c13n #31

c13n

2025 年 11 月 19 日

第 I 部

循环队列（Circular Queue）数据结构

马浩琨

Sep 13

在日常生活中，我们经常遇到排队的场景，例如在超市结账时，顾客按照先来后到的顺序依次处理，这就是队列的直观体现。队列是一种先进先出（FIFO）的线性数据结构，它只允许在队尾添加元素，在队头删除元素。这种特性使得队列在计算机科学中广泛应用于任务调度、缓冲处理等场景。

然而，当使用普通数组实现队列时，我们会遇到一个棘手的问题：假溢出。假溢出指的是数组前端有可用的空闲空间，但由于队尾指针已经到达数组末尾，无法再插入新元素，导致空间浪费。这种现象不仅降低了存储效率，还可能引发程序错误。为了解决这个问题，循环队列应运而生。循环队列通过将数组首尾相连，逻辑上形成一个环，从而高效利用空间，避免了假溢出。

1 循环队列的核心思想与工作原理

循环队列是一种基于固定大小数组的数据结构，它通过两个指针（front 和 rear）的循环移动来实现队列操作。其核心在于“循环”二字：当指针到达数组末尾时，通过取模运算，下一个位置会回到数组开头，从而形成逻辑上的环。

初始化时，front 和 rear 指针都设置为 0，表示队列为空。入队操作时，首先检查队列是否已满，然后将元素放入 rear 指向的位置，并将 rear 指针前进一位，使用公式 `rear = (rear + 1) % capacity` 来实现循环。出队操作时，检查队列是否为空，然后取出 front 指向的元素，并将 front 指针前进一位，同样使用公式 `front = (front + 1) % capacity`。

取模运算在这里扮演了关键角色。例如，假设数组容量为 5，当 rear 为 4 时，`(4 + 1) % 5 = 0`，这意味着 rear 指针会从末尾回到起点，实现了循环移动。这种机制确保了队列空间的高效利用，避免了假溢出。

2 循环队列的边界条件处理

在循环队列中，一个重要的挑战是如何区分队列“空”和“满”的状态。因为当队列空时，front 等于 rear；当队列满时，front 也可能等于 rear，这会导致混淆。为了解决这个问题，常用的方法有两种：浪费一个空间法和维护计数器法。

浪费一个空间法是最常用且直观的方法。它规定数组中始终浪费一个单元，作为“满”的标识。判空条件是 `front == rear`，判满条件是 `(rear + 1) % capacity == front`。这种方法的优点是逻辑清晰，代码简单，缺点是牺牲了一个存储单元。例如，如果数组容量为 5，那么实际只能存储 4 个元素。

维护计数器法则是通过一个额外变量 count 来记录队列中的元素个数。判空时检查 `count == 0`，判满时检查 `count == capacity`。这种方法的优点是不浪费空间，但缺点是需要维护计数器，增加了操作的开销。

在本文的后续代码实现中，我们将采用浪费一个空间法，因为它更常见且易于理解。

3 代码实现

以下是一个基于 Python 的循环队列实现。我们定义一个类 CircularQueue，包含成员变量 queue（列表）、front、rear 和 capacity。

```python
class CircularQueue:
    def __init__(self, k):
        self.capacity = k
        self.queue = [None] * k # 初始化固定大小的数组
        self.front = 0
        self.rear = 0

    def enqueue(self, value):
        if self.is_full():
            raise Exception("Queue is full")
        self.queue[self.rear] = value
        self.rear = (self.rear + 1) % self.capacity # 循环移动 rear 指针

    def dequeue(self):
        if self.is_empty():
            raise Exception("Queue is empty")
        item = self.queue[self.front]
        self.front = (self.front + 1) % self.capacity # 循环移动 front 指针
        return item

    def get_front(self):
        if self.is_empty():
            return -1
        return self.queue[self.front]

    def get_rear(self):
        if self.is_empty():
            return -1
        return self.queue[(self.rear - 1) % self.capacity] # 注意 rear 指向的是下一个空位，所以取前一个位置

    def is_empty(self):
        return self.front == self.rear

    def is_full(self):
        return (self.rear + 1) % self.capacity == self.front
```

在初始化方法 `__init__` 中，我们创建了一个大小为 k 的数组，并初始化指针。入队方法 enqueue 首先检查队列是否已满，然后插入元素并更新 rear 指针。出队方法 dequeue 检查是否为空，然后取出元素并更新 front 指针。获取队头和队尾元素的方法中，需要注意

rear 指针指向的是下一个空位，因此获取队尾时需要使用 (rear - 1) % capacity 来定位最后一个元素。判空和判满方法基于浪费一个空间法的条件实现。

4 循环队列的应用场景

循环队列在计算机系统中有着广泛的应用。在操作系统中，它用于进程调度和消息传递，确保任务按照顺序处理。在网络领域，循环队列作为数据包缓冲区，帮助实现流量控制，避免数据丢失。在多媒体应用中，如音乐播放器或视频流，循环队列用于管理播放列表或缓冲数据，提供流畅的用户体验。此外，在任何需要高效、有界的生产者-消费者模型场景中，循环队列都能发挥其优势，例如实时数据处理和事件队列管理。

