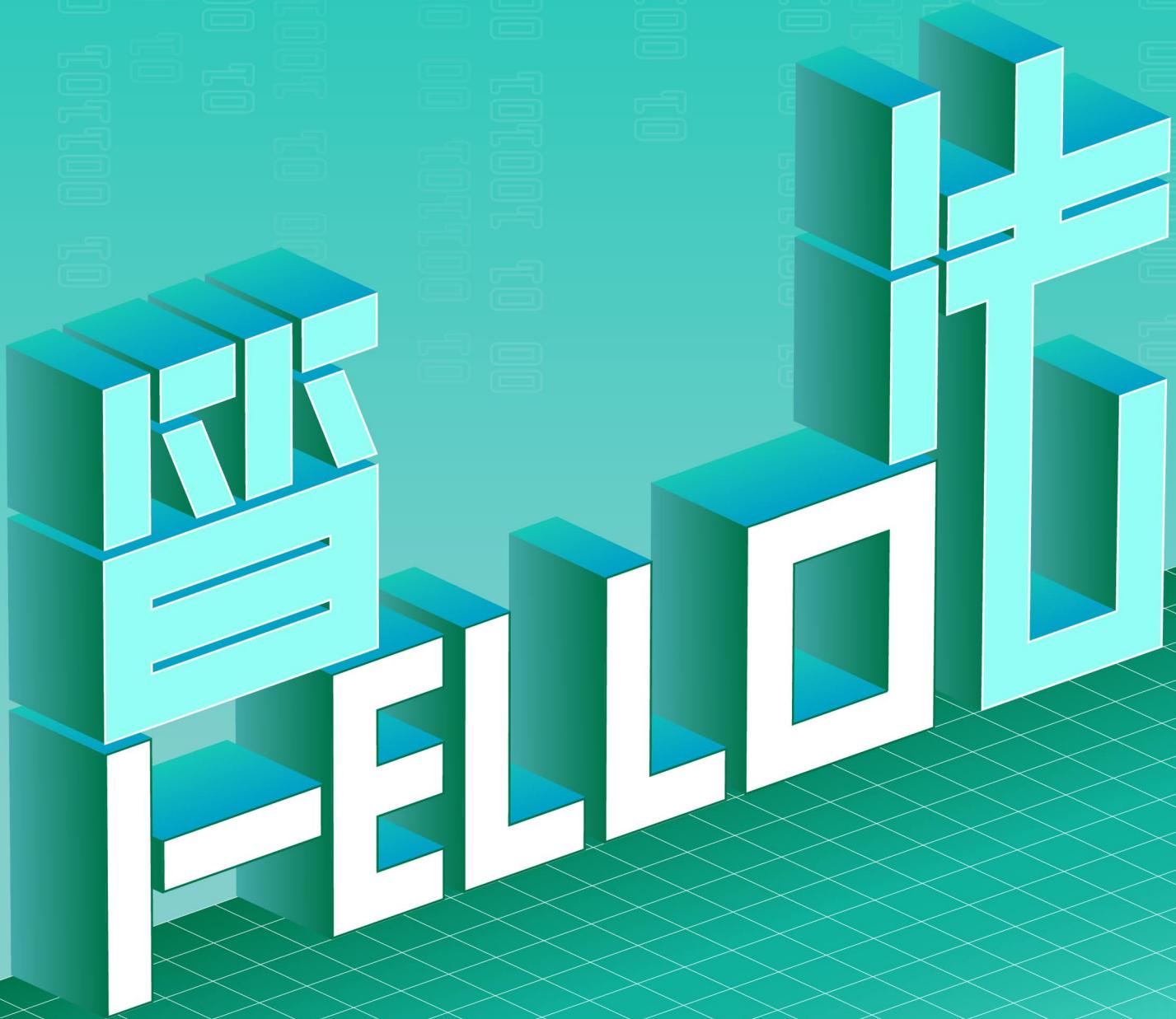


I-ELLO I-ELLO I-ELLO I-ELLO I-ELLO I-ELLO I-ELLO

# > 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

Python 语言版

靳宇栋 (Krahets)



Release 1.0.0b3  
2023-05-10

---

## 序

两年前，我在力扣上分享了《剑指 Offer》系列题解，受到了许多朋友的喜爱与支持。在此期间，我回答了众多读者的评论问题，其中最常见的一个问题是“如何入门学习算法”。我逐渐也对这个问题产生了浓厚的兴趣。

两眼一抹黑地刷题似乎是最受欢迎的方法，简单直接且有效。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，并在挫折中步步退缩。通读教材书籍也是一种常见做法，但对于面向求职的同学来说，毕业季、投递简历、准备笔试面试已经占据了大部分精力，厚重的书籍往往变成了一项艰巨的挑战。

如果你也面临类似的困扰，那么很幸运这本书找到了你。本书是我对此问题给出的答案，虽然不一定正确，但至少是一次积极的尝试。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，相信你可以更加自如地应对刷题和阅读文献，逐步构建起完整的知识体系。

本书中的代码附有可一键运行的源文件，托管于 [github.com/krahets/hello-algo](https://github.com/krahets/hello-algo) 仓库。动画在 PDF 内的展示效果受限，可访问 [hello-algo.com](https://hello-algo.com) 网页版以获得更优的阅读体验。

## 致谢

本书在开源社区众多贡献者的共同努力下不断成长。感谢每一位投入时间与精力的撰稿人，是他们无私奉献使这本书越变越好，他们是（按照 GitHub 自动生成的顺序）：krahets, justin-tse, sjinzh, Reanon, Gonglja, nuomi1, S-N-O-R-L-A-X, danielsss, RiverTwilight, msk397, gyt95, zhuoqinyue, FangYuan33, mingXta, Xia-Sang, hpstory, guowei-gong, GN-Yu, JoseHung, IsChristina, pengchzn, qualifier1024, Guanngxu, xBLACKICEx, Cathay-Chen, what-is-me, L-Super, Slone123c, mgisr, longranger2, xiongsp, gvenusleo, WSL0809, Wonderdch, a16su, JeffersonHuang, xjr7670, MolDuM, XC-Zero, DullSword, iron-irax, yi427, boloboloda, huawuque404, 4yDX3906, ZJKung, xb534, siqyka, ZnYang2018, beintentional, luluxia, GaochaoZhu, weibk, dshlstarr, ShiMaRing, fbigm, dyizheng, iStig, YuelinXin, szu17dmy, hezhizhen, fanchenggang, Keynman, youshaoXG, lipusheng, Javesun99, tao363, czruby, gltianwen, liuxjerry, yabo083。

本书的代码审阅工作由 Gonglja, justin - tse, krahets, nuomi1, Reanon, sjinzh 完成（按照首字母顺序排列）。感谢他们付出的时间与精力，正是他们确保了各语言代码的规范与统一。

## 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

<b>0. 写在前面</b>	<b>1</b>
0.1. 关于本书 . . . . .	1
0.2. 如何使用本书 . . . . .	3
0.3. 小结 . . . . .	7
<b>1. 引言</b>	<b>8</b>
1.1. 算法无处不在 . . . . .	8
1.2. 算法是什么 . . . . .	9
1.3. 小结 . . . . .	11
<b>2. 复杂度分析</b>	<b>12</b>
2.1. 算法效率评估 . . . . .	12
2.2. 时间复杂度 . . . . .	13
2.3. 空间复杂度 . . . . .	26
2.4. 小结 . . . . .	32
<b>3. 数据结构简介</b>	<b>34</b>
3.1. 数据与内存 . . . . .	34
3.2. 数据结构分类 . . . . .	38
3.3. 小结 . . . . .	39
<b>4. 数组与链表</b>	<b>40</b>
4.1. 数组 . . . . .	40
4.2. 链表 . . . . .	44
4.3. 列表 . . . . .	49
4.4. 小结 . . . . .	53
<b>5. 栈与队列</b>	<b>55</b>
5.1. 栈 . . . . .	55
5.2. 队列 . . . . .	61
5.3. 双向队列 . . . . .	67
5.4. 小结 . . . . .	75
<b>6. 二分查找</b>	<b>76</b>
6.1. 二分查找 . . . . .	76
<b>7. 散列表</b>	<b>80</b>
7.1. 哈希表 . . . . .	80
7.2. 哈希冲突 . . . . .	85
7.3. 小结 . . . . .	88
<b>8. 树</b>	<b>89</b>
8.1. 二叉树 . . . . .	89
8.2. 二叉树遍历 . . . . .	95
8.3. 二叉树数组表示 . . . . .	100
8.4. 二叉搜索树 . . . . .	102
8.5. AVL 树 * . . . . .	110
8.6. 小结 . . . . .	120
<b>9. 堆</b>	<b>122</b>
9.1. 堆 . . . . .	122

9.2. 建堆操作 *	129
9.3. 小结	131
<b>10. 图</b>	<b>132</b>
10.1. 图	132
10.2. 图基础操作	136
10.3. 图的遍历	142
10.4. 小结	149
<b>11. 排序算法</b>	<b>151</b>
11.1. 排序算法	151
11.2. 冒泡排序	152
11.3. 插入排序	155
11.4. 快速排序	157
11.5. 归并排序	163
11.6. 桶排序	167
11.7. 计数排序	169
11.8. 基数排序	173
11.9. 小结	176
<b>12. 搜索算法</b>	<b>178</b>
12.1. 搜索算法	178
12.2. 哈希优化策略	180
12.3. 小结	182
<b>13. 回溯算法</b>	<b>184</b>
13.1. 回溯算法	184
13.2. 全排列问题	191
13.3. N 皇后问题	196
<b>14. 附录</b>	<b>201</b>
14.1. 编程环境安装	201
14.2. 一起参与创作	202

# 0. 写在前面

## 0.1. 关于本书

本项目旨在创建一本开源免费、新手友好的数据结构与算法入门教程。

- 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，提问与评论通常可在两日内得到回复；

### 0.1.1. 读者对象

若您是「算法初学者」，从未接触过算法，或者已经有一些刷题经验，对数据结构与算法有模糊的认识，在会与不会之间反复横跳，那么这本书正是为您量身定制！

如果您是「算法老手」，已经积累一定刷题量，熟悉大部分题型，那么本书可助您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。

若您是「算法专家」，我们期待收到您的宝贵建议，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

本书主要内容包括：

- **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：**常见基本数据类型，数据在内存中的存储形式、数据结构的分类方法。涉及数组、链表、栈、队列、散列表、树、堆、图等数据结构，内容包括定义、优缺点、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、分治算法等，内容涵盖定义、应用场景、优缺点、时空效率、实现方法、示例题目等。

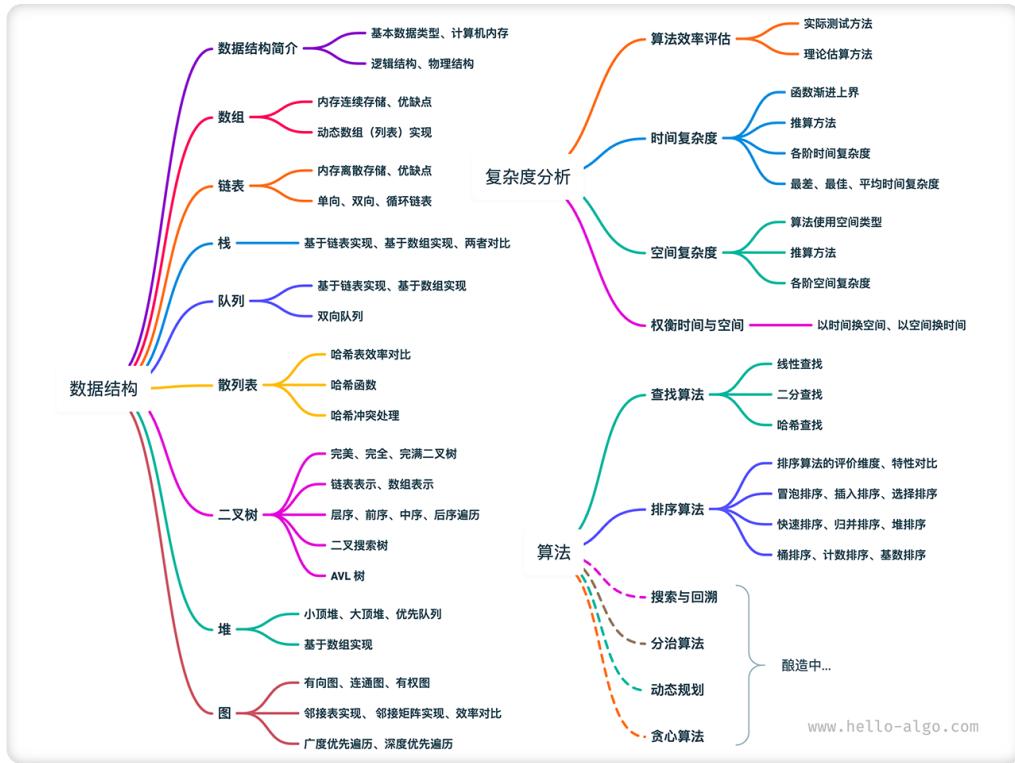


Figure 0-1. Hello 算法内容结构

### 0.1.3. 致谢

在本书的创作过程中，我得到了许多人的帮助，包括但不限于：

- 感谢我在公司的导师李汐博士，在深入交谈中您鼓励我“行动起来”，坚定了我写这本书的决心。
- 感谢我的女朋友泡泡作为本书的首位读者，从算法小白的角度提出许多宝贵建议，使得本书更适合新手阅读。
- 感谢腾宝、琦宝、飞宝为本书起了一个富有创意的名字，唤起大家写下第一行代码“Hello World!”的美好回忆。
- 感谢苏潼为本书设计了精美的封面和 LOGO，并在我的强迫症下多次耐心修改。
- 感谢 @squidfunk 提供的写作排版建议，以及杰出的开源项目 [Material-for-MkDocs](#)。

在写作过程中，我阅读了许多关于数据结构与算法的教材和文章。这些作品为本书提供了优秀的范本，确保了本书内容的准确性与品质。在此感谢所有老师和前辈们的杰出贡献！

本书倡导“手脑并用”的学习方法，在这方面深受《动手学深度学习》的启发。在此向各位读者强烈推荐这本优秀著作，包括[中文版](#)、[英文版](#)、[李沐老师 bilibili 主页](#)。

衷心感谢我的父母，正是你们一直以来的支持与鼓励，让我有机会做这些富有趣味的事。

## 0.2. 如何使用本书



为了获得最佳的阅读体验，建议您通读本节内容。

### 0.2.1. 算法学习路线

从总体上看，我们可以将学习数据结构与算法的过程划分为三个阶段：

1. **算法入门**。我们需要熟悉各种数据结构的特点和用法，学习不同算法的原理、流程、用途和效率等方面内容。
2. **刷算法题**。建议从热门题目开刷，如剑指 Offer 和 LeetCode Hot 100，先积累至少 100 道题目，熟悉主流的算法问题。初次刷题时，“知识遗忘”可能是一个挑战，但请放心，这是很正常的。我们可以按照“艾宾浩斯遗忘曲线”来复习题目，通常在进行 3-5 轮的重复后，就能将其牢记在心。
3. **搭建知识体系**。在学习方面，我们可以阅读算法专栏文章、解题框架和算法教材，以不断丰富知识体系。在刷题方面，可以尝试采用进阶刷题策略，如按专题分类、一题多解、一解多题等，相关的刷题心得可以在各个社区找到。

作为一本入门教程，本书内容主要涵盖“第一阶段”，旨在帮助你更高效地展开第二和第三阶段的学习。

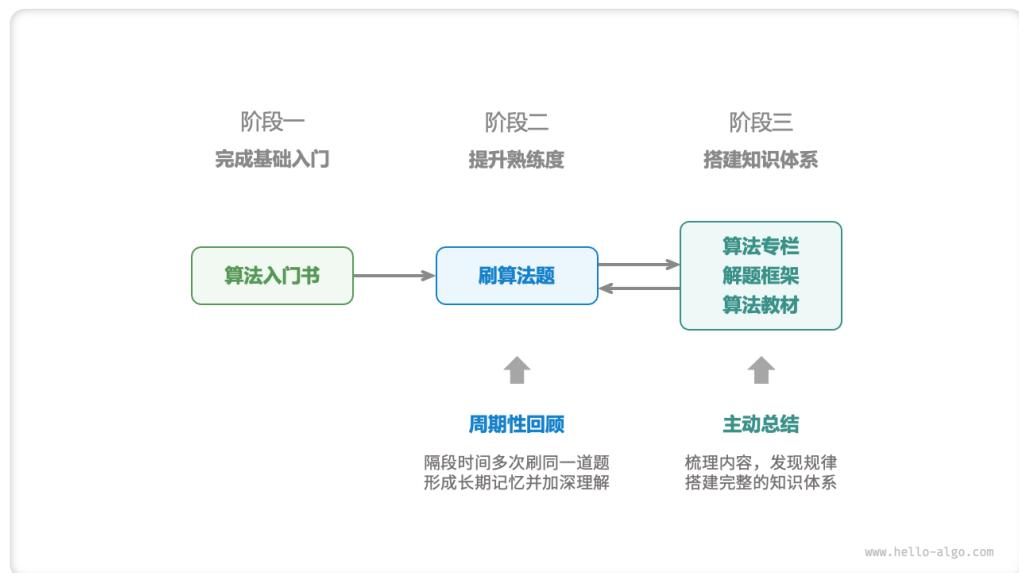


Figure 0-2. 算法学习路线

### 0.2.2. 行文风格约定

标题后标注 \* 是的选读章节，内容相对困难。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」。请务必记住这些名词，包括英文翻译，以便后续阅读文献时使用。

**加粗的文字** 表示重点内容或总结性语句，这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

相较于文字，视频和图片具有更高的信息密度和结构化程度，因此更易于理解。在本书中，**重点和难点知识将主要通过动画和图解形式展示**，而文字则作为动画和图片的解释与补充。

在阅读本书时，如果发现某段内容提供了动画或图解，建议以图为主线，以文字（通常位于图像上方）为辅，综合两者来理解内容。



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，并附有测试样例，可直接运行。

如果学习时间有限，建议你至少通读并运行所有代码。如果时间充裕，建议参照代码自行敲一遍。与仅阅读代码相比，编写代码的过程往往能带来更多收获。

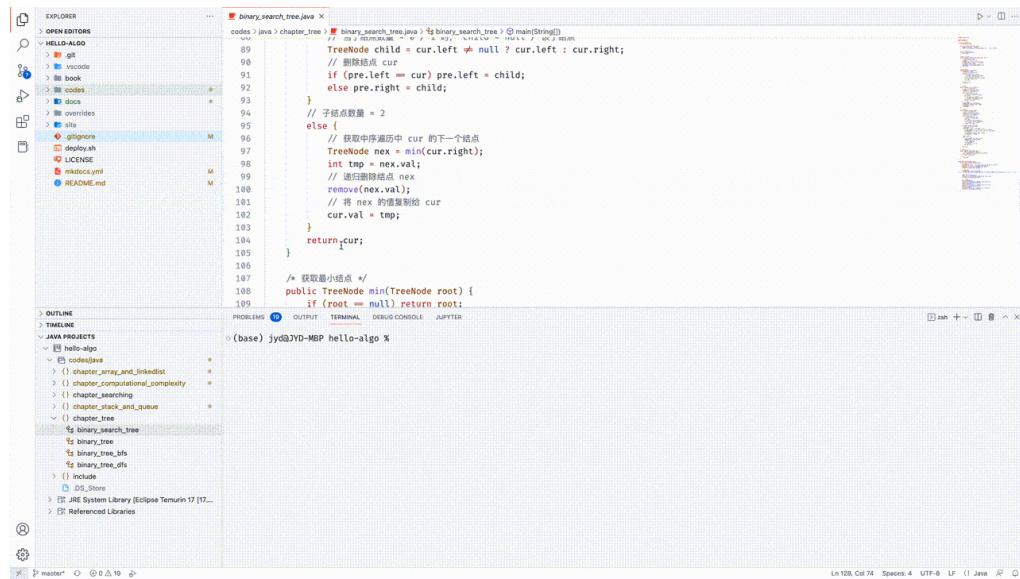


Figure 0-4. 运行代码示例

**第一步：安装本地编程环境。**请参照[附录教程](#)进行安装，如果已安装则可跳过此步骤。

**第二步：下载代码仓。**如果已经安装 Git，可以通过以下命令克隆本仓库。

```
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，然后在本地解压即可。

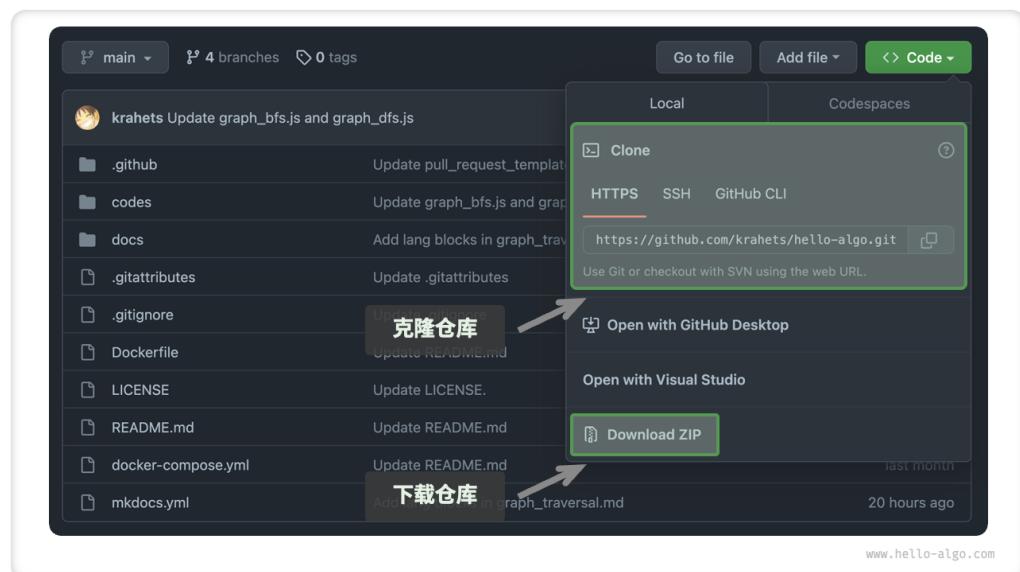


Figure 0-5. 克隆仓库与下载代码

**第三步：运行源代码。**如果代码块顶部标有文件名称，则可以在仓库的 **codes** 文件夹中找到相应的源代码文件。源代码文件将帮助你节省不必要的调试时间，让你能够专注于学习内容。



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”那些没学明白的知识点。**欢迎在评论区提出你的问题**，我和其他小伙伴们将竭诚为你解答，一般情况下可在两天内得到回复。

同时，也希望您能在评论区多花些时间。一方面，您可以了解大家遇到的问题，从而查漏补缺，这将有助于激发更深入的思考。另一方面，希望您能慷慨地回答其他小伙伴的问题、分享您的见解，让大家共同学习和进步。

Figure 0-7. 评论区示例

### 0.3. 小结

- 本书的主要受众是算法初学者。如果已有一定基础，本书能帮助您系统回顾算法知识，书内源代码也可作为“刷题工具库”使用。
- 书中内容主要包括复杂度分析、数据结构、算法三部分，涵盖了该领域的大部分主题。
- 对于算法新手，在初学阶段阅读一本入门书籍至关重要，可以少走许多弯路。
- 书内的动画和图解通常用于介绍重点和难点知识。阅读本书时，应给予这些内容更多关注。
- 实践乃学习编程之最佳途径。强烈建议运行源代码并亲自敲打代码。
- 本书网页版的每个章节都设有讨论区，欢迎随时分享你的疑惑与见解。

# 1. 引言

## 1.1. 算法无处不在

当我们听到“算法”这个词时，很自然地会想到数学。然而实际上，许多算法并不涉及复杂数学，而是更多地依赖于基本逻辑，这些逻辑在我们的日常生活中处处可见。

在正式探讨算法之前，有一个有趣的事事实值得分享：**实际上，你已经学会了许多算法，并习惯将他们应用到日常生活中了。**下面，我将举两个具体例子来证实这一点。

**例一：组装积木。**一套积木，除了包含许多零件之外，还附有详细的组装说明书。我们按照说明书一步步操作，就能组装出精美的积木模型。

从数据结构与算法的角度来看，积木的各种形状和连接方式代表数据结构，而组装说明书上的一系列步骤则是算法。

**例二：查阅字典。**在字典里，每个汉字都对应一个拼音，而字典是按照拼音的英文字母顺序排列的。假设我们需要查找一个拼音首字母为 *r* 的字，通常会这样操作：

1. 翻开字典约一半的页数，查看该页首字母是什么（假设为 *m*）；
2. 由于在英文字母表中 *r* 位于 *m* 之后，所以排除字典前半部分，查找范围缩小到后半部分；
3. 不断重复步骤 1-2，直至找到拼音首字母为 *r* 的页码为止。

The diagram illustrates a binary search process for finding a character with a specific pinyin initial ('r') in a dictionary. It consists of four panels:

- Step 1:** Shows a horizontal array of letters from 'a' to 'z'. An arrow points to the middle, labeled "一半页码处" (halfway through the page). Below the array, a box contains: "字典按照拼音的英文字母表顺序排列" and "查字典目标：找到任意一个拼音首字母为 r 的字".
- Step 2:** Shows the array again, but with the first half (a-m) faded. An arrow points to the second half (n-z), labeled "一半页码处". Below the array, a box contains: "查字典步骤：" followed by steps 1 and 2.
- Step 3:** Shows the array with the second half faded. An arrow points to the middle of the remaining half, labeled "一半页码处". Below the array, a box contains: "查字典步骤：" followed by steps 1, 2, and 3.
- Step 4:** Shows the array with the first half of the second half faded. An arrow points to the middle of the remaining quarter, labeled "一半页码处". Below the array, a box contains: "查字典步骤：" followed by steps 1, 2, 3, and 4.

Each panel includes the URL [www.hello-algo.com](http://www.hello-algo.com) at the bottom right.



Figure 1-1. 查字典步骤

查阅字典这个小学生必备技能，实际上就是著名的「二分查找」。从数据结构的角度，我们可以把字典视为一个已排序的「数组」；从算法的角度，我们可以将上述查字典的一系列操作看作是「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能够通过编程将数据结构存储在内存中，同时编写代码调用 CPU 和 GPU 执行算法。这样一来，我们就能把生活中的问题转移到计算机上，以更高效的方式解决各种复杂问题。



阅读至此，如果你对数据结构、算法、数组和二分查找等概念仍感到一知半解，那么太好了！因为这正是本书存在的意义。接下来，这本书将一步步引导你深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，具有清晰的输入和输出定义。
- 解具有确定性，即给定相同的输入时，输出始终相同。
- 具有可行性，在有限步骤、时间和内存空间下可完成。

### 1.2.2. 数据结构定义

「数据结构 Data Structure」是计算机中组织和存储数据的方式。为了提高数据存储和操作性能，数据结构的设计目标包括：

- 空间占用尽量减少，节省计算机内存。
- 数据操作尽可能快速，涵盖数据访问、添加、删除、更新等。
- 提供简洁的数据表示和逻辑信息，以利于算法高效运行。

数据结构设计是一个充满权衡的过程，这意味着要在某方面取得优势，往往需要在另一方面作出妥协。例如，链表相较于数组，在数据添加和删除操作上更加便捷，但牺牲了数据访问速度；图相较于链表，提供了更丰富的逻辑信息，但需要占用更大的内存空间。

### 1.2.3. 数据结构与算法的关系

「数据结构」与「算法」高度相关且紧密结合，具体表现在：

- 数据结构是算法的基石。数据结构为算法提供了结构化存储的数据，以及用于操作数据的方法。
- 算法是数据结构发挥的舞台。数据结构本身仅存储数据信息，通过结合算法才能解决特定问题。
- 特定算法通常有对应最优的数据结构。算法通常可以基于不同的数据结构进行实现，但最终执行效率可能相差很大。

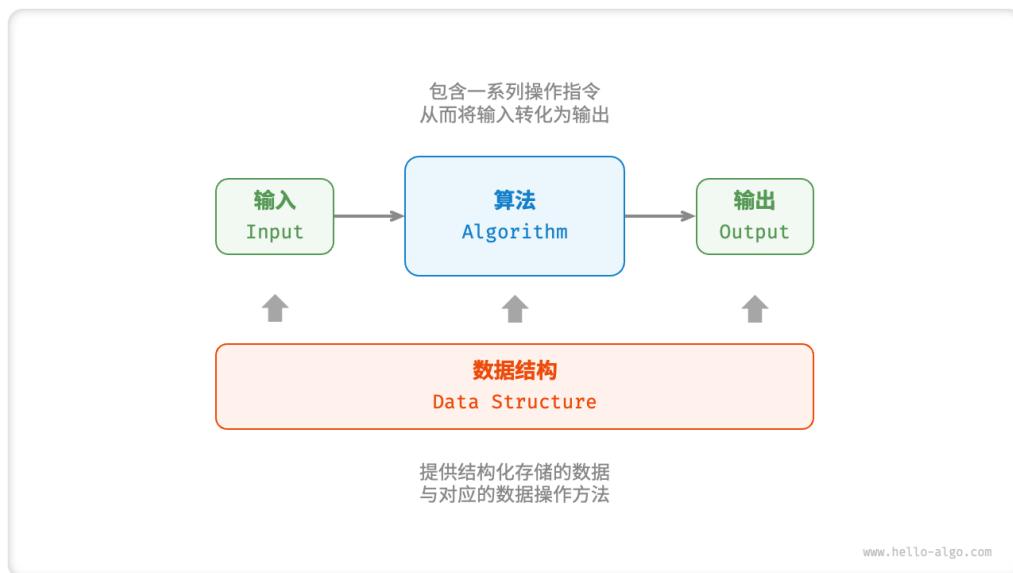


Figure 1-2. 数据结构与算法的关系

类比「LEGO 乐高」和「数据结构与算法」，则对应关系如下表所示。

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

值得注意的是，数据结构与算法独立于编程语言。正因如此，本书得以提供多种编程语言的实现。



### 约定俗成的简称

在实际讨论时，我们通常会将「数据结构与算法」简称为「算法」。例如，众所周知的 LeetCode 算法题目，实际上同时考察了数据结构和算法两方面的知识。

## 1.3. 小结

- 算法在日常生活中无处不在，并不是遥不可及的高深知识。实际上，我们已经在不知不觉中学习了许多“算法”，用以解决生活中的大小问题。
- 查阅字典的原理与二分查找算法相一致。二分查找体现了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，而数据结构是计算机中组织和存储数据的方式。
- 数据结构与算法紧密相连。数据结构是算法的基石，而算法则是发挥数据结构作用的舞台。
- 乐高积木对应于数据，积木形状和连接方式代表数据结构，拼装积木的步骤则对应算法。

## 2. 复杂度分析

### 2.1. 算法效率评估

#### 2.1.1. 算法评价维度

从总体上看，算法设计追求以下两个层面的目标：

1. 找到问题解法。算法需要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，我们希望找到尽可能高效的算法。

因此，在能够解决问题的前提下，算法效率成为主要的评价维度，主要包括：

- 时间效率，即算法运行速度的快慢。
- 空间效率，即算法占用内存空间的大小。

简而言之，我们的目标是设计“既快又省”的数据结构与算法。掌握评估算法效率的方法则至关重要，因为只有了解评价标准，我们才能进行算法之间的对比分析，从而指导算法设计与优化过程。

#### 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，它们都能解决同一问题，现在需要对比这两个算法的效率。我们最直接的方法就是找一台计算机，运行这两个算法，并监控记录它们的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，但也存在较大局限性。

**难以排除测试环境的干扰因素。**硬件配置会影响算法的性能表现。例如，在某台计算机中，算法 A 的运行时间比算法 B 短；但在另一台配置不同的计算机中，我们可能得到相反的测试结果。这意味着我们需要在各种机器上进行测试，而这是不现实的。

**展开完整测试非常耗费资源。**随着输入数据量的变化，算法会表现出不同的效率。例如，输入数据量较小时，算法 A 的运行时间可能短于算法 B；而输入数据量较大时，测试结果可能相反。因此，为了得到有说服力的结论，我们需要测试各种规模的输入数据，这样需要占用大量的计算资源。

##### 理论估算

由于实际测试具有较大的局限性，我们可以考虑仅通过一些计算来评估算法的效率。这种估算方法被称为「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。**这个定义有些拗口，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”，因此我们可以将复杂度分为「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”表示复杂度与输入数据量有关，反映了算法运行效率与输入数据量之间的关系；
- “增长趋势”表示复杂度分析关注的是算法时间与空间的增长趋势，而非具体的运行时间或占用空间；

复杂度分析克服了实际测试方法的弊端。首先，它独立于测试环境，因此分析结果适用于所有运行平台。其次，它可以体现不同数据量下的算法效率，尤其是在大数据量下的算法性能。

如果你对复杂度分析的概念仍感到困惑，无需担心，我们会在后续章节详细介绍。

### 2.1.3. 复杂度分析重要性

复杂度分析为我们提供了一把评估算法效率的“标尺”，告诉我们执行某个算法所需的时间和空间资源，并使我们能够对比不同算法之间的效率。

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。从这个角度看，复杂度分析可能不太适合作为第一章的内容。然而，当我们讨论某个数据结构或算法的特点时，我们难以避免要分析其运行速度和空间使用情况。因此，在深入学习数据结构与算法之前，建议读者先对复杂度建立初步的了解，并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

运行时间可以直观且准确地反映算法的效率。然而，如果我们想要准确预估一段代码的运行时间，应该如何操作呢？

1. 确定运行平台，包括硬件配置、编程语言、系统环境等，这些因素都会影响代码的运行效率。
2. 评估各种计算操作所需的运行时间，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. 统计代码中所有的计算操作，并将所有操作的执行时间求和，从而得到运行时间。

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
def algorithm(n: int) -> None:
    a = 2      # 1 ns
    a = a + 1 # 1 ns
    a = a * 2 # 10 ns
    # 循环 n 次
    for _ in range(n): # 1 ns
        print(0)       # 5 ns
```

然而实际上，**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，因为算法需要在各种不同的平台上运行。其次，我们很难获知每种操作的运行时间，这给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」采取了一种不同的方法，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”这个概念较为抽象，我们通过一个例子来加以理解。假设输入数据大小为  $n$ ，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  $n$  增大呈线性增长。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
def algorithm_A(n: int) -> None:
    print(0)

# 算法 B 时间复杂度：线性阶
def algorithm_B(n: int) -> None:
    for _ in range(n):
        print(0)

# 算法 C 时间复杂度：常数阶
def algorithm_C(n: int) -> None:
    for _ in range(1000000):
        print(0)
```

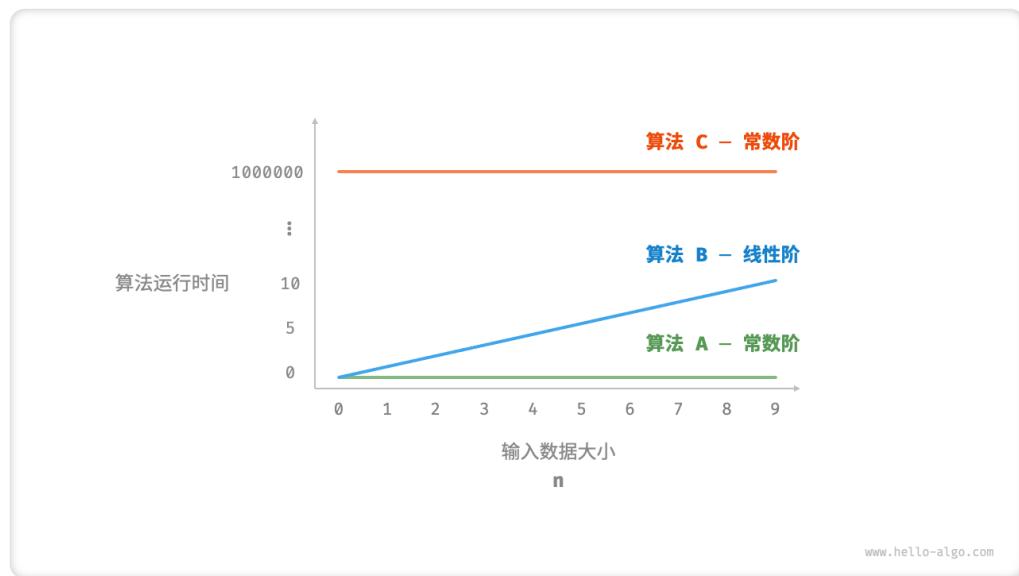


Figure 2-1. 算法 A, B, C 的时间增长趋势

相较于直接统计算法运行时间，时间复杂度分析有哪些优势和局限性呢？

**时间复杂度能够有效评估算法效率。**例如，算法 B 的运行时间呈线性增长，在  $n > 1$  时比算法 A 慢，在  $n > 1000000$  时比算法 C 慢。事实上，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，这正是时间增长趋势所表达的含义。

**时间复杂度的推算方法更简便。**显然，运行平台和计算操作类型都与算法运行时间的增长趋势无关。因此在时间复杂度分析中，我们可以简单地将所有计算操作的执行时间视为相同的“单位时间”，从而将“计算操作的运行时间的统计”简化为“计算操作的数量的统计”，这样的简化方法大大降低了估算难度。

**时间复杂度也存在一定的局限性。**例如，尽管算法 A 和 C 的时间复杂度相同，但实际运行时间差别很大。同样，尽管算法 B 的时间复杂度比 C 高，但在输入数据大小  $n$  较小时，算法 B 明显优于算法 C。在这些情况下，我们很难仅凭时间复杂度判断算法效率高低。当然，尽管存在上述问题，复杂度分析仍然是评判算法效率最有效且常用的方法。

### 2.2.3. 函数渐近上界

设算法的计算操作数量是一个关于输入数据大小  $n$  的函数，记为  $T(n)$ ，则以下算法的操作数量为

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

```
def algorithm(n: int) -> None:
    a: int = 1 # +1
    a = a + 1 # +1
    a = a * 2 # +1
    # 循环 n 次
    for i in range(n): # +1
        print(0) # +1
```

$T(n)$  是一次函数，说明时间增长趋势是线性的，因此可以得出时间复杂度是线性阶。

我们将线性阶的时间复杂度记为  $O(n)$ ，这个数学符号称为「大 O 记号 Big-O Notation」，表示函数  $T(n)$  的「渐近上界 Asymptotic Upper Bound」。

推算时间复杂度本质上是计算“操作数量函数  $T(n)$ ”的渐近上界。接下来，我们来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

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

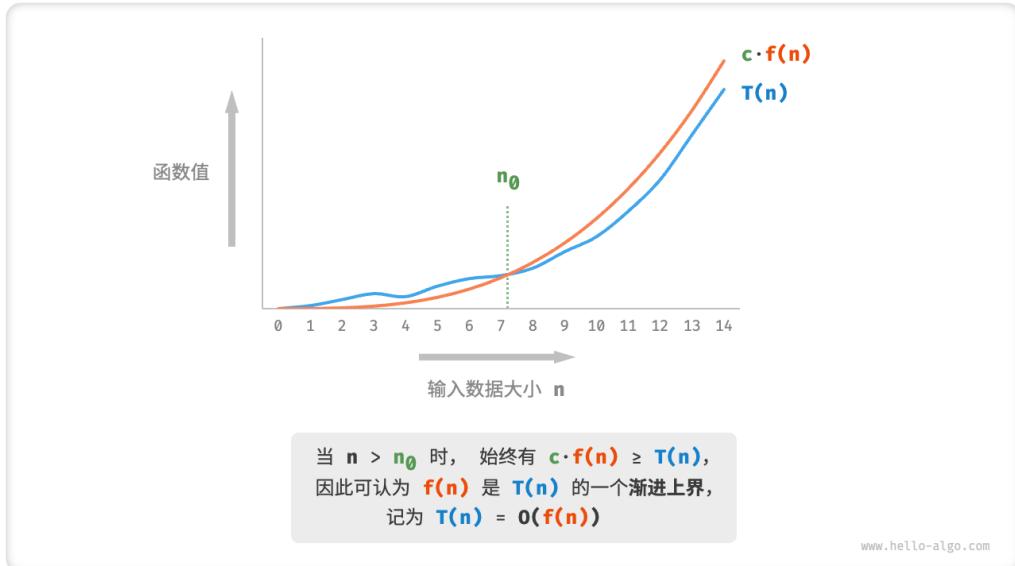


Figure 2-2. 函数的渐近上界

从本质上讲，计算渐近上界就是寻找一个函数  $f(n)$ ，使得当  $n$  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。

#### 2.2.4. 推算方法

渐近上界的数学味儿有点重，如果你感觉没有完全理解，也无需担心。因为在实际使用中，我们只需要掌握推算方法，数学意义可以逐渐领悟。

根据定义，确定  $f(n)$  之后，我们便可得到时间复杂度  $O(f(n))$ 。那么如何确定渐近上界  $f(n)$  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

针对代码，逐行从上到下计算即可。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简化技巧：

1. 忽略与  $n$  无关的操作。因为它们都是  $T(n)$  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、 $5n + 1$  次等，都可以简化记为  $n$  次，因为  $n$  前面的系数对时间复杂度没有影响。
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。

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

最终，两者都能推出相同的时间复杂度结果，即  $O(n^2)$ 。

```
def algorithm(n: int) -> None:
    a: int = 1 # +0 (技巧 1)
    a = a + n # +0 (技巧 1)
    # +n (技巧 2)
    for i in range(5 * n + 1):
        print(0)
    # +n*n (技巧 3)
    for i in range(2 * n):
        for j in range(n + 1):
            print(0)
```

## 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，最高阶的项将发挥主导作用，其他项的影响都可以被忽略。

以下表格展示了一些例子，其中一些夸张的值是为了强调“系数无法撼动阶数”这一结论。当  $n$  趋于无穷大时，这些常数变得无足轻重。

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  $n$ ，常见的复杂度类型包括（按照从低到高的顺序排列）：

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

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

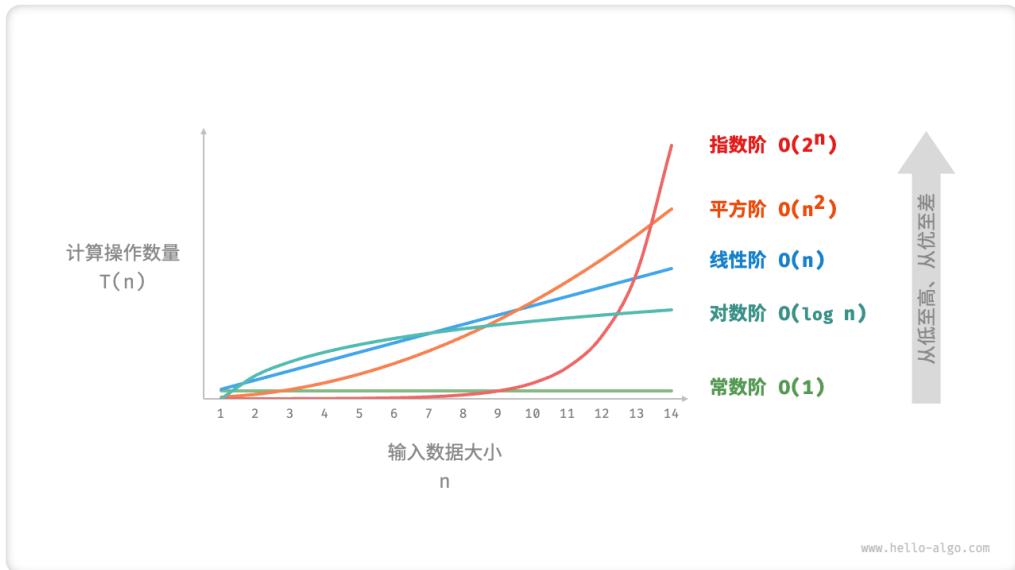


Figure 2-3. 时间复杂度的常见类型



部分示例代码需要一些预备知识，包括数组、递归算法等。如果遇到不理解的部分，请不要担心，可以在学习完后面章节后再回顾。现阶段，请先专注于理解时间复杂度的含义和推算方法。

### 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，尽管操作数量 `size` 可能很大，但由于其与数据大小  $n$  无关，因此时间复杂度仍为  $O(1)$ 。

```
# === File: time_complexity.py ===
def constant(n: int) -> int:
    """ 常数阶 """
    count: int = 0
    size: int = 100000
    for _ in range(size):
        count += 1
    return count
```

### 线性阶 $O(n)$

线性阶的操作数量相对于输入数据大小以线性级别增长。线性阶通常出现在单层循环中。

```
# === File: time_complexity.py ===
def linear(n: int) -> int:
    """ 线性阶 """
    count: int = 0
    for _ in range(n):
        count += 1
    return count
```

遍历数组和遍历链表等操作的时间复杂度均为  $O(n)$ ，其中  $n$  为数组或链表的长度。



### 如何确定输入数据大小 $n$ ？

数据大小  $n$  需根据输入数据的类型来具体确定。例如，在上述示例中，我们直接将  $n$  视为输入数据大小；在下面遍历数组的示例中，数据大小  $n$  为数组的长度。

```
# === File: time_complexity.py ===
def array_traversal(nums: list[int]) -> int:
    """ 线性阶（遍历数组） """
    count: int = 0
    # 循环次数与数组长度成正比
    for num in nums:
        count += 1
    return count
```

## 平方阶 $O(n^2)$

平方阶的操作数量相对于输入数据大小以平方级别增长。平方阶通常出现在嵌套循环中，外层循环和内层循环都为  $O(n)$ ，因此总体为  $O(n^2)$ 。

```
# === File: time_complexity.py ===
def quadratic(n: int) -> int:
    """ 平方阶 """
    count: int = 0
    # 循环次数与数组长度成平方关系
    for i in range(n):
        for j in range(n):
            count += 1
    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，外层循环执行  $n - 1$  次，内层循环执行  $n - 1, n - 2, \dots, 2, 1$  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.py ===
def bubble_sort(nums: list[int]) -> int:
    """ 平方阶（冒泡排序） """
    count: int = 0 # 计数器
    # 外循环：待排序元素数量为 n-1, n-2, ..., 1
    for i in range(len(nums) - 1, 0, -1):
        # 内循环：冒泡操作
        for j in range(i):
            if nums[j] > nums[j + 1]:
                # 交换 nums[j] 与 nums[j + 1]
                tmp: int = 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  $O(2^n)$**



生物学的“细胞分裂”是指数阶增长的典型例子：初始状态为 1 个细胞，分裂一轮后变为 2 个，分裂两轮后变为 4 个，以此类推，分裂  $n$  轮后有  $2^n$  个细胞。

指数阶增长非常迅速，在实际应用中通常是不可接受的。若一个问题使用「暴力枚举」求解的时间复杂度为  $O(2^n)$ ，那么通常需要使用「动态规划」或「贪心算法」等方法来解决。

```
# === File: time_complexity.py ===
def exponential(n: int) -> int:
    """ 指数阶（循环实现） """
    count: int = 0
    base: int = 1
    # cell 每轮一分为二，形成数列 1, 2, 4, 8, ..., 2^(n-1)
    for _ in range(n):
        for _ in range(base):
            count += 1
        base *= 2
    # count = 1 + 2 + 4 + 8 + ... + 2^(n-1) = 2^n - 1
    return count
```

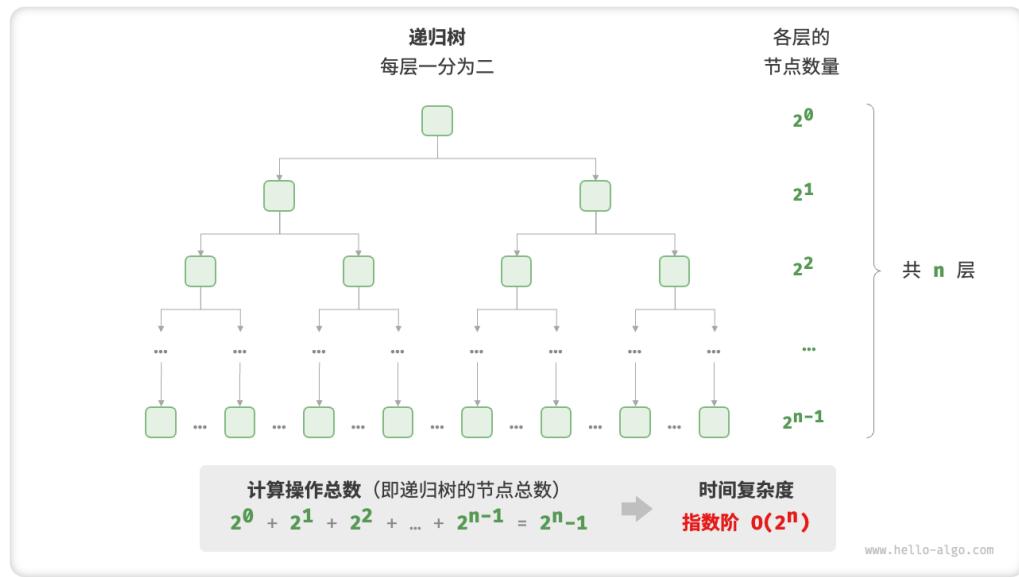


Figure 2-5. 指数阶的时间复杂度

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，经过  $n$  次分裂后停止。

```
# == File: time_complexity.py ==
def exp_recur(n: int) -> int:
    """ 指数阶 (递归实现) """
    if n == 1:
        return 1
    return exp_recur(n - 1) + exp_recur(n - 1) + 1
```

### 对数阶 $O(\log n)$

与指数阶相反，对数阶反映了“每轮缩减到一半的情况”。对数阶仅次于常数阶，时间增长缓慢，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，体现了“一分为多”和“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.py ==
def logarithmic(n: float) -> int:
    """ 对数阶 (循环实现) """
    count: int = 0
    while n > 1:
        n = n / 2
        count += 1
    return count
```

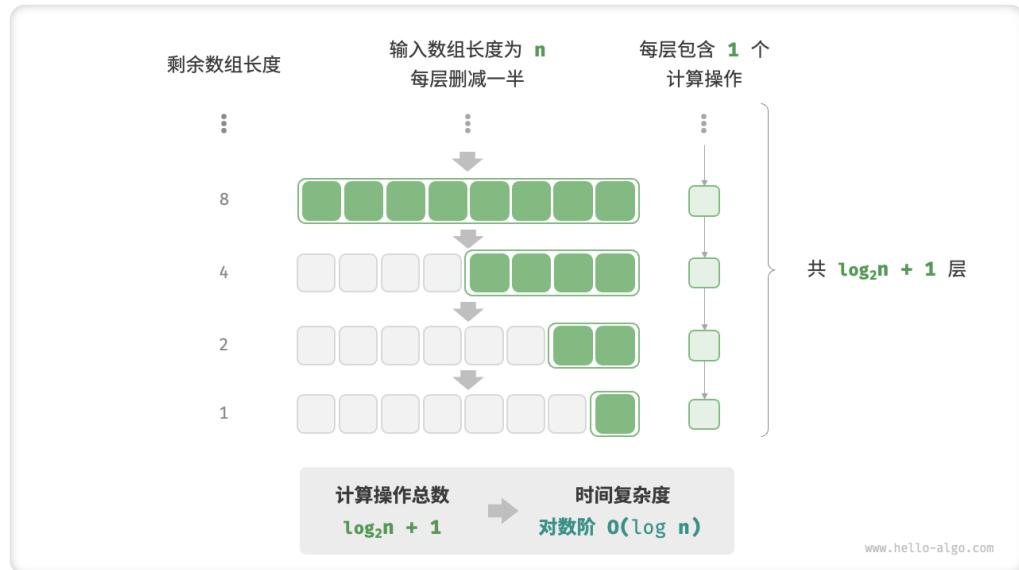


Figure 2-6. 对数阶的时间复杂度

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.py ==
def log_recur(n: float) -> int:
    """ 对数阶 (递归实现) """
    if n <= 1:
        return 0
    return log_recur(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通常为  $O(n \log n)$ ，例如快速排序、归并排序、堆排序等。

```
# == File: time_complexity.py ==
def linear_log_recur(n: float) -> int:
    """ 线性对数阶 """
    if n <= 1:
        return 1
    count: int = linear_log_recur(n // 2) + linear_log_recur(n // 2)
    for _ in range(n):
        count += 1
    return count
```

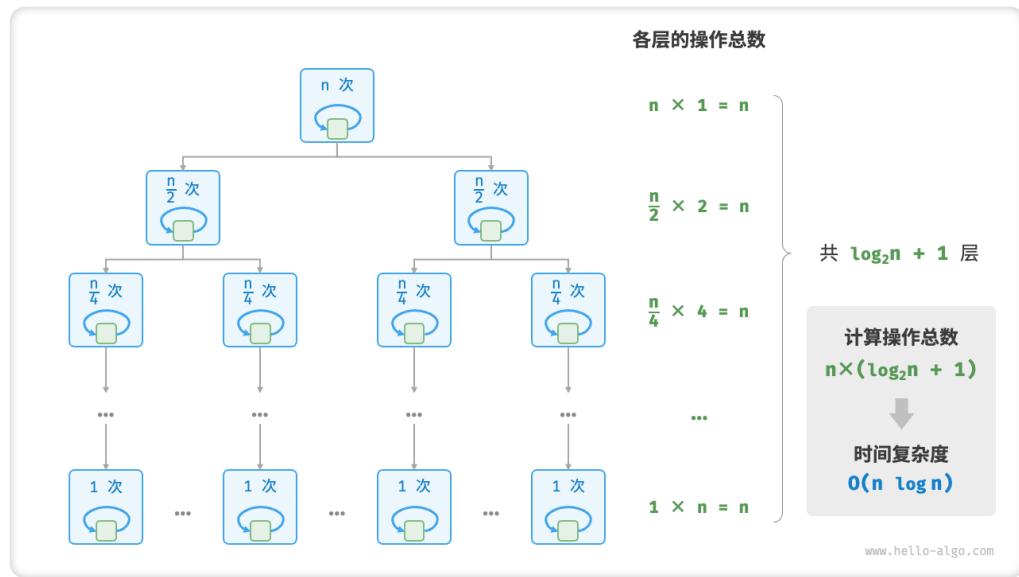


Figure 2-7. 线性对数阶的时间复杂度

### 阶乘阶 $O(n!)$

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

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

阶乘通常使用递归实现。例如以下代码，第一层分裂出  $n$  个，第二层分裂出  $n - 1$  个，以此类推，直至第  $n$  层时终止分裂。

```
# === File: time_complexity.py ===
def factorial_recur(n: int) -> int:
    """ 阶乘阶（递归实现） """
    if n == 0:
        return 1
    count: int = 0
    # 从 1 个分裂出 n 个
    for _ in range(n):
        count += factorial_recur(n - 1)
    return count
```

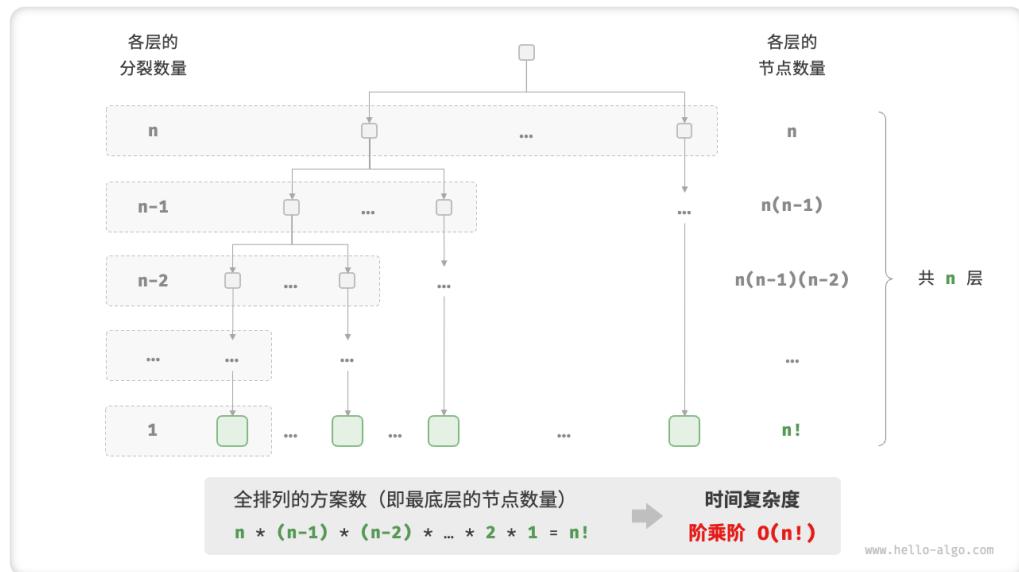


Figure 2-8. 阶乘阶的时间复杂度

### 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固定的，而是与输入数据的分布有关。例如，假设输入一个长度为  $n$  的数组 `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，需要完整遍历数组，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」。相应地，“函数渐近下界”用 $\Omega$ 记号来表示，代表「最佳时间复杂度」。

