「右旋」代码中的把所有的 `left` 替换为 `right`、所有的 `right` 替换为 `left`，即可得到「左旋」代码。

```
// === File: avl_tree.cpp ===
/* 左旋操作 */
TreeNode* leftRotate(TreeNode* node) {
    TreeNode* child = node->right;
    TreeNode* grandChild = child->left;
    // 以 child 为原点, 将 node 向左旋转
    child->left = node;
    node->right = grandChild;
    // 更新结点高度
    updateHeight(node);
    updateHeight(child);
    // 返回旋转后子树的根结点
    return child;
}
```

Case 3 - 先左后右

对于下图的失衡结点 3，单一使用左旋或右旋都无法使子树恢复平衡，此时需要「先左旋后右旋」，即先对 `child` 执行「左旋」，再对 `node` 执行「右旋」。

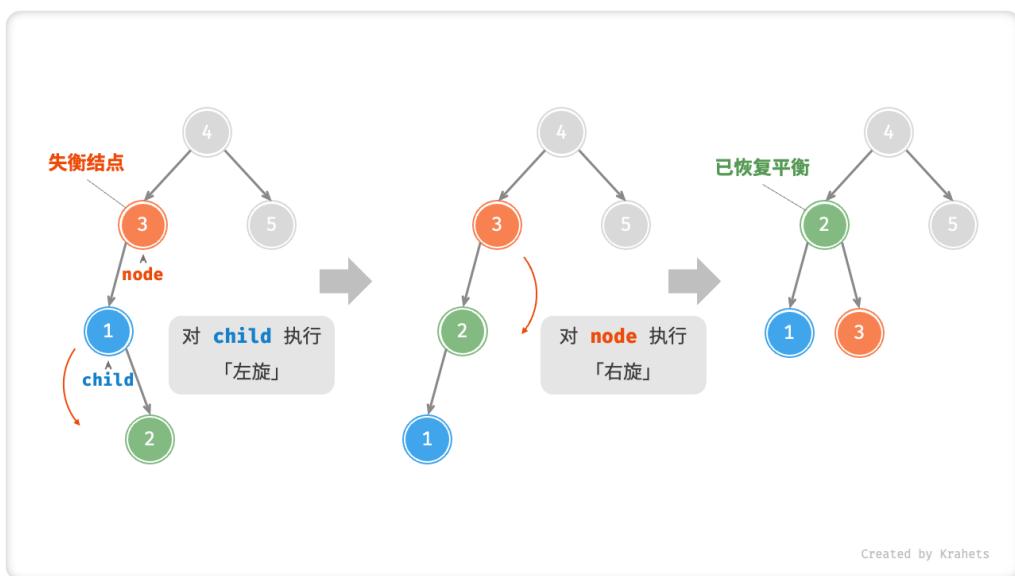


Figure 7-29. 先左旋后右旋

Case 4 - 先右后左

同理，取以上失衡二叉树的镜像，则需要「先右旋后左旋」，即先对 `child` 执行「右旋」，然后对 `node` 执行「左旋」。

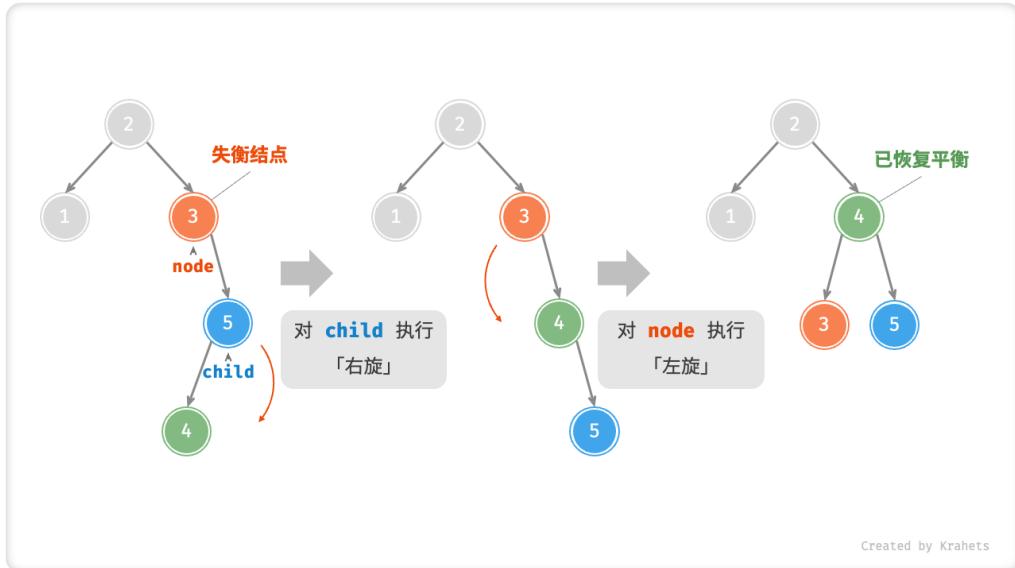


Figure 7-30. 先右旋后左旋

旋转的选择

下图描述的四种失衡情况与上述 Cases 逐个对应，分别需采用 右旋、左旋、先右后左、先左后右 的旋转操作。

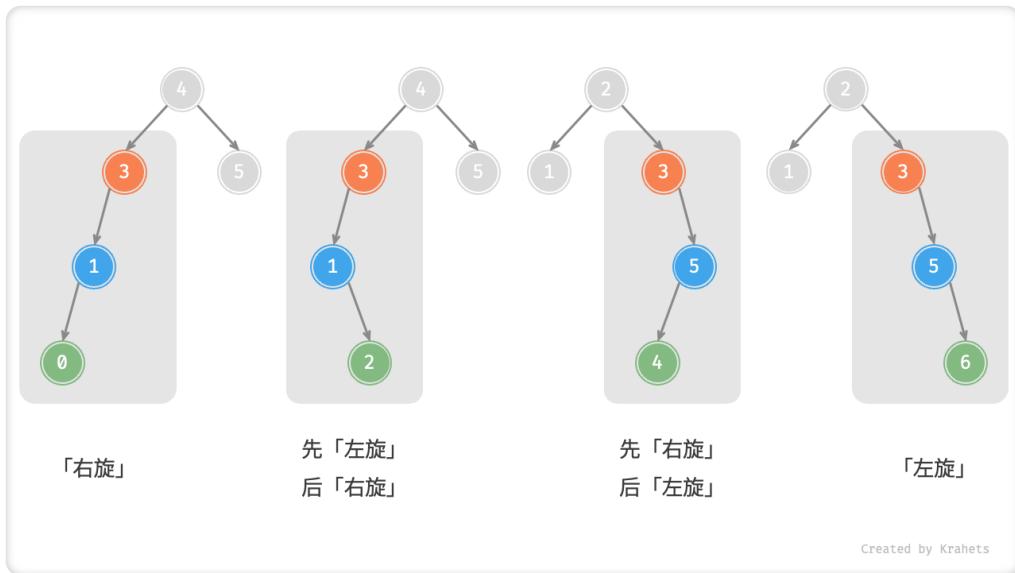


Figure 7-31. AVL 树的四种旋转情况

具体地，在代码中使用 失衡结点的平衡因子、较高一侧子结点的平衡因子 来确定失衡结点属于上图中的哪种情况。

失衡结点的平衡因子	子结点的平衡因子	应采用的旋转方法
> 0 (即左偏树)	≥ 0	右旋
> 0 (即左偏树)	< 0	先左旋后右旋
< 0 (即右偏树)	≤ 0	左旋
< 0 (即右偏树)	> 0	先右旋后左旋

为方便使用，我们将旋转操作封装成一个函数。至此，我们可以使用此函数来旋转各种失衡情况，使失衡结点重新恢复平衡。

```
// === File: avl_tree.cpp ===
/* 执行旋转操作，使该子树重新恢复平衡 */
TreeNode* rotate(TreeNode* node) {
    // 获取结点 node 的平衡因子
    int _balanceFactor = balanceFactor(node);
    // 左偏树
    if (_balanceFactor > 1) {
        if (balanceFactor(node->left) >= 0) {
            // 右旋
            return rightRotate(node);
        } else {
            // 先左旋后右旋
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }
    }
    // 右偏树
    if (_balanceFactor < -1) {
        if (balanceFactor(node->right) <= 0) {
            // 左旋
            return leftRotate(node);
        } else {
            // 先右旋后左旋
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }
    }
    // 平衡树，无需旋转，直接返回
    return node;
}
```

7.4.3. AVL 树常用操作

插入结点

「AVL 树」的结点插入操作与「二叉搜索树」主体类似。不同的是，在插入结点后，从该结点到根结点的路径上会出现一系列「失衡结点」。所以，**我们需要从该结点开始，从底至顶地执行旋转操作，使所有失衡结点恢复平衡。**

```
// === File: avl_tree.cpp ===
/* 插入结点 */
TreeNode* insert(int val) {
    root = insertHelper(root, val);
    return root;
}

/* 递归插入结点（辅助方法） */
TreeNode* insertHelper(TreeNode* node, int val) {
    if (node == nullptr) return new TreeNode(val);
    /* 1. 查找插入位置，并插入结点 */
    if (val < node->val)
        node->left = insertHelper(node->left, val);
    else if (val > node->val)
        node->right = insertHelper(node->right, val);
    else
        return node; // 重复结点不插入，直接返回
    updateHeight(node); // 更新结点高度
    /* 2. 执行旋转操作，使该子树重新恢复平衡 */
    node = rotate(node);
    // 返回子树的根结点
    return node;
}
```

删除结点

「AVL 树」删除结点操作与「二叉搜索树」删除结点操作总体相同。类似地，**在删除结点后，也需要从底至顶地执行旋转操作，使所有失衡结点恢复平衡。**

```
// === File: avl_tree.cpp ===
/* 删除结点 */
TreeNode* remove(int val) {
    root = removeHelper(root, val);
    return root;
}

/* 递归删除结点（辅助方法） */

```

```
TreeNode* removeHelper(TreeNode* node, int val) {
    if (node == nullptr) return nullptr;
    /* 1. 查找结点，并删除之 */
    if (val < node->val)
        node->left = removeHelper(node->left, val);
    else if (val > node->val)
        node->right = removeHelper(node->right, val);
    else {
        if (node->left == nullptr || node->right == nullptr) {
            TreeNode* child = node->left != nullptr ? node->left : node->right;
            // 子结点数量 = 0，直接删除 node 并返回
            if (child == nullptr) {
                delete node;
                return nullptr;
            }
            // 子结点数量 = 1，直接删除 node
            else {
                delete node;
                node = child;
            }
        } else {
            // 子结点数量 = 2，则将中序遍历的下个结点删除，并用该结点替换当前结点
            TreeNode* temp = getInOrderNext(node->right);
            node->right = removeHelper(node->right, temp->val);
            node->val = temp->val;
        }
    }
    updateHeight(node); // 更新结点高度
    /* 2. 执行旋转操作，使孩子树重新恢复平衡 */
    node = rotate(node);
    // 返回子树的根结点
    return node;
}

/* 获取中序遍历中的下一个结点（仅适用于 root 有左子结点的情况） */
TreeNode* getInOrderNext(TreeNode* node) {
    if (node == nullptr) return node;
    // 循环访问左子结点，直到叶结点时为最小结点，跳出
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}
```

查找结点

「AVL 树」的结点查找操作与「二叉搜索树」一致，在此不再赘述。

7.4.4. AVL 树典型应用

- 组织存储大型数据，适用于高频查找、低频增删场景；
- 用于建立数据库中的索引系统；



为什么红黑树比 AVL 树更受欢迎？

红黑树的平衡条件相对宽松，因此在红黑树中插入与删除结点所需的旋转操作相对更少，结点增删操作相比 AVL 树的效率更高。

7.5. 小结

二叉树

- 二叉树是一种非线性数据结构，代表着“一分为二”的分治逻辑。二叉树的结点包含「值」和两个「指针」，分别指向左子结点和右子结点。
- 选定二叉树中某结点，将其左（右）子结点以下形成的树称为左（右）子树。
- 二叉树的术语较多，包括根结点、叶结点、层、度、边、高度、深度等。
- 二叉树的初始化、结点插入、结点删除操作与链表的操作方法类似。
- 常见的二叉树类型包括完美二叉树、完全二叉树、完满二叉树、平衡二叉树。完美二叉树是理想状态，链表则是退化后的最差状态。
- 二叉树可以使用数组表示，具体做法是将结点值和空位按照层序遍历的顺序排列，并基于父结点和子结点之间的索引映射公式实现指针。

二叉树遍历

- 二叉树层序遍历是一种广度优先搜索，体现着“一圈一圈向外”的层进式遍历方式，通常借助队列来实现。
- 前序、中序、后序遍历是深度优先搜索，体现着“走到头、再回头继续”的回溯遍历方式，通常使用递归实现。

二叉搜索树

- 二叉搜索树是一种高效的元素查找数据结构，查找、插入、删除操作的时间复杂度皆为 $O(\log n)$ 。二叉搜索树退化为链表后，各项时间复杂度劣化至 $O(n)$ ，因此如何避免退化是非常重要的课题。
- AVL 树又称平衡二叉搜索树，其通过旋转操作，使得在不断插入与删除结点后，仍然可以保持二叉树的平衡（不退化）。

- AVL 树的旋转操作分为右旋、左旋、先右旋后左旋、先左旋后右旋。在插入或删除结点后，AVL 树会从底至顶地执行旋转操作，使树恢复平衡。

8. 堆

8.1. 堆

「堆 Heap」是一棵限定条件下的「完全二叉树」。根据成立条件，堆主要分为两种类型：

- 「大顶堆 Max Heap」，任意结点的值 \geq 其子结点的值；
- 「小顶堆 Min Heap」，任意结点的值 \leq 其子结点的值；

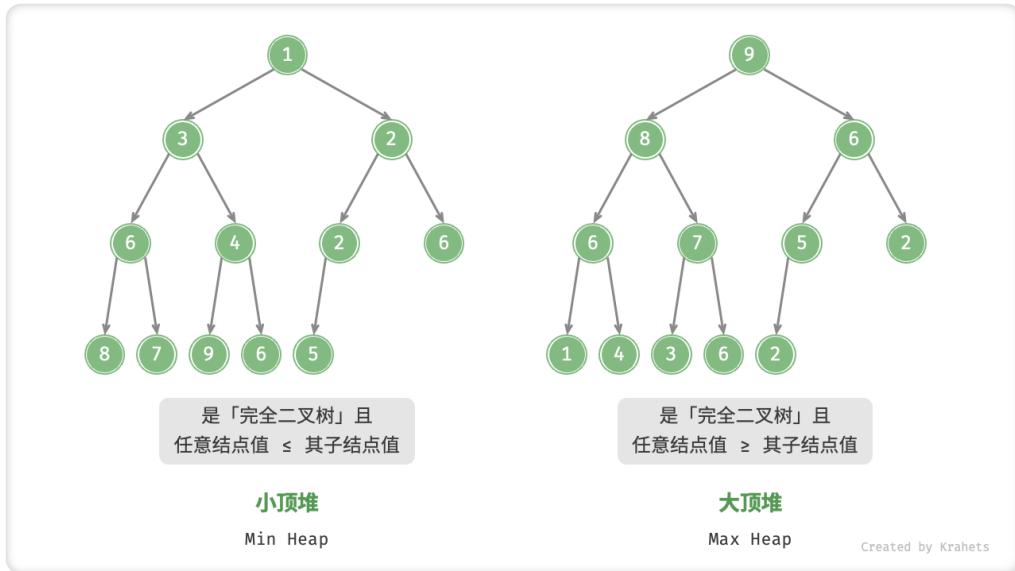


Figure 8-1. 小顶堆与大顶堆

8.1.1. 堆术语与性质

- 由于堆是完全二叉树，因此最底层结点靠左填充，其它层结点皆被填满。
- 二叉树中的根结点对应「堆顶」，底层最靠右结点对应「堆底」。
- 对于大顶堆 / 小顶堆，其堆顶元素（即根结点）的值最大 / 最小。

8.1.2. 堆常用操作

值得说明的是，多数编程语言提供的是「优先队列 Priority Queue」，其是一种抽象数据结构，**定义为具有出队优先级的队列**。

而恰好，**堆的定义与优先队列的操作逻辑完全吻合**，大顶堆就是一个元素从大到小出队的优先队列。从使用角度看，我们可以将「优先队列」和「堆」理解为等价的数据结构。因此，本文与代码对两者不做特别区分，统一使用「堆」来命名。

堆的常用操作见下表（方法命名以 Java 为例）。

方法	描述	时间复杂度
add()	元素入堆	$O(\log n)$
poll()	堆顶元素出堆	$O(\log n)$
peek()	访问堆顶元素（大 / 小顶堆分别为最大 / 小值）	$O(1)$
size()	获取堆的元素数量	$O(1)$
isEmpty()	判断堆是否为空	$O(1)$

我们可以直接使用编程语言提供的堆类（或优先队列类）。



类似于排序中“从小到大排列”和“从大到小排列”，“大顶堆”和“小顶堆”可仅通过修改 Comparator 来互相转换。

```
// === File: heap.cpp ===
/* 初始化堆 */
// 初始化小顶堆
priority_queue<int, vector<int>, greater<int>> minHeap;
// 初始化大顶堆
priority_queue<int, vector<int>, less<int>> maxHeap;

/* 元素入堆 */
maxHeap.push(1);
maxHeap.push(3);
maxHeap.push(2);
maxHeap.push(5);
maxHeap.push(4);

/* 获取堆顶元素 */
int peek = maxHeap.top(); // 5

/* 堆顶元素出堆 */
// 出堆元素会形成一个从大到小的序列
maxHeap.pop(); // 5
maxHeap.pop(); // 4
maxHeap.pop(); // 3
maxHeap.pop(); // 2
maxHeap.pop(); // 1

/* 获取堆大小 */
int size = maxHeap.size();
```

```

/* 判断堆是否为空 */
bool isEmpty = maxHeap.empty();

/* 输入列表并建堆 */
vector<int> input{1, 3, 2, 5, 4};
priority_queue<int, vector<int>, greater<int>> minHeap(input.begin(), input.end());

```

8.1.3. 堆的实现

下文实现的是「大顶堆」，若想转换为「小顶堆」，将所有大小逻辑判断取逆（例如将 \geq 替换为 \leq ）即可，有兴趣的同学可自行实现。

堆的存储与表示

在二叉树章节我们学过，「完全二叉树」非常适合使用「数组」来表示，而堆恰好是一棵完全二叉树，因而我们采用「数组」来存储「堆」。

二叉树指针。使用数组表示二叉树时，元素代表结点值，索引代表结点在二叉树中的位置，而结点指针通过索引映射公式来实现。

具体地，给定索引 i ，那么其左子结点索引为 $2i + 1$ 、右子结点索引为 $2i + 2$ 、父结点索引为 $(i - 1)/2$ （向下整除）。当索引越界时，代表空结点或结点不存在。