循环队列的主要优点是高效利用固定大小的内存空间，解决了假溢出问题，并且所有操作的时间复杂度都是 $O(1)$。与普通数组队列相比，循环队列在空间利用上更高效，而普通队列可能因为假溢出浪费空间。在操作时间复杂度上，两者都是 $O(1)$，但循环队列的实现稍复杂，需要处理循环和边界条件。普通队列适用于无界或数据量未知的场景，而循环队列更适合有界、固定大小的缓冲区场景。

通过本文的讲解，希望读者能深入理解循环队列的核心思想，并动手实现它。这种数据结构不仅提升了效率，也体现了计算机科学中的精巧设计。鼓励读者在实际项目中应用循环队列，并探索其变种，如动态扩容版本。

第 II 部

哈希表数据结构

马浩琨

Sep 14, 2025

在日常计算中，我们经常需要快速查找和存储数据，例如在字典中查询单词、在电话簿中寻找联系人，或在搜索引擎中检索信息。这些场景都要求高效的数据访问机制。传统的数据结构如数组或链表，在查找操作时可能需要遍历整个集合，导致时间复杂度为 O(n)，这在数据量大的情况下效率低下。二叉搜索树虽然将查找效率提升到 O(log n)，但仍然存在改进空间。哈希表作为一种经典的数据结构，通过巧妙的「时空权衡」策略，在平均情况下能够实现近乎 O(1) 的查找、插入和删除操作，从而成为许多高性能应用的核心组件。本文将带领读者从理论出发，逐步实现一个功能完整的哈希表，使用 Python 语言进行演示，并深入分析其性能特性。

5 哈希表的核心概念

哈希表是一种基于哈希函数构建的数据结构，它通过将键映射到数组中的特定位置来实现快速数据访问。其核心组件包括哈希函数、底层数组和冲突解决策略。哈希函数的作用是将任意大小的输入键转换为一个固定大小的整数值，即哈希值。理想的哈希函数应具备确定性、高效性和均匀性。确定性确保相同的键总是产生相同的哈希值；高效性要求计算速度快；均匀性则保证哈希值分布均匀，以减少冲突的发生。

在哈希表中，我们使用数组作为底层存储，每个数组元素称为一个「哈希桶」。通过哈希函数计算出的哈希值需要映射到数组索引，通常使用取模运算：$index = \text{hash}(key) \bmod \text{capacity}$，其中 capacity 表示数组的容量。为了管理哈希表的性能，我们引入负载因子的概念，定义为当前元素数量与数组容量的比值：$\text{load\_factor} = \frac{\text{number\_of\_entries}}{\text{capacity}}$。负载因子是触发动态扩容的关键指标，当它超过预设阈值时，哈希表需要进行扩容以维持高效操作。

6 哈希冲突：不可避免的挑战

尽管哈希函数旨在均匀分布键，但由于「鸽巢原理」的存在，冲突是不可避免的。鸽巢原理指出，如果键的数量超过数组容量，那么至少有两个键会映射到同一索引。因此，哈希表必须采用有效的冲突解决策略。主流方法包括链地址法和开放定址法。

链地址法通过将每个哈希桶实现为一个链表来处理冲突。所有映射到同一索引的键值对都存储在该链表中。插入操作时，新元素被追加到链表末尾；查找或删除操作时，需要遍历链表以匹配键。这种方法的优点是实现简单且能有效处理冲突，但缺点是需要额外的指针空间，且缓存性能较差。

开放定址法则将所有元素直接存储在数组中，当冲突发生时，按照某种探测序列寻找下一个空闲槽位。常见探测方法包括线性探测、平方探测和双重哈希。线性探测使用公式 $index = (\text{hash}(key) + i) \mod \text{capacity}$ 进行顺序查找；平方探测使用 $index = (\text{hash}(key) + i^2) \mod \text{capacity}$ 以减少聚集现象；双重哈希则引入第二个哈希函数计算步长。开放定址法的优点是数据集中在数组中，缓存友好，但实现复杂，删除操作需使用标记删除法，且容易产生聚集。

7　动手实现一个哈希表（采用链地址法）

我们将使用 Python 实现一个基于链地址法的哈希表。首先定义 HashMap 类，初始化时设置初始容量和负载因子阈值。代码中，我们使用列表来表示哈希桶，每个桶初始化为空列表。

```
class HashMap:
    def __init__(self, capacity=10, load_factor_threshold=0.75):
        self.capacity = capacity
        self.load_factor_threshold = load_factor_threshold
        self.size = 0
        self.buckets = [[] for _ in range(self.capacity)]
```

这里，capacity 默认值为 10，load_factor_threshold 设置为 0.75。buckets 是一个列表的列表，每个元素代表一个哈希桶，初始为空。size 记录当前元素数量，用于计算负载因子。