```
# == File: worst_best_time_complexity.py ==
def random_numbers(n: int) -> list[int]:
    """ 生成一个数组，元素为：1, 2, ..., n，顺序被打乱 """
    # 生成数组 nums =: 1, 2, 3, ..., n
    nums: list[int] = [i for i in range(1, n + 1)]
    # 随机打乱数组元素
    random.shuffle(nums)
    return nums

def find_one(nums: list[int]) -> int:
    """ 查找数组 nums 中数字 1 所在索引 """
    for i in range(len(nums)):
        # 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
```



实际应用中我们很少使用「最佳时间复杂度」，因为通常只有在很小概率下才能达到，可能会带来一定的误导性。相反，「最差时间复杂度」更为实用，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可能很小，因此并不能最真实地反映算法运行效率。相较之下，「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 $\Theta$ 记号来表示。

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。在这种情况下，我们通常使用最差时间复杂度作为算法效率的评判标准。



### 为什么很少看到 $\Theta$ 符号？

可能由于 $O$ 符号过于朗朗上口，我们常常使用它来表示「平均复杂度」，但从严格意义上讲，这种做法并不规范。在本书和其他资料中，若遇到类似“平均时间复杂度 $O(n)$ ”的表述，请将其直接理解为 $\Theta(n)$ 。

## 2.3. 空间复杂度

「空间复杂度 Space Complexity」用于衡量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非常类似。

### 2.3.1. 算法相关空间

算法运行过程中使用的内存空间主要包括以下几种：

- 「输入空间」用于存储算法的输入数据；
- 「暂存空间」用于存储算法运行过程中的变量、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」。

暂存空间可以进一步划分为三个部分：

- 「暂存数据」用于保存算法运行过程中的各种常量、变量、对象等。
- 「栈帧空间」用于保存调用函数的上下文数据。系统在每次调用函数时都会在栈顶部创建一个栈帧，函数返回后，栈帧空间会被释放。
- 「指令空间」用于保存编译后的程序指令，在实际统计中通常忽略不计。

因此，在分析一段程序的空间复杂度时，我们一般统计 **暂存数据、输出数据、栈帧空间** 三部分。

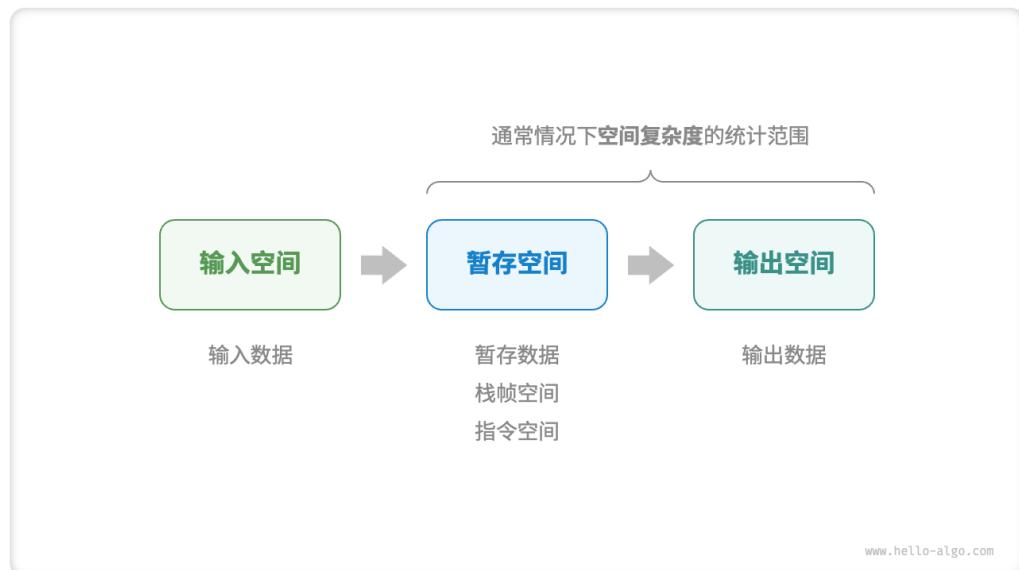


Figure 2-9. 算法使用的相关空间

```
class Node:
    """ 类 """
    def __init__(self, x: int):
        self.val: int = x           # 节点值
```

```

self.next: Optional[Node] = None # 指向下一节点的指针（引用）

def function() -> int:
    """ 函数 """
    # do something...
    return 0

def algorithm(n) -> int: # 输入数据
    A: int = 0           # 暂存数据（常量，一般用大写字母表示）
    b: int = 0           # 暂存数据（变量）
    node = Node(0)       # 暂存数据（对象）
    c: int = function() # 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与时间复杂度大致相同，只是将统计对象从“计算操作数量”转为“使用空间大小”。与时间复杂度不同的是，**我们通常只关注「最差空间复杂度」**，这是因为内存空间是一项硬性要求，我们必须确保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**，分别是输入数据的最差分布和算法运行过程中的最差时间点。

- **以最差输入数据为准**。当  $n < 10$  时，空间复杂度为  $O(1)$ ；但当  $n > 10$  时，初始化的数组 `nums` 占用  $O(n)$  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**。例如，程序在执行最后一行之前，占用  $O(1)$  空间；当初始化数组 `nums` 时，程序占用  $O(n)$  空间；因此最差空间复杂度为  $O(n)$ ；

```

def algorithm(n: int) -> None:
    a: int = 0           # O(1)
    b: List[int] = [0] * 10000   # O(1)
    if n > 10:
        nums: List[int] = [0] * 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()` 在运行过程中会同时存在  $n$  个未返回的 `recur()`，从而占用  $O(n)$  的栈帧空间。

```

def function() -> int:
    # do something
    return 0

def loop(n: int) -> None:
    """ 循环 O(1) """
    for _ in range(n):

```

```

function()

def recur(n: int) -> int:
    """ 递归 O(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

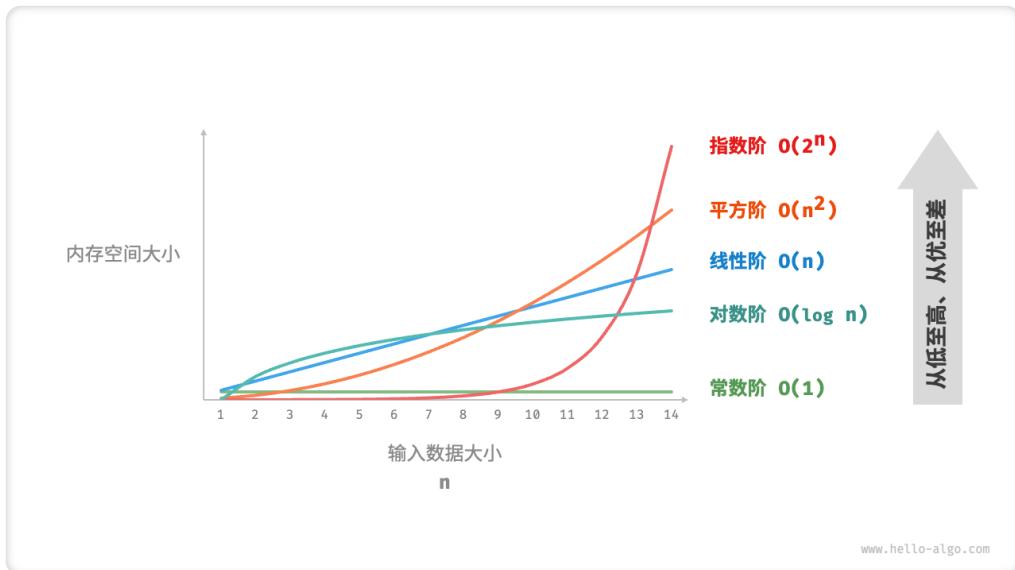


Figure 2-10. 空间复杂度的常见类型



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，现阶段我们先专注于理解空间复杂度的含义和推算方法。

#### 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.py ===
def constant(n: int) -> None:
    """ 常数阶 """
    # 常量、变量、对象占用 O(1) 空间
    a: int = 0
    nums: list[int] = [0] * 10000
    node = ListNode(0)
    # 循环中的变量占用 O(1) 空间
    for _ in range(n):
        c: int = 0
    # 循环中的函数占用 O(1) 空间
    for _ in range(n):
        function()
```

## 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.py ===
def linear(n: int) -> None:
    """ 线性阶 """
    # 长度为 n 的列表占用 O(n) 空间
    nums: list[int] = [0] * n
    # 长度为 n 的哈希表占用 O(n) 空间
    mapp = dict[int, str]()
    for i in range(n):
        mapp[i] = str(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.py ===
def linear_recur(n: int) -> None:
    """ 线性阶（递归实现） """
    print(" 递归 n =", n)
    if n == 1:
        return
    linear_recur(n - 1)
```

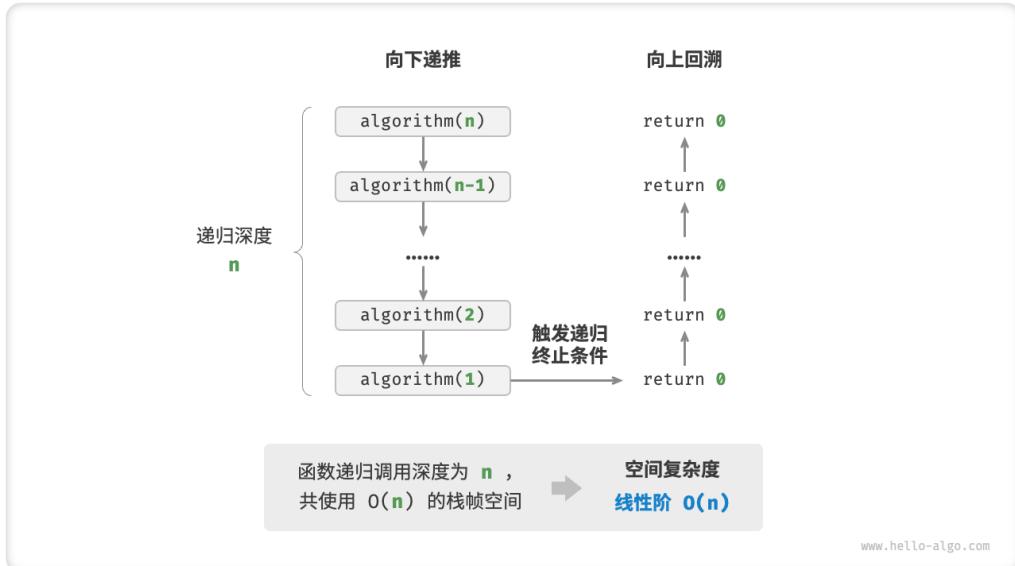


Figure 2-11. 递归函数产生的线性阶空间复杂度

### 平方阶 $O(n^2)$

平方阶常见于矩阵和图，元素数量与  $n$  成平方关系。

```
# === File: space_complexity.py ===
def quadratic(n: int) -> None:
    """ 平方阶 """
    # 二维列表占用  $O(n^2)$  空间
    num_matrix: list[list[int]] = [[0] * n for _ in range(n)]
```

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  $\frac{n}{2}$ ，因此总体占用  $O(n^2)$  空间。

```
# === File: space_complexity.py ===
def quadratic_recur(n: int) -> int:
    """ 平方阶 (递归实现) """
    if n <= 0:
        return 0
    # 数组 nums 长度为 n, n-1, ..., 2, 1
    nums: list[int] = [0] * n
    return quadratic_recur(n - 1)
```

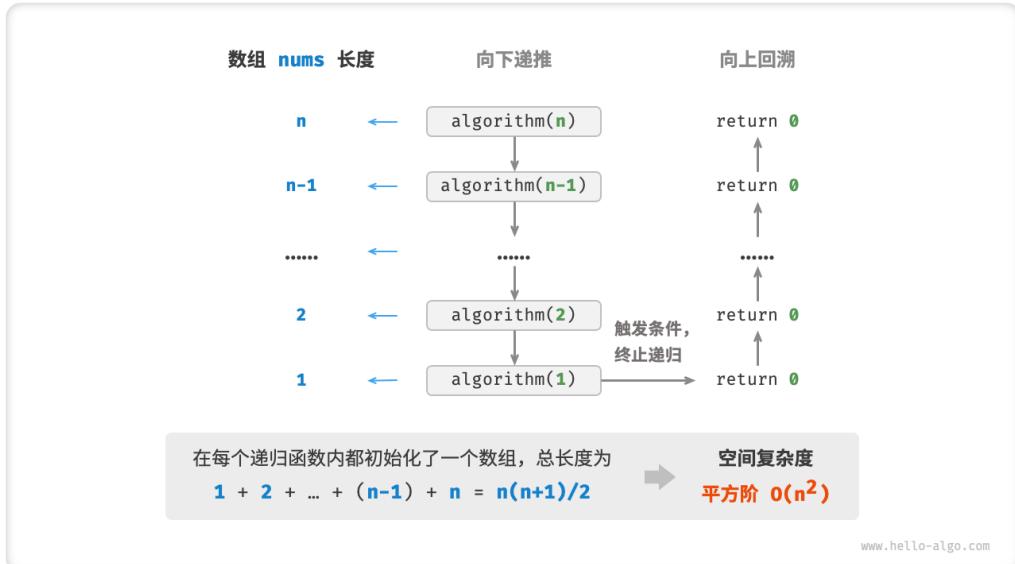


Figure 2-12. 递归函数产生的平方阶空间复杂度

### 指数阶 $O(2^n)$

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

```
# === File: space_complexity.py ===
def build_tree(n: int) -> TreeNode | None:
    """ 指数阶（建立满二叉树） """
    if n == 0:
        return None
    root = TreeNode(0)
    root.left = build_tree(n - 1)
    root.right = build_tree(n - 1)
    return root
```

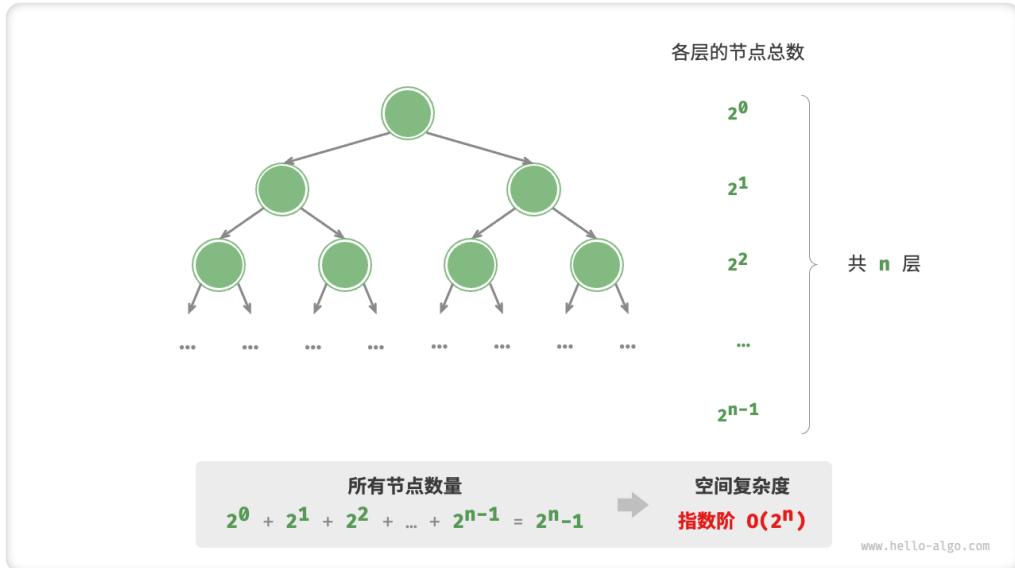


Figure 2-13. 满二叉树产生的指数阶空间复杂度

### 对数阶 $O(\log n)$

对数阶常见于分治算法和数据类型转换等。

例如“归并排序”算法，输入长度为  $n$  的数组，每轮递归将数组从中点划分为两半，形成高度为  $\log n$  的递归树，使用  $O(\log n)$  栈帧空间。

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  $O(\log_{10} n) = O(\log n)$ 。

#### 2.3.4. 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达到最优。然而在实际情况中，同时优化时间复杂度和空间复杂度通常是非常困难的。

**降低时间复杂度通常需要以提升空间复杂度为代价，反之亦然。**我们将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，则称为“以时间换空间”。

选择哪种思路取决于我们更看重哪个方面。在大多数情况下，时间比空间更宝贵，因此以空间换时间通常是更常用的策略。当然，在数据量很大的情况下，控制空间复杂度也是非常重要的。

## 2.4. 小结

### 算法效率评估

- 时间效率和空间效率是评价算法性能的两个关键维度。
- 我们可以通过实际测试来评估算法效率，但难以消除测试环境的影响，且会耗费大量计算资源。

- 复杂度分析可以克服实际测试的弊端，分析结果适用于所有运行平台，并且能够揭示算法在不同数据规模下的效率。

## 时间复杂度

- 时间复杂度用于衡量算法运行时间随数据量增长的趋势，可以有效评估算法效率，但在某些情况下可能失效，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。
- 最差时间复杂度使用大  $O$  符号表示，即函数渐近上界，反映当  $n$  趋向正无穷时， $T(n)$  的增长级别。
- 推算时间复杂度分为两步，首先统计计算操作数量，然后判断渐近上界。
- 常见时间复杂度从小到大排列有  $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n), O(n!)$  等。
- 某些算法的时间复杂度非固定，而是与输入数据的分布有关。时间复杂度分为最差、最佳、平均时间复杂度，最佳时间复杂度几乎不用，因为输入数据一般需要满足严格条件才能达到最佳情况。
- 平均时间复杂度反映算法在随机数据输入下的运行效率，最接近实际应用中的算法性能。计算平均时间复杂度需要统计输入数据分布以及综合后的数学期望。

## 空间复杂度

- 类似于时间复杂度，空间复杂度用于衡量算法占用空间随数据量增长的趋势。
- 算法运行过程中的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，其中栈帧空间通常仅在递归函数中影响空间复杂度。
- 我们通常只关注最差空间复杂度，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  $O(1), O(\log n), O(n), O(n^2), O(2^n)$  等。

## 3. 数据结构简介

### 3.1. 数据与内存

#### 3.1.1. 基本数据类型

谈及计算机中的数据，我们会想到文本、图片、视频、语音、3D 模型等各种形式。尽管这些数据的组织形式各异，但它们都由各种基本数据类型构成。

「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」按照不同的长度分为 byte, short, int, long。在满足取值范围的前提下，我们应该尽量选取较短的整数类型，以减小内存空间占用；
- 「浮点数」表示小数，按长度分为 float, double，选用规则与整数相同。
- 「字符」在计算机中以字符集形式保存，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完成编号到字符的转换。
- 「布尔」代表逻辑中的“是”与“否”，其占用空间需根据编程语言确定。

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，可以表示  $2^{32}$  个不同的数字；
2. 将最高位视为符号位，0 代表正数，1 代表负数，一共可表示  $2^{31}$  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“源码、反码、补码”的相关知识，有兴趣的同学可以查阅学习；

其他整数类型 byte, short, long 的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 \*



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，等学完必读章节后再单独攻克。

细心的你可能会发现：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？按理说 float 需要表示小数，取值范围应该变小才对。

实际上，这是因为浮点数 float 采用了不同的表示方式。根据 IEEE 754 标准，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

$$\begin{aligned} S &\in \{0, 1\}, \quad E \in \{1, 2, \dots, 254\} \\ (1 + N) &= (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) \subset [1, 2 - 2^{-23}] \end{aligned}$$



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

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

现在我们可以回答最初的问题：**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。

**尽管浮点数 float 扩展了取值范围，但其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浮点数的精度，这是因为：

- 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，此处不再详述。

### 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**，它的核心是“结构”，而非“数据”。如果想要表示“一排数字”，我们自然会想到使用「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，但至于具体存储的是整数 int、小数 float、还是字符 char，则与“数据结构”无关。换句话说，基本数据类型提供了数据的“内容类型”，而数据结构提供了数据的“组织方式”。

```
# Python 的 list 可以自由存储各种基本数据类型和对象
list = [0, 0.0, 'a', False]
```

### 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 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）。

在算法运行过程中，相关数据都存储在内存中。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为表格中的每个单元格分配编号，确保每个内存空间都有唯一的内存地址。有了这些地址，程序便可以访问内存中的数据。

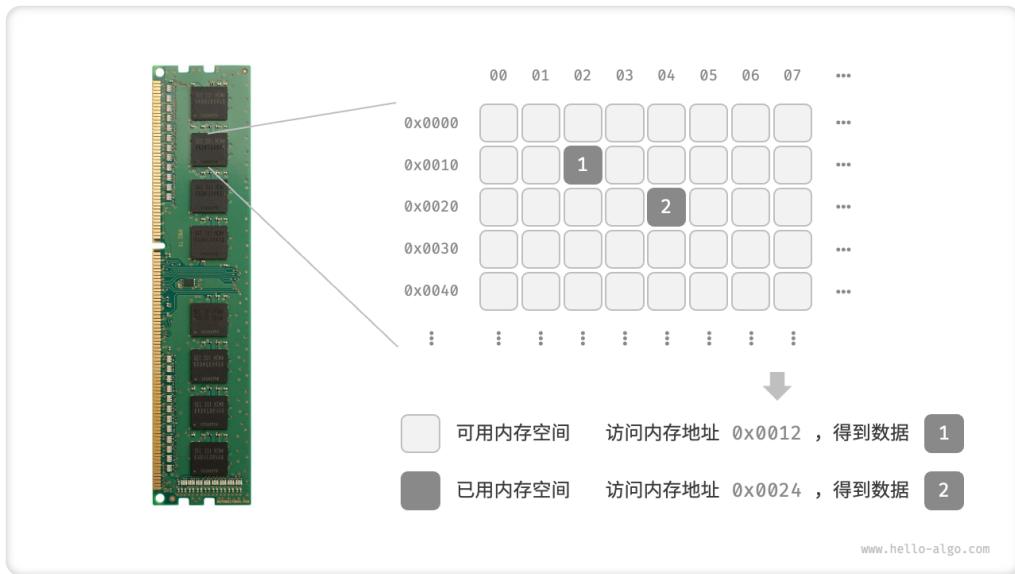


Figure 3-2. 内存条、内存空间、内存地址

在数据结构与算法的设计中，**内存资源是一个重要的考虑因素**。内存是所有程序的共享资源，当内存被某个程序占用时，其他程序无法同时使用。我们需要根据剩余内存资源的实际情况来设计算法。例如，算法所占用的内存峰值不应超过系统剩余空闲内存；如果运行的程序很多并且缺少大量连续的内存空间，那么所选用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

数据结构可以从逻辑结构和物理结构两个维度进行分类。

### 3.2.1. 逻辑结构：线性与非线性

「逻辑结构」揭示了数据元素之间的逻辑关系。在数组和链表中，数据按照顺序依次排列，体现了数据之间的线性关系；而在树中，数据从顶部向下按层次排列，表现出祖先与后代之间的派生关系；图则由节点和边构成，反映了复杂的网络关系。

逻辑结构通常分为「线性」和「非线性」两类。线性结构比较直观，指数据在逻辑关系上呈线性排列；非线性结构则相反，呈非线性排列，例如网状或树状结构。

- 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；

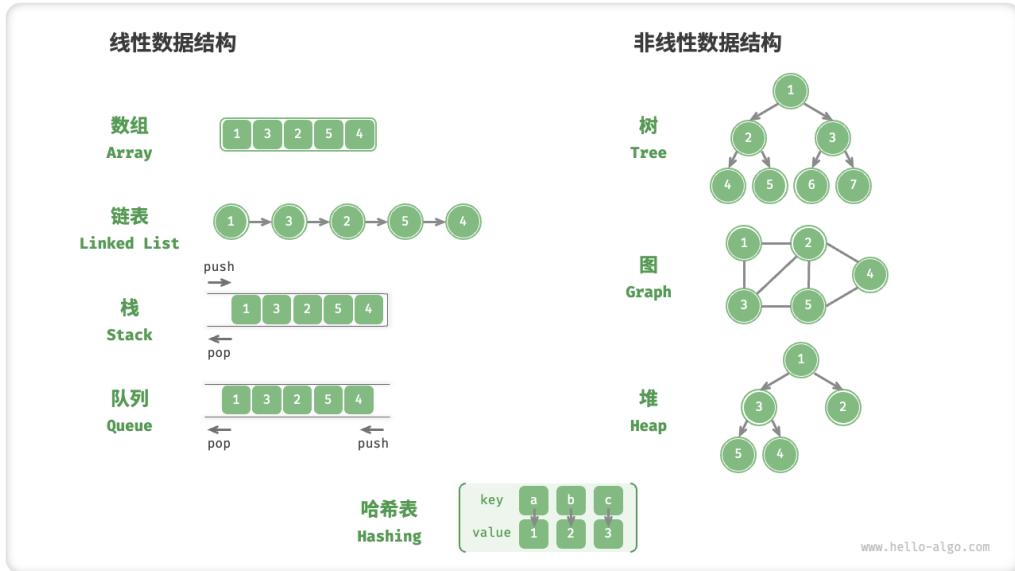


Figure 3-3. 线性与非线性数据结构

### 3.2.2. 物理结构：连续与离散



如若阅读起来有困难，建议先阅读下一章“数组与链表”，然后再回头理解物理结构的含义。

「物理结构」体现了数据在计算机内存中的存储方式，可以分为数组的连续空间存储和链表的离散空间存储。物理结构从底层决定了数据的访问、更新、增删等操作方法，同时在时间效率和空间效率方面呈现出互补的特点。

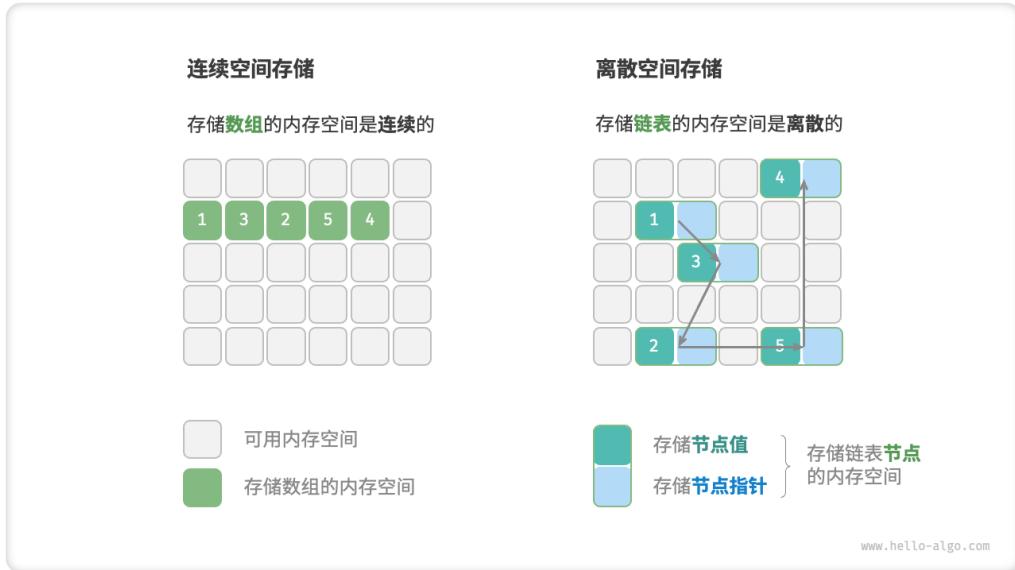


Figure 3-4. 连续空间存储与离散空间存储

所有数据结构都是基于数组、链表或二者的组合实现的。例如，栈和队列既可以使用数组实现，也可以使用链表实现；而哈希表的实现可能同时包含数组和链表。

- 基于数组可实现：栈、队列、哈希表、树、堆、图、矩阵、张量（维度  $\geq 3$  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」，这意味着此类数据结构在初始化后长度不可变。相对应地，基于链表实现的数据结构被称为「动态数据结构」，这类数据结构在初始化后，仍可以在程序运行过程中对其长度进行调整。



数组与链表是其他所有数据结构的“底层积木”，建议读者投入更多时间深入了解这两种基本数据结构。

### 3.3. 小结

- 计算机中的基本数据类型包括整数 byte, short, int, long、浮点数 float, double、字符 char 和布尔 boolean，它们的取值范围取决于占用空间大小和表示方式。
- 当程序运行时，数据被存储在计算机内存中。每个内存空间都拥有对应的内存地址，程序通过这些内存地址访问数据。
- 数据结构可以从逻辑结构和物理结构两个角度进行分类。逻辑结构描述了数据元素之间的逻辑关系，而物理结构描述了数据在计算机内存中的存储方式。
- 常见的逻辑结构包括线性、树状和网状等。通常我们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。哈希表的实现可能同时包含线性和非线性结构。
- 物理结构主要分为连续空间存储（数组）和离散空间存储（链表）。所有数据结构都是由数组、链表或两者的组合实现的。

## 4. 数组与链表

### 4.1. 数组

「数组 Array」是一种线性数据结构，其将相同类型元素存储在连续的内存空间中。我们将元素在数组中的位置称为元素的「索引 Index」。

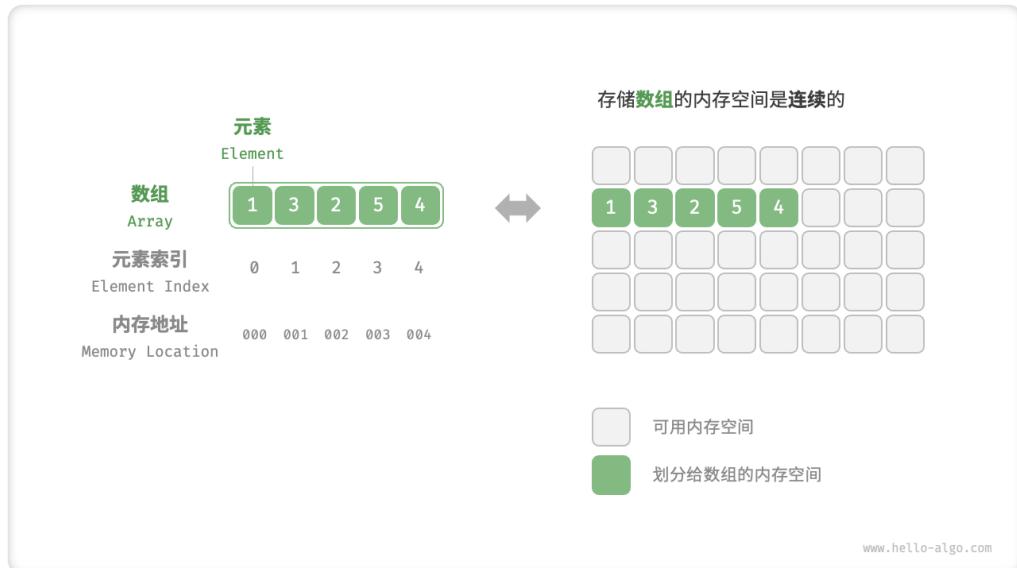


Figure 4-1. 数组定义与存储方式

**数组初始化。**通常有无初始值和给定初始值两种方式，我们可根据需求选择合适的方法。在未给定初始值的情况下，数组的所有元素通常会被初始化为默认值 0。

```
# == File: array.py ==
# 初始化数组
arr: List[int] = [0] * 5 # [ 0, 0, 0, 0, 0 ]
nums: List[int] = [1, 3, 2, 5, 4]
```

#### 4.1.1. 数组优点

**在数组中访问元素非常高效。**由于数组元素被存储在连续的内存空间中，因此计算数组元素的内存地址非常容易。给定数组首个元素的地址和某个元素的索引，我们可以使用以下公式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
```



### 为什么数组元素的索引要从 0 开始编号呢？

观察上图，我们发现数组首个元素的索引为 0，这似乎有些反直觉，因为从 1 开始计数会更自然。

然而，从地址计算公式的角度看，索引本质上表示的是内存地址的偏移量。首个元素的地址偏移量是 0，因此索引为 0 也是合理的。

访问元素的高效性带来了诸多便利。例如，我们可以在  $O(1)$  时间内随机获取数组中的任意一个元素。

```
# === File: array.py ===
def random_access(nums: list[int]) -> int:
    """ 随机访问元素 """
    # 在区间 [0, len(nums)-1] 中随机抽取一个数字
    random_index = random.randint(0, len(nums) - 1)
    # 获取并返回随机元素
    random_num = nums[random_index]
    return random_num
```

### 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.py ==
def extend(nums: list[int], enlarge: int) -> list[int]:
    """ 扩展数组长度 """
    # 初始化一个扩展长度后的数组
    res = [0] * (len(nums) + enlarge)
    # 将原数组中的所有元素复制到新数组
    for i in range(len(nums)):
        res[i] = nums[i]
    # 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。

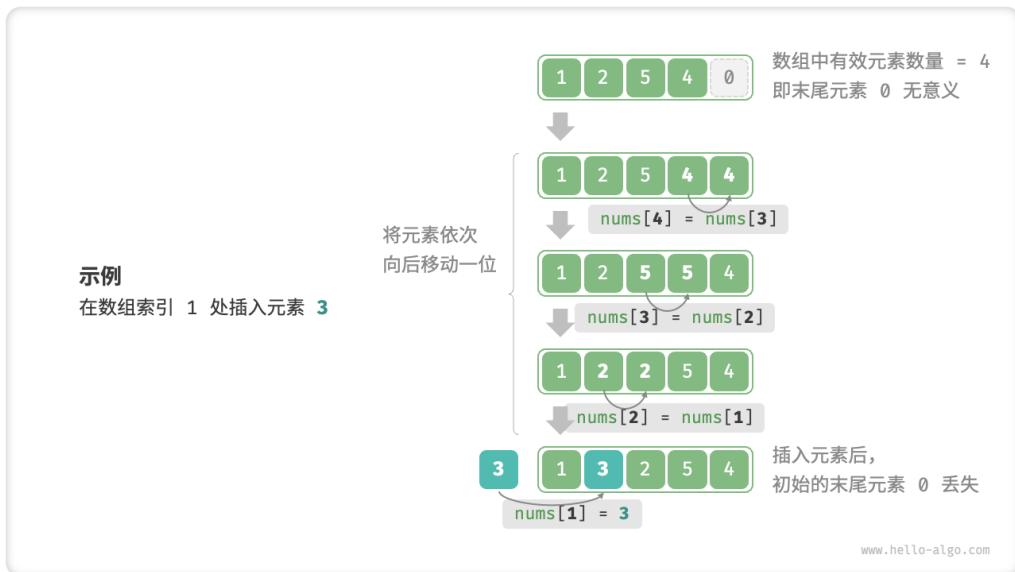


Figure 4-3. 数组插入元素

```
# == File: array.py ==
def insert(nums: list[int], num: int, index: int) -> None:
    """ 在数组的索引 index 处插入元素 num """
    # 把索引 index 以及之后的所有元素向后移动一位
    for i in range(len(nums) - 1, index, -1):
        nums[i] = nums[i - 1]
    # 将 num 赋给 index 处元素
    nums[index] = num
```

删除元素也类似，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。

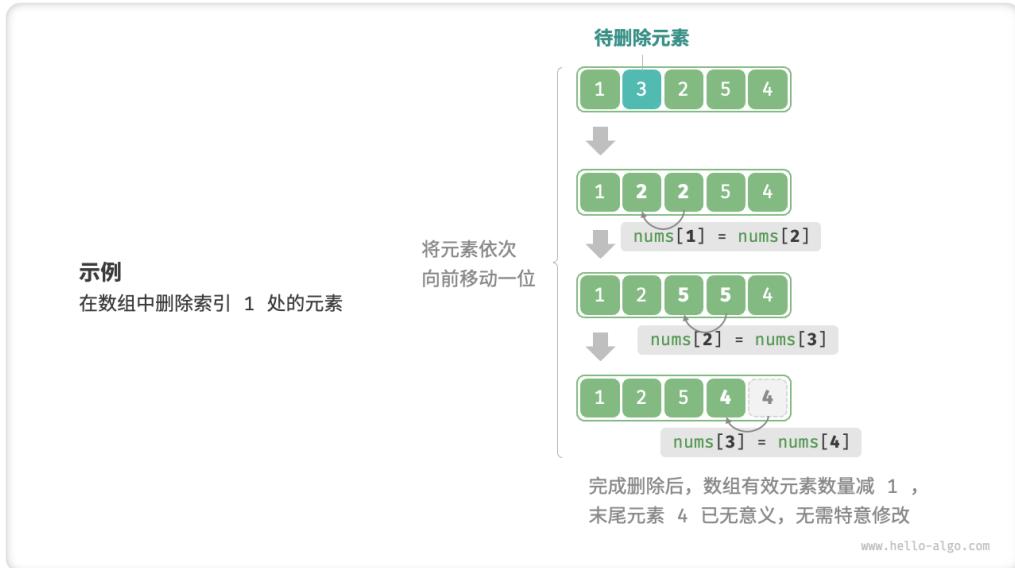


Figure 4-4. 数组删除元素

```
# == File: array.py ==
def remove(nums: list[int], index: int) -> None:
    """ 删除索引 index 处元素 """
    # 把索引 index 之后的所有元素向前移动一位
    for i in range(index, len(nums) - 1):
        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丢失。
- **内存浪费**：我们可以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.py ==
def traverse(nums: list[int]) -> None:
    """ 遍历数组 """
    count = 0
    # 通过索引遍历数组
    for i in range(len(nums)):
        count += 1
    # 直接遍历数组
```

```
for num in nums:  
    count += 1  
# 同时遍历数据索引和元素  
for i, num in enumerate(nums):  
    count += 1
```

**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.py ===  
def find(nums: list[int], target: int) -> int:  
    """ 在数组中查找指定元素 """  
    for i in range(len(nums)):  
        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

内存空间是所有程序的公共资源，排除已被占用的内存空间，空闲内存空间通常散落在内存各处。在上一节中，我们提到存储数组的内存空间必须是连续的，而当我们需要申请一个非常大的数组时，空闲内存中可能没有这么大的连续空间。与数组相比，链表更具灵活性，它可以被存储在非连续的内存空间中。

「链表 Linked List」是一种线性数据结构，其每个元素都是一个节点对象，各个节点之间通过指针连接，从当前节点通过指针可以访问到下一个节点。由于指针记录了下个节点的内存地址，因此无需保证内存地址的连续性，从而可以将各个节点分散存储在内存各处。

链表「节点 Node」包含两项数据，一是节点「值 Value」，二是指向下一节点的「指针 Pointer」，或称「引用 Reference」。

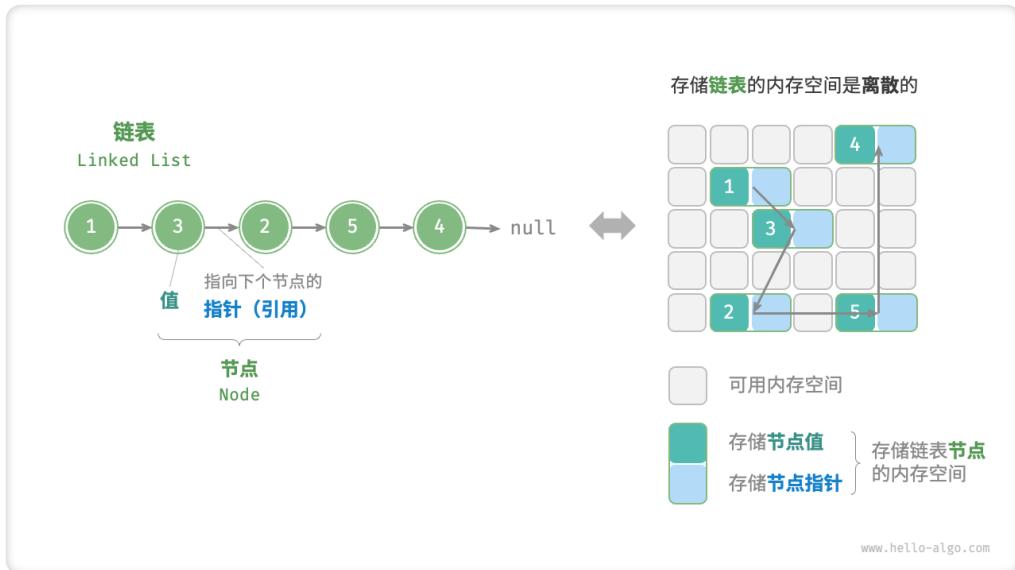


Figure 4-5. 链表定义与存储方式

```
class ListNode:
    """ 链表节点类 """
    def __init__(self, val: int):
        self.val: int = val           # 节点值
        self.next: Optional[ListNode] = None # 指向下一节点的指针 (引用)
```



### 尾节点指向什么？

我们将链表的最后一个节点称为「尾节点」，其指向的是“空”，在 Java, C++, Python 中分别记为 `null`, `nullptr`, `None`。在不引起歧义的前提下，本书都使用 `null` 来表示空。



### 如何称呼链表？

在编程语言中，数组整体就是一个变量，例如数组 `nums`，包含各个元素 `nums[0]`, `nums[1]` 等等。而链表是由多个节点对象组成，我们通常将头节点当作链表的代称，例如头节点 `head` 和链表 `head` 实际上是同义的。

**链表初始化方法。**建立链表分为两步，第一步是初始化各个节点对象，第二步是构建引用指向关系。完成后，即可以从链表的头节点（即首个节点）出发，通过指针 `next` 依次访问所有节点。

```
# === File: linked_list.py ===
# 初始化链表 1 -> 3 -> 2 -> 5 -> 4
# 初始化各个节点
```

```

n0 = ListNode(1)
n1 = ListNode(3)
n2 = ListNode(2)
n3 = ListNode(5)
n4 = 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

链表中插入与删除节点的操作效率高。例如，如果我们想在链表中间的两个节点  $A, B$  之间插入一个新节点  $P$ ，我们只需要改变两个节点指针即可，时间复杂度为  $O(1)$ ；相比之下，数组的插入操作效率要低得多。

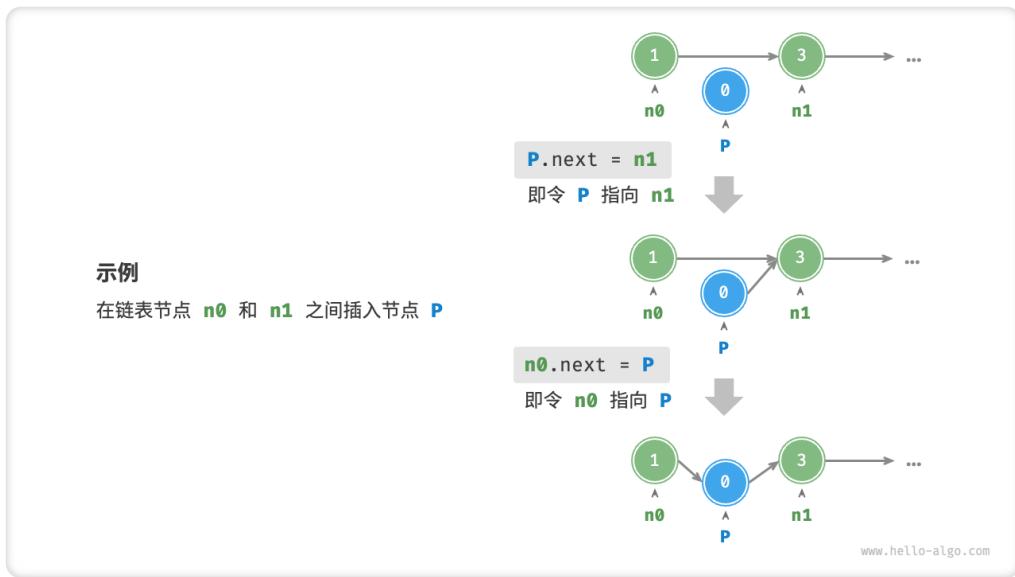


Figure 4-6. 链表插入节点

```

# == File: linked_list.py ==
def insert(n0: ListNode, P: ListNode) -> None:
    """ 在链表的节点 n0 之后插入节点 P """
    n1 = n0.next
    P.next = n1
    n0.next = P

```

在链表中删除节点也非常方便，只需改变一个节点的指针即可。如下图所示，尽管在删除操作完成后，节点  $P$  仍然指向  $n_1$ ，但实际上  $P$  已经不再属于此链表，因为遍历此链表时无法访问到  $P$ 。

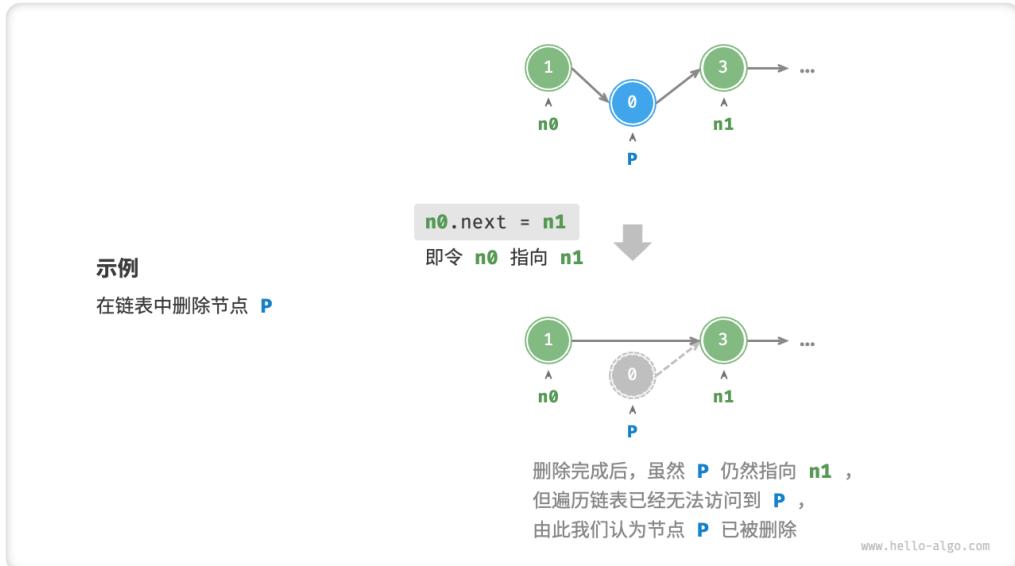


Figure 4-7. 链表删除节点

```
# === File: linked_list.py ===
def remove(n0: ListNode) -> None:
    """ 删除链表的节点 n0 之后的首个节点 """
    if not n0.next:
        return
    # n0 -> P -> n1
    P = n0.next
    n1 = P.next
    n0.next = n1
```

### 4.2.2. 链表缺点

**链表访问节点效率较低。**如上节所述，数组可以在  $O(1)$  时间下访问任意元素。然而，链表无法直接访问任意节点，这是因为系统需要从头节点出发，逐个向后遍历直至找到目标节点。例如，若要访问链表索引为 `index` (即第 `index + 1` 个) 的节点，则需要向后遍历 `index` 轮。

```
# === File: linked_list.py ===
def access(head: ListNode, index: int) -> ListNode | None:
    """ 访问链表中索引为 index 的节点 """
    for _ in range(index):
        if not head:
            return None
        head = head.next
    return head
```

**链表的内存占用较大。**链表以节点为单位，每个节点除了保存值之外，还需额外保存指针（引用）。这意味着在相同数据量的情况下，链表比数组需要占用更多的内存空间。

### 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` 的节点，输出节点在链表中的索引。

```
# == File: linked_list.py ==
def find(head: ListNode, target: int) -> int:
    """ 在链表中查找值为 target 的首个节点 """
    index = 0
    while head:
        if head.val == target:
            return index
        head = head.next
        index += 1
    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。单向链表的节点包含值和指向下一节点的指针（引用）两项数据。我们将首个节点称为头节点，将最后一个节点成为尾节点，尾节点指向 `null`。

**环形链表。**如果我们令单向链表的尾节点指向头节点（即首尾相接），则得到一个环形链表。在环形链表中，任意节点都可以视作头节点。

**双向链表。**与单向链表相比，双向链表记录了两个方向的指针（引用）。双向链表的节点定义同时包含指向后继节点（下一节点）和前驱节点（上一节点）的指针。相较于单向链表，双向链表更具灵活性，可以朝两个方向遍历链表，但相应地也需要占用更多的内存空间。

```
class ListNode:
    """ 双向链表节点类 """
    def __init__(self, val: int):
        self.val: int = val           # 节点值
        self.next: Optional[ListNode] = None  # 指向后继节点的指针（引用）
        self.prev: Optional[ListNode] = None  # 指向前驱节点的指针（引用）
```

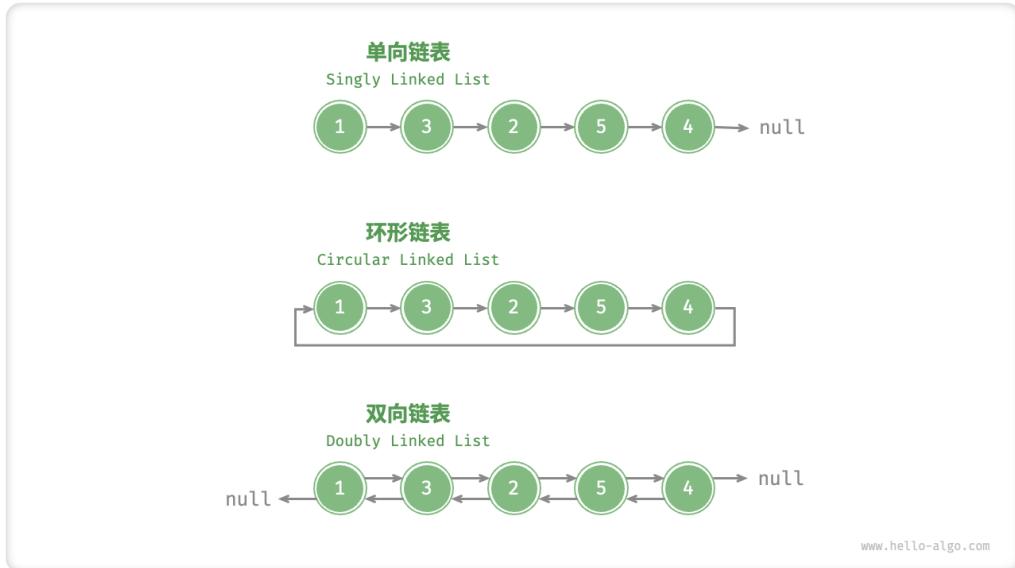


Figure 4-8. 常见链表种类

### 4.3. 列表

**数组长度不可变导致实用性降低。**在许多情况下，我们事先无法确定需要存储多少数据，这使数组长度的选择变得困难。若长度过小，需要在持续添加数据时频繁扩容数组；若长度过大，则会造成内存空间的浪费。

为解决此问题，出现了一种被称为「动态数组 Dynamic Array」的数据结构，即长度可变的数组，也常被称为「列表 List」。列表基于数组实现，继承了数组的优点，并且可以在程序运行过程中动态扩容。在列表中，我们可以自由添加元素，而无需担心超过容量限制。

#### 4.3.1. 列表常用操作

**初始化列表。**通常我们会使用“无初始值”和“有初始值”的两种初始化方法。

```
# == File: list.py ==
# 初始化列表
# 无初始值
list1: List[int] = []
# 有初始值
list: List[int] = [1, 3, 2, 5, 4]
```

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

```
# == File: list.py ==
# 访问元素
num: int = 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。相较于数组，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  $O(1)$ ，但插入和删除元素的效率仍与数组相同，时间复杂度为  $O(N)$ 。

```
# === File: list.py ===  
# 清空列表  
list.clear()  
  
# 尾部添加元素  
list.append(1)  
list.append(3)  
list.append(2)  
list.append(5)  
list.append(4)  
  
# 中间插入元素  
list.insert(3, 6) # 在索引 3 处插入数字 6  
  
# 删除元素  
list.pop(3)      # 删除索引 3 处的元素
```

遍历列表。与数组一样，列表可以根据索引遍历，也可以直接遍历各元素。

```
# === File: list.py ===  
# 通过索引遍历列表  
count: int = 0  
for i in range(len(list)):  
    count += 1  
  
# 直接遍历列表元素  
count: int = 0  
for n in list:  
    count += 1
```

拼接两个列表。给定一个新列表 `list1`，我们可以将该列表拼接到原列表的尾部。

```
# === File: list.py ===  
# 拼接两个列表  
list1: List[int] = [6, 8, 7, 10, 9]  
list += list1 # 将列表 list1 拼接到 list 之后
```

**排序列表。**排序也是常用的方法之一。完成列表排序后，我们便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。

```
# == File: list.py ==
# 排序列表
list.sort() # 排序后，列表元素从小到大排列
```

### 4.3.2. 列表实现 \*

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

- **初始容量：**选取一个合理的数组初始容量。在本示例中，我们选择 10 作为初始容量。
- **数量记录：**声明一个变量 size，用于记录列表当前元素数量，并随着元素插入和删除实时更新。根据此变量，我们可以定位列表尾部，以及判断是否需要扩容。
- **扩容机制：**插入元素时可能超出列表容量，此时需要扩容列表。扩容方法是根据扩容倍数创建一个更大的数组，并将当前数组的所有元素依次移动至新数组。在本示例中，我们规定每次将数组扩容至之前的 2 倍。

本示例旨在帮助读者直观理解列表的工作机制。实际编程语言中，列表实现更加标准和复杂，各个参数的设定也非常有考究，例如初始容量、扩容倍数等。感兴趣的读者可以查阅源码进行学习。

```
# == File: my_list.py ==
class MyList:
    """ 列表类简易实现 """

    def __init__(self):
        """ 构造方法 """
        self.__capacity: int = 10 # 列表容量
        self.__nums: list[int] = [0] * self.__capacity # 数组（存储列表元素）
        self.__size: int = 0 # 列表长度（即当前元素数量）
        self.__extend_ratio: int = 2 # 每次列表扩容的倍数

    def size(self) -> int:
        """ 获取列表长度（即当前元素数量） """
        return self.__size

    def capacity(self) -> int:
        """ 获取列表容量 """
        return self.__capacity

    def get(self, index: int) -> int:
        """ 访问元素 """
        # 索引如果越界则抛出异常，下同
        if index < 0 or index >= self.__size:
            raise IndexError("索引越界")
```

```
return self.__nums[index]

def set(self, num: int, index: int) -> None:
    """ 更新元素 """
    if index < 0 or index >= self.__size:
        raise IndexError("索引越界")
    self.__nums[index] = num

def add(self, num: int) -> None:
    """ 尾部添加元素 """
    # 元素数量超出容量时，触发扩容机制
    if self.size() == self.capacity():
        self.extend_capacity()
    self.__nums[self.__size] = num
    self.__size += 1

def insert(self, num: int, index: int) -> None:
    """ 中间插入元素 """
    if index < 0 or index >= self.__size:
        raise IndexError("索引越界")
    # 元素数量超出容量时，触发扩容机制
    if self.__size == self.capacity():
        self.extend_capacity()
    # 索引 i 以及之后的元素都向后移动一位
    for j in range(self.__size - 1, index - 1, -1):
        self.__nums[j + 1] = self.__nums[j]
    self.__nums[index] = num
    # 更新元素数量
    self.__size += 1

def remove(self, index: int) -> int:
    """ 删除元素 """
    if index < 0 or index >= self.__size:
        raise IndexError("索引越界")
    num = self.__nums[index]
    # 索引 i 之后的元素都向前移动一位
    for j in range(index, self.__size - 1):
        self.__nums[j] = self.__nums[j + 1]
    # 更新元素数量
    self.__size -= 1
    # 返回被删除元素
    return num

def extend_capacity(self) -> None:
    """ 列表扩容 """
    # 新建一个长度为 self.__size 的数组，并将原数组拷贝到新数组
```

```

self.__nums = self.__nums + [0] * self.capacity() * (self.__extend_ratio - 1)
# 更新列表容量
self.__capacity = len(self.__nums)

def to_array(self) -> list[int]:
    """ 返回有效长度的列表"""
    return self.__nums[: self.__size]

```

## 4.4. 小结

- 数组和链表是两种基本数据结构，分别代表数据在计算机内存中的连续空间存储和离散空间存储方式。两者的优缺点呈现出互补的特性。
- 数组支持随机访问、占用内存较少；但插入和删除元素效率低，且初始化后长度不可变。
- 链表通过更改指针实现高效的节点插入与删除，且可以灵活调整长度；但节点访问效率低、占用内存较多。常见的链表类型包括单向链表、循环链表、双向链表。
- 动态数组，又称列表，是基于数组实现的一种数据结构。它保留了数组的优势，同时可以灵活调整长度。列表的出现极大地提高了数组的易用性，但可能导致部分内存空间浪费。
- 下表总结并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

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他数据，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节点，这样的多次“搬运”降低了整体效率。

- 下表对比了数组与链表在各种操作上的效率。

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

操作	数组	链表
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）原则的线性数据结构。

我们可以将栈类比为桌面上的一摞盘子，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。我们将盘子替换为各种类型的元素（如整数、字符、对象等），就得到了栈数据结构。

在栈中，我们把堆叠元素的顶部称为「栈顶」，底部称为「栈底」。将把元素添加到栈顶的操作叫做「入栈」，而删除栈顶元素的操作叫做「出栈」。

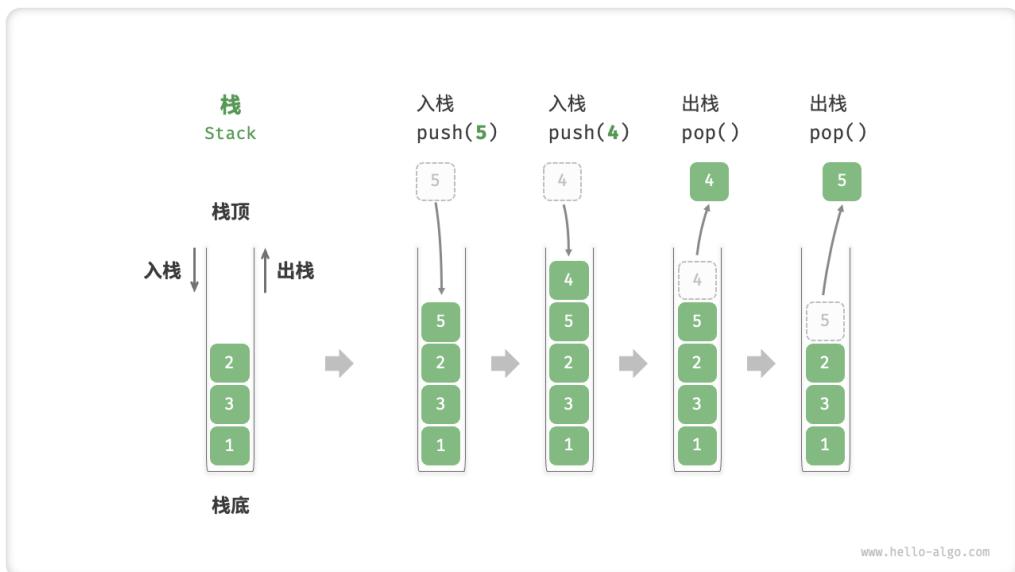


Figure 5-1. 栈的先入后出规则

#### 5.1.1. 栈常用操作

栈的常用操作如下表所示，具体的方法名需要根据所使用的编程语言来确定。在此，我们以常见的 `push()` , `pop()` , `peek()` 命名为例。

方法	描述	时间复杂度
<code>push()</code>	元素入栈（添加至栈顶）	$O(1)$
<code>pop()</code>	栈顶元素出栈	$O(1)$
<code>peek()</code>	访问栈顶元素	$O(1)$

通常情况下，我们可以直接使用编程语言内置的栈类。然而，某些语言可能没有专门提供栈类，这时我们可以将该语言的「数组」或「链表」视作栈来使用，并通过“脑补”来忽略与栈无关的操作。

```
# === File: stack.py ===
# 初始化栈
# Python 没有内置的栈类，可以把 List 当作栈来使用
stack: List[int] = []

# 元素入栈
stack.append(1)
stack.append(3)
stack.append(2)
stack.append(5)
stack.append(4)

# 访问栈顶元素
peek: int = stack[-1]

# 元素出栈
pop: int = stack.pop()

# 获取栈的长度
size: int = len(stack)

# 判断是否为空
is_empty: bool = len(stack) == 0
```

### 5.1.2. 栈的实现

为了深入了解栈的运行机制，我们来尝试自己实现一个栈类。

栈遵循先入后出的原则，因此我们只能在栈顶添加或删除元素。然而，数组和链表都可以在任意位置添加和删除元素，**因此栈可以被视为一种受限制的数组或链表**。换句话说，我们可以“屏蔽”数组或链表的部分无关操作，使其对外表现的逻辑符合栈的特性。