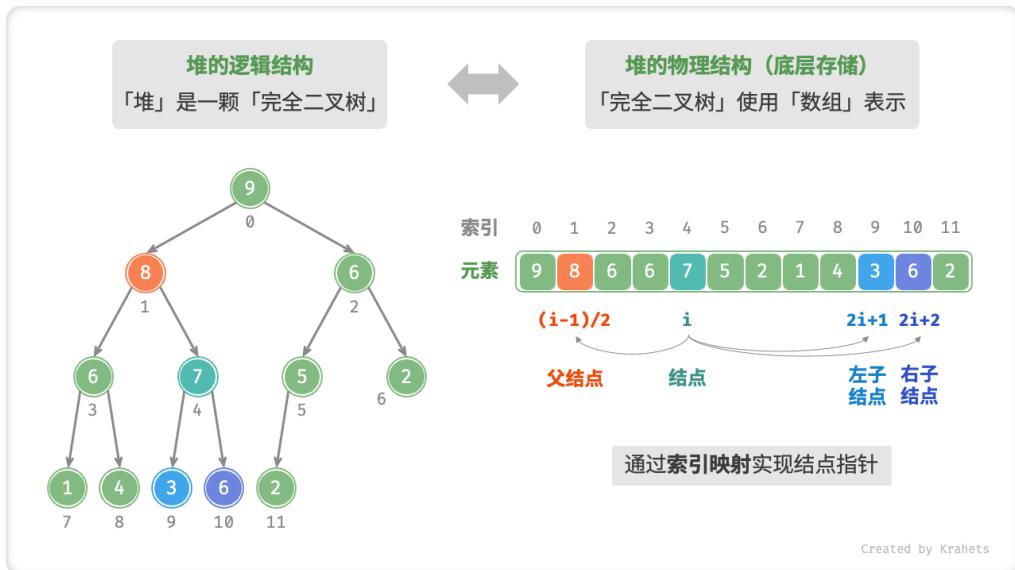


Figure 8-2. 堆的表示与存储

我们将索引映射公式封装成函数，以便后续使用。

```
// === File: my_heap.cpp ===
/* 获取左子结点索引 */
int left(int i) {
    return 2 * i + 1;
}

/* 获取右子结点索引 */
int right(int i) {
    return 2 * i + 2;
}

/* 获取父结点索引 */
int parent(int i) {
    return (i - 1) / 2; // 向下取整
}
```

访问堆顶元素

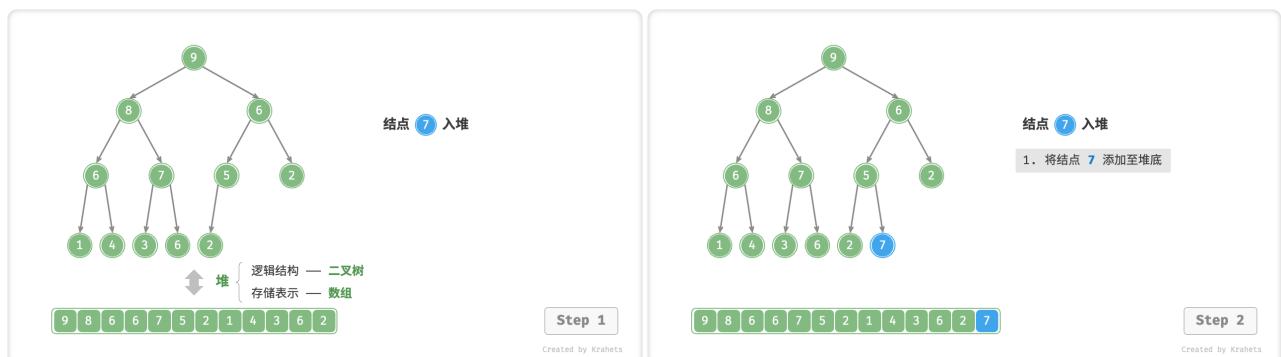
堆顶元素是二叉树的根结点，即列表首元素。

```
// === File: my_heap.cpp ===
/* 访问堆顶元素 */
int peek() {
    return maxHeap[0];
}
```

元素入堆

给定元素 `val`，我们先将其添加到堆底。添加后，由于 `val` 可能大于堆中其它元素，此时堆的成立条件可能已经被破坏，因此需要修复从插入结点到根结点这条路径上的各个结点，该操作被称为「堆化 Heapify」。

考虑从入堆结点开始，**从底至顶执行堆化**。具体地，比较插入结点与其父结点的值，若插入结点更大则将它们交换；并循环以上操作，从底至顶地修复堆中的各个结点；直至越过根结点时结束，或当遇到无需交换的结点时提前结束。



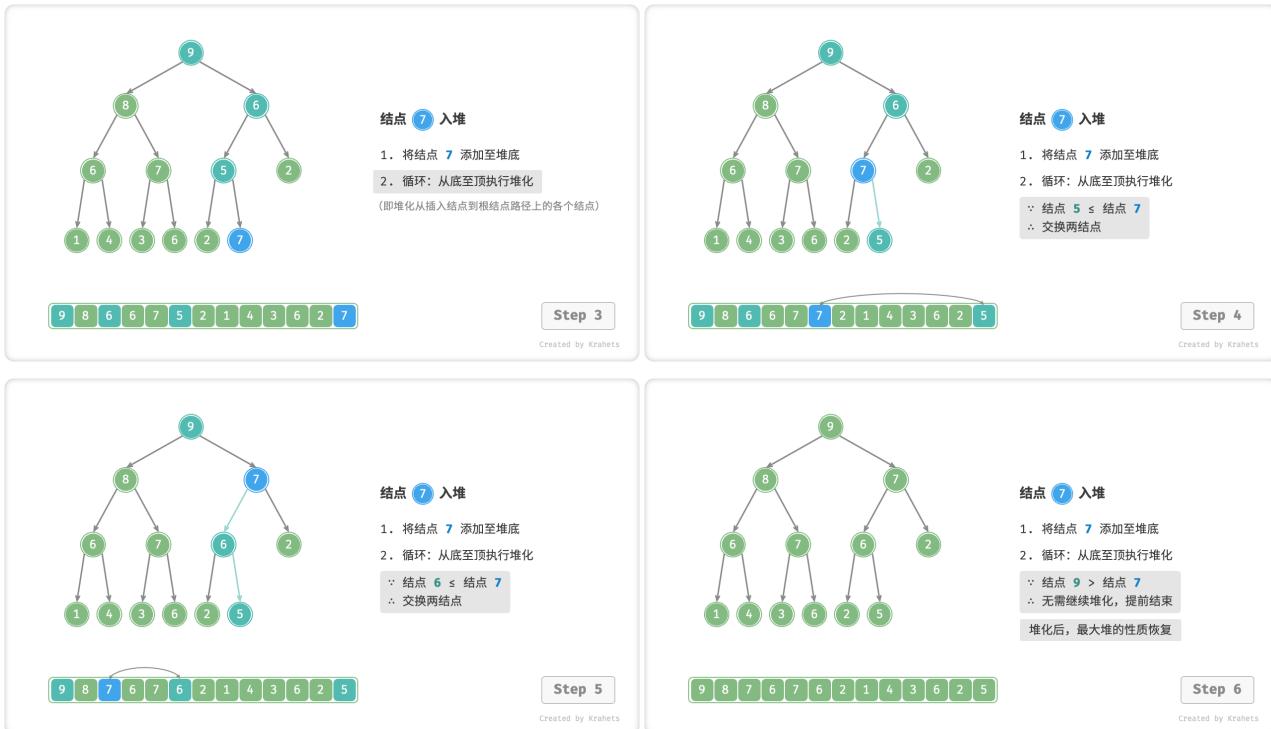


Figure 8-3. 元素入堆步骤

设结点总数为 n ，则树的高度为 $O(\log n)$ ，易得堆化操作的循环轮数最多为 $O(\log n)$ ，因而元素入堆操作的时间复杂度为 $O(\log n)$ 。

```
// === File: my_heap.cpp ===
/* 元素入堆 */
void push(int val) {
    // 添加结点
    maxHeap.push_back(val);
    // 从底至顶堆化
    siftUp(size() - 1);
}

/* 从结点 i 开始，从底至顶堆化 */
void siftUp(int i) {
    while (true) {
        // 获取结点 i 的父结点
        int p = parent(i);
        // 当“越过根结点”或“结点无需修复”时，结束堆化
        if (p < 0 || maxHeap[i] <= maxHeap[p])
            break;
        // 交换两结点
        swap(maxHeap[i], maxHeap[p]);
        // 循环向上堆化
        i = parent(i);
    }
}
```

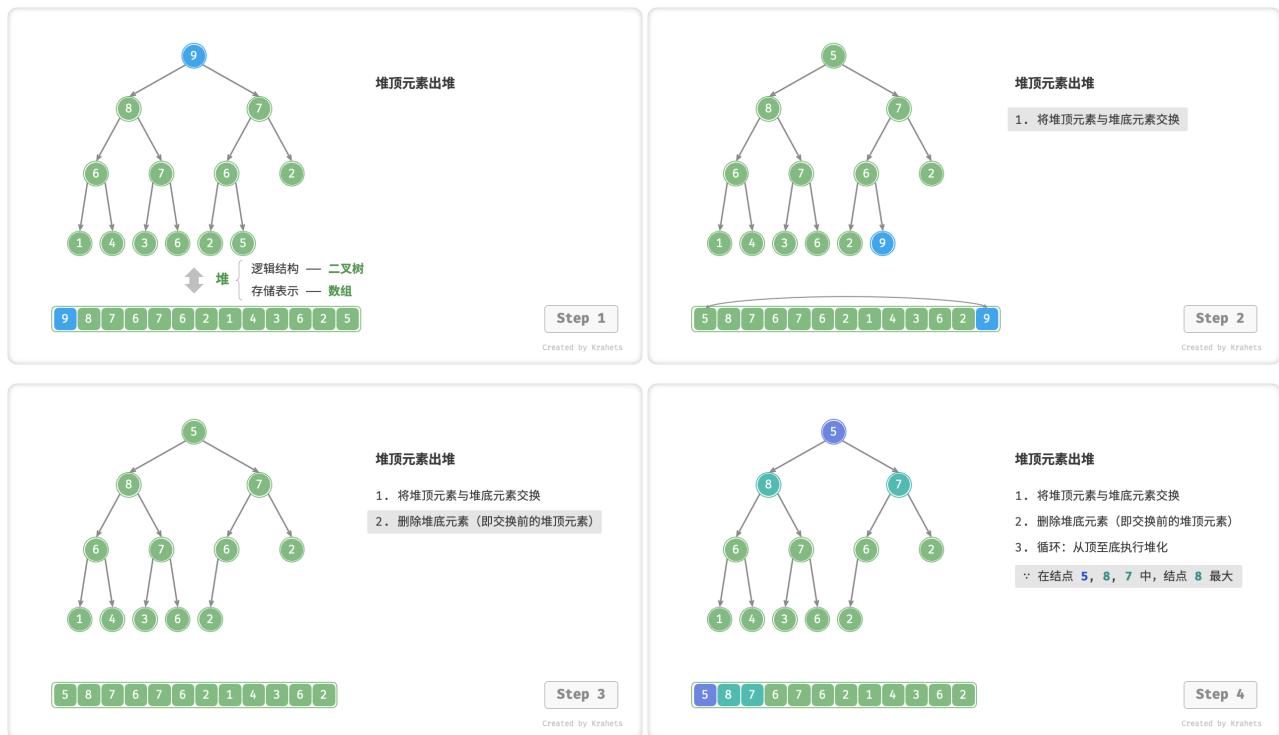
```
i = p;
}
}
```

堆顶元素出堆

堆顶元素是二叉树根结点，即列表首元素，如果我们直接将首元素从列表中删除，则二叉树中所有结点都会随之发生移位（索引发生变化），这样后续使用堆化修复就很麻烦了。为了尽量减少元素索引变动，采取以下操作步骤：

1. 交换堆顶元素与堆底元素（即交换根结点与最右叶结点）；
2. 交换完成后，将堆底从列表中删除（注意，因为已经交换，实际上删除的是原来的堆顶元素）；
3. 从根结点开始，**从顶至底执行堆化**；

顾名思义，**从顶至底堆化的操作方向与从底至顶堆化相反**，我们比较根结点的值与其两个子结点的值，将最大的子结点与根结点执行交换，并循环以上操作，直到越过叶结点时结束，或当遇到无需交换的结点时提前结束。



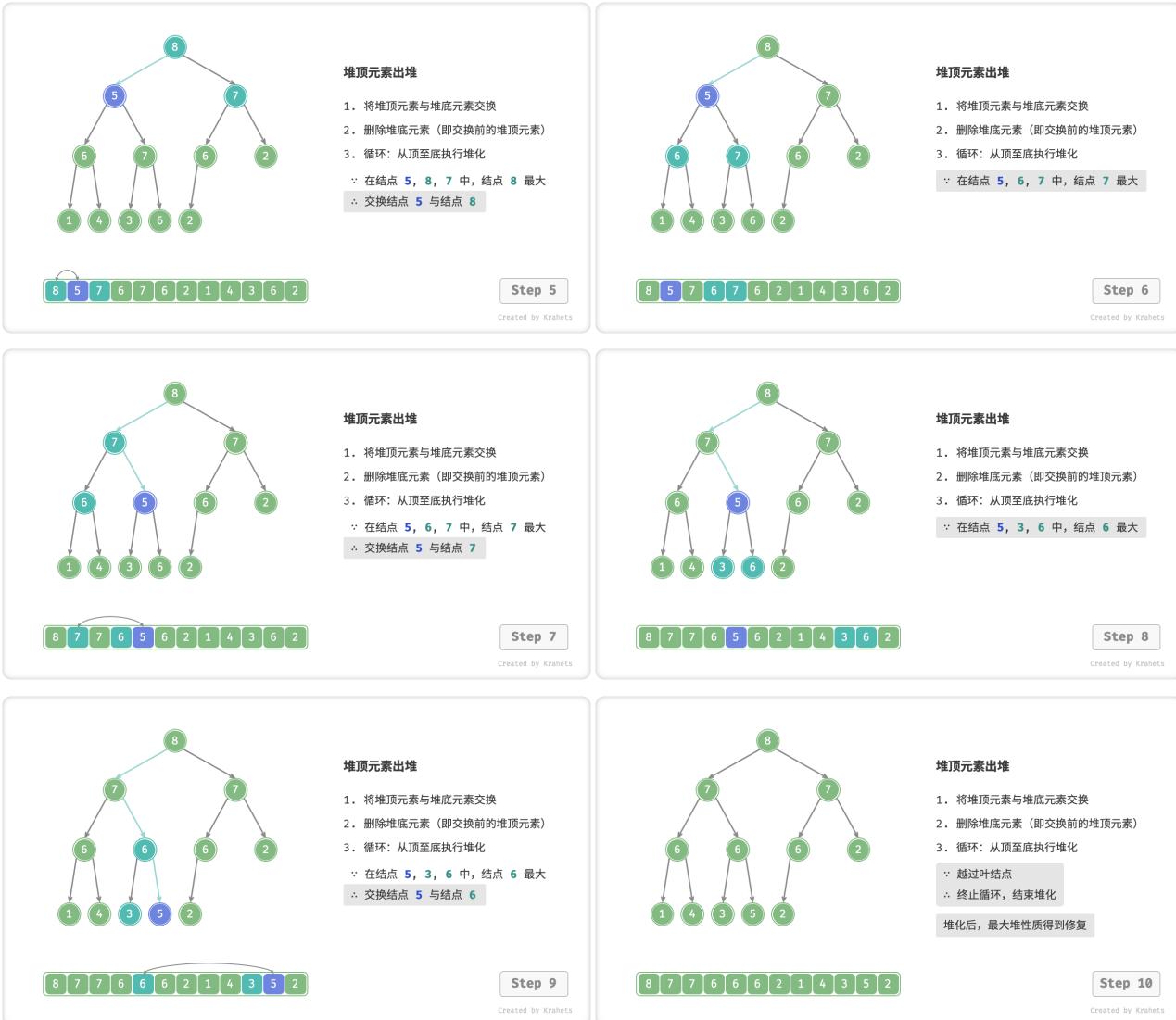


Figure 8-4. 堆顶元素出堆步骤

与元素入堆操作类似，堆顶元素出堆操作的时间复杂度为 $O(\log n)$ 。

```
// === File: my_heap.cpp ===
/* 元素出堆 */
void poll() {
    // 判空处理
    if (empty()) {
        throw out_of_range(" 堆为空");
    }
    // 交换根结点与最右叶结点（即交换首元素与尾元素）
    swap(maxHeap[0], maxHeap[size() - 1]);
    // 删除结点
    maxHeap.pop_back();
```

```
// 从顶到底堆化
siftDown(0);
}

/* 从结点 i 开始，从顶到底堆化 */
void siftDown(int i) {
    while (true) {
        // 判断结点 i, l, r 中值最大的结点，记为 ma
        int l = left(i), r = right(i), ma = i;
        // 若结点 i 最大或索引 l, r 越界，则无需继续堆化，跳出
        if (l < size() && maxHeap[l] > maxHeap[ma])
            ma = l;
        if (r < size() && maxHeap[r] > maxHeap[ma])
            ma = r;
        // 若结点 i 最大或索引 l, r 越界，则无需继续堆化，跳出
        if (ma == i)
            break;
        swap(maxHeap[i], maxHeap[ma]);
        // 循环向下堆化
        i = ma;
    }
}
```

8.1.4. 堆常见应用

- **优先队列。**堆常作为实现优先队列的首选数据结构，入队和出队操作时间复杂度为 $O(\log n)$ ，建队操作为 $O(n)$ ，皆非常高效。
- **堆排序。**给定一组数据，我们使用其建堆，并依次全部弹出，则可以得到有序的序列。当然，堆排序一般无需弹出元素，仅需每轮将堆顶元素交换至数组尾部并减小堆的长度即可。
- **获取最大的 k 个元素。**这既是一道经典算法题目，也是一种常见应用，例如选取热度前 10 的新闻作为微博热搜，选取前 10 销量的商品等。

8.2. 建堆操作 *

如果我们想要根据输入列表来生成一个堆，这样的操作被称为「建堆」。

8.2.1. 两种建堆方法

借助入堆方法实现

最直接地，考虑借助「元素入堆」方法，先建立一个空堆，再将列表元素依次入堆即可。

基于堆化操作实现

然而，存在一种更加高效的建堆方法。设元素数量为 n ，我们先将列表所有元素原封不动添加进堆，然后迭代地对各个结点执行「从顶至底堆化」。当然，无需对叶结点执行堆化，因为其没有子结点。

```
// === File: my_heap.cpp ===
/* 构造方法，根据输入列表建堆 */
MaxHeap(vector<int> nums) {
    // 将列表元素原封不动添加进堆
    maxHeap = nums;
    // 堆化除叶结点以外的其他所有结点
    for (int i = parent(size() - 1); i >= 0; i--) {
        siftDown(i);
    }
}
```