接下来实现 set 方法用于插入或更新键值对。首先计算键的哈希值并映射到索引，然后遍历对应桶的链表。如果找到相同键，则更新值；否则追加新键值对。完成后检查负载因子，若超过阈值则调用 _resize 方法扩容。

```
    def set(self, key, value):
        index = hash(key) % self.capacity
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))
        self.size += 1
        if self.load_factor > self.load_factor_threshold:
            self._resize()
```

hash(key) 是 Python 内置函数，返回键的哈希值。取模运算确保索引在数组范围内。遍历桶中元素时，使用枚举来访问索引和键值对。如果键已存在，更新值并返回；否则添加新元素并增加 size。负载因子通过属性计算，我们稍后定义。

get 方法用于查找键对应的值。计算索引后，遍历桶链表，找到匹配键则返回值，否则返回 None。

```
    def get(self, key):
        index = hash(key) % self.capacity
        bucket = self.buckets[index]
        for k, v in bucket:
            if k == key:
                return v
```

```
        return None
```

delete 方法类似，遍历桶链表找到并删除键值对。

```
    def delete(self, key):
        index = hash(key) % self.capacity
        bucket = self.buckets[index]
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                self.size -= 1
                return
```

动态扩容通过 _resize 方法实现。当负载因子超过阈值时，创建一个新数组，容量通常翻倍，然后重新哈希所有元素到新数组中。

```
    def _resize(self):
        new_capacity = self.capacity * 2
        new_buckets = [[] for _ in range(new_capacity)]
        for bucket in self.buckets:
            for key, value in bucket:
                index = hash(key) % new_capacity
                new_buckets[index].append((key, value))
        self.buckets = new_buckets
        self.capacity = new_capacity
```

这里，新容量为原容量的两倍。遍历所有旧桶中的键值对，重新计算哈希索引并插入新桶。最后更新哈希表的 buckets 和 capacity。
此外，我们可以添加辅助方法如 load_factor 属性和 len 方法。

```
    @property
    def load_factor(self):
        return self.size / self.capacity

    def __len__(self):
        return self.size
```

load_factor 使用属性装饰器实现计算属性，返回当前负载因子。len 方法返回元素数量，便于使用 len() 函数。

8 性能分析与进阶话题

哈希表的性能高度依赖于哈希函数和冲突解决策略。在最佳情况下，当键均匀分布时，操作时间复杂度为 O(1)；最坏情况下，所有键冲突，性能退化为 O(n)。平均情况下，基于负载因子，时间复杂度通常为 O(1)。负载因子的选择直接影响性能；例如，阈值 0.75 在空间和时间效率之间提供了平衡。

设计一个好的哈希函数是关键。虽然加密哈希函数如 MD5 或 SHA-1 提供均匀分布，但计算开销大，不适合哈希表。实用哈希函数如 FNV-1 或 DJB2 针对字符串优化，能高效产生均匀哈希值。例如，DJB2 哈希函数使用迭代计算：$hash = ((hash << 5) + hash) + char$，其中 char 是字符串的字符代码。

冲突解决策略的选择也影响性能。链地址法简单可靠，但额外空间开销；开放定址法缓存友好，但容易聚集。在实际应用中，如 Python 的 dict 或 Java 的 HashMap，都采用优化策略结合动态扩容以确保高效性。

哈希表以其高效的查找、插入和删除操作，成为计算机科学中不可或缺的数据结构。优势在于平均 O(1) 的时间复杂度，但劣势包括不支持顺序遍历和性能对哈希函数的依赖。经典应用场景包括数据库索引、缓存系统如 Redis、字典实现以及对象属性存储。通过亲手实现哈希表，读者可以更深入理解其内部机制，并尝试优化如哈希函数选择或冲突策略，以提升实践能力。哈希表不仅是理论上的经典，更是工程中的利器，值得每一位开发者掌握。

第 III 部

基数排序（Radix Sort）算法

黄梓淳

Sep 15, 2025

排序算法是计算机科学中的基础主题，传统上我们依赖于比较操作，如快速排序或归并排序，它们通过元素间的比较来确定顺序。但有一个问题：排序一定要通过两两比较吗？答案是否定的。基数排序（Radix Sort）提供了一种全新的视角，它是一种非比较型整数排序算法，基于键值的各位数字或字符进行逐位处理。这种方法的核心思想是稳定地按位排序，从而避免了许多比较操作带来的开销。

基数排序的核心价值在于其线性时间复杂度，通常表示为 $O(n \times k)$，其中 n 是元素个数，k 是最大位数。这使得它在处理大规模固定长度数据时表现出色，例如排序手机号码、学号或 IP 地址。在本文中，我将带领您深入理解基数排序的原理，亲手实现它，并分析其性能与适用边界。通过这篇文章，您将不仅学会如何编码，还能 grasp 其背后的思想。

9　算法核心原理剖析

基数排序的基本思想是逐位处理数字，从最低位（LSD，Least Significant Digit）或最高位（MSD，Most Significant Digit）开始排序。一个经典的类比是扑克牌排序：假设您有一副牌，您可能先按花色排序，再按数字排序，但为了保持顺序，需要确保排序是稳定的。稳定性意味着相同键值的元素在排序后保持原有相对顺序，这对基数排序至关重要，因为高位排序时不能打乱低位已排好的顺序。