#### 基于链表的实现

使用链表来实现栈时，我们可以将链表的头节点视为栈顶，尾节点视为栈底。

对于入栈操作，我们只需将元素插入链表头部，这种节点插入方法被称为“头插法”。而对于出栈操作，只需将头节点从链表中删除即可。

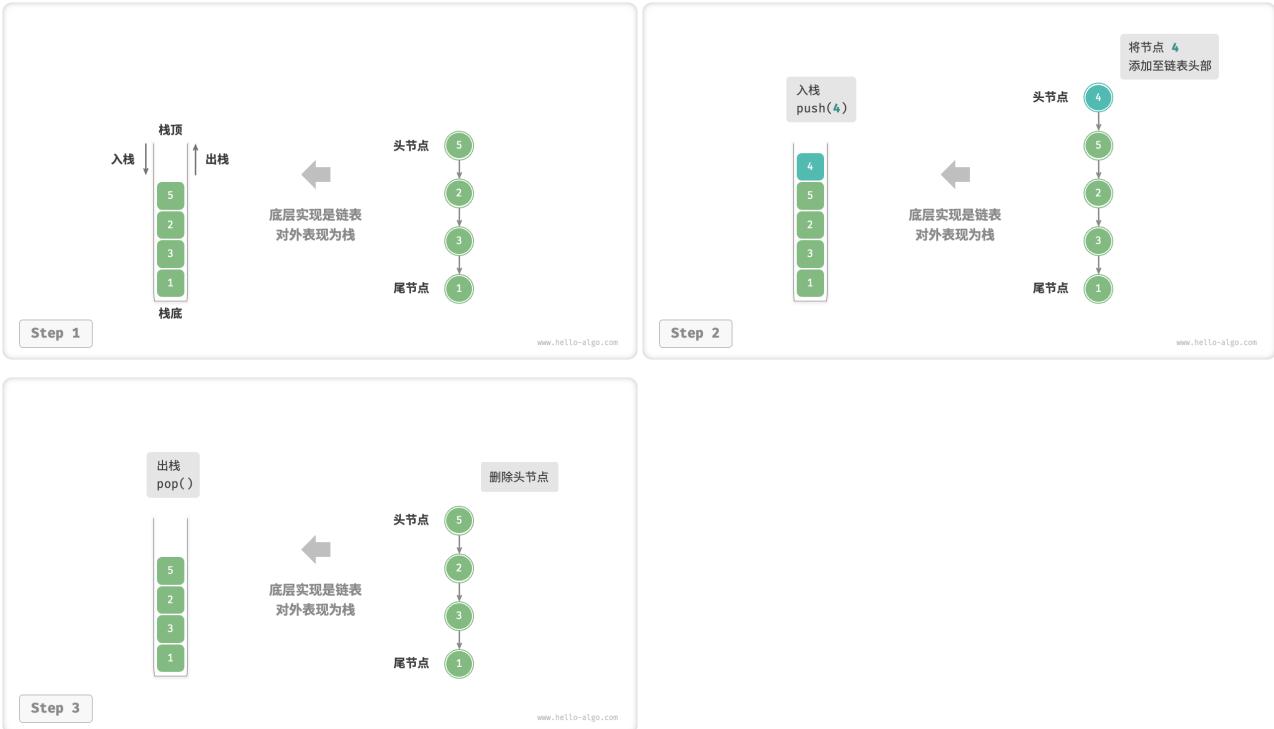


Figure 5-2. 基于链表实现栈的入栈出栈操作

以下是基于链表实现栈的示例代码。

```
# === File: linkedlist_stack.py ===
class LinkedListStack:
    """ 基于链表实现的栈"""

    def __init__(self):
        """ 构造方法"""
        self.__peek: ListNode | None = None
        self.__size: int = 0

    def size(self) -> int:
        """ 获取栈的长度"""
        return self.__size

    def is_empty(self) -> bool:
        """ 判断栈是否为空"""
        return not self.__peek

    def push(self, val: int) -> None:
        """ 入栈"""
        node = ListNode(val)
        node.next = self.__peek
        self.__peek = node
        self.__size += 1
```

```

    self.__peek = node
    self.__size += 1

def pop(self) -> int:
    """ 出栈 """
    num: int = self.peek()
    self.__peek = self.__peek.next
    self.__size -= 1
    return num

def peek(self) -> int:
    """ 访问栈顶元素 """
    # 判空处理
    if not self.__peek:
        return None
    return self.__peek.val

def to_list(self) -> list[int]:
    """ 转化为列表用于打印 """
    arr: list[int] = []
    node = self.__peek
    while node:
        arr.append(node.val)
        node = node.next
    arr.reverse()
    return arr

```

## 基于数组的实现

在基于「数组」实现栈时，我们可以将数组的尾部作为栈顶。在这样的设计下，入栈与出栈操作就分别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会源源不断地增加，因此我们可以使用动态数组，这样就无需自行处理数组扩容问题。以下为示例代码。

```
# == File: array_stack.py ==
class ArrayStack:
    """ 基于数组实现的栈"""

    def __init__(self) -> None:
        """ 构造方法"""
        self.__stack: list[int] = []

    def size(self) -> int:
        """ 获取栈的长度"""
        return len(self.__stack)

    def is_empty(self) -> bool:
        """ 判断栈是否为空"""
        return self.__stack == []

    def push(self, item: int) -> None:
        """ 入栈"""
        self.__stack.append(item)

    def pop(self) -> int:
        """ 出栈"""
        if self.is_empty():
            raise IndexError(" 栈为空")
        return self.__stack.pop()

    def peek(self) -> int:
        """ 访问栈顶元素"""
        if self.is_empty():
            raise IndexError(" 栈为空")
```

```
return self._stack[-1]

def to_list(self) -> list[int]:
    """ 返回列表用于打印 """
    return self._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，但这已超出了栈的定义范畴，因此一般不会用到。

#### 时间效率

在基于数组的实现中，入栈和出栈操作都是在预先分配好的连续内存中进行，具有很好的缓存本地性，因此效率较高。然而，如果入栈时超出数组容量，会触发扩容机制，导致该次入栈操作的时间复杂度变为  $O(n)$ 。

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效率降低的问题。但是，入栈操作需要初始化节点对象并修改指针，因此效率相对较低。不过，如果入栈元素本身就是节点对象，那么可以省去初始化步骤，从而提高效率。

综上所述，当入栈与出栈操作的元素是基本数据类型（如 `int`, `double`）时，我们可以得出以下结论：

- 基于数组实现的栈在触发扩容时效率会降低，但由于扩容是低频操作，因此平均效率更高；
- 基于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，系统会为列表分配“初始容量”，该容量可能超过实际需求。并且，扩容机制通常是按照特定倍率（例如 2 倍）进行扩容，扩容后的容量也可能超出实际需求。因此，**基于数组实现的栈可能造成一定的空间浪费**。

然而，由于链表节点需要额外存储指针，**因此链表节点占用的空间相对较大**。

综上，我们不能简单地确定哪种实现更加节省内存，需要针对具体情况分析。

### 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，浏览器就会将上一个网页执行入栈，这样我们就可以通过「后退」操作回到上一页面。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，那么需要两个栈来配合实现。
- **程序内存管理。**每次调用函数时，系统都会在栈顶添加一个栈帧，用于记录函数的上下文信息。在递归函数中，向下递推阶段会不断执行入栈操作，而向上回溯阶段则会执行出栈操作。

## 5.2. 队列

「队列 Queue」是一种遵循先入先出（First In, First Out）规则的线性数据结构。顾名思义，队列模拟了排队现象，即新来的人不断加入队列的尾部，而位于队列头部的人逐个离开。

我们把队列的头部称为「队首」，尾部称为「队尾」，把将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」。

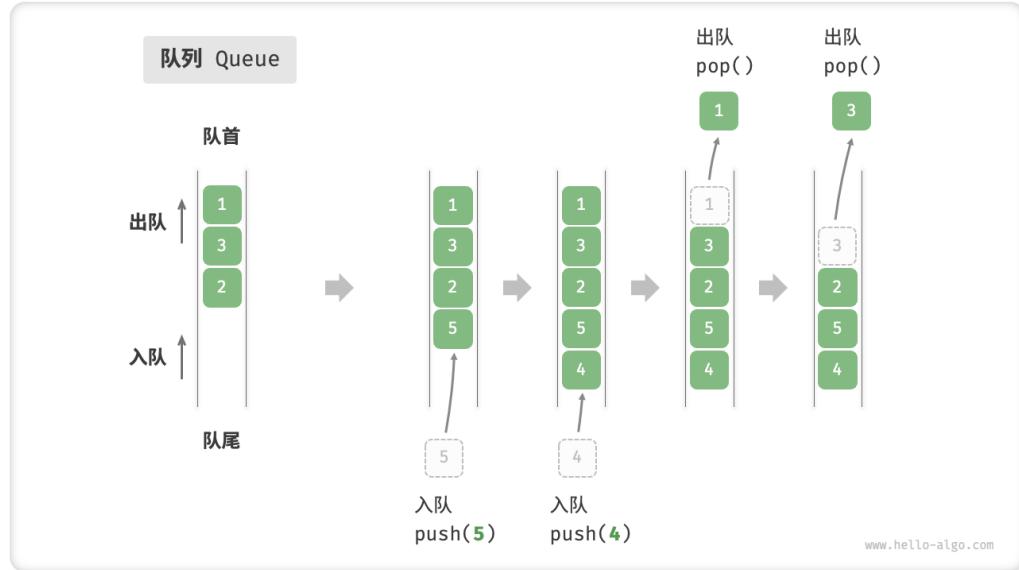


Figure 5-4. 队列的先入先出规则

### 5.2.1. 队列常用操作

队列的常见操作如下表所示。需要注意的是，不同编程语言的方法名称可能会有所不同。我们在此采用与栈相同的方法命名。

方法名	描述	时间复杂度
push()	元素入队，即将元素添加至队尾	$O(1)$
pop()	队首元素出队	$O(1)$
peek()	访问队首元素	$O(1)$

我们可以直接使用编程语言中现成的队列类。

```
# === File: queue.py ===
# 初始化队列
# 在 Python 中，我们一般将双向队列类 deque 看作队列使用
# 虽然 queue.Queue() 是纯正的队列类，但不太好用，因此不建议
```

```

que: Deque[int] = collections.deque()

# 元素入队
que.append(1)
que.append(3)
que.append(2)
que.append(5)
que.append(4)

# 访问队首元素
front: int = que[0];

# 元素出队
pop: int = que.popleft()

# 获取队列的长度
size: int = len(que)

# 判断队列是否为空
is_empty: bool = len(que) == 0

```

### 5.2.2. 队列实现

为了实现队列，我们需要一种数据结构，可以在一端添加元素，并在另一端删除元素。因此，链表和数组都可以用来实现队列。

#### 基于链表的实现

对于链表实现，我们可以将链表的「头节点」和「尾节点」分别视为队首和队尾，规定队尾仅可添加节点，而队首仅可删除节点。



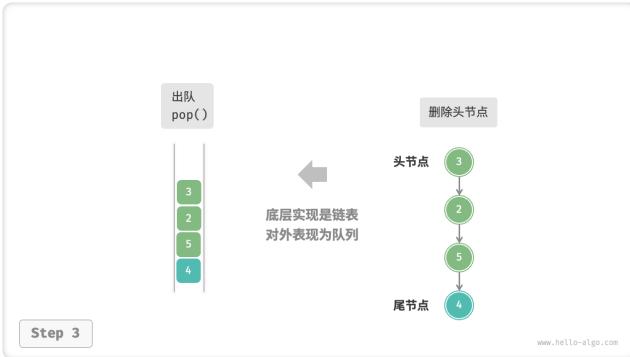


Figure 5-5. 基于链表实现队列的入队出队操作

以下是用链表实现队列的示例代码。

```
# == File: linkedlist_queue.py ==
class LinkedListQueue:
    """ 基于链表实现的队列"""

    def __init__(self):
        """ 构造方法"""
        self.__front: ListNode | None = None # 头节点 front
        self.__rear: ListNode | None = None # 尾节点 rear
        self.__size: int = 0

    def size(self) -> int:
        """ 获取队列的长度"""
        return self.__size

    def is_empty(self) -> bool:
        """ 判断队列是否为空"""
        return not self.__front

    def push(self, num: int) -> None:
        """ 入队"""
        # 尾节点后添加 num
        node = ListNode(num)
        # 如果队列为空，则令头、尾节点都指向该节点
        if self.__front is None:
            self.__front = node
            self.__rear = node
        # 如果队列不为空，则将该节点添加到尾节点后
        else:
            self.__rear.next = node
            self.__rear = node
        self.__size += 1
```

```
def pop(self) -> int:
    """ 出队 """
    num = self.peek()
    # 删除头节点
    self.__front = self.__front.next
    self.__size -= 1
    return num

def peek(self) -> int:
    """ 访问队首元素 """
    if self.size() == 0:
        print(" 队列为空")
        return False
    return self.__front.val

def to_list(self) -> list[int]:
    """ 转化为列表用于打印 """
    queue = []
    temp = self.__front
    while temp:
        queue.append(temp.val)
        temp = temp.next
    return queue
```

## 基于数组的实现

由于数组删除首元素的时间复杂度为  $O(n)$ ，这会导致出队操作效率较低。然而，我们可以采用以下巧妙方法来避免这个问题。

我们可以使用一个变量 `front` 指向队首元素的索引，并维护一个变量 `queSize` 用于记录队列长度。定义 `rear = front + queSize`，这个公式计算出的 `rear` 指向队尾元素之后的下一个位置。

基于此设计，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` 增加 1；
- 对于出队操作，只需将 `front` 增加 1，并将 `queSize` 减少 1；

可以看到，入队和出队操作都只需进行一次操作，时间复杂度均为  $O(1)$ 。

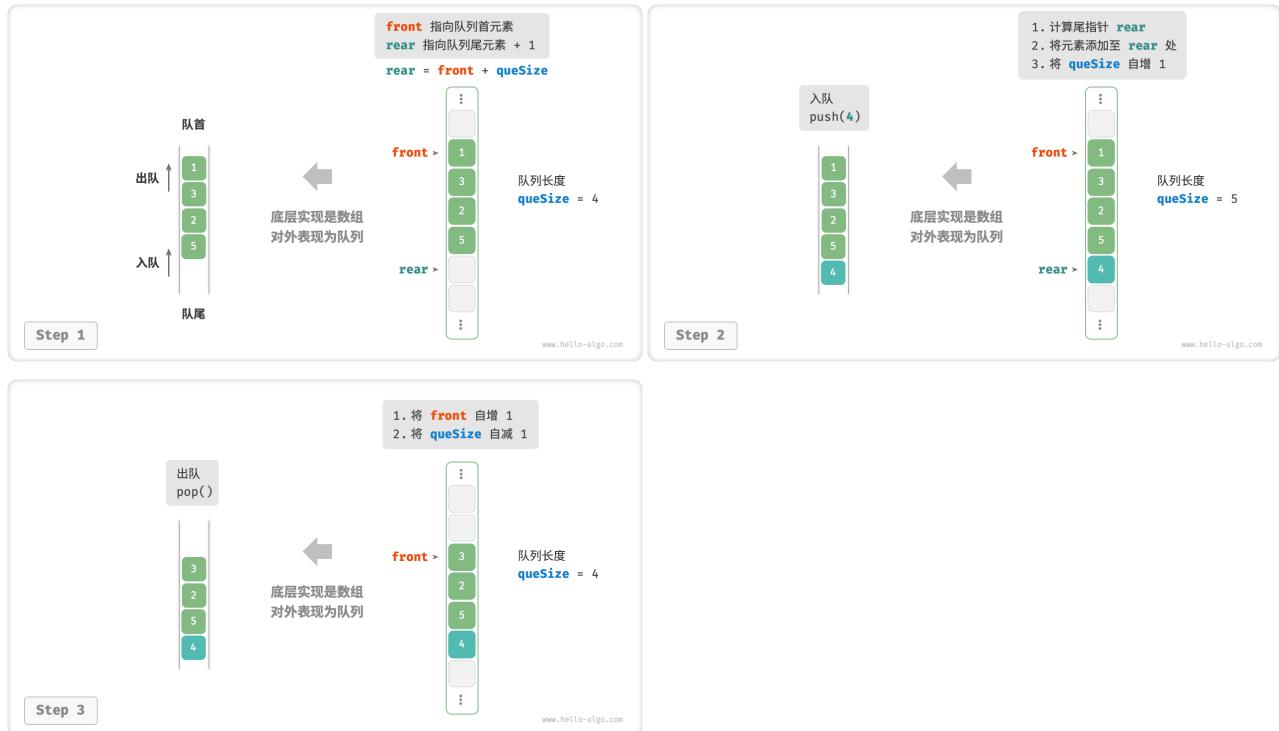


Figure 5-6. 基于数组实现队列的入队出队操作

你可能会发现一个问题：在不断进行入队和出队的过程中，`front` 和 `rear` 都在向右移动，当它们到达数组尾部时就无法继续移动了。为解决此问题，我们可以将数组视为首尾相接的「环形数组」。

对于环形数组，我们需要让 `front` 或 `rear` 在越过数组尾部时，直接回到数组头部继续遍历。这种周期性规律可以通过“取余操作”来实现，代码如下所示。

```
# == File: array_queue.py ==
class ArrayQueue:
    """ 基于环形数组实现的队列 """

    def __init__(self, size: int) -> None:
        """ 构造方法 """
        self.__nums: list[int] = [0] * size # 用于存储队列元素的数组
        self.__front: int = 0 # 队首指针，指向队首元素
        self.__size: int = 0 # 队列长度

    def capacity(self) -> int:
        """ 获取队列的容量 """
        return len(self.__nums)

    def size(self) -> int:
        """ 获取队列的长度 """
        return self.__size
```

```
def is_empty(self) -> bool:
    """ 判断队列是否为空 """
    return self.__size == 0

def push(self, num: int) -> None:
    """ 入队 """
    if self.__size == self.capacity():
        raise IndexError("队列已满")
    # 计算尾指针，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 F
    rear: int = (self.__front + self.__size) % self.capacity()
    # 将 num 添加至队尾
    self.__nums[rear] = num
    self.__size += 1

def pop(self) -> int:
    """ 出队 """
    num: int = self.peek()
    # 队首指针向后移动一位，若越过尾部则返回到数组头部
    self.__front = (self.__front + 1) % self.capacity()
    self.__size -= 1
    return num

def peek(self) -> int:
    """ 访问队首元素 """
    if self.is_empty():
        raise IndexError("队列为空")
    return self.__nums[self.__front]

def to_list(self) -> list[int]:
    """ 返回列表用于打印 """
    res: list[int] = [0] * self.size()
    j: int = self.__front
    for i in range(self.size()):
        res[i] = self.__nums[(j % self.capacity())]
        j += 1
    return res
```

以上实现的队列仍然具有局限性，即其长度不可变。然而，这个问题不难解决，我们可以将数组替换为动态数组，从而引入扩容机制。有兴趣的同学可以尝试自行实现。

两种实现的对比结论与栈一致，在此不再赘述。

### 5.2.3. 队列典型应用

- **淘宝订单。**购物者下单后，订单将加入队列中，系统随后会根据顺序依次处理队列中的订单。在双十一期间，短时间内会产生海量订单，高并发成为工程师们需要重点攻克的问题。
- **各类待办事项。**任何需要实现“先来后到”功能的场景，例如打印机的任务队列、餐厅的出餐队列等。队列在这些场景中可以有效地维护处理顺序。

## 5.3. 双向队列

对于队列，我们仅能在头部删除或在尾部添加元素。然而，「双向队列 Deque」提供了更高的灵活性，允许在头部和尾部执行元素的添加或删除操作。

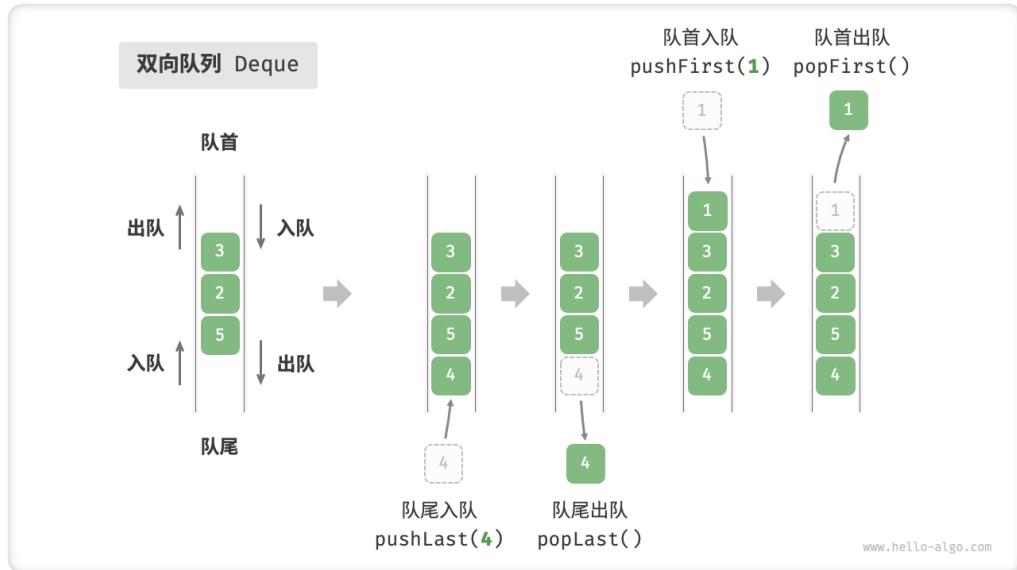


Figure 5-7. 双向队列的操作

### 5.3.1. 双向队列常用操作

双向队列的常用操作如下表所示，具体的方法名称需要根据所使用的编程语言来确定。

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	$O(1)$
popFirst()	删除队首元素	$O(1)$
popLast()	删除队尾元素	$O(1)$
peekFirst()	访问队首元素	$O(1)$

方法名	描述	时间复杂度
peekLast()	访问队尾元素	$O(1)$

同样地，我们可以直接使用编程语言中已实现的双向队列类。

```
# === File: deque.py ===
# 初始化双向队列
deque: Deque[int] = collections.deque()

# 元素入队
deque.append(2)      # 添加至队尾
deque.append(5)
deque.append(4)
deque.appendleft(3)   # 添加至队首
deque.appendleft(1)

# 访问元素
front: int = deque[0] # 队首元素
rear: int = deque[-1] # 队尾元素

# 元素出队
pop_front: int = deque.popleft() # 队首元素出队
pop_rear: int = deque.pop()     # 队尾元素出队

# 获取双向队列的长度
size: int = len(deque)

# 判断双向队列是否为空
is_empty: bool = len(deque) == 0
```

### 5.3.2. 双向队列实现 \*

双向队列的实现与队列类似，可以选择链表或数组作为底层数据结构。

#### 基于双向链表的实现

回顾上一节内容，我们使用普通单向链表来实现队列，因为它可以方便地删除头节点（对应出队操作）和在尾节点后添加新节点（对应入队操作）。

对于双向队列而言，头部和尾部都可以执行入队和出队操作。换句话说，双向队列需要实现另一个对称方向的操作。为此，我们采用「双向链表」作为双向队列的底层数据结构。

我们将双向链表的头节点和尾节点视为双向队列的队首和队尾，同时实现在两端添加和删除节点的功能。



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.py ===
class ListNode:
    """ 双向链表节点 """
    def __init__(self, val: int) -> None:
        """ 构造方法 """
        self.val: int = val
        self.next: ListNode | None = None # 后继节点引用 (指针)
        self.prev: ListNode | None = None # 前驱节点引用 (指针)

class LinkedListDeque:
```

```
""" 基于双向链表实现的双向队列"""

def __init__(self) -> None:
    """ 构造方法"""
    self.front: ListNode | None = None # 头节点 front
    self.rear: ListNode | None = None # 尾节点 rear
    self.__size: int = 0 # 双向队列的长度

def size(self) -> int:
    """ 获取双向队列的长度"""
    return self.__size

def is_empty(self) -> bool:
    """ 判断双向队列是否为空"""
    return self.size() == 0

def push(self, num: int, is_front: bool) -> None:
    """ 入队操作"""
    node = ListNode(num)
    # 若链表为空，则令 front, rear 都指向 node
    if self.is_empty():
        self.front = self.rear = node
    # 队首入队操作
    elif is_front:
        # 将 node 添加至链表头部
        self.front.prev = node
        node.next = self.front
        self.front = node # 更新头节点
    # 队尾入队操作
    else:
        # 将 node 添加至链表尾部
        self.rear.next = node
        node.prev = self.rear
        self.rear = node # 更新尾节点
    self.__size += 1 # 更新队列长度

def push_first(self, num: int) -> None:
    """ 队首入队"""
    self.push(num, True)

def push_last(self, num: int) -> None:
    """ 队尾入队"""
    self.push(num, False)

def pop(self, is_front: bool) -> int:
    """ 出队操作"""
    if self.is_empty():
        raise IndexError("队列为空")
    node = self.front if is_front else self.rear
    num = node.data
    if is_front:
        self.front = node.next
        if self.front:
            self.front.prev = None
        else:
            self.rear = None
    else:
        self.rear = node.prev
        if self.rear:
            self.rear.next = None
        else:
            self.front = None
    self.__size -= 1
    return num
```

```
# 若队列为空，直接返回 None
if self.is_empty():
    return None
# 队首出队操作
if is_front:
    val: int = self.front.val # 暂存头节点值
    # 删除头节点
    fnext: ListNode | None = self.front.next
    if fnext != None:
        fnext.prev = None
        self.front.next = None
    self.front = fnext # 更新头节点
# 队尾出队操作
else:
    val: int = self.rear.val # 暂存尾节点值
    # 删除尾节点
    rprev: ListNode | None = self.rear.prev
    if rprev != None:
        rprev.next = None
        self.rear.prev = None
    self.rear = rprev # 更新尾节点
    self.__size -= 1 # 更新队列长度
return val

def pop_first(self) -> int:
    """ 队首出队 """
    return self.pop(True)

def pop_last(self) -> int:
    """ 队尾出队 """
    return self.pop(False)

def peek_first(self) -> int:
    """ 访问队首元素 """
    return None if self.is_empty() else self.front.val

def peek_last(self) -> int:
    """ 访问队尾元素 """
    return None if self.is_empty() else self.rear.val

def to_array(self) -> list[int]:
    """ 返回数组用于打印 """
    node: ListNode | None = self.front
    res: list[int] = [0] * self.size()
    for i in range(self.size()):
        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。在队列的实现基础上，仅需增加“队首入队”和“队尾出队”的方法。

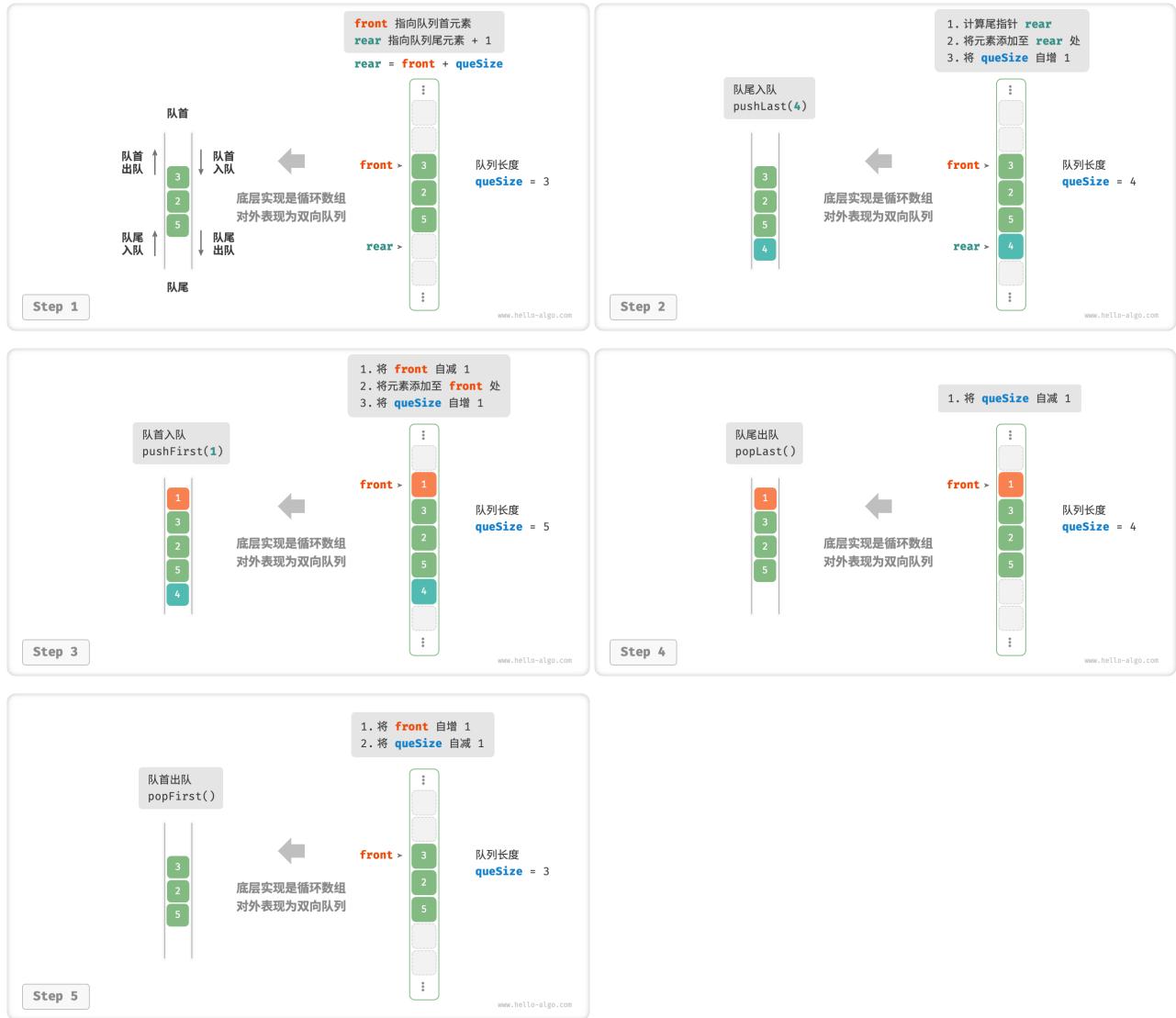


Figure 5-9. 基于数组实现双向队列的入队出队操作

以下是具体实现代码。

```
# == File: array_deque.py ==
class ArrayDeque:
    """ 基于环形数组实现的双向队列"""

    def __init__(self, capacity: int) -> None:
        """ 构造方法"""
        self.__nums: list[int] = [0] * capacity
        self.__front: int = 0
        self.__size: int = 0

    def capacity(self) -> int:
        """ 获取双向队列的容量"""
        return len(self.__nums)

    def size(self) -> int:
        """ 获取双向队列的长度"""
        return self.__size

    def is_empty(self) -> bool:
        """ 判断双向队列是否为空"""
        return self.__size == 0

    def index(self, i: int) -> int:
        """ 计算环形数组索引"""
        # 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 + self.capacity()) % self.capacity()

    def push_first(self, num: int) -> None:
        """ 队首入队"""
        if self.__size == self.capacity():
            print(" 双向队列已满")
            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
        self.__front = self.index(self.__front - 1)
        # 将 num 添加至队首
        self.__nums[self.__front] = num
        self.__size += 1

    def push_last(self, num: int) -> None:
        """ 队尾入队"""
        if self.__size == self.capacity():
            print(" 双向队列已满")
            return
```

```
# 计算尾指针，指向队尾索引 + 1
rear = self.index(self.__front + self.__size)
# 将 num 添加至队尾
self.__nums[rear] = num
self.__size += 1

def pop_first(self) -> int:
    """ 队首出队 """
    num = self.peek_first()
    # 队首指针向后移动一位
    self.__front = self.index(self.__front + 1)
    self.__size -= 1
    return num

def pop_last(self) -> int:
    """ 队尾出队 """
    num = self.peek_last()
    self.__size -= 1
    return num

def peek_first(self) -> int:
    """ 访问队首元素 """
    if self.is_empty():
        raise IndexError(" 双向队列为空 ")
    return self.__nums[self.__front]

def peek_last(self) -> int:
    """ 访问队尾元素 """
    if self.is_empty():
        raise IndexError(" 双向队列为空 ")
    # 计算尾元素索引
    last = self.index(self.__front + self.__size - 1)
    return self.__nums[last]

def to_array(self) -> list[int]:
    """ 返回数组用于打印 """
    # 仅转换有效长度范围内的列表元素
    res = []
    for i in range(self.__size):
        res.append(self.__nums[self.index(self.__front + i)])
    return res
```

### 5.3.3. 双向队列应用

双向队列兼具栈与队列的逻辑，因此它可以实现这两者的所有应用场景，同时提供更高的自由度。

我们知道，软件的“撤销”功能通常使用栈来实现：系统将每次更改操作 `push` 到栈中，然后通过 `pop` 实现撤销。然而，考虑到系统资源的限制，软件通常会限制撤销的步数（例如仅允许保存 50 步）。当栈的长度超过 50 时，软件需要在栈底（即队首）执行删除操作。**但栈无法实现该功能，此时就需要使用双向队列来替代栈。** 请注意，“撤销”的核心逻辑仍然遵循栈的先入后出原则，只是双向队列能够更加灵活地实现一些额外逻辑。

## 5.4. 小结

- 栈是一种遵循先入后出原则的数据结构，可通过数组或链表来实现。
- 从时间效率角度看，栈的数组实现具有较高的平均效率，但在扩容过程中，单次入栈操作的时间复杂度会降低至  $O(n)$ 。相比之下，基于链表实现的栈具有更为稳定的效率表现。
- 在空间效率方面，栈的数组实现可能导致一定程度的空间浪费。但需要注意的是，链表节点所占用的内存空间比数组元素更大。
- 队列是一种遵循先入先出原则的数据结构，同样可以通过数组或链表来实现。在时间效率和空间效率的对比上，队列的结论与前述栈的结论相似。
- 双向队列是一种具有更高自由度的队列，它允许在两端进行元素的添加和删除操作。

# 6. 二分查找

## 6.1. 二分查找

「二分查找 Binary Search」利用数据的有序性，通过每轮减少一半搜索范围来定位目标元素。

给定一个长度为  $n$  的有序数组 `nums`，元素按从小到大的顺序排列。数组索引的取值范围为：

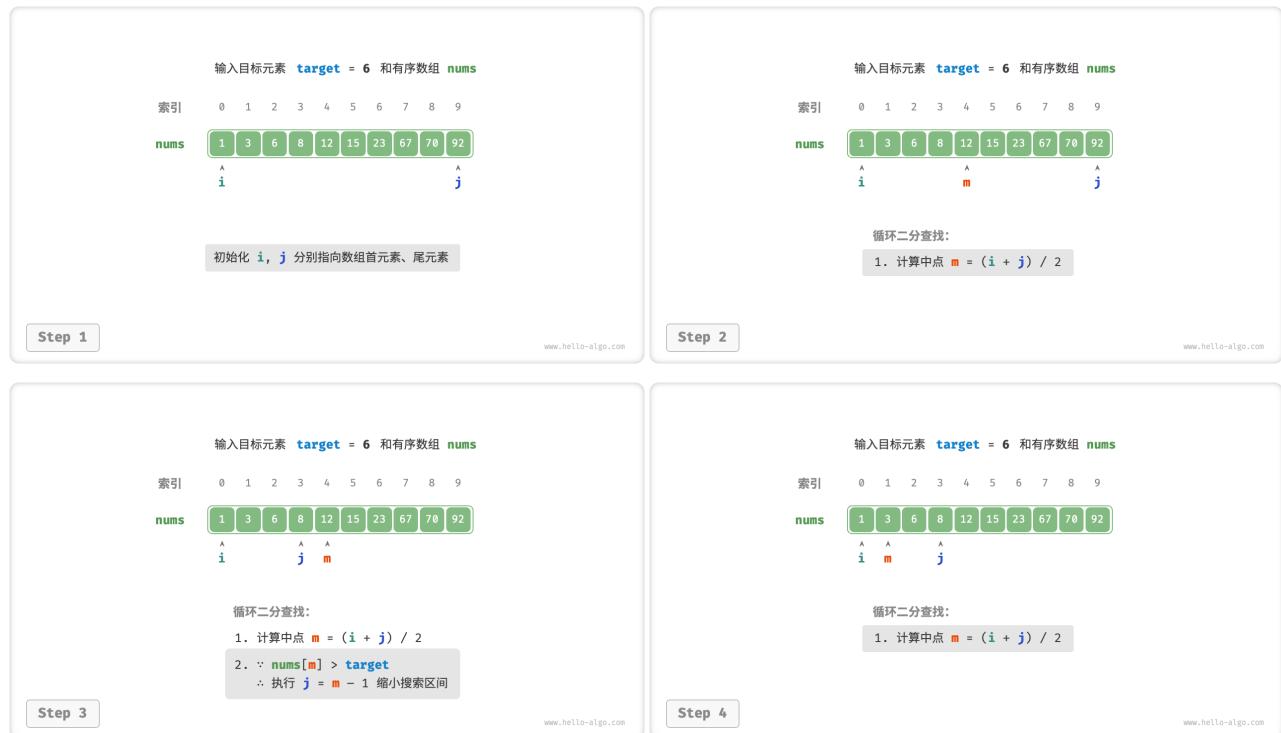
$$0, 1, 2, \dots, n - 1$$

我们通常使用以下两种方法来表示这个取值范围：

1. 双闭区间  $[0, n - 1]$ ，即两个边界都包含自身；在此方法下，区间  $[i, i]$  仍包含 1 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；在此方法下，区间  $[i, i)$  不包含元素；

### 6.1.1. 双闭区间实现

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



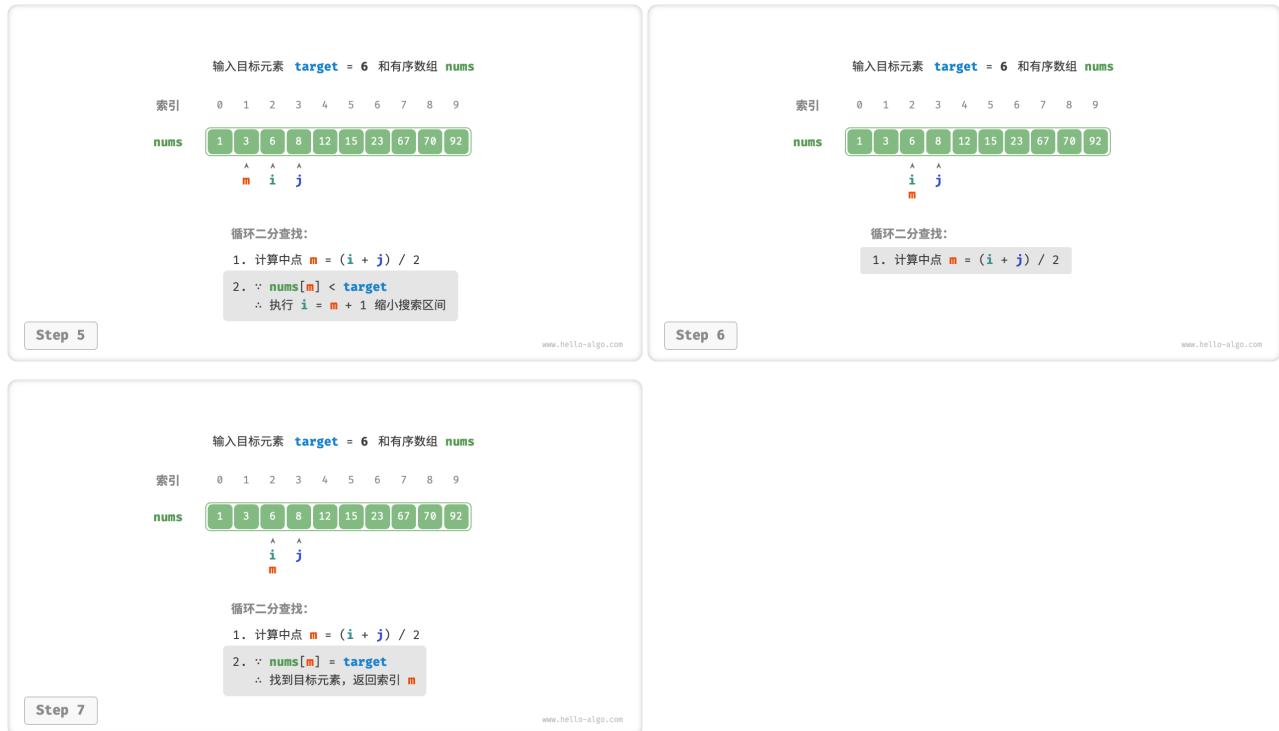


Figure 6-1. 二分查找步骤

二分查找在“双闭区间”表示下的代码如下所示。

```
# === File: binary_search.py ===
def binary_search(nums: list[int], target: int) -> int:
    """ 二分查找（双闭区间） """
    # 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
    i, j = 0, len(nums) - 1
    while i <= j:
        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
        elif nums[m] > target: # 此情况说明 target 在区间 [i, m-1] 中
            j = m - 1
        else:
            return m # 找到目标元素，返回其索引
    return -1 # 未找到目标元素，返回 -1
```

需要注意的是，当数组长度非常大时，加法  $i + j$  的结果可能会超出 `int` 类型的取值范围。在这种情况下，我们需要采用一种更安全的计算中点的方法。

```
# Python 中的数字理论上可以无限大（取决于内存大小）
# 因此无需考虑大数越界问题
```

### 6.1.2. 左闭右开实现

我们可以采用“左闭右开”的表示法，编写具有相同功能的二分查找代码。

```
# == File: binary_search.py ==
def binary_search1(nums: list[int], target: int) -> int:
    """ 二分查找（左闭右开） """
    # 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
    i, j = 0, len(nums)
    # 循环，当搜索区间为空时跳出（当 i = j 时为空）
    while i < j:
        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
        elif 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
    return -1 # 未找到目标元素，返回 -1
```

对比这两种代码写法，我们可以发现以下不同点：

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 $[0, n)$	$i = 0, j = n$	$i = m + 1, j = m$	$i = j$

在“双闭区间”表示法中，由于对左右两边界的定义相同，因此缩小区间的  $i$  和  $j$  的处理方法也是对称的，这样更不容易出错。因此，建议采用“双闭区间”的写法。

### 6.1.3. 复杂度分析

**时间复杂度  $O(\log n)$** ：其中  $n$  为数组长度；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。

### 6.1.4. 优点与局限性

二分查找效率很高，主要体现在：

- **二分查找的时间复杂度较低。** 对数阶在大数据量情况下具有显著优势。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  $\log_2 2^{20} = 20$  轮循环。

- **二分查找无需额外空间。**与哈希查找相比，二分查找更加节省空间。

然而，并非所有情况下都可使用二分查找，原因如下：

- **二分查找仅适用于有序数据。**若输入数据无序，为了使用二分查找而专门进行排序，得不偿失。因为排序算法的时间复杂度通常为  $O(n \log n)$ ，比线性查找和二分查找都更高。对于频繁插入元素的场景，为保持数组有序性，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。**二分查找需要跳跃式（非连续地）访问元素，而在链表中执行跳跃式访问的效率较低，因此不适合应用在链表或基于链表实现的数据结构。
- **小数据量下，线性查找性能更佳。**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，当数据量  $n$  较小时，线性查找反而比二分查找更快。

## 7. 散列表

### 7.1. 哈希表

哈希表通过建立「键 key」与「值 value」之间的映射，实现高效的元素查询。具体而言，我们向哈希表输入一个 key，则可以在  $O(1)$  时间内获取对应的 value。

以一个包含  $n$  个学生的数据库为例，每个学生都有“姓名 name”和“学号 id”两项数据。假如我们希望实现查询功能，例如“输入一个学号，返回对应的姓名”，则可以采用哈希表来实现。



Figure 7-1. 哈希表的抽象表示

除哈希表外，我们还可以使用数组或链表实现查询功能，各项操作的时间复杂度如下表所示。

在哈希表中增删查改的时间复杂度都是  $O(1)$ ，全面胜出！因此，哈希表常用于对查找效率要求较高的场景。

	数组	链表	哈希表
查找元素	$O(n)$	$O(n)$	$O(1)$
插入元素	$O(1)$	$O(1)$	$O(1)$
删除元素	$O(n)$	$O(n)$	$O(1)$

#### 7.1.1. 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.py ===
# 初始化哈希表
mapp: Dict = {}

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

# 查询操作
# 向哈希表输入键 key , 得到值 value
name: str = mapp[15937]

# 删除操作
# 在哈希表中删除键值对 (key, value)
mapp.pop(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.py ===
# 遍历哈希表
# 遍历键值对 key->value
for key, value in mapp.items():
    print(key, "->", value)
# 单独遍历键 key
for key in mapp.keys():
    print(key)
# 单独遍历值 value
for value in mapp.values():
    print(value)
```

### 7.1.2. 哈希函数

哈希表的底层实现为数组，同时可能包含链表、二叉树（红黑树）等数据结构，以提高查询性能（将在下节讨论）。

首先考虑最简单的情况，仅使用一个数组来实现哈希表。通常，我们将数组中的每个空位称为「桶 Bucket」，用于存储键值对。

我们将键值对 `key, value` 封装成一个类 `Entry`，并将所有 `Entry` 放入数组中。这样，数组中的每个 `Entry` 都具有唯一的索引。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」。

设哈希表的数组为 `buckets`，哈希函数为 `f(x)`，那么查询操作的步骤如下：

1. 输入 `key`，通过哈希函数计算出索引 `index`，即 `index = f(key)`；
2. 通过索引在数组中访问到键值对 `entry`，即 `entry = buckets[index]`，然后从 `entry` 中获取对应的 `value`；

以学生数据 `key 学号 -> value 姓名` 为例，我们可以设计如下哈希函数：

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

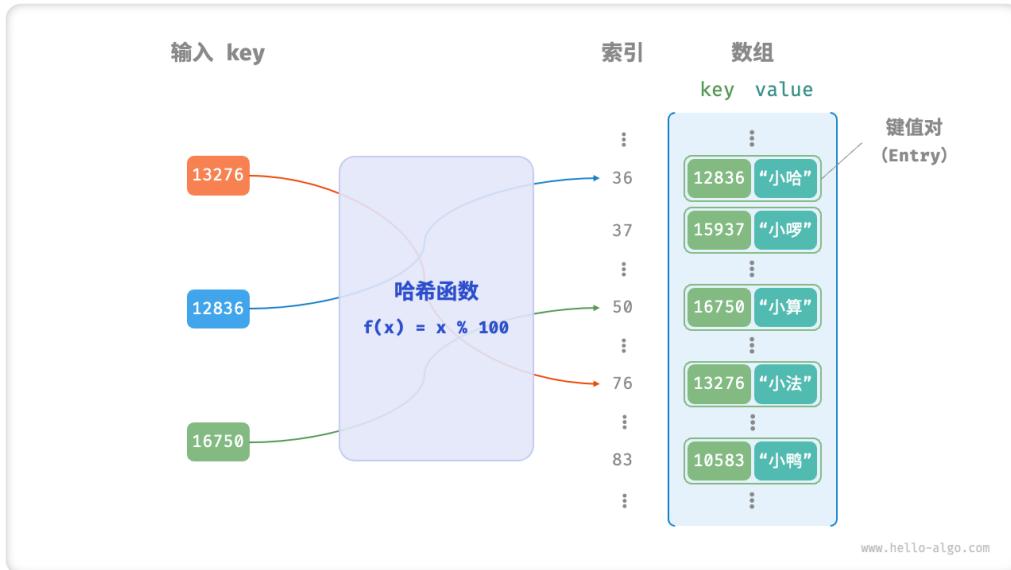


Figure 7-2. 哈希函数工作原理

```
# == File: array_hash_map.py ==
class Entry:
    """ 键值对 int->String"""

    def __init__(self, key: int, val: str):
        self.key = key
        self.val = val

class ArrayHashMap:
    """ 基于数组简易实现的哈希表"""

    def __init__(self):
        """ 构造方法"""
        # 初始化数组，包含 100 个桶
        self.buckets: list[Entry | None] = [None] * 100

    def hash_func(self, key: int) -> int:
        """ 哈希函数"""
        index: int = key % 100
        return index
```

```
def get(self, key: int) -> str:
    """ 查询操作"""
    index: int = self.hash_func(key)
    pair: Entry = self.buckets[index]
    if pair is None:
        return None
    return pair.val

def put(self, key: int, val: str) -> None:
    """ 添加操作"""
    pair = Entry(key, val)
    index: int = self.hash_func(key)
    self.buckets[index] = pair

def remove(self, key: int) -> None:
    """ 删除操作"""
    index: int = self.hash_func(key)
    # 置为 None , 代表删除
    self.buckets[index] = None

def entry_set(self) -> list[Entry]:
    """ 获取所有键值对"""
    result: list[Entry] = []
    for pair in self.buckets:
        if pair is not None:
            result.append(pair)
    return result

def key_set(self) -> list[int]:
    """ 获取所有键"""
    result: list[int] = []
    for pair in self.buckets:
        if pair is not None:
            result.append(pair.key)
    return result

def value_set(self) -> list[str]:
    """ 获取所有值"""
    result: list[str] = []
    for pair in self.buckets:
        if pair is not None:
            result.append(pair.val)
    return result

def print(self) -> None:
```

```
""" 打印哈希表"""
for pair in self.buckets:
    if pair is not None:
        print(pair.key, "->", pair.val)
```

### 7.1.3. 哈希冲突

细心的你可能已经注意到，在某些情况下，哈希函数  $f(x) = x$  可能无法正常工作。具体来说，当输入的 key 后两位相同时，哈希函数的计算结果也会相同，从而指向同一个 value。例如，查询学号为 12836 和 20336 的两个学生时，我们得到：

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

这两个学号指向了同一个姓名，这显然是错误的。我们把这种情况称为「哈希冲突 Hash Collision」。在后续章节中，我们将讨论如何解决哈希冲突的问题。

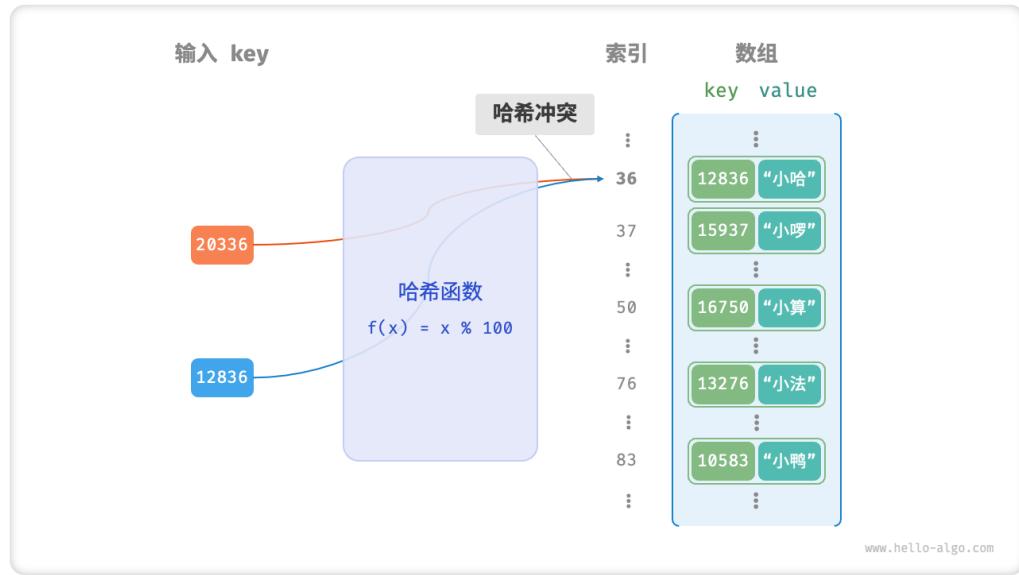


Figure 7-3. 哈希冲突示例

综上所述，一个优秀的哈希函数应具备以下特性：

- 尽可能减少哈希冲突的发生；
- 查询效率高且稳定，能够在绝大多数情况下达到  $O(1)$  时间复杂度；
- 较高的空间利用率，即使“键值对占用空间 / 哈希表总占用空间”比例最大化；

## 7.2. 哈希冲突

在理想情况下，哈希函数应为每个输入生成唯一的输出，实现 key 和 value 的一一对应。然而实际上，向哈希函数输入不同的 key 却产生相同输出的情况是存在的，这种现象被称为「哈希冲突 Hash Collision」。哈希冲突可能导致查询结果错误，从而严重影响哈希表的可用性。

那么，为何会出现哈希冲突呢？从本质上讲，由于哈希函数的输入空间通常远大于输出空间，因此多个输入产生相同输出的情况是不可避免的。例如，若输入空间为全体整数，而输出空间为固定大小的数组，则必然有多个整数映射至同一数组索引。

为了减轻哈希冲突，一方面，**可以通过扩大哈希表容量来降低冲突概率**。极端情况下，当输入空间和输出空间大小相等时，哈希表等同于数组，每个 key 都对应唯一的数组索引，可谓“大力出奇迹”。

另一方面，**可以考虑优化哈希表的表示以缓解哈希冲突**，常用方法包括「链式地址 Separate Chaining」和「开放寻址 Open Addressing」。

### 7.2.1. 哈希表扩容

哈希函数的最后一步通常是对桶数量  $n$  取余，作用是将哈希值映射到桶索引范围，从而将 key 放入对应的桶中。当哈希表容量越大（即  $n$  越大）时，多个 key 被分配到同一个桶中的概率就越低，冲突就越少。

因此，**当哈希表内的冲突总体较为严重时，编程语言通常通过扩容哈希表来缓解冲突**。类似于数组扩容，哈希表扩容需将所有键值对从原哈希表迁移至新哈希表，开销较大。

编程语言通常使用「负载因子 Load Factor」来衡量哈希冲突的严重程度，**定义为哈希表中元素数量除以桶数量**，常作为哈希表扩容的触发条件。在 Java 中，当负载因子  $> 0.75$  时，系统会将 HashMap 容量扩展为原先的 2 倍。

### 7.2.2. 链式地址

在原始哈希表中，每个桶仅能存储一个键值对。**链式地址将单个元素转换为链表，将键值对作为链表节点，将所有发生冲突的键值对都存储在同一链表中**。



Figure 7-4. 链式地址

链式地址下，哈希表的操作方法包括：

- **查询元素：**输入  $key$ ，经过哈希函数得到数组索引，即可访问链表头节点，然后遍历链表并对比  $key$  以查找目标键值对。
- **添加元素：**先通过哈希函数访问链表头节点，然后将节点（即键值对）添加到链表中。
- **删除元素：**根据哈希函数的结果访问链表头部，接着遍历链表以查找目标节点，并将其删除。

尽管链式地址法解决了哈希冲突问题，但仍存在一些局限性，包括：

- **占用空间增大：**由于链表或二叉树包含节点指针，相比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；

为了提高操作效率，可以将链表转换为「AVL 树」或「红黑树」，将查询操作的时间复杂度优化至  $O(\log n)$ 。

### 7.2.3. 开放寻址

「开放寻址」方法不引入额外的数据结构，而是通过“多次探测”来解决哈希冲突，探测方主要包括线性探测、平方探测、多次哈希。

#### 线性探测

「线性探测」采用固定步长的线性查找来解决哈希冲突。

**插入元素：**若出现哈希冲突，则从冲突位置向后线性遍历（步长通常为 1），直至找到空位，将元素插入其中。

**查找元素：**在出现哈希冲突时，使用相同步长进行线性查找，可能遇到以下两种情况。

1. 找到对应元素，返回 value 即可；
2. 若遇到空位，说明目标键值对不在哈希表中；

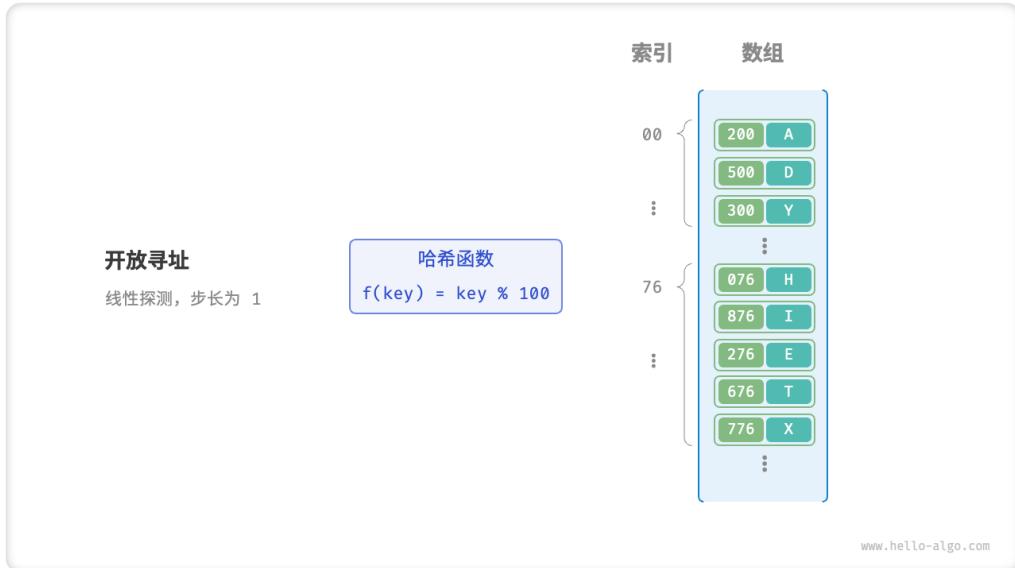


Figure 7-5. 线性探测

线性探测存在以下缺陷：

- **不能直接删除元素。**删除元素会在数组内产生一个空位，查找其他元素时，该空位可能导致程序误判元素不存在（即上述第 2. 种情况）。因此，需要借助一个标志位来标记已删除元素。
- **容易产生聚集。**数组内连续被占用位置越长，这些连续位置发生哈希冲突的可能性越大，进一步促使这一位置的“聚堆生长”，最终导致增删查改操作效率降低。

### 多次哈希

顾名思义，「多次哈希」方法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：**在相同的哈希函数顺序下进行查找，存在以下两种情况：

1. 如果找到目标元素，则返回之；
2. 若遇到空位或已尝试所有哈希函数，则说明哈希表中不存在该元素；

与线性探测相比，多次哈希方法不易产生聚集，但多个哈希函数会增加额外的计算量。



#### 哈希表设计方案

Java 采用「链式地址」。自 JDK 1.8 以来，当 HashMap 内数组长度大于 64 且链表长度大于 8 时，链表会被转换为「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。