8.2.2. 复杂度分析

对于第一种建堆方法，元素入堆的时间复杂度为 $O(n \log n)$ ，而平均长度为 $\frac{n}{2}$ ，因此该方法的总体时间复杂度为 $O(n \log n)$ 。

那么，第二种建堆方法的时间复杂度是多少呢？我们来展开推算一下。

- 完全二叉树中，设结点总数为 n ，则叶结点数量为 $(n + 1)/2$ ，其中 / 为向下整除。因此在排除叶结点后，需要堆化结点数量为 $(n - 1)/2$ ，即为 $O(n)$ ；
- 从顶至底堆化中，每个结点最多堆化至叶结点，因此最大迭代次数为二叉树高度 $O(\log n)$ ；

将上述两者相乘，可得时间复杂度为 $O(n \log n)$ 。然而，该估算结果仍不够准确，因为我们没有考虑到二叉树底层结点远多于顶层结点的性质。

下面我们来尝试展开计算。为了减小计算难度，我们假设树是一个「完美二叉树」，该假设不会影响计算结果的正确性。设二叉树（即堆）结点数量为 n ，树高度为 h 。上文提到，**结点堆化最大迭代次数等于该结点到叶结点的距离，而这正是“结点高度”**。因此，我们将各层的“结点数量 \times 结点高度”求和，即可得到所有结点的堆化的迭代次数总和。

$$T(h) = 2^0 h + 2^1(h - 1) + 2^2(h - 2) + \dots + 2^{(h-1)} \times 1$$

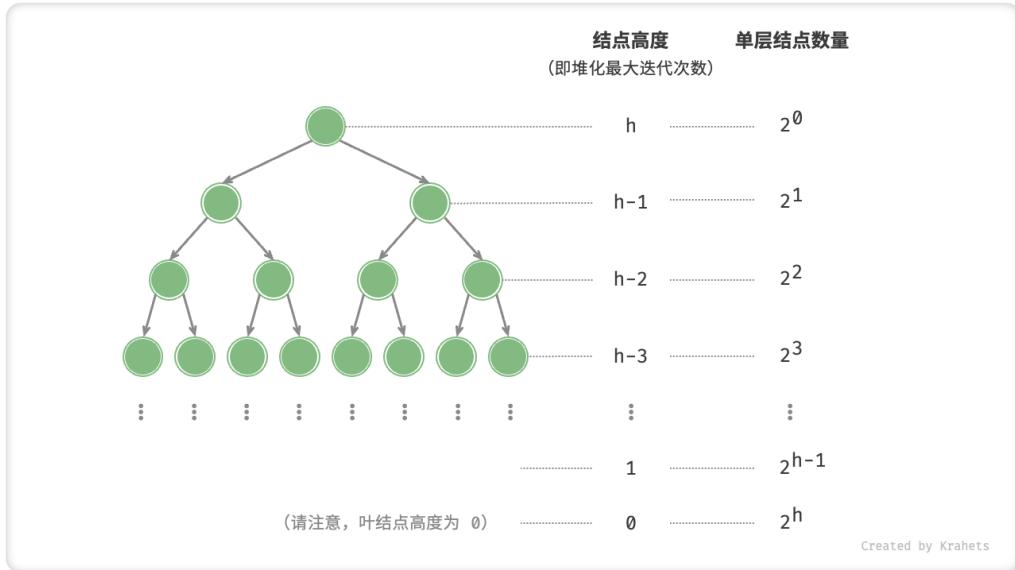


Figure 8-5. 完美二叉树的各层结点数量

化简上式需要借助中学的数列知识，先对 $T(h)$ 乘以 2，易得

$$\begin{aligned} T(h) &= 2^0h + 2^1(h - 1) + 2^2(h - 2) + \cdots + 2^{h-1} \times 1 \\ 2T(h) &= 2^1h + 2^2(h - 1) + 2^3(h - 2) + \cdots + 2^h \times 1 \end{aligned}$$

使用错位相减法，令下式 $2T(h)$ 减去上式 $T(h)$ ，可得

$$2T(h) - T(h) = T(h) = -2^0h + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h$$

观察上式， $T(h)$ 是一个等比数列，可直接使用求和公式，得到时间复杂度为

$$\begin{aligned} T(h) &= 2 \frac{1 - 2^h}{1 - 2} - h \\ &= 2^{h+1} - h \\ &= O(2^h) \end{aligned}$$

进一步地，高度为 h 的完美二叉树的结点数量为 $n = 2^{h+1} - 1$ ，易得复杂度为 $O(2^h) = O(n)$ 。以上推算表明，输入列表并建堆的时间复杂度为 $O(n)$ ，非常高效。

8.3. 小结

- 堆是一棵限定条件下的完全二叉树，根据成立条件可分为大顶堆和小顶堆。大（小）顶堆的堆顶元素最大（小）。
- 优先队列定义为一种具有出队优先级的队列。堆是实现优先队列的最常用数据结构。

- 堆的常用操作和对应时间复杂度为元素入堆 $O(\log n)$ 、堆顶元素出堆 $O(\log n)$ 、访问堆顶元素 $O(1)$ 等。
- 完全二叉树非常适合用数组来表示，因此我们一般用数组来存储堆。
- 堆化操作用于修复堆的特性，在入堆和出堆操作中都会使用到。
- 输入 n 个元素并建堆的时间复杂度可以被优化至 $O(n)$ ，非常高效。

9. 图

9.1. 图

「图 Graph」是一种非线性数据结构，由「顶点 Vertex」和「边 Edge」组成。我们可将图 G 抽象地表示为一组顶点 V 和一组边 E 的集合。例如，以下表示一个包含 5 个顶点和 7 条边的图

$$\begin{aligned}V &= \{1, 2, 3, 4, 5\} \\E &= \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (4, 5)\} \\G &= \{V, E\}\end{aligned}$$

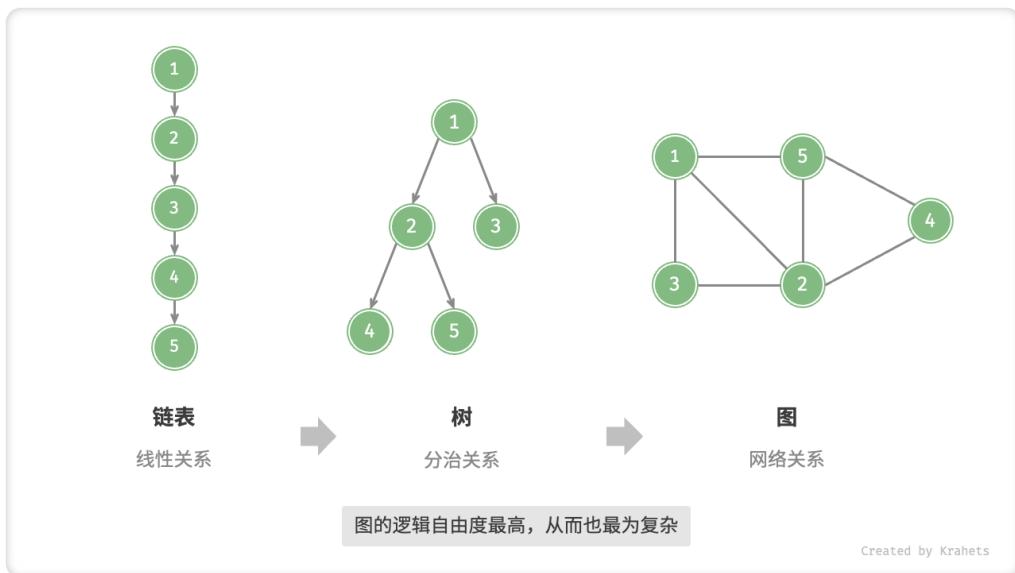


Figure 9-1. 链表、树、图之间的关系

那么，图与其他数据结构的关系是什么？如果我们把「顶点」看作结点，把「边」看作连接各个结点的指针，则可将「图」看成一种从「链表」拓展而来的数据结构。**相比线性关系（链表）和分治关系（树），网络关系（图）的自由度更高，也从而更为复杂。**

9.1.1. 图常见类型

根据边是否有方向，分为「无向图 Undirected Graph」和「有向图 Directed Graph」。

- 在无向图中，边表示两顶点之间“双向”的连接关系，例如微信或 QQ 中的“好友关系”；
- 在有向图中，边是有方向的，即 $A \rightarrow B$ 和 $A \leftarrow B$ 两个方向的边是相互独立的，例如微博或抖音上的“关注”与“被关注”关系；

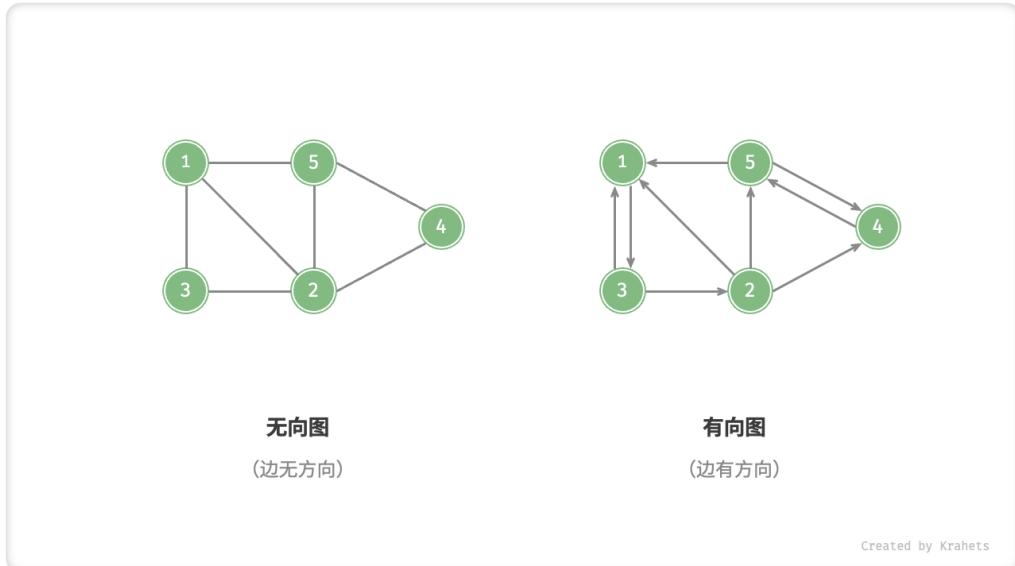


Figure 9-2. 有向图与无向图

根据所有顶点是否连通，分为「连通图 Connected Graph」和「非连通图 Disconnected Graph」。

- 对于连通图，从某个顶点出发，可以到达其余任意顶点；
- 对于非连通图，从某个顶点出发，至少有一个顶点无法到达；

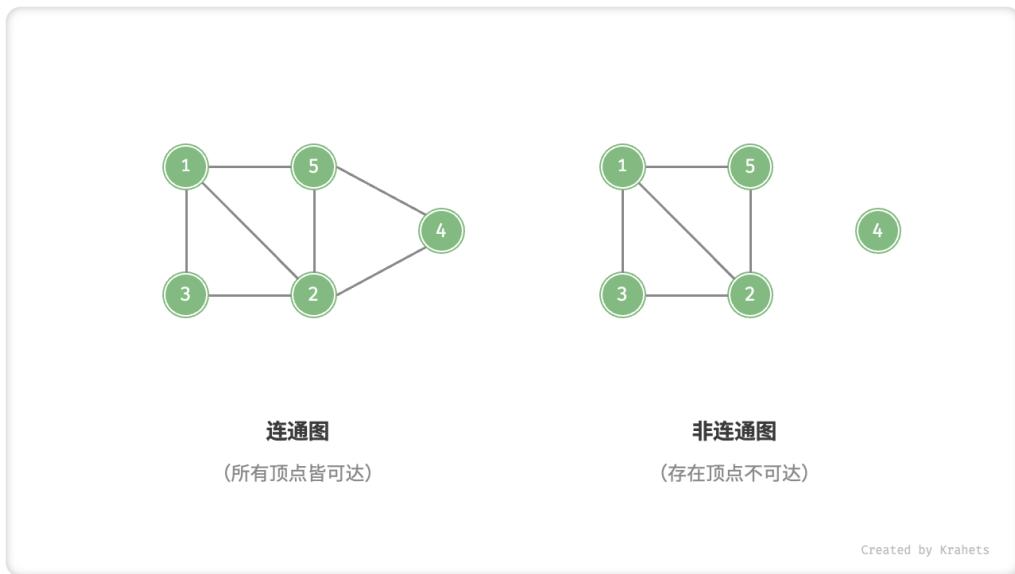


Figure 9-3. 连通图与非连通图

我们可以给边添加“权重”变量，得到「有权图 Weighted Graph」。例如，在王者荣耀等游戏中，系统会根据共同游戏时间来计算玩家之间的“亲密度”，这种亲密度网络就可以使用有权图来表示。

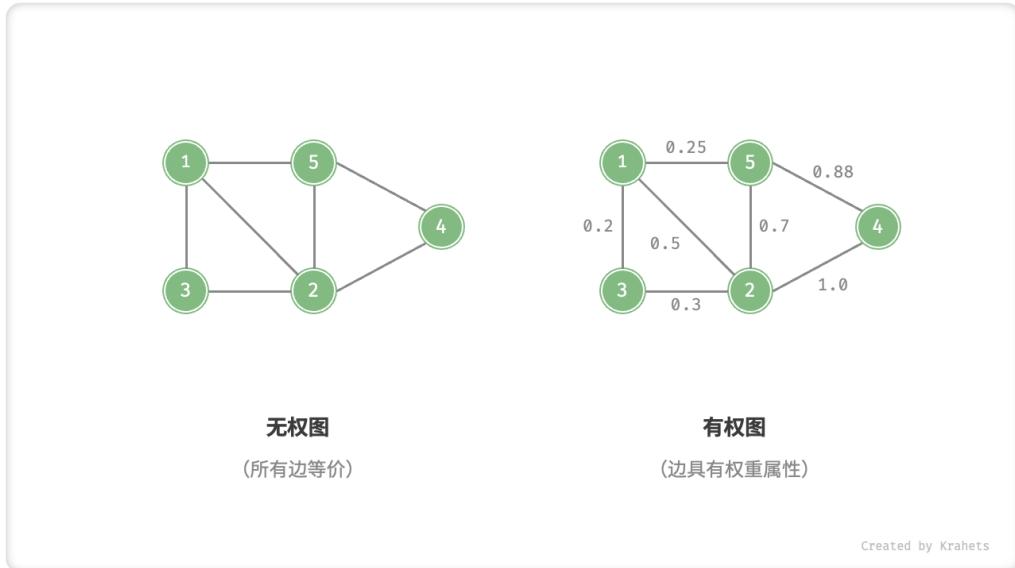


Figure 9-4. 有权图与无权图

9.1.2. 图常用术语

- 「邻接 Adjacency」：当两顶点之间有边相连时，称此两顶点“邻接”。例如，上图中顶点 1 的邻接顶点为顶点 2, 3, 5。
- 「路径 Path」：从顶点 A 到顶点 B 走过的边构成的序列，被称为从 A 到 B 的“路径”。例如，上图中 1, 5, 2, 4 是顶点 1 到顶点 4 的一个路径。
- 「度 Degree」表示一个顶点具有多少条边。对于有向图，「入度 In-Degree」表示有多少条边指向该顶点，「出度 Out-Degree」表示有多少条边从该顶点指出。

9.1.3. 图的表示

图的常用表示方法有「邻接矩阵」和「邻接表」。以下使用「无向图」来举例。

邻接矩阵

设图的顶点数量为 n ，「邻接矩阵 Adjacency Matrix」使用一个 $n \times n$ 大小的矩阵来表示图，每一行（列）代表一个顶点，矩阵元素代表边，使用 1 或 0 来表示两个顶点之间有边或无边。

如下图所示，记邻接矩阵为 M 、顶点列表为 V ，则矩阵元素 $M[i][j] = 1$ 代表着顶点 $V[i]$ 到顶点 $V[j]$ 之间有边，相反地 $M[i][j] = 0$ 代表两顶点之间无边。

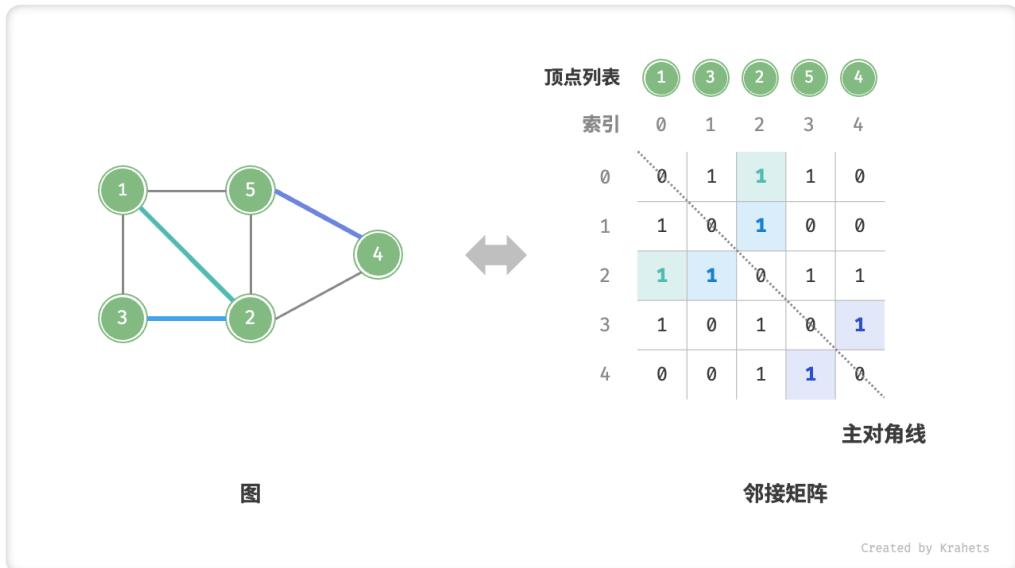


Figure 9-5. 图的邻接矩阵表示

邻接矩阵具有以下性质：

- 顶点不能与自身相连，因而邻接矩阵主对角线元素没有意义。
- 「无向图」两个方向的边等价，此时邻接矩阵关于主对角线对称。
- 将邻接矩阵的元素从 1, 0 替换为权重，则能够表示「有权图」。

使用邻接矩阵表示图时，我们可以直接通过访问矩阵元素来获取边，因此增删查操作的效率很高，时间复杂度均为 $O(1)$ 。然而，矩阵的空间复杂度为 $O(n^2)$ ，内存占用较大。

邻接表

「邻接表 Adjacency List」使用 n 个链表来表示图，链表结点表示顶点。第 i 条链表对应顶点 i ，其中存储了该顶点的所有邻接顶点（即与该顶点相连的顶点）。