在 LSD 方法中，我们从最低位（如个位）开始，逐步向高位推进。例如，考虑数组 `[170, 45, 75, 90, 802, 24, 2, 66]`。首先，我们找到最大数字的位数（这里是 3，因为 802 有三位）。然后，进行三轮排序：第一轮按个位排序，第二轮按十位，第三轮按百位。每一轮使用一个稳定的排序算法（通常是计数排序）来处理当前位。可视化过程如下：初始数组为 `[170, 45, 75, 90, 802, 24, 2, 66]`；按个位排序后，顺序变为 `[170, 90, 802, 2, 24, 45, 75, 66]`；按十位排序后，变为 `[802, 2, 24, 45, 66, 170, 75, 90]`；最后按百位排序，得到最终有序数组 `[2, 24, 45, 66, 75, 90, 170, 802]`。这个过程展示了如何通过逐位稳定排序达到整体有序。

10　实现细节与代码解析

在实现基数排序时，我们选择计数排序作为辅助算法，因为它具有稳定性和线性时间复杂度，完美契合基数排序的需求。计数排序的核心是统计每个数字出现的次数，并通过累积计数来确定元素位置。下面，我将使用 Python 语言逐步实现 LSD 基数排序，并对代码进行详细解读。

首先，我们需要两个辅助函数：一个用于获取数组中的最大数字的位数，另一个用于获取数字在特定位上的数字。函数 `getMaxDigits` 遍历数组，找到最大数字并计算其位数。这通过将数字转换为字符串并取长度来实现，或者通过数学运算如连续除以 10。函数 `getDigit` 则提取数字在指定位上的数字，例如对于数字 123 和位索引 1（从右向左，0-based），它返回十位数字 2。

核心函数 `radixSort` 的实现步骤如下：计算最大位数 k，然后循环 k 次（从最低位到最高位）。在每一轮循环中，初始化一个大小为 10 的计数数组（因为十进制数字范围是 0-9）。接着，进行计数阶段：遍历数组，统计当前位上每个数字出现的次数。然后，将计数数组转换为累积计数数组，这有助于确定每个数字的最终位置。最后，重构数组：从后向前遍历原数组，根据当前位数字和计数数组，将元素放入临时输出数组的正确位置。从后向前遍历是

为了保持稳定性，确保相同数字的元素顺序不变。完成后，将输出数组复制回原数组。

以下是 Python 代码实现：

```
def getMaxDigits(arr):
    # 找到数组中的最大数字
    max_val = max(arr)
    # 计算最大数字的位数：通过转换为字符串取长度
    return len(str(max_val))

def getDigit(num, digit):
    # 获取数字在指定位上的数字，digit 从 0 开始（0 表示个位）
    # 例如，getDigit(123, 1) 返回 2（十位）
    return (num // (10 ** digit)) % 10

def radixSort(arr):
    # 获取最大位数
    k = getMaxDigits(arr)
    # 临时输出数组
    output = [0] * len(arr)
    # 进行 k 轮排序
    for digit in range(k):
        # 初始化计数数组，大小为 10（0-9）
        count = [0] * 10
        # 计数阶段：统计当前位上每个数字的出现次数
        for num in arr:
            d = getDigit(num, digit)
            count[d] += 1
        # 累加计数：将计数数组转换为累积计数
        for i in range(1, 10):
            count[i] += count[i-1]
        # 重构数组：从后向前遍历原数组，以保持稳定性
        for i in range(len(arr)-1, -1, -1):
            num = arr[i]
            d = getDigit(num, digit)
            output[count[d] - 1] = num
            count[d] -= 1
        # 将输出数组复制回原数组
        arr = output[:]
    return arr
```

代码解读：在 getMaxDigits 函数中，我们使用 max 函数找到最大值，然后通过 len(str(max_val)) 计算位数，这是一种简单直接的方法。在 getDigit 函数中，我们使用整数除法和模运算来提取特定位上的数字，例如 (num // (10 ** digit)) %

10 计算数字在 digit 位上的值。在 radixSort 函数中，外层循环运行 k 次，对应每位排序。计数数组 count 初始化为全零，然后遍历数组统计数字出现次数。累加计数步骤将 count 数组转换为每个数字的结束索引加一。重构数组时，从后向前遍历原数组，确保稳定性：将元素放入输出数组的指定位置，并递减计数。最后，复制输出数组回原数组，完成排序。这个实现的时间复杂度为 $O(n \times k)$，空间复杂度为 $O(n + 10)$，由于 10 是常数，通常简化为 $O(n)$。

11 进阶讨论与变体

基数排序的复杂度分析显示，时间复杂度为 $O(n \times k)$，其中 n 是元素个数，k 是最大位数。由于 k 通常相对较小（例如，对于 32 位整数，k 最大为 10），这可以被视为线性时间复杂度，优于许多比较排序算法如快速排序的 $O(n \log n)$。空间复杂度为 $O(n + b)$，b 是基数大小（这里 b=10），主要开销来自输出数组和计数数组。

LSD 和 MSD 是基数排序的两种变体。LSD 从最低位开始排序，实现简单，适用于位数较少的数，但必须完成所有位的排序。MSD 从最高位开始，类似递归的桶排序，可能不需要比较所有位（如果高位已能区分大小），但实现更复杂，需要处理递归开销和空桶，适用于字符串排序或字典序场景。