Golang 采用「链式地址」。Go 规定每个桶最多存储 8 个键值对，超出容量则连接一个溢出桶；当溢出桶过多时，会执行一次特殊的等量扩容操作，以确保性能。

### 7.3. 小结

- 哈希表能够在  $O(1)$  时间内将键 key 映射到值 value，效率非常高。
- 常见的哈希表操作包括查询、添加与删除键值对、遍历键值对等。
- 哈希函数将 key 映射为数组索引（桶），以便访问对应的值 value。
- 两个不同的 key 可能在经过哈希函数后得到相同的索引，导致查询结果出错，这种现象被称为哈希冲突。
- 缓解哈希冲突的方法主要有扩容哈希表和优化哈希表的表示方法。
- 负载因子定义为哈希表中元素数量除以桶数量，反映了哈希冲突的严重程度，常用作触发哈希表扩容的条件。与数组扩容类似，哈希表扩容操作也会产生较大的开销。
- 链式地址通过将单个元素转化为链表，将所有冲突元素存储在同一个链表中，从而解决哈希冲突。然而，过长的链表会降低查询效率，可以通过将链表转换为 AVL 树或红黑树来改善。
- 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，相对线性探测不易产生聚集，但多个哈希函数增加了计算量。
- 不同编程语言采取了不同的哈希表实现策略。例如，Java 的 HashMap 使用链式地址，而 Python 的 Dict 采用开放寻址。

# 8. 树

## 8.1. 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。与链表类似，二叉树的基本单元是节点，每个节点包含一个「值」和两个「指针」。

```
class TreeNode:
    """ 二叉树节点类 """
    def __init__(self, val: int):
        self.val: int = val           # 节点值
        self.left: Optional[TreeNode] = None  # 左子节点指针
        self.right: Optional[TreeNode] = None # 右子节点指针
```

节点的两个指针分别指向「左子节点」和「右子节点」，同时该节点被称为这两个子节点的「父节点」。当给定一个二叉树的节点时，我们将该节点的左子节点及其以下节点形成的树称为该节点的「左子树」，同理可得「右子树」。

在二叉树中，除叶节点外，其他所有节点都包含子节点和非空子树。例如，在以下示例中，若将“节点 2”视为父节点，则其左子节点和右子节点分别是“节点 4”和“节点 5”，左子树是“节点 4 及其以下节点形成的树”，右子树是“节点 5 及其以下节点形成的树”。

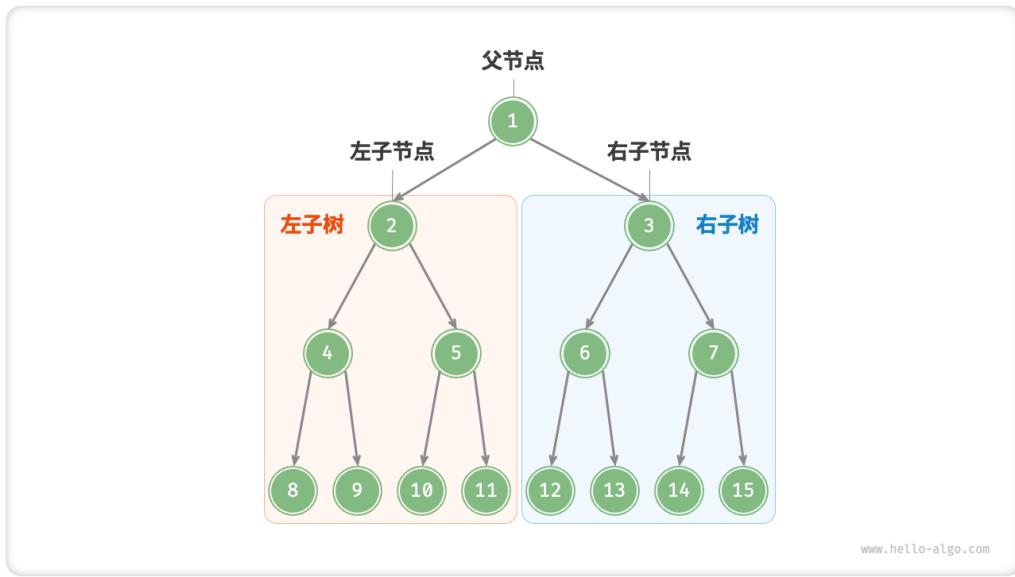


Figure 8-1. 父节点、子节点、子树

### 8.1.1. 二叉树常见术语

二叉树涉及的术语较多，建议尽量理解并记住。

- 「根节点 Root Node」：位于二叉树顶层的节点，没有父节点；
- 「叶节点 Leaf Node」：没有子节点的节点，其两个指针均指向 null；
- 节点的「层 Level」：从顶至底递增，根节点所在层为 1；
- 节点的「度 Degree」：节点的子节点的数量。在二叉树中，度的范围是 0, 1, 2；
- 「边 Edge」：连接两个节点的线段，即节点指针；
- 二叉树的「高度」：从根节点到最远叶节点所经过的边的数量；
- 节点的「深度 Depth」：从根节点到该节点所经过的边的数量；
- 节点的「高度 Height」：从最远叶节点到该节点所经过的边的数量；

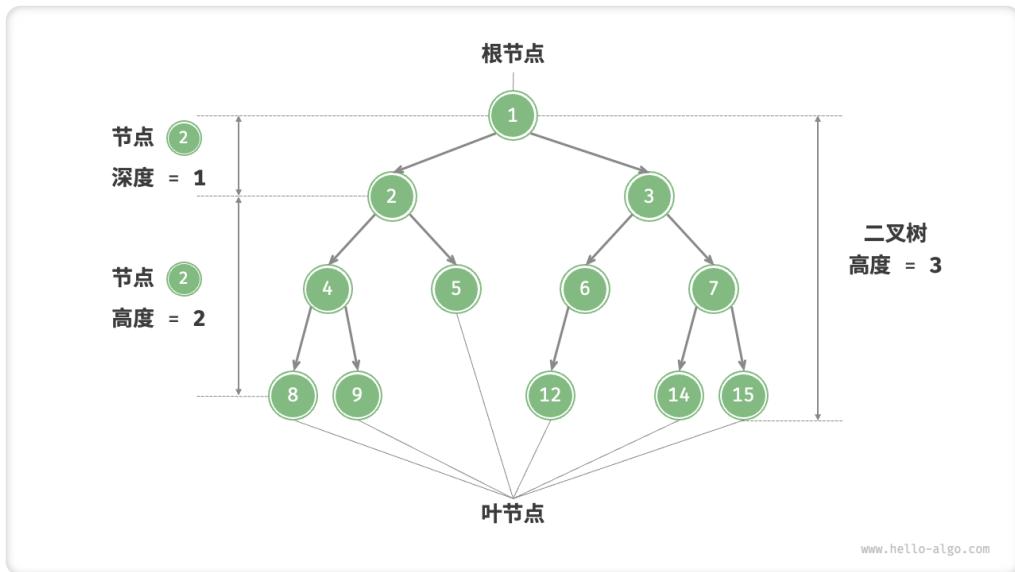


Figure 8-2. 二叉树的常用术语



### 高度与深度的定义

请注意，我们通常将「高度」和「深度」定义为“走过边的数量”，但有些题目或教材可能会将其定义为“走过节点的数量”。在这种情况下，高度和深度都需要加 1。

#### 8.1.2. 二叉树基本操作

**初始化二叉树。**与链表类似，首先初始化节点，然后构建引用指向（即指针）。

```
# === File: binary_tree.py ===
# 初始化二叉树
# 初始化节点
n1 = TreeNode(val=1)
n2 = TreeNode(val=2)
n3 = TreeNode(val=3)
n4 = TreeNode(val=4)
```

```
n5 = TreeNode(val=5)
# 构建引用指向（即指针）
n1.left = n2
n1.right = n3
n2.left = n4
n2.right = n5
```

**插入与删除节点。**与链表类似，通过修改指针来实现插入与删除节点。

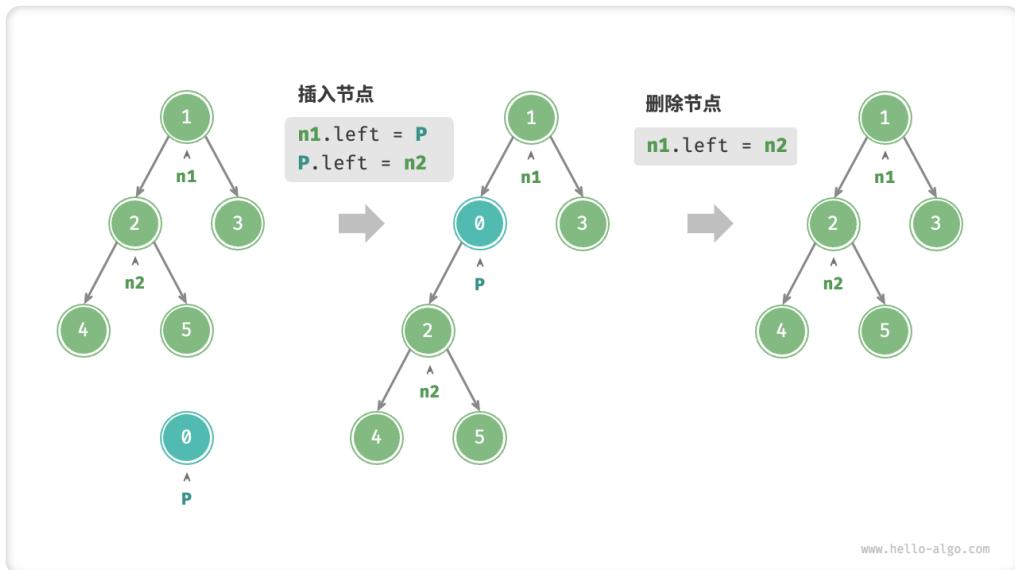


Figure 8-3. 在二叉树中插入与删除节点

```
# === File: binary_tree.py ===
# 插入与删除节点
p = TreeNode(0)
# 在 n1 -> n2 中间插入节点 p
n1.left = p
p.left = n2
# 删除节点 p
n1.left = n2
```



需要注意的是，插入节点可能会改变二叉树的原有逻辑结构，而删除节点通常意味着删除该节点及其所有子树。因此，在二叉树中，插入与删除操作通常是由一套操作配合完成的，以实现有实际意义的操作。

### 8.1.3. 常见二叉树类型

#### 完美二叉树

「完美二叉树 Perfect Binary Tree」除了最底层外，其余所有层的节点都被完全填满。在完美二叉树中，叶节点的度为 0，其余所有节点的度都为 2；若树高度为  $h$ ，则节点总数为  $2^{h+1} - 1$ ，呈现标准的指数级关系，反映了自然界中常见的细胞分裂现象。



在中文社区中，完美二叉树常被称为「满二叉树」，请注意区分。

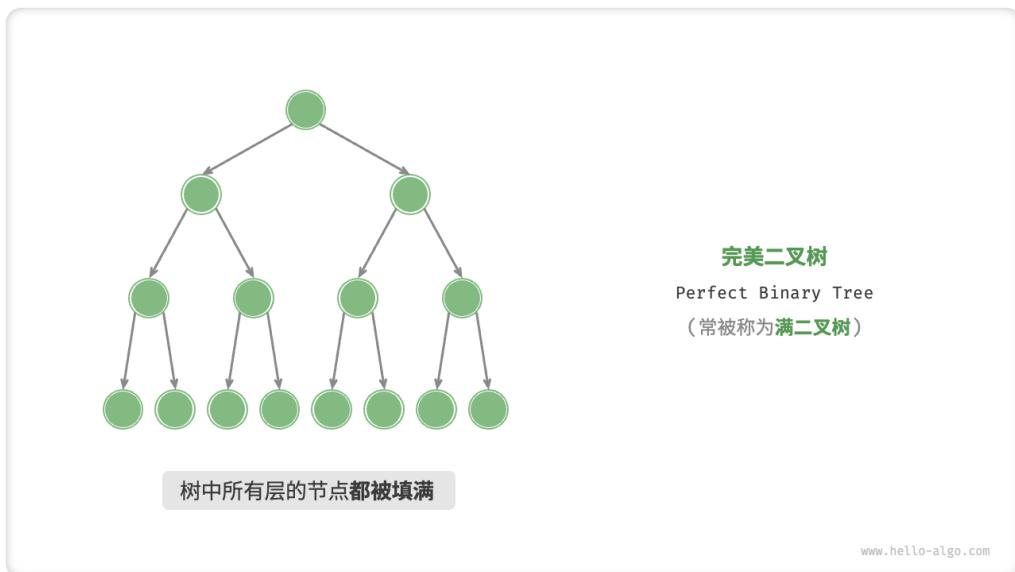


Figure 8-4. 完美二叉树

#### 完全二叉树

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

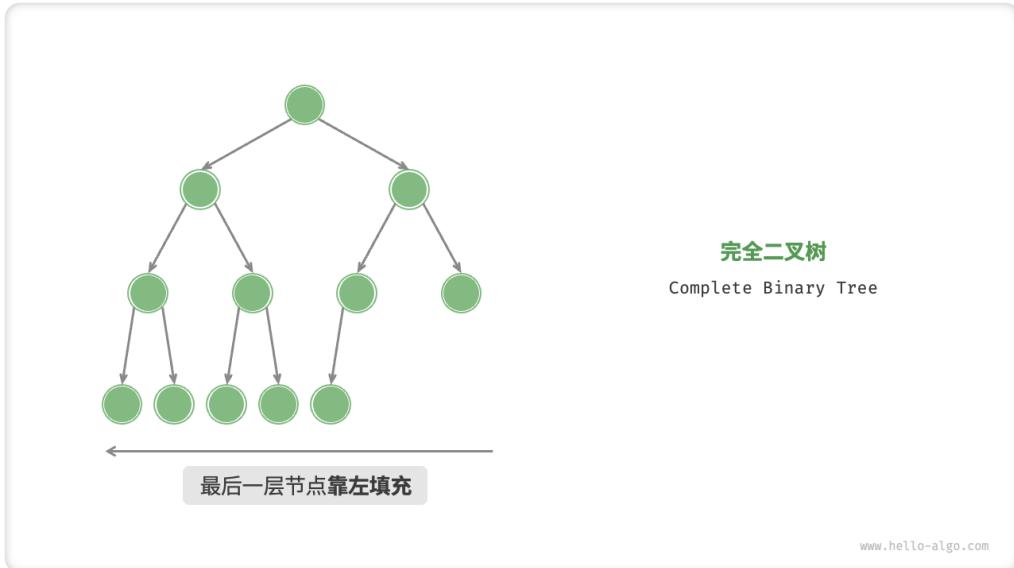


Figure 8-5. 完全二叉树

### 完满二叉树

「完满二叉树 Full Binary Tree」除了叶节点之外，其余所有节点都有两个子节点。

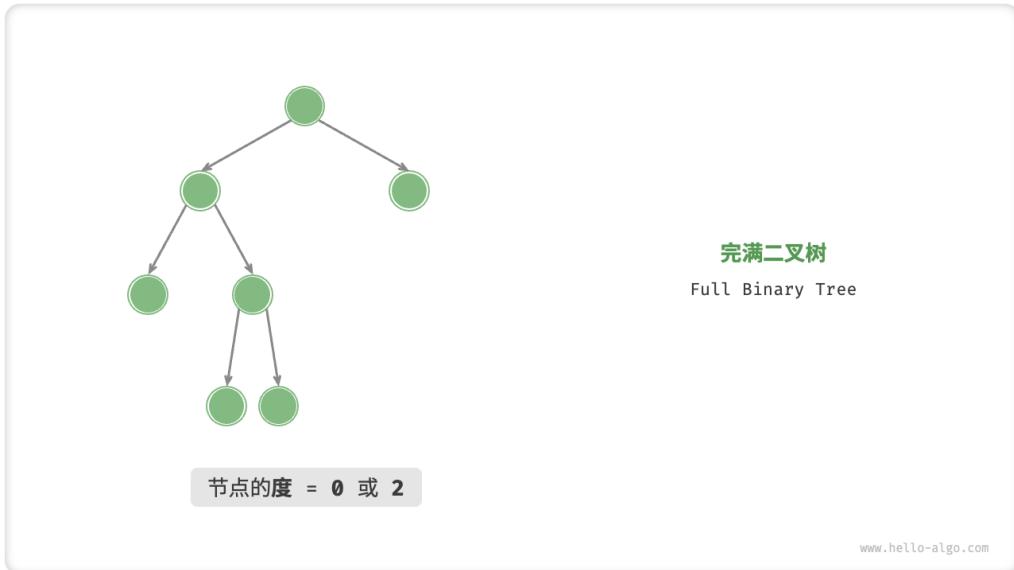


Figure 8-6. 完满二叉树

### 平衡二叉树

「平衡二叉树 Balanced Binary Tree」中任意节点的左子树和右子树的高度之差的绝对值不超过 1。

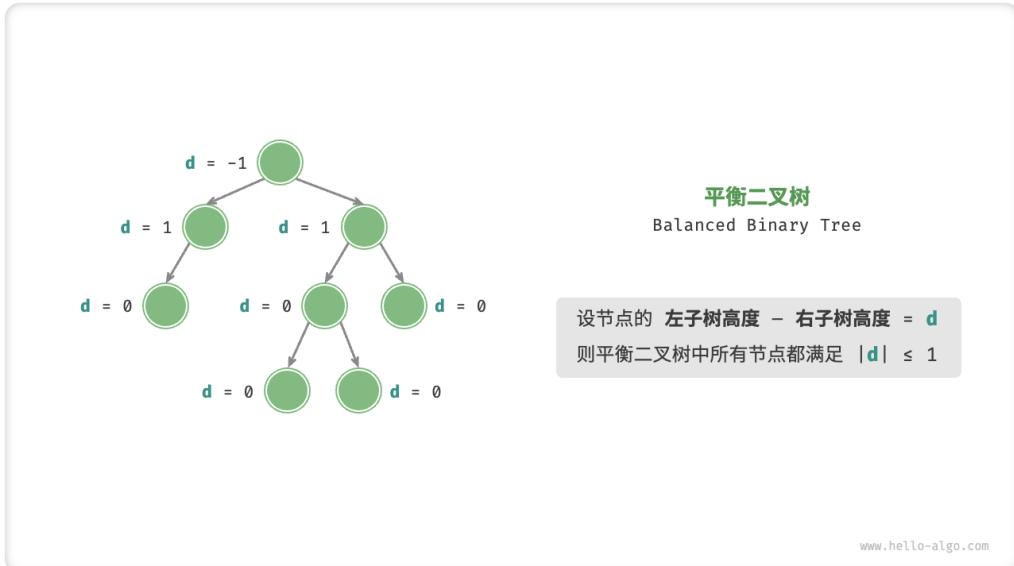


Figure 8-7. 平衡二叉树

#### 8.1.4. 二叉树的退化

当二叉树的每层节点都被填满时，达到「完美二叉树」；而当所有节点都偏向一侧时，二叉树退化为「链表」。

- 完美二叉树是理想情况，可以充分发挥二叉树“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  $O(n)$ ；

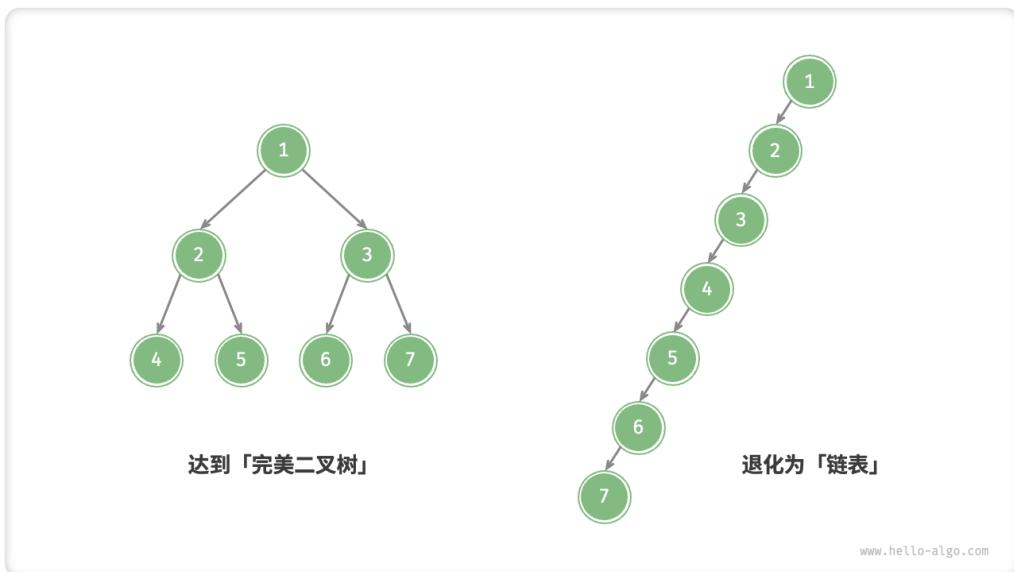


Figure 8-8. 二叉树的最佳与最差结构

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

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

## 8.2. 二叉树遍历

从物理结构的角度来看，树是一种基于链表的数据结构，因此其遍历方式是通过指针逐个访问节点。然而，树是一种非线性数据结构，这使得遍历树比遍历链表更加复杂，需要借助搜索算法来实现。

二叉树常见的遍历方式包括层序遍历、前序遍历、中序遍历和后序遍历等。

### 8.2.1. 层序遍历

「层序遍历 Level-Order Traversal」从顶部到底部逐层遍历二叉树，并在每一层按照从左到右的顺序访问节点。

层序遍历本质上属于「广度优先搜索 Breadth-First Traversal」，它体现了一种“一圈一圈向外扩展”的逐层搜索方式。

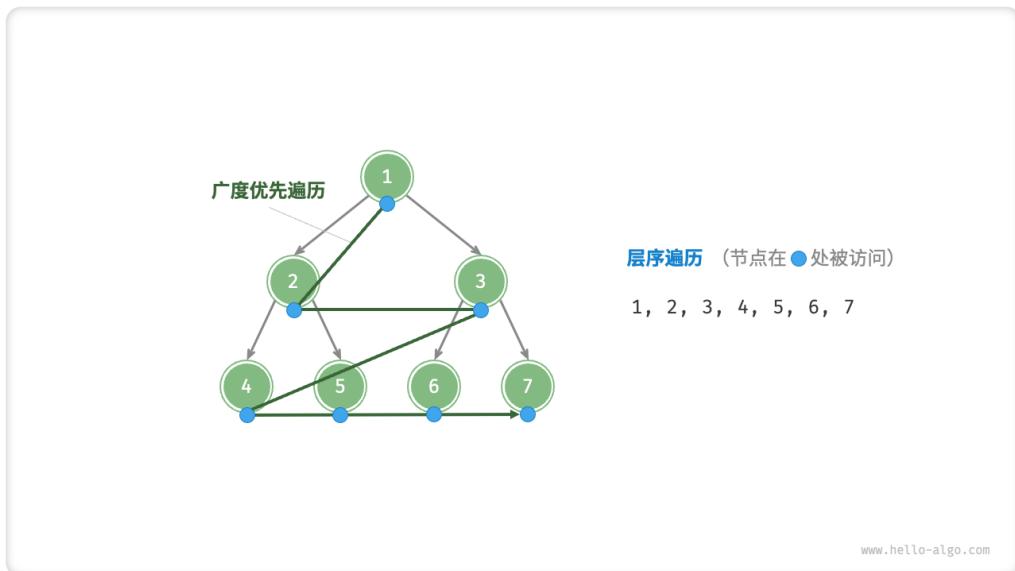


Figure 8-9. 二叉树的层序遍历

## 算法实现

广度优先遍历通常借助「队列」来实现。队列遵循“先进先出”的规则，而广度优先遍历则遵循“逐层推进”的规则，两者背后的思想是一致的。

```
# === File: binary_tree_bfs.py ===
def level_order(root: TreeNode | None) -> list[int]:
    """ 层序遍历 """
    # 初始化队列，加入根节点
    queue: deque[TreeNode] = deque()
    queue.append(root)
    # 初始化一个列表，用于保存遍历序列
    res: list[int] = []
    while queue:
        node: TreeNode = queue.popleft() # 队列出队
        res.append(node.val) # 保存节点值
        if node.left is not None:
            queue.append(node.left) # 左子节点入队
        if node.right is not None:
            queue.append(node.right) # 右子节点入队
    return res
```

## 复杂度分析

**时间复杂度：**所有节点被访问一次，使用  $O(n)$  时间，其中  $n$  为节点数量。

**空间复杂度：**在最差情况下，即满二叉树时，遍历到最底层之前，队列中最多同时存在  $\frac{n+1}{2}$  个节点，占用  $O(n)$  空间。

### 8.2.2. 前序、中序、后序遍历

相应地，前序、中序和后序遍历都属于「深度优先遍历 Depth-First Traversal」，它体现了一种“先走到尽头，再回溯继续”的遍历方式。

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，在这个过程中，在每个节点都会遇到三个位置，分别对应前序遍历、中序遍历和后序遍历。

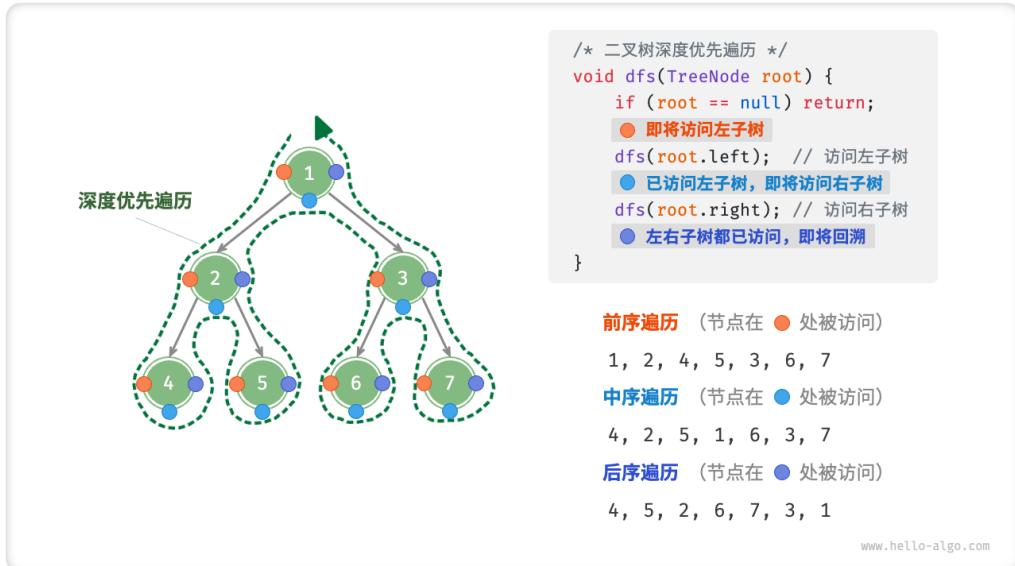


Figure 8-10. 二叉搜索树的前、中、后序遍历

位置	含义	此处访问节点时对应
橙色圆圈处	刚进入此节点, 即将访问该节点的左子树	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.py ===
def pre_order(root: TreeNode | None) -> None:
    """ 前序遍历 """
    if root is None:
        return
    # 访问优先级: 根节点 -> 左子树 -> 右子树
    res.append(root.val)
    pre_order(root.left)
    pre_order(root.right)

def in_order(root: TreeNode | None) -> None:
    """ 中序遍历 """
    if root is None:
        return
    # 访问优先级: 左子树 -> 根节点 -> 右子树
    in_order(root.left)

```

```
res.append(root.val)
in_order(root=root.right)

def post_order(root: TreeNode | None) -> None:
    """ 后序遍历 """
    if root is None:
        return
    # 访问优先级: 左子树 -> 右子树 -> 根节点
    post_order(root=root.left)
    post_order(root=root.right)
    res.append(root.val)
```



我们也可以仅基于循环实现前、中、后序遍历，有兴趣的同学可以自行实现。

递归过程可分为“递”和“归”两个相反的部分。“递”表示开启新方法，程序在此过程中访问下一个节点；“归”表示函数返回，代表该节点已经访问完毕。如下图所示，为前序遍历二叉树的递归过程。

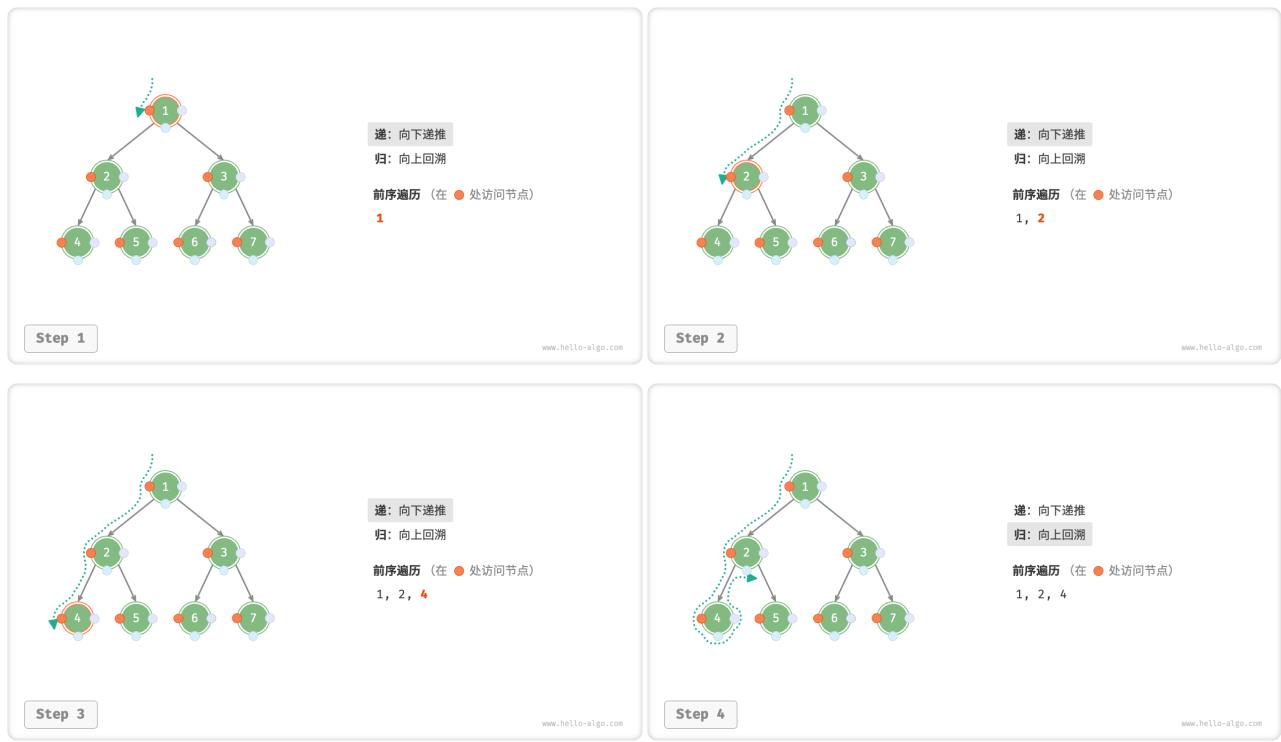




Figure 8-11. 前序遍历的递归过程

## 复杂度分析

**时间复杂度：**所有节点被访问一次，使用  $O(n)$  时间，其中  $n$  为节点数量。

**空间复杂度：**在最差情况下，即树退化为链表时，递归深度达到  $n$ ，系统占用  $O(n)$  栈帧空间。

### 8.3. 二叉树数组表示

在链表表示下，二叉树的存储单元为节点 `TreeNode`，节点之间通过指针相连接。在上节中，我们学习了在链表表示下的二叉树的各项基本操作。

那么，能否用「数组」来表示二叉树呢？答案是肯定的。

#### 8.3.1. 表示完美二叉树

先分析一个简单案例，给定一个完美二叉树，我们将节点按照层序遍历的顺序编号（从 0 开始），此时每个节点都对应唯一的索引。

根据层序遍历的特性，我们可以推导出父节点索引与子节点索引之间的“映射公式”：若节点的索引为  $i$ ，则该节点的左子节点索引为  $2i + 1$ ，右子节点索引为  $2i + 2$ 。

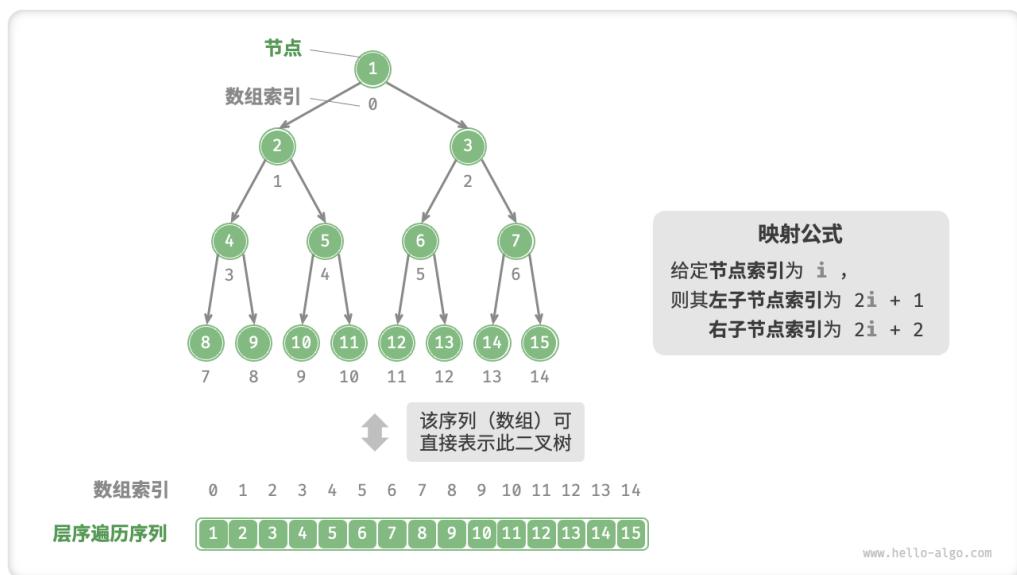


Figure 8-12. 完美二叉树的数组表示

**映射公式的作用相当于链表中的指针。**如果我们将节点按照层序遍历的顺序存储在一个数组中，那么对于数组中的任意节点，我们都可以通过映射公式来访问其子节点。

#### 8.3.2. 表示任意二叉树

然而，完美二叉树只是一个特例。在二叉树的中间层，通常存在许多 null，而层序遍历序列并不包含这些 null。我们无法仅凭该序列来推测 null 的数量和分布位置，这意味着存在多种二叉树结构都符合该层序遍历序列。显然在这种情况下，上述的数组表示方法已经失效。

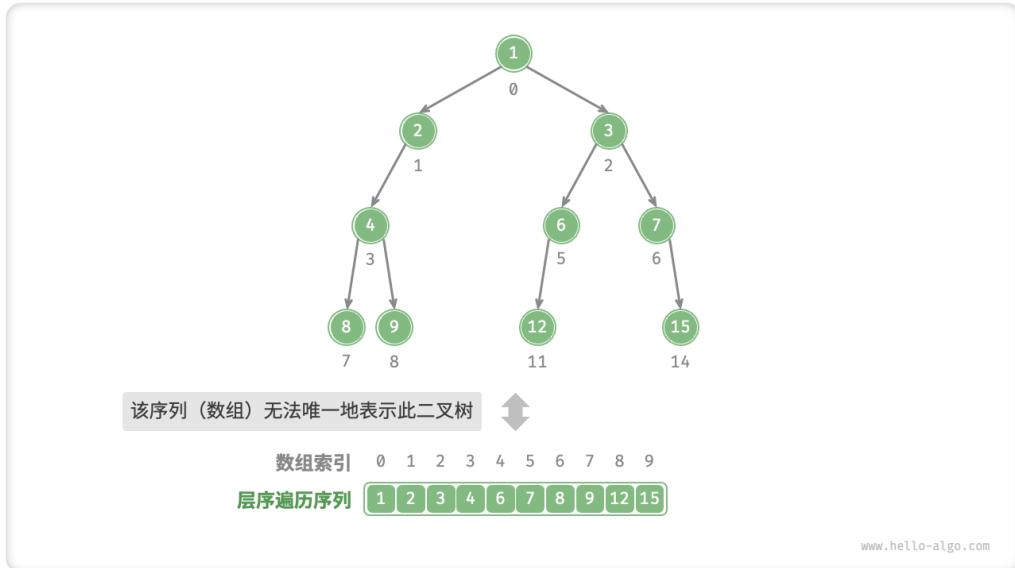


Figure 8-13. 层序遍历序列对应多种二叉树可能性

为了解决此问题，我们可以考虑在层序遍历序列中显式地写出所有 null。如下图所示，这样处理后，层序遍历序列就可以唯一表示二叉树了。

```
# 二叉树的数组表示
# 直接使用 None 来表示空位
tree = [1, 2, 3, 4, None, 6, 7, 8, 9, None, None, 12, None, None, 15]
```

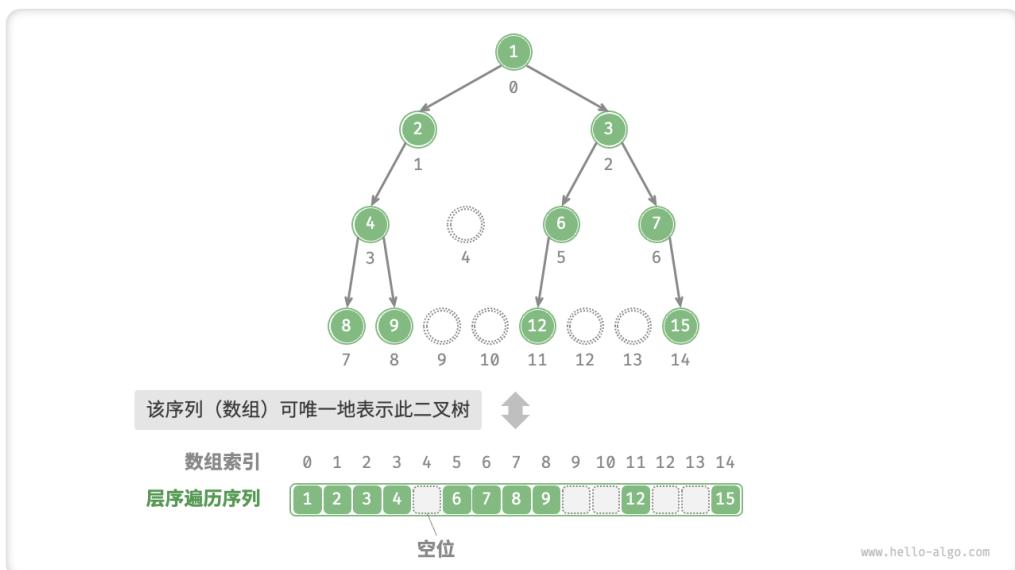


Figure 8-14. 任意类型二叉树的数组表示

### 8.3.3. 优势与局限性

二叉树的数组表示存在以下优点：

- 数组存储在连续的内存空间中，缓存友好，访问与遍历速度较快；
- 不需要存储指针，比较节省空间；
- 允许随机访问节点；

然而，数组表示也具有一些局限性：

- 数组存储需要连续内存空间，因此不适合存储数据量过大的树。
- 增删节点需要通过数组插入与删除操作实现，效率较低；
- 当二叉树中存在大量 null 时，数组中包含的节点数据比重较低，空间利用率较低。

完全二叉树非常适合使用数组来表示。回顾完全二叉树的定义，null 只出现在最底层且靠右的位置，这意味着所有 null 一定出现在层序遍历序列的末尾。因此，在使用数组表示完全二叉树时，可以省略存储所有 null。

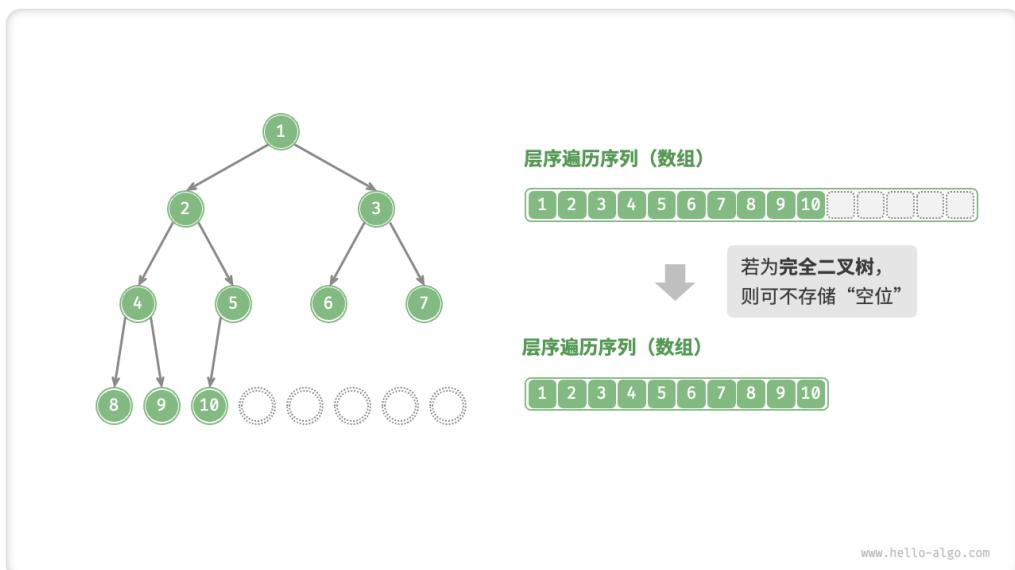


Figure 8-15. 完全二叉树的数组表示

### 8.4. 二叉搜索树

「二叉搜索树 Binary Search Tree」满足以下条件：

1. 对于根节点，左子树中所有节点的值  $<$  根节点的值  $<$  右子树中所有节点的值；
2. 任意节点的左、右子树也是二叉搜索树，即同样满足条件 1.；

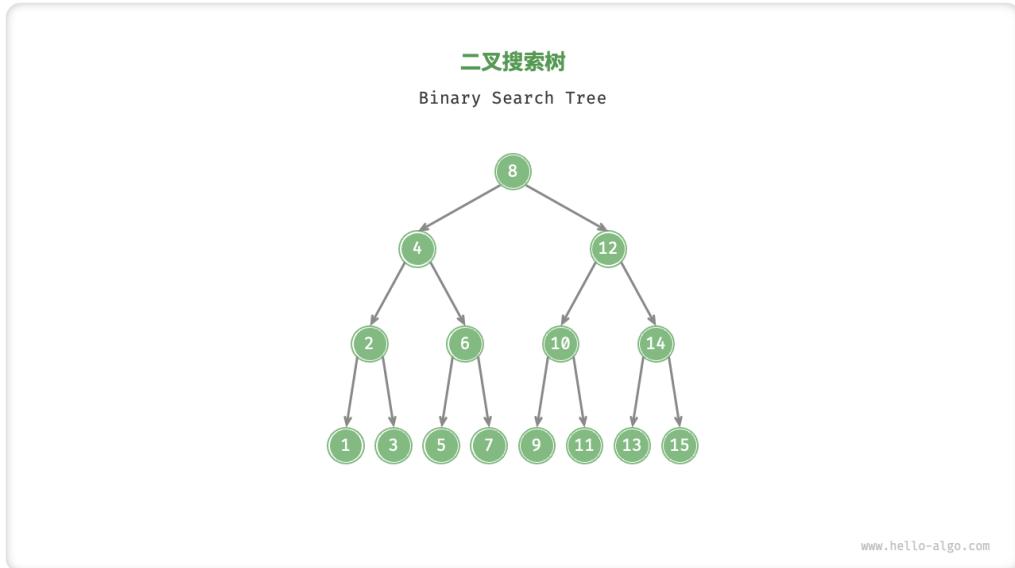


Figure 8-16. 二叉搜索树

#### 8.4.1. 二叉搜索树的操作

##### 查找节点

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

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





Figure 8-17. 二叉搜索树查找节点示例

二叉搜索树的查找操作与二分查找算法的工作原理一致，都是每轮排除一半情况。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.py ==
def search(self, num: int) -> TreeNode | None:
    """ 查找节点 """
    cur: TreeNode | None = self.__root
    # 循环查找，越过叶节点后跳出
    while cur is not None:
        # 目标节点在 cur 的右子树中
        if cur.val < num:
            cur = cur.right
        # 目标节点在 cur 的左子树中
        elif cur.val > num:
            cur = cur.left
        # 找到目标节点，跳出循环
        else:
            break
    return cur
```

## 插入节点

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

1. **查找插入位置：**与查找操作相似，从根节点出发，根据当前节点值和 `num` 的大小关系循环向下搜索，直到越过叶节点（遍历至 `null`）时跳出循环；
2. **在该位置插入节点：**初始化节点 `num`，将该节点置于 `null` 的位置；

二叉搜索树不允许存在重复节点，否则将违反其定义。因此，若待插入节点在树中已存在，则不执行插入，直接返回。

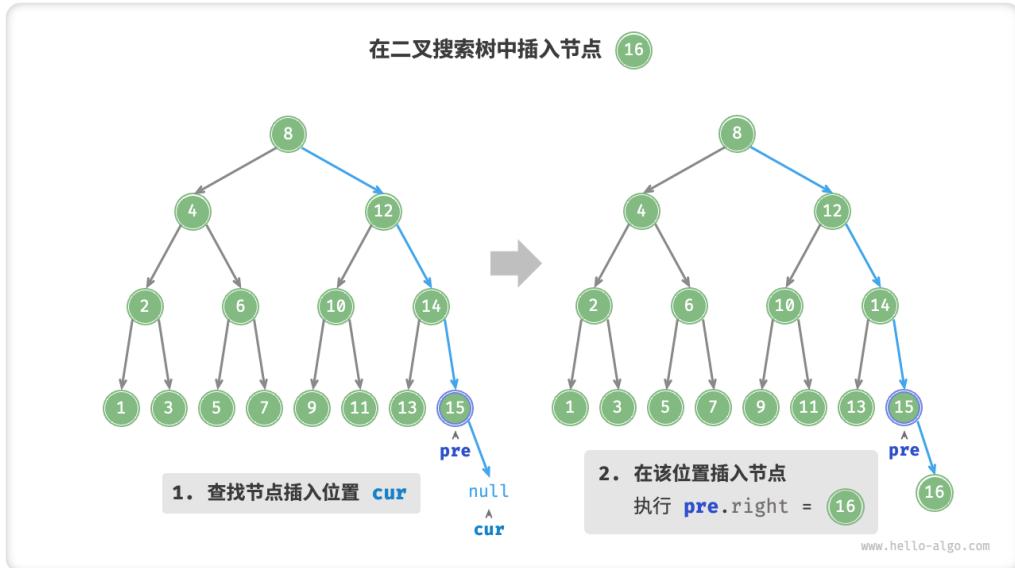


Figure 8-18. 在二叉搜索树中插入节点

```
# == File: binary_search_tree.py ==
def insert(self, num: int) -> None:
    """ 插入节点 """
    # 若树为空, 直接提前返回
    if self.__root is None:
        return

    # 循环查找, 越过叶节点后跳出
    cur, pre = self.__root, None
    while cur is not None:
        # 找到重复节点, 直接返回
        if cur.val == num:
            return
        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

    # 插入节点 val
    node = TreeNode(num)
    if pre.val < num:
        pre.right = node
    else:
        pre.left = node
```

为了插入节点，我们需要利用辅助节点 `pre` 保存上一轮循环的节点，这样在遍历至 `null` 时，我们可以获取到其父节点，从而完成节点插入操作。

与查找节点相同，插入节点使用  $O(\log n)$  时间。

### 删除节点

与插入节点类似，我们需要在删除操作后维持二叉搜索树的“左子树 < 根节点 < 右子树”的性质。首先，我们需要在二叉树中执行查找操作，获取待删除节点。接下来，根据待删除节点的子节点数量，删除操作需分为三种情况：

当待删除节点的子节点数量 = 0 时，表示待删除节点是叶节点，可以直接删除。

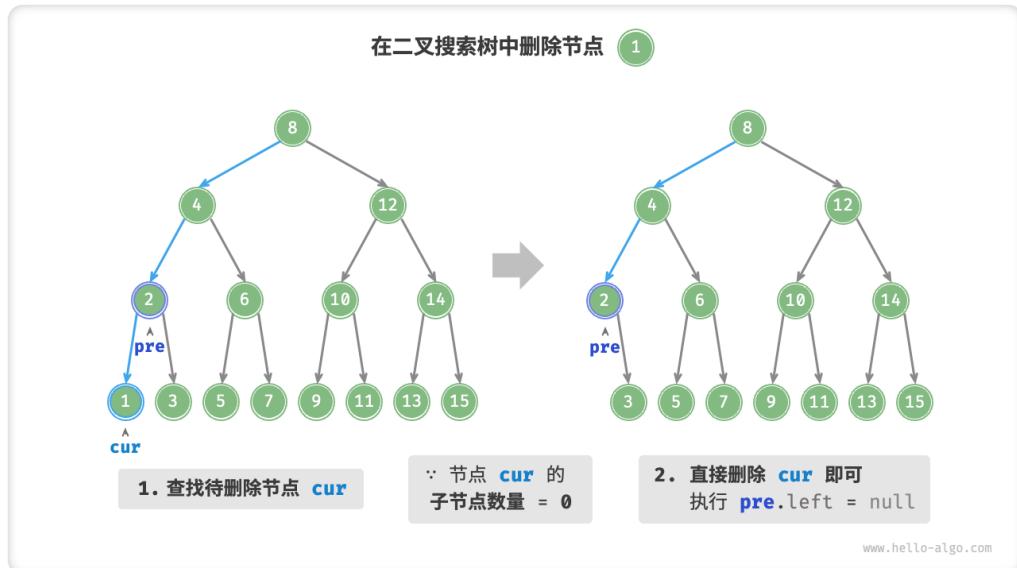


Figure 8-19. 在二叉搜索树中删除节点（度为 0）

当待删除节点的子节点数量 = 1 时，将待删除节点替换为其子节点即可。

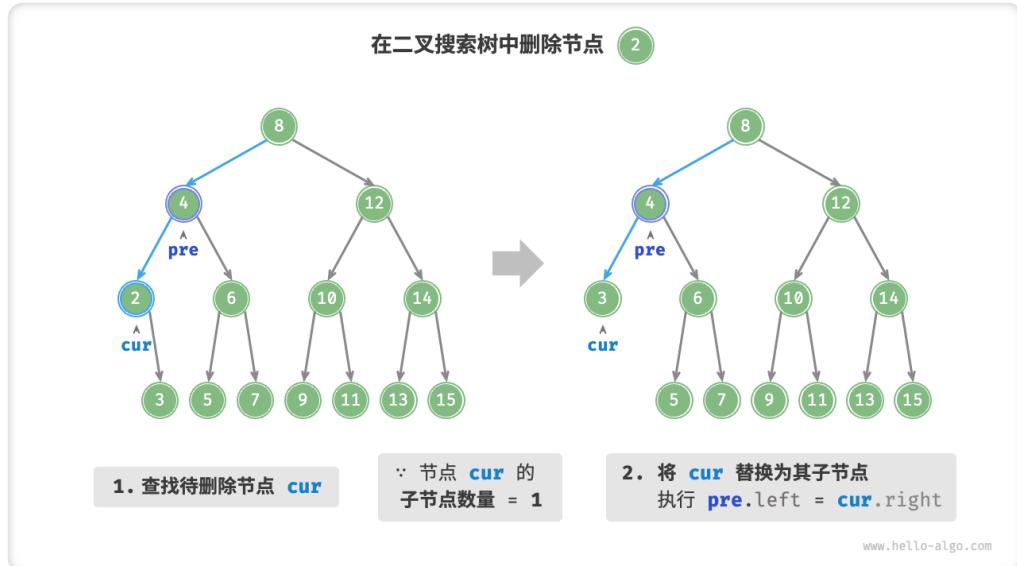


Figure 8-20. 在二叉搜索树中删除节点（度为 1）

当待删除节点的子节点数量 = 2 时，删除操作分为三步：

1. 找到待删除节点在“中序遍历序列”中的下一个节点，记为 `tmp`；
2. 在树中递归删除节点 `tmp`；
3. 用 `tmp` 的值覆盖待删除节点的值；

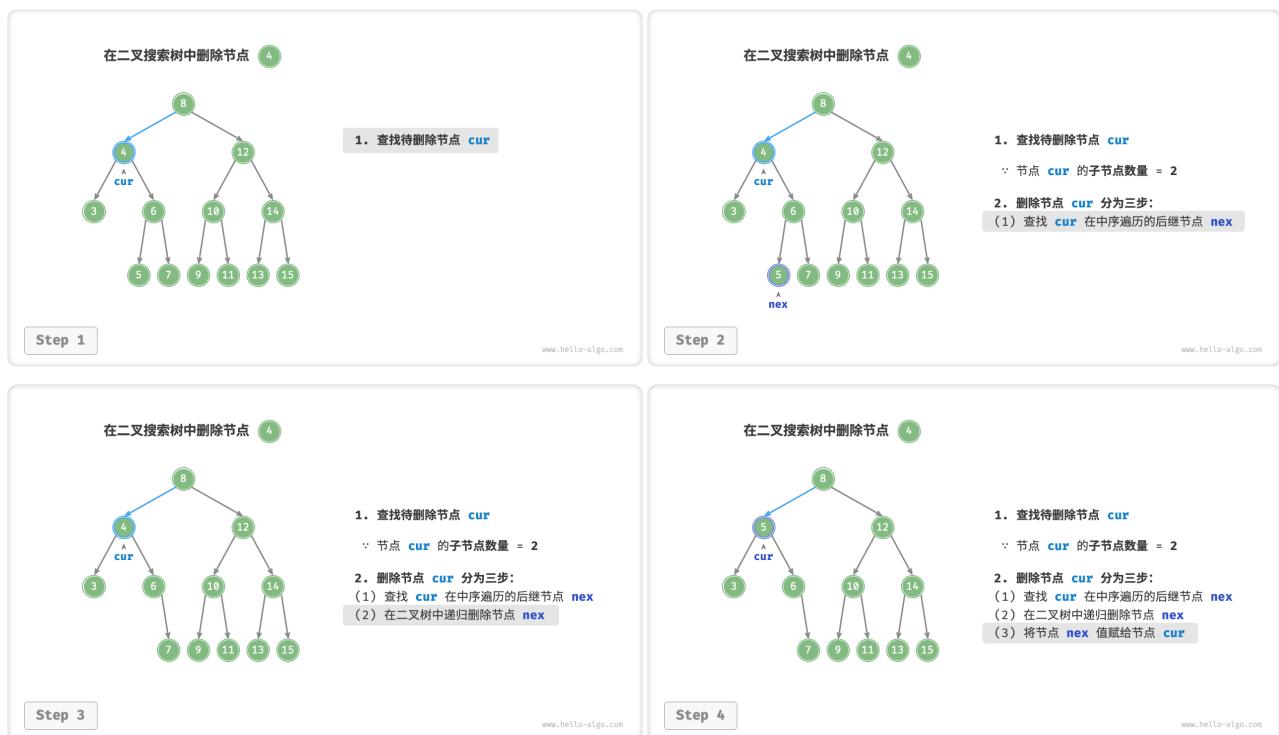


Figure 8-21. 二叉搜索树删除节点示例

删除节点操作同样使用  $O(\log n)$  时间，其中查找待删除节点需要  $O(\log n)$  时间，获取中序遍历后继节点需要  $O(\log n)$  时间。

```
# == File: binary_search_tree.py ==
def remove(self, num: int) -> None:
    """ 删除节点 """
    # 若树为空，直接提前返回
    if self._root is None:
        return

    # 循环查找，越过叶节点后跳出
    cur, pre = self._root, None
    while cur is not None:
        # 找到待删除节点，跳出循环
        if cur.val == num:
            break
        pre = cur
        # 待删除节点在 cur 的右子树中
        if cur.val < num:
            cur = cur.right
        # 待删除节点在 cur 的左子树中
        else:
            cur = cur.left
    # 若无待删除节点，则直接返回
    if cur is None:
        return

    # 子节点数量 = 0 or 1
    if cur.left is None or cur.right is None:
        # 当子节点数量 = 0 / 1 时，child = null / 孩子节点
        child = cur.left or cur.right
        # 删除节点 cur
        if pre.left == cur:
            pre.left = child
        else:
            pre.right = child
    # 子节点数量 = 2
    else:
        # 获取中序遍历中 cur 的下一个节点
        tmp: TreeNode = cur.right
        while tmp.left is not None:
            tmp = tmp.left
        # 递归删除节点 tmp
        self.remove(tmp.val)
        # 用 tmp 覆盖 cur
        cur.val = tmp.val
```

## 排序

我们知道，二叉树的中序遍历遵循“左 → 根 → 右”的遍历顺序，而二叉搜索树满足“左子节点 < 根节点 < 右子节点”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节点，从而得出一个重要性质：二叉搜索树的中序遍历序列是升序的。