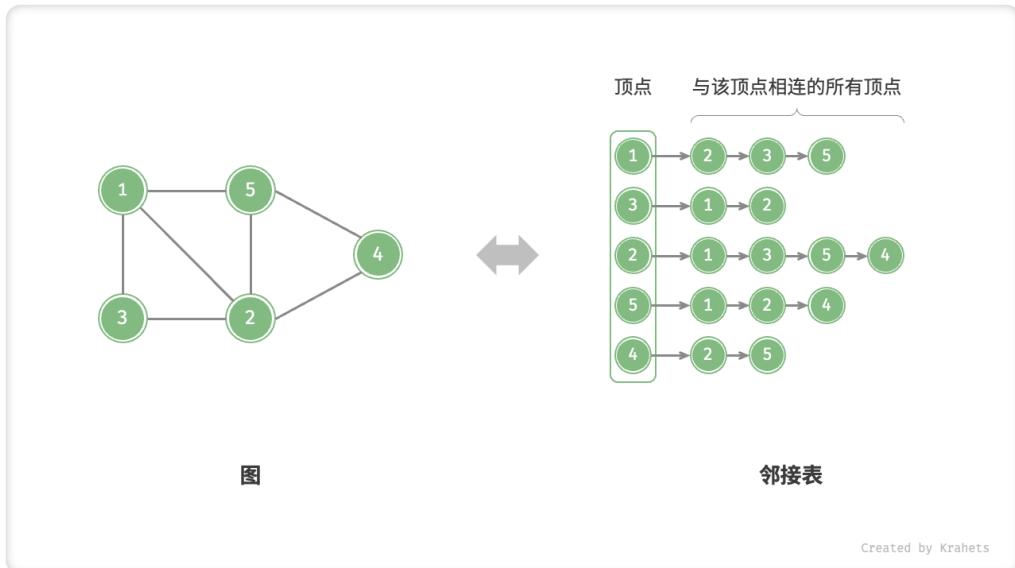


Figure 9-6. 图的邻接表表示

邻接表仅存储存在的边，而边的总数往往远小于 n^2 ，因此更加节省空间。但是，因为在邻接表中需要通过遍历链表来查找边，所以其时间效率不如邻接矩阵。

观察上图发现，邻接表结构与哈希表「链地址法」非常相似，因此我们也可以用类似方法来优化效率。比如，当链表较长时，可以把链表转化为 AVL 树或红黑树，从而将时间效率从 $O(n)$ 优化至 $O(\log n)$ ，还可以通过中序遍历获取有序序列；还可以将链表转化为哈希表，将时间复杂度降低至 $O(1)$ 。

9.1.4. 图常见应用

现实中的许多系统都可以使用图来建模，对应的待求解问题也可以被约化为图计算问题。

	顶点	边	图计算问题
社交网络	用户	好友关系	潜在好友推荐
地铁线路	站点	站点间的连通性	最短路线推荐
太阳系	星体	星体间的万有引力作用	行星轨道计算

9.2. 图基础操作

图的基础操作分为对「边」的操作和对「顶点」的操作，在「邻接矩阵」和「邻接表」这两种表示下的实现方式不同。

9.2.1. 基于邻接矩阵的实现

设图的顶点总数为 n ，则有：

- **添加或删除边**：直接在邻接矩阵中修改指定边的对应元素即可，使用 $O(1)$ 时间。而由于是无向图，因此需要同时更新两个方向的边。
- **添加顶点**：在邻接矩阵的尾部添加一行一列，并全部填 0 即可，使用 $O(n)$ 时间。
- **删除顶点**：在邻接矩阵中删除一行一列。当删除首行首列时达到最差情况，需要将 $(n - 1)^2$ 个元素“向左上移动”，从而使用 $O(n^2)$ 时间。
- **初始化**：传入 n 个顶点，初始化长度为 n 的顶点列表 `vertices`，使用 $O(n)$ 时间；初始化 $n \times n$ 大小的邻接矩阵 `adjMat`，使用 $O(n^2)$ 时间。



Figure 9-7. 邻接矩阵的初始化、增删边、增删顶点

以下是基于邻接矩阵表示图的实现代码。

```
// === File: graph_adjacency_matrix.cpp ===
/* 基于邻接矩阵实现的无向图类 */
class GraphAdjMat {
    vector<int> vertices;           // 顶点列表，元素代表“顶点值”，索引代表“顶点索引”
    vector<vector<int>> adjMat; // 邻接矩阵，行列索引对应“顶点索引”

public:
    /* 构造方法 */
    GraphAdjMat(const vector<int>& vertices, const vector<vector<int>>& edges) {
        // 添加顶点
        for (int val : vertices) {
            addVertex(val);
        }
        // 添加边
        // 请注意，edges 元素代表顶点索引，即对应 vertices 元素索引
        for (const vector<int>& edge : edges) {
            addEdge(edge[0], edge[1]);
        }
    }

    /* 获取顶点数量 */
    int size() const {
        return vertices.size();
    }

    /* 添加顶点 */
    void addVertex(int val) {
        int n = size();
        // 向顶点列表中添加新顶点的值
        vertices.push_back(val);
        // 在邻接矩阵中添加一行
        adjMat.emplace_back(n, 0);
        // 在邻接矩阵中添加一列
        for (vector<int>& row : adjMat) {
            row.push_back(0);
        }
    }

    /* 删除顶点 */
    void removeVertex(int index) {
        if (index >= size()) {
            throw out_of_range("顶点不存在");
        }
        // 在顶点列表中移除索引 index 的顶点
        vertices.erase(vertices.begin() + index);
        // 在邻接矩阵中删除索引 index 的行
    }
}
```

```
adjMat.erase(adjMat.begin() + index);
// 在邻接矩阵中删除索引 index 的列
for (vector<int>& row : adjMat) {
    row.erase(row.begin() + index);
}
}

/* 添加边 */
// 参数 i, j 对应 vertices 元素索引
void addEdge(int i, int j) {
    // 索引越界与相等处理
    if (i < 0 || j < 0 || i >= size() || j >= size() || i == j) {
        throw out_of_range("顶点不存在");
    }
    // 在无向图中, 邻接矩阵沿主对角线对称, 即满足 (i, j) == (j, i)
    adjMat[i][j] = 1;
    adjMat[j][i] = 1;
}

/* 删除边 */
// 参数 i, j 对应 vertices 元素索引
void removeEdge(int i, int j) {
    // 索引越界与相等处理
    if (i < 0 || j < 0 || i >= size() || j >= size() || i == j) {
        throw out_of_range("顶点不存在");
    }
    adjMat[i][j] = 0;
    adjMat[j][i] = 0;
}

/* 打印邻接矩阵 */
void print() {
    cout << "顶点列表 = ";
    PrintUtil::printVector(vertices);
    cout << "邻接矩阵 =" << endl;
    PrintUtil::printVectorMatrix(adjMat);
}
};
```

9.2.2. 基于邻接表的实现

设图的顶点总数为 n 、边总数为 m ，则有：

- **添加边：**在顶点对应链表的尾部添加边即可，使用 $O(1)$ 时间。因为是无向图，所以需要同时添加两个方向的边。

- **删除边**: 在顶点对应链表中查询与删除指定边, 使用 $O(m)$ 时间。与添加边一样, 需要同时删除两个方向的边。
- **添加顶点**: 在邻接表中添加一个链表即可, 并以新增顶点为链表头结点, 使用 $O(1)$ 时间。
- **删除顶点**: 需要遍历整个邻接表, 删除包含指定顶点的所有边, 使用 $O(n + m)$ 时间。
- **初始化**: 需要在邻接表中建立 n 个结点和 $2m$ 条边, 使用 $O(n + m)$ 时间。

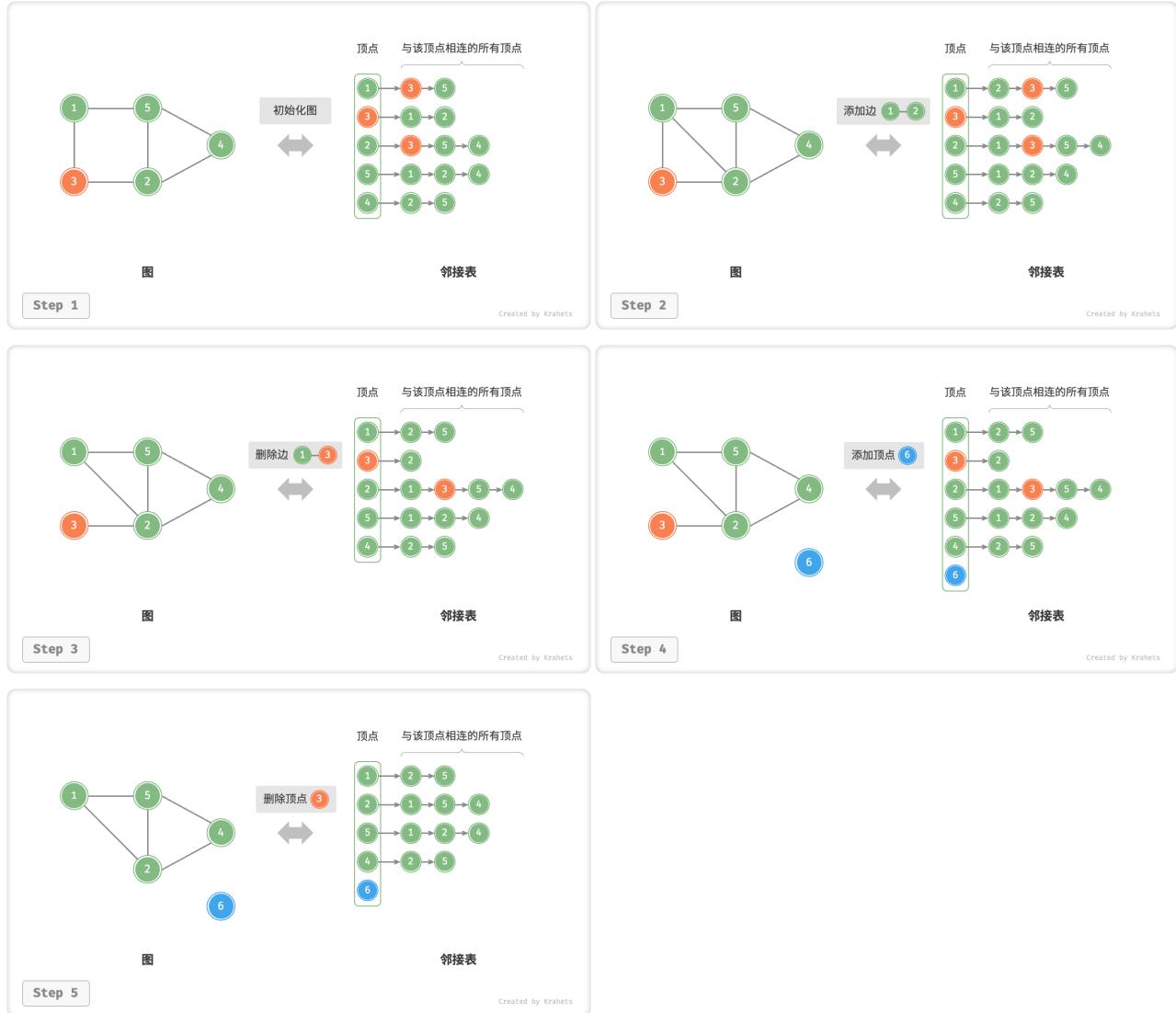


Figure 9-8. 邻接表的初始化、增删边、增删顶点

基于邻接表实现图的代码如下所示。细心的同学可能注意到, 我们在邻接表中使用 `Vertex` 结点类来表示顶点, 这样做的原因是:

- 如果我们选择通过顶点值来区分不同顶点, 那么值重复的顶点将无法被区分。
- 如果类似邻接矩阵那样, 使用顶点列表索引来区分不同顶点。那么, 假设我们想要删除索引为 i 的顶点, 则需要遍历整个邻接表, 将其中 $> i$ 的索引全部执行 -1 , 这样操作效率太低。
- 因此我们考虑引入顶点类 `Vertex`, 使得每个顶点都是唯一的对象, 此时删除顶点时就无需改动其余顶点了。

```
// === File: graph_adjacency_list.cpp ===
/* 基于邻接表实现的无向图类 */
class GraphAdjList {
public:
    // 邻接表, key: 顶点, value: 该顶点的所有邻接顶点
    unordered_map<Vertex*, vector<Vertex*>> adjList;

    /* 在 vector 中删除指定结点 */
    void remove(vector<Vertex*> &vec, Vertex *vet) {
        for (int i = 0; i < vec.size(); i++) {
            if (vec[i] == vet) {
                vec.erase(vec.begin() + i);
                break;
            }
        }
    }

    /* 构造方法 */
    GraphAdjList(const vector<vector<Vertex*>>& edges) {
        // 添加所有顶点和边
        for (const vector<Vertex*>& edge : edges) {
            addVertex(edge[0]);
            addVertex(edge[1]);
            addEdge(edge[0], edge[1]);
        }
    }

    /* 获取顶点数量 */
    int size() { return adjList.size(); }

    /* 添加边 */
    void addEdge(Vertex* vet1, Vertex* vet2) {
        if (!adjList.count(vet1) || !adjList.count(vet2) || vet1 == vet2)
            throw invalid_argument("不存在顶点");
        // 添加边 vet1 -> vet2
        adjList[vet1].push_back(vet2);
        adjList[vet2].push_back(vet1);
    }

    /* 删除边 */
    void removeEdge(Vertex* vet1, Vertex* vet2) {
        if (!adjList.count(vet1) || !adjList.count(vet2) || vet1 == vet2)
            throw invalid_argument("不存在顶点");
        // 删除边 vet1 -> vet2
        remove(adjList[vet1], vet2);
        remove(adjList[vet2], vet1);
    }
}
```

```
}

/* 添加顶点 */
void addVertex(Vertex* vet) {
    if (adjList.count(vet)) return;
    // 在邻接表中添加一个新链表
    adjList[vet] = vector<Vertex*>();
}

/* 删除顶点 */
void removeVertex(Vertex* vet) {
    if (!adjList.count(vet))
        throw invalid_argument(" 不存在顶点");
    // 在邻接表中删除顶点 vet 对应的链表
    adjList.erase(vet);
    // 遍历其它顶点的链表，删除所有包含 vet 的边
    for (auto& [key, vec] : adjList) {
        remove(vec, vet);
    }
}

/* 打印邻接表 */
void print() {
    cout << " 邻接表 =" << endl;
    for (auto& [key, vec] : adjList) {
        cout << key->val << ":" ;
        PrintUtil::printVector(vetsToVals(vec));
    }
}
};
```

9.2.3. 效率对比

设图中共有 n 个顶点和 m 条边，下表为邻接矩阵和邻接表的时间和空间效率对比。

	邻接矩阵	邻接表（链表）	邻接表（哈希表）
判断是否邻接	$O(1)$	$O(m)$	$O(1)$
添加边	$O(1)$	$O(1)$	$O(1)$
删除边	$O(1)$	$O(m)$	$O(1)$
添加顶点	$O(n)$	$O(1)$	$O(1)$
删除顶点	$O(n^2)$	$O(n + m)$	$O(n)$

	邻接矩阵	邻接表（链表）	邻接表（哈希表）
内存空间占用	$O(n^2)$	$O(n + m)$	$O(n + m)$

观察上表，貌似邻接表（哈希表）的时间与空间效率最优。但实际上，在邻接矩阵中操作边的效率更高，只需要一次数组访问或赋值操作即可。总结以上，邻接矩阵体现“以空间换时间”，邻接表体现“以时间换空间”。

9.3. 图的遍历



图与树的关系

树代表的是“一对多”的关系，而图则自由度更高，可以代表任意“多对多”关系。本质上，可以把树看作是图的一类特例。那么显然，树遍历操作也是图遍历操作的一个特例，两者的方法是非常类似的，建议你在学习本章节的过程中将两者融会贯通。

「图」与「树」都是非线性数据结构，都需要使用「搜索算法」来实现遍历操作。

类似地，图的遍历方式也分为两种，即「广度优先遍历 Breadth-First Traversal」和「深度优先遍历 Depth-First Traversal」，也称「广度优先搜索 Breadth-First Search」和「深度优先搜索 Depth-First Search」，简称为 BFS 和 DFS。

9.3.1. 广度优先遍历

广度优先遍历是一种由近及远的遍历方式，从距离最近的顶点开始访问，并一层层向外扩张。具体地，从某个顶点出发，先遍历该顶点的所有邻接顶点，随后遍历下个顶点的所有邻接顶点，以此类推……