处理负数时，基数排序需要额外步骤。常见方法是将数组分割成负数和正数两部分。对负数部分，取绝对值后使用基数排序，然后反转顺序（因为负数取绝对值后排序顺序相反）。对正数部分直接排序，最后合并两部分。另一种方法是偏移法：将所有数加上一个最小值偏移量，使其变为非负数，排序后再减回去。例如，如果数组中有负数，先找到最小值 min_val，然后将每个元素加上 abs(min_val)，排序后再减去 abs(min_val)。

基数排序的优点包括线性时间复杂度，高效处理大规模固定长度数据，如整数或字符串。缺点是非原地排序，需要额外空间；对数据类型有限制（只适用于可分解为位的类型）；当 k 很大时（数字非常长），效率可能下降。

典型应用场景包括数据库中对整数键的排序、计算机图形学中的算法（如深度排序）、以及后缀数组的构造。在这些领域，基数排序的线性性能优势明显。

结束语：基数排序是一种独特而高效的算法，它突破了传统比较排序的局限。通过理解其原理和实现，您可以更好地应用它到实际问题中。我鼓励您动手实现一遍代码，以加深印象。

12 互动与延伸

思考题：如何修改代码来排序字符串数组？例如，对单词列表按字典序排序，这可以通过将每个字符视为一位，使用类似方法处理。如果数字的进制不是十进制，比如二进制或十六进制，算法需要调整基数大小（b），例如二进制时 b=2，计数数组大小相应改变。

相关资源：您可以访问可视化排序网站如 VisuAlgo，观看基数排序的动态演示。对于更深入的讨论，推荐阅读关于 MSD 基数排序的学术文章，以探索其递归实现和优化。

第 IV 部

平衡二叉树（Balanced Binary Tree）

数据结构

黄京

Sep 16, 2025

本文将深入探讨平衡二叉树的核心概念，重点解析为何需要它、它是如何通过旋转操作维持平衡的。我们将以最经典的 AVL 树为例，逐步拆解其四大旋转操作，并最终用代码（C++）完整实现一个具备插入、删除和查询功能的 AVL 树数据结构。无论你是正在学习数据结构的学生，还是希望巩固基础的开发者，这篇文章都将为你提供清晰的指引。

二叉搜索树（BST）是一种常见的数据结构，它具有高效的搜索、插入和删除操作，理想情况下的时间复杂度为 $O(h)$，其中 h 是树的高度。BST 的定义基于每个节点的值大于其左子树的所有值且小于其右子树的所有值。然而，BST 有一个致命的缺陷：当插入的数据是有序的，例如序列 1, 2, 3, 4, 5，BST 会退化成一条链表，此时高度 h 等于节点数 n，操作时间复杂度降至 $O(n)$，效率急剧下降。

为了解决这个问题，平衡二叉树应运而生。平衡二叉树的核心思想是在 BST 的基础上，通过某些策略控制树的高度，使其尽可能保持「矮胖」的形状，从而保证操作效率始终维持在 $O(\log n)$。常见的平衡树类型包括 AVL 树、红黑树和 Treap 等。本文将聚焦于最基础且易于理解的 AVL 树，因为它提供了严格的平衡保证和相对简单的实现方式。

13　AVL 树：定义与核心概念

AVL 树是一种自平衡的二叉搜索树，由 Adelson-Velsky 和 Landis 在 1962 年提出。它首先满足 BST 的所有性质，但额外要求每个节点的平衡因子（Balance Factor, BF）必须保持在 -1、0 或 1 的范围内。平衡因子定义为该节点左子树的高度减去右子树的高度，即 $BF = h_{\text{left}} - h_{\text{right}}$。如果任何节点的平衡因子超出这个范围，树就被认为是不平衡的，需要通过旋转操作来调整。

计算节点高度是 AVL 树操作的基础。通常，我们约定空节点的高度为 -1（这便于计算），而非空节点的高度为其左右子树高度的最大值加一。例如，一个叶子节点的高度为 0，因为它没有子节点。在代码中，我们会频繁调用一个辅助函数来获取或更新节点的高度，以确保平衡因子的正确计算。

14　维持平衡的魔法：AVL 旋转

当插入或删除节点导致平衡因子超出允许范围时，AVL 树通过旋转操作来恢复平衡。旋转的目的是在不破坏 BST 排序性质的前提下，对局部子树进行重组，以降低高度。旋转操作分为四种基本类型，对应不同的不平衡情况。

第一种情况是左左（LL）情况。这发生在节点的平衡因子大于 1（即左子树比右子树高太多），且其左孩子的平衡因子大于或等于 0（表示不平衡源于左孩子的左子树）。解决方案是执行右旋操作：以该节点为轴，将其左孩子提升为新的根节点，原节点成为新根节点的右孩子，同时调整子树指针以维持 BST 性质。

第二种情况是右右（RR）情况。这发生在节点的平衡因子小于 -1（即右子树比左子树高太多），且其右孩子的平衡因子小于或等于 0（表示不平衡源于右孩子的右子树）。解决方案是执行左旋操作：以该节点为轴，将其右孩子提升为新的根节点，原节点成为新根节点的左孩子，并调整子树指针。