利用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，无需额外排序，非常高效。

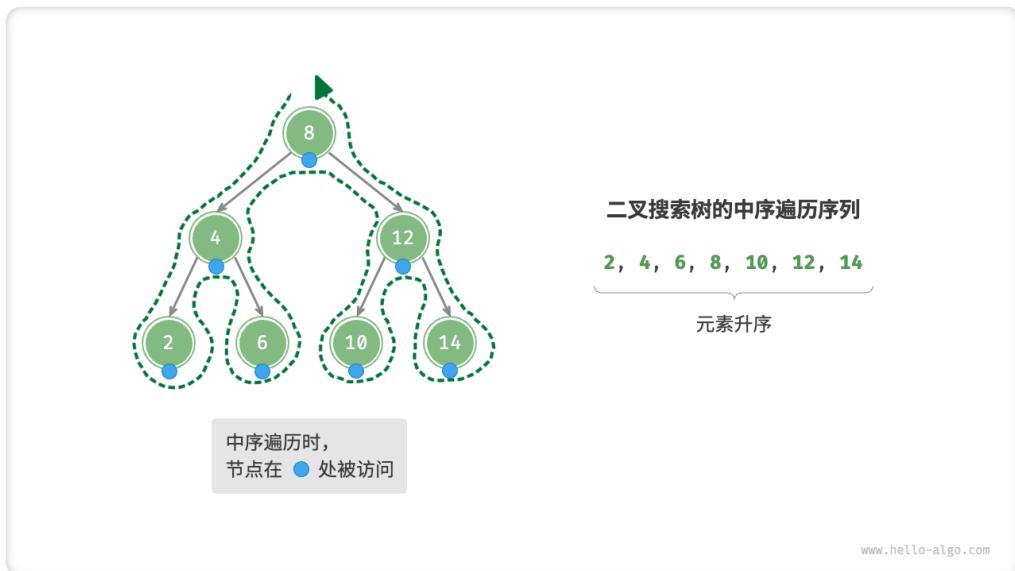


Figure 8-22. 二叉搜索树的中序遍历序列

### 8.4.2. 二叉搜索树的效率

给定一组数据，我们考虑使用数组或二叉搜索树存储。

观察可知，二叉搜索树的各项操作的时间复杂度都是对数阶，具有稳定且高效的性能表现。只有在高频添加、低频查找删除的数据适用场景下，数组比二叉搜索树的效率更高。

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

在理想情况下，二叉搜索树是“平衡”的，这样就可以在  $\log n$  轮循环内查找任意节点。

然而，如果我们在二叉搜索树中不断地插入和删除节点，可能导致二叉树退化为链表，这时各种操作的时间复杂度也会退化为  $O(n)$ 。

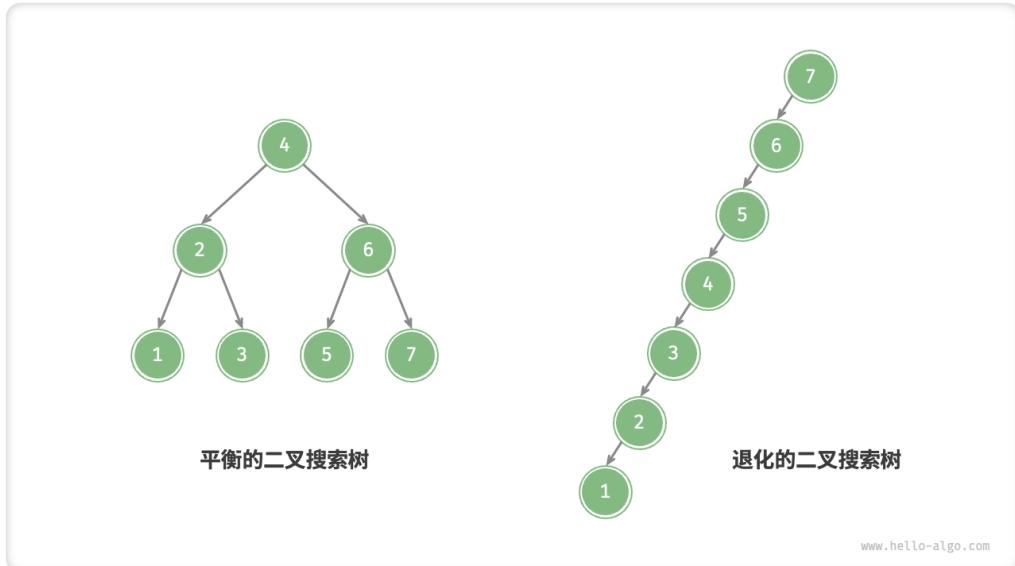


Figure 8-23. 二叉搜索树的平衡与退化

#### 8.4.3. 二叉搜索树常见应用

- 用作系统中的多级索引，实现高效的查找、插入、删除操作。
- 作为某些搜索算法的底层数据结构。
- 用于存储数据流，以保持其有序状态。

### 8.5. AVL 树 \*

在二叉搜索树章节中，我们提到了在多次插入和删除操作后，二叉搜索树可能退化为链表。这种情况下，所有操作的时间复杂度将从  $O(\log n)$  恶化为  $O(n)$ 。

如下图所示，经过两次删除节点操作，这个二叉搜索树便会退化为链表。

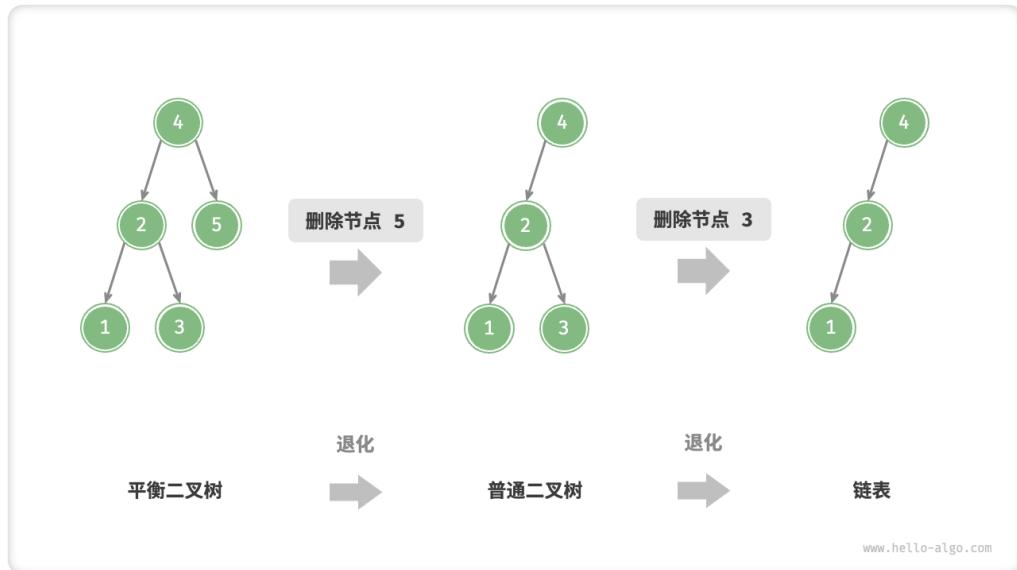


Figure 8-24. AVL 树在删除节点后发生退化

再例如，在以下完美二叉树中插入两个节点后，树将严重向左倾斜，查找操作的时间复杂度也随之恶化。

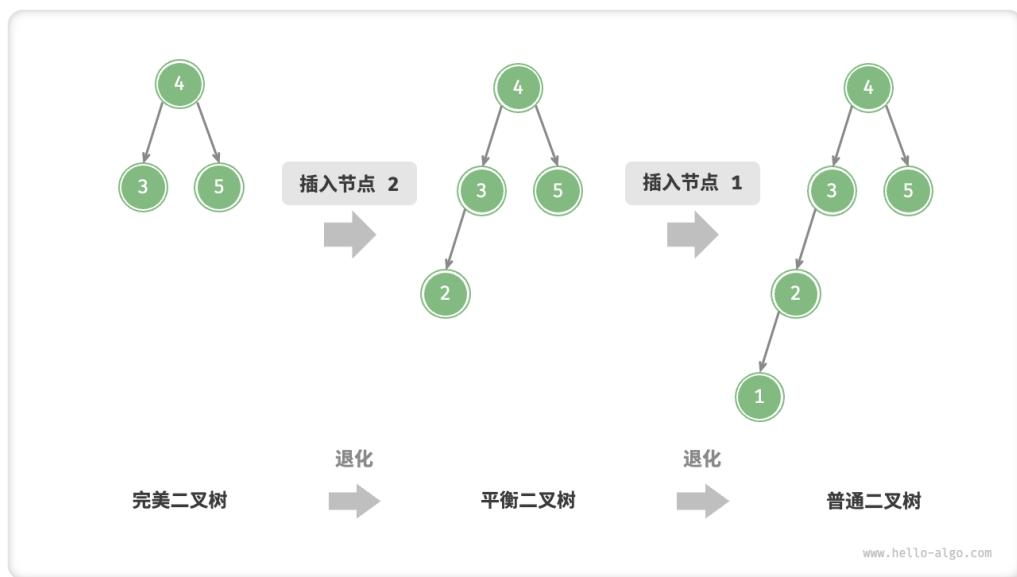


Figure 8-25. AVL 树在插入节点后发生退化

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」。论文中详细描述了一系列操作，确保在持续添加和删除节点后，AVL 树不会退化，从而使得各种操作的时间复杂度保持在  $O(\log n)$  级别。换句话说，在需要频繁进行增删查改操作的场景中，AVL 树能始终保持高效的数据操作性能，具有很好的应用价值。

### 8.5.1. AVL 树常见术语

「AVL 树」既是二叉搜索树也是平衡二叉树，同时满足这两类二叉树的所有性质，因此也被称为「平衡二叉搜索树」。

#### 节点高度

在操作 AVL 树时，我们需要获取节点的高度，因此需要为 AVL 树的节点类添加 `height` 变量。

```
class TreeNode:
    """AVL 树节点类"""
    def __init__(self, val: int):
        self.val: int = val           # 节点值
        self.height: int = 0          # 节点高度
        self.left: Optional[TreeNode] = None  # 左子节点引用
        self.right: Optional[TreeNode] = None # 右子节点引用
```

「节点高度」是指从该节点到最远叶节点的距离，即所经过的“边”的数量。需要特别注意的是，叶节点的高度为 0，而空节点的高度为 -1。我们将创建两个工具函数，分别用于获取和更新节点的高度。

```
# == File: avl_tree.py ==
def height(self, node: TreeNode | None) -> int:
    """ 获取节点高度 """
    # 空节点高度为 -1，叶节点高度为 0
    if node is not None:
        return node.height
    return -1

def __update_height(self, node: TreeNode | None):
    """ 更新节点高度 """
    # 节点高度等于最高子树高度 + 1
    node.height = max([self.height(node.left), self.height(node.right)]) + 1
```

#### 节点平衡因子

节点的「平衡因子 Balance Factor」定义为节点左子树的高度减去右子树的高度，同时规定空节点的平衡因子为 0。我们同样将获取节点平衡因子的功能封装成函数，方便后续使用。

```
# == File: avl_tree.py ==
def balance_factor(self, node: TreeNode | None) -> int:
    """ 获取平衡因子 """
    # 空节点平衡因子为 0
    if node is None:
```

```

    return 0
# 节点平衡因子 = 左子树高度 - 右子树高度
return self.height(node.left) - self.height(node.right)

```



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

### 8.5.2. AVL 树旋转

AVL 树的特点在于「旋转 Rotation」操作，它能够在不影响二叉树的中序遍历序列的前提下，使失衡节点重新恢复平衡。换句话说，旋转操作既能保持树的「二叉搜索树」属性，也能使树重新变为「平衡二叉树」。

我们将平衡因子绝对值  $> 1$  的节点称为「失衡节点」。根据节点失衡情况的不同，旋转操作分为四种：右旋、左旋、先右旋后左旋、先左旋后右旋。下面我们将详细介绍这些旋转操作。

#### 右旋

如下图所示，节点下方为平衡因子。从底至顶看，二叉树中首个失衡节点是“节点 3”。我们关注以该失衡节点为根节点的子树，将该节点记为 `node`，其左子节点记为 `child`，执行「右旋」操作。完成右旋后，子树已经恢复平衡，并且仍然保持二叉搜索树的特性。

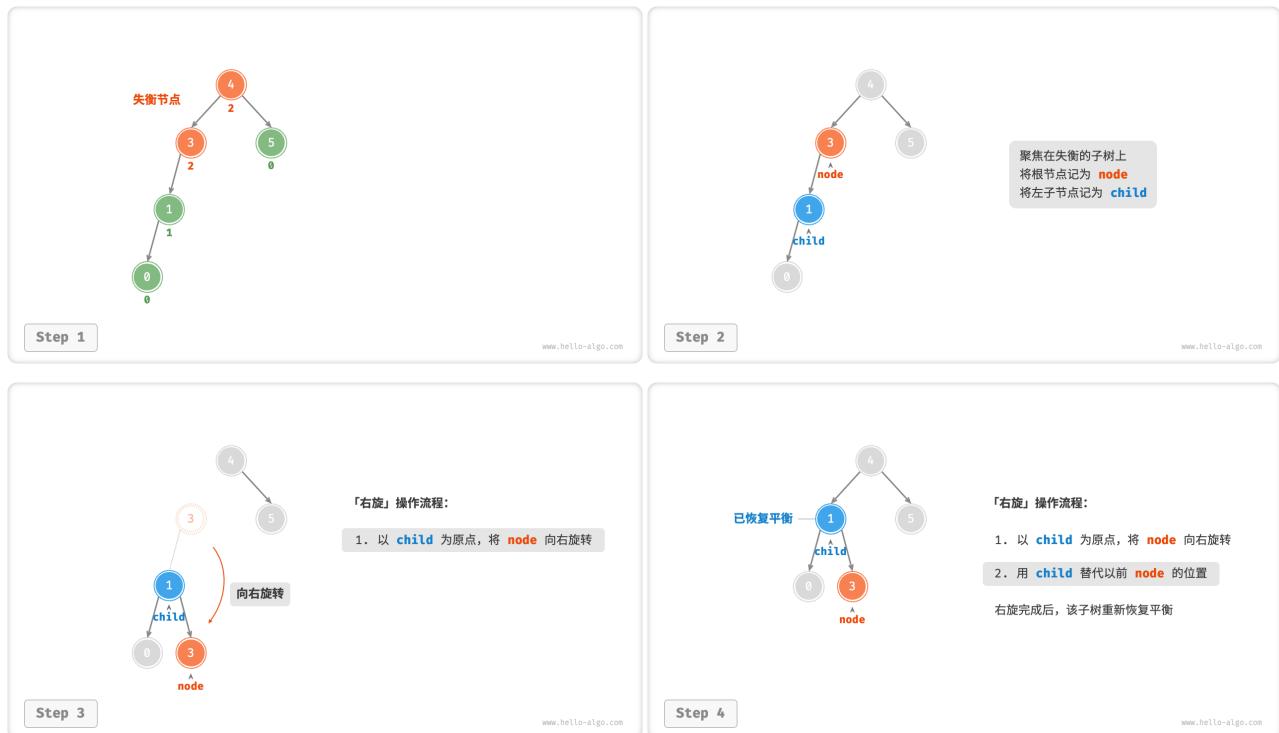


Figure 8-26. 右旋操作步骤

此外，如果节点 `child` 本身有右子节点（记为 `grandChild`），则需要在「右旋」中添加一步：将 `grandChild` 作为 `node` 的左子节点。

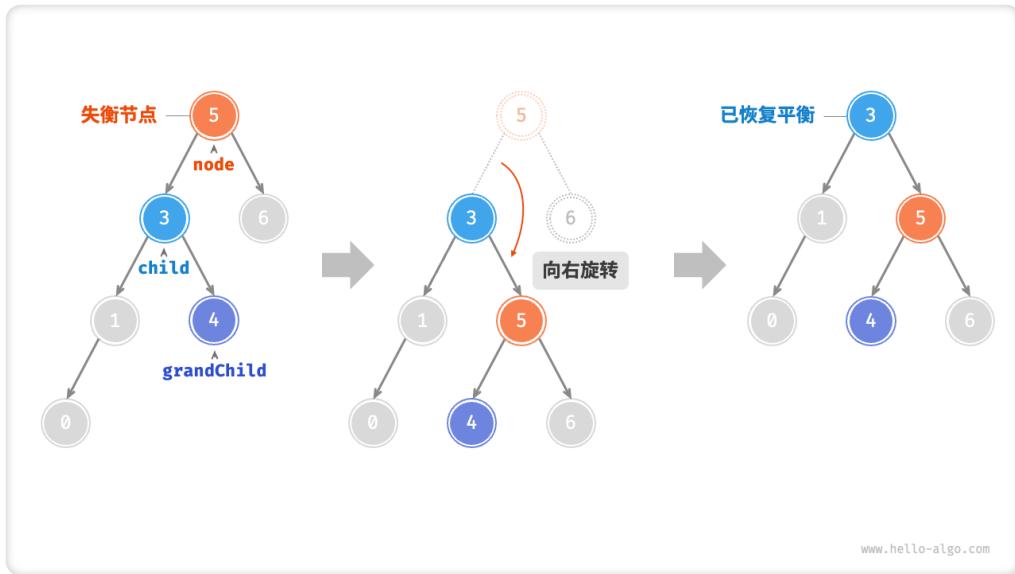


Figure 8-27. 有 `grandChild` 的右旋操作

“向右旋转”是一种形象化的说法，实际上需要通过修改节点指针来实现，代码如下所示。

```
# === File: avl_tree.py ===
def __right_rotate(self, node: TreeNode | None) -> TreeNode | None:
    """ 右旋操作 """
    child = node.left
    grand_child = child.right
    # 以 child 为原点，将 node 向右旋转
    child.right = node
    node.left = grand_child
    # 更新节点高度
    self.__update_height(node)
    self.__update_height(child)
    # 返回旋转后子树的根节点
    return child
```

## 左旋

相应的，如果考虑上述失衡二叉树的“镜像”，则需要执行「左旋」操作。

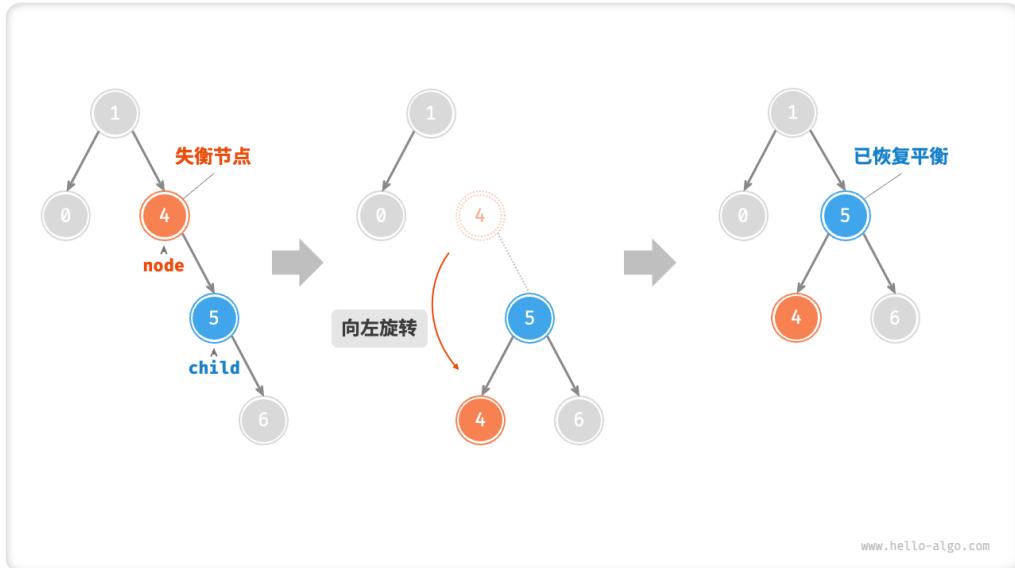
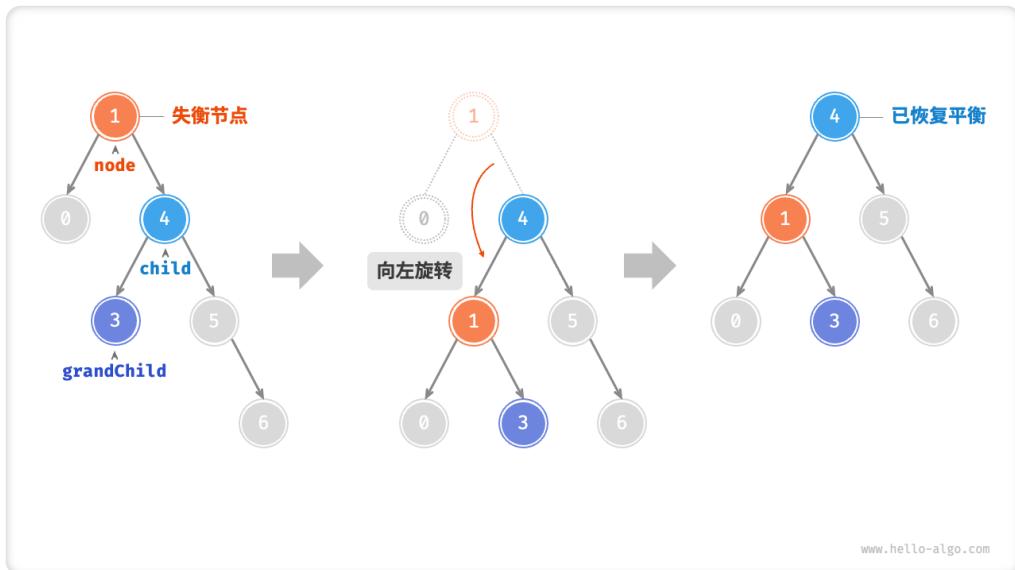


Figure 8-28. 左旋操作

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

Figure 8-29. 有 `grandChild` 的左旋操作

可以观察到，右旋和左旋操作在逻辑上是镜像对称的，它们分别解决的两种失衡情况也是对称的。基于对称性，我们可以轻松地从右旋的代码推导出左旋的代码。具体地，只需将「右旋」代码中的把所有的 `left` 替换为 `right`，将所有的 `right` 替换为 `left`，即可得到「左旋」代码。

```
# == File: avl_tree.py ==
def __left_rotate(self, node: TreeNode | None) -> TreeNode | None:
    """ 左旋操作 """
    child = node.right
    grand_child = child.left
    # 以 child 为原点, 将 node 向左旋转
    child.left = node
    node.right = grand_child
    # 更新节点高度
    self.__update_height(node)
    self.__update_height(child)
    # 返回旋转后子树的根节点
    return child
```

### 先左旋后右旋

对于下图中的失衡节点 3，仅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」。

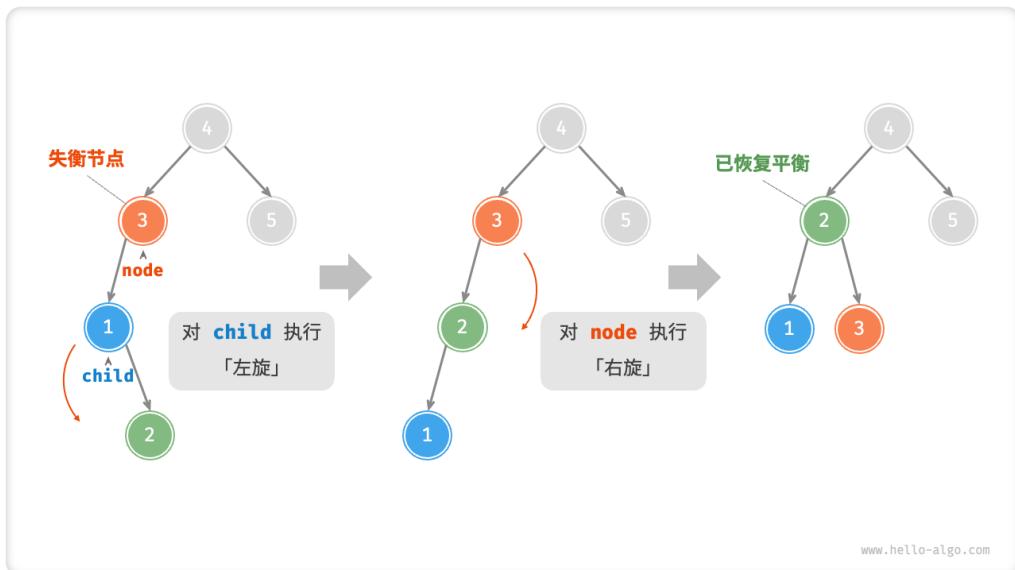


Figure 8-30. 先左旋后右旋

### 先右旋后左旋

同理，对于上述失衡二叉树的镜像情况，需要先右旋后左旋，即先对 `child` 执行「右旋」，然后对 `node` 执行「左旋」。

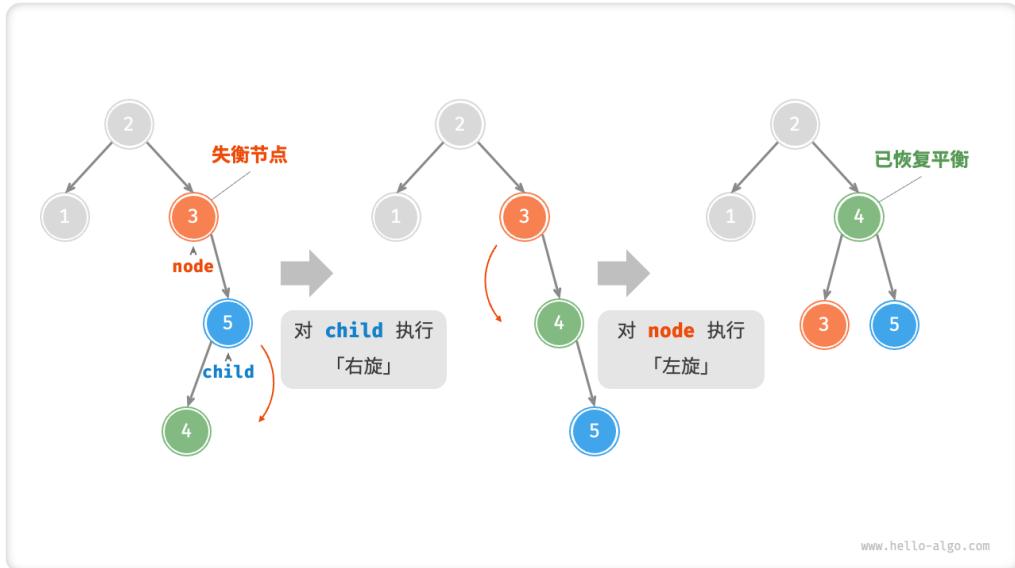


Figure 8-31. 先右旋后左旋

### 旋转的选择

下图展示的四种失衡情况与上述案例逐个对应，分别需要采用右旋、左旋、先右后左、先左后右的旋转操作。

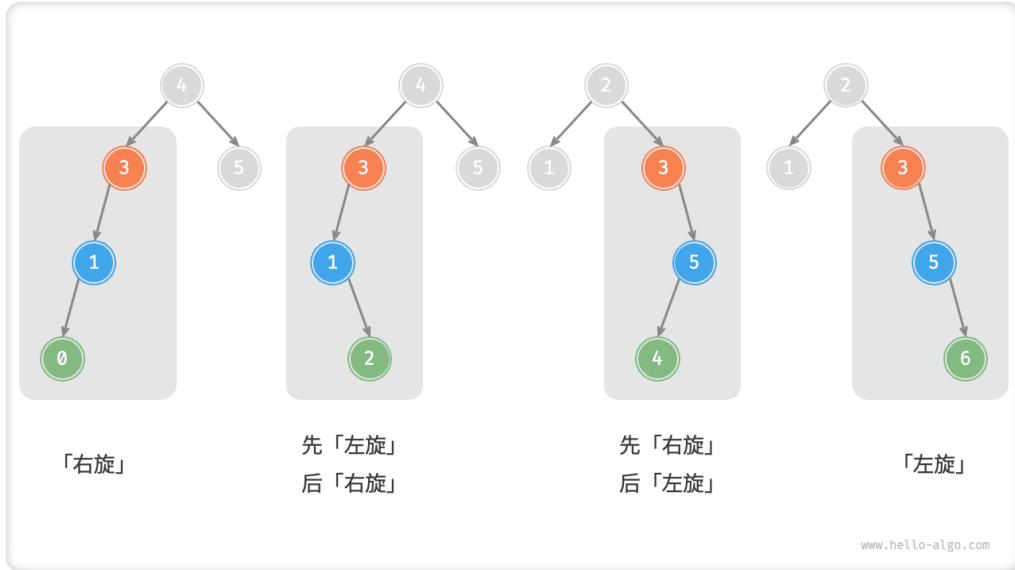


Figure 8-32. AVL 树的四种旋转情况

在代码中，我们通过判断失衡节点的平衡因子以及较高一侧子节点的平衡因子的正负号，来确定失衡节点属于上图中的哪种情况。

失衡节点的平衡因子	子节点的平衡因子	应采用的旋转方法
$> 0$ (即左偏树)	$\geq 0$	右旋
$> 0$ (即左偏树)	$< 0$	先左旋后右旋
$< 0$ (即右偏树)	$\leq 0$	左旋
$< 0$ (即右偏树)	$> 0$	先右旋后左旋

为了便于使用，我们将旋转操作封装成一个函数。有了这个函数，我们就能对各种失衡情况进行旋转，使失衡节点重新恢复平衡。

```
# == File: avl_tree.py ==
def __rotate(self, node: TreeNode | None) -> TreeNode | None:
    """ 执行旋转操作，使该子树重新恢复平衡 """
    # 获取节点 node 的平衡因子
    balance_factor = self.balance_factor(node)
    # 左偏树
    if balance_factor > 1:
        if self.balance_factor(node.left) >= 0:
            # 右旋
            return self.__right_rotate(node)
        else:
            # 先左旋后右旋
            node.left = self.__left_rotate(node.left)
            return self.__right_rotate(node)
    # 右偏树
    elif balance_factor < -1:
        if self.balance_factor(node.right) <= 0:
            # 左旋
            return self.__left_rotate(node)
        else:
            # 先右旋后左旋
            node.right = self.__right_rotate(node.right)
            return self.__left_rotate(node)
    # 平衡树，无需旋转，直接返回
    return node
```

### 8.5.3. AVL 树常用操作

#### 插入节点

「AVL 树」的节点插入操作与「二叉搜索树」在主体上类似。唯一的区别在于，在 AVL 树中插入节点后，从该节点到根节点的路径上可能会出现一系列失衡节点。因此，我们需要从这个节点开始，自底向上执行旋转操作，使所有失衡节点恢复平衡。

```
# == File: avl_tree.py ===
def insert(self, val) -> None:
    """ 插入节点"""
    self.__root = self.__insert_helper(self.__root, val)

def __insert_helper(self, node: TreeNode | None, val: int) -> TreeNode:
    """ 递归插入节点（辅助方法）"""
    if node is None:
        return TreeNode(val)
    # 1. 查找插入位置，并插入节点
    if val < node.val:
        node.left = self.__insert_helper(node.left, val)
    elif val > node.val:
        node.right = self.__insert_helper(node.right, val)
    else:
        # 重复节点不插入，直接返回
        return node
    # 更新节点高度
    self.__update_height(node)
    # 2. 执行旋转操作，使该子树重新恢复平衡
    return self.__rotate(node)
```

## 删除节点

类似地，在二叉搜索树的删除节点方法的基础上，需要从底至顶地执行旋转操作，使所有失衡节点恢复平衡。

```
# == File: avl_tree.py ===
def remove(self, val: int) -> None:
    """ 删除节点"""
    self.__root = self.__remove_helper(self.__root, val)

def __remove_helper(self, node: TreeNode | None, val: int) -> TreeNode | None:
    """ 递归删除节点（辅助方法）"""
    if node is None:
        return None
    # 1. 查找节点，并删除之
    if val < node.val:
        node.left = self.__remove_helper(node.left, val)
    elif val > node.val:
        node.right = self.__remove_helper(node.right, val)
    else:
        if node.left is None or node.right is None:
            child = node.left or node.right
            # 子节点数量 = 0，直接删除 node 并返回
            if child is None:
```

```
    return None
    # 子节点数量 = 1，直接删除 node
    else:
        node = child
else:
    # 子节点数量 = 2，则将中序遍历的下个节点删除，并用该节点替换当前节点
    temp = node.right
    while temp.left is not None:
        temp = temp.left
    node.right = self.__remove_helper(node.right, temp.val)
    node.val = temp.val
# 更新节点高度
self.__update_height(node)
# 2. 执行旋转操作，使该子树重新恢复平衡
return self.__rotate(node)
```

## 查找节点

AVL 树的节点查找操作与二叉搜索树一致，在此不再赘述。

### 8.5.4. AVL 树典型应用

- 组织和存储大型数据，适用于高频查找、低频增删的场景；
- 用于构建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节点所需的旋转操作相对较少，在节点增删操作上的平均效率高于 AVL 树。

## 8.6. 小结

- 二叉树是一种非线性数据结构，体现“一分为二”的分治逻辑。每个二叉树节点包含一个值以及两个指针，分别指向其左子节点和右子节点。
- 对于二叉树中的某个节点，其左（右）子节点及其以下形成的树被称为该节点的左（右）子树。
- 二叉树的相关术语包括根节点、叶节点、层、度、边、高度和深度等。
- 二叉树的初始化、节点插入和节点删除操作与链表操作方法类似。
- 常见的二叉树类型有完美二叉树、完全二叉树、满二叉树和平衡二叉树。完美二叉树是最理想的状态，而链表是退化后的最差状态。
- 二叉树可以用数组表示，方法是将节点值和空位按层序遍历顺序排列，并根据父节点与子节点之间的索引映射关系来实现指针。

- 二叉树的层序遍历是一种广度优先搜索方法，它体现了“一圈一圈向外”的分层遍历方式，通常通过队列来实现。
- 前序、中序、后序遍历皆属于深度优先搜索，它们体现了“走到尽头，再回头继续”的回溯遍历方式，通常使用递归来实现。
- 二叉搜索树是一种高效的元素查找数据结构，其查找、插入和删除操作的时间复杂度均为  $O(\log n)$ 。当二叉搜索树退化为链表时，各项时间复杂度会劣化至  $O(n)$ 。
- AVL 树，也称为平衡二叉搜索树，它通过旋转操作，确保在不断插入和删除节点后，树仍然保持平衡。
- AVL 树的旋转操作包括右旋、左旋、先右旋再左旋、先左旋再右旋。在插入或删除节点后，AVL 树会从底向顶执行旋转操作，使树重新恢复平衡。

# 9. 堆

## 9.1. 堆

「堆 Heap」是一种满足特定条件的完全二叉树，可分为两种类型：

- 「大顶堆 Max Heap」，任意节点的值  $\geq$  其子节点的值；
- 「小顶堆 Min Heap」，任意节点的值  $\leq$  其子节点的值；

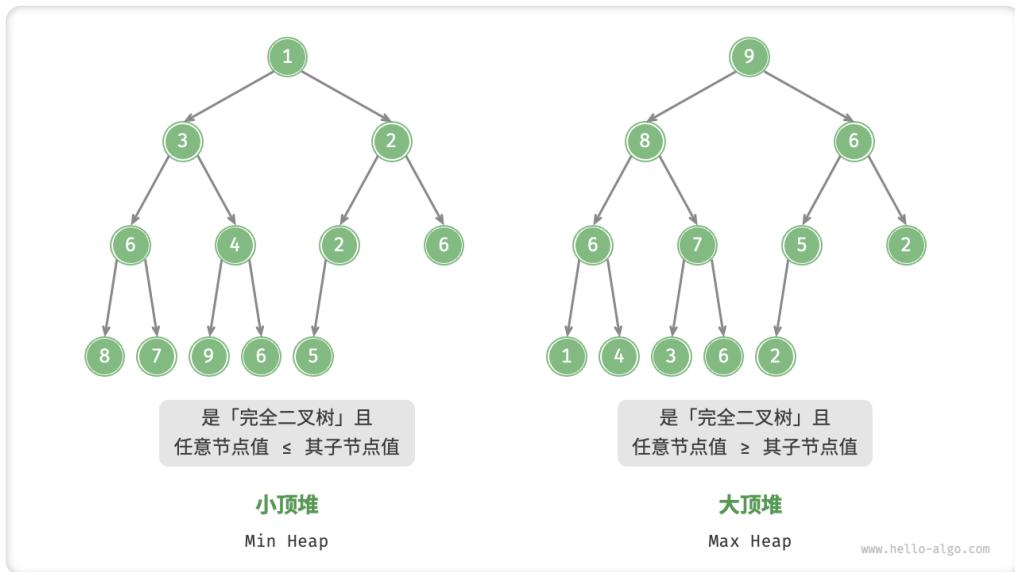


Figure 9-1. 小顶堆与大顶堆

堆作为完全二叉树的一个特例，具有以下特性：

- 最底层节点靠左填充，其他层的节点都被填满。
- 我们将二叉树的根节点称为「堆顶」，将底层最靠右的节点称为「堆底」。
- 对于大顶堆（小顶堆），堆顶元素（即根节点）的值分别是最大（最小）的。

### 9.1.1. 堆常用操作

需要指出的是，许多编程语言提供的是「优先队列 Priority Queue」，这是一种抽象数据结构，定义为具有优先级排序的队列。

实际上，**堆通常用作实现优先队列，大顶堆相当于元素按从大到小顺序出队的优先队列**。从使用角度来看，我们可以将「优先队列」和「堆」看作等价的数据结构。因此，本书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，方法名需要根据编程语言来确定。

方法名	描述	时间复杂度
push()	元素入堆	$O(\log n)$
pop()	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	$O(1)$

在实际应用中，我们可以直接使用编程语言提供的堆类（或优先队列类）。



类似于排序算法中的“从小到大排列”和“从大到小排列”，我们可以通过修改 Comparator 来实现“小顶堆”与“大顶堆”之间的转换。

```
# === File: heap.py ===
# 初始化小顶堆
min_heap, flag = [], 1
# 初始化大顶堆
max_heap, flag = [], -1

# Python 的 heapq 模块默认实现小顶堆
# 考虑将“元素取负”后再入堆，这样就可以将大小关系颠倒，从而实现大顶堆
# 在本示例中，flag = 1 时对应小顶堆，flag = -1 时对应大顶堆

# 元素入堆
heapq.heappush(max_heap, flag * 1)
heapq.heappush(max_heap, flag * 3)
heapq.heappush(max_heap, flag * 2)
heapq.heappush(max_heap, flag * 5)
heapq.heappush(max_heap, flag * 4)

# 获取堆顶元素
peek: int = flag * max_heap[0] # 5

# 堆顶元素出堆
# 出堆元素会形成一个从大到小的序列
val = flag * heapq.heappop(max_heap) # 5
val = flag * heapq.heappop(max_heap) # 4
val = flag * heapq.heappop(max_heap) # 3
val = flag * heapq.heappop(max_heap) # 2
val = flag * heapq.heappop(max_heap) # 1
```

```
# 获取堆大小
size: int = len(max_heap)

# 判断堆是否为空
is_empty: bool = not max_heap

# 输入列表并建堆
min_heap: List[int] = [1, 3, 2, 5, 4]
heappq.heapify(min_heap)
```

### 9.1.2. 堆的实现

下文实现的是大顶堆。若要将其转换为小顶堆，只需将所有大小逻辑判断取逆（例如，将 $\geq$ 替换为 $\leq$ ）。感兴趣的读者可以自行实现。

#### 堆的存储与表示

我们在二叉树章节中学习到，完全二叉树非常适合用数组来表示。由于堆正是一种完全二叉树，我们将采用数组来存储堆。

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

具体而言，给定索引  $i$ ，其左子节点索引为  $2i + 1$ ，右子节点索引为  $2i + 2$ ，父节点索引为  $(i - 1)/2$ （向下取整）。当索引越界时，表示空节点或节点不存在。

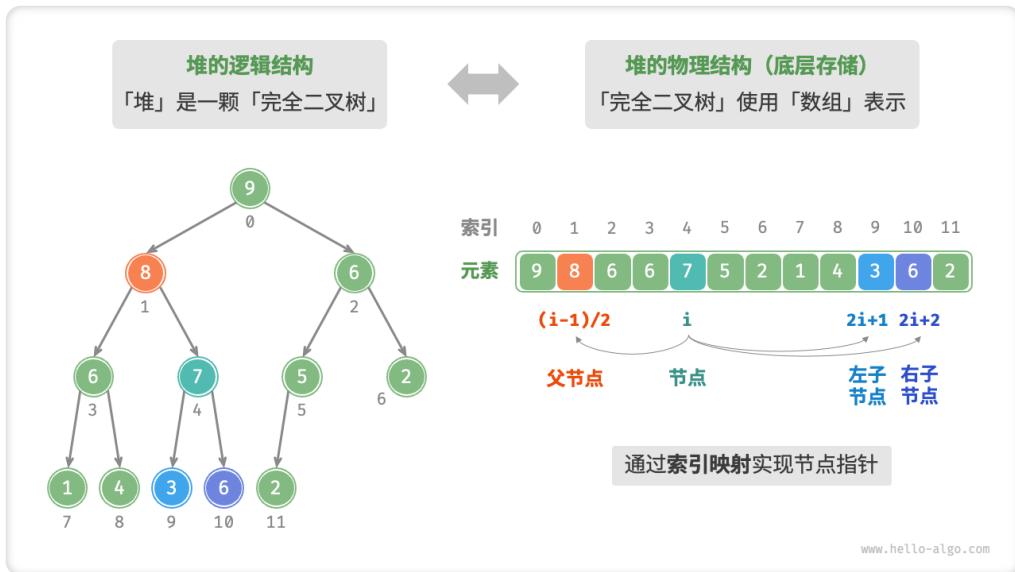


Figure 9-2. 堆的表示与存储

我们可以将索引映射公式封装成函数，方便后续使用。

```
# == File: my_heap.py ==
def left(self, i: int) -> int:
    """ 获取左子节点索引 """
    return 2 * i + 1

def right(self, i: int) -> int:
    """ 获取右子节点索引 """
    return 2 * i + 2

def parent(self, i: int) -> int:
    """ 获取父节点索引 """
    return (i - 1) // 2 # 向下整除
```

## 访问堆顶元素

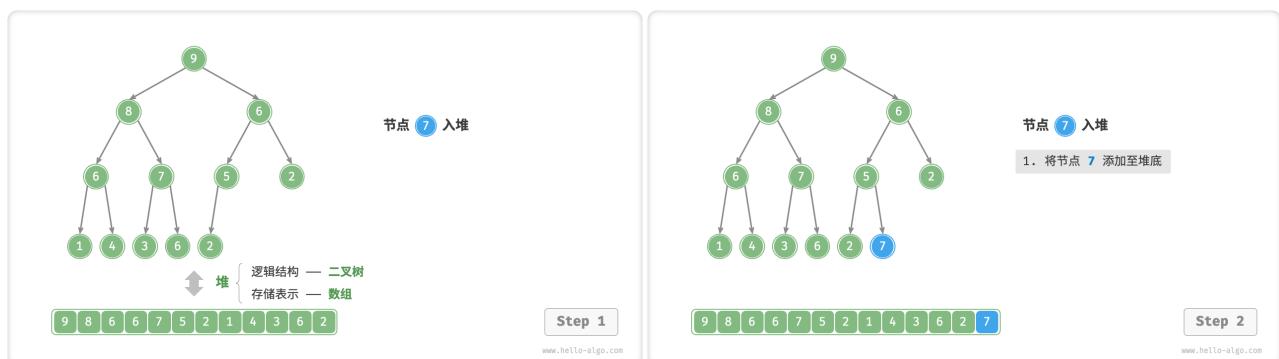
堆顶元素即为二叉树的根节点，也就是列表的首个元素。

```
# == File: my_heap.py ==
def peek(self) -> int:
    """ 访问堆顶元素 """
    return self.max_heap[0]
```

## 元素入堆

给定元素 `val`，我们首先将其添加到堆底。添加之后，由于 `val` 可能大于堆中其他元素，堆的成立条件可能已被破坏。因此，**需要修复从插入节点到根节点的路径上的各个节点**，这个操作被称为「堆化 Heapify」。

考虑从入堆节点开始，**从底至顶执行堆化**。具体来说，我们比较插入节点与其父节点的值，如果插入节点更大，则将它们交换。然后继续执行此操作，从底至顶修复堆中的各个节点，直至越过根节点或遇到无需交换的节点时结束。



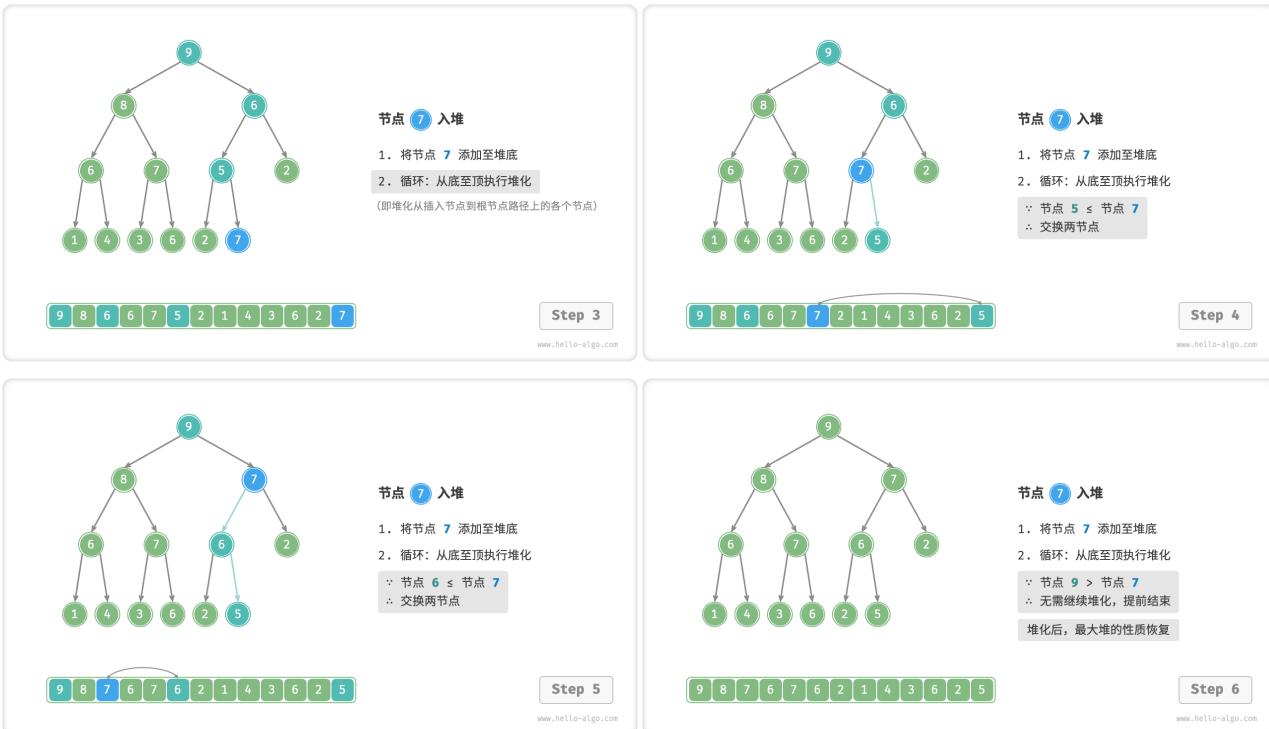


Figure 9-3. 元素入堆步骤

设节点总数为  $n$ ，则树的高度为  $O(\log n)$ 。由此可知，堆化操作的循环轮数最多为  $O(\log n)$ ，元素入堆操作的时间复杂度为  $O(\log n)$ 。

```
# === File: my_heap.py ===
def push(self, val: int):
    """ 元素入堆 """
    # 添加节点
    self.max_heap.append(val)
    # 从底至顶堆化
    self.sift_up(self.size() - 1)

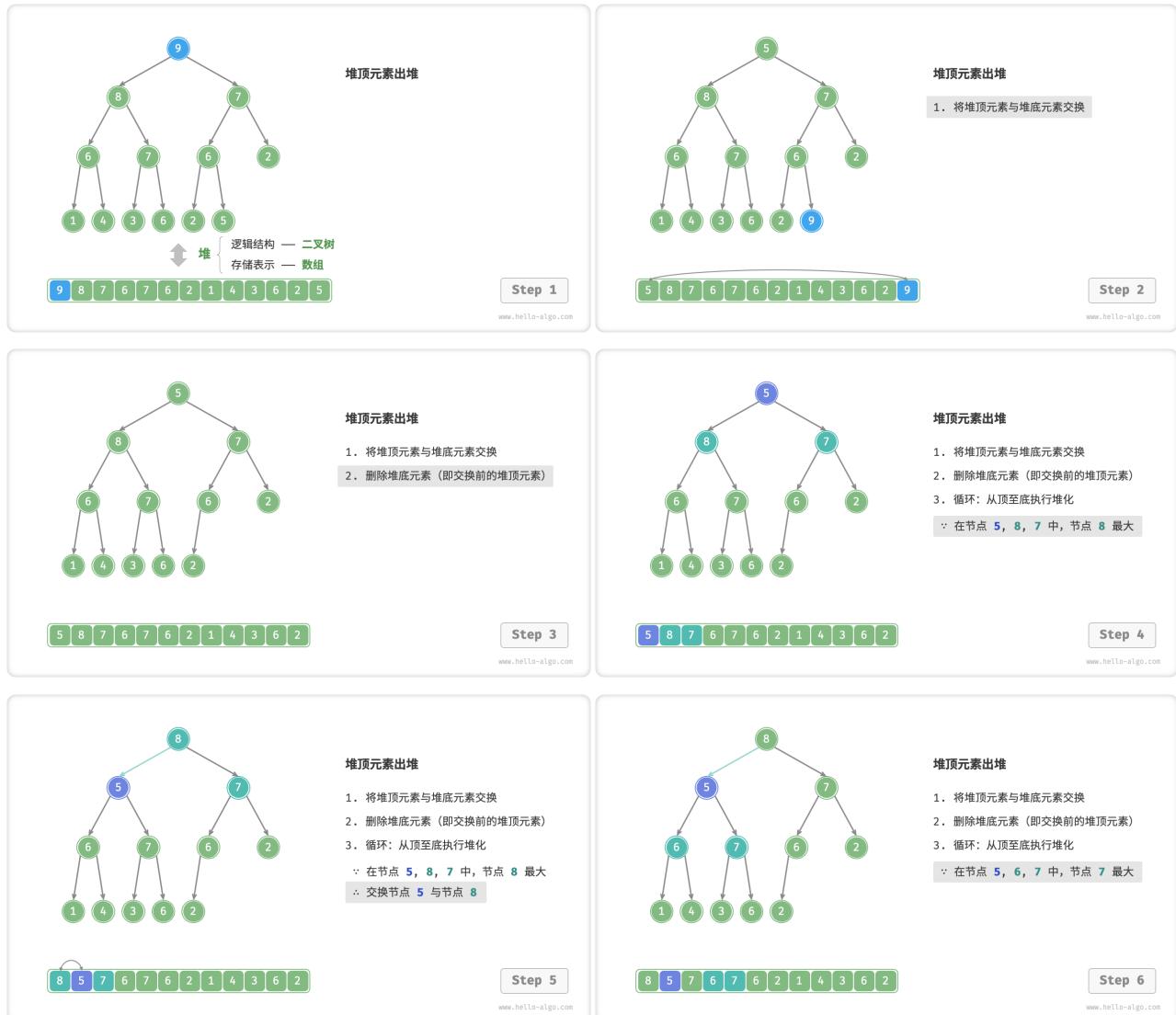
def sift_up(self, i: int):
    """ 从节点 i 开始，从底至顶堆化 """
    while True:
        # 获取节点 i 的父节点
        p = self.parent(i)
        # 当“越过根节点”或“节点无需修复”时，结束堆化
        if p < 0 or self.max_heap[i] <= self.max_heap[p]:
            break
        # 交换两节点
        self.swap(i, p)
        # 循环向上堆化
        i = p
```

### 堆顶元素出堆

堆顶元素是二叉树的根节点，即列表首元素。如果我们直接从列表中删除首元素，那么二叉树中所有节点的索引都会发生变化，这将使得后续使用堆化修复变得困难。为了尽量减少元素索引的变动，我们采取以下操作步骤：

1. 交换堆顶元素与堆底元素（即交换根节点与最右叶节点）；
2. 交换完成后，将堆底从列表中删除（注意，由于已经交换，实际上删除的是原来的堆顶元素）；
3. 从根节点开始，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**，我们将根节点的值与其两个子节点的值进行比较，将最大的子节点与根节点交换；然后循环执行此操作，直到越过叶节点或遇到无需交换的节点时结束。



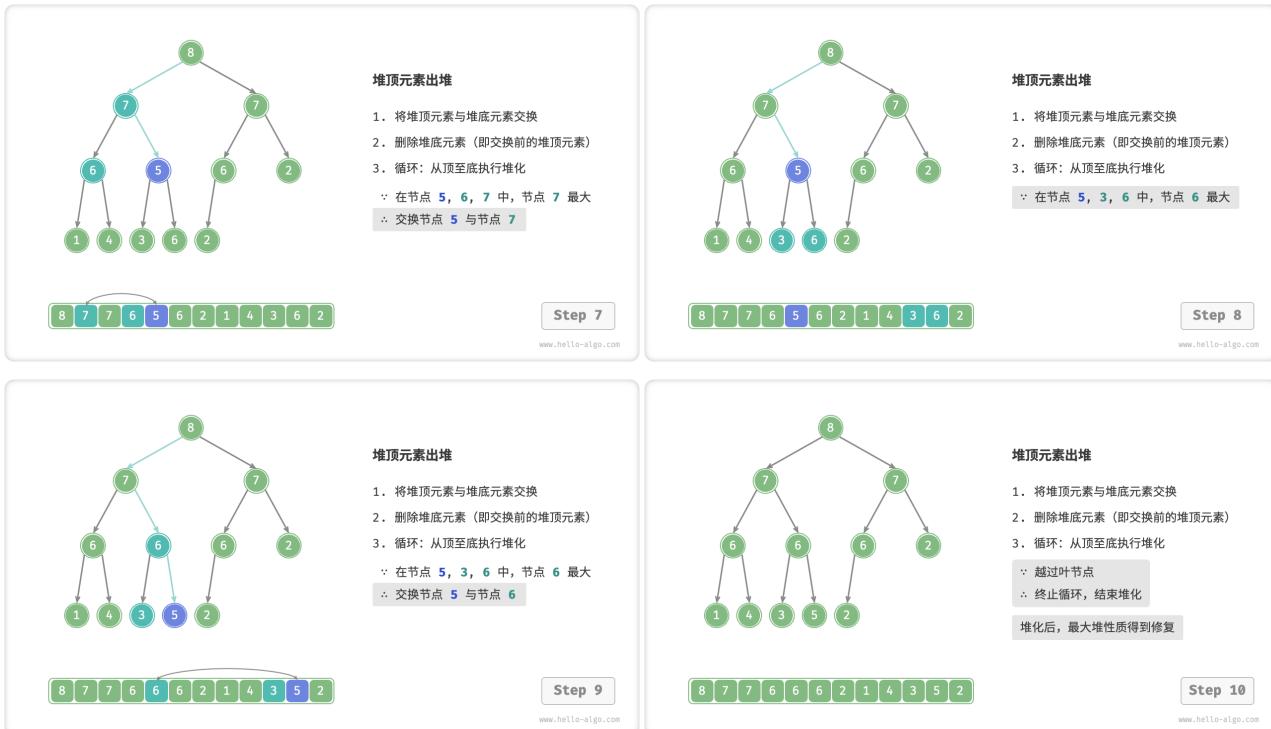


Figure 9-4. 堆顶元素出堆步骤

与元素入堆操作相似，堆顶元素出堆操作的时间复杂度也为  $O(\log n)$ 。

```
# == File: my_heap.py ==
def pop(self) -> int:
    """ 元素出堆 """
    # 判空处理
    if self.is_empty():
        raise IndexError("堆为空")
    # 交换根节点与最右叶节点（即交换首元素与尾元素）
    self.swap(0, self.size() - 1)
    # 删除节点
    val = self.max_heap.pop()
    # 从顶到底堆化
    self.sift_down(0)
    # 返回堆顶元素
    return val

def sift_down(self, i: int):
    """ 从节点 i 开始，从顶到底堆化 """
    while True:
        # 判断节点 i, l, r 中值最大的节点，记为 ma
        l, r, ma = self.left(i), self.right(i), i
        if l < self.size() and self.max_heap[l] > self.max_heap[ma]:
```

```

    ma = l
    if r < self.size() and self.max_heap[r] > self.max_heap[ma]:
        ma = r
    # 若节点 i 最大或索引 l, r 越界，则无需继续堆化，跳出
    if ma == i:
        break
    # 交换两节点
    self.swap(i, ma)
    # 循环向下堆化
    i = ma

```

### 9.1.3. 堆常见应用

- **优先队列**: 堆通常作为实现优先队列的首选数据结构，其入队和出队操作的时间复杂度均为  $O(\log n)$ ，而建队操作为  $O(n)$ ，这些操作都非常高效。
- **堆排序**: 给定一组数据，我们可以用它们建立一个堆，然后依次将所有元素弹出，从而得到一个有序序列。当然，堆排序的实现方法并不需要弹出元素，而是每轮将堆顶元素交换至数组尾部并缩小堆的长度。
- **获取最大的  $k$  个元素**: 这是一个经典的算法问题，同时也是一种典型应用，例如选择热度前 10 的新闻作为微博热搜，选取销量前 10 的商品等。

## 9.2. 建堆操作 \*

如果我们想要根据输入列表生成一个堆，这个过程被称为「建堆」。

### 9.2.1. 借助入堆方法实现

最直接的方法是借助“元素入堆操作”实现，首先创建一个空堆，然后将列表元素依次添加到堆中。

设元素数量为  $n$ ，则最后一个元素入堆的时间复杂度为  $O(\log n)$ 。在依次添加元素时，堆的平均长度为  $\frac{n}{2}$ ，因此该方法的总体时间复杂度为  $O(n \log n)$ 。

### 9.2.2. 基于堆化操作实现

有趣的是，存在一种更高效的建堆方法，其时间复杂度仅为  $O(n)$ 。我们先将列表所有元素原封不动添加到堆中，然后迭代地对各个节点执行“从顶到底堆化”。当然，**我们不需要对叶节点执行堆化操作**，因为它们没有子节点。

```

# === File: my_heap.py ===
def __init__(self, nums: list[int]):
    """ 构造方法 """
    # 将列表元素原封不动添加进堆

```

```
self.max_heap = nums
# 堆化除叶节点以外的其他所有节点
for i in range(self.parent(self.size() - 1), -1, -1):
    self.sift_down(i)
```

### 9.2.3. 复杂度分析

为什么第二种建堆方法的时间复杂度是  $O(n)$ ？我们来展开推算一下。

- 完全二叉树中，设节点总数为  $n$ ，则叶节点数量为  $(n + 1)/2$ ，其中 / 为向下整除。因此，在排除叶节点后，需要堆化的节点数量为  $(n - 1)/2$ ，复杂度为  $O(n)$ ；
- 在从顶至底堆化的过程中，每个节点最多堆化到叶节点，因此最大迭代次数为二叉树高度  $O(\log n)$ ；

将上述两者相乘，可得到建堆过程的时间复杂度为  $O(n \log n)$ 。然而，这个估算结果并不准确，因为我们没有考虑到二叉树底层节点数量远多于顶层节点的特性。

接下来我们来进行更为详细的计算。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）节点数量为  $n$ ，树高度为  $h$ 。上文提到，节点堆化最大迭代次数等于该节点到叶节点的距离，而该距离正是“节点高度”。