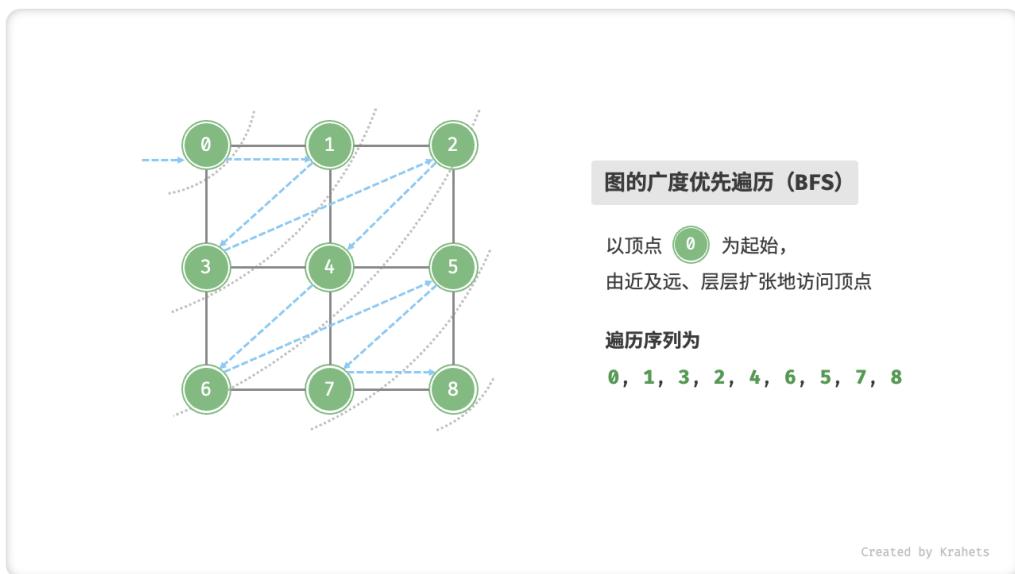


Figure 9-9. 图的广度优先遍历

算法实现

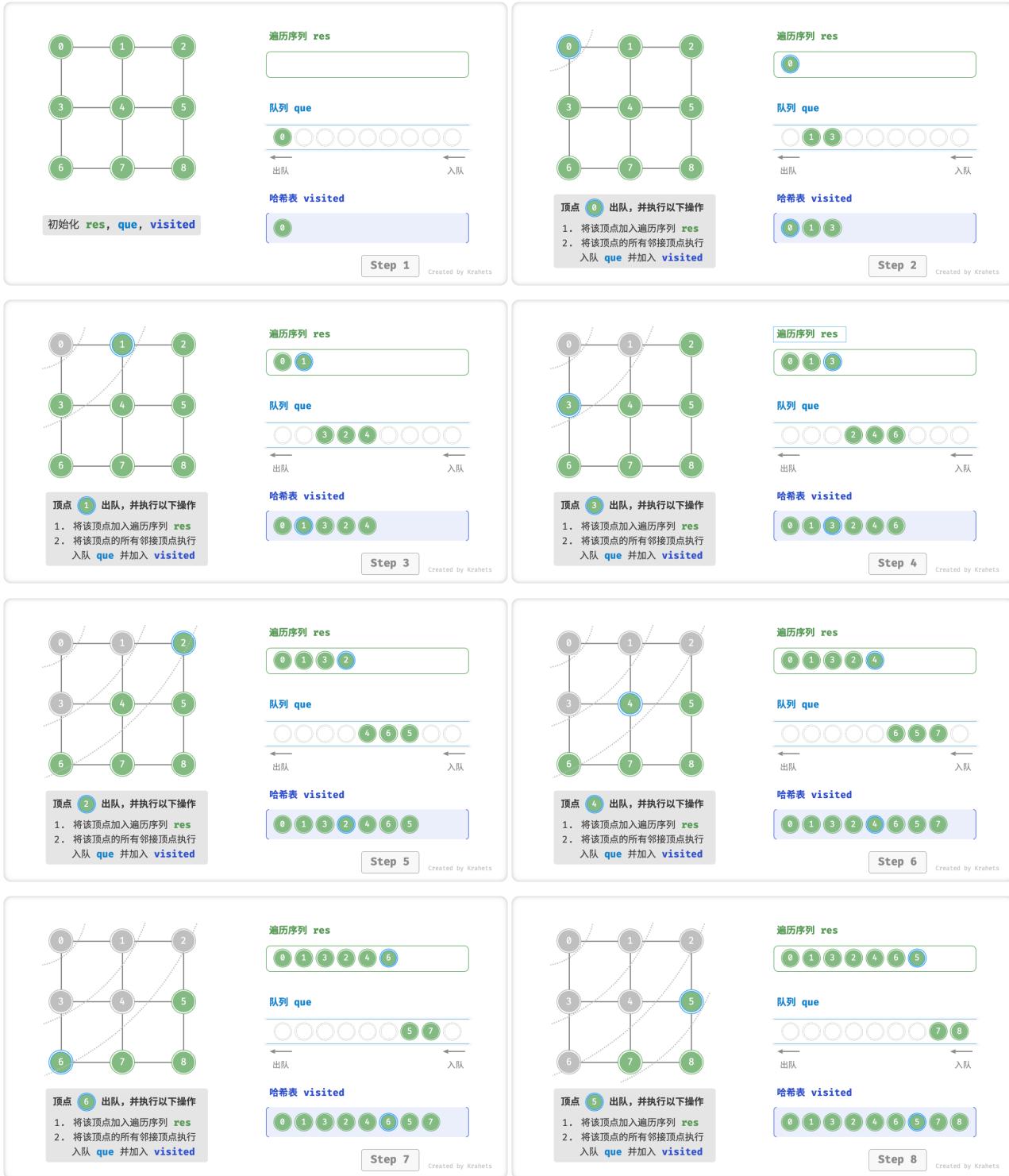
BFS 常借助「队列」来实现。队列具有“先入先出”的性质，这与 BFS “由近及远”的思想是异曲同工的。

1. 将遍历起始顶点 `startVet` 加入队列，并开启循环；
2. 在循环的每轮迭代中，弹出队首顶点并记录访问，并将该顶点的所有邻接顶点加入到队列尾部；
3. 循环 2.，直到所有顶点访问完成后结束；

为了防止重复遍历顶点，我们需要借助一个哈希表 `visited` 来记录哪些结点已被访问。

```
// == File: graph_bfs.cpp ==
/* 广度优先遍历 BFS */
// 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
vector<Vertex*> graphBFS(GraphAdjList &graph, Vertex *startVet) {
    // 顶点遍历序列
    vector<Vertex*> res;
    // 哈希表，用于记录已被访问过的顶点
    unordered_set<Vertex*> visited = { startVet };
    // 队列用于实现 BFS
    queue<Vertex*> que;
    que.push(startVet);
    // 以顶点 vet 为起点，循环直至访问完所有顶点
    while (!que.empty()) {
        Vertex *vet = que.front();
        que.pop();           // 队首顶点出队
        res.push_back(vet); // 记录访问顶点
        // 遍历该顶点的所有邻接顶点
        for (auto adjVet : graph.adjList[vet]) {
            if (visited.count(adjVet))
                continue;          // 跳过已被访问过的顶点
            que.push(adjVet);     // 只入队未访问的顶点
            visited.emplace(adjVet); // 标记该顶点已被访问
        }
    }
    // 返回顶点遍历序列
    return res;
}
```

代码相对抽象，建议对照以下动画图示来加深理解。



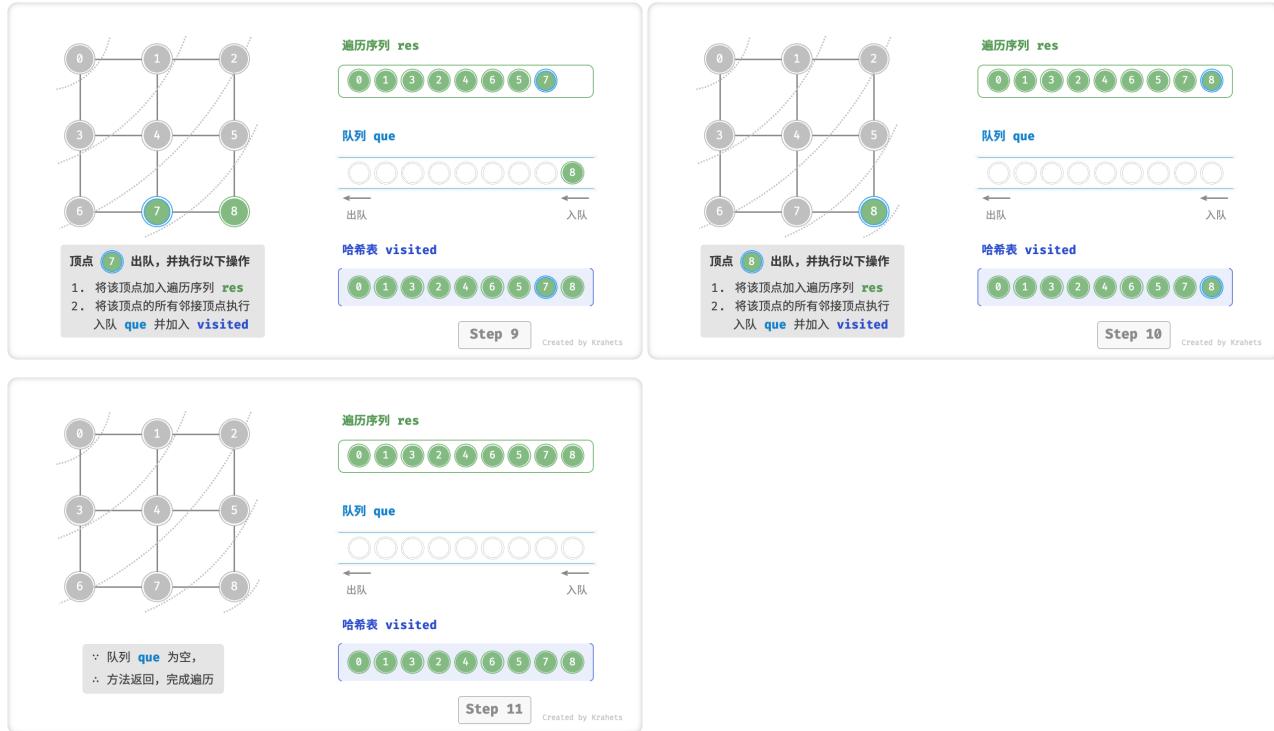


Figure 9-10. 图的广度优先遍历步骤



广度优先遍历的序列是否唯一？

不唯一。广度优先遍历只要求“由近及远”，而多个相同距离的顶点的遍历顺序允许被任意打乱。以上图为例，顶点 1, 3 的访问顺序可以交换、顶点 2, 4, 6 的访问顺序也可以任意交换、以此类推……

复杂度分析

时间复杂度：所有顶点都会入队、出队一次，使用 $O(|V|)$ 时间；在遍历邻接顶点的过程中，由于是无向图，因此所有边都会被访问 2 次，使用 $O(2|E|)$ 时间；总体使用 $O(|V| + |E|)$ 时间。

空间复杂度：列表 res，哈希表 visited，队列 que 中的顶点数量最多为 $|V|$ ，使用 $O(|V|)$ 空间。

9.3.2. 深度优先遍历

深度优先遍历是一种优先走到底、无路可走再回头的遍历方式。具体地，从某个顶点出发，不断地访问当前结点的某个邻接顶点，直到走到尽头时回溯，再继续走到底 + 回溯，以此类推……直至所有顶点遍历完成时结束。

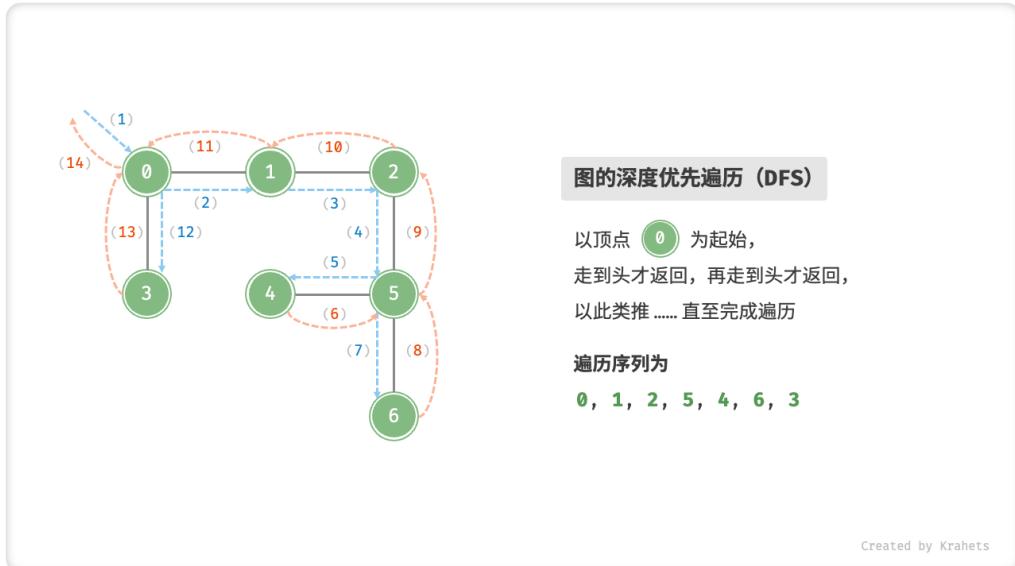


Figure 9-11. 图的深度优先遍历

算法实现

这种“走到头 + 回溯”的算法形式一般基于递归来实现。与 BFS 类似，在 DFS 中我们也需要借助一个哈希表 `visited` 来记录已被访问的顶点，以避免重复访问顶点。

```
// === File: graph_dfs.cpp ===
/* 深度优先遍历 DFS 辅助函数 */
void dfs(GraphAdjList& graph, unordered_set<Vertex*>& visited, vector<Vertex*>& res, Vertex* vet) {
    res.push_back(vet); // 记录访问顶点
    visited.emplace(vet); // 标记该顶点已被访问
    // 遍历该顶点的所有邻接顶点
    for (Vertex* adjVet : graph.adjList[vet]) {
        if (visited.count(adjVet))
            continue; // 跳过已被访问过的顶点
        // 递归访问邻接顶点
        dfs(graph, visited, res, adjVet);
    }
}

/* 深度优先遍历 DFS */
// 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
vector<Vertex*> graphDFS(GraphAdjList& graph, Vertex* startVet) {
    // 顶点遍历序列
    vector<Vertex*> res;
    // 哈希表，用于记录已被访问过的顶点
    unordered_set<Vertex*> visited;
    dfs(graph, visited, res, startVet);
}
```

```

    return res;
}

```

深度优先遍历的算法流程如下图所示，其中

- 直虚线代表向下递推，代表开启了一个新的递归方法来访问新顶点；
- 曲虚线代表向上回溯，代表此递归方法已经返回，回溯到了开启此递归方法的位置；

为了加深理解，请你将图示与代码结合起来，在脑中（或者用笔画下来）模拟整个 DFS 过程，包括每个递归方法何时开启、何时返回。





Figure 9-12. 图的深度优先遍历步骤



深度优先遍历的序列是否唯一？

与广度优先遍历类似，深度优先遍历序列的顺序也不是唯一的。给定某顶点，先往哪个方向探索都行，都是深度优先遍历。

以树的遍历为例，“根 → 左 → 右”、“左 → 根 → 右”、“左 → 右 → 根” 分别对应前序、中序、后序遍历，体现三种不同的遍历优先级，而三者都属于深度优先遍历。

复杂度分析

时间复杂度：所有顶点都被访问一次；所有边都被访问了 2 次，使用 $O(2|E|)$ 时间；总体使用 $O(|V| + |E|)$ 时间。

空间复杂度：列表 `res`，哈希表 `visited` 顶点数量最多为 $|V|$ ，递归深度最大为 $|V|$ ，因此使用 $O(|V|)$ 空间。

9.4. 小结

- 图由顶点和边组成，可以表示为一组顶点和一组边构成的集合。
- 相比线性关系（链表）和分治关系（树），网络关系（图）的自由度更高，也从而更为复杂。
- 有向图的边存在方向，连通图中的任意顶点都可达，有权图的每条边都包含权重变量。
- 邻接矩阵使用方阵来表示图，每一行（列）代表一个顶点，矩阵元素代表边，使用 1 或 0 来表示两个顶点之间有边或无边。邻接矩阵的增删查操作效率很高，但占用空间大。
- 邻接表使用多个链表来表示图，第 i 条链表对应顶点 i ，其中存储了该顶点的所有邻接顶点。邻接表相对邻接矩阵更加节省空间，但由于需要通过遍历链表来查找边，因此时间效率较低。
- 当邻接表中的链表过长时，可以将其转化为红黑树或哈希表，从而提升查询效率。
- 从算法思想角度分析，邻接矩阵体现“以空间换时间”，邻接表体现“以时间换空间”
- 图可以用于建模各类现实系统，例如社交网络、地铁线路等。
- 树是图的一种特例，树的遍历也是图的遍历的一种特例。
- 图的广度优先遍历是一种由近及远、层层扩张的搜索方式，常借助队列实现。
- 图的深度优先遍历是一种优先走到底、无路可走再回头的搜索方式，常基于递归来实现。

10. 查找算法

10.1. 线性查找

「线性查找 Linear Search」是一种最基础的查找方法，其从数据结构的一端开始，依次访问每个元素，直到另一端后停止。

10.1.1. 算法实现

线性查找实质上就是遍历数据结构 + 判断条件。比如，我们想要在数组 `nums` 中查找目标元素 `target` 的对应索引，那么可以在数组中进行线性查找。

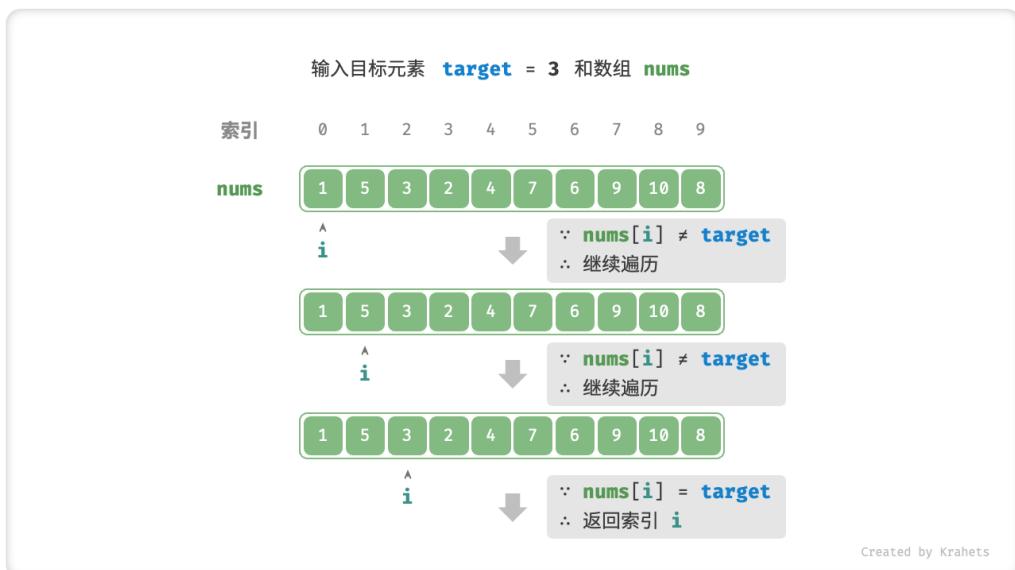


Figure 10-1. 在数组中线性查找元素

```
// === File: linear_search.cpp ===
/* 线性查找（数组） */
int linearSearchArray(vector<int>& nums, int target) {
    // 遍历数组
    for (int i = 0; i < nums.size(); i++) {
        // 找到目标元素，返回其索引
        if (nums[i] == target)
            return i;
    }
    // 未找到目标元素，返回 -1
    return -1;
}
```

再比如，我们想要在给定一个目标结点值 `target`，返回此结点对象，也可以在链表中进行线性查找。

```
// === File: linear_search.cpp ===
/* 线性查找（链表） */
ListNode* linearSearchLinkedList(ListNode* head, int target) {
    // 遍历链表
    while (head != nullptr) {
        // 找到目标结点，返回之
        if (head->val == target)
            return head;
        head = head->next;
    }
    // 未找到目标结点，返回 nullptr
    return nullptr;
}
```

10.1.2. 复杂度分析

时间复杂度 $O(n)$ ：其中 n 为数组或链表长度。

空间复杂度 $O(1)$ ：无需使用额外空间。

10.1.3. 优点与缺点

线性查找的通用性极佳。由于线性查找是依次访问元素的，即没有跳跃访问元素，因此数组或链表皆适用。

线性查找的时间复杂度太高。在数据量 n 很大时，查找效率很低。

10.2. 二分查找

「二分查找 Binary Search」利用数据的有序性，通过每轮缩小一半搜索区间来查找目标元素。

使用二分查找有两个前置条件：

- 要求输入数据是有序的，这样才能通过判断大小关系来排除一半的搜索区间；
- 二分查找仅适用于数组，而在链表中使用效率很低，因为其在循环中需要跳跃式（非连续地）访问元素。