第三种情况是左右（LR）情况。这发生在节点的平衡因子大于 1，但其左孩子的平衡因子

小于 0（表示不平衡源于左孩子的右子树）。解决方案需要两步：先对左孩子执行左旋操作（将其转化为 LL 情况），再对原节点执行右旋操作。

第四种情况是右左（RL）情况。这发生在节点的平衡因子小于 -1，但其右孩子的平衡因子大于 0（表示不平衡源于右孩子的左子树）。解决方案同样需要两步：先对右孩子执行右旋操作（将其转化为 RR 情况），再对原节点执行左旋操作。

理解这些旋转的关键在于可视化子树的重组过程，但本文避免使用图片，因此建议读者在脑海中模拟指针的调整。旋转操作完成后，树的高度会减少，平衡因子恢复正常，从而确保整体效率。

15 代码实现：手把手构建 AVL 树

我们将使用 C++ 语言实现 AVL 树。代码实现分为多个步骤，从定义节点类到实现核心操作函数。每个代码块后会有详细解读，以帮助理解。

首先，定义树节点类。节点包含整数值 `val`、左右子节点指针 `left` 和 `right`，以及高度 `height`。我们选择存储高度而非平衡因子，因为平衡因子可以通过高度计算得出。

```
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    int height;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr), height(0)
        ↪ {}
};
```

这段代码定义了一个简单的节点类，构造函数初始化值、指针和高度。高度初始化为 0，但后续会通过更新函数调整。

接下来，实现辅助函数。`getHeight` 函数用于获取节点高度，处理空节点情况；`updateHeight` 函数更新节点高度基于其子节点高度；`getBalanceFactor` 函数计算平衡因子。

```
int getHeight(TreeNode* node) {
    if (node == nullptr) return -1;
    return node->height;
}

void updateHeight(TreeNode* node) {
    if (node == nullptr) return;
    node->height = 1 + std::max(getHeight(node->left), getHeight(node
        ↪ ->right));
}

int getBalanceFactor(TreeNode* node) {
```

```
    if (node == nullptr) return 0;
    return getHeight(node->left) - getHeight(node->right);
}
```

getHeight 函数检查节点是否为空，如果是则返回 -1，否则返回存储的高度。updateHeight 函数重新计算节点高度为左右子树高度的最大值加一。getBalanceFactor 函数返回左子树高度减右子树高度，直接使用高度值计算。

现在，实现旋转函数。左旋和右旋是基本操作，它们返回调整后的新根节点。

```
TreeNode* rotateLeft(TreeNode* y) {
    TreeNode* x = y->right;
    TreeNode* T2 = x->left;
    x->left = y;
    y->right = T2;
    updateHeight(y);
    updateHeight(x);
    return x;
}

TreeNode* rotateRight(TreeNode* x) {
    TreeNode* y = x->left;
    TreeNode* T2 = y->right;
    y->right = x;
    x->left = T2;
    updateHeight(x);
    updateHeight(y);
    return y;
}
```

在 rotateLeft 函数中，y 是原根节点，x 是其右孩子。操作将 x 的左子树（T2）attached 到 y 的右边，然后更新 y 和 x 的高度。右旋类似，但方向相反。旋转后，BST 性质保持不变，因为值的相对顺序没有改变。

基于旋转函数，实现平衡函数 balance。它检查节点的平衡因子，并根据四种情况调用相应的旋转。

```
TreeNode* balance(TreeNode* node) {
    if (node == nullptr) return nullptr;
    int bf = getBalanceFactor(node);
    if (bf > 1) {
        if (getBalanceFactor(node->left) >= 0) {
            return rotateRight(node); // LL case
        } else {
            node->left = rotateLeft(node->left); // LR case: first left
                ↪ rotate left child
```

```
        return rotateRight(node);
      }
   }
   if (bf < -1) {
      if (getBalanceFactor(node->right) <= 0) {
         return rotateLeft(node); // RR case
      } else {
         node->right = rotateRight(node->right); // RL case: first
             ↪ right rotate right child
         return rotateLeft(node);
      }
   }
   return node; // no need to balance
}
```

这个函数首先计算平衡因子。如果平衡因子大于 1，检查左孩子的平衡因子以区分 LL 或 LR 情况，并执行相应旋转。类似地处理平衡因子小于 -1 的情况。旋转后返回新根节点。

实现插入操作。插入是递归的，先执行标准 BST 插入，然后更新高度并平衡节点。

```
TreeNode* insert(TreeNode* node, int key) {
   if (node == nullptr) {
      return new TreeNode(key);
   }
   if (key < node->val) {
      node->left = insert(node->left, key);
   } else if (key > node->val) {
      node->right = insert(node->right, key);
   } else {
      return node; // duplicate keys not allowed
   }
   updateHeight(node);
   return balance(node);
}
```