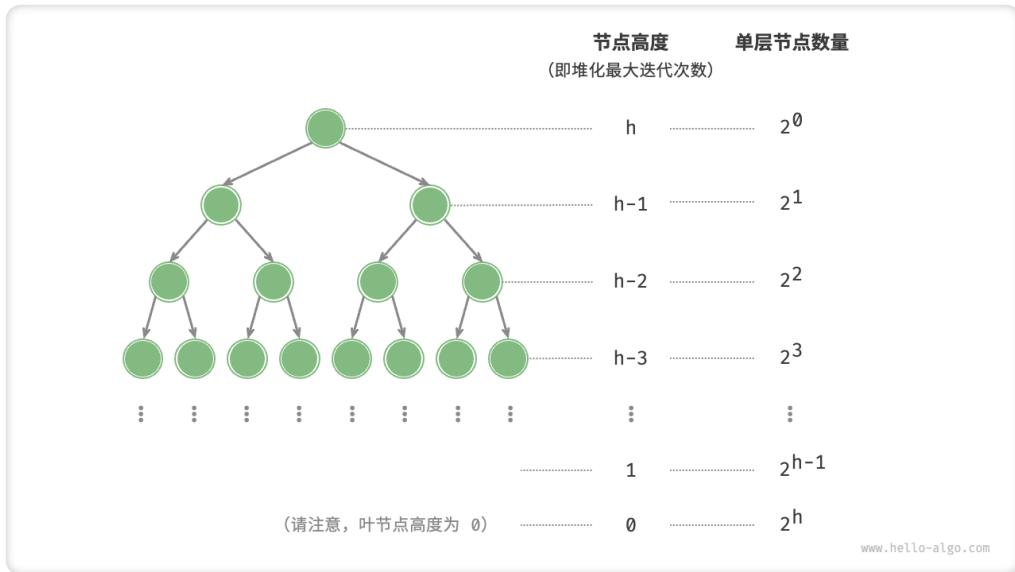


Figure 9-5. 完美二叉树的各层节点数量

因此，我们可以将各层的“节点数量  $\times$  节点高度”求和，从而得到所有节点的堆化迭代次数的总和。

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

化简上式需要借助中学的数列知识，先对  $T(h)$  乘以 2，得到

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

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，发现  $T(h)$  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  $h$  的完美二叉树的节点数量为  $n = 2^{h+1} - 1$ ，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。

### 9.3. 小结

- 堆是一棵完全二叉树，根据成立条件可分为大顶堆和小顶堆。大（小）顶堆的堆顶元素是最大（小）的。
- 优先队列的定义是具有出队优先级的队列，通常使用堆来实现。
- 堆的常用操作及其对应的时间复杂度包括：元素入堆  $O(\log n)$ 、堆顶元素出堆  $O(\log n)$  和访问堆顶元素  $O(1)$  等。
- 完全二叉树非常适合用数组表示，因此我们通常使用数组来存储堆。
- 堆化操作用于维护堆的性质，在入堆和出堆操作中都会用到。
- 输入  $n$  个元素并建堆的时间复杂度可以优化至  $O(n)$ ，非常高效。

# 10. 图

## 10.1. 图

「图 Graph」是一种非线性数据结构，由「顶点 Vertex」和「边 Edge」组成。我们可以将图  $G$  抽象地表示为一组顶点  $V$  和一组边  $E$  的集合。以下示例展示了一个包含 5 个顶点和 7 条边的图。

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

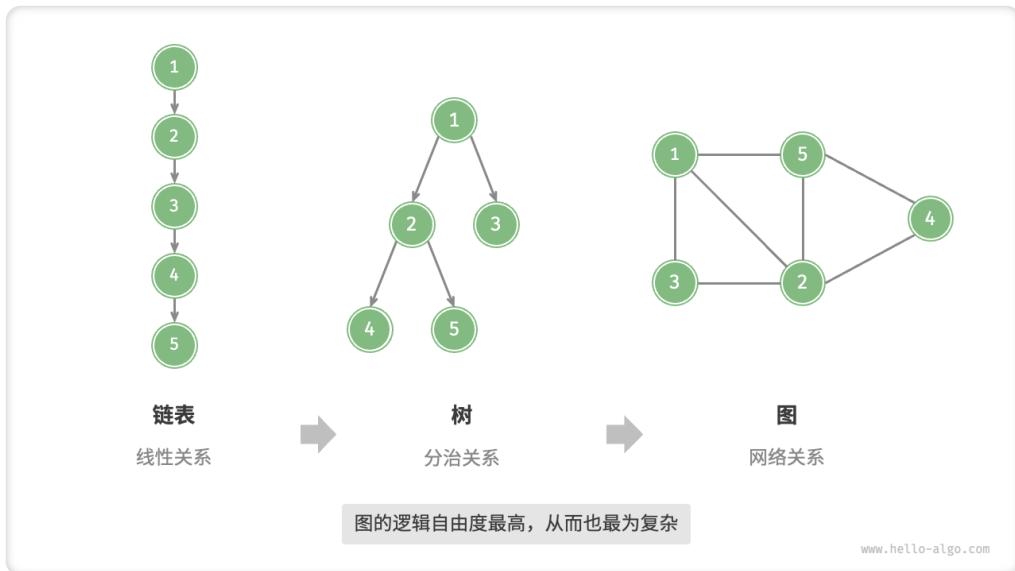


Figure 10-1. 链表、树、图之间的关系

那么，图与其他数据结构的关系是什么？如果我们把「顶点」看作节点，把「边」看作连接各个节点的指针，则可将「图」看作是一种从「链表」拓展而来的数据结构。**相较于线性关系（链表）和分治关系（树），网络关系（图）的自由度更高，从而更为复杂。**

### 10.1.1. 图常见类型

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

- 在无向图中，边表示两顶点之间的“双向”连接关系，例如微信或 QQ 中的“好友关系”；
- 在有向图中，边具有方向性，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；

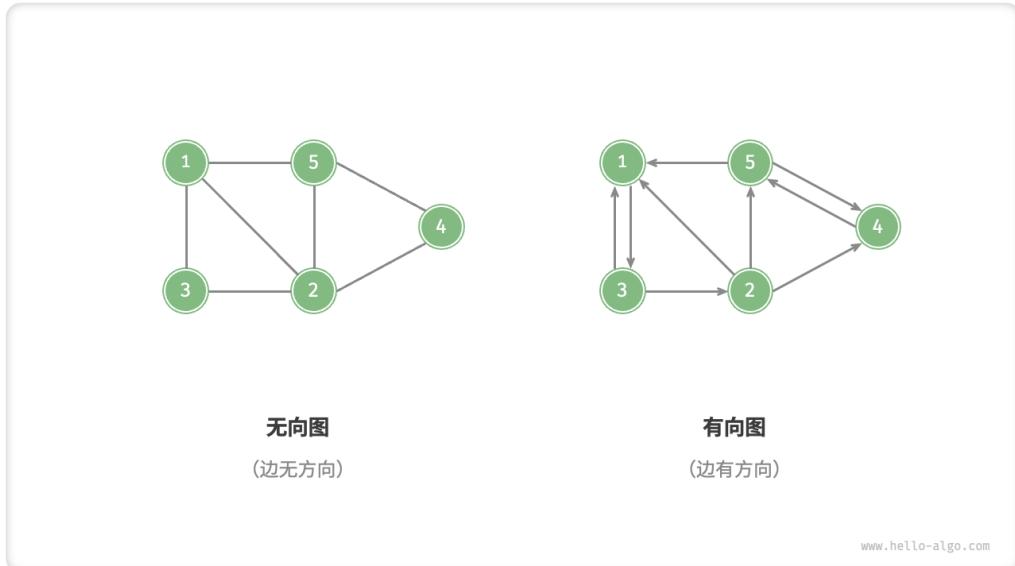


Figure 10-2. 有向图与无向图

根据所有顶点是否连通，可分为「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；

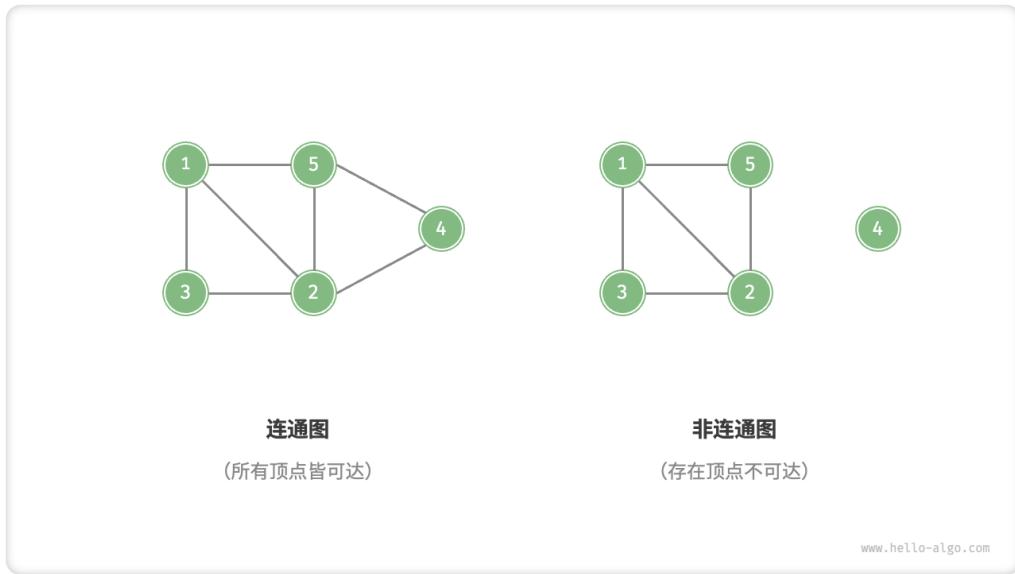


Figure 10-3. 连通图与非连通图

我们还可以为边添加“权重”变量，从而得到「有权图 Weighted Graph」。例如，在王者荣耀等手游中，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用有权图来表示。

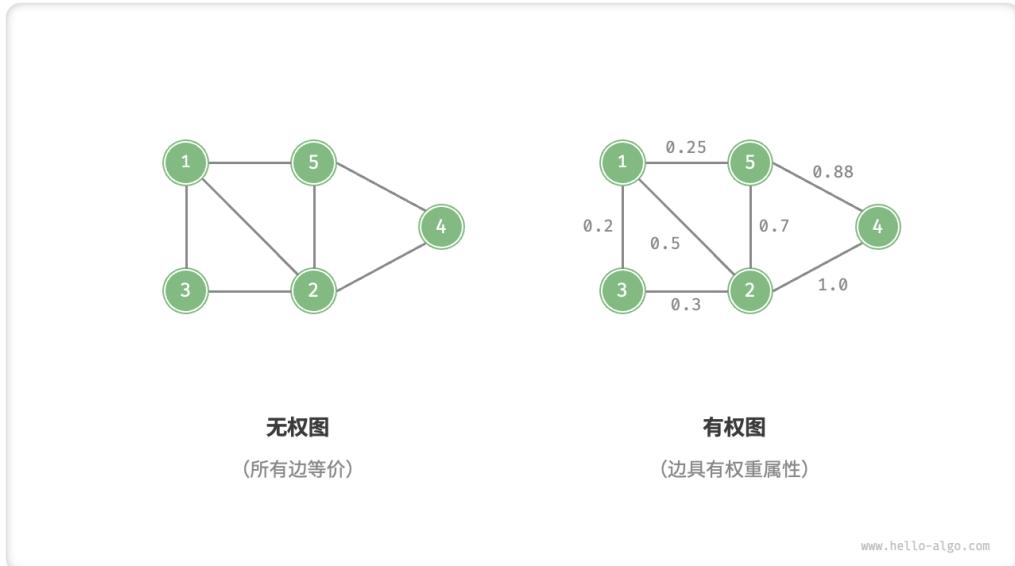


Figure 10-4. 有权图与无权图

### 10.1.2. 图常用术语

- 「邻接 Adjacency」：当两顶点之间存在边相连时，称这两顶点“邻接”。在上图中，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 经过的边构成的序列被称为从 A 到 B 的“路径”。在上图中，边序列 1-5-2-4 是顶点 1 到顶点 4 的一条路径。
- 「度 Degree」表示一个顶点拥有的边数。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。

### 10.1.3. 图的表示

图的常用表示方法包括「邻接矩阵」和「邻接表」。以下使用无向图进行举例。

#### 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，用 1 或 0 表示两个顶点之间是否存在边。

如下图所示，设邻接矩阵为  $M$ 、顶点列表为  $V$ ，那么矩阵元素  $M[i][j] = 1$  表示顶点  $V[i]$  到顶点  $V[j]$  之间存在边，反之  $M[i][j] = 0$  表示两顶点之间无边。

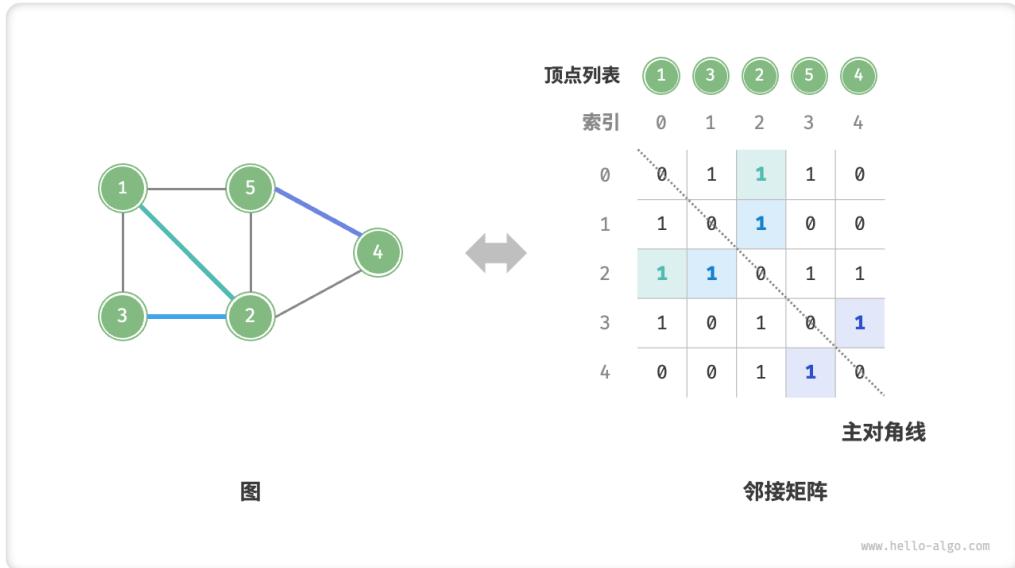


Figure 10-5. 图的邻接矩阵表示

邻接矩阵具有以下特性：

- 顶点不能与自身相连，因此邻接矩阵主对角线元素没有意义。
- 对于无向图，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，则可表示有权图。

使用邻接矩阵表示图时，我们可以直接访问矩阵元素以获取边，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  $O(n^2)$ ，内存占用较多。

### 邻接表

「邻接表 Adjacency List」使用  $n$  个链表来表示图，链表节点表示顶点。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）。

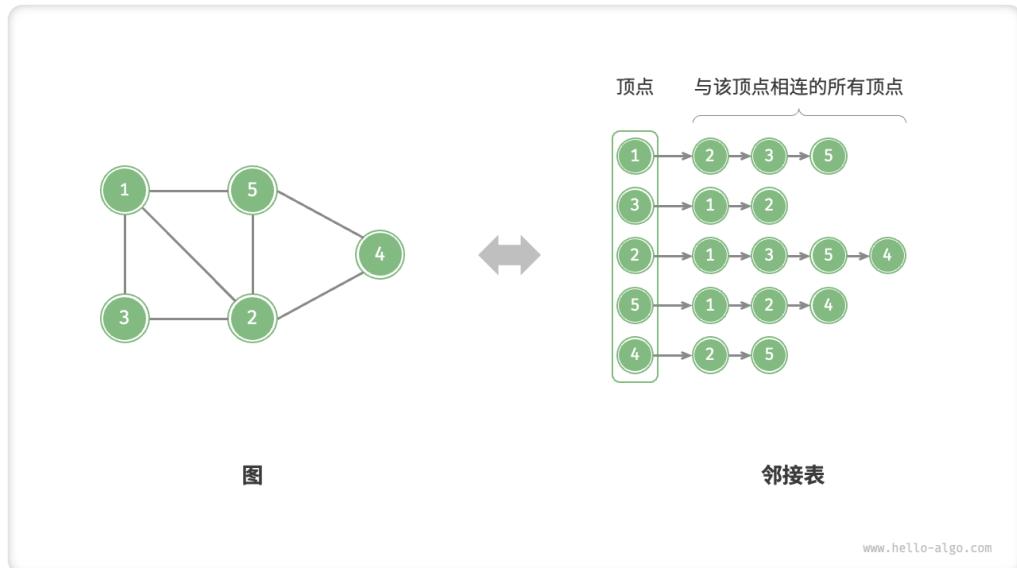


Figure 10-6. 图的邻接表表示

邻接表仅存储实际存在的边，而边的总数通常远小于  $n^2$ ，因此它更加节省空间。然而，在邻接表中需要通过遍历链表来查找边，因此其时间效率不如邻接矩阵。

观察上图可发现，邻接表结构与哈希表中的「链地址法」非常相似，因此我们也可以采用类似方法来优化效率。例如，当链表较长时，可以将链表转化为 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；此外，还可以将链表转换为哈希表，将时间复杂度降低至  $O(1)$ 。

#### 10.1.4. 图常见应用

实际应用中，许多系统都可以用图来建模，相应的待求解问题也可以约化为图计算问题。

	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

## 10.2. 图基础操作

图的基础操作可分为对「边」的操作和对「顶点」的操作。在「邻接矩阵」和「邻接表」两种表示方法下，实现方式有所不同。

### 10.2.1. 基于邻接矩阵的实现

给定一个顶点数量为  $n$  的无向图，则有：

- **添加或删除边**：直接在邻接矩阵中修改指定的边即可，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。

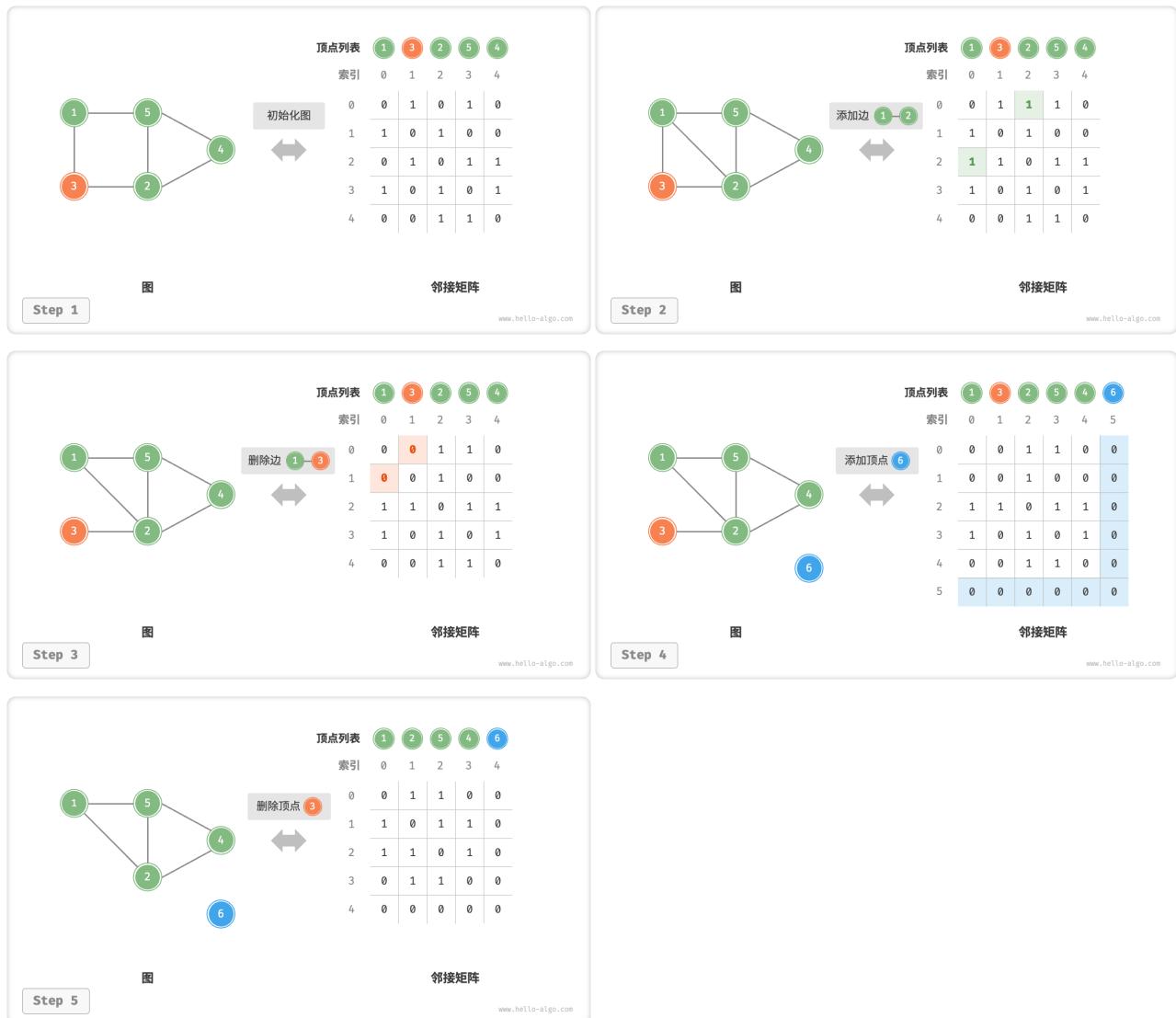


Figure 10-7. 邻接矩阵的初始化、增删边、增删顶点

以下是基于邻接矩阵表示图的实现代码。

```
# == File: graph_adjacency_matrix.py ==
class GraphAdjMat:
    """ 基于邻接矩阵实现的无向图类"""

    # 顶点列表，元素代表“顶点值”，索引代表“顶点索引”
    vertices: list[int] = []
    # 邻接矩阵，行列索引对应“顶点索引”
    adj_mat: list[list[int]] = []

    def __init__(self, vertices: list[int], edges: list[list[int]]) -> None:
        """ 构造方法"""
        self.vertices: list[int] = []
        self.adj_mat: list[list[int]] = []
        # 添加顶点
        for val in vertices:
            self.add_vertex(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 e in edges:
            self.add_edge(e[0], e[1])

    def size(self) -> int:
        """ 获取顶点数量"""
        return len(self.vertices)

    def add_vertex(self, val: int) -> None:
        """ 添加顶点"""
        n = self.size()
        # 向顶点列表中添加新顶点的值
        self.vertices.append(val)
        # 在邻接矩阵中添加一行
        new_row = [0] * n
        self.adj_mat.append(new_row)
        # 在邻接矩阵中添加一列
        for row in self.adj_mat:
            row.append(0)

    def remove_vertex(self, index: int) -> None:
        """ 删除顶点"""
        if index >= self.size():
            raise IndexError()
        # 在顶点列表中移除索引 index 的顶点
        self.vertices.pop(index)
        # 在邻接矩阵中删除索引 index 的行
        self.adj_mat.pop(index)
        # 在邻接矩阵中删除索引 index 的列
```

```
for row in self.adj_mat:
    row.pop(index)

def add_edge(self, i: int, j: int) -> None:
    """ 添加边 """
    # 参数 i, j 对应 vertices 元素索引
    # 索引越界与相等处理
    if i < 0 or j < 0 or i >= self.size() or j >= self.size() or i == j:
        raise IndexError()
    # 在无向图中, 邻接矩阵沿主对角线对称, 即满足 (i, j) == (j, i)
    self.adj_mat[i][j] = 1
    self.adj_mat[j][i] = 1

def remove_edge(self, i: int, j: int) -> None:
    """ 删除边 """
    # 参数 i, j 对应 vertices 元素索引
    # 索引越界与相等处理
    if i < 0 or j < 0 or i >= self.size() or j >= self.size() or i == j:
        raise IndexError()
    self.adj_mat[i][j] = 0
    self.adj_mat[j][i] = 0

def print(self) -> None:
    """ 打印邻接矩阵 """
    print(" 顶点列表 =", self.vertices)
    print(" 邻接矩阵 =")
    print_matrix(self.adj_mat)
```

### 10.2.2. 基于邻接表的实现

设无向图的顶点总数为  $n$ 、边总数为  $m$ ，则有：

- **添加边**：在顶点对应链表的末尾添加边即可，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找并删除指定边，使用  $O(m)$  时间。在无向图中，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，并将新增顶点作为链表头节点，使用  $O(1)$  时间。
- **删除顶点**：需遍历整个邻接表，删除包含指定顶点的所有边，使用  $O(1)$  时间。
- **初始化**：在邻接表中创建  $n$  个顶点和  $2m$  条边，使用  $O(n + m)$  时间。



Figure 10-8. 邻接表的初始化、增删边、增删顶点

以下是基于邻接表实现图的代码示例。细心的同学可能注意到，我们在邻接表中使用 `Vertex` 节点类来表示顶点，这样做的原因有：

- 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  $> i$  的索引全部减 1，这样操作效率较低。
- 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.py ==
class GraphAdjList:
    """ 基于邻接表实现的无向图类 """
```

```
def __init__(self, edges: list[list[Vertex]]) -> None:
    """ 构造方法 """
    # 邻接表, key: 顶点, value: 该顶点的所有邻接顶点
    self.adj_list = dict[Vertex, Vertex]()
    # 添加所有顶点和边
    for edge in edges:
        self.add_vertex(edge[0])
        self.add_vertex(edge[1])
        self.add_edge(edge[0], edge[1])

def size(self) -> int:
    """ 获取顶点数量 """
    return len(self.adj_list)

def add_edge(self, vet1: Vertex, vet2: Vertex) -> None:
    """ 添加边 """
    if vet1 not in self.adj_list or vet2 not in self.adj_list or vet1 == vet2:
        raise ValueError()
    # 添加边 vet1 - vet2
    self.adj_list[vet1].append(vet2)
    self.adj_list[vet2].append(vet1)

def remove_edge(self, vet1: Vertex, vet2: Vertex) -> None:
    """ 删除边 """
    if vet1 not in self.adj_list or vet2 not in self.adj_list or vet1 == vet2:
        raise ValueError()
    # 删除边 vet1 - vet2
    self.adj_list[vet1].remove(vet2)
    self.adj_list[vet2].remove(vet1)

def add_vertex(self, vet: Vertex) -> None:
    """ 添加顶点 """
    if vet in self.adj_list:
        return
    # 在邻接表中添加一个新链表
    self.adj_list[vet] = []

def remove_vertex(self, vet: Vertex) -> None:
    """ 删除顶点 """
    if vet not in self.adj_list:
        raise ValueError()
    # 在邻接表中删除顶点 vet 对应的链表
    self.adj_list.pop(vet)
    # 遍历其他顶点的链表, 删除所有包含 vet 的边
    for vertex in self.adj_list:
```

```
if vet in self.adj_list[vertex]:  
    self.adj_list[vertex].remove(vet)  
  
def print(self) -> None:  
    """ 打印邻接表 """  
    print(" 邻接表 =")  
    for vertex in self.adj_list:  
        tmp = [v.val for v in self.adj_list[vertex]]  
        print(f"{vertex.val}: {tmp},")
```

### 10.2.3. 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	$O(n^2)$	$O(n + m)$	$O(n)$
内存空间占用	$O(n^2)$	$O(n + m)$	$O(n + m)$

观察上表，似乎邻接表（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。综合来看，邻接矩阵体现了“以空间换时间”的原则，而邻接表体现了“以时间换空间”的原则。

## 10.3. 图的遍历



### 图与树的关系

树代表的是“一对多”的关系，而图则具有更高的自由度，可以表示任意的“多对多”关系。因此，我们可以把树看作是图的一种特例。显然，**树的遍历操作也是图的遍历操作的一种特例**，建议你在学习本章节时融会贯通两者的概念与实现方法。

「图」和「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。

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

### 10.3.1. 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。具体来说，从某个顶点出发，先遍历该顶点的所有邻接顶点，然后遍历下一个顶点的所有邻接顶点，以此类推，直至所有顶点访问完毕。

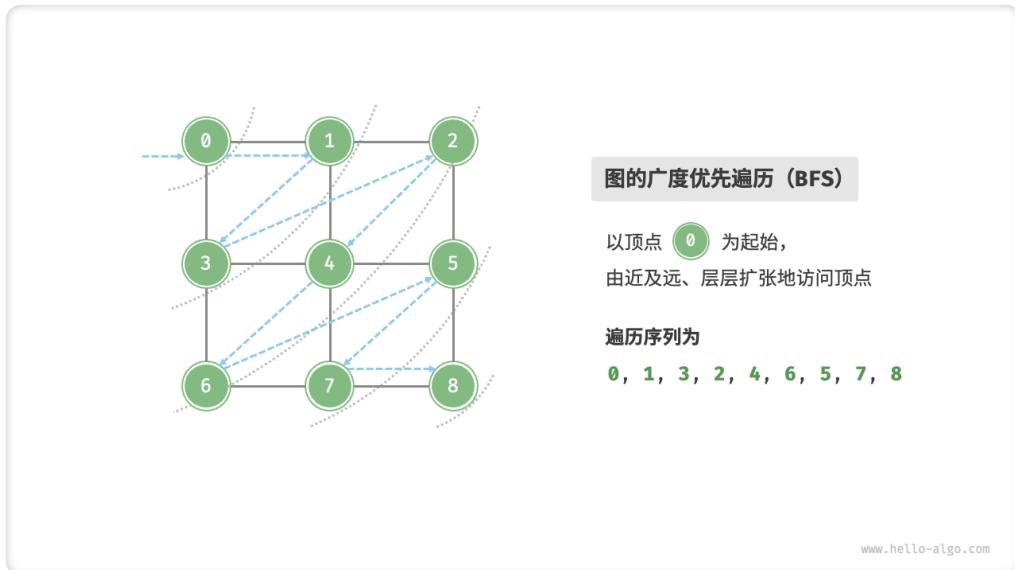


Figure 10-9. 图的广度优先遍历

#### 算法实现

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

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，然后将该顶点的所有邻接顶点加入到队列尾部；
3. 循环步骤 2.，直到所有顶点被访问完成后结束；

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

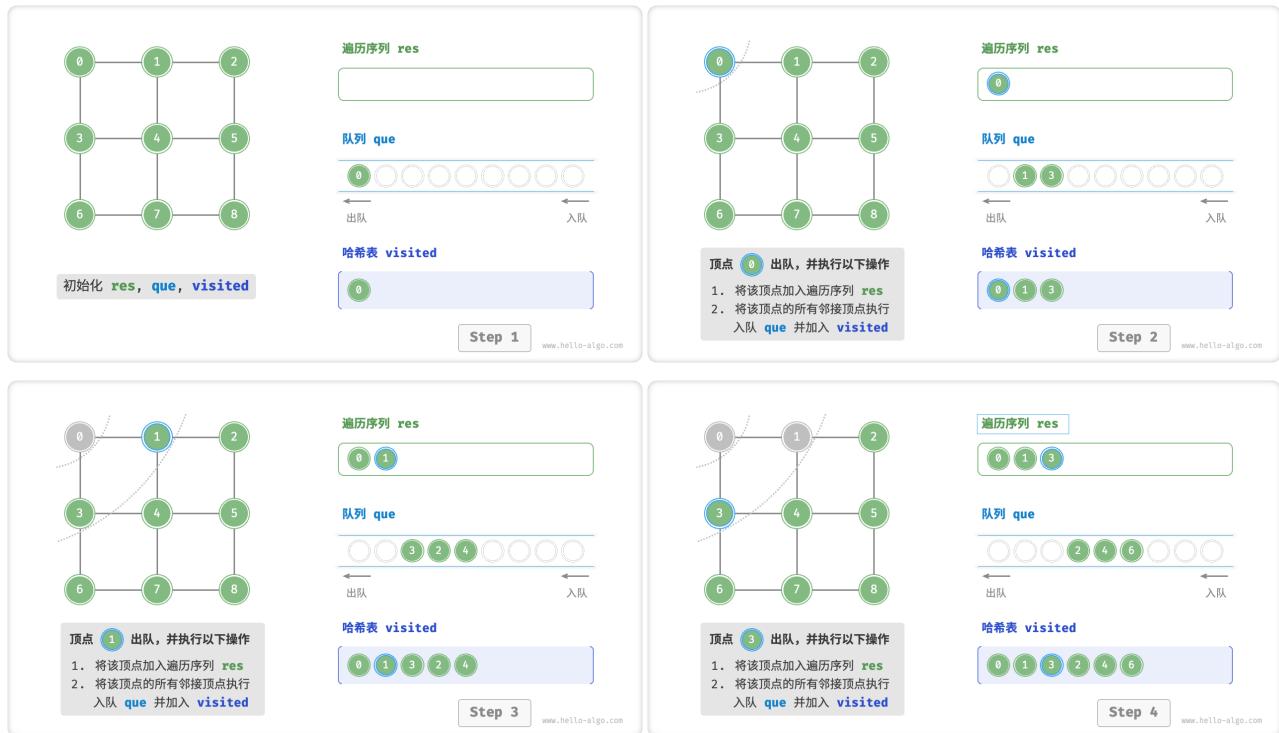
```
# === File: graph_bfs.py ===
def graph_bfs(graph: GraphAdjList, start_vet: Vertex) -> list[Vertex]:
    """ 广度优先遍历 BFS """
    # 使用邻接表来表示图，以便获取指定顶点的所有邻接顶点
    # 顶点遍历序列
    res = []
    # 哈希表，用于记录已被访问过的顶点
```

```

visited = set[Vertex]([start_vet])
# 队列用于实现 BFS
que = deque[Vertex]([start_vet])
# 以顶点 vet 为起点，循环直至访问完所有顶点
while len(que) > 0:
    vet = que.popleft() # 队首顶点出队
    res.append(vet) # 记录访问顶点
    # 遍历该顶点的所有邻接顶点
    for adj_vet in graph.adj_list[vet]:
        if adj_vet in visited:
            continue # 跳过已被访问过的顶点
        que.append(adj_vet) # 只入队未访问的顶点
        visited.add(adj_vet) # 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。



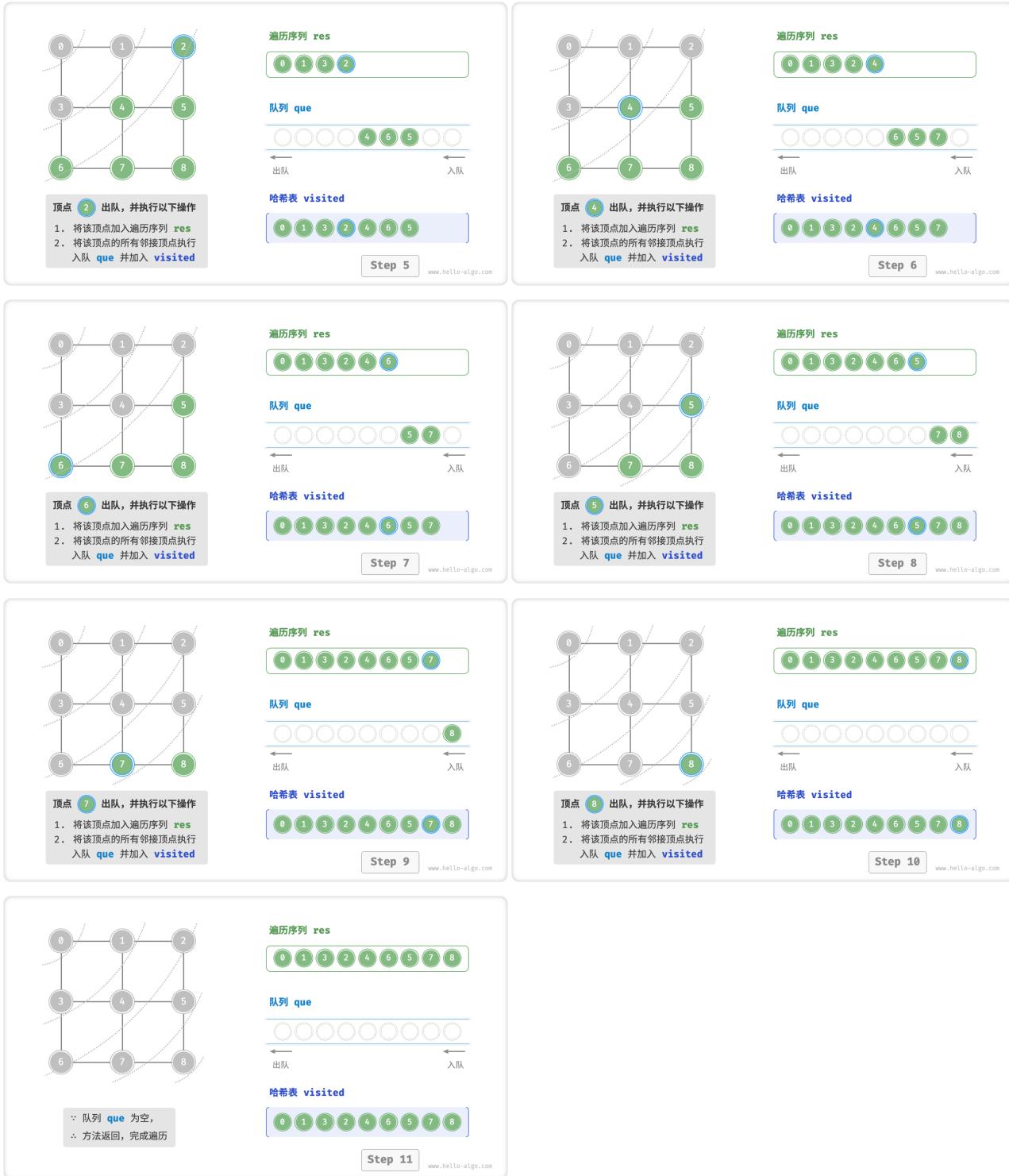


Figure 10-10. 图的广度优先遍历步骤



### 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按“由近及远”的顺序遍历，而多个相同距离的顶点的遍历顺序是允许被任意打乱的。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。

## 复杂度分析

**时间复杂度：**所有顶点都会入队并出队一次，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。

### 10.3.2. 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，访问当前顶点的某个邻接顶点，直到走到尽头时返回，再继续走到尽头并返回，以此类推，直至所有顶点遍历完成。

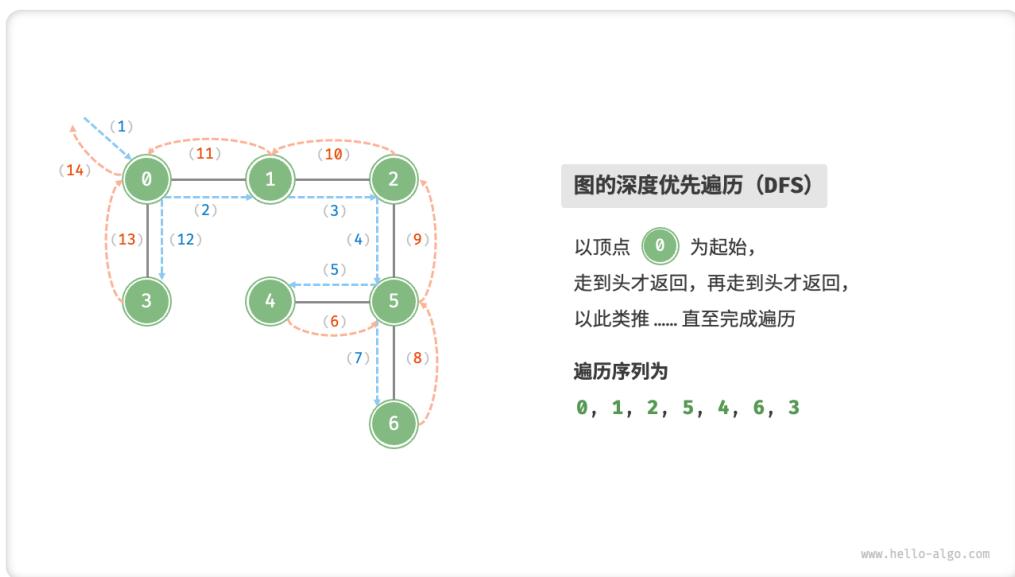


Figure 10-11. 图的深度优先遍历

## 算法实现

这种“走到尽头 + 回溯”的算法形式通常基于递归来实现。与 BFS 类似，在 DFS 中我们还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.py ==
def dfs(graph: GraphAdjList, visited: set[Vertex], res: list[Vertex], vet: Vertex):
    """ 深度优先遍历 DFS 辅助函数 """
    res.append(vet) # 记录访问顶点
    visited.add(vet) # 标记该顶点已被访问
    # 遍历该顶点的所有邻接顶点
    for adjVet in graph.adj_list[vet]:
        if adjVet in visited:
            continue # 跳过已被访问过的顶点
        # 递归访问邻接顶点
        dfs(graph, visited, res, adjVet)

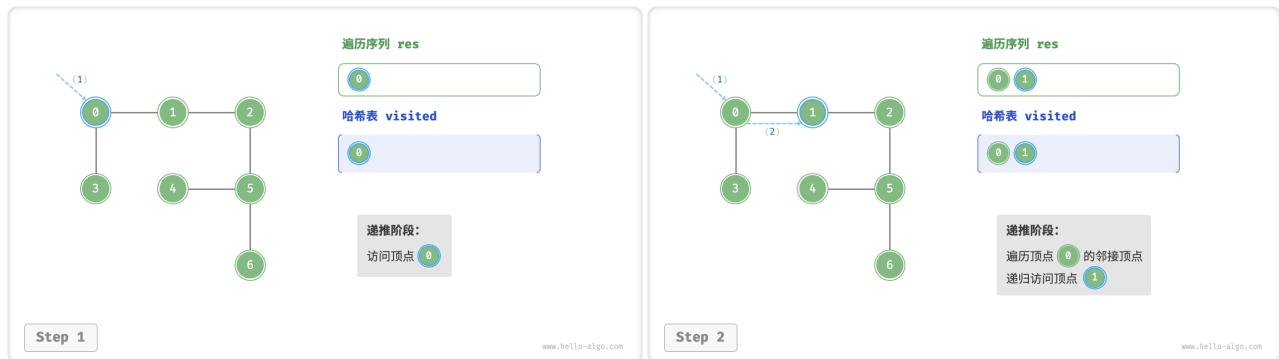
def graph_dfs(graph: GraphAdjList, start_vet: Vertex) -> list[Vertex]:
    """ 深度优先遍历 DFS """
    # 顶点遍历序列
    res = []
    # 哈希表，用于记录已被访问过的顶点
    visited = set[Vertex]()
    dfs(graph, visited, res, start_vet)
    return res

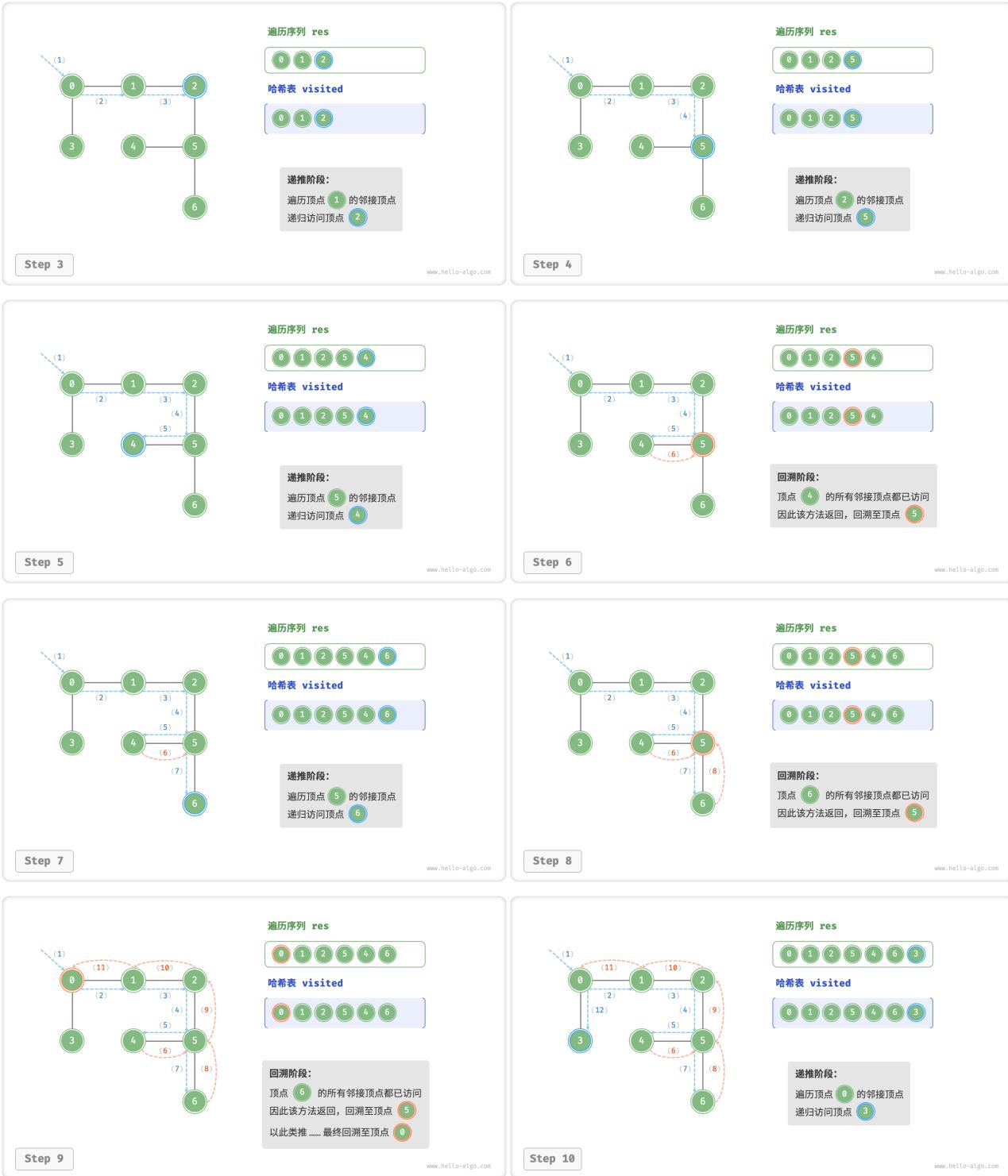
```

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，表示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，表示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，建议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。





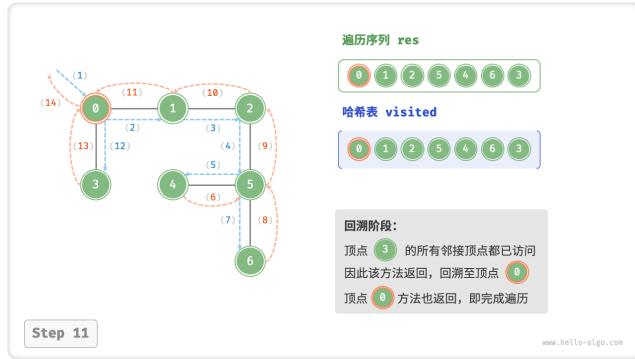


Figure 10-12. 图的深度优先遍历步骤



### 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可以，即邻接顶点的顺序可以任意打乱，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，它们展示了三种不同的遍历优先级，然而这三者都属于深度优先遍历。

## 复杂度分析

**时间复杂度：**所有顶点都会被访问 1 次，使用  $O(|V|)$  时间；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` 顶点数量最多为  $|V|$ ，递归深度最大为  $|V|$ ，因此使用  $O(|V|)$  空间。

## 10.4. 小结

- 图由顶点和边组成，可以被表示为一组顶点和一组边构成的集合。
- 相较于线性关系（链表）和分治关系（树），网络关系（图）具有更高的自由度，因而更为复杂。
- 有向图的边具有方向性，连通图中的任意顶点均可达，有权图的每条边都包含权重变量。
- 邻接矩阵利用矩阵来表示图，每一行（列）代表一个顶点，矩阵元素代表边，用 1 或 0 表示两个顶点之间有边或无边。邻接矩阵在增删查操作上效率很高，但空间占用较多。
- 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。邻接表相对于邻接矩阵更加节省空间，但由于需要遍历链表来查找边，时间效率较低。
- 当邻接表中的链表过长时，可以将其转换为红黑树或哈希表，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”。
- 图可用于建模各类现实系统，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，通常借助队列实现。

- 图的深度优先遍历是一种优先走到底、无路可走时再回溯的搜索方式，常基于递归来实现。

# 11. 排序算法

## 11.1. 排序算法

「排序算法 Sorting Algorithm」用于对一组数据按照特定顺序进行排列。排序算法有着广泛的应用，因为有序数据通常能够被更有效地查找、分析和处理。

在排序算法中，数据类型可以是整数、浮点数、字符或字符串等；顺序的判断规则可根据需求设定，如数字大小、字符 ASCII 码顺序或自定义规则。

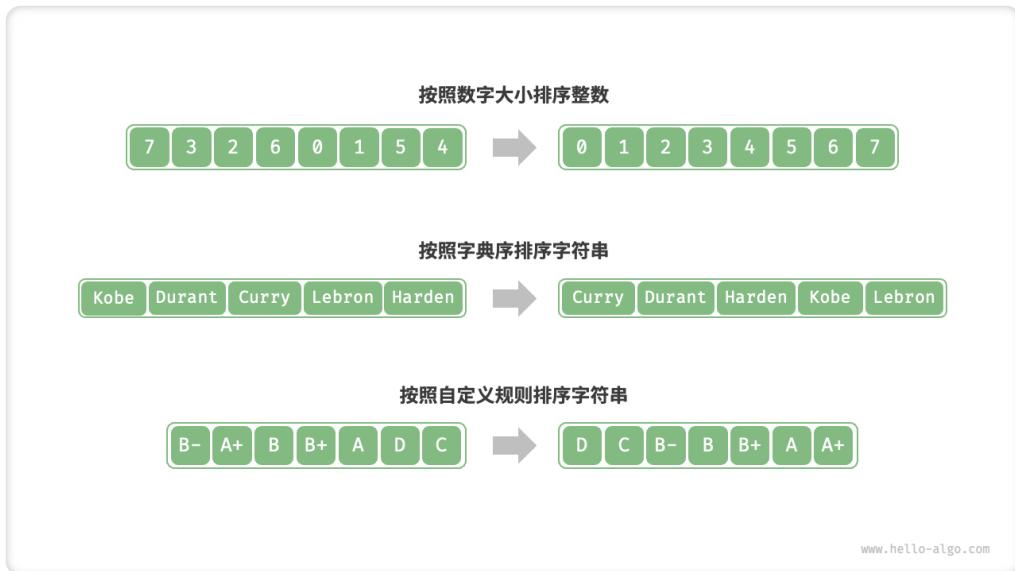


Figure 11-1. 数据类型和判断规则示例

### 11.1.1. 评价维度

**运行效率：**我们期望排序算法的时间复杂度尽量低，且总体操作数量较少（即时间复杂度中的常数项降低）。对于大数据量情况，运行效率显得尤为重要。

**就地性：**顾名思义，「原地排序」通过在原数组上直接操作实现排序，无需借助额外的辅助数组，从而节省内存。通常情况下，原地排序的数据搬运操作较少，运行速度也更快。

**稳定性：**「稳定排序」在完成排序后，相等元素在数组中的相对顺序不发生改变。稳定排序是优良特性，也是多级排序场景的必要条件。

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。在这种情况下，「非稳定排序」可能导致输入数据的有序性丧失。

```
# 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)
```

**自适应性:**「自适应排序」的时间复杂度会受输入数据的影响，即最佳、最差、平均时间复杂度并不完全相等。

自适应性需要根据具体情况来评估。如果最差时间复杂度差于平均时间复杂度，说明排序算法在某些数据下性能可能劣化，因此被视为负面属性；而如果最佳时间复杂度优于平均时间复杂度，则被视为正面属性。

**是否基于比较:**「基于比较的排序」依赖于比较运算符（ $<$ ,  $=$ ,  $>$ ）来判断元素的相对顺序，从而排序整个数组，理论最优时间复杂度为  $O(n \log n)$ 。而「非比较排序」不使用比较运算符，时间复杂度可达  $O(n)$ ，但其通用性相对较差。

### 11.1.2. 理想排序算法

**运行快、原地、稳定、正向自适应、通用性好。**显然，迄今为止尚未发现兼具以上所有特性的排序算法。因此，在选择排序算法时，需要根据具体的数据特点和问题需求来决定。

接下来，我们将共同学习各种排序算法，并基于上述评价维度对各个排序算法的优缺点进行分析。

## 11.2. 冒泡排序

「冒泡排序 Bubble Sort」的工作原理类似于泡泡在水中的浮动。在水中，较大的泡泡会最先浮到水面。

「冒泡操作」利用元素交换操作模拟了上述过程，具体做法为：从数组最左端开始向右遍历，依次比较相邻元素大小，如果“左元素  $>$  右元素”就交换它们。遍历完成后，最大的元素会被移动到数组的最右端。

在完成一次冒泡操作后，数组的最大元素已位于正确位置，接下来只需对剩余  $n - 1$  个元素进行排序。



Figure 11-2. 冒泡操作步骤

### 11.2.1. 算法流程

设输入数组长度为  $n$ ，整个冒泡排序的步骤为：

- 完成第一轮「冒泡」后，数组的最大元素已位于正确位置，接下来只需对剩余  $n - 1$  个元素进行排序；
- 对剩余  $n - 1$  个元素执行冒泡操作，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；
- 如此类推，经过  $n - 1$  轮冒泡操作，整个数组便完成排序；



Figure 11-3. 冒泡排序流程

```
# == File: bubble_sort.py ==
def bubble_sort(nums: list[int]) -> None:
    """ 冒泡排序 """
    n: int = len(nums)
    # 外循环：待排序元素数量为 n-1, n-2, ..., 1
    for i in range(n - 1, 0, -1):
        # 内循环：冒泡操作
        for j in range(i):
            if nums[j] > nums[j + 1]:
                # 交换 nums[j] 与 nums[j + 1]
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
```

### 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依次为  $n - 1, n - 2, \dots, 2, 1$ ，总和为  $\frac{(n-1)n}{2}$ ，因此使用  $O(n^2)$  时间。在引入下文的 `flag` 优化后，最佳时间复杂度可达到  $O(n)$ ，所以它是“自适应排序”。

**空间复杂度  $O(1)$ ：**指针  $i, j$  使用常数大小的额外空间，因此是“原地排序”。

由于冒泡操作中遇到相等元素不交换，因此冒泡排序是“稳定排序”。

### 11.2.3. 效率优化

我们发现，如果某轮冒泡操作中没有执行任何交换操作，说明数组已经完成排序，可直接返回结果。因此，可以增加一个标志位 `flag` 来监测这种情况，一旦出现就立即返回。

经过优化，冒泡排序的最差和平均时间复杂度仍为  $O(n^2)$ ；但当输入数组完全有序时，可达到最佳时间复杂度  $O(n)$ 。

```
# === File: bubble_sort.py ===
def bubble_sort_with_flag(nums: list[int]) -> None:
    """ 冒泡排序（标志优化） """
    n: int = len(nums)
    # 外循环：待排序元素数量为 n-1, n-2, ..., 1
    for i in range(n - 1, 0, -1):
        flag: bool = False # 初始化标志位
        # 内循环：冒泡操作
        for j in range(i):
            if nums[j] > nums[j + 1]:
                # 交换 nums[j] 与 nums[j + 1]
                nums[j], nums[j + 1] = nums[j + 1], nums[j]
                flag = True # 记录交换元素
        if not 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。具体来说，选择一个待排序的元素作为基准值 `base`，将 `base` 与其左侧已排序区间的元素逐一比较大小，并将其插入到正确的位置。

回顾数组插入操作，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。

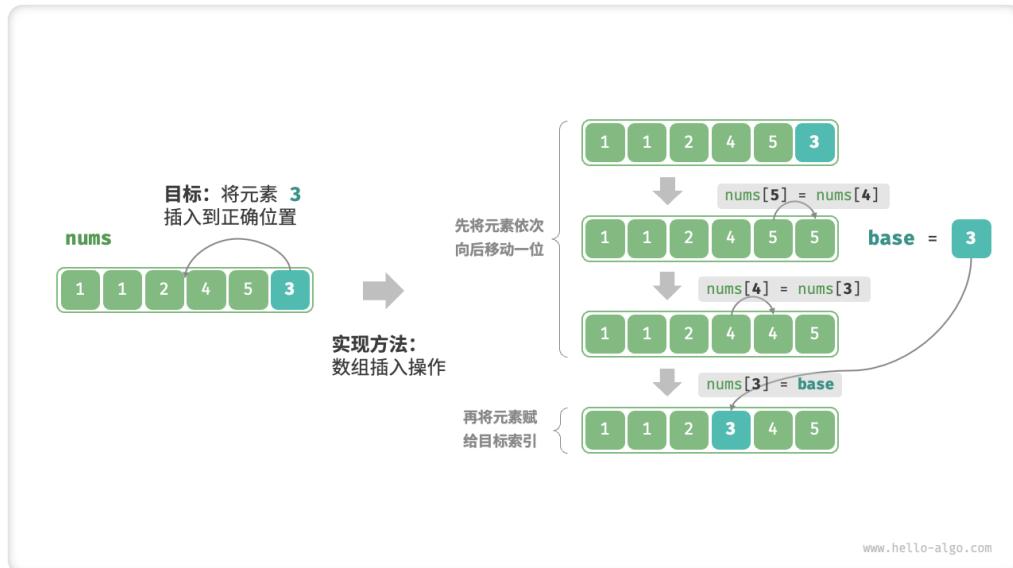


Figure 11-4. 单次插入操作

### 11.3.1. 算法流程

插入排序的整体流程如下：

- 首先，选取数组的第 2 个元素作为 `base`，执行插入操作后，数组的前 2 个元素已排序。
- 接着，选取第 3 个元素作为 `base`，执行插入操作后，数组的前 3 个元素已排序。
- 以此类推，在最后一轮中，选取数组尾元素作为 `base`，执行插入操作后，所有元素均已排序。