10.2.1. 算法实现

给定一个长度为 n 的排序数组 `nums`，元素从小到大排列。数组的索引取值范围为

$$0, 1, 2, \dots, n - 1$$

使用「区间」来表示这个取值范围的方法主要有两种：

1. 双闭区间 $[0, n - 1]$ ，即两个边界都包含自身；此方法下，区间 $[0, 0]$ 仍包含一个元素；
2. 左闭右开 $[0, n)$ ，即左边界包含自身、右边界不包含自身；此方法下，区间 $[0, 0)$ 为空；

“双闭区间”实现

首先，我们先采用“双闭区间”的表示，在数组 `nums` 中查找目标元素 `target` 的对应索引。

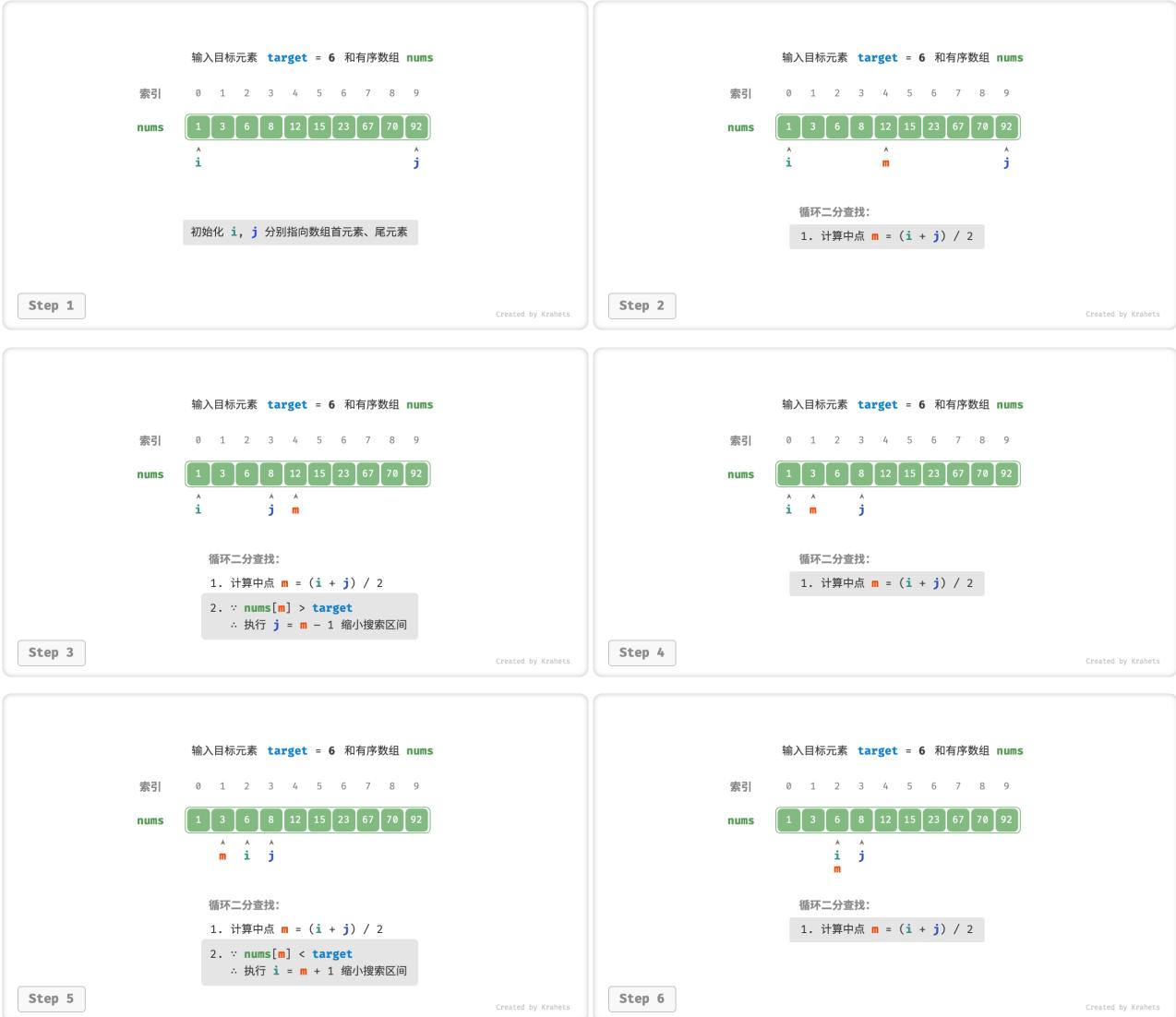




Figure 10-2. 二分查找步骤

二分查找“双闭区间”表示下的代码如下所示。

```
// === File: binary_search.cpp ===
/* 二分查找（双闭区间） */
int binarySearch(vector<int>& nums, int target) {
    // 初始化双闭区间 [0, n-1]，即 i, j 分别指向数组首元素、尾元素
    int i = 0, j = nums.size() - 1;
    // 循环，当搜索区间为空时跳出（当 i > j 时为空）
    while (i <= j) {
        int m = (i + j) / 2;          // 计算中点索引 m
        if (nums[m] < target)        // 此情况说明 target 在区间 [m+1, j] 中
            i = m + 1;
        else if (nums[m] > target)   // 此情况说明 target 在区间 [i, m-1] 中
            j = m - 1;
        else                         // 找到目标元素，返回其索引
            return m;
    }
    // 未找到目标元素，返回 -1
    return -1;
}
```

“左闭右开”实现

当然，我们也可以使用“左闭右开”的表示方法，写出相同功能的二分查找代码。

```
// === File: binary_search.cpp ===
/* 二分查找（左闭右开） */
int binarySearch1(vector<int>& nums, int target) {
    // 初始化左闭右开 [0, n)，即 i, j 分别指向数组首元素、尾元素 +1
    int i = 0, j = nums.size();
    // 循环，当搜索区间为空时跳出（当 i = j 时为空）
    while (i < j) {
```

```

int m = (i + j) / 2;           // 计算中点索引 m
if (nums[m] < target)         // 此情况说明 target 在区间 [m+1, j) 中
    i = m + 1;
else if (nums[m] > target)   // 此情况说明 target 在区间 [i, m) 中
    j = m;
else                          // 找到目标元素，返回其索引
    return m;
}
// 未找到目标元素，返回 -1
return -1;
}

```

两种表示对比

对比下来，两种表示的代码写法有以下不同点：

表示方法	初始化指针	缩小区间	循环终止条件
双闭区间 $[0, n - 1]$	$i = 0, j = n - 1$	$i = m + 1, j = m - 1$	$i > j$
左闭右开 $[0, n)$	$i = 0, j = n$	$i = m + 1, j = m$	$i = j$

观察发现，在“双闭区间”表示中，由于对左右两边界的定义是相同的，因此缩小区间的 i, j 处理方法也是对称的，这样更不容易出错。综上所述，建议你采用“双闭区间”的写法。

大数越界处理

当数组长度很大时，加法 $i + j$ 的结果有可能会超出 `int` 类型的取值范围。在此情况下，我们需要换一种计算中点的写法。

```

// (i + j) 可能超出 int 的取值范围
int m = (i + j) / 2;
// 更换为此写法则不会越界
int m = i + (j - i) / 2;

```

10.2.2. 复杂度分析

时间复杂度 $O(\log n)$ ：其中 n 为数组或链表长度；每轮排除一半的区间，因此循环轮数为 $\log_2 n$ ，使用 $O(\log n)$ 时间。

空间复杂度 $O(1)$ ：指针 i, j 使用常数大小空间。

10.2.3. 优点与缺点

二分查找效率很高，体现在：

- **二分查找时间复杂度低。**对数阶在数据量很大时具有巨大优势，例如，当数据大小 $n = 2^{20}$ 时，线性查找需要 $2^{20} = 1048576$ 轮循环，而二分查找仅需要 $\log_2 2^{20} = 20$ 轮循环。
- **二分查找不需要额外空间。**相对于借助额外数据结构来实现查找的算法来说，其更加节约空间使用。

但并不意味着所有情况下都应使用二分查找，这是因为：

- **二分查找仅适用于有序数据。**如果输入数据是无序的，为了使用二分查找而专门执行数据排序，那么是得不偿失的，因为排序算法的时间复杂度一般为 $O(n \log n)$ ，比线性查找和二分查找都更差。再例如，对于频繁插入元素的场景，为了保持数组的有序性，需要将元素插入到特定位置，时间复杂度为 $O(n)$ ，也是非常昂贵的。
- **二分查找仅适用于数组。**由于在二分查找中，访问索引是“非连续”的，因此链表或者基于链表实现的数据结构都无法使用。
- **在小数据量下，线性查找的性能更好。**在线性查找中，每轮只需要 1 次判断操作；而在二分查找中，需要 1 次加法、1 次除法、1~3 次判断操作、1 次加法（减法），共 4~6 个单元操作；因此，在数据量 n 较小时，线性查找反而比二分查找更快。

10.3. 哈希查找



在数据量很大时，「线性查找」太慢；而「二分查找」要求数据必须是有序的，并且只能在数组中应用。那么是否有方法可以同时避免上述缺点呢？答案是肯定的，此方法被称为「哈希查找」。

「哈希查找 Hash Searching」借助一个哈希表来存储需要的「键值对 Key Value Pair」，我们可以在 $O(1)$ 时间下实现“键 → 值”映射查找，体现着“以空间换时间”的算法思想。

10.3.1. 算法实现

如果我们想要给定数组中的一个目标元素 `target`，获取该元素的索引，那么可以借助一个哈希表实现查找。



Figure 10-3. 哈希查找数组索引

```
// === File: hashing_search.cpp ===
/* 哈希查找（数组） */
int hashingSearchArray(unordered_map<int, int> map, int target) {
    // 哈希表的 key: 目标元素, value: 索引
    // 若哈希表中无此 key , 返回 -1
    if (map.find(target) == map.end())
        return -1;
    return map[target];
}
```

再比如，如果我们想要给定一个目标结点值 `target`，获取对应的链表结点对象，那么也可以使用哈希查找实现。



Figure 10-4. 哈希查找链表结点

```
// === File: hashing_search.cpp ===
/* 哈希查找（链表） */
ListNode* hashingSearchLinkedList(unordered_map<int, ListNode*> map, int target) {
    // 哈希表的 key: 目标结点值, value: 结点对象
    // 若哈希表中无此 key , 返回 nullptr
    if (map.find(target) == map.end())
        return nullptr;
    return map[target];
}
```

10.3.2. 复杂度分析

时间复杂度 $O(1)$ ：哈希表的查找操作使用 $O(1)$ 时间。

空间复杂度 $O(n)$ ：其中 n 为数组或链表长度。

10.3.3. 优点与缺点

在哈希表中，**查找、插入、删除操作的平均时间复杂度都为 $O(1)$** ，这意味着无论是高频增删还是高频查找场景，哈希查找的性能表现都非常好。当然，一切的前提是保证哈希表未退化。

即使如此，哈希查找仍存在一些问题，在实际应用中，需要根据情况灵活选择方法。

- 辅助哈希表 **需要使用 $O(n)$ 的额外空间**，意味着需要预留更多的计算机内存；
- 建立和维护哈希表需要时间，因此哈希查找 **不适合高频增删、低频查找的使用场景**；
- 当哈希冲突严重时，哈希表会退化为链表，**时间复杂度劣化至 $O(n)$** ；
- **当数据量很小时，线性查找比哈希查找更快**。这是因为计算哈希映射函数可能比遍历一个小型数组更慢；

10.4. 小结

- 线性查找是一种最基础的查找方法，通过遍历数据结构 + 判断条件实现查找。
- 二分查找利用数据的有序性，通过循环不断缩小一半搜索区间来实现查找，其要求输入数据是有序的，并且仅适用于数组或基于数组实现的数据结构。
- 哈希查找借助哈希表来实现常数阶时间复杂度的查找操作，体现以空间换时间的算法思想。
- 下表总结对比了查找算法的各种特性和时间复杂度。

	线性查找	二分查找	哈希查找
适用数据结构	数组、链表	数组	数组、链表
输入数据要求	无	有序	无

	线性查找	二分查找	哈希查找
平均时间复杂度查找 / 插入 / 删 除	$O(n)$ / $O(1)$ / $O(n)$	$O(\log n)$ / $O(n)$ / $O(n)$	$O(1)$ / $O(1)$ / $O(1)$
最差时间复杂度查找 / 插入 / 删 除	$O(n)$ / $O(1)$ / $O(n)$	$O(\log n)$ / $O(n)$ / $O(n)$	$O(n)$ / $O(n)$ / $O(n)$
空间复杂度	$O(1)$	$O(1)$	$O(n)$

11. 排序算法

11.1. 排序简介

「排序算法 Sorting Algorithm」使得列表中的所有元素按照从小到大的顺序排列。

- 待排序的列表的 **元素类型** 可以是整数、浮点数、字符、或字符串；
- 排序算法可以根据需要设定 **判断规则**，例如数字大小、字符 ASCII 码顺序、自定义规则；

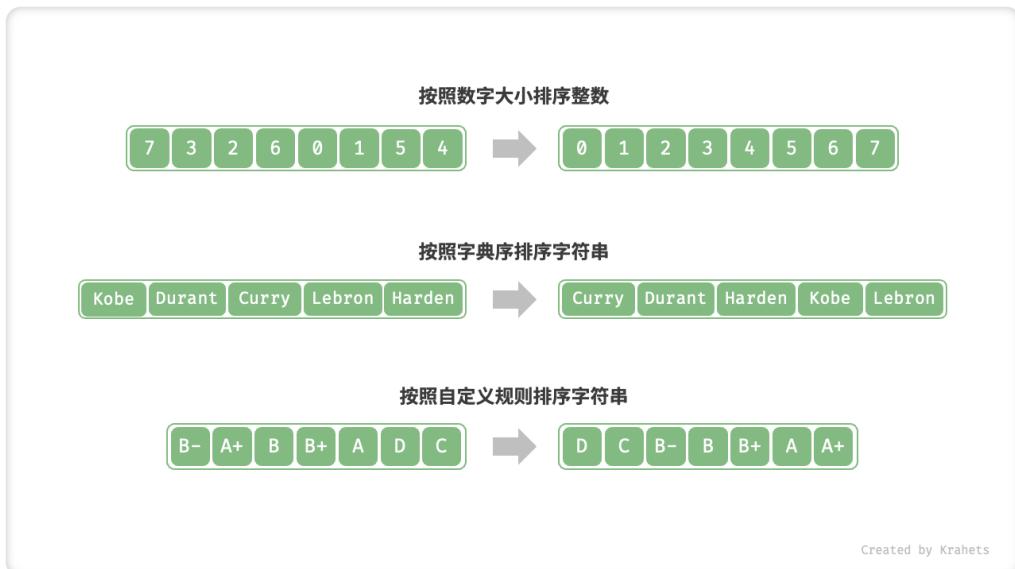


Figure 11-1. 排序中不同的元素类型和判断规则

11.1.1. 评价维度

排序算法主要可根据 **稳定性**、**就地性**、**自适应性**、**比较类** 来分类。

稳定性

- 「稳定排序」在完成排序后，**不改变** 相等元素在数组中的相对顺序。
- 「非稳定排序」在完成排序后，相等元素在数组中的相对位置 **可能被改变**。

假设我们有一个存储学生信息的表格，第 1, 2 列分别是姓名和年龄。那么在以下示例中，「非稳定排序」会导致输入数据的有序性丢失。因此「稳定排序」是很好的特性，在**多级排序中是必须的**。

```
# 输入数据是按照姓名排序好的
# (name, age)
('A', 19)
```

```
('B', 18)
('C', 21)
('D', 19)
('E', 23)

# 假设使用非稳定排序算法按年龄排序列表,
# 结果中 ('D', 19) 和 ('A', 19) 的相对位置改变,
# 输入数据按姓名排序的性质丢失
('B', 18)
('D', 19)
('A', 19)
('C', 21)
('E', 23)
```

就地性

- 「原地排序」无需辅助数据，不使用额外空间；
- 「非原地排序」需要借助辅助数据，使用额外空间；

「原地排序」不使用额外空间，可以节约内存；并且一般情况下，由于数据操作减少，原地排序的运行效率也更高。

自适应性

- 「自适应排序」的时间复杂度受输入数据影响，即最佳 / 最差 / 平均时间复杂度不相等。
- 「非自适应排序」的时间复杂度恒定，与输入数据无关。

我们希望 **最差 = 平均**，即不希望排序算法的运行效率在某些输入数据下发生劣化。

比较类

- 「比较类排序」基于元素之间的比较算子（小于、相等、大于）来决定元素的相对顺序。
- 「非比较类排序」不基于元素之间的比较算子来决定元素的相对顺序。

「比较类排序」的时间复杂度最优为 $O(n \log n)$ ；而「非比较类排序」可以达到 $O(n)$ 的时间复杂度，但通用性较差。

11.1.2. 理想排序算法

- **运行快**，即时间复杂度低；
- **稳定排序**，即排序后相等元素的相对位置不变化；
- **原地排序**，即运行中不使用额外的辅助空间；
- **正向自适应性**，即算法的运行效率不会在某些输入数据下发生劣化；

然而，没有排序算法同时具备以上所有特性。排序算法的选型使用取决于具体的列表类型、列表长度、元素分布等因素。

11.2. 冒泡排序

「冒泡排序 Bubble Sort」是一种最基础的排序算法，非常适合作为第一个学习的排序算法。顾名思义，「冒泡」是该算法的核心操作。

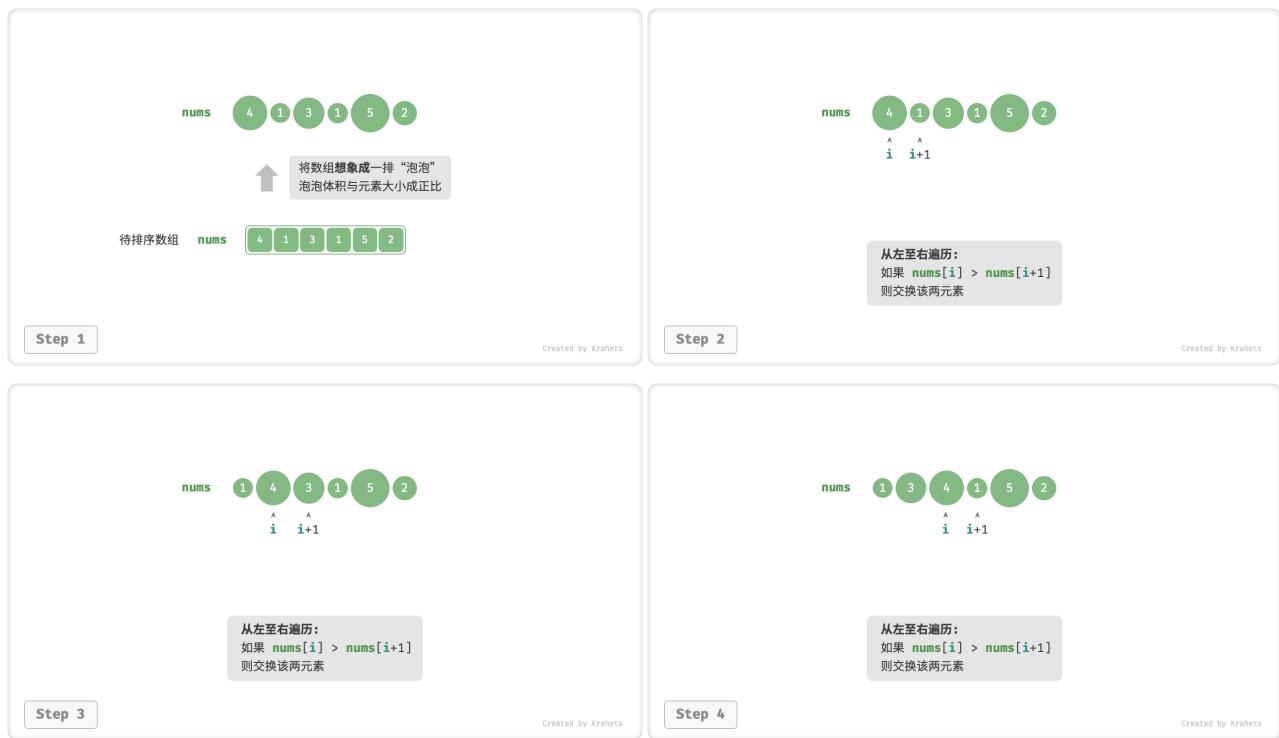


为什么叫“冒泡”