插入函数递归地找到合适位置插入新节点。插入后，更新当前节点高度并调用 balance 来恢复平衡。返回调整后的根节点。

实现删除操作。删除同样递归，处理三种情况：无子节点、有一个子节点、有两个子节点。删除后更新高度并平衡。

```
TreeNode* remove(TreeNode* node, int key) {
   if (node == nullptr) return nullptr;
   if (key < node->val) {
      node->left = remove(node->left, key);
   } else if (key > node->val) {
```

```
        node->right = remove(node->right, key);
    } else {
        if (node->left == nullptr || node->right == nullptr) {
            TreeNode* temp = node->left ? node->left : node->right;
            delete node;
            return temp;
        } else {
            TreeNode* temp = node->right;
            while (temp->left != nullptr) {
                temp = temp->left;
            }
            node->val = temp->val;
            node->right = remove(node->right, temp->val);
        }
    }
    updateHeight(node);
    return balance(node);
}
```

删除函数首先找到要删除的节点。如果节点有一个或无子节点，直接删除并返回子节点。如果有两个子节点，找到中序遍历后继（右子树的最小值），复制值到当前节点，并递归删除后继节点。最后更新高度并平衡。

查找操作与普通 BST 相同，无需修改。

```
bool search(TreeNode* node, int key) {
    if (node == nullptr) return false;
    if (key == node->val) return true;
    if (key < node->val) return search(node->left, key);
    return search(node->right, key);
}
```

查找函数递归搜索值，返回是否存在。

16　测试与验证

为了验证 AVL 树的正确性，我们编写一个简单的测试程序。测试数据使用有序序列 [10, 20, 30, 40, 50, 25] 来演示 AVL 树如何避免退化。

```
#include <iostream>
#include <algorithm>
void inOrder(TreeNode* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    std::cout << node->val << " ";
```

```
    inOrder(node->right);
}
int main() {
    TreeNode* root = nullptr;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
    std::cout << "In-order traversal: ";
    inOrder(root); // should output sorted values
    std::cout << "\nHeight of tree: " << getHeight(root) << std::endl;
        ↪ // should be small
    // Test deletion if implemented
    root = remove(root, 30);
    std::cout << "After deletion, in-order: ";
    inOrder(root);
    std::cout << std::endl;
    return 0;
}
```

中序遍历应输出有序序列（10, 20, 25, 30, 40, 50），证明 BST 性质维持。树高度应远小于节点数，表示平衡。删除操作后，树应保持平衡和有序。

AVL 树通过严格的平衡条件确保了操作的时间复杂度为 $O(\log n)$，非常适合读多写少的场景。然而，其缺点在于插入和删除时需要频繁旋转，可能导致性能开销。相比之下，红黑树采用近似平衡，旋转次数较少，广泛应用于标准库如 C++ STL 的 map 和 set。

对于进一步学习，建议探索红黑树、B 树和 Splay 树等其他平衡数据结构。这些结构在不同场景下各有优势，例如 B 树用于数据库和文件系统。

17 附录 & 互动

完整代码可参考 GitHub 仓库示例。欢迎在评论区提问或分享您的实现经验。参考文献包括经典算法书籍和在线资源，如 Introduction to Algorithms by Cormen et al.。

第 V 部

GPU 编程语言的设计原理与实现

叶家炜

Sep 17, 202

随着人工智能和高性能计算的飞速发展，图形处理单元（GPU）已成为现代计算的核心算力来源。然而，GPU 的架构与传统中央处理单元（CPU）存在本质差异，这导致了专门编程语言的需求。CPU 设计侧重于低延迟处理，拥有少量复杂核心和精细的内存缓存层次结构，而 GPU 则专注于高吞吐量并行计算，采用大量简单核心和高带宽显存体系。这种架构鸿沟意味着传统 CPU 语言如 C++ 无法高效直接映射到 GPU 上，因为它们缺乏对大规模数据并行的原生支持。

并行范式的转变进一步凸显了专用语言的重要性。从任务并行转向数据并行，要求开发者管理成千上万个线程，这需要新的抽象来简化编程。GPU 编程语言如 CUDA、OpenCL 和 HIP 应运而生，它们通过提供线程层次结构和内存模型抽象，弥合了硬件与软件之间的 gap。本文将深入探讨这些语言的设计原理和实现机制，帮助读者理解如何解锁 GPU 的强大算力。

18 核心设计原理：抽象与权衡

GPU 编程语言的设计核心是在抽象程度和硬件控制力之间进行权衡。这种权衡体现在并行模型、内存模型和执行调度模型中。首先，并行模型抽象采用 SPMD（单程序多数据）模式，其中所有线程执行相同代码但处理不同数据。这种模型通过分层线程层次结构实现，包括线程、线程块和网格。例如，在 CUDA 中，线程块内的线程可以协作并通过共享内存通信，而网格则提供了可扩展性，使代码能够适应不同规模的 GPU 硬件。

SIMT（单指令多线程）执行模型是另一个关键抽象，它与 SIMD（单指令多数据）类似但更具灵活性。在 SIMT 中，线程以 warp 或 wavefront 为单位执行，允许线程在分支处产生分歧，但会带来性能开销。例如，如果线程在一个 warp 内执行不同分支，硬件会串行处理这些分支，导致效率降低。这要求程序员在编写代码时尽量减少分支分歧，以优化性能。