Figure 11-5. 插入排序流程

```
# == File: insertion_sort.py ==
def insertion_sort(nums: list[int]) -> None:
    """ 插入排序 """
    # 外循环: base = nums[1], nums[2], ..., nums[n-1]
    for i in range(1, len(nums)):
        base: int = nums[i]
        j: int = i - 1
        # 内循环: 将 base 插入到左边的正确位置
        while j >= 0 and nums[j] > base:
            nums[j + 1] = nums[j] # 1. 将 nums[j] 向右移动一位
            j -= 1
        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，每次插入操作分别需要循环  $n-1, n-2, \dots, 2, 1$  次，求和得到  $\frac{(n-1)n}{2}$ ，因此时间复杂度为  $O(n^2)$ 。当输入数组完全有序时，插入排序达到最佳时间复杂度  $O(n)$ ，因此是“自适应排序”。

**空间复杂度  $O(1)$** ：指针  $i, j$  使用常数大小的额外空间，所以插入排序是“原地排序”。

在插入操作过程中，我们会将元素插入到相等元素的右侧，不会改变它们的顺序，因此是“稳定排序”。

### 11.3.3. 插入排序优势

回顾冒泡排序和插入排序的复杂度分析，两者的循环轮数都是  $\frac{(n-1)n}{2}$ 。然而，它们之间存在以下差异：

- 冒泡操作基于元素交换实现，需要借助一个临时变量，共涉及 3 个单元操作；
- 插入操作基于元素赋值实现，仅需 1 个单元操作；

粗略估计下来，冒泡排序的计算开销约为插入排序的 3 倍，因此插入排序更受欢迎。实际上，许多编程语言（如 Java）的内置排序函数都采用了插入排序，大致思路为：

- 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  $O(n^2)$ ；

尽管插入排序的时间复杂度高于快速排序，但在数据量较小的情况下，插入排序实际上更快。这是因为在数据量较小时，复杂度中的常数项（即每轮中的单元操作数量）起主导作用。这个现象与「线性查找」和「二分查找」的情况相似。

## 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，运行高效，应用广泛。

快速排序的核心操作是「哨兵划分」，其目标是：选择数组中的某个元素作为“基准数”，将所有小于基准数的元素移到其左侧，而大于基准数的元素移到其右侧。具体来说，哨兵划分的流程为：

1. 选取数组最左端元素作为基准数，初始化两个指针  $i$  和  $j$  分别指向数组的两端；
2. 设置一个循环，在每轮中使用  $i$  ( $j$ ) 分别寻找第一个比基准数大 (小) 的元素，然后交换这两个元素；
3. 循环执行步骤 2.，直到  $i$  和  $j$  相遇时停止，最后将基准数交换至两个子数组的分界线；

哨兵划分完成后，原数组被划分成三部分：左子数组、基准数、右子数组，且满足“左子数组任意元素  $\leq$  基准数  $\leq$  右子数组任意元素”。因此，我们接下来只需对这两个子数组进行排序。



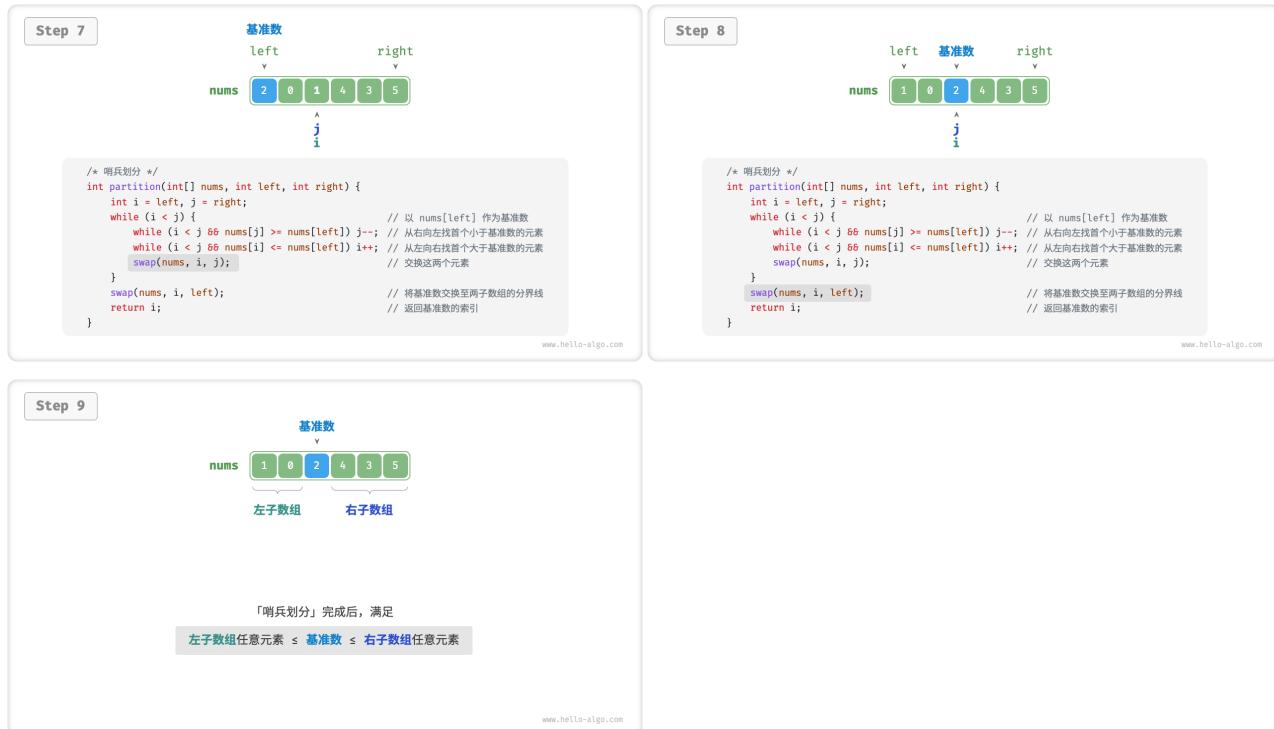


Figure 11-6. 哨兵划分步骤



### 快速排序的分治思想

哨兵划分的实质是将一个较长数组的排序问题简化为两个较短数组的排序问题。

```

# === File: quick_sort.py ===
def partition(self, nums: list[int], left: int, right: int) -> int:
    """
    哨兵划分"""

    # 以 nums[left] 作为基准数
    i, j = left, right
    while i < j:
        while i < j and nums[j] >= nums[left]:
            j -= 1 # 从右向左找首个小于基准数的元素
        while i < j and nums[i] <= nums[left]:
            i += 1 # 从左向右找首个大于基准数的元素
        # 元素交换
        nums[i], nums[j] = nums[j], nums[i]
    # 将基准数交换至两子数组的分界线
    nums[i], nums[left] = nums[left], nums[i]
    return i # 返回基准数的索引

```

### 11.4.1. 算法流程

1. 首先，对原数组执行一次「哨兵划分」，得到待排序的左子数组和右子数组；
2. 然后，对左子数组和右子数组分别递归执行「哨兵划分」；
3. 持续递归，直至子数组长度为 1 时终止，从而完成整个数组的排序；



Figure 11-7. 快速排序流程

```
# === File: quick_sort.py ===
def quick_sort(self, nums: list[int], left: int, right: int) -> None:
    """ 快速排序 """
    # 子数组长度为 1 时终止递归
    if left >= right:
        return
    # 哨兵划分
    pivot: int = self.partition(nums, left, right)
    # 递归左子数组、右子数组
    self.quick_sort(nums, left, pivot - 1)
    self.quick_sort(nums, pivot + 1, right)
```

### 11.4.2. 算法特性

**时间复杂度  $O(n \log n)$** ：在平均情况下，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。

在最差情况下，每轮哨兵划分操作都将长度为  $n$  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；因此快速排序是“自适应排序”。

**空间复杂度  $O(n)$** ：在输入数组完全倒序的情况下，达到最差递归深度  $n$ 。由于未使用辅助数组，因此算法是“原地排序”。

在哨兵划分的最后一步，基准数可能会被交换至相等元素的右侧，因此是“非稳定排序”。

### 11.4.3. 快排为什么快？

从名称上就能看出，快速排序在效率方面应该具有一定的优势。尽管快速排序的平均时间复杂度与「归并排序」和「堆排序」相同，但通常快速排序的效率更高，原因如下：

- **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  $O(n^2)$ ，没有归并排序稳定，但在绝大多数情况下，快速排序能在  $O(n \log n)$  的时间复杂度下运行。
- **缓存使用效率高：**在执行哨兵划分操作时，系统可将整个子数组加载到缓存，因此访问元素的效率较高。而像「堆排序」这类算法需要跳跃式访问元素，从而缺乏这一特性。
- **复杂度的常数系数低：**在上述三种算法中，快速排序的比较、赋值、交换等操作的总数量最少。这与「插入排序」比「冒泡排序」更快的原因类似。

### 11.4.4. 基准数优化

快速排序在某些输入下的时间效率可能降低。举一个极端例子，假设输入数组是完全倒序的，由于我们选择最左端元素作为基准数，那么在哨兵划分完成后，基准数被交换至数组最右端，导致左子数组长度为  $n - 1$ 、右子数组长度为 0。如此递归下去，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」。

为了尽量避免这种情况发生，**我们可以优化哨兵划分中的基准数的选取策略**。例如，我们可以随机选取一个元素作为基准数。然而，如果运气不佳，每次都选到不理想的基准数，效率仍然不尽如人意。

需要注意的是，编程语言通常生成的是“伪随机数”。如果我们针对伪随机数序列构建一个特定的测试样例，那么快速排序的效率仍然可能劣化。

为了进一步改进，我们可以在数组中选取三个候选元素（通常为数组的首、尾、中点元素），**并将这三个候选元素的中位数作为基准数**。这样一来，基准数“既不太小也不太大”的概率将大幅提升。当然，我们还可以选取更多候选元素，以进一步提高算法的稳健性。采用这种方法后，时间复杂度劣化至  $O(n^2)$  的概率大大降低。

```
# === File: quick_sort.py ===
def median_three(self, nums: list[int], left: int, mid: int, right: int) -> int:
    """ 选取三个元素的中位数 """
    # 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
    elif (nums[mid] < nums[left]) ^ (nums[mid] < nums[right]):
        return mid
    return right

def partition(self, nums: list[int], left: int, right: int) -> int:
    """ 哨兵划分（三数取中值） """
    # 以 nums[left] 作为基准数
```

```

med: int = self.median_three(nums, left, (left + right) // 2, right)
# 将中位数交换至数组最左端
nums[left], nums[med] = nums[med], nums[left]
# 以 nums[left] 作为基准数
i, j = left, right
while i < j:
    while i < j and nums[j] >= nums[left]:
        j -= 1 # 从右向左找首个小于基准数的元素
    while i < j and nums[i] <= nums[left]:
        i += 1 # 从左向右找首个大于基准数的元素
    # 元素交换
    nums[i], nums[j] = nums[j], nums[i]
# 将基准数交换至两子数组的分界线
nums[i], nums[left] = nums[left], nums[i]
return i # 返回基准数的索引

```

#### 11.4.5. 尾递归优化

在某些输入下，快速排序可能占用空间较多。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，递归树的高度会达到  $n - 1$ ，此时需要占用  $O(n)$  大小的栈帧空间。

为了防止栈帧空间的累积，我们可以在每轮哨兵排序完成后，比较两个子数组的长度，仅对较短的子数组进行递归。由于较短子数组的长度不会超过  $\frac{n}{2}$ ，因此这种方法能确保递归深度不超过  $\log n$ ，从而将最差空间复杂度优化至  $O(\log n)$ 。

```

# === File: quick_sort.py ===
def quick_sort(self, nums: list[int], left: int, right: int) -> None:
    """ 快速排序（尾递归优化） """
    # 子数组长度为 1 时终止
    while left < right:
        # 哨兵划分操作
        pivot: int = self.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:
            self.quick_sort(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:
            self.quick_sort(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]

```



### 哨兵划分中“从右往左查找”与“从左往右查找”的顺序可以交换吗？

不行，当我们以最左端元素为基准数时，必须先“从右往左查找”再“从左往右查找”。这个结论有些反直觉，我们来剖析一下原因。

哨兵划分 `partition()` 的最后一步是交换 `nums[left]` 和 `nums[i]`。完成交换后，基准数左边的元素都  $\leq$  基准数，这就要求最后一步交换前 `nums[left] \geq nums[i]` 必须成立。假设我们先“从左往右查找”，那么如果找不到比基准数更小的元素，则会在  $i == j$  时跳出循环，此时可能 `nums[j] == nums[i] > nums[left]`。也就是说，此时最后一步交换操作会把一个比基准数更大的元素交换至数组最左端，导致哨兵划分失败。

举个例子，给定数组 `[0, 0, 0, 0, 1]`，如果先“从左向右查找”，哨兵划分后数组为 `[1, 0, 0, 0, 0]`，这个结果是不正确的。

再深入思考一下，如果我们选择 `nums[right]` 为基准数，那么正好反过来，必须先“从左往右查找”。

## 11.5. 归并排序

「归并排序 Merge Sort」基于分治思想实现排序，包含“划分”和“合并”两个阶段：

1. 划分阶段：通过递归不断地将数组从中点处分开，将长数组的排序问题转换为短数组的排序问题；
2. 合并阶段：当子数组长度为 1 时终止划分，开始合并，持续地将左右两个较短的有序数组合并为一个较长的有序数组，直至结束；

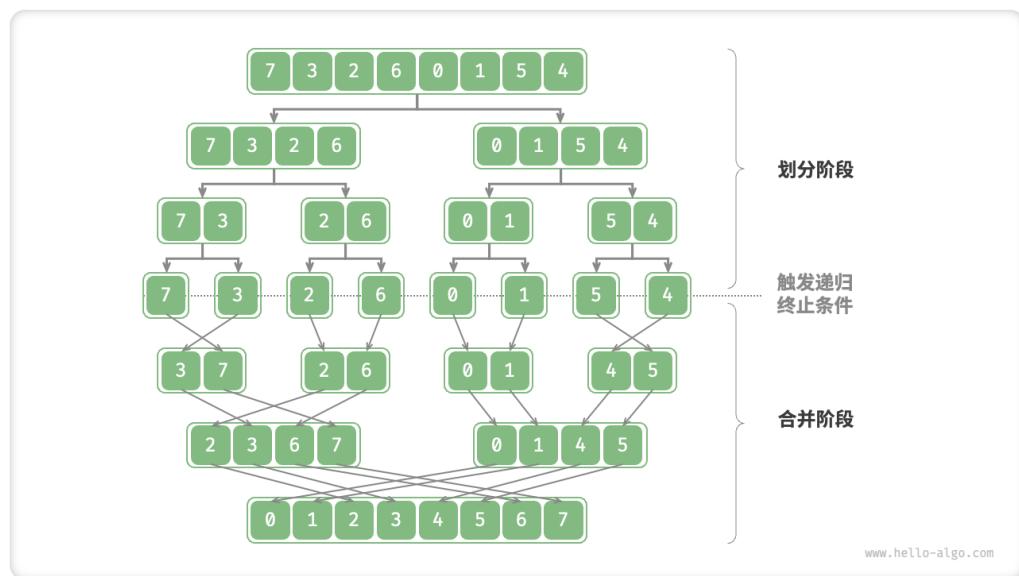


Figure 11-8. 归并排序的划分与合并阶段

### 11.5.1. 算法流程

“划分阶段”从顶到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. 递归执行步骤 1.，直至子数组区间长度为 1 时，终止递归划分；

“合并阶段”从底至顶地将左子数组和右子数组合并为一个有序数组。需要注意的是，从长度为 1 的子数组开始合并，合并阶段中的每个子数组都是有序的。

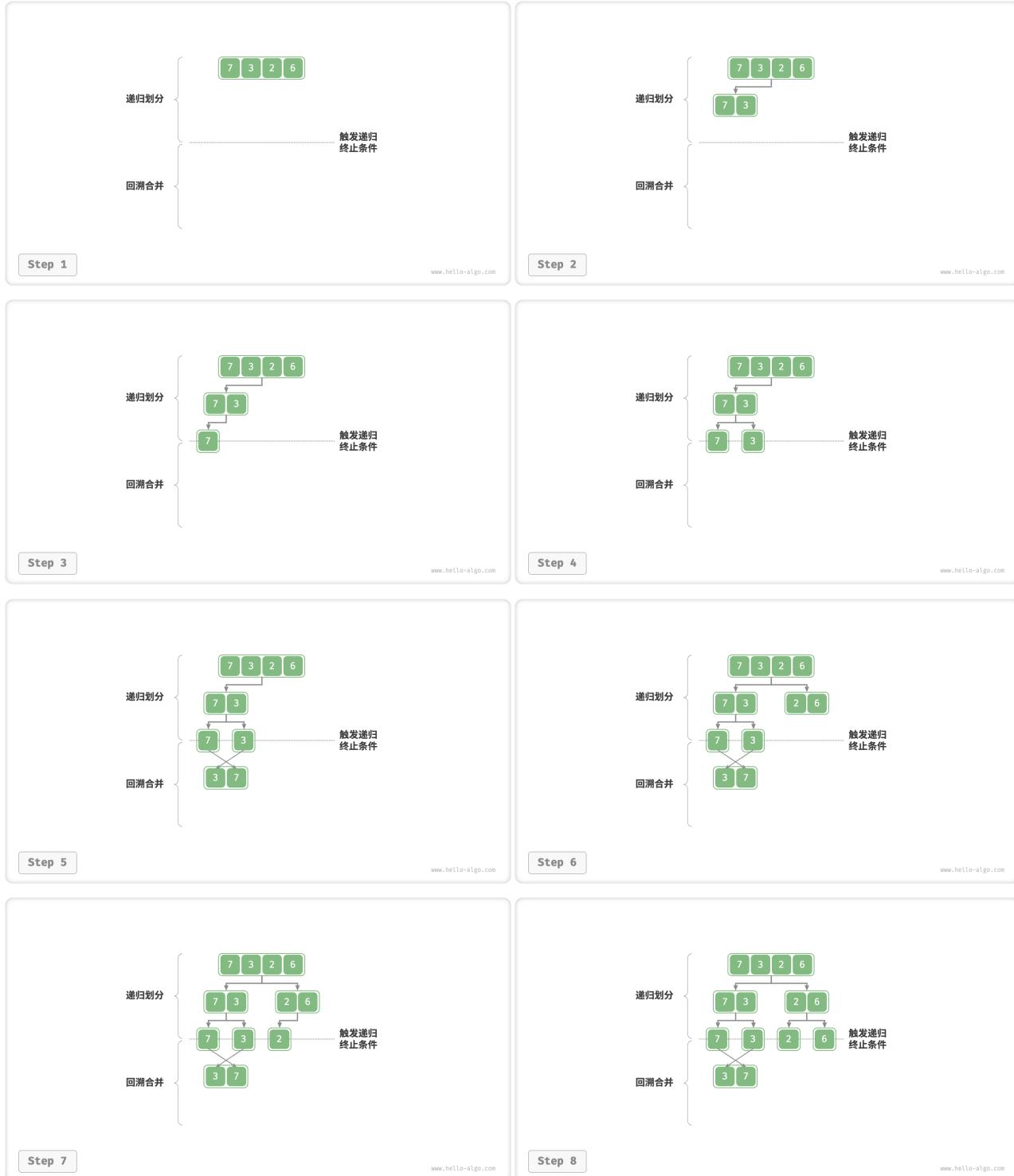




Figure 11-9. 归并排序步骤

观察发现，归并排序的递归顺序与二叉树的后序遍历相同，具体来看：

- **后序遍历**：先递归左子树，再递归右子树，最后处理根节点。
- **归并排序**：先递归左子数组，再递归右子数组，最后处理合并。

```
# === File: merge_sort.py ===
def merge(nums: list[int], left: int, mid: int, right: int) -> None:
    """ 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]
    # 初始化辅助数组
    tmp: list[int] = list(nums[left : right + 1])
    # 左子数组的起始索引和结束索引
    left_start: int = 0
    left_end: int = mid - left
    # 右子数组的起始索引和结束索引
    right_start: int = mid + 1 - left
    right_end: int = right - left
    # i, j 分别指向左子数组、右子数组的首元素
    i: int = left_start
    j: int = right_start
    # 通过覆盖原数组 nums 来合并左子数组和右子数组
    for k in range(left, right + 1):
        # 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 > left_end:
            nums[k] = tmp[j]
            j += 1
        # 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++
        elif j > right_end or tmp[i] <= tmp[j]:
            nums[k] = tmp[i]
            i += 1
        # 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]
        j += 1

def merge_sort(nums: list[int], left: int, right: int) -> None:
    """ 归并排序 """
    # 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
    mid: int = (left + right) // 2 # 计算中点
    merge_sort(nums, left, mid) # 递归左子数组
    merge_sort(nums, mid + 1, right) # 递归右子数组
    # 合并阶段
    merge(nums, left, mid, right)
```

合并方法 `merge()` 代码中的难点包括：

- 在阅读代码时，需要特别注意各个变量的含义。`nums` 的待合并区间为 `[left, right]`，但由于 `tmp` 仅复制了 `nums` 该区间的元素，因此 `tmp` 对应区间为 `[0, right - left]`。
- 在比较 `tmp[i]` 和 `tmp[j]` 的大小时，还需考虑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，如果左子数组已经被合并完了，那么不需要继续比较，直接合并右子数组元素即可。

### 11.5.2. 算法特性

**时间复杂度  $O(n \log n)$** ：划分产生高度为  $\log n$  的递归树，每层合并的总操作数量为  $n$ ，因此总体时间复杂度为  $O(n \log n)$ 。

**空间复杂度  $O(n)$** ：递归深度为  $\log n$ ，使用  $O(\log n)$  大小的栈帧空间；合并操作需要借助辅助数组实现，使用  $O(n)$  大小的额外空间；因此是“非原地排序”。

在合并过程中，相等元素的次序保持不变，因此归并排序是“稳定排序”。

### 11.5.3. 链表排序 \*

归并排序在排序链表时具有显著优势，空间复杂度可以优化至  $O(1)$ ，原因如下：

- 由于链表仅需改变指针就可实现节点的增删操作，因此合并阶段（将两个短有序链表合并为一个长有序链表）无需创建辅助链表。
- 通过使用“迭代划分”替代“递归划分”，可省去递归使用的栈帧空间；

具体实现细节比较复杂，有兴趣的同学可以查阅相关资料进行学习。

## 11.6. 桶排序

前述的几种排序算法都属于“基于比较的排序算法”，它们通过比较元素间的大小来实现排序。此类排序算法的时间复杂度无法超越  $O(n \log n)$ 。接下来，我们将探讨几种“非比较排序算法”，它们的时间复杂度可以达到线性水平。

「桶排序 Bucket Sort」是分治思想的一个典型应用。它通过设置一些具有大小顺序的桶，每个桶对应一个数据范围，将数据平均分配到各个桶中；然后，在每个桶内部分别执行排序；最终按照桶的顺序将所有数据合并。

### 11.6.1. 算法流程

考虑一个长度为  $n$  的数组，元素是范围  $[0, 1)$  的浮点数。桶排序的流程如下：

1. 初始化  $k$  个桶，将  $n$  个元素分配到  $k$  个桶中；
2. 对每个桶分别执行排序（本文采用编程语言的内置排序函数）；
3. 按照桶的从小到大的顺序，合并结果；

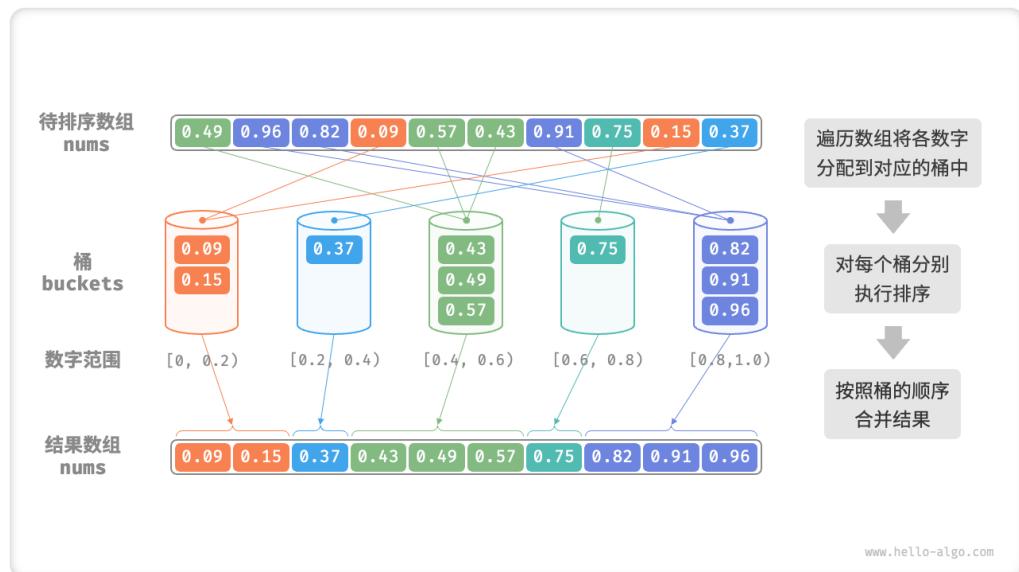


Figure 11-10. 桶排序算法流程

```
# == File: bucket_sort.py ==
def bucket_sort(nums: list[float]) -> None:
    """
    桶排序"""
    # 初始化 k = n/2 个桶，预期向每个桶分配 2 个元素
    k = len(nums) // 2
    buckets = [[] for _ in range(k)]
    # 1. 将数组元素分配到各个桶中
    for num in nums:
        # 输入数据范围 [0, 1)，使用 num * k 映射到索引范围 [0, k-1]
        i = int(num * k)
        buckets[i].append(num)
    # 2. 对每个桶分别执行排序
    for bucket in buckets:
        bucket.sort()
    # 3. 按照桶的顺序合并结果
    i = 0
    for bucket in buckets:
        for num in bucket:
            nums[i] = num
            i += 1
```

```
# 将 num 添加进桶 i
buckets[i].append(num)

# 2. 对各个桶执行排序 5
for bucket in buckets:
    # 使用内置排序函数，也可以替换成其他排序算法
    bucket.sort()

# 3. 遍历桶合并结果
i = 0
for bucket in buckets:
    for num in bucket:
        nums[i] = num
    i += 1
```



#### 桶排序的适用场景是什么？

桶排序适用于处理体量很大的数据。例如，输入数据包含 100 万个元素，由于空间限制，系统内存无法一次性加载所有数据。此时，可以将数据分成 1000 个桶，然后分别对每个桶进行排序，最后将结果合并。

#### 11.6.2. 算法特性

**时间复杂度  $O(n + k)$ ：**假设元素在各个桶内平均分布，那么每个桶内的元素数量为  $\frac{n}{k}$ 。假设排序单个桶使用  $O(\frac{n}{k} \log \frac{n}{k})$  时间，则排序所有桶使用  $O(n \log \frac{n}{k})$  时间。当桶数量  $k$  比较大时，时间复杂度则趋向于  $O(n)$ 。合并结果时需要遍历  $n$  个桶，花费  $O(k)$  时间。

在最坏情况下，所有数据被分配到一个桶中，且排序该桶使用  $O(n^2)$  时间，因此是“自适应排序”。

**空间复杂度  $O(n + k)$ ：**需要借助  $k$  个桶和总共  $n$  个元素的额外空间，属于“非原地排序”。

桶排序是否稳定取决于排序桶内元素的算法是否稳定。

#### 11.6.3. 如何实现平均分配

桶排序的时间复杂度理论上可以达到  $O(n)$ ，关键在于将元素均匀分配到各个桶中，因为实际数据往往不是均匀分布的。例如，我们想要将淘宝上的所有商品按价格范围平均分配到 10 个桶中，但商品价格分布不均，低于 100 元的非常多，高于 1000 元的非常少。若将价格区间平均划分为 10 份，各个桶中的商品数量差距会非常大。

为实现平均分配，我们可以先设定一个大致的分界线，将数据粗略地分到 3 个桶中。分配完毕后，再将商品较多的桶继续划分为 3 个桶，直至所有桶中的元素数量大致相等。这种方法本质上是创建一个递归树，使叶节点的值尽可能平均。当然，不一定要每轮将数据划分为 3 个桶，具体划分方式可根据数据特点灵活选择。

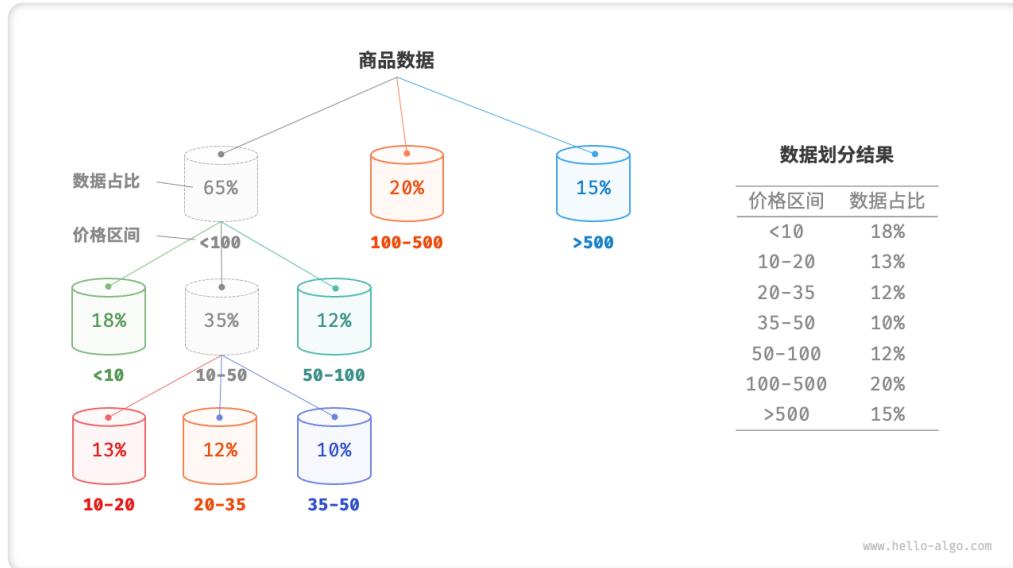


Figure 11-11. 递归划分桶

如果我们提前知道商品价格的概率分布，则可以根据数据概率分布设置每个桶的价格分界线。值得注意的是，数据分布并不一定需要特意统计，也可以根据数据特点采用某种概率模型进行近似。如下图所示，我们假设商品价格服从正态分布，这样就可以合理地设定价格区间，从而将商品平均分配到各个桶中。

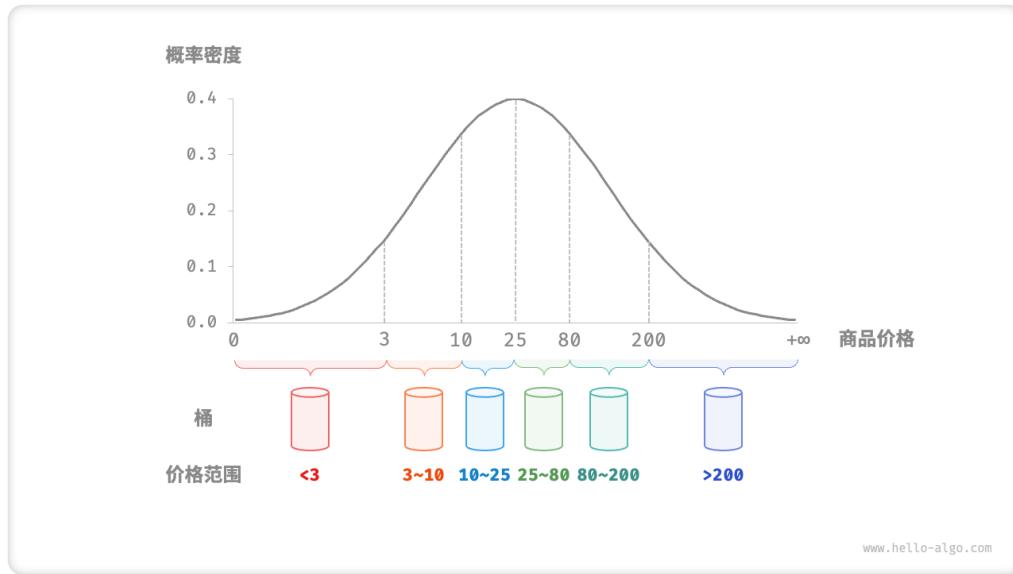


Figure 11-12. 根据概率分布划分桶

## 11.7. 计数排序

「计数排序 Counting Sort」通过统计元素数量来实现排序，通常应用于整数数组。

### 11.7.1. 简单实现

先来看一个简单的例子。给定一个长度为  $n$  的数组 `nums`，其中的元素都是“非负整数”。计数排序的整体流程如下：

1. 遍历数组，找出数组中的最大数字，记为  $m$ ，然后创建一个长度为  $m + 1$  的辅助数组 `counter`；
2. 借助 `counter` 统计 `nums` 中各数字的出现次数，其中 `counter[num]` 对应数字 `num` 的出现次数。统计方法很简单，只需遍历 `nums`（设当前数字为 `num`），每轮将 `counter[num]` 增加 1 即可。
3. 由于 `counter` 的各个索引天然有序，因此相当于所有数字已经被排序好了。接下来，我们遍历 `counter`，根据各数字的出现次数，将它们按从小到大的顺序填入 `nums` 即可。

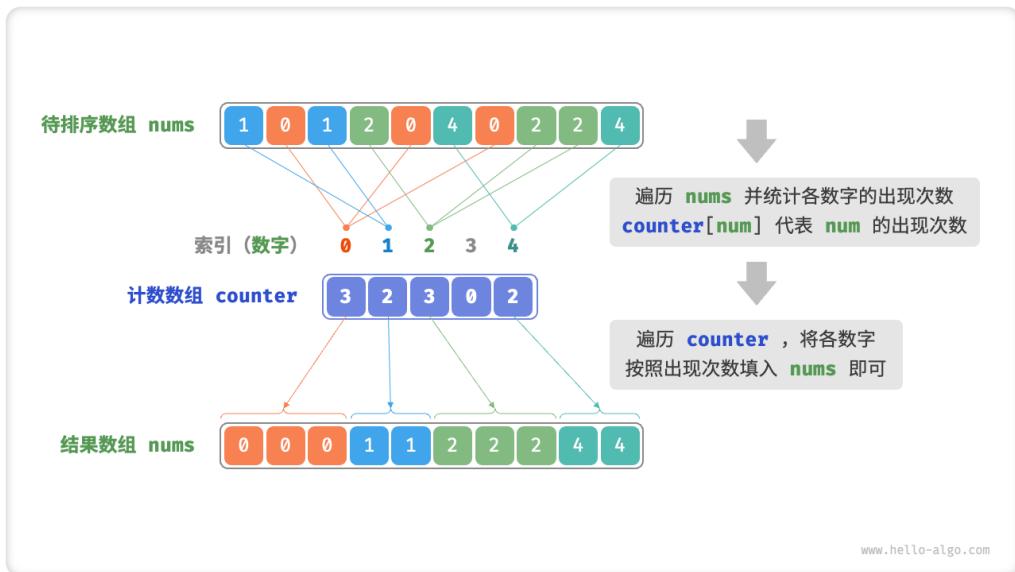


Figure 11-13. 计数排序流程

```
# === File: counting_sort.py ===
def counting_sort_naive(nums: list[int]) -> None:
    """ 计数排序 """
    # 简单实现，无法用于排序对象
    # 1. 统计数组最大元素 m
    m = 0
    for num in nums:
        m = max(m, num)
    # 2. 统计各数字的出现次数
    # counter[num] 代表 num 的出现次数
    counter = [0] * (m + 1)
    for num in nums:
        counter[num] += 1
    # 3. 遍历 counter，将各元素填入原数组 nums
    i = 0
    for num in range(m + 1):
        for _ in range(counter[num]):
```

```
nums[i] = num
i += 1
```



### 计数排序与桶排序的联系

从桶排序的角度看，我们可以将计数排序中的计数数组 `counter` 的每个索引视为一个桶，将统计数量的过程看作是将各个元素分配到对应的桶中。本质上，计数排序是桶排序在整型数据下的一个特例。

## 11.7.2. 完整实现

细心的同学可能发现，如果输入数据是对象，上述步骤 3. 就失效了。例如，输入数据是商品对象，我们想要按照商品价格（类的成员变量）对商品进行排序，而上述算法只能给出价格的排序结果。

那么如何才能得到原数据的排序结果呢？我们首先计算 `counter` 的「前缀和」。顾名思义，索引 `i` 处的前缀和 `prefix[i]` 等于数组前 `i` 个元素之和，即

$$\text{prefix}[i] = \sum_{j=0}^i \text{counter}[j]$$

前缀和具有明确的意义，`prefix[num] - 1` 代表元素 `num` 在结果数组 `res` 中最后一次出现的索引。这个信息非常关键，因为它告诉我们各个元素应该出现在结果数组的哪个位置。接下来，我们倒序遍历原数组 `nums` 的每个元素 `num`，在每轮迭代中执行：

1. 将 `num` 填入数组 `res` 的索引 `prefix[num] - 1` 处；
2. 令前缀和 `prefix[num]` 减小 1，从而得到下次放置 `num` 的索引；

遍历完成后，数组 `res` 中就是排序好的结果，最后使用 `res` 覆盖原数组 `nums` 即可。



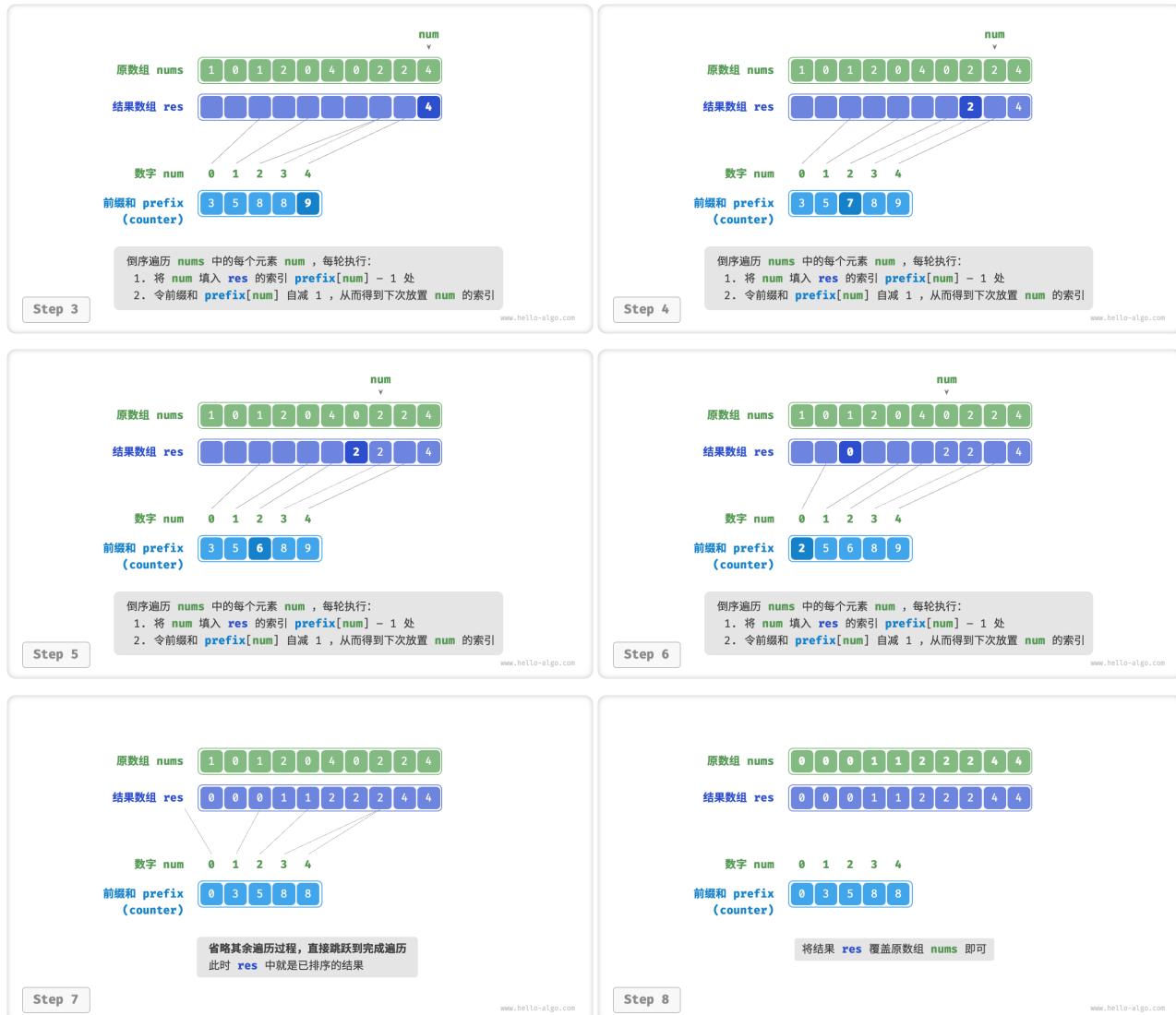


Figure 11-14. 计数排序步骤

计数排序的实现代码如下所示。

```
# == File: counting_sort.py ==
def counting_sort(nums: list[int]) -> None:
    """ 计数排序 """
    # 完整实现, 可排序对象, 并且是稳定排序
    # 1. 统计数组最大元素 m
    m = max(nums)
    # 2. 统计各数字的出现次数
    # counter[num] 代表 num 的出现次数
    counter = [0] * (m + 1)
    for num in nums:
        counter[num] += 1
```

```
# 3. 求 counter 的前缀和，将“出现次数”转换为“尾索引”
# 即 counter[num]-1 是 num 在 res 中最后一次出现的索引
for i in range(m):
    counter[i + 1] += counter[i]
# 4. 倒序遍历 nums，将各元素填入结果数组 res
# 初始化数组 res 用于记录结果
n = len(nums)
res = [0] * n
for i in range(n - 1, -1, -1):
    num = nums[i]
    res[counter[num] - 1] = num # 将 num 放置到对应索引处
    counter[num] -= 1 # 令前缀和自减 1，得到下次放置 num 的索引
# 使用结果数组 res 覆盖原数组 nums
for i in range(n):
    nums[i] = res[i]
```

### 11.7.3. 算法特性

**时间复杂度  $O(n + m)$** ：涉及遍历 `nums` 和遍历 `counter`，都使用线性时间。一般情况下  $n \gg m$ ，时间复杂度趋于  $O(n)$ 。

**空间复杂度  $O(n + m)$** ：借助了长度分别为  $n$  和  $m$  的数组 `res` 和 `counter`，因此是“非原地排序”。

**稳定排序**：由于向 `res` 中填充元素的顺序是“从右向左”的，因此倒序遍历 `nums` 可以避免改变相等元素之间的相对位置，从而实现“稳定排序”。实际上，正序遍历 `nums` 也可以得到正确的排序结果，但结果是“非稳定”的。

### 11.7.4. 局限性

看到这里，你也许会觉得计数排序非常巧妙，仅通过统计数量就可以实现高效的排序工作。然而，使用计数排序的前置条件相对较为严格。

**计数排序只适用于非负整数**。若想要将其用于其他类型的数据，需要确保这些数据可以被转换为非负整数，并且在转换过程中不能改变各个元素之间的相对大小关系。例如，对于包含负数的整数数组，可以先给所有数字加上一个常数，将全部数字转化为正数，排序完成后再转换回去即可。

**计数排序适用于数据量大但数据范围较小的情况**。比如，在上述示例中  $m$  不能太大，否则会占用过多空间。而当  $n \ll m$  时，计数排序使用  $O(m)$  时间，可能比  $O(n \log n)$  的排序算法还要慢。

## 11.8. 基数排序

上一节我们介绍了计数排序，它适用于数据量  $n$  较大但数据范围  $m$  较小的情况。假设我们需要对  $n = 10^6$  个学号进行排序，而学号是一个 8 位数字，这意味着数据范围  $m = 10^8$  非常大，使用计数排序需要分配大量内存空间，而基数排序可以避免这种情况。

「基数排序 Radix Sort」的核心思想与计数排序一致，也通过统计个数来实现排序。在此基础上，基数排序利用数字各位之间的递进关系，依次对每一位进行排序，从而得到最终的排序结果。

### 11.8.1. 算法流程

以学号数据为例，假设数字的最低位是第 1 位，最高位是第 8 位，基数排序的步骤如下：

1. 初始化位数  $k = 1$ ；
2. 对学号的第  $k$  位执行「计数排序」。完成后，数据会根据第  $k$  位从小到大排序；
3. 将  $k$  增加 1，然后返回步骤 2. 继续迭代，直到所有位都排序完成后结束；

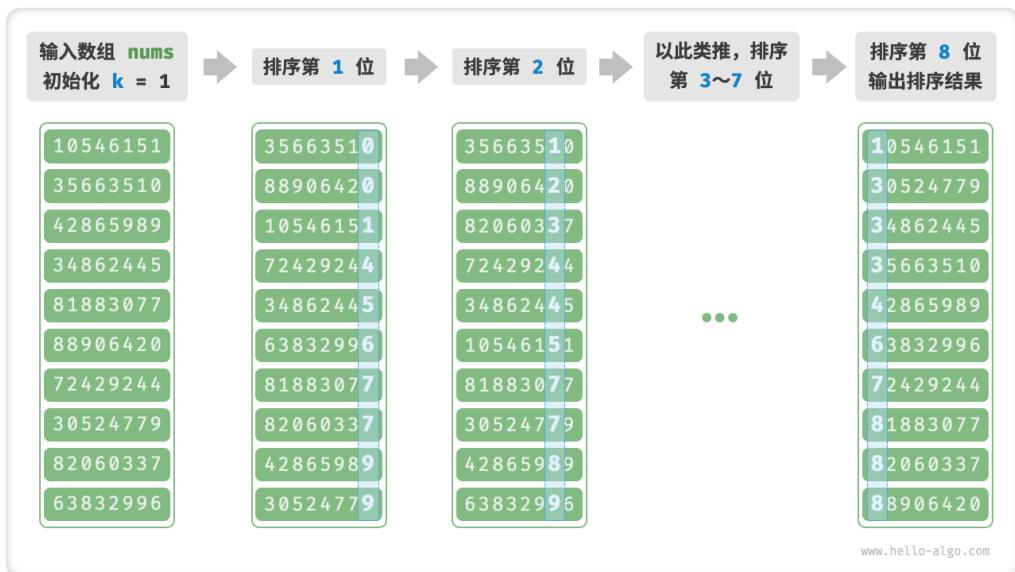


Figure 11-15. 基数排序算法流程

下面来剖析代码实现。对于一个  $d$  进制的数字  $x$ ，要获取其第  $k$  位  $x_k$ ，可以使用以下计算公式：

$$x_k = \lfloor \frac{x}{d^{k-1}} \rfloor \mod d$$

其中  $\lfloor a \rfloor$  表示对浮点数  $a$  向下取整，而  $\mod d$  表示对  $d$  取余。对于学号数据， $d = 10$  且  $k \in [1, 8]$ 。

此外，我们需要小幅改动计数排序代码，使之可以根据数字的第  $k$  位进行排序。

```
# == File: radix_sort.py ==
def digit(num: int, exp: int) -> int:
    """ 获取元素 num 的第 k 位，其中 exp = 10^(k-1) """
    # 传入 exp 而非 k 可以避免在此重复执行昂贵的次方计算
    return (num // exp) % 10

def counting_sort_digit(nums: list[int], exp: int) -> None:
```

```
""" 计数排序（根据 nums 第 k 位排序）"""
# 十进制的位范围为 0~9，因此需要长度为 10 的桶
counter = [0] * 10
n = len(nums)
# 统计 0~9 各数字的出现次数
for i in range(n):
    d = digit(nums[i], exp) # 获取 nums[i] 第 k 位，记为 d
    counter[d] += 1 # 统计数字 d 的出现次数
# 求前缀和，将“出现个数”转换为“数组索引”
for i in range(1, 10):
    counter[i] += counter[i - 1]
# 倒序遍历，根据桶内统计结果，将各元素填入 res
res = [0] * n
for i in range(n - 1, -1, -1):
    d = digit(nums[i], exp)
    j = counter[d] - 1 # 获得 d 在数组中的索引 j
    res[j] = nums[i] # 将当前元素填入索引 j
    counter[d] -= 1 # 将 d 的数量减 1
# 使用结果覆盖原数组 nums
for i in range(n):
    nums[i] = res[i]

def radix_sort(nums: list[int]) -> None:
    """ 基数排序 """
    # 获取数组的最大元素，用于判断最大位数
    m = max(nums)
    # 按照从低位到高位的顺序遍历
    exp = 1
    while exp <= m:
        # 对数组元素的第 k 位执行计数排序
        # k = 1 -> exp = 1
        # k = 2 -> exp = 10
        # 即 exp = 10^(k-1)
        counting_sort_digit(nums, exp)
        exp *= 10
```



### 为什么从最低位开始排序？

在连续的排序轮次中，后一轮排序会覆盖前一轮排序的结果。举例来说，如果第一轮排序结果  $a < b$ ，而第二轮排序结果  $a > b$ ，那么第二轮的结果将取代第一轮的结果。由于数字的高位优先级高于低位，我们应该先排序低位再排序高位。

### 11.8.2. 算法特性

**时间复杂度  $O(nk)$** ：设数据量为  $n$ 、数据为  $d$  进制、最大位数为  $k$ ，则对某一位执行计数排序使用  $O(n + d)$  时间，排序所有  $k$  位使用  $O((n + d)k)$  时间。通常情况下， $d$  和  $k$  都相对较小，时间复杂度趋向  $O(n)$ 。

**空间复杂度  $O(n + d)$** ：与计数排序相同，基数排序需要借助长度为  $n$  和  $d$  的数组 `res` 和 `counter`，因此它是一种“非原地排序”。

基数排序与计数排序一样，都属于稳定排序。相较于计数排序，基数排序适用于数值范围较大的情况，但前提是数据必须可以表示为固定位数的格式，且位数不能过大。例如，浮点数不适合使用基数排序，因为其位数  $k$  过大，可能导致时间复杂度  $O(nk) \gg O(n^2)$ 。

## 11.9. 小结

- 冒泡排序通过交换相邻元素来实现排序。通过添加一个标志位来实现提前返回，我们可以将冒泡排序的最佳时间复杂度优化到  $O(n)$ 。
- 插入排序每轮将待排序区间内的元素插入到已排序区间的正确位置，从而完成排序。虽然插入排序的时间复杂度为  $O(n^2)$ ，但由于单元操作相对较少，它在小数据量的排序任务中非常受欢迎。
- 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，导致时间复杂度劣化至  $O(n^2)$ 。引入中位数基准数或随机基准数可以降低这种劣化的概率。尾递归方法可以有效地减少递归深度，将空间复杂度优化到  $O(\log n)$ 。
- 归并排序包括划分和合并两个阶段，典型地体现了分治策略。在归并排序中，排序数组需要创建辅助数组，空间复杂度为  $O(n)$ ；然而排序链表的空间复杂度可以优化至  $O(1)$ 。
- 桶排序包含三个步骤：数据分桶、桶内排序和合并结果。它同样体现了分治策略，适用于数据体量很大的情况。桶排序的关键在于对数据进行平均分配。
- 计数排序是桶排序的一个特例，它通过统计数据出现的次数来实现排序。计数排序适用于数据量大但数据范围有限的情况，并且要求数据能够转换为正整数。
- 基数排序通过逐位排序来实现数据排序，要求数据能够表示为固定位数的数字。

	时间复杂度		空间复杂度		稳定性	就地性	自适应性	基于比较	
	最佳	平均	最差	最差					
遍历排序 $O(n^2)$	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
分治排序 $O(n \log n)$	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
线性排序 $O(n)$	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
线性排序 $O(n)$	桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	稳定	非原地	自适应	非比较
	计数排序	$O(n + m)$	$O(n + m)$	$O(n + m)$	$O(n + m)$	稳定	非原地	非自适应	非比较
线性排序 $O(n)$	基数排序	$O(n k)$	$O(n k)$	$O(n k)$	$O(n + b)$	稳定	非原地	非自适应	非比较

差	中	优
---	---	---

$n$  为数据量大小  
桶排序中， $k$  为桶数量  
计数排序中， $m$  为数据范围  
基数排序中， $k$  为最大位数，数据为  $b$  进制

www.hello-algo.com

Figure 11-16. 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，如同其他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。
- 总的来说，我们希望找到一种排序算法，具有高效率、稳定、原地以及正向自适应性等优点。然而，正如其他数据结构和算法一样，没有一种排序算法能够同时满足所有这些条件。在实际应用中，我们需要根据数据的特性来选择合适的排序算法。

# 12. 搜索算法

## 12.1. 搜索算法

「搜索算法 Searching Algorithm」用于在数据结构（例如数组、链表、树或图）中搜索一个或一组满足特定条件的元素。

我们已经学过数组、链表、树和图的遍历方法，也学过哈希表、二叉搜索树等可用于实现查询的复杂数据结构。因此，搜索算法对于我们来说并不陌生。在本节，我们将从更加系统的视角切入，重新审视搜索算法。

### 12.1.1. 暴力搜索

暴力搜索通过遍历数据结构的每个元素来定位目标元素。

- 「线性搜索」适用于数组和链表等线性数据结构。它从数据结构的一端开始，逐个访问元素，直到找到目标元素或到达另一端仍没有找到目标元素为止。
- 「广度优先搜索」和「深度优先搜索」是图和树的两种遍历策略。广度优先搜索从初始节点开始逐层搜索，由近及远地访问各个节点。深度优先搜索是从初始节点开始，沿着一条路径走到头为止，再回溯并尝试其他路径，直到遍历完整个数据结构。

暴力搜索的优点是简单且通用性好，**无需对数据做预处理和借助额外的数据结构**。

然而，**此类算法的时间复杂度为  $O(n)$** ，其中  $n$  为元素数量，因此在数据量较大的情况下性能较差。

### 12.1.2. 自适应搜索

自适应搜索利用数据的特有属性（例如有序性）来优化搜索过程，从而更高效地定位目标元素。

- 「二分查找」利用数据的有序性实现高效查找，仅适用于数组。
- 「哈希查找」利用哈希表将搜索数据和目标数据建立为键值对映射，从而实现查询操作。
- 「树查找」在特定的树结构（例如二叉搜索树）中，基于比较节点值来快速排除节点，从而定位目标元素。

此类算法的优点是效率高，**时间复杂度可达到  $O(\log n)$  甚至  $O(1)$** 。

然而，**使用这些算法往往需要对数据进行预处理**。例如，二分查找需要预先对数组进行排序，哈希查找和树查找都需要借助额外的数据结构，维护这些数据结构也需要额外的时间和空间开支。



自适应搜索算法常被称为查找算法，主要关注在特定数据结构中快速检索目标元素。

### 12.1.3. 搜索方法选取

给定大小为  $n$  的一组数据，我们可以使用线性搜索、二分查找、树查找、哈希查找等多种方法在该数据中搜索目标元素。各个方法的工作原理如下图所示。

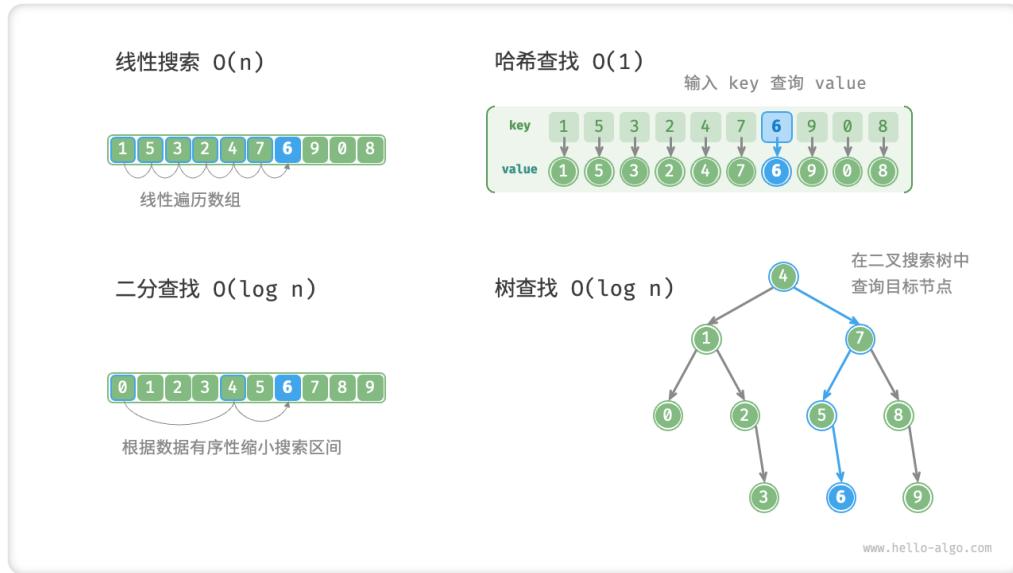


Figure 12-1. 多种搜索策略

上述几种方法的操作效率与特性如下表所示。

	线性搜索	二分查找	树查找	哈希查找
查找元素	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
插入元素	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
删除元素	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
额外空间	$O(1)$	$O(1)$	$O(n)$	$O(n)$
数据预处理	/	排序 $O(n \log n)$	建树 $O(n \log n)$	建哈希表 $O(n)$
数据是否有序	无序	有序	有序	无序

除了以上表格内容，搜索算法的选择还取决于数据体量、搜索性能要求、数据查询与更新频率等。

#### 线性搜索

- 通用性较好，无需任何数据预处理操作。假如我们仅需查询一次数据，那么其他三种方法的数据预处理的时间比线性搜索的时间还要更长。
- 适用于体量较小的数据，此情况下时间复杂度对效率影响较小。
- 适用于数据更新频率较高的场景，因为该方法不需要对数据进行任何额外维护。

#### 二分查找

- 适用于大数据量的情况，效率表现稳定，最差时间复杂度为  $O(\log n)$ 。
- 数据量不能过大，因为存储数组需要连续的内存空间。
- 不适用于高频增删数据的场景，因为维护有序数组的开销较大。

### 哈希查找

- 适合对查询性能要求很高的场景，平均时间复杂度为  $O(1)$ 。
- 不适合需要有序数据或范围查找的场景，因为哈希表无法维护数据的有序性。
- 对哈希函数和哈希冲突处理策略的依赖性较高，具有较大的性能劣化风险。
- 不适合数据量过大的情况，因为哈希表需要额外空间来最大程度地减少冲突，从而提供良好的查询性能。

### 树查找

- 适用于海量数据，因为树节点在内存中是离散存储的。
- 适合需要维护有序数据或范围查找的场景。
- 在持续增删节点的过程中，二叉搜索树可能产生倾斜，时间复杂度劣化至  $O(n)$ 。
- 若使用 AVL 树或红黑树，则各项操作可在  $O(\log n)$  效率下稳定运行，但维护树平衡的操作会增加额外开销。

## 12.2. 哈希优化策略

在算法题中，我们常通过将线性查找替换为哈希查找来降低算法的时间复杂度。我们借助一个算法题来加深理解。



### 两数之和

给定一个整数数组 `nums` 和一个整数目标值 `target`，请在数组中搜索“和”为目标值 `target` 的两个整数，并返回他们在数组中的索引。注意，数组中同一个元素在答案里不能重复出现。返回任意一个解即可。

### 12.2.1. 线性查找：以时间换空间

考虑直接遍历所有可能的组合。开启一个两层循环，在每轮中判断两个整数的和是否为 `target`，若是，则返回它们的索引。

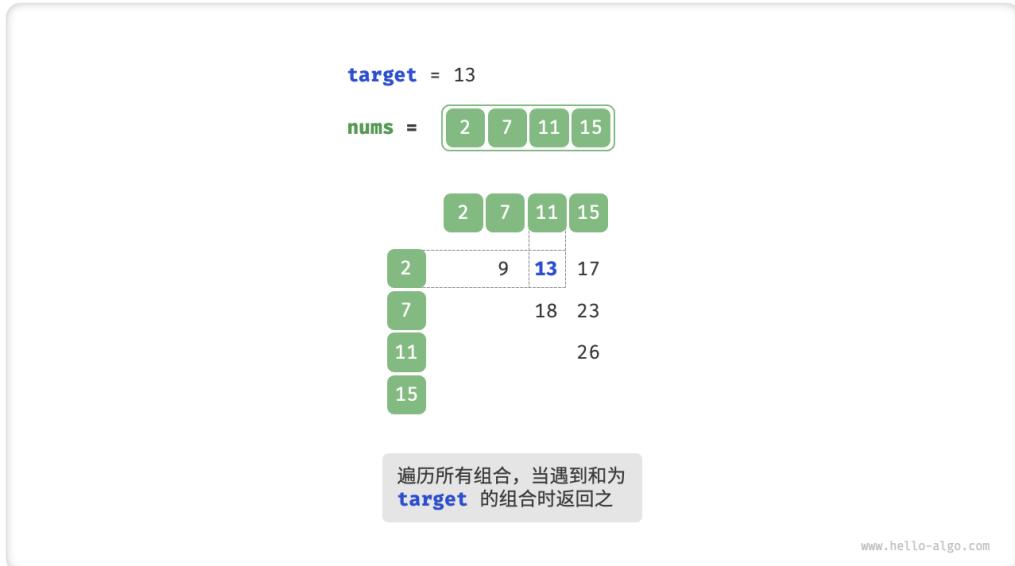


Figure 12-2. 线性查找求解两数之和