在水中，越大的泡泡浮力越大，所以最大的泡泡会最先浮到水面。

「冒泡」操作则是在模拟上述过程，具体做法为：从数组最左端开始向右遍历，依次对比相邻元素大小，若 **左元素 > 右元素** 则将它俩交换，最终可将最大元素移动至数组最右端。

完成此次冒泡操作后，数组最大元素已在正确位置，接下来只需排序剩余 $n - 1$ 个元素。



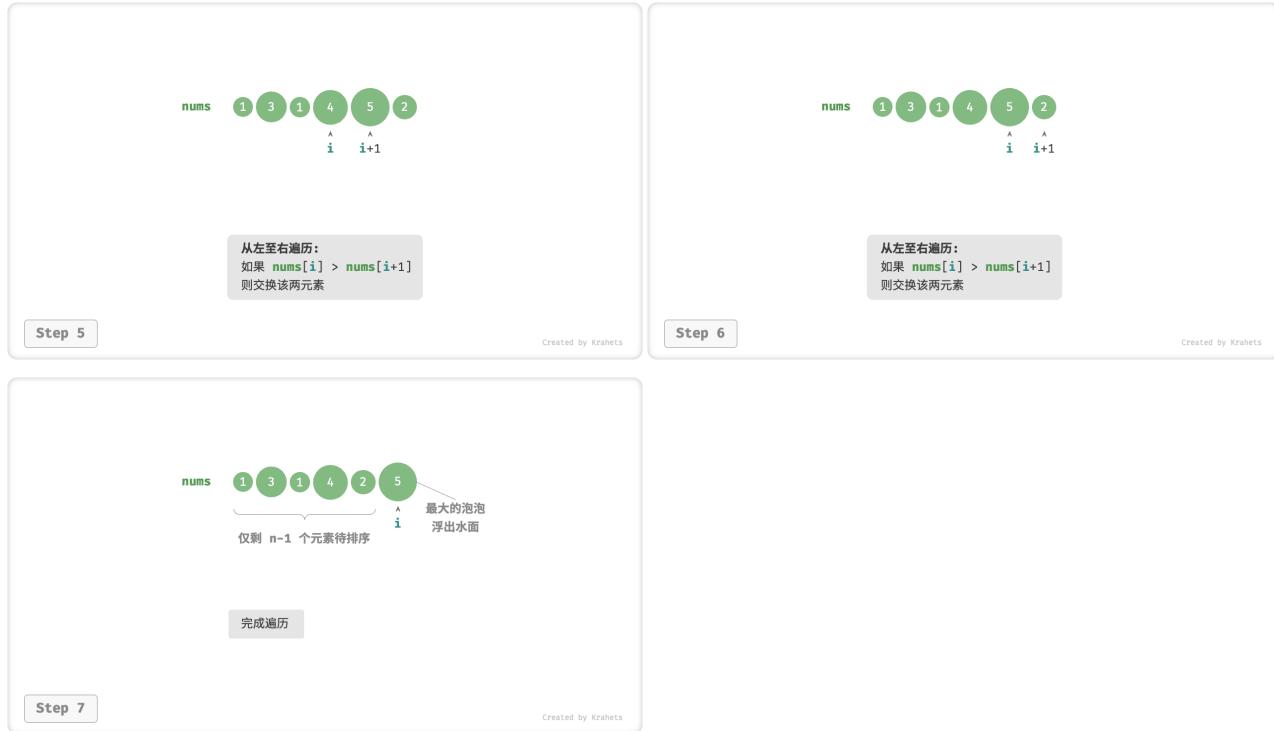


Figure 11-2. 冒泡操作步骤

11.2.1. 算法流程

1. 设数组长度为 n ，完成第一轮「冒泡」后，数组最大元素已在正确位置，接下来只需排序剩余 $n - 1$ 个元素。
2. 同理，对剩余 $n - 1$ 个元素执行「冒泡」，可将第二大元素交换至正确位置，因而待排序元素只剩 $n - 2$ 个。
3. 以此类推……循环 $n - 1$ 轮「冒泡」，即可完成整个数组的排序。



Figure 11-3. 冒泡排序流程

```
// === File: bubble_sort.cpp ===
/* 冒泡排序 */
void bubbleSort(vector<int>& nums) {
    // 外循环: 待排序元素数量为 n-1, n-2, ..., 1
    for (int i = nums.size() - 1; i > 0; i--) {
        // 内循环: 冒泡操作
        for (int j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // 交换 nums[j] 与 nums[j + 1]
                // 这里使用了 std::swap() 函数
                swap(nums[j], nums[j + 1]);
            }
        }
    }
}
```

11.2.2. 算法特性

时间复杂度 $O(n^2)$ ：各轮「冒泡」遍历的数组长度为 $n - 1, n - 2, \dots, 2, 1$ 次，求和为 $\frac{(n-1)n}{2}$ ，因此使用 $O(n^2)$ 时间。

空间复杂度 $O(1)$ ：指针 i, j 使用常数大小的额外空间。

原地排序：指针变量仅使用常数大小额外空间。

稳定排序：不交换相等元素。

自适应排序：引入 `flag` 优化后（见下文），最佳时间复杂度为 $O(N)$ 。

11.2.3. 效率优化

我们发现，若在某轮「冒泡」中未执行任何交换操作，则说明数组已经完成排序，可直接返回结果。考虑可以增加一个标志位 `flag` 来监听该情况，若出现则直接返回。

优化后，冒泡排序的最差和平均时间复杂度仍为 $O(n^2)$ ；而在输入数组 **已排序** 时，达到 **最佳时间复杂度** $O(n)$ 。

```
// === File: bubble_sort.cpp ===
/* 冒泡排序（标志优化）*/
void bubbleSortWithFlag(vector<int>& nums) {
    // 外循环：待排序元素数量为 n-1, n-2, ..., 1
    for (int i = nums.size() - 1; i > 0; i--) {
        bool flag = false; // 初始化标志位
        // 内循环：冒泡操作
        for (int j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // 交换 nums[j] 与 nums[j + 1]
                // 这里使用了 std::swap() 函数
                swap(nums[j], nums[j + 1]);
                flag = true; // 记录交换元素
            }
        }
        if (!flag) break; // 此轮冒泡未交换任何元素，直接跳出
    }
}
```

11.3. 插入排序

「插入排序 Insertion Sort」是一种基于 **数组插入操作** 的排序算法。

「插入操作」原理：选定某个待排序元素为基准数 `base`，将 `base` 与其左侧已排序区间元素依次对比大小，并插入到正确位置。

回忆数组插入操作，我们需要将从目标索引到 `base` 之间的所有元素向右移动一位，然后再将 `base` 赋值给目标索引。

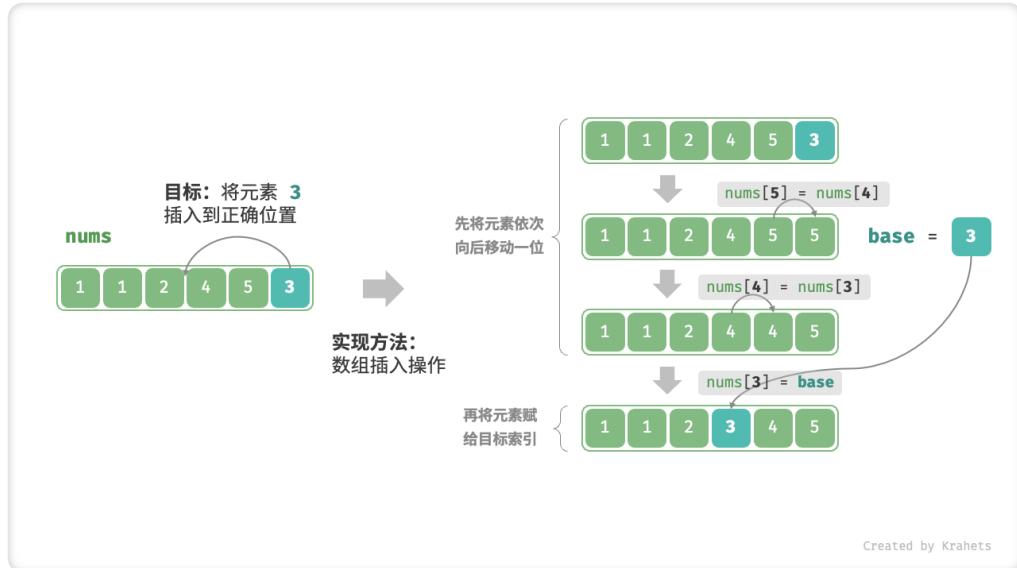


Figure 11-4. 单次插入操作

11.3.1. 算法流程

- 第 1 轮先选取数组的 第 2 个元素 为 `base`，执行「插入操作」后，数组前 2 个元素已完成排序。
- 第 2 轮选取 第 3 个元素 为 `base`，执行「插入操作」后，数组前 3 个元素已完成排序。
- 以此类推……最后一轮选取 数组尾元素 为 `base`，执行「插入操作」后，所有元素已完成排序。



Figure 11-5. 插入排序流程

```
// === File: insertion_sort.cpp ===
/* 插入排序 */
```

```

void insertionSort(vector<int>& nums) {
    // 外循环: base = nums[1], nums[2], ..., nums[n-1]
    for (int i = 1; i < nums.size(); i++) {
        int base = nums[i], j = i - 1;
        // 内循环: 将 base 插入到左边的正确位置
        while (j >= 0 && nums[j] > base) {
            nums[j + 1] = nums[j]; // 1. 将 nums[j] 向右移动一位
            j--;
        }
        nums[j + 1] = base; // 2. 将 base 赋值到正确位置
    }
}

```

11.3.2. 算法特性

时间复杂度 $O(n^2)$ ：最差情况下，各轮插入操作循环 $n-1, n-2, \dots, 2, 1$ 次，求和为 $\frac{(n-1)n}{2}$ ，使用 $O(n^2)$ 时间。

空间复杂度 $O(1)$ ：指针 i, j 使用常数大小的额外空间。

原地排序：指针变量仅使用常数大小额外空间。

稳定排序：不交换相等元素。

自适应排序：最佳情况下，时间复杂度为 $O(n)$ 。

11.3.3. 插入排序 vs 冒泡排序



虽然「插入排序」和「冒泡排序」的时间复杂度皆为 $O(n^2)$ ，但实际运行速度却有很大差别，这是为什么呢？

回顾复杂度分析，两个方法的循环次数都是 $\frac{(n-1)n}{2}$ 。但不同的是，「冒泡操作」是在做 **元素交换**，需要借助一个临时变量实现，共 3 个单元操作；而「插入操作」是在做 **赋值**，只需 1 个单元操作；因此，可以粗略估计出冒泡排序的计算开销约为插入排序的 3 倍。

插入排序运行速度快，并且具有原地、稳定、自适应的优点，因此很受欢迎。实际上，包括 Java 在内的许多编程语言的排序库函数的实现都用到了插入排序。库函数的大致思路：

- 对于 **长数组**，采用基于分治的排序算法，例如「快速排序」，时间复杂度为 $O(n \log n)$ ；
- 对于 **短数组**，直接使用「插入排序」，时间复杂度为 $O(n^2)$ ；

在数组较短时，复杂度中的常数项（即每轮中的单元操作数量）占主导作用，此时插入排序运行地更快。这个现象与「线性查找」和「二分查找」的情况类似。

11.4. 快速排序

「快速排序 Quick Sort」是一种基于“分治思想”的排序算法，速度很快、应用很广。

快速排序的核心操作为「哨兵划分」，其目标为：选取数组某个元素为 基准数，将所有小于基准数的元素移动至其左边，大于基准数的元素移动至其右边。「哨兵划分」的实现流程为：

1. 以数组最左端元素作为基准数，初始化两个指针 i, j 指向数组两端；
2. 设置一个循环，每轮中使用 i / j 分别寻找首个比基准数大 / 小的元素，并交换此两元素；
3. 不断循环步骤 2.，直至 i, j 相遇时跳出，最终把基准数交换至两个子数组的分界线；

「哨兵划分」执行完毕后，原数组被划分成两个部分，即 **左子数组** 和 **右子数组**，且满足 **左子数组任意元素 < 基准数 < 右子数组任意元素**。因此，接下来我们只需要排序两个子数组即可。



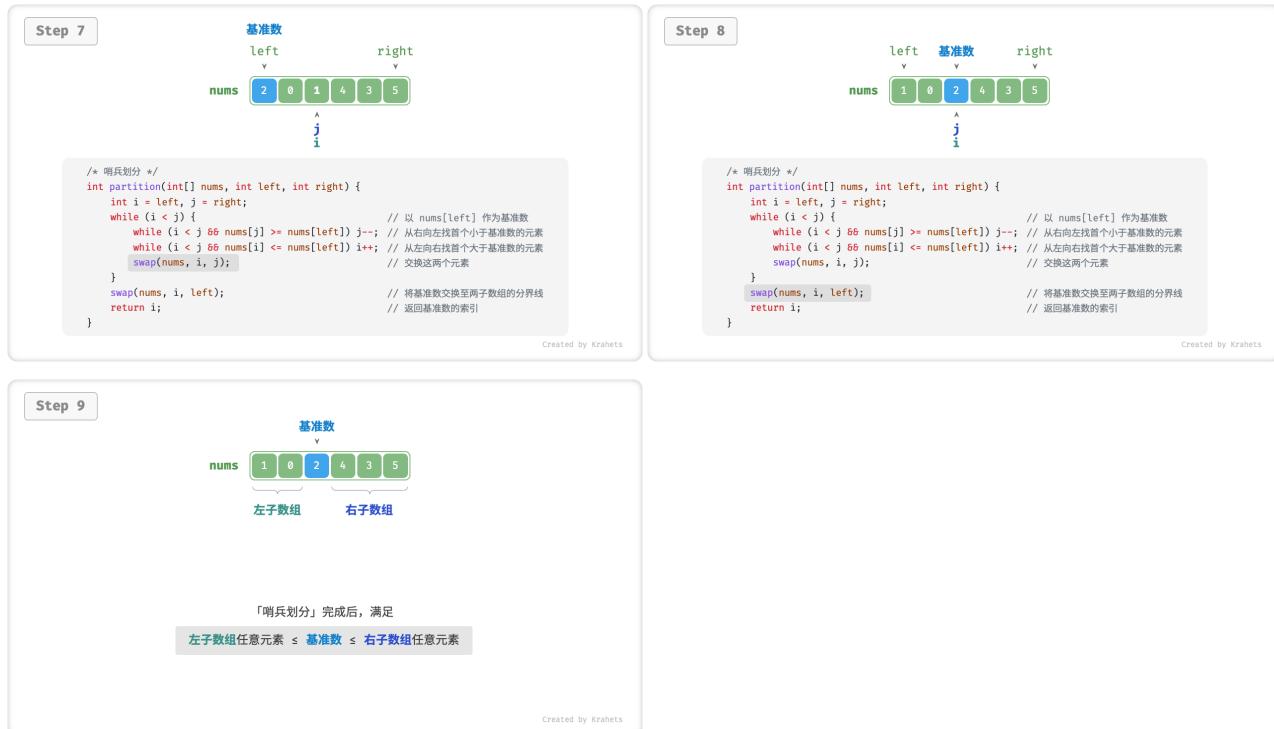


Figure 11-6. 哨兵划分步骤



快速排序的分治思想

哨兵划分的实质是将一个长数组的排序问题 简化为 两个短数组的排序问题。

```
// === File: quick_sort.cpp ===
/* 元素交换 */
void swap(vector<int>& nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}

/* 哨兵划分 */
int partition(vector<int>& nums, int left, int right) {
    // 以 nums[left] 作为基准数
    int i = left, j = right;
    while (i < j) {
        while (i < j && nums[j] >= nums[left])
            j--; // 从右向左找首个小于基准数的元素
        while (i < j && nums[i] <= nums[left])
            i++; // 从左向右找首个大于基准数的元素
        swap(nums, i, j);
    }
    swap(nums, i, left);
    return i;
}
```

```

    swap(nums, i, j); // 交换这两个元素
}
swap(nums, i, left); // 将基准数交换至两子数组的分界线
return i;           // 返回基准数的索引
}

```

11.4.1. 算法流程

- 首先，对数组执行一次「哨兵划分」，得到待排序的 **左子数组** 和 **右子数组**；
- 接下来，对 **左子数组** 和 **右子数组** 分别 **递归执行**「哨兵划分」……
- 直至子数组长度为 1 时 **终止递归**，即可完成对整个数组的排序；

观察发现，快速排序和「二分查找」的原理类似，都是以对数阶的时间复杂度来缩小处理区间。



Figure 11-7. 快速排序流程

```

// === File: quick_sort.cpp ===
/* 快速排序 */
void quickSort(vector<int>& nums, int left, int right) {
    // 子数组长度为 1 时终止递归
    if (left >= right)
        return;
    // 哨兵划分
    int pivot = partition(nums, left, right);
    // 递归左子数组、右子数组
    quickSort(nums, left, pivot - 1);
    quickSort(nums, pivot + 1, right);
}

```

11.4.2. 算法特性

平均时间复杂度 $O(n \log n)$ ：平均情况下，哨兵划分的递归层数为 $\log n$ ，每层中的总循环数为 n ，总体使用 $O(n \log n)$ 时间。