内存模型抽象则通过分层内存空间来实现，包括全局内存、共享内存、本地内存、常量内存和纹理内存。每种内存具有不同的作用域、生命周期和性能特征。共享内存尤其重要，它为线程块内的线程提供低延迟共享存储，但需要显式同步，如使用 `__syncthreads()` 函数来确保数据一致性。GPU 通常采用弱一致性或释放一致性模型，这意味着内存操作顺序可能不严格，程序员必须通过内存栅栏（memory fence）等机制来控制可见性。

执行与调度模型侧重于大规模线程的隐式管理。程序员定义逻辑线程网格，而运行时系统和硬件调度器负责将线程映射到物理计算单元。延迟隐藏是关键优化技术，通过快速切换正在执行的 warp 来掩盖内存访问和算术操作的延迟，这依赖于高吞吐量而非大缓存。这种设计使得 GPU 能够高效处理数据并行任务，但要求程序员理解底层硬件行为以避免性能瓶颈。

19 语言实现的关键技术剖析

GPU 编程语言的实现涉及编译器、运行时系统和驱动程序等多个层面。编译器前端通常基于现有主机语言如 C++ 进行扩展，通过添加关键字和语法来区分设备代码。例如，在 CUDA 中，使用 `__global__` 修饰符标识内核函数，`__device__` 用于设备函数。编译器进行语法和语义分析时，会验证代码合法性，如禁止在设备代码中使用主机端特性如动态内存分配或异常处理。

编译器中端和后端负责将高级代码转换为中间语言和目标代码。中间语言如 NVIDIA 的 PTX 或 AMD 的 GCN IL 充当“并行汇编”，提供硬件无关的抽象。PTX 代码随后通过即时编译（JIT）或提前编译（AOT）转换为具体 GPU 的机器指令（如 SASS）。关键优化包括寄存器

分配，它直接影响活跃 warp 数量和性能；循环展开和分支预测优化，以处理 GPU 上的分支问题；以及计算与数据访问的重排，以优化指令流水线。例如，一个简单的 SAXPY 内核代码在 CUDA 中可能如下所示：

```
__global__ void saxpy(float *x, float *y, float a, int n) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < n) {
      y[i] = a * x[i] + y[i];
   }
}
```

这段代码定义了并行向量运算，其中每个线程计算一个元素。`blockIdx.x` 和 `threadIdx.x` 用于计算线程索引，确保数据并行性。编译器会优化寄存器使用，以减少内存访问延迟。
运行时系统管理设备初始化、内存传输和内核启动。例如，`cudaMalloc` 用于设备内存分配，`cudaMemcpy` 用于主机与设备间数据传输。内核启动是异步操作，允许重叠计算和数据传输以提高效率。驱动程序作为桥梁，将编译好的指令发送给 GPU 调度器，完成最终执行。

20 案例研究：主流语言的实现对比

NVIDIA CUDA 的设计哲学以性能优先为核心，与硬件深度集成，提供丰富的生态系统。其实现依赖于闭源编译器 NVCC 和专有驱动程序，支持完善的性能分析工具如 Nsight。CUDA 通过直接暴露硬件特性，允许专家级调优，但牺牲了部分可移植性。
OpenCL 则强调开放标准和跨平台支持，目标是兼容不同厂商的 CPU、GPU 和 FPGA 等设备。实现上，各个厂商提供自己的编译器，将 OpenCL C 代码编译为 SPIR-V 中间表示，再转换为本地代码。这种设计虽然提高了可移植性，但可能导致性能优化上的妥协，因为需要处理硬件差异。
AMD HIP 和 ROCm 平台采用开源哲学，旨在提供类似 CUDA 的开发体验，同时支持跨平台编译。HIP 代码可以通过 `hipcc` 编译器编译为 AMD 的 GCN 代码或通过薄层移植到 CUDA 代码，实现“编写一次，多处编译”的目标。这种实现平衡了性能和可移植性，但仍在生态系统中追赶 CUDA。

21 高级抽象与未来趋势

随着技术发展，更高层次的抽象成为趋势，通过库和框架如 Thrust、CUB、Jax 和 Triton 来提升开发效率。这些工具隐藏底层细节，允许开发者专注于算法而非硬件优化。例如，Jax 提供自动微分和 JIT 编译，简化了机器学习工作流。
异构计算推动 SYCL 和 oneAPI 等模型的发展，试图通过单源 C++ 方式统一 CPU 和加速器编程。SYCL 基于标准 C++，提供跨平台支持，减少代码重复。未来，机器学习编译器基础设施如 MLIR 将 enable 更灵活的优化和重定向，支持多种硬件后端。
特定领域语言（DSL）为图形、AI 和科学计算等领域定制，以获取极致性能和易用性。这些趋势表明，GPU 编程语言将继续演化，更好地驾驭新兴硬件如 DPU 和新型内存。

22 结论

GPU 编程语言的成功源于其精准的抽象设计，既暴露足够硬件细节供专家调优，又隐藏复杂性以降低入门门槛。核心思想是在性能、可移植性和开发效率之间进行永恒权衡。随着硬件架构持续演进，未来语言和模型将不断创新，以释放更强大的算力潜能。展望未来，开发者应关注抽象层次的提升和跨平台解决方案，以应对日益复杂的计算需求。