```
# == File: two_sum.py ==
def two_sum_brute_force(nums: list[int], target: int) -> list[int]:
    """ 方法一：暴力枚举 """
    # 两层循环，时间复杂度 O(n^2)
    for i in range(len(nums) - 1):
        for j in range(i + 1, len(nums)):
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
```

此方法的时间复杂度为  $O(n^2)$ ，空间复杂度为  $O(1)$ ，在大数据量下非常耗时。

### 12.2.2. 哈希查找：以空间换时间

考虑借助一个哈希表，键值对分别为数组元素和元素索引。循环遍历数组，每轮执行：

1. 判断数字 `target - nums[i]` 是否在哈希表中，若是则直接返回这两个元素的索引；
2. 将键值对 `num[i]` 和索引 `i` 添加进哈希表；

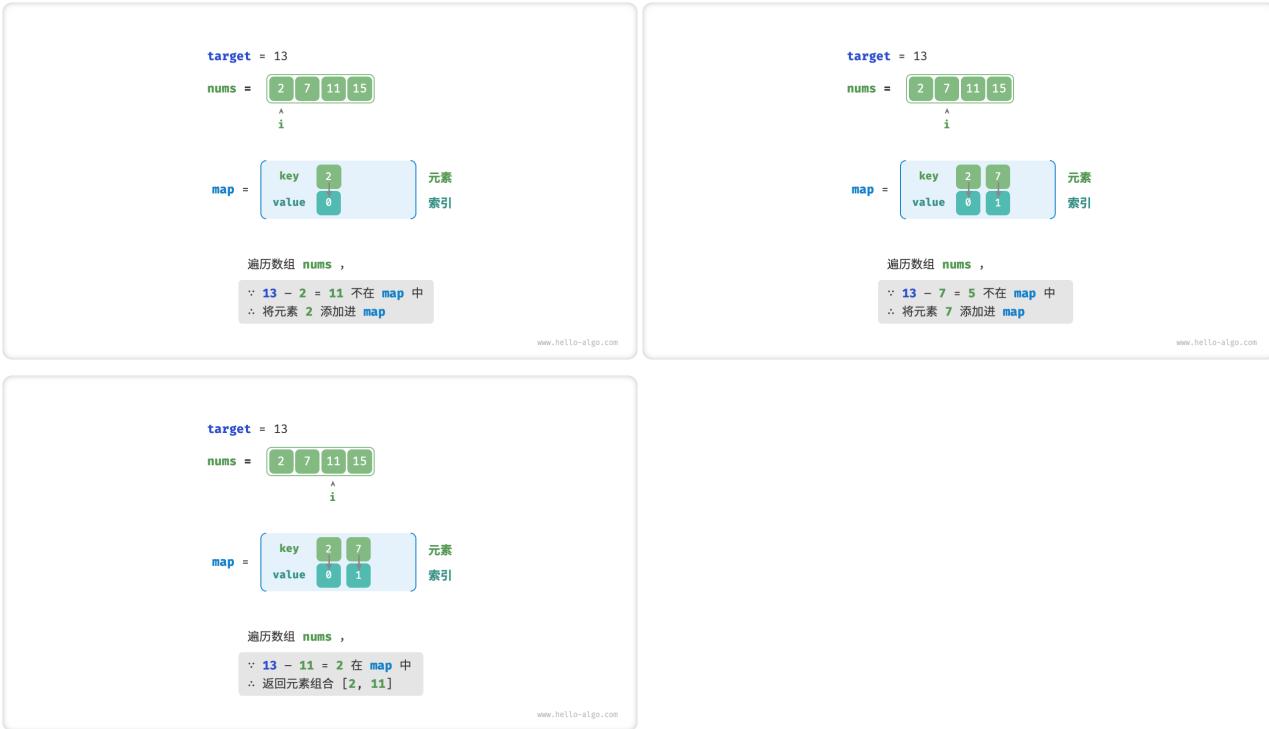


Figure 12-3. 辅助哈希表求解两数之和

实现代码如下所示，仅需单层循环即可。

```
# == File: two_sum.py ==
def two_sum_hash_table(nums: list[int], target: int) -> list[int]:
    """
    方法二：辅助哈希表"""
    # 辅助哈希表，空间复杂度 O(n)
    dic = {}
    # 单层循环，时间复杂度 O(n)
    for i in range(len(nums)):
        if target - nums[i] in dic:
            return [dic[target - nums[i]], i]
        dic[nums[i]] = i
    return []
```

此方法通过哈希查找将时间复杂度从  $O(n^2)$  降低至  $O(n)$ ，大幅提升运行效率。

由于需要维护一个额外的哈希表，因此空间复杂度为  $O(n)$ 。尽管如此，该方法的整体时空效率更为均衡，因此它是本题的最优解法。

### 12.3. 小结

- 二分查找依赖于数据的有序性，通过循环逐步缩减一半搜索区间来实现查找。它要求输入数据有序，且仅适用于数组或基于数组实现的数据结构。

- 暴力搜索通过遍历数据结构来定位数据。线性搜索适用于数组和链表，广度优先搜索和深度优先搜索适用于图和树。此类算法通用性好，无需对数据预处理，但时间复杂度  $O(n)$  较高。
- 哈希查找、树查找和二分查找属于高效搜索方法，可在特定数据结构中快速定位目标元素。此类算法效率高，时间复杂度可达  $O(\log n)$  甚至  $O(1)$ ，但通常需要借助额外数据结构。
- 实际中，我们需要对数据体量、搜索性能要求、数据查询和更新频率等因素进行具体分析，从而选择合适的搜索方法。
- 线性搜索适用于小型或频繁更新的数据；二分查找适用于大型、排序的数据；哈希查找适合对查询效率要求较高且无需范围查询的数据；树查找适用于需要维护顺序和支持范围查询的大型动态数据。
- 用哈希查找替换线性查找是一种常用的优化运行时间的策略，可将时间复杂度从  $O(n)$  降低至  $O(1)$ 。

# 13. 回溯算法

## 13.1. 回溯算法

「回溯算法 Backtracking Algorithm」是一种通过穷举来解决问题的方法，它的核心思想是从一个初始状态出发，暴力搜索所有可能的解决方案，当遇到正确的解则将其记录，直到找到解或者尝试了所有可能的选择都无法找到解为止。

回溯算法通常采用「深度优先搜索」来遍历解空间。在二叉树章节中，我们提到前序、中序和后序遍历都属于深度优先搜索。下面，我们从二叉树的前序遍历入手，逐步了解回溯算法的工作原理。



### 例题一：在二叉树中搜索并返回所有值为 7 的节点

**解题思路：**前序遍历这颗树，并判断当前节点的值是否为 7，若是则将该节点的值加入到结果列表 `res` 之中。

```
# === File: preorder_traversal_i_compact.py ===
def pre_order(root: TreeNode) -> None:
    """ 前序遍历：例题一 """
    if root is None:
        return
    if root.val == 7:
        # 记录解
        res.append(root)
    pre_order(root.left)
    pre_order(root.right)
```

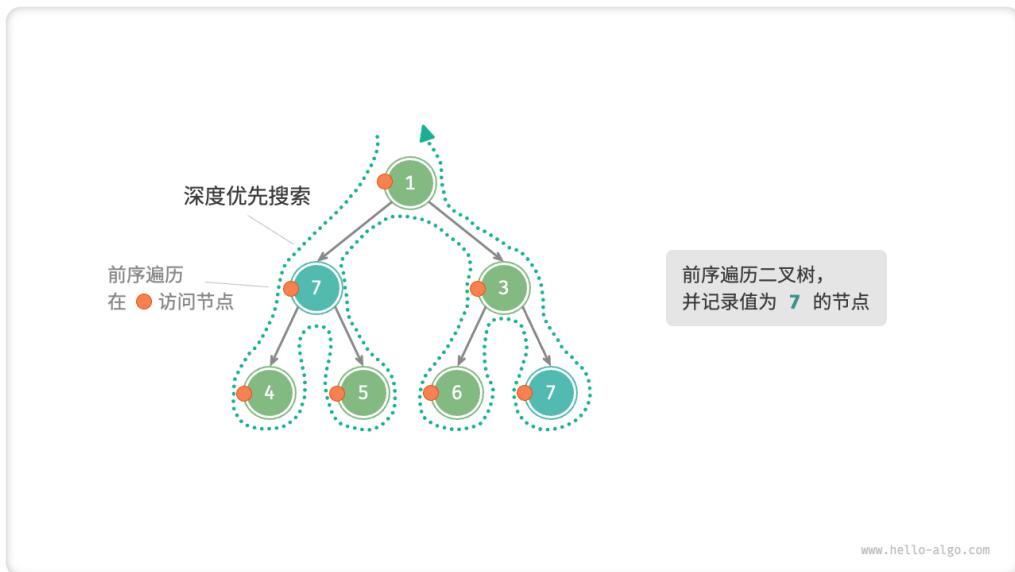


Figure 13-1. 在前序遍历中搜索节点

### 13.1.1. 尝试与回退

之所以称之为回溯算法，是因为该算法在搜索解空间时会采用“尝试”与“回退”的策略。当算法在搜索过程中遇到某个状态无法继续前进或无法得到满足条件的解时，它会撤销上一步的选择，退回到之前的状态，并尝试其他可能的选择。

对于例题一，访问每个节点都代表一次“尝试”，而越过叶结点或返回父节点的 `return` 则表示“回退”。

值得说明的是，回退并不等价于函数返回。为解释这一点，我们对例题一稍作拓展。

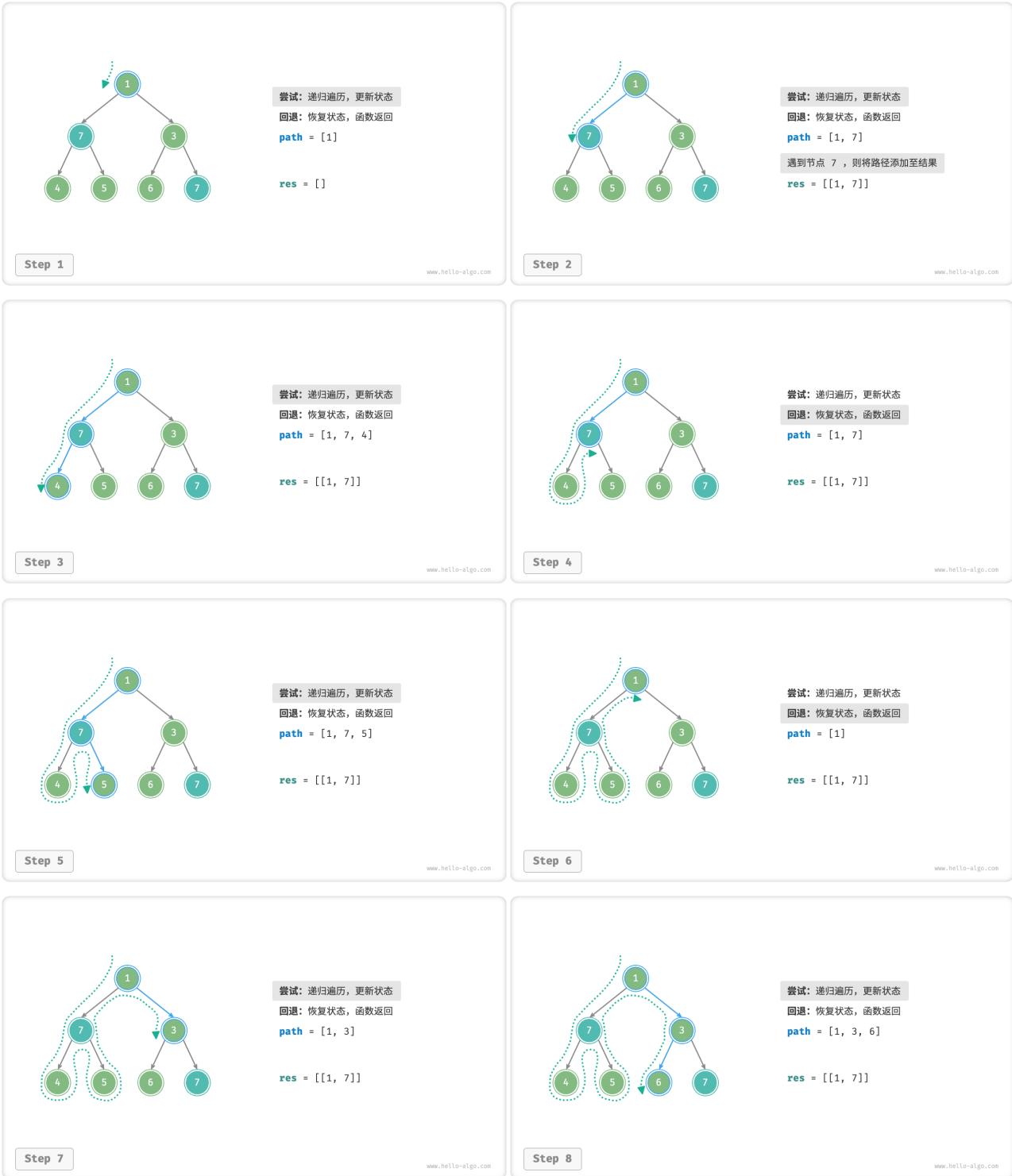


在二叉树中搜索所有值为 7 的节点，返回根节点到这些节点的路径 \*\*\*\*

**解题思路：**在例题一代码的基础上，我们需要借助一个列表 `path` 记录访问过的节点路径。当访问到值为 7 的节点时，则复制 `path` 并添加进结果列表 `res`。遍历完成后，`res` 中保存的就是所有的解。

```
# === File: preorder_traversal_iii_compact.py ===
def pre_order(root: TreeNode) -> None:
    """ 前序遍历：例题二 """
    if root is None:
        return
    # 尝试
    path.append(root)
    if root.val == 7:
        # 记录解
        res.append(list(path))
    pre_order(root.left)
    pre_order(root.right)
    # 回退
    path.pop()
```

在每次“尝试”中，我们通过将当前节点添加进 `path` 来记录路径；而在“回退”前，我们需要将该节点从 `path` 中弹出，以恢复本次尝试之前的状态。换句话说，我们可以将尝试和回退理解为“前进”与“撤销”，两个操作是互为相反的。



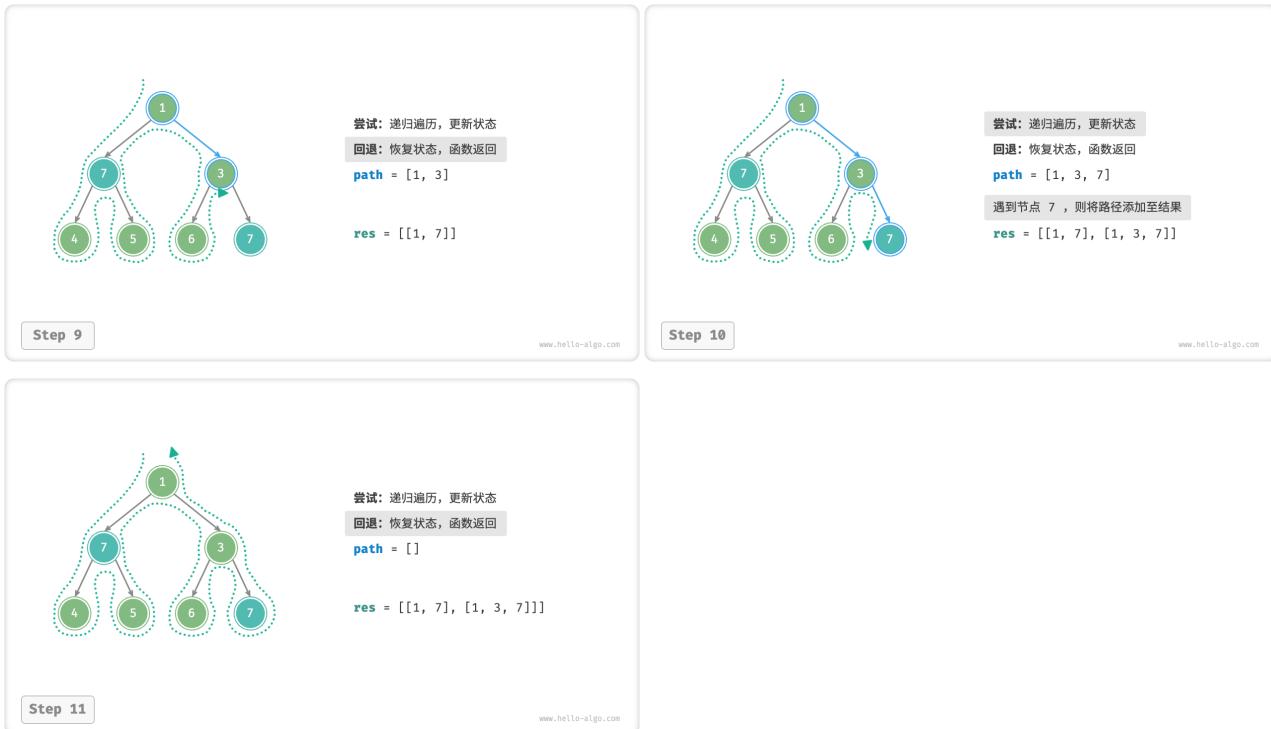


Figure 13-2. 尝试与回退

### 13.1.2. 剪枝

复杂的回溯问题通常包含一个或多个约束条件，**约束条件通常可用于“剪枝”**。



**例题三：**在二叉树中搜索所有值为 7 的节点，返回根节点到这些节点的路径，路径中不能包含值为 3 的节点 \*\*\*\*

**解题思路：**在例题二的基础上添加剪枝操作，当遇到值为 3 的节点时，则终止继续搜索。

```
# == File: preorder_traversal_iii_compact.py ==
def pre_order(root: TreeNode) -> None:
    """ 前序遍历：例题三 """
    # 剪枝
    if root is None or root.val == 3:
        return
    # 尝试
    path.append(root)
    if root.val == 7:
        # 记录解
        res.append(list(path))
    pre_order(root.left)
    pre_order(root.right)
    # 回退
    path.pop()
```

```
pre_order(root.left)
pre_order(root.right)
# 回退
path.pop()
```

剪枝是一个非常形象的名词。在搜索过程中，我们利用约束条件“剪掉”了不满足约束条件的搜索分支，避免许多无意义的尝试，从而提升搜索效率。

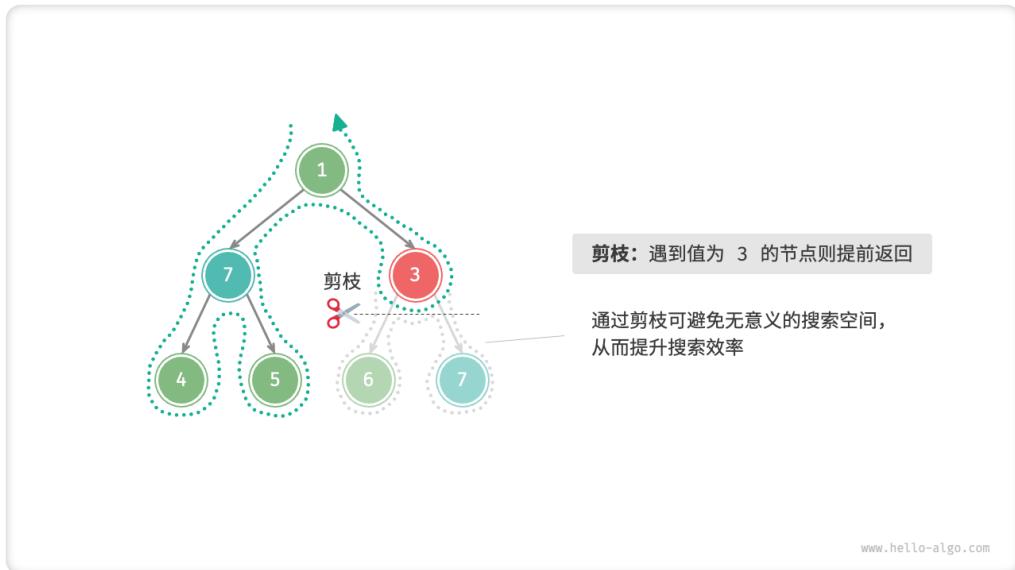


Figure 13-3. 根据约束条件剪枝

### 13.1.3. 常用术语

为了更清晰地分析算法问题，我们总结一下回溯算法中常用术语的含义，并对照例题三给出对应示例。

名词	定义	例题三
解 Solution	解是满足问题特定条件的答案。回溯算法的目标是找到一个或多个满足条件的解	根节点到节点 7 的所有路径，且路径中不包含值为 3 的节点
状态 State	状态表示问题在某一时刻的情况，包括已经做出的选择	当前已访问的节点路径，即 <code>path</code> 节点列表
约束条件 Constraint	约束条件是问题中限制解的可行性的条件，通常用于剪枝	要求路径中不能包含值为 3 的节点
尝试 Attempt	尝试是在搜索过程中，根据当前状态和可用选择来探索解空间的过程。尝试包括做出选择，更新状态，检查是否为解	递归访问左（右）子节点，将节点添加进 <code>path</code> ，判断节点的值是否为 7

名词	定义	例题三
回退 Backtracking	回退指在搜索中遇到到不满足约束条件或无法继续搜索的状态时，撤销前面做出的选择，回到上一个状态	当越过叶结点、结束结点访问、遇到值为 3 的节点时终止搜索，函数返回
剪枝 Pruning	剪枝是根据问题特性和约束条件避免无意义的搜索路径的方法，可提高搜索效率	当遇到值为 3 的节点时，则终止继续搜索



解、状态、约束条件等术语是通用的，适用于回溯算法、动态规划、贪心算法等。

#### 13.1.4. 框架代码

回溯算法可用于解决许多搜索问题、约束满足问题和组合优化问题。为提升代码通用性，我们希望将回溯算法的“尝试、回退、剪枝”的主体框架提炼出来。

设 `state` 为问题的当前状态，`choices` 表示当前状态下可以做出的选择，则可得到以下回溯算法的框架代码。

```
def backtrack(state: State, choices: list[choice], res: list[state]) -> None:
    """ 回溯算法框架 """
    # 判断是否为解
    if is_solution(state):
        # 记录解
        record_solution(state, res)
        return
    # 遍历所有选择
    for choice in choices:
        # 剪枝：判断选择是否合法
        if is_valid(state, choice):
            # 尝试：做出选择，更新状态
            make_choice(state, choice)
            backtrack(state, choices, res)
            # 回退：撤销选择，恢复到之前的状态
            undo_choice(state, choice)
```

下面，我们尝试基于此框架来解决例题三。在例题三中，状态 `state` 是节点遍历路径，选择 `choices` 是当前节点的左子节点和右子节点，结果 `res` 是路径列表，实现代码如下所示。

```
# === File: preorder_traversal_iii_template.py ===
def is_solution(state: list[TreeNode]) -> bool:
    """ 判断当前状态是否为解 """
    return state and state[-1].val == 7
```

```
def record_solution(state: list[TreeNode], res: list[list[TreeNode]]):
    """ 记录解 """
    res.append(list(state))

def is_valid(state: list[TreeNode], choice: TreeNode) -> bool:
    """ 判断在当前状态下, 该选择是否合法 """
    return choice is not None and choice.val != 3

def make_choice(state: list[TreeNode], choice: TreeNode):
    """ 更新状态 """
    state.append(choice)

def undo_choice(state: list[TreeNode], choice: TreeNode):
    """ 恢复状态 """
    state.pop()

def backtrack(state: list[TreeNode], choices: list[TreeNode], res: list[list[TreeNode]]):
    """ 回溯算法: 例题三 """
    # 检查是否为解
    if is_solution(state):
        # 记录解
        record_solution(state, res)
        return
    # 遍历所有选择
    for choice in choices:
        # 剪枝: 检查选择是否合法
        if is_valid(state, choice):
            # 尝试: 做出选择, 更新状态
            make_choice(state, choice)
            # 进行下一轮选择
            backtrack(state, [choice.left, choice.right], res)
            # 回退: 撤销选择, 恢复到之前的状态
            undo_choice(state, choice)
```

相较于基于前序遍历的实现代码，基于回溯算法框架的实现代码虽然显得啰嗦，但通用性更好。实际上，**所有回溯问题都可以在该框架下解决**。我们需要根据具体问题来定义 `state` 和 `choices`，并实现框架中的各个方法。

### 13.1.5. 典型例题

**搜索问题：**这类问题的目标是找到满足特定条件的解决方案。

- 全排列问题：给定一个集合，求出其所有可能的排列组合。
- 子集和问题：给定一个集合和一个目标和，找到集合中所有和为目标和的子集。

- 汉诺塔问题：给定三个柱子和一系列大小不同的圆盘，要求将所有圆盘从一个柱子移动到另一个柱子，每次只能移动一个圆盘，且不能将大圆盘放在小圆盘上。

**约束满足问题：**这类问题的目标是找到满足所有约束条件的解。

- $n$  皇后：在  $n \times n$  的棋盘上放置  $n$  个皇后，使得它们互不攻击。
- 数独：在  $9 \times 9$  的网格中填入数字  $1 \sim 9$ ，使得每行、每列和每个  $3 \times 3$  子网格中的数字不重复。
- 图着色问题：给定一个无向图，用最少的颜色给图的每个顶点着色，使得相邻顶点颜色不同。

**组合优化问题：**这类问题的目标是在一个组合空间中找到满足某些条件的最优解。

- 0-1 背包问题：给定一组物品和一个背包，每个物品有一定的价值和重量，要求在背包容量限制内，选择物品使得总价值最大。
- 旅行商问题：在一个图中，从一个点出发，访问所有其他点恰好一次后返回起点，求最短路径。
- 最大团问题：给定一个无向图，找到最大的完全子图，即子图中的任意两个顶点之间都有边相连。

在接下来的章节中，我们将一起攻克几个经典的回溯算法问题。

## 13.2. 全排列问题

全排列问题是回溯算法的一个典型应用。它的定义是在给定一个集合（如一个数组或字符串）的情况下，找出这个集合中元素的所有可能的排列。

如下表所示，列举了几个示例数组和其对应的所有排列。

输入数组	所有排列
[1]	[1]
[1, 2]	[1, 2], [2, 1]
[1, 2, 3]	[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]

### 13.2.1. 无重复的情况



输入一个整数数组，数组中不包含重复元素，返回所有可能的排列。

从回溯算法的角度看，我们可以把生成排列的过程想象成一系列选择的结果。假设输入数组为  $[1, 2, 3]$ ，如果我们先选择 1、再选择 3、最后选择 2，则获得排列  $[1, 3, 2]$ 。回退表示撤销一个选择，之后继续尝试其他选择。

从回溯算法代码的角度看，候选集合 `choices` 是输入数组中的所有元素，状态 `state` 是截至目前已被选择的元素。注意，每个元素只允许被选择一次，因此在遍历选择时，应当排除已经选择过的元素。

如下图所示，我们可以将搜索过程展开成一个递归树，树中的每个节点代表当前状态 `state`。从根节点开始，经过三轮选择后到达叶节点，每个叶节点都对应一个排列。

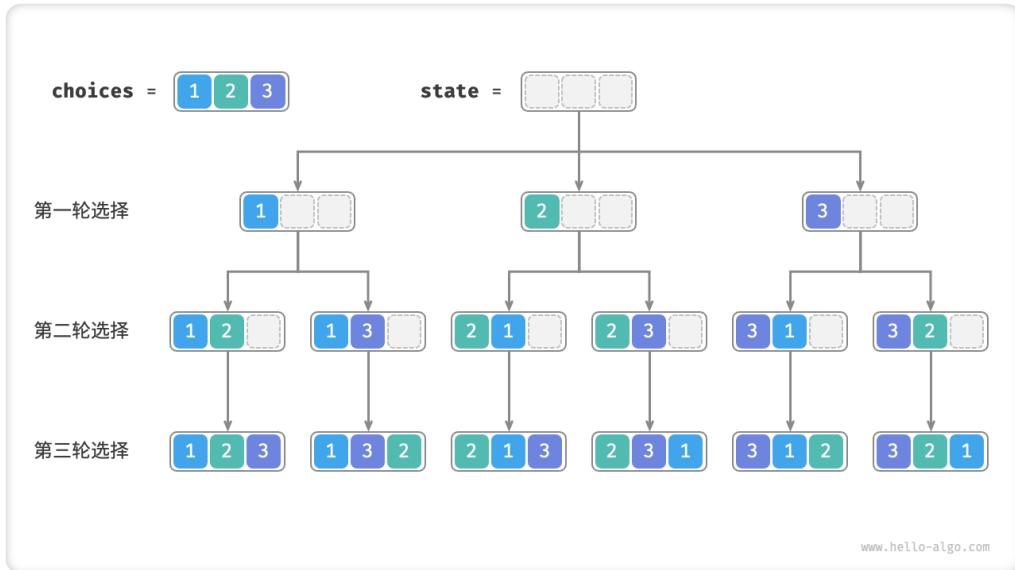


Figure 13-4. 全排列的递归树

想清楚以上信息之后，我们就可以在框架代码中做“完形填空”了。为了缩短代码行数，我们不单独实现框架代码中的各个函数，而是将他们展开在 `backtrack()` 函数中。

```
# === File: permutations_i.py ===
def backtrack(
    state: list[int], choices: list[int], selected: list[bool], res: list[list[int]]
):
    """回溯算法：全排列 I"""
    # 当状态长度等于元素数量时，记录解
    if len(state) == len(choices):
        res.append(list(state))
        return
    # 遍历所有选择
    for i, choice in enumerate(choices):
        # 剪枝：不允许重复选择元素
        if not selected[i]:
            # 尝试：做出选择，更新状态
            selected[i] = True
            state.append(choice)
            # 进行下一轮选择
            backtrack(state, choices, selected, res)
            # 回退：撤销选择，恢复到之前的状态
            selected[i] = False
            state.pop()
```

```
def permutations_i(nums: list[int]) -> list[list[int]]:
    """ 全排列 I """
    res = []
    backtrack(state=[], choices=nums, selected=[False] * len(nums), res=res)
    return res
```

需要重点关注的是，我们引入了一个布尔型数组 `selected`，它的长度与输入数组长度相等，其中 `selected[i]` 表示 `choices[i]` 是否已被选择。我们利用 `selected` 避免某个元素被重复选择，从而实现剪枝。

如下图所示，假设我们第一轮选择 1，第二轮选择 3，第三轮选择 2，则需要在第二轮剪掉元素 1 的分支，在第三轮剪掉元素 1, 3 的分支。从本质上理解，此剪枝操作可将搜索空间大小从  $O(n^n)$  降低至  $O(n!)$ 。

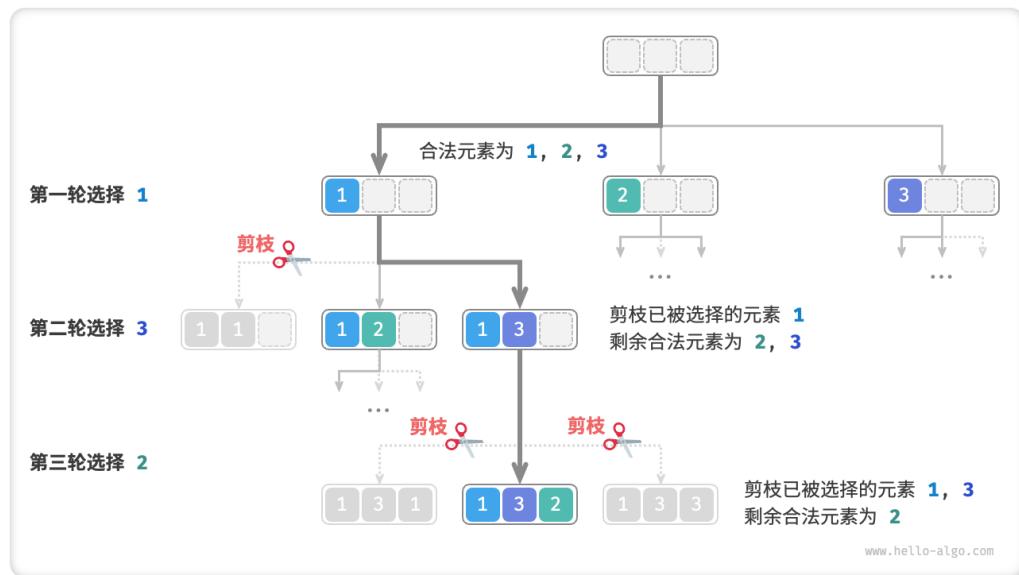


Figure 13-5. 全排列剪枝示例

### 13.2.2. 考虑重复的情况



输入一个整数数组，数组中可能包含重复元素，返回所有不重复的排列。

假设输入数组为  $[1, 1, 2]$ 。为了方便区分两个重复的元素 1，接下来我们将第二个元素记为  $\hat{1}$ 。如下图所示，上述方法生成的排列有一半都是重复的。

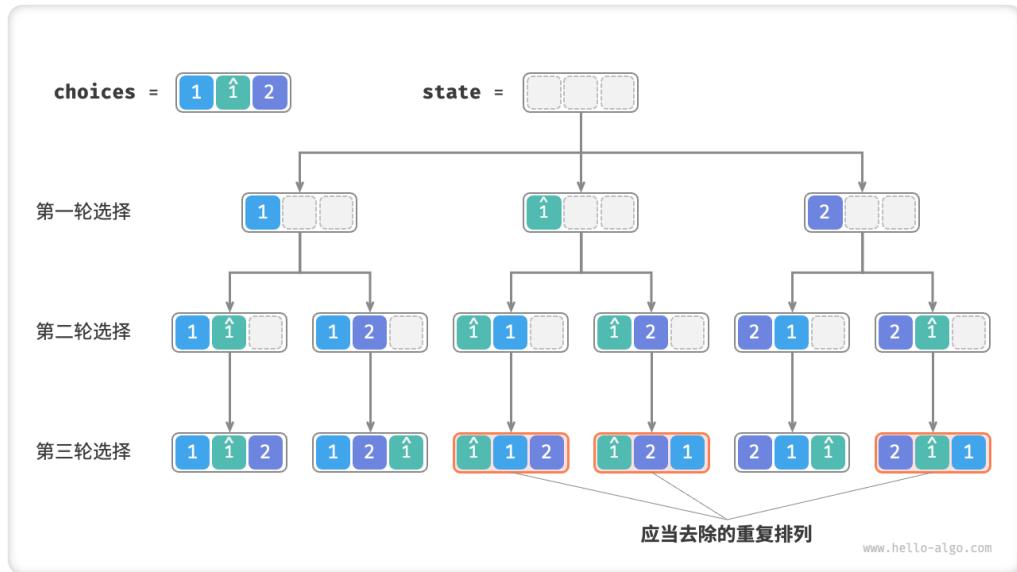


Figure 13-6. 重复排列

那么，如何去除重复的排列呢？最直接地，我们可以借助一个哈希表，直接对排列结果进行去重。然而，这样做不够优雅，因为生成重复排列的搜索分支是没有必要的，应当被提前识别并剪枝，这样可以提升算法效率。

观察发现，在第一轮中，选择 1 或选择  $\hat{1}$  是等价的，因为在这两个选择之下生成的所有排列都是重复的。因此，我们应该把  $\hat{1}$  剪枝掉。同理，在第一轮选择 2 后，第二轮选择中的 1 和  $\hat{1}$  也会产生重复分支，因此也需要将第二轮的  $\hat{1}$  剪枝。

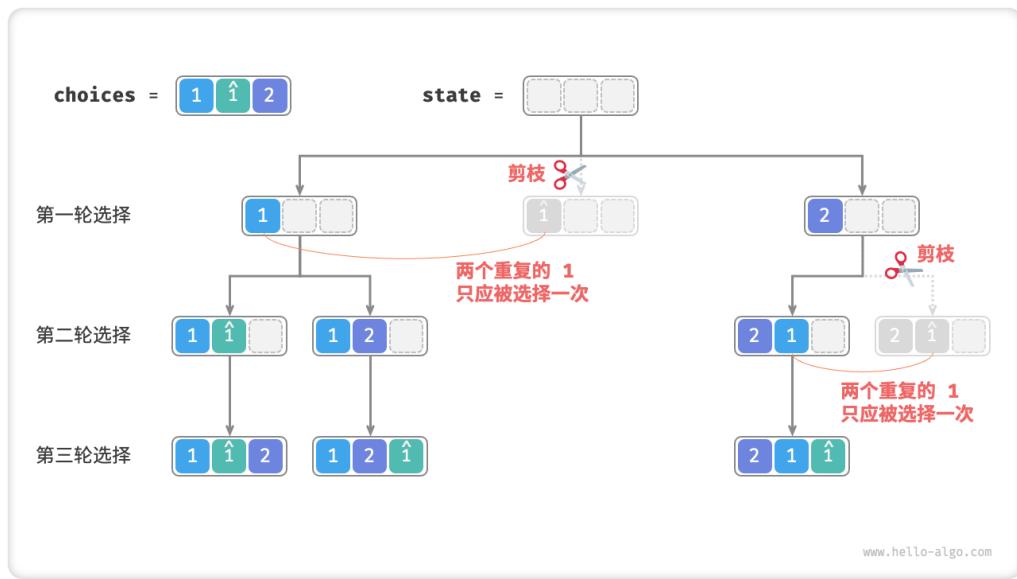


Figure 13-7. 重复排列剪枝

本质上看，我们的目标是实现在某一轮选择中，多个相等的元素仅被选择一次。因此，在上一题的代码的基础上，我们考虑在每一轮选择中开启一个哈希表 `duplicated`，用于记录该轮中已经尝试过的元素，并将重复元素剪枝。

```
# == File: permutations_ii.py ==
def backtrack(
    state: list[int], choices: list[int], selected: list[bool], res: list[list[int]]
):
    """ 回溯算法：全排列 II """
    # 当状态长度等于元素数量时，记录解
    if len(state) == len(choices):
        res.append(list(state))
        return
    # 遍历所有选择
    duplicated = set[int]()
    for i, choice in enumerate(choices):
        # 剪枝：不允许重复选择元素 且 不允许重复选择相等元素
        if not selected[i] and choice not in duplicated:
            # 尝试：做出选择，更新状态
            duplicated.add(choice) # 记录选择过的元素值
            selected[i] = True
            state.append(choice)
            # 进行下一轮选择
            backtrack(state, choices, selected, res)
            # 回退：撤销选择，恢复到之前的状态
            selected[i] = False
            state.pop()
    def permutations_ii(nums: list[int]) -> list[list[int]]:
        """ 全排列 II """
        res = []
        backtrack(state=[], choices=nums, selected=[False] * len(nums), res=res)
        return res
```

注意，虽然 `selected` 和 `duplicated` 都起到剪枝的作用，但他们剪掉的是不同的分支：

- **剪枝条件一：**整个搜索过程中只有一个 `selected`。它记录的是当前状态中包含哪些元素，作用是避免某个元素在 `state` 中重复出现。
- **剪枝条件二：**每轮选择（即每个开启的 `backtrack` 函数）都包含一个 `duplicated`。它记录的是在遍历中哪些元素已被选择过，作用是保证相等元素只被选择一次，以避免产生重复的搜索分支。

下图展示了两个剪枝条件的生效范围。注意，树中的每个节点代表一个选择，从根节点到叶节点的路径上的各个节点构成一个排列。

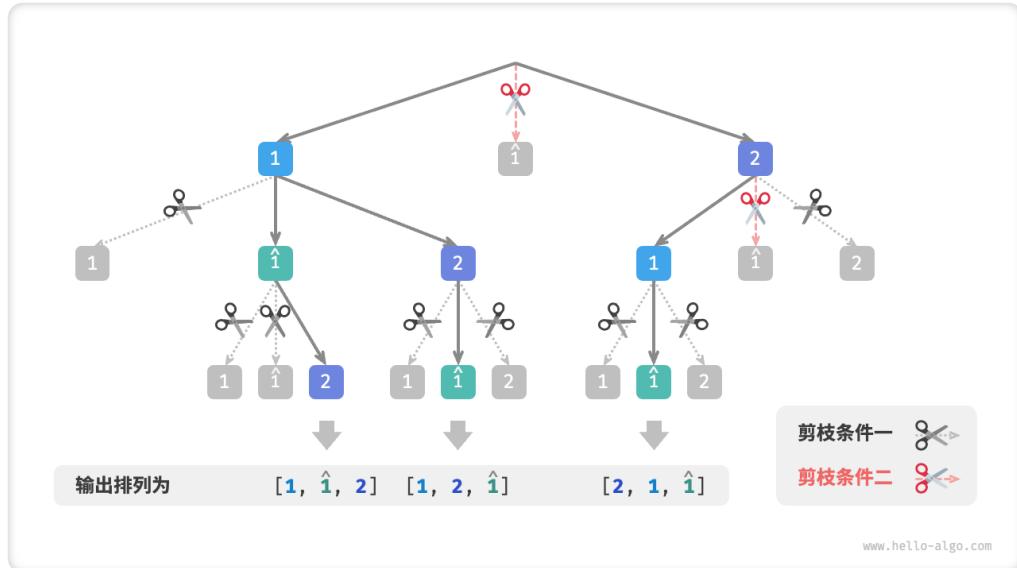


Figure 13-8. 两种剪枝条件的作用范围

### 13.2.3. 复杂度分析

假设元素两两之间互不相同，则  $n$  个元素共有  $n!$  种排列（阶乘）；在记录结果时，需要复制长度为  $n$  的列表，使用  $O(n)$  时间。因此，时间复杂度为  $O(n!n)$ 。

最大递归深度为  $n$ ，使用  $O(n)$  栈帧空间。`selected` 使用  $O(n)$  空间。同一时刻最多共有  $n$  个 `duplicated`，使用  $O(n^2)$  空间。因此，全排列 I 的空间复杂度为  $O(n)$ ，全排列 II 的空间复杂度为  $O(n^2)$ 。

## 13.3. N 皇后问题



根据国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。给定  $n$  个皇后和一个  $n \times n$  大小的棋盘，寻找使得所有皇后之间无法相互攻击的摆放方案。

如下图所示，当  $n = 4$  时，共可以找到两个解。从回溯算法的角度看， $n \times n$  大小的棋盘共有  $n^2$  个格子，给出了所有的选择 `choices`。在逐个放置皇后的过程中，棋盘状态在不断地变化，每个时刻的棋盘就是状态 `state`。

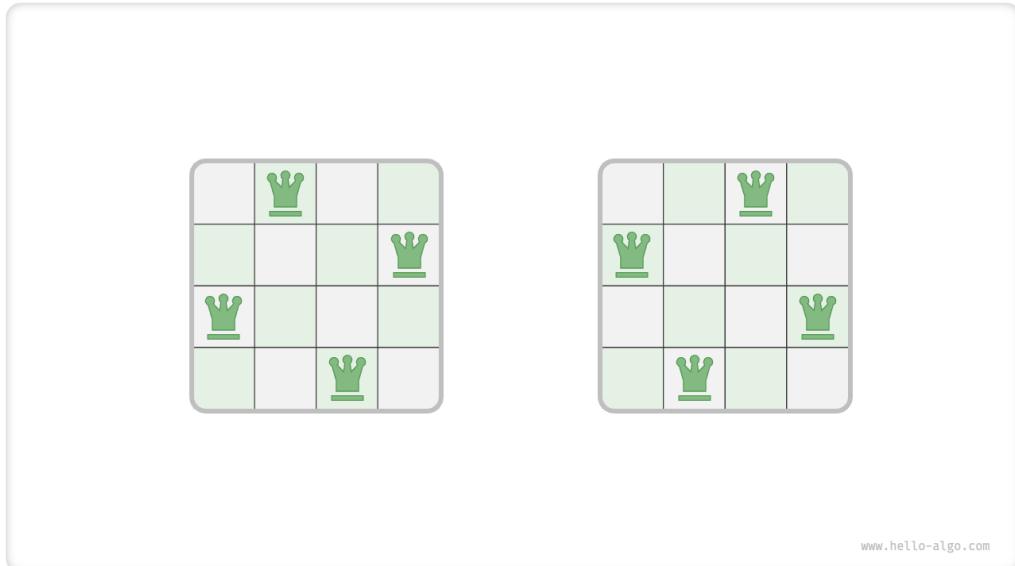


Figure 13-9. 4 皇后问题的解

本题共有三个约束条件：**多个皇后不能在同一行、同一列和同一对角线**。值得注意的是，对角线分为主对角线 \ 和副对角线 / 两种。

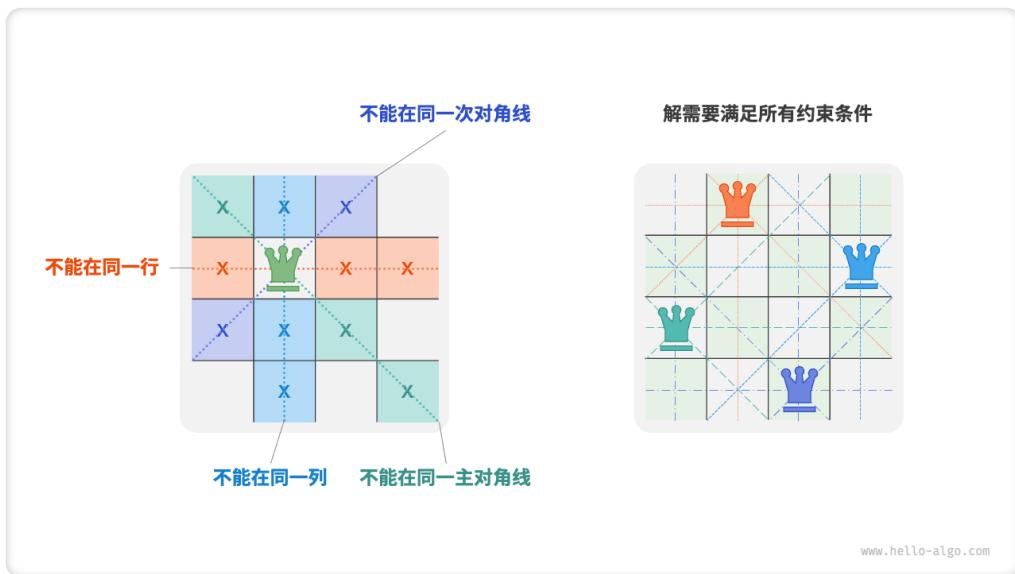


Figure 13-10. n 皇后问题的约束条件

皇后的数量和棋盘的行数都为  $n$ ，因此我们容易得到第一个推论：**棋盘每行都允许且只允许放置一个皇后**。这意味着，我们可以采取逐行放置策略：从第一行开始，在每行放置一个皇后，直至最后一行结束。**此策略起到了剪枝的作用**，它避免了同一行出现多个皇后的所有搜索分支。

下图展示了 4 皇后问题的逐行放置过程。受篇幅限制，下图仅展开了第一行的一个搜索分支。在搜索过程中，我们将不满足列约束和对角线约束的方案都剪枝了。

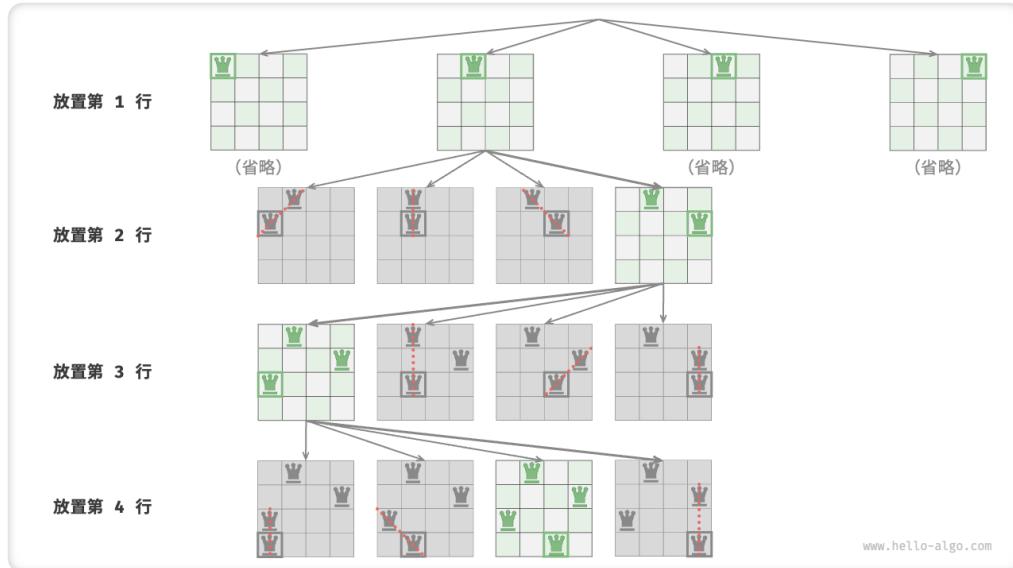


Figure 13-11. 逐行放置策略

为了实现根据列约束剪枝，我们可以利用一个长度为  $n$  的布尔型数组 `cols` 记录每一列是否有皇后。在每次决定放置前，我们通过 `cols` 将已有皇后的列剪枝，并在回溯中动态更新 `cols` 的状态。

那么，如何处理对角线约束呢？设棋盘中某个格子的行列索引为  $(\text{row}, \text{col})$ ，观察矩阵的某条主对角线，我们发现该对角线上所有格子的行索引减列索引相等，即  $\text{row} - \text{col}$  为恒定值。换句话说，若两个格子满足  $\text{row}_1 - \text{col}_1 == \text{row}_2 - \text{col}_2$ ，则这两个格子一定处在一条主对角线上。

利用该性质，我们可以借助一个数组 `diags1` 来记录每条主对角线上是否有皇后。注意， $n$  维方阵  $\text{row} - \text{col}$  的范围是  $[-n + 1, n - 1]$ ，因此共有  $2n - 1$  条主对角线。

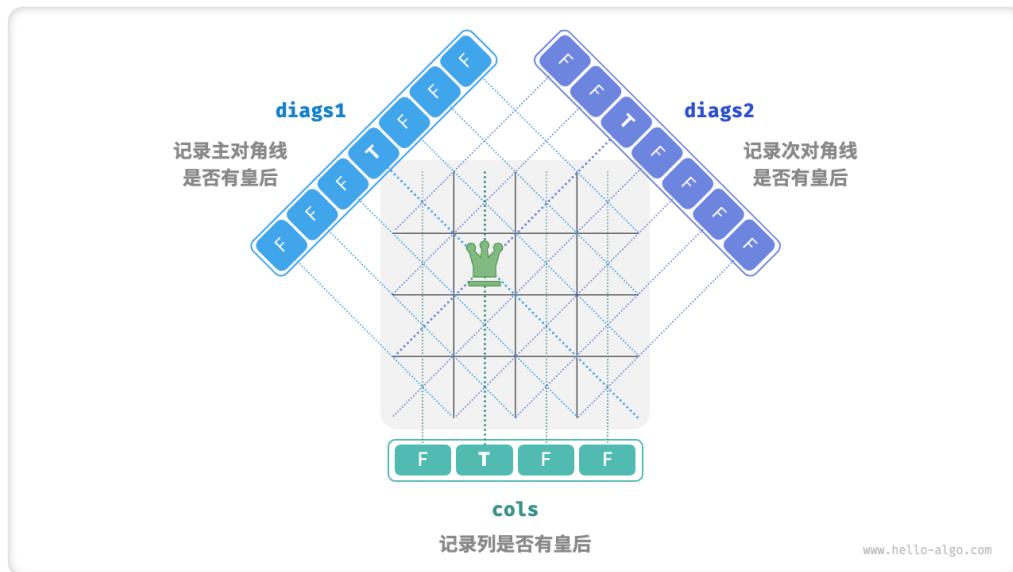


Figure 13-12. 处理列约束和对角线约束

同理，次对角线上的所有格子的 `row + col` 是恒定值。我们可以使用同样的方法，借助数组 `diag2` 来处理次对角线约束。

根据以上分析，我们便可以写出  $n$  皇后的解题代码。

```
# === File: n_queens.py ===
def backtrack(
    row: int,
    n: int,
    state: list[list[str]],
    res: list[list[list[str]]],
    cols: list[bool],
    diags1: list[bool],
    diags2: list[bool],
):
    """ 回溯算法：N 皇后 """
    # 当放置完所有行时，记录解
    if row == n:
        res.append([list(row) for row in state])
        return
    # 遍历所有列
    for col in range(n):
        # 计算该格子对应的主对角线和副对角线
        diag1 = row - col + n - 1
        diag2 = row + col
        # 剪枝：不允许该格子所在（列 或 主对角线 或 副对角线）包含皇后
        if not (cols[col] or diags1[diag1] or diags2[diag2]):
            # 尝试：将皇后放置在该格子
            state[row][col] = "Q"
            cols[col] = diags1[diag1] = diags2[diag2] = True
            # 放置下一行
            backtrack(row + 1, n, state, res, cols, diags1, diags2)
            # 回退：将该格子恢复为空位
            state[row][col] = "#"
            cols[col] = diags1[diag1] = diags2[diag2] = False

def n_queens(n: int) -> list[list[list[str]]]:
    """ 求解 N 皇后 """
    # 初始化 n*n 大小的棋盘，其中 'Q' 代表皇后，'#' 代表空位
    state = [[="#" for _ in range(n)] for _ in range(n)]
    cols = [False] * n # 记录列是否有皇后
    diags1 = [False] * (2 * n - 1) # 记录主对角线是否有皇后
    diags2 = [False] * (2 * n - 1) # 记录副对角线是否有皇后
    res = []
    backtrack(0, n, state, res, cols, diags1, diags2)

    return res
```

### 13.3.1. 复杂度分析

逐行放置  $n$  次，考虑列约束，则从第一行到最后一行分别有  $n, n - 1, \dots, 2, 1$  个选择，因此时间复杂度为  $O(n!)$ 。实际上，根据对角线约束的剪枝也能够大幅地缩小搜索空间，因而搜索效率往往优于以上时间复杂度。

`state` 使用  $O(n^2)$  空间，`cols`, `diags1`, `diags2` 皆使用  $O(n)$  空间。最大递归深度为  $n$ ，使用  $O(n)$  栈帧空间。因此，空间复杂度为  $O(n^2)$ 。

# 14. 附录

## 14.1. 编程环境安装

### 14.1.1. VSCode

本书推荐使用开源轻量的 VSCode 作为本地 IDE，下载并安装 [VSCode](#)。

### 14.1.2. 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 Extension Pack for Java。

### 14.1.3. 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。
3. (可选) 打开 Settings 页面，搜索 `Clang_format_fallback_style` 代码格式化选项，设置为`{ BasedOnStyle: Microsoft, BreakBeforeBraces: Attach }`。

### 14.1.4. 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。
3. (可选) 在命令行输入 `pip install black`，安装代码格式化工具。

### 14.1.5. 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。

### 14.1.6. 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。
3. (可选) 在 VSCode 的插件市场中搜索 `Prettier`，安装代码格式化工具。

### 14.1.7. 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#`。

### 14.1.8. 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](#)。

### 14.1.9. 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](#)。

## 14.2. 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。

由于作者能力有限，书中难免存在一些遗漏和错误，请您谅解。如果您发现了笔误、失效链接、内容缺失、文字歧义、解释不清晰或行文结构不合理等问题，请协助我们进行修正，以帮助其他读者获得更优质的学习资源。所有[撰稿人](#)将在仓库和网站主页上展示，以感谢他们对开源社区的无私奉献！

### 14.2.1. 内容微调

在每个页面的右上角有一个「编辑」图标，您可以按照以下步骤修改文本或代码：

1. 点击编辑按钮，如果遇到“需要 Fork 此仓库”的提示，请同意该操作；
2. 修改 Markdown 源文件内容，并确保内容正确，同时尽量保持排版格式的统一；
3. 在页面底部填写修改说明，然后点击“Propose file change”按钮；页面跳转后，点击“Create pull request”按钮即可发起拉取请求。



Figure 14-1. 页面编辑按键

由于图片无法直接修改，因此需要通过新建 Issue 或评论留言来描述图片问题，我们会尽快重新绘制并替换图片。

### 14.2.2. 内容创作

如果您有兴趣参与此开源项目，包括将代码翻译成其他编程语言、扩展文章内容等，那么需要实施 Pull Request 工作流程：

1. 登录 GitHub，将[本仓库](#) Fork 到个人账号下；
2. 进入您的 Fork 仓库网页，使用 git clone 命令将仓库克隆至本地；
3. 在本地进行内容创作，并通过运行测试以验证代码的正确性；
4. 将本地所做更改 Commit，然后 Push 至远程仓库；
5. 刷新仓库网页，点击“Create pull request”按钮即可发起拉取请求；

### 14.2.3. Docker 部署

我们可以通过 Docker 来部署本项目。执行以下脚本，稍等片刻后，即可使用浏览器打开 <http://localhost:8000> 来访问本项目。

```
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
```