最差时间复杂度 $O(n^2)$ ：最差情况下，哨兵划分操作将长度为 n 的数组划分为长度为 0 和 $n - 1$ 的两个子数组，此时递归层数达到 n 层，每层中的循环数为 n ，总体使用 $O(n^2)$ 时间。

空间复杂度 $O(n)$ ：输入数组完全倒序下，达到最差递归深度 n 。

原地排序：只在递归中使用 $O(\log n)$ 大小的栈帧空间。

非稳定排序：哨兵划分操作可能改变相等元素的相对位置。

自适应排序：最差情况下，时间复杂度劣化至 $O(n^2)$ 。

11.4.3. 快排为什么快？

从命名能够看出，快速排序在效率方面一定“有两把刷子”。快速排序的平均时间复杂度虽然与「归并排序」和「堆排序」一致，但实际 **效率更高**，这是因为：

- **出现最差情况的概率很低**：虽然快速排序的最差时间复杂度为 $O(n^2)$ ，不如归并排序，但绝大部分情况下，快速排序可以达到 $O(n \log n)$ 的复杂度。
- **缓存使用效率高**：哨兵划分操作时，将整个子数组加载入缓存中，访问元素效率很高。而诸如「堆排序」需要跳跃式访问元素，因此不具有此特性。
- **复杂度的常数系数低**：在提及的三种算法中，快速排序的 **比较、赋值、交换** 三种操作的总体数量最少（类似于「插入排序」快于「冒泡排序」的原因）。

11.4.4. 基准数优化

普通快速排序在某些输入下的时间效率变差。举个极端例子，假设输入数组是完全倒序的，由于我们选取最左端元素为基准数，那么在哨兵划分完成后，基准数被交换至数组最右端，从而**左子数组长度为 $n - 1$ 、右子数组长度为 0**。这样进一步递归下去，**每轮哨兵划分后的右子数组长度都为 0**，分治策略失效，快速排序退化为「冒泡排序」了。

为了尽量避免这种情况发生，我们可以优化一下基准数的选取策略。首先，在哨兵划分中，我们可以**随机选取一个元素作为基准数**。但如果运气很差，每次都选择到比较差的基准数，那么效率依然不好。

进一步地，我们可以在数组中选取 3 个候选元素（一般为数组的首、尾、中点元素），**并将三个候选元素的中位数作为基准数**，这样基准数“既不大也不小”的概率就大大提升了。当然，如果数组很长的话，我们也可以选取更多候选元素，来进一步提升算法的稳健性。采取该方法后，时间复杂度劣化至 $O(n^2)$ 的概率极低。

```
// === File: quick_sort.cpp ===
/* 选取三个元素的中位数 */
int medianThree(vector<int>& nums, int left, int mid, int right) {
    // 此处使用异或运算来简化代码
    // 异或规则为 0 ^ 0 = 1 ^ 1 = 0, 0 ^ 1 = 1 ^ 0 = 1
```

```

if ((nums[left] < nums[mid]) ^ (nums[left] < nums[right]))
    return left;
else if ((nums[mid] < nums[left]) ^ (nums[mid] < nums[right]))
    return mid;
else
    return right;
}

/* 哨兵划分（三数取中值） */
int partition(vector<int>& nums, int left, int right) {
    // 选取三个候选元素的中位数
    int med = medianThree(nums, left, (left + right) / 2, right);
    // 将中位数交换至数组最左端
    swap(nums, left, med);
    // 以 nums[left] 作为基准数
    int i = left, j = right;
    while (i < j) {
        while (i < j && nums[j] >= nums[left])
            j--;           // 从右向左找首个小于基准数的元素
        while (i < j && nums[i] <= nums[left])
            i++;           // 从左向右找首个大于基准数的元素
        swap(nums, i, j); // 交换这两个元素
    }
    swap(nums, i, left); // 将基准数交换至两子数组的分界线
    return i;             // 返回基准数的索引
}

```

11.4.5. 尾递归优化

普通快速排序在某些输入下的空间效率变差。仍然以完全倒序的输入数组为例，由于每轮哨兵划分后右子数组长度为 0，那么将形成一个高度为 $n - 1$ 的递归树，此时使用的栈帧空间大小劣化至 $O(n)$ 。

为了避免栈帧空间的累积，我们可以在每轮哨兵排序完成后，判断两个子数组的长度大小，仅递归排序较短的子数组。由于较短的子数组长度不会超过 $\frac{n}{2}$ ，因此这样做能保证递归深度不超过 $\log n$ ，即最差空间复杂度被优化至 $O(\log n)$ 。

```

// === File: quick_sort.cpp ===
/* 快速排序（尾递归优化） */
void quickSort(vector<int>& nums, int left, int right) {
    // 子数组长度为 1 时终止
    while (left < right) {
        // 哨兵划分操作
        int pivot = partition(nums, left, right);
        // 对两个子数组中较短的那个执行快排
        if (pivot - left < right - pivot) {

```

```

        quickSort(nums, left, pivot - 1); // 递归排序左子数组
        left = pivot + 1; // 剩余待排序区间为 [pivot + 1, right]
    } else {
        quickSort(nums, pivot + 1, right); // 递归排序右子数组
        right = pivot - 1; // 剩余待排序区间为 [left, pivot - 1]
    }
}
}
}

```

11.5. 归并排序

「归并排序 Merge Sort」是算法中“分治思想”的典型体现，其有「划分」和「合并」两个阶段：

1. 划分阶段：通过递归不断 **将数组从中点位置划分开**，将长数组的排序问题转化为短数组的排序问题；
2. 合并阶段：划分到子数组长度为 1 时，开始向上合并，不断将 **左、右两个短排序数组 合并为一个长排序数组**，直至合并至原数组时完成排序；

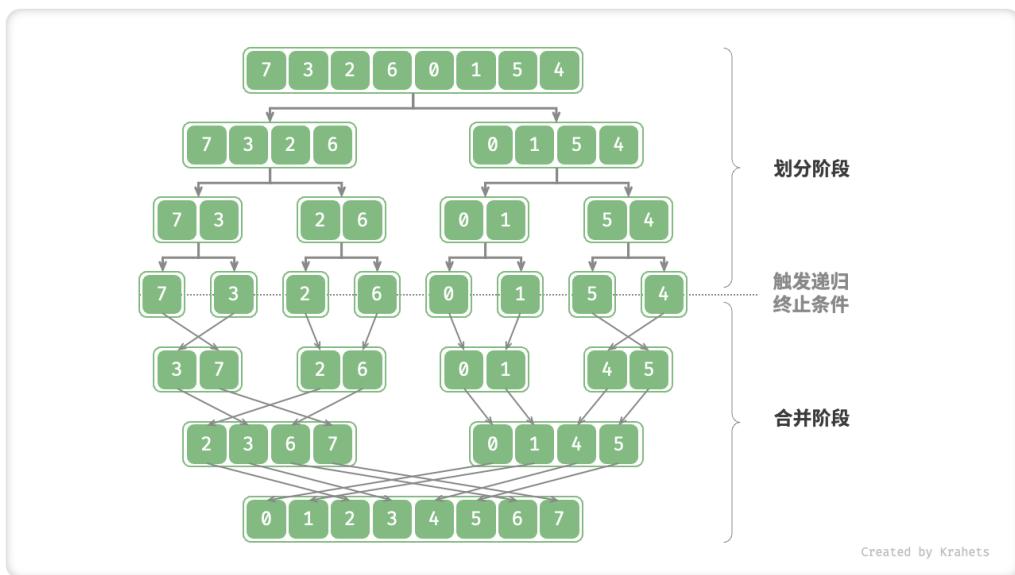


Figure 11-8. 归并排序的划分与合并阶段

11.5.1. 算法流程

「递归划分」从顶至底递归地 **将数组从中点切为两个子数组**，直至长度为 1；

1. 计算数组中点 `mid`，递归划分左子数组（区间 `[left, mid]`）和右子数组（区间 `[mid + 1, right]`）；
2. 递归执行 1. 步骤，直至子数组区间长度为 1 时，终止递归划分；

「回溯合并」从底至顶地将左子数组和右子数组合并为一个 **有序数组**；

需要注意，由于从长度为 1 的子数组开始合并，所以 **每个子数组都是有序的**。因此，合并任务本质是要 **将两个有序子数组合并为一个有序数组**。





Figure 11-9. 归并排序步骤

观察发现，归并排序的递归顺序就是二叉树的「后序遍历」。

- **后序遍历**: 先递归左子树、再递归右子树、最后处理根结点。
- **归并排序**: 先递归左子树、再递归右子树、最后处理合并。

```
// === File: merge_sort.cpp ===
/* 合并左子数组和右子数组 */
// 左子数组区间 [left, mid]
// 右子数组区间 [mid + 1, right]
void merge(vector<int>& nums, int left, int mid, int right) {
    // 初始化辅助数组
    vector<int> tmp(nums.begin() + left, nums.begin() + right + 1);
    // 左子数组的起始索引和结束索引
    int leftStart = left - left, leftEnd = mid - left;
    // 右子数组的起始索引和结束索引
    int rightStart = mid + 1 - left, rightEnd = right - left;
    // i, j 分别指向左子数组、右子数组的首元素
    int i = leftStart, j = rightStart;
    // 通过覆盖原数组 nums 来合并左子数组和右子数组
    for (int k = left; k <= right; k++) {
        // 若“左子数组已全部合并完”，则选取右子数组元素，并且 j++
        if (i > leftEnd)
            nums[k] = tmp[j++];
        // 否则，若“右子数组已全部合并完”或“左子数组元素 <= 右子数组元素”，则选取左子数组元素，并且 i++
        else if (j > rightEnd || tmp[i] <= tmp[j])
            nums[k] = tmp[i++];
        // 否则，若“左右子数组都未全部合并完”且“左子数组元素 > 右子数组元素”，则选取右子数组元素，并且 j++
        else
            nums[k] = tmp[j++];
    }
}
/* 归并排序 */
```

```

void mergeSort(vector<int>& nums, int left, int right) {
    // 终止条件
    if (left >= right) return;           // 当子数组长度为 1 时终止递归
    // 划分阶段
    int mid = (left + right) / 2;       // 计算中点
    mergeSort(nums, left, mid);         // 递归左子数组
    mergeSort(nums, mid + 1, right);     // 递归右子数组
    // 合并阶段
    merge(nums, left, mid, right);
}

```

下面重点解释一下合并方法 `merge()` 的流程：

1. 初始化一个辅助数组 `tmp` 暂存待合并区间 $[left, right]$ 内的元素，后续通过覆盖原数组 `nums` 的元素来实现合并；
2. 初始化指针 `i`, `j`, `k` 分别指向左子数组、右子数组、原数组的首元素；
3. 循环判断 `tmp[i]` 和 `tmp[j]` 的大小，将较小的先覆盖至 `nums[k]`，指针 `i`, `j` 根据判断结果交替前进（指针 `k` 也前进），直至两个子数组都遍历完，即可完成合并。

合并方法 `merge()` 代码中的主要难点：

- `nums` 的待合并区间为 $[left, right]$ ，而因为 `tmp` 只复制了 `nums` 该区间元素，所以 `tmp` 对应区间为 $[0, right - left]$ ，需要特别注意代码中各个变量的含义。
- 判断 `tmp[i]` 和 `tmp[j]` 的大小的操作中，还需考虑当子数组遍历完成后的索引越界问题，即 `i > leftEnd` 和 `j > rightEnd` 的情况，索引越界的优先级是最高的，例如如果左子数组已经被合并完了，那么不用继续判断，直接合并右子数组元素即可。

11.5.2. 算法特性

- **时间复杂度** $O(n \log n)$ ：划分形成高度为 $\log n$ 的递归树，每层合并的总操作数量为 n ，总体使用 $O(n \log n)$ 时间。
- **空间复杂度** $O(n)$ ：需借助辅助数组实现合并，使用 $O(n)$ 大小的额外空间；递归深度为 $\log n$ ，使用 $O(\log n)$ 大小的栈帧空间。
- **非原地排序**：辅助数组需要使用 $O(n)$ 额外空间。
- **稳定排序**：在合并时可保证相等元素的相对位置不变。
- **非自适应排序**：对于任意输入数据，归并排序的时间复杂度皆相同。

11.5.3. 链表排序 *

归并排序有一个很特别的优势，用于排序链表时有很好的性能表现，空间复杂度可被优化至 $O(1)$ ，这是因为：

- 由于链表可仅通过改变指针来实现结点增删，因此“将两个短有序链表合并为一个长有序链表”无需使用额外空间，即回溯合并阶段不用像排序数组一样建立辅助数组 `tmp`；

- 通过使用「迭代」代替「递归划分」，可省去递归使用的栈帧空间；

详情参考：[148. 排序链表](#)

11.6. 小结

- 冒泡排序通过交换相邻元素来实现排序。通过增加标志位实现提前返回，我们可将冒泡排序的最佳时间复杂度优化至 $O(N)$ 。
- 插入排序每轮将待排序区间内元素插入至已排序区间的正确位置，从而实现排序。插入排序的时间复杂度虽为 $O(N^2)$ ，但因为总体操作少而很受欢迎，一般用于小数据量的排序工作。
- 快速排序基于哨兵划分操作实现排序。在哨兵划分中，有可能每次都选取到最差的基准数，从而导致时间复杂度劣化至 $O(N^2)$ ，通过引入中位数基准数或随机基准数可大大降低劣化概率。尾递归方法可以有效减小递归深度，将空间复杂度优化至 $O(\log N)$ 。
- 归并排序包含划分和合并两个阶段，是分而治之的标准体现。对于归并排序，排序数组需要借助辅助数组，空间复杂度为 $O(N)$ ；而排序链表的空间复杂度可以被优化至 $O(1)$ 。
- 下图总结对比了各个排序算法的运行效率与特性。其中，桶排序中 k 为桶的数量；基数排序仅适用于正整数、字符串、特定格式的浮点数， k 为最大数字的位数。

	时间复杂度				空间复杂度		稳定性	就地性	自适应性	比较类
	最佳	平均	最差	最差	稳定性	就地性				
冒泡排序	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	稳定	原地	自适应	比较		
插入排序	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$	稳定	原地	自适应	比较		
选择排序	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$	非稳定	原地	非自适应	比较		
快速排序	$O(N \log N)$	$O(N \log N)$	$O(N^2)$	$O(\log N)$	非稳定	原地	自适应	比较		
归并排序	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(N)$	稳定	非原地	非自适应	比较		
堆排序	$O(N \log N)$	$O(N \log N)$	$O(N \log N)$	$O(1)$	非稳定	原地	非自适应	比较		
基数排序	$O(N k)$	$O(N k)$	$O(N k)$	$O(N + k)$	稳定	非原地	非自适应	非比较		
桶排序	$O(N + k)$	$O(N + k)$	$O(N^2)$	$O(N)$	稳定	非原地	自适应	非比较		

差 中 优

Created by Krahets

Figure 11-10. 排序算法对比

- 总体来看，我们追求运行快、稳定、原地、正向自适应性的排序。显然，如同其它数据结构与算法一样，同时满足这些条件的排序算法并不存在，我们需要根据问题特点来选择排序算法。

12. 附录

12.1. 编程环境安装

12.1.1. 安装 VSCode

本书推荐使用开源轻量的 VSCode 作为本地 IDE，下载并安装 [VSCode](#)。

12.1.2. Java 环境

1. 下载并安装 [OpenJDK](#) (版本需满足 > JDK 9)。
2. 在 VSCode 的插件市场中搜索 `java`，安装 Java Extension Pack。

12.1.3. C/C++ 环境

1. Windows 系统需要安装 [MinGW](#) ([配置教程](#))，MacOS 自带 Clang 无需安装。
2. 在 VSCode 的插件市场中搜索 `c++`，安装 C/C++ Extension Pack。

12.1.4. Python 环境

1. 下载并安装 [Miniconda3](#)。
2. 在 VSCode 的插件市场中搜索 `python`，安装 Python Extension Pack。

12.1.5. Go 环境

1. 下载并安装 `go`。
2. 在 VSCode 的插件市场中搜索 `go`，安装 Go。
3. 快捷键 `Ctrl + Shift + P` 呼出命令栏，输入 `go`，选择 `Go: Install/Update Tools`，全部勾选并安装即可。

12.1.6. JavaScript 环境

1. 下载并安装 [node.js](#)。
2. 在 VSCode 的插件市场中搜索 `javascript`，安装 JavaScript (ES6) code snippets。

12.1.7. C# 环境

1. 下载并安装 [.Net 6.0](#)；
2. 在 VSCode 的插件市场中搜索 `c#`，安装 `c#`。

12.1.8. Swift 环境

1. 下载并安装 [Swift](#)；
2. 在 VSCode 的插件市场中搜索 `swift`，安装 [Swift for Visual Studio Code](#)。

12.1.9. Rust 环境

1. 下载并安装 [Rust](#)；
2. 在 VSCode 的插件市场中搜索 `rust`，安装 [rust-analyzer](#)。

12.2. 一起参与创作



开源的魅力

纸质书籍的两次印刷的间隔时间往往需要数年，内容更新非常不方便。但在本开源 HTML 书中，内容更迭的时间被缩短至数日甚至几个小时。

由于作者水平有限，书中内容难免疏漏谬误，请您谅解。如果发现笔误、无效链接、内容缺失、文字歧义、解释不清晰、行文结构不合理等问题，请您帮忙修正，以帮助其他读者获取更优质的学习内容。所有[撰稿人](#)将被展示在仓库与网站主页，以感谢他们对开源社区的无私奉献！

12.2.1. 内容微调

每个页面的右上角都有一个「编辑」图标，你可以按照以下步骤修改文字或代码：

1. 点击编辑按钮，如果遇到提示“需要 Fork 此仓库”，请通过；
2. 修改 Markdown 源文件内容，并检查内容正确性，尽量保持排版格式统一；
3. 在页面底部填写更改说明，然后单击“Propose file change”按钮；页面跳转后，点击“Create pull request”按钮发起拉取请求即可。



Figure 12-1. 页面编辑按键

图片无法直接修改，需要通过新建 Issue 或评论留言来描述图片问题，我会第一时间重新画图并替换图片。

12.2.2. 内容创作

如果您想要参与本开源项目，包括翻译代码至其他编程语言、拓展文章内容等，那么需要实施 Pull Request 工作流程：

1. 登录 GitHub，并 Fork [本仓库](#) 至个人账号；
2. 进入 Fork 仓库网页，使用 `git clone` 克隆该仓库至本地；
3. 在本地进行内容创作，并通过运行测试来验证代码正确性；
4. 将本地更改 Commit，并 Push 至远程仓库；
5. 刷新仓库网页，点击“Create pull request”按钮发起拉取请求即可；

12.2.3. Docker 部署

你可以使用 Docker 来部署本项目。稍等片刻，即可使用浏览器打开 <http://localhost:8000> 访问本项目。

```
git clone https://github.com/krahets/hello-algo.git
cd hello-algo
docker-compose up -d
```

使用以下命令即可删除部署。

```
docker-compose down
```